

# C#语言参考

## 注意

本文并非最终文档，在最终的商业版本发布前，可能会有重大修改。而且，它属微软公司所有。它是根据接受者和微软公司间的非公开协议公布的。

本文档只是为了报告的目的提供的，并且，在本文档中，微软公司并没有作任何明示或暗示的担保。对本文档中的内容进行更改不会给出提示。

用户要对由于使用本文档所造成的全部危险和后果负责。用户有责任遵守所有有效的版权法律。

虽然没有用版权来限制，但是如果没有 Microsoft 公司明确的书面许可，本文档的任何部分都不可以被复制、存储或引入检索系统，或用任何手段进行传播（电子的、机械的，影印，记录或其他方式）或其它任何用途。

微软拥有涉及本文档主要内容的专利、专有应用程序、商标、版权和其它知识产权。除非有微软公司提供的任何书面的许可，提供本文档并没有给你任何关于这些专利、商标、版权和其它知识产权的许可。

未出版作品。© 1999-2002 Microsoft 公司。版权所有。

Microsoft, Windows, Visual Basic, and Visual C++ 是微软公司在美国和/或其他国家或地区的注册商标或商标。

这里所提及的其它产品和公司的名称可能是他们各自所有者的商标。

# 目录

<b>1. 介绍</b>	<b>1</b>
1.1 Hello, world	1
1.2 类型	2
1.2.1 预定义类型	3
1.2.2 转换	5
1.2.3 数组类型	6
1.2.4 类型系统一致	7
1.3 变量和参数	8
1.4 自动内存管理	11
1.5 表达式	13
1.6 声明	15
1.7 类	18
1.7.1 常数	20
1.7.2 域	20
1.7.3 方法	21
1.7.4 属性	22
1.7.5 事件	23
1.7.6 操作符	24
1.7.7 索引	25
1.7.8 实例构造函数	26
1.7.9 析构函数	27
1.7.10 静态构造函数	27
1.7.11 继承	28
1.8 结构	29
1.9 接口	30
1.10 代表	31
1.11 联合	32
1.12 名称空间和集合	32
1.13 版本	34
1.14 特征	36
<b>2. 语法结构</b>	<b>39</b>
2.1 翻译的阶段	39
2.2 文法符号	39
2.3 预处理	40
2.3.1 预处理声明	40
2.3.2 #if, #elif, #else, #endif	41
2.3.3 预处理控制行	42
2.3.4 #line	43
2.3.5 预处理标识符	43
2.3.6 预处理表达式	43
2.3.7 与空白交互作用	44
2.4 语法分析	45

2.4.1 输入 .....	45
2.4.2 输入字符 .....	45
2.4.3 行结束符 .....	45
2.4.4 注释 .....	45
2.4.5 空白 .....	46
2.4.6 标记 .....	47
2.5 句法分析 .....	47
2.5.1 标识符 .....	47
2.5.2 关键字 .....	48
2.5.3 数据符号 .....	48
2.5.4 操作符和标点 .....	53
2.5.5 Unicode 字符转意字符序列 .....	53
<b>3. 基本概念 .....</b>	<b>55</b>
3.1 声明 .....	55
3.2 成员 .....	57
3.2.1 名称空间成员 .....	57
3.2.2 结构成员 .....	57
3.2.3 枚举成员 .....	57
3.2.4 类成员 .....	58
3.2.5 接口成员 .....	58
3.2.6 数组成员 .....	58
3.2.7 代表成员 .....	58
3.3 成员访问 .....	58
3.3.1 声明可访问性 .....	58
3.3.2 可访问性域 .....	59
3.3.3 保护的访问 .....	61
3.3.4 可访问性约束 .....	62
3.4 签名和重载 .....	62
3.5 范围 .....	63
3.5.1 名称隐藏 .....	65
3.6 名称空间或类型名称 .....	67
3.6.1 合格的名称 .....	68
<b>4. 类型 .....</b>	<b>69</b>
4.1 数值类型 .....	69
4.1.1 默认构造函数 .....	70
4.1.2 结构类型 .....	71
4.1.3 简单类型 .....	71
4.1.4 整数类型 .....	72
4.1.5 浮点类型 .....	73
4.1.6 十进制类型 .....	74
4.1.7 布尔类型 .....	74
4.1.8 枚举类型 .....	74
4.2 引用类型 .....	74
4.2.1 类类型 .....	75

4.2.2 对象类型 .....	75
4.2.3 字符串类型 .....	75
4.2.4 接口类型 .....	76
4.2.5 数组类型 .....	76
4.2.6 代表类型 .....	76
4.3 包装和解包 .....	76
4.3.1 包装转换 .....	76
4.3.2 解包转换 .....	77
<b>5. 变量.....</b>	<b>79</b>
5.1 变量类型 .....	79
5.1.1 静态变量 .....	79
5.1.2 实例变量 .....	79
5.1.3 数组元素 .....	80
5.1.4 数值参数 .....	80
5.1.5 引用参数 .....	80
5.1.6 输出参数 .....	80
5.1.7 局部变量 .....	81
5.2 默认值 .....	81
5.3 明确赋值 .....	81
5.3.1 初始赋值变量 .....	83
5.3.2 非初始赋值变量 .....	84
5.4 变量引用 .....	84
<b>6. 转换.....</b>	<b>85</b>
6.1 隐式转换 .....	85
6.1.1 同一性转换 .....	85
6.1.2 隐式数值转换 .....	85
6.1.3 隐式枚举转换 .....	86
6.1.4 隐式引用转换 .....	86
6.1.5 转换 .....	86
6.1.6 隐式常数表达式转换 .....	86
6.1.7 用户自定义隐式转换 .....	87
6.2 显式转换 .....	87
6.2.1 显式数值转换 .....	87
6.2.2 显式枚举类型转换 .....	88
6.2.3 显式引用类型转换 .....	88
6.2.4 解包转换 .....	89
6.2.5 用户自定义显式转换 .....	89
6.3 标准转换 .....	89
6.3.1 标准隐式转换 .....	89
6.3.2 标准显式转换 .....	90
6.4 用户定义转换 .....	90
6.4.1 允许的用户自定义转换 .....	90
6.4.2 用户自定义转换的取值 .....	90
6.4.3 用户自定义隐式转换 .....	91

6.4.4 用户自定义显式类型转换.....	91
<b>7. 表达式.....</b>	<b>93</b>
7.1 表达式分类.....	93
7.1.1 表达式的数值.....	94
7.2 操作符.....	94
7.2.1 操作符优先级和结合顺序.....	94
7.2.2 操作符重载.....	96
7.2.3 一元操作符重载分析.....	97
7.2.4 二元操作符重载分析.....	97
7.2.5 候选用户定义操作符.....	98
7.2.6 数字升级.....	98
7.3 成员查询.....	99
7.3.1 基类型.....	100
7.4 函数成员.....	100
7.4.1 参数列表.....	102
7.4.2 重载分析.....	103
7.4.3 功能成员调用.....	105
7.4.4 虚拟函数成员查找.....	106
7.4.5 接口函数成员查找.....	106
7.5 主要的表达式.....	106
7.5.1 文字.....	106
7.5.2 简单名称.....	107
7.5.3 加括号的表达式.....	108
7.5.4 成员访问.....	108
7.5.5 调用表达式.....	110
7.5.6 元素访问.....	112
7.5.7 This 访问.....	113
7.5.8 基本访问.....	114
7.5.9 递增和递减后缀操作符.....	114
7.5.10 new 操作符.....	115
7.5.11 typeof 操作符.....	119
7.5.12 sizeof 操作符.....	119
7.5.13 checked 和 unchecked 操作符.....	120
7.6 一元表达式.....	122
7.6.1 一元正值运算符.....	122
7.6.2 一元负值运算符.....	122
7.6.3 逻辑非运算符.....	123
7.6.4 按位求补运算符.....	123
7.6.5 间接运算符.....	123
7.6.6 地址运算符.....	123
7.6.7 前缀增量和减量运算符.....	123
7.6.8 强制类型转换表达式.....	124
7.7 算术运算符.....	125
7.7.1 乘法运算符.....	125

7.7.2 除法运算符 .....	126
7.7.3 余数运算符 .....	127
7.7.4 加法运算符 .....	127
7.7.5 减法运算符 .....	129
7.8 移位运算符 .....	130
7.9 关系运算符 .....	131
7.9.1 整数比较运算符 .....	132
7.9.2 浮点比较运算符 .....	132
7.9.3 小数比较运算符 .....	133
7.9.4 布尔相等运算符 .....	133
7.9.5 枚举比较运算符 .....	133
7.9.6 引用类型相等运算符 .....	133
7.9.7 字符串相等运算符 .....	135
7.9.8 代表相等运算符 .....	135
7.9.9 is 运算符 .....	135
7.10 逻辑运算符 .....	136
7.10.1 整数逻辑运算符 .....	136
7.10.2 枚举逻辑运算符 .....	136
7.10.3 布尔逻辑运算符 .....	137
7.11 条件逻辑运算符 .....	137
7.11.1 布尔条件逻辑运算符 .....	137
7.11.2 用户自定义的条件逻辑运算符 .....	138
7.12 条件运算符 .....	138
7.13 赋值运算符 .....	139
7.13.1 简单赋值 .....	139
7.13.2 组合赋值 .....	141
7.13.3 事件赋值 .....	142
7.14 表达式 .....	142
7.15 常量表达式 .....	142
7.16 布尔表达式 .....	143
<b>8. 语句 .....</b>	<b>145</b>
8.1 终点与可达性 .....	145
8.2 块 .....	146
8.2.1 语句列表 .....	147
8.3 空语句 .....	147
8.4 标号语句 .....	148
8.5 声明语句 .....	148
8.5.1 局部变量声明 .....	148
8.5.2 局部常量声明 .....	149
8.6 表达式语句 .....	150
8.7 选择语句 .....	150
8.7.1 if 语句 .....	150
8.7.2 Switch 语句 .....	151
8.8 重复语句 .....	154

8.8.1 while 语句 .....	155
8.8.2 do 语句 .....	155
8.8.3 for 语句 .....	155
8.8.4 foreach 语句 .....	157
8.9 跳转语句 .....	158
8.9.1 break 语句 .....	158
8.9.2 continue 语句 .....	159
8.9.3 goto 语句 .....	159
8.9.4 return 语句 .....	160
8.9.5 throw 语句 .....	160
8.10 try 语句 .....	161
8.11 checked 和 unchecked 语句 .....	163
8.12 lock 语句 .....	163
<b>9. 名称空间 .....</b>	<b>165</b>
9.1 编译单元 .....	165
9.2 名称空间声明 .....	165
9.3 使用指示 .....	166
9.3.1 使用别名指示 .....	167
9.3.2 使用名称空间指示 .....	169
9.4 名称空间成员 .....	170
9.5 类型声明 .....	171
<b>10. 类 .....</b>	<b>173</b>
10.1 类声明 .....	173
10.1.1 类修饰符 .....	173
10.1.2 类基础规范 .....	174
10.1.3 类主体 .....	176
10.2 类成员 .....	176
10.2.1 继承 .....	177
10.2.2 new 修饰符 .....	177
10.2.3 访问修饰符 .....	178
10.2.4 要素类型 .....	178
10.2.5 静态和实例成员 .....	178
10.2.6 嵌套类型 .....	179
10.3 常数 .....	179
10.4 域 .....	180
10.4.1 静态和实例域 .....	181
10.4.2 只读域 .....	182
10.4.3 域的初始化 .....	183
10.4.4 变量初始化函数 .....	183
10.5 方法 .....	185
10.5.1 方法参数 .....	186
10.5.2 静态和实例方法 .....	189
10.5.3 虚拟方法 .....	190
10.5.4 覆盖方法 .....	191



10.5.5 抽象方法 .....	193
10.5.6 外部方法 .....	194
10.5.7 方法主体 .....	195
10.5.8 方法重载 .....	195
10.6 属性 .....	195
10.6.1 静态属性 .....	196
10.6.2 虚拟属性 .....	196
10.6.3 覆盖属性 .....	197
10.6.4 抽象属性 .....	198
10.6.5 访问符 .....	198
10.7 事件 .....	203
10.8 索引 .....	206
10.8.1 索引重载 .....	209
10.8.2 虚拟索引 .....	209
10.8.3 覆盖索引 .....	209
10.8.4 抽象索引 .....	210
10.9 操作符 .....	210
10.9.1 一元操作符 .....	211
10.9.2 二元操作符 .....	211
10.9.3 转换操作符 .....	212
10.10 实例构造函数 .....	213
10.10.1 构造函数初始化函数 .....	214
10.10.2 实例变量初始化函数 .....	214
10.10.3 构造函数执行 .....	214
10.10.4 默认构造函数 .....	216
10.10.5 私有构造函数 .....	217
10.10.6 可选的构造函数参数 .....	217
10.11 析构函数 .....	217
10.12 静态构造函数 .....	218
10.12.1 类加载和初始化 .....	219
<b>11. 结构 .....</b>	<b>221</b>
11.1 结构声明 .....	221
11.1.1 结构修饰符 .....	221
11.1.2 接口 .....	221
11.1.3 结构体 .....	221
11.2 结构成员 .....	221
11.3 结构例子 .....	221
11.3.1 数据库整数类型 .....	221
11.3.2 数据库布尔类型 .....	223
<b>12. 数组 .....</b>	<b>225</b>
12.1 数组类型 .....	225
12.1.1 System.Array 类型 .....	226
12.2 数组创建 .....	226
12.3 数组元素访问 .....	226

12.4 数组成员 .....	226
12.5 数组协方差 .....	226
12.6 数组初始化函数 .....	227
<b>13. 接口 .....</b>	<b>229</b>
13.1 接口声明 .....	229
13.1.1 接口修饰符 .....	229
13.1.2 基本接口 .....	229
13.1.3 接口主体 .....	230
13.2 接口成员 .....	230
13.2.1 接口方法 .....	231
13.2.2 接口属性 .....	231
13.2.3 接口事件 .....	232
13.2.4 接口索引 .....	232
13.2.5 接口成员访问 .....	232
13.3 完全有效的接口成员名称 .....	234
13.4 接口实现 .....	234
13.4.1 显式接口成员实现程序 .....	235
13.4.2 接口映射 .....	237
13.4.3 接口实现程序继承 .....	239
13.4.4 接口重新实现程序 .....	240
13.4.5 抽象类和接口 .....	242
<b>14. 枚举 .....</b>	<b>243</b>
14.1 枚举声明 .....	243
14.2 枚举成员 .....	244
14.3 枚举数值和操作 .....	246
<b>15. 代表 .....</b>	<b>247</b>
15.1 代表声明 .....	247
15.1.1 可合并的代表类型 .....	248
15.2 代表实例化 .....	248
15.3 多点传送代表 .....	248
15.4 代表调用 .....	248
<b>16. 异常 .....</b>	<b>249</b>
16.1 异常的产生 .....	249
16.2 System.Exception 类 .....	249
16.3 异常怎样被处理 .....	249
16.4 通用异常类 .....	250
<b>17. 属性 .....</b>	<b>251</b>
17.1 属性类 .....	251
17.1.1 AttributeUsage 属性 .....	251
17.1.2 位置的和名称的参数 .....	252
17.1.3 属性参数类型 .....	252
17.2 规范 .....	253

17.3 属性实例 .....	255
17.3.1 一个属性的编译 .....	255
17.3.2 一个属性实例的运行时检索 .....	256
17.4 保留的属性 .....	256
17.4.1 AttributeUsage 属性 .....	256
17.4.2 条件属性 .....	257
17.4.3 废弃的属性 .....	259
<b>18. 危险代码 .....</b>	<b>261</b>
18.1 危险代码 .....	261
18.2 指针类型 .....	261
<b>19. 互用性 .....</b>	<b>263</b>
19.1 COMImport 属性 .....	263
19.2 COMRegisterFunction 属性 .....	263
19.3 COMSourceInterfaces 属性 .....	264
19.4 COMVisible 属性 .....	264
19.5 DispId 属性 .....	264
19.6 DllImport 属性 .....	264
19.7 FieldOffset 属性 .....	265
19.8 GlobalObject 属性 .....	265
19.9 Guid 属性 .....	266
19.10 HasDefaultInterface 属性 .....	266
19.11 ImportedFromTypeLib 属性 .....	266
19.12 In 和 Out 属性 .....	266
19.13 InterfaceType 属性 .....	267
19.14 MarshalAs 属性 .....	267
19.15 NoIDispatch 属性 .....	268
19.16 NonSerialized 属性 .....	268
19.17 Predeclared 属性 .....	268
19.18 Preservesig 属性 .....	268
19.19 Serializable 属性 .....	269
19.20 StructLayout 属性 .....	269
19.21 TypeLibFunc 属性 .....	269
19.22 TypeLibType 属性 .....	269
19.23 TypeLibVar 属性 .....	270
19.24 支持的枚举 .....	270
<b>20. 参考 .....</b>	<b>273</b>



# 1. 介绍

C#是一种简单、现代、面向对象和类型安全的编程语言，由 C 和 C++ 发展而来。C#（发音为“C 霏普”）牢固地植根于 C 和 C++ 语言族谱中，并且会很快被 C 和 C++ 程序员所熟悉。C# 的目标在于把 Visual Basic 的高生产力和 C++ 本身的能力结合起来。

C# 作为 Microsoft Visual Studio 7.0 的一部分提供给用户。除了 C# 以外，Visual Studio 还支持 Visual Basic、Visual C++ 和描述语言 VBScript 和 Jscript。所有这些语言都提供对 Microsoft .NET 平台的访问能力，它包括一个通用的执行引擎和一个丰富的类库。Microsoft .NET 平台定义了一个“通用语言子集” (CLS)，是一种混合语言，它可以增强 CLS 兼容语言和类库间的无缝协同工作能力。对于 C# 开发者，这意味着既是 C# 是一种新的语言，它已经可以对用老牌工具如 Visual Basic 和 Visual C++ 使用的丰富类库进行完全访问。C# 自己并没有包含一个类库。

本章的其余部分描述这种语言的基本特性。以后的章节将会详细描述规则和例外，并且有些时候以数学的方式描述，而这章会致力于对整体的简单而清楚地介绍。这样的目的是给读者一个关于语言的介绍，这样可以使读者可以更容易地开始编写程序和继续阅读后面的章节。

## 1.1 Hello, world

规范的“Hello, World”程序可以按照下面例子编写：

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

C# 程序的源代通常存储在一个或多个扩展名为 .cs 的文件中，例如 hello.cs。如果使用 Visual Studio 提供的命令行编译器，这样的程序可以用命令行命令来编译

```
csc hello.cs
```

这样就会生成一个名为 hello.exe 的可执行程序。程序的输出如下：

```
Hello, world
```

下面对这个程序进行详细的研究：

- 使用 System；指令涉及到一个名称空间(namespace)叫作 System，这是在 Microsoft .NET 类库中提供的。这个名称空间包括在 Main 方法中使用的 Console 类。名称空间提供了一种用来组织一个类库的分层方法。使用“using”命令后，就可以无障碍地使用名称空间中的各种类型成员。“Hello, world”程序中使用的 Console.WriteLine 是 System.Console.WriteLine 的简写。
- Main 方法是类 Hello 中的一个成员，它有 static 的说明符，所以它是类 Hello 中的一个方法而不是此类中的实例。
- 对于一个应用程序的主入口点 - 称开始执行的方法 - 通常是一个称为 Main 的静态方法。
- “Hello, world”的输出是通过使用类库产生的。语言本身并没有提供类库。作为替代，它使用一个通用类库，这个类库也可以被诸如 Visual Basic 和 Visual C++ 的语言所使用。

对于 C 和 C++ 开发者来说，会有兴趣知道对一些没有出现在 “Hello, world” 程序的东西。

- 程序没有把 Main 设为全局方法。在全局级别上不支持方法和变量；这些元素通常包含在类型声明当中（例如，类或结构的声明）。
- 程序中不使用 “::” 或 “->” 操作符。“::” 不再是一个操作符，而 “->” 操作符也只是在程序的某个小片断中才会使用。操作符 “.” 用于符合名称，例如 Console.WriteLine。
- 程序中不包括前向声明。因为声明的顺序不重要，所以不再需要前向声明。
- 程序中不使用 #include 关键字。程序中的从属关系是象征性的而不是字面上地。这个系统消除了在用不同语言编写的程序间的障碍。例如，Console 类可以用另外一种语言编写。

## 1.2 类型

C# 支持两种类型：数据类型和引用类型。数据类型包括一些简单类型（例如，char、int 和 float），枚举类型和结构类型。引用类型包括类类型、接口类型、代表(delegate)类型和数组类型。

数据类型和引用类型的区别在于，数据类型变量直接包含它们的数据，然而引用类型数据是存储对于对象的引用。对于引用类型，有可能两个变量引用相同的对象，因而可能出现对一个变量的操作影响到其它变量所引用对象的情况。对于数据类型，每个变量都有它们自己对数据的拷贝，所以不太可能因为对一个进行操作而影响到其它变量。

例子

```
using System;
class Class1
{
    public int Value = 0;
}
class Test
{
    static void Main() {
        int val1 = 0;
        int val2 = val1;
        val2 = 123;

        Class1 ref1 = new Class1();
        Class1 ref2 = ref1;
        ref2.Value = 123;

        Console.WriteLine("Values: {0}, {1}", val1, val2);
        Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
    }
}
```

从此例可以说明两者间的不同。程序的输出如下

```
values: 0, 123
Refs: 123, 123
```

对局部变量 val1 的赋值没有影响到局部变量 val2，因为两个局部变量都是数据类型（int 类型），并且每个数据类型的局部变量都有它们自己的存储。与此相对的是，对于 ref.Value 的赋值 ref.Value=123 对 ref1 和 ref2 都有影响。

这两行

```
Console.WriteLine("Values: {0}, {1}", val1, val2);
Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
```

值得更多的注释，因为它们可以让我们看到 Console.WriteLine 的一些字符串格式输出方式，这些实际上是使用了一些变量。第一个变量是一个字符串，它包含一些数字的占位符如{0}和{1}。每个占位符指向一个变量。占位符{0}指向第二个变量，占位符{1}指向第三个变量，等等。在输出被送到控制台前，这些占位符会被它们相对应的变量所替换。

开发者可以通过枚举和结构声明定义新数据类型，可以通过类、接口和代表声明来定义新引用类型。例子

```
using System;
public enum Color
{
    Red, Blue, Green
}
public struct Point
{
    public int x, y;
}
public interface IBase
{
    void F();
}
public interface IDerived: IBase
{
    void G();
}
public class A
{
    protected void H() {
        Console.WriteLine("A.H");
    }
}
public class B: A, IDerived
{
    public void F() {
        Console.WriteLine("B.F, implementation of IDerived.F");
    }
    public void G() {
        Console.WriteLine("B.G, implementation of IDerived.G");
    }
    override protected void H() {
        Console.WriteLine("B.H, override of A.H");
    }
}
public delegate void EmptyDelegate();
```

这里用一两个例子说明每种类型声明，以后的章节里会更详细地描述类型声明。

### 1.2.1 预定义类型

C#提供了一系列预定义类型，其中大多数对 C 和 C++程序员来说都是比较熟悉的。

预定义引用类型是对象和字符串。类型对象是所有其它类型的最根本的基础类型，而类型字符串要用来说明 Unicode 字符串数据。

预定义数据类型包括有符号和无符号整数类型、浮点数类型、二进制、字符和十进制类型。有符号整数类型有 sbyte、short、int 和 long；无符号整数类型有 byte、ushort、uint 和 ulong；而浮点类型有 float 和 double。

二进制类型用来表示二进制数据:值或者是真或者是假。包含二进制使得编写自说明代码变得容易，并且也帮助消除所有由于程序员在应当使用 “ == ” 时错误的使用了 “ = ” 造成的很普通的 C++代码错误。在 C#中,下面的例子

```
int i = ...;
F(i);
if (i = 0) // Bug: the test should be (i == 0)
    G();
```

是非法的，因为表达式 i = 0 的类型是 int，而 if 声明需要一个二进制类型的表达式。

Char 类型用来说明 Unicode 字符。某个 char 类型变量说明一个单 16 位 Unicode 字符。

十进制类型适合应用在不能接受舍入误差计算中。通常的例子包括商业计算，例如税收计算和货币转换。十进制类型提供了 28 个有效位。

下面的表中列出了预定义类型，并且指出了如何为每一个类型赋值。

类型	描述	例子
object	所有其它类型的最根本的基础类型	object o = null;
string	字符串类型；一个字符传是一个 Unicode 字符序列	string s = "Hello";
sbyte	8-bit 有符号整数类型	sbyte val = 12;
short	16-bit 有符号整数类型	short val = 12;
int	32-bit 有符号整数类型	int val = 12;
long	64-bit 有符号整数类型	long val1 = 12; long val2 = 34L;
byte	8-bit 无符号整数类型	byte val1 = 12; byte val2 = 34U;
ushort	16-bit 无符号整数类型	ushort val1 = 12; ushort val2 = 34U;
uint	32-bit 无符号整数类型	uint val1 = 12; uint val2 = 34U;
ulong	64-bit 无符号整数类型	ulong val1 = 12; ulong val2 = 34U; ulong val3 = 56L; ulong val4 = 78UL;
float	单精度浮点数类型	float val = 1.23F;
double	双精度浮点数类型	double val1 = 1.23; double val2 = 4.56D;
bool	二进制类型; 一个二进制数据不是真就是假	bool val1 = true; bool val2 = false;
char	字符类型; 一个字符数据是一个 Unicode 字符	char val = 'h';
decimal	精确十进制类型，有 28 个有效位	decimal val = 1.23M;



每一个预定义类型都是某个系统提供的类型的简写。例如，关键词 `int` 是一个名为 `System.Int32` 结构的简写。虽然更好的考虑是使用关键词而不是完全的系统类型名称，但是这两个名称可以交换使用。

例如 `int` 的预定义数据类型在某些地方被认为是特别的，但在大多数地方会像其它结构一样被正确对待。操作符重载使得编程人员可以定义同预定义数据类型行为相同的类型。例如，一个 `Digit` 结构可以支持与整数类型相同的数学操作，并且可以定义 `Digit` 和预定义类型间的转换。

预定义类型可以允许操作符重载它们自己。例如，比较符 `=` 和 `!=` 对应不同的预定义类型有不同的语意：

- 如果两个 `int` 类型的表达式代表了相同的整数据，它们被认为是相等的。
- 如果两个 `object` 类型的表达式都指向相同的对象或者都是空的，它们被认为是相等的。
- 如果字符串实例有相同的长度并且在每个字符的位置都相同，或者都为空，这两个字符串类型的表达式就被认为是相等的。

例子

```
class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

产生下面的输出

```
True
False
```

因为第一个比较符比较两个 `string` 类型的表达式，而第二个比较符比较两个 `object` 类型的表达式。

### 1.2.2 转换

有两种类型的转换隐式转换和显式转换。隐式转换应用于需要小心地仔细检查就可以安全地实现的转换。例如，从 `int` 到 `long` 就是一个隐式转换。隐式转换通常都能成功，并且不会带来失去信息的后果。就像例子中所示，隐式转换可以隐式地实现。

```
using System;
class Test
{
    static void Main() {
        int intValue = 123;
        long longValue = intValue;
        Console.WriteLine("(long) {0} = {1}", intValue, longValue);
    }
}
```

这里隐式地把 `int` 转换为 `long`。

相反，显式转换必须要个安排好的表达式才能实现。例子

```
using System;
```

```

class Test
{
    static void Main() {
        long longValue = Int64.MaxValue;
        int intValue = (int) longValue;
        Console.WriteLine("(int) {0} = {1}", longValue, intValue);
    }
}

```

用显式转换把 long 转换到 int。输出为：

```
(int) 9223372036854775807 = -1
```

因为发生溢出。

### 1.2.3 数组类型

数组可以是一维或多维。支持“矩形”数组也支持“不规则”数组。

一维数组是最普通的类型，所以，从这里开始讲是个不错的开始。例子

```

using System;
class Test
{
    static void Main() {
        int[] arr = new int[5];
        for (int i = 0; i < arr.Length; i++)
            arr[i] = i * i;
        for (int i = 0; i < arr.Length; i++)
            Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
    }
}

```

创建一个一维 int 数据数组，初始化数组元素并且把每个元素打印出来。程序的输出为：

```

arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16

```

前面例子中使用的 int[] 类型是一个数组类型。数组类型的写法是前面一个非数组类型，其后跟一个或多个对中括号。例子

```

class Test
{
    static void Main() {
        int[] a1;           // single-dimensional array of int
        int[,] a2;          // 2-dimensional array of int
        int[,,] a3;         // 3-dimensional array of int
        int[][] j2;         // "jagged" array: array of (array of int)
        int[][][] j3;       // array of (array of (array of int))
    }
}

```

介绍了各种使用 int 类型元素的数组类型定义局部变量的方法。

数组是引用类型，所以声明一个数组变量只是为对此数组的引用设置了空间。数组实例的实际创建是通过数组初始化程序和数组创建表达式。例子

```

class Test
{
    static void Main() {
        int[] a1 = new int[] {1, 2, 3};
        int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
        int[,,] a3 = new int[10, 20, 30];

        int[][] j2 = new int[3][];
        j2[0] = new int[] {1, 2, 3};
        j2[1] = new int[] {1, 2, 3, 4, 5, 6};
        j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
    }
}

```

给出了各种数组创建表达式。变量 a1、a2 和 a3 表示矩形数组，而变量 j2 表示了一个不规则数组。这里没必要对在这里按照数组的形状分类感到惊讶。矩形数组通常有矩形的形状，给出数组每一维的长度，它的矩形形状就清楚了。例如，a3 的三维长度分别为 10、20 和 30，并且不难看出这个数组包括 10\*20\*30 个元素。

相反，变量 j2 表示一个“不规则”数组，或者称为“数组的数组”。特别是，j2 表示了一个 int 数组组成的数组，或者说是一个一维 int[] 类型的数组。这些 int[] 变量中的每一个都可以独自被初始化，同时允许数组有一个不规则形状。例子中为每个 int[] 数组定义了不同的长度。很清楚地可以看到，j2[0] 的长度是 3，j2[1] 的长度是 6，j2[2] 的长度是 9。

元素的类型和数组的维数是一个数组类型的一部分，但每一维的长度不是。这项不同从语法上看更清楚，每一维的长度是在数组创建表达式中指出而不是在数组类型中。例如下面的声明

```
int[,,] a3 = new int[10, 20, 30];
```

为 int[,,] 类型的数组和一个数组创建表达式 new int[10,20,30]。

对于局部变量和域声明，允许一种简写形式，这样就不需要去再声明数组类型。例如，下面的例子

```
int[] a1 = new int[] {1, 2, 3};
```

可以简化为

```
int[] a1 = {1, 2, 3};
```

程序语意没有任何变化。

一个数组初始化程序的上下文，例如 {1,2,3}，被用来决定数组被初始化的类型。例子

```

class Test
{
    static void Main() {
        short[] a = {1, 2, 3};
        int[] b = {1, 2, 3};
        long[] c = {1, 2, 3};
    }
}

```

说明相同的数组初始化程序可以用于许多不同数组类型的初始化。因为需要上下文来决定一个数组初始化程序的类型，所以不可能在表达式上下文中使用数组初始化程序。

#### 1.2.4 类型系统一致

C# 提供了“统一类型系统”。包括数据类型的所有类型都从 object 类型派生出来。可以从任何数据调用 object 的方法，甚至像 int 这样“简单的”类型的数据。例子

```
using System;
```

```
class Test
{
    static void Main() {
        Console.WriteLine(3.ToString());
    }
}
```

用一个整数数据符号来调用在 object 中定义的 ToString 方法。

下面这个例子

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

更加有趣。一个 int 数据可以被转换为 object 并且还可以转换回 int 类型。这个例子说明了 *boxing* 和 *unboxing* 当一个数据类型变量需要转换为引用类型时，一个名为 box 的对象被分配来保存数据，并且数据被拷贝到 box 中。Unboxing 恰恰相反。当一个对象 box 要恢复到它原始的数据类型的时候，数据被从对象 box 中拷贝到适当的存储位置。

类型系统一致提供了有对象性质的数值，而不用引入不需要的开支。在不需要 int 数值表现得像对象的程序中，int 数值只是简单的 32 位数值。对于需要 int 数值的行为像一个对象的程序，这项能力是可以实现的。这个能力把数值类型当做大多数语言中存在的数值类型和引用类型间的桥梁。例如类 Stack 可以提供 Push 和 Pop 方法，并返回一个 object 数值。

```
public class Stack
{
    public object Pop() {...}
    public void Push(object o) {...}
}
```

因为 C# 有统一类型系统，Stack 类可以用来为任何类型数据生成堆栈，包括像 int 这样的数据类型。

### 1.3 变量和参数

变量扮演存储的角色。每个变量有一个类型，这个类型决定那些数据可以被存储在这个变量中。局部变量是在方法、属性或索引中声明的变量。一个局部变量通常通过指定的类型名称和说明符来定义，它指定了变量名称和一个任意的初始值，如下：

```
int a;
int b = 1;
```

但也有可能一个局部变量声明包括多个说明符。对于 a 和 b 的声明可以写成：

```
int a, b = 1;
```

一个变量在它的数据可以使用前，必须被明确分配数据(\$**错误！未找到引用源。**)。例子

```

class Test
{
    static void Main() {
        int a;
        int b = 1;
        int c = a + b;
        ...
    }
}

```

是非法的，因为试图在一个变量被分配数据前就试图使用它。

域 (§10.4) 是一种变量，它与某个类或结构或者某个类或结构的实例相关联。一个用 `static` 修饰符声明的域定义了一个静态变量，而不用这种修饰符声明的域定义一个实例变量。例子

```

using System.Data;
class Employee
{
    private static DataSet ds;
    public string Name;
    public decimal Salary;
    ...
}

```

介绍了 `Employee` 类，它有一个私有静态变量和两个公用实例变量。

形式参数声明同样定义变量。这里有四种类型的参数：数据参数，引用参数，输出参数和参量（`param`）参数。

数据参数用来做“入”参数传递，一个自变量的数据通过它传递到方法中，而对参数的修改不会影响到原始的自变量。数据参数指向它自己在存储器中的位置，它与变量存储位置有明确的区分。次存储位置通过把拷贝相应变量的数据来初始化。例子

```

using System;
class Test {
    static void F(int p) {
        Console.WriteLine("p = {0}", p);
        p++;
    }
    static void Main() {
        int a = 1;
        Console.WriteLine("pre: a = {0}", a);
        F(a);
        Console.WriteLine("post: a = {0}", a);
    }
}

```

说明了一个方法 `F`，它有一个名为 `p` 的数据参数。这个例子产生下面的输出：

```

pre: a = 1
p = 1
post: a = 1

```

甚至数据 `p` 被改动。

引用参数是用作“通过引用”参数传递，这里，参数表现为调用者提供变量的别名。引用参数自己并不定义存储位置，而是指向相应变量的存储位置。对引用参数的修改马上会直接地影响到相应的变量。引用参数用一个 `ref` 修饰符来声明。例子

```

using System;

```

```

class Test {
    static void Swap(ref int a, ref int b) {
        int t = a;
        a = b;
        b = t;
    }

    static void Main() {
        int x = 1;
        int y = 2;

        Console.WriteLine("pre: x = {0}, y = {1}", x, y);
        Swap(ref x, ref y);
        Console.WriteLine("post: x = {0}, y = {1}", x, y);
    }
}

```

说明了有两个引用参数的方法 Swap。程序的输出如下：

```

pre: x = 1, y = 2
post: x = 2, y = 1

```

关键词 `ref` 必须在形式参数中声明并且在其中使用。在 `call` 位置使用 `ref` 要求对参数特殊注意，这样，一个开发人员在阅读此段代码的时候就可以理解到，由于此调用变量将发生变化。

除了调用者所提供变量的初始化数据不重要以外，输出参数与引用参数相似。用一个 `out` 修饰符来声明一个输出参数。例子

```

using System;

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

    static void Main() {
        for (int i = 1; i < 10; i++)
            for (int j = 1; j < 10; j++) {
                int ans, r;
                Divide(i, j, out ans, out r);
                Console.WriteLine("{0} / {1} = {2}r{3}", i, j, ans, r);
            }
    }
}

```

介绍了一个包括两个输出参数的 `Divide` 方法，一个是除的结果，另外一个为余数。

对于数据，引用和输出参数在调用者提供的变量和代表它们的参数间有一个一一对应的关系。参量参数可以允许多对一的关系：多个变量可以由一个参量参数来代表。换句话说参量参数可以接受长度变化的变量列表。

参量参数用一个 `params` 修饰符来声明。对于一个给定的方法，只能有一个参量参数，并且通常指定为最后一个参数。参量参数通常是一维数组类型。调用程序可以只是传送这种数组类型的一个单独的变量，也可以是这种数组类型中，于数组元素类型相同的任意多个变量。例如，下面的例子

```

using System;

```

```

class Test
{
    static void F(params int[] args) {
        Console.WriteLine("# of arguments: {0}", args.Length);
        for (int i = 0; i < args.Length; i++)
            Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(new int[] {1, 2, 3, 4});
    }
}

```

介绍了一个 F 方法，它有可变数量的 int 变量和许多对这个方法的调用。输出是：

```

# of arguments: 0
# of arguments: 1
    args[0] = 1
# of arguments: 2
    args[0] = 1
    args[1] = 2
# of arguments: 3
    args[0] = 1
    args[1] = 2
    args[2] = 3
# of arguments: 4
    args[0] = 1
    args[1] = 2
    args[2] = 3
    args[3] = 4

```

在介绍中出现的大部分例子都使用 Console 类中的 WriteLine 方法。如例子中完全用参量参数进行变量替换。

```

int a = 1, b = 2;
Console.WriteLine("a = {0}, b = {1}", a, b);

```

WriteLine 方法提供了多种传递少量变量的方法，而其中一种使用了参量参数。

```

namespace System
{
    public class Console
    {
        public static void WriteLine(string s) {...}
        public static void WriteLine(string s, object a) {...}
        public static void WriteLine(string s, object a, object b) {...}
        ...
        public static void WriteLine(string s, params object[] args) {...}
    }
}

```

## 1.4 自动内存管理

手工内存管理需要开发者管理内存块的分配和重新分配。手工管理内存既耗时又困难。提供自动内存管理可以使开发者从繁重的任务中解放出来。在大多数情况下，自动内存管理可以在没有反面影响表现和性能的情况下增加代码的质量，提高开发者生产力。

## 例子

```

using System;
public class Stack
{
    private Node first = null;
    public bool Empty {
        get {
            return (first == null);
        }
    }
    public object Pop() {
        if (first == null)
            throw new Exception("Can't Pop from an empty Stack.");
        else {
            object temp = first.Value;
            first = first.Next;
            return temp;
        }
    }
    public void Push(object o) {
        first = new Node(o, first);
    }
    class Node
    {
        public Node Next;
        public object Value;
        public Node(object value): this(value, null) {}
        public Node(object value, Node next) {
            Next = next;
            Value = value;
        }
    }
}

```

介绍了一个作为 Node 实例链接表执行类 Stack。Node 实例在 Push 方法中创建，当不再需要时被碎片收集。当其它程序没有任何可能去访问 Node 实例时，它就符合碎片收集的条件了。例如当一个项目被从 Stack 中移走，相关的 Node 实例就变为符合碎片收集的条件。

## 例子

```

class Test
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 0; i < 10; i++)
            s.Push(i);
        s = null;
    }
}

```

介绍了一个使用 Stack 类的测试程序。Stack 被创建并且初始化为包含 10 个元素，然后被赋值为数据 null。一旦变量 s 被赋值为 null 后，Stack 和相应的 10 个 Node 实例就变成符合碎片收集的条件了。碎片收集程序马上就被允许进行清理，虽然还不需要这样做。



对于一个通常是使用自动内存管理，但有时需要精细控制或希望有些许性能提高的程序员来说，C#提供了编写“非安全”代码的能力。这样的代码可以由指针类型直接处理，而且 `fix` 对像可以暂时保护这些代码，防止被碎片收集程序收集。从开发者和用户角度来看，这个“非安全”代码属性实际上是“安全”属性。非安全代码必须用 `unsafe` 修饰符明确标明，这样开发者就不会在偶然的情况下使用这项非安全属性，并且编译器和执行程序一起来保证非安全代码不会伪装为安全代码。

例子

```
using System;
class Test
{
    unsafe static void WriteLocations(byte[] arr) {
        fixed (byte *p_arr = arr) {
            byte *p_elem = p_arr;
            for (int i = 0; i < arr.Length; i++) {
                byte value = *p_elem;
                string addr = int.Format((int) p_elem, "x");
                Console.WriteLine("arr[{0}] at 0x{1} is {2}", i, addr, value);
                p_elem++;
            }
        }
    }
    static void Main() {
        byte[] arr = new byte[] {1, 2, 3, 4, 5};
        WriteLocations(arr);
    }
}
```

介绍了名为 `WriteLocations` 的非安全方法，它选定一个数组实例并且用指针反复对元素进行操作。每个数组元素的标号，数据和位置写到控制台。这个程序的一个可能的输出为：

```
arr[0] at 0x8E0360 is 1
arr[1] at 0x8E0361 is 2
arr[2] at 0x8E0362 is 3
arr[3] at 0x8E0363 is 4
arr[4] at 0x8E0364 is 5
```

但是，准确的存储位置肯定要发生变化。

## 1.5 表达式

C# 包括一元操作符，二元操作符和一个三元操作符。在下面的表中，对操作符进行了总结，按照从最高到最低的优先顺序列出这些操作符：

Section	Category	Operators
0	基本的	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
错误！ 未找到 引用 源。	一元的	+ - ! ~ ++x --x (T)x
错误！ 未找到 引用 源。	乘法的	* / %
错误！ 未找到 引用 源。	加法的	+ -
错误！ 未找到 引用 源。	移位	<< >>
错误！ 未找到 引用 源。	关系	< > <= >= is
错误！ 未找到 引用 源。	等式	== !=
错误！ 未找到 引用 源。	逻辑与	&
错误！ 未找到 引用 源。	逻辑异或	^
错误！ 未找到 引用 源。	逻辑或	
错误！	条件与	&&

未找到引用源。		
错误！未找到引用源。	条件或	
错误！未找到引用源。	条件的	?:
错误！未找到引用源。	赋值	= *= /= %= += -= <<= >>= &= ^=  =

当一个表达式包括几个操作符，操作符的优先级控制顺序，这里对每个单独操作符进行等效。例如，因为\*操作符的优先级比+操作符高，所以表达式  $x+y*z$  等效为  $x+(y*z)$ 。

当一个操作数出现在两个有相同优先级的操作符之间时，操作符的传递关系控制操作实现的顺序：

- 除了赋值操作符，所有二元操作符是左向传递，意味着操作是从左向右进行。例如， $x+y+z$  等效为  $(x+y)+z$ 。
- 赋值操作符和条件操作符是右向传递的，意味着操作是从右向左进行。例如， $x=y=z$  等效为  $x=(y=z)$ 。

使用括号可以控制优先级和传递关系。例如， $x+y*z$  先把  $y$  和  $z$  相乘，然后把结果同  $x$  相加。但是  $(x+y)*z$  首先把  $x$  和  $y$  相加，然后把结果乘以  $z$ 。

### 1.6 声明

虽然有些值得注意的增加和修改，C# 的大多数声明都是从 C 和 C++ 继承的。下面的表列出了可用的声明的类型，并且为每个提供了例子。

声明	例子
声明列表和块声明	<pre>static void Main() {     F();     G();     {         H();         I();     } }</pre>
标号声明和 goto 声明	<pre>static void Main(string[] args) {     if (args.Length == 0)         goto done;     Console.WriteLine(args.Length);  done:     Console.WriteLine("Done"); }</pre>
局部常量声明	<pre>static void Main() {     const float pi = 3.14;     const int r = 123;     Console.WriteLine(pi * r * r); }</pre>
局部变量声明	<pre>static void Main() {     int a;     int b = 2, c = 3;     a = 1;     Console.WriteLine(a + b + c); }</pre>
表达式声明	<pre>static int F(int a, int b) {     return a + b; }  static void Main() {     F(1, 2); // Expression statement }</pre>
If 声明	<pre>static void Main(string[] args) {     if (args.Length == 0)         Console.WriteLine("No args");     else         Console.WriteLine("Args"); }</pre>
Switch 声明	<pre>static void Main(string[] args) {     switch (args.Length) {         case 0:             Console.WriteLine("No args");             break;         case 1:             Console.WriteLine("One arg ");             break;         default:             int n = args.Length;             Console.WriteLine("{0} args", n);             break;     } }</pre>
while 声明	<pre>static void Main(string[] args) {     int i = 0;     while (i &lt; args.length) {         Console.WriteLine(args[i]);         i++;     } }</pre>
do 声明	<pre>static void Main() {     string s;</pre>

	<pre>do { s = Console.ReadLine(); } while (s != "Exit"); }</pre>
for 声明	<pre>static void Main(string[] args) {     for (int i = 0; i &lt; args.Length; i++)         Console.WriteLine(args[i]); }</pre>
Foreach 声明	<pre>static void Main(string[] args) {     foreach (string s in args)         Console.WriteLine(s); }</pre>
Break 声明	<pre>static void Main(string[] args) {     int i = 0;     while (true) {         if (i &gt; args.Length)             break;         Console.WriteLine(args[i++]);     } }</pre>
continue 声明	<pre>static void Main(string[] args) {     int i = 0;     while (true) {         Console.WriteLine(args[i++]);         if (i &gt; args.Length)             continue;         break;     } }</pre>
return 声明	<pre>static int F(int a, int b) {     return a + b; }  static void Main() {     Console.WriteLine(F(1, 2));     return; }</pre>
throw 声明 and try 声明	<pre>static int F(int a, int b) {     if (b == 0)         throw new Exception("Divide by zero");     return a / b; }  static void Main() {     try {         Console.WriteLine(F(5, 0));     }     catch (Exception e) {         Console.WriteLine("Error");     } }</pre>
checked 和 unchecked 声明	<pre>static void Main() {     int x = 100000, y = 100000;     Console.WriteLine(checked(x * y));     Console.WriteLine(checked(x * y)); // Error     Console.WriteLine(x * y);          // Error }</pre>
lock 声明	<pre>static void Main() {     A a = ...     lock(a) {         a.P = a.P + 1;     } }</pre>

问题

我们要在这里添加一些章节说明 C# 与 C 和 C++ 不同的地方。这些应该是：

- *Goto 约束*
- *不是所有的表达式都可用被用作声明*
- *if, while, 和 do 声明需要二进制表达式*
- *对于 switch 声明没有失败*
- *foreach 声明*
- *例外处理*
- *Checked 和 unchecked 声明*
- *Lock 声明*

### 1.7 类

类声明定义新的引用类型。一个类可以从其它类继承，并且可以没用接口或有多个接口。

类的成员可以包括常数、域、方法、属性、索引、事件、操作符、构造函数、析构器和嵌套类型声明。每个成员有相关的访问能力，这控制了可以访问这个成员的程序文本的区域。有访问能力有五种可能形式。在下表中进行总结。

形式	直观意义
public	访问不受限制
protected	访问只限于此程序或类中包含的类型
internal	访问只限于此程序
protected internal	访问只限于此程序或类中包含的类型
private	访问只限于所包含的类型

例子

```
using System;
class MyClass
{
    public MyClass() {
        Console.WriteLine("Constructor");
    }
    public MyClass(int value) {
        MyField = value;
        Console.WriteLine("Constructor");
    }
    ~MyClass() {
        Console.WriteLine("Destructor");
    }
    public const int MyConstant = 12;
    public int MyField = 34;
```

```

public void MyMethod(){
    Console.WriteLine("MyClass.MyMethod");
}
public int MyProperty {
    get {
        return MyField;
    }
    set {
        MyField = value;
    }
}
public int this[int index] {
    get {
        return 0;
    }
    set {
        Console.WriteLine("this[{0}] was set to {1}", index, value);
    }
}
public event EventHandler MyEvent;
public static MyClass operator+(MyClass a, MyClass b) {
    return new MyClass(a.MyField + b.MyField);
}
internal class MyNestedClass
{}
}

```

介绍了一个包含每种类型成员类。例子

```

class Test
{
    static void Main() {
        // Constructor usage
        MyClass a = new MyClass();
        MyClass b = new MyClass(123);

        // Constant usage
        Console.WriteLine("MyClass.MyConstant = {0}", MyClass.MyConstant);

        // Field usage
        a.MyField++;
        Console.WriteLine("a.MyField = {0}", a.MyField);

        // Method usage
        a.MyMethod();

        // Property usage
        a.MyProperty++;
        Console.WriteLine("a.MyProperty = {0}", a.MyProperty);

        // Indexer usage
        a[3] = a[1] = a[2];
        Console.WriteLine("a[3] = {0}", a[3]);

        // Event usage
        a.MyEvent += new EventHandler(MyHandler);

        // overloaded operator usage
        MyClass c = a + b;
    }

    static void MyHandler(object sender, EventArgs e) {
        Console.WriteLine("Test.MyHandler");
    }
}

```

```

        internal class MyNestedClass
        {}
    }

```

介绍如何使用这些成员。

### 1.7.1 常数

一个常数是一个代表常数值的类成员：某个可以在编译时计算的数值。只要没有循环从属关系，允许常数依赖同一程序中的其它常数。例子

```

class Constants
{
    public const int A = 1;
    public const int B = A + 1;
}

```

包括一个名为 Constants 的类，有两个公共常数。

常数是隐式静态类型，可以通过类来访问，

```

class Test
{
    static void Main() {
        Console.WriteLine("A = {0}, B = {1}", Constants.A, Constants.B);
    }
}

```

在此例中打印输出 Constants.A 和 Constants.B 的数值。

### 1.7.2 域

域是一个代表和某对象或类相关的变量的成员。例子

```

class Color
{
    internal ushort redPart;
    internal ushort bluePart;
    internal ushort greenPart;

    public Color(ushort red, ushort blue, ushort green) {
        redPart = red;
        bluePart = blue;
        greenPart = green;
    }

    ...
}

```

介绍了一个 Color 类，它有局部的实例域，分别叫做 redPart、greenPart 和 bluePart。域可以是静态的，在下面的例子中

```

class Color
{
    public static Color Red = new Color(0xFF, 0, 0);
    public static Color Blue = new Color(0, 0xFF, 0);
    public static Color Green = new Color(0, 0, 0xFF);
    public static Color White = new Color(0, 0, 0);
    public static Color Black = new Color(0xFF, 0xFF, 0xFF);
    ...
}

```

介绍了 Red、Blue、Green、White 和 Black 的静态域。



静态域在这里并不太合适。当 Color 类被加载后，域就被初始化了，但是在初始化之后，并不能阻止用户改变它们。而那樣的改动可能会引起别的使用 Color 并认为数值不会变的程序的不可预见的错误。只读域可以用来避免这一错误的发生。对于一个只读域的赋值，只会在相同类中的部分声明和构造函数中发生。这样，就可以通过给静态域添加只读修饰符来增强 Color 类：

```
class Color
{
    internal ushort redPart;
    internal ushort bluePart;
    internal ushort greenPart;

    public Color(ushort red, ushort blue, ushort green) {
        redPart = red;
        bluePart = blue;
        greenPart = green;
    }

    public static readonly Color Red = new Color(0xFF, 0, 0);
    public static readonly Color Blue = new Color(0, 0xFF, 0);
    public static readonly Color Green = new Color(0, 0, 0xFF);
    public static readonly Color white = new Color(0, 0, 0);
    public static readonly Color Black = new Color(0xFF, 0xFF, 0xFF);
}
```

### 1.7.3 方法

方法是一个执行可以由对像或类完成的计算或行为的成员。方法有一个形式参数列表（可能为空），一个返回数值（或 void），并且可以是静态也可以是非静态。静态方法要通过类来访问。非静态方法，也称为实例方法，通过类的实例来访问。例子

```
using System;

public class Stack
{
    public static Stack Clone(Stack s) {...}
    public static Stack Flip(Stack s) {...}
    public object Pop() {...}
    public void Push(object o) {...}
    public override string ToString() {...}
    ...
}

class Test
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 1; i < 10; i++)
            s.Push(i);

        Stack flipped = Stack.Flip(s);
        Stack cloned = Stack.Clone(s);

        Console.WriteLine("Original stack: " + s.ToString());
        Console.WriteLine("Flipped stack: " + flipped.ToString());
        Console.WriteLine("Cloned stack: " + cloned.ToString());
    }
}
```

介绍了 Stack，它有许多静态方法（Clone 和 Flip）和许多实例方法（Push、Pop 和 ToString）。

方法可以被重复调用，这意味着只要有一个唯一的签名，多个方法可能有相同的名称。方法的签名包括方法、数据、修饰符和它的形式参数的各种类型的名称。方法的签名不包括 return 类型。例子

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object o) {
        Console.WriteLine("F(object)");
    }
    static void F(int value) {
        Console.WriteLine("F(int)");
    }
    static void F(int a, int b) {
        Console.WriteLine("F(int, int)");
    }
    static void F(int[] values) {
        Console.WriteLine("F(int[])");
    }
    static void Main() {
        F();
        F(1);
        F((object)1);
        F(1, 2);
        F(new int[] {1, 2, 3});
    }
}
```

介绍了有一个成员方法 F 的类。程序的输出为

```
F()
F(int)
F(object)
F(int, int)
F(int[])
```

#### 1.7.4 属性

属性是提供对对象或类的特性进行访问的成员。属性的例子包括字符串的长度，字体的大小，窗口的焦点，用户的名字，等等。属性是域的自然扩展。两者都是用相关类型成员命名，并且访问域和属性的语法是相同的。然而，与域不同，属性不指示存储位置。作为替代，属性有存取程序，它指定声明的执行来对它们进行读或写。

属性是由属性声明定义的。属性声明的第一部分看起来和域声明相当相似。第二部分包括一个 get 存取程序和一个 set 存取程序。在下面的例子类 Button 定义了一个 Caption 属性。

```
public class Button
{
    private string caption;
    public string Caption {
        get {
            return caption;
        }
    }
}
```

```

        set {
            caption = value;
            Repaint();
        }
    }
}

```

像 Caption 属性一样的读写都可以的属性包括 get 和 set 存取程序。当属性的值要被读出的时候，会调用 get 存取程序；当要写属性值的时候，会调用 set 存取程序。Properties 在 set 存取程序中，属性的新值赋给一个名为 value 的隐含参数。

属性的声明是相对直接了当的，但是属性显式它自己的数值是在使用的时候而不是在声明的时候。可以按照对域进行读写的方法来读写 Caption 属性：

```

Button b = new Button();
b.Caption = "ABC";      // set
string s = b.Caption;   // get
b.Caption += "DEF";     // get & set

```

### 1.7.5 事件

事件是使得对像和类提供通知的成员。一个类通过提供事件声明来定义一个事件，这看起来与域和事件声明相当类似，但是有一个 event 关键字。这个声明的类型必须是 delegate 类型。

在这个例子中

```

public delegate void EventHandler(object sender, Event e);
public class Button
{
    public event EventHandler Click;
    public void Reset() {
        Click = null;
    }
}

```

Button 类定义了一个类型为 EventHandler 的 Click 事件。在 Button 类中，Click 成员与一个 EventHandler 类型的私有域相对应。然而，在 Button 类外，Click 成员只能用在+=和-=操作符的左边。这在添加和删除事件句柄方面限制客户代码。例子

```

using System;
public class Form1
{
    public Form1() {
        // Add Button1_Click as an event handler for Button1's Click event
        Button1.Click += new EventHandler(Button1_Click);
    }

    Button Button1 = new Button();

    void Button1_Click(object sender, Event e) {
        Console.WriteLine("Button1 was clicked!");
    }

    public void Disconnect() {
        Button1.Click -= new EventHandler(Button1_Click);
    }
}

```

介绍了类 Form1，它为 Button1 的 Click 事件添加了 Button1\_Click 作为事件句柄。在 Disconnect 方法中，去掉了事件句柄。

如例子中所示，类 Button 需要被重写来使用像属性一样的事件声明而不是像域一样的事件声明。

```
public class Button
{
    public event EventHandler Click {
        get {...}
        set {...}
    }
    public void Reset() {
        Click = null;
    }
}
```

这个改变不会影响到客户代码，但是因为 Click 的事件句柄不需要用域来实现，所以允许类 Button 的执行更灵活。

### 1.7.6 操作符

操作符是一个定义了可以用来使用在类的实例上的表达式操作符所代表的意义的对象。这里有三种可以定义的操作符：一元操作符，二元操作符和转换操作符。

下面的例子定义了 Digit 类型，它可以描述 0 到 9 间的小数 - 整数值。

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }
    public Digit(int value): this((byte) value) {}
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
    public static Digit operator+(Digit a, Digit b) {
        return new Digit(a.value + b.value);
    }
    public static Digit operator-(Digit a, Digit b) {
        return new Digit(a.value - b.value);
    }
    public static bool operator==(Digit a, Digit b) {
        return a.value == b.value;
    }
    public static bool operator!=(Digit a, Digit b) {
        return a.value != b.value;
    }
    public override bool Equals(object value) {
        return this == (Digit) value;
    }
}
```

```

        public override int GetHashCode() {
            return value.GetHashCode();
        }
        public override string ToString() {
            return value.ToString();
        }
    }
    class Test
    {
        static void Main() {
            Digit a = (Digit) 5;
            Digit b = (Digit) 3;
            Digit plus = a + b;
            Digit minus = a - b;
            bool equals = (a == b);
            Console.WriteLine("{0} + {1} = {2}", a, b, plus);
            Console.WriteLine("{0} - {1} = {2}", a, b, minus);
            Console.WriteLine("{0} == {1} = {2}", a, b, equals);
        }
    }

```

Digit 类型定义了下面的操作符：

- 从 Digit 到 byte 的隐式转换操作符。
- 从 byte 到 Digit 的隐式转换操作符
- 把两个 Digit 数值相加并返回一个 Digit 数值的加法操作符。
- 把一共 Digit 数值与其它 Digit 数值相减，并返回一共 digit 数值的减法操作符。
- 比较两个 digit 数值的等式和非等式。

### 1.7.7 索引

索引(indexer)是使得对象可以像数组一样被索引的成员。然而属性使类似域的访问变得可能，索引使得类似数组的访问变得可能。

作为一个例子，考虑前面给出的类 Stack。这个类会需要执行类似数组的访问，所以可能会不通过执行不需要的 Push 和 Pop 操作而检查或改变堆栈中的项目。Stack 的构造像个列表，但是需要提供方便的数组存取。

索引的声明类似于属性的声明，最大的不同在于索引是无名的（由于 this 是被索引，所以用于声明中的名称是 this）而且索引包含索引参数。索引参数在方括号中提供。例子

```

using System;
public class Stack
{
    private Node GetNode(int index) {
        Node temp = first;
        while (index > 0) {
            temp = temp.Next;
            index--;
        }
        return temp;
    }
}

```

```

        public object this[int index] {
            get {
                if (!ValidIndex(index))
                    throw new Exception("Index out of range.");
                else
                    return GetNode(index).Value;
            }
            set {
                if (!ValidIndex(index))
                    throw new Exception("Index out of range.");
                else
                    GetNode(index).Value = value;
            }
        }
        ...
    }
}

class Test
{
    static void Main() {
        Stack s = new Stack();

        s.Push(1);
        s.Push(2);
        s.Push(3);

        s[0] = 33; // Changes the top item from 3 to 33
        s[1] = 22; // Changes the middle item from 2 to 22
        s[2] = 11; // Changes the bottom item from 1 to 11
    }
}

```

介绍了一个 Stack 中的索引

### 1.7.8 实例构造函数

实例构造函数是实现类中实例进行初始化的行为的成员。

例子

```

using System;

class Point
{
    public double x, y;

    public Point() {
        this.x = 0;
        this.y = 0;
    }

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

    public static double Distance(Point a, Point b) {
        double xdiff = a.x - b.x;
        double ydiff = a.y - b.y;
        return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
    }

    public override string ToString() {
        return string.Format("{0}, {1}", x, y);
    }
}

```

```

class Test
{
    static void Main() {
        Point a = new Point();
        Point b = new Point(3, 4);
        double d = Point.Distance(a, b);
        Console.WriteLine("Distance from {0} to {1} is {2}", a, b, d);
    }
}

```

介绍了一个类 Point，它提供了两个公用的构造函数。一个没有参数的 Point 构造函数和一个有两个 double 参数的构造函数。

如果类中没有提供构造函数，那么会自动提供一个没有参数的构造函数。

### 1.7.9 析构函数

析构函数是实现破坏一个类的实例的行为的成员。析构函数不能有参数，不能任何修饰符而且不能被调用。析构函数在碎片收集时会被自动调用。

例子

```

using System;
class Point
{
    public double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    ~Point() {
        Console.WriteLine("Destructed {0}", this);
    }
    public override string ToString() {
        return string.Format("{0}, {1}", x, y);
    }
}

```

介绍了一个有析构函数的类 Point。

### 1.7.10 静态构造函数

静态构造函数是实现对一个类进行初始化的行为的成员。静态构造函数不能有参数，不能有修饰符而且不能被调用，当类被加载时，类的静态构造函数自动被调用。

例子

```

using System.Data;
class Employee
{
    private static DataSet ds;
    static Employee() {
        ds = new DataSet(...);
    }
    public string Name;
    public decimal Salary;
    ...
}

```

介绍了一个有静态构造函数的类 `Employee`，这个函数对静态域进行初始化。

### 1.7.11 继承

类支持单继承，`object` 类型是所有类的基类。

前面所介绍的例子中的类都是隐含地从 `object` 派生而来的。例子

```
class A
{
    public void F() { Console.WriteLine("A.F"); }
}
```

介绍了从 `object` 派生出来的类 `A`。例子

```
class B: A
{
    public void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        b.F();           // Inherited from A
        b.G();           // Introduced in B

        A a = b;         // Treat a B as an A
        a.F();
    }
}
```

介绍了从类 `A` 中派生出来的类 `B`。类 `B` 继承了类 `A` 的方法 `F`，并且创建了自己的方法 `G`。

方法，属性和索引都可以是虚的，这意味着他们可以在派生的类中被重写。例子

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() {
        base.F();
        Console.WriteLine("B.F");
    }
}

class Test
{
    static void Main() {
        B b = new B();
        b.F();

        A a = b;
        a.F();
    }
}
```

介绍了有虚方法 `F` 的类 `A`，而类 `B` 替换了 `F`。B 中的替换方法包含了一个对 `A` 中被替换的方法的调用 `base.F()`。



可以通过包括 `abstract` 修饰符来说明一个类是不完整的，只是用作其它类的基类。这样的类被称为抽象类。抽象类可以指定抽象函数 - 非抽象派生类必须实现的成员。例子

```
using System;
abstract class A
{
    public abstract F();
}
class B: A
{
    public override F() { Console.WriteLine("B.F"); }
}
class Test
{
    static void Main() {
        B b = new B();
        B.F();
        A a = b;
        a.F();
    }
}
```

介绍了抽象类 A 中的抽象类 F，非抽象类 B 提供了对此方法的实现。

## 1.8 结构

类和结构的相似之处有很多 - 结构可以实现接口，并且可以有同类一样的成员。结构和类在很多重要的方面也不相同，无论如何：结构是数值类型而不是引用类型，而且结构不支持继承。结构的数值或是存储“在堆栈中”或是“在线”。仔细的程序员可以通过明智地使用结构来提高性能。

例如，对 Point 使用结构而不是类，会在成员的存储位置上造成很大不同。下面的程序创建并初始化了一个 100 个单元的数组。如果 Point 用类实现，程序就会占用 101 个分立对象，一个分给数组另外 100 个分给每个元素。

```
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*i);
    }
}
```

如果 Point 用结构来代替，

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

```

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

```

这样测试程序只会占用一个对象，代表的数组。Point 实例在数组内在线分配。这种优化可能会被错误使用。由于当作为数值参数传输一个结构实例时会造成要创建结构，因此用结构替代类也许会使程序变得缓慢和庞大。小心的数据结构和算法设计是无可替代的。

## 1.9 接口

接口定义了一个连接。一个类或这结构必须根据它的连接来实现接口。接口可以把方法、属性、索引和事件作为成员。

例子

```

interface IExample
{
    string this[int index] { get; set; }
    event EventHandler E;
    void F(int value);
    string P { get; set; }
}

public delegate void EventHandler(object sender, Event e);

```

介绍了一个包括一个索引、一个事件 E、一个方法 F 和一个属性 P。

接口可以使用多个继承。在例子中

```

interface IControl
{
    void Paint();
}

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

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

interface IComboBox: ITextBox, IListBox {}

```

接口 IcomboBox 继承了 ItextBox 和 IListbox。

类和结构可以实现多个接口。在例子中

```

interface IDataBound
{
    void Bind(Binder b);
}

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

```

类 `EditBox` 从类 `Control` 中派生并且实现了 `IControl` 和 `IDataBound`。

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

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

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

```
class Test
{
    static void Main() {
        EditBox editbox = new EditBox();
        editbox.Paint(); // error: EditBox does not have a Paint method
        IControl control = editbox;
        control.Paint(); // calls EditBox's implementation of Paint
    }
}
```

## 1.10 代表

因为 C++ 和一些其它语言可以用函数指针来进行访问，所以代表 (delegates) 使得这一特定情况变得可能。与函数指针不同，代表是面向对象的，类型安全并且是可靠的。

代表是引用类型，它从公共基类：`System.Delegate` 派生出来。一个代表实例压缩了一个方法 - 一个可调用的实体。对于静态方法，一个可调用实体由类和类中的静态方法组成。

代表的一个有趣而且有用的特性是它不知道或不关心与它相关的对象的类型。对象所要做的所有事情是方法的签名和代表的签名相匹配。这使得代表很适合“匿名”调用。这是个很有用的功能。

定义和使用代表分为三步：声明、实例化和调用。用 `delegate` 声明语法来声明代表。例子

```
delegate void SimpleDelegate();
```

声明了一个名为 `SimpleDelegate` 的代表，它没有任何参数并且返回 `void` 值。

例子

```
class Test
{
    static void F() {
        System.Console.WriteLine("Test.F");
    }
    static void Main() {
        SimpleDelegate d = new SimpleDelegate(F);
        d();
    }
}
```

创建了一个 `SimpleDelegate` 实例，并且马上就对它进行调用。

对于为一个方法实例化一个代表并且马上调用它没有什么好说的，理由是它会比直接调用方法要简单得多。代表在匿名使用时会显式它的用处。例子

```
void MultiCall(SimpleDelegate d, int count) {
    for (int i = 0; i < count; i++)
        d();
}
```

介绍了一个方法 MultiCall，它重复调用 SimpleDelegate。方法 MultiCall 并不知道或这关心哪种类型方法是 SimpleDelegate 的目标方法，这种方法有什么样的可达性，或这些方法是静态的还是非静态的。所有关心的事情是目标方法的签名是否与一致。

## 1.11 联合

联合类型的声明为一个符号常数相关的组定义了一个类型名称。联合应用于“多选择”的场合，这里运行时间是由在编译时所知道的固定的选择数目所决定。

例子

```
enum Color
{
    Red,
    Blue,
    Green
}

class Shape
{
    public void Fill(Color color) {
        switch(color) {
            case Color.Red:
                ...
                break;
            case Color.Blue:
                ...
                break;
            case Color.Green:
                ...
                break;
            default:
                break;
        }
    }
}
```

介绍了一个联合 color 和一个使用这个联合的方法。方法 Fill 的名称使人很容易明白形状可以用所给的颜色中的一种进行填充。

因为使用联合可以使用的代码更可读还可以自归档，所以使用联合比使用整数常数要好 - 当然，很多语言中通常没有联合。代码的自归档特点也使得开发工具可以帮助编写代码的和进行一些其它的“设计者”行为。例如，使用 Color 而不是 int 作为参数类型，使得精确代码编辑器可以给出 Color 的数值。

## 1.12 名称空间和集合

迄今为止所提供的程序除了依靠一些系统提供的如 System.Console 类以外，都是基于自身的。而对于实际程序来说，由许多不同的片断组成是很普通的。例如，一个完整的程序也许是基于几个不同的成分，包括一些内部开发的一些独立软件商处购买的程序。

**名称空间(Namespace)**和**集合**使得这个基于成分的系统成为可能。名称空间提供了一个逻辑组织系统。名称空间既可以用作一个程序的“内部”组织系统，也可以用作“外部”组织系统。外部组织是一种使用其它程序提供的公开程序元素的方法。

**集合 Assemblies** 是用于物理包装和配置。集合表现得像一个类型的容器。一个集合中可以包含类型、用于实现这些类型的可执行代码和对于其它集合的连接。

这里有两种主要的集合种类：应用程序和库。应用程序有一个主入口，并且通常使用文件扩展名.exe；库没有主入口点，并且通常使用文件扩展名.dll。

为了介绍名称空间和集合的使用，节中我们再看一看前面给出的程序“Hello, world”，并且把它分为两部分：一个提供消息和信息的库和一个独立的显式它们的应用程序。

库中将包含一个简单的类名为 HelloMessage。例子

```
// HelloLibrary.cs
namespace Microsoft.CSharp.Introduction
{
    public class HelloMessage
    {
        public string Message {
            get {
                return "Hello, world";
            }
        }
    }
}
```

用一个名称空间 Microsoft.Csharp.Introduction 来为类 HelloMessage 命名。类 HelloMessage 提供了一个只读属性名为 Message。名称空间可以嵌套，而声明

```
namespace Microsoft.CSharp.Introduction
{...}
```

是对多级名称空间嵌套的简写：

```
namespace Microsoft
{
    namespace CSharp
    {
        namespace Introduction
        {....}
    }
}
```

“Hello,world”程序各部分的下一步是用类 HelloMessage 写一个控制台应用程序。这个类的全称为 Microsoft.CSharp.Introduction.HelloMessage，它要被使用，但是这个名字太长而且很笨拙。一个简单一些的方法是使用名称空间指示，这使得它可以无条件使用所有名称空间中的类型。例子

```
// HelloApp.cs
using Microsoft.CSharp.Introduction;
class HelloApp
{
    static void Main() {
        HelloMessage m = new HelloMessage();
        System.Console.WriteLine(m.Message);
    }
}
```

介绍了指向 `Microsoft.CSharp.Introduction` 名称空间的名称空间指示。HelloMessage 所发生的事情是对 `Microsoft.CSharp.Introduction.HelloMessage` 进行了简写。

C#也使使用别名进行定义成为可能。使用别名指示为一个类型定义别名。这样的别名在两个库中的名称发生冲突或使用相当大的名称空间中的小部分类型时，就显得很有用了。例子

```
using MessageSource = Microsoft.CSharp.Introduction.HelloMessage;
```

介绍了使用别名指示把 MessageSoure 定义为类 HelloMessage 的别名。

我们所写的代码可以被编译为一个包含了类 HelloMessage 的库和一个包含了类 HelloApp 的应用程序。编译的具体过程是根据所使用的编译器会有所不同。用 visual Studio 7.0 提供的命令行编译器，正确的使用为：

```
csc /target:library HelloLibrary.cs
```

它生成类库 HelloLibrary.dll 和

```
csc /reference:HelloLibrary.dll HelloApp.cs
```

它生成应用程序 HelloApp.exe.

