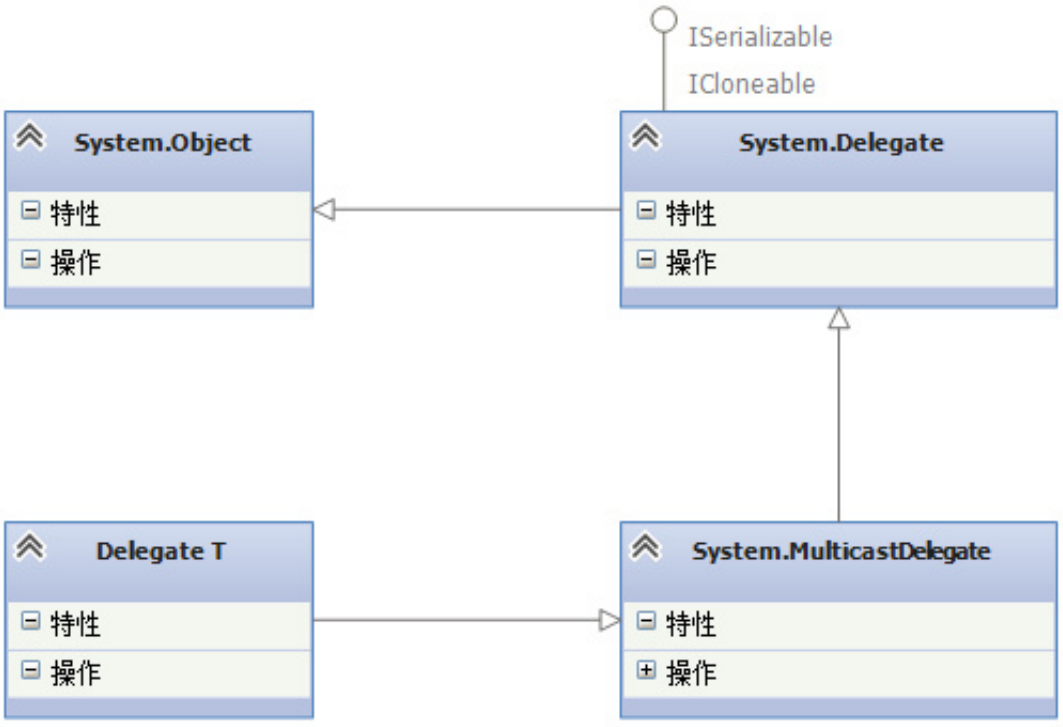
# .NET 基础拾遗（4）：委托、事件、反射与特性 - 文章 - 伯乐在线

一、委托基础1.1 简述委托的基本原理

委托这个概念对C++程序员来说并不陌生，因为它和C++中的函数指针非常类似，很多码农也喜欢称委托为安全的函数指针。无论这一说法是否正确，委托的的确确实现了和函数指针类似的功能，那就是提供了程序回调指定方法的机制。

在委托内部，包含了一个指向某个方法的指针（这一点上委托实现机制和C++的函数指针一致），为何称其为安全的呢？因此委托和其他.NET成员一样是一种类型，任何委托对象都是继承自System.Delegate的某个派生类的一个对象，下图展示了在.NET中委托的类结构：

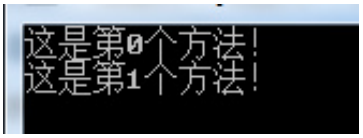



从上图也可以看出，任何自定义的委托都继承自基类System.Delegate，在这个类中，定义了大部分委托的特性。那么，下面可以看看在.NET中如何使用委托：

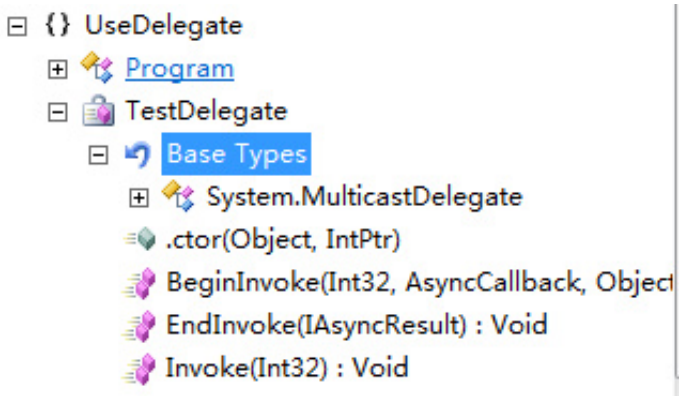
```
C#// 定义的一个委托
        public delegate void TestDelegate(int i);
        public class Program
        {
                public static void Main(string[] args)
                {
                        // 定义委托实例
```

```
                TestDelegate td = new TestDelegate(PrintMessage);
                // 调用委托方法
                td(0);
                td.Invoke(1);
                Console.ReadKey();
            }
            public static void PrintMessage(int i)
            {
                Console.WriteLine("这是第{0}个方法！", i.ToString());
            }
        }
```

运行结果如下图所示：




上述代码中定义了一个名为TestDelegate的新类型，该类型直接继承自System.MulticastDelegate，而且其中会包含一个名为Invoke、BeginInvoke和EndInvoke的方法，这些步骤都是由C#编译器自动帮我们完成的，可以通过Reflector验证一下如下图所示：




需要注意的是，委托既可以接受实例方法，也可以接受静态方法（如上述代码中接受的就是静态方法），其区别我们在1.2中详细道来。最后，委托被调用执行时，C#编译器可以接收一种简化程序员设计的语法，例如上述代码中的：td(1)。但是，本质上，委托的调用其实就是执行了在定义委托时所生成的Invoke方法。

1.2 委托回调静态方法和实例方法有何区别？

首先，我们知道静态方法可以通过类名来访问而无需任何实例对象，当然在静态方法中也就不能访问类型中任何非静态成员。相反，实例方法则需要通过具体的实例对象来调用，可以访问实例对象中的任何成员。

其次，当一个实例方法被调用时，需要通过实例对象来访问，因此可以想象当绑定一个实例方法到委托时必须同时让委托得到实例方法的代码段和实例对象的信息，这样在委托被回调的时候.NET才能成功地执行该实例方法。

下图展示了委托内部的主要结构：


① _target是一个指向目标实例的引用，当绑定一个实例方法给委托时，该参数会作为一个指针指向该方法所在类型的一个实例对象。相反，当绑定一个静态方法时，该参数则被设置为null。

② _methodPtr则是一个指向绑定方法代码段的指针，这一点和C++的函数指针几乎一致。绑定静态方法或实例方法在这个成员的设置上并没有什么不同。

System.MulticastDelegate在内部结构上相较System.Delegate增加了一个重要的成员变量：_prev，它用于指向委托链中的下一个委托，这也是实现多播委托的基石。


1.3 神马是链式委托？

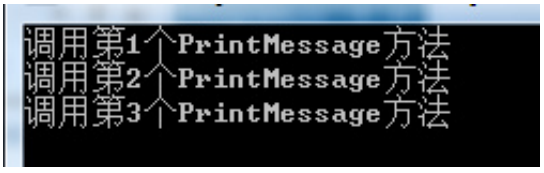
链式委托也被称为“多播委托”，其本质是一个由多个委托组成的链表。回顾上面1.2中的类结构，System.MulticastDelegate类便是为链式委托而设计的。当两个及以上的委托被链接到一个委托链时，调用头部的委托将导致该链上的所有委托方法都被执行。

下面看看在.NET中，如何申明一个链式委托：

```
C#// 定义的一个委托
        public delegate void TestMulticastDelegate();
        public class Program
        {
                public static void Main(string[] args)
                {
                        // 申明委托并绑定第一个方法
                        TestMulticastDelegate tmd = new
TestMulticastDelegate(PrintMessage1);
                        // 绑定第二个方法
                        tmd += new TestMulticastDelegate(PrintMessage2);
                        // 绑定第三个方法
                        tmd += new TestMulticastDelegate(PrintMessage3);
                        // 调用委托
                        tmd();
```

```
            Console.ReadKey();
        }
        public static void PrintMessage1()
        {
            Console.WriteLine("调用第1个PrintMessage方法");
        }
        public static void PrintMessage2()
        {
            Console.WriteLine("调用第2个PrintMessage方法");
        }
        public static void PrintMessage3()
        {
            Console.WriteLine("调用第3个PrintMessage方法");
        }
    }
```

