Visual Studio.NET 7.0 丛书

C#语言使用手册

武 装 等编著

国防工业出版社

·北京·

图书在版编目(CIP)数据

C#语言使用手册/武装等编著.—北京:国防工业出版社,2001.6
(Microsoft Visual Studio.NET 7.0 丛书)
ISBN 7-118-02526-7

Ⅰ.V… Ⅱ.武… Ⅲ.C#语言-程序设计
Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2001)第 20331 号

国防工业出版社出版发行
(北京市海淀区紫竹院南路 23 号)
(邮政编码 100044)
北京奥隆印刷厂印刷
新华书店经售
*
开本 787×1092 1/16 印张 15¾ 359 千字
2001 年 6 月第 1 版 2001 年 6 月北京第 1 次印刷
印数:1-4000 册 定价:20.00 元

前　言

Visual Studio.NET 7.0 为微软 VS 系列的最新版本，对 Web 开发部分进行了全新的改进，而.NET 框架正是其中最具代表性的创新。因版本延续与称呼方便之故，Visual Studio.NET 7.0 也被称为 Visual Studio 7.0。.NET 框架是一种用于构建、配置和运行 Web 服务和应用程序的多语言环境，并将成为未来 Windows 环境下网络开发的主要工具。

C#是 Visual Studio 7.0 中引入的一种新型编程语言，它是由 C 和 C++语言发展而来的，具有更简洁、更先进、类型安全以及面向对象等特点。开发 C#语言的初衷，就是为了创建能够运行于.NET 框架平台上的，具有更广泛应用范围的商用程序。与 Visual C++和 Visual Basic 不同，C#是一种全新的语言，既没有向上兼容的问题，也没有以前版本语言的限制，因而能够充分利用.NET 框架所提供的优势。

C#具有简单、先进、面向对象和类型安全等特点。C#的目标是将 Visual Basic 和 C++的优势整合在一起。C#能够得到通用开发语言的服务支持，例如语言互用、冗码剔除、安全性提高以及改进的版本支持等。综合考虑 C#的种种优势，可以肯定的是它将取代 C 和 C++而成为面向对象开发的主流工具。

本书并非简单地罗列知识点，而是以实例讲解贯穿始终，向读者全面介绍 C#的语言知识点和编程技巧。这使得读者能够掌握并灵活运用这些知识点，迅速掌握 C#这门新兴的编程语言。书中不但详细地介绍了 C#语言使用，而且通过正反实例深入剖析了编译器对具体语法的处理过程。这样使得读者不是流于表面地学会 C#语言的基本用法，而且真正掌握这门语言的精髓。

本书除封面署名作者外，沈冰、汤春明、许颖、周一兵、黄剑波、黄君玲、吴强、李义、于佳音、季洪飞、沈鹏、刘树声、薛文涛、林茵茵、王江辉、胡建明、龚雪梅、廖晓筠、赵立峰、李国梁、陈民生、付强等都为本书的出版付出了不同程度的劳动，在此一并表示感谢。由于时间所限，书中错误和疏漏之处在所难免，敬请指正。

编　者

第 1 章　C#语言概述

C#是一种简单而先进的编程语言，它派生自 C 和 C++，具有面向对象和类型安全（type-safe）等特点。C#的目标是综合 Visual Basic 和 C++的优势。这一点在使用 Visual Studio 7.0 开发 C# Windows 程序时，非常直观地体现出来：开发界面完全类似于 Visual Basic；而语言特点却与 C++一脉相承。

1.1　C#程序分析

本书中的 C#程序都是使用 Visual Studio 7.0 编写的。Visual Studio 7.0 的用户界面比起以前版本有了很大改动。这里首先简述使用 Visual Studio 7.0 编写 C#程序的步骤。

