第53章

C#、Visual Basic、C++/CLI 和 F#

本章内容：

- 名称空间
- 定义类型
- 方法
- 数组
- 控制语句
- 循环
- 异常处理
- 继承
- 资源管理
- 委托
- 事件
- 泛型
- LINQ 查询
- C++/CLI 混合本地代码和托管代码

C#是专为.NET 设计的编程语言。编写.NET 应用程序可以使用 50 多种语言，例如，Eiffel、Smalltalk、COBOL、Haskell、Pizza、Pascal、Delphi、Oberon、Prolog，以及 Ruby 等。Microsoft 还发布了 C#、Visual Basic、C++/CLI、J#、JScript.NET 和 F#。

每种语言都有优缺点；一些任务很容易用一种语言完成，但用另一种语言完成就很复杂。.NET Framework 中的类总相同，但语言的语法从.NET Framework 中抽象出了各种功能。例如，C#的 using 语句很便于使用实现 IDisposable 接口的对象。其他语言要实现该功能就需要较多代码。

Microsoft 最常用的.NET 语言是 C#和 Visual Basic。C#是专为.NET 设计的新语言，其理念来自于 C++、Java、Pascal 和其他语言。Visual Basic 植根于 Visual Basic 6，用.NET 的面向对象功能进行了扩展。

C++/CLI 是 C++的一种扩展，基于 ECMA 标准(ECMA 372)。C++/CLI 的一大优点是可以将本地代码与托管代码混合起来。我们可以扩展已有的本地 C++应用程序，添加.NET 功能，将.NET 类添加到本地库中，使它们能用于其他.NET 语言(如 C#)。还可以用 C++/CLI 编写完全托管的应用程序。

F#是 Visual Studio 中的一种新语言，它提供对函数编程的特殊支持。它可以与.NET 很好地合作，因为它也支持传统的面向对象编程。在 F#中，函数编程是指，函数可以用作值，这样，就很容易从

多个函数中构建函数，进行函数的通道化，其中函数逐个地链接起来。

本章介绍如何将.NET 应用程序从一种语言转换为另一种语言。Visual Basic 或 C++/CLI 示例代码可以映射到 C#，反之亦然。要进行这个比较，不需要学习 F#或 C++/CLI，因为本章主要介绍如何把 C#映射到其他语言上，而不讨论其他语言的核心概念和理念。

要理解本章的内容，读者应知道 C#，并已阅读了本书的前几章。但不必了解 Visual Basic、C++/CLI 和 F#。

53.1 名称空间

把.NET 类型组织到名称空间中。这 4 种语言定义和使用名称空间的语法完全不同。

为了导入名称空间，C#使用 using 关键字。C++/CLI 完全基于 C++语法和 using namespace 语句，Visual Basic 定义 Imports 关键字，F#定义 open 关键字，来导入名称空间。

在 C#和 Visual Basic 中，可以给类或其他名称空间定义别名。在 C++/CLI 中，别名只能引用其他名称空间，不能引用类。C++需要用 namespace 关键字定义别名，这个关键字也用于定义名称空间。Visual Basic 同样使用 Imports 关键字。

要定义名称空间，这 4 种语言都使用 namespace 关键字，但仍有一个区别。在 C++/CLI 中，不能用一条名称空间语句定义层次结构的名称空间，而必须嵌套名称空间。项目设置有一个重要的区别：在 C#的项目设置中定义名称空间，就是定义默认的名称空间，该名称空间会显示在添加到项目中的所有新项的代码中。在 Visual Basic 的项目设置中，定义项目中所有项使用的根名称空间。在源代码中声明的名称空间只定义根名称空间中的子名称空间。

```
// C#
using System;
using System.Collections.Generic;
using Assm = Wrox.ProCSharp.Assemblies;
namespace Wrox.ProCSharp.Languages
{
}

// C++/CLI
using namespace System;
using namespace System::Collections::Generic;
namespace Assm = Wrox::ProCSharp::Assemblies;
namespace Wrox
{
   namespace ProCSharp
   {
      namespace Languages
      {
   }
   }
}
' Visual Basic
Imports System
```

```
Imports System.Collections.Generic
Imports Assm = Wrox.ProCSharp.Assemblies
Namespace Wrox.ProCSharp.Languages

End Namespace

// F#
namespace Wrox.ProCSharp.Languages
open System
open System.Collections.Generic
```

53.2　定义类型

.NET 区分引用类型和值类型。在 C#中，引用类型用类定义，值类型用结构定义。除了引用类型和值类型之外，本节还介绍如何定义接口(引用类型)和枚举(值类型)。

53.2.1　引用类型

要声明引用类型，C#和 Visual Basic 使用 class 关键字。在 C++/CLI 中，类和结构基本相同，不能像 C#和 Visual Basic 那样区分引用类型和值类型。C++/CLI 有一个 ref 关键字，它定义托管类。定义 ref class 或 ref struct 可以创建引用类型。

在 C#和 C++/CLI 中，类用花括号括起来。C++/CLI 要求在类声明的最后加上分号。Visual Basic 在类的最后使用 End Class 语句。F#创建类和所有其他类型的方式是：使用 type 关键字和后面的类型名以及可选的 as 标识符，as 标识符定义实例标识符的名称：

```
// C#
public class MyClass
{
}

// C++/CLI
public ref class MyClass
{
};
public ref struct MyClass2
{
};

' Visual Basic
Public Class MyClass2
End Class

// F# signature file
type MyClass() as this =
// ... members of the class
```

在使用引用类型时，需要声明一个变量，必须在托管堆上给该对象分配内存。在声明引用类型的句柄时，C++/CLI 会定义句柄操作符^，它有点类似于 C++指针*。gcnew 操作符分配托管堆上的内存。使用 C++/CLI 还可以在本地声明变量，但对于引用类型，仍在托管堆上给该对象分配内存。在 Visual Basic 中，变量声明以 Dim 语句开头，其后是变量名。对于 new 和对象类型，需要在托管

堆上分配内存。

```
// C#
MyClass obj = new MyClass();
var obj2 = new MyClass()

// C++/CLI
MyClass^ obj = gcnew MyClass();
MyClass obj2;

' Visual Basic
Dim obj as New MyClass2()
```

如果引用类型没有引用内存，这 4 种语言就使用其他关键字：C#定义 null 字面量，C++/CLI 定义 nullptr(NULL 仅对本地对象有效)，Visual Basic 定义 Nothing。F#一般不使用 null 值——用 F#定义的类型不允许使用这个值。使用不是 F#语言定义的类型时，可能得到 null 值。

表 53-1 列出了预定义的引用类型。C++/CLI 没有像其他语言那样定义对象和字符串类型。当然，可以使用.NET Framework 定义的类。

表 53-1

.NET 类型	C#	C++/CLI	Visual Basic	F#
System.Object	object	未定义	Object	obj
System.String	string	未定义	String	string

53.2.2 值类型

要声明值类型，C#使用 struct 关键字；C++/CLI 使用 value 关键字，Visual Basic 使用 Structure 关键字，F#使用 type 关键字和[<Struct>]特性。另一种选择是使用 struct 关键字和 end 关键字：

```
// C#
public struct MyStruct
{
}

// C++/CLI
public value class MyStruct
{
};

' Visual Basic
Public Structure MyStruct
End Structure

// F#
[<Struct>]
type MyStruct =
   val x : int
type MyStruct2 =
   struct
        val x : int
   end
```