其运行结果如下图所示：




可以看到，调用头部的委托导致了所有委托方法的执行。通过前面的分析我们也可以知道：为委托+=增加方法以及为委托-=移除方法让我们看起来像是委托被修改了，其实它们并没有被修改。事实上，委托是恒定的。在为委托增加和移除方法时实际发生的是创建了一个新的委托，其调用列表是增加和移除后的方法结果。

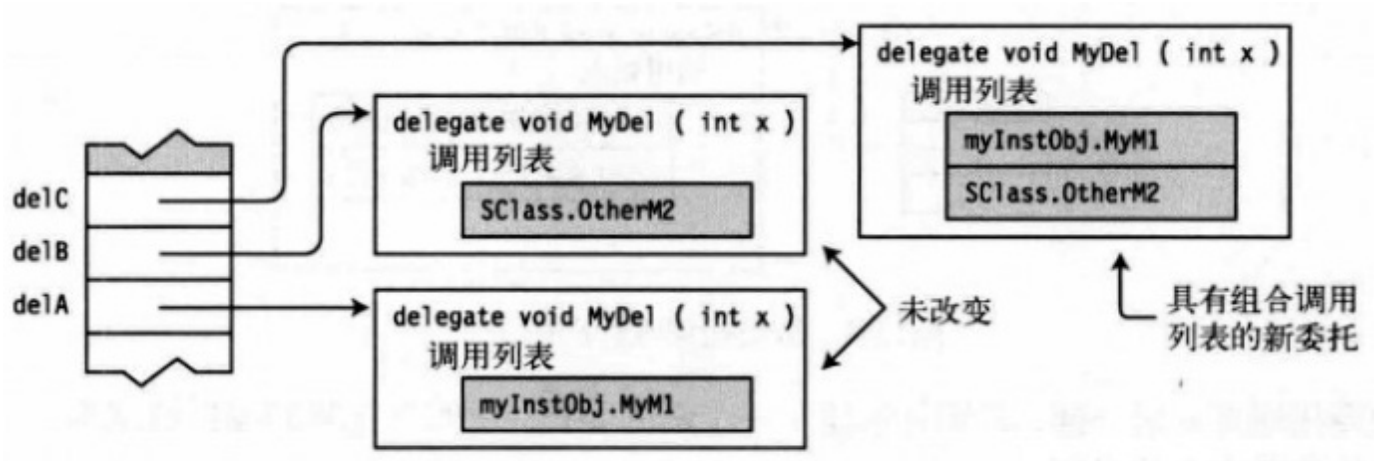



图13-6 组合委托

另一方面，+= 或-= 这是一种简单明了的写法，回想在WindowsForm或者ASP.NET WebForms开发时，当添加一个按钮事件，VS便会自动为我们生成类似的代码，这样一想是不是又很熟悉了。

现在，我们再用一种更简单明了的方法来写：

C#

```
        TestMulticastDelegate tmd = PrintMessage1;
```

```
        tmd += PrintMessage2;
        tmd += PrintMessage3;
        tmd();
```

其执行结果与上图一致，只不过C#编译器的智能化已经可以帮我们省略了很多代码。

最后，我们要用一种比较复杂的方法来写，但是却是链式委托的核心所在：

C#

```
    TestMulticastDelegate tmd1 = new                TestMulticastDelegate(PrintMessage1);
        TestMulticastDelegate tmd2 = new    TestMulticastDelegate(PrintMessage2);
        TestMulticastDelegate tmd3 = new         TestMulticastDelegate(PrintMessage3);
        // 核心本质：将三个委托串联起来
        TestMulticastDelegate tmd = tmd1 + tmd2 + tmd3;
        tmd.Invoke();
```

我们在实际开发中经常使用第二种方法，但是却不能不了解方法三，它是链式委托的本质所在。

1.4 链式委托的执行顺序是怎么样的？

前面我们已经知道链式委托的基本特性就是一个以委托组成的链表，而当委托链上任何一个委托方法被调用时，其后面的所有委托方法都将会被依次地顺序调用。那么问题来了，委托链上的顺序是如何形成的？这里回顾一下上面1.3中的示例代码，通过Reflector反编译一下，一探究竟：

Disassembler

```
public static void Main(string[] args)
{
    TestMulticastDelegate tmd = new TestMulticastDelegate(Program.PrintMessage1);
    tmd = (TestMulticastDelegate) Delegate.Combine(tmd, new TestMulticastDelegate(Program.PrintMessage2));
    tmd = (TestMulticastDelegate) Delegate.Combine(tmd, new TestMulticastDelegate(Program.PrintMessage3));
    tmd();
    Console.ReadKey();
}
```

从编译后的结果可以看到，+=的本质又是调用了Delegate.Combine方法，该方法将两个委托链接起来，并且把第一个委托放在第二个委托之前，因此可以将两个委托的相加理解为Deletegate.Combine(Delegate a,Delegate b)的调用。我们可以再次回顾System.MulticastDelegate的类结构：


其中_prev成员是一个指向下一个委托成员的指针，当某个委托被链接到当前委托的后面时，该成员会被设置为指向那个后续的委托实例。.NET也是依靠这一个引用来逐一找到当前委托的所有后续委托并以此执行方法。

那么，问题又来了？程序员能够有能力控制链式委托的执行顺序呢？也许我们会说，只要在定义时按照需求希望的顺序来依次添加就可以了。但是，如果要在定义完成之后突然希望改变执行顺序呢？又或者，程序需要按照实际的运行情况再来决定链式委托的执行顺序呢？

接下来就是见证奇迹的时刻：

```
C#// 申明委托并绑定第一个方法
        TestMulticastDelegate tmd = new TestMulticastDelegate(PrintMessage1);
        // 绑定第二个方法
        tmd += new TestMulticastDelegate(PrintMessage2);
        // 绑定第三个方法
        tmd += new TestMulticastDelegate(PrintMessage3);
        // 获取所有委托方法
        Delegate[] dels = tmd.GetInvocationList();
```

上述代码调用了定义在System.MulticastDelegate中的GetInvocationList()方法，用以获得整个链式委托中的所有委托。接下来，我们就可以按照我们所希望的顺序去执行它们。

1.5 可否定义有返回值方法的委托链？

委托的方法既可以是无返回值的，也可以是有返回值的，但如果多一个带返回值的方法被添加到委托链中时，我们需要手动地调用委托链上的每个方法，否则只能得到委托链上最后被调用的方法的返回值。

为了验证结论，我们可以通过如下代码进行演示：

```
C#// 定义一个委托
        public delegate string GetStringDelegate();
        class Program
        {
                static void Main(string[] args)
                {
                        // GetSelfDefinedString方法被最后添加
                        GetStringDelegate myDelegate1 = GetDateTimeString;
                        myDelegate1 += GetTypeNameString;
                        myDelegate1 += GetSelfDefinedString;
                        Console.WriteLine(myDelegate1());
                        Console.WriteLine();
                        // GetDateTimeString方法被最后添加
                        GetStringDelegate myDelegate2 = GetSelfDefinedString;
                        myDelegate2 += GetTypeNameString;
                        myDelegate2 += GetDateTimeString;
                        Console.WriteLine(myDelegate2());
                        Console.WriteLine();
                        // GetTypeNameString方法被最后添加
                        GetStringDelegate myDelegate3 = GetSelfDefinedString;
                        myDelegate3 += GetDateTimeString;
```

```
                myDelegate3 += GetTypeNameString;
                Console.WriteLine(myDelegate3());
                Console.ReadKey();
            }
            static string GetDateTimeString()
            {
                return DateTime.Now.ToString();
            }
            static string GetTypeNameString()
            {
                return typeof(Program).ToString();
            }
            static string GetSelfDefinedString()
            {
                string result = "我是一个字符串！";
                return result;
            }
        }
```

