架构师面试题 - 设计模式专题

设计模式（DesignPattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

本篇为设计模式面试专题，总共收录了 35 道常见面试题及答案解析，希望能帮到你~

目录

1、什么是设计模式?

就是经过实践验证的用来解决特定环境下特定问题的解决方案

2、设计模式用来干什么?

寻找合适的对象

决定对象的粒度

指定对象的接口

描述对象的实现

运用复用机制

重复使用经过实践验证的正确的，用来解决某一类问题的解决方案来达到减少工作量、提高正确率等目的

3、什么是对象粒度?

对象中方法的多少就是粒度

4、基本的 Java 编程设计应遵循的规则?

面向接口编程，优先使用对象组合

5、设计模式的应用范围

所能解决的特定的一类问题中

6、简述什么是单例模式，以及他解决的问题，应用的环境，解决的方案，模式的本质

在任何时间内只有一个类实例存在的模式，需要有一个从中进行全局访问和维护某种类型数据的区域的环境下使用单例模式，解决方案就是保证一个类只有一个类实例存在，本质就是实例共用同一块内存区域

7、简述什么是工厂模式，以及他解决的问题，应用的环境，解决的方案，模式的本质

利用工厂来解决接口选择的问题的模式

应用环境：当一个类无法预料要创建哪种类的对象或是一个类需要由子类来指定创建的对象时，就需要用到工厂模式

解决方案：定义一个创建对象的接口,让子类来决定具体实例化哪一个类

本质就是根据不同的情况来选择不同的接口

8、述什么是值对象模式，以及他解决的问题，应用的环境，解决的方案，模式的本质

用来把一组数据封装成一个对象的模式

解决问题：在远程方法的调用次数增加的时候，相关的应用程序性能将会有很大的下降

解决方案：使用值对象的时候，可以通过仅仅一次方法调用来取得整个对象，而不是使用多次方法调用以得到对象中每个域的数值

本质：就是把需要传递的多个值封装成一个对象一次性传过去

9、代码示例：值对象模式的实现方法

```
public class UserModel{
    private String userId;
```

```
    private String userName;
    public void setUserId(String id){
        this.userId = id;
    }
    public String getUserId(){
        return userId;
    }

    public void setUserName(String name){
        this.userName = name;
    }
    public String getUserName(){
        return userName;
    }
}
```

10、简述什么是 DAO 模式，以及他解决的问题，应用的环境，解决的方案，模式的本质

数据访问对象

解决问题：根据数据源不同，数据访问也不同。根据存储的类型（关系数据库、

面向对象数据库、纯文件等）和供应商实现不同，持久性存储（如数据库）的访问差

别也很大。如何对存储层以外的模块屏蔽这些复杂性，以提供统一的调用存储实现。

程序的分布式问题

解决方案：将数据访问逻辑抽象为特殊的资源，也就是说将系统资源的接口从

其底层访问机制中隔离出来；通过将数据访问的调用打包，数据访问对象可以

促进对于不同数据库类型和模式的数据访问。

DAO 的本质就是一层屏蔽一种变化

本质：分层，是系统组件和数据源中间的适配器。（一层屏蔽一种变化）

11、代码示例：DAO 模式的实现方法

```
public interface CustomerDAO {
public int insertCustomer(...);
```

```
public Collection selectCustomersVO(...);
}
```

12、什么是开放-封闭法则(OCP)

可扩展但是不可以更改已有的模块

对扩展是开放的 对修改是封闭

13、什么是替换法则(LSP)

使用指向基类（超类）的引用的函数，必须能够在不知道具体派生类（子类）对象类型的情况下使用

14、如何综合使用我们学过的设计模式来构建合理的应用程序结构

是采用接口进行隔离，然后同时暴露值对象和工厂类，如果是需要数据存储的功能，又会通过 DAO 模式去与数据存储层交互。

15、请列举出在 JDK 中几个常用的设计模式?

单例模式（Singleton pattern）用于 Runtime，Calendar 和其他的一些类中；

工厂模式（Factory pattern）被用于各种不可变的类如 Boolean，像 Boolean.valueOf；

观察者模式（Observer pattern）被用于 Swing 和很多的事件监听中；

装饰器设计模式（Decorator design pattern）被用于多个 Java IO 类中。

16、什么是设计模式？你是否在你的代码里面使用过任何设计模式?

设计模式是世界上各种各样程序员用来解决特定设计问题的尝试和测试的方法。设计模式是代码可用性的延伸。

17、Java 中什么叫单例设计模式？请用 Java 写出线程安全的单例模式

单例模式重点在于在整个系统上共享一些创建时较耗资源的对象。整个应用中只维护一个特定类实例，它被所有组件共同使用。Java.lang.Runtime 是单例模式的经典例子。从 Java 5 开始你可以使用枚举（enum）来实现线程安全的单例。

18、在 Java 中，什么叫观察者设计模式（observer design pattern）？

观察者模式是基于对象的状态变化和观察者的通讯，以便他们作出相应的操作。简单的例子就是一个天气系统，当天气变化时必须在展示给公众的视图中进行反映。这个视图对象是一个主体，而不同的视图是观察者。

19、使用工厂模式最主要的好处是什么？在哪里使用?