### 1.13 版本

*版本 Versioning* 是使得组分随着时间流逝保持一致的方法。如果基于先前版本的代码在编译后可以同新版本一起工作，那么说这一部分的新版本与以前的版本是代码兼容的。与此相同，如果一个基于旧版本的程序没有经过再编译就可以同新版本程序一起工作，那么说这部分的新版本和以前版本是二进制兼容。

大多数语言根本不支持二进制兼容，而还有许多在是代码兼容变得容易方面所做甚少。实际上，通常是一些语言有缺陷，这使得不破坏至少是某些客户代码而使类随时间更新变得不可能。

作为一个例子，想像这样一个情况，一个基类编写者编写了一个名为 Base 的类。在最初的版本，Base 不包括方法 F。从 Base 中派生出一个名为 Derived 的部件，并且引入了 F。这个派生的类和它依赖的类 Base 一起发放给用户，配置到无数客户端和服务上。

```
// Author A
namespace A
{
    class Base // version 1
    {
    }
}

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

迄今为止，很好。但是现在开始出现版本问题了。Base 的作者制作了一个新版本，并且添加了他自己的方法 F。

```
// Author A
namespace A
{
    class Base // version 2
    {
        public virtual void F() { // added in version 2
            System.Console.WriteLine("Base.F");
        }
    }
}
```

新版本的 Base 应该从源代码和二进制都与最初的版本兼容。（如果连简单的添加一个方法都不行，那么基类就不能更新了。）不幸的是，类 Base 中的新 F 使得 Derived 中的 F 变得意义不明。难道 Derived 要覆盖 Base 中的 F？这看起来不像，在 Derived 被编译的时候，Base 甚至还没有一个 F！此外，如果 Derived 的 F 把 Base 中的 F 覆盖掉，那么它必须满足由 Base 确定的协定，一项在 Derived 被编写时不确定的协议？在某些情况下，这是可能的。例如 Base 中也许要被覆盖的 F 的约定经常调用基类。Derived 中的 F 不可能依据这样一个约定。

C# 通过要求开发者明确他们的意图来处理这种版本问题。在前面的代码实例化中，由于 Base 中甚至没有 F，因此代码是清楚的。很显然，由于不存在名为 F 的基本法，因此 Derived 中的 F 被认做是一个新方法而不是对一个基类方法的覆盖。

```
// Author A
namespace A
{
    class Base
    {
    }
}

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

如果 Base 增加了 F，而且变为一个新版本，那么 Derived 的二进制版本的意图仍然清楚，Derived 中的 F 语意不相关，不能被看做是一个替代。

然而，当 Derived 被重新编译，意义就不清楚了。Derived 的作者也许会趋向于用他的 F 覆盖掉 Base 的 F，或者把它隐藏起来。由于意图不清楚，编译器会产生一个警告，而且默认做法是用 Derived 的 F 把 Base 中的 F 隐藏起来。这个行为过程重复了在 Derived 不被再编译的语意。这个警告只是用来告诉 Derived 的作者 Base 中的方法 F 的存在。

如果 Derived 的 F 从语意上与 Base 中的 F 无关，那么 Derived 的作者可以表达这个意图，并且，实际上可以关掉警告，在 F 的声明中使用 new 关键字。

```

// Author A
namespace A
{
    class Base                // version 2
    {
        public virtual void F() { // added in version 2
            System.Console.WriteLine("Base.F");
        }
    }
}

// Author B
namespace B
{
    class Derived: A.Base     // version 2a: new
    {
        new public virtual void F() {
            System.Console.WriteLine("Derived.F");
        }
    }
}

```

另外，Derived 的作者应该考虑将来，并且决定 Derived 中的 F 是否应该替代 Base 中的。这样的意图可以通过使用 override 关键字来表示，如下所示。

```

// Author A
namespace A
{
    class Base                // version 2
    {
        public virtual void F() { // added in version 2
            System.Console.WriteLine("Base.F");
        }
    }
}

// Author B
namespace B
{
    class Derived: A.Base     // version 2b: override
    {
        public override void F() {
            base.F();
            System.Console.WriteLine("Derived.F");
        }
    }
}

```

Derived 的作者还有其它的选择，就是改变 F 的名称，这样就会完全避免名称的冲突。虽然这样的改动会破坏 Derived 的源码和二进制兼容性，但是兼容性的重要性是根据情况变化的。如果 Derived 不提供给其它程序，那么改变 F 的名称是一个不错的主意，它将增强程序的可读性，并且将不会有任何关于 F 的意义的冲突。

## 1.14 特征

C# 是一个程序上的语言，所以同所有程序上的语言相似，它有一些说明的元素。例如，一个类中一个方法的访问能力可以通过修饰符 public、protected、internal、protected internal 和 private 来区分。因为它支持特征，所以程序员可以发明出新的声明信息，为各种各样的程序实体指定声明信息，并且在运行时找回这些声明信息。程序通过使用特征来指定这个附加的声明信息。



例如，一个框架可以定义 HelpAttribute 特征，它可以放在程序元素如类和方法，使得开发者可以从程序元素中提供标记来对它们进行归档。例子

```
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute: System.Attribute
{
    public HelpAttribute(string url) {
        this.url = url;
    }
    public string Topic = null;
    private string url;
    public string Url {
        get { return url; }
    }
}
```

定义了一个名为 HelpAttribute 的特征类（或者简称为 Help），它有一个位置参数（字符串 url）和一个命名的数组（字符串 Topic）。位置参数用特征类中公共构造函数的形式参数定义，而命名参数用特征类的公共读写属性来定义。

例子

```
[Help("http://www.mycompany.com/.../Class1.htm")]
public class Class1
{
    [Help("http://www.mycompany.com/.../Class1.htm", Topic = "F")]
    public void F() {}
}
```

介绍了许多对特征的使用。

对于一个给定程序元素的特征信息可以在运行时通过使用反射支持找回。例子

```
using System;
class Test
{
    static void Main() {
        Type type = typeof(Class1);
        object[] arr = type.GetCustomAttributes(typeof(HelpAttribute));
        if (arr.Length == 0)
            Console.WriteLine("Class1 has no Help attribute.");
        else {
            HelpAttribute ha = (HelpAttribute) arr[0];
            Console.WriteLine("Url = {0}, Topic = {1}", ha.Url, ha.Topic);
        }
    }
}
```

察看 Class1 是否有 Help 特征，并且如果特征存在，则写出相关的 Topic 和 Url 数值。



## 2. 语法结构

### 2.1 翻译的阶段

一个 C# 程序由一个或多个源文件组成。一个源文件是一个统一字符编码的字符的有序序列。源文件通常和文件系统中的文件有一一对应关系，但是这个对应关系并不需要。

从概念来讲，一个程序在编译时有四步：

1. 预处理，一种文本到文本的转换，这使得可以对程序文本进行条件包含和删除。
2. 语法分析，它把输入字符序列转换为一个标记序列。
3. 句法分析，它把标记序列转换为可执行代码。

### 2.2 文法符号

C# 的词汇和句子的文法散布在整个文章中。词汇文法定义如能把字符组合为形式标记；句子的文法定义了如何把标记组合为 C# 程序。

文法生成包括无词尾符号和有词尾符号。在文法生成当中，无词尾符号用意大利体表示，而有词尾符号用定宽字体。每一个无词尾符号定义为一系列产品（production）。这一系列产品的第一行是无词尾符号的名称，接下来是一个冒号。对于一个产品，每个连续的锯齿状的行的右手边同左手边类似是无词尾符号。例子：

```
nonsense:
    terminal1
    terminal2
```

定义了一个名为 *nonsense* 的无词尾符号，有两个产品，一个在右手边是 `terminal1`，一个在左手边是 `terminal2`。

选项通常列为单独的一行，虽然有时有很多选项，短语“one of”会在选项前面。这里有一个对把每个选项都列在单独一行的简单缩写的方法。例子

```
letter: one of
    A  B  C  a  b  c
```

简写为：

```
letter: one of
    A
    B
    C
    a
    b
    c
```

如 *identifier<sub>opt</sub>*，一个写在下方的前缀“*<sub>opt</sub>*”用来作为简写来指明一个可选的符号。例子

```
whole:
    first-part second-partopt last-part
```

是下面的缩写:

```
whole:
    first-part last-part
    first-part second-part last-part
```

## 2.3 预处理

预阶段是一个文本到文本的转换阶段，在预处理过程中，使能进行代码的条件包含和排除。

```
pp-unit:
    pp-groupopt

pp-group:
    pp-group-part
    pp-group pp-group-part

pp-group-part:
    pp-tokensopt new-line
    pp-declaration
    pp-if-section
    pp-control-line
    pp-line-number

pp-tokens:
    pp-token
    pp-tokens pp-token

pp-token:
    identifier
    keyword
    literal
    operator-or-punctuator

new-line:
    The carriage return character (U+000D)
    The line feed character (U+000A)
    The carriage return character followed by a line feed character
    The line separator character (U+2028)
    The paragraph separator character (U+2029)
```

### 2.3.1 预处理声明

在预处理过程中，为了使用名称可以被定义和取消定义。`#define` 定义一个标识符。`#undef` “反定义” 一个标识符，如果一个标识符在以前已经被定义了，那么它就变成了不明确的。如果一个标识符已经被定义了，它的语意就等同于 `true`；如果一个标识符没有意义，那么它的语意等同于 `false`。

```
pp-declaration:
    #define pp-identifier
    #undef pp-identifier
```

例子:

```
#define A
#undef B
class C
{
```

```

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

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

```

变为:

```

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

```

如果有一个 *pp-unit*, 声明就必须用 *pp-token* 元素进行。换句话说, #define 和 #undef 必须在文件中任何“真正代码”前声明, 否则在编译时会发生错误。因此, 也许会像下面的例子一样散布 #if 和 #define:

```

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

```

因为 #define 放在了真实代码后面, 所以下面的例子是非法的:

```

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

```

一个 #undef 也许会“反定义”一个没有定义的名称。下面的例子中定义了一个名字并且对它进行了两次反定义, 第二个 #undef 没有效果, 但还是合法的。

```

#define A
#undef A
#undef A

```

### 2.3.2 #if, #elif, #else, #endif

*pp-if-section* 用来对程序文本的一部件进行有条件地包括和排除。

*pp-if-section*:

*pp-if-group pp-elif-groups<sub>opt</sub> pp-else-group<sub>opt</sub> pp-endif-line*

*pp-if-group*:

*#if pp-expression new-line pp-group<sub>opt</sub>*

```

pp-elif-groups
    pp-elif-group
    pp-elif-groups pp-elif-group

pp-elif-group:
    #elif pp-expression new-line groupopt

pp-else-group:
    #else new-line groupopt

pp-endif-line
    #endif new-line

```

例子:

```

#define Debug
class Class1
{
    #if Debug
        void Trace(string s) {}
    #endif
}

```

变成:

```

class Class1
{
    void Trace(string s) {}
}

```

如果这部分可以嵌套。例子：

```

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

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

```

### 2.3.3 预处理控制行

特性 `#error` 和 `#warning` 使得代码可以把警告和错误的条件报告给编译程序，来查出标准的编译时的警告和错误。

```

pp-control-line:
    #error pp-message
    #warning pp-message

pp-message:
    pp-tokensopt

```

例子

```

#warning Code review needed before check-in

```

```
#define DEBUG
#if DEBUG && RETAIL
    #error A build can't be both debug and retail!
#endif
class Class1
{...}
```

总是产生警告（“Code review needed before check-in”），并且如果预处理修饰符 DEBUG 和 RETAIL 都被定义，还会产生错误。

### 2.3.4 #line

#line 的特点使得开发者可以改变行的数量和编译器输出时使用的源文件名称，例如警告和错误。如果没有行指示符，那么行的数量和文件名称就会自动由编译器定义。#line 指示符通常用于编程后的工具，它从其它文本输入产生 C#源代码。

```
pp-line-number:
    #line integer-literal
    #line integer-literal string-literal

pp-integer-literal:
    decimal-digit
    decimal-digits decimal-digit

pp-string-literal:
    " pp-string-literal-characters "

pp-string-literal-characters:
    pp-string-literal-character
    pp-string-literal-characters pp-string-literal-character

pp-string-literal-character:
    Any character except " (U+0022), and white-space
```

### 2.3.5 预处理标识符

预处理标识符使用和规则 C#标识符文法相似的文法：

```
pp-identifier:
    pp-available-identifier

pp-available-identifier:
    A pp-identifier-or-keyword that is not true or false

pp-identifier-or-keyword:
    identifier-start-character identifier-part-charactersopt
```

true 和 false 符号不是合法的预定义指示符，所以不能用于#define 的定义和#undef 的反定义。

### 2.3.6 预处理表达式

操作符 !, ==, !=, && 和 || 是允许的预定义表达式。在预定义表达式中，圆括号可以用来分组。

```
pp-expression:
    pp-equality-expression
```

*pp-primary-expression:*

```
true
false
pp-identifier
( pp-expression )
```

*pp-unary-expression:*

```
pp-primary-expression
! pp-unary-expression
```

*pp-equality-expression:*

```
pp-equality-expression == pp-logical-and-expression
pp-equality-expression != pp-logical-and-expression
```

*pp-logical-and-expression:*

```
pp-unary-expression
pp-logical-and-expression && pp-unary-expression
```

*pp-logical-or-expression:*

```
pp-logical-and-expression
pp-logical-or-expression || pp-logical-and-expression
```

### 2.3.7 与空白交互作用

条件编译标识符必须在一行的第一个非空白位置。

一个单行注释可以跟在条件编译指示符或 *pp-control-line* 标识符后面。例如：

```
#define Debug // Defined if the build is a debug build
```

对于 *pp-control-line* 标识符，一行的剩余组成 *pp-message*，独立于此行的注释。例子：

```
#warning // TODO: Add a better warning
```

会有一个注释为“// TODO: Add a better warning”的警告。

一个多行注释的起始和结束可以不在同一行中，就像条件编译标识符。例子

```
/* This comment is illegal because it
ends on the same line*/ #define Debug
/* This is comment is illegal because it is on the same line */ #define
Retail
#define A /* This is comment is illegal because it is on the same line */
#define B /* This comment is illegal because it starts
on the same line */
```

结果将是编译时错误。

可以形成一个条件编译标识符的数据符号可能会隐含在注释中。例子

```
// This entire line is a comment. #define Debug
/* This text would be a cc directive but it is commented out:
   #define Retail
*/
```

不包含任何条件编译标识符，然而完全由空白组成。



## 2.4 语法分析

语法分析阶段把输入字符流转换为标记流。

### 2.4.1 输入

```

input:
    input-elementsopt

input-elements:
    input-element
    input-elements input-element

input-element:
    comment
    white-space
    token

```

### 2.4.2 输入字符

```

input-character:
    any Unicode character

```

### 2.4.3 行结束符

```

line-terminator:
    The carriage return character (U+000D)
    The line feed character (U+000A)
    The carriage return character followed by a line feed character
    The line separator character (U+2028)
    The paragraph separator character (U+2029)

```

### 2.4.4 注释

支持两种形式的注释：规则注释和单行注释。

规则注释以/\*开始，并且以\*/结束。规则注释可以占用一行的一部分，单行或多行。例子

```

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

```

包括多个规则注释。

单行注释开始与字符// 并且延伸到行的结束。例子

```
// 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"
        Console.WriteLine("Hello, world");
    }
}
```

介绍了多个单行注释。

```
comment:
    regular-comment
    one-line-comment

regular-comment:
    / * rest-of-regular-comment

rest-of-regular-comment:
    * rest-of-regular-comment-star
    not-star rest-of-regular-comment

rest-of-regular-comment-star:
    /
    * rest-of-regular-comment-star
    not-star-or-slash rest-of-regular-comment

not-star:
    Any input-character except *

not-star-or-slash:
    Any input-character except * and /

one-line-comment:
    / / one-line-comment-text line-terminator

one-line-comment-text:
    input-character
    one-line-comment-text input-character
```

例子:

```
// This is a comment
int i;

/* This is a
   multiline comment */
int j;
```

## 2.4.5 空白

```
white-space:
    new-line
    The tab character (U+0009)
    The vertical tab character (U+000B)
    The form feed character (U+000C)
    The "control-Z" or "substitute" character (U+001A)
    All characters with Unicode class "Zs"
```

## 2.4.6 标记

这里有五种标记：标识符、关键字、数据符号、操作符和标点。因为空白像是标记的分割符，所以被忽略了。

```
token:
    identifier
    keyword
    literal
    operator-or-punctuator
```

## 2.5 句法分析

句法分析阶段把标记流转换为可执行代码。

### 2.5.1 标识符

标识符的规则符合统一字符编码标准 3.0，除了下划线允许使用起首大写字母，格式化字符（类 Cf）不允许用于标识符而统一字符编码标准中的 escape 字符允许用在标识符中。

```
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-charactersopt

identifier-start-character:
    letter-character
    underscore-character

identifier-part-characters:
    identifier-part-character
    identifier-part-characters identifier-part-character

identifier-part-character:
    letter-character
    combining-character
    decimal-digit-character
    underscore-character

letter-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
    A unicode-character-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-character-escape-sequence representing a character of classes Mn or Mc

decimal-digit-character:
    A Unicode character of the class Nd
    A unicode-character-escape-sequence representing a character of the class Nd
```

*underscore-character:*

A Unicode character of the class Pc

A *unicode-character-escape-sequence* representing a character of the class Pc

合法标识符的例子包括“identifier1”, “\_identifier2”, 和 “@if”。

前缀“@”使得可以在标识符中使用关键词。实际上字符@不是标识符的一部分，所以如果没有这个前缀，可能在另外一种语言中被视为通常的标识符。不是关键字也在标识符中使用前缀@是允许的，但是这是一种很不好的风格。

## 例子

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

class Class1
{
    static void M {
        @class.@static(true);
    }
}
```

定义了一个名为“class”的类，有一个静态方法名为“static”，他使用了一个名为“bool”的参数。

**2.5.2 关键字**

关键字是类似于标识符的保留字符序列，除非用@字符开头，否则不能用作标识符。

*keyword:* one of

abstract	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	static	string	struct	switch
this	throw	true	try	typeof
uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while	

**2.5.3 数据符号**

数据符号是数值的源代码表示。

*literal:*

*boolean-literal*  
*integer-literal*  
*real-literal*  
*character-literal*  
*string-literal*  
*null-literal*

### 2.5.3.1 二进制数据符号

这里有两种二进制数值：true 和 false。

*boolean-literal:*

true  
false

### 2.5.3.2 整数数据符号

整数数据符号有两种可能的形式：十进制和十六进制。

*integer-literal:*

*decimal-integer-literal*  
*hexadecimal-integer-literal*

*decimal-integer-literal:*

*decimal-digits* *integer-type-suffix*<sub>opt</sub>

*decimal-digits:*

*decimal-digit*  
*decimal-digits* *decimal-digit*

*decimal-digit:* one of

0 1 2 3 4 5 6 7 8 9

*integer-type-suffix:* one of

U u L l UL Ul uL ul LU Lu lU lu

*hexadecimal-integer-literal:*

0x *hex-digits* *integer-type-suffix*<sub>opt</sub>

*hex-digits:*

*hex-digit*  
*hex-digits* *hex-digit*

*hex-digit:* one of

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

整数数据符号的类型按下面确定：

- 如果数据符号没有后缀，它就有这些类型中的第一个，这些类型可以表示出它的数值：int、uint、long、ulong。
- 如果数据符号有后缀 U 或 u，它就有这些类型中的第一个，这些类型可以表示出它的数值：uint、ulong。
- 如果数据符号有后缀 L 或 l，If the literal is suffixed by L or l, 它就有这些类型中的第一个，这些类型可以表示出它的数值：long、ulong。

- 如果数据符号有后缀 `UL`、`Ul`、`uL`、`uI`、`LU`、`Lu`、`lU` 或 `lu`，它的类型就是 `ulong`。

如果用整数数据符号表示的数值超出了 `ulong` 类型，就会产生错误。

为了允许最小的可能的 `int` 和 `long` 数值可以用十进制整数描述，存在下面两个规则：

- 当一个十进制整数数据符号的数值为  $2^{31}$ ，并且没有整数类型后缀，而操作数有一元负操作符 (§ 错误！未找到引用源。) 时，结果是 `int` 类型的常数，数值为  $-2^{31}$ 。在所有其它情况下，这样的十进制整数数据符号是 `uint` 类型。
- 当一个十进制整数数据符号的数值为  $2^{63}$ ，并且没有整数类型后缀或整数类型后缀 `L` 或 `l`，而操作数有一元负操作符 (§ 错误！未找到引用源。) 时，结果是 `long` 类型的常数，数值为  $-2^{63}$ 。在所有其它情况下，这样的十进制整数数据符号是 `ulong` 类型。

### 2.5.3.3 实数据符号

*real-literal:*

```
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt
. decimal-digits exponent-partopt real-type-suffixopt
decimal-digits exponent-part real-type-suffixopt
decimal-digits real-type-suffix
```

*exponent-part:*

```
e signopt decimal-digits
E signopt decimal-digits
```

*sign:* one of

```
+ -
```

*real-type-suffix:* one of

```
F f D d M m
```

如果没有指定 `real` 类型后缀，实数据符号的类型是 `double`。否则，实类型后缀决定了实数据符号，如下：

- 一个实数据以 `F` 或 `f` 为后缀是 `float` 类型。例如数据符号 `1f`、`1.5f`、`1e10f`、和 `-123.456F` 都是 `float` 类型数据。
- 一个实数据以 `D` 或 `d` 为后缀是 `double` 类型。例如数据符号 `1d`、`1.5d`、`1e10d`、和 `-123.456d` 都是 `double` 类型数据。
- 一个实数据以 `M` 或 `m` 为后缀是 `decimal` 类型。例如数据符号 `1m`、`1.5m`、`1e10m`、和 `-123.456m` 都是 `decimal` 类型数据。

如果所指定的数据符号不能用指定类型表示，在编译时会产生错误。

### 2.5.3.4 字符数据符号

字符数据符号是一个用号括起来的单个字符，如 `'a'`。

*character-literal:*

```
' character '
```

*character:*  
    *single-character*  
    *simple-escape-sequence*  
    *hexadecimal-escape-sequence*  
    *unicode-character-escape-sequence*

*single-character:*  
    Any character except ' (U+0027), \ (U+005C), and *new-line*

*simple-escape-sequence:* one of  
    \' \' \' \' \0 \a \b \f \n \r \t \v

*hexadecimal-escape-sequence:*  
    \x *hex-digit* *hex-digit*<sub>opt</sub> *hex-digit*<sub>opt</sub> *hex-digit*<sub>opt</sub>

在一个单转意符序列或一个十六进制转意符序列中，一个跟在反斜杠字符(\)后面的字符必然是下面的字符之一：'、"、\、0、a、b、f、n、r、t、x、v。否则，在编译是会发生错误。

一个简单的转意符序列表示了统一的字符编码的字符编码，如下表所示。

转意序列	字符名称	Unicode 编码
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

2.5.3.5 字符串数据符号

C# 支持两种形式的字符串数据符号：规则字符串数据符号和逐字的字符串数据符号。规则字符串数字符号由用双引号括起 0 或更多字符组成，例如"Hello, world",并且也许会包括简单转意序列（例如\t表示 tab 字符）和十六进制转意序列。

逐字的字符串数据符号由一个@字符后面跟着双引号括起的 0 或者更多字符组成。一个简单的例子是@"Hello, world"。在一个逐字字符串数据符号中，分割符间的字符通常认为是逐字的，只有引用转意序列例外。特别的是，简单转意序列和十六进制转意序列在逐字字符串数据符号中不支持。一个逐字字符串数据符号可能会跨越很多行。

*string-literal:*  
    *regular-string-literal*  
    *verbatim-string-literal*

*regular-string-literal*:  
 " *regular-string-literal-characters*<sub>opt</sub> "

*regular-string-literal-characters*:  
*regular-string-literal-character*  
*regular-string-literal-characters* *regular-string-literal-character*

*regular-string-literal-character*:  
*single-regular-string-literal-character*  
*simple-escape-sequence*  
*hexadecimal-escape-sequence*  
*unicode-character-escape-sequence*

*single-regular-string-literal-character*:  
 Any character except " (U+0022), \ (U+005C), and *new-line*

*verbatim-string-literal*:  
 @" *verbatim-string-literal-characters*<sub>opt</sub> "

*verbatim-string-literal-characters*:  
*verbatim-string-literal-character*  
*verbatim-string-literal-characters* *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*:  
 ""

### 例子

```
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"
string f = @"Joe said ""Hello"" to me"; // Joe said "Hello"
string g = "\\server\\share\\file.txt"; // \\server\\share\\file.txt
string h = @"\\server\\share\\file.txt"; // \\server\\share\\file.txt
string i = "one\ntwo\nthree";
string j = @"one
two
three";
```

介绍了多种字符串数据符号。最后一个字符串数据 j 是逐字字符串数据，它横跨了很多行。在引号间的字符，包括空白如转行字符，都是逐字复制的。

### 2.5.3.6 null 数据字符

*null-literal*:  
 null



## 2.5.4 操作符和标点

这里有许多种操作符和标点。操作符用于表达式来描述操作涉及到一个或多个操作数。例如，表达式 `a + b` 使用 `+` 操作符来把 `a` 和 `b` 相加。标点用于组织和分割。例如标点 `;` 是用来分割在声明列表中出现的声明。

*operator-or-punctuator: one of*

<code>{</code>	<code>}</code>	<code>[</code>	<code>]</code>	<code>(</code>	<code>)</code>	<code>.</code>	<code>,</code>	<code>:</code>	<code>;</code>
<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>&amp;</code>	<code> </code>	<code>^</code>	<code>!</code>	<code>~</code>
<code>=</code>	<code>&lt;</code>	<code>&gt;</code>	<code>?</code>	<code>++</code>	<code>--</code>	<code>&amp;&amp;</code>	<code>  </code>	<code>&lt;&lt;</code>	<code>&gt;&gt;</code>
<code>==</code>	<code>!=</code>	<code>&lt;=</code>	<code>&gt;=</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>&amp;=</code>
<code> =</code>	<code>^=</code>	<code>&lt;&lt;=</code>	<code>&gt;&gt;=</code>	<code>-&gt;</code>					

## 2.5.5 Unicode 字符转意字符序列

一个 Unicode 字符转意字符序列代表了一个 Unicode 字符。Unicode 字符转意字符序列在标识符，字符串数据符号和字符数据符号中是被允许的。

*unicode-character-escape-sequence:*

`\u hex-digit hex-digit hex-digit hex-digit`

不能实现多重转换。例如字符串数据 `"\u005Cu005c"` 与 `"\u005c"` rather than `"\\"` 是相等的。(Unicode 数值 `\u005c` 是字符 `"\"`。)

例子

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

介绍了许多 `\u0066` 的使用，它是字母 `"f"` 的字符转意序列。这个程序等价于

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



## 3. 基本概念

### 3.1 声明

C#程序中的声明定义了程序的重要声明。C#程序用名称空间(§错误！未找到引用源。)来组织，它可以包含类型声明和嵌套声明。类型声明(§9.5)用来定义类(§10)、结构(§11)、接口(§11)、联合(§14)和代表(§15)。在类型定义中被允许的的成员种类要根据类型声明的形式决定。例如，类的声明可以包含实例构造函数(§10.10)、析构函数(§10.11)、静态构造函数(§10.12)、常数(§10.3)、域(§10.4)、方法(§错误！未找到引用源。)、属性(§10.6)、事件(§10.7)、索引(§10.8)、操作符和嵌套类型。

一个声明在声明所属的声明域定义了一个名称。除了重载构造函数、方法、索引和操作符名称，在一个声明域中有两个或更多介绍有相同名称成员的声明时，是错误的。对一个声明域中，包含有相同名称的不同种类成员是永远不可能的。例如，一个声明域中不能包括有相同名称的域和方法。

这里有许多种不同类型的声明域，如下所示。

- 在所有程序的源文件中，不包括嵌套名称空间声明的名称空间成员声明都是一个单独的组合声明域，称为全局声明域。
- 在所有程序的源文件中，名称空间成员声明和有相同完整正确的名称空间名称的名称空间声明都是一个单独的组合声明域。
- 每个类，结构或接口声明都会创建一个新的声明域。名称通过类成员声明、结构成员什么或接口成员声明介绍到这个声明域中。除了重载构造函数声明和静态构造函数声明，一个类或结构成员声明不能引入与类或结构名称相同的成员。一个类，结构或接口允许方法和索引的重载。此外，一个类或结构允许构造函数和操作符的重载声明。例如，一个类、结构或接口可能会包含多个用相同名称声明的方法，而在他们各自签名(§3.4)中提供了不同的方法声明。注意基类不影响类的声明域。而基本接口不影响一个接口的声明域。这样，一个派生的类或接口可以用和继承的成员相同的名称声明一个成员。这样的成员被称为隐藏了继承的成员。
- 每个枚举声明创建一个新的声明域。名称通过枚举成员声明介绍到声明域中。
- 每个块或者转换块 为局部变量创建一个分立声明域。名称通过局部变量声明被引入到这个声明域。如果一个块是构造函数或方法声明的主体，在形参列表中声明的参数是这个块的局部变量声明域的成员。块的局部变量声明域包括任何嵌套块。因而，在一个嵌套块中不太可能用与嵌套块中的局部变量有相同名称声明一个局部变量。
- 每个块或转换块为标签创建一个分立的声明域。名称通过标签声明被引入到这个声明域，而名称通过 goto 声明引入到这个声明域。块的局部变量声明域包括任何嵌套块。因而，在一个嵌套块中不太可能用与嵌套块中的标签相同名称声明一个标签。

在名称被声明的文本顺序通常并不重要。特别是，文本顺序对于声明和使用名称空间、类型、常数、方法、属性、事件、索引、操作符、构造函数、析构函数和静态构造函数来说并不重要。声明顺序在下面的途径中才是重要的：

- 域声明和局部变量的声明顺序决定了他们的初始化是按什么顺序执行。
- 局部变量必须在他们被使用前定义(§3.5)。

- 当常数表达式数值被忽略，枚举成员声明的声明顺序是重要的 (§14.2)。

名称空间的声明域是“开放的”，而两个有相同的完全名称的名称空间声明将放到相同的名称空间空间中。例如

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

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

上面声明的两个名称空间声明捐献给相同的声明域，这里声明了两个有完全合格名称的类 Megacorp.Data.Customer 和 Megacorp.Data.Order。因为两个声明属于相同的声明域，如果每个都包含一个对相同名称的类的声明，就会产生错误。

块的声明域包括任何嵌套块。这样，在下面的例子中，方法 F 和 G 有错误，因为名称 i 已经在 outer 块中声明了，就不能再在 inner 块中声明。然而，由于两个 i 是在分立的非嵌套块中声明的，方法 H 和 I 是有效的。

```
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();
    }
}
```

## 3.2 成员

名称空间和类型都有成员。一个实体的成员通常通过使用一个有效的名称来得到，此名称从一个对实体的引用开始，跟着一个“.”代号，然后是成员的名称。

类型的成员或者在类型中声明或者从类型的基类继承。当从基类继承一个类型，基类中除了构造函数和析构函数的所有成员都称为派生类型的成员。一个基类成员声明可访问性并不控制是否成员被继承，除了构造函数和析构函数的其它任何成员都可以被继承。然而，一个被继承的成员也许不能在派生的类型中进行访问，或者因为它的声明可访问性([§错误！未找到引用源。](#))或者是因为他被类型中自己的声明隐藏了([§3.5.1.2](#))。

### 3.2.1 名称空间成员

没有嵌套名称空间的名称空间和类型是全局名称空间的成员。这直接与在全局声明域里声明的名称相符合。

在某个名称空间中定义的名称空间和类型是那个名称空间的成员。这直接与在声明域里声明的名称相符合。

名称空间没有访问限制。不能声明私有、保护或内部的名称空间，并且名称空间名称通常是公共可访问的。

### 3.2.2 结构成员

一个结构的成员是在这个结构中声明的而且从类 `object` 中继承的成员。

与结构类型成员直接相关的简单类型的成员由简单类型给出别名：

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

### 3.2.3 枚举成员

一个枚举的成员是在枚举中声明的常数，而且这些成员从类 `object` 中继承。

### 3.2.4 类成员

一个类的成员是在此类中声明的成员，并且是从基类中继承的成员（除了类 `object` 没有基类）。从基类中继承的成员包括基类中的常数、域、方法、属性、事件、索引、操作符和类型，但是不包括基类的构造函数、析构函数和静态构造函数。基类的成员不管他们的访问能力就可以继承。

一个类声明可能包括对常数、域、方法、属性、事件、索引、操作符、构造函数、析构函数、静态构造函数和类型的声明。

`Object` 和 `string` 的成员直接与由他们定别名的类类型成员相对应：

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

### 3.2.5 接口成员

一个接口的成员是在此接口中和所有这个接口的基础接口中声明的成员，并且这个接口从类 `object` 中继承。

### 3.2.6 数组成员

数组的成员是从类 `System.Array` 继承的成员。

### 3.2.7 代表成员

代表的成员是从类 `System.Delegate` 继承的成员。

## 3.3 成员访问

成员的声明允许通过成员访问来控制。成员的可访问性是由一些成员已经声明的访问性建立的，如果有立即包含（`containing`）类型，这些成员就同这些立即包含类型相结合。

当访问一个特殊成员被允许时，成员被称为可访问的。相反，当对一个成员的访问被禁止，这个成员就被称为不可访问的。当成员的可访问域 (§3.3.2) 中包括了访问发生的文本地址时，就允许对一个成员进行访问。

### 3.3.1 声明可访问性

一个成员的声明可访问性可以是下面几个之一：

- 公共的，它通过在成员声明中加 `public` 修饰符来选择。公共的直觉意义是“无限制访问”。
- 保护的内部的（意思是保护或内部）是通过在成员声明中包括一个 `protected` 和一个 `internal` 修饰符来选择。保护的内部的直觉意义是“对这个程序或从包含类中继承的类型的访问受限制”。
- 保护的，它通过在成员声明中包括 `protected` 修饰符来选择。保护的直觉意义是“对从包含类或从包含类中继承的类型的访问受限制。”
- 内部的，它通过在成员声明中包括 `internal` 修饰符来选择。内部的直觉意义是“对这个程序的访问受限制”。
- 私有的，它通过在成员声明中包括 `private` 修饰符来选择。私有的直觉意义是“对于包含类型的访问受限制”。

根据成员声明发生地方的上下文，只有特定的声明可访问性被允许。而且，当一个成员声明不包括任何存储修饰符时，声明发生地方的上下文决定默认的声明可访问性。

- 名称空间隐含有一个 public 声明可访问性。在名称空间声明中不需要访问修饰符。
- 在编译单元或名称空间中的类型声明可以用 public 或 internal 声明可访问性，而默认的是内部声明可访问性。
- 类成员可以是五种声明可访问性中的任意一个，默认的是私有声明可访问性。（注意一个声明为一个类的成员的类型可以有五个声明可访问性的任何一个，然而一个声明为名称空间成员的类型只能是公共或内部声明可访问性。
- 结构成员可以是公共、内部或私有声明可访问性，默认的是私有声明可访问性。结构成员不能有保护或者内部保护声明可访问性。
- 接口成员隐含有一个公共声明可访问性。在接口成员声明中不允许访问修饰符。
- 枚举成员隐含有一个公共声明可访问性。在枚举成员声明中不允许访问修饰符。

### 3.3.2 可访问性域

一个成员的可访问性域是（也许是脱节的）程序文字的一部分，在这里，允许对成员进行访问。为了定义一个成员的可访问性域，如果不在类型里声明，一个成员就被说成是顶级的，而如果它在另外一个类型里声明，这个成员就被称为嵌套的。而且，程序的程序文字就像所有包括在程序源文件中的程序文字一样定义，而一个类型的程序文字就像所有在类、结构接口或枚举的结构体中包含在开始和结束符号“{”和“}”中的程序文字一样定义（可能包括有嵌套的类型）。

一个预定义类型（例如 object、int 或 double）的可访问性域是没有限制的。

一个在程序 P 中声明的顶级类型 T 的可访问性域定义如下：

- 如果 T 的声明可访问性是公共的，那么 T 的可访问性域是 P 的程序文字和任何引用 P 的程序。
- 如果 T 的声明可访问性是内部的，T 的声明可访问性就是 P 的程序文字。

从这些它所跟随的定义来看，顶级类型的可访问性域通常至少是声明类型的程序的程序文字。

程序 P 中在类型 T 中声明的成员 M 的可访问性域的定义如下（注意 M 本身也许就是个类型）：

- 如果 M 的声明可访问性是公共的，M 的可访问性域是 T 的可访问性域。
- 如果 M 的声明可访问性是内部保护的，M 的可访问性域就是 T 的可访问性域和 P 的程序文字的交集和在 P 外面声明的并且从 T 继承的程序文字。
- 如果 M 的声明可访问域是保护的，M 的可访问性域就是 T 的可访问性域和 T 的程序文字的交集和任何从 T 中继承的类型。
- 如果 M 的声明是内部的，M 的可访问性域就是 T 的可访问性域和 P 的程序文字的交集。
- 如果 M 的声明是私有的，M 的可访问性域就是 T 的程序文字。

从这些它所跟随的定义来看，嵌套成员的可访问性域通常至少是声明成员的类型程序文字。而且，一个成员的可访问性域永远不会比成员被声明的类型的可访问性域包含更多。

从直觉来说，当一个类型或成员 M 被访问，下面的步骤就是进行估计以确保访问被允许：

- 首先，如果M按一个类型声明（与一个编译单元或名称空间相对），如果那个类型是不可访问的，就会发生错误。
- 这样，如果M是公共的，访问就被允许。
- 另外，如果M是内部保护的，如果访问发生在M被声明的程序中访问就是被允许的，或者如果访问发生在从M被声明的类继承的类中并且是通过派生类类型发生(\$**错误！未找到引用源。**)，访问也是被允许的。
- 另外，如果M是保护的，如果访问发生在M被声明的类中，或者在从M被声明的类继承的类中并且是通过派生类类型发生(\$**错误！未找到引用源。**)，访问就是被允许的。
- 另外，如果M是内部的，如果发生在M被声明的程序中，访问就是被允许的。
- 另外，如果M是私有的，如果访问发生在M被声明的类型中，访问就是被允许的。
- 另外，如果成员的类型是不可访问的，就会发生错误。

在例子中

```
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;
    }
}
```

类和成员有下面的可访问性域：

- A 和 A.X 的可访问性域是没有限制的。
- A.Y、B、B.X、B.Y、B.C、B.C.X 和 B.C.Y 的可访问性域是包含程序的程序文字。
- A.Z 的可访问性域是 A 的程序文字。
- B.Z 和 B.D 的可访问性域是 B 的程序文字，包括 B.C 和 B.D 的程序文字。
- B.C.Z 的可访问性域是 B.C 的程序文字。
- B.D.X、B.D.Y 和 B.D.Z 的可访问性域是 B.D 的程序文字。



如所示的例子，一个成员的可访问性域永远不会比比包含类型的大。例如甚至所有 X 成员有公共的声明可访问性，除了 A.X 外都有被包含类型约束的可访问性域。

如§3.2中所描述，基类中除了构造函数和析构函数的所有成员都是从派生类型继承的。这甚至包括基类的私有成员。然而，一个私有成员的可访问性域只包括声明成员的类型程序文字。在例子中

```
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
    }
}
```

类 B 从类 A 中继承私有成员 X。因为成员是私有的，所以只有在 A 的类结构体中才能对它进行访问。这样在方法 A.F 中允许对 b.x 的访问，但是在方法 B.F 中是失败的。

### 3.3.3 保护的访问

当一个保护成员在他被声明的类的程序文字外被访问，并且当一个内部保护成员在他被声明的程序的程序文字外被访问，访问就要求通过访问发生的派生类中进行。让声明了一个保护成员 M 的 B 作为一个基类，并且让 D 作为从 B 派生的类。在 D 的类结构体内，可以通过下面的某种形式来访问 M：

- 格式 M 的一个无效的类型名称和一个基本的表达式。
- 一个格式 T.M 的基本表达式，T 是由 D 或者从 D 派生的类提供的。
- 一个格式 E.M 的基本表达式，E 是由 D 或者从 D 派生的类提供的。
- 一个格式 base.M 的基本表达式。

除了这些形式的访问，一个派生类可以在构造函数初始化时访问基类的保护的构造函数 (§10.10.1)。

在这个例子中

```
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
    }
}
```

由于访问或者通过 A 的实例发生或者在 A 的派生类中发生，因此，在 A 中可以通过 A 和 B 的实例来访问 X。然而，由于 A 不是从 B 中派生的，因此在 B 中不可能通过 A 的实例访问 x。

### 3.3.4 可访问性约束

C#语言中的许多结构需要一种至少和一个成员或其它类型相同可访问的类型。如果 T 是 M 可访问性域的一个超集，那么类型 T 就要求至少可和成员或类型 M 一样可访问。换句话说，如果 T 在所有 M 可访问的上下文中都可访问，那么 T 至少和 M 一样可访问。

下面的有一些可访问性的约束：

- 一个类类型的直接基类必须至少同类类型本身同样可访问。
- 一个接口类型的外部基本接口必须至少同接口类型本身同样可访问。
- 代表类型的返回类型和参数类型必须至少同代表类型本身同样可访问。
- 常数的类型必须至少同常数本身同样可访问。
- 域的类型必须至少同域本身同样可访问。
- 一个方法的返回类型和参数类型必须至少同方法本身同样可访问。
- 属性的类型必须至少同属性本身同样可访问。
- 事件的类型必须至少同事件本身同样可访问。
- 参数的类型必须至少同索引本身同样可访问。
- 一个操作符的返回类型和参数类型必须至少同操作符本身同样可访问。
- 构造函数的参数类型必须至少同构造函数本身同样可访问。

在这个例子中

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

因为 A 不能与 B 一样可访问，所以类比是有错误的。

同样，在例子中

```
class A {...}
public class B
{
    A F() {...}
    internal A G() {...}
    public A H() {...}
}
```

因为 A 不能与 B 一样可访问，所以 B 中的方法 H 也是有错误的。

### 3.4 签名和重载

方法、构造函数、标签和操作符按照它们的签名分类：

- 一个的签名由方法和对象的名称、修饰符和它的形参的类型组成。方法的签名中不包括返回类型。
- 一个函数的签名由数字修饰符和它的形参的类型组成。
- 一个标签的签名是由数字和它的形参的类型组成。标签的签名不包括元素类型。
- 操作符的签名由操作符的名称和数字还有它的形参的类型组成。操作符的签名不包括结果类型。

签名激活了类、结构和接口中成员的重载机制：

- 方法的重载允许类、结构或接口用相同的名称声明多个方法，并且所提供的方法的签名都是唯一的。
- 构造函数的重载允许一个类或结构声明多个构造函数，所提供的构造函数的签名都是唯一的。
- 标签的重载允许类、结构或接口声明多个标签，所提供的标签的签名都是唯一的。
- 操作符的重载允许一个类或结构用相同的名称声明多个操作符，所提供的操作符的签名都是唯一的。

下面的例子介绍了一系列方法声明重载和它们的签名。

```
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)
    void F(int x, int y); // F(int, int)
    int F(string s);    // F(string)
    int F(int x);       // F(int)
}
```

注意参数修饰符是签名的一部分。这样 `F(int)`、`F(ref int)` 和 `F(out int)` 都是唯一的签名。此外注意第二个和最后一个方法的声明的返回类型，它们的签名都是 `F(int)`。这样，在编译上面的例子时会在第二个和最后一个例子产生错误。

### 3.5 范围

一个名字的范围就是程序文本的区域，在这个范围内可以通过声明的名称来查找实体，名称没用条件限制。范围可以嵌套，并且内层的范围可以对从外层范围来的名称的意义进行再次声明。从外层范围来的名称称为在内层范围覆盖的程序文本区域内被隐藏了，并且对外层名称的访问只能通过限定名称。

- 在被不包括嵌套名称空间声明的名称空间成员声明中表示的名称空间成员的范围对于每个编译单元的程序文本来说是完整的。
- 在一个全称为 `N` 的名称空间声明中由名称空间成员声明表示的名称空间成员的范围是任何一个全称为 `N` 或由与 `N` 有相同的修饰符序列的名称空间声明的名称空间结构体。
- 通过使用指示符定义或引入名称的范围扩展到使用指示符发生的编译单元或名称空间结构体的名称空间成员声明中。一个使用指示符可能会使得在某个编译单元或名称空间结构体中得到从 0 到多个名称空间或类型名称，但是并不把任何新成员放到主要声明域中。换句话说，使用指示符并没有传递，而只是影响它出现的编译单元或名称空间结构体。
- 在类成员声明中表示的成员的表示所在的类结构体。另外，一个类成员的范围扩展到派生类的类结构体，这些派生类包含在成员的可访问性域 (§3.3.2) 中。
- 在结构成员声明中表示的成员的表示所在的范围表示所在的结构体。
- 在枚举成员声明中表示的成员的表示所在的枚举结构体。
- 在构造函数声明中表示的参数的范围，是这个构造函数声明的构造函数初始化程序和块。

- 在方法声明中表示的参数的范围是这个方法声明的方法结构体。
- 在标签声明中表示的参数的范围是这个标签声明的访问声明。
- 在操作符声明中表示的参数的范围是这个操作符声明的块。
- 在局部变量声明中表示的局部变量的范围是声明出现的块。从执行局部变量的变量声明的文本位置引用局部变量是错误的。
- 在 for 声明的 for 初始化程序中表示的局部变量的范围是 for 声明的 for 初始化、for 条件、for ( 循环指示符 ) 和所包含的声明。
- 在标签声明中表示的标签的范围是表示所在的块。

在名称空间、类、结构或枚举成员的范围中，可以把成员引用到执行对对象的声明的文字位置。例如

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

这里，对于 F 来说在没有被表示前把 I 引入是有效的。

在一个局部变量的范围中，在执行局部变量变量说明符的文本位置引入局部变量是错误的。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);      // Legal
    }
    void H() {
        int a = 1, b = ++a;   // Legal
    }
}
```

在上面的方法 F 中，第一次对 I 赋值很明确没有引用在外层范围声明的域。它指向局部变量而且它是错误的，因为它从文字上在变量的声明之前。在方法 G 里，在初始化程序中使用 j 来声明是合法的，因为使用并没有在变量声明之前。在方法 H 中，一个后来的变量声明合法的引用了一个局部变量，这个局部变量在早先的变量声明时声明过了。

对于局部变量的确定范围的规则是要保证用于表达式中的名称的意义与块中的相同。如果局部变量的范围只是从他的声明延伸到块结束的地方，那么就像上面的例子中所示，第一个赋值将赋给实例变量，而第二个赋值将赋给局部变量，如果块的声明在以后被重新分配，就有可能导致错误。

在块内的名称的意义可能会根据使用名称的上下文有所不同。在例子中

```

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.ToString()); // writes "Type: A"
    }
}

```

名称 A 用在一个表达式中来引用局部变量 A，而在一个类型中引用类 A。

### 3.5.1 名称隐藏

典型情况下，一个实体的范围比实体本身的声明域要涉及更多的程序文字。特别是，一个实体的范围可能会包括声明，这些声明会引入一些包含有相同名称的实体的新声明域。这样的声明会使原来的实体变为隐藏。相反，当一个实体不是隐藏的时候，称为可视。

当嵌套引起范围重叠时和当继承引起范围重叠时，就会出现名称隐藏。两种隐藏的特征在下面的章节中描述。

#### 3.5.1.1 通过嵌套隐藏

出现在当名称空间中的名称空间和类型的嵌套，或者类和结构中的类型嵌套，或者是参数和局部变量声明的都会造成通过嵌套隐藏名称的出现。通过嵌套隐藏名称通常是“安静地”发生，也就是当外部名称被内部名称隐藏起来时，不会报告错误和警告。

在这个例子中

```

class A
{
    int i = 0;
    void F() {
        int i = 1;
    }
    void G() {
        i = 1;
    }
}

```

在方法 F 中，实例变量 I 被局部变量 I 隐藏了，但是在方法 G 中，I 还是引用实例变量。

当一个内部范围的名称隐藏一个外部范围的名称，它会把那个名称的所有重载都隐藏。在例子中

```

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) {}
    }
}

```

对 F(1)的调用实际调用了在内层声明的 F，因为所有 F 的外部事件都被内部声明隐藏了。出于相同的原因，调用 F("Hello")是错误的。

### 3.5.1.2 通过继承隐藏

名称通过继承隐藏在类或结构重新对从基类继承来的名称声明时发生。这种类型的名称隐藏有下面形式中的一种：

- 一个引入类或结构的常数、域、属性、事件或类型隐藏了所有基类中名称相同的成员。
- 一个引入类或结构的方法隐藏了所有有相同名称的非方法基类成员和所有有相同签名的基类成员（方法名称和参数数目、修饰符和类型）。
- 一个引入类或结构的索引隐藏所有有相同签名的基类索引（参数数目和类型）。

管理操作符声明 (§10.9) 的规则使得一个派生类可以用与基类中的一个操作符相同的签名声明一个操作符。这样，操作符就不会隐藏其它的操作符。

与隐藏外部范围的名​​称相反，隐藏继承范围的可访问名称就会报告警告。在例子中

```

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

```

在 Derived 中 F 的声明产生了一个警告报告。隐藏一个继承的名称肯定不是错误，因为这样将排除基类单独更新。例如，因为 Base 后来的版本把不在先前版本类中的 F 方法引入，这样就会产生上面的情况。如果上面的情况是一个错误，那么任何对于在独立版本类库中的基类的变更就会潜在地使得派生类变为无效。

由于继承名称引起的警告可以通过使用新修饰符来消除：

```

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

```

新的修饰符指出 Derived 中的 F 是“新”，并且它实际上是要隐藏继承成员。

新成员的声明只是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
}

```

在上面的例子中，Derived 中 F 的声明隐藏了从 Base 继承的 F，但是由于 Derived 中的新 F 有私有的可访问性，所以它的范围不会扩展到 MoreDerived。这样，MoreDerived.G 中对 F() 的调用是有效的，并且会引用 Base.F。

### 3.6 名称空间或类型名称

C# 程序中的许多上下文都需要名称空间名称和类型名称来确认。名称的任何形式都是写作一个或多个由“.”符号分开的标识符。

```

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier
    namespace-or-type-name . identifier

```

一个类型名称是一个指向某个类型的名称空间或类型名称。后面的分析如下描述，类型名称的名称空间或类型名称必须指向一个类型，如果是其它就会产生错误。

一个名称空间名称是一个指向名称空间的名称空间名称或类型。后面的分析如下描述，名称空间名称的名称空间或类型名称必须指向一个类型，如果是其它就会产生错误。

名称空间或类型名称的意义如下所示：

- 如果名称空间或类型名称由单独的指示符组成：
  - 如果名称空间或类型名称出现在类或结构的声明之内，那么从类或结构声明开始并且在嵌套类或结构声明中延续（如果有），如果有所给名称的成员存在，是可访问的并且指示了一种类型，那么名称空间或类型名称指向那个成员。注意无类型成员（构造函数、常数、域、方法、属性、索引和操作符）在确定名称空间或类型名称时是被忽略的。
  - 否则，如果有名称空间或类型名称出现（如果有）的声明，同每个嵌套名称空间声明（如果有）一起，并且由全局名称空间结束，下面的步骤会被估计直到实体的位置确定：
    - 如果名称空间包含一个给定名称的名称空间成员，那么名称空间或类型名称指向那个成员并且根据成员被分类为一个名称空间或一个类型。
    - 否则，如果名称空间声明包括一个把一个引入的名称空间或类型和一个给定名称联系起来的使用别名指示，那么名称空间或类型名称指向那个名称空间或类型。
    - 否则，如果由名称空间声明的使用名称空间指示引入的名称空间包含一个确切类型，那么名称空间或类型名称指向那个类型。

- 否则，如果由名称空间声明的使用名称空间指示引入的名称空间包含多个确切类型，那么名称空间或类型名称是不明确的，并且会发生错误。
- 否则，名称空间或类型名称就是未定义的并且会发生错误。
- 否则，名称空间或类型名称是 *N.I* 的形式，这里 *N* 是一个有所有除了最右边一个指示符组成的名称空间或类型名称，而 *I* 是最右边的指示符。*N* 是最先确定的名称空间或类型名称，如果 *N* 的确定不成功，那么就会发生错误，否则，*N.I* 如下确定：
  - 如果 *N* 是名称空间而 *I* 是那个名称空间中一个可访问的成员，那么 *N.I* 指向那个成员并且根据成员分类为名称空间或类型。
  - 如果 *N* 是一个类或类型而 *I* 是 *N* 中的一个可访问类型，那么 *N.I* 指向那种类型。
  - 否则，*N.I* 是一个无效的名称空间或类型名称，并且会产生一个错误。

### 3.6.1 合格的名称

每个名称空间和类型都有一个完全限制名称，它在其它名称空间或类型中间唯一确定某个名称空间或类型。一个名称空间或变量的完全限制的名称 *N* 由下面决定：

- 如果 *N* 是全局名称空间的一个成员，它的完全限制的名称就是 *N*。
- 否则，它的完全限制的名称是 *S.N*，这里 *S* 是声明了 *N* 的名称空间或类型的完全限制的名称。

换句话说，*N* 的完全限制的名称是指向 *N* 的标识符的完全分等级路径，从全局名称空间开始。因为名称空间或类型的每个成员都要有一个唯一的名称，它是由于一个名称空间或类型的完全限制的名称总是唯一的造成。

下面的例子介绍了许多名称空间和类型声明，和与它们相对应的完全限制的名称。