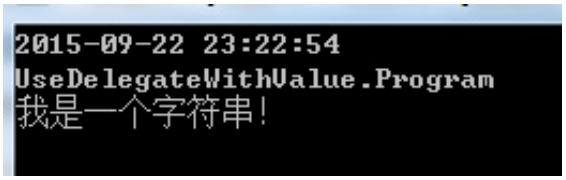
其运行结果如下图所示：


从上图可以看到，虽然委托链中的所有方法都被正确执行，但是我们只得到了最后一个方法的返回值。在这种情况下，我们应该如何得到所有方法的返回值呢？回顾刚刚提到的GetInvocationList()方法，我们可以利用它来手动地执行委托链中的每个方法。

C#

```
        GetStringDelegate myDelegate1 = GetDateTimeString;
        myDelegate1 += GetTypeNameString;
        myDelegate1 += GetSelfDefinedString;
        foreach (var del in myDelegate1.GetInvocationList())
        {
                Console.WriteLine(del.DynamicInvoke());
         }
```

通过上述代码，委托链中每个方法的返回值都不会丢失，下图是执行结果：



1.6 简述委托的应用场合

委托的功能和其名字非常类似，在设计中其思想在于将工作委派给其他特定的类型、组件、方法或程序集。委托的使用者可以理解为工作的分派者，在通常情况下使用者清楚地知道哪些工作需要执行、执行的结果又是什么，但是他不会亲自地去做这些工作，而是恰当地把这些工作分派出去。

这里，我们假设要写一个日志子系统，该子系统的需求是使用者希望的都是一个单一的方法传入日志内容和日志类型，而日志子系统会根据具体情况来进行写日志的动作。对于日志子系统的设计者来说，写一条日志可能需要包含一系列的工作，而日志子系统决定把这些工作进行适当的分派，这时就需要使用一个委托成员。

下面的代码展示了该日志子系统的简单实现方式：

① 定义枚举：日志的类别

C#

② 定义委托，由日志使用者直接执行来完成写日志的工作

C#

```
1
public delegate void Log(string content, LogType type);
```

③ 定义日志管理类，在构造方法中为记录日志委托定义了默认的逻辑（这里采用了部分类的书写，将各部分的委托方法分隔开，便于理解）

C#

```
public sealed partial class LogManager:IDisposable
        {
                private Type _componentType;
                private String _logfile;
                private FileStream _fs;
                public Log WriteLog;                    //用来写日志的委托
                //锁
                private static object mutext = new object();
                //严格控制无参的构造方法
                private LogManager()
                {
                        WriteLog = new Log(PrepareLogFile);
                        WriteLog += OpenStream; //打开流
                        WriteLog += AppendLocalTime;        //添加本地时间
                        WriteLog += AppendSeperator;        //添加分隔符
                        WriteLog += AppendComponentType;//添加模块类别
                        WriteLog += AppendSeperator;        //添加分隔符
                        WriteLog += AppendType;                 //添加日志类别
                        WriteLog += AppendSeperator;        //添加分隔符
                        WriteLog += AppendContent;            //添加内容
```

```
                WriteLog += AppendNewLine;              //添加回车
                WriteLog += CloseStream;                //关闭流
        }
        /// <summary>
        /// 构造方法
        /// </summary>
        /// <param name="type">使用该日志的类型</param>
        /// <param name="file">日志文件全路径</param>
        public LogManager(Type type, String file):this()
        {
                _logfile = file;
                _componentType = type;
        }
        /// <summary>
        /// 释放FileStream对象
        /// </summary>
        public void Dispose()
        {
                if (_fs != null)
                        _fs.Dispose();
                GC.SuppressFinalize(this);
        }
        ~LogManager()
        {
                if (_fs != null)
                        _fs.Dispose();
        }
}
/// <summary>
/// 委托链上的方法（和日志文件有关的操作）
/// </summary>
public sealed partial class LogManager:IDisposable
{
        /// <summary>
        /// 如果日志文件不存在，则新建日志文件
        /// </summary>
        private void PrepareLogFile(String content, LogType type)
        {
                //只允许单线程创建日志文件
                lock(mutext)
                {
                        if (!File.Exists(_logfile))
```

```
                                using (FileStream fs = File.Create(_logfile))
                                { }
                        }
                }
                /// <summary>
                /// 打开文件流
                /// </summary>
                private void OpenStream(String content, LogType type)
                {
                        _fs = File.Open(_logfile, FileMode.Append);
                }
                /// <summary>
                /// 关闭文件流
                /// </summary>
                private void CloseStream(String content, LogType type)
                {
                        _fs.Close();
                        _fs.Dispose();
                }
        }
        /// <summary>
        /// 委托链上的方法（和日志时间有关的操作）
        /// </summary>
        public sealed partial class LogManager : IDisposable
        {
                /// <summary>
                /// 为日志添加当前UTC时间
                /// </summary>
                private void AppendUTCTime(String content, LogType type)
                {
                        String time=DateTime.Now.ToUniversalTime().ToString();
                        Byte[] con = Encoding.Default.GetBytes(time);
                        _fs.Write(con, 0, con.Length);
                }
                /// <summary>
                /// 为日志添加本地时间
                /// </summary>
                private void AppendLocalTime(String content, LogType type)
                {
                        String time = DateTime.Now.ToLocalTime().ToString();
                        Byte[] con = Encoding.Default.GetBytes(time);
                        _fs.Write(con, 0, con.Length);
```

```
        }
    }
    /// <summary>
    /// 委托链上的方法（和日志内容有关的操作）
    /// </summary>
    public sealed partial class LogManager : IDisposable
    {
        /// <summary>
        /// 添加日志内容
        /// </summary>
        private void AppendContent(String content, LogType type)
        {
            Byte[] con = Encoding.Default.GetBytes(content);
            _fs.Write(con, 0, con.Length);
        }
        /// <summary>
        /// 为日志添加组件类型
        /// </summary>
        private void AppendComponentType(String content, LogType type)
        {
            Byte[] con = Encoding.Default.GetBytes(_componentType.ToString());
            _fs.Write(con, 0, con.Length);
        }
        /// <summary>
        /// 添加日志类型
        /// </summary>
        private void AppendType(String content, LogType type)
        {
            String typestring = String.Empty;
            switch (type)
            {
                case LogType.Debug:
                    typestring = "Debug";
                    break;
                case LogType.Error:
                    typestring = "Error";
                    break;
                case LogType.Info:
                    typestring = "Info";
                    break;
                case LogType.Trace:
                    typestring = "Trace";
```

```
                        break;
                    case LogType.Warn:
                        typestring = "Warn";
                        break;
                    default:
                        typestring = "";
                        break;
                }
                Byte[] con = Encoding.Default.GetBytes(typestring);
                _fs.Write(con, 0, con.Length);
            }
        }
        /// <summary>
        /// 委托链上的方法（和日志的格式控制有关的操作）
        /// </summary>
        public sealed partial class LogManager : IDisposable
        {
            /// <summary>
            /// 添加分隔符
            /// </summary>
            private void AppendSeperator(String content, LogType type)
            {
                Byte[] con = Encoding.Default.GetBytes(" | ");
                _fs.Write(con, 0, con.Length);
            }
            /// <summary>
            /// 添加换行符
            /// </summary>
            private void AppendNewLine(String content, LogType type)
            {
                Byte[] con = Encoding.Default.GetBytes("\r\n");
                _fs.Write(con, 0, con.Length);
            }
        }
        /// <summary>
        /// 修改所使用的时间类型
        /// </summary>
        public sealed partial class LogManager : IDisposable
        {
            /// <summary>
            /// 设置使用UTC时间
            /// </summary>
```

```
        public void UseUTCTime()
        {
                WriteLog = new Log(PrepareLogFile);
                WriteLog += OpenStream;
                WriteLog += AppendUTCTime;
                WriteLog += AppendSeperator;
                WriteLog += AppendComponentType;
                WriteLog += AppendSeperator;
                WriteLog += AppendType;
                WriteLog += AppendSeperator;
                WriteLog += AppendContent;
                WriteLog += AppendNewLine;
                WriteLog += CloseStream;
        }
        /// <summary>
        /// 设置使用本地时间
        /// </summary>
        public void UseLocalTime()
        {
                WriteLog = new Log(PrepareLogFile);
                WriteLog += OpenStream;
                WriteLog += AppendLocalTime;
                WriteLog += AppendSeperator;
                WriteLog += AppendComponentType;
                WriteLog += AppendSeperator;
                WriteLog += AppendType;
                WriteLog += AppendSeperator;
                WriteLog += AppendContent;
                WriteLog += AppendNewLine;
                WriteLog += CloseStream;
        }
}
```

