Contents

介绍

2020/11/2 • • Edit Online

C#(读作"See Sharp")是一种简单易用的新式编程语言，不仅面向对象，还类型安全。C#具有 C 语言系列中的根，并将对 C、C++和 Java 程序员立即熟悉。C#由 ECMA 国际标准化为*ecma-334*标准，并按 iso/lec（*iso/iec 23270*标准）标准化。Microsoft 的C# .NET Framework 编译器是这两个标准的一致性实现。

C# 是一种面向对象的语言。不仅如此，C# 还进一步支持***面向组件的***编程。当代软件设计越来越依赖采用自描述的独立功能包形式的软件组件。此类组件的关键特征包括：为编程模型提供属性、方法和事件；包含提供组件声明性信息的特性；包含自己的文档。C#提供语言构造以直接支持这些概念，并C#使用一种非常自然的语言来创建和使用软件组件。

多项 C# 功能有助于构造可靠耐用的应用程序：***垃圾回收***自动回收未使用的对象占用的内存;***异常处理***提供了一种结构化的可扩展方法用于错误检测和恢复;而且，语言的***类型安全***设计使得不可能从未初始化的变量中进行读取，将数组索引在其边界之外，或者执行未检查的类型转换。

C# 采用***统一的类型系统***。所有 C# 类型（包括 `int` 和 `double` 等基元类型）均继承自一个根 `object` 类型。因此，所有类型共用一组通用运算，任何类型的值都可以一致地进行存储、传输和处理。此外，C# 还支持用户定义的引用类型和值类型，从而支持对象动态分配以及轻量级结构的内嵌式存储。

为了确保程序C#和库能够以兼容的方式在一段时间内演化，重点在于设计中C#的***版本控制***。许多编程语言很少关注这个问题，因此，当引入新版依赖库时，用这些语言编写的程序会出现更多不必要的中断现象。由版本C#控制注意事项直接影响的设计方面包括单独 `virtual` 的和 `override` 修饰符、方法重载决策的规则以及对显式接口成员声明的支持。

本章的其余部分介绍了该C#语言的重要功能。尽管后面的章节介绍了以详细的方式（有时也是数学方式）的规则和例外，但本章采用的是以完整性为代价来提高清晰度和简洁性。目的是为读者提供一篇有助于编写早期程序和阅读后续章节的语言。

Hello world

"Hello, World"程序历来都用于介绍编程语言。下面展示了此程序的 C# 代码：

```
using System;

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

C# 源文件的文件扩展名通常为 `.cs`。假设 "Hello, World" 程序存储在文件 `hello.cs` 中，则可以使用命令行通过 Microsoft C#编译器编译该程序

```
csc hello.cs
```

这会生成一个名为 `hello.exe` 的可执行程序集。此应用程序在运行时生成的输出为

```
Hello, World
```

“Hello, World”程序始于引用 `System` 命名空间的 `using` 指令。命名空间提供了一种用于组织 C# 程序和库的分层方法。命名空间包含类型和其他命名空间。例如，`System` 命名空间包含许多类型（如程序中引用的 `Console` 类）和其他许多命名空间（如 `IO` 和 `Collections`）。借助引用给定命名空间的 `using` 指令，可以非限定的方式使用作为相应命名空间成员的类型。由于使用 `using` 指令，因此程序可以使用 `Console.WriteLine` 作为 `System.Console.WriteLine` 的简写。

“Hello, World”程序声明的 `Hello` 类只有一个成员，即 `Main` 方法。方法 `Main` 是 `static` 通过修饰符声明的。实例方法可以使用关键字 `this` 引用特定的封闭对象实例，而静态方法则可以在不引用特定对象的情况下运行。按照约定，`Main` 静态方法是程序的入口点。

程序的输出是由 `System` 命名空间中 `Console` 类的 `WriteLine` 方法生成。此类由 .NET Framework 类库提供，默认情况下，由 Microsoft C#编译器自动引用。请注意C#，自身没有单独的运行时库。相反，.NET Framework 是的C#运行时库。

程序结构

C# 中的关键组织结构概念包括***程序***、***命名空间***、***类型***、***成员***和***程序集***。C# 程序由一个或多个源文件组成。程序声明类型，而类型则包含成员，并被整理到命名空间中。类型示例包括类和接口。成员示例包括字段、方法、属性和事件。编译完的 C# 程序实际上会打包到程序集中。程序集通常具有文件扩展名 `.exe` 或 `.dll`，具体取决于它们是否实现***应用程序***或*库*。

示例

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

在名 `Stack` `Acme.Collections` 为的命名空间中声明一个名为的类。此类的完全限定的名称为 `Acme.Collections.Stack`。此类包含多个成员：一个 `top` 字段、两个方法（`Push` 和 `Pop`）和一个 `Entry` 嵌套类。`Entry` 类还包含三个成员：一个 `next` 字段、一个 `data` 字段和一个构造函数。假定示例的源代码存储在 `acme.cs` 文件中，以下命令行

```
csc /t:library acme.cs
```

将示例编译成库(不含 `Main` 入口点的代码),并生成 `acme.dll` 程序集。

程序集包含*中间语言*(IL)指令形式的可执行代码和*元数据*形式的符号信息。执行前,程序集中的 IL 代码会被 .NET 公共语言运行时的实时 (JIT) 编译器自动转换成处理器专属代码。

由于程序集是包含代码和元数据的自描述功能单元,因此无需在 C# 中使用 `#include` 指令和头文件。只需在编译程序时引用特定的程序集,即可在 C# 程序中使用此程序集中包含的公共类型和成员。例如,此程序使用 `acme.dll` 程序集中的 `Acme.Collections.Stack` 类:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

如果程序 `test.cs` 存储在文件中,则编译时 `test.cs` , `acme.dll` 可以使用编译器的 `/r` 选项来引用程序集:

```
csc /r:acme.dll test.cs
```

这会创建 `test.exe` 可执行程序集,它将在运行时输出以下内容:

```
100
10
1
```

使用 C#,可以将程序的源文本存储在多个源文件中。编译多文件 C# 程序时,可以将所有源文件一起处理,并且源文件可以随意相互引用。从概念上讲,就像是所有源文件在处理前被集中到一个大文件中一样。在 C# 中,永远都不需要使用前向声明,因为声明顺序无关紧要(除了极少数的例外情况)。C# 并不限制源文件只能声明一种公共类型,也不要求源文件的文件名必须与其中声明的类型相匹配。

类型和变量

C# 有两种类型:*值类型*和*引用类型*。值类型的变量直接包含数据,而引用类型的变量则存储对数据(称为"对象")的引用。对于引用类型,两个变量可以引用同一对象;因此,对一个变量执行的运算可能会影响另一个变量引用的对象。借助值类型,每个变量都有自己的数据副本;因此,对一个变量执行的运算不会影响另一个变量(`ref` 和 `out` 参数变量除外)。

C#的值类型进一步划分为*简单类型*、*枚举类型*、*结构类型*和*可以为 null 的类型*, C#并且的引用类型进一步分为*类类型*、*接口类型*、*数组类型*和*委托类型*。

下表提供了C#的类型系统概述。

值类型	简单类型	有符号的整型：`sbyte`、`short`、`int`、`long`
		无符号的整型：`byte`、`ushort`、`uint`、`ulong`
		Unicode 字符：`char`
		IEEE 浮点：`float`、`double`
		高精度小数：`decimal`
		布尔：`bool`
	枚举类型	格式为 `enum E {...}` 的用户定义类型
	结构类型	格式为 `struct S {...}` 的用户定义类型
	可以为 null 的类型	值为 `null` 的其他所有值类型的扩展
引用类型	类类型	其他所有类型的最终基类：`object`
		Unicode 字符串：`string`
		格式为 `class C {...}` 的用户定义类型
	接口类型	格式为 `interface I {...}` 的用户定义类型
	数组类型	一维和多维，例如 `int[]` 和 `int[,]`
	委托类型	格式为的用户定义的类型，例如 `delegate int D(...)`

八个整型类型支持带符号或不带符号格式的 8 位、16 位、32 位和 64 位值。

这两个浮点类型（`float` 和 `double`）使用32位单精度和64位双精度 IEEE 754 格式表示。

`decimal` 类型是适用于财务和货币计算的 128 位数据类型。

C#的 `bool` 类型用于表示布尔值（`true` 或 `false` 的值）。

C# 使用 Unicode 编码处理字符和字符串。`char` 类型表示 UTF-16 代码单元，`string` 类型表示一系列 UTF-16 代码单元。

下表汇总了C#的数值类型。

𝚝𝚝	𝚝𝚝	𝚝𝚝	𝚝𝚝/𝚝𝚝
有符号整型	8	`sbyte`	-128…127
	16	`short`	-32768 … 32、767
	32	`int`	-2147483648 … 2, 147, 483, 647
	64	`long`	-9223372036854775808 … 9, 223, 372, 036, 854, 77 5, 807
无符号的整型	8	`byte`	0…255
	16	`ushort`	0 … 65, 535
	32	`uint`	0 … 4、294次、967、
	64	`ulong`	0 … 18、446、744、073、为 709、551、615
浮点	32	`float`	1.5 × 10 ^ −45到3.4 × 10 ^ 38, 7位精度
	64	`double`	5.0 × 10 ^ −324到1.7 × 10 ^ 308, 15位精度
Decimal	128	`decimal`	1.0 × 10 ^ −28到7.9 × 10 ^ 28, 28位精度

C# 程序使用***类型声明***创建新类型。类型声明指定新类型的名称和成员。C#类型的五个类别是用户定义的类别：类类型、结构类型、接口类型、枚举类型和委托类型。

类类型定义包含数据成员（字段）和函数成员（方法、属性等）的数据结构。类类型支持单一继承和多形性，即派生类可以扩展和专门针对基类的机制。

结构类型类似于类类型，因为它表示包含数据成员和函数成员的结构。但是，与类不同的是，结构是值类型，不需要进行堆分配。结构类型不支持用户指定的继承，并且所有结构类型均隐式继承自类型 `object`。

接口类型将协定定义为公共函数成员的命名集。实现接口的类或结构必须提供接口的函数成员的实现。接口可以从多个基接口继承，类或结构可以实现多个接口。

委托类型表示对具有特定参数列表和返回类型的方法的引用。通过委托，可以将方法视为可分配给变量并可作为参数传递的实体。委托类似于其他一些语言中的函数指针概念，但与函数指针不同的是，委托不仅面向对象，还类型安全。

类、结构、接口和委托类型都支持泛型，因此可以用其他类型参数化。

枚举类型是具有已命名常数的不同类型。每个枚举类型都有一个基础类型，该类型必须是八个整型类型之一。枚举类型的值集与基础类型的值集相同。

C# 支持任意类型的一维和多维数组。与上述类型不同，数组类型无需先声明即可使用。相反，数组类型是通过在类型名称后面添加方括号构造而成。`int[]` 例如，是的一维 `int` `int` 数组，`int[,]` 是的二维数组 `int[][]`，是的 `int` 一维数组，它是的一维数组。

可以为 null 的类型也无需声明即可使用。对于每个不可以为 null `T` 的值类型，都有 `T?` 一个对应的可以为 null 的 `null` 类型，该类型可以保存附加值。例如，`int?` 是一个可以容纳任何32位整数或值 `null` 的类型。

C#的类型系统是统一的，因此任何类型的值都可以被视为对象。每种 C# 类型都直接或间接地派生自 `object` 类类型，而 `object` 是所有类型的最终基类。只需将值视为类型 `object`，即可将引用类型的值视为对象。值类型的值通过执行***装箱***和***取消装箱***操作被视为对象。在以下示例中，`int` 值被转换成 `object`，然后又恢复成 `int`。

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;          // Boxing
        int j = (int)o;        // Unboxing
    }
}
```

当值类型的值转换为类型 `object` 时，将分配一个对象实例（也称为 "box"）来保存值，并将该值复制到该框中。相反，将 `object` 引用强制转换为值类型时，会进行检查以确定所引用的对象是否为正确的值类型的框，如果检查成功，则会将框中的值复制出来。

C#的统一类型系统实际上意味着值类型可以 "按需" 成为对象。鉴于这种统一性，使用类型 `object` 的常规用途库可以与引用类型和值类型结合使用。

C# 有多种*变量*，其中包括字段、数组元素、局部变量和参数。变量表示存储位置，每个变量都有一种类型，用于确定可以在变量中存储的值，如下表所示。

tttt	ttttt
不可以为 null 的值类型	具有精确类型的值
可以为 null 的值类型	空值或该精确类型的值
`object`	空引用、对任何引用类型的对象的引用或对任何值类型的装箱值的引用
类类型	空引用、对该类类型的实例的引用，或对派生自该类类型的类的实例的引用
接口类型	空引用、对实现接口类型的类类型的实例的引用，或对实现接口类型的值类型的装箱值的引用
数组类型	空引用、对该数组类型的实例的引用，或对兼容的数组类型实例的引用
委托类型	空引用或对该委托类型的实例的引用

表达式

表达式是在***操作数***和***运算符***的基础之上构造而成。表达式的运算符指明了向操作数应用的运算。运算符的示例包括 `+`、`-`、`*`、`/` 和 `new`。操作数的示例包括文本、字段、局部变量和表达式。

如果表达式包含多个运算符，那么运算符的*优先级*决定了各个运算符的计算顺序。例如，表达式 `x + y * z` 相当于计算 `x + (y * z)`，因为 `*` 运算符的优先级高于 `+` 运算符。

大多数运算符都可以*重载*。借助运算符重载，可以为一个或两个操作数为用户定义类或结构类型的运算指定用户定义运算符实现代码。

下表汇总了C#的运算符，按优先级从高到低的顺序列出了运算符类别。同一类别的运算符的优先级也相同。

基本	`x.m`	成员访问
	`x(...)`	方法和委托调用
	`x[...]`	数组和索引器访问
	`x++`	后递增
	`x--`	后递减
	`new T(...)`	对象和委托创建
	`new T(...){...}`	使用初始值设定项创建对象
	`new {...}`	匿名对象初始值设定项
	`new T[...]`	数组创建
	`typeof(T)`	获取 `T` 的 `System.Type` 对象
	`checked(x)`	在已检查的上下文中计算表达式
	`unchecked(x)`	在未检查的上下文中计算表达式
	`default(T)`	获取类型为 `T` 的默认值
	`delegate {...}`	匿名函数（匿名方法）
一元	`+x`	标识
	`-x`	求反
	`!x`	逻辑求反
	`~x`	按位求反
	`++x`	前递增
	`--x`	前递减
	`(T)x`	将 `x` 显式转换为类型 `T`
	`await x`	异步等待 `x` 完成

[	[[	[
乘法	`x * y`	乘法
	`x / y`	除号
	`x % y`	余数
加法	`x + y`	相加、字符串串联、委托组合
	`x - y`	相减、委托移除
移位	`x << y`	左移
	`x >> y`	右移
关系和类型测试	`x < y`	小于
	`x > y`	大于
	`x <= y`	小于或等于
	`x >= y`	大于或等于
	`x is T`	如果 `x` 是 `T`，则返回 `true`；否则，返回 `false`
	`x as T`	返回类型为 `T` 的 `x`；如果 `x` 的类型不是 `T`，则返回 `null`
相等	`x == y`	等于
	`x != y`	不等于
逻辑“与”	`x & y`	整型按位 AND，布尔型逻辑 AND
逻辑 XOR	`x ^ y`	整型按位 XOR，布尔型逻辑 XOR
逻辑“或”	`x | y`	整型按位“或”，布尔型逻辑“或”
条件“与”	`x && y`	仅 `y` 当 `x` 为时才计算 `true`
条件“或”	`x || y`	仅 `y` 当 `x` 为时才计算 `false`
null 合并	`x ?? y`	如果为，则计算结果 `x` 为，否则为 `null` `x` `y`
条件运算	`x ? y : z`	如果 `x` 为 `true`，则计算 `y`；如果 `x` 为 `false`，则计算 `z`
赋值或匿名函数	`x = y`	赋值

	`x op= y`	复合赋值;支持的运算符 `*=` 为 `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`
	`(T x) => y`	匿名函数(lambda 表达式)

语句

程序操作使用*语句*进行表示。C# 支持几种不同的语句，其中许多语句是从嵌入语句的角度来定义的。

使用***代码块***，可以在允许编写一个语句的上下文中编写多个语句。代码块是由一系列在分隔符 `{` 和 `}` 内编写的语句组成。

声明语句用于声明局部变量和常量。

表达式语句用于计算表达式。可用作语句的表达式包括方法调用、使用运算符的 `new` 对象分配、使用 `=` 和复合赋值运算符的赋值、增量和减量运算（使用） `++` and `--` 运算符和 await 表达式。

*选择语句*用于根据一些表达式的值从多个可能的语句中选择一个以供执行。这一类语句包括 `if` 和 `switch` 语句。

迭代语句用于重复执行嵌入语句。这一类语句包括 `while`、`do`、`for` 和 `foreach` 语句。

跳转语句用于转移控制权。这一类语句包括 `break`、`continue`、`goto`、`throw`、`return` 和 `yield` 语句。

`try` ... `catch` 语句用于捕获在代码块执行期间发生的异常，`try` ... `finally` 语句用于指定始终执行的最终代码，无论异常发生与否。

`checked` 和 `unchecked` 语句用于控制整型算术运算和转换的溢出检查上下文。

`lock` 语句用于获取给定对象的相互排斥锁定，执行语句，然后解除锁定。

`using` 语句用于获取资源，执行语句，然后释放资源。

下面是每种语句的示例

局部变量声明

```
static void Main() {
   int a;
   int b = 2, c = 3;
   a = 1;
   Console.WriteLine(a + b + c);
}
```

局部常量声明

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

Expression 语句

```
static void Main() {
    int i;
    i = 123;               // Expression statement
    Console.WriteLine(i);  // Expression statement
    i++;                   // Expression statement
    Console.WriteLine(i);  // Expression statement
}
```

`if` **语句**

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

`switch` **语句**

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

`while` **语句**

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

`do` **语句**

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

`for` **语句**

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

`foreach` **语句**

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

`break` **语句**

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

`continue` **语句**

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

`goto` **语句**

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

`return` **语句**

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

`yield` **语句**

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

`throw` 和 `try` 语句

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

`checked` 和 `unchecked` 语句

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

`lock` 语句

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

`using` 语句

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

类和对象

*类*是最基本的 C# 类型。类是一种数据结构，可在一个单元中就将状态（字段）和操作（方法和其他函数成员）结合起来。类为动态创建的类*实例*（亦称为"*对象*"）提供了定义。类支持*继承*和*多形性*，即*派生类*可以扩展和专门针对*基类*的机制。

新类使用类声明进行创建。类声明的开头是标头，指定了类的特性和修饰符、类名、基类（若指定）以及类实现的接口。标头后面是类主体，由在分隔符 `{` 和 `}` 内编写的成员声明列表组成。

以下是简单类 `Point` 的声明：

```
public class Point
{
    public int x, y;

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

类实例是使用 `new` 运算符进行创建，此运算符为新实例分配内存，调用构造函数来初始化实例，并返回对实例的引用。以下语句创建两个 `Point` 对象，并将对这些对象的引用存储在两个变量中：

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

当不再使用对象时，将自动回收对象占用的内存。既没必要，也无法在 C# 中显式解除分配对象。

成员

类的成员为***静态成员***或***实例成员***。静态成员属于类，而实例成员则属于对象（类实例）。

下表提供了类可以包含的成员种类的概述。

常量	与类相关联的常量值
字段	类的常量
方法	类可以执行的计算和操作
属性	与读取和写入类的已命名属性相关联的操作
索引器	与将类实例编入索引(像处理数组一样)相关联的操作
事件	类可以生成的通知
运算符	类支持的转换和表达式运算符
构造函数	初始化类实例或类本身所需的操作
析构函数	永久放弃类实例前要执行的操作
类型	类声明的嵌套类型

可访问性

每个类成员都有关联的可访问性，用于控制能够访问成员的程序文本区域。可访问性有五种可能的形式。下表概述了这些报表。

`public`	访问不受限
`protected`	只能访问此类或派生自此类的类
`internal`	只能访问此程序
`protected internal`	只能访问此程序或派生自此类的类
`private`	只能访问此类

类型参数

类定义可能会按如下方式指定一组类型参数：在类名后面用尖括号括住类型参数名称列表。可以在类声明的主体中使用类型参数来定义类的成员。在以下示例中，`Pair` 的类型参数是 `TFirst` 和 `TSecond`：

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

声明为采用类型参数的类类型称为泛型类类型。结构、接口和委托类型也可以是泛型。

使用泛型类时，必须为每个类型参数提供类型自变量：

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

提供类型参数的泛型类型(如上文 `Pair<int,string>` 所示)称为构造类型。

基类

类声明可能会按如下方式指定基类:在类名和类型参数后面编写冒号和基类名。省略基类规范与从 `object` 类型派生相同。在以下示例中, `Point3D` 的基类是 `Point` , `Point` 的基类是 `object` :

```
public class Point
{
    public int x, y;

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

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

类继承其基类的成员。继承意味着类隐式包含其基类的所有成员(实例和静态构造函数除外)和基类的析构函数。派生类可以其继承的类添加新成员,但无法删除继承成员的定义。在上面的示例中, `Point3D` 从 `Point` 继承了 `x` 和 `y` 字段,每个 `Point3D` 实例均包含三个字段(`x` 、 `y` 和 `z`)。

可以将类类型隐式转换成其任意基类类型。因此,类类型的变量可以引用相应类的实例或任意派生类的实例。例如,类声明如上, `Point` 类型的变量可以引用 `Point` 或 `Point3D` :

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

字段

字段是与类或类的实例关联的变量。

使用 `static` 修饰符声明的字段定义***静态字段***。静态字段只指明一个存储位置。无论创建多少个类实例,永远只有一个静态字段副本。

未使用 `static` 修饰符声明的字段定义***实例字段***。每个类实例均包含相应类的所有实例字段的单独副本。

在以下示例中,每个 `Color` 类实例均包含 `r` 、 `g` 和 `b` 实例字段的单独副本,但分别只包含 `Black` 、 `White` 、 `Red` 、 `Green` 和 `Blue` 静态字段的一个副本:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

如上面的示例所示，可以使用 `readonly` 修饰符声明***只读字段***。对 `readonly` 字段的赋值只能作为字段声明的一部分出现，或者出现在同一类的构造函数中。

方法

方法是实现对象或类可执行的计算或操作的成员。***静态方法***是通过类进行访问。*实**例方法***是通过类实例进行访问。

方法具有一个（可能为空）***参数***列表，这些参数表示传递给方法的值或变量***引用，后者***指定方法所计算和返回的值的类型。如果不返回值，则 `void` 为方法的返回类型。

方法可能也包含一组类型参数，必须在调用方法时指定类型自变量，这一点与类型一样。与类型不同的是，通常可以根据方法调用的自变量推断出类型自变量，无需显式指定。

在声明方法的类中，方法的*签名*必须是唯一的。方法签名包含方法名称、类型参数数量及其参数的数量、修饰符和类型。方法签名不包含返回类型。

参数

参数用于将值或变量引用传递给方法。方法参数从调用方法时指定的***自变量***中获取其实际值。有四类参数：值参数、引用参数、输出参数和参数数组。

值参数用于传递输入参数。值参数对应于局部变量，从为其传递的自变量中获取初始值。修改值参数不会影响为其传递的自变量。

可以指定默认值，从而省略相应的自变量，这样值参数就是可选的。

引用参数用于传递输入和输出参数。为引用参数传递的自变量必须是变量，并且在方法执行期间，引用参数指明的存储位置与自变量相同。引用参数使用 `ref` 修饰符进行声明。下面的示例展示了如何使用 `ref` 参数。

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);           // Outputs "2 1"
    }
}
```

输出参数用于传递输出参数。输出参数与引用参数类似，不同之处在于，调用方提供的自变量的初始值并不重要。输出参数使用 `out` 修饰符进行声明。下面的示例展示了如何使用 `out` 参数。

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

参数数组允许向方法传递数量不定的自变量。参数数组使用 `params` 修饰符进行声明。参数数组只能是方法的最后一个参数，且参数数组的类型必须是一维数组类型。类的 `WriteLine` 和方法是参数数组用法的很好示例。`Write` `System.Console` 它们的声明方式如下。

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

在使用参数数组的方法中，参数数组的行为与数组类型的常规参数完全相同。不过，在调用包含参数数组的方法时，要么可以传递参数数组类型的一个自变量，要么可以传递参数数组的元素类型的任意数量自变量。在后一种情况中，数组实例会自动创建，并初始化为包含给定的自变量。以下示例：

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

等同于编写以下代码：

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

方法主体和局部变量

方法的主体指定在调用方法时要执行的语句。

方法主体可以声明特定于方法调用的变量。此类变量称为***局部变量***。局部变量声明指定了类型名称、变量名称以及可能的初始值。下面的示例声明了初始值为零的局部变量 `i` 和无初始值的局部变量 `j`。

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# 要求必须先***明确赋值***局部变量，然后才能获取其值。例如，如果上面的 `i` 声明未包含初始值，那么编译器会在后面使用 `i` 时报告错误，因为在后面使用时 `i` 不会在程序中进行明确赋值。

方法可以使用 `return` 语句将控制权返回给调用方。在返回 `void` 的方法中，`return` 语句无法指定表达式。在返回非 `void` 的方法中，`return` 语句必须包含用于计算返回值的表达式。

静态和实例方法

使用 `static` 修饰符声明的方法是***静态方法***。静态方法不对特定的实例起作用，只能直接访问静态成员。

不带 `static` 修饰符声明的方法是***实例方法***。实例方法对特定的实例起作用，并能够访问静态和实例成员。其中调用实例方法的实例可以作为 `this` 显式访问。在静态方法中引用 `this` 会生成错误。

以下 `Entity` 类包含静态和实例成员。

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

每个 `Entity` 实例均有一个序列号(很可能包含此处未显示的其他一些信息)。`Entity` 构造函数(类似于实例方法)将新实例初始化为包含下一个可用的序列号。由于构造函数是实例成员，因此可以访问 `serialNo` 实例字段和 `nextSerialNo` 静态字段。

`GetNextSerialNo` 和 `SetNextSerialNo` 静态方法可以访问 `nextSerialNo` 静态字段，但如果直接访问 `serialNo` 实例字段，则会生成错误。

下面的示例演示如何使用 `Entity` 类。

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

请注意，`SetNextSerialNo` 和 `GetNextSerialNo` 静态方法是在类中调用，而 `GetSerialNo` 实例方法则是在类实例中调用。

虚方法、重写方法和抽象方法

如果实例方法声明中有 `virtual` 修饰符，可以将实例方法称为"***虚方法***"。如果不 `virtual` 存在修饰符，则称该方法为***非虚拟方法***。

调用虚方法时，为其调用方法的实例的***运行时类型***决定了要调用的实际方法实现代码。调用非虚方法时，实例的***编译时类型***是决定性因素。

可以在派生类中***重写***虚方法。当实例方法声明包含 `override` 修饰符时，此方法将重写具有相同签名的继承的虚方法。但如果虚方法声明中引入新方法，重写方法声明通过提供相应方法的新实现代码，专门针对现有的继承虚方法。

抽象方法是无实现的虚方法。抽象方法使用 `abstract` 修饰符进行声明，只允许在也声明 `abstract` 的类中使用。必须在所有非抽象派生类中重写抽象方法。

下面的示例声明了一个抽象类 `Expression`，用于表示表达式树节点；还声明了三个派生类（`Constant`、`VariableReference` 和 `Operation`），用于实现常量、变量引用和算术运算的表达式树节点。（这类似于，但不与表达式树类型中引入的表达式树类型相混淆）。

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

上面的四个类可用于进行算术表达式建模。例如，使用这些类的实例，可以按如下方式表示表达式 `x + 3`。

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

调用 `Expression` 实例的 `Evaluate` 方法可以计算给定的表达式并生成 `double` 值。方法采用作为参数 `Hashtable`，其中包含变量名称(作为项键)和值(作为项的值)。`Evaluate` 方法是一个虚拟抽象方法，这意味着非抽象派生类必须重写它以提供实际实现。

`Constant` 的 `Evaluate` 实现代码只返回存储的常量。`VariableReference` 的实现查找哈希表中的变量名并返回结果值。`Operation` 实现代码先计算左右操作数(以递归方式调用其 `Evaluate` 方法)，然后执行给定的算术运算。

以下程序使用 `Expression` 类根据不同的 `x` 和 `y` 值计算表达式 `x * (y + 2)`。

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "16.5"
    }
}
```

方法重载

借助方法*重载*，同一类中可以有多个同名的方法，只要这些方法具有唯一签名即可。编译如何调用重载的方法时，编译器使用*重载决策*来确定要调用的特定方法。重载决策查找与自变量最匹配的方法;如果找不到最佳匹配项，则会报告错误。下面的示例展示了重载决策的实际工作方式。`Main` 方法中每个调用的注释指明了实际调用的方法。

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                // Invokes F()
        F(1);               // Invokes F(int)
        F(1.0);             // Invokes F(double)
        F("abc");           // Invokes F(object)
        F((double)1);       // Invokes F(double)
        F((object)1);       // Invokes F(object)
        F<int>(1);          // Invokes F<T>(T)
        F(1, 1);            // Invokes F(double, double)
    }
}
```

如示例所示，可以随时将自变量显式转换成确切的参数类型，并/或显式提供类型自变量，从而选择特定的方法。

其他函数成员

包含可执行代码的成员统称为类的***函数成员***。上一部分介绍了作为主要函数成员类型的方法。本部分介绍支持的其他类型的函数成员C#：构造函数、属性、索引器、事件、运算符和析构函数。

下面的代码演示了一个名 `List<T>` 为的泛型类，该类实现对象的可扩充列表。此类包含最常见类型函数成员的多个示例。

```
public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
        get { return count; }
    }
    public int Capacity {
```

```
    public int Capacity {
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
}
```

构造函数

C# 支持实例和静态构造函数。***实例构造函数***是实现初始化类实例所需执行的操作的成员。***静态构造函数***是实现在首次加载类时初始化类本身所需执行的操作的成员。

构造函数的声明方式与方法一样，都没有返回类型，且与所含类同名。如果构造函数声明包含`static`修饰符，则声明静态构造函数。否则，声明的是实例构造函数。

可以重载实例构造函数。例如，`List<T>` 类声明两个实例构造函数：一个没有参数，另一个需要使用 `int` 参数。实例构造函数使用 `new` 运算符进行调用。以下语句使用 `List` 类的 `List<string>` 每个构造函数分配两个实例。

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

与其他成员不同，实例构造函数不能被继承，且类中只能包含实际已声明的实例构造函数。如果没有为类提供实例构造函数，则会自动提供不含参数的空实例构造函数。

属性

属性是字段的自然扩展。两者都是包含关联类型的已命名成员，用于访问字段和属性的语法也是一样的。不过，与字段不同的是，属性不指明存储位置。相反，属性包含*访问器*，用于指定在读取或写入属性值时要执行的语句。

属性 `get` 的声明方式与字段类似，不同之处在于声明以取值函数和/ `set` 或分隔符 `{ }` 结尾，而不是以分号结束。同时 `get` 具有访问器 `set` 和访问器的属性是***读写属性*** `get` ，只有访问器的属性是***只读属性***，而只有 `set` 访问器的属性是***只写属性***。

`get` 访问器与具有属性类型返回值的无参数方法相对应。除了作为赋值的目标，当在表达式 `get` 中引用属性时，将调用属性的访问器来计算属性的值。

访问器对应于方法，该方法具有一个名 `value` 为的参数，没有返回类型。 `set` 当属性作为赋值的目标或作为 `++` 或 `--` 的操作数进行引用时，将 `set` 使用提供新值的自变量调用访问器。

`List<T>` 类声明以下两个属性：`Count` 和 `Capacity`（分别为只读和读写）。下面的示例展示了如何使用这些属性。

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;            // Invokes get accessor
int j = names.Capacity;         // Invokes get accessor
```

类似于字段和方法，C# 支持实例属性和静态属性。静态属性是用修饰符声明 `static` 的，实例属性是在没有它的情况下声明的。

属性的访问器可以是虚的。如果属性声明包含 `virtual`、`abstract` 或 `override` 修饰符，则适用于属性的访问器。

索引器

借助***索引器***成员，可以将对象编入索引（像处理数组一样）。索引器的声明方式与属性类似，不同之处在于，索引器成员名称格式为 `this` 后跟在分隔符 `[` 和 `]` 内写入的参数列表。这些参数在索引器的访问器中可用。类似于属性，索引器分为读写、只读和只写索引器，且索引器的访问器可以是虚的。

`List` 类声明一个需要使用 `int` 参数的读写索引器。借助索引器，可以使用 `int` 值将 `List` 实例编入索引。例如

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

索引器可以进行重载。也就是说，类可以声明多个索引器，只要其参数的数量或类型不同即可。

事件

借助***事件***成员，类或对象可以提供通知。事件的声明方式与字段类似，不同之处在于声明 `event` 包含关键字，而类型必须是委托类型。

在声明事件成员的类中，事件的行为与委托类型的字段完全相同（前提是事件不是抽象的，且不声明访问器）。字段存储对委托的引用，委托表示已添加到事件的事件处理程序。如果不存在任何事件句柄，则字段 `null` 为。

`List<T>` 类声明一个 `Changed` 事件成员，指明已向列表添加了新项。此 `Changed` 事件 `null` 由虚拟方法引发，该方法首先检查事件是否为（表示没有处理程序）。`OnChanged` 引发事件的概念恰恰等同于调用由事件表示的委托，因此，没有用于引发事件的特殊语言构造。

客户端通过***事件处理程序***响应事件。使用 `+=` 和 `-=` 运算符分别可以附加和删除事件处理程序。下面的示例展示了如何向 `List<string>` 的 `Changed` 事件附加事件处理程序。

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);        // Outputs "3"
    }
}
```

对于需要控制事件的基础存储的高级方案，事件声明可以显式提供 `add` 和 `remove` 访问器，这在某种程度上与属性的 `set` 访问器类似。

运算符

运算符是定义向类实例应用特定表达式运算符的含义的成员。可以定义三种类型的运算符：一元运算符、二元运算符和转换运算符。所有运算符都必须声明为 `public` 和 `static`。

`List<T>` 类声明两个运算符（`operator==` 和 `operator!=`），因此定义了向 `List` 实例应用这些运算符的表达式的新含义。具体而言，运算符将两 `List<T>` 个实例的相等性定义为使用其 `Equals` 方法来比较每个包含的对象。下面的示例展示了如何使用 `==` 运算符比较两个 `List<int>` 实例。

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);        // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);        // Outputs "False"
    }
}
```

第一个 `Console.WriteLine` 输出 `True`，因为两个列表包含的对象不仅数量相同，而且值和顺序也相同。如果 `List<T>` 未定义 `operator==`，那么第一个 `Console.WriteLine` 会输出 `False`，因为 `a` 和 `b` 引用不同的 `List<int>` 实例。

析构函数

析构函数是实现析构类的实例所需的操作的成员。析构函数不能有参数，它们不能具有可访问性修饰符，也不能被显式调用。在垃圾回收过程中会自动调用实例的析构函数。

垃圾回收器允许使用广泛的纬度来确定何时收集对象和运行析构函数。具体而言，析构函数调用的时间是不确定的，析构函数可以在任何线程上执行。出于这些原因和其他原因，类应仅在没有其他任何解决方案可行时才实现析构函数。

处理对象析构的更好方法是使用 `using` 语句。

结构

结构是可以包含数据成员和函数成员的数据结构，这一点与类一样；与类不同的是，结构是值类型，无需进行堆分配。结构类型的变量直接存储结构数据，而类类型的变量存储对动态分配的对象的引用。结构类型不支持用户指定的继承，并且所有结构类型均隐式继承自类型 `object`。

结构对包含值语义的小型数据结构特别有用。复数、坐标系中的点或字典中的键值对都是结构的典型示例。对小型数据结构使用结构（而不是类）在应用程序执行的内存分配次数上存在巨大差异。例如，以下程序创建并初始化包含 100 个点的数组。通过将 `Point` 实现为类，可单独实例化 101 个对象，一个对象用于数组，其他所有对象分别用于 100 个元素。

```
class Point
{
    public int x, y;

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

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

另一种方法是 `Point` 创建结构。

```
struct Point
{
    public int x, y;

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

现在，仅实例化一个对象（即用于数组的对象），`Point` 实例存储内嵌在数组中。

结构构造函数使用 `new` 运算符进行调用，但这不并表示要分配内存。结构构造函数只返回结构值本身（通常在堆栈的临时位置中），并在必要时复制此值，而非动态分配对象并返回对此对象的引用。

借助类，两个变量可以引用同一对象；因此，对一个变量执行的运算可能会影响另一个变量引用的对象。借助结构，每个变量都有自己的数据副本；因此，对一个变量执行的运算不会影响另一个变量。例如，下面的代码段生成的输出取决于 `Point` 是类还是结构。

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

如果 `Point` 是一个类，则输出 `a` 为 `20` ，并且 `b` 引用相同的对象。如果 `Point` 是一个结构，则输出为 `10` ， `b` 因为的赋值 `a` 创建值的副本 `a.x` ，而此副本不受对的后续赋值的影响。

以上示例突出显示了结构的两个限制。首先，复制整个结构通常比复制对象引用效率更低，因此通过结构进行的赋值和值参数传递可能比通过引用类型成本更高。其次，除 `ref` 和 `out` 参数以外，无法创建对结构的引用，这就表示在很多应用场景中都不能使用结构。

数组

数组是一种数据结构，其中包含许多通过计算索引访问的变量。数组中的变量（亦称为数组的***元素***）均为同一种类型，我们将这种类型称为数组的***元素类型***。

数组类型是引用类型，声明数组变量只是为引用数组实例预留空间。实际的数组实例是在运行时使用 `new` 运算符动态创建的。操作指定新数组实例的长度，该长度在实例的生存期内是固定的。 `new` 数组元素的索引介于 `0` 到 `Length - 1` 之间。 `new` 运算符自动将数组元素初始化为其默认值（例如，所有数值类型的默认值为 0，所有引用类型的默认值为 `null` ）。

以下示例创建 `int` 元素数组，初始化此数组，然后打印输出此数组的内容。

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

上面的示例创建一*维数组*，并对其执行运算。C# 还支持*多维数组*。数组类型的维数（亦称为数组类型的*秩*）是 1 与数组类型方括号内的逗号数量相加的结果。下面的示例分配一个一维、二维和三维数组。

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

`a1` 数组包含 10 个元素，`a2` 数组包含 50 个元素 (10 × 5)，`a3` 数组包含 100 个元素 (10 × 5 × 2)。

数组的元素类型可以是任意类型（包括数组类型）。包含数组类型元素的数组有时称为*交错数组*，因为元素数组的长度不必全都一样。以下示例分配由 `int` 数组构成的数组：

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

第一行创建包含三个元素的数组，每个元素都是 `int[]` 类型，并且初始值均为 `null`。后面的代码行将这三个元素初始化为引用长度不同的各个数组实例。

运算符允许使用数组初始值设定项指定数组元素的初始值，*数组初始值设定项*是在分隔符 `{` 和 `}` 之间编写的表达式的列表。`new` 以下示例分配 `int[]`，并将其初始化为包含三个元素。

```
int[] a = new int[] {1, 2, 3};
```

请注意，数组的长度是从和 `{` `}` 之间的表达式数推断出来的。局部变量和字段声明可以进一步缩短，这样就不用重新声明数组类型了。

```
int[] a = {1, 2, 3};
```

以上两个示例等同于以下示例：

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

接口

接口定义了可由类和结构实现的协定。接口可以包含方法、属性、事件和索引器。接口不提供所定义的成员的实现代码，仅指定必须由实现接口的类或结构提供的成员。

接口可以采用***多重继承***。在以下示例中，接口 `IComboBox` 同时继承自 `ITextBox` 和 `IListBox`。

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

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

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

类和结构可以实现多个接口。在以下示例中，类 `EditBox` 同时实现 `IControl` 和 `IDataBound`。

```
interface IDataBound
{
    void Bind(Binder b);
}

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

当类或结构实现特定接口时，此类或结构的实例可以隐式转换成相应的接口类型。例如

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

如果已知实例不是静态地实现特定接口，可以使用动态类型显式转换功能。例如，下面的语句使用动态类型强制转换来获取对象的 `IControl` 和 `IDataBound` 接口实现。由于对象的实际类型为 `EditBox`，因此强制转换成功。

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

`EditBox` 在以前的类中，`Paint` `IControl` 接口 `Bind` 中的方法和 `IDataBound` 接口中的方法是使用 `public` 成员实现的。C#还支持***显式接口成员实现***，使用该类或结构可以避免成为成员 `public`。显式接口成员实现代码是使用完全限定的接口成员名称进行编写。例如，`EditBox` 类可以使用显式接口成员实现代码来实现 `IControl.Paint` 和 `IDataBound.Bind` 方法，如下所示。

```
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类型进行访问。例如，只有 `IControl.Paint` 先将 `EditBox` 引用 `EditBox` `IControl` 转换为接口类型，才能调用上一个类提供的实现。

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error, no such method
IControl control = editBox;
control.Paint();                        // Ok
```

枚举

枚举类型是包含一组已命名常量的独特值类型。下面的示例声明并使用 `Color` 名为三个常数值 `Green` 、 `Red` 、和 `Blue` 的枚举类型。

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

每个枚举类型都有一个对应的整型类型，称为枚举类型的***基础类型***。不显式声明基础类型的枚举类型具有基础类型 `int` 。枚举类型的存储格式和可能值的范围由其基础类型决定。枚举类型可以采用的值集不受其枚举成员限制。特别是，枚举的基础类型的任何值都可以转换为枚举类型，并且是该枚举类型的一个不同的有效值。

下面的示例声明一个名 `Alignment` 为且基础类型为的 `sbyte` 枚举类型。

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

如前面的示例所示，枚举成员声明可以包含指定成员的值的常量表达式。每个枚举成员的常数值必须在该枚举的基础类型的范围内。当枚举成员声明未显式指定一个值时，将为该成员赋予值零（如果它是枚举类型中的第一个成员）或上一次的枚举成员的值加1。

可以使用类型强制转换将枚举值转换为整数值，反之亦然。例如

```
int i = (int)Color.Blue;        // int i = 2;
Color c = (Color)2;             // Color c = Color.Blue;
```

任何枚举类型的默认值都是整数值零，转换为枚举类型。如果变量自动初始化为默认值，则这是给定给枚举类型的变量的值。为了使枚举类型的默认值易于使用，文本 `0` 隐式转换为任何枚举类型。因此，可以运行以下命令。

```
Color c = 0;
```

委托

委托类型表示对具有特定参数列表和返回类型的方法的引用。通过委托，可以将方法视为可分配给变量并可作为参数传递的实体。委托类似于其他一些语言中的函数指针概念，但与函数指针不同的是，委托不仅面向对象，还类型安全。

下面的示例声明并使用 `Function` 委托类型。

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

`Function` 委托类型实例可以引用需要使用 `double` 自变量并返回 `double` 值的方法。方法将给定 `Function` 的应用于的元素 `double[]`， `double[]` 并将返回结果。`Apply` 在 `Main` 方法中，`Apply` 用于向 `double[]` 应用三个不同的函数。

委托可以引用静态方法（如上面示例中的 `Square` 或 `Math.Sin`）或实例方法（如上面示例中的 `m.Multiply`）。引用实例方法的委托还会引用特定对象，通过委托调用实例方法时，该对象会变成调用中的 `this`。

还可以使用匿名函数创建委托，这些函数是便捷创建的“内联方法”。匿名函数可以查看周围方法的局部变量。因此，可以更轻松地编写上面的乘数示例，而无 `Multiplier` 需使用类：

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

委托的一个有趣且有用的属性是，它不知道也不关心所引用的方法的类；只关心引用的方法是否具有与委托相同的参数和返回类型。

特性

C# 程序中的类型、成员和其他实体支持使用修饰符来控制其行为的某些方面。例如，方法的可访问性是由 `public`、`protected`、`internal` 和 `private` 修饰符控制。C# 整合了这种能力，以便可以将用户定义类型的声明性信息附加到程序实体，并在运行时检索此类信息。程序通过定义和使用***特性***来指定此类额外的声明性信息。

以下示例声明了 `HelpAttribute` 特性，可将其附加到程序实体，以提供指向关联文档的链接。

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

所有特性类均派生自 `System.Attribute` .NET Framework 提供的基类。特性的应用方式为，在相关声明前的方括号内指定特性的名称以及任意自变量。如果特性的名称以结尾 `Attribute`，则引用该属性时可以省略此部分名称。例如，可按如下方法使用 `HelpAttribute` 特性。

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

此示例将附加 `HelpAttribute` `Widget` 到类，并将 `HelpAttribute` 另一 `Display` 类附加到类中的方法。特性类的公共构造函数控制了将特性附加到程序实体时必须提供的信息。可以通过引用特性类的公共读写属性（如上面示例对 `Topic` 属性的引用），提供其他信息。

下面的示例演示如何使用反射在运行时检索给定程序实体的属性信息。

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

通过反射请求获得特定特性时，将调用特性类的构造函数（由程序源提供信息），并返回生成的特性实例。如果是通过属性提供其他信息，那么在特性实例返回前，这些属性会设置为给定值。

词法结构

2020/11/2 • • Edit Online

节目

C #***程序***由一个或多个***源文件***组成，该文件正式称为***编译单元***(编译单元)。源文件是 Unicode 字符的有序序列。源文件与文件系统中的文件通常具有一对一的对应关系，但不需要此函件。为获得最大的可移植性，建议使用 UTF-8 编码对文件系统中的文件进行编码。

从概念上讲，程序是使用三个步骤编译的：

1. 转换，将特定字符已知和编码方案中的文件转换为 Unicode 字符序列。
2. 词法分析，将 Unicode 输入字符流转换为标记流。
3. 语法分析，将令牌流转换为可执行代码。

语法

此规范提供使用两个语法的 c # 编程语言的语法。***词法语法***(词法语法)定义 Unicode 字符如何合并为行终止符、空格、注释、标记和预处理指令。***句法文法***(句法文法)定义如何组合词汇语法生成的标记以形成 c # 程序。

语法表示法

词法语法和句法语法以巴科斯-诺尔范式形式显示，使用 ANTLR 语法工具的表示法。

词法语法

C # 的词法语法显示在词法分析、标记和预处理指令中。词法语法的终端符号是 Unicode 字符集的字符，并且词法语法指定如何将字符组合到一起形成标记(标记)、空白(空格)、注释(注释)和预处理指令(预处理指令)。

C # 程序中的每个源文件都必须符合词法文法的*输入*生产(词法分析)。

语法语法

C # 的句法语法在本章后面的章节和附录中提供。语法语法的终端符号是由词法语法定义的标记，句法语法指定如何组合标记以形成 c # 程序。

C # 程序中的每个源文件都必须符合句法文法*compilation_unit*生产(编译单元)。

词法分析

*输入*生产定义 c # 源文件的词法结构。C # 程序中的每个源文件都必须符合此词法文法生产。

```
input
    : input_section?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* new_line
    | pp_directive
    ;

input_element
    : whitespace
    | comment
    | token
    ;
```

五个基本元素构成 c # 源文件的词法结构：行终止符（行终止符）、空白（空格）、注释（注释）、标记（标记）和预处理指令（预处理指令）。在这些基本元素中，只有标记在 c # 程序的句法语法中非常重要（句法语法）。

C # 源文件的词法处理包括将文件缩减为一系列标记，后者成为句法分析的输入。行终止符、空白和注释可用于分隔标记，预处理指令可能会导致跳过源文件的各个部分，否则，这些词法元素不会影响 c # 程序的语法结构。

在内插字符串文本（内插字符串文本）的情况下，单个标记最初由词法分析生成，但被分解为多个输入元素，这些输入元素会反复进入词法分析状态，直到所有内插字符串文本均已解决。然后，生成的令牌作为句法分析的输入。

当多个词法语法生产与源文件中的一系列字符匹配时，词法处理始终形成可能的最长词汇元素。例如，字符序列 `//` 作为单行注释的开头处理，因为该词法元素比单个 `/` 标记长。

行终止符

行结束符将 c # 源文件中的字符分为多行。

```
new_line
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Carriage return character (U+000D) followed by line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;
```

为了与添加文件结尾标记的源代码编辑工具兼容，若要将源文件视为一系列正确终止的行，请按顺序将以下转换应用于 c # 程序中的每个源文件：

- 如果源文件的最后一个字符是 Control Z 字符（ `U+001A` ），则删除该字符。
- 如果源文件不 `U+000D` 是空的，并且源文件的最后一个字符不是回车符（ `U+000D` ）、换行符（ `U+000A` ）、行分隔符（ `U+2028` ）或段落分隔符（），则将回车符（）添加到源文件的末尾 `U+2029` 。

注释

支持两种形式的注释：单行注释和分隔注释。S 注释以字符 `//` 开头，并扩展到源行的末尾。分隔注释以字符 `/*` 开头，以字符 `*/` 结尾。分隔注释可能跨多行。

```
comment
    : single_line_comment
    | delimited_comment
    ;

single_line_comment
    : '//' input_character*
    ;

input_character
    : '<Any Unicode character except a new_line_character>'
    ;

new_line_character
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

delimited_comment
    : '/*' delimited_comment_section* asterisk+ '/'
    ;

delimited_comment_section
    : '/'
    | asterisk* not_slash_or_asterisk
    ;

asterisk
    : '*'
    ;

not_slash_or_asterisk
    : '<Any Unicode character except / or *>'
    ;
```

注释不嵌套。字符序列 `/*` `*/` 在注释中没有特殊含义，并且 `//` 字符序列在 `//` `/*` 分隔注释中没有特殊含义。

在字符和字符串文本中不处理注释。

示例

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

包含分隔注释。

示例

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

显示若干单行注释。

空格

空格定义为带有 Unicode 类 Zs 的任何字符(包括空格字符)以及水平制表符、垂直制表符、换行符和换页符。

```
whitespace
    : '<Any character with Unicode class Zs>'
    | '<Horizontal tab character (U+0009)>'
    | '<Vertical tab character (U+000B)>'
    | '<Form feed character (U+000C)>'
    ;
```

令牌

有多种类型的令牌:标识符、关键字、文本、运算符和标点符号。空白和注释不是标记,不过它们充当标记的分隔符。

```
token
    : identifier
    | keyword
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | interpolated_string_literal
    | operator_or_punctuator
    ;
```

Unicode 字符转义序列

Unicode 字符转义序列表示一个 Unicode 字符。Unicode 字符转义序列在标识符(标识符)、字符文本(字符文本)和常规字符串文本(字符串文字)中进行处理。不会在任何其他位置(例如,使用 operator、标点符号或关键字)处理 Unicode 字符转义。

```
unicode_escape_sequence
    : '\\u' hex_digit hex_digit hex_digit hex_digit
    | '\\U' hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit
    ;
```

Unicode 转义序列表示由 " `\u` " 或 "" 字符后面的十六进制数构成的单个 unicode 字符 `\U` 。由于 c # 使用字符和字符串值中的 Unicode 码位的16位编码,字符文本中不允许使用 U + 10000 到 U + 10FFFF 范围内的 Unicode 字符,而是使用字符串文本中的 Unicode 代理项对来表示。不支持0x10FFFF 以上的码位的 Unicode 字符。

不会执行多个转换。例如,字符串文本 " `\u005Cu005C` " 等效于 " `\u005C` ",而不是 " `\` "。Unicode 值 `\u005C` 为字符 " `\` "。

示例

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

显示了的几个用法 `\u0066`，它是字母 "" 的转义序列 `f`。该程序等效于

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

标识符

本节中给出的标识符规则完全与 Unicode 标准附录31推荐的规则相对应，只不过允许将下划线作为初始字符（如 C 编程语言中的传统）、标识符中允许使用 Unicode 转义序列，并允许将 " `@` " 字符作为前缀，以使关键字可用作标识符。

```
identifier
    : available_identifier
    | '@' identifier_or_keyword
    ;

available_identifier
    : '<An identifier_or_keyword that is not a keyword>'
    ;

identifier_or_keyword
    : identifier_start_character identifier_part_character*
    ;

identifier_start_character
    : letter_character
    | '_'
    ;

identifier_part_character
    : letter_character
    | decimal_digit_character
    | connecting_character
    | combining_character
    | formatting_character
    ;

letter_character
    : '<A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    | '<A unicode_escape_sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    ;

combining_character
    : '<A Unicode character of classes Mn or Mc>'
    | '<A unicode_escape_sequence representing a character of classes Mn or Mc>'
    ;

decimal_digit_character
    : '<A Unicode character of the class Nd>'
    | '<A unicode_escape_sequence representing a character of the class Nd>'
    ;

connecting_character
    : '<A Unicode character of the class Pc>'
    | '<A unicode_escape_sequence representing a character of the class Pc>'
    ;

formatting_character
    : '<A Unicode character of the class Cf>'
    | '<A unicode_escape_sequence representing a character of the class Cf>'
    ;
```

有关上面提到的 Unicode 字符类的信息，请参阅 Unicode 标准版本3.0，第4.5 节。

有效标识符的示例包括 " `identifier1` "、" `_identifier2` " 和 " `@if` "。

符合标准的程序中的标识符必须是 Unicode 范式 C 定义的规范格式，如 Unicode 标准附录15所定义。如果遇到非范式规范的标识符，则该行为是实现定义的;但是，不需要诊断。

前缀 " `@` " 允许将关键字用作标识符，这在与其他编程语言交互时非常有用。该字符 `@` 实际上不是标识符的一部分，因此标识符可能以其他语言显示为普通标识符，不含前缀。带有前缀的标识符 `@` 称为***逐字标识符***。`@` 允许对不是关键字的标识符使用前缀，但强烈建议不要使用它作为样式。

示例：

```
class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
    }
}
```

定义名为 "" 的类，该类具有名为 "" `class` 的静态方法 `static`，该方法采用名为 "" 的参数 `bool`。请注意，由于关键字中不允许使用 Unicode 转义，因此标记 " `cl\u0061ss` " 是标识符，与 "" 具有相同的标识符 `@class`。

如果两个标识符在应用以下转换后相同，则将其视为相同：

- 删除前缀 " `@` "（如果使用）。
- 每个*unicode_escape_sequence*都转换为其对应的 unicode 字符。
- 删除任何*formatting_character*。

包含两个连续下划线字符（ `U+005F` ）的标识符保留给实现使用。例如，实现可能提供以两个下划线开头的扩展关键字。

关键字

关键字是类似于标识符的字符序列（保留），不能用作标识符，除非以 `@` 字符开头。

```
keyword
    : 'abstract' | 'as'       | 'base'       | 'bool'      | 'break'
    | 'byte'     | 'case'     | 'catch'      | 'char'      | 'checked'
    | 'class'    | 'const'    | 'continue'   | 'decimal'   | 'default'
    | 'delegate' | 'do'       | 'double'     | 'else'      | 'enum'
    | 'event'    | 'explicit' | 'extern'     | 'false'     | 'finally'
    | 'fixed'    | 'float'    | 'for'        | 'foreach'   | 'goto'
    | 'if'       | 'implicit' | 'in'         | 'int'       | 'interface'
    | 'internal' | 'is'       | 'lock'       | 'long'      | 'namespace'
    | 'new'      | 'null'     | 'object'     | 'operator'  | 'out'
    | 'override' | 'params'   | 'private'    | 'protected' | 'public'
    | 'readonly' | 'ref'      | 'return'     | 'sbyte'     | 'sealed'
    | 'short'    | 'sizeof'   | 'stackalloc' | 'static'    | 'string'
    | 'struct'   | 'switch'   | 'this'       | 'throw'     | 'true'
    | 'try'      | 'typeof'   | 'uint'       | 'ulong'     | 'unchecked'
    | 'unsafe'   | 'ushort'   | 'using'      | 'virtual'   | 'void'
    | 'volatile' | 'while'
    ;
```

在语法中的某些位置，特定标识符具有特殊意义，但不是关键字。此类标识符有时称为 "上下文关键字"。例如，在属性声明中，" `get` " 和 " `set` " 标识符具有特殊意义（取值函数）。此位置不允许使用或以外的标识符 `get` `set`，因此，此使用不会与使用这些字词作为标识符冲突。在其他情况下，如 `var` 在隐式类型化局部变量声明（局部变量声明）中使用标识符 "" 时，上下文关键字可能与声明的名称冲突。在这种情况下，声明的名称优先于将标识符用作上下文关键字。

文字

文本是值的源代码表示形式。

```
literal
    : boolean_literal
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | null_literal
    ;
```

布尔值文字

有两个布尔文本值：`true` 和 `false` 。

```
boolean_literal
    : 'true'
    | 'false'
    ;
```

*Boolean_literal*的类型为 `bool` 。

整数文本

整数文本用于写入类型为、、和的值 `int` `uint` `long` `ulong` 。整数文本具有两种可能的形式：decimal 和十六进制。

```
integer_literal
    : decimal_integer_literal
    | hexadecimal_integer_literal
    ;

decimal_integer_literal
    : decimal_digit+ integer_type_suffix?
    ;

decimal_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;

integer_type_suffix
    : 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU' | 'lu'
    ;

hexadecimal_integer_literal
    : '0x' hex_digit+ integer_type_suffix?
    | '0X' hex_digit+ integer_type_suffix?
    ;

hex_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f';
```

确定整数文本的类型，如下所示：

- 如果文字没有后缀，则其值可以表示为以下类型的第一个类型：`int` 、 `uint` 、 `long` 和 `ulong` 。
- 如果文本的后缀为 `U` 或 `u` ，则它具有以下类型中可以表示其值的第一个类型： `uint` 、 `ulong` 。
- 如果文本的后缀为 `L` 或 `l` ，则它具有以下类型中可以表示其值的第一个类型： `long` 、 `ulong` 。
- 如果文本的后缀为 `UL` 、 `Ul` 、、、、、 `uL` `ul` `LU` `Lu` `lU` 或，则 `lu` 它属于类型 `ulong` 。

如果整数文本所表示的值超出了该类型的范围 `ulong` ，则会发生编译时错误。

作为样式，建议 `L` 在写入类型的文本时使用 "" 而不是 "" `l` `long` ，因为这样可以很容易地将字母 " `l` " 与数字 " `1` " 混淆。

若要允许尽可能小的 `int` 和 `long` 值写入为十进制整数文本，请满足以下两个规则：

- 当值为2147483648（2 ^ 31）且没有*integer_type_suffix*的*decimal_integer_literal*显示为紧跟一元减号运算符（一元减号运算符）的标记时，结果为类型 `int` 值为-2147483648 (-2 ^ 31)的常量。在所有其他情况下，这类*decimal_integer_literal*的类型为 `uint` 。
- 当*decimal_integer_literal*值为9223372036854775808（2 ^ 63）且没有*integer_type_suffix*或*integer_type_suffix*时 `L` ，或在 `l` 一元减号运算符（一元减号运算符）之后立即显示标记时，结果为类型的常量，其 `long` 值为-9223372036854775808 (-2 ^ 63)。在所有其他情况下，这类*decimal_integer_literal*的类型为 `ulong` 。

真实文本

真实文本用于写入类型为 `float` 、和的值 `double` `decimal` 。

```
real_literal
    : decimal_digit+ '.' decimal_digit+ exponent_part? real_type_suffix?
    | '.' decimal_digit+ exponent_part? real_type_suffix?
    | decimal_digit+ exponent_part real_type_suffix?
    | decimal_digit+ real_type_suffix
    ;

exponent_part
    : 'e' sign? decimal_digit+
    | 'E' sign? decimal_digit+
    ;

sign
    : '+'
    | '-'
    ;

real_type_suffix
    : 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
    ;
```

如果未指定*real_type_suffix*，则真实文本的类型为 `double` 。否则，实数类型后缀将确定真实文本的类型，如下所示：

- 用 `F` 或作为后缀的实文本 `f` 的类型为 `float` 。例如，文本、、 `1f` `1.5f` `1e10f` 和 `123.456F` 都是类型 `float` 。
- 用 `D` 或作为后缀的实文本 `d` 的类型为 `double` 。例如，文本、、 `1d` `1.5d` `1e10d` 和 `123.456D` 都是类型 `double` 。
- 用 `M` 或作为后缀的实文本 `m` 的类型为 `decimal` 。例如，文本、、 `1m` `1.5m` `1e10m` 和 `123.456M` 都是类型 `decimal` 。通过采用精确值将此文本转换为一个 `decimal` 值，并在必要时使用银行家舍入（decimal 类型）舍入为最接近的可表示值。除非值舍入或值为零（在这种情况下，正负号为0），否则文本中的任何小数位数都将保留。因此， `2.900m` 将分析文本，以形成带有符号 `0` 、系数 `2900` 和刻度的小数 `3` 。

如果指定的文本不能用指定的类型表示，则会发生编译时错误。

类型或的实字的值 `float` `double` 是通过使用 IEEE "舍入到最近的" 模式确定的。

请注意，在实际文本中，小数点后始终需要小数位数。例如， `1.3F` 是一个真实文本，但 `1.F` 不是。

字符文本

字符文本表示单个字符，通常由引号中的字符组成，如中所示 `'a'` 。

注意：ANTLR 语法表示法会使以下混乱！ 在 ANTLR 中，当您编写时， `\'` 它代表一个引号 `'` 。当你编写时， `\\` 它代表单个反斜杠 `\` 。因此，字符文本的第一个规则意味着以单个单引号开始，然后是一个字符，然后是一个引号。还有11个可能的简单转义序列 `\'` :、、、 `\"` `\\` `\0` 、 `\a` 、 `\b` `\f` `\n` `\r` `\t` `\v` 、、、、、。

```
character_literal
    : '\'' character '\''
    ;

character
    : single_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;

single_character
    : '<Any character except \' (U+0027), \\ (U+005C), and new_line_character>'
    ;

simple_escape_sequence
    : '\\\'' | '\\"' | '\\\\' | '\\0' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v'
    ;

hexadecimal_escape_sequence
    : '\\x' hex_digit hex_digit? hex_digit? hex_digit?;
```

在字符中跟在反斜杠字符()后面的字符 `\` 必须是下列字符之一：*character* `'` 、 `"` 、 `\` 、、 `0` `a` `b` `f` `n` `r` `t` `u` `U` `x` `v` 、、、、、、、、、。否则，将发生编译时错误。

十六进制转义序列表示单个 Unicode 字符，其值由 "" 后面的十六进制数构成 `\x` 。

如果字符文本表示的值大于 `U+FFFF` ，则会发生编译时错误。

字符文本中的 Unicode 字符转义序列（unicode 字符转义序列）必须在到的范围内 `U+0000` `U+FFFF` 。

简单的转义序列表示 Unicode 字符编码，如下表中所述。

IIIII	IIIII	UNICODE II
`\'`	单引号	`0x0027`
`\"`	双引号	`0x0022`
`\\`	反斜杠	`0x005C`
`\0`	Null	`0x0000`
`\a`	警报	`0x0007`
`\b`	Backspace	`0x0008`
`\f`	换页	`0x000C`
`\n`	换行	`0x000A`
`\r`	回车	`0x000D`
`\t`	水平制表符	`0x0009`
`\v`	垂直制表符	`0x000B`

*Character_literal*的类型为 `char` 。

字符串文本

C # 支持两种形式的字符串文本：***常规字符串文本***和***原义字符串***。

正则字符串文字由零个或多个字符括在双引号中，如中所示， `"hello"` 并且可能包括简单转义序列（如 `\t` 制表符）和十六进制和 Unicode 转义序列。

原义字符串包含一个 `@` 字符，后跟一个双引号字符、零个或多个字符和一个右双引号字符。一个简单的示例是 `@"hello"` 。在原义字符串文本中，分隔符之间的字符按原义解释，唯一的例外是*quote_escape_sequence*。具体而言，简单转义序列和十六进制和 Unicode 转义序列不会在原义字符串文本中处理。原义字符串文本可以跨多个行。

```
string_literal
    : regular_string_literal
    | verbatim_string_literal
    ;

regular_string_literal
    : '"' regular_string_literal_character* '"'
    ;

regular_string_literal_character
    : single_regular_string_literal_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;

single_regular_string_literal_character
    : '<Any character except " (U+0022), \\ (U+005C), and new_line_character>'
    ;

verbatim_string_literal
    : '@"' verbatim_string_literal_character* '"'
    ;

verbatim_string_literal_character
    : single_verbatim_string_literal_character
    | quote_escape_sequence
    ;

single_verbatim_string_literal_character
    : '<any character except ">'
    ;

quote_escape_sequence
    : '""'
    ;
```

在 regular_string_literal_character 中的反斜杠字符（ `\` ）后面的*regular_string_literal_character*字符必须是下列字符之一： `'` 、 `"` 、 `\` 、 `0` 、 `a` `b` `f` `n` `r` `t` `u` `U` `x` `v` 、、、、、、、、、。否则，将发生编译时错误。

示例

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                 // hello, world

string c = "hello \t world";                // hello      world
string d = @"hello \t world";               // hello \t world

string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";     // Joe said "Hello" to me

string g = "\\\\server\\share\\file.txt";   // \\server\share\file.txt
string h = @"\\server\share\file.txt";      // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

显示各种字符串文本。最后一个字符串文字 `j` 是跨多行的逐字字符串。引号之间的字符(包括空白字符，如换行符)会逐字保留。

由于十六进制转义序列可以具有可变数量的十六进制数字，因此字符串文本 `"\x123"` 包含一个具有十六进制值123的单个字符。若要创建一个字符串，该字符串包含的字符的十六进制值为12，后跟字符3，则可以写 `"\x00123"` `"\x12" + "3"` 。

*String_literal*的类型为 `string` 。

每个字符串文本不一定会生成新的字符串实例。如果两个或更多的字符串文本在同一程序中出现，则根据字符串相等运算符(字符串相等运算符)进行等效时，这些字符串将引用相同的字符串实例。例如，生成的输出

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

是 `True` 因为两个文本引用相同的字符串实例。

内插字符串文本

内插字符串与字符串文本类似，但包含用 and 分隔的孔，其中的 `{` `}` 表达式可以出现。在运行时，将对表达式进行计算，目的是将其文本窗体替换为发生该洞的位置的字符串。字符串内插的语法和语义在节(内插字符串)中进行了介绍。

与字符串文本一样，内插字符串文本可以是正则为或是原义字符串。内插正则字符串文本由 `$"` 和分隔 `"` ，并由和分隔逐字字符串。 `$@"` `"`

与其他文本一样，内插字符串的词法分析最初会根据下面的语法产生单个令牌。但是，在句法分析之前，内插字符串的单个标记将被分解为包含该洞的字符串部分的多个标记，而洞中发生的输入元素会在词法上重新并非。这反过来会生成更多的内插字符串文字，但如果词法上正确，最终将导致一系列标记，以便进行语法分析。

```
interpolated_string_literal
    : '$' interpolated_regular_string_literal
    | '$' interpolated_verbatim_string_literal
    ;

interpolated_regular_string_literal
    : interpolated_regular_string_whole
    | interpolated_regular_string_start  interpolated_regular_string_literal_body
interpolated_regular_string_end
```

```
interpolated_regular_string_end
    ;

interpolated_regular_string_literal_body
    : regular_balanced_text
    | interpolated_regular_string_literal_body interpolated_regular_string_mid regular_balanced_text
    ;

interpolated_regular_string_whole
    : '"' interpolated_regular_string_character* '"'
    ;

interpolated_regular_string_start
    : '"' interpolated_regular_string_character* '{'
    ;

interpolated_regular_string_mid
    : interpolation_format? '}' interpolated_regular_string_characters_after_brace? '{'
    ;

interpolated_regular_string_end
    : interpolation_format? '}' interpolated_regular_string_characters_after_brace? '"'
    ;

interpolated_regular_string_characters_after_brace
    : interpolated_regular_string_character_no_brace
    | interpolated_regular_string_characters_after_brace interpolated_regular_string_character
    ;

interpolated_regular_string_character
    : single_interpolated_regular_string_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_regular_string_character_no_brace
    : '<Any interpolated_regular_string_character except close_brace_escape_sequence and any
hexadecimal_escape_sequence or unicode_escape_sequence designating } (U+007D)>'
    ;

single_interpolated_regular_string_character
    : '<Any character except \" (U+0022), \\ (U+005C), { (U+007B), } (U+007D), and new_line_character>'
    ;

open_brace_escape_sequence
    : '{{'
    ;

close_brace_escape_sequence
    : '}}'
    ;

regular_balanced_text
    : regular_balanced_text_part+
    ;

regular_balanced_text_part
    : single_regular_balanced_text_character
    | delimited_comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' regular_balanced_text ')'
    | '[' regular_balanced_text ']'
    | '{' regular_balanced_text '}'
    ;
```

```
single_regular_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B), } (U+007D) and new_line_character>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;

interpolation_format
    : ':' interpolation_format_character+
    ;

interpolation_format_character
    : '<Any character except \" (U+0022), : (U+003A), { (U+007B) and } (U+007D)>'
    ;

interpolated_verbatim_string_literal
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start interpolated_verbatim_string_literal_body
interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_literal_body
    : verbatim_balanced_text
    | interpolated_verbatim_string_literal_body interpolated_verbatim_string_mid verbatim_balanced_text
    ;

interpolated_verbatim_string_whole
    : '@"' interpolated_verbatim_string_character* '"'
    ;

interpolated_verbatim_string_start
    : '@"' interpolated_verbatim_string_character* '{'
    ;

interpolated_verbatim_string_mid
    : interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '{'
    ;

interpolated_verbatim_string_end
    : interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '"'
    ;

interpolated_verbatim_string_characters_after_brace
    : interpolated_verbatim_string_character_no_brace
    | interpolated_verbatim_string_characters_after_brace interpolated_verbatim_string_character
    ;

interpolated_verbatim_string_character
    : single_interpolated_verbatim_string_character
    | quote_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_verbatim_string_character_no_brace
    : '<Any interpolated_verbatim_string_character except close_brace_escape_sequence>'
    ;

single_interpolated_verbatim_string_character
    : '<Any character except \" (U+0022), { (U+007B) and } (U+007D)>'
    ;

verbatim_balanced_text
    : verbatim_balanced_text_part+
    ;

verbatim_balanced_text_part
    : single_verbatim_balanced_text_character
    | comment
```

```
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' verbatim_balanced_text ')'
    | '[' verbatim_balanced_text ']'
    | '{' verbatim_balanced_text '}'
    ;

single_verbatim_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B) and } (U+007D)>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;
```

*Interpolated_string_literal*令牌重新解释为多个令牌和其他输入元素，如下所示：*interpolated_string_literal*中出现的顺序：

- 以下各项分别作为单独的标记重新解释：前导 `$` 符号、*interpolated_regular_string_whole*、*interpolated_regular_string_start*、*interpolated_regular_string_mid*、*interpolated_regular_string_end*、*interpolated_verbatim_string_whole*、*interpolated_verbatim_string_start*、*interpolated_verbatim_string_mid*和*interpolated_verbatim_string_end*。
- 在这些*regular_balanced_text*和*verbatim_balanced_text*之间发生的情况重新处理为*input_section*（词法分析），并且重新解释为输入元素的结果序列。这些转换可能会将内插字符串文本标记包含为重新解释。

语法分析会将令牌重新组合到*interpolated_string_expression*（内插的字符串）。

示例 TODO

Null 文本

```
null_literal
    : 'null'
    ;
```

*Null_literal*可以隐式转换为引用类型或可以为 null 的类型。

运算符和标点符号

有多种运算符和标点符号。表达式中使用运算符来描述涉及一个或多个操作数的操作。例如，表达式 `a + b` 使用 `+` 运算符添加两个操作数 `a` 和 `b`。标点符号用于分组和分隔。

```
operator_or_punctuator
    : '{'  | '}'  | '['  | ']'  | '('   | ')'  | '.'  | ','  | ':'  | ';'
    | '+'  | '-'  | '*'  | '/'  | '%'   | '&'  | '|'  | '^'  | '!'  | '~'
    | '='  | '<'  | '>'  | '?'  | '??'  | '::' | '++' | '--' | '&&' | '||'
    | '->' | '==' | '!=' | '<=' | '>='  | '+=' | '-=' | '*=' | '/=' | '%='
    | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'
    ;

right_shift
    : '>>'
    ;

right_shift_assignment
    : '>>='
    ;
```

*Right_shift*和*right_shift_assignment*生产中的竖线用于指示，与句法语法中的其他生产不同，标记之间不允许任何类型的字符（甚至不允许使用空格）。将对这些生产进行特殊处理，以便能够正确地处理*type_parameter_list*（类型参数）。

预处理指令

预处理指令提供按条件跳过源文件部分的功能，报告错误和警告条件，以及描述源代码的不同区域。术语 "预处理指令" 仅用于与 C 和 c + + 编程语言的一致性。在 c # 中，没有单独的预处理步骤;预处理指令作为词法分析阶段的一部分进行处理。

```
pp_directive
    : pp_declaration
    | pp_conditional
    | pp_line
    | pp_diagnostic
    | pp_region
    | pp_pragma
    ;
```

以下预处理指令可用：

- `#define` 和 `#undef`，分别用于定义和取消定义条件编译符号（声明指令）。
- `#if`、`#elif`、`#else` 和 `#endif`，用于有条件地跳过源代码的各个部分（条件编译指令）。
- `#line`，用于控制发出的错误和警告的行号（行指令）。
- `#error` 和 `#warning`，分别用于发出错误和警告（诊断指令）。
- `#region` 和 `#endregion`，用于显式标记源代码的各个部分（Region 指令）。
- `#pragma`，用于指定编译器的可选上下文信息（杂注指令）。

预处理指令始终占用一行单独的源代码，并始终以 `#` 字符和预处理指令名称开头。空格可能出现在字符和 `#` `#` 指令名称之间。

包含、、、、、、或指令的源行 `#define` `#undef` `#if` `#elif` `#else` `#endif` `#line` `#endregion` 可以以单行注释结束。`/* */` 在包含预处理指令的源行上不允许使用带分隔符的注释（注释的样式）。

预处理指令不是标记，不是 c # 语法语法的一部分。但是，预处理指令可用于包含或排除标记序列，并以这种方式影响 c # 程序的含义。例如，编译后，程序：

```
#define A
#undef B

class C
{
#if A
    void F() {}
#else
    void G() {}
#endif

#if B
    void H() {}
#else
    void I() {}
#endif
}
```

生成与程序完全相同的标记序列：

```
class C
{
    void F() {}
    void I() {}
}
```

因此，在语义上，这两个程序在语法上非常不同，它们是相同的。

"条件编译符"号

、、和指令提供的条件编译功能 `#if` `#elif` `#else` `#endif` 通过预处理表达式（预处理表达式）和条件编译符号进行控制。

```
conditional_symbol
    : '<Any identifier_or_keyword except true or false>'
    ;
```

条件编译符号有两种可能的状态：***已定义***或***未定义***。在源文件的词法处理开始时，不定义条件编译符号，除非它已由外部机制（例如命令行编译器选项）显式定义。`#define` 处理指令时，该指令中名为的条件编译符号将在该源文件中进行定义。在 `#undef` 处理同一符号的指令之前，或在到达源文件末尾之前，该符号保持为已定义。这意味着，`#define` `#undef` 一个源文件中的和指令对同一程序中的其他源文件不起作用。

在预处理表达式中引用时，定义的条件编译符号具有布尔值 `true` ，未定义的条件编译符号具有布尔值 `false` 。在预处理表达式中引用条件编译符号之前，不需要显式声明它们。相反，未声明的符号只是未定义的，因此具有值 `false` 。

条件编译符号的命名空间是不同的，并且独立于 C# 程序中的所有其他命名实体。条件编译符号只能在 `#define` 和 `#undef` 指令以及预处理表达式中引用。

预处理表达式

预处理表达式可以出现在 `#if` 和指令中 `#elif` 。 `!` `==` `!=` `&&` `||` 预处理表达式中允许使用运算符、、和，括号可用于分组。

```
pp_expression
    : whitespace? pp_or_expression whitespace?
    ;

pp_or_expression
    : pp_and_expression
    | pp_or_expression whitespace? '||' whitespace? pp_and_expression
    ;

pp_and_expression
    : pp_equality_expression
    | pp_and_expression whitespace? '&&' whitespace? pp_equality_expression
    ;

pp_equality_expression
    : pp_unary_expression
    | pp_equality_expression whitespace? '==' whitespace? pp_unary_expression
    | pp_equality_expression whitespace? '!=' whitespace? pp_unary_expression
    ;

pp_unary_expression
    : pp_primary_expression
    | '!' whitespace? pp_unary_expression
    ;

pp_primary_expression
    : 'true'
    | 'false'
    | conditional_symbol
    | '(' whitespace? pp_expression whitespace? ')'
    ;
```

在预处理表达式中引用时，定义的条件编译符号具有布尔值 `true`，未定义的条件编译符号具有布尔值 `false`。

预处理表达式的计算始终产生布尔值。预处理表达式的计算规则与常量表达式的计算规则相同(常数表达式)，只不过只能引用的用户定义实体是条件编译符号。

声明指令

声明指令用于定义或取消定义条件编译符号。

```
pp_declaration
    : whitespace? '#' whitespace? 'define' whitespace conditional_symbol pp_new_line
    | whitespace? '#' whitespace? 'undef' whitespace conditional_symbol pp_new_line
    ;

pp_new_line
    : whitespace? single_line_comment? new_line
    ;
```

指令的处理 `#define` 会使给定的条件编译符号成为定义，并从跟在指令后面的源行开始。同样，处理 `#undef` 指令会使给定的条件编译符号变成未定义的，从该指令后面的源行开始。

`#define` `#undef` 源文件中的任何和指令必须出现在源文件中的第一个*标记*(标记)之前; 否则，将发生编译时错误。在直观的术语中，`#define` 和 `#undef` 指令必须位于源文件中的任何 "真实代码" 之前。

示例：

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

有效，因为 `#define` 指令位于源文件中的第一个标记（ `namespace` 关键字）之前。

下面的示例会导致编译时错误，因为它会 `#define` 跟随真实代码：

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

`#define` 可以定义已定义的条件编译符号，而不会 `#undef` 对该符号进行任何干预。下面的示例定义条件编译符号 `A` ，然后再次定义它。

```
#define A
#define A
```

`#undef` 可能 "取消定义" 未定义的条件编译符号。下面的示例定义条件编译符号 `A` ，然后将其取消定义两次；虽然第二个 `#undef` 不起作用，但仍有效。

```
#define A
#undef A
#undef A
```

条件编译指令

条件编译指令用于有条件地包含或排除源文件的某些部分。

```
pp_conditional
    : pp_if_section pp_elif_section* pp_else_section? pp_endif
    ;

pp_if_section
    : whitespace? '#' whitespace? 'if' whitespace pp_expression pp_new_line conditional_section?
    ;

pp_elif_section
    : whitespace? '#' whitespace? 'elif' whitespace pp_expression pp_new_line conditional_section?
    ;

pp_else_section:
    | whitespace? '#' whitespace? 'else' pp_new_line conditional_section?
    ;

pp_endif
    : whitespace? '#' whitespace? 'endif' pp_new_line
    ;

conditional_section
    : input_section
    | skipped_section
    ;

skipped_section
    : skipped_section_part+
    ;

skipped_section_part
    : skipped_characters? new_line
    | pp_directive
    ;

skipped_characters
    : whitespace? not_number_sign input_character*
    ;

not_number_sign
    : '<Any input_character except #>'
    ;
```

如语法所示，必须按顺序(按顺序)、`#if` 指令、零个或多个 `#elif` 指令、零个或一个 `#else` 指令以及指令来写入条件编译指令 `#endif` 。在指令与源代码的条件部分之间。每个部分都由前面的指令控制。条件部分本身可能包含嵌套的条件编译指令，前提是这些指令构成了完整的集。

*Pp_conditional*最多为常规词法处理选择一个包含的*conditional_section*:

- 和指令的*pp_expression*按 `#if` `#elif` 顺序进行计算，直到有一个结果 `true` 。如果表达式产生了 `true` ，则选择相应指令的*conditional_section* 。
- 如果所有*pp_expression*均 `false` 为 yield，并且 `#else` 存在指令，则选择指令的*conditional_section* `#else` 。
- 否则，不会选择任何*conditional_section* 。

选定的*conditional_section*(如果有)将作为正常*input_section*进行处理:节中包含的源代码必须符合词法语法;标记是从节中的源代码生成的;部分中的和预处理指令具有指定的效果。

剩余的*conditional_section*(如果有)将作为*skipped_section*s 进行处理:除预处理指令以外，部分中的源代码无需遵守词法语法;不会从该部分中的源代码生成任何标记;部分中的和预处理指令必须在词法上是正确的，但不会进行处理。在作为*skipped_section*进行处理的*conditional_section*中，任何嵌套的*conditional_section*(包含在嵌套 `#if` … `#endif` 和 `#region` … 构造中 `#endregion`)也作为*skipped_section*处理。

下面的示例说明了条件编译指令如何嵌套：

```
#define Debug       // Debugging on
#undef Trace        // Tracing off

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}
```

除预处理指令外，跳过的源代码不受词法分析的限制。例如，尽管部分中出现未终止的注释，以下内容仍有效 `#else` ：

```
#define Debug        // Debugging on

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}
```

但请注意，即使在源代码中跳过的部分，预处理指令也需要在词法上正确。

当预处理指令出现在多行输入元素中时，不会对其进行处理。例如，程序：

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
        world
#else
        Nebraska
#endif
        ");
    }
}
```

输出结果为：

```
hello,
#if Debug
        world
#else
        Nebraska
#endif
```

在特殊情况下，处理的预处理指令集可能取决于*pp_expression*的计算。示例：

```
#if X
    /*
#else
    /* */ class Q { }
#endif
```

`class` `Q` `{` `}` 不管是否 `X` 定义了，始终会生成相同的令牌流()。如果 `X` 定义了，则仅处理的指令为 `#if` 和 `#endif` ，因为有多行注释。如果 `X` 未定义，则三个指令(`#if` 、 `#else` 、 `#endif`)是指令集的一部分。

诊断指令

诊断指令用于显式生成错误和警告消息，其报告方式与其他编译时错误和警告的方式相同。

```
pp_diagnostic
    : whitespace? '#' whitespace? 'error' pp_message
    | whitespace? '#' whitespace? 'warning' pp_message
    ;

pp_message
    : new_line
    | whitespace input_character* new_line
    ;
```

示例：

```
#warning Code review needed before check-in

#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

始终产生警告("签入前需要代码评审")，并且如果同时定义了条件符号和，则生成编译时错误("A build 不能同时为调试和零售") `Debug` `Retail` 。请注意， *pp_message*可以包含任意文本;具体而言，它不需要包含格式正确的标记，如单词中的单引号所示 `can't` 。

区域指令

区域指令用于显式标记源代码区域。

```
pp_region
    : pp_start_region conditional_section? pp_end_region
    ;

pp_start_region
    : whitespace? '#' whitespace? 'region' pp_message
    ;

pp_end_region
    : whitespace? '#' whitespace? 'endregion' pp_message
    ;
```

无语义含义附加到区域;区域旨在供程序员或自动工具用来标记源代码的一部分。或指令中指定的 `#region` 消息 `#endregion` 同样没有语义含义; 它仅用于标识区域。匹配 `#region` 的和 `#endregion` 指令可能具有不同的*pp_message*。

区域的词法处理：

```
#region
...
#endregion
```

完全对应于格式为的条件编译指令的词法处理：

```
#if true
...
#endif
```

行指令

行指令可用于更改编译器在输出（如警告和错误）中报告的行号和源文件名，以及由调用方信息特性（调用方信息特性）使用的行号。

行指令最常用于从其他某些文本输入生成 c # 源代码的元编程工具。

```
pp_line
    : whitespace? '#' whitespace? 'line' whitespace line_indicator pp_new_line
    ;

line_indicator
    : decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;

file_name
    : '"' file_name_character+ '"'
    ;

file_name_character
    : '<Any input_character except ">'
    ;
```

当不 `#line` 存在任何指令时，编译器会在其输出中报告真实的行号和源文件名。当处理 `#line` 包含不是的*line_indicator*的指令时，编译器会将 `default` 指令后的行视为具有给定的行号（和文件名，如果指定）。

`#line default` 指令反转所有前面 #line 指令的作用。编译器会报告后续行的真实行信息，就像未 `#line` 处理过指令一样。

`#line hidden` 指令对错误消息中报告的文件和行号没有影响，但会影响源级别调试。调试时， `#line hidden` 指令和后续指令之间的所有行 `#line` （即 not `#line hidden` ）都没有行号信息。单步执行调试器中的代码时，将完全跳过这些行。

请注意，在不处理转义字符的情况下， *file_name*与正则字符串文字不同;" `\` " 字符只是在*file_name*中指定普通反斜杠字符。

Pragma 指令

`#pragma` 预处理指令用于指定编译器的可选上下文信息。指令中提供的信息 `#pragma` 永远不会更改程序语义。

```
pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

C # 提供了 `#pragma` 控制编译器警告的指令。将来版本的语言可能包含其他 `#pragma` 指令。为了确保与其他 c # 编译器的互操作性，Microsoft c # 编译器不会发出未知指令的编译错误 `#pragma` ；因此，此类指令将生成警告。

Pragma warning

`#pragma warning` 指令用于在编译后续程序文本期间禁用或还原所有或一组特定的警告消息。

```
pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

`#pragma warning` 省略警告列表的指令将影响所有警告。 `#pragma warning` 包含警告列表的指令只影响列表中指定的那些警告。

`#pragma warning disable` 指令禁用所有或给定的一组警告。

`#pragma warning restore` 指令将所有或给定的警告集还原到编译单元开头处生效的状态。请注意，如果在外部禁用了特定警告，则 `#pragma warning restore` 将不会重新启用该警告。

下面的示例演示如何使用 `#pragma warning` Microsoft c # 编译器中的警告编号来暂时禁用引用过时成员时所报告的警告。

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
    Foo();
#pragma warning restore 612
    }
}
```

基本概念

2020/11/2 • • Edit Online

应用程序启动

具有***入口点***的程序集称为***应用程序***。当应用程序运行时，将创建一个新的***应用程序域***。同一台计算机上可以同时存在多个不同的应用程序实例，每个实例都有自己的应用程序域。

应用程序域通过充当应用程序状态的容器来实现应用程序隔离。应用程序域充当应用程序中定义的类型和它所使用的类库的容器和边界。加载到一个应用程序域中的类型不同于加载到另一个应用程序域中的相同类型，并且不会在应用程序域之间直接共享对象的实例。例如，每个应用程序域都有自己的静态变量副本用于这些类型，并且每个应用程序域最多运行一次类型的静态构造函数。实现可自由提供特定于实现的策略，也可以创建和销毁应用程序域。

当执行环境调用指定方法(称为应用程序的入口点)时，将发生***应用程序启动***。此入口点方法始终 `Main` 命名，可以具有以下签名之一：

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

如图所示，入口点可能会根据需要返回 `int` 值。此返回值用于应用程序终止(应用程序终止)。

入口点还可以有一个形参。参数可以具有任何名称，但参数的类型必须是 `string[]`。如果正式参数存在，则执行环境将创建并传递一个 `string[]` 参数，该参数包含在启动应用程序时指定的命令行参数。`string[]` 参数决不会为 null，但如果未指定命令行参数，则其长度可能为零。

由于C#支持方法重载，因此类或结构可能包含某些方法的多个定义，前提是每个定义具有不同的签名。但是，在一个程序中，任何类或结构都不能包含多个称为 `Main` 的方法，这些方法的定义将其限定为作为应用程序入口点使用。但允许 `Main` 的其他重载版本，但前提是它们具有多个参数，或者其唯一参数不是类型 `string[]`。

应用程序可以由多个类或结构组成。其中的多个类或结构可能包含一个称为 `Main` 方法，该方法的定义将其限定为作为应用程序入口点使用。在这种情况下，必须使用外部机制(例如命令行编译器选项)来选择其中一个 `Main` 方法作为入口点。

在C#中，每个方法都必须定义为类或结构的成员。通常，方法的声明的可访问性(已声明的可访问性)由其声明中指定的访问修饰符(访问修饰符)确定，并且类似于类型的声明的可访问性由其声明中指定的访问修饰符决定。为了使给定类型的给定方法可调用，该类型和成员都必须可访问。但是，应用程序入口点是一种特殊情况。具体而言，执行环境可以访问应用程序的入口点，无论其声明的可访问性以及其封闭类型声明的声明的可访问性。

应用程序入口点方法不能在泛型类声明中。

在所有其他方面，入口点方法的行为类似于非入口点的行为。

应用程序终止

应用程序终止会向执行环境返回控制权。

如果应用程序的***入口点***方法的返回类型为 `int`，则返回的值将用作应用程序的***终止状态代码***。此代码的目的

是允许向执行环境进行成功或失败的通信。

如果入口点方法的返回类型为 "`void`"，则达到用于终止该方法的右大括号（`}`），或执行没有表达式的 `return` 语句将导致终止状态代码 `0`。

在应用程序终止之前，将调用其所有尚未进行垃圾回收的对象的析构函数，除非已取消此类清理（例如，通过调用库方法 `GC.SuppressFinalize`）。

声明

C#程序中的声明定义程序的构成元素。C#使用命名空间（命名空间）对程序进行组织，其中可以包含类型声明和嵌套命名空间声明。类型声明（类型声明）用于定义类（类）、结构（结构）、接口（接口）、枚举（枚举）和委托（委托）。类型声明中允许的成员种类取决于类型声明的形式。例如，类声明可以包含常量（常量）、字段（字段）、方法（方法）、属性（属性）、事件（事件）、索引器（索引器）、运算符（运算符）、实例构造函数（实例构造函数）、静态构造函数（静态构造函数）、析构函数（析构函数）和嵌套类型（嵌套类型）的声明。

声明定义声明***空间***中声明所属的名称。除重载成员（签名和重载）外，具有两个或多个将在声明空间中引入同名成员的声明是编译时错误。声明空间不能包含具有相同名称的不同类型的成员。例如，声明空间不能包含具有相同名称的字段和方法。

有几种不同类型的声明空间，如下所述。

- 在程序的所有源文件中，不包含封闭*namespace_declaration*的*namespace_member_declaration*是单个组合声明空间的成员，称为***全局声明空间***。
- 在程序的所有源文件中，在具有相同完全限定的命名空间名称的*namespace_declaration*中*namespace_member_declaration*s 是单个组合声明空间的成员。
- 每个类、结构或接口声明都创建新的声明空间。名称通过*class_member_declaration*s、*struct_member_declaration*s、*interface_member_declaration*s 或*type_parameter*引入到此声明空间。除了重载实例构造函数声明和静态构造函数声明，类或结构不能包含与类或结构同名的成员声明。类、结构或接口允许声明重载方法和索引器。此外，类或结构允许声明重载实例构造函数和运算符。例如，类、结构或接口可能包含多个具有相同名称的方法声明，前提是这些方法声明在其签名中不同（签名和重载）。请注意，基类不涉及类的声明空间，并且基接口不涉及接口的声明空间。因此，允许派生类或接口声明与继承成员同名的成员。这种成员被称为*隐藏*继承成员。
- 每个委托声明都将创建一个新的声明空间。通过形参（*fixed_parameter*s 和*parameter_array*s）和*type_parameter*将名称引入到此声明空间。
- 每个枚举声明都将创建一个新的声明空间。通过*enum_member_declaration*s将名称引入此声明空间。
- 每个方法声明、索引器声明、运算符声明、实例构造函数声明和匿名函数都会创建一个名为***局部变量声明空间***的新声明空间。通过形参（*fixed_parameter*s 和*parameter_array*s）和*type_parameter*将名称引入到此声明空间。函数成员或匿名函数（如果有）的主体被视为嵌套在局部变量声明空间内。局部变量声明空间和嵌套局部变量声明空间包含具有相同名称的元素是错误的。因此，在嵌套的声明空间内，不能在封闭声明空间中声明与本地变量或常量同名的局部变量或常数。可能有两个声明空间包含具有相同名称的元素，前提是两个声明空间都不包含另一个。
- 每个*块*或*switch_block*，以及*for*、*foreach*和*using*语句，都将为局部变量和本地常量创建局部变量声明空间。名称通过*local_variable_declaration*s 和*local_constant_declaration*引入到此声明空间。请注意，在函数成员或匿名函数的主体中发生的块嵌套在这些函数为其参数声明的局部变量声明空间内。因此，例如，具有本地变量和同名参数的方法是错误的。
- 每个*块*或*switch_block*都为标签创建一个单独的声明空间。名称通过*labeled_statement*引入到此声明空间中，并且通过*goto_statement*引用这些名称。块的***标签声明空间***包括任何嵌套块。因此，在嵌套块中，不能声明与封闭块中的标签具有相同名称的标签。

声明名称的文本顺序通常不重要。特别是，文本顺序对于命名空间、常量、方法、属性、事件、索引器、运算符、实例构造函数、析构函数、静态构造函数和类型的声明和使用并不重要。声明顺序在以下方面非常重要：

- 字段声明和局部变量声明的声明顺序决定了其初始值设定项（如果有）的执行顺序。

- 局部变量必须在使用之前定义(范围)。
- 省略*constant_expression*值时，枚举成员声明(enum 成员)的声明顺序很重要。

命名空间的声明空间为 "open 已结束"，两个具有相同完全限定名称的命名空间声明会导致相同的声明空间。例如

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

上面的两个命名空间声明对同一声明空间构成，在本例中，用完全限定名称声明两个类 `Megacorp.Data.Customer` 和 `Megacorp.Data.Order`。由于这两个声明涉及到相同的声明空间，因此，如果每个声明都包含具有相同名称的类的声明，则会导致编译时错误。

如上所述，块的声明空间包括任何嵌套块。因此，在下面的示例中，`F` 和 `G` 方法导致编译时错误，因为名称 `i` 是在外部块中声明的，不能在内部块中重新声明。但 `H` 和 `I` 方法是有效的，因为这两个 `i` 在单独的非嵌套块中声明。

```
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();
    }
}
```

Members

命名空间和类型具有*成员*。实体的成员通常通过使用以实体引用开头的限定名称来提供，后跟一个 "`.`" 标记，后跟成员的名称。

类型成员既可以在类型声明中声明，也可以从类型的基类*继承*。当类型从基类继承时，基类的所有成员（实例构造函数、析构函数和静态构造函数除外）将成为派生类型的成员。基类成员的声明的可访问性不会控制是否继承成员，继承扩展到任何不是实例构造函数、静态构造函数或析构函数的成员。但是，在派生的类型中，可能无法访问继承的成员，这可能是由于其声明的可访问性（已声明可访问性）或由类型本身中的声明隐藏（通过继承隐藏）。

命名空间成员

没有封闭命名空间的命名空间和类型是*全局命名空间*的成员。这直接对应于全局声明空间中声明的名称。

命名空间中声明的命名空间和类型是该命名空间的成员。这直接对应于命名空间声明空间中声明的名称。

命名空间没有任何访问限制。不能声明私有、受保护的或内部命名空间，并且命名空间名称始终是可公开访问的。

结构成员

结构的成员是在结构中声明的成员和从该结构的直接基类继承的成员 `System.ValueType` 和间接基类 `object`。

简单类型的成员直接与简单类型化名为的结构类型成员相对应：

- `sbyte` 的成员是 `System.SByte` 结构的成员。
- `byte` 的成员是 `System.Byte` 结构的成员。
- `short` 的成员是 `System.Int16` 结构的成员。
- `ushort` 的成员是 `System.UInt16` 结构的成员。
- `int` 的成员是 `System.Int32` 结构的成员。
- `uint` 的成员是 `System.UInt32` 结构的成员。
- `long` 的成员是 `System.Int64` 结构的成员。
- `ulong` 的成员是 `System.UInt64` 结构的成员。
- `char` 的成员是 `System.Char` 结构的成员。
- `float` 的成员是 `System.Single` 结构的成员。
- `double` 的成员是 `System.Double` 结构的成员。
- `decimal` 的成员是 `System.Decimal` 结构的成员。
- `bool` 的成员是 `System.Boolean` 结构的成员。

枚举成员

枚举的成员是枚举中声明的常量和从枚举的直接基类继承的成员 `System.Enum` 和间接基类 `System.ValueType` 和 `object`。

类成员

类的成员是在类中声明的成员和从基类继承的成员（没有基类的类 `object` 除外）。从基类继承的成员包括常数、字段、方法、属性、事件、索引器、运算符和基类的类型，但不包括基类的实例构造函数、析构函数和静态构造函数。继承基类成员，而不考虑其可访问性。

类声明可以包含常量、字段、方法、属性、事件、索引器、运算符、实例构造函数、析构函数、静态构造函数和类型的声明。

`object` 和 `string` 的成员直接对应于其别名的类类型的成员：

- `object` 的成员是 `System.Object` 类的成员。
- `string` 的成员是 `System.String` 类的成员。

接口成员

接口的成员是在接口和接口的所有基接口中声明的成员。类 `object` 中的成员并不严格地说是任何接口(接口成员)的成员。但是，可以通过成员查找在任何接口类型中(成员查找)使用类 `object` 中的成员。

数组成员

数组成员是继承自类 `System.Array` 的成员。

委托成员

委托的成员是继承自类 `System.Delegate` 的成员。

成员访问

成员的声明允许控制成员访问。成员的可访问性是由成员的声明的可访问性(已声明可访问性)建立的，该成员与直接包含类型的可访问性(如果有)结合使用。

如果允许访问特定成员，则认为该成员是可*访问*的。相反，当不允许访问特定成员时，该成员被称为***不可***访问。如果访问发生的文本位置包含在成员的可访问域(可访问域)中，则允许访问成员。

声明的可访问性

成员的***声明可访问性***可以是以下项之一：

- Public：通过在成员声明中包含 `public` 修饰符来选择。`public` 的直观含义为 "访问不受限制"。
- Protected：通过在成员声明中包含 `protected` 修饰符来选择。`protected` 的直观含义为 "访问限制为包含类或派生自包含类的类型"。
- 内部，通过在成员声明中包含 `internal` 修饰符来选择。`internal` 的直观含义是 "仅限访问此程序"。
- 受保护的内部(即受保护的或内部)，通过在成员声明中包括 `protected` 和 `internal` 修饰符来选择。`protected internal` 的直观含义为 "访问此程序或派生自包含类的类型"。
- Private，通过在成员声明中包含 `private` 修饰符来选择。`private` 的直观含义为 "访问限制为包含类型"。

根据成员声明发生的上下文，仅允许某些类型的声明的可访问性。此外，当成员声明中不包含任何访问修饰符时，在其中进行声明的上下文会确定默认的声明的可访问性。

- 命名空间隐式具有 `public` 声明的可访问性。命名空间声明中不允许使用访问修饰符。
- 在编译单元或命名空间中声明的类型可以具有 `public` 或 `internal` 声明可访问性，并且默认为 `internal` 声明的可访问性。
- 类成员可以具有五种声明的可访问性，并且默认为 `private` 声明的可访问性。(请注意，声明为类成员的类型可以具有五种声明可访问性中的任何一种，而声明为命名空间成员的类型只能具有 `public` 或 `internal` 声明的可访问性。)
- 结构成员可以有 `public`、`internal` 或 `private` 声明的可访问性，并且默认为 `private` 声明的可访问性，因为结构是隐式密封的。结构中引入的结构成员(即，不是由该结构继承)不能具有 `protected` 或 `protected internal` 声明可访问性。(请注意，声明为结构成员的类型可以有 `public`、`internal` 或 `private` 声明的可访问性，而声明为命名空间成员的类型只能具有 `public` 或 `internal` 声明的可访问性。)
- 接口成员隐式具有 `public` 声明的可访问性。接口成员声明中不允许使用访问修饰符。
- 枚举成员隐式具有 `public` 声明的可访问性。枚举成员声明中不允许使用访问修饰符。

辅助功能域

成员的***可访问域***包含允许访问成员的程序文本(可能不连续)部分。出于定义成员的可访问性域的目的，如果成员未在类型内声明，则称为*顶级*成员，如果在另一种类型中声明成员，则称***嵌套***成员。此外，程序的***程序文本***定义为程序的所有源文件中包含的所有程序文本，类型的程序文本定义为该类型的*type_declaration*中包含的所有程序文本(包括在类型中嵌套的类型)。

预定义类型的可访问域(例如 `object`、`int` 或 `double`)不受限制。

在程序 `P` 中声明的顶级未绑定类型 `T`（绑定和未绑定类型）的可访问域定义如下：

- 如果 `public``T` 的声明的可访问性，`T` 的可访问域就是 `P` 的程序文本和引用 `P` 的任何程序。
- 如果 `T` 的已声明可访问性为 `internal`，则 `T` 的可访问域就是 `P` 的程序文本。

从这些定义开始，顶级未绑定类型的可访问域始终至少是声明该类型的程序的程序文本。

构造类型 `T<A1, ..., An>` 的可访问域是未绑定的泛型类型的可访问域的交集 `T` 和 `A1, ..., An` 类型参数的可访问域。

嵌套成员 `M` 的可访问域在程序 `P` 中 `T` 声明的类型中进行了定义，如下所示（请注意，`M` 自身可能是一个类型）：

- 如果 `M` 的已声明可访问性为 `public`，则 `M` 的可访问域就是 `T` 的可访问域。
- 如果 `protected internal``M` 的声明的可访问性，则允许 `D` 是 `P` 的程序文本和从 `T` 派生的任何类型的程序文本的并集（在 `P` 之外声明）。`M` 的可访问域是 `T` 的可访问域与 `D` 的交集。
- 如果 `protected``M` 的声明的可访问性，则允许 `D` 是 `T` 的程序文本和从 `T` 派生的任何类型的程序文本的联合。`M` 的可访问域是 `T` 的可访问域与 `D` 的交集。
- 如果 `M` 的已声明可访问性为 `internal`，则 `M` 的可访问域就是 `T` 的可访问域与 `P` 的程序文本之间的交集。
- 如果 `M` 的已声明可访问性为 `private`，则 `M` 的可访问域就是 `T` 的程序文本。

从这些定义开始，嵌套成员的可访问域始终至少是声明该成员的类型的程序文本。此外，它还会将成员的可访问域与声明该成员的类型的可访问性域完全相同。

直观地说，当访问某个类型或成员 `M` 时，将评估以下步骤以确保允许访问：

- 首先，如果在类型（而不是编译单元或命名空间）中声明 `M`，则当该类型不可访问时，将发生编译时错误。
- 然后，如果 `public``M`，则允许访问。
- 否则，如果 `protected internal``M`，则允许访问，如果它发生在声明了 `M` 的程序中，或发生在派生自类的类 `M` 中，而该类是通过派生类类型（对实例成员的受保护的访问）进行的。
- 否则，如果 `protected``M`，则允许访问，前提是它在声明了 `M` 的类中发生，或发生在派生 `M` 自类的类中，而该类是通过派生类类型（对实例成员的受保护的访问）进行的。
- 否则，如果 `internal``M`，则允许访问，如果在声明 `M` 的程序内发生了访问。
- 否则，如果 `private``M`，则允许访问，如果在声明 `M` 的类型内发生了访问。
- 否则，将无法访问该类型或成员，并发生编译时错误。

示例中

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

类和成员具有以下可访问域：

- `A` 和 `A.X` 的可访问域是不受限制的。
- `A.Y`、`B`、`B.X`、`B.Y`、`B.C`、`B.C.X` 和 `B.C.Y` 的可访问域是包含程序的程序文本。
- `A.Z` 的可访问域是 `A` 的程序文本。
- `B.Z` 和 `B.D` 的可访问域是 `B` 的程序文本，包括 `B.C` 和 `B.D` 的程序文本。
- `B.C.Z` 的可访问域是 `B.C` 的程序文本。
- `B.D.X` 和 `B.D.Y` 的可访问域是 `B` 的程序文本，包括 `B.C` 和 `B.D` 的程序文本。
- `B.D.Z` 的可访问域是 `B.D` 的程序文本。

如示例所示，成员的可访问域绝不会超出包含类型的可访问域。例如，即使所有 `X` 成员都具有公共声明的可访问性，但所有 `A.X` 都具有受包含类型约束的可访问域。

如成员中所述，基类的所有成员（实例构造函数、析构函数和静态构造函数除外）都是由派生类型继承的。这包括甚至是基类的私有成员。但是，私有成员的可访问域只包含声明该成员的类型的程序文本。示例中

```
class A
{
    int x;

    static void F(B b) {
        b.x = 1;        // Ok
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;        // Error, x not accessible
    }
}
```

`B` 类继承 `A` 类中的私有成员 `x`。由于此成员是私有的，因此只能在 `A` 的*class_body*内访问。因此，对 `b.x` 的访问成功在 `A.F` 方法中，但在 `B.F` 方法中失败。

实例成员的受保护访问

当 `protected` 实例成员在声明它的类的程序文本之外访问时，以及当 `protected internal` 实例成员在声明它的程序的程序文本之外访问时，必须在从其声明的类派生的类声明中进行访问。此外，需要通过派生类类型的实例或从其构造的类类型进行访问。此限制可防止一个派生类访问其他派生类的受保护成员，即使这些成员是从同一个基类继承时也是如此。

让 `B` 成为 `M` 声明受保护实例成员的基类，并让 `D` 成为派生自 `B` 的类。在 `D` 的*class_body*中，对 `M` 的访问可以采用以下形式之一：

- `M` 格式的非限定的*type_name*或*primary_expression* 。
- 格式为 `E.M` 的*primary_expression*，前提是 `E` 的类型为 `T` 或派生自 `T` 的类，其中 `T` 是类类型 `D`，或者是从构造的类类型 `D`
- `base.M` 形式的*primary_expression* 。

除这些形式的访问权限外，派生类还可以访问*constructor_initializer*中的基类的受保护实例构造函数（构造函数初始值设定项）。

示例中

```
public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;        // Ok
        b.x = 1;        // Ok
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;        // Error, must access through instance of B
        b.x = 1;        // Ok
    }
}
```

在 `A` 中，可以通过 `A` 和 `B` 的实例访问 `x`，因为在这两种情况下，都可以通过 `A` 实例或从 `A` 派生的类进行访问。但是，在 `B` 中，无法通过 `A` 的实例访问 `x`，因为 `A` 不从 `B` 派生。

