洲洋1984

< 2021年10月 >

日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

昵称：洲洋1984
园龄：4年
粉丝：4
关注：0
+加关注

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

随笔档案
2020年2月(27)
2020年1月(6)
2018年8月(1)

阅读排行榜
1. Java 分布式框架面试题合集(12690)
2. Java 中的各种锁和 CAS + 面试题(3280)
3. JVM 面试题汇总(1593)
4. MyBatis 介绍(1474)
5. 对数据库的基本操作步骤(1137)

推荐排行榜
1. Java 中的各种锁和 CAS + 面试题(1)

最新评论
1. Re:设计模式常见面试题汇总
懒汉式私有化构造才行
--小妖来巡山
2. Re:设计模式常见面试题汇总
打扰了,眼花了
--小妖来巡山
3. Re:设计模式常见面试题汇总
恶汉单例有问题,构造方法需要私有化
--小妖来巡山

设计模式常见面试题汇总

1.说一下设计模式？你都知道哪些？

答：设计模式总共有 23 种，总体来说可以分为三大类：创建型模式（ Creational Patterns ）、结构型模式（ Structural Patterns ）和行为型模式（ Behavioral Patterns ）。

分类	**包含**	**关注点**
创建型模式	工厂模式、抽象工厂模式、单例模式、建造者模式、原型模式	关注于对象的创建，

下面会对常用的设计模式分别做详细的说明。

2.什么是单例模式？

答：单例模式是一种常用的软件设计模式，在应用这个模式时，单例对象的类必须保证只有一个实例存在，整个系统只能使用一个对象实例。

优点：不会频繁地创建和销毁对象，浪费系统资源。

使用场景：IO 、数据库连接、Redis 连接等。

单例模式代码实现：

```
class Singleton {
    private static Singleton instance = new Singleton();
    public static Singleton getInstance() {
        return instance;
    }
}
```

单例模式调用代码：

```
public class Lesson7\_3 {
    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();
        System.out.println(singleton1 == singleton2);
    }
}
```

程序的输出结果：true

可以看出以上单例模式是在类加载的时候就创建了，这样会影响程序的启动速度，那如何实现单例模式的延迟加载？在使用时再创建？

单例延迟加载代码：

```
// 单例模式-延迟加载版
class SingletonLazy {
    private static SingletonLazy instance;
    public static SingletonLazy getInstance() {
        if (instance == null) {
            instance = new SingletonLazy();
        }
        return instance;
    }
}
```

以上为非线程安全的，单例模式如何支持多线程？

使用 synchronized 来保证，单例模式的线程安全代码：

```
class SingletonLazy {
    private static SingletonLazy instance;
    public static synchronized SingletonLazy getInstance() {
        if (instance == null) {
            instance = new SingletonLazy();
        }
        return instance;
    }
```

```
}
```

3.什么是简单工厂模式?

答：简单工厂模式又叫静态工厂方法模式，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。比如，一台咖啡机就可以理解为一个工厂模式，你只需要按下想喝的咖啡品类的按钮（摩卡或拿铁），它就会给你生产一杯相应的咖啡，你不需要管它内部的具体实现，只要告诉它你的需求即可。

优点：

- 工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消费”产品；简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象；
- 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以减少使用者的记忆量；
- 通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。

缺点：

- 不易拓展，一旦添加新的产品类型，就不得不修改工厂的创建逻辑；
- 产品类型较多时，工厂的创建逻辑可能过于复杂，一旦出错可能造成所有产品的创建失败，不利于系统的维护。

简单工厂示意图如下：

简单工厂**代码实现**：

```
class Factory {
    public static String createProduct(String product) {
        String result = null;
        switch (product) {
            case "Mocca":
                result = "摩卡";
                break;
            case "Latte":
                result = "拿铁";
                break;
            default:
                result = "其他";
                break;
        }
        return result;
    }
}
```

4.什么是抽象工厂模式?

