

51 | 适配器模式：代理、适配器、桥接、装饰，这四个模式有何区别？

2020-02-28 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 13:03 大小 10.47M



前面几节课我们学习了代理模式、桥接模式、装饰器模式，今天，我们再来学习一个比较常用的结构型模式：适配器模式。这个模式相对来说还是比较简单、好理解的，应用场景也很具体，总体上来讲比较好掌握。


关于适配器模式，今天我们主要学习它的两种实现方式，类适配器和对象适配器，以及 5 种常见的应用场景。同时，我还会通过剖析 slf4j 日志框架，来给你展示这个模式在真实项目中的应用。除此之外，在文章的最后，我还对代理、桥接、装饰器、适配器，这 4 种代码结构非常相似的设计模式做简单的对比，对这几节内容做一个简单的总结。

话不多说，让我们正式开始今天的学习吧！

适配器模式的原理与实现

适配器模式的英文翻译是 **Adapter Design Pattern**。顾名思义，这个模式就是用来做适配的，它将不兼容的接口转换为可兼容的接口，让原本由于接口不兼容而不能一起工作的类可以一起工作。对于这个模式，有一个经常被拿来解释它的例子，就是 USB 转接头充当适配器，把两种不兼容的接口，通过转接变得可以一起工作。

原理很简单，我们再来看下它的代码实现。适配器模式有两种实现方式：类适配器和对象适配器。其中，类适配器使用继承关系来实现，对象适配器使用组合关系来实现。具体的代码实现如下所示。其中，ITarget 表示要转化成的接口定义。Adaptee 是一组不兼容 ITarget 接口定义的接口，Adaptor 将 Adaptee 转化成一组符合 ITarget 接口定义的接口。

 复制代码

```
1 // 类适配器：基于继承
2 public interface ITarget {
3     void f1();
4     void f2();
5     void fc();
6 }
7
8 public class Adaptee {
9     public void fa() { //... }
10    public void fb() { //... }
11    public void fc() { //... }
12 }
13
14 public class Adaptor extends Adaptee implements ITarget {
15     public void f1() {
16         super.fa();
17     }
18
19     public void f2() {
20         //...重新实现f2()...
21     }
22
23     // 这里fc()不需要实现，直接继承自Adaptee，这是跟对象适配器最大的不同点
24 }
25
26 // 对象适配器：基于组合
27 public interface ITarget {
28     void f1();
29     void f2();
30     void fc();
31 }
32
33 public class Adaptee {
```

```
34     public void fa() { //... }
35     public void fb() { //... }
36     public void fc() { //... }
37 }
38
39 public class Adaptor implements ITarget {
40     private Adaptee adaptee;
41
42     public Adaptor(Adaptee adaptee) {
43         this.adaptee = adaptee;
44     }
45
46     public void f1() {
47         adaptee.fa(); //委托给Adaptee
48     }
49
50     public void f2() {
51         //...重新实现f2()...
52     }
53
54     public void fc() {
55         adaptee.fc();
56     }
57 }
```

针对这两种实现方式，在实际的开发中，到底该如何选择使用哪一种呢？判断的标准主要有两个，一个是 Adaptee 接口的个数，另一个是 Adaptee 和 ITarget 的契合程度。

如果 Adaptee 接口并不多，那两种实现方式都可以。

如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都相同，那我们推荐使用类适配器，因为 Adaptor 复用父类 Adaptee 的接口，比起对象适配器的实现方式，Adaptor 的代码量要少一些。

如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都不相同，那我们推荐使用对象适配器，因为组合结构相对于继承更加灵活。

适配器模式应用场景总结

原理和实现讲完了，都不复杂。我们再来看，到底什么时候会用到适配器模式呢？

一般来说，适配器模式可以看作一种“补偿模式”，用来补救设计上的缺陷。应用这种模式算是“无奈之举”。如果在设计初期，我们就能协调规避接口不兼容的问题，那这种模式就

没有应用的机会了。

前面我们反复提到，适配器模式的应用场景是“接口不兼容”。那在实际的开发中，什么情况下才会出现接口不兼容呢？我建议你先自己思考一下这个问题，然后再来看我下面的总结。

1. 封装有缺陷的接口设计

假设我们依赖的外部系统在接口设计方面有缺陷（比如包含大量静态方法），引入之后会影响到我们自身代码的可测试性。为了隔离设计上的缺陷，我们希望对外部系统提供的接口进行二次封装，抽象出更好的接口设计，这个时候就可以使用适配器模式了。