日志管理类定义了一些列符合Log委托的方法，这些方法可以被添加到记录日志的委托对象之中，以构成整个日志记录的动作。在日后的扩展中，主要的工作也集中在添加新的符合Log委托定义的方法，并且将其添加到委托链上。

④ 在Main方法中调用LogManager的Log委托实例来写日志，LogManager只需要管理这个委托，负责分派任务即可。

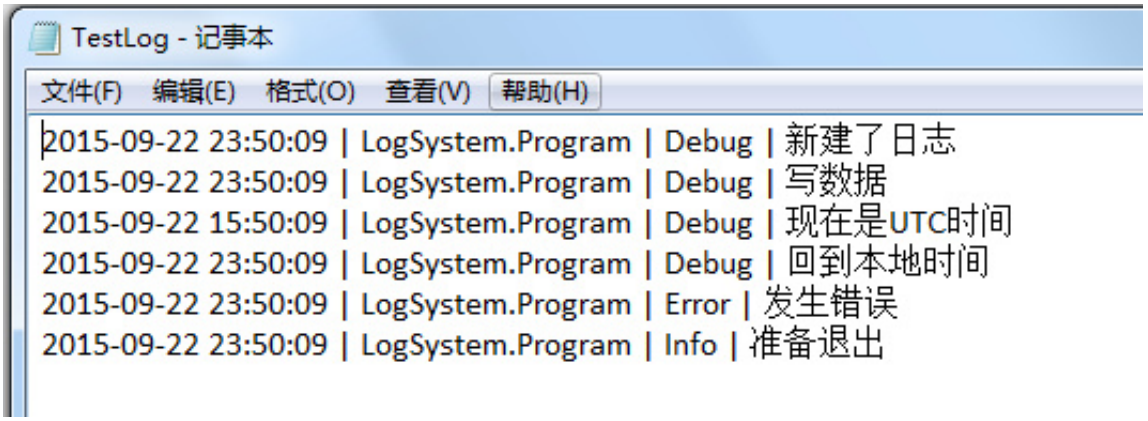
C#

```
class Program
        {
                static void Main(string[] args)
```

```
            {
                //使用日志
                using (LogManager logmanager =
                    new LogManager(Type.GetType("LogSystem.Program"),
"C:\\TestLog.txt"))
                {
                    logmanager.WriteLog("新建了日志", LogType.Debug);
                    logmanager.WriteLog("写数据", LogType.Debug);
                    logmanager.UseUTCTime();
                    logmanager.WriteLog("现在是UTC时间", LogType.Debug);
                    logmanager.UseLocalTime();
                    logmanager.WriteLog("回到本地时间", LogType.Debug);
                    logmanager.WriteLog("发生错误", LogType.Error);
                    logmanager.WriteLog("准备退出", LogType.Info);
                }
                Console.ReadKey();
            }
        }
```

代码中初始化委托成员的过程既是任务分派的过程，可以注意到LogManager的UseUTCTime和UseLocalTime方法都是被委托成员进行了重新的分配，也可以理解为任务的再分配。

下图是上述代码的执行结果，将日志信息写入了C:\TestLog.txt中：

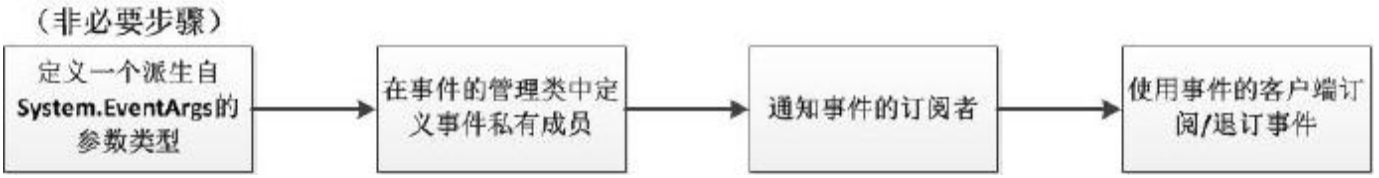



二、事件基础

事件这一名称对于我们.NET码农来说肯定不会陌生，各种技术框架例如WindowsForm、ASP.NET WebForm都会有事件这一名词，并且所有的定义都基本相同。在.NET中，事件和委托在本质上并没有太多的差异，实际环境下事件的运用却比委托更加广泛。

2.1 简述事件的基本使用方法

在Microsoft的产品文档上这样来定义的事件：事件是一种使对象或类能够提供通知的成员。客户端可以通过提供事件处理程序为相应的事件添加可执行代码。设计和使用事件的全过程大概包括以下几个步骤：




下面我们来按照规范的步骤来展示一个通过控制台输出事件的使用示例：

① 定义一个控制台事件ConsoleEvent的参数类型ConsoleEventArgs

```
C#/// <summary>
        /// 管理控制台，在输出前发送输出事件
        /// </summary>
        public class ConsoleManager
        {
                // 定义控制台事件成员对象
                public event EventHandler<ConsoleEventArgs> ConsoleEvent;
                /// <summary>
                /// 控制台输出
                /// </summary>
                public void ConsoleOutput(string message)
                {
                        // 发送事件
                        ConsoleEventArgs args = new ConsoleEventArgs(message);
                        SendConsoleEvent(args);
                        // 输出消息
                        Console.WriteLine(message);
                }
                /// <summary>
                /// 负责发送事件
                /// </summary>
                /// <param name="args">事件的参数</param>
                protected virtual void SendConsoleEvent(ConsoleEventArgs args)
                {
                        // 定义一个临时的引用变量，确保多线程访问时不会发生问题
                        EventHandler<ConsoleEventArgs> temp = ConsoleEvent;
                        if (temp != null)
                        {
                                temp(this, args);
                        }
                }
        }
```

③ 定义了事件的订阅者Log，在其中通过控制台时间的管理类公开的事件成员订阅其输出事件ConsoleEvent

```
C#
class Program
        {
                static void Main(string[] args)
                {
                        // 控制台事件管理者
                        ConsoleManager cm = new ConsoleManager();
                        // 控制台事件订阅者
                        Log log = new Log(cm);
                        cm.ConsoleOutput("测试控制台输出事件");
                        cm.ConsoleOutput("测试控制台输出事件");
                        cm.ConsoleOutput("测试控制台输出事件");
                        Console.ReadKey();
                }
        }
```

当该程序执行时，ConsoleManager负责在控制台输出测试的字符串消息，与此同时，订阅了控制台输出事件的Log类对象会在指定的日志文件中写入这些字符串消息。可以看出，这是一个典型的观察者模式的应用，也可以说事件为观察者模式提供了便利的实现基础。

2.2 事件和委托有神马联系？

事件的定义和使用方式与委托极其类似，那么二者又是何关系呢？经常听人说，委托本质是一个类型，而事件本质是一个特殊的委托类型的实例。关于这个解释，最好的办法莫过于通过查看原代码和编译后的IL代码进行分析。