```
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
}
```



## 4. 类型

C#语言的类型被分为三类：数值类型、引用类型和指针类型。

```
type:
    value-type
    reference-type
    pointer-type
```

指针类型只能用在不安全代码，并且将在§18.2中进行讨论。

数值类型与引用类型所不同的是，数值类型变量直接含有它们的数据，然而引用类型的变量存储对它们的数据的引用，就是后面要介绍的对象。对于引用类型，可能会出现两个变量引用相同对象的情况，这样对于一个变量的操作就有可能影响到由其它变量引用的对象。对于数值类型，每个变量都有它们自己对数据的拷贝，这样就不可能出现一个对变量的操作影响到另外一个的情况。

C#的类型系统是统一的，这样任何类型的数据都可以被看做对象。C#中的任何类型都直接或间接地从 object 类类型派生，而 object 是所有类型的最基本类。引用类型的数值被看做通过对象，这些对象通过把数值看做类型对象来简化。数值类型的数值通过包装和解包操作来被当做对象 (§4.3)。

### 4.1 数值类型

数值类型既是一个结构类型也是枚举类型。C#提供了一系列预定义结构类型，称为简单类型。简单类型通过保留字指定，并且进一步分成数字类型，整数类型和浮点数类型。

```
value-type:
    struct-type
    enum-type

struct-type:
    type-name
    simple-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

*enum-type:*

*type-name*

所有数值类型都隐式地从类 `object` 继承。不允许任何类型从数值类型派生，因而数值类型是被默认封闭的。

一个数值类型的变量通常包含一个那种类型的数值。不像引用类型，数值类型的数值不能为 `null` 或是引用一个进一步派生类型的变量。

对某个数值类型的变量赋值就会创建一个对所赋数值的拷贝，它复制了引用而不是引用所指定的对象。

#### 4.1.1 默认构造函数

所有类型都隐含地声明了一个公共的无参数的构造函数，称为默认构造函数。默认构造函数返回一个初始值为零的实例，为数值类型的默认数值：

- 对于所有单独类型，默认数值是由一个零位格式产生的数值：
  - 对于 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long` 和 `ulong`，默认的数值为 0。
  - 对于 `char`，默认的数值为 `'\x0000'`。
  - 对于 `float`，默认的数值是 `0.0f`。
  - 对于 `double`，默认的数值为 `0.0d`。
  - 对于 `decimal`，默认的数值为 `0.0m`。
  - 对于 `bool`，默认的数值为 `false`。
- 对于一个枚举类型 `E`，默认数值是 0。
- 对于结构类型，默认数值是把所有数值类型域设为它们的默认类型并且把所有引用类型域设为 `null` 的数值。

像其它任何构造函数一样，一个数值类型的默认的构造函数用 `new` 操作符调用。在下面的例子中，变量 `i` 和 `j` 都初始化为 0。

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

因为每个数值类型隐含的都有公共无参数构造函数，所以让一个结构类型包含一个外部声明的无参数构造函数是不可能的。一个结构类型可以允许声明一个参数化的构造函数。例如

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

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

如果已经给出上面的声明，那么语句

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

都会创建一个 Point，其中 x 和 y 被初始化为 0。

4.1.2 结构类型

一个结构类型是一个数值类型，它可以声明构造函数、常数、域、方法、属性、索引、操作符和嵌套类型。结构类型在§11中描述。

4.1.3 简单类型

C#提供了一系列的预定义结构类型，称为简单类型。这些简单类型通过关键词确定，但是这些关键词可以在 System 名称空间中的预定义结构类型关键词起简单的别名，就像下面表中所示。

关键字	有别名的类型
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

一个简单类型和它有别名的结构类型是不可分辨的。换句话说，当写下保留字 byte 时和写 System.Byte 确实没有什么区别，并且用 System.Int32 也与用保留字 int 相同。

因为一个简单类型代表了一个结构类型，所以每个简单类型都有成员。例如，int 有在 System.Int32 中声明的成员和从 System.Object 中继承的成员，并且下面的语句是允许的：

```
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
```

注意，整数文字上是 int 类型数据，并且同时也是 System.Int32 结构类型的数据。

简单类型与其它结构类型，其它结构类型允许包含附加操作符：

- 大多数简单类型允许通过使用文字来创建 (§**错误！未找到引用源。**)。例如，123 是 int 类型量，而 'a' 是字符类型量。C#使得不用对其它结构类型文字进行预定义，而其它结构类型数据基本上是通过那些结构类型的构造函数来创建。
- 当一个表达式的运算符都是简单类型常数时，编译器在编译时就可以对这个表达式进行赋值。这样一个表达式称为常数表达式 (§**错误！未找到引用源。**)。包括其它结构类型定义的操作符的表达式通常意味着运行时赋值。
- 通过 const 声明，就有可能声明一个简单类型 (§10.3) 的常数。不可能有其它结构类型的常数，但是 static readonly 域提供了相似的作用。
- 包括简单类型的转换可以参加由其它结构类型定义的转换操作符的赋值，但是用户定义的转换操作符不能参与另外一个用户定义操作符的赋值 (§**错误！未找到引用源。**)。

#### 4.1.4 整数类型

C#支持九种整数类型：sbyte、byte、short、ushort、int、uint、long、ulong 和 char。这些整数类型有下面的大小和数值范围：

- sbyte 类型表示有符号的 8 位整数，数值范围为 - 128 到 127。
- byte 类型表示无符号 8 位整数，数值范围为 0 到 255。
- short 类型表示有符号 16 位整数，数值范围为 - 32768 到 32767。
- ushort 类型表示无符号 16 位整数，数值范围为 0 到 65535。
- int 类型表示有符号 32 位整数，数值范围为 -2147483648 到 2147483647。
- uint 类型表示无符号 32 位整数，数值范围为 0 到 4294967295。
- long 类型表示有符号 64 位整数，数值范围为 -9223372036854775808 到 9223372036854775807。
- ulong 类型表示无符号 64 位整数，数值范围为 0 到 18446744073709551615。
- char 类型表示无符号 16 位整数，数值范围为 0 到 65535。char 类型的可能数值集符合 Unicode 字符集。

整数类型一元和二元操作符总是按有符号 32 位精度、无符号 32 位精度、有符号 64 位精度或无符号 64 位精度进行操作。

- 对于一元+和~操作符，操作数被转换为类型 T，这里 T 是 int、uint、long 和 ulong 中第一个可以完全代表操作数的所有可能值的类型。操作使用类型 T 的精度来实现，而结果的精度也是 T。
- 对于一元操作符 -，操作数被转换为类型 T，这里 T 是 int 和 long 中第一个可以完全代表操作数的所有可能值的类型。操作使用类型 T 的精度来实现，而结果的精度也是 T。一元操作符 - 不能应用于 ulong 类型操作数。
- 对于二元操作符+、-、\*、/、%、&、^、|、==、!=、>、<、>=和<= 操作符，操作数被转换为类型 T，这里 T 是 int、uint、long 和 ulong 中第一个可以完全代表操作数的所有可能值的类型。操作使用类型 T 的精度来实现，而结果的精度也是 T（或相关操作符 bool）。
- 对于二元操作符<<和>> 操作符，操作数被转换为类型 T，这里 T 是 int、uint、long 和 ulong 中第一个可以完全代表操作数的所有可能值的类型。操作使用类型 T 的精度来实现，而结果的精度也是 T

char 类型被分类为一种整数类型，但是它在两点上不同于其它整数类型：

- 没有从其它类型到字符类型的隐含的转换。甚至，即使 sbyte、byte 和 ushort 类型的数据完全可以用 char 类型代表，但是从 sbyte、byte 和 ushort 类型到 char 的隐含转换也不存在。
- char 类型的常数必须写成字符文字。字符常量可以只是写成与一个斜杠结合的整数文字。例如，(char)10 与 '\x000A' 相同。

checked 和 unchecked 操作符和语句用来控制检查整数类型算术操作和转换 (§7.5.13) 的溢出。在一段 checked 上下文中，一个溢出产生一个编译时错误或者引起扔出一个 OverflowException。在一段 unchecked 的上下文里，溢出被忽略并且不需要送到目标类型的任何高端位被丢弃。

#### 4.1.5 浮点类型

C# 支持两个浮点类型：float 和 double。float 和 double 类型用 32 位单精度和 64 位双精度 IEEE754 格式来表示，它提供了一系列数值：

- 正零和负零。在大多数情况下，正零和负零与简单的零值相同，但是它们的使用中间有一些区别。
- 正无穷大和负无穷大。无穷大是由一个非零成员除以零的操作产生的。例如，1.0 / 0.0 产生正无穷大，而 -1.0 / 0.0 产生负无穷大。
- 非数字数据，通常缩写为 NaN。NaN 是无效的浮点数操作产生的，例如零除以零。
- 形如  $s \times m \times 2^e$  的非零数据有限集，这里 s 是 1 或者 -1，而 m 和 e 由具体浮点数类型决定：对于 float， $0 < m < 2^{24}$  和  $-149 \leq e \leq 104$ ，对于 double， $0 < m < 2^{53}$  和  $-1075 \leq e \leq 970$ 。

float 类型可以代表的数值范围大约从  $1.5 \times 10^{-45}$  到  $3.4 \times 10^{38}$ ，有 7 位数字位精度。

double 类型可以代表的数值范围大约从  $5.0 \times 10^{-324}$  到  $1.7 \times 10^{308}$ ，有 15 到 16 位数字位精度。

如果二元运算符的一个操作数是浮点类型，那么其它操作数必须是整数类型或者是浮点数类型，并且操作按下面求值：

- 如果一个操作数是整数类型，那么那个操作数会被转换为与其它操作数一样的浮点数类型。
- 如果操作数是 double 类型，其它操作数就要转换为 double，操作就要按照 double 类型的范围和精度来进行，而且计算的结果也是 double 类型（对于相关操作，或者是 bool）。
- 否则，操作至少使用 float 的范围和精度，而且计算的结果也是 float 类型（对于相关操作，或者是 bool）。

包括赋值操作符的浮点操作符，从不产生异常。在异常情况下，浮点数操作会产生下面介绍的零、无穷大或 NaN 作为替代：

- 如果浮点数操作的结果对于目标形式来说太小，操作的结果就会转换为正零或负零。
- 如果浮点数操作的结果对于目标形式来说太大，操作的结果就会转换为正无穷大或负无穷大。
- 如果浮点数的操作是无效的，操作的结果就会转换为 NaN。
- 如果一个或所有浮点操作的操作数都是 NaN，那么操作的结果就变为 NaN。

浮点数操作可以用比操作结果的类型更高的精度来执行。例如，一些硬件结构支持一个比 double 类型更大范围和更高精度的“扩展的”或“long double”浮点数类型，并且会隐含地使用这个更高的精度来实现浮点数操作。只有在性能要额外付出时，这样的硬件结构才会被用来实现精度小一些的浮点数操作，而不需要执行同时丧失性能和精度，C# 允许所有的浮点数操作使用更高的精度类型。与给出更高精度的结果不同，这样几乎没有任何可测量的影响。在形如  $x * y / z$  的表达式中，这里的乘法产生一

个超出 double 类型范围的结果，但是后面的除法带来一个回到 double 范围的暂时结果，实际上在大一些的范围形式计算这个表达式会产生有限的结果而不是无穷大。

#### 4.1.6 十进制类型

十进制类型是一个 128 位数据类型，适合金融和货币计算。十进制类型可以代表的数值范围是从  $1.0 \times 10^{-28}$  到大约  $7.9 \times 10^{28}$ ，有 28 到 29 个有效数字位。

十进制类型数值的有限集合形式为  $s \times m \times 10^e$ ，这里  $s$  是 1 或者 -1， $0 \leq m < 2^{96}$  而  $-28 \leq e \leq 0$ 。十进制类型不支持有符号零、无穷大和 NaN。

一个十进制数由 96 位整数和十位幂表示。对于一个绝对数值小于 1.0m 的十进制数，数据就是第 28 个十进制位，但是没有更多。对于绝对值大于或等于 1.0m 的十进制数，数据可能是 28 或 29 数字位。与 float 和 double 类型相比，如 0.1 的十进制小数成员可以就用十进制表示。在用 float 和 double 表示时，这样的成员经常为无穷小数，使得这些表示有更大的舍入误差。

如果一个二元操作符的操作数是十进制类型，其它操作数也必须是整数类型或十进制类型。如果要使用一个整数类型操作数，在操作被执行前它会被转换为十进制数。

十进制类型的数值的操作就是 28 或 29 数字位，但是不会多于 28 十进制位。结果为最接近的可表示的数值，当结果与两个可表示数值都距离都相等时，选择在最小数据位上为奇数的数值。

如果十进制算术操作产生了一个在舍入后对于十进制形式太小的数据，操作的结果就变为零。如果一个十进制算术操作产生了一个对于十进制形式太大的数据，就会抛出一个 `OverflowException` 错误。

十进制类型比浮点类型有更高的精度但是有更小的范围。这样，从浮点数类型转换到十进制类型也许会产生溢出的异常，并且从十进制类型转换到浮点数类型也许会有精度损失。出于这些原因，不存在浮点数类型和十进制类型间的隐式转换，并且也没有显式的情况，在同一个表达式中把浮点数和十进制操作数混合在一起是不可能的。

#### 4.1.7 布尔类型

`bool` 类型表示布尔逻辑量，`bool` 类型的可能值为 `true` 和 `false`。

在 `bool` 和其它类型间不存在标准的转换。特别是，`bool` 类型与整数类型截然不同，`bool` 数据不能用于使用整数类型的地方，反之亦然。

在 C 和 C++ 语言中，零整数值或空指针可以被转换为布尔数值 `false`，而非零整数值或非空指针可以转换为布尔数值 `true`。在 C# 中，这样的转换由显式地把整数数值和零比较或显式地把对象和 `null` 比较来实现。

#### 4.1.8 枚举类型

枚举类型是一种有名称常数的独特类型。每个枚举类型都有前级类型，可以是 `byte`、`short`、`int` 或 `long`。枚举类型通过枚举声明来定义 (§14.1)。

### 4.2 引用类型

引用类型是一个类类型、一个接口类型、一个数组类型或是一个代表类型。

```

reference-type:
    class-type
    interface-type
    array-type
    delegate-type

class-type:
    type-name
    object
    string

interface-type:
    type-name

array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    [ dim-separatorsopt ]

dim-separators:
    ,
    dim-separators ,

delegate-type:
    type-name

```

一个引用数值是对于一个那种类型实例的引用，后面称为对象。特殊数值 `null` 是所有引用类型都适用的，并且表示缺乏实例。

#### 4.2.1 类类型

类类型定义了一个包括数据成员（常数、域和事件）、函数成员（方法、属性、索引、操作符、构造函数和析构函数）和嵌套类型。类类型支持继承，因为这种机制派生的类可以对基类进行扩展和特殊化。使用对象创建表达式 (§7.5.10.1) 来创建类类型的实例。

类类型将在 §10 讨论。

#### 4.2.2 对象类型

`object` (对象) 类型是所有其它类型的最基本类。C# 中的任何一个类型都是直接或间接地从 `object` 类类型派生的。

`object` 关键字是预定义 `System.Object` 类的简化的别名。使用 `object` 跟使用 `System.Object` 是相同的，反之亦然。

#### 4.2.3 字符串类型

字符串类型是直接从 `object` 派生的包装好的类类型。字符串类的实例表示统一的字符编码标准字符串。

字符串类型的数据写成一串文字 (§错误！未找到引用源。 )。

`string` 关键字是预定义 `System.String` 类的简化的别名。使用 `string` 跟使用 `System.String` 是相同的，反之亦然。

#### 4.2.4 接口类型

一个接口定义了一个协定。一个实现了接口的类或结构必须遵守它的协定。一个接口也许会从多个基本接口继承，而一个类或结构可以实现多个接口。

接口类型在§13中详述。

#### 4.2.5 数组类型

数组是一种数据结构，它包含了通过计算索引访问的变量成员。包含于数组中的变量，也称为数组的元素，都有相同的类型，而这个类型被称为数组的类型。

数组类型在§错误！未找到引用源。中详述。

#### 4.2.6 代表类型

代表是一种指向一个静态方法或一个对象的对象实例和对象方法的数据结构。

在 C 或 C++ 中与代表相同的是函数指针，但是功能指针只能指向静态函数，而代表可以指向静态和实例方法。在后面，代表不仅存储对于方法的入口点的引用，同时也存储对调用方法的对象实例的引用。

代表类型在§15中详述。

### 4.3 包装和解包

包装(boxing)和解包(unboxing)是 C# 类型系统中重要的概念。它通过允许任何数值类型的数据被转换为任何形式类型的对象提供了数值类型和引用类型间的紧密联系。包装和解包使得对在其中任何类型都可以最终被看作对象的类型系统的统一的观察变为可能。

#### 4.3.1 包装转换

包装转换允许任何数值类型可以隐式地转换为 `object` 类型或任何由数值类型实现的接口类型。包装一个数值类型的数据包括对对象实例的定位和把数值类型数据拷贝到那个实例中。

包装数值类型的数据的实际过程，可以通过想像一个对那种类型的包装类的实际例子来解释。对于数值类型 `T`，包装类要按下面定义：

```
class T_Box
{
    T value;
    T_Box(T t) {
        value = t;
    }
}
```

对于类型 `T` 的数值 `v` 的包装现在由执行表达式 `T_Box(v)` 来代替,并且返回类型为 `object` 的结果实例。这样，语句

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

从概念上符合



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

如上面的 `T_Box` 和 `int_Box` 的包装类型实际不存在，而被包装数据的动态类型实际上并不是一个类类型。作为替代，类型 `T` 的一个被包装的数据有动态类型 `T`，而使用 `is` 操作符的动态类型检查可以很方便地引用 `T`。例如，

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

将在控制台输出字符串“Box contains an int”。

包装转换隐式地把被包装的数据进行了备份。这与从引用类型到 `object` 类型的转换不同，在那里数据一直引用相同的实例，并被认为几乎不从类型 `object` 派生。例如，给出声明

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

下面的语句

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

因为在发生把 `p` 赋值给 `box` 的隐含包装操作时，`p` 被拷贝，所以将在控制台上输出数值 10。如果 `Point` 被声明为一个类，因为 `p` 和 `box` 将引用相同的实例，就会输出 20。

#### 4.3.2 解包转换

解包转换允许任何 `object` 类型或从任何由数值类型实现的接口类型，可以显式地转换为任何数值类型。一个解包操作由几部分动作组成，首先检查 `object` 实例是一个所给数值类型的被包装数据，然后把数值从实例中拷贝出来。

参考前面章节描述的假象的包装类型，从对象 `box` 到数值类型 `T` 的解包转换包括执行表达式 `((T_Box)box).value`。这样，语句

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

从概念上符合

```
object box = new int_Box(123);
int i = ((int_Box)box).value;
```

对于为了在运行时提供数值类型的解包转换，源变量数据必须是一个指向一个早先对那个数值类型数据打包创建的对象。如果源变量是 `null` 或引用一个不相关的对象，就会抛出一个 `InvalidCastException` 错误。



## 5. 变量

变量代表数据的实际存储位置。每各变量所能存储的数值由它本身的类型决定。C++ 语言是一种类型安全语言 (type - safe language, TSL)，而且 C++ 编译器保证每一个数值被保存在相应的变量中。变量的数值可以通过赋值或者 ++ 或 -- 运算符改变。

在变量被赋值以前，变量自身的类型必须被明确地声明。

在下面的章节中我们会提到，变量或者被初始化的或者未初始化的。一个初始化的变量在被定义时被赋予了一个确定的初始值，而未初始化的变量在定义时并未被赋予确定的初始值。对于一个在程序某处被认为具有确定数值的 IUA，必然在指向这一位置的所有可能的执行路径上存在赋值操作。

### 5.1 变量类型

C++ 共有七种变量类型：静态变量，实例变量，数组元素，数值参数，引用参数，输出参数和局部变量。下面的部分将分别对每一种变量类型做相关描述。

例子：

```
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 是一个实例变量，v[0] 是一个数组元素，a 是数值参数，b 是引用参数，c 是一个输出参数，i 是一个局部变量。

#### 5.1.1 静态变量

使用 static 修饰符定义的变量称为静态变量。静态变量在被创建并加载后生效，当被卸载后失效。

静态变量的初始值为此类型变量的默认值（参见§[错误！未找到引用源。](#)节）。

为了方便明确赋值检查，静态变量被认为是初始化过的。

#### 5.1.2 实例变量

一个没有 static 修饰符声明的域被称为实例变量。

#### 5.1.2.1 类中的实例变量

当创建某类的一个实例的时候，隶属于该类的实例变量也被生成，当不再有关于这个实例的引用而且实例的析构函数执行了以后，此实例变量失效。类中实例变量的初始值为这种类型变量的默认值（§5.2）。为了方便进行明确赋值检查，类中的实例变量是初始化过的。

#### 5.1.2.2 结构体中的实例变量

一个结构体中的实例变量与隶属与该结构体的结构体变量寿命相同。换句话说，结构体中的实例变量和其中的其他变量一样被同时创建，并且同时失效。而且该结构体中的实例变量的初始赋值状态和其中的其余变量一致。当一个结构体变量被是初始化过的，结构体的实例变量也是如此；反之如果一个结构体变量是未经初始化的时，结构体的实例变量也是未经初始化的。

#### 5.1.3 数组元素

当任意一个数组实例被创建时，这个数组的元素也被同时创建，当不再有任何正对这个数组实例的引用时，它的元素也就此失效。

数组中每个元素的初始值为该数字元素类型的默认值。为了方便明确赋值检查，所有的数字元素都被认为是初始化的。

#### 5.1.4 数值参数

当一个不带有 `ref` 或 `out` 修饰参数被声明时，我们称它为数值参数。

当被隶属的函数子句 `function member` (`method`, `constructor`, `accessor`, `operator`) 调用时，数值参数自动生成，同时被赋以调用中的参数值。当函数成员返回后，数值参数失效。

为了方便明确赋值检查，所有的数值参数都被认为是初始化过的。

#### 5.1.5 引用参数

当一个带有 `ref` 修饰语的参数被声明时，我们称之为引用参数。

引用参数本身并不创建新的存储空间。同时，引用参数指向函数子句调用中作为参数给出的相关变量表征的存储空间。这样，此形式参数的数值总是等于它所指向的变量。

下面时关于引用参数的赋值规则。请注意它们同§5.1.6 节中所给出的输出参数相关规则的区别。

- 在一个变量被传递给函数子句调用中相关引用参数之前，它自身必须被明确赋值，详见§5.3。
- 在函数子句界定的范围内，引用参数被认为是初始化过的。

在结构体类型的方法实例或存取程序实例中，关键字 `this` 就象是此结构体类型的引用参数，详见§7.5.7。

#### 5.1.6 输出参数

当一个带有 `out` 修饰语的参数被声明时，我们称之为输出参数。

输出参数本身并不创建新的存储空间。同时，输出参数指向函数子句调用中作为参数给出的相关变量表征的存储空间。这样，此输出参数的数值总是等于它所指向的变量。

下面时关于输出参数的赋值规则。请注意它们同§5.1.5 节中所给出的形式参数相关规则的区别。

- 在一个变量被传递给函数子句调用中相关输出参数之前，它自身不需要被明确地赋值，详见§5.3 节。
- 在函数子句调用中，每个被传递给输出参数的变量被认为在该执行路径中已被赋值。
- 在函数子句界定的范围内，输出参数被认为是初始化过的。
- 在函数子句返回之前，每一个输出参数必须被明确地赋值，详见§5.3 节。

在结构体类型的构造函数中，关键字 `this` 就象是此结构体类型的输出参数，详见§7.5.7 节。

### 5.1.7 局部变量

局部变量被局部变量声明语句创建，该语句可以在 `block` 块，`for` 循环语句或者 `switch` 分支语句中出现。当控制权进入 `block` 块，`for` 循环语句或者 `switch` 分支语句时，其中的相关局部变量被创建。当控制权离开 `block` 块，`for` 循环语句或者 `switch` 分支语句时，其中的相关局部变量随即失效。

局部变量不会被自动初始化，也就是说它不会有缺省值。为了方便明确赋值检查，局部变量被认为是初始化过的。局部变量声明语句可以包括一个变量初始化器，此时该变量在除它的变量初始化器表达式内的完全的有效范围中被认为是明确赋值的。

在一个局部变量的有效范围中，在它被声明之前的所有关于它的引用都被是错误的。

## 5.2 默认值

下面几种类型的变量在初始化时被自动赋予相应的默认值：

- 静态变量
- 类实例中的实例变量
- 数组元素

变量的默认值直接取决于它自身的类型和下面几种因素：

- 对于数值型的变量，默认值就是被此数值类型构造函数计算时使用的数值，详见§4.1.1 节。
- 对于形式型变量，默认值为 `null`。

## 5.3 明确赋值

在一个特定的函数子句可执行代码位置，如果通过静态流分析某一个变量被编译器证明为被自动初始化或者为至少一条赋值语句的执行对象，那么该变量将被认为是明确赋值的。关于明确赋值的相关规则如下所示：

- 一个初始化过的变量（详见§5.3.1 节）被认为是明确赋值的。
- 对于一个在特定位置的未初始化的变量（详见§5.3.2 节），如果所有可能的指向该位置的执行路径满足以下若干条件中的任何一个，那么它也被认为是明确赋值的：
  - a. 在某一个赋值语句中该变量作为左操作数出现，详见§7.13.17 节。
  - b. 任何一个调用表达式（详见§7.5.5 节）或者目标创建表达式（详见§7.10.1 节）将该变量作为一个输出参数传递。
  - c. 作为局部变量，该变量的局部变量声明语句包含变量初始化器。

一个结构体类型的实例变量的明确赋值状态将同时被单独和共同跟踪。另外，除了声明的各项规则之外，下面的各项规则适用于结构体类型变量和它们的实例变量：

- 如果一个实例变量中包含被明确赋值的结构体类型变量，则该变量被认为是明确赋值。
- 如果一个结构体类型的变量中的所有实例变量都被认为是明确赋值的，那么这个结构体变量也是明确赋值的。

明确赋值是下列各项的前提：

- 任何时刻，当一个变量获得自身的值时，它就是明确赋值的。这样就可以杜绝不确定的数值出现。除去下列情况，表达式中的变量都会获得相关变量数值：
  - a. 变量作为简单赋值语句的左操作数。
  - b. 变量作为一个输出参数被传递
  - c. 变量作为结构体类型的变量并在成员访问中作为左操作数出现。
- 当变量被作为形式参数被传递时，它本身必须是被明确赋值的。这样才能确保被调用的函数子句认为该形式参数是被明确赋值的。
- 无论函数子句在何处返回（通过返回语句 `return` 和程序执行到函数子句的末尾），所有函数子句中的输出参数都必须是被明确赋值的。这样就确保函数子句不会返回不具备明确数值的输出参数，也就使编译器认为函数子句把某一变量当作输出参数等同于都给变量赋值。
- 结构体类型的构造函数自何处返回，其中的 `this` 变量都必须使被明确赋值的。

下面的例程告诉我们 `try` 语句的不同 `block` 会使如何影响明确赋值的。

```
class A
{
    static void F() {
        int i, j;
        try {
            // neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
            j = 2;
            // i and j definitely assigned
        }
        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }
        finally {
            // neither i nor j definitely assigned
            i = 4;
            // i definitely assigned
            j = 5;
            // i and j definitely assigned
        }
        // i and j definitely assigned
    }
}
```

```

    }
}

```

静态流分析在检测明确赋值状态时将考虑 `&&` , `||` , `?` : 这些运算符的特殊运算。在下面例程中的每一个 method 我们将会看到

```

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
    }

    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
    }
}

```

变量 `i` 在 `if` 语句中的一个嵌套语句中是被明确赋值的，而在其余位置并不如此。在 `F` method 中的 `if` 语句的第一个嵌套语句中，因为表达式 `i = y` 被事先执行，所以变量 `i` 在是被明确赋值的。而在这个 `if` 语句的第二个嵌套语句中，由于变量 `i` 未被赋值，所以它被认为是未被明确赋值的。请注意，如果变量 `x` 的数值是负的，那么变量 `i` 是会被赋值的。同样，在 `G` 方法中，变量 `i` 在第二个嵌套语句中是被明确赋值的而在第一个嵌套语句中并不是这样。

### 5.3.1 初始赋值变量

下面所列各种类型的变量属于初始赋值变量：

- 静态变量
- 类实例中的实例变量
- 被初始赋值的结构体类型变量中的实例变量
- 数组元素
- 数值参数
- 形式参数

### 5.3.2 非初始赋值变量

下面类型的变量属于非初始赋值变量：

- 未被初始赋值的结构体变量中的实例变量
- 输出参数，包括结构体 `construct` 到 `r` 中的 `this` 变量
- 局部变量

### 5.4 变量引用

变量引用(variable-reference)是一种我们归类为变量的表达式。变量引用指向一个特定的存储地址，我们可以从这里获得它存储的当前值或者存入一个新的数值。在 C 和 C++ 中变量引用被称作 lvalue(左值)。

变量引用:

表达式

下面的结构需要一个表达式充当变量引用:

- 赋值表达式的左侧（可以是属性获取或索引获取程序）。
- 一个在方法或构造函数构造器调用中作为 `ref` 或 `out` 参数被传递的变量



## 6. 转换

### 6.1 隐式转换

下面几种类型的转换被称之为隐式转换

- 同一性转换
- 隐式数值转换
- 隐式枚举转换
- 隐式引用转换
- 包装转换
- 隐式常数表达式转换
- 用户自定义隐式转换

隐式转换可以在很多种情况下发生，包括§7.4.3 节中的函数子句调用，§7.6.8 节中的 `cast` 计算表达式和§7.13 节中的赋值语句。

预定义的隐式转换总会成功而且不会引发异常，适当合理的用户自定义隐式转换也可以展示出这些特性。

#### 6.1.1 同一性转换

同一性转换把任何类型转换为统一的类型。只有在所需转换的实体可以被转换到一种特定的类型时才可以进行同一性转换。

#### 6.1.2 隐式数值转换

隐式数值转换可以在下面类型中发生：

- 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 到 float 以及从 long 到 double 类型的转换可能会造成精度的损失,但并不会造成数量上的损失。除此之外的其他隐式数值转换不会损失任何信息。这里不存在转到 char 类型的隐式数值转换,也就是说其他的整型数据不会被自动地转换为字符型数据。

### 6.1.3 隐式枚举转换

一个隐式枚举转换允许小数-整数实字(decimal - integer - literal)被转换成任意的枚举类型。

### 6.1.4 隐式引用转换

隐式 reference 转换可以在下面类型之间发生：

- 从任意引用类型到对象。
- 从任意类类型 S 到任意类类型 T, 只要 S 是由 T 派生出来的。
- 从任意类类型 S 到任意接口类型 T, 只要 S 实现 T。
- 从任意接口类型 S 到任意接口类型 T, 只要 S 是由 T 派生出来的。
- 从一个带有元素类型  $S_E$  的数组类型 S 到一个带有元素类型  $T_E$  的数组类型 T, 只要下述各项均成立：
  - S 和 T 只是元素类型不同。换句话说, S 和 T 有相同的维度。
  - $S_E$  和  $T_E$  都是引用类型。
  - 存在从  $S_E$  到  $T_E$  隐式引用转换。
- 从任意数组类型到 System.Array。
- 从任意代表类型到 System.Delegate。
- 从任意数组类型或代表类型到 System.ICloneable。
- 从 null 类型到任意引用类型。

隐式引用转换指的是在引用类型间肯定可以成功的类型转换,它们是不需要实时检测的。

引用转换,不管是显式或是隐式的都不会改变被转换对象的引用一致性。换句话说,当引用转换发生时,它并未改变被转换对象的数值。

### 6.1.5 转换

包装转换允许任何数值类型被隐式地转换为类型对象,或者任何由这个数值类型实现的接口类型。包装一个数值类型的数值包括分配一个对象实例并且将数值类型的数值复制到实例当中。

在§4.3.1 节种我们会进一步地探讨包装转换。

### 6.1.6 隐式常数表达式转换

隐式常数表达式转换允许下列类型的转换：

- §7.15 节中的整型常数表达式可以被转换成为 sbyte, byte, short, ushort, uint 或者 ulong 类型,只有这个整型常数表达式的数值未超出目标类型的取值范围就行。
- 一个 long 类型的常数表达式可以被转换成为 ulong 类型,只有这个常数表达式的取值非负就行。

### 6.1.7 用户自定义隐式转换

一个用户自定义转换包括可选的标准隐式转换，紧随其后的是一个用户自定义的隐式转换运算符，之后是另一个可选的标准隐式转换。在§6.4.3 节中我们将详尽地描述用户自定义转换的取值规则。

## 6.2 显式转换

下列各种类型的转换属于显式转换：

- 所有隐式转换
- 显式数值转换
- 显式枚举转换
- 显式引用转换
- 显式引用转换
- 解包转换
- 用户自定义显式转换

显式转换可以在§7.6.8 节中的 `cast` 表达式中出现。

显式转换并不总是能够成功，比如那些已知的损失信息的转换和那些足以使用显式说明的不同类型之间的转换。显式转换包括所有隐式转换，这就意味允许冗余的 `cast` 表达式。

### 6.2.1 显式数值转换

显式数值转换指的是那些从一种数字类型到另外一种数字类型之间不能通过隐式类型转换实现的转换包括如下类型：

- 从 `sbyte` 到 `byte`，`ushort`，`uint`，`ulong` 或 `char` 类型的转换。
- 从 `byte` 到 `sbyte` 和 `char` 类型的转换。
- 从 `short` 到 `sbyte`，`byte`，`ushort`，`uint`，`ulong` 或 `char` 类型的转换。
- 从 `ushort` 到 `sbyte`，`byte`，`short` 或 `char` 类型的转换。
- 从 `int` 到 `sbyte`，`byte`，`short`，`ushort`，`uint`，`ulong` 或 `char` 类型的转换。
- 从 `uint` 到 `sbyte`，`byte`，`short`，`ushort`，`int` 或 `char` 类型的转换。
- 从 `long` 到 `sbyte`，`byte`，`short`，`ushort`，`int`，`uint`，`ulong` 或 `char` 类型的转换。
- 从 `ulong` 到 `sbyte`，`byte`，`short`，`ushort`，`int`，`uint`，`long` 或 `char` 类型的转换。
- 从 `char` 到 `sbyte`，`byte` 或 `short` 类型的转换。
- 从 `float` 到 `sbyte`，`byte`，`short`，`ushort`，`int`，`uint`，`long`，`ulong`，`char` 或 `decimal` 类型的转换。
- 从 `double` 到 `sbyte`，`byte`，`short`，`ushort`，`int`，`uint`，`long`，`ulong`，`char`，`float` 或 `decimal` 类型的转换。
- 从 `decimal` 到 `sbyte`，`byte`，`short`，`ushort`，`int`，`uint`，`long`，`ulong`，`char`，`float` 或 `double` 类型的转换。

由于形式类型转换包括所有隐式类型转换和显式数值转换，所以往往会出现§7.6.8 节中那样的使用 `cast` 表达式完成从一种数字类型到另外一种数字类型的转换。

显式数值转换有可能造成自身信息损失或者异常。一个显式的数值转换是通过下述方式进行的：

- 对于从一种整型到另外一种整型的转换，具体过程取决于溢出检测的包含转换操作的程序段：  
在检测完成的程序段，如果源操作数未超出目标类型数据的取值范围，转换可以顺利完成；但是如果源操作数超出目标类型数据的取值范围，转换将生成一个溢出异常。  
在未完成检测的程序段，转换总是可以顺利完成，但是源操作数的最高位会被略去。
- 对于从 float，double 或 decimal 类型到整型的转换，源操作数是接近 0 的整数，此值就是转换的结果。假如转换得到的整数超出目标类型允许的数值范围，转换将产生一个溢出异常。
- 对于从双精度到浮点数类型的转换，此双精度数值会逼近最为接近的浮点数数值。假如此双精度数太小无法使用浮点数表示，那么转换的结果将是 0 + 或 0 - 。假如此双精度数太大无法使用浮点数表示，那么转换结果将是 -∞ 或 +∞。假如此双精度数为 NaN，那么转换的结果也是 NaN。
- 对于从浮点数或双精度到十进制类型的转换，源操作数会被转换成相应的十进制数，而当此十进制数达到或超过 28 位长时，转换结果将是数值最接近的某一个十进制数。假如此源操作数太小无法用十进制数表示时，转换的结果将时 0。当源操作数位 NaN，∞ 或者超出十进制数的表示范围时，转换将产生一个 InvalidCast 异常。
- 对于从十进制数到浮点数或双精度类型的转换，此十进制数会被转换为数值上最接近的浮点数或双精度数。虽然此转换会损失精度，但并不会造成异常。

### 6.2.2 显式枚举类型转换

显式枚举类型转换指：

- 从 sbyte，byte，short，ushort，int，uint，long，ulong，char，float，double 或十进制到任意枚举类型的转换
- 从任意枚举类型到 sbyte，byte，short，ushort，int，uint，long，ulong，char，float，double 或十进制类型的转换
- 从某一种枚举类型到另外一种枚举类型的转换

显式枚举类型转换是通过把源枚举类型看作此类型的原始类型，之后进行相应类型之间的显式或隐式类型转换。例如，如果有一个枚举类型 E 且原始类型为 int，那么从 E 到 byte 的转换将被作为一个显示的从 int 到 byte 的数字转换处理 (§6.2.1 节)，而从 byte 到的 E 转换将被作为一个显示的从 byte 到 int 的数字转换处理 (§6.1.2 节)。

### 6.2.3 显式引用类型转换

显式引用类型转换指：

- 从对象(object)到任意引用类型之间的转换
- 从任意类类型 S 到任意类类型 T 之间的转换，其中 S 是 T 的基类。
- 从任意类类型 S 到任意类类型 T 之间的转换，其中 S 没有被封装，并且 S 没有实现 T。
- 从任意接口类型 S 到任意类类型 T 之间的转换，其中 T 没有被封装，并且 T 没有实现 S。
- 从任意接口类型 S 到任意接口类型之间的转换，其中 S 不是从 T 派生的。

- 从任意具有元素  $S_E$  的数组类型  $S$  到任意具有元素  $T_E$  的数组类型  $T$  之间的转换，只有满足下列条件才能进行：

$S$  和  $T$  只在数字元素类型上存在不同。换句话说， $S$  和  $T$  维数相同。

不论  $S_E$  或  $T_E$  都必须为引用类型。

从  $S_E$  或  $T_E$  之间的转换为显式引用转换。

- 从 `System.Array` 到任意数组类型之间的转换。
- 从 `System.Delegate` 到任意代表类型之间的转换。
- 从 `System.ICloneable` 到任意数组类型或者代表类型之间的转换。

显式引用转换需要实时检测保证其正确进行的引用类型之间的转换。

对于一个 `null` 类型或由源变量引用的实际的对象类型，必须是能够被隐式的引用转换来转换为目标类型的类型 (§6.1.4 节)。假如显式引用转换失败，转换会生成一个 `InvalidCast` 异常。

引用转换，无论是显式的或隐式的，并不改变被转换对象的引用一致性。换句话说，当一个引用转换改变数值的类型时，并未改变数值本身。

#### 6.2.4 解包转换

解包转换允许从类型对象到任意数值类型，或者从任意接口类型到任意实现该接口类型的数值类型间的转换。解包操作包括首先检测对象实例为给定数值类型包装的数值，之后从那个实例中复制出数值。解包转换将在 §4.3.2 节中进一步说明。

#### 6.2.5 用户自定义显式转换

用户自定义显式转换包括可选的标准显式转换，紧随其后的是一个用户自定义的显式转换运算符，之后是另一个可选的标准显式转换。在 §6.4.4 节中我们将详尽地描述用户自定义转换的取值规则。

### 6.3 标准转换

标准转换指那些可以在用户自定义转换中出现的预定义的转换。

#### 6.3.1 标准隐式转换

下列隐式转换属于标准隐式转换：

- 同一性类型转换(见 §6.1.1 节)
- 隐式数值类型转换(见 §6.1.2 节)
- 隐式引用转换(见 §6.1.4 节)
- 包装转换(见 §6.1.5 节)
- 隐式常数表达式转换(见 §6.1.6 节)

标准隐式转换是对立于用户自定义隐式转换而言的。

### 6.3.2 标准显式转换

标准显式转换包括所有标准隐式转换以及显式转换的子集，这是标准的隐式转换存在的原因。换句话说，假如存在一个从类型 A 到类型 B 的标准隐式转换，那么必然存在这两种类型相互之间的标准显式转换。

## 6.4 用户定义转换

C# 允许可以被用户自定义转换调用的预定义显式和隐式转换。用户自定义转换是通过§10.9.3 节类和结构体中出现的转换运算符引入的。

### 6.4.1 允许的用户自定义转换

C# 只允许某些用户自定义转换被声明。而且，它不允许重新定义一个已经存在的显式或者隐式转换。当所有下列的各项条件都满足时，类或结构体才被允许声明一个从源类型 S 到目标类型 T 的转换：

- S 和 T 是不同的类型
- S 或者 T 必须是包含转换操作符声明的类或结构类型。
- S 或者 T 是对象或一个接口类型。
- T 不是 S 的父类，S 也不是 T 的父类。

对于用户自定义转换的限定将在§10.9.3 节中做进一步地描述。

### 6.4.2 用户自定义转换的取值

用户自定义转换将称作源类型的数值转换为另外一种称作目标类型的数据类型。用户自定义转换的取值以寻找从原类型到目标类型的特定的转换操作符为中心，具体可以划分为以下几个步骤：

- 寻找要进行用户自定义类型转换的类和结构体。此集合包括原类型和它的父类以及目标类型和它的父类（我们假设只有类和结构体可以声明用户自定义操作符，非类类型没有父类。）。
- 根据这个类型集合决定哪种用户自定义转换操作符适用。对于适用的转换操作符，它必须可以进行从原类型到操作符自变量类型的标准转换（见§6.3 节），而且可以进行从操作符结果类型到目标类型的转换。
- 从这样使用的用户自定义操作符集合中找出最适用的操作符。总的来说，最适用的操作符的自变量类型与原类型最为接近，而且它的结果类型同目标类型最为接近。创建最使用用户自定义操作符的确切规则如下所示。

一旦特定的用户自定义转换操作符被确定，真正的用户自定义转换执行可以分为以下三个步骤：

1. 如果需要，进行一个标准转换把源数据类型转换为用户自定义转换操作符变量类型。
2. 启动用户自定义转换操作符进行转换
3. 如果需要，进行一个标准转换把用户自定义转换结果数据类型转换为目标数据类型。

用户自定义转换永远不会涉及到多余一个用户自定义转换操作符。换句话说，从类型 S 到类型 T 的用户自定义转换不会先进行一次从 S 到 X 的转换，之后再进行一次从 X 到 T 的转换。

下面给出用户自定义显式或隐式转换取值的确切定义。定义使用如下术语：

- 如果存在一个由类型 A 到类型 B 的标准隐式转换（见§6.3.1 节），如果 A 或者 B 为接口类型，那么 A 被 B 包含，或者说 B 包含 A）。
- 在类型集中的最大的包含类型是包含所有其余类型的类型。假如并不存在这样的一种类型，我们称这个类型集合没有最大包含类型。用更直观的话说，最大包含类型是集合中“最大”的类型 - 集合中所有其余类型都可以转换成这种类型。
- 在类型集中的最大被包含类型是可以被其余所有类型包含的类型。如果集合中并不存在这样被所有其余类型包含的类型，那么我们称此集合没有最大被包含类型。用更直观的话说，最大包含类型是集合中“最小”的类型 - 这种类型可以转换成集合中所有其余类型。

### 6.4.3 用户自定义隐式转换

一个从类型 S 到类型 T 的用户自定义隐式转换依照下面所述方式进行：

- 寻找要进行用户自定义转换操作的类型集合，比如说我们称之为 D。此集合包括 S（S 为一个类和结构体）和 S 的父类（如果 S 是一个类），T（T 为一个类和结构体）和 T 的父类（如果 T 是一个类）。
- 寻找适用的用户自定义转换操作符，比如我们称之为 U。此集合包括若干在 D 中声明的可以完成从一个包含 S 的类型到一个被 T 包含的类型之间转换的用户自定义隐式转换操作符。假如 U 是空的，那么转换是未定义的，将会有错误发生。
- 找到 U 中操作符最适用的源类型，比如说为 S<sub>x</sub>。

如果 U 中任意操作符都是进行自 S 的转换，那么 S<sub>x</sub> 就是 S。

否则，S<sub>x</sub> 是 U 中操作符源类型的最大被包含类型。如果无法找到最大被包含类型，那么转换将会发生错误。

- 找到 U 中操作符最适用的目标类型，比如说为 T<sub>x</sub>。

如果 U 中任意操作符都可以转换称类型 T，那么 T<sub>x</sub> 就是 T。

否则，T<sub>x</sub> 是 U 中操作符目标类型的最大包含类型。如果不存在这样的最大包含类型，那么转换将会发生错误。

- 假如 U 中存在由类型 S<sub>x</sub> 到 T<sub>x</sub> 的用户自定义转换操作符，那么这个操作符就是最适用的转换操作符。如果不存在这样的操作符，或者存在不止一个这样的操作符，那么转换都会发生错误。

假如 S 不是 S<sub>x</sub>，进行自 S 到 S<sub>x</sub> 的标准隐式类型转换。

启动适用的操作符完成自 S<sub>x</sub> 到 T<sub>x</sub> 的转换。

假如 T<sub>x</sub> 不是 T，进行自 T<sub>x</sub> 到 T 的标准隐式类型转换。

### 6.4.4 用户自定义显式类型转换

一个从类型 S 到类型 T 的用户自定义显式转换依照下面所述方式进行：

- 寻找要进行用户自定义转换操作的类型集合，比如说我们称之为 D。此集合包括 S（S 为一个类和结构体）和 S 的父类（如果 S 是一个类），T（T 为一个类和结构体）和 T 的父类（如果 T 是一个类）。

- 寻找适用的用户自定义转换操作符，比如我们称之为  $U$ 。此集合包括若干在  $D$  中声明的可以完成从一个包含  $S$  的类型到一个被  $T$  包含的类型之间转换的用户自定义隐式转换操作符。假如  $U$  是空的，那么转换是未定义的，将会有错误发生。
- 找到  $U$  中操作符最适用的源类型，比如说为  $S_x$ 。  
如果  $U$  中任意操作符都是进行自  $S$  的转换，那么  $S_x$  就是  $S$ 。  
否则，假如  $U$  中所有操作符源都是完成自包含  $S$  的类型的转换，那么  $S_x$  就是这些操作符源类型的最大被包含类型。如果无法找到最大被包含类型，那么转换将会发生错误。  
否则  $S_x$  就是  $U$  中操作符原类型的最大包含类型。如果不存在这样的最大包含类型，转换将发生错误。
- 找到  $U$  中操作符最适用的目标类型，比如说为  $T_x$ 。  
如果  $U$  中任意操作符都可以转换称类型  $T$ ，那么  $T_x$  就是  $T$ 。  
否则，如果如果  $U$  中任意操作符都可以转换得到被  $T$  包含的类型，那么  $T_x$  就是这些操作符目标类型的最大包含类型。如果不存在这样的最大包含类型，转换将发生错误。  
否则， $T_x$  是  $U$  中操作符目标类型的最大包含类型。如果不存在这样的最大包含类型，那么转换将会发生错误。
- 假如  $U$  中存在由类型  $S_x$  到  $T_x$  的用户自定义转换操作符，那么这个操作符就是最适用的转换操作符。如果不存在这样的操作符，或者存在不止一个这样的操作符，那么转换都会发生错误。  
假如  $S$  不是  $S_x$ ，进行自  $S$  到  $S_x$  的标准显式类型转换。  
启动适用的操作符完成自  $S_x$  到  $T_x$  的转换。  
假如  $T_x$  不是  $T$ ，进行自  $T_x$  到  $T$  的标准显式类型转换。



## 7. 表达式

一个表达式就是指定一个计算的一系列一个操作符和操作数。本章定义语法、求值的顺序和表达式的意义。

### 7.1 表达式分类

一个表达式可以归类为下面的一种：

- 一个数值。每个数值都有相应的类型。
- 一个变量。每个变量都有相关的类型，也就是变量声明的类型。
- 一个名称空间。通过这种归类的一个表达式只能表现为一个成员访问 (§7.5.4) 的左手部分。在任何其它上下文中，一个表达式被分类为一个名称空间会造成错误。
- 一种类型。通过这种归类的一个表达式只能表现为一个成员访问 (§7.5.4) 的左手部分。在任何其它上下文中，一个表达式被分类为一个类型会造成错误。
- 一个方法组，这是一系列由成员查找 (§7.3) 产生的重载方法。一个方法组可以有相关的实例表达式。当调用一个实例方法时，对实例表达式的求值的结果就变成用 `this` (§7.5.7) 修饰的实例。一个方法组只允许用于一个调用表达式 (§7.5.5) 或一个创建代表表达式 (§7.5.10.3) 中。在任何其它上下文中，一个表达式被分类为一个方法组会造成错误。
- 一个属性访问，每个属性访问都有相应的类型，也就是属性的类型。此外，一个属性访问也可以有一个相关的实例表达式。当一个实例属性访问的访问程序（`get` 或 `set` 模块）被调用的时候，对实例表达式的求值就变为用 `this` 修饰的实例 (§7.5.7)。
- 一个事件访问。每个事件访问都有相应的类型，也就是事件的类型。此外，一个事件访问也可以有一个相关的实例表达式。一个事件访问可能被表现为 `+=` 和 `-=` 操作符 (§ 错误！未找到引用源。) 的操作数的左手部分。在任何其它上下文中，一个表达式被分类为一个事件访问会造成错误。
- 一个索引访问。每个索引访问都有相应的类型，也就是索引的类型。此外，一个索引访问也可以有一个相关的实例表达式和一个相关的参数列表。当一个索引访问的访问程序（`get` 或 `set` 模块）被调用的时候，对实例表达式的求值就变为用 `this` 修饰的参数列表 (§7.5.7)。
- 空。这发生在 表达式是一个返回类型为 `void` 的方法的调用时。一个表达式被分类为空只在语句表达式的文字中有效 (§ 错误！未找到引用源。 )。

一个表达式的最后结果不会是一个名称空间、类型、方法组或是事件访问。而且，如前面所述，这些表达式的分类只是一个中间结构，只允许在某些地方存在。

一个属性访问或索引访问总是在执行一个对 `get` 访问符或 `set` 访问符时作为数值被重分类。特殊的访问符由属性或者索引访问的上下文觉得：如果访问的目的是赋值，`set` 访问符就被调用来赋新的数值 (§ 错误！未找到引用源。 )。否则，`get` 访问符被调用来获得当前的数值 (§7.1.1)。

### 7.1.1 表达式的数值

大多数涉及到一个表达式的结构基本上都需要表达式给出一个数值。在那样的情况下，如果实际表达式给出一个名称空间、一个类型、一个方法组或空，就会产生错误。然而，如果表达式表示一个属性访问、一个索引访问或是一个变量，属性、索引或变量的值就会被隐含地替代：

- 变量的数值就是当前存储在由变量指定的存储位置的数值。一个变量必须在他的数值可以被获得前明确赋值(**§错误！未找到引用源。**)，否则就会产生一个编译时的错误。
- 属性访问表达式的数值通过调用属性的 get 访问符来获得。如果属性没有 get 访问符，就会产生错误。否则，就会执行一个函数成员的调用 (§7.4.3)，而且调用的结果变为属性访问表达式的数值。
- 索引访问表达式的数值通过调用索引的 get 访问符来获得。如果索引没有 get 访问符，就会产生错误。否则，就会执行一个与属性访问表达式相关的参数列表的函数成员的调用 (§7.4.3)，而且调用的结果变为属性访问表达式的数值。

## 7.2 操作符

表达式由操作数和操作符来构造。表达式的操作符指示出对操作数采取哪种操作。操作符的例子包括 +、-、\*、/ 和 new。操作数的例子包括文字、域、局部变量和表达式。

这力有三种类型的操作符：

- 一元操作符。一元操作符有一个操作数并且或是使用前缀符号（例如 - x）或是使用后缀符号（例如 x++）。
- 二元操作符。二元操作符有两个操作数并且使用中间符号（例如 x+y）。
- 三元操作符。只有一个三元操作符 ? :。三元操作符有三个操作数并且使用中间符号 (c? x: y)。

表达式中操作符求值的顺序由操作符的优先级和结合顺序决定 (§7.2.1)。

一些操作符可以被重载。操作符重载允许指定用户定义操作符的执行，这里一个或多个操作数为用户定义的类型或结构类型 (§7.2.2)。

### 7.2.1 操作符优先级和结合顺序

当一个表达式包含多个操作符，操作符的优先级控制单个操作符求值的顺序。例如，表达式  $x + y * z$  被求值为  $x + (y * z)$ ，因为 \* 操作符比 + 操作符有更高的优先级。操作符的优先级是由与它相关的语法创建确定的。例如，由一个乘法表达式序列组成的加法表达式被 + 或 - 分开，这时就会给 + 或 - 比 \*、/ 和 % 操作符低一些的优先级。

下面的表中从高到低总结了所有操作符的优先级顺序：

节	种类	操作符
0	初级的	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
错误！ 未找到 引用 源。	一元的	+ - ! ~ ++x --x (T)x
错误！ 未找到 引用 源。	乘法的	* / %
错误！ 未找到 引用 源。	加法的	+ -
错误！ 未找到 引用 源。	移位	<< >>
错误！ 未找到 引用 源。	关系的	< > <= >= is
错误！ 未找到 引用 源。	相等的	== !=
错误！ 未找到 引用 源。	逻辑与	&
错误！ 未找到 引用 源。	逻辑异或	^
错误！ 未找到 引用 源。	逻辑或	
错误！	条件与	&&

未找到引用源。		
错误！未找到引用源。	条件或	
错误！未找到引用源。	条件的	?:
错误！未找到引用源。	赋值	= *= /= %= += -= <<= >>= &= ^=  =

当一个操作数在两个有相同优先级的操作符中间时，操作符的结合顺序控制操作怎样实现：

- 除了赋值操作符，所有二元操作符都是左结合的，意思就是操作从左向右完成。例如， $x + y + z$  被求值为  $(x + y) + z$ 。
- 赋值操作符和条件操作符都是右结合的，意思就是操作从右向左完成。例如， $x = y = z$  被求值为  $x = (y = z)$ 。

优先级和结合顺序可以通过使用括号来控制。例如  $x + y * z$  先把  $y$  和  $z$  相乘，然后再把结果和  $x$  相加，但是  $(x + y) * z$  先把  $x$  和  $y$  相加，然后在把结果和  $z$  相乘。

7.2.2 操作符重载

所有一元和二元操作符都有预定义的执行方式，在任何表达式中都会自动实行。除了预定义的执行方式外，用户定义的执行方式可以通过包括类和结构(§10.9)中的操作符声明来引入。用户定义的操作符执行通常比预定义操作符声明的优先级高：只有当没有可使用的用户定义的操作符执行存在时才会考虑预定义的操作符执行。

可重载一元操作符有：

+ - ! ~ ++ -- true false

可重载二元操作符有：

+ - \* / % & | ^ << >> == != > < >= <=

只有上面列出的操作符可以被重载。另外，不能重载成员访问、方法调用或=、&&、||、?:、new、typeof、sizeof 和 is 操作符。

当一个二元 操作符被重载，相应的赋值操作符也被隐式地重载。例如，一个操作符\*的重载同时也是操作符\*=的重载。这在§错误！未找到引用源。中将被说明。注意赋值操作符自己( = )不能被重载。一个赋值通常把一个数值的位方式的复制放到变量里。

某些操作，如 (T)x，通过提供用户定义转换(§错误！未找到引用源。)重载。

元素访问，例如 a[x]，也被看做是一个重载操作符。用户定义的索引通过索引来支持(§10.8)。

在表达式中，使用操作符符号来实行操作符，而在声明中，用功能符号来实现操作符。下面的表中介绍了操作符和一元、二元操作符的功能符号之间的关系。在第一行，表示任何可重载一元操作符。在第二行，*op* 表示二元操作符++和—。在第三行，*op* 表示任何可重载操作符。

操作符号	功能符号
<i>op</i> x	操作符 <i>op</i> (x)
x <i>op</i>	操作符 <i>op</i> (x)
x <i>op</i> y	操作符 <i>op</i> (x, y)

用户定义的操作符声明通常需要至少一个参数是包含操作声明的类或类型。这样，就不允许用户定义的操作符与预定义的操作符有相同的签名。

用户定义的操作符声明不能修改操作符的语法、优先级或结合顺序。例如，\*操作符通常是一个二元操作符，通常有§7.2.1中指出的优先级等级，并且通常是左结合。

虽然用户定义的操作符可以实现任何它希望的计算，与那些直觉上希望不同的产生结果的执行是很让人失望的。例如，操作符==的执行应该比较两个操作数并且返回相应的结果。

从§7.5到§**错误！未找到引用源。**，描述的独立操作符指出了操作符的预定义执行和使用在每个操作符上的任何附加规则。这个描述采用了一元操作符重载协议、二元操作符重载协议和数字升级（*numeric promotion*）的形式，对它们的定义会在后面的章节找到。

7.2.3 一元操作符重载分析

一个有 *op* x 或 x *op* 形式的操作，按照下面过程进行，这里 *op* 是一个可重载的一元操作符，而 x 是类型 X 的一个表达式：

- 由 x 提供的为 *op*(x)操作候选的用户定义操作符集是使用§7.2.5中的规则决定的。
- 如果候选的用户定义操作符集不是空的，那么这个就会称为操作的候选操作符集。否则，预定义的一元操作符 *op* 就成为候选操作符集。所给的操作符的预定义执行在详述操作符（§7.5 和§**错误！未找到引用源。**）时会介绍。
- §7.4.2的重载分析规则被应用于候选操作符集，来选择关于参数列表(x)的最好的操作符，而这个操作符变为重载分析过程的结果。如果重载分析在选择一个最好的操作符时失败了，就会产生一个错误。

7.2.4 二元操作符重载分析

一个有 x *op* y 形式的二元操作按下面进行：（这里 *op* 是一个可重载的二元操作符，x 是一个类型 X 的表达式，y 是一个类型 Y 的表达式）

- 为了操作 *op*(x, y)由 x 和 Y 提供的候选用户定义操作符集是确定的。由 s 提供的候选操作符和由 Y 提供的候选操作符联合组成了候选操作符集是通过使用§7.2.5的规则确定的。如果 X 和 Y 是相同的类型，或者如果 X 和 Y 是从一个公共基础类型派生的，那么共享的候选操作符只在联合集中出现一次。
- 如果候选用户操作符集不是空的，那么就会变成操作的候选操作符集。否则，预定义的一元操作符 *op* 就成为候选操作符集。所给的操作符的预定义执行在详述操作符（§7.5 和§**错误！未找到引用源。**）时会介绍。

- §7.4.2的重载分析规则被应用于候选操作符集，来选择关于参数列表(x, y)的最好的操作符，而这个操作符变为重载分析过程的结果。如果重载分析在选择一个最好的操作符时失败了，就会产生一个错误。

### 7.2.5 候选用户定义操作符

给出一个类型 T 和一个操作符 *op*(A)，这里 *op* 是一个可重载操作符而 A 是一个参数列表，这个为了操作符 *op*(A) 由 T 提供的候选用户定义操作符集按下面确定：

- 对于 T 中声明的所有操作符 *op*，如果对于参数列表 A 来说至少有一个操作符是可用的，那么候选操作符集就会包括所有 T 中声明的可用操作符 *op*。
- 否则，如果 T 是 object，候选操作符即为空。
- 否则，由 T 提供的候选操作符集就是由 T 的直接基类提供的候选操作符集。

### 7.2.6 数字升级

数字升级由自动进行某种预定义的一元和二元数字操作符的操作数的转换组成。数字升级不是独立的机制，而是一种使用重载分析来预定义操作符的效果。数字升级不会影响用户定义的操作符的求值，虽然用户定义的操作符可以被执行类展示相似的效果。

作为一个数字升级的例子，考虑二元操作符\*的预定义执行：

