细细品味C#

——抽象类、接口、委托、反射

精

华

集

锦

csAxp

虾皮工作室

http://www.cnblogs.com/xia520pi/

2011年7月29日

目录

1、抽象类与抽象方法

1.1、版权声明

文章出处：http://www.cnblogs.com/wuhui369161243/archive/2009/03/29/1424677.html
文章作者：Me 丶紫龙

1.2、内容详情

朋友曾问我抽象类是否至少要有一个抽象方法，我查了很多资料，结果都是：“抽象类允许（但不要求）抽象类包含抽象成员”。但是一个抽象类里不写抽象方法就没有意义了，既然如此，还不如直接写个普通类？ 在一个抽象类里可以不声明抽象方法，这在语法上是没问题的，但实际来说，这样是没有任何意义的。也就是说，你为什么会选择写一个抽象类呢？当然是为了想某个方法能够被 OVERRIDE，以实现多态。后来查找 MSDN 结果如下：abstract 修饰符可以和类、方法、属性、索引器及事件一起使用。

在类声明中使用 abstract 修饰符以指示类只能是其他类的基类。

【抽象类】具有以下特性：

- 抽象类不能实例化。
- 抽象类可以包含抽象方法和抽象访问器。
- 不能用 sealed 修饰符修改抽象类，这意味着该类不能被继承。
- 从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实实现。

在方法或属性声明中使用 abstract 修饰符以指示此方法或属性不包含实现。

【抽象方法】具有以下特性：

- 抽象方法是隐式的 virtual 方法。
- 只允许在抽象类中使用抽象方法声明。
- 因为抽象方法声明不提供实实现，所以没有方法体；方法声明只是以一个分号结束，并且在签名后没有大括号 ({ })。例如：
- public abstract void MyMethod();
- 实现由 overriding 方法提供，它是非抽象类的成员。
- 在抽象方法声明中使用 static 或 virtual 修饰符是错误的。

除了在声明和调用语法上不同外，抽象属性的行为与抽象方法一样。

- 在静态属性上使用 abstract 修饰符是错误的。
- 在派生类中，通过包括使用 override 修饰符的属性声明可以重写抽象的继承属性。

抽象类必须为所有接口成员提供实现。

MSDN 中 C#语言规范的：10.1.1.1 抽象类。此文如下：
abstract 修饰符用于表示所修饰的类是不完整的，并且它只能用作基类。抽象类与非抽象类在以下方面是不同的：

- 抽象类不能直接实例化，并且对抽象类使用 new 运算符是编译时错误。虽然一些变量和值在编译时的类型可以是抽象的，但是这样的变量和值必须或者为 null，或者含有对非抽象类的实例的引用（此非抽象类是从抽象类派生的）。
- 允许（但不要求）抽象类包含抽象成员。
- 抽象类不能被密封。

当从抽象类派生非抽象类时，这些非抽象类必须具体实现所继承的所有抽象成员，从而重写那些抽象成员。在下面的示例中

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
{
    public void G() {}
}

class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

抽象类 A 引入抽象方法 F。类 B 引入另一个方法 G，但由于它不提供 F 的实现，B 也必须声明为抽象类。类 C 重写 F，并提供一个具体实现。由于 C 中没有了抽象成员，因此可以（但并非必须）将 C 声明为非抽象类。

有关抽象类和抽象方法的问题，我个人也做了一些总结，如下：
abstract 关键字用于将类指定为抽象类，这些抽象类可以派生出其他类。

- 一个抽象类可以同时包含抽象方法和非抽象方法。
- 抽象方法的目的在于指定派生类必须实现与这一方法关联的行为。
- 抽象方法只在派生类中真正实现，这表明抽象方法只存放函数原型(方法的返回类型，使用的名称及参数)，而不涉及主体代码。
- 如果父类被声明为抽象类，并存在未实现的抽象方法，那么子类就必须实现父类中所有的 abstract 成员，除非该类也是抽象的。
- 抽象类不能被实例化，使用 override 关键字可在派生类中实现抽象方法，经 override 声明重写的方法，其签名必须与 override 方法的签名一致。

例如：

```
 1abstract class A
 2{
 3       public abstract void F();
 4}
 5
 6abstract class B: A
 7{
 8       public void G()
 9       {}
10}
11
12class C: B
13{
14       public override void F()
15       {
16              // actual implementation of F
17       }
18}
19
```

说明：抽象类 A 引入抽象方法 F。类 B 引入另一个方法 G，但由于它不提供 F 的实现，B 也必须声明为抽象类。类 C 重写 F，并提供一个具体实现。由于 C 中没有了抽象成员，因此可以（但并非必须）将 C 声明为非抽象类。

2、接口基础教程

2.1、版权声明

文章出处：http://www.cnblogs.com/aspxer/category/97963.html
文章作者：喝酒当喝汤

2.2、内容详情

2.2.1、接口概述

接口（interface）用来定义一种程序的协定。实现接口的类或者结构要与接口的定义严格一致。有了这个协定，就可以抛开编程语言的限制（理论上）。接口可以从多个基接口继承，而类或结构可以实现多个接口。接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或接口必须提供的成员。

接口好比一种模版，这种模版定义了对象必须实现的方法，其目的就是让这些方法可以作为接口实例被引用。接口不能被实例化。类可以实现多个接口并且通过这些实现的接口被索引。接口变量只能索引实现该接口的类的实例。例子：

```
interface IMyExample {
    string this[int index] { get ; set ; }
    event EventHandler Even ;
    void Find(int value) ;
    string Point { get ; set ; }
}
public delegate void EventHandler(object sender, Event e) ;
```

上面例子中的接口包含一个索引 this、一个事件 Even、一个方法 Find 和一个属性 Point。

接口可以支持多重继承。就像在下例中，接口"IComboBox"同时从"ITextBox"和"IListBox"继承。

```
interface IControl {
void Paint( ) ;
}
interface ITextBox: IControl {
void SetText(string text) ;
}
```

```
interface IListBox: IControl {
void SetItems(string[] items) ;
}
interface IComboBox: ITextBox, IListBox { }
```

类和结构可以多重实例化接口。就像在下例中，类"EditBox"继承了类"Control"，同时从"IDataBound"和"IControl"继承。

```
interface IDataBound {
   void Bind(Binder b) ;
}
public class EditBox: Control, IControl, IDataBound {
   public void Paint( ) ;
   public void Bind(Binder b) {...}
}
```

在上面的代码中，"Paint"方法从"IControl"接口而来；"Bind"方法从"IDataBound"接口而来，都以"public"的身份在"EditBox"类中实现。

说明：

1）C#中的接口是独立于类来定义的。这与 C++模型是对立的，在 C++中接口实际上就是抽象基类。

2）接口和类都可以继承多个接口。

3）而类可以继承一个基类，接口根本不能继承类。这种模型避免了 C++的多继承问题，C++中不同基类中的实现可能出现冲突。因此也不再需要诸如虚拟继承和显式作用域这类复杂机制。C#的简化接口模型有助于加快应用程序的开发。

4）一个接口定义一个只有抽象成员的引用类型。C#中一个接口实际所做的，仅仅只存在着方法标志，但根本就没有执行代码。这就暗示了不能实例化一个接口，只能实例化一个派生自该接口的对象。

5）接口可以定义方法、属性和索引。所以，对比一个类，接口的特殊性是：当定义一个类时，可以派生自多重接口，而你只能可以从仅有的一个类派生。

接口与组件

接口描述了组件对外提供的服务。在组件和组件之间、组件和客户之间都通过接口进行交互。因此组件一旦发布，它只能通过预先定义的接口来提供合理的、一致的服务。这种接口定义之间的稳定性使客户应用开发者能够构造出坚固的应用。一个组件可以实现多个组件接口，而一个特定的组件接口也可以被多个组件来实现。

组件接口必须是能够自我描述的。这意味着组件接口应该不依赖于具体的实现，将实现和接口分离彻底消除了接口的使用者和接口的实现者之间的耦合关系，增强了信息的封装程度。同时这也要求组件接口必须使用一种与组件实现无关的语言。目前组件接口的描述标准是 IDL 语言。

由于接口是组件之间的协议，因此组件的接口一旦被发布，组件生产者就应该尽可能地保持接口不变，任何对接口语法或语义上的改变，都有可能造成现有组件与客户之间的联系遭到破坏。

每个组件都是自主的，有其独特的功能，只能通过接口与外界通信。当一个组件需要提供新的服务时，可以通过增加新的接口来实现。不会影响原接口已存在的客户。而新的客户可以重新选择新的接口来获得服务。

组件化程序设计

组件化程序设计方法继承并发展了面向对象的程序设计方法。它把对象技术应用于系统设计，对面向对象的程序设计的实现过程作了进一步的抽象。我们可以把组件化程序设计方法用作构造系统的体系结构层次的方法，并且可以使用面向对象的方法很方便地实现组件。

组件化程序设计强调真正的软件可重用性和高度的互操作性。它侧重于组件的产生和装配，这两方面一起构成了组件化程序设计的核心。组件的产生过程不仅仅是应用系统的需求，组件市场本身也推动了组件的发展，促进了软件厂商的交流与合作。组件的装配使得软件产品可以采用类似于搭积木的方法快速地建立起来，不仅可以缩短软件产品的开发周期，同时也提高了系统的稳定性和可靠性。

组件程序设计的方法有以下几个方面的特点：

1） 编程语言和开发环境的独立性；
2） 组件位置的透明性；
3） 组件的进程透明性；
4） 可扩充性；
5） 可重用性；
6） 具有强有力的基础设施；
7） 系统一级的公共服务；

C#语言由于其许多优点，十分适用于组件编程。但这并不是说 C#是一门组件编程语言，也不是说 C#提供了组件编程的工具。我们已经多次指出，组件应该具有与编程语言无关的特性。请读者记住这一点：组件模型是一种规范，不管采用何种程序语言设计组件，都必须遵守这一规范。比如组装计算机的例子，只要各个厂商为我们提供的配件规格、接口符合统一的标准，这些配件组合起来就能协同工作，组件编程也是一样。我们只是说，利用C#语言进行组件编程将会给我们带来更大的方便。

2.2.2、定义接口

从技术上讲，接口是一组包含了函数型方法的数据结构。通过这组数据结构，客户代码可以调用组件对象的功能。

定义接口的一般形式为：

```
[attributes] [modifiers] interface identifier [:base-list] {interface-body}[;]
```

说明：

1）attributes（可选）：附加的定义性信息。

2）modifiers（可选）： 允许使用的修饰符有 new 和四个访问修饰符。分别是：new、public、protected、internal、 private。在一个接口定义中同一修饰符不允许出现多次，new 修饰符只能出现在嵌套接口中，表示覆盖了继承而来的同名成员。The public, protected, internal, and private 修饰符定义了对接口的访问权限。

3）指示器和事件。

4）identifier：接口名称。

5）base-list（可选）:包含一个或多个显式基接口的列表，接口间由逗号分隔。

6）interface-body：对接口成员的定义。

7）接口可以是命名空间或类的成员，并且可以包含下列成员的签名： 方法、属性、索引器 。

8）一个接口可从一个或多个基接口继承。

接口这个概念在 C#和 Java 中非常相似。接口的关键词是 interface，一个接口可以扩展一个或者多个其他接口。按照惯例，接口的名字以大写字母"I"开头。下面的代码是 C#接口的一个例子，它与 Java 中的接口完全一样：

```
interface IShape {
   void Draw ( ) ;
}
```

如果你从两个或者两个以上的接口派生，父接口的名字列表用逗号分隔，如下面的代码所示：

```
interface INewInterface: IParent1, IParent2 { }
```

然而，与 Java 不同，C#中的接口不能包含域（Field）。另外还要注意，在 C#中，接口内的所有方法默认都是公用方法。在 Java 中，方法定义可以带有 public 修饰符（即使这并非必要），但在 C#中，显式为接口的方法指定 public 修饰符是非法的。例如，下面的 C#接口将产生一个编译错误。

```
interface IShape { public void Draw( ) ; }
```

下面的例子定义了一个名为 IControl 的接口，接口中包含一个成员方法 Paint：

```
interface IControl {
   void Paint( ) ;
}
```

在下例中，接口 IInterface 从两个基接口 IBase1 和 IBase2 继承：

```
interface IInterface: IBase1, IBase2 {
   void Method1( ) ;
   void Method2( ) ;
}
```

接口可由类实现。实现的接口的标识符出现在类的基列表中。例如：

```
class Class1: Iface1, Iface2 {
   // class  成员。
}
```

类的基列表同时包含基类和接口时，列表中首先出现的是基类。例如：

```
class ClassA: BaseClass, Iface1, Iface2 {
   // class 成员。
}
```

以下的代码段定义接口 IFace，它只有一个方法：

```
interface IFace {
   void ShowMyFace( ) ;
}
```

不能从这个定义实例化一个对象，但可以从它派生一个类。因此，该类必须实现 ShowMyFace 抽象方法：

```
class CFace:IFace
{
   public void ShowMyFace( ) {
      Console.WriteLine(" implementation " ) ;
   }
}
```

基接口

一个接口可以从零或多个接口继承，那些被称为这个接口的显式基接口。当一个接口有比零多的显式基接口时，那么在接口的定义中的形式为，接口标识符后面跟着由一个冒号":"和一个用逗号","分开的基接口标识符列表。

接口基：

接口类型列表说明：

1）一个接口的显式基接口必须至少同接口本身一样可访问。例如，在一个公共接口的基接口中指定一个私有或内部的接口是错误的。

2）一个接口直接或间接地从它自己继承是错误的。

3）接口的基接口都是显式基接口，并且是它们的基接口。换句话说，基接口的集合完全由显式基接口和它们的显式基接口等等组成。在下面的例子中

```
interface IControl {
   void Paint( ) ;
}
interface ITextBox: IControl {
   void SetText(string text) ;
}
interface IListBox: IControl {
   void SetItems(string[] items) ;
}
interface IComboBox: ITextBox, IListBox { }
```

IComboBox 的基接口是 IControl, ITextBox, 和 IlistBox。

4）一个接口继承它的基接口的所有成员。换句话说，上面的接口 IComboBox 就像 Paint 一样继承成员 SetText 和 SetItems。

5）一个实现了接口的类或结构也隐含地实现了所有接口的基接口。

接口主体

一个接口的接口主体定义接口的成员。

2.2.3、定义接口成员

接口可以包含一个和多个成员，这些成员可以是方法、属性、索引指示器和事件，但不能是常量、域、操作符、构造函数或析构函数，而且不能包含任何静态成员。接口定义创建新的定义空间，并且接口定义直 接包含的接口成员定义将新成员引入该定义空间。

说明：

1）接口的成员是从基接口继承的成员和由接口本身定义的成员。

2）接口定义可以定义零个或多个成员。接口的成员必须是方法、属性、事件或索引器。接口不能包含常数、字段、运算符、实例构造函数、析构函数或类型，也不能包含任何种类的静态成员。

3）定义一个接口，该接口对于每种可能种类的成员都包含一个：方法、属性、事件和索引器。

4）接口成员默认访问方式是 public。接口成员定义不能包含任何修饰符，比如成员

定义前不能加 abstract，public，protected，internal，private，virtual，override 或 static 修饰符。

5）接口的成员之间不能相互同名。继承而来的成员不用再定义，但接口可以定义与继承而来的成员同名的成员，这时我们说接口成员覆盖了继承而来的成员，这不会导致错误，但编译器会给出一个警告。关闭警告提示的方式是在成员定义前加上一个 new 关键字。但如果没有覆盖父接口中的成员，使用 new 关键字会导致编译器发出警告。

6）方法的名称必须与同一接口中定义的所有属性和事件的名称不同。此外，方法的签名必须与同一接口中定义的所有其他方法的签名不同。

7）属性或事件的名称必须与同一接口中定义的所有其他成员的名称不同。

8）一个索引器的签名必须区别于在同一接口中定义的其他所有索引器的签名。

9）接口方法声明中的属性（attributes），返回类型（return-type），标识符（identifier），和形式参数列表（formal-parameter-lis）与一个类的方法声明中的那些有相同的意义。一个接口方法声明不允许指定一个方法主体，而声明通常用一个分号结束。

10）接口属性声明的访问符与类属性声明的访问符相对应，除了访问符主体通常必须用分号。因此，无论属性是读写、只读或只写，访问符都完全确定。

11）接口索引声明中的属性（attributes），类型（type），和形式参数列表（formal-parameter-list）与类的索引声明的那些有相同的意义。

下面例子中接口 IMyTest 包含了索引指示器、事件 E、 方法 F、 属性 P 这些成员：

```
interface IMyTest{
    string this[int index] { get; set; }
    event EventHandler E ;
    void F(int value) ;
    string P { get; set; }
}
public delegate void EventHandler(object sender, EventArgs e) ;
```

下面例子中接口 IStringList 包含每个可能类型成员的接口：一个方法，一个属性，一个事件和一个索引。

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

接口成员的全权名

使用接口成员也可采用全权名（fully qualified name）。接口的全权名称是这样构成的。接口名加小圆点"." 再跟成员名比如对于下面两个接口：

```
interface IControl {
    void Paint( ) ;
}
interface ITextBox: IControl {
    void GetText(string text) ;
}
```

其中 Paint 的全权名是 IControl.Paint，GetText 的全权名是 ITextBox. GetText。当然，全权名中的成员名称必须是在接口中已经定义过的，比如使用 ITextBox.Paint.就是不合理的。

如果接口是名字空间的成员，全权名还必须包含名字空间的名称。

```
namespace System
{
    public interface IDataTable {
        object Clone( ) ;
    }
}
```

那么 Clone 方法的全权名是 System. IDataTable.Clone。

2.2.4、访问接口

对接口成员的访问

对接口方法的调用和采用索引指示器访问的规则与类中的情况也是相同的。如果底层成员的命名与继承而来的高层成员一致，那么底层成员将覆盖同名的高层成员。但由于接口支持多继承，在多继承中，如果两个父接口含有同名的成员，这就产生了二义性（这也正是C#中取消了类的多继承机制的原因之一），这时需要进行显式的定义：

```
using System ;
interface ISequence {
    int Count { get; set; }
}
interface IRing {
    void Count(int i) ;
}
interface IRingSequence: ISequence, IRing { }
    class CTest {
        void Test(IRingSequence rs) {
            //rs.Count(1) ; 错误, Count 有二义性
            //rs.Count = 1; 错误, Count 有二义性
```

```
        ((ISequence)rs).Count = 1; //  正确
        ((IRing)rs).Count(1) ; //  正确调用 IRing.Count
      }
}
```

上面的例子中，前两条语句 rs .Count(1)和 rs .Count = 1 会产生二义性，从而导致编译时错误，因此必须显式地给 rs 指派父接口类型，这种指派在运行时不会带来额外的开销。

再看下面的例子：

```
using System ;
interface IInteger {
    void Add(int i) ;
}
interface IDouble {
    void Add(double d) ;
}
interface INumber: IInteger, IDouble {}
    class CMyTest {
    void Test(INumber Num) {
       // Num.Add(1) ; 错误
       Num.Add(1.0) ; //  正确
       ((IInteger)n).Add(1) ; //  正确
       ((IDouble)n).Add(1) ; //  正确
    }
}
```

调用 Num.Add(1) 会导致二义性，因为候选的重载方法的参数类型均适用。但是，调用 Num.Add(1.0) 是允许的，因为 1.0 是浮点数参数类型与方法 IInteger.Add（）的参数类型不一致，这时只有 IDouble.Add 才是适用的。不过只要加入了显式的指派，就决不会产生二义性。

接口的多重继承的问题也会带来成员访问上的问题。例如：

```
interface IBase {
    void FWay(int i) ;
}
interface ILeft: IBase {
    new void FWay (int i) ;
}
interface IRight: IBase
{ void G( ) ; }
interface IDerived: ILeft, IRight { }
```

```
class CTest {
    void Test(IDerived d) {
       d. FWay (1) ; //  调用 ILeft. FWay
       ((IBase)d). FWay (1) ; //  调用 IBase. FWay
       ((ILeft)d). FWay (1) ; //  调用 ILeft. FWay
       ((IRight)d). FWay (1) ; //  调用 IBase. FWay
    }
}
```

上例中，方法 IBase.FWay 在派生的接口 ILeft 中被 Ileft 的成员方法 FWay 覆盖了。所以对 d. FWay (1)的调用实际上调用了。虽然从 IBase-> IRight-> IDerived 这条继承路径上来看,ILeft.FWay 方法是没有被覆盖的。我们只要记住这一点：一旦成员被覆盖以后，所有对其的访问都被覆盖以后的成员"拦截"了。

类对接口的实现

前面我们已经说过，接口定义不包括方法的实现部分。接口可以通过类或结构来实现。我们主要讲述通过类来实现接口。用类来实现接口时，接口的名称必须包含在类定义中的基类列表中。

下面的例子给出了由类来实现接口的例子。其中 ISequence 为一个队列接口，提供了向队列尾部添加对象的成员方法 Add( )，IRing 为一个循环表接口，提供了向环中插入对象的方法 Insert(object obj)，方法返回插入的位置。类 RingSquence 实现了接口 ISequence 和接口 IRing。

```
using System ;
interface ISequence {
    object Add( ) ;
}
interface ISequence {
    object Add( ) ;
}
interface IRing {
    int Insert(object obj) ;
}
class RingSequence: ISequence, IRing
{
    public object Add( ) {…}
    public int Insert(object obj) {…}
}
```

如果类实现了某个接口，类也隐式地继承了该接口的所有父接口，不管这些父接口有没有在类定义的基类表中列出。看下面的例子：

```
using System ;
interface IControl {
    void Paint( );
}
interface ITextBox: IControl {
    void SetText(string text);
}
interface IListBox: IControl {
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox { }
```

这里，接口 IcomboBox 继承了 ItextBox 和 IlistBox。类 TextBox 不仅实现了接口 ITextBox，还实现了接口 ITextBox 的父接口 IControl。

前面我们已经看到，一个类可以实现多个接口。再看下面的例子：

```
interface IDataBound {
    void Bind(Binder b);
}
public class EditBox: Control, IControl, IDataBound {
    public void Paint( );
    public void Bind(Binder b) {...}
}
```

类 EditBox 从类 Control 中派生并且实现了 Icontrol 和 IdataBound。在前面的例子中接口 Icontrol 中的 Paint 方法和 IdataBound 接口中的 Bind 方法都用类 EditBox 中的公共成员实现。C#提供一种实现这些方法的可选择的途径，这样可以使执行这些的类避免把这些成员设定为公共的。接口成员可以用有效的名称来实现。例如，类 EditBox 可以改作方法 Icontrol.Paint 和 IdataBound.Bind 来来实现。

```
public class EditBox: IControl, IDataBound {
    void IControl.Paint( ) {...}
    void IDataBound.Bind(Binder b) {...}
}
```

因为通过外部指派接口成员实现了每个成员，所以用这种方法实现的成员称为外部接口成员。外部接口成员可以只是通过接口来调用。例如，Paint 方法中 EditBox 的实现可以只是通过创建 Icontrol 接口来调用。

```
class Test {
    static void Main( ) {
      EditBox editbox = new EditBox( );
```

```
        editbox.Paint( ); //错误: EditBox 没有 Paint 事件
        IControl control = editbox;
        control.Paint( ); // 调用 EditBox 的 Paint 事件
    }
}
```