示例中

```
class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

允许 `x` 三个赋值，因为它们都是通过从泛型类型构造的类类型的实例进行的。

辅助功能约束

语言中的C#几个构造要求类型必须至少具有与成员或其他类型*一样的可访问性*。如果 `T` 的可访问域是 `M` 的可访问域的超集，则 `M` 类型 `T` 称为 "成员" 或 "类型" 的可访问性。换句话说，如果在可访问 `M` 的所有上下文中都可以访问 `T`，则 `T` 至少可作为 `M` 访问。

存在以下可访问性约束：

- 类类型的直接基类必须至少具有与类类型本身相同的可访问性。
- 接口类型的显式基接口必须至少具有与接口类型本身相同的可访问性。
- 委托类型的返回类型和参数类型必须至少具有与委托类型本身相同的可访问性。
- 常量的类型必须至少具有与常量本身相同的可访问性。
- 字段的类型必须至少具有与字段本身相同的可访问性。
- 方法的返回类型和参数类型必须至少具有与方法本身相同的可访问性。
- 属性的类型必须至少具有与属性本身相同的可访问性。
- 事件的类型必须至少具有与事件本身相同的可访问性。
- 索引器的类型和参数类型必须至少具有与索引器本身相同的可访问性。
- 运算符的类型和参数类型必须至少具有与运算符本身相同的可访问性。
- 实例构造函数的参数类型必须至少具有与实例构造函数本身相同的可访问性。

示例中

```
class A {...}

public class B: A {...}
```

`B` 类会导致编译时错误，因为 `A` 至少与 `B` 一样可访问。

同样，在示例中

```
class A {...}

public class B
{
    A F() {...}

    internal A G() {...}

    public A H() {...}
}
```

`B` 中的 `H` 方法会导致编译时错误，因为 `A` 的返回类型并不至少与方法相同。

签名和重载

方法、实例构造函数、索引器和运算符按其*签名*特征：

- 方法的签名由方法的名称、类型参数的数目以及其每个形参的类型和种类（值、引用或输出）组成，按从左到右的顺序排列。出于这些目的，形参的类型中出现的方法的任何类型形参均由其名称标识，但其在方法的类型实参列表中的序号位置。具体而言，方法的签名不包括返回类型、可以为最右侧参数指定的 `params` 修饰符，也不包括可选的类型参数约束。
- 实例构造函数的签名由其每个形参的类型和类型（值、引用或输出）组成，按从左到右的顺序排列。实例构造函数的签名专门不包含可以为最右侧参数指定的 `params` 修饰符。
- 索引器的签名由其每个形参的类型组成，按从左到右的顺序排列。索引器的签名具体不包括元素类型，也不包括可以为最右侧参数指定的 `params` 修饰符。
- 运算符的签名由运算符的名称和它的每个形参的类型组成，按从左到右的顺序排列。运算符的签名具体不包括结果类型。

签名是用于在类、结构和接口中*重载*成员的启用机制：

- 方法的重载允许类、结构或接口声明多个具有相同名称的方法，前提是它们的签名在该类、结构或接口中是唯一的。
- 实例构造函数的重载允许类或结构声明多个实例构造函数，前提是它们的签名在该类或结构中是唯一的。
- 索引器的重载允许类、结构或接口声明多个索引器，前提是这些索引器的签名在该类、结构或接口中是唯一的。
- 运算符的重载允许类或结构声明多个具有相同名称的运算符，前提是它们的签名在该类或结构中是唯一的。

尽管 `out` 和 `ref` 参数修饰符被视为签名的一部分，但在单个类型中声明的成员不能仅通过 `ref` 和 `out` 在签名中有所不同。如果在同一类型中声明两个成员时，如果两个 `out` 方法中的所有参数都已更改为 `ref` 修饰符，则会发生编译时错误。对于签名匹配的其他目的（例如，隐藏或重写），`ref` 和 `out` 被视为签名的一部分，并且彼此之间不匹配。（这一限制是为了C#使程序可以轻松地翻译为在公共语言基础结构（CLI）上运行，而这并不提供一种方法来定义仅在 `ref` 和 `out` 上不同的方法。）

对于签名，将 `object` 和 `dynamic` 类型视为相同。因此，在一种类型中声明的成员可以不区分签名，只是 `object` 和 `dynamic`。

下面的示例演示一组重载方法声明及其签名。

```
interface ITest
{
    void F();                       // F()

    void F(int x);                  // F(int)

    void F(ref int x);              // F(ref int)

    void F(out int x);              // F(out int)      error

    void F(int x, int y);           // F(int, int)

    int F(string s);                // F(string)

    int F(int x);                   // F(int)          error

    void F(string[] a);             // F(string[])

    void F(params string[] a);      // F(string[])     error
}
```

请注意，任何 `ref` 和 `out` 参数修饰符(方法参数)都是签名的一部分。因此，`F(int)` 和 `F(ref int)` 是唯一的签名。但是，不能在同一接口内声明 `F(ref int)` 和 `F(out int)`，因为它们的签名只是 `ref` 和 `out` 的不同之处。另请注意，返回类型和 `params` 修饰符不是签名的一部分，因此不可能仅基于返回类型或包含或排除 `params` 修饰符而重载。同样，上面标识的方法的声明 `F(int)` 和 `F(params string[])` 会导致编译时错误。

范围

名称的***作用域***是程序文本的区域，在该区域中，可以引用名称声明的实体，而无需限定名称。可以***嵌套***作用域，内部作用域可以重新声明外部作用域中的名称的含义(但这并不能删除嵌套块内的声明施加的限制，不能声明与封闭块中的局部变量同名的局部变量)。然后，在内部范围所涵盖的程序文本区域中*隐藏*外部范围的名称，并且只能通过限定名称来访问外部名称。

- 由不包含任何封闭*namespace_declaration*的*namespace_member_declaration*（命名空间成员）声明的命名空间成员的作用域是整个程序文本。
- 由完全限定名称 `N` 的*namespace_declaration*中*namespace_member_declaration*声明的命名空间成员的范围是每个*namespace_declaration*的*namespace_body*，其完全限定名称是 `N` 或以 `N` 开头，后跟一个句点。
- *Extern_alias_directive*定义的名称的作用域超出了其立即包含编译单元或命名空间正文的*using_directive*s、*global_attributes*和*namespace_member_declaration*。*Extern_alias_directive*不会将任何新成员分配给基础声明空间。换句话说，*extern_alias_directive*是不可传递的，而是只影响它发生的编译单元或命名空间体。
- *Using_directive*（使用指令）定义或导入的名称的作用域扩展到发生*using_directive*的*compilation_unit*或*namespace_body*的*namespace_member_declaration*。*Using_directive*可以使零个或多个命名空间、类型或成员名称在特定*compilation_unit*或*namespace_body*中可用，但不会将任何新成员分配给基础声明空间。换句话说，*using_directive*是不可传递的，而只会影响其发生的*compilation_unit*或*namespace_body*。
- *Class_declaration*（类声明）上的*type_parameter_list*声明的类型参数的作用域是该*class_body*的*class_base*、*type_parameter_constraints_clause*和*class_declaration*。
- *Struct_declaration*（结构声明）上的*type_parameter_list*声明的类型参数的作用域是该*struct_body*的*struct_interfaces*、*type_parameter_constraints_clause*和*struct_declaration*。
- *Interface_declaration*（接口声明）上的*type_parameter_list*声明的类型参数的作用域是该*interface_body*的*interface_base*、*type_parameter_constraints_clause*和*interface_declaration*。
- 由*delegate_declaration*（委托声明）上的*type_parameter_list*声明的类型参数的作用域是该*type_parameter_constraints_clause*的*return_type*、*formal_parameter_list*和*delegate_declaration*。
- *Class_member_declaration*（类体）声明的成员的作用域是在其中进行声明的*class_body*。此外，类成员的作用域扩展到包含在该成员的可访问域（可访问域）中的派生类的*class_body*。

- *Struct_member_declaration*（结构成员）声明的成员的作用域是在其中进行声明的*struct_body*。
- *Enum_member_declaration*（枚举成员）声明的成员的作用域是在其中进行声明的*enum_body*。
- 在*method_declaration*（方法）中声明的参数的作用域是该*method_declaration*的*method_body*。
- *Indexer_declaration*（索引器）中声明的参数的作用域是该*indexer_declaration*的*accessor_declarations*。
- 在*operator_declaration*（运算符）中声明的参数的作用域是该*operator_declaration*的*块*。
- 在*constructor_declaration*（实例构造函数）中声明的参数的作用域是该*constructor_declaration*的*constructor_initializer*和*块*。
- 在*lambda_expression*（匿名函数表达式）中声明的参数的作用域是该*lambda_expression*的*anonymous_function_body*
- 在*anonymous_method_expression*（匿名函数表达式）中声明的参数的作用域是该*anonymous_method_expression*的*块*。
- 在*labeled_statement*（标记语句）中声明的标签的作用域是在其中进行声明的*块*。
- 在*local_variable_declaration*（局部变量声明）中声明的局部变量的作用域是在其中进行声明的块。
- 在 `switch` 语句的*switch_block*中声明的局部变量的作用域（switch 语句）为*switch_block*。
- 在 `for` 语句的*for_initializer*中声明的局部变量的作用域（for 语句）是*for_initializer*、*for_condition*、*for_iterator*和 `for` 语句包含的*语句*。
- 在*local_constant_declaration*中声明的局部常量的作用域（局部常量声明）是在其中进行声明的块。在*constant_declarator*之前的文本位置引用本地常量是编译时错误。
- 声明为*foreach_statement*、*using_statement*、*lock_statement*或*query_expression*一部分的变量的作用域由给定构造的扩展确定。

在命名空间、类、结构或枚举成员的作用域内，可以引用成员声明之前的文本位置中的成员。例如

```
class A
{
    void F() {
        i = 1;
    }

    int i = 0;
}
```

此处，在声明之前，`F` 引用 `i` 是有效的。

在局部变量的作用域内，在本地变量*local_variable_declarator*之前的文本位置引用本地变量是编译时错误。例如

```
class A
{
    int i = 0;

    void F() {
        i = 1;                  // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = (j = 1);        // Valid
    }

    void H() {
        int a = 1, b = ++a;     // Valid
    }
}
```

在上面的 `F` 方法中，第一次分配到 `i` 并不引用在外部作用域中声明的字段。相反，它是指局部变量，它会导致

编译时错误，因为它在此变量的声明之前。在 `G` 方法中，在 `j` 的声明的初始值设定项中使用 `j` 是有效的，因为在*local_variable_declarator*之前不会使用。在 `H` 方法中，后续*local_variable_declarator*正确引用在同一*local_variable_declaration*中的早期*local_variable_declarator*中声明的局部变量。

局部变量的作用域规则旨在确保表达式上下文中使用的名称的含义在块中始终相同。如果本地变量的作用域只是从其声明扩展到块的末尾，则第一个赋值将分配给实例变量，第二个赋值将分配给局部变量，这可能会导致编译时错误（如果以后要重新排列块的语句）。

块内的名称含义可能因使用该名称的上下文而异。示例中

```
using System;

class A {}

class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                           // expression context

        Type t = typeof(A);                     // type context

        Console.WriteLine(s);                   // writes "hello, world"
        Console.WriteLine(t);                   // writes "A"
    }
}
```

名称 `A` 用于在表达式上下文中引用局部变量 `A`，在类型上下文中用于引用类 `A`。

名称隐藏

实体的作用域通常比实体的声明空间包含更多的程序文本。具体而言，实体的范围可能包括引入新声明空间的声明，这些声明空间包含具有相同名称的实体。此类声明会使原始实体处于***隐藏状态***。相反，如果实体未隐藏，则称该实体***可见***。

如果范围通过嵌套重叠，并且作用域通过继承重叠，则会发生名称隐藏。以下各节将介绍这两种隐藏类型的特征。

通过嵌套隐藏

由于嵌套命名空间或命名空间中的类型、类或结构中的嵌套类型以及参数和局部变量声明的结果，因此可以通过嵌套命名空间。

示例中

```
class A
{
    int i = 0;

    void F() {
        int i = 1;
    }

    void G() {
        i = 1;
    }
}
```

在 `F` 方法中，实例变量 `i` 由局部变量 `i` 隐藏，但在 `G` 方法中，`i` 仍引用实例变量。

当内部范围中的名称隐藏外部范围内的名称时，它将隐藏该名称的所有重载匹配项。示例中

```
class Outer
{
    static void F(int i) {}

    static void F(string s) {}

    class Inner
    {
        void G() {
            F(1);              // Invokes Outer.Inner.F
            F("Hello");        // Error
        }

        static void F(long l) {}
    }
}
```

调用 `F(1)` 调用在 `Inner` 中声明的 `F`，因为 `F` 的所有外部匹配项都将被内部声明隐藏。出于同一原因，调用 `F("Hello")` 会导致编译时错误。

通过继承隐藏

当类或结构重新声明从基类继承的名称时，通过继承进行名称隐藏。此类名称隐藏采用以下形式之一：

- 类或结构中引入的常数、字段、属性、事件或类型隐藏了具有相同名称的所有基类成员。
- 类或结构中引入的方法会隐藏所有具有相同名称的非方法基类成员，并隐藏具有相同签名的所有基类方法（方法名称和参数计数、修饰符和类型）。
- 类或结构中引入的索引器会隐藏具有相同签名的所有基类索引器（参数计数和类型）。

控制运算符声明（运算符）的规则使得派生类不可能使用与基类中的运算符相同的签名来声明运算符。因此，运算符永远不会彼此隐藏。

与隐藏外部作用域中的名称相反，从继承的作用域中隐藏可访问的名称会导致报告警告。示例中

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    public void F() {}        // Warning, hiding an inherited name
}
```

`Derived` 中 `F` 的声明会导致报告警告。隐藏继承名称特别不是错误，因为这会阻止基类的单独演化。例如，由于 `Base` 的更高版本引入了在早期版本的类中不存在的 `F` 方法，因此可能会出现上述情况。如果上述情况是错误的，则对单独的版本控制类库中的基类进行的任何更改都可能导致派生类无效。

隐藏继承名称导致的警告可以通过使用 `new` 修饰符来消除：

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}
```

`new` 修饰符指示 `Derived` 中的 `F` 为 "new"，并且实际上用于隐藏继承的成员。

新成员的声明仅在新成员的范围内隐藏继承的成员。

```
class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {}    // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }          // Invokes Base.F
}
```

在上面的示例中，`Derived` 中的 `F` 的声明隐藏了继承自 `Base` 的 `F`，但由于 `F` 中的新 `Derived` 具有私有访问权限，因此它的作用域不会扩展到 `MoreDerived`。因此，`MoreDerived.G` 中 `F()` 调用有效，并将调用 `Base.F`。

命名空间和类型名称

程序中的多个上下文需要指定*namespace_name*或*type_name*。C#

```
namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;
```

*Namespace_name*是引用命名空间的*namespace_or_type_name* 。按照下面所述的解决方法，*namespace_name*的*namespace_or_type_name*必须引用命名空间，否则将发生编译时错误。*Namespace_name*中不能存在类型参数（类型参数）（只有类型可以具有类型参数）。

*Type_name*是引用类型的*namespace_or_type_name* 。按照下面所述的解决方法，*type_name*的*namespace_or_type_name*必须引用类型，否则将发生编译时错误。

如果*namespace_or_type_name*是符合条件的成员，则其含义如命名空间别名限定符中所述。否则，*namespace_or_type_name*具有以下四种形式之一：

- `I`
- `I<A1, ..., Ak>`
- `N.I`
- `N.I<A1, ..., Ak>`

其中 `I` 是单个标识符，`N` 为*namespace_or_type_name*，`<A1, ..., Ak>` 为可选的*type_argument_list*。当未指定*type_argument_list*时，请考虑 `k` 为零。

确定*namespace_or_type_name*的含义，如下所示：

- 如果*namespace_or_type_name*格式为 `I` 或 `I<A1, ..., Ak>` 格式：
  - 如果 `K` 为零且*namespace_or_type_name*出现在泛型方法声明(方法)中，并且如果该声明包含名称为 `I` 的类型参数(类型参数)，则*namespace_or_type_name*将引用该类型参数。
  - 否则，如果*namespace_or_type_name*出现在类型声明内，则对于每个实例类型 `T` (实例类型)，以该类型声明的实例类型开头，并继续使用每个封闭类或结构声明的实例类型(如果有)：
    - 如果 `K` 为零，且 `T` 的声明包含名称为 `I` 的类型参数，则*namespace_or_type_name*将引用该类型参数。
    - 否则，如果*namespace_or_type_name*出现在类型声明的主体中，并且 `T` 或其任何基类型包含名称 `I` 和 `K` 类型参数的嵌套可访问类型，则*namespace_or_type_name*引用使用给定类型参数构造的类型。如果有多个这样的类型，则选择在派生程度较大的类型中声明的类型。请注意，在确定*namespace_or_type_name*的含义时，将忽略非类型成员(常数、字段、方法、属性、索引器、运算符、实例构造函数、析构函数和静态构造函数)和具有不同数目的类型参数的类型成员。
  - 如果前面的步骤不成功，则对于每个命名空间 `N`，从发生*namespace_or_type_name*的命名空间开始，继续每个封闭命名空间(如果有)，并以全局命名空间结束，将计算以下步骤，直到找到实体：
    - 如果 `K` 为零且 `I` 是 `N` 中的命名空间的名称，则：
      - 如果*namespace_or_type_name*发生的位置由 `N` 的命名空间声明括起来，并且命名空间声明包含将名称 `I` 与命名空间或类型相关联的*extern_alias_directive*或*using_alias_directive*，则*namespace_or_type_name*为不明确，并发生编译时错误。
      - 否则，*namespace_or_type_name*引用 `N` 中名为 `I` 的命名空间。
    - 否则，如果 `N` 包含名称 `I` 且 `K` 类型参数的可访问类型，则：
      - 如果 `K` 为零，且*namespace_or_type_name*的位置由 `N` 的命名空间声明括起来，并且命名空间声明包含将名称 using_alias_directive 与命名空间或类型相关联的*extern_alias_directive*或 * `I` *，则*namespace_or_type_name*是不明确的，并发生编译时错误。
      - 否则，*namespace_or_type_name*引用用给定类型参数构造的类型。
    - 否则，如果*namespace_or_type_name*发生的位置由 `N` 的命名空间声明括起来：
      - 如果 `K` 为零，且命名空间声明包含将名称 `I` 与导入的命名空间或类型相关联的*extern_alias_directive*或*using_alias_directive*，则*namespace_or_type_name*引用该命名空间或类型。
      - 否则，如果*using_namespace_directive*s 和*using_alias_directive*s 的命名空间和类型声明都包含一个名称 `I` 并 `K` 类型参数的可访问类型，则*namespace_or_type_name*引用使用给定类型参数构造的类型。
      - 否则，如果*using_namespace_directive*s 和命名空间声明的*using_alias_directive*s 中导入的命名空间和类型声明包含多个具有名称 `I` 并 `K` 类型参数的可访问类型，则*namespace_or_type_name*为不明确，并发生错误。
    - 否则，*namespace_or_type_name*是未定义的，并且发生编译时错误。
- 否则，*namespace_or_type_name*的格式为 `N.I` 或 `N.I<A1, ..., Ak>` 形式。`N` 首先解析为*namespace_or_type_name*。如果 `N` 的解析不成功，则会发生编译时错误。否则，按如下方式解析 `N.I` 或 `N.I<A1, ..., Ak>`：
  - 如果 `K` 为零且 `N` 引用命名空间，并且 `N` 包含名称 `I` 的嵌套命名空间，则*namespace_or_type_name*引用该嵌套命名空间。
  - 否则，如果 `N` 引用命名空间，并且 `N` 包含名称 `I` 和 `K` 类型参数的可访问类型，则*namespace_or_type_name*引用使用给定类型参数构造的类型。
  - 否则，如果 `N` 引用一个(可能构造的)类或结构类型，并且 `N` 或它的任何一个基类包含名称 `I` 和 `K` 类型参数的嵌套可访问类型，则*namespace_or_type_name*引用使用给定类型参数构造的类型。如果有多个这样的类型，则选择在派生程度较大的类型中声明的类型。请注意，如果将 `N.I` 的含义确定为

解析 `N` 的基类规范的一部分，则 `N` 的直接基类被视为 object（基类）。

- 否则，`N.I` 是无效*namespace_or_type_name*，并发生编译时错误。

只允许*namespace_or_type_name*引用静态类（静态类）

- *Namespace_or_type_name*是窗体 `T.I` *namespace_or_type_name*的 `T`，或
- *Namespace_or_type_name*是窗体 `typeof(T)` 的*typeof_expression*（参数列表1）中的 `T`。

完全限定的名称

每个命名空间和类型都有一个***完全限定的名称***，该名称唯一地标识命名空间或类型。命名空间或类型 `N` 的完全限定名的确定方式如下：

- 如果 `N` 是全局命名空间的成员，则会 `N` 其完全限定名称。
- 否则，其完全限定名称为 `S.N`，其中 `S` 是声明 `N` 的命名空间或类型的完全限定名称。

换句话说，`N` 的完全限定名是从全局命名空间开始 `N` 的标识符的完整层次结构路径。由于命名空间或类型的每个成员都必须具有唯一的名称，因此命名空间或类型的完全限定名称始终是唯一的。

下面的示例显示了几个命名空间和类型声明及其关联的完全限定名称。

```
class A {}                // A

namespace X               // X
{
    class B               // X.B
    {
        class C {}        // X.B.C
    }

    namespace Y           // X.Y
    {
        class D {}        // X.Y.D
    }
}

namespace X.Y             // X.Y
{
    class E {}            // X.Y.E
}
```

自动内存管理

C#采用自动内存管理，使开发人员无手动分配和释放由对象占用的内存。自动内存管理策略是由***垃圾回收器***实现的。对象的内存管理生命周期如下所示：

1. 创建对象时，将为其分配内存，运行构造函数，并将对象视为实时对象。
2. 如果对象或它的任何部分不能通过任何可能的执行继续进行访问，而不是运行析构函数，则该对象被视为不再使用，并可用于销毁。C#编译器和垃圾回收器可以选择对代码进行分析，以确定将来可能会使用哪些对对象的引用。例如，如果作用域中的局部变量是唯一的对象引用，但在过程中从当前执行点到执行的任何可能的继续，则不会引用该局部变量，垃圾回收器可能（但不是需要）将对象视为不再使用。
3. 对象符合析构条件后，在某些以后未指定的情况下，将运行该对象的析构函数（如果有）。正常情况下，对象的析构函数只运行一次，不过实现特定的 Api 可能会允许重写此行为。
4. 一旦运行某个对象的析构函数，如果该对象或它的任何部分不能通过任何可能的执行继续进行访问，包括运行析构函数，则该对象将被视为不可访问，并且该对象便可用于收集。
5. 最后，在对象变为符合集合条件后，垃圾回收器将释放与该对象关联的内存。

垃圾回收器维护有关对象使用情况的信息，并使用此信息进行内存管理决策，如在内存中查找新创建的对象的位

置、何时重新定位对象以及对象不再使用或无法访问。

与其他假设存在垃圾回收器的语言一样，C#设计使垃圾回收器可以实现各种内存管理策略。例如，C#不要求运行析构函数，或在对象符合条件时收集对象，或者在任何特定的线程上以任何特定的顺序运行析构函数。

可以通过类 `System.GC` 上的静态方法控制垃圾回收器的行为。此类可用于请求进行集合、运行（或不运行）等操作。

由于垃圾回收器允许广泛的纬度来决定何时收集对象和运行析构函数，因此一致的实现可能产生与下面的代码所示不同的输出。程序

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;

    public B(object o) {
        Ref = o;
    }

    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

创建类 `A` 实例和类 `B` 的实例。如果为变量 `b` 分配了值 `null`，则这些对象将可以进行垃圾回收，因为在此之后，任何用户编写的代码就不能访问它们。输出可以是

```
Destruct instance of A
Destruct instance of B
```

或

```
Destruct instance of B
Destruct instance of A
```

因为该语言对对象的垃圾回收顺序没有约束。

在微妙的情况下，"符合析构条件" 和 "符合集合条件" 之间的区别非常重要。例如，应用于对象的

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}

class B
{
    public A Ref;

    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}
```

在上述程序中，如果垃圾回收器选择在 `B` 的析构函数之前运行 `A` 的析构函数，则该程序的输出可能是：

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

请注意，尽管 `A` 的实例未被使用，并且 `A` 的析构函数已运行，但仍有可能从另一个析构函数调用 `A` 的方法（在本例中为 `F`）。另外，请注意，运行析构函数可能会导致对象再次可从主线程序使用。在这种情况下，运行 `B` 的析构函数会导致先前未使用的 `A` 实例成为可从实时引用 `Test.RefA` 访问的实例。调用 `WaitForPendingFinalizers` 后，`B` 的实例可用于集合，但 `A` 的实例不是，因为引用 `Test.RefA`。

为了避免混淆和意外行为，通常情况下，析构函数只对存储在其自身字段中的数据执行清理，而不对引用的对象或静态字段执行任何操作。

使用析构函数的替代方法是让类实现 `System.IDisposable` 接口。这允许对象的客户端确定何时释放对象的资

源，通常是通过在 `using` 语句（using 语句）中将对象作为资源来访问。

执行顺序

C#程序的执行将继续，使每个正在执行的线程的副作用被保留在关键执行点上。***副作用***定义为可变字段的读取或写入、对非易失性变量的写入、写入外部资源和引发异常。必须保留这些副作用的顺序的关键执行点是对可变字段的引用（可变字段）、`lock` 语句（lock 语句）以及线程创建和终止。执行环境可随意更改C#程序的执行顺序，但要服从以下限制：

- 数据依赖关系在执行线程中保留。也就是说，将计算每个变量的值，就好像该线程中的所有语句都是按原程序顺序执行的一样。
- 保留初始化排序规则（字段初始化和变量初始值设定项）。
- 对于易失读写（可变字段），会保留副作用的顺序。此外，如果执行环境可以推断出未使用表达式的值并且不会产生所需的副作用（包括通过调用方法或访问可变字段引起的任何副作用），则执行环境不需要计算表达式的一部分。当异步事件（如由另一个线程引发的异常）中断程序执行时，不保证可观察到的副作用在原始程序顺序中可见。

类型

2020/11/2 • • Edit Online

C#语言的类型分为两个主要类别：*值类型*和*引用类型*。值类型和引用类型都可以是采用一个或多个*类型参数*的*泛型类型*。类型参数可以同时指定值类型和引用类型。

```
type
    : value_type
    | reference_type
    | type_parameter
    | type_unsafe
    ;
```

类型（指针）的最终类别仅在不安全代码中提供。指针类型中对此进行了进一步讨论。

值类型在值类型的变量中直接包含其数据时不同于引用类型，而引用类型的变量存储对其数据的*引用*，后者被称为*对象*。对于引用类型，两个变量可以引用同一对象，因此，对一个变量执行的运算可能会影响另一个变量所引用的对象。对于值类型，每个变量都有自己的数据副本，对一个变量执行的操作不可能影响另一个。

C#的类型系统是统一的，因此任何类型的值都可以被视为对象。每种 C# 类型都直接或间接地派生自 `object` 类类型，而 `object` 是所有类型的最终基类。只需将值视为类型 `object`，即可将引用类型的值视为对象。通过执行装箱和取消装箱操作（装箱和取消装箱），值类型的值被视为对象。

值类型

值类型是结构类型或枚举类型。C#提供一组称为*简单类型*的预定义的结构类型。简单类型通过保留字进行标识。

```
value_type
    : struct_type
    | enum_type
    ;

struct_type
    : type_name
    | simple_type
    | nullable_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

nullable_type
    : non_nullable_value_type '?'
    ;

non_nullable_value_type
    : type
    ;

enum_type
    : type_name
    ;
```

与引用类型的变量不同，值类型的变量仅在值类型是可以为 null 的类型时才可以包含值 `null`。对于每个不可为 null 的值类型，都有一个对应的可以为 null 的值类型，用于表示相同的一组值加上 `null` 的值。

对值类型的变量赋值会创建要分配的值的副本。这不同于对引用类型的变量的赋值，后者复制引用，而不是由引用标识的对象。

System.object 类型

所有值类型都隐式继承自类 `System.ValueType`，而该类又从类 `object` 继承。任何类型都不能从值类型派生，因此值类型将隐式密封（密封类）。

请注意，`System.ValueType` 本身并不 *value_type*。相反，它是一个 *class_type*，所有 *value_type* 都自动派生。

默认构造函数

所有值类型都隐式声明称为***默认构造函数***的公共无参数实例构造函数。默认构造函数返回一个名为的零初始化实例，该实例称为 "值类型" 的***默认值***：

- 对于所有*simple_type*，默认值是由所有零的位模式生成的值：
  - 对于 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long` 和 `ulong`，默认值为 `0`。
  - 对于 `char`，默认值为 "`'\x0000'`"。
  - 对于 `float`，默认值为 "`0.0f`"。
  - 对于 `double`，默认值为 "`0.0d`"。
  - 对于 `decimal`，默认值为 "`0.0m`"。
  - 对于 `bool`，默认值为 "`false`"。
- 对于为 `E`，默认值为 `0`，转换为类型 `E`。
- 对于*struct_type*，默认值是通过将所有值类型的字段设置为其默认值，并将所有引用类型字段设置为 `null` 生成的值。
- 对于*nullable_type*，默认值为 `HasValue` 属性为 false 且 `Value` 属性未定义的实例。默认值也称为可以为 null 的类型的***null 值***。

与任何其他实例构造函数一样，值类型的默认构造函数使用 `new` 运算符进行调用。出于效率方面的考虑，此要求实际上不会使实现生成构造函数调用。在下面的示例中，变量 `i` 和 `j` 都初始化为零。

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

由于每个值类型都隐式具有一个公共的无参数实例构造函数，因此结构类型不可能包含无参数构造函数的显式声明。但允许结构类型声明参数化实例构造函数（构造函数）。

结构类型

结构类型是一种值类型，可以声明常量、字段、方法、属性、索引器、运算符、实例构造函数、静态构造函数和嵌套类型。结构声明中描述了结构类型的声明。

简单类型

C#提供一组称为*简单类型*的预定义的结构类型。简单类型通过保留字标识，但是这些保留字只是 `System` 命名空间中预定义的结构类型的别名，如下表中所述。

IIII	IIIII
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`

[[[	[[[[
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

因为简单类型是结构类型的别名，所以每个简单类型都具有成员。例如，`int` 在 `System.Int32` 中声明了成员，并且从 `System.Object` 继承了成员，并且允许以下语句：

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();        // System.Int32.ToString() instance method
string t = 123.ToString();      // System.Int32.ToString() instance method
```

简单类型不同于其他结构类型，简单类型允许某些附加操作：

- 大多数简单类型允许通过编写文本(文本)来创建值。例如，`123` 是类型 `int` 的文字，`'a'` 是 `char` 类型的文字。C#通常不会对结构类型的文本进行任何设置，并且最终总是通过这些结构类型的实例构造函数创建其他结构类型的非默认值。
- 如果表达式的操作数都是简单类型常量，则编译器可能会在编译时计算表达式。此类表达式称为*constant_expression*（常量表达式）。涉及其他结构类型定义的运算符的表达式不会被视为常量表达式。
- 通过 `const` 声明，可以声明简单类型（常量）的常量。不能有其他结构类型的常量，但 `static readonly` 字段提供了类似的效果。
- 涉及简单类型的转换可以参与对由其他结构类型定义的转换运算符的计算，但用户定义的转换运算符永远不能参与对另一个用户定义的运算符的求值（计算为用户定义的转换）。

整型

C#支持9整型类型：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong` 和 `char`。整数类型具有以下大小和值范围：

- `sbyte` 类型表示有符号的8位整数，其值介于-128 和127之间。
- `byte` 类型表示值介于0到255之间的无符号8位整数。
- `short` 类型表示有符号的16位整数，其值介于-32768 和32767之间。
- `ushort` 类型表示值介于0到65535之间的16位无符号整数。
- `int` 类型表示有符号32位整数，其值介于-2147483648 和2147483647之间。
- `uint` 类型表示无符号32位整数，其值介于0和4294967295之间。
- `long` 类型表示有符号64位整数，其值介于-9223372036854775808 和9223372036854775807之间。
- `ulong` 类型表示值介于0和18446744073709551615之间的未签名64位整数。
- `char` 类型表示值介于0到65535之间的16位无符号整数。`char` 类型的可能值集对应于 Unicode 字符集。尽管 `char` 与 `ushort` 具有相同的表示形式，但并不允许对一种类型允许的所有操作。

整型一元运算符和二元运算符始终操作有符号32位精度、无符号32位精度、有符号64位精度或无符号的64位精

度：

- 对于一元 `+` 和 `~` 运算符，操作数转换为类型 `T`，其中 `T` 是 `int`、`uint`、`long` 和 `ulong` 中的第一个，可以完全表示操作数的所有可能值。然后，使用 `T` 类型的精度执行运算，并 `T` 结果的类型。
- 对于一元 `-` 运算符，操作数转换为类型 `T`，其中 `T` 是 `int` 和 `long` 中的第一个，可以完全表示操作数的所有可能值。然后，使用 `T` 类型的精度执行运算，并 `T` 结果的类型。一元 `-` 运算符不能应用于 `ulong` 类型的操作数。
- 对于二进制 `+`，`-`、`*`、`/`、`%`、`&`、`^`、`|`、`==`、`!=`、`>`、`<`、`>=` 和 `<=` 运算符，操作数将转换为类型 `T`，其中 `T` 是 `int`、`uint`、`long` 和 `ulong` 中的第一个，可以完全表示这两个操作数的所有可能值。然后，使用 `T` 类型的精度执行该操作，结果的类型为 `T`（或关系运算符的 `bool`）。不允许一个操作数的类型为 `long`，另一个操作数的类型 `ulong` 为二元运算符。
- 对于二进制 `<<` 和 `>>` 运算符，左操作数转换为类型 `T`，其中 `T` 是 `int`、`uint`、`long` 和 `ulong` 中的第一个，可以完全表示操作数的所有可能值。然后，使用 `T` 类型的精度执行运算，并 `T` 结果的类型。

`char` 类型分类为整型类型，但它在两个方面与其他整型类型不同：

- 无法将其他类型隐式转换为 `char` 类型。具体而言，即使 `sbyte`、`byte` 和 `ushort` 类型都具有使用 `char` 类型完全表示的值范围，从 `sbyte`、`byte` 或 `ushort` 到 `char` 的隐式转换不存在。
- 必须将 `char` 类型的常量编写为*character_literals* 或*integer_literals*，并将强制转换为类型 `char`。例如，`(char)10` 和 `'\x000A'` 相同。

`checked` 和 `unchecked` 运算符和语句用于控制整型算术运算和转换的溢出检查（选中和未选中的运算符）。在 `checked` 上下文中，溢出会生成编译时错误或引发 `System.OverflowException`。在 `unchecked` 上下文中，将忽略溢出，并且不会丢弃任何不适合目标类型的高序位。

浮点类型

C#支持两种浮点类型：`float` 和 `double`。`float` 和 `double` 类型使用32位单精度和64位双精度 IEEE 754 格式表示，这些格式提供以下值集：

- 正零和负零。在大多数情况下，正零和负零的行为与简单值零的行为相同，但某些运算区分这两个（除法运算符）。
- 正无穷大和负无穷大。此类操作会产生无穷大，将非零数除以零。例如，`1.0 / 0.0` 产生正无穷，`-1.0 / 0.0` 产生负无穷。
- ***非数字***值，通常缩写为 NaN。Nan 由无效的浮点运算生成，如将零除以零。
- `s * m * 2^e` 形式的非零的有限值集，其中 `s` 为1或-1，`m` 和 `e` 由特定浮点类型确定：对于 `float`、`0 < m < 2^24` 和 `-149 <= e <= 104` 对于 `double`，`0 < m < 2^53` 和 `-1075 <= e <= 970`。非标准化浮点数被视为有效的非零值。

`float` 类型可以表示从大约 `1.5 * 10^-45` 到 `3.4 * 10^38` 精度为7位的值。

`double` 类型可以表示从大约 `5.0 * 10^-324` 到 `1.7 × 10^308` 精度为15-16 位的值。

如果二元运算符的一个操作数为浮点类型，则另一个操作数必须为整型类型或浮点类型，并且运算将按如下方式计算：

- 如果其中一个操作数为整型类型，则该操作数将转换为另一个操作数的浮点类型。
- 然后，如果任意一个操作数为 `double` 类型，则另一个操作数将转换为 `double`，则使用至少 `double` 范围和精度执行操作，并且结果的类型为 `double`（或关系运算符的 `bool`）。
- 否则，使用至少 `float` 范围和精度执行操作，并且结果的类型为 `float`（或关系运算符的 `bool`）。

浮点运算符（包括赋值运算符）从不产生异常。相反，在异常情况下，浮点运算产生零、无穷或 NaN，如下所述：

- 如果浮点运算的结果对于目标格式来说太小，则运算的结果将变为正零或负零。
- 如果浮点运算的结果对于目标格式来说太大，则运算的结果将变为正无穷或负无穷。

- 如果浮点运算无效，则运算的结果将变成 NaN。
- 如果浮点运算的一个或两个操作数为 NaN，则运算结果变成 NaN。

与运算的结果类型相比，浮点运算的精度可能更高。例如，某些硬件体系结构支持比 `double` 类型更大的范围和精度的 "扩展" 或 "long double" 浮点类型，并使用这种更高的精度类型隐式执行所有浮点运算。仅在性能较高的情况下，才可以使用较小的精度来执行浮点运算，而不需要实现使，C#而不是要求实现更高的精度类型用于所有浮点运算。除了提供更精确的结果，这种情况很少会带来任何实实在在的影响。但是，在形式 `x * y / z` 的表达式中，乘法产生的结果不在 `double` 范围内，而后续除法会将临时结果返回到 `double` 范围中，这就是在中计算表达式的事实较大范围格式可能导致生成有限的结果而不是无穷。

Decimal 类型

`decimal` 类型是适用于财务和货币计算的 128 位数据类型。`decimal` 类型可以表示从 `1.0 * 10^-28` 到大约 `7.9 * 10^28` 28-29 有效位的值。

`decimal` 类型的有限值集采用 `(-1)^s * c * 10^-e` 形式，其中符号 `s` 是0或1，系数 `c` 由 `0 <= *c* < 2^96` 给定，小数位数 `e` 为 `0 <= e <= 28`。`decimal` 类型不支持有符号的零、无穷大或 NaN。`decimal` 表示为96位整数，由10的幂进行缩放。对于绝对值小于 `1.0m` 的 `decimal`，该值精确到28个小数位，但再也不是。对于绝对值大于或等于 `1.0m``decimal`s，该值精确到28或29位。与 `float` 和 `double` 数据类型相反，十进制小数值(如0.1)可精确表示 `decimal` 表示形式。在 `float` 和 `double` 表示形式中，此类数字通常为无限小数，使这些表示形式更容易出现舍入错误。

如果二元运算符的一个操作数为 `decimal` 类型，则另一个操作数必须为整型类型或 `decimal` 类型。如果存在整数类型操作数，则在执行操作之前，它将转换为 `decimal`。

对类型 `decimal` 的值执行运算的结果是:计算确切的结果(根据每个运算符的定义保留小数位数)，然后舍入以适合表示形式。结果将舍入为最接近的可表示值，并且，当结果同样接近两个可表示值时，为在最小有效位位置(称为 "银行家舍入")中具有偶数的值。如果结果为零，则其符号始终为0，小数位数为0。

如果十进制算术运算生成的值小于或等于绝对值 `5 * 10^-29`，则运算结果将变为零。如果 `decimal` 算术运算生成的结果对 `decimal` 格式而言太大，则会引发 `System.OverflowException`。

`decimal` 类型具有比浮点类型更高的精度，但范围更小。因此，从浮点类型到 `decimal` 的转换可能会产生溢出异常，而从 `decimal` 到浮点类型的转换可能会导致精度损失。由于这些原因，浮点类型和 `decimal` 之间不存在隐式转换，并且没有显式强制转换，因此不能在同一个表达式中混合使用浮点和 `decimal` 操作数。

Bool 类型

`bool` 类型表示布尔的逻辑数量。`bool` 类型的可能值 `true` 和 `false`。

`bool` 和其他类型之间不存在标准转换。特别是，`bool` 类型是不同的，并且不同于整型，`bool` 并且不能用于替代整数值，反之亦然。

在 C 和C++语言中，零整数或浮点值，或者 null 指针可以转换为布尔值 `false`、非零整数或浮点值，或者可以将非 null 指针转换为布尔值 `true`。在C#中，此类转换通过将整数或浮点值显式比较为零来实现，或通过将对象引用显式比较为 `null` 来实现。

枚举类型

枚举类型是具有已命名常数的不同类型。每个枚举类型都有一个基础类型，该类型必须是 `byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`long` 或 `ulong`。枚举类型的值集与基础类型的值集相同。枚举类型的值不局限于命名常量的值。枚举类型是通过枚举声明(枚举声明)定义的。

可以为 null 的类型

可以为 null 的类型可以表示其*基础类型*的所有值以及其他 null 值。可以为 null 的类型写入 `T?`，其中 `T` 为基础类型。此语法是 `System.Nullable<T>` 的速记，这两种形式可互换使用。

与此相反，不*可以为 null 的值类型*为除 `System.Nullable<T>` 和其速记 `T?` 以外的任何值类型(适用于任何 `T`)

以及任何被约束为不可为 null 的值类型的类型参数(即，具有 `struct` 的任何类型参数约束)。

`System.Nullable<T>` 类型为 `T` (类型参数约束)指定值类型约束，这意味着可以为 null 的类型的基础类型可以是任何不可以为 null 的值类型。可以为 null 的类型的基础类型不能是可以为 null 的类型或引用类型。例如，`int??` 和 `string?` 是无效类型。

可以为 null 的类型的实例 `T?` 有两个公共只读属性：

- `bool` 类型的 `HasValue` 属性
- `T` 类型的 `Value` 属性

`HasValue` 为 true 的实例被称为非 null。非 null 实例包含已知值，`Value` 返回该值。

`HasValue` 为 false 的实例称为 null。空实例具有未定义的值。尝试读取 null 实例的 `Value` 将导致引发 `System.InvalidOperationException`。访问可以为 null 的实例的 `Value` 属性的过程称为***解包***。

除了默认构造函数，每个可以为 null 的类型 `T?` 都有一个公共构造函数，该构造函数采用 `T` 类型的单个自变量。给定 `T` 类型 `x` 的值，该窗体的构造函数调用

```
new T?(x)
```

创建 `Value` 属性 `x` 的 `T?` 的非 null 实例。为给定的值创建可为 null 的类型的非 null 实例的过程称为***包装***。

可从 `null` 文本使用隐式转换，以 `T?` (Null 文本转换)和从 `T` 到 `T?` (隐式可为 null 转换)。

引用类型

引用类型是类类型、接口类型、数组类型或委托类型。

```
reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    ;

class_type
    : type_name
    | 'object'
    | 'dynamic'
    | 'string'
    ;

interface_type
    : type_name
    ;

array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;

delegate_type
    : type_name
    ;
```

引用类型值是对类型*实例*的引用，后者称为*对象*。`null` 的特殊值与所有引用类型兼容，并指示缺少实例。

类类型

类类型定义一个数据结构，该结构包含数据成员（常量和字段）、函数成员（方法、属性、事件、索引器、运算符、实例构造函数、析构函数和静态构造函数）和嵌套类型。类类型支持继承，这是一个派生类可以扩展和特殊化基类的机制。类类型的实例是使用*object_creation_expression*s（对象创建表达式）创建的。

类中对类类型进行了说明。

某些预定义的C#类类型在语言中具有特殊含义，如下表中所述。

`System.Object`	所有其他类型的最终基类。请参阅对象类型。
`System.String`	C#语言的字符串类型。请参阅字符串类型。
`System.ValueType`	所有值类型的基类。请参阅system.object 类型。
`System.Enum`	所有枚举类型的基类。请参阅枚举。
`System.Array`	所有数组类型的基类。请参阅数组。

`System.Delegate`	所有委托类型的基类。请参阅委托。
`System.Exception`	所有异常类型的基类。请参见异常。

对象类型

`object` 类类型是所有其他类型的最终基类。C#直接或间接派生自 `object` 类类型的每个类型。

关键字 `object` 只是预定义的类 `System.Object` 的别名。

动态类型

`dynamic` 类型(如 `object`)可引用任何对象。将运算符应用于类型 `dynamic` 的表达式时，其解决方法会推迟到程序运行。因此，如果运算符无法合法地应用于被引用对象，则在编译过程中不会提供错误。当运算符的解析在运行时失败时，将引发异常。

其目的是允许动态绑定，这在动态绑定中进行了详细说明。

`dynamic` 被视为等同于 `object`，但以下方面除外：

- `dynamic` 类型的表达式的操作可以动态绑定(动态绑定)。
- 如果两者都是候选项，则类型推理(类型推理)优先于 `object` `dynamic`。

由于这种等效性，以下内容包含：

- `object` 和 `dynamic` 之间存在隐式标识转换，在将 `dynamic` 替换为 `object`
- 与 `object` 的隐式和显式转换也适用于 `dynamic`。
- 将 `dynamic` 替换为 `object` 时相同的方法签名被视为相同的签名
- 类型 `dynamic` 在运行时与 `object` 不区分。
- `dynamic` 类型的表达式称为*动态表达式*。

字符串类型

`string` 类型是直接从 `object` 继承的密封类类型。`string` 类的实例表示 Unicode 字符串。

`string` 类型的值可以编写为字符串文本(字符串文字)。

关键字 `string` 只是预定义的类 `System.String` 的别名。

接口类型

接口定义协定。实现接口的类或结构必须遵循它的协定。接口可以从多个基接口继承，类或结构可以实现多个接口。

接口类型在接口中进行了介绍。

数组类型

数组是一种数据结构，其中包含零个或多个通过计算索引访问的变量。数组中包含的变量(也称为数组的元素)都属于同一类型，并且此类型称为数组的元素类型。

数组中介绍了数组类型。

委托类型

委托是指一个或多个方法的数据结构。对于实例方法，它还引用其相应的对象实例。

C 或C++中委托的最接近等效项是函数指针，但函数指针只能引用静态函数，而委托可以引用静态和实例方法。在后一种情况下，委托不仅存储对方法入口点的引用，还存储对调用方法的对象实例的引用。

委托中介绍了委托类型。

装箱和取消装箱

装箱和取消装箱的概念是类型系统C#的核心。它通过允许将*value_type*的任何值相互转换为类型 `object`，在*value_type*s 和*reference_type*之间提供桥梁。装箱和取消装箱可实现类型系统的统一视图，其中，任何类型的值最终都可以被视为对象。

装箱转换

装箱转换允许将*value_type*隐式转换为*reference_type*。存在以下装箱转换：

- 从任何*value_type*到类型 `object`。
- 从任何*value_type*到类型 `System.ValueType`。
- 从任何*non_nullable_value_type*到*value_type*实现的任何*interface_type*。
- 从任何*nullable_type*到由*nullable_type*的基础类型实现的任何*interface_type*。
- 从任何为到类型 `System.Enum`。
- 从具有基础为的任何*nullable_type*到类型 `System.Enum`。
- 请注意，如果在运行时从值类型到引用类型的转换（涉及类型参数的隐式转换）结束，则从类型参数进行的隐式转换将作为装箱转换执行。

将*non_nullable_value_type*的值装箱包括分配对象实例，并将*non_nullable_value_type*值复制到该实例中。

如果将*nullable_type*的值装箱为 `null` 值（`HasValue` 为 `false`）或解包和装箱基础值的结果，则会生成空引用。

将*non_nullable_value_type*的值装箱的实际过程最好通过应该构想泛型***装箱类***的存在进行说明，该类的行为类似于声明为以下形式：

```
sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

`T` 类型的值 `v` 的装箱现在包含执行表达式 `new Box<T>(v)`，并将生成的实例作为类型 `object` 的值返回。因此，语句

```
int i = 123;
object box = i;
```

概念上对应于

```
int i = 123;
object box = new Box<int>(i);
```

上面的装箱类（如 `Box<T>`）实际上不存在，装箱值的动态类型实际上不是类类型。相反，类型 `T` 的装箱值具有动态类型 `T`，使用 `is` 运算符的动态类型检查只需引用类型 `T` 即可。例如，应用于对象的

```
int i = 123;
object box = i;
if (box is int) {
    Console.Write("Box contains an int");
}
```

会在控制台上输出字符串 "`Box contains an int`"。

装箱转换表示生成装箱的值的副本。这不同于*reference_type*到类型 `object` 的转换，在这种情况下，该值会继续引用相同的实例，并且只被视为 `object` 派生程度较小的类型。例如，给定声明

```
struct Point
{
    public int x, y;

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

以下语句

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

会在控制台上输出值10，因为在 `p` 分配给 `box` 时发生的隐式装箱操作会导致复制 `p` 的值。改为 `Point` 声明 `class`，因为 `p` 和 `box` 将引用同一个实例，所以值为20。

取消装箱转换

取消装箱转换允许将*reference_type*显式转换为*value_type*。存在以下取消装箱转换：

- 从类型 `object` 到任何*value_type*。
- 从类型 `System.ValueType` 到任何*value_type*。
- 从任何*interface_type*到任何实现*interface_type*的*non_nullable_value_type* 。
- 从任何*interface_type*到任何基础类型都实现*interface_type*的*nullable_type* 。
- 从类型 `System.Enum` 到任何为。
- 从类型 `System.Enum` 到具有基础为的任何*nullable_type* 。
- 请注意，如果在运行时结束将引用类型转换为值类型（显式动态转换），则会将显式转换为类型参数，以取消装箱转换。

对*non_nullable_value_type*的取消装箱操作包括：首先检查对象实例是否是给定*non_nullable_value_type*的装箱值，然后将值复制到该实例之外。

如果 `null` 的源操作数为 null，则为对*nullable_type*的取消装箱将生成*nullable_type*的 null 值；否则，会将对象实例取消装箱的结果返回到*nullable_type*的基础类型。

使用上一节中所述的虚部装箱类，`box` 到*value_type* `T` 的对象的取消装箱转换包含执行表达式 `((Box<T>)box).value`。因此，语句

```
object box = 123;
int i = (int)box;
```

概念上对应于

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

若要在运行时成功转换到给定的*non_nullable_value_type* ，源操作数的值必须是该*non_nullable_value_type*的装

箱值的引用。如果 `null` 源操作数，则会引发 `System.NullReferenceException` 。如果源操作数是对不兼容的对象的引用，则会引发 `System.InvalidCastException` 。

若要在运行时成功转换到给定的*nullable_type*，源操作数的值必须是 `null` 或对*nullable_type*基础*non_nullable_value_type*的装箱值的引用。如果源操作数是对不兼容的对象的引用，则会引发 `System.InvalidCastException` 。

构造类型

泛型类型声明本身表示***未绑定的泛型类型***，该类型用作 "蓝图" 以形成多种不同类型，方法是应用***类型参数***。类型参数以紧跟在泛型类型名称后面的尖括号（ `<` 和 `>` ）来编写。包含至少一个类型参数的类型称为***构造类型***。构造类型可用于语言中可显示类型名称的大多数位置。未绑定的泛型类型只能在*typeof_expression* （typeof 运算符）中使用。

构造类型还可以在表达式中用作简单名称（简单名称），也可以在访问成员（成员访问）时使用。

计算*namespace_or_type_name*时，只考虑具有正确数量的类型参数的泛型类型。因此，只要类型具有不同数目的类型参数，就可以使用相同的标识符来识别不同的类型。在同一程序中混合泛型和非泛型类时，这很有用：

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;

    class X
    {
        Queue q1;            // Non-generic Widgets.Queue
        Queue<int> q2;       // Generic Widgets.Queue
    }
}
```

*Type_name*可以标识构造的类型，即使它不直接指定类型参数也是如此。如果类型嵌套在泛型类声明中，并且包含声明的实例类型隐式用于名称查找（泛型类中的嵌套类型），则可能会发生这种情况。

```
class Outer<T>
{
    public class Inner {...}

    public Inner i;               // Type of i is Outer<T>.Inner
}
```

在不安全代码中，构造类型不能用作*unmanaged_type* （指针类型）。

类型参数

类型实参列表中的每个实参只是一*种类型*。

```
type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;
```

在不安全代码(不安全代码)中，*type_argument*可能不是指针类型。每个类型参数都必须满足相应类型参数的任何约束(类型参数约束)。

打开和关闭的类型

所有类型都可以分类为***开放式类型***或***关闭类型***。开放式类型是一种涉及类型参数的类型。更具体地说：

- 类型参数定义开放类型。
- 当且仅当数组的元素类型为开放类型时，该类型才是开放类型。
- 当且仅当一个或多个类型参数是开放类型时，构造类型是开放类型。当且仅当其包含类型的一个或多个类型参数或其包含类型的类型参数是开放类型时，构造的嵌套类型是开放类型。

关闭的类型是不是开放类型的类型。

在运行时，泛型类型声明中的所有代码在通过将类型自变量应用于泛型声明而创建的封闭式构造类型的上下文中执行。泛型类型中的每个类型形参都绑定到特定的运行时类型。所有语句和表达式的运行时处理始终出现在关闭的类型中，并且打开的类型仅在编译时处理期间出现。

每个封闭式构造类型都有自己的静态变量集，它们不与任何其他封闭构造类型共享。由于打开的类型在运行时不存在，因此没有与开放式类型关联的静态变量。如果两个封闭式构造类型是从同一个未绑定的泛型类型构造的，则这两个封闭式构造类型都是相同的类型，并且其对应的类型参数是相同的类型

绑定类型和未绑定类型

术语 "***未绑定类型***" 引用非泛型类型或未绑定的泛型类型。术语 "***绑定类型***" 是指非泛型类型或构造类型。

未绑定类型引用由类型声明声明的实体。未绑定的泛型类型本身不是类型，不能用作变量、参数或返回值的类型，也不能用作基类型。无法引用未绑定的泛型类型的唯一构造是 `typeof` 表达式(typeof 运算符)。

满足约束

只要引用构造类型或泛型方法，就会对照对泛型类型或方法(类型参数约束)声明的类型参数约束来检查提供的类型参数。对于每个 `where` 子句，对照每个约束检查对应于命名类型参数 `A` 类型参数，如下所示：

- 如果约束是类类型、接口类型或类型参数，则让 `C` 用提供的类型参数替换约束中出现的任何类型参数的约束。若要满足约束，必须为类型 `A` 可转换为以下类型之一 `C`：
  - 标识转换(标识转换)
  - 隐式引用转换(隐式引用转换)
  - 如果类型 A 是不可为 null 的值类型，则装箱转换(装箱转换)。
  - 从类型参数 `A` 到 `C` 的隐式引用、装箱或类型参数转换。
- 如果约束是引用类型约束(`class`)，则类型 `A` 必须满足以下条件之一：
  - `A` 是接口类型、类类型、委托类型或数组类型。请注意，`System.ValueType` 和 `System.Enum` 是满足此约束的引用类型。
  - `A` 是已知为引用类型的类型参数(类型参数约束)。
- 如果约束是值类型约束(`struct`)，则类型 `A` 必须满足以下条件之一：
  - `A` 是结构类型或枚举类型，但不能是可以为 null 的类型。请注意，`System.ValueType` 和 `System.Enum`

是不满足此约束的引用类型。
    - `A` 是具有值类型约束(类型参数约束)的类型参数。
- 如果约束是 `new()` 的构造函数约束，则类型 `A` 不得 `abstract`，并且必须具有公共的无参数构造函数。如果满足以下条件之一，则满足此要求：
    - `A` 是一种值类型，因为所有值类型都具有公共的默认构造函数(默认构造函数)。
    - `A` 是具有构造函数约束的类型参数(类型参数约束)。
    - `A` 是具有值类型约束(类型参数约束)的类型参数。
    - `A` 是一种不 `abstract` 的类，包含不带参数的显式声明的 `public` 构造函数。
    - `A` 不 `abstract` 并且具有默认的构造函数(默认构造函数)。

如果给定的类型参数不满足一个或多个类型参数的约束，则会发生编译时错误。

由于类型参数不是继承的，因此永远不会继承约束。在下面的示例中，`D` 需要指定其类型参数上的约束 `T` 以便 `T` 满足由基类 `B<T>` 强制的约束。与此相反，类 `E` 无需指定约束，因为 `List<T>` 为任何 `T` 实现 `IEnumerable`。

```
class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}
```

类型参数

类型参数是一个标识符，它指定在运行时参数绑定到的值类型或引用类型。

```
type_parameter
    : identifier
    ;
```

由于类型参数可以使用许多不同的实际类型参数进行实例化，因此类型参数与其他类型相比，操作和限制略有不同。这些方法包括：

- 类型参数不能直接用于声明基类(基类)或接口(变量类型参数列表)。
- 对类型参数进行成员查找的规则取决于应用于该类型参数的约束(如果有)。成员查找中对它们进行了详细介绍。
- 类型参数的可用转换取决于应用于该类型参数的约束(如果有)。其中详细介绍了涉及类型参数和显式动态转换的隐式转换。
- 文本 `null` 无法转换为类型参数给定的类型，除非已知该类型参数是引用类型(涉及类型参数的隐式转换)。但是，可以改为使用 `default` 表达式(默认值表达式)。此外，如果类型参数具有值类型约束，则可以使用 `==` 和 `!=`(引用类型相等运算符)与 `null` 比较类型参数指定的类型的值。
- 如果类型参数受*constructor_constraint*或值类型约束(类型参数约束)的约束，则只能将 `new` 表达式(对象创建表达式)与类型参数一起使用。
- 类型参数不能在属性中的任何位置使用。
- 在成员访问(成员访问)或类型名称(命名空间和类型名称)中不能使用类型参数来标识静态成员或嵌套类型。
- 在不安全代码中，类型参数不能用作*unmanaged_type*(指针类型)。

类型参数只是一种编译时构造。在运行时，每个类型参数都绑定到一个运行时类型，该类型是通过向泛型类型声明提供类型参数来指定的。因此，使用类型参数声明的变量的类型将在运行时是封闭式构造类型(开放和闭合类型)。所有涉及类型参数的语句和表达式的运行时执行都使用作为该参数的类型参数提供的实际类型。

表达式树类型

表达式树允许将 lambda 表达式表示为数据结构而不是可执行代码。表达式树是格式为 `System.Linq.Expressions.Expression<D>` 的***表达式树类型***的值，其中 `D` 为任意委托类型。对于本规范的其余部分，我们将使用速记 `Expression<D>` 引用这些类型。

如果从 lambda 表达式到委托类型 `D` 存在转换，则 `Expression<D>` 的表达式树类型也存在转换。尽管将 lambda 表达式转换为委托类型会生成一个委托，该委托引用 lambda 表达式的可执行代码，但转换为表达式树类型会创建 lambda 表达式的表达式树表示形式。

表达式树是 lambda 表达式的有效内存中数据表示形式，并使 lambda 表达式的结构透明和显式。

与委托类型 `D` 一样，`Expression<D>` 称为具有参数和返回类型，它们与 `D` 的类型相同。

下面的示例将 lambda 表达式表示为可执行代码，并表示为表达式树。由于 `Func<int,int>` 存在转换，因此 `Expression<Func<int,int>>` 也存在转换：

```
Func<int,int> del = x => x + 1;                  // Code

Expression<Func<int,int>> exp = x => x + 1;      // Data
```

按照这些分配，委托 `del` 引用返回 `x + 1` 的方法，并且表达式树 `exp` 引用用于描述表达式 `x => x + 1` 的数据结构。

在将 lambda 表达式转换为表达式树类型时，泛型类型 `Expression<D>` 的确切定义以及用于构造表达式树的准确规则都超出了此规范的范围。

要做出显式操作，需要注意以下两项：

- 并非所有 lambda 表达式都可以转换为表达式树。例如，具有语句体的 lambda 表达式和包含赋值表达式的 lambda 表达式不能表示。在这些情况下，转换仍存在，但会在编译时失败。匿名函数转换中详细介绍了这些异常。
- `Expression<D>` 提供了一个实例方法 `Compile`，该方法生成类型 `D` 的委托：

  ```
  Func<int,int> del2 = exp.Compile();
  ```

  调用此委托将导致执行表达式树所表示的代码。因此，根据上面的定义，del 和 del2 是等效的，以下两个语句将具有相同的效果：

  ```
  int i1 = del(1);

  int i2 = del2(1);
  ```

  执行此代码后，`i1` 和 `i2` 都具有 `2` 的值。

变量

2020/11/2 • • Edit Online

变量表示存储位置。每个变量都有一个类型，用于确定可以在变量中存储的值。C#是一种类型安全的语言，C#编译器保证变量中存储的值始终为适当的类型。可以通过赋值或使用 `++` 和 `--` 运算符来更改变量的值。

在可以获取变量的值之前，必须对其进行*明确*赋值（明确赋值）。

如以下各节所述，变量*最初已赋值*或*最初未分配*。最初分配的变量具有定义完善的初始值，并始终被视为明确赋值。初始未赋值的变量没有初始值。对于在某个特定位置被视为明确赋值的初始未赋值的变量，对该变量的赋值必须出现在通向该位置的每个可能的执行路径中。

变量类别

C#定义七类变量：静态变量、实例变量、数组元素、值参数、引用参数、输出参数和局部变量。以下各节介绍了其中的每个类别。

示例中

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

`x` 是一个静态变量，`y` 它是一个实例变量 `v[0]`，是一个数组元素 `a`，是一个 `c` 值参数 `b`，是一个引用参数，是一个输出参数 `i`，是一个本地变量.

静态变量

使用 `static` 修饰符声明的字段称为*静态变量*。静态变量在为其包含类型的静态构造函数（静态构造函数）的执行之前就已存在，如果关联的应用程序域停止存在，则不再存在。

静态变量的初始值是变量类型的默认值（默认值）。

出于明确赋值检查的目的，静态变量被视为初始赋值。

实例变量

不使用 `static` 修饰符声明的字段称为*实例变量*。

类中的实例变量

当创建该类的新实例时，该类的实例变量就会成为存在，如果没有对该实例的引用，并且已执行了该实例的析构函数（如果有），就不再存在。

类的实例变量的初始值是变量类型的默认值（默认值）。

出于明确赋值检查的目的，类的实例变量被视为初始赋值。

结构中的实例变量

结构的实例变量与它所属的结构变量的生存期完全相同。换而言之，当结构类型的变量变为存在或不再存在时，该结构的实例变量也是如此。

结构的实例变量的初始赋值状态与包含结构变量的初始赋值状态相同。换言之，当结构变量被视为初始已赋值，因此它的实例变量也被视为未赋值时，它的实例变量会同样取消分配。

数组元素

当创建数组实例时，数组的元素便会存在，并且在没有对该数组实例的引用时停止存在。

数组中每个元素的初始值为数组元素类型的默认值（默认值）。

出于明确赋值检查的目的，数组元素被视为初始已赋值。

值参数

不带 `ref` 或 `out` 修饰符声明的参数是一个*值参数*。

值参数会在调用该参数所属的函数成员（方法、实例构造函数、访问器或运算符）或匿名函数时成为存在，并使用调用中给定的自变量的值进行初始化。当函数成员或匿名函数返回时，值参数通常不存在。但是，如果值参数由匿名函数（匿名函数表达式）捕获，则其生存期将至少扩展到从该匿名函数创建的委托或表达式树符合垃圾回收的条件。

出于明确赋值检查的目的，值参数被视为初始已赋值。

引用参数

使用 `ref` 修饰符声明的参数是一个*引用参数*。

引用参数不会创建新的存储位置。相反，引用参数与给定作为函数成员或匿名函数调用中的参数的变量表示相同的存储位置。因此，引用参数的值始终与基础变量相同。

以下明确赋值规则适用于引用参数。请注意输出参数中所述的输出参数的不同规则。

- 在函数成员或委托调用中将变量作为引用参数传递之前，必须对其进行明确赋值（明确赋值）。
- 在函数成员或匿名函数中，引用参数被视为初始赋值。

在结构类型的实例方法或实例访问器中，关键字 `this` 的行为与结构类型（此访问）的引用参数完全相同。

输出参数

使用 `out` 修饰符声明的参数是一个*输出参数*。

Output 参数不会创建新的存储位置。相反，output 参数表示作为函数成员或委托调用中的参数提供的相同存储位置。因此，output 参数的值始终与基础变量相同。

以下明确的赋值规则适用于 output 参数。请注意引用参数中所述的引用参数的不同规则。

- 在函数成员或委托调用中将变量作为 output 参数传递之前，不需要明确赋值。
- 完成函数成员或委托调用的正常完成后，作为输出参数传递的每个变量都被视为在该执行路径中分配。
- 在函数成员或匿名函数中，output 参数被视为初始未赋值。
- 函数成员或匿名函数的每个输出参数都必须在函数成员或匿名函数正常返回之前明确赋值（明确赋值）。

在结构类型的实例构造函数中，关键字 `this` 的行为与结构类型的输出参数（此访问）完全相同。

局部变量

*局部变量*由*local_variable_declaration*声明，该变量可能出现在*块*、*for_statement*、*switch_statement*或*using_statement*中。或*try_statement*的*foreach_statement*或*specific_catch_clause* 。

局部变量的生存期是程序执行的一部分，在该过程中，将保证存储的存储空间。此生存期至少会将输入内容扩展到*块*、*for_statement*、*switch_statement*、*using_statement*、*foreach_statement*或与其关联的*specific_catch_clause*，直到*该块*的执行、*for_statement*、*switch_statement*、*using_statement*、*foreach_statement*或*specific_catch_clause*以任何方式结束。（输入封闭的*块*或调用方法会挂起，但不会结束、执行当前*块*、*for_statement*、*switch_statement*、*using_statement*、*foreach_statement*或*specific_catch_clause*。）如果通过匿名函数（捕获的外部变量）捕获本地变量，则其生存期至少将一直扩展到从匿名函数创建的委托或表

达式树以及引用捕获的变量适用于垃圾回收。

如果以递归方式进入父*block*、*for_statement*、*switch_statement*、*using_statement*、*foreach_statement*或*specific_catch_clause*，则将创建一个新的本地变量实例，每次计算时间及其*local_variable_initializer*（如果有）。

*Local_variable_declaration*引入的本地变量不会自动初始化，因此没有默认值。出于明确赋值检查的目的，*local_variable_declaration*引入的局部变量被视为初始未分配。*Local_variable_declaration*可以包括*local_variable_initializer*，在这种情况下，仅在初始化表达式（声明语句）后才将变量视为明确赋值。

在*local_variable_declaration*引入的局部变量范围内，在其*local_variable_declarator*之前的文本位置引用该局部变量是编译时错误。如果局部变量声明是隐式的（局部变量声明），则在其*local_variable_declarator*中引用该变量也是错误的。

*Foreach_statement*或*specific_catch_clause*引入的局部变量在整个范围内被视为已明确赋值。

局部变量的实际生存期取决于实现。例如，编译器可能会以静态方式确定块中的某个局部变量仅用于该块的一小部分。使用此分析，编译器可以生成代码，使变量的存储具有比包含块更短的生存期。

局部引用变量所引用的存储将独立于该本地引用变量的生存期（自动内存管理）进行回收。

默认值

以下类别的变量会自动初始化为其默认值：

- 静态变量。
- 类实例的实例变量。
- 数组元素。

变量的默认值取决于变量的类型，并按如下所示确定：

- 对于*value_type*的变量，默认值与*value_type*的默认构造函数（默认构造函数）所计算的值相同。
- 对于*reference_type*的变量，默认值为`null`。

默认值的初始化通常是通过使内存管理器或垃圾回收器将内存初始化为全部位零来完成的，然后再将其分配给使用。出于此原因，可以方便地使用所有位数表示空引用。

明确赋值

在函数成员的可执行代码中的给定位置，***如果编译器***可以通过特定的静态流分析（确定明确赋值的精确规则）来证明变量已自动初始化或已成为至少一个赋值的目标。非正式地说，明确赋值的规则如下：

- 最初分配的变量（最初分配的变量）始终被视为明确赋值。
- 如果指向该位置的所有可能的执行路径至少包含以下内容之一，则初始未分配的变量（最初未分配的变量）将被视为在给定位置明确赋值：
  - 简单赋值（简单赋值），其中变量是左操作数。
  - 将变量作为输出参数传递的调用表达式（调用表达式）或对象创建表达式（对象创建表达式）。
  - 对于本地变量，则为包含变量初始值设定项的局部变量声明（局部变量声明）。

最初分配的变量、最初未分配的变量和用于确定明确赋值的精确规则中介绍了上述非正式规则基础的正式规范。

*Struct_type*变量的实例变量的明确赋值状态是单独跟踪的。除了以上规则，以下规则适用于*struct_type*变量及其实例变量：

- 如果实例变量的包含*struct_type*变量被视为已明确赋值，则将其视为明确赋值。
- 如果将*struct_type*变量的每个实例变量都视为明确赋值，则该变量将被视为明确赋值。

在以下上下文中，明确赋值是必需的：

- 变量必须在获取其值的每个位置明确赋值。这可确保永远不会出现未定义的值。表达式中的变量的出现位置被视为获取该变量的值，但在
  - 变量是简单赋值的左操作数，
  - 变量作为 output 参数传递，或
  - 变量是*struct_type*变量，作为成员访问的左操作数出现。
- 变量必须在作为引用参数传递的每个位置上进行明确赋值。这可确保被调用的函数成员可以考虑最初分配的引用参数。
- 函数成员的所有输出参数都必须在函数成员返回的每个位置明确赋值（通过 `return` 语句或通过执行到达函数成员主体的末尾）。这可确保函数成员不在输出参数中返回未定义的值，从而使编译器可以考虑使用变量作为输出参数的函数成员调用，该参数等效于对变量赋值的输出参数。
- 必须在该实例构造函数返回的每个位置明确分配*struct_type*实例构造函数的变量。`this`

最初分配的变量

以下类别的变量被分类为初始分配：

- 静态变量。
- 类实例的实例变量。
- 最初分配的结构变量的实例变量。
- 数组元素。
- 值参数。
- 引用参数。
- `catch` 子句 `foreach` 或语句中声明的变量。

初始未赋值变量

以下类别的变量被分类为初始未赋值：

- 初始未分配的结构变量的实例变量。
- 输出参数，包括 `this` 结构实例构造函数的变量。
- 局部变量（ `catch` 子句 `foreach` 或语句中声明的变量除外）。

确定明确赋值的精确规则

为了确定每个已使用的变量是否已明确赋值，编译器必须使用与本部分中所述等效的进程。

编译器处理每个具有一个或多个初始未赋值变量的函数成员的正文。对于每个初始未赋值的变量*v*，编译器会在函数成员的以下各点处确定*v*的***明确赋值状态***：

- 在每个语句的开头
- 每个语句的终结点（终结点和可访问性）
- 在将控制转移到另一条语句或语句结束点的每个弧线上
- 在每个表达式的开头
- 在每个表达式的末尾

*v*的明确赋值状态可以是：

- 明确赋值。这表明，在所有可能的控制流到目前为止，*v*已分配有一个值。
- 未明确赋值。对于类型 `bool` 为的表达式末尾的变量状态，未明确赋值的变量的状态可能（但不一定）属于以下子状态之一：
  - 在 true 表达式后明确赋值。此状态表明，如果布尔表达式的计算结果为 true，则会明确赋值*v*；如果布尔表达式的计算结果为 false，则不一定赋值。
  - 在 false 表达式后明确赋值。此状态表明，如果布尔表达式的计算结果为 false，则会明确赋值*v*；如果布尔表达式的计算结果为 true，则不一定赋值。

以下规则控制如何在每个位置确定变量*v*的状态。

语句的一般规则

- 在函数成员主体的开头，*v*未明确赋值。
- 在任何无法访问的语句开始时，均明确赋值*v*。
- *V*在任何其他语句开始时的明确赋值状态是通过检查*v*在该语句开头的所有控制流传输上的明确赋值状态。如果在所有此类控制流传输上明确分配了（且仅当）*v*，则在语句的开头明确赋值*v*。可能的控制流传输集的确定方式与检查语句可访问性（终结点和可访问性）相同。
- *V*在`checked`块、`do` `unchecked`、、`if`、、、、、、或的终结点上的明确赋值状态`while` `for` `foreach` `lock` `using` `switch`语句是通过检查以该语句的终结点为目标的所有控制流传输上的*v*的明确赋值状态来确定的。如果对所有此类控制流传输明确赋值*v*，则*v*在语句的终结点上是明确赋值的。本来在语句的结束点上不明确赋值*v*。可能的控制流传输集的确定方式与检查语句可访问性（终结点和可访问性）相同。

Block 语句、checked 和 unchecked 语句

如果控件传输到块中语句列表的第一条语句（如果语句列表为空，则为到块的终结点），则控件上的*v*的明确赋值状态与块之前的*v*的明确赋值语句相同。、`checked`或`unchecked`语句。

表达式语句

对于包含表达式*expr*的表达式语句*stmt*：

- 在*stmt*开始时，*v*在*expr*开头具有相同的明确赋值状态。
- 如果在*expr*的末尾指定了"*v*"，则它在*stmt*的终结点上进行明确赋值;本来它不是在*stmt*终结点明确赋值的。

声明语句

- 如果*stmt*是没有初始值设定项的声明语句，则*v*在*stmt*的终结点上的明确赋值状态与*stmt*的开头相同。
- 如果*stmt*是带有初始值设定项的声明语句，则*v*的明确赋值状态被确定为，但对于带有初始值设定项的每个声明使用一个赋值语句（顺序为声明）。

If 语句

对于窗体的语句`if` stmt:

```
if ( expr ) then_stmt else else_stmt
```

- 在*stmt*开始时，*v*在*expr*开头具有相同的明确赋值状态。
- 如果在*expr*结束时，*v*是明确赋值的，则在到*then_stmt*的控制流传输上，并将其指定为*else_stmt*，如果没有 else 子句，则将其分配给*stmt*的终结点。
- 如果在*expr*的末尾，*v*的状态为"的确赋值为 true expression"，则它在控制流到*then_stmt*的传输上明确赋值，而不是在控制流传输上明确分配到任一*else_* 如果没有 else 子句，则为 stmt 或*stmt*的终结点。
- 如果*v*在*expr*的末尾具有状态"在假表达式后明确赋值"，则它在控制流到*else_stmt*的传输上明确赋值，在控制流到 then_stmt 的传输上未明确赋值。当且仅当在*then_stmt*的终结点明确赋值时，它才会在*stmt*的终结点上进行明确赋值。
- 否则，在对*then_stmt*或*else_stmt*的控制流传输上，*v*被视为未明确分配，如果不存在 else 子句，则被视为*stmt*的终结点。

Switch 语句

使用控制表达式 *expr*的语句`switch` stmt:

- *Expr*开头的*v*的明确赋值状态与*stmt*开头的*v*的状态相同。
- 到可访问的 switch 块语句列表的控制流传输上，*v*的明确分配状态与*expr*末尾的*v*的明确赋值状态相同。

While 语句

对于窗体的语句`while` stmt:

```
while ( expr ) while_body
```

- 在*stmt*开始时，*v*在*expr*开头具有相同的明确赋值状态。

- 如果在*expr*结束时，*v*是明确赋值的，则它在控制流到*while_body*的传递和到*stmt*的结束点上都是明确赋值的。
- 如果在*expr*的末尾，*v*的状态为 "明确分配给 true 表达式"，则它在控制流到*while_body*的传输上明确赋值，但不是在*stmt*终结点明确赋值。
- 如果在*expr*的末尾，*v*的状态为 "指定为 false expression 后明确赋值"，则它在控制流到*stmt*终点的传输上明确赋值，但并不是在控制流传输时明确赋值。_*body*。

Do 语句

对于窗体的语句 `do` stmt：

```
do do_body while ( expr ) ;
```

- *v*在从*stmt*开始到*do_body*的控制流传输上具有相同的明确赋值状态，就像在*stmt*的开头。
- 对于位于*do_body*的终结点的*expr*，*v*具有相同的明确赋值状态。
- 如果在*expr*结束时，*v*是明确赋值的，则它在控制流到*stmt*终点的传输上明确赋值。
- 如果在*expr*的末尾，*v*的状态为 "在假表达式后明确赋值"，则它在控制流到*stmt*终点的传输上明确赋值。

For 语句

对以下形式的 `for` 语句的明确赋值检查：

```
for ( for_initializer ; for_condition ; for_iterator ) embedded_statement
```

像编写语句一样完成：

```
{
    for_initializer ;
    while ( for_condition ) {
        embedded_statement ;
        for_iterator ;
    }
}
```

如果 `for` 语句中省略*for_condition*，则对明确赋值的计算将继续执行，就像上述扩展 `true` 中已将*for_condition*替换为。

Break、continue 和 goto 语句

由 `break`、或语句 `goto` 导致的控制流传输上的 v 的明确赋值状态与语句开头的 v 的明确赋值状态相同。
`continue`

Throw 语句

对于窗体的语句*stmt*

```
throw expr ;
```

*Expr*开头的*v*的明确赋值状态与*stmt*开头的*v*的明确赋值状态相同。

Return 语句

对于窗体的语句*stmt*

```
return expr ;
```

- *Expr*开头的*v*的明确赋值状态与*stmt*开头的*v*的明确赋值状态相同。
- 如果*v*是 output 参数，则必须为其赋值：

- *expr*之后
- 或 `finally` 包含语句 `return` 的块 `try` - `finally` 的结尾- 。 `try` `catch` - `finally`

对于窗体的语句 stmt：

```
return ;
```

- 如果*v*是 output 参数，则必须为其赋值：
  - *stmt*之前
  - 或 `finally` 包含语句 `return` 的块 `try` - `finally` 的结尾- 。 `try` `catch` - `finally`

Try-catch 语句

对于窗体的语句*stmt* ：

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
```

- 在*try_block*开始时， *v*的明确赋值状态与*stmt*开头的*v*的明确赋值状态相同。
- *Catch_block_i*开头的*v*的明确分配状态（适用于任何*i*）与*stmt*开头的*v*的明确赋值状态相同。
- 如果在*try_block*的终结点和每个 catch_block_i（*对于每个，则为*1 到 n 的每个 ）明确分配*v*时，).

Try-catch 语句

对于窗体的语句 `try` stmt：

```
try try_block finally finally_block
```

- 在*try_block*开始时， *v*的明确赋值状态与*stmt*开头的*v*的明确赋值状态相同。
- 在*finally_block*开始时， *v*的明确赋值状态与*stmt*开头的*v*的明确赋值状态相同。
- 如果（且仅当）满足以下条件之一，则会明确分配*v*在*stmt*终结点上的明确赋值状态：
  - 在*try_block*的终结点明确赋值*v*
  - 在*finally_block*的终结点明确赋值*v*

如果控制 `goto` 流传输（例如，语句）是在*try_block*中开始，并在*try_block*之外结束，则在该控制流传输上， *v*也被视为明确赋值（如果*v*是在*finally_block*的终结点明确赋值。（这并不是唯一的情况，如果对此控制流传输的另一个原因明确指定*v*，则仍将其视为明确分配。）

Try-catch-finally 语句

对 `try` 以下形式 `catch` 的语句的 `finally` 明确赋值分析：- -

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
finally *finally_block*
```

如 `try` 语句-是包含语句的 `try` 语句一样完成： `finally` - `catch`

```
try {
    try try_block
    catch(...) catch_block_1
    ...
    catch(...) catch_block_n
}
finally finally_block
```

下面的示例演示 `try` 语句的不同块(try 语句)如何影响明确赋值。

```
class A
{
    static void F() {
        int i, j;
        try {
            goto LABEL;
            // neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
        }

        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }

        finally {
            // neither i nor j definitely assigned
            j = 5;
            // j definitely assigned
            }
        // i and j definitely assigned
        LABEL:;
        // j definitely assigned

    }
}
```

Foreach 语句

对于窗体的语句 `foreach` stmt:

```
foreach ( type identifier in expr ) embedded_statement
```

- *Expr*开头的*v*的明确赋值状态与*stmt*开头的*v*的状态相同。
- 将控制流传输到*embedded_statement*或*stmt*终结点时，*v*的明确分配状态与*expr*末尾处的*v*状态相同。

Using 语句

对于窗体的语句 `using` stmt:

```
using ( resource_acquisition ) embedded_statement
```

- 在*resource_acquisition*开始时，*v*的明确赋值状态与*stmt*开头的*v*的状态相同。
- 控制流传输到*embedded_statement*的*v*的明确分配状态与*resource_acquisition*末尾的*v*状态相同。

Lock 语句

对于窗体的语句 `lock` stmt:

```
lock ( expr ) embedded_statement
```

- *Expr*开头的*v*的明确赋值状态与*stmt*开头的*v*的状态相同。
- 控制流传输到*embedded_statement*的*v*的明确分配状态与*expr*末尾处的*v*状态相同。

Yield 语句

对于窗体的语句`yield return` stmt:

```
yield return expr ;
```

- *Expr*开头的*v*的明确赋值状态与*stmt*开头的*v*的状态相同。
- 在*stmt*结束时，*v*的明确赋值状态与*expr*末尾的*v*状态相同。
- `yield break`语句对明确赋值状态没有影响。

简单表达式的一般规则

以下规则适用于这些类型的表达式：文本(文本)、简单名称(简单名称)、成员访问表达式(成员访问)、非索引的基本访问表达式(基本访问)、`typeof`表达式(typeof 运算符)、默认值表达式(默认值表达式)和`nameof`表达式(Nameof 表达式)。

- 此类表达式结束时，*v*的明确赋值状态与表达式开头*v*的明确赋值状态相同。

带有嵌入式表达式的表达式的一般规则

以下规则适用于这些类型的表达式：带圆括号的表达式(带括号的表达式)、元素访问表达式(元素访问)、带有索引的基本访问表达式(基本访问)、增量和减量表达式(后缀递增和递减运算符，前缀增量和减量运算符)，强制转换表达式(强制转换表达式)`+`，`-`一元`~`，，，`*`表达式、二进制`+`、`-`、`*`、`/`、`%`、`<<`、`>>`、`<`、`<=`、`>`、`>=`、`==`、`!=`、`is`、`as`、`&`、`|`、`^`表达式(算术运算符、移位运算符、关系和类型测试运算符、逻辑运算符)、复合赋值表达式(复合`checked`赋值)和`unchecked`表达式(checked 和 unchecked 运算符)以及数组和委托创建表达式(new 运算符)。

其中每个表达式都具有一个或多个按固定顺序无条件计算的子表达式。例如，二元`%`运算符计算运算符的左侧，然后计算右侧。索引操作将计算索引表达式的值，然后按从左至右的顺序计算每个索引表达式。对于具有子表达式*e1*、*e2*、...、*eN*的表达式*expr*，按以下顺序计算：

- 在*e1*开始时，*v*的明确赋值状态与*expr*开头的明确赋值状态相同。
- *Ei*(*i*大于1)开头的*v*的明确赋值状态与上一个子表达式末尾的明确赋值状态相同。
- 在*expr*结束时，*v*的明确赋值状态与*eN*末尾的明确赋值状态相同

调用表达式和对象创建表达式

对于以下形式的调用表达式*expr*：

```
primary_expression ( arg1 , arg2 , ... , argN )
```

或形式的对象创建表达式：

```
new type ( arg1 , arg2 , ... , argN )
```

- 对于调用表达式，*v*之前的*primary_expression*的明确赋值状态与*expr*前面的*v*的状态相同。
- 对于调用表达式，v 之前的*v*的明确赋值状态与 *primary_expression*后*v*的状态相同。
- 对于对象创建表达式，v 之前的*v*的明确赋值状态与 *expr*前面的*v*的状态相同。
- 对于每个*参数 argi*，*argi*之后的`ref` *v*的明确赋值状态由标准表达式规则确定，忽略任何或`out`修饰符。
- 对于每个大于一的参数*argi*，*argi*之前的*v*的明确赋值状态与上一个*arg*后*v*的状态相同。

- 如果在任何参数中将变量 v `out` 作为参数(即，`out v` 形式的参数)传递，则*expr*后的*v*状态将明确赋值。本来*expr*后*v*的状态与*argN*后*v*的状态相同。
- For 数组初始值设定项(数组创建表达式)、对象初始值设定项(对象初始值设定项)、集合初始值设定项(集合初始值设定项)和匿名对象初始值设定项(匿名对象创建表达式)，明确的赋值状态由定义这些构造的扩展确定。

简单赋值表达式

对于以下 `w = expr_rhs` 形式*的表达式表达式*:

- *Expr_rhs*之前的*v*的明确赋值状态与*expr*之前的*v*的明确赋值状态相同。
- *Expr*后*v*的明确赋值状态由确定:
  - 如果*w*是与*v*相同的变量，则*expr*后*v*的明确赋值状态是明确赋值的。
  - 否则，如果在结构类型的实例构造函数中发生分配，则如果*w*是属性访问，它指定了正在构造的实例上自动实现的属性*P*，而*v*是的隐藏支持字段*P*，则*expr*后*v*的明确赋值状态为明确赋值。
  - 否则，在*expr*后，*v*的明确赋值状态与*expr_rhs*后*v*的明确赋值状态相同。

&& (条件和)表达式

对于以下 `expr_first && expr_second` 形式*的表达式表达式*:

- *Expr_first*之前的*v*的明确赋值状态与*expr*之前的*v*的明确赋值状态相同。
- 如果*expr_first*后*v*的状态为明确赋值或 "true expression 之后明确赋值"，则*expr_second*之前*v*的明确赋值状态为明确赋值。否则，不会明确赋值。
- *Expr*后*v*的明确赋值状态由确定:
  - 如果*expr_first*是一个具有值 `false` 的常量表达式，则在*expr*后*v*的明确赋值状态与*expr_first*后*v*的明确赋值状态相同。
  - 否则，如果*expr_first*后*v*的状态为明确赋值，则*expr*后*v*的状态将明确赋值。
  - 否则，如果*expr_second*后*v*的状态为 "已明确分配"，而 " *expr_first* " *的状态*为 "在 false 表达式后明确赋值"，则*expr*后*v*的状态肯定是已.
  - 否则，如果*expr_second*后*v*的状态为 "明确赋值" 或 "true expression 之后明确赋值"，则*expr*后*v*的状态为 "在 true 表达式后明确赋值"。
  - 否则，如果*expr_first*后*v*的状态为 "在假表达式后明确赋值"，并且*expr_second*之后的*v*状态是 *"在false*表达式后明确赋值"，则*expr*是 "在假表达式后明确赋值"。
  - 否则，不明确赋值*expr*后的*v*状态。

示例中

```
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
        // i not definitely assigned
    }
}
```

在 `if` 语句 `i` 的一个嵌入语句中(而不是在另一个语句中)，该变量被视为已明确赋值。在方法 `if` `F` 的语句中，变量 `i` 在第一个嵌入语句中是明确赋值的，因为在执行 `(i = y)` 此嵌入语句之前，表达式的执行始终为。与此相反，第 `i` 二个嵌入语句中的变量并不是明确 `x >= 0` 赋值的，因为可能已对 false 进行 `i` 了测试，导致变量未赋值。

||（条件或）表达式

对于以下 `expr_first || expr_second` 形式*的表达式表达式*：

- *Expr_first*之前的*v*的明确赋值状态与*expr*之前的*v*的明确赋值状态相同。
- 如果*expr_first*后*v*的状态为明确赋值或 "在 false 表达式后明确赋值"，则*expr_second*之前*v*的明确赋值状态为明确赋值。否则，不会明确赋值。
- *Expr*后的*v*的明确赋值语句由确定：
  - 如果*expr_first*是一个具有值 `true` 的常量表达式，则在*expr*后*v*的明确赋值状态与*expr_first*后*v*的明确赋值状态相同。
  - 否则，如果*expr_first*后*v*的状态为明确赋值，则*expr*后*v*的状态将明确赋值。
  - 否则，如果*expr_second*后*v*的状态是明确赋值的，并且*expr_first*之后的*v*状态是 "在真正表达式后明确赋值"，则*expr*后*v*的状态肯定是已.
  - 否则，如果*expr_second*后*v*的状态是明确赋值的或 "在假表达式后明确赋值"，则*expr*后*v*的状态为 "在 false 表达式后明确赋值"。
  - 否则，如果*expr_first*后*v*的状态为 "在 true 表达式后明确赋值"，并且*expr_second*之后的*v*状态是 "true expression 之后明确赋值"，则在 expr 后的*v*状态为 "true 表达式后明确赋值"。
  - 否则，不明确赋值*expr*后的*v*状态。

示例中

```
class A
{
    static void G(int x, int y) {
        int i;
        if (x >= 0 || (i = y) >= 0) {
            // i not definitely assigned
        }
        else {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}
```

在 `if` 语句 `i` 的一个嵌入语句中（而不是在另一个语句中），该变量被视为已明确赋值。在方法 `if` `G` 的语句中，变量 `i` 在第二个嵌入语句中是明确赋值的，因为在 `(i = y)` 执行此嵌入语句之前，表达式的执行始终为。与此相反，第 `i` 一个嵌入语句中的变量不是明确赋值的 `x >= 0` ，因为可能测试了 true，导致变量 `i` 被分配。

!（逻辑非）表达式

对于以下 `! expr_operand` 形式*的表达式表达式*：

- *Expr_operand*之前的*v*的明确赋值状态与*expr*之前的*v*的明确赋值状态相同。
- *Expr*后*v*的明确赋值状态由确定：
  - 如果在 * expr_operand * 之后的*v*状态为明确赋值，则*expr*后*v*的状态将明确赋值。
  - 如果未明确赋值 * expr_operand * 之后的*v*状态，则不明确赋值*expr*后的*v*状态。
  - 如果在 * expr_operand * 之后的*v*状态是 "在假表达式后明确赋值"，则*expr*后面的*v*状态是 "在真正表达式后明确赋值"。
  - 如果在 * expr_operand * 之后的*v*状态是 "在 true 表达式后明确赋值"，则*expr*后*v*的状态为 "在假表达式后明确赋值"。

??（null 合并）表达式

对于以下 `expr_first ?? expr_second` 形式*的表达式表达式*：

- *Expr_first*之前的*v*的明确赋值状态与*expr*之前的*v*的明确赋值状态相同。
- *Expr_second*之前的*v*的明确赋值状态与*expr_first*后*v*的明确赋值状态相同。

- *Expr*后的*v*的明确赋值语句由确定：
  - 如果*expr_first*是一个值为 null 的常量表达式（常数表达式），则在*expr*后，*v*的状态与*expr_second*后*v*的状态相同。
- 否则，在*expr*后，*v*的状态与*expr_first*后*v*的明确赋值状态相同。

?：（条件）表达式

对于以下 `expr_cond ? expr_true : expr_false` 形式*的表达式表达式*：

- *Expr_cond*之前的*v*的明确赋值状态与*expr*前面的*v*的状态相同。
- 当且仅当以下其中一项保留时，*v 前 v*的明确赋值*状态才是*明确赋值的：
  - *expr_cond*是具有值的常量表达式 `false`
  - *expr_cond*后*v*的状态是明确赋值的，或者是 "true 表达式后明确赋值"。
- 当且仅当以下其中一项保留时，*v 前 v*的明确赋值*状态才是*明确赋值的：
  - *expr_cond*是具有值的常量表达式 `true`
- *expr_cond*之后的*v*状态是明确赋值的，或者是 "在假表达式后明确赋值"。
- *Expr*后*v*的明确赋值状态由确定：
  - 如果*expr_cond*是 `true` 具有值的常量表达式（常数表达式），则*expr*后*v*的状态与*expr_true*后*v*的状态相同。
  - 否则，如果*expr_cond*是 `false` 具有值的常量表达式（常数表达式），则*expr*后*v*的状态与*expr_false*后*v*的状态相同。
  - 否则，如果*expr_true*后*v*的状态为 "已明确分配" 并且 expr_false*的*状态为 "已明确分配"，则*expr*后*v*的状态将明确赋值。
  - 否则，不明确赋值*expr*后的*v*状态。

匿名函数

对于带有主体（*块*或*表达式*）*正文*的*lambda_expression*或*anonymous_method_expression expr*：

- 在*body*之前，外部变量*v*的明确赋值状态与*expr*前面的*v*的状态相同。也就是说，外部变量的明确赋值状态是从匿名函数的上下文继承而来的。
- *Expr*后面的外部变量*v*的明确赋值状态与*expr*前面的*v*的状态相同。

示例

```
delegate bool Filter(int i);

void F() {
    int max;

    // Error, max is not definitely assigned
    Filter f = (int n) => n < max;

    max = 5;
    DoWork(f);
}
```

生成编译时错误，因为 `max` 在声明匿名函数的位置未明确赋值。示例

```
delegate void D();

void F() {
    int n;
    D d = () => { n = 1; };

    d();

    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

还会生成编译时错误，因为对匿名函数 `n` 的赋值不会影响匿名函数 `n` 之外的明确赋值状态。

变量引用

*Variable_reference*是分类为变量的*表达式*。*Variable_reference*表示可访问的存储位置以提取当前值并存储新值。

```
variable_reference
    : expression
    ;
```

在 C 和C++中， *variable_reference*称为*lvalue*。

变量引用的原子性

以下数据类型的读取和写入是原子的： `bool` 、 `char` `short` `ushort` `byte` `sbyte` 、、 `uint` 、、 `int` 、、、和引用类型。 `float` 此外，在前面的列表中，具有基础类型的枚举类型的读取和写入也是原子的。其他类型（ `long` 包括、 `ulong` 、 `double` 、以及 `decimal` 用户定义类型）的读取和写入不能保证为原子性。除了为此目的而设计的库函数外，不保证原子读修改写入，如递增或递减。

转换

2020/11/2 • • Edit Online

*转换*允许将表达式视为特定类型。转换可能会导致将给定类型的表达式视为具有不同的类型，也可能导致不具有类型的表达式获得类型。转换可以是*隐式*的或*显式*的，这确定是否需要显式强制转换。例如，从类型 `int` 到类型 `long` 的转换是隐式的，因此类型 `int` 的表达式可隐式视为类型 `long`。从类型 `long` 到类型 `int`，相反的转换是显式的，因此需要显式强制转换。

```
int a = 123;
long b = a;         // implicit conversion from int to long
int c = (int) b;    // explicit conversion from long to int
```

某些转换是由语言定义的。程序还可以定义自己的转换（用户定义的转换）。

隐式转换

以下转换归类为隐式转换：

- 标识转换
- 隐式数值转换
- 隐式枚举转换
- 隐式内插字符串转换
- 隐式可为 null 转换
- Null 文本转换
- 隐式引用转换
- 装箱转换
- 隐式动态转换
- 隐式常量表达式转换
- 用户定义的隐式转换
- 匿名函数转换
- 方法组转换

隐式转换可能会在多种情况下发生，包括函数成员调用（动态重载决策的编译时检查）、强制转换表达式（强制转换表达式）和赋值（赋值运算符）。

预定义的隐式转换始终会成功，并且永远不会引发异常。正确设计的用户定义隐式转换也应显示这些特征。

出于转换目的，`object` 和 `dynamic` 类型被视为等效类型。

但是，动态转换（隐式动态转换和显式动态转换）仅适用于 `dynamic` 类型（动态类型）的表达式。

标识转换

标识转换从任何类型转换为同一类型。此转换存在，因此，已具有所需类型的实体可被视为可转换为该类型。

- 因为 `object` 和 `dynamic` 被视为等效的，所以在 `object` 和 `dynamic` 之间存在标识转换，并且在用 `dynamic` 替换所有匹配项时，构造类型之间是相同的。

隐式数值转换

隐式数值转换为：

- 从 `sbyte` 到 `short`、`int`、`long`、`float`、`double` 或 `decimal`。
- 从 `byte` 到 `short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float`、`double` 或 `decimal`。
- 从 `short` 到 `int`、`long`、`float`、`double` 或 `decimal`。
- 从 `ushort` 到 `int`、`uint`、`long`、`ulong`、`float`、`double` 或 `decimal`。
- 从 `int` 到 `long`、`float`、`double` 或 `decimal`。
- 从 `uint` 到 `long`、`ulong`、`float`、`double` 或 `decimal`。
- 从 `long` 到 `float`、`double` 或 `decimal`。
- 从 `ulong` 到 `float`、`double` 或 `decimal`。
- 从 `char` 到 `ushort`、`int`、`uint`、`long`、`ulong`、`float`、`double` 或 `decimal`。
- 从 `float` 到 `double`。

从 `int`、`uint`、`long` 或 `ulong` 到 `float` 以及从 `long` 或 `ulong` 到 `double` 的转换可能会导致精度损失，但永远不会导致数量级损失。其他隐式数值转换不会丢失任何信息。

不存在到 `char` 类型的隐式转换，因此其他整型类型的值不会自动转换为 `char` 类型。

隐式枚举转换

隐式枚举转换允许*decimal_integer_literal* `0` 转换为任何*enum_type*，以及基础类型为*enum_type*的任何*nullable_type*。在后一种情况下，转换将通过转换为基础*enum_type*并包装结果（可以为 null 的类型）来进行计算。

隐式内插字符串转换

隐式内插字符串转换允许*interpolated_string_expression*（内插的字符串）转换为 `System.IFormattable` 或 `System.FormattableString`（实现 `System.IFormattable`）。

应用此转换时，字符串值不是由内插字符串组成的。相反，将创建一个 `System.FormattableString` 实例，如内插字符串中所述。

隐式可为 null 转换

对不可以为 null 的值类型进行操作的预定义隐式转换也可用于这些类型的可以为 null 的形式。对于从不可为 null 的值类型转换为不可以为 null 的值类型的每个预定义隐式标识和数值转换 `S` `T`，以下隐式可为 null 的转换：

- 从 `S?` 到 `T?` 的隐式转换。
- 从 `S` 到 `T?` 的隐式转换。

基于从 `S` 到 `T` 的基础转换计算隐式可为 null 的转换，如下所示：

- 如果可为 null 的转换来自 `S?` 到 `T?`：
  - 如果源值为 null（`HasValue` 属性为 false），则结果为类型 `T?` 的 null 值。
  - 否则，转换将作为从 `S?` 到 `S` 的解包进行计算，后跟从 `S` 到 `T` 的底层转换，后跟从 `T` 到 `T?` 的包装（可为 null 的类型）。
- 如果将可为 null 的转换来自 `S` 到 `T?`，则会将转换计算为从 `S` 到 `T` 后跟从 `T` 到 `T?` 的基础转换。

Null 文本转换

存在从 `null` 文本到任何可以为 null 的类型的隐式转换。此转换生成给定可以为 null 的类型的 null 值（可为null的类型）。

隐式引用转换

隐式引用转换为：

- 从任何*reference_type*到 `object` 和 `dynamic`。
- 从任何*class_type* `S` 到任何*class_type* `T`，提供的 `S` 派生自 `T`。

- 从任何*class_type* `S` 到任何*interface_type* `T`，提供 `S` 实现 `T`。
- 从任何*interface_type* `S` 到任何*interface_type* `T`，提供的 `S` 派生自 `T`。
- 在元素类型为的*array_type* `S` 中 `SE` 到元素类型 `T` 的*array_type* `TE`，前提是满足以下所有条件：
  - `S` 和 `T` 仅在元素类型上存在差异。换句话说，`S` 和 `T` 具有相同的维数。
  - `SE` 和 `TE` 是*reference_type*。
  - 存在从 `SE` 到 `TE` 的隐式引用转换。
- 从任何*array_type*到 `System.Array` 及其实现的接口。
- 从一维数组类型 `S[]` 到 `System.Collections.Generic.IList<T>` 及其基接口，前提是存在从 `S` 到 `T` 的隐式标识或引用转换。
- 从任何*delegate_type*到 `System.Delegate` 及其实现的接口。
- 从 null 文本到任何*reference_type*。
- 从任何*reference_type*到*reference_type* `T` 如果它具有隐式标识或到*reference_type* `T0` 的引用转换，并且 `T0` 的标识转换为 `T`。
- 从任何*reference_type*到接口或委托类型 `T` 如果它具有隐式标识或到接口或委托类型的引用转换 `T0` 并且 `T0` 可转换为 `T` 的变体（差异转换）。
- 涉及称为引用类型的类型参数的隐式转换。有关涉及类型参数的隐式转换的更多详细信息，请参阅涉及类型参数的隐式转换。

隐式引用转换是*reference_type*之间的转换，这些转换可证明始终成功，因此不需要在运行时进行检查。

引用转换、隐式或显式转换决不会更改正在转换的对象的引用标识。换言之，虽然引用转换可以更改引用的类型，但它不会更改引用的对象的类型或值。

装箱转换

装箱转换允许*value_type*隐式转换为引用类型。存在从任何*non_nullable_value_type*到 `object` 和 `dynamic` 的装箱转换，以 `System.ValueType` 和*interface_type*实现的任何*non_nullable_value_type* 。此外，可以将*enum_type*转换为类型 `System.Enum`。

如果且仅当存在从基础*non_nullable_value_type*到引用类型的装箱转换，则从*nullable_type*到引用类型的装箱转换。

如果某个值类型具有到接口类型的装箱转换，则该值类型具有到该接口类型的装箱转换 `I` `I0` 并且 `I0` 具有到 `I` 的标识转换。

如果某个值类型具有到接口类型的装箱转换或委托类型的装箱转换，则该值类型会将其装箱转换 `I` `I0` 并 `I0` 区分方差（变化转换） `I`。

将*non_nullable_value_type*的值装箱包括分配对象实例并将*value_type*值复制到该实例中。结构可以装箱到类型 `System.ValueType`，因为这是所有结构（继承）的基类。

将*nullable_type*的值装箱将继续执行以下操作：

- 如果源值为 null（ `HasValue` 属性为 false），则结果为目标类型的空引用。
- 否则，结果是对通过解包和装箱源值生成的装箱 `T` 的引用。

装箱转换中进一步介绍了装箱转换。

隐式动态转换

存在从类型 `dynamic` 到任何类型 `T` 的表达式的隐式动态转换。转换是动态绑定的（动态绑定），这意味着将在运行时从表达式的运行时类型中查找隐式转换，以便 `T`。如果未找到任何转换，则会引发运行时异常。

请注意，此隐式转换似乎违反了隐式转换开始时的建议，隐式转换应永远不会引发异常。但它不是转换本身，而是*查找*导致异常的转换。运行时异常的风险在使用动态绑定时是固有的。如果不需要转换的动态绑定，则可以先将表达式转换为 `object`，然后转换为所需的类型。

下面的示例阐释了隐式动态转换：