```
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);
```

当重载分析规则 (§7.4.2) 被应用于这个操作符集的时候，作用是选择第一个操作符，为此存在操作数类型隐式的转换。例如，对于操作 *b \* s*，这里 *b* 是 byte 而 *s* 是 short，重载分析选择操作符 *\*(int, int)* 作为最好的操作符。这样，效果为 *b* 和 *s* 被转换为 int 而结果的类型是 int。与此类似，对于操作 *i \* d*，这里 *i* 是一个 int 而 *d* 是一个 double，重载分析选择操作符 *\*(double, double)* 作为最好的操作符。

#### 7.2.6.1 一元数字升级

一元数字升级在进行预定义的+、-和~一元操作符操作时发生。一元数字升级简单地由类型 sbyte、byte、short、ushort 或 char 的操作数转换为 int 类型组成。另外，对于一元操作符，一元数字升级把 uint 类型操作数转换为 long 类型。

#### 7.2.6.2 二元数字升级

二元数字升级在操作符为预定义的+、-、\*、/、%、&、|、^、==、!=、>、<、>=和<= 二元操作符时发生。二元数字升级隐式地把所有操作数转换为一个统一的类型，而有些不相关的操作符也变为操作结果的类型。二元数字升级有下面应用组成，按顺序列在下面：

- 如果操作数是十进制类型，其它操作数就被转换为十进制类型，而如果其它的操作数为 float 或 double 类型，就会产生一个错误。
- 另外，如果操作数是 double 类型，其它操作数就被转换为 double 类型。
- 另外，如果操作数是 float 类型，其它操作数就被转换为 float 类型。

- 如果操作数是 `ulong` 类型，其它操作数就被转换为 `ulong` 类型，而如果其它的操作数为 `sbyte`、`short`、`int` 或 `long` 类型，就会产生一个错误。
- 另外，如果操作数是 `long` 类型，其它操作数就被转换为 `long` 类型。
- 如果操作数是 `uint` 类型，其它的操作数为 `sbyte`、`short` 或 `int` 类型，所有的操作数都会被转换为 `long`。
- 另外，如果操作数是 `uint` 类型，其它操作数就被转换为 `uint` 类型。
- 另外，所有操作数都被转换为 `int` 类型。

注意，第一个规则不接受任何掺杂了 `double` 和 `float` 类型的十进制类型。这个规则是根据在十进制类型与 `double` 和 `float` 类型间没有隐式转换来确定的。

也要注意，当其它操作数是有符号整数类型时，不允许操作数为 `ulong` 类型。原因是没有一个整数类型可以同时满足 `ulong` 和有符号整数类型的全部取值范围。

在上面的所有情况中，一个表达式可以被用来显式地把一个操作数转换为另一种和其它操作数一致的类型。

在例子中

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

会产生一个编译时错误，因为一个十进制数不能乘以 `double`。这个操作可以通过显式地把第二个操作数转换为十进制类解决：

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

### 7.3 成员查询

成员查询是根据在一个类型的上下文中一个名称的意义是确定的来进行。成员查询会发生在对一个表达式中的简单名称 (§7.5.2) 或成员访问 (§7.5.4) 进行求值的时候。

类 `T` 中的名称 `A` 的成员查询按下面步骤进行：

- 首先，`T` 中声明的名为 `N` 的所有可访问 (§3.3) 成员和 `T` 的基本类型 (§7.3.1) 已经被构造。包括 `override` 修饰符的声明被排斥在外。如果没有名为 `N` 的成员存在或可访问，那么查询产生无匹配，而下面的步骤就不再进行。
- 下面，被其它成员隐藏的成员被从集合中除去，对于集合中任何一个成员 `S.M`，使用下面的规则：（这里在类型 `S` 中，成员 `M` 被声明）
  - 如果 `M` 是一个常数、域、属性、事件、类型或枚举成员，那么所有在 `S` 的基类中声明的成员都要被从集合中去掉。
  - 如果 `M` 是一个方法，那么 `S` 的一个基本类型中声明的所有非方法成员都要从这个集合中去掉，并且 `S` 的基本类型中声明的所有与 `M` 有相同签名的方法都要从这个集合中去掉。
- 最后，把隐含成员去掉后，查询的结果就确定了：
  - 如果集合是由单个非方法成员构成，那么这个成员就是查询的结果。

- 另外，如果集合中只包含方法，那么这个方法组就是查询的结果。
- 另外，如果查询是不明确的，就会产生编译时错误（这个情况只能发生在一个有多个直接基本接口的接口的成员查询中）。

对于不同于接口的类型中的成员查询和严格单继承的接口中的成员查询（继承链中的每个接口没有或只有一个直接基接口），成员查询规则的作用是简单的，派生的成员隐藏了有相同名称或签名的基本成员。这样的单继承查询不会是不明确的。不明确的查询只可能在多继承接口成员查询时发生，这将在§13.2.5中描述。

### 7.3.1 基类型

出于成员查询的目的，类型 T 可以被考虑有下面的基类型：

- 如果 T 是对象(object)类型，那么 T 没有基类型。
- 如果 T 是一个数值类型，那么 T 的基类型是类类型对象(object)。
- 如果 T 是一个类类型，那么 T 的基类型是 T 的基类，包括类类型对象(object)。
- 如果 T 是一个接口类型，那么 T 的基类型是 T 的基接口和类类型对象(object)。
- 如果 T 是一个数组类型，那么 T 的基类型就是类类型 System.Array 和对象(object)。
- 如果 T 是一个代表类型，那么 T 的基类型就是类类型 System.Delegate 和对象(object)。

## 7.4 函数成员

函数成员是一些包含可执行语句的成员。函数成员总是类型的成员，并且不能是名称空间的成员。C# 定义了下面五种函数成员：

- 构造函数
- 方法
- 属性
- 索引
- 用户定义操作符

在函数成员中包含的语句通过函数成员调用来执行。函数成员调用的事件语法是根据不同的函数成员种类来定。然而，所有的函数成员调用都是表达式，它允许参数被传送到函数成员中，并且允许函数成员计算和返回一个结果。

一个函数成员调用的参数列表 (§7.4.1)提供了函数成员实际数值或的对参数引用的变量。

构造函数、方法、索引和操作符的引用，使用重载分析来确定调用哪个候选功能参数集。这个过程在§7.4.2中描述。

一旦在编译时一个详细的函数成员被确定，可能会通过重载分析，实际运行时调用函数成员的过程将在§7.4.3中描述。

下面的表格总结了在构造五种函数成员发生的过程。在表中，e、x、y 和 value 指示表达式被作为变量或数值分类，T 表示一个作为类型分类的表达式，F 是一个方法的简单名称，P 是一个属性的简单名称。



结构	例子	描述
构造函数调用	<code>new T(x, y)</code>	重载分析被应用于在类或结构 T 中选择最好的构造函数。构造函数被调用，有参数列表 (x, y)。
方法调用	<code>F(x, y)</code>	重载分析被应用于在包含的类或结构中选择最好的方法 F。方法调用有参数列表 (x, y)。如果方法不是 static，实例表达式就是 <code>this</code> 。
	<code>T.F(x, y)</code>	重载分析被应用于在类或结构 T 中选择方法 F。如果方法不是 static，会发生错误，方法调用有参数列表 (x, y)。
	<code>e.F(x, y)</code>	重载分析被应用于在 e 给定的类、结构或接口中选择方法 F。如果方法不是 static，会发生错误，方法调用有参数列表 (x, y)。
属性访问	<code>P</code>	在包含类或结构中的属性的 get 访问符被调用。如果 P 是只写的，会产生错误，如果 P 不是静态的，实例表达式就是 <code>this</code> 。
	<code>P = value</code>	在包含类或结构中的属性 P 的 set 访问符被调用，并且有参数列表 (value)。如果 P 是只读的，会产生错误，如果 P 不是静态的，实例表达式就是 <code>this</code> 。
	<code>T.P</code>	在类或结构 T 中的属性的 get 访问符被调用。如果 P 是只写的，会产生错误，如果 P 不是静态的，实例表达式就是 <code>this</code> 。
	<code>T.P = value</code>	在类或结构 T 中的属性 P 的 set 访问符被调用，并且有参数列表 (value)。如果 P 是只读的，会产生错误，如果 P 不是静态的，实例表达式就是 <code>this</code> 。
	<code>e.P</code>	在类型 e 提供的类、结构或接口中的属性的 get 访问符被调用。如果 P 是只写的，会产生错误。
	<code>e.P = value</code>	在类型 e 提供的类、结构或接口中的属性 P 的 set 访问符被调用，并且有参数列表 (value)。如果 P 是只读的，会产生错误。
索引访问	<code>e[x, y]</code>	重载分析被用于选择类型 e 提供的类、结构或接口。索引的 get 访问符被调用，有实例表达式 e 和参数列表 (x, y)。如果 P 是只写的，会产生错误。
	<code>e[x, y] = value</code>	重载分析被用于选择类型 e 提供的类、结构或接口。索引的 set 访问符被调用，有实例表达式 e 和参数列表 (x, y, value)。如果 P 是只读的，会产生错误。
操作符调用	<code>-x</code>	重载分析被用于在类型 x 所给的类或结构中选择最好的一元操作符。所选择的操作符被调用，有参数列表 (x)。

结构	例子	描述
	<code>x + y</code>	重载分析被用于在类型 <code>x</code> 和 <code>y</code> 所给的类或结构中选择最好的一元操作符。所选择的操作符被调用，有参数列表 ( <code>x</code> , <code>y</code> )。

7.4.1 参数列表

每种函数成员调用都包括一个参数列表，它提供了对函数成员的参数的引用的真实数据或变量。指定函数成员的参数列表的语法要根据函数成员的种类：

- 对于构造函数、方法和代表，参数被指定为一个参数列表，如下面所描述的一样。
- 对于属性，当调用 `get` 访问符时参数列表是空的，并且当调用 `set` 访问符时由所指定的复制操作符的右边操作数构成。
- 对于标签，参数列表由标签访问中在方括号中指定的表达式构成。当调用 `set` 访问符时，参数列表另外包括所指定的复制操作符的右边操作数。
- 对于用户定义的操作符，参数列表由一元操作符的单操作数或二元操作符的两个操作数构成。

属性、索引和用户定义的操作符的复制通常要传递数值参数 (§10.5.1.1)。这些函数成员的种类不支持引用和输出参数。

一个构造函数、方法或代表调用的参数按一个参数列表指定：

```
argument-list:
    argument
    argument-list , argument

argument:
    expression
    ref variable-reference
    out variable-reference
```

一个参数列表由零或者更多的参数组成，用逗号隔开。每个参数可以使用下面方法中的一种：

- 一个表达式，表示参数作为数值参数传递 (§10.5.1.1)。
- 关键词 `ref` 之后加一个变量引用 (§**错误！未找到引用源。**)，指出参数作为引用参数传送 (§10.5.1.2)。一个变量必须在它能作为一个引用参数传送前被赋值 (§**错误！未找到引用源。**)。
- 关键词 `out` 之后加有一个变量引用 (§**错误！未找到引用源。**)，指出参数作为输出参数传送 (§10.5.1.2)。一个变量必须在它能作为一个引用参数传送前被赋值 (§**错误！未找到引用源。**)。

在运行函数成员调用过程 (§7.4.3)期间，引用一个参数列表的表达式被按顺序求值，从左到右，如下所示：

- 对于一个数值参数，参数表达式被求值并且隐式的转换 (§6.1)为相应的参数类型。结果数据变为函数成员调用中数值参数的初始值。
- 对于一个引用或输出参数，变量引用被求值，并且结果存储的位置变为在函数成员调用中的参数所表示的存储位置。如果变量引用为一个引用或输出参数是一个引用类型的数组元素，就会有一个运行时检查，来使得那个数组的元素类型与参数的类型是一致的。如果检查失败，就会抛出 `ArrayTypeMismatchException` 信息。

一个参数列表的表达式通常按他们书写的顺序求值。这样，例子

```
class Test
{
    static void F(int x, int y, int z) {
        Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
    }
}
```

产出输出

x = 0, y = 1, z = 2

数组的共同变化规则 (§12.5)允许一个数组类型 A[] 的数值变为对一个数组类型 B[] 的实例的引用，提供了一个从 B 到 A 的一个隐式引用转换。因为这项规则，当一个引用类型的数组元素被作为一个引用或输出参数传递时，需要进行运行时检查来数组的实际元素与那个参数的相同。在例子中

```
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
    }
}
```

第二个对 F 的引用使得抛出了一个 `ArrayTypeMismatchException` 信息，因为 b 的实际元素类型是字符串而不是对象。

#### 7.4.2 重载分析

重载分析是一种用来从参数列表和候选函数成员集中选择最好的函数成员来调用的机制。在下面的关于 C# 的文字中，重载分析选择函数成员来调用：

- 方法的调用，在一个调用表达式中命名 (§7.5.5)。
- 构造函数的调用，在对象创建表达式中命名 (§7.5.10.1)。
- 索引访问符的调用通过一个元素访问 (§7.5.6)。
- 预定义或用户定义操作符的调用，在表达式中引用 (§7.2.3 和 §7.2.4)。

这些上下文的每一个都以自己的方法定义了一个候选函数成员集和参数列表。然而，一旦候选函数成员和参数列表被确定，对于最优函数成员的选择在所有情况下都是相同的：

- 首先，候选函数成员集被简化到只包括用类提供参数列表 (§7.4.2.1) 的函数成员。如果被简化的集合是空的，就会产生错误。
- 这样，所给出的可用候选函数成员，而集合中的最优函数成员被确定了。如果集合中只包含一个函数成员，那么这个函数成员就是最优函数成员。另外，最优函数成员是一个在考虑所给的参数列表时比其它所有函数成员都要好的函数成员，通过使用 §7.4.2.2 中的规则把每个函数成员于其它所有函数成员比较得出。如果这里不只是一个函数成员比其它所有的函数成员要好，那么函数成员调用就是不确定的并且会产生错误。

下面的章节定义了可使用函数成员和更好的函数成员的确切含义。

#### 7.4.2.1 可使用函数成员

一个函数成员在下面所有都为真的时候，就被认为是考虑参数列表的可使用函数成员：

- A 中的参数数量是于函数成员声明中的参数数量相同。
- 对于 A 中的每个参数，参数的传输模式与相应的参数的参数传输模式是相同的，并且
  - 对于一个输入参数，从参数类型到相应的参数类型的隐式的转换是存在的，或
  - 对于一个 ref 或 out 参数，参数的类型被指定为相应参数的类型。

#### 7.4.2.2 更好的函数成员

给出一个参数列表 A，其中有参数类型集  $A_1, A_2, \dots, A_N$  和两个有参数类型  $P_1, P_2, \dots, P_N$  和  $Q_1, Q_2, \dots, Q_N$  的可使用函数成员  $M_P$  和  $M_Q$ ，如果  $M_P$  定义为比  $M_Q$  更好的函数成员

- 对于每个参数，从  $A_x$  到  $P_x$  的隐式转换不会比从  $A_x$  到  $Q_x$  的隐式转换更糟，并且
- 至少有一个参数，从  $A_x$  到  $P_x$  的转换比从  $A_x$  到  $Q_x$  的转换要好。

#### 7.4.2.3 更好的转换

给出一个隐式转换  $C_1$ ，它从类型 S 转换为类型  $T_1$ ，和一个隐式转换  $C_2$  它从类型 S 转换到类型  $T_2$ ，两个转换中更好的转换由下面决定：

- 如果  $T_1$  和  $T_2$  是相同的类型，没有哪种转换更好。
- 如果 S 是  $T_1$ ， $C_1$  是更好的转换。
- 如果 S 是  $T_2$ ， $C_2$  是更好的转换。
- 如果从  $T_1$  到  $T_2$  的隐式转换存在，但没有从  $T_2$  到  $T_1$  的隐式转换存在，那么  $C_1$  是更好的转换。
- 如果从  $T_2$  到  $T_1$  的隐式转换存在，但没有从  $T_1$  到  $T_2$  的隐式转换存在，那么  $C_2$  是更好的转换。
- 如果  $T_1$  是 sbyte 而  $T_2$  是 byte、ushort、uint 或 ulong， $C_1$  是更好的转换。
- 如果  $T_2$  是 sbyte 而  $T_1$  是 byte、ushort、uint 或 ulong， $C_2$  是更好的转换。
- 如果  $T_1$  是 short 而  $T_2$  是 ushort、uint 或 ulong， $C_1$  是更好的转换。
- 如果  $T_2$  是 short 而  $T_1$  是 ushort、uint 或 ulong， $C_2$  是更好的转换。
- 如果  $T_1$  是 int 而  $T_2$  是 uint 或 ulong， $C_1$  是更好的转换。
- 如果  $T_2$  是 int 而  $T_1$  是 uint 或 ulong， $C_2$  是更好的转换。
- 如果  $T_1$  是 long 而  $T_2$  是 ulong， $C_1$  是更好的转换。
- 如果  $T_2$  是 long 而  $T_1$  是 ulong， $C_2$  是更好的转换。
- 另外，没有转换是更好的。

如果按这种规则定义的隐式转换  $C_1$  是比隐式转换  $C_2$  更好的转换，那么也有  $C_2$  比  $C_1$  更差的情况。

### 7.4.3 功能成员调用

这节中描述在运行时调用一个特殊的功能成员发生的过程。假设一个编译时的过程已经决定要调用的特殊成员，很可能是通过对候选功能成员集使用重载分析来实现。

出于描述引用过程的目的，函数成员被分为两种：

- 静态函数成员。这里有静态方法、构造函数、静态属性访问符和用户定义的操作符。静态函数成员通常是非虚的。
- 实例函数成员。这里有实例方法、实例属性访问符和索引访问符。实例函数成员可以是非虚拟或虚拟，并且通常是在一共特殊实例中引用。这个实例由一个实例表达式计算，并且它在功能成员中变的可访问，就像 `this`(§7.5.7)。

函数成员的运行时过程由下面的步骤组成，这里 `M` 是函数成员，而如果 `M` 是一个实例成员，`E` 是实例表达式：

- 如果 `M` 是一共静态函数成员：
  - 参数列表就像§7.4.1中描述一样被求值。
  - `M` 被调用。
- 如果 `M` 是一个在数值变量中声明的实例函数成员：
  - `E` 被求值。如果这个求值引起了一个例外，那么就不需要执行以后的步骤。
  - 如果 `E` 没有被划分为变量，那么 `E` 类型的暂时局部变量将被创建，并且 `E` 的数值分配给那个变量。于是 `E` 被重新分类为对那个暂时局部变量的引用。这个暂时变量是可访问的就像 `M` 中的 `this`，但是不能用其它任何方法。这样，只有当 `E` 是一个真正的变量时，调用者才有可能观察 `M` 对 `this` 所做的改变。
  - 参数列表就像§7.4.1中描述一样被求值。
  - `M` 被调用。被 `E` 引用的变量变为被 `this` 引用的变量。
- 如果 `M` 是一个在引用类型中声明的实例函数成员：
  - `E` 被求值。如果这个求值引起了一个例外，那么就不需要执行以后的步骤。
  - 参数列表就像§7.4.1中描述一样被求值。
  - `E` 的类型是数值类型，一个打包转换(§错误！未找到引用源。)被实现来把 `E` 转换为 `object` 类型，而 `E` 在后面的步骤中被认为是 `object` 类型。
  - `E` 的值被检查为有效的。如果 `E` 的数值为 `null`，就会抛出一个 `NullReferenceException` 例外，而后面的步骤将不会执行。
  - 用来调用的函数成员执行由下面决定：如果 `M` 是一个非虚拟的函数成员，那么 `M` 是用来调用的函数成员执行。另外，`M` 是一个虚拟函数成员而用来调用的函数成员执行被通过虚拟函数成员查询(§7.4.4)或接口函数成员查询(§7.4.5)来决定。
  - 上面步骤中确定的函数成员执行被调用。被 `E` 引用的对象变为被 `this` 引用的对象。

#### 7.4.3.1 被包装实例的调用

在下面的情况中，一个以数值类型被执行的函数成员，可以通过那个数值类型的被包装实例来引用：

- 当函数成员是一个从类型 `object` 继承的方法的替代，并且是通过一个类型 `object` 的实例表达式调用的时。
- 当函数成员是一个接口函数成员的执行，并且是通过一个接口类型的实例表达式调用的时。
- 当函数成员通过一个代表被调用时。

在这些情况中，被包装实例被认为是包含了数值类型的变量，并且这种变量变成函数成员调用中被 `this` 引用的变量。这特别说明当一个函数成员在被包装实例中被调用时，就允许函数成员修改包含在被包装实例中的数值。

#### 7.4.4 虚拟函数成员查找

##### 问题

我们需要编写这节。

#### 7.4.5 接口函数成员查找

##### 问题

我们需要编写这节。

### 7.5 主要的表达式

*primary-expression:*  
*literal*  
*simple-name*  
*parenthesized-expression*  
*member-access*  
*invocation-expression*  
*element-access*  
*this-access*  
*base-access*  
*post-increment-expression*  
*post-decrement-expression*  
*new-expression*  
*typeof-expression*  
*sizeof-expression*  
*checked-expression*  
*unchecked-expression*

#### 7.5.1 文字

包含一个文字 (§ 错误！未找到引用源。 ) 的主要表达式被划分为数据。数据的类型根据下面的文字确定：

- 二进制文字是类型 `bool`。这里有两个可能的二进制文字 `true` 和 `false`。
- 整数文字是类型 `int`、`uint`、`long` 或 `ulong`，由文字的数值确定并且类型后缀 (§2.5.3.2) 可有可无。
- 实文字是类型 `float`、`double` 或 `decimal`，由文字的数值确定并且类型后缀 (§2.5.3.2) 可有可无。
- 字符文字是类型 `char`。

- 字符串文字是类型字符串。
- null 文字是类型 null。

### 7.5.2 简单名称

一个简单名称包含单个标识符。

*simple-name:*  
*identifier*

一个简单名称按下面进行求值和分类：

- 如果简单名称出现在块内，并且如果块中包含有给定名称的局部变量或参数，那么简单名称就是引用那个局部变量或参数，并且被分类为变量。
- 另外，对于每种类型 T，由立即嵌套类、结构或枚举声明开始，后面跟着每个嵌套的外部类或结构声明（如果有），如果 T 中的一个简单名称成员查询符合：
  - 如果 T 是立即嵌套类或结构类型并且查询确定一个或多个方法，结果是有一个方法组和一个相关的 this 的实例表达式。
  - 如果 T 是立即嵌套类或结构类型，如果查询确定一个实例成员，并且如果引用发生在一个构造函数、一个实例方法或一个实例访问符的程序体中，结果与形式为 this.E 的成员访问 (§7.5.4) 相同，这里 E 是简单名称。
  - 另外，结果与形式为 T.E 的成员访问 (§7.5.4) 相同，这里 E 是简单名称。在这种情况下，简单名称引用一个实例成员是个错误。
- 另外，从简单名称发生（如果有）的名称空间声明开始，跟着每个嵌套名称空间声明（如果有），并且用全局名称空间结束，下面的步骤会一直被求值，直到实体位置确定：
  - 如果名称空间包含一个有给定名称的名称空间成员，那么简单名称指向那个成员，并且根据成员作为名称空间或类型被分类。
  - 另外，如果名称空间声明包含把给定名称与输入名称空间或类型相关联的使用别名指示，那么简单名称引用那个名称空间或类型。
  - 另外，如果被名称空间声明的使用名称空间指示引入的名称空间包含有给定名称的一个类型，那么简单名称引用那个类型。
  - 另外，如果被名称空间声明的使用名称空间指示引入的名称空间包含有给定名称的超过一个类型，那么简单名称是不确定的并且会产生错误。
- 另外，由简单名称给出的名称是未定义的，并且会发生错误。

#### 7.5.2.1 块中的不变量意义

每次出现把一个所给的标识符作为表达式中的简单名称的事件时，任何其它与立即嵌套块 (§**错误！未找到引用源。**) 或转换块 (§**错误！未找到引用源。**) 中表达式的简单名称相同的标识符都必须引用相同的实体这项规则保证一个表达式上下文中名称的意义在一个块中是相同的。

例子

```
class Test
{
    double x;
```

```

    void F(bool b) {
        x = 1.0;
        if (b) {
            int x = 1;
        }
    }
}

```

是错误的，因为 `x` 在外部块中（在 `if` 语句中包含嵌套块的扩展）引用了不同的实体。相反，例子

```

class Test
{
    double x;
    void F(bool b) {
        if (b) {
            x = 1.0;
        }
        else {
            int x = 1;
        }
    }
}

```

是允许的，因为名称 `x` 在外部块中永远不会使用。

注意，不变量含义的角色指示应用于简单名称。一个标识符有一个作为简单名称的含义而另一个含义作为一个成员访问的右操作数 (§7.5.4)。例如：

```

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

```

上面的例子演示了在构造函数中作为参数名称的域和普通式样。在例子中，简单名称 `x` 和 `y` 引用参数，但是没有通过访问域来防止成员访问表达式 `this.x` 和 `this.y`。

### 7.5.3 加括号的表达式

一个加括号的表达式由一个附上一个括号的表达式组成。

*parenthesized-expression:*  
 ( *expression* )

一个被加上括号的表达式通过对括号中的表达式进行求值来求值。如果括号中的表达式表示一个名称空间、类型或方法组，会产生错误。另外，加括号的表达式的结果是对包含的表达式求值的结果。

### 7.5.4 成员访问

一个成员访问由主要表达式或预定义类型，跟着一个“.”符号，再跟着一个标识符来构成。

*member-access:*  
*primary-expression* . *identifier*  
*predefined-type* . *identifier*



*predefined-type:* one of

bool	byte	char	decimal	double	float	int	long
object	sbyte	short	string	uint	ulong	ushort	

一个形式为 E.I 的成员访问是一个标识符，这里 E 是一个主要的表达式或预定义的类型，它如下被求值和分类：

- 如果 E 是一个名称空间而 I 是那个名称空间的可访问成员的名称，那么结果是那个成员，并且根据这个成员被分类为一个名称空间或一个类型。
- 如果 E 是一个预定义类型或一个分类为类型主要类型，而且 E 中 I 的成员查询 (§7.3) 产生了一个匹配，那么 E.I 被如下求值和分类：
  - 如果 I 确定一个类型，那么结果就是那种类型。
  - 如果 I 确定一个或多个方法，那么结果是一个没有相关实例表达式的方法组。
  - 如果 I 确定了一个静态属性，那么结果是一个没有相关实例表达式的属性访问。
  - 如果 I 确定了一个静态域：
    - 如果域是只读的，并且引用在声明了域的类或结构的静态构造函数外面发生，那么结果是一个数值，也就是 E 中静态域 I 的数值。
    - 另外，结果是变量，也就是 E 中的静态域 I。
  - 如果 I 确定了一个静态事件：
    - 如果引用发生于事件被声明的类或域中，那么 E.I 就像 I 是一个静态域或属性一样被处理。
    - 另外，结果是一个没有相关实例表达式的事件访问。
  - 如果 I 确定一个常数，那么结果是数值，也就是那个常数的数值。
  - 如果 I 确定一个枚举成员，那么结果是数值，也就是那个枚举成员的数值。
  - 另外，E.I 是一个无效的成员引用，这样就会发生错误。
- 如果 E 是属性访问、索引访问、变量或数值，它的类型是 T，并且 I 的一个成员查询 (§7.3) 产生一个匹配，那么 E.I 按下面被求值和分类：
  - 首先，如果 E 是一个属性或者索引访问，那么属性或索引访问的数值就被获得 (§7.1.1)，并且 E 被重新分类为一个数值。
  - 如果 I 确定一个或多个方法，那么结果是一个与 E 的实例表达式相关的方法组。
  - 如果 I 确定一个实例属性，那么结果是一个与 E 的实例表达式相关的属性访问。
  - 如果 T 是一个类类型，并且 I 指定了那个类类型的一共实例域：
    - 如果 E 的数值是 null，那么会抛出一个 `NullReferenceException` 异常。
    - 另外，如果域是只读，并且引用在声明了域的类的实例构造函数外发生，那么结果是一个数值，也就是被 E 引用的变量中的域 I 的值。
    - 另外，结果是一个变量，也就是被 E 引用的对象中的域 I
  - 如果 T 是一共结构类型，并且 I 指定了那个结构类型的实例域：

- 如果 E 是一个数值，或者如果域是只读，并且引用发生在域被声明的结构的实例构造函数外面，那么结构是一共数值，也就是由 E 给出的结构实例中的域 I 的数值。
- 另外，结果是一个变量，也就是由 E 给出的实例结构中的域 I。
- 如果 I 确定了一个实例事件：
  - 如果引用发生在声明了事件的类或结构的外面，那么 E.I 就像 I 是一共实例域或属性一样被处理。
  - 另外，结果是一个事件访问，并且与 E 的实例表达式相关。
- 另外，E.I 是一个无效的成员引用，并且会发生错误。

#### 7.5.4.1 同样的简单名称和类型名称

在形式 E.I 的成员访问中，如果 E 是一个简单标识符，并且如果 E 的作为一个简单名称 (§7.5.2) 的意义是个常数、域、属性、局部变量或参数，与 E 作为一个类型名称 (§3.6) 的意义相同，那么 E 的两个可能的意义都是允许的。由于在所有情况 I 都必须是 E 的成员，因此 E.I 的两个可能的含义不会是不明确的。换句话说，如果规则完全允许对 E 的静态成员的访问，这里就会发生一个错误。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
    }
}
```

在类 A 中，那些引用 Color 类型的 Color 标识符的出现是被强调的，而那些引用 Color 域的不被强调。

#### 7.5.5 调用表达式

一个调用表达式用来调用一个方法。

*invocation-expression:*  
*primary-expression* ( *argument-list<sub>opt</sub>* )

调用表达式的主要表达式必须是一个代表类型的方法组或数值。如果主表达式是一个方法组，调用表达式就是一个方法调用 (§7.5.5.1)。如果主表达式是一个代表类型的数值，那么调用表达式就是一个代表调用 (§7.5.5.2)。如果主表达式既不是方法组也不是代表类型的数值，会产生错误。

可选的参数列表提供了数值或对方法的参数的引用。

对调用表达式求值的结果按下面进行分类：

- 如果调用表达式调用了一个返回 void 的方法或代表，结果就是空。一个被分类为空的表达式不能是任何操作符的操作数，并且它只被允许存在于语句声明表达式的上下文中(§**错误！未找到引用源。**)。
- 另外，结果是被方法或代表返回的类型的数值。

#### 7.5.5.1 方法调用

对于一个方法调用调用表达式的主表达式必须是一个方法组。这个方法组确定一个要调用的方法或一系列重载方法，从当中选择一个特殊的方法来调用。在后面的情况中，要调用的特殊方法的确定要基于由参数列表中的参数的类型提供的上下文。

编译时形式 M(A)的方法调用过程由下面步骤组成：（这里 M 是一个方法组，而 A 是可选的参数列表）

- 方法调用的候选方法集合被构造。开始于与 M 相关的方法集合，它可由前面的成员查询创建 (§7.3)，这个集合被缩小到只包括那些与参数列表 A 相关的可用方法。集合的缩小包括把下面的规则使用于集合中的每个方法 T.N，这里 T 是在声明了方法 N 的类型：
  - 如果 N 与 A (§7.4.2.1) 不相关，那么 N 被从集合中去掉。
  - 如果 N 与 A (§7.4.2.1) 相关，那么在 T 的基本类型中声明的方法被从集合中去掉。
- 如果最后的候选方法集合是空的，那么就没有可用的方法存在，并且产生一个错误。如果候选方法并不是全都在相同的类型中声明，方法调用就是不明确的，会产生一个错误（这后面的一个情况只可能在有多个直接基接口的接口中一个方法的调用是发生，就像 §13.2.5 中所描述的）。
- 候选集合中最好的方法是使用 §7.4.2 中的重载分析规则来确定。如果不能确定单个最好的方法，那么方法调用是不明确的，并且会产生错误。
- 给定一个最好的方法，方法的调用在在方法组的上下文中是有效的：如果最好的方法是一个静态方法，方法组必须通过一种类型从简单名称或方法访问得到结果。如果最好的方法是实例方法，方法组必须从简单名称，通过一个变量或数值从成员访问或基访问来获得结果。如果这项需求都不能满足，就会产生一个编译时错误。

一旦一个方法已经被选择并且在运行时通过上面的步骤生效，实际运行时调用根据在 §7.4.3 中描述的功能成员调用的规则来进行。

在上面描述的分析规则的直觉效果像下面一样：为了确定被方法调用所调用的方法的位置，从被方法调用指出的类型开始，并且沿着继承链向上找，直到至少找到一个在那个类型中声明的可使用、可访问非覆盖的方法，然后调用这样选择的方法。

#### 7.5.5.2 代表调用

对于代表调用，调用表达式的基本表达式必须是一个代表类型的数值。而且，把代表类型看作有相同参数列表的功能成员，此代表类型必须可用于调用表达式的参数列表 (§7.4.2.1)。

形式 D(A)的代表调用的运行时过程由下面的步骤组成：（这里 D 是一个代表类型的基本表达式，而 A 是一个可选的参数列表。）

- D 被求值，如果这个求值造成了一个例外，就不会进行下面的步骤。
- D 数值被检查为有效的。如果 D 是 null，就会抛出一个 `NullReferenceException` 并且不会进行下面的步骤。

- 另外，D 指向一个代表实例。一个功能成员调用 (§7.4.3) 在被代表引用的方法中实现。如果方法是一个实例方法，调用的实例就变为被代表引用的实例。

### 7.5.6 元素访问

一个元素访问由几个部分组成：一个基本表达式，跟着一个“[”符号，跟着一个表达式列表，跟着一个“]”符号。表达式列表由一个或多个表达式构成，用逗号分开。

```

element-access:
    primary-expression [ expression-list ]

expression-list:
    expression
    expression-list , expression

```

如果元素访问的基本表达式是一个数组类型的数值，那么元素访问是数组访问 (§7.5.6.1)。另外，基本表达式必须是一个有一个或多个索引成员的类、结构或接口类型的变量或数值，并且元素访问就是一个索引访问 (§7.5.6.2)。

#### 7.5.6.1 数组访问

对于一个数组访问，元素访问的基本表达式必须是一个数组类型的数值。在表达式列表中的表达式的成员必须与数组类型的维数相同，而每个表达式必须是 `int` 类型或是一个可以隐式地转换为 `int` 的类型。

对一个数组访问求值的结果是这个数组的元素类型的变量，也就是在表达式列表中表达式的数值选择的数组成员。

形式 `P[A]` 的一个数组访问的运行时过程由下面的步骤组成：（这里 `P` 是数组类型的基本表达式而 `A` 是一个表达式列表。）

- `P` 被求值。如果这个求值造成了一个异常，下面的步骤就不会进行了。
- 表达式列表的索引表达式按从左到右的顺序求值。在对每个索引表达式求值后，会实现一个到类型 `int` 的隐式转换。如果一个索引表达式的求值或后来的隐式转换 (§6.1) 造成了一个异常，那么不会对更多的索引表达式进行求值，而且不会执行下面的步骤。
- `P` 的数值被检查为有效的。如果 `P` 的数值是 `null`，就会抛出一个 `NullReferenceException` 异常，并且不会继续执行下面的步骤。
- 表达式列表中的每个表达式的数值都要被检查，来确定被 `P` 引用的数组的每个维数都没有超过实际边界。如果一个或多个数值超出边界，就会抛出一个 `IndexOutOfRangeException` 异常，并且不会继续执行下面的步骤。
- 由索引表达式给出的数组元素位置是计算出来的，并且这个位置成为数组访问的结果。

#### 7.5.6.2 索引访问

对于一个索引访问，元素访问的基本表达式必须是一个类、结构或接口类型的变量和数值，并且这个类型必须执行一个或多个可用于元素访问的表达式列表的索引。

形式 `P[A]` 的索引访问的编译时过程由下面的步骤组成：（这里 `P` 是一个类、结构或接口类型 `T` 的基本表达式，而 `A` 是一个表达式列表）

- 构造由 `T` 提供的索引集合。这个集合由所有在 `T` 中声明的索引，或 `T` 中在当前的上下文 (§3.3) 中可访问而且不是隐藏的声明的类型组成。

- 这个集合被缩小到只包括那些可用的并且没有被其他索引隐藏的索引。下面的规则被应用于集合中的每个索引 S.I，这里 S 是一各声明了索引 I 的类型：
  - 如果 I 不能用于 A (§7.4.2.1)，那么 I 被从集合中去掉。
  - 如果 I 可用于 A (§7.4.2.1)，那么所有在基本类型 S 中声明的索引都被从这个集合中去掉。
- 如果最后的候选索引的集合为空，那么就不会有可用索引存在，并且会产生一个错误。如果候选索引并不都在同一个类型中声明，索引访问就是不确定的，并且会产生错误。（后面的这种情况只会在一个索引访问有多个直接基接口的接口的实例时发生。）
- 候选索引集合中最优索引使用§7.4.2中的重载分析规则来确定。如果不能确定一个最优的索引，索引访问就是不确定的，并且会发生错误。
- 索引访问执行的结果是一个被作为索引访问分类的表达式。索引访问表达式引用在前面步骤确定的索引，并且有一个相关的 P 的实例表达式和一个相关的 A 的参数列表。

根据所使用的上下文，一个索引访问引起对或者索引的 get 访问符或者 set 访问符的调用。如果索引访问世一个赋值得目标，会调用 set 访问符来赋新值(§**错误！未找到引用源。**)。在所有其他情况，调用 get 访问符来获得当前数值(§7.1.1)。

### 7.5.6.3 字符串索引

字符串类实现了一个索引，它允许一个字符串的单个字符被访问。这个字符串类的索引按下面来声明：

```
public char this[int index] { get; }
```

换句话说，一个只读的索引诱一个类型为 int 的参数，并且会返回一个 char 类型的元素。为了索引参数传递的数值必须大于或等于零，并且要小于字符串的长度。

## 7.5.7 This 访问

this 访问由保留字 this 构成。

*this-access:*  
this

只在构造函数、实例方法或实例访问符的程序体中允许 this 访问。它下面的意义中的一种：

- 当 this 用于一个类的构造函数中基本表达式中的时候，它被分类为数值。数据的类型是引用发生的类，而数据为对被构造的对象的引用。
- 当 this 被用于一个类中的实例方法或实例访问符的一个基本表达式中，它被分类为一个数据。数据的类型为引用发生的类，而这个数据是对一个对象的引用，为这个对象，一个方法或访问符被引用。
- 当 this 被用在一个结构的构造函数的基本表达式中，它被分类为一个变量。变量的类型是一个结构，在这个结构中这个引用发生，并且变量所代表的结构被构造。一个结构的构造函数的变量 this 与结构类型的 out 参数相同，这说明变量必须通过构造函数的每个执行途径明确赋值。
- 当 this 用于一个结构的实例方法或实例访问符中的基本表达式中，它作为变量被分类。变量的类型是发生了引用的结构。并且变量代表这个结构，为了这个结构方法和访问符被调用。一个结构的实例方法的行为与结构类型的 ref 参数相同。

在一个不同于前面列出的上下文中的基本表达式中使用 this 是一个错误。特别，在一个静态方法、静态属性访问符或一个域声明的变量初始化函数中不可能使用 this。

### 7.5.8 基本访问

一个基本访问由以下组成：一个保留字 `base`，或者跟一个符号“.”和一个标识符，或者跟一个用方括号圈起来的表达式列表：

```
base-access:
    base . identifier
    base [ expression-list ]
```

基本访问被用于访问基本的类成员，这些类成员载当前类或者结构中被同样名称的成员所隐藏。一个基本访问只在一个构造函数、一个实例方法或一个实例访问符的程序体中被允许使用。当在一个类或结构中 `base.I` 发生时，`I` 必须表示那个类或类型的基本类型的成员。同样，当一个类中 `base[E]` 发生时，可用的索引必须存在于基本类中。

在编译时，形式 `base.I` 和 `base[E]` 的基本访问就像它们被写成 `((B)this).I` 和 `((B)this)[E]` 一样被求值，这里 `B` 是发生构造的类或结构的基本类。因此，除了 `this` 被视为基类的一个实例外，`base.I` 和 `base[E]` 与 `this.I` 和 `this[E]` 相符合。

当一个基本访问引用一个功能成员（一个方法、属性和索引），对于功能调用 (§7.4.3) 来说，这个功能成员被认为是非虚拟的。因此，在一个虚拟功能成员的覆盖中，一个基本访问可以被用来调用功能成员的继承执行。如果被基本访问引用的功能成员是抽象的，会产生错误。

### 7.5.9 递增和递减后缀操作符

```
post-increment-expression:
    primary-expression ++

post-decrement-expression:
    primary-expression --
```

增加和减少后缀操作的操作数必须是一个分类为变量、属性访问或索引访问的表达式。操作的结果是与操作数同类型的数据。

如果增加或减少后缀操作的操作数是一个属性或者索引访问，这个属性或者索引必须有 `get` 和 `set` 访问符。如果这点没有实现，会产生一个编译时错误。

一元操作符重载分析 (§7.2.3) 被用来选择特殊的操作符实现。预定义 `++` 和 `--` 操作符可以用于下面的类型：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 和任何枚举类型。预定义的 `++` 操作符返回一个把参数加一产生的数据，而预定义的 `--` 操作符返回一个把参数减一产生的数据。

形式 `x++` 或 `x--` 的增加或减少后缀操作的运行时过程由下面的步骤组成：

- 如果 `x` 被作为变量分类：
  - `x` 被求值以产生变量。
  - `x` 的数据被存储。
  - 被选择的操作符被调用，有存储的数据作为它的参数。
  - 操作符返回的数据存储在由 `x` 的求值提供的位置。
  - `x` 的存储的数据变为操作的结果。
- 如果 `x` 被作为属性或者索引访问分类：

- 与  $x$  相关的实例表达式（如果  $x$  不是静态的）和参数列表（如果  $x$  是一个索引访问）被求值，其结果用于后来的 `get` 和 `set` 访问符调用。
- $x$  的 `get` 访问符被调用，并且返回值被存储。
- 被选择的操作符被调用，有  $x$  的存储的数据作为它的参数。
- $x$  的 `set` 访问符被调用，被这个操作符返回的数据作为它的数值参数。
- $x$  的被存储的数据变为操作的结果。

像§错误！未找到引用源。中所描述的，`++`和`-`操作符也支持前缀符合。`x++`或`x-`的结果是  $x$  在操作前的数据，然而，`++x` 或`-x` 的结果是  $x$  在操作后的数据。在任何一种情况， $x$  本身在操作后有相同的数值。

一个操作符`++`或操作符`-`的执行可以通过或者是前缀或者是后缀符号来调用。对两种符合都可以有分立操作符执行。

### 7.5.10 new 操作符

`new` 操作符被用来创建类型的新实例。

*new-expression:*  
*object-creation-expression*  
*array-creation-expression*  
*delegate-creation-expression*

这里有三种形式的 `new` 表达式：

- 对象创建表达式被用来创建类类型和数值类型的新实例。
- 数组创建表达式被用来创建数组类型的新实例。
- 代表创建表达式被用来创建代表类型的新实例。

`new` 操作符暗示了一种类型的一个实例的创建，但是没有必有暗示内存的动态分配。特别是，数值类型的实例不需要超出它们所处的变量的附加内存，并且当 `new` 被用来创建数值类型的实例时，不会发生动态分配。

#### 7.5.10.1 对象创建表达式

一个对象创建表达式被用来创建一个类类型或数据类型的新实例。

*object-creation-expression:*  
`new type ( argument-listopt )`

对象创建表达式必须是一个类类型或数值类型。类型不能是抽象类类型。

可选的参数列表 (§7.4.1)只有在类型是累类型或一个结构类型的时候才被允许。

形式 `new T(A)`的对象创建表达式的编译时过程由下面步骤组成：（这里  $T$  是一个类类型或数值类型，而  $A$  是一个可选的参数列表）

- 如果  $T$  是一个数值类型而  $A$  没有出现：
  - 对象创建表达式是一个默认得构造函数调用。对象创建表达式的结果是类型  $T$  的数据，也就是 §4.1.1中定义的  $T$  的默认数值。
- 另外，如果  $T$  是一个类类型或结构类型：

- 如果 T 是一个抽象类类型，会发生一个错误。
- 用于调用的构造函数通过使用§7.4.2中的重载分析规则来调用。候选构造函数集合由所有在 T 中声明的可访问构造函数组成。如果候选构造函数集合是空的，或如果不能确定一个单独的最优的构造函数，会发生错误。
- 对象创建表达式的结果是一个类型 T 的数据，也就是通过调用构造函数产生的数据在上面的步骤中确定。
- 另外，对象创建表达式如果是无效的，会发生一个错误。

形式 `new T(A)` 的对象创建表达式的运行时过程由下面的步骤组成：（这里 T 是类类型和结构类型而 A 是一个可选参数列表）

- 如果 T 是一个类类型：
  - 类 T 的一个新实例被分配。如果这里没有足够可用的内存来分配新的实例，会抛出一个 `OutOfMemoryException` 异常，并且不会执行下面的步骤。
  - 新实例的所有域被初始化为它们默认得数值 (§错误！未找到引用源。 )。
  - 根据功能成员调用 (§7.4.3) 的规则调用构造函数。一个新分配实例的引用自动被传送到构造函数并且实例可以通过构造函数 `this` 进行访问。
- 如果 T 是一个结构类型：
  - 一个类型 T 的实例通过分配一个暂时局部变量来创建。由于结构类型的构造函数需要被明确地给每个创建实例的域进行赋值，因此不需要暂时变量的初始化。
  - 根据功能成员调用 (§7.4.3) 的规则调用构造函数。一个新分配实例的引用自动被传送到构造函数并且实例可以通过构造函数 `this` 进行访问。

#### 7.5.10.2 数组创建表达式

一个数组创建表达式被用来创建数组类型的新实例。

*array-creation-expression:*

```
new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt
new array-type array-initializer
```

第一种形式的数组创建表达式分配了这种类型的一个数组实例，这个类型是通过删除每个表达式列表中地独立表达式而得到的结果。例如，数组创建表达式 `new int[10, 20]` 生成了一个类型为 `int[,]` 的数组实例，而数组创建表达式 `int[10][,]` 生成了一个类型为 `int[][,]` 的数组。表达式列表中的每个表达式必须是 `int` 类型或者是一个可以隐式地转换为 `int` 类型的类型。每个表达式的数值决定了新分配的数组实例相应维数的长度。

如果一个第一种形式的数组创建表达式包括一个数组初始化程序，表达式列表中的每个表达式就必须是一个常数，并且由表达式列表确定的秩和数组长度必须与数组初始化程序相匹配。

再第二种形式的数组创建表达式中，由数组类型确定的秩必须与数组初始化函数相匹配。每一维的长度可以从每个数值初始化函数的相应的嵌套级别中的基本数量来推算。这样，表达式

```
new int[,] {{0, 1}, {2, 3}, {4, 5}};
```

与下面列出的一致

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}};
```



数组初始化程序将在§12.6中描述。

对一个数组创建表达式求值的结果作为数值被分类，也就是一个对新分配数组实例的引用。数组创建表达式的运行时过程由下面的步骤组成：

- 表达式列表中的某维长度表达式按照从左到右的顺序被求值。在对每个表达式求值后，会执行一个隐式地转换(§6.1)，转换为 int 类型。如果对一个表达式的求值或后来的隐式转换造成了一个异常，那么就不会对其他表达式进行求值，并且不会进行下面的步骤。
- 为某维长度计算的数值被生效。如果这些数值中的一个或多个小于零，就会抛出一个 `IndexOutOfRangeException` 异常，并且不会进行下面的步骤。
- 有给定维长度的数组实例被分配。如果没有足够的内存来分配给新的实例，就会抛出一个 `OutOfMemoryException` 异常，并且不会进行下面的步骤。
- 新数组实例的所有元素被初始化为它们的默认数值(§错误！未找到引用源。 )。
- 如果数组创建表达式包括一个数组初始化函数，那么数组初始化函数中的每一个表达式都要被求值，并且把值赋给与它相应的数组元素。求值和赋值按照在数组初始化函数中书写的顺序被执行，换句话说，元素按照增加的索引顺序被初始化，最右边的维最先增加。如果给定的表达式求值或后面对相应数组元素的赋值造成了一个例外，那么就不会初始化其他元素（并且剩下的元素将保持他们默认得数值）。

一个数组创建表达式允许用一个数组类型的元素实例化一个数组，但是那样的一个数组的元素必须手动初始化。例如，语句

```
int[][] a = new int[100][];
```

创建了一个一维数组，有 100 个 `int[]` 类型的元素。每个元素的初始值为 `null`。相同的数组创建表达式也例子数组是不可能的，而语句

```
int[][] a = new int[100][5]; // Error
```

是一个错误。子数组的例示必须变为手动实现，如下。

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

当数组有“矩形”形状的时候，也就是子数组都有相同的长度时，使用多维数组更为有效。在上面的例子里，数组的实例化创建 101 个对象 - 一个外部数组组合 100 个子数组。相反，

```
int[,] = new int[100, 5];
```

创建了一个单独的对象，一个两维数组并且在一个语句中完成了分配。

### 7.5.10.3 代表创建表达式

代表创建表达式用来创建代表类型的新实例。

```
delegate-creation-expression:
new delegate-type ( expression )
```

代表创建表达式的参数必须是一个方法组或是一个代表类型的数据。如果参数是个方法组，它为一个实例方法确定用于创建代表的方法和对象。如果参数是代表类型的数据，它就确定一个用于创建一个备份的代表实例。

形式 `new D(E)` 的代表创建表达式的编译时的过程由下面的步骤组成：（这里 D 是一个代表类型而 E 是一个表达式）

- 如果 E 是一个方法组：
  - 如果从基本访问得到一个方法组，会发生错误。
  - 由 E 确定的方法集合必须包括一个与 D 的签名和返回类型正好一致的方法，并且这成为新创建的代表所指定的方法。如果没有匹配的方法存在，或者如果存在多个匹配的方法，那么会产生错误。如果所选的方法是一个实例方法，与 E 相关的实例方法就决定了这个代表的目标对象。
  - 在方法调用中，所选的方法必须与方法组的上下文一致：如果方法是一个静态方法，方法组就必须通过类型从简单名称或成员访问获得结果。如果方法是一个实例方法，方法组必须通过变量或数据从简单名称或成员访问获得结果。如果所选择的方法与方法组的上下文不匹配，就会发生一个错误。
  - 如果结果是一个类型 D 的数据，也就是一个指向所选的方法和目标对象的新创建代表。
- 否则，如果 E 是一个代表类型的数据：
  - E 的代表类型必须有相同的签名并且返回的类型为 D，否则会有错误发生。
  - 如果结果是一个类型 D 的数据，也就是一个指向所选的方法和目标对象的新创建代表。
- 否则，对象创建表达式是无效的，并且会发生一个错误。

形式 `new D(E)` 的代表创建表达式的运行时过程由下面的步骤组成：（这里 D 是代表类型而 E 是一个表达式）

- 如果 E 是一个方法组：
  - 如果在编译时所选择的方法是一个静态方法，代表的目标对象就是 `null`。否则，所选择的方法是一个实例方法，并且代表的目标方法通过与 E 相关的实例表达式确定：
    - 实例表达式被求值。如果这个求值引起了一个异常，就不会进行下面的步骤。
    - 如果实例表达式是引用类型，由实例表达式计算的数值就成为目标对象。如果目标对象是 `null`，就会抛出一个 `NullReferenceException` 异常，并且不会执行下面的步骤。
    - 如果实例表达式是数值类型的，就会执行一个包装操作（§**错误！未找到引用源。**）来把数据转换为对象，并且这个对象变为目标对象。
  - 一个代表类型 D 新实例被分配。如果没有足够的内存来分配新实例，就会抛出一个 `OutOfMemoryException` 异常，并且不会执行下面的操作。
  - 新代表实例，通过一个对在编译时确定的方法的引用，和一个对前面计算的目标对象的引用被初始化。
- 如果 E 是一个代表类型的数据：
  - E 被求值。如果这个求值引起了一个异常，就不会执行下面的步骤。
  - 如果 E 的数值是 `null`，就会抛出一个 `NullReferenceException` 的异常，并且不会执行下面的步骤。
  - 一个代表类型 D 新实例被分配。如果没有足够的内存来分配新实例，就会抛出一个 `OutOfMemoryException` 异常，并且不会执行下面的操作。
  - 新代表类型通过对与 E 提供的代表实例相同的方法和对象的引用进行初始化。

当代表被实例化并且在代表的完全生命周期中都保持为常数时，代表所指向的方法和对象就被确定。换句话说，一旦被创建，代表的目标方法获函数就不能被更改。

不可能创建一个指向一个构造函数、属性、索引或用户定义操作符的代表。

就像上面所描述的，当从一个方法组创建一个代表，代表的签名和返回类型确定选择哪个可重载方法。在例子中

```
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;
    }
}
```

域 A.f 用一个指向第二个 Square 方法的代表初始化，因为那个方法与 DoubleFunc 的签名和返回类型相匹配。如果第二个 Square 方法不被实现，就会产生一个编译时错误。

### 7.5.11 typeof 操作符

对于一个类型，typeof 操作符被用来获得 System.Type 对象。

```
typeof-expression:
    typeof ( type )
```

对于候选类型，*typeof-expression* 的结果是 System.Type 对象。

例子

```
class Test
{
    static void Main() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[])
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i].Name);
        }
    }
}
```

产生下面的输出：

```
Int32
Int32
String
Double[]
```

注意 int 和 System.Int32 是相同的类型。

### 7.5.12 sizeof 操作符

#### 问题

我们需要编写这章。

*sizeof-expression:*  
 sizeof ( type )

### 7.5.13 checked 和 unchecked 操作符

checked 和 unchecked 操作符被用来为整数类型算术操作和转换控制溢出检查文字。

*checked-expression:*  
 checked ( expression )

*unchecked-expression:*  
 unchecked ( expression )

checked 操作符对选中的文字中包含的表达式进行求值，而 unchecked 操作符为未选中的文字中包含的表达式求值。选中表达式或未选中表达式都与括号内的表达式 (§7.5.3) 相关，除非在给定的溢出检查上下文中对包含的表达式进行求值。

溢出检查也可以通过 checked 和 unchecked 语句进行控制 (§**错误！未找到引用源。**)。

下面的操作被由 checked 和 unchecked 操作符和语句建立的溢出检查文字所影响：

- 预定义的++和--一元操作符 (§7.5.9和§**错误！未找到引用源。**)，当操作数是一个整数类型。
- 预定义的-一元操作符 (§**错误！未找到引用源。**)，当操作数是一个整数类型。
- 预定义的+、-、\*和/二元操作符 (§**错误！未找到引用源。**)，当两个操作数都是整数类型。
- 从一个整数类型到另外一个整数类型的外部数字转换 (§**错误！未找到引用源。**)。

当上面操作中的一个产生了一个结果，此结果由于太大而不能在目的类型中表示的时候，操作被执行的上下文控制结果的行为：

- 在一个 checked 文字中，如果操作是一个常数表达式 (§**错误！未找到引用源。**)，就会产生一个编译时错误。否则，当在运行时实现操作时，就会抛出一个 `OverflowException` 异常。
- 在一个 unchecked 中文字中，结果被通过去掉不符合目标类型的高端位进行删节。

当一个非常数表达式（一个在运行时求值得表达式）没有被加在任何 checked 或 unchecked 操作符或语句中时，运行时表达式求值的溢出的影响取决于外部因素（例如编译器切换和执行环境设置）。影响保证为或者是 checked 求值的或者是 unchecked 求值的。

对于常数表达式（可以完全在编译时求值的表达式），默认的溢出检查文字总是 checked。除非一个常数表达式明显地被放在一个 unchecked 上下文中，在编译时发生的表达式求值溢出通常造成一个编译时错误。

在例子中

```
class Test
{
    static int x = 1000000;
    static 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
    }
}

```

由于没有一个表达式可以在编译时被求值，所以不会报告编译时错误。在运行时，方法 F() 会抛出一个 `OverflowException` 异常，而方法 G() 会返回 -727379968（由于超出范围造成的低 32 位输出）。方法 H() 的行为要根据所编辑的默认溢出检查文字，但是它或者与 F() 相同或者与 G() 相同。

在例子中

```

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
    }
}

```

因为表达式在 `checked` 上下文中被求值，所以报告当对 F() 和 H() 中的常数表达式进行求值时发生的溢出造成了编译时错误。而在对 G() 中的常数表达式进行求值时也发生了溢出，但是由于求值是在 `unchecked` 上下文中发生，溢出就没有被报告。

`checked` 和 `unchecked` 操作符只影响与文字包含在“(”和“)”符合间的操作相关的溢出检查文字。这个操作符对于作为对包含的表达式求值的结果引用的功能成员没有影响。在这个例子中

```

class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}

```

F() 中所使用的 `checked` 不会影响在 `Multiply()` 中使用的 `x * y` 求值，而 `x * y` 因此在默认溢出检查文字中进行求值。

当用十六进制符合编写有符号整数类型常数时，`unchecked` 操作符是很方便的。例如：

```

class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);
    public const int HighBit = unchecked((int)0x80000000);
}

```

上面所有的十六进制常数都是 `uint` 类型的。因为常数超出来 `int` 的数值范围，如果没有 `unchecked` 操作符，对 `int` 的舍入将会产生编译时错误。

## 7.6 一元表达式

### 问题

我们需要编写这一节。

```
unary-expression:
    primary-expression
    + unary-expression
    - unary-expression
    ! unary-expression
    ~ unary-expression
    * unary-expression
    & unary-expression
    pre-increment-expression
    pre-decrement-expression
    cast-expression
```

### 7.6.1 一元正值运算符

对于 `+x` 形式的操作，一元运算符重载解析 (§7.2.3) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。预定义的一元正值运算符为：

```
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);
```

对于以上每个运算符，其结果就是操作数的值。

### 7.6.2 一元负值运算符

对于 `-x` 形式的操作，一元运算符重载解析 (§7.2.3) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。预定义的负值操作符为：

- 整数取负：

```
int operator -(int x);
long operator -(long x);
```

结果是 0 减去 `x` 的计算值。在 `checked` 环境中，如果 `x` 的值是最大的负 `int` 或 `long` 型整数，将产生 `OverflowException` 异常（上溢出异常）。在 `unchecked` 环境中，如果 `x` 的值是最大的负 `int` 或 `long` 型整数，结果是同一值，并且不报告有溢出。如果取负运算符的操作数是 `uint` 类型，它会被转换成 `long` 类型，而且结果的类型也为 `long`。有一个规则是例外，即允许 `int` 型值 `-2147483648` ( $-2^{31}$ ) 被写成十进制整数文字 (§2.5.3.2)。如果取负运算符的操作数是 `ulong` 类型，将产生错误。有一个规则是例外，即允许 `long` 型值 `-9223372036854775808` ( $-2^{63}$ ) 被写成十进制整数文字（错误！未找到引用源。）。)

- 浮点取负：

```
float operator -(float x);
double operator -(double x);
```

结果是符号反转的 `x` 值。如果 `x` 是 `NaN`，结果仍是 `NaN`。

- 小数取负：

```
decimal operator -(decimal x);
```

结果是 0 减去 x 的计算值。

### 7.6.3 逻辑非运算符

对于 !x 形式的操作，一元运算符重载解析 (§7.2.3) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。只有一种预定义的逻辑非运算符：

```
bool operator !(bool x);
```

该运算符计算操作数的逻辑非：如果操作数是 true，结果就是 false；如果操作数是 false，结果就是 true。

### 7.6.4 按位求补运算符

对于 ~x 形式的操作，一元运算符重载解析 (§7.2.3) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。预定义的按位求补运算符为：

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

对于以上每个运算符，操作的结果是 x 的按位补码。

每个枚举类型 E 隐含地提供下列按位求补运算符：

```
E operator ~(E x);
```

当 x 是带有基本类型 U 的枚举类型 E 的表达式，那么 ~x 的计算结果与 (E)(~(U)x) 的计算结果完全相同。

### 7.6.5 间接运算符

### 7.6.6 地址运算符

### 7.6.7 前缀增量和减量运算符

*pre-increment-expression:*

```
++ unary-expression
```

*pre-decrement-expression:*

```
-- unary-expression
```

前缀增量或减量运算的操作数必须是变量类表达式、属性访问或索引访问。操作的结果是一个与操作数同类型的值。

如果前缀增量或减量运算是属性或索引访问，该属性或索引必须具有 get 和 set 两个访问函数。如果不是这样，将出现编译错误。

一元运算符重载解析 (§7.2.3) 用于选择特定的运算符实现。预定义的 ++ 和 -- 运算符存在于以下类型中：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal 以及任何 enum 类型。预定义的 ++ 运算符返回值为参数加 1，预定义的 -- 运算符返回值为参数减 1。

++x 或 --x 形式的前缀增量或减量操作的运行时处理包括以下几步：

- 如果  $x$  为变量：
  - 计算  $x$  产生变量。
  - 所选运算符被调用时参数为  $x$  的值。
  - 运算符返回的值存储在  $x$  的计算值所在的位置。
  - 运算符返回的值成为操作的结果。
- 如果  $x$  为属性或索引访问：
  - 计算实例表达式（如果  $x$  不是 `static`）、参数列表（如果  $x$  是索引访问）和  $x$ ，结果用于下面的 `get` 和 `set` 访问函数调用。
  - 调用  $x$  的 `get` 访问函数。
  - 将 `get` 访问函数的返回值作为参数，调用所选运算符。
  - 将运算符的返回值作为 `value` 参数，调用  $x$  的 `set` 访问函数。
  - 运算符返回的值成为操作的结果。

`++` 和 `--` 运算符也支持后缀表示法，如 §7.5.9 所述。 $x++$  或  $x--$  的结果是操作前  $x$  的值，而  $++x$  或  $--x$  的值是操作后  $x$  的值。在每一种情况下， $x$  本身在操作后都具有同样的值。

用后缀和前缀表示法都可以调用运算符 `++` 或运算符 `--` 的实现。这两种表示法有相同的运算符实现。

### 7.6.8 强制类型转换表达式

**强制类型转换表达式**用于显式地将表达式转换成一个给定类型。

*cast-expression:*  
     ( *type* ) *unary-expression*

设  $T$  为类型， $E$  为一元表达式， $(T)E$  形式的 **强制类型转换表达式** 将  $E$  的值显式地转换（§**错误！未找到引用源。**）为类型  $T$ 。如果没有将  $E$  的类型显式地转换到  $T$ ，将出现错误。另外，结果是显式转换产生的数值。即使  $E$  表示一个变量，结果也总是一个数值。

**强制类型转换表达式**的语法导致某些句法歧义。例如，表达式  $(x)-y$  既可以被解释为 **强制类型转换表达式**（将  $-y$  的强制类型转换为类型  $x$ ）也可解释为**附加表达式**与**加括号表达式**的结合（计算  $x - y$  的值）。

为解决 **强制类型转换表达式**的多义性，存在以下规则：只有当下列至少一条为真时，包含在括号中的一个或多个标记（*tokens*）（§**错误！未找到引用源。**）序列才被认为是 **强制类型转换表达式**：

- 标记的序列在语法上对类型是正确的，对表达式是不正确的。
- 标记的序列在语法上对类型是正确的，并且结束括号后紧跟的标记是“`~`”、“`!`”、“`(`”、标识符（§2.5.1）、文字（§**错误！未找到引用源。**）或除 `is` 之外的关键字（§**错误！未找到引用源。**）。

以上规则意味着只有当 **强制类型转换表达式**的结构是明显的，它才被认为是一个 **强制类型转换表达式**。

上述“正确的语法”只意味着标志的序列必须遵从特定语法。它并不考虑构成的标识符的实际意义。例如，如果  $x$  和  $y$  为标识符，那么  $x.y$  对于类型而言语法正确，即使  $x.y$  实际上不表示一个类型。



从所遵从的消除歧义规则来看，如果  $x$  和  $y$  是标识符， $(x)y$ 、 $(x)(y)$  和  $(x)(-y)$  是强制类型转换表达式，但  $(x)-y$  不是，即使  $x$  标识一个类型。然而，如果  $x$  是一个标识预定义类型（如 `int`）的关键字，那么所有四种形式都是强制类型转换表达式（因为关键字自己不可能是一个表达式）。

7.7 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