上例中，类 EditBox 从 Control 类继承并同时实现了 IControl and IDataBound 接口。EditBox 中的 Paint 方法来自 IControl 接口，Bind 方法来自 IDataBound 接口，二者在 EditBox 类中都作为公有成员实现。当然，在 C# 中我们也可以选择不作为公有成员实现接口。

如果每个成员都明显地指出了被实现的接口，通过这种途径被实现的接口我们称之为显式接口成员（explicit interface member）。 用这种方式我们改写上面的例子：

```
public class EditBox: IControl, IDataBound {
    void IControl.Paint( ) {…}
    void IDataBound.Bind(Binder b) {…}
}
```

显式接口成员只能通过接口调用。例如：

```
class CTest {
    static void Main( ) {
        EditBox editbox = new EditBox( ) ;
        editbox.Paint( ) ; //错误:不同的方法
        IControl control = editbox;
        control.Paint( ) ; //调用 EditBox 的 Paint 方法
    }
}
```

上述代码中对 editbox.Paint( )的调用是错误的，因为 editbox 本身并没有提供这一方法。control.Paint( )是正确的调用方式。

注释：接口本身不提供所定义的成员的实现，它仅仅说明这些成员，这些成员必须依靠实现接口的类或其它接口的支持。

2.2.5、实现接口

显式实现接口成员

为了实现接口，类可以定义显式接口成员执行体（Explicit interface member implementations）。显式接口成员执行体可以是一个方法、一个属性、一个事件或者是一个索引指示器的定义，定义与该成员对应的全权名应保持一致。

```
using System ;
interface ICloneable {
    object Clone( ) ;
}
interface IComparable {
    int CompareTo(object other) ;
}
class ListEntry: ICloneable, IComparable {
    object ICloneable.Clone( ) {…}
    int IComparable.CompareTo(object other) {…}
}
```

上面的代码中 ICloneable.Clone 和 IComparable.CompareTo 就是显式接口成员执行体。

说明：

1）不能在方法调用、属性访问以及索引指示器访问中通过全权名访问显式接口成员执行体。事实上，显式接口成员执行体只能通过接口的实例，仅仅引用接口的成员名称来访问。

2）显式接口成员执行体不能使用任何访问限制符，也不能加上 abstract, virtual, override 或 static 修饰符。

3）显式接口成员执行体和其他成员有着不同的访问方式。因为不能在方法调用、属性访问以及索引指示器访问中通过全权名访问，显式接口成员执行体在某种意义上是私有的。但它们又可以通过接口的实例访问，也具有一定的公有性质。

4）只有类在定义时，把接口名写在了基类列表中，而且类中定义的全权名、类型和返回类型都与显式接口成员执行体完全一致时，显式接口成员执行体才是有效的，例如：

```
class Shape: ICloneable {
object ICloneable.Clone( ) {…}
int IComparable.CompareTo(object other) {…}
}
```

使用显式接口成员执行体通常有两个目的：

1）因为显式接口成员执行体不能通过类的实例进行访问，这就可以从公有接口中把接口的实现部分单独分离开。如果一个类只在内部使用该接口，而类的使用者不会直接使用到该接口，这种显式接口成员执行体就可以起到作用。

2）显式接口成员执行体避免了接口成员之间因为同名而发生混淆。如果一个类希望对名称和返回类型相同的接口成员采用不同的实现方式，这就必须要使用到显式接口成员执行体。如果没有显式接口成员执行体，那么对于名称和返回类型不同的接口成员，类也无法进行实现。

下面的定义是无效的，因为 Shape 定义时基类列表中没有出现接口 IComparable。

```
class Shape: ICloneable
{
object ICloneable.Clone( ) {…}
}
class Ellipse: Shape
{
object ICloneable.Clone( ) {…}
}
```

在 Ellipse 中定义 ICloneable.Clone 是错误的，因为 Ellipse 即使隐式地实现了接口 ICloneable，ICloneable 仍然没有显式地出现在 Ellipse 定义的基类列表中。

接口成员的全权名必须对应在接口中定义的成员。如下面的例子中，Paint 的显式接口成员执行体必须写成 IControl.Paint。

```
using System ;
interface IControl
{
    void Paint( ) ;
}
interface ITextBox: IControl
{
    void SetText(string text) ;
}
class TextBox: ITextBox
{
    void IControl.Paint( ) {…}
    void ITextBox.SetText(string text) {…}
}
```

实现接口的类可以显式实现该接口的成员。当显式实现某成员时，不能通过类实例访问该成员，而只能通过该接口的实例访问该成员。显式接口实现还允许程序员继承共享相同成员名的两个接口，并为每个接口成员提供一个单独的实现。

下面例子中同时以公制单位和英制单位显示框的尺寸。Box 类继承 IEnglishDimensions 和 IMetricDimensions 两个接口，它们表示不同的度量衡系统。两个接口有相同的成员名 Length 和 Width。

程序清单 1 DemonInterface.cs

```
interface IEnglishDimensions {
float Length ( ) ;
float Width ( ) ;
```

```
}
interface IMetricDimensions {
float Length ( ) ;
float Width ( ) ;
}
class Box : IEnglishDimensions, IMetricDimensions {
float lengthInches ;
float widthInches ;
public Box(float length, float width) {
lengthInches = length ;
widthInches = width ;
}
float IEnglishDimensions.Length( ) {
return lengthInches ;
}
float IEnglishDimensions.Width( ) {
return widthInches ;
}
float IMetricDimensions.Length( ) {
return lengthInches * 2.54f ;
}
float IMetricDimensions.Width( ) {
return widthInches * 2.54f ;
}
public static void Main( ) {
//定义一个实类对象 "myBox"::
Box myBox = new Box(30.0f, 20.0f);
// 定义一个接口" eDimensions"::
IEnglishDimensions eDimensions = (IEnglishDimensions) myBox;
IMetricDimensions mDimensions = (IMetricDimensions) myBox;
// 输出:
System.Console.WriteLine(" Length(in): {0}", eDimensions.Length( ));
System.Console.WriteLine(" Width (in): {0}", eDimensions.Width( ));
System.Console.WriteLine(" Length(cm): {0}", mDimensions.Length( ));
System.Console.WriteLine(" Width (cm): {0}", mDimensions.Width( ));
}
}
```

输出：Length(in): 30，Width (in): 20，Length(cm): 76.2，Width (cm): 50.8

代码讨论：如果希望默认度量采用英制单位，请正常实现 Length 和 Width 这两个方法，并从 IMetricDimensions 接口显式实现 Length 和 Width 方法：

```
public float Length( ) {
return lengthInches ;
}
public float Width( ){
return widthInches;
}
float IMetricDimensions.Length( ) {
return lengthInches * 2.54f ;
}
float IMetricDimensions.Width( ) {
return widthInches * 2.54f ;
}
```

这种情况下，可以从类实例访问英制单位，而从接口实例访问公制单位：

```
System.Console.WriteLine("Length(in): {0}", myBox.Length( )) ;
System.Console.WriteLine("Width (in): {0}", myBox.Width( )) ;
System.Console.WriteLine("Length(cm): {0}", mDimensions.Length( )) ;
System.Console.WriteLine("Width (cm): {0}", mDimensions.Width( )) ;
```

继承接口实现

接口具有不变性，但这并不意味着接口不再发展。类似于类的继承性，接口也可以继承和发展。

注意：接口继承和类继承不同，首先，类继承不仅是说明继承，而且也是实现继承；而接口继承只是说明继承。也就是说，派生类可以继承基类的方法实现，而派生的接口只继承了父接口的成员方法说明，而没有继承父接口的实现，其次，C#中类继承只允许单继承，但是接口继承允许多继承，一个子接口可以有多个父接口。

接口可以从零或多个接口中继承。从多个接口中继承时，用":"后跟被继承的接口名字，多个接口名之间用","分割。被继承的接口应该是可以访问得到的，比如从 private 类型或 internal 类型的接口中继承就是不允许的。接口不允许直接或间接地从自身继承。和类的继承相似，接口的继承也形成接口之间的层次结构。

请看下面的例子：

```
using System ;
interface IControl {
void Paint( ) ;
}
interface ITextBox: IControl {
void SetText(string text) ;
```

```
}
interface IListBox: IControl {
void SetItems(string[] items) ;
}
interface IComboBox: ITextBox, IListBox { }
```

对一个接口的继承也就继承了接口的所有成员，上面的例子中接口 ITextBox 和 IListBox 都从接口 IControl 中继承，也就继承了接口 IControl 的 Paint 方法。接口 IComboBox 从接口 ITextBox 和 IListBox 中继承，因此它应该继承了接口 ITextBox 的 SetText 方法和 IListBox 的 SetItems 方法，还有 IControl 的 Paint 方法。
一个类继承了所有被它的基本类提供的接口实现程序。

不通过显式的实现一个接口，一个派生类不能用任何方法改变它从它的基本类继承的接口映射。例如，在声明中

```
interface IControl {
void Paint( );
}
class Control: IControl {
public void Paint( ) {...}
}
class TextBox: Control {
new public void Paint( ) {...}
}
```

TextBox 中的方法 Paint 隐藏了 Control 中的方法 Paint ，但是没有改变从 Control.Paint 到 IControl.Paint 的映射，而通过类实例和接口实例调用 Paint 将会有下面的影响

```
Control c = new Control( ) ;
TextBox t = new TextBox( ) ;
IControl ic = c ;
IControl it = t ;
c.Paint( ) ; // 影响 Control.Paint( ) ;
t.Paint( ) ; // 影响 TextBox.Paint( ) ;
ic.Paint( ) ; // 影响 Control.Paint( ) ;
it.Paint( ) ; // 影响 Control.Paint( ) ;
```

但是，当一个接口方法被映射到一个类中的虚拟方法，派生类就不可能覆盖这个虚拟方法并且改变接口的实现函数。例如，把上面的声明重新写为

```
interface IControl {
void Paint( ) ;
}
```

```
class Control: IControl {
public virtual void Paint( ) {...}
}
class TextBox: Control {
public override void Paint( ) {...}
}
```

就会看到下面的结果：

```
Control c = new Control( ) ;
TextBox t = new TextBox( ) ;
IControl ic = c ;
IControl it = t ;
c.Paint( ) ; // 影响 Control.Paint( );
t.Paint( ) ; // 影响 TextBox.Paint( );
ic.Paint( ) ; // 影响 Control.Paint( );
it.Paint( ) ; // 影响 TextBox.Paint( );
```

由于显式接口成员实现程序不能被声明为虚拟的，就不可能覆盖一个显式接口成员实现程序。一个显式接口成员实现程序调用另外一个方法是有效的，而另外的那个方法可以被声明为虚拟的以便让派生类可以覆盖它。例如:

```
interface IControl {
    void Paint( ) ;
}
class Control: IControl {
    void IControl.Paint( ) { PaintControl( ); }
    protected virtual void PaintControl( ) {...}
}
class TextBox: Control {
    protected override void PaintControl( ) {...}
}
```

这里，从 Control 继承的类可以通过覆盖方法 PaintControl 来对 IControl.Paint 的实现程序进行特殊化。

重新实现接口

我们已经介绍过，派生类可以对基类中已经定义的成员方法进行重载。类似的概念引入到类对接口的实现中来，叫做接口的重实现（re-implementation）。继承了接口实现的类可以对接口进行重实现。这个接口要求是在类定义的基类列表中出现过的。对接口的重实现也必须严格地遵守首次实现接口的规则，派生的接口映射不会对为接口的重实现所建立的接口映射产生任何影响。

下面的代码给出了接口重实现的例子：

```
interface IControl {
    void Paint( ) ;
    class Control: IControl
    void IControl.Paint( ) {…}
    class MyControl: Control, IControl
    public void Paint( ) {}
}
```

实际上就是：Control 把 IControl.Paint 映射到了 Control.IControl.Paint 上，但这并不影响在 MyControl 中的重实现。在 MyControl 中的重实现中，IControl.Paint 被映射到 MyControl.Paint 之上。

在接口的重实现时，继承而来的公有成员定义和继承而来的显式接口成员的定义参与到接口映射的过程。

```
using System ;
interface IMethods {
    void F( ) ;
    void G( ) ;
    void H( ) ;
    void I( ) ;
}
class Base: IMethods {
    void IMethods.F( ) { }
    void IMethods.G( ) { }
    public void H( ) { }
    public void I( ) { }
}
class Derived: Base, IMethods {
    public void F( ) { }
    void IMethods.H( ) { }
}
```

这里，接口 IMethods 在 Derived 中的实现把接口方法映射到了 Derived.F,Base.IMethods.G, Derived.IMethods.H, 还有 Base.I。前面我们说过，类在实现一个接口时，同时隐式地实现了该接口的所有父接口。同样，类在重实现一个接口时同时，隐式地重实现了该接口的所有父接口。

```
using System ;
interface IBase {
    void F( ) ;
```

```
}
interface IDerived: IBase {
    void G( ) ;
}
class C: IDerived {
    void IBase.F( ) {
    //对 F 进行实现的代码…
}
void IDerived.G( ) {
    //对 G 进行实现的代码…
}
}
class D: C, IDerived {
    public void F( ) {
    //对 F 进行实现的代码…
}
public void G( ) {
    //对 G 进行实现的代码…
}
}
```

这里，对 IDerived 的重实现也同样实现了对 IBase 的重实现，把 IBase.F 映射到了 D.F。

映射接口

类必须为在基类表中列出的所有接口的成员提供具体的实现。在类中定位接口成员的实现称之为接口映射（interface mapping ）。

映射，数学上表示一一对应的函数关系。接口映射的含义也是一样，接口通过类来实现，那么对于在接口中定义的每一个成员，都应该对应着类的一个成员来为它提供具体的实现。

类的成员及其所映射的接口成员之间必须满足下列条件：

1）如果 A 和 B 都是成员方法，那么 A 和 B 的名称、类型、形参表（包括参数个数和每一个参数的类型）都应该是一致的。

2）如果 A 和 B 都是属性，那么 A 和 B 的名称、类型应当一致，而且 A 和 B 的访问器也是类似的。但如果 A 不是显式接口成员执行体，A 允许增加自己的访问器。

3）如果 A 和 B 都是时间那么 A 和 B 的名称、类型应当一致。

4）如果 A 和 B 都是索引指示器，那么 A 和 B 的类型、形参表（包括参数个数和每一个参数的类型）应当一致。而且 A 和 B 的访问器也是类似的。但如果 A 不是显式接口成员执行体，A 允许增加自己的访问器。

那么，对于一个接口成员，怎样确定由哪一个类的成员来实现呢？即一个接口成员映射的是哪一个类的成员？在这里，我们叙述一下接口映射的过程。假设类 C 实现了一个接

口 IInterface，Member 是接口 IInterface 中的一个成员，在定位由谁来实现接口成员 Member，即 Member 的映射过程是这样的：

1）如果 C 中存在着一个显式接口成员执行体，该执行体与接口 IInterface 及其成员 Member 相对应，则由它来实现 Member 成员。

2）如果条件（1）不满足，且 C 中存在着一个非静态的公有成员，该成员与接口成员 Member 相对应，则由它来实现 Member 成员。

3）如果上述条件仍不满足，则在类 C 定义的基类列表中寻找一个 C 的基类 D，用 D 来代替 C。

4）重复步骤 1-- 3 ，遍历 C 的所有直接基类和非直接基类，直到找到一个满足条件的类的成员。

5）如果仍然没有找到，则报告错误。

下面是一个调用基类方法来实现接口成员的例子。类 Class2 实现了接口 Interface1，类 Class2 的基类 Class1 的成员也参与了接口的映射，也就是说类 Class2 在对接口 Interface1 进行实现时，使用了类 Class1 提供的成员方法 F 来实现接口 Interface1 的成员方法 F：

```
interface Interface1 {
    void F( ) ;
}
class Class1 {
    public void F( ) { }
    public void G( ) { }
}
class Class2: Class1, Interface1 {
    new public void G( ) {}
}
```

注意：接口的成员包括它自己定义的成员，而且包括该接口所有父接口定义的成员。在接口映射时，不仅要对接口定义体中显式定义的所有成员进行映射，而且要对隐式地从父接口那里继承来的所有接口成员进行映射。

在进行接口映射时，还要注意下面两点：

1）在决定由类中的哪个成员来实现接口成员时，类中显式说明的接口成员比其它成员优先实现。

2）使用 Private、protected 和 static 修饰符的成员不能参与实现接口映射。例如：

```
interface ICloneable {
    object Clone( ) ;
}
class C: ICloneable {
    object ICloneable.Clone( ) {…}
    public object Clone( ) {…}
}
```

例子中成员 ICloneable.Clone 称为接口 ICloneable 的成员 Clone 的实现者，因为它是

显式说明的接口成员，比其它成员有着更高的优先权。

如果一个类实现了两个或两个以上名字、类型和参数类型都相同的接口，那么类中的一个成员就可能实现所有这些接口成员：

```
interface IControl {
    void Paint( ) ;
}
interface IForm {
    void Paint( ) ;
}
class Page: IControl, IForm {
    public void Paint( ) {…}
}
```

这里，接口 IControl 和 IForm 的方法 Paint 都映射到了类 Page 中的 Paint 方法。当然也可以分别用显式的接口成员分别实现这两个方法：

```
interface IControl {
    void Paint( ) ;
}
interface IForm {
    void Paint( ) ;
}
class Page: IControl, IForm {
    public void IControl.Paint( ) {
    //具体的接口实现代码
}
public void IForm.Paint( ) {
    //具体的接口实现代码
}
}
```

上面的两种写法都是正确的。但是如果接口成员在继承中覆盖了父接口的成员，那么对该接口成员的实现就可能必须映射到显式接口成员执行体。看下面的例子：

```
interface IBase {
    int P { get; }
}
interface IDerived: IBase {
    new int P( ) ;
}
```

接口 IDerived 从接口 IBase 中继承，这时接口 IDerived 的成员方法覆盖了父接口的成员方法。因为这时存在着同名的两个接口成员，那么对这两个接口成员的实现如果不采用显式接口成员执行体，编译器将无法分辨接口映射。所以，如果某个类要实现接口 IDerived，在类中必须至少定义一个显式接口成员执行体。采用下面这些写法都是合理的：

//一：对两个接口成员都采用显式接口成员执行体来实现

```
lass C: IDerived {
    int IBase.P
    get
    { //具体的接口实现代码 }
      int IDerived.P( ){
      //具体的接口实现代码 }
    }
```

//二：对 Ibase 的接口成员采用显式接口成员执行体来实现

```
class C: IDerived {
    int IBase.P
    get {//具体的接口实现代码}
      public int P( ){
      //具体的接口实现代码 }
    }
```

//三：对 IDerived 的接口成员采用显式接口成员执行体来实现

```
class C: IDerived{
    public int P
    get {//具体的接口实现代码}
    int IDerived.P( ){
    //具体的接口实现代码}
}
```

另一种情况是，如果一个类实现了多个接口，这些接口又拥有同一个父接口，这个父接口只允许被实现一次。

```
using System ;
interface IControl {
    void Paint( ) ;
    interface ITextBox: IControl {
    void SetText(string text) ;
}
interface IListBox: IControl {
```

```
    void SetItems(string[] items) ;
}
class ComboBox: IControl, ITextBox, IListBox {
    void IControl.Paint( ) {…}
    void ITextBox.SetText(string text) {…}
    void IListBox.SetItems(string[] items) {…}
}
```

上面的例子中，类 ComboBox 实现了三个接口：IControl，ITextBox 和 IListBox。如果认为 ComboBox 不仅实现了 IControl 接口，而且在实现 ITextBox 和 IListBox 的同时，又分别实现了它们的父接口 IControl。实际上，对接口 ITextBox 和 IListBox 的实现，分享了对接口 IControl 的实现。

我们对 C#的接口有了较全面的认识，基本掌握了怎样应用 C#的接口编程，但事实上，C#的不仅仅应用于.NET 平台，它同样支持以前的 COM，可以实现 COM 类到.NET 类的转换，如 C#调用 API。

2.2.6、接口转换

C#中不仅支持.Net 平台，而且支持 COM 平台。为了支持 COM 和.Net，C# 包含一种称为属性的独特语言特性。一个属性实际上就是一个 C# 类，它通过修饰源代码来提供元信息。属性使 C# 能够支持特定的技术，如 COM 和 .Net，而不会干扰语言规范本身。C#提供将 COM 接口转换为 C#接口的属性类。另一些属性类将 COM 类转换为 C# 类。执行这些转换不需要任何 IDL 或类工厂。

现在部署的任何 COM 组件都可以在接口转换中使用。通常情况下，所需的调整是完全自动进行的。

特别是，可以使用运行时可调用包装 (RCW) 从 .NET 框架访问 COM 组件。此包装将 COM 组件提供的 COM 接口转换为与 .NET 框架兼容的接口。对于 OLE 自动化接口，RCW 可以从类型库中自动生成；对于非 OLE 自动化接口，开发人员可以编写自定义 RCW，手动将 COM 接口提供的类型映射为与 .NET 框架兼容的类型。

使用 ComImport 引用 COM 组件

COM Interop 提供对现有 COM 组件的访问，而不需要修改原始组件。使用 ComImport 引用 COM 组件常包括下面 几个方面的问题：

1）创建 COM 对象。
2）确定 COM 接口是否由对象实现。
3）调用 COM 接口上的方法。
4）实现可由 COM 客户端调用的对象和接口。

创建 COM 类包装

要使 C# 代码引用 COM 对象和接口，需要在 C# 中包含 COM 接口的定义。完成此操作的最简单方法是使用 TlbImp.exe（类型库导入程序），它是一个包括在 .NET 框架 SDK 中的命令行工具。TlbImp 将 COM 类型库转换为 .NET 框架元数据，从而有效地创建一个可以从任何托管语言调用的托管包装。用 TlbImp 创建的 .NET 框架元数据可以通过 /R 编译器选项包括在 C# 内部版本中。如果使用 Visual Studio 开发环境，则只需添加对 COM 类型库的引用，将为您自动完成此转换。

TlbImp 执行下列转换：
1）COM coclass 转换为具有无参数构造函数的 C# 类。
2）COM 结构转换为具有公共字段的 C# 结构。

检查 TlbImp 输出的一种很好的方法是运行 .NET 框架 SDK 命令行工具 Ildasm.exe（Microsoft 中间语言反汇编程序）来查看转换结果。

虽然 TlbImp 是将 COM 定义转换为 C# 的首选方法，但也不是任何时候都可以使用它（例如，在没有 COM 定义的类型库时或者 TlbImp 无法处理类型库中的定义时，就不能使用该方法）。在这些情况下，另一种方法是使用 C# 属性在 C# 源代码中手动定义 COM 定义。创建 C# 源映射后，只需编译 C# 源代码就可产生托管包装。