```
object o  = "object"
dynamic d = "dynamic";

string s1 = o; // Fails at compile-time -- no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i     = d; // Compiles but fails at run-time -- no conversion exists
```

`s2` 和 `i` 的分配都采用隐式动态转换，在这种情况下，将在运行时暂停操作的绑定。在运行时，隐式转换是从 `d` -- `string` --到目标类型的运行时类型中查找的。找到 `string` 但不能 `int` 的转换。

隐式常量表达式转换

隐式常数表达式转换允许以下转换：

- 如果 `short` 的值在目标类型的范围内，则可以将 `int` 类型的*constant_expression*（常数表达式）转换为类型 `sbyte`、`byte`、`ushort`、`uint`、`ulong` 或 constant_expression。
- `long` 类型的*constant_expression*可转换为类型 `ulong`，前提是*constant_expression*的值不是负数。

涉及类型参数的隐式转换

`T` 的给定类型参数存在以下隐式转换：

- 从 `T` 到其有效的基类 `C`，从 `T` 到 `C` 的任何基类，从 `T` 到 `C` 实现的任何接口。在运行时，如果 `T` 是值类型，则转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 `T` 到接口类型 `I` 在 `T` 的有效接口集中，并从 `T` 到 `I` 的任何基接口。在运行时，如果 `T` 是值类型，则转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 `T` 到类型形参 `U`，提供 `T` 取决于 `U`（类型形参约束）。在运行时，如果 `U` 是值类型，则 `T` 和 `U` 都必须是相同的类型，并且不执行任何转换。否则，如果 `T` 是值类型，则转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 null 文本到 `T`，前提是已知 `T` 为引用类型。
- 从 `T` 到引用类型 `I` 如果它具有到引用类型的隐式转换 `S0` 并且 `S0` 具有到 `S` 的标识转换。在运行时，转换的执行方式与转换到 `S0` 的方式相同。
- 从 `T` 到接口类型 `I` 如果它具有到接口或委托类型的隐式转换 `I0` 并且 `I0` 可以转换为 `I`（差异转换）。在运行时，如果 `T` 是值类型，则转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。

如果已知 `T` 是引用类型（类型参数约束），则上述转换全都归类为隐式引用转换（隐式引用转换）。如果 `T` 不知道是引用类型，则上述转换归类为装箱转换（装箱转换）。

用户定义的隐式转换

用户定义的隐式转换包括一个可选的标准隐式转换，然后执行用户定义的隐式转换运算符，然后执行另一个可选的标准隐式转换。用于评估用户定义的隐式转换的确切规则在处理用户定义的隐式转换中进行了介绍。

匿名函数转换和方法组转换

匿名函数和方法组本身没有类型，但可能会隐式转换为委托类型或表达式树类型。匿名函数转换和方法组转换中的方法组转换更详细地介绍了匿名函数转换。

显式转换

以下转换归类为显式转换：

- 所有隐式转换。
- 显式数值转换。
- 显式枚举转换。

- 可以为 null 的显式转换。
- 显式引用转换。
- 显式接口转换。
- 取消装箱转换。
- 显式动态转换
- 用户定义的显式转换。

显式转换可以出现在强制转换表达式中(强制转换表达式)。

显式转换集包括所有隐式转换。这意味着允许冗余强制转换表达式。

不是隐式转换的显式转换是转换，无法证明始终成功，已知的转换可能会丢失信息，并且跨类型域的转换非常不同于显式图解.

显式数值转换

显式数值转换是指从*numeric_type*到另一个*numeric_type*的转换，隐式数值转换(隐式数值转换)尚不存在：

- 从 `sbyte` 到 `byte`、`ushort`、`uint`、`ulong` 或 `char`。
- 从 `byte` 到 `sbyte` 和 `char`。
- 从 `short` 到 `sbyte`、`byte`、`ushort`、`uint`、`ulong` 或 `char`。
- 从 `ushort` 到 `sbyte`、`byte`、`short` 或 `char`。
- 从 `int` 到 `sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong` 或 `char`。
- 从 `uint` 到 `sbyte`、`byte`、`short`、`ushort`、`int` 或 `char`。
- 从 `long` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`ulong` 或 `char`。
- 从 `ulong` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long` 或 `char`。
- 从 `char` 到 `sbyte`、`byte` 或 `short`。
- 从 `float` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char` 或 `decimal`。
- 从 `double` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float` 或 `decimal`。
- 从 `decimal` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float` 或 `double`。

由于显式转换包括所有隐式和显式数字转换，因此始终可以使用强制转换表达式(强制转换表达式)将任何*numeric_type*转换为任何其他*numeric_type* 。

显式数值转换可能会丢失信息，或可能导致引发异常。显式数值转换的处理方式如下：

- 对于从整型转换为另一整型类型的转换，处理过程取决于发生转换的溢出检查上下文(检查和未检查的运算符)：
  - 在 `checked` 上下文中，如果源操作数的值在目标类型的范围内，则转换成功；但如果源操作数的值超出了目标类型的范围，则会引发 `System.OverflowException`。
  - 在 `unchecked` 上下文中，转换始终会成功，并按如下所示继续。
    - 如果源类型大于目标类型，则通过放弃其“额外”最高有效位来截断源值。结果会被视为目标类型的值。
    - 如果源类型小于目标类型，则源值是符号扩展或零扩展，以使其与目标类型的大小相同。如果源类型带符号，则是符号扩展；如果源类型是无符号的，则是零扩展。结果会被视为目标类型的值。
    - 如果源类型与目标类型的大小相同，则源值将被视为目标类型的值。
- 对于从 `decimal` 到整数类型的转换，将源值向零舍入到最接近的整数值，并且此整数值将成为转换的结果。如果生成的整数值超出了目标类型的范围，则会引发 `System.OverflowException`。
- 对于从 `float` 或 `double` 到整数类型的转换，处理取决于发生转换的溢出检查上下文(检查和未检查的运算符)：
  - 在 `checked` 上下文中，转换过程如下所示：

    - 如果操作数的值为 NaN 或无穷大，则会引发 `System.OverflowException`。
    - 否则，源操作数向零舍入到最接近的整数值。如果此整数值在目标类型的范围内，则此值为转换的结果。
    - 否则，将会引发 `System.OverflowException`。
  - 在 `unchecked` 上下文中，转换始终会成功，并按如下所示继续。
    - 如果操作数的值为 NaN 或无穷大，则转换的结果是目标类型的未指定值。
    - 否则，源操作数向零舍入到最接近的整数值。如果此整数值在目标类型的范围内，则此值为转换的结果。
    - 否则，转换的结果是目标类型的未指定值。
- 对于从 `double` 到 `float` 的转换，`double` 值舍入为最接近的 `float` 值。如果 `double` 值太小而无法表示为 `float`，则结果将变为零或负零。如果 `double` 值太大而无法表示为 `float`，则结果将变为正无穷或负无穷。如果 `double` 值为 NaN，则结果也为 NaN。
- 对于从 `float` 或 `double` 到 `decimal` 的转换，将源值转换为 `decimal` 表示形式，并在需要时舍入到第28位小数后最接近的数(decimal 类型)。如果源值太小而无法表示为 `decimal`，则结果将变为零。如果源值为 NaN、无限大或太大而无法表示为 `decimal`，则会引发 `System.OverflowException`。
- 对于从 `decimal` 到 `float` 或 `double` 的转换，`decimal` 值舍入为最接近的 `double` 或 `float` 值。虽然这种转换可能会丢失精度，但它永远不会引发异常。

显式枚举转换

显式枚举转换为：

- 从 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double` 或 `decimal` 到任何 *enum_type*。
- 从任何 *enum_type* 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double` 或 `decimal`。
- 从任何 *enum_type* 到任何其他 *enum_type*。

处理两种类型之间的显式枚举转换的方式是将任何参与的 *enum_type* 视为 *enum_type* 的基础类型，然后在生成的类型之间执行隐式或显式数字转换。例如，给定 *enum_type* `E`，以及基础类型 `int`，则 `E` 从 `byte` 到 `int` 的显式数字转换(从 `byte` 到 `byte` 的显式数字转换处理)，并将从 `E` 到 `byte` 的转换处理为从 `int` 到的隐式数值转换(隐式数值转换)。

显式可为 null 的转换

*显式可以为 null 的转换*允许对不可以为 null 的值类型进行操作的预定义显式转换也可用于这些类型的可以为 null 的形式。对于从不可为 null 的值类型转换为不可以为 null 的值类型的每个预定义的显式转换 `S` `T`（标识转换、隐式数值转换、隐式枚举转换、显式数字转换和显式枚举转换），可以为 null 的转换如下：

- 从 `S?` 到 `T?` 的显式转换。
- 从 `S` 到 `T?` 的显式转换。
- 从 `S?` 到 `T` 的显式转换。

基于从 `S` 到 `T` 的基础转换计算可以为 null 的转换，如下所示：

- 如果可为 null 的转换来自 `S?` 到 `T?`：
  - 如果源值为 null（`HasValue` 属性为 false），则结果为类型 `T?` 的 null 值。
  - 否则，转换将作为从 `S?` 到 `S` 的解包进行计算，后跟从 `S` 到 `T` 的底层转换，然后是从 `T` 到 `T?` 的包装。
- 如果将可为 null 的转换来自 `S` 到 `T?`，则会将转换计算为从 `S` 到 `T` 后跟从 `T` 到 `T?` 的基础转换。
- 如果将可为 null 的转换来自 `S?` 到 `T`，则会将转换计算为从 `S?` 到 `S` 的解包，后跟从 `S` 到 `T` 的基础转换。

请注意，如果值为 `null`，则尝试解包的值会引发异常。

显式引用转换

显式引用转换为：

- 从 `object` 和 `dynamic` 到任何其他*reference_type*。
- 从任何*class_type* `S` 到任何*class_type* `T`，提供的 `S` 是 `T` 的基类。
- 从任何*class_type* `S` 到任何*interface_type* `T`，提供的 `S` 不是密封的，`S` 未实现 `T`。
- 从任何*interface_type* `S` 到任何*class_type* `T`，提供的 `T` 未密封或未提供 `T` 实现 `S`。
- 从任何*interface_type* `S` 到任何*interface_type* `T`，提供的 `S` 不是从 `T` 派生的。
- 在元素类型为的*array_type* `S` 中 `SE` 到元素类型 `T` 的*array_type* `TE`，前提是满足以下所有条件：
  - `S` 和 `T` 仅在元素类型上存在差异。换句话说，`S` 和 `T` 具有相同的维数。
  - `SE` 和 `TE` 是*reference_type*。
  - 存在从 `SE` 到 `TE` 的显式引用转换。
- 从 `System.Array` 及其实现的接口到任何*array_type*。
- 从一维数组类型 `S[]` 到 `System.Collections.Generic.IList<T>` 及其基接口，前提是存在从 `S` 到 `T` 的显式引用转换。
- 从 `System.Collections.Generic.IList<S>` 及其基接口到一维数组类型 `T[]`，前提是有显式标识或从 `S` 到 `T` 的引用转换。
- 从 `System.Delegate` 及其实现的接口到任何*delegate_type*。
- 从引用类型到引用类型 `T` 如果它具有到引用类型的显式引用转换 `T0` 并且 `T0` 具有 `T` 的标识转换。
- 从引用类型到接口或委托类型 `T` 如果它具有到接口或委托类型的显式引用转换 `T0` 并且 `T0` 可以转换为 `T` 或 `T` 变体转换为 `T0`（变体转换）。
- 从 `D<S1...Sn>` 到 `D<T1...Tn>`，其中 `D<X1...Xn>` 是一个泛型委托类型，`D<S1...Sn>` 与 `D<T1...Tn>` 不兼容或等同于 `Xi`，而对于每个类型参数 `D`，则以下保留：
  - 如果 `Xi` 固定，则 `Si` 与 `Ti` 相同。
  - 如果 `Xi` 是协变的，则存在从 `Si` 到 `Ti` 的隐式或显式标识或引用转换。
  - 如果 `Xi` 为逆变，则 `Si` 和 `Ti` 均为相同或同时为这两个引用类型。
- 涉及称为引用类型的类型参数的显式转换。有关涉及类型参数的显式转换的详细信息，请参阅涉及类型参数的显式转换。

显式引用转换是需要运行时检查以确保它们正确的引用类型之间的转换。

若要在运行时成功进行显式引用转换，源操作数的值必须为 `null`，或源操作数引用的对象的实际类型必须是可通过隐式引用转换（隐式引用转换）或装箱转换（装箱转换）转换为目标类型的类型。如果显式引用转换失败，则会引发 `System.InvalidCastException`。

引用转换、隐式或显式转换决不会更改正在转换的对象的引用标识。换言之，虽然引用转换可以更改引用的类型，但它不会更改引用的对象的类型或值。

取消装箱转换

取消装箱转换允许将引用类型显式转换为*value_type*。从类型 `object`、`dynamic` 和 `System.ValueType` 到任何*non_nullable_value_type*，以及从任何*interface_type*到实现*non_nullable_value_type*的任意*interface_type*的取消装箱转换。此外，可以将 `System.Enum` 类型取消装箱到任何*enum_type*。

如果从引用类型到*nullable_type*的基础*non_nullable_value_type*的取消装箱转换存在，则取消装箱转换将从引用类型转换为*nullable_type*。

值类型 `S` 具有从接口类型的取消装箱转换 `I` 如果它具有从接口类型的取消装箱转换 `I0` 并且 `I0` 具有到 `I` 的标识转换。

值类型 `S` 具有从接口类型的取消装箱转换 `I` 如果该类型具有从接口或委托类型的取消装箱转换 `I0` 并且 `I0` 可以转换为 `I` 或 `I` 可转换为 `I0`（变体转换）。

取消装箱操作包括：首先检查对象实例是否是给定*value_type*的装箱值，然后将值复制到该实例之外。取消装箱对*nullable_type*的空引用将生成*nullable_type*的 null 值。结构可以从类型 `System.ValueType` 取消装箱，因为这是所有结构（继承）的基类。

取消装箱转换中进一步介绍了取消装箱转换。

显式动态转换

存在从 `dynamic` 类型的表达式到任何类型 `T` 的显式动态转换。转换是动态绑定的（动态绑定），这意味着将在运行时从表达式的运行时类型中查找显式转换，以便 `T`。如果未找到任何转换，则会引发运行时异常。

如果不需要转换的动态绑定，则可以先将表达式转换为 `object`，然后转换为所需的类型。

假定定义了下面的类：

```
class C
{
    int i;

    public C(int i) { this.i = i; }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

下面的示例阐释了显式动态转换：

```
object o  = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

在编译时可以找到 `o` 到 `C` 的最佳转换，以便成为显式引用转换。这会在运行时失败，因为 `"1"` 事实上并不是 `C`。`d` 转换为 `C` 但是，作为显式动态转换，会将其挂起到运行时，其中，用户从 `d` -- `string` 到 `C` 的运行时类型进行了定义的转换，并成功了。

涉及类型参数的显式转换

`T` 的给定类型参数存在以下显式转换：

- 从有效的基类 `C` 的 `T` 到 `T` 和从 `C` 的任何基类到 `T`。在运行时，如果 `T` 是值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从任何接口类型到 `T`。在运行时，如果 `T` 是值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从 `T` 到任何*interface_type* `I` 如果尚未从 `T` 到 `I` 的隐式转换。在运行时，如果 `T` 是值类型，则转换将作为装箱转换执行，后跟显式引用转换。否则，转换将作为显式引用转换或标识转换执行。
- 从类型参数 `U` 到 `T`，提供 `T` 依赖于 `U`（类型参数约束）。在运行时，如果 `U` 是值类型，则 `T` 和 `U` 都必须是相同的类型，并且不执行任何转换。否则，如果 `T` 是值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。

如果已知 `T` 是引用类型，则上述转换全都归类为显式引用转换（显式引用转换）。如果 `T` 不知道是引用类型，则上述转换归类为取消装箱转换（取消装箱转换）。

以上规则不允许从不受约束的类型形参直接转换为非接口类型，这可能会令人吃惊。此规则的原因是为了防止混淆并使此类转换的语义清晰。例如，请考虑以下声明：

```
class X<T>
{
    public static long F(T t) {
        return (long)t;               // Error
    }
}
```

如果允许 `t` 到 `int` 的直接显式转换，则很可能会认为 `X<int>.F(7)` 会返回 `7L`。但是，它不会这样，因为仅当已知类型在绑定时是数字时才考虑标准数字转换。为了使语义清晰明了，必须改为编写以上示例：

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;       // Ok, but will only work when T is long
    }
}
```

此代码现在将进行编译，但执行 `X<int>.F(7)` 稍后会在运行时引发异常，因为无法直接将装箱 `int` 转换为 `long`。

用户定义的显式转换

用户定义的显式转换包含一个可选的标准显式转换，然后执行用户定义的隐式或显式转换运算符，然后执行另一个可选的标准显式转换。用于评估用户定义的显式转换的确切规则在处理用户定义的显式转换中进行了介绍。

标准转换

标准转换是那些预定义的转换，这些转换可作为用户定义的转换的一部分出现。

标准隐式转换

以下隐式转换归类为标准隐式转换：

- 标识转换（标识转换）
- 隐式数值转换（隐式数值转换）
- 隐式可为 null 转换（隐式可为 null 转换）
- 隐式引用转换（隐式引用转换）
- 装箱转换（装箱转换）
- 隐式常量表达式转换（隐式动态转换）
- 涉及类型参数的隐式转换（涉及类型参数的隐式转换）

标准隐式转换专门排除用户定义的隐式转换。

标准显式转换

标准显式转换均为标准隐式转换，以及与之相反的标准隐式转换的显式转换的子集。换言之，如果存在从类型 `A` 到类型 `B` 的标准隐式转换，则从类型 `A` 到类型 `B`，并从类型 `B` 到类型 `A`，都存在标准的显式转换。

用户定义的转换

C#允许使用*用户定义的转换*来扩充预定义的隐式和显式转换。用户定义的转换通过在类和结构类型中声明转换运算符（转换运算符）引入。

允许的用户定义的转换

C#仅允许声明某些用户定义的转换。特别是，不能重新定义已存在的隐式或显式转换。

对于给定的源类型 `S` 和目标类型 `T`，如果 `S` 或 `T` 是可以为 null 的类型，则允许 `S0` 和 `T0` 引用它们的基础类型，否则 `S0` 和 `T0` 分别等于 `S` 和 `T`。仅当满足以下所有条件时，才允许类或结构声明从源类型 `S` 转换为

目标类型 `T`：

- `S0` 和 `T0` 属于不同类型。
- `S0` 或 `T0` 是发生运算符声明的类或结构类型。
- `S0` 和 `T0` 都不是*interface_type*的。
- 如果不包括用户定义的转换，则从 `S` 到 `T` 或从 `T` 到 `S` 的转换不存在。

适用于用户定义的转换的限制将在转换运算符中进一步讨论。

提升的转换运算符

给定一个用户定义的转换运算符，该运算符将不可以为 null 的值类型 `S` 转换为不可以为 null 的值类型 `T`，存在从 `S?` 转换为 `T?` 的***提升转换运算符***。此提升转换运算符执行从 `S?` 解包到 `S`，然后执行从 `S` 到 `T` 的用户定义的转换，后跟从 `T` 到 `T?` 的包装，只不过 null 值 `S?` 直接转换为 null 值 `T?`。

提升的转换运算符与其基础用户定义转换运算符具有相同的隐式或显式分类。"用户定义的转换" 一词适用于用户定义的转换运算符和提升的转换运算符。

计算用户定义的转换

用户定义的转换将值从其类型（称为***源类型***）转换为另一种类型，称为***目标类型***。计算用户定义的转换中心，查找特定源和目标类型的***最特定***用户定义转换运算符。此确定分为几个步骤：

- 查找要将用户定义的转换运算符视为其中的一组类和结构。此集由源类型及其基类和目标类型及其基类组成（隐式假设只有类和结构可以声明用户定义的运算符，而非类类型没有基类）。对于此步骤，如果源类型或目标类型为*nullable_type*，则改为使用它们的基础类型。
- 从该类型集中确定哪些用户定义的转换运算符适用。要使转换运算符适用，必须可以执行标准转换（标准转换），使其从源类型转换为运算符的操作数类型，并且必须能够执行从运算符的结果类型到目标类型的标准转换。
- 从一组适用的用户定义的运算符，确定哪个运算符明确是最具体的。一般来说，最特定的运算符是操作数类型与源类型 "最接近" 的运算符，其结果类型与目标类型 "最近"。用户定义的转换运算符优先于提升转换运算符。以下部分定义了用于建立最具体的用户定义转换运算符的确切规则。

确定了最特定的用户定义转换运算符后，用户定义的转换的实际执行操作包括多达三个步骤：

- 首先，如果需要，执行从源类型到用户定义或提升的转换运算符的操作数类型的标准转换。
- 接下来，调用用户定义或提升的转换运算符来执行转换。
- 最后，如果需要，执行从用户定义转换运算符的结果类型到目标类型的标准转换。

用户定义的转换的计算从不涉及多个用户定义的转换运算符或提升的转换运算符。换句话说，从类型 `S` 到类型 `T` 的转换永远不会首先执行从 `S` 到 `X` 的用户定义的转换，然后执行从 `X` 到 `T` 的用户定义的转换。

以下各节提供了用户定义的隐式或显式转换的确切计算定义。定义使用以下术语：

- 如果从类型 `A` 到类型 `B` 存在标准隐式转换（标准隐式转换），并且 `A` 和 `B` 都不*interface_type*，则 `A` 被视为***包含*** `B`，`B` 称为***包含*** `A`。
- 一组类型中包含的***最大的类型***是一种包含集内所有其他类型的类型。如果没有一种类型包含所有其他类型，则该集没有最大的包含类型。更直观地说，最包含的类型是集中的 "最大" 类型，每个其他类型都可以隐式转换为一种类型。
- 类型集中***最常被包含的类型***是一种类型，它由集中的所有其他类型所包含。如果所有其他类型都不包含任何一种类型，则该集不包含大多数被包含的类型。更直观地说，最包含的类型是集中的 "最小" 类型，这种类型可以隐式转换为其他类型。

处理用户定义的隐式转换

用户定义的从类型 `S` 到类型的隐式转换 `T` 的处理方式如下：

- 确定 `S0` 和 `T0` 的类型。如果 `S` 或 `T` 是可以为 null 的类型，`S0` 和 `T0` 是其基础类型，则 `S0` 和 `T0` 分别

等于 `S` 和 `T`。

- 查找要将用户定义的转换运算符视为 `D` 类型集。此集由 `S0`（如果 `S0` 为类或结构）、`S0` 的基本类（如果 `S0` 是类）和 `T0`（如果 `T0` 是类或结构）。
- 查找一组适用的用户定义的转换运算符，`U`。此集由由 `D` 中的类或结构声明的用户定义的和提升的隐式转换运算符组成，它们从包含 `S` 的类型转换为 `T` 包含的类型。如果 `U` 为空，则转换未定义，并发生编译时错误。
- 查找 `U` 中运算符的最特定的源类型 `SX`：
  - 如果 `U` 中的任何运算符从 `S` 转换，则 `SX` 为 `S`。
  - 否则，`SX` 是 `U` 中运算符的组合源类型集中最常包含的类型。如果找不到完全包含的类型，则转换是不明确的，并发生编译时错误。
- 查找 `U` 中运算符的最特定目标类型 `TX`：
  - 如果 `U` 中的任何运算符转换为 `T`，则 `TX` 为 `T`。
  - 否则，`TX` 是 `U` 中运算符的组合目标类型集中最包含的类型。如果找不到完全包含的类型，则转换是不明确的，并发生编译时错误。
- 查找最具体的转换运算符：
  - 如果 `U` 只包含一个从 `SX` 转换为 `TX` 的用户定义的转换运算符，则这是最具体的转换运算符。
  - 否则，如果 `U` 只包含一个从 `SX` 转换为 `TX` 的提升转换运算符，则这是最具体的转换运算符。
  - 否则，转换是不明确的，并发生编译时错误。
- 最后，应用转换：
  - 如果未 `SX``S`，则执行从 `S` 到 `SX` 的标准隐式转换。
  - 调用最特定的转换运算符，以将 `SX` 转换为 `TX`。
  - 如果未 `T``TX`，则执行从 `TX` 到 `T` 的标准隐式转换。

处理用户定义的显式转换

用户定义的从类型 `S` 到类型 `T` 的显式转换的处理方式如下：

- 确定 `S0` 和 `T0` 的类型。如果 `S` 或 `T` 是可以为 null 的类型，`S0` 和 `T0` 是其基础类型，则 `S0` 和 `T0` 分别等于 `S` 和 `T`。
- 查找要将用户定义的转换运算符视为 `D` 类型集。此集由 `S0`（如果 `S0` 为类或结构）、`S0`（如果 `S0` 是类）的基类、`T0`（如果 `T0` 是类或结构）以及 `T0`（如果 `T0` 是类）的基类。
- 查找一组适用的用户定义的转换运算符，`U`。此集由 `D` 中的类或结构声明的用户定义的隐式或显式转换运算符，由 `S` 转换为包含 `T` 包含或包含的类型。如果 `U` 为空，则转换未定义，并发生编译时错误。
- 查找 `U` 中运算符的最特定的源类型 `SX`：
  - 如果 `U` 中的任何运算符从 `S` 转换，则 `SX` 为 `S`。
  - 否则，如果 `U` 中的任何运算符从包含 `S` 的类型进行转换，则 `SX` 是这些运算符的一组源类型中包含程度最高的类型。如果找不到最能包含的类型，则转换是不明确的，并发生编译时错误。
  - 否则，`SX` 是 `U` 中运算符的组合源类型集中的最包含的类型。如果找不到完全包含的类型，则转换是不明确的，并发生编译时错误。
- 查找 `U` 中运算符的最特定目标类型 `TX`：
  - 如果 `U` 中的任何运算符转换为 `T`，则 `TX` 为 `T`。
  - 否则，如果 `U` 中的任何运算符转换为 `T` 包含的类型，则在这些运算符的组合目标类型集中，`TX` 是最包含的类型。如果找不到完全包含的类型，则转换是不明确的，并发生编译时错误。
  - 否则，`TX` 是 `U` 中运算符的组合目标类型集中最常被包含的类型。如果找不到最能包含的类型，则转换是不明确的，并发生编译时错误。
- 查找最具体的转换运算符：
  - 如果 `U` 只包含一个从 `SX` 转换为 `TX` 的用户定义的转换运算符，则这是最具体的转换运算符。
  - 否则，如果 `U` 只包含一个从 `SX` 转换为 `TX` 的提升转换运算符，则这是最具体的转换运算符。
  - 否则，转换是不明确的，并发生编译时错误。

- 最后，应用转换：
  - 如果未 `SX``S`，则执行从 `S` 到 `SX` 的标准显式转换。
  - 调用最特定的用户定义转换运算符，以将 `SX` 转换为 `TX`。
  - 如果未 `T``TX`，则执行从 `TX` 到 `T` 的标准显式转换。

匿名函数转换

*Anonymous_method_expression*或*lambda_expression*归类为匿名函数（匿名函数表达式）。表达式没有类型，但可以隐式转换为兼容的委托类型或表达式树类型。具体而言，`F` 的匿名函数与 `D` 提供的委托类型兼容：

- 如果 `F` 包含*anonymous_function_signature*，则 `D` 和 `F` 具有相同数量的参数。
- 如果 `F` 不包含*anonymous_function_signature*，则 `D` 可能具有零个或多个任意类型的参数，前提是没有 `D` 的参数具有 `out` 参数修饰符。
- 如果 `F` 具有显式类型化参数列表，则 `D` 中的每个参数都具有与 `F` 中的相应参数相同的类型和修饰符。
- 如果 `F` 具有隐式类型参数列表，则 `D` 没有 `ref` 或 `out` 参数。
- 如果 `F` 的主体是一个表达式，并且 `D` 具有 `void` 返回类型或 `F` 为 async 并且 `D` 具有返回类型 `Task`，则当 `F` 的每个参数都给定 `D` 中的相应参数的类型时，`F` 的主体将被视为*statement_expression*（expression 语句）。
- 如果 `F` 的主体是一个语句块，并且 `D` 具有 `void` 返回类型或 `F` 为 async 并且 `D` 具有返回类型 `Task`，则当 `F` 的每个参数都给定 `D` 中对应参数的类型时，`F` 的主体为有效语句块（wrt块），其中没有 `return` 语句指定表达式。
- 如果 `F` 的主体是一个表达式，*并且* `F` *为非 async 并且* `D` 具有非 void 返回类型 `T`，*或* `F` 为 async 并且 `D` 具有返回类型 `Task<T>`，则当 `F` 的每个参数都给定 `D` 中对应参数的类型时，`F` 的主体是可隐式转换为 `T` 的有效表达式（wrt表达式）。
- 如果 `F` 的主体是一个语句块，*并且* `F` *为非 async 并且* `D` 具有非 void 返回类型 `T`，*或* `F` 为 async 并且 `D` 具有返回类型 `Task<T>`，则当 `F` 的每个参数都给定 `D` 中对应参数的类型时，`F` 的主体为有效语句块（wrt块），其中每个 `return` 语句指定一个可隐式转换为 `T` 的表达式。

为了简洁起见，本部分使用了任务类型 `Task` 和 `Task<T>`（异步函数）的缩写形式。

如果 `F` 与 `D` 的委托类型兼容，则 lambda 表达式 `F` 与表达式树类型兼容 `Expression<D>`。请注意，这不适用于匿名方法，只适用于 lambda 表达式。

某些 lambda 表达式不能转换为表达式树类型：即使转换*存在*，它也会在编译时失败。如果 lambda 表达式为，则为这种情况：

- 具有*块*体
- 包含简单赋值运算符或复合赋值运算符
- 包含动态绑定的表达式
- 为异步

下面的示例使用泛型委托类型 `Func<A,R>` 它表示一个函数，该函数采用 `A` 类型的参数并返回 `R` 类型的值：

```
delegate R Func<A,R>(A arg);
```

在分配中

```
Func<int,int> f1 = x => x + 1;               // Ok

Func<int,double> f2 = x => x + 1;            // Ok

Func<double,int> f3 = x => x + 1;            // Error

Func<int, Task<int>> f4 = async x => x + 1;   // Ok
```

每个匿名函数的参数和返回类型都是从匿名函数所分配到的变量类型确定的。

第一个分配成功将匿名函数转换为委托类型 `Func<int,int>` 因为当 `x` 提供类型 `int` 时，`x+1` 是可隐式转换为类型 `int` 的有效表达式。

同样，第二个赋值语句会成功将匿名函数转换为委托类型 `Func<int,double>` 因为 `x+1`（类型 `int`）的结果可隐式转换为类型 `double`。

但是，第三个赋值是编译时错误，因为当 `double``x` 给定类型时，`x+1` 的结果（类型 `double`）无法隐式转换为类型 `int`。

第四个赋值成功将匿名 async 函数转换为委托类型 `Func<int, Task<int>>` 因为 `x+1`（类型 `int`）的结果可隐式转换为任务类型 `Task<int>` 的结果类型 `int`。

匿名函数可能会影响重载决策，并参与类型推理。有关更多详细信息，请参阅函数成员。

匿名函数转换到委托类型的计算

将匿名函数转换为委托类型会生成一个委托实例，该实例引用匿名函数和在计算时处于活动状态的已捕获外部变量的（可能为空）集。调用委托时，将执行匿名函数的主体。使用委托引用的捕获外部变量集来执行正文中的代码。

从匿名函数生成的委托的调用列表包含单个项。委托的确切目标对象和目标方法未指定。具体而言，它不指定是 `null` 委托的目标对象、封闭函数成员的 `this` 值还是其他某个对象。

允许（但不要求）将语义完全相同的匿名函数转换为同一委托类型（但不是必需的），以返回相同的委托实例。此处使用的术语在语义上是相同的，这意味着在所有情况下，匿名函数的执行将在给定相同参数的情况下生成相同的效果。此规则允许对如下代码进行优化。

```
delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

由于两个匿名函数委托具有相同的（空）捕获的外部变量集，而且由于匿名函数在语义上是相同的，因此编译器允许委托引用相同的目标方法。的确，允许编译器从两个匿名函数表达式返回完全相同的委托实例。

对表达式树类型的匿名函数转换的计算

将匿名函数转换为表达式树类型会生成一个表达式树（表达式树类型）。更准确地说，对匿名函数转换的评估导致了表示匿名函数本身的结构的对象结构。表达式树的准确结构以及用于创建它的确切过程是定义的实现。

实现示例

本部分介绍了可能实现的匿名函数转换(其他C#构造)。此处所述的实现基于 Microsoft C#编译器所使用的相同原则，但它并不是强制性实现，也不是唯一可行的方法。它只是简单地提到了转换为表达式树，因为其确切语义超出了本规范的范围。

本部分的其余部分提供了一些代码示例，其中包含具有不同特征的匿名函数。对于每个示例，提供了仅使用其他C#构造的代码的相应转换。在示例中，假设标识符 `D` 表示以下委托类型：

```
public delegate void D();
```

匿名函数的最简单形式是捕获无外部变量：

```
class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}
```

这可以转换为引用编译器生成的静态方法(在其中放置匿名函数的代码)的委托实例化：

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }

    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

在下面的示例中，匿名函数引用 `this` 的实例成员：

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

这可以转换为包含匿名函数代码的编译器生成的实例方法：

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```

在此示例中，匿名函数捕获本地变量：

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

局部变量的生存期现在必须至少扩展到匿名函数委托的生存期。这可以通过将本地变量 "提升" 到编译器生成的类的字段来实现。局部变量的实例化（局部变量的实例化）随后对应于创建编译器生成的类的实例，并且访问本地变量对应于访问编译器生成的类的实例中的字段。此外，匿名函数会成为编译器生成的类的实例方法：

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

最后，下面的匿名函数捕获 `this`，以及两个具有不同生存期的局部变量：

```
class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```

在这里，将为每个在其中捕获局部变量的语句块创建一个编译器生成的类，以便不同块中的局部变量可以具有独立生存期。`__Locals2` 的实例（内部语句块的编译器生成的类）包含局部变量 `z` 和引用 `__Locals1` 的实例的字段。`__Locals1` 的实例（外部语句块的编译器生成的类）包含局部变量 `y` 和引用封闭函数成员 `this` 的字段。利用这些数据结构，可以通过 `__Local2` 的实例访问所有捕获的外部变量，并且可以将匿名函数的代码实现为该类的实例方法。

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }

    class __Locals1
    {
        public Test __this;
        public int y;
    }

    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;

        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}
```

在此处用于捕获局部变量的相同技术也可以在将匿名函数转换为表达式树时使用：对编译器生成的对象的引用可以存储在表达式树中，对局部变量的访问可以是表示为这些对象上的字段访问。此方法的优点是允许在委托和表达式树之间共享 "提升" 的局部变量。

方法组转换

存在从方法组（表达式分类）到兼容委托类型的隐式转换（隐式转换）。给定委托类型 `D` 和归类为方法组的表达式 `E` 时 `D` `E`，如果 `E` 至少包含一个适用于其正常窗体（适用函数成员）的方法，该方法可用于通过使用 `D` 的参数类型和修饰符构造的参数列表，如下所述。

以下中描述了从方法组 `E` 到委托类型的转换的编译时应用程序 `D`。请注意，是否存在从 `E` 到 `D` 的隐式转换，并不保证转换的编译时应用程序将成功且不出错。

- 选择与窗体 `E(A)` 的方法调用（方法调用）相对应 `M` 单个方法，并进行以下修改：
  - 参数列表 `A` 是一个表达式列表，每个表达式都分类为一个变量，并且具有 `D` 的*formal_parameter_list*中相应参数的类型和修饰符（`ref` 或 `out`）。
  - 被视为的候选方法只是那些适用于其正常窗体（适用函数成员）的方法，而不是仅适用于其扩展窗体的方法。
- 如果方法调用的算法产生错误，则会发生编译时错误。否则，算法将生成单个最佳方法，`M` 与 `D` 具有相同数量的参数，并将转换视为存在。
- 所选方法 `M` 必须兼容（委托兼容性） `D` 委托类型，否则将发生编译时错误。
- 如果所选方法 `M` 是实例方法，则与 `E` 关联的实例表达式确定委托的目标对象。
- 如果所选方法 M 是通过实例表达式上的成员访问方式表示的扩展方法，则该实例表达式将确定委托的目标对象。
- 转换结果为类型 `D` 的值，即引用所选方法和目标对象的新创建的委托。
- 请注意，如果方法调用的算法未能找到实例方法，但在处理 `E(A)` 的调用作为扩展方法调用（扩展方法调用）时，此过程可能会导致创建扩展方法的委托。因此，创建的委托将捕获扩展方法以及其第一个参数。

下面的示例演示方法组转换：

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test
{
    static string F(object o) {...}

    static void G() {
        D1 d1 = F;            // Ok
        D2 d2 = F;            // Ok
        D3 d3 = F;            // Error -- not applicable
        D4 d4 = F;            // Error -- not applicable in normal form
        D5 d5 = F;            // Error -- applicable but not compatible

    }
}
```

赋值 `d1` 会将方法组 `F` 隐式转换为类型 `D1` 的值。

对 `d2` 的赋值演示了如何创建一个委托，该委托指向的派生程度较低（逆变）的参数类型和派生程度更高（协变）返回类型的方法。

如果该方法不适用，则分配给 `d3` 说明不存在转换。

对 `d4` 的赋值显示了方法必须以其正常形式适用的方式。

对 `d5` 的赋值显示了如何允许委托和方法的参数和返回类型仅与引用类型不同。

与所有其他隐式和显式转换一样，转换运算符可用于显式执行方法组转换。因此，示例

```
object obj = new EventHandler(myDialog.OkClick);
```

可以改为写入

```
object obj = (EventHandler)myDialog.OkClick;
```

方法组可能会影响重载决策，并参与类型推理。有关更多详细信息，请参阅函数成员。

方法组转换的运行时计算如下所示：

- 如果在编译时选择的方法是实例方法，或者它是作为实例方法访问的扩展方法，则将从与 `E` 关联的实例表达式确定委托的目标对象：
  - 计算实例表达式。如果此计算导致异常，则不执行其他步骤。
  - 如果实例表达式为*reference_type*，则由实例表达式计算的值将成为目标对象。如果所选的方法是实例方法，并且目标对象是 `null` 的，则会引发 `System.NullReferenceException`，而不执行进一步的步骤。
  - 如果实例表达式为*value_type*，则将执行装箱操作（装箱转换）以将值转换为对象，此对象将成为目标对象。
- 否则，所选方法为静态方法调用的一部分，并且 `null` 委托的目标对象。
- 分配 `D` 委托类型的新实例。如果内存不足，无法分配新的实例，则会引发 `System.OutOfMemoryException`，而

不执行进一步的步骤。

- 使用对在编译时确定的方法以及对上面计算的目标对象的引用来初始化新的委托实例。

表达式

2020/11/2 • • Edit Online

表达式是运算符和操作数构成的序列。本章定义语法、操作数和运算符的计算顺序，以及表达式的含义。

表达式分类

表达式分类为以下类别之一：

- 一个 值。每个值都有关联的类型。
- 一个变量。每个变量都有关联的类型，即变量的声明类型。
- 一个命名空间。具有此分类的表达式只能作为*member_access*（成员访问）的左侧出现。在任何其他上下文中，归类为命名空间的表达式会导致编译时错误。
- 一种类型。具有此分类的表达式只能作为*member_access*（成员访问）的左侧，或者作为 `as` 运算符（as 运算符）、`is` 运算符（is 运算符）或 `typeof` 运算符（typeof 运算符）的操作数出现。在其他任何上下文中，归类为类型的表达式会导致编译时错误。
- 一个方法组，它是从成员查找（成员查找）生成的一组重载方法。方法组可以有一个关联的实例表达式和一个关联的类型参数列表。调用实例方法时，实例表达式的计算结果将成为由 `this` （此访问）表示的实例。*Invocation_expression*（调用表达式）、*delegate_creation_expression*（委托创建表达式）和 is 运算符的左侧都允许使用方法组，并可将其隐式转换为兼容的委托类型（方法组转换）。在其他任何上下文中，归类为方法组的表达式会导致编译时错误。
- 空文本。具有此分类的表达式可隐式转换为引用类型或可以为 null 的类型。
- 匿名函数。具有此分类的表达式可隐式转换为兼容的委托类型或表达式目录树类型。
- 属性访问。每个属性访问都有关联的类型，即属性的类型。而且，属性访问可能具有关联的实例表达式。调用实例属性访问的访问器（`get` 或 `set` 块）时，实例表达式的计算结果将成为 `this` （此访问）表示的实例。
- 事件访问。每个事件访问都有关联的类型，即事件的类型。而且，事件访问可能具有关联的实例表达式。事件访问可能显示为 `+=` 和 `-=` 运算符（事件分配）的左操作数。在其他任何上下文中，归类为事件访问的表达式会导致编译时错误。
- 索引器访问。每个索引器访问都有关联的类型，即索引器的元素类型。此外，索引器访问具有关联的实例表达式和关联的参数列表。调用索引器访问的访问器（`get` 或 `set` 块）时，实例表达式的计算结果将成为 `this` （此访问）表示的实例，而参数列表的计算结果将成为调用的参数列表。
- 无变化。当表达式是 `void` 的返回类型的方法的调用时，会发生这种情况。归类为 nothing 的表达式仅在*statement_expression*（expression 语句）的上下文中有效。

表达式的最终结果绝不会是命名空间、类型、方法组或事件访问。相反，如上所述，这些类别的表达式是仅在某些上下文中允许的中间构造。

通过执行*get 访问器*或*set 访问器*的调用，始终可以将属性访问或索引器访问重新分类为值。特定访问器由属性或索引器访问的上下文确定：如果访问是赋值的目标，则调用*set 访问器*来分配新值（简单赋值）。否则，将调用*get 访问器*以获取当前值（表达式的值）。

表达式的值

大多数涉及表达式的构造都需要表达式来表示*值*。在这种情况下，如果实际表达式表示命名空间、类型、方法组或不执行任何操作，则会发生编译时错误。但是，如果表达式表示属性访问、索引器访问或变量，则将隐式替换属性、索引器或变量的值：

- 变量的值只是当前存储在变量标识的存储位置中的值。必须先将变量视为明确赋值（明确赋值），才能获得其值，否则将发生编译时错误。
- 属性访问表达式的值是通过调用属性的*get 访问器*获取的。如果该属性没有*get 访问器*，则会发生编译时错

误。否则，将执行函数成员调用(对动态重载决策进行编译时检查)，并且调用的结果将成为属性访问表达式的值。

- 索引器访问表达式的值是通过调用索引器的*get 访问器*获取的。如果索引器没有*get 访问器*，则会发生编译时错误。否则，将使用与索引器访问表达式关联的参数列表来执行函数成员调用(动态重载决策的编译时检查)，并且调用的结果将成为索引器访问表达式的值。

静态和动态绑定

根据构成表达式的类型或值(参数、操作数、接收方)确定操作含义的过程通常称为"***绑定***"。例如，方法调用的含义是根据接收方和参数的类型确定的。运算符的含义取决于其操作数的类型。

在C#编译时，通常基于其构成表达式的编译时类型确定操作的含义。同样，如果表达式包含错误，编译器将检测并报告错误。此方法称为***静态绑定***。

但是，如果表达式是动态表达式(即类型为 `dynamic`)，则表示它参与的任何绑定都应基于其运行时类型(即，它在运行时表示的对象的实际类型)，而不是它在编译时所具有的类型。这样，此类操作的绑定就会推迟到运行该程序的时间。这称为*动态绑定*。

当动态绑定操作时，编译器不会执行任何检查。相反，如果运行时绑定失败，则在运行时将错误报告为异常。

中C#的以下操作服从绑定：

- 成员访问：`e.M`
- 方法调用：`e.M(e1, ..., eN)`
- 委托调用：`e(e1, ..., eN)`
- 元素访问：`e[e1, ..., eN]`
- 对象创建：`new C(e1, ..., eN)`
- 重载的一元运算符：`+`、`-`、`!`、`~`、`++`、`--`、`true`、`false`
- 重载的二元运算符：`+`、`-`、`*`、`/`、`%`、`&`、`&&`、`|`、`||`、`??`、`^`、`<<`、`>>`、`==`、`!=`、`>`、`<`、`>=`、`<=`
- 赋值运算符：`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`<<=`、`>>=`
- 隐式和显式转换

如果不涉及动态表达式，C#则默认为静态绑定，这意味着在选择过程中使用构成表达式的编译时类型。但是，当上面列出的操作中的其中一个构成表达式为动态表达式时，将改为动态绑定该操作。

绑定时间

在编译时进行静态绑定，而动态绑定在运行时发生。在下面几节中，术语"***绑定时间***"指编译时或运行时，具体取决于绑定发生的时间。

下面的示例演示了静态和动态绑定的概念以及绑定时间：

```
object  o = 5;
dynamic d = 5;

Console.WriteLine(5);  // static  binding to Console.WriteLine(int)
Console.WriteLine(o);  // static  binding to Console.WriteLine(object)
Console.WriteLine(d);  // dynamic binding to Console.WriteLine(int)
```

前两个调用是静态绑定的：根据参数的编译时类型选取 `Console.WriteLine` 的重载。因此，绑定时间为编译时。

第三个调用是动态绑定的：根据参数的运行时类型选取 `Console.WriteLine` 的重载。出现这种情况的原因是，参数是动态表达式--它的编译时类型是 `dynamic`。因此，第三次调用的绑定时间是运行时。

动态绑定

动态绑定的目的是允许C#程序与*动态对象*交互，即不遵循C#类型系统的常规规则的对象。动态对象可以是具有不同类型系统的其他编程语言中的对象，也可以是以编程方式设置以实现不同操作的绑定语义的对象。

动态对象实现其自身语义的机制是定义的实现。定义的给定接口再次实现--由动态对象实现，以向C#运行时发出信号，指示它们具有特殊语义。因此，无论动态对象上的操作是动态绑定的，无论是在该文档中指定的C#，而不是在本文档中指定的语义，都需要接管。

尽管动态绑定的目的是允许与动态对象进行互操作，C#但允许动态绑定所有对象，无论它们是否为动态的。这允许动态对象的更平滑集成，因为它们的操作结果可能不是动态对象，而是在编译时对程序员来说是未知类型。此外，动态绑定还有助于消除容易出错的基于反射的代码，即使在没有任何对象是动态对象时也是如此。

以下各节描述了在应用动态绑定时与语言中的每个构造完全相同的内容、所应用的编译时检查（如果有），以及编译时结果和表达式分类的定义。

构成表达式的类型

静态绑定操作时，构成表达式的类型（例如，接收方、参数、索引或操作数）始终被认为是该表达式的编译时类型。

动态绑定操作时，将根据构成表达式的编译时类型以不同的方式确定构成表达式的类型：

- 编译时类型 `dynamic` 的构成表达式被视为在运行时使用表达式计算结果的实际值类型
- 在运行时将其编译时类型为类型参数的构成表达式视为具有类型参数绑定到的类型
- 否则，构成表达式被视为具有其编译时类型。

运算符

表达式是从***操作数***和***运算符***构造的。表达式的运算符指明了向操作数应用的运算。运算符的示例包括 `+`、`-`、`*`、`/` 和 `new`。操作数的示例包括文本、字段、局部变量和表达式。

有三种类型的运算符：

- 一元运算符。一元运算符采用一个操作数，并使用前缀表示法（如 `--x`）或后缀表示法（如 `x++`）。
- 二元运算符。二元运算符采用两个操作数，并使用中缀表示法（如 `x + y`）。
- 三元运算符。只存在一个三元运算符 `?:`;它采用三个操作数，并使用中缀表示法（`c ? x : y`）。

表达式中运算符的计算顺序由运算符的*优先级*和*关联*性（运算符优先级和结合性）决定。

表达式中的操作数从左到右进行计算。例如，在 `F(i) + G(i++) * H(i)` 中，使用旧值 `i` 调用方法 `F`，然后使用旧值 `i` 调用方法 `G`，最后，将使用新值 `H` 调用方法 `i`。这与运算符优先级不同。

特定的运算符可*重载*。运算符重载允许为操作指定用户定义的运算符实现，其中一个或两个操作数属于用户定义的类或结构类型（运算符重载）。

运算符优先级和关联性

如果表达式包含多个运算符，那么运算符的*优先级*决定了各个运算符的计算顺序。例如，表达式 `x + y * z` 的计算结果为 `x + (y * z)`，因为 `*` 运算符的优先级高于 binary `+` 运算符。运算符的优先级由其关联的语法生产的定义来确定。例如，*additive_expression*由 `+` 或 `-` 运算符分隔的一系列*multiplicative_expression*组成，从而使 `+` 和 `-` 运算符的优先级低于 `*`、`/` 和 `%` 运算符。

下表按优先级从高到低的顺序汇总了所有运算符：

SECTION	II	III
主要表达式	Primary	`x.y` `f(x)` `a[x]` `x++` `x--` `new` `typeof` `default` `checked` `unchecked` `delegate`

SECTION	𝙸𝙸	𝙸𝙸𝙸
一元运算符	一元	`+` `-` `!` `~` `++x` `--x` `(T)x`
算术运算符	乘法	`*` `/` `%`
算术运算符	加法	`+` `-`
移位运算符	移位	`<<` `>>`
关系和类型测试运算符	关系和类型测试	`<` `>` `<=` `>=` `is` `as`
关系和类型测试运算符	相等	`==` `!=`
逻辑运算符	逻辑"与"	`&`
逻辑运算符	逻辑 XOR	`^`
逻辑运算符	逻辑"或"	`|`
条件逻辑运算符	条件"与"	`&&`
条件逻辑运算符	条件"或"	`||`
Null 合并运算符	null 合并	`??`
条件运算符	条件	`?:`
赋值运算符，匿名函数表达式	赋值和 lambda 表达式	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=` `=>`

如果两个运算符在优先级相同的两个运算符之间出现，则运算符的关联性将控制运算的执行顺序：

- 除了赋值运算符和 null 合并运算符外，所有二元运算符都是***左结合***运算符，这意味着运算从左至右执行。例如，`x + y + z` 将计算为 `(x + y) + z`。
- 赋值运算符、null 合并运算符和条件运算符（`?:`）是***右结合***运算符，这意味着运算从右到左执行。例如，`x = y = z` 将计算为 `x = (y = z)`。

可以使用括号控制优先级和结合性。例如，`x + y * z` 先计算 `y` 乘 `z`，并将结果与 `x` 相加，而 `(x + y) * z` 则先计算 `x` 加 `y`，然后将结果与 `z` 相乘。

运算符重载

所有一元运算符和二元运算符都具有在任何表达式中自动可用的预定义实现。除了预定义的实现外，还可以通过在类和结构（运算符）中包括 `operator` 声明来引入用户定义的实现。用户定义的运算符实现始终优先于预定义运算符实现：只有当不存在适用的用户定义运算符实现时，才会考虑预定义运算符实现，如一元运算符重载决策和二元运算符重载决策中所述。

可***重载的一元运算符***是：

```
+   -   !   ~   ++   --   true   false
```

尽管 `true` 和 `false` 并不显式在表达式中使用（因此它们不会以运算符优先级和相关性的优先级表的顺序包括

在内)，但是它们被视为运算符，因为它们是在多个表达式上下文中调用的：布尔表达式(布尔表达式)和涉及条件运算符的表达式(条件逻辑运算符)。

可*重载的二元运算符*是：

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

只能重载上面列出的运算符。具体而言，无法重载成员访问、方法调用或 `=`、`&&`、`||`、`??`、`?:`、`=>`、`checked`、`unchecked`、`new`、`typeof`、`default`、`as`和 `is` 运算符。

重载二元运算符时，也会隐式重载相应的赋值运算符(若有)。例如，运算符 `*` 的重载也是运算符 `*=`的重载。这将在复合赋值中进一步说明。请注意，赋值运算符本身(`=`)无法重载。赋值始终将值的简单的逐位副本用于变量。

强制转换操作(如 `(T)x`)是通过提供用户定义的转换(用户定义的转换)而重载的。

元素访问(如 `a[x]`)不被视为可重载的运算符。而是通过索引器(索引器)支持用户定义的索引。

在表达式中，运算符是使用运算符表示法引用的，而在声明中，使用函数表示法引用运算符。下表显示了一元运算符和二元运算符的运算符和函数表示法之间的关系。在第一个条目中，*op*表示任何可重载的一元前缀运算符。在第二个条目中，*op*表示一元后缀 `++` 和 `--` 运算符。在第三个条目中，*op*表示任何可重载的二元运算符。

表达式	声明
`op x`	`operator op(x)`
`x op`	`operator op(x)`
`x op y`	`operator op(x,y)`

用户定义的运算符声明始终需要至少一个参数成为包含运算符声明的类或结构类型。因此，用户定义的运算符不能与预定义的运算符具有相同的签名。

用户定义的运算符声明不能修改运算符的语法、优先级或关联性。例如，`/` 运算符始终为二元运算符，始终具有在运算符优先级和关联性中指定的优先级别，并且始终为左结合。

尽管用户定义的运算符可以执行 pleases 的任何计算，但强烈不建议使用生成的实现，而不是所需的结果。例如，`operator ==` 的实现应比较两个操作数的相等性并返回相应的 `bool` 结果。

通过条件逻辑运算符对主要表达式中的单个运算符进行的说明指定运算符的预定义实现以及适用于每个运算符的任何其他规则。这些说明使用术语 "***一元运算符重载解析***"、"***二元运算符重载解析***" 和 "***数值升级***"，其中包括以下各节。

一元运算符重载决策

`op x` 或 `x op` 形式的操作(其中 `op` 是一个可重载的一元运算符)，并且 `x` 是 `X` 类型的表达式，则按以下方式处理：

- `X` 为操作 `operator op(x)` 提供的候选用户定义运算符集是使用候选用户定义运算符的规则确定的。
- 如果候选用户定义的运算符集不为空，则这将成为操作的候选运算符集。否则，预定义的一元 `operator op` 实现(包括其提升形式)会成为操作的候选运算符集。给定运算符的预定义实现在运算符的说明(主表达式和一元运算符)中指定。
- 重载决策的重载决策规则应用于候选运算符集，以选择与 `(x)` 参数列表相关的最佳运算符，而此运算符将成为重载决策过程的结果。如果重载决策未能选择单个最佳运算符，则会发生绑定时错误。

二元运算符重载决策

`x op y` 格式的操作，其中 `op` 是可重载的二元运算符，`x` 是 `X` 类型的表达式，而 `y` 是类型 `Y` 的表达式，则按如下方式进行处理：

- 确定操作 `operator op(x,y)` `X` 和 `Y` 提供的候选用户定义运算符集。该集由 `X` 提供的候选运算符与 `Y` 提供的候选运算符联合组成，每个运算符都是使用候选用户定义的运算符的规则确定的。如果 `X` 和 `Y` 属于同一类型，或者如果 `X` 和 `Y` 派生自一个公共的基类型，则仅在组合集内发生一次共享候选运算符。
- 如果候选用户定义的运算符集不为空，则这将成为操作的候选运算符集。否则，预定义的二进制 `operator op` 实现（包括其提升形式）会成为操作的候选运算符集。给定运算符的预定义实现是在运算符的说明（通过条件逻辑运算符进行算术运算符）中指定的。对于预定义的枚举和委托运算符，唯一认为的运算符是由作为一个操作数的绑定时类型的枚举或委托类型定义的运算符。
- 重载决策的重载决策规则应用于候选运算符集，以选择与 `(x,y)` 参数列表相关的最佳运算符，而此运算符将成为重载决策过程的结果。如果重载决策未能选择单个最佳运算符，则会发生绑定时错误。

候选用户定义的运算符

给定一个类型 `T` 和一个操作 `operator op(A)`，其中 `op` 是一个可重载的运算符，并且 `A` 是一个参数列表，则按如下方式确定 `T` 为 `operator op(A)` 提供的候选用户定义运算符集：

- 确定 `T0` 类型。如果 `T` 是可以为 null 的类型，则 `T0` 是其基础类型，否则 `T0` 等于 `T`。
- 对于 `T0` 中的所有 `operator op` 声明以及此类运算符的所有提升形式，如果至少有一个运算符适用于参数列表 `A`，则候选运算符集包含 `T0` 中的所有此类适用运算符。
- 否则，如果 `object``T0`，则候选运算符集为空。
- 否则，`T0` 提供的候选运算符集是 `T0` 的直接基类提供的候选运算符集，如果 `T0` 为类型参数，则为 `T0` 的有效基类。

数值提升

数值提升包括自动执行预定义的一元运算符和二元数值运算符的操作数的某些隐式转换。数值升级不是一种不同的机制，而是将重载决策应用于预定义运算符的效果。尽管可以实现用户定义的运算符来表现出类似的效果，但具体的数值升级却不会影响用户定义的运算符的计算。

作为数值升级的一个示例，请考虑二元 `*` 运算符的预定义实现：

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

当重载决策规则（重载决策）应用于这组运算符时，其效果是选择在操作数类型中存在隐式转换的第一个运算符。例如，对于操作 `b * s`，其中 `b` 为 `byte`，而 `s` 为 `short`，则重载决策将 `operator *(int,int)` 选为最佳运算符。因此，其效果是将 `b` 和 `s` 转换为 `int`，并 `int` 结果的类型。同样，对于操作 `i * d`（其中，`i` 是一个 `int` 并且 `d` 为 `double`），重载决策选择 `operator *(double,double)` 作为最佳运算符。

一元数值促销

一元数值升级发生于预定义 `+`、`-` 和 `~` 一元运算符的操作数。一元数值提升只包含将类型 `sbyte`、`byte`、`short`、`ushort` 或 `char` 的操作数转换为类型 `int`。此外，对于一元 `-` 运算符，一元数值提升将类型 `uint` 的操作数转换为类型 `long`。

二进制数值升级

二元数值升级发生在预定义 `+`、`-`、`*`、`/`、`%`、`&`、`|`、`^`、`==`、`!=`、`>`、`<`、`>=` 和 `<=` 二元运算符的操作数。二进制数值升级会隐式地将两个操作数转换为通用类型，在非关系运算符的情况下，也会成为运算的结果类型。二进制数值升级包括应用下列规则，顺序如下：

- 如果任一操作数为类型 `decimal`，则将另一个操作数转换为类型 `decimal`，如果另一个操作数的类型为

`float` 或 `double`，则会发生绑定时错误。

- 否则，如果其中一个操作数为 `double` 类型，则另一个操作数将转换为类型 `double`。
- 否则，如果其中一个操作数为 `float` 类型，则另一个操作数将转换为类型 `float`。
- 否则，如果其中一个操作数为 `ulong` 类型，则另一个操作数将转换为类型 `ulong`，如果另一个操作数的类型为 `sbyte`、`short`、`int` 或 `long`，则会发生绑定时错误。
- 否则，如果其中一个操作数为 `long` 类型，则另一个操作数将转换为类型 `long`。
- 否则，如果其中一个操作数的类型为 `uint`，另一个操作数的类型为 `sbyte`、`short` 或 `int`，则这两个操作数都将转换为类型 `long`。
- 否则，如果其中一个操作数为 `uint` 类型，则另一个操作数将转换为类型 `uint`。
- 否则，两个操作数都将转换为 `int` 类型。

请注意，第一个规则禁止将 `decimal` 类型与 `double` 和 `float` 类型一起使用的任何操作。规则如下所示，这是因为 `decimal` 类型与 `double` 和 `float` 类型之间没有隐式转换。

另请注意，如果另一个操作数为有符号整数类型，则操作数不能 `ulong` 类型。原因在于，不存在可以表示全部 `ulong` 范围和有符号整数类型的整数类型。

在上述两种情况下，强制转换表达式可用于将一个操作数显式转换为与另一个操作数兼容的类型。

示例中

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

发生绑定时错误，因为 `decimal` 不能与 `double` 相乘。通过将第二个操作数显式转换为 `decimal` 来解决此错误，如下所示：

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

提升的运算符

提升运算符允许对不可以为 null 的值类型进行运算的预定义运算符和用户定义的运算符也可用于这些类型的可以为 null 的形式。提升运算符是根据满足某些要求的预定义运算符和用户定义的运算符构造的，如下所述：

- 对于一元运算符

  ```
  +  ++  -  --  !  ~
  ```

  如果操作数和结果类型都是不可为 null 的值类型，则会存在运算符的提升形式。提升形式是通过向操作数和结果类型添加单个 `?` 修饰符来构造的。如果操作数为 null，则提升运算符将生成 null 值。否则，提升运算符将对操作数进行解包，应用基础运算符并包装结果。

- 对于二元运算符

  ```
  +  -  *  /  %  &  |  ^  <<  >>
  ```

  如果操作数和结果类型都是不可以为 null 的值类型，则该运算符的提升形式存在。提升形式是通过向每个操作数和结果类型添加单个 `?` 修饰符来构造的。如果有一个或两个操作数为空（`bool?` 类型的 `&` 和 `|` 运算符的异常，如布尔逻辑运算符中所述），则提升的运算符将生成 null 值。否则，提升运算符将对操作数进行解包，应用基础运算符并包装结果。

- 对于相等运算符

```
==   !=
```

  如果操作数类型既是不可为 null 的值类型，也不是 `bool` 的结果类型，则运算符的提升形式存在。提升形式是通过向每个操作数类型添加单个 `?` 修饰符来构造的。提升的运算符将两个 null 值视为相等，并将 null 值视为非 null 值。如果两个操作数都非空，则提升运算符将对操作数进行解包，并应用基础运算符来生成 `bool` 结果。

- 对于关系运算符

```
<   >   <=   >=
```

  如果操作数类型既是不可为 null 的值类型，也不是 `bool` 的结果类型，则运算符的提升形式存在。提升形式是通过向每个操作数类型添加单个 `?` 修饰符来构造的。如果一个或两个操作数为 null，则提升运算符将生成值 `false`。否则，提升运算符将对操作数进行解包，并应用基础运算符来产生 `bool` 结果。

成员查找

成员查找是指确定类型上下文中名称含义的进程。成员查找在计算表达式中的*simple_name*（简单名称）或*member_access*（成员访问）的过程中可能发生。如果*simple_name*或*member_access*作为*invocation_expression*（方法调用）的*primary_expression*出现，则称成员被调用。

如果成员是方法或事件，或者它是委托类型（委托）或类型 `dynamic`（动态类型）的常量、字段或属性，则称成员是*invocable*。

成员查找不仅考虑了成员的名称，还考虑了成员具有的类型参数的数目以及该成员是否可访问。出于成员查找的目的，泛型方法和嵌套泛型类型具有其各自声明中指示的类型参数的数目，并且所有其他成员都有零个类型参数。

类型 `T` 的 `K` 类型参数 `N` 名称的成员查找按如下方式进行处理：

- 首先，确定一组名为 `N` 的可访问成员：
  - 如果 `T` 是类型参数，则该集是在每个指定为 `object` `N` `T` 的主约束或辅助约束（类型参数约束）的每个类型中名为 `N` 的可访问成员集的并集。
  - 否则，该集由 `T` 中名为 `N` 的所有可访问（成员访问）成员组成，其中包括 `object` 中名为 `N` 的继承成员和可访问成员。如果 `T` 是构造类型，则通过替换类型参数来获取成员集，如构造类型成员中所述。包含 `override` 修饰符的成员将从集合中排除。
- 接下来，如果 `K` 为零，则将删除所有声明都包含类型参数的嵌套类型。如果 `K` 不为零，则删除所有具有不同数量的类型参数的成员。请注意，当 `K` 为零时，不会删除具有类型参数的方法，因为类型推理过程（类型推理）可能能够推断类型参数。
- 接下来，如果*调用*成员，则将从集合中删除所有非*invocable*成员。
- 接下来，将从集合中删除其他成员隐藏的成员。对于集中的每个成员 `S.M`（其中 `S` 是声明成员 `M` 的类型），将应用以下规则：
  - 如果 `M` 是常量、字段、属性、事件或枚举成员，则将从集合中删除在 `S` 的基类型中声明的所有成员。
  - 如果 `M` 是类型声明，则从该集中删除所有在 `S` 的基类型中声明的非类型，并且与在 `S` 的基类型中声明的 `M` 类型参数相同的所有类型声明都将从集合中删除。
  - 如果 `M` 是一种方法，则会从该集内删除在 `S` 的基类型中声明的所有非方法成员。
- 接下来，将从集合中删除类成员隐藏的接口成员。仅当 `T` 是类型参数并且 `T` 既具有 `object` 的有效基类，也具有非空的有效接口集（类型参数约束）时，此步骤才会生效。对于集合中的每个成员 `S.M`（其中 `S` 是声明成员 `M` 的类型），如果 `S` 为除 `object` 之外的类声明，则应用以下规则：

- 如果 `M` 是常量、字段、属性、事件、枚举成员或类型声明，则会从集中删除在接口声明中声明的所有成员。
- 如果 `M` 是方法，则将从集合中删除在接口声明中声明的所有非方法成员，并且将从该集内删除与在接口声明中声明的 `M` 具有相同签名的所有方法。

- 最后，删除隐藏成员后，将确定查找的结果：
  - 如果集由不是方法的单个成员组成，则此成员是查找的结果。
  - 否则，如果该集仅包含方法，则此方法组是查找的结果。
  - 否则，查找将不明确，并发生绑定时错误。

对于类型参数和接口以外的类型中的成员查找以及严格为单一继承的接口中的成员查找（继承链中的每个接口都具有完全零或一个直接基接口），查找规则的效果是只是派生成员将隐藏具有相同名称或签名的基成员。这种单一继承查找从来都不明确。接口成员访问中介绍了可能从多继承接口中的成员查找产生的歧义。

基类型

出于成员查找的目的，类型 `T` 被视为具有以下基本类型：

- 如果 `object``T`，则 `T` 没有基类型。
- 如果 `T` 是*enum_type*，则 `T` 的基类型是类类型 `System.Enum`、`System.ValueType` 和 `object`。
- 如果 `T` 是*struct_type*，则 `T` 的基类型是类类型 `System.ValueType` 和 `object`。
- 如果 `T` 是*class_type*，则 `T` 的基类型是 `T` 的基类，包括类类型 `object`。
- 如果 `T` 是*interface_type*，则 `T` 的基类型是 `T` 和类类型 `object` 的基接口。
- 如果 `T` 是*array_type*，则 `T` 的基类型是类类型 `System.Array` 和 `object`。
- 如果 `T` 是*delegate_type*，则 `T` 的基类型是类类型 `System.Delegate` 和 `object`。

函数成员

函数成员是包含可执行语句的成员。函数成员始终是类型的成员，不能是命名空间的成员。C#定义以下类别的函数成员：

- 方法
- 属性
- Events
- 索引器
- 用户定义的运算符
- 实例构造函数
- 静态构造函数
- 析构函数。

除析构函数和静态构造函数（无法显式调用）以外，函数成员中包含的语句是通过函数成员调用来执行的。用于编写函数成员调用的实际语法取决于特定的函数成员类别。

函数成员调用的参数列表（参数列表）为函数成员的参数提供实际值或变量引用。

调用泛型方法时，可以使用类型推理来确定要传递给方法的类型参数集。此过程在类型推理中进行了介绍。

调用方法、索引器、运算符和实例构造函数会使用重载决策来确定要调用的候选函数成员集中哪一组。重载决策中介绍了此过程。

在绑定时确定了特定函数成员（可能通过重载决策）后，调用函数成员的实际运行时过程将在编译时检查动态重载决策中进行说明。

下表总结了构造中发生的处理，涉及到可显式调用的六类函数成员。在表中，`e`、`x`、`y` 和 `value` 指示归类为变量或值的表达式，`T` 指示归类为类型的表达式，`F` 是方法的简单名称，`P` 是属性的简单名称。

方法调用	`F(x,y)`	应用重载决策，以选择包含类或结构中 `F` 的最佳方法。方法是通过参数列表 `(x,y)` 调用的。如果未 `static` 方法，则 `this` 实例表达式。
	`T.F(x,y)`	应用重载决策，以选择类或结构 `T` 中 `F` 的最佳方法。如果未 `static` 方法，则会发生绑定时错误。方法是通过参数列表 `(x,y)` 调用的。
	`e.F(x,y)`	重载决策适用于在类、结构或接口中选择由 `e` 类型给定的最佳方法 F。如果 `static` 方法，则会发生绑定时错误。方法是通过实例表达式 `e` 调用的，而参数列表 `(x,y)`。
和	`P`	调用包含类或结构中的属性 `P` 的 `get` 访问器。如果 `P` 是只写的，则会发生编译时错误。如果未 `static``P`，则 `this` 实例表达式。
	`P = value`	包含类或结构中的属性 `P` 的 `set` 访问器是通过参数列表 `(value)` 调用的。如果 `P` 是只读的，则会发生编译时错误。如果未 `static``P`，则 `this` 实例表达式。
	`T.P`	调用类或结构 `T` 中 `P` 属性的 `get` 访问器。如果 `P` 未 `static` 或 `P` 是只写的，则会发生编译时错误。
	`T.P = value`	类或结构 `T` 中 `P` 属性的 `set` 访问器是通过参数列表 `(value)` 调用的。如果 `P` 未 `static` 或 `P` 为只读，则会发生编译时错误。
	`e.P`	`e` 类型指定的类、结构或接口中的属性 `P` 的 `get` 访问器在实例表达式 `e` 中调用。如果 `P` `static` 或 `P` 是只写的，则会发生绑定时错误。
	`e.P = value`	`e` 类型给定的类、结构或接口中的属性 `P` 的 `set` 访问器是使用实例表达式 `e` 和参数列表 `(value)` 调用的。如果 `P` `static` 或 `P` 为只读，则会发生绑定时错误。
事件访问	`E += value`	调用包含类或结构中的事件 `E` 的 `add` 访问器。如果 `E` 不是静态的，则 `this` 实例表达式。

	`E -= value`	调用包含类或结构中的事件 `E` 的 `remove` 访问器。如果 `E` 不是静态的，则 `this` 实例表达式。
	`T.E += value`	调用类或结构 `T` 中 `E` 事件的 `add` 访问器。如果 `E` 不是静态的，则会发生绑定时错误。
	`T.E -= value`	调用类或结构 `T` 中 `E` 事件的 `remove` 访问器。如果 `E` 不是静态的，则会发生绑定时错误。
	`e.E += value`	`e` 类型给定的类、结构或接口中的事件 `E` 的 `add` 访问器在实例表达式 `e` 中调用。如果 `E` 是静态的，则会发生绑定时错误。
	`e.E -= value`	`e` 类型给定的类、结构或接口中的事件 `E` 的 `remove` 访问器在实例表达式 `e` 中调用。如果 `E` 是静态的，则会发生绑定时错误。
索引器访问	`e[x,y]`	重载决策适用于在由 e 类型给定的类、结构或接口中选择最佳索引器。索引器的 `get` 访问器在实例表达式 `e` 和参数列表 `(x,y)` 调用。如果索引器是只写的，则会发生绑定时错误。
	`e[x,y] = value`	重载决策适用于在类、结构或接口中选择由 `e` 类型给定的最佳索引器。索引器的 `set` 访问器在实例表达式 `e` 和参数列表 `(x,y,value)` 调用。如果索引器是只读的，则会发生绑定时错误。
运算符调用	`-x`	应用重载决策，以选择类或结构中由 `x` 类型给定的最佳一元运算符。用参数列表 `(x)` 调用所选运算符。
	`x + y`	重载决策适用于在由 `x` 类型或 `y` 类型给定的类或结构中选择最佳的二进制运算符。用参数列表 `(x,y)` 调用所选运算符。
实例构造函数调用	`new T(x,y)`	重载决策适用于在类或结构 `T` 中选择最佳实例构造函数。实例构造函数用参数列表 `(x,y)` 调用。

参数列表

每个函数成员和委托调用都包含一个参数列表，该列表为函数成员的参数提供实际值或变量引用。指定函数成员调用的参数列表的语法取决于函数成员类别：

- 对于实例构造函数、方法、索引器和委托，参数被指定为*argument_list*，如下所述。对于索引器，调用 `set` 访问器时，自变量列表还包括指定为赋值运算符的右操作数的表达式。

- 对于属性，调用 `get` 访问器时，参数列表是空的，并且包含在调用 `set` 访问器时指定为赋值运算符的右操作数的表达式。
- 对于事件，参数列表包含指定为 `+=` 或 `-=` 运算符的右操作数的表达式。
- 对于用户定义的运算符，参数列表包含一元运算符的单个操作数或二元运算符的两个操作数。

属性(属性)、事件(事件)和用户定义的运算符(运算符)的参数始终作为值参数传递(值参数)。索引器(索引器)的参数始终作为值参数(值参数)或参数数组(参数数组)传递。这些类别的函数成员不支持 Reference 和 output 参数。

实例构造函数、方法、索引器或委托调用的参数指定为*argument_list*:

```
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'ref' variable_reference
    | 'out' variable_reference
    ;
```

*Argument_list*由一个或多个由逗号分隔的*参数*组成。每个自变量都包含一个可选的*argument_name*后跟一个*argument_value*。带有*argument_name*的*参数*称为***命名参数***，而没有*argument_name*的*参数*是***位置参数***。在*argument_list*中的命名参数后，位置参数出现错误。

*Argument_value*可以采用以下形式之一：

- 一个*表达式*，指示参数作为值参数传递(值参数)。
- 关键字 `ref` 后跟*variable_reference*(变量引用)，指示参数作为引用参数(引用参数)传递。在将变量作为引用参数传递之前，必须对其进行明确赋值(明确赋值)。关键字 `out` 后跟*variable_reference*(变量引用)，指示参数作为输出参数传递(输出参数)。在将变量作为 output 参数传递的函数成员调用之后，变量被视为明确赋值(明确赋值)。

对应的参数

对于参数列表中的每个参数，必须在要调用的函数成员或委托中包含相应的参数。

以下内容中使用的参数列表的确定方式如下：

- 对于在类中定义的虚拟方法和索引器，将从函数成员的最特定声明或重写中选取参数列表，从接收方的静态类型开始，然后搜索其基类。
- 对于接口方法和索引器，从接口类型和搜索基接口开始，选取参数列表作为成员的最明确定义。如果找不到唯一的参数列表，则将构造一个具有不可访问的名称且不带可选参数的参数列表，以便调用不能使用命名参数或省略可选参数。
- 对于分部方法，使用定义分部方法声明的参数列表。
- 对于所有其他函数成员和委托，只有一个参数列表，该列表是使用的。

参数或参数的位置定义为参数列表或参数列表中其前面的参数或参数的数目。

按如下所示建立函数成员参数的对应参数：

- 实例构造函数、方法、索引器和委托的*argument_list*中的参数：

- 一个位置参数，其中固定参数出现在参数列表中的同一位置，该参数对应于该参数。
- 函数成员的位置自变量，其正常形式中调用的参数数组对应于参数数组，该参数数组必须出现在参数列表中的同一位置。
- 函数成员的位置自变量，其中的参数数组以其展开形式调用，其中没有固定参数出现在参数列表中的同一位置，它对应于参数数组中的一个元素。
- 命名参数对应于参数列表中具有相同名称的参数。
- 对于索引器，调用 `set` 访问器时，指定为赋值运算符的右操作数的表达式对应于 `set` 访问器声明的隐式 `value` 参数。

- 对于属性，当调用 `get` 访问器时，没有参数。调用 `set` 访问器时，指定为赋值运算符的右操作数的表达式对应于 `set` 访问器声明的隐式 `value` 参数。
- 对于用户定义的一元运算符(包括转换)，单个操作数对应于运算符声明的单个参数。
- 对于用户定义的二进制运算符，左操作数对应于第一个参数，右侧操作数对应于运算符声明的第二个参数。

参数列表的运行时计算

在运行时处理函数成员调用期间(对动态重载决策进行编译时检查)，将按从左至右的顺序计算参数列表的表达式或变量引用，如下所示：

- 对于值参数，将计算参数表达式，并执行到相应参数类型的隐式转换(隐式转换)。生成的值将成为函数成员调用中 value 参数的初始值。
- 对于引用参数或输出参数，将对变量引用进行计算，并将生成的存储位置成为函数成员调用中的参数所表示的存储位置。如果作为引用或输出参数提供的变量引用是*reference_type*的数组元素，则执行运行时检查以确保数组的元素类型与参数的类型相同。如果此检查失败，则会引发 `System.ArrayTypeMismatchException` 。

方法、索引器和实例构造函数可以将其最右侧的参数声明为参数数组(参数数组)。此类函数成员的调用方式可以是普通形式，也可以是其展开形式，具体取决于适用的(适用的函数成员)：

- 当使用参数数组的函数成员以其正常形式调用时，为参数数组提供的参数必须是可隐式转换为参数数组类型的单个表达式。在这种情况下，参数数组的作用与值参数完全相同。
- 当具有参数数组的函数成员以其展开形式调用时，该调用必须为参数数组指定零个或多个位置参数，其中每个参数都是可隐式转换为参数数组元素类型的表达式。在这种情况下，调用会创建一个参数数组类型的实例，该实例的长度与参数的数量相对应，使用给定的参数值初始化数组实例的元素，并使用新创建的数组实例作为实际实际.

自变量列表的表达式始终按写入它们的顺序进行计算。因此，示例

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

生成输出

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

如果存在从 `B` 到 `A` 的隐式引用转换，数组协方差规则(数组协方差)允许 `A[]` 为数组类型 `B[]` 的实例的引用。由于这些规则，当*reference_type*的数组元素作为引用或输出参数传递时，需要运行时检查以确保数组的实际元

素类型与参数的类型相同。示例中

```
class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);        // Ok
        F(ref b[1]);        // ArrayTypeMismatchException
    }
}
```

`F` 的第二次调用导致引发 `System.ArrayTypeMismatchException`，因为 `b` 的实际元素类型为 `string` 而不是 `object`。

当具有参数数组的函数成员以其展开形式调用时，该调用的处理方式与使用数组初始值设定项（数组创建表达式）的数组创建表达式在展开的参数周围插入的方式完全相同。例如，给定声明

```
void F(int x, int y, params object[] args);
```

对方法的展开形式的以下调用

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

完全对应于

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

特别要注意的是，如果参数数组提供的参数为零，则将创建一个空数组。

当使用相应的可选参数从函数成员中省略参数时，将隐式传递函数成员声明的默认参数。因为这些始终是常量，所以其计算不会影响剩余参数的计算顺序。

类型推理

如果在未指定类型实参的情况下调用泛型方法，则*类型推理*进程将尝试推断调用的类型参数。类型推理的存在允许使用更方便的语法来调用泛型方法，并允许程序员避免指定冗余类型信息。例如，给定方法声明：

```
class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}
```

无需显式指定类型参数即可调用 `Choose` 方法：

```
int i = Chooser.Choose(5, 213);                // Calls Choose<int>

string s = Chooser.Choose("foo", "bar");       // Calls Choose<string>
```

通过类型推理，`int` 和 `string` 的类型参数由方法的自变量确定。

类型推理作为方法调用(方法调用)的绑定时处理的一部分发生，并在调用的重载解析步骤之前发生。如果在方法调用中指定了特定方法组，但没有将任何类型参数指定为方法调用的一部分，则会将类型推理应用于方法组中的每个泛型方法。如果类型推理成功，则使用推断出的类型参数来确定参数的类型，以供后续重载决策使用。如果重载决策选择泛型方法作为要调用的方法，则会将推断的类型参数用作调用的实际类型参数。如果特定方法的类型推理失败，则该方法将不参与重载决策。类型推理的失败本身不会导致绑定时错误。但是，当重载决策失败并找不到任何适用的方法时，通常会导致绑定时错误。

如果所提供的参数数目不同于方法中的参数数量，则推理会立即失败。否则，假定泛型方法具有以下签名：

```
Tr M<X1,...,Xn>(T1 x1, ..., Tm xm)
```

使用窗体的方法调用 `M(E1...Em)` 类型推理的任务是查找 `S1...Sn` 的每个类型参数的唯一类型参数 `X1...Xn` 以便调用 `M<S1...Sn>(E1...Em)` 变为有效。

在推断过程中，每个类型参数 `Xi` 都*固定*到特定类型 `Si` 或未*使用关联*的*界限*集进行固定。每个边界都是 `T` 的一种类型。最初，每个类型变量 `Xi` 都不包含一组空的界限。

类型推理分阶段发生。每个阶段都将尝试根据上一阶段的发现来推导更多类型变量的类型参数。第一阶段会对边界进行一些初始推断，而第二阶段则将类型变量修复为特定类型，并推导出进一步的边界。第二阶段可能需要重复一次。

注意：仅当调用泛型方法时，才会发生类型推理。用于转换方法组的类型推理中介绍了用于转换方法组的类型推理和查找一组表达式的最佳通用类型中介绍的方法。

第一阶段

对于每个方法参数 `Ei`：

- 如果 `Ei` 是匿名函数，则*显式参数类型推理*(显式参数类型推断)是从 `Ei` 到 `Ti`
- 否则，如果 `Ei` 具有类型 `U` 并且 `xi` 是值参数，则将*从* `U` *到* `Ti` 进行*下限推理*。
- 否则，如果 `Ei` 具有类型 `U` 并且 `xi` 是 `ref` 或 `out` 参数，则将*从* `U` *到* `Ti` 进行*精确推断*。
- 否则，不会对此参数进行推理。

第二阶段

第二个阶段按如下方式进行：

- 不*依赖于*(依赖)任何 `Xj` *的所有未固定的*类型 `Xi` 变量都是固定的(正在修复)。
- 如果不存在这样的类型变量，则所有*未固定的类型变量* `Xi` 都是*固定*的，其中所有的保留都是：
  - 至少有一个类型变量 `Xj` 依赖于 `Xi`
  - `Xi` 具有非空的界限集
- 如果不存在这样的类型变量，并且仍有未*固定*的类型变量，则类型推理将失败。
- 否则，如果不存在进一步的未*固定*类型变量，则类型推理将成功。
- 否则，对于具有相应参数类型的所有参数 `Ei` `Ti` 其中*输出类型*(输出类型)包含未*固定*的类型变量 `Xj` 但*输入类型*(输入类型)不存在，则将*从* `Ei` Output type inferences*到* `Ti` 进行*输出类型推理*。然后，重复第二阶段。

输入类型

如果 `E` 是方法组或隐式类型的匿名函数，并且 `T` 为委托类型或表达式树类型，则 `T` 的所有参数类型均为*类型*为 `T` 的 `E` 的*输入类型*。

输出类型

如果 `E` 为方法组或匿名函数，并且 `T` 为委托类型或表达式树类型，则 `T` 的返回类型为*类型*为 `T` 的 `E` 的*输出类型*。

依赖关系

未*固定的类型*变量 `Xi` *直接依赖于*未固定的类型变量 `Xj` 如果对于具有类型 `Xj` 的某些参数 `Tk` `Ek`，`Ek` *在类型*为 `Tk` 的输出类型为 `Xi` 的*输出类型*中发生。

如果 `Xj` *直接依赖于* `Xi` 或者 `Xi` *直接*依赖于 `Xk` 并且 `Xk` *依赖于* `Xj`，则 `Xj` *依赖于* `Xi`。因此 "依赖于" 是传递的，而不是 "直接依赖于" 的反身闭包。

输出类型推断

*输出类型推断*是通过以下方式*从* `E` *到*类型 `T` 的表达式进行的：

- 如果 `E` 是 `U`（推断返回类型）且 `T` 为委托类型或具有返回类型 `Tb` 的表达式树类型的匿名函数，则将*从* `U` Lower-bound inferences *到* `Tb` 进行 *下限推理*。
- 否则，如果 `E` 为方法组，`T` 是参数类型为 `T1...Tk` 和返回类型 `Tb` 的委托类型或表达式树类型，并且 `E` 生成一个返回类型 `T1...Tk` 的单个方法，则*从* `U` *到* `U` *生成一个方法*。
- 否则，如果 `E` 是 `U` 类型的表达式，则将*从* `U` *到* `T` 进行 *下限推理*。
- 否则，不进行推断。

显式参数类型推断

*显式参数类型推理*是通过以下方式*从* `E` *到*类型 `T` 的表达式进行的：

- 如果 `E` *为*的参数类型 `U1...Uk` 为-1 的显式类型匿名函数，`T` 是具有参数类型 `V1...Vk` 的委托类型或表达式树类型，则对每个 `Ui` 进行精确推理（精确推断）从到对应的 0。`Ui``Vi`

精确推断

*从*类型 `U` *到*类型 `V` 的*精确推理*如下所示：

- 如果 `V` 是未*固定*的 `Xi` 之一，则 `U` 添加到 `Xi` 的完全限定集。
- 否则，将通过检查是否存在以下任何情况来确定 `V1...Vk` 和 `U1...Uk`：
  - `V` 是 `V1[...]` 数组类型，`U` 是同一排名 `U1[...]` 的数组类型
  - `V` 是 `V1?` 类型，`U` 为类型 `U1?`
  - `V` 是构造类型 `C<V1...Vk>` 并且 `U` 为构造类型 `C<U1...Uk>`

  如果上述任何情况都适用，则将*从*每个 `Ui` *到*相应的 `Vi` 进行*确切的推理*。
- 否则，不进行推断。

下限推理

*从*类型 `U` *到*类型 `V` 的 *下限推理*如下所示：

- 如果 `V` 是未*固定*的 `Xi` 之一，则 `U` 添加到 `Xi` 的下限集。
- 否则，如果 `V` 是类型 `V1?` 并且 `U` 为类型 `U1?` 则从 `U1` 到 `V1` 进行下限推理。
- 否则，将通过检查是否存在以下任何情况来确定 `U1...Uk` 和 `V1...Vk`：
  - `V` 是 `V1[...]` 数组类型，`U` 是同一排名 `U1[...]`（或其有效基类型为 `U1[...]` 的类型参数）的数组类型
  - `V` 是 `IEnumerable<V1>`、`ICollection<V1>` 或 `IList<V1>` 中的一种，`U` 是一维数组类型 `U1[]`（或其有效基类型为 `U1[]` 的类型参数）
  - `V` 是 `C<V1...Vk>` 构造的类、结构、接口或委托类型，并且有一个唯一类型 `C<U1...Uk>` 以便 `U`（或者，如果 `U` 是一个类型参数，其有效的基类或其有效接口集的任何成员）等同于、从（直接或间

接）`C<U1...Uk>` 继承。

（"唯一性" 限制意味着在 case 接口 `C<T> {} class U: C<X>, C<Y> {}` 中，从 `U` 推断到 `C<T>` 时不进行推理，因为 `U1` 可以 `X` 或 `Y`。）

如果这些情况中的任何一种都适用，则将*从*每个 `Ui` *推导为相应*的 `Vi`，如下所示：

- 如果 `Ui` 已知为引用类型，则进行*精确推理*
- 否则，如果 `U` 是数组类型，则进行*下限推理*
- 否则，如果 `C<V1...Vk>` `V`，则推理依赖于 `C` 的第 i 个类型参数：
  - 如果它是协变的，则进行*下限推理*。
  - 如果它是逆变的，则进行*上限推断*。
  - 如果它是固定的，则进行*精确推理*。

- 否则，不进行推断。

上限推理

*从*类型 `U` *到*类型 `V` 的*上限推理*如下所示：

- 如果 `V` 是未*固定*的 `Xi` 之一，则 `U` 添加到 `Xi` 的上限。
- 否则，将通过检查是否存在以下任何情况来确定 `V1...Vk` 和 `U1...Uk`：
  - `U` 是 `U1[...]` 数组类型，`V` 是同一排名 `V1[...]` 的数组类型
  - `U` 是 `IEnumerable<Ue>`、`ICollection<Ue>` 或 `IList<Ue>`，`V` 是一维数组类型 `Ve[]`
  - `U` 是 `U1?` 类型，`V` 为类型 `V1?`
  - `U` 是构造的类、结构、接口或委托类型 `C<U1...Uk>` 并且 `V` 是一个与相同的类、结构、接口或委托类型，继承自（直接或间接），或实现（直接或间接）唯一类型 `C<V1...Vk>`

    （"唯一性" 限制意味着如果有 `interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}`，则从 `C<U1>` 推断到 `V<Q>` 时不进行推理。不从 `U1` 到 `X<Q>` 或 `Y<Q>` 进行推断。）

  如果这些情况中的任何一种都适用，则将*从*每个 `Ui` *推导为相应*的 `Vi`，如下所示：

  - 如果 `Ui` 已知为引用类型，则进行*精确推理*
  - 否则，如果 `V` 是数组类型，则进行*上限推理*
  - 否则，如果 `C<U1...Uk>` `U`，则推理依赖于 `C` 的第 i 个类型参数：
    - 如果它是协变的，则进行*上限推理*。
    - 如果它是逆变的，则进行*下限推理*。
    - 如果它是固定的，则进行*精确推理*。
- 否则，不进行推断。

更正

使用一组界限 `Xi` 未*固定*的类型变量是*固定*的，如下所示：

- *候选类型*集 `Uj` 以 `Xi` 的边界集中的所有类型的集合开头。
- 然后，我们依次检查每个绑定 for `Xi`：对于每个完全绑定 `U`，`Xi` `Uj` 的所有类型，这些类型与从候选集删除 `U` 不完全相同。对于 `Xi` 的每个下限 `U`，从候选集中将*不*会从 `U` 中删除任何 `Uj` 类型的隐式转换。对于 `Xi` 的每个上限 *`U`*，*`Uj`* 将从候选集删除没有隐式转换到 `U` 的所有类型。
- 如果其余的候选类型 `Uj` 有一个唯一的类型 `V` 从该类型中有一个到所有其他候选类型的隐式转换，则 `Xi` 固定到 `V`。
- 否则，类型推理将失败。

推断出的返回类型

推断出的匿名函数的返回类型 `F` 在类型推理和重载决策过程中使用。只能为所有参数类型已知的匿名函数确定推理出的返回类型，因为这些参数类型是显式给定的，它是通过匿名函数转换提供的，或者是在对封闭泛型的类型推理期间推断的方法调用。

按如下方式确定***推断结果类型***：

- 如果 `F` 的主体是具有类型的***表达式***，则推断出的结果类型 `F` 是该表达式的类型。
- 如果 `F` 的主体是一个*块*，块的 `return` 语句中的表达式集有一个最常见的类型 `T`（查找一组表达式的最佳通用类型），则将 `T` 推理出的结果 `F` 类型。
- 否则，无法推断 `F` 的结果类型。

按如下方式确定***推断出的返回类型***：

- 如果 `F` 是异步的，并且 `F` 的主体是分类为 nothing（表达式分类）的表达式，或者是没有 return 语句的语句块包含表达式，则推断出的返回类型为 `System.Threading.Tasks.Task`
- 如果 `F` 为 async 并且 `T` 推理结果类型，则推断出的返回类型为 `System.Threading.Tasks.Task<T>`。
- 如果 `F` 不是异步的，并且 `T` 推断结果类型，则推断出的返回类型为 `T`。
- 否则，将无法为 `F` 推断返回类型。

作为涉及匿名函数的类型推理的示例，请考虑在 `System.Linq.Enumerable` 类中声明 `Select` 扩展方法：

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource,TResult>(
            this IEnumerable<TSource> source,
            Func<TSource,TResult> selector)
        {
            foreach (TSource element in source) yield return selector(element);
        }
    }
}
```

假定 `System.Linq` 命名空间是使用 `using` 子句导入的，并且给定的类 `Customer` 具有类型 `string` 的 `Name` 属性，则可以使用 `Select` 方法选择客户列表的名称：

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

通过将调用重写为静态方法调用来处理 `Select` 的扩展方法调用（扩展方法调用）：

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

由于未显式指定类型参数，因此将使用类型推理来推断类型参数。首先，`customers` 自变量与 `source` 参数相关，推理 `T` 为 `Customer`。然后，使用上面所述的匿名函数类型推理过程，`c` 给定类型 `Customer`，并且表达式 `c.Name` 与 `selector` 参数的返回类型相关，并推断 `S` 为 `string`。因此，调用等效于

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

结果的类型为 `IEnumerable<string>`。

下面的示例演示匿名函数类型推理如何允许类型信息在泛型方法调用中的参数之间 "流"。给定方法：

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
    return f2(f1(value));
}
```

调用的类型推理：

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

按如下所示进行操作：首先，参数 `"1:15:30"` 与 `value` 参数相关，推理 `X` 为 `string`。然后，将 `s` 第一个匿名函数的参数 `string`，并将表达式 `TimeSpan.Parse(s)` 与 `f1` 的返回类型相关联，推断 `Y` 为 `System.TimeSpan`。最后，为第二个匿名函数 `t` 的参数赋予 `System.TimeSpan` 推断出的类型，并且表达式 `t.TotalSeconds` 与 `f2` 的返回类型相关，推断 `Z` 为 `double`。因此，调用的结果为 `double` 类型。

用于转换方法组的类型推理

与泛型方法的调用类似，当包含泛型方法的方法组 `M` 转换为给定委托类型 `D`（方法组转换）时，也必须应用类型推理。给定方法

```
Tr M<X1...Xn>(T1 x1 ... Tm xm)
```

并且方法组 `M` 分配给委托类型 `D` 推断类型的任务是查找类型参数 `S1...Sn` 以便表达式：

```
M<S1...Sn>
```

与 `D` 成为兼容的（委托声明）。

与泛型方法调用的类型推理算法不同，在本例中，只有参数*类型*，没有参数*表达式*。具体而言，不存在任何匿名函数，因此不需要进行多个推理阶段。

相反，所有 `Xi` 都被视为未*固定*的，并且从 `D` 的每个参数类型*到* `M` 的相应参数 `Tj` 类型，将*从*每个参数类型 `Uj` 进行*下限推理*。如果没有找到任何 `Xi` 的边界，则类型推理将失败。否则，所有 `Xi` 都*固定*到相应的 `Si`，这是类型推理的结果。

查找一组表达式的最佳通用类型

在某些情况下，需要为一组表达式推断通用类型。特别是，隐式类型化数组的元素类型和带*块*体的匿名函数的返回类型是通过这种方式找到的。

直观地说，给定一组表达式 `E1...Em` 此推理应等效于调用方法

```
Tr M<X>(X x1 ... X xm)
```

作为参数的 `Ei`。

更准确地说，推理最初使用未*固定*的类型变量 `X`。然后*从*每个 `Ei` *到* `X` 进行*输出类型推断*。最后，`X` 是*固定*的，如果成功，则结果类型 `S` 是生成的表达式的最佳通用类型。如果不存在这样的 `S`，则表达式没有最常见的类型。

重载决策

重载决策是用于选择在给定参数列表和一组候选函数成员时调用的最佳函数成员的绑定时机制。重载决策选择要在中C#的下列不同上下文中调用的函数成员：

- 调用*invocation_expression*中名为的方法（方法调用）。
- 调用*object_creation_expression*中名为的实例构造函数（对象创建表达式）。
- 通过*element_access*（元素访问）调用索引器访问器。

- 表达式中引用的预定义运算符或用户定义的运算符的调用(一元运算符重载决策和二元运算符重载决策)。

其中每个上下文都定义候选函数成员的集合和参数列表，其独特方式为，如以上所列部分中所述。例如，方法调用的候选项不包括标记为 `override`（成员查找）的方法，并且派生类中的任何方法都适用时，基类中的方法不是候选项(方法调用)。

确定候选函数成员和参数列表后，在所有情况下，最佳函数成员的选择是相同的：

- 给定一组适用的候选函数成员后，该集合中的最佳函数成员就是。如果集只包含一个函数成员，则该函数成员是最佳函数成员。否则，最佳函数成员就是比给定自变量列表更好的所有其他函数成员的一个函数成员，前提是将每个函数成员与在更好的函数成员中使用规则的所有其他函数成员进行比较。如果不是正好有一个比所有其他函数成员更好的函数成员，则函数成员调用是不明确的，并发生绑定时错误。

以下部分定义了术语***适用的函数成员***和***更好的函数成员***的精确含义。

适用的函数成员

当满足以下所有条件时，函数成员被称为适用于自变量列表的***函数成员*** `A`：

- `A` 中的每个参数都对应于函数成员声明中的一个参数(如相应的参数中所述)，并且没有参数对应的任何参数都是可选参数。
- 对于 `A` 中的每个自变量，参数的参数传递模式(即值、`ref` 或 `out`)与相应参数的参数传递模式相同，并且
  - 对于值参数或参数数组，存在从参数到相应参数类型的隐式转换(隐式转换)或
  - 对于 `ref` 或 `out` 参数，参数的类型与对应参数的类型相同。毕竟，`ref` 或 `out` 参数是传递的参数的别名。

对于包含参数数组的函数成员，如果函数成员适用于上述规则，则称它适用于其***正常形式***。如果包含参数数组的函数成员在其正常形式下不适用，则函数成员可以采用其***展开形式***：

- 通过使用参数数组元素类型的零个或多个值参数替换函数成员声明中的参数数组，使参数列表中的参数数目与参数的总数 `A` 匹配，来构造展开后的形式。如果 `A` 的参数少于函数成员声明中的固定参数数目，则无法构造函数成员的展开形式，因此不适用。
- 否则，如果为中的每个参数 `A` 参数传递模式，则展开形式适用，参数的参数传递模式与相应参数的参数传递模式相同，并且
  - 对于固定值参数或扩展创建的值参数，存在从自变量的类型到相应参数类型的隐式转换(隐式转换)或
  - 对于 `ref` 或 `out` 参数，参数的类型与对应参数的类型相同。

更好的函数成员

出于确定更好的函数成员的目的，一个被去除的自变量列表 A 构造，它以其在原始参数列表中出现的顺序仅包含自变量表达式本身。

按以下方式构造每个候选函数成员的参数列表：

- 如果函数成员仅适用于扩展窗体，则使用展开的窗体。
- 将从参数列表中删除不具有相应参数的可选参数
- 参数会进行重新排序，以便它们在与参数列表中的相应参数相同的位置上发生。

给定一个参数列表 `A`，其中包含一组参数表达式 `{E1, E2, ..., En}` 和两个适用的函数成员 `Mp`，并 `Mq` 参数类型 `{P1, P2, ..., Pn}` 和 `{Q1, Q2, ..., Qn}`，则 `Mp` 定义为比 `Mq` ***更好的函数成员***

- 对于每个参数，从 `Ex` 到 `Qx` 的隐式转换并不优于从 `Ex` 到 `Px` 的隐式转换和
- 对于至少一个参数，从 `Ex` 到 `Px` 的转换优于从 `Ex` 到 `Qx` 的转换。

当执行此计算时，如果 `Mp` 或 `Mq` 适用于其展开形式，则 `Px` 或 `Qx` 引用参数列表的展开形式的参数。

如果参数类型序列 `{P1, P2, ..., Pn}` 和 `{Q1, Q2, ..., Qn}` 等效(即每个 `Pi` 的标识转换为相应的 `Qi`)，则将按顺序应用以下附加中断规则，以确定更好的函数成员。

- 如果 `Mp` 为非泛型方法并且 `Mq` 为泛型方法，则 `Mp` 比 `Mq` 更好。
- 否则，如果 `Mp` 适用于其正常窗体并且 `Mq` 具有 `params` 数组，并且仅适用于其展开形式，则 `Mp` 比 `Mq` 更好。
- 否则，如果 `Mp` 具有比 `Mq` 更多的声明参数，则 `Mp` 比 `Mq` 更好。如果两种方法都有 `params` 数组，并且仅适用于其展开的形式，则会发生这种情况。
- 否则，如果 `Mp` 的所有参数都有相应的参数，而默认参数需要替换为 `Mq` 中的至少一个可选参数，则 `Mp` 比 `Mq` 更好。
- 否则，如果 `Mp` 具有比 `Mq` 更具体的参数类型，则 `Mp` 比 `Mq` 更好。让 `{R1, R2, ..., Rn}` 和 `{S1, S2, ..., Sn}` 表示 `Mp` 和 `Mq` 的未实例化和未扩展的参数类型。`Mp` 的参数类型比 `Mq` 的更具体，因为对于每个参数，`Rx` 不太具体于 `Sx`，并且对于至少一个参数，`Rx` 比 `Sx` 更为具体：
  - 类型参数不如非类型参数的特定类型。
  - 如果至少有一个类型自变量是更具体的类型，并且没有类型参数的特定于其他中的相应类型参数，则递归方式比另一个构造类型（具有相同数量的类型参数）更为具体。
  - 如果第一个数组的元素类型比第二个元素的类型更具体，则数组类型比另一个数组类型更具体（具有相同维数的数组）。
- 否则，如果一个成员是非提升运算符，而另一个成员是提升运算符，则不提升的运算符更好。
- 否则，两个函数成员都不会更好。

表达式的更好转换

给定一个隐式转换 `C1`，该转换将 `E` 类型的表达式转换为类型 `T1`，并从表达式 `E` 转换为类型 `T2` 的隐式转换 `C2`，`C1` 是比 `C2` ***更好的转换***（如果 `E` 并不完全匹配 `T2` 和至少一个保留项）：

- `E` 完全匹配 `T1`（完全匹配的表达式）
- `T1` 是比 `T2` 更好的转换目标（更好的转换目标）

完全匹配的表达式

给定表达式 `E` 和类型 `T`，如果下列其中一项保留，`E` 与 `T` 完全匹配：

- `E` 具有类型 `S`，并且存在从 `S` 到 `T` 的标识转换
- `E` 是一种匿名函数，`T` 为委托类型 `D` 或表达式目录树类型 `Expression<D>` 和以下保留之一：
  - `E` 在 `D` 参数列表的上下文中存在的推断返回类型 `X`（推断返回类型），并且存在从 `X` 到返回类型的标识转换 `D`
  - `E` 为非异步的，并且 `D` 具有返回类型 `Y` 或者 `E` 为 async 并且 `D` 具有返回类型 `Task<Y>`，以及以下保留项之一：
    - `E` 的主体是一个完全匹配的表达式 `Y`
    - `E` 的主体是一个语句块，其中每个 return 语句返回一个完全匹配的表达式 `Y`

更好的转换目标

如果 `T1` 和 `T2` 两种不同的类型，`T1` 是比 `T2` 更好的转换目标，前提是不存在从 `T2` 到 `T1` 的隐式转换，并且至少包含以下其中一项：

- 从 `T1` 到 `T2` 存在的隐式转换
- `T1` 为委托类型 `D1` 或表达式树类型 `Expression<D1>`，`T2` 为委托类型 `D2` 或表达式树类型 `Expression<D2>`，`D1` 具有返回类型 `S1` 和以下保留之一：
  - `D2` 返回 void
  - `D2` 具有 `S2` 的返回类型，`S1` 是比 `S2` 更好的转换目标
- `T1` `Task<S1>`，`T2` `Task<S2>`，`S1` 是比 `S2` 更好的转换目标
- `T1` `S1` 或 `S1?`，其中 `S1` 是有符号整数类型，`T2` `S2` 或 `S2?`，其中 `S2` 是无符号整数类型。具体而言：
  - `S1` 是 `sbyte` 的，`S2` 为 `byte`、`ushort`、`uint` 或 `ulong`
  - `S1` 是 `short` 并且 `S2` 为 `ushort`、`uint` 或 `ulong`
  - `S1` 是 `int` 并且 `S2` 为 `uint` 或 `ulong`

- `S1` 是 `long` 并且 `S2` 为 `ulong`

泛型类中的重载

尽管声明的签名必须是唯一的，但类型参数的替换可能导致相同的签名。在这种情况下，以上重载解决方法的分解规则将选取最特定的成员。

下面的示例显示了根据此规则有效和无效的重载：

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);                  // Overload resolution for G<int>.F1
    int F1(int i);                // will pick non-generic

    void F2(I1<U> a);             // Valid overload
    void F2(I2<U> a);
}

class G2<U,V>
{
    void F3(U u, V v);            // Valid, but overload resolution for
    void F3(V v, U u);            // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);        // Valid, but overload resolution for
    void F4(I1<V> v, U u);        // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2);      // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);             // valid overload
    void F6(out V v);
}
```

动态重载解析的编译时检查

对于大多数动态绑定操作，解析的可能候选项集在编译时是未知的。但在某些情况下，在编译时，候选集是已知的：

- 具有动态参数的静态方法调用
- 接收方不是动态表达式的实例方法调用
- 接收方不是动态表达式的索引器调用
- 具有动态参数的构造函数调用

在这些情况下，将对每个候选项执行有限的编译时检查，以查看是否有可能在运行时应用它们。此检查包括以下步骤：

- 分部类型推理：使用类型推理的规则推断不直接或间接依赖于类型 `dynamic` 的参数的任何类型参数。其余类型参数是未知的。
- 部分适用性检查：根据适用的函数成员检查适用性，但忽略类型未知的参数。
- 如果没有候选项通过此测试，则会发生编译时错误。

函数成员调用

本节介绍在运行时调用特定函数成员的过程。假定绑定时间进程已确定要调用的特定成员（可能通过将重载决策应用于一组候选函数成员）。

出于描述调用过程的目的，函数成员分为两类：

- 静态函数成员。这些是实例构造函数、静态方法、静态属性访问器和用户定义的运算符。静态函数成员始终

是非虚拟的。

- 实例函数成员。这些是实例方法、实例属性访问器和索引器访问器。实例函数成员既不是虚拟的也不是虚拟的，始终在特定的实例上调用。实例由实例表达式计算，它在函数成员内可作为 `this`（此访问）进行访问。

函数成员调用的运行时处理包括以下步骤，其中 `M` 是函数成员，如果 `M` 为实例成员，`E` 为实例表达式：

- 如果 `M` 是静态函数成员：
  - 参数列表按参数列表中所述进行计算。
  - 调用 `M`。
- 如果 `M` 是在*value_type*中声明的实例函数成员：
  - 计算 `E`。如果此计算导致异常，则不执行其他步骤。
  - 如果 `E` 未分类为变量，则创建 `E` 类型的临时本地变量，并将 `E` 的值分配给该变量。然后，将 `E` 重新分类为对该临时本地变量的引用。临时变量在 `M` 内作为 `this` 访问，但不能以其他方式访问。因此，只有在 `E` 为真正的变量时，调用方才能观察 `M` 对 `this` 所做的更改。
  - 参数列表按参数列表中所述进行计算。
  - 调用 `M`。`E` 引用的变量成为 `this` 引用的变量。
- 如果 `M` 是在*reference_type*中声明的实例函数成员：
  - 计算 `E`。如果此计算导致异常，则不执行其他步骤。
  - 参数列表按参数列表中所述进行计算。
  - 如果 `E` 的类型为*value_type*，则执行装箱转换（装箱转换）以将 `E` 转换为类型 `object`，并在以下步骤中将 `E` 视为类型 `object`。在这种情况下，`M` 只能是 `System.Object` 的成员。
  - 检查 `E` 的值是否有效。如果 `null``E` 的值，则将引发 `System.NullReferenceException`，并且不执行任何其他步骤。
  - 确定要调用的函数成员实现：
    - 如果 `E` 的绑定时类型是接口，则调用的函数成员是由 `E` 引用的实例的运行时类型提供的 `M` 的实现。此函数成员是通过以下方式确定的：应用接口映射规则（接口映射），以确定 `E` 所引用的实例的运行时类型提供的 `M` 实现。
    - 否则，如果 `M` 为虚拟函数成员，则调用的函数成员是由 `E` 引用的实例的运行时类型提供的 `M` 的实现。此函数成员是通过应用规则来确定的，这些规则用于确定 `E` 所引用的实例的运行时类型 `M` 的派生程度最高的实现（虚拟方法）。
    - 否则，`M` 为非虚拟函数成员，调用的函数成员 `M` 本身。
  - 调用上述步骤中确定的函数成员实现。`E` 引用的对象成为 `this` 引用的对象。

在装箱实例上调用

在*value_type*中实现的函数成员可在以下情况下通过*value_type*的装箱实例进行调用：

- 当函数成员是从类型 `object` 继承的方法的 `override` 时，将通过类型 `object` 的实例表达式调用。
- 当函数成员是接口函数成员的实现并通过*interface_type*的实例表达式调用时。
- 函数成员通过委托调用时。

在这些情况下，已装箱的实例被视为包含*value_type*的变量，此变量将成为函数成员调用中 `this` 所引用的变量。具体而言，这意味着当对装箱实例调用函数成员时，该函数成员可以修改装箱实例中包含的值。

主要表达式

主要表达式包括表达式的最简单形式。

```
primary_expression
    : primary_no_array_creation_expression
    | array_creation_expression
    ;

primary_no_array_creation_expression
    : literal
    | interpolated_string_expression
    | simple_name
    | parenthesized_expression
    | member_access
    | invocation_expression
    | element_access
    | this_access
    | base_access
    | post_increment_expression
    | post_decrement_expression
    | object_creation_expression
    | delegate_creation_expression
    | anonymous_object_creation_expression
    | typeof_expression
    | checked_expression
    | unchecked_expression
    | default_value_expression
    | nameof_expression
    | anonymous_method_expression
    | primary_no_array_creation_expression_unsafe
    ;
```

主表达式划分*array_creation_expression*s 和*primary_no_array_creation_expression*之间。以这种方式对待数组创建表达式，而不是将其与其他简单的表达式窗体一起列出，而是允许语法禁止可能的代码混乱，如

```
object o = new int[3][1];
```

否则，会将其解释为

```
object o = (new int[3])[1];
```

文本

由*文本*(文本)组成的*primary_expression*分类为值。

内插字符串

*Interpolated_string_expression*包含一个 `$` 符号后跟一个常规或逐字字符串文本，其中的洞由 `{` 和 `}` 分隔，其中包含表达式和格式规范。内插字符串表达式是已分解为单独标记的*interpolated_string_literal*的结果，如内插字符串文本中所述。

```
interpolated_string_expression
    : '$' interpolated_regular_string
    | '$' interpolated_verbatim_string
    ;

interpolated_regular_string
    : interpolated_regular_string_whole
    | interpolated_regular_string_start interpolated_regular_string_body interpolated_regular_string_end
    ;

interpolated_regular_string_body
    : interpolation (interpolated_regular_string_mid interpolation)*
    ;

interpolation
    : expression
    | expression ',' constant_expression
    ;

interpolated_verbatim_string
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start interpolated_verbatim_string_body interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_body
    : interpolation (interpolated_verbatim_string_mid interpolation)+
    ;
```

内插中的*constant_expression*必须具有到 `int` 的隐式转换。

*Interpolated_string_expression*分类为值。如果它立即转换为 `System.IFormattable` 或使用隐式内插字符串转换（隐式内插的字符串转换）`System.FormattableString`，则内插字符串表达式具有该类型。否则，它的类型为 `string`。

如果内插字符串的类型是 `System.IFormattable` 或 `System.FormattableString`，则表示对 `System.Runtime.CompilerServices.FormattableStringFactory.Create` 的调用。如果类型 `string`，则表达式的含义是对 `string.Format` 的调用。在这两种情况下，调用的参数列表都包含格式字符串文本和每个内插的占位符，以及与占位符相对应的每个表达式的参数。

格式字符串文字按如下方式构造，其中 `N` 是*interpolated_string_expression*中的内插数：

- 如果*interpolated_regular_string_whole*或*interpolated_verbatim_string_whole*后跟 `$` 符号，则格式字符串文本就是该标记。
- 否则，格式字符串文本包含：
  - 首先*interpolated_regular_string_start*或*interpolated_verbatim_string_start*
  - 然后，将每个数字从 `0` `I` 到 `N-1`：
    - `I` 的十进制表示形式
    - 然后，如果相应的*内插*具有*constant_expression*，则 `,`（逗号），后跟的值的十进制表示形式*constant_expression*
    - 然后，*interpolated_regular_string_mid*、*interpolated_regular_string_end*、*interpolated_verbatim_string_mid*或*interpolated_verbatim_string_end*后跟相应的内插。

后续参数只是*内插*中的*表达式*（如果有），按顺序排列。

TODO：示例。

简单名称

*Simple_name*包含标识符，可选择后跟类型参数列表：

