[**.NET中的设计模式——一步步发现装饰模式**](http://www.cnblogs.com/niyw/archive/2010/10/10/1847253.html)

**前言**

　　过去我的一个朋友常说，学习任何编程语言最困难的部分是运行“Hello World”，之后一切都很容易。多年以后，我才意识到他说的很对。学习设计模式的基本目标是要用它，尤其是帮助那些有扎实的OOP基础，而对设计模式很困惑的人在设计中应用它。我不会为不同设计模式写很全面的参考，但我希望这些文章能让你入门。设计模式与特定的语言无关。虽然我用C#写了很多示例，但我尽量避免一些C#特有的结构，因此它面向大部分人，尤其是使用C++的人。

　　装饰器模式允许我们动态为对象添加行为。下面我们先介绍一个场景，然后寻找替代方法。这会帮助我们认清该模式的真实用途，尤其在灵活性这方面。

**思考过程**

　　今天我们参考的不是一个实际的场景，有时可能会很奇怪，但它会帮助我们弄清这个模式及相关概念。我们的目标是添加将不同来源的信息存储到磁盘文件上的功能。所以，第一步我们定义一个接口，并实现它。**IMessageWriter**和**IMessageReader**接口分别用于写入和读取信息，他们的实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | interface IMessageWriter {      string Message { set; }      void WriteMessage(string filePath);  }  class MessageWriter : IMessageWriter {      private string message;      public string Message { set{message =value;} }      public virtual void WriteMessage(string filePath) {          File.WriteAllText(filePath, message);      }  }  interface IMessageReader {      string ReadMessage(string filePath);  }  class MessageReader : IMessageReader {      public virtual string ReadMessage(string filePath) {          if (File.Exists(filePath))              return File.ReadAllText(filePath);          return null;      }  } |

信息作为Message属性存储，MessageWriter的方法WriteMessage把它写到指定的文件。同样，MessageReader的方法ReadMessage从指定的文件读取，并以字符串的形式返回。现在假设客户提出了新需求。

1. 对某些信息在读和写文件之前，我们需要验证用户；
2. 对某些信息我们希望加密后保存，来防止别人读取，并且我们需要以64位编码保存加密信息；
3. 对某些信息，我们都需要这些功能；

很奇怪吧，呵呵，首先我们不用装饰器分析不同的解决方案，这会使我们对这个简单的设计模式认识更加清晰。

**传统解决方案**

你决定在原来行为上使用继承。你从MessageWriter继承EncryptedMessageWriter来实现加密行为。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class EncryptedMessageWriter : MessageWriter {      public override void WriteMessage(string filePath) {          //加密消息          //转换到64位编码          Message = "base64StringYouGotFromAboveCode";//存储信息          base.WriteMessage(filePath);      }  } |

同样，你从EncrytedMessageWriter继承SecureMessageWriter来实现用户验证。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class SecureMessageWriter : EncryptedMessageWriter {      public override void WriteMessage(string filePath) {          if (ValidateUser())              base.WriteMessage(filePath);          else              Console.WriteLine("No message saved,user validation failed.");      }      private bool ValidateUser() {          //验证用户,失败返回false          return true;      }  } |

　　现在我们能写入加密的信息，或经过用户验证后的加密信息。那么如果需要写入一些只需要用户验证而不需要加密的简单文本信息时，我们该怎么办？你可以在EncryptedMessageWriter中写入一些丑陋的判断，在不需要加密的时候跳过加密。假设遇到此类情况你还这么做，那么那些操作换个顺序呢，例如我们想先加密后验证，如果验证失败，则除64位编码加密消息外在做点别的。很显然，上面的组织结构无法处理这种情况。谁能阻止用户提出更多的需求，像消息需要数据签名，大消息需要压缩或不需要加密，对于某些信息，写到磁盘后，你必须在消息队列中输入文件路径和时间戳以便其他程序读取，甚至写到数据库中，等等等等？！显然不能。

