C#

语言规范

版本 5.0

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。
请将更正、建议及其他反馈信息发送至 csharp@microsoft.com

注意

© 1999-2012 Microsoft Corporation 。保留所有权利。

Microsoft、Windows、Visual Basic、Visual C# 和 Visual C++ 是 Microsoft Corporation 在美国和/或其他国家/地区的注册商标或商标。

本文提及的其他产品和公司名称可能是其各自所有者的商标。

目录

1. 介绍 1
1.1 Hello world 1
1.2 程序结构 2
1.3 类型和变量 3
1.4 表达式 6
1.5 语句 8
1.6 类和对象 12
1.6.1 成员 12
1.6.2 可访问性 13
1.6.3 类型形参 13
1.6.4 基类 14
1.6.5 字段 14
1.6.6 方法 15
1.6.6.1 参数 15
1.6.6.2 方法体和局部变量 16
1.6.6.3 静态方法和实例方法 17
1.6.6.4 虚方法、重写方法和抽象方法 18
1.6.6.5 方法重载 20
1.6.7 其他函数成员 20
1.6.7.1 构造函数 22
1.6.7.2 属性 23
1.6.7.3 索引器 23
1.6.7.4 事件 24
1.6.7.5 运算符 24
1.6.7.6 析构函数 25
1.7 结构 25
1.8 数组 26
1.9 接口 27
1.10 枚举 28
1.11 委托 29
1.12 特性 30
2. 词法结构 33
2.1 程序 33
2.2 文法 33
2.2.1 文法表示法 33
2.2.2 词法文法 34
2.2.3 句法文法 34
2.3 语法分析 34
2.3.1 行结束符 35
2.3.2 注释 35
2.3.3 空白 36
2.4 标记 37
2.4.1 Unicode 字符转义序列 37

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

2.4.2 标识符 38
2.4.3 关键字 39
2.4.4 文本 40
2.4.4.1 布尔值 40
2.4.4.2 整数 40
2.4.4.3 实数 41
2.4.4.4 字符 42
2.4.4.5 字符串 43
2.4.4.6 null 文本 45
2.4.5 运算符和标点符号 45
2.5 预处理指令 45
2.5.1 条件编译符号 46
2.5.2 预处理表达式 47
2.5.3 声明指令 47
2.5.4 条件编译指令 48
2.5.5 诊断指令 51
2.5.6 区域指令 51
2.5.7 行指令 52
2.5.8 Pragma 指令 52
2.5.8.1 Pragma warning 52

3. 基本概念 55

3.1 应用程序启动 55
3.2 应用程序终止 56
3.3 声明 56
3.4 成员 58
3.4.1 命名空间成员 58
3.4.2 结构成员 58
3.4.3 枚举成员 59
3.4.4 类成员 59
3.4.5 接口成员 59
3.4.6 数组成员 59
3.4.7 委托成员 59
3.5 成员访问 60
3.5.1 已声明可访问性 60
3.5.2 可访问域 60
3.5.3 实例成员的受保护访问 63
3.5.4 可访问性约束 64
3.6 签名和重载 65
3.7 范围 66
3.7.1 名称隐藏 68
3.7.1.1 通过嵌套隐藏 68
3.7.1.2 通过继承隐藏 69
3.8 命名空间和类型名称 70
3.8.1 完全限定名 72

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

3.9 自动内存管理 73
3.10 执行顺序 75

4. 类型 77

4.1 值类型 77
4.1.1 System.ValueType 类型 78
4.1.2 默认构造函数 78
4.1.3 结构类型 79
4.1.4 简单类型 79
4.1.5 整型 80
4.1.6 浮点型 81
4.1.7 decimal 类型 82
4.1.8 bool 类型 82
4.1.9 枚举类型 82
4.1.10 可以为 null 的类型 83
4.2 引用类型 83
4.2.1 类类型 84
4.2.2 对象类型 84
4.2.3 dynamic 类型 84
4.2.4 string 类型 85
4.2.5 接口类型 85
4.2.6 数组类型 85
4.2.7 委托类型 85
4.3 装箱和拆箱 85
4.3.1 装箱转换 85
4.3.2 拆箱转换 86
4.4 构造类型 87
4.4.1 类型实参 88
4.4.2 开放和封闭类型 88
4.4.3 绑定和未绑定类型 89
4.4.4 满足约束 89
4.5 类型形参 90
4.6 表达式树类型 90
4.7 dynamic 类型 91

5. 变量 92

5.1 变量类别 92
5.1.1 静态变量 92
5.1.2 实例变量 92
5.1.2.1 类中的实例变量 92
5.1.2.2 结构中的实例变量 93
5.1.3 数组元素 93
5.1.4 值参数 93
5.1.5 引用参数 93
5.1.6 输出形参 93
5.1.7 局部变量 94

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

5.2 默认值 94
5.3 明确赋值 95
5.3.1 初始已赋值变量 95
5.3.2 初始未赋值变量 96
5.3.3 确定明确赋值的细则 96
5.3.3.1 一般语句规则 96
5.3.3.2 块语句、checked 和 unchecked 语句 97
5.3.3.3 表达式语句 97
5.3.3.4 声明语句 97
5.3.3.5 if 语句 97
5.3.3.6 switch 语句 97
5.3.3.7 while 语句 98
5.3.3.8 do 语句 98
5.3.3.9 for 语句 98
5.3.3.10 break、continue 和 goto 语句 98
5.3.3.11 throw 语句 99
5.3.3.12 return 语句 99
5.3.3.13 try-catch 语句 99
5.3.3.14 try-finally 语句 99
5.3.3.15 try-catch-finally 语句 100
5.3.3.16 foreach 语句 100
5.3.3.17 using 语句 101
5.3.3.18 lock 语句 101
5.3.3.19 yield 语句 101
5.3.3.20 简单表达式的一般规则 101
5.3.3.21 带有嵌入表达式的表达式的一般规则 101
5.3.3.22 调用表达式和对象创建表达式 102
5.3.3.23 简单赋值表达式 102
5.3.3.24 && 表达式 102
5.3.3.25 || 表达式 103
5.3.3.26 !表达式 104
5.3.3.27 ?? 表达式 104
5.3.3.28 ?: 表达式 105
5.3.3.29 匿名函数 105
5.4 变量引用 106
5.5 变量引用的原子性 106
6. 转换 107
6.1 隐式转换 107
6.1.1 标识转换 108
6.1.2 隐式数值转换 108
6.1.3 隐式枚举转换 108
6.1.4 可以为 null 的隐式转换 108
6.1.5 null 文本转换 109
6.1.6 隐式引用转换 109

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

6.1.7 装箱转换 110
6.1.8 隐式动态转换 110
6.1.9 隐式常量表达式转换 110
6.1.10 涉及类型形参的隐式转换 110
6.1.11 用户定义的隐式转换 111
6.1.12 匿名函数转换和方法组转换 111
6.2 显式转换 111
6.2.1 显式数值转换 112
6.2.2 显式枚举转换 113
6.2.3 可以为 null 的显式转换 113
6.2.4 显式引用转换 114
6.2.5 拆箱转换 115
6.2.6 显式动态转换 115
6.2.7 涉及类型参数的显式转换 116
6.2.8 用户定义的显式转换 117
6.3 标准转换 117
6.3.1 标准隐式转换 117
6.3.2 标准显式转换 117
6.4 用户定义的转换 117
6.4.1 允许的用户定义转换 117
6.4.2 提升转换运算符 118
6.4.3 用户定义转换的计算 118
6.4.4 用户定义的隐式转换 119
6.4.5 用户定义的显式转换 120
6.5 匿名函数转换 121
6.5.1 匿名函数转换为委托类型的计算 122
6.5.2 匿名函数转换为表达式树类型的计算 123
6.5.3 实现示例 123
6.6 方法组转换 125
7. 表达式 129
7.1 表达式的分类 129
7.1.1 表达式的值 130
7.2 静态和动态绑定 130
7.2.1 绑定时间 131
7.2.2 动态绑定 131
7.2.3 构成表达式的类型 131
7.3 运算符 131
7.3.1 运算符的优先级和顺序关联性 132
7.3.2 运算符重载 133
7.3.3 一元运算符重载决策 134
7.3.4 二元运算符重载决策 134
7.3.5 候选用户定义运算符 134
7.3.6 数值提升 135
7.3.6.1 一元数值提升 135

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.3.6.2 二元数值提升 135
7.3.7 提升运算符 136
7.4 成员查找 137
7.4.1 基类型 138
7.5 函数成员 138
7.5.1 实参列表 140
7.5.1.1 对应形参 141
7.5.1.2 实参列表的运行时计算 142
7.5.2 类型推断 143
7.5.2.1 第一阶段 144
7.5.2.2 第二阶段 144
7.5.2.3 输入类型 145
7.5.2.4 输出类型 145
7.5.2.5 依赖 145
7.5.2.6 输出类型推断 145
7.5.2.7 参数类型显式推断 145
7.5.2.8 精确推断 145
7.5.2.9 下限推断 145
7.5.2.10 上限推断 146
7.5.2.11 固定 147
7.5.2.12 推断返回类型 147
7.5.2.13 方法组转换的类型推断 148
7.5.2.14 查找一组表达式的最通用类型 149
7.5.3 重载决策 149
7.5.3.1 适用函数成员 149
7.5.3.2 更好的函数成员 150
7.5.3.3 表达式的更佳转换 151
7.5.3.4 类型的更佳转换 151
7.5.3.5 更佳转换目标 151
7.5.3.6 泛型类中的重载 152
7.5.4 动态重载决策的编译时检查 152
7.5.5 函数成员调用 153
7.5.5.1 已装箱实例上的调用 154
7.6 基本表达式 154
7.6.1 文本 155
7.6.2 简单名称 155
7.6.2.1 块中的固定含义 156
7.6.3 带括号的表达式 157
7.6.4 成员访问 157
7.6.4.1 相同的简单名称和类型名称 159
7.6.4.2 语法多义性 159
7.6.5 调用表达式 160
7.6.5.1 方法调用 161
7.6.5.2 扩展方法调用 162
7.6.5.3 委托调用 164

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.6.6 元素访问 164
7.6.6.1 数组访问 165
7.6.6.2 索引器访问 165
7.6.7 this 访问 166
7.6.8 基访问 166
7.6.9 后缀增量和后缀减量运算符 167
7.6.10 new 运算符 168
7.6.10.1 对象创建表达式 168
7.6.10.2 对象初始值设定项 169
7.6.10.3 集合初始值设定项 171
7.6.10.4 数组创建表达式 172
7.6.10.5 委托创建表达式 174
7.6.10.6 匿名对象创建表达式 175
7.6.11 typeof 运算符 177
7.6.12 checked 和 unchecked 运算符 179
7.6.13 默认值表达式 181
7.6.14 匿名方法表达式 181
7.7 一元运算符 181
7.7.1 一元加运算符 181
7.7.2 一元减运算符 182
7.7.3 逻辑否定运算符 182
7.7.4 按位求补运算符 182
7.7.5 前缀增量和减量运算符 183
7.7.6 强制转换表达式 183
7.7.7 Await 表达式 184
7.7.7.1 可等待的表达式 185
7.7.7.2 await 表达式的分类 185
7.7.7.3 await 表达式的运行时计算 185
7.8 算术运算符 186
7.8.1 乘法运算符 186
7.8.2 除法运算符 187
7.8.3 余数运算符 188
7.8.4 加法运算符 189
7.8.5 减法运算符 191
7.9 移位运算符 192
7.10 关系和类型测试运算符 194
7.10.1 整数比较运算符 194
7.10.2 浮点比较运算符 195
7.10.3 小数比较运算符 196
7.10.4 布尔相等运算符 196
7.10.5 枚举比较运算符 196
7.10.6 引用类型相等运算符 196
7.10.7 字符串相等运算符 198
7.10.8 委托相等运算符 198
7.10.9 相等运算符和 null 199

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.10.10 is 运算符 199
7.10.11 as 运算符 199
7.11 逻辑运算符 200
7.11.1 整数逻辑运算符 201
7.11.2 枚举逻辑运算符 201
7.11.3 布尔逻辑运算符 201
7.11.4 可以为 null 的布尔逻辑运算符 202
7.12 条件逻辑运算符 202
7.12.1 布尔条件逻辑运算符 203
7.12.2 用户定义的条件逻辑运算符 203
7.13 空合并运算符 203
7.14 条件运算符 204
7.15 匿名函数表达式 205
7.15.1 匿名函数签名 207
7.15.2 匿名函数体 207
7.15.3 重载决策 207
7.15.4 匿名函数与动态绑定 208
7.15.5 外层变量 208
7.15.5.1 捕获的外层变量 208
7.15.5.2 局部变量实例化 209
7.15.6 匿名函数表达式计算 211
7.16 查询表达式 211
7.16.1 查询表达式的多义性 213
7.16.2 查询表达式转换 213
7.16.2.1 带继续符的 select 和 groupby 子句 213
7.16.2.2 显式范围变量类型 214
7.16.2.3 退化查询表达式 214
7.16.2.4 from、let、where、join 和 orderby 子句 215
7.16.2.5 select 子句 218
7.16.2.6 Groupby 子句 219
7.16.2.7 透明标识符 219
7.16.3 查询表达式模式 220
7.17 赋值运算符 221
7.17.1 简单赋值 222
7.17.2 复合赋值 224
7.17.3 事件赋值 225
7.18 表达式 225
7.19 常量表达式 225
7.20 布尔表达式 227
8. 语句 229
8.1 结束点和可到达性 229
8.2 块 231
8.2.1 语句列表 231
8.3 空语句 232

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

8.4 标记语句 232
8.5 声明语句 233
8.5.1 局部变量声明 233
8.5.2 局部常量声明 234
8.6 表达式语句 235
8.7 选择语句 235
8.7.1 if 语句 235
8.7.2 switch 语句 236
8.8 迭代语句 239
8.8.1 while 语句 239
8.8.2 do 语句 240
8.8.3 for 语句 240
8.8.4 foreach 语句 241
8.9 跳转语句 244
8.9.1 break 语句 245
8.9.2 continue 语句 246
8.9.3 goto 语句 246
8.9.4 return 语句 247
8.9.5 throw 语句 248
8.10 try 语句 249
8.11 checked 语句和 unchecked 语句 252
8.12 lock 语句 252
8.13 using 语句 253
8.14 yield 语句 255

9. 命名空间 257

9.1 编译单元 257
9.2 命名空间声明 257
9.3 Extern 别名 258
9.4 using 指令 259
9.4.1 using 别名指令 259
9.4.2 Using 命名空间指令 262
9.5 命名空间成员 264
9.6 类型声明 264
9.7 命名空间别名限定符 265
9.7.1 别名的唯一性 266

10. 类 267

10.1 类声明 267
10.1.1 类修饰符 267
10.1.1.1 抽象类 268
10.1.1.2 密封类 268
10.1.1.3 静态类 268
10.1.2 分部修饰符 269
10.1.3 类型形参 269
10.1.4 类基本规范 269

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.1.4.1 基类 270
10.1.4.2 接口实现 271
10.1.5 类型形参约束 272
10.1.6 类体 276
10.2 分部类型 276
10.2.1 特性 276
10.2.2 修饰符 277
10.2.3 类型形参和约束 277
10.2.4 基类 277
10.2.5 基接口 278
10.2.6 成员 278
10.2.7 分部方法 278
10.2.8 名称绑定 281
10.3 类成员 281
10.3.1 实例类型 282
10.3.2 构造类型的成员 283
10.3.3 继承 283
10.3.4 new 修饰符 284
10.3.5 访问修饰符 284
10.3.6 构成类型 285
10.3.7 静态成员和实例成员 285
10.3.8 嵌套类型 286
10.3.8.1 完全限定名 286
10.3.8.2 已声明可访问性 286
10.3.8.3 隐藏 287
10.3.8.4 this 访问 287
10.3.8.5 对包含类型的私有和受保护成员的访问 288
10.3.8.6 泛型类中的嵌套类型 289
10.3.9 保留成员名称 289
10.3.9.1 为属性保留的成员名称 290
10.3.9.2 为事件保留的成员名称 291
10.3.9.3 为索引器保留的成员名称 291
10.3.9.4 为析构函数保留的成员名称 291
10.4 常量 291
10.5 字段 292
10.5.1 静态字段和实例字段 294
10.5.2 只读字段 294
10.5.2.1 对常量使用静态只读字段 295
10.5.2.2 常量和静态只读字段的版本控制 295
10.5.3 可变字段 296
10.5.4 字段初始化 297
10.5.5 变量初始值设定项 297
10.5.5.1 静态字段初始化 298
10.5.5.2 实例字段初始化 299
10.6 方法 300

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.6.1 方法形参 301
10.6.1.1 值参数 303
10.6.1.2 引用参数 303
10.6.1.3 输出形参 304
10.6.1.4 形参数组 305
10.6.2 静态方法和实例方法 307
10.6.3 虚方法 308
10.6.4 重写方法 309
10.6.5 密封方法 311
10.6.6 抽象方法 312
10.6.7 外部方法 313
10.6.8 分部方法 314
10.6.9 扩展方法 314
10.6.10 方法体 315
10.6.11 方法重载 315
10.7 属性 315
10.7.1 静态属性和实例属性 316
10.7.2 访问器 317
10.7.3 自动实现的属性 321
10.7.4 可访问性 322
10.7.5 虚、密封、重写和抽象访问器 323
10.8 事件 325
10.8.1 类似字段的事件 327
10.8.2 事件访问器 328
10.8.3 静态事件和实例事件 329
10.8.4 虚、密封、重写和抽象访问器 329
10.9 索引器 330
10.9.1 索引器重载 333
10.10 运算符 333
10.10.1 一元运算符 335
10.10.2 二元运算符 335
10.10.3 转换运算符 336
10.11 实例构造函数 338
10.11.1 构造函数初始值设定项 339
10.11.2 实例变量初始值设定项 340
10.11.3 构造函数执行 340
10.11.4 默认构造函数 342
10.11.5 私有构造函数 342
10.11.6 可选的实例构造函数形参 343
10.12 静态构造函数 343
10.13 析构函数 345
10.14 迭代器 347
10.14.1 枚举器接口 347
10.14.2 可枚举接口 347
10.14.3 产生类型 347

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.14.4 枚举器对象 347
10.14.4.1 MoveNext 方法 348
10.14.4.2 Current 属性 349
10.14.4.3 Dispose 方法 349
10.14.5 可枚举对象 349
10.14.5.1 GetEnumerator 方法 350
10.14.6 实现示例 350
10.15 异步函数 355
10.15.1 返回任务的异步函数计算 356
10.15.2 返回 void 的异步函数计算 356

11. 结构 357

11.1 结构声明 357
11.1.1 结构修饰符 357
11.1.2 分部修饰符 357
11.1.3 结构接口 358
11.1.4 结构体 358
11.2 结构成员 358
11.3 类和结构的区别 358
11.3.1 值语义 359
11.3.2 继承 359
11.3.3 赋值 360
11.3.4 默认值 360
11.3.5 装箱和拆箱 360
11.3.6 this 的含义 362
11.3.7 字段初始值设定项 362
11.3.8 构造函数 362
11.3.9 析构函数 363
11.3.10 静态构造函数 363
11.4 结构示例 364
11.4.1 数据库整数类型 364
11.4.2 数据库布尔类型 365

12. 数组 369

12.1 数组类型 369
12.1.1 System.Array 类型 370
12.1.2 数组和泛型 IList 接口 370
12.2 数组创建 370
12.3 数组元素访问 371
12.4 数组成员 371
12.5 数组协变 371
12.6 数组初始值设定项 371

13. 接口 373

13.1 接口声明 373
13.1.1 接口修饰符 373

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

13.1.2 分部修饰符 373
13.1.3 Variant 类型形参列表 374
13.1.3.1 变化安全性 374
13.1.3.2 变化转换 375
13.1.4 基接口 375
13.1.5 接口体 376
13.2 接口成员 376
13.2.1 接口方法 377
13.2.2 接口属性 377
13.2.3 接口事件 378
13.2.4 接口索引器 378
13.2.5 接口成员访问 378
13.3 完全限定接口成员名 380
13.4 接口实现 380
13.4.1 显式接口成员实现 381
13.4.2 所实现接口的唯一性 383
13.4.3 泛型方法实现 384
13.4.4 接口映射 385
13.4.5 接口实现继承 387
13.4.6 接口重新实现 389
13.4.7 抽象类和接口 390
14. 枚举 391
14.1 枚举声明 391
14.2 枚举修饰符 391
14.3 枚举成员 392
14.4 System.Enum 类型 394
14.5 枚举值和运算 394
15. 委托 395
15.1 委托声明 395
15.2 委托兼容性 397
15.3 委托实例化 397
15.4 委托调用 398
16. 异常 401
16.1 导致异常的原因 401
16.2 System.Exception 类 401
16.3 异常的处理方式 401
16.4 公共异常类 402
17. 特性 403
17.1 特性类 403
17.1.1 特性用法 403
17.1.2 定位和命名参数 404
17.1.3 特性参数类型 405

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

17.2 特性说明 405
17.3 特性实例 410
17.3.1 特性的编译 410
17.3.2 特性实例的运行时检索 411
17.4 保留特性 411
17.4.1 AttributeUsage 特性 411
17.4.2 Conditional 特性 412
17.4.2.1 条件方法 412
17.4.2.2 条件特性类 414
17.4.3 Obsolete 特性 415
17.4.4 调用方信息特性 416
17.4.4.1 CallerLineNumber 特性 417
17.4.4.2 CallerFilePath 特性 417
17.4.4.3 CallerMemberName 特性 417
17.5 互操作的特性 417
17.5.1 与 COM 和 Win32 组件的互操作 418
17.5.2 与其他 .NET 语言的互操作 418
17.5.2.1 IndexerName 特性 418
18. 不安全代码 419
18.1 不安全上下文 419
18.2 指针类型 421
18.3 固定和可移动变量 424
18.4 指针转换 424
18.4.1 指针数组 425
18.5 表达式中的指针 426
18.5.1 指针间接寻址 427
18.5.2 指针成员访问 427
18.5.3 指针元素访问 428
18.5.4 address-of 运算符 429
18.5.5 指针递增和递减 429
18.5.6 指针算术运算 430
18.5.7 指针比较 431
18.5.8 sizeof 运算符 431
18.6 fixed 语句 432
18.7 固定大小缓冲区 435
18.7.1 固定大小缓冲区的声明 435
18.7.2 表达式中的固定大小缓冲区 436
18.7.3 明确赋值 检查 437
18.8 堆栈分配 437
18.9 动态内存分配 438
A. 文档注释 441
A.1 介绍 441
A.2 建议的标记 442
A.2.1 <c> \t "See <c>" \b 443

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

A.2.2 <code> \t "See <code>" \b ... 443
A.2.3 <example> \t "See <example>" \b ... 444
A.2.4 <exception> \t "See <exception>" \b ... 444
A.2.5 <include> ... 445
A.2.6 <list> \t "See <list>" \b ... 446
A.2.7 <para> \t "See <para>" \b ... 446
A.2.8 <param> \t "See <param>" \b ... 447
A.2.9 <paramref> \t "See <paramref>" \b ... 447
A.2.10 <permission> \t "See <permission>" \b ... 448
A.2.11 <remark> \t "See <remarks>" \b ... 448
A.2.12 <returns> \t "See <returns>" \b ... 448
A.2.13 <see> \t "See <see>" \b ... 449
A.2.14 <seealso> \t "See <seealso>" \b ... 449
A.2.15 <summary> \t "See <summary>" \b ... 450
A.2.16 <value> \t "See <value>" \b ... 450
A.2.17 <typeparam> ... 450
A.2.18 <typeparamref> ... 451
A.3 处理文档文件 ... 451
A.3.1 ID 字符串格式 ... 451
A.3.2 ID 字符串示例 ... 452
A.4 示例 ... 456
A.4.1 C# 源代码 ... 456
A.4.2 生成的 XML ... 458

B. 语法 ... 462

B.1 词法文法 ... 462
B.1.1 行结束符 ... 462
B.1.2 注释 ... 462
B.1.3 空白 ... 463
B.1.4 标记 ... 463
B.1.5 Unicode 字符转义序列 ... 463
B.1.6 标识符 ... 463
B.1.7 关键字 ... 465
B.1.8 文本 ... 465
B.1.9 运算符和标点符号 ... 467
B.1.10 预处理指令 ... 467
B.2 句法文法 ... 470
B.2.1 基本概念 ... 470
B.2.2 类型 ... 470
B.2.3 变量 ... 471
B.2.4 表达式 ... 471
B.2.5 语句 ... 478
B.2.6 命名空间 ... 482
B.2.7 类 ... 482
B.2.8 结构 ... 490
B.2.9 数组 ... 490
B.2.10 接口 ... 491

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

B.2.11 枚举 492
B.2.12 委托 493
B.2.13 特性 493
B.3 不安全代码的语法扩展 494
C. 参考资料 499

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

1. 介绍

C#（读作“See Sharp”）是一种简洁、现代、面向对象且类型安全的编程语言。C# 起源于 C 语言家族，因此，对于 C、C++ 和 Java 程序员，可以很快熟悉这种新的语言。C# 已经分别由 ECMA International 和 ISO/IEC 组织接受并确立了标准，它们分别是 ECMA-334 标准和 ISO/IEC 23270 标准。Microsoft 用于 .NET Framework 的 C# 编译器就是根据这两个标准实现的。

C# 是面向对象的语言，然而 C# 进一步提供了对面向组件 (component-oriented) 编程的支持。现代软件设计日益依赖于自包含和自描述功能包形式的软件组件。这种组件的关键在于，它们通过属性、方法和事件来提供编程模型；它们具有提供了关于组件的声明性信息的特性；同时，它们还编入了自己的文档。C# 提供的语言构造直接支持这些概念，这使得 C# 语言自然而然成为创建和使用软件组件之选。

有助于构造健壮、持久的应用程序的若干 C# 特性：垃圾回收 (Garbage collection) 将自动回收不再使用的对象所占用的内存；异常处理 (exception handling) 提供了结构化和可扩展的错误检测和恢复方法；类型安全 (type-safe) 的语言设计则避免了读取未初始化的变量、数组索引超出边界或执行未经检查的类型强制转换等情形。

C# 具有一个同一类型系统 (unified type system)。所有 C# 类型（包括诸如 `int` 和 `double` 之类的基元类型）都继承于单个根类型：`object`。因此，所有类型都共享一组通用操作，并且任何类型的值都能够以一致的方式进行存储、传递和操作。此外，C# 同时支持用户定义的引用类型和值类型，既允许对象的动态分配，也允许轻量结构的内联存储。

为了确保 C# 程序和库能够以兼容的方式逐步演进，C# 的设计中充分强调了版本控制 (versioning)。许多编程语言不太重视这一点，导致采用那些语言编写的程序常常因为其所依赖的库的更新而无法正常工作。C# 的设计在某些方面直接考虑到版本控制的需要，其中包括单独使用的 `virtual` 和 `override` 修饰符、方法重载决策规则以及对显式接口成员声明的支持。

本章的其余部分将描述 C# 语言的基本特征。尽管后面的章节会更为详尽，有时甚至逻辑缜密地对规则和例外情况进行描述，但本章的描述力求简洁明了，因而难免会牺牲完整性。这样做是为了向读者提供关于该语言的概貌，一方面使读者能尽快上手编写程序，另一方面为阅读后续章节提供指导。

1.1 Hello world

按照约定俗成的惯例，我们先从“Hello, World”程序着手介绍这一编程语言。下面是它的 C# 程序：

```
using System;

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

C# 源文件的扩展名通常是 `.cs`。假定“Hello, World”程序存储在文件 `hello.cs` 中，可以使用下面的命令行调用 Microsoft C# 编译器编译这个程序：

```
csc hello.cs
```

编译后将生成一个名为 `hello.exe` 的可执行程序集。当此应用程序运行时，输出结果如下：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
Hello, World
```

“Hello, World”程序的开头是一个 `using` 指令，它引用了 `System` 命名空间。命名空间提供了一种分层的方式来组织 C# 程序和库。命名空间中包含有类型及其他命名空间 — 例如，`System` 命名空间包含若干类型（如此程序中引用的 `Console` 类）以及若干其他命名空间（如 `IO` 和 `Collections`）。如果使用 `using` 指令引用了某一给定命名空间，就可以通过非限定方式使用作为该命名空间成员的类型。正是由于使用了 `using` 指令，该程序才可以采用 `Console.WriteLine` 这一简化形式代替 `System.Console.WriteLine`。

“Hello, World”程序中声明的 `Hello` 类只有一个成员，即名为 `Main` 的方法。`Main` 方法是使用 `static` 修饰符声明的。实例方法可以使用关键字 `this` 来引用特定的封闭对象实例，而静态方法的操作不需要引用特定对象。按照惯例，名为 `Main` 的静态方法将作为程序的入口点。

该程序的输出由 `System` 命名空间中的 `Console` 类的 `WriteLine` 方法产生。此类由 .NET Framework 类库提供，默认情况下，Microsoft C# 编译器自动引用该类库。注意，C# 语言本身没有单独的运行库。事实上，.NET Framework 就是 C# 运行库。

1.2 程序结构

C# 中的组织结构的关键概念是程序 (program)、命名空间 (namespace)、类型 (type)、成员 (member) 和程序集 (assembly)。C# 程序由一个或多个源文件组成。程序中声明类型，类型包含成员，并且可按命名空间进行组织。类和接口就是类型的示例。字段 (field)、方法、属性和事件是成员的示例。在编译 C# 程序时，它们被物理地打包为程序集。程序集通常具有文件扩展名 `.exe` 或 `.dll`，具体取决于它们是实现应用程序 (application) 还是实现库 (library)。

下面的示例

```
using System;

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

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

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

      class Entry
      {
         public Entry next;
         public object data;

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在名为 Acme.Collections 的命名空间中声明了一个名为 Stack 的类。Acme.Collections.Stack 是此类的完全限定名。该类包含几个成员：一个名为 top 的字段，两个分别名为 Push 和 Pop 的方法和一个名为 Entry 的嵌套类。Entry 类还包含三个成员：一个名为 next 的字段、一个名为 data 的字段和一个构造函数。假定将此示例的源代码存储在文件 acme.cs 中，执行以下命令行：

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

将此示例编译为一个库（没有 Main 入口点的代码），并产生一个名为 acme.dll 的程序集。

程序集包含中间语言 (Intermediate Language, IL) 指令形式的可执行代码和元数据 (metadata) 形式的符号信息。在执行程序集之前，.NET 公共语言运行时的实时 (JIT) 编译器将程序集中的 IL 代码自动转换为特定于处理器的代码。

由于程序集是一个自描述的功能单元，它既包含代码又包含元数据，因此，C# 中不需要 #include 指令和头文件。若要在 C# 程序中使用某特定程序集中包含的公共类型和成员，只需在编译程序时引用该程序集即可。例如，下面的程序将使用来自 acme.dll 程序集的 Acme.Collections.Stack 类：

```
using System;
using Acme.Collections;

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

如果此程序存储在文件 test.cs 中，那么在编译 test.cs 时，可以使用编译器的 /r 选项引用 acme.dll 程序集：

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

这样将创建名为 test.exe 的可执行程序集，运行结果如下：

```
100
10
1
```

C# 允许将一个程序的源文本存储在多个源文件中。在编译多个文件组成的 C# 程序时，所有源文件将一起处理，并且源文件可以自由地相互引用 — 从概念上讲，就像是在处理之前将所有源文件合并为一个大文件。C# 中从不需要前向声明，因为除了极少数的例外情况，声明顺序无关紧要。C# 不限制一个源文件只能声明一个公共类型，也不要求源文件的名称与该源文件中声明的类型匹配。

1.3 类型和变量

C# 中的类型有两种：值类型 (value type) 和引用类型 (reference type)。值类型的变量直接包含它们的数据，而引用类型的变量存储对它们的数据的引用，后者称为对象。对于引用类型，两个变量可能引用同一个对象，因此对一个变量的操作可能影响另一个变量所引用的对象。对于值类型，每个变量都有它们自己的数据副本（除 ref 和 out 参数变量外），因此对一个变量的操作不可能影响另一个变量。

C# 的值类型进一步划分为简单类型 (simple type)、枚举类型 (enum type)、结构类型 (struct type) 和可以为 null 的类型 (nullable type)，C# 的引用类型进一步划分为类类型 (class type)、接口类型 (interface type)、数组类型 (array type) 和委托类型 (delegate type)。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

下表为 C# 类型系统的概述。

类别		说明
值 类型	简单类型	有符号整型：sbyte、short、int、long
		无符号整型：byte、ushort、uint、ulong
		Unicode 字符：char
		IEEE 浮点：float、double
		高精度小数型：decimal
		布尔：bool
	枚举类型	enum E {...} 形式的用户定义的类型
	结构类型	struct S {...} 形式的用户定义的类型
	可以为 null 的类型	其他所有具有 null 值的值类型的扩展
引用 类型	类类型	所有其他类型的最终基类：object
		Unicode 字符串：string
		class C {...} 形式的用户定义的类型
	接口类型	interface I {...} 形式的用户定义的类型
	数组类型	一维和多维数组，例如 int[] 和 int[,]
	委托类型	delegate int D(...) 形式的用户定义的类型

八种整型类型分别支持 8 位、16 位、32 位和 64 位整数值的有符号和无符号的形式。

两种浮点类型：float 和 double，分别使用 32 位单精度和 64 位双精度的 IEEE 754 格式表示。

decimal 类型是 128 位的数据类型，适合用于财务计算和货币计算。

C# 的 bool 类型用于表示值为 true 或 false 的布尔值。

在 C# 中，字符和字符串处理使用 Unicode 编码。char 类型表示一个 UTF-16 编码单元，string 类型表示 UTF-16 编码单元的序列。

下表总结了 C# 的数值类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

类别	位数	类型	范围/精度
有符号整型	8	sbyte	–128...127
	16	short	–32,768...32,767
	32	int	–2,147,483,648...2,147,483,647
	64	long	–9,223,372,036,854,775,808...9,223,372,036,854,775,807
无符号整型	8	byte	0...255
	16	ushort	0...65,535
	32	uint	0...4,294,967,295
	64	ulong	0...18,446,744,073,709,551,615
浮点	32	float	1.5×10^{-45} 至 3.4×10^{38}，7 位精度
	64	double	1.5×10^{-324} 至 1.7×10^{308}，15 位精度
十进制小数	128	decimal	1.0×10^{-28} 至 7.9×10^{28}，28 位精度

C# 程序使用类型声明 (type declaration) 创建新类型。类型声明指定新类型的名称和成员。在 C# 类型分类中，有五类是用户可定义的：类类型 (class type)、结构类型 (struct type)、接口类型 (interface type)、枚举类型 (enum type) 和委托类型 (delegate type)。

类类型定义了一个包含数据成员（字段）和函数成员（方法、属性等）的数据结构。类类型支持单一继承和多态，这些是派生类可用来扩展和专用化基类的机制。

结构类型与类类型相似，表示一个带有数据成员和函数成员的结构。但是，与类不同，结构是一种值类型，并且不需要堆分配。结构类型不支持用户指定的继承，并且所有结构类型都隐式地从类型 `object` 继承。

接口类型定义了一个协定，作为一个公共函数成员的命名集。实现某个接口的类或结构必须提供该接口的函数成员的实现。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

委托类型表示对具有特定参数列表和返回类型的方法的引用。通过委托，我们能够将方法作为实体赋值给变量和作为参数传递。委托类似于在其他某些语言中的函数指针的概念，但是与函数指针不同，委托是面向对象的，并且是类型安全的。

类类型、结构类型、接口类型和委托类型都支持泛型，因此可以通过其他类型将其参数化。

枚举类型是具有命名常量的独特的类型。每种枚举类型都具有一个基础类型，该基础类型必须是八种整型之一。枚举类型的值集和它的基础类型的值集相同。

C# 支持由任何类型组成的一维和多维数组。与以上列出的类型不同，数组类型不必声明就可以使用。实际上，数组类型是通过在某个类型名后加一对方括号来构造的。例如，`int[]` 是一维 `int` 数组，`int[,]` 是二维 `int` 数组，`int[][]` 是一维 `int` 数组的一维数组。

可以为 null 的类型也不必声明就可以使用。对于每个不可以为 null 的值类型 `T`，都有一个相应的可以为 null 的类型 `T?`，该类型可以容纳附加值 `null`。例如，`int?` 类型可以容纳任何 32 位整数或 `null` 值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

C# 的类型系统是统一的，因此任何类型的值都可以按对象处理。C# 中的每个类型直接或间接地从 object 类类型派生，而 object 是所有类型的最终基类。引用类型的值都被视为object 类型，被简单地当作对象来处理。值类型的值则通过对其执行装箱和拆箱*操作*按对象处理。下面的示例将 int 值转换为 object，然后又转换回 int。

```
using System;

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

当将值类型的值转换为类型 object 时，将分配一个对象实例（也称为“箱子”）以包含该值，并将值复制到该箱子中。反过来，当将一个 object 引用强制转换为值类型时，将检查所引用的对象是否含有正确的值类型，如果有，则将箱子中的值复制出来。

C# 的统一类型系统实际上意味着值类型可以“按需”转换为对象。因为统一，所以使用类型 object 的通用库可以与引用类型和值类型一同使用。

C# 中存在几种变量 (variable)，包括字段、数组元素、局部变量和参数。变量表示了存储位置，并且每个变量都有一个类型，以决定什么样的值能够存入变量，如下表所示。

变量类型	可能的内容
不可以为 null 的值类型	类型完全相同的值
可以为 null 的值类型	null 值或类型完全相同的值
object	null 引用、对任何引用类型的对象的引用，或者对任何值类型的装箱值的引用
类类型	null 引用、对该类类型的实例的引用，或者对从该类类型派生的类的实例的引用
接口类型	null 引用、对实现该接口类型的类类型的实例的引用，或者对实现该接口类型的值类型的装箱值的引用
数组类型	null 引用、对该数组类型的实例的引用，或者对兼容数组类型的实例的引用
委托类型	null 引用或对该委托类型的实例的引用

1.4 表达式

表达式由操作数 (operand) 和运算符 (operator) 构成。表达式的运算符指示对操作数适用什么样的运算。运算符的示例包括+、-、*、/ 和 new。操作数的示例包括文本、字段、局部变量和表达式。

当表达式包含多个运算符时，运算符的优先级 (precedence) 控制各运算符的计算顺序。例如，表达式 x + y * z 按 x + (y * z) 计算，因为 * 运算符的优先级高于 + 运算符。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

大多数运算符都可以重载(overload)。运算符重载允许指定用户定义的运算符实现来执行运算，这些运算的操作数中至少有一个，甚至所有操作数都属于用户定义的类类型或结构类型。

下表总结了 C# 运算符，并按优先级从高到低的顺序列出各运算符类别。同一类别中的运算符优先级相同。

类别	表达式	说明
基本	`x.m`	成员访问
	`x(...)`	方法和委托调用
	`x[...]`	数组和索引器访问
	`x++`	后增量
	`x--`	后减量
	`new T(...)`	对象和委托创建
	`new T(...){...}`	使用初始值设定项创建对象
	`new {...}`	匿名对象初始值设定项
	`new T[...]`	数组创建
	`typeof(T)`	获取 `T` 的 `System.Type` 对象
	`checked(x)`	在 checked 上下文中计算表达式
	`unchecked(x)`	在 unchecked 上下文中计算表达式
	`default(T)`	获取类型 `T` 的默认值
	`delegate {...}`	匿名函数（匿名方法）
一元	`+x`	恒等
	`-x`	求相反数
	`!x`	逻辑求反
	`~x`	按位求反
	`++x`	前增量
	`--x`	前减量
	`(T)x`	将 `x` 显式转换为类型 `T`
	`await x`	异步等待 `x` 完成
乘法	`x * y`	乘法
	`x / y`	除法
	`x % y`	求余

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

加减	`x + y`	加法、字符串串联、委托组合
	`x - y`	减法、委托移除
移位	`x << y`	左移
	`x >> y`	右移
关系和类型检测	`x < y`	小于
	`x > y`	大于
	`x <= y`	小于或等于
	`x >= y`	大于或等于
	`x is T`	如果 `x` 为 `T`，则返回 `true`，否则返回 `false`
	`x as T`	返回转换为类型 `T` 的 `x`，如果 `x` 不是 `T` 则返回 `null`
相等	`x == y`	等于
	`x != y`	不等于
逻辑 AND	`x & y`	整型按位 AND，布尔逻辑 AND
逻辑 XOR	`x ^ y`	整型按位 XOR，布尔逻辑 XOR
逻辑 OR	`x | y`	整型按位 OR，布尔逻辑 OR
条件 AND	`x && y`	仅当 `x` 为 `true` 时，才对 `y` 求值
条件 OR	`x || y`	仅当 `x` 为 `false` 时，才对 `y` 求值
null 合并	`x ?? y`	如果 `x` 为 `null`，则计算结果为 `y`，否则计算结果为 `x`
条件	`x ? y : z`	如果 `x` 为 `true`，则对 `y` 求值；如果 `x` 为 `false`，则对 `z` 求值
赋值或匿名函数	`x = y`	赋值
	`x` *op*`= y`	复合赋值；支持的运算符有： `*= /= %= += -= <<= >>= &= ^= |=`
	`(T x) => y`	匿名函数（lambda 表达式）

1.5 语句

程序的操作是使用语句 (statement) 来表示的。C# 支持几种不同的语句，其中许多以嵌入语句的形式定义。

block 用于在只允许使用单个语句的上下文中编写多条语句。块由位于一对大括号 `{` 和 `}` 之间的语句列表组成。

声明语句 (declaration statement) 用于声明局部变量和常量。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

表达式语句 (expression statement) 用于对表达式求值。可用作语句的表达式包括方法调用、使用 new 运算符的对象分配、使用 = 和复合赋值运算符的赋值、使用 ++ 和 -- 运算符的增量和减量运算以及 await 表达式。

选择语句 (selection statement) 用于根据表达式的值从若干个给定的语句中选择一个来执行。这一组中有 if 和 switch 语句。

迭代语句 (iteration statement) 用于重复执行嵌入语句。这一组中有 while、do、for 和 foreach 语句。

跳转语句 (jump statement) 用于转移控制。这一组中有 break、continue、goto、throw、return 和 yield 语句。

try...catch 语句用于捕获在块的执行期间发生的异常，try...finally 语句用于指定终止代码，不管是否发生异常，该代码都始终要执行。

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

lock 语句用于获取某个给定对象的互斥锁，执行一个语句，然后释放该锁。

using 语句用于获得一个资源，执行一个语句，然后释放该资源。

下表列出了 C# 的各语句，并提供每个语句的示例。

语句	示例

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

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

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

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

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

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

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

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

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

return 语句	static int Add(int a, int b) {
	  return a + b;
	}
	static void Main() {
	  Console.WriteLine(Add(1, 2));
	  return;
	}

yield 语句	static IEnumerable<int> Range(int from, int to) {
	  for (int i = from; i < to; i++) {
	  yield return i;
	  }
	  yield break;
	}
	static void Main() {
	  foreach (int x in Range(-10,10)) {
	  Console.WriteLine(x);
	  }
	}

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

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

1.6 类和对象

类 (class) 是最基础的 C# 类型。类是一个数据结构，将状态（字段）和操作（方法和其他函数成员）组合在一个单元中。类为动态创建的类实例 (instance) 提供了定义，实例也称为对象 (object)。类支持继承 (inheritance) 和多态性 (polymorphism)，这是派生类 (derived class) 可用来扩展和专用化基类 (base class) 的机制。

使用类声明可以创建新的类。类声明以一个声明头开始，其组成方式如下：先指定类的特性和修饰符，然后是类的名称，接着是基类（如有）以及该类实现的接口。声明头后面跟着类体，它由一组位于一对大括号 { 和 } 之间的成员声明组成。

下面是一个名为 Point 的简单类的声明：

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

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

类的实例使用 new 运算符创建，该运算符为新的实例分配内存、调用构造函数初始化该实例，并返回对该实例的引用。下面的语句创建两个 Point 对象，并将对这两个对象的引用存储在两个变量中：

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

当不再使用对象时，该对象占用的内存将自动收回。在 C# 中，没有必要也不可能显式释放分配给对象的内存。

1.6.1 成员

类的成员或者是静态成员 (static member)，或者是实例成员 (instance member)。静态成员属于类，实例成员属于对象（类的实例）。

下表提供了类所能包含的成员种类的概述。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

成员	说明
常量	与类关联的常量值
字段	类的变量
方法	类可执行的计算和操作
属性	与读写类的命名属性相关联的操作
索引器	与以数组方式索引类的实例相关联的操作
事件	可由类生成的通知
运算符	类所支持的转换和表达式运算符
构造函数	初始化类的实例或类本身所需的操作
析构函数	在永久丢弃类的实例之前执行的操作
类型	类所声明的嵌套类型

1.6.2 可访问性

类的每个成员都有关联的可访问性，它控制能够访问该成员的程序文本区域。有五种可能的可访问性形式。下表概述了这些可访问性。

可访问性	含义
public	访问不受限制
protected	访问仅限于此类或从此类派生的类
internal	访问仅限于此程序
protected internal	访问仅限于此程序或从此类派生的类
private	访问仅限于此类

1.6.3 类型形参

类定义可以通过在类名后添加用尖括号括起来的类型参数名称列表来指定一组类型参数。类型参数可用于在类声明体中定义类的成员。在下面的示例中，Pair 的类型参数为 TFirst 和 TSecond：

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

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

当使用泛型类时，必须为每个类型参数提供类型实参：

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

提供了类型实参的泛型类型（例如上面的 Pair<int,string>）称为构造的类型。

1.6.4 基类

类声明可通过在类名和类型参数后面添加一个冒号和基类的名称来指定一个基类。省略基类的指定等同于从类型 object 派生。在下面的示例中，Point3D 的基类是 Point，而 Point 的基类是 object：

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

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

public class Point3D: Point
{
   public int z;

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

类继承其基类的成员。继承意味着一个类隐式地将它的基类的所有成员当作自己的成员，但基类的实例构造函数、静态构造函数和析构函数除外。派生类能够在继承基类的基础上添加新的成员，但是它不能移除继承成员的定义。在前面的示例中，Point3D 从 Point 继承了 x 和 y 字段，并且每个 Point3D 实例均包含三个字段：x、y 和 z。

从某个类类型到它的任何基类类型存在隐式的转换。因此，类类型的变量可以引用该类的实例或任何派生类的实例。例如，对于前面给定的类声明，Point 类型的变量既可以引用 Point 也可以引用 Point3D：

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

1.6.5 字段

字段是与类或类的实例关联的变量。

使用 static 修饰符声明的字段定义了一个静态字段(static field)。一个静态字段只标识一个存储位置。无论对一个类创建多少个实例，它的静态字段永远都只有一个副本。

不使用 static 修饰符声明的字段定义了一个实例字段 (instance field)。类的每个实例都为该类的所有实例字段包含一个单独副本。

在下面的示例中，Color 类的每个实例都有实例字段 r、g 和 b 的单独副本，但是 Black、White、Red、Green 和 Blue 静态字段只存在一个副本：

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

   private byte r, g, b;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

如上面的示例所示，可以使用 readonly 修饰符声明只读字段(read-only field)。给 readonly 字段的赋值只能作为字段声明的组成部分出现，或在同一个类中的构造函数中出现。

1.6.6 方法

方法 (method) 是一种成员，用于实现可由对象或类执行的计算或操作。静态方法 (static method) 通过类来访问。实例方法 (instance method) 通过类的实例来访问。

方法具有一个参数 (parameter) 列表（可以为空），表示传递给该方法的值或变量引用；方法还具有一个返回类型 (return type)，指定该方法计算和返回的值的类型。如果方法不返回值，则其返回类型为 void。

与类型一样，方法也可以有一组类型参数，当调用方法时必须为类型参数指定类型实参。与类型不同的是，类型实参经常可以从方法调用的实参推断出，而无需显式指定。

方法的签名 (signature) 在声明该方法的类中必须唯一。方法的签名由方法的名称、类型参数的数目以及该方法的参数的数目、修饰符和类型组成。方法的签名不包含返回类型。

1.6.6.1 参数

参数用于向方法传递值或变量引用。方法的参数从调用该方法时指定的实参 (argument) 获取它们的实际值。有四类参数：值参数、引用参数、输出参数和参数数组。

值参数 (value parameter) 用于传递输入参数。一个值参数相当于一个局部变量，只是它的初始值来自为该形参传递的实参。对值参数的修改不影响为该形参传递的实参。

值参数可以是可选的，通过指定默认值可以省略对应的实参。

引用参数 (reference parameter) 用于传递输入和输出参数。为引用参数传递的实参必须是变量，并且在方法执行期间，引用参数与实参变量表示同一存储位置。引用参数使用 ref 修饰符声明。下面的示例演示 ref 参数的用法。

```
using System;

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

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

输出参数 (output parameter) 用于传递输出参数。对于输出参数来说，调用方提供的实参的初始值并不重要。除此之外，输出参数与引用参数类似。输出参数是用 out 修饰符声明的。下面的示例演示 out 参数的用法。

```
using System;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

参数数组 (parameter array) 允许向方法传递可变数量的实参。参数数组使用 `params` 修饰符声明。只有方法的最后一个参数才可以是参数数组，并且参数数组的类型必须是一维数组类型。`System.Console` 类的 `Write` 和 `WriteLine` 方法就是参数数组用法的很好示例。它们的声明如下。

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

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

   ...
}
```

在使用参数数组的方法中，参数数组的行为完全就像常规的数组类型参数。但是，在具有参数数组的方法的调用中，既可以传递参数数组类型的单个实参，也可以传递参数数组的元素类型的任意数目的实参。在后一种情况下，将自动创建一个数组实例，并使用给定的实参对它进行初始化。示例：

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

等价于以下语句：

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

1.6.6.2 方法体和局部变量

方法体指定了在调用该方法时将执行的语句。

方法体可以声明仅用在该方法调用中的变量。这样的变量称为局部变量 (local variable)。局部变量声明指定了类型名称、变量名称，还可指定初始值。下面的示例声明一个初始值为零的局部变量 `i` 和一个没有初始值的变量 `j`。

```
using System;

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

C# 要求在对局部变量明确赋值 (definitely assigned) 之后才能获取其值。例如，如果前面对 i 的声明中未包括初始值，则编译器将针对随后对 i 的使用报错，因为 i 在程序中的这些位置还没有明确赋值。

方法可以使用 return 语句将控制返回到它的调用方。在返回 void 的方法中，return 语句不能指定表达式。在返回非 void 的方法中，return 语句必须含有一个计算返回值的表达式。

1.6.6.3 静态方法和实例方法

使用 static 修饰符声明的方法为静态方法 (static method)。静态方法不对特定实例进行操作，并且只能直接访问静态成员。

不使用 static 修饰符声明的方法为实例方法 (instance method)。实例方法对特定实例进行操作，并且能够访问静态成员和实例成员。在调用实例方法的实例上，可以通过 this 显式地访问该实例。而在静态方法中引用 this 是错误的。

下面的 Entity 类具有静态成员和实例成员。

```
class Entity
{
   static int nextSerialNo;

   int serialNo;

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

   public int GetSerialNo() {
      return serialNo;
   }

   public static int GetNextSerialNo() {
      return nextSerialNo;
   }

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

每个 Entity 实例都包含一个序号（我们假定这里省略了一些其他信息）。Entity 构造函数（类似于实例方法）使用下一个可用的序号来初始化新的实例。由于该构造函数是一个实例成员，它既可以访问 serialNo 实例字段，也可以访问 nextSerialNo 静态字段。

GetNextSerialNo 和 SetNextSerialNo 静态方法可以访问 nextSerialNo 静态字段，但是如果直接访问 serialNo 实例字段就会产生错误。

下面的示例演示 Entity 类的使用。

```
using System;

class Test
{
   static void Main() {
      Entity.SetNextSerialNo(1000);

      Entity e1 = new Entity();
      Entity e2 = new Entity();

      Console.WriteLine(e1.GetSerialNo());            // Outputs "1000"
      Console.WriteLine(e2.GetSerialNo());            // Outputs "1001"
      Console.WriteLine(Entity.GetNextSerialNo());    // Outputs "1002"
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

注意：SetNextSerialNo 和 GetNextSerialNo 静态方法是在类上调用的，而 GetSerialNo 实例方法是在该类的实例上调用的。

1.6.6.4 虚方法、重写方法和抽象方法

若一个实例方法的声明中含有 virtual 修饰符，则称该方法为虚方法。若其中没有 virtual 修饰符，则称该方法为非虚方法。

在调用一个虚方法时，该调用所涉及的实例的运行时类型 (runtime type) 确定了要实际调用的方法实现。在非虚方法调用中，实例的编译时类型 (compile-time type) 负责做出此决定。

虚方法可以在派生类中重写 (override)。当某个实例方法声明包括 override 修饰符时，该方法将重写所继承的具有相同签名的虚方法。虚方法声明用于引入新方法，而重写方法声明则用于使现有的继承虚方法专用化（通过提供该方法的新实现）。

抽象 (abstract) 方法是没有实现的虚方法。抽象方法使用 abstract 修饰符进行声明，并且只允许出现在同样被声明为 abstract 的类中。抽象方法必须在每个非抽象派生类中重写。

下面的示例声明一个抽象类 Expression，它表示一个表达式目录树节点；它有三个派生类 Constant、VariableReference 和 Operation，它们分别实现了常量、变量引用和算术运算的表达式目录树节点。（这与第 4.6 节中介绍的表达式树类型相似，但不要混淆）。

```
using System;
using System.Collections;

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

public class Constant: Expression
{
   double value;

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

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

public class VariableReference: Expression
{
   string name;

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

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

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

上面的四个类可用于为算术表达式建模。例如，使用这些类的实例，表达式 x + 3 可如下表示。

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

代码中调用了 Expression 实例的 Evaluate 方法，以计算给定表达式的值，从而生成一个 double 值。该方法接受一个包含变量名称（作为哈希表项的键）和值（作为项的值）的 Hashtable 作为实参。Evaluate 方法是一个虚抽象方法，意味着非抽象派生类必须重写该方法以提供实际的实现。

Constant 的 Evaluate 实现只是返回所存储的常量。VariableReference 的实现在哈希表中查找变量名称，并返回产生的值。Operation 的实现先对左操作数和右操作数求值（通过递归调用它们的 Evaluate 方法），然后执行给定的算术运算。

下面的程序使用 Expression 类，对于不同的 x 和 y 值，计算表达式 x * (y + 2) 的值。

```
using System;
using System.Collections;

class Test
{
   static void Main() {

      Expression e = new Operation(
         new VariableReference("x"),
         '*',
         new Operation(
            new VariableReference("y"),
            '+',
            new Constant(2)
         )
      );

      Hashtable vars = new Hashtable();

      vars["x"] = 3;
      vars["y"] = 5;
      Console.WriteLine(e.Evaluate(vars));      // Outputs "21"
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
            vars["x"] = 1.5;
            vars["y"] = 9;
            Console.WriteLine(e.Evaluate(vars));      // Outputs "16.5"
        }
    }
```

1.6.6.5 方法重载

方法重载 (overloading) 允许同一类中的多个方法具有相同名称，条件是这些方法具有唯一的签名。在编译一个重载方法的调用时，编译器使用重载决策 (overload resolution) 确定要调用的特定方法。重载决策将查找与参数最佳匹配的方法，如果没有找到任何最佳匹配的方法则报告错误信息。下面的示例演示重载决策的工作机制。`Main` 方法中的每个调用的注释表明实际调用的方法。

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

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

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

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

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

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

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

正如该示例所示，总是通过显式地将实参强制转换为确切的参数类型和/或显式地提供类型实参，来选择一个特定的方法。

1.6.7 其他函数成员

包含可执行代码的成员统称为类的函数成员 (function member)。前一节描述的方法是函数成员的主要类型。本节介绍了 C# 支持的其他类型的函数成员：构造函数、属性、索引器、事件、运算符和析构函数。

下表演示一个名为 `List<T>` 的泛型类，它实现一个可增长的对象列表。该类包含了几种最常见的函数成员的示例。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
public class List<T>
{
```

`const int defaultCapacity = 4;`	常量
`T[] items;` `int count;`	字段

`public List(int capacity = defaultCapacity) {`
`  items = new T[capacity];`
`}`
	构造函数

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

`public T this[int index] {`
`  get {`
`  return items[index];`
`  }`
`  set {`
`  items[index] = value;`
`  OnChanged();`
`  }`
`}`
	索引器

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

`public event EventHandler Changed;`	事件

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

`}`	

1.6.7.1 构造函数

C# 支持两种构造函数：实例构造函数和静态构造函数。实例构造函数 (instance constructor) 是实现初始化类实例所需操作的成员。静态构造函数 (static constructor) 是一种用于在第一次加载类本身时实现其初始化所需操作的成员。

构造函数的声明如同方法一样，不过它没有返回类型，并且它的名称与其所属的类的名称相同。如果构造函数声明包含 `static` 修饰符，则它声明了一个静态构造函数。否则，它声明的是一个实例构造函数。

实例构造函数可以被重载。例如，`List<T>` 类声明了两个实例构造函数，一个无参数，另一个接受一个 `int` 参数。实例构造函数使用 `new` 运算符进行调用。下面的语句分别使用 `List<string>` 类的每个构造函数分配两个 `List` 实例。

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

实例构造函数不同于其他成员，它是不能被继承的。一个类除了其中实际声明的实例构造函数外，没有其他的实例构造函数。如果没有为某个类提供任何实例构造函数，则将自动提供一个不带参数的空的实例构造函数。

1.6.7.2 属性

属性 (property) 是字段的自然扩展。属性和字段都是命名的成员，都具有相关的类型，且用于访问字段和属性的语法也相同。然而，与字段不同，属性不表示存储位置。相反，属性有访问器 (accessor)，这些访问器指定在它们的值被读取或写入时需执行的语句。

属性的声明与字段类似，不同的是属性声明以位于定界符 { 和 } 之间的一个 get 访问器和/或一个 set 访问器结束，而不是以分号结束。同时具有 get 访问器和 set 访问器的属性是读写属性 (read-write property)，只有 get 访问器的属性是只读属性 (read-only property)，只有 set 访问器的属性是只写属性 (write-only property)。

get 访问器相当于一个具有属性类型返回值的无形参方法。除了作为赋值的目标，当在表达式中引用属性时，将调用该属性的 get 访问器以计算该属性的值。

set 访问器相当于具有一个名为 value 的参数并且没有返回类型的方法。当某个属性作为赋值的目标被引用，或者作为 ++ 或 -- 的操作数被引用时，将调用 set 访问器，并传入提供新值的实参。

List<T> 类声明了两个属性 Count 和 Capacity，它们分别是只读属性和读写属性。下面是这些属性的使用示例。

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

与字段和方法相似，C# 同时支持实例属性和静态属性。静态属性使用 static 修饰符声明，而实例属性的声明不带该修饰符。

属性的访问器可以是虚的。当属性声明包括 virtual、abstract 或 override 修饰符时，修饰符应用于该属性的访问器。

1.6.7.3 索引器

索引器 (indexer) 是这样一个成员：它支持按照索引数组的方法来索引对象。索引器的声明与属性类似，不同的是该成员的名称是 this，后跟一个位于定界符 [和] 之间的参数列表。在索引器的访问器中可以使用这些参数。与属性类似，索引器可以是读写、只读和只写的，并且索引器的访问器可以是虚的。

该 List 类声明了单个读写索引器，该索引器接受一个 int 参数。该索引器使得通过 int 值对 List 实例进行索引成为可能。例如

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

索引器可以被重载，这意味着一个类可以声明多个索引器，只要其参数的数量和类型不同即可。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

1.6.7.4 事件

事件 (event) 是一种使类或对象能够提供通知的成员。事件的声明与字段类似，不同的是事件的声明包含 `event` 关键字，并且类型必须是委托类型。

在声明事件成员的类中，事件的行为就像委托类型的字段（前提是该事件不是抽象的并且未声明访问器）。该字段存储对一个委托的引用，该委托表示已添加到该事件的事件处理程序。如果尚未添加事件处理程序，则该字段为 `null`。

`List<T>` 类声明了一个名为 `Changed` 的事件成员，它指示已将一个新项添加到列表中。`Changed` 事件由 `OnChanged` 虚方法引发，后者先检查该事件是否为 `null`（表明没有处理程序）。“引发一个事件”与“调用一个由该事件表示的委托”这两个概念完全等效，因此没有用于引发事件的特殊语言构造。

客户端通过事件处理程序 (event handler) 来响应事件。事件处理程序使用 `+=` 运算符附加，使用 `-=` 运算符移除。下面的示例向 `List<string>` 类的 `Changed` 事件附加一个事件处理程序。

```
using System;

class Test
{
   static int changeCount;

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

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

对于要求控制事件的底层存储的高级情形，事件声明可以显式提供 `add` 和 `remove` 访问器，它们在某种程度上类似于属性的 `set` 访问器。

1.6.7.5 运算符

运算符 (operator) 是一种类成员，它定义了可应用于类实例的特定表达式运算符的含义。可以定义三类运算符：一元运算符、二元运算符和转换运算符。所有运算符都必须声明为 `public` 和 `static`。

`List<T>` 类声明了两个运算符 `operator ==` 和 `operator !=`，从而为将那些运算符应用于 `List` 实例的表达式赋予了新的含义。具体而言，上述运算符将两个 `List<T>` 实例的相等关系定义为逐一比较其中所包含的对象（使用所包含对象的 `Equals` 方法）。下面的示例使用 `==` 运算符比较两个 `List<int>` 实例。

```
using System;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

第一个 Console.WriteLine 输出 True，原因是两个列表包含的对象数目、对象顺序和对象值都相同。如果 List<T> 未定义 operator ==，则第一个 Console.WriteLine 将输出 False，原因是 a 和 b 引用的是不同的 List<int> 实例。

1.6.7.6 析构函数

析构函数 (destructor) 是一种用于实现销毁类实例所需操作的成员。析构函数不能带参数，不能具有可访问性修饰符，也不能被显式调用。垃圾回收期间会自动调用所涉及实例的析构函数。

垃圾回收器在决定何时回收对象和运行析构函数方面允许有广泛的自由度。具体而言，析构函数调用的时机并不是确定的，析构函数可以在任何线程上执行。由于这些以及其他原因，仅当没有其他可行的解决方案时，才应在类中实现析构函数。

using 语句提供了更好的对象析构方法。

1.7 结构

像类一样，结构 (struct) 是能够包含数据成员和函数成员的数据结构。但是与类不同，结构是值类型，不需要堆分配。结构类型的变量直接存储该结构的数据，而类类型的变量则存储对动态分配的对象的引用。结构类型不支持用户指定的继承，并且所有结构类型都隐式地从类型 object 继承。

结构对于具有值语义的小型数据结构尤为有用。复数、坐标系中的点或字典中的“键-值”对都是结构的典型示例。对小型数据结构而言，使用结构而不使用类会大大节省需要为应用程序分配的内存数量。例如，下面的程序创建并初始化一个含有 100 个点的数组。对于作为类实现的 Point，实例化了 101 个单独对象，其中，数组需要一个，其 100 个元素每个都需要一个。

```
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);
   }
}
```

一种替代办法是将 Point 定义为结构。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

现在，只有一个对象被实例化（即用于数组的那个对象），而 Point 实例内联存储在数组中。

结构构造函数也是使用 new 运算符调用，但是这并不意味着会分配内存。结构构造函数并不动态分配对象并返回对它的引用，而是直接返回结构值本身（通常是堆栈上的一个临时位置），然后根据需要复制该结构值。

对于类，两个变量可能引用同一对象，因此对一个变量进行的操作可能影响另一个变量所引用的对象。对于结构，每个变量都有自己的数据副本，对一个变量的操作不会影响另一个变量。例如，下面的代码段产生的输出取决于 Point 是类还是结构。

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

如果 Point 是类，输出将是 20，因为 a 和 b 引用同一对象。如果 Point 是结构，输出将是 10，因为 a 对 b 的赋值创建了该值的一个副本，因此接下来对 a.x 的赋值不会影响 b 这一副本。

前一示例突出了结构的两个限制。首先，复制整个结构通常不如复制对象引用的效率高，因此结构的赋值和值参数传递可能比引用类型的开销更大。其次，除了 ref 和 out 参数，不可能创建对结构的引用，这样限制了结构的应用范围。

1.8 数组

数组 (array) 是一种包含若干变量的数据结构，这些变量都可以通过计算索引进行访问。数组中包含的变量（又称数组的元素）具有相同的类型，该类型称为数组的元素类型。

数组类型为引用类型，因此数组变量的声明只是为数组实例的引用留出空间。实际的数组实例在运行时使用 new 运算符动态创建。new 运算符指定新数组实例的长度 (length)，它在该实例的生存期内是固定不变的。数组元素的索引范围从 0 到 Length - 1。new 运算符自动将数组的元素初始化为它们的默认值，例如将所有数值类型初始化为零，将所有引用类型初始化为 null。

下面的示例创建一个 int 元素的数组，初始化该数组，并打印该数组的内容。

```
using System;

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

此示例创建并操作一个一维数组 (single-dimensional array)。C# 还支持多维数组 (multi-dimensional array)。数组类型的维数也称为数组类型的秩 (rank)，它是数组类型的方括号之间的逗号个数加 1。下面的示例分别分配一个一维数组、一个二维数组和一个三维数组。

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

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

数组的元素类型可以是任意类型，包括数组类型。对于数组元素的类型为数组的情况，我们有时称之为交错数组 (jagged array)，原因是元素数组的长度不必全都相同。下面的示例分配一个由 int 数组组成的数组：

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

第一行创建一个具有三个元素的数组，每个元素的类型为 int[] 并具有初始值 null。接下来的代码行使用对不同长度的数组实例的引用分别初始化这三个元素。

new 运算符允许使用数组初始值设定项 (array initializer) 指定数组元素的初始值，数组初始值设定项是在一个位于定界符 { 和 } 之间的表达式列表。下面的示例分配并初始化具有三个元素的 int[]。

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

注意数组的长度是从 { 和 } 之间的表达式个数推断出来的。对于局部变量和字段声明，可以进一步简写，从而不必再次声明数组类型。

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

前面的两个示例都等效于下面的示例：

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

1.9 接口

接口 (interface) 定义了一个可由类和结构实现的协定。接口可以包含方法、属性、事件和索引器。接口不提供它所定义的成员的实现 — 它仅指定实现该接口的类或结构必须提供的成员。

接口可支持多重继承。在下面的示例中，接口 IComboBox 同时从 ITextBox 和 IListBox 继承。

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

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

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

interface IComboBox: ITextBox, IListBox {}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

类和结构可以实现多个接口。在下面的示例中，类 EditBox 实现了 IControl 和 IDataBound。

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

当类或结构实现某个特定接口时，该类或结构的实例可以隐式地转换为该接口类型。例如

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

在无法静态知道某个实例是否实现某个特定接口的情况下，可以使用动态类型强制转换。例如，下面的语句使用动态类型强制转换获取对象的 IControl 和 IDataBound 接口实现。由于该对象的实际类型为 EditBox，此强制转换成功。

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

在前面的 EditBox 类中，来自 IControl 接口的 Paint 方法和来自 IDataBound 接口的 Bind 方法是使用 public 成员实现的。C# 还支持显式接口成员实现，类或结构可以使用它来避免将成员声明为 public。显式接口成员实现使用完全限定的接口成员名。例如，EditBox 类可以使用显式接口成员实现来实现 IControl.Paint 和 IDataBound.Bind 方法，如下所示。

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

显式接口成员只能通过接口类型来访问。例如，要调用上面 EditBox 类提供的 IControl.Paint 实现，必须首先将 EditBox 引用转换为 IControl 接口类型。

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

1.10 枚举

枚举类型 (enum type) 是具有一组命名常量的独特的值类型。下面的示例声明并使用一个名为 Color 的枚举类型，该枚举具有三个常量值 Red、Green 和 Blue。

```
using System;
enum Color
{
   Red,
   Green,
   Blue
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

每个枚举类型都有一个相应的整型类型，称为该枚举类型的基础类型(underlying type)。没有显式声明基础类型的枚举类型所对应的基础类型是 `int`。枚举类型的存储格式和取值范围由其基础类型确定。一个枚举类型的值域不受它的枚举成员限制。具体而言，一个枚举的基础类型的任何一个值都可以被强制转换为该枚举类型，成为该枚举类型的一个独特的有效值。

下面的示例声明了一个名为 `Alignment`、基础类型为 `sbyte` 的枚举类型。

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

如前面的示例所示，枚举成员的声明中包含常量表达式，用于指定该成员的值。每个枚举成员的常数值必须在该枚举的基础类型的范围之内。如果枚举成员声明未显式指定一个值，该成员将被赋予值零（如果它是该枚举类型中的第一个值）或前一个枚举成员（按照文本顺序）的值加 1。

可以使用类型强制转换将枚举值转换为整型值，反之亦然。例如

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

任何枚举类型的默认值都是转换为该枚举类型的整型值零。在变量被自动初始化为默认值的情况下，该默认值就是赋予枚举类型的变量的值。为了便于获得枚举类型的默认值，文本 `0` 隐式地转换为任何枚举类型。因此，下面的语句是允许的。

```
Color c = 0;
```

1.11 委托

委托类型 (delegate type) 表示对具有特定参数列表和返回类型的方法的引用。通过委托，我们能够将方法作为实体赋值给变量和作为参数传递。委托类似于在其他某些语言中的函数指针的概念，但是与函数指针不同，委托是面向对象的，并且是类型安全的。

下面的示例声明并使用一个名为 `Function` 的委托类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
using System;
delegate double Function(double x);
class Multiplier
{
   double factor;
   public Multiplier(double factor) {
      this.factor = factor;
   }
   public double Multiply(double x) {
      return x * factor;
   }
}
class Test
{
   static double Square(double x) {
      return x * x;
   }
   static double[] Apply(double[] a, Function f) {
      double[] result = new double[a.Length];
      for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
      return result;
   }
   static void Main() {
      double[] a = {0.0, 0.5, 1.0};
      double[] squares = Apply(a, Square);
      double[] sines = Apply(a, Math.Sin);
      Multiplier m = new Multiplier(2.0);
      double[] doubles =  Apply(a, m.Multiply);
   }
}
```

Function 委托类型的实例可以引用任何接受 double 实参并返回 double 值的方法。Apply 方法将给定的 Function 应用于 double[] 的元素，并返回含有结果的 double[]。在 Main 方法中，Apply 用于将三个不同的函数应用于一个 double[]。

委托既可以引用静态方法（例如前一示例中的 Square 或 Math.Sin），也可以引用实例方法（例如前一示例中的 m.Multiply）。引用了实例方法的委托也就引用了一个特定的对象，当通过该委托调用这个实例方法时，该对象在调用中成为 this。

也可以使用匿名函数创建委托，这是即时创建的“内联方法”。匿名函数可以查看外层方法的局部变量。因此，可以在不使用 Multiplier 类的情况下更容易地写出上面的乘法器示例：

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

委托的一个有趣且有用的属性在于，它不知道也不关心它所引用的方法的类；它仅关心所引用的方法是否与委托具有相同的参数和返回类型。

1.12 特性

C# 程序中的类型、成员和其他实体都支持修饰符，这些修饰符控制它们的行为的某些方面。例如，方法的可访问性是使用 public、protected、internal 和 private 修饰符来控制的。C# 使此功能一般化，以便能够将用户定义类型的声明信息附加到程序实体，并在运行时检索。这种附加的声明信息是程序通过定义和使用特性 (attribute) 来指定的。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

下面的示例声明一个 HelpAttribute 特性，该特性可放置在程序实体上，以便提供指向其关联文档的链接。

```
using System;

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

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

   public string Url {
      get { return url; }
   }

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

所有特性类都从 .NET Framework 提供的 System.Attribute 基类派生而来。可以通过在相关声明之前紧邻的方括号内提供特性名和任何实参来应用特性。如果特性的名称以 Attribute 结尾，在引用该特性时可以省略此名称后缀。例如，HelpAttribute 特性可以按如下方式使用。

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

此示例将一个 HelpAttribute 附加到 Widget 类，并且将另一个 HelpAttribute 附加到该类中的 Display 方法。特性类的公共构造函数控制在将特性附加到程序实体时，必须提供的信息。可以通过引用特性类的公共读写属性提供附加信息，例如前面对 Topic 属性的引用。

下面的示例演示如何使用反射在运行时检索给定程序实体的特性信息。

```
using System;
using System.Reflection;

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

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

当通过反射请求特定特性时，将使用程序源中提供的信息调用特性类的构造函数，并返回生成的特性实例。如果通过属性提供了附加信息，那些属性将在返回特性实例之前被设置为给定的值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

2. 词法结构

2.1 程序

C# 程序 (program) 由一个或多个源文件 (source file) 组成，源文件的正式名称是编译单元 (compilation unit)（第 9.1 节）。源文件是有序的 Unicode 字符序列。源文件与文件系统中的文件通常具有一对一的对应关系，但这种对应关系不是必需的。为实现可移植性的最大化，建议这些文件在文件系统中应按 UTF-8 编码规范编码。

从概念上讲，程序的编译分三个步骤：

1. 转换，这一步将用特定字符指令系统和编码方案编写的文件转换为 Unicode 字符序列。
2. 词法分析，这一步将 Unicode 输入字符流转换为标记流。
3. 句法分析，这一步将标记流转换为可执行代码。

2.2 文法

本规范采用两种文法 (grammar) 来表示 C# 编程语言的语法 (syntax)。词法文法 (lexical grammar)（第 2.2.2 节）规定怎样将 Unicode 字符组合成行结束符、空白、注释、标记和预处理指令等。句法文法 (syntactic grammar)（第 2.2.3 节）规定如何将那些由词法文法产生的标记组合成 C# 程序。

2.2.1 文法表示法

词法文法和句法文法用文法产生式 (grammar production) 来表示。每个文法产生式定义一个非结束符号和它可能的扩展（由非结束符或结束符组成的序列）。在文法产生式中，*non-terminal* 符号显示为斜体，而 `terminal` 符号显示为等宽字体。

文法产生式的第一行是该产生式所定义的非结束符号的名称，后跟一个冒号。每个后续的缩进行列出一个可能的扩展，它是以非结束符或结束符组成的序列的形式给出的。例如，产生式：

while-statement:
    `while` `(` *boolean-expression* `)` *embedded-statement*

定义了一个 *while-statement*，它是这样构成的：由标记 `while` 开始，后跟标记“`(`”、*boolean-expression*、标记“`)`”，最后是一个 *embedded-statement*。

当有不止一个可能的非结束符号扩展时，列出这些可能的扩展（每个扩展单独占一行）。例如，产生式：

statement-list:
    statement
    statement-list *statement*

定义一个 *statement-list*，它或仅含有一个 *statement*，或由一个 *statement-list* 和随后跟着的一个 *statement* 组成。换言之，定义是递归的，语句列表由一个或多个语句组成。

一个符号若以下标“$_{opt}$”作其后缀，就表明该符号是可选的。产生式：

block:
    `{` *statement-list*$_{opt}$ `}`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

是以下产生式的简短形式：

block:
    { }
    { *statement-list* }

它定义了一个 *block*，此块由一个用“{”和“}”标记括起来的可选 *statement-list* 组成。

可选项通常在单独的行上列出，但是当有许多可选项时，可以在单行上给定的扩展列表之后加上短语“之一”。这只是在单独一行上列出每个可选项的简短形式。例如，产生式：

real-type-suffix: one of
    F f D d M m

是以下产生式的简短形式：

real-type-suffix:
    F
    f
    D
    d
    M
    m

2.2.2 词法文法

C# 的词法文法在第 2.3、2.4 和 2.5 节中介绍。词法文法的结束符号为 Unicode 字符集的字符，并且词法文法指定如何组合字符以构成标记（第 2.4 节）、空白（第 2.3.3 节）、注释（第 2.3.2 节）和预处理指令（第 2.5 节）。

C# 程序中的每个源文件都必须符合词法文法的 *input* 产生式（第 2.3 节）。

2.2.3 句法文法

本章后面的章节和附录介绍 C# 的句法文法。句法文法的结束符号是由词法文法定义的标记，句法文法指定如何组合这些标记以构成 C# 程序。

C# 程序中的每个源文件都必须符合句法文法的 *compilation-unit* 产生式（第 9.1 节）。

2.3 语法分析

input 产生式定义 C# 源文件的词法结构。C# 程序中的每个源文件都必须符合此词法文法产生式。

input:
    input-section$_{opt}$

input-section:
    input-section-part
    input-section input-section-part

input-section-part:
    input-elements$_{opt}$ *new-line*
    pp-directive

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

input-element:
whitespace
comment
token

C# 源文件的词法结构由五个基本元素组成：行结束符（第 2.3.1 节）、空白（第 2.3.3 节）、注释（第 2.3.2 节）、标记（第 2.4 节）和预处理指令（第 2.5 节）。在这些基本元素中，只有标记在 C# 程序的句法文法（第 2.2.3 节）中具有重要意义。

对 C# 源文件的词法处理就是将文件缩减成标记序列，该序列然后即成为句法分析的输入。行结束符、空白和注释可用于分隔标记，预处理指令可导致跳过源文件中的某些节，除此之外这些词法元素对 C# 程序的句法结构没有任何影响。

当有若干词法文法产生式与源文件中的一个字符序列匹配时，词法处理总是构成尽可能最长的词法元素。例如，字符序列 // 按单行注释的开头处理，这是因为该词法元素比一个 / 标记要长。

2.3.1 行结束符

行结束符将 C# 源文件的字符划分为行。

new-line:
Carriage return character (U+000D)
Line feed character (U+000A)
Carriage return character (U+000D) followed by line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

为了与添加文件尾标记的源代码编辑工具兼容，并能够以正确结束的行序列的形式查看源文件，下列转换按顺序应用到 C# 程序中的每个源文件：

- 如果源文件的最后一个字符为 Control-Z 字符 (U+001A)，则删除此字符。
- 如果源文件非空并且源文件的最后一个字符不是回车符 (U+000D)、换行符 (U+000A)、行分隔符 (U+2028) 或段落分隔符 (U+2029)，则将在源文件的结尾添加一个回车符 (U+000D)。

2.3.2 注释

支持两种形式的注释：单行注释和带分隔符的注释。单行注释 (Single-line comment) 以字符 // 开头并延续到源行的结尾。带分隔符的注释 (Delimited comment) 以字符 /* 开头，以字符 */ 结束。带分隔符的注释可以跨多行。

comment:
single-line-comment
delimited-comment

single-line-comment:
// *input-characters*$_{opt}$

input-characters:
input-character
input-characters input-character

input-character:
Any Unicode character except a *new-line-character*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

new-line-character:
Carriage return character (U+000D)
Line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

delimited-comment:
/* *delimited-comment-text*$_{opt}$ *asterisks* /

delimited-comment-text:
delimited-comment-section
delimited-comment-text *delimited-comment-section*

delimited-comment-section:
/
asterisks$_{opt}$ *not-slash-or-asterisk*

asterisks:
*
asterisks *

not-slash-or-asterisk:
Any Unicode character except / or *

注释不嵌套。字符序列 /* 和 */ 在 // 注释中没有任何特殊含义，字符序列 // 和 /* 在带分隔符的注释中没有任何特殊含义。

在字符和字符串内不处理注释。

下面的示例

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

包含一个带分隔符的注释。

下面的示例

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

演示了若干单行注释。

2.3.3 空白

空白被定义为任何含 Unicode 类 Zs 的字符（包括空白字符）以及水平制表符、垂直制表符和换页符。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

whitespace:
Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

2.4 标记

有几类标记：标识符、关键字、文本、运算符和标点符号。空白和注释不是标记，但它们可充当标记的分隔符。

token:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1 Unicode 字符转义序列

Unicode 字符转义序列表示一个 Unicode 字符。Unicode 字符转义序列在标识符（第 2.4.2 节）、字符（第 2.4.4.4 节）和规则字符串（第 2.4.4.5 节）中处理。不在其他任何位置处理 Unicode 字符转义（例如，在构成运算符、标点符号或关键字时）。

unicode-escape-sequence:
\u *hex-digit hex-digit hex-digit hex-digit*
\U *hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit*

Unicode 转义序列表示由“\u”或“\U”字符后面的十六进制数字构成的单个 Unicode 字符。由于 C# 在字符和字符串值中使用 Unicode 代码点的 16 位编码，因此从 U+10000 到 U+10FFFF 的 Unicode 字符不能在字符中使用，在字符串中则用一个 Unicode 代理项对来表示。不支持代码数据点在 0x10FFFF 以上的 Unicode 字符。

不执行多次转换。例如，字符串文本“\u005Cu005C”等同于“\u005C”，而不是“\”。Unicode 值 \u005C 是字符“\”。

下面的示例

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

演示了 \u0066（它是字母“f”的转义序列）的一些用法。该程序等效于

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

2.4.2 标识符

本节给出的标识符规则完全符合Unicode 标准附件 31 推荐的规则，但以下情况除外：允许将下划线用作初始字符（这是 C 编程语言的传统），允许在标识符中使用 Unicode 转义序列，以及允许“@”字符作为前缀以使关键字能够用作标识符。

identifier:
    available-identifier
    `@` *identifier-or-keyword*

available-identifier:
    An *identifier-or-keyword* that is not a *keyword*

identifier-or-keyword:
    identifier-start-character *identifier-part-characters*$_{opt}$

identifier-start-character:
    letter-character
    _ (the underscore character `U+005F`)

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

identifier-part-character:
    letter-character
    decimal-digit-character
    connecting-character
    combining-character
    formatting-character

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

combining-character:
    A Unicode character of classes Mn or Mc
    A *unicode-escape-sequence* representing a character of classes Mn or Mc

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

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

formatting-character:
    A Unicode character of the class Cf
    A *unicode-escape-sequence* representing a character of the class Cf

有关上面提到的 Unicode 字符类的信息，请参见《Unicode 标准 3.0 版》的第 4.5 节。

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

符合规范的程序中的标识符必须遵循由“Unicode 标准化格式 C”（按“Unicode 标准附录 15”中的定义）定义的规范格式。当遇到非“标准化格式 C”格式的标识符时，怎样处理它可由 C 的具体实现确定，但是不要求诊断。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

使用前缀“@”可以将关键字用作标识符，这在与其他编程语言建立接口时很有用。字符 @ 并不是标识符的实际组成部分，因此在其他语言中可能将此标识符视为不带前缀的正常标识符。带 @ 前缀的标识符称作逐字标识符 (verbatim identifier)。允许将 @ 前缀用于非关键字的标识符，但是（从代码书写样式的意义上）强烈建议不要这样做。

示例：

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

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

定义一个名为“class”的类，该类具有一个名为“static”的静态方法，此方法带一个名为“bool”的参数。请注意，由于在关键字中不允许使用Unicode 转义符，因此标记“cl\u0061ss”是标识符，与“@class”标识符相同。

两个标识符如果在按顺序实施了下列转换后相同，则被视为相同：

- 如果使用了前缀“@”，移除它。
- 将每个 *unicode-escape-sequence* 转换为它的对应 Unicode 字符。
- 移除所有 *formatting-character*。

包含两个连续下划线字符 (U+005F) 的标识符被保留供具体实现使用。例如，一个实现可以设置它自己的以两个下划线开头的扩展关键字。

2.4.3 关键字

关键字 (keyword) 是类似标识符的保留的字符序列，不能用作标识符（以 @ 字符开头时除外）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

keyword: one of

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

在文法中的某些位置，特定的标识符有特殊的含义，但不是关键字。这类标识符有时称为“上下文关键字”。例如，在属性声明中，“`get`”和“`set`”标识符有特殊的含义（第 10.7.2 节）。在这些位置从不允许使用 `get` 或 `set` 之外的标识符，因此此用法与这些词用作标识符并不冲突。在其他情况下，如对于隐式类型化局部变量声明（第 8.5.1 节）中的标识符“`var`”，上下文关键字可能与声明的名称冲突。在这类情况下，声明的名称优先于将标识符用作上下文关键字。

2.4.4 文本

文本 (literal) 是一个值的源代码表示形式。

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

2.4.4.1 布尔值

有两个布尔文本值：`true` 和 `false`。

boolean-literal:
`true`
`false`

boolean-literal 的类型是 `bool`。

2.4.4.2 整数

整数文本用于编写类型为 `int`、`uint`、`long` 和 `ulong` 的值。整数有两种可能的形式：十进制和十六进制。

integer-literal:
decimal-integer-literal
hexadecimal-integer-literal

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

decimal-integer-literal:
decimal-digits *integer-type-suffix*$_{opt}$

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*$_{opt}$
0X *hex-digits* *integer-type-suffix*$_{opt}$

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

一个整数的类型按下面的方式确定：

- 如果该整数没有后缀，则它属于以下所列的类型中第一个能够表示其值的那个类型：int、uint、long、ulong。
- 如果该文本带有后缀 U 或 u，则它属于以下所列的类型中第一个能够表示其值的那个类型：uint 和 ulong。
- 如果该文本带有后缀 L 或 l，则它属于以下所列的类型中第一个能够表示其值的那个类型：long 和 ulong。
- 如果该文本带有后缀 UL、Ul、uL、ul、LU、Lu、lU 或 lu，它的类型为 ulong。

如果整数表示的值超出了 ulong 类型的范围，则将发生编译时错误。

从书写风格（样式）的角度出发，建议在书写类型 long 的文本时使用“L”而不是“l”，因为字母“l”容易与数字“1””混淆。

为允许尽可能小的 int 和 long 值写为十进制整数，有下面两个规则：

- 当具有值 2147483648 (2^{31}) 且没有 *integer-type-suffix* 的一个 *decimal-integer-literal* 作为标记紧接在一元负运算符标记（第 7.7.2 节）后出现时，结果为具有值 −2147483648 (-2^{31}) 的 int 类型常量。在所有其他情况下，这样的 *decimal-integer-literal* 属于 uint 类型。
- 当具有值 9223372036854775808 (2^{63}) 的一个 *decimal-integer-literal*（没带 *integer-type-suffix*，或带有 *integer-type-suffix* L 或 l）作为一个标记紧跟在一个一元负运算符标记（第 7.7.2 节）后出现时，结果是具有值 −9223372036854775808 (-2^{63}) 的 long 类型的常量。在所有其他情况下，这样的 *decimal-integer-literal* 属于 ulong 类型。

2.4.4.3 实数

实数文本用于编写类型为 float、double 和 decimal 的值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

real-literal:
decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
`.` *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
decimal-digits *exponent-part* *real-type-suffix*$_{opt}$
decimal-digits *real-type-suffix*

exponent-part:
`e` *sign*$_{opt}$ *decimal-digits*
`E` *sign*$_{opt}$ *decimal-digits*

sign: one of
`+` `-`

real-type-suffix: one of
`F` `f` `D` `d` `M` `m`

如果未指定 *real-type-suffix*，则实数的类型为 `double`。否则，实数类型后缀确定实数的类型，如下所示：

- 以 `F` 或 `f` 为后缀的实数文本的类型为 `float`。例如，文本 `1f`、`1.5f`、`1e10f` 和 `123.456F` 的类型都是 `float`。
- 以 `D` 或 `d` 为后缀的实数文本的类型为 `double`。例如，文本 `1d`、`1.5d`、`1e10d` 和 `123.456D` 的类型都是 `double`。
- 以 `M` 或 `m` 为后缀的实数文本的类型为 `decimal`。例如，文本 `1m`、`1.5m`、`1e10m` 和 `123.456M` 的类型都是 `decimal`。此实数通过取精确值转换为 `decimal` 值，如果有必要，用银行家舍入法（第 4.1.7 节）舍入为最接近的可表示值。保留该实数的所有小数位数，除非值被舍入或者值为零（在后一种情况中，符号和小数位数为 0）。因此，实数 `2.900m` 经分析后将形成这样的一个小数：符号为 `0`、系数为 `2900`，小数位数为 `3`。

如果一个给定的实数不能用指定的类型表示，则会发生编译时错误。

使用 IEEE “就近舍入”模式确定类型 `float` 或 `double` 的实数的值。

注意在实数中，小数点后必须始终是十进制数字。例如，`1.3F` 是实数，但 `1.F` 不是。

2.4.4.4 字符

字符表示单个字符，通常由置于引号中的一个字符组成，如 `'a'`。

character-literal:
`'` *character* `'`

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

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

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

hexadecimal-escape-sequence:
`\x` *hex-digit* *hex-digit*$_{opt}$ *hex-digit*$_{opt}$ *hex-digit*$_{opt}$

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在 *character* 中跟在反斜杠字符 (\) 之后的字符必须是以下字符之一：'、"、\、0、a、b、f、n、r、t、u、U、x 和 v。其他情况下，将发生编译时错误。

十六进制转义序列表示单个 Unicode 字符，它的值由“\x”后接十六进制数组成。

如果一个字符表示的值大于 U+FFFF，则将发生编译时错误。

字符中的 Unicode 字符转义序列（第 2.4.1 节）必须在 U+0000 到 U+FFFF 的范围内。

一个简单转义序列表示一个 Unicode 字符编码，详见下表。

转义序列	字符名称	Unicode 编码
\'	单引号	0x0027
\"	双引号	0x0022
\\	反斜杠	0x005C
\0	Null	0x0000
\a	警报	0x0007
\b	Backspace	0x0008
\f	换页符	0x000C
\n	换行符	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

character-literal 的类型为 char。

2.4.4.5 字符串

C# 支持两种形式的字符串：常规字符串 (regular string literal) 和原义字符串 (verbatim string literal)。

正则字符串由包含在双引号中的零个或多个字符组成（如 "hello"），并且可以包含简单转义序列（如表示制表符的 \t）、十六进制转义序列和 Unicode 转义序列。

原义字符串由 @ 字符后跟开始的双引号字符、零个或多个字符以及结束的双引号字符组成。一个简单的示例就是 @"hello"。在原义字符串中，分隔符之间的字符逐字解释，唯一的例外是 *quote-escape-sequence*。具体而言，在原义字符串中不处理简单转义序列以及十六进制和 Unicode 转义序列。原义字符串可以跨多行。

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

regular-string-literal:
    " *regular-string-literal-characters*$_{opt}$ "

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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-escape-sequence

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

verbatim-string-literal:
    @" *verbatim-string-literal-characters*$_{opt}$ "

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:
    ""

在 *regular-string-literal-character* 中跟在反斜杠字符 (\) 之后的字符必须是以下字符之一：'、"、\、0、a、b、f、n、r、t、u、U、x、v。其他情况下，将发生编译时错误。

下面的示例

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

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

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

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

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

演示了各种不同的字符串。最后一个字符串 j 是跨多行的原义字符串。引号之间的字符（包括空白，如换行符等）也逐字符保留。

由于十六进制转义序列可以包含数目可变的十六进制数字，因此字符串 "\x123" 只包含一个具有十六进制值 123 的字符。若要创建一个包含具有十六进制值 12 的字符，后跟一个字符 3 的字符串，可以改写为 "\x00123" 或 "\x12" + "3"。

string-literal 的类型为 string。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

每个字符串文本不一定产生新的字符串实例。当根据字符串相等运算符（第7.10.7节）确认为相等的两个或更多个字符串出现在同一个程序中时，这些字符串引用相同的字符串实例。例如，

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

产生的输出为`True`，这是因为两个字符串引用相同的字符串实例。

2.4.4.6 null 文本

null-literal:
`null`

可以将 *null-literal* 隐式转换为引用类型或可以为 null 的类型。

2.4.5 运算符和标点符号

有若干种运算符和标点符号。运算符在表达式中用于描述涉及一个或多个操作数的运算。例如，表达式 `a + b` 使用 `+` 运算符添加两个操作数 `a` 和 `b`。标点符号用于分组和分隔。

operator-or-punctuator: one of

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

right-shift:
`>|>`

right-shift-assignment:
`>|>=`

right-shift 和 *right-shift-assignment* 产生式中的竖线用来表示：和采用句法文法的其他产生式不同，在标记之间不允许有任何类型的字符（甚至不允许空白）。为了能正确处理 *type-parameter-list*（第10.1.3节），要对这些产生式进行特殊处理。

2.5 预处理指令

预处理指令提供按条件跳过源文件中的节、报告错误和警告条件，以及描绘源代码的不同区域的能力。使用术语“预处理指令”只是为了与 C 和 C++ 编程语言保持一致。在 C# 中没有单独的预处理步骤；预处理指令按词法分析阶段的一部分处理。

pp-directive:
pp-declaration
pp-conditional
pp-line
pp-diagnostic
pp-region
pp-pragma

下面是可用的预处理指令：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- #define 和 #undef，分别用于定义和取消定义条件编译符号（第 2.5.3 节）。
- #if、#elif、#else 和 #endif，用于按条件跳过源代码中的节（第 2.5.4 节）。
- #line，用于控制行号（在发布错误和警告信息时使用）（第 2.5.7 节）。
- #error 和 #warning，分别用于发出错误和警告（第 2.5.5 节）。
- #region 和 #endregion，用于显式标记源代码中的节（第 2.5.6 节）。
- #pragma，用于为编译器指定可选的上下文信息（第 2.5.8 节）。

预处理指令总是占用源代码中的单独一行，并且总是以 # 字符和预处理指令名称开头。# 字符的前面以及 # 字符与指令名称之间可以出现空白符。

包含 #define、#undef、#if、#elif、#else、#endif、#line 或 #endregion 指令的源代码行可以用单行注释结束。在包含预处理指令的源行上不允许使用带分隔符的注释（/* */ 样式的注释）。

预处理指令既不是标记，也不是 C# 句法文法的组成部分。但是，可以用预处理指令包含或排除标记序列，并且可以以这种方式影响 C# 程序的含义。例如，编译后，程序：

```
#define A
#undef B

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

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

产生与下面的程序完全相同的标记序列：

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

因此，尽管上述两个程序在词法分析中完全不同，但它们在句法分析中是相同的。

2.5.1 条件编译符号

#if、#elif、#else 和 #endif 指令提供的条件编译功能是通过预处理表达式（第 2.5.2 节）和条件编译符号来控制的。

conditional-symbol:
    Any *identifier-or-keyword* except `true` or `false`

条件编译符号具有两种可能的状态：已定义 (defined) 或未定义 (undefined)。在源文件词法处理开始时，条件编译符号除非已由外部机制（如命令行编译器选项）显式定义，否则是未定义的。当处理 #define 指令时，在该指令中指定的条件编译符号在那个源文件中成为已定义的符号。此后，该符号就一直保持已定义的状态，直到处理一条关于同一符号的 #undef 指令，或者到达源文件的结尾。这意味着一个源文件中的 #define 和 #undef 指令对同一程序中的其他源文件没有任何影响。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

条件编译符号的命名空间与 C# 程序中的所有其他命名实体截然不同。只能在 `#define` 和 `#undef` 指令以及预处理表达式中引用条件编译符号。

2.5.2 预处理表达式

预处理表达式可以出现在 `#if` 和 `#elif` 指令中。在预处理表达式中允许使用 `!`、`==`、`!=`、`&&` 和 `||` 运算符，并且可以使用括号进行分组。

pp-expression:
    whitespace$_{opt}$ *pp-or-expression* *whitespace*$_{opt}$

pp-or-expression:
    pp-and-expression
    pp-or-expression *whitespace*$_{opt}$ `||` *whitespace*$_{opt}$ *pp-and-expression*

pp-and-expression:
    pp-equality-expression
    pp-and-expression *whitespace*$_{opt}$ `&&` *whitespace*$_{opt}$ *pp-equality-expression*

pp-equality-expression:
    pp-unary-expression
    pp-equality-expression *whitespace*$_{opt}$ `==` *whitespace*$_{opt}$ *pp-unary-expression*
    pp-equality-expression *whitespace*$_{opt}$ `!=` *whitespace*$_{opt}$ *pp-unary-expression*

pp-unary-expression:
    pp-primary-expression
    `!` *whitespace*$_{opt}$ *pp-unary-expression*

pp-primary-expression:
    `true`
    `false`
    conditional-symbol
    `(` *whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$ `)`

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

预处理表达式的计算总是产生一个布尔值。预处理表达式的计算规则与常量表达式（第 7.19 节）相同，唯一的例外是：在这里，唯一可引用的用户定义实体是条件编译符号。

2.5.3 声明指令

声明指令用于定义或取消定义条件编译符号。

pp-declaration:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `define` *whitespace* *conditional-symbol* *pp-new-line*
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `undef` *whitespace* *conditional-symbol* *pp-new-line*

pp-new-line:
    whitespace$_{opt}$ *single-line-comment*$_{opt}$ *new-line*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

对 #define 指令的处理使给定的条件编译符号成为已定义的符号（从跟在指令后面的源代码行开始）。类似地，对 #undef 指令的处理使给定的条件编译符号成为未定义的符号（从跟在指令后面的源代码行开始）。

源文件中的任何 #define 和 #undef 指令都必须出现在源文件中第一个 *token*（第 2.4 节）的前面，否则将发生编译时错误。直观地讲，#define 和 #undef 指令必须位于源文件中所有“实代码”的前面。

示例：

```
#define Enterprise

#if Professional || Enterprise
   #define Advanced
#endif

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

是有效的，这是因为 #define 指令位于源文件中第一个标记（namespace 关键字）的前面。

下面的示例产生编译时错误，因为 #define 指令在实代码后面出现：

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

#define 指令可用于重复地定义一个已定义的条件编译符号，而不必对该符号插入任何 #undef。下面的示例定义一个条件编译符号 A，然后再次定义它。

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

#undef 可以“取消定义”一个本来已经是未定义的条件编译符号。下面的示例定义一个条件编译符号 A，然后两次取消定义该符号；第二个 #undef 没有作用但仍是有效的。

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

2.5.4 条件编译指令

条件编译指令用于按条件包含或排除源文件中的某些部分。

pp-conditional:
    pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*

pp-if-section:
    whitespace$_{opt}$ # *whitespace*$_{opt}$ if *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-elif-sections:
    pp-elif-section
    pp-elif-sections *pp-elif-section*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

pp-elif-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `elif` *whitespace pp-expression pp-new-line conditional-section*$_{opt}$

pp-else-section:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `else` *pp-new-line conditional-section*$_{opt}$

pp-endif:
whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endif` *pp-new-line*

conditional-section:
input-section
skipped-section

skipped-section:
skipped-section-part
skipped-section skipped-section-part

skipped-section-part:
skipped-characters$_{opt}$ *new-line*
pp-directive

skipped-characters:
whitespace$_{opt}$ *not-number-sign input-characters*$_{opt}$

not-number-sign:
Any *input-character* except #

按照语法的规定，条件编译指令必须写成集的形式，集的组成依次为：一个 `#if` 指令、一个或多个 `#elif` 指令（或没有）、一个或多个 `#else` 指令（或没有）和一个 `#endif` 指令。指令之间是源代码的条件节。每节代码直接位于它前面的那个指令控制。条件节本身可以包含嵌套的条件编译指令，前提是这些指令构成完整的指令集。

pp-conditional 最多只能选择它所包含的 *conditional-section* 之一去做通常的词法处理：

- 按顺序计算 `#if` 和 `#elif` 指令的 *pp-expression*，直到得出 `true` 值。如果表达式的结果为 `true`，则选择对应指令的 *conditional-section*。
- 如果所有 *pp-expression* 的结果都为 `false` 并且存在 `#else` 指令，则选择 `#else` 指令的 *conditional-section*。
- 否则不选择任何 *conditional-section*。

所选的 *conditional-section*（如果有）作为正常 *input-section* 进行处理：节中包含的源代码必须符合词法文法；标记由节中的源代码生成；并且节中的预处理指令具有规定的效果。

剩余的 *conditional-section*（如果有）作为 *skipped-sections* 进行处理：除了预处理指令，节中的源代码不必一定要符合词法文法；不从节中的源代码生成任何词法标记；节中的预处理指令必须在词法上正确，但不另外处理。在按 *skipped-section* 处理的 *conditional-section* 中，任何嵌套的 *conditional-section*（包含在嵌套的 `#if`...`#endif` 和 `#region`...`#endregion` 构造中）也按 *skipped-section* 处理。

下面的示例阐释如何嵌套条件编译指令：

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

除预处理指令外，跳过的源代码与词法分析无关。例如，尽管在 `#else` 节中有未结束的注释，但下面的示例仍然有效：

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

但请注意，即使是在源代码的跳过节中，也要求预处理指令在词法上正确。

当预处理指令出现在多行输入元素的内部时，不作为预处理指令处理。例如，程序：

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

输出结果为：

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

在特殊情况下，处理的预处理指令集合可有取决于 *pp-expression* 的计算结果。示例：

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

总是生成同样的标记流 (class Q { })，不管是否定义了 X。如果定义了 X，由于多行注释的缘故，只处理 #if 和 #endif 指令。如果未定义 X，则这三个指令（#if、#else、#endif）都是指令集的组成部分。

2.5.5 诊断指令

诊断指令用于显式生成错误信息和警告消息，这些信息的报告方式与其他编译时错误和警告相同。

pp-diagnostic:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `error` *pp-message*
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `warning` *pp-message*

pp-message:
    new-line
    whitespace *input-characters*$_{opt}$ *new-line*

示例：

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

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

class Test {...}
```

总是产生一个警告（“Code review needed before check-in”），如果同时定义条件符号 Debug 和 Retail，则产生一个编译时错误（“A build can't be both debug and retail”）。请注意，*pp-message* 可以包含任意文本；具体而言，它可以包含格式不正确的标记，比如单词 can't 中的单引号。

2.5.6 区域指令

区域指令用于显式标记源代码的区域。

pp-region:
    pp-start-region *conditional-section*$_{opt}$ *pp-end-region*

pp-start-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `region` *pp-message*

pp-end-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endregion` *pp-message*

区域不具有任何附加的语义含义；区域旨在由程序员或自动工具用来标记源代码中的节。#region 或 #endregion 指令中指定的消息同样不具有任何语义含义；它只是用于标识区域。匹配的 #region 和 #endregion 指令可能具有不同的 *pp-message*。

区域的词法处理：

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

与以下形式的条件编译指令的词法处理完全对应：

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

2.5.7 行指令

行指令可用于变更编译器在输出（如警告和错误）中报告的行号和源文件名称，以及调用方信息特性（第 17.4.4 节）所使用的行号和源文件名称。

行指令最常用于从某些其他文本输入生成 C# 源代码的元编程工具。

pp-line:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `line` *whitespace* *line-indicator* *pp-new-line*

line-indicator:
    decimal-digits *whitespace* *file-name*
    decimal-digits
    `default`
    `hidden`

file-name:
    `"` *file-name-characters* `"`

file-name-characters:
    file-name-character
    file-name-characters *file-name-character*

file-name-character:
    Any *input-character* except "

当不存在 `#line` 指令时，编译器在它的输出中报告真实的行号和源文件名称。当处理的 `#line` 指令包含不是 `default` 的 *line-indicator* 时，编译器将该指令后面的行视为具有给定的行号（如果指定了，还包括文件名）。

`#line default` 指令消除前面所有 #line 指令的影响。编译器报告后续行的真实行信息，就像尚未处理任何 `#line` 指令一样。

`#line hidden` 指令对错误信息中报告的文件号和行号无效，但对源代码级调试确实有效。调试时，`#line hidden` 指令和后面的 `#line` 指令（不是 `#line hidden`）之间的所有行都没有行号信息。在调试器中逐句执行代码时，将全部跳过这些行。

注意，*file-name* 与正则字符串的不同之处在于前者不处理转义字符；“\”字符在 *file-name* 中只是表示一个普通的反斜杠字符。

2.5.8 Pragma 指令

`#pragma` 预处理指令用来向编译器指定可选的上下文信息。`#pragma` 指令中提供的信息永远不会更改程序语义。

pp-pragma:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `pragma` *whitespace* *pragma-body* *pp-new-line*

pragma-body:
    pragma-warning-body

C# 提供 `#pragma` 指令以控制编译器警告。此语言将来的版本可能包含更多的 `#pragma` 指令。为了确保与其他 C# 编译器的互操作性，Microsoft C# 编译器对于未知的 `#pragma` 指令不会发出编译错误；但是这类指令确实会生成警告。

2.5.8.1 Pragma warning

`#pragma warning` 指令用于在编译后续程序文本的过程中禁用或恢复所有或特定的一部分警告消息 tex。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

pragma-warning-body:
warning *whitespace warning-action*
warning *whitespace warning-action whitespace warning-list*

warning-action:
disable
restore

warning-list:
decimal-digits
warning-list whitespace$_{opt}$, *whitespace*$_{opt}$ *decimal-digits*

省略了警告列表的 #pragma warning 指令将影响所有警告。包含警告列表的 #pragma warning 指令只影响该列表中列出的警告。

#pragma warning disable 指令将禁用所有警告或给定的一组警告。

#pragma warning restore 指令将所有警告或指定警告恢复为在编译单元的开始处有效的状态。请注意，如果在外部禁用了特定的警告，则 #pragma warning restore（无论是恢复所有警告还是恢复特定警告）将不会重新启用该警告。

下面的示例演示的是使用 #pragma warning 临时禁用在通过 Microsoft C# 编译器中的警告编号引用已过时的成员时报告的警告。

```
using System;

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

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

3. 基本概念

3.1 应用程序启动

具有入口点 (entry point) 的程序集称为应用程序 (application)。应用程序运行时，将创建新的应用程序域 (application domain)。同一台计算机上可能会同时运行着同一个应用程序的若干个实例，此时，每一个实例都拥有各自的应用程序域。

应用程序域用作应用程序状态的容器，以此隔离应用程序。应用程序域作为应用程序中和它使用的类库中所定义的类型的容器和边界。同一个类型若被加载到不同的应用程序域中就成为各自独立的客体，由它们在各自应用程序域中产生的实例亦不可直接共享。例如，对于这些类型的静态变量，每个应用程序域都有自己的副本，并且这些类型的静态构造函数在每个应用程序域中也要（最多）运行一次。关于如何处理程序域的创建和销毁，各实现可以按具体情况确定自己的策略或机制。

当执行环境调用指定的方法（称为应用程序的入口点）时发生应用程序启动(application startup)。此入口点方法总是被命名为 `Main`，可以具有下列签名之一：

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

如上所示，入口点可以选择返回一个 `int` 值。此返回值用于应用程序终止（第 3.2 节）。

入口点可以包含一个形参（可选）。该参数可以具有任意名称，但参数的类型必须为 `string[]`。如果存在形参，执行环境会创建并传递一个包含命令行实参的 `string[]` 实参，这些命令行实参是在启动应用程序时指定的。`string[]` 参数永远不能为 null，但如果没有指定命令行参数，它的长度可以为零。

由于 C# 支持方法重载，因此类或结构可以包含某个方法的多个定义（前提是每个定义有不同的签名）。但在一个程序内，没有任何类或结构可以包含一个以上的名为 `Main` 的方法，因为 Main 的定义限定它只能被用作应用程序的入口点。允许使用 `Main` 的其他重载版本，前提是它们具有一个以上的参数，或者它们的唯一参数的类型不是 `string[]`。

应用程序可由多个类或结构组成。在这些类或结构中，可能会有若干个拥有自己的 `Main` 方法，因为 Main 的定义限定它只能被用作应用程序的入口点。这样的情况下，必须利用某种外部机制（如命令行编译器的选项）来选择其中一个 `Main` 方法用作入口点。

在 C# 中，每个方法都必须定义为类或结构的成员。通常，方法的已声明可访问性（第 3.5.1 节）由其声明中指定的访问修饰符（第 10.3.5 节）确定。同样，类型的已声明可访问性由其声明中指定的访问修饰符确定。为了能够调用给定类型的给定方法，类型和成员都必须是可访问的。然而，应用程序入口点是一种特殊情况。具体而言，执行环境可以访问应用程序的入口点，无论它本身的可访问性和封闭它的类型的可访问性是如何在声明语句中设置的。

应用程序入口点方法不能位于泛型类声明中。

在所有其他方面，入口点方法的行为与非入口点方法类似。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

3.2 应用程序终止

应用程序终止 (application termination) 将控制返回给执行环境。

如果应用程序的入口点 (entry point) 方法的返回类型为 `int`，则返回的值用作应用程序的终止状态代码 (termination status code)。此代码的用途是允许与执行环境进行关于应用程序运行状态（成功或失败）的通信。

如果入口点方法的返回类型为 `void`，那么在到达终止该方法的右大括号 (})，或者执行不带表达式的 `return` 语句时，将产生终止状态代码 `0`。

在应用程序终止之前，将调用其中还没有被垃圾回收的所有对象的析构函数，除非已将这类清理功能设置为取消使用（例如，通过调用库方法 `GC.SuppressFinalize`）。

3.3 声明

C# 程序中的声明定义程序的构成元素。C# 程序是用命名空间（第 9 章）组织起来的，一个命名空间可以包含类型声明和嵌套的命名空间声明。类型声明（第 9.6 节）用于定义类（第 10 章）、结构（第 10.14 节）、接口（第 13 章）、枚举（第 14 章）和委托（第 15 章）。在一个类型声明中可以使用哪些类型作为其成员，取决于该类型声明的形式。例如，类声明可以包含常量声明（第 10.4 节）、字段声明（第 10.5 节）、方法声明（第 10.6 节）、属性声明（第 10.7 节）、事件声明（第 10.8 节）、索引器声明（第 REF10.9 节）、运算符声明（第 10.10 节）、实例构造函数声明（第 10.11 节）、静态构造函数声明（第 10.12 节）、析构函数声明（第 10.13 节）和嵌套类型声明（第 10.3.8 节）。

一个声明在它自己所属的那个声明空间 (declaration space) 中定义一个名称。除非是重载成员（第 3.6 节），否则，在同一个声明空间下若有两个以上的声明语句声明了具有相同名称的成员，就会产生编译时错误。同一个声明空间内绝不能包含不同类型的同名成员。例如，声明空间绝不能包含同名的字段和方法。

有若干种不同类型的声明空间，如下所述。

- 在程序的所有源文件中，*namespace-member-declaration* 若没有被置于任何一个 *namespace-declaration* 下，则属于一个称为全局声明空间 (global declaration space) 的组合声明空间。
- 在程序的所有源文件中，一个 *namespace-member-declaration* 若在 *namespace-declaration* 中具有相同的完全限定的命名空间名称，它就属于一个组合声明空间。
- 每个类、结构或接口声明创建一个新的声明空间。名称将通过 *class-member-declaration*、*struct-member-declaration*、*interface-member-declaration* 或 *type-parameter* 引入此声明空间。除了重载实例构造函数声明和静态构造函数声明外，类或结构不能包含与该类或结构同名的成员声明。类、结构或接口允许声明重载方法和索引器。另外，类或结构允许重载实例构造函数和运算符的声明。例如，类、结构或接口可以包含多个同名的方法声明，前提是这些方法声明的签名（第 3.6 节）不同。注意，基类与类的声明空间无关，基接口与接口的声明空间无关。因此，允许在派生类或接口内声明与所继承的成员同名的成员。我们说这类成员隐藏 (hide) 了它们继承的那些成员。
- 每个委托声明创建一个新的声明空间。名称通过形参（*fixed-parameter* 和 *parameter-array*）和 *type-parameter* 引入此声明空间。
- 每个枚举声明创建一个新的声明空间。名称通过 *enum-member-declarations* 引入此声明空间。
- 每个方法声明、索引器声明、运算符声明、实例构造函数声明和匿名函数均创建一个称为局部变量声明空间 (local variable declaration space) 的新声明空间。名称将通过形参（*fixed-parameter* 和 *parameter-array*）和 *type-parameter* 引入此声明空间。函数成员或匿名函数的主体（如果有）将视为