① 回顾刚刚的代码，在ConsoleManager类中定义了一个事件成员

```
C#
1
public event EventHandler<ConsoleEventArgs> ConsoleEvent;
```

EventHandler是.NET框架中提供的一种标准的事件模式，它是一个特殊的泛型委托类型，通过查看元数据可以验证这一点：

```
C#
[Serializable]
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

正如上面代码所示，我们定义一个事件时，实际上是定义了一个特定的委托成员实例。该委托没有返回值，并且有两个参数：一个事件源和一个事件参数。而当事件的使用者订阅该事件时，其本质就是将事件的处理方法加入到委托链之中。

② 下面通过Reflector来查看一下事件ConsoleEvent的IL代码（中间代码），可以更方便地看到这一点：

首先，查看EventHandler的IL代码，可以看到在C#编译器编译delegate代码时，编译后是成为了一个class。

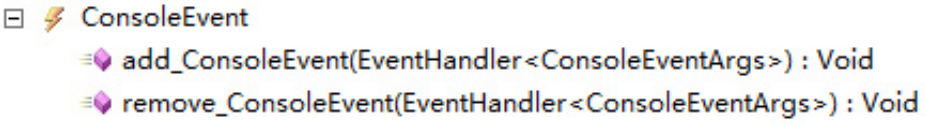
```
.class public auto ansi serializable sealed EventHandler<TEventArgs>
    extends System.MulticastDelegate
{
    .custom instance void __DynamicallyInvokableAttribute::.ctor()
    .method public hidebysig specialname rtspecialname instance void .ctor(
    {
        .custom instance void __DynamicallyInvokableAttribute::.ctor()
    }

    .method public hidebysig newslot virtual instance class System.IAsyncRes
    {
        .custom instance void __DynamicallyInvokableAttribute::.ctor()
    }

    .method public hidebysig newslot virtual instance void EndInvoke(class S
    {
        .custom instance void __DynamicallyInvokableAttribute::.ctor()
    }

    .method public hidebysig newslot virtual instance void Invoke(object sen
    {
        .custom instance void __DynamicallyInvokableAttribute::.ctor()
    }
```

其次，当C#编译器编译event代码时，会首先为类型添加一个EventHandler<T>的委托实例对象，然后为其增加一对add/remove方法用来实现从委托链中添加和移除方法的功能。




通过查看add_ConsoleEvent的IL代码，可以清楚地看到订阅事件的本质是调用Delegate的Combine方法将事件处理方法绑定到委托链中。

```
1
2
3
4
5
6
7
8
9
10
11
12
L_0000: ldarg.0
        L_0001: ldfld class [mscorlib]System.EventHandler`1&lt;class
ConsoleEventDemo.ConsoleEventArgs&gt; ConsoleEventDemo.ConsoleManager::ConsoleEvent
        L_0006: stloc.0
```

```
        L_0007: ldloc.0
        L_0008: stloc.1
        L_0009: ldloc.1
        L_000a: ldarg.1
        L_000b: call class [mscorlib]System.Delegate
[mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate, class
[mscorlib]System.Delegate)
        L_0010: castclass [mscorlib]System.EventHandler`1&lt;class
ConsoleEventDemo.ConsoleEventArgs&gt;
        L_0015: stloc.2
        L_0016: ldarg.0
        L_0017: ldflda class [mscorlib]System.EventHandler`1&lt;class
ConsoleEventDemo.ConsoleEventArgs&gt; ConsoleEventDemo.ConsoleManager::ConsoleEvent
```

> Summary：事件是一个特殊的委托实例，提供了两个供订阅事件和取消订阅的方法：add_event和remove_event，其本质都是基于委托链来实现。

2.3 如何设计一个带有很多事件的类型？

多事件的类型在实际应用中并不少见，尤其是在一些用户界面的类型中（例如在WindowsForm中的各种控件）。这些类型动辄将包含数十个事件，如果为每一个事件都添加一个事件成员，将导致无论使用者是否用到所有事件，每个类型对象都将占有很大的内存，那么对于系统的性能影响将不言而喻。事实上，.NET的开发小组运用了一种比较巧妙的方式来避免这一困境。

Solution：当某个类型具有相对较多的事件时，我们可以考虑显示地设计订阅、取消订阅事件的方法，并且把所有的委托链表存储在一个集合之中。这样做就能避免在类型中定义大量的委托成员而导致类型过大。

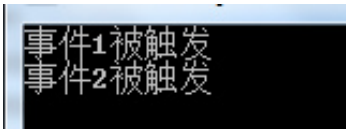
下面通过一个具体的实例来说明这一设计：

① 定义包含大量事件的类型之一：使用EventHandlerList成员来存储所有事件

C#

```
class Program
        {
                static void Main(string[] args)
                {
                        using(MultiEventClass mec = new MultiEventClass())
                        {
                                Customer customer = new Customer(mec);
                                mec.RiseEvent1();
                                mec.RiseEvent2();
                        }
                        Console.ReadKey();
                }
        }
```

最终运行结果如下图所示：




总结EventHandlerList的用法，在多事件类型中为每一个事件都定义了一套成员，包括事件的委托原型、事件的订阅和取消订阅方法，在实际应用中，可能需要定义事件专用的参数类型。这样的设计主旨在于改动包含多事件的类型，而订阅事件的客户并不会察觉这样的改动。设计本身不在于减少代码量，而在于有效减少多事件类型对象的大小。

2.4 如何使用事件模拟场景：猫叫->老鼠逃跑 & 主人惊醒

这是一个典型的观察者模式的应用场景，事件的发源在于猫叫这个动作，在猫叫之后，老鼠开始逃跑，而主人则会从睡梦中惊醒。可以发现，主人和老鼠这两个类型的动作相互之间没有联系，但都是由猫叫这一事件触发的。

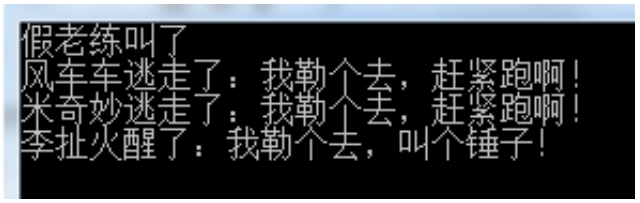
设计的大致思路在于，猫类包含并维护一个猫叫的动作，主人和老鼠的对象实例需要订阅猫叫这一事件，保证猫叫这一事件发生时主人和老鼠可以执行相应的动作。

（1）设计猫类，为其定义一个猫叫的事件CatCryEvent：

C#