```
simple_name
    : identifier type_argument_list?
    ;
```

*Simple_name*格式为 `I` 或格式为 `I<A1,...,Ak>`，其中 `I` 是单个标识符，`<A1,...,Ak>` 是可选*type_argument_list*。当未指定*type_argument_list*时，请考虑 `K` 为零。将按如下所示对*simple_name*进行评估和分类：

- 如果 `K` 为零且*simple_name*出现在*块*中，并且*块*的（或封闭*块*的）局部变量声明空间（声明）包含名称 `I` 的局部变量、参数或常量，则*simple_name*引用该局部变量、参数或常量，并归类为变量或值。
- 如果 `K` 为零，且*simple_name*出现在泛型方法声明的主体中，并且该声明包含名称为 `I` 的类型参数，则*simple_name*引用该类型参数。
- 否则，对于每个实例类型 `T`（实例类型），从立即封闭类型声明的实例类型开始，并继续执行每个封闭类或结构声明的实例类型（如果有）：
  - 如果 `K` 为零，且 `T` 的声明包含名称为 `I` 的类型参数，则*simple_name*将引用该类型参数。
  - 否则，如果 `T` 中的 `I` 的成员查找（成员查找）`K` 类型参数生成匹配：
    - 如果 `T` 是直接封闭类或结构类型的实例类型，并且查找标识了一个或多个方法，则结果为具有 `this` 的关联实例表达式的方法组。如果指定了类型参数列表，它将用于调用泛型方法（方法调用）。
    - 否则，如果 `T` 是直接封闭类或结构类型的实例类型，如果查找标识实例成员，并且引用出现在实例构造函数、实例方法或实例访问器的主体中，则结果与 `this.I` 窗体的成员访问（成员访问）相同。仅当 `K` 为零时，才会发生这种情况。
    - 否则，结果将与表单的成员访问（成员访问）`T.I` 或 `T.I<A1,...,Ak>` 相同。在这种情况下，*simple_name*引用实例成员，则为绑定时错误。
- 否则，对于每个命名空间 `N`，从发生*simple_name*的命名空间开始，继续每个封闭命名空间（如果有），并以全局命名空间结束，将计算以下步骤直到找到实体：
  - 如果 `K` 为零且 `I` 是 `N` 中的命名空间的名称，则：
    - 如果*simple_name*发生的位置由 `N` 的命名空间声明括起来，并且命名空间声明包含将名称 `I` 与命名空间或类型相关联的*extern_alias_directive*或*using_alias_directive*，则*simple_name*为不明确，并发生编译时错误。
    - 否则，*simple_name*引用 `N` 中名为 `I` 的命名空间。
  - 否则，如果 `N` 包含名称 `I` 且 `K` 类型参数的可访问类型，则：
    - 如果 `K` 为零，且*simple_name*的位置由 `N` 的命名空间声明括起来，并且命名空间声明包含将名称 using_alias_directive 与命名空间或类型相关联的*extern_alias_directive*或 * `I` *，则*simple_name*是不明确的，并发生编译时错误。
    - 否则，*namespace_or_type_name*引用用给定类型参数构造的类型。
  - 否则，如果*simple_name*发生的位置由 `N` 的命名空间声明括起来：
    - 如果 `K` 为零，且命名空间声明包含将名称 `I` 与导入的命名空间或类型相关联的*extern_alias_directive*或*using_alias_directive*，则*simple_name*引用该命名空间或类型。
    - 否则，如果由*using_namespace_directive*s 和*using_static_directive*s 命名空间声明导入的命名空间和类型声明包含一个名称 `I` 和 `K` 类型参数的可访问类型或非扩展静态成员，则*simple_name*引用使用给定类型参数构造的该类型或成员。
    - 否则，如果命名空间声明中*using_namespace_directive*s 所导入的命名空间和类型包含多个具有名称 `I` 和 `K` 类型参数的可访问的类型或非扩展方法静态成员，则*simple_name*不明确，并发生错误。

  请注意，整个步骤与处理*namespace_or_type_name*（命名空间和类型名称）的相应步骤完全相同。

- 否则，*simple_name*是未定义的，并且发生编译时错误。

带括号的表达式

*Parenthesized_expression*包含用括号括起来的*表达式*。

```
parenthesized_expression
    : '(' expression ')'
    ;
```

通过计算括号内的*表达式*来计算*parenthesized_expression* 。如果括号内的*表达式*表示命名空间或类型，则会发生编译时错误。否则，*parenthesized_expression*的结果是包含的*表达式*的计算结果。

成员访问

*Member_access*包含*primary_expression*、*predefined_type*或*qualified_alias_member*，后跟一个 "`.`" 标记，后跟一个*标识符*，可选择后跟*type_argument_list*。

```
member_access
    : primary_expression '.' identifier type_argument_list?
    | predefined_type '.' identifier type_argument_list?
    | qualified_alias_member '.' identifier
    ;

predefined_type
    : 'bool'   | 'byte'  | 'char'  | 'decimal' | 'double' | 'float' | 'int' | 'long'
    | 'object' | 'sbyte' | 'short' | 'string'  | 'uint'   | 'ulong' | 'ushort'
    ;
```

*Qualified_alias_member*生产是在命名空间别名限定符中定义的。

*Member_access*的格式为 `E.I` 或格式 `E.I<A1, ..., Ak>`，其中 `E` 是主表达式，`I` 是单个标识符，`<A1, ..., Ak>` 是可选*type_argument_list*。当未指定*type_argument_list*时，请考虑 `K` 为零。

*Primary_expression*类型 `dynamic` 的*member_access*是动态绑定的（动态绑定）。在这种情况下，编译器将成员访问分类为 `dynamic` 类型的属性访问。下面的规则使用运行时类型（而不是*primary_expression*的编译时类型）在运行时应用*member_access*的含义。如果此运行时分类导致了方法组，则成员访问必须是*invocation_expression*的*primary_expression* 。

将按如下所示对*member_access*进行评估和分类：

- 如果 `K` 为零且 `E` 为命名空间并且 `E` 包含名称 `I` 的嵌套命名空间，则结果为该命名空间。
- 否则，如果 `E` 是一个命名空间，并且 `E` 包含名称 `I` 和 `K` 类型参数的可访问类型，则结果为使用给定类型参数构造的类型。
- 如果 `E` 是*predefined_type*或归类为类型的*primary_expression* ，则如果 `E` 不是类型形参，并且 `I` 中 `E` 的成员查找（成员查找）使用 `K` ，则类型参数将生成匹配项，然后按如下方式计算 和分类：
  - 如果 `I` 标识一个类型，则结果是使用给定的类型参数构造的类型。
  - 如果 `I` 标识一个或多个方法，则结果是没有关联的实例表达式的方法组。如果指定了类型参数列表，它将用于调用泛型方法（方法调用）。
  - 如果 `I` 确定 `static` 属性，则结果为无关联的实例表达式的属性访问。
  - 如果 `I` 确定 `static` 字段：
    - 如果字段为 `readonly` 并且引用出现在声明该字段的类或结构的静态构造函数之外，则结果为值，即 `E` 中的静态字段 `I` 的值。
    - 否则，结果为变量，即 `E` 中 `I` 的静态字段。
  - 如果 `I` 标识 `static` 事件：
    - 如果引用出现在声明事件的类或结构中，并且声明事件时没有*event_accessor_declarations* （事

件)，则 `E.I` 的处理方式与 `I` 为静态字段的方式完全相同。
    - 否则，结果是没有关联的实例表达式的事件访问。
  - 如果 `I` 标识一个常量，则结果为值，即该常量的值。
  - 如果 `I` 标识枚举成员，则结果为值，即枚举成员的值。
  - 否则，`E.I` 是无效的成员引用，并发生编译时错误。
- 如果 `E` 是属性访问、索引器访问、变量或值、`T` 的类型，并且在 `T` 中包含 `K` 的 `I` 的成员查找(成员查找)，类型参数会生成匹配项，然后按如下所示对 `E.I` 进行计算和分类：
  - 首先，如果 `E` 是属性或索引器访问，则获取属性或索引器访问的值(表达式的值)，并将 `E` 重新分类为值。
  - 如果 `I` 标识了一个或多个方法，则结果将是具有 `E` 的关联实例表达式的方法组。如果指定了类型参数列表，它将用于调用泛型方法(方法调用)。
  - 如果 `I` 标识实例属性，
    - 如果 `E` `this`，`I` 将标识一个自动实现的属性(自动实现的属性)，而不使用 setter，并且引用出现在类或结构类型 `T` 的实例构造函数中，则结果是一个变量，即 `I` 给定的 `T` 的实例中的 `this` 所给定的自动属性的隐藏支持字段。
    - 否则，结果为具有 `E` 的关联实例表达式的属性访问。
  - 如果 `T` 是*class_type*并且 `I` 标识该*class_type*的实例字段：
    - 如果 `null``E` 的值，则将引发 `System.NullReferenceException`。
    - 否则，如果字段 `readonly` 并且引用出现在声明该字段的类的实例构造函数之外，则结果为值，即 `E` 引用的对象中的字段的值 `I`。
    - 否则，结果为变量，即 `E` 所引用的对象中 `I` 字段。
  - 如果 `T` 是*struct_type*并且 `I` 标识该*struct_type*的实例字段：
    - 如果 `E` 是一个值，或如果该字段 `readonly` 并且引用出现在声明该字段的结构的实例构造函数之外，则结果是一个值，即 `E` 指定的结构实例中的字段的值 `I`。
    - 否则，结果为变量，即 `E` 指定的结构实例中的字段 `I`。
  - 如果 `I` 标识实例事件：
    - 如果引用出现在声明了事件的类或结构中，并且事件是在没有*event_accessor_declarations*(事件)的情况下声明的，并且引用不是作为 `+=` 或 `-=` 运算符的左侧出现，则 `E.I` 的处理方式与 `I` 为实例字段的方式完全相同。
    - 否则，结果为具有 `E` 的关联实例表达式的事件访问。
- 否则，会尝试将 `E.I` 处理为扩展方法调用(扩展方法调用)。如果此操作失败，`E.I` 是无效的成员引用，并发生绑定时错误。

相同的简单名称和类型名称

在窗体 `E.I` 的成员访问中，如果 `E` 是单个标识符，并且 `E` 为*simple_name*(简单名称)是与 *`E`*(命名空间和类型名称)具有相同类型的常数、字段、属性、局部变量或参数，则允许 type_name 的两种可能含义。`E.I` 的两个可能含义不明确，因为 `I` 必须是这两种情况下 `E` 类型的成员。换言之，该规则只允许访问静态成员和嵌套类型的 `E`，在这种情况下，会发生编译时错误。例如：

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
}

class A
{
    public Color Color;                // Field Color of type Color

    void F() {
        Color = Color.Black;           // References Color.Black static member
        Color = Color.Complement();    // Invokes Complement() on Color field
    }

    static void G() {
        Color c = Color.White;         // References Color.White static member
    }
}
```

语法多义性

*Simple_name*的生产(简单名称)和*member_access* (成员访问)可以在表达式的语法中带来歧义。例如，语句：

```
F(G<A,B>(7));
```

可以解释为对具有两个自变量的 `F` 的调用 `G < A` 和 `B > (7)` 。或者，它可以解释为对 `F` 的调用，该参数是一个参数，它是对具有两个类型参数和一个正则参数的泛型方法的调用 `G` 。

如果可以将标记序列(在上下文中)作为*simple_name* (简单名称)、*member_access* (成员访问)或*pointer_member_access* (指针成员访问)以*type_argument_list* (类型参数)结尾，则将检查紧跟 `>` 结束标记的标记。如果是

```
(  )  ]  }  :  ;  ,  .  ?  ==  !=  |  ^
```

然后， *type_argument_list*将作为*simple_name*的一部分保留， *member_access*或*pointer_member_access* ，并放弃标记序列的任何其他可能分析。否则，不会将*type_argument_list*视为*simple_name*、*member_access*或*pointer_member_access*的一部分，即使没有其他可能的标记序列分析也是如此。请注意，分析*namespace_or_type_name*中的*type_argument_list* (命名空间和类型名称)时，不会应用这些规则。语句

```
F(G<A,B>(7));
```

根据此规则，将解释为对 `F` 的调用，该参数是一个参数，它是对具有两个类型参数和一个正则参数的泛型方法的调用 `G` 。语句

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

每个都将解释为对具有两个自变量的 `F` 的调用。语句

```
x = F < A > +y;
```

将解释为小于运算符、大于运算符和一元正运算符，就好像语句已 `x = (F < A) > (+y)` 编写，而不是

以*type_argument_list*后跟二元加运算符的*simple_name*。在语句中

```
x = y is C<T> + z;
```

标记 `C<T>` 被解释为具有*type_argument_list*的*namespace_or_type_name*。

调用表达式

用于调用方法的*invocation_expression*。

```
invocation_expression
    : primary_expression '(' argument_list? ')'
    ;
```

如果以下至少一项保留，则*invocation_expression*是动态绑定的（动态绑定）：

- *Primary_expression*具有 `dynamic` 的编译时类型。
- 至少一个可选*argument_list*的参数 `dynamic` 编译时类型，而*primary_expression*没有委托类型。

在这种情况下，编译器将*invocation_expression*分类为 `dynamic` 类型的值。下面的规则用于确定*invocation_expression*的含义，然后使用运行时类型（而不是*primary_expression*的参数和编译时类型 `dynamic` 的编译时类型）在运行时应用。如果*primary_expression*没有 `dynamic` 的编译时类型，则方法调用会经历有限的编译时检查，如编译时检查动态重载决策中所述。

*Invocation_expression*的*primary_expression*必须为方法组或*delegate_type*的值。如果*primary_expression*为方法组，则*invocation_expression*为方法调用（方法调用）。如果*primary_expression*是*delegate_type*的值，则*Invocation_expression*为委托调用（委托调用）。如果*primary_expression*既不是方法组也不是*delegate_type*的值，则发生绑定时错误。

可选*argument_list*（参数列表）为方法的参数提供值或变量引用。

评估*invocation_expression*的结果归类如下：

- 如果*invocation_expression*调用返回 `void` 的方法或委托，则结果为 nothing。仅在*statement_expression*（expression 语句）的上下文中或作为*lambda_expression*（匿名函数表达式）的主体时，才允许将表达式归类为 nothing。否则，会发生绑定时错误。
- 否则，结果为方法或委托返回的类型的值。

方法调用

对于方法调用，*invocation_expression*的*primary_expression*必须为方法组。方法组标识一种要调用的方法或用于从中选择要调用的特定方法的重载方法集。在后一种情况下，确定要调用的特定方法的方式取决于*argument_list*中的参数类型所提供的上下文。

对形式的方法调用的绑定时处理 `M(A)`，其中 `M` 是一个方法组（可能包括一个*type_argument_list*），而 `A` 是一个可选*argument_list*，其中包含以下步骤：

- 构造方法调用的候选方法集。对于每个与方法组相关联 `F` 方法 `M`：
  - 如果 `F` 不是泛型的，则在以下情况下 `F` 为候选项：
    - `M` 没有类型参数列表，并且
    - `F` 适用于 `A`（适用的函数成员）。
  - 如果 `F` 是泛型且 `M` 没有类型参数列表，则在以下情况下 `F` 是候选项：
    - 类型推理（类型推理）成功，推断调用的类型参数列表，
    - 将推断类型参数替换为相应的方法类型参数后，F 的参数列表中的所有构造类型都满足其约束（满足约束），`F` 的参数列表适用于 `A`（适用的函数成员）。
  - 如果 `F` 是泛型并且 `M` 包含类型参数列表，则在以下情况下 `F` 是候选项：

    - `F` 具有与类型参数列表中提供的相同数量的方法类型参数，并且
    - 将类型参数替换为相应的方法类型参数后，F 的参数列表中的所有构造类型都满足其约束（满足约束），并且 `F` 的参数列表适用于 `A`（适用的函数成员）。
- 候选方法集减少为仅包含派生程度最高的类型中的方法：对于集中 `C.F` 的每个方法，其中 `C` 是声明方法 `F` 的类型，而在 `C` 的基类型中声明的所有方法将从集合中删除。此外，如果 `C` 是 `object` 以外的类类型，则将从集合中删除在接口类型中声明的所有方法。（仅当方法组是具有非 object 和非空有效接口集的有效基类的类型形参上的成员查找结果时，这一规则才会生效。）
- 如果生成的候选方法集为空，则将放弃按照以下步骤进行的进一步处理，而是尝试将调用作为扩展方法调用（扩展方法调用）进行处理。如果此操作失败，则不存在适用的方法，并且发生绑定时错误。
- 使用重载决策的重载决策规则标识候选方法集的最佳方法。如果无法识别单个最佳方法，则方法调用是不明确的，并发生绑定时错误。在执行重载决策时，将在替换相应方法类型参数的类型参数（提供或推断）后考虑泛型方法的参数。
- 执行所选最佳方法的最终验证：
    - 方法在方法组的上下文中进行验证：如果最佳方法为静态方法，则方法组必须通过类型*simple_name*或*member_access* 。如果最佳方法是实例方法，则方法组必须由*simple_name*、*member_access*通过变量或值或*base_access*导致。如果这两个要求都不成立，则发生绑定时错误。
    - 如果最佳方法是泛型方法，则会对照泛型方法上声明的约束（满足约束）检查类型参数（提供或推断）。如果任何类型自变量不满足类型参数上的相应约束，则会发生绑定时错误。

按照上述步骤在绑定时选择并验证方法后，将根据动态重载解析的编译时检查中所述的函数成员调用规则来处理实际的运行时调用。

上述解决方法的直观效果如下所示：若要查找由方法调用调用的特定方法，请从方法调用指示的类型开始，然后继续继承链，直到至少有一个适用的、可访问，但却找到了非替代方法声明。然后，对该类型中声明的适用的、可访问的、非重写方法集执行类型推理和重载决策，并调用此方法。如果未找到方法，请改为尝试将调用作为扩展方法调用来处理。

扩展方法调用

在方法调用中（装箱实例上的调用），其中一个窗体

```
expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )
```

如果调用的正常处理找不到适用的方法，则会尝试将构造作为扩展方法调用来处理。如果*expr*或任意*参数*的编译时类型 `dynamic`，则扩展方法将不适用。

目标是查找最佳*type_name* `C`，以便可以进行相应的静态方法调用：

```
C . identifier ( expr )

C . identifier ( expr , args )

C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )
```

如果满足以下条件，扩展方法 `Ci.Mj` ***适用***：

- `Ci` 是非泛型非泛型类
- `Mj` 的名称是*标识符*

- 当作为如下所示的静态方法应用于参数时，`Mj` 是可访问和适用的
- 存在从*expr*到 `Mj` 的第一个参数类型的隐式标识、引用或装箱转换。

搜索 `C` 按如下方式进行：

- 从最接近的封闭命名空间声明开始，继续每个封闭命名空间声明，并以包含编译单元结束，接下来尝试查找一组候选的扩展方法：
  - 如果给定的命名空间或编译单元直接包含 `Ci` 具有符合条件的扩展 `Mj` 方法的非泛型类型声明，则这些扩展方法集是候选集。
  - 如果在给定的命名空间或编译单元中*using_namespace_directive*s 导入的命名空间中 `Ci` 导入*using_static_declarations*并直接声明的类型 `Mj`，则这些扩展方法集是候选集。
- 如果在任何封闭命名空间声明或编译单元中均未找到候选集，则会发生编译时错误。
- 否则，将重载决策应用于候选集，如中所述（重载决策）。如果未找到单个最佳方法，则会发生编译时错误。
- `C` 是将最佳方法声明为扩展方法的类型。

将 `C` 用作目标，然后将方法调用处理为静态方法调用（对动态重载决策进行编译时检查）。

前面的规则意味着实例方法优先于扩展方法，内部命名空间声明中提供的扩展方法优先于外部命名空间声明中提供的扩展方法和该扩展直接在命名空间中声明的方法优先于通过使用命名空间指令导入到同一命名空间中的扩展方法。例如：

```
public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}

class X
{
    static void Test(A a, B b, C c) {
        a.F(1);             // E.F(object, int)
        a.F("hello");       // E.F(object, string)

        b.F(1);             // B.F(int)
        b.F("hello");       // E.F(object, string)

        c.F(1);             // C.F(object)
        c.F("hello");       // C.F(object)
    }
}
```

在此示例中，`B` 的方法优先于第一个扩展方法，`C` 的方法优先于这两个扩展方法。

```
public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
        public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
    }
}

namespace N2
{
    using N1;

    public static class E
    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i); }
    }

    class Test
    {
        static void Main(string[] args)
        {
            1.F();
            2.G();
            3.H();
        }
    }
}
```

此示例的输出为：

```
E.F(1)
D.G(2)
C.H(3)
```

`D.G` 优先于 `C.G`，`E.F` 优先于 `D.F` 和 `C.F`。

委托调用

对于委托调用，*invocation_expression*的*primary_expression*必须是*delegate_type*的值。此外，将*delegate_type*视为与*delegate_type*具有相同参数列表的函数成员，*delegate_type*必须适用于与*invocation_expression*的*argument_list*相关的（适用的函数成员）。

`D(A)` 格式的委托调用的运行时处理，其中 `D` 是*delegate_type*的*primary_expression*，`A` 是一个可选的*argument_list*，其中包含以下步骤：

- 计算 `D`。如果此计算导致异常，则不执行其他步骤。
- 检查 `D` 的值是否有效。如果 `null``D` 的值，则将引发 `System.NullReferenceException`，并且不执行任何其他步骤。
- 否则，`D` 是对委托实例的引用。对委托调用列表中的每个可调用实体执行函数成员调用（对动态重载决策进行编译时检查）。对于包含实例和实例方法的可调用实体，调用实例是包含在可调用实体中的实例。

元素访问

*Element_access*包含一个*primary_no_array_creation_expression*，后跟一个 "`[`" 标记，后跟一个*argument_list*，

后跟一个 "`]`" 标记。*Argument_list*由一个或多个由逗号分隔的*参数*组成。

```
element_access
    : primary_no_array_creation_expression '[' expression_list ']'
    ;
```

不允许*element_access*的*argument_list*包含 `ref` 或 `out` 参数。

如果以下至少一项保留，则*element_access*是动态绑定的(动态绑定)：

- *Primary_no_array_creation_expression*具有 `dynamic` 的编译时类型。
- 至少一个*argument_list*的表达式 `dynamic` 有编译时类型，而*primary_no_array_creation_expression*没有数组类型。

在这种情况下，编译器将*element_access*分类为 `dynamic` 类型的值。下面的规则用于确定*element_access*的含义，然后使用运行时类型(而不是*primary_no_array_creation_expression*和*argument_list*具有编译时类型 `dynamic` 的表达式的编译时类型)在运行时应用。如果*primary_no_array_creation_expression*没有 `dynamic` 的编译时类型，则元素访问会经历有限的编译时检查，如对动态重载决策进行编译时检查中所述。

如果*element_access*的*primary_no_array_creation_expression*是*array_type*的值，则*element_access*是数组访问(数组访问)。否则，*primary_no_array_creation_expression*必须是具有一个或多个索引器成员的类、结构或接口类型的变量或值，在这种情况下，*element_access*是索引器访问(索引器访问)。

数组访问

对于数组访问，*element_access*的*primary_no_array_creation_expression*必须是*array_type*的值。而且，不允许数组访问的*argument_list*包含命名参数。*Argument_list*中的表达式的数目必须与*array_type*的排名相同，并且每个表达式的类型必须为 `int`、`uint`、`long`、`ulong`，或者必须能够隐式转换为这些类型中的一个或多个。

计算数组访问的结果是数组的元素类型的一个变量，即由*argument_list*中的表达式的值所选择的数组元素。

`P[A]` 格式的数组访问的运行时处理，其中 `P` 是*array_type*的*primary_no_array_creation_expression*，`A` 是*argument_list*，其中包含以下步骤：

- 计算 `P`。如果此计算导致异常，则不执行其他步骤。
- 将按从左到右的顺序计算*argument_list*的索引表达式。对于每个索引表达式的求值，将执行隐式转换(隐式转换)为以下类型之一：`int`、`uint`、`long``ulong`。在此列表中选择了隐式转换的第一种类型。例如，如果索引表达式的类型为 `short` 则执行到 `int` 的隐式转换，因为从 `short` 到 `int` 的隐式转换和从 `short` 到 `long` 都是可能的。如果索引表达式的计算或后续的隐式转换导致异常，则不会再计算索引表达式，也不会执行任何其他步骤。
- 检查 `P` 的值是否有效。如果 `null``P` 的值，则将引发 `System.NullReferenceException`，并且不执行任何其他步骤。
- 根据 `P` 引用的数组实例的每个维度的实际界限检查*argument_list*中每个表达式的值。如果一个或多个值超出范围，则会引发 `System.IndexOutOfRangeException`，而不执行进一步的步骤。
- 计算索引表达式给定的数组元素的位置，此位置成为数组访问的结果。

索引器访问

对于索引器访问，*element_access*的*primary_no_array_creation_expression*必须是类、结构或接口类型的变量或值，并且此类型必须实现适用于*element_access*的*argument_list*的一个或多个索引器。

对窗体 `P[A]` 的索引器访问的绑定时处理，其中 `P` 是类、结构或接口类型 `T` 的*primary_no_array_creation_expression*，`A` 是*argument_list*，其中包含以下步骤：

- 构造由 `T` 提供的一组索引器。该集包含在 `T` 中声明的所有索引器，或不 `override` 声明并且可在当前上下文(成员访问)中访问的 `T` 基类型。
- 该集被简化为那些适用且不被其他索引器隐藏的索引器。以下规则应用于集中的每个索引器 `S.I`，其中 `S`

是声明索引器 `I` 的类型：

- 如果 `I` 不适用于 `A`（适用的函数成员），则从集中删除 `I`。
- 如果 `I` 适用于 `A`（适用的函数成员），则将从集合中删除在 `S` 的基类型中声明的所有索引器。
- 如果 `I` 适用于 `A`（适用的函数成员），而 `S` 是 `object` 以外的类类型，则将从集合中删除在接口中声明的所有索引器。

- 如果生成的候选索引器集为空，则不存在适用的索引器，并且发生绑定时错误。
- 使用重载决策的重载决策规则识别候选索引器集的最佳索引器。如果无法识别单个最佳索引器，索引器访问不明确，并发生绑定时错误。
- 将按从左到右的顺序计算*argument_list*的索引表达式。处理索引器访问的结果是分类为索引器访问的表达式。索引器访问表达式引用前面步骤中确定的索引器，并且具有 `P` 的关联实例表达式和 `A` 的关联自变量列表。

索引器访问会导致索引器的*get*访问器或*set 访问*器的调用，具体取决于使用它的上下文。如果索引器访问是赋值的目标，则调用*set 访问器*来分配新值（简单赋值）。在所有其他情况下，将调用*get 访问器*以获取当前值（表达式的值）。

此访问

*This_access*包含保留字 `this`。

```
this_access
    : 'this'
    ;
```

只能在实例构造函数、实例方法或实例访问器的*块*中使用*this_access* 。它具有下列含义之一：

- 当在类的实例构造函数中的*primary_expression*中使用 `this` 时，它将分类为值。值的类型为在其中发生使用的类的实例类型（实例类型），值是对正在构造的对象的引用。
- 当在类的实例方法或实例访问器中的*primary_expression*中使用 `this` 时，它将分类为值。值的类型为在其中发生使用的类的实例类型（实例类型），值是对为其调用方法或访问器的对象的引用。
- 如果在结构的实例构造函数中的*primary_expression*中使用 `this`，则将其归类为变量。变量的类型是在其中发生使用的结构的实例类型（实例类型），而变量表示正在构造的结构。结构的实例构造函数的 `this` 变量的行为与结构类型的 `out` 参数完全相同，具体而言，这意味着必须在实例构造函数的每个执行路径中明确分配变量。
- 如果在结构的实例方法或实例访问器中的*primary_expression*中使用 `this`，则将其归类为变量。变量的类型是发生使用的结构的实例类型（实例类型）。
  - 如果方法或访问器不是迭代器（迭代器），则 `this` 变量表示为其调用方法或访问器的结构，其行为与结构类型的 `ref` 参数完全相同。
  - 如果方法或访问器是迭代器，则 `this` 变量表示为其调用了方法或访问器的结构的副本，其行为与结构类型的值参数完全相同。

如果在不是上面列出的上下文中的*primary_expression*中使用 `this`，则会发生编译时错误。具体而言，不能引用静态方法、静态属性访问器或字段声明的*variable_initializer*中的 `this`。

基本访问权限

*Base_access*包含保留字 `base` 后跟 "`.`" 标记、标识符或用方括号括起来的*argument_list* ：

```
base_access
    : 'base' '.' identifier
    | 'base' '[' expression_list ']'
    ;
```

*Base_access*用于访问当前类或结构中名称类似的成员隐藏的基类成员。只能在实例构造函数、实例方法或实例

访问器的*块*中使用*base_access*。当在类或结构中发生 `base.I` 时，`I` 必须表示该类或结构的基类的成员。同样，在类中发生 `base[E]` 时，基类中必须存在适用的索引器。

在绑定时，`base.I` 和 `base[E]` 形式的*base_access*表达式的计算方式与 `((B)this).I` 和 `((B)this)[E]` 编写时完全一样，其中 `B` 是出现构造的类或结构的基类。因此，`base.I` 和 `base[E]` 对应于 `this.I` 和 `this[E]`，只不过 `this` 被视为基类的实例。

当*base_access*引用虚拟函数成员（方法、属性或索引器）时，将更改在运行时调用哪个函数成员（动态重载决策的编译时检查）。调用的函数成员是通过查找与 `B` 相关的函数成员的派生程度最高（虚拟方法）来确定的，而不是与 `this` 的运行时类型相关，而在非基本访问中通常是如此。因此，在 `virtual` 函数成员的 `override` 中，*base_access*可用于调用函数成员的继承实现。如果*base_access*引用的函数成员是抽象的，则会发生绑定时错误。

后缀增量和减量运算符

```
post_increment_expression
    : primary_expression '++'
    ;

post_decrement_expression
    : primary_expression '--'
    ;
```

后缀递增或递减运算的操作数必须是分类为变量、属性访问或索引器访问的表达式。操作的结果是与操作数类型相同的值。

如果*primary_expression*具有编译时类型 `dynamic` 则运算符是动态绑定的（动态绑定），*post_increment_expression*或*post_decrement_expression*具有编译时类型 `dynamic` 并且以下规则在运行时使用*primary_expression*的运行时类型应用。

如果后缀递增或递减运算的操作数是属性或索引器访问，则属性或索引器必须同时具有 `get` 和 `set` 访问器。如果不是这种情况，则会发生绑定时错误。

一元运算符重载决策（一元运算符重载决策）应用于选择特定的运算符实现。为以下类型存在预定义的 `++` 和 `--` 运算符：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 和任何枚举类型。预定义的 `++` 运算符返回通过向操作数添加1而生成的值，预定义的 `--` 运算符返回通过从操作数中减去1而生成的值。在 `checked` 上下文中，如果此加法或减法的结果超出了结果类型的范围，且结果类型为整型类型或枚举类型，则将引发 `System.OverflowException`。

`x++` 或 `x--` 形式的后缀递增或递减运算的运行时处理包括以下步骤：

- 如果 `x` 归类为变量：
  - 计算 `x` 以产生变量。
  - 保存 `x` 的值。
  - 将调用选定的运算符，并将 `x` 的保存值作为其参数。
  - 运算符返回的值存储在 `x` 的计算给定的位置。
  - 的保存值 `x` 成为操作的结果。
- 如果 `x` 归类为属性或索引器访问：
  - 实例表达式（如果 `x` 不 `static`）和参数列表（如果 `x` 是索引器访问）与 `x` 关联，并在后续的 `get` 和 `set` 访问器调用中使用结果。
  - 调用 `x` 的 `get` 访问器，并保存返回值。
  - 将调用选定的运算符，并将 `x` 的保存值作为其参数。
  - 使用运算符作为其 `value` 参数返回的值调用 `x` 的 `set` 访问器。
  - 的保存值 `x` 成为操作的结果。

`++` 和 `--` 运算符还支持前缀表示法(前缀增量和减量运算符)。通常，`x++` 或 `x--` 的结果是操作前 `x` 的值，而 `++x` 或 `--x` 的结果是操作后 `x` 的值。在任一情况下，操作后 `x` 本身具有相同的值。

可以使用后缀或前缀表示法来调用 `operator ++` 或 `operator --` 实现。对于这两个表示法，不能有单独的运算符实现。

调用 new 运算符

`new` 运算符用于创建类型的新实例。

有三种形式的 `new` 表达式：

- 对象创建表达式用于创建类类型和值类型的新实例。
- 数组创建表达式用于创建数组类型的新实例。
- 委托创建表达式用于创建委托类型的新实例。

`new` 运算符表示创建类型的实例，但不一定表示动态分配内存。具体而言，值类型的实例不需要超出它们所在的变量的额外内存，并且当 `new` 用于创建值类型的实例时，不会发生动态分配。

对象创建表达式

*Object_creation_expression*用于创建*class_type*或*value_type*的新实例。

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    ;

object_or_collection_initializer
    : object_initializer
    | collection_initializer
    ;
```

*Object_creation_expression*的*类型*必须是*class_type*、*value_type*或*type_parameter*。*类型*不能是 `abstract` *class_type*。

仅当*类型*为*class_type*或*struct_type*时，才允许使用可选*argument_list*（参数列表）。

如果对象创建表达式包括对象初始值设定项或集合初始值设定项，则它可以省略构造函数参数列表和外括号。省略构造函数参数列表和外括号等效于指定空参数列表。

处理包括对象初始值设定项或集合初始值设定项的对象创建表达式包括首先处理实例构造函数，然后处理对象初始值设定项(对象初始值设定项)或集合初始值设定项(集合初始值设定项)指定的成员或元素初始化。

如果可选*argument_list*中的任何参数具有编译时类型 `dynamic` 则*object_creation_expression*为动态绑定(动态绑定)，并且以下规则在运行时使用编译时类型为 `dynamic` 的*argument_list*的参数的运行时类型。但是，对象创建会经历有限的编译时检查，如对动态重载决策进行编译时检查中所述。

格式为 `new T(A)` 的*object_creation_expression*的绑定时处理，其中 `T` 是*class_type*或*value_type*，`A` 是可选的*argument_list*，其中包含以下步骤：

- 如果 `T` 是*value_type*并且 `A` 不存在：
  - *Object_creation_expression*是默认的构造函数调用。*Object_creation_expression*的结果是 `T` 类型的值，即在 system.string类型中定义 `T` 的默认值。
- 否则，如果 `T` 是*type_parameter*并且 `A` 不存在：
  - 如果没有为 `T` 指定任何值类型约束或构造函数约束(类型参数约束)，将发生绑定时错误。
  - *Object_creation_expression*的结果是类型参数已绑定到的运行时类型的值，即调用该类型的默认构造函数的结果。运行时类型可以是引用类型或值类型。
- 否则，如果 `T` 是*class_type*或*struct_type*:

- 如果 `T` 是 `abstract` *class_type*，则会发生编译时错误。
- 使用重载决策的重载决策规则确定要调用的实例构造函数。候选实例构造函数集由 `T` 中声明的所有可访问实例构造函数组成，它们适用于 `A`（适用的函数成员）。如果候选实例构造函数集为空，或者无法识别单个最佳实例构造函数，则会发生绑定时错误。
- *Object_creation_expression*的结果是 `T` 类型的值，即通过调用上述步骤中确定的实例构造函数生成的值。

- 否则，*object_creation_expression*无效，并发生绑定时错误。

即使*object_creation_expression*是动态绑定的，编译时类型仍会 `T`。

格式为 `new T(A)` 的*object_creation_expression*的运行时处理，其中 `T` 是*class_type*或*struct_type*，`A` 是可选的*argument_list*，其中包含以下步骤：

- 如果 `T` 是*class_type*：
  - 分配 `T` 类的新实例。如果内存不足，无法分配新的实例，则会引发 `System.OutOfMemoryException`，而不执行进一步的步骤。
  - 新实例的所有字段都初始化为其默认值（默认值）。
  - 实例构造函数根据函数成员调用的规则进行调用（对动态重载决策进行编译时检查）。对新分配的实例的引用会自动传递到实例构造函数，并且可以从该构造函数内作为 `this` 访问该实例。
- 如果 `T` 是*struct_type*：
  - `T` 类型的实例是通过分配临时本地变量创建的。由于需要*struct_type*的实例构造函数来明确地为要创建的实例的每个字段赋值，因此不需要初始化临时变量。
  - 实例构造函数根据函数成员调用的规则进行调用（对动态重载决策进行编译时检查）。对新分配的实例的引用会自动传递到实例构造函数，并且可以从该构造函数内作为 `this` 访问该实例。

对象初始值设定项

*对象初始值设定项*为零个或多个字段、属性或对象的索引元素指定值。

```
object_initializer
    : '{' member_initializer_list? '}'
    | '{' member_initializer_list ',' '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : initializer_target '=' initializer_value
    ;

initializer_target
    : identifier
    | '[' argument_list ']'
    ;

initializer_value
    : expression
    | object_or_collection_initializer
    ;
```

对象初始值设定项由成员初始值设定项的序列组成，括在 `{` 和 `}` 标记中，并用逗号分隔。每个*member_initializer*为初始化指定一个目标。*标识符*必须命名要初始化的对象的可访问字段或属性，而用方括号括起来的*argument_list*必须在要初始化的对象上指定可访问索引器的参数。对象初始值设定项为同一个字段或属性包含多个成员初始值设定项是错误的。

每个*initializer_target*后跟一个等号和一个表达式、一个对象初始值设定项或集合初始值设定项。对象初始值设

定项中的表达式不能引用它要初始化的新创建的对象。

在按与目标的赋值(简单赋值)相同的方式处理等号后，指定表达式的成员初始值设定项。

在等号后指定对象初始值设定项的成员初始值设定项是***嵌套的对象初始值设定***项，即嵌入对象的初始化。嵌套的对象初始值设定项中的赋值被视为对该字段或属性的成员的赋值，而不是为字段或属性赋值。嵌套的对象初始值设定项不能应用于值类型的属性，也不能应用于具有值类型的只读字段。

在等号后指定集合初始值设定项的成员初始值设定项是嵌入集合的初始化。将初始值设定项中给定的元素添加到目标所引用的集合，而不是将新集合分配给目标字段、属性或索引器。目标必须为满足集合初始值设定项中指定的要求的集合类型。

索引初始值设定项的参数将始终只计算一次。因此，即使参数不会使用(例如，由于空的嵌套初始值设定项)，也会对其副作用进行评估。

下面的类表示一个具有两个坐标的点：

```
public class Point
{
    int x, y;

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

可以创建和初始化 `Point` 的实例，如下所示：

```
Point a = new Point { X = 0, Y = 1 };
```

这与

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

其中 `__a` 是其他不可见且无法访问的临时变量。下面的类表示从两个点创建的矩形：

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

可以创建和初始化 `Rectangle` 的实例，如下所示：

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

这与

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

其中 `__r`、`__p1` 和 `__p2` 是临时变量，这些变量不可见且不可访问。

如果 `Rectangle` 的构造函数分配两个嵌入的 `Point` 实例

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();

    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

以下构造可用于初始化嵌入的 `Point` 实例，而不是分配新实例：

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

这与

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

给定 C 的适当定义，以下示例：

```
var c = new C {
    x = true,
    y = { a = "Hello" },
    z = { 1, 2, 3 },
    ["x"] = 5,
    [0,0] = { "a", "b" },
    [1,2] = {}
};
```

等效于这一系列的赋值：

```
C __c = new C();
__c.x = true;
__c.y.a = "Hello";
__c.z.Add(1);
__c.z.Add(2);
__c.z.Add(3);
string __i1 = "x";
__c[__i1] = 5;
int __i2 = 0, __i3 = 0;
__c[__i2,__i3].Add("a");
__c[__i2,__i3].Add("b");
int __i4 = 1, __i5 = 2;
var c = __c;
```

其中 `__c` (等等)生成的变量不可见且无法访问源代码。请注意，`[0,0]` 的参数只计算一次，即使从未使用过，也会对 `[1,2]` 的参数进行一次计算。

集合初始值设定项

集合初始值设定项指定集合的元素。

```
collection_initializer
    : '{' element_initializer_list '}'
    | '{' element_initializer_list ',' '}'
    ;

element_initializer_list
    : element_initializer (',' element_initializer)*
    ;

element_initializer
    : non_assignment_expression
    | '{' expression_list '}'
    ;

expression_list
    : expression (',' expression)*
    ;
```

集合初始值设定项由元素初始值设定项序列组成，括在 `{` 和 `}` 标记之间，并用逗号分隔。每个元素初始值设定项指定一个元素，该元素将被添加到要初始化的集合对象，并由 `{` 和 `}` 令牌中包含的表达式列表以及用逗号分隔。单表达式元素初始值设定项可以不带大括号编写，但是不能是赋值表达式，以避免成员初始值设定项出现歧义。*Non_assignment_expression*生产是在expression中定义的。

下面是包含集合初始值设定项的对象创建表达式的示例：

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

集合初始值设定项应用于的集合对象必须是实现 `System.Collections.IEnumerable` 或发生编译时错误的类型。对于按顺序排列的每个指定元素，集合初始值设定项使用元素初始值设定项的表达式列表作为参数列表调用目标对象上的 `Add` 方法，并对每个调用应用常规成员查找和重载解析。因此，集合对象必须具有适用于每个元素初始值设定项 `Add` 名称的相应实例或扩展方法。

下面的类表示一个联系人，其中包含名称和电话号码的列表：

```
public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();

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

    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

可以按如下所示创建和初始化 `List<Contact>`：

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

这与

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

其中 `__clist`、`__c1` 和 `__c2` 是临时变量，这些变量不可见且不可访问。

数组创建表达式

*Array_creation_expression*用于创建*array_type*的新实例。

```
array_creation_expression
    : 'new' non_array_type '[' expression_list ']' rank_specifier* array_initializer?
    | 'new' array_type array_initializer
    | 'new' rank_specifier array_initializer
    ;
```

第一种形式的数组创建表达式将分配从表达式列表中删除每个表达式所得到的类型的数组实例。例如，数组创建表达式 `new int[10,20]` 生成 `int[,]` 类型的数组实例，而数组创建表达式 `new int[10][,]` 生成类型 `int[][,]` 的数组。表达式列表中的每个表达式的类型必须为 `int`、`uint`、`long` 或 `ulong`，或可隐式转换为这些类型中的一个或多个。每个表达式的值决定了新分配的数组实例中相应维度的长度。由于数组维度的长度必须为非负，因此在表达式列表中有一个具有负值的*constant_expression*编译时错误。

除非在不安全的上下文中(不安全上下文)，否则不指定数组的布局。

如果第一种形式的数组创建表达式包含数组初始值设定项，则表达式列表中的每个表达式都必须是常量，并且表达式列表指定的秩和维度长度必须与数组初始值设定项的匹配项和维度长度匹配。

在第二个或第三个窗体的数组创建表达式中，指定数组类型或秩说明符的秩必须与数组初始值设定项的秩匹配。各个维度长度是从数组初始值设定项的每个相应嵌套级别中的元素数推断出来的。因此，表达式

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

完全对应于

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

第三种形式的数组创建表达式称为***隐式类型化数组创建表达式***。它类似于第二个窗体，只不过数组的元素类型不是显式给定的，而是确定为数组初始值设定项中的一组表达式的最佳通用类型（查找一组表达式的最佳通用类型）。对于多维数组（即，其中*rank_specifier*至少包含一个逗号），此集包含在嵌套*array_initializer*中找到的所有*表达式*。

数组初始值设定项中进一步介绍了数组初始值设定项。

计算数组创建表达式的结果是一个值，即对新分配的数组实例的引用。数组创建表达式的运行时处理包括以下步骤：

- 将按从左到右的顺序计算*expression_list*的维度长度表达式。对每个表达式进行以下计算后，将执行隐式转换（隐式转换）为以下类型之一：`int`、`uint`、`long`、`ulong`。在此列表中选择了隐式转换的第一种类型。如果表达式的计算或后续的隐式转换导致异常，则不会计算进一步的表达式，也不会执行任何其他步骤。
- 按如下所示验证维度长度的计算值。如果一个或多个值小于零，则会引发 `System.OverflowException`，而不执行进一步的步骤。
- 分配具有给定维度长度的数组实例。如果内存不足，无法分配新的实例，则会引发 `System.OutOfMemoryException`，而不执行进一步的步骤。
- 新数组实例的所有元素都初始化为其默认值（默认值）。
- 如果数组创建表达式包含数组初始值设定项，则计算数组初始值设定项中的每个表达式并将其分配给其相应的数组元素。计算和分配按表达式在数组初始值设定项中写入的顺序执行，换言之，元素按递增的索引顺序进行初始化，最右边的维度首先增长。如果给定表达式的计算或对相应数组元素的后续赋值导致异常，则不会初始化其他元素（因此剩余的元素将具有其默认值）。

数组创建表达式允许实例化包含数组类型的元素的数组，但此类数组的元素必须手动初始化。例如，语句

```
int[][] a = new int[100][];
```

创建具有100类型 `int[]` 元素的一维数组。每个元素的初始值都是 `null`。同一个数组创建表达式不可能同时实例化子数组和语句

```
int[][] a = new int[100][5];        // Error
```

导致编译时错误。子数组的实例化必须改为手动执行，如下所示

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

当数组的数组具有 "矩形" 形状时，即当子数组的长度完全相同时，使用多维数组更为有效。在上面的示例中，数组数组的实例化将创建101对象，即一个外部数组和100子数组。相反，

```
int[,] = new int[100, 5];
```

仅创建单个对象和二维数组，并在单个语句中完成分配。

下面是隐式类型的数组创建表达式的示例：

```
var a = new[] { 1, 10, 100, 1000 };                     // int[]

var b = new[] { 1, 1.5, 2, 2.5 };                       // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" };                   // Error
```

最后一个表达式会导致编译时错误，因为 `int` 和 `string` 都不能隐式转换为另一种类型，因此没有最常见的类型。在这种情况下，必须使用显式类型化的数组创建表达式，例如指定要 `object[]` 的类型。或者，可以将其中一个元素强制转换为公共基类型，后者随后将成为推断元素类型。

隐式类型的数组创建表达式可与匿名对象初始值设定项（匿名对象创建表达式）结合，以创建匿名类型的数据结构。例如：

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

委托创建表达式

*Delegate_creation_expression*用于创建*delegate_type*的新实例。

```
delegate_creation_expression
    : 'new' delegate_type '(' expression ')'
    ;
```

委托创建表达式的参数必须是方法组、匿名函数或编译时类型的值 `dynamic` 或*delegate_type*。如果参数为方法组，则它将标识方法，并为实例方法标识要为其创建委托的对象。如果参数是匿名函数，它将直接定义委托目标的参数和方法主体。如果参数是一个值，则它标识要创建副本的委托实例。

如果*表达式*具有编译时类型 `dynamic`，则*delegate_creation_expression*是动态绑定的（动态绑定），下面的规则将使用*表达式*的运行时类型在运行时应用。否则，规则将在编译时应用。

格式为 `new D(E)` 的*delegate_creation_expression*的绑定时处理，其中 `D` 是一个*delegate_type*，`E` 是一个*表达式*，包含以下步骤：

- 如果 `E` 是方法组，则将使用与从 `E` 到 `D` 的方法组转换相同的方式来处理委托创建表达式。
- 如果 `E` 是匿名函数，则以与从 `E` 到 `D` 的匿名函数转换相同的方式处理委托创建表达式。
- 如果 `E` 是值，则 `E` 必须与 `D` 兼容（委托声明），并且结果是对类型 `D` 的新创建委托的引用，该委托引用与 `E` 相同的调用列表。如果 `E` 与 `D` 不兼容，则会发生编译时错误。

格式为 `new D(E)` 的*delegate_creation_expression*的运行时处理，其中 `D` 是一个*delegate_type*，`E` 是一个*表达式*，它包含以下步骤：

- 如果 `E` 是方法组，则将委托创建表达式作为方法组转换（方法组转换）作为 `E` `D` 进行计算。
- 如果 `E` 是匿名函数，则委托创建将作为从 `E` 到 `D`（匿名函数转换）的匿名函数转换进行计算。

- 如果 `E` 是*delegate_type*的值：
  - 计算 `E` 。如果此计算导致异常，则不执行其他步骤。
  - 如果 `null``E` 的值，则将引发 `System.NullReferenceException` ，并且不执行任何其他步骤。
  - 分配 `D` 委托类型的新实例。如果内存不足，无法分配新的实例，则会引发 `System.OutOfMemoryException` ，而不执行进一步的步骤。
  - 使用与 `E` 给定的委托实例相同的调用列表来初始化新的委托实例。

委托的调用列表是在对委托进行实例化时确定的，然后在该委托的整个生存期内保持不变。换句话说，创建委托后，不能更改其目标可调用的实体。合并两个委托或从另一个委托（委托声明）中删除一个委托时，将生成一个新委托;现有委托的内容未发生更改。

不能创建引用属性、索引器、用户定义的运算符、实例构造函数、析构函数或静态构造函数的委托。

如上所述，从方法组创建委托时，委托的形参列表和返回类型确定要选择的重载方法。示例中

```
delegate double DoubleFunc(double x);

class A
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }

    static double Square(double x) {
        return x * x;
    }
}
```

使用引用第二个 `Square` 方法的委托初始化 `A.f` 字段，因为该方法与 `DoubleFunc` 的形参列表和返回类型完全匹配。如果第二个 `Square` 方法不存在，则会发生编译时错误。

匿名对象创建表达式

*Anonymous_object_creation_expression*用于创建匿名类型的对象。

```
anonymous_object_creation_expression
    : 'new' anonymous_object_initializer
    ;

anonymous_object_initializer
    : '{' member_declarator_list? '}'
    | '{' member_declarator_list ',' '}'
    ;

member_declarator_list
    : member_declarator (',' member_declarator)*
    ;

member_declarator
    : simple_name
    | member_access
    | base_access
    | null_conditional_member_access
    | identifier '=' expression
    ;
```

匿名对象初始值设定项声明匿名类型并返回该类型的实例。匿名类型是直接从 `object` 继承的无类类类型。匿名类型的成员是从用于创建该类型实例的匿名对象初始值设定项中推断出的只读属性的序列。具体而言，是窗体的匿名对象初始值设定项

```
new { p1 = e1, p2 = e2, ..., pn = en }
```

声明窗体的匿名类型

```
class __Anonymous1
{
    private readonly T1 f1;
    private readonly T2 f2;
    ...
    private readonly Tn fn;

    public __Anonymous1(T1 a1, T2 a2, ..., Tn an) {
        f1 = a1;
        f2 = a2;
        ...
        fn = an;
    }

    public T1 p1 { get { return f1; } }
    public T2 p2 { get { return f2; } }
    ...
    public Tn pn { get { return fn; } }

    public override bool Equals(object __o) { ... }
    public override int GetHashCode() { ... }
}
```

其中每个 `Tx` 都是对应的表达式 `ex` 的类型。*Member_declarator*中使用的表达式的类型必须为。因此，*member_declarator*中的表达式为 null 或匿名函数是编译时错误。这也是表达式具有不安全类型的编译时错误。

匿名类型及其 `Equals` 方法的参数的名称由编译器自动生成，不能在程序文本中引用。

在同一程序中，两个用相同顺序指定相同名称和编译时类型的属性序列的匿名对象初始值设定项将生成相同的匿名类型的实例。

示例中

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

允许在最后一行上进行赋值，因为 `p1` 和 `p2` 是相同的匿名类型。

匿名类型上的 `Equals` 和 `GetHashcode` 方法会重写从 `object` 继承的方法，并在属性的 `Equals` 和 `GetHashcode` 中定义，因此，当且仅当所有属性都相等时，相同匿名类型的两个实例才相等。

成员声明符可以缩写为简单名称(类型推理)、成员访问(动态重载决策的编译时检查)、基访问(基本访问)或 null 条件成员访问(null 条件表达式作为投影初始值设定项)。这称为***投影初始值设定项***，是的声明和对具有相同名称的属性赋值的简写形式。具体而言，是窗体的成员声明符

```
identifier
expr.identifier
```

与以下各项完全等效：

```
identifier = identifier
identifier = expr.identifier
```

因此，在投影初始值设定项中，*标识符*既选择值，又选择要向其分配值的字段或属性。直观而言，投影初始值设定项不只是一个值，而是值的名称。

Typeof 运算符

`typeof` 运算符用于获取类型的 `System.Type` 对象。

```
typeof_expression
    : 'typeof' '(' type ')'
    | 'typeof' '(' unbound_type_name ')'
    | 'typeof' '(' 'void' ')'
    ;

unbound_type_name
    : identifier generic_dimension_specifier?
    | identifier '::' identifier generic_dimension_specifier?
    | unbound_type_name '.' identifier generic_dimension_specifier?
    ;

generic_dimension_specifier
    : '<' comma* '>'
    ;

comma
    : ','
    ;
```

第一种形式的*typeof_expression*由 `typeof` 关键字后跟带括号的*类型*组成。此窗体的表达式的结果是指定类型的 `System.Type` 对象。对于任何给定类型，只有一个 `System.Type` 对象。这意味着对于类型 `T`，`typeof(T) == typeof(T)` 始终为 true。无法 `dynamic` *类型*。

第二种形式的*typeof_expression*包含一个 `typeof` 关键字，后跟一个带圆括号的*unbound_type_name*。*Unbound_type_name*与*type_name*（命名空间和类型名称）非常相似，不同之处在于*unbound_type_name*包含*type_name*包含*type_argument_list*的*generic_dimension_specifier*。当*typeof_expression*的操作数为满足*unbound_type_name*和*type_name*语法的标记序列时，即当它既不包含*generic_dimension_specifier*也不包含*type_argument_list*时，将标记序列视为*type_name*。确定*unbound_type_name*的含义，如下所示：

- 将标记序列替换为*type_name*，方法是将每个*generic_dimension_specifier*替换为具有相同的逗号数的*type_argument_list*，并且关键字 `object` 每个*type_argument*。
- 计算生成的*type_name*，同时忽略所有类型参数约束。
- *Unbound_type_name*解析为与生成的构造类型（绑定的和未绑定的类型）关联的未绑定的泛型类型。

*Typeof_expression*的结果是生成的未绑定泛型类型的 `System.Type` 对象。

第三种形式的*typeof_expression*包含 `typeof` 关键字，后跟带括号的 `void` 关键字。此形式的表达式的结果是表示缺少类型的 `System.Type` 对象。`typeof(void)` 返回的类型对象不同于为任何类型返回的类型对象。此特殊类型对象在允许反射到语言中的方法的类库中很有用，在这种情况下，这些方法希望有一种方法来表示任何方法（包括 void 方法）的返回类型以及 `System.Type` 的实例。

可在类型参数上使用 `typeof` 运算符。结果是绑定到类型参数的运行时类型的 `System.Type` 对象。`typeof` 运算符还可用于构造类型或未绑定的泛型类型（绑定类型和未绑定类型）。未绑定的泛型类型的 `System.Type` 对象与实例类型的 `System.Type` 对象不同。实例类型在运行时始终是封闭式构造类型，因此它的 `System.Type` 对象取决于正在使用的运行时类型参数，而未绑定的泛型类型没有类型参数。

示例

```
using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}

class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

生成以下输出：

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

请注意，`int` 和 `System.Int32` 是相同的类型。

另请注意，`typeof(X<>)` 的结果不取决于类型参数，但 `typeof(X<T>)` 的结果。

checked 和 unchecked 运算符

`checked` 和 `unchecked` 运算符用于控制整型算术运算和转换的***溢出检查上下文***。

```
checked_expression
    : 'checked' '(' expression ')'
    ;

unchecked_expression
    : 'unchecked' '(' expression ')'
    ;
```

`checked` 运算符在已检查的上下文中计算包含的表达式，而 `unchecked` 运算符在未检查的上下文中计算包含的表达式。*Checked_expression*或*unchecked_expression*完全对应于*parenthesized_expression*（带括号的表达式），只是在给定溢出检查上下文中计算包含的表达式。

还可以通过 `checked` 和 `unchecked` 语句（checked 和 unchecked 语句）控制溢出检查上下文。

以下操作受 `checked` 和 `unchecked` 运算符和语句所建立的溢出检查上下文的影响：

- 当操作数为整数类型时，预定义的 `++` 和 `--` 一元运算符（后缀递增和递减运算符以及前缀增量和减量运算符）。
- 当操作数为整数类型时，预定义的 `-` 一元运算符（一元负运算符）。
- 预定义的 `+`、`-`、`*` 和 `/` 二元运算符（算术运算符），这两个操作数都是整数类型。
- 显式数值转换（显式数字转换）从一个整型类型转换为另一个整型类型，或从 `float` 或 `double` 转换为整型类型。

当上述操作之一产生的结果太大而无法在目标类型中表示时，执行该操作的上下文控制产生的行为：

- 在 `checked` 上下文中，如果操作是常量表达式（常量表达式），则会发生编译时错误。否则，在运行时执行操作时，将引发 `System.OverflowException`。
- 在 `unchecked` 上下文中，将放弃不适合目标类型的任何高序位，从而截断结果。

对于非常量表达式（在运行时计算的表达式），该表达式未由任何 `checked` 或 `unchecked` 运算符或语句括起来，则默认溢出检查上下文将 `unchecked`，除非 `checked` 计算的外部因素（如编译器开关和执行环境配置）调用。

对于常量表达式（可在编译时完全计算的表达式），默认溢出检查上下文始终 `checked`。除非在 `unchecked` 上下文中显式放置了一个常数表达式，否则在该表达式的编译时计算期间发生的溢出将始终导致编译时错误。

匿名函数的主体不受 `checked` 或发生匿名函数 `unchecked` 上下文的影响。

示例中

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);      // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Depends on default
    }
}
```

不会报告编译时错误，因为在编译时无法计算两个表达式。在运行时，`F` 方法引发 `System.OverflowException`，`G` 方法返回-727379968（超出范围的结果的32位）。`H` 方法的行为取决于编译的默认溢出检查上下文，但它与 `F` 或与 `G` 相同。

示例中

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);      // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Compile error, overflow
    }
}
```

在对 `F` 和 `H` 中的常量表达式进行计算时所发生的溢出会导致报告编译时错误，因为在 `checked` 上下文中对表达式进行求值。在 `G` 中计算常数表达式时也会发生溢出，但由于在 `unchecked` 上下文中发生了计算，因此不会报告溢出。

`checked` 和 `unchecked` 运算符仅对那些在 "`(`" 和 "`)`" 标记中包含的按 运算符对在计算包含的表达式时被调用的函数成员不起作用。示例中

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

在 `F` 中使用 `checked` 不会影响 `Multiply` 中的 `x * y` 计算，因此在默认溢出检查上下文中对 `x * y` 进行求值。

用十六进制表示法编写带符号整数类型的常量时，`unchecked` 运算符非常方便。例如：

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}
```

上述两个十六进制常量均为类型 `uint`。由于常量不在 `int` 范围内（没有 `unchecked` 运算符），因此强制转换为 `int` 会产生编译时错误。

`checked` 和 `unchecked` 运算符和语句使程序员能够控制某些数值计算的某些方面。但是，某些数值运算符的行为取决于其操作数的数据类型。例如，两个小数相乘始终导致溢出异常，即使在显式 `unchecked` 的构造中也是如此。同样，如果在显式 `checked` 的构造中，将两个浮点数相乘会导致溢出异常。此外，其他运算符永远不会受检查模式（无论是默认的还是显式的）的影响。

默认值表达式

默认值表达式用于获取类型的默认值（默认值）。通常，默认值表达式用于类型参数，因为如果类型参数是值类型或引用类型，则它可能不是已知的。（除非已知类型参数是引用类型，否则不存在从 `null` 文本到类型参数的转

换。）

```
default_value_expression
    : 'default' '(' type ')'
    ;
```

如果*default_value_expression*中的*类型*在运行时计算为引用类型，则结果将 `null` 转换为该类型。如果*default_value_expression*中的*类型*在运行时计算为值类型，则结果为*Value_type*的默认值（默认构造函数）。

如果类型是引用类型或已知为引用类型的类型参数（类型参数约束），则*default_value_expression*是常量表达式（常量表达式）。此外，如果该类型为以下值类型之一，则*default_value_expression*为常量表达式：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal`、`bool`或任意枚举类型。

Nameof 表达式

*Nameof_expression*用于获取作为常量字符串的程序实体的名称。

```
nameof_expression
    : 'nameof' '(' named_entity ')'
    ;

named_entity
    : simple_name
    | named_entity_target '.' identifier type_argument_list?
    ;

named_entity_target
    : 'this'
    | 'base'
    | named_entity
    | predefined_type
    | qualified_alias_member
    ;
```

语法上说，*named_entity*操作数始终是表达式。由于 `nameof` 不是保留关键字，因此，nameof 表达式在语法上始终不明确，因为调用简单名称 `nameof`。出于兼容性原因，如果名称的名称查找（简单名称）`nameof` 成功，则无论调用是否合法，都将表达式视为*invocation_expression* 。否则为*nameof_expression*。

*Nameof_expression*的*named_entity*的含义是表达式的含义;即，作为*simple_name*、*base_access*或*member_access*。但是，在简单名称和成员访问中描述的查找会导致错误，因为实例成员是在静态上下文中找到的，所以*nameof_expression*不产生这样的错误。

如果*named_entity*指定方法组具有*type_argument_list*，则会发生编译时错误。*Named_entity_target*的类型 `dynamic` 为编译时错误。

*Nameof_expression*是 `string` 类型的常量表达式，在运行时不起作用。具体而言，它的*named_entity*不会进行评估，出于明确的赋值分析目的将被忽略（简单表达式的一般规则）。它的值是可选的最终*type_argument_list*之前*named_entity*的最后一个标识符，转换方式如下：

- 如果使用前缀 "`@`"，则将其删除。
- 每个*unicode_escape_sequence*都转换为其对应的 unicode 字符。
- 删除任何*formatting_characters* 。

当在标识符之间测试相等性时，标识符中会应用相同的转换。

TODO：示例

匿名方法表达式

*Anonymous_method_expression*是定义匿名函数的两种方法之一。匿名函数表达式中进一步介绍了这些功能。

一元运算符

`?`、`+`、`-`、`!`、`~`、`++`、`--`、强制转换和 `await` 运算符称为一元运算符。

```
unary_expression
    : primary_expression
    | null_conditional_expression
    | '+' unary_expression
    | '-' unary_expression
    | '!' unary_expression
    | '~' unary_expression
    | pre_increment_expression
    | pre_decrement_expression
    | cast_expression
    | await_expression
    | unary_expression_unsafe
    ;
```

如果*unary_expression*的操作数 `dynamic` 编译时类型，则它是动态绑定的(动态绑定)。在这种情况下，将 `dynamic` *unary_expression*的编译时类型，并将使用操作数的运行时类型在运行时进行以下描述的解决方法。

Null 条件运算符

仅当操作数为非 null 时，null 条件运算符才将操作列表应用于其操作数。否则，将 `null` 应用运算符的结果。

```
null_conditional_expression
    : primary_expression null_conditional_operations
    ;

null_conditional_operations
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;
```

操作列表可以包括成员访问和元素访问操作(可能自身为空条件)以及调用。

例如，表达式 `a.b?[0]?.c()` 是一种具有*primary_expression* `a.b` 和*null_conditional_operations* `?[0]` (null 条件元素访问)、`?.c` (null 条件成员访问)和 `()` (调用)的*null_conditional_expression* 。

对于具有*primary_expression* `P` 的*null_conditional_expression* `E`，让 `E0` 是通过从每个包含一个的 `?` *null_conditional_operations*中删除前导 `E` 而获得的表达式。从概念上讲，`E0` 是将计算的表达式，如果 `?` 所表示的 null 检查没有找到 `null`，则会计算该表达式。

此外，让 `E1` 是通过从 `E` 中的第一个*null_conditional_operations*只删除前导 `?` 来获取的表达式。这可能会导致*主表达式*(如果只有一个 `?`)或另一个*null_conditional_expression*。

例如，如果 `E` 是表达式 `a.b?[0]?.c()`，则 `E0` 为表达式 `a.b[0].c()`，`E1` 为表达式 `a.b[0]?.c()`。

如果 `E0` 分类为 "无"，则 `E` 分类为 "无"。否则，将 E 分类为值。

`E0` 和 `E1` 用于确定 `E` 的含义：

- 如果 `E` 作为*statement_expression* `E` 的含义与语句相同

  ```
  if ((object)P != null) E1;
  ```

  但 P 只计算一次。

- 否则，如果 `E0` 归类为 nothing，则会发生编译时错误。
- 否则，让 `T0` 成为 `E0` 的类型。
  - 如果 `T0` 是未知类型参数，或者是不可以为 null 的值类型，则会发生编译时错误。
  - 如果 `T0` 是不可以为 null 的值类型，则 `E` 的类型 `T0?`，并且 `E` 的含义与

    ```
    ((object)P == null) ? (T0?)null : E1
    ```

    但 `P` 只计算一次。
  - 否则，E 的类型为 T0，E 的含义与

    ```
    ((object)P == null) ? null : E1
    ```

    但 `P` 只计算一次。

如果 `E1` 本身就是*null_conditional_expression*，则将再次应用这些规则，并为 `null` 嵌套测试，直到没有更多的 `?`，并且表达式已缩小到主-表达式 `E0`。

例如，如果表达式 `a.b?[0]?.c()` 作为语句表达式出现，如语句中所示：

```
a.b?[0]?.c();
```

其含义等效于：

```
if (a.b != null) a.b[0]?.c();
```

这同样等效于：

```
if (a.b != null) if (a.b[0] != null) a.b[0].c();
```

除了 `a.b` 和 `a.b[0]` 只计算一次。

如果它发生在使用其值的上下文中，如下所示：

```
var x = a.b?[0]?.c();
```

并且假设最终调用的类型不是可以为 null 的值类型，其含义等效于：

```
var x = (a.b == null) ? null : (a.b[0] == null) ? null : a.b[0].c();
```

除了 `a.b` 和 `a.b[0]` 只计算一次。

空条件表达式作为投影初始值设定项

如果在*anonymous_object_creation_expression*（匿名对象创建表达式）结束时，只允许使用 null 条件表达式作为*member_declarator*（也可以是 null 条件）成员访问。语法上，此要求可表示为：

```
null_conditional_member_access
    : primary_expression null_conditional_operations? '?' '.' identifier type_argument_list?
    | primary_expression null_conditional_operations '.' identifier type_argument_list?
    ;
```

这是上述*null_conditional_expression*语法的一种特殊情况。在匿名对象创建表达式中*member_declarator*的生产仅包括*null_conditional_member_access*。

空条件表达式作为语句表达式

如果以 null 条件表达式结束，则仅允许将其作为*statement_expression*（expression 语句）。语法上，此要求可表示为：

```
null_conditional_invocation_expression
    : primary_expression null_conditional_operations '(' argument_list? ')'
    ;
```

这是上述*null_conditional_expression*语法的一种特殊情况。然后，表达式语句中*statement_expression*的生产仅包括*null_conditional_invocation_expression*。

一元加运算符

对于 `+x` 形式的操作，将应用一元运算符重载决策（一元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型是运算符的返回类型。预定义的一元加法运算符是：

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

对于每个运算符，结果只是操作数的值。

一元减法运算符

对于 `-x` 形式的操作，将应用一元运算符重载决策（一元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型是运算符的返回类型。预定义的求反运算符是：

- 整数取反：

  ```
  int operator -(int x);
  long operator -(long x);
  ```

  通过从零中减去 `x` 来计算结果。如果 `x` 的值是操作数类型的最小可表示的值（对于 `int` 为-2 ^ 31，对于 `long`，则为-2 ^ 63），则不能在操作数类型内表示 `x` 的数学求反。如果在 `checked` 上下文中发生此情况，则会引发 `System.OverflowException`；如果它出现在 `unchecked` 上下文内，则结果为操作数的值，并且不报告溢出。

  如果求反运算符的操作数为类型 `uint`，则将其转换为类型 `long`，并 `long` 结果的类型。例外情况是允许将 `int` 值-2147483648（-2 ^ 31）编写为十进制整数文本（整数文本）的规则。

  如果求反运算符的操作数为类型 `ulong`，则会发生编译时错误。例外情况是允许将 `long` 值-9223372036854775808（-2 ^ 63）写入为十进制整数文本（整数文本）的规则。

- 浮点反：

```
float operator -(float x);
double operator -(double x);
```

结果为其符号反转 `x` 的值。如果 `x` 为 NaN，则结果也为 NaN。

- 小数求反：

```
decimal operator -(decimal x);
```

通过从零中减去 `x` 来计算结果。Decimal 求反等效于使用 `System.Decimal` 类型的一元减号运算符。

逻辑非运算符

对于 `!x` 形式的操作，将应用一元运算符重载决策（一元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型是运算符的返回类型。只存在一个预定义的逻辑求反运算符：

```
bool operator !(bool x);
```

此运算符计算操作数的逻辑非运算：如果操作数为 `true`，则结果为 `false`。如果操作数为 `false`，则结果为 `true`。

按位求补运算符

对于 `~x` 形式的操作，将应用一元运算符重载决策（一元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型是运算符的返回类型。预定义的按位求补运算符是：

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

对于每个运算符，操作的结果是 `x` 的按位求补。

每个枚举类型 `E` 隐式提供以下按位求补运算符：

```
E operator ~(E x);
```

计算 `~x` 的结果（其中，`x` 是 `E` 具有基础类型 `U` 的枚举类型的表达式）与计算 `(E)(~(U)x)` 完全相同，不同之处在于，在 `E` 上下文（检查的和未检查的运算符）中始终执行到 `unchecked` 的转换。

前缀增量和减量运算符

```
pre_increment_expression
    : '++' unary_expression
    ;

pre_decrement_expression
    : '--' unary_expression
    ;
```

前缀递增或递减运算的操作数必须是分类为变量、属性访问或索引器访问的表达式。操作的结果是与操作数类型相同的值。

如果前缀递增或递减运算的操作数是属性或索引器访问，则属性或索引器必须同时具有 `get` 和 `set` 访问器。如果不是这种情况，则会发生绑定时错误。

一元运算符重载决策(一元运算符重载决策)应用于选择特定的运算符实现。为以下类型存在预定义的 `++` 和 `--` 运算符：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 和任何枚举类型。预定义的 `++` 运算符返回通过向操作数添加1而生成的值，预定义的 `--` 运算符返回通过从操作数中减去1而生成的值。在 `checked` 上下文中，如果此加法或减法的结果超出了结果类型的范围，且结果类型为整型类型或枚举类型，则将引发 `System.OverflowException`。

`++x` 或 `--x` 形式的前缀递增或递减操作的运行时处理包括以下步骤：

- 如果 `x` 归类为变量：
  - 计算 `x` 以产生变量。
  - 将 `x` 所选运算符作为其参数。
  - 运算符返回的值存储在 `x` 的计算给定的位置。
  - 运算符返回的值成为操作的结果。
- 如果 `x` 归类为属性或索引器访问：
  - 实例表达式(如果 `x` 不 `static`)和参数列表(如果 `x` 是索引器访问)与 `x` 关联，并在后续的 `get` 和 `set` 访问器调用中使用结果。
  - 调用 `x` 的 `get` 访问器。
  - 使用 `get` 访问器返回的值作为其参数调用所选运算符。
  - 使用运算符作为其 `value` 参数返回的值调用 `x` 的 `set` 访问器。
  - 运算符返回的值成为操作的结果。

`++` 和 `--` 运算符还支持后缀表示法(后缀递增和递减运算符)。通常，`x++` 或 `x--` 的结果是操作前 `x` 的值，而 `++x` 或 `--x` 的结果是操作后 `x` 的值。在任一情况下，操作后 `x` 本身具有相同的值。

可以使用后缀或前缀表示法来调用 `operator++` 或 `operator--` 实现。对于这两个表示法，不能有单独的运算符实现。

强制转换表达式

*Cast_expression*用于将表达式显式转换为给定类型。

```
cast_expression
    : '(' type ')' unary_expression
    ;
```

格式为 `(T)E` 的*cast_expression*，其中 `T` 是一*种类型*，`E` 为*unary_expression*，则执行 `E` 的值到类型 `T` 的显式转换(显式转换)。如果不存在从 `E` 到 `T` 的显式转换，则发生绑定时错误。否则，结果为显式转换生成的值。即使 `E` 表示变量，结果也始终归类为值。

*Cast_expression*的语法导致某些语义歧义。例如，表达式 `(x)-y` 可以解释为*cast_expression*(将 `-y` 强制转换为类型 `x`)，或作为*additive_expression*与*parenthesized_expression*组合(这将计算该值 `x - y`)。

若要解决*cast_expression*歧义，请注意以下规则：仅当满足以下条件之一时，括在括号中的一个或多个*标记*为的序列(空格)才被视为*cast_expression*的开头：

- 标记序列对于*类型*是正确的语法，而不是*表达式*的语法。
- 标记序列对于*类型*是正确的语法，紧跟在右括号后面的标记是标记 "`~`"、标记 "`!`"、令牌 "`(`"、*标识符*(Unicode 字符转义序列)、*文本*(文本)或除 `as` 和 `is` 之外的任何*关键字*(关键字)。

上面的 "更正语法" 一词只是指标记序列必须符合特定的语法生产。具体而言，它不考虑任何构成标识符的实际含义。例如，如果 `x` 和 `y` 是标识符，则 `x.y` 是类型的正确语法，即使 `x.y` 实际上不表示类型也是如此。

从消除歧义规则开始，如果 `x` 和 `y` 为标识符、`(x)y`、`(x)(y)` 和 `(x)(-y)`，则 cast_expression 为 `(x)-y`，即使 `x` 标识一个类型，也是如此。但是，如果 `x` 是标识预定义类型(如 `int`)的关键字，则所有四个窗体*cast_expression*s(因为这种关键字本身不可能是表达式)。

Await 表达式

Await 运算符用于暂停封闭异步函数的计算，直到操作数表示的异步操作完成。

```
await_expression
    : 'await' unary_expression
    ;
```

只允许在异步函数（迭代器）的主体中使用*await_expression* 。在最近的封闭 async 函数中， *await_expression*可能不会出现在以下位置：

- 嵌套（非异步）匿名函数内
- 在*lock_statement*块内
- 在不安全的上下文中

请注意， *await_expression*不能出现在*query_expression*内的大多数位置，因为这些是在语法上转换为使用非 async lambda 表达式。

在异步函数内部，不能将 `await` 用作标识符。因此，await 表达式和涉及标识符的各种表达式之间没有语法多义性。在异步函数的外部，`await` 充当普通标识符。

*Await_expression*的操作数称为***任务***。它表示一个异步操作，该操作在计算*await_expression*时可能会也可能不会完成。Await 运算符的用途是在等待的任务完成之前挂起封闭异步函数的执行，然后获取其结果。

可等待表达式

Await 表达式的任务需要***可等待***。如果以下包含其中一项，则表达式 `t` 为可等待：

- `t` 的编译时类型为 `dynamic`
- `t` 具有一个名为 `GetAwaiter` 的可访问实例或扩展方法，其中没有任何参数和类型参数，以及以下所有保留 `A` 的返回类型：
  - `A` 实现接口 `System.Runtime.CompilerServices.INotifyCompletion`（对于简洁起见，为 `INotifyCompletion`）
  - `A` 具有类型的可访问的可读实例属性 `IsCompleted` `bool`
  - `A` 具有可访问的实例方法 `GetResult`，无任何参数和类型参数

`GetAwaiter` 方法的目的是获取任务的***awaiter*** 。类型 `A` 称为 await 表达式的***awaiter 类型***。

`IsCompleted` 属性的目的是确定任务是否已完成。如果是这样，则无需暂停评估。

`INotifyCompletion.OnCompleted` 方法的目的是将 "延续" 注册到任务;也就是说，将在任务完成后调用的委托（类型为 `System.Action`）。

`GetResult` 方法的目的是在任务完成后获取任务的结果。此结果可能会成功完成（可能包含结果值），也可能是由 `GetResult` 方法引发的异常。

Await 表达式的分类

表达式 `await t` 的分类方式与 `(t).GetAwaiter().GetResult()` 表达式的方式相同。因此，如果 `GetResult` 的返回类型为 `void`，则*await_expression*分类为 nothing。如果 `T`，则将*await_expression*分类为类型 `T` 的值。

Await 表达式的运行时计算

在运行时，表达式 `await t` 的计算方法如下：

- 通过 `(t).GetAwaiter()` 计算表达式来获取 awaiter `a`。
- 通过计算表达式 `(a).IsCompleted` 获取 `bool` `b`。
- 如果 `false` `b`，则评估将取决于 `a` 是否实现了接口 `System.Runtime.CompilerServices.ICriticalNotifyCompletion`（对于简洁起见，也称为

`ICriticalNotifyCompletion`)。此检查是在绑定时进行的;例如，如果 `a` 的编译时间类型 `dynamic`，则在运行时，否则为。让 `r` 表示恢复委托(迭代器)：

- 如果 `a` 未实现 `ICriticalNotifyCompletion`，则计算 `(a as (INotifyCompletion)).OnCompleted(r)` 表达式。
- 如果 `a` 实现 `ICriticalNotifyCompletion`，则计算 `(a as (ICriticalNotifyCompletion)).UnsafeOnCompleted(r)` 表达式。
- 然后，将挂起计算，并将控制权返回给 async 函数的当前调用方。

- 紧随之后(如果 `true``b`)，或在以后调用恢复委托(如果 `b` `false`)，将计算表达式 `(a).GetResult()`。如果返回值，则该值是*await_expression*的结果。否则，结果为 nothing。

Awaiter 实现的接口方法 `INotifyCompletion.OnCompleted` 和 `ICriticalNotifyCompletion.UnsafeOnCompleted` 应导致委托 `r` 最多调用一次。否则，封闭异步函数的行为是不确定的。

算术运算符

`*`、`/`、`%`、`+`和 `-` 运算符称为算术运算符。

```
multiplicative_expression
    : unary_expression
    | multiplicative_expression '*' unary_expression
    | multiplicative_expression '/' unary_expression
    | multiplicative_expression '%' unary_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
```

如果算术运算符的操作数具有编译时类型 `dynamic`，则该表达式将动态绑定(动态绑定)。在这种情况下，将 `dynamic` 表达式的编译时类型，并将使用编译时类型为 `dynamic` 的操作数的运行时类型在运行时执行下面描述的解决方法。

乘法运算符

对于 `x * y` 形式的操作，将应用二元运算符重载决策(二元运算符重载决策)来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

下面列出了预定义的乘法运算符。运算符都计算 `x` 和 `y` 的产品。

- 整数乘法：

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

在 `checked` 上下文中，如果该产品超出了结果类型的范围，则会引发 `System.OverflowException`。在 `unchecked` 上下文中，不会报告溢出，并且放弃结果类型范围外的任何重要高序位。

- 浮点乘法：

```
float operator *(float x, float y);
double operator *(double x, double y);
```

根据 IEEE 754 算法的规则计算产品。下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 为正有限值。`z` 是 `x * y` 的结果。如果结果对于目标类型来说太大，则 `z` 无限大。如果结果对于目标类型来说太小，则 `z` 为零。

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-z	+0	-0	+inf	-inf	NaN
-x	-z	+z	-0	+0	-inf	+inf	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+inf	+inf	-inf	NaN	NaN	+inf	-inf	NaN
-inf	-inf	+inf	NaN	NaN	-inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数乘法：

```
decimal operator *(decimal x, decimal y);
```

如果生成的值太大而无法表示为 `decimal` 格式，则会引发 `System.OverflowException`。如果结果值太小而无法表示为 `decimal` 格式，则结果为零。在进行任何舍入之前，结果的小数位数是两个操作数的刻度之和。

Decimal 等效于使用 `System.Decimal` 类型的乘法运算符。

除法运算符

对于 `x / y` 形式的操作，将应用二元运算符重载决策（二元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

下面列出了预定义的除法运算符。运算符都计算 `x` 和 `y` 的商。

- 整数相除：

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

如果右操作数的值为零，则会引发 `System.DivideByZeroException`。

相除将结果舍入到零。因此，结果的绝对值是小于或等于两个操作数的商的绝对值的最大整数。如果两个操作数具有相同的符号，则结果为零或正；如果两个操作数具有相反的符号，则结果为零或负。

如果左操作数是 `int` 或 `long` 值的最小表示，且右操作数 `-1`，则会发生溢出。在 `checked` 上下文中，这将导致引发 `System.ArithmeticException`（或其子类）。在 `unchecked` 上下文中，它的实现定义为：是否引发了 `System.ArithmeticException`（或其中的子类），或者溢出与左操作数的结果值无报告。

- 浮点除法：

```
float operator /(float x, float y);
double operator /(double x, double y);
```

根据 IEEE 754 算法的规则计算商。下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 为正有限值。`z` 是 `x / y` 的结果。如果结果对于目标类型来说太大，则 `z` 无限大。如果结果对于目标类型来说太小，则 `z` 为零。

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-z	+inf	-inf	+0	-0	NaN
-x	-z	+z	-inf	+inf	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+inf	+inf	-inf	+inf	-inf	NaN	NaN	NaN
-inf	-inf	+inf	-inf	+inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数部分：

```
decimal operator /(decimal x, decimal y);
```

如果右操作数的值为零，则会引发 `System.DivideByZeroException`。如果生成的值太大而无法表示为 `decimal` 格式，则会引发 `System.OverflowException`。如果结果值太小而无法表示为 `decimal` 格式，则结果为零。结果的小数位数是最小的小数位数，将结果等于最接近的可表示的十进制值到真正的数学结果。

小数部分等效于使用 `System.Decimal` 类型的除法运算符。

余数运算符

对于 `x % y` 形式的操作，将应用二元运算符重载决策（二元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

下面列出了预定义的余数运算符。运算符都计算 `x` 和 `y` 之间的相除余数。

- 整数余数：

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

`x % y` 的结果是 `x - (x / y) * y` 生成的值。如果 `y` 为零，则将引发 `System.DivideByZeroException`。

如果左操作数是 `int` 或 `long` 值的最小值，且右操作数 `-1`，则将引发 `System.OverflowException`。在任何情况下，`x % y` 引发异常，`x / y` 不会引发异常。

- 浮点余数：

```
float operator %(float x, float y);
double operator %(double x, double y);
```

下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 为正有限值。`z` 是 `x % y` 的结果，并按 `x - n * y` 计算，其中 `n` 是小于或等于 `x / y` 的最大整数。计算余数的这种方法类似于用于整数操作数，但不同于 IEEE 754 定义（其中 `n` 是最接近 `x / y` 的整数）。

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数余数：

```
decimal operator %(decimal x, decimal y);
```

如果右操作数的值为零，则会引发 `System.DivideByZeroException`。在进行舍入之前，结果的小数位数是两个操作数的刻度中较大的值，并且如果非零，则结果的符号与 `x` 相同。

小数余数等效于使用 `System.Decimal` 类型的余数运算符。

加法运算符

对于 `x + y` 形式的操作，将应用二元运算符重载决策（二元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

下面列出了预定义的加法运算符。对于数值类型和枚举类型，预定义的加法运算符计算两个操作数之和。当一个或两个操作数为字符串类型时，预定义的加法运算符将连接操作数的字符串表示形式。

- 整数加法：

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

在 `checked` 上下文中，如果求和超出了结果类型的范围，则会引发 `System.OverflowException`。在 `unchecked` 上下文中，不会报告溢出，并且放弃结果类型范围外的任何重要高序位。

- 浮点加法：

```
float operator +(float x, float y);
double operator +(double x, double y);
```

Sum 根据 IEEE 754 算法的规则进行计算。下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 是非零的有限值，`z` 是 `x + y` 的结果。如果 `x` 和 `y` 的量相同但符号相反，`z` 为正零。如果 `x + y` 太大而无法在目标类型中表示，则 `z` 使用与 `x + y` 相同的无穷。

	y	+0	-0	+inf	-inf	NaN
x	z	x	x	+inf	-inf	NaN
+0	y	+0	+0	+inf	-inf	NaN
-0	y	+0	-0	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数加：

```
decimal operator +(decimal x, decimal y);
```

如果生成的值太大而无法表示为 `decimal` 格式，则会引发 `System.OverflowException`。在进行舍入之前，结果的小数位数是两个操作数的刻度中较大的一个。

Decimal 加法等效于使用 `System.Decimal` 类型的加法运算符。

- 枚举加法。每个枚举类型都隐式提供以下预定义运算符，其中 `E` 是枚举类型，`U` 是 `E` 的基础类型：

```
E operator +(E x, U y);
E operator +(U x, E y);
```

在运行时，这些运算符的计算结果与 `(E)((U)x + (U)y)` 完全相同。

- 字符串串联：

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

二元 `+` 运算符的这些重载执行字符串串联。如果 `null` 字符串串联的操作数，则将替换空字符串。否则，通过调用从类型 `object` 继承的虚拟 `ToString` 方法，将任何非字符串参数转换为其字符串表示形式。如果 `ToString` 返回 `null`，则将替换空字符串。

```
using System;

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");        // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);               // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);               // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);               // displays d = 2.900
    }
}
```

字符串串联运算符的结果是一个字符串，该字符串由左操作数的字符后跟右操作数的字符组成。字符串串联运算符从不返回 `null` 值。如果没有足够的内存可用于分配生成的字符串，可能会引发 `System.OutOfMemoryException`。

- 委托组合。每个委托类型都隐式提供以下预定义运算符，其中 `D` 为委托类型：

```
D operator +(D x, D y);
```

二元 `+` 运算符在两个操作数均为某个委托类型 `D` 时执行委托组合。（如果操作数具有不同的委托类型，则会发生绑定时错误。）如果 `null` 第一个操作数，则操作的结果为第二个操作数的值（即使还 `null`）。否则，如果 `null` 第二个操作数，则运算结果为第一个操作数的值。否则，该操作的结果是一个新的委托实例，它在调用时调用第一个操作数，然后调用第二个操作数。有关委托组合的示例，请参阅减法运算符和委托调用。由于 `System.Delegate` 不是委托类型，因此不会为其定义 `operator +`。

减法运算符

对于 `x - y` 形式的操作，将应用二元运算符重载决策（二元运算符重载决策）来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

下面列出了预定义的减法运算符。运算符都从 `x` 中减去 `y`。

- 整数减法：

```
int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);
```

在 `checked` 上下文中，如果差异超出了结果类型的范围，则会引发 `System.OverflowException`。在 `unchecked` 上下文中，不会报告溢出，并且放弃结果类型范围外的任何重要高序位。

- 浮点减法：

```
float operator -(float x, float y);
double operator -(double x, double y);
```

根据 IEEE 754 算法的规则计算差异。下表列出了非零有限值、零、无穷大和 Nan 的所有可能组合的结果。在表中，`x` 和 `y` 是非零的有限值，`z` 是 `x - y` 的结果。如果 `x` 和 `y` 相等，`z` 为正零。如果 `x - y` 太大而无法在目标类型中表示，则 `z` 使用与 `x - y` 相同的无穷。

	y	+0	-0	+inf	-inf	NaN
x	z	x	x	-inf	+inf	NaN
+0	-y	+0	+0	-inf	+inf	NaN
-0	-y	-0	+0	-inf	+inf	NaN
+inf	+inf	+inf	+inf	NaN	+inf	NaN
-inf	-inf	-inf	-inf	-inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数减：

```
decimal operator -(decimal x, decimal y);
```

如果生成的值太大而无法表示为 `decimal` 格式，则会引发 `System.OverflowException`。在进行舍入之前，结果的小数位数是两个操作数的刻度中较大的一个。

Decimal 等效于使用 `System.Decimal` 类型的减法运算符。

- 枚举减法。每个枚举类型都隐式提供以下预定义运算符，其中 `E` 是枚举类型，`U` 是 `E` 的基础类型：

```
U operator -(E x, E y);
```

此运算符的计算结果与 `(U)((U)x - (U)y)` 完全相同。换言之，运算符计算 `x` 和 `y` 的序号值之间的差异，结果的类型为枚举的基础类型。

```
E operator -(E x, U y);
```

此运算符的计算结果与 `(E)((U)x - y)` 完全相同。换言之，运算符从枚举的基础类型中减去一个值，从而生成一个枚举值。

- 委托删除。每个委托类型都隐式提供以下预定义运算符，其中 `D` 为委托类型：

```
D operator -(D x, D y);
```

二元 `-` 运算符在两个操作数均为某个委托类型 `D` 时执行委托删除。如果操作数具有不同的委托类型，则会发生绑定时错误。如果第一个操作数为 `null`，则操作结果为 `null`。否则，如果 `null` 第二个操作数，则运算结果为第一个操作数的值。否则，两个操作数都表示具有一个或多个项的调用列表（委托声明），并且结果是一个新的调用列表，其中包含第一个操作数的列表（从中删除第二个操作数的条目），前提是第二个操作数的列表是第一个操作数的正确连续子列表。（若要确定子列表相等性，请将对应的项与委托相等运算符（委托相等运算符）进行比较。）否则，结果为左操作数的值。进程中的两个操作数都不是更改的。如果第二个操作数的列表与第一个操作数列表中连续条目的多个子列表匹配，则会删除最右侧匹配的子列表。如果删除行为导致出现空列表，则结果为 `null`。例如：

```
delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1;   // M1 + M2 + M2 + M1
        cd3 -= cd1;                      // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;                // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;                // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;                // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;                // => M1 + M2 + M2 + M1
    }
}
```

移位运算符

`<<` 和 `>>` 运算符用于执行移位运算。

```
shift_expression
    : additive_expression
    | shift_expression '<<' additive_expression
    | shift_expression right_shift additive_expression
    ;
```

如果*shift_expression*的操作数 `dynamic` 编译时类型，则该表达式将动态绑定（动态绑定）。在这种情况下，将 `dynamic` 表达式的编译时类型，并将使用编译时类型为 `dynamic` 的操作数的运行时类型在运行时执行下面描述的解决方法。

对于 `x << count` 或 `x >> count` 形式的操作，将应用二进制运算符重载决策（二元运算符重载决策）以选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

在声明重载移位运算符时，第一个操作数的类型必须始终为包含运算符声明的类或结构，并且第二个操作数的类型必须始终 `int`。

下面列出了预定义的移位运算符。

- 左移：

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

`<<` 运算符将 `x` 按如下所述计算的位数向左移动。

丢弃 `x` 的结果类型范围外的高序位，剩余位将向左移动，并将低序位空位位置设置为零。

- 右移：

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

`>>` 运算符将 `x` 向右移位按下面所述计算的位数。

当 `x` 的类型为 `int` 或 `long` 时，将放弃 `x` 的低序位，并向右移动剩余的位，并将高顺序空位位置设置为零（如果 `x` 为非负值，则设置为1）。

当 `x` 的类型为 `uint` 或 `ulong` 时，将丢弃 `x` 的低序位，其余位将向右移动，而高阶空位位置设置为零。

对于预定义的运算符，将计算要移位的位数，如下所示：

- 当 `x` 的类型 `int` 或 `uint` 时，移位计数由 `count` 的低序位5位提供。换句话说，移位计数由 `count & 0x1F` 计算得出。
- 当 `x` 的类型 `long` 或 `ulong` 时，移位计数由 `count` 的低序位六位给定。换句话说，移位计数由 `count & 0x3F` 计算得出。

如果生成的移位计数为零，则移位运算符只返回 `x` 的值。

移位运算从不导致溢出，并在 `checked` 和 `unchecked` 上下文中产生相同的结果。

当 `>>` 运算符的左操作数为有符号整数类型时，运算符将执行算术右移，其中操作数的最高有效位（符号位）的值将传播到高阶空位位置。当 `>>` 运算符的左操作数是无符号整数类型时，运算符将执行逻辑移位，其中高阶空位位置始终设置为零。若要执行从操作数类型推断得出的相反运算，可以使用显式强制转换。例如，如果 `x` 是 `int` 类型的变量，则运算 `unchecked((int)((uint)x >> y))` 执行 `x` 的逻辑向右移位。

关系和类型测试运算符

`==`、`!=`、`<`、`>`、`<=`、`>=`、`is` 和 `as` 运算符称为关系和类型测试运算符。

```
relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression '<=' shift_expression
    | relational_expression '>=' shift_expression
    | relational_expression 'is' type
    | relational_expression 'as' type
    ;

equality_expression
    : relational_expression
    | equality_expression '==' relational_expression
    | equality_expression '!=' relational_expression
    ;
```

Is运算符中介绍了 `is` 运算符，并在as 运算符中描述了 `as` 运算符。

`==`、`!=`、`<`、`>`、`<=` 和 `>=` 运算符是***比较运算符***。

如果比较运算符的操作数 `dynamic` 编译时类型，则该表达式将动态绑定（动态绑定）。在这种情况下，将 `dynamic` 表达式的编译时类型，并将使用编译时类型为 `dynamic` 的操作数的运行时类型在运行时执行下面描述的解决方法。

对于 `x` *op* `y` 形式的操作，其中*op*是比较运算符，将应用重载决策（二元运算符重载决策）以选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

以下各节介绍了预定义的比较运算符。所有预定义的比较运算符都返回 `bool` 类型的结果，如下表所述。

OPERATION	结果
`x == y`	`true` 如果 `x` 等于 `y``false`，则为; 否则为。
`x != y`	如果 `x` 不等于 `y``false`，则 `true`；否则为。
`x < y`	如果 `x` 小于 `y``false`，则 `true`；否则为。
`x > y`	如果 `x` 大于 `y``false`，则 `true`；否则为。
`x <= y`	如果 `x` 小于或等于 `y``false`，则 `true`；否则为。
`x >= y`	如果 `x` 大于或等于 `y``false`，则 `true`；否则为。

整数比较运算符

预定义的整数比较运算符是：

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

其中每个运算符比较两个整数操作数的数值，并返回一个 `bool` 值，该值指示是否 `true` 或 `false` 特定关系。

浮点比较运算符

预定义的浮点比较运算符是：

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

运算符根据 IEEE 754 标准的规则对操作数进行比较：

- 如果其中一个操作数为 NaN，则结果为 `true` 的所有运算符（`!=` 除外）`false`。对于任意两个操作数，`x != y` 始终产生与 `!(x == y)` 相同的结果。但是，当一个或两个操作数为 NaN 时，`<`、`>`、`<=` 和 `>=` 运算符不会生成与相对运算符逻辑求反的结果。例如，如果 `x` 和 `y` 均为 NaN，则 `false``x < y`，但 `!(x >= y)` `true`。
- 当两个操作数均为 NaN 时，运算符比较两个浮点操作数的值与排序相关

  ```
  -inf < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +inf
  ```

  其中 `min` 和 `max` 是可以用给定浮点格式表示的最小和最大的正有限值。此顺序的明显效果如下：

  - 负零和正零被视为相等。
  - 负无穷被视为小于所有其他值，但等于其他负无穷。
  - 正无穷视为大于所有其他值，但等于其他正无穷。

小数比较运算符

预定义的小数比较运算符是：

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

其中每个运算符比较两个 decimal 操作数的数值，并返回一个 `bool` 值，该值指示是否 `true` 或 `false` 特定关系。每个 decimal 比较等效于使用 `System.Decimal` 类型的对应关系或相等运算符。

布尔相等运算符

预定义的布尔相等运算符是：

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

如果 `x` 和 `y` 均 `true`，则 `true` `==` 的结果，或者 `x` 和 `y` 都 `false`。否则，结果为 `false`。

如果 `x` 和 `y` 均 `true`，则 `false` `!=` 的结果，或者 `x` 和 `y` 都 `false`。否则，结果为 `true`。当操作数的类

型为 `bool` 时，`!=` 运算符产生与 `^` 运算符相同的结果。

枚举比较运算符

每个枚举类型都隐式提供以下预定义的比较运算符：

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

计算 `x op y` 的结果，其中 `x` 和 `y` 是 `E` 基础类型为的枚举类型的表达式，`U` 是比较运算符之一，与评估 `op` 完全相同。换言之，枚举类型比较运算符只是比较两个操作数的基础整数值。

引用类型相等运算符

预定义的引用类型相等运算符是：

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

运算符返回比较两个引用是否相等或不相等的结果。

由于预定义的引用类型相等运算符接受 `object` 类型的操作数，因此它们适用于未声明适用的 `operator ==` 和 `operator !=` 成员的所有类型。相反，任何适用的用户定义的相等运算符都可以有效地隐藏预定义的引用类型相等运算符。

预定义的引用类型相等运算符需要以下项之一：

- 两个操作数都是已知为*reference_type*或文本 `null` 的类型的值。而且，显式引用转换（显式引用转换）是从任一操作数的类型到另一个操作数的类型。
- 一个操作数是类型 `T` 的值，其中 `T` 为*type_parameter*，另一个操作数为文本 `null`。此外 `T` 没有值类型约束。

除非满足这些条件之一，否则将发生绑定时错误。这些规则的明显含义是：

- 使用预定义的引用类型相等运算符比较已知在绑定时不同的两个引用是绑定时错误。例如，如果操作数的绑定时类型是 `A` 和 `B` 的两个类类型，并且 `A` 和 `B` 都不是派生自另一个，则两个操作数不可能引用同一个对象。因此，操作被视为绑定时错误。
- 预定义的引用类型相等运算符不允许比较值类型操作数。因此，除非结构类型声明自己的相等运算符，否则不能比较该结构类型的值。
- 预定义的引用类型相等运算符永远不会导致其操作数发生装箱操作。执行此类装箱操作会毫无意义，因为对新分配的装箱实例的引用必须与所有其他引用不同。
- 如果将类型参数类型 `T` 的操作数与 `null` 进行比较，并且 `T` 的运行时类型是值类型，则将 `false` 比较的结果。

下面的示例检查是否 `null` 不受约束的类型参数类型的参数。

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

即使 `T` 可以表示一个值类型，也允许 `x == null` 构造，并且仅当 `T` 为值类型时，才将结果定义为 `false`。

对于 `x == y` 或 `x != y` 形式的操作，如果存在任何适用的 `operator ==` 或 `operator !=`，运算符重载决策（二元运算符重载决策）规则将选择该运算符而不是预定义的引用类型相等运算符。但是，始终可以通过将一个或两个操作数显式强制转换为类型 `object` 来选择预定义的引用类型相等运算符。示例

```
using System;

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

生成输出

```
True
False
False
False
```

`s` 和 `t` 变量引用包含相同字符的两个不同的 `string` 实例。第一个比较输出 `True`，因为当两个操作数为类型 `string` 时，将选择预定义的字符串相等运算符（字符串相等运算符）。其余的比较 `False` 输出，因为当两个操作数为类型 `object` 时，将选择预定义的引用类型相等运算符。

请注意，上述方法对于值类型没有意义。示例

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

输出 `False` 因为强制转换会创建对两个不同的装箱 `int` 值实例的引用。

字符串相等运算符

预定义的字符串相等运算符是：

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

如果满足以下任一条件，则将两个 `string` 值视为相等：

- 这两个值都是 `null`。
- 对于在每个字符位置都具有相同长度和相同字符的字符串实例，这两个值都是非空引用。

字符串相等运算符比较字符串值而不是字符串引用。当两个单独的字符串实例包含完全相同的字符序列时，字符串的值相等，但引用不同。如引用类型相等运算符中所述，引用类型相等运算符可用于比较字符串引用而不是

字符串值。

委托相等运算符

每个委托类型都隐式提供以下预定义的比较运算符：

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

两个委托实例被视为相等，如下所示：

- 如果 `null` 的任何一个委托实例，则当且仅当两者都 `null` 时，它们才相等。
- 如果委托具有不同的运行时类型，则它们永远不相等。
- 如果两个委托实例都具有调用列表（委托声明），则当且仅当其调用列表具有相同的长度，并且一个调用列表中的每一项都等于（如下所定义）到另一个调用列表中的相应项时，这些实例相等。

以下规则控制调用列表项的相等性：

- 如果两个调用列表项引用相同的静态方法，则这些项相等。
- 如果两个调用列表项引用相同的目标对象上的相同非静态方法（由引用相等运算符定义），则这些项相等。
- 在语义上完全相同的*anonymous_method_expression*s 或*lambda_expression*s 的计算中，允许（但不要求）具有相同（可能为空）一组捕获的外部变量实例的调用列表项是相等的。

相等运算符和 null

`==` 和 `!=` 运算符允许一个操作数是可以为 null 的类型的值，另一个操作数是 `null` 文本，即使该操作没有预定义或用户定义的运算符（在 unlifted 或提升形式）。

对于某个窗体的操作

```
x == null
null == x
x != null
null != x
```

其中 `x` 是可以为 null 的类型的表达式，如果运算符重载决策（二元运算符重载决策）未能找到适用的运算符，则结果将改为从 `x` 的 `HasValue` 属性进行计算。具体而言，前两个窗体转换为 `!x.HasValue`，最后两个窗体转换为 `x.HasValue`。

Is 运算符

`is` 运算符用于动态检查对象的运行时类型与给定类型是否兼容。操作的结果 `E is T`，其中 `E` 是一个表达式，而 `T` 是一个类型，它是一个布尔值，指示是否可以通过引用转换、装箱转换或取消装箱转换将 `E` 成功转换为类型 `T`。在将类型参数替换为所有类型参数后，按如下所示计算运算：

- 如果 `E` 是匿名函数，则会发生编译时错误。
- 如果 `E` 为方法组或 `null` 文本，则如果 `E` 的类型为引用类型或可以为 null 的类型并且 `E` 的值为 null，则结果为 false。
- 否则，让 `D` 表示 `E` 的动态类型，如下所示：
  - 如果 `E` 的类型为引用类型，则 `D` 是 `E` 的实例引用的运行时类型。
  - 如果 `E` 的类型是可以为 null 的类型，则 `D` 是可以为 null 的类型的基础类型。
  - 如果 `E` 的类型是不可以为 null 的值类型，则 `D` 是 `E` 的类型。
- 操作的结果取决于 `D` 和 `T`，如下所示：
  - 如果 `T` 是引用类型，如果 `D` 和 `T` 为同一类型，则结果为 true；如果 `D` 是引用类型，并且存在从 `D` 到 `T` 的隐式引用转换，或者 `D` 是一个值类型，并且 `D` 存在从 `T` 到的装箱转换。
  - 如果 `T` 是可以为 null 的类型，则如果 `D` 是 `T` 的基础类型，则结果为 true。

- 如果 `T` 是不可以为 null 的值类型，则如果 `D` 和 `T` 为同一类型，则结果为 true。
- 否则，结果为 false。

请注意，`is` 运算符不考虑用户定义的转换。

As 运算符

`as` 运算符用于将值显式转换为给定引用类型或可以为 null 的类型。与强制转换表达式（强制转换表达式）不同，`as` 运算符永远不会引发异常。相反，如果不可能指定的转换，则将 `null` 结果值。

在 `E as T` 形式的操作中，`E` 必须是表达式，并且 `T` 必须是引用类型、已知为引用类型的类型参数或可以为 null 的类型。此外，必须至少满足以下条件之一，否则将发生编译时错误：

- 标识（标识转换）、隐式引用（隐式引用转换）、装箱（装箱转换）、显式可为 null（显式引用转换）或取消装箱（取消装箱转换）转换 `E` 到 `T` 中。
- `E` 类型或 `T` 是开放类型。
- `E` 是 `null` 文本。

如果未 `dynamic``E` 的编译时类型，则操作 `E as T` 生成的结果与

```
E is T ? (T)(E) : (T)null
```

不同的是 `E` 只计算一次。编译器可以优化 `E as T` 最多执行一次动态类型检查，而不是上述扩展隐含的两种动态类型检查。

如果 `dynamic``E` 的编译时类型，则与强制转换运算符不同，`as` 运算符不会动态绑定（动态绑定）。因此，在这种情况下，展开是：

```
E is T ? (T)(object)(E) : (T)null
```

请注意，某些转换（例如用户定义的转换）无法与 `as` 运算符一起使用，应改为使用强制转换表达式执行。

示例中

```
class X
{

    public string F(object o) {
        return o as string;        // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;             // Ok, T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;             // Error, U is unconstrained
    }
}
```

`G` 的类型参数 `T` 称为引用类型，因为它具有类约束。但 `H` 的类型参数 `U` 不是;因此，不允许在 `H` 中使用 `as` 运算符。

逻辑运算符

`&`、`^` 和 `|` 运算符称为逻辑运算符。

```
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
```

如果逻辑运算符的操作数具有编译时类型 `dynamic`，则该表达式将动态绑定(动态绑定)。在这种情况下，将 `dynamic` 表达式的编译时类型，并将使用编译时类型为 `dynamic` 的操作数的运行时类型在运行时执行下面描述的解决方法。

对于 `x op y` 形式的操作，其中 `op` 为逻辑运算符之一，将应用重载决策(二元运算符重载决策)来选择特定的运算符实现。操作数将转换为所选运算符的参数类型，结果的类型就是运算符的返回类型。

以下各节介绍了预定义的逻辑运算符。

整数逻辑运算符

预定义的整数逻辑运算符为：

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

`&` 运算符计算两个操作数的按位逻辑 `AND`，`|` 运算符计算两个操作数的按位逻辑 `OR`，而 `^` 运算符计算两个操作数的按位逻辑独占 `OR`。这些操作不能有溢出。

枚举逻辑运算符

每个枚举类型 `E` 隐式提供以下预定义的逻辑运算符：

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

计算 `x op y` 的结果，其中 `x` 和 `y` 是 `E` 基础类型为的枚举类型的表达式，`U` 是逻辑运算符之一，与评估 `op` 完全相同。换言之，枚举类型逻辑运算符只对两个操作数的基础类型执行逻辑运算。