(1) 启动 Visual Studio 7.0，此时将出现如图 1-1 所示的启动页。

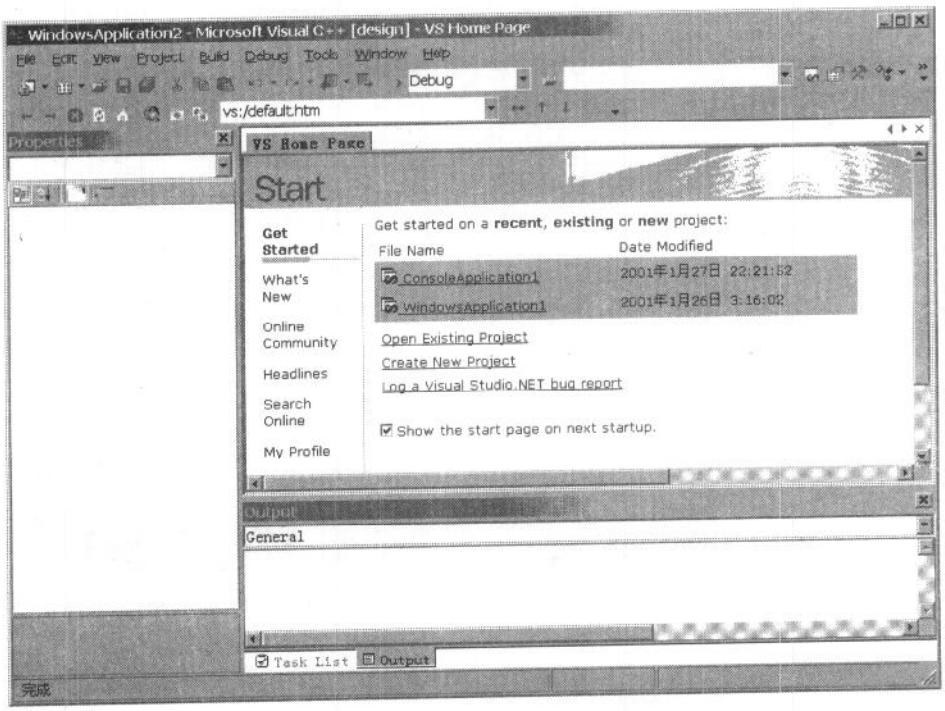

图 1-1　Visual Studio 7.0 的启动页

(2) 点击启动页中的 Create New Project，或选择 File|New|Project 菜单命令，此时

出现如图 1-2 所示的 New Project 对话框。

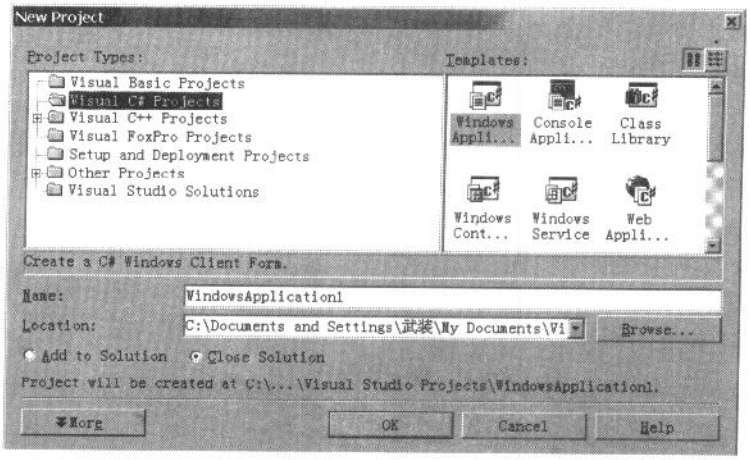

图 1-2 New Project 对话框

(3) 在 Project Type 列表框中选择 Visual C# Projects，此时右侧的 Templates 列表框中出现项目模板，如图 1-2 所示。

(4) 由于本书的主要目的是向读者介绍 C#语言的编程方法，因此选择 Console Application 模板，以建立控制台应用程序。然后将 Name 文本框中的默认项目名改为 Example，并按下 OK 按钮。此时将出现如图 1-3 所示的源代码编辑窗口。C#程序的默认文件扩展名为.cs，因此此程序的源文件被保存为 Example.cs。