```
class Program
        {
                static void Main(string[] args)
                {
                        Cat cat = new Cat("假老练");
                        Mouse mouse1 = new Mouse("风车车", cat);
                        Mouse mouse2 = new Mouse("米奇妙", cat);
                        Master master = new Master("李扯火", cat);
                        // 毛开始叫了，老鼠和主人有不同的反应
                        cat.CatCry();
                        Console.ReadKey();
                }
        }
```

这里定义了一只猫，两只老鼠与一个主人，当猫的CatCry方法被执行到时，会触发猫叫事件CatCryEvent，此时就会通知所有这一事件的订阅者。本场景的关键之处就在于主人和老鼠的动作应该完全由猫叫来触发。下面是场景模拟代码的运行结果：




三、反射基础3.1 反射的基本原理是什么？其实现的基石又是什么？

反射是一种动态分析程序集、模块、类型及字段等目标对象的机制，它的实现依托于元数据。元数据，

就是描述数据的数据。在CLR中，元数据就是对一个模块定义或引用的所有东西的描述系统。

3.2 .NET中提供了哪些类型实现反射？

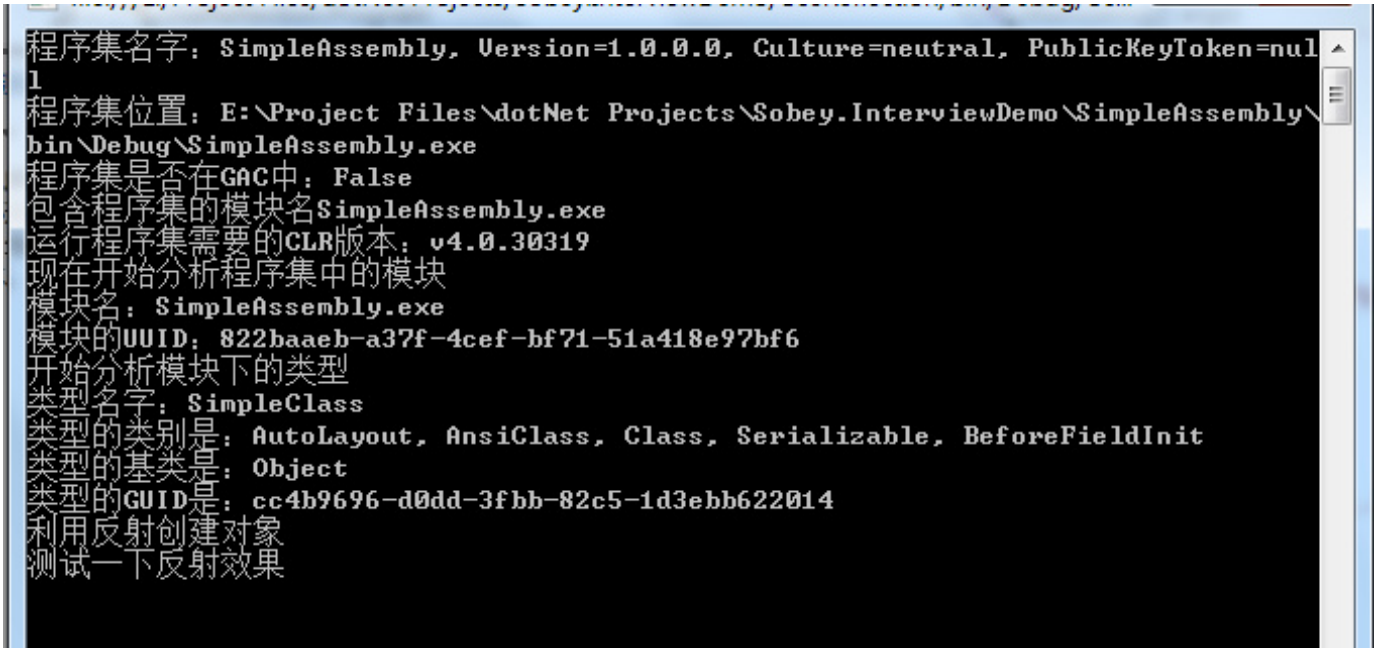
在.NET中，为我们提供了丰富的可以用来实现反射的类型，这些类型大多数都定义在System.Reflection命名空间之下，例如Assembly、Module等。利用这些类型，我们就可以方便地动态加载程序集、模块、类型、方法和字段等元素。

下面我们来看一个使用示例，首先是创建一个程序集SimpleAssembly，其中有一个类为SimpleClass：

C#

```
[PermissionSetAttribute(SecurityAction.Demand, Name = "FullTrust")]
        class Program
        {
                static void Main(string[] args)
                {
                        Assembly assembly =
Assembly.LoadFrom(@"..\..\..\SimpleAssembly\bin\Debug\SimpleAssembly.exe");
                        AnalyseHelper.AnalyzeAssembly(assembly);
                        // 创建一个程序集中的类型的对象
                        Console.WriteLine("利用反射创建对象");
                        string[] paras = { "测试一下反射效果" };
                        object obj = assembly.CreateInstance(assembly.GetModules()
[0].GetTypes()[0].ToString(), true, BindingFlags.CreateInstance, null, paras, null, null);
                        Console.WriteLine(obj);
                        Console.ReadKey();
                }
        }
```

上面的代码按照 程序集->模块->类型 三个层次的顺序来动态分析一个程序集，当然还可以继续递归类型内部的成员，最后通过CreateInstance方法来动态创建了一个类型，这些都是反射经常被用来完成的功能，执行结果如下图所示：

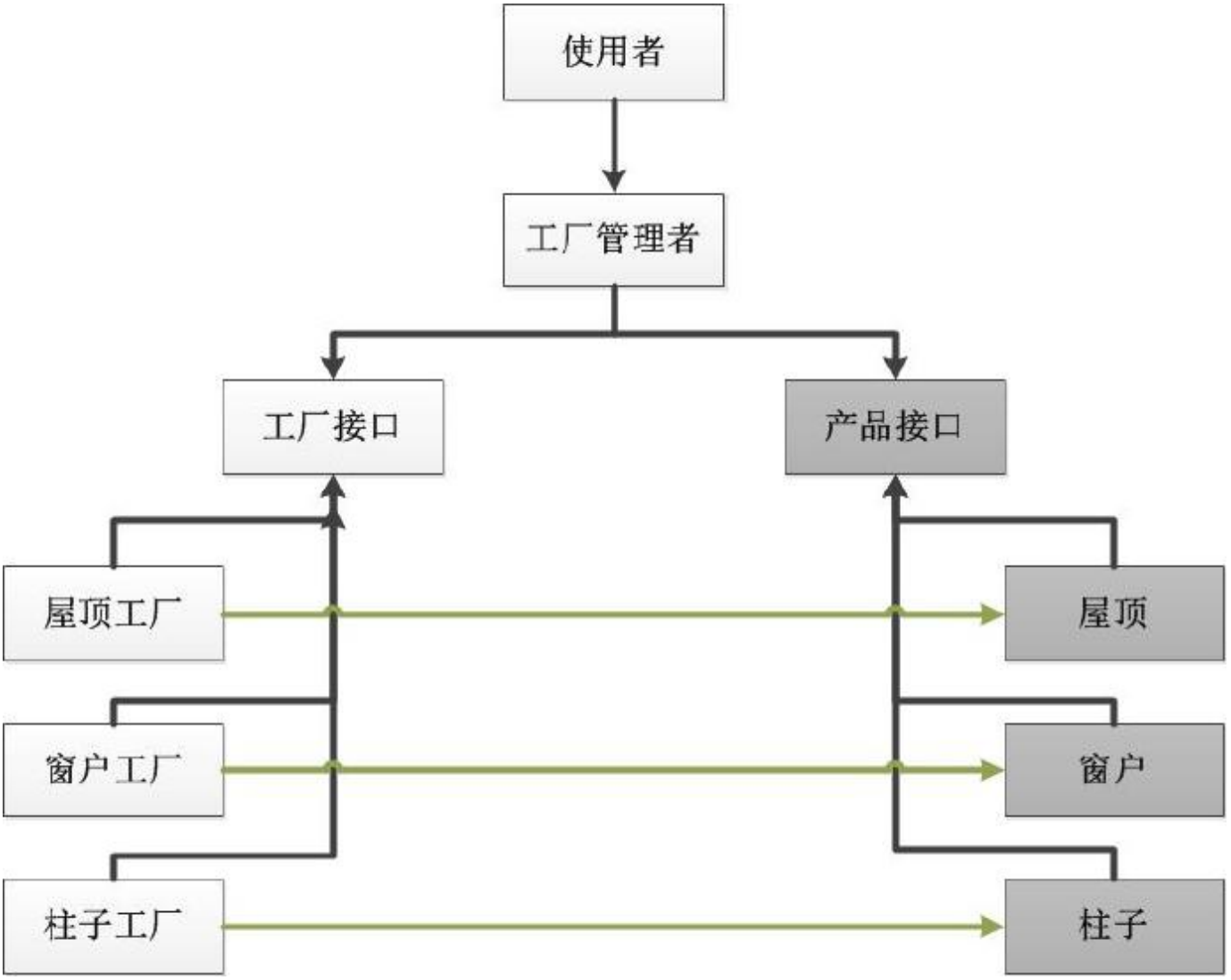



3.3 如何使用反射实现工厂模式？

工厂模式是一种比较常用的设计模式，其基本思想在于使用不同的工厂类型来打造不同产品的部件。例如，我们在打造一间屋子时，可能需要窗户、屋顶、门、房梁、柱子等零部件。有的屋子需要很多根柱子，而有的屋子又不需要窗户。在这样的需求下，就可以使用工厂模式。

（1）工厂模式的传统实现和其弊端

下图展示了针对屋子设计的传统工厂模式架构图：




上图的设计思路是：

①使用者告诉工厂管理者需要哪个产品部件；

②工厂管理者分析使用者传入的信息，生成合适的实现工厂接口的类型对象；

③通过工厂生产出相应的产品，返回给使用者一个实现了该产品接口的类型对象；

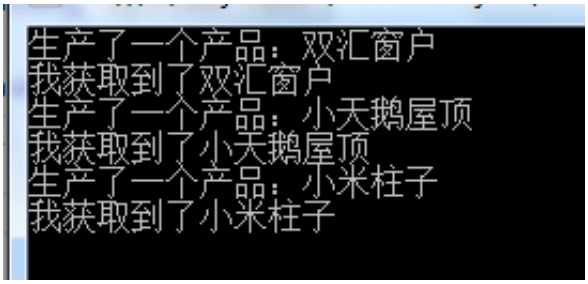
通过上述思路，实现代码如下：

①首先是定义工厂接口，产品接口与产品类型的枚举

