

51-适配器模式：代理、适配器、桥接、装饰，这四个模式有何区别？

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。

继承实现：
组合实现

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。

```
// 类适配器：基于继承
public interface ITarget {
    void f1();
    void f2();
    void fc();
}

public class Adaptee {
    public void fa() { //... }
    public void fb() { //... }
    public void fc() { //... }
}

public class Adaptor extends Adaptee implements ITarget {
    public void f1() {
        super.fa();
    }

    public void f2() {
        //...重新实现f2()...
    }

    // 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
}

// 对象适配器：基于组合
public interface ITarget {
    void f1();
    void f2();
}
```

```

void fc();
}

public class Adaptee {
    public void fa() { //... }
    public void fb() { //... }
    public void fc() { //... }
}

public class Adaptor implements ITarget {
    private Adaptee adaptee;

    public Adaptor(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void f1() {
        adaptee.fa(); //委托给Adaptee
    }

    public void f2() {
        //...重新实现f2()...
    }

    public void fc() {
        adaptee.fc();
    }
}

```

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：

```
public class CD { //这个类来自外部sdk，我们无权修改它的代码
    //...
    public static void staticFunction1() { //... }

    public void uglyNamingFunction2() { //... }

    public void tooManyParamsFunction3(int paramA, int paramB, ...) { //... }

    public void lowPerformanceFunction4() { //... }
}

// 使用适配器模式进行重构
public class ITarget {
    void function1();
    void function2();
    void fuction3(ParamsWrapperDefinition paramsWrapper);
    void function4();
    //...
}

// 注意：适配器类的命名不一定非得末尾带Adaptor
public class CDAdaptor extends CD implements ITarget {
    //...
    public void function1() {
        super.staticFunction1();
    }

    public void function2() {
        super.uglyNamingFuction2();
    }

    public void function3(ParamsWrapperDefinition paramsWrapper) {
        super.tooManyParamsFunction3(paramsWrapper.getParamA(), ...);
    }

    public void function4() {
        //...reimplement it...
    }
}
```

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。

```
public class ASensitiveWordsFilter { // A敏感词过滤系统提供的接口
```

```

//text是原始文本，函数输出用***替换敏感词之后的文本
public String filterSexyWords(String text) {
    // ...
}

public String filterPoliticalWords(String text) {
    // ...
}
}

public class BSensitiveWordsFilter { // B敏感词过滤系统提供的接口
    public String filter(String text) {
        //...
    }
}

public class CSensitiveWordsFilter { // C敏感词过滤系统提供的接口
    public String filter(String text, String mask) {
        //...
    }
}

// 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
public class RiskManagement {
    private ASensitiveWordsFilter aFilter = new ASensitiveWordsFilter();
    private BSensitiveWordsFilter bFilter = new BSensitiveWordsFilter();
    private CSensitiveWordsFilter cFilter = new CSensitiveWordsFilter();

    public String filterSensitiveWords(String text) {
        String maskedText = aFilter.filterSexyWords(text);
        maskedText = aFilter.filterPoliticalWords(maskedText);
        maskedText = bFilter.filter(maskedText);
        maskedText = cFilter.filter(maskedText, "***");
        return maskedText;
    }
}

// 使用适配器模式进行改造
public interface ISensitiveWordsFilter { // 统一接口定义
    String filter(String text);
}

public class ASensitiveWordsFilterAdaptor implements ISensitiveWordsFilter {
    private ASensitiveWordsFilter aFilter;
    public String filter(String text) {
        String maskedText = aFilter.filterSexyWords(text);
        maskedText = aFilter.filterPoliticalWords(maskedText);
        return maskedText;
    }
}

//...省略BSensitiveWordsFilterAdaptor、CSensitiveWordsFilterAdaptor...

// 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，
// 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。
public class RiskManagement {
    private List<ISensitiveWordsFilter> filters = new ArrayList<>();

    public void addSensitiveWordsFilter(ISensitiveWordsFilter filter) {
        filters.add(filter);
    }

    public String filterSensitiveWords(String text) {
        String maskedText = text;
        for (ISensitiveWordsFilter filter : filters) {
            maskedText = filter.filter(maskedText);
        }
    }
}

```

```
}  
    return maskedText;  
}  
}
```

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：

```
// 外部系统A  
public interface IA {  
    //...  
    void fa();  
}  
  
public class A implements IA {  
    //...  
    public void fa() { //... }  
}  
  
// 在我们的项目中，外部系统A的使用示例  
public class Demo {  
    private IA a;  
    public Demo(IA a) {  
        this.a = a;  
    }  
    //...  
}  
  
Demo d = new Demo(new A());  
  
// 将外部系统A替换成外部系统B  
public class BAdaptor implements IA {  
    private B b;  
    public BAdaptor(B b) {  
        this.b = b;  
    }  
    public void fa() {  
        //...  
        b.fb();  
    }  
}  
  
// 借助BAdaptor，Demo的代码中，调用IA接口的地方都无需改动，  
// 只需要将BAdaptor如下注入到Demo即可。  
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Itertor。代码示例如下所示：