答：抽象工厂模式是在简单工厂的基础上将未来可能需要修改的代码抽象出来，通过继承的方式让子类去做决定。

比如，以上面的咖啡工厂为例，某天我的口味突然变了，不想喝咖啡了想喝啤酒，这个时候如果直接修改简单工厂里面的代码，这种做法不但不够优雅，也不符合软件设计的“开闭原则”，因为每次新增品类都要修改原来的代码。这个时候就可以使用抽象工厂类了，抽象工厂里只声明方法，具体的实现交给子类（子工厂）去实现，这个时候再有新增品类的需求，只需要新创建代码即可。

抽象工厂实现代码如下：

```
public class AbstractFactoryTest {
    public static void main(String[] args) {
        // 抽象工厂
        String result = (new CoffeeFactory()).createProduct("Latte");
        System.out.println(result); // output:拿铁
    }
}
// 抽象工厂
abstract class AbstractFactory{
    public abstract String createProduct(String product);
}
// 啤酒工厂
class BeerFactory extends AbstractFactory{
    @Override
    public String createProduct(String product) {
        String result = null;
        switch (product) {
            case "Hans":
                result = "汉斯";
                break;
            case "Yanjing":
                result = "燕京";
                break;
```

```
            default:
                result = "其他啤酒";
                break;
        }
        return result;
    }
}
/\* \* 咖啡工厂 \*/
class CoffeeFactory extends AbstractFactory{
    @Override
    public String createProduct(String product) {
        String result = null;
        switch (product) {
            case "Mocca":
                result = "摩卡";
                break;
            case "Latte":
                result = "拿铁";
                break;
            default:
                result = "其他咖啡";
                break;
        }
        return result;
    }
}
```

5.什么是观察者模式?

观察者模式是定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。 **优点**：

- 观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色；
- 观察者模式在观察目标和观察者之间建立一个抽象的耦合；
- 观察者模式支持广播通信；
- 观察者模式符合开闭原则（对拓展开放，对修改关闭）的要求。

缺点：

- 如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间；
- 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃；
- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

在观察者模式中有如下角色：

- Subject：抽象主题（抽象被观察者），抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象；
- ConcreteSubject：具体主题（具体被观察者），该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知；
- Observer：抽象观察者，是观察者者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己；
- ConcrereObserver：具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

观察者模式实现代码如下。

1）定义观察者（消息接收方）

```
/\* \* 观察者（消息接收方） \*/
interface Observer {
    public void update(String message);
}
/\* \* 具体的观察者（消息接收方） \*/
class ConcrereObserver implements Observer {
    private String name;

    public ConcrereObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + ": " + message);
    }
}
```

2）定义被观察者（消息发送方）

```
/\* \* 被观察者（消息发布方） \*/
interface Subject {
    // 增加订阅者
    public void attach(Observer observer);
    // 删除订阅者
    public void detach(Observer observer);
    // 通知订阅者更新消息
    public void notify(String message);
}
/\* \* 具体被观察者（消息发布方） \*/
class ConcreteSubject implements Subject {
    // 订阅者列表（存储信息）
    private List<Observer> list = new ArrayList<Observer>();
    @Override
    public void attach(Observer observer) {
        list.add(observer);
    }
    @Override
    public void detach(Observer observer) {
        list.remove(observer);
    }
    @Override
    public void notify(String message) {
        for (Observer observer : list) {
            observer.update(message);
        }
    }
}
```

3）代码调用

```
public class ObserverTest {
    public static void main(String[] args) {
        // 定义发布者
        ConcreteSubject concreteSubject = new ConcreteSubject();
        // 定义订阅者
        ConcrereObserver concrereObserver = new ConcrereObserver("老王");
        ConcrereObserver concrereObserver2 = new ConcrereObserver("Java");
        // 添加订阅
        concreteSubject.attach(concrereObserver);
        concreteSubject.attach(concrereObserver2);
        // 发布信息
        concreteSubject.notify("更新了");
    }
}
```

程序执行结果如下：

> 老王：更新了
>
> Java：更新了

6.什么是装饰器模式?

