第3章 适配器模式

提出问题

> 问题描述：访问数据库，抽取并处理数据后以特定格式返回给用户。假设现有系统已有基础数据工具类 `DataTool` ，提供 `processData()` 方法用于从数据库中提取数据，并返回基础数据对象。如果增加一个新需求，要求提供返回JSON格式数据的功能，应该如何扩展程序。

```
public class DataTool implements IDataProcess{
    public Data processData(DB db){
        Data dd = new DictData();
        // 从数据库中抽取和处理数据，返回基础数据对象
        return dd;
    }
}
public interface IDataProcess {
    public Data processData(DB db);
}
```

扩展一个功能，最直接的方式就是设计实现一个新工具类，专门用于从数据库提取和处理数据，并返回JSON格式数据。

```
public class NewDataTool implements INewDataProcess{
    public JSONData processData(DB db){
        JSONData jd = new JSONData();
        // 从数据库中抽取和处理数据，返回JSON数据
        return jd;
    }
}
public interface INewDataProcess {
    public JSONData processData(DB db);
}
```

这种设计方式代码复用性差，数据库连接、提取和处理代码重复。引出问题：当接口不兼容时，原系统的代码怎么在新系统中得到复用。

模式名称

适配器模式（Adapter）或包装器模式（Wrapper）

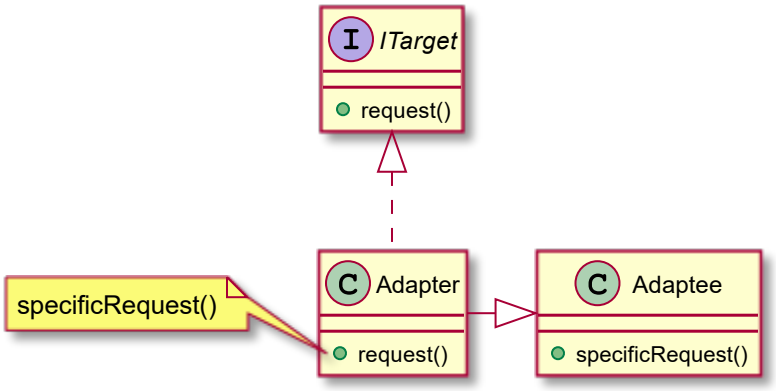
设计意图

适配去模式功能是将一个类的接口转换为客户端期望的另一种接口，使因接口不兼容的类也能在一起工作。

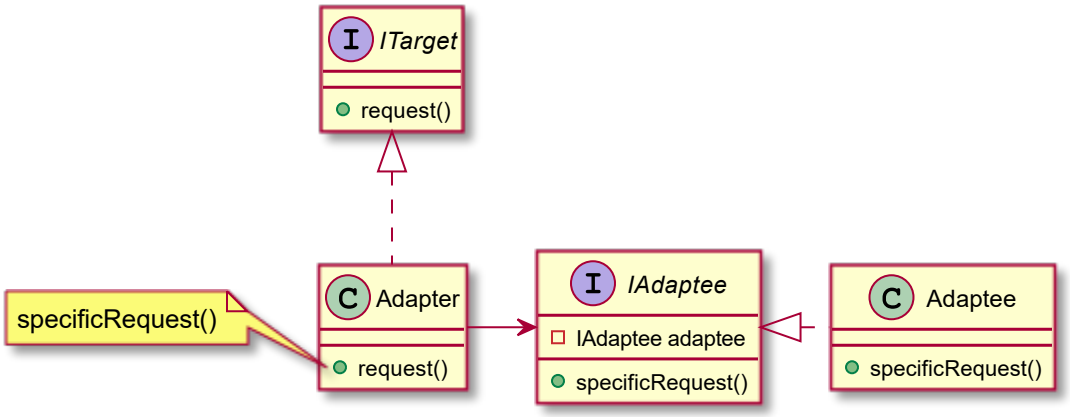
> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