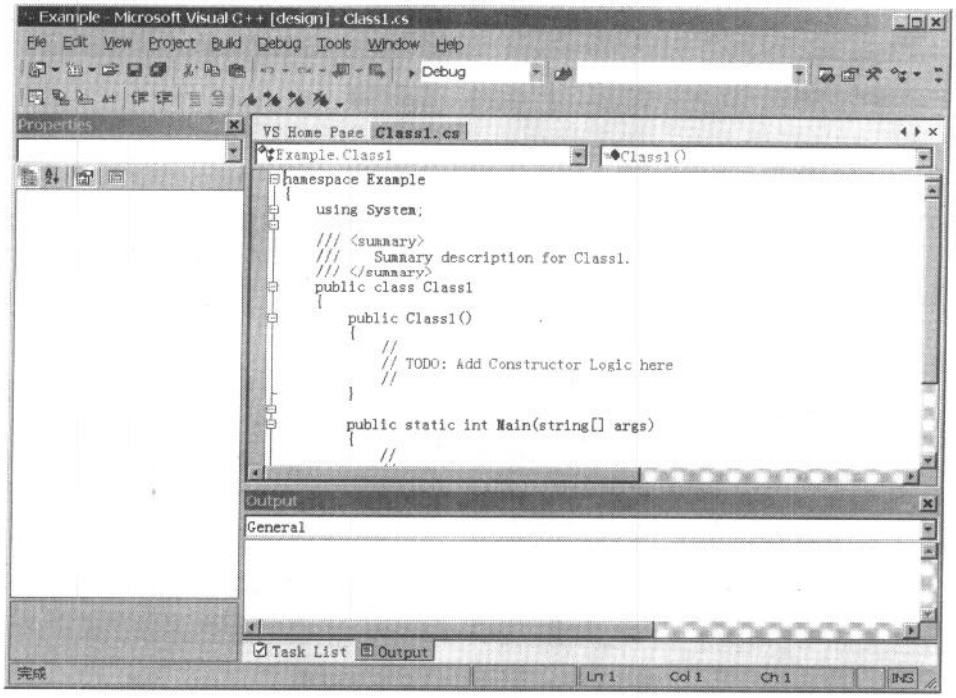

图 1-3 源代码编辑窗口

C#程序由多个源文件构成。源文件就是一个符合特定规则的 Unicode 字符序列。通常，源文件与文件系统中的文件具有一一对应性，但是 C#并不需要这种对应性。

由图 1-3 可以看到 Visual Studio 7.0 默认生成的代码，其中包括：名称空间（namespace）Example、using System 指令、类 Class1 和 Main 方法。程序中“///”后为 XML 代码，C#是 Visual Studio 7.0 中唯一具有此能力的编程语言。

名称空间是一种组织类库元素的分层方法，在 C#中被广泛采用。在 C#程序中，有时需要指定名称空间或类型名。这两种名称都由一个或多个标识符组成，不同标识符间以“.”作为分隔符。using 指令允许不通过限定名就能使用指定名称空间中的成员。Main 方法是 C#程序的入口点，是程序控制起始和终止的位置。也就是说，只有包含 Main 方法的程序才能在编译后被运行。

C#程序和 C++的另一不同在于：类、结构、接口的定义和实现在一起，而不是像后者那样是分开的。

“Hello, world!”是一个经典的编程入门示例。这个例子用 C#编写如下：

```
class Example
{
    static void Main()
    {
        Console.WriteLine("Hello, world!");
    }
}
```

其中 Console.WriteLine 为 System.Console.WriteLine 的简写。System 为名称空间，Console 为该名称空间中定义的一个类，而 WriteLine 为 Console 类中定义的一个静态方法，用以向控制台输出指定内容。

按下 F5 可以编译项目，得到可执行文件 Example.exe。运行该文件将输出：

```
Hello, world!
```

“Hello, world!”的输出是通过 CLS 类库（System 名称空间为其根层次）生成的。C#本身并没有提供类库，而是使用通用类库。Visual Basic 和 Visual C++也能够使用该类库。此外，还可以使用控制台指令 csc 编译 C#程序。