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

嵌套在局部变量声明空间中。如果局部变量声明空间和嵌套的局部变量声明空间包含具有相同名称的元素，则会发生错误。因此，在嵌套声明空间中不可能声明与封闭它的声明空间中的局部变量或常量同名的局部变量或常量。只要两个声明空间彼此互不包含，这两个声明空间就可以包含同名的元素。

- 每个 *block* \b *或 switch-block* \b \t "See declaration space, block and" \b \t "See declaration space, block and" \b *以及 for*、*foreach* 和 *using* 语句都会为局部变量和局部常量创建一个局部变量声明空间 \b \t "See declaration space, block and" \b \t "See declaration space, switch block" 。名称将通过 *local-variable-declaration* 和 *local-constant-declaration* 引入此声明空间。请注意，作为函数成员的主体或匿名函数的主体出现或出现在该主体之中的块将嵌套在这些函数为其参数声明的局部变量声明空间中。因此，如果某个方法的局部变量和参数具有相同名称，则会发生错误。
- 每个 *block* 或 *switch-block* 都为标签创建一个单独的声明空间。名称将通过 *labeled-statement* 引入此声明空间，并通过 *goto-statement* 进行引用。块的*标签声明空间*可包含任何嵌套块。因此，在嵌套块中不可能声明与封闭它的块中的标签同名的标签。

声明名称的文本顺序通常不重要。具体而言，声明和使用命名空间、常量、方法、属性、事件、索引器、运算符、实例构造函数、析构函数、静态构造函数和类型时，文本顺序并不重要。在下列情况下声明顺序非常重要：

- 字段声明和局部变量声明的声明顺序确定其初始值设定项（如果有）的执行顺序。
- 在使用局部变量前必须先定义它们（第3.7 节）。
- 当省略 *constant-expression* 值时，枚举成员声明（第 14.3 节）的声明顺序非常重要。

命名空间的声明空间是“开放式的”，两个具有相同的完全限定名的命名空间声明共同构成同一个声明空间。例如

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

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

上面的两个命名空间声明为同一声明空间提供了成员，在本例中它们分别声明了具有完全限定名 `Megacorp.Data.Customer` 和 `Megacorp.Data.Order` 的两个类。由于两个声明共同构成同一个声明空间，因此如果每个声明中都包含一个同名类的声明，则将导致编译时错误。

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
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();
   }
}
```

3.4 成员

命名空间和类型具有成员 (member)。通常可以通过限定名来访问实体的成员。限定名以对实体的引用开头，后跟一个“.”标记，再接成员的名称。

类型的成员或者是在该类型声明中声明的，或者是从该类型的基类继承 (inherit) 的。当类型从基类继承时，基类的所有成员（实例构造函数、析构函数和静态构造函数除外）都成为派生类型的成员。基类成员的声明可访问性并不控制该成员是否可继承：继承性可扩展到任何成员，只要它们不是实例构造函数、静态构造函数或析构函数。然而，在派生类型中可能不能访问已被继承的成员，原因或者是因为其已声明可访问性（第 3.5.1 节），或者是因为它已被类型本身中的声明所隐藏（第 3.7.1.2 节）。

3.4.1 命名空间成员

命名空间和类型若没有封闭它的命名空间，则属于全局命名空间 (global namespace) 的成员。这直接对应于全局声明空间中声明的名称。

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

命名空间没有访问限制。不可能把命名空间设置成私有的、受保护的或内部的，命名空间名称始终是可公开访问的。

3.4.2 结构成员

结构的成员是在结构中声明的成员以及继承自结构的直接基类 `System.ValueType` 和间接基类 `object` 的成员。

简单类型的成员直接对应于结构类型的成员，此简单类型正是该结构的化名：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

3.4.3 枚举成员

枚举的成员是在枚举中声明的常量以及继承自枚举的直接基类 System.Enum 和间接基类 System.ValueType 和 object 的成员。

3.4.4 类成员

类的成员是在类中声明的成员和从该类的基类（没有基类的 object 类除外）继承的成员。从基类继承的成员包括基类的常量、字段、方法、属性、事件、索引器、运算符和类型，但不包括基类的实例构造函数、析构函数和静态构造函数。基类成员被是否继承与它们的可访问性无关。

类声明可以包含以下对象的声明：常量、字段、方法、属性、事件、索引器、运算符、实例构造函数、析构函数、静态构造函数和类型。

object 和 string 的成员直接对应于它们所化名的类类型的成员：

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

3.4.5 接口成员

接口的成员是在接口中和该接口的所有基接口中声明的成员。严格地说，类 object 中的成员不是任何接口的成员（第 13.2 节）。但是，通过在任何接口类型中进行成员查找，可获得类 object 中的成员（第 7.4 节）。

3.4.6 数组成员

数组的成员是从类 System.Array 继承的成员。

3.4.7 委托成员

委托的成员是从类 System.Delegate 继承的成员。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

3.5 成员访问

成员的声明可用于控制对该成员的访问。成员的可访问性是由该成员的声明可访问性（第3.5.1 节）和直接包含它的那个类型的可访问性（若它存在）结合起来确定的。

如果允许访问特定成员，则称该成员是可访问的 (accessible)。相反，如果不允许访问特定成员，则称该成员是不可访问的 (inaccessible)。当引发访问的源代码的文本位置在某成员的可访问域（第3.5.2 节）中时，允许对该成员进行访问。

3.5.1 已声明可访问性

成员的已声明可访问性 (declared accessibility) 可以是下列类型之一：

- Public，选择它的方法是在成员声明中包括 `public` 修饰符。`public` 的直观含义是“访问不受限制”。
- Protected，选择它的方法是在成员声明中包括 `protected` 修饰符。`protected` 的直观含义是“访问范围限定于它所属的类或从该类派生的类型”。
- Internal，选择它的方法是在成员声明中包括 `internal` 修饰符。`internal` 的直观含义是“访问范围限定于此程序”。
- Protected internal（意为受保护或内部的），选择它的方法是在成员声明中包括 `protected` 和 `internal` 修饰符。`protected internal` 的直观含义是“访问范围限定于此程序或那些由它所属的类派生的类型”。
- Private，选择它的方法是在成员声明中包括 `private` 修饰符。`private` 的直观含义是“访问范围限定于它所属的类型”。

声明一个成员时所能选择的已声明可访问性的类型，依赖于该成员声明出现处的上下文。此外，当成员声明不包含任何访问修饰符时，声明发生处的上下文会为该成员选择一个默认的已声明可访问性。

- 命名空间隐式具有 `public` 已声明可访问性。在命名空间声明中不允许使用访问修饰符。
- 编译单元或命名空间中声明的类型可以具有 `public` 或 `internal` 已声明可访问性，默认的已声明可访问性为 `internal`。
- 类成员可具有五种已声明可访问性中的任何一种，默认为 `private` 已声明可访问性。（请注意，声明为类成员的类型可具有五种已声明可访问性中的任何一种，而声明为命名空间成员的类型只能具有 `public` 或 `internal` 已声明可访问性。）
- 结构成员可以具有 `public`、`internal` 或 `private` 已声明可访问性并默认为 `private` 已声明可访问性，这是因为结构是隐式密封的。结构的成员若是在此结构中声明的（也就是说，不是由该结构从它的基类中继承的），则不能具有 `protected` 或 `protected internal` 已声明可访问性。（请注意，声明为结构成员的类型可具有 `public`、`internal` 或 `private` 已声明可访问性，而声明为命名空间成员的类型只能具有 `public` 或 `internal` 已声明可访问性。）
- 接口成员隐式地具有 `public` 已声明可访问性。在接口成员声明中不允许使用访问修饰符。
- 枚举成员隐式地具有 `public` 已声明可访问性。在枚举成员声明中不允许使用访问修饰符。

3.5.2 可访问域

一个成员的可访问域 (accessibility domain) 由（可能是不连续的）程序文本节组成，从该域中可以访问该成员。出于定义成员可访问域的目的，如果成员不是在某个类型内声明的，就称该成员是顶级 (top-

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

level) 的；如果成员是在其他类型内声明的，就称该成员是嵌套 (nested) 的。此外，程序的程序文本 (program text) 定义为包含在该程序的所有源文件中的全部程序文本，而类型的程序文本定义为包含在该类型（可能还包括该类型中的嵌套类型）的 *type-declaration* 中的所有程序文本。

预定义类型（如 `object`、`int` 或 `double`）的可访问域无限制。

在程序 `P` 中声明的顶级未绑定类型 `T`（第 4.4.3 节）的可访问域定义如下：

- 如果 `T` 的已声明可访问性为 `public`，则 `T` 的可访问域是 `P` 以及引用 `P` 的任何程序的程序文本。
- 如果 `T` 的已声明可访问性为 `internal`，则 `T` 的可访问域就是 `P` 的程序文本。

从这些定义可以推断出：顶级未绑定类型的可访问域始终至少是声明了该类型的程序的程序文本。

构造类型 $T<A_1, \ldots, A_N>$ 的可访问域是未绑定的泛型类型 `T` 的可访问域和类型实参 $A_1, \ldots, A_N$ 的可访问域的交集。

在程序 `P` 内的类型 `T` 中声明的嵌套成员 `M` 的可访问域定义如下（注意 `M` 本身可能就是一个类型）：

- 如果 `M` 的已声明可访问性为 `public`，则 `M` 的可访问域就是 `T` 的可访问域。
- 如果 `M` 的已声明可访问性是 `protected internal`，则设 `D` 表示 `P` 的程序文本和从 `T`（在 `P` 外部声明）派生的任何类型的程序文本的并集。`M` 的可访问域是 `T` 与 `D` 的可访问域的交集。
- 如果 `M` 的已声明可访问性是 `protected`，则设 `D` 表示 `T` 的程序文本和从 `T` 派生的任何类型的程序文本的并集。`M` 的可访问域是 `T` 与 `D` 的可访问域的交集。
- 如果 `M` 的已声明可访问性为 `internal`，则 `M` 的可访问域就是 `T` 的可访问域与 `P` 的程序文本之间的交集。
- 如果 `M` 的已声明可访问性为 `private`，则 `M` 的可访问域就是 `T` 的程序文本。

从这些定义可以看出：嵌套成员的可访问域总是至少为声明该成员的类型的程序文本。还可以看出：成员的可访问域包含的范围决不会比声明该成员的类型的可访问域更广。

直观地讲，当访问类型或成员 `M` 时，按下列步骤进行计算以确保允许进行访问：

- 首先，如果 `M` 是在某个类型（相对于编译单元或命名空间）内声明的，则当该类型不可访问时将会发生编译时错误。
- 然后，如果 `M` 为 `public`，则允许进行访问。
- 否则，如果 `M` 为 `protected internal`，则当访问发生在声明了 `M` 的程序中，或发生在从声明 `M` 的类派生的类中并通过派生类类型（第 3.5.3 节）进行访问时，允许进行访问。
- 否则，如果 `M` 为 `protected`，则当访问发生在声明了 `M` 的类中，或发生在从声明 `M` 的类派生的类中并通过派生类类型（第 3.5.3 节）进行访问时，允许进行访问。
- 否则，如果 `M` 为 `internal`，则当访问发生在声明了 `M` 的程序中时允许进行访问。
- 否则，如果 `M` 为 `private`，则当访问发生在声明了 `M` 的类型中时允许进行访问。
- 否则，类型或成员不可访问，并发生编译时错误。

在下面的示例中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

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

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

类和成员具有下列可访问域：

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

如示例所示，成员的可访问域绝不会大于包含它的类型的可访问域。例如，即使所有的 X 成员都具有公共级的已声明可访问性，但除了 A.X 外，所有其他成员的可访问域都受包含类型的约束。

如第 3.4 节中所描述的那样，基类的所有成员（实例构造函数、析构函数和静态构造函数除外）都由派生类型继承。这甚至包括基类的私有成员。但是，私有成员的可访问域只包括声明该成员的类型的程序文本。在下面的示例中

```
class A
{
   int x;

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

类 B 继承类 A 的私有成员 x。因为该成员是私有的，所以只能在 A 的 *class-body* 中对它进行访问。因此，对 b.x 的访问在 A.F 方法中取得了成功，在 B.F 方法中却失败了。

3.5.3 实例成员的受保护访问

当在声明了某个 protected 实例成员的类的程序文本之外访问该实例成员时，以及当在声明了某个 protected internal 实例成员的程序的程序文本之外访问该实例成员时，这种访问必须发生在声明了该成员的类的一个派生类的类声明中。而且，要求这种访问通过该成员所属类的派生类类型的实例或从它构造的类类型的实例发生。此限制阻止一个派生类访问其他派生类的受保护成员，即使成员继承自同一个基类也是如此。

假定 B 是一个基类，它声明了一个受保护的实例成员 M，而 D 是从 B 派生的类。在 D 的 *class-body* 中，对 M 的访问可采取下列形式之一：

- M 形式的非限定 *type-name* 或 *primary-expression*。
- E.M 形式的 *primary-expression*，假定 E 的类型是 T 或从 T 派生的类，其中 T 为类类型 D 或从 D 构造的类类型
- base.M 形式的 *primary-expression*。

除了上述访问形式外，派生类还可以在 *constructor-initializer* 中（第 10.11.1 节）访问基类的受保护的实例构造函数。

在下面的示例中

```
public class A
{
   protected int x;

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

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

在 A 中可以通过 A 和 B 的实例访问 x，这是因为在两种情况下访问都通过 A 的实例或从 A 派生的类发生。但是在 B 中，由于 A 不从 B 派生，所以不可能通过 A 的实例访问 x。

在下面的示例中

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

对 x 的三个赋值是允许的，因为它们全都通过从该泛型类型构造的类类型的实例进行。

3.5.4 可访问性约束

C# 语言中的有些构造要求某个类型至少与某个成员或其他类型具有同样的可访问性(at least as accessible as)。如果 T 的可访问域是 M 可访问域的超集，我们就说类型 T 至少与成员或类型 M 具有同样的可访问性。换言之，如果 T 在可访问 M 的所有上下文中都是可访问的，则 T 至少与 M 具有同样的可访问性。

存在下列可访问性约束：

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

在下面的示例中

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

B 类导致编译时错误，因为 A 并不具有至少与 B 相同的可访问性。

同样，在示例中

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

B 中的方法 H 导致编译时错误，因为返回类型 A 并不具有至少与该方法相同的可访问性。

3.6 签名和重载

方法、实例构造函数、索引器和运算符是由它们的签名 (signature) 来刻画的：

- 方法签名由方法的名称、类型形参的个数和它的每一个形参的类型和形参传递模式（值、引用或输出）组成（按从左到右的顺序）。为了实现这些目的，形参的类型中出现的方法的任何类型形参都不是由其名称标识的，而是由它在方法的类型实参列表中的序号位置标识的。需注意的是，方法签名既不包含返回类型和 params 修饰符（它可用于指定最右边的形参），也不包含可选类型形参约束。
- 实例构造函数签名由它的每一个形参的类型和形参传递模式（值、引用或输出）组成（按从左到右的顺序）。具体而言，实例构造函数的签名不包含可为最右边的参数指定的 params 修饰符。
- 索引器签名由它的每一个形参的类型组成（按从左到右的顺序）。具体而言，索引器签名既不包含元素类型，也不包含可为最右边的形参指定的 params 形参数组签名和 修饰符。
- 运算符签名由运算符的名称和它的每一个形参的类型组成（按从左到右的顺序）。具体而言，运算符的签名不包含结果类型。

如果同一成员类型的两个签名具有相同的名称、类型形参的个数和形参传递模式，并且其对应的类型之间存在标识转换，则会将这两个签名视为*同一签名*（第 6.1.1 节）。

签名是对类、结构和接口的成员实施重载 (overloading) 的机制：

- 方法重载允许类、结构或接口用同一个名称声明多个方法，条件是它们的签名在该类、结构或接口中是唯一的。
- 实例构造函数重载允许类或结构声明多个实例构造函数，条件是它们的签名在该类或结构中是唯一的。
- 索引器重载允许类、结构或接口声明多个索引器，条件是它们的签名在该类、结构或接口中是唯一的。
- 运算符重载允许类或结构用同一名称声明多个运算符，条件是它们的签名在该类或结构中是唯一的。

虽然 out 和 ref 参数修饰符被视为签名的一部分，但是在同一个类型中声明的成员不能仅通过 ref 和 out 在签名上加以区分。在同一类型中声明了两个成员时，如果将这两个方法中带有 out 修饰符的所有形参更改为 ref 修饰符会使这两个成员的签名相同，则会发生编译时错误。出于签名匹配的其他目的（如隐藏或重写），ref 和 out 被视为签名的组成部分，并且互不匹配。（此限制使 C# 程序能够方便地进行转换，以便能在公共语言基础结构 (CLI) 上运行，CLI 并未提供任何方式来定义仅通过 ref 和 out 就能加以区分的方法。）

下面的示例演示一组重载方法声明及其签名。

```
interface ITest
{
   void F();                  // F()
   void F(int x);             // F(int)
   void F(ref int x);         // F(ref int)
   void F(out int x);         // F(out int)     error
   void F(int x, int y);      // F(int, int)
   int F(string s);           // F(string)
```

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

请注意，任何 ref 和 out 参数修饰符（第 10.6.1 节）都是签名的一部分。因此，F(int) 和 F(ref int) 是唯一的签名。但是，F(ref int) 和 F(out int) 不能在同一个接口中声明，因为它们的签名仅 ref 和 out 不同。此外，返回类型和 params 修饰符不是签名的组成部分，所以不可能仅基于返回类型或是否存在 params 修饰符来实施重载。因此，上面列出的关于方法 F(int) 和 F(params string[]) 的声明会导致编译时错误。

3.7 范围

名称的范围 (scope) 是一个程序文本区域，在其中可以引用由该名称声明的实体，而不对该名称加以限定。范围可以嵌套 (nested)，并且内部范围可以重新声明外部范围中的名称的含义（但这并不会取消第 3.3 节强加的限制，即在嵌套块中不可能声明与它的封闭块中的局部变量同名的局部变量）。因此，我们就说，外部范围中的这个同名的名称在由内部范围覆盖的程序文本区域中是隐藏的 (hidden)，对外部名称只能通过它的限定名才能从内部范围来访问。

- 由 *namespace-member-declaration*（第 9.5 节）所声明的命名空间成员的范围，如果没有其他封闭它的 *namespace-declaration*，则它的范围是整个程序文本。
- *namespace-declaration* 中 *namespace-member-declaration* 所声明的命名空间成员的范围是这样定义的，如果该命名空间成员声明的完全限定名为 N，则其声明的命名空间成员的范围是，完全限定名为 N 或以 N 开头后跟句点的每个 *namespace-declaration* 的 *namespace-body*。
- *extern-alias-directive* 范围定义的名称范围扩展到直接包含其的编译单元或命名空间体的 *using-directives*、*global-attributes* 和 *namespace-member-declarations*。*extern-alias-directive* 不会把任何新成员提供给基础声明空间。换言之，*extern-alias-directive* 不具传递性，它仅影响它在其中出现的编译单元或命名空间体。
- 由 *using-directive*（第 9.4 节）定义或导入的名称的范围扩展到出现 *using-directive* 的 *compilation-unit* 或 *namespace-body* 内的整个 *namespace-member-declarations* 中。*using-directive* 可以使零个或更多个命名空间或者类型名称在特定的 *compilation-unit* 或 *namespace-body* 中可用，但不会把任何新成员提供给基础声明空间。换言之，*using-directive* 不具传递性，它仅影响它在其中出现的 *compilation-unit* 或 *namespace-body*。
- 由 *class-declaration*（第 10.1 节）的 *type-parameter-list* 声明的类型形参的范围是该 *class-declaration* 的 *class-base*、*type-parameter-constraints-clauses* 和 *class-body*。
- 由 *struct-declaration*（第 11.1 节）的 *type-parameter-list* 声明的类型形参的范围是该 *struct-declaration* 的 *struct-interfaces*、*type-parameter-constraints-clauses* 和 *struct-body*。
- 由 *interface-declaration*（第 13.1 节）的 *type-parameter-list* 声明的类型形参的范围是该 *interface-declaration* 的 *interface-base*、*type-parameter-constraints-clauses* 和 *interface-body*。
- 由 *delegate-declaration*（第 15.1 节）的 *type-parameter-list* 声明的类型形参的范围是该 *delegate-declaration* 的 *return-type*、*formal-parameter-list* 和 *type-parameter-constraints-clauses*。
- 由 *class-member-declaration*（第 10.1.6 节）所声明的成员范围是该声明所在的那个 *class-body*。此外，类成员的范围扩展到该成员的可访问域（第 3.5.2 节）中包含的那些派生类的 *class-body*。
- 由 *struct-member-declaration*（第 11.2 节）声明的成员范围是该声明所在的 *struct-body*。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 由 *enum-member-declaration* （第 14.3 节）声明的成员范围是该声明所在的 *enum-body*。
- 在 *method-declaration*（第 10.6 节）中声明的形参范围是该 *method-declaration* 的 *method-body*。
- 在 *indexer-declaration*（第 10.9 节）中声明的形参范围是该 *indexer-declaration* 的 *accessor-declarations*。
- 在 *operator-declaration*（第 10.10 节）中声明的形参范围是该 *operator-declaration* 的 *block*。
- 在 *constructor-declaration*（第 10.11 节）中声明的形参范围是该 *constructor-declaration* 的 *constructor-initializer* 和 *block*。
- 在 *lambda-expression*（第 7.15 节）中声明的形参范围是该 *lambda-expression* 的 *lambda-expression-body*。
- 在 *anonymous-method-expression*（第 7.15 节）中声明的形参范围为该 *anonymous-method-expression* 的 *block*。
- 在 *labeled-statement*（第 8.4 节）中声明的标签范围是该声明所在的 *block*。
- 在 *local-variable-declaration*（第 8.5.1 节）中声明的局部变量范围是该声明所在的块。
- 在 `switch` 语句（第 8.7.2 节）的 *switch-block* 中声明的局部变量范围是该 *switch-block*。
- 在 `for` 语句（第 8.8.3 节）的 *for-initializer* 中声明的局部变量范围是该 `for` 语句的 *for-initializer*、*for-condition*、*for-iterator* 以及所包含的 *statement*。
- 在 *local-constant-declaration*（第 8.5.2 节）中声明的局部常量范围是该声明所在的块。在某局部常量 *constant-declarator* 之前的文本位置中引用该局部常量是编译时错误。
- 作为 *foreach-statement*、*using-statement*、*lock-statement* 或 *query-expression* 一部分声明的变量的范围由给定构造的扩展确定。

在命名空间、类、结构或枚举成员的范围内，可以在位于该成员的声明之前的文本位置引用该成员。例如

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

这里，`F` 在声明 `i` 之前引用它是有效的。

在局部变量的范围内，在位于该局部变量的 *local-variable-declarator* 之前的文本位置引用该局部变量是编译时错误。例如

```
class A
{
   int i = 0;
   void F() {
      i = 1;              // Error, use precedes declaration
      int i;
      i = 2;
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

在上面的 F 方法中，第一次明确给 i 赋值时，并未引用在外部范围声明的字段。相反，它所引用的是局部变量 i，这会导致编译时错误，因为它在文本上位于该变量的声明之前。在方法 G 中，在 j 的声明初始值设定项中使用 j 是有效的，因为并未在 *local-variable-declarator* 之前使用。在方法 H 中，后面的 *local-variable-declarator* 正确引用在同一 *local-variable-declaration* 内的前面的 *local-variable-declarator* 中声明的局部变量。

局部变量的范围规则旨在保证表达式上下文中使用的名称的含义在块中总是相同。如果局部变量的范围仅从它的声明扩展到块的结尾，则在上面的示例中，第一次赋值将会分配给实例变量，第二次赋值将会分配给局部变量，如果后来重新排列块的语句，则可能会导致编译时错误。

块中名称的含义可能因该名称的使用上下文而异。在下面的示例中

```
using System;

class A {}

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

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

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

名称 A 在表达式上下文中用来引用局部变量 A，在类型上下文中用来引用类 A。

3.7.1 名称隐藏

实体的范围通常比该实体的声明空间包含更多的程序文本。具体而言，实体的范围可能包含一些声明，它们会引入一些新的声明空间，其中可能含有与该实体同名的新实体。这类声明导致原始的实体变为隐藏的 (hidden)。相反，当实体不是隐藏的时，就说它是可见的 (visible)。

当范围之间相重叠（或通过嵌套重叠，或通过继承重叠）时会发生名称隐藏。以下各节介绍这两种隐藏类型的特性。

3.7.1.1 通过嵌套隐藏

以下各项活动会导致发生通过嵌套的名称隐藏：在命名空间内嵌套其他命名空间或类型；在类或结构中的嵌套类型；声明形参和局部变量。

在下面的示例中

```
class A
{
   int i = 0;

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

在方法 F 中，实例变量 i 被局部变量 i 隐藏，但在方法 G 中，i 仍引用该实例变量。

当内部范围中的名称隐藏外部范围中的名称时，它隐藏该名称的所有重载匹配项。在下面的示例中

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

   static void F(string s) {}

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

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

由于 F 的所有外部匹配项都被内部声明隐藏，因此调用 F(1) 将调用在 Inner 中声明的 F。由于同样的原因，调用 F("Hello") 将导致编译时错误。

3.7.1.2 通过继承隐藏

当类或结构重新声明从基类继承的名称时，会发生通过继承的名称隐藏。这种类型的名称隐藏采取下列形式之一：

- 类或结构中引入的常量、字段、属性、事件或类型会把所有同名的基类成员隐藏起来。
- 类或结构中引入的方法将隐藏所有同名的非方法基类成员以及所有具有相同签名的基类方法（第3.6节）。
- 类或结构中引入的索引器将隐藏具有相同签名的所有基类索引器（第3.6 节）。

管理运算符声明（第 10.10 节）的规则使派生类不可能声明与基类中的运算符具有相同签名的运算符。因此，运算符从不相互隐藏。

与隐藏外部范围中的名称相反，隐藏继承范围中的可访问名称会导致发出警告。在下面的示例中

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

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

在 Derived 中声明 F 会导致报告一个警告。准确地说，隐藏继承的名称不是一个错误，因为这会限制基类按自身情况进行改进。例如，由于更高版本的 Base 引入了该类的早期版本中不存在的 F 方法，可能会发生上述情况。如果上述情况是一个错误，当基类属于一个单独进行版本控制的类库时，对该基类的任何 更改都有可能导致它的派生类变得无效。

通过使用 new 修饰符可以消除因隐藏继承的名称导致的警告：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

new 修饰符指示 Derived 中的 F 是“新的”，并且实际上是有意隐藏继承的成员。

在声明一个新成员时，仅在该新成员的范围内隐藏被继承的成员。

```
class Base
{
   public static void F() {}
}
class Derived: Base
{
   new private static void F() {}   // Hides Base.F in Derived only
}
class MoreDerived: Derived
{
   static void G() { F(); }         // Invokes Base.F
}
```

在上面的示例中，Derived 中的 F 声明将隐藏从 Base 继承的 F，但由于 Derived 中的新 F 具有私有访问权限，它的范围不会扩展到 MoreDerived。因此，MoreDerived.G 中的调用 F() 是有效的，它将调用 Base.F。

3.8 命名空间和类型名称

C# 程序中的若干上下文要求指定 *namespace-name* 或 *type-name*。

namespace-name:
namespace-or-type-name

type-name:
namespace-or-type-name

namespace-or-type-name:
identifier type-argument-list$_{opt}$
namespace-or-type-name . *identifier type-argument-list*$_{opt}$
qualified-alias-member

namespace-name 是引用一个命名空间的 *namespace-or-type-name*。根据如下所述的解析过程，*namespace-name* 的 *namespace-or-type-name* 必须引用一个命名空间，否则将发生编译时错误。*namespace-name* 中不能存在任何类型实参（第 4.4.1 节），只有类型才能具有类型实参。

type-name 是引用一个类型的 *namespace-or-type-name*。根据如下所述的解析过程，*type-name* 的 *namespace-or-type-name* 必须引用一个类型，否则将发生编译时错误。

如果 *namespace-or-type-name* 是 qualified-alias-member，则其含义如第 9.7 节中所述。否则，*namespace-or-type-name* 具有下列四种形式之一：

- I
- I<A_1, ..., A_K>

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- `N.I`
- `N.I<`A_1`, ..., `A_K`>`

其中 `I` 是单个标识符，`N` 是 *namespace-or-type-name*，`<`A_1`, ..., `A_K`>` 是可选的 *type-argument-list*。如果未指定 *type-argument-list* 时，则可将 `K` 视为零。

namespace-or-type-name 的含义按下述步骤确定：

- 如果 *namespace-or-type-name* 的形式为 `I` 或 `I<`A_1`, ..., `A_K`>`:
  - 如果 `K` 为零，*namespace-or-type-name* 出现在泛型方法声明（第 10.6 节）中，且该声明包含名为 `I` 的类型形参（第 10.1.3 节），则 *namespace-or-type-name* 引用该类型形参。
  - 否则，如果 *namespace-or-type-name* 出现在类型声明中，则对于每个实例类型 `T`（第 10.3.1 节），从该类型声明的实例类型开始，对每个封闭类或结构声明(如果有)的实例类型继续如下过程：
    - 如果 `K` 为零，并且 `T` 的声明包含名为 `I` 的类型形参，则 *namespace-or-type-name* 引用该类型形参。
    - 否则，如果 *namespace-or-type-name* 出现在该类型声明的体中，且 `T` 或其任一基类型包含具有名称 `I` 和 `K` 个类型形参的嵌套可访问类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。如果存在多个这样的类型，则选择在派生程度较大的类型中声明的类型。请注意，在确定 *namespace-or-type-name* 的含义时，将忽略非类型成员（常量、字段、方法、属性、索引器、运算符、实例构造函数、析构函数和静态构造函数）和具有不同数目的类型形参的类型成员。
  - 如果当时前面的步骤不成功，则对于每个命名空间 `N`，从出现 *namespace-or-type-name* 的命名空间开始，继续到每个封闭命名空间（如果有）且到全局命名空间结束，对下列步骤进行计算直到找到实体：
    - 如果 `K` 为零，并且 `I` 为 `N` 中的命名空间的名称，则：
      - 如果出现 *namespace-or-type-name* 的位置包含在 `N` 的命名空间声明中，并且该命名空间声明包含将名称 `I` 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *namespace-or-type-name* 是不明确的，并将发生编译时错误。
      - 否则，*namespace-or-type-name* 引用 `N` 中名为 `I` 的命名空间。
    - 否则，如果 `N` 包含一个具有名称 `I` 且有 `K` 个类型形参的可访问类型，则：
      - 如果 `K` 为零，并且出现 *namespace-or-type-name* 的位置包含在 `N` 的命名空间声明中，并且该命名空间声明包含将名称 `I` 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *namespace-or-type-name* 是不明确的，并将发生编译时错误。
      - 否则，*namespace-or-type-name* 引用利用给定类型实参构造的该类型。
    - 否则，如果出现 *namespace-or-type-name* 的位置包含在 `N` 的命名空间声明中：
      - 如果 `K` 为零，并且该命名空间声明包含一个将名称 `I` 与一个导入的命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *namespace-or-type-name* 引用该命名空间或类型。
      - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间恰好包含一个具有名称 `I` 且有 `K` 个类型形参的类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

  - 否则，如果该命名空间声明的 *using-namespace-directives* 导入的命名空间包含多个具有名称 `I` 且有 `K` 个类型形参的类型，则 *namespace-or-type-name* 是不明确的，并将导致发生错误。
- 否则，*namespace-or-type-name* 未定义，并将导致发生编译时错误。

- 否则，*namespace-or-type-name* 的形式为 `N.I` 或 `N.I<A1, ..., AK>`.`N` 首先解析为 *namespace-or-type-name*。如果对 `N` 的解析不成功，则发生编译时错误。否则，`N.I` 或 `N.I<A1, ..., AK>` 按如下方式进行解析：
  - 如果 `K` 为零，`N` 引用一个命名空间，并且 `N` 包含名为 `I` 的嵌套命名空间，则 *namespace-or-type-name* 引用该嵌套命名空间。
  - 否则，如果 `N` 引用一个命名空间，并且 `N` 包含一个具有名称 `I` 且有 `K` 个类型形参的可访问类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。
  - 否则，如果 `N` 引用一个（可能是构造的）类或结构类型，并且 `N` 或其任一基类包含一个具有名称 `I` 且有 `K` 个类型形参的嵌套可访问类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。如果存在多个这样的类型，则选择在派生程度较大的类型中声明的类型。请注意，如果要将 `N.I` 的含义确定为解析 `N` 的基类指定的一部分，则将 `N` 的直接基类视为对象（第 10.1.4.1 节）。
  - 否则，`N.I` 是无效的 *namespace-or-type-name* 并将发生编译时错误。

仅当下列条件成立时才允许 *namespace-or-type-name* 引用静态类（第 10.1.1.3 节）

- *namespace-or-type-name* 是 `T.I` 形式的 *namespace-or-type-name* 中的 `T`，或者
- *namespace-or-type-name* 是 `typeof(T)` 形式的 *typeof-expression*（第 7.5.11 节）中的 `T`。

3.8.1 完全限定名

每个命名空间和类型都具有一个完全限定名 (fully qualified name)，该名称在所有其他命名空间或类型中唯一标识该命名空间或类型。命名空间或类型 `N` 的完全限定名按下面这样确定：

- 如果 `N` 是全局命名空间的成员，则它的完全限定名为 `N`。
- 否则，它的完全限定名为 `S.N`，其中 `S` 是声明了 `N` 的命名空间或类型的完全限定名。

换言之，`N` 的完全限定名是从全局命名空间开始通向 `N` 的标识符的完整分层路径。由于命名空间或类型的每个成员都必须具有唯一的名称，因此，如果将这些成员名称置于命名空间或类型的完全限定名之后，这样构成的成员完全限定名一定符合唯一性。

下面的示例演示了若干命名空间和类型声明及其关联的完全限定名。

```
class A {}           // A
namespace X          // X
{
   class B           // X.B
   {
      class C {}     // X.B.C
   }
   namespace Y       // X.Y
   {
      class D {}     // X.Y.D
   }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

3.9 自动内存管理

C# 使用自动内存管理，它使开发人员不再需要以手动方式分配和释放对象占用的内存。自动内存管理策略由垃圾回收器 (garbage collector) 实现。一个对象的内存管理生存周期如下所示：

1. 当创建对象时，为其分配内存，运行构造函数，将该对象被视为活对象。
2. 在后续执行过程中，如果不会再访问该对象或它的任何部分（除了运行它的析构函数），则将该对象视为不再使用，可以销毁。C# 编译器和垃圾回收器可以通过分析代码，确定哪些对象引用可能在将来被使用。例如，如果范围内的某个局部变量是现有的关于此对象的唯一引用，但在当前执行点之后的任何后续执行过程中，该局部变量都不会再被引用，那么垃圾回收器可以（但不是必须）认为该对象不再被使用。
3. 一旦对象符合销毁条件，在稍后某个时间将运行该对象的析构函数（第 10.13 节）（如果有）。在普通情况下，对象的析构函数仅运行一次，但特定于实现的 API 可允许忽略此行为。
4. 一旦运行对象的析构函数，如果该对象或它的任何部分无法由任何可能的执行继续（包括运行析构函数）访问，则该对象被视为不可访问，可以回收。
5. 最后，在对象变得符合回收条件后，垃圾回收器将释放与该对象关联的内存。

垃圾回收器维护对象的使用信息，并利用此信息做出内存管理决定，如在内存中的何处安排一个新创建的对象、何时重定位对象以及对象何时不再被使用或不可访问。

与其他假定存在垃圾回收器的语言一样，C# 也旨在使垃圾回收器可以实现广泛的内存管理策略。例如，C# 并不要求一定要运行析构函数，不要求对象一符合条件就被回收，也不要求析构函数以任何特定的顺序或在任何特定的线程上运行。

垃圾回收器的行为在某种程度上可通过类 `System.GC` 的静态方法来控制。该类可用于请求执行一次回收操作、运行（或不运行）析构函数，等等。

由于垃圾回收器在决定何时回收对象和运行析构函数方面可以有很大的选择范围，它的一个符合条件的实现所产生的输出可能与下面的代码所显示的不同。程序

```
using System;

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

class B
{
   object Ref;

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

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

创建类 A 的一个实例和类 B 的一个实例。当给变量 b 赋值 null 后，这些对象变得符合垃圾回收条件，这是因为从此往后，任何用户编写的代码不可能再访问这些对象。输出可以为

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

或

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

这是因为该语言对于对象的垃圾回收顺序没有强加约束。

在一些微妙的情况下，“符合销毁条件”和“符合回收条件”之间的区别会很重要。例如，

```
using System;

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

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

class B
{
   public A Ref;

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

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

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

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

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

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

请注意，虽然 A 的实例没有使用，并且 A 的析构函数已被运行过了，但仍可能从其他析构函数调用 A 的方法（此例中是指 F）。还请注意，运行析构函数可能导致对象再次从主干程序中变得可用。在此例中，运行 B 的析构函数导致了先前没有被使用的 A 的实例变得可从当前有效的引用 Test.RefA 访问。调用 WaitForPendingFinalizers 后，B 的实例符合回收条件，但由于引用 Test.RefA 的缘故，A 的实例不符合回收条件。

为了避免混淆和意外的行为，好的做法通常是让析构函数只对存储在它们对象本身字段中的数据执行清理，而不对它所引用的其他对象或静态字段执行任何操作。

另一种使用析构函数的方法是允许类实现 System.IDisposable 接口。这样的话，对象的客户端就可以确定何时释放该对象的资源，通常是通过在 using 语句（第 8.13 节）中以资源形式访问该对象。

3.10 执行顺序

C# 程序执行时，在临界执行点保留每个执行线程的副作用。副作用 (side effect) 副作用 定义为对可变字段的读取或写入、对非可变变量的写入、对外部资源的写入以及异常的引发。临界执行点（这些副作用的顺序必须保存在其中）是指下列各活动：引用一些可变字段（第 10.5.3 节）；引用 lock 语句（第 8.12 节）；引用线程的创建与终止。执行环境可以随便更改 C# 程序的执行顺序，但受下列约束限制：

- 在执行线程中需保持数据依赖性。就是说，在计算每个变量的值时，就好像线程中的所有语句都是按原始程序顺序执行的。
- 保留初始化的排序规则（第 10.5.4 节和第 10.5.5 节）。
- 对于不稳定读写（第 10.5.3 节），副作用的顺序需保持不变。此外，执行环境甚至可以不需要计算一个表达式的各个部分，如果它能推断出表达式的值是“不会被使用的”而且不会产生有效的副作用（包括由调用方法或访问不稳定字段导致的任何副作用）。当程序执行被异步事件（例如其他线程引发的异常）中断时，它不保证可观察到的副作用以原有的程序顺序出现。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

4. 类型

C# 语言的类型划分为两大类：值类型 (Value type) 和引用类型 (reference type)。值类型和引用类型都可以为泛型类型 (generic type)，泛型类型采用一个或多个类型参数。类型参数可以指定值类型和引用类型。

type:
value-type
reference-type
type-parameter

第三种类型是指针，只能用在不安全代码中。第 18.2 节对此做了进一步的探讨。

值类型与引用类型的不同之处在于：值类型的变量直接包含其数据，而引用类型的变量存储对其数据的引用 (reference)，后者称为对象 (object)。对于引用类型，两个变量可能引用同一个对象，因此对一个变量的操作可能影响另一个变量所引用的对象。对于值类型，每个变量都有自己的数据副本，对一个变量的操作不可能影响另一个变量。

C# 的类型系统是统一的，因此任何类型的值都可以按对象处理。C# 中的每个类型直接或间接地从 `object` 类类型派生，而 `object` 是所有类型的最终基类。引用类型的值都被视为 `object` 类型，被简单地当作对象来处理。值类型的值则通过对其执行装箱和拆箱操作（第 4.3 节）按对象处理。

4.1 值类型

一个值类型或是结构类型，或是枚举类型。C# 提供称为简单类型 (simple type) 的预定义结构类型集。简单类型通过保留字标识。

value-type:
struct-type
enum-type

struct-type:
type-name
simple-type
nullable-type

simple-type:
numeric-type
`bool`

numeric-type:
integral-type
floating-point-type
`decimal`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

integral-type:
    sbyte
    byte
    short
    ushort
    int
    uint
    long
    ulong
    char

floating-point-type:
    float
    double

nullable-type:
    non-nullable-value-type ?

non-nullable-value-type:
    type

enum-type:
    type-name

与引用类型的变量不同的是，仅当该值类型是可以为null 的类型时，值类型的变量才可包含 null 值。对于每个不可以为 null 的值类型，都存在一个对应的可以为null 的值类型，该类型表示相同的值集加上 null 值。

对值类型变量赋值时，会创建所赋的值的一个副本。这不同于引用类型的变量赋值，引用类型的变量赋值复制的是引用而不是由引用标识的对象。

4.1.1 System.ValueType 类型

所有值类型均从类 System.ValueType 隐式继承，后者又从类 object 继承。任何类型都不可能从值类型派生，因此，所有值类型都是隐式密封的（第 10.1.1.2 节）。

请注意，System.ValueType 本身不是 *value-type*。而是 *class-type*，所有 *value-type* 都从它自动派生。

4.1.2 默认构造函数

所有值类型都隐式声明一个称为默认构造函数 (default constructor) 的公共无参数实例构造函数。默认构造函数返回一个零初始化实例，它就是该值类型的默认值 (default value)：

- 对于所有 *simple-type*，默认值是由所有位都置零的位模式产生的值：
  - 对于 sbyte、byte、short、ushort、int、uint、long 和 ulong，默认值为 0。
  - 对于 char，默认值为 '\x0000'。
  - 对于 float，默认值为 0.0f。
  - 对于 double，默认值为 0.0d。
  - 对于 decimal，默认值为 0.0m。
  - 对于 bool，默认值为 false。
- 对于 *enum-type* E，默认值为 0，该值被转换为类型 E。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 对于 *struct-type*，默认值是通过将所有值类型字段设置为它们的默认值并将所有引用类型字段设置为 null 而产生的值。
- 对于 *nullable-type*，默认值是一个其 HasValue 属性为 false 且 Value 属性未定义的实例。默认值也称为可以为 null 的类型的 null 值 (null value)。

与任何其他实例构造函数一样，值类型的默认构造函数也是用 new 运算符调用的。出于效率原因，实际上，不必故意调用它的构造函数。在下面的示例中，变量 i 和 j 都被初始化为零。

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

由于每个值类型都隐式地具有一个公共无形参实例构造函数，因此，一个结构类型中不可能包含一个关于无形参构造函数的显式声明。但允许结构类型声明参数化实例构造函数（第 11.3.8 节）。

4.1.3 结构类型

结构类型是一种值类型，它可以声明常量、字段、方法、属性、索引器、运算符、实例构造函数、静态构造函数和嵌套类型。结构类型的声明在第 11.1 节中说明。

4.1.4 简单类型

C# 提供称为简单类型 (simple type) 的预定义结构类型集。简单类型通过保留字标识，而这些保留字只是 System 命名空间中预定义结构类型的别名，详见下表。

保留字	别名类型
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

由于简单类型是结构类型的别名，所以每个简单类型都具有成员。例如，int 具有在 System.Int32 中声明的成员以及从 System.Object 继承的成员，并且允许使用下面的语句：

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

简单类型与其他结构类型的不同之处在于，简单类型允许某些附加的操作：

- 大多数简单类型允许通过编写 *literals*（第2.4.4 节）来创建值。例如，123 是类型为 int 的文本，'a' 是类型为 char 的文本。C# 没有普遍地为结构类型设置类似的以文本创建值的规则，所以其他结构类型的非默认值最终总是通过这些结构类型的实例构造函数来创建的。
- 当表达式的操作数都是简单类型常量时，编译器可以在编译时计算表达式。这样的表达式称为 *constant-expression*（第 7.19 节）。涉及其他结构类型所定义的运算符的表达式不被视为常量表达式。
- 通过 const 声明可以声明简单类型（第 10.4 节）的常量。常量不可能属于其他结构类型，但 static readonly 字段提供了类似的效果。
- 涉及简单类型的转换可以参与由其他结构类型定义的转换运算符的计算，但用户定义的转换运算符永远不能参与其他用户定义运算符的计算（第6.4.3 节）。

4.1.5 整型

C# 支持 9 种整型：sbyte、byte、short、ushort、int、uint、long、ulong 和 char。整型具有以下所列的大小和取值范围：

- sbyte 类型表示有符号 8 位整数，其值介于 –128 和 127 之间。
- byte 类型表示无符号 8 位整数，其值介于 0 和 255 之间。
- short 类型表示有符号 16 位整数，其值介于 –32768 和 32767 之间。
- ushort 类型表示无符号 16 位整数，其值介于 0 和 65535 之间。
- int 类型表示有符号 32 位整数，其值介于 –2147483648 和 2147483647 之间。
- uint 类型表示无符号 32 位整数，其值介于 0 和 4294967295 之间。
- long 类型表示有符号 64 位整数，其值介于 –9223372036854775808 和 9223372036854775807 之间。
- ulong 类型表示无符号 64 位整数，其值介于 0 和 18446744073709551615 之间。
- char 类型表示无符号 16 位整数，其值介于 0 和 65535 之间。char 类型的可能值集与 Unicode 字符集相对应。虽然 char 的表示形式与 ushort 相同，但是可以对一种类型进行的所有计算并非都可以对另一种类型执行。

整型一元运算符和二元运算符总是对有符号 32 位精度、无符号的 32 位精度、有符号 64 位精度或无符号 64 位精度进行计算：

- 对于一元运算符 +，操作数转换为类型 T，其中 T 是 int、uint、long 和 ulong 中第一个可以完全表示操作数的所有可能值的类型。然后用 T 类型的精度执行运算，结果的类型是 T 类型。
- 对于一元运算符 –，操作数转换为类型 T，其中 T 是 int 和 long 中第一个可以完全表示操作数的所有可能值的类型。然后用 T 类型的精度执行运算，结果的类型是 T 类型。一元运算符 – 不能应用于类型 ulong 的操作数。
- 对于 +、–、*、/、%、&、^、|、==、!=、>、<、>= 和 <= 二元运算符，操作数转换为类型 T，其中 T 是 int、uint、long 和 ulong 中第一个可以完全表示两个操作数的所有可能值的类型。然后用 T 类型的精度执行运算，运算的结果的类型也属于 T（对于关系运算符为 bool）。对于二元运算符，不允许一个操作数为 long 类型而另一个操作数为 ulong 类型。
- 对于二元运算符 << 和 >>，左操作数转换为类型 T，其中 T 是 int、uint、long 和 ulong 中第一个可以完全表示操作数的所有可能值的类型。然后用 T 类型的精度执行运算，结果的类型是 T 类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

char 类型归类为整型类型，但它在以下两个方面不同于其他整型：

- 不存在从其他类型到 char 类型的隐式转换。具体而言，即使 sbyte、byte 和 ushort 类型具有完全可以用 char 类型来表示的值范围，也不存在从 sbyte、byte 或 ushort 到 char 的隐式转换。
- char 类型的常量必须写成 *character-literal*l 或带有强制转换为类型 char 的 *integer-literal*。例如，(char)10 与 '\x000A' 相同。

checked 和 unchecked 运算符和语句用于控制整型算术运算和转换（第 7.6.12 节）的溢出检查。在 checked 上下文中，溢出产生编译时错误或导致引发 System.OverflowException。在 unchecked 上下文中将忽略溢出，任何与目标类型不匹配的高序位都被放弃。

4.1.6 浮点型

C# 支持两种浮点型：float 和 double。float 和 double 类型用 32 位单精度和 64 位双精度 IEEE 754 格式来表示，这些格式提供以下几组值：

- 正零和负零。大多数情况下，正零和负零的行为与简单的值零相同，但某些运算会区别对待此两种零（第 7.8.2 节）。
- 正无穷大和负无穷大。无穷大是由非零数字被零除这样的运算产生的。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 ***(Not-a-Number)***值，常缩写为 NaN。NaN 是由无效的浮点运算（如零被零除）产生的。
- 以 $s \times m \times 2^e$ 形式表示的非零值的有限集，其中 s 为 1 或 −1，m 和 e 由特殊的浮点类型确定：对于 float，为 $0 < m < 2^{24}$ 并且 $-149 \leqslant e \leqslant 104$；对于 double，为 $0 < m < 2^{53}$ 并且 $-1075 \leqslant e \leqslant 970$。非标准化的浮点数被视为有效非零值。

float 类型可表示精度为 7 位、在大约 1.5×10^{-45} 到 3.4×10^{38} 的范围内的值。

double 类型可表示精度为 15-16 位、在大约 1.5×10^{-324} 到 1.7×10^{308} 的范围内的值。

如果二元运算符的一个操作数为浮点型，则另一个操作数必须为整型或浮点型，并且运算按下面这样计算：

- 如果一个操作数为整型，则该操作数转换为与另一个操作数的类型相同的浮点型。
- 然后，如果任一操作数的类型为 double，则另一个操作数转换为 double。至少用 double 范围和精度执行运算，结果的类型为 double（对于关系运算符则为 bool）。
- 否则，至少用 float 范围和精度执行运算，结果的类型为 float（对于关系运算符则为 bool）。

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

- 如果浮点运算的结果对于目标格式太小，则运算结果变成正零或负零。
- 如果浮点运算的结果对于目标格式太大，则运算结果变成正无穷大或负无穷大。
- 如果浮点运算无效，则运算的结果变成 NaN。
- 如果浮点运算的一个或两个操作数为 NaN，则运算的结果变成 NaN。

可以用比运算的结果类型更高的精度来执行浮点运算。例如，某些硬件结构支持比 double 类型具有更大的范围和精度的“extended”或“long double”浮点型，并隐式地使用这种更高精度类型执行所有浮点运算。只有性能开销过大，才能使这样的硬件结构用“较低”的精度执行浮点运算。C# 采取的是允

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

许将更高的精度类型用于所有浮点运算，而不是强求执行规定的精度，造成同时损失性能和精度。除了传递更精确的结果外，这样做很少会产生任何可察觉的效果。但是，在 `x * y / z` 形式的表达式中，如果其中的乘法会产生超出 `double` 范围的结果，而后面的除法使临时结果返回到 `double` 范围内，则以更大范围的格式去计算该表达式，可能会产生有限值的结果（本来应是无穷大）。若要强制将浮点类型的值转换为其类型的精确精度，则可使用显式强制转换。

4.1.7 decimal 类型

`decimal` 类型是 128 位的数据类型，适合用于财务计算和货币计算。`decimal` 类型可以表示具有 28 或 29 个有效数字、从 1.0×10^{-28} 到大约 7.9×10^{28} 范围内的值。

`decimal` 类型的有限值集的形式为 $^{s} \times c \times 10^{-e}$，其中符号 s 是 0 或 1，系数 c 由 $0 \leqslant c < c < 2^{96}$ 给定，小数位数 e 满足 $0 \leqslant e \leqslant 28$。`decimal` 类型不支持有符号的零、无穷大或 NaN。`decimal` 可用一个以 10 的幂表示的 96 位整数来表示。对于绝对值小于 `1.0m` 的 `decimal`，它的值最多精确到第 28 位小数。对于绝对值大于或等于 `1.0m` 的 `decimal`，它的值精确到小数点后第 28 或 29 位。与 `float` 和 `double` 数据类型相反，十进制小数数字（如 0.1）可以精确地用 `decimal` 表示形式来表示。在 `float` 和 `double` 表示形式中，这类数字通常变成无限小数，使这些表示形式更容易发生舍入错误。

如果二元运算符的一个操作数为 `decimal` 类型，则另一个操作数必须为整型或 `decimal` 类型。如果存在一个整型操作数，它将在执行运算前转换为 `decimal`。

`decimal` 类型值的运算结果是这样得出的：先计算一个精确结果（按每个运算符的定义保留小数位数），然后舍入以适合表示形式。结果舍入到最接近的可表示值，当结果同样地接近于两个可表示值时，舍入到最小有效位数位置中为偶数的值（这称为“银行家舍入法”）。零结果总是包含符号 0 和小数位数 0。

如果十进制算术运算产生一个绝对值小于或等于 5×10^{-29} 的值，则运算结果变为零。如果 `decimal` 算术运算生成的结果对于 `decimal` 格式来说太大，则将引发 `System.OverflowException`。

与浮点型相比，`decimal` 类型具有较高的精度，但取值范围较小。因此，从浮点型到 `decimal` 的转换可能会产生溢出异常，而从 `decimal` 到浮点型的转换则可能导致精度损失。由于这些原因，在浮点型和 `decimal` 之间不存在隐式转换，如果没有显式地标出强制转换，就不可能在同一表达式中同时使用浮点操作数和 `decimal` 操作数。

4.1.8 bool 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` 和其他类型之间不存在标准转换。具体而言，`bool` 类型与整型截然不同，不能用 `bool` 值代替整数值，反之亦然。

在 C 和 C++ 语言中，零整数或浮点值或 null 指针可以转换为布尔值 falsefalse 非零整数或浮点值或非 null 指针可以转换为布尔值 `true`。在 C# 中，这种转换是通过显式地将整数或浮点值与零进行比较，或者显式地将对象引用与 `null` 进行比较来完成的。

4.1.9 枚举类型

枚举类型是具有命名常量的独特的类型。每个枚举类型都有一个基础类型，该基础类型必须为 `byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`long` 或 `ulong`。枚举类型的值集和它的基础类型的值集相同。枚举类型的值并不只限于那些命名常量的值。枚举类型是通过枚举声明（第 14.1 节）定义的。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

4.1.10 可以为 null 的类型

可以为 null 的类型可以表示其基础类型(underlying type) 的所有值和一个额外的 null 值。可以为 null 的类型写作 `T?`，其中 `T` 是基础类型。此语法是 `System.Nullable<T>` 的简写形式，这两种形式可以互换使用。

相反，不可以为 null 的值类型 (non-nullable value type) 可以是除 `System.Nullable<T>` 及其简写形式 `T?`（对于任何类型的 `T`)）之外的任何值类型，加上约束为不可以为 null 的值类型的任何类型参数（即具有 `struct` 约束的任何类型参数）。`System.Nullable<T>` 类型指定 `T` (的值类型约束（第 10.1.5 节)，这意味着可以为 null 的类型的基础类型可以是任何不可以为null 的值类型。可以为 null 的类型的基础类型不能是可以为 null 的类型或引用类型。例如，`int??` 和 `string?` 是无效类型。

可以为 null 的类型 `T?` 的实例有两个公共只读属性：

- 类型 `bool` 的 `HasValue` 的属性
- 类型 `T` 的 `Value` 的属性

`HasValue` 为 true 的实例称为非 null。非 null 实例包含一个已知值，可通过 `Value` 返回该值。

`HasValue` 为 false 的实例称为 null。null 实例有一个不确定的值。尝试读取 null 实例的 `Value` 将导致引发 `System.InvalidOperationException`。访问可以为 null 的实例的 `Value` 属性的过程称作解包 (unwrapping)。

除了默认构造函数之外，每个可以为 null 的类型 `T?` 都有一个具有类型为 `T` 的单个实参的公共构造函数。例如，给定一个类型为 `T` 的值 `x`，调用形如

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

的构造函数将创建 `T?` 的非 null 实例，其 `Value` 属性为 `x`。为一个给定值创建可以为null 的类型的非 null 实例的过程称作包装 (wrapping)。

从 `null` 文本转换为 `T?`（第 6.1.5 节）以及从 `T` 转换为 `T?`（第 6.1.4 节）可使用隐式转换。

4.2 引用类型

引用类型是类类型、接口类型、数组类型或委托类型。

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

class-type:
    type-name
    `object`
    `string`

interface-type:
    type-name

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

non-array-type:
    type

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

rank-specifier:
[*dim-separators*$_{opt}$]

dim-separators:
,
dim-separators ,

delegate-type:
type-name

引用类型值是对该类型的某个实例 (instance) 的一个引用，后者称为对象 (object)。null 值比较特别，它兼容于所有引用类型，用来表示“没有被引用的实例”。

4.2.1 类类型

类类型定义包含数据成员、函数成员和嵌套类型的数据结构，其中数据成员包括常量和字段，函数成员包括方法、属性、事件、索引器、运算符、实例构造函数、析构函数和静态构造函数。类类型支持继承，继承是派生类可用来扩展和专门化基类的一种机制。类类型的实例是用 *object-creation-expressions*（第 7.6.10.1 节）创建的。

有关类类型的介绍详见第 10 章。

某些预定义类类型在 C# 语言中有特殊含义，如下表所示。

类类型	说明
System.Object	所有其他类型的最终基类。请参见第 4.2.2 节。
System.String	C# 语言的字符串类型。请参见第 4.2.4 节。
System.ValueType	所有值类型的基类。请参见第 4.1.1 节。
System.Enum	所有枚举类型的基类。请参见第 14 节。
System.Array	所有数组类型的基类。请参见第 12 节。
System.Delegate	所有委托类型的基类。请参见第 15 节。
System.Exception	所有异常类型的基类。请参见第 16 节。

4.2.2 对象类型

object 类类型是所有其他类型的最终基类。C# 中的每种类型都是直接或间接从 object 类类型派生的。

关键字 object 只是预定义类 System.Object 的别名。

4.2.3 dynamic 类型

dynamic 类型与 object 一样，可以引用任何对象。在将运算符应用于 dynamic 类型的表达式时，其解析会推迟到程序运行时进行。因此，如果运算符不能合法地应用于引用的对象，在编译过程中不会报告任何错误。而是在运行时解析运算符失败时，会引发异常。

在第 4.7 节中进一步介绍了动态类型，在第 7.2.2 节中进一步介绍了动态绑定。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

4.2.4 string 类型

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

string 类型的值可以写为字符串（第 2.4.4.5 节）。

关键字 string 只是预定义类 System.String 的别名。

4.2.5 接口类型

一个接口定义一个协定。实现某接口的类或结构必须遵守该接口定义的协定。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

有关接口类型的介绍详见第 13 章。

4.2.6 数组类型

数组是一种数据结构，它包含可通过计算索引访问的零个或更多个变量。数组中包含的变量（又称数组的元素）具有相同的类型，该类型称为数组的元素类型。

有关数组类型的介绍详见第 12 章。

4.2.7 委托类型

委托是引用一个或多个方法的数据结构。对于实例方法，委托还可引用实例方法对应的对象实例。

在 C 或 C++ 中与委托最接近的是函数指针，但函数指针只能引用静态函数，而委托则既可以引用静态方法，也可以引用实例方法。在后一种情况中，委托不仅存储了一个对该方法入口点的引用，还存储了一个对相应的对象实例的引用，该方法就是通过此对象实例被调用的。

有关委托类型的介绍详见第 15 章。

4.3 装箱和拆箱

装箱和拆箱的概念是 C# 的类型系统的核心。它在 *value-type* 和 *reference-type* 之间架起了一座桥梁，使得任何 *value-type* 的值都可以转换为 object 类型的值，反过来转换也可以。装箱和拆箱使我们能够统一地来考察类型系统，其中任何类型的值最终都可以按对象处理。

4.3.1 装箱转换

装箱转换允许将 *value-type* 隐式转换为 *reference-type*。存在下列装箱转换：

- 从任何 *value-type* 到 object 类型。
- 从任何 *value-type* 到 System.ValueType 类型。
- 从任何 *non-nullable-value-type* 到 *value-type* 实现的任何 *interface-type*。
- 从任何 *nullable-type* 到由 *nullable-type* 的基础类型实现的任何 *interface-type*。
- 从任何 *enum-type* 到类型 System.Enum。
- 从任何具有基础 *enum-type* 的 *nullable-type* 到 System.Enum 类型。

请注意，对类型形参进行隐式转换将以装箱转换的形式执行（如果在运行时它最后从值类型转换到引用类型（第 6.1.10 节））。

将 *non-nullable-value-type* 的一个值装箱包括以下操作：分配一个对象实例，然后将 *non-nullable-value-type* 的值复制到该实例中。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

对 *nullable-type* 的值装箱时，如果该值为 null 值（HasValue 为 false），将产生一个 null 引用；否则将产生对基础值解包和装箱的结果。

最能说明 *non-nullable-value-type* 的值的实际装箱过程的办法是，设想有一个泛型装箱类(boxing class)，其行为与下面声明的类相似：

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

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

T 类型值 v 的装箱过程现在包括执行表达式 new Box<T>(v)) 和将结果实例作为 object 类型的值返回。因此，下面的语句

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

在概念上相当于

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

实际上，像上面这样的 Box<T> 装箱类并不存在，并且装箱值的动态类型也不会真的属于一个类类型。相反，T 类型的装箱值属于动态类型 T，若用 is 运算符来检查动态类型，也仅能引用类型 T。例如，

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

将在控制台上输出字符串“Box contains an int”。

装箱转换隐含着复制一份待装箱的值。这不同于从 *reference-type* 到 object 类型的转换，在后一种转换中，转换后的值继续引用同一实例，只是将它当作派生程度较小的 object 类型而已。例如，给定下面的声明

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

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

则下面的语句

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

将在控制台上输出值 10，因为将 p 赋值给 box 是一个隐式装箱操作，它将复制 p 的值。假如将 Point 声明为 class，则输出值为 20，因为 p 和 box 将引用同一实例。

4.3.2 拆箱转换

取消装箱转换允许将 *reference-type* 显式转换为 *value-type*。存在以下拆箱转换：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 从 object 类型到任何 *value-type*。
- 从 System.ValueType 类型到任何 *value-type*。
- 从任何 *interface-type* 到实现了该 *interface-type* 的任何 *non-nullable-value-type*。
- 从任何 *interface-type* 到其基础类型实现了该 *interface-type* 的任何 *nullable-type*。
- 从 System.Enum 类型到任何 *enum-type*。
- 从 System.Enum 类型到任何具有基础 *enum-type* 的 *nullable-type*。

请注意，到类型形参的显式转换将以取消装箱转换的形式执行（如果在运行时它结束从引用类型到值类型（第 6.2.6 节）的转换）。

对 *non-nullable-value-type* 取消装箱的操作包括下列步骤：首先检查对象实例是否是给定 *non-nullable-value-type* 的装箱值，然后将该值从实例中复制出来。

对 *nullable-type* 取消装箱在源操作数为 null 时会产生 *nullable-type* 的 null 值；否则将产生从对象实例到 *nullable-type* 的基础类型的取消装箱的包装结果。

参照前一节中关于假想的装箱类的描述，从对象 box 到 *value-type* T 的取消装箱转换包括执行表达式 ((Box<T>)box).value。因此，下面的语句

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

在概念上相当于

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

为使针对给定 *non-nullable-value-type* 的取消装箱转换在运行时取得成功，源操作数的值必须是对该 *non-nullable-value-type* 的装箱值的引用。如果源操作数为 null，则将引发 System.NullReferenceException。如果源操作数是对不兼容对象的引用，则将引发 System.InvalidCastException。

为使针对给定 *nullable-type* 的取消装箱转换在运行时取得成功，源操作数的值必须是 null 或是对该 *nullable-type* 的基础 *non-nullable-value-type* 的装箱值的引用。如果源操作数是对不兼容对象的引用，则将引发 System.InvalidCastException。

4.4 构造类型

泛型类型声明本身表示未绑定的泛型类型 (unbound generic type)，它通过应用类型实参 (type argument) ***arguments*** 被用作构成许多不同类型的“蓝图”。类型实参编写在紧跟在泛型类型的名称后面的尖括号（< 和 >）中。至少包括一个类型实参的类型称为构造类型 (constructed type)。构造类型可以在语言中能够出现类型名的大多数地方使用。未绑定的泛型类型只能在 *typeof-expression*（第 7.6.11 节）中使用。

构造类型还可以在表达式中用作简单名称（第 7.6.2 节）或在访问成员时使用（第 7.6.4 节）。

在计算 *namespace-or-type-name* 时，仅考虑具有正确数目的类型形参的泛型类型。因此，可以使用同一个标识符标识不同的类型，前提是那些类型具有不同数目的类型形参。当在同一程序中混合使用泛型和非泛型类时，这是很有用的：

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

即使未直接指定类型形参，*type-name* 也可以标识构造类型。当某个类型嵌套在泛型类声明中，并且包含该类型的声明的实例类型被隐式用于名称查找（第 10.3.8.6 节）时，就会出现这种情况：

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

在不安全代码中，构造类型不能用作 *unmanaged-type*（第 18.2 节）。

4.4.1 类型实参

类型实参列表中的每个实参都只是一个 *type*。

type-argument-list:
    < *type-arguments* >

type-arguments:
    type-argument
    type-arguments , *type-argument*

type-argument:
    type

在不安全代码（第 18 章）中，*type-argument* 不可以是指针类型。每个类型实参都必须满足对应的类型形参上的所有约束（第 10.1.5 节）。

4.4.2 开放和封闭类型

所有类型都可归类为开放类型 (open type) 或封闭类型 (closed type)。开放类型是包含类型形参的类型。更明确地说：

- 类型形参定义开放类型。
- 当且仅当数组元素类型是开放类型时，该数组类型才是开放类型。
- 当且仅当构造类型的一个或多个类型实参为开放类型时，该构造类型才是开放类型。当且仅当构造的嵌套类型的一个或多个类型实参或其包含类型的类型实参为开放类型时，该构造的嵌套类型才是开放类型。

封闭类型是不属于开放类型的类型。

在运行时，泛型类型声明中的所有代码都在一个封闭构造类型的上下文中执行，这个封闭构造类型是通过将类型实参应用该泛型声明来创建的。泛型类型中的每个类型形参都绑定到特定的运行时类型。所有语句和表达式的运行时处理都始终使用封闭类型，开放类型仅出现在编译时处理过程中。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

每个封闭构造类型都有自己的静态变量集，任何其他封闭构造类型都不会共享这些变量。由于开放类型在运行时并不存在，因此不存在与开放类型关联的静态变量。如果两个封闭构造类型是从相同的未绑定泛型类型构造的，并且它们的对应类型实参属于相同类型，则这两个封闭构造类型是相同类型。

4.4.3 绑定和未绑定类型

术语未绑定类型 (unbound type) 是指非泛型类型或未绑定的泛型类型。术语绑定类型 (bound type) 是指非泛型类型或构造类型。

未绑定类型是指类型声明所声明的实体。未绑定泛型类型本身不是一种类型，不能用作变量、参数或返回值的类型，也不能用作基类型。可以引用未绑定泛型类型的唯一构造是 `typeof` 表达式（第 7.6.11 节）。

4.4.4 满足约束

每当引用构造类型或泛型方法时，都会根据泛型类型或方法（第 10.1.5 节）上声明的类型形参约束对所提供的类型实参进行检查。对于每个 `where` 子句，将根据每个约束检查与命名的类型形参相对应的类型实参 `A`，如下所示：

- 如果约束为类类型、接口类型或类型形参，则假设 `C` 表示该约束，并用所提供的类型实参替换出现在该约束中的任何类型形参。若要满足该约束，必须可通过下列方式之一将类型 `A` 转换为类型 `C`：
  - 标识转换（第 6.1.1 节）
  - 隐式引用转换（第 6.1.6 节）
  - 装箱转换（第 6.1.7 节）-- 前提是类型 A 为不可以为 null 的值类型。
  - 从类型形参 `A` 到 `C` 的隐式引用、装箱或类型形参转换。
- 如果约束为引用类型约束 (`class`) 则类型 `A` 必须满足下列条件之一：
  - `A` 为接口类型、类类型、委托类型或数组类型。请注意，`System.ValueType` 和 `System.Enum` 是满足此约束的引用类型。
  - `A` 是已知为引用类型的类型形参（第 10.1.5 节）。
- 如果约束为值类型约束 (`struct`) 则类型 `A` 必须满足下列条件之一：
  - `A` 为结构类型或枚举类型，但不是可以为 null 的类型。请注意，`System.ValueType` 和 `System.Enum` 是不满足此约束的引用类型。
  - `A` 为具有值类型约束的类型形参（第 10.1.5 节）。
- 如果约束为构造函数约束 `new()`，则类型 `A` 一定不能为 `abstract`，并且必须具有公共无形参构造函数。如果下列条件之一成立，则满足此条件：
  - `A` 为值类型，因为所有值类型都具有公共默认构造函数（第 4.1.2 节）。
  - `A` 为具有构造函数约束的类型形参（第 10.1.5 节）。
  - `A` 为具有值类型约束的类型形参（第 10.1.5 节）。
  - `A` 是不为 `abstract` 并且包含显式声明的无参数 `public` 构造函数的类。
  - `A` 不为 `abstract`，并且具有默认构造函数（第 10.11.4 节）。

如果给定的类型实参未满足一个或多个类型形参的约束，则会发生编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由于类型形参未被继承，因此约束也从不被继承。在下面的示例中，D 需要指定其类型形参 T 上的约束，以便 T 满足基类 B<T> 所施加的约束。相反，类 E 不需要指定约束，因为对于任何 T，List<T> 都实现 IEnumerable。

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

4.5 类型形参

类型形参是指定形参在运行时要绑定到的值类型或引用类型的标识符。

type-parameter:
identifier

由于类型形参可使用许多不同的实际类型实参进行实例化，因此类型形参具有与其他类型稍微不同的操作和限制。这包括：

- 不能直接使用类型形参声明基类（第 10.2.4 节）或接口（第 13.1.3 节）。
- 类型形参上的成员查找规则取决于应用到该类型形参的约束（如果有）。这将在第 7.4 节中详细描述。
- 类型形参的可用转换取决于应用到该类型形参的约束（如果有）。这将在第 6.1.10 和 6.2.6 节中详细描述。
- 如果事先不知道由类型形参给出的类型是引用类型（第 6.1.10 节），不能将标识 null 转换为该类型。不过，可以改为使用 default 表达式（第 7.6.13 节）。此外，具有由类型形参给出的类型的值可以使用 == 和 != 与 null 进行比较（第 7.10.6 节），除非该类型形参具有值类型约束。
- 仅当类型形参受 *constructor-constraint* 或值类型约束（第 7.6.10.1 节）的约束时，才能将 new 表达式（第 10.1.5 节）与类型形参联合使用。
- 不能在特性中的任何位置上使用类型形参。
- 不能在成员访问（第 7.6.4 节）或类型名称（第 3.8 节）中使用类型形参标识静态成员或嵌套类型。
- 在不安全代码中，类型形参不能用作 *unmanaged-type*（第 18.2 节）。

作为类型，类型形参纯粹是一个编译时构造。在运行时，每个类型形参都绑定到一个运行时类型，运行时类型是通过向泛型类型声明提供类型实参来指定的。因此，使用类型形参声明的变量的类型在运行时将是封闭构造类型（第 4.4.2 节）。涉及类型形参的所有语句和表达式的运行时执行都使用作为该形参的类型实参提供的实际类型。

4.6 表达式树类型

表达式树允许将 lambda 表达式表示为数据结构而非可执行代码。表达式目录树是 System.Linq.Expressions.Expression<D> 形式的表达式目录树类型 (expression tree type) 的值，其中 D 是任何委托类型。对于本规范的其余部分，我们将使用简写形式 Expression<D> 引用这些类型。

如果存在从 lambda 表达式到委托类型 D 的转换，则也存在到表达式树类型 Expression<D> 的转换。而 lambda 表达式到委托类型的转换生成引用该 lambda 表达式的可执行代码的委托，到表达式树类型的转换创建该 lambda 表达式的表达式树表示形式。

表达式树是 lambda 表达式在内存中的高效数据表示形式，使 lambda 表达式的结构透明而又清晰。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

与委托类型 D 一样，Expression<D> 具有与 D 相同的参数和返回类型。

下面的示例将 lambda 表达式表示为可执行代码和表达式树。因为存在到 Func<int,int> 的转换，所以也存在到 Expression<Func<int,int>> 的转换：

```
Func<int,int> del = x => x + 1;                  // Code
Expression<Func<int,int>> exp = x => x + 1;      // Data
```

进行上面的赋值之后，委托 del 引用返回 x + 1 的方法，表达式目录树 exp 引用描述表达式 x => x + 1 的数据结构。

泛型类型 Expression<D> 的确切定义以及将 lambda 表达式转换为表达式树类型时有关构造表达式树的准确规则不在本规范的介绍范围之内，将另作说明。

有两个要点需要明确指出：

- 并非所有 lambda 表达式都能转换为表达式树。例如，具有语句体的 lambda 表达式和包含赋值表达式的 lambda 表达式不能这样表示。在这些情况下，转换仍存在，但在编译时将失败。第 0 节中详细介绍了这些例外情况。
- Expression<D> 提供一个实例方法 Compile，该方法产生一个类型为 D 的委托：

  ```
  Func<int,int> del2 = exp.Compile();
  ```

  调用此委托将导致执行表达式树所表示的代码。因此，根据上面的定义，del 和 del2 等效，而且下面的两个语句也将等效：

  ```
  int i1 = del(1);
  int i2 = del2(1);
  ```

  执行此代码后，i1 和 i2 的值都为 2。

4.7 dynamic 类型

dynamic 类型在 C# 中具有特殊含义。其用途在于允许进行动态绑定（在第 7.2.2 节中进行了详细介绍）。

dynamic 被视为与 object 相同，除了以下这些方面：

- 对 dynamic 类型的表达式进行的运算可以动态绑定（第 7.2.2 节）。
- 类型推断（第 7.5.2 节）在 dynamic 和 object 都是候选项时，会优先考虑前者。

由于此等效性，因此存在以下情况：

- object 与 dynamic 之间存在隐式标识转换，将 dynamic 替换为 object 时相同的构造类型之间也存在隐式标识转换
- 与 object 之间的隐式和显式转换也适用于 dynamic。
- 在将 dynamic 替换为 object 时相同的方法签名视为是相同的签名

dynamic 类型在运行时与 object 没有区别。

dynamic 类型的表达式称为动态表达式 (dynamic expression)。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

5. 变量

变量表示存储位置。每个变量都具有一个类型，用于确定哪些值可以存储在该变量中。C# 是一种类型安全的语言，C# 编译器保证存储在变量中的值总是具有合适的类型。通过赋值或使用 ++ 和 -- 运算符可以更改变量的值。

在可以获取变量的值之前，变量必须已明确赋值 (definitely assigned)（第 5.3 节）。

如下面的章节所述，变量是初始已赋值 (initially assigned) 或初始未赋值 (initially unassigned)。初始已赋值的变量有一个正确定义了的初始值，并且总是被视为已明确赋值。初始未赋值的变量没有初始值。为了使初始未赋值的变量在某个位置被视为已明确赋值，变量赋值必须发生在通向该位置的每个可能的执行路径中。

5.1 变量类别

C# 定义了 7 类变量：静态变量、实例变量、数组元素、值参数、引用参数、输出参数和局部变量。后面的章节将介绍其中的每一种类别。

在下面的示例中

```
class A
{
   public static int x;
   int y;

   void F(int[] v, int a, ref int b, out int c) {
      int i = 1;
      c = a + b++;
   }
}
```

x 是静态变量，yy 是实例变量，v[0] 是数组元素，a 是值参数，b 是引用参数，c 是输出参数，i 是局部变量。

5.1.1 静态变量

用 static 修饰符声明的字段称为静态变量 (static variable)。静态变量在包含了它的那个类型的静态构造函数（第 10.12 节）执行之前就存在了，在退出关联的应用程序域时不复存在。

静态变量的初始值是该变量的类型的默认值（第 5.2 节）。

出于明确赋值检查的目的，静态变量被视为初始已赋值。

5.1.2 实例变量

未用 static 修饰符声明的字段称为实例变量 (instance variable)。

5.1.2.1 类中的实例变量

类的实例变量在创建该类的新实例时开始存在，在所有对该实例的引用都已终止，并且已执行了该实例的析构函数（若有）时终止。

类实例变量的初始值是该变量的类型的默认值（第 5.2 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

出于明确赋值检查的目的，类的实例变量被视为初始已赋值。

5.1.2.2 结构中的实例变量

结构的实例变量与它所属的结构变量具有完全相同的生存期。换言之，当结构类型的变量开始存在或停止存在时，该结构的实例变量也随之存在或消失。

结构的实例变量与包含它的结构变量具有相同的初始赋值状态。换言之，当结构变量本身被视为初始已赋值时，它的实例变量也被视为初始已赋值。而当结构变量被视为初始未赋值时，它的实例变量同样被视为未赋值。

5.1.3 数组元素

数组的元素在创建数组实例时开始存在，在没有对该数组实例的引用时停止存在。

每个数组元素的初始值都是其数组元素类型的默认值（第 5.2 节）。

出于明确赋值检查的目的，数组元素被视为初始已赋值。

5.1.4 值参数

未用 `ref` 或 `out` 修饰符声明的参数为值参数 (value parameter)。

值形参在调用该形参所属的函数成员（方法、实例构造函数、访问器或运算符）或匿名函数时开始存在，并用调用中给定的实参的值初始化。当返回该函数成员或匿名函数时值形参通常停止存在。但是，如果值形参被匿名函数（第 7.15 节）捕获，则其生存期将至少延长到从该匿名函数创建的委托或表达式目录树可以被垃圾回收为止。

出于明确赋值检查的目的，值形参被视为初始已赋值。

5.1.5 引用参数

引用形参是用 `ref` 修饰符声明的形参。

引用形参不创建新的存储位置。它表示在对该函数成员或匿名函数调用中以实参形式给出的变量所在的存储位置。因此，引用形参的值总是与基础变量相同。

下面的明确赋值规则适用于引用形参。注意第 5.1.6 节中描述的输出形参的不同规则。

- 变量在可以作为引用形参在函数成员或委托调用中传递之前，必须已明确赋值（第 5.3 节）。
- 在函数成员或匿名函数内部，引用形参被视为初始已赋值。

在结构类型的实例方法或实例访问器内部，`this` 关键字的行为与该结构类型的引用形参完全相同（第 7.6.7 节）。

5.1.6 输出形参

用 `out` 修饰符声明的形参是输出形参。

输出形参不创建新的存储位置。而输出形参表示在对该函数成员或委托调用中以实参形式给出的变量所在的存储位置。因此，输出形参的值总是与基础变量相同。

下面的明确赋值规则应用于输出形参。注意第 5.1.5 节中描述的引用形参的不同规则。

- 变量在可以作为输出形参在函数成员或委托调用中传递之前无需明确赋值。
- 在正常完成函数成员或委托调用之后，每个作为输出形参传递的变量都被认为在该执行路径中已赋值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 在函数成员或匿名函数内部，输出形参被视为初始未赋值。
- 函数成员或匿名函数的每个输出形参在该函数成员或匿名函数正常返回前都必须已明确赋值（第5.3节）。

在结构类型的实例构造函数内部，`this`关键字的行为与结构类型的输出形参完全相同（第7.6.7节）。

5.1.7 局部变量

局部变量(local variable) 可通过 *local-variable-declaration* 来声明，此声明可以出现在 *block*、*for-statement*、*switch-statement* 或 *using-statement* 中；也可由 *foreach-statement* 或 *try-statement* 的 *specific-catch-clause* 来声明。

局部变量的生存期是程序执行过程中的某一“段”，在此期间，一定会为该局部变量保留存储。此生存期从进入关联的 *block*、*for-statement*、*switch-statementusing-statement*、*foreach-statement* 或 *specific-catch-clause* 开始，至少延长到该 *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement* 或 *specific-catch-clause* 以任何方式结束为止。（进入封闭 *block* 或调用方法会挂起（但不会结束）当前的 *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement* 或 *specific-catch-clause* 的执行）。如果局部变量被匿名函数捕获（第7.15.5.1节），其生存期将至少延长到从该匿名函数创建的委托或表达式目录树以及引用该被捕获变量的其他所有对象可以被垃圾回收为止。

如果以递归方式进入父 *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement* 或 *specific-catch-clause*，则每次都创建局部变量的新实例，并且重新计算它的 *local-variable-initializer*（如果有）。

由 *local-variable-declaration* 引入的局部变量不自动初始化，因此没有默认值。出于明确赋值检查的目的，由 *local-variable-declaration* 引入的局部变量被视为未赋初始值。*local-variable-declaration* 可包括 *local-variable-initializer*，在此情况下，只有在初始化表达式之后，变量才被视为已明确赋值（第 5.3.3.4 节）。

在由 *local-variable-declaration* 引入的局部变量的范围内，在 *local-variable-declarator* 之前的文本位置引用该局部变量是编译时错误。如果局部变量的声明是隐式的（第8.5.1节），则在 *local-variable-declarator* 内引用该变量也是错误的。

foreach-statement 或 *specific-catch-clause* 引入的局部变量被视为在它的整个范围内已明确赋值。

局部变量的实际生存期依赖于具体实现。例如，编译器可能静态地确定块中的某个局部变量只用于该块的一小部分。使用这种分析，编译器生成的代码可能会使该变量存储的生存期短于包含该变量的块的生存期。

局部引用变量所引用的存储的回收与该局部引用变量（第3.9节）的生存期无关。

5.2 默认值

以下类别的变量自动初始化为它们的默认值：

- 静态变量。
- 类实例的实例变量。
- 数组元素。

变量的默认值取决于该变量的类型，并按下如下规则确定：

- 对于 *value-type* 的变量，默认值与该 *value-type* 的默认构造函数（第4.1.2节）所计算的值相同。
- 对于 *reference-type* 的变量，默认值为`null`。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

初始化为默认值的实现方法一般是让内存管理器或垃圾回收器在分配内存以供使用之前，将内存初始化为“所有位归零”。由于这个原因，使用所有位归零来表示 null 引用很方便。

5.3 明确赋值

在函数成员可执行代码中的给定位置，如果编译器可通过特定的静态流程分析（第5.3.3 节）证明变量已自动初始化或已成为至少一个赋值的目标，则称该变量已明确赋值 (definitely assigned)。非正式地讲，明确赋值的规则为：

- 初始已赋值的变量（第 5.3.1 节）总是被视为已明确赋值。
- 如果所有可能通向给定位置的执行路径都至少包含以下内容之一，则初始未赋值的变量（第 5.3.2 节）被视为在该位置已明确赋值：
  - 将变量作为左操作数的简单赋值（第 7.17.1 节）。
  - 将变量作为输出形参传递的调用表达式（第 7.6.5 节）或对象创建表达式（第 7.6.10.1 节）。
  - 对于局部变量，包含变量初始值设定项的局部变量声明（第 8.5.1 节）。

以上非正式规则所基于的正式规范在第5.3.1 节、第 5.3.2 节和第 5.3.3 节中说明。

关于对一个 *struct-type* 变量的实例变量是否明确赋值，既可个别地也可作为整体进行跟踪。除了上述规则，下面的规则也应用于 *struct-type* 变量及其实例变量：

- 如果一个实例变量的包含它的那个 *struct-type* 变量被视为已明确赋值，则该实例变量被视为已明确赋值。
- 如果一个 *struct-type* 变量的每个实例变量都被视为已明确赋值，则该结构类型变量被视为已明确赋值。

在下列上下文中要求实施明确赋值：

- 变量必须在获取其值的每个位置都已明确赋值。这确保了从来不会出现未定义的值。变量在表达式中出现被视为要获取该变量的值，除非当
  - 该变量为简单赋值的左操作数，
  - 该变量作为输出形参传递，或者
  - 该变量为 *struct-type* 变量并作为成员访问的左操作数出现。
- 变量必须在它作为引用形参传递的每个位置都已明确赋值。这确保了被调用的函数成员可以将引用形参视为初始已赋值。
- 函数成员的所有输出形参必须在函数成员返回的每个位置都已明确赋值，返回位置包括通过 `return` 语句实现的返回，或者通过执行语句到达函数成员体结尾的返回。这确保了函数成员不在输出形参中返回未定义的值，从而使编译器能够把一个对函数成员的调用当作对某些变量的赋值，这些变量在该调用中被当作输出形参传递。
- *struct-type* 实例构造函数的 `this` 变量必须在该实例构造函数返回的每个位置明确赋值。

5.3.1 初始已赋值变量

以下类别的变量属于初始已赋值变量：

- 静态变量。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 类实例的实例变量。
- 初始已赋值结构变量的实例变量。
- 数组元素。
- 值形参。
- 引用参数。
- 在 `catch` 子句或 `foreach` 语句中声明的变量。

5.3.2 初始未赋值变量

以下类别的变量属于初始未赋值变量：

- 初始未赋值结构变量的实例变量。
- 输出形参，包括结构实例构造函数的 `this` 变量。
- 局部变量，在 `catch` 子句或 `foreach` 语句中声明的那些除外。

5.3.3 确定明确赋值的细则

为了确定每个已使用变量都已明确赋值，编译器必须使用与本节中描述的进程等效的进程。

编译器处理每个具有一个或多个初始未赋值变量的函数成员的体。对于每个初始未赋值的变量 *v*，编译器在函数成员中的下列每个点上确定 *v* 的明确赋值状态 (definite assignment state)：

- 在每个语句的开头处
- 在每个语句的结束点（第 8.1 节）
- 在每个将控制转移到另一个语句或语句结束点的 arc 上
- 在每个表达式的开头处
- 在每个表达式的结尾处

v 的明确赋值状态可以是：

- 明确赋值。这表明在能达到该点的所有可能的控制流上，*v* 都已赋值。
- 未明确赋值。当在 `bool` 类型表达式结尾处确定变量的状态时，未明确赋值的变量的状态可能（但不一定）属于下列子状态：
  - 在 true 表达式后明确赋值。此状态表明如果该布尔表达式计算为 true，则 *v* 是明确赋值的，但如果布尔表达式计算为 false，则不一定要赋值。
  - 在 false 表达式后明确赋值。此状态表明如果该布尔表达式计算为 false，则 *v* 是明确赋值的，但如果布尔表达式计算为 true，则不一定要赋值。

下列规则控制变量 *v* 的状态在每个位置是如何确定的。

5.3.3.1 一般语句规则

- *v* 在函数成员体的开头处不是明确赋值的。
- *v* 在任何无法访问的语句的开头处都是明确赋值的。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 在任何其他语句开头处，为了确定 *v* 的明确赋值状态，请检查以该语句开头处为目标的所有控制流转移上的 *v* 的明确赋值状态。当且仅当 *v* 在所有此类控制流转移上是明确赋值的时，*v* 才在该语句的开始处明确赋值。确定可能的控制流转移集的方法与检查语句可访问性的方法（第 8.1 节）相同。
- 在块、checked、unchecked、if、while、 do、for、foreach、lock、using 或 switch 等语句的结束点处，为了确定 *v* 的明确赋值状态，需检查以该语句结束点为目标的所有控制流转移上的 *v* 的明确赋值状态。如果 *v* 在所有此类控制流转移上是明确赋值的，则 *v* 在该语句结束点明确赋值。否则，*v* 在语句结束点处不是明确赋值的。确定可能的控制流转移集的方法与检查语句可访问性的方法（第 8.1 节）相同。

5.3.3.2 块语句、checked 和 unchecked 语句

在指向位于某块中语句列表的第一个语句（如果语句列表为空，则指向该块的结束点）的控制转移上，*v* 的明确赋值状态与块语句、checked 或 unchecked 语句之前的 *v* 的明确赋值状态相同。

5.3.3.3 表达式语句

对于由表达式 *expr* 组成的表达式语句 *stmt*：

- *v* 在 *expr* 的开头处与在 *stmt* 的开头处具有相同的明确赋值状态。
- 如果 *v* 在 *expr* 的结尾处明确赋值，则它在 *stmt* 的结束点也明确赋值；否则，它在 *stmt* 的结束点也不明确赋值。

5.3.3.4 声明语句

- 如果 *stmt* 是不带有初始值设定项的声明语句，则 *v* 在 *stmt* 的结束点与在 *stmt* 的开头处具有相同的明确赋值状态。
- 如果 *stmt* 是带有初始值设定项的声明语句，则确定 *v* 的明确赋值状态时可把 *stmt* 当作一个语句列表，其中每个带有初始值设定项的声明对应一个赋值语句（按声明的顺序）。

5.3.3.5 if 语句

对于以下形式的 if 语句 *stmt*：

```
if ( expr ) then-stmt else else-stmt
```

- *v* 在 *expr* 的开头处与在 *stmt* 的开头处具有相同的明确赋值状态。
- 如果 *v* 在 *expr* 的结尾处明确赋值，则它在指向 *then-stmt* 和 *else-stmt* 或指向 *stmt* 的结束点（如果没有 else 子句）的控制流转移上是明确赋值的。
- 如果 *v* 在 *expr* 的结尾处具有“在 true 表达式后明确赋值”状态，则它在指向 *then-stmt* 的控制流转移上是明确赋值的，在指向 *else-stmt* 或指向 *stmt* 的结束点（如果没有 else 子句）的控制流转移上不是明确赋值的。
- 如果 *v* 在 *expr* 的结尾处具有“在 false 表达式后明确赋值”状态，则它在指向 *else-stmt* 的控制流转移上是明确赋值的，在指向 *then-stmt* 的控制流转移上不是明确赋值的。此后，当且仅当它在 *stmt* 的结束点是明确赋值的时，它在 *then-stmt* 的结束点才是明确赋值的。
- 否则，认为 *v* 在指向 *then-stmt* 或 *else-stmt*，或指向 *stmt* 的结束点（如果没有 else 子句）的控制流转移上都不是明确赋值的。

5.3.3.6 switch 语句

在带有控制表达式 *expr* 的 switch 语句 *stmt* 中：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 位于 *expr* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的状态相同。
- 在指向可访问的 switch 块语句列表的控制流转移上，*v* 的明确赋值状态就是 *v* 在 *expr* 结尾处的明确赋值状态。

5.3.3.7 while 语句

对于以下形式的 `while` 语句 *stmt*：

```
while ( expr ) while-body
```

- *v* 在 *expr* 的开头处与在 *stmt* 的开头处具有相同的明确赋值状态。
- 如果 *v* 在 *expr* 的结尾处明确赋值，则它在指向 *while-body* 和指向 *stmt* 结束点的控制流转移上是明确赋值的。
- 如果 *v* 在 *expr* 的结尾处具有“在 true 表达式后明确赋值”状态，则它在指向 *while-body* 的控制流转移上是明确赋值的，但在 *stmt* 的结束点处不是明确赋值的。
- 如果 *v* 在 *expr* 的结尾处具有“在 false 表达式后明确赋值”状态，则它在指向 *stmt* 的结束点的控制流转移上是明确赋值的，但在指向 *while-body* 的控制流转移上不是明确赋值的。

5.3.3.8 do 语句

对于以下形式的 `do` 语句 *stmt*：

```
do do-body while ( expr ) ;
```

- *v* 在从 *stmt* 的开头处到 *do-body* 的控制流转移上的明确赋值状态与在 *stmt* 的开头处的状态相同。
- *v* 在 *expr* 的开头处与在 *do-body* 的结束点具有相同的明确赋值状态。
- 如果 *v* 在 *expr* 的结尾处是明确赋值的，则它在指向 *stmt* 的结束点的控制流转移上是明确赋值的。
- 如果 *v* 在 *expr* 的结尾处的状态为“在 false 表达式后明确赋值”，则它在指向 *stmt* 的结束点的控制流转移上是明确赋值的。

5.3.3.9 for 语句

对具有以下形式的 `for` 语句进行的明确赋值检查：

```
for ( for-initializer ; for-condition ; for-iterator ) embedded-statement
```

就如执行下列语句一样：

```
{
    for-initializer ;
    while ( for-condition ) {
        embedded-statement ;
        for-iterator ;
    }
}
```

如果 `for` 语句中省略了 *for-condition*，则在确定关于明确赋值的状态时，可把上述展开语句列表中的 *for-condition* 当作 `true`。

5.3.3.10 break、continue 和 goto 语句

由 `break`、`continue` 或 `goto` 语句引起的控制流转移上的 *v* 的明确赋值状态与 *v* 在该语句开头处的明确赋值状态是一样的。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

5.3.3.11 throw 语句

对于以下形式的语句 *stmt*

```
throw expr ;
```

位于 *expr* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的明确赋值状态相同。

5.3.3.12 return 语句

对于以下形式的语句 *stmt*

```
return expr ;
```

- 位于 *expr* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的明确赋值状态相同。
- 如果 *v* 是输出形参，则它必须在下列两个位置之一被明确赋值：
  - 在 *expr* 之后
  - 或者在包含 `return` 语句的 `try-finally` 或 `try-catch-finally` 的 `finally` 块的结尾处。

对于以下形式的语句 stmt：

```
return ;
```

- 如果 *v* 是输出形参，则它必须在下列两个位置之一被明确赋值：
  - 在 *stmt* 之前
  - 或者在包含 `return` 语句的 `try-finally` 或 `try-catch-finally` 的 `finally` 块的结尾处。

5.3.3.13 try-catch 语句

对于以下形式的语句 *stmt*：

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
```

- 位于 *try-block* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的明确赋值状态相同。
- 位于 *catch-block-i*（对于所有的 *i*）开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的明确赋值状态相同。
- 当且仅当 *v* 在 *try-block* 和每个 *catch-block-i*（每个 *i* 从 1 到 *n*）的结束点明确赋值时，*stmt* 结束点处的 *v* 的明确赋值状态才是明确赋值的。

5.3.3.14 try-finally 语句

对于以下形式的 `try` 语句 *stmt*：

```
try try-block finally finally-block
```

- 位于 *try-block* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的明确赋值状态相同。
- 位于 *finally-block* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的明确赋值状态相同。
- 当且仅当下列条件中至少有一个为真时，位于 *stmt* 结束点处的 *v* 的明确赋值状态才是明确赋值的：
  - *v* 在 *try-block* 的结束点明确赋值

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- *v* 在 *finally-block* 的结束点明确赋值

如果控制流转移（例如，goto 语句）从 *try-block* 内开始，在 *try-block* 外结束，那么如果 *v* 在 *finally-block* 的结束点明确赋值，*v* 也被认为在该控制流转移上明确赋值。（这不是必要条件，如果 *v* 由于其他原因在该控制流转移上明确赋值，则它仍被视为明确赋值）。

5.3.3.15 try-catch-finally 语句

对具有以下形式的 try-catch-finally 语句进行的明确赋值分析：

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
finally finally-block
```

在进行明确赋值分析时，可把该语句当作包含了 try-catch 语句的 try-finally 语句，如下所示：

```
try {
   try try-block
   catch(...) catch-block-1
   ...
   catch(...) catch-block-n
}
finally finally-block
```

下面的示例演示 try 语句（第 8.10 节）的不同块如何影响明确赋值状态。\r \h

```
class A
{
   static void F() {
      int i, j;
      try {
         goto LABEL;
         // neither i nor j definitely assigned
         i = 1;
         // i definitely assigned
      }

      catch {
         // neither i nor j definitely assigned
         i = 3;
         // i definitely assigned
      }

      finally {
         // neither i nor j definitely assigned
         j = 5;
         // j definitely assigned
      }
      // i and j definitely assigned
     LABEL:;
      // j definitely assigned

   }
}
```

5.3.3.16 foreach 语句

对于以下形式的 foreach 语句 *stmt*：

```
foreach ( type identifier in expr ) embedded-statement
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 位于 *expr* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的状态相同。
- 在指向 *embedded-statement* 或指向 *stmt* 结束点处的控制流转移上，*v* 的明确赋值状态与位于 *expr* 结尾处的 *v* 的状态相同。

5.3.3.17 using 语句

对于以下形式的 `using` 语句 *stmt*：

```
using ( resource-acquisition ) embedded-statement
```

- 位于 *resource-acquisition* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的状态相同。
- 在指向 *embedded-statement* 的控制流转移上，*v* 的明确赋值状态与位于 *resource-acquisition* 结尾处的 *v* 的状态相同。

5.3.3.18 lock 语句

对于以下形式的 `lock` 语句 *stmt*：

```
lock ( expr ) embedded-statement
```

- 位于 *expr* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的状态相同。
- 在指向 *embedded-statement* 的控制流转移上，*v* 的明确赋值状态与位于 *expr* 结尾处的 *v* 的状态相同。

5.3.3.19 yield 语句

对于以下形式的 `yield return` 语句 *stmt*：

```
yield return expr ;
```

- 位于 *expr* 开头处的 *v* 的明确赋值状态与位于 *stmt* 开头处的 *v* 的状态相同。
- 位于 *stmt* 结尾处的 *v* 的明确赋值状态与位于 *expr* 结尾处的 *v* 的状态相同。

`yield break` 语句对明确赋值状态没有任何影响。

5.3.3.20 简单表达式的一般规则

以下规则适用于这些类型的表达式：文本（第 7.6.1 节）、简单名称（第 7.6.2 节）、成员访问表达式（第 7.6.4 节）、非索引基访问表达式（第 7.6.8 节）、`typeof` 表达式（第 7.6.11 节）和默认值表达式（第 7.6.13 节）。

- 位于此类表达式结尾处的 *v* 的明确赋值状态与位于表达式开头处的 *v* 的明确赋值状态相同。

5.3.3.21 带有嵌入表达式的表达式的一般规则

下列规则应用于这些类型的表达式：带括号的表达式（第 7.6.3 节）；元素访问表达式（第 7.6.6 节）；带索引的基访问表达式（第 7.6.8 节）；增量和减量表达式（第 7.6.9 节、第 7.7.5 节）；强制转换表达式（第 7.7.6 节）；一元 `+`、`-`、`~`、`*` 表达式；二元 `+`、`-`、`*`、`/`、`%`、`<<`、`>>`、`<`、`<=`、`>`、`>=`、`==`、`!=`、`is`、`as`、`&`、`|`、`^` 表达式（第 7.8 节、第 7.9 节、第 7.10 节、第 7.11 节）；复合赋值表达式（第 7.17.2 节）；`checked` 和 `unchecked` 表达式（第 7.6.12 节）；数组和委托创建表达式（第 7.6.10 节）。

这些表达式的每一个都有一个和多个按固定顺序无条件计算的子表达式。例如，二元运算符 `%` 先计算运算符左边的值，然后计算右边的值。索引操作先计算索引表达式，然后按从左到右的顺序计算每个索引表达式。对于具有子表达式 $expr_1$、$expr_2$、…、$expr_n$ 的表达式 *expr*，按下列顺序计算：

- 位于 $expr_1$ 开头处的 *v* 的明确赋值状态与位于 *expr* 开头处的 v 的明确赋值状态相同。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 位于 $expr_i$（i 大于 1）开头处的 *v* 的明确赋值状态与位于 $expr_{i-1}$ 结尾处的 v 的明确赋值状态相同。
- 位于 *expr* 结尾处的 *v* 的明确赋值状态与位于 $expr_n$ 结尾处的 v 的明确赋值状态相同。

5.3.3.22 调用表达式和对象创建表达式

对于以下形式的调用表达式 *expr*：

primary-expression (arg_1 , arg_2 , … , arg_n)

或以下形式的对象创建表达式：

`new` *type* (arg_1 , arg_2 , … , arg_n)

- 对于调用表达式，位于 *primary-expression* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的状态相同。
- 对于调用表达式，位于 arg_1 之前的 *v* 的明确赋值状态与位于 *primary-expression* 之后的 *v* 的明确赋值状态相同。
- 对于对象创建表达式，位于 arg_1 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的状态相同。
- 对于每一个参数 arg_i，位于 arg_i 之后的 *v* 的明确赋值状态由标准表达式规则决定，其中忽略所有的 `ref` 或 `out` 修饰符。
- 对于每一个 *i* 大于 1 的实参 arg_i，位于 arg_i 之前的 *v* 的明确赋值状态与位于 arg_{i-1} 之后的 *v* 的状态相同。
- 如果变量 *v* 是被作为 `out` 参数传递（即，形式为“`out` *v*”的参数），则无论将它用作哪一个参数，在 *expr* 之后，*v* 的状态是明确赋值的。否则，位于 *expr* 之后的 *v* 的状态与位于 arg_n 之后的 *v* 的状态相同。
- 对于数组初始值设定项（第 7.6.10.4 节）、对象初始值设定项（第 7.6.10.2 节）、集合初始值设定项（第 7.6.10.3 节）和匿名对象初始值设定项（第 7.6.10.6 节），明确赋值状态由定义这些构造所依据的扩展决定。

5.3.3.23 简单赋值表达式

对于具有形式 *w* = *expr-rhs* 的表达式 *expr*：

- 位于 *expr-rhs* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。
- 如果 *w* 与 *v* 是同一变量，则位于 *expr* 之后的 *v* 的明确赋值状态是明确赋值的。否则，位于 *expr* 之后的 *v* 的明确赋值状态与位于 *expr-rhs* 之后的 *v* 的明确赋值状态相同。

5.3.3.24 && 表达式

对于形式为 *expr-first* && *expr-second* 的表达式 *expr*：

- 位于 *expr-first* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。
- 如果位于 *expr-first* 之后的 *v* 的状态是明确赋值的或为“在 true 表达式后明确赋值”，则位于 *expr-second* 之前的 *v* 的明确赋值状态是明确赋值的。否则，它就不是明确赋值的。
- 位于 *expr* 之后的 *v* 的明确赋值状态取决于：
  - 如果 *expr-first* 是值为 `false` 的常量表达式，则位于 *expr* 之后的 *v* 的明确赋值状态与位于 *expr-first* 之后的 *v* 的明确赋值状态相同。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，如果在 *expr-first* 之后，*v* 的状态是明确赋值的，则在 *expr* 之后的 *v* 的状态也是明确赋值的。
- 否则，如果位于 *expr-second* 之后的 *v* 的状态是明确赋值的，而且位于 *expr-first* 之后的 *v* 的状态为“在 false 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是明确赋值的。
- 否则，如果位于 *expr-second* 之后的 *v* 的状态是明确赋值的或为“在 true 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是“在 true 表达式后明确赋值”。
- 否则，如果位于 *expr-first* 之后的 *v* 的状态是“在 false 表达式后明确赋值”，而且位于 *expr-second* 之后的 *v* 的状态是“在 false 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是“在 false 表达式后明确赋值”。
- 否则，在 *expr* 之后，*v* 的状态就不是明确赋值的。

在下面的示例中

```
class A
{
   static void F(int x, int y) {
      int i;
      if (x >= 0 && (i = y) >= 0) {
         // i definitely assigned
      }
      else {
         // i not definitely assigned
      }
      // i not definitely assigned
   }
}
```

变量 `i` 被视为在 `if` 语句的一个嵌入语句中已明确赋值，而在另一个嵌入语句中未明确赋值。在 `F` 方法中的 `if` 语句中，由于总是在第一个嵌入语句执行前执行表达式 `(i = y)`，因此变量 `i` 在第一个嵌入语句中已明确赋值。相反，变量 `i` 在第二个嵌入语句中没有明确赋值，因 `x >= 0` 可能已测试为 false，从而导致变量 `i` 未赋值。

5.3.3.25 || 表达式

对于形式为 *expr-first* `||` *expr-second* 的表达式 *expr*：

- 位于 *expr-first* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。
- 如果位于 *expr-first* 之后的 *v* 的状态是明确赋值的或“在 false 表达式后明确赋值”，则位于 *expr-second* 之前的 *v* 的明确赋值状态是明确赋值的。否则，它就不是明确赋值的。
- 位于 *expr* 之后的 *v* 的明确赋值状态取决于：
  - 如果 *expr-first* 是值为 `true` 的常量表达式，则位于 *expr* 之后的 *v* 的明确赋值状态与位于 *expr-first* 之后的 *v* 的明确赋值状态相同。
  - 否则，如果在 *expr-first* 之后，*v* 的状态是明确赋值的，则在 *expr* 之后的 *v* 的状态也是明确赋值的。
  - 否则，如果位于 *expr-second* 之后的 *v* 的状态是明确赋值的，而且位于 *expr-first* 之后的 *v* 的状态为“在 true 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是明确赋值的。
  - 否则，如果位于 *expr-second* 之后的 *v* 的状态是明确赋值的或是“在 false 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是“在 false 表达式后明确赋值”。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，如果位于 *expr-first* 之后的 *v* 的状态是“在 true 表达式后明确赋值”，而且位于 *expr-second* 之后的 *v* 的状态是“在 true 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是“在 true 表达式后明确赋值”。
- 否则，在 *expr* 之后，*v* 的状态就不是明确赋值的。

在下面的示例中

```
class A
{
   static void G(int x, int y) {
      int i;
      if (x >= 0 || (i = y) >= 0) {
         // i not definitely assigned
      }
      else {
         // i definitely assigned
      }
      // i not definitely assigned
   }
}
```

变量 `i` 被视为在 `if` 语句的一个嵌入语句中已明确赋值，而在另一个嵌入语句中未明确赋值。在 `G` 方法中的 `if` 语句中，由于总是在第二个嵌入语句执行前执行表达式 `(i = y)`，因此变量 `i` 在第一个嵌入语句中已明确赋值。相反，在第一个嵌入语句中，变量 `i` 的状态不是明确赋值的，因为 `x >= 0` 可能已测试为 true，从而导致变量 `i` 未赋值。

5.3.3.26 !表达式

对于形式为 ! *expr-operand* 的表达式 *expr*：

- 位于 *expr-operand* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。
- 位于 *expr* 之后的 *v* 的明确赋值状态取决于：
  - 如果在 *expr-operand* 之后，*v* 的状态是明确赋值的，则在 *expr* 之后，*v* 的状态也是明确赋值的。
  - 如果在 *expr-operand* 之后，*v* 的状态不是明确赋值的，则在 *expr* 之后，*v* 的状态也不是明确赋值的。
  - 如果位于 *expr-operand* 之后的 *v* 的状态是“在 false 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是“在 true 表达式后明确赋值”。
  - 如果位于 *expr-operand* 之后的 *v* 的状态是“在 true 表达式后明确赋值”，则位于 *expr* 之后的 *v* 的状态是“在 false 表达式后明确赋值”。

5.3.3.27 ?? 表达式

对于形式为 *expr-first* ?? *expr-second* 的表达式 *expr*：

- 位于 *expr-first* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。
- 位于 *expr-second* 之前的 *v* 的明确赋值状态与位于 *expr-first* 之后的 *v* 的明确赋值状态相同。
- 位于 *expr* 之后的 *v* 的明确赋值状态取决于：
  - 如果 *expr-first* 是值为 null 的常量表达式（第 7.19 节），则位于 *expr* 之后的 *v* 的状态与位于 *expr-second* 之后的 *v* 的状态相同。
- 否则，位于 *expr* 之后的 *v* 的状态与位于 *expr-first* 之后的 *v* 的明确赋值状态相同。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

5.3.3.28 ?: 表达式

对于形式为 *expr-cond* ? *expr-true* : *expr-false* 的表达式 *expr*：

- 位于 *expr-cond* 之前的 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。
- 当且仅当下列条件中有一个符合时，*expr-true* 之前的 *v* 的明确赋值状态才是明确赋值的：
  - *expr-cond* 是值为 `false` 的常量表达式
  - 在 *expr-cond* 之后的 *v* 的状态是明确赋值的或“在 true 表达式后明确赋值的”。
- 当且仅当下列条件中有一个符合时，*expr-false* 之前的 *v* 的明确赋值状态才是明确赋值的：
  - *expr-cond* 是值为 `true` 的常量表达式
- 在 *expr-cond* 之后的 *v* 的状态是明确赋值的或“在 false 表达式后明确赋值的”。
- 位于 *expr* 之后的 *v* 的明确赋值状态取决于：
  - 如果 *expr-cond* 是值为 `true` 的常量表达式（第 7.19 节），则位于 *expr* 之后的 *v* 的状态与位于 *expr-true* 之后的 *v* 的状态相同。
  - 否则，如果 *expr-cond* 是值为 `false` 的常量表达式（第 7.19 节），则位于 *expr* 之后的 *v* 的状态与位于 *expr-false* 之后的 *v* 的状态相同。
  - 否则，如果位于 *expr-true* 之后的 *v* 的状态是明确赋值的，而且位于 *expr-false* 之后的 *v* 的状态也是明确赋值的，则位于 *expr* 之后的 *v* 的状态是明确赋值的。
  - 否则，在 *expr* 之后，*v* 的状态就不是明确赋值的。

5.3.3.29 匿名函数

对于具有体（*block* 或 *expression*）*body* 的 *lambda-expression* 或 *anonymous-method-expression expr*：

- 位于 *body* 之前的外层变量 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的状态相同。即，外层变量的明确赋值状态是从匿名函数的上下文中继承的。
- 位于 *expr* 之后的外层变量 *v* 的明确赋值状态与位于 *expr* 之前的 *v* 的明确赋值状态相同。

下面的示例

```
delegate bool Filter(int i);

void F() {
   int max;

   // Error, max is not definitely assigned
   Filter f = (int n) => n < max;

   max = 5;
   DoWork(f);
}
```

生成一个编译时错误，因为在声明匿名函数的位置，`max` 尚未明确赋值。下面的示例

```
delegate void D();

void F() {
   int n;
   D d = () => { n = 1; };

   d();
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        // Error, n is not definitely assigned
        Console.WriteLine(n);
    }
```

也生成一个编译时错误，因为匿名函数中对 n 的赋值并不影响匿名函数之外的 n 的明确赋值状态。

5.4 变量引用

variable-reference 是一个 *expression*，它被归类为一个变量。*variable-reference* 表示一个存储位置，访问它可以获取当前值以及存储新值。

variable-reference:
    expression

在 C 和 C++ 中，*variable-reference* 称为 *lvalue*。

5.5 变量引用的原子性

下列数据类型的读写是原子形式的：bool、char、byte、sbyte、short、ushort、uint、int、float 和引用类型。除此之外，当枚举类型的基础类型的属于上述类型之一时，对它的读写也是原子的。其他类型的读写，包括 long、ulong、double 和 decimal 以及用户定义的类型，都不一定是原子的。除专为该目的设计的库函数以外，对于增量或减量这类操作也不能保证进行原子的读取、修改和写入。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

6. 转换

转换(conversion) 使表达式可以被视为一种特定类型。转换可导致将给定类型的表达式视为具有不同的类型，或其可导致没有类型的表达式获得一种类型。转换可以是隐式的 (implicit) 或显式的 (explicit)，这将确定是否需要显式地强制转换。例如，从 int 类型到 long 类型的转换是隐式的，因此 int 类型的表达式可隐式地按 long 类型进行处理。从 long 类型到 int 类型的反向转换是显式的，因此需要显式地强制转换。

```
int a = 123;
long b = a;       // implicit conversion from int to long
int c = (int) b;  // explicit conversion from long to int
```