设计结构

类适配器采用继承方式实现



对象适配器采用对象组合方式实现



适配器模式参与者：

- 目标角色（Target）：客户端需要的接口。
- 适配器（Adapter）：实现原有接口到目标接口的适配。
- 源角色（Adaptee）：需要被适配的接口。

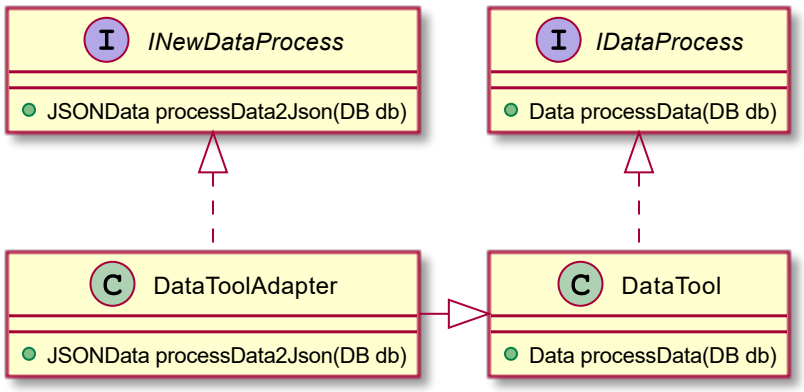
类适配器的局限性：（1）在需要适配多个类时，需要多继承的支持。（2）耦合度高，降低系统的可扩展性。

解决问题

针对前述问题，设计一个类适配器，通过继承的方式复用`DataTool`类中的`processData()`方法代码，只需要增量实现`Data`对象到`JSONData`对象的转换代码。

```
public class DataToolAdapter extends DataTool implements INewDataProcess {
    public JsonData processData2Json(DB db){
        JSONData jd = new JSONData();
        Data dd = processData(db); // 代码复用
        // 将字典数据转换为JSON数据，少量代码
        return jd;
    }
}
public interface INewDataProcess {
    public JSONData processData2Json(DB db);
}
```

类图如下：



假设现存提取文本数据和提取图片数据的工具类，要求创建输出PDF文档（文本+图片）的工具类时，上述适配器类无法通过继承复用两个类的代码，这时需要采用对象适配器来实现。

现有的工具类如下：

```
public interface IDataProcess2Text {
    public TextData process2Text(DB db);
}
public class TextTool implements IDataProcess2Text{
    public TextData process2Text(DB db) {
        TextData td = new TextData();
        // 数据库中提取数据返回文本
        return td;
    }
}

public interface IDataProcess2Image {
    public ImageData process2Image(DB db);
}
public class ImageTool implements IDataProcess2Image{
    public ImageData process2Image(DB db) {
        ImageData img = new ImageData();
        // 数据库中提取数据返回图片
        return img;
    }
}
```