在 C++/CLI 中，可以把值类型分配到栈上、内置堆上(使用 new 操作符)或托管堆上(使用 gcnew 操作符)。C#和 Visual Basic 没有这些选项，但用 C++/CLI 混合本地代码和托管代码时，这些选项就变得非常重要。

```
// C#
MyStruct ms;

// C++/CLI
MyStruct ms1;
MyStruct* pms2 = new MyStruct();
MyStruct^ hms3 = gcnew MyStruct();

' Visual Basic
Dim ms as MyStruct
```

表 53-2 列出了这 4 种语言中预定义的值类型。在 C++/CLI 中，char 类型的大小为 1 字节，用于存储 ASCII 字符。在 C#中，char 类型的大小为 2 字节，用于存储 Unicode 字符，而 C++/CLI 为此使用 wchar_t 类型。C++的 ANSI 标准只定义了 short <= int <= long。在 32 位计算机上，int 和 long 的大小都是 32 位。要在 C++中定义 64 位的变量，需要使用 long long。

表　53-2

.NET 类型	C#	C++/CLI	Visual Basic	F#	大小
Char	char	wchar_t	Char	char	2 字节
Boolean	bool	bool	Boolean	bool	1 字节，包含 true 或 false
Int16	short	short	Short	int16	2 字节
UInt16	ushort	unsigned short	Ushort	uint16	2 字节，无符号
Int32	int	int	Integer	int	4 字节
UInt32	uint	unsigned Int	UInteger	uint	4 字节，无符号
Int64	long	long long	Long	int64	8 字节
UInt64	ulong	unsigned long long	ULong	uint64	8 字节，无符号

53.2.3　类型推断

C#允许在定义本地变量时，不显式声明数据类型，而使用 var 关键字。类型从指定的初始值中推断出来。在 Visual Basic 中，只要打开 Option infer，就可以使用 Dim 关键字推断变量的类型。这个功能需要使用编译器选项 infer+，或者在 Visual Basic 中使用项目配置页面。在 F#中，推断值、变量、参数和返回值的类型：

```
// C#
var x = 3;

' Visual Basic
Dim x = 3

// F#
let x = 3
```

53.2.4 接口

这 3 种语言在定义接口方面非常类似，它们都使用 interface 关键字。但 F#不同，因为其接口用仅包含抽象成员的类型定义：


```
   // C#
public interface IDisplay
{
   void Display();
}
```

代码段 CSharp/Person.cs


```
// C++/CLI
public interface class IDisplay
{
   void Display();
};
```

代码段 CPPCLI/Person.h


```
' Visual Basic
Public Interface IDisplay
   Sub Display
End Interface
```

代码段 VisualBasic/Person.vb


```
// F#
[<Interface>]
type public IDisplay
   abstract member Display : unit - > unit
```

代码段 FSharp/Person.fs

实现接口的方式则不同。C#和 C++/CLI 在类名后面加上冒号，之后是接口名，并实现用该接口定义的方法。在 C++/CLI 中，方法必须声明为 virtual。Visual Basic 使用 Implements 关键字实现接口，接口定义的方法也需要附加 Implements 关键字。F#在类型定义中列出已实现的接口。接口用 interface 和 with 关键字实现，并实现下述接口的所有成员：


```
   // C#
public class Person: IDisplay
{
   public void Display()
   {
   }
}
```

代码段 CSharp/Person.cs


```
// C# explicit interface implementation
public class Person: IDisplay
{
   void IDisplay.Display()
   {
   }
}
```

```
// C++/CLI
public ref class Person: IDisplay
{
public:
   virtual void Display() { }
};
```

代码段 CPPCLI/Person.h


```
' Visual Basic
Public Class Person
   Implements IDisplay
   Public Sub Display Implements IDisplay.Display
   End Sub
End Class
```

代码段 VisualBasic/Person.vb


```
// F#
type Person as this =
  interface IDisplay with
     member this.Display() = printfn "%s %s" this.FirstName this.LastName
```

代码段 FSharp/Person.fs

53.2.5　枚举

这 3 种语言在定义枚举方面非常类似，它们都使用 enum 关键字(只有 Visual Basic 使用新的一行代码，而不是用逗号分隔元素)。F#使用 type 关键字和匹配表达式“|”：


```
// C#
public enum Color
{
   Red, Green, Blue
}
```

代码段 CSharp/Color.cs


```
// C++/CLI
public enum class Color
{
   Red, Green, Blue
};
```

代码段 CPPCLI/Color.h


```
' Visual Basic
Public Enum Color
   Red
   Green
   Blue
End Enum
```

代码段 VisualBasic/Color.vb


```
// F#
type Color =
| Red = 0
| Green = 1
```

```
    | Blue = 2
```

代码段 FSharp/Color.fs

53.3 方法

除了 F#之外，总是在类中声明方法。C++/CLI 的语法非常类似于 C#，除了访问修饰符不包含在方法声明中，而是写在方法声明之前。访问修饰符必须用冒号结束。在 Visual Basic 中，使用 Sub 关键字定义方法。在 F#中，可以用 let 关键字定义方法。Foo 有一个参数 x，返回 x 加 3 的结果。参数类型和返回类型可以用函数声明，如下所示。对于 add 函数，x 和 y 的类型是 int，返回一个 int。如果没有声明类型，类型就由编译器推断。对于 Foo 函数，x 也是 int，因为 3 要与 x 相加，而 3 是 int。在类中定义方法使用 member 关键字。实例方法有一个用类定义中的 as 标识符定义的前缀。虽然 F#不需要使用 this、Me 或 self 访问当前实例，但可以定义任意标识符。

```
// C#
public class MyClass
{
   public void Foo()
   {
   }
}

// C++/CLI
public ref class MyClass
{
public:
   void Foo()
   {
   }
};

' Visual Basic
Public Class MyClass1
   Public Sub Foo
   End Sub
End Class

// F# function
let foo x = x + 3
let add (x : int, y : int) : int = x + y

// method within a class
type MyClass as this =
   member this.Foo() =
      printfn "MyClass.Foo"
```

53.3.1 方法的参数和返回类型

在 C#和 C++/CLI 中，传递给方法的参数在圆括号中定义。参数的类型在变量名的前面声明。如果从方法中返回一个值，该方法就用返回值的类型定义，而不是 void。

Visual Basic 使用 Sub 语句声明没有返回值的方法，用 Function 语句声明有返回类型的方法。返

回类型放在方法名和圆括号的后面。参数中的变量声明和类型在 Visual Basic 中的顺序也不同：类型在变量的后面，而在 C#和 C++/CLI 中，变量在类型的后面。

在 F#中，如果方法没有返回值，它就声明为 unit，unit 类似于 C#中的 void。Foo()方法声明为接收一个 int 参数，并返回一个 int：

```
// C#
public class MyClass
{
   public int Foo(int i)
   {
      return 2 * i;
   }
}

// C++/CLI
public ref class MyClass
{
public:
   int Foo(int i)
   {
      return 2 * i;
   }
};
' Visual Basic
Public Class MyClass2
   Public Sub Foo1(ByVal i as Integer)
   End Sub
   Public Function Foo(ByVal i As Integer) As Integer
      Return 2 * i
   End Sub
End Class

// F#
type MyClass as this =
   member this.Foo(i : int) : int = i * 2
```

