相关主题 RECOMMEND ARTICLE
- 你必须知道的.NET之对接口和抽象类
- ASP.Net中利用CSS实现多界面两法
- 将ASP页面转换成HTM静态页面的方法
- ASP.NET技术获取IP与MAC地址的方法
- ASP.NET移动开发之SelectionList控件
- ASP.NET中为GridView添加删除提示框
- 理解ASP.NET与客户端缓存之HTTP协议
- ASP.NET中Session的状态保持方式浅议

MORE

推荐文章 RECOMMEND ARTICLE
- 游戏音乐制作案例之《战火 红色警戒》音效制作揭秘
- 英雄连Online 原画
- 游戏音乐制作案例之《乱武天下》
- 游戏音乐制作案例之《诛仙》
- 《鹿鼎记》最新原画
- MIDP2.1规范的新特性
- 3D游戏编程入门经典(6)
- Introduction to 3d game engine design using directx-9 and c#(10)

MORE

热门文章 HOT ARTICLE
- [电子书下载]游戏设计 — 原理与实践
- [电子书下载]网络游戏开发
- 游戏设计全过程
- [电子书下载]游戏设计技术
- [电子书下载]游戏设计理论
- CS游戏人物模型制作教程
- CG人物插画基本流程
- [转贴]MAX高级人头教程

MORE

您的位置： .NET

文章标题	.NET下的设计模式研究之桥接模式		
来源：	[ogdev]	浏览：	[592]

概述

在软件系统中，某些类型由于自身的逻辑，它具有两个或多个维度的变化，那么如何应对这种“多维度的变化”?如何利用面向对象的技术来使得该类型能够轻松的沿着多个方向进行变化，而又不引入额外的复杂度?这就要使用Bridge模式。

意图

将抽象部分与实现部分分离，使它们都可以独立的变化。[GOF 《设计模式》]

结构图

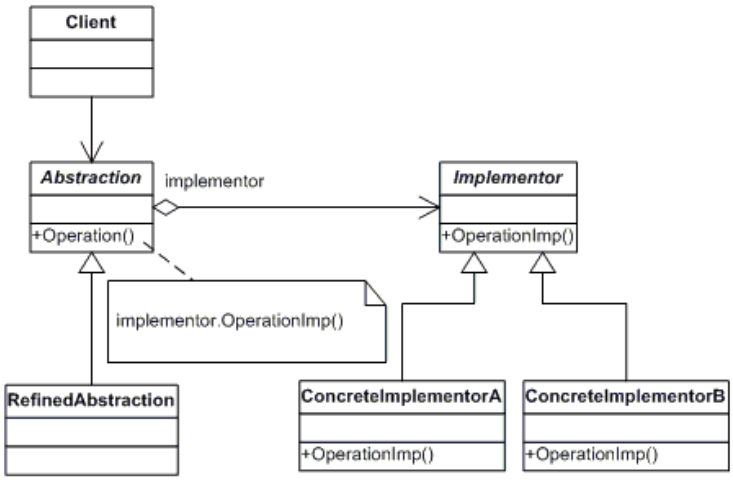


图1 Bridge模式结构图

生活中的例子

桥接模式将抽象部分与它的实现分离，使它们能够独立地变化。一个普通的开关控制的电灯、电风扇等等，都是桥接的例子。开关的目的是将设备打开或关闭。实际的开关可以是简单的双刀拉链开关，也可以是调光开关。

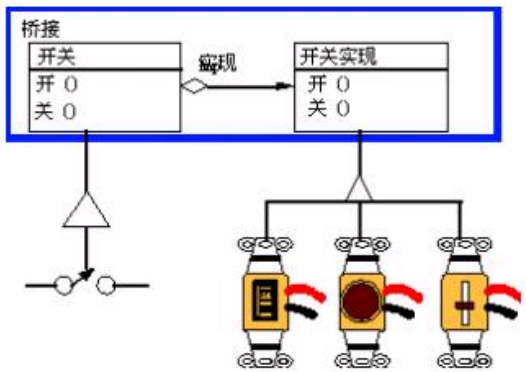


图2 使用电子开关例子的桥接对象图

桥接模式解说

在创建型模式里面，我曾经提到过抽象与实现，抽象不应该依赖于具体实现细节，实现细节应该依赖于抽象。看下面这幅图：

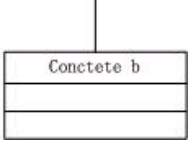


图3 抽象不应该依赖于实现细节

在这种情况下，如果抽象B稳定，而实现细节b变化，这时用创建型模式来解决没有问题。但是如果抽象B也不稳定，也是变化的，该如何解决?这就要用到Bridge模式了。

我们仍然用日志记录工具这个例子来说明Bridge模式。现在我们要开发一个通用的日志记录工具，它支持数据库记录DatabaseLog和文本文件记录FileLog两种方式，同时它既可以运行在.NET平台，也可以运行在Java平台上。