```
C#
class Customer
        {
                static void Main(string[] args)
                {
                        // 根据需要获得不同的产品零件
                        IProduct window = FactoryManager.GetProduct(RoomParts.Window);
                        Console.WriteLine("我获取到了{0}",window.GetName());
                        IProduct roof = FactoryManager.GetProduct(RoomParts.Roof);
                        Console.WriteLine("我获取到了{0}", roof.GetName());
```

```
                IProduct pillar = FactoryManager.GetProduct(RoomParts.Pillar);
                Console.WriteLine("我获取到了{0}", pillar.GetName());
                Console.ReadKey();
            }
        }
```

在Customer类中，我们通过工厂管理类根据需要的不同零件类型获取到了不同的产品零件，其运行结果如下图所示：




当一个新的产品—地板需要被添加时，我们需要改的地方是：添加零件枚举记录、添加针对地板的工厂类、添加新地板产品类，修改工厂管理类（在switch中添加一条case语句），这样设计的优点在于无论添加何种零件，产品使用者都不需要关心内部的变动，可以一如既往地使用工厂管理类来得到希望的零件，而缺点也有以下几点：

①工厂管理类和工厂类族耦合；

②每次添加新的零件都需要添加一对工厂类和产品类，类型会越来越多；

（2）基于反射的工厂模式的实现

利用反射机制可以实现更加灵活的工厂模式，这一点体现在利用反射可以动态地获知一个产品由哪些零部件组成，而不再需要用一个switch语句来逐一地寻找合适的工厂。

①产品、枚举和以上一致，这里的改变主要在于添加了两个自定义的特性，这两个特性会被分别附加在产品类型和产品接口上：

```
C#/// <summary>
        /// 产品接口
        /// </summary>
        [ProductList(new Type[] { typeof(Roof), typeof(Window), typeof(Pillar) })]
        public interface IProduct
        {
            string GetName();
        }
        /// <summary>
        /// 屋顶
        /// </summary>
        [Product(RoomParts.Roof)]
        public class Roof : IProduct
```

```
        {
                // 实现接口，返回产品名字
                public string GetName()
                {
                        return "小天鹅屋顶";
                }
        }
        /// <summary>
        /// 窗户
        /// </summary>
        [Product(RoomParts.Window)]
        public class Window : IProduct
        {
                // 实现接口，返回产品名字
                public string GetName()
                {
                        return "双汇窗户";
                }
        }
        /// <summary>
        /// 柱子
        /// </summary>
        [Product(RoomParts.Pillar)]
        public class Pillar : IProduct
        {
                // 实现接口，返回产品名字
                public string GetName()
                {
                        return "小米柱子";
                }
        }
```

③下面是修改后的工厂类，由于使用了反射特性，这里一个工厂类型就可以生产所有的产品：

```
C#/// <summary>
        /// 工厂管理者
        /// </summary>
        public class FactoryManager
        {
                public static IProduct GetProduct(RoomParts part)
                {
                        // 一共只有一个工厂
                        Factory factory = new Factory();
```

```
                IProduct product = factory.Produce(part);
                Console.WriteLine("生产了一个产品：{0}", product.GetName());
                return product;
            }
        }
```

上述代码中最主要的变化在于两点：其一是工厂管理类不再需要根据不同的零件寻找不同的工厂，因为只有一个工厂负责处理所有的产品零件；其二是产品类型和产品接口应用了两个自定义特性，来方便工厂进行反射。ProductAttribute附加在产品类上，标注了当前类型代表了哪个产品零件。而ProductListAttribute则附加在产品接口之上，方便反射得知一共有多少产品零件。

这时需要添加一个新的地板产品零件类型时，我们需要做的是：添加零件枚举记录，添加代表地板的类型，修改添加在IProduct上的属性初始化参数（增加地板类型），可以看到这时调用者、工厂管理类和工厂都不再需要对新添加的零件进行改动，程序只需要添加必要的类型和枚举记录即可。当然，这样的设计也存在一定缺陷：反射的运行效率相对较低，在产品零件相对较多时，每生产一个产品就需要反射遍历这是一件相当耗时的工作。

四、特性基础

特性机制可以帮助程序员以申明的方式进行编程，而不再需要考虑实现的细节。

4.1 神马是特性？如何自定义一个特性？

（1）特性是什么？

特性是一种有别于普通命令式编程的编程方式，通常被称为申明式编程方式。所谓申明式编程方式就是指程序员只需要申明某个模块会有怎样的特性，而无需关心如何去实现。下面的代码就是特性在ASP.NET MVC中的基本使用方式：

C#

```
[HttpPost]
        public ActionResult Add(UserInfo userInfo)
        {
            if (ModelState.IsValid)
            {
                // To do fun
            }
            return RedirectToAction("Index");
        }
```

当一个特性被添加到某个元素上时，该元素就被认为具有了这个特性所代表的功能或性质，例如上述代码中Add方法在添加了HttpPost特性之后，就被认为只有遇到以POST的方式请求该方法时才会被执行。

> Note：特性在被编译器编译时，和传统的命令式代码不同，它会被以二进制数据的方式写入模块文件的元数据之中，而在运行时再被解读使用。特性也是经常被反射机制应用的元素，因为它本身是以元数据的形式存放的。

（2）如何自定义特性

除了直接使用.NET中内建的所有特性之外，我们也可以建立自己的特性来实现业务逻辑。在上面反射工厂的实现中就使用到了自定义特性。具体来说，定义一个特性的本质就是定义一个继承自System.Attribute类的类型，这样的类型就被编译器认为是一个特性类型。

下面我们看看如何自顶一个特性并使用该特性：

①定义一个继承自System.Attribute的类型MyCustomAttribute

C#

