# https://dss2.bdstatic.com/70cFvnSh_Q1YnxGkpoWK1HF6hhy/it/u=2626760717,1159921095&fm=26&gp=0.jpg适配器模式

## 问题引入

生活中经常会出现两个对象因接口不兼容而不能一起工作的实例。在软件设计中也会出现这样的情况：组件库中某个组件能够满足我们当前的使用需求，但是因为它们与系统当前的接口不兼容所以不能直接使用，如果重新开发成本又太高，这时可以使用适配器模式解决这些问题。

## 定义和特点

适配器模式（Adapter Pattern）的定义如下：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

适配器模式的主要**优点**如下。

1）客户端通过适配器可以透明地调用目标接口，增加了类的透明度。

2）复用了现存的类，适配器模式实际上拓展了原有类的功能，遵循了开闭原则。

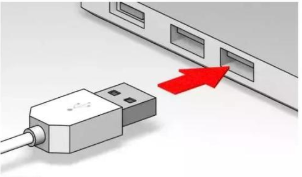
3）将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。

适配器模式的主要**缺点**如下：

1. 过多地使用适配器，会使系统变得非常零乱，不易整体进行把握。其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

## 模式的结构与实现

**类适配器模式**可采用多重继承方式实现，如 [C++](http://c.biancheng.net/cplus/) 可定义一个适配器类来同时继承当前系统的业务接口和现有组件库中已经存在的组件接口；[Java](http://c.biancheng.net/java/)不支持多继承，但可以定义一个适配器类来实现当前系统的业务接口，同时又继承现有组件库中已经存在的组件。

**对象适配器模式**可釆用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口。（有点像静态代理模式）

适配器模式（Adapter）包含以下主要角色：

* 目标接口（Target）：当前系统业务所期待的接口，它可以是抽象类或接口。
* 适配者类（Adaptee）：它是被访问和适配的现存组件库中的组件接口。
* 适配器类（Adapter）：它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。

假设现有一台笔记本，没有网线插槽，只有USB插槽，现在想要使用网线上网。在此问题中，USB插头就是目标接口，网线插头就是适配者类，网线USB转接器就是适配器类。

类适配器模式的结构图如图：



对象适配器模式的结构图如图：



## 实例

|  |  |
| --- | --- |
| */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-17 09:33  \* 说明：网线传输的数据  \*/* public class NetData *{* private String data;   public NetData*(*String data*) {* this.data = data;  *}* public String getData*() {* return data;  *}* @Override  public String toString*() {* return "NetData{" +  "data='" + data + '\'' +  '}';  *} }* | */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-17 09:33  \* 说明：USB传输的数据  \*/* public class UsbData *{* private String data;   public UsbData*(*String data*) {* this.data = data;  *}* public String getData*() {* return data;  *}* @Override  public String toString*() {* return "UsbData{" +  "data='" + data + '\'' +  '}';  *} }* |

|  |  |
| --- | --- |
| */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-17 09:40  \* 说明：网线  \*/* public interface NetLink *{  /\*\*  \* 通过网线插头传输数据  \*  \** ***@param*** *netDataImpl - 传输的数据  \*/* void netLinkPlug*(*NetData netDataImpl*)*; *}* | */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-17 09:41  \* 说明：USB线  \*/* public interface UsbLink *{  /\*\*  \* 通过USB接头传输数据  \*  \** ***@param*** *usbData 传输的数据  \*/* void usbLinkPlug*(*UsbData usbData*)*; *}* |

|  |  |
| --- | --- |
| public class NetLinkImpl implements NetLink *{* @Override  public void netLinkPlug*(*NetData netDataImpl*) {* System.*out*.println*(*"使用网线接口传输数据" + netDataImpl*)*;  *} }* | public class UsbLinkImpl implements UsbLink *{* @Override  public void usbLinkPlug*(*UsbData usbData*) {* System.*out*.println*(*"通过USB接口传输数据" + usbData*)*;  *} }* |

|  |  |
| --- | --- |
| */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-17 09:37  \* 说明：USB网线转接口（类适配器模式）  \*/* public class Usb2NetAdaptor extends NetLinkImpl implements UsbLink *{*  */\*\*  \* USB接口  \*  \** ***@param*** *usbData 通过USB接口传输的数据  \*/* @Override  public void usbLinkPlug*(*UsbData usbData*) {* this.netLinkPlug*(*usbData2NetData*(*usbData*))*;  *}   /\*\*  \* 通过USB接口传输的数据转换为网线能够传输的数据  \*  \** ***@param*** *usbData -通过USB接口传输的数据  \** ***@return*** *- 网线能够传输的数据  \*/* private NetData usbData2NetData*(*UsbData usbData*) {* return new NetData*(*usbData.getData*())*;  *}* public static void main*(*String*[]* args*) {* UsbData usbData = new UsbData*(*"hello"*)*;  Usb2NetAdaptor usb2NetAdaptor = new Usb2NetAdaptor*()*;  usb2NetAdaptor.usbLinkPlug*(*usbData*)*;  *} }* | */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-17 09:52  \* 说明：USB网线转接口（对象适配器模式）  \*/* public class Usb2NetAdaptor2 implements UsbLink *{* private NetLink netLink;   public Usb2NetAdaptor2*(*NetLink netLink*) {* this.netLink = netLink;  *}   /\*\*  \* USB接口  \*  \** ***@param*** *usbData 通过USB接口传输的数据  \*/* @Override  public void usbLinkPlug*(*UsbData usbData*) {* this.netLink.netLinkPlug*(*usbData2NetData*(*usbData*))*;  *}   /\*\*  \* 通过USB接口传输的数据转换为网线能够传输的数据  \*  \** ***@param*** *usbData -通过USB接口传输的数据  \** ***@return*** *- 网线能够传输的数据  \*/* private NetData usbData2NetData*(*UsbData usbData*) {* return new NetData*(*usbData.getData*())*;  *}* public static void main*(*String*[]* args*) {* UsbData usbData = new UsbData*(*"hello"*)*;  Usb2NetAdaptor usb2NetAdaptor = new Usb2NetAdaptor*()*;  usb2NetAdaptor.usbLinkPlug*(*usbData*)*;  *} }* |

## 应用场景

适配器模式（Adapter）通常适用于以下场景。

* 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
* 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

## 模式拓展

适配器模式（Adapter）可扩展为双向适配器模式，双向适配器类既可以把适配者接口转换成目标接口，也可以把目标接口转换成适配者接口，其结构图如图：