```

7.7.1 乘法运算符

对于  $x * y$  形式的操作，二元运算符重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的乘法运算符为如下所列。运算符计算  $x$  和  $y$  的乘积。

- 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` 环境中，如果乘积在结果类型的数值范围之外，将产生 `OverflowException` 异常（上溢出异常）。在 `unchecked` 环境中，不报告有溢出，结果值的有效高位将被舍掉。

- 浮点乘法：

```

float operator *(float x, float y);
double operator *(double x, double y);

```

乘积的计算是依据 IEEE 754 算法的规则。下表列出非零有限值、零、无穷大和 NaN 的所有可能组合的计算结果。表中， $x$  和  $y$  是正的有限值， $z$  是  $x * y$  的结果。如果结果对于目标类型来说太大， $z$  就是无穷大。如果结果对于目标类型来说太小， $z$  就是零。

	$+y$	$-y$	$+0$	$-0$	$+\infty$	$-\infty$	NaN
$+x$	$z$	$-z$	$+0$	$-0$	$+\infty$	$-\infty$	NaN
$-x$	$-z$	$z$	$-0$	$+0$	$-\infty$	$+\infty$	NaN
$+0$	$+0$	$-0$	$+0$	$-0$	NaN	NaN	NaN
$-0$	$-0$	$+0$	$-0$	$+0$	NaN	NaN	NaN
$+\infty$	$+\infty$	$-\infty$	NaN	NaN	$+\infty$	$-\infty$	NaN
$-\infty$	$-\infty$	$+\infty$	NaN	NaN	$-\infty$	$+\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数乘法：

```
decimal operator *(decimal x, decimal y);
```

如果结果值太大而不能用 decimal 格式表示，将产生 `OverflowException` 异常（上溢出异常）。

如果结果值太小而不能用 decimal 格式表示，结果为零。

## 7.7.2 除法运算符

对于  $x / y$  形式的操作，二元运算符重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的除法运算符为如下所列。运算符计算  $x$  和  $y$  的商。

- 整数除法：

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

如果右操作数的值为零，产生 `DivideByZeroException` 异常（除零异常）。

除法将结果向零取整，结果的绝对值是小于两个操作数之商绝对值的最大可能的整数。当两个操作数符号相同时，结果为零或正的；当两个操作数符号相反时，结果为零或负的。

如果左操作数是最大的负 `int` 或 `long` 型数值，而右操作数为  $-1$ ，将产生上溢出。在 `checked` 环境中，将产生 `OverflowException` 异常（上溢出异常）。在 `unchecked` 环境中，不报告有溢出，结果改为左操作数的值。

- 浮点除法：

```
float operator /(float x, float y);
double operator /(double x, double y);
```

商的计算是依据 IEEE 754 算法的规则。下表列出非零有限值、零、无穷大和 NaN 的所有可能组合的计算结果。表中， $x$  和  $y$  是正的有限值， $z$  是  $x / y$  的结果。如果结果对于目标类型来说太大， $z$  就是无穷大。如果结果对于目标类型来说太小， $z$  就是零。

	$+y$	$-y$	$+0$	$-0$	$+\infty$	$-\infty$	NaN
$+x$	$z$	$-z$	$+\infty$	$-\infty$	$+0$	$-0$	NaN
$-x$	$-z$	$z$	$-\infty$	$+\infty$	$-0$	$+0$	NaN
$+0$	$+0$	$-0$	NaN	NaN	$+0$	$-0$	NaN
$-0$	$-0$	$+0$	NaN	NaN	$-0$	$+0$	NaN
$+\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	NaN	NaN	NaN
$-\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数除法：

```
decimal operator /(decimal x, decimal y);
```

如果右操作数的值为零，产生 `DivideByZeroException` 异常（除零异常）。如果结果值太大而不能用 decimal 格式表示，将产生 `OverflowException` 异常（上溢出异常）。如果结果值太小而不能用 decimal 格式表示，结果为零。

7.7.3 余数运算符

对于 `x % y` 形式的操作，二元运算符重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的余数运算符为如下所列。运算符计算 `x` 和 `y` 的相除的余数。

- 整数余数：

```
int operator %(int x, int y);
int operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

`x % y` 的结果是由 `x - (x / y) * y` 产生的值。如果 `y` 是零，产生 `DivideByZeroException` 异常（除零异常）。余数运算符从不引起溢出。

- 浮点余数：

```
float operator %(float x, float y);
double operator %(double x, double y);
```

商的计算是依据 IEEE 754 算法的规则。下表列出非零有限值、零、无穷大和 NaN 的所有可能组合的计算结果。表中，`x` 和 `y` 是正的有限值，`z` 是 `x % y` 的结果，按 `x - n * y` 计算而得，其中 `n` 是小于或等于 `x / y` 的最大可能的整数。这种计算余数的方法类似于整数操作数所用的方法，而不同于 IEEE 754 的定义（该定义中 `n` 是最接近于 `x / y` 的整数）。

	+y	-y	+0	-0	+∞	-∞	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
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-∞	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，产生 `DivideByZeroException` 异常（除零异常）。如果结果值太大而不能用 `decimal` 格式表示，将产生 `OverflowException` 异常（上溢出异常）。如果结果值太小而不能用 `decimal` 格式表示，结果为零。

7.7.4 加法运算符

对于 `x + y` 形式的操作，二元运算符重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的加法运算符为如下所列。对于数字和枚举类型，预定义的加法运算符计算两个操作数的和。当一个或两个操作数是字符串类型时，预定义的加法运算符将连接操作符代表的字符串。

- 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 环境中，如果和在结果类型的数值范围之外，将产生 OverflowException 异常（上溢出异常）。在 unchecked 环境中，不报告有溢出，结果值的有效高位将被丢弃。

- 浮点加法：

```
float operator +(float x, float y);
double operator +(double x, double y);
```

和的计算是依据 IEEE 754 算法的规则。下表列出非零有限值、零、无穷大和 NaN 的所有可能组合的计算结果。表中，x 和 y 是非零有限值，z 是 x + y 的结果。如果 x 和 y 大小相等符号相反，z 为正零。如果 x + y 的结果对于目标类型来说太大，z 就是无穷大，符号与 x + y 的符号相同。如果 x + y 的结果对于目标类型来说太小，z 为零，符号与 x + y 的符号相同。

	y	+0	-0	+∞	-∞	NaN
x	z	x	x	+∞	-∞	NaN
+0	y	+0	+0	+∞	-∞	NaN
-0	y	+0	-0	+∞	-∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
-∞	-∞	-∞	-∞	NaN	-∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数加法：

```
decimal operator +(decimal x, decimal y);
```

如果结果值太大而不能用 decimal 格式表示，将产生 OverflowException 异常（上溢出异常）。如果结果值太小而不能用 decimal 格式表示，结果为零。

- 枚举加法。每种枚举类型隐含地提供下列预定义的运算符，其中 E 是 enum 类型，U 是 E 的基础类型：

```
E operator +(E x, U y);
E operator +(U x, E y);
```

运算符完全按照 (E)((U)x + (U)y) 计算。

- 字符串连接：

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

当一个或两个操作数为 string 类型时，二元 + 运算符进行字符串连接。如果字符串连接的一个操作数为 null，则用一个空字符串代替。另外，通过调用从类型 object 继承来的虚方法 ToString()，任何非字符串参数将被转换成字符串表示法。如果 ToString() 返回 null，则用一个空字符串代替。

字符串连接运算符的结果是一个字符串，由左操作数的字符后面连接右操作数的字符组成。字符串连接运算符不返回 null 值。如果没有足够的内存分配给结果字符串，将可能产生 OutOfMemoryException 异常。

- 代表组合。每种代表类型隐含地提供下列预定义的运算符，其中 D 是代表类型：

D operator +(D x, D y);

当一个或两个操作数是代表类型 D 时，二元 + 运算符进行代表组合。如果第一个操作数是 null，那么操作的结果是第二个操作数的值。否则，如果第二个操作数是 null，操作结果是第一个操作数的值。另外，如果 D 是组合代表类型 (§15.1.1)，那么操作的结果是一个新的代表实例，该实例在被调用时先调用第一个操作数，然后调用第二个操作数。否则，D 不是组合代表类型，并且产生 MulticastNotSupportedException 异常。

7.7.5 减法运算符

对于 x - y 形式的操作，二元运算符重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的减法运算符为如下所列。运算符从 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 环境中，如果差在结果类型的数值范围之外，将产生 overflowException 异常（上溢出异常）。在 unchecked 环境中，不报告有溢出，结果值的有效高位将被舍弃。

- 浮点减法：

float operator -(float x, float y);  
double operator -(double x, double y);

差的计算是依据 IEEE 754 算法的规则。下表列出非零有限值、零、无穷大和 NaN 的所有可能组合的计算结果。表中，x 和 y 是非零有限值，z 是 x - y 的结果。如果 x 和 y 大小相等，z 为正零。如果 x - y 的结果对于目标类型来说太大，z 就是无穷大，符号与 x - y 的符号相同。如果 x - y 的结果对于目标类型来说太小，z 为零，符号与 x - y 的符号相同。

	y	+0	-0	+∞	-∞	NaN
x	z	x	x	-∞	+∞	NaN
+0	-y	+0	+0	-∞	+∞	NaN
-0	-y	-0	+0	-∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
-∞	-∞	-∞	-∞	-∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 小数减法：

decimal operator -(decimal x, decimal y);

如果结果值太大而不能用 decimal 格式表示，将产生 overflowException 异常（上溢出异常）。如果结果值太小而不能用 decimal 格式表示，结果为零。

- 枚举减法。每种枚举类型隐含地提供下列预定义的运算符，其中 E 是 enum 类型，U 是 E 的基础类型：

U operator -(E x, E y);

运算符按照  $(U)((U)x - (U)y)$  计算。换句话说，该运算符依次计算  $x$  和  $y$  值的差，结果的类型是枚举的基础类型。

```
E operator -(E x, U y);
```

运算符按照  $(E)((U)x - y)$  计算。换句话说，该运算符从枚举的基础类型中减去一个值，产生枚举值。

- 代表去除。每种代表类型隐含地提供下列预定义的运算符，其中  $D$  是代表类型：

```
D operator -(D x, D y);
```

当一个或两个操作数为代表类型  $D$ ，二元  $-$  运算符完成代表去除。

### 问题

我们需要描述代表类型的减法操作符的语义。

## 7.8 移位运算符

$\ll$  和  $\gg$  运算符用于完成移位操作。

```
shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression
```

对于  $x \ll \text{count}$  或  $x \gg \text{count}$  形式的操作，二元运算符重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

当声明重载移位操作时，第一个操作数的类型必须总是包含运算符声明的类或结构，第二个操作数必须总是 `int`。

预定义的移位操作如下所列。

- 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);
```

$\ll$  运算符按照以下的叙述计算将  $x$  左移的位数。

$x$  的高位被舍弃，余下的位左移，空的低位置零。

- 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);
```

$\gg$  运算符按照以下的叙述计算将  $x$  右移的位数。

当  $x$  为 `int` 或 `long` 类型时， $x$  的低位被舍弃，余下的位右移，如果  $x$  为非负，空的高位置零，如果  $x$  为负，空的高位置 1。

当  $x$  为 `uint` 或 `ulong` 类型时， $x$  的低位被舍弃，余下的位右移，空的高位置零。

对于预定义的运算符，移动的位数计算如下：

- 当 x 的类型为 int 或 uint 时，移位计数由 count 的低五位给定。换句话说，移位计数由 count & 0x1F 计算而来。
- 当 x 的类型为 long 或 ulong 时，移位计数由 count 的低六位给定。换句话说，移位计数由 count & 0x3F 计算而来。

如果移位计数为零，移位操作就简单地返回 x 的值。

移位操作不会产生溢出，且在 checked 和 unchecked 环境中结果都相同。

当 >> 的左操作数为有符号的整型类型时，运算符执行算术右移，其中操作数的最高位（符号位）被移进高位的空位。当 >> 运算符的左操作数为无符号整型类型时，运算符执行逻辑右移，其中高位的空位总是置零。若要执行从操作数类型推断的相反的操作，可以用显式强制类型转换。例如，如果 x 是 int 类型的变量，操作 (int)((uint)x >> y) 完成 x 的逻辑右移。

## 7.9 关系运算符

==、!=、<、>、<=、>= 和 is 运算符被称为关系运算符。

```
relational-expression:
    shift-expression
relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
relational-expression is reference-type

equality-expression:
    relational-expression
equality-expression == relational-expression
equality-expression != relational-expression
```

is 运算符将在 §错误！未找到引用源。 中叙述。

==、!=、<、>、<= 和 >= 运算符作为一组被称为比较运算符。对于 x op y 形式的操作，其中 op 为一个比较运算符，重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的比较运算符在下面部分叙述。所有预定义的比较运算符返回类型为 bool 的结果，如下表所述。

操作	结果
x == y	如果 x 等于 y 为 true，否则为 false
x != y	如果 x 不等于 y 为 true，否则为 false
x < y	如果 x 小于 y 为 true，否则为 false
x > y	如果 x 大于 y 为 true，否则为 false
x <= y	如果 x 小于或等于 y 为 true，否则为 false
x >= y	如果 x 大于或等于 y 为 true，否则为 false

### 7.9.1 整数比较运算符

预定义的整数比较运算符为：

```
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);
```

以上每个运算符都是比较两个整型操作数的数值，返回的 bool 值显示特定关系是真 (true) 还是假 (false)。

### 7.9.2 浮点比较运算符

预定义的浮点比较运算符为：

```
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);
```

根据 IEEE 754 标准的规则，运算符对两个操作数进行比较：

- 如果任一操作数为 NaN，除 != 外的所有运算符的结果为 false，!= 运算符的结果为 true。对任意两个操作数， $x \neq y$  与  $!(x == y)$  产生相同的结果。然而，当一个或两个操作数为 NaN 时，<、>、<= 和 >= 运算符与其相反运算符的逻辑非结果不同。例如，如果 x 或 y 是 NaN，那么  $x < y$  是 false，但  $!(x >= y)$  是 true。



- 当两个操作数都不是 NaN 时，运算符比较两个浮点操作数数值的大小顺序

$$-\infty < -\max < \dots < -\min < -0.0 == +0.0 < +\min < \dots < +\max < +\infty$$

其中 min 和 max 是能用给定浮点格式表示的最小和最大的正有限数。该顺序值得注意的结果是：

- 负零和正零被认为是相等的。
- 负无穷大被认为小于所有其它值，但是等于另一个负无穷大。
- 正无穷大被认为大于所有其它值，但是等于另一个正无穷大。

### 7.9.3 小数比较运算符

预定义的小数比较运算符为：

```
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);
```

以上每个运算符都是比较两个小数操作数的数值，返回的 bool 值显示特定关系是真 (true) 还是假 (false)。

### 7.9.4 布尔相等运算符

预定义的布尔相等运算符为：

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

若 x 和 y 都是 true 或者 x and y 都是 false，== 的结果为 true；否则结果为 false。

若 x 和 y 都是 true 或者 x and y 都是 false，!= 的结果为 false；否则结果为 true。当操作数为 bool 类型时，!= 运算符与 ^ 运算符产生的结果相同。

### 7.9.5 枚举比较运算符

每种枚举类型隐含地提供下列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);
```

当 x 和 y 为具有基础类型 u 的枚举类型 E，且 op 为一种比较运算符时，x op y 的计算结果与 ((u)x) op ((u)y) 的计算结果相同。换句话说，枚举类型比较运算符只简单地比较两个操作数的基础整型值。

### 7.9.6 引用类型相等运算符

预定义的引用类型相等运算符为：

```
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地结果。

由于预定义的引用类型相等运算符接受 `object` 类型的操作数，它们适用于所有不声明可行的 `operator ==` 和 `operator !=` 成员的类型。反之，任何可行的用户自定义相等运算符实际上隐藏了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操作数为 *引用类型值* 或 `null` 值，此外需要从一个操作数类型到另一个操作数类型的隐含转换存在。除非这两个条件都为真，否则将出现编译错误。这些规则中值得注意的是：

- 在编译时用预定义的引用类型相等运算符去比较已知为不同的两个引用是错误的。例如，如果操作数的编译时类型为两个类类型 `A` 和 `B`，且 `A` 和 `B` 都不是从另一方派生的，那么这两个操作数不可能引用同一对象。因此，这一操作被认为是编译错误。
- 预定义的引用类型相等运算符不允许比较数值类型的操作数。因此，除非结构（`struct`）类型声明自己的相等运算符，否则不可能比较结构类型的值。
- 预定义的引用类型相等运算符不会对其操作数进行单元操作（boxing operation）。这样的单元操作是没有意义的，因为对新分配的单元实例的引用必然与其它引用不同。

对于 `x == y` 或 `x != y` 形式的操作，如果任何可行的运算符 `operator ==` 或 `operator !=` 存在，运算符重载解析（§7.2.4）规则将选择运算符而不是预定义的引用类型相等运算符。然而，通过显式地将一个或两个操作数的强制类型转换为类型 `object`，就有可能选择引用类型相等运算符。示例

```
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);
    }
}
```

产生的输出为

```
True
False
False
False
```

`s` 和 `t` 变量指向两个包含相同字符的不同的 `string` 实例。第一个比较输出为 `True` 是因为当两个操作数都为类型 `string` 时，选择了预定义的字符串相等运算符（§**错误！未找到引用源。**）。其余比较都输出 `False` 是因为当一个或两个操作数为类型 `object` 时，选择了预定义的引用类型相等运算符。

注意，以上技术对数值类型没有意义。示例

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        Console.WriteLine((object)i == (object)j);
    }
}
```

输出 False，因为强制类型转换产生了对两个 int 值不同实例的引用。

### 7.9.7 字符串相等运算符

预定义的字符串相等运算符为：

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

当下列之一为真时两个 string 值被认为是相等的：

- 两个值都为 null。
- 两个值都为字符串实例（长度相等且每个字符位置的字符相同）的非空引用。

字符串相等运算符比较字符串的 *值* 而不是字符串的 *引用*。当两个不同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 §**错误！未找到引用源。** 所述，引用类型相等运算符用于比较字符串引用而不是字符串的值。

### 7.9.8 代表相等运算符

每种代表类型 D 隐含地提供下列预定义的比较运算符：

```
bool operator ==(System.Delegate x, D y);
bool operator ==(D x, System.Delegate y);
bool operator !=(System.Delegate x, D y);
bool operator !=(D x, System.Delegate y);
```

在下面情况下两个代表实例被认为相等：

- 如果任一代表实例都为 null，当且仅当两者都为 null 时，它们才相等。
- 如果任一代表实例是由另一个代表表示的，当且仅当它们是同一个代表实例表示的时，它们才相等。否则，
- 如果任一代表实例是非多点传送代表，当且仅当两者都是非多点传送代表，且：
  - 两者指向同一静态方法，或
  - 两者指向同一目标对象的同一非静态方法。
 时，它们才相等。
- 如果任一代表实例都是多点传送代表，它们相等地条件是：当且仅当它们的调用列表具有相同长度，且每个代表的调用列表等于其相应代表的调用列表，且次序相等。

注意，根据以上定义，只要有同样的返回值和参数类型，不同类型的代表也可以是相等的。

### 7.9.9 is 运算符

is 运算符用于检查一个对象的运行时类型是否与给定类型一致。在 e is T 形式的操作中，e 必须是 *引用类型* 的表达式，而 T 必须是一个 *引用类型*。如果不是这样，将出现编译错误。

如果 e 不为 null，且存在隐含的引用转换（§**错误！未找到引用源。**）（从被 e 引用的实例的运行时类型转换到 T 给定的类型）时，e is T 操作返回 true。换句话说，e is T 检查出 e 不为 null，完成 (T)(e) 形式的 *类型强制表达式*（§**错误！未找到引用源。**），这里不产生异常。

如果  $e$  是  $T$  在编译时已知为总是 `true` 或总是 `false`，将产生编译错误。如果存在  $e$  的编译时类型到  $T$  的隐式引用转换，操作就总为 `true`。如果没有  $e$  的编译时类型到  $T$  的隐式或显式引用转换，操作就总为 `false`。

## 7.10 逻辑运算符

`&`、`^` 和 `|` 运算符被称为逻辑运算符。

```
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
```

对于  $x \text{ op } y$  形式的操作，其中  $op$  为一个逻辑运算符，重载解析 (§7.2.4) 用于选择特定的运算符实现。操作数被转换成所选运算符的参数类型，结果的类型是该运算符的返回类型。

预定义的逻辑运算符在后面部分叙述。

### 7.10.1 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与 (AND)，`|` 运算符计算两个操作数的按位逻辑或 (OR)，`^` 运算符计算两个操作数的按位逻辑异或 (exclusive OR)。这些操作不会出现溢出。

### 7.10.2 枚举逻辑运算符

每种枚举类型  $E$  隐含地提供下列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);
```

当  $x$  和  $y$  是具有基础类型  $U$  的枚举类型  $E$  的表达式，而  $op$  是一个逻辑运算符时， $x \text{ op } y$  的计算结果与  $(E)((U)x) \text{ op } ((U)y)$  计算结果相同。换句话说，枚举类型逻辑运算符对两个操作数的基础类型进行逻辑操作。

### 7.10.3 布尔逻辑运算符

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);
```

若  $x$  和  $y$  都是 `true`， $x \& y$  的结果为 `true`；否则结果为 `false`。

若  $x$  或者  $y$  是 `true`， $x | y$  的结果为 `true`；否则结果为 `false`。

若  $x$  是 `true` 且  $y$  是 `false`，或者  $x$  是 `false` 且  $y$  是 `true`， $x \wedge y$  的结果为 `true`；否则结果为 `false`。  
当操作数为 `bool` 类型时， $\wedge$  运算符与  $!=$  运算符产生的结果相同。

### 7.11 条件逻辑运算符

$\&\&$  和  $||$  运算符被称为条件逻辑运算符。有时，它们也被称为“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  $|$  运算符的条件形式。

- 除了只有在  $x$  为 `true` 时计算  $y$  以外， $x \&\& y$  操作与  $x \& y$  操作相当。
- 除了只有在  $x$  为 `false` 时计算  $y$  以外， $x || y$  操作与  $x | y$  操作相当。

$x \&\& y$  或  $x || y$  形式的操作在处理时应用二元运算符重载解析 (§7.2.4)，就像  $x \& y$  或  $x | y$  形式的操作一样。然后，

- 如果重载解析没有找到最好的运算符，或者重载解析选择了一个预定义的整数逻辑运算符，将出现错误。
- 如果所选运算符是一个预定义的布尔逻辑运算符 (§错误！未找到引用源。)，操作将按照 §错误！未找到引用源。中叙述的处理。
- 若所选运算符是用户定义的运算符，操作将按照 §错误！未找到引用源。中叙述的处理。

直接重载条件逻辑运算符是不可能的。但是，由于条件逻辑运算符的求值是根据普通逻辑运算符，而在某些约束下，普通逻辑运算符的重载也被认为是条件逻辑运算符的重载。§错误！未找到引用源。中将进一步叙述。

#### 7.11.1 布尔条件逻辑运算符

当  $\&\&$  或  $||$  的操作数是 `bool` 类型，或当操作数的类型不能解释为可行的 `operator &` 或 `operator |` 却可解释为到 `bool` 的隐式转换，操作将按如下方式处理：

- 操作  $x \&\& y$  按  $x ? y : \text{false}$  求值。换句话说， $x$  先求值，并转换为 `bool` 类型。然后，如果  $x$  为 `true`， $y$  求值，并转换为 `bool` 类型，这就是操作的结果。否则，操作的结果为 `false`。

- 操作 `x || y` 按 `x ? true: y` 求值。换句话说，`x` 先求值，并转换为 `bool` 类型。然后，如果 `x` 为 `true`，操作的结果为 `true`。否则，`y` 求值，并转换为 `bool` 类型，这就是操作的结果。

### 7.11.2 用户自定义的条件逻辑运算符

当 `&&` 或 `||` 的操作数类型声明了可行的用户自定义的 `operator &` 或 `operator |` 时，以下两项必须为真，其中 `T` 是所选运算符声明的类型：

- 返回值和所选运算符的每个参数类型必须为 `T`。换句话说，运算符必须计算类型为 `T` 的两个操作数的逻辑 AND 和逻辑 OR，且必须返回类型为 `T` 的结果。
- `T` 必须包含 `operator true` 和 `operator false` 的声明。

如果以上两个条件不满足，将出现编译错误。另外，`&&` 或 `||` 操作的运算是通过组合用户自定义的 `operator true` 或 `operator false` 与所选的用户自定义运算符：

- `x && y` 操作的求值与 `T.false(x) ? x: T.&(x, y)` 相同，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。换句话说，`x` 先求值，在其结果之上调用 `operator false` 确定 `x` 是否明确为假。然后，如果 `x` 明确为假，操作的结果就是先前计算的 `x` 的值。否则，`y` 求值，在先前计算的 `x` 值基础上调用所选 `operator &`，`y` 的计算值产生操作的结果。
- `x || y` 操作的求值与 `T.true(x) ? x: T.|(x, y)` 相同，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。换句话说，`x` 先求值，在其结果之上调用 `operator true` 确定 `x` 是否明确为真。然后，如果 `x` 明确为真，操作的结果就是先前计算的 `x` 的值。否则，`y` 求值，在先前计算的 `x` 值基础上调用所选 `operator |`，`y` 的计算值产生操作的结果。

在以上每一个操作中，`x` 给出的表达式只求值一次，给出的表达式要么不求值，要么只求值一次。

有关实现 `operator true` 和 `operator false` 的类型的例子，参见 §11.3.2。

### 7.12 条件运算符

`?:` 运算符被称为条件运算符，有时也被称为三元运算符。

*conditional-expression:*

*conditional-or-expression*

*conditional-or-expression ? expression : expression*

`b ? x: y` 形式的条件表达式首先对条件 `b` 求值。然后，如果 `b` 为 `true`，计算 `x`，其值为操作的结果。否则，计算 `y`，其值为操作结果。条件表达式不会对 `x` 和 `y` 都进行计算。

条件表达式具有“右结合性”，意思是操作从右向左组合。例如，`a ? b: c ? d: e` 形式表达式的计算与 `a ? b: (c ? d: e)` 相同。

`?:` 运算符的第一个操作数必须是类型可以被隐式转换为 `bool` 的表达式，或者是类型为实现 `operator true` 的表达式。如果这两个要求都不满足，将出现编译错误。

`?:` 运算符的第二个和第三个操作数控制条件表达式的类型。假设 `x` 和 `y` 是第二个和第三个操作数的类型。那么，

- 如果 `x` 和 `y` 是同一类型，那么这一类型就是条件表达式的类型。
- 要不然，如果存在 `x` 到 `y` 而不是 `y` 到 `x` 的隐式转换 (§6.1)，那么 `y` 就是条件表达式的类型。
- 要不然，如果存在 `y` 到 `x` 而不是 `x` 到 `y` 的隐式转换 (§6.1)，那么 `x` 就是条件表达式的类型。

- 否则，将不能确定表达式的类型，出现编译错误。

b? x: y 形式条件表达式的运行时处理包括以下几步：

- 首先，b 求值，b 的 bool 值确定：
  - 如果存在从 b 的类型到 bool 的隐式转换，那么，该隐式转换的执行产生一个 bool 值。
  - 否则，b 的类型所决定的 operator true 被调用产生 bool 值。
- 如果以上步骤产生的 bool 值为 true，那么计算 x 的值，该值被转换成条件表达式的类型，成为条件表达式的结果。
- 否则，计算 y 的值，该值被转换成条件表达式的类型，成为条件表达式的结果。

## 7.13 赋值运算符

赋值运算符给变量、属性和索引单元赋一个新值。

*assignment:*

*unary-expression assignment-operator expression*

*assignment-operator: one of*

*= += -= \*= /= %= &= |= ^= <=>=*

赋值的左操作数必须是变量、属性访问或索引访问等类型的表达式。

= 运算符被称为**简单赋值运算符**。它将右操作数的值赋给左操作数给出的变量、属性或索引单元。简单赋值运算符在 §**错误！未找到引用源。** 中叙述。

用二元运算符作前缀与 = 字符构成的运算符称为**组合赋值运算符**。这些运算符在两个操作数执行标明的操作，然后将结果值赋给左操作数给出的变量、属性或索引单元。组合赋值运算符在 §**错误！未找到引用源。** 中叙述。

赋值运算符为“右结合性”，意思是操作从右向左组合。例如，a = b = c 形式的表达式求值与 a = (b = c) 相同。

### 7.13.1 简单赋值

= 运算符被称为简单赋值运算符。在简单赋值中，右操作数必须是一个类型可隐含地转换到左操作数的表达式。该操作将右操作数的值赋给左操作数给出的变量、属性或索引单元。

简单赋值表达式的结果是赋给左操作数的值。结果与左操作数具有相同的类型，并且总是数值。

如果左操作数是属性或索引访问，该属性或索引必须具有 set 访问函数。如果不是这样，将出现编译错误。

对于 x = y 形式的简单赋值，其运行时处理包含以下几步：

- 如果 x 为变量：
  - 计算 x 产生变量。
  - 计算 y，如果需要，通过隐式转换 (§6.1) 转换成 x 的类型。
  - 如果 x 给出的变量是一个引用类型的数组单元，将进行运行时检查以保证计算出的 y 值与 x 作为一个单元的数组实例相匹配。如果 y 为 null，或者存在着被 y 引用的实例的实际类型到包含

x 的数组实例的实际单元类型之间的隐含引用转换（§错误！未找到引用源。），那么检查成功。否则，产生 `ArrayTypeMismatchException` 异常。

- y 计算的结果值与转换被存储在计算 x 所给出的位置。
- 如果 x 为属性或索引访问：
  - 计算实例表达式（如果 x 不是 `static`）、参数列表（如果 x 是一个索引访问）以及 x，结果用于后面的 `set` 访问函数调用。
  - 计算 y，如果需要，通过隐式转换（§6.1）转换成 x 的类型。
  - x 的 `set` 访问函数被调用，其 `value` 参数为 y 的计算值。

数组联合变化规则（array co-variance rules）（§12.5）允许数组类型 `A[]` 的一个值是数组类型 `B[]` 的一个实例的引用，倘若存在从 B 到 A 的隐式引用转换。由于这些规则，对一个引用类型的数组单元赋值时需要运行时检查，以确保将要赋的值与数组实例匹配。在以下示例中

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;           // ok
oa[1] = "Hello";        // ok
oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

最后一个赋值导致 `ArrayTypeMismatchException` 异常出现，因为 `ArrayList` 的实例不能存储在 `string[]` 的单元中。

当结构体类型（*struct-type*）中声明的属性或索引是一个赋值的目标时，实例表达式连同属性和索引访问必须是变量。如果实例表达式为数值，将出现编译错误。

给出的声明：

```
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; }
    }
}

```

在下例中

```

Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;

```

给 p.X、p.Y、r.A 和 r.B 赋值是允许的，因为 p 和 r 是变量。然而，在下例中

```

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;

```

赋值都是非法的，因为 r.A 和 r.B 不是变量。

### 7.13.2 组合赋值

$x \text{ op} = y$  形式的操作在处理时运用二元运算符重载解析 (§7.2.4)，该操作同  $x \text{ op } y$  形式的操作一样。那么，

- 如果所选运算符的返回类型可隐式转换为  $x$  的类型，该操作就像  $x = x \text{ op } y$  一样求值，只是  $x$  只求值一次。
- 另外，如果所选运算符是预定义的运算符，如果所选运算符的返回类型可显式转换为  $x$  的类型，且如果  $y$  可隐式转换为  $x$  的类型，那么该操作可像  $x = (T)(x \text{ op } y)$  一样求值，其中  $T$  是  $x$  的类型，只是  $x$  只求值一次。
- 否则，组合赋值就是非法的，会出现编译错误。

“只求值一次”的意思是在  $x \text{ op } y$  的计算中，任何  $x$  的组成表达式的结果都被暂时保存，然后在给  $x$  赋值时重用。例如，赋值  $A()[B()] += C()$ ，其中  $A$  是返回  $\text{int}[]$  的一个方法，而  $B$  和  $C$  是返回  $\text{int}$  的方法，这些方法按  $A$ 、 $B$ 、 $C$  顺序只被调用一次。

当组合赋值的左操作数是属性访问或索引访问时，属性或索引必须具有 `get` 访问函数和 `set` 访问函数两者。如果不是这样，将出现编译错误。

上述第二个规则允许  $x \text{ op} = y$  的求值在某些情况下像  $x = (T)(x \text{ op } y)$  一样。当左操作符为 `sbyte`、`byte`、`short`、`ushort` 或 `char` 类型时，这样的规则使得预定义的运算符可以像组合运算符一样使用。即使两个参数都为这些类型之一时，预定义的运算符也会产生 `int` 类型的结果，如 §7.2.6.2 所述。因此，没有强制类型转换就不可能将结果赋给左操作数。

对于预定义的运算符，该规则的直观结果就是，如果  $x \text{ op } y$  和  $x = y$  都被允许，则  $x \text{ op} = y$  被允许。在如下示例中

```

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是相应简单赋值已经有错误了。

### 7.13.3 事件赋值

#### 问题

我们需要编写这一节。

### 7.14 表达式

表达式为条件表达式或赋值。

*expression:*  
*conditional-expression*  
*assignment*

### 7.15 常量表达式

常量表达式是在编译时就可完全求值的表达式。

*constant-expression:*  
*expression*

常量表达式的类型可以是以下类型之一：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string、枚举类型或空（null）类型。在常量表达式中允许以下结构：

- 文字（包括 null）。
- 对类和结构体类型中 const 成员的引用。
- 对枚举类型成员的引用。
- 括号括起来的子表达式。
- 强制类型转换表达式，提供的目标类型为上面所列之一。
- 预定义的 +、-、! 以及 ~ 等一元运算符。
- 预定义的 +、-、\*、/、%、<<、>>、&、|、^、&&、||、==、!=、<、>、<= 以及 >= 等二元运算符，提供的每个操作数是以上所列之一。
- ?: 条件运算符。

只要表达式是以上所列的类型之一，并且只包含上面所列的结构，该表达式就可以在编译时赋值。即使该表达式是一个包含非常量结构的较大表达式的子表达式时也是正确的。

除了在运行时求值可能产生异常，在编译时赋值可能产生编译错误之外，常量表达式在编译时赋值与非常量表达式在运行时赋值使用同样的规则。

除非常量表达式被明确地置于 unchecked 环境中，否则出现在整型算术运算符中的溢出，以及表达式在编译时赋值的转换都会导致编译错误 (§7.5.13)。

常数表达式在下列情况下使用。在这些情况下，如果表达式在编译时不能完全求值就会出错。

- 常量声明 (§10.3)。
- 枚举成员声明 (§14.2)。
- switch 语句的 case 标号 (§错误！未找到引用源。 )。
- goto case 语句 (§错误！未找到引用源。 )。
- 包含初始值的数组创建表达式中的维数长度 (§7.5.10.2)。
- 属性 (§错误！未找到引用源。 )。

如果常量表达式的值在目的类型的数值范围之内，隐含的常量表达式转换 (§错误！未找到引用源。 ) 允许 int 类型的常量表达式转换为 sbyte、byte、short、ushort、uint 或 ulong 类型。

## 7.16 布尔表达式

布尔表达式是产生 bool 类型结果的表达式。

*boolean-expression:*  
*expression*

if-语句 (§错误！未找到引用源。 )、while-语句 (§错误！未找到引用源。 )、do-语句 (§错误！未找到引用源。 ) 或 for-语句 (§错误！未找到引用源。 ) 中的控制条件表达式是布尔表达式。?: 运算符 (§错误！未找到引用源。 ) 的控制条件表达式与布尔表达式虽然遵循同样的规则，但是基于运算符优先级的原因，它被归为条件或表达式 (conditional-or-expression)。

布尔表达式必须是可以被隐式转换为 bool 的类型，或实现 operator true 的类型。如果两者都不满足，就会出现编译错误。

当布尔表达式不是可以被隐式转换为 bool 的类型，而是实现 operator true 的类型，那么按照表达式的计算，该类型提供的 operator true 实现被调用，产生一个 bool 值。

§11.3.2 中的 DBBool 结构体类型提供一个实现 operator true 类型的例子。



## 8. 语句

C# 提供各式各样的语句。当中的绝大部分语句对于那些进行过 C 和 C++ 编程的人员来讲都很熟悉。

*语句：*

*标号语句*  
*声明语句*  
*嵌套语句*

*嵌套语句：*

*块语句*  
*空语句*  
*表达式语句*  
*选择语句*  
*重复 iteration 语句*  
*跳转语句*  
*try 语句*  
*checked 语句*  
*unchecked 语句*  
*lock 语句*

嵌入语句出现在语句的内部。与通常意义下的语句不同，嵌入语句不允许在其中出现声明语句和带标号语句

比如下列代码

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

有错误，因为 if 语句在自身的分支中需要的是嵌入语句而不是通常的语句。假如允许这样的代码，那么变量 i 会被声明，但是它并不能被适用。

### 8.1 终点与可达性

每条语句都有终点。更直观地说，语法上语句书写完成的地方就是该语句的终点。对于复合语句（嵌套了其他语句）来说，它的执行规则详尽描述了当嵌套语句到达终点时采取的操作。比如说，当程序执行到一个块中某语句的末尾，那么程序将继续执行块的下一条语句。

如果一条语句可以被执行，那么我们称它为是可达的(reachable)。相反，如果一条语句无法被执行，我们称之为是不可达的(unreachable)。

下面看一个例子

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
Label:
    Console.WriteLine("reachable");
}
```

第二个 Console.WriteLine 调用是不可到达的因为这条语句不会被执行。

当便要此段代码时，编译器会发出语句无法到达的警告。这并不代表语句本身有错使之无法到达。

判断一个语句是否可以到达，编译器是通过按照可达性规则进行流分析而得出结论的。下面的流分析考虑进来的§**错误！未找到引用源。**节中决定该语句行为的常数表达式的数值，但是非常数表达式的可能值并未被考虑。换句话说，为了控制流分析，某类型的非常数表达式往往会被认为可以去该类型中的任何值。

请看下面的例子

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

if 语句中的布尔表达式是一个常数表达式因为运算符==的两个操作数都是常数。该常数表达式在编译是会被取值，它的值为假，因此 Console.WriteLine 语句对于会被认为是不可到达的。但是如果变量 i 变成一个局部变量，如下所示

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

Console.WriteLine 调用语句将是可达的，即使它本身并未被真的执行。

函数子句中的块语句往往总是被认为是可达的。通过依次评估块中的每一条语句，任何语句的可达性都可以被确定。

下面的代码段中

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

第二个 Console.WriteLine 语句的可达性是这样确定的：

- 首先由于 method F 中的块是可以到达的，所有第一个 Console.WriteLine 语句是可达的。
- 其次，由于第一个 Console.WriteLine 语句是可达的，它的结尾是可达的。
- 再次，由于第一个 Console.WriteLine 语句的结尾是可达的，if 语句是可达的。
- 最后，由于 if 语句中的布尔表达式值为真，第二个 Console.WriteLine 语句是可达的。

有两种情况会使程序语句因为可达性发生错误：

- 由于 switch 语句中不允许任意分支之间发生跳转，所以可能会使该 switch 语句分支语句在可达性检测时出错。如果此错误发生，很典型的情况就是缺少 break 语句。

## 8.2 块

块语句把一系列语句包装成一个语句。块的一般格式如下：

```
block:
    { statement-listopt }
```

块语句包括一个可选的如§**错误！未找到引用源。**节中的语句列表，它们被包围在一对花括号中。假如此语句列表被省略，我们称块为空。

块中可以出现§错误！未找到引用源。节中的声明语句。在块中声明的变量或者常数有效区间到块结尾为止。

在块当中，某一表达式中出现的某名称含义必须保持一致(详见§7.5.2.1节)。

块是按照下面方式被执行的：

- 如果块空，程序跳转到块的结尾。
- 如果块非空，程序会依次执行语句列表当中的每一语句。当程序控制到达语句列表结尾时，它将被转移到块的结尾。

如果块本身是可以到达的，那么块中的语句列表也是可以到达的。

如果块为空或者块中语句列表的结尾可以到达，那么该块的结尾也可到达。

### 8.2.1 语句列表

语句列表包括一系列依次书写的语句。它可以在会或者 switch 语句中出现（分别详见§错误！未找到引用源。节和§错误！未找到引用源。节）。其一般格式如下：

```
statement-list:
    statement
    statement-list statement
```

语句列表都是从第一条语句开始执行的。当执行到该语句的结尾时，程序将会继续执行下一条语句。当程序执行到列表的最后一条语句时，程序控制也被转移到该列表的结尾。

如果下列所述各项条件中的任意一项满足，语句列表中的语句为可到达的：

- 语句为列表中的第一条语句，而且语句列表本身可到达。
- 语句列表中的上一语句的结尾可到达。
- 当语句为带标号的语句时，标号语句被一条可到达的 goto 语句引用。

如果语句列表中最后一条语句的结尾是可到达的，那么该列表的结尾也可到达。

### 8.3 空语句

空语句不进行任何操作，其一般格式如下：

```
empty-statement:
    ;
```

在需要语句但又不进行任何操作的时候我们使用空语句。

执行空语句时只需把程序控制转移到语句的结尾。这样如果空语句是可到达的，那么它的结尾也是可以到达的。

在书写一个空的 while 语句时，我们可以用到空语句：

```
bool ProcessMessage() {...}
void ProcessMessages() {
    while (ProcessMessage());
}
```

同样，在块语句的“}”之前，我们可以通过使用一个空语句来声明一个标号：

```

void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}

```

## 8.4 标号语句

标号语句允许一条语句使用标号前缀。标号语句可以以块的形式出现，但不能以嵌套语句的形式出现。其格式如下：

*labeled-statement:*  
*identifier* : *statement*

标号语句通过给定一个标识符 *identifier* 来声明它的标号。标号的有效区间为它所在的块内,包括此块的嵌套块。不允许有两个名字相同的标号的有效区间相互重叠。

标号可以在它的有效区间内被 goto 语句引用（详见§**错误！未找到引用源。**节）。这意味着 goto 语句可以在块内或由块内到块外完成程序控制的转移，但不能把程序控制转移到子块中。

标号声明有自己的规则，与别的标识符无关。详见下面的例子

```

int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}

```

上面的例程中 x 作为参数和标号两次出现，这是符合语法规则有效的。

执行一条标号语句时，就是执行标号后面的语句。

另外，除了通常情况下控制流提供的可达性之外，假如一个标号被一条可达到的 goto 语句引用，那么我们认为此标号语句也是可达到的。

## 8.5 声明语句

声明语句声明一个局部常量或变量。声明语句可以在块中出现，当是不允许在嵌套语句中出现。其格式如下：

*declaration-statement:*  
*local-variable-declaration* ;  
*local-constant-declaration* ;

### 8.5.1 局部变量声明

局部变量声明语句声明一个或多个局部变量。其格式如下：

*local-variable-declaration:*  
*type* *variable-declarators*  
*variable-declarators:*  
*variable-declarator*  
*variable-declarators* , *variable-declarator*



*variable-declarator:*  
*identifier*  
*identifier* = *variable-initializer*

*variable-initializer:*  
*expression*  
*array-initializer*

被声明的局部变量的类型由声明语句引入的变量类型决定。在类型之后跟有一个变量声明符清单，其中的每一项都引入一个新的变量。每个变量声明符包括一个标识符，一个可选的赋值符号“=”以及一个变量初始器，它会给出变量的初始制。

局部变量的值是通过在表达式中使用简单名称获得的（请参见§7.5.2节），局部变量的值可以使用赋值语句修改（参见§错误！未找到引用源。节）。局部变量在每一次取值时必须被明确赋值（参见§错误！未找到引用源。节）。

变量的有效区间从它自身的标识符出现开始到它的声明语句所在的块结尾为止。在局部变量的有效区间内，不能再次声明另外的拥有同样名字的变量或者常量。

在一条局部变量声明语句中声明多个变量等价于多条每次只声明一个同类型局部变量的局部变量声明语句。不仅如此，变量声明语句中的变量初始器就好像在变量声明语句中插入的一条赋值语句。

请看下面的例子

```
void F() {
    int x = 1, y, z = x * 2;
}
```

与下面语句完全相当：

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

### 8.5.2 局部常量声明

一个局部常量声明语句声明一个或者多个局部常量。其格式如下：

*local-constant-declaration:*  
 const *type* *constant-declarators*

*constant-declarators:*  
*constant-declarator*  
*constant-declarators* , *constant-declarator*

*constant-declarator:*  
*identifier* = *constant-expression*

被声明的局部常量的类型由声明语句引入的常量类型决定。在类型之后跟有一个常量声明符清单，其中的每一项都引入一个新的常量。每个常量声明符包括一个标识符，一个赋值符号“=”以及一个给出常量数值的常量表达式（参见§错误！未找到引用源。节）。

局部常量声明语句中的类型和常数表达式必须遵从有关常量成分声明的规则（参见§10.3节）。

局部常量的值是通过在表达式中使用简单名称获得的（请参见§7.5.2节）。

常量的有效区间从声明开始到它的声明语句所在的块结尾为止。在局部常量的有效区间内，不能再次声明另外的拥有同样名字的变量或者常量。

## 8.6 表达式语句

表达式语句对列出的表达式求值。又表达式计算出的数值，无论多少都被丢弃。其格式如下：

```
expression-statement:
    statement-expression ;

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression
```

并不是所有表达式都允许以语句的形式出现。尤其是象  $x + y$  和  $x = 1$  这样的没有 side - effect 的表达式，它们只是用来求值。

表达式语句计算它列出的表达式的值，之后将程序的控制转移到表达式语句的末尾。

## 8.7 选择语句

选择语句依据一个控制表达式的计算值从一系列可能被执行的语句选择出要执行的语句。其格式如下：

```
选择语句：
    if 语句
    switch 语句
```

### 8.7.1 if 语句

if 语句依据括号中的布尔表达式选择相关语句执行。其格式如下：

```
if-statement:
    if ( boolean-expression ) embedded-statement
    if ( boolean-expression ) embedded-statement else embedded-statement

boolean-expression:
    expression
```

else 分支与最近的 if 语句构成一个 if - else 对。If 语句并不一定必须有 else 分支。这样如下所示的 if 语句

```
if (x) if (y) F(); else G();
```

等价于下面的语句

```

if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}

```

if 语句按照下面的方式执行：

- 求布尔表达式的值（参见§错误！未找到引用源。节）。
- 如果布尔表达式为真，程序将跳转到分语句的第一个嵌套语句执行。当程序执行到此嵌套语句的结尾，程序控制将转移到 if 语句的结尾。
- 如果布尔表达式的值为假，而且分语句中有 else 分支，程序跳转到分语句的第二个嵌套语句执行。当程序执行到此嵌套语句的结尾时，程序控制将转移到 if 语句的结尾。
- 如果布尔表达式的值为假，而且分语句中没有 else 分支，程序控制将转移到 if 语句的结尾。

假如 if 语句本身可到达而且它的布尔表达式值为真，那么 if 语句的第一个嵌套语句可到达。

假如 if 语句本身可到达而且它有第二个嵌套语句并且它的布尔表达式值为假，那么 if 语句的第二个嵌套语句可到达。

假如 if 语句的任意一个嵌套语句的结尾可到达，那么 if 语句的结尾也可到达。

另外，假如一个没有 else 分支的 if 语句本身可到达而且它的布尔表达式值为假，那么 if 语句的结尾可到达。

### 8.7.2 Switch 语句

分支语句依据控制表达式的值选择执行相关的语句。其格式如下：

```

switch-statement:
    switch ( expression ) switch-block

switch-block:
    { switch-sectionsopt }

switch-sections:
    switch-section
    switch-sections switch-section

switch-section:
    switch-labels statement-list

switch-labels:
    switch-label
    switch-labels switch-label

switch-label:
    case constant-expression :
    default :

```

switch 语句包括关键字 switch 和其后的括号表达式（我们称之为 switch 表达式）和 switch 块。Switch 块包含有零个或者多个由括号包围的 switch 分支。每个 switch 分支又由一个或者多个 switch 标号和跟在它们后面的语句列表组成（详见§错误！未找到引用源。节）。

Switch 语句的控制类型由 switch 表达式决定。如果 switch 表达式的类型为 sbyte, byte, short, ushort, int, uint, long, ulong, char, string 或者枚举型, 那么这就是 switch 语句的控制类型。否则, 必须有一个用户自定义隐式转换语句 (参见§错误!未找到引用源。节) 把 switch 表达式的类型转换为下述控制类型类型中的一种: sbyte, byte, short, ushort, int, int, long, ulong, char, string。如果没有这样的一条转换语句或者有多条这样的转换语句存在, 将产生编译错误。

每一个 switch 分支的常数表达式必须取得某个可被隐式转换成 switch 语句控制类型的数值。如果同一个 switch 有两个或两个以上的 switch 分支的常量表达式取得相同的值, 那么编译时会出错。

每一个 switch 语句最多只能有一个 default 标号分支。

Switch 语句时按照如下方式执行的:

- 首先计算出 switch 表达式的值并转换到控制类型。
- 如果 switch 表达式的值等于某一个 switch 分支的常量表达式的值, 那么程序控制跳转到这个 case 标号后的语句列表中。
- 如果 switch 表达式的值无法与 switch 语句中任何一个 case 常量表达式的值匹配而且 switch 语句中有 default 分支, 程序控制会跳转到 default 标号后的语句列表中。
- 如果 switch 表达式的值无法与 switch 语句中任何一个 case 常量表达式的值匹配而且 switch 语句中没有 default 分支, 程序控制会跳转到 switch 语句的结尾。

如果一个 switch 语句的语句列表决定结尾是可到达的, 将会产生编译错误。这就是我们所说的“无失败(no fall through)”规则。请看下面的例子

```
switch (i) {
    case 0:
        CaseZero();
        break;
    case 1:
        CaseOne();
        break;
    default:
        CaseOthers();
        break;
}
```

此 switch 语句是正确有效的, 因为没有任意 switch 分支的结尾是可达的。与 C 和 C++ 不同, 程序控制不能在相邻 switch 分支中转移, 请看下面的例子

```
switch (i) {
    case 0:
        CaseZero();
    case 1:
        CaseZeroOrOne();
    default:
        CaseAny();
}
```

这个 switch 语句是错误的。当程序要在不同的 switch 分支间跳转时必须在跳离的 switch 分支中加入 goto case 语句或者 goto default 语句。请看下面的例子:

```

switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}

```

switch 语句可以有多个 switch 分支，如下面例程所示。

```

switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
    CaseTwo();
    break;
default:
    CaseTwo();
    break;
}

```

这个 switch 语句是合法的。此例子并未违背 “no fall through” 规则因为在 case2 分支和 default 分支属于 switch 语句中的同一个分支。

“无失败”规则可以防止象 C 和 C++ 语言中的 break 语句被忽略的常见错误。同样因为这个规则，可以在不影响 switch 语句功能的情况下重新排列其中的 switch 分支。如同下面的例程中，即使我们调换几个 switch 分支的顺序，整个 switch 语句的功能没有丝毫变化。

```

switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}

```

典型 switch 分支的语句列表往往是以 break 或 goto case 或者 goto default 语句作为结尾，但是也有可能某些结构使得这些语句列表的结尾无法到达。比如，布尔表达式恒为真的 while 循环语句就永远无法执行到它的结尾。同样，throw 或 return 语句总是跳转程序控制而无法执行到自身的结尾。下面的 switch 语句是正确的。

```

switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}

```

一个 switch 语句的控制类型可以是字符串类型的，如下面例程所示：

```

void DoCommand(string command) {
    switch (command.ToLower()) {
        case "run":
            DoRun();
            break;
        case "save":
            DoSave();
            break;
        case "quit":
            DoQuit();
            break;
        default:
            InvalidCommand(command);
            break;
    }
}

```

就像§**错误！未找到引用源。**节中的字符串相等操作符，switch 语句是区分大小写的，只有 switch 表达式的值与 case 分支常数的值完全匹配，此 case 标号后的语句才会被执行。正如上面的例子所示，可以通过把 switch 表达式字符串和 case 分支常量都转换为小写的方法把此 switch 语句变成大小写区分的。

当一个 switch 语句的控制类型为字符串类型时，null 可以作为 case 分支常量。

Switch 会可以包含有声明语句（参见§**错误！未找到引用源。**节）。Switch 块中局部变量和常量的有效区间从它们被声明开始到 switch 块结束为止。

在 switch 块中，表达式中使用的使用的名字的含义必须始终保持一致（参见§7.5.2.1节）。

如果 switch 语句是可到达的而且下面各项中的一项或多项被满足，那么某一 switch 分支的语句列表也是可到达的：

- switch 表达式为非常数值。
- Switch 表达式的值为常数且等于该 switch 分支的 case 标号常量。
- Switch 表达式的值为常量但不等于任何一个 switch 分支的 case 标号常量，而且此 switch 语句含义 default 分支标号分支。
- 某个 switch 标号被一条可到达的 goto case 或 goto default 语句引用

如果满足下列各项中的一项，则我们认为 switch 语句的结尾是可以到达的：

- switch 语句包含有一条可到达的跳出 switch 语句的 break 语句。
- switch 语句可到达且 switch 表达式的值为非常量并且 switch 语句中不含有 default 分支。
- switch 语句可到达且 switch 表达式的值为常量但不与语句中的任一 case 标号常量匹配并且 switch 语句中不含有 default 分支。

## 8.8 重复语句

重复语句反复执行某一嵌套语句。

*重复语句：*  
*while 语句*  
*do 语句*  
*for 语句*  
*foreach 语句*

### 8.8.1 while 语句

while 依据其表达式的值条件执行它的嵌套语句零次或者多次。其格式如下：

```
while-statement:
    while ( boolean-expression ) embedded-statement
```

while 语句是按下面方式执行的：

- 计算其自己的布尔表达式（参见§错误！未找到引用源。节）。
- 如果布尔表达式值为真，程序执行 while 语句的嵌套语句。当程序执行到嵌套语句的结尾（或许为一条 continue 语句），程序将重写会到 while 语句的开头。
- 如果布尔表达式值为假，程序跳转到 while 语句的结尾。

在 while 语句的嵌套语句当中，break 语句（参见§错误！未找到引用源。节）可以使程序跳转到 while 语句的结尾（终止嵌套语句的执行），continue 语句（参见§错误！未找到引用源。节）也可以使程序跳转到嵌套语句的结尾（以便开始下一次嵌套语句的执行）。

如果 While 语句是可到达的而且它的布尔表达式值不恒等于假，则 while 语句的嵌套语句可到达。

如果下列各项中至少有一项被满足，那么 while 语句的结尾为可到达的：

- 存在跳出 while 语句的可到达的 break 语句。
- while 语句本身可到达而且它的布尔表达式值不恒为真。

### 8.8.2 do 语句

do 语句根据其布尔表达式的值有条件的执行它的嵌套语句零次或者多次。其格式如下：

```
do-statement:
    do embedded-statement while ( boolean-expression ) ;
```

do 语句是按下面方式执行的：

- 首先执行其嵌套语句
- 当程序执行到嵌套语句的结尾时（有可能是因为 continue 语句），计算 do 语句布尔表达式的值（参见§错误！未找到引用源。节）。如果其值为真，程序跳转到 do 语句的开头。否则，程序跳转到 do 语句的结尾。

在 do 语句的嵌套语句当中，break 语句（参见§错误！未找到引用源。节）可以使程序跳转到 do 语句的结尾（终止嵌套语句的执行），continue 语句（参见§错误！未找到引用源。节）也可以使程序跳转到嵌套语句的结尾（以便开始下 do 语句的执行）。

如果 While 语句是可到达的而且它的布尔表达式值不恒等于假，则 while 语句的嵌套语句可到达。

如果下列各项中至少有一项被满足，那么 do 语句的结尾为可到达的：

- 存在跳出 do 语句的可到达的 break 语句。
- 嵌套语句的结尾可到达而且 do 语句的布尔表达式值不恒为真。

### 8.8.3 for 语句

for 语句首先计算一系列初始表达式的值，接下来当条件成立时，执行其嵌套语句，之后计算重复表达式的值并根据其值决定下一步的操作。其格式如下：

```

for-statement:
    for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) 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-list , statement-expression

```

for 语句的初始条件部分如果存在，可能为一个局部变量声明语句和（参见§**错误！未找到引用源。**节）或者一系列用逗号分割的表达式（参见§**错误！未找到引用源。**节）。此局部变量的有效区间从它被声明开始到嵌套语句结束为止。有效区间包括 for 语句执行条件部分和 for 语句重复条件部分。

For 语句中如果有 for 语句执行条件部分，那么它必然式一个布尔表达式（参见§**错误！未找到引用源。**节）。

For 语句中如果有 for 语句重复条件部分，它有一系列用逗号分割的表达式构成（参见§**错误！未找到引用源。**节）。

For 语句按照下面的方式被执行：

- 如果存在 for 语句初始条件部分，其中所列的变量声明语句或表达式被依次执行。这一步只执行一遍。
- 如果存在 for 语句执行条件部分，计算此布尔表达式的值。
- 如果不存在 for 语句重复条件部分或者该部分表达式的值为真，程序执行 for 语句的嵌套语句。如果程序执行到嵌套语句的结尾（有可能式执行了 continue 语句）而且存在 for 语句重复条件部分，计算其中表达式的值，and then another iteration is preformed，并返回到上面的步骤计算 for 语句执行条件部分布尔表达式的值。
- 如果存在 for 语句执行条件部分但其布尔表达式的值为假，程序跳转到 for 语句的结尾。

在 for 语句的嵌套语句当中，break 语句（参见§**错误！未找到引用源。**节）可以使程序跳转到 for 语句的结尾（终止嵌套语句的执行），continue 语句（参见§**错误！未找到引用源。**节）也可以使程序跳转到嵌套语句的结尾（以便开始下 for 语句的执行）。

如果下列各项中至少有一项被满足，那么 for 语句的嵌套语句为可达到的：

- for 语句可到达而且不存在 for 语句执行条件部分。
- for 语句可到达而且 for 语句的执行条件部分布尔表达式值不恒为假。

如果下列各项中至少有一项被满足，那么 for 语句的结尾为可达到的：

- 存在跳出 for 语句的可达到的 break 语句。
- for 语句可到达而且 for 语句的执行条件部分布尔表达式值不恒为真。



### 8.8.4 foreach 语句

foreach 语句列举出一个集合(collection)的所有元素，并执行关于集合中每个元素的嵌套语句。Foreach 语句格式如下所示。其格式如下：

*foreach-statement:*  
 foreach ( *type identifier in expression* ) *embedded-statement*

foreach 语句括号中的 *type* 和 *identifier* 用来声明该语句的重复变量。重复变量相当于一个只读的局部变量，它的有效区间为整个嵌套语句内。在 foreach 语句执行过程中，重复变量代表着当前操作针对的集合中相关元素。假如嵌套语句试图对重复变量赋值或者把重复变量当作 ref 或者 out 参数传递，都会产生编译错误。

Foreach 语句的表达式类型必须为集合(collection)类型（正如下面定义的那样），必要存在显式转换（参见§**错误！未找到引用源。**节）将集合的元素类型转换为重复变量的类型。

如果所有下列各项均成立，则我们认为类型 C 为集合(collection)类型：

- C 包含一个 public 类型的实例方法，其内有返回值类型为结构体类型或类类型或者接口类型的 signature GetEnumerator()。
- E 包含一个 public 类型的实例方法，其内有一个返回值类型为布尔的 signature MoveNext()。
- E 包含一个叫做 current 的可读的 public 实例特性。此特性的类型为 collection 的元素类型。

System.Array 类型（参见§12.1.1节）为 collection 类型，而且所有数组类型均由 System.Array 派生而来，所有 foreach 语句中可以出现任意数组类型的表达式。对一维数组，foreach 语句将依次顺序列出数组元素。对于多维数组，foreach 语句将逐行地列举出数字元素。

Foreach 语句按照如下方式运行

- 对集合表达式求值以便获得一个集合(collection)类型的实例。此实例在后续操作中以 C 被引用。假如 C 为 reference 类型并且值为 null，将产生 NullReference 异常。
- 通过方法调用 c.GetEnumerator() 获得一个计数器(enumerator)实例。该调用返回的计数器(enumerator)存储在一个临时的局部变量中，在后续操作中以 e 被引用。嵌套语句不可以访问该临时变量。假如 e 为 reference 类型而且值为 null，将产生一个 NullReference 异常。
- 计数器(enumerator)被通过方法调用 e.MoveNext() 传递给集合的下一个元素。
- 假如方法调用 e.MoveNext() 的返回值为真，将执行下列各项操作：
  - 可以通过 e.Current 求得当前计数器(enumerator)的数值，并通过显式转换（参见§**错误！未找到引用源。**节）将其类型转换为重复变量的类型。转换结果保存在重复变量中，可以被嵌套语句访问。
  - 程序控制转移到嵌套语句。当程序执行到嵌套语句结尾时（有可能是执行了 continue 语句），从上面步骤开始再一次执行 foreach 语句。
- 如果 e.MoveNext() 的返回值为假，程序控制转移到 foreach 语句结尾。

在 foreach 语句的嵌套语句内，break 语句（参见§**错误！未找到引用源。**节）可以用来将程序控制这样到 foreach 语句的结尾（即终止嵌套语句的重复执行），continue 语句（参见§**错误！未找到引用源。**节）也可以将程序控制这样到 foreach 语句的结尾（进行 foreach 语句的下一执行）。

