二

14 谈谈你知道的设计模式？-极客时间

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

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。https://en.wikipedia.org/wiki/Design_Patterns。

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。

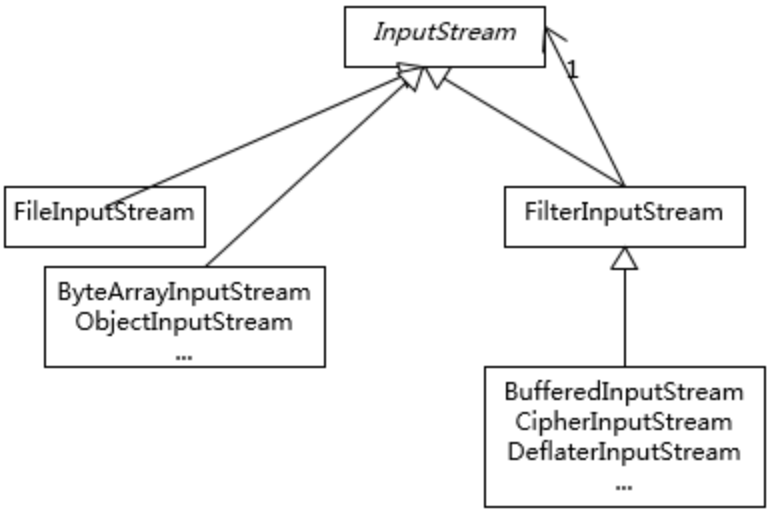
识别装饰器模式，可以通过**识别类设计特征**来进行判断，也就是其类构造函数以**相同的**抽象类或者接口为输入参数。

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。

上一页 下一页