某些转换由语言定义。程序也可以定义自己的转换（第 6.4 节）。

语言中的一些转换定义为从表达式到类型，另一些定义为从类型到类型。从某个类型开始的转换将应用于具有该类型的所有表达式。

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

Color c0 = 0;        // The expression 0 converts implicitly to enum types
Color c1 = (Color)1; // other int expressions need explicit conversion
```

6.1 隐式转换

下列转换属于隐式转换：

- 标识转换
- 隐式数值转换
- 隐式枚举转换。
- 可以为 null 的隐式转换
- null 文本转换
- 隐式引用转换
- 装箱转换
- 隐式动态转换
- 隐式常量表达式转换
- 用户定义的隐式转换
- 匿名函数转换
- 方法组转换

隐式转换可以在多种情况下发生，包括函数成员调用（第 7.5.4 节）、强制转换表达式（第 7.7.6 节）和赋值（第 7.17 节）。

预定义的隐式转换总是会成功，从来不会导致引发异常。正确设计的用户定义隐式转换同样应表现出这些特性。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由于转换的原因，将类型 object 和 dynamic 视为等效。

然而，动态转换（第 6.1.8 和 6.2.6 节）仅适用于类型为 dynamic（第 4.7 节）的表达式。

6.1.1 标识转换

标识转换是在同一类型（可为任何类型）内进行转换。这种转换的存在，是为了使已具有所需类型的实体可被认为是可转换的（转换为该类型）。

因为将 object 和 dynamic 视为等效，所以在 object 和 dynamic 之间以及对于在将出现的所有 dynamic 替换为 object 时相同的构造类型之间，存在标识转换。

In most cases an identity conversion has no effect at runtime. However, since floating point operations may be performed at higher precision than prescribed by their type (§4.1.6), assignment of their results may result in a loss of precision, and explicit casts are guaranteed to reduce precision to what is prescribed by the type.

6.1.2 隐式数值转换

隐式数值转换为：

- 从 sbyte 到 short、int、long、float、double 或 decimal。
- 从 byte 到 short、ushort、int、uint、long、ulong、float、double 或 decimal。
- 从 short 到 int、long、float、double 或 decimal。
- 从 ushort 到 int、uint、long、ulong、float、double 或 decimal。
- 从 int 到 long、float、double 或 decimal。
- 从 uint 到 long、ulong、float、double 或 decimal。
- 从 long 到 float、double 或 decimal。
- 从 ulong 到 float、double 或 decimal。
- 从 char 到 ushort、int、uint、long、ulong、float、double 或 decimal。
- 从 float 到 double。

从 int、uint、long 或 ulong 到 float 的转换以及从 long 或 ulong 到 double 的转换可能导致精度损失，但决不会影响数值大小。其他的隐式数值转换决不会丢失任何信息。

不存在向 char 类型的隐式转换，因此其他整型的值不会自动转换为 char 类型。

6.1.3 隐式枚举转换

隐式枚举转换允许将 *decimal-integer-literal* 0（或 0L 等）转换为任何 *enum-type* 以及任何基础类型为 *enum-type* 的 *nullable-type*。在后一种情况下，此转换通过转换为基础 *enum-type* 并包装结果（第 4.1.10 节）来计算。

6.1.4 可以为 null 的隐式转换

对不可以为 null 的值类型执行的预定义隐式转换也可用于这些类型的可以为 null 的形式。对于每种从不可以为 null 的值类型 S 转换为不可以为 null 的值类型 T 的预定义隐式标识和数值转换，都存在如下可以为 null 的隐式转换：

- 从 S? 到 T? 的隐式转换。
- 从 S 到 T? 的隐式转换。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

基于从 S 到 T 的基础转换来计算可以为 null 的隐式转换如下进行：

- 如果可以为 null 的转换是从 S? 到 T?：
  - 如果源值为 null（HasValue 属性为 false），则结果为 T? 类型的 null 值。
  - 否则，转换计算过程为从 S? 解包为 S，然后进行从 S 到 T 的基础转换，最后从 T 包装（第 4.1.10 节）为 T?。
- 如果可以为 null 的转换是从 S 到 T?，则转换计算过程为从 S 到 T 的基础转换，然后从 T 包装为 T?。

6.1.5 null 文本转换

从 null 文本到任何可以为 null 的类型存在隐式转换。这种转换产生可以为 null 的给定类型的 null 值（第 4.1.10 节）。

6.1.6 隐式引用转换

隐式引用转换为：

- 从任何 *reference-type* 到 object 和 dynamic。
- 从任何 *class-type* S 到任何 *class-type* T（前提是 S 是从 T 派生的）。
- 从任何 *class-type* S 到任何 *interface-type* T（前提是 S 实现了 T）。
- 从任何 *interface-type* S 到任何 *interface-type* T（前提是 S 是从 T 派生的）。
- 从元素类型为 S_E 的 *array-type* S 到元素类型为 T_E 的 *array-type* T（前提是以下所列条件均成立）：
  - S 和 T 只有元素类型不同。换言之，S 和 T 具有相同的维数。
  - S_E 和 T_E 都是 *reference-type*。
  - 存在从 S_E 到 T_E 的隐式引用转换。
- 从任何 *array-type* 到 System.Array 及其实现的接口。
- 从一维数组类型 S[] 到 System.Collections.Generic.IList<T> 及其基接口（前提是存在从 S 到 T 的隐式标识或引用转换）。
- 从任何 *delegate-type* 到 System.Delegate 及其实现的接口。
- 从 null 文本到任何 *reference-type*。
- 从任何 *reference-type* 到 *reference-type* T （前提是它具有到 *reference-type* T_0 的隐式标识或引用转换，且 T_0 具有到 T 的标识转换）。
- 从任何 *reference-type* 到接口或委托类型 T（前提是它具有到接口或委托类型 T_0 的隐式标识或引用转换，且 T_0 可变化转换为（第 13.1.3.2 节）T）。
- 涉及已知为引用类型的类型参数的隐式转换。有关涉及类型参数的隐式转换的更多详细信息，请参见第 6.1.10 节。

隐式引用转换是指 *reference-type* 之间的转换，可以证明这些转换总能成功，因此不需要在运行时进行任何检查。

引用转换无论是隐式的还是显式的，都不会更改被转换的对象的引用标识。换言之，虽然引用转换可能更改引用的类型，但决不会更改所引用对象的类型或值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

6.1.7 装箱转换

装箱转换允许将 *value-type* 隐式转换为引用类型。存在从任何 *non-nullable-value-type* 到 `object` 和 `dynamic`、`System.ValueType` 以及到 *non-nullable-value-type* 实现的任何 *interface-type* 的装箱转换。此外，*enum-type* 还可以转换为 `System.Enum` 类型。

存在从 *nullable-type* 到 *non-nullable-value-type* 到该引用类型的装箱转换。

如果值类型具有到接口类型 I_0 的装箱转换，且 I_0 具有到接口类型 `I` 的标识转换，则值类型具有到 `I` 的装箱转换。

如果值类型具有到接口或委托类型 I_0 的装箱转换，且 I_0 变化转换为（第 13.1.3.2 节）接口类型 `I`，则值类型具有到 `I` 的装箱转换。

将 *non-nullable-value-type* 的值装箱包括以下操作：分配一个对象实例，然后将 *value-type* 的值复制到该实例中。结构可装箱为类型 `System.ValueType`，因为该类型是所有结构的基类（第 11.3.2 节）。

nullable-type 的值的装箱的过程如下：

- 如果源值为 null（`HasValue` 属性为 false），则结果为目标类型的 null 引用。
- 否则，结果为对经过源值解包和装箱后所产生的装箱 `T` 的引用。

有关装箱转换的介绍详见第 4.3.1 节。

6.1.8 隐式动态转换

存在从 `dynamic` 类型的表达式到任何类型 `T` 的隐式动态转换。转换是动态绑定（第 7.2.2 节），这意味着会在运行时看到从表达式的运行时类型到 `T` 的隐式转换。如果未发现任何转换，则会引发运行时异常。

请注意，此隐式转换似乎违背了第 0 节开头的建议（即隐式转换绝不得导致异常）。然而，不是转换本身，而是转换的 finding 导致异常。运行时异常的风险是使用动态绑定所固有的。如果不需要转换的动态绑定，则可以先将表达式转换为 `object`，然后转换为所需的类型。

下面的示例说明隐式动态转换：

```
object o  = “object”
dynamic d = “dynamic”;

string s1 = o; // Fails at compile-time – no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i     = d; // Compiles but fails at run-time – no conversion exists
```

对 `s2` 和 `i` 的赋值都使用隐式动态转换，其中运算的绑定在运行时之前一直挂起。在运行时，可看到从 `d` 的运行时类型 (`string`) 到目标类型的隐式转换。可发现到 `string` 而不是到 `int` 的转换。

6.1.9 隐式常量表达式转换

隐式常量表达式转换允许进行以下转换：

- `int` 类型的 *constant-expression*（第 7.19 节）可以转换为 `sbyte`、`byte`、`short`、`ushort`、`uint` 或 `ulong` 类型（前提是 *constant-expression* 的值在目标类型的范围之内）。
- `long` 类型的 *constant-expression* 可以转换为 `ulong` 类型（前提是 *constant-expression* 的值非负）。

6.1.10 涉及类型形参的隐式转换

给定的类型形参 `T` 存在下列隐式转换：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 从 T 到其有效基类 C、从 T 到 C 的任何基类，以及从 T 到 C 实现的任何接口。在运行时，如果 T 为值类型，转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 T 到 T 的有效接口集中的接口类型 I 和从 T 到 I 的任何基接口。在运行时，如果 T 为值类型，转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 T 到类型形参 U，前提是 T 依赖 U（第 10.1.5 节）。在运行时，如果 U 是值类型，则 T 和 U 必须是相同类型，并且不执行任何转换。否则，如果 T 为值类型，转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 null 文本到 T（假定 T 已知为引用类型）。
- 从 T 到引用类型 I（前提是它具有到引用类型 S_0 的隐式转换，且 S_0 具有到 S 的标识转换）。在运行时，转换的执行方式与到 S_0 的转换相同。
- 从 T 到接口类型 I（前提是它具有到接口或委托类型 I_0 的隐式转换，且 I_0 可变化转换为 I（第 13.1.3.2 节））。在运行时，如果 T 为值类型，转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。

如果 T 已知为引用类型（第 10.1.5 节），则上面的转换全都归类为隐式引用转换（第 6.1.6 节）。如果 T 已知不为引用类型，则上面的转换全都归类为装箱转换（第 6.1.7 节）。

6.1.11 用户定义的隐式转换

用户定义的隐式转换由以下三部分组成：先是一个标准的隐式转换（可选）；然后是执行用户定义的隐式转换运算符；最后是另一个标准的隐式转换（可选）。计算用户定义的隐式转换的确切规则详见第 6.4.4 节中的说明。

6.1.12 匿名函数转换和方法组转换

匿名函数和方法组本身没有类型，但可以隐式转换为委托类型或表达式树类型。匿名函数转换和方法组转换的更多信息分别详见第 0 节和第 6.6 节。

6.2 显式转换

下列转换属于显式转换：

- 所有隐式转换。
- 显式数值转换。
- 显式枚举转换。
- 可以为 null 的显式转换。
- 显式引用转换。
- 显式接口转换。
- 拆箱转换。
- 显式动态转换
- 用户定义的显式转换。

显式转换可以出现在强制转换表达式（第 7.7.6 节）中。

显式转换集包括所有隐式转换。这意味着允许使用冗余的强制转换表达式。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

不是隐式转换的显式转换是这样的一类转换：它们不能保证总是成功，知道有可能丢失信息，变换前后的类型显著不同，以至值得使用显式表示法。

6.2.1 显式数值转换

显式数值转换是指从一个 *numeric-type* 到另一个 *numeric-type* 的转换，此转换不能用已知的隐式数值转换（第 0 节）实现，它包括：

- 从 sbyte 到 byte、ushort、uint、ulong 或 char。
- 从 byte 到 sbyte 和 char。
- 从 short 到 sbyte、byte、ushort、uint、ulong 或 char。
- 从 ushort 到 sbyte、byte、short 或 char。
- 从 int 到 sbyte、byte、short、ushort、uint、ulong 或 char。
- 从 uint 到 sbyte、byte、short、ushort、int 或 char。
- 从 long 到 sbyte、byte、short、ushort、int、uint、ulong 或 char。
- 从 ulong 到 sbyte、byte、short、ushort、int、uint、long 或 char。
- 从 char 到 sbyte、byte 或 short。
- 从 float 到 sbyte、byte、short、ushort、int、uint、long、ulong、char 或 decimal。
- 从 double 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 decimal。
- 从 decimal 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 double。

由于显式转换包括所有隐式和显式数值转换，因此总是可以使用强制转换表达式（第 7.7.6 节）从任何 n*numeric-type* 转换为任何其他的 *numeric-type*。

显式数值转换有可能丢失信息或导致引发异常。显式数值转换按下面所述处理：

- 对于从一个整型到另一个整型的转换，处理取决于该转换发生时的溢出检查上下文（第 7.6.12 节）：
  - 在 checked 上下文中，如果源操作数的值在目标类型的范围内，转换就会成功，但如果源操作数的值在目标类型的范围外，则会引发 System.OverflowException。
  - 在 unchecked 上下文中，转换总是会成功并按下面这样继续。
    - 如果源类型大于目标类型，则截断源值（截去源值中容不下的最高有效位）。然后将结果视为目标类型的值。
    - 如果源类型小于目标类型，则源值或按符号扩展或按零扩展，以使它的大小与目标类型相同。如果源类型是有符号的，则使用按符号扩展；如果源类型是无符号的，则使用按零扩展。然后将结果视为目标类型的值。
    - 如果源类型的大小与目标类型相同，则源值被视为目标类型的值。
- 对于从 decimal 到整型的转换，源值向零舍入到最接近的整数值，该整数值成为转换的结果。如果得到的整数值在目标类型的范围之外，则会引发 System.OverflowException。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 对于从 float 或 double 到整型的转换，处理取决于发生该转换时的溢出检查上下文（第 7.6.12 节）：
  - 在 checked 上下文中，如下所示进行转换：
    - 如果操作数的值是 NaN 或无穷大，则将引发 System.OverflowException。
    - 否则，源操作数会向零舍入到最接近的整数值。如果该整数值处于目标类型的范围内，则该值就是转换的结果。
    - 否则，将引发 System.OverflowException。
  - 在 unchecked 上下文中，转换总是会成功并按下面这样继续。
    - 如果操作数的值是 NaN 或 infinite，则转换的结果是目标类型的一个未经指定的值。
    - 否则，源操作数会向零舍入到最接近的整数值。如果该整数值处于目标类型的范围内，则该值就是转换的结果。
    - 否则，转换的结果是目标类型的一个未经指定的值。
- 对于从 double 到 float 的转换，double 值将舍入为最接近的 float 值。如果 double 值过小，无法表示为 float 值，则结果为正零或负零。如果 double 值过大，无法表示为 float 值，则结果为正无穷大或负无穷大。如果 double 值为 NaN，则结果也为 NaN。
- 对于从 float 或 double 到 decimal 的转换，源值将转换为 decimal 表示形式，并且在需要时，将它在第 28 位小数位上舍入到最接近的数字（第 4.1.7 节）。如果源值过小，无法表示为 decimal，则结果变成零。如果源值为 NaN、无穷大或者太大而无法表示为 decimal，则将引发 System.OverflowException 异常。
- 对于从 decimal 到 float 或 double 的转换，decimal 值将舍入为最接近的 double 或 float 值。虽然这种转换可能会损失精度，但决不会导致引发异常。

6.2.2 显式枚举转换

显式枚举转换为：

- 从 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 decimal 到任何 *enum-type*。
- 从任何 *enum-type* 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 decimal。
- 从任何 *enum-type* 到任何其他 *enum-type*。

两种类型之间的显式枚举转换是通过将任何参与的 *enum-type* 都按该 *enum-type* 的基础类型处理，然后在产生的类型之间执行隐式或显式数值转换进行的。例如，给定具有 int 基础类型的 *enum-type* E，从 E 到 byte 的转换按从 int 到 byte 的显式数值转换（第 6.2.1 节）来处理，而从 byte 到 E 的转换按从 byte 到 int 的隐式数值转换（第 0 节）来处理。

6.2.3 可以为 null 的显式转换

可以为 null 的显式转换 (Explicit nullable conversion) 允许将对不可以为 null 的值类型执行的预定义显式转换也用于这些类型的可以为 null 的形式。对于从不可以为 null 的值类型 S 转换为不可以为 null 的值类型 T 的每一种预定义显式转换（第 6.1.1 节、第 6.1.2 节、第 6.1.3 节、第 6.2.1 节和第 6.2.2 节），都存在下列可以为 null 的转换：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 从 S? 到 T? 的显式转换。
- 从 S 到 T? 的显式转换。
- 从 S? 到 T 的显式转换。

基于从 S 到 T 的基础转换计算可以为 null 的转换过程如下：

- 如果可以为 null 的转换是从 S? 到 T?：
  - 如果源值为 null（HasValue 属性为 false），则结果为 T? 类型的 null 值。
  - 否则，转换计算过程为从 S? 解包为 S，然后进行从 S 到 T 的基础转换，最后从 T 包装为 T?。
- 如果可以为 null 的转换是从 S 到 T?，则转换计算过程为从 S 到 T 的基础转换，然后从 T 包装为 T?。
- 如果可以为 null 的转换是从 S? 到 T，则转换计算过程为从 S? 到 S 的解包，然后从 S 到 T 的基础转换。

请注意，如果可以为 null 的值为 null，则尝试对该值解包将引发异常。

6.2.4 显式引用转换

显式引用转换为：

- 从 object 和 dynamic 到任何其他 *reference-type*。
- 从任何 *class-type* S 到任何 *class-type* T（前提是 S 为 T 的基类）。
- 从任何 *class-type* S 到任何 *interface-type* T（前提是 S 未密封并且 S 未实现 T）。
- 从任何 *interface-type* S 到任何 *class-type* T（前提是 T 未密封或 T 实现 S）。
- 从任何 *interface-type* S 到任何 *interface-type* T（前提是 S 不是从 T 派生的）。
- 从元素类型为 S_E 的 *array-type* S 到元素类型为 T_E 的 *array-type* T（前提是以下所列条件均成立）：
  - S 和 T 只有元素类型不同。换言之，S 和 T 具有相同的维数。
  - S_E 和 T_E 都是 *reference-type*。
  - 存在从 S_E 到 T_E 的显式引用转换。
- 从 System.Array 及其实现的接口到任何 *array-type*。
- 从一维数组类型 S[] 到 System.Collections.Generic.IList<T> 及其基接口（前提是存在从 S 到 T 的显式标识或引用转换）。
- 从 System.Collections.Generic.IList<S> 及其基接口到一维数组类型 T[]（前提是存在从 S 到 T 的显式标识或引用转换）。
- 从 System.Delegate 及其实现的接口到任何 *delegate-type*。
- 从引用类型到引用类型 T（前提是它具有到引用类型 T_0 的显式引用转换，且 T_0 具有到 T 的标识转换）。
- 从接口类型到接口或委托类型 T（前提是它具有到接口或委托类型 T_0 的显式引用转换，且 T_0 可变化转换为 T，或 T 可变化转换为 T_0（第 13.1.3.2 节））。
- 从 $D<S_1 \cdots S_n>$ 到 $D<T_1 \cdots T_n>$，其中 $D<X_1 \cdots X_n>$ 是泛型委托类型，$D<S_1 \cdots S_n>$ 与 $D<T_1 \cdots T_n>$ 不兼容或不相同，并且，对于 D 的每个类型形参 X_i，存在以下情况：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果 X_i 是固定的，则 S_i 与 T_i 相同。
- 如果 X_i 是协变的，且存在从 S_i 到 T_i 的隐式或显式标识或引用转换。
- 如果 X_i 是逆变的，则 S_i 与 T_i 相同或同为引用类型。

- 涉及已知为引用类型的类型形参的显式转换。有关涉及类型形参的显式转换的更多详细信息，请参见第 6.2.7 节。

显式引用转换是那些需要运行时检查以确保它们正确的引用类型之间的转换。

要使显式引用转换在运行时成功，源操作数的值必须为 null，或源操作数所引用的对象的实际类型必须是可通过隐式引用转换（第 6.1.6 节）或装箱转换（第 6.1.7 节）转换为目标类型的类型。如果显式引用转换失败，则将引发 System.InvalidCastException。

引用转换无论是隐式的还是显式的，都不会更改被转换的对象的引用标识。换言之，虽然引用转换可能更改引用的类型，但决不会更改所引用对象的类型或值。

6.2.5 拆箱转换

取消装箱转换允许将引用类型显式转换为 *value-type*。存在从类型 object、dynamic 和 System.ValueType 到任何 *non-nullable-value-type* 的取消装箱转换，也存在从任何 *interface-type* 到实现 *interface-type* 的任何 *non-nullable-value-type* 的取消装箱转换。而且，类型 System.Enum 可以取消装箱为任何 *enum-type*。

存在从引用类型到 *nullable-type* 的取消装箱转换，条件是存在从该引用类型到 *nullable-type* 的基础 *non-nullable-value-type*e 的取消装箱转换。

如果值类型 S 具有从来自接口类型 I_0 的取消装箱转换，且 I_0 具有到接口类型 I 的标识转换，则它具有来自 I 的取消装箱转换。

如果值类型 S 具有来自接口或委托类型 I_0 的取消装箱转换，且 I_0 可变化转换为 I 或 I 可变化转换为 I_0（第 13.1.3.2 节），则它具有来自 I 的取消装箱转换。

取消装箱操作包括以下两个步骤：首先检查对象实例是否为给定 *value-type* 的装箱值，然后从实例中复制该值。对 *nullable-type* 的 null 引用取消装箱会产生 *nullable-type* 的 null 值。结构可以从类型 System.ValueType 取消装箱，因为该类型是所有结构的基类（第 11.3.2 节）。

有关取消装箱转换的进一步介绍详见第 4.3.2 节。

6.2.6 显式动态转换

存在从 dynamic 类型的表达式到任何类型 T 的显式动态转换。转换是动态绑定（第 7.2.2 节），这意味着会在运行时看到从表达式的运行时类型到 T 的显式转换。如果未发现任何转换，则会引发运行时异常。

如果不需要转换的动态绑定，则可以先将表达式转换为 object，然后转换为所需的类型。

假定定义了下面的类：

```
class C
{
   int i;
   public C(int i) { this.i = i; }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

下面的示例说明显式动态转换：

```
object o  = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

在编译时发现 o 到 C 的最佳转换，为显式引用转换。这会在运行时失败，因为“1”实际上不是 C。然而，d 到 C 的转换作为显式动态转换，会在运行时之前一直挂起，从 d 的运行时类型(string)到 C 的用户定义转换会出现并成功。

6.2.7 涉及类型参数的显式转换

给定的类型形参 T 存在下列显式转换：

- 从 T 的有效基类 C 到 T 和从 C 的任何基类到 T。在运行时，如果 T 为值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从任何接口类型到 T。在运行时，如果 T 为值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从 T 到任何 *interface-type* I，前提是尚未存在从 T 到 I 的隐式转换。在运行时，如果 T 为值类型，则转换将先作为装箱转换执行，然后作为显式引用转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从类型形参 U 到 T，前提是 T 依赖 U（第 10.1.5 节）。在运行时，如果 U 是值类型，则 T 和 U 必须是相同类型，并且不执行任何转换。否则，如果 T 为值类型，则转换会作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。

如果 T 已知为引用类型，则上面的转换全都归类为显式引用转换（第 6.2.4 节）。如果 T 已知不为引用类型，则上面的转换全都归类为取消装箱转换（第 6.2.5 节）。

上面的规则不允许从未受约束的类型形参到非接口类型的直接显式转换，这可能有点奇怪。其原因是为了防止混淆，并使得此类转换的语义更清楚。例如，请考虑下面的声明：

```
class X<T>
{
    public static long F(T t) {
        return (long)t;            // Error
    }
}
```

如果允许从 t 到 int 的直接显式转换，则极有可能认为 X<int>.F(7) 将返回 7L。但结果不是这样，因为仅当在绑定时已知类型将是数字时，才会考虑标准数字转换。为了使语义清楚，必须将上面的示例改写为：

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;    // Ok, but will only work when T is long
    }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

此代码现在可以正常编译，但是在运行时执行 `X<int>.F(7)` 将引发异常，因为不能将装箱的 `int` 直接转换为 `long`。

6.2.8 用户定义的显式转换

用户定义的显式转换由以下三个部分组成：先是一个标准的显式转换（可选），然后是执行用户定义的隐式或显式转换运算符，最后是另一个标准的显式转换（可选）。计算用户定义的显式转换的确切规则详见第 6.4.5 节中的说明。

6.3 标准转换

标准转换是那些预先定义的转换，它们可以作为用户定义转换的组成部分出现。

6.3.1 标准隐式转换

下列隐式转换属于标准隐式转换：

- 标识转换（第 6.1.1 节）
- 隐式数值转换（第 0 节）
- 可以为 null 的隐式转换（第 6.1.4 节）
- Null 文本转换（第 6.1.5 节）
- 隐式引用转换（第 6.1.6 节）
- 装箱转换（第 6.1.7 节）
- 隐式常量表达式转换（第 6.1.8 节）
- 涉及类型形参的隐式转换（第 6.1.10 节）

标准隐式转换特别排除了用户定义的隐式转换。

6.3.2 标准显式转换

标准显式转换包括所有的标准隐式转换以及一个显式转换的子集，该子集是由那些与已知的标准隐式转换反向的转换组成的。换言之，如果存在一个从 `A` 类型到 `B` 类型的标准隐式转换，则一定存在与其对应的两个标准显式转换（一个是从 `A` 类型到 `B` 类型，另一个是从 `B` 类型到 `A` 类型）。

6.4 用户定义的转换

C# 允许通过用户定义的转换 (user-defined conversion) 来增加预定义的隐式和显式转换。用户定义的转换是通过在类类型和结构类型中声明转换运算符（第 10.10.3 节）而引入的。

6.4.1 允许的用户定义转换

C# 只允许声明某些用户定义的转换。具体而言，不可能重新定义已存在的隐式或显式转换。

对于给定的源类型 `S` 和目标类型 `T`，如果 `S` 或 `T` 是可以为 null 的类型，则让 S_0 和 T_0 引用它们的基础类型，否则 S_0 和 T_0 分别等于 `S` 和 `T`。仅当以下条件皆为真时，才允许类或结构声明从源类型 `S` 到目标类型 `T` 的转换：

- S_0 和 T_0 是不同的类型。
- S_0 和 T_0 中有一个是声明该运算符的类类型或结构类型。
- S_0 和 T_0 都不是 *interface-type*。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 除用户定义的转换之外，不存在从 S 到 T 或从 T 到 S 的转换。

适用于用户定义的转换的限制在第 10.10.3 节中有进一步讨论。

6.4.2 提升转换运算符

给定一个从不可以为 null 的值类型 S 到不可以为 null 的值类型 T 的用户定义转换运算符，存在从 S? 转换为 T? 的提升转换运算符 (lifted conversion operator)。这个提升转换运算符执行从 S? 到 S 的解包，接着是从 S 到 T 的用户定义转换，然后是从 T 到 T? 的包装，null 值的 S? 直接转换为 null 值的 T? 除外。

提升的转换运算符与其基础用户定义转换运算符具有相同的隐式或显式类别。术语“用户定义的转换”适用于用户定义转换运算符和提升转换运算符的使用。

6.4.3 用户定义转换的计算

用户定义的转换将一个值从它所属的类型（称为源类型 (source type)）***e*** 转换为另一个类型（称为目标类型 (target type)）。用户定义的转换的计算集中在查找符合特定的源类型和目标类型的最精确的 (most specific) 用户定义转换运算符。此确定过程分为几个步骤：

- 查找考虑从中使用用户定义的转换运算符的类和结构集。此集由源类型及其基类和目标类型及其基类组成（隐式假定只有类和结构可以声明用户定义的运算符，并且不属于类的类型不具有任何基类）。为了执行本步骤，如果源类型或目标类型为 *nullable-type*，则改为使用它们的基础类型。
- 通过该类型集确定适用的用户定义转换运算符和提升转换运算符。一个转换运算符如满足下述条件就是适用的：必须可以通过执行标准转换（第 6.3 节）来使源类型转换为该运算符的操作数所要求的类型，并且必须可以通过执行标准转换来使运算符的结果类型转换为目标类型。
- 由适用的用户定义运算符集，明确地确定哪一个运算符是最精确的。一般而言，最精确的运算符是操作数类型“最接近”源类型并且结果类型“最接近”目标类型的运算符。用户定义的转换运算符比提升转换运算符优先级高。后面的章节定义了建立最精确的用户定义转换运算符的确切规则。

确定了最精确的用户定义转换运算符后，用户定义转换的实际执行包括三个步骤：

- 首先，如果需要，执行一个标准转换，将源类型转换为用户定义转换运算符或提升转换运算符的操作数所要求的类型。
- 然后，调用用户定义转换运算符或提升转换运算符来执行转换。
- 最后，如果需要，再执行一个标准转换，将用户定义转换运算符或提升转换运算符的结果类型转换为目标类型。

用户定义转换的计算从不涉及多个用户定义转换运算符或提升转换运算符。换言之，从 S 类型到 T 类型的转换决不会首先执行从 S 到 X 的用户定义转换，然后执行从 X 到 T 的用户定义转换。

后面的章节给出了用户定义的隐式或显式转换的确切定义。这些定义使用下面的术语：

- 如果存在从 A 类型到 B 类型的标准隐式转换（第 6.3.1 节），并且 A 和 B 都不是 *interface-type*，则称 A 被 B 包含 (encompassed by)、称 B 包含 (encompass) A。
- 如果存在从表达式 E 到类型 B 的标准隐式转换（第 6.3.1 节），并且 B 和 E 的类型（如果它有一个类型）都不是 *interface-type*，则称 E 被 B 包含 (encompassed by) 并称 B 包含 (encompass) E。
- 类型集包含程度最大的类型是该集中包含所有其他类型的类型。如果没有一个类型包含所有其他类型，则集中没有包含程度最大的类型。更直观地讲，包含程度最大的类型是集中的“最大”类型，每个其他类型均可隐式转换为该类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 在一个类型集中，被包含程度最大的类型是指这样一个类型：它被该类型集中的所有其他类型所包含。如果没有一个类型被所有其他类型包含，则集中没有被包含程度最大的类型。更直观地讲，被包含程度最大的类型是集中的“最小”类型，该类型可隐式转换为每个其他类型。

6.4.4 用户定义的隐式转换

从表达式 E 到类型 T 的用户定义的隐式转换按下面这样处理：

- 确定类型 S、S_0 和 T_0。
  - 如果 E 具有一个类型，则让 S 成为该类型。
  - 如果 S 或 T 是可为 null 的类型，则让 S_U 和 T_U 成为其基础类型，否则让 S_U 和 T_U 分别成为 S 和 T。
  - 如果 S_U 或 T_U 是类型形参，则让 S_0 和 T_0 成为其有效的基础类，否则让 S_0 和 T_0 分别成为 S_U 和 T_U。
- 查找类型集 D，将从该类型集考虑用户定义的转换运算符。此集由 S_0（如果 S_0 存在并且是类或结构）、S_0 的基类（如果 S_0 存在并且是类）和 T_0（如果 T_0 是类或结构）组成。仅当不存在到 D 集中已包含的另一种类型的标识转换时，才会向此集添加类型。
- 查找适用的用户定义转换运算符和提升转换运算符集 U。此集合由用户定义的隐式转换运算符和提升隐式转换运算符组成，这些运算符是在 D 中的类或结构内声明的，用于从包含 E 的类型转换为被 T 包含的类型。如果 U 为空，则转换未定义并且发生编译时错误。
- 在 U 中查找运算符的最精确的源类型 S_X：
  - 如果 S 存在并且 U 中的任何运算符都将从 S 转换，则 S_X 为 S。
  - 否则，S_X 在 U 中运算符的合并源类型集中是被包含程度最大的类型。如果无法恰好找到一个被包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
- 在 U 中查找运算符的最精确的目标类型 T_X：
  - 如果 U 中的任何运算符均转换为 T，则 T_X 为 T。
  - 否则，T_X 是 U 中运算符的合并目标类型集中包含程度最大的类型。如果无法恰好找到一个包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
- 查找最具体的转换运算符：
  - 如果 U 中只含有一个从 S_X 转换到 T_X 的用户定义转换运算符，则这就是最精确的转换运算符。
  - 否则，如果 U 恰好包含一个从 S_X 转换到 T_X 的提升转换运算符，则这就是最具体的转换运算符。
  - 否则，转换是不明确的，并发生编译时错误。
- 最后，应用转换：
  - 如果 E 尚不具有类型 S_X，则执行从 E 到 S_X 的标准隐式转换。
  - 调用最具体的转换运算符，以从 S_X 转换到 T_X。
  - 如果 T_X 不是 T，则执行从 T_X 到 T 的标准隐式转换。

如果存在从类型 S 的变量到 T 的用户定义的隐式转换，则存在从类型 S 到类型 T 的用户定义的隐式转换。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

6.4.5 用户定义的显式转换

从表达式 E 到类型 T 的用户定义的显式转换按下面这样处理：

- 确定类型 S、S_0 和 T_0。
  - 如果 E 具有一个类型，则让 S 成为该类型。
  - 如果 S 或 T 是可为 null 的类型，则让 S_U 和 T_U 成为其基础类型，否则让 S_U 和 T_U 分别成为 S 和 T。
  - 如果 S_U 或 T_U 是类型形参，则让 S_0 和 T_0 成为其有效的基础类，否则让 S_0 和 T_0 分别成为 S_U 和 T_U。
- 查找类型集 D，将从该类型集考虑用户定义的转换运算符。此集由 S_0（如果 S_0 存在并且是类或结构）、S_0 的基类（如果 S_0 存在并且是类）、T_0（如果 T_0 是类或结构）和 T_0 的基类（如果 T_0 是类）组成。仅当不存在到 D 集中已包含的另一种类型的标识转换时，才会向此集添加类型。
- 查找适用的用户定义转换运算符和提升转换运算符集 U。此集由用户定义的和提升的隐式或显式转换运算符组成，这些运算符是在 D 中的类或结构内声明的，用于从包含 E 或被 S（如果存在）包含的类型转换为包含 T 或被 T 包含的类型。如果 U 为空，则转换未定义并且发生编译时错误。
- 在 U 中查找运算符的最精确的源类型 S_X：
  - 如果 S 存在并且 U 中的任何运算符都将从 S 转换，则 S_X 为 S。
  - 否则，如果 U 中的任何运算符都从包含 E 的类型转换，则 S_X 是这些运算符的合并源类型集中被包含程度最大的类型。如果找不到被包含程度最大的类型，则转换是不明确的，并且会出现编译时错误。
  - 否则，S_X 在 U 中运算符的合并源类型集中是被包含程度最大的类型。如果无法恰好找到一个包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
- 在 U 中查找运算符的最精确的目标类型 T_X：
  - 如果 U 中的任何运算符均转换为 T，则 T_X 为 T。
  - 否则，如果 U 中的任何运算符都转换为被 T 包含的类型，则 T_X 是这些运算符的合并目标类型集中包含程度最大的类型。如果无法恰好找到一个包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
  - 否则，T_X 是 U 中运算符的合并目标类型集中包含程度最大的类型。如果找不到被包含程度最大的类型，则转换是不明确的，并且会出现编译时错误。
- 查找最具体的转换运算符：
  - 如果 U 中只含有一个从 S_X 转换到 T_X 的用户定义转换运算符，则这就是最精确的转换运算符。
  - 否则，如果 U 恰好包含一个从 S_X 转换到 T_X 的提升转换运算符，则这就是最具体的转换运算符。
  - 否则，转换是不明确的，并发生编译时错误。
- 最后，应用转换：
  - 如果 E 尚不具有类型 S_X，则执行从 E 到 S_X 的标准显式转换。
  - 调用最具体的用户定义转换运算符来执行从 S_X 到 T_X 转换。
  - 如果 T_X 不是 T，则执行从 T_X 到 T 的标准显式转换。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

如果存在从类型 S 类型的变量到 T 的用户定义的显式转换，则存在从类型 S 到类型 T 的用户定义的显式转换。

6.5 匿名函数转换

anonymous-method-expression 或 *lambda-expression* 被归类为匿名函数（第 7.15 节）。这个表达式没有类型，但是可隐式转换为兼容的委托类型或表达式目录树类型。具体而言，匿名函数 F 与委托类型 D 兼容：

- 如果 F 包含 *anonymous-function-signature*，则 D 与 F 的形参个数相同。
- 如果 F 不包含 *anonymous-function-signature*，则 D 可以有零个或多个任意类型的形参，只要 D 的任何形参都没有 out 形参修饰符。
- 如果 F 具有显式类型化形参列表，则 D 中的每个形参与 F 中的对应形参具有相同的类型和修饰符。
- 如果 F 具有隐式类型化形参列表，则 D 没有 ref 或 out 形参。
- 如果 F 的主体是一个表达式且 D 具有 void 返回类型，*或* F 是异步的且 D 具有返回类型 Task，则将 F 的每个参数均指定为 D 中对应参数的类型时，F 的主体是有效表达式（请参考第 7 章），该表达式将允许作为 *statement-expression*（第 8.6 节）。
- 如果 F 的主体是一个语句块且 D 具有 void 返回类型，*或* F 是异步的且 D 具有返回类型 Task，则将 F 的每个参数均指定为 D 中对应参数的类型时，F 的主体是有效语句块（请参考第 8.2 节），在该语句块中没有 return 语句指定了表达式。
- 在 F 的函数体为表达式时，如果 F 为非异步且 D 具有非空返回类型 T，或 F 为异步且 D 具有返回类型 Task<T>，则将 F 的每个参数均指定为 D 中对应参数的类型时，F 的函数体是有效表达式（请参考第 7 节），该表达式可隐式转换为 T。
- 在 F 的函数体为语句块时，如果 F 为非异步且 D 具有非空返回类型 T，或 F 为异步且 D 具有返回类型 Task<T>，则将 F 的每个参数均指定为 D 中对应参数的类型时，F 的函数体是一个带有不可到达的结束点的有效语句块（请参考第 8.2 节），在该语句块中每个 return 语句都指定一个可隐式转换为 T 的返回类型的表达式。

为简洁起见，本节使用任务类型的简写 Task 和 Task<T>（第 **错误!未找到引用源。** 节）。

如果 F 与委托类型 D 兼容，则 lambda 表达式 F 与表达式树类型 Expression<D> 兼容。注意，这一点不适用于匿名方法，而仅适用于 lambda 表达式

某些 lambda 表达式不能转换为表达式树类型：即使存在转换，该过程也会在编译时失败。lambda 表达式符合以下条件时发生上述情况：

- 具有 *block* 体
- 包含简单或复合赋值运算符
- 包含动态绑定表达式
- 是异步的

下面的示例使用一个用于表示函数的泛型委托类型 Func<A,R>，该函数采用一个类型为 A 的实参并返回一个类型为 R 的值：

```
delegate R Func<A,R>(A arg);
```

在下面的赋值中，

```
Func<int,int> f1 = x => x + 1;              // Ok
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
Func<int,double> f2 = x => x + 1;            // Ok
Func<double,int> f3 = x => x + 1;            // Error
Func<int, Task<int>> f4 = async x => x + 1;  // Ok
```

每个匿名函数的参数类型和返回类型均由匿名函数所赋给的变量的类型来决定。

第一个赋值将匿名函数成功转换为委托类型 Func<int,int>，因为当为 x 指定的类型是 int 时，x + 1 是一个可以隐式转换为类型 int 的有效表达式。

同样，第二个赋值将匿名函数成功转换为委托类型 Func<int,double>，因为 x + 1 所得的结果（类型为 int）可隐式转换为类型 double。

但是，第三个赋值会出现编译时错误，因为当为 x 指定的类型是 double 时，x + 1 所得的结果（类型为 double）不可隐式转换为类型 int。

第四个赋值将匿名异步函数成功转换为委托类型 Func<int, Task<int>>，因为 x + 1 所得的结果（类型为 int）可隐式转换为任务类型 Task<int> 的结果类型 int。

匿名函数可能会影响重载决策，并参与类型推断。请参见第 7.5 节，以了解有关的详细信息。

6.5.1 匿名函数转换为委托类型的计算

匿名函数转换为委托类型会生成一个委托实例，该实例引用匿名函数以及所捕获的、在计算时处于活动状态的外层变量的集（可能为空）。当调用委托时，将执行匿名函数体。使用委托引用的被捕获外层变量集执行方法体中的代码。

从匿名函数生成的委托的调用列表只包含一个项。该委托的确切目标对象和目标方法并未指定。具体而言，没有指定该委托的目标对象是 null、包容函数成员的 this 值，还是某个其他对象。

允许（但不要求）将具有相同的被捕获外层变量实例集（可能为空集）的语义上相同的匿名函数转换为相同的委托类型以返回相同的委托实例。此处所用的术语“语义上相同”表示，无论何种情况，只要给定相同的参数，匿名函数的执行就会产生相同的结果。此规则允许优化如下面这样的代码。

```
delegate double Function(double x);

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

   static void F(double[] a, double[] b) {
      a = Apply(a, (double x) => Math.Sin(x));
      b = Apply(b, (double y) => Math.Sin(y));
      ...
   }
}
```

由于两个匿名函数委托具有相同的被捕获外层变量集（空集），并且这两个匿名函数语义上相同，所以允许编译器使这两个委托引用同一个目标方法。实际上，允许编译器从这两个匿名函数表达式返回同一个委托实例。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

6.5.2 匿名函数转换为表达式树类型的计算

将匿名函数转换为表达式目录树类型会产生一个表达式目录树（第 4.6 节）。更确切地说，匿名函数转换的计算会导致构造对象结构，该结构表示匿名函数本身的结构。表达式树的精确结构以及创建该目录树的确切过程为定义的实现。

6.5.3 实现示例

本节从其他 C# 构造的角度描述可能的匿名函数转换实现方法。此处所描述的实现基于 Microsoft C# 编译器所使用的相同原理，但决非强制性的实现方式，也不是唯一可能的实现方式。本节仅简述到表达式树的转换，因为它们的准确语义超出了本规范的范围。

本节的其余部分提供了多个代码示例，其中包含具有不同特点的匿名函数。对于每个示例，提供了到仅使用其他 C# 构造的代码的相应转换。在这些示例中，假定标识符 D 表示下面的委托类型：

```
public delegate void D();
```

匿名函数的最简单形式是不捕获外层变量的形式：

```
class Test
{
   static void F() {
      D d = () => { Console.WriteLine("test"); };
   }
}
```

这可转换为引用编译器生成的静态方法的委托实例化，匿名函数的代码就位于该静态方法中：

```
class Test
{
   static void F() {
      D d = new D(__Method1);
   }

   static void __Method1() {
      Console.WriteLine("test");
   }
}
```

在下面的示例中，匿名函数引用 this 的实例成员：

```
class Test
{
   int x;

   void F() {
      D d = () => { Console.WriteLine(x); };
   }
}
```

这可转换为包含该匿名函数代码的、编译器生成的实例方法：

```
class Test
{
   int x;

   void F() {
      D d = new D(__Method1);
   }

   void __Method1() {
      Console.WriteLine(x);
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在此示例中，匿名函数捕获一个局部变量：

```
class Test
{
   void F() {
      int y = 123;
      D d = () => { Console.WriteLine(y); };
   }
}
```

局部变量的生存期现在必须至少延长为匿名函数委托的生存期。这可以通过将局部变量“提升”到编译器生成的类的字段来实现。之后，局部变量的实例化（第 7.15.5.2 节）对应于为编译器生成的类创建实例，而访问局部变量则对应于访问编译器生成的类的实例中的字段。而且，匿名函数将会成为编译器生成类的实例方法：

```
class Test
{
   void F() {
      __Locals1 __locals1 = new __Locals1();
      __locals1.y = 123;
      D d = new D(__locals1.__Method1);
   }

   class __Locals1
   {
      public int y;

      public void __Method1() {
         Console.WriteLine(y);
      }
   }
}
```

最后，下面的匿名函数捕获 this 以及两个具有不同生存期的局部变量：

```
class Test
{
   int x;

   void F() {
      int y = 123;
      for (int i = 0; i < 10; i++) {
         int z = i * 2;
         D d = () => { Console.WriteLine(x + y + z); };
      }
   }
}
```

此处，将为捕获局部变量的每一个语句块分别创建一个编译器生成的类，这样不同块中的局部变量可以有独立的生存期。__Locals2（对应于内层语句块的编译器生成类）的实例包含局部变量 z 和引用 __Locals1 的实例的字段。__Locals1（对应于外层语句块的编译器生成类）的实例包含局部变量 y 和引用包容函数成员的 this 的字段。使用这些数据结构，可以通过 __Local2 的实例访问所有被捕获外层变量，匿名函数的代码从而可以实现为该类的实例方法。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   void F() {
      __Locals1 __locals1 = new __Locals1();
      __locals1.__this = this;
      __locals1.y = 123;
      for (int i = 0; i < 10; i++) {
         __Locals2 __locals2 = new __Locals2();
         __locals2.__locals1 = __locals1;
         __locals2.z = i * 2;
         D d = new D(__locals2.__Method1);
      }
   }

   class __Locals1
   {
      public Test __this;
      public int y;
   }

   class __Locals2
   {
      public __Locals1 __locals1;
      public int z;

      public void __Method1() {
         Console.WriteLine(__locals1.__this.x + __locals1.y + z);
      }
   }
}
```

此处用于捕获局部变量的技术也可用于将匿名函数转换为表达式树：对编译器所生成对象的引用可以存储在表达式树中，对局部变量的访问可以表示为这些对象上的字段访问。这种方法的优点是允许“提升的”局部变量在委托和表达式树之间共享。

6.6 方法组转换

存在从方法组（第 7.1 节）到兼容委托类型的隐式转换（第 6.1 节）。对于给定的委托类型 D 和归类为方法组的表达式 E，如果下述条件成立则存在从 E 到 D 的隐式转换：E 至少包含一个方法，该方法能够以其正常形式（第 7.5.3.1 节）应用于使用 D 的形参类型和修饰符构造的实参列表，如下所述。

从方法组 E 转换到委托类型 D 的编译时应用在下面的部分中描述。注意，存在从 E 到 D 的隐式转换并不保证该转换的编译时应用会成功和不会出错。

- 对于 E(A) 形式的方法调用（第 7.6.5.1 节），仅选择一个方法 M，并进行以下修改：
  - 实参列表 A 是一个表达式列表，其中的每个表达式均可归类为一个变量并且具有 D 的 *formal-parameter-list* 中的对应形参的类型和修饰符（ref 或 out） – 类型 dynamic 的形参除外，其中对应的表达式具有类型 object 而不是 dynamic。
  - 所考虑的候选方法仅为那些可以其正常形式加以应用并且不会省略任何可选形参的方法（第 7.5.3.1 节）。因此，如果候选方法只能以其展开形式应用，或者它们的一个或多个可选形参在 D 中没有对应的形参，则将忽略这些方法。
- 如果第 7.6.5.1 节的算法产生与 D 具有相同形参个数的最佳方法 M，则将视作转换存在。
- 即使转换存在，如果选定的方法 M 与委托类型 D 不兼容（第 15.2 节），则将出现编译时错误。
- 如果选定的方法 M 是实例方法，则与 E 关联的实例表达式确定委托的目标对象。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果选定的方法 M 是通过实例表达式上的成员访问表示的扩展方法，则该实例表达式将确定委托的目标对象。
- 转换的结果是类型 D 的值，即一个引用选定方法和目标对象的新创建的委托。

请注意，在下述情形中此过程可能会导致创建到扩展方法的委托：第 7.6.5.1 节的算法未能找到实例方法，但在以扩展方法调用（第 7.6.5.2 节）的形式处理 E(A) 的调用时却取得成功。因此而创建的委托将捕获该扩展方法及其第一个参数。

下面的示例演示方法组转换：

```
delegate string D1(object o);
delegate object D2(string s);
delegate object D3();
delegate string D4(object o, params object[] a);
delegate string D5(int i);
class Test
{
   static string F(object o) {...}
   static void G() {
      D1 d1 = F;         // Ok
      D2 d2 = F;         // Ok
      D3 d3 = F;         // Error – not applicable
      D4 d4 = F;         // Error – not applicable in normal form
      D5 d5 = F;         // Error – applicable but not compatible
   }
}
```

对 d1 的赋值隐式将方法组 F 转换为 D1 类型的值。

对 d2 的赋值演示如何才能创建一个到具有派生程度较小（逆变）的形参类型和派生程度较大（协变）的返回类型的方法的委托。

对 d3 的赋值演示在方法不适用时如何不存在转换。

对 d4 的赋值演示方法必须如何以其正常形式应用。

对 d5 的赋值演示如何允许委托和方法的形参和返回类型仅对引用类型存在不同。

与所有其他隐式和显式转换一样，强制转换运算符可用于显式执行方法组转换。因此，示例

```
object obj = new EventHandler(myDialog.OkClick);
```

可改写为

```
object obj = (EventHandler)myDialog.OkClick;
```

方法组可能影响重载决策，并参与类型推断。请参见第 7.5 节，以了解有关的详细信息。

方法组转换的运行时计算如下进行：

- 如果在编译时选定的方法是一个实例方法，或者是一个以实例方法访问的扩展方法，则委托的目标对象由与 E 关联的实例表达式来确定：
  - 计算实例表达式。如果此计算导致异常，则不执行进一步的操作。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

  - 如果实例表达式为 *reference-type*，则由实例表达式计算的值将成为目标对象。如果选择的方法为实例方法，并且目标对象为 `null`，则引发 `System.NullReferenceException` 并且不执行进一步的步骤。
  - 如果实例表达式为 *value-type*，则执行装箱操作（第 4.3.1 节）以将值转换为对象，然后此对象将成为目标对象。
- 否则，选定的方法为静态方法调用的一部分，而委托的目标对象为 `null`。
- 为委托类型 `D` 的一个新实例分配存储位置。如果没有足够的可用内存来为新实例分配存储位置，则引发 `System.OutOfMemoryException`，并且不执行进一步的操作。
- 用对在编译时确定的方法的引用和对上面计算的目标对象的引用初始化新委托实例。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7. 表达式

表达式是一个运算符和操作数的序列。本章定义语法、操作数和运算符的计算顺序以及表达式的含义。

7.1 表达式的分类

一个表达式可归类为下列类别之一：

- 值。每个值都有关联的类型。
- 变量。每个变量都有关联的类型，称为该变量的已声明类型。
- 命名空间。归为此类的表达式只能出现在 *member-access*（第 7.6.4 节）的左侧。在任何其他上下文中，归类为命名空间的表达式将导致编译时错误。
- 类型。归为此类的表达式只能出现在 *member-access*（第 7.6.4 节）的左侧，或作为 `as` 运算符（第 7.10.11 节）、`is` 运算符（第 REF7.10.10 节）或 `typeof` 运算符（第 7.6.11 节）的操作数。在任何其他上下文中，归类为类型的表达式将导致编译时错误。
- 方法组。它是一组重载方法，是成员查找（第 7.4 节）的结果。方法组可能具有关联的实例表达式和关联的类型实参列表。当调用实例方法时，实例表达式的计算结果成为由 `this`（第 7.6.7 节）表示的实例。在 *invocation-expression*（第 7.6.5 节）和 *delegate-creation-expression*（第 7.6.10.5 节）中允许使用方法组，且这两种表达式的左边均为运算符，可以隐式转换为兼容的委托类型（第 6.6 节）。在任何其他上下文中，归类为方法组的表达式将导致编译时错误。
- null 文本。归类为 null 文本的表达式可以隐式转换为引用类型或可以为 null 的类型。
- 匿名函数。归类为匿名函数的表达式可以隐式转换为兼容的委托类型或表达式目录树类型。
- 属性访问。每个属性访问都有关联的类型，即该属性的类型。此外，属性访问可以有关联的实例表达式。当调用实例属性访问的访问器（`get` 或 `set` 块）时，实例表达式的计算结果将成为由 `this`（第 7.6.7 节）表示的实例。
- 事件访问。每个事件访问都有关联的类型，即该事件的类型。此外，事件访问还可以有关联的实例表达式。事件访问可作为 `+=` 和 `-=` 运算符（第 7.17.3 节）的左操作数出现。在任何其他上下文中，归类为事件访问的表达式将导致编译时错误。
- 索引器访问。每个索引器访问都有关联的类型，即该索引器的元素类型。此外，索引器访问还可以有关联的实例表达式和关联的参数列表。当调用索引器访问的访问器（`get` 或 `set` 块）时，实例表达式的计算结果将成为由 `this`（第 7.6.7 节）表示的实例，而实参列表的计算结果将成为调用的形参列表。
- Nothing。这出现在当表达式是调用一个具有 `void` 返回类型的方法时。归类为 Nothing 的表达式仅在 *statement-expression*（第 8.6 节）的上下文中有效。

表达式的最终结果绝不会是一个命名空间、类型、方法组或事件访问。恰如以上所述，这些类别的表达式是只能在特定上下文中使用的中间构造。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

通过执行对 *get-accessor* 或 *set-accessor* 的调用，属性访问或索引器访问总是被重新归类为值。该特殊访问器由属性或索引器访问的上下文确定：如果访问是赋值的目标，则通过调用 *set-accessor* 来赋新值（第 7.17.1 节）。否则，通过调用 *get-accessor* 来获取当前值（第 7.1.1 节）。

7.1.1 表达式的值

大多数含有表达式的构造最后都要求表达式表示一个值 (value)。在此情况下，如果实际的表达式表示命名空间、类型、方法组或Nothing，则将发生编译时错误。但是，如果表达式表示属性访问、索引器访问或变量，则将它们隐式替换为相应的属性、索引器或变量的值：

- 变量的值只是当前存储在该变量所标识的存储位置的值。必须先将变量视为已明确赋值（第 5.3 节）才可以获取其值，否则将出现编译时错误。
- 通过调用属性的 *get-accessor* 可获取属性访问表达式的值。如果属性没有 *get-accessor*，则会出现编译时错误。否则将执行函数成员调用（第 7.5.4 节），然后调用结果将成为属性访问表达式的值。
- 通过调用索引器的 *get-accessor* 可获取索引器访问表达式的值。如果索引器没有 *get-accessor*，则会出现编译时错误。否则，将使用与索引器访问表达式关联的参数列表来执行函数成员调用（第 7.5.4 节）然后调用结果将成为索引器访问表达式的值。

7.2 静态和动态绑定

根据构成表达式（参数、操作数、接收器）的类型或值确定操作含义的过程通常称为绑定。例如，方法调用的含义是根据接收器和参数的类型确定的。运算符的含义是根据其操作数的类型确定的。

在 C# 中，操作的含义通常在编译时根据其构成表达式的编译时类型确定。同样，如果表达式包含错误，编译器将检测并报告该错误。此方法称为静态绑定。

但是，如果表达式为动态表达式（即类型为 `dynamic`），则这指示它所参与的任何绑定都应基于其运行时类型（即它在运行时所表示的对象的实际类型），而不是它在编译时的类型。因此，此类操作的绑定推迟到要在程序运行过程中执行此操作的时间。这称为动态绑定 (dynamic binding)。

当操作是动态绑定时，编译器只执行很少检查或根本不执行检查。而当运行时绑定失败时，错误将在运行时报告为异常。

C# 中的以下操作会进行绑定：

- 成员访问：`e.M`
- 调用方法：$e.M(e_1,\cdots,e_n)$
- 委托调用：$e(e_1,\cdots,e_n)$
- 元素访问：$e[e_1,\cdots,e_n]$
- 对象创建：$\texttt{new } C(e_1,\cdots,e_n)$
- 重载的一元运算符：`+`、`-`、`!`、`~`、`++`、`--`、`true`、`false`
- 重载的二元运算符：`+`、`-`、`*`、`/`、`%`、`&`、`&&`、`|`、`||`、`??`、`^`、`<<`、`>>`、`==`、`!=`、`>`、`<`、`>=`、`<=`
- 赋值运算符：`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`<<=`、`>>=`
- 隐式转换和显式转换

不涉及动态表达式时，C# 默认为静态绑定，这表示在选择过程中使用构成表达式的编译时类型。但是，当上面列出的操作中的构成表达式之一为动态表达式时，操作会改为动态绑定。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.2.1 绑定时间

静态绑定在编译时进行，而动态绑定在运行时进行。在以下各节中，术语绑定时间(binding-time) 指编译时或运行时，具体取决于进行绑定的时间。

下面的示例演示静态绑定和动态绑定的表示法以及绑定时间的表示法：

```
object o  = 5;
dynamic d = 5;

Console.WriteLine(5); //  static binding to Console.WriteLine(int)
Console.WriteLine(o); //  static binding to Console.WriteLine(object)
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)
```

前两个调用是静态绑定的：Console.WriteLine 的重载是基于其参数的编译时类型选择的。因此，绑定时间为编译时。

第三个调用是动态绑定的：Console.WriteLine 的重载是基于其参数的运行时类型选择的。出现这种情况是因为参数为动态表达式（其编译时类型为 dynamic）。因此，第三个调用的绑定时间为运行时。

7.2.2 动态绑定

动态绑定的用途是允许 C# 程序与动态对象（dynamic object，即不遵循 C# 类型系统的一般规则的对象）进行交互。动态对象可以是来自具有不同类型系统的其他编程语言的对象，也可以是以编程方式设置为针对不同操作实现其自己的绑定语义的对象。

动态对象用于实现其自己语义的机制由实现定义。动态对象实现给定接口（再次定义的实现），以便向 C# 运行时发送信号，指示这些对象具有特殊语义。因此，只要对动态对象的操作是动态绑定的，就将采用其自己的绑定语义，而不是本文档中指定的 C# 绑定语义。

尽管动态绑定的用途是允许与动态对象进行互操作，然而 C# 允许对所有对象（无论是否为动态对象）进行动态绑定。这允许更加顺畅地集成动态对象，因为虽然对这些对象进行的操作的结果本身可能不是动态对象，但仍是程序员在编译时未知的类型。即使所涉及的对象都不是动态对象，动态绑定也有助于消除易于出错的基于反射的代码。

以下各节对于语言中的每种构造，介绍了应用动态绑定的确切时间、应用何种编译时检查（如果有）以及编译时结果和表达式分类是什么。

7.2.3 构成表达式的类型

当操作静态绑定时，构成表达式的类型（例如，接收器、实参、索引或操作数）通常视为该表达式的编译时类型。

当操作动态绑定时，构成表达式的类型由不同的方式确定，具体取决于构成表达式的编译时类型：

- 编译时类型为 dynamic 的构成表达式视为具有该表达式在运行时计算的实际值的类型
- 编译时类型为类型形参的构成表达式视为有类型形参在运行时绑定到的类型
- 否则，构成表达式视为具有其编译时类型

7.3 运算符

表达式由操作数 (operand) 和运算符 (operator) 构成。表达式的运算符指示对操作数适用什么样的运算。运算符的示例包括+、-、*、/ 和 new。操作数的示例包括文本、字段、局部变量和表达式。

有三类运算符：

- 一元运算符。一元运算符带一个操作数并使用前缀表示法（如 -x）或后缀表示法（如 x++）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 二元运算符。二元运算符带两个操作数并且全都使用中缀表示法（如 `x + y`）。
- 三元运算符。只存在一个三元运算符 `?:`，它带三个操作数并使用中缀表示法 (`c? x: y`)。

表达式中运算符的计算顺序由运算符的优先级 (precedence) 和关联性 (associativity)（第 7.3.1 节）决定。

表达式中的操作数从左到右进行计算。例如，在 `F(i) + G(i++) * H(i)` 中，`F` 方法是使用 `i` 的旧值调用的，然后 `G` 方法也是使用 `i` 的旧值进行调用，最后 `H` 方法使用 `i` 的新值调用。这与运算符的优先级无关。

某些运算符可以重载 (overloaded)。运算符重载允许指定用户定义的运算符实现来执行某些运算，这些运算的操作数中至少有一个，甚至两个都属于用户定义的类或结构类型（第 7.3.2 节）。

7.3.1 运算符的优先级和顺序关联性

当表达式包含多个运算符时，运算符的优先级 (precedence) 控制各运算符的计算顺序。例如，表达式 `x + y * z` 按 `x + (y * z)` 计算，因为 `*` 运算符的优先级高于二元 `+` 运算符。运算符的优先级由运算符的关联语法产生式的定义确定。例如，*additive-expression* 由以 `+` 或 `+` 或 `-` 运算符分隔的 *multiplicative-expression* 序列组成，因而 `+` 和 `-` 运算符的优先级比 `*`、`/` 和 `%` 运算符要低。

下表按照从最高到最低的优先级顺序概括了所有的运算符：

章节	类别	运算符
7.6	基本	`x.y  f(x)  a[x]  x++  x--  new` `typeof  default  checked  unchecked  delegate`
7.7	一元	`+  -  !  ~  ++x  --x  (T)x  await x`
7.8	乘法	`*  /  %`
7.8	加减	`+  -`
7.9	移位	`<<  >>`
7.10	关系和类型检测	`<  >  <=  >=  is  as`
7.10	相等	`==  !=`
7.11	逻辑 AND	`&`
7.11	逻辑 XOR	`^`
7.11	逻辑 OR	`|`
7.12	条件 AND	`&&`
7.12	条件 OR	`||`
7.13	null 合并	`??`
7.14	条件	`?:`
7.17, 7.15	赋值和 lambda 表达式	`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=` `=>`

当操作数出现在具有相同优先级的两个运算符之间时，运算符的顺序关联性控制运算的执行顺序：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 除赋值运算符和 null 合并运算符外，所有二元运算符均为左结合，表示从左向右执行运算。例如，x + y + z 可以按 (x + y) + z 进行计算。
- 赋值运算符、null 合并运算符和条件运算符 (?:) 为右结合，表示从右向左执行运算。例如，x = y = z 可以按 x = (y = z)进行计算。

优先级和顺序关联性都可以用括号控制。例如，x + y * z 先将 y 乘以 z，然后将结果与 x 相加，而 (x + y) * z 先将 x 与 y 相加，然后再将结果乘以 z。

7.3.2 运算符重载

所有一元和二元运算符都具有可自动用于任何表达式的预定义实现。除了预定义实现外，还可通过在类或结构（第 10.10 节）中包括 operator 声明来引入用户定义的实现。用户定义的运算符实现的优先级始终高于预定义运算符实现的优先级：仅当不存在适用的用户定义运算符实现时才考虑预定义的运算符实现，如第 7.3.3 节和第 7.3.4 节中所述。

可重载的一元运算符 (overloadable unary operator) 有：

```
+   -   !   ~   ++   --   true   false
```

虽然不在表达式中显式使用 true 和 false（因而未包括在第 7.3.1 节的优先级表中），但仍将它们视为运算符，原因是它们在多种表达式上下文中被调用：布尔表达式（第 7.20 节）以及涉及条件（第 7.14 节）运算符和条件逻辑运算符（第 7.12 节）的表达式。

可重载的二元运算符 (overloadable binary operator) 有：

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

只有以上所列的运算符可以重载。具体而言，不可能重载成员访问、方法调用或 =、&&、||、??、?:、=>、checked、unchecked、new、typeof、default、as 和 s 运算符。

当重载一个二元运算符时，也会隐式重载相应的赋值运算符（若有）。例如，运算符 * 的重载也是运算符 *= 的重载。第 7.17.2 节对此有进一步描述。请注意，赋值运算符本身 (=) 不能重载。赋值总是简单地将值按位复制到变量中。

强制转换运算（如 (T)x）通过提供用户定义的转换（第 6.4 节）来重载。

元素访问（如 a[x]）不被视为可重载的运算符。但是，可通过索引器（第 10.9 节）支持用户定义的索引。

在表达式中，使用运算符表示法来引用运算符，而在声明中，使用函数表示法来引用运算符。下表显示了一元运算符和二元运算符的运算符表示法和函数表示法之间的关系。在第一项中，*op* 表示任何可重载的一元前缀运算符。在第二项中，*op* 表示 ++ 和 -- 一元后缀运算符。在第三项中，*op* 表示任何可重载的二元运算符。

运算符表示法	函数表示法
op x	operator *op*(x)
x *op*	operator *op*(x)
x *op* y	operator *op*(x, y)

用户定义的运算符声明总是要求至少一个参数为包含运算符声明的类或结构类型。因此，用户定义的运算符不可能具有与预定义运算符相同的签名。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

用户定义的运算符声明不能修改运算符的语法、优先级或顺序关联性。例如，/ 运算符始终为二元运算符，始终具有在第 7.3.1 节中指定的优先级，并且始终左结合。

虽然用户定义的运算符可以执行它想执行的任何计算，但是强烈建议不要采用产生的结果与直觉预期不同的实现。例如，operator == 的实现应比较两个操作数是否相等，然后返回一个适当的 bool 结果。

在从第 7.6 节到第 7.12 节的关于各运算符的说明中，运算符的预定义实现以及适用于各运算符的任何其他规则都有规定。在这些说明中使用了“一元运算符重载决策”(unary operator overload resolution)、“二元运算符重载决策”(binary operator overload resolution) 和“数值提升”(numeric promotion) 这样的术语，在后面的章节中可以找到它们的定义。

7.3.3 一元运算符重载决策

op x 或 x *op* 形式的运算（其中 *op* 是可重载一元运算符，x 是 X 类型的表达式）按如下方式处理：

- 对于由 X 为运算 operator *op*(x) 提供的候选的用户定义运算符集，应根据第 7.3.5 节中的规则来确定。
- 如果候选的用户定义运算符集不为空，则它就会成为运算的候选运算符集。否则，预定义一元 operator *op* 实现（包括它们的提升形式）将成为关于该运算的候选运算符集。关于给定运算符的预定义实现，在有关运算符的说明（第 7.6 节和第 7.7 节）中指定。
- 第 7.5.3 节中的重载决策规则应用于候选运算符集，以选择一个关于参数列表 (x) 的最佳运算符，此运算符将成为重载决策过程的结果。如果重载决策未能选出单个最佳运算符，则发生绑定时错误。

7.3.4 二元运算符重载决策

x *op* y 形式的运算（其中 *op* 是可重载的二元运算符，x 是 X 类型的表达式，y 是 Y 类型的表达式）按如下方式处理：

- 确定 X 和 Y 为运算 operator *op*(x, y) 提供的候选用户定义运算符集。该集包括由 X 提供的候选运算符和由 Y 提供的候选运算符的并集，每个候选运算符都使用第 7.3.5 节中的规则来确定。对于并集，将按如下方式合并候选项：
  - 如果 X 和 Y 为同一类型，或者 X 和 Y 派生自一个公共基类型，则两者共有的候选运算符只在该并集中出现一次。
  - 如果 X 和 Y 之间存在标识转换，Y 提供的运算符 op_Y 与 X 提供的 op_X 具有相同的返回类型，并且 op_Y 的操作数类型具有到 op_X 的对应操作数类型的标识转换，则该集中仅出现 op_X。
- 如果候选的用户定义运算符集不为空，则它就会成为运算的候选运算符集。否则，预定义二元 operator *op* 实现（包括它们的提升形式）将成为关于该运算的候选运算符集。关于给定运算符的预定义实现，在有关运算符的说明（第 7.8 节到第 7.12 节）中指定。对于预定义的枚举和委托运算符而言，所考虑的唯一运算符是那些由枚举或委托类型（即其中一个操作数的绑定时间类型）定义的运算符。
- 第 7.5.3 节中的重载决策规则应用于候选运算符集，以选择一个关于参数列表 (x, y) 的最佳运算符，此运算符将成为重载决策过程的结果。如果重载决策未能选出单个最佳运算符，则发生绑定时错误。

7.3.5 候选用户定义运算符

给定一个 T 类型和运算 operator *op*(A)，其中 *op* 是可重载的运算符，A 是参数列表，对 T 为 operator *op*(A) 提供的候选用户定义运算符集按如下方式确定：

- 确定类型 T_0。如果 T 是可以为 null 的类型，则 T_0 是其基础类型；否则 T_0 等于 T。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 对于 T_0 中的所有 operator *op* 声明和此类运算符的提升形式，如果关于参数列表 A 至少有一个运算符是适用的（第 7.5.3.1 节），则候选运算符集将由 T_0 中所有适用的此类运算符组成。
- 否则，如果 T_0 为 object，则候选运算符集为空。
- 否则，T_0 提供的候选运算符集为 T_0 的直接基类提供的候选运算符集，或者为 T_0 的有效基类（如果 T_0 为类型参数）。

7.3.6 数值提升

数值提升包括自动为预定义一元和二元数值运算符的操作数执行某些隐式转换。数值提升不是一个独特的机制，而是一种将重载决策应用于预定义运算符所产生的效果。数值提升尤其不影响用户定义运算符的计算，尽管可以实现用户定义运算符以表现类似的效果。

作为数值提升的示例，请看二元运算符 * 的预定义实现：

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

当重载决策规则（第 7.5.3 节）应用于此运算符集时，这些运算符中第一个能满足下述条件的运算符将被选中：存在从操作数类型的隐式转换。例如，对于运算 b * s（其中 b 为 byte，s 为 short），重载决策将选择 operator *(int, int) 作为最佳运算符。因此，效果是 b 和 s 转换为 int，并且结果的类型为 int。同样，对于运算 i * d（其中 i 为 int，d 为 double），重载决策将选择 operator *(double, double) 作为最佳运算符。

7.3.6.1 一元数值提升

一元数值提升是针对预定义的 +、 - 和 ~ 一元运算符的操作数发生的。一元数值提升仅包括将 sbyte、byte、short、ushort 或 char 类型的操作数转换为 int 类型。此外，对于 - 一元运算符，一元数值提升将 uint 类型的操作数转换为 long 类型。

7.3.6.2 二元数值提升

二元数值提升是针对预定义的 +、 -、 *、 /、 %、 &、 |、 ^、 ==、 !=、 >、 <、 >= 和 <= 二元运算符的操作数发生的。二元数值提升隐式地将两个操作数都转换为一个公共类型，如果涉及的是非关系运算符，则此公共类型还成为运算的结果类型。二元数值提升应按下列规则进行（以它们在此出现的顺序）：

- 如果有一个操作数的类型为 decimal，则另一个操作数转换为 decimal 类型；否则，如果另一个操作数的类型为 float 或 double，则发生绑定时错误。
- 否则，如果有一个操作数的类型为 double，则另一个操作数转换为 double 类型。
- 否则，如果有一个操作数的类型为 float，则另一个操作数转换为 float 类型。
- 否则，如果有一个操作数的类型为 ulong，则另一个操作数将转换为 ulong 类型；否则，如果另一个操作数的类型为 sbyte、short、int 或 long，则将发生绑定时错误。
- 否则，如果有一个操作数的类型为 long，则另一个操作数转换为 long 类型。
- 否则，如果有一个操作数的类型为 uint，而另一个操作数的类型为 sbyte、short 或 int，则两个操作数均将转换为 long 类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，如果有一个操作数的类型为 uint，则另一个操作数转换为 uint 类型。
- 否则，两个操作数都转换为 int 类型。

请注意，第一个规则不允许将 decimal 类型与 double 和 float 类型混用。该规则遵循这样的事实：在 decimal 类型与 double 和 float 类型之间不存在隐式转换。

还需要注意的是，当一个操作数为有符号的整型时，另一个操作数的类型不可能为 ulong 类型。原因是不存在一个既可以表示 ulong 的全部范围，又能表示有符号整数的整型类型。

在以上两种情况下，都可以使用强制转换表达式显式地将一个操作数转换为与另一个操作数兼容的类型。

在下面的示例中

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

由于 decimal 类型不能与 double 类型相乘，因此发生绑定时错误。通过将第二个操作数显式转换为 decimal 消除此错误，如下所示：

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

7.3.7 提升运算符

提升运算符 (lifted operator) 允许操作不可以为 null 的值类型的预定义运算符及用户定义运算符，亦可用于这些类型的可以为 null 的形式。提升运算符是根据符合某些要求的预定义和用户定义运算符构造而成的，如下所述：

- 对于一元运算符

  ```
  +   ++   -   --   !   ~
  ```

  如果操作数和结果类型都为不可以为 null 的值类型，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到操作数和结果类型构造而成的。如果操作数为 null，则提升运算符产生一个 null 值。否则，提升运算符对该操作数进行解包，应用基础运算符，并包装结果。

- 对于二元运算符

  ```
  +   -   *   /   %   &   |   ^   <<   >>
  ```

  如果操作数和结果类型都为不可以为 null 的值类型，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到每个操作数和结果类型构造的。如果一个操作数为 null 或两个操作数皆为 null，则提升运算符产生一个 null 值（bool? 类型的 & 和 | 运算符除外，如第 7.11.3 节所述）。否则，提升运算符对这些操作数进行解包，应用基础运算符，并包装结果。

- 对于相等运算符

  ```
  ==   !=
  ```

  如果两个操作数类型都为不可以为 null 的值类型，并且结果类型为 bool，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到每个操作数类型构造的。该提升运算符认为两个 null 值相等，null 值不等于任何非 null 值。如果两个操作数都为非 null，则提升运算符对这两个操作数进行解包，并应用基础运算符以产生 bool 结果。

- 对于关系运算符

  ```
  <   >   <=   >=
  ```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

如果两个操作数类型都为不可以为 null 的值类型，并且结果类型为 bool，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到每个操作数类型构造的。如果一个操作数为 null 或两个操作数都为 null，则提升运算符产生 false 值。否则，提升运算符对这些操作数进行解包，并应用基础运算符以产生 bool 结果。

7.4 成员查找

成员查找成员查找是确定类型上下文中的名称含义的过程。成员查找可以作为表达式中计算 *simple-name*（第 7.6.2 节）或 *member-access*（第 7.6.4 节）的过程的一部分进行。如果 *simple-name* 或 *member-access* 以 *invocation-expression*（第 7.6.5.1 节）的 *primary-expression* 形式出现，则称调用该成员。

如果成员是方法或事件，或者如果成员是委托类型（第 15 章）或 dynamic 类型（第 4.7 节）的常量、字段或属性，则称该成员是可以调用的。

成员查找不仅考虑成员的名称，而且考虑该成员具有的类型形参的数目以及该成员是否可访问。对成员查找来说，泛型方法和嵌套泛型类型具有的类型形参数目就是在它们各自的声明中所指定的数目，其他所有成员则具有零个类型形参。

类型 T 中的具有 K 个类型形参的名称 N 的成员查找过程如下：

- 首先确定名为 N 的可访问的成员的集：
  - 如果 T 是类型形参，则该集是被指定为 T 的主要约束或次要约束（第 10.1.5 节）的每个类型中名为 N 的可访问成员集与 object 中名为 N 的可访问成员集的并集。
  - 否则，该集由 T 中所有名为 N 的可访问（第 3.5 节）成员（包括继承的成员）和 object 中名为 N 的可访问成员构成。如果 T 为构造类型，则按第 10.3.2 节中所述通过替换类型实参来获取成员集。包含 override 修饰符的成员不包括在此集中。
- 下一步，如果 K 为零，则移除声明中包含类型形参的所有嵌套类型。如果 K 不为零，则移除所有具有不同数目的类型形参的成员。注意，当 K 为零时，将不会移除具有类型形参的方法，因为类型推断过程（第 7.5.2 节）也许能够推断出类型实参。
- 接着，如果调用该成员，则从该集中移除所有不可调用的成员。
- 然后，从该集中移除被其他成员隐藏的成员。对于该集中的每个成员 S.M（其中 S 是声明了成员 M 的类型），应用下面的规则：
  - 如果 M 是一个常量、字段、属性、事件或枚举成员，则从该集中移除在 S 的基类型中声明的所有成员。
  - 如果 M 是一个类型声明，则从该集中移除在 S 的基类型中声明的所有非类型，并从该集中移除与在 S 的基类型中声明的 M 具有相同数目的类型形参的所有类型声明。
  - 如果 M 是方法，则从该集移除在 S 的基类型中声明的所有非方法成员。
- 然后，从该集中移除被类成员隐藏的接口成员。仅当 T 为类型形参，并且 T 同时具有除 object 以外的有效基类和非空有效接口集（第 10.1.5 节）时，此步骤才会产生效果。对于该集中的每个成员 S.M（其中 S 是声明了成员 M 的类型），如果 S 是除 object 以外的类声明，则应用下面的规则：
  - 如果 M 是一个常量、字段、属性、事件、枚举成员或类型声明，则从该集中移除在接口声明中声明的所有成员。
  - 如果 M 是一个方法，则从该集中移除在接口声明中声明的所有非方法成员，并从该集中移除与在接口声明中声明的 M 具有相同签名的所有方法。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 最后，移除了隐藏成员后，按下述规则确定查找结果：
  - 如果该集由单个非方法成员组成，则此成员即为查找的结果。
  - 否则，如果该集只包含方法，则这组方法为查找的结果。
  - 否则，该查找是不明确的，将会发生绑定时错误。

对于非类型形参和接口的类型中的成员查找，以及严格单一继承的接口（继承链中的每个接口都只有零个或一个直接基接口）中的成员查找，这些查找规则的效果就相当于派生成员隐藏具有相同名称或签名的基成员。这种单一继承查找决不会产生多义性。有关多重继承接口中的成员查找可能引起的多义性的介绍详见第 13.2.5 节。

7.4.1 基类型

出于成员查找的目的，类型 `T` 被视为具有下列基类型：

- 如果 `T` 为 `object` 或 `dynamic`，则 `T` 没有基类型。
- 如果 `T` 为 *enum-type*，则 `T` 的基类型为类类型 `System.Enum`、`System.ValueType` 和 `object`。
- 如果 `T` 为 *struct-type*，则 `T` 的基类型为类类型 `System.ValueType` 和 `object`。
- 如果 `T` 为 *class-type*，则 `T` 的基类型为 `T` 的基类，其中包括类类型 `object`。
- 如果 `T` 为 *interface-type*，则 `T` 的基类型为 `T` 的基接口和类类型 `object`。
- 如果 `T` 为 *array-type*，则 `T` 的基类型为类类型 `System.Array` 和 `object`。
- 如果 `T` 为 *delegate-type*，则 `T` 的基类型为类类型 `System.Delegate` 和 `object`。

7.5 函数成员

函数成员是包含可执行语句的成员。函数成员总是类型的成员，不能是命名空间的成员。C# 定义了以下类别的函数成员：

- 方法
- 属性
- 事件
- 索引器
- 用户定义运算符
- 实例构造函数
- 静态构造函数
- 析构函数

除了析构函数和静态构造函数（它们不能被显式调用），函数成员中包含的语句通过函数成员调用执行。编写函数成员调用的实际语法取决于具体的函数成员类别。

函数成员调用中所带的实参列表（第 7.5.1 节）为函数成员的形参提供实际值或变量引用。

泛型方法的调用可能会使用类型推断确定要传递到方法的类型实参集。有关此过程的介绍详见第 7.5.2 节。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

调用方法、索引器、运算符和实例构造函数时，使用重载决策来确定要调用的候选函数成员集。有关此过程的介绍详见第 7.5.3 节。

在绑定时（可能通过重载决策）确定了具体的函数成员后，有关运行时调用函数成员的实际过程的介绍详见第 7.5.4 节。

下表概述了在涉及六个可被显式调用的函数成员类别的构造中发生的处理过程。在下表中，e、x、y 和 value 代表变量或值类别的表达式，T 代表类型的表达式，F 是一个方法的简单名称，P 是一个属性的简单名称。

构造	示例	说明
方法调用	F(x, y)	应用重载决策以在包含类或结构中选择最佳的方法 F。以参数列表 (x, y) 调用该方法。如果该方法不为 static，则用 this 来表示对应的实例。
	T.F(x, y)	应用重载决策以在类或结构 T 中选择最佳的方法 F。如果该方法不为 static，则发生绑定时错误。以参数列表 (x, y) 调用该方法。
	e.F(x, y)	应用重载决策以在 e 的类型给定的类、结构或接口中选择最佳方法 F。如果该方法为 static，则发生绑定时错误。用实例表达式 e 和参数列表 (x, y) 调用该方法。
属性访问	P	调用包含类或结构中的属性 P 的 get 访问器。如果 P 是只写的，则发生编译时错误。如果 P 不为 static，则用 this 来表示对应的实例。
	P = value	用参数列表 (value) 调用包含类或结构中的属性 P 的 set 访问器。如果 P 是只读的，则发生编译时错误。如果 P 不为 static，则用 this 来表示对应的实例。
	T.P	调用包含类或结构 T 中的属性 P 的 get 访问器。如果 P 不为 static，或者 P 是只写的，则发生编译时错误。
	T.P = value	用参数列表 (value) 调用包含类或结构 T 中的属性 P 的 set 访问器。如果 P 不为 static，或者 P 是只读的，则发生编译时错误。
	e.P	用实例表达式 e 调用由 e 的类型给定的类、结构或接口中属性 P 的 get 访问器。如果 P 为 static，或者 P 是只写的，则发生绑定时错误。
	e.P = value	用实例表达式 e 和参数列表 (value) 调用 e 的类型给定的类、结构或接口中属性 P 的 set 访问器。如果 P 为 static，或者 P 是只读的，则发生绑定时错误。
事件访问	E += value	调用包含类或结构中的事件 E 的 add 访问器。如果 E 不是静态的，则用 this 来表达对应的实例。
	E -= value	调用包含类或结构中的事件 E 的 remove 访问器。如果 E 不是静态的，则用 this 来表达对应的实例。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

构造	示例	说明
	`T.E += value`	调用包含类或结构 `T` 中的事件 `E` 的 `add` 访问器。如果 `E` 不是静态的，则发生绑定时错误。
	`T.E -= value`	调用包含类或结构 `T` 中的事件 `E` 的 `remove` 访问器。如果 `E` 不是静态的，则发生绑定时错误。
	`e.E += value`	用实例表达式 `e` 调用由 `e` 的类型给定的类、结构或接口中事件 `E` 的 `add` 访问器。如果 `E` 是静态的，则发生绑定时错误。
	`e.E -= value`	用实例表达式 `e` 调用由 `e` 的类型给定的类、结构或接口中事件 `E` 的 `remove` 访问器。如果 `E` 是静态的，则发生绑定时错误。
索引器访问	`e[x, y]`	应用重载决策以在 `e` 的类型给定的类、结构或接口中选择最佳的索引器。用实例表达式 `e` 和参数列表 `(x, y)` 调用该索引器的 `get` 访问器。如果索引器是只写的，则发生绑定时错误。
	`e[x, y] = value`	应用重载决策以在 `e` 的类型给定的类、结构或接口中选择最佳的索引器。用实例表达式 `e` 和参数列表 `(x, y, value)` 调用该索引器的 `set` 访问器。如果索引器是只读的，则发生绑定时错误。
运算符调用	`-x`	应用重载决策以在 `x` 的类型给定的类或结构中选择最佳的一元运算符。用参数列表 `(x)` 调用选定的运算符。
	`x + y`	应用重载决策以在 `x` 和 `y` 的类型给定的类或结构中选择最佳的二元运算符。用参数列表 `(x, y)` 调用选定的运算符。
实例构造函数调用	`new T(x, y)`	应用重载决策以在类或结构 `T` 中选择最佳的实例构造函数。用参数列表 `(x, y)` 调用该实例构造函数。

7.5.1 实参列表

每个函数成员和委托调用均包括一个实参列表，其中列出函数成员形参的实际值或变量引用。用于指定函数成员调用的实参列表的语法取决于函数成员类别：

- 对于实例构造函数、方法、索引器和委托，将实参指定为 *argument-list*，如下所述。对于索引器，当调用 `set` 访问器时，实参列表还需附加上一个表达式，该表达式被指定为赋值运算符的右操作数。
- 对于属性，当调用 `get` 访问器时，实参列表是空的；而当调用 `set` 访问器时，实参列表由指定为赋值运算符的右操作数的表达式组成。
- 对于事件，实参列表由指定为 `+=` 或 `-=` 运算符的右操作数的表达式组成。
- 对于用户定义的运算符，实参列表由一元运算符的单个操作数或二元运算符的两个操作数组成。

对于属性（第 10.7 节）、事件（第 10.8 节）和用户定义运算符（第 10.10 节），其实参始终以值形参（第 10.6.1.1 节）的形式来传递。索引器（第 10.9 节）的实参始终以值形参（第 10.6.1.1 节）或形参数组（第 10.6.1.4 节）的形式来传递。这些函数成员类别不支持引用形参和输出形参。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

实例构造函数、方法、索引器或委托调用的实参指定为 *argument-list*：

argument-list:
    argument
    argument-list , *argument*

argument:
    argument-name$_{opt}$ *argument-value*

argument-name:
    identifier :

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

argument-list 由一个或多个 *argument* 组成，各实参之间用逗号分隔。每个实参由一个可选的 *argument-name* 及后跟的 *argument-value* 构成。带有 *argument-name* 的 *argument* 称为命名实参 (named argument)，而没有 *argument-name* 的 *argument* 称为位置实参 (positional argument)。在 *argument-list* 中，位置实参出现在命名实参后是错误的。

argument-value 可以采用下列形式之一：

- *expression*，指示将实参以值形参（第 10.6.1.1 节）的形式传递。
- 后跟 *variable-reference*（第 5.4 节）的关键字 `ref`，指示将实参以引用形参（第 10.6.1.2 节）的形式来传递。变量在可以作为引用形参传递之前，必须先明确赋值（第 5.3 节）。
- 后跟 *variable-reference*（第 5.4 节）的关键字 `out`，指示将实参以输出形参（第 10.6.1.3 节）的形式来传递。在将变量作为输出形参传递的函数成员调用之后，可认为该变量已明确赋值（第 5.3 节）。

形式分别确定实参 的***形参传递模式***：*值*、*引用*或*输出*。

7.5.1.1 对应形参

对于实参列表中的每个实参，在所调用的函数成员或委托中必须存在对应形参。

后面使用的形参列表按以下方式确定：

- 对于类中定义的虚方法和索引器，形参列表从函数成员的最具体声明或重写中选取，方法是从接收器的静态类型开始，在其基类中进行搜索。
- 对于接口方法和索引器，形参列表从成员的最具体定义选取，方法是从接口类型开始，在基接口中进行搜索。如果未找到唯一的形参列表，则构造一个具有不可访问名称且没有可选形参的形参列表，从而使调用不能使用命名形参或省略可选实参。
- 对于分部方法，将使用定义分部方法声明的形参列表。
- 对于所有其他函数成员和委托，只有一个形参列表（即使用的形参列表）。

某个实参或形参的位置定义为实参列表或形参列表中，位于该实参或形参之前的实参或形参的数量。

函数成员实参的对应形参按以下方式建立：

- 实例构造函数、方法、索引器和委托的 *argument-list* 中的实参：
    - 位置实参对应于出现在形参列表中相同位置上的固定形参。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

  - 如果函数成员的某个位置实参使用正常形式调用了某个形参数组，则该位置实参对应于该参数数组（必须出现在形参列表中的相同位置上）。
  - 如果函数成员的某个位置实参使用展开形式调用了某个形参数组，并且形参列表中的相同位置上没有出现任何固定形参，则该位置实参对应于该形参数组中的某个元素。
  - 命名实参对应于形参列表中具有相同名称的形参。
  - 对于索引器，当调用 `set` 访问器时，作为赋值运算符的右操作数指定的表达式对应于 `set` 访问器声明的隐式 `value` 形参。
- 对于属性，在调用 `get` 访问器时没有实参。当调用 `set` 访问器时，作为赋值运算符的右操作数指定的表达式对应于 `set` 访问器声明的隐式 `value` 形参。
- 对于用户定义的一元运算符（包括转换），单个操作数对应于运算符声明的单个形参。
- 对于用户定义的二元运算符，左操作数对应于运算符声明的第一个形参，右操作数对应于第二个形参。

7.5.1.2 实参列表的运行时计算

在函数成员调用（第 7.5.4 节）的运行时处理期间，将按顺序从左到右计算实参列表的表达式或变量引用，具体规则如下：

- 对于值形参，计算实参表达式并执行到相应的形参类型的隐式转换（第 0 节）。结果值在函数成员调用中成为该值形参的初始值。
- 对于引用形参或输出形参，计算对应的变量引用，所得的存储位置在函数成员调用中成为该形参表示的存储位置。如果作为引用形参或输出形参给定的变量引用是一个 *reference-type* 的数组元素，则执行运行时检查以确保该数组的元素类型与形参类型相同。如果此检查失败，将引发 `System.ArrayTypeMismatchException`。

方法、索引器和实例构造函数可以将其最右边的形参声明为形参数组（第 10.6.1.4 节）。调用此类函数成员可采取标准形式或展开形式两种形式中适用的形式（第 7.5.3.1 节）：

- 当具有形参数组的函数成员以其正常形式调用时，为该形参数组给定的实参必须是一个可隐式转换（第 0 节）为形参数组类型的表达式。在此情况下，形参数组的作用与值形参完全一样。
- 当具有形参数组的函数成员以其展开形式调用时，调用必须为形参数组指定零个或多个位置实参，其中每个实参都是一个可隐式转换（第 0 节）为该形参数组的元素类型的表达式。在此情况下，调用会创建一个该形参数组类型的实例，其所含的元素个数等于给定的实参个数，再用给定的实参值初始化此数组实例的每个元素，然后将新创建的数组实例用作实参。

实参列表的表达式始终按其书写的顺序进行计算。因此，示例

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

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

产生输出

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

如果存在从 B 到 A 的隐式引用转换，则数组协变规则（第 12.5 节）允许数组类型 A[] 的值是对数组类型 B[] 的实例的引用。根据这些规则，当 *reference-type* 的数组元素作为引用或输出形参传递时，需要执行运行时检查以确保数组的实际元素类型与形参类型相同。在下面的示例中

```
class Test
{
   static void F(ref object x) {...}

   static void Main() {
      object[] a = new object[10];
      object[] b = new string[10];
      F(ref a[0]);      // Ok
      F(ref b[1]);      // ArrayTypeMismatchException
   }
}
```

第二个 F 调用将导致引发 System.ArrayTypeMismatchException，原因是 b 的实际元素类型是 string 而不是 object。

当具有形参数组的函数成员以其展开形式调用时，对调用的处理方式完全类似于如下过程：在展开的形参周围插入具有数组初始值设定项（第 7.6.10.4 节）的数组创建表达式。例如，给定下面的声明

```
void F(int x, int y, params object[] args);
```

以下方法的展开形式的调用

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

完全对应于

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

请特别注意，当为形参数组指定的实参的个数为零时，将创建一个空数组。

从具有对应可选形参的函数成员省略实参时，将隐式传递函数成员声明的默认实参。因为这些实参始终为常量，所以其计算将不会影响剩余实参的计算顺序。

7.5.2 类型推断

不指定类型实参而调用泛型方法时，类型推断 (type inference) 过程将尝试为调用推断类型实参。类型推断的存在允许使用更方便的语法调用泛型方法，并使得程序员不必指定多余的类型信息。例如，给定下面的方法声明：

```
class Chooser
{
   static Random rand = new Random();

   public static T Choose<T>(T first, T second) {
      return (rand.Next(2) == 0)? first: second;
   }
}
```

可以在不显式指定类型实参的情况下调用 Choose 方法：

```
int i = Chooser.Choose(5, 213);              // Calls Choose<int>
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
string s = Chooser.Choose("foo", "bar");     // Calls Choose<string>
```

借助于类型实参推断，可通过传递给方法的实参来确定类型实参 int 和 string。

类型推断在方法调用（第 7.6.5.1 节）的绑定时处理过程中进行，发生在调用的重载决策步骤之前。当在方法调用中指定了特定的方法组，并且没有在方法调用中指定类型实参时，将会对该方法组中的每个泛型方法应用类型推断。如果类型推断成功，则使用推断出的类型实参确定用于后续重载解析的实参的类型。如果重载决择选择一个泛型方法作为要调用的方法，则使用推断出的类型实参作为用于调用的实际类型实参。如果特定方法的类型推断失败，则该方法不参与重载决策。类型推断失败本身不会导致绑定时错误。但是，当重载决策未能找到任何适用的方法时，它通常会导致绑定时错误。

如果所提供的实参的数目与方法中的形参的数目不同，则推断立即失败。否则，假定泛型方法具有以下签名：

T_r M<X_1…X_n>(T_1 x_1 … T_m x_m)

对于 M(E_1 …E_m) 形式的方法调用，类型推断的任务是为每个类型形参 X_1…X_n 找到唯一的类型实参 S_1…S_n，以使 M<S_1…S_n>(E_1…E_m) 调用有效。

在推断过程中，每个类型形参 X_i 或者固定到一个特定类型 S_i 或者未固定，而具有一组关联*的界限*。每个界限都属于某个类型 T。最初，每个类型变量 X_i 均未固定，具有一组空的界限。

类型推断分阶段进行。每个阶段都将尝试基于上一阶段的发现为更多类型变量推断类型实参。第一阶段进行一些初始的界限推断，而第二阶段将类型变量固定到特定类型并推断其他界限。第二阶段可能需要重复多次。

注意：类型推断不仅仅在调用泛型方法时发生。方法组转换的类型推断详见第 7.5.2.13 节中的说明，查找一组表达式的最佳通用类型详见第 7.5.2.14 节中的说明。

7.5.2.1 第一阶段

对于每个方法实参 E_i：

- 如果 E_i 为匿名函数，则从 E_i 到 T_i 进行显式参数类型推断（第 7.5.2.7 节）
- 否则，如果 E_i 具有类型 U 且 x_i 为值形参，则从 U 到 T_i 进行下限推断。
- 否则，如果 E_i 具有类型 U 且 x_i 为 ref 或 out 形参，则将*从* U *到* T_i 进行*精确推断*。
- 否则，将不对此实参进行推断。

7.5.2.2 第二阶段

第二阶段如下进行：

- 所有不依赖（第 7.5.2.5 节）任何 X_j 的未固定类型变量 X_i 都将被固定（第 7.5.2.10 节）。
- 如果不存在这样的类型变量，则固定所有未固定的类型变量 X_i，为此应符合下述所有规则：
  - 至少有一个依赖 X_i 的类型变量 X_j
  - X_i 具有非空界限集
- 如果不存在此类类型变量，但仍有未固定的类型变量，则类型推断会失败。
- 否则，如果不存在其他任何未固定的类型变量，则类型推断将成功。
- 否则，对于具有相应形参类型 T_i (其中输出类型（第 7.5.2.4 节）包含非固定类型变量 X_j，但输入类型（第 7.5.2.3 节）不包含此变量)的所有实参 E_i，从 E_i 到 T_i 进行输出类型推断（第 7.5.2.6 节）。然后重复第二阶段。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.5.2.3 输入类型

如果 E 为方法组或隐式类型化的匿名函数，并且 T 为委托类型或表达式目录树类型，则 T 的所有形参类型都是具有类型 T 的 E 的输入 类型。

7.5.2.4 输出类型

如果 E 是一个方法组或匿名函数并且 T 是委托类型或表达式目录树类型，则 T 的返回类型是类型为 T 的 E 的输出类型。

7.5.2.5 依赖

*未固定的*类型变量 X_i 在下述情形中*直接依赖未固定的*类型变量 X_j：对于类型为 T_k 的某个实参 E_k，X_j 出现在类型为 T_k 的 E_k 的*输入类型*中，而 X_i 则出现在类型为 T_k 的 E_k 的*输出类型*中。

X_j 在下述情形中*依赖* X_i：X_j *直接依赖* X_i，或者 X_i *直接依赖* X_k，而 X_k *又依赖于* X_j。因而，“依赖”具有传递性，但不形成“直接依赖”的自反闭包。

7.5.2.6 输出类型推断

按照以下方法从表达式 E 到 类型 T 进行输出类型推断：

- 如果 E 为具有推断返回类型 U（第 7.5.2.12 节）的匿名函数，且 T 为具有返回类型 T_b 的委托类型或表达式目录树类型，则从 U 到 T_b 进行下限推断（第 7.5.2.9 节）。
- 否则，如果 E 为方法组，T 为具有参数类型 $T_1 \cdots T_k$ 和返回类型 T_b 的委托类型或表达式目录树类型，且具有类型 $T_1 \cdots T_k$ 的 E 的重载决策产生了具有返回类型 U 的单个方法，则从 U 到 T_b 进行下限推断。
- 否则，如果 E 为具有类型 U 的表达式，则从 U 到 T 进行下限推断。
- 否则，不进行任何推断。

7.5.2.7 参数类型显式推断

按照以下方法从表达式 E 到类型 T 进行显式参数类型推断：

- 如果 E 为具有参数类型 $U_1 \cdots U_k$ 的显式类型匿名函数，T 为具有参数类型 $V_1 \cdots V_k$ 的委托类型或表达式目录树类型，则对于每个 U_i，从 U_i 到对应的 V_i 进行准确推断（第 7.5.2.8 节）。

7.5.2.8 精确推断

按如下所述从类型 U 到类型 V 进行精确推断：

- 如果 V 是未固定的 X_i 之一，则将 U 添加到 X_i 的精确界限集中。
- 否则，通过检查是否存在以下任何一种情况来确定集合 $V_1 \cdots V_k$ 和 $U_1 \cdots U_k$ ：
  - V 是数组类型 $V_1[\cdots]$ ， U 是具有相同秩的数组类型 $U_1[\cdots]$
  - V 是类型 $V_1?$，U 是类型 $U_1?$
  - V 是构造类型 $C<V_1 \cdots V_k>$ and U 是构造类型 $CC<U_1 \cdots U_k>$

  如果存在以上任意情况，则从每个 U_i 到对应的 V_i 进行精确推断。
- 否则，不进行任何推断。

7.5.2.9 下限推断

按如下所述从类型 U 到类型 V 进行下限推断：

- 如果 V 是未固定的 X_i 之一，则将 U 添加到 X_i 的精确下限界限集中。
- 否则，如果 V 为 V_1? 类型，而 U 为 U_1? 类型，则从 U_1 到 V_1 进行下限推断。
- 否则，通过检查是否存在以下任意一种情况，确定 $U_1 \cdots U_k$ 和 $V_1 \cdots V_k$ 这两个集：
  - V 是数组类型 $V_1[\cdots]$，U 是具有相同秩的数组类型 $U_1[\cdots]$
  - V 是 IEnumerable<V_1>、ICollection<V_1> 或 IList<V_1> 之一，U 是一维数组类型 $U_1[]$
  - V 是构造类、结构、接口或委托类型 C<$V_1 \cdots V_k$>，并且存在唯一类型 C<$U_1 \cdots U_k$>，使 U（或者，如果 U 是类型形参，则为其有效基类或其有效接口集的任意成员）等于、（直接或间接）继承自或者（直接或间接）实现 C<$U_1 \cdots U_k$>。

    （“唯一性”限制表示对于 interface C<T>{} class U: C<X>, C<Y>{}，不进行从 U 到 C<T> 的推断，因为 U_1 可以是 X 或 Y。）

  如果存在以上任意情况，则从每个 U_i 到对应的 V_i 进行推断，如下所示：
  - 如果不知道 U_i 为引用类型，则进行精确推断
  - 否则，如果 U 为数组类型，则进行下限推断
  - 否则，如果 V 为 C<$V_1 \cdots V_k$>，则推断依赖于 C 的第 i 个类型参数：
    - 如果该参数是协变的，则进行下限推断。
    - 如果该参数是逆变的，则进行上限推断。
    - 如果该参数是固定的，则进行精确推断。
- 否则，不进行任何推断。

7.5.2.10 上限推断

按如下所述从类型 U 到类型 V 进行上限推断：

- 如果 V 是未固定的 X_i 之一，则将 U 添加到 X_i 的精确上限界限集中。
- 否则，通过检查是否存在以下任何一种情况来确定集合 $V_1 \cdots V_k$ 和 $U_1 \cdots U_k$：
  - U 是数组类型 $U_1[\cdots]$，V 是具有相同秩的数组类型 $V_1[\cdots]$
  - U 是 IEnumerable<U_e>、ICollection<U_e> 或 IList<U_e> 之一，V 是一维数组类型 $V_e[]$
  - U 是类型 U_1?，V 是类型 V_1?
  - U 是构造类、结构、接口或委托类型 C<$U_1 \cdots U_k$>，V 是等于、（直接或间接）继承自或者（直接或间接）实现唯一类型 C<$V_1 \cdots V_k$> 的类、结构、接口或委托类型

    （“唯一性”限制表示如果我们有 interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}，则不进行从 C<U_1> 到 V<Q> 的推断。也不进行从 U_1 到 X<Q> 或 Y<Q> 的推断。）

  如果存在以上任意情况，则从每个 U_i 到对应的 V_i 进行推断，如下所示：
  - 如果不知道 U_i 为引用类型，则进行精确推断
  - 否则，如果 V 为数组类型，则进行上限推断
  - 否则，如果 U 为 C<$U_1 \cdots U_k$>，则推断依赖于 C 的第 i 个类型参数：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果该参数是协变的，则进行上限推断。
- 如果该参数是逆变的，则进行下限推断。
- 如果该参数是固定的，则进行精确推断。

- 否则，不进行任何推断。

7.5.2.11 固定

具有界限集的未固定类型变量 X_i 按如下方式固定：

- 候选类型 U_j 的集以所有类型的集形式在 X_i 的界限集中开始。
- 然后我们依次检查 X_i 的每个界限：对于 X_i 的每个精确界限 U，将与 U 不同的所有类型 U_j 都从候选集中移除。对于 X_i 的每个下限 U，将不存在从 U 进行的隐式转换的所有类型 U_j 都从候选集中移除。对于 X_i 的每个上限 U，将不存在从其到 U 进行的隐式转换的所有类型 U_j 都从候选集中移除。
- 如果在其余的候选类型 U_j 中，存在唯一类型 V（该类型可从所有其他候选类型隐式转换而来），则将 X_i 固定到 V。
- 否则，类型推断将失败。

7.5.2.12 推断返回类型

匿名函数 F 的推断返回类型 (Inferred return type) 在类型推断和重载决策期间使用。匿名函数的推断返回类型仅能在所有参数类型均已知的情况下确定，因为参数类型是隐式给出的；是通过匿名函数转换提供的；或者是在封闭泛型方法调用上进行类型推断期间推断出的。

推断结果类型按如下方式确定：

- 如果 F 的主体是具有某个类型的 *expression*，则 F 的推断结果类型为该表达式的类型。
- 如果 F 的函数体是一个 *block* 并且该块的 `return` 语句中的表达式集具有最通用类型 T（第 7.5.2.14 节），则 F 的推断返回类型为 T。
- 否则，无法为 F 推断返回类型。

推断返回类型按如下方式确定：

- 如果 F 为异步且 F 的主体是归类为 Nothing 的表达式（第 7.1 节）或一个语句块（其中的 return 语句没有表达式），则推断返回类型为 `System.Threading.Tasks.Task`
- 如果 F 为异步，且具有推断结果类型 T，则推动返回类型为 `System.Threading.Tasks.Task<T>`。
- 如果 F 为非异步，且具有推断结果类型 T，则推断返回类型为 T。
- 否则，无法为 F 推断返回类型。

请考虑使用在 `System.Linq.Enumerable` 类中声明的 `Select` 扩展方法，作为涉及匿名函数的类型推断示例：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
namespace System.Linq
{
   public static class Enumerable
   {
      public static IEnumerable<TResult> Select<TSource,TResult>(
         this IEnumerable<TSource> source,
         Func<TSource,TResult> selector)
      {
         foreach (TSource element in source) yield return selector(element);
      }
   }
}
```

假定 System.Linq 命名空间是使用 using 子句导入的，并假定类 Customer 具有类型为 string 的 Name 属性，则 Select 方法可用于选择客户列表中的名称：

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

Select 的扩展方法调用（第 7.6.5.2 节）是通过重写对静态方法调用的调用而进行处理的：

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

因为类型实参不是显式指定的，所以类型推断用于推断类型实参。首先，将 customers 实参关联到 source 形参，推断出 T 为 Customer。然后，使用上述匿名函数类型推断过程，为 c 指定类型 Customer，将表达式 c.Name 与 selector 形参的返回类型相关联，推断出 S 为 string。因而，此调用等效于

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

并且结果的类型为 IEnumerable<string>。

下面的示例演示匿名函数类型推断如何允许类型信息在泛型方法调用的实参之间“流动”。给定方法：

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
   return f2(f1(value));
}
```

针对此调用的类型推断：

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

过程如下：首先将实参"1:15:30"关联到 value 形参，推断出 X 为 string。然后，为第一个匿名函数的形参 s 指定推断类型 string，并将表达式 TimeSpan.Parse(s) 与 f1 的返回类型相关联，推断出 Y 为 System.TimeSpan。最后，为第二个匿名函数的形参 t 指定推断类型 System.TimeSpan，并将表达式 t.TotalSeconds 与 f2 的返回类型相关联，推断出 Z 为 double。因而，调用结果为 double 类型。

7.5.2.13 方法组转换的类型推断

与泛型方法的调用类似，当将包含泛型方法的方法组 M 转换为给定的委托类型 D（§（第 6.6)节）时也必须应用类型推断。给定一个方法

$T_r\ M<X_1 \cdots X_n>(T_1\ x_1\ \cdots\ T_m\ x_m)$

和分配给委托类型 D 的方法组 M，则类型推断的任务是查找类型实参 $S_1 \cdots S_n$，以使表达式：

$M<S_1 \cdots S_n>$

与 D 兼容（第 15.1 节）。

与泛型方法调用的类型推断算法不同的是，在这种情形下，只有实参类型，而没有实参表达式。特别是没有匿名函数，因此不需要进行多阶段推断。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

而认为所有 X_i 均未固定，并从 D 的每个实参类型 U_j 到 M 的对应形参类型 T_j 进行下限推断。如果没有为任何 X_i 找到界限，则类型推断将失败。否则，所有将 X_i 均固定到对应的 S_i，它们是类型推断的结果。

7.5.2.14 查找一组表达式的最通用类型

在某些情形下，需要为一组表达式推断出通用类型。特别是，用这种方式找到隐式类型化数组的元素类型和具有 *block* 体的匿名函数的返回类型的情形。

从直观上看，给定一组表达式 $E_1 \cdots E_m$，此推断应等效于调用某个方法