根据我们的设计经验，应该把不同的日志记录方式分别作为单独的对象来对待，并为日志记录类抽象出一个基类Log出来，各种不同的日志记录方式都继承于该基类:

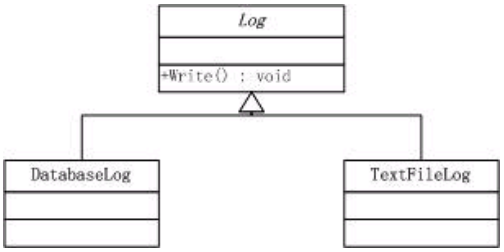


图4 Log类结构图

实现代码如下:

```
public abstract class Log
{
  public abstract void Write(string log);
}
public class DatabaseLog : Log
{
  public override void Write(string log)
  {
    //......Log Database
  }
}

public class TextFileLog : Log
{
  public override void Write(string log)
  {
    //......Log Text File
  }
}
```

另外考虑到不同平台的日志记录，对于操作数据库、写入文本文件所调用的方式可能是不一样的，为此对于不同的日志记录方式，我们需要提供各种不同平台上的实现，对上面的类做进一步的设计得到了下面的结构图:

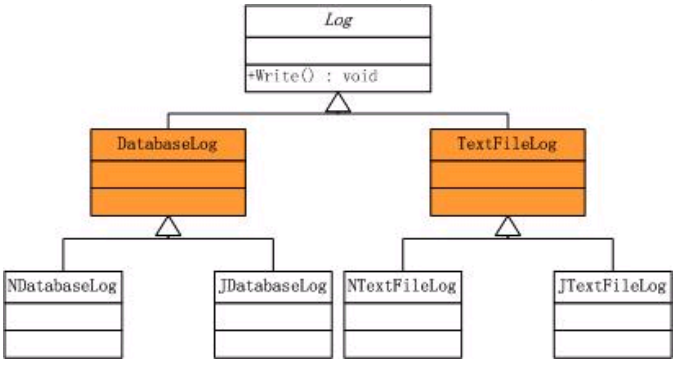


图5

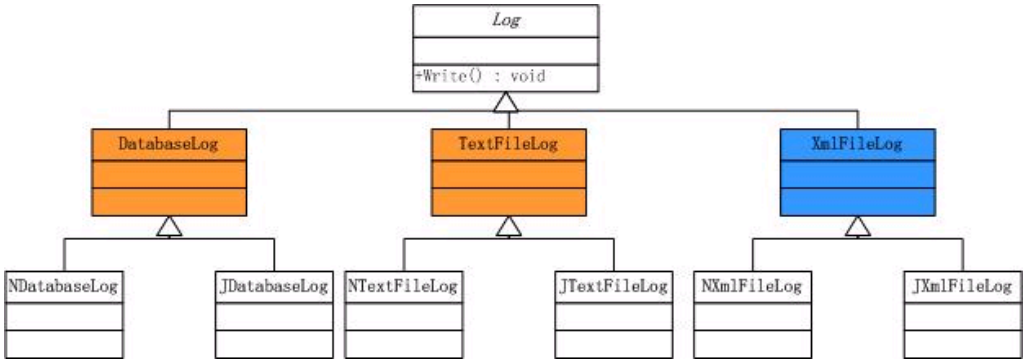
实现代码如下:

```
public class NDatabaseLog : DatabaseLog
{
  public override void Write(string log)
  {
    //......(.NET平台)Log Database
  }
}

public class JDatabaseLog : DatabaseLog
{
  public override void Write(string log)
  {
    //......(Java平台)Log Database
  }
}
public class NTextFileLog : TextFileLog
```

```
{
  public override void Write(string log)
  {
    //......(.NET平台)Log Text File
  }
}
public class JTextFileLog : TextFileLog
{
  public override void Write(string log)
  {
    //......(Java平台)Log TextFile
  }
}
```

现在的这种设计方案本身是没有任何错误的，假如现在我们要引入一种新的xml文件的记录方式，则上面的类结构图会变成：



如图中蓝色的部分所示，我们新增加了一个继承于Log基类的子类，而没有修改其它的子类，这样也符合了开放-封闭原则。如果我们引入一种新的平台，比如说我们现在开发的日志记录工具还需要支持Borland平台，此时该类结构又变成了：