如果 foreach 语句是可到达的，它其中的嵌套语句也是可到达的。同样，该 foreach 语句结尾也是可到达的。

## 8.9 跳转语句

跳转语句进行无条件跳转。

*跳转语句：*  
*break 语句*  
*continue 语句*  
*goto 语句*  
*return 语句*  
*throw 语句*

jump 将要跳向的位置称为 jump 语句的目标地址。

对于块内的 jump 语句，当目标地址在块外时，我们称 jump 跳出块。一旦 jump 语句跳出块，将无法回到块内。

try 语句的介入会使得 jump 语句的执行变得复杂。如果没有 try 语句，则 jump 语句只是完成跳转到它的目标地址的功能。如果有 try 语句介入，该语句的执行就复杂化了。假如 jump 语句中有包括 finally 块在内的一个或者多个 try 块语句程序控制将首先交给最内层的 try 语句。当程序执行到 finally 块的结尾，程序控制将转移到外一层 try 块语句的 finally 块。依次类推，直到所有 try 块语句的 finally 块都被执行过。请看下面的例子。

```
static void F() {
    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");
}
```

在两重 try 块中的 finally 块语句都被执行以后，程序控制将跳转到 jump 语句的目标地址。

### 8.9.1 break 语句

break 语句跳出包含它的 switch，while，do，for，或 foreach 语句。其格式如下：

*break-statement:*  
 break ;

break 语句的目标地址为包含它的 switch，while，do，for 或 foreach 语句的结尾。假如 break 不是在 switch，while，do，for 或者 foreach 语句的块中，将会发生编译错误。

当有 switch，while，do，for 或 foreach 语句相互嵌套的时候，break 语句只是跳出直接包含它的那个语句块。如果要在多处嵌套语句中完成转移，必须使用 goto 语句（参见§[错误！未找到引用源。](#)节）。

Break 语句无法跳出 finally 块语句（参见§[错误！未找到引用源。](#)节）。当 finally 块语句中出 break 语句时，break 语句目标地址必须在同一个 finally 语句内，否则将产生编译错误。

Break 语句按照下面方式执行：

- 当 break 语句要跳出一个或者多个包含有 finally 块语句的 try 块时，程序总是先转移到最内层 try 块的 finally 块语句。当程序执行到该 finally 块的结尾时，在转移的外一层 try 块的 finally 块语句。依此类推，直到所有 try 块的 finally 语句都被执行过为止。
- 程序控制跳转到 break 语句目标地址。

由于 break 语句为无条件的跳转到它的目标地址，所以 break 语句的结尾是不可到达的。

### 8.9.2 continue 语句

continue 语句重新开始新一次包含它的 while，do，for 或者 foreach 语句的执行。其格式如下：

```
continue-statement:
    continue ;
```

continue 语句的目标地址为直接包含它的 while，do，for 或者 foreach 语句的嵌套语句结尾。假如 continue 语句不被 while，do，for 或者 foreach 语句包含，将产生编译错误。

当有 while，do，for 或 foreach 语句相互嵌套的时候，break 语句只是适用于直接包含它的那个语句块。如果要在多处嵌套语句中完成转移，必须使用 goto 语句（参见§**错误！未找到引用源。**节）。

continue 语句无法跳出 finally 块语句（参见§**错误！未找到引用源。**节）。当 finally 块语句中出 break 语句时，break 语句目标地址必须在同一个 finally 语句内，否则将产生编译错误。

continue 语句按照下面方式执行：

- 当 continue 语句要跳出一个或者多个包含有 finally 块语句的 try 块时，程序总是先转移到最内层 try 块的 finally 块语句。当程序执行到该 finally 块的结尾时，在转移的外一层 try 块的 finally 块语句。依此类推，直到所有 try 块的 finally 语句都被执行过为止。
- 程序控制跳转到 continue 语句目标地址。

由于 continue 语句为无条件的跳转到它的目标地址，所以 continue 语句的结尾是不可到达的。

### 8.9.3 goto 语句

goto 语句跳转到由它后面标号注释的语句。其格式如下：

```
goto-statement:
    goto identifier ;
    goto case constant-expression ;
    goto default ;
```

goto *identifier* 语句的目标地址由标识符标注的语句。如果函数中不存在名为标识符的标号或者 goto 语句在标号的有效区间外，都会产出编译错误。

goto case 的目标地址为它所在 switch 语句中与之匹配的 case 分支的语句列表。如果 goto case 语句没有包含在 switch 语句中，或者 case 常量表达式不能被隐式转换（参见§6.1节）成直接包含的 switch 语句的 governing 类型，或者 switch 语句中不含有同 goto case 语句相匹配的 case 分支，将产生编译错误。

goto default 的目标地址为它所在 switch 语句（参见§**错误！未找到引用源。**节）中与之匹配的 default 分支的语句列表。如果 goto default 语句没有包含在 switch 语句中，或者 switch 语句中不含有同 goto default 语句相匹配的 default 分支，将产生编译错误。

goto 语句无法跳出 finally 块语句（参见§**错误！未找到引用源。**节）。当 finally 块语句中出 break 语句时，break 语句目标地址必须在同一个 finally 语句内，否则将产生编译错误。

goto 语句按照下面方式执行：

- 当 goto 语句要跳出一个或者多个包含有 finally 块语句的 try 块时，程序总是先转移到最内层 try 块的 finally 块语句。当程序执行到该 finally 块的结尾时，在转移的外一层 try 块的 finally 块语句。依此类推，直到所有 try 块的 finally 语句都被执行过为止。
- 程序控制跳转到 goto 语句目标地址。

由于 goto 语句为无条件的跳转到它的目标地址，所以 goto 语句的结尾是不可到达的。

#### 8.9.4 return 语句

return 语句从它所在的子函数返回该子函数的调用者。其格式如下：

```
return-statement:
    return expressionopt ;
```

不带表达式的 return 语句只有在不需要计算数值的子函数中出现，也就是说一个返回类型为 void 的方法，属性或索引的 set 访问程序，构造函数或析构函数。

带表达式的 return 语句只能用在计算数值的函数成员中，也就是说一个非 void 返回类型的方法，属性或索引的 get 访问程序，或用户定义的操作符。必须存在一个隐式转换（参见§6.1节）把表达式的类型转换成子函数的返回类型。

不允许在 finally 块语句（参见§**错误！未找到引用源。**节）中出现 return 语句。

return 语句按照下面方式执行：

- 当 return 语句跟有表达式时，先求这个表达式的值，再通过隐式转换将此值转换成子函数的返回类型。转换的结果作为返回值传递给调用者。
- 当 return 语句被一个或者多个包含有 finally 块语句的 try 块时，程序总是先转移到最内层 try 块的 finally 块语句。当程序执行到该 finally 块的结尾时，在转移的外一层 try 块的 finally 块语句。依此类推，直到所有 try 块的 finally 语句都被执行过为止。
- 程序控制跳转到 return 语句所在函数的调用者。

由于 return 语句为无条件的跳转到它的目标地址，所以 return 语句的结尾是不可到达的。

#### 8.9.5 throw 语句

throw 语句抛出一个异常。其格式如下：

```
throw-statement:
    throw expressionopt ;
```

跟有表达式的 throw 语句通过求解表达式的值抛出一个异常。表达式必须得到一个类类型 System.Exception 的数值或者由 System.Exception 派生出的类类型的数值。如果此表达式的数值为 null，throw 语句将会抛出一个 NullReference 异常。

不跟有表达式的 Throw 语句只再 catch 块中出现。它 re-throws 当前正被 catch 处理的表达式。

由于 throw 语句为无条件的转移程序控制，所以 throw 语句的结尾是不可到达的。

当一个异常被抛出是，程序控制会转移到 try 语句中第一 catch 语句，它会处理此异常。从异常被抛出开始到程序控制转移到相应的异常处理器的过程称为异常的传播。异常的传递会自动重复以下步骤直到找到匹配的异常处理器为止。在下面的描述中，抛出点指的是异常被抛出的位置。

- 在当前的函数中，每一个包含异常抛出点的 try 语句都会被检测到。对于任意一个语句 S，从最内层的 try 语句开始到最外层的 try 语句结束，将会执行下列各步骤：
- 如果 S 的 try 语句包含有异常抛出点或者 S 有一条或者多条 catch 语句，程序将会根据 catch 语句的出现次序依次检测直到找到匹配的异常处理器为止。第一条找出该异常类型或者其父类型的 catch 语句就是匹配的语句。同意的 catch 语句同所有类型的异常相匹配。如果找到匹配的 catch 语句并且将程序控制转移到该 catch 块语句则异常传递完成。
- 否则，如果 S 的 try 块语句或者 catch 块语句包含有异常抛出点而且 S 语句包含一个 finally 块语句，程序控制将转移到该 finally 语句。如果 finally 块语句抛出另外一个异常，当前异常的处理终止。否则，当程序执行到 finally 语句的结尾时，继续处理当前异常。
- 当当前的子函数调用中没有异常处理器，函数调用将被终止。上面步骤将被重复执行并在该子函数调用者的函数调用语句处产生一个异常抛出点。
- 如果异常处理终止了当前线程或者进程的所有函数调用，表明该线程或者进程没有匹配的异常处理器，该线程或进程将按照实现定义的方式自行终止。

## 8.10 try 语句

try 语句提供一种在块语句执行过程中捕获异常的机制。Try 语句可以进一步指定某一个在程序控制离开 try 语句后执行的块语句。其格式如下：

```
try-statement:
    try block catch-clauses
    try block finally-clause
    try block catch-clauses finally-clause

catch-clauses:
    specific-catch-clauses general-catch-clauseopt
    specific-catch-clausesopt general-catch-clause

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause

specific-catch-clause:
    catch ( class-type identifieropt ) block

general-catch-clause:
    catch block

finally-clause:
    finally block
```

有三种形式的 try 语句

- 跟有一个或者多个 catch 块语句的 try 块语句。
- 跟有一个 finally 块语句的突然一块语句。
- 跟有一个或者多个 catch 块语句和一个 finally 语句的 try 块语句。

当 catch 语句指定一个类类型，此类必须为 System.Exception 或者由 System.Exception 派生出的类型。

当一个 catch 语句同时指定了类类型和标识符时，即声明了一个同名同类型的异常变量。异常变量相当于一个只读的局部变量，它的有效区间为 catch 块语句内部。在 catch 块语句执行过程中，异常变量代表当前正被处理的异常。如果 catch 试图对异常变量赋值或者把异常变量当作 ref 和 out 参数传递，都将产生编译错误。

除非 catch 语句包括异常变量名，否则无法在 catch 块中访问异常对象。

如果 catch 既没有指定异常类型又没有指定异常变量名，此 catch 语句称为通用 catch 语句。Try 语句只能存在一条通用 catch 语句，而且它必须是 try 块中的最后一条 catch 语句。通用 catch 语句格式如下：

```
catch {...}
```

它等价于

```
catch (System.Exception) {...}
```

如果某一 catch 语句指定的类型同它之前的 catch 语句指定的类型一致或者由此类型派生而来，都会发生错误。由于 catch 语句是依照它们出现的次序被检查以确定某异常的处理器，所有如果无此限制，将有可能出现无法到达的 catch 语句。

在 catch 块内，不带表达式的 throw 语句（参见§**错误！未找到引用源。**节）可能会重复抛出正被 catch 块处理的异常。

对于 break, continue 和 goto 语句，如果它们试图跳出 finally 块，都是错误的。在 finally 块中出现的 break, continue 或 goto 语句，它们的目标地址必须在 finally 块语句内，否则将产生编译错误。

Finally 块语句中不允许出现 return 语句。

Try 语句是按照下面方式执行的：

- 程序控制转移到 try 块
- 当程序控制到达了 try 块的结尾时：
  - 如果 try 语句中存在 finally 块，执行此 finally 块语句。
  - 程序控制转移到 try 语句的结尾。
- 如果在 try 块语句的执行过程中，有异常被传递到该 try 语句：
- 将依照 catch 语句出现的次序确定与此异常匹配的异常处理器。第一个指定异常类型或其父类型的 catch 语句就是匹配的 catch 语句。通用的 catch 语句同所有异常类型匹配。如果找到此匹配的 catch 语句：
  - 如果此 catch 语句声明了异常变量，异常对象将被赋值给异常变量。
  - 程序控制转移到此匹配的 catch 块语句。
  - 当程序控制到达 catch 块的结尾时：
    - 如果 try 块中存在 finally 块语句，执行此 finally 块语句。
    - 程序控制转移到 try 语句的结尾。
  - 如果在 try 块语句的执行过程中，有异常被传递到该 try 语句：

- 如果 try 块中存在 finally 块语句，执行此 finally 块语句。
- 异常被传递到直接包含此 try 块的 try 语句。
- 如果 try 语句中不存在 catch 语句或者没有与异常匹配的 catch 语句：
  - 如果 try 块中存在 finally 块语句，执行此 finally 块语句。
  - 异常被传递到直接包含此 try 块的 try 语句。

在程序控制离开一个 try 块时，该块中 finally 块语句中的语句都会被执行。无论是程序正常执行或者是执行了 break, continue 或 goto 语句，或者是把异常传递到 try 块语句之外，都是这样的。

如果在 finally 块语句的执行过程中抛出了一个异常，异常将被传递到下一个直接包围此 try 块的 try 块语句。如果此时，另外一个异常正被传递，则此异常丢失。关于异常的传递请详见 throw 语句中进一步讨论（参见§**错误！未找到引用源。**节）。

如果一个 try 语句是可到达的，那么它其中的 try 块也是可到达的。

如果一个 try 语句是可到达的，那么它其中的 catch 块也是可到达的。

如果一个 try 语句是可到达的，那么它其中的 finally 块也是可到达的。

当下述两项均成立时，try 块语句的结尾是可到达的：

- try 块的结尾可到达或者至少一条 catch 块语句结尾可到达。
- 如果存在 finally 块语句，此块语句的结尾可到达。

## 8.11 checked 和 unchecked 语句

checked 和 unchecked 语句用来控制整型算是操作和转换操作的溢出检查上下文。

*checked-statement:*  
checked block

*unchecked-statement:*  
unchecked block

checked 语句引起在选中的上下文中对 *block* 中的所有表达式进行计算，而 unchecked 语句引起在非选中上下文中对 *block* 中的所有表达式进行计算。

Checked 和 unchecked 语句等价与节中的 checked 和 unchecked 操作符，除了它们是针对块而不是表达式的。

## 8.12 lock 语句

lock 语句首先获得指定对象的互斥锁，之后执行一条语句，而后解锁。语句格式如下：

*lock-statement:*  
lock ( *expression* ) *embedded-statement*

lock 语句的表达式必须取得一个*引用类型*的值。lock 语句中的表达式不能进行隐式的包装转换（参见§**错误！未找到引用源。**节），所有如果此表达式取得一个*数值类型*的值将是错误的。

Lock 语句格式如下

lock (x) ...

x 是一个*引用类型*的表达式。上面写法除了只对 X 求了一次值外等价于下面写法

```
System.Threading.Monitor.Enter(x);
try {
    ...
}
finally {
    System.Threading.Monitor.Exit(x);
}
```

System.Threading.Monitor 的方法 Enter 和 Exit 为实现定义的。

对于类中 system.Type 的对象可以很方便地用作该类中静态方法的互斥锁。请看下面的例子：

```
class Cache
{
    public static void Add(object x) {
        lock (typeof(Cache)) {
            ...
        }
    }
    public static void Remove(object x) {
        lock (typeof(Cache)) {
            ...
        }
    }
}
```



## 9. 名称空间

C#程序用名称空间来组织。名称空间用在程序的“内部”组织系统，也用在“外部”组织系统 - 一种表现输出到其他程序的程序元素。

使用指示使得使用名称空间变得非常容易。

### 9.1 编译单元

一个编译单元定义了一个源文件全部的结构。一个编译单元由没有或多个后面跟着没有或多个名称空间成员声明的使用指示组成。

*compilation-unit:*  
*using-directives<sub>opt</sub> namespace-member-declarations<sub>opt</sub>*

一个 C#程序包括一个或多个编译单元，每个都包含在分立的源文件中。当 C#程序被编译，所有的编译单元都一起处理。这样，编译单元可以互相依靠，有可能出现循环的形式。

编译单元的使用指示影响那个编译单元的名称空间成员声明，但是不会对其他编译单元有影响。

一个程序的每个编译单元的名称空间成员声明把成员放到一个单独的声明域中，称为全局名称空间。例如：

```
File A.cs:
    class A {}

File B.cs:
    class B {}
```

两个编译单元合成一个单独的全局名称空间，在这种情况下声明了两个有完全名称的 A 和 B。因为两个编译单元指向同一个声明域，如果每个都包含一个有相同名称的成员的声明就会产生错误。

### 9.2 名称空间声明

一个名称空间声明由关键词 `namespace`，跟着一个名称空间名称和主体，最后有一个可选的分号组成。

*namespace-declaration:*  
*namespace qualified-identifier namespace-body ;<sub>opt</sub>*

*qualified-identifier:*  
*identifier*  
*qualified-identifier . identifier*

*namespace-body:*  
*{ using-directives<sub>opt</sub> namespace-member-declarations<sub>opt</sub> }*

一个名称空间声明也许会在编译单元中作为最高级别的声明发生，或者在另一个名称空间声明中作为一个成员声明。当名称空间声明在一个编译单元中作为最高级别声明发生时，名称空间变为全局名称空间的一个成员。当名称空间声明在另外一个名称空间声明中发生时，内部的名称空间变为外部名称空间的成员。在另一种情况下，名称空间的名称在名称空间中必须是唯一的。

名称空间是隐含的 `public` 类型并且名称空间的声明可以包含任何访问修饰符。

在一个名称空间体中，可选的使用指示把其他名称空间或类型的名称引入，并允许他们被直接引用，从而替代通过有效的名称。可选的名称空间成员声明把成员归为名称空间的声明域中。注意所有使用指示必须在成员声明前出现。

名称空间声明的有效标识符可以使一个单独的标识符或是一个由“.”符号分开的标识符序列。下面的形式允许程序定义一个嵌套名称空间，而不用从词汇上嵌套许多名称空间声明。例如，

```
namespace N1.N2
{
    class A {}
    class B {}
}
```

从语义上与下面相同。

```
namespace N1
{
    namespace N2
    {
        class A {}
        class B {}
    }
}
```

名称空间是开放的，而两个有相同完全有效的名称的名称空间声明被放到相同的声明域中(§3.1)。在例子中

```
namespace N1.N2
{
    class A {}
}
namespace N1.N2
{
    class B {}
}
```

上面的两个名称空间声明指向同一个声明域，在这种情况下，要用完全有效名称 N1.N2.A 和 N1.N2.B 对两个类进行声明。因为两个声明指向同一个声明域，所以如果每个所包含的成员的声明有相同的名称，就会是一个错误。

### 9.3 使用指示

使用指示使得在其他名称空间中定义的名称空间或类型的使用变得容易。使用指示影响名称空间和类型名称和简单名称的名称分析过程，但是不像声明，使用指示不把新成员放到它们所使用的编译单元或名称空间的基本声明域中。

```
using-directives:
    using-directive
    using-directives using-directive

using-directive:
    using-alias-directive
    using-namespace-directive
```

一个使用别名指示(§9.3.1)为一个名称空间和类型指定别名。

一个使用名称空间指示(§9.3.2)引入名称空间的类型成员。

使用指示的范围扩展到它的立即包含编译单元或名称空间主体的名称空间成员声明之上。使用指示的范围不包括与它等同的使用指示。这样，等同的使用指示不会互相影响，而它们书写的先后顺序并不重要。

### 9.3.1 使用别名指示

使用别名指示引入一个标识符，它作为直接包含的编译单元或名称空间主体中的名称空间或类型的别名。

```
using-alias-directive:
    using identifier = namespace-or-type-name ;
```

在一个包含使用别名指示的编译单元或名称空间中的成员声明中，被使用别名指示引入的标识符可以用来引用给定的名称空间或类型。例如：

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using A = N1.N2.A;
    class B: A {}
}
```

这里，在名称空间 N3 中的成员声明中，A 是 N1.N2.A 的别名，而 N3.B 从类 N1.N2.A 中派生。可以通过为 N1.N2 创建别名 R 并且引用 R.A 来获得相同的效果：

```
namespace N3
{
    using R = N1.N2;
    class B: R.A {}
}
```

使用别名指示的标识符在直接包含使用别名指示的编译单元或名称空间的声明域中必须是唯一的。例如：

```
namespace N3
{
    class A {}
}
namespace N3
{
    using A = N1.N2.A;    // Error, A already exists
}
```

这里 N3 已经包含成员 A，所以另一个使用别名指示要使用那个标识符就是一个错误。同样，在相同的编译单元或名称空间主体重的两个或多个使用别名指示用相同的名称来声明别名，也是错误的。

使用别名指示使得一个在编译单元或名称空间主体中的别名变得可用，但是它不把任何新成员介绍到基本的声明域中。换句话说，使用别名指示不是传递而是影响它所在的编译单元或名称空间主体。在例子中

```
namespace N3
{
    using R = N1.N2;
}
```

```
namespace N3
{
    class B: R.A {}           // Error, R unknown
}
```

引入 R 的使用别名指示的范围扩展到它所在地名称空间主体重的成员声明，而 R 在第二个名称空间声明中使密封的。然而，把使用别名指示放在包含的编译单元中使得别名在所有名称空间声明中都变得可用：

```
using R = N1.N2;
namespace N3
{
    class B: R.A {}
}
namespace N3
{
    class C: R.A {}
}
```

就像规则成员一样，使用别名指示引入的名称被嵌套范围中的有相似名称的成员所隐藏。在例子中

```
using R = N1.N2;
namespace N3
{
    class R {}
    class B: R.A {}           // Error, R has no member A
}
```

因为 R 指向 N3.F 而不是 N1.N2，所以 B 的声明中对 R.A 的引用造成了错误。

使用别名指示的书写顺序并不重要，而被使用别名指示引用的名称空间和类型名称的分析既不被使用别名指示自己影响也不被其他直接包含的编译单元或名称空间主体中的使用指示所影响。换句话说，使用别名指示的名称空间或类型名称就像直接包含编译单元或名称空间主体没有使用指示一样被隔离。在下面的例子中

```
namespace N1.N2 {}
namespace N3
{
    using R1 = N1;           // OK
    using R2 = N1.N2;        // OK
    using R3 = R1.N2;        // Error, R1 unknown
}
```

最后一个使用别名指示是错误的，因为它没有被第一个使用别名指示影响。

使用别名指示可以为任何名称空间和类型创建一个别名，包括它出现的名称空间和在这个名称空间中任何嵌套的名称空间或类型。

通过别名访问名称空间或类型可以产生与通过它所声明的名称访问名称空间或类型相同的结果。换句话说，给出

```
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 和 R2.A 是完全相同的，并且都指向完全有效名称为 N1.N2.A 的类。

### 9.3.2 使用名称空间指示

使用名称空间指示把名称空间中包含的类型引入到直接包含编译单元或名称空间主体，使得每种类型的标识符可以无限制地使用。

*using-namespace-directive:*  
 using namespace-name ;

在包括使用名称空间指示的编译单元或名称空间主体中的成员声明中，给定名称空间中包含的类型可以直接引用。例如：

```

namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1.N2;
    class B: A {}
}

```

这里，在名称空间 N3 的成员声明中，N1.N2 成员的类型直接可用，而类 N3.B 从类 N1.N2.A 中派生。

使用名称空间指示把给定的名称空间中包含的类型引入，但是并不引入嵌套名称空间。在例子中

```

namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1;
    class B: N2.A {}      // Error, N2 unknown
}

```

使用名称空间指示引入包含在 N1 中的类型，但是不是 N1 中的嵌套的名称空间。这样，B 的声明中的对 N2.A 的引用就是错误的，因为范围内没有名为 N2 的成员。

不像使用别名指示，使用名称空间指示可以引入在所包含的编译单元或名称空间主体中标识符已经定义的类型。实际上，被使用名称空间指示引入的名称被装入的编译单元或名称空间主体中有相似名称的成员所隐藏。例如：

```

namespace N1.N2
{
    class A {}
}

```

```

    class B {}
}
namespace N3
{
    using N1.N2;
    class A {}
}

```

这里，在名称空间 N3 中的成员声明中，A 指向 N3.A 而不是 N1.N2.A。

当一个名称空间被在包含有相同名称的类型的相同的编译单元或名称空间主体中引入时，对那个名称的引用就被认为是不明确的。在例子中

```

namespace N1
{
    class A {}
}
namespace N2
{
    class A {}
}
namespace N3
{
    using N1;
    using N2;
    class B: A {}           // Error, A is ambiguous
}

```

N1 和 N2 都包含成员 A，而因为 N3 把 N1 和 N2 都引入了，所以在 N3 中对 A 的引用时错误的。在这种情况下，这个冲突可以通过对 A 的引用的限制，或引入使用详细的 A 的使用别名指示来解决。例如：

```

namespace N3
{
    using N1;
    using N2;
    using A = N1.A;
    class B: A {}           // A means N1.A
}

```

像使用别名指示一样，使用名称空间指示不会把任何新成员放到编译单元和名称空间的主要声明域中，但只是它所在地编译单元或名称空间主体。

被使用名称空间指示引用的名称空间名称按照被使用别名指示引用的名称空间和类型名称的方法进行处理。这样，在相同的编译单元或名称空间主体中的使用名称空间指示并不会互相影响，并且可以按任何顺序书写。

## 9.4 名称空间成员

名称空间成员声明或者是一个名称空间声明 (§9.2) 或者是一个类型声明 (§9.5)。

```

namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations namespace-member-declaration

```

*namespace-member-declaration:*  
*namespace-declaration*  
*type-declaration*

一个编译单元或一个名称空间主体可以包含名称空间成员声明，而这样的声明把新成员引入包含编译单元或名称空间主体的主要声明域中。

## 9.5 类型声明

类型声明是类声明(§10.1)、结构声明(§11)、接口声明(§13.1)、枚举声明(§14.1)或代表声明(§15.1)之一。

*type-declaration:*  
*class-declaration*  
*struct-declaration*  
*interface-declaration*  
*enum-declaration*  
*delegate-declaration*

类型声明可以作为编译单元中的顶级声明或作为名称空间、类或结构中的成员声明出现。

当一个类型 T 的类型声明作为编译单元中的顶级声明出现时，新声明的类型的完全有效名称就是简单的 T。当对类型 T 的类型声明在名称空间、类或类型中出现时，新声明的类型的完全有效名称就是 N.T，这里 N 是包含名称空间、类或结构的完全有效名称。

在一个类或结构中声明的类型被成为嵌套类型(§10.2.6)。

所允许的访问修饰符合对一个类型声明的默认访问是由声明发生的上下文决定(§**错误！未找到引用源。**)：

- 在编译单元或名称空间中声明的类型可以是公共或内部的访问。默认的是内部访问。
- 在类中声明的类型可以是公共、保护的内部、保护的、内部或者是私有访问。默认的是私有访问。
- 在结构中的类型声明可以是公共、内部或私有的访问。默认的是私有的访问。





# 10. 类

类是一个数据结构，包含数据成员（常数、域和事件）、功能成员（方法、属性、索引、操作符、构造函数和析构函数）和嵌套类型。类类型支持继承，一种机制，派生的类可以对基类进行扩展和特殊化。

## 10.1 类声明

类声明是一个类型声明 (§9.5)，它声明了一个新类。

*class-declaration:*  
*attributes<sub>opt</sub> class-modifiers<sub>opt</sub> class identifier class-base<sub>opt</sub> class-body ;<sub>opt</sub>*

一个类声明的组成如下：一系列可选的属性 (§错误！未找到引用源。)，跟着一系列可选的类修饰符 (§10.1.1)，跟着关键词 `class` 和一个作为类的名称的标识符，跟着一个可选的类的基本说明 (§10.1.2)，跟着一个类主体 (§10.1.3)，还可以选择跟一个分号。

### 10.1.1 类修饰符

一个类声明可以包含一个类修饰符序列：

*class-modifiers:*  
*class-modifier*  
*class-modifiers class-modifier*  
*class-modifier:*  
`new`  
`public`  
`protected`  
`internal`  
`private`  
`abstract`  
`sealed`

相同的修饰符在一个类声明中出现多次是错误的。

`new` 修饰符只允许出现在嵌套类中，它指定了一个类通过相同的名称隐藏了一个继承成员。就像在 §10.2.2 中所描述的一样。

`public`、`protected`、`internal` 和 `private` 修饰符控制类的访问能力。根据类声明所在的上下文，这些修饰符中的一些也许不被允许 (§错误！未找到引用源。 )。

`abstract` 和 `sealed` 修饰符在后面的章节中讨论。

#### 10.1.1.1 抽象类

抽象类被用来指出一个类未完成而且只能成为其他类的一个基类。一个抽象类与非抽象类在下面几方面不同：

- 一个抽象类不能被实例化，在一个抽象类上使用 `new` 操作符是错误的。当然可以有编译时类型为抽象的变量和数值，这样的变量和数值必须或者为 `null` 或者包含对从抽象类型派生的非抽象类的实例的引用。

- 一个抽象类可以（但是不需要）包含抽象成员。
- 一个抽象类不能是密封的。

当一个非抽象类从一个抽象类派生，非抽象类必须包括所有抽象成员的实际实现。这样的执行通过覆盖抽象成员来提供。在例子中

```
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 引入一个抽象方法 F。类 B 引入一个附加的方法 G，但是没有提供一个 F。B，因此也必须被声明为抽象。类 C 覆盖 F 并且提供了一个实际的执行。由于 C 中没有突出的抽象成员，所以 C 被允许为非抽象（但是不需要）。

#### 10.1.1.2 封装类

`sealed` 修饰符被用来提供从一个类的派生。如果一个封装类被确定为另一个类的基类，就会发生错误。

封装类也不能是任何抽象类。

`sealed` 修饰符主要用于防止非计划的派生，但是它也使得某些运行时的优化成为可能。另外，因为密封的类被认为不会有任何派生类，就有可能把密封类实例中的虚拟功能成员调用转换为非虚拟调用。

#### 10.1.2 类基础规范

一个类声明可能会包括类基础规范，它定义了类的直接基本类和由类实现的接口。

```
class-base:
    : class-type
    : interface-type-list
    : class-type , interface-type-list

interface-type-list:
    interface-type
    interface-type-list , interface-type
```

##### 10.1.2.1 基类

当一个类类型被包括在类基础中时，它指定了所声明的类的直接基类。如果一个类声明没有类基础，或类基础列表指示接口类型，直接基础类就被假定为 `object`。一个类从直接基类中继承成员，如§10.2.1中所述。

在例子中

```
class A {}
class B: A {}
```

类 A 被称为 B 的直接基类，而 B 被称为从 A 派生。由于 A 没有明确地指定一个直接基类，它的直接基类隐含的是 object。

类类型的直接基类必须至少像类类型自己(§错误！未找到引用源。 )一样可访问。例如，一个公共类从一个私有或内部类派生是错误的。

类类型的直接基类不能是下面类型中的任何一个：System.Array、System.Delegate、System.Enum 或 System.ValueType。

类的基类是直接基类和它的基类。换句话说，基类的集合是直接基类关系中传递的结尾。参考上面的例子，B 的基类是 A 和 object。

除了类 object，每个类都有一个明确的直接基类。object 类没有直接基类并且是所有其他类的最终基类。

当一个类 B 从类 A 中派生时，A 依赖 B 就是一个错误。类 A 直接依赖它的直接基类（如果有）并且直接依赖其中直接嵌套的类（如果有）。给出这个定义，类所依赖的类的完整集合就是直接依赖关系中传递的结尾。

例子

```
class A: B {}
class B: C {}
class C: A {}
```

是错误的，因为这些类循环依赖它们自己。同样地，例子

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

是错误的，因为 A 依赖 B.C(它的直接基类)，它依赖 B（它的直接包含类），而 B 循环依赖 A。

注意一个类不会依赖一个有嵌套的类。在例子中

```
class A
{
    class B: A {}
}
```

B 依赖 A（因为 A 既是它的直接基类又是它的直接包含类），但是 A 不依赖 B（由于 B 既不是 A 的基类也不是 A 的包含类）。因此，例子是有效的。

不可能从一个密封的类派生。在例子中

```
sealed class A {}
class B: A {}           // Error, cannot derive from a sealed class
```

类 B 是错误的，因为它试图从密封的类 A 中派生。

### 10.1.2.2 接口实现

一个类基础规范包括一个接口类型的列表，在这种情况下类被称为实现了给定的接口类型。接口实现将在§13.4中讨论。

### 10.1.3 类主体

类的类主体定义了类的成员。

```
class-body:
    { class-member-declarationsopt }
```

## 10.2 类成员

一个类的成员由被它的泪成员声明引入的成员和从直接基类中继承的成员组成。

```
class-member-declarations:
    class-member-declaration
    class-member-declarations class-member-declaration

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
```

类的成员被分为下面的几种：

- 常数，它代表了与类相关的常数数据 (§10.3)。
- 域，它是类的变量 (§10.4)。
- 方法，它实现了可以被类实现的计算和行为 (§**错误！未找到引用源。**)。
- 属性，它定义了命名的属性和与对这个属性进行读写的相关行为 (§10.6)。
- 事件，它定义了由类产生的公告 (§10.7)。
- 索引，它允许类的实例通过与数组相同的方法来索引 (§10.8)。
- 操作符，它定义了可以被应用于类的实例上的表达式操作符 (§10.9)。
- 实例构造函数，它执行需要对类的实例进行初始化的动作 (§10.10)。
- 析构函数，它执行在类的实例要被永远丢弃前要实现的动作 (§10.11)。
- 静态构造函数，它执行对类本身进行初始化的动作 (§10.12)。
- 类型，它代表位于类中的类型 (§9.5)。

包含可执行代码的成员全部被当作类的功能成员。类的功能成员是类的方法、属性、索引，操作符、构造函数和析构函数。

一个类声明创建一个新的声明域 (§3.1)，而直接被类声明包含的类成员声明把新的成员引入这个声明域。下面的规则应用于磊成员声明：

- 构造函数和析构函数必须与直接包含类有相同的名称。所有其他的成员必须有与直接包含类不同的名称。
- 常数、域、事件或类型的名称必须与在相同类中声明的其他成员的名称不同。
- 方法的名称必须与同一个类中声明的其他非方法的名称不同。另外，方法的签名 (§3.4) 必须与同一个类中声明的所有其他方法不同。
- 一个索引得签名必须与同一个类中声明的所有其他索引的名称不同。
- 一个操作符的签名必须与同一个类中声明的所有其他操作符的签名不同。

一个类的继承成员 (§10.2.1) 很明确不是这个类的声明域的一部分。因此，一个派生类允许用与继承成员（它的作用时隐藏继承成员）相同的名称或签名来声明。

### 10.2.1 继承

一个类从它的直接基类继承成员。继承意味着一个类隐含地包含除了构造函数和析构函数的直接基类的所有成员。一些继承的重要方面如下：

- 继承是传递的。如果 C 从 B 派生，并且 B 从 A 派生，那么 C 继承在 B 中声明的成员同时也继承 A 中声明的成员。
- 一个派生类扩展它的直接基类。一个派生类可以添加同它继承的成员一样的新成员，但是这不能去掉一个继承的成员的定義。
- 构造函数和析构函数不能被继承，但是所有其他成员可以，不管它们声明的可访问性 (§3.3)。然而，更近他们声明的可访问性，被继承的成员在派生类中也许不能被访问。
- 一个派生类可以通过用相同的名称或签名声明一个新的成员的方法隐藏继承的成员 - 它只是在派生类中修改成员的可访问性。
- 一个类的实例包含在类和它的基类中声明的所有实例域的备份，并且存在一个从继承类类型到任何基类类型的隐式转换 (§错误！未找到引用源。)。因此，一个对派生类的引用可以被看作一个对基类实例的引用。
- 一个类可以声明虚拟方法、属性和索引，并且派生类可以覆盖这些功能成员的执行。这使得类可以展示多态行为，此行为被功能成员调用实现，而这个调用从各方面要根据实例的运行时类型，通过这个类型，方法成员被调用。

### 10.2.2 new 修饰符

一个类成员声明可以用与继承成员相同的名称或签名来声明一个成员。当这发生时，派生类成员被称作隐藏了基类成员。隐藏一个继承成员并不被认为是错误的，但是会造成编译器给出警告。为了禁止这个警告，派生类成员的声明可以包括一个 new 修饰符来指出派生成员要隐藏基成员。这个题目将在 §3.5.1.2 中讨论。

如果一个 new 修饰符被包括在一个没有隐藏继承成员的声明中，就会对这种情况给出一个警告。这个警告可以通过去掉 new 修饰符来禁止。

在同一个声明中使用 new 和 override 修饰符是错误的。

### 10.2.3 访问修饰符

一个类成员声明可以为声明访问能力的五个可用类型中的一个(**\$错误！未找到引用源。**)：公共的、保护的、内部保护的、内部的或私有的。除了 `protected internal` 是组合外，指定多于一个访问修饰符是错误的。当一个类成员声明不包括任何访问修饰符，默认得声明的访问能力为私有的。

### 10.2.4 要素类型

在一个成员的声明中引用的类型被称为成员的要素 (Constituent) 类型。可能的要素类型包括：常数、域、属性、事件或索引类型，方法或操作符的返回类型和方法的参数类型，索引，操作符或构造函数。

成员的要素类型必须最少同成员本身一样可访问(**\$错误！未找到引用源。**)。

### 10.2.5 静态和实例成员

类的成员可以是静态成员或是实例成员。通常来讲，一般把静态成员看作属于类而把实例成员看作属于对象 (类的实例)。

当一个域、方法、属性、事件、操作符或构造函数声明包含静态修饰符，它就声明了一个静态成员。另外，一个常数或类型声明隐含地声明了一个静态成员。静态成员有下面的特征：

- 当一个实例成员在形式 `E.M` 的成员访问(**\$7.5.4**)中被引用时，`E` 必须表示一个类型。`E` 表示一个实例是错误的。
- 一个静态域确定一个存储位置。不管类中有多少实例被创建，只会有一个静态域的备份。
- 一个静态功能成员 (方法、属性、索引、操作符或构造函数) 不会在一个指定的实例中操作，而在一个静态功能成员中使用 `this` 也是错误的。

当一个域、方法、属性、事件、索引、构造函数或析构函数不包括 `static` 操作符，它就声明了一个实例成员。一个实例成员有时被称为非静态成员。实例成员有下面的特点：

- 当一个实例成员在形式 `E.M` 的成员访问中(**\$7.5.4**)被引用时，`E` 必须表示一个实例。`E` 表示一个类型是错误的。
- 一个类中的每个实例都包括类的实例域的分立备份。
- 一个实例功能成员 (方法、属性、访问符、索引访问符、构造函数或析构函数) 在类中的一个给定实例中操作，而这个实例就可以通过 `this` 来访问(**\$7.5.7**)。

下面的例子演示了访问静态和实例成员的规则：

```
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
}
}

```

方法 F 介绍了在一会各实例功能成员中，一个简单名称 (§7.5.2) 可以被用来访问实例成员和静态成员。方法 G 介绍了在一个实例功能成员中，通过简单名称访问实例成员是错误的。方法 Main 介绍了在一个成员访问中 (§7.5.4)，实例成员必须通过实例访问，而静态成员必须通过类型访问。

## 10.2.6 嵌套类型

### 问题

我们需要编写这节。

## 10.3 常数

常数是一个类成员，它代表一个常数数据：一个可以在编译时被计算的数据。一个常数声明引入了一个或多个给定类型的常数。

```

constant-declaration:
    attributesopt constant-modifiersopt const type constant-declarators ;

constant-modifiers:
    constant-modifier
    constant-modifiers constant-modifier

constant-modifier:
    new
    public
    protected
    internal
    private

constant-declarators:
    constant-declarator
    constant-declarators , constant-declarator

constant-declarator:
    identifier = constant-expression

```

一个常数声明可以包括一系列属性 (§错误！未找到引用源。)，一个 new 修饰符 (§10.2.2)，和一个四个访问修饰符的有效组合 (§10.2.3)。属性和修饰符可以用于被常数声明声明的所有成员。甚至常数被认为是静态成员，一个常数声明既不需要也不运行一个 static 修饰符。

常数声明的类型指定了被声明引入的成员的类型。这个类型被一个常数说明符列表跟随，每个引入一个新成员。一个常数说明符由下面几部分组成：为成员命名的标识符，跟着一个“=”符号，跟着一个给出成员数值得常数表达式 (§错误！未找到引用源。 )。

在常数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、一个枚举类型活一个引用类型。每个常数表达式必须产生一个目标类型或者可以通过隐含转换 (§6.1) 转换为目标类型的数据。

一个常数类型必须至少同常数本身一样可以访问(**§错误！未找到引用源。**)。

一个常数自己就可以参与一个常数表达式。因此，一个常数可以被用在任何需要常数表达式的结构中。那样的结构的一个例子包括 case 标签，goto case 语句，enum 成员声明，属性和其他常数声明。

就像在**§错误！未找到引用源。**中所描述的一样，一个常数表达式是一个在编译时可以被完全求值得表达式。由于创建一个不同于 string 的引用类型的非空数据的方法是使用 new 操作符，而由于 new 操作符不能用在常数表达式中，所以唯一一个不同于 string 的引用类型地常数的数值就是 null。

当要得到一个常数数据的符号名称，而当数据的类型在一个常数声明中不被允许或一个数据不能在编译时通过一个常数表达式进行计算得到时，就会用一个 readonly 域(**§10.4.2**)来代替。

一声明多个常数的常数声明与单独的常数的多个声明有相同的特征、修饰符和类型。例如：

```
class A
{
    public const double x = 1.0, y = 2.0, z = 3.0;
}
```

等同于

```
class A
{
    public const double x = 1.0;
    public const double y = 2.0;
    public const double z = 3.0;
}
```

只要依赖不是环行的，在相同的程序中常数被允许依赖其他常数。编译器自动排列来对以适当地顺序排列的常数声明求值。在例子中

```
class A
{
    public const int x = B.z + 1;
    public const int y = 10;
}

class B
{
    public const int z = A.y + 1;
}
```

编译器首先对 y 求值，然后对 z 求值，最后对 x 求值，产生数据 10、11 和 12。常数声明可以依赖其他程序中的常数，但是那样的依赖关系只在一个方向可能。对于上面的例子，如果 A 和 B 在不同的程序中声明，A.x 就有可能依赖 B.z，但是 B.z 不能同时依赖 A.x。

## 10.4 域

一个域是一个成员，这个成员代表一个与一个对象或类相关的变量。一个域声明把一个或多个给定类型的域引入

```
field-declaration:
    attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:
    field-modifier
    field-modifiers field-modifier
```



```

field-modifier:
    new
    public
    protected
    internal
    private
    static
    readonly

variable-declarators:
    variable-declarator
    variable-declarators , variable-declarator

variable-declarator:
    identifier
    identifier = variable-initializer

variable-initializer:
    expression
    array-initializer

```

一个域声明可能包括属性集合(§错误！未找到引用源。)、一个 new 修饰符 (§10.2.2)、四个访问修饰符的有效组合 (§10.2.3)、一个静态修饰符 (§10.4.1) 合一个 readonly 修饰符 (§10.4.2)。属性和修饰符可用于由域声明声明的所有成员。

域声明的类型指定了被声明引入的成员的类型。这个类型后面跟着一个变量声明符列表，其中的每一个引入一个新成员。一个变量声明符由下面组成：一个为成员命名的标识符，也许会跟着一个“=”符号和一个给出变量的初始数值得变量初始化程序 (§10.4.4)。

类型的域必须至少和域本身一样可访问 (§错误！未找到引用源。 )。

域的数值在一个表达式中使用一个简单名称 (§7.5.2) 或一个成员访问 (§7.5.4) 来获得。域的数值通过赋值来修改 (§错误！未找到引用源。 )。

一个声明了多个域的域声明与有同样的属性、修饰符和类型的单个域或多个声明等同。例如

```

class A
{
    public static int X = 1, Y, Z = 100;
}

```

等同于

```

class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}

```

#### 10.4.1 静态和实例域

当一个域声明中包含 static 修饰符时，由声明引入的域就是静态域。当没有使用 static 修饰符时，由声明引入的域就是实例域。静态域和实例域是两个被 C# 支持的各种变量 (§5) 中的一种，而在那个时候作为静态变量和实例变量来引用。

一个静态域正确地指定一个存储位置。无论有多少类的实例被创建，这里永远只有一个静态域的备份。当在一个域中声明的类型被加载时，静态域就成为实际存在，并且当它在声明的域中的变量被卸载时，就不再存在了。

一个类的实例包含了类中所有实例域的分立的备份。当一个类的一个新实例被创建，一个实例域就成为实际存在，而当没有对那个实例的引用并且执行了实例的析构函数，它就不再存在了。

当域在形式 `E.M` 的成员访问中被引用，如果 `M` 是静态域，`E` 必须表示一个类型，而如果 `M` 是一个实例域，`E` 必须表示一个实例。

静态和实例成员间的区别将在§10.2.5中讨论。

#### 10.4.2 只读域

当一个域声明包括 `readonly` (只读) 修饰符的时候，对引入声明的域的赋值只能作为声明的一部分或在相同类中的构造函数中发生。特别地，对于一个只读域的赋值只允许在下面的文字中：

- 在引入这个域的变量声明符（通过在声明中引入一个变量初始化函数）中。
- 对于一个实例域，在包含了域声明的类的实例构造函数中或者在一个静态域中，在包含量域声明的类的静态构造函数中。这些也是唯一的文字，在这里把只读域作为一个 `out` 或 `ref` 参数传送是有效的。

试图对一个只读域赋值或在其他文字中把它作为 `out` 或 `ref` 参数传送是错误的。

##### 10.4.2.1 对常数使用静态只读域

当需要一个常数数据的符号名称时，静态只读域是有用处的，但不是在一个 `const` 声明中不允许数据的类型，或在编译时不能通过常数表达式对数据进行计算时。在例子中

```
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` 成员不能作为 `const` 成员声明，因为它们的数值不能在编译时计算。然而，把成员按照静态只读域来声明有相同的效果。

##### 10.4.2.2 常数的形式和静态只读域

常数和只读域有不同的二进制形式的语法。当一个表达式引用一个常数，常数的数值就会在编译时获得，但是当一个表达式引用一个只读域时，域的数值直到运行时都不会被获得。考虑有两个单独程序的应用程序：

```

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}
namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}

```

名称空间 Program1 和 Program2 代表两个分开编译的程序。因为 Program1.Utils.X 是作为静态只读域来声明，Console.WriteLine 语句的数值输出在编译时是不知道的，但是会在运行时获得。因此，如果 X 的数值发生变化而 Program1 被重新编译，那么甚至在 Program2 没有被编译的情况下，Console.WriteLine 语句也将输出新的数值。然而，如果 X 是一个常数，X 的数值就将在 Program2 被编译时获得，并且直到 Program2 被重新编译，都会在 Program1 中保持不被影响。

### 10.4.3 域的初始化

一个域的初始数值是这个类型的域的默认数值(**\$错误！未找到引用源。**)。当一个类被加载，所有静态域被初始化为他们默认得数值，而当一个类的实例被创建，所有实例域都会被初始化为他们的默认数值。在这个默认初始化没有发生前，对域的数值进行观察是不可能的，并且因此 这个域永远不会是“未初始化的”。例子

```

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}

```

产生下面的输出

```
b = False, i = 0
```

因为在类被加载时，b 被自动地初始化为它默认得数值，而当这个类的实例被创建时，i 被自动地初始化为它的默认数值。

### 10.4.4 变量初始化函数

域声明可以包括变量初始化函数。对于静态域，变量初始化函数与在类被加载时执行的赋值语句相应。对于实例域，变量初始化程序与在类的一个实例被创建时执行的赋值语句相应。

例子

```

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";
}

```

```

        static void Main() {
            Test t = new Test();
            Console.WriteLine("x = {0}, i = {1}, s = {2}", x, t.i, t.s);
        }
    }

```

产生下面的输出

```
x = 1.414213562373095, i = 100, s = Hello
```

因为在类被加载时发生对 *x* 的赋值，而类的一个新实例被创建时对 *i* 和 *s* 进行赋值。

在§10.4.3中描述的默认数值初始化在所有域中发生，包括有变量初始化函数的域。因此，当一个域被加载，所有静态域首先被初始化为它们默认的值，并且接下来静态域初始化函数按照文字的顺序被执行。与此类似，当一个类的实例被创建，所有实例域首先被初始化为它们默认的值，然后实例域初始化函数按照文字顺序执行初始化。

对于有变量初始化函数的静态域，可以以它们默认数值状态被查看，虽然这作为一种风格是很不好的。例子

```

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}

```

展示了这个表现。不管 *a* 和 *b* 的循环定义，程序是合法的。它产生下面的输出

```
a = 1, b = 2
```

因为在它们的初始化程序被执行前，静态域 *a* 和 *b* 被初始化为 0(int 的默认数值)。当 *a* 的初始化程序运行时，*b* 的数值是零，而 *a* 被初始化为 1。当 *b* 的初始化程序运行时，*a* 的数值已经是 1，而 *b* 被初始化为 2。

#### 10.4.4.1 静态域初始化

一个类的静态域初始化函数与在进入类的静态构造函数的时候马上执行的赋值序列相应。变量初始化函数按照在类声明中的文字顺序执行。类加载和初始化过程将在§10.12中描述。

#### 10.4.4.2 实例域初始化

一个类的实例域初始化函数与在进入类的实例构造函数的时候马上执行的赋值序列相应。变量初始化函数按照在类声明中的文字顺序执行。类实例创建和初始化过程将在§10.10中描述。

一个实例域的变量初始化函数不能引用被创建的实例。由于变量初始化函数通过简单名称引用任何实例成员是错误的，因此，在一个变量初始化函数中引用 *this* 是错误的。在例子中

```

class A
{
    int x = 1;
    int y = x + 1;    // Error, reference to instance member of this
}

```

*y* 的变量初始化函数是错误的，因为它引用了一个被创建的实例的成员。

## 10.5 方法

一个方法是一个执行可以被任何对象或类实现的计算或行动的成员。方法用方法声明来声明：

```

method-declaration:
    method-header method-body

method-header:
    attributesopt method-modifiersopt return-type member-name ( formal-parameter-listopt )

method-modifiers:
    method-modifier
    method-modifiers method-modifier

method-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    override
    abstract
    extern

return-type:
    type
    void

member-name:
    identifier
    interface-type . identifier

method-body:
    block
    ;
  
```

一个方法声明可以包含一系列属性(§**错误！未找到引用源。**)，一个 new 修饰符(§10.2.2)，四个访问修饰符的一种有效组合(§10.2.3)，static(§10.5.2)、virtual(§10.5.3)、override (§10.5.4)或 abstract (§10.5.5)修饰符中的一个，和一个 extern (§10.5.6)修饰符

方法声明的返回类型指定了被方法计算并返回地数据的类型。如果方法不返回一个数值，那么返回类型就是 void。

成员名称指定了方法的名称。除非方法是一个外部接口成员执行，成员名称就只是一个标识符。对于外部接口成员执行(§**错误！未找到引用源。**)，成员名称由接口类型跟着一个“.”和一个标识符来组成。

可选的形式参数列表指定了方法的参数(§10.5.1)。

返回类型和每个方法的形式参数列表中引用的类型必须至少同方法本身一样可访问(§**错误！未找到引用源。**)。

对于抽象和外部的的方法，方法主体简单地由分号组成。对于其他方法，方法主体由指定在方法被调用时执行的语句组成。

方法的名称和形式参数列表定义了方法的签名 (§3.4)。特别是，方法的签名由它的名称和成员、修饰符合它的形式参数的类型组成。返回类型不是方法签名的一部分，也不是形式参数的名称。

方法的名称必须与所有其他在相同的类中声明的方法的名称不同。另外，方法的签名必须与在同一方法中声明的方法的签名不同。

### 10.5.1 方法参数

方法的参数（如果有）被方法的形式参数列表所声明。

```
formal-parameter-list:
    formal-parameter
    formal-parameter-list , formal-parameter

formal-parameter:
    attributesopt parameter-modifieropt type identifier

parameter-modifier:
    ref
    out
    params
```

形式参数列表由零或多个被逗号分开的形式参数组成。一个形式参数由一个可选的属性集合(§**错误！未找到引用源。**)、一个类型和一个标识符组成。每个形式参数声明了一个有给定名称的给定类型的参数。

一个方法声明为参数和局部变量创建了一个分立的声明域。名称通过方法的形式参数列表和在方法的主体中的局部变量声明引入到这个声明域中。声明域中的所有名称必须是唯一的。这样，一个参数或局部变量与另外的参数或局部变量有相同名称是错误的。

一个方法调用 (§7.5.5.1) 创建了一个方法的形式参数和局部变量的备份，指向那个调用，而调用的参数列表把数值或变量的引用赋给新创建的形式参数。在方法的主体中，形式参数可以被在简单名称表达式中的标识符引用 (§7.5.2)。

这里有四种形式参数：

- 数值参数，它不用任何修饰符声明。
- 引用参数，它用 `ref` 修饰符声明。
- 输出参数，它用 `out` 修饰符声明。
- 参量参数，它用 `params` 修饰符声明。

就像在 §3.4 中所描述的，参数修饰符是方法声明的一部分。

#### 10.5.1.1 变量参数

没有修饰符的变量声明是一个数值参数。一个数值参数与一个局部变量相应，这个局部变量从方法调用中提供的相应赋值来获得它的初始数值。

当一个形式参数是一个数值参数时，方法调用中的相关参数必须是一个类型的表达式，而它可以隐式地转换 (§6.1) 为形式参数类型。

一个方法被允许把新值赋给数值变量。这样的赋值只影响被数值参数表示的局部存储位置 - 它们对在方法调用中的实际参数没有影响。

### 10.5.1.2 引用参数

用一个 `ref` 修饰符声明的参数是一个引用参数。不像数值参数，引用参数不创建新的存储位置。作为替代，一个输出参数与作为方法调用中给定的参数一样代表相同的存储位置。

当一个形式参数是一个引用参数，方法调用中相应的参数必须由下面形式组成：关键字 `ref`，跟着与形式参数类型相同的变量引用 (§ 错误！未找到引用源。 )。一个变量必须在它可以被按照引用参数传递前定义。

在一个方法中，一个引用参数通常被认为什明确定义的。

例子

```
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);
    }
}
```

产生输出

```
i = 2, j = 1
```

对于 `Main` 中的 `Swap` 的调用，`x` 代表 `i` 而 `y` 代表 `j`。这样，这个调用就有把 `i` 和 `j` 的数值交换的效果。

在一个有引用参数的方法中，多个名称代表相同的存储位置是可能的。在例子中

```
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 `F` 的调用为 `a` 和 `b` 都传送了一个引用 `s`。因此，对于那个调用，名称 `s`、`a` 和 `b` 都指向相同的存储位置，并且三个赋值都修改了实例域 `s`。

### 10.5.1.3 输出参数

用 `out` 修饰符声明的参数是一个输出参数。与引用参数相似，输出参数不能创建一个新的存储位置。作为替代，一个输出参数与作为方法调用中给定的参数一样代表相同的存储位置。

当一个形式参数是一个输出参数时，方法调用中相关的参数必须由关键词 `out` 后面跟着与形式参数类型相同的变量引用 (§ 错误！未找到引用源。 )来组成。一个变量在他被作为一个输出参数传送前需要被明确赋值，但是在一个变量作为输出参数被传送的调用之后，此变量被认为已经被明确赋值了。

在一个方法中，就像局部变量，一个输出参数开始被认为没有被赋值，并且必须在它的数值使用前要被明确赋值。

方法的每个输出参数必须在方法返回前被明确赋值。

输出参数典型地被使用于方法中，此方法产生多个返回数值。例如：

```
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);
    }
}
```

这个例子产生下面的输出：

```
c:\windows\System\
hello.txt
```

注意 `dir` 和 `name` 变量在它们被传送到 `SplitPath` 前可以是未被赋值的，并且它们在这个调用后要被明确赋值。

#### 10.5.1.4 参量参数

用 `params` 修饰符声明的参数是参量参数。一个参量参数必须是形式参数列表中的最后一个，而参量参数的类型必须是一个单维数组类型。例如，类型 `int[]` 和 `int[][]` 可以被用作参量参数类型，但是类型 `int[,]` 不能用作这种方式。

一个参量参数使得调用程序可以用两种方法中的一种提供数据。

- 调用程序必须指定一个可以隐式转换 (§6.1) 为形式参数类型的表达式。在这种情况下，参量参数正好与一个数值参数的行为相同。
- 调用程序可以从指定零或多个表达式二者中选择一个，这里每个表达式的类型被隐式地转换为 (§6.1) 形式参数类型的元素类型。在这种情况下，参量参数用一个包含由调用程序提供的数据的形式参量类型的数组进行初始化。

一个方法被允许把新的数值赋给一个参量参数。这样的赋值只影响被参量参数表现的局部存储位置。

例子

```
void F(params int[] values) {
    Console.WriteLine("values contains %0 items", values.Length);
    foreach (int value in values)
        Console.WriteLine("\t%0", value);
}
```



```
void G() {
    int i = 1, j = 2, k = 3;
    F(new int[] {i, j, k});
    F(i, j, k);
}
```

介绍了一个有类型 `int[]` 的参量参数的方法 `F`。在方法 `G` 中，介绍了 `F` 的两个调用。在第一个调用中，`F` 有一个 `int[]` 类型的参数。在第二个调用中，`F` 被三个 `int` 类型的表达式调用。每个调用的输出是相同的：

```
values contains 3 items:
1
2
3
```

一个参量参数可以单独传送给另一个参量参数。在例子中

```
void F(params object[] fparam) {
    Console.WriteLine(fparam.Length);
}

void G(params object[] gparam) {
    Console.WriteLine(gparam.Length);
    F(gparam);
}

void H() {
    G(1, 2, 3);
}
```

方法 `G` 有一个 `object[]` 类型的参量参数。当这个参数被作为一个方法 `F` 的实际参数使用时，它被单独传送而没有修饰符。输出为：

```
3
3
```

例子

```
void F(params object[] fparam) {
    Console.WriteLine(fparam.Length);
}

void G(params object[] gparam) {
    Console.WriteLine(gparam.Length);
    F((object) gparam); // Note: cast to (object)
}

void H() {
    G(1, 2, 3);
}
```

介绍了也可以通过添加一个 `cast` 把参量参数作为一个单独数据传送。输出为：

```
3
1
```

### 10.5.2 静态和实例方法

当一个方法声明中包括一个 `static` 修饰符，方法就被称为是静态方法。当没有使用 `static` 修饰符，方法就被称为实例方法。

一个静态方法不对指定的实例进行操作，而在静态方法中使用 `this` 是错误的。在一个静态方法中包括一个 `virtual`、`abstract` 或 `override` 修饰符也是错误的。

实例方法在一个类的给定实例上操作，并且这个实例可以被作为 `this` 访问 (§7.5.7)。

静态和实例方法间的不同将在 §10.2.5 中讨论。

### 10.5.3 虚拟方法

当一个实例方法声明中包含一个 `virtual` 修饰符时，方法就被称为是一个虚拟方法。当没有 `virtual` 修饰符时，方法被称为是一个非虚拟方法。

对于一个包含 `virtual` 修饰符又包含 `static`、`abstract` 或 `override` 之一的修饰符是错误的。

一个非虚拟方法的执行是不变的：不管方法在它被声明的类的实例中还是在派生类中的实例中被调用，执行都是相同的。相反，虚拟方法的执行可以被派生类改变。改变一个继承虚拟方法的过程是一个覆盖的方法 (§10.5.4)。

在一个虚拟方法调用中，所调用的实例的运行时类型决定要调用的实际方法。在一个非虚拟方法调用中，实例的编译时类型是决定因素。在更精确的条件中，当一个名为 `N` 的有一个参数列表 `A` 的方法在一个有编译时类型 `C` 和运行时类型 `R` 的实例中被调用（这里 `R` 既不是 `C`，也不是一个从 `C` 中派生的类）时，调用按照下面执行：

- 首先，重载分析被应用于 `C`、`N` 和 `A`，来从在 `C` 中声明或被 `C` 继承的方法集合中选择一个指定方法 `M`。这在 §7.5.5.1 中描述。
- 然后，如果 `M` 是一个非虚拟方法，`M` 就被调用。
- 另外，`M` 是一个虚拟方法，大多数 `M` 的与 `R` 有关的派生执行被调用。

对于每个在一个类中声明或被这个类继承的虚拟方法，这里存在与那个类有关的方法多数是派生实现的。与类 `R` 有关的虚拟方法的多数派生执行由下面决定：

- 如果 `R` 包含 `M` 的引入虚拟声明，那么它就是 `M` 的最可派生执行。
- 另外，如果 `R` 包含 `M` 的替代，那么它是 `M` 的最可派生执行。
- 另外，`M` 的最可派生执行于 `R` 的直接基类是相同的。

下面的例子演示了在虚拟和非虚拟方法间的不同：

```
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();
    }
}
```

在例子中，`A` 引入了一个非虚拟方法 `F` 和一个虚拟方法 `G`。类 `B` 引入了一个 `new` 非虚拟方法 `F`，这样就隐藏了继承的 `F`，并且覆盖了继承的方法 `G`。例子产生下面的输出：

```
A.F
B.F
B.G
B.G
```

注意语句 `a.G()` 调用 `B.G`，而不是 `A.G`。这是因为实例（为 `B`）的运行时类型，不是实例（为 `A`）的编译时类型，决定了要调用的实际方法执行。

因为允许方法隐藏继承的方法，所以一个类包含有相同签名的虚拟方法是可能的。由于除了最可派生方法，其他方法都被隐藏，因此这不会产生一个不确定问题。在例子中

```
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();
    }
}
```

类 `C` 和 `D` 包含两个有相同签名的虚拟方法：一个被 `A` 引入而一个被 `C` 引入。被 `C` 引入的方法隐藏了从 `A` 继承的方法。这样，`D` 中的覆盖声明覆盖了被 `C` 引入的方法，而 `D` 覆盖被 `A` 引入的方法是可能的。例子产生下面的输出：