```
Tr M<X>(X x1 ··· X xm)
```

其中 E_i 为实参。

更确切地说，推断从未固定的类型变量 X 开始。然后从每个 E_i 到 X 进行输出类型推断。最后固定 X，如果成功，结果类型 S 就是表达式的最佳通用结果类型。如果不存在此类 S，则表达式没有最佳通用类型。

7.5.3 重载决策

重载决策是一种绑定时机制，用于在给定了参数列表和一组候选函数成员的情况下，选择一个最佳函数成员来实施调用。在 C# 内，重载决策在下列不同的上下文中选择一个应调用的函数成员：

- 调用在 *invocation-expression*（第 7.6.5.1 节）中命名的方法。
- 调用在 *object-creation-expression*（第 7.6.10.1 节）中命名的实例构造函数。
- 通过 *element-access*（第 7.6.6 节）调用索引器访问器。
- 调用表达式（第 7.3.3 节和第 7.3.4 节）中引用的预定义运算符或用户定义运算符。

这些上下文中的每一个都以自己的唯一方式定义候选函数成员集和实参列表，上面列出的章节对此进行了详细说明。例如，方法调用的候选集不包括标记为 `override`（第 7.4 节）的方法，而且如果派生类中的任何方法适用（第 7.6.5.1 节），则基类中的方法不是候选方法。

一旦确定了候选函数成员和实参列表，对最佳函数成员的选择在所有情况下都相同，都遵循下列规则：

- 如果给定了适用的候选函数成员集，则在其中选出最佳函数成员。如果该集只包含一个函数成员，则该函数成员为最佳函数成员。否则，最佳函数成员的选择依据是：各成员对给定的实参列表的匹配程度。比其他所有函数成员匹配程度都高的那个函数成员就是最佳函数成员，但有一个前提：必须使用第 7.5.3.2 节中的规则将每个函数成员与其他所有函数成员进行比较。如果不是正好有一个函数成员比所有其他函数成员都好，则函数成员调用不明确并发生绑定时错误。

下面几节定义有关术语适用的函数成员 (applicable function member) 和更好的函数成员 (better function member) 的准确含义。

7.5.3.1 适用函数成员

当满足以下所有条件时，就称函数成员对于实参列表 A 是一个适用的函数成员：

- A 中的每个实参对应于函数成员声明中的一个形参（如第 7.5.1.1 节中所述），最多一个实参对应于一个形参，并且没有对应实参的所有形参都是可选形参。
- 对于 A 中的每个实参，实参的形参传递模式（第 7.5.1 节）与对应形参（第 1.6.6.1 节）的形参传递模式相同，而且
  - 对于值形参或形参数组，存在从实参表达式到对应形参的类型的隐式转换（第 0 节），或者

- 对于 ref 或 out 形参，实参表达式的类型与对应形参的类型相同。ref 或 out 形参毕竟只是传递的实参的别名。

对于包含参数数组的函数成员，如果按上述规则判定该函数成员是适用的，则称它以正常形式 (normal form) 适用。如果包含参数数组的函数成员以正常形式不适用，则该函数成员可能以展开形式 (expanded form) 适用：

- 构造展开形式的方法是：用形参数组的元素类型的零个或更多值参数替换函数成员声明中的形参数组，使实参列表 A 中的实参数目匹配总的形参数目。如果 A 中的实参比函数成员声明中的固定形参的数目少，则该函数成员的展开形式无法构造，因而可判定该函数成员不适用。
- 否则，如果对于 A 中的每个实参，它的实参传递模式与相应形参的形参传递模式相同，并且下列条件成立，则称该成员函数以展开形式适用：
  - 对于固定值形参或展开操作所创建的值形参，存在从实参类型到对应的形参类型的隐式转换（第 0 节），或者
  - 对于 ref 或 out 参数，实参的类型与对应形参的类型相同。

7.5.3.2 更好的函数成员

为确定更好的函数成员，构造一个精炼的实参列表 A，其中只包含实参表达式本身，采用其出现在原始实参列表中的顺序。

每个候选函数成员的形参列表按以下方式构造：

- 如果函数成员只以展开形式适用，则使用展开形式。
- 从形参列表中移除没有对应实参的可选形参。
- 对形参重新排序，从而使其出现在与实参列表中的对应实参相同的位置上。

给定一个带有实参表达式集 { $E_1, E_2, \ldots, E_N$ } 的实参列表 A 和带有形参类型 { $P_1, P_2, \ldots, P_N$ } 和 { $Q_1, Q_2, \ldots, Q_N$ } 的两个适用的函数成员 M_P 和 M_Q，则在以下情形中，M_P 将定义为比 M_Q ***更好的函数成员***：

- 对于每个实参，从 E_X 到 Q_X 的隐式转换不如从 E_X 到 P_X 的隐式转换好，并且
- 对于至少一个参数而言，从 E_X 到 P_X 的转换比从 E_X 到 Q_X 的转换更好。

当执行此计算时，如果 M_P 或 M_Q 以展开形式适用，则 P_X 或 Q_X 所代表的是展开形式的参数列表中的参数。

在形参类型序列 {$P_1, P_2, \cdots, P_N$} 和 {$Q_1, Q_2, \cdots, Q_N$} 等效（即每个 P_i 都有到对应 Q_i 的标识转换）的情况下，将应用以下附加规则来确定更好的函数成员。

- 如果 M_P 是非泛型方法而 M_Q 是泛型方法，则 M_P 比 M_Q 好。
- 否则，如果 M_P 在正常形式下适用，M_Q 有一个 params 数组并且仅在其展开形式下适用，则 M_P 比 M_Q 好。
- 否则，如果 M_P 具有比 M_Q 更多的已声明形参，则 M_P 比 M_Q 好。如果两个方法都有 params 数组，并且都仅在其展开形式下适用，就可能出现这种情况。
- 否则，如果 M_P 的所有形参都有对应实参，而需要使用默认实参替换 M_Q 中的至少一个可选形参，则 M_P 比 M_Q 好。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，如果 M_P 具有比 M_Q 更明确的形参类型，则 M_P 比 M_Q 好。假设 $\{R_1, R_2, \cdots, R_N\}$ 和 $\{S_1, S_2, \cdots, S_N\}$ 表示 M_P 和 M_Q 的未实例化和未展开的形参类型。如果对于每个形参，R_X 都不比 S_X 更不明确，并且至少对于一个形参，R_X 比 S_X 更明确，则 M_P 的形参类型比 M_Q 的形参类型更明确：
  - 类型形参不如非类型形参明确。
  - 递归地，如果某个构造类型至少有一个类型实参更明确，并且没有类型实参比另一个构造类型（两者具有相同数目的类型实参）中的对应类型实参更不明确，则某个构造类型比另一个构造类型更明确。
  - 如果一个数组类型的元素类型比另一个数组类型的元素类型更明确，则第一个数组类型比第二个数组类型（具有相同的维数）更明确。
- 否则，如果一个成员是非提升运算符而另一个是提升运算符，则非提升运算符更佳。
- 否则，两个函数成员都不是更好的。

7.5.3.3 表达式的更佳转换

给定一个从表达式 E 转换到类型 T_1 的隐式转换 C_1，以及一个从表达式 E 转换到类型 T_2 的隐式转换 C_2，如果至少符合以下条件之一，则 C_1 与 C_2 相比是更佳转换 (better conversion)：

- E 具有类型 S，并存在从 S 到 T_1 的标识转换，但不存在从 S 到 T_2 的标识转换
- E 不是匿名函数，并且与 T_2 相比，T_1 是更佳转换目标（第 7.5.3.5 节）
- E 是匿名函数，T_1 是委托类型 D_1 或表达式树类型 Expression<D_1>，T_2 是委托类型 D_2 或表达式树类型 Expression<D_2> 并符合以下条件之一：
  - D_1 与 D_2 相比是更佳转换目标
  - D_1 和 D_2 有相同的形参列表，并且符合以下条件之一：
    - D_1 具有返回类型 Y_1，D_2 具有返回类型 Y_2，对于 E，在形参列表上下文中存在推断返回类型 X（第 7.5.2.12 节），并且从 X 到 Y_1 的转换比从 X 到 Y_2 的转换好
    - E 为异步，D_1 具有返回类型 Task<Y_1>，D_2 具有返回类型 Task<Y_2>，对于 E，在形参列表上下文中存在推断返回类型 Task<X>（第 7.5.2.12 节），并且从 X 到 Y_1 的转换比从 X 到 Y_2 的转换好
    - D_1 具有返回类型 Y 且 D_2 返回 void

7.5.3.4 类型的更佳转换

给定一个从类型 S 转换到类型 T_1 的转换 C_1 以及一个从类型 S 转换到类型 T_2 的转换 C_2，如果至少符合以下条件之一，则 C_1 与 C_2 相比是更佳转换 (better conversion)：

- 存在从 S 到 T_1 的标识转换，但不存在从 S 到 T_2 的标识转换
- T_1 与 T_2 相比是更佳转换目标（第 §7.5.3.5 节）

7.5.3.5 更佳转换目标

给定两个不同类型 T_1 和 T_2，如果至少符合以下条件之一，则 T_1 与 T_2 相比是更佳转换目标：

- 存在从 T_1 到 T_2 的隐式转换，不存在从 T_2 到 T_1 的隐式转换
- T_1 为有符号的整型，T_2 为无符号的整型。具体包括：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- T_1 为 sbyte 且 T_2 为 byte、ushort、uint 或 ulong
- T_1 为 short 且 T_2 为 ushort、uint 或 ulong
- T_1 为 int 且 T_2 为 uint 或 ulong
- T_1 为 long，T_2 为 ulong。

7.5.3.6 泛型类中的重载

虽然声明的签名必须唯一，但是在替换类型实参时可能会导致出现完全相同的签名。在此类情形中，上述重载决策的附加规则将挑选最明确的成员。

下面的示例根据此规则演示有效和无效的重载：

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
   int F1(U u);                // Overload resulotion for G<int>.F1
   int F1(int i);              // will pick non-generic

   void F2(I1<U> a);           // Valid overload
   void F2(I2<U> a);
}

class G2<U,V>
{
   void F3(U u, V v);          // Valid, but overload resolution for
   void F3(V v, U u);          // G2<int,int>.F3 will fail

   void F4(U u, I1<V> v);      // Valid, but overload resolution for
   void F4(I1<V> v, U u);      // G2<I1<int>,int>.F4 will fail

   void F5(U u1, I1<V> v2);    // Valid overload
   void F5(V v1, U u2);

   void F6(ref U u);           // valid overload
   void F6(out V v);
}
```

7.5.4 动态重载决策的编译时检查

即使动态绑定操作的重载决策在运行时发生，有时在编译时便能够知道将从中选择重载的函数成员的列表：

- 对于针对类型或其静态类型不是 dynamic 的值的方法调用（第 7.6.5.1 节），编译时将知道方法组中的可访问方法集。
- 对于对象创建表达式（第 7.6.10.1 节），编译时将知道类型中的可访问构造函数集。
- 对于索引器访问（第 7.6.6.2 节），编译时将知道接收器中的可访问索引器集。

在以上情况中，将对已知函数成员集中的每个成员执行有限的编译时检查，以了解成员是否已知在运行时绝不会调用。对于每个函数成员 F，将构成已修改形参和实参的列表：

- 首先，如果 F 是泛型方法并且提供了类型实参，则将用形参列表中的类型形参替换这些实参。但是，如果未提供类型实参，则不会发生此类替换。
- 然后，将省略其类型是开放式（即，包含类型形参，请参见第 4.4.2 节）的任何形参及其对应的形参。

若要 F 通过检查，必须满足下列所有条件：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 依照第 7.5.3.1 节，F 的已修改形参的列表可应用于已修改实参的列表。
- 已修改形参的列表中的所有构造类型满足其约束（第 4.4.4 节）。
- 如果在上一步中替换了 F 的类型形参，则满足其约束。
- 如果 F 是静态方法，则方法组不得从其接收器在编译时已知将为变量还是值的 *member-access* 生成。
- 如果 F 是实例方法，则方法组不得从其接收器在编译时已知将为类型的 *member-access* 生成。

如果没有候选通过此测试，则产生编译时错误。

7.5.5 函数成员调用

本节描述在运行时发生的调用一个特定的函数成员的进程。这里假定绑定时进程已确定了要调用的特定成员（可能采用重载决策从一组候选函数成员中选出）。

为了描述调用进程，将函数成员分成两类：

- 静态函数成员。包括实例构造函数、静态方法、静态属性访问器和用户定义的运算符。静态函数成员总是非虚的。
- 实例函数成员。包括实例方法、实例属性访问器和索引器访问器。实例函数成员不是非虚的就是虚的，并且总是在特定的实例上调用。该实例由实例表达式计算，并可在函数成员内以 this（第 7.6.7 节）的形式对其进行访问。

函数成员调用的运行时处理包括以下步骤（其中 M 是函数成员，如果 M 是实例成员，则 E 是实例表达式）：

- 如果 M 是静态函数成员，则：
  - 实参列表按照第 7.5.1 节中的说明进行计算。
  - 调用 M。
- 如果 M 是在 *value-type* 中声明的实例函数成员，则：
  - 计算 E。如果该计算导致异常，则不执行进一步的操作。
  - 如果 E 没有被归类为一个变量，则创建一个与 E 同类型的临时局部变量，并将 E 的值赋给该变量。这样，E 就被重新归类为对该临时局部变量的一个引用。该临时变量在 M 中可以以 this 的形式被访问，但不能以任何其他形式访问。因此，仅当 E 是真正的变量时，调用方才可能观察到 M 对 this 所做的更改。
  - 实参列表按照第 7.5.1 节中的说明进行计算。
  - 调用 M。E 引用的变量成为 this 引用的变量。
- 如果 M 是在 *reference-type* 中声明的实例函数成员，则：
  - 计算 E。如果该计算导致异常，则不执行进一步的操作。
  - 实参列表按照第 7.5.1 节中的说明进行计算。
  - 如果 E 的类型为 *value-type*，则执行装箱转换（第 4.3.1 节）以将 E 转换为 object 类型，并在下列步骤中，将 E 视为 object 类型。在这种情况下，M 只能是 System.Object 的成员。
  - 检查 E 的值是否有效。如果 E 的值为 null，则将引发 System.NullReferenceException，并且不执行进一步的步骤。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 要调用的函数成员实现按以下规则确定：
  - 如果 E 的绑定时类型是接口，则调用的函数成员是 M 的实现，此实现由 E 引用的实例的运行时类型提供。确定此函数成员时，应用接口映射规则（第 13.4.4 节）确定由 M 引用的实例运行时类型提供的 E 实现。
  - 否则，如果 M 是虚函数成员，则调用的函数成员是由 E 引用的实例运行时类型提供的 M 实现。确定此函数成员时，对于 E 引用的实例的运行时类型，应用“确定 M 的派生程度最大的实现”的规则（第 10.6.3 节）。
  - 否则，M 是非虚函数成员，调用的函数成员是 M 本身。
- 调用在上一步中确定的函数成员实现。E 引用的对象成为 this 引用的对象。

7.5.5.1 已装箱实例上的调用

在下列情形中，可以通过 *value-type* 的已装箱实例来调用以该 *value-type* 实现的函数成员：

- 当该函数成员是从 object 类型继承的，且具有 override 修饰符，并通过 object 类型的实例表达式被调用时。
- 当函数成员是接口函数成员的实现并且通过 *interface-type* 的实例表达式被调用时。
- 当函数成员通过委托被调用时。

在这些情形中，将已装箱实例视为包含 *value-type* 的变量，并且此变量将在函数成员调用中成为 this 引用的变量。具体而言，这表示当调用已装箱实例的函数成员时，该函数成员可以修改已装箱实例中包含的值。

7.6 基本表达式

基本表达式包括最简单的表达式形式。

primary-expression:
primary-no-array-creation-expression
array-creation-expression

primary-no-array-creation-expression:
literal
simple-name
parenthesized-expression
member-access
invocation-expression
element-access
this-access
base-access
post-increment-expression
post-decrement-expression
object-creation-expression
delegate-creation-expression
anonymous-object-creation-expression
typeof-expression
checked-expression
unchecked-expression
default-value-expression
anonymous-method-expression

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

基本表达式分为 *array-creation-expression* 和 *primary-no-array-creation-expression*。采用这种方式处理数组创建表达式（而不是将它与其他简单的表达式形式一起列出），使语法能够禁止可能的代码混乱，如

```
object o = new int[3][1];
```

被另外解释为

```
object o = (new int[3])[1];
```

7.6.1 文本

由 *literal*（第 2.4.4 节）组成的 *primary-expression* 属于值类别。

7.6.2 简单名称

simple-name 由一个标识符以及后跟的可选类型实参列表构成：

simple-name:
identifier type-argument-list$_{opt}$

simple-name 的形式为 I 或 I<A_1, ..., A_K>，其中 I 是单个标识符，<A_1, ..., A_K> 是可选的 *type-argument-list*。如果未指定 *type-argument-list* 时，则可将 K 视为零。*simple-name* 的计算和分类方式如下：

- 如果 K 为零，*simple-name* 在某个 *block* 内出现，并且该 *block*（或包容 *block*）的局部变量声明空间（第 3.3 节）包含一个名为 I 的局部变量、形参或常量，则 *simple-name* 将引用该局部变量、形参或常量，并将归为变量或值类别。
- 如果 K 为零，并且 *simple-name* 出现在泛型方法声明体中，并且该声明包含名为 I 的类型形参，则 *simple-name* 将引用该类型形参。
- 否则，对于每个实例类型 T（第 10.3.1 节），从直接的包容类型声明的实例类型开始，对每个包容类或结构声明（如果有）的实例类型继续进行如下过程：
  - 如果 K 为零，并且 T 的声明包含名为 I 的类型形参，则 *simple-name* 将引用该类型形参。
  - 否则，如果在 T 中对具有 K 个类型实参的 I 进行成员查找（第 7.4 节）得到匹配项：
    - 如果 T 为直接包容类或结构类型的实例类型，并且该查找标识了一个或多个方法，则结果是一个具有 this 的关联实例表达式的方法组。如果指定了类型实参列表，则将在调用泛型方法时使用它（第 7.6.5.1 节）。
    - 否则，如果 T 为直接包容类或结构类型的实例类型，如果查找标识出一个实例成员，并且引用发生在实例构造函数、实例方法或实例访问器的 *block* 内，则结果与 this.I 形式的成员访问（第 7.6.4 节）相同。仅当 K 为零时才会发生这种情况。
    - 否则，结果与 T.I 或 T.I<A_1, ..., 形式的成员访问（第 7.6.4 节）相同。A_K>.在此情况下，*simple-name* 引用实例成员将发生绑定时错误。
- 否则，对于每个命名空间 N，从出现 *simple-name* 的命名空间开始，依次继续每个包容命名空间（如果有），到全局命名空间为止，对下列步骤进行计算，直到找到实体：
  - 如果 K 为零，并且 I 为 N 中的命名空间的名称，则：
    - 如果出现 *simple-name* 的位置包含在 N 的命名空间声明中，并且该命名空间声明中包含将名称 I 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *simple-name* 是不明确的，并将发生编译时错误。
    - 否则，*simple-name* 引用 N 中名为 I 的命名空间。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，如果 N 包含一个具有名称 I 且有 K 个类型形参的可访问类型，则：
  - 如果 K 为零，并且出现 *simple-name* 的位置包含在 N 的命名空间声明中，并且该命名空间声明中包含将名称 I 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *simple-name* 是不明确的，并将发生编译时错误。
  - 否则，*namespace-or-type-name* 引用利用给定类型实参构造的该类型。
- 否则，如果出现 *simple-name* 的位置包含在 N 的命名空间声明中：
  - 如果 K 为零，并且该命名空间声明中包含一个将名称 I 与一个导入的命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *simple-name* 将引用该命名空间或类型。
  - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间中只包含一个名为 I 且有 K 个类型形参的类型，则 *simple-name* 将引用通过给定的类型实参构造的该类型。
  - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间中包含多个名为 I 且有 K 个类型形参的类型，则 *simple-name* 是不明确的，并将导致发生错误。

  注意这整个步骤与 *namespace-or-type-name*（第 3.8 节）的处理中对应的步骤完全相同。

- 否则，*simple-name* 是未定义的，并将出现编译时错误。

7.6.2.1 块中的固定含义

对于表达式或声明符中以完整 *simple-name* 形式出现的每个给定标识符，在出现处最近的外层局部变量声明空间（第 3.3 节）内，表达式或声明符中以完整 *simple-name* 形式出现的每个其他同一标识符都必须引用相同的实体。该规则确保在给定的块、switch 块、for、foreach 或 using 语句或匿名函数中，名称的含义总是相同。

下面的示例

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

将产生编译时错误，这是因为 x 引用外部块（其范围包括 if 语句中的嵌套块）中的不同实体。相反，示例

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

是允许的，这是因为在外部块中从未使用过名称 x。

注意固定含义的规则仅适用于简单名称。同一标识符在作为简单名称时有一种意义，而在作为一个成员访问（第 7.6.4 节）的右操作数时具有另一种意义，这是完全合法的。例如：

```
struct Point
{
   int x, y;

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

上面的示例阐释了一个将字段名用作实例构造函数中的参数名的通用模式。在该示例中，简单名称 x 和 y 引用参数，但这并不妨碍成员访问表达式 this.x 和 this.y 访问字段。

7.6.3 带括号的表达式

parenthesized-expression 由一个用括号括起来的 *expression* 组成。

parenthesized-expression:
(*expression*)

通过计算括号内的 *expression* 来计算 *parenthesized-expression*。如果括号内的 *expression* 表示命名空间或类型，将发生编译时错误。否则，*parenthesized-expression* 的结果为所含 *expression* 的计算结果。

7.6.4 成员访问

member-access 的组成部分包括：一个 *primary-expression*、一个 *predefined-type* 或 *qualified-alias-member*，后面依次是一个“.”标记、一个 *identifier* 和一个 *type-argument-list*（可选）。

member-access:
primary-expression . *identifier* *type-argument-list*$_{opt}$
predefined-type . *identifier* *type-argument-list*$_{opt}$
qualified-alias-member . *identifier* *type-argument-list*$_{opt}$

predefined-type: one of

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

qualified-alias-member 产生式在第 9.7 节中定义。

member-access 的形式为 E.I 或 E.I<A$_1$, ..., A$_K$>，其中 E 是 primary-expression， I 是单个标识符，<A$_1$, ..., A$_K$> 是可选的 *type-argument-list*。如果未指定 *type-argument-list* 时，则可将 K 视为零。

member-access 类型为 dynamic 的 *primary-expression* 是动态绑定的（第 7.2.2 节）。在这种情况下，编译器将成员访问归类为 dynamic 类型的属性访问。随后在运行时应用以下用于确定 *member-access* 含义

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

的规则（使用 *primary-expression* 的运行时类型而不是编译时类型）。如果此运行时分类形成方法组，则成员访问必须为 *invocation-expression* 的 *primary-expression*。

member-access 的计算和分类方式如下：

- 如果 K 为零，E 是命名空间，并且 E 包含名为 I 的嵌套命名空间，则结果为该命名空间。
- 否则，如果 E 为命名空间，并且 E 包含具有名称为 I 且有 K 个类型形参的可访问类型，则结果为利用给定类型实参构造的该类型。
- 如果 E 是一个 *predefined-type* 或一个归类为类型的 *primary-expression*，E 不是类型形参，并且在 E 中对具有 K 个类型形参的 I 进行成员查找（第 7.4 节）得到匹配项，则 E.I 的计算和分类方式如下：
  - 如果 I 标识一个类型，则结果为使用给定类型实参构造的该类型。
  - 如果 I 标识一个或多个方法，则结果为一个没有关联的实例表达式的方法组。如果指定了类型实参列表，则将在调用泛型方法时使用它（第 7.6.5.1 节）。
  - 如果 I 标识一个 static 属性，则结果为一个没有关联的实例表达式的属性访问。
  - 如果 I 标识一个 static 字段，则：
    - 如果该字段为 readonly 并且引用发生在声明该字段的类或结构的静态构造函数外，则结果为值，即 E 中静态字段 I 的值。
    - 否则，结果为变量，即 E 中的静态字段 I。
  - 如果 I 标识一个 static 事件，则：
    - 如果引用发生在声明了该事件的类或结构内，并且事件不是用 *event-accessor-declarations*（第 10.8 节）声明的，则完全将 I 视为静态字段来处理 E.I。
    - 否则，结果为没有关联的实例表达式的事件访问。
  - 如果 I 标识一个常量，则结果为值，即该常量的值。
  - 如果 I 标识枚举成员，则结果为值，即该枚举成员的值。
  - 否则，E.I 是无效成员引用，并且会出现编译时错误。
- 如果 E 是类型为 T 的属性访问、索引器访问、变量或值，并且在 T 中对具有 K 个类型实参的 I 进行成员查找（第 7.4 节）时得到匹配项，则 E.I 的计算和分类方式如下：
  - 首先，如果 E 为属性访问或索引器访问，则获取该属性访问或索引器访问的值（第 7.1.1 节），并将 E 重新归为值类别。
  - 如果 I 标识一个或多个方法，则结果为具有 E 的关联实例表达式的方法组。如果指定了类型实参列表，则将在调用泛型方法时使用它（第 7.6.5.1 节）。
  - 如果 I 标识实例属性，则结果为具有 E 的关联实例表达式的属性访问。
  - 如果 T 为 *class-type* 并且 I 标识此 *class-type* 的一个实例字段，则：
    - 如果 E 的值为 null，则将引发 System.NullReferenceException。
    - 否则，如果字段为 readonly 并且引用发生在声明字段的类的实例构造函数外，则结果为值，即 E 引用的对象中字段 I 的值。
    - 否则，结果为变量，即 E 引用的对象中的字段 I。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果 T 为 *struct-type* 并且 I 标识此 *struct-type* 的实例字段，则：
  - 如果 E 为值，或者如果字段为 readonly 并且引用发生在声明字段的结构的实例构造函数外，则结果为值，即 E 给定的结构实例中字段 I 的值。
  - 否则，结果为变量，即 E 给定的结构实例中的字段 I。
- 如果 I 标识实例事件，则：
  - 如果引用发生在声明事件的类或结构内，且事件不是用 *event-accessor-declaration*（第 10.8 节）声明的，并且引用未作为 += 或 -= 运算符的左侧内容出现，则完全将 I 视为实例字段来处理 E.I。
  - 否则，结果为具有 E 的关联实例表达式的事件访问。

- 否则，将尝试将 E.I 当作扩展方法调用（第 7.6.5.2 节）来处理。如果处理失败，则表明 E.I 是无效成员引用，并将出现绑定时错误。

7.6.4.1 相同的简单名称和类型名称

在 E.I 形式的成员访问中，如果 E 为单个标识符，并且 E 可能有两种含义：作为 *simple-name*（第 7.6.2 节）的 E，作为 type-name（第 3.8 节）的 *type-name*。只要前者所标识的对象实体（无论是常量、字段、属性、局部变量或参数）所属的类型就是以后者命名的类型，则 E 的这两种可能的含义都是允许的。在此规则下，E.I 可能有两种含义，但它们永远是明确的，因为在两种情况下，I 都必须一定是类型 E 的成员。换言之，此规则在访问 E 的静态成员和嵌套类型时，能简单地避免本来可能发生的编译时错误。例如：

```
struct Color
{
   public static readonly Color White = new Color(...);
   public static readonly Color Black = new Color(...);

   public Color Complement() {...}
}

class A
{
   public Color Color;              // Field Color of type Color

   void F() {
      Color = Color.Black;          // References Color.Black static member
      Color = Color.Complement();   // Invokes Complement() on Color field
   }

   static void G() {
      Color c = Color.White;        // References Color.White static member
   }
}
```

在类 A 中，引用 Color 类型的 Color 标识符的那些匹配项带下划线，而引用 Color 字段的那些匹配项不带下划线。

7.6.4.2 语法多义性

simple-name（第 7.6.2 节）和 *member-access*（第 7.6.4 节）的产生式可能引起表达式的语法多义性。例如，语句：

```
F(G<A,B>(7));
```

可解释为用两个实参 G < A 和 B > (7) 调用 F。或者，也可以将它解释为用一个实参调用 F，该实参是使用两个类型实参和一个常规实参对泛型方法 G 的调用。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

如果可将某个标记序列分析（在上下文中）为以 *type-argument-list*（第 7.6.2 节）结尾的 *simple-name*（第 7.6.4 节）、*member-access*（第 4.4.1 节）或 *pointer-member-access*（第 18.5.2 节），则会检查紧随结束 > 标记之后的标记。如果它是下列标记之一

```
(  )  ]  }  :  ;  ,  .  ?  ==  !=  |  ^
```

则将 *type-argument-list* 保留为 *simple-name*、*member-access* 或 *pointer-member-access* 的一部分，并丢弃该标记序列的其他任何可能的分析。否则，不将 *type-argument-list* 视为 *simple-name*、*member-access* 或 *pointer-member-access* 的一部分，即使不存在该标记序列的其他可能的分析。注意，在分析 *namespace-or-type-name*（第 3.8 节）中的 *type-argument-list* 时，将不应用这些规则。语句

```
F(G<A,B>(7));
```

将（按照此规则）被解释为使用一个实参对 F 进行调用，该实参是使用两个类型实参和一个常规实参对泛型方法 G 的调用。语句

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

都被解释为使用两个实参调用 F。语句

```
x = F < A > +y;
```

将被解释为小于运算符、大于运算符和一元加运算符，如同语句 x = (F < A) > (+y)，而不是在带 *type-argument-list* 的 *simple-name* 后面跟着一个一元加运算符。在语句

```
x = y is C<T> + z;
```

中，标记 C<T> 被解释为带 *type-argument-list* 的 *namespace-or-type-name*。

7.6.5 调用表达式

invocation-expression 用于调用方法。

invocation-expression:
primary-expression (*argument-list*$_{opt}$)

如果至少符合以下条件之一，则 *invocation-expression* 是动态绑定的（第 7.2.2 节）：

- *primary-expression* 具有编译时类型 dynamic。
- 可选 *argument-list* 中至少有一个实参具有编译时类型 dynamic，并且 *primary-expression* 没有委托类型。

在此情况下，编译器将 *invocation-expression* 归类为 dynamic 类型的值。随后在运行时应用以下用于确定 *invocation-expression* 的含义的规则（使用具有编译时类型 dynamic 的 *primary-expression* 和参数的运行时类型而不是编译时类型）。如果 *primary-expression* 没有编译时类型 dynamic，则方法调用将进行有限的编译时检查，如第 7.5.4 节所述。

invocation-expression 的 *primary-expression* 必须是方法组或 *delegate-type* 的值。如果 *primary-expression* 是方法组，则 *invocation-expression* 为方法调用（第 7.6.5.1 节）。如果 *primary-expression* 是 *delegate-type* 的值，则 *invocation-expression* 为委托调用（第 7.6.5.3 节）。如果 *primary-expression* 既非方法组亦非 *delegate-type* 的值，则会出现绑定时错误。

可选的 *argument-list*（第 7.5.1 节）列出的值或变量引用将在调用时传递给方法的参数。

invocation-expression 的计算结果按如下方式进行分类：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果 *invocation-expression* 调用的方法或委托返回 `void`，则结果为Nothing。Nothing 类别的表达式只能在 *statement-expression*（第 8.6 节）的上下文中使用或用作 *lambda-expression*（第 7.15 节）的体。否则会发生绑定时错误。
- 否则，结果是由方法或委托返回的类型的值。

7.6.5.1 方法调用

对于方法调用，*invocation-expression* 的 *primary-expression* 必须是方法组。方法组标识要调用的方法，或者标识从中选择要调用的特定方法的重载方法集。在后一种情形中，具体调用哪个方法取决于 *argument-list* 中的参数的类型所提供的上下文。

`M(A)` 形式（其中 `M` 是方法组并且可能包括 *type-argument-list*，`A` 是可选的 *argument-list*）的方法调用的绑定时处理包括以下步骤：

- 构造方法调用的候选方法集。对于与方法组 `M` 关联的每个方法 `F`：
  - 如果 `F` 是非泛型的，则在满足以下条件时，`F` 是候选方法：
    - `M` 没有类型实参列表，并且
    - 对 `A` 来说，`F` 是适用的（第 7.5.3.1 节）。
  - 如果 `F` 是泛型的，并且 `M` 没有类型实参列表，则在满足以下条件时，`F` 是候选方法：
    - 类型推断（第 7.5.2 节）成功，为该调用推断出一个类型实参列表，并且
    - 一旦使用推断出的类型实参替换对应的方法类型形参，则 F 的形参列表中的所有构造类型都满足它们的约束（第 4.4.4 节），并且对 `A` 来说，`F` 的形参列表是适用的（第 7.5.3.1 节）。
  - 如果 `F` 是泛型的，并且 `M` 包含类型实参列表，则在满足以下条件时，`F` 是候选方法：
    - `F` 具有的方法类型形参数目与类型实参列表中提供的数目相同，并且
    - 一旦使用类型实参替换对应的方法类型形参，则 F 的形参列表中的所有构造类型都满足它们的约束（第 4.4.4 节），并且对 `A` 来说，`F` 的形参列表是适用的（第 7.5.3.1 节）。
- 候选方法集被减少到仅包含派生程度最大的类型中的方法：对于该集中的每个方法 `C.F`（其中 `C` 是声明了方法 `F` 的类型），将从该集中移除在 `C` 的基类型中声明的所有方法。此外，如果 `C` 是 `object` 以外的类类型，则从该集中移除在接口类型中声明的所有方法。（仅当该方法组是具有除 `object` 以外的有效基类和非空有效接口集的类型形参上的成员查找的结果时，后一条规则才有效。）
- 如果得到的候选方法集为空，则将放弃后续步骤中的其他处理，而尝试以扩展方法调用（第 7.6.5.2 节）的形式处理该调用。如果此操作失败，则不存在适用的方法，并将出现绑定时错误。
- 使用第 7.5.3 节中的重载决策规则确定候选方法集中的最佳方法。如果无法确定单个最佳方法，则该方法调用是不明确的，并发生绑定时错误。在执行重载解析时，将在使用类型实参（提供或推断出的）替换对应的方法类型形参之后考虑泛型方法的参数。
- 所选最佳方法的***最终验证***按如下方式执行：
  - 该方法在方法组的上下文中进行验证：如果该最佳方法是静态方法，则方法组必须是从 *simple-name* 或通过某个类型从 *member-access* 产生的。如果该最佳方法为实例方法，则方法组必须是从 *simple-name*、通过某个变量或值从 *member-access* 或从 *base-access* 产生的。如果两个要求都不满足，则发生绑定时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果该最佳方法是泛型方法，则根据泛型方法上声明的约束（第 4.4.4 节）检查类型实参（提供或推断出的）。如果任何类型实参不满足类型形参上的对应约束，则会发生绑定时错误。

通过以上步骤在绑定时选定并验证了方法后，将根据第 7.5.4 节中说明的函数成员调用规则处理实际的运行时调用。

上述决策规则的直观效果如下：为找到方法调用所调用的特定方法，从方法调用指示的类型开始，在继承链中一直向上查找，直到至少找到一个适用的、可访问的、非重写的方法声明。然后对该类型中声明的适用的、可访问的、非重写的方法集执行类型推断和重载决策，并调用由此选定的方法。如果找不到方法，则改为尝试以扩展方法调用的形式处理该调用。

7.6.5.2 扩展方法调用

在以下形式之一的方法调用（第 7.5.5.1 节）中

expr . *identifier* ( )

expr . *identifier* (*args*)

expr . *identifier* < *typeargs* > ( )

expr . *identifier* < *typeargs* > (*args*)

如果正常的调用处理找不到适用的方法，则将尝试以扩展方法调用的形式处理该构造。如果 *expr* 或任意 *args* 具有编译时类型 `dynamic`，将不应用扩展方法。

目标是查找最佳的 *type-name* `C`，以便可以进行相应的静态方法调用：

`C` . *identifier* (*expr*)

`C` . *identifier* (*expr* , *args*)

`C` . *identifier* < *typeargs* > (*expr*)

`C` . *identifier* < *typeargs* > (*expr* , *args*)

如果满足以下各项，则扩展方法 $C_i.M_j$ 将***符合条件***：

- C_i 为非泛型、非嵌套类
- M_j 的名称为 *identifier*
- M_j 作为如上所示的静态方法应用于参数时是可访问且适用的
- 存在从 *expr* 到 M_j. 的第一个参数的类型的隐式标识、引用或装箱转换。

对 `C` 的搜索操作如下：

- 从最接近的封闭命名空间声明开始，接下来是每个封闭命名空间声明，最后是包含编译单元，搜索将连续进行以找到候选的扩展方法集：
  - 如果给定的命名空间或编译单元直接包含具有适当扩展方法 M_j 的非泛型类型声明 C_i，则这些扩展方法的集合为候选集。
  - 如果使用给定命名空间或编译单元中的命名空间指令导入的命名空间直接包含具有适当扩展方法 M_j 的非泛型类型声明 C_i，则这些扩展方法的集合为候选集。
- 如果在任何封闭命名空间声明或编译单元中都找不到候选集，则会出现编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，对候选集应用重载决策（如第 7.5.3 节所述）。如果找不到一个最佳方法，则会出现编译时错误。
- C 是将最佳方法声明为扩展方法的类型。

如果将 C 用作目标，则将以静态方法调用（第 7.5.4 节）的形式处理该方法调用。

上述规则表示，实例方法优先于扩展方法，内部命名空间声明中可用的扩展方法优先于外部命名空间声明中可用的扩展方法，并且直接在命名空间中声明的扩展方法优先于通过 using 命名空间指令导入该命名空间的扩展方法。例如：

```
public static class E
{
   public static void F(this object obj, int i) { }
   public static void F(this object obj, string s) { }
}

class A { }

class B
{
   public void F(int i) { }
}

class C
{
   public void F(object obj) { }
}

class X
{
   static void Test(A a, B b, C c) {
      a.F(1);           // E.F(object, int)
      a.F("hello");     // E.F(object, string)

      b.F(1);           // B.F(int)
      b.F("hello");     // E.F(object, string)

      c.F(1);           // C.F(object)
      c.F("hello");     // C.F(object)
   }
}
```

在该示例中，B 的方法优先于第一个扩展方法，而 C 的方法优先于这两个扩展方法。

```
public static class C
{
   public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
   public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
   public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
   public static class D
   {
      public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
      public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
   }
}

namespace N2
{
   using N1;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public static class E
        {
           public static void F(this int i) { Console.WriteLine("E.F({0})", i); }
        }
        class Test
        {
           static void Main(string[] args)
           {
              1.F();
              2.G();
              3.H();
           }
        }
    }
```

该示例的输出为：

```
E.F(1)
D.G(2)
C.H(3)
```

D.G 优先于 C.G，而 E.F 优先于 D.F 和 C.F。

7.6.5.3 委托调用

对于委托调用，*invocation-expression* 的 *primary-expression* 必须是 *delegate-type* 的值。另外，将 *delegate-type* 视为与 *delegate-type* 具有相同的参数列表的函数成员，*delegate-type* 对于 *invocation-expression* 的 *argument-list* 必须是适用的（第 7.5.3.1 节）。

D(A) 形式（其中 D 是 *delegate-type* 的 *primary-expression*，A 是可选的 *argument-list*）的委托调用的运行时处理包括以下步骤：

- 计算 D。如果此计算导致异常，则不执行进一步的操作。
- 检查 D 的值是否有效。如果 D 的值为 null，则将引发 System.NullReferenceException，并且不执行进一步的步骤。
- 否则，D 是一个对委托实例的引用。对该委托的调用列表中的每个可调用实体，执行函数成员调用（第 7.5.4 节）。对于由实例和实例方法组成的可调用实体，用于调用的实例是包含在可调用实体中的实例。

7.6.6 元素访问

一个 *element-access* 包括一个 *primary-no-array-creation-expression*，再后接“[”标记、*argument-list* 和“]”标记。*argument-list* 由一个或多个 *argument* 组成，各实参之间用逗号分隔。

element-access:
    primary-no-array-creation-expression [*argument-list*]

element-access 的 *argument-list* 不允许包含 ref 或 out 参数。

如果至少符合以下条件之一，则 *element-access* 是动态绑定的（第 7.2.2 节）：

- *primary-no-array-creation-expression* 具有编译时类型 dynamic。
- *argument-list* 中至少有一个表达式具有编译时类型 dynamic，并且 *primary-no-array-creation-expression* 没有数组类型。

在此情况下，编译器将 *element-access* 归类为 dynamic 类型的值。随后在运行时应用以下用于确定 *element-access* 的含义的规则（使用具有编译时类型 dynamic 的 *primary-no-array-creation-expression* 和

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

argument-list 表达式的运行时类型而不是编译时类型）。如果 *primary-no-array-creation-expression* 没有编译时类型 dynamic，则元素访问将进行有限的编译时检查，如第 7.5.4 节所述。

如果 *element-access* 的 *primary-no-array-creation-expression* 是 *array-type* 的值，则该 *element-access* 是数组访问（第 7.6.6.1 节）。否则，该 *primary-no-array-creation-expression* 必须是具有一个或多个索引器成员的类、结构或接口类型的变量或值，在这种情况下，*element-access* 为索引器访问（第 7.6.6.2 节）。

7.6.6.1 数组访问

对于数组访问，*element-access* 的 *primary-no-array-creation-expression* 必须是 *array-type* 的值。此外，数组访问的 *argument-list* 不允许包含命名参数。*argument-list* 中表达式的个数必须与 *array-type* 的秩相同，并且每个表达式都必须属于 int、uint、long、ulong 类型，或者必须可以隐式转换为这些类型中的一种或多种。

数组访问的计算结果是数组的元素类型的变量，即由 *argument-list* 中表达式的值选定的数组元素。

P[A] 形式（其中 P 是 *array-type* 的 *primary-no-array-creation-expression*，A 是 *argument-list*）的数组访问运行时处理包括以下步骤：

- 计算 P。如果此计算导致异常，则不执行进一步的操作。
- *argument-list* 的索引表达式按从左到右的顺序计算。计算每个索引表达式后，执行到下列类型之一的隐式转换（第 0 节）：int、uint、long、ulong。选择此列表中第一个存在相应隐式转换的类型。例如，如果索引表达式是 short 类型，则将执行到 int 的隐式转换，这是因为可以执行从 short 到 int 和从 short 到 long 的隐式转换。如果计算索引表达式或后面的隐式转换时导致异常，则不再进一步计算索引表达式，并且不再执行进一步的操作。
- 检查 P 的值是否有效。如果 P 的值为 null，则将引发 System.NullReferenceException，并且不执行进一步的步骤。
- 针对由 P 引用的数组实例的每个维度的实际界限，检查 *argument-list* 中每个表达式的值。如果一个或多个值超出了范围，则引发 System.IndexOutOfRangeException，并且不再执行进一步的操作。
- 计算由索引表达式给定的数组元素的位置，此位置将成为数组访问的结果。

7.6.6.2 索引器访问

对于索引器访问，*element-access* 的 *primary-no-array-creation-expression* 必须是类、结构或接口类型的变量或值，并且此类型必须实现一个或多个对于 *element-access* 的 *argument-list* 适用的索引器。

P[A] 形式（其中 P 是类、结构或接口类型 T 的一个 *primary-no-array-creation-expression*，A 是 *argument-list*）的索引器访问绑定时处理包括以下步骤：

- 构造由 T 提供的索引器集。该集由 T 或 T 的基类型中声明的所有符合下列条件的索引器组成：它们不是经 override 声明的，并且在当前上下文（第 3.5 节）中可以访问。
- 将该集缩减为那些适用的并且不被其他索引器隐藏的索引器。对该集中的每个索引器 S.I（其中 S 为声明索引器 I 的类型）应用下列规则：
  - 如果 I 对于 A（第 7.5.3.1 节）不适用，则 I 从集中移除。
  - 如果 I 对于 A（第 7.5.3.1 节）适用，则从该集中移除在 S 的基类型中声明的所有索引器。
  - 如果 I 对于 A（第 7.5.3.1 节）适用并且 S 为非 object 的类类型，则从该集中移除在接口中声明的所有索引器。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果结果候选索引器集为空，则不存在适用的索引器，并发生绑定时错误。
- 使用第 7.5.3 节中的重载决策规则确定候选索引器集中的最佳索引器。如果无法确定单个最佳索引器，则该索引器访问是不明确的，并发生绑定时错误。
- *argument-list* 的索引表达式按从左到右的顺序计算。索引器访问的处理结果是属于索引器访问类别的表达式。索引器访问表达式引用在上一步骤中确定的索引器，并具有 `P` 的关联实例表达式和 `A` 的关联参数列表。

根据索引器访问的使用上下文，索引器访问导致调用该索引器的 *get-accessor* 或 *set-accessor*。如果索引器访问是赋值的目标，则调用 *set-accessor* 以赋新值（第 7.17.1 节）。在其他所有情况下，调用 *get-accessor* 以获取当前值（第 7.1.1 节）。

7.6.7 this 访问

this-access 由保留字 `this` 组成。

this-access:
    `this`

this-access 只能在实例构造函数、实例方法或实例访问器的 *block* 中使用。它具有下列含义之一：

- 当 `this` 在类的实例构造函数内的 *primary-expression* 中使用时，它属于值类别。此时，该值的类型是使用 this 的类实例类型（第 10.3.1 节），并且该值就是对所构造的对象的引用。
- 当 `this` 在类的实例方法或实例访问器内的 *primary-expression* 中使用时，它属于值类别。此时，该值的类型是使用 this 的类实例类型（第 10.3.1 节），并且该值就是对为其调用方法或访问器的对象的引用。
- 当 `this` 在结构的实例构造函数内的 *primary-expression* 中使用时，它属于变量类别。该变量的类型是使用 this 的结构实例类型（第 10.3.1 节），并且该变量表示的正是所构造的结构。结构实例构造函数的 `this` 变量的行为与结构类型的 `out` 参数完全一样，具体而言，这表示该变量在实例构造函数的每个执行路径中必须已明确赋值。
- 当 `this` 在结构的实例方法或实例访问器内的 *primary-expression* 中使用时，它属于变量类别。该变量的类型就是使用 this 的结构实例类型（第 10.3.1 节）。
    - 如果方法或访问器不是迭代器（第 10.14 节），则 `this` 变量表示为其调用方法或访问器的结构，并且其行为与结构类型的 `ref` 参数完全相同。
    - 如果方法或访问器是迭代器，则 `this` 变量表示为其调用方法或访问器的结构的 copy，并且其行为与结构类型的 value 参数完全相同。

在以上列出的上下文以外的上下文内的 *primary-expression* 中使用 `this` 是编译时错误。具体说就是不能在静态方法、静态属性访问器中或字段声明的 *variable-initializer* 中引用 `this`。

7.6.8 基访问

base-access 由保留字 `base`，后接一个“`.`”标记和一个标识符或一个用方括号括起来的 *argument-list* 组成：

base-access:
    `base` `.` *identifier*
    `base` `[` *argument-list* `]`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

base-access 用于访问被当前类或结构中名称相似的成员隐藏的基类成员。*base-access* 只能在实例构造函数、实例方法或实例访问器的 *block* 中使用。当 `base.I` 出现在类或结构中时，`I` 必须表示该类或结构的基类的一个成员。同样，当 `base[E]` 出现在类中时，该类的基类中必须存在适用的索引器。

在绑定时，`base.I` 和 `base[E]` 形式的 *base-access* 表达式完全等价于 `((B)this).I` 和 `((B)this)[E]`（其中 `B` 是构造所涉及的类或结构的基类）。因此，`base.I` 和 `base[E]` 相当于 `this.I` 和 `this[E]`，不同的是，`this` 被视为基类的实例。

当某个 *base-access* 引用虚函数成员（方法、属性或索引器）时，确定在运行时（第 7.5.4 节）调用哪个函数成员的规则有一些更改。确定调用哪一个函数成员的方法是，查找该函数成员相对于 `B`（而不是相对于 `this` 的运行时类型，在非基访问中通常如此）的派生程度最大的实现（第 10.6.3 节）。因此，在 `virtual` 函数成员的 `override` 中，可以使用 *base-access* 调用该函数成员的被继承了的实现。如果 *base-access* 引用的函数成员是抽象的，则发生绑定时错误。

7.6.9 后缀增量和后缀减量运算符

post-increment-expression:
primary-expression `++`

post-decrement-expression:
primary-expression `--`

后缀增量或后缀减量运算符的操作数必须是属于变量、属性访问或索引器访问类别的表达式。该运算的结果是与操作数类型相同的值。

如果 *primary-expression* 具有编译时类型 `dynamic`，则运算符动态绑定（第 7.2.2 节），*post-increment-expression* 或 *post-decrement-expression* 具有编译时类型 `dynamic`，并且在运行时通过 *primary-expression* 的运行时类型应用以下规则。

如果后缀增量或减量运算的操作数是属性或索引器访问，则属性或索引器必须同时具有 `get` 和 `set` 访问器。如果不是这样，则发生绑定时错误。

将使用一元运算符重载决策（第 7.3.3 节）选择特定的运算符实现。以下类型存在预定义的 `++` 和 `--` 运算符：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 以及任何枚举类型。预定义 `++` 运算符返回的结果值为操作数加上 1，预定义 `--` 运算符返回的结果值为操作数减去 1。在 `checked` 上下文中，如果此加法或减法运算的结果在结果类型的范围之外，且结果类型为整型或枚举类型，则会引发 `System.OverflowException`。

`x++` 或 `x--` 形式的后缀增量或后缀减量运算的运行时处理包括以下步骤：

- 如果 `x` 属于变量：
  - 计算 `x` 以产生变量。
  - 保存 `x` 的值。
  - 调用选定的运算符，将 `x` 的保存值作为参数。
  - 运算符返回的值存储在由 `x` 的计算结果给定的位置中。
  - `x` 的保存值成为运算结果。
- 如果 `x` 属于属性或索引器访问：
  - 计算与 `x` 关联的实例表达式（如果 `x` 不是 `static`）和参数列表（如果 `x` 是索引器访问），结果用于后面的对 `get` 和 `set` 访问器调用。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 调用 x 的 get 访问器并保存返回的值。
- 调用选定的运算符，将 x 的保存值作为参数。
- 调用 x 的 set 访问器，将运算符返回的值作为其 value 参数。
- x 的保存值成为运算结果。

++ 和 -- 运算符也支持前缀表示法（第 7.7.5 节）。通常，x++ 或 x-- 的结果是运算“之前”x 的值，而 ++x 或 --x 的结果是运算“之后”x 的值。在任何一种情况下，运算后 x 本身都具有相同的值。

operator ++ 或 operator -- 的实现既可以用后缀表示法调用，也可以用前缀表示法调用。但是，不能让这两种表示法分别去调用该运算符的不同的实现。

7.6.10 new 运算符

new 运算符用于创建类型的新实例。

有三种形式的 new 表达式：

- 对象创建表达式用于创建类类型和值类型的新实例。
- 数组创建表达式用于创建数组类型的新实例。
- 委托创建表达式用于创建委托类型的新实例。

new 运算符表示创建类型的一个实例，但并非暗示要为它动态分配内存。具体而言，值类型的实例不要求在表示它的变量以外有额外的内存，因而，在使用 new 创建值类型的实例时不发生动态分配。

7.6.10.1 对象创建表达式

object-creation-expression 用于创建 *class-type* 或 *value-type* 的新实例。

object-creation-expression:
    new *type* (*argument-list*$_{opt}$) *object-or-collection-initializer*$_{opt}$
    new *type* *object-or-collection-initializer*

object-or-collection-initializer:
    object-initializer
    collection-initializer

object-creation-expression 的 *type* 必须是 *class-type*、*value-type* 或 *type-parameter*。该 *type* 不能是 abstract *class-type*。

仅当 *type* 为 *class-type* 或 *struct-type* 时才允许使用可选的 *argument-list*（第 7.5.1 节）。

对象创建表达式可以省略构造函数参数列表和封闭括号，前提是该表达式中包括对象初始值设定项或集合初始值设定项。省略构造函数参数列表和封闭括号与指定空的参数列表等效。

对包括对象初始值设定项或集合初始值设定项的对象创建表达式的处理包括：首先处理实例构造函数，然后处理对象初始值设定项（第 7.6.10.2 节）或集合初始值设定项（第 7.6.10.3 节）指定的成员或元素初始化。

如果可选 *argument-list* 中的任意参数具有编译时类型 dynamic，则 *object-creation-expression* 动态绑定（第 7.2.2 节），并在运行时使用 *argument-list* 中具有编译时类型 dynamic 的那些参数的运行时类型应用以下规则。但是，对象创建进行有限的编译时检查，如第 7.5.4 节所述。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

new T(A) 形式（其中 T 是 *class-type* 或 *value-type*，A 是可选 *argument-list*）的 *object-creation-expression* 的绑定时处理包括以下步骤：

- 如果 T 是 *value-type* 且 A 不存在：
  - *object-creation-expression* 是默认构造函数调用。*object-creation-expression* 的结果是 T 类型的一个值，即在第 4.1.1 节中定义的 T 的默认值。
- 否则，如果 T 是 *type-parameter* 且 A 不存在：
  - 如果还没有为 T 指定值类型约束或构造函数约束（第 10.1.5 节），则会出现绑定时错误。
  - *object-creation-expression* 的结果是类型参数所绑定到的运行时类型的值，即调用该类型的默认构造函数所产生的结果。运行时类型可以是引用类型或值类型。
- 否则，如果 T 是 *class-type* 或 *struct-type*：
  - 如果 T 是 abstract *class-type*，则会发生编译时错误。
  - 使用第 7.5.3 节中的重载决策规则确定要调用的实例构造函数。候选实例构造函数集由 T 中声明的适用于 A（第 7.5.3.1 节）的所有可访问实例构造函数组成。如果候选实例构造函数集为空，或者无法标识单个最佳实例构造函数，则发生绑定时错误。
  - *object-creation-expression* 的结果是 T 类型的值，即由调用在上面的步骤中确定的实例构造函数所产生的值。
- 否则，*object-creation-expression* 无效，并发生绑定时错误。

即使 *object-creation-expression* 是动态绑定的，编译时类型仍为 T。

new T(A) 形式（其中 T 是 *class-type* 或 *struct-type*，A 是可选 *argument-list* 的 *object-creation-expression* 的运行时处理包括以下步骤：

- 如果 T 是 *class-type*：
  - 为 T 类的一个新实例分配存储位置。如果没有足够的可用内存来为新实例分配存储位置，则引发 System.OutOfMemoryException，并且不执行进一步的操作。
  - 新实例的所有字段都将初始化为它们的默认值（第 5.2 节）。
  - 根据函数成员调用（第 7.5.4 节）的规则来调用实例构造函数。对新分配的实例的引用会自动传递给实例构造函数，因而，可以从实例构造函数中用 this 来访问该实例。
- 如果 T 是 *struct-type*：
  - 通过分配一个临时局部变量来创建类型 T 的实例。由于要求 *struct-type* 的实例构造函数为所创建的实例的每个字段明确赋值，因此不需要初始化此临时变量。
  - 根据函数成员调用（第 7.5.4 节）的规则来调用实例构造函数。对新分配的实例的引用会自动传递给实例构造函数，因而，可以从实例构造函数中用 this 来访问该实例。

7.6.10.2 对象初始值设定项

对象初始值设定项 (object initializer) 为某个对象的零个或多个字段或属性指定值。

object-initializer:
{ *member-initializer-list*$_{opt}$ }
{ *member-initializer-list* , }

member-initializer-list:
member-initializer
member-initializer-list , *member-initializer*

member-initializer:
identifier = *initializer-value*

initializer-value:
expression
object-or-collection-initializer

对象初始值设定项包含一系列成员初始值设定项，它们括在“{”和“}”标记中并且用“,”分隔。每个成员初始值设定项必须命名所初始化的对象的可访问字段或属性，后接等号以及表达式或者对象初始值设定项或集合初始值设定项。如果对象初始值设定项对于同一个字段或属性包括多个成员初始值设定项，则会发生错误。对象初始值设定项无法引用它所初始化的新创建的对象。

在等号后面指定表达式的成员初始值设定项的处理方式与对字段或属性赋值（第 7.17.1 节）的方式相同。

在等号后面指定对象初始值设定项的成员初始值设定项是嵌套对象初始值设定项 (nested object initializer)，即嵌入对象的初始化。嵌套对象初始值设定项中的赋值不是对字段或属性赋新值，这些赋值被视为对字段或属性的成员的赋值。嵌套对象初始值设定项不能应用于具有值类型的属性，也不能应用于具有值类型的只读字段。

在等号后面指定集合初始值设定项的成员初始值设定项是嵌入集合的初始化。初始值设定项中给出的元素不是将新集合赋给字段或属性，这些元素将被添加到字段或属性引用的集合中。字段或属性必须是符合第 7.6.10.3 节中指定的要求的集合类型。

下面的类表示一个具有两个坐标的点：

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

   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

可以使用下面的语句创建和初始化 `Point` 的实例：

```
Point a = new Point { X = 0, Y = 1 };
```

此语句与下面的语句等效

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

其中，`__a` 是以其他方式不可见且不可访问的临时变量。下面的类表示通过两个点创建的一个矩形。

```
public class Rectangle
{
   Point p1, p2;

   public Point P1 { get { return p1; } set { p1 = value; } }
   public Point P2 { get { return p2; } set { p2 = value; } }
}
```

可以使用下面的语句创建和初始化 `Rectangle` 的实例：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
Rectangle r = new Rectangle {
   P1 = new Point { X = 0, Y = 1 },
   P2 = new Point { X = 2, Y = 3 }
};
```

此语句与下面的语句等效

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

其中 __r、__p1 和 __p2 是以其他方式不可见且不可访问的临时变量。

如果 Rectangle's 的构造函数分配以下两个嵌入式 Point 实例

```
public class Rectangle
{
   Point p1 = new Point();
   Point p2 = new Point();

   public Point P1 { get { return p1; } }
   public Point P2 { get { return p2; } }
}
```

则以下构造可用于初始化嵌入的 Point 实例（而非为新实例赋值）：

```
Rectangle r = new Rectangle {
   P1 = { X = 0, Y = 1 },
   P2 = { X = 2, Y = 3 }
};
```

此语句与下面的语句等效

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

7.6.10.3 集合初始值设定项

集合初始值设定项指定集合中的元素。

collection-initializer:
{ *element-initializer-list* }
{ *element-initializer-list* , }

element-initializer-list:
element-initializer
element-initializer-list , *element-initializer*

element-initializer:
non-assignment-expression
{ *expression-list* }

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

expression-list:
expression
expression-list , *expression*

集合初始值设定项包含一系列元素初始值设定项，它们括在“{”和“}”标记中并且用“,”分隔。每个元素初始值设定项指定要添加到所初始化的集合对象中的元素，它由括在“{”和“}”标记中并且用“,”分隔的表达式列表组成。写入单个表达式元素初始值设定项时可以不使用大括号，但不能是赋值表达式，以避免与成员初始值设定项产生歧义。*non-assignment-expression* 产生式是在第 7.18 节中定义的。

下面是包括集合初始值设定项的对象创建表达式的一个示例：

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

集合初始值设定项应用到的集合对象必须是实现 System.Collections.IEnumerable 的类型，否则会出现编译时错误。对于按顺序指定的每个元素，集合初始值设定项将调用目标对象的 Add 方法（将元素初始值设定项的表达式列表用作参数列表），从而对每个调用都应用正常重载决策。因此对于每个元素初始值设定项，集合对象必须包含适用的 Add 方法。

下面的类表示一个联系人，包括姓名和电话号码列表：

```
public class Contact
{
   string name;
   List<string> phoneNumbers = new List<string>();
   public string Name { get { return name; } set { name = value; } }
   public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

可以使用如下语句创建和初始化 List<Contact>：

```
var contacts = new List<Contact> {
   new Contact {
      Name = "Chris Smith",
      PhoneNumbers = { "206-555-0101", "425-882-8080" }
   },
   new Contact {
      Name = "Bob Harris",
      PhoneNumbers = { "650-555-0199" }
   }
};
```

此语句与下面的语句等效

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

其中 __clist、__c1 和 __c2 是以其他方式不可见且不可访问的临时变量。

7.6.10.4 数组创建表达式

array-creation-expression 用于创建 *array-type* 的新实例。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

array-creation-expression:
new *non-array-type* [*expression-list*] *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
new *array-type array-initializer*
new *rank-specifier array-initializer*

第一种形式的数组创建表达式分配一个数组实例，其类型是从表达式列表中删除每个表达式所得到的类型。例如，数组创建表达式 `new int[10, 20]` 产生 `int[,]` 类型的数组实例，数组创建表达式 `new int[10][,]` 产生 `int[][,]` 类型的数组。表达式列表中的每个表达式必须属于 `int`、`uint`、`long` 或 `ulong` 类型，或者可以隐式转换为一种或多种这些类型。每个表达式的值确定新分配的数组实例中相应维度的长度。由于数组维度的长度必须非负，因此，当表达式列表中出现带有负值的 *constant-expression* 时，将出现一个编译时错误。

除了在不安全的上下文（第 18.1 节）中外，数组的布局是未指定的。

如果第一种形式的数组创建表达式包含数组初始值设定项，则表达式列表中的每个表达式必须是常量，并且表达式列表指定的秩和维度长度必须匹配数组初始值设定项的秩和维度长度。

在第二种或第三种形式的数组创建表达式中，指定数组类型的秩或秩说明符必须匹配数组初始值设定项的秩。各维度长度从数组初始值设定项的每个对应嵌套层数中的元素数推断出。因此，表达式

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

完全对应于

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

第三种形式的数组创建表达式称为隐式类型化数组创建表达式 (implicitly typed array creation expression)。这种形式与第二种形式类似，不同的是数组的元素类型未显式指定，而是被确定为数组初始值设定项中表达式集的最通用类型（第 7.5.2.14 节）。对于多维数组（即其中的 *rank-specifier* 包含至少一个逗号的数组），此集由嵌套 *array-initializer* 中找到的所有 *expression* 组成。

数组初始值设定项的介绍详见第 12.6 节。

数组创建表达式的计算结果属于值类别，即对新分配的数组实例的一个引用。数组创建表达式的运行时处理包括以下步骤：

- *expression-list* 的维度长度表达式按从左到右的顺序计算。计算每个表达式后，执行到下列类型之一的隐式转换（第 0 节）：`int`、`uint`、`long`、`ulong`。选择此列表中第一个存在相应隐式转换的类型。如果表达式计算或后面的隐式转换导致异常，则不计算其他表达式，并且不执行其他步骤。
- 维度长度的计算值按下面这样验证。如果一个或多个值小于零，则引发 `System.OverflowException` 并且不执行进一步的步骤。
- 分配具有给定维度长度的数组实例。如果没有足够的可用内存来为新实例分配存储位置，则引发 `System.OutOfMemoryException`，并且不执行进一步的操作。
- 将新数组实例的所有元素初始化为它们的默认值（第 5.2 节）。
- 如果数组创建表达式包含数组初始值设定项，则计算数组初始值设定项中的每个表达式的值，并将该值赋值给它的相应数组元素。计算和赋值按数组初始值设定项中各表达式的写入顺序执行，换言之，按递增的索引顺序初始化元素，最右边的维度首先增加。如果给定表达式的计算或其面向相应数组元素的赋值导致异常，则不初始化其他元素（剩余的元素将因此具有它们的默认值）。

数组创建表达式允许实例化一个数组，并且它的元素也属于数组类型，但必须手动初始化这类数组的元素。例如，语句

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
int[][] a = new int[100][];
```

创建一个包含 100 个 `int[]` 类型的元素的一维数组。每个元素的初始值为 `null`。想让数组创建表达式同时也实例化它所指定的子数组是不可能的，因而，语句

```
int[][] a = new int[100][5];     // Error
```

会导致编译时错误。实例化子数组必须改为手动执行，如下所示

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

当多个数组中的某个数组具有“矩形”形状时，即当子数组全都具有相同的长度时，使用多维数组更有效。在上面的示例中，实例化一个数组的数组时，实际上创建了 101 个对象（1 个外部数组和 100 个子数组）。相反，

```
int[,] = new int[100, 5];
```

只创建单个对象（即一个二维数组）并在单个语句中完成分配。

下面是隐式类型化的数组创建表达式的示例：

```
var a = new[] { 1, 10, 100, 1000 };                      // int[]
var b = new[] { 1, 1.5, 2, 2.5 };                        // double[]
var c = new[,] { { "hello", null }, { "world", "!" } };  // string[,]
var d = new[] { 1, "one", 2, "two" };                    // Error
```

最后一个表达式会导致编译时错误，原因是 `int` 和 `string` 都不可隐式转换为其他类型，因而不存在最通用类型。在这种情况下，必须使用显式类型化的数组创建表达式，例如将类型指定为 `object[]`。也可以将其中某个元素强制转换为公共基类型，之后该类型将成为推断出的元素类型。

隐式类型化的数组创建表达式可以与匿名对象初始值设定项（第 7.6.10.6 节）组合，以创建匿名类型化的数据结构。例如：

```
var contacts = new[] {
   new {
      Name = "Chris Smith",
      PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
   },
   new {
      Name = "Bob Harris",
      PhoneNumbers = new[] { "650-555-0199" }
   }
};
```

7.6.10.5 委托创建表达式

delegate-creation-expression 用于创建 *delegate-type* 的新实例。

delegate-creation-expression:
`new` *delegate-type* `(` *expression* `)`

委托创建表达式的参数必须是属于编译时类型 `dynamic` 或 *delegate-type* 的方法组、匿名函数或值。如果参数是方法组，则它标识方法和（对于实例方法）为其创建委托的对象。如果实参是匿名函数，则它直接定义委托目标的形参和方法体。如果实参是值，则它标识要创建其副本的委托实例。

如果 *expression* 具有编译时类型 `dynamic`，则 *delegate-creation-expression* 动态绑定（第 7.2.2 节），并在运行时使用 *expression* 的运行时类型应用以下规则。否则在编译时应用这些规则。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

new D(E) 形式（其中 D 是 *delegate-type*，E 是 *expression*）的 *delegate-creation-expression* 的绑定时处理包括以下步骤：

- 如果 E 为方法组，则委托创建表达式的处理方式与从 E 到 D 的方法组转换（第 6.6 节）方式相同。
- 如果 E 为匿名函数，则委托创建表达式的处理方式与从 E 到 D 的匿名函数转换（第 0 节）方式相同。
- 如果 E 是值，则 E 必须与 D 兼容（第 15.1 节），并且结果为对新创建的 D 类型委托的引用（引用的调用列表与 E 相同）。如果 E 与 D 不一致，则会发生编译时错误。

new D(E) 形式（其中 D 是 *delegate-type*，E 是 *expression*）的 *delegate-creation-expression* 的运行时处理包括以下步骤：

- 如果 E 为方法组，则委托创建表达式的计算方式与从 E 到 D 的方法组转换（第 6.6 节）方式相同。
- 如果 E 为匿名函数，则委托创建的计算方式与从 E 到 D 的匿名函数转换（第 0 节）方式相同。
- 如果 E 是 *delegate-type* 的值：
  - 计算 E。如果此计算导致异常，则不执行进一步的操作。
  - 如果 E 的值为 null，则将引发 System.NullReferenceException，并且不执行进一步的步骤。
  - 为委托类型 D 的一个新实例分配存储位置。如果没有足够的可用内存来为新实例分配存储位置，则引发 System.OutOfMemoryException，并且不执行进一步的操作。
  - 用与 E 给定的委托实例相同的调用列表初始化新委托实例。

委托的调用列表在实例化委托时确定并在委托的整个生存期期间保持不变。换句话说，一旦创建了委托，就不可能更改它的可调用目标实体。当组合两个委托或从一个委托中移除另一个委托（第 15.1 节）时，将产生新委托；现有委托的内容不更改。

不可能创建引用属性、索引器、用户定义的运算符、实例构造函数、析构函数或静态构造函数的委托。

如上所述，当从方法组创建一个委托时，需根据该委托的形参表和返回类型来确定要选择的重载方法。在下面的示例中

```
delegate double DoubleFunc(double x);

class A
{
   DoubleFunc f = new DoubleFunc(Square);

   static float Square(float x) {
      return x * x;
   }

   static double Square(double x) {
      return x * x;
   }
}
```

A.f 字段将由引用第二个 Square 方法的委托初始化，因为该方法与 DoubleFunc 的形参表和返回类型完全匹配。如果第二个 Square 方法不存在，则将发生编译时错误。

7.6.10.6 匿名对象创建表达式

anonymous-object-creation-expression 用于创建匿名类型的对象。

```
anonymous-object-creation-expression:
   new  anonymous-object-initializer
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

anonymous-object-initializer:
    { *member-declarator-list*$_{opt}$ }
    { *member-declarator-list* , }

member-declarator-list:
    member-declarator
    member-declarator-list , *member-declarator*

member-declarator:
    simple-name
    member-access
    base-access
    identifier = *expression*

匿名对象初始值设定项声明一个匿名类型并返回该类型的一个实例。匿名类型是直接从 object 继承的无名类类型。匿名类型的成员是只读属性序列，这些属性从用于创建该类型实例的匿名对象初始值设定项推断。具体而言，以下形式的匿名对象初始值设定项

```
new { p1 = e1 , p2 = e2 , ··· pn = en }
```

声明一个以下形式的匿名类型

```
class __Anonymous1
{
   private readonly T1 f1 ;
   private readonly T2 f2 ;
   ···
   private readonly Tn fn ;

   public __Anonymous1(T1 a1, T2 a2, ···, Tn an) {
      f1 = a1 ;
      f2 = a2 ;
      ···
      fn = an ;
   }

   public T1 p1 { get { return f1 ; } }
   public T2 p2 { get { return f2 ; } }
   ···
   public Tn pn { get { return fn ; } }

   public override bool Equals(object __o) { ··· }
   public override int GetHashCode() { ··· }
}
```

其中每个 T_x 都是对应表达式 e_x 的类型。*member-declarator* 中使用的表达式必须具有类型。因此，*member-declarator* 中的表达式为 null 或匿名函数会产生编译时错误。表达式具有不安全类型也是编译时错误。

匿名类型及其 Equals 方法的参数的名称由编译器自动生成，不能在程序文本中引用。

在同一程序内，如果两个匿名对象初始值设定项以相同顺序指定具有相同名称和编译时类型的属性的序列，则它们会生成相同匿名类型的实例。

在下面的示例中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

最后一行的赋值是允许的，原因是 p1 和 p2 属于同一匿名类型。

匿名类型的 Equals 和 GetHashcode 方法将重写从 object 继承的方法，并根据属性的 Equals 和 GetHashcode 进行定义，以便当且仅当同一匿名类型的两个实例的所有属性都相等时，该两个实例才相等。

成员声明符可以缩写为简单名称（第 7.5.2 节）、成员访问（第 7.5.4 节）或基访问（第 7.6.8 节）。这称为投影初始值设定项 (projection initializer)，且为具有相同名称的属性的声明和赋值的简写形式。具体而言，以下形式的成员声明符

identifier *expr* . *identifier*

分别完全等效于以下内容：

identifer = *identifier* *identifier* = *expr* . *identifier*

因此，在投影初始值设定项中，*identifier* 选择值以及将值赋予的字段或属性。从直观上看，投影初始值设定项项目不仅是值，而且是值的名称。

7.6.11 typeof 运算符

typeof 运算符用于获取某种类型的 System.Type \b \b 对象。

typeof-expression:
typeof (*type*)
typeof (*unbound-type-name*)
typeof (void)

unbound-type-name:
identifier *generic-dimension-specifier*$_{opt}$
identifier :: *identifier* *generic-dimension-specifier*$_{opt}$
unbound-type-name . *identifier* *generic-dimension-specifier*$_{opt}$

generic-dimension-specifier:
< *commas*$_{opt}$ >

commas:
,
commas ,

typeof-expression 的第一种形式由 typeof 关键字后接带括号的 *type* 组成。这种形式的表达式的结果是与给定的类型对应的 System.Type 对象。任何给定的类型都只有一个 System.Type 对象。这意味着对于类型 T，typeof(T) == typeof(T) 始终为 true。*type* 不能为 dynamic。

typeof-expression 的第二种形式由 typeof 关键字后接带括号的 *unbound-type-name* 组成。*unbound-type-name* 与 *type-name*（第 3.8 节）非常类似，不同的是，*unbound-type-name* 包含 *generic-dimension-specifier*，而 *type-name* 则包含 *type-argument-list*。当 *typeof-expression* 的操作数为一系列同时满足 *unbound-type-name* 和 *type-name* 语法的标记（即该操作数既不包含 *generic-dimension-specifier* 也不包含 *type-argument-list*）时，可将标记序列视为 *type-name*。*unbound-type-name* 的含义按下述步骤确定：

- 通过将每个 *generic-dimension-specifier* 替换为与每个 *type-argument-list* 具有相同数目的逗号和关键字 object 的 *type-argument*，从而将标记序列转换为 *type-name*。
- 计算结果 *type-name*，同时忽略所有类型形参约束。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- *unbound-type-name* 解析为与结果构造类型关联的未绑定的泛型类型（第 4.4.3 节）。

typeof-expression 的结果是所产生的未绑定泛型类型的 System.Type 对象。

typeof-expression 的第三种形式由 typeof 关键字后接带括号的 void 关键字组成。这种形式的表达式的结果是一个表示“类型不存在”的 System.Type 对象。这种通过 typeof(void) 返回的类型对象与为任何类型返回的类型对象截然不同。这种特殊的类型对象在这样的类库中很有用：它允许在源语言中能仔细考虑一些方法，希望有一种方式以用 System.Type 的实例来表示任何方法（包括 void 方法）的返回类型。

typeof 运算符可以在类型形参上使用。结果为绑定到该类型形参的运行时类型的 System.Type 对象。typeof 运算符还可以在构造类型或未绑定的泛型类型上使用（第 4.4.3 节）。未绑定的泛型类型的 System.Type 对象与实例类型的 System.Type 对象不同。实例类型在运行时始终为封闭构造类型，因此其 System.Type 对象依赖于使用的运行时类型参数，而未绑定泛型类型没有类型参数。

下面的示例

```
using System;

class X<T>
{
   public static void PrintTypes() {
      Type[] t = {
         typeof(int),
         typeof(System.Int32),
         typeof(string),
         typeof(double[]),
         typeof(void),
         typeof(T),
         typeof(X<T>),
         typeof(X<X<T>>),
         typeof(X<>)
      };
      for (int i = 0; i < t.Length; i++) {
         Console.WriteLine(t[i]);
      }
   }
}

class Test
{
   static void Main() {
      X<int>.PrintTypes();
   }
}
```

产生下列输出：

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

请注意，int 和 System.Int32 是相同的类型。

还要注意，typeof(X<>) 的结果不依赖于类型参数，而 typeof(X<T>) 的结果则依赖。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.6.12 checked 和 unchecked 运算符

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

checked-expression:
checked (*expression*)

unchecked-expression:
unchecked (*expression*)

checked 运算符在 checked 上下文中计算所包含的表达式，unchecked 运算符在 unchecked 上下文中计算所包含的表达式。除了在给定的溢出检查上下文中计算所包含的表达式外，*checked-expression* 或 *unchecked-expression* 表达式与 *parenthesized-expression*（第 7.6.3 节）完全对应。

也可以通过 checked 和 unchecked 语句（第 8.11 节）控制溢出检查上下文。

下列运算受由 checked 和 unchecked 运算符和语句所确定的溢出检查上下文影响：

- 预定义的 ++ 和 -- 一元运算符（第 7.6.9 节和第 7.7.5 节）（当操作数为整型时）。
- 预定义的 - 一元运算符（第 7.7.2 节）（当操作数为整型时）。
- 预定义的 +、-、* 和 / 二元运算符（第 7.8 节）（当两个操作数均为整型时）。
- 从一个整型到另一个整型或从 float 或 double 到整型的显式数值转换（第 6.2.1 节）。

当上面的运算之一产生的结果太大，无法用目标类型表示时，执行运算的上下文控制由此引起的行为：

- 在 checked 上下文中，如果运算发生在一个常量表达式（第 7.19 节）中，则发生编译时错误。否则，当在运行时执行运算时，引发 System.OverflowException。
- 在 unchecked 上下文中，计算的结果被截断，放弃不适合目标类型的任何高序位。

对于不用任何 checked 或 unchecked 运算符或语句括起来的非常量表达式（在运行时计算的表达式），除非外部因素（如编译器开关和执行环境配置）要求 checked 计算，否则默认溢出检查上下文为 unchecked。

对于常量表达式（可以在编译时完全计算的表达式），默认溢出检查上下文总是 checked。除非将常量表达式显式放置在 unchecked 上下文中，否则在表达式的编译时计算期间发生的溢出总是导致编译时错误。

匿名函数体不受运行该匿名函数的 checked 或 unchecked 上下文的影响。

在下面的示例中

```
class Test
{
   static readonly int x = 1000000;
   static readonly int y = 1000000;

   static int F() {
      return checked(x * y);     // Throws OverflowException
   }

   static int G() {
      return unchecked(x * y);   // Returns -727379968
   }

   static int H() {
      return x * y;              // Depends on default
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由于在编译时没有表达式可以计算，所以不报告编译时错误。在运行时，F 方法将引发 System.OverflowException，而 G 方法将返回 –727379968（从超出范围的结果中取较低的 32 位）。H 方法的行为取决于编译时设定的默认溢出检查上下文，但它不是与 F 相同就是与 G 相同。

在下面的示例中

```
class Test
{
   const int x = 1000000;
   const int y = 1000000;

   static int F() {
      return checked(x * y);      // Compile error, overflow
   }

   static int G() {
      return unchecked(x * y);    // Returns -727379968
   }

   static int H() {
      return x * y;               // Compile error, overflow
   }
}
```

在计算 F 和 H 中的常量表达式时发生的溢出导致报告编译时错误，原因是表达式是在 checked 上下文中计算的。在计算 G 中的常量表达式时也发生溢出，但由于计算是在 unchecked 上下文中发生的，所以不报告溢出。

checked 和 unchecked 运算符只影响原文包含在“(”和“)”标记中的那些运算的溢出检查上下文。这些运算符不影响因计算包含的表达式而调用的函数成员。在下面的示例中

```
class Test
{
   static int Multiply(int x, int y) {
      return x * y;
   }

   static int F() {
      return checked(Multiply(1000000, 1000000));
   }
}
```

在 F 中使用 checked 不影响 Multiply 中的 x * y 计算，因此将在默认溢出检查上下文中计算 x * y。

当以十六进制表示法编写有符号整型的常量时，unchecked 运算符很方便。例如：

```
class Test
{
   public const int AllBits = unchecked((int)0xFFFFFFFF);

   public const int HighBit = unchecked((int)0x80000000);
}
```

上面的两个十六进制常量均为 uint 类型。因为这些常量超出了 int 范围，所以如果不使用 unchecked 运算符，强制转换到 int 将产生编译时错误。

checked 和 unchecked 运算符和语句允许程序员控制一些数值计算的某些方面。当然，某些数值运算符的行为取决于其操作数的数据类型。例如，两个小数相乘总是导致溢出异常，即使是在显式 unchecked 构造内也如此。同样，两个浮点数相乘从不会导致溢出异常，即使是在显式 checked 构造内也如此。另外，其他运算符从不受检查模式（不管是默认的还是显式的）的影响。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.6.13 默认值表达式

默认值表达式用于获取某个类型的默认值（第5.2 节）。通常，默认值表达式用于类型参数，因为可能并不知道类型参数是值类型还是引用类型。（不存在从 null 文本到类型参数的转换，除非类型参数已知为引用类型。）

default-value-expression:
    default (*type*)

如果 *default-value-expression* 中的 *type* 在运行时计算为引用类型，则结果将为转换为该类型的 null。如果 *default-value-expression* 中的 *type* 在运行时计算为值类型，则结果为 *value-type* 的默认值（第 4.1.2 节）。

如果类型为引用类型或已知为引用类型（第 7.19 节）的类型参数，则 *default-value-expression* 为常量表达式（第 10.1.5 节）。此外，如果类型为以下值类型之一，则 *default-value-expression* 也为常量表达式：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool 或任何枚举类型。

7.6.14 匿名方法表达式

anonymous-method-expression 是用于定义匿名函数的两种方式之一。有关这些内容的进一步介绍详见第 7.15 节。

7.7 一元运算符

+、-、!、~、++、--、cast 和 await 运算符被称为一元运算符。

unary-expression:
    primary-expression
    + *unary-expression*
    - *unary-expression*
    ! *unary-expression*
    ~ *unary-expression*
    pre-increment-expression
    pre-decrement-expression
    cast-expression
    await-expression

如果 *unary-expression* 的操作数具有编译时类型 dynamic，则它是动态绑定的（第 7.2.2 节）。在此情况下，*unary-expression* 的编译时类型为 dynamic，并且会在运行时使用操作数的运行时类型进行下面所述的决策。

7.7.1 一元加运算符

对于 +x 形式的运算，应用一元运算符重载决策（第 7.3.3 节）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。预定义的一元加运算符为：

```
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);
```

对于这些运算符，结果只是操作数的值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.7.2 一元减运算符

对于 –x 形式的运算，应用一元运算符重载决策（第 7.3.3 节）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。预定义的否定运算符为：

- 整数否定：

```
int operator –(int x);
long operator –(long x);
```

  通过从 0 中减去 x 来计算结果。如果 x 的值是操作数类型的最小可表示值（对 int 是 -2^{31}，对 long 是 -2^{63}），则 x 的算术否定在操作数类型中不可表示。如果这种情况发生在 checked 上下文中，则引发 System.OverflowException；如果它发生在 unchecked 上下文中，则结果是操作数的值而且不报告溢出。

  如果否定运算符的操作数为 uint 类型，则它转换为 long 类型，并且结果的类型为 long。有一个例外，那就是允许将 int 值 –2147483648 (-2^{31}) 写为十进制整数（第 2.4.4.2 节）的规则。

  如果否定运算符的操作数为 ulong 类型，则发生编译时错误。有一个例外，那就是允许将 long 值 –9223372036854775808 (-2^{63}) 写为十进制整数（第 2.4.4.2 节）的规则。

- 浮点否定：

```
float operator –(float x);
double operator –(double x);
```

  结果是符号被反转的 x 的值。如果 x 为 NaN，则结果也为 NaN。

- 小数否定：

```
decimal operator –(decimal x);
```

  通过从 0 中减去 x 来计算结果。小数否定等效于使用 System.Decimal 类型的一元减运算符。

7.7.3 逻辑否定运算符

对于 !x 形式的运算，应用一元运算符重载决策（第 7.3.3 节）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。只存在一个预定义的逻辑否定运算符：

```
bool operator !(bool x);
```

此运算符计算操作数的逻辑否定：如果操作数为 true，则结果为 false。如果操作数为 false，则结果为 true。

7.7.4 按位求补运算符

对于 ~x 形式的运算，应用一元运算符重载决策（第 7.3.3 节）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果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);
```

对于每个运算符，运算结果为 x 的按位求补。

每个 E 枚举类型都隐式地提供下列按位求补运算符：

```
E operator ~(E x);
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

计算 ~x（其中 x 是基础类型为 U 的枚举类型 E 的表达式）的结果正好与计算 (E)(~(U)x) 的结果相同，但始终执行到 E 的转换，如同在 unchecked 上下文（第 7.6.12 节）中一样。

7.7.5 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属于变量、属性访问或索引器访问类别的表达式。该运算的结果是与操作数类型相同的值。

如果前缀增量或减量运算的操作数是属性或索引器访问，则属性或索引器必须同时具有 get 和 set 访问器。如果不是这样，则发生绑定时错误。

将使用一元运算符重载决策（第 7.3.3 节）选择特定的运算符实现。以下类型存在预定义的 ++ 和 -- 运算符：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal 以及任何枚举类型。预定义 ++ 运算符返回的结果值为操作数加上 1，预定义 -- 运算符返回的结果值为操作数减去 1。在 checked 上下文中，如果此加法或减法运算的结果在结果类型的范围之外，且结果类型为整型或枚举类型，则会引发 System.OverflowException。

++x 或 --x 形式的前缀增量或前缀减量运算的运行时处理包括以下步骤：

- 如果 x 属于变量：
  - 计算 x 以产生变量。
  - 调用选定的运算符，将 x 的值作为参数。
  - 运算符返回的值存储在由 x 的计算结果给定的位置中。
  - 运算符返回的值成为该运算的结果。
- 如果 x 属于属性或索引器访问：
  - 计算与 x 关联的实例表达式（如果 x 不是 static）和参数列表（如果 x 是索引器访问），结果用于后面的对 get 和 set 访问器调用。
  - 调用 x 的 get 访问器。
  - 调用选定的运算符，将 get 访问器返回的值作为参数。
  - 调用 x 的 set 访问器，将运算符返回的值作为其 value 参数。
  - 运算符返回的值成为该运算的结果。

++ 和 -- 运算符也支持后缀表示法（第 7.6.9 节）。通常，x++ 或 x-- 的结果是运算“之前”x 的值，而 ++x 或 --x 的结果是运算“之后”x 的值。在任何一种情况下，运算后 x 本身都具有相同的值。

operator ++ 或 operator -- 的实现既可以用后缀表示法调用，也可以用前缀表示法调用。但是，不能让这两种表示法分别去调用该运算符的不同的实现。

7.7.6 强制转换表达式

cast-expression 用于将表达式显式转换为给定类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

cast-expression:
(*type*) *unary-expression*

(T)E 形式（其中 T 是 *type*，E 是 *unary-expression*）的 *cast-expression* 执行把 E 的值转换到类型 T 的显式转换（第 6.2 节）。如果不存在从 E 到 T 的显式转换，将发生绑定时错误。否则，结果为显式转换产生的值。即使 E 表示变量，结果也总是为值类别。

cast-expression 的语法可能导致某些语法多义性。例如，表达式 (x)－y 既可以按 *cast-expression* 解释（－y 到类型 x 的强制转换），也可以按结合了 *parenthesized-expression* 的 *additive-expression* 解释（计算 x － y) 的值）。

为解决 *cast-expression* 多义性问题，存在以下规则：仅当以下条件至少有一条成立时，括在括号中的一个或多个 *token*（第 2.3.3 节）序列才被视为 *cast-expression* 的开头：

- 标记的序列对于 *type* 是正确的语法，但对于 *expression* 则不是。
- 标记的序列对于 *type* 是正确的语法，而且紧跟在右括号后面的标记是标记“~”、标记“!”、标记“(”、*identifier*（第 2.4.1 节）、*literal*（第 2.4.4 节）或除 as 和 is 外的任何 *keyword*（第 2.4.3 节）。

上面出现的术语“正确的语法”仅指标记的序列必须符合特定的语法产生式。它并没有特别考虑任何构成标识符的实际含义。例如，如果 x 和 y 是标识符，则 x.y 对于类型是正确的语法，即使 x.y 实际并不表示类型。

从上述消除歧义规则可以得出下述结论：如果 x 和 y 是标识符，则 (x)y、(x)(y) 和 (x)(-y) 为 *cast-expression*，但 (x)-y 不是，即使 x 标识的是类型，也同样如此。然而，如果 x 是一个标识预定义类型（如 int）的关键字，则所有四种形式均为 *cast-expression*（因为这种关键字本身不可能是表达式）。

7.7.7 Await 表达式

await 运算符用于挂起封闭的异步函数的计算，直到由操作数表示的异步操作已完成。

await-expression:
await *unary-expression*

await-expression 只允许在异步函数（第 **错误!未找到引用源。** 节）体中使用。在最近的封闭异步函数内，*await-expression* 不能出现在以下位置：

- 嵌套（非异步）匿名函数内部
- *try-statement* 的 catch 或 finally 块中
- *lock-statement* 块内部
- 不安全上下文中

请注意，*await-expression* 不能出现在 *query-expression* 内的大多数位置，因为这些位置将在语法上转换为使用非异步 lambda 表达式。

在异步函数内部，await 不能用作标识符。因此，await 表达式和涉及标识符的各种表达式之间不存在语法多义性。在异步函数外部，await 充当正常标识符。

await-expression 的操作数称为***任务***。它表示不一定在计算 *await-expression* 时完成的异步操作。await 运算符的用途是挂起封闭式异步函数的执行，直到等待的任务完成，然后获取其结果。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.7.7.1 可等待的表达式

await 表达式任务必须是*可等待的*。如果存在以下情况之一，则表达式 *t* 是可等待的：

- *t* is of compile time type dynamic
- *t* has an accessible instance or extension method called GetAwaiter with no parameters and no type parameters, and a return type *A* for which all of the following hold:
  - *A* 实现了接口 System.Runtime.CompilerServices.INotifyCompletion（为简洁起见，以下简称为 INotifyCompletion）
  - *A* 具有一个可访问的可读实例属性 IsCompleted，其类型为 bool
  - *A* 具有一个不带参数和类型参数的可访问实例方法 GetResult

GetAwaiter 方法的用途是获取任务的***awaiter***。类型 *A* 称为 await 表达式的***awaiter 类型***。

IsCompleted 属性的用途是确定任务是否已完成。如果已完成，则无需挂起计算。

INotifyCompletion.OnCompleted 方法的用途是向任务注册“继续符”；即类型为 System.Action 的委托，任务完成时将调用该委托。

GetResult 方法的用途是获取任务完成后的结果。此结果可能会是“成功完成”，并可能带有结果值，也可能是一个异常（由 GetResult 方法引发）。

7.7.7.2 await 表达式的分类

表达式 await *t* 的分类方式与表达式 (*t*).GetAwaiter().GetResult() 相同。因此，如果 GetResult 的返回类型是 void，则可将 *await-expression* 归类为 Nothing。如果它具有非 void 返回类型 *T*，则可将 *await-expression* 归类为 *T* 类型的值。

7.7.7.3 await 表达式的运行时计算

在运行时，按以下方式计算表达式 await *t*：

- 通过计算表达式 (*t*).GetAwaiter() 来获取 awaiter *a*。
- 通过计算表达式 (*a*).IsCompleted 来获取 bool *b*。
- 如果 *b* 为 false，则计算将取决于 *a* 是否实现了接口 System.Runtime.CompilerServices.ICriticalNotifyCompletion（为简洁起见，以下简称 ICriticalNotifyCompletion）。此检查将在绑定时完成；即如果 *a* 具有编译时类型 dynamic，则在运行时完成，否则在编译时完成。让 *r* 表示恢复委托（第 **错误!未找到引用源。** 节）：
  - 如果 *a* 未实现 ICriticalNotifyCompletion，则将计算表达式 ((*a*) as INotifyCompletion).OnCompleted(*r*)。
  - 如果 *a* 实现了 ICriticalNotifyCompletion，则将计算表达式 ((*a*) as ICriticalNotifyCompletion).UnsafeOnCompleted(*r*)。
  - 然后将挂起计算，并将控制返回给异步函数的当前调用方。
- 随后立即（如果 *b* 为 true）或在以后调用恢复委托时（如果 *b* 为 false）对表达式 (*a*).GetResult() 进行计算。如果它返回值，则该值是 *await 表达式*的结果。否则，结果为 Nothing。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

接口方法 INotifyCompletion.OnCompleted 和 ICriticalNotifyCompletion.UnsafeOnCompleted 的 awaiter 实现应导致最多调用一次委托 *r*。否则，封闭式异步函数的行为将不确定。

7.8 算术运算符

*、/、%、+ 和 – 运算符称为算术运算符。

multiplicative-expression:
unary-expression
multiplicative-expression * *unary-expression*
multiplicative-expression / *unary-expression*
multiplicative-expression % *unary-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression – *multiplicative-expression*

如果算术运算符的某个操作数具有编译时类型 dynamic，则表达式是动态绑定的（第 7.2.2 节）。在此情况下，表达式的编译时类型为 dynamic，并且会在运行时使用具有编译时类型 dynamic 的操作数的运行时类型进行下面所述的决策。

7.8.1 乘法运算符

对于 x * y 形式的运算，应用二元运算符重载决策（第 7.3.4 节）以选择特定的运算符实现。操作数转换为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。

下面列出了预定义的乘法运算符。这些运算符均计算 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 上下文中，如果积超出结果类型的范围，则引发 System.OverflowException。在 unchecked 上下文中，不报告溢出并且结果类型范围外的任何有效高序位都被放弃。

- 浮点乘法：

```
float operator *(float x, float y);
double operator *(double x, double y);
```

  根据 IEEE 754 算术运算法则计算乘积。下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在该表中，x 和 y 是正有限值，z 是 x * y 的结果。如果结果对目标类型而言太大，则 z 为无穷大。如果结果对目标类型而言太小，则 z 为零。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

	+y	–y	+0	–0	+∞	–∞	NaN
+x	+z	–z	+0	–0	+∞	–∞	NaN
–x	–z	+z	–0	+0	–∞	+∞	NaN
+0	+0	–0	+0	–0	NaN	NaN	NaN
–0	–0	+0	–0	+0	NaN	NaN	NaN
+∞	+∞	–∞	NaN	NaN	+∞	–∞	NaN
–∞	–∞	+∞	NaN	NaN	–∞	+∞	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，不能用 decimal 格式表示，则将引发 System.OverflowException。如果结果值太小，无法用 decimal 格式表示，则结果为零。在进行任何舍入之前，结果的小数位数是两个操作数的小数位数的和。

小数乘法等效于使用 System.Decimal 类型的乘法运算符。

7.8.2 除法运算符

对于 x / y 形式的运算，应用二元运算符重载决策（第 7.3.4 节）以选择特定的运算符实现。操作数转换为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。

下面列出了预定义的除法运算符。这些运算符均计算 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，则引发 System.DivideByZeroException 导常。

除法将结果舍入到零。因此，结果的绝对值是小于或等于两个操作数的商的绝对值的最大可能整数。当两个操作数符号相同时，结果为零或正；当两个操作数符号相反时，结果为零或负。

如果左操作数为最小可表示 int 或 long 值，右操作数为 –1，则发生溢出。在 checked 上下文中，这会导致引发 System.ArithmeticException（或其子类）。在 unchecked 上下文中，它由实现定义为或者引发 System.ArithmeticException（或其子类），或者不以左操作数的结果值报告溢出。

- 浮点除法：

```
float operator /(float x, float y);
double operator /(double x, double y);
```

根据 IEEE 754 算法法则计算商。下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在该表中，x 和 y 是正有限值，z 是 x / y 的结果。如果结果对目标类型而言太大，则 z 为无穷大。如果结果对目标类型而言太小，则 z 为零。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

	+y	–y	+0	–0	+∞	–∞	NaN
+x	+z	–z	+∞	–∞	+0	–0	NaN
–x	–z	+z	–∞	+∞	–0	+0	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

- 小数除法：

```
decimal operator /(decimal x, decimal y);
```

如果右操作数的值为零，则引发 System.DivideByZeroException 导常。如果结果值太大，不能用 decimal 格式表示，则将引发 System.OverflowException。如果结果值太小，无法用 decimal 格式表示，则结果为零。结果的小数位数是最小的小数位数，它保留等于最接近真实算术结果的可表示小数值的结果。

小数除法等效于使用 System.Decimal 类型的除法运算符。

7.8.3 余数运算符

对于 x % y 形式的运算，应用二元运算符重载决策（第 7.3.4 节）以选择特定的运算符实现。操作数转换为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。

下面列出了预定义的余数运算符。这些运算符均计算 x 除以 y 的余数。

- 整数余数：

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

x % y 的结果是由 x – (x / y) * y 生成的值。如果 y 为零，则将引发 System.DivideByZeroException。

如果左侧的操作数是最小的 int 或 long 值，且右侧的操作数是 -1，则将引发 System.OverflowException。只要 x % y 不引发异常，x / y 也不会引发异常。

- 浮点余数：

```
float operator %(float x, float y);
double operator %(double x, double y);
```

下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在该表中，x 和 y 是有限的正值。z 是 x % y 的结果，按照 x – n * y 进行计算，其中 n 是小于或等于 x / y 的最大可能整数。这种计算余数的方法类似于用于整数操作数的方法，但不同于 IEEE 754 定义（在此定义中，n 是最接近 x / y 的整数）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

	+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，则引发 System.DivideByZeroException 导常。在进行任何舍入之前，结果的小数位数是两个操作数中较大的小数位数，而且结果的符号与 x 的相同（如果非零）。

小数余数等效于使用 System.Decimal 类型的余数运算符。

7.8.4 加法运算符

对于 x + y 形式的运算，应用二元运算符重载决策（第 7.3.4 节）以选择特定的运算符实现。操作数转换为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。

下面列出了预定义的加法运算符。对于数值和枚举类型，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为 string 类型时，预定义的加法运算符把两个操作数的字符串表示形式串联起来。

- 整数加法：

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

在 checked 上下文中，如果和超出结果类型的范围，则引发 System.OverflowException。在 unchecked 上下文中，不报告溢出并且结果类型范围外的任何有效高序位都被放弃。

- 浮点加法：

```
float operator +(float x, float y);
double operator +(double x, double y);
```

根据 IEEE 754 算术运算法则计算和。下表列出了非零有限值、零、无穷大和 NaN 的所有可能组合的结果。在该表中，x 和 y 是非零有限值，z 是 x + y 的结果。如果 x 和 y 的绝对值相同但符号相反，则 zz 为正零。如果 x + y 太大，不能用目标类型表示，则 z 是与 x + y 具有相同符号的无穷大。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

	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，不能用 decimal 格式表示，则将引发 System.OverflowException。在进行任何舍入之前，结果的小数位数是两个操作数中较大的小数位数。

  小数加法等效于使用 System.Decimal 类型的加法运算符。

- 枚举加法。每个枚举类型都隐式提供下列预定义运算符，其中 E 为枚举类型，U 为 E 的基础类型：

  ```
  E operator +(E x, U y);
  E operator +(U x, E y);
  ```

  在运行时，这些运算符完全按 (E)((U)x + (U)y) 计算。

- 字符串串联：

  ```
  string operator +(string x, string y);
  string operator +(string x, object y);
  string operator +(object x, string y);
  ```

  这些二元 + 运算符的重载执行字符串串连。在字符串串联运算中，如果它的一个操作数为 null，则用空字符串来替换此操作数。否则，任何非字符串参数都通过调用从 object 类型继承的虚 ToString 方法，转换为它的字符串表示形式。如果 ToString 返回 null，则将替换成空字符串。

  ```
  using System;

  class Test
  {
     static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");     // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);            // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);            // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);            // displays d = 2.900
     }
  }
  ```

  字符串串联运算符的结果是一个字符串，由左操作数的字符后接右操作数的字符组成。字符串串联运算符从不返回 null 值。如果没有足够的内存可用于分配得到的字符串，则可能引发 System.OutOfMemoryException。

- 委托组合。每个委托类型都隐式提供以下预定义运算符，其中 D 是委托类型：

  ```
  D operator +(D x, D y);
  ```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

当两个操作数均为某个委托类型D时，二元 + 运算符执行委托组合。（如果操作数具有不同的委托类型，则发生绑定时错误。）如果第一个操作数为null，则运算结果为第二个操作数的值（即使此操作数也为null）。否则，如果第二个操作数为null，则运算结果为第一个操作数的值。否则，运算结果是一个新委托实例，该实例在被调用时调用第一个操作数，然后调用第二个操作数。有关委托组合的示例，请参见第 7.8.5 节和第 15.4 节。由于 System.Delegate 不是委托类型，因此不为它定义 operator +。

7.8.5 减法运算符

对于 x – y 形式的运算，应用二元运算符重载决策（第7.3.4 节）以选择特定的运算符实现。操作数转换为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。

下面列出了预定义的减法运算符。这些运算符均从 x 中减去 y。

- 整数减法：

```
int operator –(int x, int y);
uint operator –(uint x, uint y);
long operator –(long x, long y);
ulong operator –(ulong x, ulong y);
```

在 checked 上下文中，如果差超出结果类型的范围，则引发System.OverflowException。在 unchecked 上下文中，不报告溢出并且结果类型范围外的任何有效高序位都被放弃。

- 浮点减法：

```
float operator –(float x, float y);
double operator –(double x, double y);
```

根据 IEEE 754 算术运算法则计算差。下表列出了非零有限值、零、无穷大和NaN 的所有可能组合的结果。在该表中，x 和 y 是非零有限值，z 是 x – y 的结果。如果 x 和 y 相等，则 z 为正零。如果 x – y 太大，不能用目标类型表示，则z 是与 x – y 具有相同符号的无穷大。

	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，不能用decimal 格式表示，则将引发 System.OverflowException。在进行任何舍入之前，结果的小数位数是两个操作数中较大的小数位数。

小数减法等效于使用 System.Decimal 类型的减法运算符。

- 枚举减法。每个枚举类型都隐式提供下列预定义运算符，其中E 为枚举类型，U 为 E 的基础类型：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
U operator –(E x, E y);
```

此运算符严格按 (U)((U)x – (U)y) 计算。换言之，运算符计算 x 和 y 的序数值之间的差，结果类型是枚举的基础类型。

```
E operator –(E x, U y);
```

此运算符严格按 (E)((U)x – y) 计算。换言之，该运算符从枚举的基础类型中减去一个值，得到枚举的值。

- 委托移除。每个委托类型都隐式提供以下预定义运算符，其中 D 是委托类型：

```
D operator –(D x, D y);
```

当两个操作数均为某个委托类型 D 时，二元 – 运算符执行委托移除。如果操作数具有不同的委托类型，则发生绑定时错误。如果第一个操作数为 null，则运算结果为 null。否则，如果第二个操作数为 null，则运算结果为第一个操作数的值。否则，两个操作数都表示包含一项或多项的调用列表（第 15.1 节），并且只要第二个操作数列表是第一个操作数列表的适当的邻接子列表，那么结果就是从第一个操作数的调用列表中移除了第二个操作数的调用列表所含各项后的一个新调用列表。（为确定子列表是否相等，用委托相等运算符（第 7.10.8 节）比较相对应的项。）否则，结果为左操作数的值。在此过程中两个操作数的列表均未被更改。如果第二个操作数的列表与第一个操作数的列表中的多个邻接项子列表相匹配，则移除最右边的那个匹配邻接项的子列表。如果移除导致空列表，则结果为 null。例如：

```
delegate void D(int x);

class C
{
   public static void M1(int i) { /* … */ }
   public static void M2(int i) { /* … */ }
}

class Test
{
   static void Main() {
      D cd1 = new D(C.M1);
      D cd2 = new D(C.M2);
      D cd3 = cd1 + cd2 + cd2 + cd1;   // M1 + M2 + M2 + M1
      cd3 -= cd1;                      // => M1 + M2 + M2

      cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
      cd3 -= cd1 + cd2;                // => M2 + M1

      cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
      cd3 -= cd2 + cd2;                // => M1 + M1

      cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
      cd3 -= cd2 + cd1;                // => M1 + M2

      cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
      cd3 -= cd1 + cd1;                // => M1 + M2 + M2 + M1
   }
}
```

7.9 移位运算符

<< 和 >> 运算符用于执行移位运算。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression right-shift additive-expression

如果 *shift-expression* 的某个操作数具有编译时类型 dynamic，则表达式是动态绑定的（第 7.2.2 节）。在此情况下，表达式的编译时类型为 dynamic，并且会在运行时使用具有编译时类型 dynamic 的操作数的运行时类型进行下面所述的决策。

对于 x << count 或 x >> count 形式的运算，应用二元运算符重载决策（第 7.3.4 节）以选择特定的运算符实现。操作数转换为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。

当声明重载移位运算符时，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，并且第二个操作数的类型必须总是 int。

下面列出了预定义的移位运算符。

- 左移位：

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

  << 运算符将 x 向左位移若干个位，具体计算方法如下所述。

  放弃 x 中经移位后会超出结果类型范围的那些高序位，将其余的位向左位移，将空出来的低序位均设置为零。

- 右移位：

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

  >> 运算符将 x 向右位移若干个位，具体计算方法如下所述。

  当 x 为 int 或 long 类型时，放弃 x 的低序位，将剩余的位向右位移，如果 x 非负，则将高序空位位置设置为零，如果 x 为负，则将其设置为 1。

  当 x 为 uint 或 ulong 类型时，放弃 x 的低序位，将剩余的位向右位移，并将高序空位位置设置为零。

对于预定义运算符，位移的位数按下面这样计算：

- 当 x 的类型为 int 或 uint 时，位移计数由 count 的低序的 5 位给出。换言之，位移计数由 count & 0x1F 计算出。
- 当 x 的类型为 long 或 ulong 时，位移计数由 count 的低序的 6 位给出。换言之，位移计数由 count & 0x3F 计算出。

如果计算位移计数的结果为零，则移位运算符只返回 x 的值。

移位运算从不会导致溢出，并且在 checked 和 unchecked 上下文中产生的结果相同。

当 >> 运算符的左操作数为有符号的整型时，该运算符执行算术右移位，在此过程中，操作数的最有效位（符号位）的值扩展到高序空位位置。当 >> 运算符的左操作数为无符号的整型时，该运算符执行逻辑右移位，在此过程中，高序空位位置总是设置为零。若要执行与由操作数类型确定的不同的移位运算，

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

可以使用显式强制转换。例如，如果 x 是 int 类型的变量，则 unchecked((int)((uint)x >> y)) 运算执行 x 的逻辑右移位。

7.10 关系和类型测试运算符

==、!=、<、>、<=、>=、is 和 as 运算符称为关系和类型测试运算符。

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*
relational-expression is *type*
relational-expression as *type*

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

is 和 as 运算符分别在第 7.10.10 节和第 7.10.11 节中说明。

==、!=、<、>、<= 和 >= 运算符为比较运算符 (comparison operator)。

如果比较运算符的某个操作数为编译时类型 dynamic，则表达式是动态绑定的（第 7.2.2 节）。在此情况下，表达式的编译时类型为 dynamic，并且会在运行时使用具有编译时类型 dynamic 的操作数的运行时类型进行下面所述的决策。

对于 x *op* y 形式（其中 *op* 为比较运算符）的运算，应用重载决策（第 7.3.4 节）以选择特定的运算符实现。操作数转换为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详见下面各节的介绍。所有预定义的比较运算符都返回 bool 类型的结果，详见下表。

运算	结果
x == y	如果 x 等于 y，则为 true，否则为 false
x != y	如果 x 不等于 y，则为 true，否则为 false
x < y	如果 x 小于 y，则为 true，否则为 false
x > y	如果 x 大于 y，则为 true，否则为 false
x <= y	如果 x 小于或等于 y，则为 true，否则为 false
x >= y	如果 x 大于或等于 y，则为 true，否则为 false

7.10.1 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);
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
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);
```

这些运算符都比较两个整数操作数的数值并返回一个bool 值，该值指示特定的关系是 true 还是 false。

7.10.2 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);
```

这些运算符根据 IEEE 754 标准法则比较操作数：

- 如果两个操作数中的任何一个为NaNN，则对于除 !=（对于此运算符，结果为 true）外的所有运算符，结果均为 false。对于任何两个操作数，x != y 始终生成与 !(x == y) 相同的结果。然而，当一个操作数或两个操作数为 NaN 时，<、>、<= 和 >= 运算符不产生与其对应的反向运算符的逻辑否定相同的结果。例如，如果 x 和 y 中的任何一个为 NaN，则 x < y 为 false，而 !(x >= y) 为 true。
- 当两个操作数都不为 NaN 时，这些运算符就按下列顺序来比较两个浮点操作数的值

  −∞ < −max < ... < −min < −0.0 == +0.0 < +min < ... < +max < +∞

  这里的 min 和 max 是可以用给定浮点格式表示的最小和最大正有限值。这样排序的显著特点是：

  - 负零和正零被视为相等。

- 负无穷大被视为小于所有其他值，但等于其他负无穷大。
- 正无穷大被视为大于所有其他值，但等于其他正无穷大。

7.10.3 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);
```

这些运算符都比较两个 decimal 操作数的数值并返回一个 bool 值，该值指示特定的关系是 true 还是 false。各小数比较等效于使用 System.Decimal 类型的相应关系运算符或相等运算符。

7.10.4 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);
```

如果 x 和 y 都为 true，或者 x 和 y 都为 false，则 == 的结果为 true。否则，结果为 false。

如果 x 和 y 都为 true，或者 x 和 y 都为 false，则 != 的结果为 false。否则，结果为 true。当操作数为 bool 类型时，!= 运算符产生与 ^ 运算符相同的结果。

7.10.5 枚举比较运算符

每种枚举类型都隐式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);
```

x *op* y（其中 x 和 y 是具有基础类型 U 的枚举类型 E 的表达式，*op* 是一个比较运算符）的计算结果与 (E)((U)x) *op* ((U)y) 的计算结果完全相同。换言之，枚举类型比较运算符只比较两个操作数的基础整数值。

7.10.6 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);
```

这些运算符返回两个引用是相等还是不相等的比较结果。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由于预定义的引用类型相等运算符接受object类型的操作数，因此它们适用于所有那些没有为自己声明适用的operator == 和 operator != 成员的类型。相反，任何适用的用户定义的相等运算符都有效地隐藏上述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要求满足以下条件之一：

- 两个操作数均为已知的 *reference-type* 类型的值或文本 null。此外，存在从其中一个操作数的类型到另一个操作数的类型的显式引用转换（第 6.2.4 节）。
- 一个操作数是类型为 T 的值，其中 T 为 *type-parameter*，另一个操作数为文本 null。此外，T 不具有值类型约束。

除非满足以下这些条件之一，否则将发生绑定时错误。这些规则中值得注意的含义是：

- 使用预定义的引用类型相等运算符比较两个在绑定时已能确定是不相同的引用时，会导致绑定时错误。例如，如果操作数的绑定时类型是两种类类型 A 和 B，并且如果 A 和 B 都不是从对方派生的，则两个操作数不可能引用同一对象。因此，此运算被认为是绑定时错误。
- 预定义的引用类型相等运算符不允许比较值类型操作数。因此，除非结构类型声明自己的相等运算符，否则不可能比较该结构类型的值。
- 预定义的引用类型相等运算符从不会导致对它们的操作数执行装箱操作。执行此类装箱操作毫无意义，这是因为对新分配的已装箱实例的引用必将不同于所有其他引用。
- 如果将类型参数类型 T 的操作数与 null 进行比较，并且T 的运行时类型为值类型，则比较结果为 false。

下面的示例检查未受约束的类型形参类型的实参是否为 null。

```
class C<T>
{
   void F(T x) {
      if (x == null) throw new ArgumentNullException();
      ...
   }
}
```

虽然 T 可能表示值类型，但是 x == null 构造是允许的，当 T 为值类型时，结果只是被定义为 false。

对于 x == y 或 x != y 形式的运算，如果存在任何适用的 operator == 或 operator !=，则运算符重载决策（第 7.3.4 节）规则将选择该运算符而不是上述的预定义的引用类型相等运算符。不过，始终可以通过将一个或两个操作数显式强制转换为 object 类型来选择预定义的引用类型相等运算符。下面的示例

```
using System;

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

产生输出

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
True
False
False
False
```

变量 s 和 t 引用两个包含相同字符的不同 string 实例。第一个比较输出 True，原因是当两个操作数都为 string 类型时选择了预定义的字符串相等运算符（第 7.10.7 节）。其余的比较全都输出 False，这是因为是在一个或两个操作数为 object 类型时选定预定义的引用类型相等运算符。

注意，以上技术对值类型没有意义。下面的示例

```
class Test
{
   static void Main() {
      int i = 123;
      int j = 123;
      System.Console.WriteLine((object)i == (object)j);
   }
}
```

输出 False，这是因为强制转换创建对已装箱 int 值的两个单独实例的引用。

7.10.7 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);
```

当下列条件中有一个为真时，两个 string 值被视为相等：

- 两个值都为 null。
- 两个值都是对字符串实例的非空引用，这两个字符串不仅具有相同的长度，而且在每个字符位置上的字符亦都彼此相同。

字符串相等运算符比较的是字符串值而不是对字符串的引用。当两个单独的字符串实例包含完全相同的字符序列时，字符串的值相等，但引用不相同。正如第 7.10.6 节中所描述的那样，引用类型相等运算符可用于比较字符串引用而不是字符串值。

7.10.8 委托相等运算符

预定义的委托相等运算符为：

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

两个委托实例按下面这样被视为相等：

- 如果两个委托实例中有一个为 null，则当且仅当它们都为 null 时相等。
- 具有不同运行时类型的委托永远不相等。
- 如果两个委托实例都具有调用列表（第 15.1) 节），则当且仅当它们的调用列表长度相同，并且一个实例的调用列表中的每项依次等于（如下面的定义）另一个的调用列表中的相应项时，这两个委托实例相等。

以下规则控制调用列表项的相等性：

- 如果两个调用列表项都引用同一静态方法，则这两项相等。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果两个调用列表项都引用同一个目标对象（引用相等运算符定义的目标对象）上的同一个非静态方法，则这两个调用列表项相等。
- 允许（但不要求）具有相同被捕获外层变量实例集（可能为空集）且语义上相同的 *anonymous-function-expression* 计算生成的调用列表项相等。

7.10.9 相等运算符和 null

== 和 != 运算符允许一个操作数是可为 null 的类型的值，另一个是 null 文本，即使运算中不存在预定义或用户定义的运算符（未提升或提升形式）。

对于下面某个形式的操作

```
x == null    null == x    x != null    null != x
```

其中 x 是可为 null 的类型的表达式，如果运算符重载决策（第 7.2.4 节）未能找到适用的运算符，则改为从 x 的 HasValue 属性计算结果。具体而言，前两种形式将转换为 !x.HasValue，后两种形式将转换为 x.HasValue。

7.10.10 is 运算符

is 运算符用于动态检查对象的运行时类型是否与给定类型兼容。E is T 运算（其中 E 为表达式，T 为类型）的结果是布尔值，表示 E 的类型是否可通过引用转换、装箱转换或取消装箱转换而成功转换为类型 T。使用类型实参替换了所有类型形参后，按如下方式计算该运算：

- 如果 E 是匿名函数，将发生编译时错误
- 如果 E 是方法组或 null 文本，或者如果 E 的类型是引用类型或可为 null 的类型并且 E 的值为 null，则结果为 false。
- 否则，根据下列规则让 D 表示 E 的动态类型：
  - 如果 E 的类型为引用类型，则 D 为 E 引用的实例的运行时类型。
  - 如果 E 的类型为可以为 null 的类型，则 D 为该可以为 null 的类型的基础类型。
  - 如果 E 的类型为不可以为 null 值的类型，则 D 为 E 的类型。
- 该操作的结果取决于 D 和 T，具体如下：
  - 如果 T 为引用类型，那么，在以下情况下结果为 true：D 和 T 为相同类型，或者 D 为引用类型并且存在从 D 到 T 的隐式引用转换，或者 D 为值类型并且存在从 D 到 T 的装箱转换。
  - 如果 T 为可以为 null 的类型，那么，当 D 为 T 的基础类型时结果为 true。
  - 如果 T 为不可以为 null 值的类型，那么，如果 D 和 T 为相同类型，则结果为 true。
  - 否则，结果为 false。

请注意，用户定义的转换不在 is 运算符考虑之列。

7.10.11 as 运算符

as 运算符用于将一个值显式转换为一个给定的引用类型或可为null 的类型。与强制转换表达式（第 7.7.6 节）不同，as 运算符从不引发异常。它采用的是：如果指定的转换不可能实施，则运算结果为 null。