53.3.2　参数修饰符

在默认情况下，值类型按值传递，引用类型按引用传递。如果作为参数传递的值类型要在主调方法中更改，那么 C#允许使用参数修饰符 ref。

C++/CLI 定义一个托管引用操作符“%”。这个操作符类似于 C++引用操作符“&”，但“%”可以用于托管类型，垃圾收集器可以跟踪这些对象，以防它们在托管堆中移动。

Visual Basic 使用 ByRef 关键字按引用传递参数。

```
// C#
public class ParameterPassing
{
   public void ChangeVal(ref int i)
   {
      i = 3;
   }
}

// C++/CLI
```

```
public ref class ParameterPassing
{
public:
   int ChangeVal(int% i)
   {
      i = 3;
   }
};

' Visual Basic
Public Class ParameterPassing
   Public Sub ChangeVal(ByRef i as Integer)
      i = 3
   End Sub
End Class
```

调用带引用参数的方法时，只有 C#语言需要使用参数修饰符。C++/CLI 和 Visual Basic 在调用带或不带参数修饰符的方法方面没有区别。这方面 C#有优势，因为可以立即看出，参数值可以在主调方法中更改。

由于调用语法没有区别，因此 Visual Basic 不允许在重载方法时仅更改修饰符。C++/CLI 编译器允许在重载方法时仅更改修饰符，但不能编译调用者，因为这个解析方法有多义性。虽然在 C#中允许重载并使用只有参数修饰符的方法，但这不是一个好的编程习惯。

```
// C#
     ParameterPassing obj = new ParameterPassing();
     int a = 1;
     obj.ChangeVal(ref a);
     Console.WriteLine(a); // writes 3

// C++/CLI
     ParameterPassing obj;
     int a = 1;
     obj.ChangeVal(a);
     Console.WriteLine(a); // writes 3

' Visual Basic
     Dim obj as new ParameterPassing()
     Dim i as Integer = 1
     obj.ChangeVal(i)
     Console.WriteLine(i) // writes 3
```

当从方法中返回一个参数时，C#还定义了 out 关键字。这个选项在 C++/CLI 和 Visual Basic 都不可用。只要调用者和被调用者在相同的应用程序域中，out 和 ref 在后台就没有区别。用 C# 的 out 参数修饰符声明的方法也可以在 C++/CLI 和 Visual Basic 中以与 ref 参数修饰符相同的方式调用。如果方法要跨多个应用程序域或进程使用，C++/CLI 和 Visual Basic 就可以使用[out]特性。

53.3.3 构造函数

在 C#和 C++/CLI 中，构造函数与类同名。Visual Basic 使用 New 过程。this 和 Me 关键字用于

访问这个实例的成员。在一个构造函数中调用另一个构造函数时，C#需要初始化一个成员。C++/CLI 和 Visual Basic 可以将构造函数作为方法调用。

F#有点区别，它使用类型声明定义构造函数。Person 类的构造函数接受两个 string 参数，其他构造函数使用 new 关键字定义。new()方法定义没有参数的构造函数。对于 Person 类，用两个参数调用构造函数。


```
// C#
public class Person
{
   public Person()
      : this("unknown", "unknown") { }
   public Person(string firstName, string lastName)
   {
      this.firstName = firstName;
      this.lastName = lastName;
   }
   private string firstName;
   private string lastName;
}
```

代码段 CSharp/Person.cs


```
// C++/CLI
public ref class Person
{
public:
   Person()
   {
      Person("unknown", "unknown");
   }
   Person(String^ firstName, String^ lastName)
   {
      this->firstName = firstName;
      this->lastName = lastName;
   }
private:
   String^ firstName;
   String^ lastName;
};
```

代码段 CPPCLI/Person.h


```
' Visual Basic
Public Class Person
   Public Sub New()
      Me.New("unknown", "unknown")
   End Sub
   Public Sub New(ByVal firstName As String, ByVal lastName As String)
      Me.MyFirstName = firstName
      Me.MyLastName = lastName
   End Sub
   Private MyFirstName As String
   Private MyLastName As String
End Class
```

代码段 VisualBasic/Person.vb


```
// F#
type Person(firstName0 : string, lastName0 : string) as this =
   let mutable firstName, lastName = firstName0, lastName0
   new () = Person("unknown", "unknown")
```

代码段 FSharp/Person.fs

53.3.4 属性

要定义属性，在属性块中 C#需要 get 和 set 存取器。C#、C++/CLI 和 Visual Basic 中还有一种速记表示法：如果 get 和 set 存取器只返回或设置一个简单的变量，就不需要自定义实现代码。这是一个自动属性。其语法与 C++/CLI 和 Visual Basic 不同，这两种语言都有 property 关键字，且需要用 set 存取器定义变量的值。C++/CLI 还需要用 get 存取器指定返回类型，用 set 存取器指定参数类型。

C++/CLI 在用简短版本编写属性时，只需定义属性的类型和名称，get 和 set 存取器由编译器自动创建。如果除了设置和返回变量之外不需要做其他工作，就可以使用这个简短版本。如果存取器的实现代码还需要做其他工作(例如，检查变量或刷新)就必须使用属性的完整语法。

F#把属性定义为类的成员，使用 get()作为 get 存取器，把 set (value)作为 set 存取器。


```
  // C#
public class Person
{
   private string firstName;
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }
   public string LastName { get; set; }
}
```

代码段 CSharp/Person.cs


```
// C++/CLI
public ref class Person
{
private:
   String^ firstName;
public:
   property String^ FirstName
   {
      String^ get()
      {
         return firstName;
      }
      void set(String^ value)
      {
         firstName = value;
      }
   }
   property String^ LastName;
};
```

代码段 CPPCLI/Person.h


```
' Visual Basic
Public Class Person
   Private myFirstname As String
   Public Property FirstName()
      Get
         Return myFirstName
      End Get
      Set(ByVal value)
         myFirstName = value
      End Set
   End Property
   Public Property LastName As String
End Class
```

代码段 VisualBasic/Person.vb


```
  // F#
type Person(firstName0 : string, lastName0 : string) as this =
   let mutable firstName, lastName = firstName0, lastName0
   member this.FirstName
      with get() = firstName
      and set(value) = firstName <- value
   member this.LastName
      with get() = lastName
      and set(value) = lastName <- value
```

代码段 FSharp/Person.fs

在 C#和 C++/CLI 中，只读属性只有 get 存取器。在 Visual Basic 中，还必须指定 ReadOnly 修饰符，只写属性必须用 WriteOnly 修饰符和 set 存取器定义。

```
' Visual Basic
   Public ReadOnly Property Name()
   Get
      Return myFirstName & " " & myLastName
   End Get
End Property
```

53.3.5　对象初始值设定项

在 C#和 Visual Basic 中，属性可以使用对象初始值设定项进行初始化。属性可以使用类似于数组初始值设定项的花括号初始化。C#和 Visual Basic 中的语法非常类似，Visual Basic 仅使用 With 关键字：

```
// C#
Person p = new Person { FirstName = "Tom", LastName = "Turbo" };

' Visual Basic
Dim p As New Person With { .FirstName = "Tom", .LastName = "Turbo" }
```