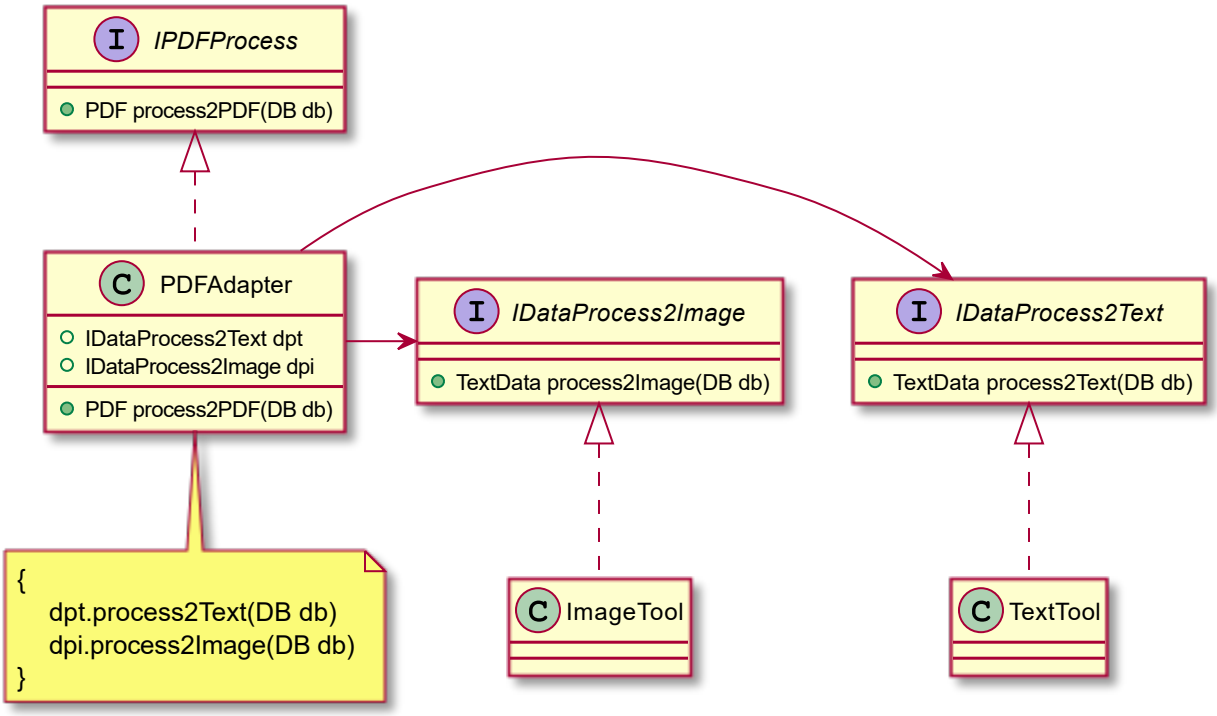
目标接口：

```
public interface IPDFProcess {
    public PDF process2PDF(DB db);
}
```

对象适配器实现PDF文档输出：

```
public class PDFAdapter implements IPDFProcess{
    private IDataProcess2Image dpi;
    private IDataProcess2Text dpt;

    public PDF process2PDF(DB db) {
        PDF pdf = new PDF();
        ImageData img = this.dpi.process2Image();
        TextData td = this.dpt.process2Text();
        // 将文本和图像组合为PDF文档
        return pdf;
    }
}
```

类图结构：



效果与适用性

优点

- 提高设计的复用性和可扩展性。
  缺点
- 增加系统复杂性，降低代码可读性。

适用性：

- 现有类接口不匹配，但又需要使用；
- 适配器模式常用于系统维护阶段。

扩展案例

充电适配器

> 标准插座（StandardSocket）提供220V交流电，不能直接给手机充电，通过适配器（

> SocketAdapter ）提供直流5V直流电，充电接口一般为USB （ IUSBSocket ）。（采用对象适配器设计）
> 标准电流接口及其实现

```
public class StandardSocket implements IStandardSocket {
    public StandardElectricity supply() {
        StandardElectricity se = new StandardElectricity();
        return se;
    }
}
public interface IStandardSocket {
    public StandardElectricity supply();
}
```

目标电流接口

```
public interface IUSBSocket {
    public USBElectricity supply();
}
```

适配器类

```
public class SocketAdapter implements IUSBSocket {
    private IStandardSocket ss;
    public USBElectricity supply() {
        USBElectricity usbe = null;
        StandardElectricity se = ss.supply();
        //将标准电流转换为直流电
        return usbe;
    }
}
```