工厂模式的最大好处是增加了创建对象时的封装层次。如果你使用工厂来创建对象，之后你可以使用更高级和更高性能的实现来替换原始的产品实现或类，这不需要在调用层做任何修改。

20、举一个用 Java 实现的装饰模式(decorator design pattern)？它是作用于对象层次还是类层次?

装饰模式增加强了单个对象的能力。Java IO 到处都使用了装饰模式，典型例子就是 Buffered 系列类如 BufferedReader 和 BufferedWriter，它们增强了 Reader 和 Writer 对象，以实现提升性能的 Buffer 层次的读取和写入。

21、在 Java 中，为什么不允许从静态方法中访问非静态变量?

Java 中不能从静态上下文访问非静态数据只是因为非静态变量是跟具体的对象实例关联的，而静态的却没有和任何实例关联。

22、设计一个 ATM 机，请说出你的设计思路?

比如设计金融系统来说，必须知道它们应该在任何情况下都能够正常工作。不管是断电还是其他情况，ATM 应该保持正确的状态（事务），想想 加锁（locking）、事务（transaction）、错误条件（error condition）、边界条件（boundary condition） 等等。尽管你不能想到具体的设计，但如果你可以指出非功能性需求，提出一些问题，想到关于边界条件，这些都会是很好的。

23、在 Java 中，什么时候用重载，什么时候用重写?

如果你看到一个类的不同实现有着不同的方式来做同一件事，那么就应该用重写（overriding），而重载（overloading）是用不同的输入做同一件事。在 Java 中，重载的方法签名不同，而重写并不是。

24、举例说明什么情况下会更倾向于使用抽象类而不是接口?

接口和抽象类都遵循”面向接口而不是实现编码”设计原则，它可以增加代码的灵活性，可以适应不断变化的需求。下面有几个点可以帮助你回答这个问题：

在 Java 中，你只能继承一个类，但可以实现多个接口。所以一旦你继承了一个类，你就失去了继承其他类的机会了。

接口通常被用来表示附属描述或行为如：Runnable、Clonable、Serializable 等等，因此

当你使用抽象类来表示行为时，你的类就不能同时是 Runnable 和 Clonable(注：这里的意思是指如果把 Runnable 等实现为抽象类的情况)，因为在 Java 中你不能继承两个类，但当你使用接口时，你的类就可以同时拥有多个不同的行为。

在一些对时间要求比较高的应用中，倾向于使用抽象类，它会比接口稍快一点。

如果希望把一系列行为都规范在类继承层次内，并且可以更好地在同一个地方进行编码，那么抽象类是一个更好的选择。有时，接口和抽象类可以一起使用，接口中定义函数，而在抽象类中定义默认的实现。

25、你所知道的设计模式有哪些?

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式已经单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

λ创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

λ结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

λ行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

26、单例设计模式?

最好理解的一种设计模式，分为懒汉式和饿汉式。

饿汉式：

```
public class Singleton {

    // 直接创建对象
    public static Singleton instance = new Singleton();

    // 私有化构造函数
    private Singleton() {
    }

    // 返回对象实例
    public static Singleton getInstance() {
        return instance;

    }
}
```

懒汉式：

```
public class Singleton {
    // 声明变量
    private static volatile Singleton singleton = null;
    // 私有构造函数
    private Singleton() {
    }
    // 提供对外方法
    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

27、工厂设计模式?

工厂模式分为工厂方法模式和抽象工厂模式。

工厂方法模式分为三种：

λ普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象。

λ多个工厂方法模式，是提供多个工厂方法，分别创建对象。

λ静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

ν普通工厂模式

```
public interface Sender {
    public void Send();
}
public class MailSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is mail sender!");
    }
}
public class SmsSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
public class SendFactory {
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
```

```
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}
```

v多个工厂方法模式

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```
public class SendFactory {
    public Sender produceMail() {
        return new MailSender();
    }
    public Sender produceSms() {
        return new SmsSender();
    }
}
public class FactoryTest {
    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.send();
    }
}
```

v静态工厂方法模式

将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
public class SendFactory {
    public static Sender produceMail() {
        return new MailSender();
    }
    public static Sender produceSms() {
        return new SmsSender();
    }
}
public class FactoryTest {
```

```
    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.send();
    }
}
```

v抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

```
public interface Provider {
    public Sender produce();
}
public interface Sender {
    public void send();
}
public class MailSender implements Sender {
    @Override
    public void send() {
        System.out.println("this is mail sender!");
    }
}
public class SmsSender implements Sender {
    @Override
    public void send() {
        System.out.println("this is sms sender!");
    }
}
public class SendSmsFactory implements Provider {
    @Override
    public Sender produce() {
        return new SmsSender();
    }
}
public class SendMailFactory implements Provider {
    @Override
    public Sender produce() {
        return new MailSender();
```