53.3.6　扩展方法

扩展方法是 LINQ 的基础。在 C#和 Visual Basic 中，可以创建扩展方法。但是其语法不同。C#在第一个参数中用 this 关键字标记扩展方法，Visual Basic 用<Extension>特性标记扩展方法。在 F#中，扩展语法的术语是类型扩展。类型扩展通过 type 声明指定类型的完全限定名称，并用 with 关键字扩展它：

```
// C#
public static class StringExtension
{
   public static void Foo(this string s)
   {
      Console.WriteLine(""Foo {0}", s);
   }
}

' Visual Basic
Public Module StringExtension
   < Extension() > _
   Public Sub Foo(ByVal s As String)
      Console.WriteLine(""Foo {0}", s)
   End Sub
End Module

// F#
module Wrox.ProCSharp.Languages.StringExtension
type System.String with
   member this.Foo = printfn "String.Foo"
```

53.4 静态成员

静态字段只为某种类型的所有对象实例化一次。C#和 C++/CLI 都使用 static 关键字；Visual Basic 则使用 Shared 关键字。

使用静态成员的方法是：指定类名，其后是“.”操作符和静态成员名。C++/CLI 使用“:”操作符访问静态成员。


```
// C#
public class Singleton
{
   private static SomeData data = null;
   public static SomeData GetData()
   {
      if (data == null)
      {
         data = new SomeData();
      }
      return data;
   }
}
// use:
SomeData d = Singleton.GetData();
```

代码段 CSharp/Singleton.cs


```
// C++/CLI
public ref class Singleton
{
private:
   static SomeData^ hData;
```

```
public:
   static SomeData^ GetData()
   {
      if (hData == nullptr)
      {
         hData = gcnew SomeData();
      }
      return hData;
   }
};
// use:
SomeData^ d = Singleton::GetData();
```

代码段 CPPCLI/Singleton.h

可从 wrox.com 下载源代码

```
  ' Visual Basic
Public Class Singleton
   Private Shared data As SomeData
   Public Shared Function GetData() As SomeData
      If data is Nothing Then
         data = new SomeData()
      End If
      Return data
   End Function
End Class
' Use:
Dim d as SomeData = Singleton.GetData()
```

代码段 VisualBasic/Singleton.fs

53.5　数组

数组在第 6 章讨论过。Array 类总是后台的.NET 数组。声明数组时，编译器会创建一个派生自 Array 基类的类。在设计 C#时，采用了 C++数组的方括号语法，并用数组初始化值设定项扩展它。

```
// C#
int[] arr1 = new int[3] {1, 2, 3};
int[] arr2 = {1, 2, 3};
```

如果在 C++/CLI 中使用方括号，就会创建一个本地 C++数组，而不是基于 Array 类的数组。为了创建.NET 数组，C++/CLI 引入了 array 关键字。这个关键字使用类似于泛型的语法，即使用尖括号。在尖括号中指定元素的类型。C++/CLI 支持数组初始化值设定项的语法与 C#相同。

```
// C++/CLI
array<int>^ arr1 = gcnew array<int>(3) {1, 2, 3};
array<int>^ arr2 = {1, 2, 3};
```

Visual Basic 给数组使用括号。它要求在数组声明中指定最后一个元素编号，而不是数组中的元素个数。在每种.NET 语言中，数组都以元素编号 0 开始，Visual Basic 也是如此。为了使之更清楚，Visual Basic 9 在数组声明中引入了 0 To number 表达式。它总是从 0 开始，To 使该语法可读性更强。

如果数组用 new 操作符初始化，那么 Visual Basic 也支持数组初始化值设定项。

```
' Visual Basic
Dim arr1(0 To 2) As Integer()
Dim arr2 As Integer() = New Integer(0 To 2) {1, 2, 3};
```

F#提供了初始化数组的不同方式。创建小型数组时，可以使用“[|”和“|]”作为左花括号和右花括号，并指定所有元素，这些元素用分号隔开。序列表达式(如 for in 语句)可以用于初始化数组中的元素，如下面的 arr2。arr3 在初始化时用 zeroCreate 类型扩展为 Array 类，以创建并初始化 20 个元素。可以使用索引器访问数组。对于数组片段，可以访问数组中的一个范围，例如，arr2.[4..]从第 5 个元素开始访问，直到最后一个元素。

```
// F#
let arr1 = [| 1; 2; 3 |]
let arr2 = [| for i in 1..10 -> i |]
let arr3 : int aray = Array.zeroCreate 20
```

53.6 控制语句

控制语句指定应运行什么代码。C#定义了 if 和 switch 语句，以及条件操作符。

53.6.1 if 语句

C#的 if 语句与 C++/CLI 版本相同，Visual Basic 使用 If-Then/Else/End If 来替代花括号。

```
// C# and C++/CLI
if (a == 3)
{
   // do this
}
else
{
   // do that
}
```

```
' Visual Basic
If a = 3 Then
   ' do this
Else
   ' do that
End If
```

53.6.2 条件操作符

C#和 C++/CLI 支持条件操作符，它是 if 语句的一个轻型版本。在 C++/CLI 中，这个操作符称为三元操作符。第一个参数必须有一个布尔值。如果结果为 true，就计算第一个表达式；否则，就计算第二个表达式。Visual Basic 在 Visual Basic Runtime Library 中提供了有类似功能的 IIf 函数。F#把 if/then/else 用作条件操作符

```
// C#
string s = a > 3 ? "one": "two";
```

```
// C++/CLI
```

```
String^ s = a > 3 ? "one": "two";

' Visual Basic
Dim s As String = IIf(a > 3, "one", "two")

// F#
if a = 3 then printfn "do this" this else printfn "do that"
```

53.6.3　switch 语句

C#和 C++/CLI 中的 switch 语句看起来很类似，但它们有重要的区别。C#支持在 case 选项中使用字符串，但 C++不支持。在 C++中，必须使用 if-else。C++/CLI 支持从一个 case 选项向下一个 case 选项隐式跳转。但在 C#中，如果没有 break 或 goto 语句，编译器就会发出警告。只有 case 中没有语句时，C#才支持从一个 case 选项向下一个 case 选项隐式跳转。

Visual Basic 用 Select/Case 语句替代 switch/case 语句。它不需要也不允许使用 break 语句。即使 Case 后面没有一条语句，也不能从一个 case 选项向下一个 case 选项隐式跳转。Case 可以用 And、Or 和 To 合并，如 3 To 5。

F#用 match 和 with 关键字提供匹配表达式。匹配表达式的每一行都以“|”开头，后跟一种模式。下面的示例对 Suit.Hear | Suit.Diamond 使用 OR 模式。最后一个模式是通配符模式：

```
// C#
string GetColor(Suit s)
{
   string color;
   switch (s)
   {
      case Suit.Heart:
      case Suit.Diamond:
         color = "Red";
         break;
      case Suit.Spade:
      case Suit.Club:
         color = "Black";
         break;
      default:
         color = "Unknown";
         break;
   }
   return color;
}

// C++/CLI
String^ GetColor(Suit s)
{
   String^ color;
   switch (s)
   {
      case Suit::Heart:
      case Suit::Diamond:
         color = "Red";
         break;
      case Suit::Spade:
      case Suit::Club:
```

```
            color = "Black";
            break;
         default:
            color = "Unknown";
            break;
      }
      return color;
   }

   ' Visual Basic
   Function GetColor(ByVal s As Suit) As String
      Dim color As String = Nothing
      Select Case s
         Case Suit.Heart And Suit.Diamond
            color = "Red"
         Case Suit.Spade And Suit.Club
            color = "Black"
         Case Else
            color = "Unknown"
      End Select

      Return color
   End Function

   // F#
   type SuitTest =
      static member GetColor(s : Suit) : string =
         match s with
         | Suit.Heart | Suit.Diamond -> "Red"
         | Suit.Spade | Suit.Club -> "Black"
         | _ -> "Unknown"
```