在 E as T 形式的操作中，E 必须为表达式，T 必须为引用类型、已知为引用类型的类型参数或可以为 null 的类型。此外，下列条件中必须至少有一条成立，否则会发生编译时错误：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 存在从 E 到 T 的以下类型转换：标识（第 6.1.1 节）、隐式可以为 null（第 6.1.4 节）、隐式引用（第 6.1.6 节）、装箱（第 6.1.7 节）、显式可以为 null（第 6.2.3 节）、显式引用（第 6.2.4 节）或取消装箱（第 6.2.5 节）转换。
- E 或 T 的类型为开放类型。
- E 为 null 文本。

如果 E 的编译时类型不是 dynamic，则运算 E as T 将生成与下面的计算相同的结果

```
E is T ? (T)(E) : (T)null
```

不同的只是：实际执行中 E 只计算一次。编译器应该优化 E as T 以最多执行一次动态类型检查，而不是上面的扩展隐含的两次动态类型检查。

如果 E 的编译时类型为 dynamic，则与强制转换运算符不同，as 运算符不是动态绑定的（第 7.2.2 节）。因此这种情况下的扩展为：

```
E is T ? (T)(object)(E) : (T)null
```

请注意，不能使用 as 运算符执行某些转换（如用户定义的转换），应改为使用强制转换表达式来执行这些转换。

在下面的示例中

```
class X
{
   public string F(object o) {
      return o as string;     // OK, string is a reference type
   }

   public T G<T>(object o) where T: Attribute {
      return o as T;          // Ok, T has a class constraint
   }

   public U H<U>(object o) {
      return o as U;          // Error, U is unconstrained
   }
}
```

G 的类型参数 T 已知为引用类型，原因是它有类约束。但 H 的类型参数 U 不是；因此，不允许在 H 中使用 as 运算符。

7.11 逻辑运算符

&、^ 和 | 运算符称为逻辑运算符。

and-expression:
equality-expression
and-expression **&** *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression **^** *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression **|** *exclusive-or-expression*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

如果逻辑运算符的某个操作数具有编译时类型dynamic，则表达式是动态绑定的（第 7.2.2 节）。在此情况下，表达式的编译时类型为dynamic，并且会在运行时使用具有编译时类型 dynamic 的操作数的运行时类型进行下面所述的决策。

对于 x *op* y 形式的运算（其中 *op* 为一个逻辑运算符），应用重载决策（第 7.3.4 节）以选择一个特定的运算符实现。操作数转换为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。

下列章节介绍了预定义的逻辑运算符。

7.11.1 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 AND，| 运算符计算两个操作数的按位逻辑 OR，而 ^ 运算符计算两个操作数的按位逻辑 XOR。这些运算不可能产生溢出。

7.11.2 枚举逻辑运算符

每个枚举类型 E 都隐式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);
```

x *op* y（其中 x 和 y 是具有基础类型 U 的枚举类型 E 的表达式，*op* 是一个逻辑运算符）的计算结果与 (E)((U)x *op* (U)y) 的计算结果完全相同。换言之，枚举类型逻辑运算符直接对两个操作数的基础类型执行逻辑运算。

7.11.3 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);
```

如果 x 和 y 均为 true，则 x & y 的结果为 true。否则，结果为 false。

如果 x 或 y 为 true，则 x | y 的结果为 true。否则，结果为 false。

如果 x 为 true 且 y 为 false，或者 x 为 false 且 y 为 true，则 x ^ y 的结果为 true。否则，结果为 false。当操作数为 bool 类型时，^ 运算符计算结果与 != 运算符相同。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.11.4 可以为 null 的布尔逻辑运算符

可以为 null 的布尔类型 bool? 可表示三个值 true、false 和 null，并且在概念上类似于 SQL 中的布尔表达式的三值类型。为了确保针对 bool? 操作数的 & 和 | 运算符产生的结果与 SQL 的三值逻辑一致，提供了下列预定义运算符：

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

下表列出了这些运算符对 true、false 和 null 值的所有组合所产生的结果。

x	y	x & y	x | y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

7.12 条件逻辑运算符

&& 和 || 运算符称为条件逻辑运算符。也称为“短路”逻辑运算符。

conditional-and-expression:
inclusive-or-expression
conditional-and-expression **&&** *inclusive-or-expression*

conditional-or-expression:
conditional-and-expression
conditional-or-expression **||** *conditional-and-expression*

&& 和 || 运算符是 & 和 | 运算符的条件版本：

- x && y 运算对应于 x & y 运算，但仅当 x 不为 false 时才计算 y。
- x || y 运算对应于 x | y 运算，但仅当 x 不为 true 时才计算 y。

如果条件逻辑运算符的某个操作数具有编译时类型 dynamic，则表达式是动态绑定的（第 7.2.2 节）。在此情况下，表达式的编译时类型为 dynamic，并且会在运行时使用具有编译时类型 dynamic 的操作数的运行时类型进行下面所述的决策。

x && y 或 x || y 形式的运算通过应用重载决策（第 7.3.4 节）来处理，就好比运算的书写形式为 x & y 或 x | y。然后，

- 如果重载决策未能找到单个最佳运算符，或者重载决策选择一个预定义的整数逻辑运算符，则发生绑定时错误。
- 否则，如果选定的运算符是一个预定义的布尔逻辑运算符（第 7.11.3 节）或可以为 null 的布尔逻辑运算符（第 7.11.4 节），则运算按第 7.12.1 节中所描述的那样进行处理。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，选定的运算符为用户定义的运算符，且运算按第 7.12.2 节中所描述的那样进行处理。

不可能直接重载条件逻辑运算符。不过，由于条件逻辑运算符按通常的逻辑运算符计算，因此通常的逻辑运算符的重载，在某些限制条件下，也被视为条件逻辑运算符的重载。第 7.12.2 节对此有进一步描述。

7.12.1 布尔条件逻辑运算符

当 && 或 || 的操作数为 bool 类型时，或者当操作数的类型本身未定义适用的 operator & 或 operator |，但确实定义了到 bool 的隐式转换时，运算按下面这样处理：

- 运算 x && y 的求值过程相当于 x ? y : false。换言之，首先计算 x 并将其转换为 bool 类型。如果 x 为 true，则计算 y 并将其转换为 bool 类型，并且这成为运算结果。否则，运算结果为 false。
- 运算 x || y 的求值过程相当于 x ? true : y。换言之，首先计算 x 并将其转换为 bool 类型。然后，如果 x 为 true，则运算结果为 true。否则，计算 y 并将其转换为 bool 类型，并且这作为运算结果。

7.12.2 用户定义的条件逻辑运算符

当 && 或 || 的操作数所属的类型声明了适用的用户定义的 operator & 或 operator | 时，下列两个条件必须都为真（其中 T 是声明的选定运算符的类型）：

- 选定运算符的返回类型和每个参数的类型都必须为 T。换言之，该运算符必须计算类型为 T 的两个操作数的逻辑 AND 或逻辑 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。

如果这两个要求中有一个未满足，则发生绑定时错误。如果这两个要求都满足，则通过将用户定义的 operator true 或 operator false 与选定的用户定义的运算符组合在一起来计算 && 运算或 || 运算：

- x && y 运算按 T.false(x) ? x : T.&(x, y) 进行计算，其中 T.false(x) 是 T 中声明的 operator false 的调用，T.&(x, y) 是选定 operator & 的调用。换言之，首先计算 x，然后对结果调用 operator false 以确定 x 是否肯定为 false。如果 x 肯定为假，则运算结果为先前为 x 计算的值。否则将计算 y，并对先前为 x 计算的值和为 y 计算的值调用选定的 operator & 以产生运算结果。
- x || y 运算按 T.true(x) ? x : T.|(x, y) 进行计算，其中 T.true(x) 是 T 中声明的 operator true 的调用，T.|(x, y) 是选定 operator | 的调用。换言之，首先计算 x，然后对结果调用 operator true 以确定 x 是否肯定为 true。然后，如果 x 肯定为真，则运算结果为先前为 x 计算的值。否则将计算 y，并对先前为 x 计算的值和为 y 计算的值调用选定的 operator | 以产生运算结果。

在这两个运算中，x 给定的表达式只计算一次，y 给定的表达式要么不计算，要么只计算一次。

有关实现了 operator true 和 operator false 的类型的示例，请参见第 11.4.2 节。

7.13 空合并运算符

?? 运算符称为空合并运算符。

null-coalescing-expression:
conditional-or-expression
conditional-or-expression ?? *null-coalescing-expression*

a ?? b 形式的空合并表达式要求 a 为可以为 null 的类型或引用类型。如果 a 为非 null，则 a ?? b 的结果为 a；否则，结果为 b。仅当 a 为 null 时，该操作才计算 b。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

空合并运算符为右结合运算符，表示操作从右向左进行组合。例如，a ?? b ?? c 形式的表达式可以按 a ?? (b ?? c) 进行计算。概括地说，E_1 ?? E_2 ?? ... ?? E_N 形式的表达式返回第一个非 null 的操作数，如果所有操作数都为 null，则返回 null。

表达式 a ?? b 的类型取决于对操作数可用的隐式转换。按照优先顺序，a ?? b 的类型为 A_0、A 或 B，其中 A 是 a 的类型（如果 a 有类型），B 是 b 的类型（如果 b 有类型），A_0 是 A 的基础类型（如果 A 是可以为 null 的类型）或 A（如果该项不是可以为 null 的类型）。具体而言，a ?? b 的处理过程如下：

- 如果 A 存在并且不是可以为 null 的类型或引用类型，将发生编译时错误。
- 如果 b 是动态表达式，则结果类型为 dynamic。在运行时，首先计算 a。如果 a 不为 null，则 aa 转换为动态类型，这成为结果。否则，计算 b，这成为结果。
- 否则，如果 A 存在并且是可以为 null 的类型，并且存在从 b 到 A_0 的隐式转换，则结果类型为 A_0。在运行时，首先计算 a。如果 a 不为 null，则 a 解包为类型 A_0，这即是结果。否则，计算 b 并转换为类型 A_0，这即是结果。
- 否则，如果 A 存在并且存在从 b 到 A 的隐式转换，则结果类型为 A。在运行时，首先计算 a。如果 a 不为 null，则 a 即是结果。否则，计算 b 并转换为类型 A，这即是结果。
- 否则，如果 A 存在并且是可为 null 的类型，b 具有类型 B 并且存在从 A_0 到 B 的隐式转换，则结果类型为 B。在运行时，首先计算 a。如果 a 不为 null，则 a 将解包为类型 A_0 并且转换为类型 B，这即是结果。否则，计算 b 并且 b 作为结果。
- 否则，如果 b 的类型为 B，并且存在从 a 到 B 的隐式转换，则结果类型为 B。在运行时，首先计算 a。如果 a 不为 null，则 a 将转换为类型 B，这即是结果。否则，计算 b 并且 b 作为结果。
- 否则，a 和 b 不兼容，并发生编译时错误。

7.14 条件运算符

?: 运算符称为条件运算符。有时，它也称为三元运算符。

conditional-expression:
null-coalescing-expression
null-coalescing-expression ? *expression* : *expression*

b ? x : y 形式的条件表达式首先计算条件 b。然后，如果 b 为 true，则将计算 x，并且它将成为运算结果。否则计算 y，并且它成为运算结果。条件表达式从不同时计算 x 和 y。

条件运算符向右关联，表示运算从右到左分组。例如，a ? b : c ? d : e 形式的表达式可以按 a ? b : (c ? d : e) 进行计算。

?: 运算符的第一个操作数必须是可以隐式转换为 bool 的表达式，或是实现 operator true 的类型的表达式。如果两个要求都不满足，则发生编译时错误。

?: 运算符的第二和第三个操作数 x 和 y 控制条件表达式的类型。

- 如果 x 具有类型 X 且 y 具有类型 Y，则
  - 如果存在从 X 到 Y 的隐式转换（第 0 节），但不存在从 Y 到 X 的隐式转换，则 Y 为条件表达式的类型。
  - 如果存在从 Y 到 X 的隐式转换（第 0 节），但不存在从 X 到 Y 的隐式转换，则 X 为条件表达式的类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，无法确定条件表达式的类型，会发生编译时错误。

- 如果 x 和 y 中只有一个具有类型，并且 x 和 y 都可隐式转换为该类型，则该类型为条件表达式的类型。
- 否则，无法确定条件表达式的类型，会发生编译时错误。

b ? x : y 形式的条件表达式的运行时处理包括以下步骤：

- 首先，计算 b，并确定 b 的 bool 值：
  - 如果存在从 b 的类型到 bool 的隐式转换，则执行该隐式转换以产生 bool 值。
  - 否则，将调用由 b 的类型定义的 operator true 以生成 bool 值。
- 如果以上步骤产生的 bool 值为 true，则计算 x 并将其转换为条件表达式的类型，且这成为条件表达式的结果。
- 否则，计算 y 并将其转换为条件表达式的类型，且这成为条件表达式的结果。

7.15 匿名函数表达式

匿名函数 (anonymous function) 是表示“内联”方法定义的表达式。匿名函数本身及其内部没有值或类型，但可以转换为兼容委托或表达式树类型。匿名函数转换的计算取决于转换的目标类型：如果是委托类型，则转换计算为引用匿名函数所定义的方法的委托值。如果目标类型为表达式目录树类型，则转换将计算以对象结构形式表示方法结构的表达式目录树。

由于历史原因，有两种匿名函数句法风格，即 *lambda-expression* 和 *anonymous-method-expression*。对于几乎所有用途，*lambda-expression* 都比 *anonymous-method-expression* 更为简洁且更具表现力，但语言中仍保留后者以便向后兼容。

lambda-expression:
async$_{opt}$ *anonymous-function-signature* => *anonymous-function-body*

anonymous-method-expression:
async$_{opt}$ delegate *explicit-anonymous-function-signature*$_{opt}$ *block*

anonymous-function-signature:
explicit-anonymous-function-signature
implicit-anonymous-function-signature

explicit-anonymous-function-signature:
(*explicit-anonymous-function-parameter-list*$_{opt}$)

explicit-anonymous-function-parameter-list:
explicit-anonymous-function-parameter
explicit-anonymous-function-parameter-list , *explicit-anonymous-function-parameter*

explicit-anonymous-function-parameter:
anonymous-function-parameter-modifier$_{opt}$ *type* *identifier*

anonymous-function-parameter-modifier:
ref
out

implicit-anonymous-function-signature:
(*implicit-anonymous-function-parameter-list*$_{opt}$)
implicit-anonymous-function-parameter

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

implicit-anonymous-function-parameter-list:
implicit-anonymous-function-parameter
implicit-anonymous-function-parameter-list , *implicit-anonymous-function-parameter*

implicit-anonymous-function-parameter:
identifier

anonymous-function-body:
expression
block

=> 运算符与赋值 (=) 运算符优先级相同，并且向右关联。

具有 async 修饰符的匿名函数是一种异步函数，并遵循第 **错误!未找到引用源。** 节中描述的规则。

lambda-expression 形式的匿名函数的参数可以显式或隐式类型化。在显式类型化参数列表中，每个参数的类型是显式声明的。在隐式类型化参数列表中，参数的类型是从匿名函数出现的上下文中推断的，具体而言，当匿名函数转换为兼容委托类型或表达式目录树类型时，该类型提供参数类型（第 0 节）。

在具有一个隐式类型化参数的匿名函数中，参数列表中可以省略括号。换言之，具有以下形式的匿名函数

(*param*) => *expr*

可以简写为

param => *expr*

anonymous-method-expression 形式的匿名函数的参数列表是可选的。如果提供了参数，则参数必须显式类型化。如果未给出参数，则匿名函数可以转换为带有不含 out 参数的参数列表的委托。

除非匿名函数出现在不可到达的语句内，否则该匿名函数的 *block* 是可到达的（第 8.1 节）。

下面是一些匿名函数示例：

```
x => x + 1                              // Implicitly typed, expression body
x => { return x + 1; }                  // Implicitly typed, statement body
(int x) => x + 1                        // Explicitly typed, expression body
(int x) => { return x + 1; }            // Explicitly typed, statement body
(x, y) => x * y                         // Multiple parameters
() => Console.WriteLine()               // No parameters
async (t1,t2) => await t1 + await t2    // Async
delegate (int x) { return x + 1; }      // Anonymous method expression
delegate { return 1 + 1; }              // Parameter list omitted
```

lambda-expression 和 *anonymous-method-expression* 的行为除以下几点外是相同的：

- *anonymous-method-expression* 允许完全省略参数列表，从而可转换为具有任意值参数列表的委托类型。
- *lambda-expression* 允许省略和推断参数类型，而 *anonymous-method-expression* 要求显式声明参数类型。
- *lambda-expression* 的主体可以为表达式或语句块，而 *anonymous-method-expression* 的主体必须为语句块。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 只有 *lambda-expression* 可具有到兼容的表达式树类型（第4.6 节）的转换。

7.15.1 匿名函数签名

匿名函数的可选 *anonymous-function-signature* 定义匿名参数的形参的名称和类型（可选）。匿名函数的参数的范围是 *anonymous-function-body*。（第 3.7 节）anonymous-method-body 与参数列表（如果提供了）一起构成声明空间（第 3.3 节）。因此，如果匿名函数的某一参数的名称与范围包括 *anonymous-method-expression* 或 *lambda-expression* 的局部变量、局部常量或参数的名称匹配，则产生编译时错误。

如果匿名函数有 *explicit-anonymous-function-signature*，则兼容委托类型和表达式目录树类型的集合限制为具有相同顺序的相同参数类型和修饰符。与方法组转换（第6.6 节）不同，匿名函数参数类型的逆变不受支持。如果匿名函数没有 *anonymous-function-signature*，则兼容的委托类型和表达式目录树类型的集将局限于那些没有 `out` 参数的委托类型和表达式目录树类型。

请注意，*anonymous-function-signature* 不能包含特性或参数数组。然而，*anonymous-function-signature* 可以与其参数列表包含参数数组的委托类型兼容。

另请注意，即使兼容，到表达式目录树类型的转换在编译时仍将失败（第 4.6 节）。

7.15.2 匿名函数体

匿名函数体（*expression* 或 *block*）遵循以下规则：

- 如果匿名函数包含签名，则可以在函数体中使用签名中指定的参数。如果匿名函数没有签名，则它可转换为带有参数的委托类型或表达式类型（第 0 节），但是这些参数在该函数体中不可访问。
- 除了在最近的封闭匿名函数签名（如果存在）中指定的 `ref` 或 `out` 参数外，该函数体访问其他 `ref` 或 `out` 参数将导致编译时错误。
- 当 `this` 的类型为结构类型时，该函数体访问 `this` 将导致编译时错误。无论访问是显式（如 `this.x`）还是隐式（如 `x`，其中 `x` 是该结构的实例成员），此规则都成立。此规则仅仅是禁止此类访问，但是不影响成员查找是否能找到该结构的成员。
- 函数体可以访问匿名函数的外层变量（第 7.15.5 节）。对某个外层变量的访问将引用该变量在计算 *lambda-expression* 或 *anonymous-method-expression*（第 7.15.6 节）时处于活动状态的实例。
- 该函数体包含目标在该函数体之外或该函数体之内所包含的匿名函数的 `goto` 语句、`break` 语句或 `continue` 语句时将导致编译时错误。
- 该函数体中的 `return` 语句从最近的封闭匿名函数调用中返回控制，而不是从封闭函数成员中返回。`return` 语句中指定的表达式必须与最近的封闭 *lambda-expression* 或 *anonymous-method-expression* 所转换为的委托类型或表达式目录树类型兼容（第 0 节）。

未显式指定除计算和调用 *lambda-expression* 或 *anonymous-method-expression* 以外，是否存在执行匿名函数块的任何其他方法。具体而言，编译器可选择通过合成一个或多个命名方法或类型来实现匿名函数。任何此类合成元素的名称都必须是为供编译器使用而保留的形式。

7.15.3 重载决策

参数列表中的匿名函数会参与类型推断和重载决策。有关确切规则，请参考第 7.5.2 节和第 7.5.3 节。

下面的示例演示匿名函数对重载决策的影响。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class ItemList<T>: List<T>
{
   public int Sum(Func<T,int> selector) {
      int sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }

   public double Sum(Func<T,double> selector) {
      double sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }
}
```

ItemList<T> 类具有两个 Sum 方法。每个方法都采用一个 selector 参数，该参数从列表项中提取值进行求和。提取的值可以为 int 或 double 型，得到的和同样为 int 或 double 型。

例如，Sum 方法可用于计算订单中明细行的列表的和。

```
class Detail
{
   public int UnitCount;
   public double UnitPrice;
   ...
}

void ComputeSums() {
   ItemList<Detail> orderDetails = GetOrderDetails(...);
   int totalUnits = orderDetails.Sum(d => d.UnitCount);
   double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
   ...
}
```

在对 orderDetails.Sum 的第一次调用中，两个 Sum 方法均适用，原因是匿名函数 d => d.UnitCount 与 Func<Detail,int> 和 Func<Detail,double> 均兼容。但是，重载决策采用了第一个 Sum 方法，原因是到 Func<Detail,int> 的转换比到 Func<Detail,double> 的转换更有利。

在对 orderDetails.Sum 的第二次调用中，只有第二个 Sum 方法适用，原因是匿名函数 d => d.UnitPrice * d.UnitCount 将生成一个 double 类型的值。因此，重载决策采用第二个 Sum 方法进行该调用。

7.15.4 匿名函数与动态绑定

匿名函数不能是动态绑定操作的接收器、参数或操作数。

7.15.5 外层变量

其范围包含 *lambda-expression* 或 *anonymous-method-expression* 的任何局部变量、值参数或参数数组称为匿名函数的外层变量 (outer variable)。在类的实例函数成员中，this 值被视为值参数，并且是该函数成员内包含的所有匿名函数的外层变量。

7.15.5.1 捕获的外层变量

某个外层变量由某个匿名函数引用时，称该外层变量已被该匿名函数捕获 (captured)。通常，局部变量的生存期仅限于该变量所关联的代码块或语句的执行期间（第 5.1.7 节）。但是，被捕获的外层变量的生存期将至少延长至从匿名函数创建的委托或表达式树可以被垃圾回收为止。

在下面的示例中

```
using System;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
delegate int D();

class Test
{
   static D F() {
      int x = 0;
      D result = () => ++x;
      return result;
   }

   static void Main() {
      D d = F();
      Console.WriteLine(d());
      Console.WriteLine(d());
      Console.WriteLine(d());
   }
}
```

局部变量 x 由匿名函数捕获，并且 x 的生存期至少延长至从 F 返回的委托可以被垃圾回收为止（这要到程序的最后才会发生）。由于对匿名函数的每次调用都对同一个 x 实例进行操作，因此该示例的输出为：

```
1
2
3
```

当局部变量或值参数由匿名函数捕获时，该局部变量或参数不再被视作固定变量（第 18.3 节），而是被视作可移动变量。因此，任何使用被捕获外层变量的地址的 unsafe 代码必须首先使用 fixed 语句固定该变量。

注意，与未捕获的变量不同，捕获的局部变量可以同时公开给多个执行线程。

7.15.5.2 局部变量实例化

当执行过程进入局部变量范围时，该变量视为被实例化 (instantiated)。例如，当调用下面的方法时，局部变量 x 被实例化和初始化三次，每一次对应于循环的一轮迭代。

```
static void F() {
   for (int i = 0; i < 3; i++) {
      int x = i * 2 + 1;
      ...
   }
}
```

但是，如果将 x 声明移到循环之外，则 x 将只实例化一次：

```
static void F() {
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      ...
   }
}
```

未捕获时，我们无法确切知道局部变量实例化的频率，因为实例化的生存期不是连续的，有可能每次实例化都只使用同一存储位置。但是，当匿名函数捕获到局部变量时，实例化的效果就会变得很明显。

下面的示例

```
using System;
delegate void D();
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static D[] F() {
      D[] result = new D[3];
      for (int i = 0; i < 3; i++) {
         int x = i * 2 + 1;
         result[i] = () => { Console.WriteLine(x); };
      }
      return result;
   }

   static void Main() {
      foreach (D d in F()) d();
   }
}
```

产生下列输出：

```
1
3
5
```

但是，当 x 的声明移到循环外时：

```
static D[] F() {
   D[] result = new D[3];
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      result[i] = () => { Console.WriteLine(x); };
   }
   return result;
}
```

输出为：

```
5
5
5
```

如果 for 循环声明了一个迭代变量，则该变量本身将被视为在该循环外部声明。因此，如果将该示例更改为捕获迭代变量本身：

```
static D[] F() {
   D[] result = new D[3];
   for (int i = 0; i < 3; i++) {
      result[i] = () => { Console.WriteLine(i); };
   }
   return result;
}
```

则将仅捕获该迭代变量的一个实例，这会产生以下输出：

```
3
3
3
```

匿名函数委托可共享某些捕获的变量，但是又具有其他变量的不同实例。例如，如果 F 更改为

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
static D[] F() {
   D[] result = new D[3];
   int x = 0;
   for (int i = 0; i < 3; i++) {
      int y = 0;
      result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
   }
   return result;
}
```

三个委托捕获相同的 x 实例，但捕获不同的 y 实例，输出为：

```
1 1
2 1
3 1
```

不同的匿名函数可捕获一个外层变量的同一个实例。在下面的示例中：

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
   static void Main() {
      int x = 0;
      Setter s = (int value) => { x = value; };
      Getter g = () => { return x; };
      s(5);
      Console.WriteLine(g());
      s(10);
      Console.WriteLine(g());
   }
}
```

两个匿名函数捕获局部变量 x 的同一个实例，并且它们因此可通过该变量进行“通信”。该示例的输出为：

```
5
10
```

7.15.6 匿名函数表达式计算

匿名函数 F 必须始终直接或通过执行委托创建表达式 new D(F) 来转换为委托类型 D 或表达式目录树类型 E。此转换将确定匿名函数的结果，如第 0 节所述。

7.16 查询表达式

查询表达式 (query expression) 为查询提供一种类似于关系和分层查询语言（如 SQL 和 XQuery）的语言集成语法。

query-expression:
    from-clause *query-body*

from-clause:
    from *type*$_{opt}$ *identifier* in *expression*

query-body:
    query-body-clauses$_{opt}$ *select-or-group-clause* *query-continuation*$_{opt}$

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

query-body-clauses:
    query-body-clause
    query-body-clauses query-body-clause

query-body-clause:
    from-clause
    let-clause
    where-clause
    join-clause
    join-into-clause
    orderby-clause

let-clause:
    `let` *identifier* `=` *expression*

where-clause:
    `where` *boolean-expression*

join-clause:
    `join` $type_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression*

join-into-clause:
    `join` $type_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression* `into` *identifier*

orderby-clause:
    `orderby` *orderings*

orderings:
    ordering
    orderings `,` *ordering*

ordering:
    expression $ordering\text{-}direction_{opt}$

ordering-direction:
    `ascending`
    `descending`

select-or-group-clause:
    select-clause
    group-clause

select-clause:
    `select` *expression*

group-clause:
    `group` *expression* `by` *expression*

query-continuation:
    `into` *identifier query-body*

查询表达式以 `from` 子句开始，以 `select` 或 `group` 子句结束。初始 `from` 子句后面可以跟零个或者多个 `from`、`let`、`where`、`join` 或 `orderby` 子句。每个 `from` 子句都是一个生成器，该生成器将引入一个包括序列 (sequence) 的元素的范围变量 (range variable)。每个 `let` 子句都会引入一个范围变量，以表示通过前一个范围变量计算的值。每个 `where` 子句都是一个筛选器，用于从结果中排除项。每个 `join` 子句都将指定的源序列键与其他序列的键进行比较，以产生匹配对。每个 `orderby` 子句都会根据指定的条件对各项进行重新排序。而最后的 `select` 或 `group` 子句根据范围变量来指定结果的表现形式。最后，可以使用 `into` 子句来“连接”查询，方法是将某一查询的结果视为后续查询的生成器。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.16.1 查询表达式的多义性

查询表达式包含许多“上下文关键字”，即在给定的上下文中有特殊含义的标识符。具体而言，这些关键字包括：from、where、join、on、equals、into、let、orderby、ascending、descending、select、group 和 by。为避免在查询表达式中将这些标识符作为关键字或简单名称混合使用，从而造成多义性，所以当这些标识符出现在查询表达式内的任何位置时，都将它们视为关键字。

为此，查询表达式是以“from *identifier*”开头后接除“;”、“=”或“,”之外的任何标记的任何表达式。

为了将这些字词用作查询表达式中的标识符，可以为其加上前缀“@”（第 2.4.2 节）。

7.16.2 查询表达式转换

C# 语言不指定查询表达式的执行语义。而是将查询表达式转换为遵循查询表达式模式（第 7.16.3 节）的方法调用。具体而言，查询表达式将转换为对具有以下名称的方法的调用：Where、Select、SelectMany、Join、GroupJoin、OrderBy、OrderByDescending、ThenBy、ThenByDescending、GroupBy 和 Cast。这些方法应该有特定的签名和结果类型，如第 7.16.3 节所述。这些方法可以是所查询对象的实例方法或对象外部的扩展方法，它们实现查询的实际执行。

从查询表达式到方法调用的转换是一种句法映射，在执行任何类型绑定或重载决策之前发生。该转换可以保证在句法上正确，但不能保证生成语法正确的 C# 代码。转换查询表达式后，以常规方法调用的方式处理生成的方法调用，而这进而可能暴露错误，例如在方法不存在、参数类型错误或方法为泛型方法且类型推断失败这些情况下。

查询表达式的处理方式为：重复应用以下转换，直到不可能进一步缩减。转换按应用顺序列出：每一部分都假设前面部分的转换已彻底执行，一旦某个部分彻底执行，之后在同一查询表达式的处理过程中便不再重新访问该部分。

不允许对查询表达式中的范围变量进行赋值。但允许 C# 实现在某些时候可以不实施此限制，因为对于此处介绍的句法转换方案，有些时候可能根本无法实施此限制。

某些转换使用由 * 指示的透明标识符注入范围变量。透明标识符的特殊属性将在之后进一步讨论（请参见第 7.16.2.7 节）。

7.16.2.1 带继续符的 select 和 groupby 子句

带继续符的查询表达式

```
from … into x …
```

转换为

```
from x in ( from … ) …
```

后面几节中的转换假定查询中没有 into 延续部分。

下面的示例

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

转换为

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
from g in
   from c in customers
   group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

其最终转换为

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

7.16.2.2 显式范围变量类型

显式指定范围变量类型的from子句

```
from T x in e
```

转换为

```
from x in ( e ) . Cast < T > ( )
```

显式指定范围变量类型的join子句

```
join T x in e on k1 equals k2
```

转换为

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

以下部分的转换假定查询没有显式范围变量类型。

下面的示例

```
from Customer c in customers
where c.City == "London"
select c
```

转换为

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

其最终转换为

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

显式范围变量类型对于查询实现非泛型IEnumerable接口的集合很有用，但对于实现泛型IEnumerable<T>接口的集合没什么用处。如果customers属于ArrayList类型，则在上面的示例中即会如此。

7.16.2.3 退化查询表达式

以下形式的查询表达式

```
from x in e select x
```

转换为

```
( e ) . Select ( x => x )
```

下面的示例

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
from c in customers
select c
```

转换为

```
customers.Select(c => c)
```

退化查询表达式是平常选择源的元素的查询表达式。后面的转换阶段会移除由其他转换步骤引入的退化查询，方法是用其源替换这些退化查询。然而，确保查询表达式的结果永不为源对象本身非常重要，因为这样会向查询的客户端暴露源的类型和标识符。因此，此步骤可通过在源上显式调用 Select 来保护直接以源代码形式写入的简并查询。然后，由 Select 实施者及其他查询操作员确保这些方法永远不会返回源对象本身。

7.16.2.4 from、let、where、join 和 orderby 子句

带有另一个 from 子句且后接一个 select 子句的查询表达式

```
from x1 in e1
from x2 in e2
select v
```

转换为

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

带有另一个 from 子句且后接除 select 以外的任何子句的查询表达式

```
from x1 in e1
from x2 in e2
…
```

转换为

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
…
```

带有 let 子句的查询表达式

```
from x in e
let y = f
…
```

转换为

```
from * in ( e ) . Select ( x => new { x , y = f } )
…
```

带有 where 子句的查询表达式

```
from x in e
where f
…
```

转换为

```
from x in ( e ) . Where ( x => f )
…
```

带有 join 子句（不含 into）且后接 select 子句的查询表达式

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

转换为

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

带有 join 子句（不含 into）且后接除 select 子句之外的其他内容的查询表达式

```
from x1 in e1
join x2 in e2 on k1 equals k2
…
```

转换为

```
from * in ( e1 ) . Join(
   e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
…
```

带有 join 子句（含 into）且后接 select 子句的查询表达式

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

转换为

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

带有 join 子句（含 into）且后接除 select 子句之外的其他内容的查询表达式

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
…
```

转换为

```
from * in ( e1 ) . GroupJoin(
   e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
…
```

带有 orderby 子句的查询表达式

```
from x in e
orderby k1 , k2 , … , kn
…
```

转换为

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
 … .
ThenBy ( x => kn )
…
```

如果排序子句指定 descending 方向指示器，则将改为生成对 OrderByDescending 或 ThenByDescending 的调用。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

下面的转换假定在每个查询表达式中没有 let、where、join 或 orderby 子句，并且最多只有一个初始 from 子句。

下面的示例

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

转换为

```
customers.
SelectMany(c => c.Orders,
    (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

下面的示例

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

转换为

```
from * in customers.
   SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

其最终转换为

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

其中 x 是编译器生成的以其他方式不可见且不可访问的标识符。

下面的示例

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

转换为

```
from * in orders.
   Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

其最终转换为

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

其中 x 是编译器生成的以其他方式不可见且不可访问的标识符。

下面的示例

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

转换为

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
   (c, o) => new { c.Name, o.OrderDate, o.Total })
```

下面的示例

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

转换为

```
from * in customers.
   GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
      (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

其最终转换为

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
   (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

其中 x 和 y 是编译器生成的以其他方式不可见且不可访问的标识符。

下面的示例

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

具有最终转换

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

7.16.2.5 select 子句

以下形式的查询表达式

```
from x in e select v
```

转换为

```
( e ) . Select ( x => v )
```

除了当 *v* 为标识符 *x* 时，转换仅为

```
( e )
```

例如

```
from c in customers.Where(c => c.City == “London”)
select c
```

仅转换为

```
customers.Where(c => c.City == “London”)
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

7.16.2.6 Groupby 子句

以下形式的查询表达式

```
from x in e group v by k
```

转换为

```
( e ) . GroupBy ( x => k , x => v )
```

除了当 *v* 为标识符 *x* 时，转换为

```
( e ) . GroupBy ( x => k )
```

下面的示例

```
from c in customers
group c.Name by c.Country
```

转换为

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

7.16.2.7 透明标识符

某些转换使用由 * 指示的透明标识符注入范围变量。透明标识符不是合适的语言功能；它们在查询表达式转换过程中仅作为中间步骤存在。

当查询转换注入透明标识符时，进一步的转换步骤将透明标识符传播到匿名函数和匿名对象初始值设定项中。在这些上下文中，透明标识符具有以下行为：

- 当透明标识符作为某个匿名函数中的参数出现时，关联匿名类型的成员自动处于该匿名函数的函数体范围内。
- 当带透明标识符的成员位于范围内时，该成员的成员也位于范围内。
- 当透明标识符作为匿名对象初始值设定项中的成员声明符出现时，会引入一个具有透明标识符的成员。

在上面所述的转换步骤中，透明标识符始终与匿名类型一起引入，目的是以一个对象的成员的形式捕获多个范围变量。允许 C# 的实现使用不同于匿名类型的机制将多个范围变量组合在一起。下面的转换示例假定使用匿名类型，演示如何转换透明标识符。

下面的示例

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

转换为

```
from * in customers.
   SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

进一步转换为

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在清除透明标识符后等效于

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

其中 x 是编译器生成的以其他方式不可见且不可访问的标识符。

下面的示例

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

转换为

```
from * in customers.
   Join(orders, c => c.CustomerID, o => o.CustomerID,
      (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

进一步缩减为

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

其最终转换为

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
   (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
   (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
   (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

其中 x、y 和 z 是编译器生成的以其他方式不可见且不可访问的标识符。

7.16.3 查询表达式模式

查询表达式模式 (Query expression pattern) 建立了一种方法模式，类型可以实现该模式来支持查询表达式。因为查询表达式通过句法映射转换为方法调用，所以类型在如何实现查询表达式模式方面具有很大灵活性。例如，该模式的方法可以以实例方法或扩展方法的形式实现，因为这两种方法具有相同的调用语法，而且方法可以请求委托或表达式树，因为匿名函数可以转换为这两者。

支持查询表达式模式的泛型类型 C<T> 的建议形式如下所示。使用泛型类型是为了演示参数和结果类型之间的正确关系，不过也可以为非泛型类型实现该模式。

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
   public C<T> Cast<T>();
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class C<T> : C
{
   public C<T> Where(Func<T,bool> predicate);
   public C<U> Select<U>(Func<T,U> selector);
   public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
      Func<T,U,V> resultSelector);
   public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
      Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);
   public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
      Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);
   public O<T> OrderBy<K>(Func<T,K> keySelector);
   public O<T> OrderByDescending<K>(Func<T,K> keySelector);
   public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
   public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
      Func<T,E> elementSelector);
}
class O<T> : C<T>
{
   public O<T> ThenBy<K>(Func<T,K> keySelector);
   public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}
class G<K,T> : C<T>
{
   public K Key { get; }
}
```

上述方法使用泛型委托类型 Func<T1, R> 和 Func<T1, T2, R>，不过也可以使用在参数和结果类型中具有相同关系的其他委托或表达式目录树类型。

请注意 C<T> 和 O<T> 之间的建议关系，该关系可确保 ThenBy 和 ThenByDescending 方法只能用于 OrderBy 或 OrderByDescending 的结果。同时请注意 GroupBy 结果的推荐形式 — 一系列序列，其中每个内部序列都有一个附加的 Key 属性。

System.Linq 命名空间为实现 System.Collections.Generic.IEnumerable<T> 接口的任何类型提供了一个查询运算符模式的实现。

7.17 赋值运算符

赋值运算符为变量、属性、事件或索引器元素赋新值。

assignment:
unary-expression assignment-operator expression

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

assignment-operator:
=
+=
-=
*=
/=
%=
&=
|=
^=
<<=
right-shift-assignment

赋值的左操作数必须是属于变量、属性访问、索引器访问或事件访问类别的表达式。

= 运算符称为*简单赋值运算符*。它将右操作数的值赋予左操作数给定的变量、属性或索引器元素。简单赋值运算符的左操作数一般不可以是一个事件访问（第 10.8.1 节中描述的例外）。简单赋值运算符的介绍详见第 7.17.1 节。

除 = 运算符以外的赋值运算符称为复合赋值运算符 (compound assignment operator)。这些运算符对两个操作数执行指示的运算，然后将结果值赋予左操作数指定的变量、属性或索引器元素。复合赋值运算符的介绍详见第 7.17.2 节。

以事件访问表达式作为左操作数的 += 和 -= 运算符称为*事件赋值运算符*。当左操作数是事件访问时，其他赋值运算符都是无效的。事件赋值运算符的介绍详见第 7.17.3 节。

赋值运算符为向右关联，即此类运算从右到左分组。例如，a = b = c 形式的表达式可以按 a = (b = c) 进行计算。

7.17.1 简单赋值

= 运算符称为简单赋值运算符。

如果简单赋值的左操作数为 E.P 或 E[E_i] 形式，其中 E 具有编译时类型 dynamic，则赋值是动态绑定的（第 7.2.2 节）。在此情况下，赋值表达式的编译时类型为 dynamic，并且会在运行时基于 E 的运行时类型进行下面所述的决策。

在简单赋值中，右操作数必须为可以隐式转换为左操作数所属类型的表达式。运算将右操作数的值赋予左操作数指定的变量、属性或索引器元素。

简单赋值表达式的结果是赋予左操作数的值。结果的类型与左操作数相同，且始终为值类别。

如果左操作数为属性或索引器访问，则该属性或索引器必须具有 set 访问器。如果不是这样，则发生绑定时错误。

x = y 形式的简单赋值的运行时处理包括以下步骤：

- 如果 x 属于变量：
  - 计算 x 以产生变量。
  - 计算 y，必要时还需通过隐式转换（第 0 节）将其转换为 x 的类型。
  - 如果 x 给定的变量是 *reference-type* 的数组元素，则执行运行时检查以确保为 y 计算的值与以 x 为其元素的那个数组实例兼容。如果 y 为 null，或存在从 y 引用的实例的实际类型到包含 x 的

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

数组实例的实际元素类型的隐式引用转换（第6.1.6节），则检查成功。否则，将引发 System.ArrayTypeMismatchException。

- y 的计算和转换后所产生的值存储在 x 的计算所确定的位置中。

- 如果 x 属于属性或索引器访问：
  - 计算与 x 关联的实例表达式（如果 x 不是 static）和参数列表（如果 x 是索引器访问），结果用于后面的对和 set 访问器调用。
  - 计算 y，必要时还需通过隐式转换（第0节）将其转换为 x 的类型。
  - 使用针对 y 计算的值作为 value 参数调用 x 的 set 访问器。

如果存在从 B 到 A 的隐式引用转换，则数组协变规则（第12.5节）允许数组类型 A[] 的值是对数组类型 B[] 的实例的引用。由于这些规则，对 *reference-type* 的数组元素的赋值需要运行时检查以确保所赋的值与数组实例兼容。在下面的示例中

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;               // Ok
oa[1] = "Hello";            // Ok
oa[2] = new ArrayList();    // ArrayTypeMismatchException
```

最后的赋值将导致引发 System.ArrayTypeMismatchException，这是因为 ArrayList 的实例不能存储在 string[] 的元素中。

当 *struct-type* 中声明的属性或索引器是赋值的目标时，与属性或索引器访问关联的实例表达式必须为变量类别。如果该实例表达式归类为值类别，则发生绑定时错误。由于第7.6.4节中所说明的原因，同样的规则也适用于字段。

给定下列声明：

```
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;
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public Point A {
           get { return a; }
           set { a = value; }
        }
        public Point B {
           get { return b; }
           set { b = value; }
        }
    }
```

在下面的示例中

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

由于 p 和 r 为变量，因此允许对 p.X、p.Y、r.A 和 r.B 进行赋值。但是，在以下示例中

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

由于 r.A 和 r.B 不是变量，因此赋值全部无效。

7.17.2 复合赋值

如果复合赋值的左操作数为 E.P 或 E[E_i] 形式，其中 E 具有编译时类型 dynamic，则赋值为动态绑定的（第 7.2.2 节）。在此情况下，赋值表达式的编译时类型为 dynamic，并且会在运行时基于 E 的运行时类型进行下面所述的决策。

x *op*= y 形式的运算是这样来处理的：应用重载决策（第 7.3.4 节），就好比运算的书写形式为 (x) *op* y。让 R 成为选定运算符的返回类型，以及 x 的 T 类型。然后，

- 如果存在从类型 R 的表达式到类型 T 的*隐式*转换，则将按 x = (T)((x) *op* y) 计算操作，但是只计算 x 一次。
- 否则，如果选定运算符为预定义的运算符，如果 R 可*显式*转换为 T，并且 y 可*隐式*转换为 T，或者运算符是移位运算符，则将按 x = (T)((x) *op* y) 计算操作，但是只计算 x 一次。
- 否则，复合赋值无效，且发生绑定时错误。

术语“只计算一次”表示：在 x *op* y 的计算中，x 的任何要素表达式的计算结果都临时保存起来，然后在执行对 x 的赋值时重用这些结果。例如，在计算赋值 A()[B()] += C() 时（其中 A 为返回 int[] 的方法，B 和 C 为返回 int 的方法），按 A、B、C 的顺序只调用这些方法一次。

当复合赋值的左操作数为属性访问或索引器访问时，属性或索引器必须同时具有 get 访问器和 set 访问器。如果不是这样，则发生绑定时错误。

上面的第二条规则允许在某些上下文中将 x *op*= y 按 x = (T)((x) *op* y) 计算。按此规则，当左操作数为 sbyte、byte、short、ushort 或 char 类型时，预定义的运算符可用作复合运算符。甚至当两个参数都为这些类型之一时，预定义的运算符也产生 intint 类型的结果，详见第 7.3.6.2 节中的介绍。因此，不进行强制转换，就不可能把结果赋值给左操作数。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

此规则对预定义运算符的直观效果只是：如果同时允许 (x) *op* y 和 x = y，则允许 x *op*= y。在下面的示例中

```
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
```

每个错误的直观理由是对应的简单赋值也发生错误。

这还意味着复合赋值运算支持提升运算。在下面的示例中

```
int? i = 0;
i += 1;            // Ok
```

使用了提升运算符 +(int?,int?)。

7.17.3 事件赋值

如果 += 或 -= 运算符的左操作数属于事件访问类别，则表达式按下面这样计算：

- 计算事件访问的实例表达式（如果有）。
- 计算 += 或 -= 运算符的右操作数，如果需要，通过隐式转换（第 0 节）转换为左操作数的类型。
- 调用该事件的事件访问器，所需的参数列表由右操作数（经过计算和必要的转换后）组成。如果运算符为 +=，则调用 add 访问器；如果运算符为 -=，则调用 remove 访问器。

事件赋值表达式不产生值。因此，事件赋值表达式只在 *statement-expression*（第 8.6 节）的上下文中是有效的。

7.18 表达式

expression 为 *non-assignment-expression* 或 *assignment*。

expression:
    non-assignment-expression
    assignment

non-assignment-expression:
    conditional-expression
    lambda-expression
    query-expression

7.19 常量表达式

constant-expression 是在编译时可以完全计算出结果的表达式。

constant-expression:
    expression

常量表达式必须为 null 文本或以下某种类型的值：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、object、string 或任何枚举类型。常量表达式中仅允许下列构造：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 文本（包括 null 文本）。
- 对类和结构类型的 const 成员的引用。
- 对枚举类型的成员的引用。
- 对 const 参数或局部变量的引用
- 带括号的子表达式，其自身是常量表达式。
- 强制转换表达式（前提是目标类型为以上列出的类型之一）。
- checked 和 unchecked 表达式
- 默认值表达式
- 预定义的一元运算符 +、 –、 ! 和 ~。
- 预定义的二元运算符 +、 –、 *、 /、 %、 <<、 >>、 &、 |、 ^、 &&、 ||、 ==、 !=、 <、 >、 <= 和 >=（前提是每个操作数都为上面列出的类型）。
- 条件运算符 ?:。

常量表达式中允许下列转换：

- 标识转换
- 数值转换
- 枚举转换
- 常量表达式转换
- 隐式和显式引用转换，条件是转换的源是计算结果为Null 值的常量表达式。

在常量表达式中，不进行其他转换，包括非 Null 值的装箱、取消装箱和隐式引用转换。例如：

```
class C {
    const object i = 5;  // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

因为需要装箱转换，i 的初始化出错。因为需要对非 null 值的隐式引用转换，str 的初始化出错。

只要表达式满足以上所列要求，则将在编译时计算该表达式。即使该表达式是另一个包含有非常量构造的较大表达式的子表达式，亦是如此。

常量表达式的编译时计算使用与非常量表达式的运行时计算相同的规则，区别仅在于：当出现错误时，运行时计算引发异常，而编译时计算导致发生编译时错误。

除非常量表达式被显式放置在 unchecked 上下文中，否则在表达式的编译时计算期间，整型算术运算和转换中发生的溢出总是导致编译时错误（第 7.19 节）。

常量表达式出现在以下列出的上下文中。在这些上下文中，如果无法在编译时充分计算表达式，则发生编译时错误。

- 常量声明（第 10.4 节）。
- 枚举成员声明（第 14.3 节）。
- 形参表的默认参数（第 10.6.1 节）

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- switch 语句的 case 标签（第 8.7.2 节）。
- goto case 语句（第 8.9.3 节）。
- 包含初始值设定项的数组创建表达式（第 7.6.10.4 节）中的维度长度。
- 特性（第 17 章）。

只要常量表达式的值在目标类型的范围内，隐式常量表达式转换（第 6.1.9 节）就允许将 int 类型的常量表达式转换为 sbyte、 byte、short、ushort、uint 或 ulong。

7.20 布尔表达式

boolean-expression 为产生 bool 类型结果的表达式，产生方式或者是直接产生，或者是通过在如下指定的某些上下文中应用 operator true 产生。

boolean-expression:
expression

if-statement（第 8.7.1 节）、*while-statement*（第 8.8.1 节）、*do-statement*（第 8.8.2 节）或 *for-statement*（第 8.8.3 节）的控制条件表达式都是 *boolean-expression*。?: 运算符（第 7.14 节）的控制条件表达式遵守与 *boolean-expression* 相同的规则，但由于运算符优先级的缘故，被归为 *conditional-or-expression*。

必须有 *boolean-expression* E 才能产生 bool 类型的值，如下所示：

- 如果 E 可隐式转换为 bool，则在运行时应用该隐式转换。
- 否则，使用一元运算符重载决策（第 7.3.3 节）查找运算符 true 对 E 的唯一最佳实现，并在运行时应用该实现。
- 如果未找到此类运算符，则发生绑定时错误。

第 11.4.2 节中的 DBBool 结构类型提供了一个实现了 operator true 和 operator false 的类型的示例。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

8. 语句

C# 提供各种语句。使用过 C 和 C++ 编程的开发人员熟悉其中大多数语句。

statement:
labeled-statement
declaration-statement
embedded-statement

embedded-statement:
block
empty-statement
expression-statement
selection-statement
iteration-statement
jump-statement
try-statement
checked-statement
unchecked-statement
lock-statement
using-statement
yield-statement

embedded-statement 非终结符用于在其他语句内出现的语句。使用 *embedded-statement*（而非 *statement*）便不需要在这些上下文中使用声明语句和标记语句。下面的示例

```
void F(bool b) {
   if (b)
      int i = 44;
}
```

将导致编译时错误，原因是 `if` 语句的 if 分支要求 *embedded-statement* 而不是 *statement*。若允许执行上述代码，则声明了变量 `i`，却永远无法使用它。但是请注意，如果是将 `i` 的声明放置在一个块中，则该示例就是有效的。

8.1 结束点和可到达性

每个语句都有一个结束点 (end point)。直观地讲，语句的结束点是紧跟在语句后面的那个位置。复合语句（包含嵌入语句的语句）的执行规则规定了当控制到达一个嵌入语句的结束点时所采取的操作。例如，当控制到达块中某个语句的结束点时，控制就转到该块中的下一个语句。

如果执行流程可能到达某个语句，则称该语句是可到达的 (reachable)。相反，如果某个语句不可能被执行，则称该语句是不可到达的 (unreachable)。

在下面的示例中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
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，这是因为不可能执行该语句。

如果编译器确定某个语句是不可到达的，将会报出警告。准确地说，语句不可到达不算是错误。

为了确定某个特定的语句或结束点是否可到达，编译器根据为各语句定义的可到达性规则进行控制流分析。控制流分析会考虑那些能控制语句行为的常量表达式（第7.19节）的值，但不考虑非常量表达式的可能值。换句话说，出于控制流分析的目的，给定类型的非常量表达式被认为具有该类型的任何可能值。

在下面的示例中

```
void F() {
   const int i = 1;
   if (i == 2) Console.WriteLine("unreachable");
}
```

if 语句的布尔表达式是常量表达式，原因是 == 运算符的两个操作数都是常量。由于该常量表达式在编译时进行计算并产生值 false，所以 Console.WriteLine 调用被认为是不可到达的。但是，如果 i 更改为局部变量

```
void F() {
   int i = 1;
   if (i == 2) Console.WriteLine("reachable");
}
```

则 Console.WriteLine 调用被认为是可到达的，即使它实际上永远不会被执行。

函数成员的 *block* 始终被认为是可到达的。通过依次计算块中各语句的可到达性规则，可以确定任何给定语句的可到达性。

在下面的示例中

```
void F(int x) {
   Console.WriteLine("start");
   if (x < 0) Console.WriteLine("negative");
}
```

第二个 Console.WriteLine 的可到达性按下面的规则确定：

- 第一个 Console.WriteLine 表达式语句是可到达的，原因是 F 方法块是可到达的。
- 第一个 Console.WriteLine 表达式语句的结束点是可到达的，原因是该语句是可到达的。
- if 语句是可到达的，原因是第一个 Console.WriteLine 表达式语句的结束点是可到达的。
- 第二个 Console.WriteLine 表达式语句是可到达的，原因是 if 语句的布尔表达式没有常量值 false。

在下列两种情况下，如果某个语句的结束点是可以到达的，则会出现编译时错误：

- 由于 switch 语句不允许一个 switch 节“贯穿”到下一个 switch 节，因此如果一个 switch 节的语句列表的结束点是可到达的，则会出现编译时错误。如果发生此错误，则通常表明该处遗漏了一个 break 语句。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果一个计算某个值的函数成员的块的结束点是可到达的，则会出现编译时错误。如果发生此错误，则通常表明该处遗漏了一个 `return` 语句。

8.2 块

block 用于在只允许使用单个语句的上下文中编写多条语句。

block:
`{` *statement-list*$_{opt}$ `}`

block 由一个扩在大括号内的可选 *statement-list*（第 8.2.1 节）组成。如果没有此语句列表，则称块是空的。

块可以包含声明语句（第 8.5 节）。在一个块中声明的局部变量或常量的范围就是该块本身。

在块内，在表达式上下文中使用的名称的含义必须始终相同（第 7.6.2.1 节）。

块按下述规则执行：

- 如果块是空的，控制转到块的结束点。
- 如果块不是空的，控制转到语句列表。当（如果）控制到达语句列表的结束点时，控制转到块的结束点。

如果块本身是可到达的，则块的语句列表是可到达的。

如果块是空的或者如果语句列表的结束点是可到达的，则块的结束点是可到达的。

包含一条或多条 `yield` 语句（第 8.14 节）的 *block* 称为迭代器块。迭代器块用于以迭代器的形式实现函数成员（第 10.14 节）。某些附加限制适用于迭代器块：

- 迭代器块中出现 `return` 语句时会产生编译时错误（但允许 `yield return` 语句）。
- 迭代器块包含不安全的上下文（第 18.1 节）时将导致编译时错误。迭代器块总是定义安全的上下文，即使其定义嵌套在不安全的上下文中也如此。

8.2.1 语句列表

语句列表 (statement list) 由一个或多个顺序编写的语句组成。语句列表出现在 *block*（第 8.2 节）和 *switch-block*（第 8.7.2 节）中。

statement-list:
statement
statement-list statement

执行一个语句列表就是将控制转到该列表中的第一个语句。当（如果）控制到达某条语句的结束点时，控制将转到下一个语句。当（如果）控制到达最后一个语句的结束点时，控制将转到语句列表的结束点。

如果下列条件中至少一个为真，则语句列表中的一个语句是可到达的：

- 该语句是第一个语句且语句列表本身是可到达的。
- 前一个语句的结束点是可到达的。
- 该语句本身是一个标记语句，并且该标签已被一个可到达的 `goto` 语句引用。

如果列表中最后一个语句的结束点是可到达的，则语句列表的结束点是可到达的。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

8.3 空语句

empty-statement 什么都不做。

empty-statement:
;

当在要求有语句的上下文中不执行任何操作时，使用空语句。

执行一个空语句就是将控制转到该语句的结束点。这样，如果空语句是可到达的，则空语句的结束点也是可到达的。

当编写一个语句体为空的 `while` 语句时，可以使用空语句：

```
bool ProcessMessage() {...}

void ProcessMessages() {
   while (ProcessMessage())
      ;
}
```

此外，空语句还可以用于在块的结束符“`}`”前声明标签：

```
void F() {
   ...
   if (done) goto exit;
   ...
   exit: ;
}
```

8.4 标记语句

labeled-statement 可以给语句加上一个标签作为前缀。标记语句可以出现在块中，但是不允许它们作为嵌入语句。

labeled-statement:
identifier : *statement*

标记语句声明了一个标签，它由 *identifier* 来命名。标签的范围为在其中声明了该标签的整个块，包括任何嵌套块。两个同名的标签若具有重叠的范围，则会产生一个编译时错误。

标签可以在该标签的范围内被 `goto` 语句（第 8.9.3 节）引用。这意味着 `goto` 语句可以在它所在的块内转移控制，也可以将控制转到该块外部，但是永远不能将控制转入该块所含的嵌套块的内部。

标签具有自己的声明空间，并不影响其他标识符。下面的示例

```
int F(int x) {
   if (x >= 0) goto x;
   x = -x;
   x: return x;
}
```

是有效的，尽管它将 `x` 同时用作参数和标签的名称。

执行一个标记语句就是执行该标签后的那个语句。

除由正常控制流程提供的可到达性外，如果一个标签由一个可到达的 `goto` 语句引用，则该标记语句是可到达的。（异常：如果 `goto` 语句在一个包含了 `finally` 块的 `try` 中，标记语句在 `try` 之外，并且 `finally` 块的结束点不可到达，则从该 `goto` 语句不可到达上述标记语句。）

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

8.5 声明语句

declaration-statement 声明局部变量或常量。声明语句可以出现在块中，但不允许它们作为嵌入语句使用。

declaration-statement:
    local-variable-declaration ;
    local-constant-declaration ;

8.5.1 局部变量声明

local-variable-declaration 声明一个或多个局部变量。

local-variable-declaration:
    local-variable-type *local-variable-declarators*

local-variable-type:
    type
    var

local-variable-declarators:
    local-variable-declarator
    local-variable-declarators , *local-variable-declarator*

local-variable-declarator:
    identifier
    identifier = *local-variable-initializer*

local-variable-initializer:
    expression
    array-initializer

local-variable-declaration 的 *local-variable-type* 要么直接指定声明引入的变量的类型，要么通过标识符 var 指示应基于初始值设定项来推断该类型。此类型后接一个 *local-variable-declarator* 列表，其中每一项都引入一个新变量。*local-variable-declarator* 由一个命名变量的 *identifier* 组成，根据需要此 identifier 后可接一个“=”标记和一个赋予变量初始值的 *local-variable-initializer*。

在局部变量声明的上下文中，标识符 var 充当上下文关键字（第 2.4.3 节）。将 *local-variable-type* 指定为 var 且作用域内没有名为 var 的类型时，则该声明为隐式类型化局部变量声明 (implicitly typed local variable declaration)，其类型从关联的初始值设定项表达式的类型推断。隐式类型化局部变量声明受到以下限制：

- *local-variable-declaration* 不能包含多个 *local-variable-declarator*。
- *local-variable-declarator* 必须包含一个 *local-variable-initializer*。
- *local-variable-initializer* 必须是 *expression*。
- 初始值设定项 *expression* 必须具有编译时类型。
- 初始值设定项 *expression* 不能引用声明的变量本身。

下面是不正确的隐式类型化局部变量声明的示例：

```
var x;                // Error, no initializer to infer type from
var y = {1, 2, 3};    // Error, array initializer not permitted
var z = null;         // Error, null does not have a type
var u = x => x + 1;   // Error, anonymous functions do not have a type
var v = v++;          // Error, initializer cannot refer to variable itself
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

可以在表达式中通过 *simple-name*（第 7.6.2 节）来获取局部变量的值，还可以通过 *assignment*（第 7.17 节）来修改局部变量的值。在使用局部变量的每个地方必须先明确对其赋值（第 5.3 节），然后才可使用它的值。

在 *local-variable-declaration* 中声明的局部变量范围是该声明所在的块。在一个局部变量的 *local-variable-declarator* 之前的文本位置中引用该局部变量是错误的。在一个局部变量的范围内声明其他具有相同名称的局部变量或常量是编译时错误。

声明了多个变量的局部变量声明等效于多个同一类型的单个变量的声明。另外，局部变量声明中的变量初始值设定项完全对应于紧跟该声明后插入的赋值语句。

下面的示例

```
void F() {
   int x = 1, y, z = x * 2;
}
```

完全对应于

```
void F() {
   int x; x = 1;
   int y;
   int z; z = x * 2;
}
```

在隐式类型化局部变量声明中，假定所声明的局部变量的类型与用于初始化该变量的表达式的类型相同。例如：

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

上述隐式类型化局部变量声明与下面显式类型化的声明完全等效：

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

8.5.2 局部常量声明

local-constant-declaration 用于声明一个或多个局部常量。

local-constant-declaration:
    `const` *type constant-declarators*

constant-declarators:
    constant-declarator
    constant-declarators `,` *constant-declarator*

constant-declarator:
    identifier `=` *constant-expression*

local-constant-declaration 的 *type* 指定由该声明引入的常量的类型。此类型后接一个 *constant-declarator* 列表，其中每一项都引入一个新常量。c*constant-declarator* 包含一个命名常量的 *identifier*，后接一个“=”标记，然后是一个对该常量赋值的 *constant-expression*（第 7.19 节）。

局部常量声明的 *type* 和 *constant-expression* 必须遵循与常量成员声明（第 10.4 节）一样的规则。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

可以在表达式中通过 *simple-name*（第 7.6.2 节）来获取局部常量的值。

局部常量的范围是在其中声明了该常量的块。在局部常量的 *constant-declarator* 之前的文本位置中引用该局部常量是错误的。在局部常量的范围内声明其他具有相同名称的局部变量或常量是编译时错误。

声明多个常量的局部常量声明等效于多个同一类型的单个常量的声明。

8.6 表达式语句

expression-statement 用于计算所给定的表达式。由此表达式计算出来的值（如果有）被丢弃。

expression-statement:
    statement-expression ;

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression
    await-expression

不是所有的表达式都允许作为语句使用。具体而言，不允许像 `x + y` 和 `x == 1` 这样只计算一个值（此值将被放弃）的表达式作为语句使用。

执行一个 *expression-statement* 就是对包含的表达式进行计算，然后将控制转到该 *expression-statement* 的结束点。如果一个 *expression-statement* 是可到达的，则 *expression-statement* 结束点也是可到达的。

8.7 选择语句

选择语句会根据表达式的值从若干个给定的语句中选择一个来执行。

selection-statement:
    if-statement
    switch-statement

8.7.1 if 语句

`if` 语句根据布尔表达式的值选择要执行的语句。

if-statement:
    `if` `(` *boolean-expression* `)` *embedded-statement*
    `if` `(` *boolean-expression* `)` *embedded-statement* `else` *embedded-statement*

`else` 部分与语法允许的、词法上最相近的上一个 `if` 语句相关联。因而，下列形式的 `if` 语句

```
if (x) if (y) F(); else G();
```

相当于

```
if (x) {
   if (y) {
      F();
   }
   else {
      G();
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

if 语句按如下规则执行：

- 计算 *boolean-expression*（第 7.20 节）。
- 如果布尔表达式产生 true，则控制转到第一个嵌入语句。当（如果）控制到达那条语句的结束点时，控制将转到 if 语句的结束点。
- 如果布尔表达式产生 false 且如果存在 else 部分，则控制转到第二个嵌入语句。当（如果）控制到达那条语句的结束点时，控制将转到 if 语句的结束点。
- 如果布尔表达式产生 false 且如果不存在 else 部分，则控制转到 if 语句的结束点。

如果 if 语句是可到达的且布尔表达式不具有常量值 false，则 if 语句的第一个嵌入语句是可到达的。

如果 if 语句是可到达的且布尔表达式不具有常量值 true，则 if 语句的第二个嵌入语句（如果存在）是可到达的。

如果 if 语句的至少一个嵌入语句的结束点是可到达的，则 if 语句的结束点是可到达的。此外，对于不具有 else 部分的 if 语句，如果 if 语句是可到达的且布尔表达式不具有常量值 true，则该 if 语句的结束点是可到达的。

8.7.2 switch 语句

switch 语句选择一个要执行的语句列表，此列表具有一个相关联的 switch 标签，它对应于 switch 表达式的值。

switch-statement:
    switch (*expression*) *switch-block*

switch-block:
    { *switch-sections*$_{opt}$ }

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-statement 包含关键字 switch，后接带括号的表达式（称为 switch 表达式），然后是一个 *switch-block*。*switch-block* 包含零个或多个括在大括号内的 *switch-section*。每个 *switch-section* 包含一个或多个 *switch-labels*，后接一个 *statement-list*（第 8.2.1 节）。

switch 语句的主导类型 (governing type) 由 switch 表达式确定。

- 如果 switch 表达式的类型为 sbyte、byte、short、ushort、int、uint、long、ulong、bool、char、string 或 *enum-type*，或者是对应于以上某种类型的可以为 null 的类型，则该类型就是 switch 语句的主导类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，必须有且只有一个用户定义的从switch表达式的类型到下列某个可能的主导类型的隐式转换（第6.4节）：sbyte、byte、short、ushort、int、uint、long、ulong、char、string或对应于以上某种类型的可以为null的类型。
- 否则，如果不存在这样的隐式转换，或者存在多个这样的隐式转换，则会发生编译时错误。

每个case标签的常量表达式都必须表示一个可隐式转换（第0节）为switch语句的主导类型的值。如果同一switch语句中的两个或更多个case标签指定同一个常量值，则会导致编译时错误。

一个switch语句中最多只能有一个default标签。

switch语句按如下规则执行：

- 计算switch表达式并将其转换为主导类型。
- 如果在该switch语句的case标签中，有一个指定的常量恰好等于switch表达式的值，控制将转到匹配的case标签后的语句列表。
- 如果在该switch语句的case标签中，指定的常量都不等于switch表达式的值，且如果存在一个default标签，则控制将转到default标签后的语句列表。
- 如果在该switch语句的case标签中，指定的常量都不等于switch表达式的值，且如果不存在default标签，则控制将转到switch语句的结束点。

如果switch节的语句列表的结束点是可到达的，将发生编译时错误。这称为“无贯穿”规则。下面的示例

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
default:
   CaseOthers();
   break;
}
```

是有效的，这是因为没有一个switch节的结束点是可到达的。与C和C++不同，执行一个switch节的过程不能“贯穿”到下一个switch节，示例

```
switch (i) {
case 0:
   CaseZero();
case 1:
   CaseZeroOrOne();
default:
   CaseAny();
}
```

会导致编译时错误。如果要在执行一个switch节后继续执行另一个switch节，则必须使用显式的goto case或goto default语句：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
switch (i) {
case 0:
   CaseZero();
   goto case 1;
case 1:
   CaseZeroOrOne();
   goto default;
default:
   CaseAny();
   break;
}
```

在一个 *switch-section* 中允许有多个标签。下面的示例

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
case 2:
default:
   CaseTwo();
   break;
}
```

是有效的。此示例不违反“无贯穿”规则，这是因为标签 `case 2:` 和 `default:` 属于同一个 *switch-section*。

“无贯穿”规则防止了在C 和 C++ 中由不经意地漏掉了 `break` 语句而引起的一类常见 Bug。另外，由于这个规则，`switch` 语句的各个 switch 节可以任意重新排列而不会影响语句的行为。例如，上面 `switch` 语句中的各节的顺序可以在不影响语句行为的情况下反转排列：

```
switch (i) {
default:
   CaseAny();
   break;
case 1:
   CaseZeroOrOne();
   goto default;
case 0:
   CaseZero();
   goto case 1;
}
```

switch 节的语句列表通常以 `break`、`goto case` 或 `goto default` 语句结束，但是也可以使用任何其他结构，只要它能保证对应的语句列表的结束点是不可到达的。例如，由布尔表达式 `true` 控制的 `while` 语句是永远无法到达其结束点的。同样，`throw` 或 `return` 语句始终将控制转到其他地方而从不到达它的结束点。因此，下列示例是有效的：

```
switch (i) {
case 0:
   while (true) F();
case 1:
   throw new ArgumentException();
case 2:
   return;
}
```

`switch` 语句的主导类型可以是 `string` 类型。例如：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
void DoCommand(string command) {
   switch (command.ToLower()) {
   case "run":
      DoRun();
      break;
   case "save":
      DoSave();
      break;
   case "quit":
      DoQuit();
      break;
   default:
      InvalidCommand(command);
      break;
   }
}
```

与字符串相等运算符（第7.10.7节），switch语句区分大小写，因而只有在switch表达式字符串与case与标签常量完全匹配时才会执行给定的switch节。

当switch语句的主导类型为string时，允许值null作为case标签常量。

*switch-block*的*statement-list*可以包含声明语句（第8.5节）。在switch块中声明的局部变量或常量的范围是该switch块。

在switch块内，表达式上下文中使用的名称的含义必须始终相同（第7.6.2.1节）。

如果switch语句是可到达的且下列条件至少有一个为真，则给定的switch节的语句列表是可到达的：

- switch表达式是一个非常量值。
- switch表达式是一个与该switch节中的某个case标签匹配的常量值。
- switch表达式是一个不与任何case标签匹配的常量值，且该switch节包含default标签。
- 该switch节的某个switch标签由一个可到达的goto case或goto default语句引用。

如果下列条件中至少有一个为真，则switch语句的结束点是可到达的：

- switch语句包含一个可到达的break语句（它用于退出switch语句）。
- switch语句是可到达的，switch表达式是非常量值，并且不存在default标签。
- switch语句是可到达的，switch表达式是不与任何case标签匹配的常量值，并且不存在任何default标签。

8.8 迭代语句

迭代语句重复执行嵌入语句。

iteration-statement:
while-statement
do-statement
for-statement
foreach-statement

8.8.1 while 语句

while语句按不同条件执行一个嵌入语句零次或多次。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

while-statement:
while (*boolean-expression*) *embedded-statement*

while 语句按如下规则执行：

- 计算 *boolean-expression*（第 7.20 节）。
- 如果布尔表达式产生 true，控制将转到嵌入语句。当（如果）控制到达嵌入语句的结束点（可能是通过执行一个 continue 语句）时，控制将转到 while 语句的开头。
- 如果布尔表达式产生 false，控制将转到 while 语句的结束点。

在 while 语句的嵌入语句内，break 语句（第 8.9.1 节）可用于将控制转到 while 语句的结束点（从而结束嵌入语句的迭代），而 continue 语句（第 8.9.2 节）可用于将控制转到嵌入语句的结束点（从而执行 while 语句的另一次迭代）。

如果 while 语句是可到达的且布尔表达式不具有常量值 false，则 while 语句的嵌入语句是可到达的。

如果下列条件中至少有一个为真，则 while 语句的结束点是可到达的：

- while 语句包含一个可到达的 break 语句（它用于退出 while 语句）。
- while 语句是可到达的且布尔表达式不具有常量值 true。

8.8.2 do 语句

do 语句按不同条件执行一个嵌入语句一次或多次。

do-statement:
do *embedded-statement* while (*boolean-expression*) ;

do 语句按如下规则执行：

- 控制转到嵌入语句。
- 当（如果）控制到达嵌入语句的结束点（可能是由于执行了一个 continue 语句）时，计算 *boolean-expression*（第 7.20 节）。如果布尔表达式产生 true，控制将转到 do 语句的起点。否则，控制转到 do 语句的结束点。

在 do 语句的嵌入语句内，break 语句（第 8.9.1 节）可用于将控制转到 do 语句的结束点（从而结束嵌入语句的迭代），而 continue 语句（第 8.9.2 节）可用于将控制转到嵌入语句的结束点。

如果 do 语句是可到达的，则 do 语句的嵌入语句是可到达的。

如果下列条件中至少有一个为真，则 do 语句的结束点是可到达的：

- do 语句包含一个可到达的 break 语句（它用于退出 do 语句）。
- 嵌入语句的结束点是可到达的且布尔表达式不具有常量值 true。

8.8.3 for 语句

for 语句计算一个初始化表达式序列，然后，当某个条件为真时，重复执行相关的嵌入语句并计算一个迭代表达式序列。

for-statement:
for ($for\text{-}initializer_{opt}$; $for\text{-}condition_{opt}$; $for\text{-}iterator_{opt}$) *embedded-statement*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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-initializer（如果存在）由一个 *local-variable-declaration*（第 8.5.1 节），或由一个用逗号分隔的 *statement-expression*（第 8.6 节）列表组成。用 *for-initializer* 声明的局部变量的范围从该变量的 *local-variable-declarator* 开始，一直延伸到嵌入语句的结尾。该范围包括 *for-condition* 和 *for-iterator*。

for-condition（如果存在）必须是一个 *boolean-expression*（第 7.20 节）。

for-iterator（如果存在）包含一个用逗号分隔的 *statement-expression*（第 8.6 节）列表。

for 语句按如下规则执行：

- 如果存在 *for-initializer*，则按变量初始值设定项或语句表达式的编写顺序执行它们。此步骤只执行一次。
- 如果存在 *for-condition*，则计算它。
- 如果不存在 *for-condition* 或如果计算产生 `true`，控制将转到嵌入语句。当（如果）控制到达嵌入语句的结束点（可能是因为执行了一个 `continue` 语句）时，则按顺序计算 *for-iterator* 的表达式（如果有），然后从上述步骤中的计算 *for-condition* 开始，执行另一次迭代。
- 如果存在 *for-condition*，且计算产生 `false`，控制将转到 `for` 语句的结束点。

在 `for` 语句的嵌入语句内，`break` 语句（第 8.9.1 节）可用于将控制转到 `for` 语句的结束点（从而结束嵌入语句的迭代），而 `continue` 语句（第 8.9.2 节）可用于将控制转到嵌入语句的结束点（从而执行 *for-iterator* 并从 *for-condition* 开始执行 `for` 语句的另一次迭代）。

如果下列条件之一为真，则 `for` 语句的嵌入语句是可到达的：

- `for` 语句是可到达的且不存在 *for-condition*。
- `for` 语句是可到达的，并且存在一个 *for-condition*，它不具有常量值 `false`。

如果下列条件中至少有一个为真，则 `for` 语句的结束点是可到达的：

- `for` 语句包含一个可到达的 `break` 语句（它用于退出 `for` 语句）。
- `for` 语句是可到达的，并且存在一个 *for-condition*，它不具有常量值 `true`。

8.8.4 foreach 语句

`foreach` 语句用于枚举一个集合的元素，并对该集合中的每个元素执行一次相关的嵌入语句。

foreach-statement:
`foreach` `(` *local-variable-type* *identifier* `in` *expression* `)` *embedded-statement*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

foreach 语句的 *type* 和 *identifier* 声明该语句的迭代变量(iteration variable)。如果以 *local-variable-type* 形式给定 var 标识符，并且作用域内没有名为 var 的类型，则称该迭代变量为隐式类型化迭代变量(implicitly typed iteration variable)，并假定其类型为 foreach 语句的元素类型，如下面所指定。迭代变量相当于一个其范围覆盖整个嵌入语句的只读局部变量。在 foreach 语句执行期间，迭代变量表示当前正在为其执行迭代的集合元素。如果嵌入语句试图修改迭代变量（通过赋值或 ++ 和 -- 运算符）或将迭代变量作为 ref 或 out 参数传递，则将发生编译时错误。

为简洁起见，下面用 IEnumerable、IEnumerator、IEnumerable<T> 和 IEnumerator<T> 指代命名空间 System.Collections 和 System.Collections.Generic 中相应的类型。

foreach 语句的编译时处理首先确定表达式的集合类型 (collection type)、枚举器类型 (enumerator type) 和元素类型 (element type)。此确定过程按如下进行：

- 如果 *expression* 的类型 X 是数组类型，则存在从 X 到 IEnumerable 接口（因为 System.Array 实现此接口）的隐式引用转换。***集合类型***为 IEnumerable 接口，***枚举器类型***为 IEnumerator 接口，而***元素类型***为数组类型 X 的元素类型。
- 如果 *expression* 的类型 X 为 dynamic，则存在从 *expression* 到 IEnumerable 接口（第 6.1.8 节）的隐式转换。***集合类型***是 IEnumerable 接口，***枚举器类型***是 IEnumerator 接口。如果以 *local-variable-type* 形式给定 var 标识符，则元素类型 (element type) 为 dynamic，否则，它为 object。
- 否则，确定类型 X 是否具有相应的 GetEnumerator 方法：
  - 在带有标识符 GetEnumerator 和不带类型参数的类型 X 上执行成员查找。如果成员查找没有产生匹配项，产生了多义性，或者产生了不是方法组的匹配项，请按如下所述检查可枚举的接口。建议在成员查找产生除方法组外的任何匹配项或没有产生匹配项的情况下发出警告。
  - 使用产生的方法组和空的参数列表执行重载决策。如果重载决策产生了不适用的方法、多义性或者单个最佳方法（但该方法是静态的或非公共的），请按如下所述检查可枚举的接口。建议在重载决策产生除无歧义的公共实例方法外的任何方法或没有产生适用方法的情况下发出警告。
  - 如果 GetEnumerator 方法的返回类型 E 不是类、结构或接口类型，则将产生错误，并且不再执行进一步的操作。
  - 在带有标识符 Current 和不带类型参数的 E 上执行成员查找。如果成员查找没有产生匹配项，结果是错误的或者是除允许读取的公共实例属性外的任何项，则将产生错误并且不再执行进一步的操作。
  - 在带有标识符 MoveNext 和不带类型参数的 E 上执行成员查找。如果成员查找没有产生匹配项，结果是错误的或者是除方法组外的任何项，则将产生错误并且不再执行进一步的操作。
  - 使用空的参数列表对方法组执行重载决策。如果重载决策产生了不适用的方法、多义性、单个最佳方法（但该方法是静态的或非公共的）或者其返回类型不是 bool，则将产生错误并且不再执行进一步的操作。
  - 集合类型 (collection type) 为 X，枚举器类型 (enumerator type) 为 E，而元素类型 (element type) 为 Current 属性的类型。
- 否则，检查可枚举的接口：
  - 如果在所有存在从 X 到 IEnumerable<T_i> 的隐式转换的类型 T_i 中有一个唯一类型 T，以使 T 不是 dynamic，并且对于所有其他 T_i，存在从 IEnumerable<T> 到 IEnumerable<T_i> 的隐式转

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

换，则集合类型为接口 `IEnumerable<T>`，枚举器类型为接口 `IEnumerator<T>`，而元素类型为 `T`。

- 否则，如果存在多个此种类型 `T`，则将产生错误并且不再执行进一步的操作。
- 否则，如果存在从 `X` 到 `System.Collections.IEnumerable` 接口的隐式转换，则*集合类型*为此接口，*枚举器类型*为接口 `System.Collections.IEnumerator`，*元素类型*为 `object`。
- 否则，将产生错误并且不再执行进一步的操作。

上述步骤如果成功，将无歧义地产生集合类型 `C`、枚举器类型 `E` 和元素类型 `T`。以下形式的 foreach 语句

```
foreach (V v in x) embedded-statement
```

然后扩展为：

```
{
   E e = ((C)(x)).GetEnumerator();
   try {
      while (e.MoveNext()) {
         V v = (V)(T)e.Current;
         embedded-statement
      }
   }
   finally {
      … // Dispose e
   }
}
```

变量 `e` 对表达式 `x` 或嵌入语句或该程序的任何其他源代码均不可见或不可访问。变量 `v` 在嵌入语句中是只读的。如果不存在从 `T` (元素类型)到 `V` (foreach 语句中的 `v` (the *local-variable-type*)的显式转换（第 6.2 节），则会出错且不会执行下面的步骤。如果 `x` 具有值 `null`，则将在运行时引发 `System.NullReferenceException`。

只要行为与上述扩展一致，便允许通过某个实现以不同方式来实现给定的 foreach 语句（如，由于性能原因）。

`v` 在 while 循环中的位置对它被 *embedded-statement* 中出现的任何匿名函数捕获的方式十分重要。

例如：

```
int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}

f();
```

如果 `v` 在 while 循环外声明，它将在所有迭代之间共享，其值在 for 循环结束后将成为最终值 `13`，该值是调用 `f` 时将输出的内容。然而，因为每次迭代都有自己的变量 `v`，在第一次迭代中由 `f` 捕获的该变量将继续保存值 `7`，该值是将输出的内容。（注意：早期版本的 C# 在 while 循环外声明 `v`。）

按照下列步骤构造 finally 块体：

- 如果存在从 `E` 到 `System.IDisposable` 接口的隐式转换，则
  - 如果 `E` 为不可以为 null 值的类型，则 finally 子句扩展为下面子句的语义等效项：

```
finally {
   ((System.IDisposable)e).Dispose();
}
```

- 否则 finally 子句扩展到下面子句的语义等效项：

```
finally {
   if (e != null) ((System.IDisposable)e).Dispose();
}
```

但如果 E 是值类型或实例化为值类型的类型形参，则从 e 到 System.IDisposable 的强制转换不会导致发生装箱。

- 否则，如果 E 是密封类型，finally 子句将扩展为一个空块：

```
finally {
}
```

- 否则，finally 子句将扩展为：

```
finally {
   System.IDisposable d = e as System.IDisposable;
   if (d != null) d.Dispose();
}
```

局部变量 d 对于任何用户代码均不可见或不可访问。尤其是，它不会与范围包括该 finally 块的其他任何变量发生冲突。

foreach 按如下顺序遍历数组的元素：对于一维数组，按递增的索引顺序遍历元素，从索引 0 开始，到索引 Length – 1 结束。对于多维数组，按这样的方式遍历元素：首先增加最右边维度的索引，然后是它的左边紧邻的维度，依此类推直到最左边的那个维度。

下列示例按照元素的顺序打印出一个二维数组中的各个元素的值：

```
using System;

class Test
{
   static void Main() {
      double[,] values = {
         {1.2, 2.3, 3.4, 4.5},
         {5.6, 6.7, 7.8, 8.9}
      };

      foreach (double elementValue in values)
         Console.Write("{0} ", elementValue);

      Console.WriteLine();
   }
}
```

所生成的输出如下：

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

在下面的示例中

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

n 的类型推断为 int，即 numbers 的元素类型。

8.9 跳转语句

跳转语句用于无条件地转移控制。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

jump-statement:
break-statement
continue-statement
goto-statement
return-statement
throw-statement

跳转语句将控制转到的位置称为跳转语句的目标 (target)。

当一个跳转语句出现在某个块内，而该跳转语句的目标在该块之外时，就称该跳转语句退出 (exit) 该块。虽然跳转语句可以将控制转到一个块外，但它永远不能将控制转到一个块的内部。

由于存在 try 语句的干扰，跳转语句的执行有时会变得复杂起来。如果没有这样的 try 语句，则跳转语句无条件地将控制从跳转语句转到它的目标。当跳转涉及到 try 语句时，执行就变得复杂一些了。如果跳转语句欲退出的是一个或多个具有相关联的 finally 块的 try 块，则控制最初转到最里层的 try 语句的 finally 块。当（如果）控制到达该 finally 块的结束点时，控制就转到下一个封闭 try 语句的 finally 块。此过程不断重复，直到执行完所有涉及的 try 语句的 finally 块。

在下面的示例中

```
using System;

class Test
{
   static void Main() {
      while (true) {
         try {
            try {
               Console.WriteLine("Before break");
               break;
            }
            finally {
               Console.WriteLine("Innermost finally block");
            }
         }
         finally {
            Console.WriteLine("Outermost finally block");
         }
      }
      Console.WriteLine("After break");
   }
}
```

在将控制转到跳转语句的目标之前，要先执行与两个 try 语句关联的 finally 块。

所生成的输出如下：

```
Before break
Innermost finally block
Outermost finally block
After break
```

8.9.1 break 语句

break 语句将退出直接封闭它的 switch、while、do、for 或 foreach 语句。

break-statement:
break ;

break 语句的目标是直接封闭它的 switch、while、do、for 或 foreach 语句的结束点。如果 break 语句不是由 switch、while、do、for 或 foreach 语句封闭的，则会发生编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

当多个 switch、while、do、for 或 foreach 语句互相嵌套时，break 语句将只应用于最里层的那个语句。若要穿越多个嵌套层转移控制，必须使用 goto 语句（第 8.9.3 节）。

break 语句不能退出 finally 块（第 8.10 节）。当 break 语句出现在 finally 块中时，该 break 语句的目标必须位于同一个 finally 块中，否则将发生编译时错误。

break 语句按如下规则执行：

- 如果 break 语句要退出的是一个或多个具有关联的 finally 块的 try 块，则控制最初将转到最里层的 try 语句的 finally 块。当（如果）控制到达该 finally 块的结束点时，控制就转到下一个封闭 try 语句的 finally 块。此过程不断重复，直到执行完所有涉及的 try 语句的 finally 块。
- 控制转到 break 语句的目标。

由于 break 语句无条件地将控制转到别处，因此永远无法到达 break 语句的结束点。

8.9.2 continue 语句

continue 语句将开始直接封闭它的 while、do、for 或 foreach 语句的一次新迭代。

continue-statement:
    continue ;

continue 语句的目标是直接封闭它的 while、do、for 或 foreach 语句的嵌入语句的结束点。如果 continue 语句不是由 while、do、for 或 foreach 语句封闭的，则会发生编译时错误。

当多个 while、do、for 或 foreach 语句互相嵌套时，continue 语句将只应用于最里层的那个语句。若要穿越多个嵌套层转移控制，必须使用 goto 语句（第 8.9.3 节）。

continue 语句不能退出 finally 块（第 8.10 节）。当 continue 语句出现在 finally 块中时，该 continue 语句的目标必须位于同一个 finally 块中，否则将发生编译时错误。

continue 语句按如下规则执行：

- 如果 continue 语句要退出的是一个或多个具有关联的 finally 块的 try 块，则控制最初将转到最里层的 try 语句的 finally 块。当（如果）控制到达该 finally 块的结束点时，控制就转到下一个封闭 try 语句的 finally 块。此过程不断重复，直到执行完所有涉及的 try 语句的 finally 块。
- 控制转到 continue 语句的目标。

由于 continue 语句无条件地将控制转到别处，因此永远无法到达 continue 语句的结束点。

8.9.3 goto 语句

goto 语句将控制转到由标签标记的语句。

goto-statement:
    goto *identifier* ;
    goto case *constant-expression* ;
    goto default ;

goto *identifier* 语句的目标是具有给定标签的标记语句。如果当前函数成员中不存在具有给定名称的标签，或者如果 goto 语句不在该标签的范围内，则发生编译时错误。此规则允许使用 goto 语句将控制转移“出”嵌套范围，但是不允许将控制转移“进”嵌套范围。在下面的示例中

```
using System;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static void Main(string[] args) {
      string[,] table = {
         {"Red", "Blue", "Green"},
         {"Monday", "Wednesday", "Friday"}
      };

      foreach (string str in args) {
         int row, colm;
         for (row = 0; row <= 1; ++row)
            for (colm = 0; colm <= 2; ++colm)
               if (str == table[row,colm])
                  goto done;

         Console.WriteLine("{0} not found", str);
         continue;
   done:
         Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
      }
   }
}
```

goto 语句用于将控制转移出嵌套范围。

goto case 语句的目标是直接封闭着它的 switch 语句（第 8.7.2 节）中的语句列表，该语句包含一个具有给定常量值的 case 标签。如果 goto case 语句不是由 switch 语句封闭的，或者 *constant-expression* 不能隐式转换（第 0 节）为直接封闭着它的 switch 语句的主导类型，或者直接封闭着它的 switch 语句不包含具有给定常量值的 case 标签，则发生编译时错误。

goto default 语句的目标是直接封闭着它的 switch 语句（第 8.7.2 节）中的语句列表，该语句包含一个 default 标签。如果 goto default 语句不是由 switch 语句封闭的，或者如果直接封闭着它的 switch 语句不包含 default 标签，则将发生编译时错误。

goto 语句不能退出 finally 块（第 8.10 节）。当 goto 语句出现在 finally 块中时，该 goto 语句的目标必须位于同一个 finally 块中，否则将发生编译时错误。

goto 语句按如下规则执行：

- 如果 goto 语句要退出的是一个或多个具有关联的 finally 块的 try 块，则控制最初将转到最里层的 try 语句的 finally 块。当（如果）控制到达该 finally 块的结束点时，控制就转到下一个封闭 try 语句的 finally 块。此过程不断重复，直到执行完所有涉及的 try 语句的 finally 块。
- 控制转到 goto 语句的目标。

由于 goto 语句无条件地将控制转到别处，因此永远无法到达 goto 语句的结束点。

8.9.4 return 语句

return 语句会将控制返回到出现 return 语句的函数的当前调用方。

return-statement:
    return *expression*$_{opt}$;

不带表达式的 return 语句只能用在不计算值的函数成员中，即只能用在结果类型（第 10.6.10 节）为 void 的方法、属性或索引器的 set 访问器、事件的 add 和 remove 访问器、实例构造函数、静态构造函数或析构函数中。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

带表达式的 return 语句只能用在计算值的函数成员中，即结果类型为非 void 的方法、属性或索引器的 get 访问器或用户定义的运算符。必须存在一个隐式转换（第 0 节），它能将该表达式的类型转换到包含它的函数成员的返回类型。

Return 语句也可以用在匿名函数表达式（第 7.15 节）的主体中，并参与决定这些函数存在哪些转换。

return 语句出现在 finally 块（第 8.10 节）中是编译时错误。

return 语句按如下规则执行：

- 如果 return 语句指定一个表达式，则计算该表达式，并将结果隐式转换为包含它的函数成员的返回类型。转换的结果将成为函数所生成的结果值。
- 如果 return 语句由一个或多个具有关联的 finally 块的 try 或 catch 块封闭，则控制最初将转到最里层的 try 语句的 finally 块。当（如果）控制到达该 finally 块的结束点时，控制就转到下一个封闭 try 语句的 finally 块。此过程不断重复，直到执行完所有封闭的 try 语句的 finally 块。
- 如果包含函数不是异步函数，则控制将与结果值（如果有）一起返回给包含函数的调用方。
- 如果包含函数是异步函数，则控制将返回给当前调用方，而结果值（如果有）将记录在返回任务中，如第 **错误!未找到引用源。** 节所述。

由于 return 语句无条件地将控制转到别处，因此永远无法到达 return 语句的结束点。

8.9.5 throw 语句

throw 语句将引发一个异常。

throw-statement:
    throw *expression*$_{opt}$;

带表达式的 throw 语句引发一个异常，此异常的值就是通过计算该表达式而产生的值。该表达式必须表示类类型 System.Exception 的值、从 System.Exception 派生的类类型的值，或者其有效基类从 System.Exception 派生的类型形参的值。如果表达式的计算产生 null，则将改为引发 System.NullReferenceException。

不带表达式的 throw 语句只能用在 catch 块中，在这种情况下，该语句重新引发当前正由该 catch 块处理的那个异常。

由于 throw 语句无条件地将控制转到别处，因此永远无法到达 throw 语句的结束点。

引发一个异常 E 时，控制转到封闭 try 语句中能够处理该异常的第一个 catch 子句。从引发一个异常开始直至将控制转到关于该异常的一个合适的异常处理程序止，这个过程称为异常传播 (exception propagation)。“传播一个异常”由重复地执行下列各步骤组成，直至找到一个与该异常匹配的 catch 子句。在此描述中，引发点 (throw point) 最初是指引发该异常的位置。

- 在当前函数成员中，检查每个封闭着引发点的 try 语句。对于每个语句 S（按从最里层的 try 语句开始，逐次向外，直到最外层的 try 语句结束），计算下列步骤：
    - 如果 S 的 try 块封闭着引发点，并且如果 S 具有一个或多个 catch 子句，则按其出现的顺序检查这些 catch 子句以找到合适的异常处理程序。指定异常类型 T（或运行时表示异常类型 T 的类型参数）的第一个 catch 子句，以便将派生自 T 的 E 的运行时类型视为匹配项。常规 catch 子句（第 8.10 节）被认为是任何异常类型的匹配项。如果找到匹配的 catch 子句，则通过将控制转到该 catch 子句的块来完成异常传播。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，如果 S 的 try 块或 catch 块封闭着引发点并且如果 S 具有 finally 块，则控制将转到 finally 块。如果在该 finally 块内引发另一个异常，则终止当前异常的处理。否则，当控制到达 finally 块的结束点时，将继续对当前异常的处理。

- 如果在当前函数调用中没有找到异常处理程序，则将终止该函数调用并发生以下情况之一：
  - 如果当前函数是非异步的，则使用与调用函数成员的语句对应的引发点，为该函数的调用方重复执行上面的步骤。
  - 如果当前函数是异步的并且返回任务，则将异常记录在返回任务中，并将该任务置于“出错”或“取消”状态，如第 **错误!未找到引用源。** 节所述。
  - 如果当前函数是异步的并且返回 void，则将通知当前线程的同步上下文，如第 **错误!未找到引用源。** 节所述。
- 如果上述异常处理终止了当前线程中的所有函数成员调用，这表明此线程没有该异常的处理程序，那么线程本身将终止。此类终止会产生什么影响，应由实现来定义。

8.10 try 语句

try 语句提供一种机制，用于捕捉在块的执行期间发生的各种异常。此外，try 语句还能让您指定一个代码块，并保证当控制离开 try 语句时，总是先执行该代码。

try-statement:
    try *block* *catch-clauses*
    try *block* *finally-clause*
    try *block* *catch-clauses* *finally-clause*

catch-clauses:
    specific-catch-clauses *general-catch-clause*$_{opt}$
    specific-catch-clauses$_{opt}$ *general-catch-clause*

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses *specific-catch-clause*

specific-catch-clause:
    catch (*type* *identifier*$_{opt}$) *block*

general-catch-clause:
    catch *block*

finally-clause:
    finally *block*

有三种可能的 try 语句形式：

- 一个 try 块后接一个或多个 catch 块。
- 一个 try 块后接一个 finally 块。
- 一个 try 块后接一个或多个 catch 块，后面再跟一个 finally 块。

当 catch 子句指定 *type* 时，该类型必须为 System.Exception、从 System.Exception 派生的类型，或者其有效基类派生自 System.Exception 的类型形参类型。

当 catch 子句同时指定 *class-type* 和 *identifier* 时，相当于声明了一个具有给定名称和类型的异常变量 (exception variable)。此异常变量相当于一个范围覆盖整个 catch 块的局部变量。在 catch 块的执行期

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

间，此异常变量表示当前正在处理的异常。出于明确赋值检查的目的，此异常变量被认为在它的整个范围内是明确赋值的。

除非 catch 子句包含一个异常变量名，否则在该 catch 块中就不可能访问当前发生的异常对象。

既不指定异常类型也不指定异常变量名的 catch 子句称为常规 catch 子句。try 语句只能有一个常规 catch 子句，而且如果存在，它必须是最后一个 catch 子句。

有些编程语言可能支持一些异常，它们不能表示为从 System.Exception 派生的对象，尽管 C# 代码可能永远不会产生这类异常。可以使用常规 catch 子句来捕捉这类异常。因此，常规的 catch 子句在语义上不同于指定了 System.Exception 类型的那些子句，因为前者还可以捕获来自其他语言的异常。

为了找到当前发生了的异常的处理程序，catch 子句是按其词法顺序进行检查的。如果 catch 子句指定的类型与同一 try 块的某个较早的 catch 子句中所指定的类型相同，或者是从该类型派生的类型，则发生编译时错误。如果没有这个限制，就可能写出不可到达的 catch 子句。

在 catch 块内，不含表达式的 throw 语句（第 8.9.5 节）可用于重新引发由该 catch 块捕捉到的异常。对异常变量的赋值不会改变上述被重新引发的异常。

在下面的示例中

```
using System;

class Test
{
   static void F() {
      try {
         G();
      }
      catch (Exception e) {
         Console.WriteLine("Exception in F: " + e.Message);
         e = new Exception("F");
         throw;            // re-throw
      }
   }

   static void G() {
      throw new Exception("G");
   }

   static void Main() {
      try {
         F();
      }
      catch (Exception e) {
         Console.WriteLine("Exception in Main: " + e.Message);
      }
   }
}
```

方法 F 捕捉到一个异常，向控制台写入一些诊断信息，更改异常变量，然后重新引发该异常。重新引发的异常是原来那个被捕获的异常，因此产生的输出为：

```
Exception in F: G
Exception in Main: G
```

如果第一个 catch 块引发了异常 e 而不是重新引发当前的异常，产生的输出就会如下所示：

```
Exception in F: G
Exception in Main: F
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

break、continue 或 goto 语句将控制转到 finally 块外部时将导致编译时错误。当一个 break、continue 或 goto 语句出现在 finally 块中时，该语句的目标必须在同一 finally 块内，否则会发生编译时错误。

return 语句出现在 finally 块中是编译时错误。

try 语句按如下规则执行：

- 控制转到 try 块。
- 当（如果）控制到达 try 块的结束点时：
  - 如果 try 语句具有 finally 块，则将执行 finally 块。
  - 控制转到 try 语句的结束点。
- 如果在 try 块执行期间有一个异常传播到 try 语句：
  - 按 catch 子句出现的顺序（如果有）逐个对其进行检查，以找到一个合适的异常处理程序。第一个指定了异常类型或该异常类型的基类型的 catch 子句被认为是一个匹配项。常规 catch 子句被认为是任何异常类型的匹配项。如果找到匹配的 catch 子句：
    - 如果匹配的 catch 子句声明一个异常变量，则异常对象被赋给该异常变量。
    - 控制转到匹配 catch 块。
    - 当（如果）控制到达 catch 块的结束点时：
      - 如果 try 语句具有 finally 块，则将执行 finally 块。
      - 控制转到 try 语句的结束点。
    - 如果在 try 块执行期间有一个异常传播到 catch 语句：
      - 如果 try 语句具有 finally 块，则将执行 finally 块。
      - 该异常就传播到更外面一层（封闭）的 try 语句。
  - 如果 try 语句没有 catch 子句或者如果没有与异常匹配的 catch 子句：
    - 如果 try 语句具有 finally 块，则将执行 finally 块。
    - 该异常就传播到更外面一层（封闭）的 try 语句。

finally 块中的语句始终在控制离开 try 语句时执行。无论是什么原因引起控制转移（正常执行到达结束点，执行了 break、continue、goto 或 return 语句，或是将异常传播到 try 语句之外），情况都是如此。

如果在执行 finally 块期间引发了一个异常，而且该异常不是在同一个 finally 块中捕获的，则该异常将被传播到下一个封闭的 try 语句。与此同时，原先那个正在传播过程中的异常（如果存在）就会被丢弃。关于传播异常的过程，在 throw 语句（第 8.9.5 节）的说明中有进一步讨论。

如果 try 语句是可到达的，则 try 语句的 try 块也是可到达的。

如果 try 语句是可到达的，则 try 语句的 catch 块也是可到达的。

如果 try 语句是可到达的，则 try 语句的 finally 块也是可到达的。

如果下列两个条件都为真，则 try 语句的结束点是可到达的：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- try 块的结束点是可到达的或者至少一个 catch 块的结束点是可到达的。
- 如果存在一个 finally 块，则此 finally 块的结束点是可到达的。

8.11 checked 语句和 unchecked 语句

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

checked-statement:
    checked *block*

unchecked-statement:
    unchecked *block*

checked 语句使 *block* 中的所有表达式都在一个选中的上下文中进行计算，而 unchecked 语句使 *block* 中的所有表达式都在一个未选中的上下文中进行计算。

checked 语句和 unchecked 语句完全等效于 checked 和 unchecked 运算符（第 7.6.12 节），不同的是它们作用于块，而不是作用于表达式。

8.12 lock 语句

lock 语句用于获取某个给定对象的互斥锁，执行一个语句，然后释放该锁。

lock-statement:
    lock (*expression*) *embedded-statement*

lock 语句的表达式必须表示一个已知的 *reference-type* 类型的值。永远不会为 lock 语句中的表达式执行隐式装箱转换（第 6.1.7 节），因此，如果该表达式表示的是一个 *value-type* 的值，则会导致一个编译时错误。

下列形式的 lock 语句

```
lock (x) ...
```

（其中 x 是一个 *reference-type* 的表达式）完全等效于

```
bool __lockWasTaken = false;
try {
   System.Threading.Monitor.Enter(x, ref __lockWasTaken);
   ...
}
finally {
   if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

不同的只是：实际执行中 x 只计算一次。

当一个互斥锁已被占用时，在同一线程中执行的代码仍可以获取和释放该锁。但是，在其他线程中执行的代码在该锁被释放前是无法获得它的。

建议不要使用锁定 System.Type 对象的方法来同步对静态数据的访问。其他代码可能会在同一类型上进行锁定，这会导致死锁。更好的方法是通过锁定私有静态对象来同步对静态数据的访问。例如：

```
class Cache
{
   private static readonly object synchronizationObject = new object();
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public static void Add(object x) {
            lock (Cache.synchronizationObject) {
                ...
            }
        }
        public static void Remove(object x) {
            lock (Cache.synchronizationObject) {
                ...
            }
        }
    }
```

8.13 using 语句

using 语句获取一个或多个资源，执行一个语句，然后释放该资源。

using-statement:
    using (*resource-acquisition*) *embedded-statement*

resource-acquisition:
    local-variable-declaration
    expression

一个资源 (resource) 是实现了 System.IDisposable 的类或结构，它只包含一个名为 Dispose 的不带形参的方法。正在使用资源的代码可以调用 Dispose 以表明不再需要该资源。如果不调用 Dispose，则最终将因为垃圾回收而对该资源进行自动释放。

如果 *resource-acquisition* 的形式是 *local-variable-declaration*，则 *local-variable-declaration* 的类型必须为 dynamic 或是可以隐式转换为 System.IDisposable 的类型。如果 *resource-acquisition* 的形式是 *expression*，则表达式必须可以隐式转换为 System.IDisposable。

在 *resource-acquisition* 中声明的局部变量是只读的，且必须包含一个初始值设定项。如果嵌入语句试图修改这些局部变量（通过赋值或 ++ 和 -- 运算符），获取它们的地址或将它们作为 ref 或 out 形参传递，则将发生编译时错误。

using 语句转换为三部分：获取、使用和释放。资源的使用部分被隐式封闭在一个含有 finally 子句的 try 语句中。此 finally 子句用于释放资源。如果所获取资源是 null，则不会对 Dispose 进行调用，也不会引发任何异常。如果资源类型为 dynamic，则会在获取期间通过隐式动态转换（第 6.1.8 节）的方式动态转换为 IDisposable，以确保在使用和释放之前转换成功。

下列形式的 using 语句

```
using (ResourceType resource = expression) statement
```

对应于下列三个可能的扩展中的一个。当 ResourceType 是不可以为 null 的值类型，扩展为

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}
```

否则，当 ResourceType 是 dynamic 之外可以为 null 的值类型时，扩展为

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      IDisposable d = (IDisposable)resource;
      if (resource != null) d.Dispose();
   }
}
```

否则，当 ResourceType 为 dynamic 时，扩展为

```
{
   ResourceType resource = expression;
   IDisposable d = resource;
   try {
      statement;
   }
   finally {
      if (d != null) d.Dispose();
   }
}
```

在上面任何一种扩展中，resource 变量在嵌入语句中都是只读的，d 变量在嵌入语句中既不可访问，也不可见。

由于性能或其他方面原因，只要该行为与上面的扩展一致，就可以以不同方式实现给定 using 语句。

下列形式的 using 语句

```
using (expression) statement
```

具有三种相同的可用扩展。在这种情况下，ResourceType 隐含地为 expression 编译时类型（如果有该类型）。否则，使用接口 IDisposable 自身作为 ResourceType。在嵌入语句中不可访问 resource 变量，并且该变量对嵌入语句不可见。

如果 *resource-acquisition* 采用 *local-variable-declaration* 的形式，则有可能获取给定类型的多个资源。下列形式的 using 语句

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

完全等效于嵌套 using 语句的序列：

```
using (ResourceType r1 = e1)
   using (ResourceType r2 = e2)
      ...
         using (ResourceType rN = eN)
            statement
```

下面的示例创建一个名为 log.txt 的文件并将两行文本写入该文件。然后该示例打开这个文件进行读取，并将它所包含的文本行复制到控制台。

```
using System;
using System.IO;

class Test
{
   static void Main() {
      using (TextWriter w = File.CreateText("log.txt")) {
         w.WriteLine("This is line one");
         w.WriteLine("This is line two");
      }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
            using (TextReader r = File.OpenText("log.txt")) {
               string s;
               while ((s = r.ReadLine()) != null) {
                  Console.WriteLine(s);
               }
            }
         }
      }
```

由于 TextWriter 和 TextReader 类实现了 IDisposable 接口，因此该示例可以使用 using 语句以确保所涉及的文件在写入或读取操作后正确关闭。

8.14 yield 语句

yield 语句用在迭代器块中（第 8.2 节），作用是向迭代器的枚举器对象（第 10.14.4 节）或可枚举对象（第 10.14.5 节）产生一个值，或者通知迭代结束。

yield-statement:
    yield return *expression* ;
    yield break ;

yield 不是保留字；它仅在紧靠 return 或 break 关键字之前使用时才具有特殊意义。在其他上下文中，yield 可用作标识符。

yield 语句可出现的位置存在几个限制，如下所述。

- 如果 yield 语句（包括两种形式）出现在 *method-body*、*operator-body* 或 *accessor-body* 之外，则会引起编译时错误。
- 如果 yield 语句（包括两种形式）出现在匿名函数内部，则会引起编译时错误。
- yield 语句（两种形式中的任一形式）出现在 try 语句的 finally 子句中时，会导致编译时错误。
- yield return 语句出现在包含任何 catch 子句的 try 语句内的任何位置时，会导致编译时错误。

下面的示例演示 yield 语句的有效用法和无效用法。

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
   try {
      yield return 1;      // Ok
      yield break;         // Ok
   }
   finally {
      yield return 2;      // Error, yield in finally
      yield break;         // Error, yield in finally
   }

   try {
      yield return 3;      // Error, yield return in try...catch
      yield break;         // Ok
   }
   catch {
      yield return 4;      // Error, yield return in try...catch
      yield break;         // Ok
   }

   D d = delegate {
      yield return 5;      // Error, yield in an anonymous function
   };
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
int MyMethod() {
   yield return 1;          // Error, wrong return type for an iterator block
}
```

yield return 语句中的表达式的类型必须能够隐式转换（第6.1 节）为迭代器的产生类型（第 10.14.3 节）。

yield return 语句按如下规则执行：

- 计算该语句中给出的表达式，隐式转换为产生类型，并赋给枚举器对象的 Current 属性。
- 迭代器块的执行被挂起。如果 yield return 语句在一个或多个 try 块内，则此时与之关联的 finally 块将*不*会执行。
- 枚举器对象的 MoveNext 方法向其调用方返回 true，指示枚举器对象成功前进到下一项。

下次调用枚举器对象的 MoveNext 方法时将从上次挂起的地方恢复迭代器块的执行。

yield break 语句按如下规则执行：

- 如果 yield break 语句由一个或多个具有关联的 finally 块的 try 块封闭，则控制最初将转到最里层的 try 语句的 finally 块。当（如果）控制到达该 finally 块的结束点时，控制就转到下一个封闭 try 语句的 finally 块。此过程不断重复，直到执行完所有封闭的 try 语句的 finally 块。
- 控制返回给迭代器块的调用方。这是枚举器对象的 MoveNext 方法或 Dispose 方法。

由于 yield break 语句无条件地将控制转到别处，因此永远无法到达 yield break 语句的结束点。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

9. 命名空间

C# 程序是利用命名空间组织起来的。命名空间既用作程序的“内部”组织系统，也用作“外部”组织系统（一种向其他程序公开自己拥有的程序元素的方法）。

using 指令（第 9.4 节）用来为命名空间的使用提供方便。

9.1 编译单元

compilation-unit 定义了源文件的总体结构。编译单元的组成方式如下：先是零个或多个 *using-directive*，后接零个或多个 *global-attributes*，然后是零个或多个 *namespace-member-declaration*。

compilation-unit:
extern-alias-directives$_{opt}$ *using-directives*$_{opt}$ *global-attributes*$_{opt}$
namespace-member-declarations$_{opt}$

一个 C# 程序由一个或多个编译单元组成，每个编译单元都用一个单独的源文件来保存。编译 C# 程序时，所有这些编译单元一起进行处理。因此，这些编译单元间可以互相依赖，甚至以循环方式互相依赖。

编译单元的 *using-directives* 影响该编译单元内的 *global-attributes* 和 *namespace-member-declarations*，但是不会影响其他编译单元。

编译单元的 *global-attributes*（第 17 章）允许指定目标程序集和模块的特性。程序集和模块充当类型的物理容器。程序集可以包含若干个在物理上分离的模块。

程序中各编译单元中的 *namespace-member-declarations* 用于为一个称为“全局命名空间”的单个声明空间提供成员。例如：

文件 A.cs：

```
class A {}
```

文件 B.cs：

```
class B {}
```

这两个编译单元是为该全局命名空间提供成员的，在本例中它们分别声明了具有完全限定名 A 和 B 的两个类。由于这两个编译单元为同一声明空间提供成员，因此如果它们分别包含了一个同名成员的声明，将会是个错误。

9.2 命名空间声明

namespace-declaration 的组成方式如下：先是关键字 namespace，后接一个命名空间名称和体，然后加一个分号（可选）。

namespace-declaration:
namespace *qualified-identifier* *namespace-body* ;$_{opt}$

qualified-identifier:
identifier
qualified-identifier . *identifier*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

namespace-body:
{ *extern-alias-directives*$_{opt}$ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ }

namespace-declaration 可以作为顶级声明出现在 *compilation-unit* 中，或是作为成员声明出现在另一个 *namespace-declaration* 内。当 *namespace-declaration* 作为顶级声明出现在 *compilation-unit* 中时，该命名空间即成为全局命名空间的一个成员。当一个 *namespace-declaration* 出现在另一个 *namespace-declaration* 内时，该内部命名空间就成为包含着它的外部命名空间的一个成员。无论是何种情况，一个命名空间的名称在它所属的命名空间内必须是唯一的。

命名空间隐式地为 `public`，而且在命名空间的声明中不能包含任何访问修饰符。

在 *namespace-body* 内，可选用 *using-directives* 来导入其他命名空间和类型的名称，这样，就可以直接地而不是通过限定名来引用它们。可选的 *namespace-member-declarations* 用于为命名空间的声明空间提供成员。请注意，所有的 *using-directives* 都必须出现在任何成员声明之前。

namespace-declaration 中的 *qualified-identifier* 可以是单个标识符，也可以是由“.”标记分隔的标识符序列。后一种形式允许一个程序直接定义一个嵌套命名空间，而不必按词法嵌套若干个命名空间声明。例如，

```
namespace N1.N2
{
   class A {}
   class B {}
}
```

在语义上等效于

```
namespace N1
{
   namespace N2
   {
      class A {}
      class B {}
   }
}
```

命名空间是可扩充的，两个具有相同的完全限定名的命名空间声明是在为同一声明空间（第 3.3 节）提供成员。在下面的示例中