```
[MyCustom("UseMyCustomAttribute")]
        class UseMyCustomAttribute
        {
                static void Main(string[] args)
                {
                        Type t = typeof(UseMyCustomAttribute);
                        // 通过GetCustomAttributes方法得到自定义特性
                        object[] attrs = t.GetCustomAttributes(false);
                        MyCustomAttribute att = attrs[0] as MyCustomAttribute;
                        Console.WriteLine(att.ClassName);
                        Console.ReadKey();
                }
        }
```

为入口方法所在的类型UseMyCustomAttribute类添加了一个自定义特性，就可以在该类的方法中通过调用该类型的GetCustomAttributes方法获取所有添加到该类型的自定义特性数组，也就可以方便使用该自定义特性所具备的性质和能力（例如代码中的属性成员可以方便获取）。

> 关于自定义特性，有几点需要注意：
>
> - 虽然没有强制规定，但按照约定最好特性类型的名字都以Attribute结尾；
> - 在C#中为了方便起见，使用特性时都可以省略特性名字后的Attribute，例如上述代码中的[MyCustom(“UseMyCustomAttribute”)]代替了[MyCustomAttribute(“UseMyCustomAttribute”)]；
> - 特性类型自身也可以添加其他的特性；

4.2 .NET中特性可以在哪些元素上使用？

特性可以被用来使用到某个元素之上，这个元素可以是字段，也可以是类型。对于类、结构等元素，特性的使用可以添加在其定义的上方，而对于程序集、模块等元素的特性来说，则需要显式地告诉编译器这些特性的作用目标。例如，在C#中，通过目标关键字加冒号来告诉编译器的使用目标：

C#// 应用在程序集
[assembly:MyCustomAttribute]
// 应用在模块
[module: MyCustomAttribute]
// 应用在类型

[type: MyCustomAttribute]

我们在设计自定义特性时，往往都具有明确的针对性，例如该特性只针对类型、接口或者程序集，限制特性的使用目标可以有效地传递设计者的意图，并且可以避免不必要的错误使用特性而导致的元数据膨胀。AttributeUsage特性就是用来限制特性使用目标元素的，它接受一个AttributeTargets的枚举对象作为输入来告诉AttributeUsage西望望对特性做何种限定。例如上面展示的一个自定义特性，使用了限制范围：

C#

```
[AttributeUsage(AttributeTargets.Class)]
public class MyCustomAttribute : Attribute
{
        .....
}
```

> Note：一般情况下，自定义特性都会被限制适用范围，我们也应该养成这样的习惯，为自己设计的特性加上AttributeUsage特性，很少会出现使用在所有元素上的特性。即便是可以使用在所有元素上，也应该显式地申明[AttributeUsage(AttributesTargets.All)]来提高代码的可读性。

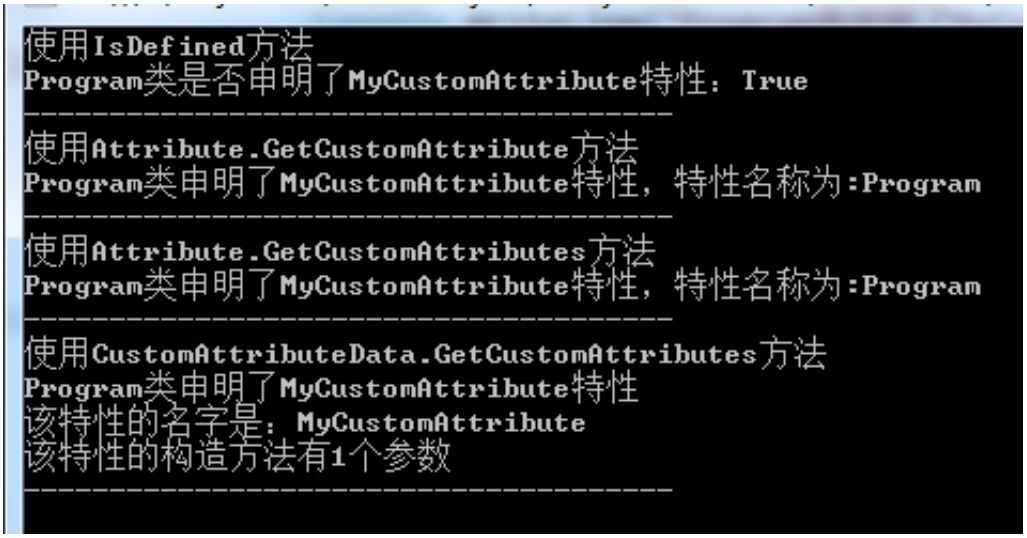
4.3 如何获知一个元素是否申明了某个特性？

在.NET中提供了很多的方法来查询一个元素是否申明了某个特性，每个方法都有不同的使用场合，但是万变不离其宗，都是基于反射机制来实现的。

首先，还是以上面的MyCustomAttribute特性为例，新建一个入口方法类Program：

```
C#// 使用CustomAttributeData.GetCustomAttributes方法
        IList<CustomAttributeData> attList =
CustomAttributeData.GetCustomAttributes(thisClass);
        if (attList.Count > 0)
        {
                Console.WriteLine("Program类申明了MyCustomAttribute特性");
                // 注意：这里可以对特性进行分析，但无法得到其实例
                CustomAttributeData attData = attList[0];
                Console.WriteLine("该特性的名字是：{0}",
attData.Constructor.DeclaringType.Name);
                Console.WriteLine("该特性的构造方法有{0}个参数",
attData.ConstructorArguments.Count);
        }
```

下图是四种方式的执行结果：




这四种方法各有其特点，但都可以实现查询某个元素是否申明了某个特性的这一功能。其中，可以看到第（4）种方式，可以对特性进行分析，但无法得到其实例。另外，自定义特性被申明为sealed表示不可继承，这是因为在特性被检查时，无法分别制定特性和其派生特性，这一点需要我们注意。

4.4 一个元素是否可以重复申明同一个特性？

对于有些业务逻辑来说，一个特性反复地申明在同一个元素上市没有必要的，但同时对于另一些逻辑来说，又非常有必要对同一元素多次申明同一特性。很幸运，.NET的特性机制完美支持了这一类业务需求。

当一个特性申明了AttributeUsage特性并且显式地将AllowMultiple属性设置为true时，该特性就可以在同一元素上多次申明，否则的话编译器将报错。

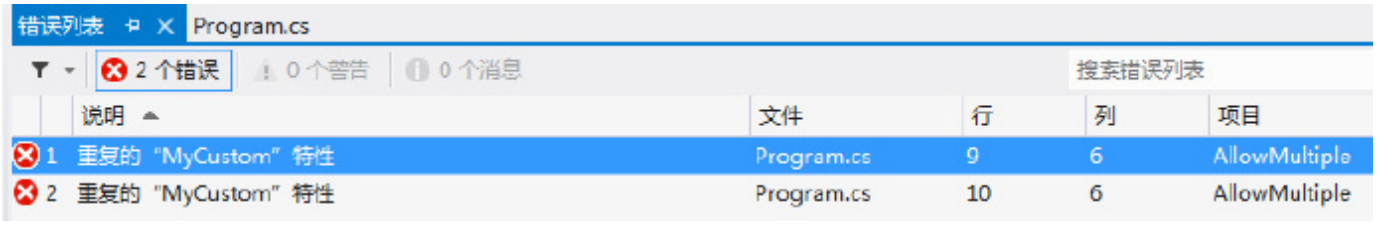
例如下面一段代码，类型Program多次申明了MyCustomAttribute特性：

C#

```
[MyCustom("Class1")]
        [MyCustom("Class2")]
        [MyCustom("Class3")]
        public class Program
        {
                public static void Main(string[] args)
                {
                }
        }
        /// <summary>
        /// 一个自定义特性MyCustomAttribute
        /// </summary>
        [AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
        public class MyCustomAttribute : Attribute
        {
                private string className;
                public MyCustomAttribute(string className)
```

```
        {
            this.className = className;
        }
        // 一个只读属性ClassName
        public string ClassName
        {
            get
            {
                return className;
            }
        }
    }
```

通常情况下，重复申明同一特性往往会传入不同的参数。此外，如果不显式地设置AllowMultiple属性时多次申明同一特性会如何呢？在这种情况下，编译器将会认为自定义特性不能多次申明在同一元素上，会出现以下的编译错误：




参考资料

（1）朱毅，《进入IT企业必读的200个.NET面试题》

（2）张子阳，《.NET之美：.NET关键技术深入解析》

（3）王涛，《你必须知道的.NET》

资源 - 优秀的工具资源导航
翻译 - 翻译传播优秀的外文文章
文章 - 国内外的精选文章
设计 - UI,网页，交互和用户体验
iOS - 专注iOS技术分享
安卓 - 专注Android技术分享
前端 - JavaScript, HTML5, CSS
Java - 专注Java技术分享
Python - 专注Python技术分享