　　让我们只关注验证，忽略其他细节，评估一下你面对情况的复杂性和严重性。目前，我们在加密消息时实现了用户验证。现在我们需要满足其他相同的功能，如：CompressedMessageWriter,DigitallySignedMessageWriter等。你唯一能做的是实现SecureCompressedMessageWriter,SecureDigitallySignedMessageWriter等。同样对其他大量的组合，像压缩加密信息，简单信息压缩等等。天哪，你真的坠入“子类地狱”了。

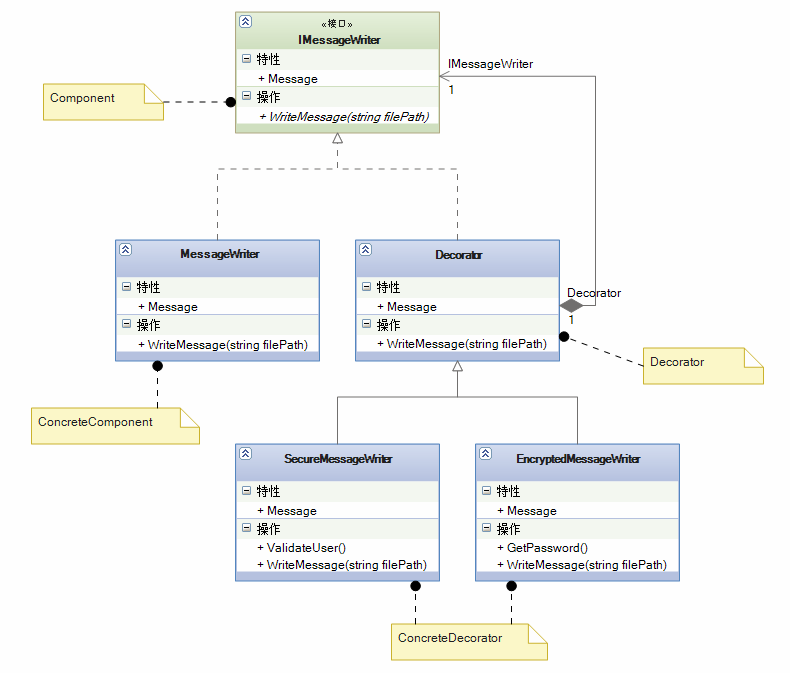
　　第二个解决方案是写一个非常大的MessageReader，处理所有提到的需求功能。随着时间流逝，它变得越来越复杂，越来越难以维护——非常不推荐这样。

　　第三个解决方案可能是上面两种方案的合并，这可能是治标不治本。

**引入装饰器模式**

　　这恰恰是装饰器模式解决的问题。如果你仔细观察上面采用继承的解决方案，你会认识到问题的根源是继承带来的静态关联。这些关联被嵌入到类之间，并且不能在运行时改变。装饰器用包含替换关联，而包含是一种非常灵活且在运行时能被更新的对象关联。

　　首先让我们看看装饰器模式究竟是什么。下面是装饰器模式的类图。



四个参与者:

1. **Component**:定义一个对象接口，可以动态的给对象添加职责. 在我们的例子中是IMessageWriter和IMessageReader；
2. **ConcreteComponent**: 定一个实现Component接口的对象。这个对象会被装饰，但它不会包含任何装饰者的信息，而装饰者不会访问它的实现。在我们例子中是MessageWriter和MessageReader。
3. **Decorator**: 包含一个Component对象的引用，定义一个与Component一致的接口。所以它包含一个指向基本行为的引用，并且实现了相同的接口，因此能被Component自己访问。客户端代码期望Component能不需要关心装饰者之间的差别处理它们。
4. **ConcreteDecorator**: 它向Component添加职责。从Decorator继承来的类可以统一添加新方法的形式添加一些额外的功能。