答：装饰器模式是指动态地给一个对象增加一些额外的功能，同时又不改变其结构。

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

装饰器模式的关键：装饰器中使用了被装饰的对象。

比如，创建一个对象“laowang”，给对象添加不同的装饰，穿上夹克、戴上帽子……，这个执行过程就是装饰者模式，实现代码如下。

1）定义顶层对象，定义行为

```
interface IPerson {
    void show();
}
```

2）定义装饰器超类

```
class DecoratorBase implements IPerson{
    IPerson iPerson;
    public DecoratorBase(IPerson iPerson){
        this.iPerson = iPerson;
    }
    @Override
```

```
    public void show() {
        iPerson.show();
    }
}
```

3）定义具体装饰器

```
class Jacket extends DecoratorBase {
    public Jacket(IPerson iPerson) {
        super(iPerson);
    }
    @Override
    public void show() {
        // 执行已有功能
        iPerson.show();
        // 定义新行为
        System.out.println("穿上夹克");
    }
}
class Hat extends DecoratorBase {
    public Hat(IPerson iPerson) {
        super(iPerson);
    }
    @Override
    public void show() {
        // 执行已有功能
        iPerson.show();
        // 定义新行为
        System.out.println("戴上帽子");
    }
}
```

4）定义具体对象

```
class LaoWang implements IPerson{
    @Override
    public void show() {
        System.out.println("什么都没穿");
    }
}
```

5）装饰器模式调用

```
public class DecoratorTest {
    public static void main(String[] args) {
        LaoWang laoWang = new LaoWang();
        Jacket jacket = new Jacket(laoWang);
        Hat hat = new Hat(jacket);
        hat.show();
    }
}
```

7.什么是模板方法模式?

答：模板方法模式是指定义一个模板结构，将具体内容延迟到子类去实现。

优点：

- 提高代码复用性：将相同部分的代码放在抽象的父类中，而将不同的代码放入不同的子类中；
- 实现了反向控制：通过一个父类调用其子类的操作，通过对子类的具体实现扩展不同的行为，实现了反向控制并且符合开闭原则。

以给冰箱中放水果为例，比如，我要放一个香蕉：开冰箱门 → 放香蕉 → 关冰箱门；如果我再要放一个苹果：开冰箱门 → 放苹果 → 关冰箱门。可以看出它们之间的行为模式都是一样的，只是存放的水果品类不同而已，这个时候就非常适用模板方法模式来解决这个问题，实现代码如下：

```
/\* \* 添加模板方法 \*/
abstract class Refrigerator {
    public void open() {
        System.out.println("开冰箱门");
    }
    public abstract void put();

    public void close() {
        System.out.println("关冰箱门");
    }
}
```

```
class Banana extends Refrigerator {
    @Override
    public void put() {
        System.out.println("放香蕉");
    }
}
class Apple extends Refrigerator {
    @Override
    public void put() {
        System.out.println("放苹果");
    }
}
/\* \* 调用模板方法 \*/
public class TemplateTest {
    public static void main(String[] args) {
        Refrigerator refrigerator = new Banana();
        refrigerator.open();
        refrigerator.put();
        refrigerator.close();
    }
}
```

程序执行结果：

> 开冰箱门
>
> 放香蕉
>
> 关冰箱门

8.什么是代理模式?