53.7 循环

使用循环，代码会重复执行，直到满足一个条件为止。C#中的循环详见第 2 章，包括 for、while、do…while 和 foreach。C#和 C++/CLI 的循环语句非常类似，而 Visual Basic 和 F#定义了不同的语句。

53.7.1 for 语句

C#和 C++/CLI 的 for 语句很类似。在 Visual Basic 中，不能在 For/To 语句中初始化变量，而必须提前初始化变量。for/To 不需要使用 Step 语句，因为默认使用 Step 1。只有不打算将递增量设置为 1，for/To 才需要使用 Step 关键字。F#使用 for/to/do 和 for/downto/do 关键字。

```
// C#
for (int i = 0; i < 100; i++)
{
   Console.WriteLine(i);
}

// C++/CLI
for (int i = 0; i < 100; i++)
{
   Console::WriteLine(i);
}
```

```
' Visual Basic
Dim count as Integer
For count = 0 To 99 Step 1
   Console.WriteLine(count)
Next

// F#
for i = 1 to 10 do
   printfn i
for i = 10 downto 1 do
   printfn i
```

53.7.2　while 和 do…while 语句

while 和 do…while 语句在 C#和 C++/CLI 中相同。Visual Basic 中的 Do While/Loop 和 Do/Loop While 的结构与它们非常类似。F#没有 do/while，但有 while/do。

```
// C#
int i = 0;
while (i < 3)
{
   Console.WriteLine(i++);
}
i = 0;
do
{
   Console.WriteLine(i++);
} while (i < 3);

// C++/CLI
int i = 0;
while (i < 3)
{
   Console::WriteLine(i++);
}
i = 0;
do
{
   Console::WriteLine(i++);
} while (i < 3);

' Visual Basic
Dim num as Integer = 0
Do While (num < 3)
   Console.WriteLine(num)
   num += 1
Loop
num = 0
Do
   Console.WriteLine(num)
   num += 1
Loop While (num < 3)

// F#
let i = 0
let mutable looping = true
```

```
while looping do
printfn "%d" i
i++
if i > = 3
   looping < - false
```

53.7.3 foreach 语句

foreach 语句使用 IEumerable 接口。foreach 语句在 ANSI C++中不存在，但它是 ANSI C++/CLI 中的一个扩展。与 C#的 foreach 语句不同，在 C++/CLI 中，for 和 each 之间有一个空格。F#使用 for/in/do 关键字提供这个功能。

```
// C#
int[] arr = {1, 2, 3};
foreach (int i in arr)
{
   Console.WriteLine(i);
}

// C++/CLI
array<int>^ arr = {1, 2, 3};
for each (int i in arr)
{
   Console::WriteLine(i);
}

' Visual Basic
Dim arr() As Integer = New Integer() {1, 2, 3}
Dim num As Integer
For Each num as Integer In arr
Console.WriteLine(num)
Next

// F#
for i in arr do
   printfn i
```

foreach 很容易遍历集合，而 C#允许使用 yield 语句创建枚举。在 Visual Basic 和 C++/CLI 中 yield 语句不可用，而必须手工实现 IEnumerable 和 IEnumerator 接口。yield 语句参见第 6 章。

53.8 异常处理

异常处理详见第 15 章。这 3 种语言的异常处理非常类似。它们都使用 try/catch/finally 处理异常，用 throw 关键字创建异常。F#使用 raise 抛出异常，使用 try/with/finally 处理异常：

可从 wrox.com 下载源代码

```
// C#
public void Method(Object o)
{
   if (o == null)
```

```
        throw new ArgumentException("Error");
  }
  public void Foo()
  {
  try
  {
     Method(null);
  }
  catch (ArgumentException ex)
  { }
  catch (Exception ex)
  { }
  finally
  { }
}
```

代码段 CSharp/ExceptionDemo.cs


```
// C++/CLI
public:
   void Method(Object^ o)
   {
      if (o == nullptr)
         throw gcnew ArgumentException("Error");
   }
   void Foo()
   {
      try
      {
         Method(nullptr);
      }
      catch (ArgumentException^ ex)
      { }
      catch (Exception^ ex)
      { }
      finally
      { }
   }
```

代码段 CPPCLI/ExceptionDemo.h


```
  ' Visual Basic
Public Sub Method(ByVal o As Object)
   If o = Nothing Then
      Throw New ArgumentException("Error")
End Sub
Public Sub Foo()
   Try
      Method(Nothing)
   Catch ex As ArgumentException
      '
   Catch ex As Exception
      '
   Finally
      '
   End Try
End Sub
```

代码段 VisualBasic/ExceptionDemo.vb


可从
wrox.com
下载源代码

```
// F#
type ExceptionDemo() as this =
   member this.Method(o : obj) =
      if o = null then
         raise(ArgumentException("error"))
   member this.Foo =
      try
         try
            this.Method(null)
         with
            | :? ArgumentException as ex -> printfn "%s" ex.Message
            | :? IOException as ex -> printfn "%s" ex.Message
      finally
         printfn "finally"
```

代码段 FSharp/ExceptionDemo.fs

53.9 继承

.NET 提供了许多关键字，用于定义多态行为，用来重写或隐藏方法；以及访问修饰符，以允许访问或不允许访问类的成员。C#的这个功能参见第 4 章。C#、C++/CLI 和 Visual Basic 的功能非常类似，但关键字不同。

53.9.1 访问修饰符

C++/CLI、Visual Basic 和 F#的访问修饰符非常类似于 C#，但有一些显著的区别。Visual Basic 使用 Friend 访问修饰符替代 internal，访问同一个程序集中的类型。C++/CLI 还有一个访问修饰符 protected private。internal protected 允许访问同一个程序集中的成员，还可以访问其他程序集中派生自基类的类型。C#和 Visual Basic 不允许访问同一个程序集中的派生类型，但在 C++/CLI 中这可以使用 protected private 来实现。这里 private 表示在程序集的外部不能访问，但在程序集的内部可以进行受保护的访问。其顺序(不管是 protected private 或 private protected)并不重要。访问修饰符在程序集中总是允许访问较多的内容，在程序集的外部总是允许访问较少的内容。F#使用 3 个访问修饰符，还允许把 protected 修饰符用于在其他.NET 语言中使用的类型。F#中的访问修饰符可以在签名文件中声明。

每种语言的访问修饰符见表 53-3。

表 53-3

C#	C++/CLI	Visual Basic	F#
public	Public	Public	public
protected	Protected	Protected	
private	Private	Private	private
internal	Internal	Friend	internal
internal protected	internal protected	Protected Friend	
无	protected private	无	

F#签名文件的扩展名是.fsi，它定义成员的签名。这种文件可以从代码文件中通过-- sig 编译器选项自动创建。根据不同的需求，只需要更改访问修饰符：

```
// F#
type public Person =
   class
      interface IDisplay
      public new : unit -> Person
      public new : firstName:string * lastName:string -> Person
      override ToString : unit -> string
      member public FirstName : string
      member public LastName : string
      member public FirstName : string with set
      member public LastName : string with set
   end
```