具体我还是举个例子来解释一下，你直接看代码应该会更清晰。具体代码如下所示：

 复制代码

```
1 public class CD { //这个类来自外部sdk，我们无权修改它的代码
2     //...
3     public static void staticFunction1() { //... }
4
5     public void uglyNamingFunction2() { //... }
6
7     public void tooManyParamsFunction3(int paramA, int paramB, ...) { //... }
8
9     public void lowPerformanceFunction4() { //... }
10 }
11
12 // 使用适配器模式进行重构
13 public class ITarget {
14     void function1();
15     void function2();
16     void fucntion3(ParamsWrapperDefinition paramsWrapper);
17     void function4();
18     //...
19 }
20 // 注意：适配器类的命名不一定非得末尾带Adaptor
21 public class CDAdaptor extends CD implements ITarget {
22     //...
23     public void function1() {
24         super.staticFunction1();
25     }
26
27     public void function2() {
28         super.uglyNamingFucntion2();
29     }
30 }
```


```
31     public void function3(ParamsWrapperDefinition paramsWrapper) {
32         super.tooManyParamsFunction3(paramsWrapper.getParamA(), ...);
33     }
34
35     public void function4() {
36         //...reimplement it...
37     }
38 }
```

2. 统一多个类的接口设计

某个功能的实现依赖多个外部系统（或者说类）。通过适配器模式，将它们的接口适配为统一的接口定义，然后我们就可以使用多态的特性来复用代码逻辑。具体我还是举个例子来解释一下。

假设我们的系统要对用户输入的文本内容做敏感词过滤，为了提高过滤的召回率，我们引入了多款第三方敏感词过滤系统，依次对用户输入的内容进行过滤，过滤掉尽可能多的敏感词。但是，每个系统提供的过滤接口都是不同的。这就意味着我们没法复用一套逻辑来调用各个系统。这个时候，我们就可以使用适配器模式，将所有系统的接口适配为统一的接口定义，这样我们可以复用调用敏感词过滤的代码。

你可以配合着下面的代码示例，来理解我刚才举的这个例子。

 复制代码


```
1  public class ASensitiveWordsFilter { // A敏感词过滤系统提供的接口
2      //text是原始文本，函数输出用***替换敏感词之后的文本
3      public String filterSexyWords(String text) {
4          // ...
5      }
6
7      public String filterPoliticalWords(String text) {
8          // ...
9      }
10 }
11
12 public class BSensitiveWordsFilter { // B敏感词过滤系统提供的接口
13     public String filter(String text) {
14         //...
15     }
16 }
17
18 public class CSensitiveWordsFilter { // C敏感词过滤系统提供的接口
19     public String filter(String text, String mask) {
```



```
20     //...
21 }
22 }
23
24 // 未使用适配器模式之前的代码：代码的可测试性、扩展性不好
25 public class RiskManagement {
26     private ASensitiveWordsFilter aFilter = new ASensitiveWordsFilter();
27     private BSensitiveWordsFilter bFilter = new BSensitiveWordsFilter();
28     private CSensitiveWordsFilter cFilter = new CSensitiveWordsFilter();
29
30     public String filterSensitiveWords(String text) {
31         String maskedText = aFilter.filterSexyWords(text);
32         maskedText = aFilter.filterPoliticalWords(maskedText);
33         maskedText = bFilter.filter(maskedText);
34         maskedText = cFilter.filter(maskedText, "***");
35         return maskedText;
36     }
37 }
38
39 // 使用适配器模式进行改造
40 public interface ISensitiveWordsFilter { // 统一接口定义
41     String filter(String text);
42 }
43
44 public class ASensitiveWordsFilterAdaptor implements ISensitiveWordsFilter {
45     private ASensitiveWordsFilter aFilter;
46     public String filter(String text) {
47         String maskedText = aFilter.filterSexyWords(text);
48         maskedText = aFilter.filterPoliticalWords(maskedText);
49         return maskedText;
50     }
51 }
52 //...省略BSensitiveWordsFilterAdaptor、CSensitiveWordsFilterAdaptor...
53
54 // 扩展性更好，更加符合开闭原则，如果添加一个新的敏感词过滤系统，
55 // 这个类完全不需要改动；而且基于接口而非实现编程，代码的可测试性更好。
56 public class RiskManagement {
57     private List<ISensitiveWordsFilter> filters = new ArrayList<>();
58
59     public void addSensitiveWordsFilter(ISensitiveWordsFilter filter) {
60         filters.add(filter);
61     }
62
63     public String filterSensitiveWords(String text) {
64         String maskedText = text;
65         for (ISensitiveWordsFilter filter : filters) {
66             maskedText = filter.filter(maskedText);
67         }
68         return maskedText;
69     }
70 }
```