对于 C 和 C++开发者来说，需要说明其他一些问题：

• 程序既不使用::也不使用->运算符。::不是 C#中的运算符，而->也只在一小部分 C#程序中使用。C#程序在限定名中使用 . 作为分隔符，例如 Console.WriteLine。

• 程序不包含提前声明（forward declaration）。在 C#程序中不需要提前声明，并且声明次序也不重要。

• 程序并不使用#include 来导入程序文本（program text）。程序间的依赖性由符号而不是程序文本处理。这个系统消除了由不同语言编写的程序间的壁垒。例如

Console 类可以由 C#编写，也可以由其他语言编写。

在本书以后的内容中，将向读者逐步介绍 C#中各个语言要素。

1.2 C#的新特性

C#能够访问下一代 Windows 服务（NGWS，Next Generation Windows Services）平台，例如通用运行引擎和丰富的类库。NGWS 软件开发工具包（SDK，Software Development Kit）定义了一个“通用语言子集（CLS，Common Language Subset）”。CLS 能够保证 CLS 兼容语言与类库间的无缝互用，因而类似于一种世界语。这意味着虽然 C#是一种全新的语言，但它依然与其他历史悠久的开发工具（例如 Visual Basic、Visual C++）一样，能使用同样丰富且功能强大的类库。值得一提的是，C#本身并不包括类库。

C#在语句、表达式和运算符方面使用了许多 C++的特性；而在类型安全、版本兼容、事件和冗码收集方面引入了很多改进。C#提供了对通用 API 的访问（.NET、COM、自动化和 C 风格的 API）。它还支持非安全代码，以便不在冗码收集器的控制下使用指针操作内存。

1.2.1 自动内存管理

手动内存管理（manual memory management）需要开发者分配和释放内存块。手动内存管理既浪费时间，也比较困难。C#中提供的自动内存管理，将开发者从这个繁重的工作中解脱出来。在大多数情况下，自动内存管理能够提高代码质量和开发效率，而不会为程序性能带来负面影响。检查如下示例程序：

```
using System;

public class Stack
{
    private Node first = null;
    public bool Empty
     {
        get
         {
            return (first == null);
        }
    }
    public object Pop()
    {
        if (first == null)
```

```
                throw new Exception("不能弹出空堆栈!");
            else
            {
                object temp = first.Value;
                first = first.Next;
                return temp;
            }
        }
        public void Push(object o)
        {
            first = new Node(o, first);
        }
        class Node
        {
            public Node Next;
            public object Value;
            public Node(object value): this(value, null) {}
            public Node(object value, Node next)
            {
                Next = next;
                Value = value;
            }
        }
    }
```

此段代码中定义的 Stack 类，由 Node 实例链表实现，而 Node 实例则由 Push 方法创建。当某个 Node 实例不会再被其他代码访问时，它就可以被冗码收集器（garbage collector）收集了。例如，当从 Stack 中删除某个条目时，相关的 Node 实例就适合于被冗码收集了。下面给出 Stack 类的使用示例：

```
class Example
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 0; i < 10; i++)
            s.Push(i);
        while (!s.Empty)
            Console.WriteLine(s.Pop());
    }
```

```
}
```

上述代码创建了一个 Stack 实例，并向其中加入 10 个元素（0～9）。接着将 Stack 实例中的元素一一弹出，并调用 Console.WriteLine 输出到控制台。当所有元素都被弹出堆栈后，s 被赋值为 null。此时 Stack 实例和相关的 10 个 Node 实例就适合被冗码收集了。

一般来说，自动内存管理已经足够，但有时还需更精细的控制。C#能够编写“不安全”代码，这种代码可以直接处理指针类型，并固定对象以临时阻止冗码收集器移动它们。不安全代码必须使用 unsafe 明确标记，这样开发者就不会偶然用到不安全特性。C#编译器和运行引擎共同确保了不安全代码不能被作为安全代码使用。