Boolean 逻辑运算符

预定义的布尔逻辑运算符为：

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

如果 `x` 和 `y` 都为 `true`，则 `x & y` 的结果为 `true`。否则，结果为 `false`。

如果 `x` 或 `y` 为 `true`，则 `true` `x | y` 的结果。否则，结果为 `false`。

如果 `x` 为 `true`，`y` 为 `false`，或 `x` `false` `y` `true`，则会 `true` `x ^ y` 的结果。否则，结果为 `false`。当操作数为类型 `bool` 时，`^` 运算符将计算与 `!=` 运算符相同的结果。

可以为 null 的布尔逻辑运算符

可以为 null 的布尔类型 `bool?` 可以表示三个值：`true`、`false` 和 `null`，在概念上类似于用于 SQL 中的布尔表达式的三值类型。为了确保 `bool?` 操作数的 `&` 和 `|` 运算符产生的结果与 SQL 的三值逻辑一致，提供了以下预定义运算符：

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

下表列出了这些运算符为 `true`、`false` 和 `null` 的所有值组合所产生的结果。

`x`	`y`	`x & y`	`x | y`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`true`	`null`	`null`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`
`false`	`null`	`false`	`null`
`null`	`true`	`null`	`true`
`null`	`false`	`false`	`null`
`null`	`null`	`null`	`null`

条件逻辑运算符

`&&` 和 `||` 运算符称为条件逻辑运算符。它们也称为 "短路" 逻辑运算符。

```
conditional_and_expression
    : inclusive_or_expression
    | conditional_and_expression '&&' inclusive_or_expression
    ;

conditional_or_expression
    : conditional_and_expression
    | conditional_or_expression '||' conditional_and_expression
    ;
```

`&&` 和 `||` 运算符是 `&` 和 `|` 运算符的条件版本：

- 操作 `x && y` 对应于操作 `x & y`，只不过仅当 `x` 未 `false` 时才会计算 `y`。
- 操作 `x || y` 对应于操作 `x | y`，只不过仅当 `x` 未 `true` 时才会计算 `y`。

如果条件逻辑运算符的操作数具有编译时类型 `dynamic`，则该表达式将动态绑定(动态绑定)。在这种情况下，将 `dynamic` 表达式的编译时类型，并将使用编译时类型为 `dynamic` 的操作数的运行时类型在运行时执行下面描述的解决方法。

`x && y` 或 `x || y` 形式的操作通过应用重载决策(二元运算符重载决策)来处理，就像该操作是 `x & y` 还是 `x | y` 编写的。那么：

- 如果重载决策未能找到单个最佳运算符，或者重载决策选择预定义的整数逻辑运算符之一，则会发生绑定时错误。
- 否则，如果所选运算符是预定义的布尔逻辑运算符之一(布尔逻辑运算符)或可以为 null 的布尔逻辑运算符(可为 null的布尔逻辑运算符)，则按照布尔条件逻辑运算符中所述处理操作。
- 否则，所选运算符是用户定义的运算符，并且按照用户定义的条件逻辑运算符中的描述处理操作。

不能直接重载条件逻辑运算符。但是，因为条件逻辑运算符是按常规逻辑运算符计算的，所以常规逻辑运算符的重载在某些限制条件下也被视为条件逻辑运算符的重载。用户定义的条件逻辑运算符中对此进行了进一步说明。

布尔条件逻辑运算符

如果 `&&` 或 `||` 的操作数的类型为 `bool`，或者当操作数的类型没有定义适用的 `operator &` 或 `operator |`，但确实定义了到 `bool` 的隐式转换时，将按如下所示处理操作：

- 运算 `x && y` 的计算结果为 `x ? y : false`。换句话说，`x` 首先计算并转换为类型 `bool`。然后，如果 `true``x`，则计算 `y`，并将其转换为类型 `bool`，这就成为了运算结果。否则，将 `false` 操作的结果。
- 运算 `x || y` 的计算结果为 `x ? true : y`。换句话说，`x` 首先计算并转换为类型 `bool`。然后，如果 `true``x`，则将 `true` 操作的结果。否则，将计算 `y`，并将其转换为类型 `bool`，这就成为了运算结果。

用户定义的条件逻辑运算符

如果 `&&` 或 `||` 的操作数是声明适用的用户定义 `operator &` 或 `operator |` 的类型，则以下两项都必须为 true，其中 `T` 是声明所选运算符的类型：

- 所选运算符的每个参数的返回类型和类型必须是 `T`。换言之，运算符必须计算 `T` 类型的两个操作数的逻辑 `AND` 或逻辑 `OR`，并且必须返回 `T` 类型的结果。
- `T` 必须包含 `operator true` 和 `operator false` 的声明。

如果不满足上述任一要求，则会发生绑定时错误。否则，通过将用户定义的 `operator true` 或 `operator false` 与选定的用户定义运算符相结合来计算 `&&` 或 `||` 操作：

- 运算 `x && y` 的计算结果为 `T.false(x) ? x : T.&(x, y)`，其中 `T.false(x)` 是在 `T` 中声明的 `operator false` 的调用，`T.&(x, y)` 是所选 `operator &` 的调用。换而言之，首先计算 `x`，并对结果调用 `operator false`，以确定 `x` 是否确实为 false。然后，如果 `x` 肯定为 false，则操作的结果是之前为 `x` 计算的值。否则，将对 `y` 进行计算，并对之前为 `x` 计算的值调用所选 `operator &`，并为 `y` 生成操作结果的

值。

- 运算 `x || y` 的计算结果为 `T.true(x) ? x : T.|(x, y)`，其中 `T.true(x)` 是在 `T` 中声明的 `operator true` 的调用，`T.|(x,y)` 是所选 `operator|` 的调用。换言之，首先计算 `x` 并对结果调用 `operator true`，以确定 `x` 是否确实为 true。然后，如果 `x` 确实为 true，则操作的结果是之前为 `x` 计算的值。否则，将对 `y` 进行计算，并对之前为 `x` 计算的值调用所选 `operator |`，并为 `y` 生成操作结果的值。

在上述任一操作中，由 `x` 给定的表达式只计算一次，并且 `y` 指定的表达式不会进行任何计算或只计算一次。

有关实现 `operator true` 和 `operator false` 的类型的示例，请参阅数据库布尔类型。

Null 合并运算符

`??` 运算符称为 null 合并运算符。

```
null_coalescing_expression
    : conditional_or_expression
    | conditional_or_expression '??' null_coalescing_expression
    ;
```

格式 `a ?? b` 的 null 合并表达式需要 `a` 为可以为 null 的类型或引用类型。如果 `a` 为非 null，则 `a ?? b` 的结果 `a`；否则，结果为 `b`。仅当 `a` 为 null 时，操作才计算 `b`。

Null 合并运算符是右结合运算符，这意味着运算从右到左分组。例如，形式 `a ?? b ?? c` 的表达式的计算结果为 `a ?? (b ?? c)`。一般来说，形式 `E1 ?? E2 ?? ... ?? En` 的表达式返回非 null 的第一个操作数，如果所有操作数为 null，则返回 null。

`a ?? b` 表达式的类型取决于在操作数上可用的隐式转换。按照优先顺序，`a ?? b` 的类型为 `A0`、`A` 或 `B`，其中 `A` 是 `a` 的类型(前提是 `a` 具有类型)，`B` 是 `b` 的类型(前提是 `b` 有一个类型)，`A0` 是 `A` 的基础类型(如果 `A` 是可以为 null 的类型)，或者 `A` 为。具体而言，按如下所示处理 `a ?? b`：

- 如果 `A` 存在，并且不是可以为 null 的类型或引用类型，则会发生编译时错误。
- 如果 `b` 是动态表达式，则结果类型为 `dynamic`。在运行时，首先计算 `a`。如果 `a` 不为 null，则 `a` 转换为动态，这将成为结果。否则，将计算 `b`，这就是结果。
- 否则，如果 `A` 存在并且是可以为 null 的类型并且存在从 `b` 到 `A0` 的隐式转换，则结果类型为 `A0`。在运行时，首先计算 `a`。如果 `a` 不为 null，`a` 将解包为类型 `A0`，这就成为了结果。否则，将计算 `b`，并将其转换为类型 `A0`，这就成为了结果。
- 否则，如果 `A` 存在，而从 `b` 到 `A` 存在隐式转换，则结果类型为 `A`。在运行时，首先计算 `a`。如果 `a` 不为 null，则 `a` 会成为结果。否则，将计算 `b`，并将其转换为类型 `A`，这就成为了结果。
- 否则，如果 `b` 具有类型 `B` 并且存在从 `a` 到 `B` 的隐式转换，则结果类型为 `B`。在运行时，首先计算 `a`。如果 `a` 不为 null，则 `a` 将解包到类型 `A0`（如果 `A` 存在并且可为 null），并转换为类型 `B`，这就成为了结果。否则，将计算 `b` 并使其成为结果。
- 否则，`a` 和 `b` 不兼容，并发生编译时错误。

条件运算符

`?:` 运算符称为条件运算符。有时也称为三元运算符。

```
conditional_expression
    : null_coalescing_expression
    | null_coalescing_expression '?' expression ':' expression
    ;
```

`b ? x : y` 首先计算 `b` 的条件表达式。然后，如果 `true``b`，则将计算 `x`，并成为操作的结果。否则，将计算

`y`，并成为操作的结果。条件表达式从不同时计算 `x` 和 `y`。

条件运算符是右结合运算符，这意味着运算从右到左分组。例如，形式 `a ? b : c ? d : e` 的表达式的计算结果为 `a ? b : (c ? d : e)`。

`?:` 运算符的第一个操作数必须是可以隐式转换为 `bool` 的表达式，或者是实现 `operator true` 的类型的表达式。如果满足这两个要求，则会发生编译时错误。

`?:` 运算符的第二个和第三个操作数 `x` 和 `y` 控制条件表达式的类型。

- 如果 `x` 具有类型 `X` 并且 `y` 具有类型 `Y` 则
  - 如果 `X` 中存在隐式转换（隐式转换）到 `Y`，但不是从 `Y` 到 `X`，则 `Y` 是条件表达式的类型。
  - 如果 `Y` 中存在隐式转换（隐式转换）到 `X`，但不是从 `X` 到 `Y`，则 `X` 是条件表达式的类型。
  - 否则，不能确定表达式类型，并且会发生编译时错误。
- 如果 `x` 和 `y` 中只有一个具有类型，并且的 `x` 和 `y` 都可以隐式转换为该类型，则这是条件表达式的类型。
- 否则，不能确定表达式类型，并且会发生编译时错误。

格式 `b ? x : y` 的条件表达式的运行时处理包括以下步骤：

- 首先，将计算 `b`，并确定 `b` 的 `bool` 值：
  - 如果从 `b` 类型到 `bool` 存在隐式转换，则执行此隐式转换以生成 `bool` 值。
  - 否则，将调用 `b` 类型定义的 `operator true` 以生成 `bool` 值。
- 如果上面的步骤生成的 `bool` 值为 `true`，则计算 `x`，并将其转换为条件表达式的类型，这就成为条件表达式的结果。
- 否则，将计算 `y`，并将其转换为条件表达式的类型，这将成为条件表达式的结果。

匿名函数表达式

匿名函数是表示 "内联" 方法定义的表达式。匿名函数本身不具有值或类型，但可以转换为兼容的委托或表达式树类型。匿名函数转换的计算取决于转换的目标类型：如果该类型为委托类型，则转换将计算为引用匿名函数定义的方法的委托值。如果该类型为表达式树类型，则转换为表达式树，该树将方法的结构表示为对象结构。

由于历史原因，匿名函数有两种语法风格，即*lambda_expression*s 和*anonymous_method_expression*s。几乎所有情况下，*lambda_expression*比*anonymous_method_expression*的更简洁、更具表现力，这仍然是用于向后兼容的语言。

```
lambda_expression
    : anonymous_function_signature '=>' anonymous_function_body
    ;

anonymous_method_expression
    : 'delegate' explicit_anonymous_function_signature? block
    ;

anonymous_function_signature
    : explicit_anonymous_function_signature
    | implicit_anonymous_function_signature
    ;

explicit_anonymous_function_signature
    : '(' explicit_anonymous_function_parameter_list? ')'
    ;

explicit_anonymous_function_parameter_list
    : explicit_anonymous_function_parameter (',' explicit_anonymous_function_parameter)*
    ;

explicit_anonymous_function_parameter
    : anonymous_function_parameter_modifier? type identifier
    ;

anonymous_function_parameter_modifier
    : 'ref'
    | 'out'
    ;

implicit_anonymous_function_signature
    : '(' implicit_anonymous_function_parameter_list? ')'
    | implicit_anonymous_function_parameter
    ;

implicit_anonymous_function_parameter_list
    : implicit_anonymous_function_parameter (',' implicit_anonymous_function_parameter)*
    ;

implicit_anonymous_function_parameter
    : identifier
    ;

anonymous_function_body
    : expression
    | block
    ;
```

`=>` 运算符具有与赋值(`=`)相同的优先级，并且是右结合运算符。

带有 `async` 修饰符的匿名函数是一个异步函数，并遵循迭代器中所述的规则。

可以显式或隐式类型地*lambda_expression*形式的匿名函数的参数。在显式类型化参数列表中，每个参数的类型都是显式声明的。在隐式类型的参数列表中，参数的类型是从发生匿名函数的上下文中推断出来的：具体而言，当匿名函数转换为兼容的委托类型或表达式目录树类型时，该类型提供参数类型(匿名函数转换)。

在具有单个隐式类型化参数的匿名函数中，可以从参数列表中省略括号。换句话说，形式的匿名函数

```
( param ) => expr
```

可以缩写为

```
param => expr
```

*Anonymous_method_expression*格式的匿名函数的参数列表是可选的。如果给定，则必须显式键入参数。如果不是，则该匿名函数可转换为包含任何参数列表（不包含 `out` 参数）的委托。

匿名函数的*块*体是可访问的（终结点和可访问性），除非匿名函数出现在无法访问的语句中。

下面是匿名函数的一些示例：

```
x => x + 1                              // Implicitly typed, expression body
x => { return x + 1; }                  // Implicitly typed, statement body
(int x) => x + 1                        // Explicitly typed, expression body
(int x) => { return x + 1; }            // Explicitly typed, statement body
(x, y) => x * y                         // Multiple parameters
() => Console.WriteLine()               // No parameters
async (t1,t2) => await t1 + await t2    // Async
delegate (int x) { return x + 1; }      // Anonymous method expression
delegate { return 1 + 1; }              // Parameter list omitted
```

*Lambda_expression*和*anonymous_method_expression*s 的行为是相同的，但以下情况除外：

- *anonymous_method_expression*允许完全省略参数列表，则可以 convertibility 委托任何值参数列表的类型。
- *lambda_expression*允许省略和推断参数类型，而*anonymous_method_expression*则需要显式声明参数类型。
- *Lambda_expression*的主体可以是表达式或语句块，而*anonymous_method_expression*的主体必须是语句块。
- 只有*lambda_expression*s 具有转换为兼容的表达式树类型（表达式树类型）。

匿名函数签名

匿名函数的可选*anonymous_function_signature*定义匿名函数的名称和可选的形参类型。匿名函数的参数的作用域是*anonymous_function_body*。（范围）与参数列表（如果给定）一起构成了声明空间（声明）。因此，匿名函数的参数名称的编译时错误与本地变量、本地常量或参数的名称（其范围包含*anonymous_method_expression*或*lambda_expression*）相匹配。

如果匿名函数有*explicit_anonymous_function_signature*，则兼容的委托类型集和表达式树类型将被限制为具有相同顺序的相同参数类型和修饰符的类型。与方法组转换（方法组转换）相比，匿名函数参数类型的差异不受支持。如果匿名函数没有*anonymous_function_signature*，则兼容的委托类型集和表达式树类型将被限制为没有 `out` 参数的类型。

请注意，*anonymous_function_signature*不能包含特性或参数数组。尽管如此，*anonymous_function_signature*可能与其参数列表包含参数数组的委托类型兼容。

另请注意，即使兼容，转换为表达式树类型也可能仍会在编译时（表达式树类型）失败。

匿名函数体

匿名函数的主体（*表达式*或*块*）将遵循以下规则：

- 如果匿名函数包含签名，则在签名中指定的参数在正文中可用。如果匿名函数没有签名，则可以将其转换为具有参数的委托类型或表达式类型（匿名函数转换），但不能在正文中访问这些参数。
- 除了在最近的封闭匿名函数的签名（如果有）中指定的 `ref` 或 `out` 参数外，主体访问 `ref` 或 `out` 参数的编译时错误。
- 当 `this` 的类型为结构类型时，主体访问 `this` 时会发生编译时错误。无论访问是显式的（如 `this.x`）还是隐式的（如在 `x`（其中 `x` 是结构的实例成员），都是如此。此规则只是禁止此类访问，不会影响成员查找是否会导致结构的成员。
- 主体有权访问匿名函数的外部变量（外部变量）。外部变量的访问将引用在计算*lambda_expression*或*anonymous_method_expression*时处于活动状态的变量实例（匿名函数表达式的计算）。

- 如果主体包含的 `goto` 语句、`break` 语句或 `continue` 语句的目标在正文外或包含的匿名函数的主体中，则它是编译时错误。
- 正文中的 `return` 语句从最近的封闭匿名函数（而不是来自封闭函数成员）的调用返回控制权。`return` 语句中指定的表达式必须可隐式转换为最接近 *lambda_expression* 或 *anonymous_method_expression* 转换为的委托类型或表达式树类型的返回类型（匿名函数转换）。

明确指出，无论是否有任何方法可以执行匿名函数的块，而不是通过计算和调用 *lambda_expression* 或 *anonymous_method_expression*。特别是，编译器可以通过综合一个或多个命名方法或类型来选择实现匿名函数。任何此类合成元素的名称必须是保留供编译器使用的形式。

重载决策和匿名函数

自变量列表中的匿名函数参与类型推理和重载解析。请参阅类型推理和重载决策，了解确切的规则。

下面的示例演示匿名函数对重载决策的影响。

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

`ItemList<T>` 类有两个 `Sum` 方法。每个都采用 `selector` 参数，该参数从列表项中提取要求和的值。提取的值可以是 `int` 或 `double`，所得的总和也可以是 `int` 或 `double`。

例如，`Sum` 方法可用于计算按订单列出的详细信息行的总和。

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

在第一次调用 `orderDetails.Sum` 中，这两个 `Sum` 方法都适用，因为匿名函数 `d => d. UnitCount` 与 `Func<Detail,int>` 和 `Func<Detail,double>` 都兼容。但重载决策将选取第一个 `Sum` 方法，因为转换到 `Func<Detail,int>` 比转换到 `Func<Detail,double>` 好。

在第二次调用 `orderDetails.Sum` 中，只会使用第二个 `Sum` 方法，因为匿名函数 `d => d.UnitPrice * d.UnitCount` 生成类型 `double` 的值。因此，重载决策为该调用选择第二个 `Sum` 方法。

匿名函数和动态绑定

匿名函数不能是动态绑定操作的接收方、参数或操作数。

外部变量

任何本地变量、值参数或其范围包含*lambda_expression*或*anonymous_method_expression*的参数数组称为匿名函数的***外部变量***。在类的实例函数成员中，`this` 值被视为值参数，并且是函数成员内包含的任何匿名函数的外部变量。

捕获的外部变量

当某个外部变量被匿名函数引用时，该外部变量被视为已被匿名函数***捕获***。通常，局部变量的生存期仅限于执行它所关联的块或语句(局部变量)。但是，已捕获的外部变量的生存期至少会进行扩展，直到从匿名函数创建的委托或表达式树变为可进行垃圾回收的条件。

示例中

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

局部变量 `x` 由匿名函数捕获，并且 `x` 的生存期至少会进行扩展，直到从 `F` 返回的委托符合垃圾回收条件(这在程序的最末尾之前不会发生)。由于匿名函数的每个调用都在 `x` 的同一个实例上运行，因此该示例的输出为：

```
1
2
3
```

当匿名函数捕获本地变量或值参数时，本地变量或参数不再被视为固定变量(固定变量)，而是被视为可移动的变量。因此，任何采用已捕获外部变量地址的 `unsafe` 代码都必须首先使用 `fixed` 语句来修复该变量。

请注意，与 uncaptured 变量不同，捕获的局部变量可以同时公开给多个执行线程。

局部变量的实例化

当执行进入变量的作用域时，本地变量被视为被***实例化***。例如，在调用以下方法时，本地变量 `x` 将实例化并初始化三次，一次针对循环的每次迭代。

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

但是，在循环外移动 `x` 的声明会导致 `x` 的单个实例化：

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

如果未捕获，则无法准确观察局部变量的实例化频率（因为实例化的生存期是不连续的），因此每个实例化都可以只使用相同的存储位置。但是，当匿名函数捕获本地变量时，实例化的效果会变得很明显。

示例

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

生成输出：

```
1
3
5
```

但是，当 `x` 的声明移到循环外时：

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

输出是：

```
5
5
5
```

如果 for 循环声明迭代变量，则该变量本身被视为在循环外部声明。因此，如果将该示例更改为捕获迭代变量自身：

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

只捕获迭代变量的一个实例，这将生成输出：

```
3
3
3
```

匿名函数委托可以共享一些捕获的变量，但其他实例却有单独的实例。例如，如果 `F` 更改为

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

这三个委托捕获 `x` 的同一实例，但 `y` 的单独实例，输出为：

```
1 1
2 1
3 1
```

单独的匿名函数可以捕获外部变量的同一个实例。在下面的示例中：

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

这两个匿名函数 `x` 捕获本地变量的同一个实例，因此它们可以通过该变量 "进行通信"。该示例的输出为：

```
5
10
```

匿名函数表达式的计算

匿名函数 `F` 必须始终转换为委托类型 `D` 或 `E`（直接或通过执行委托创建表达式 `new D(F)`）的表达式树类型。此转换确定匿名函数的结果，如匿名函数转换中所述。

查询表达式

查询表达式为类似于关系和分层查询语言（如 SQL 和 XQuery）的查询提供了语言集成语法。

```
query_expression
    : from_clause query_body
    ;

from_clause
    : 'from' type? identifier 'in' expression
    ;

query_body
    : query_body_clauses? select_or_group_clause query_continuation?
    ;

query_body_clauses
    : query_body_clause
    | query_body_clauses query_body_clause
    ;

query_body_clause
    : from_clause
    | let_clause
    | where_clause
    | join_clause
    | join_into_clause
    | orderby_clause
    ;

let_clause
    : 'let' identifier '=' expression
    ;

where_clause
    : 'where' boolean_expression
    ;

join_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression
    ;

join_into_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression 'into' identifier
    ;

orderby_clause
    : 'orderby' orderings
    ;

orderings
    : ordering (',' ordering)*
    ;

ordering
    : expression ordering_direction?
    ;
```

```
ordering_direction
    : 'ascending'
    | 'descending'
    ;

select_or_group_clause
    : select_clause
    | group_clause
    ;

select_clause
    : 'select' expression
    ;

group_clause
    : 'group' expression 'by' expression
    ;

query_continuation
    : 'into' identifier query_body
    ;
```

查询表达式以 `from` 子句开头，以 `select` 或 `group` 子句结束。初始 `from` 子句后面可以跟零个或多个 `from`、`let`、`where`、`join` 或 `orderby` 子句。每个 `from` 子句都是引入一个*范围变量*的生成器，该变量范围是*序列*的元素。每个 `let` 子句引入一个范围变量，表示通过以前的范围变量计算得出的值。每个 `where` 子句是一个从结果中排除项的筛选器。每个 `join` 子句将源序列的指定键与另一个序列的键进行比较，从而生成匹配对。每个 `orderby` 子句会根据指定的条件对项进行重新排序。Final `select` 或 `group` 子句根据范围变量指定结果的形状。最后，`into` 子句可用于将一个查询的结果作为一个生成器在后续查询中处理。

查询表达式中的多义性

查询表达式包含许多 "上下文关键字"，即在给定上下文中具有特殊意义的标识符。具体来说，这些是 `from`、`where`、`join`、`on`、`equals`、`into`、`let`、`orderby`、`ascending`、`descending`、`select`、`group` 和 `by`。为了避免因混合使用这些标识符作为关键字或简单名称而导致的查询表达式中出现歧义，在查询表达式中的任何位置出现时，这些标识符都被视为关键字。

为此，查询表达式是以 "`from identifier`" 开头的任何表达式，后跟除 "`;`"、"`=`" 或 "`,`" 之外的任何标记。

为了在查询表达式中使用这些字词作为标识符，可以使用 "`@`"（标识符）作为前缀。

查询表达式转换

该C#语言不指定查询表达式的执行语义。相反，查询表达式被转换为符合*查询表达式模式*（查询表达式模式）的方法调用。具体而言，将查询表达式转换为名为 `Where`、`Select`、`SelectMany`、`Join`、`GroupJoin`、`OrderBy`、`OrderByDescending`、`ThenBy`、`ThenByDescending`、`GroupBy` 和 `Cast` 的方法调用。这些方法应具有特定的签名和结果类型，如查询表达式模式中所述。这些方法可以是要查询的对象的实例方法或对象外部的扩展方法，并实现查询的实际执行。

从查询表达式转换为方法调用是在执行任何类型绑定或重载决策之前发生的语法映射。转换确保语法正确，但不保证产生语义正确C#的代码。转换查询表达式后，生成的方法调用将作为常规方法调用进行处理，这可能会导致错误（例如，如果方法不存在，如果参数具有错误的类型），或者如果方法是泛型，则类型推理失败。

通过重复应用以下翻译来处理查询表达式，直到不能进一步缩减。翻译按照应用程序顺序列出：每个部分都假设之前部分中的翻译已进行了详尽的操作，一旦完成，就不会再在处理同一查询表达式时再次使用部分。

不允许在查询表达式中分配范围变量。但允许C#实现并非始终强制实施此限制，因为有时可能不会使用此处提供的语法转换方案。

某些翻译注入范围变量，其中包含由 `*` 表示的透明标识符。透明标识符中进一步讨论了透明标识符的特殊属性。

带有延续的 Select 和 groupby 子句

具有延续的查询表达式

```
from ... into x ...
```

转换为

```
from x in ( from ... ) ...
```

以下各部分中的翻译假设查询没有 `into` 的继续符。

示例

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

转换为

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

的最终转换是

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

显式范围变量类型

显式指定范围变量类型的 `from` 子句

```
from T x in e
```

转换为

```
from x in ( e ) . Cast < T > ( )
```

显式指定范围变量类型的 `join` 子句

```
join T x in e on k1 equals k2
```

转换为

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

以下部分中的翻译假设查询没有显式范围变量类型。

示例

```
from Customer c in customers
where c.City == "London"
select c
```

转换为

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

的最终转换是

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

显式范围变量类型可用于查询实现非泛型 `IEnumerable` 接口的集合，但不能用于泛型 `IEnumerable<T>` 接口。在上述示例中，如果 `customers` 的类型 `ArrayList`，就会出现这种情况。

退化查询表达式

格式为的查询表达式

```
from x in e select x
```

转换为

```
( e ) . Select ( x => x )
```

示例

```
from c in customers
select c
```

转换为

```
customers.Select(c => c)
```

退化查询表达式是完全选择源中的元素的表达式。转换的更晚阶段会删除其他翻译步骤引入的退化查询，方法是将其替换为其源。但要确保查询表达式的结果绝不是源对象本身，这一点很重要，因为这会将源的类型和标识显示到查询的客户端。因此，此步骤通过显式调用源 `Select` 来保护直接在源代码中编写的退化查询。然后由 `Select` 和其他查询运算符的实施者来确保这些方法从不返回源对象本身。

From、let、where、join 和 orderby 子句

包含第二个 `from` 子句后跟 `select` 子句的查询表达式

```
from x1 in e1
from x2 in e2
select v
```

转换为

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

一个查询表达式，该表达式的第二个 `from` 子句后跟除 `select` 子句之外的其他内容：

```
from x1 in e1
from x2 in e2
...
```

转换为

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
```

带有 `let` 子句的查询表达式

```
from x in e
let y = f
...
```

转换为

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

带有 `where` 子句的查询表达式

```
from x in e
where f
...
```

转换为

```
from x in ( e ) . Where ( x => f )
...
```

带有 `join` 子句且不带 `into` 的查询表达式，后面跟有 `select` 子句

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

转换为

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

一个查询表达式，其中的 `join` 子句没有 `into`，后面跟有 `select` 子句以外的其他内容

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

转换为

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

带有 `into` 后跟 `select` 子句的 `join` 子句的查询表达式

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

转换为

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

包含 `into` 后跟除 `select` 子句之外的 `join` 子句的查询表达式

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

转换为

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
...
```

带有 `orderby` 子句的查询表达式

```
from x in e
orderby k1 , k2 , ..., kn
...
```

转换为

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
... .
ThenBy ( x => kn )
...
```

如果 order 子句指定 `descending` 方向指示器，则改为生成 `OrderByDescending` 或 `ThenByDescending` 的调用。

以下翻译假设每个查询表达式中没有 `let`、`where`、`join` 或 `orderby` 子句，且不超过一个初始 `from` 子句。

示例

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

转换为

```
customers.
SelectMany(c => c.Orders,
     (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

示例

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

转换为

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

的最终转换**是**

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

其中 `x` **是**编译**器生成的**标识**符**，**否**则**不可**见**且**不可**访问。

示例

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

转换为

```
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

的最终转换**是**

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

其中 `x` **是**编译**器生成的**标识**符**，**否**则**不可**见**且**不可**访问。

示例

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

转换为

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

示例

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

转换为

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

的最终转换**是**

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

其中 `x` **和** `y` **是**编译**器生成的**标识**符**，**否则不可**见**且不可**访问。

示例

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

具有最终翻译

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

Select 子句

格式为**的**查询**表达式**

```
from x in e select v
```

转换为

```
( e ) . Select ( x => v )
```

除了 v 是标识符 x 时，转换只是

```
( e )
```

例如

```
from c in customers.Where(c => c.City == "London")
select c
```

只转换为

```
customers.Where(c => c.City == "London")
```

Groupby 子句

格式为的查询表达式

```
from x in e group v by k
```

转换为

```
( e ) . GroupBy ( x => k , x => v )
```

除非 v 是标识符 x，否则转换为

```
( e ) . GroupBy ( x => k )
```

示例

```
from c in customers
group c.Name by c.Country
```

转换为

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

透明标识符

某些翻译注入范围变量，其中包含由 `*` 表示的***透明标识符***。透明标识符不是正确的语言功能;它们仅作为查询表达式转换过程中的一个中间步骤存在。

当查询转换注入透明标识符时，进一步的转换步骤会将透明标识符传播到匿名函数和匿名对象初始值设定项。在这些上下文中，透明标识符具有以下行为：

- 当透明标识符作为匿名函数中的参数出现时，关联的匿名类型的成员将自动出现在匿名函数主体的范围内。
- 当具有透明标识符的成员位于范围内时，该成员的成员也会在范围内。
- 当透明标识符作为匿名对象初始值设定项中的成员声明符出现时，它会引入一个具有透明标识符的成员。

- 在上面所述的翻译步骤中，透明标识符始终与匿名类型一起引入，目的是将多个范围变量捕获为单个对象的成员。允许实现C#使用与匿名类型不同的机制将多个范围变量组合在一起。以下翻译示例假设使用匿名类型，并演示如何将透明标识符转换为。

示例

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

转换为

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

这会进一步转换为

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

当清除透明标识符时，它等效于

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

其中 `x` 是编译器生成的标识符，否则不可见且不可访问。

示例

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

转换为

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

这会进一步减小到

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

的最终转换是

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
    (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
    (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

其中 `x`、`y` 和 `z` 是编译器生成的标识符，否则不可见且不可访问。

查询表达式模式

*查询表达式模式*建立了一种方法，这些方法可实现类型以支持查询表达式。由于查询表达式是通过语法映射转换为方法调用的，因此在实现查询表达式模式的方式上，类型具有相当大的灵活性。例如，可以将模式的方法作为实例方法或扩展方法实现，因为这两个方法具有相同的调用语法，并且方法可以请求委托或表达式树，因为匿名函数可同时转换为两者。

下面显示了支持查询表达式模式的泛型类型 `C<T>` 的推荐形状。使用泛型类型来说明参数和结果类型之间的正确关系，但也可以实现非泛型类型的模式。

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

上述方法使用 `Func<T1,R>` 和 `Func<T1,T2,R>` 的泛型委托类型，但它们同样可以使用参数和结果类型中具有相同关系的其他委托或表达式树类型。

请注意 `C<T>` 和 `O<T>` 之间的推荐关系，这可确保 `ThenBy` 和 `ThenByDescending` 方法仅对 `OrderBy` 或 `OrderByDescending` 的结果可用。另请注意 `GroupBy` 结果的建议形状（序列序列），其中每个内部序列都有附加的 `Key` 属性。

`System.Linq` 命名空间为实现 `System.Collections.Generic.IEnumerable<T>` 接口的任何类型提供查询运算符模式的实现。

赋值运算符

赋值运算符将新值分配给变量、属性、事件或索引器元素。

```
assignment
    : unary_expression assignment_operator expression
    ;

assignment_operator
    : '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | right_shift_assignment
    ;
```

赋值运算的左操作数必须是分类为变量、属性访问、索引器访问或事件访问的表达式。

`=` 运算符称为*简单赋值运算符*。它将右操作数的值分配给左操作数给定的变量、属性或索引器元素。简单赋值运算符的左操作数不能是事件访问(如类似于字段的事件中所述)。简单赋值运算符在简单赋值中进行了介绍。

除 `=` 运算符以外的赋值运算符称为*复合赋值运算符*。这些运算符对两个操作数执行指定的运算，然后将结果值分配给左操作数给定的变量、属性或索引器元素。复合赋值运算符在复合赋值中介绍。

使用事件访问表达式作为左操作数的 `+=` 和 `-=` 运算符称为*事件赋值运算符*。其他赋值运算符对于作为左操作数的事件访问无效。事件赋值运算符在事件分配中介绍。

赋值运算符是右结合运算符，这意味着运算从右到左分组。例如，形式 `a = b = c` 的表达式的计算结果为 `a = (b = c)`。

简单赋值

`=` 运算符称为简单赋值运算符。

如果简单赋值的左操作数的形式为 `E.P` 或 `E[Ei]` `E` 具有编译时类型 `dynamic`，则分配是动态绑定的(动态绑定)。在这种情况下，赋值表达式的编译时类型是 `dynamic` 的，并且基于 `E` 的运行时类型，将在运行时进行以下描述的解决方法。

在简单赋值中，右操作数必须是可隐式转换为左操作数类型的表达式。操作将右操作数的值分配给左操作数给定的变量、属性或索引器元素。

简单赋值表达式的结果是赋给左操作数的值。结果与左操作数的类型相同，并且始终归类为值。

如果左操作数是属性或索引器访问，则属性或索引器必须具有 `set` 访问器。如果不是这种情况，则会发生绑定时错误。

对形式 `x = y` 的简单赋值处理的运行时处理包括以下步骤：

- 如果 `x` 归类为变量：
  - 计算 `x` 以产生变量。
  - 计算 `y`，并在需要时通过隐式转换转换为 `x` 的类型(隐式转换)。
  - 如果 `x` 给定的变量是*reference_type*的数组元素，则执行运行时检查，以确保为 `y` 计算的值与 `x` 为元素的数组实例兼容。如果 `y` `null`，或者存在从 `y` 引用的实例的实际类型到包含 `x` 的数组实例的实际元素类型的隐式引用转换(隐式引用转换)，则检查成功。否则，将会引发 `System.ArrayTypeMismatchException`。
  - `y` 的计算和转换生成的值将存储到 `x` 计算得出的位置。
- 如果 `x` 归类为属性或索引器访问：

- 实例表达式（如果 `x` 不 `static`）和参数列表（如果 `x` 是索引器访问）与 `x` 关联，并在后续的 `set` 访问器调用中使用结果。
- 计算 `y`，并在需要时通过隐式转换转换为 `x` 的类型（隐式转换）。
- 调用 `x` 的 `set` 访问器时，会将计算的值 `y` 为其 `value` 参数。

如果存在从 `B` 到 `A` 的隐式引用转换，数组协方差规则（数组协方差）允许 `A[]` 为数组类型 `B[]` 的实例的引用。由于这些规则，对 *reference_type* 的数组元素赋值需要运行时检查，以确保所赋的值与数组实例兼容。示例中

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;               // Ok
oa[1] = "Hello";            // Ok
oa[2] = new ArrayList();    // ArrayTypeMismatchException
```

最后一个分配导致引发 `System.ArrayTypeMismatchException`，因为 `ArrayList` 的实例无法存储在 `string[]` 的元素中。

如果在 *struct_type* 中声明的属性或索引器是赋值目标，则必须将与属性或索引器访问关联的实例表达式归类为变量。如果实例表达式归类为值，则会发生绑定时错误。由于成员访问，相同的规则也适用于字段。

给定以下声明：

```
struct Point
{
    int x, y;

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

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

struct Rectangle
{
    Point a, b;

    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }

    public Point A {
        get { return a; }
        set { a = value; }
    }

    public Point B {
        get { return b; }
        set { b = value; }
    }
}
```

示例中

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

允许分配到 `p.X`、`p.Y`、`r.A` 和 `r.B`，因为 `p` 和 `r` 都是变量。但在示例中

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

分配均无效，因为 `r.A` 和 `r.B` 不是变量。

复合赋值

如果复合赋值的左操作数的形式为 `E.P` 或 `E[Ei]` `E` 具有编译时类型 `dynamic`，则分配是动态绑定的(动态绑定)。在这种情况下，赋值表达式的编译时类型是 `dynamic` 的，并且基于 `E` 的运行时类型，将在运行时进行以下描述的解决方法。

通过应用二元运算符重载决策(二元运算符重载决策)来处理窗体 `x op= y` 的操作，就像将操作写入 `x op y` 一样。那么：

- 如果所选运算符的返回类型可隐式转换为 `x` 的类型，则该运算将计算为 `x = x op y`，只不过 `x` 只计算一次。
- 否则，如果所选运算符是预定义的运算符，则如果所选运算符的返回类型可显式转换为 `x` 的类型，并且如果 `y` 可隐式转换为 `x` 类型或运算符为移位运算符，则该运算将计算为 `x = (T)(x op y)`，其中 `T` 是 `x` 的类型，但 `x` 只计算一次。
- 否则，复合分配无效，并发生绑定时错误。

术语 "仅计算一次" 表示在 `x op y` 的计算中，将临时保存 `x` 的所有构成表达式的结果，并在对 `x` 执行赋值时重用。例如，在赋值 `A()[B()] += C()` 中，其中 `A` 是返回 `int[]` 的方法，并且 `B` 和 `C` 是返回 `int` 的方法，这些方法只调用一次，并按顺序 `A``B`、`C`。

当复合赋值的左操作数是属性访问或索引器访问时，属性或索引器必须同时具有 `get` 访问器和 `set` 访问器。如果不是这种情况，则会发生绑定时错误。

上面的第二个规则允许 `x op= y` 在某些上下文中计算为 `x = (T)(x op y)`。存在该规则，以便当左操作数的类型为 `sbyte`、`byte`、`short`、`ushort` 或 `char` 时，预定义运算符可用作复合运算符。即使两个参数都属于这些类型之一，预定义运算符也会生成 `int` 类型的结果，如二进制数值升级中所述。因此，在没有强制转换的情况下，不能将结果赋给左操作数。

预定义运算符规则的直观效果只是允许 `x op y` 和 `x = y` 都允许 `x op= y`。示例中

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;             // Ok
b += 1000;          // Error, b = 1000 not permitted
b += i;             // Error, b = i not permitted
b += (byte)i;       // Ok

ch += 1;            // Error, ch = 1 not permitted
ch += (char)1;      // Ok
```

每个错误的直观原因在于，相应的简单分配也会导致错误。

这也意味着复合赋值操作支持提升的操作。示例中

```
int? i = 0;
i += 1;             // Ok
```

使用提升运算符 `+(int?,int?)` 。

事件分配

如果 `+=` 或 `-=` 运算符的左操作数分类为事件访问，则表达式的计算方式如下：

- 计算事件访问的实例表达式（如果有）。
- 计算 `+=` 或 `-=` 运算符的右操作数，并在需要时通过隐式转换转换为左操作数的类型（隐式转换）。
- 调用事件的事件访问器，其中包含右操作数（在计算后，如有必要）转换后的参数列表。如果运算符是 `+=` 的，则调用 `add` 访问器;如果运算符是 `-=` 的，则调用 `remove` 访问器。

事件赋值表达式不生成值。因此，事件分配表达式仅在*statement_expression*（expression 语句）的上下文中有效。

表达式

*表达式*可以是*non_assignment_expression*或*赋值*。

```
expression
    : non_assignment_expression
    | assignment
    ;

non_assignment_expression
    : conditional_expression
    | lambda_expression
    | query_expression
    ;
```

常量表达式

*Constant_expression*是可以在编译时完全计算的表达式。

```
constant_expression
    : expression
    ;
```

常数表达式必须是 `null` 文本或具有以下类型之一的值： `sbyte` 、 `byte` 、 `short` 、 `ushort` 、 `int` 、 `uint` 、 `long` 、

`ulong`、`char`、`float`、`double`、`decimal`、`bool`、`object`、`string`或任何枚举类型。常量表达式中仅允许使用以下构造：

- 文本（包括 `null` 文本）。
- 对类类型和结构类型 `const` 成员的引用。
- 对枚举类型的成员的引用。
- 对 `const` 参数或局部变量的引用
- 带括号的子表达式，它们本身就是常量表达式。
- 如果目标类型是上面列出的类型之一，则强制转换表达式。
- `checked` 和 `unchecked` 表达式
- 默认值表达式
- Nameof 表达式
- 预定义的 `+`、`-`、`!` 和 `~` 一元运算符。
- 预定义的 `+`、`-`、`*`、`/`、`%`、`<<`、`>>`、`&`、`|`、`^`、`&&`、`||`、`==`、`!=`、`<`、`>`、`<=` 和 `>=` 二进制运算符，前提是每个操作数都是上面列出的类型。
- `?:` 条件运算符。

常数表达式中允许以下转换：

- 标识转换
- 数值转换
- 枚举转换
- 常数表达式转换
- 隐式和显式引用转换，前提是转换的源是计算结果为 null 值的常量表达式。

不允许在常数表达式中使用其他转换，包括非 null 值的装箱、取消装箱和隐式引用转换。例如：

```
class C {
    const object i = 5;         // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

i 的初始化是错误的，因为需要装箱转换。Str 的初始化是错误的，因为需要从非 null 值进行隐式引用转换。

只要表达式满足上面列出的要求，就会在编译时计算表达式。即使表达式是包含非常量构造的更大的表达式的子表达式，也是如此。

常数表达式的编译时计算使用与非常量表达式的运行时计算相同的规则，只不过运行时计算会引发异常，编译时计算会导致发生编译时错误。

除非将常量表达式显式置于 `unchecked` 的上下文中，否则在表达式的编译时计算过程中发生的溢出将始终导致编译时错误（常量表达式）。

常数表达式出现在下面列出的上下文中。在这些上下文中，如果表达式在编译时无法完全计算，则会发生编译时错误。

- 常量声明（常量）。
- 枚举成员声明（枚举成员）。
- 形参表的默认参数（方法参数）
- `switch` 语句 `case` 标签（switch 语句）。
- `goto case` 语句（goto 语句）。
- 数组创建表达式中的维度长度（数组创建表达式），其中包含初始值设定项。
- 特性（特性）。

如果常量表达式的值在目标类型的范围内，则隐式常量表达式转换（隐式常数表达式转换）允许 `int` 类型的常量表达式转换为 `sbyte`、`byte`、`short`、`ushort`、`uint` 或 `ulong`。

Boolean 表达式

*Boolean_expression*是生成 `bool` 类型的结果的表达式;直接或通过在某些上下文中 `operator true` 应用程序，如以下所示。

```
boolean_expression
    : expression
    ;
```

If_statement（if 语句）、*while_statement*（while 语句）、*do_statement*（do 语句）或*for_statement*（for 语句）的控制条件表达式为*boolean_expression*。`?:` 运算符（条件运算符）的控制条件表达式遵循与*boolean_expression*相同的规则，但由于运算符优先级的原因，将归类为*conditional_or_expression*。

需要*boolean_expression* `E` 才能生成类型 `bool` 的值，如下所示：

- 如果 `E` 可隐式转换为 `bool` 则在运行时将应用隐式转换。
- 否则，一元运算符重载决策（一元运算符重载决策）用于查找 `E` 上运算符 `true` 的唯一最佳实现，并且该实现在运行时应用。
- 如果找不到这样的运算符，则发生绑定时错误。

数据库布尔类型中 `DBBool` 结构类型提供了实现 `operator true` 和 `operator false` 的类型的示例。

语句

2020/11/2 • • Edit Online

C#提供各种语句。对于在 C 和C++中进行了编程的开发人员，这些语句中的大部分将很熟悉。

```
statement
    : labeled_statement
    | declaration_statement
    | embedded_statement
    ;

embedded_statement
    : block
    | empty_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | try_statement
    | checked_statement
    | unchecked_statement
    | lock_statement
    | using_statement
    | yield_statement
    | embedded_statement_unsafe
    ;
```

*Embedded_statement*非终止符用于在其他语句中显示的语句。使用*embedded_statement*而不是*语句*不包括在这些上下文中声明语句和标记语句的使用。示例

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

导致编译时错误，因为 `if` 语句需要*embedded_statement*而不是*语句*用于 if 分支。如果允许此代码，则将声明变量 `i`，但绝不能使用它。但请注意，通过将 `i` 的声明放置在块中，该示例是有效的。

终结点和可访问性

每个语句都有一个*终结点*。简而言之，语句的结束点是紧跟在语句之后的位置。复合语句的执行规则（包含嵌入语句的语句）指定在控件到达嵌入语句的终结点时所采取的操作。例如，当控件到达块中语句的终点时，控制将转移到块中的下一条语句。

如果语句可以通过执行到达，则表明该语句是可*访问*的。相反，如果不可能执行语句，则认为该语句是无法*访问*的。

示例中

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
    Label:
    Console.WriteLine("reachable");
}
```

无法访问 `Console.WriteLine` 的第二个调用，因为不可能执行该语句。

如果编译器确定无法访问某个语句，则会报告警告。它特别不是错误，无法访问语句。

若要确定特定的语句或终结点是否可访问，编译器将根据为每个语句定义的可访问性规则执行流分析。流分析考虑了控制语句行为的常量表达式（常量表达式）的值，但不考虑非常量表达式的可能值。换句话说，出于控制流分析的目的，给定类型的非常量表达式被视为具有该类型的任何可能值。

示例中

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

`if` 语句的布尔表达式是常量表达式，因为 `==` 运算符的两个操作数都是常量。在编译时计算常量表达式，并 `false` 生成值时，`Console.WriteLine` 调用被视为不可访问。但是，如果 `i` 更改为本地变量

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

`Console.WriteLine` 调用被视为可访问，但实际上，它将永远不会执行。

函数成员的*块*始终被认为是可访问的。通过依次计算块中每个语句的可访问性规则，可以确定任何给定语句的可访问性。

示例中

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

确定第二个 `Console.WriteLine` 的可访问性，如下所示：

- 第一个 `Console.WriteLine` expression 语句是可访问的，因为 `F` 方法的块是可访问的。
- 第一个 `Console.WriteLine` expression 语句的结束点是可访问的，因为该语句是可访问的。
- 由于第一个 `Console.WriteLine` 表达式语句的结束点是可访问的，因此可以访问 `if` 语句。
- 第二个 `Console.WriteLine` expression 语句是可访问的，因为 `if` 语句的布尔表达式不具有 `false` 的常量值。

在以下两种情况下，可能会发生编译时错误，导致语句的终结点可到达：

- 由于 `switch` 语句不允许 switch 节 "贯穿" 到下一个开关部分，因此在开关部分的语句列表的终结点可访问时，会发生编译时错误。如果发生此错误，则通常指示缺少 `break` 语句。
- 对于计算要访问的值的函数成员块，它是一个编译时错误。如果发生此错误，则通常指示缺少 `return` 语句。

Blocks

使用*代码块*，可以在允许编写一个语句的上下文中编写多个语句。

```
block
    : '{' statement_list? '}'
    ;
```

*块*由括在大括号中的可选*statement_list*（语句列表）组成。如果省略了语句列表，则称块为空。

块可以包含声明语句（声明语句）。块中声明的局部变量或常量的范围为块。

块的执行方式如下：

- 如果块为空，控制将被传输到块的终结点。
- 如果块不为空，则将控制转移到语句列表。当和如果控件到达语句列表的终点时，控制将被传输到块的终结点。

如果块本身是可访问的，则块的语句列表是可访问的。

如果块为空或语句列表的终结点是可访问的，则块的终结点是可访问的。

包含一个或多个 `yield` 语句（yield 语句）的*块*称为迭代器块。迭代器块用于将函数成员作为迭代器（迭代器）实现。迭代器块适用于一些附加限制：

- `return` 语句出现在迭代器块中（但允许 `yield return` 语句），这是编译时错误。
- 迭代器块包含不安全的上下文（不安全上下文）是编译时错误。迭代器块始终定义安全上下文，即使其声明嵌套在不安全的上下文中也是如此。

语句列表

语句列表包含一个或多个按顺序编写的语句。语句列表在*块*s（块）中出现，在*switch_block*s 中（switch 语句）。

```
statement_list
    : statement+
    ;
```

语句列表通过将控制转移到第一条语句来执行。当和如果控件到达语句的结束点时，控制将转移到下一个语句。当和如果控件到达最后一个语句的终点时，控件将被传输到语句列表的终结点。

如果以下至少一个条件为 true，则语句列表中的语句是可访问的：

- 语句是第一条语句，语句列表本身是可访问的。
- 可以访问前面语句的终点。
- 语句是标记的语句，并且标签由可访问的 `goto` 语句引用。

如果列表中最后一个语句的结束点是可访问的，则该语句列表的终结点是可访问的。

空语句

*Empty_statement*不执行任何操作。

```
empty_statement
    : ';'
    ;
```

在需要语句的上下文中没有要执行的操作时，将使用空语句。

空语句的执行只是将控制转移到语句的终结点。因此，如果可以访问空语句，则可以访问空语句的结束点。

使用 null 体写入 `while` 语句时，可以使用空语句：

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

此外，空语句还可用于在块的关闭 "`}`" 之前声明标签：

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

带标签的语句

*Labeled_statement*允许语句以标签为前缀。标记语句允许出现在块中，但不允许作为嵌入语句使用。

```
labeled_statement
    : identifier ':' statement
    ;
```

标记语句使用*标识符*给定的名称声明标签。标签的作用域是在其中声明标签的整个块，包括任何嵌套块。对于具有相同名称的两个同名标签，它是编译时错误。

可以从标签范围内的 `goto` 语句（goto 语句）引用标签。这意味着 `goto` 语句可以将控制转移到块中，而不是块中的块。

标签具有自己的声明空间，不会干扰其他标识符。示例

```
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

有效，并使用名称 `x` 同时作为参数和标签。

标记语句的执行与标签后面的语句的执行完全对应。

除了正常控制流提供的可访问性外，如果标签由可访问的 `goto` 语句引用，则可以访问标记的语句。（异常：如果 `goto` 语句位于包含 `finally` 块的 `try` 中，且标记语句位于 `try` 外，并且无法访问 `finally` 块的终结点，则无法从该 `goto` 语句访问标记的语句。）

声明语句

声明局部变量或常量的*declaration_statement* 。声明语句在块中是允许的，但不允许作为嵌入语句。

```
declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    ;
```

局部变量声明

*Local_variable_declaration*声明一个或多个局部变量。

```
local_variable_declaration
    : local_variable_type local_variable_declarators
    ;

local_variable_type
    : type
    | 'var'
    ;

local_variable_declarators
    : local_variable_declarator
    | local_variable_declarators ',' local_variable_declarator
    ;

local_variable_declarator
    : identifier
    | identifier '=' local_variable_initializer
    ;

local_variable_initializer
    : expression
    | array_initializer
    | local_variable_initializer_unsafe
    ;
```

*Local_variable_declaration*的*local_variable_type*直接指定声明引入的变量类型，或者使用标识符 `var` 指示应基于初始值设定项推断类型。该类型后跟一个*local_variable_declarator*s 列表，其中每个都引入一个新变量。*Local_variable_declarator*包含命名变量的*标识符*，可选择后跟 "`=`" 标记和提供变量初始值的*local_variable_initializer* 。

在局部变量声明的上下文中，标识符 var 用作上下文关键字（关键字）。如果将*local_variable_type*指定为 `var` 并且未在范围中指定名为 `var` 的类型，则声明为***隐式类型化局部变量声明***，其类型是从关联的初始值设定项表达式的类型推断而来的。隐式类型的局部变量声明受到以下限制：

- *Local_variable_declaration*不能包含多个*local_variable_declarator*。
- *Local_variable_declarator*必须包含*local_variable_initializer*。
- *Local_variable_initializer*必须是*表达式*。
- 初始值设定项*表达式*必须具有编译时类型。
- 初始值设定项*表达式*不能引用声明的变量本身

下面是错误的隐式类型化局部变量声明的示例：

```
var x;              // Error, no initializer to infer type from
var y = {1, 2, 3};  // Error, array initializer not permitted
var z = null;       // Error, null does not have a type
var u = x => x + 1; // Error, anonymous functions do not have a type
var v = v++;        // Error, initializer cannot refer to variable itself
```

局部变量的值是使用*simple_name*（简单名称）在表达式中获取的，使用*赋值*（赋值运算符）修改本地变量的值。局部变量在获取其值的每个位置都必须明确赋值（明确赋值）。

在*local_variable_declaration*中声明的局部变量的作用域是在其中进行声明的块。在本地变量*local_variable_declarator*之前的文本位置引用局部变量是错误的。在局部变量的作用域内，使用相同的名称声明另一个局部变量或常量时，会发生编译时错误。

声明多个变量的局部变量声明等效于多个具有相同类型的单个变量的声明。此外，局部变量声明中的变量初始值设定项完全对应于紧接在声明后插入的赋值语句。

示例

```
void F() {
    int x = 1, y, z = x * 2;
}
```

完全对应于

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

在隐式类型的局部变量声明中，声明的局部变量的类型将被视为与用于初始化变量的表达式的类型相同。例如：

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

上面的隐式类型局部变量声明完全等效于以下显式类型化声明：

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

局部常量声明

一个*local_constant_declaration*声明一个或多个本地常量。

```
local_constant_declaration
    : 'const' type constant_declarators
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

*Local_constant_declaration*的*类型*指定声明引入的常量的类型。该类型后跟一个*constant_declarator*s 列表，其中每个都引入一个新常量。*Constant_declarator*包含一个*标识符*，该标识符对常量进行命名，后跟一个 "`=`" 标记，后跟一个提供常量值的*constant_expression*（常数表达式）。

局部常量声明的*类型*和*constant_expression*必须遵循与常量成员声明（常量）相同的规则。

局部变量的值是使用*simple_name*（简单名称）在表达式中获取的。

局部常数的作用域是在其中进行声明的块。在*constant_declarator*之前的文本位置引用本地常量是错误的。在局部常数的范围内，使用相同的名称声明另一个局部变量或常量时，会发生编译时错误。

声明多个常量的局部常量声明等效于多个具有相同类型的单个常量的声明。

表达式语句

*Expression_statement*计算给定表达式的值。由表达式计算的值（如果有）将被丢弃。

```
expression_statement
    : statement_expression ';'
    ;

statement_expression
    : invocation_expression
    | null_conditional_invocation_expression
    | object_creation_expression
    | assignment
    | post_increment_expression
    | post_decrement_expression
    | pre_increment_expression
    | pre_decrement_expression
    | await_expression
    ;
```

并非所有表达式都允许作为语句。具体而言，不允许使用只计算值（将被丢弃）的表达式（如 `x + y` 和 `x == 1`）作为语句。

执行*expression_statement*将计算包含的表达式，然后将控制转移到*expression_statement*的终结点。如果*expression_statement*可访问，则可到达*expression_statement*的终结点。

选择语句

选择语句根据某个表达式的值为执行选择多个可能的语句之一。

```
selection_statement
    : if_statement
    | switch_statement
    ;
```

If 语句

`if` 语句基于布尔表达式的值来选择要执行的语句。

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

`else` 部分与语法允许的上一个词法上最近的 `if` 相关联。因此，形式的 `if` 语句

```
if (x) if (y) F(); else G();
```

等效于

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

执行 `if` 语句，如下所示：

- 计算*boolean_expression*（布尔表达式）。
- 如果布尔表达式产生 `true`，则将控制转移到第一个嵌入语句。当和如果控件到达该语句的终点时，控制将被传输到 `if` 语句的终点。
- 如果布尔表达式产生 `false` 并且 `else` 部分存在，则将控制权转移到第二个嵌入语句。当和如果控件到达该语句的终点时，控制将被传输到 `if` 语句的终点。
- 如果布尔表达式产生 `false` 并且 `else` 部分不存在，则将控制权转移到 `if` 语句的终点。

如果 `if` 语句可访问且布尔表达式不具有 `false` 的常量值，则可以访问 `if` 语句的第一个嵌入语句。

如果 `if` 语句的第二个嵌入语句（如果存在）可访问，如果 `if` 语句可访问且布尔表达式不具有 `true` 的常量值。

如果可以访问一个或多个嵌入语句的结束点，则可到达 `if` 语句的终点。此外，如果 `if` 语句是可访问的，并且布尔表达式没有 `true` 的常量值，则可访问没有 `else` 部分的 `if` 语句的结束点。

Switch 语句

Switch 语句为执行选择一个语句列表，其中包含与 switch 表达式的值相对应的关联 switch 标签。

```
switch_statement
    : 'switch' '(' expression ')' switch_block
    ;

switch_block
    : '{' switch_section* '}'
    ;

switch_section
    : switch_label+ statement_list
    ;

switch_label
    : 'case' constant_expression ':'
    | 'default' ':'
    ;
```

*Switch_statement*包含关键字 `switch`，后跟带括号的表达式（称为 switch 表达式），后跟一个*switch_block*。*Switch_block*由零个或多个*switch_section*组成，括在大括号中。每个*switch_section*都包含一个或多个*switch_label*，后跟一个*statement_list*（语句列表）。

`switch` 语句的***管理类型***由 switch 表达式建立。

- 如果 switch 表达式的类型为 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`bool`、`char`、`string` 或*enum_type*，或者它是对应于其中一个类型的可以为 null 的类型，则这是 `switch` 语句的管理类型。
- 否则，必须有一个用户定义的隐式转换（用户定义的转换）从 switch 表达式的类型到以下可能的管辖类型之一：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`string` 或，这是与这些类型之一对应的可以为 null 的类型。
- 否则，如果不存在这样的隐式转换，或者存在多个这样的隐式转换，则会发生编译时错误。

每个 `case` 标签的常量表达式必须表示一个可隐式转换为 `switch` 语句的管理类型的值。如果同一 `switch` 语句中的两个或多个 `case` 标签指定相同的常量值，则会发生编译时错误。

Switch 语句中最多只能有一个 `default` 标签。

执行 `switch` 语句，如下所示：

- 将计算 switch 表达式并将其转换为管理类型。
- 如果在同一 `switch` 语句的 `case` 标签中指定的其中一个常量等于 switch 表达式的值，则会将控制转移到匹配的 `case` 标签之后的语句列表中。
- 如果在同一 `switch` 语句的 `case` 标签中指定的常量均不等于 switch 表达式的值，并且如果存在 `default` 标签，则会将控制转移到后面的 `default` 标签后面的语句列表中。
- 如果在同一 `switch` 语句的 `case` 标签中指定的常量均不等于 switch 表达式的值，并且如果不存在 `default` 标签，则会将控制转移到 `switch` 语句的终点。

如果开关部分的语句列表的结束点是可访问的，则会发生编译时错误。这称为 "不贯穿" 规则。示例

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}
```

有效，因为没有开关部分具有可访问的终结点。与 C 和C++不同，switch 节的执行不允许 "贯穿" 到下一个开关部分，示例

```
switch (i) {
case 0:
    CaseZero();
case 1:
    CaseZeroOrOne();
default:
    CaseAny();
}
```

导致编译时错误。当执行 switch 节后，若要执行另一个 switch 节，必须使用显式 `goto case` 或 `goto default` 语句：

```
switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}
```

*Switch_section*允许使用多个标签。示例

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}
```

有效。该示例不违反 "no 贯穿" 规则，因为标签 `case 2:` 和 `default:` 属于同一*switch_section*。

"不贯穿" 规则可防止 C 中发生的常见错误类，以及C++在意外省略 `break` 语句时。此外，由于此规则，可以任意重新排列 `switch` 语句的 switch 部分，而不会影响语句的行为。例如，上面的 `switch` 语句的各个部分可以反向，而不会影响语句的行为：

```
switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}
```

开关部分的语句列表通常以 `break`、`goto case` 或 `goto default` 语句结束，但允许不能使用呈现语句列表的终结点的任何构造。例如，已知由布尔表达式 `true` 控制的 `while` 语句永远不会到达其终结点。同样，`throw` 或 `return` 语句始终会将控制转移到其他位置，而永远不会到达终结点。因此，下面的示例是有效的：

```
switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```

`switch` 语句的管理类型可能是 `string` 类型。例如：

```
void DoCommand(string command) {
    switch (command.ToLower()) {
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidCommand(command);
        break;
    }
}
```

与字符串相等运算符（string 相等运算符）一样，`switch` 语句区分大小写，并且仅当 switch 表达式字符串与 `case` 标签常量完全匹配时，才会执行给定的 switch 节。

当 `string``switch` 语句的管理类型时，允许将值 `null` 为 case 标签常量。

*Switch_block*的*statement_list*可能包含声明语句（声明语句）。Switch 块中声明的局部变量或常量的范围为 switch 块。

如果 `switch` 语句可访问，且至少满足以下条件之一，则给定 switch 节的语句列表是可访问的：

- Switch 表达式是一个非常量的值。
- Switch 表达式是一个与开关部分中的 `case` 标签匹配的常量值。
- Switch 表达式是不与任何 `case` 标签匹配的常量值，并且开关部分包含 `default` 标签。
- 开关部分的开关标签由可访问的 `goto case` 或 `goto default` 语句引用。

如果以下至少一个条件为 true，则可到达 `switch` 语句的终结点：

- `switch` 语句包含可访问的 `break` 语句，该语句可退出 `switch` 语句。
- `switch` 语句是可访问的，则 switch 表达式为非常量值，并且不存在任何 `default` 标签。
- `switch` 语句是可访问的，则 switch 表达式是一个与任何 `case` 标签都不匹配的常量值，并且不存在 `default` 标签。

迭代语句

迭代语句重复执行嵌入语句。

```
iteration_statement
    : while_statement
    | do_statement
    | for_statement
    | foreach_statement
    ;
```

While 语句

`while` 语句有条件地执行一个嵌入语句零次或多次。

```
while_statement
    : 'while' '(' boolean_expression ')' embedded_statement
    ;
```

执行 `while` 语句，如下所示：

- 计算*boolean_expression*（布尔表达式）。
- 如果布尔表达式产生 `true`，则将控制转移到嵌入语句。当和如果控件到达嵌入语句的结束点（可能是执行 `continue` 语句）时，控制将转移到 `while` 语句的开头。
- 如果布尔表达式产生 `false`，则将控制转移到 `while` 语句的终点。

在 `while` 语句的嵌入语句中，`break` 语句（break 语句）可用于将控制转移到 `while` 语句的终结点（从而结束嵌入语句的迭代），并且 `continue` 语句（continue 语句）可用于将控制转移到嵌入语句的终结点（从而执行 `while` 语句的另一次迭代）。

如果 `while` 语句可访问且布尔表达式不具有 `false` 的常量值，则可以访问 `while` 语句的嵌入语句。

如果以下至少一个条件为 true，则可到达 `while` 语句的终结点：

- `while` 语句包含可访问的 `break` 语句，该语句可退出 `while` 语句。
- `while` 语句是可访问的，并且布尔表达式没有 `true` 的常量值。

Do 语句

`do` 语句有条件地执行一次或多次嵌入式语句。

```
do_statement
    : 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
    ;
```

执行 `do` 语句，如下所示：

- 控件将被传输到嵌入语句。
- 当和如果控件到达嵌入语句的终结点（可能是执行 `continue` 语句时），将计算*boolean_expression*（布尔表达式）。如果布尔表达式产生 `true`，则将控制转移到 `do` 语句的开头。否则，控制将转移到 `do` 语句的终点。

在 `do` 语句的嵌入语句中，`break` 语句（break 语句）可用于将控制转移到 `do` 语句的终结点（从而结束嵌入语句的迭代），而 `continue` 语句（continue 语句）可用于将控制转移到嵌入语句的终结点。

如果 `do` 语句可访问，则可以访问 `do` 语句的嵌入语句。

如果以下至少一个条件为 true，则可到达 `do` 语句的终结点：

- `do` 语句包含可访问的 `break` 语句，该语句可退出 `do` 语句。
- 嵌入语句的结束点是可访问的，并且布尔表达式没有 `true` 的常量值。

For 语句

`for` 语句计算一系列初始化表达式，然后，在条件为 true 时，重复执行嵌入语句并计算迭代表达式的序列。

```
for_statement
    : 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')' embedded_statement
    ;

for_initializer
    : local_variable_declaration
    | statement_expression_list
    ;

for_condition
    : boolean_expression
    ;

for_iterator
    : statement_expression_list
    ;

statement_expression_list
    : statement_expression (',' statement_expression)*
    ;
```

For_initializer(如果存在)由*local_variable_declaration*(局部变量声明)或以逗号分隔的*statement_expression*s(expression 语句)的列表组成。*For_initializer*声明的局部变量的作用域从变量的*local_variable_declarator*开始,并延伸到嵌入语句的末尾。范围包括*for_condition*和*for_iterator*。

For_condition(如果存在)必须是*boolean_expression*(布尔表达式)。

For_iterator(如果存在)由以逗号分隔的*Statement_expression*s(expression 语句)列表组成。

For 语句的执行方式如下:

- 如果*for_initializer*存在,变量初始值设定项或语句表达式将按照它们的写入顺序执行。此步骤只执行一次。
- 如果*for_condition*存在,则将对其进行计算。
- 如果*for_condition*不存在或者计算产生 `true`,则将控制转移到嵌入语句。当和如果控件到达嵌入语句的结束点(可能是执行 `continue` 语句)时,将按顺序计算*for_iterator*的表达式(如果有),然后执行另一次迭代,从对上述步骤中的*for_condition*进行计算开始。
- 如果*for_condition*存在并且计算生成 `false`,则控制将转移到 `for` 语句的终结点。

在 `for` 语句的嵌入语句内,`break` 语句(break 语句)可用于将控制转移到 `for` 语句的终结点(从而结束嵌入语句的迭代),而 `continue` 语句(continue 语句)可用于将控制转移到嵌入语句的终结点(从而执行*for_iterator*并执行 `for` 语句的另一次迭代,从*for_condition*开始)。

如果满足以下条件之一,则可以访问 `for` 语句的嵌入语句:

- `for` 语句是可访问的,并且不存在*for_condition* 。
- `for` 语句是可访问的,并且*for_condition*存在并且没有 `false` 的常量值。

如果以下至少一个条件为 true,则可到达 `for` 语句的终结点:

- `for` 语句包含可访问的 `break` 语句,该语句可退出 `for` 语句。
- `for` 语句是可访问的,并且*for_condition*存在并且没有 `true` 的常量值。

Foreach 语句

`foreach` 语句枚举集合中的元素,并对集合中的每个元素执行嵌入语句。

```
foreach_statement
    : 'foreach' '(' local_variable_type identifier 'in' expression ')' embedded_statement
    ;
```

`foreach` 语句的*类型*和*标识符*声明语句的***迭代变量***。如果 `var` 标识符作为*local_variable_type*提供，且没有任何名为 `var` 的类型在范围内，则将迭代变量称为***隐式类型的迭代变量***，并将其类型视为 `foreach` 语句的元素类型，如下所示。迭代变量对应于一个只读局部变量，该局部变量具有扩展到嵌入语句的范围。在 `foreach` 语句执行过程中，迭代变量表示当前正在为其执行迭代的集合元素。如果嵌入的语句尝试修改迭代变量(通过赋值或 `++` 和 `--` 运算符)，或将迭代变量作为 `ref` 或 `out` 参数传递，则会发生编译时错误。

在以下情况下，为简洁起见，`IEnumerable`、`IEnumerator``IEnumerable<T>` 和 `IEnumerator<T>` 引用命名空间中的相应类型 `System.Collections` 和 `System.Collections.Generic`。

Foreach 语句的编译时处理首先确定表达式的***集合类型***、***枚举器类型***和***元素类型***。这种决定将按如下方式进行：

- 如果*表达式*的类型 `X` 为数组类型，则从 `X` 到 `IEnumerable` 接口(因为 `System.Array` 实现此接口)存在隐式引用转换。***集合类型***是 `IEnumerable` 接口，***枚举器类型***为 `IEnumerator` 接口，***元素类型***是数组类型 `X` 的元素类型。
- 如果 `dynamic` `X` 的*表达式*的类型，则将从*expression*到 `IEnumerable` 接口(隐式动态转换)的隐式转换。***集合类型***是 `IEnumerable` 接口，***枚举器类型***为 `IEnumerator` 接口。如果将 `var` 标识符指定为*local_variable_type*则 `dynamic` 元素类型，否则将 `object` 该***元素类型***。
- 否则，请确定类型 `X` 是否具有适当的 `GetEnumerator` 方法：
  - 在类型 `X` 上执行成员查找，标识符 `GetEnumerator`，不包含任何类型参数。如果成员查找不会生成匹配项，或者它产生了多义性，或者产生了不是方法组的匹配项，请检查可枚举的接口，如下所述。如果成员查找产生了除方法组以外的任何内容，则建议发出警告。
  - 使用生成的方法组和空参数列表执行重载决策。如果重载决策导致没有适用的方法、导致歧义或产生单个最佳方法，但该方法是静态的或非公共的，请按如下所述检查可枚举的接口。如果重载决策产生了除明确的公共实例方法之外的任何内容，或者没有适用方法，则建议发出警告。
  - 如果 `GetEnumerator` 方法的返回类型 `E` 不是类、结构或接口类型，则会生成错误，并且不会执行任何其他步骤。
  - 成员查找在标识符为 `Current` 且无类型参数 `E` 上执行。如果成员查找没有生成任何匹配项，则结果为错误，或结果为除允许读取的公共实例属性之外的任何内容，并不执行任何其他步骤。
  - 成员查找在标识符为 `MoveNext` 且无类型参数 `E` 上执行。如果成员查找没有生成任何匹配项，则结果为错误，或结果为除方法组外的任何内容，并不执行任何其他步骤。
  - 使用空参数列表对方法组执行重载决策。如果重载决策导致没有适用的方法，导致歧义，或者导致单个最佳方法，但该方法是静态的或非公共的，或者它的返回类型不 `bool`，则会生成错误，并且不会执行任何其他步骤。
  - ***集合类型***为 `X`，***枚举器类型***为 `E`，***元素类型***是 `Current` 属性的类型。
- 否则，请检查可枚举的接口：
  - 如果 `Ti` 的所有类型都是从 `X` 到 `IEnumerable<Ti>` 的隐式转换，则有一个唯一类型 `T`，`T` 不 `dynamic`，而对于所有其他 `Ti` 存在从 `IEnumerable<T>` 到 `IEnumerable<Ti>` 的隐式转换，则该***集合类型***为接口 `IEnumerable<T>`，***枚举器类型***为接口 `IEnumerator<T>`，且***元素类型***为 `T`。
  - 否则，如果有多个这样的类型 `T`，则会生成错误，并且不会执行任何其他步骤。
  - 否则，如果存在从 `X` 到 `System.Collections.IEnumerable` 接口的隐式转换，则***集合类型***为此接口，***枚举器类型***为接口 `System.Collections.IEnumerator`，并且***元素类型***是 `object` 的。
  - 否则，将生成错误，并且不执行任何其他步骤。

如果成功，上述步骤会明确生成 `C`、枚举器类型 `E` 和元素类型 `T` 的集合类型。窗体的 foreach 语句

```
foreach (V v in x) embedded_statement
```

然后扩展为：

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded_statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

变量 `e` 对 `x` 或程序的嵌入语句或任何其他源代码都不可见或可访问。变量 `v` 在嵌入语句中是只读的。如果没有从 `T`（元素类型）到 `V`（foreach 语句中的*local_variable_type*）的显式转换（显式转换），则会生成错误，并且不会执行任何其他步骤。如果 `x` 的值 `null`，则在运行时将引发 `System.NullReferenceException`。

允许实现以不同的方式实现给定的 foreach 语句（例如出于性能原因），前提是该行为与上述扩展一致。

While 循环内 `v` 的位置对于在*embedded_statement*中发生的任何匿名函数的捕获是非常重要的。

例如：

```
int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}

f();
```

如果 `v` 是在 while 循环之外声明的，则它将在所有迭代之间共享，而 for 循环后，其值将是最终值 `13`，这是 `f` 的调用的输出。相反，由于每个迭代都有其自己的变量 `v`，因此，第一次迭代中 `f` 捕获的变量将继续保存值 `7`，这就是要打印的值。（注意：更早版本C#的声明在 while 循环外 `v`。）

Finally 块的主体按照以下步骤进行构造：

- 如果存在从 `E` 到 `System.IDisposable` 接口的隐式转换，则
  - 如果 `E` 是不可以为 null 的值类型，则将 finally 子句扩展为语义等效项：

    ```
    finally {
        ((System.IDisposable)e).Dispose();
    }
    ```

  - 否则，finally 子句将扩展到语义等效项：

    ```
    finally {
        if (e != null) ((System.IDisposable)e).Dispose();
    }
    ```

    除非 `E` 是值类型，或者是实例化为值类型的类型参数，否则，将 `e` 转换为 `System.IDisposable` 将不会导致装箱发生。
- 否则，如果 `E` 是密封类型，则 finally 子句将扩展为空块：

```
finally {
}
```

- 否则，finally 子句将扩展为：

```
finally {
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

局部变量 `d` 对任何用户代码都不可见或不可访问。具体而言，它不会与范围包含 finally 块的任何其他变量发生冲突。

`foreach` 遍历数组元素的顺序如下：对于一维数组元素按递增的索引顺序进行遍历，从索引 `0` 开始，以 index `Length - 1` 结尾。对于多维数组，会遍历元素，以便先增加最右侧维度的索引，然后再增加下一个左侧维度，依此类推。

下面的示例按元素顺序输出二维数组中的每个值：

```
using System;

class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);

        Console.WriteLine();
    }
}
```

生成的输出如下所示：

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

示例中

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

`n` 的类型将被推断为 `int`（`numbers` 的元素类型）。

跳转语句

跳转语句无条件传输控制。

```
jump_statement
    : break_statement
    | continue_statement
    | goto_statement
    | return_statement
    | throw_statement
    ;
```

跳转语句将控制转移到的位置称为跳转语句的*目标*。

如果跳转语句发生在块中，并且该跳转语句的目标在该块外，则可以使用跳转语句*退出*块。尽管一个跳转语句可以将控制转移到块外，但它永远不能将控制转移到块中。

由于存在干预 `try` 语句，执行跳转语句非常复杂。在缺少此类 `try` 语句的情况下，跳转语句会无条件地将控制从跳转语句转移到其目标。如果存在此类干预 `try` 语句，则执行会更复杂。如果跳转语句退出一个或多个具有关联 `finally` 块的 `try` 块，则控件最初会传输到最内层 `try` 语句的 `finally` 块。当和如果控件到达 `finally` 块的终点时，控制将转移到下一个封闭 `try` 语句的 `finally` 块。重复此过程，直到执行了所有干预 `try` 语句 `finally` 块。

示例中

```
using System;

class Test
{
    static void Main() {
        while (true) {
            try {
                try {
                    Console.WriteLine("Before break");
                    break;
                }
                finally {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

在控件传输到跳转语句的目标之前，将执行与两个 `try` 语句关联的 `finally` 块。

生成的输出如下所示：

```
Before break
Innermost finally block
Outermost finally block
After break
```

Break 语句

`break` 语句退出最近的封闭 `switch`、`while`、`do`、`for` 或 `foreach` 语句。

```
break_statement
    : 'break' ';'
    ;
```

`break` 语句的目标是最近的封闭 `switch`、`while`、`do`、`for` 或 `foreach` 语句的终点。如果 `break` 语句未由 `switch`、`while`、`do`、`for` 或 `foreach` 语句括起来，则会发生编译时错误。

当多个 `switch`、`while`、`do`、`for` 或 `foreach` 语句彼此嵌套时，`break` 语句仅适用于最内层的语句。若要跨多个嵌套级别传递控制，必须使用 `goto` 语句(goto 语句)。

`break` 语句无法退出 `finally` 块(try 语句)。如果 `break` 语句发生在 `finally` 块中，则 `break` 语句的目标必须在同一 `finally` 块内;否则，会发生编译时错误。

执行 `break` 语句，如下所示：

- 如果 `break` 语句退出一个或多个具有关联 `finally` 块的 `try` 块，则控件最初会传输到最内层 `try` 语句的 `finally` 块。当和如果控件到达 `finally` 块的终点时，控制将转移到下一个封闭 `try` 语句的 `finally` 块。重复此过程，直到执行了所有干预 `try` 语句 `finally` 块。
- 控件传输到 `break` 语句的目标。

由于 `break` 语句无条件地将控制转移到其他位置，因此无法访问 `break` 语句的终点。

Continue 语句

`continue` 语句启动最近的封闭 `while`、`do`、`for` 或 `foreach` 语句的新迭代。

```
continue_statement
    : 'continue' ';'
    ;
```

`continue` 语句的目标是最近的封闭 `while`、`do`、`for` 或 `foreach` 语句的嵌入语句的终点。如果 `continue` 语句未由 `while`、`do`、`for` 或 `foreach` 语句括起来，则会发生编译时错误。

当多个 `while`、`do`、`for` 或 `foreach` 语句彼此嵌套时，`continue` 语句仅适用于最内层的语句。若要跨多个嵌套级别传递控制，必须使用 `goto` 语句(goto 语句)。

`continue` 语句无法退出 `finally` 块(try 语句)。如果 `continue` 语句发生在 `finally` 块中，则 `continue` 语句的目标必须在同一 `finally` 块内;否则，会发生编译时错误。

执行 `continue` 语句，如下所示：

- 如果 `continue` 语句退出一个或多个具有关联 `finally` 块的 `try` 块，则控件最初会传输到最内层 `try` 语句的 `finally` 块。当和如果控件到达 `finally` 块的终点时，控制将转移到下一个封闭 `try` 语句的 `finally` 块。重复此过程，直到执行了所有干预 `try` 语句 `finally` 块。
- 控件传输到 `continue` 语句的目标。

由于 `continue` 语句无条件地将控制转移到其他位置，因此无法访问 `continue` 语句的终点。

goto 语句

`goto` 语句将控制转移到由标签标记的语句。

```
goto_statement
    : 'goto' identifier ';'
    | 'goto' 'case' constant_expression ';'
    | 'goto' 'default' ';'
    ;
```

`goto` *identifier*语句的目标是带有给定标签的标记语句。如果当前函数成员中不存在具有给定名称的标签，或者如果 `goto` 语句不在标签范围内，则会发生编译时错误。此规则允许使用 `goto` 语句将控制转移出嵌套作用域，但不允许使用嵌套作用域。示例中

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                         goto done;

            Console.WriteLine("{0} not found", str);
            continue;
    done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

`goto` 语句用于将控制转移出嵌套作用域。

`goto case` 语句的目标是直接封闭 `switch` 语句(switch 语句)中的语句列表，其中包含具有给定常数值的 `case` 标签。如果 `goto case` 语句未使用 `switch` 语句括起来，则如果*constant_expression*不能隐式转换为最近的封闭 `switch` 语句的管理类型，或者最近的封闭 `switch` 语句不包含具有给定常数值的 `case` 标签，则会发生编译时错误。

`goto default` 语句的目标是直接封闭 `switch` 语句(switch 语句)中的语句列表，其中包含一个 `default` 标签。如果 `goto default` 语句未用 `switch` 语句括起来，或者最近的封闭 `switch` 语句不包含 `default` 标签，则会发生编译时错误。

`goto` 语句无法退出 `finally` 块(try 语句)。如果 `goto` 语句发生在 `finally` 块中，则 `goto` 语句的目标必须在同一 `finally` 块中，否则将发生编译时错误。

执行 `goto` 语句，如下所示：

- 如果 `goto` 语句退出一个或多个具有关联 `finally` 块的 `try` 块，则控件最初会传输到最内层 `try` 语句的 `finally` 块。当和如果控件到达 `finally` 块的终点时，控制将转移到下一个封闭 `try` 语句的 `finally` 块。重复此过程，直到执行了所有干预 `try` 语句 `finally` 块。
- 控件传输到 `goto` 语句的目标。

由于 `goto` 语句无条件地将控制转移到其他位置，因此无法访问 `goto` 语句的终点。

Return 语句

`return` 语句将控制权返回到 `return` 语句显示函数的当前调用方。

```
return_statement
    : 'return' expression? ';'
    ;
```

不带表达式的 `return` 语句只能在不计算值的函数成员中使用，即，具有结果类型(方法主体) `void` 的方法、属性或索引器的 `set` 访问器、事件的 `add` 和 `remove` 访问器、实例构造函数、静态构造函数或析构函数。

带有表达式的 `return` 语句只能在计算值的函数成员中使用，即，具有非 void 结果类型的方法、属性或索引器的 `get` 访问器或用户定义的运算符。隐式转换(隐式转换)必须存在于表达式的类型到包含函数成员的返回类型。

Return 语句还可用于匿名函数表达式(匿名函数表达式)的主体中，并参与确定哪些转换存在这些函数。

`return` 语句出现在 `finally` 块中(try 语句)，这是编译时错误。

执行 `return` 语句，如下所示：

- 如果 `return` 语句指定一个表达式，将计算该表达式，并通过隐式转换将生成的值转换为包含函数的返回类型。转换的结果成为函数生成的结果值。
- 如果 `return` 语句由一个或多个具有关联 `finally` 块的 `try` 或 `catch` 块括起来，则控件最初会传输到最内层 `finally` 语句的 `try` 块。当和如果控件到达 `finally` 块的终点时，控制将转移到下一个封闭 `try` 语句的 `finally` 块。此过程将重复进行，直到执行完所有封闭 `try` 语句 `finally` 块。
- 如果包含函数不是异步函数，则会将控件返回到包含函数的调用方，同时返回结果值(如果有)。
- 如果包含函数是一个异步函数，则将控制权返回给当前调用方，并在返回任务中记录结果值(如果有)，如(枚举器接口)中所述。

由于 `return` 语句无条件地将控制转移到其他位置，因此无法访问 `return` 语句的终点。

Throw 语句

`throw` 语句引发异常。

```
throw_statement
    : 'throw' expression? ';'
    ;
```

带有表达式的 `throw` 语句将引发通过计算表达式生成的值。表达式必须表示从 `System.Exception` 派生的类类型 `System.Exception` 类类型的值，或者是具有其有效基类的 `System.Exception` (或子类)的类型参数类型的值。如果表达式的计算生成 `null`，则改为引发 `System.NullReferenceException`。

不带表达式的 `throw` 语句只能用在 `catch` 块中，在这种情况下，该语句会重新引发该 `catch` 块当前正在处理的异常。

由于 `throw` 语句无条件地将控制转移到其他位置，因此无法访问 `throw` 语句的终点。

当引发异常时，控制将转移到可处理异常的封闭 `try` 语句中的第一个 `catch` 子句。从引发的异常点到将控件传输到适当的异常处理程序的时间点的过程称为 "***异常传播***"。异常的传播包括重复计算以下步骤，直到找到与异常匹配的 `catch` 子句。在此说明中，引发异常的位置最初为***引发点***的位置。

- 在当前函数成员中，将检查包含引发点的每个 `try` 语句。对于 `S` 的每个语句，从最内层的 `try` 语句开始，到最外面的 `try` 语句结束，计算以下步骤：
  - 如果 `S` 的 `try` 块包含了序列点，并且如果 S 具有一个或多个 `catch` 子句，则将根据 " Try" 语句部分中指定的规则，按显示顺序检查 `catch` 子句以查找异常的合适处理程序。如果找到匹配的 `catch` 子句，则通过将控制转移到该 `catch` 子句的块来完成异常传播。
  - 否则，如果 `try` 块或 `catch` 块 `S` 包含了引发点，并且 `S` 有一个 `finally` 块，则将控制转移到 `finally` 块。如果 `finally` 块引发另一个异常，则终止处理当前异常。否则，当控件到达 `finally` 块的终点时，将继续处理当前异常。
- 如果在当前函数调用中未找到异常处理程序，则终止函数调用，并发生以下情况之一：
  - 如果当前函数为非异步，则会为函数调用方重复上述步骤，并将引发点与调用函数成员的语句对

应。

  - 如果当前函数为 async 并返回任务，则会在返回任务中记录异常，该异常将被置于 "枚举器接口" 中所述的 "出错" 或 "已取消" 状态。
  - 如果当前函数为 async 和 void 返回，则会通知当前线程的同步上下文，如可枚举接口中所述。
- 如果异常处理终止当前线程中的所有函数成员调用，指示该线程没有异常的处理程序，则该线程本身将终止。此类终止的影响是由实现定义的。

Try 语句

`try` 语句提供一种机制，用于捕获在执行块期间发生的异常。此外，`try` 语句还可以指定在控制离开 `try` 语句时始终执行的代码块。

```
try_statement
    : 'try' block catch_clause+
    | 'try' block finally_clause
    | 'try' block catch_clause+ finally_clause
    ;

catch_clause
    : 'catch' exception_specifier? exception_filter?  block
    ;

exception_specifier
    : '(' type identifier? ')'
    ;

exception_filter
    : 'when' '(' expression ')'
    ;

finally_clause
    : 'finally' block
    ;
```

有三种可能的 `try` 语句形式：

- 后跟一个或多个 `catch` 块的 `try` 块。
- 后跟 `finally` 块的 `try` 块。
- 后跟一个或多个 `catch` 块后跟一个 `finally` 块的 `try` 块。

当 `catch` 子句指定一个*exception_specifier*时，该类型必须是 `System.Exception`、从 `System.Exception` 派生的类型或将 `System.Exception`（或子类）作为其有效基类的类型参数类型。

当 `catch` 子句使用*标识符*同时指定*exception_specifier*时，将声明一个具有给定名称和类型的***异常变量***。异常变量对应于一个本地变量，该变量的作用域扩展了 `catch` 子句。在*exception_filter*和*块*的执行期间，异常变量表示当前正在处理的异常。出于明确赋值检查的目的，在其整个范围内将异常变量视为明确赋值。

除非 `catch` 子句包含异常变量名称，否则无法访问筛选器和 `catch` 块中的异常对象。

不指定*exception_specifier*的 `catch` 子句称为一般 `catch` 子句。

某些编程语言可能会支持不能表示为派生自 `System.Exception` 的对象的异常，但C#代码不会生成此类异常。可以使用常规 `catch` 子句来捕捉此类异常。因此，一般的 `catch` 子句在语义上与指定类型 `System.Exception` 的子句不同，因为前者还可以捕获其他语言的异常。

若要查找异常的处理程序，请按词法顺序检查 `catch` 子句。如果 `catch` 子句指定了一种类型，但没有指定异常筛选器，则在同一 `try` 语句中，后面的 `catch` 子句的编译时错误为指定与该类型相同或派生自该类型的类型。

如果 `catch` 子句未指定任何类型并且没有筛选器，则它必须是该 `try` 语句的最后一个 `catch` 子句。

在 `catch` 块中，不带表达式的 `throw` 语句（throw 语句）可用于重新引发由 `catch` 块捕获的异常。对异常变量的赋值不会改变重新引发的异常。

示例中

```
using System;

class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;                // re-throw
        }
    }

    static void G() {
        throw new Exception("G");
    }

    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

方法 `F` 捕获异常，将一些诊断信息写入控制台，更改异常变量，并重新引发异常。重新引发的异常是原始异常，因此生成的输出为：

```
Exception in F: G
Exception in Main: G
```

如果第一个 catch 块引发 `e` 而不是重新引发当前异常，则生成的输出将如下所示：

```
Exception in F: G
Exception in Main: F
```

`break`、`continue` 或 `goto` 语句将控制转移到 `finally` 块，这是编译时错误。当 `finally` 块中出现 `break`、`continue` 或 `goto` 语句时，语句的目标必须在同一 `finally` 块中，否则将发生编译时错误。

`return` 语句发生在 `finally` 块中是编译时错误。

执行 `try` 语句，如下所示：

- 控件传输到 `try` 块。
- 当和如果控件到达 `try` 块的终点时：
  - 如果 `try` 语句具有 `finally` 块，则将执行 `finally` 块。
  - Control 会传输到 `try` 语句的终点。

- 如果在执行 `try` 块期间将异常传播到 `try` 语句：
  - `catch` 子句（如果有）将按外观的顺序进行检查，以查找适用于异常的处理程序。如果 `catch` 子句未指定类型，或指定异常类型或异常类型的基类型：
    - 如果 `catch` 子句声明异常变量，则会将异常对象分配给异常变量。
    - 如果 `catch` 子句声明异常筛选器，则会计算筛选器。如果计算结果为 `false`，catch 子句不是匹配项，并且搜索将继续执行适用于适当处理程序的任何后续 `catch` 子句。
    - 否则，`catch` 子句被视为匹配，并将控制转移到匹配的 `catch` 块。
    - 当和如果控件到达 `catch` 块的终点时：
      - 如果 `try` 语句具有 `finally` 块，则将执行 `finally` 块。
      - Control 会传输到 `try` 语句的终点。
    - 如果在执行 `catch` 块期间将异常传播到 `try` 语句：
      - 如果 `try` 语句具有 `finally` 块，则将执行 `finally` 块。
      - 异常将传播到下一个封闭 `try` 语句。
  - 如果 `try` 语句没有 `catch` 子句，或者如果没有 `catch` 子句与异常匹配，则为：
    - 如果 `try` 语句具有 `finally` 块，则将执行 `finally` 块。
    - 异常将传播到下一个封闭 `try` 语句。

当控制离开 `try` 语句时，始终会执行 `finally` 块的语句。无论是由于执行 `break`、`continue`、`goto` 或 `return` 语句导致控件传输，还是由于将异常传播到 `try` 语句而导致的，都是如此。

如果在执行 `finally` 块期间引发了异常，并且未在同一 finally 块中捕获该异常，则该异常将传播到下一个封闭 `try` 语句。如果正在传播其他异常，则该异常将丢失。传播异常的过程将在 `throw` 语句（throw 语句）的说明中进一步讨论。

如果 `try` 语句可访问，则可访问 `try` 语句的 `try` 块。

如果 `try` 语句可访问，则可访问 `try` 语句 `catch` 块。

如果 `try` 语句可访问，则可访问 `try` 语句的 `finally` 块。

如果以下两个条件均为 true，则可以访问 `try` 语句的终结点：

- `try` 块的终结点是可访问的，或者至少有一个 `catch` 块可访问的终结点。
- 如果存在 `finally` 块，则可访问 `finally` 块的终结点。

Checked 和 unchecked 语句

`checked` 和 `unchecked` 语句用于控制整型算术运算和转换的***溢出检查上下文***。

```
checked_statement
    : 'checked' block
    ;

unchecked_statement
    : 'unchecked' block
    ;
```

`checked` 语句导致在已检查的上下文中计算*块*中的所有表达式，而 `unchecked` 语句导致在未检查的上下文中计算*块*中的所有表达式。

`checked` 和 `unchecked` 语句完全等效于 `checked` 和 `unchecked` 运算符（checked 和 unchecked 运算符），只不过它们在块而不是表达式上操作。

Lock 语句

`lock` 语句获取给定对象的互斥锁，执行语句，然后释放该锁。

```
lock_statement
    : 'lock' '(' expression ')' embedded_statement
    ;
```

`lock` 语句的表达式必须表示已知为*reference_type*的类型的值。对于 `lock` 语句的表达式，不会执行任何隐式装箱转换(装箱转换)，因此，表达式会出现编译时错误，以表示*value_type*的值。

窗体的 `lock` 语句

```
lock (x) ...
```

其中 `x` 是*reference_type*的表达式，它完全等效于

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

不同的是 `x` 只计算一次。

持有互斥锁时，在同一执行线程中执行的代码也可以获取和释放锁。但是，在其他线程中执行的代码被阻止获取锁定，直到锁定被释放。

不建议锁定 `System.Type` 对象以同步对静态数据的访问。其他代码可能会锁定同一类型，这可能会导致死锁。更好的方法是通过锁定专用静态对象来同步对静态数据的访问。例如：

```
class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }

    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```

using 语句

`using` 语句获取一个或多个资源，执行语句，然后释放资源。

```
using_statement
    : 'using' '(' resource_acquisition ')' embedded_statement
    ;

resource_acquisition
    : local_variable_declaration
    | expression
    ;
```

*资源*是实现 `System.IDisposable` 的类或结构，其中包括一个名为 `Dispose` 的无参数方法。使用资源的代码可以调用 `Dispose`，以指示不再需要资源。如果未调用 `Dispose`，则最终将作为垃圾回收的结果发生。

如果*local_variable_declaration resource_acquisition*的形式，则*local_variable_declaration*的类型必须是 `dynamic` 或可隐式转换为 `System.IDisposable` 的类型。如果*resource_acquisition*的形式为*expression*，则此表达式必须可隐式转换为 `System.IDisposable`。

在*resource_acquisition*中声明的局部变量是只读的，并且必须包含初始值设定项。如果嵌入的语句尝试修改这些本地变量(通过赋值或 `++` 和 `--` 运算符)，则会发生编译时错误，请获取它们的地址，或将其作为 `ref` 或 `out` 参数传递。

`using` 语句转换为三个部分：获取、使用和处理。资源的使用隐式包含在包含 `finally` 子句的 `try` 语句中。此 `finally` 子句将释放资源。如果获取 `null` 资源，则不会调用任何 `Dispose`，并且不会引发异常。如果资源的类型为 `dynamic` 则通过隐式动态转换(隐式动态转换)将其动态转换为在获取期间 `IDisposable`，以确保转换在使用和处置之前成功。

窗体的 `using` 语句

```
using (ResourceType resource = expression) statement
```

对应于三个可能的扩展中的一个。如果 `ResourceType` 是不可为 null 的值类型，则扩展为

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}
```

否则，当 `ResourceType` 是可以为 null 的值类型或引用类型而不是 `dynamic` 时，展开是

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}
```

否则，当 `dynamic``ResourceType` 时，展开是

```
{
    ResourceType resource = expression;
    IDisposable d = (IDisposable)resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
```

在任一扩展中，`resource` 变量在嵌入语句中是只读的，并且在嵌入的语句中不可访问 `d` 变量，并且不可见。

允许实现以不同方式实现给定的 using 语句（例如出于性能原因），前提是该行为与上述扩展一致。

窗体的 `using` 语句

```
using (expression) statement
```

具有相同的三个可能的扩展。在这种情况下 `ResourceType` 隐式编译时 `expression` 类型（如果有）。否则，接口 `IDisposable` 自身用作 `ResourceType`。在嵌入的语句中，不能访问 `resource` 变量，并且该变量不可见。

如果*resource_acquisition*采用*local_variable_declaration*的形式，则可以获取给定类型的多个资源。窗体的 `using` 语句

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

完全等效于一系列嵌套的 `using` 语句：

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
            using (ResourceType rN = eN)
                statement
```

下面的示例创建一个名为 `log.txt` 的文件，并向该文件写入两行文本。然后，该示例将打开该文件以进行读取，并将包含的文本行复制到控制台。

```
using System;
using System.IO;

class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }

        }
    }
}
```

由于 `TextWriter` 和 `TextReader` 类实现了 `IDisposable` 接口，因此该示例可以使用 `using` 语句，以确保在执行写入或读取操作后正确关闭基础文件。

Yield 语句

在迭代器块(块)中使用 `yield` 语句，以向迭代器的枚举器对象(枚举器对象)或可枚举对象(可枚举对象)生成值或表示迭代结束。

```
yield_statement
    : 'yield' 'return' expression ';'
    | 'yield' 'break' ';'
    ;
```

`yield` 不是保留字;仅当紧靠在 `return` 或 `break` 关键字之前使用时，它才具有特殊意义。在其他上下文中，`yield` 可以用作标识符。

`yield` 语句可以出现的位置有多个限制，如下所述。

- 如果 `yield` 语句(窗体)出现在*method_body*外，*operator_body*或*accessor_body*将出现编译时错误。
- 在匿名函数内出现的 `yield` 语句(任一窗体)的编译时错误。
- 在 `try` 语句的 `finally` 子句中出现 `yield` 语句(窗体)的编译时错误。
- 如果 `yield return` 语句出现在包含任何 `catch` 子句的 `try` 语句中的任何位置，则会发生编译时错误。

下面的示例演示 `yield` 语句的一些有效和无效用法。

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;        // Ok
        yield break;           // Ok
    }
    finally {
        yield return 2;        // Error, yield in finally
        yield break;           // Error, yield in finally
    }

    try {
        yield return 3;        // Error, yield return in try...catch
        yield break;           // Ok
    }
    catch {
        yield return 4;        // Error, yield return in try...catch
        yield break;           // Ok
    }

    D d = delegate {
        yield return 5;        // Error, yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;            // Error, wrong return type for an iterator block
}
```

隐式转换(隐式转换)必须从 `yield return` 语句中表达式的类型到迭代器的 yield 类型(yield 类型)存在。

执行 `yield return` 语句，如下所示：

- 计算语句中给定的表达式，将其隐式转换为 yield 类型，并将其分配给枚举器对象的 `Current` 属性。

- 迭代器块的执行被挂起。如果 `yield return` 语句位于一个或多个 `try` 块中，则不会执行关联的 `finally` 块。
- 枚举器对象的 `MoveNext` 方法将 `true` 返回到其调用方，指示枚举器对象已成功地前进到下一项。

对枚举器对象的 `MoveNext` 方法的下一次调用会从其最后挂起的位置继续执行迭代器块。

执行 `yield break` 语句，如下所示：

- 如果 `yield break` 语句由一个或多个具有关联 `finally` 块的 `try` 块括起来，则控件最初会传输到最内部 `try` 语句的 `finally` 块。当和如果控件到达 `finally` 块的终点时，控制将转移到下一个封闭 `try` 语句的 `finally` 块。此过程将重复进行，直到执行完所有封闭 `try` 语句 `finally` 块。
- 控制权将返回给迭代器块的调用方。这是枚举器对象的 `MoveNext` 方法或 `Dispose` 方法。

由于 `yield break` 语句无条件地将控制转移到其他位置，因此无法访问 `yield break` 语句的终点。

命名空间

2020/11/2 • • Edit Online

C# 程序的组织结构使用的命名空间。使用命名空间为"内部"组织系统的程序，并作为"外部"组织系统 — 一种方法提供向其他程序公开的程序元素。

Using 指令 (Using 指令) 提供，以便命名空间的使用。

编译单元

一个*compilation_unit*定义源文件的整体结构。编译单元包含零个或多*using_directive*s 后跟零个或多*global_attributes*跟零个或多*namespace_member_declaration*s.

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes? namespace_member_declaration*
    ;
```

C# 程序包含一个或多个编译单元，每个包含在单独的源代码文件中。编译 C# 程序时，所有编译单元一起进行处理。因此，编译单元可以依赖于彼此，可能是以循环方式。

*Using_directive*秒的编译单元影响*global_attributes*并*namespace_member_declaration*s 该编译单元，但不起任何作用其他编译单元。

Global_attributes (属性) 编译单元的允许的目标程序集和模块的属性的规范。程序集和模块作为物理容器的类型。程序集可能包含多个物理上独立的模块。

*Namespace_member_declaration*s 程序的每个编译单元将成员分配到名为全局命名空间的单个声明空间。例如：

文件 `A.cs`：

```
class A {}
```

文件 `B.cs`：

```
class B {}
```

两个编译单元中分配到单一的全局命名空间，在这种情况下声明两个类的完全限定名称 `A` 和 `B`。因为这两个编译单元中分配到同一声明空间，重要的是成员的一个错误如果每个包含具有相同名称的声明。

命名空间声明

一个*namespace_declaration*包含关键字 `namespace` 后，跟命名空间名称和正文后，跟分号。

```
namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';'?
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive* namespace_member_declaration* '}'
    ;
```

一个*namespace_declaration*中的顶级声明中可能产生*compilation_unit*或在另一个的成员声明为*namespace_declaration*。当*namespace_declaration*作为中的顶级声明发生*compilation_unit*，该命名空间成为全局命名空间的成员。当*namespace_declaration*出现在另一个*namespace_declaration*，内部命名空间成为外部命名空间的成员。在任一情况下，命名空间名称必须是包含命名空间内唯一的。

命名空间为隐式 `public` 和命名空间声明不能包含任何访问修饰符。

内*namespace_body*，可选*using_directive*s 导入其他命名空间、类型和成员，使他们能够通过限定名而不是直接引用的名称。可选*namespace_member_declaration*s 将成员分配到的命名空间声明空间。请注意，所有*using_directive*s 必须出现在之前的任何成员声明。

*Qualified_identifier*的*namespace_declaration*可能是单个标识符或分隔标识符一系列"`.`"令牌。后一种窗体允许的程序，而不必从词法上嵌套多个命名空间声明中定义的嵌套命名空间。例如，应用于对象的

```
namespace N1.N2
{
    class A {}

    class B {}
}
```

在语义上等效于

```
namespace N1
{
    namespace N2
    {
        class A {}

        class B {}
    }
}
```

命名空间是可扩充的且具有相同的完全限定名称的两个命名空间声明参与到同一声明空间 (声明)。在示例

```
namespace N1.N2
{
    class A {}
}

namespace N1.N2
{
    class B {}
}
```

上面的两个命名空间声明参与到同一声明空间，在这种情况下声明两个类的完全限定名称 `N1.N2.A` 和 `N1.N2.B` 。由于向同一声明空间分配的两个声明，它就已出错，如果每个包含一个具有相同名称的成员的声明。

外部别名

*Extern_alias_directive*引入了可作为一个命名空间的别名的标识符。外部程序的源代码和也适用于嵌套的命名空间的别名的命名空间的别名的命名空间的规范。

```
extern_alias_directive
    : 'extern' 'alias' identifier ';'
    ;
```

作用域*extern_alias_directive*通过进行了扩展*using_directive*s，*global_attributes*和*namespace_member_declaration*s 的直接包含编译单元或命名空间正文。

中包含的编译单元或命名空间体*extern_alias_directive*，通过引入的标识符*extern_alias_directive*可以用于引用别名的命名空间。它是编译时错误*标识符*为单词 `global` 。

*Extern_alias_directive*使别名可在特定的编译单元或命名空间体，但它不会影响到基础的声明空间的任何新成员。换而言之，*extern_alias_directive*不是可传递的但，而是只影响的编译单元或命名空间体其中发生。

下面的程序声明，并使用两个外部别名 `X` 和 `Y` ，每个表示不同的命名空间层次结构的根的：

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

该程序声明是否存在的外部别名 `X` 和 `Y` ，但实际定义的别名是外部的程序。具有相同名称 `N.B` 现在可以作为引用类 `X.N.B` 并 `Y.N.B` ，或使用的命名空间别名限定符 `X::N.B` 和 `Y::N.B` 。如果程序声明外部别名为其提供任何外部定义，就会出错。

using 指令

Using 指令便于您使用的命名空间和其他命名空间中定义的类型。使用的名称解析过程的指令影响*namespace_or_type_name*s (Namespace 和类型名称) 和*simple_name*s (简单名称)，而与声明中，使用指令并不参与到编译单元或在其中使用它们的命名空间的基础的声明空间的新成员。

```
using_directive
    : using_alias_directive
    | using_namespace_directive
    | using_static_directive
    ;
```

一个*using_alias_directive* (Using 别名指令) 引入了命名空间或类型的别名。

一个*using_namespace_directive* (使用的命名空间指令) 导入的命名空间的类型成员。

一个*using_static_directive* (Using static 指令) 导入的嵌套的类型和类型的静态成员。

作用域*using_directive*通过进行了扩展*namespace_member_declaration*s 其直接包含的编译单元或命名空间体。作用域*using_directive*专门不包括其对等方*using_directive*s。因此，对等互连*using_directive*s 不影响，并且它们编写的顺序并不重要。

Using 别名指令

一个*using_alias_directive*引入了可作为命名空间或最近的封闭的编译单元或命名空间体中的类型的别名的标识符。

```
using_alias_directive
    : 'using' identifier '=' namespace_or_type_name ';'
    ;
```

中包含的编译单元或命名空间体中的成员声明*using_alias_directive*，通过引入的标识符*using_alias_directive*可引用给定命名空间或类型。例如：

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;

    class B: A {}
}
```

更高版本中的成员声明中`N3`命名空间，`A`是其别名`N1.N2.A`，因此类`N3.B`派生自类`N1.N2.A`。可以通过创建别名来获取相同的效果`R`有关`N1.N2`然后引用`R.A`：

```
namespace N3
{
    using R = N1.N2;

    class B: R.A {}
}
```

*标识符*的*using_alias_directive*中的编译单元或立即包含命名空间声明空间必须是唯一*using_alias_directive*. 例如：

```
namespace N3
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;        // Error, A already exists
}
```

更高版本，`N3`已有一个成员`A`，因此它是用于编译时错误*using_alias_directive*使用该标识符。同样，它是两个或多个的编译时错误*using_alias_directive*s 同一编译单元或命名空间体中声明相同的名称的别名。

一个*using_alias_directive*使别名可在特定的编译单元或命名空间体，但它不会影响到基础的声明空间的任何新成员。换而言之，*using_alias_directive*不可传递，但而不是只影响的编译单元或命名空间体进行的。在示例

```
namespace N3
{
    using R = N1.N2;
}

namespace N3
{
    class B: R.A {}            // Error, R unknown
}
```

作用域*using_alias_directive*引入`R`仅扩展到在其中包含它，命名空间体中的成员声明使`R`未知第二个命名空间声明中。但是，放置*using_alias_directive*中包含的编译单元会导致为这两个命名空间声明内变为可用的别名：

```
using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}
```

就像常规成员名称引起*using_alias_directive*s 隐藏由嵌套作用域中的同名成员。在示例

```
using R = N1.N2;

namespace N3
{
    class R {}

    class B: R.A {}        // Error, R has no member A
}
```

对引用`R.A`中的声明`B`导致编译时错误，因为`R`指`N3.R`，而不`N1.N2`。

依据的顺序*using_alias_directive*s 是没有意义，和分辨率*namespace_or_type_name*所引用的*using_alias_directive*不会影响*using_alias_directive*本身或由其他*using_directive*中直接包含的编译单元或命名空间体。换而言之，*namespace_or_type_name*的*using_alias_directive*得到解决，如同直接包含的编译单元或命名空间体具有无*using_directive*s。一个*using_alias_directive*但是可能会受*extern_alias_directive*中直接包含的编译单元或命名空间体。在示例

```
namespace N1.N2 {}

namespace N3
{
    extern alias E;

    using R1 = E.N;        // OK

    using R2 = N1;         // OK

    using R3 = N1.N2;      // OK

    using R4 = R2.N2;      // Error, R2 unknown
}
```

上次*using_alias_directive*导致编译时错误，因为它不受第一个*using_alias_directive*。第一个*using_alias_directive*不会导致错误以来外部别名的作用域 `E` 包括*using_alias_directive*。

一个*using_alias_directive*可以创建任何命名空间或类型，包括在其中它将显示的命名空间的别名和任何命名空间或类型嵌套在该命名空间。

通过别名访问命名空间或类型会产生与通过其声明的名称访问该命名空间或类型相同的结果。例如，给定

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;            // refers to N1.N2.A
        R1.N2.A b;            // refers to N1.N2.A
        R2.A c;               // refers to N1.N2.A
    }
}
```

名称 `N1.N2.A`， `R1.N2.A`，并 `R2.A` 等效项和所有引用的类的完全限定的名称是 `N1.N2.A`。

使用别名可以名称封闭式构造类型，但不提供类型实参的情况下无法名称未绑定的泛型类型声明。例如：

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;          // Error, cannot name unbound generic type

    using X = N1.A.B;        // Error, cannot name unbound generic type

    using Y = N1.A<int>;     // Ok, can name closed constructed type

    using Z<T> = N1.A<T>;    // Error, using alias cannot have type parameters
}
```

使用命名空间指令

一个*using_namespace_directive*导入到最近的封闭编译单元或命名空间正文，包含命名空间中启用要使用而无需限定每种类型的标识符的类型。

```
using_namespace_directive
    : 'using' namespace_name ';'
    ;
```

中包含的编译单元或命名空间体中的成员声明*using_namespace_directive*，可以直接引用给定命名空间中包含的类型。例如：

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1.N2;

    class B: A {}
}
```

更高版本中的成员声明中 `N3` 命名空间的类型成员 `N1.N2` 可直接使用，并因此类 `N3.B` 派生自类 `N1.N2.A` 。

一个*using_namespace_directive*导入给定的命名空间中包含的类型，但专门不导入嵌套的命名空间。在示例

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;

    class B: N2.A {}        // Error, N2 unknown
}
```

*using_namespace_directive*中包含的类型导入 `N1` ，但命名空间不嵌套在 `N1` 。因此，对引用 `N2.A` 中的声明 `B` 导致编译时错误，因为没有成员名为 `N2` 作用域中。

与不同*using_alias_directive*即*using_namespace_directive*可以导入标识符已定义封闭的编译单元或命名空间体中的类型。实际上，名称是通过导入*using_namespace_directive*封闭的编译单元或命名空间体中的同名成员隐藏的。例如：

```
namespace N1.N2
{
    class A {}

    class B {}
}

namespace N3
{
    using N1.N2;

    class A {}
}
```

此处，在中的成员声明内 `N3` 命名空间， `A` 是指 `N3.A` 而非 `N1.N2.A` 。

当多个命名空间或类型由导入*using_namespace_directive*s 或*using_static_directive*同一编译单元或命名空间体中包含相同的名称，对引用类型作为该名称*type_name*被视为不明确。在示例

```
namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}

namespace N3
{
    using N1;

    using N2;

    class B: A {}               // Error, A is ambiguous
}
```

这两 `N1` 并 `N2` 包含成员 `A`，并且因为 `N3` 导入两者，引用 `A` 中 `N3` 会导致编译时错误。在此情况下，冲突可以解决通过对引用的限定 `A`，或通过引入*using_alias_directive*选取特定 `A`。例如：

```
namespace N3
{
    using N1;

    using N2;

    using A = N1.A;

    class B: A {}               // A means N1.A
}
```

此外，当多个命名空间或类型由导入*using_namespace_directive*s 或*using_static_directive*同一编译单元或命名空间体中包含类型或成员的条件对作为该名称引用相同的名称，*simple_name*被视为不明确。在示例

```
namespace N1
{
    class A {}
}

class C
{
    public static int A;
}

namespace N2
{
    using N1;
    using static C;

    class B
    {
        void M()
        {
            A a = new A();   // Ok, A is unambiguous as a type-name
            A.Equals(2);     // Error, A is ambiguous as a simple-name
        }
    }
}
```

`N1` 包含类型成员 `A`，并 `C` 包含静态字段 `A`，并且因为 `N2` 导入两者，引用 `A` 作为*simple_name*已不明确以及编

译时出现错误。

像*using_alias_directive*即*using_namespace_directive*不参与到编译单元或命名空间的基础的声明空间的任何新成员，但而不是仅影响它在其中出现的编译单元或命名空间体。

_ *名称*所引用的*using_namespace_directive*解析为相同的方式*namespace_or_type_name*所引用的*using_alias_directive*。因此，*using_namespace_directive*同一编译单元或命名空间体中不会影响彼此和可以任何顺序写入。

Using static 指令

一个*using_static_directive*导入的嵌套的类型和静态成员直接到最近的封闭编译单元或命名空间正文，类型声明中包含启用为每个成员和类型的标识符使用而无需限定。

```
using_static_directive
    : 'using' 'static' type_name ';'
    ;
```

中包含的编译单元或命名空间体中的成员声明*using_static_directive*，可以访问嵌套类型和直接的声明中所包含的静态成员（除了扩展方法）给定的类型可以直接引用。例如：

```
namespace N1
{
    class A
    {
        public class B{}
        public static B M(){ return new B(); }
    }
}

namespace N2
{
    using static N1.A;
    class C
    {
        void N() { B b = M(); }
    }
}
```

更高版本中的成员声明中`N2`静态成员和嵌套的类型的命名空间`N1.A`直接可用，因此该方法`N`能够同时引用`B`和`M`的成员`N1.A`.