　　到目前为止，我们已有两部分，分别是Component：基本行为，即在我们例子中是IMessageWriter和用于读的IMessageReader；和ConcreteComponent，即我们实现的读写行为：MessageWriter和MessageReader。

　　下面是我们实现的SecureMessageWriter和EncryptedMessageWriter。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | class SecureMessageWriter : IMessageWriter {          private string message;          //被装饰者          private IMessageWriter messageWriter;          public SecureMessageWriter(IMessageWriter msgWriter) {              this.messageWriter = msgWriter;          }          public string Message {              set { message = value; }          }          public void WriteMessage(string filePath) {              if (this.ValidateUser()) {//添加新的行为                  //正如你所见,在调用被装饰者的标准方法前我们添加了验证行为                  messageWriter.Message = this.message;                  messageWriter.WriteMessage(filePath);              }              else                  Console.WriteLine("");          }          private bool ValidateUser() {              //验证用户代码              return true;          }      }      class EncryptedMessageWriter : IMessageWriter {          private string message;          //被装饰者          private IMessageWriter msgWriter;          public EncryptedMessageWriter(IMessageWriter msgWriter) {              this.msgWriter = msgWriter;          }          public string Message {              set { message = value; }          }          public void WriteMessage(string filePath) {              this.msgWriter.Message = "encrytedMsgInBase64";//加密信息              //被装饰得行为              this.msgWriter.WriteMessage(filePath);          }          private string GetPassword() {              Console.WriteLine("Please provide security password");              return Console.ReadLine();          }      } |

**这里有问题吗?????**

　　我刚刚说这个模式有四个参与者，我已向你展示了Component(IMessageReader，IMessageWriter)，ConcreteComponent(MessageReader,MessageWriter)和ConcreteDecorator(SecureMessageWriter，EncryptedMessageWriter)。但Decorator那里去了？在我们的例子中，我们仅仅添加了已存在的行为，没有引进新的行为。我们没有改变其他结构。在这种情况下，我们忽略了实现Decorator，并沿用主要层次结构。这里我不在展示读有关的类，它们仅仅是反向处理。

**我们学到了什么**

　　现在如果我需要一个经过用户验证的简单信息写操作，我会这么做：

|  |  |
| --- | --- |
| 1 | IMessageWriter msgWriter = new SecureMessageWriter(new MessageWriter()); |

　　我用SecureMessageWriter装饰了MessageWriter，它现在在写信息到磁盘前会先验证用户。如果同时需要验证用户和加密信息，我会这么做：

|  |  |
| --- | --- |
| 1 | IMessageWriter msgWriter = new SecureMessageWriter(new EncryptionMessageWriter(new MessageWriter())); |

1. 装饰器使我们在大多数情况下避免构造复杂基类，撰写大量的代码。
2. 装饰器允许我们以不同的顺序进行不同的组合，这在别的方法来说不太容易。
3. 相比为不同的行为与合并实现不同的基类，我们可以按需实现单独的需求行为。

**回到现实**

这一部分我们会看到装饰器模式在实际中应用的例子。

**同步包**

　　善于使用.ET中旧集合类，如Queue，ArrayList等的人可能还记得许多类提供的(集合)同步函数。它把集合实例自己作为参数传入，并返回一个同步集合。尽管集合类自己并不同步，但这个方法返回的实际上是从集合类自己继承过来的一个装饰器。例如，在下面代码中当我们调用Syncrhonized(a1)时，我们会收到一个从ArrayList继承来的SyncArrayList的实例。

|  |  |
| --- | --- |
| 1  2 | ArrayList al = new ArrayList();  al = ArrayList.Synchronized(al); |

SyncArrayList存储通过属性\_list传递的ArrayList，并且按照同样的方法重载不同的实例方法。

|  |  |
| --- | --- |
| 1  2  3  4  5 | public override int Add(object value) {         lock (this.\_root){             return this.\_list.Add(value);         }     } |