3. 替换依赖的外部系统

当我们把项目中依赖的一个外部系统替换为另一个外部系统的时候，利用适配器模式，可以减少对代码的改动。具体的代码示例如下所示：

 复制代码

```
1  // 外部系统A
2  public interface IA {
3      //...
4      void fa();
5  }
6  public class A implements IA {
7      //...
8      public void fa() { //... }
9  }
10 // 在我们的项目中，外部系统A的使用示例
11 public class Demo {
12     private IA a;
13     public Demo(IA a) {
14         this.a = a;
15     }
16     //...
17 }
18 Demo d = new Demo(new A());
19
20 // 将外部系统A替换成外部系统B
21 public class BAdaptor implements IA {
22     private B b;
23     public BAdaptor(B b) {
24         this.b = b;
25     }
26     public void fa() {
27         //...
28         b.fb();
29     }
30 }
31 // 借助BAdaptor，Demo的代码中，调用IA接口的地方都无需改动，
32 // 只需要将BAdaptor如下注入到Demo即可。
33 Demo d = new Demo(new BAdaptor(new B()));
```


4. 兼容老版本接口

在做版本升级的时候，对于一些要废弃的接口，我们不直接将其删除，而是暂时保留，并且标注为 deprecated，并将内部实现逻辑委托为新的接口实现。这样做的好处是，让使用它

的项目有个过渡期，而不是强制进行代码修改。这也可以粗略地看作适配器模式的一个应用场景。同样，我还是通过一个例子，来进一步解释一下。

JDK1.0 中包含一个遍历集合容器的类 Enumeration。JDK2.0 对这个类进行了重构，将它改名为 Iterator 类，并且对它的代码实现做了优化。但是考虑到如果将 Enumeration 直接从 JDK2.0 中删除，那使用 JDK1.0 的项目如果切换到 JDK2.0，代码就会编译不通过。为了避免这种情况的发生，我们必须把项目中所有使用到 Enumeration 的地方，都修改为使用 Iterator 才行。

单独一个项目做 Enumeration 到 Iterator 的替换，勉强还能接受。但是，使用 Java 开发的项目太多了，一次 JDK 的升级，导致所有的项目不做代码修改就会编译报错，这显然是不合理的。这就是我们经常所说的不兼容升级。为了做到兼容使用低版本 JDK 的老代码，我们可以暂时保留 Enumeration 类，并将其实现替换为直接调用 Iterator。代码示例如下所示：

 复制代码

```
1 public class Collections {
2     public static Enumeration enumeration(final Collection c) {
3         return new Enumeration() {
4             Iterator i = c.iterator();
5
6             public boolean hasMoreElements() {
7                 return i.hasNext();
8             }
9
10            public Object nextElement() {
11                return i.next();
12            }
13        }
14    }
15 }
```

5. 适配不同格式的数据

前面我们讲到，适配器模式主要用于接口的适配，实际上，它还可以用在不同格式的数据之间的适配。比如，把从不同征信系统拉取的不同格式的征信数据，统一为相同的格式，以方便存储和使用。再比如，Java 中的 Arrays.asList() 也可以看作一种数据适配器，将数组类型的数据转化为集合容器类型。


```
1 List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");
```

[复制代码](#)

剖析适配器模式在 Java 日志中的应用

Java 中有很多日志框架，在项目开发中，我们常常用它们来打印日志信息。其中，比较常用的有 log4j、logback，以及 JDK 提供的 JUL(java.util.logging) 和 Apache 的 JCL(Jakarta Commons Logging) 等。

大部分日志框架都提供了相似的功能，比如按照不同级别（debug、info、warn、error.....）打印日志等，但它们却并没有实现统一的接口。这主要可能是历史的原因，它不像 JDBC 那样，一开始就制定了数据库操作的接口规范。

如果我们只是开发一个自己用的项目，那用什么日志框架都可以，log4j、logback 随便选一个就好。但是，如果我们开发的是一个集成到其他系统的组件、框架、类库等，那日志框架的选择就没那么随意了。