```
namespace N1.N2
{
   class A {}
}
namespace N1.N2
{
   class B {}
}
```

上面的两个命名空间声明为同一声明空间提供了成员，在本例中它们分别声明了具有完全限定名 `N1.N2.A` 和 `N1.N2.B` 的两个类。由于两个声明为同一声明空间提供成员，因此如果它们分别包含一个同名成员的声明，就将出现错误。

9.3 Extern 别名

extern-alias-directive 引入了一个作为命名空间别名的标识符。对已有别名的命名空间的指定是在程序的源代码外部进行的，这种指定也应用于该已有别名的命名空间的嵌套命名空间。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

extern-alias-directives:
extern-alias-directive
extern-alias-directives *extern-alias-directive*

extern-alias-directive:
`extern alias` *identifier* `;`

extern-alias-directive 范围定义的名称范围扩展到直接包含其的编译单元或命名空间体的 *using-directives*、*global-attributes* 和 *namespace-member-declarations*。

在包含 *extern-alias-directive* 的编译单元或命名空间体内，由 *extern-alias-directive* 引入的标识符可用于引用已有别名的命名空间。如果该 *identifier* 为单词 `global`，则会发生编译时错误。

extern-alias-directive 使别名可用在特定编译单元或命名空间体内，但是它不会向基础声明空间提供任何新成员。换言之，*extern-alias-directive* 不具传递性，它仅影响它在其中出现的编译单元或命名空间体。

下面的程序声明并使用两个外部别名（`X` 和 `Y`），每个别名都代表不同命名空间层次结构的根：

```
extern alias X;
extern alias Y;

class Test
{
   X::N.A a;
   X::N.B b1;
   Y::N.B b2;
   Y::N.C c;
}
```

该程序声明存在 extern 别名 `X` 和 `Y`，但这些别名的实际定义在该程序的外部。同名的 `N.B` 类现在可分别通过 `X.N.B` 和 `Y.N.B` 引用，或者使用命名空间别名限定符通过 `X::N.B` 和 `Y::N.B` 引用。如果没有为程序声明的 extern 别名提供外部定义，则会发生错误。

9.4 using 指令

Using 指令 (using directives) 方便了对在其他命名空间中定义的命名空间和类型的使用。using 指令影响 *namespace-or-type-name*（第 3.8 节）和 *simple-name*（第 7.6.2 节）的名称解析过程，与声明不同，using 指令不会向在其中使用它们的编译单元或命名空间的基础声明空间中提供新成员。

using-directives:
using-directive
using-directives *using-directive*

using-directive:
using-alias-directive
using-namespace-directive

using-alias-directive（第 9.4.1 节）用于为一个命名空间或类型引入一个别名。

using-namespace-directive（第 9.4.2 节）用于导入一个命名空间的类型成员。

using-directive 的范围扩展到直接包含它的编译单元或命名空间体内的所有 *namespace-member-declarations*。具体而言，*using-directive* 的范围不包括与它对等的 *using-directive*。因此，对等 *using-directive* 互不影响，而且按什么顺序编写它们也无关紧要。

9.4.1 using 别名指令

using-alias-directive 为一个命名空间或类型（在直接包容该指令的编译单元或命名空间体内）引入用作别名的标识符。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

using-alias-directive:
    `using` *identifier* `=` *namespace-or-type-name* `;`

在包含 *using-alias-directive* 的编译单元或命名空间体内的成员声明中，由 *using-alias-directive* 引入的标识符可用于引用给定的命名空间或类型。例如：

```
namespace N1.N2
{
   class A {}
}

namespace N3
{
   using A = N1.N2.A;

   class B: A {}
}
```

上面的示例中，在 N3 命名空间中的声明成员内，A 是 N1.N2.A 的别名，因此类 N3.B 从类 N1.N2.A 派生。通过为 N1.N2 创建别名 R，然后再引用 R.A，可以获得相同效果：

```
namespace N3
{
   using R = N1.N2;

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

using-alias-directive 中的 *identifier* 在直接包含该 *using-alias-directive* 的编译单元或命名空间的声明空间内必须是唯一的。例如：

```
namespace N3
{
   class A {}
}

namespace N3
{
   using A = N1.N2.A;      // Error, A already exists
}
```

上例中，NN33 已包含了成员 A，因此 *using-alias-directive* 使用 A 作为标识符会导致一个编译时错误。同样，如果同一个编译单元或命名空间体中的两个或更多 *using-alias-directive* 用相同名称声明别名，也会导致一个编译时错误。

using-alias-directive 使别名可用在特定编译单元或命名空间体内，但是它不会向基础声明空间提供任何新成员。换句话说，*using-alias-directive* 不具传递性，它仅影响它在其中出现的编译单元或命名空间体。在下面的示例中

```
namespace N3
{
   using R = N1.N2;
}

namespace N3
{
   class B: R.A {}         // Error, R unknown
}
```

引入 R 的 *using-alias-directive* 的范围只扩展到包含它的命名空间体中的成员声明，因此 R 在第二个命名空间声明中是未知的。但是，如果将 *using-alias-directive* 放置在包含它的编译单元中，则该别名在两个命名空间声明中都将可用：

```
using R = N1.N2;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
namespace N3
{
   class B: R.A {}
}
namespace N3
{
   class C: R.A {}
}
```

和常规成员一样，*using-alias-directive* 引入的名称在嵌套范围中也可被具有相似名称的成员所隐藏。在下面的示例中

```
using R = N1.N2;
namespace N3
{
   class R {}
   class B: R.A {}      // Error, R has no member A
}
```

B 的声明中对 R.A 的引用将导致编译时错误，原因是这里的 R 所引用的是 N3.R 而不是 N1.N2。

编写 *using-alias-directive* 的顺序并不重要，对由 *using-alias-directive* 引用的 *namespace-or-type-name* 的解析过程既不受 *using-alias-directive* 本身影响，也不受直接包含着该指令的编译单元或命名空间体中的其他 *using-directive* 影响。换句话说，对 *using-alias-directive* 的 *namespace-or-type-name* 的解析，就如同在直接包含该指令的编译单元或命名空间体中根本没有 *using-directive* 一样来处理。但是，*using-alias-directive* 可能会受直接包含该指令的编译单元或命名空间体中的 *extern-alias-directive* 影响。在下面的示例中

```
namespace N1.N2 {}
namespace N3
{
   extern alias E;
   using R1 = E.N;      // OK
   using R2 = N1;       // OK
   using R3 = N1.N2;    // OK
   using R4 = R2.N2;    // Error, R2 unknown
}
```

最后一个 *using-alias-directive* 导致编译时错误，原因是它不受第一个 *using-alias-directive* 影响。第一个 *using-alias-directive* 不会产生错误，因为 extern 别名 E 的范围包括 *using-alias-directive*。

using-alias-directive 可以为任何命名空间或类型创建别名，包括它在其中出现的命名空间本身，以及嵌套在该命名空间中的其他任何命名空间或类型。

对一个命名空间或类型进行访问时，无论用它的别名，还是用它的所声明的名称，结果是完全相同的。例如，给定

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using R1 = N1;
   using R2 = N1.N2;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   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 是等效的，它们都引用完全限定名为 N1.N2.A 的类。

using 别名可以命名封闭构造类型，但是不能命名未提供类型实参的未绑定泛型类型声明。例如：

```
namespace N1
{
   class A<T>
   {
      class B {}
   }
}

namespace N2
{
   using W = N1.A;          // Error, cannot name unbound generic type
   using X = N1.A.B;        // Error, cannot name unbound generic type
   using Y = N1.A<int>;     // Ok, can name closed constructed type
   using Z<T> = N1.A<T>;    // Error, using alias cannot have type parameters
}
```

9.4.2 Using 命名空间指令

using-namespace-directive 将一个命名空间中所包含的类型导入到直接包容该指令的编译单元或命名空间体中，从而可以直接使用每个被导入的类型的标识符而不必加上它们的限定名。

using-namespace-directive:
   using *namespace-name* ;

在包含 *using-namespace-directive* 的编译单元或命名空间体中的成员声明内，可以直接引用包含在给定命名空间中的那些类型。例如：

```
namespace N1.N2
{
   class A {}
}

namespace N3
{
   using N1.N2;

   class B: A {}
}
```

上面的示例中，在 N3 命名空间中的成员声明内，N1.N2 的类型成员是直接可用的，所以类 N3.B 从类 N1.N2.A 派生。

using-namespace-directive 导入包含在给定命名空间中的类型，但要注意，它不导入嵌套的命名空间。在下面的示例中

```
namespace N1.N2
{
   class A {}
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
namespace N3
{
   using N1;
   class B: N2.A {}     // Error, N2 unknown
}
```

using-namespace-directive 导入包含在 N1 中的类型，但是不导入嵌套在 N1 中的命名空间。因此，在 B 的声明中引用 N2.A 导致编译时错误，原因是在涉及的范围内没有名为 N2 的成员。

与 *using-alias-directive* 不同，*using-namespace-directive* 可能导入一些特定类型，它们的标识符已在包容编译单元或命名空间体中定义。事实上，*using-namespace-directive* 导入的名称会被包容编译单元或命名空间体中具有类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 {}
}
```

此处，在 N3 命名空间中的成员声明内，A 引用 N3.A 而不是 N1.N2.A。

当由同一编译单元或命名空间体中的 *using-namespace-directive* 导入多个命名空间时，如果它们所包含的类型中有重名的，则对该名称的引用被认为是歧义。在下面的示例中

```
namespace N1
{
   class A {}
}
namespace N2
{
   class A {}
}
namespace N3
{
   using N1;
   using N2;
   class B: A {}           // Error, A is ambiguous
}
```

N1 和 N2 都包含一个成员 A，而由于 N3 将两者都导入，所以在 N3 中引用 A 会导致一个编译时错误。在这种情况下，可通过两种办法解决冲突：限定对 A 的引用；引入一个选取特定 A 的 *using-alias-directive*。例如：

```
namespace N3
{
   using N1;
   using N2;
   using A = N1.A;
   class B: A {}           // A means N1.A
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

同 *using-alias-directive* 一样，*using-namespace-directive* 不会向编译单元或命名空间的基础声明空间提供任何新成员，因而，它仅影响它出现在其中的编译单元或者命名空间体。

对 *using-namespace-directive* 所引用的 *namespace-name* 的解析方式，与对 *using-alias-directive* 所引用的 *namespace-or-type-name* 的解析方式相同。因此，同一编译单元或命名空间体中的 *using-namespace-directive* 互不影响，而且可以按照任何顺序编写。

9.5 命名空间成员

namespace-member-declaration 或是一个 *namespace-declaration*（第 9.2 节），或是一个 *type-declaration*（第 9.6 节）。

namespace-member-declarations:
namespace-member-declaration
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
namespace-declaration
type-declaration

编译单元或命名空间体可以包含 *namespace-member-declarations*，而此类声明则为与包含它们的编译单元或命名空间体的基础声明空间提供新成员。

9.6 类型声明

type-declaration 是 *class-declaration*（第 10.1 节）、*struct-declaration*（第 11.1 节）、*interface-declaration*（第 13.1 节）、*enum-declaration*（第 14.1 节）或 *delegate-declaration*（第 15.1 节）。

type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration

type-declaration 可以作为顶级声明出现在编译单元中，或者作为成员声明出现在命名空间、类或结构内部。

当类型 `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限定名正好是 `T`。当类型 `T` 的类型声明出现在命名空间、类或结构内时，新声明的类型的完全限定名是 `N.T`，其中 `N` 是包含它的命名空间、类或结构的完全限定名。

在类或结构内声明的类型称为嵌套类型（第 10.3.8 节）。

在一个类型声明中允许使用哪些访问修饰符以及具有何种默认访问属性，取决于进行了该声明的上下文（第 3.5.1 节）：

- 在编译单元或命名空间中声明的类型可以具有 `public` 或 `internal` 访问属性。默认为 `internal` 访问权限。
- 在类中声明的类型可以具有 `public`、`protected internal`、`protected`、`internal` 或 `private` 访问权限。默认为 `private` 访问权限。
- 在结构中声明的类型可以具有 `public`、`internal` 或 `private` 访问权限。默认为 `private` 访问权限。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

9.7 命名空间别名限定符

命名空间别名限定符 ***namespace alias qualifier*** `::` 让类型名称的查找不受引入的新类型和新成员的影响成为可能。命名空间别名限定符总是出现在两个标识符之间，这两个标识符分别称为左标识符和右标识符。与普通的 `.` 标识符不同，`::` 限定符的左标识符仅作为 extern 或 using 别名进行查找。

qualified-alias-member 定义如下：

> *qualified-alias-member:*
> *identifier* `::` *identifier* *type-argument-list*$_{opt}$

qualified-alias-member 可用作 *namespace-or-type-name*（第 3.8 节）或用作 *member-access*（第 7.6.4 节）中的左操作数。

qualified-alias-member 具有下列两种形式之一：

- `N::I<`A_1`, ...,`A_K`>`，其中 `N` 和 `I` 表示标识符，`<`A_1`, ..., `A_K`>` 是类型实参列表。（`K` 始终至少为 1。）
- `N::I,`，其中 `N` 和 `I` 表示标识符。（在此情况下，`K` 视作 0。）

如果使用此表示法，*qualified-alias-member* 的含义按下面的过程确定：

- 如果 `N` 为标识符 `global`，则在全局命名空间中搜索 `I`：
  - 如果全局命名空间包含名为 `I` 的命名空间，并且 `K` 为 0，则 *qualified-alias-member* 即表示该命名空间。
  - 否则，如果该全局命名空间包含名为 `I` 的非泛型类型，并且 `K` 为 0，则 *qualified-alias-member* 即表示该类型。
  - 否则，如果该全局命名空间包含名为 `I` 的带有 `K` 个类型形参的类型，则 *qualified-alias-member* 即表示使用给定的类型实参构造的该类型。
  - 否则，*qualified-alias-member* 是不确定的，并发生编译时错误。
- 否则，从直接包含 *qualified-alias-member* 的命名空间声明（第 9.2 节）（如果有）开始，继续处理每一个包含它的命名空间声明（如果有），最后在包含 *qualified-alias-member* 的编译单元结束，在找到实体之前，将计算下列步骤：
  - 如果命名空间声明或编译单元包含将 `N` 与某个类型相关联的 *using-alias-directive*，则 *qualified-alias-member* 是不确定的，并发生编译时错误。
  - 否则，如果命名空间声明或编译单元包含将 `N` 与某个命名空间相关联的 *extern-alias-directive* 或 *using-alias-directive*，则：
    - 如果与 `N` 关联的命名空间包含名为 `I` 的命名空间，并且 `K` 为 0，则 *qualified-alias-member* 即表示该命名空间。
    - 否则，如果与 `N` 关联的命名空间包含名为 `I` 的非泛型类型，并且 `K` 为 0，则 *qualified-alias-member* 即表示该类型。
    - 否则，如果与 `N` 关联的命名空间包含名为 `I` 的带有 `K` 个类型形参的类型，则 *qualified-alias-member* 即表示使用给定的类型实参构造的该类型。
    - 否则，*qualified-alias-member* 是不确定的，并发生编译时错误。
- 否则，*qualified-alias-member* 是不确定的，并发生编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

注意，将命名空间别名限定符与引用某个类型的别名一起使用将导致编译时错误。另请注意，如果标识符 N 为 global，则在全局命名空间中执行查找，即使存在将 global 与某个类型或命名空间关联的 using 别名。

9.7.1 别名的唯一性

每个编译单元和命名空间体对于 extern 别名和 using 别名都有单独的声明空间。因此，虽然 extern 别名或 using 别名的名称在直接包含它们的编译单元或命名空间体中声明的 extern 别名和 using 别名集中必须唯一，但是允许别名与类型或命名空间同名，只要它仅与 :: 限定符连用。

在下面的示例中

```
namespace N
{
   public class A {}
   public class B {}
}
namespace N
{
   using A = System.IO;
   class X
   {
      A.Stream s1;         // Error, A is ambiguous
      A::Stream s2;        // Ok
   }
}
```

名称 A 在第二个命名空间体中有两种可能的含义，因为类 A 和 using 别名 A 都在范围中。因此，在限定名 A.Stream 中使用的 A 是不确定的，并会导致发生编译时错误。但是，将 A 与 :: 限定符连用则不是错误，因为将 A 只作为命名空间别名进行查找。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10. 类

类是一种数据结构，它可以包含数据成员（常量和字段）、函数成员（方法、属性、事件、索引器、运算符、实例构造函数、静态构造函数和析构函数）以及嵌套类型。类类型支持继承，继承是一种机制，它使派生类可以对基类进行扩展和专用化。

10.1 类声明

class-declaration 是一个 *type-declaration*（第 9.6 节），它用于声明一个新类。

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `partial`$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$
class-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *class-body* `;`$_{opt}$

A *class-declaration* 的组成结构如下：开头是一组可选 *attributes*（第 17 章），然后依次是一组可选 *class-modifiers*（第 10.1.1 节）、可选 `partial` 修饰符、关键字 `class` 和用于命名类的 *identifier*、可选 *type-parameter-list*（第 10.1.3 节）、可选 *class-base* 规范（第 10.1.4 节）、一组可选 *type-parameter-constraints-clauses*（第 10.1.5 节）、*class-body*（第 10.1.6 节），最后是一个分号（可选）。

只有提供了 *type-parameter-list* 后，类声明才可以提供 *type-parameter-constraints-clauses*。

type-parameter-list 的类声明是一个泛型类声明 (generic class declaration)。此外，任何嵌套在泛型类声明或泛型结构声明中的类本身就是一个泛型类声明，因为必须为包含类型提供类型形参才能创建构造类型。

10.1.1 类修饰符

class-declaration 可以根据需要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`
`static`

同一修饰符在一个类声明中多次出现是编译时错误。

`new` 修饰符适用于嵌套类。它指定类隐藏同名的继承成员，详见第 10.3.4 节中的介绍。如果在不是嵌套类声明的类声明中使用 `new` 修饰符，则会导致编译时错误。

`public`、`protected`、`internal` 和 `private` 修饰符将控制类的可访问性。根据类声明所处的上下文，这其中的一些修饰符可能不允许使用（第 3.5.1 节）。

在以下几节中对 `abstract`、`sealed` 和 `static` 修饰符进行了讨论。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.1.1.1 抽象类

abstract 修饰符用于表示所修饰的类是不完整的，并且它只能用作基类。抽象类与非抽象类在以下方面是不同的：

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

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

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

抽象类 A 引入抽象方法 F。类 B 引入另一个方法 G，但由于它不提供 F 的实现，B 也必须声明为抽象类。类 C 重写 F，并提供一个具体实现。由于 C 没有抽象成员，因此 C 可以（但不要求）是非抽象的。

10.1.1.2 密封类

sealed 修饰符用于防止从所修饰的类派生出其他类。如果一个密封类被指定为其他类的基类，则会发生编译时错误。

密封类不能同时为抽象类。

sealed 修饰符主要用于防止非有意的派生，但是它还能促使某些运行时优化。具体而言，由于密封类永远不会有任何派生类，所以对密封类的实例的虚函数成员的调用可以转换为非虚调用来处理。

10.1.1.3 静态类

static 修饰符用于标记声明为静态类 (static class) 的类。静态类不能实例化，不能用作类型，而且仅可以包含静态成员。只有静态类才能包含扩展方法的声明（第 10.6.9 节）。

静态类声明受以下限制：

- 静态类不能包含 sealed 或 abstract 修饰符。但是，注意，因为无法实例化静态类或从静态类派生，所以静态类的行为就好像既是密封的又是抽象的。
- 静态类不能包含 *class-base* 规范（第 10.1.4 页），不能显式指定基类或所实现接口的列表。静态类隐式从 object 类型继承。
- 静态类只能包含静态成员（第 10.3.7 节）。注意，常量和嵌套类型归为静态成员。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 静态类不能含有声明的可访问性为 protected 或 protected internal 的成员。

违反上述任何限制都将导致编译时错误。

静态类没有实例构造函数。静态类中不能声明实例构造函数，并且对于静态类也不提供任何默认实例构造函数（第 10.11.4 节）。

静态类的成员并不会自动成为静态的，成员声明中必须显式包含一个 static 修饰符（常量和嵌套类型除外）。当一个类嵌套在一个静态的外层类中时，除非该类显式包含 static 修饰符，否则该嵌套类不是静态类。

10.1.1.3.1 引用静态类类型

如果下列条件成立，则允许 *namespace-or-type-name*（第 3.8 节）引用静态类：

- *namespace-or-type-name* 是 T.I 形式的 *namespace-or-type-name* 中的 T，或者
- *namespace-or-type-name* 是 typeof(T) 形式的 *typeof-expression*（第 7.5.11 节）中的 T。

如果下列条件成立，则允许 *primary-expression*（第 7.5 节）引用静态类：

- *primary-expression* 为 E.I 形式的 *member-access*（第 7.5.4 节）中的 E。

在任何其他上下文中，引用静态类将导致编译时错误。例如，将静态类用作基类、成员的构成类型（第 10.3.8 节）、泛型类型实参或类型形参约束都是错误的。同样，静态类不可用于数组类型、指针类型、new 表达式、强制转换表达式、is 表达式、as 表达式、sizeof 表达式或默认值表达式。

10.1.2 分部修饰符

partial 修饰符用于指示此 *class-declaration* 是分部类型声明。包容命名空间或类型声明中的多个同名分部类型声明按照第 10.2 节中指定的规则组合成一个类型声明。

如果程序文本的各独立段是在不同的上下文中产生或维护的，则在这些段上分布类声明非常有用。例如，类声明的某一部分可能是计算机生成的，而另一部分可能是手动创作的。将这两部分文本分开可以防止某人所做的更新与他人所做的更新发生冲突。

10.1.3 类型形参

类型形参是一个简单标识符，代表一个为创建构造类型而提供的类型实参的占位符。类型形参是将来提供的类型的形式占位符。而类型实参（第 4.4.1 节）是在创建构造类型时替换类型形参的实际类型。

type-parameter-list:
    < *type-parameters* >

type-parameters:
    $attributes_{opt}$ *type-parameter*
    type-parameters , $attributes_{opt}$ *type-parameter*

type-parameter:
    identifier

类声明中的每个类型形参在该类的声明空间（第 3.3 节）中定义一个名称。因此，它不能与另一个类型形参或该类中声明的成员具有相同的名称。类型形参不能与类型本身具有相同的名称。

10.1.4 类基本规范

类声明可以包含 *class-base* 规范，它定义该类的直接基类和由该类直接实现的接口（第 13 章）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

class-base:
: *class-type*
: *interface-type-list*
: *class-type* , *interface-type-list*

interface-type-list:
interface-type
interface-type-list , *interface-type*

类声明中指定的基类可以是构造类类型（第 4.4 节）。基类本身不能是类型形参，但在其作用域中可以包含类型形参。

```
class Extend<V>: V {}          // Error, type parameter used as base class
```

10.1.4.1 基类

当 *class-type* 中包含一个 *class-base* 时，它表示该类就是所声明的类的直接基类。如果一个类声明中没有 *class-base*，或 *class-base* 只列出接口类型，则假定直接基类就是 `object`。一个类会从它的直接基类继承成员，如第 10.3.3 节中所述。

在下面的示例中

```
class A {}
class B: A {}
```

称类 AA 为类 `B` 的直接基类，而称 `B` 是从 `A` 派生的。由于 `A` 没有显式地指定直接基类，它的直接基类隐含地为 `object`。

对于构造类类型，如果泛型类声明中指定了基类，则通过将基类声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument* 来获得构造类型的基类。假设有下面的泛型类声明

```
class B<U,V> {...}
class G<T>: B<string,T[]> {...}
```

构造类型 `G<int>` 的基类将是 `B<string,int[]>`。

类类型的直接基类必须至少与类类型（第 3.5.2 节）本身具有同样的可访问性。例如，试图从 `private` 或 `internal` 类派生一个 `public` 类，会导致编译时错误。

类类型的直接基类不能为下列任一类型：`System.Array`、`System.Delegate`、`System.MulticastDelegate`、`System.Enum` 或 `System.ValueType`。另外，泛型类声明不能将 `System.Attribute` 用作直接或间接基类。

在确定类 `B` 的直接基类规范 `A` 的含义时，将 `B` 的直接基类临时假定为 `object`。这在直观上确保基类规范的含义无法递归依赖其自身。示例：

```
class A<T> {
   public class B{}
}
class C : A<C.B> {}
```

是错误的，因为在基类规范 `A<C.B>` 中，将 `C` 的直接基类视为 `object`，因此(根据第 3.8 节中的规则)不将 `C` 视为具有成员 `B`。

一个类类型的基类包括它的直接基类以及该直接基类的基类。换句话说，基类集是直接基类关系的传递闭包。在上面的示例中，`B` 的基类是 `A` 和 `object`。在下面的示例中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class A {...}
class B<T>: A {...}
class C<T>: B<IComparable<T>> {...}
class D<T>: C<T[]> {...}
```

D<int> 的基类为 C<int[]>、B<IComparable<int[]>>、A 和 object。

除了类 object，每个类类型都只有一个直接基类。object 类没有任何直接基类，并且是所有其他类的终极基类。

当类 B 从类 A 派生时，A 依赖于 B 会导致编译时错误。类直接依赖于 (directly depends on) 它的直接基类（如果有），并且还直接依赖于它直接嵌套于其中的类（如果有）。从上述定义可以推出：一个类所依赖的类的完备集就是此直接依赖于(directly depends on) 关系的自反和传递闭包。

下面的示例

```
class A: A {}
```

是错误的，因为该类依赖于其自身。同样，示例

```
class A: B {}
class B: C {}
class C: A {}
```

是错误的，因为这些类之间循环依赖。最终，示例

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

也会导致编译时错误，原因是 A 依赖于 B.C（它的直接基类），B.C 依赖于 B（它的直接包容类），而 B 又循环地依赖于 A。

请注意，一个类不依赖于嵌套在它内部的类。在下面的示例中

```
class A
{
   class B: A {}
}
```

B 依赖于 A（原因是 A 既是它的直接基类又是它的直接包容类），但是 A 不依赖于 B（因为 B 既不是 A 的基类也不是它的包容类）。因此，此示例是有效的。

不能从一个 sealed 类派生出别的类。在下面的示例中

```
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 sealed 类 A 中派生。

10.1.4.2 接口实现

class-base 规范中可以包含一个接口类型列表，这表示所声明的类直接实现所列出的各个接口类型。第 13.4 节对接口实现进行了进一步讨论。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.1.5 类型形参约束

泛型类型和方法声明可以选择通过包括*type-parameter-constraints-clause*来指定类型形参约束。

type-parameter-constraints-clauses:
type-parameter-constraints-clause
type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
`where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
primary-constraint
secondary-constraints
constructor-constraint
primary-constraint `,` *secondary-constraints*
primary-constraint `,` *constructor-constraint*
secondary-constraints `,` *constructor-constraint*
primary-constraint `,` *secondary-constraints* `,` *constructor-constraint*

primary-constraint:
class-type
`class`
`struct`

secondary-constraints:
interface-type
type-parameter
secondary-constraints `,` *interface-type*
secondary-constraints `,` *type-parameter*

constructor-constraint:
`new` `(` `)`

每个*type-parameter-constraints-clause*都包括标记`where`，后面跟着类型形参的名称，再后面则跟着一个冒号和该类型形参的约束列表。每个类型形参最多只能有一个 `where`子句，并且`where`子句可以按任何顺序列出。与属性访问器中的`get`和`set`标记一样，`where`标记也不是关键字。

`where`子句中给出的约束列表可以包括以下任一组件，依次为：一个主要约束、一个或多个次要约束\b 以及构造函数约束\b \b `new()`。

主要约束可以是类类型、引用类型约束(reference type constraint) `class`，也可以是值类型约束(value type constraint) `struct`。次要约束可以是*type-parameter*，也可以是*interface-type*。

引用类型约束指定用于类型形参的类型实参必须是引用类型。所有类类型、接口类型、委托类型、数组类型和已知将是引用类型（将在下面定义）的类型形参都满足此约束。

值类型约束指定用于类型形参的类型实参必须是不可以为 null 值的类型。所有不可以为 null 的结构类型、枚举类型和具有值类型约束的类型形参都满足此约束。注意，虽然可以为 null 的类型（第 4.1.10 节）被归为值类型，但是不满足值类型约束。具有值类型约束的类型形参还不能具有*constructor-constraint*。

指针类型从不允许作为类型实参，并且不被视为满足引用类型或值类型约束。

如果约束是类类型、接口类型或类型形参，则该类型指定用于该类型形参的每个类型实参必须支持的最低“基类型”。每当使用构造类型或泛型方法时，都会在编译时根据类型形参上的约束检查类型实参。所提供的类型实参必须满足第 4.4.4 节中给出的条件。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

class-type 约束必须满足下列规则：

- 该类型必须是类类型。
- 该类型一定不能是 `sealed`。
- 该类型不能是以下类型之一：`System.Array`、`System.Delegate`、`System.Enum` 或 `System.ValueType`。
- 该类型一定不能是 `object`。由于所有类型都派生自 `object`，允许这样的约束没有任何作用。
- 给定的类型形参至多只能有一个约束可以是类类型。

指定为 *interface-type* 约束的类型必须满足下列规则：

- 该类型必须是接口类型。
- 不能在给定的 `where` 子句中多次指定某个类型。

在任一情况下，该约束都可以包括关联的类型或方法声明的任何类型形参作为构造类型的组成部分，并且可以包括被声明的类型。

指定为类型形参约束的任何类或接口类型必须至少与声明的泛型类型或方法具有相同的可访问性（第 3.5.4 节）。

指定为 *type-parameter* 约束的类型必须满足下列规则：

- 该类型必须是类型形参。
- 不能在给定的 `where` 子句中多次指定某个类型。

此外，类型形参的依赖关系图中一定不能存在循环，其中依赖性是通过下列方式定义的传递关系：

- 如果类型形参 `T` 用作类型形参 `S` 的约束，则 `S` 依赖 (depend on) `T`。
- 如果类型形参 `S` 依赖类型形参 `T`，并且 `T` 依赖类型形参 `U`，则 `S` ***依赖*** `U`。

根据这个关系，如果类型形参依赖自身（直接或间接），则会产生编译时错误。

相互依赖的类型形参之间的任何约束都必须一致。如果类型形参 `S` 依赖类型形参 `T`，则：

- `T` 一定不能具有值类型约束。否则，`T` 被有效地密封，使得 `S` 将被强制为与 `T` 相同的类型，从而消除了使用这两个类型形参的需要。
- 如果 `S` 具有值类型约束，则 `T` 不能具有 *class-type* 约束。
- 如果 `S` 具有 *class-type* 约束 `A`，`T` 具有 *class-type* 约束 `B`，则必须存在从 `A` 到 `B` 的标识转换或隐式引用转换或者从 `B` 到 `A` 的隐式引用转换。
- 如果 `S` 还依赖类型形参 `U`，并且 `U` 具有 *class-type* 约束 `A`，`T` 具有 *class-type* 约束 `B`，则必须存在从 `A` 到 `B` 的标识转换或隐式引用转换或者从 `B` 到 `A` 的隐式引用转换。

`S` 具有值类型约束而 `T` 具有引用类型约束是有效的。这实际上将 `T` 限制为类型 `System.Object`、`System.ValueType`、`System.Enum` 和任何接口类型。

如果类型形参的 `where` 子句包括构造函数约束（具有 `new()` 形式），则可以使用 `new` 运算符创建该类型的实例（第 7.6.10.1 节）。用于具有构造函数约束的类型形参的任何类型实参都必须具有公共的无形参构造函数（任何值类型都隐式地存在此构造函数），或者是具有值类型约束或构造函数约束的类型形参（有关详细信息，请参见第 10.1.5 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

下面是约束的示例：

```
interface IPrintable
{
   void Print();
}
interface IComparable<T>
{
   int CompareTo(T value);
}
interface IKeyProvider<T>
{
   T GetKey();
}
class Printer<T> where T: IPrintable {...}
class SortedList<T> where T: IComparable<T> {...}
class Dictionary<K,V>
   where K: IComparable<K>
   where V: IPrintable, IKeyProvider<K>, new()
{
   ...
}
```

下面的示例是错误的，因为它将导致类型形参的依赖关系发生循环：

```
class Circular<S,T>
   where S: T
   where T: S           // Error, circularity in dependency graph
{
   ...
}
```

下面的示例演示其他无效情况：

```
class Sealed<S,T>
   where S: T
   where T: struct      // Error, T is sealed
{
   ...
}
class A {...}
class B {...}
class Incompat<S,T>
   where S: A, T
   where T: B           // Error, incompatible class-type constraints
{
   ...
}
class StructWithClass<S,T,U>
   where S: struct, T
   where T: U
   where U: A           // Error, A incompatible with struct
{
   ...
}
```

类型 C 的*动态抹除*为类型 C_0，构造如下：

- 如果 C 为*嵌套类型* Outer.Inner，则 C_0 为嵌套类型 $Outer_0.Inner_0$。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果 C 为*构造类型* G<A¹、⋯ 和 Aⁿ> 以及类型实参 A¹、⋯ 和 Aⁿ，则 C_o 为构造类型 G<A^1_o、⋯ 和 A^n_o>。
- 如果 C 为*数组类型* E[]，则 C_o 为数组类型 E_o[]。
- 如果 C 为 *指针类型* E*，则 C_o 为指针类型 E_o*。
- 如果 C 为 dynamic，则 C_o 为 object.
- 否则，C_o 为 C。

类型形参 T 的有效基类 (effective base class) 定义如下：

让 R 成为类型集以便：

- 对于属于 *type-parameter* 的 T *的每个约束*，R 包含其有效基类。
- 对于属于 *struct-type* 的 T 的每个约束，R 包含 System.ValueType。
- 对于属于 *enumeration type* 的 T 的每个约束，R 包含 System.Enum。
- 对于属于 *delegate type* 的 T 的每个约束，R 包含其动态抹除。
- 对于属于 *array type* 的 T 的每个约束，R 包含 System.Array。
- 对于属于 *class-type* 的 T 的每个约束，R 包含其动态抹除。

则

- 如果 T 具有值类型约束，则其*有效基类*为 System.ValueType。
- 否则，如果 R 为空，则*有效基类*为 object。
- 否则，T 的*有效基类*是集 R 的被包含程度最大的类型（第 6.4.3 节）。如果该集没有被包含的类型，则 T 的*有效基类*为 object。一致性规则确保存在被包含程度最大的类型。

如果类型形参是其约束继承自基类方法的方法类型形参，则将在类型替换后计算*有效基类*。

这些规则可确保有效基类始终为 *class-type*。

类型形参 T 的有效接口集 (effective interface set) 定义如下：

- 如果 T 没有 *secondary-constraints*，则其有效接口集为空。
- 如果 T 具有 *interface-type* 约束，但是没有 *type-parameter* 约束，则其有效接口集为其 *interface-type* 约束的动态抹除集。
- 如果 T 没有 *interface-type* 约束，但是具有 *type-parameter* 约束，则其有效接口集为其 *type-parameter* 约束的有效接口集的并集。
- 如果 T 同时具有 *interface-type* 约束和 *type-parameter* 约束，则其有效接口集为其 *interface-type* 约束的动态抹除集和其 *type-parameter* 约束的有效接口集的并集。

如果类型形参具有引用类型约束，或其有效基类不是 object 或 System.ValueType，则该类型形参将视为一个引用类型 (known to be a reference type)。

受约束的类型形参类型的值可用于访问约束所暗示的实例成员。在下面的示例中

```
interface IPrintable
{
   void Print();
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Printer<T> where T: IPrintable
{
   void PrintOne(T x) {
      x.Print();
   }
}
```

可直接对 x 调用 IPrintable 的方法，因为 T 被约束为始终实现 IPrintable。

10.1.6 类体

一个类的 *class-body* 用于定义该类的成员。

class-body:
{ *class-member-declarations*$_{opt}$ }

10.2 分部类型

类型声明可以分为多个分部类型声明 (partial type declaration)。类型声明由它的各部分按照本节中的规则进行构造，因此在程序编译时和运行时的其余处理过程中，类型声明按单个声明处理。

如果 *class-declaration*、*struct-declaration* 或 *interface-declaration* 包含 partial 修饰符，则它表示分部类型声明。partial 不是关键字，仅在它紧靠关键字 class、struct 或 interface 中的某一个之前出现在类型声明中或紧靠类型 void 之前出现在方法声明中时充当修饰符。在其他上下文中，它可用作正常标识符。

分部类型声明中的每一部分都必须包括一个 partial 修饰符。它必须和其他部分具有相同名称，并且必须与其他部分在同一命名空间或类型声明中声明。partial 修饰符的出现指示其他位置可能还有类型声明的其他部分，但是这些其他部分并非必须存在；对于只具有一个声明的类型，包含 partial 修饰符也是有效的。

分部类型的所有部分必须一起编译，以便这些部分可在编译时合并为单个类型声明。特别指出的是，分部类型不允许对已经编译的类型进行扩展。

可使用 partial 修饰符在多个部分中声明嵌套类型。通常，其包含类型也使用 partial 声明，并且嵌套类型的每个部分均在该包含类型的不同部分中声明。

不允许使用 partial 修饰符声明委托或枚举。

10.2.1 特性

分部类型的特性是通过组合每个部分的特性（不指定顺序）来确定的。如果对多个部分放置同一个特性，则相当于多次对该类型指定此特性。例如，下面的两个部分：

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

相当于下面的声明：

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

类型形参的特性以类似的方式进行组合。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.2.2 修饰符

当分部类型声明指定了可访问性（public、protected、internal 和 private 修饰符）时，它必须与所有其他指定了可访问性的部分一致。如果分部类型的所有部分都未指定可访问性，则会向该分部类型提供相应的默认可访问性（第3.5.1 节）。

如果嵌套类型的一个或多个分部声明包含 new 修饰符，则在嵌套类型对继承的成员进行了隐藏（第3.7.1.2 节）的情况不会报告任何警告。

如果某个类的一个或多个分部声明包含 abstract 修饰符，则该类被视为抽象类（第10.1.1.1 节）。否则，该类被视为非抽象类。

如果某个类的一个或多个分部声明包含 sealed 修饰符，则该类被视为密封类（第 10.1.1.2 节）。否则，该类被视为非封闭类。

注意，一个类不能既是抽象类又是密封类。

当 unsafe 修饰符用于某个分部类型声明时，只有该特定部分才被视为不安全的上下文（第 18.1 节）。

10.2.3 类型形参和约束

如果在多个部分中声明泛型类型，则每个部分都必须声明类型形参。每个部分都必须有相同数目的类型形参，并且每个类型形参按照顺序有相同的名称。

当分部泛型类型声明包含约束（where 子句）时，该约束必须与包含约束的所有其他部分一致。具体而言，包含约束的每个部分都必须有针对相同的类型形参集的约束，并且对于每个类型形参，主要、次要和构造函数约束集都必须等效。如果两个约束集包含相同的成员，则它们等效。如果某个分部泛型类型的任何部分都未指定类型形参约束，则该类型形参被视为无约束。

下面的示例

```
partial class Dictionary<K,V>
   where K: IComparable<K>
   where V: IKeyProvider<K>, IPersistable
{
   ...
}

partial class Dictionary<K,V>
   where V: IPersistable, IKeyProvider<K>
   where K: IComparable<K>
{
   ...
}

partial class Dictionary<K,V>
{
   ...
}
```

是正确的，因为包含约束的那些部分（前两个）实际上分别对相同的类型形参集指定了相同的主要、次要和构造函数约束集。

10.2.4 基类

当一个分部类声明包含基类说明时，它必须与包含基类说明的所有其他部分一致。如果某个分部类的任何部分都不包含基类说明，则基类将为System.Object（第 10.1.4.1 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.2.5 基接口

在多个部分中声明的类型的基接口集是每个部分中指定的基接口的并集。一个特定基接口在每个部分中只能指定一次，但是允许多个部分指定相同的基接口。任何给定基接口的成员只能有一个实现。

在下面的示例中

```
partial class C: IA, IB {...}
partial class C: IC {...}
partial class C: IA, IB {...}
```

类 C 的基接口集为 IA、IB 和 IC。

通常，每个部分都提供了在该部分上声明的接口的实现；但这不是必需的。一个部分可能提供在另一个部分上声明的接口的实现：

```
partial class X
{
   int IComparable.CompareTo(object o) {...}
}
partial class X: IComparable
{
   ...
}
```

10.2.6 成员

除了分部方法（第 10.2.7 节），在多个部分中声明的类型的成员集仅仅是在每个部分中声明的成员集的并集。所有部分的类型声明主体共享相同的声明空间（第 3.3 节），并且每个成员的范围（第 3.7 节）都扩展到所有部分的主体。任何成员的可访问性域总是包含包容类型的所有部分；在一个部分中声明的 private 成员可从其他部分随意访问。在类型的多个部分中声明同一个成员将引起编译时错误，除非该成员是带有 partial 修饰符的类型。

```
partial class A
{
   int x;                  // Error, cannot declare x more than once
   partial class Inner     // Ok, Inner is a partial type
   {
      int y;
   }
}
partial class A
{
   int x;                  // Error, cannot declare x more than once
   partial class Inner     // Ok, Inner is a partial type
   {
      int z;
   }
}
```

类型中的成员的顺序对于C# 代码无关紧要，但是在与其他语言或环境交互时，这可能很重要。在这些情况下，没有对分为多个部分声明的类型内的成员顺序进行定义。

10.2.7 分部方法

分部方法可以在类型声明的一个部分中定义，而在另一个部分中实现。实现是可选的；如果所有部分都未实现分部方法，则将从组合各部分而构成的类型声明中，移除分部方法声明和所有对它的调用。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

分部方法不能定义访问修饰符，而隐含为 private。它们的返回类型必须是 void，而且它们的形参不能带有 out 修饰符。在方法声明中，仅当标识符 partial 紧靠 void 类型之前出现时，才将它识别为特殊关键字，否则，将它用作正常标识符。分部方法不能显式实现接口方法。

有两种类型的分部方法声明：如果方法声明体是一个分号，则称该声明是定义分部方法声明 (defining partial method declaration)。如果以 *block* 形式给定该声明体，则称该声明是实现分部方法声明 (implementing partial method declaration)。在类型声明的各个部分，只能有一个具有给定签名的定义分部方法声明，也只能有一个具有给定签名的实现分部方法声明。如果给定了实现分部方法声明，则必须存在相应的定义分部方法声明，并且这两个声明必须符合以下指定的内容：

- 这两个声明必须具有相同的修饰符（但不必采用同一顺序）、方法名、类型形参数目和形参数目。
- 声明中相应的形参必须具有相同的修饰符（但不必采用同一顺序）和相同的类型（类型形参名称中的模不同）。
- 声明中的相应类型形参必须具有相同的约束（类型形参名称中的模不同）。

实现分部方法声明可以与相应的定义分部方法声明出现在同一部分。

只有定义分部方法会参与重载决策。因此，无论是否给定实现声明，调用表达式都可以解析分部方法的调用。因为分部方法始终返回 void，所以此类调用表达式始终为表达式语句。而且，因为分部方法隐含为 private，所以此类语句将始终在声明了该分部方法的类型声明的其中某一部分出现。

如果分部类型声明中的任何部分都不包含给定分部方法的实现声明，则调用它的任何表达式语句都将仅从组合的类型声明中移除。因此，调用表达式（包括所有构成表达式）在运行时将不起作用。分部方法本身也将从组合的类型声明中移除，不再是其中的成员。

如果给定的分部方法存在实现声明，则分部方法的调用将保留。分部方法将产生类似于实现分部方法声明的方法声明，但以下内容除外：

- 不包括 partial 修饰符
- 结果方法声明中的特性是未指定顺序的定义分部方法声明和实现分部方法声明的组合特性。不移除重复项。
- 结果方法声明中的形参的特性是未指定顺序的定义分部方法声明和实现分部方法声明的相应形参的组合特性。不移除重复项。

如果为分部方法 M 指定的是定义声明而不是实现声明，则应用以下限制：

- 如果创建该方法的委托（第 7.6.10.5 节），则会出现编译时错误。
- 如果在转换为表达式目录树类型（第 6.5.2 节）的匿名函数内引用 M，则会出现编译时错误。
- 作为调用 M 的一部分出现的表达式不影响明确赋值状态（第 5.3 节），这可能会导致编译时错误。
- M 不能是应用程序的入口点（第 3.1 节）。

分部方法对于允许类型声明的一部分自定义另一部分的行为（例如由工具生成的行为）非常有用。请考虑以下分部类声明：

```
partial class Customer
{
   string name;
   public string Name {
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }
    partial void OnNameChanging(string newName);
    partial void OnNameChanged();
}
```

如果该类不与其他任何部分一起编译，则将移除定义分部方法声明及其调用，产生的组合类声明将等效于以下内容：

```
class Customer
{
    string name;
    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

但是，假设给定的是另一部分，该部分提供分部方法的实现声明：

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }
    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

那么，产生的组合类声明将等效于以下内容：

```
class Customer
{
    string name;
    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }
    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

10.2.8 名称绑定

虽然可扩展类型的每个部分都必须在同一命名空间中声明，但是这些部分通常在不同的命名空间声明下编写。因此，每个部分可能存在不同的using指令（第9.4节）。当解释一个部分内的简单名称（第7.5.2节）时，只考虑包容该部分的命名空间定义的using指令。这可能会导致同一标识符在不同部分中具有不同的含义：

```
namespace N
{
    using List = System.Collections.ArrayList;
    partial class A
    {
        List x;           // x has type System.Collections.ArrayList
    }
}
namespace N
{
    using List = Widgets.LinkedList;
    partial class A
    {
        List y;           // y has type Widgets.LinkedList
    }
}
```

10.3 类成员

一个类的成员由两部分组成：由它的*class-member-declaration*引入的成员；从它的直接基类继承来的成员。

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

一个类类型的成员分为下列几种类别：

- 常量，表示与该类相关联的常量值（第10.4节）。
- 字段，即该类的变量（第10.5节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 方法，用于实现可由该类执行的计算和操作（第 10.6 节）。
- 属性，用于定义一些命名特性以及与读取和写入这些特性相关的操作（第 10.7 节）。
- 事件，用于定义可由该类生成的通知（第 10.8 节）。
- 索引器，使该类的实例可按与数组相同的（语法）方式进行索引（第 10.9 节）。
- 运算符，用于定义表达式运算符，通过它对该类的实例进行运算（第 10.10 节）。
- 实例构造函数，用于实现初始化该类的实例所需的操作（第 10.11 节）
- 析构函数，用于实现在永久地放弃该类的一个实例之前要执行的操作（第 10.13 节）。
- 静态构造函数，用于实现初始化该类自身所需的操作（第 10.12 节）。
- 类型，用于表示一些类型，它们是该类的局部类型（第 10.3.8 节）。

可以包含可执行代码的成员统称为该类类型的 *function members*。类类型的函数成员包括：方法、属性、事件、索引器、运算符、实例构造函数、析构函数和该类类型的静态构造函数。

class-declaration 将创建一个新的声明空间（第 3.3 节），而直接包含在该 *class-declaration* 内的 *class-member-declarations* 将向此声明空间中引入新成员。下列规则适用于 *class-member-declarations*：

- 实例构造函数、静态构造函数和析构函数必须具有与直接包容它们的类相同的名称。所有其他成员的名称必须与该类的名称不同。
- 常量、字段、属性、事件或类型的名称必须不同于在同一个类中声明的所有其他成员的名称。
- 方法的名称必须不同于在同一个类中声明的所有其他非方法的名称。此外，必须不同于在同一类中声明的所有其他方法的签名（第 3.6 节），并且在同一类中声明的两种方法的签名不能只有 `ref` 和 `out` 不同。
- 实例构造函数的签名必须不同于在同一类中声明的所有其他实例的签名，并且在同一类中声明的两个构造函数的签名不能只有 `ref` 和 `out` 不同。
- 索引器的签名必须不同于在同一个类中声明的所有其他索引器的签名。
- 运算符的签名必须不同于在同一个类中声明的所有其他运算符的签名。

类类型的继承成员（第 10.3.3 节）不是类的声明空间的组成部分。因此，一个派生类可以使用与所继承的成员相同的名称或签名来声明自己的新成员（这同时也隐藏了被继承的同名成员）。

10.3.1 实例类型

每个类声明都有一个关联的绑定类型（第 4.4.3 节），即实例类型(instance type)。对于泛型类声明，实例类型是通过从该类型声明创建构造类型（第 4.4 节）来构成的，所提供的每个类型实参替换对应的类型形参。由于实例类型使用类型形参，因此只能在类型形参的作用域中使用该实例类型；也就是在类声明的内部。对于在类声明中编写的代码，实例类型为 `this` 的类型。对于非泛型类，实例类型就是所声明的类。下面显示几个类声明以及它们的实例类型：

```
class A<T>                  // instance type: A<T>
{
   class B {}               // instance type: A<T>.B
   class C<U> {}            // instance type: A<T>.C<U>
}
class D {}                  // instance type: D
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.3.2 构造类型的成员

构造类型的非继承成员是通过将成员声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument* 来获得的。替换过程基于类型声明的语义含义，并不只是文本替换。

例如，给定下面的泛型类声明

```
class Gen<T,U>
{
   public T[,] a;
   public void G(int i, T t, Gen<U,T> gt) {...}
   public U Prop { get {...} set {...} }
   public int H(double d) {...}
}
```

构造类型 `Gen<int[],IComparable<string>>` 具有以下成员：

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

泛型类声明 `Gen` 中的成员 `a` 的类型是“`T` 的二维数组”，因此上面的构造类型中的成员 `a` 的类型是“`int` 的一维数组的二维数组”，或 `int[,][]`。

在实例函数成员中，类型 pe of `this` 是包含这些成员的声明的实例类型（第 10.3.1 节）。

泛型类的所有成员都可以直接或作为构造类型的一部分使用任何包容类 (enclosing class) 中的类型形参。当在运行时使用特定的封闭构造类型（第 4.4.2 节）时，所使用的每个类型形参都被替换成提供给该构造类型的实际类型实参。例如：

```
class C<V>
{
   public V f1;
   public C<V> f2 = null;

   public C(V x) {
      this.f1 = x;
      this.f2 = this;
   }
}

class Application
{
   static void Main() {
      C<int> x1 = new C<int>(1);
      Console.WriteLine(x1.f1);     // Prints 1

      C<double> x2 = new C<double>(3.1415);
      Console.WriteLine(x2.f1);     // Prints 3.1415
   }
}
```

10.3.3 继承

一个类继承 (inherit) 它的直接基类类型的成员。继承意味着一个类隐式地将它的直接基类类型的所有成员当作自己的成员，但基类的实例构造函数、析构函数和静态构造函数除外。继承的一些重要性质为：

- 继承是可传递的。如果 `C` 从 `B` 派生，而 `B` 从 `A` 派生，则 `C` 将既继承在 `B` 中声明的成员，又继承在 `A` 中声明的成员。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 派生类扩展它的直接基类。派生类能够在继承基类的基础上添加新的成员，但是它不能移除继承成员的定义。
- 实例构造函数、析构函数和静态构造函数是不可继承的，但所有其他成员是可继承的，无论它们所声明的可访问性（第 3.5 节）如何。但是，根据它们所声明的可访问性，有些继承成员在派生类中可能是无法访问的。
- 派生类可以通过声明具有相同名称或签名的新成员来隐藏 (hide)（第 3.7.1.2 节）那个被继承的成员。但是，请注意隐藏继承成员并不移除该成员，它只是使被隐藏的成员在派生类中不可直接访问。
- 类的实例含有在该类中以及它的所有基类中声明的所有实例字段集，并且存在一个从派生类类型到它的任一基类类型的隐式转换（第 6.1.6 节）。因此，可以将对某个派生类实例的引用视为对它的任一个基类实例的引用。
- 类可以声明虚的方法、属性和索引器，而派生类可以重写这些函数成员的实现。这使类展示出“多态性行为”特征，也就是说，同一个函数成员调用所执行的操作可能是不同的，这取决于用来调用该函数成员的实例的运行时类型。

构造类类型的继承成员是直接基类类型的成员（第 10.1.4.1 节），用构造类型的类型实参替换 *base-class-specification* 中出现的每个相应的类型形参，可以找到这些继承成员。反过来，通过将 *base-class-specification* 的相应 *type-argument* 替换为成员声明中的每个 *type-parameter*，又可以转换这些成员。

```
class B<U>
{
   public U F(long index) {...}
}

class D<T>: B<T[]>
{
   public T G(string s) {...}
}
```

在上面的示例中，构造类型 `D<int>` 具有一个非继承的成员 `public int G(string s)`，该成员是通过将类型形参 `T` 替换为类型实参 `int` 来获得的。`D<int>` 还有一个从类声明 `B` 继承的成员。先用 `int` 替换基类说明 `B<T[]>` 中的 `T` 来确定 `D<int>` 的基类类型 `B<int[]>`，以此来确定该继承成员。然后，作为 `B` 的类型实参，`int[]` 将用来替换 `public U F(long index)` 中的 `U`，从而得到继承的成员 `public int[] F(long index)`。

10.3.4 new 修饰符

class-member-declaration 中可以使用与一个被继承的成员相同的名称或签名来声明一个成员。发生这种情况时，就称该派生类成员隐藏 (hide) 了基类成员。隐藏一个继承的成员不算是错误，但这确实会导致编译器发出警告。若要取消此警告，派生类成员的声明中可以包含一个 `new` 修饰符，表示派生成员是有意隐藏基成员的。第 3.7.1.2 节对本主题进行了进一步讨论。

如果在不隐藏所继承成员的声明中包含 `new` 修饰符，将对此状况发出警告。通过移除 `new` 修饰符可取消显示此警告。

10.3.5 访问修饰符

class-member-declaration 可以具有下列五种可能的已声明可访问性（第 3.5.1 节）中的任意一种：`public`、`protected internal`、`protected`、`internal` 或 `private`。除 `protected internal` 组合外，指定一个以上的访问修饰符会导致编译时错误。当 *class-member-declaration* 不包含任何访问修饰符时，假定为 `private`。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.3.6 构成类型

在成员声明中所使用的类型称为成员的构成类型。可能的构成类型包括常量、字段、属性、事件或索引器类型，方法或运算符的返回类型，以及方法、索引器、运算符和实例构造函数的形参类型。成员的构成类型必须至少具有与该成员本身相同的可访问性（第 3.5.4 节）。

10.3.7 静态成员和实例成员

类的成员或者是静态成员 (static member)，或者是实例成员 (instance member)。一般说来，可以这样来理解：静态成员属于类类型，而实例成员属于对象（类类型的实例）。

当字段、方法、属性、事件、运算符或构造函数声明中含有 `static` 修饰符时，它声明静态成员。此外，常量或类型声明会隐式地声明静态成员。静态成员具有下列特征：

- 在 `E.M` 形式的 *member-access*（第 7.6.4 节）中引用静态成员 `M` 时，`E` 必须表示含有 `M` 的那个类型。`E` 若表示一个实例，则会导致编译时错误。
- 静态字段只标识一个要由给定的封闭类类型的所有实例共享的存储位置。无论对一个给定的封闭式类类型创建了多少个实例，它的静态字段永远都只有一个副本。
- 静态函数成员（方法、属性、事件、运算符或构造函数）不能作用于具体的实例，在这类函数成员中引用 `this` 会导致编译时错误。

当字段、方法、属性、事件、索引器、构造函数或析构函数的声明中不包含 `static` 修饰符时，它声明实例成员。（实例成员有时称为非静态成员。）实例成员具有以下特点：

- 在 `E.M` 形式的 *member-access*（第 7.6.4 节）中引用实例成员 `M` 时，`E` 必须表示某个含有 `M` 的类型的一个实例。`E` 若表示类型，则会导致绑定时错误。
- 类的每个实例分别包含一组该类的所有实例字段。
- 实例函数成员（方法、属性、索引器、实例构造函数或析构函数）作用于类的给定实例，此实例可以用 `this` 访问（第 7.6.7 节）。

下列示例阐释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
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

F 方法显示，在实例函数成员中，*simple-name*（第 7.6.2 节）既可用于访问实例成员也可用于访问静态成员。G 方法显示，在静态函数成员中，通过 *simple-name* 访问实例成员会导致编译时错误。Main 方法显示，在 *member-access*（第 7.6.4 节）中，实例成员必须通过实例访问，静态成员必须通过类型访问。

10.3.8 嵌套类型

在类或结构声明内部声明的类型称为嵌套类型 (nested type)。在编译单元或命名空间内声明的类型称为非嵌套类型 (non-nested type)。

在下面的示例中

```
using System;
class A
{
   class B
   {
      static void F() {
         Console.WriteLine("A.B.F");
      }
   }
}
```

类 B 是嵌套类型，这是因为它在类 A 内声明，而由于类 A 在编译单元内声明，因此它是非嵌套类型。

10.3.8.1 完全限定名

嵌套类型的完全限定名（第 3.8.1 节）为 S.N，其中 S 是声明了 N 类型的那个类型的完全限定名。

10.3.8.2 已声明可访问性

非嵌套类型可以具有 public 或 internal 已声明可访问性，并且在默认情况下具有 internal 已声明可访问性。嵌套类型也可以具有上述两种声明可访问性，外加一种或更多种其他的声明可访问性，具体取决于包含它的那个类型是类还是结构：

- 在类中声明的嵌套类型可以具有五种形式的已声明可访问性（public、protected internal、protected、internal 或 private）中的任一种，而且与其他类成员一样，默认的已声明可访问性为 private。
- 在结构中声明的嵌套类型可以具有三种已声明可访问性（public、internal 或 private）中的任一种形式，而且与其他结构成员一样，默认的已声明可访问性为 private。

下面的示例

```
public class List
{
   // Private data structure
   private class Node
   {
      public object Data;
      public Node Next;

      public Node(object data, Node next) {
         this.Data = data;
         this.Next = next;
      }
   }

   private Node first = null;
   private Node last = null;

   // Public interface
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

声明了一个私有嵌套类 Node。

10.3.8.3 隐藏

嵌套类型可以隐藏（第 3.7.1 节）基成员。对嵌套类型声明允许使用 new 修饰符，以便可以明确表示隐藏。下面的示例

```
using System;
class Base
{
   public static void M() {
      Console.WriteLine("Base.M");
   }
}
class Derived: Base
{
   new public class M
   {
      public static void F() {
         Console.WriteLine("Derived.M.F");
      }
   }
}
class Test
{
   static void Main() {
      Derived.M.F();
   }
}
```

演示嵌套类 M，该类隐藏了在 Base 中定义的方法 M。

10.3.8.4 this 访问

关于 *this-access*（第 7.6.7 节），嵌套类型和包含它的那个类型并不具有特殊的关系。准确地说，在嵌套类型内，this 不能用于引用包含它的那个类型的实例成员。当需要在嵌套类型内部访问包含它的那个类型的实例成员时，通过将代表所需实例的 this 作为一个实参传递给该嵌套类型的构造函数，就可以进行所需的访问了。以下示例

```
using System;
class C
{
   int i = 123;
   public void F() {
      Nested n = new Nested(this);
      n.G();
   }
   public class Nested
   {
      C this_c;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}
```

演示了此技巧。C 实例创建了一个 Nested 实例并将代表它自己的 this 传递给 Nested 的构造函数，这样，就可以对 C 的实例成员进行后续访问了。

10.3.8.5 对包含类型的私有和受保护成员的访问

嵌套类型可以访问包含它的那个类型可访问的所有成员，包括该类型自己的具有 private 和 protected 声明可访问性的成员。下面的示例

```
using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

class Test
{
    static void Main() {
        C.Nested.G();
    }
}
```

演示包含有嵌套类 Nested 的类 C。在 Nested 内，方法 G 将调用在 C 中定义的静态方法 F，而 F 具有 private 已声明可访问性。

嵌套类型还可以访问在包含它的那个类型的基类型中定义的受保护成员。在下面的示例中

```
using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Derived: Base
{
   public class Nested
   {
      public void G() {
         Derived d = new Derived();
         d.F();      // ok
      }
   }
}

class Test
{
   static void Main() {
      Derived.Nested n = new Derived.Nested();
      n.G();
   }
}
```

嵌套类 Derived.Nested 通过对一个 Derived 的实例进行调用，访问在 Derived 的基类 Base 中定义的受保护方法 F。

10.3.8.6 泛型类中的嵌套类型

泛型类声明可以包含嵌套的类型声明。包容类的类型形参可以在嵌套类型中使用。嵌套类型声明可以包含仅适用于该嵌套类型的附加类型形参。

泛型类声明中包含的每个类型声明都隐式地是泛型类型声明。在编写对嵌套在泛型类型中的类型的引用时，必须指定其包容构造类型（包括其类型实参）。但是可在外层类中不加限定地使用嵌套类型；在构造嵌套类型时可以隐式地使用外层类的实例类型。下面的示例演示三种不同的引用从 Inner 创建的构造类型的正确方法；前两种方法是等效的：

```
class Outer<T>
{
   class Inner<U>
   {
      public static void F(T t, U u) {...}
   }

   static void F(T t) {
      Outer<T>.Inner<string>.F(t, "abc");    // These two statements have
      Inner<string>.F(t, "abc");             // the same effect

      Outer<int>.Inner<string>.F(3, "abc");  // This type is different

      Outer.Inner<string>.F(t, "abc");       // Error, Outer needs type arg
   }
}
```

嵌套类型中的类型形参可以隐藏外层类型中声明的成员或类型形参，但这是一种不好的编程风格：

```
class Outer<T>
{
   class Inner<T>    // Valid, hides Outer's T
   {
      public T t;    // Refers to Inner's T
   }
}
```

10.3.9 保留成员名称

为便于基础 C# 运行时的实现，对于每个属性、事件或索引器的源成员声明，实现都必须根据成员声明的种类、名称和类型保留两个方法签名。如果程序声明一个成员，而该成员的签名与这些保留签名之一匹配，那么即使基础运行时实现不使用这些保留签名，仍会出现编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

保留名称不会引入声明，因此它们不参与成员查找。但是，一个声明的关联的保留方法签名的确参与继承（第 10.3.3 节），而且可以使用 new 修饰符（第 10.3.4 节）隐藏起来。

保留这些名称有三个目的：

- 使基础的实现可以通过将普通标识符用作一个方法名称，从而对 C# 语言的功能进行 get 或 set 访问。
- 使其他语言可以通过将普通标识符用作一个方法名称，对 C# 语言的功能进行 get 或 set 访问，从而实现交互操作。
- 使保留成员名称的细节在所有的 C# 实现中保持一致，这有助于确保被一个符合本规范的编译器所接受的源程序也可被另一个编译器接受。

析构函数（第 10.13 节）的声明也会导致一个签名被保留（第 10.3.9.4 节）。

10.3.9.1 为属性保留的成员名称

对于类型 T 的属性 P（第 10.7 节），保留了下列签名：

```
T get_P();
void set_P(T value);
```

即使该属性是只读或者只写的，这两个签名仍然都被保留。

在下面的示例中

```
using System;

class A
{
   public int P {
      get { return 123; }
   }
}

class B: A
{
   new public int get_P() {
      return 456;
   }

   new public void set_P(int value) {
   }
}

class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      Console.WriteLine(a.P);
      Console.WriteLine(b.P);
      Console.WriteLine(b.get_P());
   }
}
```

类 A 定义了只读属性 P，从而保留了 get_P 和 set_P 方法的签名。类 B 从 A 派生并隐藏了这两个保留的签名。此例产生输出：

```
123
123
456
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.3.9.2 为事件保留的成员名称

对于委托类型 T 的事件 E（第 10.8 节），保留了下列签名：

```
void add_E(T handler);
void remove_E(T handler);
```

10.3.9.3 为索引器保留的成员名称

对于类型 T 的且具有形参列表 L 的索引器（第 10.9 节），保留了下列签名：

```
T get_Item(L);
void set_Item(L, T value);
```

即使索引器是只读或者只写的，这两个签名仍然都被保留。

此外，保留成员名称 Item。

10.3.9.4 为析构函数保留的成员名称

对于包含析构函数（第 10.13 节）的类，保留了下列签名：

```
void Finalize();
```

10.4 常量

常量 (constant) 是表示常量值（即，可以在编译时计算的值）的类成员。*constant-declaration* 可引入一个或多个给定类型的常量。

constant-declaration:
attributes$_{opt}$ *constant-modifiers*$_{opt}$ `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*

constant-declaration 可包含一组 *attributes*（第 17 章）、一个 new 修饰符（第 10.3.4 节）和一个由四个访问修饰符构成的有效组合（第 10.3.5 节）。特性和修饰符适用于所有由 *constant-declaration* 所声明的成员。虽然常量被认为是静态成员，但在 *constant-declaration* 中既不要求也不允许使用 `static` 修饰符。同一个修饰符在一个常量声明中多次出现是错误的。

constant-declaration 的 *type* 用于指定由声明引入的成员的类型。类型后接一个 *constant-declarator* 列表，该列表中的每个声明符引入一个新成员。*constant-declarator* 包含一个用于命名该成员的 *identifier*，后接一个“=”标记，然后跟一个对该成员赋值的 *constant-expression*（第 7.19 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在常量声明中指定的 *type* 必须是 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string、 *enum-type* 或 *reference-type*。每个 *constant-expression* 所产生的值必须属于目标类型，或者可以通过一个隐式转换（第0) 节）转换为目标类型。

常量的 *type* 必须至少与常量本身（第3.5.4 节）具有同样的可访问性。

使用 *simple-name*（第 7.6.2 节）或 *member-access*（第 7.6.4 节）从表达式获取常量的值。

常量本身可以出现在 *constant-expression* 中。因此，常量可用在任何需要 *constant-expression* 的构造中。这样的构造示例包括 case 标签、goto case 语句、enum 成员声明、属性和其他的常量声明。

如第 7.19 节中所描述，*constant-expression* 是在编译时就可以完全计算出来的表达式。由于创建 string 以外的 *reference-type* 的非 null 值的唯一方法是应用 new 运算符，但 *constant-expression* 中不允许使用 new 运算符，因此，除 string 以外的 *reference-types* 常量的唯一可能的值是 null。

如果需要一个具有常量值的符号名称，但是该值的类型不允许在常量声明中使用，或在编译时无法由 *constant-expression* 计算出该值，则可以改用 readonly 字段（第 10.5.2 节）。

声明了多个常量的一个常量声明等效于具有相同特性、修饰符和类型的多个常量的声明，其中每个声明均只声明一个常量。例如

```
class A
{
   public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

相当于

```
class A
{
   public const double X = 1.0;
   public const double Y = 2.0;
   public const double Z = 3.0;
}
```

一个常量可以依赖于同一程序内的其他常量，只要这种依赖关系不是循环的。编译器会自动地安排适当的顺序来计算各个常量声明。在下面的示例中

```
class A
{
   public const int X = B.Z + 1;
   public const int Y = 10;
}

class B
{
   public const int Z = A.Y + 1;
}
```

编译器首先计算 A.Y，然后计算 B.Z，最后计算 A.X，产生值 10、11 和 12。常量声明也可以依赖于其他程序中的常量，但这种依赖关系只能是单方向的。上面的示例中，如果 A 和 B 在不同的程序中声明，A.X 可以依赖于 B.Z，但是 B.Z 就无法同时再依赖于 A.Y 了。

10.5 字段

字段 (field) 是一种表示与对象或类关联的变量的成员。*field-declaration* 用于引入一个或多个给定类型的字段。

field-declaration:
attributes$_{opt}$ *field-modifiers*$_{opt}$ *type* *variable-declarators* ;

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

field-modifiers:
field-modifier
field-modifiers field-modifier

field-modifier:
`new`
`public`
`protected`
`internal`
`private`
`static`
`readonly`
`volatile`

variable-declarators:
variable-declarator
variable-declarators , *variable-declarator*

variable-declarator:
identifier
identifier = *variable-initializer*

variable-initializer:
expression
array-initializer

field-declaration 可以包含一组 *attributes*（第 17 章），一个 `new` 修饰符（第 10.3.4 节），由四个访问修饰符组成的一个有效组合（第 10.3.5 节）和一个 `static` 修饰符（第 10.5.1 节）。此外，*field-declaration* 可以包含一个 `readonly` 修饰符（第 10.5.2 节）或一个 `volatile` 修饰符（第 10.5.3 节），但不能同时包含这两个修饰符。特性和修饰符适用于由该 *field-declaration* 所声明的所有成员。同一个修饰符在一个字段声明中多次出现是错误的。

field-declaration 的 *type* 用于指定由该声明引入的成员的类型。类型后接一个 *variable-declarator* 列表，其中每个变量声明符引入一个新成员。*variable-declarator* 包含一个用于命名该成员的 *identifier*，还可以根据需要再后接一个“=”标记，以及一个用于赋予成员初始值的 *variable-initializer*（第 10.5.5 节）。

字段的 *type* 必须至少与字段本身（第 3.5.4 节）具有同样的可访问性。

使用 *simple-name*（第 7.6.2 节）或 *member-access*（第 7.6.4 节）从表达式获得字段的值。使用 *assignment*（第 7.17 节）修改非只读字段的值。可以使用后缀增量和减量运算符（第 7.6.9 节）以及前缀增量和减量运算符（第 7.7.5 节）获取和修改非只读字段的值。

声明了多个字段的一个字段声明等效于具有相同特性、修饰符和类型的多个字段的声明，其中每个声明均只声明一个字段。例如

```
class A
{
   public static int X = 1, Y, Z = 100;
}
```

相当于

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class A
{
   public static int X = 1;
   public static int Y;
   public static int Z = 100;
}
```

10.5.1 静态字段和实例字段

当一个字段声明中含有 `static` 修饰符时，由该声明引入的字段为静态字段 (static field)。当不存在 `static` 修饰符时，由该声明引入的字段为 i*instance fields*。静态字段和实例字段是 C# 所支持的几种变量（第 5 章）中的两种，它们有时被分别称为静态变量 (static variable) 和实例变量 (instance variable)。

静态字段不是特定实例的一部分，而是在封闭类型的所有实例之间共享（第4.4.2 节）。不管创建了多少个封闭式类类型的实例，对于关联的应用程序域来说，在任何时候静态字段都只会有一个副本。

例如：

```
class C<V>
{
   static int count = 0;

   public C() {
      count++;
   }

   public static int Count {
      get { return count; }
   }
}


class Application
{
   static void Main() {
      C<int> x1 = new C<int>();
      Console.WriteLine(C<int>.Count);    // Prints 1

      C<double> x2 = new C<double>();
      Console.WriteLine(C<int>.Count);    // Prints 1

      C<int> x3 = new C<int>();
      Console.WriteLine(C<int>.Count);    // Prints 2
   }
}
```

实例字段属于某个实例。具体而言，类的每个实例都包含了该类的所有实例字段的一个单独的集合。

若用 `E.M` 形式的 *member-access*（第 7.6.4 节）来引用一个字段，如果 `M` 是静态字段，则 `E` 必须表示含有 `M` 的一个类型，但如果 `M` 是实例字段，则 E 必须表示一个含有 `M` 的类型的某个实例。

第 10.3.7 节对静态成员和实例成员之间的差异进行了进一步讨论。

10.5.2 只读字段

当 *field-declaration* 中含有 `readonly` 修饰符时，该声明所引入的字段为只读字段 (readonly field)。给只读字段的直接赋值只能作为声明的组成部分出现，或在同一类中的实例构造函数或静态构造函数中出现。（在这些上下文中，只读字段可以被多次赋值。）准确地说，只在下列上下文中允许对 `readonly` 字段进行直接赋值：

- 在用于引入该字段的 *variable-declarator* 中（通过在声明中包括一个 *variable-initializer*）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 对于实例字段，在包含字段声明的类的实例构造函数中；对于静态字段，在包含字段声明的类的静态构造函数中。这些也是可以将 readonly 字段作为 out 或 ref 形参进行传递的仅有的上下文。

在其他任何上下文中，试图对 readonly 字段进行赋值或将它作为 out 或 ref 形参传递都会导致编译时错误。

10.5.2.1 对常量使用静态只读字段

如果需要一个具有常量值的符号名称，但该值的类型不允许在 const 声明中使用，或者无法在编译时计算出该值，则 static readonly 字段就可以发挥作用了。在下面的示例中

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

   private byte red, green, blue;

   public Color(byte r, byte g, byte b) {
      red = r;
      green = g;
      blue = b;
   }
}
```

Black、White、Red、Green 和 Blue 成员不能声明为 const 成员，这是因为在编译时无法计算它们的值。不过，改为将它们声明为 static readonly 能达到基本相同的效果。

10.5.2.2 常量和静态只读字段的版本控制

常量和只读字段具有不同的二进制版本控制语义。当表达式引用常量时，该常量的值在编译时获取，但是当表达式引用只读字段时，要等到运行时才获取该字段的值。请考虑一个包含两个单独程序的应用程序：

```
using System;

namespace Program1
{
   public class Utils
   {
      public static readonly int X = 1;
   }
}

namespace Program2
{
   class Test
   {
      static void Main() {
         Console.WriteLine(Program1.Utils.X);
      }
   }
}
```

Program1 和 Program2 命名空间表示两个单独编译的程序。由于 Program1.Utils.X 声明为静态只读字段，因此 Console.WriteLine 语句要输出的值在编译时是未知的，直到在运行时才能获取。因此，如果更改了 X 的值并重新编译 Program1，则 Console.WriteLine 语句会输出新值，即使未重新编译 Program2 也是如此。但如果 X 为常量，则 X 的值已在编译 Program2 时获取，则该值仍不会受到 Program1 中更改的影响，除非重新编译 Program2。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.5.3 可变字段

当*field-declaration*中含有`volatile`修饰符时，该声明引入的字段为可变字段(volatile field)。

由于采用了优化技术（它会重新安排指令的执行顺序），在多线程的程序运行环境下，如果不采取同步（如由*lock-statement*（第8.12节）所提供的）控制手段，则对于非可变字段的访问可能会导致意外的和不可预见的结果。这些优化可以由编译器、运行时系统或硬件执行。但是，对于可变字段，优化时的这种重新排序必须遵循以下规则：

- 读取一个可变字段称为可变读取(volatile read)。可变读取具有“获取语义”；也就是说，按照指令序列，所有排在可变读取之后的对内存的引用，在执行时也一定排在它的后面。
- 写入一个可变字段称为可变写入(volatile write)。可变写入具有“释放语义”；也就是说，按照指令序列，所有排在可变写入之前的对内存的引用，在执行时也一定排在它的前面。

这些限制能确保所有线程都会观察到由其他任何线程所执行的可变写入（按照原来安排的顺序）。一个遵循本规范的实现并非必须做到：使可变写入的执行顺序，在所有正在执行的线程看来都是一样的。可变字段的类型必须是下列类型中的一种：

- *reference-type*。
- 类型`byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`char`、`float`、`bool`、`System.IntPtr`或`System.UIntPtr`。
- 枚举基类型为`byte`、`sbyte`、`short`、`ushort`、`int`或`uint`的*enum-type*。

下面的示例

```
using System;
using System.Threading;

class Test
{
   public static int result;
   public static volatile bool finished;

   static void Thread2() {
      result = 143;
      finished = true;
   }

   static void Main() {
      finished = false;

      // Run Thread2() in a new thread
      new Thread(new ThreadStart(Thread2)).Start();

      // Wait for Thread2 to signal that it has a result by setting
      // finished to true.
      for (;;) {
         if (finished) {
            Console.WriteLine("result = {0}", result);
            return;
         }
      }
   }
}
```

产生下列输出：

```
result = 143
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在本示例中，方法 Main 将启动一个新线程，该线程运行方法 Thread2。该方法将一个值存储在叫做 result 的非可变字段中，然后将 true 存储在可变字段 finished 中。主线程将等待字段 finished 被设置为 true，然后读取字段 result。由于已将 finished 声明为 volatile，主线程必须从字段 result 读取值 143。如果字段 finished 未被声明为 volatile，则存储 finished 之后，主线程可看到存储 result，因此主线程从字段 result 读取值 0。将 finished 声明为 volatile 字段可以防止任何此类不一致性。

10.5.4 字段初始化

字段（无论是静态字段还是实例字段）的初始值都是字段的类型的默认值（第 5.2 节）。在此默认初始化发生之前不可能看到字段的值，因此字段永远不会是“未初始化的”。下面的示例

```
using System;

class Test
{
   static bool b;
   int i;

   static void Main() {
      Test t = new Test();
      Console.WriteLine("b = {0}, i = {1}", b, t.i);
   }
}
```

产生输出

```
b = False, i = 0
```

这是因为 b 和 i 都被自动初始化为默认值。

10.5.5 变量初始值设定项

字段声明可能包括 *variable-initializer*。对于静态字段，变量初始值设定项相当于在类初始化期间执行的赋值语句。对于实例字段，变量初始值设定项相当于创建类的实例时执行的赋值语句。

下面的示例

```
using System;

class Test
{
   static double x = Math.Sqrt(2.0);
   int i = 100;
   string s = "Hello";

   static void Main() {
      Test a = new Test();
      Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
   }
}
```

产生输出

```
x = 1.4142135623731, i = 100, s = Hello
```

这是因为对 x 的赋值发生在静态字段初始值设定项执行时，而对 i 和 s 的赋值发生在实例字段初始值设定项执行时。

第 10.5.4 节中描述的默认值初始化对所有字段都发生，包括具有变量初始值设定项的字段。因此，当初始化一个类时，首先将该类中的所有静态字段初始化为它们的默认值，然后以文本顺序执行各个静态字

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

段初始值设定项。与此类似，创建类的一个实例时，首先将该实例中的所有实例字段初始化为它们的默认值，然后以文本顺序执行各个实例字段初始值设定项。

在具有变量初始值设定项的静态字段处于默认值状态时，也有可能访问它们。但是，为了培养良好的编程风格，强烈建议不要这么做。下面的示例

```
using System;

class Test
{
   static int a = b + 1;
   static int b = a + 1;

   static void Main() {
      Console.WriteLine("a = {0}, b = {1}", a, b);
   }
}
```

演示此行为。尽管 a 和 b 的定义是循环的，此程序仍是有效的。它产生以下输出：

```
a = 1, b = 2
```

这是因为静态字段 a 和 b 在它们的初始值设定项执行之前被初始化为 0（int 的默认值）。当 a 的初始值设定项运行时，b 的值为零，所以 a 被初始化为 1。当 b 的初始值设定项运行时，a 的值已为 1，所以 b 被初始化为 2。

10.5.5.1 静态字段初始化

类的静态字段变量初始值设定项对应于一个赋值序列，这些赋值按照它们在类声明中出现的文本顺序执行。如果类中存在静态构造函数（第 10.12 节）\r\h，则在静态构造函数即将执行之前，将执行静态字段初始值设定项。否则，静态字段初始值设定项在第一次使用该类的静态字段之前先被执行，但实际执行时间依赖于具体的实现。下面的示例

```
using System;

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

   public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}

class A
{
   public static int X = Test.F("Init A");
}

class B
{
   public static int Y = Test.F("Init B");
}
```

或者产生如下输出：

```
Init A
Init B
1 1
```

或者产生如下输出：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
Init B
Init A
1 1
```

这是因为 X 的初始值设定项和 Y 的初始值设定项的执行顺序无法预先确定，上述两种顺序都有可能发生；唯一能够确定的是：它们一定会在对那些字段的引用之前发生。但是，下面的示例：

```
using System;

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

   public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}

class A
{
   static A() {}

   public static int X = Test.F("Init A");
}

class B
{
   static B() {}

   public static int Y = Test.F("Init B");
}
```

所产生的输出必然是：

```
Init B
Init A
1 1
```

这是因为关于何时执行静态构造函数的规则（在第 10.12 节中定义）进行了这样的规定：B 的静态构造函数（以及 B 的静态字段初始值设定项）必须在 A 的静态构造函数和字段初始值设定项之前运行。

10.5.5.2 实例字段初始化

类的实例字段变量初始值设定项对应于一个赋值序列，它在当控制进入该类的任一个实例构造函数（第 10.11.1 节）时立即执行。这些变量初始值设定项按它们出现在类声明中的文本顺序执行。第 10.11 节中对类实例的创建和初始化过程进行了进一步描述。

实例字段的变量初始值设定项不能引用正在创建的实例。因此，在变量初始值设定项中引用 this 是编译时错误，同样，在变量初始值设定项中通过 *simple-name* 引用任何一个实例成员也是一个编译时错误。在下面的示例中

```
class A
{
   int x = 1;
   int y = x + 1;    // Error, reference to instance member of this
}
```

y 的变量初始值设定项导致编译时错误，原因是它引用了正在创建的实例的成员。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.6 方法

方法 (method) 是一种成员，用于实现可由对象或类执行的计算或操作。方法是使用 *method-declaration* 来声明的：

method-declaration:
    method-header *method-body*

method-header:
    attributes$_{opt}$ *method-modifiers*$_{opt}$ `partial`$_{opt}$ *return-type* *member-name* *type-parameter-list*$_{opt}$
        `(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$

method-modifiers:
    method-modifier
    method-modifiers *method-modifier*

method-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`
    `async`

return-type:
    type
    `void`

member-name:
    identifier
    interface-type `.` *identifier*

method-body:
    block
    `;`

method-declaration 可包含一组 *attributes*（第 17 章）和一个由四个访问修饰符构成的有效组合（第 10.3.5 节），`new`（第 10.3.4 节）、`static`（第 10.6.2 节）、`virtual`（第 10.6.3 节）、`override`（第 10.6.4 节）、`sealed`（第 10.6.5 节）、`abstract`（第 10.6.6 节）和 `extern`（第 10.6.7 节）。

如果以下所有条件为真，则所述的声明就具有一个有效的修饰符组合：

- 该声明包含一个由访问修饰符（第 10.3.5 节）组成的有效组合。
- 该声明中所含的修饰符没有彼此相同的。
- 该声明最多包含下列修饰符中的一个：`static`、`virtual` 和 `override`。
- 该声明最多包含下列修饰符中的一个：`new` 和 `override`。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果声明中包含 abstract 修饰符，则该声明不包含下列任何修饰符：static、virtual、sealed 或 extern。
- 如果声明中包含 private 修饰符，则该声明不包含下列任何修饰符：virtual、override 或 abstract。
- 如果声明包含 sealed 修饰符，则该声明还包含 override 修饰符。
- 如果声明中包含 partial 修饰符，则该声明不包含下列任何修饰符：new、public、protected、internal、private、virtual、sealed、override、abstract 或 extern。

具有“async”修饰符的方法是一种异步函数，并遵循第**错误!未找到引用源。**节中描述的规则。

方法声明的 *return-type* 用于指定由该方法计算和返回的值的类型。如果方法不返回一个值，则它的 *return-type* 为 void。如果声明包含 partial 修饰符，则返回类型必须为 void。

member-name 用于指定方法的名称。除非方法是一个显式接口成员的实现（第 13.4.1 节），否则 *member-name* 仅是一个 *identifier*。对于显式接口成员实现，*member-name* 由一个 *interface-type* 并后接一个“.”和一个 *identifier* 组成。

可选的 *type-parameter-list* 用于指定方法的类型形参（第 10.1.3 节）。如果指定了 *type-parameter-list*，则方法是泛型方法 (generic method)。如果方法具有 extern 修饰符，则不能指定 *type-parameter-list*。

可选的 *formal-parameter-list* 用于指定方法的形参（第 10.6.1 节）。

可选的 *type-parameter-constraints-clauses* 用于指定对各个类型形参（第 10.1.5 节）的约束，仅在同时提供了 *type-parameter-list* 的情况下才可以指定，该方法没有 override 修饰符。

return-type 和在方法的 *formal-parameter-list* 中引用的各个类型必须至少具有和方法本身相同的可访问性（第 3.5.4 节）。

对于 abstract 和 extern 方法，*method-body* 只由一个分号组成。对于 partial 方法，*method-body* 由一个分号或由一个 *block* 组成。对于所有其他方法，*method-body* 由一个 *block* 组成，该块用于指定在调用方法时要执行哪些语句。

如果 *method-body* 包含分号，则声明可能不包含 async 修饰符。

一个方法的名称、类型形参列表和形参表定义了该方法的签名（第 3.6 节）。准确地说，一个方法的签名由它的名称、类型形参的数目以及它的形参的数目、修饰符和类型组成。为此，出现在形参类型中的方法的任何类型形参都不按名称标识，而是按其在方法的类型实参列表中的序号位置标识。返回类型不是方法签名的一部分，类型形参或形参的名称也不是。

方法的名称必须不同于在同一个类中声明的所有其他非方法的名称。此外，必须不同于在同一类中声明的所有其他方法的签名，并且在同一类中声明的两种方法的签名不能只有 ref 和 out 不同。

方法的 *type-parameter* 作用于整个 *method-declaration* 范围，并且可在整个该范围中用于构成 *return-type*、*method-body* 和 *type-parameter-constraints-clauses*（但是不包括 *attributes*）中的类型。

所有形参和类型形参都不能同名。

10.6.1 方法形参

一个方法的形参（如果有）是由该方法的 *formal-parameter-list* 来声明的。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

formal-parameter-list:
    fixed-parameters
    fixed-parameters , *parameter-array*
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters , *fixed-parameter*

fixed-parameter:
    attributes$_{opt}$ *parameter-modifier*$_{opt}$ *type* *identifier* *default-argument*$_{opt}$

default-argument:
    = *expression*

parameter-modifier:
    `ref`
    `out`
    `this`

parameter-array:
    attributes$_{opt}$ `params` *array-type* *identifier*

形参表包含一个或多个由逗号分隔的形参，其中只有最后一个形参才可以是*parameter-array*。

fixed-parameter 包含一组可选 *attributes*（第 17 节）、一个可选的 `ref`、`out` 或 `this` 修饰符、一个 *type*、一个 *identifier* 以及一个可选的 *default-argument*。每个 *fixed-parameter* 均声明了一个形参，指定了该形参的名称及其所属的类型。`this` 修饰符将方法指定为扩展方法，仅允许在静态方法的第一个形参上使用该修饰符。第 10.6.9 节中对扩展方法进行了进一步描述。

具有 *default-argument* 的 *fixed-parameter* 称为可选形参 (optional parameter)，没有 *default-argument* 的 *fixed-parameter* 称为必选形参 (required parameter)。在 *formal-parameter-list* 中，必选形参不能出现在可选形参之后。

`ref` 或 `out` 形参不能有 *default-argument*。*default-argument* 中的 *expression* 必须为下列各项之一：

- *constant-expression*
- `new S()` 形式的表达式，其中 `S` 是值类型
- `default(S)` 形式的表达式，其中 `S` 是值类型

expression 必须能够由标识或不可为 null 的转换隐式转换为形参的类型。

如果实现分步方法声明（第 10.2.7 节）、显式接口成员实现（第 13.4.1 节）或单参数索引器声明（第 10.9 节）中出现可选参数，编译器应发出警告，因为这些成员不能以允许忽略实参的方式调用。

parameter-array 包含一组可选的 *attributes*（第 17 章）、一个 `params` 修饰符、一个 *array-type* 和一个 *identifier*。形参数组声明单个具有给定名称且属于给定数组类型的形参。形参数组中的 *array-type* 必须是一维数组类型（第 12.1 节）。在方法调用中，形参数组可以用单个给定数组类型的实参来指定，也可以用零个或多个该数组元素类型的实参来指定。第 10.6.1.4 节中对形参数组进行了进一步描述。

parameter-array 可以出现在可选形参之后，但不能具有默认值，如果忽略 *parameter-array* 的实参，则会创建一个空数组。

以下示例说明不同种类的形参：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
public void M(
   ref int      i,
   decimal      d,
   bool         b = false,
   bool?        n = false,
   string       s = "Hello",
   object       o = null,
   T            t = default(T),
   params int[] a
) { }
```

在 M 的 *formal-parameter-list* 中，i 是必需的引用参数，d 是必需的值参数，b、s、o 和 t 是可选的值参数，a 是参数数组。

方法声明为所声明的形参、类型形参和局部变量创建单独的声明空间。该方法的类型形参表与形参表和在该方法的 *block* 中的局部变量声明将它们所声明的名称提供给此声明空间。如果方法声明空间的两个成员具有相同的名称，则会发生错误。如果方法声明空间和嵌套的声明空间的局部变量声明空间包含同名的元素，则会发生错误。

执行一个方法调用（第 7.6.5.1 节）时，创建关于该方法的形参和局部变量的一个副本（仅适用于本次调用），而该调用所提供的实参列表则用于将所含的值或变量引用赋给新创建的形参。在方法的 *block* 内，形参可以在 *simple-name* 表达式（第 7.6.2 节）中由它们的标识符引用。

有四种形参：

- 值形参，声明时不带任何修饰符。
- 引用形参，用 ref 修饰符声明。
- 输出形参，用 out 修饰符声明。
- 形参数组，用 params 修饰符声明。

如第 3.6 节中所述，ref 和 out 修饰符是方法签名的组成部分，但 params 修饰符不是。

10.6.1.1 值参数

声明时不带修饰符的形参是值形参。一个值形参对应于一个局部变量，只是它的初始值来自该方法调用所提供的相应实参。

当形参是值形参时，方法调用中的对应实参必须是表达式，并且它的类型可以隐式转换（第 0 节）为形参的类型。

允许方法将新值赋给值参数。这样的赋值只影响由该值形参表示的局部存储位置，而不会影响在方法调用时由调用方给出的实参。

10.6.1.2 引用参数

引用形参是用 ref 修饰符声明的形参。与值形参不同，引用形参并不创建新的存储位置。相反，引用形参表示的存储位置恰是在方法调用中作为实参给出的那个变量所表示的存储位置。

当形参为引用形参时，方法调用中的对应实参必须由关键字 ref 并后接一个与形参类型相同的 *variable-reference*（第 5.3.3 节）组成。变量在可以作为引用形参传递之前，必须先明确赋值。

在方法内部，引用形参始终被认为是明确赋值的。

声明为迭代器（第 10.14 节）的方法不能有引用形参。

下面的示例

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
using System;

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

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

产生输出

```
i = 2, j = 1
```

对于对 Swap 的调用（在 Main 中），x 表示 i，y 表示 j。因此，该调用具有交换 i 和 j 的值的效果。

在采用引用形参的方法中，多个名称可能表示同一存储位置。在下面的示例中

```
class A
{
   string s;

   void F(ref string a, ref string b) {
      s = "One";
      a = "Two";
      b = "Three";
   }

   void G() {
      F(ref s, ref s);
   }
}
```

在 G 中调用 F 时，分别为 a 和 b 传递了一个对 s 的引用。因此，对于该调用，名称 s、a 和 b 全都引用同一存储位置，并且三个赋值全都修改了同一个实例字段 s。

10.6.1.3 输出形参

用 out 修饰符声明的形参是输出形参。类似于引用形参，输出形参不创建新的存储位置。相反，输出形参表示的存储位置恰是在该方法调用中作为实参给出的那个变量所表示的存储位置。

当形参为输出形参时，方法调用中的相应实参必须由关键字 out 并后接一个与形参类型相同的 *variable-reference*（第 5.3.3 节）组成。变量在可以作为输出形参传递之前不一定需要明确赋值，但是在将变量作为输出形参传递的调用之后，该变量被认为是明确赋值的。

在方法内部，与局部变量相同，输出形参最初被认为是未赋值的，因而必须在使用它的值之前明确赋值。

在方法返回之前，该方法的每个输出形参都必须明确赋值。

声明为分部方法（第 10.2.7 节）或迭代器（第 10.14 节）的方法不能有输出形参。

输出形参通常用在需要产生多个返回值的方法中。例如：

```
using System;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
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);
   }
}
```

此例产生输出：

```
c:\Windows\System\
hello.txt
```

请注意，dir 和 namename 变量在它们被传递给 SplitPath 之前可以是未赋值的，而它们在调用之后就被认为是明确赋值的了。

10.6.1.4 形参数组

用 params 修饰符声明的形参是形参数组。如果形参表包含一个形参数组，则该形参数组必须位于该列表的最后而且它必须是一维数组类型。例如，类型 string[] 和 string[][] 可用作形参数组的类型，但类型 string[,] 不能。不能将 params 修饰符与 ref 和 out 修饰符组合起来使用。

在一个方法调用中，允许以下列两种方式之一来为形参数组指定对应的实参：

- 为形参数组提供的实参可以是单个表达式，它的类型可以隐式转换（第 0 节）为形参数组的类型。在此情况下，形参数组的作用与值形参完全一样。
- 此外，调用时可以为形参数组指定零个或多个实参，其中每个实参都是一个表达式，它的类型可隐式转换（第 0 节）为形参数组的元素的类型。在此情况下，调用会创建一个该形参数组类型的实例，其所含的元素个数等于给定的实参个数，再用给定的实参值初始化此数组实例的每个元素，然后将新创建的数组实例用作实参。

除了允许在调用中使用可变数量的实参，形参数组与同一类型的值形参（第 10.6.1.1 节）完全等效。

下面的示例

```
using System;

class Test
{
   static void F(params int[] args) {
      Console.Write("Array contains {0} elements:", args.Length);
      foreach (int i in args)
         Console.Write(" {0}", i);
      Console.WriteLine();
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   static void Main() {
      int[] arr = {1, 2, 3};
      F(arr);
      F(10, 20, 30, 40);
      F();
   }
}
```

产生输出

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

F 的第一次调用只是将数组 a 作为值形参传递。F 的第二次调用自动创建一个具有给定元素值的四元素 int[] 并将该数组实例作为值形参传递。与此类似，F 的第三次调用创建一个零元素的 int[] 并将该实例作为值形参传递。第二次和第三次调用完全等效于编写下列代码：

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

执行重载决策时，具有形参数组的方法的正常形式或扩展形式（第 7.5.3.1 节）都是适用的。只有在方法的正常形式不适用，并且在同一类型中尚未声明与方法的扩展形式具有相同签名的适用方法时，上述的方法扩展形式才可供选用。

下面的示例

```
using System;

class Test
{
   static void F(params object[] a) {
      Console.WriteLine("F(object[])");
   }

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

   static void F(object a0, object a1) {
      Console.WriteLine("F(object,object)");
   }

   static void Main() {
      F();
      F(1);
      F(1, 2);
      F(1, 2, 3);
      F(1, 2, 3, 4);
   }
}
```

产生输出

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

在该示例中，在同一个类中，已经声明了两个常规方法，它们的签名与具有形参数组的那个方法的扩展形式相同。因此，在执行重载决策时不考虑这些扩展形式，因而第一次和第三次方法调用将选择常规方法。当在某个类中声明了一个具有形参数组的方法时，同时再声明一些与该方法的扩展形式具有相同的

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

签名的常规方法，这种情况比较常见。这样做可以避免为数组配置内存空间（若调用具有形参数组的方法的扩展形式，则无法避免）。

当形参数组的类型为 object[] 时，在方法的正常形式和单个 object 形参的扩展形式之间可能产生潜在的多义性。产生此多义性的原因是 object[] 本身可隐式转换为 object。然而，此多义性并不会造成任何问题，这是因为可以在需要时通过插入一个强制转换来解决它。

下面的示例

```
using System;

class Test
{
   static void F(params object[] args) {
      foreach (object o in args) {
         Console.Write(o.GetType().FullName);
         Console.Write(" ");
      }
      Console.WriteLine();
   }

   static void Main() {
      object[] a = {1, "Hello", 123.456};
      object o = a;
      F(a);
      F((object)a);
      F(o);
      F((object[])o);
   }
}
```

产生输出

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

在 F 的第一次和最后一次调用中，F 的正常形式是适用的，这是因为存在一个从实参类型到形参类型的转换（这里，两者都是 object[] 类型）。因此，重载决策选择 F 的正常形式，而且将该实参作为常规的值形参传递。在第二次和第三次调用中，F 的正常形式不适用，这是因为不存在从实参类型到形参类型的转换（类型 object 不能隐式转换为类型 object[]）。但是，F 的扩展形式是适用的，因此重载决策选择它。因此，这个调用都创建了一个具有单个元素的、类型为 object[] 的数组，并且用给定的实参值（它本身是对一个 object[] 的引用）初始化该数组的唯一元素。

10.6.2 静态方法和实例方法

若一个方法声明中含有 static 修饰符，则称该方法为静态方法。若其中没有 static 修饰符时，则称该方法为实例方法。

静态方法不对特定实例进行操作，在静态方法中引用 this 会导致编译时错误。

实例方法对类的某个给定的实例进行操作，而且可以用 this（第 7.6.7 节）来访问该实例。

在 E.M 形式的 *member-access*（第 7.6.4 节）中引用一个方法时，如果 M 是静态方法，则 E 必须表示含有 M 的一个类型，而如果 M 是实例方法，则 E 必须表示含有 M 的类型的一个实例。

第 10.3.7 节对静态成员和实例成员之间的差异进行了进一步讨论。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.6.3 虚方法

若一个实例方法的声明中含有 virtual 修饰符，则称该方法为虚方法。若其中没有 virtual 修饰符，则称该方法为非虚方法。

非虚方法的实现是一成不变的：无论该方法是在声明它的类的实例上调用还是在派生类的实例上调用，实现均相同。与此相反，虚方法的实现可以由派生类取代。取代所继承的虚方法的实现的过程称为重写 (overriding) 该方法（第 10.6.4 节）。

在虚方法调用中，该调用所涉及的那个实例的运行时类型 (run-time type) 确定了要被调用的究竟是该方法的哪一个实现。在非虚方法调用中，相关的实例的编译时类型 (compile-time type) 是决定性因素。准确地说，当在具有编译时类型 C 和运行时类型 R 的实例（其中 R 为 C 或者从 C 派生的类）上用实参列表 A 调用名为 N 的方法时，调用按下述规则处理：

- 首先，将重载决策应用于 C、N 和 A，以从在 C 中声明的和由 C 继承的方法集中选择一个特定的方法 M。第 7.6.5.1 节中对此进行了介绍。
- 然后，如果 M 为非虚方法，则调用 M。
- 否则（M 为虚方法），就会调用就 R 而言 M 的派生程度最大的那个实现。

对于在一个类中声明的或者由类继承的每个虚方法，存在一个就该类而言的方法的派生程度最大的实现 (most derived implementation)。就类 R 而言虚方法 M 的派生度最大的实现按下述规则确定：

- 如果 R 包含 M 的引入 virtual 声明，则这是 M 的派生程度最大的实现。
- 否则，如果 R 包含关于 M 的 override，则这是 M 的派生程度最大的实现。
- 否则，就 R 而言 M 的派生程度最大的实现与就 R 的直接基类而言 M 的派生程度最大的实现相同。

下列实例阐释虚方法和非虚方法之间的区别：

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

在该示例中，A 引入一个非虚方法 F 和一个虚方法 G。类 B 引入一个新的非虚方法 F，从而隐藏了继承的 F，并且还重写了继承的方法 G。此例产生输出：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
A.F
B.F
B.G
B.G
```

请注意，语句 a.G() 调用 B.G，而不调用 A.G。这是因为，对调用哪个实际方法实现起决定作用的是该实例的运行时类型（即 B），而不是该实例的编译时类型（即 A）。

由于方法可以隐藏继承来的方法，因此同一个类中可以包含若干个具有相同签名的虚方法。这不会造成多义性问题，因为除派生程度最大的那个方法外，其他方法都被隐藏起来了。在下面的示例中

```
using System;

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

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

class C: B
{
   new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
   public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
   static void Main() {
      D d = new D();
      A a = d;
      B b = d;
      C c = d;
      a.F();
      b.F();
      c.F();
      d.F();
   }
}
```

C 类和 D 类包含两个具有相同签名的虚方法：一个是 A 引入的，另一个是 C 引入的。但是，由 C 引入的方法隐藏了从 A 继承的方法。因此，D 中的重写声明所重写的是由 C 引入的方法，D 不可能重写由 A 引入的方法。此例产生输出：

```
B.F
B.F
D.F
D.F
```

请注意，通过访问 D 的实例（借助一个派生程度较小的类型，它的方法没有被隐藏起来），可以调用被隐藏的虚方法。

10.6.4 重写方法

若一个实例方法声明中含有 override 修饰符，则称该方法为重写方法 (override method)。重写方法用相同的签名重写所继承的虚方法。虚方法声明用于引入新方法，而重写方法声明则用于使现有的继承虚方法专用化（通过提供该方法的新实现）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由 override 声明所重写的那个方法称为已重写了的基方法 (overridden base method)。对于在类 C 中声明的重写方法 M，已重写的基方法是通过检查 C 的各个基类类型来确定的，该检查过程如下：从 C 的直接基类类型开始检查，然后依次检查每个后续的直接基类类型，直到在给定的基类类型中至少找到一个在用类型实参替换后与 M 具有相同签名的可访问方法。为了查找已重写了的基方法，可访问方法可以这样来定义：如果一个方法是 public、是 protected、是 protected internal，或者是 internal 并且与 C 声明在同一程序中，则认为它是可访问的。

除非下列所有项对于一个重写声明皆为真，否则将会出现编译时错误：

- 可以按照上面描述的规则找到一个已重写了的基方法。
- 只有一个此类重写的基方法。此限制仅在基类类型是构造类型时（在这种情况下，用类型实参替换会使两个方法的签名相同）才有效。
- 该已重写了的基方法是一个虚的、抽象或重写方法。换句话说，已重写了的基方法不能是静态或非虚方法。
- 已重写了的基方法不是密封方法。
- 重写方法和已重写了的基方法具有相同的返回类型。
- 重写声明和已重写了的基方法具有相同的声明可访问性。换句话说，重写声明不能更改所对应的虚方法的可访问性。但是，如果已重写的基方法是 protected internal，并且声明它的程序集不是包含重写方法的程序集，则重写方法声明的可访问性必须是 protected。
- 重写声明不指定 type-parameter-constraints-clauses，而是从重写的基方法继承约束。请注意，在重写方法中作为类型形参的约束可能会被继承约束中的类型实参替换。在显式指定约束（如值类型或密封类型）时，这可能导致约束不合法。

下面的示例演示重写规则如何对泛型类起作用：

```
abstract class C<T>
{
   public virtual T F() {...}
   public virtual C<T> G() {...}
   public virtual void H(C<T> x) {...}
}
class D: C<string>
{
   public override string F() {...}          // Ok
   public override C<string> G() {...}       // Ok
   public override void H(C<T> x) {...}      // Error, should be C<string>
}
class E<T,U>: C<U>
{
   public override U F() {...}               // Ok
   public override C<U> G() {...}            // Ok
   public override void H(C<T> x) {...}      // Error, should be C<U>
}
```

重写声明可以使用 *base-access*（第 7.6.8 节）访问已重写了的基方法。在下面的示例中

```
class A
{
   int x;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   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);
   }
}
```

在 B 中调用 base.PrintFields() 时将调用 A 中声明的 PrintFields 方法。*base-access* 禁用了虚调用机制，它只是将那个基方法视为非虚方法。如果将 B 中的调用改写为 ((A)this).PrintFields()，它将递归调用在 B 中声明的 PrintFields 方法，而不调用在 A 中声明的该方法，因为 PrintFields 是虚方法，而且 ((A)this) 的运行时类型为 B。

只有在包含了 override 修饰符时，一个方法才能重写另一个方法。在所有其他情况下，声明一个与继承了的方法具有相同签名的方法只会使那个被继承的方法隐藏起来。在下面的示例中

```
class A
{
   public virtual void F() {}
}

class B: A
{
   public virtual void F() {}    // Warning, hiding inherited F()
}
```

B 中的 F 方法不包含 override 修饰符，因此不重写 A 中的 F 方法。相反，B 中的 F 方法隐藏 A 中的方法，并且由于该声明中没有包含 new 修饰符，从而会报告一个警告。

在下面的示例中

```
class A
{
   public virtual void F() {}
}

class B: A
{
   new private void F() {}       // Hides A.F within body of B
}

class C: B
{
   public override void F() {}   // Ok, overrides A.F
}
```

B 中的 F 方法将隐藏从 A 中继承的虚 F 方法。由于 B 中的新 F 具有私有访问权限，它的范围只包括 B 的类体而没有延伸到 C。因此，允许 C 中的 F 声明重写从 A 继承的 F。

10.6.5 密封方法

当实例方法声明包含 sealed 修饰符时，称该方法为密封方法 (sealed method)。如果实例方法声明包含 sealed 修饰符，则它必须也包含 override 修饰符。使用 sealed 修饰符可以防止派生类进一步重写该方法。

下面的示例

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
using System;
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
   public virtual void G() {
      Console.WriteLine("A.G");
   }
}
class B: A
{
   sealed override public void F() {
      Console.WriteLine("B.F");
   }
   override public void G() {
      Console.WriteLine("B.G");
   }
}
class C: B
{
   override public void G() {
      Console.WriteLine("C.G");
   }
}
```

类 B 提供两个重写方法：一个是带有 sealed 修饰符的 F 方法，另一个是没有 sealed 修饰符的 G 方法。通过使用 sealed modifier，B 就可以防止 C 进一步重写 F。

10.6.6 抽象方法

当实例方法声明包含 abstract 修饰符时，称该方法为抽象方法 (abstract method)。虽然抽象方法同时隐含为虚方法，但是它不能有 virtual 修饰符。

抽象方法声明引入一个新的虚方法，但不提供该方法的实现。相反，非抽象类的派生类需要重写该方法以提供它们自己的实现。由于抽象方法不提供任何实际实现，因此抽象方法的 *method-body* 只由一个分号组成。

只允许在抽象类（第 10.1.1.1 节）中使用抽象方法声明。

在下面的示例中

```
public abstract class Shape
{
   public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawEllipse(r);
   }
}
public class Box: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawRect(r);
   }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

Shape 类定义了一个可以绘制自身的几何形状对象的抽象概念。Paint 方法是抽象的，这是因为没有有意义的默认实现。Ellipse 和 Box 类是具体的 Shape 实现。由于这些类是非抽象的，因此要求它们重写 Paint 方法并提供实际实现。

如果一个 *base-access*（第 7.6.8 节）引用的是一个抽象方法，则会导致编译时错误。在下面的示例中

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

class B: A
{
   public override void F() {
      base.F();                  // Error, base.F is abstract
   }
}
```

调用 base.F() 导致了编译时错误，原因是它引用了抽象方法。

在一个抽象方法声明中可以重写虚方法。这使一个抽象类可以强制从它的派生类重新实现该方法，并使该方法的原始实现不再可用。在下面的示例中

```
using System;

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

abstract class B: A
{
   public abstract override void F();
}

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

类 A 声明一个虚方法，类 B 用一个抽象方法重写此方法，而类 C 重写该抽象方法以提供它自己的实现。

10.6.7 外部方法

当方法声明包含 extern 修饰符时，称该方法为外部方法 (external method)。外部方法是在外部实现的，编程语言通常是使用 C# 以外的语言。由于外部方法声明不提供任何实际实现，因此外部方法的 *method-body* 只由一个分号组成。外部方法不可以是泛型。

extern 修饰符通常与 DllImport 属性（第 17.5.1 节）一起使用，从而使外部方法可以由 DLL（动态链接库）实现。执行环境可以支持其他用来提供外部方法实现的机制。

当外部方法包含 DllImport 属性时，该方法声明必须同时包含一个 static 修饰符。此示例说明 extern 修饰符和 DllImport 属性的使用：

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Path
{
   [DllImport("kernel32", SetLastError=true)]
   static extern bool CreateDirectory(string name, SecurityAttribute sa);

   [DllImport("kernel32", SetLastError=true)]
   static extern bool RemoveDirectory(string name);

   [DllImport("kernel32", SetLastError=true)]
   static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

   [DllImport("kernel32", SetLastError=true)]
   static extern bool SetCurrentDirectory(string name);
}
```

10.6.8 分部方法

若一个方法声明中含有 partial 修饰符，则称该方法为分部方法 (partial method)。只能将分部方法声明为分部类型（第 10.2 节）的成员，而且要受多种约束。第 10.2.7 节中对分部方法进行了进一步描述。

10.6.9 扩展方法

当方法的第一个形参包含 this 修饰符时，称该方法为扩展方法 (extension method)。只能在非泛型、非嵌套静态类中声明扩展方法。扩展方法的第一个形参不能带有除 this 之外的其他修饰符，而且形参类型不能是指针类型。

下面是一个声明两个扩展方法的静态类的示例：

```
public static class Extensions
{
   public static int ToInt32(this string s) {
      return Int32.Parse(s);
   }

   public static T[] Slice<T>(this T[] source, int index, int count) {
      if (index < 0 || count < 0 || source.Length – index < count)
         throw new ArgumentException();
      T[] result = new T[count];
      Array.Copy(source, index, result, 0, count);
      return result;
   }
}
```

扩展方法是常规静态方法。另外，如果它的包容静态类在范围之内，则可以使用实例方法调用语法（第 7.6.5.2 节）来调用扩展方法，同时将接收器表达式用作第一个实参。

下面的程序使用上面声明的扩展方法：

```
static class Program
{
   static void Main() {
      string[] strings = { "1", "22", "333", "4444" };
      foreach (string s in strings.Slice(1, 2)) {
         Console.WriteLine(s.ToInt32());
      }
   }
}
```

Slice 方法在 string[] 上可用，ToInt32 方法在字符串上可用，原因是它们都已声明为扩展方法。该程序的含义与下面使用普通静态方法调用的程序相同：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
static class Program
{
   static void Main() {
      string[] strings = { "1", "22", "333", "4444" };
      foreach (string s in Extensions.Slice(strings, 1, 2)) {
         Console.WriteLine(Extensions.ToInt32(s));
      }
   }
}
```

10.6.10 方法体

方法声明中的 *method-body* 或者由一个 *block*，或者由一个分号组成。

抽象和外部方法声明不提供方法实现，所以它们的方法体只包含一个分号。对于任何其他方法，方法体是一个块（第 8.2 节），它包含了在调用该方法时应执行的语句。

如果返回类型为 `void`，或者如果方法是异步方法并且返回类型为 `System.Threading.Tasks.Task`，则方法的*返回类型*为 `void`。否则，非异步方法的结果类型为其返回类型，且返回类型为 `System.Threading.Tasks.Task<T>` 的异步方法的结果类型为 *T*。

当方法的结果类型为 `void` 时，不允许该方法体中的 `return` 语句（第 8.9.4 节）指定表达式。如果一个 void 方法的方法体的执行正常完成（即控制自方法体的结尾离开），则该方法只是返回到它的当前调用方。

如果方法的返回类型不是 `void`，则方法体中的每个 `return` 语句都必须指定一个可隐式转换为结果类型的表达式。对于一个返回值的方法，其方法体的结束点必须是不可到达的。换句话说，在执行返回值的方法中，不允许控制自方法体的结尾离开。

在下面的示例中

```
class A
{
   public int F() {}       // Error, return value required
   public int G() {
      return 1;
   }
   public int H(bool b) {
      if (b) {
         return 1;
      }
      else {
         return 0;
      }
   }
}
```

返回值的 `F` 方法导致编译时错误，原因是控制可以超出方法体的结尾。`G` 和 `H` 方法是正确的，这是因为所有可能的执行路径都以一个指定返回值的 return 语句结束。