```
    }
}
public class Test {
    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.send();
    }
}
```

28、建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象， 所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 Test 结合起来得到的。

```
public class Builder {
    private List<Sender> list = new ArrayList<Sender>();
    public void produceMailSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new MailSender());
        }
    }
    public void produceSmsSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new SmsSender());
        }
    }
}
public class Builder {
    private List<Sender> list = new ArrayList<Sender>();
    public void produceMailSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new MailSender());
        }
    }
    public void produceSmsSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new SmsSender());
        }
    }
```

```
}
public class TestBuilder {
    public static void main(String[] args) {
        Builder builder = new Builder();
        builder.produceMailSender(10);
    }
}
```

29、适配器设计模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

v类的适配器模式

```
public class Source {
    public void method1() {
        System.out.println("this is original method!");
    }
}
public interface Targetable {
    /* 与原类中的方法相同 */
    public void method1();
    /* 新类的方法 */
    public void method2();
}
public class Adapter extends Source implements Targetable {
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}
public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
```

v对象的适配器模式

基本思路和类的适配器模式相同，只是将 Adapter 类作修改，这次不继承 Source 类，而是持有 Source 类的实例，以达到解决兼容性的问题。

```
public class Wrapper implements Targetable {
    private Source source;
    public Wrapper(Source source) {
        super();
        this.source = source;
    }
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
    @Override
    public void method1() {
        source.method1();
    }
}
public class AdapterTest {
    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

v接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

30、装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

```
public interface Sourceable {
    public void method();
}
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
public class Decorator implements Sourceable {
    private Sourceable source;
    public Decorator(Sourceable source) {
        super();
        this.source = source;
    }
    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}

public class DecoratorTest {
    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}
```

31、策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

```
public interface ICalculator {
    public int calculate(String exp);
}
public class Minus extends AbstractCalculator implements ICalculator {
    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "-");
        return arrayInt[0] - arrayInt[1];
    }
}
public class Plus extends AbstractCalculator implements ICalculator {
    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "\+");
        return arrayInt[0] + arrayInt[1];
    }
}
public class AbstractCalculator {
    public int[] split(String exp, String opt) {
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;
    }
}
public class StrategyTest {
    public static void main(String[] args) {
        String exp = "2+8";
```

```
        ICalculator cal = new Plus();
        int result = cal.calculate(exp);
        System.out.println(result);
    }
}
```

32、观察者模式（Observer）

观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

```
public interface Observer {
    public void update();
}
public class Observer1 implements Observer {
    @Override
    public void update() {
        System.out.println("observer1 has received!");
    }
}
public class Observer2 implements Observer {
    @Override
    public void update() {
        System.out.println("observer2 has received!");
    }
}
public interface Subject {
    /*增加观察者*/
    public void add(Observer observer);
    /*删除观察者*/
    public void del(Observer observer);
    /*通知所有的观察者*/
    public void notifyObservers();
    /*自身的操作*/
    public void operation();
}
public abstract class AbstractSubject implements Subject {
    private Vector<Observer> vector = new Vector<Observer>();
```

```
    @Override
    public void add(Observer observer) {
        vector.add(observer);
    }
    @Override
    public void del(Observer observer) {
        vector.remove(observer);
    }
    @Override
    public void notifyObservers() {
        Enumeration<Observer> enumo = vector.elements();
        while (enumo.hasMoreElements()) {
            enumo.nextElement().update();
        }
    }
}
public class MySubject extends AbstractSubject {
    @Override
    public void operation() {
        System.out.println("update self!");
        notifyObservers();
    }
}
public class ObserverTest {
    public static void main(String[] args) {
        Subject sub = new MySubject();
        sub.add(new Observer1());
        sub.add(new Observer2());
        sub.operation();
    }
}
```

33、微服务架构的六种常用设计模式是什么?

代理设计模式

聚合设计模式

链条设计模式

聚合链条设计模式

数据共享设计模式

异步消息设计模式

34、设计模式是什么，设计模式有什么作用?

设计模式是一套被反复使用的、多数人知晓、经过分类编目的优秀代码设计经验的总结。特定环境下特定问题的处理方法。

1）重用设计和代码 重用设计比重用代码更有意义，自动带来代码重用

2）提高扩展性 大量使用面向接口编程，预留扩展插槽，新的功能或特性很容易加入到系统中来

3）提高灵活性 通过组合提高灵活性，可允许代码修改平稳发生，对一处修改不会波及到其他模块

4） 提高开发效率 正确使用设计模式，可以节省大量的时间

35、Spring 框架中都用到了哪些设计模式?

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的：

代理模式—在 AOP 和 remoting 中被用的比较多。

单例模式—在 spring 配置文件中定义的 bean 默认为单例模式。

模板方法—用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

前端控制器—Spring 提供了 DispatcherServlet 来对请求进行分发。

视图帮助(View Helper)—Spring 提供了一系列的 JSP 标签，高效宏来辅助将分散的代码整合在视图里。

依赖注入—贯穿于 BeanFactory / ApplicationContext 接口的核心理念。

工厂模式—BeanFactory 用来创建对象的实例