比如，项目中用到的某个组件使用 log4j 来打印日志，而我们项目本身使用的是 logback。将组件引入到项目之后，我们的项目就相当于有了两套日志打印框架。每种日志框架都有自己特有的配置方式。所以，我们要针对每种日志框架编写不同的配置文件（比如，日志存储的文件地址、打印日志的格式）。如果引入多个组件，每个组件使用的日志框架都不一样，那日志本身的管理工作就变得非常复杂。所以，为了解决这个问题，我们需要统一日志打印框架。

如果你是做 Java 开发的，那 Slf4j 这个□日志框架你肯定不陌生，它相当于 JDBC 规范，提供了一套打印日志的统一接口规范。不过，它只定义了接口，并没有提供具体的实现，需要配合其他日志框架（log4j、logback.....）来使用。

不仅如此，Slf4j 的出现晚于 JUL、JCL、log4j 等日志框架，所以，这些日志框架也不可能牺牲掉版本兼容性，将接口改造成符合 Slf4j 接口规范。Slf4j 也事先考虑到了这个问题，所以，它不仅仅提供了统一的接口定义，还提供了针对不同日志框架的适配器。对不同日志框架的接口进行二次封装，适配成统一的 Slf4j 接口定义。具体的代码示例如下所示：

```
1 // slf4j统一的接口定义
```

[复制代码](#)

```
2 package org.slf4j;
3 public interface Logger {
4     public boolean isTraceEnabled();
5     public void trace(String msg);
6     public void trace(String format, Object arg);
7     public void trace(String format, Object arg1, Object arg2);
8     public void trace(String format, Object[] argArray);
9     public void trace(String msg, Throwable t);
10
11     public boolean isDebugEnabled();
12     public void debug(String msg);
13     public void debug(String format, Object arg);
14     public void debug(String format, Object arg1, Object arg2);
15     public void debug(String format, Object[] argArray);
16     public void debug(String msg, Throwable t);
17
18     //...省略info、warn、error等一堆接口
19 }
20
21 // log4j日志框架的适配器
22 // Log4jLoggerAdapter实现了LocationAwareLogger接口,
23 // 其中LocationAwareLogger继承自Logger接口,
24 // 也就相当于Log4jLoggerAdapter实现了Logger接口。
25 package org.slf4j.impl;
26 public final class Log4jLoggerAdapter extends MarkerIgnoringBase
27     implements LocationAwareLogger, Serializable {
28     final transient org.apache.log4j.Logger logger; // log4j
29
30     public boolean isDebugEnabled() {
31         return logger.isDebugEnabled();
32     }
33
34     public void debug(String msg) {
35         logger.log(FQCN, Level.DEBUG, msg, null);
36     }
37
38     public void debug(String format, Object arg) {
39         if (logger.isDebugEnabled()) {
40             FormattingTuple ft = MessageFormatter.format(format, arg);
41             logger.log(FQCN, Level.DEBUG, ft.getMessage(), ft.getThrowable());
42         }
43     }
44
45     public void debug(String format, Object arg1, Object arg2) {
46         if (logger.isDebugEnabled()) {
47             FormattingTuple ft = MessageFormatter.format(format, arg1, arg2);
48             logger.log(FQCN, Level.DEBUG, ft.getMessage(), ft.getThrowable());
49         }
50     }
51
52     public void debug(String format, Object[] argArray) {
53         if (logger.isDebugEnabled()) {
```

```
54     FormattingTuple ft = MessageFormatter.arrayFormat(format, argArray);
55     logger.log(FQCN, Level.DEBUG, ft.getMessage(), ft.getThrowable());
56 }
57 }
58
59 public void debug(String msg, Throwable t) {
60     logger.log(FQCN, Level.DEBUG, msg, t);
61 }
62 //...省略一堆接口的实现...
63 }
```

所以，在开发业务系统或者开发框架、组件的时候，我们统一使用 Slf4j 提供的接口来编写打印日志的代码，具体使用哪种日志框架实现（log4j、logback.....），是可以动态地指定的（使用 Java 的 SPI 技术，这里我不多解释，你自行研究吧），只需要将相应的 SDK 导入到项目中即可。