10.6.11 方法重载

第 7.5.2 节对方法重载决策规则进行了描述。

10.7 属性

属性 (property) 是一种用于访问对象或类的特征的成员。属性的示例包括字符串的长度、字体的大小、窗口的标题、客户的名称，等等。属性是字段的自然扩展，此两者都是具有关联类型的命名成员，而且访问字段和属性的语法是相同的。然而，与字段不同，属性不表示存储位置。相反，属性有访问器

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

(accessor)，这些访问器指定在它们的值被读取或写入时需执行的语句。因此属性提供了一种机制，它把读取和写入对象的某些特性与一些操作关联起来；甚至，它们还可以对此类特性进行计算。

属性是使用 *property-declaration* 声明的：

property-declaration:
    attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* `{` *accessor-declarations* `}`

property-modifiers:
    property-modifier
    property-modifiers *property-modifier*

property-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `static`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

member-name:
    identifier
    interface-type `.` *identifier*

property-declaration 可包含一组 *attributes*（第 17 章）和一个由四个访问修饰符构成的有效组合（第 10.3.5 节），`new`（第 10.3.4 节）、`static`（第 10.6.2 节）、`virtual`（第 10.6.3 节）、`override`（第 10.6.4 节）、`sealed`（第 10.6.5 节）、`abstract`（第 10.6.6 节）和 `extern`（第 10.6.7 节）。

在有效的修饰符组合方面，属性声明与方法声明（第 10.6 节）遵循相同的规则。

属性声明中的 *type* 用于指定该声明所引入的属性的类型，而 *member-name* 则指定该属性的名称。除非该属性是一个显式的接口成员实现，否则 *member-name* 就只是一个 *identifier*。对于显式接口成员实现（第 13.4.1 节），*member-name* 由一个 *interface-type* 并后接一个“`.`”和一个 *identifier* 组成。

属性的 *type* 必须至少与属性本身（第 3.5.4 节）具有同样的可访问性。

accessor-declarations（必须括在“`{`”和“`}`”标记中）声明属性的访问器（第 10.7.2 节）。访问器指定与属性的读取和写入相关联的可执行语句。

虽然访问属性的语法与访问字段的语法相同，但是属性并不归类为变量。因此，不能将属性作为 `ref` 或 `out` 参数传递。

属性声明包含 `extern` 修饰符时，称该属性为外部属性 (external property)。因为外部属性声明不提供任何实际的实现，所以它的每个 *accessor-declarations* 都仅由一个分号组成。

10.7.1 静态属性和实例属性

当属性声明包含 `static` 修饰符时，称该属性为静态属性 (static property)。当不存在 `static` 修饰符时，称该属性为实例属性 (instance property)。

静态属性不与特定实例相关联，因此在静态属性的访问器内引用 `this` 会导致编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

实例属性与类的一个给定实例相关联，并且该实例可以在属性的访问器内作为 this（第 7.6.7 节）来访问。

在 E.M 形式的 *member-access*（第 7.6.4 节）中引用属性时，如果 M 是静态属性，则 E 必须表示包含 M 的类型，如果 M 是实例属性，则 E 必须表示包含 M 的类型的一个实例。

第 10.3.7 节对静态成员和实例成员之间的差异进行了进一步讨论。

10.7.2 访问器

属性的 *accessor-declarations* 指定与读取和写入该属性相关联的可执行语句。

accessor-declarations:
get-accessor-declaration *set-accessor-declaration*$_{opt}$
set-accessor-declaration *get-accessor-declaration*$_{opt}$

get-accessor-declaration:
attributes$_{opt}$ *accessor-modifier*$_{opt}$ `get` *accessor-body*

set-accessor-declaration:
attributes$_{opt}$ *accessor-modifier*$_{opt}$ `set` *accessor-body*

accessor-modifier:
`protected`
`internal`
`private`
`protected internal`
`internal protected`

accessor-body:
block
`;`

访问器声明由一个 *get-accessor-declaration* 或一个 *set-accessor-declaration* 组成，或者由两者共同组成。每个访问器声明都包含标记 get 或 set，后接可选的 *accessor-modifier* 和 *accessor-body*。

accessor-modifier 的使用具有下列限制：

- *accessor-modifier* 不可用在接口中或显式接口成员实现中。
- 对于没有 override 修饰符的属性或索引器，仅当该属性或索引器同时带有 get 和 set 访问器时，才允许使用 *accessor-modifier*，并且只能用于其中的一个访问器。
- 对于包含 override 修饰符的属性或索引器，访问器必须匹配被重写的访问器的 *accessor-modifier*（如果存在）。
- *accessor-modifier* 声明的可访问性的限制性必须严格高于属性或索引器本身所声明的可访问性。准确地说：
  - 如果属性或索引器声明了 public 可访问性，则任何 *accessor-modifier* 可能为 protected internal、internal、protected 或 private。
  - 如果属性或索引器声明了 protected internal 可访问性，则任何 *accessor-modifier* 可能为 internal、protected 或 private。
  - 如果属性或索引器声明了 internal 或 protected 可访问性，则 *accessor-modifier* 必须为 private。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 如果属性或索引器声明了 `private` 可访问性，则任何 *accessor-modifier* 都不可使用。

对于 `abstract` 和 `extern` 属性，每个指定访问器的 *accessor-body* 只是一个分号。非 abstract、非 extern 属性可以是自动实现的属性 (automatically implemented property)，在这种情况下，必须给定 get 访问器和 set 访问器，而且其体均由一个分号组成（第 10.7.3 节）。对于其他任何非 abstract、非 extern 属性的访问器，*accessor-body* 是一个 *block*，它指定调用相应访问器时需执行的语句。

`get` 访问器相当于一个具有属性类型返回值的无形参方法。除了作为赋值的目标，当在表达式中引用属性时，将调用该属性的 `get` 访问器以计算该属性的值（第 7.1.1 节）。`get` 访问器体必须符合第 10.6.10 节中所描述的适用于值返回方法的规则。具体而言，`get` 访问器体中的所有 `return` 语句都必须指定一个可隐式转换为属性类型的表达式。另外，`get` 访问器的结束点必须是不可到达的。

`set` 访问器相当于一个具有单个属性类型值形参和 `void` 返回类型的方法。`set` 访问器的隐式参数始终命名为 `value`。当一个属性作为赋值（第 7.17 节）的目标，或者作为 `++` 或 `--` 运算符（第 7.6.9 节、第 7.7.5 节）的操作数被引用时，就会调用 `set` 访问器，所传递的实参（其值为赋值右边的值或者 `++` 或 `--` 运算符的操作数）将提供新值（第 7.17.1 节）。`set` 访问器体必须符合第 10.6.10 节中所描述的适用于 `void` 方法的规则。具体而言，不允许 `set` 访问器体中的 `return` 语句指定表达式。由于 `set` 访问器隐式具有名为 `value` 的形参，因此在 `set` 访问器中，若在局部变量或常量声明中出现该名称，则会导致编译时错误。

根据 `get` 和 `set` 访问器是否存在，属性可按下列规则分类：

- 同时包含 `get` 访问器和 `set` 访问器的属性称为读写 (read-write) 属性。
- 只具有 `get` 访问器的属性称为只读 (read-only) 属性。将只读属性作为赋值目标会导致编译时错误。
- 只具有 `set` 访问器的属性称为只写 (write-only) 属性。除了作为赋值的目标外，在表达式中引用只写属性是编译时错误。

在下面的示例中

```
public class Button: Control
{
   private string caption;

   public string Caption {
      get {
         return caption;
      }
      set {
         if (caption != value) {
            caption = value;
            Repaint();
         }
      }
   }

   public override void Paint(Graphics g, Rectangle r) {
      // Painting code goes here
   }
}
```

`Button` 控件声明了一个公共 `Caption` 属性。`Caption` 属性的 `get` 访问器返回存储在私有 `caption` 字段中的字符串。`set` 访问器检查新值是否与当前值不同，如果新值与当前值不同，它将存储新值并重新绘制控件。属性常常跟在上面显示的模式后面：`get` 访问器只返回一个存储在私有字段中的值，而 `set` 访问器则用于修改该私有字段，然后执行一些必要的操作，以完全更新所涉及的对象的状态。

下面的示例根据上述 `Button` 类演示如何使用 `Caption` 属性：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
Button okButton = new Button();
okButton.Caption = "OK";         // Invokes set accessor
string s = okButton.Caption;     // Invokes get accessor
```

此处，通过向属性赋值调用 set 访问器，而 get 访问器则通过在表达式中引用该属性来调用。

属性的 get 和 set 访问器都不是独立的成员，也不能单独地声明一个属性的访问器。因此，读写属性的两个访问器不可能具有不同的可访问性。下面的示例

```
class A
{
   private string name;

   public string Name {          // Error, duplicate member name
      get { return name; }
   }

   public string Name {          // Error, duplicate member name
      set { name = value; }
   }
}
```

不是声明单个读写属性。相反，它声明了两个同名的属性，一个是只读的，一个是只写的。由于在同一个类中声明的两个成员不能同名，此示例将导致发生一个编译时错误。

当在一个派生类中用与某个所继承的属性相同的名称声明一个新属性时，该派生属性将会隐藏所继承的属性（同时在读取和写入方面）。在下面的示例中

```
class A
{
   public int P {
      set {...}
   }
}

class B: A
{
   new public int P {
      get {...}
   }
}
```

B 中的 P 属性同时在读取和写入方面隐藏 A 中的 P 属性。因此，在下列语句中

```
B b = new B();
b.P = 1;       // Error, B.P is read-only
((A)b).P = 1;  // Ok, reference to A.P
```

向 b.P 赋值会导致编译时错误，原因是 B 中的只读属性 P 隐藏了 A 中的只写属性 P。但是，仍可以使用强制转换来访问那个被隐藏了的 P 属性。

与公共字段不同，属性在对象的内部状态和它的公共接口之间提供了一种隔离手段。请看此示例：

```
class Label
{
   private int x, y;
   private string caption;

   public Label(int x, int y, string caption) {
      this.x = x;
      this.y = y;
      this.caption = caption;
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public int X {
            get { return x; }
        }

        public int Y {
            get { return y; }
        }

        public Point Location {
            get { return new Point(x, y); }
        }

        public string Caption {
            get { return caption; }
        }
    }
```

此处，Label 类将使用两个 int 字段（x 和 y）来存储它的位置。该位置同时采用两种方式公共地公开：X 和 Y 属性，以及类型 Point 的 Location 属性。如果在 Label 的未来版本中采用 Point 结构在内部存储此位置更为方便，则可以不影响类的公共接口就完成更改：

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}
```

相反，如果 x 和 y 是 public readonly 字段，则无法对 Label 类进行上述更改。

通过属性公开状态并不一定比直接公开字段效率低。具体而言，当属性是非虚的且只包含少量代码时，执行环境可能会用访问器的实际代码替换对访问器进行的调用。此过程称为内联 (inlining)，它使属性访问与字段访问一样高效，而且仍保留了属性的更高灵活性。

由于调用 get 访问器在概念上等效于读取字段的值，因此使 get 访问器具有可见的副作用被认为是不好的编程风格。在下面的示例中

```
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

Next 属性的值取决于该属性以前被访问的次数。因此，访问此属性会产生可见的副作用，而此属性应当作为一个方法实现。

get 访问器的“无副作用”约定并不意味着 get 访问器应当始终被编写为只返回存储在字段中的值。事实上，get 访问器经常通过访问多个字段或调用方法以计算属性的值。但是，正确设计的 get 访问器不会执行任何导致对象的状态发生可见变化的操作。

属性还可用于将某个资源的初始化延迟到第一次引用该资源时执行。例如：

```
using System.IO;
public class Console
{
   private static TextReader reader;
   private static TextWriter writer;
   private static TextWriter error;
   public static TextReader In {
      get {
         if (reader == null) {
            reader = new StreamReader(Console.OpenStandardInput());
         }
         return reader;
      }
   }
   public static TextWriter Out {
      get {
         if (writer == null) {
            writer = new StreamWriter(Console.OpenStandardOutput());
         }
         return writer;
      }
   }
   public static TextWriter Error {
      get {
         if (error == null) {
            error = new StreamWriter(Console.OpenStandardError());
         }
         return error;
      }
   }
}
```

Console 类包含三个属性：In、Out 和 Error，它们分别表示三种标准设备：输入、输出和错误设备。通过将这些成员作为属性公开，Console 类可以将它们的初始化延迟到它们被实际使用时。例如，在下列示例中，仅在第一次引用 Out 属性时

```
Console.Out.WriteLine("hello, world");
```

才创建输出设备的基础 TextWriter。但是，如果应用程序不引用 In 和 Error 属性，则不会创建这些设备的任何对象。

10.7.3 自动实现的属性

将属性指定为自动实现的属性时，隐藏的后备字段将自动可用于该属性，并实现访问器以执行对该后备字段的读写的操作。

以下示例

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
 public class Point {
   public int X { get; set; } // automatically implemented
   public int Y { get; set; } // automatically implemented
}
```

等效于下面的声明：

```
public class Point {
   private int x;
   private int y;
   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

由于支持字段不可访问，因此只能通过属性访问器对其进行读写，即使在包含类型中也是如此。这意味着自动实现的只读或只写属性没有意义，且不允许使用这两种属性。但是，可通过不同的方式为每个访问器设置访问级别。因而，私有后备字段的只读属性的效果可能类似于：

```
public class ReadOnlyPoint {
   public int X { get; private set; }
   public int Y { get; private set; }
   public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

此限制还意味着只能使用结构的标准构造函数来实现具有自动实现的属性的结构类型的明确赋值，原因是为该属性本身赋值需要为该结构明确赋值。这意味着用户定义的构造函数必须调用默认构造函数。

10.7.4 可访问性

如果一个访问器带有 *accessor-modifier*，则该访问器的可访问域（第3.5.2 节）通过使用 *accessor-modifier* 所声明的可访问性来确定。如果访问器没有 *accessor-modifier*，则该访问器的可访问域根据属性或索引器所声明的可访问性来确定。

accessor-modifier 存在与否对于成员查找（第7.3 节）或重载决策（第 7.5.3 节）毫无影响。属性或索引器的修饰符始终直接确定所绑定到的属性或索引器，并不考虑访问的上下文。

一旦选择了某个特定的属性或索引器，则所涉及的特定访问器的可访问域将用来确定该访问器的使用是否有效：

- 如果作为值（第 7.1.1 节）使用，则 `get` 访问器必须存在并且可访问。
- 如果作为简单赋值的目标使用（第 7.17.1 节），则 `set` 访问器必须存在并且可访问。
- 如果作为复合赋值的目标（第 7.17.2 节）或作为 `++` 或 `--` 运算符的目标（第 7.5.9 节、第 7.6.5 节）使用，则 `get` 访问器和 `set` 访问器都必须存在并且可访问。

在下面的示例中，属性 `A.Text` 被属性 `B.Text` 隐藏，甚至是在只调用 `set` 访问器的上下文中。相比之下，属性 `B.Count` 对于类 `M` 不可访问，因此改用可访问的属性 `A.Count`。

```
class A
{
   public string Text {
      get { return "hello"; }
      set { }
   }

   public int Count {
      get { return 5; }
      set { }
   }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class B: A
{
   private string text = "goodbye";
   private int count = 0;

   new public string Text {
      get { return text; }
      protected set { text = value; }
   }

   new protected int Count {
      get { return count; }
      set { count = value; }
   }
}

class M
{
   static void Main() {
      B b = new B();
      b.Count = 12;           // Calls A.Count set accessor
      int i = b.Count;        // Calls A.Count get accessor
      b.Text = "howdy";       // Error, B.Text set accessor not accessible
      string s = b.Text;      // Calls B.Text get accessor
   }
}
```

用来实现接口的访问器不能含有*accessor-modifier*。如果仅使用一个访问器实现接口，则另一个访问器可以用 *accessor-modifier* 声明：

```
public interface I
{
   string Prop { get; }
}

public class C: I
{
   public Prop {
      get { return "April"; }    // Must not have a modifier here
      internal set {...}         // Ok, because I.Prop has no set accessor
   }
}
```

10.7.5 虚、密封、重写和抽象访问器

virtual 属性声明指定属性的访问器是虚的。virtual 修饰符适用于读写属性的两个访问器（读写属性的访问器不可能只有一个是虚的）。但是，如果其中一个访问器是private，则将忽略该访问器的 virtual 修饰符，因为它对 virtual 和 private 无意义。

abstract 属性声明指定属性的访问器是虚的，但不提供访问器的实际实现。另外，非抽象派生类还要求通过重写属性以提供它们自己的访问器实现。由于抽象属性声明的访问器不提供实际实现，因此它的 *accessor-body* 只由一个分号组成。

同时包含 abstract 和 override 修饰符的属性声明表示属性是抽象的并且重写一个基属性。此类属性的访问器也是抽象的。

只能在抽象类（第 10.1.1.1 节）中使用抽象属性声明。通过用一个指定 override 指令的属性声明，可以在派生类中来重写被继承的虚属性的访问器。这称为重写属性声明(overriding property declaration)。重写属性声明并不声明新属性。相反，它只是对现有虚属性的访问器的实现进行专用化。

重写属性声明必须指定与所继承的属性完全相同的可访问性修饰符、类型和名称。如果被继承的属性只有单个访问器（即该属性是只读或只写的），则重写属性必须只包含该访问器。如果被继承的属性同时

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

包含两个访问器（即该属性是读写的），则重写属性既可以仅包含其中任一个访问器，也可同时包含两个访问器。

重写属性声明可以包含 sealed 修饰符。此修饰符的使用可以防止派生类进一步重写该属性。密封属性的访问器也是密封的。

除了在声明和调用语法中的差异，虚的、密封、重写和抽象访问器与虚的、密封、重写和抽象方法具有完全相同的行为。具体而言，第 10.6.3、10.6.4、10.6.5 和 10.6.6 节中描述的规则都适用，就好像访问器是相应形式的方法一样：

- get 访问器相当于一个无形参方法，该方法具有属性类型的返回值以及与包含属性相同的修饰符。
- set 访问器相当于一个方法，该方法具有单个属性类型的值形参、void 返回类型以及与包含属性相同的修饰符。

在下面的示例中

```
abstract class A
{
   int y;

   public virtual int X {
      get { return 0; }
   }

   public virtual int Y {
      get { return y; }
      set { y = value; }
   }

   public abstract int Z { get; set; }
}
```

X 是虚只读属性，Y 是虚读写属性，而 Z 是抽象读写属性。由于 Z 是抽象的，所以包含类 A 也必须声明为抽象的。

下面演示了一个从 A 派生的类：

```
class B: A
{
   int z;

   public override int X {
      get { return base.X + 1; }
   }

   public override int Y {
      set { base.Y = value < 0? 0: value; }
   }

   public override int Z {
      get { return z; }
      set { z = value; }
   }
}
```

此处，X、Y 和 Z 的声明是重写属性声明。每个属性声明都与它们所继承的属性的可访问性修饰符、类型和名称完全匹配。X 的 get 访问器和 Y 的 set 访问器使用 base 关键字来访问所继承的访问器。Z 的声明重写了两个抽象访问器，因此在 B 中不再有抽象的函数成员，B 也可以是非抽象类。

如果属性声明为 override，则进行重写的代码必须能够访问被重写的访问器。此外，属性或索引器本身以及访问器所声明的可访问性都必须与被重写的成员和访问器所声明的可访问性相匹配。例如：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
public class B
{
   public virtual int P {
      protected set {...}
      get {...}
   }
}
public class D: B
{
   public override int P {
      protected set {...}        // Must specify protected here
      get {...}                  // Must not have a modifier here
   }
}
```

10.8 事件

事件 (event) 是一种使对象或类能够提供通知的成员。客户端可以通过提供事件处理程序(event handler)为相应的事件添加可执行代码。

事件是使用 *event-declaration* 来声明的：

event-declaration:
attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type variable-declarators* `;`
attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *type member-name* `{` *event-accessor-declarations* `}`

event-modifiers:
event-modifier
event-modifiers event-modifier

event-modifier:
`new`
`public`
`protected`
`internal`
`private`
`static`
`virtual`
`sealed`
`override`
`abstract`
`extern`

event-accessor-declarations:
add-accessor-declaration remove-accessor-declaration
remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:
attributes$_{opt}$ `add` *block*

remove-accessor-declaration:
attributes$_{opt}$ `remove` *block*

event-declaration 可包含一组 *attributes*（第 17 节）和四个访问修饰符（第 10.3.5 节）的有效组合 `new`（第 10.3.4 节）、以及 `static`（第 10.6.2 节）、`virtual`（第 10.6.3 节）、`override`（第 10.6.4 节）、`sealed`（第 10.6.5 节）、`abstract`（第 10.6.6 节）和 `extern`（第 10.6.7 节）修饰符。

在有效的修饰符组合方面，事件声明与方法声明（第 10.6 节）遵循相同的规则。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

事件声明的 *type* 必须是 *delegate-type*（第 4.2 节），而该 *delegate-type* 必须至少具有与事件本身一样的可访问性（第 3.5.4 节）。

事件声明可能包含 *event-accessor-declaration*，但如果不包含，则对于非 extern、非 abstract 事件，编译器将自动提供（第 10.8.1 节）；对于 extern 事件，访问器由外部提供。

省略了 *event-accessor-declaration* 的事件声明用于定义一个或多个事件（每个 *variable-declarator* 各表示一个事件）。特性和修饰符适用于由此类 *event-declaration* 声明的所有成员。

若 *event-declaration* 既包含 `abstract` 修饰符又包含以大括号分隔的 *event-accessor-declaration*，则会导致编译时错误。

当事件声明包含 `extern` 修饰符时，称该事件为外部事件 (external event)。因为外部事件声明不提供任何实际的实现，所以在一个外部事件声明中既包含 `extern` 修饰符又包含 *event-accessor-declaration* 是错误的。

带 `abstract` 或 `external` 修饰符的事件声明的 *variable-declarator* 包括 *variable-initializer* 将产生编译时错误。

事件可用作 `+=` 和 `-=` 运算符（第 7.17.3 节）左边的操作数。这些运算符分别用于将事件处理程序添加到所涉及的事件或从该事件中移除事件处理程序，而该事件的访问修饰符用于控制允许这类运算的上下文。

由于 `+=` 和 `-=` 是仅有的能够在声明了某个事件的类型的外部对该事件进行的操作，因此，外部代码可以为一个事件添加和移除处理程序，但是不能以其他任何方式来获取或修改基础的事件处理程序列表。

在 `x += y` 或 `x -= y` 形式的运算中，如果 `x` 是一个事件，而且该引用发生在声明了 `x` 事件的类型之外，则这种运算结果的类型为 `void`（这正好与该运算的实际效果相反，它用于给 `xx` 赋值，应该具有 `x` 所属的类型）。此规则能够禁止外部代码以间接方式来检查一个事件的基础委托。

下面的示例演示如何将事件处理程序添加到 `Button` 类的实例：

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
   public event EventHandler Click;
}

public class LoginDialog: Form
{
   Button OkButton;
   Button CancelButton;

   public LoginDialog() {
      OkButton = new Button(...);
      OkButton.Click += new EventHandler(OkButtonClick);
      CancelButton = new Button(...);
      CancelButton.Click += new EventHandler(CancelButtonClick);
   }

   void OkButtonClick(object sender, EventArgs e) {
      // Handle OkButton.Click event
   }

   void CancelButtonClick(object sender, EventArgs e) {
      // Handle CancelButton.Click event
   }
}
```

此处，`LoginDialog` 实例构造函数将创建两个 `Button` 实例并将事件处理程序附加到 `Click` 事件。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.8.1 类似字段的事件

在包含事件声明的类或结构的程序文本内，某些事件可以像字段一样使用。若要以这种方式使用，事件不能是 abstract 或 extern，而且不能显式包含 *event-accessor-declaration*。此类事件可以用在任何允许使用字段的上下文中。该字段含有一个委托（第 15 章），它引用已添加到相应事件的事件处理程序列表。如果尚未添加任何事件处理程序，则该字段包含 null。

在下面的示例中

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
   public event EventHandler Click;

   protected void OnClick(EventArgs e) {
      if (Click != null) Click(this, e);
   }

   public void Reset() {
      Click = null;
   }
}
```

Click 在 Button 类中用作一个字段。如上例所示，可以在委托调用表达式中检查、修改和使用字段。Button 类中的 OnClick 方法将“引发”Click 事件。“引发一个事件”与“调用一个由该事件表示的委托”这两个概念完全等效，因此没有用于引发事件的特殊语言构造。请注意，在委托调用之前有一个检查，以确保该委托不是 null。

在 Button 类的声明外，Click 成员只能用在 += 和 -= 运算符的左边，如

```
b.Click += new EventHandler(…);
```

将一个委托追加到 Click 事件的调用列表，而

```
b.Click -= new EventHandler(…);
```

则从 Click 事件的调用列表中移除一个委托。

当编译一个类似字段的事件时，编译器会自动创建一个存储区来存放相关的委托，并为事件创建相应的访问器以向委托字段中添加或移除事件处理程序。添加或移除操作是线程安全的，并且可能会（但不要求）在为实例事件的包含对象加锁（第 8.12 节）的情况下进行，或者在为静态事件的类型对象（第 7.6.10.6 节）加锁的情况下进行。

因此，下列形式的实例事件声明：

```
class X
{
   public event D Ev;
}
```

将编译为如下语句：

```
class X
{
   private D __Ev;  // field to hold the delegate

   public event D Ev {
      add {
         /* add the delegate in a thread safe way */
      }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}
```

在类 X, 中，对 += 和 -= 运算符左侧的 Ev on 的引用将导致调用添加和移除访问器的操作。所有其他对 Ev 的引用将在编译时改为引用隐藏字段 __Ev（第 7.6.4 节）。名称“__Ev”是任意的；隐藏字段可以具有任何名称或根本没有名称。

10.8.2 事件访问器

事件声明通常省略 *event-accessor-declaration*，如上面的 Button 示例中所示。但会有一些特殊情况，例如，为每个事件设置一个字段所造成的内存开销，有时会变得不可接受。在这种情况下，可以在类中包含 *event-accessor-declaration*，并采用专用机制来存储事件处理程序列表。

事件的 *event-accessor-declarations* 指定与添加和移除事件处理程序相关联的可执行语句。

访问器声明由一个 *add-accessor-declaration* 和一个 *remove-accessor-declaration* 组成。每个访问器声明都包含标记 add 或 remove，后接一个 *block*。与 *add-accessor-declaration* 相关联的 *block* 指定添加事件处理程序时要执行的语句，而与 *remove-accessor-declaration* 相关联的 *block* 指定移除事件处理程序时要执行的语句。

每个 *add-accessor-declaration* 和 *remove-accessor-declaration* 相当于一个方法，它具有一个属于事件类型的值形参并且其返回类型为 void。事件访问器的隐式形参名为 value。当事件用在事件赋值中时，就会调用适当的事件访问器。具体而言，如果赋值运算符为 +=，则使用添加访问器，而如果赋值运算符为 -=，则使用移除访问器。在两种情况下，赋值运算符的右操作数都用作事件访问器的实参。*add-accessor-declaration* 或 *remove-accessor-declaration* 的块必须符合第 10.6.10 节所描述的用于 void 方法的规则。具体而言，不允许此类块中的 return 语句指定表达式。

由于事件访问器隐式具有一个名为 value 的形参，因此在事件访问器中声明的局部变量或常量若使用该名称，就会导致编译时错误。

在下面的示例中

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        // Invoke the MouseUp event
        protected void OnMouseUp(MouseEventArgs args) {
           MouseEventHandler handler;
           handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
           if (handler != null)
              handler(this, args);
        }
     }
```

Control 类为事件实现了一个内部存储机制。AddEventHandler 方法将委托值与键关联，GetEventHandler 方法返回当前与键关联的委托，而 RemoveEventHandler 方法将移除一个委托使它不再成为指定事件的一个事件处理程序。可以推断：在这样设计的基础存储机制下，当一个键所关联的委托值为 null 时，不会有存储开销，从而使未处理的事件不占任何存储空间。

10.8.3 静态事件和实例事件

当事件声明包含 static 修饰符时，称该事件为静态事件 (static event)。当不存在 static 修饰符时，称该事件为实例事件 (instance event)。

静态事件不和特定实例关联，因此在静态事件的访问器中引用 this 会导致编译时错误。

实例事件与类的给定实例关联，此实例在该事件的访问器中可以用 this（第 7.6.7 节）来访问。

在 E.M 形式的 *member-access*（第 7.6.4 节）中引用事件时，如果 M 为静态事件，则 E 必须表示包含 M 的类型，如果 M 为实例事件，则 E 必须表示包含 M 的类型的一个实例。

第 10.3.7 节对静态成员和实例成员之间的差异进行了进一步讨论。

10.8.4 虚、密封、重写和抽象访问器

virtual 事件声明指定事件的访问器是虚的。virtual 修饰符适用于事件的两个访问器。

abstract 事件声明指定事件的访问器是虚的，但是不提供这些访问器的实际实现。而且，非抽象派生类需要通过重写事件来提供它们自己的访问器实现。因为抽象事件声明不提供任何实际的实现，所以它无法提供以大括号界定的 *event-accessor-declaration*。

同时包含 abstract 和 override 修饰符的事件声明指定该事件是抽象的并重写一个基事件。此类事件的访问器也是抽象的。

只允许在抽象类（第 10.1.1.1 节）中使用抽象事件声明。

继承的虚事件的访问器可以在相关的派生类中用一个指定 override 修饰符的事件声明来进行重写。这称为重写事件声明 (overriding event declaration)。重写事件声明不声明新事件。实际上，它只是专用化了现有虚事件的访问器的实现。

重写事件声明必须采用与被重写事件完全相同的可访问性修饰符、类型和名称。

重写事件声明可以包含 sealed 修饰符。使用此修饰符可以防止相关的派生类进一步重写该事件。密封事件的访问器也是密封的。

重写事件声明包含 new 修饰符会导致编译时错误。

除了在声明和调用语法中的差异，虚的、密封、重写和抽象访问器与虚的、密封、重写和抽象方法具有完全相同的行为。具体而言，第 10.6.3、10.6.4、10.6.5 和 10.6.6 节中描述的规则都适用，就好像访问器是相应形式的方法一样。每个访问器都对应于一个方法，它只有一个属于所涉及的事件类型的值形参、返回类型为 void，且具有与包含事件相同的修饰符。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.9 索引器

索引器 (indexer) 是这样一个成员：它使对象能够用与数组相同的方式进行索引。索引器是使用 *indexer-declaration* 来声明的：

indexer-declaration:
    attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* `{` *accessor-declarations* `}`

indexer-modifiers:
    indexer-modifier
    indexer-modifiers *indexer-modifier*

indexer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `virtual`
    `sealed`
    `override`
    `abstract`
    `extern`

indexer-declarator:
    type `this` `[` *formal-parameter-list* `]`
    type *interface-type* `.` `this` `[` *formal-parameter-list* `]`

indexer-declaration 可包含一组 *attributes*（第 17 节）和一个由四个访问修饰符构成的有效组合（第 10.3.5 节），`new`（第 10.3.4 节）、`virtual`（第 10.6.3 节）、`override`（第 10.6.4 节）、`sealed`（第 10.6.5 节）、`abstract`（第 10.6.6 节）和 `extern`（第 10.6.7 节）。

关于有效的修饰符组合，索引器声明与方法声明（第 10.6 节）遵循相同的规则（唯一的例外是：在索引器声明中不允许使用静态修饰符）。

修饰符 `virtual`、`override` 和 `abstract` 相互排斥，但有一种情况除外。`abstract` 和 `override` 修饰符可以一起使用以便抽象索引器可以重写虚索引器。

索引器声明的 *type* 用于指定由该声明引入的索引器的元素类型。除非索引器是一个显式接口成员的实现，否则该 *type* 后要跟一个关键字 `this`。而对于显式接口成员的实现，该 *type* 后要先跟一个 *interface-type*、一个“`.`”，再跟一个关键字 `this`。与其他成员不同，索引器不具有用户定义的名称。

formal-parameter-list 用于指定索引器的形参。索引器的形参表对应于方法的形参表（第 10.6.1 节），不同之处仅在于索引器的形参表中必须至少含有一个形参，并且不允许使用 `ref` 和 `out` 形参修饰符。

索引器的 *type* 和在 *formal-parameter-list* 中引用的每个类型都必须至少具有与索引器本身相同的可访问性（第 3.5.4 节）。

accessor-declarations（第 10.7.2 节）（它必须被括在“`{`”和“`}`”标记内）用于声明该索引器的访问器。这些访问器用来指定与读取和写入索引器元素相关联的可执行语句。

虽然访问索引器元素的语法与访问数组元素的语法相同，但是索引器元素并不属于变量。因此，不可能将索引器元素作为 `ref` 或 `out` 实参传递。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

索引器的形参表定义索引器的签名（第 3.6 节）。具体而言，索引器的签名由其形参的数量和类型组成。但索引器元素的类型和形参的名称都不是索引器签名的组成部分。

索引器的签名必须不同于在同一个类中声明的所有其他索引器的签名。

索引器和属性在概念上非常类似，但在下列方面有所区别：

- 属性由它的名称标识，而索引器由它的签名标识。
- 属性是通过 *simple-name*（第 7.6.2 节）或是 *member-access*（第 7.6.4 节）来访问的，而索引器元素则是通过 *element-access*（第 7.6.6.2 节）来访问的。
- 属性可以是 `static` 成员，而索引器始终是实例成员。
- 属性的 `get` 访问器对应于不带形参的方法，而索引器的 `get` 访问器对应于与索引器具有相同的形参表的方法。
- 属性的 `set` 访问器对应于具有名为 `value` 的单个形参的方法，而索引器的 `set` 访问器对应于与索引器具有相同的形参表加上一个名为 `value` 的附加形参的方法。
- 若在索引器访问器内使用与该索引器的形参相同的名称来声明局部变量，就会导致一个编译时错误。
- 在重写属性声明中，被继承的属性是使用语法 `base.P` 访问的，其中 `P` 为属性名称。在重写索引器声明中，被继承的索引器是使用语法 `base[E]` 访问的，其中 `E` 是一个用逗号分隔的表达式列表。

除上述差异以外，所有在第 10.7.2 节和第 10.7.3 节中定义的规则都适用于索引器访问器以及属性访问器。

当索引器声明包含 `extern` 修饰符时，称该索引器为外部索引器 (external indexer)。因为外部索引器声明不提供任何实际的实现，所以它的每个 *accessor-declarations* 都由一个分号组成。

下面的示例声明了一个 `BitArray` 类，该类实现了一个索引器，用于访问位数组中的单个位。

```
using System;

class BitArray
{
   int[] bits;
   int length;

   public BitArray(int length) {
      if (length < 0) throw new ArgumentException();
      bits = new int[((length - 1) >> 5) + 1];
      this.length = length;
   }

   public int Length {
      get { return length; }
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public bool this[int index] {
            get {
                if (index < 0 || index >= length) {
                    throw new IndexOutOfRangeException();
                }
                return (bits[index >> 5] & 1 << index) != 0;
            }
            set {
                if (index < 0 || index >= length) {
                    throw new IndexOutOfRangeException();
                }
                if (value) {
                    bits[index >> 5] |= 1 << index;
                }
                else {
                    bits[index >> 5] &= ~(1 << index);
                }
            }
        }
    }
```

BitArray 类的实例所占的内存远少于相应的 bool[]（这是由于前者的每个值只占一位，而后者的每个值要占一个字节），而且，它可以执行与 bool[] 相同的操作。

下面的 CountPrimes 类使用 BitArray 和经典的“筛选”算法计算 1 和给定的最大数之间质数的数目：

```
class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}
```

请注意，访问 BitArray 的元素的语法与用于 bool[] 的语法完全相同。

下面的示例演示一个具有两个形参的索引器的 26 × 10 网格类。第一个形参必须是 A – Z 范围内的大写或小写字母，而第二个形参必须是 0 – 9 范围内的整数。

```
using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public int this[char c, int col] {
            get {
                c = Char.ToUpper(c);
                if (c < 'A' || c > 'Z') {
                    throw new ArgumentException();
                }
                if (col < 0 || col >= NumCols) {
                    throw new IndexOutOfRangeException();
                }
                return cells[c - 'A', col];
            }

            set {
                c = Char.ToUpper(c);
                if (c < 'A' || c > 'Z') {
                    throw new ArgumentException();
                }
                if (col < 0 || col >= NumCols) {
                    throw new IndexOutOfRangeException();
                }
                cells[c - 'A', col] = value;
            }
        }
    }
```

10.9.1 索引器重载

第 7.5.2 节中描述了索引器重载决策规则。

10.10 运算符

运算符 (operator) 是一种用来定义可应用于类实例的表达式运算符的含义的成员。运算符是使用 *operator-declaration* 来声明的：

operator-declaration:
    attributes$_{opt}$ *operator-modifiers* *operator-declarator* *operator-body*

operator-modifiers:
    operator-modifier
    operator-modifiers *operator-modifier*

operator-modifier:
    `public`
    `static`
    `extern`

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* `)`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

overloadable-binary-operator:
    +
    -
    *
    /
    %
    &
    |
    ^
    <<
    right-shift
    ==
    !=
    >
    <
    >=
    <=

conversion-operator-declarator:
    `implicit operator` *type* `(` *type identifier* `)`
    `explicit operator` *type* `(` *type identifier* `)`

operator-body:
    block
    ;

有三类可重载运算符：一元运算符（第 10.10.1 节）、二元运算符（第 10.10.2 节）和转换运算符（第 10.10.3 节）。

当运算符声明包含 `extern` 修饰符时，称该运算符为外部运算符 (external operator)。因为外部运算符不提供任何实际的实现，所以它的 *operator-body* 由一个分号组成。对于所有其他运算符，*operator-body* 由一个 *block* 组成，它指定在调用该运算符时需要执行的语句。运算符的 *block* 必须遵循第 10.6.10 节中所描述的适用于值返回方法的规则。

下列规则适用于所有的运算符声明：

- 运算符声明必须同时包含一个 `public` 和一个 `static` 修饰符。
- 运算符的参数必须是值参数（第 5.1.4 节）。在运算符声明中指定 `ref` 或 `out` 形参会导致编译时错误。
- 运算符的签名（第 10.10.1、10.10.2、10.10.3 节）必须不同于在同一个类中声明的所有其他运算符的签名。
- 运算符声明中引用的所有类型都必须具有与运算符本身相同的可访问性（第 3.5.4 节）。
- 同一修饰符在一个运算符声明中多次出现是错误的。

每个运算符类别都有附加的限制，将在下列几节中说明。

与其他成员一样，在基类中声明的运算符由派生类继承。由于运算符声明始终要求声明运算符的类或结构参与运算符的签名，因此在派生类中声明的运算符不可能隐藏在基类中声明的运算符。因此，运算符声明中永远不会要求也不允许使用 `new` 修饰符。

关于一元和二元运算符的其他信息可以在第 7.3 节中找到。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

关于转换运算符的其他信息可以在第 6.4 节中找到。

10.10.1 一元运算符

下列规则适用于一元运算符声明，其中 T 表示包含运算符声明的类或结构的实例类型：

- 一元 +、-、! 或 ~ 运算符必须具有单个 T 或 T? 类型的形参，并且可以返回任何类型。
- 一元 ++ 或 -- 运算符必须带有单个 T 或 T? 类型的形参并且必须返回与它相同或由它派生的类型。
- 一元 true 或 false 运算符必须采用单个 T 或 T? 类型的形参，并且必须返回类型 bool。

一元运算符的签名由运算符标记（+、-、 !、~、++、--、true 或 false）以及单个形参的类型构成。返回类型不是一元运算符的签名的组成部分，形参的名称也不是。

一元运算符 true 和 false 要求成对的声明。如果类只声明了这两个运算符的其中一个而没有声明另一个，将发生编译时错误。第 7.12.2 节和第 7.20 节中对 true 和 false 运算符做了进一步的介绍。

下面的示例演示了对一个整数向量类的 operator ++ 的实现以及随后对它的使用：

```
public class IntVector
{
   public IntVector(int length) {...}
   public int Length {...}             // read-only property
   public int this[int index] {...}    // read-write indexer
   public static IntVector operator ++(IntVector iv) {
      IntVector temp = new IntVector(iv.Length);
      for (int i = 0; i < iv.Length; i++)
         temp[i] = iv[i] + 1;
      return temp;
   }
}
class Test
{
   static void Main() {
      IntVector iv1 = new IntVector(4);   // vector of 4 x 0
      IntVector iv2;
      iv2 = iv1++;   // iv2 contains 4 x 0, iv1 contains 4 x 1
      iv2 = ++iv1;   // iv2 contains 4 x 2, iv1 contains 4 x 2
   }
}
```

请注意此运算符方法如何返回通过向操作数添加 1 而产生的值，就像后缀增量和减量运算符（第 7.6.9 节）以及前缀增量和减量运算符（第 7.7.5 节）一样。与在 C++ 中不同，此方法并不需要直接修改其操作数的值。实际上，修改操作数的值会违反后缀递增运算符的标准语义。

10.10.2 二元运算符

下列规则适用于二元运算符声明，其中 T 表示包含运算符声明的类或结构的实例类型：

- 二元非移位运算符必须带有两个形参，其中至少有一个必须为类型 T 或 T?，并且可返回其中的任一类型。
- 二元 << 或 >> 运算符必须带有两个形参，其中第一个必须具有类型 T 或 T?，第二个必须具有类型 int 或 int?，并且可返回其中的任一类型。

二元运算符的签名由运算符标记（+、-、*、/、%、&、|、^、<<、>>、==、!=、>、<、>= 或 <=）以及两个形参的类型构成。它本身的返回类型及形参的名称不是二元运算符签名的组成部分。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

某些二元运算符要求成对地声明。对于要求成对地声明的运算符，若声明了其中一个，就必须对另一个作出相匹配的声明。如果返回类型 R_1 和 R_2 之间、操作数类型 P_1 和 P_2 之间、操作数类型 Q_1 和 Q_2 之间存在标识转换，则两个运算符声明 $R_1\ op_1(P_1, Q_1)$ 和 $R_2\ op_2(P_2, Q_2)$ 匹配。下列运算符要求成对的声明：

- `operator ==` 和 `operator !=`
- `operator >` 和 `operator <`
- `operator >=` 和 `operator <=`

10.10.3 转换运算符

转换运算符声明引入用户定义的转换 (user-defined conversion)（第 6.4 节），此转换可以扩充预定义的隐式和显式转换。

包含 `implicit` 关键字的转换运算符声明引入用户定义的隐式转换。隐式转换可以在多种情况下发生，包括函数成员调用、强制转换表达式和赋值。第 0 节对此有进一步描述。

包含 `explicit` 关键字的转换运算符声明引入用户定义的显式转换。显式转换可以发生在强制转换表达式中，第 6.2 节中对此进行了进一步描述。

转换运算符将某个源类型（由该转换运算符的形参类型指定）转换为目标类型（由该转换运算符的返回类型指定）。

对于给定的源类型 `S` 和目标类型 `T`，如果 `S` 或 `T` 是可以为 null 的类型，则让 S_0 和 T_0 引用它们的基础类型，否则 S_0 和 T_0 分别等于 `S` 和 `T`。仅当以下条件皆为真时，才允许类或结构声明从源类型 `S` 到目标类型 `T` 的转换：

- S_0 和 T_0 是不同的类型。
- S_0 和 T_0 中有一个是声明该运算符的类类型或结构类型。
- S_0 和 T_0 都不是 *interface-type*。
- 除用户定义的转换之外，不存在从 `S` 到 `T` 或从 `T` 到 `S` 的转换。

为了实现这些规则，将任何与 `S` 或 `T` 关联的类型形参都视为与其他类型没有继承关系的特有类型，而且忽略对这些类型形参的任何约束。

在下面的示例中

```
class C<T> {...}
class D<T>: C<T>
{
   public static implicit operator C<int>(D<T> value) {...}    // Ok
   public static implicit operator C<string>(D<T> value) {...} // Ok
   public static implicit operator C<T>(D<T> value) {...}      // Error
}
```

前两个运算符声明是允许的，因为根据第 10.9.3 节，`T` 和 `int` 以及 `string` 分别被视为没有关系的唯一类型。但是，第三个运算符是错误的，因为 `C<T>` 是 `D<T>` 的基类。

从第二条规则可以推知，转换运算符必须将声明了该运算符的类或结构类型或者作为目标类型，或者作为源类型。例如，一个类或结构类型 `C` 可以定义从 `C` 到 `int` 和从 `int` 到 `C` 的转换，但不能定义从 `int` 到 `bool` 的转换。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

不能直接重新定义一个已存在的预定义转换。因此，不允许转换运算符将 object 转换为其他类型或将其他类型转换为 object，这是因为已存在隐式和显式转换来执行 object 与所有其他类型之间的转换。同样，转换的源类型和目标类型不能是对方的基类型，这是由于已经存在这样的转换。

但是，对于特定类型实参，可以在泛型类型上声明这样的运算符，即这些运算符指定了已经作为预定义转换而存在的转换。在下面的示例中

```
struct Convertible<T>
{
   public static implicit operator Convertible<T>(T value) {...}
   public static explicit operator T(Convertible<T> value) {...}
}
```

当把类型 object 指定为 T 的类型实参时，第二个运算符将声明一个已经存在的转换（存在从任何类型到类型 object 的隐式转换，因此也存在显式转换）。

在两个类型之间存在预定义转换的情况下，这些类型之间的任何用户定义的转换将被忽略。具体包括：

- 如果存在从类型 S 到类型 T 的预定义隐式转换（第 6.1 节），则从 S 到 T 的所有用户定义的转换（隐式或显式）将被忽略。
- 如果存在从类型 S 到类型 T 的预定义显式转换（第 6.2 节），则从 S 到 T 的所有用户定义的显式转换将被忽略。此外：
  - 如果 T 是接口类型，则会忽略从 S 到 T 的用户定义的隐式转换。
  - 否则，仍会考虑从 S 到 T 的用户定义的隐式转换。

对于除 object 以外的所有类型，上面的 Convertible<T> 类型声明的运算符都不会与预定义的转换发生冲突。例如：

```
void F(int i, Convertible<int> n) {
   i = n;                        // Error
   i = (int)n;                   // User-defined explicit conversion
   n = i;                        // User-defined implicit conversion
   n = (Convertible<int>)i;      // User-defined implicit conversion
}
```

但是对于类型 object，除了下面这个特例之外，预定义的转换将在其他所有情况下隐藏用户定义的转换：

```
void F(object o, Convertible<object> n) {
   o = n;                        // Pre-defined boxing conversion
   o = (object)n;                // Pre-defined boxing conversion
   n = o;                        // User-defined implicit conversion
   n = (Convertible<object>)o;   // Pre-defined unboxing conversion
}
```

不允许使用用户定义的转换从 *interface-type* 进行转换或转换为 interface-type。具体而言，此限制确保了在转换为 *interface-type* 时不会发生任何用户定义的转换，以及只有在要转换的对象实际上实现了指定的 *interface-type* 时，到该 *interface-type* 的转换才会成功。

转换运算符的签名由源类型和目标类型组成。（请注意，这是唯一一种其返回类型参与签名的成员形式。）转换运算符的 implicit 或 explicit 类别不是运算符签名的组成部分。因此，类或结构不能同时声明具有相同源类型和目标类型的 implicit 和 explicit 转换运算符。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

一般来说，如果设计一个用户定义的隐式转换，就应当确保执行该转换时决不会引发异常，并且也决不会丢失信息。如果用户定义的转换可能导致引发异常（例如，由于源实参超出范围）或丢失信息（如放弃高序位），则该转换应该定义为显式转换。

在下面的示例中

```
using System;
public struct Digit
{
   byte value;
   public Digit(byte value) {
      if (value < 0 || value > 9) throw new ArgumentException();
      this.value = value;
   }
   public static implicit operator byte(Digit d) {
      return d.value;
   }
   public static explicit operator Digit(byte b) {
      return new Digit(b);
   }
}
```

从 `Digit` 到 `byte` 的转换是隐式的，这是因为它永远不会引发异常或丢失信息，但是从 `byte` 到 `Digit` 的转换是显式的，这是因为 `Digit` 只能表示 `byte` 的可能值的一个子集。

10.11 实例构造函数

实例构造函数 (instance constructor) 是实现初始化类实例所需操作的成员。实例构造函数是使用 *constructor-declaration* 来声明的：

constructor-declaration:
attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator* *constructor-body*

constructor-modifiers:
constructor-modifier
constructor-modifiers *constructor-modifier*

constructor-modifier:
`public`
`protected`
`internal`
`private`
`extern`

constructor-declarator:
identifier `(` *formal-parameter-list*$_{opt}$ `)` *constructor-initializer*$_{opt}$

constructor-initializer:
`:` `base` `(` *argument-list*$_{opt}$ `)`
`:` `this` `(` *argument-list*$_{opt}$ `)`

constructor-body:
block
`;`

constructor-declaration 可以包含一组 *attributes*（第 17 章）、四个访问修饰符（第 10.3.5 节）的有效组合和一个 `extern`（第 10.6.7 节）修饰符。一个构造函数声明中同一修饰符不能多次出现。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

constructor-declarator 中的 *identifier* 必须是声明了该实例构造函数的那个类的名称。如果指定了任何其他名称，则发生编译时错误。

可选的实例构造函数的 *formal-parameter-list* 必须遵循与方法的 *formal-parameter-list*（第 10.6 节）同样的规则。此形参表定义实例构造函数的签名（第 3.6 节），并且在调用中控制重载决策（第 7.5.2 节）过程以选择某个特定实例的构造函数。

在实例构造函数的 *formal-parameter-list* 中引用的各个类型必须至少具有与构造函数本身相同的可访问性（第 3.5.4 节）。

可选的 *constructor-initializer* 用于指定在执行此实例构造函数的 *constructor-body* 中给出的语句之前需要调用的另一个实例构造函数。第 10.11.1 节对此有进一步描述。

当构造函数声明中包含 `extern` 修饰符时，称该构造函数为外部构造函数 (external constructor)。因为外部构造函数声明不提供任何实际的实现，所以它的 *constructor-body* 仅由一个分号组成。对于所有其他构造函数，*constructor-body* 都由一个 *block* 组成，它用于指定初始化该类的一个新实例时需要执行的语句。这正好相当于一个具有 `void` 返回类型的实例方法的 *block*（第 10.6.10 节）。

实例构造函数是不能继承的。因此，一个类除了自己声明的实例构造函数外，不可能有其他的实例构造函数。如果一个类不包含任何实例构造函数声明，则会自动地为该类提供一个默认实例构造函数（第 10.11.4 节）。

实例构造函数是由 *object-creation-expression*（第 7.6.10.1 节）并通过 *constructor-initializer* 调用的。

10.11.1 构造函数初始值设定项

除了类 `object` 的实例构造函数以外，所有其他的实例构造函数都隐式地包含一个对另一个实例构造函数的调用，该调用紧靠在 *constructor-body* 的前面。要隐式调用的构造函数是由 *constructor-initializer* 确定的：

- `base(`*argument-list*$_{opt}$`)` 形式的实例构造函数初始值设定项导致调用直接基类中的实例构造函数。该构造函数是根据 *argument-list* 和第 7.5.3 节中的重载决策规则选择的。候选实例构造函数集由直接基类中包含的所有可访问的实例构造函数组成，如果直接基类中未声明任何实例构造函数，则候选实例构造函数集由默认构造函数（第 10.11.4 节）组成。如果此集为空，或者无法标识单个最佳实例构造函数，就会发生编译时错误。
- `this(`*argument-list*$_{opt}$`)` 形式的实例构造函数初始值设定项导致调用该类本身所声明的实例构造函数。该构造函数是根据 *argument-list* 和第 7.5.3 节中的重载决策规则选择的。候选实例构造函数集由类本身声明的所有可访问的实例构造函数组成。如果此集为空，或者无法标识单个最佳实例构造函数，就会发生编译时错误。如果实例构造函数声明中包含调用构造函数本身的构造函数初始值设定项，则发生编译时错误。

如果一个实例构造函数中没有构造函数初始值设定项，将会隐式地添加一个 `base()` 形式的构造函数初始值设定项。因此，下列形式的实例构造函数声明

```
C(...) {...}
```

完全等效于

```
C(...): base() {...}
```

实例构造函数声明中的 *formal-parameter-list* 所给出的形参范围包含该声明的实例构造函数初始值设定项。因此，构造函数初始值设定项可以访问该构造函数的形参。例如：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class A
{
   public A(int x, int y) {}
}
class B: A
{
   public B(int x, int y): base(x + y, x - y) {}
}
```

实例构造函数初始值设定项不能访问正在创建的实例。因此在构造函数初始值设定项的实参表达式中引用 this 属于编译时错误，就像实参表达式通过 *simple-name* 引用任何实例成员属于编译时错误一样。

10.11.2 实例变量初始值设定项

当实例构造函数没有构造函数初始值设定项时，或仅具有 base(...) 形式的构造函数初始值设定项时，该构造函数就会隐式地执行在该类中声明的实例字段的初始化操作，这些操作由对应的字段声明中的 *variable-initializer* 指定。这对应于一个赋值序列，它们会在进入构造函数时，在对直接基类的构造函数进行隐式调用之前立即执行。这些变量初始值设定项按它们出现在类声明中的文本顺序执行。

10.11.3 构造函数执行

变量初始值设定项被转换为赋值语句，而这些语句将在对基类实例构造函数进行调用之前执行。这种排序确保了在执行任何访问该实例的语句之前，所有实例字段都已按照它们的变量初始值设定项进行了初始化。

给定示例

```
using System;
class A
{
   public A() {
      PrintFields();
   }
   public virtual void PrintFields() {}
}
class B: A
{
   int x = 1;
   int y;
   public B() {
      y = -1;
   }
   public override void PrintFields() {
      Console.WriteLine("x = {0}, y = {1}", x, y);
   }
}
```

当使用 new B() 创建 B 的实例时，产生如下输出：

```
x = 1, y = 0
```

x 的值为 1，这是由于变量初始值设定项是在调用基类实例构造函数之前执行的。但是，y 的值为 0（int 型变量的默认值），这是因为对 y 的赋值直到基类构造函数返回之后才执行。

可以这样设想来帮助理解：将实例变量初始值设定项和构造函数初始值设定项视为自动插入到 *constructor-body* 之前的语句。下面的示例

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
using System;
using System.Collections;

class A
{
   int x = 1, y = -1, count;

   public A() {
      count = 0;
   }

   public A(int n) {
      count = n;
   }
}

class B: A
{
   double sqrt2 = Math.Sqrt(2.0);
   ArrayList items = new ArrayList(100);
   int max;

   public B(): this(100) {
      items.Add("default");
   }

   public B(int n): base(n - 1) {
      max = n;
   }
}
```

包含若干个变量初始值设定项，还包含两种形式（base 和 this）的构造函数初始值设定项。此示例对应于下面演示的代码，其中每个注释指示一个自动插入的语句（用于自动插入的构造函数调用的语法是无效的，而只是用来阐释此机制）。

```
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");
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        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;
        }
    }
```

10.11.4 默认构造函数

如果一个类不包含任何实例构造函数声明，则会自动地为该类提供一个默认实例构造函数。默认构造函数只是调用直接基类的无形参构造函数。对于一个抽象类，它的默认构造函数的声明可访问性是受保护的。而对于非抽象类，它的默认构造函数的声明可访问性是公共的。因此，默认构造函数始终为下列形式：

```
protected C(): base() {}
```

或

```
public C(): base() {}
```

其中 C 为类的名称。如果重载决策无法确定基类构造函数初始值设定项的唯一最佳候选，则产生编译时错误。

在下面的示例中

```
class Message
{
    object sender;
    string text;
}
```

由于类不包含任何实例构造函数声明，因此就为它提供了一个默认构造函数。因而，此示例完全等效于

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

10.11.5 私有构造函数

当类 T 只声明了私有实例构造函数时，则在 T 的程序文本外部，既不可能从 T 派生出新的类，也不可能直接创建 T 的任何实例。因此，如果欲设计一个类，它只包含静态成员而且有意使它不能被实例化，则只需给它添加一个空的私有实例构造函数，即可达到目的。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 类用于将相关的方法和常量组合在一起，但是它不能被实例化。因此它声明了单个空的私有实例构造函数。若要取消默认构造函数的自动生成，必须至少声明一个实例构造函数。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.11.6 可选的实例构造函数形参

this(...) 形式的构造函数初始值设定项通常与重载一起使用，以实现可选的实例构造函数形参。在下面的示例中

```
class Text
{
   public Text(): this(0, 0, null) {}

   public Text(int x, int y): this(x, y, null) {}

   public Text(int x, int y, string s) {
      // Actual constructor implementation
   }
}
```

前两个实例构造函数只为调用中没有传递过来的实参提供相应的默认值。这两个构造函数都使用 this(...) 构造函数初始值设定项来调用实际完成初始化新实例工作的第三个实例构造函数。这样，实际效果就是该实例构造函数具有可选的形参：

```
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");
```

10.12 静态构造函数

静态构造函数 (static constructor) 是一种用于实现初始化封闭式类类型所需操作的成员。静态构造函数是使用 *static-constructor-declaration* 来声明的：

static-constructor-declaration:
attributes$_{opt}$ *static-constructor-modifiers identifier* ( ) *static-constructor-body*

static-constructor-modifiers:
extern$_{opt}$ static
static extern$_{opt}$

static-constructor-body:
block
;

static-constructor-declaration 可包含一组 *attributes*（第 17 章）和一个 extern 修饰符（第 10.6.7 节）。

static-constructor-declaration 的 *identifier* 必须是声明了该静态构造函数的那个类的名称。如果指定了任何其他名称，则发生编译时错误。

当静态构造函数声明包含 extern 修饰符时，称该静态构造函数为外部静态构造函数 (external static constructor)。因为外部静态构造函数声明不提供任何实际的实现，所以它的 *static-constructor-body* 由一个分号组成。对于所有其他的静态构造函数声明，*static-constructor-body* 都是一个 *block*，它指定当初始化该类时需要执行的语句。这正好相当于具有 void 返回类型的静态方法的 *method-body*（第 10.6.10 节）。

静态构造函数是不可继承的，而且不能被直接调用。

封闭式类类型的静态构造函数在给定应用程序域中至多执行一次。应用程序域中第一次发生以下事件时将触发静态构造函数的执行：

- 创建类类型的实例。
- 引用类类型的任何静态成员。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

如果类中包含用来开始执行的 Main 方法（第 3.1 节），则该类的静态构造函数将在调用 Main 方法之前执行。

若要初始化新的封闭式类类型，需要先为该特定的封闭类型创建一组新的静态字段（第 10.5.1 节）。将其中的每个静态字段初始化为默认值（第 5.2 节）。下一步，为这些静态字段执行静态字段初始值设定项（第 10.4.5.1 节）。最后，执行静态构造函数。

下面的示例

```
using System;

class Test
{
   static void Main() {
      A.F();
      B.F();
   }
}

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

class B
{
   static B() {
      Console.WriteLine("Init B");
   }
   public static void F() {
      Console.WriteLine("B.F");
   }
}
```

一定产生输出：

```
Init A
A.F
Init B
B.F
```

因为 A 的静态构造函数的执行是通过调用 A.F 触发的，而 B 的静态构造函数的执行是通过调用 B.F 触发的。

上述过程有可能构造出循环依赖关系，其中，带有变量初始值设定项的静态字段能够在其处于默认值状态时被观测。

下面的示例

```
using System;

class A
{
   public static int X;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   static A() {
      X = B.Y + 1;
   }
}
class B
{
   public static int Y = A.X + 1;
   static B() {}
   static void Main() {
      Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
   }
}
```

产生输出

```
X = 1, Y = 2
```

要执行 Main 方法，系统在运行类 B 的静态构造函数之前首先要运行 B.Y 的初始值设定项。因为引用了 A.X 的值，所以 Y 的初始值设定项导致运行 A 的静态构造函数。这样，A 的静态构造函数将继续计算 X 的值，从而获取 Y 的默认值 0，而 A.X 被初始化为 1。这样就完成了运行 A 的静态字段初始值设定项和静态构造函数的进程，控制返回到 Y 的初始值的计算，计算结果变为 2。

由于静态构造函数只为每个封闭构造类类型执行一次，因此对于无法通过约束（第 10.1.5 节）在编译时进行检查的类型形参来说，此处是强制进行运行时检查的方便位置。例如，下面的类型使用静态构造函数检查类型实参是否为一个枚举：

```
class Gen<T> where T: struct
{
   static Gen() {
      if (!typeof(T).IsEnum) {
         throw new ArgumentException("T must be an enum");
      }
   }
}
```

10.13 析构函数

析构函数 (destructor) 是一种用于实现销毁类实例所需操作的成员。析构函数是用 *destructor-declaration* 来声明的：

destructor-declaration:
attributes$_{opt}$ `extern`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*

destructor-body:
block
`;`

destructor-declaration 可以包括一组 *attributes*（第 17 章）。

destructor-declarator 的 *identifier* 必须就是声明了该析构函数的那个类的名称。如果指定了任何其他名称，则发生编译时错误。

当析构函数声明包含 extern 修饰符时，称该析构函数为外部析构函数 (external destructor)。因为外部析构函数声明不提供任何实际的实现，所以它的 *destructor-body* 由一个分号组成。对于所有其他析构函数，*destructor-body* 都由一个 *block* 组成，它指定当销毁该类的一个实例时需要执行的语句。*destructor-body* 正好对应于具有 void 返回类型（第 10.6.10 节）的实例方法的 *method-body*。

析构函数是不可继承的。因此，除了自己所声明的析构函数外，一个类不具有其他析构函数。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由于析构函数要求不能带有形参，因此它不能被重载，所以一个类至多只能有一个析构函数。

析构函数是自动调用的，它不能被显式调用。当任何代码都不再可能使用一个实例时，该实例就符合被销毁的条件。此后，它所对应的实例析构函数随时均可能被调用。销毁一个实例时，按照从派生程度最大到派生程度最小的顺序，调用该实例的继承链中的各个析构函数。析构函数可以在任何线程上执行。有关控制何时及如何执行析构函数的规则的进一步讨论，请参见第 3.9 节。

下列示例的输出

```
using System;
class A
{
   ~A() {
      Console.WriteLine("A's destructor");
   }
}
class B: A
{
   ~B() {
      Console.WriteLine("B's destructor");
   }
}
class Test
{
   static void Main() {
      B b = new B();
      b = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();
   }
}
```

为

```
B's destructor
A's destructor
```

这是由于继承链中的析构函数是按照从派生程度最大到派生程度最小的顺序调用的。

析构函数是通过重写 System.Object 中的虚方法 Finalize 实现的。C# 程序中不允许重写此方法或直接调用它（或它的重写）。例如，下列程序

```
class A
{
   override protected void Finalize() {}  // error
   public void F() {
      this.Finalize();                    // error
   }
}
```

包含两个错误。

编译器的行为就像此方法和它的重写根本不存在一样。因此，以下程序：

```
class A
{
   void Finalize() {}                     // permitted
}
```

是有效的，所显示的方法隐藏了 System.Object 的 Finalize 方法。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

有关从析构函数引发异常时的行为的讨论，请参见第 16.3 节。

10.14 迭代器

使用迭代器块（第 7.5 节）实现的函数成员（第 8.2 节）称为迭代器 (iterator)。

只要相应函数成员的返回类型是枚举器接口（第 10.14.1 节）之一或可枚举接口（第 10.14.2 节）之一，迭代器块就可用作该函数成员的函数体。它可以作为 *method-body*、*operator-body* 或 *accessor-body* 出现，而不能将事件、实例构造函数、静态构造函数和析构函数作为迭代器来实现。

当使用迭代器块实现函数成员时，为该函数成员的形参列表指定任何 `ref` 或 `out` 形参将产生编译时错误。

10.14.1 枚举器接口

枚举器接口 (enumerator interface) 为非泛型接口 `System.Collections.IEnumerator` 和泛型接口 `System.Collections.Generic.IEnumerator<T>` 的所有实例化。为简洁起见，本章中将这些接口分别表示为 `IEnumerator` 和 `IEnumerator<T>`。

10.14.2 可枚举接口

可枚举接口 (enumerable interface) 为非泛型接口 `System.Collections.IEnumerable` 和泛型接口 `System.Collections.Generic.IEnumerable<T>` 的所有实例化。为简洁起见，本章中将这些接口分别表示为 `IEnumerable` 和 `IEnumerable<T>`。

10.14.3 产生类型

迭代器产生一系列值，所有值的类型均相同。此类型称为迭代器的产生类型 (yield type)。

- 返回 `IEnumerator` 或 `IEnumerable` 的迭代器的产生类型是 `object`。
- 返回 `IEnumerator<T>` 或 `IEnumerable<T>` 的迭代器的产生类型是 `T`。

10.14.4 枚举器对象

如果返回枚举器接口类型的函数成员是使用迭代器块实现的，调用该函数成员不会立即执行迭代器块中的代码。而是先创建并返回一个枚举器对象 (enumerator object)。此对象封装了在迭代器块中指定的代码，并且在调用该枚举器对象的 `MoveNext` 方法时执行该迭代器块中的代码。枚举器对象具有下列特点：

- 它实现了 `IEnumerator` 和 `IEnumerator<T>`，其中 `T` 为迭代器的产生类型。
- 它实现了 `System.IDisposable`。
- 它以传递给该函数成员的实参值（如果存在）和实例值的副本进行初始化。
- 它有四种可能的状态：运行前 (before)、运行中 (running)、挂起 (suspended) 和运行后 (after)，并且初始状态为运行前 (before) 状态。

枚举器对象通常是编译器生成的枚举器类的一个实例，它封装了迭代器块中的代码，并实现了枚举器接口，但也可能实现其他方法。如果枚举器类由编译器生成，则该类将直接或间接嵌套在包含该函数成员的类中，它将具有私有可访问性，并且它将具有一个供编译器使用的保留名称（第 2.4.2 节）。

枚举器对象可实现除上面指定的那些接口以外的其他接口。

下面的各节将描述由枚举器对象所提供的 `IEnumerable` 和 `IEnumerable<T>` 接口实现的 `MoveNext`、`Current` 和 `Dispose` 成员的确切行为。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

请注意，枚举器对象不支持 IEnumerator.Reset 方法。调用此方法将导致引发 System.NotSupportedException。

10.14.4.1 MoveNext 方法

枚举器对象的 MoveNext 方法封装了迭代器块的代码。调用 MoveNext 方法将执行迭代器块中的代码，并相应设置枚举器对象的 Current 属性。MoveNext 执行的具体操作取决于调用 MoveNext 时的枚举器对象的状态：

- 如果枚举器对象的状态为运行前 (before)，则调用 MoveNext 会：
  - 将状态更改为运行中 (running)。
  - 将迭代器块的形参（包括 this）初始化为实参值以及初始化该枚举器对象时所保存的实例值。
  - 从头开始执行迭代器块，直到执行被中断（如后文所述）。
- 如果枚举器对象的状态为运行中 (running)，则调用 MoveNext 的结果不确定。
- 如果枚举器对象的状态为挂起 (suspended)，则调用 MoveNext 将：
  - 将状态更改为运行中 (running)。
  - 将所有局部变量和形参（包括 this）的值恢复为迭代器块的执行上次挂起时保存的值。注意，这些变量所引用对象的内容可能自上次调用 MoveNext 之后已经发生更改。
  - 恢复执行紧跟在引起执行挂起的 yield return 语句后面的迭代器块，并一直继续，直到执行中断（如后文所述）。
- 如果枚举器对象的状态为运行后 (after)，则调用 MoveNext 将返回 false。

当 MoveNext 执行迭代器块时，可以采用四种方式来中断执行：通过 yield return 语句、通过 yield break 语句、到达迭代器块的末尾以及引发异常并将异常传播到迭代器块之外。

- 当遇到 yield return 语句时（第 8.14 节）：
  - 计算该语句中给出的表达式，隐式转换为产生类型，并赋给枚举器对象的 Current 属性。
  - 迭代器体的执行被挂起。所有局部变量和形参（包括 this）的值被保存，此 yield return 语句的位置也被保存。如果 yield return 语句在一个或多个 try 块内，则此时与之关联的 finally 块将 *不*会执行。
  - 枚举器对象的状态更改为挂起 (suspended)。
  - MoveNext 方法向其调用方返回 true，指示迭代成功前进至下一个值。
- 当遇到 yield break 语句时（第 8.14 节）：
  - 如果 yield break 语句在一个或多个 try 块内，则与之关联的 finally 块将执行。
  - 枚举器对象的状态更改为运行后 (after)。
  - MoveNext 方法向其调用方返回 false，指示迭代完成。
- 当遇到迭代器体的结束处时：
  - 枚举器对象的状态更改为运行后 (after)。
  - MoveNext 方法向其调用方返回 false，指示迭代完成。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 当引发异常并传播到迭代器块之外时：
  - 通过异常传播机制执行迭代器体内的相应 `finally` 块。
  - 枚举器对象的状态更改为运行后 (after)。
  - 异常继续传播至 `MoveNext` 方法的调用方。

10.14.4.2 Current 属性

枚举器对象的 `Current` 属性将受迭代器块中的 `yield return` 语句影响。

当枚举器对象处于挂起 (suspended) 状态时，`Current` 的值为上一次调用 `MoveNext` 时设置的值。当枚举器对象处于运行前 (before)、运行中 (running) 或运行后 (after) 状态时，访问 `Current` 的结果不确定。

对于产生类型不是 `object` 的迭代器，通过枚举器对象的 `IEnumerable` 实现来访问 `Current` 的结果对应于通过枚举器对象的 `IEnumerator<T>` 实现来访问 `Current` 并将该结果强制转换为 `object`。

10.14.4.3 Dispose 方法

`Dispose` 方法用于通过使枚举器对象变为运行后 (after) 状态来清除迭代。

- 如果枚举器对象的状态为运行前 (before)，则调用 `Dispose` 将把状态更改为运行后 (after)。
- 如果枚举器对象的状态为运行中 (running)，则调用 `Dispose` 的结果不确定。
- 如果枚举器对象的状态为挂起 (suspended)，则调用 `Dispose` 将：
  - 将状态更改为运行中 (running)。
  - 执行所有 finally 块，就像最后执行的 `yield return` 语句是 `yield break` 语句一样。如果这导致引发异常，并且异常传播到迭代器体之外，则枚举器对象的状态设置为运行后 (after)，并且将异常传播到 `Dispose` 方法的调用方。
  - 将状态更改为运行后 (after)。
- 如果枚举器对象的状态为运行后 (after)，则调用 `Dispose` 没有任何作用。

10.14.5 可枚举对象

如果返回可枚举接口类型的函数成员是使用迭代器块实现的，调用该函数成员不会立即执行迭代器块中的代码。而是先创建并返回一个可枚举对象 (enumerable object)。可枚举对象的 `GetEnumerator` 方法返回一个封装有迭代器块中指定的代码的枚举器对象，当调用该枚举器对象的 `MoveNext` 方法时，将执行迭代器块中的代码。可枚举对象具有下列特点：

- 它实现了 `IEnumerable` 和 `IEnumerable<T>`，其中 `T` 为迭代器的产生类型。
- 它以传递给该函数成员的实参值（如果存在）和实例值的副本进行初始化。

可枚举对象通常是编译器生成的可枚举类的实例，它封装了迭代器块中的代码，并实现了可枚举接口，但也可能实现其他方法。如果可枚举类由编译器生成，则该类将直接或间接嵌套在包含该函数成员的类中，它将具有私有可访问性，并且它将具有供编译器使用的保留名称（第 2.4.2 节）。

可枚举对象可实现除上面指定的那些接口以外的其他接口。具体而言，可枚举对象还可实现 `IEnumerator` 和 `IEnumerator<T>`，从而使其既可作为可枚举对象，也可作为枚举器对象。在该类型的实现中，首次调用可枚举对象的 `GetEnumerator` 方法时，将返回可枚举对象本身。对可枚举对象的 `GetEnumerator` 的后续调用（如果存在），将返回可枚举对象的副本。因此，每个返回的枚举器都有自己的状态，一个枚举器中的更改不会影响其他枚举器。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

10.14.5.1 GetEnumerator 方法

可枚举对象实现了 `IEnumerable` 和 `IEnumerable<T>` 接口的 `GetEnumerator` 方法。这两种 `GetEnumerator` 方法的实现是相同的，都是获取并返回一个可用的枚举器对象。枚举器对象是以初始化该可枚举对象时保存的实例值和实参值进行初始化的，此外，枚举器对象函数如第 10.14.4 节所述。

10.14.6 实现示例

本节从标准 C# 构造的角度描述迭代器可能的实现。此处所描述的实现基于 Microsoft C# 编译器所使用的相同原理，但决非是强制性的实现方式，也不是唯一可能的实现方式。

下面的 `Stack<T>` 类使用一个迭代器实现其 `GetEnumerator` 方法。该迭代器以自顶向下的顺序枚举堆栈的元素。

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
   T[] items;
   int count;

   public void Push(T item) {
      if (items == null) {
         items = new T[4];
      }
      else if (items.Length == count) {
         T[] newItems = new T[count * 2];
         Array.Copy(items, 0, newItems, 0, count);
         items = newItems;
      }
      items[count++] = item;
   }

   public T Pop() {
      T result = items[--count];
      items[count] = default(T);
      return result;
   }

   public IEnumerator<T> GetEnumerator() {
      for (int i = count - 1; i >= 0; --i) yield return items[i];
   }
}
```

`GetEnumerator` 方法可转换为编译器生成的枚举器类的实例化，该类封装了迭代器块中的代码，如下所示。

```
class Stack<T>: IEnumerable<T>
{
   ...

   public IEnumerator<T> GetEnumerator() {
      return new __Enumerator1(this);
   }

   class __Enumerator1: IEnumerator<T>, IEnumerator
   {
      int __state;
      T __current;
      Stack<T> __this;
      int i;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
            public __Enumerator1(Stack<T> __this) {
               this.__this = __this;
            }

            public T Current {
               get { return __current; }
            }

            object IEnumerator.Current {
               get { return __current; }
            }

            public bool MoveNext() {
               switch (__state) {
                  case 1: goto __state1;
                  case 2: goto __state2;
               }
               i = __this.count - 1;
            __loop:
               if (i < 0) goto __state2;
               __current = __this.items[i];
               __state = 1;
               return true;
            __state1:
               --i;
               goto __loop;
            __state2:
               __state = 2;
               return false;
            }

            public void Dispose() {
               __state = 2;
            }

            void IEnumerator.Reset() {
               throw new NotSupportedException();
            }
         }
      }
```

在前面的转换中，迭代器块中的代码转换为状态机，并置于枚举器类的 `MoveNext` 方法中。此外，局部变量 `i` 转换为枚举器对象中的字段，这样它就可以在 `MoveNext` 的多次调用之间继续存在。

下面的示例打印整数 1 到 10 的简单乘法表。该示例中的 `FromTo` 方法使用迭代器实现，并且返回一个可枚举对象。

```
using System;
using System.Collections.Generic;

class Test
{
   static IEnumerable<int> FromTo(int from, int to) {
      while (from <= to) yield return from++;
   }

   static void Main() {
      IEnumerable<int> e = FromTo(1, 10);
      foreach (int x in e) {
         foreach (int y in e) {
            Console.Write("{0,3} ", x * y);
         }
         Console.WriteLine();
      }
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

FromTo 方法可转换为编译器生成的可枚举类的实例化，该类封装了迭代器块中的代码，如下所示。

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
   ...
   static IEnumerable<int> FromTo(int from, int to) {
      return new __Enumerable1(from, to);
   }

   class __Enumerable1:
      IEnumerable<int>, IEnumerable,
      IEnumerator<int>, IEnumerator
   {
      int __state;
      int __current;
      int __from;
      int from;
      int to;
      int i;

      public __Enumerable1(int __from, int to) {
         this.__from = __from;
         this.to = to;
      }

      public IEnumerator<int> GetEnumerator() {
         __Enumerable1 result = this;
         if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
            result = new __Enumerable1(__from, to);
            result.__state = 1;
         }
         result.from = result.__from;
         return result;
      }

      IEnumerator IEnumerable.GetEnumerator() {
         return (IEnumerator)GetEnumerator();
      }

      public int Current {
         get { return __current; }
      }

      object IEnumerator.Current {
         get { return __current; }
      }

      public bool MoveNext() {
         switch (__state) {
         case 1:
            if (from > to) goto case 2;
            __current = from++;
            __state = 1;
            return true;
         case 2:
            __state = 2;
            return false;
         default:
            throw new InvalidOperationException();
         }
      }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

可枚举类同时实现了可枚举接口和枚举器接口，使其既可作为可枚举对象，也可作为枚举器对象。首次调用 GetEnumerator 方法时将返回该可枚举对象本身。对可枚举对象的 GetEnumerator 的后续调用（如果存在），将返回可枚举对象的副本。因此，每个返回的枚举器都有自己的状态，一个枚举器中的更改不会影响其他枚举器。Interlocked.CompareExchange 方法用于确保线程安全操作。

from 和 to 形参转换为可枚举类中的字段。因为 from 是在迭代器块中修改的，所以引入附加 __from 字段以保存提供给每个枚举器中的 from 的初始值。

如果在 __state 为 0 时调用 MoveNext 方法，则该方法将引发 InvalidOperationException。这可防止在未事先调用 GetEnumerator 的情况下将可枚举对象用作枚举器对象。

下面的示例演示一个简单的树类。Tree<T> 类使用一个迭代器实现其 GetEnumerator 方法。迭代器按照中缀顺序枚举树的元素。

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   static void Main() {
      Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
      foreach (int i in ints) Console.Write("{0} ", i);
      Console.WriteLine();

      Tree<string> strings = MakeTree(
         "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
      foreach (string s in strings) Console.Write("{0} ", s);
      Console.WriteLine();
   }
}
```

GetEnumerator 方法可转换为编译器生成的枚举器类的实例化，该类封装了迭代器块中的代码，如下所示。

```
class Tree<T>: IEnumerable<T>
{
   ...
   public IEnumerator<T> GetEnumerator() {
      return new __Enumerator1(this);
   }
   class __Enumerator1 : IEnumerator<T>, IEnumerator
   {
      Node<T> __this;
      IEnumerator<T> __left, __right;
      int __state;
      T __current;

      public __Enumerator1(Node<T> __this) {
         this.__this = __this;
      }

      public T Current {
         get { return __current; }
      }

      object IEnumerator.Current {
         get { return __current; }
      }

      public bool MoveNext() {
         try {
            switch (__state) {

            case 0:
               __state = -1;
               if (__this.left == null) goto __yield_value;
               __left = __this.left.GetEnumerator();
               goto case 1;

            case 1:
               __state = -2;
               if (!__left.MoveNext()) goto __left_dispose;
               __current = __left.Current;
               __state = 1;
               return true;

            __left_dispose:
               __state = -1;
               __left.Dispose();

            __yield_value:
               __current = __this.value;
               __state = 2;
               return true;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
                case 2:
                   __state = -1;
                   if (__this.right == null) goto __end;
                   __right = __this.right.GetEnumerator();
                   goto case 3;

                case 3:
                   __state = -3;
                   if (!__right.MoveNext()) goto __right_dispose;
                   __current = __right.Current;
                   __state = 3;
                   return true;

                __right_dispose:
                   __state = -1;
                   __right.Dispose();

                __end:
                   __state = 4;
                   break;

                }
             }
             finally {
                if (__state < 0) Dispose();
             }
             return false;
          }
          public void Dispose() {
             try {
                switch (__state) {

                case 1:
                case -2:
                   __left.Dispose();
                   break;

                case 3:
                case -3:
                   __right.Dispose();
                   break;

                }
             }
             finally {
                __state = 4;
             }
          }
          void IEnumerator.Reset() {
             throw new NotSupportedException();
          }
       }
    }
```

foreach 语句中使用的编译器生成的临时变量被提升为枚举器对象的__left 和__right 字段。枚举器对象的__state 字段得到了妥善的更新，以便在引发异常时正确地调用正确的Dispose() 方法。注意，不可能使用简单的 foreach 语句写入转换后的代码。

10.15 异步函数

带有 async 修饰符的方法（第 10.6 节）或匿名函数（第 7.15 节）称为***异步函数***。通常，术语***异步***用于描述具有 async 修饰符的任何类型的函数。

在异步函数的形参列表中指定任何 ref 或 out 参数会导致编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

异步方法的 *return-type* 必须为 `void` 或***任务类型***。任务类型为 `System.Threading.Tasks.Task`，且类型从 `System.Threading.Tasks.Task<T>` 构建。为简洁起见，本章中将这些类型分别表示为 `Task` 和 `Task<T>`。返回任务类型的异步方法称为返回任务的方法。

任务类型的确切定义是由实现定义的，但从语言的角度来看，任务类型的状态分为*未完成*、*成功*或*出错*。*出错*任务将记录相关异常。*succeeded* `Task<T>`记录类型 *T* 的结果。任务类型是可等待的，因此任务可以作为 await 表达式（第 7.7.7 节）的操作数。

异步函数调用可以通过其主体中的 await 表达式（第 7.7.7 节）挂起计算。计算可以在稍后通过***恢复委托***挂起 await 表达式时恢复。恢复委托的类型为 `System.Action`，调用它时，异步函数调用的计算将从 await 表达式的中止位置恢复。如果异步函数调用从未挂起，则该函数调用的***当前调用方***为原始调用方，否则为恢复委托的最近一次调用方。

10.15.1 返回任务的异步函数计算

调用返回任务的异步函数将导致生成返回的任务类型的实例。这称为异步函数的***返回任务***。任务最初处于*未完成*状态。

然后，异步函数体将一直计算，直到它挂起（由于到达 await 表达式）或终止，此时控制将返回到调用方，并返回任务。

当异步函数体终止时，返回的任务将脱离未完成状态：

- 如果由于到达 return 语句或函数体末尾而导致函数体终止，则任何结果值均将记录在返回任务中，并将该任务置于*成功*状态。
- 如果由于未捕获的异常（第 8.9.5 节）而导致函数体终止，该异常将记录在返回任务中，并将该任务置于*出错*状态。

10.15.2 返回 void 的异步函数计算

如果异步函数的返回类型是 `void`，则计算将在以下方面不同于上述函数：由于没有任务返回，该函数改为将完成状态和异常传递给当前线程的***同步上下文***。同步上下文的确切定义取决于实现，而且只是当前运行的线程所在“位置”的表示形式。返回 void 的异步函数的计算在开始、成功完成或导致引发未捕获的异常时，均会通知同步上下文。

这将使得上下文可以跟踪其下有多少返回 void 的异步函数正在运行，并决定如何传播由这些函数产生的异常。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

11. 结构

结构与类的相似之处在于，它们都表示可以包含数据成员和函数成员的数据结构。但是，与类不同，结构是一种值类型，并且不需要堆分配。结构类型的变量直接包含了该结构的数据，而类类型的变量所包含的只是对相应数据的一个引用（被引用的数据称为“对象”）。

结构对于具有值语义的小型数据结构尤为有用。复数、坐标系中的点或字典中的“键-值”对都是结构的典型示例。这些数据结构的关键之处在于：它们只有少量数据成员，它们不要求使用继承或引用标识，而且它们适合使用值语义（赋值时直接复制值而不是复制它的引用）方便地实现。

如第 4.1.4 节中所描述，C# 提供的简单类型，如 `int`、`double` 和 `bool`，实际上全都是结构类型。正如这些预定义类型是结构一样，也可以使用结构和运算符重载在C# 语言中实现新的“基元”类型。在本章结尾（第 11.4 节）给出了这种类型的两个示例。

11.1 结构声明

struct-declaration 是一种用于声明新结构的 *type-declaration*（第 9.6 节）：

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ `partial`$_{opt}$ `struct` *identifier* *type-parameter-list*$_{opt}$
struct-interfaces$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *struct-body* `;`$_{opt}$

struct-declaration 的组成结构如下：开头是一组可选 *attributes*（第 17 章），然后依次是一组可选 *struct-modifiers*（第 11.1.1 节）、可选 `partial` 修饰符、关键字 `struct` 和命名结构的 *identifier*、可选 *type-parameter-list* 规范（第 10.1.3 节）、可选 *struct-interfaces* 规范（第 11.1.2 节）、可选 *type-parameters-constraints-clauses* 规范（第 10.1.5 节）、*struct-body*（第 11.1.4 节），最后是一个分号（可选）。

11.1.1 结构修饰符

struct-declaration 可以根据需要包含一个结构修饰符序列：

struct-modifiers:
struct-modifier
struct-modifiers *struct-modifier*

struct-modifier:
`new`
`public`
`protected`
`internal`
`private`

同一修饰符在结构声明中出现多次是编译时错误。

结构声明的修饰符与类声明（第 10.1 节）的修饰符具有相同的意义。

11.1.2 分部修饰符

`partial` 修饰符指示该 *struct-declaration* 是分部类型声明。封闭命名空间或类型声明中同名的多个分部结构声明，按照第 10.2 节中指定的规则组合形成一个结构声明。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

11.1.3 结构接口

结构声明中可以含有一个*struct-interfaces*规范，这种情况下称该结构直接实现给定的接口类型。

struct-interfaces:
: *interface-type-list*

第13.4节对接口实现进行了进一步讨论。

11.1.4 结构体

结构的*struct-body*用于定义该结构所包含的成员。

struct-body:
{ *struct-member-declarations*$_{opt}$ }

11.2 结构成员

结构的成员由两部分组成：由结构的*struct-member-declaration*引入的成员，以及从类型`System.ValueType`继承的成员。

struct-member-declarations:
struct-member-declaration
struct-member-declarations struct-member-declaration

struct-member-declaration:
constant-declaration
field-declaration
method-declaration
property-declaration
event-declaration
indexer-declaration
operator-declaration
constructor-declaration
static-constructor-declaration
type-declaration

除了在第11.3节中指出的区别外，在从第10.3节到第10.14节中关于类成员的说明也适用于结构成员。

11.3 类和结构的区别

结构在以下几个重要方面和类是不同的：

- 结构是值类型（第11.3.1节）。
- 所有结构类型均从类`System.ValueType`（第11.3.2节）隐式继承。
- 对结构类型变量进行赋值意味着将创建所赋的值的一个副本（第11.3.3节）。
- 结构的默认值的计算如下：将所有值类型字段设置为它们的默认值，并将所有引用类型字段设置为`null`，这样就产生了该结构的默认值（第11.3.4节）。
- 使用装箱和取消装箱操作在结构类型和`object`之间进行转换（第11.3.5节）。
- 对于结构，`this`的意义不同（第7.6.7节）。
- 结构的实例字段声明中不允许包含变量初始值设定项（第11.3.7节）。
- 在结构中不能声明无形参的实例构造函数（第11.3.8节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 在结构中不允许声明析构函数（第 11.3.9 节）。

11.3.1 值语义

结构是值类型（第 4.1 节）且被称为具有值语义。另一方面，类是引用类型（第 4.2 节）且被称为具有引用语义。

结构类型的变量直接包含了该结构的数据，而类类型的变量所包含的只是对相应数据的一个引用（被引用的数据称为“对象”）。当结构 B 包含类型为 A 的实例字段，且 A 是结构类型时，A 依赖于 B 或从 B 构造的类型将产生编译时错误。如果结构 X 包含结构 Y 类型的实例字段，则 X ***直接依赖于*** a struct Y。从上述定义可以推出：一个结构所依赖的结构的完整集合就是此直接依赖于 (directly depends on) 关系的传递闭包。例如

```
struct Node
{
   int data;
   Node next; // error, Node directly depends on itself
}
```

是错误的，因为 Node 包含自身类型的实例字段。请再看一个示例

```
struct A { B b; }
struct B { C c; }
struct C { A a; }
```

是错误的，因为类型 A、 B 和 C 都彼此相互依赖。

对于类，两个变量可能引用同一对象，因此对一个变量进行的操作可能影响另一个变量所引用的对象。对于结构，每个变量都有它们自己的数据副本（除 ref 和 out 形参变量外），因此对一个变量的操作不可能影响其他变量。另外，由于结构不是引用类型，因此结构类型的值不可能为 null。

给定下列声明

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

代码段

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

输出值 10。将 a 赋值给 b 时将创建该值的副本，因此，b 不会受到为 a.x 赋值的影响。假如 Point 被改为声明为类，则输出将为 100，因为 a 和 b 引用同一对象。

11.3.2 继承

所有结构类型均从类 System.ValueType 隐式继承，后者又从类 object 继承。一个结构声明可以指定实现的接口列表，但是不能指定基类。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

结构类型永远不会是抽象的，并且始终是隐式密封的。因此在结构声明中不允许使用 abstract 和 sealed 修饰符。

由于对结构不支持继承，所以结构成员的声明可访问性不能是 protected 或 protected internal。

结构中的函数成员不能是 abstract 或 virtual，而 override 修饰符只适用于重写从 System.ValueType 继承的方法。

11.3.3 赋值

对结构类型变量进行赋值意味着将创建所赋的值的一个副本。这不同于对类类型变量的赋值，后者所复制的是引用，而不是复制由该引用所标识的对象。

与赋值类似，将结构作为值形参传递或者作为函数成员的结果返回时，也创建了该结构的一个副本。但是，结构仍可通过 ref 或 out 形参以引用方式传递给函数成员。

当结构的属性或索引器是赋值的目标时，与属性或索引器访问关联的实例表达式必须为变量类别。如果该实例表达式归类为值类别，则发生编译时错误。第 7.17.1 节对此进行了进一步详细的描述。

11.3.4 默认值

如第 5.2 节中所述，有几种变量在创建时自动初始化为它们的默认值。对于类类型和其他引用类型的变量，默认值为 null。但是，由于结构是不能为 null 的值类型，结构的默认值是通过将所有值类型字段设置为它们的默认值，并将所有引用类型字段设置为 null 而产生的值。

引用上面声明的 Point 结构，下面的示例

```
Point[] a = new Point[100];
```

将数组中的每个 Point 初始化为通过将 x 和 y 字段设置为零而产生的值。

结构的默认值对应于该结构的默认构造函数所返回的值（第 4.1.2 节）。与类不同，结构不允许声明无形参实例构造函数。相反，每个结构隐式地具有一个无形参实例构造函数，该构造函数始终返回相同的值，即通过将所有的值类型字段设置为它们的默认值，并将所有引用类型字段设置为 null 而得到的值。

设计一个结构时，要设法确保它的默认初始化状是有效的状态。在下面的示例中

```
using System;

struct KeyValuePair
{
   string key;
   string value;

   public KeyValuePair(string key, string value) {
      if (key == null || value == null) throw new ArgumentException();
      this.key = key;
      this.value = value;
   }
}
```

用户定义的实例构造函数不允许出现 null 值（除非在显式调用时）。但 KeyValuePair 变量可能会被初始化为它的默认值，这样，key 和 value 字段就都为 null，所以，设计该结构时，必须正确处理好此问题。

11.3.5 装箱和拆箱

一个类类型的值可以转换为 object 类型或由该类实现的接口类型，这只需在编译时把对应的引用当作另一个类型处理即可。与此类似，一个 object 类型的值或者接口类型的值也可以被转换回类类型而不必更改相应的引用。当然，在这种情况下，需要进行运行时类型检查。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

由于结构不是引用类型，上述操作对结构类型是以不同的方式实现的。当结构类型的值被转换为 object 类型或由该结构实现的接口类型时，就会执行一次装箱操作。反之，当 object 类型的值或接口类型的值被转换回结构类型时，会执行一次取消装箱操作。与对类类型进行的相同操作相比，主要区别在于：装箱操作会把相关的结构值复制为已被装箱的实例，而取消装箱则会从已被装箱的实例中复制出一个结构值。因此，在装箱或取消装箱操作后，对已取消装箱的结构进行的更改不会影响已装箱的结构。

当结构类型重写从 System.Object 继承的虚方法（如 Equals、GetHashCode 或 ToString）时，通过该结构类型的实例进行的虚方法调用不会导致装箱。即使将该结构用作类型形参，并且通过类型形参类型的实例进行调用，情况也是如此。例如：

```
using System;

struct Counter
{
   int value;

   public override string ToString() {
      value++;
      return value.ToString();
   }
}

class Program
{
   static void Test<T>() where T: new() {
      T x = new T();
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
   }

   static void Main() {
      Test<Counter>();
   }
}
```

该程序的输出为：

```
1
2
3
```

虽然让 ToString 具有副作用是一种不好的做法，但是该示例证明了 x.ToString() 的三个调用没有发生装箱。

类似地，在受约束的类型形参上访问成员时，从来不会隐式地进行装箱。例如，假设接口 ICounter 包含可用于修改值的方法 Increment。如果将 ICounter 用作约束，则将通过对调用了 Increment 的变量（而不是装箱副本）的引用来调用 Increment 方法的实现。

```
using System;

interface ICounter
{
   void Increment();
}

struct Counter: ICounter
{
   int value;

   public override string ToString() {
      return value.ToString();
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   void ICounter.Increment() {
      value++;
   }
}

class Program
{
   static void Test<T>() where T: ICounter, new() {
      T x = new T();
      Console.WriteLine(x);
      x.Increment();               // Modify x
      Console.WriteLine(x);
      ((ICounter)x).Increment();   // Modify boxed copy of x
      Console.WriteLine(x);
   }

   static void Main() {
      Test<Counter>();
   }
}
```

对 Increment 的第一个调用将修改变量 x 中的值。这与对 Increment 的第二个调用不等效，第二个调用修改 x 的装箱副本中的值。因此，该程序的输出为：

```
0
1
1
```

有关装箱和取消装箱的详细信息，请参见第 4.3 节。

11.3.6 this 的含义

在类的实例构造函数和实例函数成员中，this 为值类别。因此，虽然 this 可以用于引用该函数成员调用所涉及的实例，但是不可能在类的函数成员中对 this 本身赋值。

在结构的实例构造函数内，this 相当于一个结构类型的 out 参数，而在结构的实例函数成员内，this 相当于一个结构类型的 ref 参数。在这两种情况下，this 本身相当于一个变量，因而有可能对该函数成员调用所涉及的整个结构进行修改（如对 this 赋值，或者将 this 作为 ref 或 out 形参传递）。

11.3.7 字段初始值设定项

如第 11.3.4 节中所述，结构的默认值就是将所有值类型字段设置为它们的默认值并将所有引用类型字段设置为 null 而产生的值。由于这个原因，结构不允许它的实例字段声明中含有变量初始值设定项。此限制只适用于实例字段。在结构的静态字段声明中可以含有变量初始值设定项。

下面的示例

```
struct Point
{
   public int x = 1;  // Error, initializer not permitted
   public int y = 1;  // Error, initializer not permitted
}
```

出现错误，因为实例字段声明中含有变量初始值设定项。

11.3.8 构造函数

与类不同，结构不允许声明无形参实例构造函数。相反，每个结构隐式地具有一个无形参实例构造函数，该构造函数始终返回相同的值，即通过将所有的值类型字段设置为其默认值，并将所有引用类型字段设置为 null 而得到的值（第 4.1.2 节）。结构可以声明具有形参的实例构造函数。例如

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
struct Point
{
   int x, y;

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

已知以上声明，语句

```
Point p1 = new Point();

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

都将创建一个 Point，其 x 和 y 将初始化为零。

一个结构的实例构造函数不能含有 base(...) 形式的构造函数初始值设定项。

如果该结构实例构造函数没有指定构造函数初始值设定项，则 this 变量就相当于一个结构类型的 out 形参，并且，与 out 形参类似，this 必须在该构造函数返回的每个位置上明确赋值（第 5.3 节）。如果该结构实例构造函数指定了构造函数初始值设定项，则 this 变量就相当于结构类型的 ref 形参，并且，与 ref 形参类似，this 被视为在进入构造函数体时已被明确赋值。请研究下面的实例构造函数实现：

```
struct Point
{
   int x, y;

   public int X {
      set { x = value; }
   }

   public int Y {
      set { y = value; }
   }

   public Point(int x, int y) {
      X = x;      // error, this is not yet definitely assigned
      Y = y;      // error, this is not yet definitely assigned
   }
}
```

在被构造的结构的所有字段已明确赋值以前，不能调用任何实例成员函数（包括 X 和 Y 属性的 set 访问器）。但是请注意，如果 Point 是类而不是结构，则允许上述的实例构造函数实现。

11.3.9 析构函数

在结构类型中不允许声明析构函数。

11.3.10 静态构造函数

结构的静态构造函数与类的静态构造函数所遵循的规则大体相同。应用程序域中第一次发生以下事件时将触发结构类型的静态构造函数的执行：

- 引用结构类型的静态成员。
- 结构类型的显式声明的构造函数被调用。

创建结构类型的默认值（第 11.3.4 节）不会触发静态构造函数。（一个示例是数组中元素的初始值。）

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

11.4 结构示例

下面的内容展示了关于应用 struct 类型的两个重要示例，它们各自创建一个类型，这些类型使用起来就像 C# 语言的预定义类型，但具有修改了的语义。

11.4.1 数据库整数类型

下面的 DBInt 结构实现了一个整数类型，它可以表示 int 类型的值的完整集合，再加上一个用于表示未知值的附加状态。具有这些特征的类型常用在数据库中。

```
using System;

public struct DBInt
{
   // The Null member represents an unknown DBInt value.

   public static readonly DBInt Null = new DBInt();

   // When the defined field is true, this DBInt represents a known value
   // which is stored in the value field. When the defined field is false,
   // this DBInt represents an unknown value, and the value field is 0.

   int value;
   bool defined;

   // Private instance constructor. Creates a DBInt with a known value.

   DBInt(int value) {
      this.value = value;
      this.defined = true;
   }

   // The IsNull property is true if this DBInt represents an unknown value.

   public bool IsNull { get { return !defined; } }

   // The Value property is the known value of this DBInt, or 0 if this
   // DBInt represents an unknown value.

   public int Value { get { return value; } }

   // Implicit conversion from int to DBInt.

   public static implicit operator DBInt(int x) {
      return new DBInt(x);
   }

   // Explicit conversion from DBInt to int. Throws an exception if the
   // given DBInt represents an unknown value.

   public static explicit operator int(DBInt x) {
      if (!x.defined) throw new InvalidOperationException();
      return x.value;
   }

   public static DBInt operator +(DBInt x) {
      return x;
   }

   public static DBInt operator -(DBInt x) {
      return x.defined ? -x.value : Null;
   }

   public static DBInt operator +(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value + y.value: Null;
   }

   public static DBInt operator -(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value - y.value: Null;
   }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
    public static DBInt operator *(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value * y.value: Null;
    }

    public static DBInt operator /(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value / y.value: Null;
    }

    public static DBInt operator %(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value % y.value: Null;
    }

    public static DBBool operator ==(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value == y.value: DBBool.Null;
    }

    public static DBBool operator !=(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value != y.value: DBBool.Null;
    }

    public static DBBool operator >(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value > y.value: DBBool.Null;
    }

    public static DBBool operator <(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }

    public static DBBool operator >=(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }

    public static DBBool operator <=(DBInt x, DBInt y) {
       return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }

    public override bool Equals(object obj) {
       if (!(obj is DBInt)) return false;
       DBInt x = (DBInt)obj;
       return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
       return value;
    }

    public override string ToString() {
       return defined? value.ToString(): “DBInt.Null”;
    }
}
```

11.4.2 数据库布尔类型

下面的 DBBool 结构实现了一个三值逻辑类型。此类型的可能值有 DBBool.True、DBBool.False 和 DBBool.Null，其中 Null 成员用于表示未知值。这样的三值逻辑类型经常用在数据库中。

```
using System;

public struct DBBool
{
   // The three possible DBBool values.

   public static readonly DBBool Null = new DBBool(0);
   public static readonly DBBool False = new DBBool(-1);
   public static readonly DBBool True = new DBBool(1);

   // Private field that stores –1, 0, 1 for False, Null, True.

   sbyte value;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
// Private instance constructor. The value parameter must be -1, 0, or 1.
DBBool(int value) {
   this.value = (sbyte)value;
}

// Properties to examine the value of a DBBool. Return true if this
// DBBool has the given value, false otherwise.
public bool IsNull { get { return value == 0; } }

public bool IsFalse { get { return value < 0; } }

public bool IsTrue { get { return value > 0; } }

// Implicit conversion from bool to DBBool. Maps true to DBBool.True and
// false to DBBool.False.
public static implicit operator DBBool(bool x) {
   return x? True: False;
}

// Explicit conversion from DBBool to bool. Throws an exception if the
// given DBBool is Null, otherwise returns true or false.
public static explicit operator bool(DBBool x) {
   if (x.value == 0) throw new InvalidOperationException();
   return x.value > 0;
}

// Equality operator. Returns Null if either operand is Null, otherwise
// returns True or False.
public static DBBool operator ==(DBBool x, DBBool y) {
   if (x.value == 0 || y.value == 0) return Null;
   return x.value == y.value? True: False;
}

// Inequality operator. Returns Null if either operand is Null, otherwise
// returns True or False.
public static DBBool operator !=(DBBool x, DBBool y) {
   if (x.value == 0 || y.value == 0) return Null;
   return x.value != y.value? True: False;
}

// Logical negation operator. Returns True if the operand is False, Null
// if the operand is Null, or False if the operand is True.
public static DBBool operator !(DBBool x) {
   return new DBBool(-x.value);
}

// Logical AND operator. Returns False if either operand is False,
// otherwise Null if either operand is Null, otherwise True.
public static DBBool operator &(DBBool x, DBBool y) {
   return new DBBool(x.value < y.value? x.value: y.value);
}

// Logical OR operator. Returns True if either operand is True, otherwise
// Null if either operand is Null, otherwise False.
public static DBBool operator |(DBBool x, DBBool y) {
   return new DBBool(x.value > y.value? x.value: y.value);
}

// Definitely true operator. Returns true if the operand is True, false
// otherwise.
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        public static bool operator true(DBBool x) {
            return x.value > 0;
        }

        // Definitely false operator. Returns true if the operand is False, false
        // otherwise.
        public static bool operator false(DBBool x) {
            return x.value < 0;
        }

        public override bool Equals(object obj) {
            if (!(obj is DBBool)) return false;
            return value == ((DBBool)obj).value;
        }

        public override int GetHashCode() {
            return value;
        }

        public override string ToString() {
            if (value > 0) return "DBBool.True";
            if (value < 0) return "DBBool.False";
            return "DBBool.Null";
        }
    }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

12. 数组

数组是一种包含若干变量的数据结构，这些变量都可以通过计算索引进行访问。数组中包含的变量（又称数组的元素）具有相同的类型，该类型称为数组的元素类型。

数组有一个“秩”，它确定和每个数组元素关联的索引个数。数组的秩又称为数组的维度。“秩”为 1 的数组称为一维数组 (single-dimensional array)。“秩”大于 1 的数组称为多维数组 (multi-dimensional array)。维度大小确定的多维数组通常称为两维数组、三维数组等。

数组的每个维度都有一个关联的长度，它是一个大于或等于零的整数。维度的长度不是数组类型的组成部分，而只与数组类型的实例相关联，它是在运行时创建实例时确定的。维度长度确定该维度索引的有效范围：如果维度长度为 N，则索引的范围可以从 0 到 N – 1（包括 N – 1）。数组中的元素总数是数组中各维度长度的乘积。如果数组的一个或多个维度的长度为零，则称该数组为空。

数组的元素类型可以是任意类型，包括数组类型。

12.1 数组类型

数组类型表示为一个 *non-array-type* 后接一个或多个 *rank-specifier*：

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

non-array-type:
    type

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

rank-specifier:
    [*dim-separators*$_{opt}$]

dim-separators:
    ,
    dim-separators ,

non-array-type 是本身不是 *array-type* 的任意 *type*。

由 *array-type* 中最左侧的 *rank-specifier* 给定数组类型的秩：*rank-specifier* 表示该数组是其秩为 1 加上 *rank-specifier* 中的“,”标记个数的数组。

数组类型的元素类型就是去掉最左边的 *rank-specifier* 后剩余表达式的类型：

- 形式为 T[R] 的数组类型是秩为 R、元素类型为非数组元素类型 T 的数组。
- 形式为 T[R][R$_1$]...[R$_N$] 的数组类型是秩为 R、元素类型为 T[R$_1$]...[R$_N$] 的数组。

实质上，在解释数组类型时，先从左到右读取 *rank-specifier*，最后才读取那个最终的非数组元素类型。例如，类型 int[][,,][,] 表示一个一维数组，该一维数组的元素类型为三维数组，该三维数组的元素类型为二维数组，该二维数组的元素类型为 int。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在运行时，数组类型的值可以为null或对该数组类型的某个实例的引用。

12.1.1 System.Array 类型

System.Array 类型是所有数组类型的抽象基类型。存在从任何数组类型到 System.Array 的隐式引用转换（第 6.1.6 节），并且存在从 System.Array 到任何数组类型的显式引用转换（第 6.2.4) 节）。请注意，System.Array 本身不是 *array-type*。相反，它是一个从中派生所有 *array-type* 的 *class-type*。

在运行时，System.Array 类型的值可以是 null 或是对任何数组类型的实例的引用。

12.1.2 数组和泛型 IList 接口

一维数组 T[] 实现了接口 System.Collections.Generic.IList<T>（缩写为 IList<T>）及其基接口。相应地，存在从 T[] 到 IList<T> 及其基接口的隐式转换。此外，如果存在从 S 到 T 的隐式引用转换，则 S[] 实现 IList<T>，并且存在从 S[] 到 IList<T> 及其基接口的隐式引用转换（第 6.1.6 节）。如果存在从 S 到 T 的显式引用转换，则存在从 S[] 到 IList<T> 及其基接口的显式引用转换（第 6.2.4 节）。例如：

```
using System.Collections.Generic;

class Test
{
   static void Main() {
      string[] sa = new string[5];
      object[] oa1 = new object[5];
      object[] oa2 = sa;

      IList<string> lst1 = sa;                  // Ok
      IList<string> lst2 = oa1;                 // Error, cast needed
      IList<object> lst3 = sa;                  // Ok
      IList<object> lst4 = oa1;                 // Ok

      IList<string> lst5 = (IList<string>)oa1;  // Exception
      IList<string> lst6 = (IList<string>)oa2;  // Ok
   }
}
```

赋值操作 lst2 = oa1 将产生编译时错误，因为从 object[] 到 IList<string> 的转换是显式转换，不是隐式转换。强制转换 (IList<string>)oa1 会导致在运行时引发异常，因为 oa1 引用 object[] 而不是 string[]。但是，强制转换 (IList<string>)oa2 不会导致在运行时引发异常，因为 oa2 引用 string[]。

如果存在从 S[] 到 IList<T> 的隐式或显式引用转换，则也存在从 IList<T> 及其基接口到 S[] 的显式引用转换（第 6.2.4 节）。

当数组类型 S[] 实现 IList<T> 时，所实现的接口的有些成员可能会引发异常。该接口的实现的确切行为不在本规范讨论的范围之内。

12.2 数组创建

数组实例是由 *array-creation-expression*（第 7.6.10.4 节）创建的，或者是由包含 *array-initializer*（第 12.6 节）的字段声明或局部变量声明创建的。

创建数组实例时，将确定秩和各维度的长度，它们在该实例的整个生存期内保持不变。换言之，对于一个已存在的数组实例，既不能更改它的秩，也不可能调整它的维度大小。

数组实例一定是数组类型。System.Array 类型是不能实例化的抽象类型。

由 *array-creation-expression* 创建的数组的元素总是被初始化为它们的默认值（第 5.2 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

12.3 数组元素访问

数组元素使用形式为 A[I1, I2, ..., IN] 的 *element-access* 表达式（第 7.6.6.1 节）进行访问，其中 A 是数组类型的表达式，每个 IX 都是 int、uint、long、ulong 类型的表达式，或者可以隐式转换为这些类型的一种或多种类型。数组元素访问的结果是变量，即由下标选定的数组元素。

此外，还可以使用 foreach 语句（第 8.8.4) 节）来枚举数组的各个元素。

12.4 数组成员

每个数组类型均继承由 System.Array 类型声明的成员。

12.5 数组协变

对于任意两个 *reference-type* A 和 B，如果存在从 A 到 B 的隐式引用转换（第 6.1.6 节）或显式引用转换（第 6.2.4 节），则也一定存在从数组类型 A[R] 到数组类型 B[R] 的相同的引用转换，其中 R 可以是任何给定的 *rank-specifier*，但这两个数组类型必须使用相同的 R。这种关系称为数组协变。具体而言，数组协变意味着数组类型 A[R] 的值实际上可能是对数组类型 B[R] 的实例的引用（如果存在从 B 到 A 的隐式引用转换）。

由于存在数组协变，对引用类型数组的元素的赋值操作会包括一个运行时检查，以确保正在赋给数组元素的值确实是允许的类型（第 7.17.1 节）。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);
   }
}
```

Fill 方法中对 array[i] 的赋值隐式包括运行时检查，该检查可确保由 value 引用的对象是 null 或与 array 的实际元素类型兼容的实例。在 Main 中，Fill 的前两次调用会成功，但第三次调用在执行 array[i] 的第一个赋值操作时会引发 System.ArrayTypeMismatchException。发生此异常是因为装箱的 int 类型不能存储在 string 数组中。

具体而言，数组协变不能扩展至 *value-type* 的数组。例如，不存在允许将 int[] 当作 object[] 来处理的转换。

12.6 数组初始值设定项

数组初始值设定项可以在字段声明（第 10.5 节）、局部变量声明（第 8.5.1 节）和数组创建表达式（第 7.6.10.4 节）中指定：

array-initializer:
    { *variable-initializer-list*$_{opt}$ }
    { *variable-initializer-list* , }

variable-initializer-list:
    variable-initializer
    variable-initializer-list , *variable-initializer*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

variable-initializer:
expression
array-initializer

数组初始值设定项包含一系列变量初始值设定项，它们括在“{”和“}”标记中并且用“,”标记分隔。每个变量初始值设定项是一个表达式，或者（在多维数组的情况下）是一个嵌套的数组初始值设定项。

数组初始值设定项所在位置的上下文确定了正在被初始化的数组的类型。在数组创建表达式中，数组类型紧靠初始值设定项之前，或者由数组初始值设定项中的表达式推断得出。在字段或变量声明中，数组类型就是所声明的字段或变量的类型。当数组初始值设定项用在字段或变量声明中时，如：

```
int[] a = {0, 2, 4, 6, 8};
```

它只是下列等效数组创建表达式的简写形式：

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

对于一维数组，数组初始值设定项必须包含一个表达式序列，这些表达式是与数组的元素类型兼容的赋值表达式。这些表达式从下标为零的元素开始，按照升序初始化数组元素。数组初始值设定项中所含的表达式的数目确定正在创建的数组实例的长度。例如，上面的数组初始值设定项创建了一个长度为 5 的 `int[]` 实例并用下列值初始化该实例：