代理模式是给某一个对象提供一个代理，并由代理对象控制对原对象的引用。

优点：

- 代理模式能够协调调用者和被调用者，在一定程度上降低了系统的耦合度；
- 可以灵活地隐藏被代理对象的部分功能和服务，也增加额外的功能和服务。

缺点：

- 由于使用了代理模式，因此程序的性能没有直接调用性能高；
- 使用代理模式提高了代码的复杂度。

举一个生活中的例子：比如买飞机票，由于离飞机场太远，直接去飞机场买票不太现实，这个时候我们就可以上携程 App 上购买飞机票，这个时候携程 App 就相当于是飞机票的代理商。

代理模式实现代码如下：

```
/\* \* 定义售票接口 \*/
interface IAirTicket {
    void buy();
}
/\* \* 定义飞机场售票 \*/
class AirTicket implements IAirTicket {
    @Override
    public void buy() {
        System.out.println("买票");
    }
}
/\* \* 代理售票平台 \*/
class ProxyAirTicket implements IAirTicket {
    private AirTicket airTicket;
    public ProxyAirTicket() {
        airTicket = new AirTicket();
    }
    @Override
    public void buy() {
        airTicket.buy();
    }
}
/\* \* 代理模式调用 \*/
public class ProxyTest {
    public static void main(String[] args) {
        IAirTicket airTicket = new ProxyAirTicket();
        airTicket.buy();
    }
}
```

9.什么是策略模式?

答：策略模式是指定义一系列算法，将每个算法都封装起来，并且使他们之间可以相互替换。

优点：遵循了开闭原则，扩展性良好。

缺点：随着策略的增加，对外暴露越来越多。

以生活中的例子来说，比如我们要出去旅游，选择性很多，可以选择骑车、开车、坐飞机、坐火车等，就可以使用策略模式，把每种出行作为一种策略封装起来，后面增加了新的交通方式了，如超级高铁、火箭等，就可以不需要改动原有的类，新增交通方式即可，这样也符合软件开发的开闭原则。 策略模式实现代码如下：

```
/\* \* 声明旅行 \*/
interface ITrip {
    void going();
}
class Bike implements ITrip {
    @Override
    public void going() {
        System.out.println("骑自行车");
    }
}
class Drive implements ITrip {
    @Override
    public void going() {
        System.out.println("开车");
    }
}
/\* \* 定义出行类 \*/
class Trip {
    private ITrip trip;

    public Trip(ITrip trip) {
        this.trip = trip;
    }

    public void doTrip() {
        this.trip.going();
    }
}
/\* \* 执行方法 \*/
public class StrategyTest {
    public static void main(String[] args) {
        Trip trip = new Trip(new Bike());
        trip.doTrip();
    }
}
```

程序执行的结果：

> 骑自行车

10.什么是适配器模式?

答：适配器模式是将一个类的接口变成客户端所期望的另一种接口，从而使原本因接口不匹配而无法一起工作的两个类能够在一起工作。

优点：

- 可以让两个没有关联的类一起运行，起着中间转换的作用；
- 灵活性好，不会破坏原有的系统。

缺点：过多地使用适配器，容易使代码结构混乱，如明明看到调用的是 A 接口，内部调用的却是 B 接口的实现。

以生活中的例子来说，比如有一个充电器是 MicroUSB 接口，而手机充电口却是 TypeC 的，这个时候就需要一个把 MicroUSB 转换成 TypeC 的适配器，如下图所示：

适配器实现代码如下：

```
/\* \* 传统的充电线 MicroUSB \*/
interface MicroUSB {
    void charger();
}
/\* \* TypeC 充电口 \*/
interface ITypeC {
    void charger();
}
class TypeC implements ITypeC {
    @Override
```

```
    public void charger() {
        System.out.println("TypeC 充电");
    }
}
/\* \* 适配器 \*/
class AdapterMicroUSB implements MicroUSB {
    private TypeC typeC;

    public AdapterMicroUSB(TypeC typeC) {
        this.typeC = typeC;
    }

    @Override
    public void charger() {
        typeC.charger();
    }
}
/\* \* 测试调用 \*/
public class AdapterTest {
    public static void main(String[] args) {
        TypeC typeC = new TypeC();
        MicroUSB microUSB = new AdapterMicroUSB(typeC);
        microUSB.charger();

    }
}
```

程序执行结果：

> TypeC 充电

11.JDK 类库常用的设计模式有哪些?

