第14讲 | 谈谈你知道的设计模式？

2018-06-05 杨晓峰

第14讲 | 谈谈你知道的设计模式？
朗读人：黄洲君　08′23″ | 3.84M

设计模式是人们为软件开发中相同表征的问题，抽象出的可重复利用的解决方案。在某种程度上，设计模式已经代表了一些特定情况的最佳实践，同时也起到了软件工程师之间沟通的“行话”的作用。理解和掌握典型的设计模式，有利于我们提高沟通、设计的效率和质量。

今天我要问你的问题是，谈谈你知道的设计模式？请手动实现单例模式，Spring 等框架中使用了哪些模式？

典型回答

大致按照模式的应用目标分类，设计模式可以分为创建型模式、结构型模式和行为型模式。

- 创建型模式，是对对象创建过程的各种问题和解决方案的总结，包括各种工厂模式（Factory、Abstract Factory）、单例模式（Singleton）、构建器模式（Builder）、原型模式（ProtoType）。

- 结构型模式，是针对软件设计结构的总结，关注于类、对象继承、组合方式的实践经验。常见的结构型模式，包括桥接模式（Bridge）、适配器模式（Adapter）、装饰者模式（Decorator）、代理模式（Proxy）、组合模式（Composite）、外观模式（Facade）、享元模式（Flyweight）等。

- 行为型模式，是从类或对象之间交互、职责划分等角度总结的模式。比较常见的行为型模式有策略模式（Strategy）、解释器模式（Interpreter）、命令模式（Command）、观察者模式（Observer）、迭代器模式（Iterator）、模板方法模式（Template Method）、访问者模式（Visitor）。

考点分析

这个问题主要是考察你对设计模式的了解和掌握程度，更多相关内容你可以参考：https://en.wikipedia.org/wiki/Design_Patterns。

我建议可以在回答时适当地举些例子，更加清晰地说明典型模式到底是什么样子，典型使用场景是怎样的。这里举个 Java 基础类库中的例子供你参考。

首先，专栏第 11 讲刚介绍过 IO 框架，我们知道 InputStream 是一个抽象类，标准类库中提供了 FileInputStream、ByteArrayInputStream 等各种不同的子类，分别从不同角度对 InputStream 进行了功能扩展，这是典型的装饰器模式应用案例。

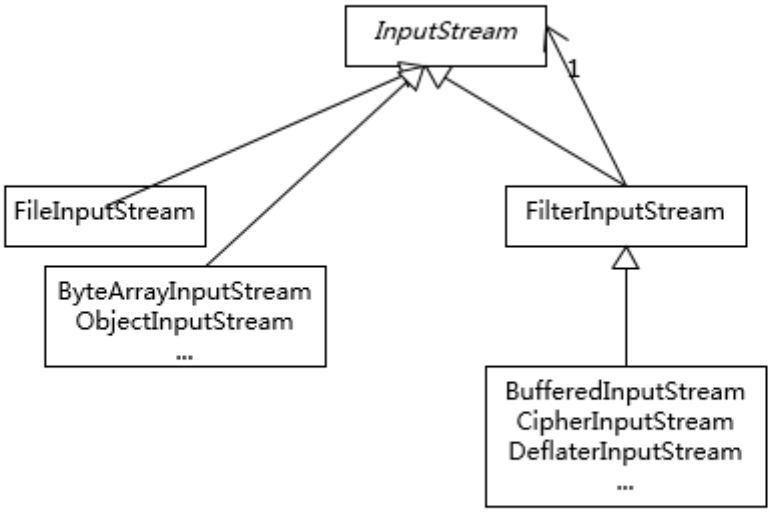
识别装饰器模式，可以通过**识别类设计特征**来进行判断，也就是其类构造函数以**相同的**抽象类或者接口为输入参数。

因为装饰器模式本质上是包装同类型实例，我们对目标对象的调用，往往会通过包装类覆盖过的方法，迂回调用被包装的实例，这就可以很自然地实现增加额外逻辑的目的，也就是所谓的“装饰”。

