实现要点

1．Adapter 模式主要应用于“希望复用一些现存的类，但是接口又与复用环境要求不一致的情况”，在遗留代码复用、类库迁移等方面非常有用。

2．Adapter 模式有对象适配器和类适配器两种形式的实现结构，但是类适配器采用“多继承”的实现方式，带来了不良的高耦合，所以一般不推荐使用。对象适配器采用“对象组合”的方式，更符合松耦合精神。

3．Adapter 模式的实现可以非常的灵活，不必拘泥于 GOF23 中定义的两种结构。例如，完全可以将 Adapter 模式中的“现存对象”作为新的接口方法参数，来达到适配的目的。

4．Adapter 模式本身要求我们尽可能地使用“面向接口的编程”风格，这样才能在后期很方便的适配。[以上几点引用自 MSDN WebCast]

效果

对于类适配器：

1．用一个具体的 Adapter 类对 Adaptee 和 Taget 进行匹配。结果是当我们想要匹配一个类以及所有它的子类时，类 Adapter 将不能胜任工作。

2．使得 Adapter 可以重定义 Adaptee 的部分行为，因为 Adapter 是 Adaptee 的一个子类。

3．仅仅引入了一个对象，并不需要额外的指针一间接得到 Adaptee.

对于对象适配器：

1．允许一个 Adapter 与多个 Adaptee，即 Adaptee 本身以及它的所有子类（如果有子类的话）同时工作。Adapter 也可以一次给所有的 Adaptee 添加功能。

2．使得重定义 Adaptee 的行为比较困难。这就需要生成 Adaptee 的子类并且使得 Adapter 引用这个子类而不是引用 Adaptee 本身。

适用性

在以下各种情况下使用适配器模式：

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。这些源类不一定有很复杂的接口。

3．（对对象适配器而言）在设计里，需要改变多个已有子类的接口，如果使用类的适配器模式，就要针对每一个子类做一个适配器，而这不太实际。

代码势力

适配器模式解说

我们还是以日志记录程序为例子说明 Adapter 模式。现在有这样一个场景：假设我们在软件开发中要使用一个第三方的日志记录工具，该日志记录工具支持数据库日志记录 DatabaseLog 和文本文件记录 FileLog 两种方式，它提供给我们的 API 接口是 Write()方法，使用方法如下：

```
Log.Write("Logging Message!");
```

当软件系统开发进行到一半时，处于某种原因不能继续使用该日志记录工具了，需要采用另外一个日志记录工具，它同样也支持数据库日志记录 DatabaseLog 和文本文件记录 FileLog 两种方式，只不过它提供给我们的 API 接口是 WriteLog()方法，使用方法如下：

```
Log.WriteLog("Logging Message!");
```

该日志记录工具的类结构图如下：

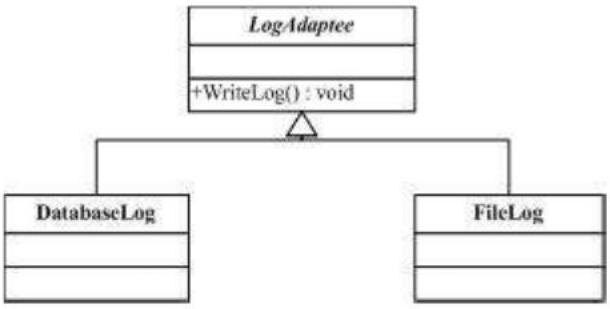

图 4 日志记录工具类结构图

它的实现代码如下：

```
public abstract class LogAdaptee
{
   public abstract void WriteLog();
}
public class DatabaseLog:LogAdaptee
{
   public override void WriteLog()
   {
      Console.WriteLine("Called WriteLog Method");
   }
}
public class FileLog:LogAdaptee
```

```
{
    public override void WriteLog()
    {
        Console.WriteLine("Called WriteLog Method");
    }
}
```

在我们开发完成的应用程序中日志记录接口中（不妨称之为ILogTarget 接口，在本例中为了更加清楚地说明，在命名上采用了 Adapter 模式中的相关角色名字），却用到了大量的Write()方法，程序已经全部通过了测试，我们不能去修改该接口。代码如下：

```
public interface ILogTarget
{
    void Write();
}
```

这时也许我们会想到修改现在的日志记录工具的 API 接口，但是由于版权等原因我们不能够修改它的源代码，此时 Adapter 模式便可以派上用场了。下面我们通过 Adapter 模式来使得该日志记录工具能够符合我们当前的需求。

前面说过，Adapter 模式有两种实现形式的实现结构，首先来看一下类适配器如何实现。现在唯一可行的办法就是在程序中引入新的类型，让它去继承 LogAdaptee 类，同时又实现已有的 ILogTarget 接口。由于 LogAdaptee 有两种类型的方式，自然我们要引入两个分别为 DatabaseLogAdapter 和 FileLogAdapter 的类。