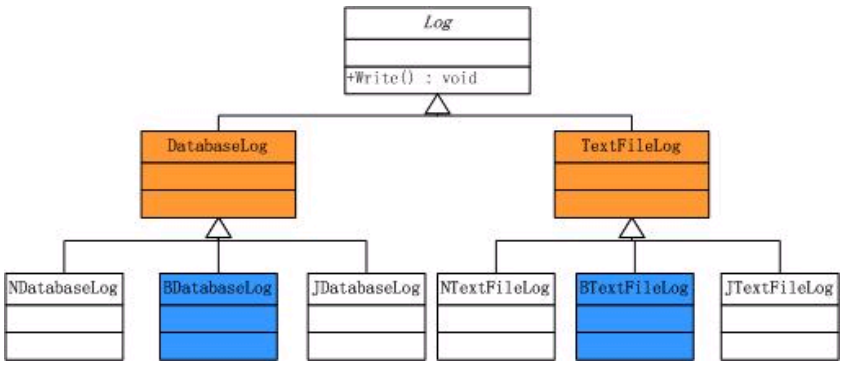
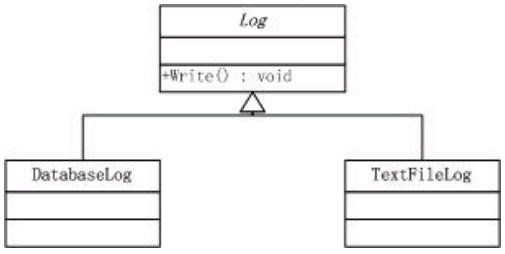


图7

同样我们没有修改任何的东西，只是增加了两个继承于DatabaseLog和TextFileLog的子类，这也符合了开放-封闭原则。

但是我们说这样的设计是脆弱的，仔细分析就可以发现，它还是存在很多问题，首先它在遵循开放-封闭原则的同时，违背了类的单一职责原则，即一个类只有一个引起它变化的原因，而这里引起Log类变化的原因却有两个，即日志记录方式的变化和日志记录平台的变化;其次是重复代码会很多，不同的日志记录方式在不同的平台上也会有一部分的代码是相同的;再次是类的结构过于复杂，继承关系太多，难于维护，最后最致命的一点是扩展性太差。上面我们分析的变化只是沿着某一个方向，如果变化沿着日志记录方式和不同的运行平台两个方向变化，我们会看到这个类的结构会迅速的变庞大。

现在该是Bridge模式粉墨登场的时候了，我们需要解耦这两个方向的变化，把它们之间的强耦合关系改成弱联系。我们把日志记录方式和不同平台上的实现分别当作两个独立的部分来对待，对于日志记录方式，类结构图仍然是：



现在我们引入另外一个抽象类ImpLog，它是日志记录在不同平台的实现的基类，结构图如下：

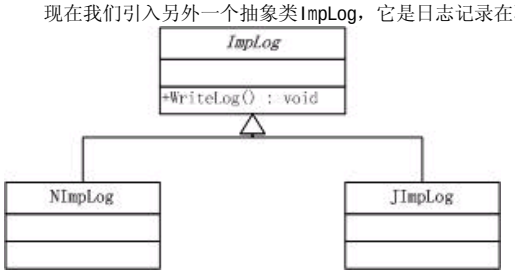


图9

实现代码如下：

```
public abstract class ImpLog
{
  public abstract void Execute(string msg);
}
public class NImpLog : ImpLog
{
  public override void Execute(string msg)
  {
    //..... .NET平台
  }
}

public class JImpLog : ImpLog
{
  public override void Execute(string msg)
  {
    //..... Java平台
  }
}
```

这时对于日志记录方式和不同的运行平台这两个类都可以独立的变化了，我们要做的工作就是把这两部分之间连接起来。那如何连接呢?在这里，Bridge使用了对象组合的方式，类结构图如下:

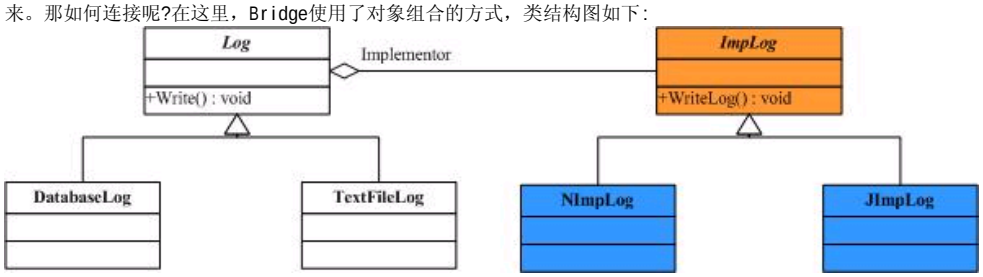


图 10

实现代码如下:

```
public abstract class Log
{
  protected ImpLog implementor;
  public ImpLog Implementor
  {
    set { implementor = value; }
  }
  public virtual void Write(string log)
  {
    implementor.Execute(log);
  }
}

public class DatabaseLog : Log
{
  public override void Write(string log)
  {
    implementor.Execute(log);
  }
}
public class TextFileLog : Log
{
  public override void Write(string log)
  {
    implementor.Execute(log);
  }
}
```