不过，你可能会说，如果一些老的项目没有使用 Slf4j，而是直接使用比如 JCL 来打印日志，那如果想要替换成其他日志框架，比如 log4j，该怎么办呢？实际上，Slf4j 不仅仅提供了从其他日志框架到 Slf4j 的适配器，还提供了反向适配器，也就是从 Slf4j 到其他日志框架的适配。我们可以先将 JCL 切换为 Slf4j，然后再将 Slf4j 切换为 log4j。经过两次适配器的转换，我们能就成功将 log4j 切换为了 logback。

代理、桥接、装饰器、适配器 4 种设计模式的区别

代理、桥接、装饰器、适配器，这 4 种模式是比较常用的结构型设计模式。它们的代码结构非常相似。笼统来说，它们都可以称为 Wrapper 模式，也就是通过 Wrapper 类二次封装原始类。

尽管代码结构相似，但这 4 种设计模式的用意完全不同，也就是说要解决的问题、应用场景不同，这也是它们的主要区别。这里我就简单说一下它们之间的区别。

代理模式：代理模式在不改变原始类接口的条件下，为原始类定义一个代理类，主要目的是控制访问，而非加强功能，这是它跟装饰器模式最大的不同。

桥接模式：桥接模式的目的是将接口部分和实现部分分离，从而让它们可以较为容易、也相对独立地加以改变。

装饰器模式：装饰者模式在不改变原始类接口的情况下，对原始类功能进行增强，并且支持多个装饰器的嵌套使用。

适配器模式：适配器模式是一种事后的补救策略。适配器提供跟原始类不同的接口，而代理模式、装饰器模式提供的都是跟原始类相同的接口。

重点回顾

好了，今天的内容到此就讲完了。让我们一块来总结回顾一下，你需要重点掌握的内容。

适配器模式是用来做适配，它将不兼容的接口转换为可兼容的接口，让原本由于接口不兼容而不能一起工作的类可以一起工作。适配器模式有两种实现方式：类适配器和对象适配器。其中，类适配器使用继承关系来实现，对象适配器使用组合关系来实现。

一般来说，适配器模式可以看作一种“补偿模式”，用来补救设计上的缺陷。应用这种模式算是“无奈之举”，如果在设计初期，我们就能协调规避接口不兼容的问题，那这种模式就没有应用的机会了。

那在实际的开发中，什么情况下才会出现接口不兼容呢？我总结下了下面这样 5 种场景：

封装有缺陷的接口设计

统一多个类的接口设计

替换依赖的外部系统

兼容老版本接口

适配不同格式的数据

课堂讨论

今天我们讲到，适配器有两种实现方式：类适配器、对象适配器。那我们之前讲到的代理模式、装饰器模式，是否也同样可以有两种实现方式（类代理模式、对象代理模式，以及类装饰器模式、对象装饰器模式）呢？

欢迎留言和我分享你的思考，如果有收获，也欢迎你把这篇文章分享给你的朋友。

关注极客时间服务号 每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 50 | 装饰器模式：通过剖析Java IO类库源码学习装饰器模式

下一篇 52 | 门面模式：如何设计合理的接口粒度以兼顾接口的易用性和通用性？

精选留言 (24)

写留言



javaadu

2020-02-28

这篇总结将前几节课串联起来了，非常赞👍

课堂讨论：

1. 代理模式支持，基于接口组合代理就是对象匹配，基于继承代理就是类匹配
2. 装饰者模式不支持，这个模式本身是为了避免继承结构爆炸而设计的

展开 ▾

1

12



唐龙

2020-02-28

C++的STL里有大量的适配器，比如迭代器适配器，容器适配器，仿函数适配器。

容器里的反向迭代器reverse_iterator就是对迭代器iterator的一层简单封装。

所谓的栈stack和单向队列queue也是对其他容器的封装，底层默认使用的是双向队列de...

展开 ▾



3



小晏子

2020-02-28

代理模式有两种实现方式：一般情况下，我们让代理类和原始类实现同样的接口。这种就是对象代理模式；但是，如果原始类并没有定义接口，并且原始类代码并不是我们开发维护的。在这种情况下，我们可以通过让代理类继承原始类的方法来实现代理模式，这种属于类代理模式。

装饰器模式没有这两种方式：装饰器模式主要解决继承关系过于复杂的问题，通过组合...

