适配器模式

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

我们通过下面的实例来演示适配器模式的使用。其中，音频播放器设备只能播放 mp3 文件，通过使用一个更高级的音频播放器来播放 vlc 和 mp4 文件。

介绍

意图：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决：主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

何时使用： 1、系统需要使用现有的类，而此类的接口不符合系统的需要。 2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。 3、通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

如何解决：继承或依赖（推荐）。

关键代码：适配器继承或依赖已有的对象，实现想要的目标接口。

应用实例： 1、美国电器 110V，中国 220V，就要有一个适配器将 110V 转化为 220V。 2、JAVA JDK 1.1 提供了 Enumeration 接口，而在 1.2 中提供了 Iterator 接口，想要使用 1.2 的 JDK，则要将以前系统的 Enumeration 接口转化为 Iterator 接口，这时就需要适配器模式。 3、在 LINUX 上运行 WINDOWS 程序。 4、JAVA 中的 jdbc。

优点： 1、可以让任何两个没有关联的类一起运行。 2、提高了类的复用。 3、增加了类的透明度。 4、灵活性好。

缺点： 1、过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，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。 2.由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

使用场景：有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。

注意事项：适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。

实现

我们有一个 *MediaPlayer* 接口和一个实现了 *MediaPlayer* 接口的实体类 *AudioPlayer*。默认情况下，*AudioPlayer* 可以播放 mp3 格式的音频文件。

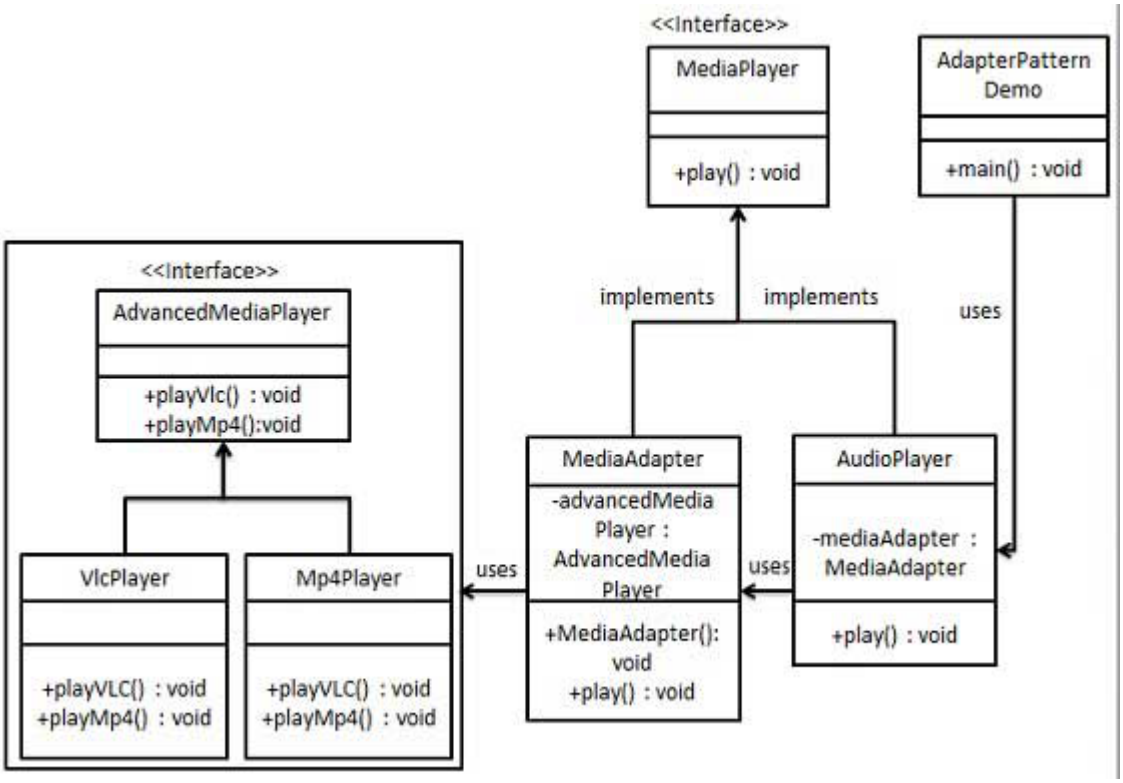
我们还有另一个接口 *AdvancedMediaPlayer* 和实现了 *AdvancedMediaPlayer* 接口的实体类。该类可以播放 vlc 和 mp4 格式的文

反馈

件。

我们想要让 *AudioPlayer* 播放其他格式的音频文件。为了实现这个功能，我们需要创建一个实现了 *MediaPlayer* 接口的适配器类 *MediaAdapter*，并使用 *AdvancedMediaPlayer* 对象来播放所需的格式。