答：JDK 常用的设计模式如下：

1）工厂模式

java.text.DateFormat 工具类，它用于格式化一个本地日期或者时间。

```
public final static DateFormat getDateInstance();
public final static DateFormat getDateInstance(int style);
public final static DateFormat getDateInstance(int style,Locale locale);
```

加密类

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("DESede");
Cipher cipher = Cipher.getInstance("DESede");
```

2）适配器模式

把其他类适配为集合类

```
List<Integer> arrayList = java.util.Arrays.asList(new Integer[]{1,2,3});
List<Integer> arrayList = java.util.Arrays.asList(1,2,3);
```

3）代理模式

如 JDK 本身的动态代理。

```
interface Animal {
    void eat();
}
class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("The dog is eating");
    }
}
class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("The cat is eating");
    }
}


//    代理类
```

```
// JDK 代理类
class AnimalProxy implements InvocationHandler {
    private Object target; // 代理对象
    public Object getInstance(Object target) {
        this.target = target;
        // 取得代理对象
        return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterf
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("调用前");
        Object result = method.invoke(target, args); // 方法调用
        System.out.println("调用后");
        return result;
    }
}

public static void main(String[] args) {
    // JDK 动态代理调用
    AnimalProxy proxy = new AnimalProxy();
    Animal dogProxy = (Animal) proxy.getInstance(new Dog());
    dogProxy.eat();
}
```

4）单例模式

全局只允许有一个实例，比如：

```
Runtime.getRuntime();
```

5）装饰器

为一个对象动态的加上一系列的动作，而不需要因为这些动作的不同而产生大量的继承类。

```
java.io.BufferedInputStream(InputStream);
java.io.DataInputStream(InputStream);
java.io.BufferedOutputStream(OutputStream);
java.util.zip.ZipOutputStream(OutputStream);
java.util.Collections.checkedList(List list, Class type) ;
```

6）模板方法模式

定义一个操作中算法的骨架，将一些步骤的执行延迟到其子类中。

比如，Arrays.sort() 方法，它要求对象实现 Comparable 接口。

```
class Person implements Comparable{
    private Integer age;
    public Person(Integer age){
        this.age = age;
    }
    @Override
    public int compareTo(Object o) {
        Person person = (Person)o;
        return this.age.compareTo(person.age);
    }
}
public class SortTest(){
    public static void main(String[] args){
        Person p1 = new Person(10);
        Person p2 = new Person(5);
        Person p3 = new Person(15);
        Person[] persons = {p1,p2,p3};
        //排序
        Arrays.sort(persons);
    }
}
```

12.IO 使用了什么设计模式?

答：IO 使用了适配器模式和装饰器模式。

- 适配器模式：由于 InputStream 是字节流不能享受到字符流读取字符那么便捷的功能，借助 InputStreamReader 将其转为 Reader 子类，因而可以拥有便捷操作文本文件方法；

- 装饰器模式：将 InputStream 字节流包装为其他流的过程就是装饰器模式，比如，包装为 FileInputStream、ByteArrayInputStream、PipedInputStream 等。

13.Spring 中都使用了哪些设计模式?

答：Spring 框架使用的设计模式如下。

- 代理模式：在 AOP 中有使用
- 单例模式：bean 默认是单例模式
- 模板方法模式：jdbcTemplate
- 工厂模式：BeanFactory
- 观察者模式：Spring 事件驱动模型就是观察者模式很经典的一个应用，比如，ContextStartedEvent 就是 ApplicationContext 启动后触发的事件
- 适配器模式：Spring MVC 中也是用到了适配器模式适配 Controller

> 点击此处下载本文源码

下一章

还没有评论

评论

欢迎，关注我的公众号 “DailyProgrammer”，每天8点为你推送精彩的技术文章！！！

洲洋1984
关注 - 0
粉丝 - 4
+加关注