执行 COM 映射需要理解的主要属性包括：
1）ComImport：它将类标记为在外部实现的 COM 类。
2）Guid：它用于为类或接口指定通用唯一标识符 (UUID)。
3）InterfaceType，它指定接口是从 IUnknown 还是从 IDispatch 派生。
4）PreserveSig，它指定是否应将本机返回值从 HRESULT 转换为 .NET 框架异常。

COM coclass 在 C# 中表示为类。这些类必须具有与其关联的 ComImport 属性。下列限制适用于这些类：

1）类不能从任何其他类继承。
2）类不能实现任何接口。
3）类还必须具有为其设置全局唯一标识符 (GUID) 的 Guid 属性。

以下示例在 C# 中声明一个 coclass：

```
// 声明一个 COM 类 FilgraphManager
[ComImport, Guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
class FilgraphManager
{ }
```

C# 编译器将添加一个无参数构造函数，可以调用此构造函数来创建 COM coclass 的实例。

创建 COM 对象

COM coclass 在 C# 中表示为具有无参数构造函数的类。使用 new 运算符创建该类的实例等效于在 C# 中调用 CoCreateInstance。使用以上定义的类，就可以很容易地实例化此类：

```
class MainClass
{
public static void Main()
{
FilgraphManager filg = new FilgraphManager();
}
}
```

声明 COM 接口

COM 接口在 C# 中表示为具有 ComImport 和 Guid 属性的接口。它不能在其基接口列表中包含任何接口，而且必须按照方法在 COM 接口中出现的顺序声明接口成员函数。

在 C# 中声明的 COM 接口必须包含其基接口的所有成员的声明，IUnknown 和 IDispatch 的成员除外（.NET 框架将自动添加这些成员）。从 IDispatch 派生的 COM 接口必须用 InterfaceType 属性予以标记。
从 C# 代码调用 COM 接口方法时，公共语言运行库必须封送与 COM 对象之间传递的参数和返回值。对于每个 .NET 框架类型均有一个默认类型，公共语言运行库将使用此默认类型在 COM 调用间进行封送处理时封送。例如，C# 字符串值的默认封送处理是封送到本机类型 LPTSTR（指向 TCHAR 字符缓冲区的指针）。可以在 COM 接口的 C# 声明中使用 MarshalAs 属性重写默认封送处理。

在 COM 中，返回成功或失败的常用方法是返回一个 HRESULT，并在 MIDL 中有一个标记为"retval"、用于方法的实际返回值的 out 参数。在 C#（和 .NET 框架）中，指示已经发生错误的标准方法是引发异常。
默认情况下，.NET 框架为由其调用的 COM 接口方法在两种异常处理类型之间提供自动映射。

返回值更改为标记为 retval 的参数的签名（如果方法没有标记为 retval 的参数，则为 void）。

标记为 retval 的参数从方法的参数列表中剥离。

任何非成功返回值都将导致引发 System.COMException 异常。

此示例显示用 MIDL 声明的 COM 接口以及用 C# 声明的同一接口（注意这些方法使用 COM 错误处理方法）。

下面是接口转换的 C#程序：

```
using System.Runtime.InteropServices;
// 声明一个 COM 接口 IMediaControl
[Guid("56A868B1-0AD4-11CE-B03A-0020AF0BA770"),
InterfaceType(ComInterfaceType.InterfaceIsDual)]
interface IMediaControl // 这里不能列出任何基接口
{
void Run();
void Pause();
void Stop();
void GetState( [In] int msTimeout, [Out] out int pfs);
void RenderFile(
[In, MarshalAs(UnmanagedType.BStr)] string strFilename);
void AddSourceFilter(
[In, MarshalAs(UnmanagedType.BStr)] string strFilename,
[Out, MarshalAs(UnmanagedType.Interface)] out object ppUnk);
[return : MarshalAs(UnmanagedType.Interface)]
object FilterCollection();
[return : MarshalAs(UnmanagedType.Interface)]
object RegFilterCollection();
void StopWhenReady();
}
```

若要防止 HRESULT 翻译为 COMException，请在 C# 声明中将 PreserveSig(true) 属性附加到方法。

下面是一个使用 C# 映射媒体播放机 COM 对象的程序。

程序清单 2 DemonCOM.cs

```
using System;
using System.Runtime.InteropServices;
namespace QuartzTypeLib
{
//声明一个 COM 接口 IMediaControl，此接口来源于媒体播放机 COM 类
[Guid("56A868B1-0AD4-11CE-B03A-0020AF0BA770"),
InterfaceType(ComInterfaceType.InterfaceIsDual)]
interface IMediaControl
{ //列出接口成员
void Run();
void Pause();
void Stop();
void GetState( [In] int msTimeout, [Out] out int pfs);
void RenderFile(
[In, MarshalAs(UnmanagedType.BStr)] string strFilename);
```

```
void AddSourceFilter(
[In, MarshalAs(UnmanagedType.BStr)] string strFilename,
[Out, MarshalAs(UnmanagedType.Interface)]
out object ppUnk);
[return: MarshalAs(UnmanagedType.Interface)]
object FilterCollection();
[return: MarshalAs(UnmanagedType.Interface)]
object RegFilterCollection();
void StopWhenReady();
}
//声明一个 COM 类:
[ComImport, Guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
class FilgraphManager //此类不能再继承其它基类或接口
{
//这里不能有任何代码 ， 系统自动增加一个缺省的构造函数
}
}
class MainClass
{
public static void Main(string[] args)
{
//命令行参数:
if (args.Length != 1)
{
DisplayUsage();
return;
}
String filename = args[0];
if (filename.Equals("/?"))
{
DisplayUsage();
return;
}
// 声明 FilgraphManager 的实类对象：
QuartzTypeLib.FilgraphManager graphManager =new QuartzTypeLib.FilgraphManager();
//声明 IMediaControl 的实类对象：:
QuartzTypeLib.IMediaControl mc =(QuartzTypeLib.IMediaControl)graphManager;
// 调用 COM 的方法：
mc.RenderFile(filename);
//运行文件.
mc.Run();
//暂借停.
Console.WriteLine("Press Enter to continue.");
Console.ReadLine();
```

```
}
private static void DisplayUsage()
{ // 显示
Console.WriteLine("媒体播放机: 播放 AVI 文件.");
Console.WriteLine("使用方法: VIDEOPLAYER.EXE 文件名");
}
}
```

运行示例：

若要显示影片示例 Clock.avi，请使用以下命令：

```
interop2 %windir%\clock.avi
```

这将在屏幕上显示影片，直到按 ENTER 键停止。

在 .NET 框架程序中通过 DllImport 使用 Win32 API

.NET 框架程序可以通过静态 DLL 入口点的方式来访问本机代码库。DllImport 属性用于指定包含外部方法的实现的 dll 位置。

DllImport 属性定义如下：

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class DllImportAttribute: System.Attribute
    {
        public DllImportAttribute(string dllName) {...}
        public CallingConvention CallingConvention;
        public CharSet CharSet;
        public string EntryPoint;
        public bool ExactSpelling;
        public bool PreserveSig;
        public bool SetLastError;
        public string Value { get {...} }
    }
}
```

说明：

1）DllImport 只能放置在方法声明上。

2）DllImport 具有单个定位参数：指定包含被导入方法的 dll 名称的 dllName 参数。

3）DllImport 具有五个命名参数：

a、CallingConvention 参数指示入口点的调用约定。如果未指定 CallingConvention，则使用默认值 CallingConvention.Winapi。

b、CharSet 参数指示用在入口点中的字符集。如果未指定 CharSet，则使用默认值 CharSet.Auto。

c、EntryPoint 参数给出 dll 中入口点的名称。如果未指定 EntryPoint，则使用方法本身的名称。

d、ExactSpelling 参数指示 EntryPoint 是否必须与指示的入口点的拼写完全匹配。如果未指定 ExactSpelling，则使用默认值 false。

e、PreserveSig 参数指示方法的签名应当被保留还是被转换。当签名被转换时，它被转换为一个具有 HRESULT 返回值和该返回值的一个名为 retval 的附加输出参数的签名。如果未指定 PreserveSig，则使用默认值 true。

f、SetLastError 参数指示方法是否保留 Win32"上一错误"。如果未指定 SetLastError，则使用默认值 false。

4）它是一次性属性类。

5）此外，用 DllImport 属性修饰的方法必须具有 extern 修饰符。

下面是 C# 调用 Win32 MessageBox 函数的示例：

```
using System;
using System.Runtime.InteropServices;
class MainApp
{ //通过 DllImport 引用 user32.dll 类。MessageBox 来自于 user32.dll 类
    [DllImport("user32.dll", EntryPoint="MessageBox")]
    public static extern int MessageBox(int hWnd, String strMessage, String strCaption, uint uiType);
    public static void Main()
    {
       MessageBox( 0, "您好，这是 PInvoke！ ", ".NET", 0 );
    }
}
```

面向对象的编程语言几乎都用到了抽象类这一概念，抽象类为实现抽象事物提供了更大的灵活性。C#也不例外, C#通过覆盖虚接口的技术深化了抽象类的应用。

2.2.7、覆盖虚接口

有时候我们需要表达一种抽象的东西，它是一些东西的概括，但我们又不能真正的看到它成为一个实体在我们眼前出现，为此面向对象的编程语言便有了抽象类的概念。C#作为一个面向对象的语言，必然也会引入抽象类这一概念。接口和抽象类使您可以创建组件交互的定义。通过接口，可以指定组件必须实现的方法，但不实际指定如何实现方法。抽象类使您可以创建行为的定义，同时提供用于继承类的一些公共实现。对于在组件中实现多态行为，接口和抽象类都是很有用的工具。

一个抽象类必须为类的基本类列表中列出的接口的所有成员提供实现程序。但是，一

个抽象类被允许把接口方法映射到抽象方法中。例如

```
interface IMethods {
    void F();
    void G();
}
abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

这里， IMethods 的实现函数把 F 和 G 映射到抽象方法中，它们必须在从 C 派生的非抽象类中被覆盖。

注意显式接口成员实现函数不能是抽象的，但是显式接口成员实现函数当然可以调用抽象方法。例如

```
interface IMethods
{
    void F();
    void G();
}
abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

这里，从 C 派生的非抽象类要覆盖 FF 和 GG， 因此提供了 IMethods 的实际实现程序。

3、抽象类与接口区别

3.1、版权声明

文章出处：http://www.cnblogs.com/xgyb12458/archive/2008/09/11/1289057.html
文章作者：xgyb12458

3.2、内容详情

抽象类与接口异同点

【相同点】

- 都不能被直接实例化，都可以通过继承实现其抽象方法。
- 都是面向抽象编程的技术基础，实现了诸多的设计模式。

【不同点】

- 接口支持多继承；抽象类不能实现多继承。
- 接口只能定义抽象规则；抽象类既可以定义规则，还可能提供已实现的成员。
- 接口是一组行为规范；抽象类是一个不完全的类，着重族的概念。
- 接口可以用于支持回调；抽象类不能实现回调，因为继承不支持。
- 接口只包含方法、属性、索引器、事件的签名，但不能定义字段和包含实现的方法；抽象类可以定义字段、属性、包含有实现的方法。
- 接口可以作用于值类型和引用类型；抽象类只能作用于引用类型。例如，Struct 就可以继承接口，而不能继承类。

通过相同与不同的比较，我们只能说接口和抽象类，各有所长，但无优略。在实际的编程实践中，

- 我们要视具体情况来酌情量才，但是以下的经验和积累，或许能给大家一些启示，除了我的一些积累之外，
- 很多都来源于经典，我相信经得起考验。所以在规则与场合中，我们学习这些经典，最重要的是学以致用，
- 当然我将以一家之言博大家之笑，看官请继续。

规则与场合

- 请记住，面向对象思想的一个最重要的原则就是：面向接口编程。
- 借助接口和抽象类，23 个设计模式中的很多思想被巧妙的实现了，我认为其精髓简单说来就是：面向抽象编程。
- 抽象类应主要用于关系密切的对象，而接口最适合为不相关的类提供通用功能。
- 接口着重于 CAN-DO 关系类型，而抽象类则偏重于 IS-A 式的关系；
- 接口多定义对象的行为；抽象类多定义对象的属性；

- 接口定义可以使用 public、protected、internal 和 private 修饰符，但是几乎所有的接口都定义为 public，原因就不必多说了。
- “接口不变”，是应该考虑的重要因素。所以，在由接口增加扩展时，应该增加新的接口，而不能更改现有接口。
- 尽量将接口设计成功能单一的功能块，以.NET Framework 为例，IDisposable、IComparable、IEquatable、IEnumerable 等都只包含一个公共方法。
- 接口名称前面的大写字母“I”是一个约定，正如字段名以下划线开头一样，请坚持这些原则。
- 在接口中，所有的方法都默认为 public。
- 如果预计会出现版本问题，可以创建“抽象类”。例如，创建了狗（Dog）、鸡（Chicken）和鸭（Duck），那么应该考虑抽象出动物（Animal）来应对以后可能出现风马牛的事情。而向接口中添加新成员则会强制要求修改所有派生类，并重新编译，所以版本式的问题最好以抽象类来实现。
- 从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实实现。
- 对抽象类不能使用 new 关键字，也不能被密封，原因是抽象类不能被实例化。
- 在抽象方法声明中不能使用 static 或 virtual 修饰符。

4、把委托说透

4.1、版权声明

文章出处：http://www.cnblogs.com/kirinboy/tag/%E5%A7%94%E6%89%98/
文章作者：姚琪琳

4.2、内容详情

4.2.1、开始委托之旅 委托与接口

委托，本是一个非常基础的.NET 概念，但前一阵子在园子里却引起轩然大波。先是Michael Tao 的随笔让人们将委托的写法与茴香豆联系到了一起，接着老赵又用一系列文章分析委托写法的演变，并告诫“嘲笑孔乙己的朋友们，你们在一味鄙视“茴”的四种写法的同时，说不定也失去了一个了解中国传统文化的机会呢!”。

在我个人看来，委托是.NET Framework 中一个非常炫的特性，绝不会向有些评论里说的那样，根本没有机会接触。恰恰相反，我们几乎每天都会接触委托，使用委托。

本系列试图从个人对于委托的理解展开，对委托的内涵和外延均加以讨论。文中有何不妥或不正确的地方，欢迎大家拍砖斧正。

好了，下面让我从一个示例开始，一步一步引入委托的概念。

从示例开始
假设一个系统的用户登录模块有如下所示的代码

```
class User
{
    public string Name { get; set; }

    public string Password { get; set; }
}

class UserService
{
    public void Register(User user)
    {
        if (user.Name == "Kirin")
```

```
            {
                Log("注册失败，已经包含名为" + user.Name + "的用户");
            }
            else
            {
                Log("注册成功！");
            }
        }
        privte void Log(string message)
        {
            Console.WriteLine(message);
        }
}
```

UserService 类封装用户登录的逻辑，并根据不同的登录情况向控制台打印不同的日志内容。当程序关闭时，所记录的日志自然也随之消失。

客户端的代码为

```
class Program
{
    static void Main(string[] args)
    {
        User user = new User { Name = "Kirin", Password = "123" };
        UserService service = new UserService();
        service.Register(user);
        Console.ReadLine();
    }
}
```

使用策略模式

然而这样的设计肯定是无法满足用户的需求的，用户肯定希望能够查看以前的日志记录，而不仅仅是程序打开以后的内容。如果我们仅仅修改 Log 方法的实现，那么用户需求再次改变时我们该如何处理呢？难道要无休止地修改 Log 方法吗？

既然日志记录的方式是变化的根源，我们自然会想到将其进行封装。我们创建一个名为 ILog 的接口。

```
interface ILog
{
    void Log(string message);
}
```

并创建两个实现了 ILog 的类，ConsoleLog 和 TextLog，分别用来向控制台和文本文件输出日志内容。

```
class ConsoleLog : ILog
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

class TextLog : ILog
{
    public void Log(string message)
    {
        using (StreamWriter sw = File.AppendText("log.txt"))
        {
            sw.WriteLine(message);
            sw.Flush();
            sw.Close();
        }
    }
}
```

在 UserService 类中添加一个 ILog 类型的属性 LogStrategy。

```
class UserService
{
    public ILog LogStrategy { get; set; }

    public UserService()
    {
        LogStrategy = new ConsoleLog();
    }

    public void Register(User user)
    {
        if (user.Name == "Kirin")
        {
            LogStrategy.Log("注册失败，已经包含名为" + user.Name + "的用户");
        }
        else
        {
```

```
                LogStrategy.Log("注册成功！");
            }
        }
}
```

客户端代码变为如下形式。

```
class Program
{
    static void Main(string[] args)
    {
        User user = new User { Name = "Kirin", Password = "123" };
        UserService service = new UserService { LogStrategy = new TextLog() };
        service.Register(user);
        Console.ReadLine();
    }
}
```

在声明 UserService 的时候，还可以将将 LogStrategy 设置为 TextLog。这样在 UserService 进行逻辑处理时，使用的 LogStrategy 即为 TextLog，日志将输出到文本文件中。

我们在干什么？我们在重构。重构的结果是什么？重构的结果是实现了一个简单的策略模式。

使用委托

然而策略模式仍然不能满足客户的需求，这是为什么呢？

1. 用户也许会希望自定义 Log 的实现。当然，你可以通过在客户代码处扩展 ILog 来实现自己的日志记录方式。如

```
class TextBoxLog : ILog
{
    private TextBox textBox;

    public TextBoxLog(TextBox textBox)
    {
        this.textBox = textBox;            this.textBox.Multiline = true;
    }

    public void Log(string message)
    {
        textBox.AppendText(message);
```

```
            textBox.AppendText(Environment.NewLine);
        }
}
```

但这种方案是否过于复杂呢？如果用户希望在 ListView 或其他控件上显示，是否需要逐个创建新类呢？并且这样的实现是否与客户端的耦合过于紧密呢？比如用户希望在 ListView 的各个列中显示日志内容、时间、来源等不同内容，那么在 ListViewLog 中对 ListView 硬编码是否很难重用呢？

2. 用户也许会希望同时使用多种日志记录方式。比如，同时向控制台、文本文件、客户端控件和事件查看器中输出日志。你当然可以在 UserService 中维护一个 List<ILog>，但这时 UserService 的职责过多，显然违反了 SRP。

下面介绍本文的主角：委托。

我们首先来创建一个名为 Log 的委托，它接收一个 string 类型的参数。

```
public delegate void Log(string message);
```

然后在 UserService 类中添加一个 Log 委托类型的属性 LogDelegate。

```
class UserService
{
    public Log LogDelegate { get; set; }       // …
}
```

在客户端，我们直接声明两个静态方法，它们都包含一个 string 类型的参数，并且没有返回值。

```
static void LogToConsole(string message)
{
    Console.WriteLine(message);
}

static void LogToTextFile(string message)
{
    using (StreamWriter sw = File.AppendText("log.txt"))
    {
        sw.WriteLine(message);
        sw.Flush();
        sw.Close();
    }
}
```

客户端声明 UserService 的代码变为

```
static void Main(string[] args)
{
    User user = new User { Name = "Kirin", Password = "123" };
    UserService service = new UserService();
    service.LogDelegate = LogToConsole;
    service.LogDelegate += LogToTextFile;
    service.Register(user);

    Console.ReadLine();
}
```

在构造委托时，我们还可以使用匿名方法和 Lambda 表达式，在老赵的文章中详细阐述了这些写法的演变。

对于何时使用委托，何时使用接口（即策略模式），MSDN 中有明确的描述：

在以下情况下，请使用委托：

- 当使用事件设计模式时。
- 当封装静态方法可取时。
- 当调用方不需要访问实现该方法的对象中的其他属性、方法或接口时。
- 需要方便的组合。
- 当类可能需要该方法的多个实现时。

在以下情况下，请使用接口：

- 当存在一组可能被调用的相关方法时。
- 当类只需要方法的单个实现时。
- 当使用接口的类想要将该接口强制转换为其他接口或类类型时。
- 当正在实现的方法链接到类的类型或标识时：例如比较方法。

您可能觉得上面的例子阐述委托和接口有些过于牵强，事实上有些时候的确很难选择使用接口还是委托。Java 中没有委托，但所有委托适用的情况同样可以使用包含单一方法的接口来实现的。在某种程度上，可以说委托是接口（仅定义了单一方法）的一种轻量级实现，它更灵活，也更方便。

到此为止，我们一步一步用委托重构了最初的代码。再接下来的随笔中，我们将开始更深一步的讨论。

4.2.2、深入理解委托

在上一篇随笔中我们通过示例逐步引入了委托，并比较了委托和接口。本文将重点剖析委托的实质。

委托在本质上仍然是一个类，我们用 delegate 关键字声明的所有委托都继承自 System.MulticastDelegate。后者又是继承自 System.Delegate 类，System.Delegate 类则继承自 System.Object。委托既然是一个类，那么它就可以被定义在任何地方，即可以定义在类的内部，也可以定义在类的外部。

正如很多资料上所说的，委托是一种类型安全的函数回调机制， 它不仅能够调用实例方法，也能调用静态方法，并且具备按顺序执行多个方法的能力。

委托揭秘

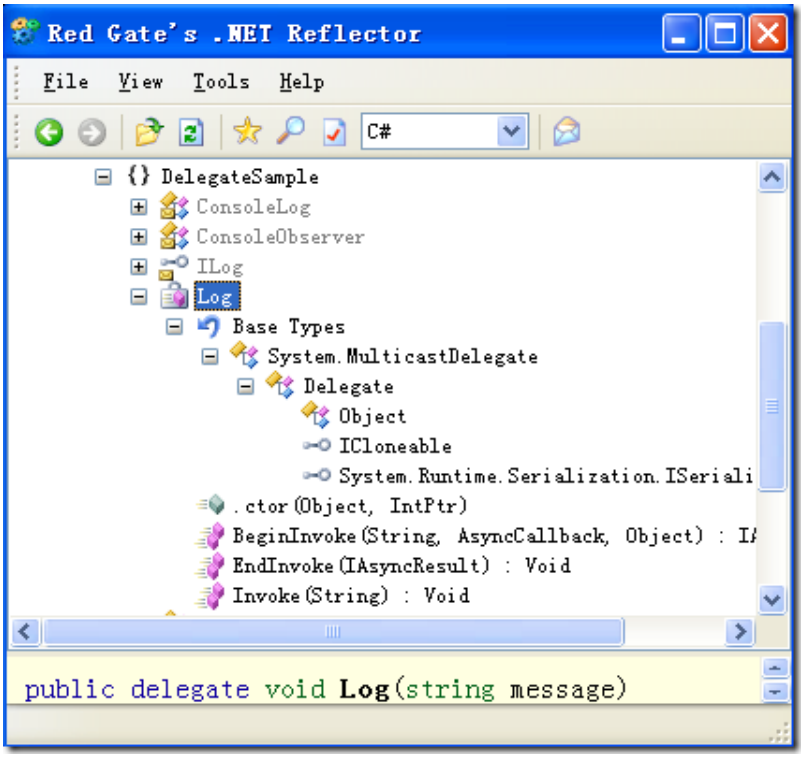
在把委托说透（1）中可以看到，委托的使用其实是很简单的。尽管如此，其内部实现仍然相当复杂。.NET 强大的编译器和 CLR 掩盖了这种复杂性。

为了解释方便，我们把（1）中的委托代码复制在下面，并做一处小小的改动，将 LogToTextFile 设置为实例方法。