AudioPlayer 使用适配器类 *MediaAdapter* 传递所需的音频类型，不需要知道能播放所需格式音频的实际类。*AdapterPatternDemo*，我们的演示类使用 *AudioPlayer* 类来播放各种格式。



步骤 1

为媒体播放器和更高级的媒体播放器创建接口。

MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

步骤 2

创建实现了 *AdvancedMediaPlayer* 接口的实体类。

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }
```

```
   @Override
   public void playMp4(String fileName) {
      //什么也不做
   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //什么也不做
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

步骤 **3**

创建实现了 *MediaPlayer* 接口的适配器类。

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      } else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

步骤 **4**

创建实现了 *MediaPlayer* 接口的实体类。

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;
```

```
   @Override
   public void play(String audioType, String fileName) {

      //播放 mp3 音乐文件的内置支持
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: "+ fileName);
      }
      //mediaAdapter 提供了播放其他文件格式的支持
      else if(audioType.equalsIgnoreCase("vlc")
         || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }
      else{
         System.out.println("Invalid media. "+
            audioType + " format not supported");
      }
   }
}
```

步骤 5

使用 AudioPlayer 来播放不同类型的音频格式。

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

步骤 6

执行程序，输出结果：

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

□ 原型模式　　桥接模式 □

1、先创建一个SD卡的接口：

```
public interface SDCard {
    //读取SD卡方法
    String readSD();
    //写入SD卡功能
    int writeSD(String msg);
}
```

2、创建SD卡接口的实现类，模拟SD卡的功能：

```
public class SDCardImpl implements SDCard {
    @Override
    public String readSD() {
        String msg = "sdcard read a msg :hello word SD";
        return msg;
    }
    @Override
    public int writeSD(String msg) {
        System.out.println("sd card write msg : " + msg);
        return 1;
    }
}
```

3、创建计算机接口，计算机提供读取SD卡方法：

```
public interface Computer {
    String readSD(SDCard sdCard);
}
```

4、创建一个计算机实例，实现计算机接口，并实现其读取SD卡方法：

```
public class ThinkpadComputer implements Computer {
    @Override
    public String readSD(SDCard sdCard) {
        if(sdCard == null)throw new NullPointerException("sd card null");
        return sdCard.readSD();
    }
}
```

5、这时候就可以模拟计算机读取SD卡功能：

```
public class ComputerReadDemo {
    public static void main(String[] args) {
        Computer computer = new ThinkpadComputer();
        SDCard sdCard = new SDCardImpl();
        System.out.println(computer.readSD(sdCard));
    }
}
```

二、接下来在不改变计算机读取**SD**卡接口的情况下，通过适配器模式读取**TF**卡：

1、创建TF卡接口：

```
public interface TFCard {
    String readTF();
```

```
    int writeTF(String msg);
}
```

2、创建TF卡实例：

```
public class TFCardImpl implements TFCard {
    @Override
    public String readTF() {
        String msg ="tf card reade msg : hello word tf card";
        return msg;
    }
    @Override
    public int writeTF(String msg) {
        System.out.println("tf card write a msg : " + msg);
        return 1;
    }
}
```

3、创建SD适配TF （也可以说是SD兼容TF，相当于读卡器）：

实现SDCard接口，并将要适配的对象作为适配器的属性引入。

```
public class SDAdapterTF implements SDCard {
    private TFCard tfCard;
    public SDAdapterTF(TFCard tfCard) {
        this.tfCard = tfCard;
    }
    @Override
    public String readSD() {
        System.out.println("adapter read tf card ");
        return tfCard.readTF();
    }
    @Override
    public int writeSD(String msg) {
        System.out.println("adapter write tf card");
        return tfCard.writeTF(msg);
    }
}
```

4、通过上面的例子测试计算机通过SD读卡器读取TF卡：

```
public class ComputerReadDemo {
    public static void main(String[] args) {
        Computer computer = new ThinkpadComputer();
        SDCard sdCard = new SDCardImpl();
        System.out.println(computer.readSD(sdCard));
        System.out.println("====================================");
        TFCard tfCard = new TFCardImpl();
        SDCard tfCardAdapterSD = new SDAdapterTF(tfCard);
        System.out.println(computer.readSD(tfCardAdapterSD));
    }
}
```

输出：

```
sdcard read a msg :hello word SD
==================================
adapter read tf card
tf card reade msg : hello word tf card
```

在这种模式下，计算机并不需要知道具体是什么卡，只需要负责操作接口即可，具体操作的什么类，由适配器决定。