1.2.2 统一类型系统

C#中提供了“统一类型系统”（unified type system），因此包括数组类型在内的所有类型，都可以被作为对象处理。从概念上来说，所有类型都派生自 object，这意味着任何值（甚至基本类型值，例如 int）都可以调用 object 中的方法。示范代码中对 int 类型的常数，调用了 Object.ToString 方法：

```
using System;
class Example
{
    static void Main()
    {
        Console.WriteLine(3.ToString());
    }
}
```

将数值类型变量转换为引用类型时，需要首先分配保存数值的对象包（box），然后将数值拷贝到包中。解包（将对象包转换为其原始数组类型）过程时，数值将被从包中拷贝到合适的存储位置。例如可将 int 值转换为 object，然后再将其转换回 int。示范代码给出了打包和解包的过程：

```
class Example
{
    static void Main()
    {
        int i = 123;
        object o = i;    // 打包
        int j = (int) o;  // 解包
    }
```

```
}
```

统一类型系统为数值类型提供了对象性质，而且不会导致附加开销。对于不需要将 int 作为对象处理的程序来说，int 值只是一个 32 位值。而对于需要将 int 作为对象处理的程序来说，这种转换功能即时可用。因此，统一类型系统在数值类型和引用类型之间架起了桥梁。例如，NGWS 类库中的 Hashtable 类定义如下：

```
public class Hashtable
{
    public void Add(object Key, object Value) {...}
    ...
}
```

由于 C#具有统一类型系统，任何类型对象都可以作为 Hashtable.Add 的 Key 和 Value 参数。

1.2.3 版本兼容

大多数语言都会遇到版本兼容问题，而 C#则不会。版本兼容实际有两个含义：如果基于先前版本的代码，能在重新编译时与新版本一起工作，则组件的新版本为与先前版本“源代码兼容”；如果基于先前版本的程序，无需重新编译就能与新版本一起工作，则组件的新版本与先前版本为“二进制兼容”。

大多数语言根本不支持二进制兼容性，而且多数语言也不怎么支持源代码兼容性。实际上，由于这些语言本身的缺陷，使得它们根本不可能支持版本兼容（也就是说，类的代码改变后，客户代码也必须改变）。

在如下示范代码中，Base 类的第一个版本不包含 F 方法，而其派生类 Dervied 中则引入了 F 方法：

```
// Author A
namespace A
{
    class Base // V 1.0
    {
    }
}
// Author B
namespace B
{
    class Derived: A.Base
    {
```

```
        public virtual void F()
        {
            System.Console.WriteLine("Derived.F");
        }
    }
}
```

Dervied 类被安装在很多客户和服务器中，并且一切都很正常。然而，Base 类的新版本出现了，并且其中实现了新的 F 方法：

```
// Author A
namespace A
{
    class Base     // V 2.0
    {
        public virtual void F()
        {
            System.Console.WriteLine("Base.F");
        }
    }
}
```

这时就会遇到版本兼容问题。Base 的新版本应该同时具有源代码兼容性和二进制兼容性。不幸的是，Base 中的 F 方法使得 Derived 中的 F 出现问题。Derived.F 是否重载了 Base.F？这实际是不可能的，因为 Derived 出现时，Base 中还没有实现 F 呢。如果 Derived.F 不重载 Base.F，那末 Derived.F 是否能遵守 Base 中指定的规则呢？这就更不可能了，因为 Derived.F 显然不可能遵守在其创建时尚不存在的规则。例如，只有重载时调用 Base.F 才能保证符合规则。

此类名称冲突是否会真的发生？首先，很多开发者是独立工作的（可能分属于不同的公司），这样就不大有可能相互合作。其次，派生组件使得名称冲突的可能性大大增强。

在 C#中，通过要求开发者明确其意图，来解决版本兼容问题。在原来的代码示例中， Derived.F 显然是一个新方法而不是重载自基类方法。而对于新版本的 Base 来说，Dervied 的二进制版本依然是清楚的——Dervied.F 与 Base.F 无联系，因此不应被作为重载函数。