```
namespace DelegateSample
{
    public delegate void Log(string message);
    class UserService
    {
        public Log LogDelegate { get; set; }
        public UserService() { }
        public void Register(User user)
        {
            if (user.Name == "Kirin")
            {
                LogDelegate("注册失败，已经包含名为" + user.Name + "的用户");
            }
            else
            {
                LogDelegate("注册成功！");
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
```

```
        {
            User user = new User { Name = "Kirin", Password = "123" };
            UserService service = new UserService();
            service.LogDelegate = LogToConsole;                Program p = new Program();
            service.LogDelegate += p.LogToTextFile;
            service.Register(user);
            Console.ReadLine();
        }
        static void LogToConsole(string message)
        {
            Console.WriteLine(message);
        }
        void LogToTextFile(string message)
        {
            using (StreamWriter sw = File.AppendText("log.txt"))
            {
                sw.WriteLine(message);
                sw.Flush();
                sw.Close();
            }
        }
    }
}
```

打开 Reflector 反编译 Log 委托，可以看到 Log 类被编译为如下形式：



在上图中可以得出如下结论：

委托是一个类

可以很清晰的看出 Log—>MulticastDelegate—>Delegate 这种继承机制。

尽管委托继承自 System.MulticastDelegate 类，但我们并不能显示地声明一个继承自 System.MulticastDelegate 类的委托。委托必须使用 delegate 关键字声明，编译器会自动为我们生成继承代码。

由于委托继承自 System.MulticastDelegate 类，自然也继承 MulticastDelegate 类的字段、属性和方法。这些成员中，最重要的当属三个非公共字段，如下表所示：

字段名称	字段类型	描述
_target	System.Object	该字段指明委托所调用的方法所在的实例类型。如果委托调用的为静态方法，该字段为null；如果为实例方法则为该方法所在的对象。
_methodPtr	System.IntPtr	标识回调方法的指针。
_invocationList	System.Object	在构建委托链时指向一个委托数组，在委托刚刚构建时通常为null。

由上表可以看出，每个委托对象实际上是对方法及其调用时操作的对象的封装。MulticastDelegate 类还定义了两个只读公有实例属性：Target 和 Method，分别对应_target 和 _methodPtr。Target 属性返回一个方法回调时操作的对象引用。如果是静态方法则返回 null。Method 属性返回一个标识回调方法的 System.Reflection.MethodInfo 对象。

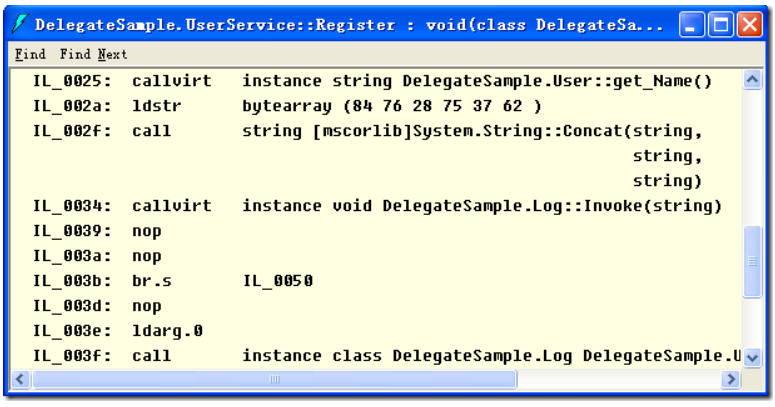
编译器自动为委托创建了 BeginInvoke、EndInvoke 和 Invoke 三个方法
当我们在像调用普通的方法一样调用委托时，如

```
LogDelegate("注册失败，已经包含名为" + user.Name + "的用户");
```

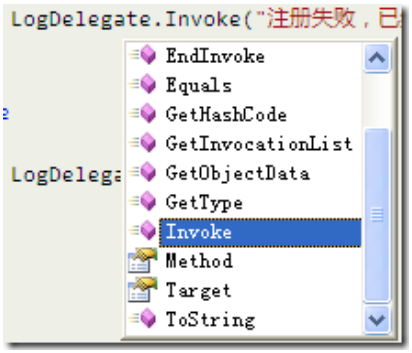
这时实际上调用的是编译器自动生成的 Invoke 方法

```
LogDelegate.Invoke("注册失败，已经包含名为" + user.Name + "的用户");
```

使用 IL DASM 查看 UserService 的 IL 代码，可以验证以上结论，如下图所示：



在使用委托时，我们也可以显示调用 Invoke 方法（CLR 2.0）。



Invoke 方法的参数和返回值与委托是一致的。在调用 Invoke 方法时，会使用_target 和 _methodPtr 字段。

BeginInvoke 和 EndInvoke 方法用来实现异步调用，本文在此不进行讨论。

委托链

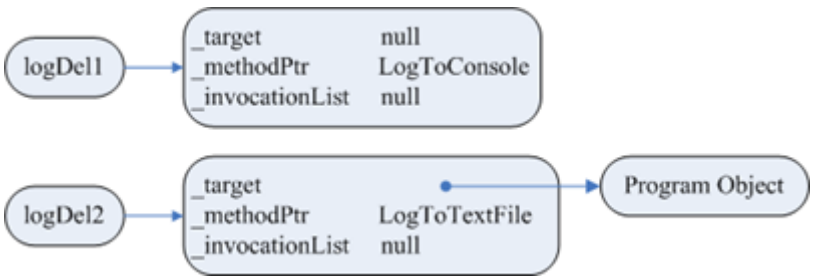
委托链是一个委托的集合，它允许我们调用这个集合中的委托所代表的所有方法（对于有返回值的方法，委托链的返回值为链表中最后一个方法的返回值，本文后面会有详细介绍）。在 Delegate 类中定义了 3 个静态方法来帮助我们操作委托链。

```
public static Delegate Combine(params Delegate[] delegates);
public static Delegate Combine(Delegate a, Delegate b);
public static Delegate Remove(Delegate source, Delegate value);
```

要理解委托链，我们首先基于前面的例子，重新声明两个委托：logDel1 和 logDel2。

```
Log logDel1 = LogToConsole;
Program p = new Program();
Log logDel2 = p.LogToTextFile;
```

这两个委托的_target、_methodPtr 和_invocationList 值分别如下图所示：

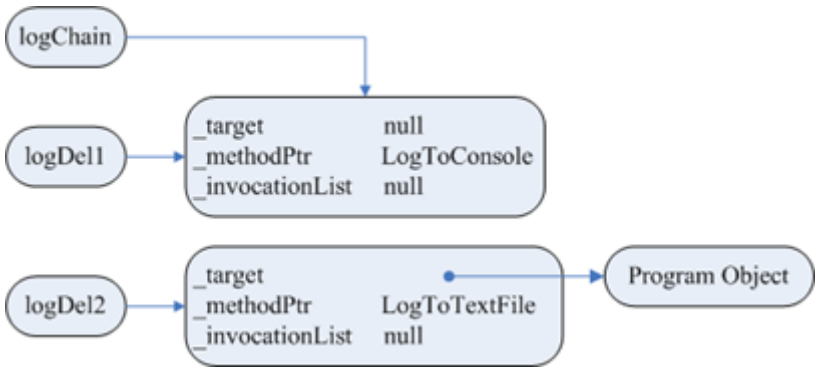


构造委托链

然后，我们使用 Combin 方法来构造一个委托链：

```
Log logChain = null;
logChain = (Log)Delegate.Combine(logChain, logDel1);
```

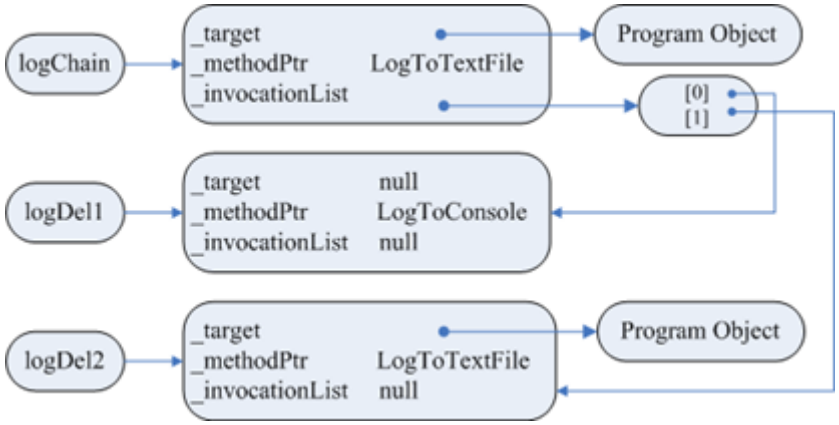
由于 logChain 初始为 null，在使用 Combin 方法构造委托链时，将返回另外一个参数 logDel1，再将 logDel1 的引用赋给 logChain。这时 logChain 将指向 logDel1 所指向的对象。



接下来我们将 logDel2 也添加到 logChain 中来：

```
logChain = (Log)Delegate.Combine(logChain, logDel2);
```

此时，由于 logChain 已经不再是 null，将重新构建一个新的委托对象。该委托对象的 _target 和_methodPtr 字段与 logDel2（第二个参数）相同，_invocationList 字段将指向一个委托数组。该委托数组中包含两个元素，第一个元素（索引为 0）指向封装了 LogToConsole 方法的委托（即 logDel1 指向的委托）；第二个元素（索引为 1）指向封装了 LogToTextFile 方法的委托（即 logDel2 指向的委托）。最后，将这个新创建的委托对象的引用赋给 logChain。



若再将一个新的委托 logDel3 添加到委托链中，则仍然会构建一个新的委托对象，并将 logDel3 的引用添加到该委托对象_invocationList 的末尾（此时链表共有 3 个元素）。然后，再将该委托对象的引用赋给 logChain。而 logChain 之前指向的委托对象则等待垃圾回收。

至此，委托链构造完毕，我们来看看如何执行委托链表中的委托。由于 logChain 仍然指向一个委托对象，因此执行委托链表的语法与执行委托是一样的：

```
logChain("执行委托链");
```

与普通的委托（如 logDel1）所不同的是，logChain 的_invocationList 字段不为 null。这时将首先遍历执行_invocationList 中的所有委托。所执行的方法的顺序与添加的顺序一致，依次为 LogToConsole、LogToTextFile。

委托 Log 的 Invoke 方法的实现用伪代码表示如下：

```
public void Invoke(string message)
{
    Delegate[] delegateSet = _InvocationList as Delegate[];
    if (delegateSet != null)
    {
        // 如果委托数组不为空，则依次执行该委托数组中的委托
        foreach (Feedback d in delegateSet)
            d(value);
    }
    else
    {
        // 如果委托数组为空，则该委托不代表一个委托链
        // 按照正常方式执行该委托
        _methodPtr.Invoke(_target, value);
    }
}
```

包含返回值的委托的 Invoke 实现如下，假设返回值为 string：

```
public string Invoke(string message)
{
    string result = null;
    Delegate[] delegateSet = _InvocationList as Delegate[];
    if (delegateSet != null)
    {
        // 如果委托数组不为空，则依次执行该委托数组中的委托
        foreach (Feedback d in delegateSet)
            result = d(value);
    }
    else
    {
        // 如果委托数组为空，则该委托不代表一个委托链
        // 按照正常方式执行该委托
        result = _methodPtr.Invoke(_target, value);
    }
    return result;
}
```

可以看到在委托链中，返回值为链表中最后一个委托的返回值。

那么如果对两个委托链调用 Combine 方法呢？

```
Log logChain = null;
Log logChain1 = null;
Log logChain2 = null;
logChain1 = (Log)Delegate.Combine(logChain1, logDel1);
logChain1 = (Log)Delegate.Combine(logChain1, logDel2);
logChain2 = (Log)Delegate.Combine(logChain2, logDel3);
logChain2 = (Log)Delegate.Combine(logChain2, logDel4);
logChain = (Log)Delegate.Combine(logChain1, logChain2);
```

最终的结果是，logChain 的_target 和_methodPtr 均与 logDel4 相同（确切地说，两个委托对象的_methodPtr 字段并不相同，但 Method 属性是相同的），而_invocationList 中委托的顺序依次为 logDel1、logDel2、logDel3、logDel4。

综上所述，可以对 Delegate.Combine(Delegate A, Delegate B)方法做如下总结：

1）如果 A 和 B 均为 null，则返回 null。
2）如果 A 或 B 一个为 null 而另一个不为 null，则返回不为 null 的委托。
3）如果 A 和 B 均不为 null，返回一个新的委托，该委托
（1）_target 字段与 B 的_target 字段的值相同
（2）Method 属性与 B 的 Method 属性的值相同
（3）_invocationList 字段为一个委托数组，该数组中委托的顺序为：A 中_invacationList 所指向的委托数组 + B 中_invacationList 所指向的委托数组。

移除委托链

Combine 方法用来向委托链中添加一个委托，而 Remove 方法用来从委托链中移除一个委托。

```
logChain = (Log)Delegate.Remove(logChain, new Log(LogToConsole));
```

当调用 Remove 时，会遍历（倒序）第一个参数（logChain）中的中的委托列表（_invocationList 字段），找到与第二个参数（new Log(LogToConsole)）的_target 和_methodPtr 字段相匹配的委托，并将其从委托列表中移除。返回值需分以下几种情况，为了描述方便，我们将 logChain 记为 A，将 new Log(LogToConsole)记为 B。

1） 如果 A 为 null，返回 null。
2）如果 B 为 null，返回 A。
3）如果 A 的_invocationList 为 null，即不包含委托链，那么如果 A 本身与 B 匹配，则返回 null，否则返回 A。
4） 如果 A 的_invocationList 中不包含与 B 匹配的委托，则返回 A。

5）如果 A 的_invocationList 中包含与 B 匹配的委托，则从链表中移除 B，然后

（1）如果 A 的链表中只剩下一个委托，则返回该委托。

（2）如果 A 的链表中还剩下多个委托，将重新构建一个新的委托 R（R 的_invocationList 字段为 A 的_invocationList 移除了 B 之后的链表），并返回 R。

注意，Remove 方法只移除源委托的_invocationList 列表中第一个匹配的委托，要想移除所有匹配的委托，可以使用 RemoveAll 方法。

有了委托链，在（1）中提出的第二个疑问就迎刃而解了。当用户希望使用多种日志记录方式的时候，使用委托链可以轻松地添加和删除某种日志记录方式，从而避免了人为地维护一个列表。

总结

本文首先介绍了委托的实质，委托是一个类，它继承自 System.MulticastDelegate，而 MulticastDelegate 又继承自 System.Delegate。然后重点剖析了委托链，讨论了如何创建和移除委托链。

4.2.3、委托与事件

在把委托说透（1）和（2）中，先后介绍了委托的语法和本质，本文重点介绍.NET 中与委托息息相关的概念——事件。在此之前，首先需要补充（2）中遗漏的一部分内容，即 C#在语法上对委托链的支持。

C#编译器为委托类型提供了+=和-=两个操作符的重载，分别对应 Delegate.Combine 和 Delegate.Remove 方法，使用这两个操作符可以大大简化委托链的构造和移除。

好了，有了+=和-=，我们就可以开始今天的话题了。

什么是事件？

事件（event）是类型中的一种成员，定义了事件成员的类型允许类型（或者类型的实例）在某些特定事情发生的时候通知其他对象。如 Button 类型的 Click 事件，在按钮被点击的时候，程序中的其他对象可以得到一个通知，并执行相应的动作。事件就是支持这种交互的类型成员。

CLR 中的事件模型是建立在委托这一机制之上的，这种关联存在其必然性。

我们知道，委托是对方法的抽象，它将方法的调用与实现相分离。方法的调用者（即委托的执行者）并不知道方法的内部是如何实现的，而方法的实现者也不知道该方法会在何时被调用。

事件也是如此。事件被触发后会执行什么样的操作，是由触发者决定的，如点击一个按钮之后是插入一条记录还是用户登录。事件的拥有者只知道什么情况下会触发事件，但并不

知道事件的具体实现。因此用委托来实现事件的机制就是自然而然的事情了。

事件与委托的关系到底是什么样呢？委托是与类、接口同一级别的概念，而事件属于类型的成员，与方法、属性、字段等是同一级别的概念。一个与事件相关联的委托的定义如下：

```
public delegate void FooEventHandler(object sender, FooEventArgs e);
```

而相应事件成员的定义为：

```
public event FooEventHandler Foo;
```

可见，事件用 event 关键字定义，其类型为一个委托类型，即事件是通过委托来实现的。

一个完整的事件定义和使用的例子如下：

```
public delegate void FooEventHandler(object sender, FooEventArgs e);

public class FooEventArgs : EventArgs { }

public class Bar
{
    public event FooEventHandler Foo;

    protected virtual void OnFoo(FooEventArgs e)
    {
        FooEventHandler handler = Foo;
        if (handler != null)
            handler(this, e);
    }

    public void SomeMethod()
    {
        // ...
        OnFoo(new FooEventArgs());
        // ...
    }
}

public class Client
{
    public Client()
    {
        Bar b = new Bar();
        b.Foo += new FooEventHandler(b_Foo);
```

```
    }

    void b_Foo(object sender, FooEventArgs e)
    {
        throw new NotImplementedException();
    }
}
```

我们注意到在 SomeMethod 方法中并没有直接调用委托，而是调用了一个辅助方法 OnFoo。在该方法中，先将 Foo 事件的引用传递给新定义的委托，然后再进行空判断，在委托不为 null 的情况下才进行调用。这样做是为了保证线程和类型的安全，我们在下面将会介绍。

还有一个需要注意的地方是，客户端为事件注册方法时，使用的是+=操作符。在本文开头已经介绍，+=对应 Delegate.Combine 方法，回顾（2）中阐述的委托链的构造，我们可以得出如下结论：在为事件注册方法时，实际上是在构造一个委托链。

事件的设计规范

《Framework Design Guidelines 2nd Edition》一书应该成为我们设计.NET 程序的规范手册。书中对于事件的定义采取了如下的规定：

事件的命名

由于通常事件以为着某种行为，因此事件的名称应该为一个动词，并用动词的时态来指明事件发生的时间。《Framework Design Guidelines 2nd Edition》对事件命名的建议如下：
1）用动词或动词短语来为事件命名。如 Clicked、Painting、DroppedDown 等等。
2）用现在时和将来时来表示“之前”和“之后”的概念，不要用 Before 和 Arfter 前缀。例如在窗体关闭之前触发的事件可以命名为 Closing，而窗体关闭之后触发的事件则应该命名为 Closed。
3）为事件处理程序（委托）的名称添加 EventHandler 后缀。如
4）使用 sender 和 e 来命名时间的两个参数。如上例。
5）为事件的数据参数类型的名称添加 EventArgs 后缀。如上例。

事件的设计

1）通常情况下，事件所对应的委托的返回值为 void，并且包含两个参数：第一个参数为触发事件的对象，通常为事件的拥有者（即上例中的 Bar 对象）。第二个参数为事件相关的数据，由事件的拥有者传递给事件的调用者。
2）在.NET 2.0 及以后的版本中自定义事件时，使用 System.EventHandler<TEventArgs>委托，而不要自定义新的委托类型。因此上例中如果在.NET 2.0 下应该定义为：

```
public event EventHandler<FooEventArgs> Foo;
```

在.NET 2.0 以前，由于不支持泛型，我们仍然需要像上面例子中那样定义。
3）为事件自定义一个 EventArgs 的子类，作为传递数据的参数。如果不需要传递任何参数，可以直接使用 EventArgs 类。
4）为每个事件编写一个受保护的虚方法作为触发方法，如上例中的 OnFoo 方法。这仅适用于 unsealed 类的非静态事件，并不适用于 struct、sealed class 和静态事件。这样做的原因是，通过 override 为子类提供一种处理事件的方式。按照惯例，该虚方法以 On 开头，以事件名称结尾，如 OnFoo 方法。

为了确保委托在调用时不抛出 NullReferenceException，在 OnXxx 方法中通常都会对委托进行判空操作，如

```
if (Xxx != null) Xxx(this, e);
```

然而仅仅这样是不够的，因为事件处理程序的添加和移除并不是线程安全的，因此在多线程环境下，Xxx 委托在判空之后很可能被 Remove，导致 Xxx 在调用时可能为 null。由于 Remove 方法将会构造一个新的委托实例，而不会改变原委托的引用，因此需要先将委托的引用传递给一个新的委托，再对这个新委托进行判空和调用等操作，这样即使原委托被 Remove，也不会 NullReferenceException。

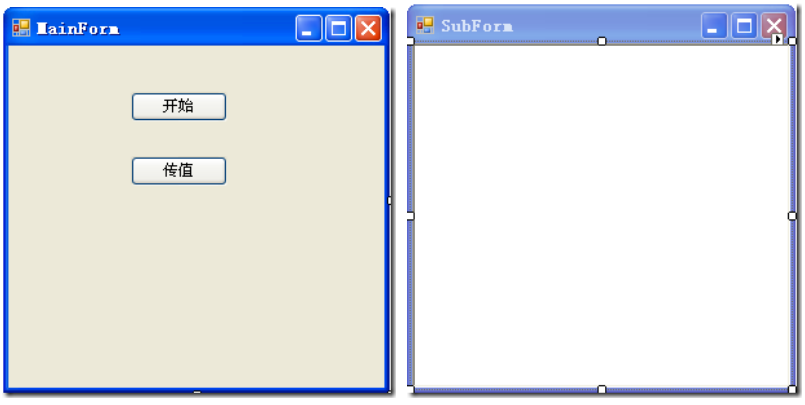
```
FooEventHandler handler = Foo;
if (handler != null) handler(this, e);
```

5）触发事件的方法有且仅有一个参数，XxxEventArgs 参数。
6）在触发非静态事件时，sender 参数不要为 null。对于静态事件，sender 参数要为 null。
7）触发事件时，如果不需要传递任何数据，数据参数可以为 EventArgs.Empty，不要为 null。

事件的应用举例

在前面随笔的评论中，有同学提出希望列举委托在窗体间传值的例子。好吧，我们就举一个简单的 WinForm 窗体传值的例子。

我们首先新建一个 Windows From 应用程序，并新建两个窗体 MainForm 和 SubForm，在 MainForm 中建立两个 Button，在 SubForm 中添加一个 RichTextBox。如下图所示：



当点击“开始”的时候，会弹出 SubForm，点击“传值”的时候，会将当前时间显示在 SubForm 的 RichTextBox 中。

需求大体就是这样了，我们该如何设计呢？

点击“传值”按钮后，会引起 SubForm 的变化。SubForm 只负责显示，它并不知道引起变化的原因。MainForm 负责引起变化，并将变化传递给 SubForm，但它并不关心 SubForm 如何进行处理。这与我们之前对事件的描述十分相似：

事件被触发后会执行什么样的操作，是由触发者决定的，如点击一个按钮之后是插入一条记录还是用户登录。事件的拥有者只知道什么情况下会触发事件，但并不知道事件的具体实现。

因此，在这个示例中，我们可以通过事件来实现传值。我们首先创建数据参数类 SendEventArgs，它包含一个 Message 属性，用来保存数据。