```
B.F
B.F
D.F
D.F
```

注意不能通过访问一个 `D` 的实例来调用隐藏的虚拟方法，对实例 `D` 的访问是通过一个最少派生类型进行，而在其中方法没有被隐藏。

#### 10.5.4 覆盖方法

当一个实例方法的声明中包含一个 `override` 修饰符时，方法被称为是一个覆盖方法。一个覆盖方法覆盖一个有相同签名的虚拟方法。然而一个虚拟方法声明引入了一个新方法，覆盖方法声明通过提供一个方法的新执行来使存在的继承的虚拟方法特殊化。

一个覆盖方法声明中包含 `new`、`static` 或 `virtual` 修饰符中的任何一个都是错误的。一个覆盖方法声明可以包含 `abstract` 修饰符。这使得虚拟方法可以被一个抽象方法覆盖。

被 `override` 声明覆盖的方法被称为被覆盖的基本方法。对于在类 `C` 中声明的一个覆盖方法 `M`，被覆盖的基本方法由检测 `C` 的每个基类，用 `C` 的直接基类作为开始并用每个继承的直接基类继续来确定，直到与 `M` 有相同签名的一个可访问的方法被确定为止。为了确定覆盖基本方法，如果它是公共的，如果它是保护的，如果它是内部保护的或者它是内部的并且在同 `C` 相同的程序中声明，这个方法就被认为是可访问的。

除非对于一个覆盖声明来说下面所有的都是 `true`，否则就会发生一个编译时错误：

- 一个被覆盖的基本方法可以像上面一样被确定。
- 被覆盖的基本类型是虚拟的、抽象的或者覆盖的方法。换句话说，被覆盖的基本方法不能是静态的或非虚拟的。
- 覆盖声明和被覆盖的基本方法有相同的声明可访问性。换句话说，一个覆盖声明不能改变虚拟方法的可访问性。

一个覆盖声明可以通过基本访问 (§7.5.8) 访问被覆盖的基本方法。在例子中

```
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()` 调用调用了在 `A` 中声明的方法 `PrintFields`。一个基本访问使得虚拟调用机制变为不可能而简单地把基本方法看作一个非虚拟方法。如果 `B` 中的调用被写为 `((A)this).PrintFields()`，它就会递归地调用在 `B` 中声明的方法 `PrintFields`，而不是在 `A` 中声明的方法。

只有通过包括一个 `override` 修饰符，一个方法才能覆盖另一个方法。在所有其他情况，一个与继承的方法有相同签名的方法完全隐藏所继承的方法。在例子中

```
class A
{
    public virtual void F() {}
}
class B: A
{
    public virtual void F() {}    // warning, hiding inherited F()
}
```

`B` 中的方法 `F` 不包括一个 `override` 修饰符并且因此不覆盖 `A` 中的方法 `F`。而 `B` 中的方法 `F` 隐藏 `A` 中的方法，并且因为声明中不包括 `new` 修饰符，会报告一个警告。

在例子中

```

class A
{
    public virtual void F() {}
}
class B: A
{
    new private void F() {}      // Hides A.F within B
}
class C: B
{
    public override void F() {}  // Ok, overrides A.F
}

```

B 中的方法 F 隐藏了从 A 中继承的虚拟方法 F。由于 B 中的新方法 F 有私有的访问，它的范围只包括 B 的类主体而不会扩展到 C。C 中 F 的声明因此允许覆盖从 A 中继承的 F。

### 10.5.5 抽象方法

当一个实例方法声明包括一个 `abstract` 修饰符，这个方法被称为是一个抽象方法。一个抽象方法隐含的也是一个虚拟方法。

一个抽象方法声明引入一个新的虚拟方法，但是不提供一个方法的执行。作为替代，需要非抽象派生类通过覆盖方法来提供它们自己的执行。因为一个抽象方法不提供实际执行，抽象方法的方法主体就完全由分号组成。

抽象方法声明只在抽象类中被允许 (§10.1.1.1)。

一个抽象方法声明中包含 `static` 或 `virtual` 修饰符是错误的。

在例子中

```

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 类定义了一个可以画出自己的几何形状对象的抽象概念。Paint 方法是抽象的，因为这里没有有意义的默认执行。Ellipse 和 Box 是有形的 Shape 的执行。因为这些类是非抽象的，它们需要覆盖 Paint 方法并且提供一个实际执行。

一个基本访问 (§7.5.8) 引用一个抽象方法是错误的。在例子中

```

class A
{
    public abstract void F();
}

```

```

class B: A
{
    public override void F() {
        base.F();           // Error, base.F is abstract
    }
}

```

因为它引用了一个抽象方法，所以会对 `base.F()` 调用报告一个错误。

一个抽象方法声明允许覆盖一个虚拟方法。这允许一个抽象类来强迫进行派生类中方法的再次执行，并且进行不可得到的方法的原始执行。在例子中

```

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 声明了一个虚拟方法，类 B 用一个抽象方法覆盖了这个方法，而类 C 的覆盖给它自己提供了执行。

### 10.5.6 外部方法

当一个方法声明中包含 `external` 修饰符时，这个方法就被称为一个外部方法。外部方法被在外部执行。因为一个外部方法声明不提供实际执行，外部方法的主体就完全由分号组成。

`extern` 修饰符典型应用于有 `DllImport` 属性 (§19.6) 的关联中，允许外部方法被 DLL（动态链接库）所执行。执行环境可以包含其他机制，由此可以提供外部的的方法。

外部方法声明也包括 `abstract` 修饰符是错误的。当外部方法包括一个 `DllImport` 属性，所声明的方法必须也包括 `static` 修饰符。

这个例子演示了 `extern` 修饰符和 `DllImport` 属性的使用：

```

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreatedDirectory(string name, SecurityAttributes sa);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemovedDirectory(string name);
    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}

```

### 10.5.7 方法主体

一个方法声明的方法主体有块和分号组成。

抽象和外部方法声明不包括一个方法执行，而抽象或外部方法的方法主体完全由分号组成。对于所有其他方法，方法主体是一个包含在方法被调用时执行的语句的块(§错误！未找到引用源。 )。

当一个方法的返回类型是 void 的时候，方法主体中的返回类型 (§错误！未找到引用源。)不允许指定一个表达式。如果一个 void 方法的方法主体地执行正常完成（就是说如果控制流程在方法主体结束的地方完成），方法就会完全返回给调用成员。

当方法的返回类型不是 void，每个方法主体中的 return 语句必须指定一个类型的表达式，这个表达式隐含转换为返回类型。返回数值的方法的方法主体的执行需要用 return 语句结束，它指定了一个表达式，或使用 throw 语句结束，它抛出来一个异常。如果方法主体的执行可以正常完成是错误的，换句话说，在一个返回数值的方法中，控制不允许作为方法主体的结束。

在例子中

```
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;
        }
    }
}
```

返回数值的方法 G 是错误的，因为控制可以作为方法主体的结束。方法 G 和 H 是正确的，因为所有可能执行路径的结束都是一个指定了一个返回数值的 return 语句。

### 10.5.8 方法重载

方法重载分析规则在§7.4.2中描述。

## 10.6 属性

一个属性是一个提供对一个对象或类的特点进行访问的成员。属性的例子包括字符串的长度，字体的大小，一个窗口的标题，一个用户的名称，等等。属性是域的正常延伸 - 都是与类型相关的有名称的成员，并且访问域和属性的语法是相同的。然而，不像域，属性不指示存储位置。作为替代，为了对它们的数值进行读写，属性有访问符来指定要执行的语句。这样，属性为读写对象的属性的相关行为提供了某个机制，并且它们允许那些属性可以被计算。

使用属性声明来声明属性：

```
property-declaration:
    attributesopt property-modifiersopt type member-name { accessor-declarations }
```

```

property-modifiers:
    property-modifier
    property-modifiers property-modifier

property-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    override
    abstract

member-name:
    identifier
    interface-type . identifier

```

属性声明可以包含特点的集合 (§**错误！未找到引用源。**)，一个 `new` 修饰符 (§10.2.2)，四个访问修饰符的有效组合 (§10.2.3)，和 `static` (§10.6.1)，`virtual` (§10.6.2)，`override` (§10.6.3) 或 `abstract` (§10.6.4) 修饰符中的一个。

属性声明的类型指定了被声明引入的属性的类型，而成员名称指定了属性的名称。除非属性是一个外部接口成员执行，否则成员名称就完全是一个标识符。对于一个外部接口成员执行 (§**错误！未找到引用源。**)，成员名称由一个接口名称跟着一个 “.” 和一个标识符组成。

属性的类型必须至少和属性本身一样可访问 (§**错误！未找到引用源。**)。

访问符声明（必须在符号 “{” 和 “}” 中）声明了属性的访问符 (§10.6.5)。访问符指定了与读写属性相关的可执行语句。

即使访问一个属性的语法同访问一个域的是相同的，一个属性也不能作为变量分类。因此，把属性作为一个 `ref` 或 `out` 参数传递是不可能的。

### 10.6.1 静态属性

当一个属性声明中包含 `static` 修饰符，这个属性就被称作静态属性。当没有 `static` 修饰符时，这个属性被称为实例属性。

一个静态属性与指定的实例不相关，并且在静态属性的访问符中使用 `this` 是错误的。在一个静态属性中包括 `virtual`、`abstract` 或 `override` 修饰符也是错误的。

一个实例属性与一个类中给定的实例相关，并且这个实例可以被属性访问符中的 `this` (§7.5.7) 访问。

当属性在形式 `E.M` 的成员访问中被引用时，如果 `M` 是一个静态属性，`E` 必须表示一个类型，而如果 `M` 是一个实例属性，`E` 必须表示一个实例。

静态和实例成员间的不同将在 §10.2.5 中讨论。

### 10.6.2 虚拟属性

当一个实例属性声明中包括 `virtual` 修饰符，这个属性就被称作虚拟属性。当没有 `virtual` 修饰符时，这个属性就被称作非虚拟属性。



一个属性声明包含 `virtual` 修饰符同时也包含 `static`、`abstract` 或 `override` 修饰符中的一个是错误的。

非虚拟属性的执行是不变的：不管属性是否在所声明的类的实例或派生类的实例中被访问，执行都是相同的。相反，虚拟属性的执行可以被派生类改变。改变一个继承德虚拟方法的执行的过程被称为属性重载 (§10.6.3)。

对于每个在类或类的继承中声明的虚拟属性，存在一个指向那个类的属性的最可派生执行。一个指向一个类 `R` 的虚拟属性 `P` 的最可派生执行由下面确定：

- 如果 `R` 包含引入的 `P` 的虚拟声明，那么这就是 `P` 的最可派生执行。
- 另外，如果 `R` 包含 `P` 的覆盖，那么这是 `P` 的最可派生执行。
- 另外，`P` 的最可派生执行与 `R` 的直接基类的最可派生执行是相同的。

因为允许属性隐藏继承的属性，对于一个类包含许多有相同签名的虚拟属性来说是可能的。由于除了最可派生属性其他的都被隐藏，因此这不会产生不明确的问题。

### 10.6.3 覆盖属性

当一个实例属性声明包括 `override` 修饰符，这个属性就被称为是一个覆盖属性。一个覆盖属性用相同的签名覆盖一个继承德虚拟属性。然而一个虚拟属性声明引入一个新属性，一个覆盖属性声明通过提供这个属性访问符或访问符的新执行来对存在的继承德虚拟属性进行特殊化。

一个覆盖属性声明包括 `new`、`static`、或 `virtual` 修饰符中的任何一个是错误的。

被覆盖声明覆盖的属性是被覆盖的基本属性。对于一个在类 `C` 中声明的覆盖属性 `P`，被覆盖的基本属性通过对 `C` 的每个基本类进行检查来确定，开始于直接基本类，继续检查每个继承德直接基本类，直到一个与 `P` 有相同签名的可访问属性被确定。出于确定被覆盖的基本属性，如果它是公共的、保护的、内部保护的或内部的，并且在与 `C` 相同的程序中声明，这个属性就被认为是可访问的。

对于一个覆盖声明来说，除非下面所有都是 `true`，就会产生一个编译时错误：

- 一个被覆盖的基本属性可以被按上面所描述的方法确定。
- 被覆盖的基本属性是一个虚拟、抽象或覆盖的属性。换句话说，被覆盖的基本属性不能是静态的或非虚拟的。
- 覆盖声明和被覆盖的基本属性有相同的声明的可访问性。换句话说，一个覆盖的声明不能改变属性的可访问性。
- 覆盖声明和被覆盖的基本声明有相同的访问符。换句话说，不管属性是读写的、只读的或只写的，一个覆盖声明都不能改变。

一个覆盖声明可以通过使用基本访问来访问被覆盖的基本属性 (§7.5.8)。在例子中

```
class A
{
    int pvalue = 1;
    public virtual int P {
        get { return pvalue; }
        set { pvalue = value; }
    }
}
```

```

class B: A
{
    public override int P {
        get { return base.P; }
        set { base.P = value; }
    }
}

```

B 中的 `base.P` 对 A 中声明的 P 属性使用 `get` 和 `set` 访问符。一个基本访问使虚拟调用机制变得不可能，并且把基本属性完全看作是一个非虚拟属性。如果 B 中的访问被写为 `((A)this).P`，就会递归地访问 B 中声明的 P 属性，而不是在 A 中声明的。

只有包括一个 `override` 修饰符，一个属性覆盖才能覆盖另外一个属性。就像 §10.5.4 中所描述的，在这种方法中，属性表现得像方法一样。

#### 10.6.4 抽象属性

当一个实例属性声明包括一个 `abstract` 修饰符，这个属性就被称为是抽象属性。一个抽象属性隐含的也是一个虚拟属性。

一个抽象属性声明引入一个新虚拟属性，但是没有提供属性访问符或访问符的执行。作为替代，非抽象派生类需要为访问符或覆盖的属性的访问符提供它们自己的执行。因为一个抽象属性声明的访问符不提供实际执行，它的访问符主体就完全由分号组成。

抽象属性声明只被允许存在于抽象类中 (§10.1.1.1)。

一个抽象属性声明包括 `static` 或 `virtual` 是错误的。

基本访问 (§7.5.8) 引用一个抽象属性是错误的。在例子中

```

class A
{
    public abstract int P { get; set; }
}

class B: A
{
    public override int P {
        get { return base.P; }      // Error, base.P is abstract
        set { base.P = value; }    // Error, base.P is abstract
    }
}

```

会给 `base.P` 访问报告一个错误，因为它们引用了一个抽象属性。

#### 10.6.5 访问符

属性的访问符声明指定了一个与读写属性相关的可执行语句。

```

accessor-declarations:
    get-accessor-declaration set-accessor-declarationopt
    set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:
    get accessor-body

set-accessor-declaration:
    set accessor-body

```

```

    accessor-body:
        block
    ;

```

访问符声明由一个 get 访问符声明，一个 set 访问符声明或两个一起组成。每个访问符声明由符号 get 或 set 后面跟着一个访问符主体来组成。对于抽象属性，每个被指定的访问符的访问符主体完全由一个分号。对于所有其他访问符，访问符主体是一个指定了在访问符被调用时所执行的语句的块。

一个 get 访问符与有一个属性类型的返回数值的无参数方法相对应。除了赋值的目标，当一个属性在一个表达式中被调用时，属性的 get 访问符就被调用来计算属性的数值 (§7.1.1)。get 访问符的主体与在 §10.5.7 中描述的返回数值的规则的规则相一致。另外，get 访问符主体中的 return 语句必须指定一个可隐含转换为属性类型的表达式。而且一个 get 访问符需要用 return 语句或 throw 语句作为结束，并且控制语句不能作为 get 访问符主体的结束出口。

一个 set 访问符与有一个属性类型的单个数值参数和 void 类型返回的方法相应。set 访问符的隐含参数通常称为 value。当属性作为赋值的对象被引用时，有一个提供了新数据 (§错误！未找到引用源。) 的参数 set 访问符就被调用。set 访问符的主体必须与在 §10.5.7 中描述的 void 方法的规则相一致。特别是，在 set 访问符主体中的 return 语句不允许指定一个表达式。

由于 set 访问符隐含有一个名为 value 的参数，所以 set 访问符中的一个局部变量声明使用那个名称就是错误的。

根据 get 和 set 访问符的存在或不存在，一个属性按下面进行分类：

- 一个既包括 get 访问符也包括 set 访问符的属性被称为读写属性。
- 一个只包括 get 访问符的属性被称为只读属性。一个只读属性被当作赋值的目标是错误的。
- 一个只有 set 访问符的属性被称为只写属性。除了作为赋值的目标，在一个表达式中引用只写属性是错误的。

#### 实现注意事项

在 Microsoft .NET 运行时，当一个类声明了类型 T 的属性 X，在相同的类中声明有下面签名中的一个的方法是错误的：

```

T get_X();
void set_X(T value);

```

Microsoft .NET 运行时为与不支持属性的程序语言兼容而保留这些签名。注意这个限制并不意味着一个 C# 程序可以使用方法语法来访问属性或是使用属性语法来访问方法。它只是表示在相同类中的按照这种样式的属性和方法是相互独立的。

在例子中

```

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
    }
}

```

控制 Button 声明了一个公共属性 Caption。Caption 属性的 get 访问符返回在私有域 caption 中存储的字符串。set 访问符检查这个新数据是否与当前的数据不同，如果是，它就存储新的数据并且对控制进行 repaint 操作。属性通常为下面的样式：get 访问符返回一个存储在私有域中的数值，而 set 访问符更改私有域并且做其他任何需要更新对象的状态的行为。

上面给出 Button 类，下面是一个 Caption 属性使用的例子：

```

Button okButton = new Button();
okButton.Caption = "OK";           // Invokes set accessor
string s = okButton.Caption;       // Invokes get accessor

```

这里，通过把一个数值赋给属性来调用 set 访问符，而通过在一个表达式中引用这个属性来调用 get 访问符。

一个属性的 get 和 set 访问符不是明显的访问符，并且不能单独声明一个属性的访问符。例子

```

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; }
    }
}

```

没有声明一个单独的读写属性。它声明了两个有相同名称的属性，一个时只读的而另一个是只写的。由于在一个相同的类中声明的两个成员不能有相同的名称，这个例子会产生一个编译时错误。

当派生类用与继承属性相同的名称声明了一个属性的时候，派生的属性把继承的读和写的属性都隐藏起来。在例子中：

```

class A
{
    public int P {
        set {...}
    }
}

```

```

class B: A
{
    new public int P {
        get {...}
    }
}

```

B 中的属性 P 隐藏了 A 中的属性 P 的读和写。这样，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

```

对 b.P 的赋值引起报告一个错误，由于 B 中的只读属性 P 隐藏了 A 中的只写属性 P。注意，那个方法可以被用来访问被隐含的属性 P。

与公共域不同，属性提供了对象的内部状态和它的公共接口的区分。注意下面的例子：

```

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; }
    }
}

```

这里，Label 类使用两个 int 域，x 和 y，来存储它的位置。位置既作为 X 和 Y 的属性，也作为类型 Point 的位置属性。在 label 的以后版本中，如果它要变得能更方便地存储 Point 内部的位置，这个变化可以被实行而不影响类的公共接口：

```

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; }
        }
    }

```

如果 *x* 和 *y* 成为 `public readonly` 域，对 `Label` 类进行那样的改变就不可能了。

通过属性给出状态比直接给出域效率低。另外，当一个属性是非虚拟的并且只包含少量代码，执行环境就可以用访问符的实际代码替代对访问符的调用。这个过程被叫做 *inlining*，并且它使得属性的访问与域的访问一样有效率，但防止了属性复杂性的增长。

由于调用一个 `get` 访问符从概念上与读一个域的数据相同，因此对于 `get` 访问符有可观察的副作用是一个不好的程序风格。在例子中

```

class Counter
{
    private int next;
    public int Next {
        get { return next++; }
    }
}

```

`Next` 属性的数据根据这个属性已经被访问过的次数变化。因此，对属性的访问产生了一个可观察到的副作用，并且这个属性要作为一个方法被执行。

`get` 访问符的“无副作用”转换并不意味着 `get` 访问符应该被写成只返回存储在域中的数据。实际上，`get` 访问符通常通过访问多个域或调用方法来计算属性的数值。但是，一个正确设计的 `get` 访问符不会引起对象的声明中可看到的改变的行为。

属性可以被用来延缓源的初始化，直到它第一次被引用。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(File.OpenStandardInput());
            }
            return reader;
        }
    }
    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(File.OpenStandardOutput());
            }
            return writer;
        }
    }
}

```

```

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(File.OpenStandardError());
            }
            return error;
        }
    }
}

```

类 Console 包含三个属性 In、Out 和 Error，它表示标准的输入、输出和错误设备。通过把这些成员作为属性提供，Console 类可以延迟它们的初始化直到它们被使用。例如，在第一个对 Out 属性的引用，在下面

```
Console.Out.WriteLine("Hello world");
```

输出设备的基本的 TextWriter 被创建。但是如果应用程序对 In 和 Error 属性不做任何引用，那么对于那些设备就不会创建任何对象。

## 10.7 事件

一个事件是一个使对象或类可以提供公告的成员。用户可以通过提供事件句柄来为事件添加可执行代码。事件使用事件声明来声明：

```

event-declaration:
    event-field-declaration
    event-property-declaration

event-field-declaration:
    attributesopt event-modifiersopt event type variable-declarators ;

event-property-declaration:
    attributesopt event-modifiersopt event type member-name { accessor-declarations }

event-modifiers:
    event-modifier
    event-modifiers event-modifier

event-modifier:
    new
    public
    protected
    internal
    private
    static

```

一个事件声明既可以是一个事件域声明也可以是事件属性声明。在每种情况中，声明都可以由属性集合 (§错误！未找到引用源。)， new 修饰符 (§10.2.2)，四个访问修饰符的有效组合 (§10.2.3) 和一个静态修饰符 (§10.2.5) 组成。

一个事件声明的类型必须是一个代表类型 (§15)，而那个代表类型必须至少同事件本身一样可访问 (§错误！未找到引用源。 )。

一个事件域声明与一个声明了一个或多个代表类型域的域声明 (§10.4) 相应。在一个事件域声明中不允许有 readonly 修饰符。

一个事件属性声明与声明了一个代表类型属性的属性声明 (§10.6) 相应。除了同时包含 `get` 访问符和 `set` 访问符的事件属性声明，成员名称和访问符声明对于那些属性声明来说都是相同的，并且不允许包含 `virtual`、`override` 和 `abstract` 修饰符。

在包含一个事件成员声明的类或结构的程序文字中，事件成员与代表类型的私有域或属性相关，而这个成员可以用于任何允许使用域或属性的上下文中。

如果一个类或结构的程序文字外面包含一个事件成员声明，这个事件成员就只能被作为 `+=` 和 `-=` 操作符 (§错误！未找到引用源。) 的右手操作数使用。这些操作符被用来为事件成员附加或去掉事件句柄，而这个事件成员的访问操作符控制操作在其中被执行的上下文。

由于 `+=` 和 `-=` 是唯一可以在声明了事件成员的类型外的事件上使用的操作，外部代码可以为一个事件添加或去掉句柄，但是不能通过任何其他途径获得或修改基本事件域或事件属性的数值。

在例子中

```
public delegate void EventHandler(object sender, Event e);
public class Button: Control
{
    public event EventHandler Click;
    protected void OnClick(Event e) {
        if (Click != null) Click(this, e);
    }
    public void Reset() {
        Click = null;
    }
}
```

对使用 `Button` 类中的 `Click` 事件域没有限制。作为演示的例子，这个域可以在代表调用表达式中被检验、修改和使用。类 `Button` 中的 `OnClick` 方法“引起”`Click` 事件。引起一个事件的概念与调用由事件成员表示的代表正好相同 - 因此，对于引起事件没有特殊的语言构造。注意代表的调用是通过检查保证代表是非空后才进行的。

在类 `Button` 的声明外面，成员 `Click` 只能被用在 `+=` 和 `-=` 操作符右手边，如下

```
b.Click += new EventHandler(...);
```

它把一个代表附加到事件 `Click` 的调用列表中，并且

```
b.Click -= new EventHandler(...);
```

它把一个代表从 `Click` 事件的调用列表中删除。

在一个形式为 `x += y` 或 `x -= y` 的操作中，当 `x` 是一个事件成员而引用在包含 `x` 的声明的类型外面发生时，操作的结果就是 `void` (在赋值后与 `x` 的数值相反)。这个规则禁止外部代码直接检查事件成员的基本代表。

下面的例子介绍了事件句柄如何附加到上面的类 `Button` 的实例中：

```
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, Event e) {
    // Handle OkButton.Click event
}
void CancelButtonClick(object sender, Event e) {
    // Handle CancelButton.Click event
}
}

```

这里，构造函数 LoginDialog 创建了两个 Button 实例，并且把事件句柄附加到事件 Click 中。

事件成员是典型域，就像上面的 Button 例子中所示。在每个事件消耗一个域存储的情况是不可接受的，一个类可以声明事件属性来替代事件域，并且使用私有机制来存储基本的代表。（设想在某种情况下，大多数事件都是未处理的，每个事件使用一个域就不能被接受。使用属性而不是域的能力允许开发人员在空间和时间上面取得一个折中方案。）

在例子中

```

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) {...}
    // Set event handler associated with key
    protected void SetEventHandler(object key, Delegate handler) {...}
    // MouseDown event property
    public event MouseEventHandler MouseDown {
        get {
            return (MouseEventHandler)GetEventHandler(mouseDownEventKey);
        }
        set {
            SetEventHandler(mouseDownEventKey, value);
        }
    }
    // MouseUp event property
    public event MouseEventHandler MouseUp {
        get {
            return (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        }
        set {
            SetEventHandler(mouseUpEventKey, value);
        }
    }
}

```

类 Control 为事件提供了一种内部存储机制。方法 SetEventHandler 用一个 key 来与代表数值相关，而方法 GetEventHandler 返回与 key 相关的当前代表。大概基本的存储机制是按照把空代表类型与 key 相关不会有消耗而设计的，因此无句柄的事件不占用存储空间。

#### 实现注意事项

在 Microsoft .NET 运行时，当一个类声明一个代表类型 T 的事件成员 X，相同的类中也声明与下面签名中的一个相同的方法是错误的：

```

void add_X(T handler);
void remove_X(T handler);

```

Microsoft .NET 运行时为与不提供操作符的语言或其他构造来附加和去除事件句柄的语音相兼容，保留了这些签名。注意这些限制并不暗示一个 C# 程序可以用方法语法来附加或去掉事件句柄。它只是表示这种样式的事件和方法在相同的类中相互独立。

当一个类声明了一个事件成员，C# 编译器自动产生前面提到的 add\_x 和 remove\_x 方法。例如，声明

```
class Button
{
    public event EventHandler Click;
}
```

可以被当作

```
class Button
{
    private EventHandler Click;
    public void add_Click(EventHandler handler) {
        Click += handler;
    }
    public void remove_Click(EventHandler handler) {
        Click -= handler;
    }
}
```

编译器还产生一个引用 add\_x 和 remove\_x 方法的事件成员。从一个 C# 程序的观点出发，这种机制只是执行细节，而它们对保留的 add\_x 和 remove\_x 签名没有明显得影响。

## 10.8 索引

一个索引是一个成员，它使得一个对象可以像一个数组一样被索引。索引用索引声明来声明：

*indexer-declaration:*

*attributes*<sub>opt</sub> *indexer-modifiers*<sub>opt</sub> *indexer-declarator* { *accessor-declarations* }

*indexer-modifiers:*

*indexer-modifier*

*indexer-modifiers* *indexer-modifier*

*indexer-modifier:*

new

public

protected

internal

private

virtual

override

abstract

*indexer-declarator:*

*type* this [ *formal-index-parameter-list* ]

*type* *interface-type* . this [ *formal-index-parameter-list* ]

*formal-index-parameter-list:*

*formal-index-parameter*

*formal-index-parameter-list* , *formal-index-parameter*

*formal-index-parameter:*

*attributes*<sub>opt</sub> *type* *identifier*

一个索引声明可以包含属性集合 (§错误！未找到引用源。), new 修饰符 (§10.2.2), 四个访问修饰符的有效组合 (§10.2.3) 和 virtual (§10.8.2)、override (§10.8.3) 或 abstract (§10.8.4) 修饰符之一。

一个索引声明的类型指定了被声明引入的索引得元素类型。除非这个索引是一个外部接口成员执行, 类型就被关键词 this 所跟随。对于一个外部接口成员执行, 类型跟着一个接口类型、一个“.”和关键词 this。不像其他成员, 索引不能有用户定义的名称。

形式索引参数列表指定了索引的参数。一个索引的形式参数列表与某个方法的行参列表相关 (§10.5.1), 除了必须至少指定一个参数外, ref 和 out 参数修饰符不被允许。

一个索引的类型和每个在形式索引参数列表中引用的类型都必须至少和索引本身一样可访问 (§错误！未找到引用源。 )。

访问符声明 ( 必须在 “{” 和 “}” 符号中间 ) 声明了索引的访问符。访问符指定了与读和写索引元素相关的可执行语句。

甚至访问一个索引元素的语法与一个数组元素的都相同, 而一个索引元素不会被按照变量分类。因此, 把一个索引元素作为 ref 或 out 参数传递是不可能的。

一个索引的形式参数列表定义了索引得签名 (§3.4)。特别是, 一个索引的签名由它的形式参数的数量和类型组成。元素类型不是索引签名的一部分, 也不是形式参数的名称。

一个索引得签名必须与所有其他索引得签名不同。

索引和属性在概念上非常相似, 但是在下面几方面不同 :

- 一个属性被它的名称所确定, 然而一个索引被它的签名确定。
- 一个属性通过一个简单名称 (§7.5.2) 或一个成员访问 (§7.5.4) 来访问, 然而一个索引元素通过元素访问 (§7.5.6.2) 来访问。
- 一个属性可以是静态成员, 然而一个索引通常是一个实例成员。
- 一个属性的 get 访问符与一个没有参数的方法相关, 然而一个索引的 get 访问符与一个同索引有相同形式参数列表的方法相关。
- 一个属性的 set 访问符语有一个名为 value 的参数相关, 然而一个索引的 set 访问符与有一个和索引有相同的形式参数列表和一个附加的名为 value 的参数的方法相关。
- 对于一个索引的访问符用和索引参数相同的名称声明一个局部变量是错误的。

除了这些不同, 在 §10.6.5 中的定义可以使用在索引访问符上, 就像属性访问符一样。

#### 执行注意

*在 Microsoft .NET 运行时, 当一个类声明一个有一个形式参数列表 P 类型为 T 的索引时, runtime, when a class declares an indexer of type T with a formal parameter list P, it is an error for the same class to also declare a method with one of the following signatures:*

```
T get_Item(P);
void set_Item(P, T value);
```

*Microsoft .NET 运行时为与不支持属性的程序语言兼容而保留这些签名。注意这个限制并不意味着一个 C# 程序可以使用方法语法来访问属性或是使用属性语法来访问方法。它只是表示在相同类中的按照这种样式的属性和方法是相互独立的。*

下面的例子声明了一个 BitArray 类, 它在一个位阵列中为访问单独的位提供了一个索引。

```

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);
            }
        }
    }
}

```

一个类 BitArray 的实例比相应的 bool[] 消耗更少的存储空间(每个数据只占用一位而不是一个字)，但是它允许和 bool[] 一样操作。

下面的类 CountPrimes 使用 BitArray 和经典的“筛选”算法来计算从 1 到一个最大给定数据间的数：

```

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);
    }
}

```

注意 BitArray 访问元素的语法与 bool[] 的完全相同。

### 10.8.1 索引重载

索引重载分析规则在§7.4.2中描述。

### 10.8.2 虚拟索引

当一个索引声明包含一个 `virtual` 修饰符，这个索引就被称为是一个虚拟索引。当没有 `virtual` 修饰符的时候，索引被称为是非虚拟索引。

对于一个包含 `virtual` 修饰符的索引声明中同时包含 `abstract` 或 `override` 修饰符中的一个是错误的。

一个非虚拟索引的执行是不变得：不管索引是在一个声明了这个索引的类的一个实例中，还是在一个派生类的实例中被访问，执行都是相同的。相反，一个虚拟索引的执行可以被派生类改变。改变这个继承的虚拟索引得执行被称为索引的重载。

因为索引允许隐藏继承的索引，所以一个类包含许多有相同签名的虚拟索引是可以的。

### 10.8.3 覆盖索引

当一个索引声明包含一个 `override` 修饰符，这个索引就被称为是覆盖的索引。一个覆盖索引用相同的签名覆盖一个继承的虚拟索引。然而一个虚拟索引声明引入了一个新索引，一个覆盖索引声明通过提供一个索引访问符或访问符的新执行来把一个存在的继承虚拟索引进行特殊化。

一个覆盖索引声明包含 `new` 或 `virtual` 中的一个修饰符是错误的。

被覆盖声明覆盖的索引被称为被覆盖的基本索引。对于一个类 `C` 中声明的覆盖索引，被覆盖的基本索引通过对 `C` 的每个基本类进行检查来确定，检查从 `C` 的直接基本类开始，接着是每个继承的直接基本类，直到查到一个有相同名称的可访问索引。出于查找被覆盖的基本类的目的，如果它是公共的，是保护的，是内部保护的或是内部的，并且在与 `C` 相同的程序中声明，一个索引就被认为是可访问的。

除非一个覆盖声明的所有下面的条件都满足，就会产生一个编译时错误：

- 一个被覆盖的基本索引可以按照上面所描述的定位。
- 被覆盖的基本索引是一个虚拟、抽象或覆盖索引。换句话说，被覆盖的索引不能是非虚拟的。
- 覆盖声明和被覆盖的基本索引有相同的声明的可访问性。换句话说，一个覆盖声明不能改变一个索引的可访问性。
- 覆盖声明和被覆盖的基本索引有相同的访问符。换句话说，不管索引是读写、只读还是只写，一个覆盖声明都不能改变。

一个覆盖声明可以使用基本访问(§7.5.8)来访问被覆盖的基本索引。在例子中

```
class A
{
    public virtual int this[int index] {
        get { return index; }
    }
}
class B: A
{
    public override int this[int index] {
        get { return base[index] + 1; }
    }
}
```

B 中的 `base[index]` 访问对在 A 中声明的索引使用 `get` 访问符。一个基本访问取消了虚拟调用机制并完全把基本索引当作一个非虚拟索引。如果 B 中的访问被写成 `((A)this)[index]`，就会是一个对 B 中声明的索引得递归访问，而不是在 A 中声明的。

一个索引只有包含一个 `override` 修饰符才能覆盖另外一个索引。按这种方法，索引的行为像方法一样，就像在§10.5.4中描述的一样。

#### 10.8.4 抽象索引

当一个索引声明包含一个 `abstract` 修饰符，这个索引就被称为是一个抽象索引。一个抽象索引隐含的也是一个虚拟索引。

一个抽象索引声明引入一个新的虚拟索引，但是并不为索引得访问符和访问符提供执行程序。作为替代，非抽象派生类需要为索引的访问符或访问符们提供一个执行。因为一个抽象索引声明的访问符不提供实际的执行程序，所以它的访问符主体完全由分号组成。

抽象索引声明只在抽象类中被允许(§10.1.1.1)。

一个抽象索引声明中包含一个 `virtual` 修饰符是错误的。

一个基本访问 (§7.5.8) 引用一个抽象索引是错误的。在例子中

```
class A
{
    public abstract int this[int index] { get; }
}
class B: A
{
    public override int this[int index] {
        get { return base[index] + 1; } // Error, base[index] is abstract
    }
}
```

会为 `base.P` 访问报告一个错误，因为他们引用了一个抽象索引。

#### 10.9 操作符

一个操作符是一个定义理一个可以被用于一个类的实例中的表达式操作符的意义的成员。操作符用操作符声明来声明：

```
operator-declaration:
    attributesopt operator-modifiers operator-declarator block

operator-modifiers:
    public static
    static public

operator-declarator:
    unary-operator-declarator
    binary-operator-declarator
    conversion-operator-declarator

unary-operator-declarator:
    type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of
    +    -    !    ~    ++    --    true    false
```

*binary-operator-declarator:*

*type operator overloadable-binary-operator ( type identifier , type identifier )*

*overloadable-binary-operator:* one of

*+ - \* / % & | ^ << >> == != > < >= <=*

*conversion-operator-declarator:*

*implicit operator type ( type identifier )*

*explicit operator type ( type identifier )*

这里有三种操作符：一元操作符 (§10.9.1)，二元操作符 (§10.9.2)，和转换操作符 (§10.9.3)。

下面的规则适用于所有操作符声明：

- 一个操作符声明必须同时包括一个 `public` 和一个 `static` 修饰符，而不允许包括任何其他修饰符。
- 操作符的参数必须是一个参数数值。一个操作符声明指定 `ref` 或 `out` 参数是错误的。
- 一个操作符的签名必须与在相同类中声明的所有其他操作符的签名不同。
- 在一个操作符声明中引用的所有类型必须至少和操作符本身一样可访问 (§ 错误！未找到引用源。 )。

每类操作符都使用附加限制，就像后面章节中所描述的一样。

像其他成员一样，在基本类中声明的操作符被派生类继承。因为操作符声明通常需要声明操作符的类或结构参与操作符的签名，所以在派生类中声明的操作符不能隐藏在基类中声明的操作符。因此，在一个操作符声明中，不需要也不允许存在 `new` 修饰符。

对于所有的操作符，操作符声明包括一个指定在操作符被调用时执行的语句。操作符的主体必须符合在 §10.5.7 中描述的数值返回规则。

有关一元和二元操作符的附加消息可以在 § 错误！未找到引用源。 中找到。

有关转换操作符的附加消息可以在 § 错误！未找到引用源。 中找到。

### 10.9.1 一元操作符

下面的规则适用于一元操作符声明，这里 `T` 代表包含操作符声明的类或结构类型：

- 一元操作符 `+`, `-`, `!`, 或 `~` 必须使用类型 `T` 的单个参数，并且可以返回任何类型。
- 一元操作符 `++` 或 `--` 必须使用类型 `T` 的单个参数，并且要返回类型 `T`。
- 一元操作符 `true` 或 `false` 必须使用类型 `T` 的单个参数，并且要返回类型 `bool`。

一元操作符的签名由操作符符号 (`+`, `-`, `!`, `~`, `++`, `--`, `true`, 或 `false`) 和单独形式参数的类型组成。返回类型不是一元操作符签名的一部分，也不是形式参数的名称。

一元操作符 `true` 和 `false` 需要成对地声明。如果一个类只声明其中一个操作符而没有声明另一个，就会发生错误。`true` 和 `false` 操作符将会在 § 错误！未找到引用源。 中描述。

### 10.9.2 二元操作符

一个二元操作符必须有两个参数，而且至少其中一个必须是声明操作符的类或结构的类型。一个二元操作符可以返回任何种类类型。

二元操作符的签名由操作符符号 (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, or `<=`) 和两个形式参数组成。返回的类型不是二元操作符签名的一部分，也不是形式参数的名称。

某些二元操作符需要成对声明。对于一对操作符中的每个声明，必须与此对中另外一个操作符的声明相匹配。当它们有相同的返回类型并且对每个参数有相同的类型时，两个操作符声明相匹配。下面是需要成对声明的操作符：

- 操作符== 和操作符!=
- 操作符>和操作符<
- 操作符>=和操作符<=

### 10.9.3 转换操作符

一个转换操作符声明引入了一个用户定义的转换 (§[错误！未找到引用源。](#))，它增加预定义的隐式和显示的转换。

一个包括关键词 `implicit` 的转换操作符声明引入了一个用户定义的隐式转换。隐式转换可能在各种情况下发生，包括功能成员调用，表达式执行和赋值。这将在§6.1中描述。

一个包括关键词 `explicit` 的转换操作符声明引入了一个用户定义的显式转换。显式转换可以在表达式执行中发生，并且将在§[错误！未找到引用源。](#)中描述。

一个转换操作符从一个由转换操作符的参数类型确定的源类型，转换到一个由转换操作符的返回类型确定的目标类型。一个类或结构允许声明从源类型 `S` 到目标类型 `T` 的转换，并保证下面所有都为真：

- `S` 和 `T` 是不同的类型。
- 或者 `S` 或者 `T` 是一个声明操作符的类或结构的类型。
- `S` 和 `T` 都不是 `object` 或接口类型。
- `T` 不是 `S` 的一个基本类，`S` 也不是 `T` 的基类。

从第二个规则可以看出一个转换操作符必须既可以转换为声明这个操作符的类或结构的类型，又可以从它得到。例如，一个类或结构类型 `C` 定义一个从 `C` 到 `int` 和从 `int` 到 `C` 的转换是可以的，但是不能从 `int` 到 `bool`。

不能定义一个预定义转换。因此，转换操作符不允许从进行从 `object` 或到 `object` 的转换，因为从 `object` 到所有其他类型的隐式和显式的转换已经都存在了。同样，由于转换将已经存在，所以转换的源或目标类型都不能是其他的基本类型。

用户定义的转换不允许使用接口类型。这个限制特别保证了在转换到一个接口类型时不会有用户定义的转换发生，并且那个到接口类型的转换，只能在被转换的对象实际上执行指定的接口类型时发生。

转换操作符的签名由源类型和目标类型组成。（注意，这只是返回类型参与签名的唯一形式。）The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) 一个转换操作符的 `implicit` 或 `explicit` 分类不是操作符签名的一部分。因此一个类或结构不能用相同的源和目标类型同时声明一个 `implicit` 和一个 `explicit` 转换。

通常，用户定义的隐式转换应该被设计成不会抛出异常而且不会丢掉信息。如果一个用户定义的转换可以产生一个异常（例如因为源变量超出了范围）或丢掉信息（例如丢掉高位），那么这个转换应该被定义为一个显式转换。

在例子中



```

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，因为它不会抛出一个异常或丢掉信息，但是从 byte 到 Digit 的转换是显式的，因为 Digit 只能表示 byte 所有可能值的子集。

## 10.10 实例构造函数

一个实例构造函数是一个执行需要对一个类的实例进行初始化的行为的成员。构造函数使用构造函数声明来声明：

```

constructor-declaration:
    attributesopt constructor-modifiersopt constructor-declarator block

constructor-modifiers:
    constructor-modifier
    constructor-modifiers constructor-modifier

constructor-modifier:
    public
    protected
    internal
    private

constructor-declarator:
    identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:
    : base ( argument-listopt )
    : this ( argument-listopt )

```

一个构造函数声明可以包括一个属性集合 (§错误！未找到引用源。) 和四个访问修饰符的有效组合 (§10.2.3)。

构造函数声明的标识符必须对声明构造函数的类进行命名。如果指定了任何其他名称，就会发生错误。

一个构造函数的可选得形式参数列表与方法的形式参数列表 (§错误！未找到引用源。) 的规则相同。定义构造函数的签名的形式参数列表 (§3.4)，决定为什么重载分析 (§7.4.2) 在一个调用中选择一个特殊的构造函数。

每个在构造函数的形式参数列表中引用的类型必须至少和构造函数 (§错误！未找到引用源。) 一样可访问。

在执行这个构造函数提供的主体中的语句之前，可选的构造函数初始化函数指定了另外一个要调用的构造函数。这将在 §10.10.1 中描述。

构造函数声明的主体指定了为了对类的一个新实例进行初始化的语句。这与一个有 void 返回类型的实例方法 (§10.5.7) 的主体相应。

构造函数不能被继承，因此，一个类不会有比在类中声明的实际构造函数更多的构造函数。如果一个类不包括构造函数声明，就会自动提供一个默认的构造函数 (§10.10.4)。

构造函数被对象创建表达式 (§7.5.10.1) 并通过构造函数初始化函数调用。

### 10.10.1 构造函数初始化函数

在构造函数的主体中的第一个语句之前，所有构造函数（除了类 object 的构造函数）隐含地都有一个对另外的构造函数的直接调用。要隐式调用的构造函数被构造函数初始化函数确定：

- 一个形式为 `base(...)` 的构造函数初始化函数造成调用直接基类中的构造函数。这个构造函数是通过使用 §7.4.2 中的重载分析规则来选择。候选构造函数的集合由所有在直接基类中声明的可访问构造函数组成。如果候选构造函数的集合是空的，或如果不能找到一个最佳的构造函数，就会发生错误。
- 一个形式为 `this(...)` 的构造函数初始化函数造成调用类自己里面的构造函数。这个构造函数是通过使用 §7.4.2 中的重载分析规则来选择。候选构造函数的集合由所有这个类本身声明的可访问构造函数组成。如果候选构造函数的集合是空的，或如果不能找到一个最佳的构造函数，就会发生错误。

如果构造函数没有构造函数初始化函数，就会隐含地提供一个形式为 `base()` 的初始化函数。因此，一个如下面形式的构造函数声明

```
C(...) { ... }
```

等同于

```
C(...): base() { ... }
```

由构造函数声明的形式参数列表给出的参数的范围包括那个声明的构造函数的初始化函数。因此，一个构造函数初始化函数被允许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) {}
}
```

一个构造函数初始化函数不能访问被创建的实例。因此在一个赋值表达式中通过简单名称引用任何实例成员是错误的。

### 10.10.2 实例变量初始化函数

当一个构造函数没有构造初始化函数或一个形式为 `base(...)` 的构造函数初始化函数，构造函数就就隐含的执行被类中声明的实例域的变量初始化函数指定的初始化。这与赋值序列相关，这些赋值在直接基类构造函数的隐式调用前，在构造函数的入口被直接执行。变量初始化函数按照它们在类声明中出现的文字顺序执行。

### 10.10.3 构造函数执行

可以把一个实例变量初始化函数和一个构造函数初始化函数，看作是自动插在构造函数主体中的第一条语句前。例子

```

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;
    }
}

```

包含了许多变量初始化函数，并且也包含了每个形式（base 和 this）的构造函数初始化函数。这个例子与下面介绍的例子相关，在那里，每条注释指明了一个自动插入的语句（自动插入构造函数调用所使用的语法不是有效的，至少用来演示这个机制）。

```

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;
    }
}

```

注意变量初始化函数被转换为赋值语句，并且那个赋值语句在对基类构造函数调用前执行。这个顺序确保了所有实例域在任何访问实例的语句执行前，被它们的变量初始化函数初始化。例如：

```

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);
    }
}

```

当 new B() 被用来创建 B 的实例时，产生下面的输出：

```
x = 1, y = 0
```

因为变量初始化函数在基类构造函数被调用前执行，所以 x 的数值是 1。可是，y 的数值是 0（int 的默认数值），这是因为对 y 的赋值直到基类构造函数返回才被执行。

#### 10.10.4 默认构造函数

如果一个类不包含任何构造函数声明，就会自动提供一个默认的构造函数。默认的构造函数通常是下面的形式

```
public C(): base() {}
```

这里 C 是类的名称。默认构造函数完全调用直接基类的无参数构造函数。如果直接基类中没有可访问的无参数构造函数，就会发生错误。在例子中

```

class Message
{
    object sender;
    string text;
}

```

因为类不包含构造函数声明，所以提供了一个默认构造函数。因此，这个例子正好等同于

```

class Message
{
    object sender;
    string text;
    public Message(): base() {}
}

```

### 10.10.5 私有构造函数

当一个类只声明了私有的构造函数时，其他类就不能从这个类派生或创建类的实例。私有构造函数通常用在只包含静态成员的类中。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 类提供了一组相关的方法和常数，但没有被例示。因此，它声明了一个单独的私有构造函数。注意至少要必须声明一个私有构造函数来避免自动生成默认的构造函数（它通常有公共的访问性）。

### 10.10.6 可选的构造函数参数

构造函数的 `this(...)` 形式通常用于与实现可选的构造函数参数的关联上。在这个例子中

```
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
    }
}
```

前两个构造函数只是为丢失的参数提供了默认的值。两个都使用了一个 `this(...)` 构造函数的初始化函数来调用第三个构造函数，它实际上做了对新实例进行初始化的工作。效果是那些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");
```

### 10.11 析构函数

析构函数是一个实现破坏一个类的实例的行为的成员。析构函数使用析构函数声明来声明：

*destructor-declaration:*  
*attributes<sub>opt</sub> ~ identifier ( ) block*

一个析构函数声明可以包含一个属性的集合（**错误！未找到引用源。**）。

一个析构函数声明的标识符必须为声明析构函数的类命名，如果指定了任何其他名称，就会发生一个错误。

析构函数声明的主体指定了为了对类的新实例进行初始化而执行的语句。这于一个有 `void` 返回类型的实例方法的主体相关 (§10.5.7)。

析构函数不能被继承。因此一个类不会有比在类中实际声明的更多的析构函数。

析构函数被自动调用，并且不能被显式地调用。当没有任何代码要使用一个实例时，这个实例就满足了使用析构函数的条件。实例的析构函数可以在实例变为符合执行析构函数条件后的任何时候执行，一个继承链中的析构函数按顺序被调用，从最可派生的到最少派生的。

## 10.12 静态构造函数

一个静态构造函数是一个实现对类进行初始化的行为的成员。静态构造函数用静态构造函数声明来声明：

*static-constructor-declaration:*  
*attributes<sub>opt</sub> static identifier ( ) block*

一个静态构造函数可以包含一系列属性 (§错误！未找到引用源。 )。

一个静态构造函数声明的标识符必须为声明了这个静态构造函数的类命名。如果指定了任何其他名称，就会发生错误。

静态构造函数声明的主体指定了为对类进行初始化要执行的语句。它与有 void 返回类型的静态方法的主体相关 (§10.5.7)。

静态构造函数不能被继承。

静态构造函数自动被调用，不能被显式调用。虽然提供了许多约束条件，但是静态构造函数执行的确切时间和顺序是不确定的：

- 一个类的静态构造函数在这个类的任何实例被创建前执行。
- 一个类的静态构造函数在类的任何静态成员被引用前执行。
- 一个类的静态构造函数在它的所有派生类的静态构造函数执行之后执行。
- 一个类的静态构造函数从不会被执行一次以上。

例子

```
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");
    }
}
```

会产生或者是下面的输出：

```
Init A
A.F
Init B
B.F
```

或者是下面的输出：

```
Init B
Init A
A.F
B.F
```

因为静态构造函数执行的确切顺序是不确定的。

例子

```
class Test
{
    static void Main() {
        Console.WriteLine("1");
        B.G();
        Console.WriteLine("2");
    }
}

class A
{
    static A() {
        Console.WriteLine("Init A");
    }
}

class B: A
{
    static B() {
        Console.WriteLine("Init B");
    }

    public static void G() {
        Console.WriteLine("B.G");
    }
}
```

保证会有下面的输出：

```
Init A
Init B
B.G
```

因为类 A 的静态构造函数必须在类 B 的静态构造函数之后执行，B 从 A 中派生。

### 10.12.1 类加载和初始化

可以构造一个循环依赖，以便可以观察到有变量初始化函数的静态域的默认数值状态。

例子

```
class A
{
    public static int X = B.Y + 1;
}

class B
{
    public static int Y = A.X + 1;
```

```
static void Main() {  
    Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);  
}
```

产生输出

X = 1, Y = 2

为了执行方法 Main，系统首先加载类 B。B 的静态构造函数就会计算 Y 的初始值，因为 A.x 的数值被引用，所以它会递归的加载 A。A 的静态构造函数接下来计算 X 的初始值，并且会去取 Y 的默认值，为零。这样 A.x 就被初始化为 1。加载 A 的过程就完成了，返回到对 Y 的初始值的计算，这个计算的结果变为 2。

如果方法 Main 在类中被加载，这个例子就会产生下面的输出

X = 2, Y = 1

由于通常不能确定调用包含此引用的类的顺序，因此在静态域初始化函数中的循环引用应该被避免。



# 11. 结构

## 问题

我们需要编写这节。虽然有一些不同，大多数介绍类的一章中的规则都可以不经修改的使用。这章将指出这些不同。

## 11.1 结构声明

```
struct-declaration:
    attributesopt struct-modifiersopt struct identifier struct-interfacesopt struct-body ;opt
```

### 11.1.1 结构修饰符

```
struct-modifiers:
    struct-modifier
    struct-modifiers struct-modifier
```

```
struct-modifier:
    new
    public
    protected
    internal
    private
```

### 11.1.2 接口

```
struct-interfaces:
    : interface-type-list
```

### 11.1.3 结构体

```
struct-body:
    { struct-member-declarationsopt }
```

## 11.2 结构成员

```
struct-member-declarations:
    struct-member-declaration
    struct-member-declarations struct-member-declaration
```

```
struct-member-declaration:
    class-member-declaration
```

## 11.3 结构例子

### 11.3.1 数据库整数类型

下面的结构 DBInt 实现了一个可以代表 int 类型的数值的完整集合的整数类型，并且加上一个代表一个未知数据的附加状态。有这些特性的类型通常用在数据库中。

```

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 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? new DBInt(-x.value): Null;
    }
    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? new DBInt(x.value + y.value): Null;
    }
    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? new DBInt(x.value - y.value): Null;
    }
    public static DBInt operator *(DBInt x, DBInt y) {
        return x.defined && y.defined? new DBInt(x.value * y.value): Null;
    }
    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? new DBInt(x.value / y.value): Null;
    }
    public static DBInt operator %(DBInt x, DBInt y) {
        return x.defined && y.defined? new DBInt(x.value % y.value): Null;
    }
}

```

```

public static DBBool operator ==(DBInt x, DBInt y) {
    return x.defined && y.defined?
        new DBBool(x.value == y.value): DBBool.Null;
}
public static DBBool operator !=(DBInt x, DBInt y) {
    return x.defined && y.defined?
        new DBBool(x.value != y.value): DBBool.Null;
}
public static DBBool operator >(DBInt x, DBInt y) {
    return x.defined && y.defined?
        new DBBool(x.value > y.value): DBBool.Null;
}
public static DBBool operator <(DBInt x, DBInt y) {
    return x.defined && y.defined?
        new DBBool(x.value < y.value): DBBool.Null;
}
public static DBBool operator >=(DBInt x, DBInt y) {
    return x.defined && y.defined?
        new DBBool(x.value >= y.value): DBBool.Null;
}
public static DBBool operator <=(DBInt x, DBInt y) {
    return x.defined && y.defined?
        new DBBool(x.value <= y.value): DBBool.Null;
}
}

```

### 11.3.2 数据库布尔类型

下面的结构 DBBool 实现了一个有三个数值的逻辑类型。这种类型的可能数值是 DBBool.True, DBBool.False, 和 DBBool.Null, 这里成员 Null 指示了一个未知数据。这样的三值逻辑通常用在数据库中。

```

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.
    int value;
    // Private constructor. The value parameter must be -1, 0, or 1.
    DBBool(int value) {
        this.value = 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;
}
}

```

## 12. 数组

数组是一个包含了一些通过计算出来的标号来访问的变量的数据结构。这些包含在一个数组中的变量通常称为变量的元素，它们都有相同的类型，而这个类型被称为数组的元素类型。

一个数组有一个秩来确定于每个数组元素相关的标号数量。数组的秩也称为数组的维数。一个秩为 1 的数组被称为单维数组，一个秩大于 1 的数组被称为多维数组。

一个数组的每一维都有相应的长度，这个长度是一个大于或等于零的整数。维的长度不是数组类型的一部分，但是更适于当一个数组类型的实例在运行时被创建时建立。一维的长度决定那维的指针的有效范围：对于长度为 N 的维，指针是在从 0 到 N - 1 所包含的范围。一个数组中元素的全部数量是数组中每一维的长度的乘积。如果数组中一个或多个维的长度为零，这个数组就被称为是空的。

数组中元素的类型可以是任何类型，包括数组类型。

### 12.1 数组类型

一个数组类型写为一个非数组类型跟着一个或多个秩指示符：

```

array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    [ dim-separatorsopt ]

dim-separators:
    ,
    dim-separators ,
  
```

非数组类型是任何本身不是数组类型的类型。

一个数组类型的秩由数组类型中最左端的秩指示符给出：一个秩指示符指出数组是一个有一加上秩标识符中符号“,”的数目的秩的数组。

数组类型的元素类型是去掉最左端的标号指示符剩下的类型：

- 形式为  $T[R]$  的数组类型是一个标号为 R 和一个非数组元素类型为 T 的数组。
- 一个形式为  $T[R][R_1] \dots [R_N]$  的数组类型是一个标号为 R 和一个元素类型为  $T[R_1] \dots [R_N]$  的数组。

实际上，秩指示符在最后的非数组元素前被从左到右读。例如类型 `int[][,,][,]` 是一个 int 类型的二维数组的三维数组的单维数组。

有一个标号的数组被称为单维数组。有多于一个标号的数组被称为多维数组，例如二维数组，三维数组，等等。

在运行时一个数组类型的数据可以是 null 或是对那种数组类型的一个实例的引用。

### 12.1.1 System.Array 类型

`System.Array` 类型是所有数组类型的抽象基本类型。存在一个从任何数组类型到 `System.Array` 类型的隐含引用转换(§错误！未找到引用源。)，和一个从 `System.Array` 到任何数组类型的显式引用转换(§错误！未找到引用源。)。注意 `System.Array` 本身就是一个数组类型。另外，它是一个类类型，所有数组类型都从它派生。

在运行时，类型 `System.Array` 的数值可以是 `null` 或者对任何数组类型的实例的引用。

### 12.2 数组创建

数组实例被数组创建表达式 (§7.5.10.2) 或被引入一个数组初始化函数的域或局部变量声明所创建 (§12.6)。

当一个数组实例被创建，每一维的标号和长度就被建立了，并用于实例的完整声明周期。换句话说，不可能改变一个已存在实例的标号，也不能对它的维数重新设置。

一个被数组创建实例创建的数组实例通常是一个数组类型。`System.Array` 类型是一个抽象类型，它不能被例示。

被数组创建表达式创建的数组元素通常被初始化为它们的默认数值 (§错误！未找到引用源。 )。

### 12.3 数组元素访问

数组元素用形式为 `A[I1, I2, ..., IN]` 的元素访问表达式 (§7.5.6.1) 访问，这里 `A` 是一个数组类型的表达式，而每个 `Ix` 是一个 `int` 类型的表达式。一个数组元素访问的结果是一个变量，也就是由标号选择的数组元素。

一个数组的元素可以通过使用一个 `foreach` 语句 (§错误！未找到引用源。 ) 来列举。

### 12.4 数组成员

每个数组类型都继承由 `System.Array` 类型声明的成员。

### 12.5 数组协方差

对于任何两个引用类型的 `A` 和 `B`，如果存在从 `A` 到 `B` 的隐式引用转换 (§错误！未找到引用源。 ) 或显式引用转换 (§错误！未找到引用源。 )，那么也存在从数组类型 `A[R]` 到数组类型 `B[R]` 的引用转换，这里 `R` 是一个任意给定的标号指示符（但是并不是在所有数组类型中都相同）。这个关系被称为数组协方差。协方差具体意味着一个数组类型 `A[R]` 的数据实际可以作为一个对数组类型 `B[R]` 的实例的引用，提供了一个从 `B` 到 `A` 的隐式引用转换。

因为数组协方差，对一个引用类型数组的元素的赋值包括一个运行时的检查，它保证被赋给数组元素的数值真的是所允许的类型 (§错误！未找到引用源。 )。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);
}
}

```

在方法 Fill 中对 array[i] 的赋值隐含地包括一个运行时检查，它保证被 value 引用的对象或是 null，或者是一个与 array 的实际元素类型一致的类型实例。在 Main 中，前两个对 Fill 的调用成功了，但是第三个调用在执行第一个向 array[i] 赋值的时候，抛出一个 ArrayTypeMismatchException 异常。这个异常的发生是因为被包装的 int 不能在 string 数组中存储。

数组调用很明确的不会扩展到数值类型的数组。例如，不存在可以允许把 int[] 看作 object[] 的转换。

## 12.6 数组初始化函数

数组初始化函数可以在域声明中 (§10.4)，在局部变量声明中 (§10.4)，和在数组创建表达式中 (§7.5.10.2) 指定：

```

array-initializer:
    { variable-initializer-listopt }
    { variable-initializer-list , }

variable-initializer-list:
    variable-initializer
    variable-initializer-list , variable-initializer

variable-initializer:
    expression
    array-initializer

```

一个数组初始化函数由括在符号“{”和“}”中，由“,”符号分开的变量初始化函数序列组成。每个变量初始化程序时一个表达式，或者在一个多维数组的情况是一个嵌套数组初始化程序。

使用数组初始化函数的上下文决定了数组要被初始化为的类型。在一个数组创建表达式中，数组类型就在初始化函数前面。在一个域或变量声明中，数组的类型就是域或者变量被声明为的类型。当一个数组初始化函数被用在域或者变量声明中，例如：

```
int[] a = {0, 2, 4, 6, 8};
```

它就是对等效的数组创建表达式的缩写：

```
int[] a = new int[] {0, 2, 4, 6, 8}
```

对于一个单维数组，数组初始化函数必须由表达式序列组成，它与数组的元素类型一致。表达式用上升的顺序对数组元素进行初始化，开始于下标为零的元素。一个数组初始化函数中的表达式数量决定了创建的数组实例的长度。例如，上面的数组初始化函数创建了一个长度为 5 的 int[] 实例，并且用下面的数值对实例进行初始化：

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

对于一个多维数组，数组初始化程序必须有与数组维数一样的嵌套级别。最外层嵌套与最左边的维数对应，而最里层的嵌套与最右边的维数对应。数组的每维的长度由数组初始化函数中相应嵌套级别的元素数量决定。对于每个嵌套的数组初始化函数，元素的数量必须与相同级别的其他数组初始化函数相同。例子：

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

创建了一个二维数组，最左端的维数是 5 而最右端的维数是 2:

```
int[,] b = new int[5, 2];
```

并且用下面的数值对数组实例进行初始化：

```
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;
```

当一个数组创建表达式包括显式的维数长度和一个数组初始化函数时，长度必须是常数表达式而每个嵌套级别的元素数量必须与相应的维的长度匹配。一些例子：

```
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
```

这里，对 *y* 的初始化是错误的，因为维长度表达式不是一个常数，并且 *z* 的初始化函数也是错误的，因为在这个初始化函数中的长度和元素数量不一致。



## 13. 接口

一个接口定义了一个协议。一个实现了一个接口的类或结构必须符合它的协议。一个接口可以从多个基本接口继承，而一个类或结构也可以实现多个接口。

接口可以包含方法、属性、事件和索引。接口自己不为它所定义的成员提供执行程序。接口只是指定必须被实现这个接口的类或接口提供的成员。

### 13.1 接口声明

一个接口声明是一个类型声明 (§9.5) 它声明了新的接口类型。

*interface-declaration:*  
*attributes*<sub>opt</sub> *interface-modifiers*<sub>opt</sub> **interface** *identifier* *interface-base*<sub>opt</sub> *interface-body* ;<sub>opt</sub>