然而，当 Derived 被重新编译时，C#的编译器将产生一个警告，并默认 Dervied.F 隐藏 Base.F。此时，可以通过关键字 new 明确表明两者毫无关系，而消除此警告：

```
// Author A
```

```
namespace A
{
    class Base                          // V 2.0
    {
        public virtual void F()
        {// 添加到 V 2.0 中
            System.Console.WriteLine("Base.F");
        }
    }
}
// Author B
namespace B
{
    class Derived: A.Base    // V 2.0a，非重载
    {
        new public virtual void F()
        {
            System.Console.WriteLine("Derived.F");
        }
    }
}
```

而另一方面，Derived 的作者可能进一步检查，并决定 Derived 的 F 应该重载 Base 的 F。此时，应通过关键字明确指定重载，如下所示：

```
// Author A
namespace A
{
    class Base          // V 2.0
    {
        public virtual void F()
        {// 添加到 V2.0 中
            System.Console.WriteLine("Base.F");
        }
    }
}
// Author B
namespace B
{
```

```
    class Derived: A.Base    // V 2.0b，重载
    {
        public override void F()
        {
            base.F();
            System.Console.WriteLine("Derived.F");
        }
    }
}
```

当然，也可以选择改变 F 的名称，以完全避免名称冲突。但这将破坏 Derived 的源代码和二进制兼容性。采用何种方式，应取决于具体情况。如果 Derived 不会用于其他程序，则应选择改变 F 名称，因为它将提高程序的可读性——不再有任何有关 F 含义的误解。

第 2 章　声明空间与名称空间

C#中的声明定义了程序的组成元素。C#程序是通过名称空间组织的，其中能够包含类型声明和嵌套名称空间声明。类型声明用于定义类、结构、接口、枚举和delegate。类型声明中允许包含的成员种类取决于其形式。例如，类声明能够包含实例构造函数、析构函数、静态构造函数、常量、字段、方法、属性、事件、分度器、运算符和嵌套类型。

2.1　声明空间

声明在其所属的声明空间（declaration space）中定义了一个名称。除重载构造函数、方法、分度器和运算符外，声明空间中不允许出现同名成员声明。而且名称空间中也不可能包含不同种类的同名成员。例如，声明空间中不会存在同名的字段和方法。

声明空间有如下几种：

• 在程序的所有源文件中，不带封闭型名称空间声明的成员声明被称为全局声明空间。

• 在程序的所有源文件中，具有相同全限定名的名称空间中的成员声明为单一组合声明空间。

• 每个类、结构或接口声明都创建了一个声明空间。名称通过类成员声明、结构成员声明或接口成员声明引入本声明空间。除了重载构造函数和静态构造函数外，类或结构中不能包含与其同名的其他成员。在类、结构和接口中可以声明重载方法和分度器；此外，在类和结构中还能声明重载构造函数和运算符。例如，类、结构或接口中可以包含多个同名方法声明（例如，彼此的参数或返回值不同）。基类（基接口）对类（接口）的名称空间没有贡献。因此，派生类（接口）允许声明与基类（基接口）中同名（非重载）的成员。此类成员被称为隐藏被继承成员。

• 每个枚举声明都创建了一个声明空间。名称通过枚举成员声明引入本声明空间。

• 每个块或 switch 块都为本地变量创建了一个单独的声明空间。名称通过本地变量声明引入本声明空间。如果块为构造函数或方法声明体，则形参列表中的参数声明为块的本地变量声明空间的成员。块的本地变量声明空间中包含其中的所有嵌套块。因此，嵌套块中的变量不能与其母块中的变量同名。

• 每个块或 switch 块都为标号创建了一个单独的声明空间。名称通过标记语句引入本声明空间，并且通过 goto 语句被引用。因此，嵌套块中的标号不能与其母块中的标号同名。

声明顺序一般来说并不重要，对于名称空间、类型、常量、方法、属性、时间、分度器、运算符、构造函数、析构函数和静态构造函数的声明和使用来说，尤其如此。不过在以下几种情况中，声明顺序是相当重要的：