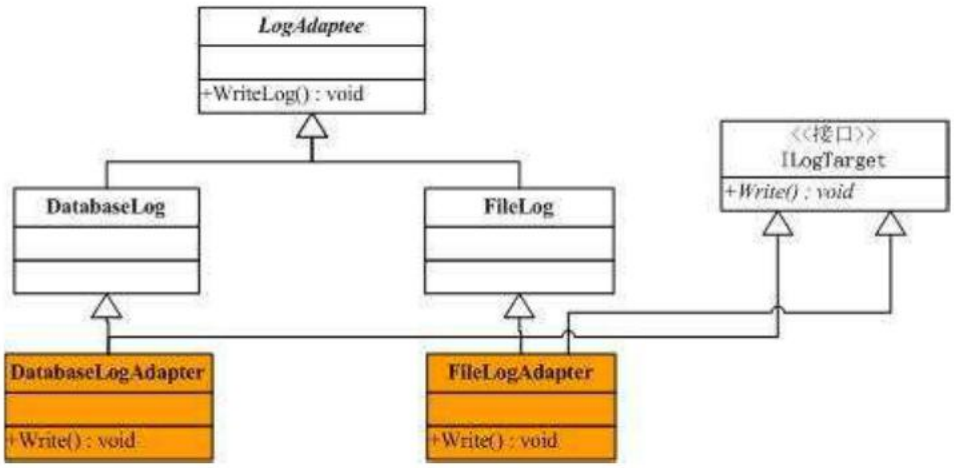

图 5 引入类适配器后的结构图

实现代码如下：

```
public class
DatabaseLogAdapter:DatabaseLog,ILogTarget
{
   public void Write()
   {
      WriteLog();
```

```
    }
}

public class FileLogAdapter:FileLog,ILogTarget
{
    public void Write()
    {
        this.WriteLog();
    }
}
```

这里需要注意的一点是我们为每一种日志记录方式都编写了它的适配类，那为什么不能为抽象类 LogAdaptee 来编写一个适配类呢？因为 DatabaseLog 和 FileLog 虽然同时继承于抽象类 LogAdaptee，但是它们具体的 WriteLog()方法的实现是不同的。只有继承于该具体类，才能保留其原有的行为。

我们看一下这时客户端的程序的调用方法：

```
public class App
{
```

```
    public static void Main()
    {
        ILogTarget dbLog = new DatabaseLogAdapter();
        dbLog.Write("Logging Database...");

        ILogTarget fileLog = new FileLogAdapter();
        fileLog.Write("Logging File...");
    }
}
```

下面看一下如何通过对象适配器的方式来达到我们适配的目的。对象适配器是采用对象组合而不是使用继承，类结构图如下：

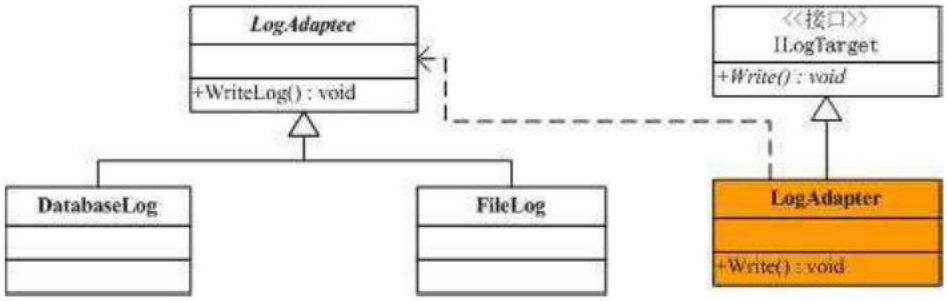

图 6 引入对象适配器后的结构图

实现代码如下：

```
public class LogAdapter:ILogTarget
```

```
{
    private LogAdaptee _adaptee;
    public LogAdapter(LogAdaptee adaptee)
    {
        this._adaptee = adaptee;
    }
    public void Write()
    {
        _adaptee.WriteLog();
    }
}
```

与类适配器相比较，可以看到最大的区别是适配器类的数量减少了，不再需要为每一种具体的日志记录方式来创建一个适配器类。同时可以看到，引入对象适配器后，适配器类不再依赖于具体的 DatabaseLog 类和 FileLog 类，更好的实现了松耦合。

再看一下客户端程序的调用方法：

```
public class App
{
```

```
    public static void Main()
    {

        ILogTarget dbLog = new LogAdapter(new
DatabaseLog());
        dbLog.Write("Logging Database...");

        ILogTarget fileLog = new LogAdapter(new
FileLog());
        fileLog.Write("Logging Database...");
    }
}
```

通过 Adapter 模式，我们很好的实现了对现有组件的复用。对比以上两种适配方式，可以总结出，在类适配方式中，我们得到的适配器类 DatabaseLogAdapter 和 FileLogAdapter 具有它所继承的父类的所有的行为，同时也具有接口 ILogTarget 的所有行为，这样其实是违背了面向对象设计原则中的类的单一职责原则，而对象适配器则更符合面向对象的精神，所以在实际应用中不太推荐类适配这种方式。再换个角度来看类适配方式，假设我们要适配出来的类在记录日

志时同时写入文件和数据库，那么用对象适配器我们会这样去写：

```
public class LogAdapter:ILogTarget
{
   private LogAdaptee _adaptee1;
   private LogAdaptee _adaptee2;

   public LogAdapter(LogAdaptee adaptee1,LogAdaptee adaptee2)
   {
      this._adaptee1 = adaptee1;
      this._adaptee2 = adaptee2;
   }
   public void Write()
   {
      _adaptee1.WriteLog();
      _adaptee2.WriteLog();
   }
```

```
}
```

如果改用类适配器，难道这样去写：

```
public class
DatabaseLogAdapter:DatabaseLog,FileLog,ILogTarget
{
   public void Write()
   {
      //WriteLog();
   }
}
```

显然是不对的，这样的解释虽说有些牵强，也足以说明一些问题，当然了并不是说类适配器在任何情况下都不使用，针对开发场景不同，某些时候还是可以用类适配器的方式。