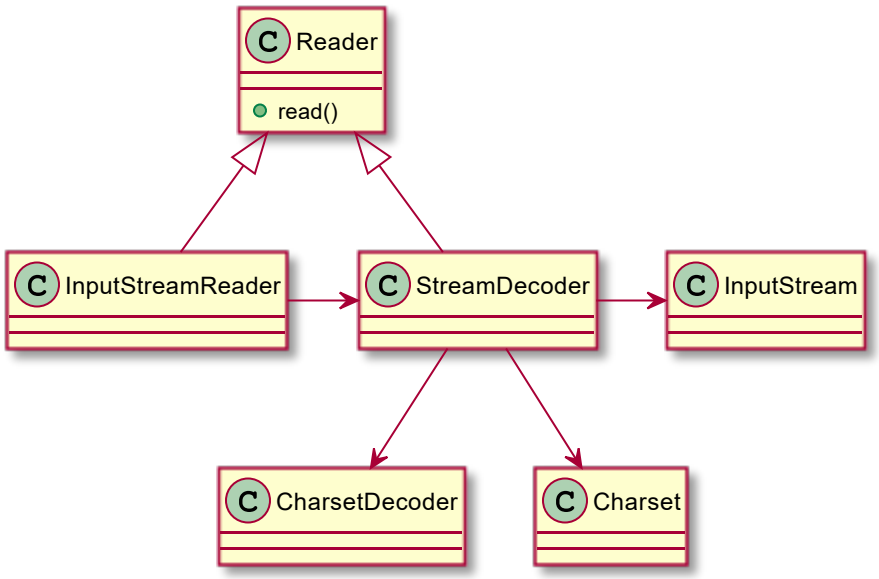
字符读取（InputStreamReader)

输入流类`InputStream`只能按字节读取，而对文本进行操作需要按字符读取。JDK中面向字符读取操作定义了抽象类`Reader`以及`read(...)`方法。具体类`InputStreamReader`的实现依赖于`CharsetDecoder`类的编码功能以及`InputStream`类的字节读取功能。这里看到`InputStreamReader`类与其他依赖类之间存在一个`StreamDecoder`类，这个类在`sun.nio`包，不属于公开的标准接口，不去管他，可以理解为`InputStreamReader`通过`StreamDecoder`整合`InputStream`、`CharsetDecoder`等依赖类的功能。因此，`InputStreamReader`类可以看作一个

适配器，将依赖类中的代码复用实现字符读取功能`read()`。

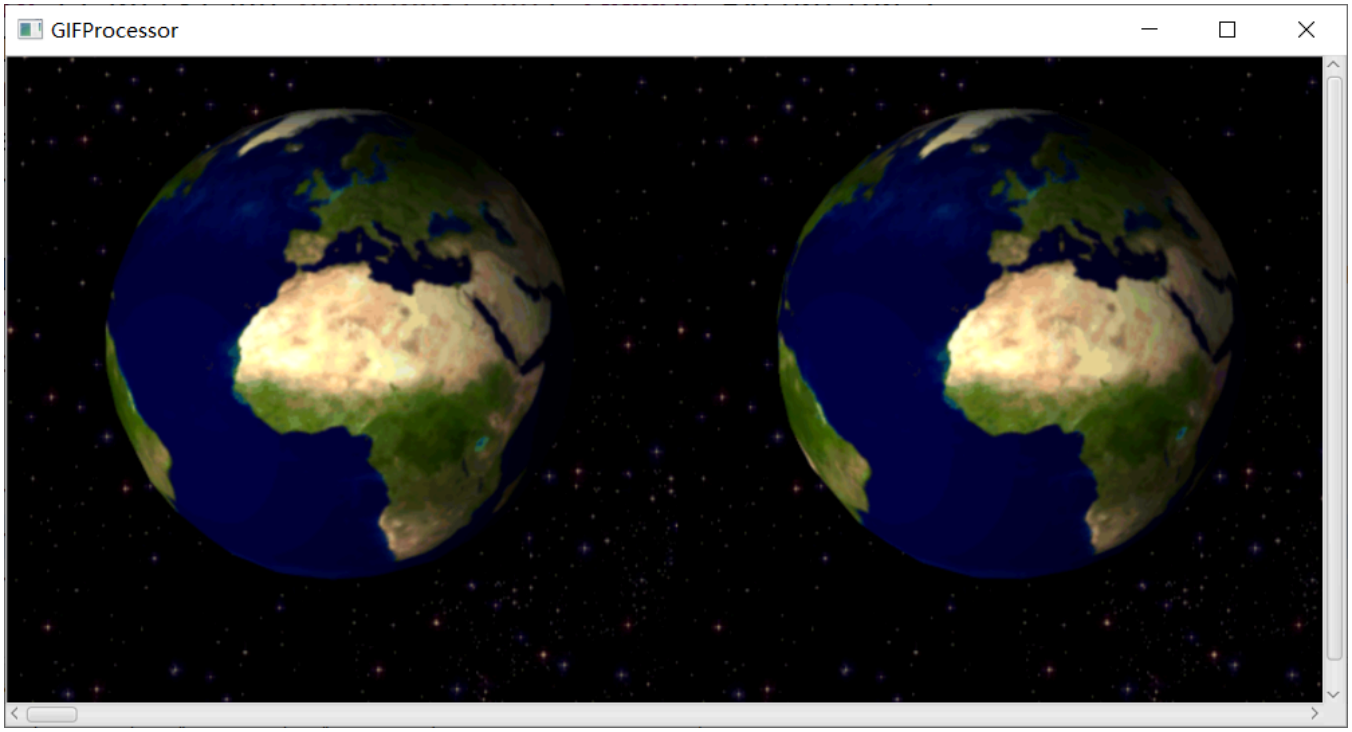
```
public abstract class Reader implements Readable, Closeable {
    public abstract int read(char cbuf[], int off, int len) throws IOException
}

// 这里的forInputStreamReader()是静态工具函数，用于创建一个StreamDecoder对象。
public class InputStreamReader extends Reader {
    private final StreamDecoder sd;
    public InputStreamReader(InputStream in) {
        super(in);
        sd = StreamDecoder.forInputStreamReader(in, this,
                Charset.defaultCharset());
    }
    public int read(char cbuf[], int offset, int length) throws IOException {
        return sd.read(cbuf, offset, length);
    }
}
```