53.9.2　关键字

对继承很重要的关键字如表 53-4 所示。

表　53-4

C#	C++/CLI	Visual Basic	F#	功　能
:	:	Implements	interface with	实现一个接口
:	:	Inherits	inherit	继承自一个基类
virtual	virtual	Overridable	abstract	声明一个支持多态性的方法
overrides	overrides	Overrides	override	重写一个虚方法
new	new	Shadows		隐藏基类中的方法
abstract	abstract	MustInherit	[< AbstractClass]	抽象类
sealed	sealed	NotInheritable	[< Sealed >]	密封类
abstract	abstract	MustOverride	abstract	抽象方法
sealed	sealed	NotOverridable		密封方法
this	this	Me		引用当前对象
base	Classname::	MyBase	base	引用基类

关键字的放置顺序在各种语言中很重要。在代码示例中，抽象基类 Base 有一个抽象方法和一个已实现的虚方法。Drived 类派生自 Base，它实现抽象方法，并重写虚方法。


```
// C#
public abstract class Base
{
   public virtual void Foo()
   {
   }
   public abstract void Bar();
}
public class Derived: Base
{
```

```
    public override void Foo()
    {
       base.Foo();
    }
    public override void Bar()
    {
    }
}
```

代码段 CSharp/InheritanceDemo.cs


```
// C++/CLI
public ref class Base abstract
{
public:
   virtual void Foo()
   {
   }
   virtual void Bar() abstract;
};
public ref class Derived: public Base
{
public:
   virtual void Foo() override
   {
      Base::Foo();
   }
   virtual void Bar() override
   {
   }
};
```

代码段 CPPCLI/InheritanceDemo.h


```
' Visual Basic
Public MustInherit Class Base
   Public Overridable Sub Foo()
   End Sub
   Public MustOverride Sub Bar()
End Class
Public class Derived
   Inherits Base
   Public Overrides Sub Foo()
      MyBase.Foo()
   End Sub
   Public Overrides Sub Bar()
   End Sub
End Class
```

代码段 VisualBasic/InheritanceDemo.vb


```
// F#
[<AbstractClass>]
type Base() as this =
   abstract Foo : unit -> unit
   default this.Foo() = printfn "Base.Foo"
   abstract Bar : unit -> unit
```

```
type Derived() as this =
   inherit Base()
   override this.Foo() =
      base.Foo()
      printfn "Derived.Foo"
override this.Bar() = printfn "Derived.Bar"
```

代码段 FSharp/InheritanceDemo.fs

53.10　资源管理

第 13 章介绍了资源管理，同时实现 IDisposable 接口和一个终结器。本节介绍这些功能在 C++/CLI、Visual Basic 和 F#中如何实现。

53.10.1　IDisposable 接口的实现

为了释放资源，IDisposable 接口定义 Dispose()方法。使用 C#、Visual Basic 和 F#时，必须实现 IDisposable 接口。在 C++/CLI 中，也实现 IDisposable 接口，但如果只编写一个析构函数，这将由编译器完成。


```
// C#
public class Resource: IDisposable
{
   public void Dispose()
   {
      // release resource
   }
}
```

代码段 CSharp/Resource.cs


```
// C++/CLI
public ref class Resource
{
public:
   ~Resource()
   {
      // release resource
   }
};
```

代码段 CPPCLI/Resource.h


```
' Visual Basic
Public Class Resource
   Implements IDisposable
   Public Sub Dispose() Implements IDisposable.Dispose
      ' release resource
   End Sub
End Class
```

代码段 VisualBasic/Resource.vb

```
// F#
type Resource() as this =
```

```
interface IDisposable with
member this.Dispose() = printfn "release resource"
```

代码段 FSharp/Resource.fs

在 C++/CLI 中，使用 delete 语句调用 Dispose()方法。

53.10.2 using 语句

C#的 using 语句使用“获取/使用/释放”模式释放不再使用的资源，甚至在出现异常的情况下也是如此。编译器创建一个 try/finally 语句，在 finally 语句中调用 Dispose()方法。Visual Basic 支持类似于 C#的 using 语句。C++/CLI 针对这个问题提供了更好的方法。如果引用类型在本地声明，编译器就创建一条 try/finally 语句，在该块的最后调用 Dispose()方法。F#提供了两种不同的结构，轻松地支持资源管理。在值超出作用域时，use 绑定会自动调用 Dispose()方法。using 表达式创建的对象必须释放，并且在所调用函数的末尾释放。下面的 bar()函数在创建 Resource 对象后调用：

```
// C#
using (Resource r = new Resource())
{
   r.Foo();
}

// C++/CLI
{
   Resource r;
   r.Foo();
}

' Visual Basic
Using r As New Resource
   r.Foo()
End Using

// F# use binding
let rs =
   use r = new Resource()
   r.Foo()
   // r.Dispose called implicitly
// F# using expression
let bar (r : Resource) =
   r.Foo()

using (new Resource()) bar
```

53.10.3 重写 Finalize()方法

如果类包含必须释放的本地资源，该类就必须重写 Object 类中的 Finalize()方法。在 C#中，这只需编写一个析构函数。C++/CLI 采用一种特殊的语法：用“!”前缀定义终结器。在终结器中，也不允许释放有终结器的对象，以确保终结的顺序。这就是 Dispose 模式定义一个带布尔参数的 Dispose()方法的原因。在 C++/CLI 中，不需要在代码中实现这种模式，因为这由编译器完成。C++/CLI 析构函

数实现两个 Dispose()方法。在 Visual Basic 中，Dispose()方法和终结器都必须手工实现。但是大多数 Visual Basic 类都不直接使用本地资源，而是借助于其他类。在 Visual Basic 中，通常不需要重写 Finalize()方法，但一般需要实现 Dispose()方法。

在 C#中，编写析构函数重写基类的 Finalize()方法。C++/CLI 的析构函数实现 IDisposable 接口。


```
// C#
public class Resource: IDisposable
{
   ~Resource // override Finalize
   {
      Dispose(false);
   }
   protected virtual void Dispose(bool disposing)
   {
      if (disposing) // dispose embedded members
      {
      }
      // release resources of this class
      GC.SuppressFinalize(this);
   }
   public void Dispose()
   {
      Dispose(true);
   }
}
```

代码段 CSharp/Resource.cs


```
// C++/CLI
public ref class Resource
{
public:
   ~Resource() // implement IDisposable
   {
      this->!Resource();
   }
   !Resource() // override Finalize
   {
      // release resource
   }
};
```

代码段 CPPCLI/Resource.h


```
' Visual Basic
Public Class Resource
   Implements IDisposable
   Public Sub Dispose() Implements IDisposable.Dispose
      Dispose(True)
      GC.SuppressFinalize(Me)
   End Sub
```

```
    Protected Overridable Sub Dispose(ByVal disposing)
        If disposing Then
            ' Release embedded resources
        End If
        ' Release resources of this class
    End Sub
    Protected Overrides Sub Finalize()
        Try
            Dispose(False)
        Finally
            MyBase.Finalize()
        End Try
    End Sub
End Class
```

代码段 VisualBasic/Resource.vb

53.11 委托

第 8 章讨论的委托是方法类型安全的指针。在这 4 种语言中，都可以使用 delegate 关键字定义委托，但使用委托的方式有区别。

示例代码使用的 Demo 类有一个静态方法 Foo()和一个实例方法 Bar()。这两个方法都由 DemoDelegate 类型的委托实例调用。把 DemoDelegate 声明为调用一个返回类型为 void 并且带 int 参数的方法。