一个*using_static_directive*专门不导入扩展的方法直接作为静态方法，但使其可用于扩展方法调用 (扩展方法调用)。在示例

```
namespace N1
{
    static class A
    {
        public static void M(this string s){}
    }
}

namespace N2
{
    using static N1.A;

    class B
    {
        void N()
        {
            M("A");      // Error, M unknown
            "B".M();     // Ok, M known as extension method
            N1.A.M("C"); // Ok, fully qualified
        }
    }
}
```

*using_static_directive*导入的扩展方法 `M` 中包含 `N1.A`，但仅作为扩展方法。因此，对的第一个引用 `M` 中的正文 `B.N` 导致编译时错误，因为没有成员名为 `M` 作用域中。

一个*using_static_directive*只会导入成员并直接在给定类型中声明的类型在基类中声明未成员和类型。

TODO：示例

多个之间存在歧义*using_namespace_directives*并*using_static_directives*中讨论了使用的命名空间指令。

Namespace 成员

一个*namespace_member_declaration*可以是*namespace_declaration* (Namespace 声明) 或*type_declaration* (的类型声明)。

```
namespace_member_declaration
    : namespace_declaration
    | type_declaration
    ;
```

编译单元或命名空间体可以包含*namespace_member_declaration*s 和此类声明参与到包含编译单元或命名空间体的基础的声明空间的新成员。

类型声明

一个*type_declaration*是*class_declaration* (类声明)， *struct_declaration* (结构声明)、一个*interface_declaration* (接口声明)、一个*enum_declaration* (枚举声明)，或*delegate_declaration* (委托声明)。

```
type_declaration
    : class_declaration
    | struct_declaration
    | interface_declaration
    | enum_declaration
    | delegate_declaration
    ;
```

一个*type_declaration*为编译单元中的顶级声明或命名空间、类或结构中的成员声明可以出现。

当一种类型的类型声明 `T` 发生作为编译单元中的顶级声明的新声明类型的完全限定的名称只是 `T` 。当一种类型的类型声明 `T` 发生的命名空间、类或结构的新声明类型的完全限定名称是 `N.T` ，其中 `N` 是包含命名空间、类或结构的完全限定的名称。

在类中声明的类型或结构称为嵌套的类型 (嵌套类型)。

允许的访问修饰符和类型声明的默认访问权限取决于在其中声明发生处的上下文 (声明可访问性):

- 在编译单元或命名空间中声明的类型可以具有 `public` 或 `internal` 访问。默认值是 `internal` 访问。
- 在类中声明的类型可以具有 `public` ， `protected internal` ， `protected` ， `internal` ，或 `private` 访问。默认值是 `private` 访问。
- 在结构中声明的类型可以具有 `public` ， `internal` ，或 `private` 访问。默认值是 `private` 访问。

Namespace 别名限定符

命名空间别名限定符 `::` 就可保证类型名称查找不受引入新类型和成员。命名空间别名限定符始终出现在称为左侧和右侧标识符的两个标识符之间。与普通 `.` 限定符，左侧标识符 `::` 查找限定符最多只能作为外部或使用别名。

一个*qualified_alias_member*定义，如下所示：

```
qualified_alias_member
    : identifier '::' identifier type_argument_list?
    ;
```

一个*qualified_alias_member*可用作*namespace_or_type_name* (Namespace 和类型名称) 或作为中的左操作数*member_access*(成员访问)。

一个*qualified_alias_member*有两种形式之一：

- `N::I<A1, ..., Ak>` 其中 `N` 并 `I` 表示的标识符，，和 `<A1, ..., Ak>` 是类型参数列表。(`K` 始终是至少一个。)
- `N::I` 其中 `N` 和 `I` 表示标识符。(在这种情况下， `K` 被视为零。)

使用此表示法的含义*qualified_alias_member* ，如下所示确定：

- 如果 `N` 的标识符 `global` ，然后搜索全局命名空间 `I` :
  - 如果全局命名空间包含名为命名空间 `I` 并 `K` 为零，则*qualified_alias_member*指的是该命名空间。
  - 否则为如果全局命名空间包含一个名为的非泛型类型 `I` 并 `K` 为零，则*qualified_alias_member*引用该类型。
  - 否则为如果全局命名空间包含一个名为类型 `I` ，其 `K` 类型参数，则*qualified_alias_member*构造具有给定的类型参数的该类型是指。
  - 否则为*qualified_alias_member*是未定义，且将发生编译时错误。
- 否则，从命名空间声明 (Namespace 声明) 直接包含*qualified_alias_member* (如果有)、执行完每个封闭命名空间声明(如果有)，并包含在编译单元结尾*qualified_alias_member*，以下步骤进行计算，直到找到一个实体就是：
  - 如果命名空间声明或编译单元中包含*using_alias_directive*将关联 `N` 类型，则*qualified_alias_member*未定义和编译时间出现错误。
  - 否则为如果命名空间声明或编译单元包含*extern_alias_directive*或*using_alias_directive*将关联 `N` 与命名空间，然后：
    - 如果与关联的命名空间 `N` 包含名为的命名空间 `I` 并 `K` 为零，则*qualified_alias_member*指的是该命名空间。
    - 否则为如果与关联的命名空间 `N` 包含一个名为的非泛型类型 `I` 并 `K` 为零，则*qualified_alias_member*引用该类型。

    - 否则为如果与关联的命名空间`N`包含名为的类型`I`具有`K`类型参数，则*qualified_alias_member*指的是使用构造类型给定的类型的参数。
    - 否则为*qualified_alias_member*是未定义，且将发生编译时错误。
- 否则为*qualified_alias_member*是未定义，且将发生编译时错误。

请注意，命名空间别名限定符与引用的类型的别名一起使用会导致编译时错误。此外请注意，如果标识符`N`是`global`，然后在全局命名空间中执行查找，即使没有使用别名将关联`global`与类型或命名空间。

别名的唯一性

每个编译单元和命名空间正文具有外部别名的单独声明空间和使用别名。因此，虽然外部别名或使用别名的名称必须是唯一的外部别名集内，且使用别名声明直接包含的编译单元或命名空间体中，别名允许具有相同的名称作为类型或命名空间，只要我t 只能用于`::`限定符。

在示例

```
namespace N
{
    public class A {}

    public class B {}
}

namespace N
{
    using A = System.IO;

    class X
    {
        A.Stream s1;            // Error, A is ambiguous

        A::Stream s2;           // Ok
    }
}
```

名称`A`第二个命名空间正文中具有两个可能的含义，因为这两个类`A`和 using 别名`A`作用域中。出于此原因，利用`A`限定名称中`A.Stream`不明确，会导致编译时错误发生。但是，利用`A`与`::`限定符不是一个错误，因为`A`查找仅作为命名空间别名。

类

2020/11/2 • • Edit Online

类是一种数据结构，它可以包含数据成员（常量和字段）、函数成员（方法、属性、事件、索引器、运算符、实例构造函数、析构函数和静态构造函数）和嵌套类型。类类型支持继承，这是一个派生类可以扩展和专用化基类的机制。

类声明

*Class_declaration*是声明新类的*type_declaration*（类型声明）。

```
class_declaration
    : attributes? class_modifier* 'partial'? 'class' identifier type_parameter_list?
      class_base? type_parameter_constraints_clause* class_body ';'?
    ;
```

*Class_declaration*包含一组可选的*特性*（特性），后跟一组可选的*class_modifier*s（类修饰符），后跟一个可选的 `partial` 修饰符 `class`，然后是一个可选的修饰符，后面跟一个可选的修饰符，*然后是*一个可选的*type_parameter_list*（类型参数），后跟一个*可选的 class_base 规范*（类基本规范），后跟通过一组可选的*type_parameter_constraints_clause*s（类型参数约束），后跟一个*class_body*（类体），后面可以跟一个分号。

类声明无法提供*type_parameter_constraints_clause*s，除非它还提供了*type_parameter_list*。

提供*type_parameter_list*的类声明是***泛型类声明***。此外，嵌套在泛型类声明或泛型结构声明中的任何类本身都是一个泛型类声明，因为必须提供包含类型的类型参数才能创建构造类型。

类修饰符

*Class_declaration*可以选择性地包含一系列类修饰符：

```
class_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'abstract'
    | 'sealed'
    | 'static'
    | class_modifier_unsafe
    ;
```

同一修饰符在类声明中多次出现会出现编译时错误。

在嵌套类上允许 `new` 修饰符。它指定类按新修饰符中所述隐藏同名的继承成员。 `new` 修饰符出现在不是嵌套类声明的类声明中是编译时错误。

`public`、`protected`、`internal` 和 `private` 修饰符控制类的可访问性。根据出现类声明的上下文，可能不允许使用其中某些修饰符（声明的可访问性）。

以下各节将讨论 `abstract`、`sealed` 和 `static` 修饰符。

抽象类

`abstract` 修饰符用于指示某个类不完整，并且它将仅用作一个基类。抽象类与非抽象类在以下方面有所不同：

- 抽象类不能直接实例化，而是在抽象类上使用 `new` 运算符的编译时错误。尽管可以具有编译时类型为抽象的变量和值，但这类变量和值必须 `null` 或包含对从抽象类型派生的非抽象类的实例的引用。
- 允许(但不要求)抽象类包含抽象成员。
- 抽象类不能是密封的。

如果非抽象类是从抽象类派生的，则非抽象类必须包含所有继承的抽象成员的实际实现，从而重写这些抽象成员。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` 引入了 `F` 抽象方法。类 `B` 引入了附加方法 `G`，但由于该方法不提供 `F` 的实现，因此 `B` 还必须声明为抽象。类 `C` 重写 `F` 并提供实际实现。由于 `C` 中没有抽象成员，因此允许(但不要求) `C` 为非抽象。

密封类

`sealed` 修饰符用于阻止从类派生。如果将密封类指定为其他类的基类，则会发生编译时错误。

密封类不能也是抽象类。

`sealed` 修饰符主要用于防止意外派生，但它还支持某些运行时优化。特别是，由于知道密封类绝不会有任何派生类，因此可以将密封类实例上的虚函数成员调用转换为非虚拟调用。

静态类

`static` 修饰符用于标记声明为*静态类*的类。静态类不能被实例化，不能用作类型并且只能包含静态成员。只有静态类可以包含扩展方法的声明(扩展方法)。

静态类声明受到下列限制：

- 静态类不能包含 `sealed` 或 `abstract` 修饰符。但请注意，由于无法对静态类进行实例化或派生，因此它的行为就像它既是密封的又是抽象的。
- 静态类不能包含*class_base*规范(类基规范)，并且无法显式指定基类或实现的接口列表。静态类隐式继承自类型 `object`。
- 静态类只能包含静态成员(静态成员和实例成员)。请注意，常量和嵌套类型归类为静态成员。
- 静态类不能包含具有 `protected` 或声明可访问性 `protected internal` 的成员。

这是一种编译时错误，违反其中的任何限制。

静态类没有实例构造函数。不能在静态类中声明实例构造函数，也不会为静态类提供默认的实例构造函数(默认构造函数)。

静态类的成员不是自动静态的，成员声明必须显式包含 `static` 修饰符(常量和嵌套类型除外)。当类嵌套在静态外部类中时，嵌套类不是静态类，除非它显式包含 `static` 修饰符。

引用静态类类型

如果有*namespace_or_type_name*（命名空间和类型名称），则允许引用静态类

- *Namespace_or_type_name*是窗体 `T.I` *namespace_or_type_name*的 `T`，或
- *Namespace_or_type_name*是窗体 `typeof(T)` 的*typeof_expression*（参数列表1）中的 `T`。

如果*primary_expression*（函数成员），则允许引用静态类

- *Primary_expression*是窗体 `E.I` 的*member_access*中的 `E`（对动态重载决策进行编译时检查）。

在其他任何上下文中，引用静态类会导致编译时错误。例如，将静态类用作基类、成员的构成类型（嵌套类型）、泛型类型参数或类型参数约束时，将会出现错误。同样，静态类不能用于数组类型、指针类型、`new` 表达式、强制转换表达式、`is` 表达式、`as` 表达式、`sizeof` 表达式或默认值表达式。

Partial 修饰符

`partial` 修饰符用于指示此*class_declaration*为分部类型声明。具有相同名称的多个分部类型声明在封闭命名空间或类型声明中组合在一起，并遵循在部分类型中指定的规则组成一个类型声明。

如果在不同的上下文中生成或维护这些段，则通过单独的程序文本段来声明的类的声明会很有用。例如，类声明的一部分可以是计算机生成的，而另一个是手动编写的。这两者的文本分离可防止更新中的一个发生更新。

类型参数

类型参数是一个简单标识符，它表示为创建构造类型而提供的类型参数的占位符。类型参数是将稍后提供的类型的正式占位符。与此相反，类型参数（类型参数）是在创建构造类型时替换类型参数的实际类型。

```
type_parameter_list
    : '<' type_parameters '>'
    ;

type_parameters
    : attributes? type_parameter
    | type_parameters ',' attributes? type_parameter
    ;

type_parameter
    : identifier
    ;
```

类声明中的每个类型参数在该类的声明空间（声明）中定义一个名称。因此，它不能与另一个类型参数或该类中声明的成员同名。类型参数不能与类型本身具有相同的名称。

类基规范

类声明可以包含*class_base*规范，该规范定义类的直接基类和类直接实现的接口（接口）。

```
class_base
    : ':' class_type
    | ':' interface_type_list
    | ':' class_type ',' interface_type_list
    ;

interface_type_list
    : interface_type (',' interface_type)*
    ;
```

类声明中指定的基类可以是构造类类型（构造类型）。基类本身不能是类型参数，但它可以涉及范围内的类型参数。

```
class Extend<V>: V {}           // Error, type parameter used as base class
```

基类

*Class_type*包含在*class_base*中时，它将指定所声明的类的直接基类。如果类声明没有*class_base*，或者*class_base*只列出接口类型，则假定直接基类是 `object` 的。类从其直接基类继承成员，如继承中所述。

示例中

```
class A {}

class B: A {}
```

类 `A` 称为 `B` 的直接基类，`B` 称为从 `A` 派生。由于 `A` 未显式指定直接基类，因此它的直接基类被隐式 `object`。

对于构造类类型，如果在泛型类声明中指定了基类，则会通过将基类声明中的每个*type_parameter*替换为构造的类型的相应*type_argument*来获取构造类型的基类。给定泛型类声明

```
class B<U,V> {...}

class G<T>: B<string,T[]> {...}
```

构造类型的基类 `G<int>` 将 `B<string,int[]>`。

类类型的直接基类必须至少具有与类类型本身相同的可访问性（可访问域）。例如，`public` 类派生自 `private` 或 `internal` 类的编译时错误。

类类型的直接基类不得为以下类型之一：`System.Array`、`System.Delegate`、`System.MulticastDelegate`、`System.Enum` 或 `System.ValueType`。此外，泛型类声明不能将 `System.Attribute` 用作直接或间接的基类。

确定类 `B` 的直接基类规范 `A` 的含义时，`B` 的直接基类暂时假设为 `object`。这可以确保基类规范的含义无法以递归方式依赖于自身。示例：

```
class A<T> {
   public class B {}
}

class C : A<C.B> {}
```

存在错误，因为在基类规范中 `A<C.B>` `C` 的直接基类被视为 `object`，因此（由命名空间和类型名称的规则）`C` 不被视为具有成员 `B`。

类类型的基类是直接基类及其基类。换言之，基类集是直接基类关系的传递闭包。参考上面的示例，`B` 的基类 `A` 并 `object`。示例中

```
class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

`D<int>` 的基类 `C<int[]>`、`B<IComparable<int[]>>`、`A` 和 `object`。

除了类 `object`，每个类类型都有一个直接基类。`object` 类没有直接基类，是其他所有类的最终基类。

当类 `B` 从类 `A` 派生时，`A` 依赖于 `B`，则会发生编译时错误。类***直接依赖于***其直接基类（如果有），并且***直接依赖于***直接嵌套它的类（如果有）。根据此定义，类所依赖的完整类集是***直接依赖***关系的反身和可传递闭包。

示例

```
class A: A {}
```

是错误的，因为类依赖于自身。同样，示例

```
class A: B {}
class B: C {}
class C: A {}
```

出现错误，因为类循环依赖于自身。最后，示例

```
class A: B.C {}

class B: A
{
    public class C {}
}
```

导致编译时错误，这是因为 `A` 依赖于 `B.C`（其直接基类），这取决于 `B`（它的直接封闭类），后者循环依赖于 `A`。

请注意，类不依赖于嵌套在其中的类。示例中

```
class A
{
    class B: A {}
}
```

`B` 依赖于 `A`（因为 `A` 既是它的直接基类，也是它的直接封闭类），但 `A` 不依赖于 `B`（因为 `B` 既不是基类，也不是 `A` 的封闭类）。因此，该示例是有效的。

不能从 `sealed` 类派生。示例中

```
sealed class A {}

class B: A {}           // Error, cannot derive from a sealed class
```

类 `B` 出错，因为它尝试从 `sealed` 类 `A` 派生。

接口实现

*Class_base*规范可能包含一系列接口类型，在这种情况下，类可以直接实现给定的接口类型。接口实现中进一步讨论了接口实现。

类型参数约束

泛型类型和方法声明可以选择通过包含*type_parameter_constraints_clause*来指定类型参数约束。

```
type_parameter_constraints_clause
    : 'where' type_parameter ':' type_parameter_constraints
    ;

type_parameter_constraints
    : primary_constraint
    | secondary_constraints
    | constructor_constraint
    | primary_constraint ',' secondary_constraints
    | primary_constraint ',' constructor_constraint
    | secondary_constraints ',' constructor_constraint
    | primary_constraint ',' secondary_constraints ',' constructor_constraint
    ;

primary_constraint
    : class_type
    | 'class'
    | 'struct'
    ;

secondary_constraints
    : interface_type
    | type_parameter
    | secondary_constraints ',' interface_type
    | secondary_constraints ',' type_parameter
    ;

constructor_constraint
    : 'new' '(' ')'
    ;
```

每个*type_parameter_constraints_clause*都包含标记 `where`，后跟类型参数的名称，后跟一个冒号和该类型参数的约束列表。每个类型参数最多只能有一个 `where` 子句，`where` 子句可以按任意顺序列出。与 `get` 并 `set` 属性访问器中的标记一样，`where` 标记不是关键字。

`where` 子句中给定的约束列表可以包括以下任何组件（按以下顺序）：单个主约束、一个或多个辅助约束以及构造函数约束 `new()`。

主约束可以是类类型或***引用类型约束*** `class` 或 `struct` ***值类型约束***。辅助约束可以是*type_parameter*或*interface_type*。

引用类型约束指定用于类型参数的类型参数必须是引用类型。已知为引用类型的所有类类型、接口类型、委托类型、数组类型和类型参数均满足此约束。

值类型约束指定用于类型参数的类型参数必须是不可以为 null 的值类型。所有不可以为 null 的结构类型、枚举类型和具有值类型约束的类型参数均满足此约束。请注意，尽管归类为值类型，但可以为 null 的类型（可为 null 的类型）不满足值类型约束。具有值类型约束的类型形参也不能具有*constructor_constraint*。

指针类型永远不允许为类型参数，并且不被视为满足引用类型或值类型约束。

如果约束是类类型、接口类型或类型参数，则该类型指定用于该类型参数的每个类型参数都必须支持的最小 "基类型"。当使用构造类型或泛型方法时，将在编译时对照类型参数上的约束检查类型参数。提供的类型参数必须满足满足约束中所述的条件。

*Class_type*约束必须满足以下规则：

- 类型必须是类类型。
- 类型不得 `sealed`。
- 类型不得为以下类型之一：`System.Array`、`System.Delegate`、`System.Enum` 或 `System.ValueType`。
- 类型不得 `object`。由于所有类型都派生自 `object`，因此如果允许，此类约束将不起作用。
- 给定类型参数最多只能有一个约束为类类型。

指定为*interface_type*约束的类型必须满足以下规则：

- 类型必须是接口类型。
- 在给定的 `where` 子句中不能多次指定一个类型。

在任一情况下，约束都可以涉及关联的类型或方法声明的任何类型参数作为构造类型的一部分，并且可以涉及所声明的类型。

指定为类型参数约束的任何类或接口类型必须至少具有与声明的泛型类型或方法相同的可访问性（辅助功能约束）。

指定为*type_parameter*约束的类型必须满足以下规则：

- 类型必须为类型参数。
- 在给定的 `where` 子句中不能多次指定一个类型。

此外，类型参数的依赖项关系图中必须没有循环，其中依赖项是由定义的传递关系：

- 如果类型参数 `T` 用作类型参数的约束 `S` 则 `S` ***依赖于*** `T`。
- 如果类型参数 `S` 依赖于类型参数 `T` 并且 `T` 依赖于类型 `U` 参数，则 `S` ***取决于*** `U`。

在给定此关系的情况下，如果类型参数依赖于自身（直接或间接），则会发生编译时错误。

所有约束必须在依赖类型参数之间保持一致。如果类型参数 `S` 依赖于类型参数 `T` 则：

- `T` 不得具有值类型约束。否则，`T` 将被有效密封，因此 `S` 强制与 `T` 的类型相同，从而无需两个类型参数。
- 如果 `S` 具有值类型约束，则 `T` 不能具有*class_type*约束。
- 如果 `S` 具有*class_type*约束 `A` 并且 `T` 具有*class_type*约束，则必须存在从 `B` 到 `A` 或从 `B` 到 `B` 的隐式引用转换。`A`
- 如果 `S` 还依赖于类型参数 `U` 并且 `U` 具有*class_type*约束 `A` 并且 `T` 具有*class_type*的约束，则必须存在从 `B` 到 `A` 或从 `B` 到 `B` 的隐式引用转换。`A`

`S` 具有值类型约束，并且 `T` 具有引用类型约束，则有效。有效地限制 `T` `System.Object`、`System.ValueType`、`System.Enum` 和任何接口类型的类型。

如果类型参数的 `where` 子句包含构造函数约束（具有 `new()` 格式），则可以使用 `new` 运算符来创建类型的实例（对象创建表达式）。用于具有构造函数约束的类型参数的任何类型参数都必须具有一个公共的无参数构造函数（此构造函数对于任何值类型都是隐式的），或是具有值类型约束或构造函数约束的类型参数（有关详细信息，请参阅类型参数约束）。

下面是约束的示例：

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

下面的示例出错，因为它在类型参数的依赖项关系图中导致循环：

```
class Circular<S,T>
    where S: T
    where T: S                // Error, circularity in dependency graph
{
    ...
}
```

下面的示例演示了其他无效情况：

```
class Sealed<S,T>
    where S: T
    where T: struct        // Error, T is sealed
{
    ...
}

class A {...}

class B {...}

class Incompat<S,T>
    where S: A, T
    where T: B                // Error, incompatible class-type constraints
{
    ...
}

class StructWithClass<S,T,U>
    where S: struct, T
    where T: U
    where U: A                // Error, A incompatible with struct
{
    ...
}
```

类型参数 `T` 的***有效基类***定义如下：

- 如果 `T` 没有主约束或类型参数约束，则其有效基类将 `object`。
- 如果 `T` 具有值类型约束，则其有效基类将 `System.ValueType`。
- 如果 `T` *class_type*约束 `C` 但没有*type_parameter*约束，则它的有效基类是 `C` 的。
- 如果 `T` 没有*class_type*约束，但具有一个或多个*type_parameter*约束，则它的有效基类是其*type_parameter*约束的一组有效基类中包含程度最高的类型（提升转换运算符）。一致性规则确保存在这种包含最多的类型。
- 如果 `T` 同时具有*class_type*约束和一个或多个*type_parameter*约束，则它的有效基类是包含 `T` 的*class_type*约束和其*type_parameter*约束的有效基类的集中包含程度最高的类型（提升转换运算符）。一致性规则确保存在这种包含最多的类型。
- 如果 `T` 具有引用类型约束但没有*class_type*约束，则它的有效基类是 `object` 的。

对于这些规则，如果 T 的约束 `V` 是*value_type*，请改用*class_type*的最特定类型 `V` 的基类型。此操作永远不会在显式给定的约束中发生，但当通过重写方法声明或接口方法的显式实现隐式继承泛型方法时，可能会出现这种情况。

这些规则确保有效的基类始终是*class_type*。

类型参数 `T` 的***有效接口集***定义如下：

- 如果 `T` 没有*secondary_constraints*，则其有效接口集为空。
- 如果 `T` 具有*interface_type*约束但没有*type_parameter*约束，则其有效接口集为其*interface_type*约束的集合。
- 如果 `T` 没有*interface_type*约束但具有*type_parameter*约束，则其有效接口集是其*type_parameter*约束的有效接口集的并集。
- 如果 `T` 同时具有*interface_type*约束和*type_parameter*约束，则其有效接口集是其一组*interface_type*约束和其*type_parameter*约束的有效接口集的并集。

如果类型参数具有引用类型约束，或者其有效的基类不是 `object` 或 `System.ValueType`，则已知该类型参数是***引用类型***。

受约束的类型参数类型的值可用于访问约束隐含的实例成员。示例中

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

可以在 `x` 上直接调用 `IPrintable` 方法，因为 `T` 被限制为始终实现 `IPrintable`。

类体

类的*class_body*定义该类的成员。

```
class_body
    : '{' class_member_declaration* '}'
    ;
```

分部类型

类型声明可跨多个***分部类型声明***拆分。类型声明是根据此部分中的规则从其部分构造的，因此在程序的编译时和运行时处理的其余部分，将它视为单个声明。

如果*class_declaration*、*struct_declaration*或*interface_declaration*表示包含 `partial` 修饰符的分部类型声明，则为。`partial` 不是关键字，并且仅在以下情况下充当修饰符：`class`、`struct` 或 `interface` 在类型声明中，或者在方法声明中的类型 `void` 之前出现。在其他上下文中，可以将其用作常规标识符。

分部类型声明的每个部分都必须包含一个 `partial` 修饰符。它必须具有相同的名称，并在与其他部分相同的命名空间或类型声明中声明。`partial` 修饰符指示类型声明的其他部分可能存在于其他位置，但不要求存在此类附加部分;它对于具有单个声明的类型是有效的，以包含 `partial` 修饰符。

分部类型的所有部分都必须一起编译，以便可以在编译时将这些部分合并为单个类型声明。具体而言，分部类型不允许扩展已编译的类型。

嵌套类型可以在多个部分中通过使用 `partial` 修饰符来声明。通常，包含类型是使用 `partial` 进行声明的，并且嵌套类型的每个部分都是在包含类型的不同部分中声明的。

委托或枚举声明中不允许使用 `partial` 修饰符。

Attributes

分部类型的属性是通过将每个部件的属性按未指定顺序组合来确定的。如果特性放置在多个部件上，则等效于在类型上多次指定属性。例如，这两个部分：

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

等效于如下所示的声明：

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

类型参数上的属性以类似的方式进行组合。

修饰符

当分部类型声明包括辅助功能规范（`public`、`protected`、`internal` 和 `private` 修饰符）时，它必须与包括辅助功能规范的所有其他部分协商。如果分部类型的任何部分都不包括辅助功能规范，则会为该类型提供适当的默认可访问性（已声明的可访问性）。

如果嵌套类型的一个或多个分部声明包含 `new` 修饰符，则在嵌套类型隐藏继承成员（通过继承隐藏）时不会报告警告。

如果某个类的一个或多个分部声明包含 `abstract` 修饰符，则将此类视为抽象类（抽象类）。否则，类被视为非抽象类。

如果某个类的一个或多个分部声明包含 `sealed` 修饰符，则将此类视为 sealed（密封类）。否则，类被视为未密封。

请注意，类不能既是抽象的又是密封的。

当对分部类型声明使用 `unsafe` 修饰符时，只有该特定部分被视为不安全的上下文（不安全上下文）。

类型参数和约束

如果泛型类型在多个部分中声明，则每个部分都必须声明类型参数。每个部分必须具有相同数量的类型参数，并且每个类型参数的名称必须相同。

当部分泛型类型声明包含约束（`where` 子句）时，约束必须与包含约束的所有其他部分一致。具体而言，包括约束的每个部分都必须具有对同一组类型参数的约束，并且对于每个类型参数，主要、次要和构造函数约束的集必须是等效的。如果两组约束包含相同的成员，则它们是等效的。如果部分泛型类型的任何部分都不指定类型参数约束，则类型参数被视为不受约束。

示例

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

是正确的，因为包含约束的部分（前两个）为同一组类型参数有效地指定了相同的一组主、辅助和构造函数约束。

基类

当分部类声明包含基类规范时，它必须与包含基类规范的所有其他部分协商。如果分部类的任何部分都不包含基类规范，则基类成为 `System.Object`（基类）。

基接口

在多个部分中声明的类型的基接口集是在每个部件上指定的基本接口的联合。特定基接口的每个部分只能命名一次，但允许多个部件命名相同的基接口。任何给定基接口的成员只能有一个实现。

示例中

```
partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

类 `C` 的基接口集是 `IA`、`IB` 和 `IC`。

通常，每个部分都提供对该部件声明的接口的实现;但这并不是必需的。部件可以为在不同的部分声明的接口提供实现：

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}

partial class X: IComparable
{
    ...
}
```

Members

除了分部方法(分部方法)以外，在多个部分中声明的类型的成员集只是每个部分中声明的成员集的并集。类型声明的所有部分的主体共享同一声明空间(声明)，并且每个成员(范围)的作用域都扩展到所有部分的主体。任何成员的可访问域始终包括封闭类型的所有部分;在一个部分中声明的 `private` 成员可从另一个部件自由地访问。在该类型的多个部分中声明同一成员是编译时错误，除非该成员是带有 `partial` 修饰符的类型。

```
partial class A
{
    int x;                    // Error, cannot declare x more than once

    partial class Inner       // Ok, Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                    // Error, cannot declare x more than once

    partial class Inner       // Ok, Inner is a partial type
    {
        int z;
    }
}
```

类型中成员的顺序对C#代码的排序很少，但在与其他语言和环境交互时可能很重要。在这些情况下，在多个部分中声明的类型中的成员顺序是不确定的。

分部方法

分部方法可在类型声明的一部分中定义，并在另一个中实现。实现是可选的;如果没有部分实现分部方法，则将从部分组合生成的类型声明中删除分部方法声明和对它的所有调用。

分部方法不能定义访问修饰符，而是隐式 `private`。它们的返回类型必须是 `void` 的，并且其参数不能具有 `out` 修饰符。仅当标识符 `partial` 在 `void` 类型之前出现时，才会被识别为方法声明中的特殊关键字;否则，可将其用作常规标识符。分部方法不能显式实现接口方法。

存在两种类型的分部方法声明：如果方法声明的主体为分号，则声明声明为***定义分部方法声明***。如果将正文作为块提供，则声明声明为***实现分部方法声明***。在类型声明的各个部分中，只能有一个用给定的签名定义分部方法声明，只能有一个实现具有给定签名的分部方法声明。如果给定了实现分部方法声明，则必须存在相应的定义分部方法声明，并且声明必须与下面指定的声明匹配：

- 声明必须具有相同的修饰符(尽管不必按相同顺序)、方法名称、类型参数的数目和参数的数目。
- 声明中的相应参数必须具有相同的修饰符(尽管不必按相同顺序)和相同的类型(在类型参数名称中取模差异)。
- 声明中的相应类型参数必须具有相同的约束(在类型参数名称中取模不同)。

实现分部方法声明可以与相应的定义分部方法声明出现在同一部分中。

只有定义分部方法参与重载决策。因此，无论是否给定了实现声明，调用表达式都可以解析为分部方法的调用。由于分部方法始终返回 `void`，因此此类调用表达式将始终为 expression 语句。此外，由于分部方法是隐式 `private` 的，因此，此类语句将始终出现在声明分部方法的类型声明的某个部分中。

如果分部类型声明的任何部分都不包含给定分部方法的实现声明，则从组合类型声明中只会删除调用它的任何表达式语句。因此，调用表达式(包括任何构成表达式)在运行时不起作用。分部方法本身也会被移除，并且不是组合类型声明的成员。

如果给定分部方法存在实现声明，则保留分部方法的调用。分部方法提供与实现分部方法声明类似的方法声明，

但以下情况除外：

- 不包含 `partial` 修饰符
- 生成的方法声明中的特性是定义的组合特性和实现分部方法声明的未指定顺序。不会删除重复项。
- 生成的方法声明的参数上的属性为定义的相应参数的组合特性，并按未指定的顺序进行实现分部方法声明。不会删除重复项。

如果为分部方法 M 提供定义声明，而不是实现声明，则以下限制适用：

- 创建委托到方法（委托创建表达式）时出现编译时错误。
- 在转换为表达式树类型的匿名函数内引用 `M` 是编译时错误（对表达式树类型的匿名函数转换的计算）。
- 作为 `M` 调用的一部分出现的表达式不影响明确的赋值状态（明确赋值），这可能会导致编译时错误。
- `M` 不能是应用程序的入口点（应用程序启动）。

分部方法有助于允许类型声明的一部分自定义另一个部件的行为，例如，由工具生成的部分。请考虑以下分部类声明：

```
partial class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

如果此类是在没有任何其他部分的情况下编译的，则将删除定义分部方法声明及其调用，并且生成的组合类声明将等效于以下内容：

```
class Customer
{
    string name;

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

假定另外提供了一个部分，后者提供了分部方法的实现声明：

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

然后，生成的组合类声明将等效于以下内容：

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

名称绑定

尽管可扩展类型的每个部分都必须在同一个命名空间内声明，但这些部分通常是在不同的命名空间声明中编写的。因此，每个部件可能都有不同的 `using` 指令（使用指令）。解释一个部分中的简单名称（类型推理）时，仅考虑包含该部分的命名空间声明的 `using` 指令。这可能导致同一标识符在不同的部分中具有不同的含义：

```
namespace N
{
    using List = System.Collections.ArrayList;

    partial class A
    {
        List x;                // x has type System.Collections.ArrayList
    }
}

namespace N
{
    using List = Widgets.LinkedList;

    partial class A
    {
        List y;                // y has type Widgets.LinkedList
    }
}
```

类成员

类的成员由其*class_member_declaration*引入的成员和从直接基类继承的成员组成。

```
class_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | destructor_declaration
    | static_constructor_declaration
    | type_declaration
    ;
```

类类型的成员分为以下几个类别：

- 常量，表示与类（常量）关联的常量值。
- 字段，它们是类（字段）的变量。
- 方法，可实现类（方法）可以执行的计算和操作。
- 属性，定义命名特性以及与读取和写入这些特性相关联的操作。属性
- 事件，定义可由类生成的通知（事件）。
- 索引器，允许以相同方式（语法为）将类的实例索引为数组（索引器）。
- 运算符，用于定义可应用于类的实例（运算符）的表达式运算符。
- 实例构造函数，用于实现初始化类的实例所需的操作（实例构造函数）
- 析构函数，用于实现在永久丢弃类的实例之前要执行的操作（析构函数）。
- 静态构造函数，用于实现初始化类本身所需的操作（静态构造函数）。
- 类型，表示类的本地类型（嵌套类型）。

可以包含可执行代码的成员统称为类类型的*函数成员*。类类型的函数成员是类类型的方法、属性、事件、索引器、运算符、实例构造函数、析构函数和静态构造函数。

*Class_declaration*创建新的声明空间（声明），并且*class_declaration*立即包含的*class_member_declaration*会将新成员引入此声明空间。以下规则适用于*class_member_declaration*：

- 实例构造函数、析构函数和静态构造函数必须具有与直接封闭类相同的名称。所有其他成员的名称必须不同于立即封闭的类的名称。
- 常数、字段、属性、事件或类型的名称必须不同于同一个类中声明的所有其他成员的名称。
- 方法的名称必须不同于同一个类中声明的所有其他非方法的名称。此外，方法的签名（签名和重载）必须不同于同一个类中声明的所有其他方法的签名，同一类中声明的两个方法的签名可能不只是 `ref` 和 `out` 不同。
- 实例构造函数的签名必须不同于同一个类中声明的所有其他实例构造函数的签名，同一类中声明的两个构造函数的签名可能不只是 `ref` 和 `out` 不同。
- 索引器的签名必须与同一类中声明的所有其他索引器的签名不同。
- 运算符的签名必须与同一类中声明的所有其他运算符的签名不同。

类类型的继承成员（继承）不属于类的声明空间。因此，允许派生类声明与继承成员具有相同名称或签名的成员（实际上隐藏了继承成员）。

实例类型

每个类声明都具有关联的绑定类型（绑定类型和未绑定类型）和*实例类型*。对于泛型类声明，实例类型通过从类型声明创建构造类型（构造类型）来形成，其中每个提供的类型参数都是对应的类型参数。由于实例类型使用类型参数，因此只能在类型参数位于范围内时使用。也就是说，在类声明中。实例类型是在类声明中编写的代码的 `this` 的类型。对于非泛型类，实例类型只是声明的类。下面显示了多个类声明及其实例类型：

```
class A<T>                          // instance type: A<T>
{
    class B {}                      // instance type: A<T>.B
    class C<U> {}                   // instance type: A<T>.C<U>
}

class D {}                          // instance type: D
```

构造类型的成员

构造类型的非继承成员是通过将成员声明中的每个*type_parameter*替换为构造类型的对应*type_argument*来获取的。替换过程基于类型声明的语义含义，而不只是文本替换。

例如，给定泛型类声明

```
class Gen<T,U>
{
    public T[,] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

构造类型 `Gen<int[],IComparable<string>>` 具有以下成员：

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

泛型类声明中的成员 `a` 的类型 `Gen` 为 "二维数组" `T`，因此以上构造类型中的成员 `a` 类型为 "一维数组的二维数组，即 `int` " 或 " `int[,][]` "。

在实例函数成员中，`this` 的类型是包含声明的实例类型（实例类型）。

泛型类的所有成员都可以直接或作为构造类型的一部分，使用任何封闭类中的类型参数。当在运行时使用特定的封闭式构造类型（开放和闭合类型）时，会将类型形参的每个使用替换为提供给构造类型的实际类型实参。例

如：

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;

    public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);        // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);        // Prints 3.1415
    }
}
```

继承

类*继承*其直接基类类型的成员。继承意味着类隐式包含其直接基类类型的所有成员，基类的实例构造函数、析构函数和静态构造函数除外。继承的一些重要方面包括：

- 继承是可传递的。如果 `C` 派生自 `B`，并且 `B` 派生自 `A`，则 `C` 将继承在 `B` 中声明的成员以及在 `A` 中声明的成员。
- 派生类扩展其直接基类。派生类可以其继承的类添加新成员，但无法删除继承成员的定义。
- 实例构造函数、析构函数和静态构造函数不能继承，但所有其他成员均为，而不考虑它们的声明可访问性（成员访问）。但是，根据其声明的可访问性，继承成员可能无法在派生类中访问。
- 派生类可以通过使用相同的名称或签名声明新成员，*隐藏*（通过继承隐藏）继承成员。但请注意，隐藏继承成员不会删除该成员，只是使该成员直接通过派生类无法访问。
- 类的实例包含在类及其基类中声明的所有实例字段的集合，以及从派生类类型到其任何基类类型的隐式转换（隐式引用转换）。因此，可以将对某个派生类的实例的引用视为对其任何基类的实例的引用。
- 类可以声明虚拟方法、属性和索引器，派生类可以重写这些函数成员的实现。这使类能够显示多态行为，其中函数成员调用执行的操作取决于通过其调用该函数成员的实例的运行时类型。

构造类类型的继承成员是直接基类类型（基类）的成员，可通过使用构造类型的类型参数替换*class_base*规范中每个对应类型参数的匹配项。这些成员又通过替换成员声明中的每个*type_parameter*来转换*class_base*规范的相应*type_argument* 。

```
class B<U>
{
    public U F(long index) {...}
}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```

在上面的示例中，构造类型 `D<int>` 具有一个非继承成员 `public int G(string s)` 通过将类型参数 `int` 替换为类型参数 `T` 来获得。`D<int>` 还具有类声明 `B` 的继承成员。此继承成员是通过以下方式确定的：通过在基类规范 `B<T[]>` 中替换 `T` `int` 来确定 `D<int>` 的基类类型 `B<int[]>`。然后，作为 `B` 的类型参数，`int[]` 替换 `public U F(long index)` 中的 `U`，从而生成继承成员 `public int[] F(long index)`。

New 修饰符

允许*class_member_declaration*声明与继承成员具有相同名称或签名的成员。出现这种情况时，会说派生类成员*隐藏*基类成员。隐藏继承成员不被视为错误，但会导致编译器发出警告。若要禁止显示该警告，派生类成员的声明可以包含一个 `new` 修饰符，以指示该派生成员用于隐藏基成员。本主题将在通过继承隐藏中进一步进行讨论。

如果在不隐藏继承成员的声明中包括了 `new` 修饰符，则会发出对该结果的警告。删除 `new` 修饰符后，就会禁止显示此警告。

访问修饰符

*Class_member_declaration*可以具有五种可能类型的声明的可访问性（已声明的可访问性）之一：`public`、`protected internal`、`protected`、`internal` 或 `private`。除了 `protected internal` 组合以外，它是一种编译时错误，用于指定多个访问修饰符。如果*class_member_declaration*不包含任何访问修饰符，则采用 `private`。

构成类型

在成员的声明中使用的类型称为成员的构成类型。可能的构成类型为常量、字段、属性、事件或索引器的类型、方法或运算符的返回类型，以及方法、索引器、运算符或实例构造函数的参数类型。成员的构成类型必须至少与该成员本身具有相同的可访问性（辅助功能约束）。

静态成员和实例成员

类的成员为***静态成员***或***实例成员***。一般来说，将静态成员视为属于类类型和属于对象的实例成员（类类型的实例）非常有用。

当字段、方法、属性、事件、运算符或构造函数声明包含 `static` 修饰符时，它将声明一个静态成员。此外，常数或类型声明隐式声明静态成员。静态成员具有以下特征：

- 在 `E.M` 形式的*member_access*（成员访问）中引用静态成员 `M` 时，`E` 必须表示包含 `M` 的类型。`E` 表示实例，则会发生编译时错误。
- 静态字段仅标识给定封闭式类类型的所有实例共享的一个存储位置。无论给定封闭式类类型创建了多少个实例，都只有一个静态字段副本。
- 静态函数成员（方法、属性、事件、运算符或构造函数）对特定实例不起作用，并且在此类函数成员中引用 `this` 是编译时错误。

当字段、方法、属性、事件、索引器、构造函数或析构函数声明不包含 `static` 修饰符时，它将声明一个实例成员。（实例成员有时称为非静态成员。）实例成员具有以下特征：

- 在 `E.M` 形式的*member_access*（成员访问）中引用实例成员 `M` 时，`E` 必须表示包含 `M` 的类型的实例。`E` 表示类型，则是绑定时错误。
- 类的每个实例都包含该类的所有实例字段的单独集。
- 实例函数成员（方法、属性、索引器、实例构造函数或析构函数）对类的给定实例进行操作，可以将此实例作为 `this`（此访问）进行访问。

下面的示例演示用于访问静态成员和实例成员的规则：

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;            // Ok, same as this.x = 1
        y = 1;            // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;            // Error, cannot access this.x
        y = 1;            // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;          // Ok
        t.y = 1;          // Error, cannot access static member through instance
        Test.x = 1;       // Error, cannot access instance member through type
        Test.y = 1;       // Ok
    }
}
```

`F` 方法表明，在实例函数成员中，可使用*simple_name*（简单名称）来访问实例成员和静态成员。`G` 方法表明，在静态函数成员中，通过*simple_name*访问实例成员是编译时错误。`Main` 方法表明，在*member_access*（成员访问）中，必须通过实例访问实例成员，并且必须通过类型访问静态成员。

嵌套类型

在类或结构声明中声明的类型称为***嵌套类型***。在编译单元或命名空间内声明的类型称为***非嵌套类型***。

示例中

```
using System;

class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

类 `B` 是嵌套类型，因为它是在类 `A` 中声明的，而类 `A` 是一个非嵌套类型，因为它是在编译单元中声明的。

完全限定的名称

嵌套类型的完全限定名（完全限定名）为 `S.N`，其中 `S` 是声明类型 `N` 的类型的完全限定名称。

声明的可访问性

非嵌套类型可以具有 `public` 或 `internal` 声明的可访问性，并且默认情况下 `internal` 声明为可访问性。嵌套类型也可以具有这些形式的声明的可访问性，还可以包含一个或多个声明的可访问性的其他形式，具体取决于包含类型是否为类或结构：

- 在类中声明的嵌套类型可以具有五种形式的声明的可访问性（`public`、`protected internal`、`protected`、`internal` 或 `private`），与其他类成员一样，默认为 `private` 声明的可访问性。
- 在结构中声明的嵌套类型可以有三种形式的声明的可访问性（`public`、`internal` 或 `private`），与其他结构成员一样，默认情况下 `private` 声明的可访问性。

示例

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

声明一个私有嵌套类 `Node`。

遮盖

嵌套类型可以隐藏基成员(命名隐藏)。在嵌套类型声明中允许使用 `new` 修饰符，以便可以显式表达隐藏。示例

```
using System;

class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}

class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}

class Test
{
    static void Main() {
        Derived.M.F();
    }
}
```

显示隐藏 `Base` 中定义的方法 `M` 的嵌套类 `M`。

此访问

嵌套类型及其包含类型与*this_access*(此访问)没有特殊关系。具体来说，不能使用嵌套类型中 `this` 引用包含类型的实例成员。如果嵌套类型需要访问其包含类型的实例成员，则可以通过为包含类型的实例提供作为嵌套

类型的构造函数参数的 `this` 来提供访问权限。下面的示例

```
using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}
```

显示此方法。实例 `C` 创建 `Nested` 的实例，并将其自己的 `this` 传递到 `Nested` 的构造函数，以提供对 `C` 实例成员的后续访问。

访问包含类型的私有和受保护成员

嵌套类型可以访问其包含类型可以访问的所有成员，包括包含类型的成员，这些成员具有 `private` 和 `protected` 声明可访问性。示例

```
using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

class Test
{
    static void Main() {
        C.Nested.G();
    }
}
```

显示 `C` 包含嵌套类 `Nested` 的类。在 `Nested` 中，方法 `G` 调用 `C` 中定义 `F` 静态方法，`F` 具有私有声明的可访问性。

嵌套类型还可以访问在其包含类型的基类型中定义的受保护成员。示例中

```
using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}

class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();        // ok
        }
    }
}

class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}
```

嵌套类 `Derived.Nested` 通过调用 `Derived` 的实例来访问 `Derived` 的 `Base` 基类中定义的受保护的方法 `F`。

泛型类中的嵌套类型

泛型类声明可以包含嵌套类型声明。可以在嵌套类型中使用封闭类的类型参数。嵌套类型声明可以包含其他仅适用于嵌套类型的类型参数。

泛型类声明中包含的每个类型声明都是隐式的泛型类型声明。写入嵌套在泛型类型中的类型的引用时，必须将

包含构造类型(包括其类型参数)命名为。但是，从外部类中可以使用嵌套类型而无需进行限定;构造嵌套类型时，可以隐式使用外部类的实例类型。下面的示例演示了三种不同的方法来引用从 `Inner` 创建的构造类型;前两个等效项：

```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }

    static void F(T t) {
        Outer<T>.Inner<string>.F(t, "abc");      // These two statements have
        Inner<string>.F(t, "abc");               // the same effect

        Outer<int>.Inner<string>.F(3, "abc");    // This type is different

        Outer.Inner<string>.F(t, "abc");         // Error, Outer needs type arg
    }
}
```

尽管编程样式不正确，但嵌套类型中的类型参数可以隐藏在外部类型中声明的成员或类型参数：

```
class Outer<T>
{
    class Inner<T>        // Valid, hides Outer's T
    {
        public T t;       // Refers to Inner's T
    }
}
```

保留的成员名称

为了便于基础C#运行时实现，对于作为属性、事件或索引器的每个源成员声明，实现必须根据成员声明的类型、名称和类型保留两个方法签名。如果程序声明的成员的签名与这些保留签名之一匹配，则这是编译时错误，即使基础运行时实现不使用这些保留。

保留名称不会引入声明，因此它们不参与成员查找。但是，声明的关联的保留方法签名确实参与了继承(继承)，并且可使用 `new` 修饰符(新修饰符)隐藏。

保留这些名称有三个用途：

- 如果为，则允许基础实现使用普通标识符作为方法名称，以获取或设置对C#语言功能的访问权限。
- 允许其他语言使用普通标识符作为方法名称进行互操作，以获取或设置对C#语言功能的访问权限。
- 为了帮助确保由一个符合的编译器接受的源接受另一个，通过使保留成员名称的细节在所有C#实现中保持一致。

析构函数的声明(析构函数)还会导致保留签名(为析构函数保留的成员名称)。

为属性保留的成员名称

对于类型 `P` 的属性(属性 `T`)，将保留以下签名：

```
T get_P();
void set_P(T value);
```

即使属性是只读的或只写的，都保留两个签名。

示例中

```
using System;

class A
{
    public int P {
        get { return 123; }
    }
}

class B: A
{
    new public int get_P() {
        return 456;
    }

    new public void set_P(int value) {
    }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

类 `A` 定义只读属性 `P`，从而保留 `get_P` 和 `set_P` 方法的签名。类 `B` 从 `A` 派生，并隐藏这两个保留的签名。该示例生成以下输出：

```
123
123
456
```

为事件保留的成员名称

对于 `T` 委托类型的事件 `E`（事件），将保留以下签名：

```
void add_E(T handler);
void remove_E(T handler);
```

为索引器保留的成员名称

对于类型 `T` `L` 的索引器（索引器），将保留以下签名：

```
T get_Item(L);
void set_Item(L, T value);
```

即使索引器是只读的或只写的，也会保留两个签名。

而且，成员名称 `Item` 保留。

为析构函数保留的成员名称

对于包含析构函数（析构函数）的类，将保留以下签名：

```
void Finalize();
```

常量

*常量*是表示常量值的类成员：可在编译时计算的值。*Constant_declaration*引入了给定类型的一个或多个常量。

```
constant_declaration
    : attributes? constant_modifier* 'const' type constant_declarators ';'
    ;

constant_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

*Constant_declaration*可以包含一组*特性*（特性）、一个 `new` 修饰符（新修饰符）和四个访问修饰符（访问修饰符）的有效组合。特性和修饰符适用于*constant_declaration*声明的所有成员。即使常量被视为静态成员，*constant_declaration*也既不需要也不允许 `static` 修饰符。同一修饰符在常数声明中多次出现是错误的。

*Constant_declaration*的*类型*指定声明引入的成员类型。该类型后跟一个*constant_declarator*s 列表，其中每个都引入一个新成员。*Constant_declarator*由命名成员的*标识符*组成，后跟一个 "`=`" 标记，后跟一个提供成员值*constant_expression*（常数表达式）。

在常量声明中指定的*类型*必须是 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal`、`bool`、`string`、*enum_type*或*reference_type*。每个*constant_expression*必须生成一个目标类型的值，或一个可通过隐式转换转换为目标类型的类型（隐式转换）。

常数的*类型*必须至少具有与常量本身相同的可访问性（辅助功能约束）。

常量的值是使用*simple_name*（简单名称）或*member_access*（成员访问）在表达式中获取的。

常数本身可以参与*constant_expression*。因此，在需要*constant_expression*的任何构造中都可以使用常量。此类构造的示例包括 `case` 标签、`goto case` 语句、`enum` 成员声明、特性和其他常量声明。

如常数表达式中所述，*constant_expression*是可在编译时完全计算的表达式。由于不 `string` 的*reference_type*的非 null 值的唯一创建方法是应用 `new` 运算符，因此，由于*constant_expression*不允许使用 `new` 运算符，因此 reference_type 之外*的* `string` *的*常量的唯一可能值为 `null`。

如果需要常数值的符号名称，但在常数声明中不允许该值的类型，或者无法在*constant_expression*编译时计算该值，则可以改用 `readonly` 字段（Readonly 字段）。

声明多个常量的常量声明等效于多个具有相同属性、修饰符和类型的单个常量声明。例如

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

等效于

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

只要依赖项不属于循环本质，就允许常量依赖于同一个程序中的其他常量。编译器会自动排列，以适当的顺序计算常量声明。示例中

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}
```

编译器首先计算 `A.Y`，然后计算 `B.Z`，最后计算 `A.X`，并 `10`、`11` 和 `12` 生成值。常数声明可能依赖于其他程序中的常量，但这种依赖关系只能在一个方向上进行。参考上面的示例，如果在不同的程序中声明了 `A` 和 `B`，则 `A.X` 可能依赖于 `B.Z`，但 `B.Z` 随后不会同时依赖于 `A.Y`。

字段

字段是表示与对象或类关联的变量的成员。*Field_declaration*引入了给定类型的一个或多个字段。

```
field_declaration
    : attributes? field_modifier* type variable_declarators ';'
    ;

field_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'readonly'
    | 'volatile'
    | field_modifier_unsafe
    ;

variable_declarators
    : variable_declarator (',' variable_declarator)*
    ;

variable_declarator
    : identifier ('=' variable_initializer)?
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

*Field_declaration*可以包含一组*属性*(特性)、一个 `new` 修饰符(新修饰符)、四个访问修饰符(访问修饰符)和一个 `static` 修饰符(静态和实例字段)的有效组合。此外，*field_declaration*可以包括 `readonly` 修饰符(Readonly 字

段)或 `volatile` 修饰符(可变字段),但不能同时包含两者。特性和修饰符适用于*field_declaration*声明的所有成员。同一修饰符在字段声明中多次出现是错误的。

*Field_declaration*的*类型*指定声明引入的成员类型。该类型后跟一个*variable_declarator*s 列表,其中每个都引入一个新成员。*Variable_declarator*包含一个标识符,*该标识符*对成员进行命名,可选择后跟 "`=`" 标记和提供该成员的初始值的*variable_initializer* (变量初始值设定项)。

字段的*类型*必须至少与字段本身具有相同的可访问性(辅助功能约束)。

字段的值是在表达式中使用*simple_name* (简单名称)或*member_access* (成员访问)获取的。使用*赋值*(赋值运算符)修改非只读字段的值。非只读字段的值可以使用后缀增量和减量运算符(后缀递增和递减运算符)以及前缀增量和减量运算符(前缀增量和减量运算符)来获取和修改。

声明多个字段的字段声明等效于具有相同属性、修饰符和类型的单个字段的多个声明。例如

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

等效于

```
class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

静态和实例字段

当字段声明包括 `static` 修饰符时,该声明引入的字段是***静态字段***。如果不存在 `static` 修饰符,则声明引入的字段为***实例字段***。静态字段和实例字段是受支持的几种变量(变量) C#中的两种,有时它们分别被称为***静态变量***和***实例变量***。

静态字段不是特定实例的一部分;而是在已关闭类型的所有实例(开放式类型和已关闭类型)之间共享。无论已创建已关闭类类型的多少实例,都只有一个静态字段副本用于关联的应用程序域。

例如:

```
class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }

    public static int Count {
        get { return count; }
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);        // Prints 1

        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);        // Prints 1

        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);        // Prints 2
    }
}
```

实例字段属于实例。具体而言，类的每个实例都包含该类的所有实例字段的单独集。

如果在窗体 `E.M` 的*member_access*（成员访问）中引用字段，`M` 为静态字段，则 `E` 必须表示包含 `M` 的类型，如果 `M` 是实例字段，则 E 必须表示包含 `M` 的类型的实例。

静态成员和实例成员之间的差异在静态成员和实例成员中进行了进一步讨论。

只读字段

当*field_declaration*包含 `readonly` 修饰符时，由声明引入的字段为***只读字段***。向 readonly 字段的直接赋值只能作为声明的一部分出现，或者出现在同一类的实例构造函数或静态构造函数中。（Readonly 字段可以在这些上下文中指定多次。）具体而言，仅允许在以下上下文中对 `readonly` 字段进行直接赋值：

- 在引入字段的*variable_declarator*中（通过在声明中包含*variable_initializer*）。
- 对于实例字段，在包含字段声明的类的实例构造函数中;对于静态字段，在包含字段声明的类的静态构造函数中。它们也是将 `readonly` 字段作为 `out` 或 `ref` 参数传递的唯一上下文。

尝试分配到 `readonly` 字段，或在任何其他上下文中将其作为 `out` 或 `ref` 参数传递是编译时错误。

使用常量的静态 readonly 字段

如果需要常数值的符号名称，但在 `const` 声明中不允许使用值的类型，或者在编译时无法计算该值时，`static readonly` 字段非常有用。示例中

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

`Black`、`White`、`Red`、`Green`和 `Blue` 成员不能声明为 `const` 成员，因为它们的值不能在编译时进行计算。但是，将它们声明 `static readonly` 的效果大致相同。

常量和静态 readonly 字段的版本控制

常量和 readonly 字段的二进制版本控制语义不同。当表达式引用常量时，将在编译时获取常量的值，但当表达式引用 readonly 字段时，不会在运行时获取字段的值。假设应用程序包含两个单独的程序：

```
using System;

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}
```

`Program1` 和 `Program2` 命名空间表示单独编译的两个程序。由于 `Program1.Utils.X` 声明为静态只读字段，因此 `Console.WriteLine` 语句输出的值在编译时是未知的，而是在运行时获取。因此，如果更改了 `X` 的值并重新编译了 `Program1`，则 `Console.WriteLine` 语句将输出新值，即使未重新编译 `Program2` 也是如此。但是，`X` 是一个常量，则 `X` 的值将在编译 `Program2` 时获得，并将不受 `Program1` 中的更改的影响，直到重新编译 `Program2` 为止。

可变字段

当*field_declaration*包含 `volatile` 修饰符时，该声明引入的字段是***可变字段***。

对于非易失性字段，重新排序指令的优化方法可能导致意外且不可预知的结果，这会导致多线程程序在没有同步的情况下（如*lock_statement*提供的字段）访问字段（lock 语句）。这些优化可以由编译器、运行时系统或硬件执行。对于可变字段，此类重新排序优化将受到限制：

- 读取可变字段称为***易失性读取***。可变读取具有 "获取语义";也就是说，保证在对指令序列中出现其后的内存进行引用之前发生。
- 可变字段的写入称为***可变写入***。可变写入具有 "发布语义";也就是说，保证在指令序列中写入指令之前的任何

内存引用后发生。

这些限制确保所有线程观察易失性写入操作(由任何其他线程执行)时的观察顺序与写入操作的执行顺序一致。一致性实现不需要提供可变写入的单个总体顺序，如所有执行线程中所示。可变字段的类型必须是以下类型之一：

- 一个*reference_type*。
- 类型 `byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`char`、`float`、`bool`、`System.IntPtr` 或 `System.UIntPtr`。
- 具有 `byte`、`sbyte`、`short`、`ushort`、`int` 或 `uint` 的枚举基类型的*enum_type*。

示例

```
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

生成输出：

```
result = 143
```

在此示例中，方法 `Main` 启动一个新线程，该线程运行 `Thread2` 的方法。此方法将值存储到名为 `result` 的非易失性字段中，然后将 `true` 存储在可变字段 `finished` 中。主线程等待字段 `finished` 设置为 `true`，然后读取 `result` 的字段。由于已 `volatile` 声明 `finished`，因此主线程必须从字段 `result` 读取值 `143`。如果未 `volatile` 声明字段 `finished`，则允许存储区 `result` 在存储到 `finished` 后对主线程可见，因此，主线程将从该字段 `0` 中读取值 `result`。将 `finished` 声明为 `volatile` 字段可防止任何此类不一致。

字段初始化

字段的初始值，无论是静态字段还是实例字段，都是字段的类型的默认值(默认值)。在发生此默认初始化之前，不可能观察字段的值，并且字段永远不会 "未初始化"。示例

```
using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

生成输出

```
b = False, i = 0
```

因为 `b` 和 `i` 都自动初始化为默认值。

变量初始值设定项

字段声明可能包括*variable_initializer*。对于静态字段，变量初始值设定项对应于在类初始化过程中执行的赋值语句。对于实例字段，变量初始值设定项对应于创建类的实例时执行的赋值语句。

示例

```
using System;

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";

    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

生成输出

```
x = 1.4142135623731, i = 100, s = Hello
```

由于在执行实例字段初始值设定项时，会执行静态字段初始值设定项并分配给 `i` 并 `s`，因此，对 `x` 的赋值。

所有字段都将发生字段初始化中所述的默认值初始化，其中包括具有变量初始值设定项的字段。因此，初始化某个类时，该类中的所有静态字段都首先初始化为其默认值，然后以文本顺序执行静态字段初始值设定项。同样，在创建类的实例时，该实例中的所有实例字段都将首先初始化为其默认值，然后以文本顺序执行实例字段初始值设定项。

具有变量初始值设定项的静态字段可以在其默认值状态中观察到。不过，强烈建议不要使用这种样式。示例

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

展示了这种行为。尽管 a 和 b 的循环定义，但程序有效。这会导致输出

```
a = 1, b = 2
```

由于静态字段 `a` 和 `b` 会在其初始值设定项执行之前初始化为 `0`（`int` 的默认值）。当 `a` 的初始值设定项运行时，`b` 的值为零，因此 `a` 初始化为 `1`。当 `b` 的初始值设定项运行时，`a` 的值已 `1`，因此 `b` 初始化为 `2`。

静态字段初始化

类的静态字段变量初始值设定项对应于按其在类声明中出现的文本顺序执行的一系列赋值。如果类中存在静态构造函数（静态构造函数），则在执行该静态构造函数之前，会立即执行静态字段初始值设定项。否则，在首次使用该类的静态字段之前，将在依赖于实现的时间执行静态字段初始值设定项。示例

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    public static int X = Test.F("Init A");
}

class B
{
    public static int Y = Test.F("Init B");
}
```

可能产生以下输出：

```
Init A
Init B
1 1
```

或输出：

```
Init B
Init A
1 1
```

由于 `X` 的初始值设定项和 `Y` 的初始值设定项的执行可能会按以下顺序发生：它们只会在对这些字段的引用之前发生。但在示例中：

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    static A() {}

    public static int X = Test.F("Init A");
}

class B
{
    static B() {}

    public static int Y = Test.F("Init B");
}
```

输出必须为：

```
Init B
Init A
1 1
```

由于静态构造函数的执行规则（在静态构造函数中定义），因此必须在 `A` 的静态构造函数和字段初始值设定项之前运行 `B` 的静态构造函数（因此 `B` 的静态字段初始值设定项）。

实例字段初始化

类的实例字段变量初始值设定项对应于进入该类的任何一个实例构造函数（构造函数初始值设定项）后立即执行的一系列赋值。变量初始值设定项将按照它们在类声明中的显示顺序执行。实例构造函数中对类实例创建和初始化过程进行了进一步说明。

实例字段的变量初始值设定项不能引用正在创建的实例。因此，在变量初始值设定项中引用 `this` 是编译时错误，因为变量初始值设定项通过*simple_name*引用任何实例成员时出现编译时错误。示例中

```
class A
{
    int x = 1;
    int y = x + 1;        // Error, reference to instance member of this
}
```

`y` 的变量初始值设定项会导致编译时错误，因为它引用正在创建的实例的成员。

方法

方法是实现对象或类可执行的计算或操作的成员。方法使用*method_declaration*s 来声明：

```
method_declaration
    : method_header method_body
    ;

method_header
    : attributes? method_modifier* 'partial'? return_type member_name type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause*
    ;

method_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | 'async'
    | method_modifier_unsafe
    ;

return_type
    : type
    | 'void'
    ;

member_name
    : identifier
    | interface_type '.' identifier
    ;

method_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

*Method_declaration*可以包含一组*属性*(特性)和四个访问修饰符(访问修饰符)、`new`(新修饰符)、`static`(静态和实例方法)、`virtual`(虚拟方法)、`override`(重写方法)、`sealed`(密封方法)、`abstract`(抽象方法)和`extern`(外部方法)修饰符的有效组合。

如果满足以下所有条件，则声明具有有效的修饰符组合：

- 声明中包含有效的访问修饰符(访问修饰符)组合。
- 声明不包含同一修饰符多次。
- 声明最多包含以下修饰符之一：`static`、`virtual`和`override`。
- 声明最多包含以下修饰符之一：`new`和`override`。
- 如果声明包含`abstract`修饰符，则声明不包括以下任何修饰符：`static`、`virtual`、`sealed`或`extern`。
- 如果声明包含`private`修饰符，则声明不包括以下任何修饰符：`virtual`、`override`或`abstract`。
- 如果声明包含`sealed`修饰符，则声明还包括`override`修饰符。
- 如果声明包含`partial`修饰符，则不包含以下任何修饰符：`new`、`public`、`protected`、`internal`、`private`、`virtual`、`sealed`、`override`、`abstract`或`extern`。

具有 `async` 修饰符的方法是一个异步函数，并遵循异步函数中所述的规则。

方法声明的*return_type*指定由方法计算并返回的值的类型。如果方法不返回值，则 `void` *return_type* 。如果声明包含 `partial` 修饰符，则返回类型必须为 `void` 。

*Member_name*指定方法的名称。除非此方法是显式接口成员实现（显式接口成员实现），否则*member_name*只是一个*标识符*。对于显式接口成员实现，*member_name*由*interface_type*后跟 "`.`" 和*标识符*组成。

可选*type_parameter_list*指定方法的类型参数（类型参数）。如果指定了*type_parameter_list*，则该方法是***泛型方法***。如果该方法具有 `extern` 修饰符，则无法指定*type_parameter_list* 。

可选*formal_parameter_list*指定方法的参数（方法参数）。

可选*type_parameter_constraints_clause*指定对单个类型参数的约束（类型参数约束），并且仅当还提供了*type_parameter_list*并且方法没有 `override` 修饰符时，才可以指定。

在方法的*formal_parameter_list*中引用的*return_type*和每个类型必须至少具有与方法本身相同的可访问性（辅助功能约束）。

*Method_body*为分号、***语句体***或***表达式主体***。语句体由*块*组成，它指定在调用方法时要执行的语句。表达式主体由 `=>` 后跟*表达式*和分号组成，并表示在调用方法时要执行的单个表达式。

对于 `abstract` 和 `extern` 方法，*method_body*只包含一个分号。对于 `partial` 方法，*method_body*可能包含分号、块体或表达式主体。对于所有其他方法，*method_body*为块体或表达式主体。

如果*method_body*包含分号，则声明可能不包括 `async` 修饰符。

方法的名称、类型参数列表和形参列表定义方法的签名（签名和重载）。具体而言，方法的签名由其名称、类型参数的数目以及其形参的数量、修饰符和类型组成。出于这些目的，形参的类型中出现的方法的任何类型形参均由其名称标识，但其在方法的类型实参列表中的序号位置。返回类型不是方法签名的一部分，也不是类型参数或形参的名称。

方法的名称必须不同于同一个类中声明的所有其他非方法的名称。此外，方法的签名必须不同于同一个类中声明的所有其他方法的签名，同一类中声明的两个方法的签名可能不只是 `ref` 和 `out` 不同。

此方法的*type_parameter*在整个*method_declaration*的范围内，可用于在*return_type*、*method_body*和*type_parameter_constraints_clause*中的整个范围内形成类型，而不是在*属性*中。

所有形参和类型形参必须具有不同的名称。

方法参数

方法的参数（如果有）由方法的*formal_parameter_list*声明。

```
formal_parameter_list
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : 'ref'
    | 'out'
    | 'this'
    ;

parameter_array
    : attributes? 'params' array_type identifier
    ;
```

形参列表包含一个或多个以逗号分隔的参数，其中只有最后一个参数可以是*parameter_array*。

*Fixed_parameter*包含一组可选的***特性***（特性）、可选的 `ref`、`out` 或 `this` 修饰符、*类型*、*标识符*和可选的*default_argument*。每个*fixed_parameter*都声明具有给定名称的给定类型的参数。`this` 修饰符将方法指定为扩展方法，并且只允许在静态方法的第一个参数上使用。扩展方法中进一步介绍了扩展方法。

使用*default_argument*的*fixed_parameter*称为***可选参数***，而没有*default_argument*的*fixed_parameter*是***必需参数***。必需的参数不能出现在*formal_parameter_list*中的可选参数之后。

`ref` 或 `out` 参数不能具有*default_argument*。*Default_argument*中的***表达式***必须是下列其中一项：

- 一个*constant_expression*
- 格式 `new S()` 其中 `S` 为值类型的表达式
- 格式 `default(S)` 其中 `S` 为值类型的表达式

表达式必须可通过标识或可为 null 的类型转换为参数的类型。

如果在实现分部方法声明（分部方法）中出现可选参数，则显式接口成员实现（显式接口成员实现），或在单参数索引器声明（索引器）中出现警告，因为这些成员永远无法通过允许省略参数的方式进行调用。

*Parameter_array*包含一组可选的***特性***（特性）、`params` 修饰符、*array_type*和*标识符*。参数数组声明具有给定名称的给定数组类型的单一参数。参数数组的*array_type*必须为一维数组类型（数组类型）。在方法调用中，参数数组允许指定给定数组类型的单个自变量，或允许指定数组元素类型的零个或多个参数。参数数组中进一步介绍了参数数组。

*Parameter_array*可能会在可选参数之后发生，但不能具有默认值，而是省略*parameter_array*的参数会导致创建一个空数组。

下面的示例演示了不同类型的参数：

```
public void M(
    ref int      i,
    decimal      d,
    bool         b = false,
    bool?        n = false,
    string       s = "Hello",
    object       o = null,
    T            t = default(T),
    params int[] a
) { }
```

在 `M` 的*formal_parameter_list*中，`i` 是必需的 ref 参数，`d` 是必需的值参数，`b`、`s`、`o` 和 `t` 是可选值参数，`a` 是参数数组。

方法声明为参数、类型参数和局部变量创建了单独的声明空间。名称通过方法的类型形参列表和形参列表以及方法*块*中的局部变量声明引入此声明空间。方法声明空间的两个成员具有相同的名称是错误的。如果嵌套声明空间的方法声明空间和局部变量声明空间包含具有相同名称的元素，则是错误的。

方法调用（方法调用）创建特定于该方法的形参和局部变量的副本，并且调用的参数列表将值或变量引用分配给新创建的形参。在方法的*块*中，可以通过参数*Simple_name*表达式（简单名称）中的标识符引用形参。

有四种形参：

- 值参数，在没有任何修饰符的情况下声明。
- 引用参数，它们是用 `ref` 修饰符声明的。
- 输出参数，它们是用 `out` 修饰符声明的。
- 参数数组，用 `params` 修饰符声明。

如签名和重载中所述，`ref` 和 `out` 修饰符是方法签名的一部分，但是 `params` 修饰符不是。

值参数

使用不带修饰符声明的参数是一个值参数。值参数对应于一个局部变量，该局部变量从方法调用中提供的相应参数获取其初始值。

如果形参为值形参，则方法调用中的相应参数必须是可隐式转换为形参类型的表达式。

允许方法为值参数赋值。此类分配只会影响 value 参数所表示的本地存储位置，它们对方法调用中给定的实参不起作用。

引用参数

使用 `ref` 修饰符声明的参数是一个引用参数。与值参数不同，引用参数不会创建新的存储位置。相反，引用参数表示作为方法调用中的自变量提供的变量所在的存储位置。

如果形参是引用参数，则方法调用中的相应参数必须包含关键字 `ref` 后跟与形参相同的类型的*variable_reference*（确定明确赋值的确切规则）。在将变量作为引用参数传递之前，必须对其进行明确赋值。

在方法中，引用参数始终被视为明确赋值。

声明为迭代器（迭代器）的方法不能有引用参数。

示例

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

生成输出

```
i = 2, j = 1
```

对于 `Main` 中 `Swap` 的调用，`x` 表示 `i`，`y` 表示 `j`。因此，调用具有交换 `i` 和 `j` 的值的效果。

在采用引用参数的方法中，多个名称可以表示相同的存储位置。示例中

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

`G` 中的 `F` 的调用同时为 `a` 和 `b` 传递了对 `s` 的引用。因此，对于该调用，名称 `s`、`a` 和 `b` 都引用相同的存储位置，三个分配都将修改实例字段 `s`。

输出参数

使用 `out` 修饰符声明的参数是一个输出参数。与引用参数类似，output 参数不会创建新的存储位置。相反，output 参数表示作为方法调用中的自变量提供的变量所在的存储位置。

如果形参为 output 参数，则方法调用中的相应参数必须包含关键字 `out` 后跟与形参相同的类型的*variable_reference*（确定明确赋值的确切规则）。在将变量作为 output 参数传递之前，不需要明确赋值，但在将变量作为 output 参数传递的调用之后，该变量被视为明确赋值。

在方法中，就像局部变量一样，output 参数最初被视为未分配，在使用其值之前必须进行明确赋值。

方法返回之前，必须为方法的每个输出参数进行明确赋值。

声明为分部方法（分部方法）或迭代器（迭代器）的方法不能有输出参数。

输出参数通常用于生成多个返回值的方法。例如：

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

该示例生成以下输出：

```
c:\Windows\System\
hello.txt
```

请注意，可以在将 `dir` 和 `name` 变量传递到 `SplitPath` 之前取消对它们的分配，并且这些变量在调用后被视为明确赋值。

参数数组

使用 `params` 修饰符声明的参数是一个参数数组。如果形参表包含参数数组，则它必须是列表中的最后一个形参，并且必须是一维数组类型。例如，`string[]` 和 `string[][]` 的类型可用作参数数组的类型，但类型 `string[,]` 不能。不能将 `params` 修饰符与修饰符结合 `ref` 和 `out`。

参数数组允许在方法调用中使用以下两种方法之一指定参数：

- 为参数数组提供的参数可以是可隐式转换为参数数组类型的单个表达式。在这种情况下，参数数组的作用与值参数完全相同。
- 此外，调用可以为参数数组指定零个或多个参数，其中每个参数都是可隐式转换为参数数组元素类型的表达式。在这种情况下，调用会创建一个参数数组类型的实例，该实例的长度与参数的数量相对应，使用给定的参数值初始化数组实例的元素，并使用新创建的数组实例作为实际实际.

除了允许在调用中使用可变数量的自变量，参数数组完全等效于同一类型的值参数（值参数）。

示例

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

生成输出

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

`F` 的第一次调用只是将数组作为值参数传递 `a`。第二次调用 `F` 会自动创建包含给定元素值的四元素 `int[]` 并将该数组实例作为值参数进行传递。同样，第三次调用 `F` 会创建一个零元素 `int[]` 并将该实例作为值参数进行传递。第二次和第三次调用完全等效于写入：

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

在执行重载决策时，具有参数数组的方法可能以其普通形式或其展开形式（适用的函数成员）的形式应用。仅当方法的正常形式不适用，并且仅当与展开的窗体具有相同签名的适用方法未在同一类型中声明时，方法的展开形式才可用。

示例

```
using System;

class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```

生成输出

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

在此示例中，具有参数数组的方法的两个可能的展开形式都已作为常规方法包含在类中。因此，在执行重载决策时不考虑这些扩展窗体，第一个和第三个方法调用会选择常规方法。当类声明具有参数数组的方法时，也不太常见，还应将一些展开形式包含为常规方法。这样做可以避免在调用具有参数数组的方法的扩展形式时，分配发生的数组实例。

当参数数组的类型 `object[]` 时，方法的正常形式和单个 `object` 参数的所花费形式之间可能存在歧义。歧义的原因是，`object[]` 本身可隐式转换为类型 `object`。但这种歧义不会带来任何问题，因为它可以通过在需要时插入强制转换来解决。

示例

```
using System;

class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }

    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

生成输出

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

在 `F` 的第一次和最后一次调用中，`F` 的正常形式适用，因为存在从实参类型到形参类型的隐式转换（两者都是 `object[]` 类型）。因此，重载决策选择 `F` 的正常形式，并将参数作为常规值参数进行传递。在第二次和第三次调用中，`F` 的正常形式不适用，因为不存在从实参类型到形参类型的隐式转换（类型 `object` 无法隐式转换为类型 `object[]`）。但 `F` 的展开形式适用，因此重载决策选择它。因此，调用会创建一个单元素 `object[]`，并使用给定的参数值（其本身是对 `object[]` 的引用）初始化数组的单个元素。

静态和实例方法

当方法声明包含 `static` 修饰符时，该方法被称为静态方法。如果不存在 `static` 修饰符，则称该方法为实例方法。

静态方法不在特定实例上操作，并且在静态方法中引用 `this` 是编译时错误。

实例方法对类的给定实例进行操作，并且该实例可以作为 `this`（此访问）进行访问。

当 `E.M` 形式的*member_access*（成员访问）中引用方法时，如果 `M` 为静态方法，则 `E` 必须表示包含 `M` 的类型，如果 `M` 是实例方法，则 `E` 必须表示包含 `M` 的类型的实例。

静态成员和实例成员之间的差异在静态成员和实例成员中进行了进一步讨论。

虚方法

当实例方法声明包含 `virtual` 修饰符时，该方法被称为虚拟方法。如果不存在 `virtual` 修饰符，则称该方法为非虚拟方法。

非虚方法的实现是不变的，无论是在声明方法的类的实例上调用方法，还是在派生类的实例中调用方法，实现都是相同的。相反，虚方法的实现可由派生类取代。取代继承的虚方法的实现的过程称为***重写***该方法（重写方法）。

在虚方法调用中，调用所针对的实例的***运行时类型***将确定要调用的实际方法实现。在非虚拟方法调用中，实例的*编译时类型*是确定因素。确切地说，当使用参数 `A` 列表调用名为 `N` 的方法时，在编译时类型 `C` 和运行时类型 `R`（其中 `R` 是 `C` 或派生自 `C` 的类）时，将按如下所示处理调用：

- 首先，将重载决策应用于 `C`、`N` 和 `A`，以便从在中声明的方法集中选择特定方法 `M`，并通过 `C` 继承。方法调用中对此进行了说明。
- 如果 `M` 为非虚拟方法，则调用 `M`。
- 否则，`M` 是一个虚拟方法，将调用与 `R` 有关的 `M` 的派生程度最高的实现。

对于在类中声明或继承的每个虚方法，存在与该类相关的方法的***最大派生实现***。与类 `R` 相关 `M` 的虚拟方法的派生程度最高的实现如下所示：

- 如果 `R` 包含 `M` 的声明引入 `virtual`，则这是 `M` 的派生程度最高的实现。
- 否则，如果 `R` 包含 `M` 的 `override`，则这是 `M` 的派生程度最高的实现。
- 否则，与 `R` 的 `M` 的派生程度最高的实现与 `R` 的直接基类的 `M` 的派生程度相同。

下面的示例阐释了虚方法和非虚方法之间的差异：

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

在此示例中，`A` 引入了一个非虚方法 `F` 和一个虚方法 `G`。类 `B` 引入了新的非虚方法 `F`，从而隐藏了继承的 `F`，并且还会重写继承的方法 `G`。该示例生成以下输出：

```
A.F
B.F
B.G
B.G
```

请注意，语句 `a.G()` 调用 `B.G`，而不是 `A.G`。这是因为实例的运行时类型（`B`），而不是实例的编译时类型（即 `A`）确定要调用的实际方法实现。

由于允许方法隐藏继承方法，因此类可能包含多个具有相同签名的虚拟方法。这并不会造成多义性问题，因为所有派生方法都是隐藏的。示例中

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

`C` 和 `D` 类包含两个具有相同签名的虚拟方法：`A` 引入的两个虚拟方法，以及 `C` 引入的两种方法。`C` 引入的方法隐藏了继承自 `A` 的方法。因此，`D` 中的重写声明会重写 `C` 引入的方法，并且 `D` 重写 `A` 引入的方法。该示例生成以下输出：

```
B.F
B.F
D.F
D.F
```

请注意，可以通过不隐藏方法的派生程度较低的类型访问 `D` 实例，从而调用隐藏的虚拟方法。

重写方法

当实例方法声明包含 `override` 修饰符时，该方法被称为***替代方法***。重写方法使用相同的签名重写继承的虚方法。但如果虚方法声明中引入新方法，重写方法声明通过提供相应方法的新实现代码，专门针对现有的继承虚方法。

`override` 声明重写的方法称为***重写基方法***。对于在类 `C` 中声明 `M` 重写方法，重写的基方法是通过检查 `C` 的每个基类类型（从 `C` 的直接基类类型开始，并继续每个连续的直接基类类型）确定的，直到至少有一个可访问方法与替换类型参数后，该方法的签名与 `M` 具有相同的签名。为了定位重写的基方法，如果 `protected``public`，则会认为该方法是可访问的（如果它是 `protected internal` 的）；或者，如果它是在 `C` 的同一程序中 `internal` 并声明的，则认为该方法是可访问的。

除非以下所有条件都适用于重写声明，否则将发生编译时错误：

- 重写的基方法可以按上文所述进行定位。

- 正好有一个这种重写基方法。仅当基类类型为构造类型时，此限制才会生效，在这种情况下，类型参数的替换会使两个方法的签名相同。
- 重写的基方法为虚拟、抽象或重写方法。换言之，重写的基方法不能是静态的或非虚的。
- 重写的基方法不是密封的方法。
- 重写方法和重写的基方法具有相同的返回类型。
- 重写声明和重写的基方法具有相同的声明的可访问性。换言之，重写声明不能更改虚拟方法的可访问性。但是，如果重写的基方法是受保护的内部方法，并且在与包含 override 方法的程序集不同的程序集中声明，则必须保护 override 方法的声明的可访问性。
- 重写声明不指定类型参数约束子句。相反，约束是从基方法继承而来的。请注意，在已重写的方法中属于类型参数的约束可能会替换为继承约束中的类型参数。当显式指定时，这可能导致约束不合法，如值类型或密封类型。

下面的示例演示重写规则如何适用于泛型类：

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}            // Ok
    public override C<string> G() {...}         // Ok
    public override void H(C<T> x) {...}        // Error, should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}                 // Ok
    public override C<U> G() {...}              // Ok
    public override void H(C<T> x) {...}        // Error, should be C<U>
}
```

重写声明可使用*base_access*（基本访问）访问重写基方法。示例中

```
class A
{
    int x;

    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}

class B: A
{
    int y;

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

`B` 中的 `base.PrintFields()` 调用调用在 `A` 中声明的 `PrintFields` 方法。*Base_access*禁用虚拟调用机制，只是将基方法视为非虚拟方法。在 `B` 中编写调用后 `((A)this).PrintFields()`，它将以递归方式调用在 `B` 中声明的 `PrintFields` 方法，而不是在 `A` 中声明的方法，因为 `PrintFields` 是虚拟的，`((A)this)` 的运行时类型 `B`。

只有通过包含 `override` 修饰符，方法才能重写另一个方法。在所有其他情况下，具有与继承方法相同的签名的方法只是隐藏了继承方法。示例中

```
class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {}        // Warning, hiding inherited F()
}
```

`B` 中的 `F` 方法不包括 `override` 修饰符，因此不会重写 `A` 中的 `F` 方法。相反，`B` 中的 `F` 方法隐藏 `A` 中的方法，并且会报告警告，因为声明不包括 `new` 修饰符。

示例中

```
class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}        // Hides A.F within body of B
}

class C: B
{
    public override void F() {}    // Ok, overrides A.F
}
```

`B` 中的 `F` 方法隐藏了从 `A` 继承的虚拟 `F` 方法。由于 `B` 中的新 `F` 具有私有访问权限，因此其范围仅包括 `B` 的类体，不会扩展到 `C`。因此，允许在 `C` 中 `F` 的声明重写继承自 `A` 的 `F`。

密封方法

当实例方法声明包含 `sealed` 修饰符时，该方法被称为***密封方法***。如果实例方法声明包含 `sealed` 修饰符，则它还必须包括 `override` 修饰符。使用 `sealed` 修饰符可防止派生类进一步重写方法。

示例中

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

类 `B` 提供两个重写方法：具有 `sealed` 修饰符的 `F` 方法和不具有的 `G` 方法。`B` 使用密封的 `modifier` 可防止 `C` 进一步重写 `F`。

抽象方法

当实例方法声明包含 `abstract` 修饰符时，该方法被称为***抽象方法***。尽管抽象方法还隐式成为虚方法，但它不能具有修饰符 `virtual`。

抽象方法声明引入新的虚方法，但不提供该方法的实现。相反，非抽象派生类需要通过重写方法来提供自己的实现。因为抽象方法不提供实际的实现，所以抽象方法的*method_body*只包含一个分号。

抽象方法声明仅允许在抽象类（抽象类）中使用。

示例中

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

`Shape` 类定义可自行绘制的几何形状对象的抽象概念。`Paint` 方法是抽象的，因为没有有意义的默认实现。`Ellipse` 和 `Box` 类是具体的 `Shape` 实现。由于这些类是非抽象类，因此需要重写 `Paint` 方法，并提供实际的实现。

Base_access（基本访问）引用抽象方法的编译时错误。示例中

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

class B: A
{
    public override void F() {
        base.F();                       // Error, base.F is abstract
    }
}
```

为 `base.F()` 调用报告编译时错误，因为它引用了抽象方法。

允许抽象方法声明重写虚拟方法。这允许抽象类强制在派生类中重新实现方法，并使方法的原始实现不可用。示例中

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

类 `A` 声明虚方法，类 `B` 使用抽象方法重写此方法，类 `C` 重写抽象方法以提供其自己的实现。

外部方法

当方法声明包含 `extern` 修饰符时，该方法被称为***外部方法***。外部方法在外部实现，通常使用以外的语言C#。由于外部方法声明不提供实际的实现，因此外部方法的*method_body*只包含一个分号。外部方法不能是泛型的。

`extern` 修饰符通常与 `DllImport` 特性结合使用（与 COM 和 Win32 组件互操作），允许 Dll（动态链接库）实现外部方法。执行环境可能支持可提供外部方法实现的其他机制。

当外部方法包含 `DllImport` 特性时，该方法声明还必须包含 `static` 修饰符。此示例演示了 `extern` 修饰符和 `DllImport` 属性的用法：

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

分部方法（概述）

当方法声明包含 `partial` 修饰符时，该方法被称为***分部方法***。分部方法只能声明为分部类型的成员（分部类型），并且受到多种限制。分部方法中进一步介绍了分部方法。

扩展方法

如果方法的第一个参数包含 `this` 修饰符，则称该方法是***扩展方法***。扩展方法只能在非泛型非嵌套静态类中声明。扩展方法的第一个参数不能包含除 `this` 以外的任何修饰符，并且参数类型不能是指针类型。

下面是一个声明两个扩展方法的静态类示例：

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

扩展方法是一个常规静态方法。此外，在其封闭静态类处于范围内时，可以使用实例方法调用语法（扩展方法调用）来调用扩展方法，使用接收方表达式作为第一个参数。

下面的程序使用上面声明的扩展方法：

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

`Slice` 方法在 `string[]` 上可用，`ToInt32` 方法可在 `string` 上使用，因为它们已被声明为扩展方法。使用普通

的静态方法调用，程序的含义与下面相同：

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

方法主体

方法声明的*method_body*包含一个块体、一个表达式体或一个分号。

如果返回类型为 `void`，则 `void` 方法的*结果类型*；如果该方法是异步的并且返回类型为 `System.Threading.Tasks.Task`，则为。否则，非异步方法的结果类型是其返回类型，并且 `T` 具有返回类型 `System.Threading.Tasks.Task<T>` 的异步方法的结果类型。

当方法具有 `void` 结果类型和块体时，不允许在块中使用 `return` 语句（return 语句）来指定表达式。如果 void 方法的执行操作正常完成（即，控制流从方法体的结尾处），该方法只会返回到当前调用方。

如果方法具有 `void` 结果和表达式体，则表达式 `E` 必须是*statement_expression*，并且正文完全等同于窗体 `{ E; }` 的块体。

当方法具有非 void 结果类型和块体时，块中的每个 `return` 语句必须指定一个可隐式转换为结果类型的表达式。返回值的方法的块体的终结点必须是无法访问的。换言之，在带有块体的值返回方法中，不允许控件流过方法体的末尾。

如果方法具有非 void 结果类型和表达式体，则表达式必须可隐式转换为结果类型，并且正文完全等效于窗体 `{ return E; }` 的块体。

示例中

```
class A
{
    public int F() {}           // Error, return value required

    public int G() {
        return 1;
    }

    public int H(bool b) {
        if (b) {
            return 1;
        }
        else {
            return 0;
        }
    }

    public int I(bool b) => b ? 1 : 0;
}
```

返回 `F` 方法返回的值会导致编译时错误，因为控件可以流过方法体的末尾。`G` 和 `H` 方法正确，因为所有可能的执行路径都以指定返回值的 return 语句结尾。`I` 方法是正确的，因为它的主体等效于其中只包含一个返回语句的语句块。

方法重载

类型推理中介绍了方法重载决策规则。

属性

属性是提供对对象或类的特征的访问的成员。属性的示例包括字符串的长度、字体的大小、窗口的标题、客户的名称，等等。属性是字段的自然扩展，它们都是具有关联类型的命名成员，并且用于访问字段和属性的语法相同。不过，与字段不同的是，属性不指明存储位置。相反，属性包含*访问器*，用于指定在读取或写入属性值时要执行的语句。因此，属性提供了一种机制，用于将操作与对象的属性的读取和写入操作相关联;而且，它们允许计算此类属性。

属性使用*property_declaration*s 来声明：

```
property_declaration
    : attributes? property_modifier* type member_name property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | property_modifier_unsafe
    ;

property_body
    : '{' accessor_declarations '}' property_initializer?
    | '=>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;
```

*Property_declaration*可以包含一组*属性*(特性)和四个访问修饰符(访问修饰符)、`new`(新修饰符)、`static`(静态和实例方法)、`virtual`(虚拟方法)、`override`(重写方法)、`sealed`(密封方法)、`abstract`(抽象方法)和`extern`(外部方法)修饰符的有效组合。

属性声明服从与方法声明(方法)相同的规则，与修饰符的有效组合有关。

属性声明的*类型*指定声明引入的属性的类型，*member_name*指定属性的名称。除非该属性是显式接口成员实现，否则*member_name*只是一个*标识符*。对于显式接口成员实现(显式接口成员实现)，*member_name*由*interface_type*后跟 "`.`" 和*标识符*组成。

属性的*类型*必须至少具有与属性本身相同的可访问性(辅助功能约束)。

*Property_body*可以包含*访问器体*或*表达式主体*。在访问器主体中，*accessor_declarations*(必须将其括在 "`{`" 和 "`}`" 标记中)，请声明属性的访问器(访问器)。访问器指定与读取和写入属性相关联的可执行语句。

由 `=>` 后跟*表达式* `E` 并且分号完全等效于语句体 `{ get { return E; } }` 的表达式体，因此仅可用于指定仅限 getter 的属性，其中，getter 的结果由单个表达式提供。

只能为自动实现的属性(自动实现的属性)提供*property_initializer*，并导致使用*表达式*提供的值初始化此类属性的基础字段。

尽管用于访问属性的语法与字段的语法相同，但属性未分类为变量。因此，不能将属性作为 `ref` 或 `out` 参数进行传递。

如果属性声明包含 `extern` 修饰符，则该属性被称为***外部属性***。由于外部属性声明不提供实际的实现，因此它的每个*accessor_declarations*都包含一个分号。

静态属性和实例属性

如果属性声明包含 `static` 修饰符，则该属性被称为***静态属性***。如果不存在 `static` 修饰符，则将属性称为***实例属性***。

静态属性不与特定实例相关联，并且是在静态属性的访问器中引用 `this` 的编译时错误。

实例属性与类的给定实例相关联，并且该实例可以作为该属性的访问器中 `this`（此访问）进行访问。

当 `E.M` 格式的*member_access*（成员访问）中引用属性时，如果 `M` 为静态属性，则 `E` 必须表示包含 `M` 的类型，如果 `M` 是实例属性，则 E 必须表示包含 `M` 的类型的实例。

静态成员和实例成员之间的差异在静态成员和实例成员中进行了进一步讨论。

Remove

属性的*accessor_declarations*指定与读取和写入属性相关联的可执行语句。

```
accessor_declarations
    : get_accessor_declaration set_accessor_declaration?
    | set_accessor_declaration get_accessor_declaration?
    ;

get_accessor_declaration
    : attributes? accessor_modifier? 'get' accessor_body
    ;

set_accessor_declaration
    : attributes? accessor_modifier? 'set' accessor_body
    ;

accessor_modifier
    : 'protected'
    | 'internal'
    | 'private'
    | 'protected' 'internal'
    | 'internal' 'protected'
    ;

accessor_body
    : block
    | ';'
    ;
```

访问器声明由*get_accessor_declaration*和/或*set_accessor_declaration*组成。每个访问器声明都包含标记 `get` 或 `set` 后跟可选的*accessor_modifier*和*accessor_body*。

*Accessor_modifier*的使用受以下限制的制约：

- *Accessor_modifier*不能在接口或显式接口成员实现中使用。
- 对于没有 `override` 修饰符的属性或索引器，仅当属性或索引器同时具有 `get` 和 `set` 访问器时，才允许*accessor_modifier*，然后只允许在其中一种访问器上使用。
- 对于包含 `override` 修饰符的属性或索引器，访问器必须与要重写的访问器的*accessor_modifier*（如果有）匹配。
- *Accessor_modifier*必须声明一个可访问性，严格比属性或索引器本身的声明的可访问性更严格。精确：
  - 如果属性或索引器具有 `public` 的声明的可访问性，则*accessor_modifier*可能是 `protected internal`、`internal`、`protected` 或 `private`。
  - 如果属性或索引器具有 `protected internal` 的声明的可访问性，则*accessor_modifier*可能是 `internal`

、`protected` 或 `private` 。
    - 如果属性或索引器具有 `internal` 或 `protected` 的声明的可访问性，则*accessor_modifier*必须 `private` 。
    - 如果属性或索引器具有 `private` 的声明的可访问性，则不能使用任何*accessor_modifier* 。

对于 `abstract` 和 `extern` 属性，所指定的每个访问器的*accessor_body*只是一个分号。非抽象的非 extern 属性可能具有每个*accessor_body*都是一个分号，在这种情况下，它是一个***自动实现的属性***（自动实现的属性）。自动实现的属性必须至少具有 get 访问器。对于任何其他非抽象非 extern 属性的访问器，*accessor_body*是一个*块*，它指定在调用相应的访问器时要执行的语句。

`get` 访问器与具有属性类型返回值的无参数方法相对应。除了作为赋值的目标以外，在表达式中引用属性时，将调用属性的 `get` 访问器来计算属性的值（表达式的值）。`get` 访问器的主体必须符合方法体中所述的返回值的方法的规则。特别是，`get` 访问器体中的所有 `return` 语句都必须指定一个可隐式转换为属性类型的表达式。此外，`get` 访问器的终结点必须是无法访问的。

`set` 访问器对应于一个方法，该方法具有属性类型的单个值参数和一个 `void` 返回类型。`set` 访问器的隐式参数始终命名为 `value` 。当属性作为赋值的目标（赋值运算符）进行引用时，或者作为 `++` 或 `--` （后缀增量和减量运算符、前缀增量和减量运算符）的操作数，使用参数（其值是赋值的右侧或 `++` 或 `--` 运算符的操作数）提供新值（简单赋值）来调用 `set` 访问器。`set` 访问器的主体必须符合方法体中所述的 `void` 方法的规则。特别是，不允许 `set` 访问器正文中的 `return` 语句指定表达式。由于 `set` 访问器隐式包含一个名为 `value` 的参数，因此，`set` 访问器中的局部变量或常量声明的编译时错误是具有该名称。

根据 `get` 和 `set` 访问器的存在情况，将按如下方式对属性进行分类：

- 同时包含 `get` 访问器和 `set` 访问器的属性称为*读写*属性。
- 仅具有 `get` 访问器的属性称为***只读***属性。只读属性是赋值的目标时，这是一个编译时错误。
- 仅具有 `set` 访问器的属性称为***只写***属性。除非作为赋值的目标，否则在表达式中引用只写属性是编译时错误。

示例中

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

`Button` 控件声明公共 `Caption` 属性。`Caption` 属性的 `get` 访问器返回私有 `caption` 字段中存储的字符串。`set` 访问器检查新值是否不同于当前值，如果是，则它存储新值并重新绘制控件。属性通常遵循上面所示的模式：`get` 访问器只返回一个存储在私有字段中的值，而 `set` 访问器则修改该私有字段，然后执行完全更新对象状态所需的任何其他操作。

假设上述 `Button` 类，以下是使用 `Caption` 属性的示例：

```
Button okButton = new Button();
okButton.Caption = "OK";            // Invokes set accessor
string s = okButton.Caption;        // Invokes get accessor
```

此处，通过将值分配给属性来调用 `set` 访问器，通过引用表达式中的属性来调用 `get` 访问器。

属性的 `get` 和 `set` 访问器不是不同的成员，并且不能单独声明属性的访问器。因此，读写属性的两个访问器不可能具有不同的可访问性。示例

```
class A
{
    private string name;

    public string Name {                // Error, duplicate member name
        get { return name; }
    }

    public string Name {                // Error, duplicate member name
        set { name = value; }
    }
}
```

不声明单个读写属性。相反，它声明了两个具有相同名称的属性，一个为只读，一个只写。由于在同一类中声明的两个成员不能具有相同的名称，因此该示例将导致发生编译时错误。

当派生类用与继承属性相同的名称声明属性时，派生属性将隐藏与读取和写入相关的继承属性。示例中

```
class A
{
    public int P {
        set {...}
    }
}

class B: A
{
    new public int P {
        get {...}
    }
}
```

`B` 中的 `P` 属性隐藏 `A` 中的 `P` 属性，与读取和写入有关。因此，在语句中

```
B b = new B();
b.P = 1;          // Error, B.P is read-only
((A)b).P = 1;     // Ok, reference to A.P
```

分配到 `b.P` 会导致报告编译时错误，因为 `B` 中的只读 `P` 属性将隐藏 `A` 中的只写 `P` 属性。但请注意，强制转换可用于访问隐藏的 `P` 属性。

与公共字段不同，属性可在对象的内部状态和其公共接口之间提供分隔。请看下面的示例：

```
class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}
```

此处，`Label` 类使用两个 `int` 字段 `x` 和 `y` 来存储其位置。该位置公开为 `X` 和 `Y` 属性，并作为 `Point` 类型的 `Location` 属性公开。如果在 `Label` 的未来版本中，在内部将位置存储为 `Point` 会更方便，而不会影响类的公共接口，则可以进行更改：

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}
```

`x` 和 `y` `public readonly` 字段，则不可能对 `Label` 类进行此类更改。

通过属性公开状态不一定比直接公开字段的效率低。特别是，当某个属性是非虚拟的并且只包含少量的代码时，执行环境可能会将对访问器的调用替换为访问器的实际代码。此过程称为***内联***，它使属性访问与字段访问一样

高效，同时还保留了属性的灵活性。

由于调用 `get` 访问器在概念上等同于读取字段的值，因此 `get` 访问器的编程样式被视为不正确的编程样式，使其具有可观察的副作用。示例中

```
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```

`Next` 属性的值取决于属性以前被访问的次数。因此，访问属性会产生一个可观察到的副作用，而属性应作为方法实现。

`get` 访问器的 "无副作用" 约定并不意味着应始终将 `get` 访问器编写为仅返回存储在字段中的值。事实上，`get` 访问器通常通过访问多个字段或调用方法来计算属性的值。但是，正确设计的 `get` 访问器不会执行任何操作，从而导致对象的状态发生变化。

在第一次引用资源之前，可以使用属性来延迟资源的初始化。例如：

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }

    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}
```

`Console` 类包含三个属性：`In`、`Out` 和 `Error`，分别表示标准输入、输出和错误设备。通过将这些成员作为属性公开，`Console` 类可以延迟其初始化，直到实际使用它们。例如，在第一次引用 `Out` 属性时，如下所示