例如，BufferedInputStream 经过包装，为输入流过程增加缓存，类似这种装饰器还可以多次嵌套，不断地增加不同层次的功能。

```
public BufferedInputStream(InputStream in)
```

我在下面的类图里，简单总结了 InputStream 的装饰模式实践。



接下来再看第二个例子。创建型模式尤其是工厂模式，在我们的代码中随处可见，我举个相对不同的 API 设计实践。比如，JDK 最新版本中 HTTP/2 Client API，下面这个创建 HttpRequest 的过程，就是典型的构建器模式（Builder），通常会被实现成fluent 风格的 API，也有人叫它方法链。

```
HttpRequest request = HttpRequest.newBuilder(new URI(uri))
                      .header(headerAlice, valueAlice)
                      .headers(headerBob, value1Bob,
                       headerCarl, valueCarl,
                       headerBob, value2Bob)
                      .GET()
                      .build();
```

使用构建器模式，可以比较优雅地解决构建复杂对象的麻烦，这里的“复杂”是指类似需要输入的参数组合较多，如果用构造函数，我们往往需要为每一种可能的输入参数组合实现相应的构造函数，一系列复杂的构造函数会让代码阅读性和可维护性变得很差。

上面的分析也进一步反映了创建型模式的初衷，即，将对象创建过程单独抽象出来，从结构上把对象使用逻辑和创建逻辑相互独立，隐藏对象实例的细节，进而为使用者实现了更加规范、统一的逻辑。

更进一步进行设计模式考察，面试官可能会：

- 希望你写一个典型的设计模式实现。这虽然看似简单，但即使是最简单的单例，也能够综合考察代码基本功。
- 考察典型的设计模式使用，尤其是结合标准库或者主流开源框架，考察你对业界良好实践的掌握程度。

在面试时如果恰好问到你不熟悉的模式，你可以稍微引导一下，比如介绍你在产品中使用了什么自己相对熟悉的模式，试图解决什么问题，它们的优点和缺点等。

下面，我会针对前面两点，结合代码实例进行分析。

知识扩展

我们来实现一个日常非常熟悉的单例设计模式。看起来似乎很简单，那么下面这个样例符合基本需求吗？

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

是不是总感觉缺了点什么？原来，Java 会自动为没有明确声明构造函数的类，定义一个 public 的无参数的构造函数，所以上面的例子并不能保证额外的对象不被创建出来，别人完全可以直接“new Singleton()”，那我们应该怎么处理呢？

不错，可以为单例定义一个 private 的构造函数（也有建议声明为枚举，这是有争议的，我个人不建议选择相对复杂的枚举，毕竟日常开发不是学术研究）。这样还有什么改进的余地吗？

专栏第 10 讲介绍 ConcurrentHashMap 时，提到过标准类库中很多地方使用懒加载（lazy-load），改善初始内存开销，单例同样适用，下面是修正后的改进版本。

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

```
            if (instance == null) {
            instance = new Singleton();
            }
        return instance;
        }
    }
```

这个实现在单线程环境不存在问题，但是如果处于并发场景，就需要考虑线程安全，最熟悉的就莫过于“双检锁”，其要点在于：

- 这里的 volatile 能够提供可见性，以及保证 getInstance 返回的是初始化完全的对象。
- 在同步之前进行 null 检查，以尽量避免进入相对昂贵的同步块。
- 直接在 class 级别进行同步，保证线程安全的类方法调用。

```
public class Singleton {
    private static volatile Singleton singleton = null;
    private Singleton() {
    }

    public static Singleton getSingleton() {
        if (singleton == null) { // 尽量避免重复进入同步块
            synchronized (Singleton.class) { // 同步.class，意味着对同步类方法调用
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

在这段代码中，争论较多的是 volatile 修饰静态变量，当 Singleton 类本身有多个成员变量时，需要保证初始化过程完成后，才能被 get 到。

在现代 Java 中，内存排序模型（JMM）已经非常完善，通过 volatile 的 write 或者 read，能保证所谓的 happen-before，也就是避免常被提到的指令重排。换句话说，构造对象的 store

指令能够被保证一定在 volatile read 之前。

当然，也有一些人推荐利用内部类持有静态对象的方式实现，其理论依据是对象初始化过程中隐含的初始化锁（有兴趣的话你可以参考jls-12.4.2 中对 LC 的说明），这种和前面的双检锁实现都能保证线程安全，不过语法稍显晦涩，未必有特别的优势。