```
public class SendEventArgs : EventArgs
{
    public string Message { get; private set; }
    public SendEventArgs(string message)
    {
        this.Message = message;
    }
}
```

然后在 MainForm 中添加一个事件：Send。

```
public event EventHandler<SendEventArgs> Send;
```

然后我们为该事件编写触发方法 OnSend：

```
protected virtual void OnSend(SendEventArgs e)
{
    EventHandler<SendEventArgs> handler = Send;
    if (handler != null)
        handler(this, e);
}
```

MainForm 中两个按钮的事件处理程序如下：

```
private void btnBegin_Click(object sender, EventArgs e)
{
    SubForm subForm = new SubForm(this);
```

```
    subForm.Show();
}

private void btnSend_Click(object sender, EventArgs e)
{
    SendEventArgs sendEventArgs = new SendEventArgs(DateTime.Now.ToString());
    OnSend(sendEventArgs);
}
```

btnBegin 按钮用来打开一个 SubForm，并将当前 MainForm 实例作为参数传入。btnSend 按钮用来构造 Send 事件的数据参数，并调用 Send 事件的触发方法。

在 SubForm 中，有一个 MainForm 类型的私有字段，用于保存构造函数里传入的参数。

private MainForm parent;构造函数中除了给 parent 字段赋值外，还要注册 parent 的 Send 事件的处理程序：

```
public SubForm(MainForm main)
{
    InitializeComponent();
    this.parent = main;
    parent.Send += new EventHandler<SendEventArgs>(parent_Send);
}
```

parent_Send 处理程序负责向 RichTextBox 中添加信息：

```
private void parent_Send(object sender, SendEventArgs e)
{
    this.rtbTime.AppendText(e.Message);
    this.rtbTime.AppendText(Environment.NewLine);
}
```

最后我们在 SubForm 的 Closing 事件里移除 parent_Send，这样就可以打开多个 SubForm 了。

```
private void SubForm_FormClosing(object sender, FormClosingEventArgs e)
{
    parent.Send -= new EventHandler<SendEventArgs>(parent_Send);
}
```

整个 Demo 的显示如下：

总结

本文重点讲解了.NET 中的事件，并对事件的设计进行了规范，最终通过一个示例加深了我们对事件的理解。

您是否从以上示例中感觉到了观察者模式的影子呢？本系列接下来的一篇随笔中，我们将会讨论委托与设计模式的微妙联系。

4.2.4、委托与设计模式

委托与很多设计模式都有着千丝万缕的联系，在前面的随笔中已经介绍了委托与策略模式的联系，本节主要来讨论委托与其他两个模式：观察者模式和模板方法模式。

委托与观察者模式

在.NET 中，很多设计模式得到了广泛应用，如 foreach 关键字实现了迭代器模式。同样的，.NET 中也内置了观察者模式的实现方式，这种方式就是委托。

观察者模式的一般实现

网上可以找到很多资料介绍观察者模式的实现，我这里介绍一种简单的退化后的观察者模式，即 Subject 类为具体类，在其之上不再进行抽象。

```
public class Subject
{
    private List<Observer> observers = new List<Observer>();

    private string state;
    public string State
    {
        set
```

```
            {
                state = value;
                NotifyObservers();
            }
            get { return state; }
        }

        public void RegisterObserver(Observer ob)
        {
            observers.Add(ob);
        }

        public void RemoveObserver(Observer ob)
        {
            observers.Remove(ob);
        }

        public void NotifyObservers()
        {
            foreach (Observer ob in observers)
                ob.Update(this);
        }
}

public abstract class Observer
{
        public abstract void Update(Subject subject);
}

public class ConsoleObserver : Observer
{
        public ConsoleObserver(Subject subject)
        {
            subject.RegisterObserver(this);
        }

        public override void Update(Subject subject)
        {
            Console.WriteLine("Subject has changed its state : " + subject.State);
        }
}
```

调用的方法很简单：

```
Subject subject = new Subject();
Observer observer = new ConsoleObserver(subject);
subject.State = "Kirin Yao";
```

Subject 类维护一个列表，负责观察者的注册和移除。当其状态发生改变时，就调用 NotifyObservers 方法通知各个观察者。

观察者模式的委托实现

在.NET 中，使用委托可以更简单更优雅地实现观察者模式。在上一篇随笔中，最后的示例其实就是一个观察者模式。MainForm 为 Subject，SubForm 为 Observer。当 MainForm 的状态发生改变时（即点击“传值”按钮时），SubForm 作为观察者响应来自 MainForm 的变化。

与上例对应的，用委托实现的观察者模式的代码大致如下：

```
namespace DelegateSample
{
    class UpdateEventArgs : EventArgs { }

    class Subject
    {
        private string state;

        public string State
        {
            get { return state; }
            set
            {
                state = value;
                OnUpdate(new UpdateEventArgs());
            }
        }

        public event EventHandler<UpdateEventArgs> Update;

        private void OnUpdate(UpdateEventArgs e)
        {
            EventHandler<UpdateEventArgs> handler = Update;
            if (handler != null)
                Update(this, e);
        }
    }
```

```
    abstract class Observer
    {
        public Subject Subject { get; set; }

        public Observer(Subject subject)
        {
            this.Subject = subject;
            this.Subject.Update += new EventHandler<UpdateEventArgs>(Subject_Update);
        }

        protected abstract void Subject_Update(object sender, UpdateEventArgs e);
    }

    class ConsoleObserver : Observer
    {
        public ConsoleObserver(Subject subject) : base(subject) { }

        protected override void Subject_Update(object sender, UpdateEventArgs e)
        {
            Subject subject = sender as Subject;
            if (subject != null)
                Console.WriteLine("Subject has changed its state : " + subject.State);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Subject subject = new Subject();
            Observer ob = new ConsoleObserver(subject);
            subject.State = "Kirin Yao";
            Console.ReadLine();
        }
    }
}
```

相比传统的观察者模式的实现方式（在 Subject 中维护一个 Observer 列表），使用委托避免了 Subject 与 Observer 之间的双向引用，Subject 作为主题类，对观察者毫无所知，降低了耦合性，语法上也更加优雅。

委托与模板方法模式

模板方法模式封装了一段通用的逻辑，将逻辑中的特定部分交给子类实现。

```
public abstract class AbstractClass
{
    public void Arithmetic()
    {
        SubArithmeticA();
        SubArithmeticB();
        SubArithmeticC();
    }

    protected abstract void SubArithmeticA();
    protected abstract void SubArithmeticB();
    protected abstract void SubArithmeticC();
}

public class ConcreteClass : AbstractClass
{
    protected override void SubArithmeticA()
    {
        //...
    }

    protected override void SubArithmeticB()
    {
        //...
    }

    protected override void SubArithmeticC()
    {
        //...
    }
}
```

然而这种继承方式的模板方法耦合度较高，特别是如果逻辑与其外部实现没有必然的从属关系的时候，用传统的模板方法就显得不那么合适了。

在某种程度上，委托可以看做是一个轻量级的模板方法实现方式，它将逻辑中的特定部分转交给注册到委托的方法来实现。从而替代了继承方式的模板方法模式中，在子类中实现特定逻辑的方式。

```
public delegate void SubArithmetic();

public class ConcreteClass
{
```

```
    public void Arithmetic()
    {
        if (SubArithmetic != null)
            SubArithmetic();
    }

    public SubArithmetic SubArithmetic { get; set; }
}
```

而 SubArithmetic 的实现交给外部：

```
ConcreteClass concrete = new ConcreteClass();
concrete.SubArithmetic = Program.SomeMethod;
concrete.Arithmetic();
```

咋一看在客户端中编写委托的方法似乎还略显麻烦，但值得注意的是，匿名方法和 Lambda 表达式为我们提供了更加简便的委托语法。在函数式编程日益盛行的今天，我们应该为.NET 提供的这种语言特性而感到庆幸。

总结

本文重点讨论委托与设计模式的关系，包括观察者模式和模板方法模式。您是否觉得委托与其他方法也有关系呢？不妨在回复中进行讨论。

到此为止，我们共发布了 4 篇随笔，讨论委托及其相关概念。

5、反射

5.1、版权声明

文章出处：http://www.cnblogs.com/JimmyZhang/category/107410.html
文章作者：Jimmy Zhang

5.2、内容详情

5.2.1、序章

引言

反射是.Net 提供给我们的一件强力武器，尽管大多数情况下我们不常用到反射，尽管我们可能也不需要精通它，但对反射的使用作以初步了解在日后的开发中或许会有所帮助。

反射是一个庞大的话题，牵扯到的知识点也很多，包括程序集、自定义特性、泛型等，想要完全掌握它非常不易。本文仅仅对反射做一个概要介绍，关于它更精深的内容，需要在实践中逐渐掌握。本文将分为下面几个部分介绍.Net 中的反射：

- 序章，我将通过一个例子来引出反射，获得对反射的第一印象。
- 反射初步、Type 类、反射普通类型。(修改中，近期发布...)
- 反射特性(Attribute)。
- xxxx (待定)
- ...

序章

如果你还没有接触过反射，而我现在就下一堆定义告诉你什么是反射，相信你一定会有当头一棒的感觉。我一直认为那些公理式的定义和概念只有在你充分懂得的时候才能较好的发挥作用。所以，我们先来看一个开发中常遇到的问题，再看看如何利用反射来解决：

在进行数据库设计的过程中，常常会建立一些基础信息表，比如说：全国的城市，又或者订单的状态。假设我们将城市的表，起名为 City，它通常包含类似这样的字段：

```
Id        Int Identity(1,1)  城市 Id
Name      Varchar(50)              城市名称
ZIP       Varchar(10)              城市邮编
... // 略
```

这个表将供许多其他表引用。假如我们在建立一个酒店预订系统，那么酒店信息表(Hotel)就会引用此表，用 CityId 字段来引用酒店所在城市。对于城市(City)表这种情况，表里存放的记录(城市信息)是不定的，意思就是说：我们可能随时会向这张表里添加新的城市(当某个城市的第一家酒店想要加入预订系统时，就需要在 City 表里新添这家酒店所在的城市)。此时，这样的设计是合理的。

1）建表及其问题

我们再看看另外一种情况，我们需要标识酒店预订的状态：未提交、已提交、已取消、受理中、已退回、已订妥、已过期。此时，很多开发人员会在数据库中建立一张小表，叫做 BookingStatus(预订状态)，然后将如上状态加入进去。

如同城市(City)表一样，在系统的其他表，比如说酒店订单表(HotelOrder)中，通过字段 StatusId 引用这个表来获取酒店预订状态。然而，几个月以后，虽然看上去和城市表的用法一样，结果却发现这个表只在数据库做联合查询或者 只在程序中调用，却从来不做修改，因为预订流程确定下来后一般是不会变更的。在应用程序中，也不会给用户提供对这个表记录的增删改操作界面。

而在程序中调用这个表时，经常是这种情况：我们需要根据预订状态对订单列表进行筛选。此时通常的做法是使用一个下拉菜单(DropDownList)，菜单的数据源(DataSource)，我们可以很轻易地通过一个 SqlDataReader 获得，我们将 DropDownList 的文本 Text 设为 Status 字段，将值 Value 设为 Id 字段。

此时，我们应该已经发现问题：

如果我们还有航班预订、游船预订，或者其他一些状态，我们需要在数据库中创建很多类似的小表，造成数据库表的数目过多。
我们使用 DropDownList 等控件获取表内容时，需要连接到数据库进行查询，潜在地影响性能。

同时，我们也注意到三点：

此表一般会在数据库联合查询中使用到。假设我们有代表酒店订单的 HotelOrder 表，它包含代表状态的 StatusId 字段，我们的查询可能会像这样：

```
Select *, (Select Status From BookingStatus Where Id = HotelOrder.StatusId) as Status From HotelOrder。
```

在应用程序中，此表经常作为 DropDownList 或者其他 List 控件的数据源。
这个表几乎从不改动。

2）数组及其问题

意识到这样设计存在问题，我们现在就想办法解决它。我们所想到的第一个办法是可以在程序中创建一个数组来表示预订状态，这样我们就可以删掉 BookingStatus 状态表(注意可以这样做是因为 BookingStatus 表的内容确定后几乎从不改动)。

```
string[] BookingStatus = {
    "NoUse", "未提交","已提交","已取消","受理中","已退回","已订妥","已过期"
};        // 注意数组的 0 号元素仅仅是起一个占位作用，以使程序简洁。因为 StatusId 从 1 开始。
```

我们先看它解决了什么：上面提到的问题 1、问题 2 都解决了，既不需要在数据库中创建表，又无需连接到数据库进行查询。

我们再看看当我们想要用文本显示酒店的预订时，该怎么做(假设有订单类 HotelOrder，其属性 StatusId 代表订单状态，为 int 类型)。

```
// GetItem 用于获取一个酒店订单对象, orderId 为 int 类型，代表订单的 Id
HotelOrder myOrder = GetItem(orderId);
lbStatus.Text = BookingStatus[myOrder.StatusId];   //lbStatus 是一个 Label 控件
```

目前为止看上去还不错，现在我们需要进行一个操作，将订单的状态改为“受理中”。

```
myOrder.StatusId = 4;
```

很不幸，我们发现了使用数组可能带来的第一个问题：不方便使用，当我们需要更新订单的状态值时，我们需要去查看 BookingStatus 数组的定义(除非你记住所有状态的数字值)，然后根据状态值在数组中的位置来给对象的属性赋值。

我们再看另一个操作，如果某个订单的状态为“已过期”,就要对它进行删除：

```
if(BookingStatus[myOrder.StatusId]=="已过期"){
    DeleteItem(myOrder);        // 删除订单
}
```

此时的问题和上面的类似：我们需要手动输入字符串“已过期”，此时 Vs2005 的智能提示发挥不了任何作用，如果我们不幸将状态值记错，或者手误打错，就将导致程序错误，较为稳妥的做法还是按下 F12 导向到 BookingStatus 数组的定义，然后将“已过期”复制过来。

现在，我们再看看如何来绑定到一个 DropDownList 下拉列表控件(Id 为 ddlStatus)上。

```
ddlStatus.DataSource = BookingStatus;
ddlStatus.DataBind();
```

但是我们发现产生的 HTML 代码是这样：

```
<select name="ddlStatus" id="ddlStatus">
    <option value="未提交">未提交</option>
    <option value="已提交">已提交</option>
    <option value="已取消">已取消</option>
    <option value="受理中">受理中</option>
    <option value="已退回">已退回</option>
    <option value="已订妥">已订妥</option>
    <option value="已过期">已过期</option>
</select>
```

我们看到，列表项的 value 值与 text 值相同，这显然不是我们想要的，怎么办呢？我们可以给下拉列表写一个数据绑定的事件处理方法。

```
protected void Page_Load(object sender, EventArgs e) {
    ddlStatus.DataSource = BookingStatus;
    ddlStatus.DataBound += new EventHandler(ddlStatus_DataBound);
    ddlStatus.DataBind();
}

void ddlStatus_DataBound(object sender, EventArgs e) {
    int i = 0;
    ListControl list = (ListControl)sender; //注意，将 sender 转换成 ListControl
    foreach (ListItem item in list.Items) {
        i++;
        item.Value = i.ToString();
    }
}
```

这样，我们使用数组完成了我们期望的效果，虽然这样实现显得有点麻烦，虽然还存在上面提到的不便于使用的问题，但这些问题我们耐心细心一点就能克服，而软件开发几乎从来就没有 100%完美的解决方案，那我们干脆就这样好了。

NOTE：在 ddlStatus_DataBound 事件中，引发事件的对象 sender 显然是 DropDownList，但是这里却没有将 sender 转换成 DropDownList，而是将它转换成基类型 ListControl。这样做是为了更好地进行代码重用，ddlStatus_DataBound 事件处理方法将不仅限于 DropDownList，对于继承自 ListControl 的其他控件，比如 RadioButtonList、ListBox 也可以不加改动地使用 ddlStatus_DataBound 方法。

如果你对事件绑定还不熟悉，请参考 C#中的委托和事件 一文。

这里也可以使用 Dictionary<String, Int>来完成，但都存在类似的问题，就不再举例了。

3）枚举及其问题

然而不幸的事又发生了... 我们的预订程序分为两部分：一部分为 B/S 端，在 B/S 端可以进行酒店订单的 创建(未提交)、提交(已提交)、取消提交(已取消)，另外还可以看到是不是已订妥；一部分为 C/S 端，为酒店的预订中心，它可以进行其他状态的操作。

此时，对于整个系统来说，应该有全部的 7 个状态。但对于 B/S 端来说，它只有 未提交、已提交、已取消、已订妥 四个状态，对应的值分别为 1、2、3、6。

我们回想一下上面是如何使用数组来解决的，它存在一个缺陷：我们默认地将订单状态值与数组的索引一一对应地联系了起来。

所以在绑定 DropDownList 时，我们采用自增的方式来设定列表项的 Value 值；或者在显示状态时，我们通过 lbStatus.Text = BookingStatus[myOrder.StatusId]; 这样的语句来完成。而当这种对应关系被打破时，使用数组的方法就失效了，因为如果不利用数组索引，我们没有额外的地方去存储状态的数字值。

此时，我们想到了使用枚举：

```
public enum BookingStatus {
    未提交 = 1,
    已提交,
    已取消,
    已订妥 = 6
}
```

我们想在页面输出一个订单的状态时，可以这样：

```
HotelOrder myOrder = GetItem(orderId);              //获取一个订单对象
lbStatus.Text = ((BookingStatus)myOrder.StatusId).ToString(); // 输出文本值
```

我们想更新订单的状态为 “已提交”：

```
myOrder.StatusId = (int)BookingStatus.已提交;
```

当状态为“已取消”时我们想执行某个操作：

```
if(BookingStatus.已取消 == (BookingStatus)myOrder.StatusId){
    // Do some action
}
```

此时，VS 2005 的智能提示已经可以发挥完全作用，当我们在 BookingStatus 后按下“.”时，可以显示出所有的状态值。

NOTE：当我们使用枚举存储状态时，myOrder 对象的 StatusId 最好为 BookingStatus 枚举类型，而非 int 类型，这样操作会更加便捷一些，但为了和前面使用数组时的情况保持统一，这里 StatusId 仍使用 int 类型。

以上三种情况使用枚举都显得非常的流畅，直到我们需要绑定枚举到 DropDownList 下拉列表的时候：我们知道，可以绑定到下拉列表的有两类对象，一类是实现了 IEnumerable 接口的可枚举集合，比如 ArrayList，String[]，List<T>；一类是实现了 IListSource 的数据源，比如 DataTable，DataSet。

NOTE：实际上 IListSource 接口的 GetList()方法返回一个 IList 接口，IList 接口又继承了 IEnumerable 接口。由此看来，IEnumerable 是实现可枚举集合的基础，在我翻译的一篇文章 C#中的枚举器 中，对这个主题做了详细的讨论。

可我们都知道：枚举 enum 是一个基本类型，它不会实现任何的接口，那么我们下来该如何做呢？

4）使用反射遍历枚举字段

最笨也是最简单的办法，我们可以先创建一个 GetDataTable 方法，此方法依据枚举的字段值和数字值构建一个 DataTable，最后返回这个构建好的 DataTable：

```
private static DataTable GetDataTable() {
    DataTable table = new DataTable();
    table.Columns.Add("Name", Type.GetType("System.String"));          //创建列
    table.Columns.Add("Value", Type.GetType("System.Int32"));          //创建列

    DataRow row = table.NewRow();
    row[0] = BookingStatus.未提交.ToString();
    row[1] = 1;
    table.Rows.Add(row);

    row = table.NewRow();
    row[0] = BookingStatus.已提交.ToString();
    row[1] = 2;
    table.Rows.Add(row);

    row = table.NewRow();
    row[0] = BookingStatus.已取消.ToString();
    row[1] = 3;
    table.Rows.Add(row);

    row = table.NewRow();
    row[0] = BookingStatus.已订妥.ToString();
    row[1] = 6;
```

```
        table.Rows.Add(row);

        return table;
    }
```

接下来，为了方便使用，我们再创建一个专门采用这个 DataTable 来设置列表控件的方法 SetListCountrol()：

```
// 设置列表
  public static void SetListControl(ListControl list) {
        list.DataSource = GetDataTable();          // 获取 DataTable
        list.DataTextField = "Name";
        list.DataValueField = "Value";
        list.DataBind();
  }
```

现在，我们就可以在页面中这样去将枚举绑定到列表控件：

```
protected void Page_Load(object sender, EventArgs e)
{
    SetListControl(ddlStatus);     // 假设页面中已有 ID 为 ddlStatus  的 DropDownList
}
```

如果所有的枚举都要通过这样去绑定到列表，我觉得还不如在数据库中直接建表，这样实在是太麻烦了，而且我们是根据枚举的文本和值去 HardCoding 出一个 DataTable 的：

```
DataRow row = table.NewRow();
row[0] = BookingStatus.未提交.ToString();
row[1] = 1;
table.Rows.Add(row);

row = table.NewRow();
row[0] = BookingStatus.已提交.ToString();
row[1] = 2;
table.Rows.Add(row);

row = table.NewRow();
row[0] = BookingStatus.已取消.ToString();
row[1] = 3;
table.Rows.Add(row);

row = table.NewRow();
row[0] = BookingStatus.已订妥.ToString();
```

```
row[1] = 6;
table.Rows.Add(row);
```

这个时候，我们想有没有办法通过遍历来实现这里？如果想要遍历这里，首先，我们需要一个包含枚举的每个字段信息的对象，这个对象至少包含两条信息，一个是字段的文本(比如“未提交”)，一个是字段的数字型值（比如 1），我们暂且管这个对象叫做 field。其次，应该存在一个可遍历的、包含了字段信息的对象(也就是 filed) 的集合，我们暂且管这个集合叫做 enumFields。

那么，上面就可以这样去实现：

```
foreach (xxxx field in enumFields)
{
    DataRow row = table.NewRow();
    row[0] = field.Name;              // 杜撰的属性，代表 文本值(比如“未提交”)
    row[1] = filed.intValue;        // 杜撰的属性，代表 数字值(比如 1)

    table.Rows.Add(row);
}
```

这段代码很不完整，我们注意到 xxxx，它应该是封装了字段信息(或者叫元数据 metadata)的对象的类型。而对于 enumFields，它的类型应该是 xxxx 这个类型的集合。这段代码是我们按照思路假想和推导出来的。实际上，.Net 中提供了 Type 类 和 System.Reflection 命名空间来帮助解决我们现在的问题。

我在后面将较详细地介绍 Type 类，现在只希望你能对反射有个第一印象，所以只简略地作以说明：Type 抽象类提供了访问类型元数据的能力，当你实例化了一个 Type 对象后，你可以通过它的属性和方法，获取类型的元数据信息，或者进一步获得该类型的成员的元数据。注意到这里，因为 Type 对象总是基于某一类型的，并且它是一个抽象类，所以我们在创建 Type 类型时，必须要提供 类型，或者类型的实例，或者类型的字符串值(Part.2 会说明)。

创建 Type 对象有很多种方法，本例中，我们使用 typeof 操作符来进行，并传递 BookingStatus 枚举：