一个接口声明由下面的方式组成：一个可选的属性集合 (§错误！未找到引用源。), 跟着一个可选的接口修饰符集合 (§13.1.1), 跟着关键词 **interface** 和一个命名接口的标识符, 还可以跟着一个可选的接口基本说明 (§13.1.2), 跟着一个接口主体 (§13.1.3), 最后可以选择跟一个分号。

#### 13.1.1 接口修饰符

一个接口声明可以包括一个接口修饰符序列：

*interface-modifiers:*  
*interface-modifier*  
*interface-modifiers* *interface-modifier*  
*interface-modifier:*  
**new**  
**public**  
**protected**  
**internal**  
**private**

对于相同的修饰符在一个接口声明中出现多次是错误的。

**new** 修饰符是在嵌套接口中唯一被允许存在的修饰符。它说明用相同的名称隐藏一个继承的成员，就像在 §10.2.2 中所描述的一样。

**public**, **protected**, **internal** 和 **private** 修饰符控制接口的访问能力。根据发生接口声明的上下文，只有这些修饰符中的一些会被允许 (§错误！未找到引用源。 )。

#### 13.1.2 基本接口

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

*interface-base:*  
 : *interface-type-list*

一个接口的显式基本接口必须至少同接口本身一样可访问 (§错误！未找到引用源。)。例如，在一个公共接口的基本接口中指定一个私有或内部的接口是错误的。

一个接口直接或间接地从它自己继承是错误的。

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

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

IComboBox 的基本接口是 IControl, ITextBox, 和 IListBox。

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

一个实现了接口的类或结构也隐含地实现了所有接口的基本接口。

### 13.1.3 接口主体

一个接口的接口主体定义了接口的成员。

```
interface-body:
    { interface-member-declarationsopt }
```

### 13.2 接口成员

一个接口的成员都是从基本接口继承的成员和被接口自己声明的成员。

```
interface-member-declarations:
    interface-member-declaration
    interface-member-declarations interface-member-declaration

interface-member-declaration:
    interface-method-declaration
    interface-property-declaration
    interface-event-declaration
    interface-indexer-declaration
```

一个接口声明会声明零或多个成员。一个接口的成员必须是方法、属性、事件或索引。一个接口不能包含常数、域、操作符、构造函数、静态构造函数或类型，也不能包括任何类型的静态成员。

所有接口成员隐含的都有公共访问。接口成员声明中包含任何修饰符是错误的。特别是，接口成员不能被用 abstract, public, protected, internal, private, virtual, override, 或 static 修饰符声明。

## 例子

```
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; }
}
```

声明了一个包含每个可能类型成员的接口：一个方法，一个属性，一个事件和一个索引。

一个接口声明创建了一个新的声明域 (§3.1)，而接口成员声明直接被把新成员引入这个声明域的接口声明所包含。下面的规则适用于接口成员声明：

- 方法的名称必须与在同一接口中声明的所有属性和事件的名称不同。另外，方法的签名 (§3.4) 必须与在同一接口中声明的其他方法的签名不同。
- 属性或事件的名称必须与在相同接口中声明所有其他成员的名称不同。
- 一个索引得签名必须与相同接口中声明的所有其他索引的签名不同。

一个接口的继承成员不是这个接口的声明域的一部分。因此，一个接口允许用与继承成员相同的名称或签名声明一个成员。当这个发生时，派生接口成员被称为隐藏了基本接口成员。隐藏一个继承成员不是错误的，但是它会造成编译器给出一个警告。为了禁止这个警告，对于派生接口成员的声明必须包含一个 `new` 修饰符来指示派生的成员要隐藏基本成员。这个题目将在 §3.5.1.2 中讨论。

如果一个 `new` 修饰符被包含在一个没有隐藏继承成员的声明中，就会对这种情况给出一个警告。可以通过去掉 `new` 修饰符来禁止这个警告。

### 13.2.1 接口方法

接口方法使用接口方法声明 (*interface-method-declaration*) 来声明：

```
interface-method-declaration:
    attributesopt newopt return-type identifier ( formal-parameter-listopt ) ;
```

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

### 13.2.2 接口属性

接口属性用接口属性声明 (*interface-property-declarations*) 来声明：

```
interface-property-declaration:
    attributesopt newopt type identifier { interface-accessors }

interface-accessors:
    get ;
    set ;
    get ; set ;
    set ; get ;
```

接口属性声明的属性 ( *attributes* ), 类型 ( *type* ), 和 标识符 ( *identifier* ) 与在一个类中的属性声明有相同的意义 (§10.6)。

接口属性声明的访问符与类属性声明的访问符 (§10.6.5) 相对应, 除了访问符主体通常必须用分号。因此, 无论属性是读写、只读或只写, 访问符都完全确定。

### 13.2.3 接口事件

接口事件用接口事件声明 ( *interface-event-declarations* ) 来声明:

```
interface-event-declaration:
    attributesopt newopt event type identifier ;
```

接口事件声明的属性 ( *attributes* ), 类型 ( *type* ), 和 标识符 ( *identifier* ) 与类的事件声明 (§10.7) 中的那些有相同的意义。

### 13.2.4 接口索引

接口索引使用接口索引声明 ( *interface-indexer-declarations* ) 来声明:

```
interface-indexer-declaration:
    attributesopt newopt type this [ formal-index-parameter-list ] { interface-accessors }
```

接口索引声明中的属性 ( *attributes* ), 类型 ( *type* ), 和形式参数列表 ( *formal-parameter-list* ) 与类的索引声明的那些有相同的意义 (§10.8)。

接口索引声明的访问符与类索引声明的访问符 (§10.6.5) 相对应, 除了访问符主体通常必须用分号。因此, 无论索引是读写、只读或只写, 访问符都完全确定。

### 13.2.5 接口成员访问

可以通过形式为 *I*.*M* 和 *I*[*A*] 的成员访问 (§7.5.4) 和索引访问 (§7.5.6.2) 表达式来访问接口成员, 这里 *I* 是接口类型的实例, *M* 是那个接口类型的一个方法、属性和事件, 而 *A* 是一个索引参数列表。

对于严格的单继承的接口 ( 每个继承链中的接口没有或有一个直接基本接口 ), 成员查找 (§7.3)、方法调用 (§7.5.5.1) 和索引访问 (§7.5.6.2) 规则的影响与类和结构的相同。更新的派生成员用相同的名称和签名隐藏了更旧的派生成员。然而, 对于多继承接口来说, 当两个或多个不相关的接口用相同的名称或签名声明对象的时候, 就会造成不明确。在这节中, 介绍了许多这种情况的例子。在所有的情况下, 隐式的嵌入可以被加在程序代码中来解决这些不确定。

在例子中

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}
```

```

class C
{
    void Test(ICollection x) {
        x.Count(1);           // Error, Count is ambiguous
        x.Count = 1;          // Error, Count is ambiguous
        ((ICollection)x).Count = 1; // Ok, invokes ICollection.Count.set
        ((ICollection)x).Count(1); // Ok, invokes ICollection.Count
    }
}

```

前两个语句造成编译时的错误，因为 Count in ICollection 的成员查找 (§7.3) 造成了不确定。就像例子中所演示的一样，通过把 x 嵌入相应的基本接口类型，来解决不确定问题。这样的嵌入没有运行时的消耗 - 它们只是由在编译时把实例看作老一些地派生类型来实现。

在例子中

```

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);           // Error, both Add methods are applicable
        n.Add(1.0);          // Ok, only IDouble.Add is applicable
        ((IInteger)n).Add(1); // Ok, only IInteger.Add is a candidate
        ((IDouble)n).Add(1);  // Ok, only IDouble.Add is a candidate
    }
}

```

调用 n.Add(1) 是不明确的，因为一个方法调用 (§7.5.5.1) 被声明为相同的类型。但是，调用 n.Add(1.0) 被允许，因为只有 IDouble.Add 是可用的。当加入了显式的嵌入，这里就只有一个候选的方法，这样就不会有不确定存在。

在例子中

```

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
    }
}

```

成员 `IBase.F` 被成员 `ILeft.F` 隐藏。因此，调用 `d.F(1)` 选择了 `ILeft.F`，甚至 `IBase.F` 出现于经过 `IRight` 的访问路径中，而没有被隐藏。

在多继承接口中隐藏的直觉规则如下：如果一个成员在任何访问路径中被隐藏，它就会所有访问路径中被隐藏。因为从 `IDerived` 到 `ILeft` 到 `IBase` 隐藏了 `IBase.F`，所以这个成员在从 `IDerived` 到 `IRight` 到 `IBase` 的访问路径中也被隐藏。

### 13.3 完全有效的接口成员名称

一个接口成员有时被它的完全有效名称 (*fully qualified name*) 所调用。一个接口成员的完全有效名称由下面的形式组成：声明这个变量的接口的名称后面跟着一个点，再跟着这个成员的名称。例如，给出声明

```

interface IControl
{
    void Paint();
}

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

```

`Paint` 的完全有效名称是 `IControl.Paint`，而 `SetText` 的完全有效名称是 `ITextBox.SetText`。

注意一个成员的完全有效名称在声明这个成员的接口中引用这个接口。因此，在上面的例子中，不可能用 `ITextBox.Paint` 引用 `Paint`。

当一个接口是一个名称空间的一部分时，这个接口成员的完全有效名称包括名称空间名称。例如

```

namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}

```

这里，方法 `Clone` 的完全有效名称是 `System.ICloneable.Clone`。

### 13.4 接口实现

接口可以以类和结构实现。为了指出一个类或结构实现一个接口，接口标识符被包含在类或结构的基本类列表中。

```

interface ICloneable
{
    object Clone();
}

```

```

interface IComparable
{
    int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}

```

一个实现了一个接口的类或结构同时也隐含实现了所有接口的基本接口。甚至类或结构没有明显列出在基本类列表中的所有基本接口时，这也是真的。

```

interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}

```

这里，类 `TextBox` 实现了 `IControl` 和 `ITextBox`。

#### 13.4.1 显式接口成员实现程序

出于实现接口的目的，一个类或结构可以声明显式接口成员实现程序（*explicit interface member implementations*）。一个显式接口成员实现程序是一个方法、属性、事件或索引声明，它引用一个完全有效接口成员名称。例如

```

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

```

这里，`ICloneable.Clone` 和 `IComparable.CompareTo` 都是显式接口成员实现程序。

在一个方法调用、属性访问或索引访问中，可以通过完全有效名称访问显式接口成员实现程序。一个显式接口成员实现程序可以只通过接口实例访问，而在那种情况下，引用完全通过成员名称实现。

一个显式接口成员实现程序中包含访问修饰符是错误的，因为它不能包含 `abstract`, `virtual`, `override`, 或 `static` 修饰符。

显式接口成员实现程序跟其他成员相比，有不同的访问能力特性。因为显式接口成员实现程序通过方法调用和属性访问的完全有效名称是不可访问的，所以，它们从感觉上是私有的。但是，由于它们可以通过一个接口实例访问，从感觉上它们又是公共的。

显式接口成员实现程序主要服务于两个目的：

- 因为显式接口成员实现程序通过类或结构实例是不可访问的，所以允许接口实现从类或结构的公共接口中执行。当一个类或结构实现一个对类或结构的成员用户不感兴趣的内部接口时，这就很有用了。
- 显式接口成员实现程序允许用相同的签名消除接口成员的歧义。没有显式接口成员实现程序，一个类或结构包含有签名和返回类型都相同的接口成员的不同实现是不可能的，而一个类或结构包含有相同的签名和不同的返回类型的所有接口成员的实现是可能的。

为了使显式接口成员实现程序有效，类或成员必须在它的基本类列表中命名一个接口，这个列表中包含一个成员，它的全部有效名称、类型和参数类型与显式接口成员实现程序的那些要完全一致。因此，在下面的类中

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

`IComparable.CompareTo` 的声明是无效的，因为 `IComparable` 没有被列在 `Shape` 的基本类列表中，并且不是 `ICloneable` 的一个基本接口。与此类似，在声明中

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

`Ellipse` 中的声明 `ICloneable.Clone` 是错误的，因为 `ICloneable` 没有在 `Ellipse` 的基本类列表中明显列出。

一个接口成员的完全有效名称必须在声明这个成员的接口中引用这个接口。因此，在这个声明中

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

`Paint` 的显式接口声明实现必须被写为 `IControl.Paint`。



### 13.4.2 接口映射

一个类或结构必须提供在类或结构的基本类列表中列出的接口中所有成员的实现。在一个类或结构中实现对接口成员的定位的过程被称为接口映射。

一个类或结构 C 的接口映射为位于 C 的基本类列表中指定的每个接口的每个成员确定实现程序。一个特殊接口成员 I.M 的实现通过对每个类或类型 S 进行检查来确定，这个检查从 C 开始，并且对 C 的每个继承的基本类重复进行，直到发现匹配，这里 I 是一个接口，而在 I 中声明了成员 M：

- 如果 S 中包含一个与 I 和 M 匹配的显式接口成员实现程序的声明，那么这个成员就是 I.M 的实现过程。
- 另外，如果 S 中包含一个与 M 匹配的非静态公共成员声明，那么这个成员就是 I.M 的实现过程。

如果执行过程不能为在 C 的基本类列表中指定的所有接口的所有成员定位，就会发生错误。注意一个接口的成员包括那些从基本接口继承的成员。

出于接口映射的目的，在下面的情况中一个类成员 A 与一个接口成员 B 相匹配：

- A 和 B 都是方法，而且 A 和 B 的地名称、类型和形式参数列表都是一样的。
- A 和 B 都是属性，A 和 B 的类型和名称是相同的，并且 A 有和 B 一样的访问符（如果它不是一个显式的接口成员执行过程，A 就被允许有附加的访问符）。
- A 和 B 都是事件，并且 A 和 B 的名称和类型是相同的。
- A 和 B 都是索引，A 和 B 的类型和形式参数列表是相同的，并且 A 有和 B 一样的访问符（如果它不是一个显式的接口成员执行过程，A 就被允许有附加的访问符）。

接口映射算法中值得注意的地方是：

- 当已经确定了可以实现接口成员的类或结构成员时，显式接口成员实现程序比同一个类和结构中的其他成员有更高的优先级。
- 私有的、保护的合静态成员不参与接口映射。

在例子中

```
interface ICloneable
{
    object Clone();
}

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

C 的成员 ICloneable.Clone 变成 ICloneable 中的 Clone 的实现程序，因为显式接口成员实现程序比其他成员有更高的优先级。

如果类或结构实现了两个和更多包含有相同的名称、类型和参数类型的成员的接口，那么把那些接口成员中的每个都映射到单独的类或结构成员是可以的。例如

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

```

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

```

这里，IControl 和 IForm 的 Paint 方法都映射到 Page 中的 Paint 方法上。当然这两个方法有分立的显式接口成员实现程序也是可以的。

如果一个类或结构实现了一个包含隐藏成员的接口，那么其中一些成员就需要通过显式接口成员实现程序来实现。例如

```

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

```

这个接口的一个实现程序至少需要一个显式接口成员实现程序，并且要使用下面的一种形式

```

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() {...}
}

```

当一个类实现多个有相同基本接口的接口，那里就只能有一个基本接口的实现程序。在例子中

```

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) {...}
}

```

对于在基本类列表中命名的 `IControl`，可以有分立的实现程序，`IControl` 被 `ITextBox` 继承，并且 `IControl` 也被 `IListBox` 继承。实际上，不要求这些接口都相同。反之，`ITextBox` 和 `IListBox` 的实现程序共享相同的 `IControl` 的实现程序，而 `ComboBox` 被认为实现了三个接口 `IControl`, `ITextBox`, 和 `IListBox`。

基本类的成员参与了接口映射。在例子中

```

interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

class Class2: Class1, Interface1
{
    new public void G() {}
}

```

`Class1` 中的方法 `F` 用于 `Class2` 的 `Interface1` 的实现成员中。

### 13.4.3 接口实现程序继承

一个类继承了所有被它的基本类提供的接口实现程序。

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

```

interface IControl
{
    void Paint();
}

class Control: IControl
{
    public void Paint() {...}
}

class TextBox: Control
{
    new public void Paint() {...}
}

```

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

```

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

```

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

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

就会看到下面的结果

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

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

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

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

#### 13.4.4 接口重新实现程序

一个继承了一个接口实现程序的类，可以通过在基本类列表中包含一个接口来重新实现这个接口。

一个接口的重新实现程序与一个接口的初始化实现程序有相同的接口映射规则。因此，继承的接口映射不会对为接口的重新实现程序建立的接口映射有任何影响。例如在声明中

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

```

class Control: IControl
{
    void IControl.Paint() {...}
}
class MyControl: Control, IControl
{
    public void Paint() {}
}

```

Control 把 IControl.Paint 映射到 Control.IControl.Paint 中的事实不会影响在 MyControl 中的重新实现程序，它把 IControl.Paint 映射到 MyControl.Paint 中。

继承的公共成员声明和继承的显式接口成员声明参与为重新实现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() {}
}

```

这里，Derived 中的重新实现程序 IMethods 把接口方法映射到 Derived.F, Base.IMethods.G, Derived.IMethods.H, 和 Base.I 中。

当一个类实现了一个接口，它隐含地也实现了所有接口的基本接口。另外，一个接口的重新实现程序也隐含的是所有接口的基本接口的重新实现程序。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() {...}
}

```

这里 `IDerived` 的重新实现程序也重新实现了 `IBase`, 并把 `IBase.F` 映射到 `D.F`。

### 13.4.5 抽象类和接口

与非抽象类相同，一个抽象类必须为类的基本类列表中列出的接口的所有成员提供实现程序。但是，一个抽象类被允许把接口方法映射到抽象方法中。例如

```
interface IMethods
{
    void F();
    void G();
}

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

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

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

```
interface IMethods
{
    void F();
    void G();
}

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

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

# 14. 枚举

一个枚举类型 ( *enum type* ) 是一个有命名的常数的独特类型。枚举声明可以在类声明可以发生的地方出现。

例子

```
using System;
enum Color
{
    Red,
    Green,
    Blue
}
```

声明了一个名为 Color 的枚举类型，并且有成员 Red, Green, 和 Blue。

## 14.1 枚举声明

一个枚举声明声明一个新的枚举类型。一个枚举声明开始于一个关键词 `enum`，还包括一个名称定义，访问能力，基本类型和 `enum` 的成员。

```
enum-declaration:
    attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-modifiers:
    enum-modifier
    enum-modifiers enum-modifier

enum-modifier:
    new
    public
    protected
    internal
    private

enum-base:
    : integral-type

enum-body:
    { enum-member-declarationsopt }
    { enum-member-declarations , }
```

每个枚举类型都有一个相应的整数类型，称为枚举类型的基本类型 ( *underlying type* )。这个基本类型可以表示任何在枚举中定义的计数器值。一个枚举声明可以显式地声明 `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` 或 `ulong` 的一个基本类型。注意那个 `char` 不能被用在一个基本类型中。一个没有显式声明一个基本类型的枚举声明基本类型为 `int`。

例子

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

声明了一个基本类型为 long 的枚举。就像例子中一样，一个开发者可以选择使用基本类型 long，来使得可以使用 long 取值范围中的数值而不是 int 取值范围中的数值，或者防止在将来发生这种情况。

## 14.2 枚举成员

枚举类型声明的主体定义了零或多个枚举成员，它们是枚举类型的被命名的常数。没有哪两个枚举成员可以有相同的名称。一个枚举声明可以不包含方法、属性、事件、操作符或类型的声明。

```
enum-member-declarations:
    enum-member-declaration
    enum-member-declarations , enum-member-declaration

enum-member-declaration:
    attributesopt identifier
    attributesopt identifier = constant-expression
```

每个枚举成员都有相应的常数数值。数据的类型是所在枚举地基本类型。每个枚举成员的常数数值必须在枚举的基本类型的范围之内。例子

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

是错误的，因为常数数值 -1, -2, 和 -3 不在基本整数类型 uint 的范围之内。

多个枚举成员可以共享同一个相关的数值。例子

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```

介绍了一个有两个枚举成员 Blue 和 Max 的成员，它们有相同的相关数值。

一个枚举成员的相关数值既使用隐式赋值，也使用显式赋值。如果枚举成员的声明有常数表达式（*constant-expression*）初始化函数，那个被隐式转换为枚举的基本类型的常数表达式的数值就是枚举成员的相关数值。如果枚举成员的声明没有初始化函数，它的相关数值就被隐含地设置，像下面一样：

- 如果枚举成员是在枚举类型中声明的第一个枚举成员，它的相关数值为零。
- 另外，枚举成员的相关数值通过把先前的枚举成员的相关数值加一获得。这个增加的数值必须在可以被基本类型代表的数值的范围之内。

例子

```
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";
        }
    }
}

```

打印出枚举成员名称和它们相应的数值。输出为：

```

Red = 0
Blue = 10
Green = 11

```

出于下面的原因：

- 枚举成员 Red 自动被赋值为零（由于它没有初始化函数并且是第一个枚举成员）；
- 枚举成员 Blue 被显式地赋值为 10；
- 而枚举成员自动地被赋予一个比前面的程序中的成员大的数值。

一个枚举成员的相关数值不管在直接或间接上，都不能使用它自己相关枚举成员的数值。不同于这个循环限制，枚举成员初始化函数可以自由地指向其他枚举成员初始化函数，而不管他们的文字位置。在一个枚举成员初始化函数中，其他枚举成员的数值通过被当作有它们的基本类型的类型，因此在引用其他枚举成员时不需要嵌入。

例子

```

enum Circular
{
    A = B
    B
}

```

是无效的，因为 A 和 B 的声明是循环的。A 从显式上依赖 B，而 B 在隐式上依赖 A。

枚举成员用一种与类中的域一样的方法名称空间和设定范围。枚举成员的范围是所包含枚举类型的主体。在那个范围中，枚举成员可以用它们自己的简单名称引用。对于所有其他代码，枚举成员的名称必

须与它的枚举类型的合在一起。枚举成员没有任何声明的访问能力 - 如果它包含的枚举类型是可访问的，这个枚举成员就是可访问的。

### 14.3 枚举数值和操作

每个枚举类型定义了一个独立的类型；一个显式的枚举转换(**\$错误！未找到引用源。**)需要用于在枚举类型和整数类型，或在两个枚举类型间进行转换。枚举类型可以有的地数值的集合不被它的枚举成员限制。特别是，一个枚举类型的基本类型的任何数值可以被嵌入到枚举类型中，并且是那个枚举类型的一个分立的有效数值。

枚举成员有它们所在的枚举类型的类型（除非在另一个枚举成员初始化函数中：看§14.2）。在枚举类型 E 中声明的枚举成员的数值得相关数值 v 是 (E)v。

下面的操作符可以北用在枚举类型的数据上：==, !=, <, >, <=, >= (**\$错误！未找到引用源。**), + (**\$错误！未找到引用源。**), - (**\$错误！未找到引用源。**), ^, &, | (**\$错误！未找到引用源。**), ~ (**\$错误！未找到引用源。**), ++, -- (§7.5.9, **\$错误！未找到引用源。**), sizeof (§7.5.12)。

每个枚举类型自动从类 System.Enum 派生。因此，这个类的继承的方法和属性可以被用在一个枚举类型的数值上。

# 15. 代表

代表使得这样的方案变为可能：其他语言 - C++, Pascal, Modula 等可以用功能指针来定位。与 C++ 的功能指针不同，代表完全是面向对象的；与 C++ 指向成员功能不同，代表把一个对象实例和方法都进行封装。

一个代表声明定义了一个从类 `System.Delegate` 延伸的类。一个代表实例封装一个方法，可调用实体。对于实例方法，一个可调用实体由一个实例和一个实例中的方法组成。对于静态方法，一个可调用实体完全只是由一个方法组成。如果你有一个代表实例和一个适当的参数集合，你就可以用参数来调用这个代表。

代表的一个有趣而又有用的特性是它不知道或不关心它引用的对象的类。只要方法的签名与代表的签名一致，任何对象都可以作。这使得代表适合作“匿名”调用”。

## 15.1 代表声明

一个代表声明 ( *delegate-declaration* ) 是一个类型声明 ( *type-declaration* ) (§9.5)，它声明一个新的代表类型。

```

delegate-declaration:
    attributesopt delegate-modifiersopt delegate result-type identifier ( formal-parameter-listopt ) ;

delegate-modifiers:
    delegate-modifier
    delegate-modifiers delegate-modifier

delegate-modifier:
    new
    public
    protected
    internal
    private
  
```

在代表声明中相同的修饰符出现多次是错误的。

`new` 修饰符只允许在被其他类型声明代表中出现。它指定那个代表用相同的名称隐藏一个继承的成员，就像在 §10.2.2 中描述的一样。

`public`, `protected`, `internal`, 和 `private` 修饰符控制代表类型的访问能力。根据代表声明发生的上下文，这些修饰符中的某些将不被允许 (§**错误！未找到引用源。**)。

形式参数列表指定了代表的签名，而结果类型 ( *result-type* ) 指定了代表的返回类型。代表的签名和返回类型必须与代表类型所包装的方法的签名和返回类型匹配。C# 中的代表类型是名称等价的，并不是结构上等价。两个有相同签名和返回类型的不同的代表类型被认为是不同的代表类型。

一个代表类型是一个从 `System.Delegate` 派生的类类型。代表类型隐含为密封的：从代表类型派生任何类型是不行的。从 `System.Delegate` 派生一个非代表类型的类也不被允许。注意，`System.Delegate` 自己不是一个代表类型，它是一个类类型，而所有代表类型都从它派生。

C# 为代表实例化和调用提供了特别的语法。除了实例化，可以用于类或类的实例的任何操作也都可以被用于代表类或实例。特别是，通过使用成员访问语法可以访问 `System.Delegate` 类型的成员。

### 15.1.1 可合并的代表类型

代表类型被分为两类：可合并的和不可合并得。一个可合并的代表类型必须满足下面的条件：

- 代表类型声明的返回类型必须是 `void`。
- 代表类型的参数都不能作为输出参数 (§10.5.1.3) 来声明。

如果试图合并 (§**错误！未找到引用源。**) 两个不可合并代表类型的实例，就会发生一个运行时的异常，除非一个或另一个实例为 `null`。

### 15.2 代表实例化

虽然代表的行为在大多数情况下与其他的类相似，C# 还是为实例化一个代表实例提供了特殊的语法。一个代表创建表达式 (*delegate-creation-expression*) (§7.5.10.3) 用来创建一个代表的新实例。最新创建的代表实例就会指向下面之一：

- 在实例创建表达式 (*delegate-creation-expression*) 中引用的方法，或是
- 目标对象（不能为 `null`）和在实例创建表达式 (*delegate-creation-expression*) 中引用的实例方法，或
- 另一个代表

一旦被实例化，代表实例通常指向相同的目标对象和方法。

### 15.3 多点传送代表

代表可以使用加操作符 (§**错误！未找到引用源。**) 来合并，并且一个代表可以通过使用减操作符来从另一个代表中去掉。一个通过合并两个或多个（非空）代表实例创建的代表实例被称为多点传送 (multicast) 代表实例。对于任何代表实例，代表实例的调用列表 (*invocation list*) 作为非多点传送的规则列表来定义，而当代表实例被调用时，它也要被调用。更多如下：

- 对于非多点传送代表实例，调用列表有代表实例自己组成。
- 对于通过把两个代表合并创建的多点传送代表实例，调用列表是通过使用创建多点传送代表的加法操作的两个操作数的调用列表合并得到的。

### 15.4 代表调用

C# 为调用一个代表提供特殊的语法。当一个非多嵌入代表被调用时，对代表用相同参数指向的方法进行调用，并且返回指向这个方法返回的相同数值。有关代表调用的详细信息参考 §7.5.5.2。如果在一个代表的调用过程中出现了一个异常，并且这个异常没有在所调用的方法中被抓住，那么对异常俘获语句的搜索会扩展到的调用这个方法中，就好像那个方法已经直接被代表指向的方法调用了一样。

多嵌入代表的调用通过按顺序调用在调用列表中的每个代表来进行。每个调用都会传送相同的参数集合。如果代表包含引用参数 (§10.5.1.2)，那么每个方法调用将引用相同的变量；通过某个调用列表中的方法对那个变量进行的改动会被调用列表中的后面的调用“看到”。

如果在多嵌入代表的调用过程中发生了一个异常，而这个异常没有被调用的方法俘获，那么对异常俘获语句的搜索会扩展到的调用这个方法中，并且调用列表中任何后面的方法都不会被调用。

# 16. 异常

C# 中的表达式为处理系统级别和应用程序级别的错误条件，提供了一个构造的、统一的合类型安全的方法。C# 中的异常机制与 C++ 中的相当相似，但是有一些重要的不同之处：

- 在 C# 中，所有的异常必须用一个从 `System.Exception` 派生的类类型的实例表示。在 C++ 中，任何类型的任何数值都可以用于表示异常。
- 在 C# 中，一个最后的块 (§ 错误！未找到引用源。) 可以被用来编写在正常指向和异常条件下都可以执行的中止代码。如果没有重复代码，那样的代码在 C++ 中很难写。
- 在 C# 中，如溢出、被零除和空解除参照等系统级的异常会有被很好定义的异常类，并且应用程序级别错误条件等同。

## 16.1 异常的产生

异常可以用两种不同的方法抛出。

- 一个 `throw` 语句 (§ 错误！未找到引用源。) 会直接无条件地抛出一个异常。控制不会接触到紧跟在 `throw` 后面的语句。
- 当操作符不能正常结束时，在某种特殊情况下，在 C# 语句和表达式执行的过程中产生的某种异常的条件产生一个异常。例如，一个整数除法操作 (§ 错误！未找到引用源。) 在分母为零时抛出一个 `System.DivideByZeroException` 异常。§ 16.4 中给出了一个可以在这种方式下发生的各种异常的列表。

## 16.2 System.Exception 类

类 `System.Exception` 是所有异常的基本类型。这个类有一些值得注意的属性，这些属性所有异常都有：

- `Message` 是一个只读属性，它是一个只读属性，可以包含一个可以被读的人的异常原因的描述。
- `InnerException` 是一个只读属性，它包含这个异常的“内部异常”。如果它不是 `null`，就指出当前的异常是作为对另外一个异常的回答而被抛出。产生当前异常的异常可以在 `InnerException` 属性中得到。

这些属性的数值可以在 `System.Exception` 构造函数中指定。

## 16.3 异常怎样被处理

异常被 `try` 语句 (§ 错误！未找到引用源。) 处理。

当发生一个异常，系统就会查找离可以处理这个异常的语句最近的 `catch` 语句，并且有异常的运行时类型决定。首先，当前的方法是查找文字上很近，并且与 `try` 语句相关的 `catch` 语句会被按顺序考虑。如果这样做失败了，就会查找调用 `try` 语句的方法和当前的方法中文字接近的地方，这个 `try` 语句围绕着对当前方法调用的点。通过对与异常被抛出的运行时类型的相同的类或基类的命名来进行查找，这个查找一直进行直到找到可以处理当前的异常的 `catch` 语句为止。一个没有对一个异常类命名的 `catch` 语句可以处理任何异常。

一旦找到匹配的 catch 语句，系统就被控制权交给 catch 语句的第一个语句。在这个 catch 语句开始执行前，系统首先执行与 try 语句相关，而且比俘获异常的语句嵌套更深的任何 finally 语句。

如果没有找到匹配的 catch 语句，下面两件事情中的一件就会发生：

- 如果对匹配的 catch 语句的查找找到了一个静态构造函数 (§10.12) 或静态域初始化函数，那么就会在引发对静态构造函数调用的地方抛出一个 `System.TypeInitializationException` 异常。`TypeInitializationException` 的内部异常包含最初抛出的异常。
- 如果对匹配的 catch 语句的查找找到了一个最初开始这个线程或程序的代码，那么这个线程或程序的执行就被中止。

## 16.4 通用异常类

下面的异常被某些 C# 操作抛出。

<code>System.OutOfMemoryException</code>	当试图通过 <code>new</code> 来分配内存而失败时抛出。
<code>System.StackOverflowException</code>	当执行栈被太多未完成的方法调用耗尽时抛出；典型情况是指非常深和很大的递归。
<code>System.NullReferenceException</code>	当 <code>null</code> 引用在造成引用的对象被需要的情况下使用时抛出。
<code>System.TypeInitializationException</code>	当一个静态构造函数抛出一个异常，并且没有任何 catch 语句来俘获它的时候抛出。
<code>System.InvalidCastException</code>	当一个从基本类型或接口到一个派生类型的转换在运行时失败时抛出。
<code>System.ArrayTypeMismatchException</code>	当因为存储元素的实例类型与数组的实际类型不匹配而造成象一个数组存储失败时抛出。
<code>System.IndexOutOfRangeException</code>	当试图通过一个比零小或者超出数组边界的标签来索引一个数组时抛出。
<code>System.MulticastNotSupportedException</code>	当试图合并两个非空代表失败时抛出；因为代表类型没有 <code>void</code> 返回类型。
<code>System.ArithmeticException</code>	一个异常的基类，它在算术操作时发生，如 <code>DivideByZeroException</code> 和 <code>OverflowException</code> 。
<code>System.DivideByZeroException</code>	当试图用整数类型数据除以零时抛出。
<code>System.OverflowException</code>	当 <code>checked</code> 中的一个算术操作溢出时抛出。

# 17. 属性

C# 语言的大多数都使得程序员者可以指定关于在程序中定义的实体的公开的消息。例如，一个类中的一个方法的访问性，可以通过用方法修饰符 `public`, `protected`, `internal` 和 `private` 对它进行修饰来指定。

C# 使得程序员可以创造声明信息的新的种类，来为各种程序实体指定声明信息，并且在运行时环境中找回属性信息。例如，一个框架也许定义了一个 `HelpAttribute` 属性，它可以被放在例如类和方法的程序元素中来提供从程序元素到它们的文档的映射。

声明信息的新种类通过属性类 (§17.1) 的声明来定义，它可能有位置的和名称的参数 (§17.1.2)。声明信息使用属性 (§17.2) 来指定 C# 程序，并且可以在运行时作为属性实例来检索 (§17.3)。

## 17.1 属性类

一个属性类的声明定义了一种可以放在声明中的新的属性。一个从抽象类 `System.Attribute` 派生的类，不管是直接派生还是间接派生都是属性类。

一个属性类的声明要受下面的附加约束的影响：

- 一个非抽象属性类必须有公共的可访问性。
- 所有其中的非抽象属性类为嵌套的类型必须有公共的可访问性。
- 一个非抽象属性必须至少有一个公共构造函数。
  - 一个属性类的每个公共构造函数的每个形式参数类型必须是属性参数类型 (§17.1.3)。

根据惯例，属性类被用一个后缀 `Attribute` 修饰。一个属性的使用可以包括也可以忽略这个后缀。

### 17.1.1 AttributeUsage 属性

`AttributeUsage` 属性被用来描述一个属性类是否可以被使用。

`AttributeUsage` 属性有一个位置参数用来使一个属性类可以指定它可以被应用的声明的类型。例子

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: System.Attribute
{ }
```

定义了一个名为 `SimpleAttribute` 的属性类，它可以被放在类声明 (*class-declarations*) 和接口声明 (*interface-declarations*) 中。例子

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

介绍了很多 `Simple` 属性的使用。这个属性用一个名为 `SimpleAttribute` 的类定义，但是对这个属性的使用可以忽略后缀 `Attribute`，因此把名称简化为 `Simple`。上面的例子从语义上同下面的例子相同

```
[SimpleAttribute] class Class1 {...}
[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` 属性有一个 `AllowMultiple` 命名的参数，它指出是否这个指示的属性可以为一个所给的实体被指定多次。一个可以在一个实体中被多次指定的属性被称为多次使用属性类 (*multi-use*)

*attribute class* )。一个在一个实体中至少可以被指定一次的属性被称为单次使用属性类 (*single-use attribute class*)。

例子

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: System.Attribute {
    public AuthorAttribute(string value);
    public string Value { get {...} }
}
```

定义了一个多次使用属性类，名为 AuthorAttribute。例子

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1 {...}
```

介绍了一个使用了两个 Author 属性的类声明。

### 17.1.2 位置的和名称的参数

属性类可以有位置的参数 (*positional parameters*) 和名称的参数 (*named parameters*)。一个属性类的每个公共构造函数为属性类定义了一个有效的位置参数序列。一个属性类的每个非静态公共读写域和属性为属性类定义了一个名称的参数。

例子

```
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: System.Attribute
{
    public HelpAttribute(string url) { // url is a positional parameter
        ...
    }
    public string Topic { // Topic is a named parameter
        get {...}
        set {...}
    }
    public string Url { get {...} }
}
```

定义了名为 HelpAttribute 的一个属性类，它有一个位置参数 (string url) 和一个名称的参数 (string Topic)。只读的属性 Url 并不定义一个名称的参数，它是非静态和公共的，但是由于它是只读的，所以它并没有定义一个名称的参数。

例子

```
[HelpAttribute("http://www.mycompany.com/.../Class1.htm")]
class Class1 {
}
[HelpAttribute("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2 {
}
```

介绍了多个属性的使用。

### 17.1.3 属性参数类型

一个属性类的位置的和名称的参数的类型被限制于属性参数类型 (*attribute parameter types*)。如果一个类型是下面的一种，就是一个属性类型：



- 下面类型中的一个: bool, byte, char, double, float, int, long, short, string.
- 类型 object.
- 类型 System.Type.
- 一个有公共可访问性的枚举类型并且其中的类型是嵌套的（如果有），而且也有公共可访问性的。

一个定义了一个不是属性参数类型的位置或名称的参数的属性类是错误的。例子

```
public class InvalidAttribute: System.Attribute
{
    public InvalidAttribute(Class1 c) {...}          // error
}
public class Class1 {
    ...
}
```

是错误的，因为它定义了一个有类型 Class1 的位置参数的属性类，而类型 Class1 不是属性参数类型。

## 17.2 规范

一个属性是一个附加声明信息的片断，它是为一个声明指定的。属性可以为类型声明（*type-declarations*）、类成员声明（*class-member-declarations*）、枚举成员声明（*enum-member-declarations*）、属性访问符声明（*property-accessor-declarations*）和形式参数（*formal-parameter*）进行声明。

属性在属性片段指定，每个属性片段被包围在方括号中，有多个属性在逗号分开的列表中指定。属性被指定的顺序和它们在片段中的安排方式不重要。属性指定的 [A][B], [B][A], [A, B], 和 [B, A] 都是相同的。

```
attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections attribute-section

attribute-section:
    [ attribute-list ]
    [ attribute-list , ]

attribute-list:
    attribute
    attribute-list , attribute

attribute:
    attribute-name attribute-argumentsopt

attribute-name:
    reserved-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-list , positional-argument

positional-argument:
    attribute-argument-expression

named-argument-list:
    named-argument
    named-argument-list , named-argument

named-argument:
    identifier = attribute-argument-expression

attribute-argument-expression:
    expression

```

**问题**

我们需要更新语法来把`[assembly: attributes]` 和`[module: attributes]` 两种形式合并到一起。

一个属性由属性名称 (*attribute-name*) 和位置的和名称的参数的可选列表组成。位置的参数 (如果有) 领先于名称的参数。一个位置的参数由属性参数表达式 (*attribute-argument-expression*) 构成；一个名称参数由名称, 跟着一个等号, 跟着一个属性参数表达式 (*attribute-argument-expression*) 构成。

属性名称 (*attribute-name*) 指定了一个保留的属性和一个属性类。如果属性名称 (*attribute-name*) 的形式是类型名称 (*type-name*) , 那么它的名称必须指向一个属性类。否则, 会产生一个编译时错误。例子

```

class Class1 {}
[Class1] class Class2 {}    // Error

```

是错误的, 因为它试图把 Class1 当作属性类使用, 但是它不是属性类。

一般而言, 属性类用后缀 Attribute 命名。一个形式为类型名称 (*type-name*) 的属性名称可以包含或忽略这个后缀。在属性名称和属性类的名称之间的准确匹配是首选的。例子

```

[AttributeUsage(AttributeTargets.All)]
public class X: System.Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: System.Attribute
{}

[X]           // refers to X
class Class1 {}

[XAttribute]  // refers to XAttribute
class Class2 {}

```

介绍了两个属性类, 名为 X 和 XAttribute。属性 [X] 指向名为 X 的类, 而属性 [XAttribute] 指向名为 [XAttribute] 的属性类。如果去掉对类 X 的声明, 那么所有的属性都指向名为 XAttribute 的属性类:

```

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: System.Attribute
{}

[X]           // refers to XAttribute
class Class1 {}

```

```
[XAttribute] // refers to XAttribute
class Class2 {}
```

在同一实体中多次使用单次使用属性类是错误的。例子

```
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: System.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 {}
```

是错误的，因为它试图在 Class1 的声明中多次使用 HelpString，而 HelpString 是一个单次属性类。

如果所有下面的语句都为真，表达式 E 就是一个属性参数表达式：

- E 的类型是一个属性参数类型 (§17.1.3)。
- 在编译时，E 的数值可以被分解为下面的一个：
  - 一个常数数据。
  - 一个 System.Type 对象。
  - 一个属性参数表达式的一维数组。

### 17.3 属性实例

一个属性实例是一个在运行时代表一个属性的实例。一个属性用一个属性、位置参数和名称参数定义。一个属性实例是一个属性类的实例，它用位置和名称参数来初始化。

属性实例的搜索涉及到所有编译时和运行时过程，就像在下面的章节中所描述的一样。

#### 17.3.1 一个属性的编译

一个有属性类 T，位置参数列表 P 和名称参数列表 N 的属性的编译过程包括下面几步：

- 对形式为 new T(P) 的对象创建表达式进行编译，是按照编译时过程步骤进行。这些步骤或是会产生一个编译时错误，或是确定一个可以在运行时调用的 T 的构造函数。把这个构造函数称为 C。
- 如果上面步骤中决定的构造函数没有公共的可访问性，就会发生一个编译时错误。
- 对每个 N 中的名称参数 Arg:
  - 把 Name 作为名称参数 Arg 的标识符。
  - Name 必须标识一个 T 中的非静态读写公共域或属性。如果 T 没有那样的域或属性，就会发生一个编译时错误。
- 为属性实例的运行时实例化保持下面的信息：属性类 T，T 中的构造函数 C，位置参数列表 P 和名称参数列表 N。

### 17.3.2 一个属性实例的运行时检索

对属性的编译产生了一个属性类  $T$ ， $T$  的构造函数  $C$ ，位置参数列表  $P$  和名称参数列表  $N$ 。给出这些信息，一个属性实例就可以在运行时按照下面的步骤进行检索：

- 为执行一个形式为  $T(P)$  的对象创建表达式，跟随运行时过程步骤，在编译时确定构造函数  $C$  的使用。这些步骤或导致一个异常，或产生一个实例  $T$ ，把这个实例称作  $O$ 。
- 对于每个  $N$  中的名称参数  $Arg$  in  $N$ ，以下面的顺序：
  - 让  $Name$  作为名称参数  $Arg$  的标识符。如果  $Name$  没有在  $O$  中指定一个非静态公共读写域或属性，那么会抛出一个异常。
  - 让  $Value$  作为对  $Arg$  属性参数表达式求值得结果。
  - 如果  $Name$  确定了一个  $O$  中的域，那么把这个域设置为数据  $Value$ 。
  - 否则， $Name$  确定一个  $O$  中的属性。把这个属性设置为数据  $Value$ 。
  - 结果是  $O$ ，一个属性类  $T$  的实例，它被初始化为有位置参数列表  $P$  和名称参数列表  $N$ 。

## 17.4 保留的属性

属性的一小部分在某些方面影响语言。这些属性包括：

- `System.AttributeUsageAttribute`，它被用来描述一个可以使用属性类的方法。
- `System.ConditionalAttribute`，它被用来定义条件方法。
- `System.ObsoleteAttribute`，它被用来把一个成员标注为废弃的。

### 17.4.1 AttributeUsage 属性

`AttributeUsage` 属性被用来描述一种属性类可以被使用的方式。

一个用 `AttributeUsage` 属性声明的类必须从 `System.Attribute` 派生，或者直接或者间接。否则，会产生一个编译时错误。

```
[AttributeUsage(AttributeTargets.Class)]
public class AttributeUsageAttribute: System.Attribute
{
    public AttributeUsageAttribute(AttributeTargets validOn) {...}
    public AttributeUsageAttribute(AttributeTargets validOn,
                                   bool allowMultiple,
                                   bool inherited) {...}

    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,

    All = Assembly | Module | Class | Struct | Enum | Constructor |
          Method | Property | Field | Event | Interface | Parameter |
          Delegate,

    ClassMembers = Class | Struct | Enum | Constructor | Method |
          Property | Field | Event | Delegate | Interface,
}

```

#### 17.4.2 条件属性

条件属性使得条件方法的定义变为可能。条件属性在一个预处理标识符的形式中指定一个条件。对于一个条件方法的调用或者被包括或者被忽略，要根据在调用的地方这个符号是否被定义来决定。如果符号被定义了，那么方法调用被包括，如果符号没有定义，那么调用被忽略。

```

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class ConditionalAttribute: System.Attribute
{
    public ConditionalAttribute(string conditionalSymbol) {...}
    public string ConditionalSymbol { get {...} }
}

```

一个条件方法要受下面的约束：

- 条件方法必须是一个类声明中的方法。如果条件属性在接口方法中被指定，那么就会发生一个编译时错误。
- 条件方法必须返回一个 void 类型。
- 条件方法不能用 override 修饰符来标注。一个条件方法可以用 virtual 修饰符来标注。覆盖这样的一个是隐含地有条件的，并且不能用条件属性来显式地标注。
- 条件方法不能是一个接口方法的实现程序，否则，会产生编译时错误。

而且，如果条件方法被用在代表创建表达式中，也会产生一个编译时错误。例子

```

#define DEBUG
class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

```

```

class Class2
{
    public static void Test() {
        Class1.M();
    }
}

```

把 `Class1.M` 声明为一个条件方法。Class2 的 `Test` 方法调用这个方法。由于预定义的符号 `DEBUG` 已经被定义，因此如果 `Class2.Test` 被调用，它就会调用 `M`。如果符号 `DEBUG` 没有被定义，那么 `Class2.Test` 将不会调用 `Class1.M`。

注意，下面这点很重要，包含或去掉一个对条件方法的调用，是被在调用的地方的预定义标识符所控制的。在例子中

```

// Begin class1.cs
class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
// End class1.cs

// Begin class2.cs
#define DEBUG
class Class2
{
    public static void G {
        Class1.F(); // F is called
    }
}
// End class2.cs

// Begin class3.cs
#undef DEBUG
class Class3
{
    public static void H {
        Class1.F(); // F is not called
    }
}
// End class3.cs

```

类 `Class2` 和 `Class3` 每个都包含对条件方法 `Class1.F` 的调用，这个条件是基于 `DEBUG` 是存在的还是不存在地。由于这个符号在 `Class2` 的上下文中被定义了，但是没有在 `Class3` 中定义，因此对 `Class2` 中的 `F` 的调用被实际进行，而对 `Class3` 中的 `F` 的调用被忽略了。

在一个继承链中使用条件方法可能会很混乱。通过 `base` 对条件方法的调用，形式为 `base.M`，要受通常条件方法调用规则的限制。在例子中

```
// Begin class1.cs
class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
// End class1.cs

// Begin class2.cs
class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M(); // base.M is not called!
    }
}
// End class2.cs

// Begin class3.cs
#define DEBUG
class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M(); // M is called
    }
}
// End class3.cs
```

Class2 包含一个在它的基类中定义的对 M 的调用。因为基本方法是根据符号 DEBUG 的存在来确定，而它没有被定义，所以这个调用就被忽略了。因此，写到控制台的方法只是"Class2.M executed"。聪明地使用 *pp-declarations* 就可以解决这样的问题。

### 17.4.3 废弃的属性

废弃的属性被用于对不再使用的程序元素进行标记。

```
[AttributeUsage(AttributeTargets.All)]
public class ObsoleteAttribute: System.Attribute
{
    public ObsoleteAttribute(string message) {...}
    public string Message { get {...} }
    public bool IsError { get {...} set {...} }
}
```





# 18. 危险代码

## 问题

我们需要编写这章，它将包含指针的使用，包含“危险”的修饰符合“确定的”语句。

## 18.1 危险代码

## 18.2 指针类型

```
pointer-type:
    unmanaged-type *
    void *

unmanaged-type:
    value-type
```



# 19. 互用性

这章中描述的属性被用在创建和 COM 程序交互的程序中。

## 19.1 COMImport 属性

当被放在一个类上，COMImport 属性就把这个类标记为一个外部实现的 COM 类。这样的类声明使得可以用一个 C# 名称调用一个 COM 类。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class COMImportAttribute: System.Attribute
    {
        public COMImportAttribute() {...}
    }
}
```

用 COMImport 属性修饰的类要受下面的限制：

- 它必须也被 Guid 属性修饰，它为被引入的 COM 类指定了 CLSID。如果一个类声明包含 COMImport 属性，但是没有包含 Guid 属性，就会发生一个编译时错误。
- 它不能有任何成员。（一个没有参数的公共构造函数会被自动提供。）
- 他必须从 object 类派生。

例子

```
using System.Runtime.InteropServices;
[COMImport, Guid("00020810-0000-0000-C000-000000000046")]
class Worksheet {}
class Test
{
    static void Main() {
        Worksheet w = new Worksheet();    // Creates an Excel worksheet
    }
}
```

声明了一个类 Worksheet，这个类作为一个类从有 CLSID "00020810-0000-0000-C000-000000000046"的 COM 引入。一个 worksheet 实例的实例化造成了一个相应的 COM 实例化。

## 19.2 COMRegisterFunction 属性

一个方法中的 COMRegisterFunction 属性的实现，指示出这个方法应该在 COM 注册过程中被调用。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class COMRegisterFunctionAttribute: System.Attribute
    {
        public ComRegisterFunctionAttribute() {...}
    }
}
```

### 19.3 COMSourceInterfaces 属性

COMSourceInterfaces 属性用来列出引入的联合类中的源接口。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class ComSourceInterfacesAttribute: System.Attribute
    {
        public ComSourceInterfacesAttribute(string value) {...}
        public string Value { get {...} }
    }
}

```

### 19.4 COMVisible 属性

COMVisible 属性用来指定一个类或接口在 COM 中是否可见。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
    public class COMVisibleAttribute: System.Attribute
    {
        public COMVisibleAttribute(bool value) {...}
        public bool Value { get {...} }
    }
}

```

### 19.5 DispId 属性

DispId 属性被用来指定一个 OLE 的自动化 DISPID。一个 DISPID 是一个整数类型数值，它在 dispinterface 中指定一个成员。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Field |
        AttributeTargets.Property)]
    public class DispIdAttribute: System.Attribute
    {
        public DispIdAttribute(int value) {...}
        public int Value { get {...} }
    }
}

```

### 19.6 DllImport 属性

DllImport 属性用来指定包含一个外部方法的实现程序的 dll 的位置。

```

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

```

```

        public bool SetLastError;
        public bool TransformSig;
        public string Value { get {...} }
    }
}

```

特别地，DllImport 属性有下面的行为：

- 它只能用在方法声明中。
- 它有一个位置参数：dllName 参数，指定包含要引入的方法的 dll 的名称。
- 它有五个名称参数：
  - CallingConvention 参数指定为入口点调用的转换。如果没有指定 CallingConvention，默认的 CallingConvention.Winapi 就会被使用。
  - CharSet 参数指定用于入口点的字符集。如果没有 CharSet 被指定，就会使用默认的 CharSet.Auto。
  - EntryPoint 参数给出 dll 中的入口点的名称。如果没有 EntryPoint 被指定，就会使用方法自己的名称。
  - ExactSpelling 参数指定是否 EntryPoint 必须与指出的入口点的拼写相匹配。如果没有指定 ExactSpelling，就会使用默认得 false。
  - SetLastError 参数指出方法是否保存 Win32 的"最后的错误"。如果没有指定 SetLastError，就会使用默认得 false。
  - TransformSig 参数指出是否要为返回数值把方法的签名转换为一个有 HRESULT 返回值和有附加的外部参数的称为 retval 量的返回数值。如果没有指定 TransformSig 数值，就会使用默认得 false。
- 它是一个单次使用属性类。

另外，一个被 DllImport 属性修饰的方法必须有 extern 修饰符。

## 19.7 FieldOffset 属性

FieldOffset 属性被用来为结构指定域的规划。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Field)]
    public class FieldOffsetAttribute: System.Attribute
    {
        public FieldOffsetAttribute(int value) {...}
        public int value { get {...} }
    }
}

```

FieldOffset 属性也许不会被放在一个作为类的成员的域声明中。

## 19.8 GlobalObject 属性

GlobalObject 属性的存在指定一个类是 COM 中的"全局"或 "appobject" 类。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class GlobalObjectAttribute: System.Attribute
    {
        public GlobalObjectAttribute() {...}
    }
}

```

### 19.9 Guid 属性

Guid 属性用来为一个类或一个接口指定一个全局的唯一标识符 (GUID)。这个信息主要用于与 COM 的互用性中。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class
                    | AttributeTargets.Interface
                    | AttributeTargets.Enum
                    | AttributeTargets.Delegate
                    | AttributeTargets.Struct)]
    public class GuidAttribute: System.Attribute
    {
        public GuidAttribute(string value) {...}
        public string Value { get {...} }
    }
}

```

位置字符串参数的形式在编译时被验证。指定一个在不是句法上有效的 GUID 的字符串参数是错误的。

### 19.10 HasDefaultInterface 属性

如果存在，HasDefaultInterface 属性指出一个类有一个默认接口。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class HasDefaultInterfaceAttribute: System.Attribute
    {
        public HasDefaultInterfaceAttribute() {...}
    }
}

```

### 19.11 ImportedFromTypeLib 属性

ImportedFromTypeLib 属性被用来指定一个模块从 COM 类型库中被引入。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Module)]
    public class ImportedFromTypeLib: System.Attribute
    {
        public ImportedFromTypeLib(string value) {...}
        public string Value { get {...} }
    }
}

```

### 19.12 In 和 out 属性

In 和 out 属性被用来为参数提供自定义集合信息。这些集合属性的所有组合都是允许的。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Parameter)]
    public class InAttribute: System.Attribute
    {
        public InAttribute() {...}
    }

    [AttributeUsage(AttributeTargets.Parameter)]
    public class OutAttribute: System.Attribute
    {
        public OutAttribute() {...}
    }
}

```

如果一个参数没有被任何集合属性修饰，那么它就是基于它的参数修饰符，就像下面一样。如果参数没有修饰符，那么集合是 [In]。如果参数有 ref 修饰符，那么集合是 [In, out]。如果参数有 out 修饰符，那么集合是 [Out]。

注意 out 是一个关键字，而 Out 是一个属性。例子

```

class Class1
{
    void M([Out] out int i) {
        ...
    }
}

```

介绍了把 out 当作参数修饰符的使用和在一个属性中的 Out 的使用。

### 19.13 InterfaceType 属性

当放在一个接口上，InterfaceType 属性指定了接口在 COM 被看作的模式。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Interface)]
    public class InterfaceTypeAttribute: System.Attribute
    {
        public InterfaceTypeAttribute(ComInterfaceType value) {...}

        public ComInterfaceType Value { get {...} }
    }
}

```

### 19.14 MarshalAs 属性

MarshalAs 属性被用来描述一个域、方法或参数的集合形式。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method |
        AttributeTargets.Parameter |
        AttributeTargets.Field)]
    public class MarshalAsAttribute: System.Attribute
    {
        public MarshalAsAttribute(UnmanagedType unmanagedType) {...}

        public UnmanagedType ArraySubType;
        public string MarshalCookie;
        public string MarshalType;
        public string MarshalTypeLibGuid;
    }
}

```

```

        public string MarshalUnmanagedType;
        public VarEnum SafeArraySubType;
        public int SizeConst;
        public short SizeParamIndex;
        public int SizeParamMultiplier;
    }
}

```

### 19.15 NoIDispatch 属性

NoIDispatch 属性的存在指示当要输出到 COM 时，类或接口要从 IUnknown 中派生而不是 IDispatch。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
    public class NoIDispatchAttribute: System.Attribute
    {
        public NoIDispatchAttribute() {...}
    }
}

```

### 19.16 NonSerialized 属性

NonSerialized 属性的存在于一个域和属性中，指出那个域或属性要被特殊化。

```

namespace System
{
    [AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)]
    public class NonSerializedAttribute: Attribute
    {
        public NonSerializedAttribute() {...}
    }
}

```

### 19.17 Predeclared 属性

Predeclared 属性的存在表示一个预声明的对象从 COM 引入。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class PredeclaredAttribute: System.Attribute
    {
        public PredeclaredAttribute() {...}
    }
}

```

### 19.18 PreserveSig 属性

PreserveSig 属性被用来把一个方法标记为在 COM 中返回 HRESULT 结果。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Property)]
    public class PreserveSigAttribute: System.Attribute
    {
        public PreserveSigAttribute(bool value) {...}
    }
}

```



```

        public bool value { get {...} }
    }
}

```

PreserveSig 属性被用来指出，通常在互用性调用中发生的 HRESULT/retval 签名转换应该被禁止。

### 19.19 Serializable 属性

Serializable 属性存在于一个类中表示那个类要被特殊化。

```

namespace System
{
    [AttributeUsage(AttributeTargets.Class
                    | AttributeTargets.Delegate
                    | AttributeTargets.Enum
                    | AttributeTargets.Struct)]
    public class SerializableAttribute: System.Attribute
    {
        public SerializableAttribute() {...}
    }
}

```

### 19.20 StructLayout 属性

StructLayout 属性被用来为一个结构指定域的布局。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
    public class StructLayoutAttribute: System.Attribute
    {
        public StructLayoutAttribute(LayoutKind value) {...}
        public CharSet CharSet;
        public bool CheckFastMarshal;
        public int Pack;
        public LayoutKind Value { get {...} }
    }
}

```

如果 LayoutKind.Explicit 被指定，那么在结构中的每个域必须都有 StructOffset 属性。如果 LayoutKind.Explicit 没有被指定，那么 StructOffset 属性的使用就被禁止了。

### 19.21 TypeLibFunc 属性

TypeLibFunc 属性被用来指定 typelib 标记，用于与 COM 互用。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class TypeLibFuncAttribute: System.Attribute
    {
        public TypeLibFuncAttribute(TypeLibFuncFlags value) {...}
        public TypeLibFuncFlags Value { get {...} }
    }
}

```

### 19.22 TypeLibType 属性

TypeLibType 属性被用来指定 typelib 标记，用于与 COM 互用。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
    public class TypeLibTypeAttribute: System.Attribute
    {
        public TypeLibTypeAttribute(TypeLibTypeFlags value) {...}
        public TypeLibTypeFlags Value { get {...} }
    }
}

```

### 19.23 TypeLibVar 属性

TypeLibVar 属性被用来指定 typelib 标记，用于与 COM 互用。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Field)]
    public class TypeLibVarAttribute: System.Attribute
    {
        public TypeLibVarAttribute(TypeLibVarFlags value) {...}
        public TypeLibVarFlags Value { get {...} }
    }
}

```

### 19.24 支持的枚举

```

namespace System.Runtime.InteropServices
{
    public enum CallingConvention
    {
        Winapi = 1,
        cdecl = 2,
        Stdcall = 3,
        Thiscall = 4,
        Fastcall = 5
    }

    public enum CharSet
    {
        None,
        Auto,
        Ansi,
        Unicode
    }

    public enum ComInterfaceType
    {
        InterfaceIsDual = 0,
        InterfaceIsIUnknown = 1,
        InterfaceIsIDispatch = 2,
    }

    public enum LayoutKind
    {
        Sequential,
        Union,
        Explicit,
    }
}

```

```

public enum TypeLibFuncFlags
{
    FRestricted = 1,
    FSource = 2,
    FBindable = 4,
    FRequestEdit = 8,
    FDisplayBind = 16,
    FDefaultBind = 32,
    FHidden = 64,
    FUsesGetLastError = 128,
    FDefaultCollelem = 256,
    FUiDefault = 512,
    FNonBrowsable = 1024,
    FReplaceable = 2048,
    FImmediateBind = 4096
}

public enum TypeLibTypeFlags
{
    FAppObject = 1,
    FCanCreate = 2,
    FLicensed = 4,
    FPreDeclId = 8,
    FHidden = 16,
    FControl = 32,
    FDual = 64,
    FNonExtensible = 128,
    FOleAutomation = 256,
    FRestricted = 512,
    FAggregatable = 1024,
    FReplaceable = 2048,
    FDispatchable = 4096,
    FReverseBind = 8192
}

public enum TypeLibVarFlags
{
    FReadOnly = 1,
    FSource = 2,
    FBindable = 4,
    FRequestEdit = 8,
    FDisplayBind = 16,
    FDefaultBind = 32,
    FHidden = 64,
    FRestricted = 128,
    FDefaultCollelem = 256,
    FUiDefault = 512,
    FNonBrowsable = 1024,
    FReplaceable = 2048,
    FImmediateBind = 4096
}

```

```

public enum UnmanagedType
{
    Bool          = 0x2,
    I1            = 0x3,
    U1            = 0x4,
    I2            = 0x5,
    U2            = 0x6,
    I4            = 0x7,
    U4            = 0x8,
    I8            = 0x9,
    U8            = 0xa,
    R4            = 0xb,
    R8            = 0xc,
    BStr          = 0x13,
    LPStr         = 0x14,
    LPWStr        = 0x15,
    LPTStr        = 0x16,
    ByValTStr     = 0x17,
    Struct        = 0x1b,
    Interface     = 0x1c,
    SafeArray     = 0x1d,
    ByValArray    = 0x1e,
    SysInt        = 0x1f,
    SysUInt       = 0x20,
    VBByRefStr    = 0x22,
    AnsiBStr      = 0x23,
    TBStr         = 0x24,
    VariantBool   = 0x25,
    FunctionPtr   = 0x26,
    LPVoid        = 0x27,
    AsAny         = 0x28,
    RPrecise      = 0x29,
    LPArray       = 0x2a,
    LPStruct      = 0x2b,
    CustomMarshaler = 0x2c,
}
}

```

## 20. 参考

Unicode Consortium. *The Unicode Standard, Version 3.0*. Addison-Wesley, Reading, Massachusetts, 2000, ISBN 0-201-616335-5.

IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985. Available from <http://www.ieee.org>.

ISO/IEC. *C++*. ANSI/ISO/IEC 14882:1998.