```
public class Singleton {
    private Singleton(){}
    public static Singleton getSingleton(){
        return Holder.singleton;
    }

    private static class Holder {
        private static Singleton singleton = new Singleton();
    }
}
```

所以，可以看出，即使是看似最简单的单例模式，在增加各种高标准需求之后，同样需要非常多的实现考量。

上面是比较学究的考察，其实实践中未必需要如此复杂，如果我们看 Java 核心类库自己的单例实现，比如java.lang.Runtime，你会发现：

- 它并没使用复杂的双检锁之类。
- 静态实例被声明为 final，这是被通常实践忽略的，一定程度保证了实例不被篡改（专栏第 6 讲介绍过，反射之类可以绕过私有访问限制），也有有限的保证执行顺序的语义。

```
private static final Runtime currentRuntime = new Runtime();
private static Version version;
// …
public static Runtime getRuntime() {
    return currentRuntime;
}


/** Don't let anyone else instantiate this class */
private Runtime() {}
```

前面说了不少代码实践，下面一起来简要看看主流开源框架，如 Spring 等如何在 API 设计中使用设计模式。你至少要有个大体的印象，如：

- BeanFactory和ApplicationContext应用了工厂模式。
- 在 Bean 的创建中，Spring 也为不同 scope 定义的对象，提供了单例和原型等模式实现。
- 我在专栏第 6 讲介绍的 AOP 领域则是使用了代理模式、装饰器模式、适配器模式等。
- 各种事件监听器，是观察者模式的典型应用。
- 类似 JdbcTemplate 等则是应用了模板模式。

今天，我与你回顾了设计模式的分类和主要类型，并从 Java 核心类库、开源框架等不同角度分析了其采用的模式，并结合单例的不同实现，分析了如何实现符合线程安全等需求的单例，希望可以对你的工程实践有所帮助。另外，我想最后补充的是，设计模式也不是银弹，要避免滥用或者过度设计。

一课一练

关于设计模式你做到心中有数了吗？你可以思考下，在业务代码中，经常发现大量 XXFacade，外观模式是解决什么问题？适用于什么场景？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

精选留言

sunlight001 7

结合流行的开源框架，或者自己的项目学设计模式是很好的办法，生学很容易看不懂学不下去，xxxfascade是门面模式，为复杂的逻辑提供简单的借口，设计模式学的时候还能明白，但是用的时候就不知道该怎么用了，我们怎么在项目中使用设计模式呢？

2018-06-05

作者回复

不必为了模式而用，优先解决开发、维护中的痛点

2018-06-06

李昭文TSOS 4

为什么我去查Runtime的源码，currentRuntime没有被final修饰呢？

2018-06-05

作者回复

什么版本？我这是最新的

2018-06-06

公众号:代码荣耀 3

门面模式形象上来讲就是在原系统之前放置了一个新的代理对象，只能通过该对象才能使用该系统，不再允许其它方式访问该系统。该代理对象封装了访问原系统的所有规则和接口方法，提供的API接口较之使用原系统会更加的简单。

举例:JUnitCore是JUnit类的 Facade模式的实现类，外部使用该代理对象与JUnit进行统一交互，驱动执行测试代码。

使用场景:当我们希望封装或隐藏原系统；当我们使用原系统的功能并希望增加一些新的功能；编写新类的成本远小于所有人学会使用或者未来维护原系统所需的成本；

缺点:违反了开闭原则。如有扩展，只能直接修改代理对象。

2018-06-05

作者回复

不错

2018-06-06

Sin0 2

有一点理解不太一致，单例模式double check中synchronized就已经可以提供可见性，volatile的作用主要体现在禁指令重排！

2018-06-05

作者回复

不冲突，sync也不是必然走到

2018-06-06

星空 1