```
public class Collections {
    public static Enumeration enumeration(final Collection c) {
        return new Enumeration() {
            Iterator i = c.iterator();

            public boolean hasMoreElements() {
                return i.hasNext();
            }

            public Object nextElement() {
                return i.next();
            }
        }
    }
}
```

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。

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

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、erro……）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：

```
// slf4j统一的接口定义
package org.slf4j;

public interface Logger {
    public boolean isTraceEnabled();
    public void trace(String msg);
    public void trace(String format, Object arg);
    public void trace(String format, Object arg1, Object arg2);
    public void trace(String format, Object[] argArray);
    public void trace(String msg, Throwable t);

    public boolean isDebugEnabled();
    public void debug(String msg);
    public void debug(String format, Object arg);
    public void debug(String format, Object arg1, Object arg2);
    public void debug(String format, Object[] argArray);
    public void debug(String msg, Throwable t);

    //...省略info、warn、error等一堆接口
}

// log4j日志框架的适配器
// Log4jLoggerAdapter实现了LocationAwareLogger接口。
// 其中LocationAwareLogger继承自Logger接口，
// 也就相当于Log4jLoggerAdapter实现了Logger接口。
package org.slf4j.impl;

public final class Log4jLoggerAdapter extends MarkerIgnoringBase
    implements LocationAwareLogger, Serializable {
    final transient org.apache.log4j.Logger logger; // log4j

    public boolean isDebugEnabled() {
        return logger.isDebugEnabled();
    }

    public void debug(String msg) {
        logger.log(FQCN, Level.DEBUG, msg, null);
    }

    public void debug(String format, Object arg) {
        if (logger.isDebugEnabled()) {
            FormattingTuple ft = MessageFormatter.format(format, arg);
            logger.log(FQCN, Level.DEBUG, ft.getMessage(), ft.getThrowable());
        }
    }

    public void debug(String format, Object arg1, Object arg2) {
```

```

    if (logger.isDebugEnabled()) {
        FormattingTuple ft = MessageFormatter.format(format, arg1, arg2);
        logger.log(FQCN, Level.DEBUG, ft.getMessage(), ft.getThrowable());
    }
}

public void debug(String format, Object[] argArray) {
    if (logger.isDebugEnabled()) {
        FormattingTuple ft = MessageFormatter.arrayFormat(format, argArray);
        logger.log(FQCN, Level.DEBUG, ft.getMessage(), ft.getThrowable());
    }
}

public void debug(String msg, Throwable t) {
    logger.log(FQCN, Level.DEBUG, msg, t);
}
//...省略一堆接口的实现...
}

```

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。

精选留言：

- javaadu 2020-02-28 06:09:09
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 [3赞]

- 小晏子 2020-02-28 08:50:58
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，通过组合来替代继承，在设计的时候，装饰器类需要跟原始类继承相同的抽象类或者接口。所以装饰器只有对象装饰器这一种。
[1赞]

- 勤劳的明酱 2020-02-28 08:27:20
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。 [1赞]

- 守拙 2020-02-28 10:09:25
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？

代理模式可以使用类代理模式的方式实现. 考虑到代理模式封装对象及控制访问的职责, 类代理模式并不能很好的适用.

装饰器模式可以使用类装饰器模式方式实现. 但会导致继承层次不可控制的严重缺陷. 极度不推荐使用此实现方式. 基于对象装饰器模式的实现可以多个装饰器间互相包裹, 这是类装饰器模式无法实现的.

- 岁月神偷 2020-02-28 08:11:56

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 08:01:20

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。

2、装饰器模式，拓展原功能的某一侧面，如缓存。这里侧面就是某几个方法，增强某几个方法的某些侧面，其余方法不变，所以用继承的方式即使用类适配器比较方便。用对象适配器，即组合方式也可以，但是比较麻烦。

- Andy_Ron 2020-02-28 07:53:34

好像有点明白了，再看一遍🤔

- Jeff.Smile 2020-02-28 07:17:44

适配器的五种使用场景:

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

- 旭东 2020-02-28 06:51:19

替换外部系统是不是不继承外部接口更好，这样和外部的耦合度更低些，个人认为

- Yayu 2020-02-28 00:45:06

请问老师，如果一门语言里没有继承特性，是不是只能用对象适配器模式了？