C#使用委托支持委托推断，其中编译器会创建一个委托实例，传递方法的地址。

在 C#和 C++/CLI 中，可以使用“+”运算符将两个委托合二为一：


```
// C#
public delegate void DemoDelegate(int x);
public class Demo
{
    public static void Foo(int x) { }
    public void Bar(int x) { }
}
    Demo d = new Demo();
    DemoDelegate d1 = Demo.Foo;
    DemoDelegate d2 = d.Bar;
    DemoDelegate d3 = d1 + d2;
    d3(11);
```

代码段 CSharp/DelegateSample.cs

C++/CLI 不支持委托推断。C++/CLI 需要创建一个委托类型的新实例，并将方法的地址传递给构造函数。


```
// C++/CLI
public delegate void DemoDelegate(int x);
public ref class Demo
{
public:
    static void Foo(int x) { }
    void Bar(int x) { }
```

```
};
Demo^ d = gcnew Demo();
DemoDelegate^ d1 = gcnew DemoDelegate( &Demo::Foo);
DemoDelegate^ d2 = gcnew DemoDelegate(d, &Demo::Bar);
DemoDelegate^ d3 = d1 + d2;
d3(11);
```

代码段 CPPCLI/DelegateSample.h

与 C++/CLI 类似，Visual Basic 也不支持委托推断。必须创建一个委托类型的新实例，传递方法的地址。Visual Basic 使用 AddressOf 运算符传递方法的地址。

因为 Visual Basic 没有为委托重载“+”操作符，所以需要调用 Delegate 类的 Combine()方法。把 Delegate 类写在方括号中，因为 Delegate 是一个 Visual Basic 关键字，不能使用同名的类。给 Delegate 加上方括号，可确保使用 Delegate 类，而不使用 Delegate 关键字。


```
' Visual Basic
Public Delegate Sub DemoDelegate(ByVal x As Integer)
Public Class Demo
   Public Shared Sub Foo(ByVal x As Integer)
     '
   End Sub
   Public Sub Bar(ByVal x As Integer)
     '
   End Sub
End Class
Dim d As New Demo()
Dim d1 As New DemoDelegate(AddressOf Demo.Foo)
Dim d2 As New DemoDelegate(AddressOf d.Bar)
Dim d3 As DemoDelegate = [Delegate].Combine(d1, d2)
d3(11)
```

代码段 VisualBasic/DelegateDemo.vb

在 F#中，作为对象的函数是头等成员。与其他.NET 语言的交互操作性一样，由于 F#也需要委托，因此语法看起来比较复杂。在 F#中声明委托时，需要把委托关键字赋予类型名，并定义参数和返回类型。对于 DemoDelegate()方法，参数的类型是 int，返回类型是与 C#中的 void 类似的 unit。

用一个静态成员 Foo()和一个实例成员 Bar()定义 Demo 类型。这两个成员都满足委托类型的要求。

变量 d1 是引用 Demo.Foo()方法的委托变量，d2 引用实例方法 Bar()。InvokeDelegate()函数的参数是 DemoDelegate 和 int，它声明为通过委托调用引用的函数。它调用 Delegate 类的 Invoke()方法，并传递 int 参数。在声明 invokeDelegate()函数后，就传递委托实例 d1 和值 33，调用该函数。为了合并委托，需要调用 Delegate.Combine()方法。因为 Combine()方法需要两个 Delegate 参数，并返回一个 Delegate 类型，所以需要强制向上转换和强制向下转换。这里 F#的语法是使用“:>”把 d1 强制转换为基类型 Delegate，使用“:? >”把 Delegate 类型 d33 强制转换为派生类 DemoDelegate。除了使用“:>”和“:?>”之外，还可以使用 upcast 和 downcast 关键字。


```
// F#
type DemoDelegate = delegate of int -> unit

type Demo() as this =
   static member Foo(x : int) =
```

```
        printfn "Foo %d" x
    member this.Bar(x : int) =
        printfn "Bar %d" x

// F# using delegate
let d = Demo()
let d1 : DemoDelegate = new DemoDelegate(Demo.Foo)
let invokeDelegate (dlg : DemoDelegate) (x : int) =
    dlg.Invoke(x)
(invokeDelegate d1 33)
let d2 : DemoDelegate = new DemoDelegate(d.Bar)
let d11 = d1 :> Delegate
let d22 = d2 :> Delegate
let d33 : Delegate = Delegate.Combine(d11, d22)
let d3 : d33 :?> DemoDelegate
(invokeDelegate d3 11)
```

代码段 FSharp/DelegateDemo.fs

53.12 事件

使用 event 关键字可以实现基于委托的订阅机制。这 4 种语言都定义了 event 关键字，提供类中的事件。下面的 EventDemo 类引发 DemoDelegate 类型的 DemoEvent 事件。

在 C#中，引发事件的语法类似于事件的方法调用。只要没有人注册事件，事件变量就是 null，所以在引发事件之前要检查事件变量是否为 null。在注册处理程序方法时，要使用“+=”运算符，在委托推断的帮助下传递处理程序方法的地址。


可从
wrox.com
下载源代码

```
// C#
    public class EventDemo
    {
        public event DemoDelegate DemoEvent;
        public void FireEvent()
        {
            if (DemoEvent != null)
                DemoEvent(44);
        }
    }
    public class Subscriber
    {
        public void Handler(int x)
        {
        // handler implementation
        }
    }
//...
EventDemo evd = new EventDemo();
Subscriber subscr = new Subscriber();
evd.DemoEvent += subscr.Handler;
evd.FireEvent();
```

代码段 CSharp/EventDemo.cs

C++/CLI 与 C#非常类似，除了触发事件不需要先查看事件变量不为 null。这由编译器创建的 IL

代码自动完成。

C#和 C++/CLI 使用“+=”运算符来注销事件。


```
// C++/CLI
   public ref class EventDemo
   {
   public:
      event DemoDelegate^ DemoEvent;
      public void FireEvent()
      {
         DemoEvent(44);
      }
   }
   public class Subscriber
   {
   public:
      void Handler(int x)
      {
         // handler implementation
      }
   }
//...
EventDemo^ evd = gcnew EventDemo();
Subscriber^ subscr = gcnew Subscriber();
evd-> DemoEvent += gcnew DemoDelegate(subscr, &Subscriber::Handler);
evd-> FireEvent();
```

代码段 CPPCLI/EventDemo.h

Visual Basic 使用不同的语法。用 Event 关键字声明事件，这与 C#和 C++/CLI 相同。但是，用 RaiseEvent 语句引发事件。RaiseEvent 语句检查事件变量是否用订阅器初始化。用于注册处理程序的 Addhandler 语句与 C#中的“+=”运算符有相同的功能。Addhandler 语句需要两个参数：第一个参数定义事件，第二个参数定义处理程序的地址。RemoveHandler 语句用于从事件中注销处理程序。


```
' Visual Basic
   Public Class EventDemo
   Public Event DemoEvent As DemoDelegate
   public Sub FireEvent()
      RaiseEvent DemoEvent(44);
   End Sub
End Class
Public Class Subscriber
   Public Sub Handler(ByVal x As Integer)
      ' handler implementation
      End Sub
   End Class
'...
Dim evd As New EventDemo()
Dim subscr As New Subscriber()
AddHandler evd.DemoEvent, AddressOf subscr.Handler
evd.FireEvent()
```