```
Type enumType = typeof(BookingStatus);
```

然后，我们应该想办法获取 封装了字段信息的对象 的集合。Type 类提供 GetFields() 方法来实现这一过程，它返回一个 FieldInfo[] 数组。实际上，也就是上面我们 enumFields 集合的类型。

```
FieldInfo[] enumFields = enumType.GetFields();
```

现在，我们就可以遍历这一集合：

```
foreach (FieldInfo field in enumFields)
{
    if (!field.IsSpecialName)
    {
        DataRow row = table.NewRow();
        row[0] = field.Name;        // 获取字段文本值
        row[1] = Convert.ToInt32(myField.GetRawConstantValue()); // 获取 int 数值
        table.Rows.Add(row);
    }
}
```

这里 field 的 Name 属性获取了枚举的文本，GetRawConstantValue()方法获取了它的 int 类型的值。

我们看一看完整的代码：

```
private static DataTable GetDataTable() {

    Type enumType = typeof(BookingStatus);      // 创建类型
    FieldInfo[] enumFields = enumType.GetFields();      //获取字段信息对象集合

    DataTable table = new DataTable();
    table.Columns.Add("Name", Type.GetType("System.String"));
    table.Columns.Add("Value", Type.GetType("System.Int32"));
   // 遍历集合
    foreach (FieldInfo field in enumFields) {
        if (!field.IsSpecialName) {
            DataRow row = table.NewRow();
            row[0] = field.Name;
            row[1] = Convert.ToInt32(field.GetRawConstantValue());
            //row[1] = (int)Enum.Parse(enumType, field.Name); //也可以这样

            table.Rows.Add(row);
        }
    }

    return table;
 }
```

注意，SetListControl()方法依然存在并有效，只是为了节省篇幅，我没有复制过来，它的使用和之前是一样的，我们只是修改了 GetDataTable()方法。

5）使用泛型来达到代码重用

观察上面的代码，如果我们现在有另一个枚举，叫做 TicketStatus，那么我们要将它绑定到列表，我们唯一需要改动的就是这里：

```
Type enumType = typeof(BookingStatus); //将 BookingStatus 改作 TicketStatus
```

既然这样，我们何不定义一个泛型类来进行代码重用呢？我们管这个泛型类叫做 EnumManager<TEnum>。

```
public static class EnumManager<TEnum>
{
    private static DataTable GetDataTable()
    {
        Type enumType = typeof(TEnum);    // 获取类型对象
        FieldInfo[] enumFields = enumType.GetFields();

        DataTable table = new DataTable();
        table.Columns.Add("Name", Type.GetType("System.String"));
        table.Columns.Add("Value", Type.GetType("System.Int32"));
        //遍历集合
        foreach (FieldInfo field in enumFields)
        {
            if (!field.IsSpecialName)
            {
                DataRow row = table.NewRow();
                row[0] = field.Name;
                row[1] = Convert.ToInt32(field.GetRawConstantValue());
                //row[1] = (int)Enum.Parse(enumType, field.Name);  也可以这样

                table.Rows.Add(row);
            }
        }
        return table;
    }

    public static void SetListControl(ListControl list)
    {
        list.DataSource = GetDataTable();
        list.DataTextField = "Name";
        list.DataValueField = "Value";
        list.DataBind();
    }
}
```

OK，现在一切都变得简便的多，以后，我们再需要将枚举绑定到列表，只要这样就行了(ddl 开头的是 DropDownList,rbl 开头的是 RadioButtonList)：

```
EnumManager<BookingStauts>.SetListControl(ddlBookingStatus);
EnumManager<TicketStatus>.SetListControl(rblTicketStatus);
```

NOTE：如果你对泛型不熟悉，请参阅 C# 中的泛型 一文。上面的实现并没有考虑到性能的问题，仅仅为了引出反射使用的一个实例。

6）Net 中反射的一个范例。

不管是 VS2005 的智能提示，还是修改变量名时的重构功能，都使用了反射功能。在.Net FCL 中，也经常能看到反射的影子，这里就向大家演示一个最常见的例子。大家知道，在 CLR 中一共有两种类型，一种是值类型，一种是引用类型。声明一个引用类型的变量并对类型实例化，会在应用程序堆(Application Heap)上分配内存，创建对象实例，然后将对象实例的内存地址返回给变量，变量保存的是内存地址，实际相当于一个指针；声明一个值类型的实例变量，则会将它分配在线程堆栈(Thread Stack)上，变量本身包含了值类型的所有字段。

现在假设我们需要比较两个对象是否相等。当我们比较两个引用类型的变量是否相等时，我们比较的是这两个变量所指向的是不是堆上的同一个实例(内存地址是否相同)。而当我们比较两个值类型变量是否相等时，怎么做呢？因为变量本身就包含了值类型所有的字段(数据)，所以在比较时，就需要对两个变量的字段进行逐个的一对一的比较，看看每个字段的值是否都相等，如果任何一个字段的值不等，就返回 false。

实际上，执行这样的一个比较并不需要我们自己编写代码，Microsoft 已经为我们提供了实现的方法：所有的值类型继承自 System.ValueType，ValueType 和所有的类型都继承自 System.Object ，Object 提供了一个 Equals()方法，用来判断两个对象是否相等。但是 ValueType 覆盖了 Object 的 Equals()方法。当我们比较两个值类型变量是否相等时，可以调用继承自 ValueType 类型的 Equals()方法。

```
public struct ValPoint {
    public int x;
    public int y;
}
static void Main(string[] args) {
    bool result;

    ValPoint A1;
    A1.x = A1.y = 3;

    ValPoint B1 = A1;                  // 复制 A 的值给 B
    result = A1.Equals(B1);
    Console.WriteLine(result);         // 输出 True;
}
```

你有没有想到当调用 Equals()方法时会发生什么事呢？前面我们已经提到如果是值类型，会对两个变量的字段进行逐个的比较，看看每个字段的值是否都相等，但是如何获取变量的所有字段，遍历字段，并逐一比较呢？此时，你应该意识到又到了用到反射的时候了，让我们使用 reflector 来查看 ValueType 类的 Equals()方法，看看微软是如何做的吧：

```
public override bool Equals(object obj) {
    if (obj == null) {
        return false;
    }
    RuntimeType type = (RuntimeType)base.GetType();
    RuntimeType type2 = (RuntimeType)obj.GetType();
    if (type2 != type) {
        return false;
    }
    object a = this;
    if (CanCompareBits(this)) {
        return FastEqualsCheck(a, obj);
    }
    // 获取所有实体字段
    FieldInfo[] fields = type.GetFields(BindingFlags.NonPublic | BindingFlags.Public |
BindingFlags.Instance);
    // 遍历字段，判断字段值是否相等
    for (int i = 0; i < fields.Length; i++) {
        object obj3 = ((RtFieldInfo)fields[i]).InternalGetValue(a, false);
        object obj4 = ((RtFieldInfo)fields[i]).InternalGetValue(obj, false);
        if (obj3 == null) {
            if (obj4 != null) {
                return false;
            }
        } else if (!obj3.Equals(obj4)) {
            return false;
        }
    }
    return true;
}
```

注意到上面加注释的那两段代码，可以看到当对值变量进行比较时，是会使用反射来实现。反射存在着性能不佳的问题(不仅如此，还存在着很多的装箱操作)，由此可见，在值类型上调用 Equals()方法开销是会很大的。但是这个例子仅仅为了说明反射的用途，我想已经达到了目的。上面的代码不能完全理解也不要紧，后面会再提到。

7）小结

看到这里，你应该对反射有了一个初步的概念(或者叫反射的一个用途)：反射是一种宽泛的叫法，它通过 System.Reflection 命名空间 并 配合 System.Type 类，提供了在运行时(Runtime)对于 类型和对象(及其成员)的基本信息 以及 元数据(metadata)的访问能力。

5.2.2、查看基本类型信息

反射概述 和 Type 类

1）反射的作用

简单来说，反射提供这样几个能力：1、查看和遍历类型(及其成员)的基本信息和程序集元数据(metadata)；2、迟绑定(Late-Binding)方法和属性。3、动态创建类型实例(并可以动态调用所创建的实例的方法、字段、属性)。序章中，我们所采用的那个例子，只是反射的一个用途：查看类型成员信息。接下来的几个章节，我们将依次介绍反射所提供的其他能力。

2）获取 Type 对象实例

反射的核心是 Type 类，这个类封装了关于对象的信息，也是进行反射的入口。当你获得了关于类型的 Type 对象后，就可以根据 Type 提供的属性和方法获取这个类型的一切信息(方法、字段、属性、事件、参数、构造函数等)。我们开始的第一步，就是获取关于类型的 Type 实例。获取 Type 对象有两种形式，一种是获取当前加载程序集中的类型(Runtime)，一种是获取没有加载的程序集的类型。

我们先考虑 Runtime 时的 Type，一般来说有三种获取方法：

2.1）使用 Type 类提供的静态方法 GetType()
比如我们想要获得 Stream 类型的 Type 实例，则可以这样：

```
Type t = Type.GetType("System.IO.Stream");
txtOutput.Text = t.ToString();
```

注意到 GetType 方法接受字符串形式的类型名称。

2.2）使用 typeof 操作符
也可以使用 C# 提供的 typeof 操作符来完成这一过程：

```
// 如果在页首写入了 using System.IO; 也可以直接用 typeof(Stream);
Type t = typeof(System.IO.Stream);
```

这时的使用有点像泛型，Stream 就好像一个类型参数一样，传递到 typeof 操作符中。

2.3）通过类型实例获得 Type 对象
我们还可以通过类型的实例来获得：

```
String name = "Jimmy Zhang";
Type t = name.GetType();
```

使用这种方法时应当注意，尽管我们是通过变量(实例)去获取 Type 对象，但是 Type 对象不包含关于这个特定对象的信息，仍是保存对象的类型(String)的信息。

3）Type 类型 及 Reflection 命名空间的组织结构

到现在为止，我已经多次提过 Type 封装了类型的信息，那么这些类型信息都包含什么内容呢？假设我们现在有一个类型的实例，它的名字叫做 demo，我们对它的信息一无所知，并通过下面代码获取了对于它的 Type 实例：

```
// 前面某处的代码实例化了 demo 对象
Type t = demo.GetType();
```

现在，我们期望 t 包含了关于 demo 的哪些信息呢？

3.1）demo 的类型的基本信息

- 我们当然首先想知道 demo 是什么类型的，也就是 demo 的类型名称。
- 我们还想知道该类型位于什么命名空间下。
- 它的基类型是什么，以及它在.Net 运行库中的映射类型。
- 它是值类型还是引用类型。
- 它是不是 Public 的。
- 它是枚举、是类、是数组、还是接口。
- 它是不是基础类型(int 等)。
- 等等 ...

Type 提供了下面的属性，用于获取类型的基本信息，常用的有下面一些：

属 性	说 明
Name	获取类型名称
FullName	类型全名
Namespace	命名空间名称
BaseType	获取对于基类的Type类型的引用
UnderlyingSystemType	在.Net中映射的类型的引用
Attributes	获取TypeAttributes位标记
IsValueType	是否值类型
IsByRef	是否由引用传递
IsEnum	是否枚举
IsClass	是否类
IsInterface	是否接口
IsSealed	是否密封类
IsPrimitive	是否基类型(比如int)
IsAbstract	是否抽象
IsPublic	是否公开
IsNotPublic	是否非公开
IsVisible	是否程序集可见
等等...	

3.2）demon 的类型的成员信息

- 我们可能还想知道它有哪些字段。
- 有些什么属性，以及关于这些属性的信息。
- 有哪些构造函数。
- 有哪些方法，方法有哪些参数，有什么样的返回值。
- 包含哪些事件。
- 实现了哪些接口。
- 我们还可以不加区分地获得它的所有 以上成员。

观察上面的列表，就拿第一条来说，我们想获取类型都有哪些字段，以及这些字段的信息。而字段都包含哪些信息呢？可能有字段的类型、字段的名称、字段是否 public、字段是否为 const、字段是否是 read only 等等，那么是不是应该将字段的这些信息也封装起来呢？

实际上，.Net 中提供了 FiledInfo 类型，它封装了关于字段的相关信息。对照上面的列表，类似的还有 PropertyInfo 类型、ConstructorInfo 类型、MethodInfo 类型、EventInfo 类型。而对于方法而言，对于它的参数，也会有 in 参数，out 参数，参数类型等信息，类似的，在 System.Reflection 命名空间下，除了有上面的提到的那么多 Info 后缀结尾的类型，还有个 ParameterInfo 类型，用于封装方法的参数信息。

最后，应该注意到 Type 类型，以及所有的 Info 类型均 继承自 MemberInfo 类型，MemberInfo 类型提供了获取类型基础信息的能力。

在 VS2005 中键入 Type，选中它，再按下 F12 跳转到 Type 类型的定义，纵览 Type 类型的成员，发现可以大致将属性和方法分成这样几组：

- IsXXXX，比如 IsAbstract，这组 bool 属性用于说明类型的某个信息。(前面的表格已经列举了一些。)
- GetXXXX()，比如 GetField()，返回 FieldInfo，这组方法用于获取某个成员的信息。
- GetXXXXs()，比如 GetFields()，返回 FieldInfo[]，这组方法用户获取某些成员信息。

还有其他的一些属性和方法，等后面遇到了再说。

由于 MemberInfo 是一个基类，当我们获得一个 MemberInfo 后，我们并不知道它是 PropertyInfo(封装了属性信息的对象)还是 FieldInfo(封装了属性信息的对象)，所以，有必要提供一个办法可以让我们加以判断，在 Reflection 命名空间中，会遇到很多的位标记，这里先介绍第一个位标记(本文管用[Flags]特性标记的枚举称为 位标记)，MemberTypes，它用于标记成员类型，可能的取值如下：

```
[Flags]
public enum MemberTypes {
    Constructor = 1,    //    该成员是一个构造函数
    Event = 2,          //    该成员是一个事件
    Field = 4,          //    该成员是一个字段
    Method = 8,         //    该成员是一个方法
    Property = 16,      //    该成员是一个属性
    TypeInfo = 32,      //    该成员是一种类型
    Custom = 64,        //    自定义成员类型
    NestedType = 128, //    该成员是一个嵌套类型
    All = 191,          //    指定所有成员类型。
}
```

反射程序集

在.Net 中，程序集是进行部署、版本控制的基本单位，它包含了相关的模块和类型，我并不打算详细地去说明程序集及其构成，只是讲述如何通过反射获取程序集信息。

在 System.Reflection 命名空间下有一个 Assembly 类型，它代表了一个程序集，并包含了关于程序集的信息。

在程序中加载程序集时，一般有这么几个方法，我们可以使用 Assembly 类型提供的静态方法 LoadFrom() 和 Load()，比如：

```
Assembly asm = Assembly.LoadFrom("Demo.dll");
```

或者

```
Assembly asm = Assembly.Load("Demo");
```

当使用 LoadFrom()方法的时候，提供的是程序集的文件名，当将一个程序集添加到项目引用中以后，可以直接写“文件名.dll”。如果想加载一个不属于当前项目的程序集，则需要给出全路径，比如：

```
Assembly asm = Assembly.LoadFrom(@"C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Web.dll");
```

使用 Load()方法的时候，只用提供程序集名称即可，不需要提供程序集的后缀名。如果想获得当前程序集，可以使用 Assembly 类型的静态方法 GetExecutingAssembly，它返回包含当前执行的代码的程序集(也就是当前程序集)。

```
Assembly as = Assembly.GetExecutingAssembly();
```

在获得一个 Type 类型实例以后，我们还可以使用该实例的 Assembly 属性来获得其所在的程序集：

```
Type t = typeof(int)
Assembly asm = t.Assembly;
```

一个程序集可能有多个模块(Module)组成，每个模块又可能包含很多的类型，但.Net 的默认编译模式一个程序集只会包含一个模块，我们现在看下 反射 提供了什么样的能力让我们获取关于程序集的信息(只列出了部分常用的)：

属 性/方 法	说 明
FullName	程序集名称
Location	程序集的路径
GetTypes()	获取程序集包含的全部类型
GetType()	获取某个类型
GetModules()	获取程序集包含的模块
GetModule()	获取某个模块
GetCustomAttributes()	获取自定义特性信息

NOTE：程序集和命名空间不存在必然联系，一个程序集可以包含多个命名空间，同一个命名空间也可以分放在几个程序集。

为了方便进行我们后面的测试，我们现在建立一个 Windows 控制台应用程序，我给它起名叫 SimpleExplore；然后再添加一个 Demo 类库项目，我们将来编写的代码就用户查看这个 Demo 项目集的类型信息 或者 是对这个程序集中的类型进行迟绑定。这个 Demon 项目只包含一个命名空间 Demo，为了体现尽可能多的类型同时又 Keep Simple，其代码如下：

```
namespace Demo {

    public abstract class BaseClass {

    }

    public struct DemoStruct { }

    public delegate void DemoDelegate(Object sender, EventArgs e);

    public enum DemoEnum {
        terrible, bad, common=4, good, wonderful=8
    }

    public interface IDemoInterface {
        void SayGreeting(string name);
    }

    public interface IDemoInterface2 {}

    public sealed class DemoClass:BaseClass, IDemoInterface,IDemoInterface2 {

        private string name;
        public string city;
        public readonly string title;
        public const string text = "Const Field";
        public event DemoDelegate myEvent;

        public string Name {
            private get { return name; }
            set { name = value; }
        }

        public DemoClass() {
            title = "Readonly Field";
        }

        public class NestedClass { }

        public void SayGreeting(string name) {
            Console.WriteLine("Morning :" + name);
        }
    }
```

```
}
```

现在我们在 SimpleExplore 项目中写一个方法 AssemblyExplor()，查看我们 Demo 项目生成的程序集 Demo.dll 定义的全部类型：

```
public static void AssemblyExplore() {
    StringBuilder sb = new StringBuilder();

    Assembly asm = Assembly.Load("Demo");

    sb.Append("FullName(全名)：" + asm.FullName + "\n");
    sb.Append("Location(路径)：" + asm.Location + "\n");

    Type[] types = asm.GetTypes();

    foreach (Type t in types) {
        sb.Append("    类型：" + t + "\n");
    }

    Console.WriteLine(sb.ToString());
}
```

然后，我们在 Main()方法中调用一下，应该可以看到这样的输出结果：

```
FullName(全名)：Demo, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Location(路径)：E:\MyApp\TypeExplorer\SimpleExplorer\bin\Debug\Demo.dll
模块: Demo.dll
    类型：Demo.BaseClass
    类型：Demo.DemoStruct
    类型：Demo.DemoDelegate
    类型：Demo.DemoEnum
    类型：Demo.IDemoInterface
    类型：Demo.IDemoInterface2
    类型：Demo.DemoClass
    类型：Demo.DemoClass+NestedClass
```

反射基本类型

这里说反射基本类型，基本类型是针对 泛型类型 来说的，因为 反射泛型 会更加复杂一些。在前面的范例中，我们获得了程序集中的所有类型，并循环打印了它们，打印结果仅仅显示出了类型的全名，而我们通常需要关于类型更详细的信息，本节我们就来看看如何进一步查看类型信息。

NOTE：因为一个程序集包含很多类型，一个类型包含很多成员(方法、属性等)，一个成员又包含很多其他的信息，所以如果我们从程序集层次开始写代码去获取每个层级的信息，那么会嵌套很多的 foreach 语句，为了阅读方便，我会去掉最外层的循环。

1）获取基本信息

有了前面 Type 一节的介绍，我想完成这里应该只是打打字而已，所以我直接写出代码，如有必要，会在注释中加以说明。我们再写一个方法 TypeExplore，用于获取类型的详细信息(记得 AssemblyExplore 只获取了类型的名称)：

```
public static void TypeExplore(Type t) {
    StringBuilder sb = new StringBuilder();

    sb.Append("名称信息：\n");
    sb.Append("Name: " + t.Name + "\n");
    sb.Append("FullName: " + t.FullName + "\n");
    sb.Append("Namespace: " + t.Namespace + "\n");

    sb.Append("\n 其他信息：\n");
    sb.Append("BaseType(基类型): " + t.BaseType + "\n");
    sb.Append("UnderlyingSystemType: " + t.UnderlyingSystemType + "\n");

    sb.Append("\n 类型信息：\n");
    sb.Append("Attributes(TypeAttributes 位标记): " + t.Attributes + "\n");
    sb.Append("IsValueType(值类型): " + t.IsValueType + "\n");
    sb.Append("IsEnum(枚举): " + t.IsEnum + "\n");
    sb.Append("IsClass(类): " + t.IsClass + "\n");
    sb.Append("IsArray(数组): " + t.IsArray + "\n");
    sb.Append("IsInterface(接口): " + t.IsInterface + "\n");
    sb.Append("IsPointer(指针): " + t.IsPointer + "\n");
    sb.Append("IsSealed(密封): " + t.IsSealed + "\n");
    sb.Append("IsPrimitive(基类型): " + t.IsPrimitive + "\n");
    sb.Append("IsAbstract(抽象): " + t.IsAbstract + "\n");
    sb.Append("IsPublic(公开): " + t.IsPublic + "\n");
    sb.Append("IsNotPublic(不公开): " + t.IsNotPublic + "\n");
    sb.Append("IsVisible: " + t.IsVisible + "\n");
    sb.Append("IsByRef(由引用传递): " + t.IsByRef + "\n");

    Console.WriteLine(sb.ToString());
}
```

然后，我们在 Main 方法中输入：

```
Type t = typeof(DemoClass);
TypeExplore(t);
```

会得到这样的输出：

```
名称信息：
Name: DemoClass
FullName: Demo.DemoClass
Namespace: Demo

其他信息：
BaseType(基类型): Demo.BaseClass
UnderlyingSystemType: Demo.DemoClass

类型信息：
Attributes(TypeAttributes 位标记): AutoLayout, AnsiClass, Class, Public, Sealed,
BeforeFieldInit
IsValueType(值类型): False
IsEnum(枚举): False
IsClass(类): True
IsArray(数组): False
IsInterface(接口): False
IsPointer(指针): False
IsSealed(密封): True
IsPrimitive(基类型): False
IsAbstract(抽象): False
IsPublic(公开): True
IsNotPublic(不公开): False
IsVisible: True
IsByRef(由引用传递): False
```

值得注意的是 Attributes 属性，它返回一个 TypeAttributes 位标记，这个标记标识了类型的一些元信息，可以看到我们熟悉的 Class、Public、Sealed。相应的，IsClass、IsSealed、IsPublic 等属性也返回为 True。

2）成员信息 与 MemberInfo 类型

我们先考虑一下对于一个类型 Type，可能会包含什么类型，常见的有字段、属性、方法、构造函数、接口、嵌套类型等。MemberInfo 类代表着 Type 的成员类型，值得注意的是 Type 类本身又继承自 MemberInfo 类，理解起来并不困难，因为一个类型经常也是另一类型的成员。Type 类提供 GetMembers()、GetMember()、FindMember()等方法用于获取某个成员类型。

我们再添加一个方法 MemberExplore()，来查看一个类型的所有成员类型。