**注意事项**

*按照此法创建同步包时，要注意叫做“自死锁”现象，这意味着一个占有锁的线程进入另一格方法(或者递归)，然后又试图获取同一个对象的锁。在Windows中，如果你使用.NET实施监控，或者内核级命名，或者无名互斥，全部都有重入机制(即递归)。所以你不会遇到此类问题，但是在其他环境(如Linux)中编程时,默认互斥类型是快速互斥（一种非递归互斥），你的代码就可能成为“自死锁”的受害者。假如使用消息，即使在Windows上，它一样没有自我意识，如果你不注意，就会给你带来这个问题。当然，对于一个简单信号，比如n=1，在第二次访问时，你一样会遇到“自死锁”。*

*同理，你可以为你的集合类实现一个只读包。不像我们到现在所看到的，与其说他们是向类上添加功能，倒不如说去掉一些。例如在重载方法Add()中可能抛出操作不支持的异常。.NET提供了ReadOnlyCollection<T>，它用于包装泛型列表。Java则提供了只读包，如UnmodifiableCollection,UnmodifiableSet等等。*  
　　Java中，你可以按照下面方式为很多集合类型获取同步包。

|  |  |
| --- | --- |
| 1 | List list = Collections.synchronizedList(new ArrayList()); |

**Java和.NET中的IO**

　　Java的java.io包和.NET的Stream类都使用了该模式。我不会就它们的实现谈论很多细节。.NET中，Stream是一个抽象类，提供了基本的行为(如Component)，从抽象类Stream继承来的类FileStream,MemoryStream是ConcreteCompents，类BufferedStream,CrytoStream等是ConcreteDecorators。你能清楚地认识到它们同样忽略了Decorator。

　　同样，在Java中，BufferedReader和FilterReader是Reader的装饰者。而BufferedReader进一步被LineNumberReader装饰。FilterReader被PushbackReader装饰。

**收获了什么**

　　装饰器模式允许我们在实现中提供扩展点。你可能注意到，在实现装饰者时，我从来没有涉及到Component类。因此，即使它并不拥有类，一样能通过动态添加行为类装饰它，甚至是递归的方式。这些额外的行为可能在几处行为之前或之后添加，或者这种行为可能被阻止。装饰者能在类中提供新方法，甚至新属性。但装饰器同样有一些问题。

1. 使用装饰器的程序员必须明白它们的意图，否则他最终可能会使用一些毫无意义的组合或序列。比如在我们的场景中，如果程序员按照这种方式定义序列：信息先被压缩，然后加密，最后验证，它将毫无意义。在现实场景中，有些组合或次序可能是灾难性的。
2. 测试一个装饰类需要提供一个模拟的被装饰类。因为我们还没有涉及到测试，所以这里就不说了。
3. 该模式在基本行为上添加一些职责。虽然给装饰者添加新的属性或方法完全合法，但这种方法导致基础类处理问题的灵活性，因为你需要用到具体的实例。

**写在最后**

　　此文为[CodeProject](http://www.codeproject.com/)上同名文章<[Decorator Pattern(A Layman to Laymen)](http://www.codeproject.com/KB/architecture/DecoratorPrimerA.aspx)>的意译和修改，文中部分内容与原文有出入。**文章通过循序渐进的方式逐步暴露问题(各种继承，子类，代码像麻花一样挤在一起)，然后用装饰模式解决(把每种需求单独构建成组件，然后像搭积木一样完成工作)，最后联系实际说明装饰模式的真正含义**，使得整篇文章有理有据，内容清晰易懂，相信对学习什么模式的人有很大帮助。限于个人能力，文中难免有错误的地方，还请大家多多指教（**特别是斜体部分:(**）。

本文参考：《设计模式可复用面向对象软件的基础》

　　　　　《[Decorator Pattern(A Layman to Laymen)](http://www.codeproject.com/KB/architecture/DecoratorPrimerA.aspx)》

　　　　　《.NET与设计模式》

              《MSDN》