代码段 VisualBasic/EventDemo.vb

Visual Basic 提供的另一种语法对于其他语言不可用：对订阅事件的方法使用 Handles 关键字。它要求用 WithEvents 关键字定义一个变量：

```
Public Class Subscriber
   Public WithEvents evd As EventDemo
   Public Sub Handler(ByVal x As Integer) Handles evd.DemoEvent
      ' Handler implementation
   End Sub
   Public Sub Action()
      evd = New EventDemo()
      evd.FireEvent()
   End Sub
End Class
```

53.13 泛型

这 4 种语言都支持泛型的创建和使用。泛型参见第 5 章。

为了使用泛型，C#借用了 C++模板的语法，用尖括号定义泛型类型。C++/CLI 使用相同的语法。在 Visual Basic 中，泛型类型用圆括号中的 Of 关键字定义。F#非常类似于 C#。


```
// C#
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
intList.Add(3);
```

代码段 CSharp/GenericsDemo.cs


```
// C++/CLI
List<int>^ intList = gcnew List<int>();
intList->Add(1);
intList->Add(2);
intList->Add(3);
```

代码段 CPPCLI/GenericsDemo.h


```
' Visual Basic
Dim intList As List(Of Integer) = New List(Of Integer)()
intList.Add(1)
intList.Add(2)
intList.Add(3)
```

代码段 VisualBasic/GenericsDemo.vb


```
// F#
let intList =
new List<int>()
intList.Add(1)
intList.Add(2)
intList.Add(3)
```

代码段 FSharp/GenericsDemo.fs

因为类声明使用尖括号，所以编译器知道要创建一个泛型类型。用 where 子句定义约束。


```
public class MyGeneric<T>
   where T: IComparable<T>
{
   private List<T>list = new List<T>();
   public void Add(T item)
   {
      list.Add(item);
   }
   public void Sort()
   {
      list.Sort();
   }
}
```

代码段 CSharp/GenericsDemo.cs

用 C++/CLI 定义泛型类型与用 C++定义模板类似。但 C++/CLI 不使用 template 关键字，而对于泛型使用 generic 关键字。where 子句类似于 C#中的 where 子句，但 C++/CLI 没有构造函数的限制。


```
generic <typename T>
where T: IComparable<T>
ref class MyGeneric
{
private:
   List<T>^ list;
public:
   MyGeneric()
   {
      list = gcnew List<T>();
   }
   void Add(T item)
   {
      list-> Add(item);
   }
   void Sort()
   {
      list-> Sort();
   }
};
```

代码段 CPPCLI/GenericsDemo.h

Visual Basic 用 Of 关键字定义泛型类，用 As 定义约束。


```
Public Class MyGeneric(Of T As IComparable(Of T))
   Private myList = New List(Of T)
   Public Sub Add(ByVal item As T)
      myList.Add(item)
   End Sub
   Public Sub Sort()
      myList.Sort()
   End Sub
End Class
```

代码段 VisualBasic/GenericsDemo.vb

F#用尖括号定义泛型类型。泛型类型用一个撇号标记。约束用 when 关键字定义。约束跟在 when 的后面。在下面的示例中，使用“:>”类型约束。它指定，泛型类型必须派生自接口或基类。也可以定义值类型约束(: struct)、引用类型约束(: not struct)和默认构造函数约束('a : (new : unit -> ' a))。不可用于其他语言的约束是空约束('a : null)。这个约束指定' a 泛型类型必须是可空的，这就允许使用所有.NET 引用类型，但不能使用不能为空的 F#类型：


```
type MyGeneric<'a> when 'a :> IComparable<'a>() as this =
   let list = new List<'a>()
   member this.Add(item : 'a) = list.Add(item)
   member this.Sort() = list.Sort()
```

代码段 FSharp/GenericsDemo.fs

53.14 LINQ 查询

语言集成的查询是 C#和 Visual Basic 的一个功能。这两种语言的语法非常类似：

LINQ 参见第 11 章。


```
// C#
var query = from r in racers
            where r.Country == "Brazil"
            orderby r.Wins descending
            select r;
```

代码段 CSharp/LinqSample.cs


```
' Visual Basic
Dim query = From r in racers _
            Where r.Country = "Brazil" _
            Order By r.Wins Descending _
            Select r
```

代码段 VisualBasic/LinqSample.vb

C++/CLI 不支持 LINQ 查询。

53.15 C++/CLI 混合本地代码和托管代码

C++/CLI 的一大优点是混合本地代码和托管代码。使用 C#中的本地代码需要通过一种机制(称为平台调用)。平台调用参见第 26 章。在 C++/CLI 中使用本地代码切实可行。

在托管类中，可以使用本地代码和托管代码，如下所示。本地类也是如此。可以在一个方法中混合使用本地代码和托管代码。


```
#pragma once
#include <iostream> // include this header file for cout
using namespace std;  // the iostream header defines the namespace std
using namespace System;
public ref class Managed
{
public:
   void MixNativeAndManaged()
   {
      cout << "Native Code" << endl;
      Console::WriteLine("Managed Code");
   }
};
```

代码段 CPPCLI/Mixing.h

在托管类中，还可以声明本地类型的字段或本地类型的指针。但不能在本地类中声明托管类型的字段或托管内型的指针。必须注意，可以在垃圾收集器清理内存时删除托管类型的实例。

为了将托管类用作本地类中的成员，C++/CLI 定义了 gcroot 关键字，它在头文件 gcroot.h 中定义。gcroot 关键字包装一个 CGHandle，CGHandle 用于跟踪本地引用中的 CLR 对象。

```
#pragma once
#include "gcroot.h"
using namespace System;
public ref class Managed
{
public:
   Managed() { }
   void Foo()
   {
      Console::WriteLine("Foo");
   }
};
public class Native
{
private:
   gcroot<Managed^> m_p;
public:
   Native()
   {
      m_p = gcnew Managed();
   }
   void Foo()
   {
      m_p -> Foo();
   }
};
```

53.16　C#的特殊功能

一些 C#语法功能没有在本章中介绍。C#定义了 yield 语句，它便于创建枚举器。这条语句对于 C++/CLI 和 Visual Basic 不可用。在这些语言中，必须手动实现枚举器。另外，C#还为可空类型定

义了特殊的语法，而在其他语言中，必须使用泛型结构 Nullable<T>。

C#允许使用不安全的代码块，在不安全的代码块中可以使用指针和指针算术。这个功能非常有利于调用本地库中的方法。Visual Basic 没有这个功能，这是 C#的一个优势。C++/CLI 不需要使用 unsafe 关键字定义不安全的代码块，C++/CLI 本身就混合了本地代码和托管代码。

53.17 小结

本章学习了如何将 C#的语法映射到 Visual Basic、C++/CLI 和 F#上。C++/CLI 定义了对 C++的扩展，以编写.NET 应用程序，并利用 C#进行语法扩展。尽管 C#和 C++/CLI 的根源相同，但有许多重要的区别。Visual Basic 没有使用花括号，但比较琐碎。F#的语法完全不同，因为它主要关注函数编程。

在语法映射上，本章探讨了如何把 C#语法映射到 Visual Basic、C++/CLI 和 F#上，介绍了其他 3 种语言的语法，如何定义类型、方法、属性，哪些关键字用于 OO 功能，资源管理如何进行，委托、事件和泛型在 4 种语言中如何实现。

尽管可以映射大多数语法，但这些语言的功能仍有区别。