```
Console.Out.WriteLine("hello, world");
```

将创建输出设备的基础 `TextWriter`。但如果应用程序不引用 `In` 和 `Error` 属性，则不会为这些设备创建任何对象。

自动实现的属性

自动实现的属性(short 的***自动属性***)是具有仅包含分号的访问器正文的非抽象非 extern 属性。自动属性必须具有 get 访问器，并且可以选择包含 set 访问器。

当属性指定为自动实现的属性时，隐藏的支持字段将自动提供给该属性，并实现访问器来读取和写入该支持字段。如果 auto 属性没有 set 访问器，则会将支持字段视为 `readonly`（Readonly 字段）。与 `readonly` 字段一样，仅 getter 自动属性也可在封闭类的构造函数的主体中分配给。此类赋值直接分配给属性的 readonly 支持字段。

自动属性可以有选择地具有*property_initializer*，该属性作为*variable_initializer*（变量初始值设定项）直接应用于支持字段。

如下示例中：

```
public class Point {
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

等效于以下声明：

```
public class Point {
    private int __x = 0;
    private int __y = 0;
    public int X { get { return __x; } set { __x = value; } }
    public int Y { get { return __y; } set { __y = value; } }
}
```

如下示例中：

```
public class ReadOnlyPoint
{
    public int X { get; }
    public int Y { get; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

等效于以下声明：

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

请注意，只读字段的赋值是合法的，因为它们发生在构造函数中。

辅助功能

如果访问器具有*accessor_modifier*，则访问器的可访问性域（可访问域）是使用*accessor_modifier*的声明的可访问

性确定的。如果访问器没有*accessor_modifier*，则访问器的可访问性域由属性或索引器的声明的可访问性决定。

*Accessor_modifier*的存在从不影响成员查找（运算符）或重载决策（重载决策）。属性或索引器的修饰符始终决定绑定到的属性或索引器，而不考虑访问的上下文。

选择特定属性或索引器后，所涉及的特定访问器的可访问域将用于确定此使用是否有效：

- 如果用法为值（表达式的值），则 `get` 访问器必须存在并且可访问。
- 如果使用情况是作为简单赋值的目标（简单分配），则 `set` 访问器必须存在并且可访问。
- 如果使用情况是作为复合赋值（复合赋值）的目标，或者作为 `++` 或 `--` 运算符（函数成员.9、调用表达式）的目标，则 `get` 访问器和 `set` 访问器都必须存在并且可访问。

在下面的示例中，属性 `B.Text``A.Text` 隐藏属性，即使仅在调用 `set` 访问器的上下文中也是如此。相反，类 `M` 无法访问属性 `B.Count`，因此改用 `A.Count` 的可访问属性。

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
    }
}

class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;            // Calls A.Count set accessor
        int i = b.Count;         // Calls A.Count get accessor
        b.Text = "howdy";        // Error, B.Text set accessor not accessible
        string s = b.Text;       // Calls B.Text get accessor
    }
}
```

用于实现接口的访问器可能没有*accessor_modifier*。如果只使用一个访问器来实现接口，则可以使用*accessor_modifier*声明另一个访问器：

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
    public string Prop {
        get { return "April"; }       // Must not have a modifier here
        internal set {...}            // Ok, because I.Prop has no set accessor
    }
}
```

虚拟、密封、重写和抽象属性访问器

`virtual` 属性声明指定属性的访问器是虚拟的。`virtual` 修饰符适用于读写属性的这两个访问器，即读写属性的一个取值函数是虚拟的。

`abstract` 属性声明指定属性的访问器是虚拟的，但是不提供访问器的实际实现。相反，非抽象派生类需要通过重写属性来为访问器提供自己的实现。因为抽象属性声明的访问器不提供实际的实现，所以其*accessor_body*只包含一个分号。

同时包含 `abstract` 和 `override` 修饰符的属性声明指定属性是抽象的并且会重写基属性。此类属性的访问器也是抽象的。

仅在抽象类(抽象类)中允许抽象属性声明。通过包括指定 `override` 指令的属性声明，可在派生类中重写继承的虚拟属性的访问器。这称为 "***重写*" *属性声明***。重写属性声明未声明新的属性。相反，它只是专用化现有虚拟属性的访问器的实现。

重写的属性声明必须将完全相同的可访问性修饰符、类型和名称指定为继承的属性。如果继承的属性只有一个访问器(即，如果继承的属性为只读或只写)，则重写属性必须仅包含该访问器。如果继承的属性包含两个访问器(即，如果继承的属性是读写的)，则重写属性可以包含单个访问器或两个访问器。

重写的属性声明可以包括 `sealed` 修饰符。使用此修饰符可防止派生类进一步重写属性。密封属性的访问器也是密封的。

除了声明和调用语法中的差异外，virtual、sealed、override 和 abstract 访问器的行为与虚拟、密封、重写和抽象方法完全相同。具体而言，在虚拟方法、替代方法、密封方法和抽象方法中描述的规则将应用为：访问器是相应形式的方法：

- `get` 访问器对应于一个无参数方法，该方法具有属性类型的返回值以及与包含属性相同的修饰符。
- `set` 访问器对应于一个方法，该方法具有属性类型的单个值参数、一个 `void` 返回类型，以及与包含属性相同的修饰符。

示例中

```
abstract class A
{
    int y;

    public virtual int X {
        get { return 0; }
    }

    public virtual int Y {
        get { return y; }
        set { y = value; }
    }

    public abstract int Z { get; set; }
}
```

`X` 是虚拟的只读属性，`Y` 为虚拟读写属性，`Z` 是抽象的读写属性。由于 `Z` 是抽象的，因此，包含类 `A` 也必须声明为抽象类。

下面显示了一个派生自 `A` 的类：

```
class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}
```

此处，`X`、`Y` 和 `Z` 的声明都是重写属性声明。每个属性声明都与相应的继承属性的可访问性修饰符、类型和名称完全匹配。`X` 的 `get` 访问器和 `Y` 的 `set` 访问器使用 `base` 关键字访问继承的访问器。`Z` 的声明会重写抽象访问器，因此，`B` 中没有未完成的抽象函数成员，并且允许 `B` 为非抽象类。

当某个属性声明为 `override` 时，重写的代码必须能够访问所有重写的访问器。此外，属性或索引器本身以及访问器的声明的可访问性必须与重写的成员和访问器的可访问性匹配。例如：

```
public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}            // Must specify protected here
        get {...}                      // Must not have a modifier here
    }
}
```

事件

事件是允许对象或类提供通知的成员。客户端可以通过提供***事件处理程序***来附加事件的可执行代码。

使用*event_declaration*s 声明事件：

```
event_declaration
    : attributes? event_modifier* 'event' type variable_declarators ';'
    | attributes? event_modifier* 'event' type member_name '{' event_accessor_declarations '}'
    ;

event_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | event_modifier_unsafe
    ;

event_accessor_declarations
    : add_accessor_declaration remove_accessor_declaration
    | remove_accessor_declaration add_accessor_declaration
    ;

add_accessor_declaration
    : attributes? 'add' block
    ;

remove_accessor_declaration
    : attributes? 'remove' block
    ;
```

*Event_declaration*可能包括一组*属性*（特性）和四个访问修饰符（访问修饰符）、`new`（新修饰符）、`static`（静态和实例方法）、`virtual`（虚拟方法）、`override`（重写方法）、`sealed`（密封方法）、`abstract`（抽象方法）和`extern`（外部方法）修饰符的有效组合。

事件声明遵循与方法声明（方法）相同的规则，与修饰符的有效组合有关。

事件声明的*类型*必须是*delegate_type*（引用类型），并且*delegate_type*必须至少与事件本身具有相同的可访问性（可访问性约束）。

事件声明可能包括*event_accessor_declarations*。但是，如果不是，则对于非 extern 非抽象事件，编译器会自动提供这些事件（类似于字段的事件）;对于 extern 事件，访问器是在外部提供的。

*Event_accessor_declarations*省略的事件声明定义了一个或多个事件，每个*variable_declarator*一个事件。特性和修饰符适用于由此类*event_declaration*声明的所有成员。

*Event_declaration*包括`abstract`修饰符和用大括号分隔的*event_accessor_declarations*，则会发生编译时错误。

当事件声明包含`extern`修饰符时，该事件被称为***外部事件***。由于外部事件声明不提供实际的实现，因此它可以同时包含`extern`修饰符和*event_accessor_declarations*。

使用`abstract`或`external`修饰符来包含*variable_initializer*时，事件声明的*variable_declarator*会出现编译时错误。

事件可用作`+=`和`-=`运算符（事件分配）的左操作数。这些运算符分别用于将事件处理程序附加到事件处理程

序或从事件中删除事件处理程序，事件的访问修饰符控制允许此类操作的上下文。

由于 `+=` 和 `-=` 是对声明事件的类型之外的事件允许的唯一操作，因此外部代码可以添加和移除事件的处理程序，但不能以任何其他方式获取或修改事件处理程序的基础列表。

在 `x += y` 或 `x -= y` 形式的操作中，当 `x` 为事件并且引用出现在包含 `x` 声明的类型之外时，操作的结果将具有类型 `void`（与具有 `x` 的类型相反，而在赋值后，`x` 的值为。此规则禁止外部代码间接检查事件的基础委托。

下面的示例演示如何将事件处理程序附加到 `Button` 类的实例：

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

此处，`LoginDialog` 实例构造函数创建两个 `Button` 实例，并将事件处理程序附加到 `Click` 事件。

类似字段的事件

在包含事件声明的类或结构的程序文本中，某些事件可以像字段一样使用。若要以这种方式使用，事件不得 `abstract` 或 `extern`，而且不能显式包含*event_accessor_declarations*。此类事件可用于任何允许字段的上下文中。该字段包含一个委托(委托)，表示已添加到事件的事件处理程序的列表。如果未添加任何事件处理程序，则字段将包含 `null`。

示例中

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}
```

`Click` 用作 `Button` 类中的字段。如示例所示，可以检查、修改字段，并将其用于委托调用表达式。`Button` 类中的 `OnClick` 方法 "引发" `Click` 事件。引发事件的概念恰恰等同于调用由事件表示的委托，因此，没有用于引发事件的特殊语言构造。请注意，委托调用前面有一个检查，该检查可确保委托为非 null。

在 `Button` 类的声明外部，`Click` 成员只能在 `+=` 和 `-=` 运算符的左侧使用，如下所示

```
b.Click += new EventHandler(...);
```

它将一个委托追加到 `Click` 事件的调用列表中，并

```
b.Click -= new EventHandler(...);
```

这将从 `Click` 事件的调用列表中移除委托。

在编译类似于字段的事件时，编译器会自动创建存储来保存委托，并为事件创建访问器，以便在委托字段中添加或删除事件处理程序。添加和移除操作是线程安全的，并且在持有实例事件的包含对象的锁(lock 语句)或静态事件的类型对象(匿名对象创建表达式)的情况下，可能(但不需要)执行此操作。

因此，以下形式的实例事件声明：

```
class X
{
    public event D Ev;
}
```

将编译为与等效的对象：

```
class X
{
    private D __Ev;  // field to hold the delegate

    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }

        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}
```

在类 `X` 中，对 `+=` 和 `-=` 运算符左侧 `Ev` 的引用会导致调用 add 和 remove 访问器。对 `Ev` 的所有其他引用将被编译为引用隐藏字段(成员访问) `__Ev`。名称 "`__Ev`" 是任意的;隐藏的字段可能有任何名称，或者根本没有名称。

事件访问器

事件声明通常会忽略*event_accessor_declarations*，如上面的 `Button` 示例中所示。执行此操作的一种情况是，这种情况涉及到每个事件的一个字段的存储成本是不可接受的。在这种情况下，类可以包括*event_accessor_declarations*，并使用专用机制来存储事件处理程序列表。

事件的*event_accessor_declarations*指定与添加和移除事件处理程序相关联的可执行语句。

访问器声明由*add_accessor_declaration*和*remove_accessor_declaration*组成。每个访问器声明都包含标记 `add` 或 `remove` 后跟一个*块*。与*add_accessor_declaration*关联的*块*指定在添加事件处理程序时要执行的语句，与*remove_accessor_declaration*关联的*块*指定在删除事件处理程序时要执行的语句。

每个*add_accessor_declaration*和*remove_accessor_declaration*对应于一个方法，其中包含事件类型的单个值参数和一个 `void` 返回类型。事件访问器的隐式参数名为 `value`。在事件分配中使用事件时，将使用适当的事件访问器。具体而言，如果 `+=` 赋值运算符，则使用 add 访问器，如果赋值运算符 `-=`，则使用 remove 访问器。在任一情况下，赋值运算符的右操作数用作事件访问器的参数。
*Add_accessor_declaration*或*remove_accessor_declaration*的块必须符合方法体中所述的 `void` 方法的规则。特别是，不允许此类块中的 `return` 语句指定表达式。

由于事件访问器隐式具有名为 `value` 的参数，因此在事件访问器中声明的局部变量或常量具有该名称是编译时错误。

示例中

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }

    // Invoke the MouseUp event
    protected void OnMouseUp(MouseEventArgs args) {
        MouseEventHandler handler;
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        if (handler != null)
            handler(this, args);
    }
}
```

`Control` 类实现事件的内部存储机制。`AddEventHandler` 方法将委托值与某个键相关联，`GetEventHandler` 方法返回当前与某个键相关联的委托，并且 `RemoveEventHandler` 方法将委托作为指定事件的事件处理程序移除。可能的基础存储机制的设计使 `null` 委托值与密钥关联不会产生费用，因此未处理的事件不会消耗任何存储。

静态和实例事件

当事件声明包含 `static` 修饰符时，该事件被称为***静态事件***。如果不存在任何 `static` 修饰符，则该事件被称为***实例事件***。

静态事件不与特定实例相关联，并且是在静态事件的访问器中引用 `this` 的编译时错误。

实例事件与类的给定实例相关联，并且可以在该事件的访问器中将此实例作为 `this`（此访问）进行访问。

当 `E.M` 形式的*member_access*（成员访问）中引用事件时，如果 `M` 为静态事件，则 `E` 必须表示包含 `M` 的类型，如果 `M` 是实例事件，则 E 必须表示包含 `M` 的类型的实例。

静态成员和实例成员之间的差异在静态成员和实例成员中进行了进一步讨论。

虚拟、密封、重写和抽象事件访问器

`virtual` 事件声明指定该事件的访问器是虚拟的。`virtual` 修饰符适用于事件的两个访问器。

`abstract` 事件声明指定事件的访问器是虚拟的，但不提供访问器的实际实现。相反，非抽象派生类需要通过重写事件为访问器提供自己的实现。因为抽象事件声明不提供实际的实现，所以它不能提供用括号分隔的*event_accessor_declarations*。

同时包含 `abstract` 和 `override` 修饰符的事件声明指定事件是抽象的并且会重写基事件。此类事件的访问器也是抽象的。

仅在抽象类(抽象类)中允许抽象事件声明。

通过包括指定 `override` 修饰符的事件声明，可在派生类中重写继承的虚拟事件的访问器。这称为***重写事件声明***。重写事件声明未声明新事件。相反，它只是专用化现有虚拟事件的访问器的实现。

重写事件声明必须指定与重写事件完全相同的可访问性修饰符、类型和名称。

重写事件声明可能包括 `sealed` 修饰符。使用此修饰符可防止派生类进一步重写事件。密封事件的访问器也是密封的。

重写事件声明包含 `new` 修饰符是编译时错误。

除了声明和调用语法中的差异外，virtual、sealed、override 和 abstract 访问器的行为与虚拟、密封、重写和抽象方法完全相同。具体而言，在虚拟方法、重写方法、密封方法和抽象方法中描述的规则就像访问器是相应窗体的方法一样。每个访问器都对应于一个方法，该方法具有事件类型的单个值参数、一个 `void` 返回类型，以及与包含事件相同的修饰符。

索引器

索引器是一个成员，使对象能够以与数组相同的方式进行索引。使用*indexer_declaration*s 声明索引器：

```
indexer_declaration
    : attributes? indexer_modifier* indexer_declarator indexer_body
    ;

indexer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | indexer_modifier_unsafe
    ;

indexer_declarator
    : type 'this' '[' formal_parameter_list ']'
    | type interface_type '.' 'this' '[' formal_parameter_list ']'
    ;

indexer_body
    : '{' accessor_declarations '}'
    | '=>' expression ';'
    ;
```

*Indexer_declaration*可能包括一组*特性*(特性)和四个访问修饰符(访问修饰符)、`new`(新修饰符)、`virtual`(虚拟方法)、`override`(重写方法)、`sealed`(密封方法)、`abstract`(抽象方法)和`extern`(外部方法)修饰符的有效组合。

索引器声明遵循与方法声明(方法)相同的规则,这些修饰符与修饰符的有效组合有关,但有一个例外,即索引器声明上不允许使用 static 修饰符。

除了一种情况外,修饰符`virtual`、`override`和`abstract`都是互斥的。`abstract`和`override`修饰符可一起使用,以便抽象索引器可以重写虚拟一个。

索引器声明的*类型*指定声明引入的索引器的元素类型。除非索引器是显式接口成员实现,否则,该*类型*后跟关键字`this`。对于显式接口成员实现,该*类型*后跟一个*interface_type*、"`.`"和关键字"`this`"。与其他成员不同,索引器没有用户定义的名称。

*Formal_parameter_list*指定索引器的参数。索引器的形参列表与方法(方法参数)的形参列表相同,不同之处在于至少必须指定一个形参,并且不允许`ref`和`out`参数修饰符。

索引器的*类型*和*formal_parameter_list*中引用的每个类型必须至少与索引器本身相同(可访问性约束)。

*Indexer_body*可以包含***访问器体***或***表达式主体***。在访问器主体中,*accessor_declarations*(必须将其括在"`{`"和"`}`"标记中),请声明属性的访问器(访问器)。访问器指定与读取和写入属性相关联的可执行语句。

由"`=>`"后跟表达式`E`并且分号完全等效于语句体`{ get { return E; } }`的表达式体,因此仅用于指定仅限 getter 的索引器,其中,getter 的结果由单个表达式提供。

尽管访问索引器元素的语法与数组元素的语法相同,但不会将索引器元素分类为变量。因此,不能将索引器元素作为`ref`或`out`参数进行传递。

索引器的形参列表定义索引器的签名(签名和重载)。具体而言,索引器的签名由其形参的数量和类型组成。形参的元素类型和名称不属于索引器的签名。

索引器的签名必须与同一类中声明的所有其他索引器的签名不同。

索引器和属性在概念上非常类似,但在以下方面有所不同:

- 属性由其名称标识,而索引器由其签名标识。
- 属性可通过*simple_name*(简单名称)或*member_access*(成员访问)访问,而索引器元素则通过*element_access*(索引器访问)访问。
- 属性可以是`static`成员,而索引器始终是实例成员。
- 属性的`get`访问器对应于不带参数的方法,而索引器的`get`访问器对应于具有与索引器相同的形参列表的方法。
- 属性的`set`访问器对应于一个方法,该方法具有一个名为`value`的参数,而索引器的`set`访问器对应于一个方法,该方法具有与索引器相同的形参列表,另外还有一个名为`value`的附加形参。
- 索引器访问器使用与索引器参数相同的名称声明局部变量是编译时错误。
- 在重写属性声明中,使用`base.P`语法访问继承属性,其中`P`是属性名称。在重写索引器声明中,继承的索引器是使用语法`base[E]`访问的,其中`E`是逗号分隔的表达式列表。
- 没有"自动实现的索引器"的概念。使用具有分号访问器的非抽象非外部索引器是错误的。

除了这些差异以外,在访问器和自动实现的属性中定义的所有规则都适用于索引器访问器以及属性访问器。

当索引器声明包含`extern`修饰符时,该索引器被称为***外部索引器***。由于外部索引器声明不提供实际的实现,因此它的每个*accessor_declarations*都包含一个分号。

下面的示例声明一个`BitArray`类,该类实现索引器以访问位数组中的单个位。

```
using System;

class BitArray
{
    int[] bits;
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```

`BitArray` 类的实例使用的内存量明显小于相应的 `bool[]`（因为前者的每个值仅占用一位而不是后者的一个字节），但它允许与 `bool[]` 相同的操作。

以下 `CountPrimes` 类使用 `BitArray` 和传统的 "埃拉托色" 算法来计算介于1和 a 之间的 primes：

```
class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}
```

请注意，用于访问 `BitArray` 的元素的语法与 `bool[]` 的语法完全相同。

下面的示例显示一个 26 * 10 网格类，该类具有一个具有两个参数的索引器。第一个参数必须是 A-z 范围内的大写或小写字母，第二个参数必须是0-9 范围内的整数。

```
using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}
```

索引器重载

类型推理中介绍了索引器重载决策规则。

运算符

运算符是一个成员，用于定义可应用于类的实例的表达式运算符的含义。使用*operator_declaration*s 声明运算符：

```
operator_declaration
    : attributes? operator_modifier+ operator_declarator operator_body
    ;

operator_modifier
    : 'public'
    | 'static'
    | 'extern'
    | operator_modifier_unsafe
    ;

operator_declarator
    : unary_operator_declarator
    | binary_operator_declarator
    | conversion_operator_declarator
    ;

unary_operator_declarator
    : type 'operator' overloadable_unary_operator '(' type identifier ')'
    ;

overloadable_unary_operator
    : '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
    ;

binary_operator_declarator
    : type 'operator' overloadable_binary_operator '(' type identifier ',' type identifier ')'
    ;

overloadable_binary_operator
    : '+'   | '-'   | '*'   | '/'   | '%'   | '&'   | '|'   | '^'   | '<<'
    | right_shift | '=='  | '!='  | '>'   | '<'   | '>='  | '<='
    ;

conversion_operator_declarator
    : 'implicit' 'operator' type '(' type identifier ')'
    | 'explicit' 'operator' type '(' type identifier ')'
    ;

operator_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

有三种类型的可重载运算符：一元运算符（一元运算符）、二元运算符（二元运算符）和转换运算符（转换运算符）。

*Operator_body*为分号、***语句体***或***表达式主体***。语句体由*块*组成，它指定在调用运算符时要执行的语句。*块*必须符合方法体中所述的返回值的方法的规则。表达式主体由 `=>` 后跟表达式和分号组成，表示在调用运算符时要执行的单个表达式。

对于 `extern` 运算符，*operator_body*只包含一个分号。对于所有其他运算符，*operator_body*为块体或表达式主体。

以下规则适用于所有运算符声明：

- 运算符声明必须同时包含 `public` 和 `static` 修饰符。
- 运算符的参数必须是值参数（值参数）。运算符声明指定 `ref` 或 `out` 参数是编译时错误。
- 运算符（一元运算符、二元运算符、转换运算符）的签名必须不同于同一个类中声明的所有其他运算符的签名。
- 运算符声明中引用的所有类型必须至少具有与运算符本身相同的可访问性（辅助功能约束）。
- 同一修饰符在运算符声明中多次出现是错误的。

每个运算符类别都有其他限制，如以下各节所述。

与其他成员一样，基类中声明的运算符由派生类继承。由于运算符声明始终需要声明运算符的类或结构参与运算符的签名，因此，派生类中声明的运算符不能隐藏在基类中声明的运算符。因此，在运算符声明中绝不需要并且不允许使用 `new` 修饰符。

有关一元运算符和二元运算符的其他信息，请参阅运算符。

有关转换运算符的其他信息，请参阅用户定义的转换。

一元运算符

以下规则适用于一元运算符声明，其中 `T` 表示包含运算符声明的类或结构的实例类型：

- 一元 `+`、`-`、`!` 或 `~` 运算符必须采用类型 `T` 或 `T?` 的单个参数，并可以返回任何类型。
- 一元 `++` 或 `--` 运算符必须采用 `T` 或 `T?` 类型的单个参数，并且必须返回该类型或从其派生的类型。
- 一元 `true` 或 `false` 运算符必须采用 `T` 或 `T?` 类型的单个参数，并且必须返回类型 `bool`。

一元运算符的签名由运算符标记（`+`、`-`、`!`、`~`、`++`、`--`、`true` 或 `false`）和单个形参的类型组成。返回类型不是一元运算符的签名的一部分，也不是形参的名称。

`true` 和 `false` 一元运算符要求成对声明。如果类声明了这些运算符中的一个，而没有同时声明其他运算符，则会发生编译时错误。用户定义的条件逻辑运算符和布尔表达式中进一步介绍了 `true` 和 `false` 运算符。

下面的示例演示了一个实现，并为整数向量类 `operator ++` 了后续用法：

```
public class IntVector
{
    public IntVector(int length) {...}

    public int Length {...}                 // read-only property

    public int this[int index] {...}        // read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);    // vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;    // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;    // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

请注意，operator 方法如何返回通过将1添加到操作数而生成的值，就像后缀增量和减量运算符（后缀递增和递减运算符）以及前缀增量和减量运算符（前缀增量和减量运算符）。与在C++中不同的是，此方法不需要直接修改其操作数的值。事实上，修改操作数值将违反后缀增量运算符的标准语义。

二元运算符

以下规则适用于二元运算符声明，其中 `T` 表示包含运算符声明的类或结构的实例类型：

- 二元非移位运算符必须采用两个参数，其中至少一个参数必须具有类型 `T` 或 `T?`，并且可以返回任何类型。
- 二进制 `<<` 或 `>>` 运算符必须采用两个参数，其中第一个参数必须具有类型 `T` 或 `T?`，而第二个参数必须具有类型 `int` 或 `int?`，并且可以返回任何类型。

二元运算符的签名由运算符标记（`+`、`-`、`*`、`/`、`%`、`&`、`|`、`^`、`<<`、`>>`、`==`、`!=`、`>`、`<`、`>=` 或 `<=`）和两个形参的类型组成。返回类型和形参的名称不是二元运算符的签名的一部分。

某些二元运算符要求成对声明。对于每个运算符的每个声明，都必须有一个对的另一个运算符的匹配声明。当两个运算符声明具有相同的返回类型，并且每个参数具有相同的类型时，它们将匹配。以下运算符要求成对声明：

- `operator ==` 和 `operator !=`
- `operator >` 和 `operator <`
- `operator >=` 和 `operator <=`

转换运算符

转换运算符声明引入了***用户定义的转换***（用户定义的转换），以补充预定义的隐式和显式转换。

包含 `implicit` 关键字的转换运算符声明引入了用户定义的隐式转换。隐式转换可能会在多种情况下发生，包括函数成员调用、强制转换表达式和赋值。隐式转换中对此进行了进一步说明。

包含 `explicit` 关键字的转换运算符声明引入了用户定义的显式转换。显式转换可在强制转换表达式中发生，并在显式转换中进一步说明。

转换运算符从转换运算符的参数类型指示的源类型转换为目标类型，由转换运算符的返回类型指示。

对于给定的源类型 `S` 和目标类型 `T`，如果 `S` 或 `T` 是可以为 null 的类型，则允许 `S0` 和 `T0` 引用它们的基础类型，否则 `S0` 和 `T0` 分别等于 `S` 和 `T`。仅当满足以下所有条件时，才允许类或结构声明从源类型 `S` 转换为目标类型 `T`：

- `S0` 和 `T0` 属于不同类型。
- `S0` 或 `T0` 是发生运算符声明的类或结构类型。
- `S0` 和 `T0` 都不是*interface_type*的。
- 如果不包括用户定义的转换，则从 `S` 到 `T` 或从 `T` 到 `S` 的转换不存在。

对于这些规则，与 `S` 或 `T` 关联的任何类型参数都视为唯一的类型，这些类型与其他类型没有继承关系，并忽略这些类型参数上的任何约束。

示例中

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}      // Ok
    public static implicit operator C<string>(D<T> value) {...}   // Ok
    public static implicit operator C<T>(D<T> value) {...}        // Error
}
```

允许使用前两个运算符声明，因为对于索引器.3，分别 `T` 和 `int` 和 `string` 被视为唯一的类型，而无任何关系。但第三个运算符是错误，因为 `C<T>` 是 `D<T>` 的基类。

从第二个规则开始，转换运算符必须在声明运算符的类或结构类型中转换为或。例如，类或结构类型 `C` 可以定义从 `C` 到 `int` 之间的转换和从 `int` 到 `C` 的转换，但不能定义从 `int` 到 `bool` 的转换。

不能直接重新定义预定义的转换。因此，不允许转换运算符从或转换为 `object`，因为 `object` 和所有其他类型之间已存在隐式和显式转换。同样，转换的源类型和目标类型都不能是另一类型的基类型，因为转换已经存在。

但是，可以在针对特定类型参数的泛型类型上声明运算符，指定已作为预定义转换存在的转换。示例中

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

当类型 `object` 指定为 `T` 的类型参数时，第二个运算符将声明已存在的转换(隐式，因此也是从任何类型到类型 `object` 的显式转换)。

如果两种类型之间存在预定义的转换，则将忽略这些类型之间的任何用户定义的转换。尤其是在下列情况下：

- 如果存在从类型 `S` 到类型 `T` 的预定义隐式转换(隐式转换)，则将忽略从 `S` 到 `T` 的所有用户定义的转换(隐式或显式转换)。
- 如果从类型 `S` 中存在预定义的显式转换(显式转换)到类型 `T`，则将忽略从 `S` 到 `T` 的任何用户定义的显式转换。此外：

如果 `T` 是接口类型，则将忽略从 `S` 到 `T` 的用户定义隐式转换。

否则，仍会考虑从 `S` 到 `T` 的用户定义隐式转换。

对于所有类型但 `object`，以上 `Convertible<T>` 类型声明的运算符不会与预定义转换冲突。例如：

```
void F(int i, Convertible<int> n) {
    i = n;                          // Error
    i = (int)n;                     // User-defined explicit conversion
    n = i;                          // User-defined implicit conversion
    n = (Convertible<int>)i;        // User-defined implicit conversion
}
```

但是，对于类型 `object`，预定义的转换在所有情况下都会隐藏用户定义的转换，但会隐藏其中一项：

```
void F(object o, Convertible<object> n) {
    o = n;                         // Pre-defined boxing conversion
    o = (object)n;                 // Pre-defined boxing conversion
    n = o;                         // User-defined implicit conversion
    n = (Convertible<object>)o;    // Pre-defined unboxing conversion
}
```

不允许用户定义的转换从或转换到*interface_type*。特别是，此限制可确保在转换为*interface_type*时不会发生用户定义的转换，并且仅当被转换的对象真正实现指定的*interface_type*时，才能成功转换为*interface_type* 。

转换运算符的签名包含源类型和目标类型。(请注意，这是返回类型参与签名的唯一成员形式。)转换运算符的 `implicit` 或 `explicit` 分类不是运算符签名的一部分。因此，类或结构不能声明具有相同源和目标类型的 `implicit` 和 `explicit` 转换运算符。

通常，用户定义的隐式转换应设计为从不引发异常，并且永远不会丢失信息。如果用户定义的转换可以增加异常(例如，由于源参数超出范围)或丢失信息(如丢弃高序位)，则应将该转换定义为显式转换。

示例中

```
using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

从 `Digit` 到 `byte` 的转换是隐式的，因为它从不引发异常或丢失信息，但从 `byte` 到 `Digit` 的转换是显式的，因为 `Digit` 只能表示 `byte` 的可能值的子集。

实例构造函数

*实**例构造函数***是实现初始化类实例所需执行的操作的成员。实例构造函数是使用*constructor_declaration*s 声明的：

```
constructor_declaration
    : attributes? constructor_modifier* constructor_declarator constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | constructor_modifier_unsafe
    ;

constructor_declarator
    : identifier '(' formal_parameter_list? ')' constructor_initializer?
    ;

constructor_initializer
    : ':' 'base' '(' argument_list? ')'
    | ':' 'this' '(' argument_list? ')'
    ;

constructor_body
    : block
    | ';'
    ;
```

*Constructor_declaration*可以包含一组*特性*（特性）、四个访问修饰符（访问修饰符）和 `extern`（外部方法）修饰符的有效组合。不允许构造函数声明多次包含同一修饰符。

*Constructor_declarator*的*标识符*必须命名声明实例构造函数的类。如果指定了其他名称，则会发生编译时错误。

实例构造函数的可选*formal_parameter_list*遵从与方法（方法）*formal_parameter_list*相同的规则。形参表定义实例构造函数的签名（签名和重载），并控制重载决策（类型推理）在调用中选择特定实例构造函数的过程。

实例构造函数的*formal_parameter_list*中引用的每个类型必须至少具有与构造函数本身相同的可访问性(辅助功能约束)。

可选*constructor_initializer*指定要在执行此实例构造函数的*constructor_body*中给定的语句之前调用的另一个实例构造函数。构造函数初始值设定项中对此进行了进一步说明。

当构造函数声明包含 `extern` 修饰符时，构造函数被称为***外部构造函数***。由于外部构造函数声明不提供实际的实现，因此它的*constructor_body*由分号组成。对于所有其他构造函数，*constructor_body*包含一个*块*，它指定用于初始化类的新实例的语句。这与实例方法的*块*完全对应，具有 `void` 返回类型(方法体)。

不继承实例构造函数。因此，类没有实例构造函数，而不是类中实际声明的构造函数。如果类不包含实例构造函数声明，则会自动提供默认实例构造函数(默认构造函数)。

实例构造函数由*object_creation_expression*(对象创建表达式)和*constructor_initializer*来调用。

构造函数初始值设定项

所有实例构造函数(类 `object` 的类构造函数除外)都隐式包含对*constructor_body*之前的其他实例构造函数的调用。隐式调用的构造函数由*constructor_initializer*确定：

- `base(argument_list)` 或 `base()` 形式的实例构造函数初始值设定项将导致调用直接基类的实例构造函数。使用*argument_list*如果存在，则选择该构造函数，并使用重载决策的重载决策规则。如果未在直接基类中声明任何实例构造函数，候选实例构造函数集则包含直接基类中包含的所有可访问实例构造函数，或者默认构造函数(默认构造函数)。如果此集为空，或者无法识别单个最佳实例构造函数，则会发生编译时错误。
- `this(argument-list)` 或 `this()` 形式的实例构造函数初始值设定项会导致调用类自身的实例构造函数。如果存在，则使用*argument_list*，并使用重载决策的重载决策规则来选择构造函数。候选实例构造函数集包含类本身中声明的所有可访问实例构造函数。如果此集为空，或者无法识别单个最佳实例构造函数，则会发生编译时错误。如果实例构造函数声明包含调用构造函数本身的构造函数初始值设定项，则会发生编译时错误。

如果实例构造函数没有构造函数初始值设定项，则会隐式提供形式 `base()` 的构造函数初始值设定项。因此，形式的实例构造函数声明

```
C(...) {...}
```

完全等效于

```
C(...): base() {...}
```

实例构造函数声明的*formal_parameter_list*给定的参数的范围包含该声明的构造函数初始值设定项。因此，可以使用构造函数初始值设定项来访问构造函数的参数。例如：

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

实例构造函数初始值设定项无法访问正在创建的实例。因此，在构造函数初始值设定项的参数表达式中引用 `this` 是编译时错误，这是因为自变量表达式的编译时错误通过*simple_name*引用任何实例成员。

实例变量初始值设定项

如果实例构造函数不具有构造函数初始值设定项，或者它具有形式 `base(...)` 形式的构造函数初始值设定项，则该构造函数将隐式执行由其类中声明的实例字段的*variable_initializer*指定的初始化。这对应于在进入构造函数之后、直接调用直接基类构造函数之前立即执行的一系列赋值。变量初始值设定项将按照它们在类声明中的显示顺序执行。

构造函数执行

变量初始值设定项转换为赋值语句，并在调用基类实例构造函数之前执行这些赋值语句。此顺序可确保在执行有权访问该实例的任何语句之前，通过其变量初始值设定项来初始化所有实例字段。

给定示例

```
using System;

class A
{
    public A() {
        PrintFields();
    }

    public virtual void PrintFields() {}
}

class B: A
{
    int x = 1;
    int y;

    public B() {
        y = -1;
    }

    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

`new B()` 用于创建 `B` 的实例时，将生成以下输出：

```
x = 1, y = 0
```

`x` 的值为1，因为在调用基类实例构造函数之前将执行变量初始值设定项。但 `y` 的值为0（`int` 的默认值），因为在基类构造函数返回后才会执行 `y` 的赋值。

将实例变量初始值设定项和构造函数初始值设定项视为自动插入*constructor_body*前面的语句是非常有用的。
示例

```
using System;
using System.Collections;

class A
{
    int x = 1, y = -1, count;

    public A() {
        count = 0;
    }

    public A(int n) {
        count = n;
    }
}

class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;

    public B(): this(100) {
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        max = n;
    }
}
```

包含若干变量初始值设定项;它还包含两个窗体的构造函数初始值设定项(`base` 和 `this`)。该示例与下面显示的代码相对应,其中每个注释指示一个自动插入的语句(用于自动插入的构造函数调用的语法无效,但仅用于说明这种机制)。

```
using System.Collections;

class A
{
    int x, y, count;

    public A() {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);                     // Invoke B(int) constructor
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0);     // Variable initializer
        items = new ArrayList(100); // Variable initializer
        A(n - 1);                   // Invoke A(int) constructor
        max = n;
    }
}
```

默认构造函数

如果类不包含实例构造函数声明，则会自动提供一个默认实例构造函数。该默认构造函数只调用直接基类的无参数构造函数。如果类是抽象类，则对默认构造函数的声明的可访问性是受保护的。否则，默认构造函数的声明的可访问性是公共的。因此，默认构造函数始终为形式

```
protected C(): base() {}
```

或

```
public C(): base() {}
```

其中 `C` 是类的名称。如果重载决策无法确定基类构造函数初始值设定项的唯一最佳候选项，则会发生编译时错误。

示例中

```
class Message
{
    object sender;
    string text;
}
```

由于类不包含实例构造函数声明，因此提供了默认的构造函数。因此，该示例完全等效于

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

私有构造函数

当类 `T` 仅声明私有实例构造函数时，不能 `T` 的程序文本外的类从 `T` 派生或直接创建 `T` 的实例。因此，如果某个类只包含静态成员，并且不应实例化，则添加一个空的私有实例构造函数会阻止实例化。例如：

```
public class Trig
{
    private Trig() {}        // Prevent instantiation

    public const double PI = 3.14159265358979323846;

    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

`Trig` 类将相关方法和常量分组，但不能进行实例化。因此，它声明了一个空的私有实例构造函数。至少必须将一个实例构造函数声明为禁止自动生成默认构造函数。

可选实例构造函数参数

构造函数初始值设定项的 `this(...)` 形式通常与重载一起使用，以实现可选的实例构造函数参数。示例中

```
class Text
{
    public Text(): this(0, 0, null) {}

    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

前两个实例构造函数只为缺少的参数提供默认值。这两种方法都使用 `this(...)` 构造函数初始值设定项调用第三个实例构造函数，该构造函数实际上用于初始化新实例。其结果是可选的构造函数参数：

```
Text t1 = new Text();                   // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);              // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

静态构造函数

静态构造函数是实现初始化封闭式类类型所需的操作的成员。静态构造函数使用*static_constructor_declaration*来声明：

```
static_constructor_declaration
    : attributes? static_constructor_modifiers identifier '(' ')' static_constructor_body
    ;

static_constructor_modifiers
    : 'extern'? 'static'
    | 'static' 'extern'?
    | static_constructor_modifiers_unsafe
    ;

static_constructor_body
    : block
    | ';'
    ;
```

*Static_constructor_declaration*可以包含一组*特性*（特性）和一个 `extern` 修饰符（外部方法）。

*Static_constructor_declaration*的*标识符*必须命名声明静态构造函数的类。如果指定了其他名称，则会发生编译时错误。

如果静态构造函数声明包含 `extern` 修饰符，则将静态构造函数称为***外部静态构造函数***。由于外部静态构造函数声明不提供实际的实现，因此它的*static_constructor_body*由分号组成。对于所有其他静态构造函数声明，*static_constructor_body*包含一个*块*，它指定要执行的语句，以便初始化类。这与 `void` 返回类型（方法体）的静态方法的*method_body*完全对应。

静态构造函数不是继承的，不能直接调用。

封闭式类类型的静态构造函数在给定的应用程序域中最多执行一次。在应用程序域中，以下事件的第一个事件会触发静态构造函数的执行：

- 创建类类型的实例。
- 引用类类型的任何静态成员。

如果类包含执行开始时所处的 `Main` 方法（应用程序启动），则在调用 `Main` 方法之前，将执行该类的静态构造函数。

若要初始化新的封闭式类类型，请先创建一组新的已关闭类型的静态字段（静态字段和实例字段）。每个静态字段都初始化为其默认值（默认值）。接下来，对这些静态字段执行静态字段初始值设定项（静态字段初始化）。最后，执行静态构造函数。

示例

```
using System;

class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}

class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}

class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}
```

必须生成输出：

```
Init A
A.F
Init B
B.F
```

因为对 `A.F` 的调用触发了 `A` 的静态构造函数的执行，并且 `B` 的静态构造函数的执行由对 `B.F` 的调用触发。

可以构造循环依赖关系，以便在其默认值状态中观察包含变量初始值设定项的静态字段。

示例

```
using System;

class A
{
    public static int X;

    static A() {
        X = B.Y + 1;
    }
}

class B
{
    public static int Y = A.X + 1;

    static B() {}

    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

生成输出

```
X = 1, Y = 2
```

若要执行 `Main` 方法，系统首先在类 `B` 的静态构造函数之前运行 `B.Y` 的初始化表达式。 `Y` 初始值设定项会导致运行 `A` 的静态构造函数，因为引用 `A.X` 的值。 `A` 的静态构造函数将继续计算 `X` 的值，并且在执行此操作时，将获取 `Y` 的默认值，即零。 `A.X` 将初始化为1。运行 `A` 的静态字段初始值设定项和静态构造函数的过程随后完成，返回 `Y` 初始值的计算结果为2。

由于静态构造函数仅对每个封闭式构造类类型执行一次，因此，在不能通过约束（类型参数约束）在编译时无法检查的类型参数上强制执行运行时检查。例如，下面的类型使用静态构造函数来强制类型参数是一个枚举：

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

析构函数。

析构函数是实现析构类的实例所需的操作的成员。使用*destructor_declaration*声明析构函数：

```
destructor_declaration
    : attributes? 'extern'? '~' identifier '(' ')' destructor_body
    | destructor_declaration_unsafe
    ;

destructor_body
    : block
    | ';'
    ;
```

*Destructor_declaration*可以包含一组*属性*（特性）。

*Destructor_declaration*的*标识符*必须命名声明析构函数的类。如果指定了其他名称，则会发生编译时错误。

当析构函数声明包含 `extern` 修饰符时，析构函数被称为***外部析构函数***。由于外部析构函数声明不提供实际的实现，因此它的*destructor_body*由分号组成。对于所有其他析构函数，*destructor_body*包含一个*块*，它指定要执行的语句，以便销毁类的实例。*Destructor_body*完全对应于具有 `void` 返回类型（方法主体）的实例方法的*method_body* 。

析构函数不是继承的。因此，一个类不包含析构函数，而析构函数可能在该类中声明。

由于析构函数不需要任何参数，因此不能重载，因此一个类最多只能有一个析构函数。

析构函数是自动调用的，不能被显式调用。当某个实例对于任何代码都无法使用该实例时，该实例将可用于销毁。在实例符合析构条件后，可能会在任何时间执行实例的析构函数。当实例销毁时，该实例的继承链中的析构函数将按顺序调用，从最常派生到最小派生。析构函数可以在任何线程上执行。若要进一步讨论控制何时以及如何执行析构函数，请参阅自动内存管理。

示例的输出

```
using System;

class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}

class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}

class Test
{
   static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
   }
}
```

is

```
B's destructor
A's destructor
```

由于继承链中的析构函数按顺序调用，从最常派生到最小派生。

析构函数是通过重写 `System.Object` 上的虚拟方法 `Finalize` 来实现的。C#不允许程序重写此方法或直接对其进行调用(或重写)。例如，程序

```
class A
{
    override protected void Finalize() {}    // error

    public void F() {
        this.Finalize();                      // error
    }
}
```

包含两个错误。

编译器的行为就像此方法和它的替代一样，根本就不存在。因此，此程序：

```
class A
{
    void Finalize() {}                        // permitted
}
```

有效，并显示的方法隐藏 `System.Object` 的 `Finalize` 方法。

有关从析构函数引发异常时的行为的讨论，请参阅如何处理异常。

迭代器

使用迭代器块（块）实现的函数成员（函数成员）称为***迭代器***。

迭代器块可以用作函数成员的主体，前提是相应函数成员的返回类型是一个枚举器接口（枚举器接口），或者是一个可枚举接口（可枚举接口）。它可以作为*method_body*、*operator_body*或*accessor_body*出现，而事件、实例构造函数、静态构造函数和析构函数不能作为迭代器实现。

使用迭代器块实现函数成员时，函数成员的形参表的编译时错误是指定任何 `ref` 或 `out` 参数。

枚举器接口

枚举器接口是 `System.Collections.IEnumerator` 的非泛型接口和泛型接口的所有实例化 `System.Collections.Generic.IEnumerator<T>`。为了简单起见，在本章中，这些接口分别作为 `IEnumerator` 和 `IEnumerator<T>` 进行引用。

可枚举接口

可***枚举接口***是 `System.Collections.IEnumerable` 的非泛型接口和泛型接口的所有实例化 `System.Collections.Generic.IEnumerable<T>`。为了简单起见，在本章中，这些接口分别作为 `IEnumerable` 和 `IEnumerable<T>` 进行引用。

Yield 类型

迭代器生成一个值序列，该序列具有相同的类型。此类型称为迭代器的***yield 类型***。

- 返回 `IEnumerator` 或 `IEnumerable` 的迭代器的 yield 类型 `object`。
- 返回 `IEnumerator<T>` 或 `IEnumerable<T>` 的迭代器的 yield 类型 `T`。

枚举器对象

当使用迭代器块实现返回枚举器接口类型的函数成员时，调用函数成员不会立即执行迭代器块中的代码。相反，将创建并返回一个***枚举器对象***。此对象封装在迭代器块中指定的代码，并在调用枚举器对象的 `MoveNext` 方法时，执行迭代器块中的代码。枚举器对象具有以下特征：

- 它实现 `IEnumerator` 和 `IEnumerator<T>`，其中 `T` 是迭代器的 yield 类型。
- 它实现 `System.IDisposable`。
- 使用传递到函数成员的参数值（如果有）和实例值的副本对其进行初始化。

- 它有四种可能的状态："*之前*"、"*正在运行*"、"已*暂停*" 和 "*之后*"，最初处于*之前*的状态。

枚举器对象通常是编译器生成的枚举器类的一个实例，它封装迭代器块中的代码并实现枚举器接口，但也可以实现其他方法。如果枚举器类由编译器生成，则该类将直接或间接嵌套在包含函数成员的类中，它将具有私有可访问性，并且它将具有为编译器使用而保留的名称（标识符）。

枚举器对象可以实现比上面指定的接口更多的接口。

以下各节描述了枚举器对象提供的 `IEnumerable` 和 `IEnumerable<T>` 接口实现的 `MoveNext`、`Current` 和 `Dispose` 成员的确切行为。

请注意，枚举器对象不支持 `IEnumerator.Reset` 方法。调用此方法将导致引发 `System.NotSupportedException`。

MoveNext 方法

枚举器对象的 `MoveNext` 方法封装迭代器块的代码。调用 `MoveNext` 方法将执行迭代器块中的代码，并根据需要设置枚举器对象的 `Current` 属性。`MoveNext` 执行的精确操作取决于调用 `MoveNext` 时枚举器对象的状态：

- 如果枚举器对象的状态在*之前*，则调用 `MoveNext`：
  - 将状态更改为*正在运行*。
  - 将迭代器块的参数（包括 `this`）初始化到参数值和初始化枚举器对象时保存的实例值。
  - 从一开始就执行迭代器块，直到中断执行（如下所述）。
- 如果枚举器对象的状态为*正在运行*，则不指定调用 `MoveNext` 的结果。
- 如果枚举器对象的状态为 "已*挂起*"，则调用 `MoveNext`：
  - 将状态更改为*正在运行*。
  - 将所有局部变量和参数的值（包括此值）还原到上次挂起执行迭代器块时保存的值。请注意，这些变量所引用的任何对象的内容可能自上一次调用 MoveNext 后发生了更改。
  - 在导致执行挂起的 `yield return` 语句之后立即继续执行迭代器块，并继续执行，直到中断执行（如下所述）。
- 如果枚举数对象的状态为*after*，则调用 `MoveNext` 将返回 `false`。

当 `MoveNext` 执行迭代器块时，可以通过以下四种方式中断执行：通过 `yield return` 语句、`yield break` 语句、遇到迭代器块的末尾以及引发的异常并传播到迭代器块中。

- 遇到 `yield return` 语句时（yield 语句）：
  - 计算语句中给定的表达式，将其隐式转换为 yield 类型，并将其分配给枚举器对象的 `Current` 属性。
  - 迭代器主体的执行将被挂起。保存所有局部变量和参数的值（包括 `this`），这与此 `yield return` 语句的位置相同。如果 `yield return` 语句位于一个或多个 `try` 块中，则不会执行关联的 `finally` 块。
  - 枚举器对象的状态将更改为 "已*挂起*"。
  - `MoveNext` 方法将 `true` 返回到其调用方，这表示迭代已成功地推进到下一个值。
- 遇到 `yield break` 语句时（yield 语句）：
  - 如果 `yield break` 语句在一个或多个 `try` 块中，则将执行关联的 `finally` 块。
  - 枚举器对象的状态将更改为*after*。
  - `MoveNext` 方法将 `false` 返回到其调用方，指示迭代已完成。
- 当遇到迭代器主体的末尾时：
  - 枚举器对象的状态将更改为*after*。
  - `MoveNext` 方法将 `false` 返回到其调用方，指示迭代已完成。
- 当引发异常并将其传播到迭代器块外时：
  - 迭代器正文中的相应 `finally` 块将由异常传播执行。
  - 枚举器对象的状态将更改为*after*。
  - 异常传播继续到 `MoveNext` 方法的调用方。

当前属性

枚举器对象的 `Current` 属性受迭代器块中 `yield return` 语句的影响。

当枚举器对象处于*挂起*状态时，`Current` 的值为*之前*对 `MoveNext` 所设置的值。如果枚举器对象位于 "*之前*"、"*正在运行*" 或 "*之后*" 状态，则不指定访问 `Current` 的结果。

对于 yield 类型不是 `object` 的迭代器，通过枚举器对象的 `IEnumerable` 实现访问 `Current` 的结果对应于通过枚举器对象的 `IEnumerator<T>` 实现访问 `Current`，并将结果强制转换为 `object`。

Dispose 方法

`Dispose` 方法用于通过将枚举器对象的状态设置为*after*来清理迭代。

- 如果枚举数对象的状态在*之前*，则调用 `Dispose` 会将状态更改为*after*。
- 如果枚举器对象的状态为*正在运行*，则不指定调用 `Dispose` 的结果。
- 如果枚举器对象的状态为 "已*挂起*"，则调用 `Dispose`：
  - 将状态更改为*正在运行*。
  - 执行所有 finally 块，就好像最后执行的 `yield return` 语句是 `yield break` 语句一样。如果这导致引发异常并将其传播到迭代器主体外，则枚举器对象的状态将设置为*after*，并将异常传播到 `Dispose` 方法的调用方。
  - 将状态更改为*after*。
- 如果枚举数对象的状态为*after*，则调用 `Dispose` 不会有任何影响。

可枚举对象

当使用迭代器块实现返回可枚举接口类型的函数成员时，调用函数成员不会立即执行迭代器块中的代码。相反，将创建并返回一个可*枚举对象*。可枚举对象的 `GetEnumerator` 方法返回一个枚举器对象，该对象封装迭代器块中指定的代码，并在调用枚举器对象的 `MoveNext` 方法时，执行迭代器块中的代码。可枚举对象具有以下特征：

- 它实现 `IEnumerable` 和 `IEnumerable<T>`，其中 `T` 是迭代器的 yield 类型。
- 使用传递到函数成员的参数值（如果有）和实例值的副本对其进行初始化。

可枚举对象通常是编译器生成的可枚举类的实例，它封装迭代器块中的代码并实现可枚举接口，但也可以实现其他方法。如果可枚举的类由编译器生成，则该类将直接或间接嵌套在包含函数成员的类中，它将具有私有可访问性，并且它将具有为编译器使用而保留的名称（标识符）。

可枚举对象可以实现比上面指定的接口更多的接口。具体而言，可枚举对象还可以实现 `IEnumerator` 和 `IEnumerator<T>`，从而使其既可用作可枚举的，也可用作枚举器。在该类型的实现中，第一次调用可枚举对象的 `GetEnumerator` 方法时，将返回可枚举对象本身。对可枚举对象的 `GetEnumerator`（如果有）的后续调用会返回可枚举对象的副本。因此，每个返回的枚举器都有其自己的状态，并且一个枚举器中的更改不会影响另一个枚举

GetEnumerator 方法

可枚举对象提供 `IEnumerable` 和 `IEnumerable<T>` 接口的 `GetEnumerator` 方法的实现。这两个 `GetEnumerator` 方法共享一个公共实现，该实现获取并返回可用的枚举器对象。初始化枚举器对象时，将在初始化可枚举对象时保存的参数值和实例值进行初始化，否则枚举器对象的功能如枚举器对象中所述。

实现示例

本部分介绍了在标准C#构造中可能的迭代器实现。此处所述的实现基于 Microsoft C#编译器使用的相同原则，但它并不意味着强制实施或唯一可能。

以下 `Stack<T>` 类使用迭代器来实现其 `GetEnumerator` 方法。迭代器按从上到下的顺序枚举堆栈中的元素。

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

可以将 `GetEnumerator` 方法转换为编译器生成的枚举器类的实例化，该类封装迭代器块中的代码，如下所示。

```
class Stack<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
        __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
        __state1:
            --i;
            goto __loop;
        __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

在前面的转换中，迭代器块中的代码将转换为状态机并放置在枚举器类的 `MoveNext` 方法中。此外，局部变量 `i` 会转换为枚举器对象中的字段，因此它可以在 `MoveNext` 的调用中继续存在。

下面的示例将打印整数1到10的简单乘法表。示例中的 `FromTo` 方法返回一个可枚举对象，并使用迭代器实现。

```
using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}
```

可以将 `FromTo` 方法转换为编译器生成的可枚举类的实例化，该枚举类封装迭代器块中的代码，如下所示。

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }

        public int Current {
```

```
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
            case 1:
                if (from > to) goto case 2;
                __current = from++;
                __state = 1;
                return true;
            case 2:
                __state = 2;
                return false;
            default:
                throw new InvalidOperationException();
            }
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

可枚举的类既实现可枚举接口又实现枚举器接口，使其既可用作可枚举的，也可用作枚举器。第一次调用 `GetEnumerator` 方法时，将返回可枚举对象本身。对可枚举对象的 `GetEnumerator`（如果有）的后续调用会返回可枚举对象的副本。因此，每个返回的枚举器都有其自己的状态，并且一个枚举器中的更改不会影响另一个枚举 `Interlocked.CompareExchange` 方法用于确保线程安全的操作。

`from` 和 `to` 参数会转换为可枚举类中的字段。因为在迭代器块中修改了 `from`，所以引入了一个附加的 `__from` 字段来保存提供给每个枚举器中的 `from` 的初始值。

如果 `0``__state`，则 `MoveNext` 方法会引发 `InvalidOperationException`。这可以防止将可枚举对象用作枚举器对象，而无需先调用 `GetEnumerator`。

下面的示例演示一个简单的 tree 类。`Tree<T>` 类使用迭代器来实现其 `GetEnumerator` 方法。迭代器按中缀顺序枚举树的元素。

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main() {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();

        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

可以将 `GetEnumerator` 方法转换为编译器生成的枚举器类的实例化，该类封装迭代器块中的代码，如下所示。

```
class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> __this;
        IEnumerator<T> __left, __right;
```

```
        int __state;
        T __current;

        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            try {
                switch (__state) {

                case 0:
                    __state = -1;
                    if (__this.left == null) goto __yield_value;
                    __left = __this.left.GetEnumerator();
                    goto case 1;

                case 1:
                    __state = -2;
                    if (!__left.MoveNext()) goto __left_dispose;
                    __current = __left.Current;
                    __state = 1;
                    return true;

                __left_dispose:
                    __state = -1;
                    __left.Dispose();

                __yield_value:
                    __current = __this.value;
                    __state = 2;
                    return true;

                case 2:
                    __state = -1;
                    if (__this.right == null) goto __end;
                    __right = __this.right.GetEnumerator();
                    goto case 3;

                case 3:
                    __state = -3;
                    if (!__right.MoveNext()) goto __right_dispose;
                    __current = __right.Current;
                    __state = 3;
                    return true;

                __right_dispose:
                    __state = -1;
                    __right.Dispose();

                __end:
                    __state = 4;
                    break;

                }
            }
            finally {
                if (__state < 0) Dispose();
            }
            return false;
        }
```

```
        public void Dispose() {
            try {
                switch (__state) {

                case 1:
                case -2:
                    __left.Dispose();
                    break;

                case 3:
                case -3:
                    __right.Dispose();
                    break;

                }
            }
            finally {
                __state = 4;
            }
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

将 `foreach` 语句中使用的编译器生成的临时内存提升为枚举器对象的 `__left` 和 `__right` 字段。将仔细更新枚举器对象的 `__state` 字段，以便在引发异常时正确调用正确的 `Dispose()` 方法。请注意，不能用简单的 `foreach` 语句来编写翻译后的代码。

异步函数

带有 `async` 修饰符的方法（方法）或匿名函数（匿名函数表达式）称为***异步函数***。通常，术语*async*用于描述具有 `async` 修饰符的任何类型的函数。

异步函数的形参列表的编译时错误是指定任何 `ref` 或 `out` 参数。

异步方法的*return_type*必须是 `void` 或***任务类型***。任务类型是 `System.Threading.Tasks.Task` 的，并且是从 `System.Threading.Tasks.Task<T>` 构造的类型。为了简单起见，在本章中，这些类型分别作为 `Task` 和 `Task<T>` 进行引用。返回任务类型的异步方法称作任务返回。

任务类型的确切定义是定义的实现，但从语言的角度来看，任务类型处于 "未完成"、"成功" 或 "出错" 状态之一。出错的任务记录相关的异常。成功的 `Task<T>` 记录 `T` 类型的结果。任务类型是可等待的，因此可以是 await 表达式（await 表达式）的操作数。

异步函数调用能够通过其主体中的 await 表达式（await 表达式）来挂起计算。稍后可以通过***恢复委托***在挂起 await 表达式时恢复计算。恢复委托的类型为 `System.Action`，当调用该函数时，异步函数调用的计算将从其停止的等待表达式中继续。如果从未挂起函数调用，则异步函数调用的***当前调用方***是原始调用方，否则为恢复委托的最新调用方。

任务返回的异步函数的计算

调用任务返回的 async 函数会导致生成返回的任务类型的实例。这称为 async 函数的***返回任务***。任务最初处于 "未完成" 状态。

然后，将计算异步函数主体，直到它被挂起（通过等待表达式）或终止，此时，将控件返回给调用方，同时返回 task。

当 async 函数的主体终止时，返回的任务将不再处于 "未完成" 状态：

- 如果函数体因到达 return 语句或正文末尾而终止，则返回的任务中会记录任何结果值，这会置于成功状态。
- 如果函数体由于未捕获的异常而终止（throw 语句），则会在返回任务中记录异常，并将其置于出错状态。

Void 返回的 async 函数的计算

如果异步函数的返回类型为 `void`，则计算方式与上述方法不同：由于没有返回任何任务，因此函数会将完成和异常传达给当前线程的***同步上下文***。同步上下文的确切定义是依赖于实现的，但它表示当前线程正在运行的 "where"。当对返回 void 的异步函数开始、成功完成或导致引发未捕获的异常时，会通知同步上下文。

这样，上下文就可以跟踪正在其下运行的空返回异步函数的数量，并决定如何传播它们发出的异常。

结构

2020/11/2 • • Edit Online

结构与类相似，它们表示可以包含数据成员和函数成员的数据结构。但是，与类不同的是，结构是值类型，不需要进行堆分配。结构类型的变量直接包含结构的数据，而类类型的变量包含对数据的引用，后者称为对象。

结构对包含值语义的小型数据结构特别有用。复数、坐标系中的点或字典中的键值对都是结构的典型示例。这些数据结构的关键在于它们有少量的数据成员，它们不需要使用继承或引用标识，并且可以使用赋值语义方便地实现这些数据结构，赋值将复制值而不是引用。

如简单类型中所述，提供的简单类型C#(如 `int`、`double`和 `bool`)实际上都是所有结构类型。正如这些预定义类型是结构一样，还可以使用结构和运算符重载来实现语言的C#新 "基元" 类型。本章末尾提供了两种类型的示例(结构示例)。

结构声明

*Struct_declaration*是声明新结构的*type_declaration* (类型声明)：

```
struct_declaration
    : attributes? struct_modifier* 'partial'? 'struct' identifier type_parameter_list?
      struct_interfaces? type_parameter_constraints_clause* struct_body ';'?
    ;
```

*Struct_declaration*包含一组可选的*特性*(特性)，后跟一组可选的*struct_modifier*s (结构修饰符)，后跟一个可选的 `partial` 修饰符 `struct`，然后是一个可选的修饰符，后面跟一个可选的修饰符，然后是一个命名该结构的*标识符*(类型参数)，后跟可选的 type_parameter_list 规范(Partial 修饰符))，后跟一个可选的*type_parameter_constraints_clause*s 规范(类型参数约束)，后跟一个*struct_body* (结构体)，后面可以跟一个分号。

结构修饰符

*Struct_declaration*可以选择性地包含一系列结构修饰符：

```
struct_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | struct_modifier_unsafe
    ;
```

同一修饰符在结构声明中出现多次是编译时错误。

结构声明的修饰符与类声明(类声明)的修饰符具有相同的含义。

Partial 修饰符

`partial` 修饰符指示此*struct_declaration*为分部类型声明。在封闭命名空间或类型声明中具有相同名称的多个分部结构声明组合在一起，并遵循在部分类型中指定的规则组成一个结构声明。

结构接口

结构声明可以包含*struct_interfaces*规范，在这种情况下，结构被称为直接实现给定的接口类型。

```
struct_interfaces
    : ':' interface_type_list
    ;
```

接口实现中进一步讨论了接口实现。

结构体

结构的*struct_body*定义结构的成员。

```
struct_body
    : '{' struct_member_declaration* '}'
    ;
```

结构成员

结构的成员由其*struct_member_declaration*引入的成员和从该类型继承的成员组成 `System.ValueType` 。

```
struct_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | static_constructor_declaration
    | type_declaration
    | struct_member_declaration_unsafe
    ;
```

除了类和结构差异中说明的差异以外，通过迭代器类成员提供的类成员的说明也适用于结构成员。

类和结构的差异

结构在几个重要方面不同于类：

- 结构是值类型（值语义）。
- 所有结构类型均隐式继承自类 `System.ValueType` （继承）。
- 对结构类型的变量赋值会创建所赋的值（赋值）的副本。
- 结构的默认值是通过将所有值类型字段设置为其默认值，并将所有引用类型字段设置为 `null` （默认值）而产生的值。
- 装箱和取消装箱操作用于在结构类型和 `object` （装箱和取消装箱）之间进行转换。
- 对于结构（此访问），`this` 的含义是不同的。
- 不允许结构的实例字段声明包含变量初始值设定项（字段初始值设定项）。
- 不允许结构声明无参数的实例构造函数（构造函数）。
- 不允许结构声明析构函数（析构函数）。

值语义

结构是值类型（值类型），被称为具有值语义。另一方面，类是引用类型（引用类型），并且称为具有引用语义。

结构类型的变量直接包含结构的数据，而类类型的变量包含对数据的引用，后者称为对象。当结构 `B` 包含类型 `A` 的实例字段并且 `A` 为结构类型时，`A` 依赖于 `B` 或从 `B` 构造的类型，则会发生编译时错误。如果 `X` 包含类型 `Y` 的实例字段，则结构 `X` ***直接依赖于***结构 `Y` 。根据此定义，结构所依赖的整个结构集是***直接取决于***关系

的传递闭包。例如

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

是一个错误，因为 `Node` 包含自己的类型的实例字段。另一个示例

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

是错误的，因为每种类型 `A`、`B` 和 `C` 都相互依赖。

对于类，两个变量可以引用同一对象，因此，对一个变量执行的运算可能会影响另一个变量所引用的对象。使用结构，每个变量都有自己的数据副本（在 `ref` 和 `out` 参数变量的情况下除外），对一个变量执行的操作不可能影响另一个变量。此外，由于结构不是引用类型，因此不可能 `null` 结构类型的值。

给定声明

```
struct Point
{
    public int x, y;

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

代码片段

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

输出 `10` 的值。将 `a` 分配给 `b` 会创建值的副本，`b` 因此不受 `a.x` 赋值的影响。`Point` 改为作为类进行声明，输出将 `100`，因为 `a` 和 `b` 将引用相同的对象。

继承

所有结构类型都隐式继承自类 `System.ValueType`，而该类又从类 `object` 继承。结构声明可以指定实现接口的列表，但结构声明不可能指定基类。

结构类型永远不是抽象类型，并且始终隐式密封。因此，在结构声明中不允许使用 `abstract` 和 `sealed` 修饰符。

由于结构不支持继承，因此结构成员的声明的可访问性不能 `protected` 或 `protected internal`。

结构中的函数成员不能是 `abstract` 或 `virtual`，只允许 `override` 修饰符重写继承自 `System.ValueType` 的方法。

赋值

对结构类型的变量赋值会创建要分配的值的副本。这不同于对类类型的变量的赋值，后者复制引用，而不是由引用标识的对象。

与赋值类似，当结构作为值参数传递或作为函数成员的结果返回时，将创建结构的副本。可以通过使用 `ref` 或 `out` 参数的函数成员引用来传递结构。

当结构的属性或索引器是赋值的目标时，与属性或索引器访问关联的实例表达式必须归类为变量。如果实例表达式归类为值，则会发生编译时错误。这将在简单的赋值中更详细地介绍。

默认值

如 "默认值" 中所述，在创建多个变量时，它们会自动初始化为其默认值。对于类类型和其他引用类型的变量，此默认值为 `null`。但是，由于结构是不能 `null` 的值类型，因此结构的默认值是通过将所有值类型字段设置为其默认值，并将所有引用类型字段设置为 `null` 生成的值。

引用上面声明的 `Point` 结构，示例

```
Point[] a = new Point[100];
```

将数组中的每个 `Point` 初始化为通过将 "`x`" 和 "`y`" 字段设置为零而生成的值。

结构的默认值对应于结构的默认构造函数返回的值(默认构造函数)。与类不同，结构不允许声明无参数的实例构造函数。相反，每个结构都隐式包含一个无参数的实例构造函数，该构造函数始终返回通过将所有值类型字段设置为其默认值，并将所有引用类型字段设置为 `null` 而产生的值。

结构应设计为将默认初始化状态视为有效状态。示例中

```
using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

用户定义实例构造函数仅在显式调用它的位置保护 null 值。如果 `KeyValuePair` 变量服从于默认值初始化，则 `key` 和 `value` 字段将为 null，并且该结构必须准备好处理此状态。

装箱和取消装箱

类类型的值可以转换为类型 `object` 或由类实现的接口类型，只需在编译时将引用视为另一种类型即可实现。同样，可以在不更改引用的情况下将类型的值 `object` 或接口类型的值转换回类类型(但在这种情况下，需要运行时类型检查)。

由于结构不是引用类型，因此对于结构类型，这些操作的实现方式有所不同。当结构类型的值转换为类型 `object` 或结构实现的接口类型时，将发生装箱操作。同样，当类型 `object` 或接口类型的值转换回结构类型时，将发生取消装箱操作。对类类型的相同操作的主要区别是装箱和取消装箱操作会将结构值复制到或传出装箱的实例。因此，在装箱或取消装箱操作后，对取消装箱的结构所做的更改不会反映在已装箱的结构中。

当结构类型重写从 `System.Object`（如 `Equals`、`GetHashCode` 或 `ToString`）继承的虚方法时，通过该结构类型的实例调用虚拟方法不会导致装箱发生。即使将结构用作类型参数，并且通过类型参数类型的实例进行调用，也是如此。例如：

```
using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main() {
        Test<Counter>();
    }
}
```

该程序的输出为：

```
1
2
3
```

尽管对 `ToString` 有副作用的样式是不正确的，但该示例演示了 `x.ToString()` 的三个调用没有发生装箱。

同样，在访问受约束的类型参数上的成员时，不会隐式进行装箱。例如，假设接口 `ICounter` 包含可用于修改值的方法 `Increment`。如果 `ICounter` 用作约束，则会调用 `Increment` 方法的实现，该方法将引用对其调用 `Increment` 的变量，而不是装箱副本。

```
using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();                    // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment();        // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}
```

对 `Increment` 的第一次调用将修改变量 `x` 中的值。这不等同于对 `Increment` 的第二次调用，后者修改 `x` 的装箱副本中的值。因此，该程序的输出为：

```
0
1
1
```

有关装箱和取消装箱的更多详细信息，请参阅装箱和取消装箱。

含义

在类的实例构造函数或实例函数成员内，`this` 分类为值。因此，虽然 `this` 可以用于引用为其调用函数成员的实例，但不能将分配给类的函数成员中 `this`。

在结构的实例构造函数中，`this` 对应于结构类型的 `out` 参数，在结构的实例函数成员内，`this` 对应于结构类型的 `ref` 参数。在这两种情况下，`this` 归类为变量，并且可以通过分配给 `this` 或将其作为 `ref` 或 `out` 参数传递来修改调用了函数成员的整个结构。

字段初始值设定项

如 "默认值" 中所述，结构的默认值由将所有值类型字段设置为其默认值，并将所有引用类型字段设置为 `null` 而得出的值。出于此原因，结构不允许实例字段声明包含变量初始值设定项。此限制仅适用于实例字段。允许结构的静态字段包含变量初始值设定项。

示例

```
struct Point
{
    public int x = 1;  // Error, initializer not permitted
    public int y = 1;  // Error, initializer not permitted
}
```

出现错误，因为实例字段声明包含变量初始值设定项。

构造函数

与类不同，结构不允许声明无参数的实例构造函数。相反，每个结构都隐式包含一个无参数的实例构造函数，该构造函数始终返回通过将所有值类型字段设置为其默认值并将所有引用类型字段设置为 null（默认构造函数）而产生的值。结构可声明具有参数的实例构造函数。例如

```
struct Point
{
    int x, y;

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

给定上述声明，语句

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

这两种方法都创建 `Point`，`x` 并 `y` 初始化为零。

不允许结构实例构造函数包含形式 `base(...)` 的构造函数初始值设定项。

如果结构实例构造函数未指定构造函数初始值设定项，则 `this` 变量对应于结构类型的 `out` 参数，类似于 `out` 参数，`this` 必须在构造函数返回的每个位置上明确赋值（明确赋值）。如果结构实例构造函数指定构造函数初始值设定项，则 `this` 变量对应于该结构类型的 `ref` 参数，类似于 `ref` 参数，在进入构造函数主体时，`this` 被视为已明确赋值。请考虑下面的实例构造函数实现：

```
struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;        // error, this is not yet definitely assigned
        Y = y;        // error, this is not yet definitely assigned
    }
}
```

在构造的所有字段都已明确赋值之前，无法调用实例成员函数（包括属性的 set 访问器 `X` 和 `Y`）。唯一的例外涉及自动实现的属性（自动实现的属性）。明确赋值规则（简单赋值表达式）专门在该结构类型的实例构造函数中将分配给结构类型的自动属性：此类赋值被视为对自动属性的隐藏支持字段的明确赋值。因此，可以使用以下内

容：

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
        X = x;      // allowed, definitely assigns backing field
        Y = y;      // allowed, definitely assigns backing field
    }
```

析构函数。

不允许结构声明析构函数。

静态构造函数

结构的静态构造函数遵循与类相同的大多数规则。在应用程序域中发生以下事件的第一个事件时，将触发结构类型的静态构造函数的执行：

- 引用结构类型的静态成员。
- 将调用结构类型的显式声明的构造函数。

创建结构类型的默认值（默认值）不会触发静态构造函数。（这是数组中元素的初始值的示例。）

结构示例

下面演示了两种使用 `struct` 类型创建类型的示例，这些类型可以与语言的预定义类型相似，但使用修改后的语义。

数据库整数类型

下面的 `DBInt` 结构实现一个整数类型，该类型可以表示 `int` 类型的完整值集，以及指示未知值的附加状态。具有这些特征的类型通常用于数据库。

```
using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();

    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.

    int value;
    bool defined;

    // Private instance constructor. Creates a DBInt with a known value.

    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }

    // The IsNull property is true if this DBInt represents an unknown value.

    public bool IsNull { get { return !defined; } }

    // The Value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.
```

```
public int Value { get { return value; } }

// Implicit conversion from int to DBInt.

public static implicit operator DBInt(int x) {
    return new DBInt(x);
}

// Explicit conversion from DBInt to int. Throws an exception if the
// given DBInt represents an unknown value.

public static explicit operator int(DBInt x) {
    if (!x.defined) throw new InvalidOperationException();
    return x.value;
}

public static DBInt operator +(DBInt x) {
    return x;
}

public static DBInt operator -(DBInt x) {
    return x.defined ? -x.value : Null;
}

public static DBInt operator +(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value + y.value: Null;
}

public static DBInt operator -(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value - y.value: Null;
}

public static DBInt operator *(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value * y.value: Null;
}

public static DBInt operator /(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value / y.value: Null;
}

public static DBInt operator %(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value % y.value: Null;
}

public static DBBool operator ==(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value == y.value: DBBool.Null;
}

public static DBBool operator !=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value != y.value: DBBool.Null;
}

public static DBBool operator >(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value > y.value: DBBool.Null;
}

public static DBBool operator <(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value < y.value: DBBool.Null;
}

public static DBBool operator >=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value >= y.value: DBBool.Null;
}

public static DBBool operator <=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value <= y.value: DBBool.Null;
}
```

```
    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        return defined? value.ToString(): "DBInt.Null";
    }
}
```

数据库布尔类型

下面的 `DBBool` 结构实现了三值逻辑类型。此类型的可能值为 `DBBool.True`、`DBBool.False` 和 `DBBool.Null`，其中 `Null` 成员指示未知值。这三值逻辑类型通常用于数据库。

```
using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or 1.

    DBBool(int value) {
        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.

    public bool IsNull { get { return value == 0; } }

    public bool IsFalse { get { return value < 0; } }

    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.

    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null, otherwise returns true or false.

    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.
```

```
    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }

    // Inequality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.

    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }

    // Logical negation operator. Returns True if the operand is False, Null
    // if the operand is Null, or False if the operand is True.

    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }

    // Logical AND operator. Returns False if either operand is False,
    // otherwise Null if either operand is Null, otherwise True.

    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }

    // Logical OR operator. Returns True if either operand is True, otherwise
    // Null if either operand is Null, otherwise False.

    public static DBBool operator |(DBBool x, DBBool y) {
        return new DBBool(x.value > y.value? x.value: y.value);
    }

    // Definitely true operator. Returns true if the operand is True, false
    // otherwise.

    public static bool operator true(DBBool x) {
        return x.value > 0;
    }

    // Definitely false operator. Returns true if the operand is False, false
    // otherwise.

    public static bool operator false(DBBool x) {
        return x.value < 0;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

数组

2020/11/2 • • Edit Online

数组是包含许多通过计算索引访问的变量的数据结构。一个数组，也称为数组的元素中包含的变量包括所有相同的类型，而这种类型称为数组的元素类型。

一个数组，具有用于确定与每个数组元素相关联的索引数的排名。数组的秩也称为数组的维数。调用具有一个级别的数组*维数组*。数组大于一项称为排名*多维数组*。特定大小的多维数组通常称为二维数组、三维数组等。

一个数组的每个维度具有一个关联的长度，它是大于或等于零的整数。维的长度不是数组，该类型的一部分，但在运行时创建的数组类型的实例时而建立。维度的长度决定了该维度的索引的有效范围：维度的长度 `N`，索引的范围可以从 `0` 到 `N - 1` 非独占。数组中元素的总数是数组中每个维度的长度的产品。如果一个或多个数组的维度的长度为零，则称该数组为空。

数组的元素类型可以是任意类型（包括数组类型）。

数组类型

数组类型被写为*non_array_type*跟一个或多个*rank_specifier*s:

```
array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;
```

一个*non_array_type*可以是任何*类型*，它是本身不*array_type*。

数组类型的秩的给定的最左侧*rank_specifier*中*array_type*:一个*rank_specifier*指示数组为排名数加一的数组"`,`"的令牌*rank_specifier*。

数组类型的元素类型是删除最左侧所得到的类型*rank_specifier*:

- 数组类型的窗体 `T[R]` 是带有秩的数组 `R` 和非数组元素类型 `T`。
- 数组类型的窗体 `T[R][R1]...[Rn]` 是带有秩的数组 `R` 和元素类型 `T[R1]...[Rn]`。

实际上，*rank_specifier*s 从左到右前的最后一个非数组元素类型读取。类型 `int[][,,][,]` 是一个一维数组的三维数组的二维数组的元素 `int`。

在运行时数组类型的值可以是 `null` 或对该数组类型的实例的引用。

System.Array 类型

类型 `System.Array` 是所有数组类型的抽象基类型。隐式引用转换 (隐式引用转换) 存在从任何数组类型设置为 `System.Array`，并显式引用转换 (显式引用转换) 存在从 `System.Array` 到任何数组类型。请注意，`System.Array` 本身不是*array_type*。相反，它是*class_type*所有*array_type*s 派生。

在运行时类型的值 `System.Array` 可以是 `null` 或对任何数组类型的实例的引用。

数组和泛型 IList 接口

一维数组 `T[]` 实现接口 `System.Collections.Generic.IList<T>` (`IList<T>` 简称) 及其基接口。相应地，没有隐式转换 `T[]` 到 `IList<T>` 及其基接口。此外，如果没有从的隐式引用转换 `S` 到 `T` 然后 `S[]` 实现 `IList<T>`，并且没有一种从隐式引用转换 `S[]` 到 `IList<T>` 及其基接口 (隐式引用转换)。如果没有显式引用转换从 `S` 到 `T` 则没有显式引用转换从 `S[]` 到 `IList<T>` 及其基接口 (显式引用转换). 例如：

```
using System.Collections.Generic;

class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;                    // Ok
        IList<string> lst2 = oa1;                   // Error, cast needed
        IList<object> lst3 = sa;                    // Ok
        IList<object> lst4 = oa1;                   // Ok

        IList<string> lst5 = (IList<string>)oa1;    // Exception
        IList<string> lst6 = (IList<string>)oa2;    // Ok
    }
}
```

赋值 `lst2 = oa1` 以来从转换将生成编译时错误 `object[]` 到 `IList<string>` 是不隐式的显式转换。强制转换 `(IList<string>)oa1` 将导致在运行时自引发异常 `oa1` 引用 `object[]` 而不 `string[]` 。但是该强制转换 `(IList<string>)oa2` 不会因为引发异常 `oa2` 引用 `string[]` 。

只要没有从的隐式或显式引用转换 `S[]` 到 `IList<T>` ，此外，还有一种从显式引用转换 `IList<T>` 及其基接口到 `S[]` (显式引用转换)。

当数组类型 `S[]` 实现 `IList<T>` ，某些实现的接口的成员可能会引发异常。接口的实现的确切行为已超出本规范的范畴。

数组创建

通过创建数组实例*array_creation_expression*s (数组创建表达式) 或由字段或局部变量声明包含*array_initializer*(数组初始值设定项)。

创建数组实例后，级别和每个维的长度建立和实例的整个生存期内保持不变。换而言之，不能更改现有数组实例，秩也不可能以调整其尺寸。

数组实例始终是数组类型。 `System.Array` 类型为抽象类型，不能实例化。

创建的数组的元素*array_creation_expression*s 始终将初始化为其默认值 (默认值)。

数组元素访问

使用访问数组元素*element_access*表达式 (数组访问) 的窗体 `A[I1, I2, ..., In]` ，其中 `A` 的数组类型和每个表达式 `Ix` 是类型的表达式 `int` ， `uint` ， `long` ， `ulong` ，或可以隐式转换为一个或多个这些类型。数组元素访问的结果是一个变量，即所选的索引的数组元素。

可以使用枚举数组的元素 `foreach` 语句 (foreach 语句)。

数组成员

每个数组类型继承的成员声明的 `System.Array` 类型。

数组协方差

对于任何两个*reference_types* `A` 并 `B`，则隐式引用转换 (隐式引用转换) 或显式引用转换 (显式引用转换) 中存在 `A` 到 `B`，然后从数组类型也存在相同的引用转换 `A[R]` 为数组类型 `B[R]`，其中 `R` 可以是任何给定*rank_specifier* (但相同数组类型)。这种关系称为***数组协方差***。数组协方差尤其意味着，数组类型的值 `A[R]` 实际上可能是对的数组类型实例的引用 `B[R]`，则从存在的隐式引用转换 `B` 到 `A`。

数组协方差，由于引用类型数组的元素赋值中包括一个运行时检查，以确保分配给数组元素的值确实是允许的类型 (简单的赋值)。例如：

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

分配给 `array[i]` 中 `Fill` 方法隐式包含一个运行时检查，以所引用的对象可确保 `value` 是 `null` 的实际元素类型与兼容实例或 `array`。在中 `Main` 的前两个调用 `Fill` 成功，但第三个调用会导致 `System.ArrayTypeMismatchException` 引发时执行的第一个分配到 `array[i]`。发生异常的一个装箱 `int` 不能存储在 `string` 数组。

数组协方差专门未扩展到的数组*value_types*。例如，不存在转换召开 `int[]` 作为要视作 `object[]`。

数组初始值设定项

可能在字段声明中指定数组初始值设定项 (字段)，本地变量声明 (局部变量声明)，和数组创建表达式 (数组创建表达式)：

```
array_initializer
    : '{' variable_initializer_list? '}'
    | '{' variable_initializer_list ',' '}'
    ;

variable_initializer_list
    : variable_initializer (',' variable_initializer)*
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

数组初始值设定项包含变量的初始值设定项，括在一系列 `{` "和" `}` "标记和分隔 `,` "令牌。每个变量的初始值设定项是一个表达式或对于多维数组，嵌套的数组初始值设定项。

在其中使用数组初始值设定项的上下文确定正在初始化数组的类型。在数组创建表达式中，数组类型立即之前初始值设定项，或从数组初始值设定项中的表达式推断。在字段或变量声明中，数组类型是字段或声明变量的类型。数组初始值设定项中使用时的字段或变量声明，如：

```
int[] a = {0, 2, 4, 6, 8};
```

它是简写为等效的数组创建表达式：

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

对于一维数组，数组初始值设定项必须包含一系列数组的元素类型与赋值兼容的表达式。表达式初始化按升序排列，从索引零开始的元素开始的数组元素。数组初始值设定项中的表达式数决定要创建的数组实例的长度。例如，上面的数组初始值设定项创建 `int[]` 长度为 5 的实例并初始化该实例使用以下值：

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

对于多维数组，数组初始值设定项必须尽可能多的嵌套数组中没有维度级别。最外层嵌套级别对应于最左边的维度，最内部的嵌套级别对应于最右边的维度。数组的每个维的长度确定数组初始值设定项中相应嵌套级别的元素数。对于每个嵌套的数组初始值设定项中，元素的数目必须与同一级别的其他数组初始值设定项相同。下面的示例：

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

创建一个二维数组的长度为 5 为最左侧的维度的两个最右边的维度的长度：

```
int[,] b = new int[5, 2];
```

然后初始化该数组实例设置以下值：

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

如果之外最右边的维度的长度为零，则假定后续维度还具有长度为零。下面的示例：

```
int[,] c = {};
```

使用长度为零的最左边和右边的维度中创建一个二维数组：

```
int[,] c = new int[0, 0];
```

当数组创建表达式中包含显式维的长度和数组初始值设定项时，长度必须是常量表达式，并在每个嵌套级别的元素的数目必须与相应的维度长度。下面是一些可能的恶意活动：

```
int i = 3;
int[] x = new int[3] {0, 1, 2};        // OK
int[] y = new int[i] {0, 1, 2};        // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};     // Error, length/initializer mismatch
```

此处的初始值设定项 `y` 会导致编译时错误，因为维度长度的表达式不是常量和的初始值设定项 `z` 导致编译时错误，因为长度和中的元素数初始值设定项不一致。

接口

2020/11/2 • • Edit Online

接口定义一个协定。类或结构实现的接口必须遵守其协定。一个接口可能从多个基接口继承，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。接口本身不提供实现，它定义的成员。接口只是指定必须由实现接口的类或结构提供的成员。

接口声明

*Interface_declaration*是*type_declaration* (类型声明)，其用于声明新的接口类型。

```
interface_declaration
    : attributes? interface_modifier* 'partial'? 'interface'
      identifier variant_type_parameter_list? interface_base?
      type_parameter_constraints_clause* interface_body ';'?
    ;
```

*Interface_declaration*包含一组可选*特性*(特性) 后，跟一组可选的*interface_modifier*s (接口修饰符) 后，跟一个可选 `partial` 修饰符，关键字后跟 `interface` 和一个*标识符*命名接口，跟一个可选*variant_type_parameter_list*规范 (变体类型的参数列表) 后，跟一个可选*interface_base*规范 (的基接口) 后，跟一个可选*type_parameter_constraints_clause*s 规范 (类型参数约束)后，跟*interface_body* (接口正文) 后，跟分号（可选）。

接口修饰符

*Interface_declaration*可以根据需要包含一系列接口修饰符：

```
interface_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | interface_modifier_unsafe
    ;
```

它是同一修饰符在接口声明中出现多次的编译时错误。

`new` 修饰符仅允许在类中定义的接口上。它指定该接口通过相同的名称，隐藏继承的成员中所述的新修饰符。

`public`， `protected`， `internal`，和 `private` 修饰符控制接口的可访问性。具体取决于接口声明发生的上下文，可能会允许仅在部分这些修饰符 (声明可访问性)。

Partial 修饰符

`partial` 修饰符指示这*interface_declaration*是分部类型声明。具有相同名称的封闭命名空间或类型声明中的多个分部接口声明组合到窗体一个接口声明，遵循的规则中指定分部类型。

变体类型参数列表

变体类型参数列表只能出现在接口和委托类型。与普通的差别*type_parameter_list*是可选的 s *variance_annotation*上每个类型参数。

```
variant_type_parameter_list
    : '<' variant_type_parameters '>'
    ;

variant_type_parameters
    : attributes? variance_annotation? type_parameter
    | variant_type_parameters ',' attributes? variance_annotation? type_parameter
    ;

variance_annotation
    : 'in'
    | 'out'
    ;
```

如果方差批注`out`，类型参数称为*协变*。如果方差批注`in`，类型参数称为***逆变***。如果没有任何变化批注，类型参数称***固定***。

在示例

```
interface C<out X, in Y, Z>
{
  X M(Y y);
  Z P { get; set; }
}
```

`X`为协变`Y`是逆变和`Z`是固定不变。

变体安全

方差批注类型形参列表中的一种类型的匹配项将限制类型的类型声明中可能发生的位置的位置。

一种类型`T`是***输出不安全***如果下列之一保存：

- `T`是逆变类型参数
- `T`为具有输出不安全的元素类型的数组类型
- `T`是一个接口或委托类型`S<A1,...,Ak>`从泛型类型构造`S<X1,...,Xk>`至少一个 where`Ai`以下之一保存：
  - `Xi`协变或固定和`Ai`是不安全的输出。
  - `Xi`是逆变或固定条件和`Ai`是输入安全的。

一种类型`T`是***输入不安全***如果下列之一保存：

- `T`是协变类型参数
- `T`是输入不安全的元素类型的数组类型
- `T`是一个接口或委托类型`S<A1,...,Ak>`从泛型类型构造`S<X1,...,Xk>`至少一个 where`Ai`以下之一保存：
  - `Xi`协变或固定和`Ai`是不安全的输入。
  - `Xi`是逆变或固定条件和`Ai`是不安全的输出。

直观地说，输出不安全类型禁止的中的输出位置，并在输入位置禁止的输入不安全类型。

一种类型是***输出安全***如果不是输出不安全，并***输入安全***如果不输入不安全。

变化转换

方差批注的目的是提供更宽松（但仍类型安全的）转换成接口和委托类型。到此结束的隐式定义 (隐式转换) 和显式转换 (显式转换) 进行可转换，按如下所示定义这一概念的使用：

一种类型`T<A1,...,An>`可变化转换为某种`T<B1,...,Bn>`如果`T`接口或委托类型使用变体类型参数声明`T<X1,...,Xn>`，并为每个变体类型参数`Xi`以下项之一包含：

- `Xi`是协变和隐式引用或标识转换存在从`Ai`到`Bi`

- `Xi` 是逆变和隐式引用或标识转换存在从 `Bi` 到 `Ai`
- `Xi` 是固定条件和标识转换存在从 `Ai` 到 `Bi`

基接口

接口可以继承自零个或多个接口类型，称为***显式基接口***的接口。时接口具有一个或多个显式基接口，然后在该接口的声明中的接口标识符后跟冒号和逗号分隔的基接口类型的列表。

```
interface_base
    : ':' interface_type_list
    ;
```

对于构造的接口类型，显式基接口构成方法为泛型类型声明中，对其执行显式基接口声明替换为每个*type_parameter*中的基接口相应的声明*type_argument*构造类型。

接口的显式基接口必须至少与接口本身相同的可访问性 (可访问性约束)。例如，它是指定的编译时错误 `private` 或 `internal` 接口中*interface_base*的 `public` 接口。

它是一个接口以直接或间接从自身继承的编译时错误。

的基接口已显式基接口并其基接口的接口。换而言之，组的基接口是完全的显式基接口，其显式基接口，诸如此类的可传递闭包。接口继承其基接口的所有成员。在示例

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

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

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

接口的基 `IComboBox` 都 `IControl`， `ITextBox`，和 `IListBox`。

换而言之，`IComboBox` 上述接口继承的成员 `SetText` 并 `SetItems` 以及 `Paint`。

每个基接口的接口必须是安全的输出 (变体安全)。类或结构还将隐式实现接口实现的所有接口的基接口。

接口体内

*Interface_body*接口的定义接口的成员。

```
interface_body
    : '{' interface_member_declaration* '}'
    ;
```

接口成员

接口的成员是继承自基接口的成员和成员声明由接口本身。

```
interface_member_declaration
    : interface_method_declaration
    | interface_property_declaration
    | interface_event_declaration
    | interface_indexer_declaration
    ;
```

接口声明可以声明零个或多个成员。接口成员必须是方法、属性、事件或索引器。接口不能包含常量、字段、运算符、实例构造函数、析构函数或类型，也不能包含任何类型的静态成员。

所有接口成员隐式都具有公共访问权限。它是接口成员声明中包含任何修饰符的编译时错误。具体而言，不能具有修饰符声明接口成员 `abstract`， `public`， `protected`， `internal`， `private`， `virtual`， `override`，或 `static`。

该示例

```
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; }
}
```

声明包含之一的每个可能类型的成员的接口：一种方法、属性、事件和索引器。

*Interface_declaration*创建一个新的声明空间 (声明)，并且*interface_member_declaration*s 立即包含*interface_declaration*引入此声明空间的新成员。以下规则适用于*interface_member_declaration*s:

- 方法的名称必须不同于所有属性和相同的接口中声明的事件的名称。此外，签名 (签名和超载) 的一种方法必须不同于同一接口中声明的所有其他方法的签名和签名不能在同一个接口中声明的两种方法，完全由不同 `ref` 和 `out`。
- 属性或事件的名称必须不同于在相同的接口中声明的所有其他成员的名称。
- 索引器的签名必须不同于同一接口中声明的所有其他索引器的签名。

接口的继承的成员是专门不属于该接口声明空间。因此，接口可以声明一个具有相同名称或签名为继承的成员的成员。当发生这种情况时，则称该派生的接口成员，若要隐藏基接口成员。隐藏继承的成员不被视为错误，但它确实会导致编译器发出警告。若要禁止显示警告，请在派生的接口成员的声明必须包括 `new` 修饰符以指示派生的成员用于隐藏基类成员。本主题讨论中进一步通过继承隐藏。

如果 `new` 修饰符包含在不隐藏继承的成员的声明，该结果发出警告。通过删除禁止显示此警告 `new` 修饰符。

请注意，在类成员 `object` 不严格地讲，任何接口的成员 (接口成员)。但是，在类成员 `object` 可通过成员查找中任何接口类型 (成员查找)。

接口方法

使用声明接口方法*interface_method_declaration*s:

```
interface_method_declaration
    : attributes? 'new'? return_type identifier type_parameter_list
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;
```

特性， *return_type*，*标识符*，以及*formal_parameter_list*的接口方法声明具有相同这意味着与方法声明在类中的 (方法)。接口方法声明不允许指定方法体中，并声明因此始终以分号结尾。

每种形式参数类型的接口方法必须是输入安全 (变体安全)，并返回类型必须为 `void` 或输出安全。此外，每个类类型约束、接口类型约束和类型参数约束上方法的任何类型参数必须是输入安全的。

这些规则可确保任何协变或逆变的接口的使用情况保持类型安全。例如，应用于对象的

```
interface I<out T> { void M<U>() where U : T; }
```

是非法的因为的使用情况 `T` 用作类型参数约束上 `U` 不是输入安全的。

已不存在此限制可能会违反类型安全方式如下：

```
class B {}
class D : B{}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

这是实际调用 `C.M<E>` 。但该调用需要 `E` 派生自 `D`，因此此处违反类型安全。

接口属性

接口属性的声明使用*interface_property_declaration*s:

```
interface_property_declaration
    : attributes? 'new'? type identifier '{' interface_accessors '}'
    ;

interface_accessors
    : attributes? 'get' ';'
    | attributes? 'set' ';'
    | attributes? 'get' ';' attributes? 'set' ';'
    | attributes? 'set' ';' attributes? 'get' ';'
    ;
```

特性，*类型*，并*标识符*接口属性声明的类中具有相同含义的属性声明 (属性)。

接口属性声明的访问器对应于类属性声明的访问器 (访问器)，只不过访问器正文必须始终是一个分号。因此，访问器只需指示属性是读写、只读的或只写。

接口属性的类型必须是输出安全是否存在 get 访问器，并且必须是输入安全如果没有 set 访问器。

接口事件

使用声明接口事件*interface_event_declaration*s:

```
interface_event_declaration
    : attributes? 'new'? 'event' type identifier ';'
    ;
```

特性，*类型*，并*标识符*接口事件声明的类中具有相同的事件声明的含义 (事件).