```
public static void MemberExplore(Type t) {
    StringBuilder sb = new StringBuilder();

    MemberInfo[] memberInfo = t.GetMembers();

    sb.Append("查看类型 " + t.Name + "的成员信息：\n");

    foreach (MemberInfo mi in memberInfo) {
        sb.Append("成员：" + mi.ToString().PadRight(40) + "  类型: " + mi.MemberType + "\n");
    }

    Console.WriteLine(sb.ToString());
}
```

然后我们在 Main 方法中调用一下。

```
MemberExplore(typeof(DemoClass));
```

产生的输出如下：

```
查看类型 DemoClass 的成员信息：
-------------------------------------------------
成员：Void add_myEvent(Demo.DemoDelegate)          类型: Method
成员：Void remove_myEvent(Demo.DemoDelegate)       类型: Method
成员：System.String get_Name()                     类型: Method
成员：Void set_Name(System.String)                 类型: Method
成员：Void SayGreeting(System.String)              类型: Method
成员：System.Type GetType()                        类型: Method
成员：System.String ToString()                     类型: Method
成员：Boolean Equals(System.Object)                类型: Method
成员：Int32 GetHashCode()                          类型: Method
成员：Void .ctor()                                 类型: Constructor
成员：System.String Name                           类型: Property
成员：Demo.DemoDelegate myEvent                    类型: Event
成员：System.String text                           类型: Field
成员：Demo.DemoClass+NestedClass                   类型: NestedType
```

我们使用了 GetMembers()方法获取了成员信息的一个数组，然后遍历了数组，打印了成员的名称和类型。如同我们所知道的：Name 属性在编译后成为了 get_Name()和 set_Name()两个独立的方法；myEvent 事件的注册(+=)和取消注册(-=)分别成为了 add_myEvent()和 remove_myEvent 方法。同时，我们发现私有(private)字段 name 没有被打印出来，另外，基类 System.Object 的成员 GetType()和 Equals()也被打印了出来。

有的时候，我们可能不希望查看基类的成员，也可能希望查看私有的成员，此时可以使用 GetMembers()的重载方法，传入 BindingFlags 位标记参数来完成。BindingFlags 位标记对如何获取成员的方式进行控制(也可以控制如何创建对象实例，后面会说明)。对于本例，如果我们想获取所有的公有、私有、静态、实例 成员，那么只需要这样修改 GetMembers()方法就可以了。

```
MemberInfo[] memberInfo = t.GetMembers(
    BindingFlags.Public |
    BindingFlags.Static |
    BindingFlags.NonPublic |
    BindingFlags.Instance |
    BindingFlags.DeclaredOnly
);
```

此时的输出如下：

```
查看类型 DemoClass 的成员信息：
------------------------------------------------
成员：Void add_myEvent(Demo.DemoDelegate)          类型: Method
成员：Void remove_myEvent(Demo.DemoDelegate)       类型: Method
成员：System.String get_Name()                     类型: Method
成员：Void set_Name(System.String)                 类型: Method
成员：Void SayGreeting(System.String)              类型: Method
成员：Void .ctor()                                 类型: Constructor
成员：System.String Name                           类型: Property
成员：Demo.DemoDelegate myEvent                    类型: Event
成员：System.String name                           类型: Field
成员：Demo.DemoDelegate myEvent                    类型: Field
成员：System.String text                           类型: Field
成员：Demo.DemoClass+NestedClass                   类型: NestedType
```

可以看到，继承自基类 System.Object 的方法都被过滤掉了，同时，打印出了私有的 name, myEvent 等字段。

现在如果我们想要获取所有的方法(Method)，那么我们可以使用 Type 类的 FindMembers()方法：

```
MemberInfo[] memberInfo = t.FindMembers(
    MemberTypes.Method,         // 说明查找的成员类型为 Method
    BindingFlags.Public |
    BindingFlags.Static |
    BindingFlags.NonPublic |
```

```
        BindingFlags.Instance |
        BindingFlags.DeclaredOnly,
        Type.FilterName,
        "*"
);
```

Type.FilterName 返回一个 MemberFilter 类型的委托，它说明按照方法名称进行过滤，最后一个参数“*”，说明返回所有名称(如果使用“Get*”，则会返回所有以 Get 开头的方法)。现在的输出如下：

```
查看类型 DemoClass 的成员信息：
------------------------------------------------
成员：Void add_myEvent(Demo.DemoDelegate)          类型: Method
成员：Void remove_myEvent(Demo.DemoDelegate)     类型: Method
成员：System.String get_Name()                        类型: Method
成员：Void set_Name(System.String)                  类型: Method
成员：Void SayGreeting(System.String)               类型: Method
```

MemberInfo 类有两个属性值得注意，一个是 DeclaringType，一个是 ReflectedType，返回的都是 Type 类型。DeclaredType 返回的是声明该成员的类型。比如说，回顾我们之前的一段代码：

```
MemberInfo[] members = typeof(DemoClass).GetMembers();
```

它将返回所有的公有成员，包括继承自基类的 Equals()等方法，对于 Equals()方法来说，它的 DeclaringType 返回的是相当于 typeof(Object) 的类型实例，因为它是在 System.Object 中被定义的；而它的 ReflectedType 返回的则是相当于 typeof(DemoClass) 类型实例，因为它是通过 DemoClass 的类型实例被获取的。

3）字段信息 与 FieldInfo 类型

如同我们之前所说，MemberInfo 是一个基类，它包含的是类型的各种成员都公有的一组信息。实际上，对于字段、属性、方法、事件 等类型成员来说，它们包含的信息显然都是不一样的，所以，.Net 中提供了 FiledInfo 类型来封装字段的信息，它继承自 MemberInfo。

如果我们希望获取一个类型的所有字段，可以使用 GetFileds()方法。我们再次添加一个方法 FieldExplore()：

```
public static void FieldExplore(Type t) {
    StringBuilder sb = new StringBuilder();

    FieldInfo[] fields = t.GetFields();
```

```
        sb.Append("查看类型 " + t.Name + "的字段信息：\n");
        sb.Append(String.Empty.PadLeft(50, '-') + "\n");

        foreach (FieldInfo fi in fields) {
            sb.Append("名称：" + fi.Name + "\n");
            sb.Append("类型：" + fi.FieldType + "\n");
            sb.Append("属性：" + fi.Attributes + "\n\n");
        }

        Console.WriteLine(sb.ToString());
}
```

产生的输出如下：

```
查看类型 DemoClass 的字段信息：
-----------------------------------------------
名称：city
类型：System.String
属性：Public

名称：title
类型：System.String
属性：Public, InitOnly

名称：text
类型：System.String
属性：Public, Static, Literal, HasDefault
```

值得一提的是 fi.FieldType 属性，它返回一个 FieldAttributes 位标记，这个位标记包含了字段的属性信息。对比我们之前定义的 DemoClass 类，可以看到，对于 title 字段，它的属性是 public, InitOnly；对于 Const 类型的 text 字段，它的属性为 Public,Static,Literal，HasDefault，由此也可以看出，声明一个 const 类型的变量，它默认就是静态 static 的，同时，由于我们给了它初始值，所以位标记中也包括 HasDefault。

针对于 FieldType 位标记，FiledInfo 类提供了一组返回为 bool 类型的属性，来说明字段的信息，常用的有：IsPublic, IsStatic, IsInitOnly, IsLiteral, IsPrivate 等。

如果我们想要获取私有字段信息，依然可以使用重载了的 GetFields[]方法，传入 BindingFlags 参数，和上面的类似，这里就不重复了。

4）属性信息 与 PropertyInfo 类型

和字段类似，也可以通过 GetProperty()方法，获取类型的所有属性信息。

```
public static void PropertyExplore(Type t) {
    StringBuilder sb = new StringBuilder();
    sb.Append("查看类型 " + t.Name + "的属性信息：\n");
    sb.Append(String.Empty.PadLeft(50, '-') + "\n");

    PropertyInfo[] properties = t.GetProperties();

    foreach (PropertyInfo pi in properties) {
        sb.Append("名称：" + pi.Name + "\n");
        sb.Append("类型：" + pi.PropertyType + "\n");
        sb.Append("可读：" + pi.CanRead + "\n");
        sb.Append("可写：" + pi.CanWrite +"\n");
        sb.Append("属性：" + pi.Attributes +"\n");
    }

    Console.WriteLine(sb.ToString());
}
```

输出如下：

```
查看类型 DemoClass 的属性信息：
-------------------------------------------------
名称：Name
类型：System.String
可读：True
可写：True
属性：None
```

从前面的章节可以看到，Name 属性会在编译后生成 Get_Name()和 Set_Name()两个方法，那么，应该可以利用反射获取这两个方法。PropertyInfo 类的 GetGetMethod()和 GetSetMethod()可以完成这个工作，它返回一个 MethodInfo 对象，封装了关于方法的信息，我们会在后面看到。

5）方法信息 与 MethodInfo 类型

与前面的类似，我们依然可以编写代码来查看类型的方法信息。

```
public static void MethodExplore(Type t) {
    StringBuilder sb = new StringBuilder();
    sb.Append("查看类型 " + t.Name + "的方法信息：\n");
    sb.Append(String.Empty.PadLeft(50, '-') + "\n");

    MethodInfo[] methods = t.GetMethods();
```

```
    foreach (MethodInfo method in methods) {
        sb.Append("名称：" + method.Name +"\n");
        sb.Append("签名：" + method.ToString() + "\n");
        sb.Append("属性：" + method.Attributes + "\n");
        sb.Append("返回值类型：" + method.ReturnType + "\n\n");
    }

    Console.WriteLine(sb.ToString());
}
```

与前面类似，MethodInfo 类也有一个 Attributes 属性，它返回一个 MethodAttribute，MethodAttribute 位标记标明了方法的一些属性，常见的比如 Abstract, Static, Virtual,Public, Private 等。

与前面不同的是，Method 可以具有参数 和 返回值，MethodInfo 类提供了 GetParameters() 方法获取 参数对象的数组，方法的参数都封装在了 ParameterInfo 类型中。查看 ParameterInfo 类型的方法与前面类似，这里就不再阐述了。

4） ConstructorInfo 类型、EventInfo 类型

从名称就可以看出来，这两个类型封装了类型 的构造函数 和 事件信息，大家都是聪明人，查看这些类型与之前的方法类似，这里就不再重复了。

5）小结

本文涉及了反射的最基础的内容，我们可以利用反射来自顶向下地查看程序集、模块、类型、类型成员的信息。反射更强大、也更有意思的内容：迟绑定方法、动态创建类型以后会再讲到。

5.2.3、反射特性

反射特性(Attribute)

可能很多人还不了解特性，所以我们先了解一下什么是特性。想想看如果有一个消息系统，它存在这样一个方法，用来将一则短消息发送给某人：

```
// title: 标题；author：作者;content：内容；receiverId：接受者 Id
public bool SendMsg(string title, string author, string content, int receiverId){
    // Do Send Action
}
```

我们很快就发现这样将参数一个个罗列到方法的参数列表中扩展性很糟糕，我们最好定

义一个 Message 类将短消息封装起来，然后给方法传递一个 Message 对象：

```
public class Message{
    private string title;
    private string author;
    private string content;
    private int receiverId;
    // 略
}
public bool SendMsg(Messag msg){
    // Do some Action
}
```

此时，我们或许应该将旧的方法删除，用这个扩展性更好的 SendMsg 方法来取代。遗憾的是我们往往不能，因为这组程序可能作为一组 API 发布，在很多客户程序中已经在使用旧版本的 SendMsg()方法，如果我们在更新程序的时候简单地删除掉旧的 SendMsg()方法，那么将造成使用老版本 SendMsg()方法的客户程序不能工作。

这个时候，我们该如果做呢？我们当然可以通过方法重载来完成，这样就不用删除旧的 SendMsg()方法了。但是如果新的 SendMsg()不仅优化了参数的传递，并且在算法和效率上也进行了全面的优化，那么我们将会迫切希望告知客户程序现在有一个全新的高性能 SendMsg()方法可供使用，但此时客户程序并不知道已经存在一个新的 SendMsg 方法，我们又该如何做呢？我们可以打电话告诉维护客户程序的程序员，或者发电子邮件给他，但这样显然不够方便，最好有一种办法能让他一编译项目，只要存在对旧版本 SendMsg()方法的调用，就会被编译器告知。

1）.Net 内置特性介绍

.Net 中可以使用特性来完成这一工作。特性是一个对象，它可以加载到程序集及程序集的对象中，这些对象包括 程序集本身、模块、类、接口、结构、构造函数、方法、方法参数等，加载了特性的对象称作特性的目标。特性是为程序添加元数据(描述数据的数据)的一种机制，通过它可以给编译器提供指示或者提供对数据的说明。

NOTE：特性的英文名称叫做 Attribute，在有的书中，将它翻译为“属性”；另一些书中，将它翻译为“特性”；由于通常我们将含有 get 和/或 set 访问器的类成员称为“属性”(英文 Property),所以本文中我将使用“特性”这个名词，以区分“属性”(Property)。

中文版的 VS2005 使用“属性”。

1.1 System.ObsoleteAttribute 特性

我们通过这个例子来看一下特性是如何解决上面的问题：我们可以给旧的 SendMsg()方法上面加上 Obsolete 特性来告诉编译器这个方法已经过时，然后当编译器发现当程序中有地方在使用这个用 Obsolete 标记过的方法时，就会给出一个警告信息。

```
namespace Attribute {

    public class Message {}

    public class TestClass {
        // 添加 Obsolete 特性
        [Obsolete("请使用新的 SendMsg(Message msg)重载方法")]
        public static void ShowMsg() {
            Console.WriteLine("这是旧的 SendMsg()方法");
        }

        public static void ShowMsg(Message msg) {
            Console.WriteLine("新 SendMsg()方法");
        }

    }

    class Program {
        static void Main(string[] args) {
            TestClass.ShowMsg();
            TestClass.ShowMsg(new Message());
        }
    }
}
```

现在运行这段代码，我们会发现编译器给出了一个警告：警告 CS0618: “Attribute.TestClass.ShowMsg()”已过时:“请使用新的 SendMsg(Message msg)重载方法”。通过使用特性，我们可以看到编译器给出了警告信息，告诉客户程序存在一个新的方法可供使用，这样，程序员在看到这个警告信息后，便会考虑使用新的 SendMsg()方法。

NOTE：简单起见，TestClass 类和 Program 位于同一个程序集中，实际上它们可以离得很远。

1.2 特性的使用方法

通过上面的例子，我们已经大致看到特性的使用方法：首先是有一对方括号“[]”，在左方括号“[”后紧跟特性的名称，比如 Obsolete，随后是一个圆括号“()”。和普通的类不同，这个圆括号不光可以写入构造函数的参数，还可以给类的属性赋值，在 Obsolete 的例子中，仅传递了构造函数参数。

NOTE：实际上，当你用鼠标框选住 Obsolete，然后按下 F12 转到定义，会发现它的全名是 ObsoleteAttribute，继承自 Attribute 类。但是这里却仅用 Obsolete 来标记方法，这是.Net 的一个约定，所有的特性应该均以 Attribute 来结尾，在为对象标记特性时如果没有添加 Attribute，编译器会自动寻找带有 Attribute 的版本。

NOTE：使用构造函数参数，参数的顺序必须同构造函数声明时的顺序相同，所有在特性中也叫位置参数(Positional Parameters)，与此相应，属性参数也叫做命名参数(Named Parameters)。在下面会详细说明。

2）自定义特性(Custom Attributes)

2.1 范例介绍

如果不能自己定义一个特性并使用它，我想你怎么也不能很好的理解特性，我们现在就自己构建一个特性。假设我们有这样一个很常见的需求：我们在创建或者更新一个类文件时，需要说明这个类是什么时候、由谁创建的，在以后的更新中还要说明在什么时候由谁更新的，可以记录也可以不记录更新的内容，以往你会怎么做呢？是不是像这样在类的上面给类添加注释：

```
//更新：Matthew, 2008-2-10, 修改 ToString()方法
//更新：Jimmy, 2008-1-18
//创建：张子阳, 2008-1-15
public class DemoClass{
    // Class Body
}
```

这样的的确确是可以记录下来，但是如果有一天我们想将这些记录保存到数据库中作以备份呢？你是不是要一个一个地去查看源文件，找出这些注释，再一条条插入数据库中呢？

通过上面特性的定义，我们知道特性可以用于给类型添加元数据，这些元数据可以用于描述类型。那么在此处，特性应该会派上用场。那么在本例中，元数据应该是：注释类型(“更新”或者“创建”)，修改人，日期，备注信息(可有可无)。而特性的目标类型是 DemoClass 类。

按照对于附加到 DemoClass 类上的元数据的理解，我们先创建一个封装了元数据的类 RecordAttribute：

```
public class RecordAttribute {
    private string recordType;          // 记录类型：更新/创建
    private string author;              // 作者
    private DateTime date;              // 更新/创建 日期
    private string memo;                // 备注

    // 构造函数，构造函数的参数在特性中也称为“位置参数”。
    public RecordAttribute(string recordType, string author, string date) {
        this.recordType = recordType;
        this.author = author;
        this.date = Convert.ToDateTime(date);
```

```
    }

    // 对于位置参数，通常只提供 get 访问器
    public string RecordType {     get { return recordType; }     }
    public string Author { get { return author; } }
    public DateTime Date { get { return date; } }

    // 构建一个属性，在特性中也叫“命名参数”
    public string Memo {
        get { return memo; }
        set { memo = value; }
    }
}
```

NOTE：注意构造函数的参数 date，必须为一个常量、Type 类型、或者是常量数组，所以不能直接传递 DateTime 类型。

这个类不光看上去，实际上也和普通的类没有任何区别，显然不能它因为名字后面跟了个 Attribute 就摇身一变成了特性。那么怎样才能让它称为特性并应用到一个类上面呢？进行下一步之前，我们看看.Net 内置的特性 Obsolete 是如何定义的：

```
namespace System {
    [Serializable]
    [AttributeUsage(6140, Inherited = false)]
    [ComVisible(true)]
    public sealed class ObsoleteAttribute : Attribute {

        public ObsoleteAttribute();
        public ObsoleteAttribute(string message);
        public ObsoleteAttribute(string message, bool error);

        public bool IsError { get; }
        public string Message { get; }
    }
}
```

2.2 添加特性的格式(位置参数和命名参数)

首先，我们应该发现，它继承自 Attribute 类，这说明我们的 RecordAttribute 也应该继承自 Attribute 类。

其次，我们发现在这个特性的定义上，又用了三个特性去描述它。这三个特性分别是：Serializable、AttributeUsage 和 ComVisible。Serializable 特性我们前面已经讲述过，ComVisible 简单来说是“控制程序集中个别托管类型、成员或所有类型对 COM 的可访问

性”(微软给的定义)。这里我们应该注意到：特性本身就是用来描述数据的元数据，而这三个特性又用来描述特性，所以它们可以认为是“元数据的元数据”(元元数据：meta-metadata)。

因为我们需要使用“元元数据”去描述我们定义的特性 RecordAttribute，所以现在我们需要首先了解一下“元元数据”。这里应该记得“元元数据”也是一个特性，大多数情况下，我们只需要掌握 AttributeUsage 就可以了，所以现在就研究一下它。我们首先看上面 AttributeUsage 是如何加载到 ObsoleteAttribute 特性上面的。

```
    [AttributeUsage(6140, Inherited = false)]
```

然后我们看一下 AttributeUsage 的定义：

```
namespace System {
    public sealed class AttributeUsageAttribute : Attribute {
        public AttributeUsageAttribute(AttributeTargets validOn);

        public bool AllowMultiple { get; set; }
        public bool Inherited { get; set; }
        public AttributeTargets ValidOn { get; }
    }
}
```

可以看到，它有一个构造函数，这个构造函数含有一个 AttributeTargets 类型的位置参数(Positional Parameter)，还有两个命名参数(Named Parameter)。注意 ValidOn 属性不是一个命名参数，因为它不包含 set 访问器。

这里大家一定疑惑为什么会这样划分参数，这和特性的使用是相关的。假如 AttributeUsageAttribute 是一个普通的类，我们一定是这样使用的：

```
// 实例化一个 AttributeUsageAttribute 类
AttributeUsageAttribute usage=new AttributeUsageAttribute(AttributeTargets.Class)
;
usage.AllowMultiple = true;   // 设置 AllowMutiple 属性
usage.Inherited = false;// 设置 Inherited 属性
```

但是，特性只写成一行代码，然后紧靠其所应用的类型(目标类型)，那么怎么办呢？微软的软件工程师们就想到了这样的办法：不管是构造函数的参数 还是 属性，统统写到构造函数的圆括号中，对于构造函数的参数，必须按照构造函数参数的顺序和类型；对于属性，采用“属性=值”这样的格式，它们之间用逗号分隔。于是上面的代码就减缩成了这样：

```
[AttributeUsage(AttributeTargets.Class, AllowMutiple=true, Inherited=false)]
```

可以看出，AttributeTargets.Class 是构造函数参数(位置参数)，而 AllowMutiple 和

Inherited 实际上是属性(命名参数)。命名参数是可选的。将来我们的 RecordAttribute 的使用方式于此相同。(为什么管他们叫参数，我猜想是因为它们的使用方式看上去更像是方法的参数吧。)

假设现在我们的 RecordAttribute 已经 OK 了，则它的使用应该是这样的：

```
[RecordAttribute("创建","张子阳","2008-1-15",Memo="这个类仅供演示")]
public class DemoClass{
// ClassBody
}
```

其中 recordType, author 和 date 是位置参数，Memo 是命名参数。

2.3 AttributeTargets 位标记

从 AttributeUsage 特性的名称上就可以看出它用于描述特性的使用方式。具体来说，首先应该是其所标记的特性可以应用于哪些类型或者对象。从上面的代码，我们看到 AttributeUsage 特性的构造函数接受一个 AttributeTargets 类型的参数，那么我们现在就来了解一下 AttributeTargets。

AttributeTargets 是一个位标记，它定义了特性可以应用的类型和对象。

```
[Flags]
public enum AttributeTargets {

    Assembly = 1,               //可以对程序集应用属性。
    Module = 2,                 //可以对模块应用属性。
    Class = 4,                  //可以对类应用属性。
    Struct = 8,                 //可以对结构应用属性，即值类型。
    Enum = 16,                  //可以对枚举应用属性。
    Constructor = 32,           //可以对构造函数应用属性。
    Method = 64,                //可以对方法应用属性。
    Property = 128,             //可以对属性 (Property) 应用属性 (Attribute)。
    Field = 256,                //可以对字段应用属性。
    Event = 512,                //可以对事件应用属性。
    Interface = 1024,           //可以对接口应用属性。
    Parameter = 2048,           //可以对参数应用属性。
    Delegate = 4096,            //可以对委托应用属性。
    ReturnValue = 8192,         //可以对返回值应用属性。
    GenericParameter = 16384,   //可以对泛型参数应用属性。
    All = 32767,   //可以对任何应用程序元素应用属性。
}
```

现在应该不难理解为什么上面我范例中用的是：