可以看到，通过对象组合的方式，Bridge模式把两个角色之间的继承关系改为了耦合的关系，从而使这两者可以从容自若的各自独立的变化，这也是Bridge模式的本意。再来看一下客户端如何去使用:

```
class App
{
  public static void Main(string[] args)
  {
    //.NET平台下的Database Log
    Log dblog = new DatabaseLog();
    dblog.Implementor = new NImpLog();
    dblog.Write();
    //Java平台下的Text File Log
    Log txtlog = new TextFileLog();
    txtlog.Implementor = new JImpLog();
    txtlog.Write();
  }
}
```

可能有人会担心说，这样不就又增加了客户程序与具体日志记录方式之间的耦合性了吗?其实这样的担心是没有必要的，因为这种耦合性是由于对象的创建所带来的，完全可以用创建型模式去解决，就不是这里我们所讨论的内容了。

最后我们再来考虑一个问题，为什么Bridge模式要使用对象组合的方式而不是用继承呢?如果采用继承的方式，则Log类，ImpLog类都为接口，类结构图如下:

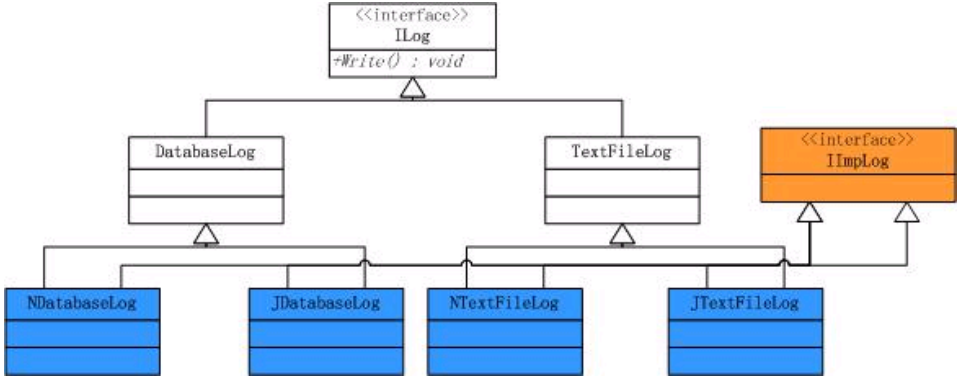


图11

实现代码如下:

```
public class NDatabaseLog : DatabaseLog, IImpLog
{
  //......
}
public class JDatabaseLog : DatabaseLog, IImpLog
{
  //......
}
public class NTextFileLog : TextFileLog, IImpLog
{
  //......
}
public class JTextFileLog : TextFileLog, IImpLog
{
  //......
}
```

如上图中蓝色的部分所示，它们既具有日志记录方式的特性，也具有接口IimpLog的特性，它已经违背了面向对象设计原则中类的单一职责原则，一个类应当仅有一个引起它变化的原因。所以采用Bridge模式往往是比采用多继承更好的方案。说到这里，大家应该对Bridge模式有一些认识了吧?如果在开发中遇到有两个方向上纵横交错的变化时，应该能够想到使用Bridge模式，当然了，有时候虽然有两个方向上的变化，但是在某一个方向上的变化并不是很剧烈的时候，并不一定要使用Bridge模式。

效果及实现要点

1.Bridge模式使用“对象间的组合关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。

2.所谓抽象和实现沿着各自维度的变化，即“子类化”它们，得到各个子类之后，便可以任意它们，从而获得不同平台上的不同型号。

3.Bridge模式有时候类似于多继承方案，但是多继承方案往往违背了类的单一职责原则(即一个类只有一个变化的原因)，复用性比较差。Bridge模式是比多继承方案更好的解决方法。

4.Bridge模式的应用一般在“两个非常强的变化维度”，有时候即使有两个变化的维度，但是某个方向的变化维度并不剧烈——换言之两个变化不会导致纵横交错的结果，并不一定要使用Bridge模式。

适用性

在以下的情况下应当使用桥梁模式:

1.如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。

2.设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。

3.一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。

4.虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

总结

Bridge模式是一个非常有用的模式，也非常复杂，它很好的符合了开放-封闭原则和优先使用对象，而不是继承这两个面向对象原则。

您在中国网游研发中心留言板发表的作品，中国网游研发中心有权在网站内转载或引用

参与本留言即表明您已经阅读并接受上述条款