接口事件的类型必须是输入安全。

接口索引器

使用声明接口索引器*interface_indexer_declaration*s:

```
interface_indexer_declaration
    : attributes? 'new'? type 'this' '[' formal_parameter_list ']' '{' interface_accessors '}'
    ;
```

特性，*类型*，并*formal_parameter_list*接口索引器声明的类 (中具有与索引器声明相同的含义索引器)。

接口索引器声明的访问器对应于类索引器声明的访问器 (索引器)，只不过访问器正文必须始终是一个分号。因此，访问器只需指示索引器为读写、只读的或只写。

接口索引器的所有形式参数类型必须是输入安全。此外，任何 `out` 或 `ref` 形参类型还必须是安全的输出。请注意，甚至 `out` 参数需是输入安全的由于基础执行平台的限制。

接口索引器的类型必须是输出安全是否存在 get 访问器，并且必须是输入安全如果没有 set 访问器。

接口成员访问

成员访问通过访问接口成员 (成员访问) 和索引器访问 (索引器访问) 形式的表达式 `I.M` 并 `I[A]`，其中 `I` 是一个接口类型，`M` 是方法、属性或事件的接口类型，和 `A` 是索引器参数列表。

接口的仅限单一继承 (继承链中的每个接口都恰好零个或一个直接的基接口)、成员查找的效果 (成员查找)，方法调用 (方法调用)，和索引器访问 (索引器访问) 规则是完全与类和结构相同：派生成员隐藏较少派生的成员具有相同名称或签名。但是，对于多个继承的接口，二义性可能发生，当两个或多个不相关的基接口声明具有相同名称或签名的成员。本部分介绍这种情况下的几个示例。在所有情况下，可以使用显式强制转换来解决歧义。

在示例

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}

class C
{
    void Test(IListCounter x) {
        x.Count(1);                 // Error
        x.Count = 1;                // Error
        ((IList)x).Count = 1;       // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1);     // Ok, invokes ICounter.Count
    }
}
```

前两个语句会导致编译时错误，因为成员查找 (成员查找) 的 `Count` 中 `IListCounter` 不明确。如示例所示，通过强制转换来解决二义性 `x` 为适当的基接口类型。这种转换具有不运行时成本 — 它们只是包含的作为派生程度更小类型在编译时查看的实例。

在示例

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);               // Invokes IInteger.Add
        n.Add(1.0);             // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);   // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);    // Only IDouble.Add is a candidate
    }
}
```

在调用 `n.Add(1)` 中选择 `IInteger.Add` 通过应用的重载决策规则重载决策。同样的调用 `n.Add(1.0)` 选择 `IDouble.Add` 。插入了显式强制转换，就只有一个候选方法，因此没有歧义。

在示例

```
interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                 // Invokes ILeft.F
        ((IBase)d).F(1);        // Invokes IBase.F
        ((ILeft)d).F(1);        // Invokes ILeft.F
        ((IRight)d).F(1);       // Invokes IBase.F
    }
}
```

`IBase.F` 情况下隐藏成员 `ILeft.F` 成员。在调用 `d.F(1)` 因此选择 `ILeft.F` ，即使 `IBase.F` 似乎未隐藏，引导完成的访问路径中 `IRight` 。

多个继承接口中的隐藏的直观规则只是这样：如果在任何一个访问路径，则隐藏该成员，它是隐藏在所有的访问路径中。因为从的访问路径 `IDerived` 到 `ILeft` 到 `IBase` 隐藏 `IBase.F` ，从访问路径中还隐藏该成员 `IDerived` 到 `IRight` 到 `IBase` 。

完全限定的接口成员名称

接口成员有时会通过其***完全限定的名称***。接口成员的完全限定的名包含的名称的接口中的成员是声明，跟一个点，然后是的成员的名称。成员的完全限定的名称引用在其中声明成员的接口。例如，给定的声明

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

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

完全限定的名称`Paint`是`IControl.Paint`和完全限定的名称`SetText`是`ITextBox.SetText`。

在上面的示例中，它不能是指`Paint`作为`ITextBox.Paint`。

如果接口是命名空间的一部分，接口成员的完全限定的名包括命名空间名称。例如

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

此处，完全限定的名称`Clone`方法是`System.ICloneable.Clone`。

接口实现

可能由类和结构实现接口。若要指示类或结构直接实现一个接口，类或结构的基列表中包含的接口标识符。例如：

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

类或结构直接实现的接口直接隐式实现的所有接口的基接口。即使类或结构不会显式列出基类列表中的所有基接口，这是如此。例如：

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

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

class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

在这里，类 `TextBox` 同时实现 `IControl` 和 `ITextBox` 。

当类 `C` 直接实现一个接口，从 C 派生的所有类还隐式实现接口。在类声明中指定的基接口可以是构造的接口类型 (构造类型)。基接口不能是自身、类型参数，但它可能涉及到作用域中的类型参数。下面的代码演示如何对一个类可以实现和扩展构造的类型：

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

泛型类声明的基接口必须满足唯一性规则中所述实现的接口的唯一性。

显式接口成员实现代码

为了实现接口，类或结构可以声明*显式接口成员实现代码*。显式接口成员实现是引用完全限定的接口成员名称的方法、属性、事件或索引器声明。例如

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

这里 `IDictionary<int,T>.this` 和 `IDictionary<int,T>.Add` 是显式接口成员实现。

在某些情况下，接口成员的名称可能不是适用于实现的类，将在其中用例的接口成员可能会实现使用显式接口成员实现。类实现文件抽象，例如，可能会实现 `Close` 成员函数，释放文件资源的影响并实现了 `Dispose` 方法的 `IDisposable` 接口使用显式接口成员的实现：

```
interface IDisposable
{
    void Dispose();
}

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

不能通过在方法调用、属性访问或索引器访问其完全限定名称访问的显式接口成员实现。显式接口成员实现只能通过接口实例访问，并且在这种情况下引用只需通过其成员名称。

它是显式接口成员实现包括访问修饰符的编译时错误，它会导致编译时错误包含修饰符 `abstract`， `virtual`，`override`，或 `static`。

显式接口成员实现具有不同的可访问性特征与其他成员。由于显式接口成员实现代码永远不能通过在方法调用或属性访问其完全限定名称，它们是在专用的意义上。但是，由于它们可以通过将接口实例访问，因此可以在某种意义上也是公共。

显式接口成员实现代码有两个主要用途：

- 显式接口成员实现代码不是通过类或结构的实例可访问的因为它们允许接口实现，若要从公共接口的类或结构中排除。此功能特别有用，当类或结构实现此类或结构的使用者不感兴趣的内部接口。
- 显式接口成员实现允许消除具有相同签名的接口成员的歧义。没有显式接口成员实现代码是类或结构不可能具有不同的实现的接口具有相同签名的成员，并作为返回类型，它是类或结构不可能具有任何实现在所有接口成员具有相同的签名，但使用不同的返回类型。

有效的显式接口成员实现，类或结构必须命名其基类列表中包含其完全限定的名称、类型和参数类型完全匹配的显式接口成员的成员的接口实现。因此，在下面的类

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

声明 `IComparable.CompareTo` 会导致编译时错误，因为 `IComparable` 的基类列表中未列出 `Shape` 并不是基接口的 `ICloneable`。同样，在声明中

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

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

声明 `ICloneable.Clone` 中 `Ellipse` 导致编译时错误，因为 `ICloneable` 的基类列表中未显式列出 `Ellipse`。

接口成员的完全限定的名称必须引用在其中声明成员的接口。因此，在下面的声明

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

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

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

显式接口成员实现的 `Paint` 必须编写为 `IControl.Paint` 。

实现的接口的唯一性

实现泛型类型声明的接口必须保持唯一的所有可能的构造类型。如果没有此规则，将无法确定要调用的某些构造的类型的正确方法。例如，假设一个泛型类声明了允许按以下方式编写：

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>                    // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

如果允许这样，它无法确定要在以下情况下执行的代码：

```
I<int> x = new X<int,int>();
x.F();
```

若要确定泛型类型声明的接口列表是否有效，请执行以下步骤：

- 让 `L` 是直接在泛型类、结构或接口声明中指定的接口的列表 `C` 。
- 将添加到 `L` 任何基接口的接口已在 `L` 。
- 删除任何重复项 `L` 。
- 如果任何可能构造从创建类型 `C` 将类型参数会代入后 `L` ，会导致在两个接口 `L` 完全相同，然后的声明 `C` 无效。确定所有可能的构造的类型时，不会考虑约束声明。

在类声明中 `X` 接口列表上方 `L` 组成 `I<U>` 和 `I<V>` 。声明无效，因为任何构造具有类型 `U` 和 `V` 正在相同的类型会导致这两个接口是相同的类型。

此外，可以在不同的继承级别统一上指定的接口：

```
interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>    // Ok
{
    void I<V>.F() {...}
}
```

此代码是有效的即使 `Derived<U,V>` 同时实现 `I<U>` 和 `I<V>` 。代码

```
I<int> x = new Derived<int,int>();
x.F();
```

调用中的方法 `Derived` ，因为 `Derived<int,int>` 有效地重新实现 `I<int>` (接口重新实现)。

泛型方法的实现

当泛型方法隐式实现接口方法，提供每个方法类型参数必须是两个声明中的等效（在任何接口类型参数替换为适当的类型参数），其中的约束方法类型参数进行标识的序号位置，从左到右。

当泛型方法显式实现接口方法时，但是，实现方法上不允许使用任何约束。相反，约束被继承的接口方法

```
interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
    public void F<T>(T t) {...}                    // Ok
    public void G<T>(T t) where T: C {...}         // Ok
    public void H<T>(T t) where T: string {...}    // Error
}
```

该方法 `C.F<T>` 隐式实现 `I<object,C,string>.F<T>` 。在这种情况下， `C.F<T>` 不是需要（也不允许）来指定该约束 `T:object` 由于 `object` 是隐式所有类型参数约束。该方法 `C.G<T>` 隐式实现 `I<object,C,string>.G<T>` 由于约束匹配那些在界面中，在接口类型参数替换为相应类型参数之后。方法的约束 `C.H<T>` 是一个错误因为密封类型 (`string` 这种情况下) 不能用作约束。由于隐式接口方法实现的约束需要匹配省略该约束也将是一个错误。因此，就无法隐式实现 `I<object,C,string>.H<T>` 。仅可以使用显式接口成员实现来实现此接口方法：

```
class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;    // Ok
        H<T>(t);
    }
}
```

在此示例中，显式接口成员实现调用具有更严格弱约束的公共方法。请注意，从赋值 `t` 到 `s` 无效，因为 `T` 继承的约束 `T:string`，即使此约束不可表示源代码中。

接口映射

类或结构必须提供的接口的类或结构的基列表中列出的所有成员的实现。查找实现接口成员的实现的类或结构中的过程被称为***接口映射***。

类或结构的接口映射 `C` 为每个接口的基的类列表中指定的每个成员定位实现 `C`。特定的接口成员的实现 `I.M`，其中 `I` 是在其中的接口成员 `M` 声明，确定通过检查每个类或结构 `S`，从开始 `C` 和为每个后续基类重复 `C`，直到找到匹配项：

- 如果 `S` 包含声明的匹配的显式接口成员实现 `I` 并 `M`，则此成员是实现 `I.M`。
- 否则为如果 `S` 包含非静态公共成员相匹配的声明 `M`，则此成员是实现 `I.M`。如果多个匹配项有一个成员，它是未指定哪个成员是实现 `I.M`。这种情况下只能如果 `S` 为构造的类型，其中泛型类型中声明的两个成员具有不同的签名，但类型参数可以让它们的签名完全相同。

如果实现找不到指定的基类列表中的所有接口的所有成员会发生编译时错误 `C`。请注意接口的成员，包括那些继承自基接口的成员。

有关接口映射，类成员的含义 `A` 与接口成员匹配 `B` 时：

- `A` 并 `B` 方法，和名称、类型和形参表的 `A` 和 `B` 完全相同。
- `A` 和 `B` 是属性、名称和类型的 `A` 和 `B` 是相同的并且 `A` 具有相同的访问器作为 `B`（`A` 允许具有其他访问器，如果它不是显式接口成员实现）。
- `A` 并 `B` 是事件，以及名称和类型的 `A` 和 `B` 完全相同。
- `A` 和 `B` 是索引器、类型和形参列表 `A` 并 `B` 是相同的和 `A` 具有相同的访问器作为 `B`（`A` 允许具有其他访问器，如果不是显式接口成员实现）。

值得注意的接口映射算法含义是：

- 确定实现接口成员的类或结构成员时，显式接口成员实现代码优先于同一类或结构中的其他成员。
- 非公共既不静态成员参与接口映射。

在示例

```
interface ICloneable
{
    object Clone();
}

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

`ICloneable.Clone` 的成员 `C` 的实现变得 `Clone` 中 `ICloneable` 因为显式接口成员实现代码优先于其他成员。

如果类或结构实现两个或多个接口包含具有相同的名称、类型和参数类型的成员，则可以将每个到单个类或结构成员上这些接口成员。例如

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

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

在这里，`Paint` 两个方法 `IControl` 并 `IForm` 映射到 `Paint` 中的方法 `Page` 。当然也很可能有两个方法的单独的显式接口成员实现代码。

如果类或结构实现包含隐藏的成员的接口，然后某些成员必须通过显式接口成员实现来实现。例如

```
interface IBase
{
    int P { get; }
}

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

此接口的实现将需要至少一个显式接口成员实现，并将采用以下形式之一

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

当一个类实现多个具有相同的基接口的接口时，可以只有一个基接口的实现。在示例

```
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) {...}
}
```

不能具有单独实现 `IControl` 在基的类列表中，名为 `IControl` 由继承 `ITextBox`，和 `IControl` 由继承 `IListBox`。实际上，是标识的没有不同，这些接口的概念。相反，实现 `ITextBox` 和 `IListBox` 共享的相同实现 `IControl`，和 `ComboBox` 只需实现三个接口，被视为 `IControl`， `ITextBox`，和 `IListBox`。

基类的成员参与接口映射。在示例

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

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

该方法 `F` 中 `Class1` 中使用 `Class2` 的实现 `Interface1`。

接口实现继承

一个类继承其基类所提供的所有接口实现。

无需显式*重新实现*接口，派生的类不能以任何方式更改它从其基类继承的接口映射。例如，在声明中

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

class Control: IControl
{
    public void Paint() {...}
}

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

`Paint`中的方法`TextBox`隐藏`Paint`中的方法`Control`，但它不会更改的映射`Control.Paint`拖动到`IControl.Paint`，以及对调用`Paint`通过类实例和接口实例将具有以下效果

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

但是，当接口方法映射到类中的虚方法时，就可以为派生类重写虚拟方法并更改接口的实现。例如，重写到上面的声明

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

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

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

现在观察到以下影响

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

显式接口成员实现代码不能声明为虚拟函数，因为它不能重写显式接口成员实现。但是，它是完全有效的显式接口成员实现调用另一个方法，并且，其他方法可以声明为虚拟函数以允许派生类重写它。例如

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

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

在这里，类派生自`Control`可以专用化的实现`IControl.Paint`通过重写`PaintControl`方法。

接口重新实现

继承的接口实现的类允许***重新实现***通过基类列表中包含的接口。

接口重新实现遵循完全相同接口映射的规则的初始实现的接口。因此，继承的接口映射不起作用的接口映射上建立的重新实现的接口。例如，在声明中

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

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

这一事实，`Control`映射`IControl.Paint`拖动到`Control.IControl.Paint`不会影响在重新实现`MyControl`，映射`IControl.Paint`拖到`MyControl.Paint`。

继承的公共成员声明和继承的显式接口成员声明参与重新实现的接口的接口映射过程。例如

```
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() {}
}
```

此处，实现 `IMethods` 中 `Derived` 映射到的接口方法 `Derived.F`， `Base.IMethods.G`， `Derived.IMethods.H`，并 `Base.I`。

当类实现一个接口，它隐式还实现了所有该接口的基接口。同样，一个接口重新实现还隐式是重新实现的所有接口的基接口。例如

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}
```

此处，重新实现 `IDerived` 还重新实现 `IBase` 映射 `IBase.F` 拖动到 `D.F`。

抽象类和接口

就像一个非抽象类，抽象类必须提供的接口的类的基类列表中列出的所有成员的实现。但是，一个抽象类被允许接口方法映射到抽象方法。例如

```
interface IMethods
{
    void F();
    void G();
}

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

此处，实现 `IMethods` 映射 `F` 并 `G` 到抽象方法，其必须重写的派生自非抽象类中 `C` 。

请注意，显式接口成员实现代码不能为抽象的但当然允许显式接口成员实现调用抽象方法。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();
}
```

此处，派生自非抽象类 `C` 所需重写 `FF` 并 `GG` ，从而提供的实际实现 `IMethods` 。

枚举

2020/11/2 • • Edit Online

枚举类型是声明一组命名常量的非重复值类型(值类型)。

示例

```
enum Color
{
    Red,
    Green,
    Blue
}
```

使用成员 `Red`、`Green`和 `Blue`声明名为 `Color` 的枚举类型。

枚举声明

枚举声明声明一个新的枚举类型。枚举声明以关键字 `enum`开头，并定义枚举的名称、可访问性、基础类型和成员。

```
enum_declaration
    : attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
    ;

enum_base
    : ':' integral_type
    ;

enum_body
    : '{' enum_member_declarations? '}'
    | '{' enum_member_declarations ',' '}'
    ;
```

每个枚举类型都有一个对应的整型类型，称为枚举类型的***基础类型***。此基础类型必须能够表示枚举中定义的所有枚举器值。枚举声明可以显式声明基础类型的 `byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`long` 或 `ulong`。请注意，`char` 不能用作基础类型。不显式声明基础类型的枚举声明具有基础类型的 `int`。

示例

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

使用 `long`的基础类型声明枚举。开发人员可以选择使用基础类型的 `long`(如本示例所示)，以允许使用 `long` 范围内但不在 `int` 范围内的值，或保留此选项以供将来使用。

枚举修饰符

*Enum_declaration*可以选择性地包含一系列枚举修饰符：

```
enum_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;
```

同一修饰符在一个枚举声明中多次出现是编译时错误。

枚举声明的修饰符与类声明(类修饰符)的修饰符具有相同的含义。但请注意，在枚举声明中不允许使用 `abstract` 和 `sealed` 修饰符。枚举不能是抽象的并且不允许派生。

枚举成员

枚举类型声明的主体定义零个或多个枚举成员，这是枚举类型的命名常量。两个枚举成员不能具有相同的名称。

```
enum_member_declarations
    : enum_member_declaration (',' enum_member_declaration)*
    ;

enum_member_declaration
    : attributes? identifier ('=' constant_expression)?
    ;
```

每个枚举成员都有一个关联的常量值。此值的类型是包含枚举的基础类型。每个枚举成员的常数值必须在该枚举的基础类型的范围内。示例

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

导致编译时错误，因为 `-1` 、`-2` 和 `-3` 的常量值不在基础整型类型 `uint` 的范围内。

多个枚举成员可能共享相同的关联值。示例

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```

显示枚举，其中两个枚举成员-- `Blue` 和 `Max` 具有相同的关联值。

枚举成员的关联值是隐式或显式分配的。如果枚举成员的声明具有*constant_expression*初始值设定项，则该常量表达式的值将隐式转换为枚举的基础类型，这是枚举成员的关联值。如果枚举成员的声明没有初始值设定项，则将隐式设置其关联值，如下所示：

- 如果枚举成员是枚举类型中声明的第一个枚举成员，则其关联值为零。
- 否则，枚举成员的关联值将通过将每个按日的枚举成员的关联值增加一个来获得。这一增加的值必须在可由基础类型表示的值范围内，否则将发生编译时错误。

示例

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}

class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);

            case Color.Green:
                return String.Format("Green = {0}", (int) c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);

            default:
                return "Invalid color";
        }
    }
}
```

输出枚举成员名称及其关联值。输出为：

```
Red = 0
Green = 10
Blue = 11
```

原因如下：

- 枚举成员 `Red` 会自动赋予零值（因为它没有初始值设定项，并且是第一个枚举成员）；
- 为枚举成员 `Green` 显式赋予 `10` 值；
- 和枚举成员 `Blue` 会自动赋给一个值，其值大于在一个上按其进行的成员。

枚举成员的关联值不能直接或间接地使用其自己的关联枚举成员的值。除了此循环限制之外，枚举成员初始值设定项可以随意引用其他枚举成员初始值设定项，而不考虑其文本位置。在枚举成员初始值设定项中，其他枚举成员的值始终被视为具有其基础类型的类型，因此在引用其他枚举成员时不需要强制转换。

示例

```
enum Circular
{
    A = B,
    B
}
```

导致编译时错误，因为 `A` 和 `B` 的声明是循环的。`A` 显式依赖于 `B`，`B` 依赖于 `A` 隐式。

枚举成员的命名方式与类中的字段完全类似。枚举成员的作用域是其包含的枚举类型的正文。在该范围内，枚举成员可以通过其简单名称引用。对于所有其他代码，枚举成员的名称必须用其枚举类型的名称进行限定。枚举成员不具有任何已声明的可访问性-如果其包含枚举类型可访问，则枚举成员是可访问的。

System.object 类型

类型 `System.Enum` 是所有枚举类型的抽象基类(这是截然不同的，不同于枚举类型的基础类型)，并且从 `System.Enum` 继承的成员可用于任何枚举类型。从任何枚举类型到 `System.Enum` 都存在装箱转换(装箱转换)，并且从 `System.Enum` 到任何枚举类型都存在取消装箱转换(取消装箱转换)。

请注意，`System.Enum` 本身不是*enum_type*。相反，它是派生所有*enum_type*的*class_type* 。类型 `System.Enum` 从类型 `System.ValueType` (系统类型)继承而来，后者又继承自类型 `object` 。在运行时，可以 `null` 类型 `System.Enum` 的值或对任何枚举类型的装箱值的引用。

枚举值和运算

每个枚举类型都定义了一个不同的类型;若要在枚举类型和整型之间进行转换，或在两个枚举类型之间进行转换，必须使用显式枚举转换(显式枚举转换)。枚举类型可以采用的值集不受其枚举成员限制。特别是，枚举的基础类型的任何值都可以转换为枚举类型，并且是该枚举类型的非重复有效值。

枚举成员具有其包含枚举类型的类型(在其他枚举成员初始值设定项中除外：请参见枚举成员)。使用关联值 `v` 在枚举类型 `E` 中声明的枚举成员的值 `(E)v` 。

以下运算符可用于枚举类型的值： `==` 、 `!=` 、 `<` 、 `>` 、 `<=` 、 `>=` (枚举比较运算符)、二进制 `+` (加法运算符)、 `-` 、、 `^``&` (枚举逻辑运算符)、 `|` (按位求补运算符)、 `~` 和 `++` (后缀递增和递减运算符以及前缀增量和减量运算符)。 `--`

每个枚举类型都自动派生自类 `System.Enum` (反过来，派生自 `System.ValueType` 和 `object`)。因此，此类的继承方法和属性可用于枚举类型的值。

委托

2020/11/2 • • Edit Online

委托可实现其他语言(例如C++，Pascal 和 Modula)使用函数指针进行寻址的方案。但C++与函数指针不同，委托是完全面向对象的，不同C++于指向成员函数的指针，委托封装对象实例和方法。

委托声明定义派生自类 `System.Delegate` 的类。委托实例封装一个调用列表，它是一个或多个方法的列表，其中每个方法都称为可调用实体。对于实例方法，可调用实体包含实例和该实例上的方法。对于静态方法，可调用实体仅包含方法。使用适当的参数集调用委托实例会导致使用给定的参数集调用每个委托的可调用实体。

委托实例的一个有趣且有用的属性是它不知道或关心它所封装的方法的类;重要的是，这些方法与委托的类型兼容(委托声明)。这使委托非常适合用于 "匿名" 调用。

委托声明

*Delegate_declaration*是声明新委托类型的*type_declaration* (类型声明)。

```
delegate_declaration
    : attributes? delegate_modifier* 'delegate' return_type
      identifier variant_type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;

delegate_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | delegate_modifier_unsafe
    ;
```

同一修饰符在一个委托声明中多次出现是编译时错误。

仅允许在另一种类型中声明的委托上使用 `new` 修饰符，在这种情况下，它指定此类委托隐藏同名的继承成员，如new 修饰符中所述。

`public`、`protected`、`internal` 和 `private` 修饰符控制委托类型的可访问性。根据委托声明发生的上下文，可能无法使用其中某些修饰符(声明的可访问性)。

委托的类型名称为*标识符*。

可选*formal_parameter_list*指定委托的参数，*return_type*指示委托的返回类型。

可选*variant_type_parameter_list* (变体类型参数列表)指定委托本身的类型参数。

委托类型的返回类型必须是 `void` 或输出安全的(差异安全)。

委托类型的所有形参类型都必须为输入安全类型。此外，任何 `out` 或 `ref` 参数类型也必须是输出安全的。请注意，由于基础执行平台的限制，甚至 `out` 参数都需要是输入安全的参数。

中的C#委托类型是等效的名称，而不是结构上等效的。具体而言，具有相同参数列表和返回类型的两个不同的委托类型被视为不同的委托类型。不过，两个不同但结构等效的委托类型的实例可以比较为等于(委托相等运算符)。

例如：

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

`A.M1` 和 `B.M1` 的方法与委托类型 `D1` 和 `D2` 兼容，因为它们具有相同的返回类型和参数列表;不过，这些委托类型是两种不同的类型，因此它们不能互换。`B.M2`、`B.M3` 和 `B.M4` 方法与委托类型 `D1` 和 `D2` 不兼容，因为它们具有不同的返回类型或参数列表。

与其他泛型类型声明一样，必须提供类型参数来创建构造的委托类型。构造委托类型的参数类型和返回类型是通过将委托声明中的每个类型参数替换为构造委托类型的相应类型参数来创建的。生成的返回类型和参数类型用于确定哪些方法与构造的委托类型兼容。例如：

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

方法 `X.F` 与委托类型兼容 `Predicate<int>` 并且方法 `X.G` 与委托类型 `Predicate<string>` 兼容。

声明委托类型的唯一方法是通过 *delegate_declaration*。委托类型是派生自 `System.Delegate` 的类类型。委托类型是隐式 `sealed` 的，因此不允许从委托类型派生任何类型。也不允许从 `System.Delegate` 派生非委托类类型。请注意，`System.Delegate` 自身不是委托类型;它是派生所有委托类型的类类型。

C#提供委托实例化和调用的特殊语法。除实例化外，可以应用于类或类实例的任何操作也可以分别应用于委托类或实例。具体而言，可以通过常规成员访问语法来访问 `System.Delegate` 类型的成员。

委托实例封装的方法集称为调用列表。当从单个方法创建委托实例（委托兼容性）时，它会封装该方法，并且它的调用列表只包含一个条目。但是，如果两个非 null 委托实例组合在一起，则会将其调用列表串联在顺序左操作数和右操作数之间，以形成包含两个或多个条目的新调用列表。

使用二进制 `+`（加法运算符）和 `+=` 运算符（复合赋值）组合委托。可以使用二进制 `-`（减法运算符）和 `-=` 运算符（复合赋值）从委托组合中删除委托。可以比较委托是否相等（委托相等运算符）。

下面的示例演示了多个委托的实例化及其相应的调用列表：

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}

}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // M1
        D cd2 = new D(C.M2);      // M2
        D cd3 = cd1 + cd2;        // M1 + M2
        D cd4 = cd3 + cd1;        // M1 + M2 + M1
        D cd5 = cd4 + cd3;        // M1 + M2 + M1 + M1 + M2
    }

}
```

实例化 `cd1` 和 `cd2` 时，它们将封装一个方法。当 `cd3` 实例化时，它具有两个方法的调用列表，按该顺序 `M1` 和 `M2`。`cd4` 的调用列表按该顺序包含 `M1`、`M2` 和 `M1`。最后，`cd5` 的调用列表按该顺序包含 `M1`、`M2`、`M1`、`M1` 和 `M2`。有关组合（以及删除）委托的更多示例，请参阅委托调用。

委托兼容性

如果满足以下所有条件，则方法或委托 `M` 与委托类型 `D` ***兼容***：

- `D` 和 `M` 具有相同数量的参数，并且 `D` 中的每个参数都具有与 `out` 中的相应参数相同的 `ref` 或 `M` 修饰符。
- 对于每个值参数（不带 `ref` 或 `out` 修饰符的参数），都存在从 `D` 中的参数类型到 `M` 中的相应参数类型的标识转换（标识转换）或隐式引用转换（隐式引用转换）。
- 对于每个 `ref` 或 `out` 参数，`D` 中的参数类型与 `M` 中的参数类型相同。
- 存在从 `M` 的返回类型到 `D` 的返回类型的标识或隐式引用转换。

委托实例化

委托的实例是由*delegate_creation_expression*（委托创建表达式）或到委托类型的转换创建的。然后，新创建的委托实例指的是：

- *Delegate_creation_expression*中引用的静态方法，或
- *Delegate_creation_expression*或中引用的目标对象（不能是 `null`）和实例方法
- 其他委托。

例如：

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);        // static method
        C t = new C();
        D cd2 = new D(t.M2);        // instance method
        D cd3 = new D(cd2);        // another delegate
    }
}
```

实例化后，委托实例始终引用相同的目标对象和方法。请记住，当合并两个委托或从另一个委托移除时，将使用其自己的调用列表生成新的委托结果;合并或移除的委托的调用列表保持不变。

委托调用

C#提供调用委托的特殊语法。当调用列表包含一个项的非 null 委托实例调用时，它将调用一个方法，该方法具有给定的参数，并返回与被引用方法相同的值。（有关委托调用的详细信息，请参阅委托调用。）如果在调用此类委托的过程中发生异常，且未在调用的方法中捕获该异常，则会在调用委托的方法中继续搜索异常 catch 子句，就好像该方法直接调用了委托引用的方法一样。

调用列表中包含多个项的委托实例的调用将按顺序同步调用调用列表中的每个方法。调用的每个方法都被传递给为委托实例指定的相同的一组参数。如果此类委托调用包括引用参数（引用参数），则将发生每个方法调用，同时引用同一变量;调用列表中的方法对该变量所做的更改将对调用列表中的方法可见。如果委托调用包含输出参数或返回值，则其最终值将来自列表中最后一个委托的调用。

如果在处理此类委托的调用过程中发生异常，且未在调用的方法中捕获该异常，则会在调用委托的方法中继续执行异常 catch 子句的搜索，并使任何方法不会调用调用列表。

尝试调用值为 null 的委托实例会导致类型 `System.NullReferenceException` 的异常。

下面的示例演示如何实例化、组合、移除和调用委托：

```
using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);                // call M1

        D cd2 = new D(C.M2);
        cd2(-2);                // call M2

        D cd3 = cd1 + cd2;
        cd3(10);                // call M1 then M2

        cd3 += cd1;
        cd3(20);                // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);                // call M1, M2, M1, then M3

        cd3 -= cd1;             // remove last M1
        cd3(40);                // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);                // call M1 then M2

        cd3 -= cd2;
        cd3(60);                // call M1

        cd3 -= cd2;             // impossible removal is benign
        cd3(60);                // call M1

        cd3 -= cd1;             // invocation list is empty so cd3 is null

        cd3(70);                // System.NullReferenceException thrown

        cd3 -= cd1;             // impossible removal is benign
    }
}
```

如语句 `cd3 += cd1;` 中所示，一个委托可以在调用列表中多次出现。在这种情况下，它只会在每次出现时调用一次。在此类调用列表中，移除该委托时，调用列表中的最后一个匹配项将被实际删除。

在执行最后一个语句之前，`cd3 -= cd1;`，委托 `cd3` 引用一个空的调用列表。尝试从空列表中删除委托（或从非空列表中删除不存在的委托）不是错误。

生成的输出为：

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

Exceptions

2020/11/2 • • Edit Online

C# 中的异常提供结构化、统一的和类型安全方式来处理系统级别和应用程序级别的错误条件。中的异常机制C#是非常类似于C++，有几个重要的区别：

- 在 C# 中，必须由派生自类类型的实例表示的所有异常 `System.Exception` 。在C++，可以使用任何类型的任何值，表示该异常。
- 在 C# 中，finally 块 (try 语句) 可用于编写在正常执行和异常情况中执行的终止代码。此类代码都很难编写C++无需复制代码。
- 在 C# 中，系统级别的异常例如溢出、被零除和 null 取消引用有明确定义的异常类，等同于应用程序级别的错误条件。

引发异常的原因

可以在两种不同方法中引发异常。

- 一个 `throw` 语句 (throw 语句) 立即无条件地将引发异常。控件永远不会到达紧跟的语句 `throw` 。
- C# 语句和表达式的处理过程中出现特定异常情况时无法正常完成该操作，在某些情况下导致异常。例如，整数除法运算 (除法运算符) 将引发 `System.DivideByZeroException` 如果分母为零。请参阅常见的异常类有关可在这种方式中出现的各种异常情况的列表。

System.Exception 类

`System.Exception` 类是所有异常的基类型。此类具有共享的所有异常的几个值得注意的属性：

- `Message` 是只读的属性类型的 `string` ，其中包含异常的原因的人工可读说明。
- `InnerException` 是只读的属性类型的 `Exception` 。如果其值为非 null，它引用导致当前异常的异常 — 也就是说，在 catch 块中引发当前异常处理 `InnerException` 。否则，其值为 null，指示此异常不由另一个异常。这种方式中链接在一起的异常对象的数目可以是任意的。

可以对的实例构造函数的调用中指定这些属性的值 `System.Exception` 。

如何处理异常

通过处理异常 `try` 语句 (try 语句)。

异常发生时，系统将搜索最近 `catch` 可以处理该异常，由该异常的运行时类型的子句。首先，当前方法搜索的词法封闭 `try` 按顺序考虑语句和 try 语句相关联的 catch 子句。如果该操作失败，搜索调用当前方法的方法从词法上封闭 `try` 对当前方法的调用点的语句。此搜索将继续，直至 `catch` 子句找到能处理当前异常，通过命名的是同一个类或基类时，所引发的异常的运行时类型的异常类。一个 `catch` 子句未命名的异常类可以处理任何异常。

一旦找到匹配的 catch 子句，则系统将准备将控制转移到 catch 子句的第一个语句。Catch 子句的执行开始之前，系统首先，按顺序执行，任何 `finally` 子句与 try 语句更多相关联，嵌套的比捕获该异常。

如果不找到任何匹配的 catch 子句，则会发生两个条件之一：

- 如果对匹配的 catch 子句的搜索到达静态构造函数 (静态构造函数) 或静态字段初始值设定项，则 `System.TypeInitializationException` 触发调用静态构造函数的时候引发。内部异常的 `System.TypeInitializationException` 包含最初引发的异常。
- 如果搜索匹配的 catch 子句达到最初启动该线程的代码，则终止线程的执行。此类终止的影响是实现定义的。

析构函数执行过程中发生的异常是特别注意。如果析构函数在执行期间，会发生异常，并且不捕获该异常，然后终止该析构函数的执行并调用（如果有）的基类的析构函数。如果没有基类 (一样的情况下 `object` 类型) 或如果没有基类的析构函数，则异常将被丢弃。

常见的异常类

某些 C# 操作通过引发以下异常。

`System.ArithmeticException`	算术运算期间出现的异常的基类，例如 `System.DivideByZeroException` 和 `System.OverflowException`。
`System.ArrayTypeMismatchException`	当为一个数组的存储会失败，因为存储的元素的实际类型与数组的实际类型不兼容时引发。
`System.DivideByZeroException`	当尝试除以零的整数值时引发。
`System.IndexOutOfRangeException`	当尝试通过索引小于零或超出数组界限的数组编制索引时引发。
`System.InvalidCastException`	从基类或接口为派生类型的显式转换在运行时失败时引发。
`System.NullReferenceException`	时引发 `null` 使引用的对象是所需的方式使用引用。
`System.OutOfMemoryException`	当尝试分配内存时引发 (通过 `new`) 失败。
`System.OverflowException`	`checked` 上下文中的算术运算溢出时引发。
`System.StackOverflowException`	引发在执行堆栈用尽了的过多挂起的方法调用;通常指明可能出现了非常深度或无限递归。
`System.TypeInitializationException`	引发时，静态构造函数引发异常，但没有 `catch` 子句存在捕获它。

特性

2020/11/2 • • Edit Online

C# 语言的许多使程序员能够指定有关在程序中定义的实体的声明性信息。例如，在类中方法的可访问性通过修饰来指定其与*method_modifier*s `public`， `protected`， `internal`，并 `private`。

C# 允许程序员发明的声明性信息，名为的新类型***属性***。然后，程序员可以将属性附加到各种程序实体，并检索在运行时环境中的属性信息。例如，可以定义一个框架 `HelpAttribute` 可以放在特定程序元素（如类和方法）以提供从这些程序元素到其文档的映射的属性。

通过用于特性类的声明定义特性 (属性类)，这可以具有位置和命名参数 (位置和命名的参数)。特性附加到使用特性规范的 C# 程序中的实体 (属性规范)，并且可以作为属性实例在运行时检索 (特性实例)。

属性类

从抽象类派生的类 `System.Attribute`，无论直接或间接地，是***特性类***。特性类的声明类型的定义一个新***特性***可放置在声明。按照约定，特性类命名为后缀 `Attribute`。使用属性可以包括或省略此后缀。

特性用法

该属性 `AttributeUsage` (AttributeUsage 特性) 用于描述如何使用特性类。

`AttributeUsage` 具有位置参数 (位置和命名的参数) 使特性类来指定在其可以使用的声明的类型。该示例

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

定义一个名为属性类 `SimpleAttribute` 可在放置*class_declaration*s 和*interface_declaration*仅。该示例

```
[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

显示的几种用法 `Simple` 属性。尽管此特性定义的名称 `SimpleAttribute`，当使用此属性时，`Attribute` 可能会省略后缀，从而导致的短名称 `Simple`。因此，上面的示例在语义上等效于下面：

```
[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` 有一个命名的参数 (位置和命名的参数) 调用 `AllowMultiple`，指示是否可以多次指定该属性为给定实体。如果 `AllowMultiple` 类的属性为 true，则该特性类是***多用途特性类***，并且在实体上多次指定。如果 `AllowMultiple` 类的属性为 false 或是未指定，则该特性类是***一次性特性类***，并且在上一个实体最多一次指定。

该示例

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

定义一个名为的多用途特性类 `AuthorAttribute` 。该示例

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

演示了一个类声明的两种用法与 `Author` 属性。

`AttributeUsage` 具有名为的另一个命名的参数 `Inherited` ，指示是否属性，当指定为基类，也由派生自该基类的类继承。如果 `Inherited` 类的属性为 true，则继承该属性。如果 `Inherited` 类的属性为 false，则不会继承该属性。如果未指定，其默认值为 true。

特性类 `X` 不具有 `AttributeUsage` 属性附加到它，如

```
using System;

class X: Attribute {...}
```

等效于以下：

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

定位和命名参数

特性类可以具有***位置参数***并***命名参数***。每个公共实例构造函数，为该特性类定义的该特性类的位置参数是有效的序列。每个非静态公共读写字段和属性的特性类定义特性类的命名的参数。

该示例

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {        // Positional parameter
        ...
    }

    public string Topic {                     // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

定义一个名为属性类 `HelpAttribute` 它具有一个位置参数 `url`，和一个命名的参数，`Topic`。尽管非静态和 public，该属性，则 `Url` 不定义一个命名的参数，因为它不是读写。

可能按如下所示使用此特性类：

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

特性参数类型

为属性类的位置和命名参数的类型仅限于***属性参数类型***，它们是：

- 以下类型之一：`bool`，`byte`，`char`，`double`，`float`，`int`，`long`，`sbyte`，`short`，`string`，`uint`，`ulong`，`ushort`.
- `object` 类型。
- `System.Type` 类型。
- 枚举类型，提供具有公共可访问性，并在其中它嵌套（如果有）的类型还具有公共可访问性 (属性规范)。
- 上述类型的一维数组。
- 构造函数参数或公共字段不包含其中一种类型，不能用作特性规范中的位置或命名参数。

特性规范

属性规范是到声明此前定义的属性的应用程序。属性是一种声明的指定的其他声明性信息。可以在全局范围内（以指定包含程序集或模块上的属性）指定特性和有关 *type_declaration*s (类型声明)，*class_member_declaration*s (类型参数约束)，*interface_member_declaration*s (接口成员)，*struct_member_declaration*s (结构成员)，*enum_member_declaration*s (枚举成员)，*accessor_declarations* (访问器)，*event_accessor_declarations* (类似字段的事件)，和 *formal_parameter_list*s (方法参数)。

以指定属性***属性部分***。属性部分包含的一对方括号，环绕一个或多个属性的逗号分隔列表。在此类列表中，指定属性和在其中部分附加到同一个程序实体的顺序排列的顺序并不重要。例如，属性规范 `[A][B]`，`[B][A]`，`[A,B]`，和 `[B,A]` 是等效的。

```
global_attributes
    : global_attribute_section+
    ;

global_attribute_section
    : '[' global_attribute_target_specifier attribute_list ']'
    | '[' global_attribute_target_specifier attribute_list ',' ']'
    ;

global_attribute_target_specifier
    : global_attribute_target ':'
    ;

global_attribute_target
    : 'assembly'
    | 'module'
    ;

attributes
    : attribute_section+
    ;

attribute_section
    : '[' attribute_target_specifier? attribute_list ']'
    | '[' attribute_target_specifier? attribute_list ',' ']'
    ;

attribute_target_specifier
    : attribute_target ':'
    ;

attribute_target
    : 'field'
    | 'event'
    | 'method'
    | 'param'
    | 'property'
    | 'return'
    | 'type'
    ;

attribute_list
    : attribute (',' attribute)*
    ;

attribute
    : attribute_name attribute_arguments?
    ;

attribute_name
    : type_name
    ;

attribute_arguments
    : '(' positional_argument_list? ')'
    | '(' positional_argument_list ',' named_argument_list ')'
    | '(' named_argument_list ')'
    ;

positional_argument_list
    : positional_argument (',' positional_argument)*
    ;

positional_argument
    : attribute_argument_expression
    ;

named_argument_list
    : named_argument (','  named_argument)*
    ;
```

```
    ;

named_argument
    : identifier '=' attribute_argument_expression
    ;

attribute_argument_expression
    : expression
    ;
```

属性组成*attribute_name*和位置和命名参数的可选列表。位置参数（如果有）优先于命名的参数。位置自变量组成*attribute_argument_expression*; 命名的参数包含的名称后, 跟一个等号后, 跟*attribute_argument_expression*, 而后者在一起与简单赋值的相同规则约束。命名参数的顺序并不重要。

*Attribute_name*标识一个特性类。如果的窗体*attribute_name*是*type_name*，请参阅此名称必须是特性类。否则，将发生编译时错误。该示例

```
class Class1 {}

[Class1] class Class2 {}    // Error
```

因为它尝试使用会导致编译时错误`Class1`作为特性类时`Class1`不是特性类。

某些上下文中允许多个目标属性的规范。程序可以显式指定目标，通过包括*attribute_target_specifier*。当属性放置在全局级别*global_attribute_target_specifier*是必需的。在所有其他位置，将应用合理的默认值，但*attribute_target_specifier*可用于确认或重写中某些不明确的情况下的默认值（或只需确认中非不明确的情况下的默认值）。因此，通常情况下，*attribute_target_specifier*s 可以在全局级别省略除外。可能不明确的上下文已得到解决，如下所示：

- 在全局范围内指定的属性可以应用到目标程序集或目标模块。无默认值为此上下文中，因此存在*attribute_target_specifier*始终需要在此上下文中。是否存在`assembly` *attribute_target_specifier*指示该特性应用于目标程序集; 是否存在`module` *attribute_target_specifier*指示该属性应用到目标模块。
- 在委托声明上指定的属性可以应用到所声明的委托或其返回值。如果没有*attribute_target_specifier*，该属性应用于该委托。是否存在`type` *attribute_target_specifier*指示该特性应用于该委托; 是否存在`return` *attribute_target_specifier*指示该特性应用于返回值。
- 方法声明上指定的属性可以应用到所声明的方法或其返回值。如果没有*attribute_target_specifier*，该特性应用于该方法。是否存在`method` *attribute_target_specifier*指示该特性应用于方法; 是否存在`return` *attribute_target_specifier*指示该特性所应用于返回值。
- 在运算符声明上指定的属性可以应用到所声明的运算符或其返回值。如果没有*attribute_target_specifier*，则此属性应用于该运算符。是否存在`method` *attribute_target_specifier*指示该特性应用于该运算符; 是否存在`return` *attribute_target_specifier*指示该特性应用于返回值。
- 指定在事件声明的省略了事件访问器的一个特性可以应用于所声明的事件、关联的字段（如果该事件不是抽象的），或关联添加和删除方法。如果没有*attribute_target_specifier*，该特性应用于该事件。是否存在`event` *attribute_target_specifier*指示该特性应用于事件; 是否存在`field` *attribute_target_specifier*指示该属性应用于字段;是否存在`method` *attribute_target_specifier*指示该特性应用于方法。
- 在属性或索引器声明上的 get 访问器声明上指定的属性可以应用到关联的方法或其返回值。如果没有*attribute_target_specifier*，该特性应用于该方法。是否存在`method` *attribute_target_specifier*指示该特性应用于方法; 是否存在`return` *attribute_target_specifier*指示该特性所应用于返回值。
- Set 访问器属性或索引器声明上指定的属性可以应用到关联的方法或其单独的隐式参数。如果没有*attribute_target_specifier*，该特性应用于该方法。是否存在`method` *attribute_target_specifier*指示该特性应用于方法; 是否存在`param` *attribute_target_specifier*指示此特性应用于参数;是否存在`return` *attribute_target_specifier*指示该特性应用于返回值。
- 为事件声明可以应用于关联的方法或其单独的参数指定在 add 或 remove 访问器声明属性。如果没有*attribute_target_specifier*，该特性应用于该方法。是否存在`method` *attribute_target_specifier*指示该特性应用于方法; 是否存在`param` *attribute_target_specifier*指示此特性应用于参数;是否存在`return`

*attribute_target_specifier*指示该特性应用于返回值。

在其他上下文中，包含*attribute_target_specifier*允许但不必要。例如，类声明可能包括或省略说明符 `type`：

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

它是错误指定了一个无效*attribute_target_specifier*。例如，说明符 `param` 不能在类声明中使用：

```
[param: Author("Brian Kernighan")]       // Error
class Class1 {}
```

按照约定，特性类命名为后缀 `Attribute` 。*Attribute_name*窗体*type_name*可以包括或省略此后缀。如果使用或不带此后缀找到特性类，则产生歧义，且会导致编译时错误。如果*attribute_name*拼写，以便其右*标识符*是逐字字符串标识符 (标识符)，则只能将不带后缀的属性匹配时，从而使这种多义性会得到解决。该示例

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                     // Error: ambiguity
class Class1 {}

[XAttribute]            // Refers to XAttribute
class Class2 {}

[@X]                    // Refers to X
class Class3 {}

[@XAttribute]           // Refers to XAttribute
class Class4 {}
```

显示两个属性名为的类 `X` 和 `XAttribute` 。该属性 `[X]` 不明确，因为它可以为引用 `X` 或 `XAttribute` 。通过使用逐字字符串标识符，在这种极少数情况下指定确切的意图。该属性 `[XAttribute]` 不是不明确 (尽管它会很是否有一个名为属性类 `XAttributeAttribute` ！)。如果类的声明 `X` 被删除，则这两个属性引用名为的特性类 `XAttribute` ，按如下所示：

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                     // Refers to XAttribute
class Class1 {}

[XAttribute]            // Refers to XAttribute
class Class2 {}

[@X]                    // Error: no attribute named "X"
class Class3 {}
```

它是相同的实体超过一次使用单用途特性类的编译时错误。该示例

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

因为它尝试使用会导致编译时错误`HelpString`，它是一个单次使用的特性类，一次以上的声明上`Class1`。

一个表达式`E`是*attribute_argument_expression*如果满足所有以下语句：

- 类型`E`特性参数类型 (属性参数类型)。
- 在编译时，值`E`可以解析为以下值之一：
  - 常量的值。
  - 一个`System.Type`对象。
  - 一维数组*attribute_argument_expression*s。

例如：

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }

    public Type P2 {
        get {...}
        set {...}
    }

    public object P3 {
        get {...}
        set {...}
    }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

一个*typeof_expression* (typeof 运算符) 用作属性参数表达式可以引用非泛型类型、封闭式构造的类型或未绑定的泛型类型，但它不能引用开放类型。这是为了确保可以在编译时解析表达式。

```
class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;                  // Error, open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x;        // Ok, closed constructed type
    [A(typeof(List<>))] int y;           // Ok, unbound generic type
}
```

特性实例

特性实例是表示在运行时的特性的实例。属性是使用属性类，位置参数，定义和命名参数。一个特性实例是使用位置和命名参数初始化特性类的实例。

检索一个特性实例涉及编译时和运行时处理，如以下各节中所述。

属性的编译

编译*特性*特性类与 `T`，*positional_argument_list* `P` 并*named_argument_list* `N`，包括以下步骤：

- 请按照用于编译的编译时处理步骤*object_creation_expression*窗体的 `new T(P)`。这些步骤导致编译时错误，或确定实例构造函数 `C` 上 `T`，可以在运行时调用。
- 如果 `C` 不具有公共可访问性，则会发生编译时错误。
- 每个*named_argument* `Arg` 中 `N`：
  - 让 `Name` 进行*标识符*的*named_argument* `Arg`。
  - `Name` 必须标识的非静态读写公共字段或属性上 `T`。如果 `T` 有无此类字段或属性，则会发生编译时错误。
- 保留的该属性的运行时实例化的以下信息：特性类 `T`，实例构造函数 `C` 上 `T`，则*positional_argument_list* `P` 并*named_argument_list* `N`。

运行时检索的特性实例

编译*特性*产生一个属性类 `T`，实例构造函数 `C` 上 `T`、一个*positional_argument_list* `P`，和一个*named_argument_list* `N`。提供此信息，一个特性实例，可以检索在运行时使用以下步骤：

- 请按照用于执行运行时处理步骤*object_creation_expression*窗体 `new T(P)`，使用的实例构造函数 `C` 在编译时确定。这些步骤导致异常，或生成一个实例 `O` 的 `T`。
- 每个*named_argument* `Arg` 中 `N`，按顺序：
  - 让 `Name` 进行*标识符*的*named_argument* `Arg`。如果 `Name` 不会在标识的非静态公共读写字段或属性 `O`，则会引发异常。
  - 让 `Value` 进行的计算结果*attribute_argument_expression*的 `Arg`。
  - 如果 `Name` 标识的字段上 `O`，然后将此字段设置为 `Value`。
  - 否则为 `Name` 标识属性上 `O`。将此属性设置为 `Value`。
  - 结果是 `O`，特性类的实例 `T` 已初始化，*positional_argument_list* `P` 并*named_argument_list* `N`.

保留的属性

少量的属性会影响以某种方式的语言。这些属性包括：

- `System.AttributeUsageAttribute` (AttributeUsage 特性)，用于描述了可以在其中使用的特性类的方法。

- `System.Diagnostics.ConditionalAttribute` (Conditional 特性)，用于定义条件的方法。
- `System.ObsoleteAttribute` (Obsolete 特性)，用于将成员标记为已过时。
- `System.Runtime.CompilerServices.CallerLineNumberAttribute``System.Runtime.CompilerServices.CallerFilePathAttribute` 并 `System.Runtime.CompilerServices.CallerMemberNameAttribute` (调用方信息特性)，用于提供有关于可选参数的调用上下文的信息。

AttributeUsage 特性

该属性 `AttributeUsage` 用于描述可以在其中使用特性类的方式。

使用修饰的类 `AttributeUsage` 属性必须派生自 `System.Attribute`，直接或间接。否则，将发生编译时错误。

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly     = 0x0001,
        Module       = 0x0002,
        Class        = 0x0004,
        Struct       = 0x0008,
        Enum         = 0x0010,
        Constructor  = 0x0020,
        Method       = 0x0040,
        Property     = 0x0080,
        Field        = 0x0100,
        Event        = 0x0200,
        Interface    = 0x0400,
        Parameter    = 0x0800,
        Delegate     = 0x1000,
        ReturnValue  = 0x2000,

        All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
    }
}
```

Conditional 特性

该属性 `Conditional`，您可以***条件方法***并***条件特性***类。

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
    }
}
```

条件方法

一种方法使用修饰 `Conditional` 属性是有条件的方法。`Conditional` 属性指示条件通过测试的条件编译符号。对条件方法的调用是包含或省略具体取决于是否在调用时能够定义此符号。如果定义了符号，则将包括调用;否

则，省略（包括评估的接收方和调用的参数）的调用。

条件方法受到以下限制：

- 条件方法必须是中的方法*class_declaration*或*struct_declaration*。如果会发生编译时错误`Conditional`接口声明中的方法上指定属性。
- 条件方法必须具有返回类型为`void`。
- 条件方法必须未标有`override`修饰符。可以使用标记，条件方法`virtual`修饰符，但是。此类方法的重写是隐式条件元素，不能显式标记与`Conditional`属性。
- 条件方法不得接口方法的实现。否则，将发生编译时错误。

如果条件方法中使用，此外，发生编译时错误*delegate_creation_expression*。该示例

```
#define DEBUG

using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

声明`Class1.M`作为条件方法。`Class2``Test`方法调用此方法。由于条件编译符号`DEBUG`定义，如果`Class2.Test`是调用，它将调用`M`。如果符号`DEBUG`有未定义，然后`Class2.Test`不会调用`Class1.M`。

请务必注意，包含或排除条件方法调用受进行调用的条件编译符号。在示例

文件`class1.cs`：

```
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

文件`class2.cs`：

```
#define DEBUG

class Class2
{
    public static void G() {
        Class1.F();               // F is called
    }
}
```

文件 `class3.cs`:

```
#undef DEBUG

class Class3
{
    public static void H() {
        Class1.F();               // F is not called
    }
}
```

类 `Class2` 并 `Class3` 每个包含对条件方法的调用 `Class1.F`，这是有条件基于是否 `DEBUG` 定义。由于的上下文中定义此符号 `Class2` 但不是 `Class3`，在调用 `F` 中 `Class2` 包括，则在调用时 `F` 中 `Class3` 省略。

使用继承链中的条件方法可以是令人困惑。对通过条件方法的调用 `base`，窗体的 `base.M`，可能会有所条件的普通方法调用规则。在示例

文件 `class1.cs`:

```
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

文件 `class2.cs`:

```
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();                        // base.M is not called!
    }
}
```

文件 `class3.cs`:

```
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                           // M is called
    }
}
```

`Class2` 包含对的调用 `M` 其基类中定义。此调用省略，因为基方法是有条件根据的符号是否存在 `DEBUG`，这是未定义。因此，该方法将写入控制台" `Class2.M executed` "仅。明智地使用*pp_declaration*s 可以消除此类问题。

条件属性类

特性类 (属性类) 使用一个或多个修饰 `Conditional` 属性是*conditional 特性类*。Conditional 特性类是因此与相关联的条件编译符号中声明其 `Conditional` 属性。本示例：

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

声明 `TestAttribute` 作为条件属性类与条件编译符号 `ALPHA` 和 `BETA` 。

属性规范 (属性规范) 的条件属性都将包含如果一个或多个其关联的条件编译符号定义在规范中，否则该属性省略规范。

请务必注意，包含或排除条件特性类的属性规范受该规范所在位置的条件编译符号。在示例

文件 `test.cs` ：

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}
```

文件 `class1.cs` ：

```
#define DEBUG

[Test]               // TestAttribute is specified

class Class1 {}
```

文件 `class2.cs` ：

```
#undef DEBUG

[Test]                // TestAttribute is not specified

class Class2 {}
```

类 `Class1` 并 `Class2` 是每个使用属性修饰 `Test` ，这是有条件基于是否 `DEBUG` 定义。由于的上下文中定义此符号 `Class1` 但不是 `Class2` 的规范 `Test` 特性，可以在 `Class1` 包含，时的规范 `Test` 特性，可以在 `Class2` 省略。

Obsolete 特性

该属性 `Obsolete` 用于标记应不再使用的类型的类型和成员。

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}
```

如果程序使用的类型或成员，使用修饰 `Obsolete` 属性，则编译器会发出警告或错误。具体而言，编译器会发出警告如果没有提供任何错误参数，或如果错误参数提供了，并且具有值 `false` 。如果指定了错误参数，并且具有值，则编译器会发出错误 `true` 。

在示例

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
        A a = new A();         // Warning
        a.F();
    }
}
```

该类 `A` 用修饰 `Obsolete` 属性。每次使用都 `A` 在 `Main` 导致一条警告，包括指定的消息，"此类已过时;改为使用类 B。"

调用方信息属性

对于如日志记录和报告目的，是有时很有用的函数成员，才能获取有关调用代码的某些编译时信息。调用方信息特性提供一种方法来以透明方式传递此类信息。

一个可选参数，加上批注与调用方信息特性之一，省略的调用中的相应参数不一定会导致要替换的默认参数值。相反，如果有关调用上下文的指定的信息不可用，该信息将作为参数值传递。

例如：

```
using System.Runtime.CompilerServices

...

public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath]   string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line "+ line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}
```

调用 `Log()` 不带任何参数将打印行号和文件路径的调用，以及在其中发生在调用的成员的名称。

调用方信息特性上可以出现任意位置，可选参数包括在委托声明中。但是，特定的调用方信息特性限制对有可以属性的参数的类型，以便将始终从被替换的值为参数类型的隐式转换。

它是错误的同时定义和实现分部方法声明的一部分的参数具有相同的调用方信息属性。应用仅在定义的一部分的调用方信息特性，而忽略调用方信息特性只能在实现部分中发生。

调用方信息不会影响重载决策。重载决策特性化的可选参数仍从调用方的源代码中省略，因为它会忽略其他省略可选参数的方式相同忽略这些参数 (重载决策)。

在源代码中显式调用函数时，将仅替换调用方信息。例如隐式父构造函数调用的隐式调用没有源位置，将不替换调用方信息。此外，动态绑定的调用将不替换调用方信息。当调用方信息特性化的参数中这种情况下省略时，改为使用参数指定的默认值。

一个例外是查询表达式。这些地址被视为语法的扩展，并调用它们展开此项可以省略可选参数与调用方信息特性，如果调用方信息将被替换。使用的位置是在查询子句从生成调用的位置。

如果为给定的参数指定多个调用方信息特性，则它们首选按以下顺序：`CallerLineNumber`，`CallerFilePath`，`CallerMemberName`。

CallerLineNumber 属性

`System.Runtime.CompilerServices.CallerLineNumberAttribute` 标准的隐式转换时允许可选参数 (标准隐式转换) 的常量值从 `int.MaxValue` 到参数的类型。这可确保最多该值的任何非负行号，可以传递不会出错。

如果从源代码中的某个位置的函数调用中省略可选参数，`CallerLineNumberAttribute`，则作为而不是默认参数值调用的参数使用一个数值，表示该位置的行号。

如果调用跨多个行，选择的行是依赖于实现的。

请注意，可能会影响的行号 `#line` 指令 (行指令)。

CallerFilePath 属性

`System.Runtime.CompilerServices.CallerFilePathAttribute` 标准的隐式转换时允许可选参数 (标准隐式转换) 从 `string` 到参数的类型。

如果从源代码中的某个位置的函数调用中省略可选参数，`CallerFilePathAttribute`，则表示该位置的文件路径的字符串文字用作而不是默认参数值调用的参数。

文件路径的格式是依赖于实现的。

请注意，文件路径可能会受到 `#line` 指令 (行指令)。

CallerMemberName 属性

`System.Runtime.CompilerServices.CallerMemberNameAttribute` 标准的隐式转换时允许可选参数 (标准隐式转换) 从 `string` 到参数的类型。

如果函数调用的函数成员正文中或在特性中的位置从应用于的函数成员本身或其返回类型、参数或类型参数在源代码省略了可选参数，`CallerMemberNameAttribute`，则表示该成员的名称的字符串文字用作而不是默认参数值调用的参数。

调用泛型方法中发生的仅对方法名称本身使用，而无需的类型参数列表。

显式接口成员实现代码中发生的调用，只能使用方法名称本身，而无需上述接口限定。

对于属性或事件访问器内发生的调用，使用的成员名称是属性或事件本身。

有关索引器访问器内发生的调用，使用的成员名称是提供的`IndexerNameAttribute`(IndexerName 特性) 上的索引器成员，如果存在或默认名称`Item`否则为。

对于实例构造函数、静态构造函数、析构函数和运算符的声明中出现该成员的调用使用的名称是依赖于实现的。

互操作的特性

注意:本部分是仅适用于的 Microsoft.NET 实现的C#。

与 COM 和 Win32 组件互操作

.NET 运行时提供了大量的属性，可使 C# 程序可以与使用 COM 和 Win32 Dll 编写的组件进行互操作。例如，`DllImport`属性可用于`static extern`方法以指示要在 Win32 DLL 中找到该方法的实现。中找不到这些属性`System.Runtime.InteropServices`命名空间，以及为这些属性的详细的文档位于.NET 运行时文档。

与其他.NET 语言的互操作

IndexerName 特性

索引器中使用索引的属性的.NET 实现和.NET 元数据中具有的名称。如果没有`IndexerName`属性时，将提供索引器，然后是名称`Item`默认情况下使用。`IndexerName`属性使开发人员将覆盖此默认值并指定一个不同的名称。

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

不安全代码

2020/11/2 • • Edit Online

根据上述章节中定义的核心 c # 语言，在将指针省略为数据类型时，它的不同之处在于 C 和 c + +。相反，c # 提供了引用和创建由垃圾回收器管理的对象的功能。此设计与其他功能结合使用，使 c # 比 C 或 c + + 更安全。在核心 c # 语言中，不能使用未初始化的变量、"无关联的" 指针或索引超出其界限的数组的表达式。这样就消除了经常灾难 C 和 c + + 程序的各种 bug。

尽管 C 或 c + + 中的每个指针类型构造实际上都具有与 c # 对应的引用类型，但在某些情况下，访问指针类型是必需的。例如，在不访问指针的情况下，与基础操作系统、访问内存映射设备或实现时间关键算法之间的交互可能是不可能的或不切实际的。为了满足这一需要，c # 提供了编写***不安全代码***的功能。

在不安全代码中，可以声明和操作指针、在指针和整型之间执行转换，以获取变量的地址等。在某种意义上，编写不安全代码非常类似于在 c # 程序中编写 C 代码。

从开发人员和用户的角度来看，不安全代码实际上是一项 "安全" 功能。必须使用修饰符清楚地标记不安全的代码 `unsafe`，因此开发人员不可能意外地使用不安全的功能，并且执行引擎可以确保不受信任的环境中不能执行不安全的代码。

不安全上下文

C # 的 unsafe 功能只能在不安全的上下文中使用。通过 `unsafe` 在类型或成员的声明中包含修饰符或使用*unsafe_statement*，引入了不安全的上下文：

- 类、结构、接口或委托的声明可能包括 `unsafe` 修饰符，在这种情况下，该类型声明的整个文本范围（包括类、结构或接口的正文）被视为不安全的上下文。
- 字段、方法、属性、事件、索引器、运算符、实例构造函数、析构函数或静态构造函数的声明可能包含 `unsafe` 修饰符，在这种情况下，该成员声明的整个文本范围被视为不安全的上下文。
- *Unsafe_statement*允许在*块*内使用不安全的上下文。相关*块*的整个文本范围被视为不安全的上下文。

相关的语法生产如下所示。

```
class_modifier_unsafe
    : 'unsafe'
    ;

struct_modifier_unsafe
    : 'unsafe'
    ;

interface_modifier_unsafe
    : 'unsafe'
    ;

delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
    : 'unsafe' block
    ;
```

示例中

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

`unsafe`结构声明中指定的修饰符导致结构声明的整个文本区成为不安全的上下文。因此，可以将 `Left` 和 `Right` 字段声明为指针类型。上面的示例也可以编写

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

此处， `unsafe` 字段声明中的修饰符将导致这些声明被视为不安全的上下文。

除了建立不安全的上下文，因而允许使用指针类型外，修饰符对 `unsafe` 类型或成员不起作用。示例中

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}

public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}
```

`unsafe`中的方法的修饰符 `F` `A` 只是导致的文本范围 `F` 成为不安全的上下文，在此上下文中，可以使用该语言的不安全功能。在的重写 `F` 中 `B` ，无需重新指定 `unsafe` 修饰符--除非当然， `F` 方法 `B` 本身需要访问不安全功能。

当指针类型是方法签名的一部分时，情况会略有不同。

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

此处，由于 `F` 的签名包含指针类型，因此只能在不安全的上下文中写入。但是，通过使整个类不安全（如中的情况） `A` 或在 `unsafe` 方法声明中包含修饰符（如中的情况），可以引入 unsafe 上下文 `B` 。

指针类型

在不安全的上下文中，*类型*(类型)可能是*pointer_type*以及*value_type*或*reference_type*。但是，也*pointer_type*可以在 `typeof` 不安全的上下文之外的表达式(匿名对象创建表达式)中使用 pointer_type。

```
type_unsafe
    : pointer_type
    ;
```

*Pointer_type*以*unmanaged_type*或关键字形式编写 `void` ，后跟一个 `*` 令牌：

```
pointer_type
    : unmanaged_type '*'
    | 'void' '*'
    ;

unmanaged_type
    : type
    ;
```

`*` 在指针类型中之前指定的类型称为指针类型的***引用类型***。它表示指针类型值指向的变量的类型。

不同于引用(引用类型的值)时，不会由垃圾回收器跟踪指针-垃圾回收器不知道指针及其指向的数据。出于此原因，不允许指针指向引用或包含引用的结构，并且指针的引用类型必须是*unmanaged_type*。

*Unmanaged_type*是*reference_type*或构造类型之外的任何类型，并且在任何嵌套级别都不包含*reference_type*或构造类型字段。换句话说，*unmanaged_type*是以下项之一：

- `sbyte` 、 `byte` 、 `short` 、 `ushort` 、 `int` 、 `uint` 、 `long` 、 `ulong` `char` `float` `double` `decimal` 、、、、或 `bool` 。
- 任何*enum_type*。
- 任何*pointer_type*。
- 任何不是构造类型并且只包含*unmanaged_type*的字段的用户定义的*struct_type* 。

混合使用指针和引用的直观规则是，引用(对象)的引用(对象)允许包含指针，但指针的引用不允许包含引用。

下表给出了指针类型的一些示例：

“	“
`byte*`	指向 `byte`
`char*`	指向 `char`
`int**`	指向指向的指针的指针 `int`
`int*[]`	指向的指针的一维数组 `int`
`void*`	指向未知类型的指针

对于给定的实现，所有指针类型都必须具有相同的大小和表示形式。

与 C 和 c + + 不同，在 c # 中声明多个指针时， `*` 仅与基础类型一起编写，而不是作为每个指针名称的前缀标点符号。例如：

```
int* pi, pj;    // NOT as int *pi, *pj;
```

具有 type 的指针的值 `T*` 表示类型的变量的地址 `T` 。指针间接寻址运算符 `*` （指针间接寻址）可用于访问此变量。例如，给定一个 `P` 类型为的变量 `int*` ，该表达式 `*P` 表示在 `int` 中包含的地址处找到的变量 `P` 。

与对象引用类似，指针可以为 `null` 。将间接寻址运算符应用于 `null` 指针将导致实现定义的行为。具有值的指针用 `null` 全部位0表示。

`void*` 类型表示指向未知类型的指针。由于引用类型未知，因此间接寻址运算符不能应用于类型的指针 `void*` ，也不能对此类指针执行任何算术运算。但类型的指针 `void*` 可以转换为任何其他指针类型（反之亦然）。

指针类型是一种单独的类型类别。与引用类型和值类型不同，指针类型不从 `object` 继承，并且指针类型与之间不存在转换 `object` 。特别是，指针不支持装箱和取消装箱（装箱和取消装箱）。但允许在不同指针类型之间以及指针类型和整型之间进行转换。指针转换中对此进行了说明。

Pointer_type 不能用作类型参数（构造类型），并且类型推理（类型推理）对泛型方法调用失败，而这种方法调用会将类型参数推断为指针类型。

Pointer_type 可以用作可变字段（可变字段）的类型。

尽管指针可以作为 `ref` 或参数传递 `out` ，但这样做可能会导致未定义的行为，因为在调用的方法返回时，指针可能设置为指向某个局部变量，而在被调用的方法返回时，或其所指向的固定对象不再是固定的。例如：

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;

        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }

    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                *px1, *px2);    // undefined behavior
        }
    }
}
```

方法可以返回某种类型的值，并且该类型可以是指针。例如，当给定一个指向连续序列的指针 `int` 、序列的元素计数和其他某个 `int` 值时，如果发生匹配，以下方法将返回该序列中的该值的地址；否则返回 `null` ：

```
unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}
```

在不安全的上下文中，有多个构造可用于对指针进行操作：

- `*` 运算符可用于执行指针间接寻址（指针间接寻址）。
- `->` 运算符可用于通过指针（指针成员访问）访问结构的成员。
- `[]` 运算符可用于对指针（指针元素访问）进行索引。
- `&` 运算符可用于获取变量的地址（运算符的地址）。
- `++` 和 `--` 运算符可用来递增和递减指针（指针递增和递减）。
- `+` 和 `-` 运算符可用于执行指针算法（指针算法）。
- 、、、、 `==` `!=` `<` `>` `<=` 和运算符可 `>=` 用于比较指针（指针比较）。
- `stackalloc` 运算符可用于从调用堆栈（固定大小缓冲区）分配内存。
- `fixed` 语句可用于暂时修复变量，以便可以获取其地址（fixed 语句）。

固定和可移动变量

Address 运算符（address 运算符）和 `fixed` 语句（fixed 语句）将变量分为两类：***固定变量*** 和 ***可移动变量***。

固定变量驻留在不受垃圾回收器操作影响的存储位置中。（固定变量的示例包括局部变量、值参数和通过取消引用指针而创建的变量。）另一方面，可移动变量驻留在由垃圾回收器进行重定位或处置的存储位置。（可移动变量的示例包括对象中的字段和数组的元素。）

`&` 运算符（地址为 "运算符"）允许在不限制的情况下获取固定变量的地址。但是，由于可移动变量可能会由垃圾回收器重定位或处置，因此只能使用 `fixed` 语句（fixed 语句）获取可移动变量的地址，并且该地址在该语句的持续时间内保持有效 `fixed` 。

确切地说，固定变量是以下项之一：

- 由引用本地变量或值参数的*simple_name*（简单名称）产生的变量，除非该变量是由匿名函数捕获的。
- 由形式的*member_access*（成员访问）产生的变量 `V.I` ，其中 `V` 是*struct_type*的固定变量。
- 由窗体的*pointer_indirection_expression*（指针间接寻址） `*P` 、窗体的*pointer_member_access*（指针成员访问） `P->I` 或窗体的*pointer_element_access*（指针元素访问）生成的变量 `P[E]` 。

所有其他变量归类为可移动变量。

请注意，静态字段归类为可移动变量。另请注意， `ref` 或 `out` 参数归类为可移动变量，即使为参数提供的参数是固定变量。最后请注意，通过取消引用指针生成的变量始终归类为固定变量。

指针转换

在不安全的上下文中，可以使用隐式转换集（隐式转换）进行扩展，以包括以下隐式指针转换：

- 从任何*pointer_type*到类型 `void*` 。
- 从 `null` 文本到任何*pointer_type*。

此外，在不安全的上下文中，可使用的显式转换集（显式转换）进行扩展，以包括以下显式指针转换：

- 从任何*pointer_type*到任何其他*pointer_type*。

- 从 `sbyte` 、 `byte` 、、、、、 `short` `ushort` `int` `uint` `long` 或 `ulong` 到任何*pointer_type*。
- 从任何*pointer_type*到、、、、、、 `sbyte` `byte` `short` `ushort` `int` `uint` `long` 或 `ulong` 。

最后，在不安全的上下文中，标准隐式转换集(标准隐式转换)包括以下指针转换：

- 从任何*pointer_type*到类型 `void*` 。

两个指针类型之间的转换决不会更改实际指针的值。换句话说，从一种指针类型到另一种类型的转换不会影响由指针提供的基础地址。

当一种指针类型转换为另一种类型时，如果结果指针没有正确对齐所引用的类型，则该行为是不确定的。通常，"正确对齐" 这一概念是可传递的：如果指向类型的指针 `A` 正确地对齐了指向类型的指针，而后者又为指向类型的指针正确对齐，则指向该 `B` 类型的 `C` 指针 `A` 会正确对齐指向类型的指针 `C` 。

请考虑以下情况：通过指向其他类型的指针访问具有一种类型的变量：

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;         // undefined
*pi = 123456;        // undefined
```

将指针类型转换为指向字节的指针时，结果将指向该变量的最低寻址字节。结果的后续增量，最大为变量的大小，产生指向该变量的其余字节的指针。例如，下面的方法以十六进制值的形式显示 double 中的八个字节中的每个字节：

```
using System;

class Test
{
    unsafe static void Main() {
      double d = 123.456e23;
        unsafe {
           byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
               Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

当然，生成的输出取决于 endian。

指针和整数之间的映射是实现定义的。但是，在带有线性地址空间的 32 * 和64位 CPU 体系结构上，转换到整数类型或从整型转换到整数类型的方式通常与 `uint` `ulong` 在这些整型类型之间的转换或值完全相同。

指针数组

在不安全的上下文中，可以构造指针的数组。对于指针数组，只允许使用某些适用于其他数组类型的转换：

- 从任何*array_type*到的隐式引用转换(隐式引用转换) `System.Array` 以及它实现的接口也适用于指针数组。但是，只要尝试通过或其实现的接口访问数组元素，就 `System.Array` 会导致运行时出现异常，因为指针类型不能转换为 `object` 。
- 隐式和显式引用转换(隐式引用转换、显式引用转换)(从一维数组类型 `S[]` 到 `System.Collections.Generic.IList<T>` 它的泛型基接口)从不适用于指针数组，因为指针类型不能用作类型参数，并且不能从指针类型转换为非指针类型。
- 显式引用转换(显式引用转换) `System.Array` 和它为任何*array_type*实现的接口都适用于指针数组。
- 从的显式引用转换(显式引用转换) `System.Collections.Generic.IList<S>` 和其基接口到一维数组类型 `T[]`

永远不会应用于指针数组，因为指针类型不能用作类型参数，并且不能从指针类型转换为非指针类型。

这些限制意味着 `foreach` foreach 语句中所述的对数组的语句扩展不能应用于指针数组。相反，窗体的 foreach 语句

```
foreach (V v in x) embedded_statement
```

其中，的类型 `x` 为形式的数组类型 `T[,,...,]`， `N` 为维数减1， `T` 或 `V` 为指针类型，使用嵌套的 for 循环展开，如下所示：

```
{
    T[,,...,] a = x;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
        V v = (V)a.GetValue(i0,i1,...,iN);
        embedded_statement
    }
}
```

变量 `a` （ `i0` ， `i1` ...）对 `iN` `x` 或*embedded_statement*或程序的任何其他源代码都不可见或可访问。此变量 `v` 在嵌入语句中是只读的。如果没有从(元素类型)到的显式转换(指针转换) `T` `V` ，则会生成错误，并且不会执行任何其他步骤。如果 `x` 具有值 `null` ， `System.NullReferenceException` 则会在运行时引发。

表达式中的指针

在不安全的上下文中，表达式可能产生指针类型的结果，但在不安全的上下文外部，表达式是指针类型的编译时错误。在不安全的上下文中，如果任何*simple_name*（简单名称）、*member_access*（成员访问）、*invocation_expression*（调用表达式）或*element_access*（元素访问）是指针类型，则会发生编译时错误。

在不安全的上下文中，*primary_no_array_creation_expression*（主表达式）和*unary_expression*（一元运算符）生产允许以下附加构造：

```
primary_no_array_creation_expression_unsafe
    : pointer_member_access
    | pointer_element_access
    | sizeof_expression
    ;

unary_expression_unsafe
    : pointer_indirection_expression
    | addressof_expression
    ;
```

以下各节介绍了这些构造。语法暗示了 unsafe 运算符的优先级和关联性。

指针间接

*Pointer_indirection_expression*包含一个星号(), `*` 后跟一个*unary_expression*。

```
pointer_indirection_expression
    : '*' unary_expression
    ;
```

一元 `*` 运算符表示指针间接寻址，用于获取指针所指向的变量。计算结果 `*P` （其中 `P` 是指针类型的表达式） `T*` 是类型的变量 `T` 。将一元 `*` 运算符应用到类型的表达式 `void*` 或不是指针类型的表达式时，会发生编译

时错误。

向指针应用一元运算符的效果 `*` `null` 是实现定义的。特别是，不能保证此操作引发 `System.NullReferenceException` 。

如果向指针分配了无效的值，则一元运算符的行为 `*` 是不确定的。用于通过一元运算符取消引用指针的无效值中的 `*` 地址不能为所指向的类型（请参阅指针转换中的示例）和变量在生存期结束后的地址进行不当调整。

出于明确赋值分析的目的，通过对窗体的表达式进行计算而生成的变量 `*P` 被视为初始赋值（最初分配的变量）。

指针成员访问

*Pointer_member_access*包含一个*primary_expression*，后跟一个 " `->` " 标记，后跟一个*标识符*和一个可选的*type_argument_list*。

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

在窗体的指针成员访问中 `P->I` ， `P` 必须是指针类型的表达式，而不是 `void*` ，并且 `I` 必须表示指向的类型的可访问成员 `P` 。

窗体的指针成员访问权限 `P->I` 完全相同 `(*P).I` 。有关指针间接寻址运算符 () 的说明 `*` ，请参阅指针间接寻址。有关成员访问运算符 () 的说明 `.` ，请参阅成员访问。

示例中

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

`->` 运算符用于通过指针访问字段和调用结构的方法。由于此操作 `P->I` 完全等效于 `(*P).I` ，因此，该 `Main` 方法的编写方法可能同样适用：

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

指针元素访问

*Pointer_element_access*包含一个*primary_no_array_creation_expression*后跟一个括在 " `[` " 和 "" 中的表达式 `]` 。

```
pointer_element_access
    : primary_no_array_creation_expression '[' expression ']'
    ;
```

在窗体的指针元素访问中 `P[E]` ， `P` 必须是指针类型的表达式，而不是 `void*` ，并且 `E` 必须是可隐式转换为 `int` 、 `uint` 、或的表达式 `long` `ulong` 。

对窗体的指针元素的访问权限 `P[E]` 完全相同 `*(P + E)` 。有关指针间接寻址运算符()的说明 `*` ，请参阅指针间接寻址。有关指针加法运算符()的说明 `+` ，请参阅指针算法。

示例中

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

指针元素访问用于在循环中初始化字符缓冲区 `for` 。由于此操作 `P[E]` 完全等效于 `*(P + E)` ，因此，该示例也可以正确编写：

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

指针元素访问运算符不会检查超出界限的错误，并且在访问超出界限的元素时的行为不确定。这与 C 和 c + + 相同。

address-of 运算符

*Addressof_expression*由 "and" 符(`&`)开头，后跟一个*unary_expression*。

```
addressof_expression
    : '&' unary_expression
    ;
```

给定一个类型为的表达式，该表达式 `E` `T` 归类为固定变量(固定变量和可移动变量)，该构造 `&E` 计算给出的变量的地址 `E` 。结果的类型为 `T*` ，并归类为值。如果未分类为变量，则会发生编译时错误 `E` ，如果归类为只读局部变量，则会发生编译时错误; `E` 如果 `E` 表示可移动变量，则会发生编译时错误。在最后一种情况下，可以使用 fixed 语句(fixed 语句)来暂时 "修复" 该变量，然后获取其地址。如成员访问中所述，在定义字段的结构或类的实例构造函数或静态构造函数外部， `readonly` 该字段被视为值，而不是变量。因此无法采用其地址。同样，不能采用常量的地址。

`&` 运算符不需要将其自变量指定为明确赋值，但在 `&` 操作后，该运算符应用于的变量在执行操作的执行路径中被视为已明确赋值。编程人员负责确保在这种情况下正确地初始化变量。

示例中

```
using System;

class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` 在用于初始化的操作后被视为已明确赋值 `&i` `p` 。中的赋值将进行 `*p` 初始化 `i` ，但包含此初始化是程序员的责任，并且如果删除分配，则不会发生编译时错误。

运算符的明确赋值规则 `&` 可避免局部变量的冗余初始化。例如，许多外部 Api 采用一个指向结构的指针，该结构由 API 填充。调用此类 Api 通常会传递本地结构变量的地址，并且如果没有规则，则需要对结构变量进行冗余初始化。

指针增量和减量

在不安全的上下文中， `++` 和 `--` 运算符(后缀递增和递减运算符以及前缀增量和减量运算符)可应用于除以外的所有类型的指针变量 `void*` 。因此，对于每个指针类型 `T*` ，以下运算符都是隐式定义的：

```
T* operator ++(T* x);
T* operator --(T* x);
```

运算符分别与和生成相同的 `x + 1` 结果 `x - 1` (指针算法)。换言之，对于类型为的指针变量， `T*` `++` 运算符将添加 `sizeof(T)` 到变量中包含的地址， `--` 运算符将从该 `sizeof(T)` 变量中包含的地址中减去。

如果指针增量或减量运算溢出了指针类型的域，则结果是实现定义的，但不会产生异常。

指针算术

在不安全的上下文中， `+` 和 `-` 运算符(加法运算符和减法运算符)可应用于除之外的所有指针类型的值 `void*` 。因此，对于每个指针类型 `T*` ，以下运算符都是隐式定义的：

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

给定 `P` 指针类型的表达式 `T*` 和 `N` 类型为 `int` 、、或的表达式时 `uint` `long` `ulong` ，表达式 `P + N` 和 `N + P` 计算类型的指针值，该类型是 `T*` 通过将添加 `N * sizeof(T)` 到给定的地址来生成的 `P` 。同样，表达式 `P - N` 会计算类型的指针值 `T*` ，结果是从给定的地址中减去 `N * sizeof(T)` `P` 。

如果有两个表达式(`P` 和 `Q`)作为指针类型 `T*` ，则表达式将 `P - Q` 计算与给定的地址之间的差异， `P` 然后将 `Q` 该差异除以 `sizeof(T)` 。结果的类型始终为 `long` 。实际上， `P - Q` 将计算为 `((long)(P) - (long)(Q)) / sizeof(T)` 。

例如：

```
using System;

class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

生成输出的：

```
p - q = -14
q - p = 14
```

如果指针算术运算溢出了指针类型的域，则会以实现定义的方式截断结果，但不会产生异常。

指针比较

在不安全的上下文中，、、、 `==` `!=` `<` `>` 、 `<=` 和 `=>` 运算符(关系和类型测试运算符)可应用于所有指针类型的值。指针比较运算符是：

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

由于存在从任何指针类型到类型的隐式转换 `void*`，因此可以使用这些运算符比较任何指针类型的操作数。比较运算符比较两个操作数给定的地址，就像它们是无符号整数。

Sizeof 运算符

`sizeof` 运算符返回给定类型的变量所占用的字节数。指定为操作数的类型 `sizeof` 必须为*unmanaged_type*（指针类型）。

```
sizeof_expression
    : 'sizeof' '(' unmanaged_type ')'
    ;
```

运算符的结果 `sizeof` 为类型的值 `int` 。对于某些预定义类型， `sizeof` 运算符将生成一个常数值，如下表所示。

`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`
`sizeof(double)`	`8`
`sizeof(bool)`	`1`

对于所有其他类型，运算符的结果 `sizeof` 是实现定义的，并归类为值而不是常量。

成员打包到结构中的顺序是未指定的。

出于对齐目的，在结构的开头、结构和结构的末尾，可能有未命名的填充。用作填充的位数不确定。

当应用于具有结构类型的操作数时，结果为该类型的变量中的总字节数，包括任何空白。

fixed 语句

在不安全的上下文中，*embedded_statement*（语句）生产允许使用其他构造，即 `fixed` 语句，该语句用于 "修复" 可移动变量，使其地址在语句的持续时间内保持不变。

```
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')' embedded_statement
    ;

fixed_pointer_declarators
    : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;

fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;

fixed_pointer_initializer
    : '&' variable_reference
    | expression
    ;
```

每个*fixed_pointer_declarator*都声明给定*pointer_type*的局部变量，并使用相应*fixed_pointer_initializer*计算出的地址初始化该局部变量。语句中声明的局部变量 `fixed` 可在该变量的声明右侧以及语句的*embedded_statement*中发生的任何*fixed_pointer_initializer*中访问 `fixed` 。语句声明的局部变量 `fixed` 被视为只读。如果嵌入的语句尝试修改此局部变量（通过赋值或 `++` 和 `--` 运算符），或将其作为 `ref` 或参数传递，则会发生编译时错误 `out` 。

*Fixed_pointer_initializer*可以是以下项之一：

- 标记 " `&` " 后跟一个*variable_reference*（用于确定明确赋值的精确规则）到非托管类型的可移动变量（固定和可移动变量） `T` ，前提是该类型可 `T*` 隐式转换为语句中给定的指针类型 `fixed` 。在这种情况下，初始值设定项将计算给定变量的地址，并且在语句的持续时间内保证变量在固定的地址上保持不变 `fixed` 。
- 包含非托管类型的元素的*array_type*的表达式 `T` ，前提是该类型可 `T*` 隐式转换为语句中给定的指针类型 `fixed` 。在这种情况下，初始值设定项计算数组中第一个元素的地址，并且保证整个数组在语句的持续时间内保持为固定的地址 `fixed` 。如果数组表达式为 null 或数组包含零个元素，则初始值设定项将计算等于零的地址。
- `string` 如果类型可 `char*` 隐式转换为语句中给定的指针类型，则为类型的表达式 `fixed` 。在这种情况下，初始值设定项将计算字符串中第一个字符的地址，并且保证整个字符串在语句的持续时间内保持为固定地址 `fixed` 。 `fixed` 如果字符串表达式为 null，则语句的行为是实现定义的。
- 引用可移动变量的固定大小缓冲区成员的*simple_name*或*member_access*，前提是固定大小缓冲区成员的类型可隐式转换为语句中给定的指针类型 `fixed` 。在这种情况下，初始值设定项将计算一个指针，该指针指向固定大小缓冲区的第一个元素（表达式中的固定大小缓冲区），并且在语句的持续时间内保证固定大小缓冲区保持为固定的地址 `fixed` 。

对于由*fixed_pointer_initializer*计算的每个地址，该 `fixed` 语句可确保在语句的持续时间内，由垃圾回收器在该地址引用的变量不会重定位或释放 `fixed` 。例如，如果由*fixed_pointer_initializer*计算的地址引用一个对象的字段或数组实例的一个元素，则该 `fixed` 语句保证在语句的生存期内不会重新定位或释放包含对象实例。

编程人员应负责确保由语句创建的指针 `fixed` 不会在执行这些语句的范围之外。例如，当语句创建的指针 `fixed` 被传递给外部 api 时，程序员应负责确保 api 不保留这些指针的内存。

固定对象可能会导致堆碎片（因为它们无法移动）。因此，仅当绝对必要时才应修复对象，而只应在尽可能最短的时间进行修复。

示例

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

演示语句的几次使用 `fixed` 。第一条语句修复并获取静态字段的地址，第二条语句修复并获取实例字段的地址，第三条语句修复并获取数组元素的地址。在每种情况下，使用 regular 运算符都是错误的， `&` 因为变量全都归类为可移动变量。

上面示例中的第四个 `fixed` 语句生成的结果类似于第三个。

此语句的示例 `fixed` 使用 `string` ：

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

在不安全的上下文数组元素中，一维数组的元素以递增索引顺序存储，以 index 开头， `0` 以 index 结尾 `Length - 1` 。对于多维数组，将存储数组元素，这样最右边的维度的索引将首先增加，然后是下一个左侧维度，依此类推。在 `fixed` 获取指向数组实例的指针的语句中 `p` `a` ，从到的指针值 `p` 用于 `p + a.Length - 1` 表示数组中元素的地址。同样，范围内的变量 `p[0]` 用于 `p[a.Length - 1]` 表示实际数组元素。考虑到数组的存储方式，可以将任何维度的数组视为线性数组。

例如：

```
using System;

class Test
{
    static void Main() {
        int[,,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)    // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
                Console.WriteLine();
            }
    }
}
```

生成输出的:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

示例中

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

`fixed` 语句用于修复数组，以便可以将其地址传递给采用指针的方法。

在示例中:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}
```

fixed 语句用于修复结构的固定大小缓冲区，使其地址可用作指针。

`char*` 通过修复字符串实例生成的值始终指向以 null 结尾的字符串。在获取指向字符串实例的指针的 fixed 语句中 `p` `s`，从到的指针值（范围 `p` 为 `p + s.Length - 1`）表示字符串中的字符的地址，指针值 `p + s.Length` 始终指向空字符（带有值的字符 `'\0'`）。

通过固定指针修改托管类型的对象可能导致未定义的行为。例如，因为字符串是不可变的，所以，程序员应负责确保不会修改指向固定字符串的指针所引用的字符。

在调用需要 "C 样式" 字符串的外部 Api 时，字符串的自动 null 终止非常方便。但请注意，字符串实例允许包含 null 字符。如果存在这样的空字符，则在被视为以 null 结尾的情况下，该字符串将会被截断 `char*` 。

固定大小的缓冲区

固定大小的缓冲区用于将 "C 样式" 内联数组声明为结构的成员，主要用于与非托管 Api 建立交互。

固定大小的缓冲区声明

固定大小缓冲区是表示给定类型的变量的固定长度缓冲区存储的成员。固定大小的缓冲区声明引入给定元素类型的一个或多个固定大小的缓冲区。固定大小的缓冲区只允许在结构声明中使用，并且只能在不安全的上下文中出现（不安全上下文）。

```
struct_member_declaration_unsafe
    : fixed_size_buffer_declaration
    ;

fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type fixed_size_buffer_declarator+ ';'
    ;

fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;

buffer_element_type
    : type
    ;

fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

固定大小的缓冲区声明可以包含一组特性(特性)、`new` 修饰符(修饰符)、四个访问修饰符(类型参数和约束)和 `unsafe` 修饰符(不安全上下文)的有效组合。特性和修饰符适用于由固定大小缓冲区声明声明的所有成员。同一修饰符在固定大小的缓冲区声明中多次出现是错误的。

固定大小的缓冲区声明不允许包含 `static` 修饰符。

固定大小缓冲区声明的 buffer 元素类型指定声明引入的缓冲区的元素类型。Buffer 元素类型必须是预定义的类型之一:、、、、、、、、、 `sbyte` `byte` `short` `ushort` `int` `uint` `long` `ulong` `char` `float` `double` 或 `bool` 。

Buffer 元素类型后跟固定大小缓冲区声明符的列表，其中每个声明符都引入一个新成员。固定大小的缓冲区声明符包含命名成员的标识符，后跟括在和标记中的常量 `[` 表达式 `]` 。常数表达式表示该固定大小缓冲区声明符引入的成员中的元素数目。常数表达式的类型必须可隐式转换为类型 `int` ，并且值必须为非零正整数。

保证固定大小缓冲区的元素在内存中按顺序排列。

声明多个固定大小缓冲区的固定大小缓冲区声明等效于具有相同属性和元素类型的单个固定大小缓冲区声明的多个声明。例如:

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

等效于

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
   public fixed int z[100];
}
```

表达式中固定大小的缓冲区

固定大小缓冲区成员的成员查找(运算符)与字段的成员查找完全相同。

使用*simple_name*（类型推理）或*member_access*（对动态重载决策进行编译时检查），可以在表达式中引用固定大小的缓冲区。

当将固定大小的缓冲区成员作为简单名称引用时，其效果与窗体的成员访问相同 `this.I` ，其中 `I` 是固定大小的缓冲区成员。

在窗体的成员访问中 `E.I` ，如果 `E` 属于结构类型，并且 `I` 该结构类型中的成员查找标识固定大小成员，则 `E.I` 将按如下方式对其进行评估：

- 如果表达式 `E.I` 未出现在不安全的上下文中，则会发生编译时错误。
- 如果 `E` 归类为值，则会发生编译时错误。
- 否则，如果 `E` 是可移动变量（固定和可移动变量），并且表达式 `E.I` 不是*fixed_pointer_initializer*（Fixed 语句），则会发生编译时错误。
- 否则， `E` 引用一个固定变量，并且该表达式的结果是指向中的固定大小缓冲区成员的第一个元素的 `I` 指针 `E` 。结果的类型为 `S*` ，其中 `S` 是的元素类型 `I` ，并归类为值。

可以使用第一个元素的指针操作访问固定大小缓冲区的后续元素。与对数组的访问不同，对固定大小缓冲区的元素的访问是不安全的操作，并且不会进行范围检查。

下面的示例声明并使用具有固定大小缓冲区成员的结构。

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

明确赋值检查

固定大小的缓冲区不受明确的赋值检查（明确赋值）的限制，并且忽略固定大小的缓冲区成员，以便对结构类型变量进行明确的赋值检查。

当固定大小缓冲区成员的最外面的包含结构变量是静态变量、类实例的实例变量或数组元素时，固定大小缓冲区的元素会自动初始化为其默认值（默认值）。在所有其他情况下，固定大小缓冲区的初始内容是不确定的。

堆栈分配

在不安全的上下文中，局部变量声明（局部变量声明）可能包括从调用堆栈中分配内存的堆栈分配初始值设定项。

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

*Unmanaged_type*指示将存储在新分配的位置的项的类型，并且*表达式*指示这些项的数目。它们一起指定了所需的分配大小。由于堆栈分配的大小不能为负，因此将项的数目指定为计算结果为负值的*constant_expression*是编译时错误。

窗体的堆栈分配初始值设定项 `stackalloc T[E]` 需要为 `T` 非托管类型（指针类型），并且 `E` 必须是类型的表达式 `int` 。构造 `E * sizeof(T)` 从调用堆栈分配字节，并将类型的指针返回 `T*` 到新分配的块。如果 `E` 是负值，则行为是不确定的。如果 `E` 为零，则不进行任何分配，并且返回的指针是实现定义的。如果内存不足，无法分配指定大小的块， `System.StackOverflowException` 则会引发。

新分配的内存的内容未定义。

或块中不允许使用堆栈分配初始值设定项 `catch` `finally` （try 语句）。

无法显式释放使用分配 `stackalloc` 的内存。在函数成员执行过程中创建的所有堆栈分配的内存块将在该函数成员返回时自动被丢弃。这对应于 `alloca` 函数，这是一个在 C 和 c + + 实现中常见的扩展。

示例中

```
using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

`stackalloc`在方法中使用初始值设定项在 `IntToString` 堆栈上分配16个字符的缓冲区。当该方法返回时，缓冲区会自动丢弃。

动态内存分配

除了运算符以外 `stackalloc` ，c # 不提供用于管理非垃圾回收内存的预定义构造。此类服务通常通过支持类库来提供，也可以从基础操作系统直接导入。例如，下面的 `Memory` 类演示了如何从 c # 访问基础操作系统的堆函数：

```
using System;
```

```
using System;
using System.Runtime.InteropServices;

public static unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    private static readonly IntPtr s_heap = GetProcessHeap();

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(s_heap, HEAP_ZERO_MEMORY, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count)
    {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd)
        {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd)
        {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }

    // Frees a memory block.
    public static void Free(void* block)
    {
        if (!HeapFree(s_heap, 0, block)) throw new InvalidOperationException();
    }

    // Re-allocates a memory block. If the reallocation request is for a
    // larger size, the additional region of memory is automatically
    // initialized to zero.
    public static void* ReAlloc(void* block, int size)
    {
        void* result = HeapReAlloc(s_heap, HEAP_ZERO_MEMORY, block, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Returns the size of a memory block.
    public static int SizeOf(void* block)
    {
        int result = (int)HeapSize(s_heap, 0, block);
        if (result == -1) throw new InvalidOperationException();
        return result;
    }

    // Heap API flags
    private const int HEAP_ZERO_MEMORY = 0x00000008;

    // Heap API functions
    [DllImport("kernel32")]
    private static extern IntPtr GetProcessHeap();

    [DllImport("kernel32")]
    private static extern void* HeapAlloc(IntPtr hHeap, int flags, UIntPtr size);

    [DllImport("kernel32")]
    private static extern bool HeapFree(IntPtr hHeap, int flags, void* block);
```

```
    private static extern bool HeapFree(IntPtr hHeap, int flags, void* block);

    [DllImport("kernel32")]
    private static extern void* HeapReAlloc(IntPtr hHeap, int flags, void* block, UIntPtr size);

    [DllImport("kernel32")]
    private static extern UIntPtr HeapSize(IntPtr hHeap, int flags, void* block);
}
```

`Memory` 下面给出了使用类的示例：

```
class Test
{
    static unsafe void Main()
    {
        byte* buffer = null;
        try
        {
            const int Size = 256;
            buffer = (byte*)Memory.Alloc(Size);
            for (int i = 0; i < Size; i++) buffer[i] = (byte)i;
            byte[] array = new byte[Size];
            fixed (byte* p = array) Memory.Copy(buffer, p, Size);
            for (int i = 0; i < Size; i++) Console.WriteLine(array[i]);
        }
        finally
        {
            if (buffer != null) Memory.Free(buffer);
        }
    }
}
```

该示例通过分配256字节的内存 `Memory.Alloc` ，并初始化值从0增加到255的内存块。然后，它分配一个256元素字节数组，并使用将 `Memory.Copy` 内存块的内容复制到字节数组中。最后，使用释放内存块， `Memory.Free` 并在控制台上输出字节数组的内容。

文档注释

2020/11/2 • • Edit Online

C#为程序员提供一种机制，以使用包含 XML 文本的特殊注释语法记录其代码。在源代码文件中，具有特定窗体的注释可用于指示工具从这些注释生成 XML，并将其置于后面。使用此语法的注释称为***文档注释***。它们必须紧跟在用户定义的类型（如类、委托或接口）或成员（如字段、事件、属性或方法）之前。XML 生成工具称为***文档生成器***。（此生成器可能是，但不一定是C#编译器本身。）文档生成器生成的输出称为 "***文档文件***"。文档文件用作***文档查看器***的输入;一种用于生成类型信息及其关联文档的某种视觉显示方式的工具。

此规范建议在文档注释中使用一组标记，但不需要使用这些标记，如果需要，还可以使用其他标记，前提是符合格式正确的 XML 的规则。

介绍

具有特殊形式的注释可用于指示工具从这些注释生成 XML，并将其置于之前。此类注释是以三个斜杠（`///`）开头的单行注释，或以斜杠和双星（`/**`）开头的分隔注释。它们必须紧跟在用户定义的类型（如类、委托或接口）或它们所批注的成员（如字段、事件、属性或方法）之前。特性部分（特性规范）被视为声明的一部分，因此文档注释必须位于应用到类型或成员的特性之前。

语法

```
single_line_doc_comment
    : '///' input_character*
    ;

delimited_doc_comment
    : '/**' delimited_comment_section* asterisk+ '/'
    ;
```

在*single_line_doc_comment*中，如果在当前*single_line_doc_comment*附近的每个*single_line_doc_comment*上的`///`字符后面有一个*空格*字符，则该*空白*字符将不包含在 XML 输出中。

在分隔的文档注释中，如果第二行中的第一个非空白字符为星号并且具有相同的可选空白字符模式，并且在分隔的文档注释中每一行的开头重复星号字符，然后，XML 输出中不包括重复模式的字符。此模式可能包含后面和星号字符后面的空白字符。

示例：

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
    /// <summary>method <c>draw</c> renders the point.</summary>
    void draw() {...}
}
```

文档注释中的文本必须根据 XML 规则（ https://www.w3.org/TR/REC-xml）。如果 XML 格式不正确，则会生成警告，并且文档文件将包含一条注释，指出遇到了错误。

尽管开发人员可以自由地创建自己的一组标记，但建议的标记中定义了一个建议的集。部分建议标记具有特殊含义：

- `<param>` 标记用于描述参数。如果使用此类标记，文档生成器必须验证指定的参数是否存在以及文档注释中是否描述了所有参数。如果此类验证失败，文档生成器会发出警告。
- `cref` 属性可以附加到任何标记，以提供对代码元素的引用。文档生成器必须验证此代码元素是否存在。如果验证失败，文档生成器会发出警告。查找 `cref` 特性中所述的名称时，文档生成器必须根据源代码中显示的 `using` 语句来区分命名空间可见性。对于泛型代码元素，不能使用常规泛型语法（即 "`List<T>`"），因为它产生了无效的 XML。可以使用大括号代替方括号（即 "`List{T}`"），也可以使用 XML 转义语法（即 "`List<T>`"）。
- `<summary>` 标记旨在供文档查看器用来显示有关某个类型或成员的其他信息。
- `<include>` 标记包含外部 XML 文件中的信息。

请注意，文档文件不提供有关类型和成员（例如，它不包含任何类型信息）的完整信息。若要获取有关某个类型或成员的此类信息，文档文件必须与实际类型或成员的反射一起使用。

建议的标记

文档生成器必须接受并处理根据 XML 规则有效的任何标记。以下标记提供用户文档中的常用功能。（当然，其他标记是可能的。）

`<c>`	`<c>`	设置类似代码的字体中的文本
`<code>`	`<code>`	设置一个或多个源代码或程序输出行
`<example>`	`<example>`	指示示例
`<exception>`	`<exception>`	标识方法可以引发的异常
`<include>`	`<include>`	包含来自外部文件的 XML
`<list>`	`<list>`	创建列表或表
`<para>`	`<para>`	允许将结构添加到文本中
`<param>`	`<param>`	描述方法或构造函数的参数
`<paramref>`	`<paramref>`	确定某一词为参数名称
`<permission>`	`<permission>`	记录成员的安全可访问性
`<remarks>`	`<remarks>`	描述有关类型的其他信息
`<returns>`	`<returns>`	描述方法的返回值
`<see>`	`<see>`	指定链接
`<seealso>`	`<seealso>`	生成 "另请参阅" 条目
`<summary>`	`<summary>`	描述类型或类型的成员
`<value>`	`<value>`	描述属性

[	SECTION	[[
`<typeparam>`		描述泛型类型参数
`<typeparamref>`		确定某个单词为类型参数名称

`<c>`

此标记提供一种机制，用于指示说明中的文本片段应设置为特殊字体，如用于代码块的。对于实际代码行，请使用 `<code>` （`<code>`）。

语法

```
<c>text</c>
```

示例：

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>

public class Point
{
    // ...
}
```

`<code>`

此标记用于设置一个或多个源代码或程序输出行，采用某种特殊字体。对于简单的代码片段，请使用 `<c>` （`<c>`）。

语法

```
<code>source code or program output</code>
```

示例：

```
/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// <example>For example:
/// <code>
///    Point p = new Point(3,5);
///    p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>

public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

`<example>`

此标记允许在注释内使用示例代码来指定如何使用方法或其他库成员。通常情况下，这也会涉及使用标记 `<code>` （`<code>`）。

语法

```
<example>description</example>
```

示例：

有关示例，请参阅 `<code>`（`<code>`）。

`<exception>`

此标记提供了一种方法，用于记录方法可能引发的异常。

语法

```
<exception cref="member">description</exception>
```

其中

- `member` 是成员的名称。文档生成器检查给定成员是否存在，并将 `member` 转换为文档文件中的规范元素名称。
- `description` 是引发异常的环境的说明。

示例：

```
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}
```

`<include>`

此标记允许包含源代码文件外部的 XML 文档中的信息。外部文件必须是格式正确的 XML 文档，并将 XPath 表达式应用于该文档以指定要包含的文档的 XML。然后，将 `<include>` 标记替换为外部文档中所选的 XML。

语法

```
<include file="filename" path="xpath" />
```

其中

- `filename` 是外部 XML 文件的文件名。文件名是相对于包含标记的文件进行解释的。
- `xpath` 是用于选择外部 XML 文件中的一些 XML 的 XPath 表达式。

示例：

如果源代码包含如下所示的声明：

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

外部文件 "文档" 具有以下内容：

```
<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
  <class name="StringList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
</extradoc>
```

然后输出相同的文档，就像源代码中所含的一样：

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { ... }
```

`<list>`

此标记用于创建列表或项的表。它可能包含一个 `<listheader>` 块来定义表或定义列表的标题行。（定义表时，只需提供标题中 `term` 的条目。）

列表中的每一项都指定了一个 `<item>` 块。创建定义列表时，必须同时指定 `term` 和 `description`。但是，对于表、项目符号列表或编号列表，只需指定 `description`。

语法

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>*description*</description>
   </listheader>
   <item>
      <term>term</term>
      <description>*description*</description>
   </item>
    ...
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

其中

- `term` 是要定义的术语，其定义在 `description` 中。
- `description` 是项目符号列表或编号列表中的项，或是 `term` 的定义。

示例：

```
public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}
```

`<para>`

此标记在其他标记内使用，如 `<summary>`（`<remarks>`）或 `<returns>`（`<returns>`），并允许将结构添加到文本中。

语法

```
<para>content</para>
```

其中 `content` 是段落的文本。

示例：

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

`<param>`

此标记用于描述方法、构造函数或索引器的参数。

语法

```
<param name="name">description</param>
```

其中

- `name` 是参数的名称。
- `description` 是参数的描述。

示例：

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<paramref>`

此标记用于指示字词是参数。可以采用某种不同的方式处理文档文件以设置此参数的格式。

语法

```
<paramref name="name"/>
```

其中 `name` 是参数的名称。

示例:

```
/// <summary>This constructor initializes the new Point to
///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<permission>`

此标记允许记录成员的安全可访问性。

语法

```
<permission cref="member">description</permission>
```

其中

- `member` 是成员的名称。文档生成器检查给定的代码元素是否存在，并将*成员*转换为文档文件中的规范元素名称。
- `description` 是对成员的访问权限的说明。

示例:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>

public static void Test() {
    // ...
}
```

`<remarks>`

此标记用于指定有关类型的额外信息。(使用 `<summary>` (`<summary>`)描述类型本身和类型成员。)

语法

```
<remarks>description</remarks>
```

其中 `description` 是注释的文本。

示例:

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remarks>Uses polar coordinates</remarks>
public class Point
{
    // ...
}
```

`<returns>`

此标记用于描述方法的返回值。

语法

```
<returns>description</returns>
```

其中 `description` 是返回值的说明。

示例:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///    without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + "," + Y + ")";
}
```

`<see>`

此标记允许在文本中指定链接。使用 `<seealso>` (`<seealso>`)指示要在 "另请参见" 部分中显示的文本。

语法

```
<see cref="member"/>
```

其中 `member` 是成员的名称。文档生成器检查给定的代码元素是否存在，并在生成的文档文件中将*成员*更改为元素名称。

示例:

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

`<seealso>`

此标记允许为 "另请参阅" 部分生成一个条目。使用 `<see>` (`<see>`)来指定文本中的链接。

语法

```
<seealso cref="member"/>
```

其中 `member` 是成员的名称。文档生成器检查给定的代码元素是否存在，并在生成的文档文件中将*成员*更改为元素名称。

示例:

```
/// <summary>This method determines whether two Points have the same
///    location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

`<summary>`

此标记可用于描述类型或类型的成员。使用 `<remarks>` (`<remarks>`)来描述类型本身。

语法

```
<summary>description</summary>
```

其中 `description` 是类型或成员的汇总。

示例:

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

`<value>`

此标记允许描述属性。

语法

```
<value>property description</value>
```

其中 `property description` 是属性的说明。

示例：

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```

`<typeparam>`

此标记用于描述类、结构、接口、委托或方法的泛型类型参数。

语法

```
<typeparam name="name">description</typeparam>
```

其中 `name` 是类型参数的名称，`description` 为其说明。

示例：

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

`<typeparamref>`

此标记用于指示字词为类型参数。可以对文档文件进行处理，以便以某种不同的方式设置此类型参数的格式。

语法

```
<typeparamref name="name"/>
```

其中 `name` 是类型参数的名称。

示例：

```
/// <summary>This method fetches data and returns a list of <typeparamref name="T"/>.</summary>
/// <param name="query">query to execute</param>
public List<T> FetchData<T>(string query) {
    ...
}
```

处理文档文件

文档生成器为源代码中使用文档注释标记的每个元素生成一个 ID 字符串。此 ID 字符串唯一标识源元素。文档查看器可以使用 ID 字符串来标识文档应用到的相应元数据/反射项。

文档文件不是源代码的层次结构表示形式;相反，它是一个平面列表，其中包含每个元素的生成的 ID 字符串。

ID 字符串格式

文档生成器在生成 ID 字符串时遵循以下规则：

- 字符串不得包含空格。
- 字符串的第一部分通过一个字符后跟一个冒号来标识所记录的成员的种类。定义以下类型的成员：

“	“
E	事件
F	字段
M	方法（包括构造函数、析构函数和运算符）
N	命名空间
P	属性（包括索引器）
T	类型（如类、委托、枚举、接口和结构）
!	错误字符串;其余字符串提供有关错误的信息。例如，文档生成器为无法解析的链接生成错误信息。

- 字符串的第二部分是元素的完全限定名，从命名空间的根开始。元素的名称、其封闭类型和命名空间用句点分隔。如果项本身的名称包含句点，则这些名称将替换为 `#(U+0023)` 字符。（假定没有元素在其名称中包含此字符。）
- 对于包含参数的方法和属性，将在参数列表的后面加上括号。对于没有参数的那些参数，将省略括号。确保自变量之间用逗号分隔。每个自变量的编码都与 CLI 签名相同，如下所示：
  - 参数由其文档名称表示，其文档名称基于其完全限定名称，如下所示进行修改：
    - 表示泛型类型的参数将追加 `` ` ``（反撇号）字符后跟类型参数的数目
    - 具有 `out` 或 `ref` 修饰符的参数在其类型名称后面具有 `@`。通过值或通过 `params` 传递的参数没有特殊表示法。
    - 作为数组的参数表示为 `[lowerbound:size, ... , lowerbound:size]`，其中逗号的数目小于1，并且每个维度的下限和大小（如果已知）以十进制表示。如果未指定下限或大小，则省略它。如果省略特定维度的下限和大小，则也会省略 `:`。交错数组由每个级别一个 `[]` 表示。
    - 具有 void 以外的指针类型的参数使用在类型名称后面的 `*` 表示。Void 指针使用 `System.Void` 的类型名称表示。
    - 引用类型上定义的泛型类型参数的参数使用 `` ` ``（反撇号）字符进行编码，后跟类型参数的从零开始的索引。
    - 使用在方法中定义的泛型类型参数的参数使用双反撇号 `` `` `` 而不是用于类型的 `` ` ``。
    - 引用构造的泛型类型的参数使用泛型类型进行编码，后跟 `{`，后跟以逗号分隔的类型参数列表，后跟 `}`。

ID 字符串示例

下面的示例分别显示C#代码片段，以及从每个支持文档注释的源元素生成的 ID 字符串：

- 类型使用其完全限定名称进行表示，并与一般信息一起使用：

```
enum Color { Red, Blue, Green }

namespace Acme
{
    interface IProcess {...}

    struct ValueType {...}

    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }

    class MyList<T>
    {
        class Helper<U,V> {...}
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
```

- 字段由其完全限定名称表示：

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }

        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}

"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- 构造函数。

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}

"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- 函数.

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}

"M:Acme.Widget.Finalize"
```

- 方法.

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }

        public static void M0() {...}
        public void M1(char c, out float f, ref ValueType v) {...}
        public void M2(short[] x1, int[,] x2, long[][] x3) {...}
        public void M3(long[][] x3, Widget[][,,] x4) {...}
        public unsafe void M4(char *pc, Color **pf) {...}
        public unsafe void M5(void *pv, double *[][,] pd) {...}
        public void M6(int i, params object[] args) {...}
    }

    class MyList<T>
    {
        public void Test(T t) { }
    }

    class UseList
    {
        public void Process(MyList<int> list) { }
        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- 属性和索引器。

```
namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- 事件。

```
namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"
```

- 一元运算符。

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

使用的一元运算符函数名称的完整集合如下所示：`op_UnaryPlus`、`op_UnaryNegation`、`op_LogicalNot`、`op_OnesComplement`、`op_Increment`、`op_Decrement`、`op_True` 和 `op_False`。

- 二元运算符。

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

使用的二元运算符函数名称的完整集如下：`op_Addition`、`op_Subtraction`、`op_Multiply`、`op_Division`、`op_Modulus`、`op_BitwiseAnd`、`op_BitwiseOr`、`op_ExclusiveOr`、`op_LeftShift`、`op_RightShift`、`op_Equality`、`op_Inequality`、`op_LessThan`、`op_LessThanOrEqual`、`op_GreaterThan` 和 `op_GreaterThanOrEqual`。

- 转换运算符的尾随 "`~`" 后跟返回类型。

```
namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

示例

C#源代码

下面的示例演示 `Point` 类的源代码：

```
namespace Graphics
{

/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
/// </summary>
public class Point
{

    /// <summary>Instance variable <c>x</c> represents the point's
    ///    x-coordinate.</summary>
    private int x;

    /// <summary>Instance variable <c>y</c> represents the point's
    ///    y-coordinate.</summary>
    private int y;

    /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
    public int X
    {
        get { return x; }
        set { x = value; }
    }

    /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    /// <summary>This constructor initializes the new Point to
    ///    (0,0).</summary>
    public Point() : this(0,0) {}

    /// <summary>This constructor initializes the new Point to
    ///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
    /// <param><c>xor</c> is the new Point's x-coordinate.</param>
    /// <param><c>yor</c> is the new Point's y-coordinate.</param>
    public Point(int xor, int yor) {
        X = xor;
        Y = yor;
    }

    /// <summary>This method changes the point's location to
    ///    the given coordinates.</summary>
    /// <param><c>xor</c> is the new x-coordinate.</param>
    /// <param><c>yor</c> is the new y-coordinate.</param>
```

```
    /// <see cref="Translate"/>
    public void Move(int xor, int yor) {
        X = xor;
        Y = yor;
    }

    /// <summary>This method changes the point's location by
    ///    the given x- and y-offsets.
    /// <example>For example:
    /// <code>
    ///    Point p = new Point(3,5);
    ///    p.Translate(-1,3);
    /// </code>
    /// results in <c>p</c>'s having the value (2,8).
    /// </example>
    /// </summary>
    /// <param><c>xor</c> is the relative x-offset.</param>
    /// <param><c>yor</c> is the relative y-offset.</param>
    /// <see cref="Move"/>
    public void Translate(int xor, int yor) {
        X += xor;
        Y += yor;
    }

    /// <summary>This method determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>o</c> is the object to be compared to the current object.
    /// </param>
    /// <returns>True if the Points have the same location and they have
    ///    the exact same type; otherwise, false.</returns>
    /// <seealso cref="operator=="/>
    /// <seealso cref="operator!="/>
    public override bool Equals(object o) {
        if (o == null) {
            return false;
        }

        if (this == o) {
            return true;
        }

        if (GetType() == o.GetType()) {
            Point p = (Point)o;
            return (X == p.X) && (Y == p.Y);
        }
        return false;
    }

    /// <summary>Report a point's location as a string.</summary>
    /// <returns>A string representing a point's location, in the form (x,y),
    ///    without any leading, training, or embedded whitespace.</returns>
    public override string ToString() {
        return "(" + X + "," + Y + ")";
    }

    /// <summary>This operator determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points have the same location and they have
    ///    the exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator!="/>
    public static bool operator==(Point p1, Point p2) {
        if ((object)p1 == null || (object)p2 == null) {
            return false;
        }

        if (p1.GetType() == p2.GetType()) {
```

```
            return (p1.X == p2.X) && (p1.Y == p2.Y);
        }

        return false;
    }

    /// <summary>This operator determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points do not have the same location and the
    ///    exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator=="/>
    public static bool operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }

    /// <summary>This is the entry point of the Point class testing
    /// program.
    /// <para>This program tests each method and operator, and
    /// is intended to be run after any non-trivial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
        // class test code goes here
    }
}
}
```

生成的 XML

下面是在给定类 `Point` 的源代码时，由一个文档生成器生成的输出，如下所示：

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-dimensional
            plane.
            </summary>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>

        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>

        <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
```

```
    <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
        <summary>This method changes the point's location to
        the given coordinates.</summary>
        <param><c>xor</c> is the new x-coordinate.</param>
        <param><c>yor</c> is the new y-coordinate.</param>
        <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
    </member>

    <member
        name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
        <summary>This method changes the point's location by
        the given x- and y-offsets.
        <example>For example:
        <code>
        Point p = new Point(3,5);
        p.Translate(-1,3);
        </code>
        results in <c>p</c>'s having the value (2,8).
        </example>
        </summary>
        <param><c>xor</c> is the relative x-offset.</param>
        <param><c>yor</c> is the relative y-offset.</param>
        <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
    </member>

    <member name="M:Graphics.Point.Equals(System.Object)">
        <summary>This method determines whether two Points have the same
        location.</summary>
        <param><c>o</c> is the object to be compared to the current
        object.
        </param>
        <returns>True if the Points have the same location and they have
        the exact same type; otherwise, false.</returns>
        <seealso
  cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
        <seealso
  cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>

    <member name="M:Graphics.Point.ToString">
        <summary>Report a point's location as a string.</summary>
        <returns>A string representing a point's location, in the form
        (x,y),
        without any leading, training, or embedded whitespace.</returns>
    </member>

    <member
   name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
        <summary>This operator determines whether two Points have the
        same
        location.</summary>
        <param><c>p1</c> is the first Point to be compared.</param>
        <param><c>p2</c> is the second Point to be compared.</param>
        <returns>True if the Points have the same location and they have
        the exact same type; otherwise, false.</returns>
        <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
        <seealso
 cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>

    <member
  name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
        <summary>This operator determines whether two Points have the
        same
        location.</summary>
        <param><c>p1</c> is the first Point to be compared.</param>
        <param><c>p2</c> is the second Point to be compared.</param>
        <returns>True if the Points do not have the same location and
        the
```

```
            exact same type; otherwise, false.</returns>
            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
            <seealso
      cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
        </member>

        <member name="M:Graphics.Point.Main">
            <summary>This is the entry point of the Point class testing
            program.
            <para>This program tests each method and operator, and
            is intended to be run after any non-trivial maintenance has
            been performed on the Point class.</para></summary>
        </member>

        <member name="P:Graphics.Point.X">
            <value>Property <c>X</c> represents the point's
            x-coordinate.</value>
        </member>

        <member name="P:Graphics.Point.Y">
            <value>Property <c>Y</c> represents the point's
            y-coordinate.</value>
        </member>
    </members>
</doc>
```