```
[AttributeUsage(AttributeTargets.Class, AllowMutiple=true, Inherited=false)]
```

而 ObsoleteAttribute 特性上加载的 AttributeUsage 是这样的：

```
[AttributeUsage(6140, Inherited = false)]
```

因为 AttributeUsage 是一个位标记，所以可以使用按位或“|”来进行组合。所以，当我们这样写时：

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface)
```

意味着既可以将特性应用到类上，也可以应用到接口上。

NOTE：这里存在着两个特例：观察上面 AttributeUsage 的定义，说明特性还可以加载到程序集 Assembly 和模块 Module 上，而这两个属于我们的编译结果，在程序中并不存在这样的类型，我们该如何加载呢？可以使用这样的语法：[assembly:SomeAttribute(parameter list)]，另外这条语句必须位于程序语句开始之前。

2.4 Inherited 和 AllowMutiple 属性

AllowMutiple 属性用于设置该特性是不是可以重复地添加到一个类型上(默认为 false)，就好像这样：

```
[RecordAttribute("更新","Jimmy","2008-1-20")]
[RecordAttribute("创建","张子阳","2008-1-15",Memo="这个类仅供演示")]
public class DemoClass{
// ClassBody
}
```

所以，我们必须显示的将 AllowMutiple 设置为 True。

Inherited 就更复杂一些了，假如有一个类继承自我们的 DemoClass，那么当我们将 RecordAttribute 添加到 DemoClass 上时，DemoClass 的子类也会获得该特性。而当特性应用于一个方法，如果继承自该类的子类将这个方法覆盖，那么 Inherited 则用于说明是否子类方法是否继承这个特性。

在我们的例子中，将 Inherited 设为 false。

2.5 实现 RecordAttribute

现在实现 RecordAttribute 应该是非常容易了，对于类的主体不需要做任何的修改，我们只需要让它继承自 Attribute 基类，同时使用 AttributeUsage 特性标记一下它就可以了(假定我

们希望可以对类和方法应用此特性)：

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Method, AllowMultiple=true, Inherited=false)]
public class RecordAttribute:Attribute {
    // 略
}
```

2.6 使用 RecordAttribute

我们已经创建好了自己的自定义特性，现在是时候使用它了。

```
[Record("更新", "Matthew", "2008-1-20", Memo = "修改 ToString()方法")]
[Record("更新", "Jimmy", "2008-1-18")]
[Record("创建", "张子阳", "2008-1-15")]
public class DemoClass {
    public override string ToString() {
        return "This is a demo class";
    }
}

class Program {
    static void Main(string[] args) {
        DemoClass demo = new DemoClass();
        Console.WriteLine(demo.ToString());
    }
}
```

这段程序简单地在屏幕上输出一个“This is a demo class”。我们的属性也好像使用“//”来注释一样对程序没有任何影响，实际上，我们添加的数据已经作为元数据添加到了程序集中。可以通过 IL DASM 看到：

3）使用反射查看自定义特性

利用反射来查看 自定义特性信息 与 查看其他信息 类似，首先基于类型(本例中是 DemoClass)获取一个 Type 对象，然后调用 Type 对象的 GetCustomAttributes()方法，获取应用于该类型上的特性。当指定 GetCustomAttributes(Type attributeType, bool inherit) 中的第一个参数 attributeType 时，将只返回指定类型的特性，否则将返回全部特性；第二个参数指定是否搜索该成员的继承链以查找这些属性。

```
class Program {
    static void Main(string[] args) {
        Type t = typeof(DemoClass);
```

```
        Console.WriteLine("下面列出应用于 {0} 的 RecordAttribute 属性：" , t);

        // 获取所有的 RecordAttributes 特性
        object[] records = t.GetCustomAttributes(typeof(RecordAttribute), false);

        foreach (RecordAttribute record in records) {
            Console.WriteLine("    {0}", record);
            Console.WriteLine("        类型：{0}", record.RecordType);
            Console.WriteLine("        作者：{0}", record.Author);
            Console.WriteLine("        日期：{0}", record.Date.ToShortDateString());
            if(!String.IsNullOrEmpty(record.Memo)){
                Console.WriteLine("        备注：{0}",record.Memo);
            }
        }
    }
}
```

输出为：

下面列出应用于 AttributeDemo.DemoClass 的 RecordAttribute 属性：

```
    AttributeDemo.RecordAttribute
        类型：更新
        作者：Matthew
        日期：2008-1-20
        备注：修改 ToString()方法
    AttributeDemo.RecordAttribute
        类型：更新
        作者：Jimmy
        日期：2008-1-18
    AttributeDemo.RecordAttribute
        类型：创建
        作者：张子阳
        日期：2008-1-15
```

好了，到了这一步，我想将这些数据录入数据库中将不再是个问题，我们关于反射自定义特性的章节也就到此为止了。

5.2.4、动态创建类型实例

动态创建对象

在前面节中，我们先了解了反射，然后利用反射查看了类型信息，并学习了如何创建自

定义特性，并利用反射来遍历它。可以说，前面三节，我们学习的都是反射是什么，在接下来的章节中，我们将学习反射可以做什么。在进行更有趣的话题之前，我们先看下如何动态地创建一个对象。

我们新建一个 Console 控制台项目，叫做 Reflection4(因为本文是 Part4，你也可以起别的名字)。然后，添加一个示范类，本文中将通过对这个示范类的操作来进行说明：

```
public class Calculator {

    private int x;
    private int y;

    public Calculator(){
        x = 0;
        y = 0;
    }

    public Calculator(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

1）使用无参数构造函数创建对象

上面这个类非常简单，它包含两个构造函数，一个是有参数的构造函数，一个是无参数的构造函数，我们先看看通过反射，使用无参数的构造函数创建对象。创建对象通常有两种方式，一种是使用 Assembly 的 CreateInstance 方法：

```
Assembly asm = Assembly.GetExecutingAssembly();
Object obj = asm.CreateInstance("Reflection4.Calculator", true);
// 输出：Calculator() invoked
```

CreateInstance 的第一个参数代表了要创建的类型实例的字符串名称，第二个参数说明是不是大小写无关(Ignore Case)。注意到 CreateInstance 返回的是一个 Object 对象，意味着如果想使用这个对象，需要进行一次类型转换。
创建对象的另一种方式是调用 Activator 类的静态方法 CreateInstance：

```
ObjectHandle handler = Activator.CreateInstance(null, "Reflection4.Calculator");
Object obj = handler.Unwrap();
```

其中 CreateInstance 的第一个参数说明是程序集的名称，为 null 时表示当前程序集；第二个参数说明要创建的类型名称。Activator.CreateInstance 返回的是一个 ObjectHandle 对象，必须进行一次 Unwrap()才能返回 Object 类型，进而可以强制转换成我们需要的类型(本例中

是 Calculator)。ObjectHandle 包含在 System.Runtime.Remoting 命名空间中，可见它是 Remoting 相关的，对于 Remoting 我暂时没有做太多研究，我们现在只要知道可以通过这种方式创建对象就可以了。

2）使用有参数构造函数创建对象

如果我们想通过有参数的构造函数创建对象，我们可以使用 Assembly 的 CreateInstanc() 的重载方法：

```
// 有参数构造函数创建对象
Assembly asm = Assembly.GetExecutingAssembly();
Object[] parameters = new Object[2];      // 定义构造函数需要的参数
parameters[0] = 3;
parameters[1] = 5;

Object obj = asm.CreateInstance("Reflection4.Calculator", true, BindingFlags.Default, null, parameters, null, null);

// 输出：Calculator(int x, int y) invoked
```

我们看一下 CreateInstance 需要提供的参数：

- 前两个在前一小节已经说明过了；
- BindingFlags 在前面我们也用到过，它用于限定对类型成员的搜索。在这里指定 Default，意思是不使用 BingdingFlags 的策略(你可以把它理解成 null，但是 BindingFlags 是值类型，所以不可能为 null，必须有一个默认值，而这个 Default 就是它的默认值)；
- 接下来的参数是 Binder，它封装了 CreateInstance 绑定对象(Calculator)的规则，我们几乎永远都会传递 null 进去，实际上使用的是预定义的 DefaultBinder；
- 接下来是一个 Object[]数组类型，它包含我们传递进去的参数，有参数的构造函数将会使用这些参数；
- 接下来的参数是一个 CultureInfo 类型，它包含了关于语言和文化的信息(简单点理解就是什么时候 ToString("c")应该显示“￥”，什么时候应该显示“$”)。

动态调用方法

接下来我们看一下如何动态地调用方法。注意，本文讨论的调用不是将上面动态创建好的对象由 Object 类型转换成 Calculator 类型再进行方法调用，这和“常规调用”就没有区别了，让我们以.Net Reflection 的方式来进行方法的调用。继续进行之前，我们为 Calculator 添加两个方法，一个实例方法，一个静态方法：

```
public int Add(){
    int total= 0;
    total = x + y;
    Console.WriteLine("Invoke Instance Method: ");
```

```
        Console.WriteLine(String.Format("[Add]: {0} plus {1} equals to {2}", x, y, total));
        return total;
}

public static void Add(int x, int y){
        int total = x + y;
        Console.WriteLine("Invoke Static Method: ");
        Console.WriteLine(String.Format("[Add]: {0} plus {1} equals to {2}", x, y, total));
}
```

调用方法的方式一般有两种：

- 在类型的 Type 对象上调用 InvokeMember()方法，传递想要在其上调用方法的对象(也就是刚才动态创建的 Calculator 类型实例)，并指定 BindingFlags 为 InvokeMethod。根据方法签名，可能还需要传递参数。
- 先通过 Type 对象的 GetMethond()方法，获取想要调用的方法对象，也就是 MethodInfo 对象，然后在该对象上调用 Invoke 方法。根据方法签名，可能还需要传递参数。

需要说明的是，使用 InvokeMember 不限于调用对象的方法，也可以用于获取对象的字段、属性，方式都是类似的，本文只说明最常见的调用方法。

1）使用 InvokeMember 调用方法

我们先看第一种方法，代码很简单，只需要两行(注意 obj 在上节已经创建，是 Calculator 类型的实例)：

```
Type t = typeof(Calculator);
int result = (int)t.InvokeMember("Add", BindingFlags.InvokeMethod, null, obj, null);
Console.WriteLine(String.Format("The result is {0}", result));

输出：
Invoke Instance Method:
[Add]: 3 plus 5 equals to 8
The result is 8
```

在 InvokeMember 方法中，第一个参数说明了想要调用的方法名称；第二个参数说明是调用方法(因为 InvokeMember 的功能非常强大，不光是可以调用方法，还可以获取/设置 属性、字段等。此枚举的详情可参看 Part.2 或者 MSDN)；第三个参数是 Binder，null 说明使用默认的 Binder；第四个参数说明是在这个对象上(obj 是 Calculator 类型的实例)进行调用；最后一个参数是数组类型，表示的是方法所接受的参数。
我们在看一下对于静态方法应该如何调用：

```
Object[] parameters2 = new Object[2];
parameters2[0] = 6;
```

```
parameters2[1] = 9;
t.InvokeMember("Add", BindingFlags.InvokeMethod, null, typeof(Calculator), parameters2);

输出：
Invoke Static Method:
[Add]: 6 plus 9 equals to 15
```

我们和上面对比一下：首先，第四个参数传递的是 typeof(Calculator)，不再是一个 Calculator 实例类型，这很容易理解，因为我们调用的是一个静态方法，它不是基于某个具体的类型实例的，而是基于类型本身；其次，因为我们的静态方法需要提供两个参数，所以我们以数组的形式将这两个参数进行了传递。

2）使用 MethodInfo.Invoke 调用方法

我们再看下第二种方式，先获得一个 MethodInfo 实例，然后调用这个实例的 Invoke 方法，我们看下具体如何做：

```
Type t = typeof(Calculator);
MethodInfo mi = t.GetMethod("Add", BindingFlags.Instance | BindingFlags.Public);
mi.Invoke(obj, null);

输出：
Invoke Instance Method:
[Add]: 3 plus 5 equals to 8
请按任意键继续. . .
```

在代码的第二行，我们先使用 GetMethod 方法获取了一个方法对象 MethodInfo，指定 BindingFlags 为 Instance 和 Public，因为有两个方法都命名为“Add”，所以在这里指定搜索条件是必须的。接着我们使用 Invoke()调用了 Add 方法，第一个参数 obj 是前面创建的 Calculator 类型实例，表明在该实例上创建方法；第二个参数为 null，说明方法不需要提供参数。

我们再看下如何使用这种方式调用静态方法：

```
Type t = typeof(Calculator);
Object[] parameters2 = new Object[2];
parameters2[0] = 6;
parameters2[1] = 9;
MethodInfo mi = t.GetMethod("Add", BindingFlags.Static | BindingFlags.Public);
mi.Invoke(null, parameters2);
// mi.Invoke(t, parameters2);     也可以这样

输出：
Invoke Static Method:
```

```
[Add]: 6 plus 9 equals to 15
```

可以看到与上面的大同小异，在 GetMethod() 方法中，我们指定为搜索 BindingFlags.Static，而不是 BindingFlags.Public，因为我们要调用的是静态的 Add 方法。在 Invoke()方法中，需要注意的是第一个参数，不能在传递 Calculator 类型实例，而应该传递 Calculator 的 Type 类型或者直接传递 null。因为静态方法不是属于某个实例的。

NOTE：通过上面的例子可以看出：使用反射可以达到最大程度上的多态，举个例子，你可以在页面上放置一个 DropDownList 控件，然后指定它的 Items 的 value 为你某个类的方法的名称，然后在 SelectedIndexChanged 事件中，利用 value 的值来调用类的方法。而在以前，你只能写一些 if else 语句，先判断 DropDownList 返回的值，根据值再决定调用哪个方法。使用这种方式，编译器在代码运行之前(或者说用户选择了某个选项之前)完全不知道哪个方法将被调用，这也就是常说的 迟绑定(Late Binding)。

Coding4Fun：遍历 System.Drawing.Color 结构

我们已经讲述了太多的基本方法和理论，现在让我们来做一点有趣的事情：大家知道在 Asp.Net 中控件的颜色设置，比如说 ForeColor, BackColor 等，都是一个 System.Draw.Color 结构类型。在某些情况下我们需要使用自定义的颜色，那么我们会使用类似这样的方式 Color.FromRgb(125,25,13)创建一个颜色值。但有时候我们会觉得比较麻烦，因为这个数字太不直观了，我们甚至需要把这个值贴到 PhotoShop 中看看是什么样的。
这时候，我们可能会想要使用 Color 结构提供的默认颜色，也就是它的 141 个静态属性，但是这些值依然是以名称，比如 DarkGreen 的形式给出的，还是不够直观，如果能把它们以色块的形式输出到页面就好了，这样我们查看起来会方便的多，以后使用也会比较便利。我已经实现了它，可以点击下面的链接查看：
效果预览：http://www.tracefact.net/demo/reflection/color.aspx

基本实现

现在我们来看一下实现过程：

先创建页面 Color.aspx(或其他名字)，然后在 Head 里添加些样式控制页面显示，再拖放一个 Panel 控件进去。样式表需要注意的是#pnColors div 部分，它定义了页面上将显示的色块的样式；Id 为 pnHolder 的 Panel 控件用于装载我们动态生成的 div。

```
<head>
<style type="text/css">
body{font-size:14px;}
h1{font-size:26px;}
#pnColors div{
    float:left;width:140px;
    padding:7px 0;
    text-align:center;
    margin:3px;
    border:1px solid #aaa;
    font-size:11px;
```

```
        font-family:verdana, arial
}
</style>
</head>

<body>
        <h1>Coding4Fun：使用反射遍历 System.Drawing.Color 结构</h1>
        <form id="form1" runat="server">
            <asp:Panel ID="pnColors" runat="server"></asp:Panel>
        </form>
</body>
```

NOTE：如果将页面命名为了 Color.aspx，那么需要在代码后置文件中修改类名，比如改成：Reflection_Color，同时页面顶部也需要修改成 Inherits="Reflection_Color"，不然会出现命名冲突的问题。

下一步的思路是这样的：我们在 phColors 中添加一系列的 div，这些 div 也就是页面上我们将要显示的色块。我们设置 div 的文本为 颜色的名称 和 RGB 数值，它的背景色我们设为相应的颜色(色块的其他样式，比如宽、边框、宽度已经在 head 中定义)。我们知道在 Asp.Net 中，并没有一个 Div 控件，只有 HtmlGenericControl，此时，我们最好定义一个 Div 让它继承自 HtmlGenericControl。

```
public class Div:HtmlGenericControl
{
        private string name;

        public Div(Color c)
            : base("div")        // 调用基类构造函数，创建一个 Div
        {
            this.name = c.Name;          // 颜色名称

            // 设置文本
            this.InnerHtml = String.Format("{0}<br />RGB({1},{2},{3})", name, c.R, c.G, c.B);

            int total = c.R + c.G + c.B;
            if (total <= 255)        // 如果底色太暗，前景色改为明色调
                this.Style.Add("color", "#eee");

            // 设置背景颜色
            this.Style.Add("background", String.Format("rgb({0},{1},{2})", c.R, c.G, c.B));
        }
}
```

如同我们前面所描述的，这个 Div 接受一个 Color 类型作为构造函数的参数，然后在构

造函数中，先设置了它的文本为 颜色名称 和 颜色的各个数值(通过 Color 结构的 R, G, B 属性获得)。然后设置了 div 的背景色为相应的 RGB 颜色。

NOTE：在上面 if(total<=255)那里，可能有的颜色本身就很暗，如果这种情况再使用黑色的前景色那么文字会看不清楚，所以我添加了判断，如果背景太暗，就将前景色调的明亮一点。

OK，现在我们到后置代码中只要做一点点的工作就可以了：

```
protected void Page_Load(object sender, EventArgs e)
{
    List<Div> list = new List<Div>();

    Type t = typeof(Color);          // 页首已经包含了 using System.Drawing;
    // 获取属性
    PropertyInfo[] properties = t.GetProperties(BindingFlags.Static | BindingFlags.Public);
    Div div;

    // 遍历属性
    foreach (PropertyInfo p in properties)
    {
        // 动态获得属性
        Color c;
        c = (Color)t.InvokeMember(p.Name, BindingFlags.GetProperty, null, typeof(Color),
null);

        div = new Div(c);
        list.Add(div);
    }

    foreach (Div item in list)     {
        pnColors.Controls.Add(item);
    }
}
```

上面的代码是很直白的：先创建一个 Div 列表，用于保存即将创建的色块。然后获取 Color 类型的 Type 实例。接着我们使用 GetProperties()方法，并指定 BindingFlags 获取所有的静态公共属性。然后遍历属性，并使用 InvokeMember()方法获取了属性值，因为返回的是一个 Object 类型，所以我们需要把它强制转换成一个 Color 类型。注意在这里 InvokeMember 的 BindingFlags 指定为 GetProperty，意为获取属性值。第四个参数为 typeof(Color)，因为颜色属性(比如 DarkGreen)是静态的，不是针对于某个实例的，如果是实例，则需要传递调用此属性的类型实例。最后，我们根据颜色创建 div，并将它加入列表，遍历列表并逐一加入到 Id 为 pnColors 的 Panal 控件中。

现在已经 OK 了，如果打开页面，应该可以看到类似这样的效果：

为列表排序

上面的页面看上去会比较乱，因为列表大致是按颜色名称排序的(Transparnet 例外)，我们最好可以让列表基于颜色进行排序。关于列表排序，我在 基于业务对象的排序 一文中已经非常详细地进行了讨论，所以这里我仅给出实现过程，而不再进行讲述。这一小节与反射无关，如果你对排序已经非常熟悉，可以跳过。

在页面上添加一个 RadioButtonList 控件，将 AutoPostBack 设为 true，我们要求可以按名称和颜色值两种方式进行排序：

排序：

```
<asp:RadioButtonList ID="rblSort" runat="server" AutoPostBack="true"
RepeatDirection="Horizontal" RepeatLayout="Flow">
    <asp:ListItem Selected="True">Name</asp:ListItem>
    <asp:ListItem>Color</asp:ListItem>
</asp:RadioButtonList>
```

在后置代码中，添加一个枚举作为排序的依据：

```
public enum SortBy{
    Name,           // 按名称排序
    Color           // 暗颜色值排序
}
```

修改 Div 类，添加 ColorValue 字段，这个字段代表颜色的值，并创建嵌套类型 ColorComparer，以及方法 GetComparer：

```
public class Div:HtmlGenericControl
{
    private int colorValue;
    private string name;

    public Div(Color c)
        : base("div")       // 调用基类构造函数，创建一个 Div
    {
        this.name = c.Name;         // 颜色名称

        this.colorValue =           // 颜色的色彩值
            c.R * 256 * 256 + c.G * 256 + c.B;

        // 设置文本
        this.InnerHtml = String.Format("{0}<br />RGB({1},{2},{3})", name, c.R, c.G, c.B);

        int total = c.R + c.G + c.B;
        if (total <= 255)       // 如果底色太暗，前景色改为明色调
            this.Style.Add("color", "#eee");
```

```
            // 设置背景颜色
            this.Style.Add("background", String.Format("rgb({0},{1},{2})", c.R, c.G, c.B));
        }

        // 返回一个 Comparer()用于排序
        public static ColorComparer GetComparer(SortBy sort) {
            return new ColorComparer(sort);
        }

        // 默认以名称排序
        public static ColorComparer GetComparer() {
            return GetComparer(SortBy.Name);
        }

        // 嵌套类型，用于排序
        public class ColorComparer : IComparer<Div>
        {
            private SortBy sort;

            public ColorComparer(SortBy sort) {
                this.sort = sort;
            }

            // 实现 IComparer<T>接口，根据 sort 判断以何为依据一进排序
            public int Compare(Div x, Div y)
            {
                if (sort == SortBy.Name)
                    return String.Compare(x.name, y.name);
                else
                    return x.colorValue.CompareTo(y.colorValue);
            }
        }
}
```

在 Page_Load 事件上面，我们添加语句，获取当前的排序依据(枚举)：

```
SortBy sort;

if (!IsPostBack) {
    sort = SortBy.Name;
} else {
    sort = (SortBy)Enum.Parse(typeof(SortBy), rblSort.SelectedValue);
}
```

在将列表输出到页面之前，我们调用列表的 Sort 方法：

```
list.Sort(Div.GetComparer(sort));     // 对列表进行排序

foreach (Div item in list)     {
      pnColors.Controls.Add(item);
}
```

好了，所有工作都完成了，再次打开页面，可以看到类似如下画面，我们可以按照名称或者颜色值来对列表进行排序显示：

总结

本文分三个部分讲述了.Net 中反射的一个应用：动态创建对象和调用对象方法(属性、字段)。我们先学习最常见的动态创建对象的两种方式，随后分别讨论了使用 Type.InvokeMember()和 MethodInfo.Invoke()方法来调用类型的实例方法和静态方法。最后，我们使用反射遍历了 System.Drawing.Color 结构，并输出了颜色值。
感谢阅读，希望这篇文章能给你带来帮助！