类图结构如下



动图处理

GIF（Graphics Interchange Format）是常用的互联网图片格式，特别是GIF89a标准使该图片格式支持延时动画（简称动图）。对动图进行复杂编辑需要对其进行解码，针对每一张图片进行编辑。例如，我们需要对一张动图进行解码，并在GUI程序中显示（如下图）。



根据需求，这里设计一个接口将GIF文件中的图片全部读取出来：

```
public interface IGIFProcessor {
    public Image[] extract(String gifPath);
}
```

现要实现这个接口是比较困难的，需要完全掌握GIF格式标准，好在已经有人完成了这个工作，只不过当时的类设计上不符合现在的需求。因此，可以采用适配器模式来设计我们自己的类。

```
public class GIFProcessor implements IGIFProcessor {
    private GifDecoder gdc;

    public GIFProcessor() {
        gdc = new GifDecoder();
    }

    @Override
    public Image[] extract(String gifPath) {
        gdc.read(gifPath);
        int n = gdc.getFrameCount();
        Image[] images = new Image[n];
        for (int i = 0; i < n; i++) {
            BufferedImage frame = gdc.getFrame(i);
            images[i] = SwingFXUtils.toFXImage(frame, null);
        }
        return images;
    }
}
```

上述代码采用对象适配器，`GifDecoder`是被适配的对象，由`Kevin Weiner`开发于2003年。继续完成主程序：

```
public class MainApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        ScrollPane pane = new ScrollPane();
        HBox fp = new HBox();
        pane.setContent(fp);
        IGIFProcessor gifp = new GIFProcessor();

        Image[] images = gifp.extract(Paths.get("earth.gif").toString());
        for(int i = 0; i < images.length; i++) {
            fp.getChildren().add(new ImageView(images[i]));
        }
        Scene scene = new Scene(pane, 800, 400);
        primaryStage.setScene(scene);
        primaryStage.setTitle("GIFProcessor");
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

运行程序得到`earth.gif`动图下的每一帧图像。程序类图结构为：