展开 ▾



3



勤劳的明酱

2020-02-28

那SpringAop是代理模式，主要功能却是增强被代理的类，这不是更符合装饰器模式。



2



f

2020-02-29

真的是好文章，讲出了别人讲不出的设计，看了很多遍理解了，但是想要融会贯通不是件容易的事，需要多实践，我需要反复看这边文章来思考这5种场景的实践，这次阅读的理解就是场景二可以指导框架设计，我们只要实现新的过滤器，add，就可以不修改源码来实现新的过滤，简直太棒了

展开 ▾



1



每天晒白牙

2020-02-28

代理模式有两种实现方式

1.代理类和原始类实现相同的接口，原始类只负责原始的业务功能，而代理类通过委托的方式调用原始类来执行业务逻辑，然后可以做一些附加功能。这也是一种基于接口而实现编程的设计思想。这就是基于组合也就是对象模式

2.如果原始类没有定义接口且不是我们开发维护的，这属于对外部类的扩展，可以使用继...

展开 ▾



1



守拙



2020-02-28

课堂讨论

今天我们讲到，适配器有两种实现方式：类适配器、对象适配器。那我们之前讲到的代理模式、装饰器模式，是否也同样可以有两种实现方式（类代理模式、对象代理模式，以及类装饰器模式、对象装饰器模式）呢？ ...

展开 ▾



1



Fstar

2020-03-02

课堂讨论的思考：

对于代理模式，一般是通过组合（对应对象代理模式）的方式实现，继承的实现方式是特殊情况（如无法修改的外部类）下不得已才使用的。可以认为有两种实现方式。

...

展开 ▾



平风造雨

2020-03-01

代理模式使用类代理的目的更明确，大多数场景中代码更少，装饰器模式就是为了解决继承爆炸，并且灵活的选择是否适配，所以装饰器模式默认都应是实例代理。

展开 ▾



柏油

2020-03-01

请问一个类如果需要两种及以上的如何实现做适配呢

展开 ▾



bin

2020-03-01

代理模式可以用类代理模式、对象代理模式。

装饰器模式可以用对象装饰器模式但不可以用类装饰器模式，因为装饰器模式就是为了避免过多的继承而设计的



相逢是缘

2020-03-01

一、定义（理解）：

将不兼容的接口转化为兼容的接口，让原本不能在一起工作的类可以在一起工作。

（适配器模式可以看作一种“补偿模式”，用来补救设计上的缺陷）

二、使用场景：

1) 封装有缺陷的接口（如包含大量的静态的方法，影响测试等）...

展开 ▾



Frank

2020-02-29

适配器模式主要用于接口的适配。将源接口适配为符合客户端调用的目标接口。适配器有两种实现方式：类适配器和对象适配器，前者基于继承，后者集合组合。推荐使用后者，因为组合灵活性比较高，例如使用组合可以将多个源接口适配为一个目标接口。

展开 ▾



高源

2020-02-29

老师讲的好啊，让你明白具体的4个模式的应用和区别，王争老师厉害啊😁要是再把这4个模式用同一个例子分别说明就更好了，这样对于像我这样的需要看代码加深印象的学习者是再好不过了，感谢老师辛勤付出辛苦了

展开 ▾



f

2020-02-29

我的天啊 值得深思 对着每个字思考使用场景

展开 ▾



糖醋排骨

2020-02-28

是不是第二个敏感词汇过滤的例子也运用了桥接模式？通过组合的方式来进行扩展。求老师解答



南山

2020-02-28

实际项目中一直有对外不使用不方便接口的封装，实际用的就是适配器的思想或者说是模

式。

只是限于对适配器的浅显认知，没有意识到时这种模式的应用，更多的是理解为工具类。

醍醐灌顶~

展开 ∨



test

2020-02-28

代理模式：让代理类继承原始类就是类代理模式。不继承则是就是对象代理模式。

装饰器模式：装饰器模式主要解决继承关系过于复杂的问题，所以不适合用类装饰器模式。

展开 ∨



岁月神偷

2020-02-28

我认为装饰器模式不能使用类匹配，类匹配形成了装饰器类与继承类强耦合的关系，本身装饰器是想对实现某一接口的所有实现类提供增强，一旦进行类匹配就变成了这个装饰器只是对某个实现类的增强



tt

2020-02-28

1、代理模式，扩展非功能需求，如缓存、日志、鉴权等，从业务流程出发，代理类仍然需要暴露和被代理类一样的接口，所以也可以用继承来实现，这样代理类对外接口的数量就完全和被代理类一样了，可以覆盖其中需要扩展非功能需求的方法。这就是类代理模式。

老师的讲解是以对象代理模式进行的，即用组合的方式。

...

展开 ∨