- 字段和本地变量的声明顺序决定了其初始化函数（如果有的话）的执行顺序。
- 本地变量必须在使用前定义。
- 忽略常量表达式的值时，枚举成员的声明顺序也十分重要。

名称空间的声明空间是开放的（open ended），全限定名相同的名称空间贡献于同样的声明空间，例如：

```
namespace Family.Parent
{
    class Father
    {
        ...
    }
}
namespace Family.Parent
{
    class Mother
    {
        ...
    }
}
```

由于上述代码中声明的两个名称空间（Family.Parent.Father 和 Family.Parent.Mother）贡献于同样的声明空间，因此如果它们包含同名成员，就会导致错误产生。

块的声明空间包含所有嵌套块，因此，在下面的示范代码中，F 和 G 方法存在错误。这是由于在外层块中已经声明了 i，从而在内层块中不能再次声明该变量。作为对比，H 和 i 方法则完全正确，因为 i 声明在两个非嵌套块中。示范代码如下：

```
class A
{
    void F()
    {
        int i = 0;
        if (true)
        {
            int i = 1;
```

```
            }
        }
    void G()
    {
            if (true)
             {
                int i = 0;
            }
            int i = 1;
    }
    void H() {
            if (true)
             {
                int i = 0;
            }
            if (true)
             {
                int i = 1;
            }
        }
    void I() {
            for (int i = 0; i < 10; i++)
                H();
            for (int i = 0; i < 10; i++)
                H();
        }
    }
```

2.2 名称空间

C#程序是通过名称空间组织起来的。名称空间既可以作为程序的内部组织系统，也可以作为其外部组织系统（使其他程序能调用本程序元素）。通过 using 指令能够方便地使用名称空间。

2.2.1 名称空间声明

编译单元定义了源文件的总体结构，其中包含零个或多个 using 指令以及名称空间声明。一个 C#程序由一个或多个编译单元组成，其中每个都包含于一个源文件中。编译单元彼此依赖，在编译 C#程序时，它们将被一起处理。编译单元中的 using 指令会

影响该单元中的名称空间成员声明，不过其他编译单元不会受到影响。

每个程序编译单元的名称空间成员声明，都贡献于同一个名称空间——全局名称空间（global namespace）。假设程序中包含两个文件：A.cs 和 B.cs，其中分别声明了类 A 和类 B，其全限定名分别为 A 和 B。由于 A.cs 和 B.cs 属于同一全局名称空间，因此，如果在 A.cs 中也声明了与 B.cs 中同名的成员，例如类 B，那么将出现错误。

名称空间由关键字 namespace 标识，它用于声明一个作用域。在名称空间的作用域中可以组织代码和创建全局唯一的类型。名称空间的声明格式如下（方括号标识可选项）：

```
namespace name[.name1[...]...]
{
type-declaration
...
}
```

其中，name、name1、…为名称空间名，可以为任意合法的标识符，其中可以包含“.”；type-declaration 为名称空间中的类型声明，它们可以为其他名称空间、类、接口、结构、枚举或 delegate。名称空间声明中不能包含任意访问修饰符，因为它已经被隐式限定为公共型（public）。名称空间声明中大括号中的内容也被称为名称空间体（namespace-body）。

名称空间可以作为编译单元的顶层声明，也可以作为另一个名称空间中的嵌入声明。编译单元的顶层声明同时也是全局名称空间的成员；而嵌入声明则为其外层名称空间的成员。无论是哪种情况，名称空间在其容器（外层名称空间）中必须为唯一的。

2.2.2 全限定名（Fully qualified names）

全限定名是名称空间和类型的唯一标识。以下规则用以确定名称空间或类型 N 的全限定名：

- 如果 N 为全局名称空间的成员，则其全限定名为 N。
- 如果 N 是在另一名称空间 S（S 为该名称空间的全限定名）中声明的成员，则其全限定名为 S.N。

换句话说，N 的全限定名为标识符的完整层次（始自全局名称空间）路径。示例如下：

```
namespace A                    // 全限定名为 A
{
    class B                    // 全限定名为 A.B
    {
        class C {}             // 全限定名为 A.B.C
    }
```