外观模式为子系统中一组接口提供一个统一访问的接口，降低了客户端与子系统之间的耦合，简化了系统复杂度。缺点是违反了开闭原则。适用于为一系列复杂的子系统提供一个友好简单的入口，将子系统与客户端解偶。公司基础paas平台用到了外观模式，具体是定义一个ServiceFacade，然后通过继承众多xxService,对外提供子xxService的服务。

2018-06-05

作者回复

业务开发很普遍

2018-06-06

李志博 1

Spring 内部的asm 模块 用到了访问者模式

2018-06-05

润兹 1

在没用facade之前，为了完成某个功能需要调用各子系统的各方法进行组合才能完成，用了facade之后相当于把多个方法调用聚合成了一个方法，方便用户调用。

2018-06-05

Walter 0

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。它向现有的系统添加一个接口，来隐藏系统的复杂性。
这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

意图：为子系统中的一组接口提供一个一致的界面，外观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。
主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。
何时使用： 1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。 2、定义系统的入口。
如何解决：客户端不与系统耦合，外观类与系统耦合。
关键代码：在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。
应用实例： 1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。 2、JAVA 的三层开发模式。
优点： 1、减少系统相互依赖。 2、提高灵活性。 3、提高了安全性。
缺点：不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。
使用场景： 1、为复杂的模块或子系统提供外界访问的模块。 2、子系统相对独立。 3、预防低水平人员带来的风险。
注意事项：在层次化结构中，可以使用外观模式定义系统中每一层的入口。

2018-06-07

作者回复

不错，业务系统多见

2018-06-07

锐 0

设计模式也不是银弹，要避免滥用或者过度设计。这句话深有体会，为了使用设计模式而使用，有时很头疼

2018-06-06

雷霹雳的爸爸 0

单态这里有一个问题问老师，如果不用double check的话，仅在声明静态成员的同时即实例化之，那么是否就不用volatile关键字修饰这个字段了？为什么？

2018-06-06

田维俊 0

公司项目是一个基于spingboot、mybatis开发的web后端管理项目。现在的问题是 不同角色登录到系统看到的模块和模块里面的数据是不一样的，有时虽然看到的模块一样，但是由于角色不一样，所以显示的数据是不一样，在这样的情况下，会经常在service层方法里面判断角色然后改变mapper的数据操作条件或调用mapper的不同方法。由于在service层频繁的判断角色感觉很不雅，新增角色就要加判断，哎，感觉可以用策略设计模式，可是不知道怎么具体设计。

2018-06-06

softpower2018 0

通过封装的方式，对外屏蔽内部复杂业务逻辑，实现使用方与具体实现的分离。门面模式

2018-06-05

yearning 0

Facade（外观模式）

接口隔离模式。处理组件中外部客户程序和组件中各种复杂的子系统高耦合情况，定义一个高层接口，为子系统中的一组接口提供一个一致（稳定）的界面，使得更简单的使用。facade简化整个组件系统的接口，同时子系统的任何变化都不会影响到facade接口。

有一个更简单的称呼，门面模式，打个比方说，你去商店，你只需要告诉店员，你需要什么，至于商店中复杂的采购系统，库存系统，收银系统一概对你不可见。

在经常使用的hibernate，当我们想插入一条用户信息，facade接口中insert(User user)，我们只要传递User对象，至于背后的操作对外部调用是不可见。

facade模式是从架构的层次去看整个系统，而是一两个类之间单纯解耦。

2018-06-05

Geek_028e77 0

facade模式 主要屏蔽系统内部细节实现，通过facade模式封装统一的接口 提供给外部调用着.有一个优势 ，当内部系统做变更 优化时，这对外部调用者来说是透明的，一定程度上降低了系统间耦合性...个人理解

2018-06-05

wutao 0

Spring中还用到了策略模式吧

2018-06-05

作者回复

当然，列出的只是简要说明

2018-06-06