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

对于多维数组，数组初始值设定项必须具有与数组维数同样多的嵌套级别。最外面的嵌套级别对应于最左边的维度，而最里面的嵌套级别对应于最右边的维度。数组各维度的长度是由数组初始值设定项中相应嵌套级别内的元素数目确定的。对于每个嵌套的数组初始值设定项，元素的数目必须与同一级别的其他数组初始值设定项所包含的元素数相同。示例：

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

创建一个二维数组，其最左边的维度的长度为5，最右边的维度的长度为 2：

```
int[,] b = new int[5, 2];
```

然后用下列值初始化该数组实例：

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

如果指定非最右边的维度的长度为零，则假定后续维度的长度也为零。示例：

```
int[,] c = {};
```

创建一个二维数组，其最左边和最右边的维度的长度均为零。

```
int[,] c = new int[0, 0];
```

当数组创建表达式同时包含显式维度长度和一个数组初始值设定项时，长度必须是常量表达式，并且各嵌套级别的元素数目必须与相应的维度长度匹配。以下是几个示例：

```
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，由于维度长度表达式不是常量，因此 `y` 的初始值设定项导致编译时错误；另外由于初始值设定项中所设定的长度和元素数目不一致，`z` 的初始值设定项也导致编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

13. 接口

一个接口定义一个协定。实现某接口的类或结构必须遵守该接口定义的协定。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或结构必须提供的成员。

13.1 接口声明

interface-declaration 是用于声明新的接口类型的*type-declaration*（第 9.6 节）。

interface-declaration:
attributes$_{opt}$ *interface-modifiers*$_{opt}$ `partial`$_{opt}$ `interface`
identifier *variant-type-parameter-list*$_{opt}$ *interface-base*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *interface-body* `;`$_{opt}$

interface-declaration 由以下内容组成：一组可选 *attributes*（第 17 章），然后依次是一组可选 *interface-modifiers*（第 13.1.1 节）、可选 `partial` 修饰符、关键字 `interface` 和用于命名接口的 *identifier*、可选 *variant-type-parameter-list* 规范（第 13.1.3 节）、可选 *interface-base* 规范（第 13.1.4 节）、可选 *type-parameter-constraints-clauses* 规范（第 10.1.5 节）、*interface-body*（第 13.1.5 节），最后是一个分号（可选）。

13.1.1 接口修饰符

interface-declaration 可以根据需要包含一个接口修饰符序列：

interface-modifiers:
interface-modifier
interface-modifiers *interface-modifier*

interface-modifier:
`new`
`public`
`protected`
`internal`
`private`

同一修饰符在一个接口声明中多次出现属于编译时错误。

`new` 修饰符仅允许在类中定义的接口中使用。它指定接口隐藏同名的继承成员，详见第 10.3.4 节中的介绍。

`public`、`protected`、`internal` 和 `private` 修饰符将控制接口的可访问性。根据接口声明所在的上下文，只允许使用这些修饰符中的一部分（第 3.5.1 节）。

13.1.2 分部修饰符

`partial` 修饰符指示此 *interface-declaration* 为分部类型声明。封闭命名空间或类型声明中的多个同名分部接口声明按照第 10.2 节中指定的规则组合成一个接口声明。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

13.1.3 Variant 类型形参列表

Variant 类型形参列表只能在接口和委托类型上出现。与普通 *type-parameter-list* 的差异在于每个类型形参上的可选 *variance-annotation*。

variant-type-parameter-list:
    < *variant-type-parameters* >

variant-type-parameters:
    attributes$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*
    variant-type-parameters , *attributes*$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*

variance-annotation:
    in
    out

如果变化批注为 out，则类型形参称为协变(covariant)。如果变化批注为 in，则类型形参称为逆变(contravariant)。如果不存在变化批注，则类型形参称为是固定的(invariant)。

在下面的示例中

```
interface C<out X, in Y, Z>
{
  X M(Y y);

  Z P { get; set; }
}
```

X 为协变，Y 为逆变，而 Z 为固定的。

13.1.3.1 变化安全性

类型的类型形参列表中出现变化批注会限制类型在类型声明中可以出现的位置。

如果存在以下情况之一，则类型 T 是输出不安全(output-unsafe)的：

- T 为逆变类型形参
- T 为具有输出不安全元素类型的数组类型
- T 为从泛型类型 S<A_1,… A_K> 构造的接口或委托类型 S<X_1, .. X_K>，其中，对于至少一个 A_i，存在以下情况之一：
  - X_i 是协变或固定的，而 A_i 是输出不安全的。
  - X_i 是逆变或固定的，而 A_i 是输入安全的。

如果存在以下情况之一，则类型 T 是输入不安全(input-unsafe)的：

- T 是协变类型形参
- T 是具有输入不安全元素类型的数组类型
- T 为从泛型类型 S<A_1,… A_K> 构造的接口或委托类型 S<X_1, .. X_K>，其中，对于至少一个 A_i，存在以下情况之一：
  - X_i 是协变或固定的，而 A_i 是输入不安全的。
  - X_i 是逆变或固定的，而 A_i 是输出不安全的。

从直观上看，应在输出位置上禁止使用输出不安全类型，在输入位置上禁止使用输入不安全类型。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

类型如果不是输出不安全的，则是输出安全 (output-safe) 的，如果不是输入不安全的，则是输入安全 (input-safe) 的。

13.1.3.2 变化转换

变化批注用于提供到接口和委托类型的更加宽松（仍然类型安全）的转换。为此，隐式转换（第 0 节）和显式转换（第 6.2 节）的定义利用了变化可转换性的概念，该概念定义如下：

如果类型 T 是使用变化类型形参 $T<X_1, \cdots, X_n>$ 声明的接口或委托类型，并且对于每个变体类型形参 X_i，都存在以下情况之一，则类型 $T<A_1, \cdots, A_n>$ 可变化转换为类型 $T<B_1, \cdots, B_n>$：

- X_i 是协变的，且存在从 A_i 到 B_i 的隐式引用或标识转换
- X_i 是协变的，且存在从 B_i 到 A_i 的隐式引用或标识转换
- X_i 是固定的，且存在从 A_i 到 B_i 的标识转换

13.1.4 基接口

接口可以继承自零个或多个接口类型，这些类型称为接口的显式基接口 (explicit base interface)。当接口具有一个或多个显式基接口时，在该接口声明中，接口标识符后就要紧跟一个冒号以及一个由逗号分隔的基接口类型列表。

interface-base:
: *interface-type-list*

对于构造接口类型，显式基接口的组成方式为：采用泛型类型声明上的显式基接口声明，并将基接口声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument*。

接口的显式基接口的可访问性必须至少与接口本身相同（第 3.5.4 节）。例如，在 public 接口的 *interface-base* 中指定 private 或 internal 接口会导致编译时错误。

接口不能从自身直接或间接继承，否则会发生编译时错误。

接口的基接口是显式基接口及其基接口。换言之，基接口集是显式基接口、后者的显式基接口（依此类推）的完全可传递的闭包。接口继承其基接口的所有成员。在下面的示例中

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

IComboBox 的基接口是 IControl、ITextBox 和 IListBox。

换言之，上面的 IComboBox 接口继承成员 SetText 和 SetItems 以及 Paint。

接口的每个基接口都必须是输出安全（第 13.1.3.1 节）的。如果一个类或结构实现某接口，则它还隐式实现该接口的所有基接口。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

13.1.5 接口体

接口的 *interface-body* 定义接口的成员。

interface-body:
{ *interface-member-declarations*$_{opt}$ }

13.2 接口成员

接口的成员包括从基接口继承的成员和由接口本身声明的成员。

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

一个接口声明可以声明零个或多个成员。接口的成员必须是方法、属性、事件或索引器。接口不能包含常量、字段、运算符、实例构造函数、析构函数或类型，也不能包含任何种类的静态成员。

所有接口成员都隐式地具有 public 访问属性。接口成员声明中包含任何修饰符都属于编译时错误。具体来说，不能使用修饰符 `abstract`、`public`、`protected`、`internal`、`private`、`virtual`、`override` 或 `static` 来声明接口成员。

下面的示例

```
public delegate void StringListEvent(IStringList sender);

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

声明了一个接口，该接口的成员涵盖了所有可能作为接口成员的种类：方法、属性、事件和索引器。

interface-declaration 创建新的声明空间（第 3.3 节），并且 *interface-declaration* 直接包含的 *interface-member-declarations* 将新成员引入了该声明空间。以下规则适用于 *interface-member-declaration*：

- 方法的名称必须与同一接口中声明的所有属性和事件的名称不同。此外，方法的签名（第 3.6 节）不能与在同一接口中声明的其他所有方法的签名相同，并且在同一接口中声明的两种方法的签名不能只有 `ref` 和 `out` 不同。
- 属性或事件的名称必须与同一接口中声明的所有其他成员的名称不同。
- 一个索引器的签名必须区别于在同一接口中声明的其他所有索引器的签名。

准确地说，接口所继承的成员不是该接口的声明空间的一部分。因此，允许接口用与它所继承的成员相同的名称或签名来声明新的成员。发生这种情况时，则称派生的接口成员隐藏了基接口成员。隐藏一个继承的成员不算是错误，但这确实会导致编译器发出警告。为了避免出现上述警告，派生接口成员的声

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

明中必须包含一个 new 修饰符，以指示该派生成员将要隐藏对应的基成员。第 3.7.1.2 节对本主题进行了进一步讨论。

如果在不隐藏所继承成员的声明中包含 new 修饰符，将对此状况发出警告。通过移除 new 修饰符可取消显示此警告。

请注意，严格来讲，类 object 中的成员不是任何接口的成员（第 13.2 节）。但是，通过在任何接口类型中进行成员查找，可获得类 object 中的成员（第 7.4 节）。

13.2.1 接口方法

接口方法是使用 *interface-method-declaration* 来声明的：

interface-method-declaration:
attributes$_{opt}$ new$_{opt}$ *return-type* *identifier* *type-parameter-list*
(*formal-parameter-list*$_{opt}$) *type-parameter-constraints-clauses*$_{opt}$;

接口方法声明中的 *attributes*、*return-type*、*identifier* 和 *formal-parameter-list* 与类中方法声明的对应项（第 10.6 节）具有相同的意义。不允许接口方法声明指定方法体，因此，声明总是以分号结尾。

接口方法的每个形参类型都必须是输入安全（第 13.1.3.1 节）的，返回类型都必须为 void 或输出安全的。而且，方法的任何类型形参上的每个类类型约束、接口类型约束和类型形参约束都必须是输入安全的。

这些规则可确保接口的任何协变或逆变使用都保持类型安全。例如，

```
interface I<out T> { void M<U>() where U : T; }
```

是非法的，因为将 T 用作 U 上的类型形参约束不是输入安全的。

如果不应用此限制，则可能通过以下方式违反类型安全：

```
class B {}
class D : B {}
class E : B {}
class C : I<D> { public void M<U>() {…} }
…
I<B> b = new C();
b.M<E>();
```

实际这是对 C.M<E>. 的调用。但是该调用要求 E 派生自 D，因此会违反类型安全。

13.2.2 接口属性

接口属性是使用 *interface-property-declaration* 来声明的：

interface-property-declaration:
attributes$_{opt}$ new$_{opt}$ *type* *identifier* { *interface-accessors* }

interface-accessors:
attributes$_{opt}$ get ;
attributes$_{opt}$ set ;
attributes$_{opt}$ get ; *attributes*$_{opt}$ set ;
attributes$_{opt}$ set ; *attributes*$_{opt}$ get ;

接口属性声明中的 *attributes*、*type* 和 *identifier* 与类中属性声明的对应项（第 10.7 节）具有相同的意义。

接口属性声明的访问器与类属性声明（第 10.7.2 节）的访问器相对应，不同之处在于接口属性声明的访问器体必须始终是一个分号。因此，访问器在这里只用于表示该属性为读写、只读还是只写。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

如果存在 get 访问器，则接口属性的类型必须是输出安全的，如果存在 set 访问器，则必须是输入安全的。

13.2.3 接口事件

接口事件是使用 *interface-event-declarations* 来声明的：

interface-event-declaration:
attributes$_{opt}$ `new`$_{opt}$ `event` *type* *identifier* `;`

接口事件声明中的 *attributes*、*type* 和 *identifier* 与类中事件声明的对应项（第 10.8 节）具有相同的意义。

接口事件的类型必须是输入安全的。

13.2.4 接口索引器

接口索引器是使用 *interface-indexer-declaration* 来声明的：

interface-indexer-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* `this [` *formal-parameter-list* `] {` *interface-accessors* `}`

接口索引器声明中的 *attributes*、*type* 和 *formal-parameter-list* 与类中索引器声明的对应项（第 10.9 节）具有相同的意义。

接口索引器声明的访问器与类索引器声明（第 10.9 节）的访问器相对应，不同之处在于接口索引器声明的访问器体必须始终是一个分号。因此，访问器在这里只用于表示该索引器为读写、只读还是只写。

接口索引器的所有形参类型都必须是输入安全的。此外，所有 `out` 或 `ref` 形参类型也必须是输出安全的。请注意，由于基础执行平台的限制，甚至 `out` 形参也必须是输入安全的。

如果存在 get 访问器，则接口索引器的类型必须是输出安全的，如果存在 set 访问器，则必须是输入安全的。

13.2.5 接口成员访问

接口成员是通过 `I.M` 形式的成员访问（第 7.6.4 节）表达式和 `I[A]` 形式的索引器访问（第 7.6.6.2 节）表达式来访问的，其中 `I` 是接口类型，`M` 是该接口类型的方法、属性或事件，`A` 是索引器参数列表。

对于严格单一继承（继承链中的每个接口均恰巧有零个或一个直接基接口）的接口，成员查找（第 7.4 节）、方法调用（第 7.6.5.1 节）和索引器访问（第 7.6.6.2 节）规则的效果与类和结构的完全相同：派生程度较大的成员隐藏具有相同名称或签名的派生程度较小的成员。然而，对于多重继承接口，当两个或更多个不相关（互不继承）的基接口中声明了具有相同名称或签名的成员时，就会发生多义性。本节列出了此类情况的几个示例。在所有情况下，都可以使用显式强制转换来解决这种多义性。

在下面的示例中

```
interface IList
{
   int Count { get; set; }
}

interface ICounter
{
   void Count(int i);
}

interface IListCounter: IList, ICounter {}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class C
{
   void Test(IListCounter x) {
      x.Count(1);                // Error
      x.Count = 1;               // Error
      ((IList)x).Count = 1;      // Ok, invokes IList.Count.set
      ((ICounter)x).Count(1);    // Ok, invokes ICounter.Count
   }
}
```

由于在 IListCounter 中对 Count 的成员查找（第 7.4 节）所获得的结果是不明确的，因此前两个语句将引起编译时错误。如示例所阐释的，将 x 强制转换为适当的基接口类型就可以消除这种多义性。此类强制转换没有运行时开销，它们只是在编译时将该实例视为派生程度较小的类型而已。

在下面的示例中

```
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);                  // Invokes IInteger.Add
      n.Add(1.0);                // Only IDouble.Add is applicable
      ((IInteger)n).Add(1);      // Only IInteger.Add is a candidate
      ((IDouble)n).Add(1);       // Only IDouble.Add is a candidate
   }
}
```

通过应用第 7.5.3 节的重载决策规则，在调用 n.Add(1) 时将选择 IInteger.Add。类似地，在调用 n.Add(1.0) 时将选择 IDouble.Add。插入显式强制转换后，就只有一个候选方法了，因此没有多义性。

在下面的示例中

```
interface IBase
{
   void F(int i);
}

interface ILeft: IBase
{
   new void F(int i);
}

interface IRight: IBase
{
   void G();
}

interface IDerived: ILeft, IRight {}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class A
{
   void Test(IDerived d) {
      d.F(1);              // Invokes ILeft.F
      ((IBase)d).F(1);     // Invokes IBase.F
      ((ILeft)d).F(1);     // Invokes ILeft.F
      ((IRight)d).F(1);    // Invokes IBase.F
   }
}
```

IBase.F 成员将被 ILeft.F 成员隐藏。因此，调用 d.F(1) 时将选择 ILeft.F，即使 IBase.F 看起来未在通过 IRight 的访问路径中隐藏。

简单地说，多重继承接口中的直观隐藏规则是：如果成员在任一访问路径中被隐藏，那么它在所有访问路径中都被隐藏。因为从 IDerived 到 ILeft 到 IBase 的访问路径隐藏了 IBase.F，所以从 IDerived 到 IRight 到 IBase 的访问路径中也隐藏了该成员。

13.3 完全限定接口成员名

接口成员有时也用它的完全限定名 (fully qualified name) 来引用。接口成员的完全限定名是这样组成的：声明该成员的接口的名称，后接一个点，再后接该成员的名称。成员的完全限定名将引用声明该成员的接口。例如，给定下列声明

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

Paint 的完全限定名是 IControl.Paint，而 SetText 的完全限定名是 ITextBox.SetText。

在上面的示例中，不能用 ITextBox.Paint 来引用 Paint。

当接口是命名空间的组成部分时，该接口的成员的完全限定名需包含命名空间名称。例如

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

此处，Clone 方法的完全限定名是 System.ICloneable.Clone。

13.4 接口实现

接口可以由类和结构来实现。为了指示类或结构直接实现了某接口，在该类或结构的基类列表中应该包含该接口的标识符。例如：

```
interface ICloneable
{
   object Clone();
}
interface IComparable
{
   int CompareTo(object other);
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class ListEntry: ICloneable, IComparable
{
   public object Clone() {...}
   public int CompareTo(object other) {...}
}
```

如果一个类或结构直接实现某接口，则它还直接隐式实现该接口的所有基接口。即使在类或结构的基类列表中没有显式列出所有基接口，也是这样。例如：

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

此处，类 TextBox 同时实现了 IControl 和 ITextBox。

如果类 C 直接实现某个接口，则由 C 派生的所有类均隐式实现该接口。类声明中指定的基接口可以是构造接口类型（第 4.4 节）。基接口本身不能是类型形参，但在其作用域中可以包含类型形参。下面的代码演示类实现和扩展构造类型的方法：

```
class C<U,V> {}
interface I1<V> {}
class D: C<string,int>, I1<string> {}
class E<T>: C<int,T>, I1<T> {}
```

泛型类声明的基接口必须满足第 13.4.2 节中所述的唯一性规则。

13.4.1 显式接口成员实现

为了实现接口，类或结构可以声明显式接口成员实现 (explicit interface member implementation)。显式接口成员实现就是一种方法、属性、事件或索引器声明，它使用完全限定接口成员名称作为标识符。例如

```
interface IList<T>
{
   T[] GetElements();
}
interface IDictionary<K,V>
{
   V this[K key];
   void Add(K key, V value);
}
class List<T>: IList<T>, IDictionary<int,T>
{
   T[] IList<T>.GetElements() {...}
   T IDictionary<int,T>.this[int index] {...}
   void IDictionary<int,T>.Add(int index, T value) {...}
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

这里，IDictionary<int,T>.this 和 IDictionary<int,T>.Add 是显式接口成员实现。

某些情况下，接口成员的名称对于实现该接口的类可能是不适当的，此时，可以使用显式接口成员实现来实现该接口成员。例如，一个实现“文件抽象”的类一般会实现一个具有释放文件资源作用的Close成员函数，同时还可能使用显式接口成员实现来实现IDisposable接口的Dispose方法：

```
interface IDisposable
{
   void Dispose();
}

class MyFile: IDisposable
{
   void IDisposable.Dispose() {
      Close();
   }

   public void Close() {
      // Do what's necessary to close the file
      System.GC.SuppressFinalize(this);
   }
}
```

在方法调用、属性访问或索引器访问中，不能直接访问“显式接口成员实现”的成员，即使用它的完全限定名也不行。“显式接口成员实现”的成员只能通过接口实例来访问，并且在通过接口实例访问时，只能用该接口成员的名称来引用。

显式接口成员实现中包含访问修饰符属于编译时错误，而且如果包含abstract、virtual、override或static修饰符也属于编译时错误。

显式接口成员实现具有与其他成员不同的可访问性特征。由于显式接口成员实现永远不能在方法调用或属性访问中通过它们的完全限定名来访问，因此，它们似乎是private（私有的）。但是，因为它们可以通过接口实例来访问，所以它们似乎又是public（公共的）。

显式接口成员实现有两个主要用途：

- 由于显式接口成员实现不能通过类或结构实例来访问，因此它们就不属于类或结构的自身的公共接口。当需在一个公用的类或结构中实现一些仅供内部使用（不允许外界访问）的接口时，这就特别有用。
- 显式接口成员实现可以消除因同时含有多个相同签名的接口成员所引起的多义性。如果没有显式接口成员实现，一个类或结构就不可能为具有相同签名和返回类型的接口成员分别提供相应的实现，也不可能为具有相同签名和不同返回类型的所有接口成员中的任何一个提供实现。

为了使显式接口成员实现有效，声明它的类或结构必须在它的基类列表中指定一个接口，而该接口必须包含一个成员，该成员的完全限定名、类型和参数类型与该显式接口成员实现所具有的完全相同。因此，在下列类中

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

   int IComparable.CompareTo(object other) {...}   // invalid
}
```

IComparable.CompareTo声明将导致编译时错误，因为IComparable未列在Shape的基类列表中，并且不是ICloneable的基接口。与此类似，在下列声明中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

Ellipse 中的 ICloneable.Clone 声明也将导致编译时错误，因为 ICloneable 未在 Ellipse 的基类列表中显式列出。

接口成员的完全限定名必须引用声明该成员的接口。因此，下列声明中

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

Paint 的显式接口成员实现必须写为 IControl.Paint。

13.4.2 所实现接口的唯一性

泛型类型声明所实现的接口必须对所有可能的构造类型都保持唯一。如果没有此规则，则无法确定要为某些构造类型调用的正确方法。例如，假设允许以如下形式声明某个泛型类：

```
interface I<T>
{
   void F();
}
class X<U,V>: I<U>, I<V>                // Error: I<U> and I<V> conflict
{
   void I<U>.F() {...}
   void I<V>.F() {...}
}
```

如果允许这样，则无法确定要在下面的情况下执行的代码：

```
I<int> x = new X<int,int>();
x.F();
```

为了确定泛型类型声明的接口列表是否有效，将执行以下步骤：

- 假设 L 是泛型类、结构或接口声明 C 中直接指定的接口列表。
- 将已经在 L 中的接口的所有基接口添加到 L。
- 移除 L 中的所有重复接口。
- 在将类型实参替换到 L 中之后，如果从 C 创建的任何可能的构造类型导致 L 中的两个接口完全相同，则 C 的声明无效。在确定所有可能的构造类型时不考虑约束声明。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在上面的类声明 X 中，接口列表 L 由 I<U> 和 I<V> 组成。该声明无效，因为任何 U 和 V 属于相同类型的构造类型都将导致这两个接口成为完全相同的类型。

可以将不同继承级别指定的接口进行统一：

```
interface I<T>
{
   void F();
}
class Base<U>: I<U>
{
   void I<U>.F() {…}
}
class Derived<U,V>: Base<U>, I<V>   // Ok
{
   void I<V>.F() {…}
}
```

虽然 Derived<U,V> 同时实现了 I<U> 和 I<V>，但是此代码是有效的。代码

```
I<int> x = new Derived<int,int>();
x.F();
```

调用 Derived 中的方法，因为 Derived<int,int> 实际重新实现了 I<int>（第 13.4.6 节）。

13.4.3 泛型方法实现

当泛型方法隐式地实现接口方法时，为每个方法类型形参提供的约束必须在两个声明中是等效的（在将任何接口类型形参替换为相应的类型实参之后），其中方法的类型形参按序号位置从左到右进行标识。

但是，如果泛型方法显式实现接口方法，则不允许对实现方法使用任何约束。而是从接口方法继承约束

```
interface I<A,B,C>
{
   void F<T>(T t) where T: A;
   void G<T>(T t) where T: B;
   void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
   public void F<T>(T t) {...}                    // Ok
   public void G<T>(T t) where T: C {...}         // Ok
   public void H<T>(T t) where T: string {...}  // Error
}
```

方法 C.F<T> 隐式实现 I<object,C,string>.F<T> 方法。在此例中，C.F<T> 不需要（也不允许）指定约束 T: object，因为 object 是所有类型形参上的隐式约束。方法 C.G<T> 隐式地实现 I<object,C,string>.G<T>，因为在将接口类型形参替换为对应的类型实参之后，该约束与接口中的约束匹配。方法 C.H<T> 的约束是错误的，因为密封类型（在此例中为 string）不能用作约束。省略该约束也是错误的，因为需要对隐式接口方法实现的约束进行匹配。因此，隐式地实现 I<object,C,string>.H<T> 是不可能的。此接口方法只能使用显式接口成员实现来实现：

```
class C: I<object,C,string>
{
   ...
   public void H<U>(U u) where U: class {...}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
    void I<object,C,string>.H<T>(T t) {
       string s = t;  // Ok
       H<T>(t);
    }
}
```

在此例中，该显式接口成员实现调用严格具有更弱约束的公共方法。注意，虽然 T: string 约束无法在源代码中表示，但从 t 到 s 的赋值是有效的，因为 T 继承该约束。

13.4.4 接口映射

类或结构必须为它的基类列表中所列出的接口的所有成员提供它自己的实现。在进行实现的类或结构中定位接口成员的实现的过程称为接口映射 (interface mapping)。

关于类或结构 C 的接口映射就是查找 C 的基类列表中指定的每个接口的每个成员的实现。对某个特定接口成员 I.M 的实现（其中 I 是声明了成员 M 的接口）的定位按下述规则执行：从 C 开始，按继承顺序，逐个检查 C 的每个后续基类（下面用 S 表示每个进行检查的类或结构），直到找到匹配项：

- 如果 S 包含一个与 I 和 M 匹配的显式接口成员实现的声明，那么此成员就是 I.M 的实现。
- 否则，如果 S 包含与 M 匹配的非静态的 public 成员声明，则此成员就是 I.M 的实现。如果找到多个匹配成员，则无法确定哪个成员是 I.M 的实现。只有 S 是构造类型（在此情况下，泛型类型中声明的两个成员具有不同的签名，但类型参数却使他们的签名相同）时，才会出现此情况。

如果不能为在 C 的基类列表中指定的所有接口的所有成员找到实现，则将发生编译时错误。请注意，接口的成员包括那些从基接口继承的成员。

根据接口映射的含义，类成员 A 在下列情况下与接口成员 B 匹配：

- A 和 B 都是方法，并且 A 和 B 的名称、类型和形参表都相同。
- A 和 B 都是属性，A 和 B 的名称和类型相同，并且 A 与 B 具有相同的访问器（如果 A 不是显式接口成员实现，则它可以具有其他访问器）。
- A 和 B 都是事件，并且 A 和 B 的名称和类型相同。
- A 和 B 都是索引器，A 和 B 的类型和形参表相同，并且 A 与 B 具有相同的访问器（如果 A 不是显式接口成员实现，则它可以具有其他访问器）。

接口映射算法中隐含着下列值得注意的特征：

- 在类或结构成员中确定哪个实现了接口成员时，显式接口成员实现比同一个类或结构中的其他成员具有更高的优先级。
- 接口映射不涉及非公共成员和静态成员。

在下面的示例中

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

C 的 ICloneable.Clone 成员将成为 ICloneable 中 Clone 的实现，这是因为显式接口成员实现优先于其他成员。

如果类或结构实现两个或更多个接口，而这些接口包含具有相同名称、类型和参数类型的成员，则这些接口成员可以全部映射到单个类或结构成员上。例如

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

在此，IControl 和 IForm 的 Paint 方法均将映射到 Page 中的 Paint 方法。当然也可以为这两个方法提供单独的显式接口成员实现。

如果类或结构实现一个包含被隐藏成员的接口，那么一些成员必须通过显式接口成员实现来实现。例如

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

此接口的实现将至少需要一个显式接口成员实现，可采取下列形式之一

```
class C: IDerived
{
   int IBase.P { get {...} }
   int IDerived.P() {...}
}
class C: IDerived
{
   public int P { get {...} }
   int IDerived.P() {...}
}
class C: IDerived
{
   int IBase.P { get {...} }
   public int P() {...}
}
```

当一个类实现多个具有相同基接口的接口时，为该基接口提供的实现只能有一个。在下面的示例中

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


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
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) {...}
}
```

在基类列表中命名的 IControl、由 ITextBox 继承的 IControl 和由 IListBox 继承的 IControl 不可能有各自不同的实现。事实上，没有为这些接口提供单独实现的打算。相反，ITextBox 和 IListBox 的实现共享相同的 IControl 实现，因而可以简单地认为 ComboBox 实现了三个接口：IControl、ITextBox 和 IListBox。

基类的成员参与接口映射。在下面的示例中

```
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 实现中。

13.4.5 接口实现继承

类继承由其基类提供的所有接口实现。

如果不显式地重新实现 (re-implementing) 接口，派生类就无法以任何方式更改它从其基类继承的接口映射。例如，在下面的声明中

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

TextBox 中的 Paint 方法将隐藏 Control 中的 Paint 方法，但这种隐藏并不变更 Control.Paint 到 IControl.Paint 的映射，因此通过类实例和接口实例对 Paint 进行的调用将具有以下效果

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

但是，当接口方法被映射到类中的虚方法上时，从该类派生的类若重写了该虚方法，则将同时更改该接口的实现。例如，将上面的声明改写为

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

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

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

将产生下列效果

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

由于显式接口成员实现不能被声明为虚的，因此不可能重写显式接口成员实现。然而，显式接口成员实现的内部完全可以调用另一个方法，只要将该方法声明为虚方法，派生类就可以重写它了。例如

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

class Control: IControl
{
   void IControl.Paint() { PaintControl(); }

   protected virtual void PaintControl() {...}
}

class TextBox: Control
{
   protected override void PaintControl() {...}
}
```

在此，从 Control 派生的类可通过重写 PaintControl 方法来专用化 IControl.Paint 的实现。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

13.4.6 接口重新实现

一个类若继承了某个接口的实现，则只要将该接口列入它的基类列表中，就可以重新实现 (re-implement) 该接口。

接口的重新实现与接口的初始实现遵循完全相同的接口映射规则。因此，继承的接口映射不会对为重新实现该接口而建立的接口映射产生任何影响。例如，在下面的声明中

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

class Control: IControl
{
   void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
   public void Paint() {}
}
```

Control 将 IControl.Paint 映射到 Control.IControl.Paint 并不影响 MyControl 中的重新实现，该重新实现将 IControl.Paint 映射到 MyControl.Paint。

继承的公共成员声明和继承的显式接口成员声明可以参与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() {}
}
```

在此，Derived 中 IMethods 的实现将接口方法映射到 Derived.F、Base.IMethods.G、Derived.IMethods.H 和 Base.I。

当类实现接口时，它还隐式实现该接口的所有基接口。与此类似，接口的重新实现也同时隐式地对该接口的所有基接口进行重新实现。例如

```
interface IBase
{
   void F();
}

interface IDerived: IBase
{
   void G();
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class C: IDerived
{
   void IBase.F() {...}
   void IDerived.G() {...}
}
class D: C, IDerived
{
   public void F() {...}
   public void G() {...}
}
```

在此，IDerived 的重新实现也重新实现了 IBase，并将 IBase.F 映射到 D.F。

13.4.7 抽象类和接口

与非抽象类类似，抽象类也必须为在该类的基类列表中列出的接口的所有成员提供它自己的实现。但是，允许抽象类将接口方法映射到抽象方法上。例如

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

这里，IMethods 的实现将 F 和 G 映射到抽象方法上，这些抽象方法必须在从 C 派生的非抽象类中重写。

注意：显式接口成员实现本身不能是抽象的，但是当然允许显式接口成员实现调用抽象方法。例如

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

此处，要求从 C 派生的非抽象类重写 FF 和 GG，从而提供 IMethods 的实际实现。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

14. 枚举

枚举类型 (enum type) 是一种独特的值类型（第 4.1 节），它用于声明一组命名的常量。

下面的示例

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

声明一个名为 Color 的枚举类型，该类型具有三个成员：Red、Green 和 Blue。

14.1 枚举声明

枚举声明用于声明新的枚举类型。枚举声明以关键字 enum 开始，然后定义该枚举的名称、可访问性、基础类型和成员。

enum-declaration:
attributes$_{opt}$ *enum-modifiers*$_{opt}$ enum *identifier* *enum-base*$_{opt}$ *enum-body* ;$_{opt}$

enum-base:
: *integral-type*

enum-body:
{ *enum-member-declarations*$_{opt}$ }
{ *enum-member-declarations* , }

每个枚举类型都有一个相应的整型类型，称为该枚举类型的基础类型 (underlying type)。此基础类型必须能够表示该枚举中定义的所有枚举数值。枚举声明可以显式地声明 byte、sbyte、short、ushort、int、uint、long 或 ulong 的基础类型。请注意 char 不能用作基础类型。没有显式地声明基础类型的枚举声明意味着所对应的基础类型是 int。

下面的示例

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

声明了一个基础类型为 long 的枚举。开发人员可以像本示例一样选择使用 long 基础类型，以便能够使用在 long 范围内而不是在 int 范围内的值，或者保留此选项供将来使用。

14.2 枚举修饰符

enum-declaration 可以根据需要包含一个枚举修饰符序列：

enum-modifiers:
enum-modifier
enum-modifiers *enum-modifier*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

enum-modifier:
    new
    public
    protected
    internal
    private

同一修饰符在一个枚举声明中多次出现属于编译时错误。

枚举声明的修饰符与类声明（第 10.1.1 节）的修饰符具有同样的意义。然而请注意，在枚举声明中不允许使用 abstract 和 sealed 修饰符。枚举不能是抽象的，也不允许派生。

14.3 枚举成员

枚举类型声明体用于定义零个或多个枚举成员，这些成员是该枚举类型的命名常量。任意两个枚举成员不能具有相同的名称。

enum-member-declarations:
    enum-member-declaration
    enum-member-declarations , *enum-member-declaration*

enum-member-declaration:
    $attributes_{opt}$ *identifier*
    $attributes_{opt}$ *identifier* = *constant-expression*

每个枚举成员均具有相关联的常量值。此值的类型就是包含了它的那个枚举的基础类型。每个枚举成员的常量值必须在该枚举的基础类型的范围之内。下面的示例

```
enum Color: uint
{
   Red = -1,
   Green = -2,
   Blue = -3
}
```

会导致编译时错误，因为常量值 -1、-2 和 –3 不处于基础整型 uint 的范围中。

多个枚举成员可以共享同一个关联值。下面的示例

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

   Max = Blue
}
```

演示一个枚举，其中的两个枚举成员（Blue 和 Max）具有相同的关联值。

一个枚举成员的关联值或隐式地、或显式地被赋值。如果枚举成员的声明中具有 *constant-expression* 初始值设定项，则该常量表达式的值（它隐式转换为枚举的基础类型）就是该枚举成员的关联值。如果枚举成员的声明不具有初始值设定项，则它的关联值按下面规则隐式地设置：

- 如果枚举成员是在枚举类型中声明的第一个枚举成员，则它的关联值为零。
- 否则，枚举成员的关联值是通过将前一个枚举成员（按照文本顺序）的关联值加 1 得到的。这样增加后的值必须在该基础类型可表示的值的范围内；否则，会出现编译时错误。

下面的示例

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
using System;

enum Color
{
   Red,
   Green = 10,
   Blue
}

class Test
{
   static void Main() {
      Console.WriteLine(StringFromColor(Color.Red));
      Console.WriteLine(StringFromColor(Color.Green));
      Console.WriteLine(StringFromColor(Color.Blue));
   }

   static string StringFromColor(Color c) {
      switch (c) {
         case Color.Red:
            return String.Format("Red = {0}", (int) c);

         case Color.Green:
            return String.Format("Green = {0}", (int) c);

         case Color.Blue:
            return String.Format("Blue = {0}", (int) c);

         default:
            return "Invalid color";
      }
   }
}
```

输出枚举成员名称和它们的关联值。输出为：

```
Red = 0
Green = 10
Blue = 11
```

原因如下：

- 枚举成员 Red 被自动赋予零值（因为它不具有初始值设定项并且是第一个枚举成员）；
- 枚举成员 Green 被显式赋予值 10；
- 而枚举成员 Blue 被自动赋予比文本上位于它前面的成员大 1 的值。

枚举成员的关联值不能直接或间接地使用它自己的关联枚举成员的值。除了这个循环性限制外，枚举成员初始值设定项可以自由地引用其他的枚举成员初始值设定项，而不必考虑它们所在的文本位置的排列顺序。在枚举成员初始值设定项内，其他枚举成员的值始终被视为属于所对应的基础类型，因此在引用其他枚举成员时，没有必要使用强制转换。

下面的示例

```
enum Circular
{
   A = B,
   B
}
```

产生编译时错误，因为 A 和 B 的声明是循环的。A 显式依赖于 B，而 B 隐式依赖于 A。

枚举成员的命名方式和作用范围与类中的字段完全类似。枚举成员的范围是包含了它的枚举类型的体。在该范围内，枚举成员可以用它们的简单名称引用。在所有其他代码中，枚举成员的名称必须用它的枚

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

举类型的名称限定。枚举成员不具有任何声明可访问性，如果一个枚举类型是可访问的，则它所含的所有枚举成员都是可访问的。

14.4 System.Enum 类型

`System.Enum` 类型是所有枚举类型的抽象基类（它是一种与枚举类型的基础类型不同的独特类型），并且从 `System.Enum` 继承的成员在任何枚举类型中都可用。存在从任何枚举类型到 `System.Enum` 的装箱转换（第 4.3.1 节），并且存在从 `System.Enum` 到任何枚举类型的取消装箱转换（第 4.3.2 节）。

请注意，`System.Enum` 本身不是 *enum-type*。而是 *class-type*，所有 *enum-type* 都是从它派生的。类型 `System.Enum` 从类型 `System.ValueType`（第 4.1.1 节）继承，而后者又从类型 `object` 继承。在运行时，类型 `System.Enum` 的值可以是 `null` 或是对任何枚举类型的装箱值的引用。

14.5 枚举值和运算

每个枚举类型都定义了一个不同的类型；需要使用显式枚举转换（第 6.2.2 节）在枚举类型和整型之间或在两个枚举类型之间进行转换。一个枚举类型的值域不受它的枚举成员限制。具体而言，一个枚举的基础类型的任何一个值都可以被强制转换为该枚举类型，成为该枚举类型的一个独特的有效值。

枚举成员所属的类型就是声明了它们的那个枚举（出现在其他枚举成员初始值设定项中时除外：请参见第 14.3 节）。在枚举类型 `E` 中声明且关联值为 `v` 的枚举成员的值为 `(E)v`。

以下运算符可用于枚举类型的值：`==`、`!=`、`<`、`>`、`<=`、`>=`（第 7.10.5 节）、二元 `+`（第 7.8.4 节）、二元 `-`（第 7.8.5 节）`^`、`&`、`|`（第 7.11.2 节）、`~`（第 7.7.4 节）、`++` 和 `--`（第 7.6.9 节和第 7.7.5 节）。

每个枚举类型自动派生自类 `System.Enum` （后者又派生自 `System.ValueType` 和 `object`） \t "See also Enum" \t "See also Enum" \t "See also Enum" \t "See Enum" \t "See also Enum" \t "See Enum" 。因此，此类的派生方法和属性可以用在枚举类型的值上。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

15. 委托

委托是用来处理其他语言（如 C++、Pascal 和 Modula）需用函数指针来处理的情况的。不过与 C++ 函数指针不同，委托是完全面向对象的；另外，C++ 指针仅指向成员函数，而委托同时封装了对象实例和方法。

委托声明定义一个从 `System.Delegate` 类派生的类。委托实例封装了一个调用列表，该列表列出了一个或多个方法，每个方法称为一个可调用实体。对于实例方法，可调用实体由该方法和一个相关联的实例组成。对于静态方法，可调用实体仅由一个方法组成。用一个适当的参数集来调用一个委托实例，就是用此给定的参数集来调用该委托实例的每个可调用实体。

委托实例的一个有趣且有用的属性是：它不知道也不关心它所封装的方法所属的类；它所关心的仅限于这些方法必须与委托的类型兼容（第 15.1 节）。这使委托非常适合于“匿名”调用。

15.1 委托声明

delegate-declaration 是一种 *type-declaration*（第 9.6 节），它声明一个新的委托类型。

delegate-declaration:
attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type*
identifier variant-type-parameter-list$_{opt}$
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

delegate-modifiers:
delegate-modifier
delegate-modifiers delegate-modifier

delegate-modifier:
`new`
`public`
`protected`
`internal`
`private`

同一修饰符在一个委托声明中多次出现属于编译时错误。

`new` 修饰符仅允许在其他类型中声明的委托上使用，在这种情况下该修饰符表示所声明的委托会隐藏具有相同名称的继承成员，详见第 10.3.4 节。

`public`、`protected`、`internal` 和 `private` 修饰符将控制委托类型的可访问性。根据委托声明所在的上下文，可能不允许使用其中某些修饰符（第 3.5.1 节）。

上述的语法产生式中，*identifier* 用于指定委托的类型名称。

可选的 *formal-parameter-list* 用于指定委托的参数，而 *return-type* 则指定委托的返回类型。

可选的 *variant-type-parameter-list*（第 13.1.3 节）指定委托本身的类型形参。

委托类型的返回类型必须为 `void` 或输出安全（第 13.1.3.1 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

委托类型的所有形参类型都必须是输入安全的。此外，所有 out 或 ref 参数类型也必须是输出安全的。请注意，由于基础执行平台的限制，甚至 out 形参也必须是输入安全的。

C# 中的委托类型是名称等效的，而不是结构等效的。具体地说，对于两个委托类型，即使它们具有相同的参数列表和返回类型，仍被认为是不同的两个委托类型。但是，两个不同但结构上等效的委托类型的实例可能比较为相等（第 7.9.8 节）。

例如：

```
delegate int D1(int i, double d);
class A
{
   public static int M1(int a, double b) {...}
}
class B
{
   delegate int D2(int c, double d);
   public static int M1(int f, double g) {...}
   public static void M2(int k, double l) {...}
   public static int M3(int g) {...}
   public static void M4(int g) {...}
}
```

方法 A.M1 和 B.M1 与委托类型 D1 和 D2 都兼容，因为这两个方法的返回类型和参数列表相同；但是，这两个委托类型是两个不同的类型，因此这两个委托类型不可互换。方法 B.M2 和 B.M3、B.M4 与委托类型 D1 和 D2 不兼容，因为这两个方法的返回类型或参数列表不同。

与其他泛型类型声明一样，必须提供类型实参才能创建构造委托类型。构造委托类型的形参类型和返回类型是通过将委托声明中的每个类型形参替换为构造委托类型的对应类型实参来创建的。结果返回类型和形参类型用于确定哪些方法与构造委托类型兼容。例如：

```
delegate bool Predicate<T>(T value);
class X
{
   static bool F(int i) {...}
   static bool G(string s) {...}
}
```

方法 X.F 与委托类型 Predicate<int> 兼容，方法 X.G 与委托类型 Predicate<string> 兼容。

声明一个委托类型的唯一方法是通过 *delegate-declaration*。委托类型是从 System.Delegate 派生的类类型。委托类型隐含为 sealed，所以不允许从一个委托类型派生任何类型。也不允许从 System.Delegate 派生非委托类类型。请注意：System.Delegate 本身不是委托类型；它是从中派生所有委托类型的类类型。

C# 提供了专门的语法用于委托类型的实例化和调用。除实例化外，所有可以应用于类或类实例的操作也可以相应地应用于委托类或委托实例。具体而言，可以通过通常的成员访问语法访问 System.Delegate 类型的成员。

委托实例所封装的方法集合称为调用列表。从某个方法创建一个委托实例时（第 15.2 节），该委托实例将封装此方法，此时，它的调用列表只包含一个“入口点”。但是，当组合两个非空委托实例时，它们的调用列表将连接在一起（按照左操作数在前、右操作数在后的顺序）以组成一个新的调用列表，其中包含两个或更多个“入口点”。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

委托是使用二元 + （第 7.8.4 节）和 += 运算符（第 7.17.2 节）进行组合的。可以使用二元 - （第 7.8.5 节）和 -= 运算符（第 7.17.2 节）将一个委托从委托组合中移除。委托间还可以进行比较以确定它们是否相等（第 7.10.8 节）。

下面的示例演示多个委托的实例化及其相应的调用列表：

```
delegate void D(int x);

class C
{
   public static void M1(int i) {...}
   public static void M2(int i) {...}
}
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);     // M1
      D cd2 = new D(C.M2);     // M2
      D cd3 = cd1 + cd2;       // M1 + M2
      D cd4 = cd3 + cd1;       // M1 + M2 + M1
      D cd5 = cd4 + cd3;       // M1 + M2 + M1 + M1 + M2
   }
}
```

实例化 cd1 和 cd2 时，它们分别封装一个方法。实例化 cd3 时，它的调用列表有两个方法 M1 和 M2，而且顺序与此相同。cd4 的调用列表中依次包含 M1、M2 和 M1。最后，cd5 的调用列表中依次包含 M1、M2、M1、M1 和 M2。有关组合(以及移除)委托的更多示例，请参见第 15.4 节。

15.2 委托兼容性

方法或委托 M 可***兼容 (compatible)***委托类型 D，前提是以下所有条件都成立：

- D 和 M 具有相同数量的形参，并且 D 中的每个形参都具有与 M 中对应形参相同的 ref 或 out 修饰符。
- 对于每个值形参（没有 ref 或 out 修饰符的形参），存在从 D 中形参类型到 M 中对应形参类型的标识转换（第 6.1.1 节）或隐式引用转换（第 6.1.6 节）。
- 对于每个 ref 或 out 参数，D 中的参数类型与 M 中的参数类型相同。
- 存在从 M 的返回类型到 D 的返回类型的标识或隐式引用转换。

15.3 委托实例化

委托的实例通过 *delegate-creation-expression*（第 7.6.10.5 节）或到委托类型的转换进行创建。因此，新创建的委托实例将引用以下各项之一：

- *delegate-creation-expression* 中引用的静态方法，或者
- *delegate-creation-expression* 中引用的目标对象（此对象不能为 null）和实例方法，或者
- 另一个委托。

例如：

```
delegate void D(int x);
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class C
{
   public static void M1(int i) {...}
   public void M2(int i) {...}
}

class Test
{
   static void Main() {
      D cd1 = new D(C.M1);    // static method
      C t = new C();
      D cd2 = new D(t.M2);    // instance method
      D cd3 = new D(cd2);     // another delegate
   }
}
```

委托实例一旦被实例化，它将始终引用同一目标对象和方法。记住，当组合两个委托或者从一个委托移除另一个时，将产生一个新的委托，该委托具有它自己的调用列表；被组合或移除的委托的调用列表将保持不变。

15.4 委托调用

C# 为调用委托提供了专门的语法。当调用非空的、调用列表仅包含一个入口点的委托实例时，它调用调用列表中的方法，委托调用所使用的参数和返回的值均与该方法的对应项相同。(有关委托调用的详细信息，请参见第 7.6.5.3 节。)如果在对这样的委托进行调用期间发生异常，而且没有在被调用的方法内捕捉到该异常，则会在调用该委托的方法内继续搜索与该异常对应的 catch 子句，就像调用该委托的方法直接调用了该委托所引用的方法一样。

如果一个委托实例的调用列表包含多个入口点，那么调用这样的委托实例就是按顺序同步地调用调用列表中所列的各个方法。以这种方式调用的每个方法都使用相同的参数集，即提供给委托实例的参数集。如果这样的委托调用包含引用参数（第 10.6.1.2 节），那么每个方法调用都将使用对同一变量的引用；这样，若调用列表中有某个方法对该变量进行了更改，则调用列表中排在该方法之后的所有方法都会见到此变更。如果委托调用包含输出参数或一个返回值，则它们的最终值就是调用列表中最后一个方法调用所产生的结果。

如果在处理此类委托的调用期间发生异常，而且没有在正被调用的方法内捕捉到该异常，则会在调用该委托的方法内继续搜索与该异常对应的 catch 子句，此时，调用列表中排在后面的任何方法将不会被调用。

试图调用其值为 null 的委托实例将导致 `System.NullReferenceException` 类型的异常。

下面的示例演示如何实例化、组合、移除和调用委托：

```
using System;

delegate void D(int x);

class C
{
   public static void M1(int i) {
      Console.WriteLine("C.M1: " + i);
   }

   public static void M2(int i) {
      Console.WriteLine("C.M2: " + i);
   }

   public void M3(int i) {
      Console.WriteLine("C.M3: " + i);
   }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);
      cd1(-1);          // call M1

      D cd2 = new D(C.M2);
      cd2(-2);          // call M2

      D cd3 = cd1 + cd2;
      cd3(10);          // call M1 then M2

      cd3 += cd1;
      cd3(20);          // call M1, M2, then M1

      C c = new C();
      D cd4 = new D(c.M3);
      cd3 += cd4;
      cd3(30);          // call M1, M2, M1, then M3

      cd3 -= cd1;       // remove last M1
      cd3(40);          // call M1, M2, then M3

      cd3 -= cd4;
      cd3(50);          // call M1 then M2

      cd3 -= cd2;
      cd3(60);          // call M1

      cd3 -= cd2;       // impossible removal is benign
      cd3(60);          // call M1

      cd3 -= cd1;       // invocation list is empty so cd3 is null

      //    cd3(70);    // System.NullReferenceException thrown

      cd3 -= cd1;       // impossible removal is benign
   }
}
```

如语句 `cd3 += cd1;` 中所演示，委托可以多次出现在一个调用列表中。这种情况下，它每出现一次，就会被调用一次。在这样的调用列表中，当移除委托时，实际上移除的是调用列表中最后出现的那个委托实例。

就在执行最后一条语句 `cd3 -= cd1;` 之前，委托 `cd3` 引用了一个空的调用列表。试图从空的列表中移除委托（或者从非空列表中移除表中没有的委托）不算是错误。

产生的输出为：

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

16. 异常

C# 中的异常用于处理系统级和应用程序级的错误状态，它是一种结构化的、统一的和类型安全的处理机制。C# 中的异常机制非常类似于 C++ 的异常机制，但是有一些重要的区别：

- 在 C# 中，所有的异常必须由从 `System.Exception` 派生的类类型的实例来表示。在 C++ 中，可以使用任何类型的任何值表示异常。
- 在 C# 中，利用 finally 块（第 8.10 节）可编写在正常执行和异常情况下都将执行的终止代码。在 C++ 中，很难在不重复代码的情况下编写这样的代码。
- 在 C# 中，系统级的异常如溢出、被零除和 null 等都对应地定义了与其匹配的异常类，并且与应用程序级的错误状态处于同等地位。

16.1 导致异常的原因

可以以两种不同的方式引发异常。

- `throw` 语句（第 8.9.5 节）用于立即无条件地引发异常。控制永远不会到达紧跟在 `throw` 后面的语句。
- 在执行 C# 语句和表达式的过程中，有时会出现一些例外情况，使某些操作无法正常完成，此时就会引发一个异常。例如，在整数除法运算（第 7.8.2 节）中，如果分母为零，则会引发 `System.DivideByZeroException`。有关可能以此方式引发的各种异常的列表，请参见第 16.4 节。

16.2 System.Exception 类

`System.Exception` 类是所有异常的基类型。此类具有一些所有异常共享的值得注意的属性：

- `Message` 是 `string` 类型的一个只读属性，它包含关于所发生异常的原因的描述（易于人工阅读）。
- `InnerException` 是 `Exception` 类型的一个只读属性。如果它的值不是 null，则它所引用的是导致了当前异常的那个异常，即表示当前异常是在处理那个 `InnerException` 的 catch 块中被引发的。否则，它的值为 null，则表示该异常不是由另一个异常引发的。以这种方式链接在一起的异常对象的数目可以是任意的。

这些属性的值可以在调用 `System.Exception` 的实例构造函数时指定。

16.3 异常的处理方式

异常是由 `try` 语句（第 8.10 节）处理的。

发生异常时，系统将搜索可以处理该异常的最近的 `catch` 子句（根据该异常的运行时类型来确定）。首先，搜索当前的方法以查找一个词法上包含着它的 `try` 语句，并按顺序考察与该 try 语句相关联的各个 catch 子句。如果上述操作失败，则在调用了当前方法的方法中，搜索在词法上包含着当前方法调用代码位置的 `try` 语句。此搜索将一直进行下去，直到找到可以处理当前异常的 `catch` 子句（该子句指定一个异常类，它与当前引发该异常的运行时类型属于同一个类或是该运行时类型所属类的一个基类）。注意，没有指定异常类的 `catch` 子句可以处理任何异常。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

找到匹配的 catch 子句后，系统将把控制转移到该 catch 子句的第一条语句。在 catch 子句的执行开始前，系统将首先按顺序执行嵌套在捕捉到该异常的 try 语句里面的所有 try 语句所对应的全部 `finally` 子句。

如果没有找到匹配的 catch 子句，则发生下列两种情况之一：

- 如果对匹配的 catch 子句的搜索到达一个静态构造函数（第 10.12 节）或静态字段初始值设定项，则在导致调用该静态构造函数的代码位置引发 `System.TypeInitializationException`。该 `System.TypeInitializationException` 的内部异常将包含最初引发的异常。
- 如果对匹配的 catch 子句的搜索到达最初启动当前线程的代码处，则该线程的执行就会终止。此类终止会产生什么影响，应由实现来定义。

特别值得注意的是在析构函数执行过程中发生的异常。如果在析构函数执行过程中发生异常且该异常未被捕获，则将终止该析构函数的执行，并调用它的基类的析构函数（如果有）。如果没有基类（如 `object` 类型中的情况），或者如果没有基类析构函数，则该异常将被忽略。

16.4 公共异常类

下列异常由某些 C# 操作引发。

`System.ArithmeticException`	在算术运算期间发生的异常（如 `System.DivideByZeroException` 和 `System.OverflowException`）的基类。
`System.ArrayTypeMismatchException`	当存储一个数组时，如果由于被存储的元素的实际类型与数组的实际类型不兼容而导致存储失败，就会引发此异常。
`System.DivideByZeroException`	在试图用零除整数值时引发。
`System.IndexOutOfRangeException`	在试图使用小于零或超出数组界限的下标索引数组时引发。
`System.InvalidCastException`	当从基类型或接口到派生类型的显式转换在运行时失败时，就会引发此异常。
`System.NullReferenceException`	在需要使用引用对象的场合，如果使用 `null` 引用，就会引发此异常。
`System.OutOfMemoryException`	在分配内存（通过 `new`）的尝试失败时引发。
`System.OverflowException`	在 `checked` 上下文中的算术运算溢出时引发。
`System.StackOverflowException`	当执行堆栈由于保存了太多挂起的方法调用而耗尽时，就会引发此异常；这通常表明存在非常深或无限的递归。
`System.TypeInitializationException`	在静态构造函数引发异常并且没有可以捕捉到它的 `catch` 子句时引发。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

17. 特性

C# 语言的一个重要特征是使程序员能够为程序中定义的实体指定声明性信息。例如，类中方法的可访问性是通过使用 *method-modifiers*（public、protected、internal 和 private）加以修饰来指定的。

C# 使程序员可以创造新的声明性信息的种类，称为特性(attributes)。然后，程序员可以将这种特性附加到各种程序实体，而且在运行时环境中还可以检索这些特性信息。例如，一个框架可以定义一个名为 HelpAttribute 的特性，该特性可以放在某些程序元素（如类和方法）上，以提供从这些程序元素到其文档说明的映射。

特性是通过特性类（第 17.1 节）的声明定义的，该声明可以具有定位和命名参数（第 17.1.2 节）。特性是使用特性说明（第 17.2 节）附加到 C# 程序中的实体上的，而且可以在运行时作为特性实例（第 17.3 节）来检索。

17.1 特性类

从抽象类 System.Attribute 派生的类（不论是直接的还是间接的）都称为特性类 (attribute class)。一个关于特性类的声明定义一种新特性 (attribute)，它可以放置在其他声明上。按照约定，特性类的名称均带有 Attribute 后缀。使用特性时可以包含或省略此后缀。

17.1.1 特性用法

特性 AttributeUsage（第 17.4.1 节）用于描述使用特性类的方式。

AttributeUsage 具有一个定位参数（第 17.1.2 节），该参数使特性类能够指定自己可以用在哪种声明上。下面的示例

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
   ...
}
```

定义了一个名为 SimpleAttribute 的特性类，此特性类只能放在 *class-declaration* 和 *interface-declaration* 上。下面的示例

```
[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

演示了 Simple 特性的几种用法。虽然此特性是用名称 SimpleAttribute 定义的，但在使用时可以省略 Attribute 后缀，从而得到简称 Simple。因此，上例在语义上等效于：

```
[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

AttributeUsage 还具有一个名为 AllowMultiple 的命名参数（第 17.1.2 节），此参数用于说明对于某个给定实体，是否可以多次指定该特性。如果特性类的 AllowMultiple 为 true，则此特性类是多次性特性类 (multi-use attribute class)，可以在一个实体上多次被指定。如果特性类的 AllowMultiple 为 false 或未指定，则此特性类是一次性特性类 (single-use attribute class)，在一个实体上最多只能指定一次。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

下面的示例

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
   private string name;

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

   public string Name {
      get { return name; }
   }
}
```

定义一个多次使用的特性类，名为 AuthorAttribute。下面的示例

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
   ...
}
```

演示了一个两次使用 Author 特性的类声明。

AttributeUsage 具有另一个名为 Inherited 的命名参数，此参数指示在基类上指定该特性时，该特性是否也会被从此基类派生的类所继承。如果特性类的 Inherited 为 true，则该特性会被继承。如果特性类的 Inherited 为 false，则该特性不会被继承。如果该值未指定，则其默认值为 true。

没有附加 AttributeUsage 特性的特性类 X，例如

```
using System;

class X: Attribute {...}
```

等效于下面的内容：

```
using System;

[AttributeUsage(
   AttributeTargets.All,
   AllowMultiple = false,
   Inherited = true)
]
class X: Attribute {...}
```

17.1.2 定位和命名参数

特性类可以具有定位参数 (positional parameter) 和命名参数 (named parameter)。特性类的每个公共实例构造函数为该特性类定义一个有效的定位参数序列。特性类的每个非静态公共读写字段和属性为该特性类定义一个命名参数。

下面的示例

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
   public HelpAttribute(string url) {     // Positional parameter
      ...
   }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
      public string Topic {                    // Named parameter
         get {...}
         set {...}
      }
      public string Url {
         get {...}
      }
   }
```

定义了一个名为 HelpAttribute 的特性类，它具有一个定位参数 (url) 和一个命名参数 (Topic)。虽然 Url 属性是非静态的和公共的，但由于它不是读写的，因此它并不定义命名参数。

此特性类可以如下方式使用：

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
   ...
}
[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
   ...
}
```

17.1.3 特性参数类型

特性类的定位参数和命名参数的类型仅限于特性参数类型 (attribute parameter type)，它们是：

- 以下类型之一：bool、byte、char、double、float、int、long、sbyte、short、string、uint、ulong、ushort。
- object 类型。
- System.Type 类型。
- 枚举类型，前提是该枚举类型具有 public 可访问性，而且所有嵌套着它的类型（如果有）也必须具有 public 可访问性（第 17.2 节）。
- 以上类型的一维数组。

不是这些类型的构造函数实参或公共字段不能用作特性规范中的位置或命名形参。

17.2 特性说明

特性规范 (Attribute specification) 就是将以前定义的特性应用到某个声明上。特性本身是一段附加说明性信息，可以把它指定给某个声明。可以在全局范围指定特性（即，在包含程序集或模块上指定特性），也可以为下列各项指定特性：*type-declaration*（第 9.6 节）、*class-member-declaration*（第 10.1.5 节）、*interface-member-declaration*（第 13.2 节）、*struct-member-declaration*（第 11.2 节）、*enum-member-declaration*（第 14.3 节）、*accessor-declaration*（第 10.7.2 节）、*event-accessor-declarations*（第 10.8.1 节）和 *formal-parameter-lists*（第 10.6.1 节）。

特性是在特性节 (attribute section) 中指定的。特性节由一对方括号组成，此方括号括着一个用逗号分隔的、含有一个或多个特性的列表。在这类列表中以何种顺序指定特性，以及附加到同一程序实体的特性节以何种顺序排列等细节并不重要。例如，特性说明 [A][B]、[B][A]、[A, B] 和 [B, A] 是等效的。

global-attributes:
    global-attribute-sections

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

global-attribute-sections:
    global-attribute-section
    global-attribute-sections *global-attribute-section*

global-attribute-section:
    `[` *global-attribute-target-specifier* *attribute-list* `]`
    `[` *global-attribute-target-specifier* *attribute-list* `,` `]`

global-attribute-target-specifier:
    global-attribute-target `:`

global-attribute-target:
    `assembly`
    `module`

attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections *attribute-section*

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `,` `]`

attribute-target-specifier:
    attribute-target `:`

attribute-target:
    `field`
    `event`
    `method`
    `param`
    `property`
    `return`
    `type`

attribute-list:
    attribute
    attribute-list `,` *attribute*

attribute:
    attribute-name *attribute-arguments*$_{opt}$

attribute-name:
    type-name

attribute-arguments:
    `(` *positional-argument-list*$_{opt}$ `)`
    `(` *positional-argument-list* `,` *named-argument-list* `)`
    `(` *named-argument-list* `)`

positional-argument-list:
    positional-argument
    positional-argument-list `,` *positional-argument*

positional-argument:
    argument-name$_{opt}$ *attribute-argument-expression*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

named-argument-list:
named-argument
named-argument-list , *named-argument*

named-argument:
identifier = *attribute-argument-expression*

attribute-argument-expression:
expression

如上所述，特性由一个 *attribute-name* 和一个可选的定位和命名参数列表组成。定位参数（如果有）列在命名参数前面。定位参数包含一个 *attribute-argument-expression*；命名参数包含一个名称，名称后接一个等号和一个 *attribute-argument-expression*，这两种参数都受简单赋值规则约束。命名参数的排列顺序无关紧要。

attribute-name 用于标识特性类。如果 *attribute-name* 的形式等同于一个 *type-name*，则此名称必须引用一个特性类。其他情况下，将发生编译时错误。下面的示例

```
class Class1 {}
[Class1] class Class2 {}   // Error
```

产生编译时错误，因为它试图将 `Class1` 用作特性类，而 `Class1` 并不是一个特性类。

某些上下文允许将一个特性指定给多个目标。程序中可以利用 *attribute-target-specifier* 来显式地指定目标。特性放置在全局级别中时，则需要 *global-attribute-target-specifier*。对于所有其他位置上的特性，则采用系统提供的合理的默认值，但是在某些目标不明确的情况下可以使用 *attribute-target-specifier* 来确认或重写默认值，也可以在目标明确的情况下使用特性目标说明符来确认默认值。因此，除在全局级别之外，通常可以省略 *attribute-target-specifier*。对于可能造成不明确性的上下文，按下述规则处理：

- 在全局范围指定的特性可以应用于目标程序集或目标模块。系统没有为此上下文提供默认形式，所以在此上下文中始终需要一个 *attribute-target-specifier*。如果存在 `assembly` *attribute-target-specifier*，则表明此特性适用于指定的目标程序集；如果存在 `module` *attribute-target-specifier*，则表明此特性适用于指定的目标模块。
- 在委托声明上指定的特性，或者适用于所声明的委托，或者适用于它的返回值。如果不存在 *attribute-target-specifier*，则此特性适用于该委托。如果存在 `type` *attribute-target-specifier*，则表明此特性适用于该委托；如果存在 `return` *attribute-target-specifier*，则表明此特性适用于返回值。
- 在方法声明上指定的特性，或者适用于所声明的方法，或者适用于它的返回值。如果不存在 *attribute-target-specifier*，则此特性适用于方法。如果存在 `method` *attribute-target-specifier*，则表明此特性适用于方法；如果存在 `return` *attribute-target-specifier*，则表明此特性适用于返回值。
- 在运算符声明上指定的特性，或者适用于所声明的运算符，或者适用于它的返回值。如果不存在 *attribute-target-specifier*，则此特性适用于该运算符。如果存在 `method` *attribute-target-specifier*，则表明此特性适用于该运算符；如果存在 `return` *attribute-target-specifier*，则表明此特性适用于返回值。
- 对于在省略了事件访问器的事件声明上指定的特性，它的目标对象有三种可能的选择：所声明的事件；与该事件关联的字段（如果该事件是非抽象事件）；与该事件关联的 add 和 remove 方法。如果不存在 *attribute-target-specifier*，则此特性适用于该事件。如果存在 `event` *attribute-target-specifier*，则表明此特性适用于该事件；如果存在 `field` *attribute-target-specifier*，则表明此特性适用于该字段；而如果存在 `method` *attribute-target-specifier*，则表明此特性适用于这些方法。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 在属性或索引器声明中的 get 访问器声明上指定的特性，或者适用于该访问器关联的方法，或者适用于它的返回值。如果不存在 *attribute-target-specifier*，则此特性适用于方法。如果存在 `method` *attribute-target-specifier*，则表明此特性适用于方法；如果存在 `return` *attribute-target-specifier*，则表明此特性适用于返回值。
- 在属性或索引器声明中的 set 访问器上指定的特性，或者可适用于该访问器关联的方法，或者适用于它的独立的隐式参数。如果不存在 *attribute-target-specifier*，则此特性适用于方法。如果存在 `method` *attribute-target-specifier*，则表明此特性适用于该方法；如果存在 `param` *attribute-target-specifier*，则表明此特性适用于该参数；而如果存在 `return` *attribute-target-specifier*，则表明此特性适用于该返回值。
- 在事件声明的添加或移除访问器声明上指定的特性，或者适用于该访问器关联的方法，或者适用于它的独立参数。如果不存在 *attribute-target-specifier*，则此特性适用于方法。如果存在 `method` *attribute-target-specifier*，则表明此特性适用于该方法；如果存在 `param` *attribute-target-specifier*，则表明此特性适用于该参数；而如果存在 `return` *attribute-target-specifier*，则表明此特性适用于该返回值。

在其他上下文中，允许包含一个 *attribute-target-specifier*，但这样做是没有必要的。例如，类声明既可以包括也可以省略说明符 `type`：

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

如果指定了无效的 *attribute-target-specifier*，则会发生错误。例如，不能将说明符 `param` 用在类声明中：

```
[param: Author("Brian Kernighan")]     // Error
class Class1 {}
```

按照约定，特性类的名称均带有 `Attribute` 后缀。*type-name* 形式的 *attribute-name* 既可以包含也可以省略此后缀。如果发现特性类中同时出现带和不带此后缀的名称，则引用时就可能出现多义性，从而导致运行时错误。如果在拼写 *attribute-name* 时，明确说明其最右边的 *identifier* 为逐字标识符（第 2.4.2 节），则它仅匹配没有后缀的特性，从而能够解决这类多义性。下面的示例

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                    // Error: ambiguity
class Class1 {}

[XAttribute]           // Refers to XAttribute
class Class2 {}

[@X]                   // Refers to X
class Class3 {}

[@XAttribute]          // Refers to XAttribute
class Class4 {}
```

演示两个分别名为 `X` 和 `XAttribute` 的特性类。特性 `[X]` 具有多义性，因为该特性既可引用 `X` 也可引用 `XAttribute`。使用逐字标识符能够在这种极少见的情况下表明确切的意图。特性 `[XAttribute]` 是

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

明确的（尽管当存在名为XAttributeAttribute的特性类时，该特性将是不明确的！）。如果移除了类X的声明，那么上述两个特性都将引用名为XAttribute的特性类，如下所示：

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                   // Refers to XAttribute
class Class1 {}

[XAttribute]          // Refers to XAttribute
class Class2 {}

[@X]                  // Error: no attribute named "X"
class Class3 {}
```

在同一个实体中多次使用单次使用的特性类属于编译时错误。下面的示例

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
   string value;

   public HelpStringAttribute(string value) {
      this.value = value;
   }

   public string Value {
      get {...}
   }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

产生编译时错误，因为它尝试在Class1的声明中多次使用单次使用的特性类HelpString。

如果表达式E满足下列所有条件，则该表达式为 *attribute-argument-expression*：

- E的类型是特性参数类型（第17.1.3节）。
- 在编译时，E的值可以解析为下列之一：
  - 常量值。
  - 一个System.Type对象。
  - *attribute-argument-expression*的一维数组。

例如：

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
   public int P1 {
      get {...}
      set {...}
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   public Type P2 {
      get {...}
      set {...}
   }
   public object P3 {
      get {...}
      set {...}
   }
}
[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

用作特性实参表达式的 *typeof-expression*（第 7.6.11 节）可引用非泛型类型、封闭构造类型或未绑定的泛型类型，但是不能引用开放类型。这用于确保在编译时可以解析表达式。

```
class A: Attribute
{
   public A(Type t) {...}
}

class G<T>
{
   [A(typeof(T))] T t;              // Error, open type in attribute
}

class X
{
   [A(typeof(List<int>))] int x;    // Ok, closed constructed type
   [A(typeof(List<>))] int y;       // Ok, unbound generic type
}
```

17.3 特性实例

特性实例 (attribute instance) 是一个实例，用于在运行时表示特性。特性是用特性类、定位参数和命名参数定义的。特性实例是一个特性类的实例，它是用定位参数和命名参数初始化后得到的。

特性实例的检索涉及编译时和运行时处理，详见后面几节中的介绍。

17.3.1 特性的编译

对于一个具有特性类 `T`、*positional-argument-list* `P` 和 *named-argument-list* `N` 的 *attribute* 的编译过程由下列步骤组成：

- 遵循形式为 `new T(P)` 的 *object-creation-expression* 的编译规则所规定的步骤进行编译时处理。这些步骤或者导致编译时错误，或者确定 `T` 上的可以在运行时调用的实例构造函数 `C`。
- 如果 `C` 不具有公共可访问性，则发生编译时错误。
- 对于 `N` 中的每个 *named-argument* `Arg`：
  - 将 `Name` 设为 *identifier*（属于 *named-argument* `Arg`）。
  - `Name` 必须标识 `T` 中的一个非静态读写字段或属性。如果 `T` 没有这样的字段或属性，则发生编译时错误。
- 保留以下信息用于特性的运行时实例化：特性类 `T`、`T` 上的实例构造函数 `C`、*positional-argument-list* `P` 和 *named-argument-list* `N`。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

17.3.2 特性实例的运行时检索

对一个 *attribute* 进行编译后，会产生一个特性类 T、一个 T 上的实例构造函数 C、一个 *positional-argument-list* P 和一个 *named-argument-list* N。给定了上述信息后，就可以在运行时使用下列步骤进行检索来生成一个特性实例：

- 遵循执行 new T(P) 形式的 *object-creation-expression*（使用在编译时确定的实例构造函数 C）的运行时处理步骤。这些步骤或者导致异常，或者产生 T 的一个实例 O。
- 对于 N 中的每个 *named-argument* Arg，按以下顺序进行处理：
  - 将 Name 设为 *identifier*（属于 *named-argument* Arg）。如果 Name 未在 O 上标识一个非静态公共读写字段或属性，则将引发异常。
  - 将 Value 设为 *attribute-argument-expression*（属于 Arg）的计算结果。
  - 如果 Name 标识 O 上的一个字段，则将此字段设置为 Value。
  - 否则，Name 将标识 O 上的一个属性。将此属性设置为 Value。
  - 结果为 O，它是已经用 *positional-argument-list* P 和 *named-argument-list* N 初始化了的特性类 T 的一个实例。

17.4 保留特性

少数特性以某种方式影响语言。这些特性包括：

- System.AttributeUsageAttribute（第 17.4.1 节），它用于描述可以以哪些方式使用特性类。
- System.Diagnostics.ConditionalAttribute（第 17.4.2 节），用于定义条件方法。
- System.ObsoleteAttribute（第 17.4.3 节），用于将某个成员标记为已过时。
- System.Runtime.CompilerServices.CallerLineNumberAttribute、System.Runtime.CompilerServices.CallerFilePathAttribute 和 System.Runtime.CompilerServices.CallerMemberNameAttribute（第 17.4.4 节），用于提供有关可选参数的调用上下文的信息。

17.4.1 AttributeUsage 特性

AttributeUsage 特性用于描述使用特性类的方式。

使用 AttributeUsage 特性修饰的类必须直接或间接从 System.Attribute 派生。其他情况下，将发生编译时错误。

```
namespace System
{
   [AttributeUsage(AttributeTargets.Class)]
   public class AttributeUsageAttribute: Attribute
   {
      public AttributeUsageAttribute(AttributeTargets validOn) {...}
      public virtual bool AllowMultiple { get {...} set {...} }
      public virtual bool Inherited { get {...} set {...} }
      public virtual AttributeTargets ValidOn { get {...} }
   }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
      public enum AttributeTargets
      {
         Assembly    = 0x0001,
         Module      = 0x0002,
         Class       = 0x0004,
         Struct      = 0x0008,
         Enum        = 0x0010,
         Constructor = 0x0020,
         Method      = 0x0040,
         Property    = 0x0080,
         Field       = 0x0100,
         Event       = 0x0200,
         Interface   = 0x0400,
         Parameter   = 0x0800,
         Delegate    = 0x1000,
         ReturnValue = 0x2000,

         All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
      }
   }
```

17.4.2 Conditional 特性

通过特性 `Conditional\b` 可实现*条件方法*和*条件特性类*的定义。

```
   namespace System.Diagnostics
   {
      [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
                      AllowMultiple = true)]
      public class ConditionalAttribute: Attribute
      {
         public ConditionalAttribute(string conditionString) {...}

         public string ConditionString { get {...} }
      }
   }
```

17.4.2.1 条件方法

用 `Conditional` 特性修饰的方法是条件方法。`Conditional` 特性通过测试条件编译符号来指示条件。当运行到一个条件方法调用时，是否执行该调用，要根据出现该调用时是否已定义了此符号来确定。如果定义了此符号，则执行该调用；否则省略该调用（包括对调用的接收器和形参的计算）。

条件方法要受到以下限制：

- 条件方法必须是 *class-declaration* 或 *struct-declaration* 中的方法。如果在接口声明中的方法上指定 `Conditional` 特性，将出现编译时错误。
- 条件方法必须具有 `void` 返回类型。
- 不能用 `override` 修饰符标记条件方法。但是，可以用 `virtual` 修饰符标记条件方法。此类方法的重写方法隐含为有条件的方法，而且不能用 `Conditional` 特性显式标记。
- 条件方法不能是接口方法的实现。其他情况下，将发生编译时错误。

此外，如果条件方法用在 *delegate-creation-expression* 中，也会发生编译时错误。下面的示例

```
   #define DEBUG

   using System;
   using System.Diagnostics;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Class1
{
   [Conditional("DEBUG")]
   public static void M() {
      Console.WriteLine("Executed Class1.M");
   }
}

class Class2
{
   public static void Test() {
      Class1.M();
   }
}
```

将 Class1.M 声明为条件方法。Class2 的 Test 方法将调用此方法。由于定义了条件编译符号 DEBUG，因此如果调用 Class2.Test，则它会调用 M。如果尚未定义符号 DEBUG，则 Class2.Test 将不会调用 Class1.M。

一定要注意包含或排除对条件方法的调用是由该调用所在处的条件编译符号控制的。在下面的示例中

文件 class1.cs：

```
using System.Diagnostics;

class Class1
{
   [Conditional("DEBUG")]
   public static void F() {
      Console.WriteLine("Executed Class1.F");
   }
}
```

文件 class2.cs：

```
#define DEBUG

class Class2
{
   public static void G() {
      Class1.F();          // F is called
   }
}
```

文件 class3.cs：

```
#undef DEBUG

class Class3
{
   public static void H() {
      Class1.F();          // F is not called
   }
}
```

类 Class2 和 Class3 均包含对条件方法 Class1.F 的调用，根据是否定义了 DEBUG，此调用是有条件的。由于在 Class2 的上下文中定义了此符号而在 Class3 的上下文中没有定义，因此在 Class2 中包含了对 F 的调用，而在 Class3 中省略了对 F 的调用。

在继承链中使用条件方法可能引起混乱。通过 base.M 形式的 base 对条件方法进行的调用受正常条件方法调用规则的限制。在下面的示例中

文件 class1.cs：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
using System;
using System.Diagnostics;

class Class1
{
   [Conditional("DEBUG")]
   public virtual void M() {
      Console.WriteLine("Class1.M executed");
   }
}
```

文件 class2.cs：

```
using System;

class Class2: Class1
{
   public override void M() {
      Console.WriteLine("Class2.M executed");
      base.M();                  // base.M is not called!
   }
}
```

文件 class3.cs：

```
#define DEBUG

using System;

class Class3
{
   public static void Test() {
      Class2 c = new Class2();
      c.M();                     // M is called
   }
}
```

Class2 包括一个对在其基类中定义的 M 的调用。此调用被省略，因为基方法是条件性的，依赖于符号 DEBUG 是否存在，而该符号在此处没有定义。因此，该方法仅向控制台写入“Class2.M executed”。审慎使用 *pp-declaration* 可以消除这类问题。

17.4.2.2 条件特性类

使用一个或多个 Conditional 特性修饰的特性类（第 17.1 节）就是条件特性类(conditional attribute class)。条件特性类因此与在其 Conditional 特性中声明的条件编译符号关联。本示例：

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

将 TestAttribute 声明为与条件编译符号 ALPHA 和 BETA 关联的条件特性类。

如果在特性说明处定义了一个或多个关联的条件编译符号，则条件特性的特性说明（第 17.2 节）也会包括在内；否则会忽略特性说明。

注意包含或排除条件特性类的特性规范是由该指定所在位置的条件编译符号控制的，这一点很重要。在下面的示例中

文件 test.cs：

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
public class TestAttribute : Attribute {}
```

文件 class1.cs:

```
#define DEBUG
[Test]            // TestAttribute is specified
class Class1 {}
```

文件 class2.cs:

```
#undef DEBUG
[Test]            // TestAttribute is not specified
class Class2 {}
```

类 Class1 和 Class2 均使用特性 Test 进行修饰，根据是否定义了 DEBUG，此特性是有条件的。由于此符号是在 Class1 而不是 Class2 的上下文中定义的，因此包含 Class1 中 Test 特性的规范，而省略 Class2 中 Test 特性的规范。

17.4.3 Obsolete 特性

Obsolete 特性用于标记不应该再使用的类型和类型成员。

```
namespace System
{
   [AttributeUsage(
      AttributeTargets.Class |
      AttributeTargets.Struct |
      AttributeTargets.Enum |
      AttributeTargets.Interface |
      AttributeTargets.Delegate |
      AttributeTargets.Method |
      AttributeTargets.Constructor |
      AttributeTargets.Property |
      AttributeTargets.Field |
      AttributeTargets.Event,
      Inherited = false)
   ]
   public class ObsoleteAttribute: Attribute
   {
      public ObsoleteAttribute() {...}
      public ObsoleteAttribute(string message) {...}
      public ObsoleteAttribute(string message, bool error) {...}
      public string Message { get {...} }
      public bool IsError { get {...} }
   }
}
```

如果程序使用了由 Obsolete 特性修饰的类型或成员，则编译器将发出警告或错误信息。具体而言，如果没有提供错误参数，或者如果提供了错误参数但该错误参数的值为 false，则编译器将发出警告。如果指定了错误参数并且该错误参数的值为 true，则会引发一个编译时错误。

在下面的示例中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
   public void F() {}
}

class B
{
   public void F() {}
}

class Test
{
   static void Main() {
      A a = new A();       // Warning
      a.F();
   }
}
```

类 A 是用 Obsolete 特性修饰的。Main 的代码中，每次使用 A 时均会导致一个包含指定消息“This class is obsolete; use class B instead”(此类已过时；请改用类 B)的警告。

17.4.4 调用方信息特性

出于日志记录和报告等目的，使用函数成员获取某些有关调用代码的编译时信息有时是很有用的。调用方信息特性提供了透明地传递此类信息的方法。

当使用调用方信息特性之一批注了可选参数时，在调用中省略相应参数不一定导致替换为默认参数值。相反，如果有关调用上下文的指定信息可用，则该信息将作为参数值进行传递。

例如：

```
using System.Runtime.CompilerServices

…

public void Log(
   [CallerLineNumber] int line = -1,
   [CallerFilePath]   string path = null,
   [CallerMemberName] string name = null
)
{
   Console.WriteLine((line < 0) ? "No line" : "Line "+ line);
   Console.WriteLine((path == null) ? "No file path" : path);
   Console.WriteLine((name == null) ? "No member name" : name);
}
```

调用不带参数的 Log() 时将输出调用的行号和文件路径，以及在其中发生调用的成员的名称。

调用方信息特性可以出现在任何位置的可选参数上，包括出现在委托声明中。但是，特定调用方信息特性对于它们可以规定其特性的参数类型有限制，因此将始终存在从替换值到参数类型的隐式转换。

在分部方法声明的定义部分和实现部分的参数上出现相同的调用方信息特性会导致出错。仅应用定义部分的调用方信息特性，而忽略仅出现在实现部分的调用方信息特性。

调用方信息不会影响重载决策。当仍从调用方的源代码中省略特性化的可选参数时，重载决策将以忽略其他省略的可选参数（第 7.5.3 节）的相同方式忽略这些参数。

只有在源代码中显式调用函数时，才会替换为调用方信息。隐式父构造函数调用等隐式调用没有源位置，因此不会替换为调用方信息。此外，动态绑定的调用将不会替换为调用方信息。在这种情况下省略调用方信息特性化参数时，将改为使用指定的参数默认值。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

查询表达式是一个例外。这些被视为是句法扩展，并且如果用户将调用扩展为使用调用方信息特性而省略可选参数，则将替换为调用方信息。所用位置是从中生成调用的查询子句的位置。

如果在给定的参数中指定了多个调用方信息特性，则它们的优先顺序如下：CallerLineNumber、CallerFilePath、CallerMemberName。

17.4.4.1 CallerLineNumber 特性

当存在从常量值 int.MaxValue 到参数类型的标准隐式转换（第 6.3.1 节）时，允许对可选参数使用 System.Runtime.CompilerServices.CallerLineNumberAttribute。这将确保可以传递任何小于等于该值的非负行号，而不会出错。

如果从源代码中的某个位置调用函数时使用 CallerLineNumberAttribute 而省略了可选形参，则表示该位置的行号的数字文本（而不是默认参数值）将用作此调用的实参。

如果该调用跨多行，则选择哪一行将取决于具体实现。

请注意，行号可能会受 #line 指令（第 2.5.7 节）影响。

17.4.4.2 CallerFilePath 特性

当存在从 string 到参数类型的标准隐式转换（第 6.3.1 节）时，允许对可选参数使用 System.Runtime.CompilerServices.CallerFilePathAttribute。

如果从源代码中的某个位置调用函数时使用 CallerFilePathAttribute 而省略了可选形参，则表示该位置的文件路径的字符串文本（而不是默认参数值）将用作此调用的实参。

文件路径的格式取决于具体实现。

请注意，文件路径可能会受 #line 指令（第 2.5.7 节）影响。

17.4.4.3 CallerMemberName 特性

当存在从 string 到参数类型的标准隐式转换（第 6.3.1 节）时，允许对可选参数使用 System.Runtime.CompilerServices.CallerMemberNameAttribute。

如果从源代码中的函数成员体内或应用于函数成员本身、函数成员返回类型、形参或类型形参的某个特性中的某个位置调用函数时使用 CallerMemberNameAttribute 而省略可选形参，则表示该成员名称的字符串文本（而不是默认参数值）将用作此调用的实参。

对于发生在泛型方法内的调用，仅使用方法名称本身，而不带类型形参列表。

对于发生在显式接口成员实现中的调用，仅使用方法名称本身，而不带前面的接口限定。

对于发生在属性访问器或事件访问器中的调用，所使用的成员名称是该属性或事件本身的名称。

对于发生在索引器访问器中的调用，所使用的成员名称是由索引器成员（如果存在）的 IndexerNameAttribute（第 17.5.2.1 节）提供的名称或默认名称 Item（如果索引器成员不存在）。

对于发生在实例构造函数、静态构造函数、析构函数和运算符的声明中的调用，所使用的成员名称取决于具体实现。

17.5 互操作的特性

注意：本节仅适用于C# 的Microsoft .NET 实现。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

17.5.1 与 COM 和 Win32 组件的互操作

.NET 运行时提供大量特性，通过这些特性，C# 程序可以与使用 COM 和 Win32 DLL 编写的组件进行交互操作。例如，可以在 `static extern` 方法上使用 `DllImport` 特性来表示该方法的实现应该到 Win32 DLL 中去查找。这些特性可在 `System.Runtime.InteropServices` 命名空间中找到，有关这些特性的详细文档可在 .NET 运行时文档中找到。

17.5.2 与其他 .NET 语言的互操作

17.5.2.1 IndexerName 特性

索引器是利用索引属性在 .NET 中实现的，并且具有一个属于 .NET 元数据的名称。如果索引器没有被指定 `IndexerName` 特性，则默认情况下将使用名称 `Item`。`IndexerName` 特性使开发人员可以重写此默认名称并指定不同的名称。

```
namespace System.Runtime.CompilerServices.CSharp
{
   [AttributeUsage(AttributeTargets.Property)]
   public class IndexerNameAttribute: Attribute
   {
      public IndexerNameAttribute(string indexerName) {...}
      public string Value { get {...} }
   }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

18. 不安全代码

如前面几章所定义，核心C# 语言没有将指针列入它所支持的数据类型，从而与 C 和 C++ 有着显著的区别。作为替代，C# 提供了各种引用类型，并能够创建可由垃圾回收器管理的对象。这种设计结合其他功能，使 C# 成为比 C 或 C++ 安全得多的语言。在核心 C# 语言中，不可能有未初始化的变量、“虚”指针或者超过数组的界限对其进行索引的表达式。这样，以往总是不断地烦扰 C 和 C++ 程序的一系列错误就不会再出现了。

尽管实际上对 C 或 C++ 中的每种指针类型构造，C# 都设置了与之对应的引用类型，但仍然会有一些场合需要访问指针类型。例如，当需要与基础操作系统进行交互、访问内存映射设备，或实现一些以时间为关键的算法时，若没有访问指针的手段，就不可能或者至少很难完成。为了满足这样的需求，C# 提供了编写不安全代码 (unsafe code) 的能力。

在不安全代码中，可以声明和操作指针，可以在指针和整型之间执行转换，还可以获取变量的地址，等等。在某种意义上，编写不安全代码很像在 C# 程序中编写 C 代码。

无论从开发人员还是从用户角度来看，不安全代码事实上都是一种“安全”功能。不安全代码必须用修饰符 `unsafe` 明确地标记，这样开发人员就不会误用不安全功能，而执行引擎将确保不会在不受信任的环境中执行不安全代码。

18.1 不安全上下文

C# 的不安全功能仅用于不安全上下文中。不安全上下文是通过在类型或成员的声明中包含一个 `unsafe` 修饰符或者通过使用 *unsafe-statement* 引入的：

- 类、结构、接口或委托的声明可以包含一个 `unsafe` 修饰符，在这种情况下，该类型声明的整个文本范围（包括类、结构或接口的体）被认为是不安全上下文。
- 在字段、方法、属性、事件、索引器、运算符、实例构造函数、析构函数或静态构造函数的声明中，也可以包含一个 `unsafe` 修饰符，在这种情况下，该成员声明的整个文本范围被认为是不安全上下文。
- *unsafe-statement* 使得可以在 *block* 内使用不安全上下文。该语句关联的 *block* 的整个文本范围被认为是不安全上下文。

下面显示了关联的语法扩展。为简洁起见，将使用省略号 (...) 来表示出现在前面章节中的产生式。

class-modifier:
    ...
    `unsafe`

struct-modifier:
    ...
    `unsafe`

interface-modifier:
    ...
    `unsafe`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

delegate-modifier:
    ...
    unsafe

field-modifier:
    ...
    unsafe

method-modifier:
    ...
    unsafe

property-modifier:
    ...
    unsafe

event-modifier:
    ...
    unsafe

indexer-modifier:
    ...
    unsafe

operator-modifier:
    ...
    unsafe

constructor-modifier:
    ...
    unsafe

destructor-declaration:
    attributes$_{opt}$ extern$_{opt}$ unsafe$_{opt}$ ~ *identifier* ( ) *destructor-body*
    attributes$_{opt}$ unsafe$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*

static-constructor-modifiers:
    extern$_{opt}$ unsafe$_{opt}$ static
    unsafe$_{opt}$ extern$_{opt}$ static
    extern$_{opt}$ static unsafe$_{opt}$
    unsafe$_{opt}$ static extern$_{opt}$
    static extern$_{opt}$ unsafe$_{opt}$
    static unsafe$_{opt}$ extern$_{opt}$

embedded-statement:
    ...
    unsafe-statement

unsafe-statement:
    unsafe *block*

在下面的示例中

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

在结构声明中指定的 unsafe 修饰符导致该结构声明的整个文本范围成为不安全上下文。因此，可以将 Left 和 Right 字段声明为指针类型。上面的示例还可以编写为

```
public struct Node
{
   public int Value;
   public unsafe Node* Left;
   public unsafe Node* Right;
}
```

此处，字段声明中的 unsafe 修饰符导致这些声明被认为是不安全上下文。

除了建立不安全上下文从而允许使用指针类型外，unsafe 修饰符对类型或成员没有影响。在下面的示例中

```
public class A
{
   public unsafe virtual void F() {
      char* p;
      ...
   }
}
public class B: A
{
   public override void F() {
      base.F();
      ...
   }
}
```

A 中 F 方法上的 unsafe 修饰符直接导致 F 的文本范围成为不安全上下文并可以在其中使用语言的不安全功能。在 B 中对 F 的重写中，不需要重新指定 unsafe 修饰符，除非 B 中的 F 方法本身需要访问不安全功能。

当指针类型是方法签名的一部分时，情况略有不同

```
public unsafe class A
{
   public virtual void F(char* p) {...}
}
public class B: A
{
   public unsafe override void F(char* p) {...}
}
```

此处，由于 F 的签名包括指针类型，因此它只能写入不安全上下文中。然而，为设置此不安全上下文，既可以将整个类设置为不安全的（如 A 中的情况），也可以仅在方法声明中包含一个 unsafe 修饰符（如 B 中的情况）。

18.2 指针类型

在不安全上下文中，*type*（第 4 章）可以是 *pointer-type*，也可以是 *value-type* 或 *reference-type*。但是，*pointer-type* 也可以在不安全上下文外部的 typeof 表达式（第 7.6.10.6 节）中使用，因为此类不是不安全的。

type:
    ...
    pointer-type

pointer-type 可表示为 *unmanaged-type* 后接一个 * 标记，或者关键字 void 后接一个 * 标记：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

pointer-type:
    unmanaged-type *
    void *

unmanaged-type:
    type

指针类型中，在 * 前面指定的类型称为该指针类型的目标类型 (referent type)。它表示该指针类型的值所指向的变量的类型。

与引用（引用类型的值）不同，指针不受垃圾回收器跟踪（垃圾回收器并不知晓指针和它们指向的数据）。出于此原因，不允许指针指向引用或者包含引用的结构，并且指针的目标类型必须是 *unmanaged-type*。

unmanaged-type 为不是 *reference-type* 或构造类型的任何类型，不在任何嵌套级别上包含 *reference-type* 或构造类型字段。换句话说，*unmanaged-type* 是下列类型之一：

- sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal 或 bool。
- 任何 *enum-type*。
- 任何 *pointer-type*。
- 非构造类型且仅包含 *unmanaged-type* 的字段的任何用户定义 *struct-type*。

将指针和引用进行混合使用时的基本规则是：引用（对象）的目标可以包含指针，但指针的目标不能包含引用。

下表给出了一些指针类型的示例：

示例	说明
byte*	指向 byte 的指针
char*	指向 char 的指针
int**	指向 int 的指针
int*[]	由指向 int 的指针构成的一维数组
void*	指向未知类型的指针

对于某个给定实现，所有的指针类型都必须具有相同的大小和表示形式。

与 C 和 C++ 不同，在 C# 中，当在同一声明中声明多个指针时，* 只与基础类型写在一起，而不充当每个指针名称的前缀标点符号。例如

```
int* pi, pj;    // NOT as int *pi, *pj;
```

类型为 T* 的一个指针的值表示类型为 T 的一个变量的地址。指针间接寻址运算符 *（第 18.5.1 节）可用于访问此变量。例如，给定 int* 类型的变量 P，则表达式 *P 表示在 P 中包含的地址处找到的 int 变量。

与对象引用类似，指针可以是 null。如果将间接寻址运算符应用于 null 指针，则其行为将由实现自己定义。值为 null 的指针表示为将该指针的所有位都置零。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

void* 类型表示指向未知类型的指针。因为目标类型是未知的，所以间接寻址运算符不能应用于 void* 类型的指针，也不能对这样的指针执行任何算术运算。但是，void* 类型的指针可以强制转换为任何其他指针类型（反之亦然）。

指针类型是一个单独类别的类型。与引用类型和值类型不同，指针类型不从 object 继承，而且不存在指针类型和 object 之间的转换。具体而言，指针不支持装箱和取消装箱（第 4.3 节）操作。但是，允许在不同指针类型之间以及指针类型与整型之间进行转换。第 18.4 节中对此进行了介绍。

pointer-type 不能用作类型实参（第 4.4 节），且类型推断（第 7.5.2 节）在泛型方法调用期间失败，因为该调用会将类型实参推断为指针类型。

pointer-type 不能用作动态绑定操作的构成表达式的类型（第 7.2.2 节）。

pointer-type 可用作易失字段的类型（第 10.5.3 节）。

虽然指针可以作为 ref 或 out 参数传递，但这样做可能会导致未定义的行为，例如，指针可能被设置为指向一个局部变量，而当调用方法返回时，该局部变量可能已不存在了；或者指针曾指向一个固定对象，但当调用方法返回时，该对象不再是固定的了。例如：

```
using System;

class Test
{
   static int value = 20;

   unsafe static void F(out int* pi1, ref int* pi2) {
      int i = 10;
      pi1 = &i;

      fixed (int* pj = &value) {
         // ...
         pi2 = pj;
      }
   }

   static void Main() {
      int i = 10;
      unsafe {
         int* px1;
         int* px2 = &i;

         F(out px1, ref px2);

         Console.WriteLine("*px1 = {0}, *px2 = {1}",
            *px1, *px2);   // undefined behavior
      }
   }
}
```

方法可以返回某一类型的值，而该类型可以是指针。例如，给定一个指向连续的 int 值序列的指针、该序列的元素个数，和另外一个 int 值，下面的方法将在该整数序列中查找与该 value 匹配的值，若找到匹配项，则返回该匹配项的地址；否则，它将返回 null：

```
unsafe static int* Find(int* pi, int size, int value) {
   for (int i = 0; i < size; ++i) {
      if (*pi == value)
         return pi;
      ++pi;
   }
   return null;
}
```

在不安全上下文中，可以使用下列几种构造操作指针：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- * 运算符可用于执行指针间接寻址（第 18.5.1 节）。
- -> 运算符可用于通过指针访问结构的成员（第 18.5.2 节）。
- [] 运算符可用于索引指针（第 18.5.3 节）。
- & 运算符可用于获取变量的地址（第 18.5.4 节）。
- ++ 和 -- 运算符可以用于递增和递减指针（第 18.5.5 节）。
- + 和 - 运算符可用于执行指针算术运算（第 18.5.6 节）。
- ==、!=、<、>、<= 和 => 运算符可以用于比较指针（第 18.5.7 节）。
- stackalloc 运算符可用于从调用堆栈中分配内存（第 18.7 节）。
- fixed 语句可用于临时固定一个变量，以便可以获取它的地址（第 18.6 节）。

18.3 固定和可移动变量

address-of 运算符（第 18.5.4 节）和 fixed 语句（第 18.6 节）将变量分为两种类别：固定变量 (Fixed variables) 和可移动变量 (moveable variables)。

固定变量驻留在不受垃圾回收器的操作影响的存储位置中。（固定变量的示例包括局部变量、值参数和由取消指针引用而创建的变量。）另一方面，可移动变量则驻留在会被垃圾回收器重定位或释放的存储位置中。（可移动变量的示例包括对象中的字段和数组的元素。）

& 运算符（第 18.5.4 节）允许不受限制地获取固定变量的地址。但是，由于可移动变量会受到垃圾回收器的重定位或释放，因此可移动变量的地址只能使用 fixed 语句（第 18.6 节）获取，而且该地址只在此 fixed 语句的生存期内有效。

准确地说，固定变量是下列之一：

- 用引用局部变量或值参数的 *simple-name*（第 7.6.2 节）表示的变量（如果该变量未由匿名函数捕获）。
- 用 V.I 形式的 *member-access*（第 7.6.4 节）表示的变量，其中 V 是 *struct-type* 的固定变量。
- 用 *P 形式的 *pointer-indirection-expression*（第 18.5.1 节）、P->I 形式的 *pointer-member-access*（第 18.5.2 节）或 P[E] 形式的 *pointer-element-access*（第 18.5.3 节）表示的变量。

所有其他变量都属于可移动变量。

请注意静态字段属于可移动变量。还请注意即使赋予 ref 或 out 形参的实参是固定变量，它们仍属于可移动变量。最后请注意，由取消指针引用而产生的变量总是属于固定变量。

18.4 指针转换

在不安全上下文中，可供使用的隐式转换的集合（第 0 节）也扩展为包括以下隐式指针转换：

- 从任何 *pointer-type* 到 void* 类型。
- 从 null 文本到任何 *pointer-type*。

另外，在不安全上下文中，可供使用的显式转换的集合（第 6.2 节）也扩展为包括以下显式指针转换：

- 从任何 *pointer-type* 到任何其他 *pointer-type*。
- 从 sbyte、byte、short、ushort、int、uint、long 或 ulong 到任何 *pointer-type*。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 从任何 *pointer-type* 到 sbyte、byte、short、ushort、int、uint、long 或 ulong。

最后，在不安全上下文中，标准隐式转换（第 6.3.1 节）的集包括下列指针转换：

- 从任何 *pointer-type* 到 void* 类型。
- 从 null 文本到任何 *pointer-type*。

两个指针类型之间的转换永远不会更改实际的指针值。换句话说，从一个指针类型到另一个指针类型的转换不会影响由指针给出的基础地址。

当一个指针类型被转换为另一个指针类型时，如果没有将得到的指针正确地对指向的类型对齐，则当结果被取消引用时，该行为将是未定义的。一般情况下，“正确对齐”的概念具有传递性：如果指向类型 A 的指针正确地与指向类型 B 的指针对齐，而此指向类型 B 的指针又正确地与指向类型 C 的指针对齐，则指向类型 A 的指针将正确地与指向类型 C 的指针对齐。

请考虑下列情况，其中具有一个类型的变量被通过指向一个不同类型的指针访问：

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;      // undefined
*pi = 123456;     // undefined
```

当一个指针类型被转换为指向字节的指针时，转换后的指针将指向原来所指变量的地址中的最低寻址字节。连续增加该变换后的指针（最大可达到该变量所占内存空间的大小），将产生指向该变量的其他字节的指针。例如，下列方法将 double 型变量中的八个字节的每一个显示为一个十六进制值：

```
using System;

class Test
{
   unsafe static void Main() {
      double d = 123.456e23;
      unsafe {
         byte* pb = (byte*)&d;
         for (int i = 0; i < sizeof(double); ++i)
            Console.Write("{0:X2} ", *pb++);
         Console.WriteLine();
      }
   }
}
```

当然，产生的输出取决于字节存储顺序 (Endianness)。

指针和整数之间的映射由实现定义。但是，在具有线性地址空间的 32 位和 64 位 CPU 体系结构上，指针和整型之间的转换通常与 uint 或 ulong 类型的值与这些整型之间的对应方向上的转换具有完全相同的行为。

18.4.1 指针数组

可以在不安全上下文中构造指针数组。只有一部分适用于其他数组类型的转换适用于指针数组:

- 从任意 *array-type* 到 System.Array 及其实现的接口的隐式引用转换（第 6.1.6 节）也适用于指针数组。但是，由于指针类型不可转换为 object，因此只要尝试通过 System.Array 或其实现的接口访问数组元素，就会导致在运行时出现异常。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 从一维数组类型 `S[]` 到 `System.Collections.Generic.IList<T>` 及其泛型基接口的隐式和显式引用转换（第 6.1.6、6.2.4 节）从不适用于指针数组，因为指针类型不能用作类型实参，且不存在从指针类型到非指针类型的转换。
- 从 `System.Array` 及其实现的接口到任意 *array-type* 的显式引用转换（第 6.2.4 节）均适用于指针数组。
- 从 `System.Collections.Generic.IList<S>` 及其基接口到一维数组类型 `T[]` 的隐式引用转换（第 6.2.4 节）在任何情况下均不适用于指针数组，因为指针类型不能用作类型实参，且不存在从指针类型到非指针类型的转换。

这些限制意味着通过第 8.8.4 节中给出的数组对 `foreach` 语句进行的扩展不能用于指针数组。而下列形式的 foreach 语句

```
foreach (V v in x) embedded-statement
```

(其中 `x` 的类型为具有 `T[,,…,]`, *n* 形式的数组类型，n 为维度数减 1，`T` 或 `V` 为指针类型)使用嵌套 for 循环扩展，如下所示：

```
{
   T[,,…,] a = x;
   V v;
   for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
   for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
   …
   for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {
      v = (V)a.GetValue(i0,i1,…,in);
      embedded-statement
   }
}
```

变量 `a`，`i0`, `i1`, … `in` 对 `x` 或 *embedded-statement* 或该程序的任何其他源代码均不可见或不可访问。变量 `v` 在嵌入语句中是只读的。如果不存在从 `T`（元素类型）到 `V` 的显式转换（第 18.4 节），则会出错且不会执行下面的步骤。如果 `x` 具有值 `null`，则将在运行时引发 `System.NullReferenceException`。

18.5 表达式中的指针

在不安全上下文中，表达式可能产生指针类型的结果，但是在不安全上下文以外，表达式为指针类型属于编译时错误。准确地说，在不安全上下文以外，如果任何 *simple-name*（第 7.6.2 节）、*member-access*（第 7.6.4 节）、*invocation-expression*（第 7.6.5 节）或 *element-access*（第 7.6.6 节）属于指针类型，则将发生编译时错误。

在不安全上下文中，*primary-no-array-creation-expression*（第 7.6 节）和 *unary-expression*（第 7.7 节）产生式允许使用下列附加构造：

primary-no-array-creation-expression:
…
pointer-member-access
pointer-element-access
sizeof-expression

unary-expression:
…
pointer-indirection-expression
addressof-expression

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

以下几节对这些构造进行了描述。相关的语法暗示了不安全运算符的优先级和结合性。

18.5.1 指针间接寻址

pointer-indirection-expression 包含一个星号 (*)，后接一个 *unary-expression*。

pointer-indirection-expression:
    `*` *unary-expression*

一元 * 运算符表示指针间接寻址并且用于获取指针所指向的变量。计算 `*P` 得到的结果（其中 `P` 为指针类型 `T*` 的表达式）是类型为 `T` 的一个变量。将一元 * 运算符应用于 `void*` 类型的表达式或者应用于不是指针类型的表达式属于编译时错误。

将一元 * 运算符应用于 `null` 指针的效果是由实现定义的。具体而言，不能保证此操作会引发 `System.NullReferenceException`。

如果已经将无效值赋给指针，则一元 * 运算符的行为是未定义的。通过一元 * 运算符取消指针引用有时会产生无效值，这些无效值包括：没能按所指向的类型正确对齐的地址（请参见第18.4 节中的示例）和超过生存期的变量的地址。

出于明确赋值分析的目的，通过计算 `*P` 形式的表达式产生的变量被认为是初始化赋过值的（第5.3.1 节）。

18.5.2 指针成员访问

pointer-member-access 包含一个 *primary-expression*，后接一个“`->`”标记，最后是一个 *identifier* 和一个可选 *type-argument-list*。

pointer-member-access:
    primary-expression `->` *identifier* *type-argument-list*$_{opt}$

在 `P->I` 形式的指针成员访问中，`P` 必须是除 `void*` 以外的某个指针类型的表达式，而 `I` 必须表示 `P` 所指向的类型的可访问成员。

`P->I` 形式的指针成员访问的计算方式与 `(*P).I` 完全相同。有关指针间接寻址运算符 (*) 的说明，请参见第 18.5.1 节。有关成员访问运算符 (.) 的说明，请参见第 7.6.4 节。

在下面的示例中

```
using System;

struct Point
{
   public int x;
   public int y;

   public override string ToString() {
      return "(" + x + "," + y + ")";
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static void Main() {
      Point point;
      unsafe {
         Point* p = &point;
         p->x = 10;
         p->y = 20;
         Console.WriteLine(p->ToString());
      }
   }
}
```

-> 运算符用于通过指针访问结构中的字段和调用结构中的方法。由于 P->I 操作完全等效于 (*P).I，因此 Main 方法可以等效地编写为：

```
class Test
{
   static void Main() {
      Point point;
      unsafe {
         Point* p = &point;
         (*p).x = 10;
         (*p).y = 20;
         Console.WriteLine((*p).ToString());
      }
   }
}
```

18.5.3 指针元素访问

pointer-element-access 包括一个 *primary-no-array-creation-expression*，后接一个用“[”和“]”括起来的表达式。

pointer-element-access:
primary-no-array-creation-expression [*expression*]

在形式为 P[E] 的指针元素访问中，P 必须为除 void* 之外的指针类型表达式，E 必须为可以隐式转换为 int、uint、long 或 ulong 的表达式。

P[E] 形式的指针元素访问的计算方式与 *(P + E) 完全相同。有关指针间接寻址运算符 (*) 的说明，请参见第 18.5.1 节。有关指针加法运算符 (+) 的说明，请参见第 18.5.6 节。

在下面的示例中

```
class Test
{
   static void Main() {
      unsafe {
         char* p = stackalloc char[256];
         for (int i = 0; i < 256; i++) p[i] = (char)i;
      }
   }
}
```

指针元素访问用于在 for 循环中初始化字符缓冲区。由于 P[E] 操作完全等效于 *(P + E)，因此示例可以等效地编写为：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static void Main() {
      unsafe {
         char* p = stackalloc char[256];
         for (int i = 0; i < 256; i++) *(p + i) = (char)i;
      }
   }
}
```

指针元素访问运算符不能检验是否发生访问越界错误，而且当访问超出界限的元素时行为是未定义的。这与 C 和 C++ 相同。

18.5.4 address-of 运算符

addressof-expression 包含一个“&”符 (&)，后接一个 *unary-expression*。

addressof-expression:
    & *unary-expression*

如果给定类型为 T 且属于固定变量（第 18.3 节）的表达式 E，构造 &E 将计算由 E 给出的变量的地址。计算的结果是一个类型为 T* 的值。如果 E 不属于变量，如果 E 属于只读局部变量，或如果 E 表示可移的变量，则将发生编译时错误。在最后一种情况中，可以先利用固定语句（第 18.6 节）临时“固定”该变量，再获取它的地址。如第 7.6.4 节中所述，如果在实例构造函数或静态构造函数之外，在结构或类中定义了 readonly 字段，则该字段被认为是一个值，而不是变量。因此，无法获取它的地址。与此类似，无法获取常量的地址。

& 运算符不要求它的参数先被明确赋值，但是在执行了 & 操作后，该运算符所应用于的那个变量在此操作发生的执行路径中被“认为是”已经明确赋值的。这意味着，由程序员负责确保在相关的上下文中对该变量实际进行合适的初始化。

在下面的示例中

```
using System;
class Test
{
   static void Main() {
      int i;
      unsafe {
         int* p = &i;
         *p = 123;
      }
      Console.WriteLine(i);
   }
}
```

初始化 p 的代码执行了 &i 操作，此后 i 被认为是明确赋值的。对 *p 的赋值实际上是初始化了 i，但设置此初始化是程序员的责任，而且如果移除此赋值语句，也不会发生编译时错误。

上述 & 运算符的明确赋值规则可以避免局部变量的冗余初始化。例如，许多外部 API 要求获取指向结构的指针，而由此 API 来填充该结构。对此类 API 进行的调用通常会传递局部结构变量的地址，而如果没有上述规则，则将需要对此结构变量进行冗余初始化。

18.5.5 指针递增和递减

在不安全上下文中，++ 和 -- 运算符（第 7.6.9 节和第 7.7.5 节）可以应用于除 void* 以外的所有类型的指针变量。因此，为每个指针类型 T* 都隐式定义了下列运算符：

```
T* operator ++(T* x);
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
T* operator --(T* x);
```

这些运算符分别产生与 x + 1 和 x - 1（第 18.5.6 节）相同的结果。换句话说，对于 T* 类型的指针变量，++ 运算符将该变量的地址加上 sizeof(T)，而 -- 运算符则将该变量的地址减去 sizeof(T)。

如果指针递增或递减运算的结果超过指针类型的域，则结果是由实现定义的，但不会产生异常。

18.5.6 指针算术运算

在不安全上下文中，+ 和 - 运算符（第 7.8.4 节和第 7.8.5 节）可以应用于除 void* 以外的所有指针类型的值。因此，为每个指针类型 T* 都隐式定义了下列运算符：

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator –(T* x, int y);
T* operator –(T* x, uint y);
T* operator –(T* x, long y);
T* operator –(T* x, ulong y);

long operator –(T* x, T* y);
```

给定指针类型 T* 的表达式 P 和类型为 int、uint、long 或 ulong 的表达式 N，表达式 P + N 和 N + P 将计算类型为 T* 的指针值，该值等于由 P 给出的地址加上 N * sizeof(T)。与此类似，表达式 P - N 将计算类型为 T* 的指针值，该值等于由 P 给出的地址减去 N * sizeof(T)。

给定指针类型 T* 的两个表达式 P 和 Q，表达式 P – Q 将先计算 P 和 Q 给出的地址之间的差，然后用 sizeof(T) 去除该差值。计算结果的类型始终为 long。实际上，P - Q 将按 ((long)(P) - (long)(Q)) / sizeof(T) 进行计算。

例如：

```
using System;

class Test
{
   static void Main() {
      unsafe {
         int* values = stackalloc int[20];
         int* p = &values[1];
         int* q = &values[15];
         Console.WriteLine("p - q = {0}", p - q);
         Console.WriteLine("q - p = {0}", q - p);
      }
   }
}
```

生成以下输出：

```
p - q = -14
q - p = 14
```

如果在执行上述指针算法时，计算结果超越该指针类型的域，则将以实现所定义的方式截断结果，但是不会产生异常。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

18.5.7 指针比较

在不安全上下文中，==、!=、<、>、<= 和 => 运算符（第 7.10 节）可以应用于所有指针类型的值。指针比较运算符有：

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

由于存在从任何指针类型到 void* 类型的隐式转换，因此可以使用这些运算符来比较任何指针类型的操作数。比较运算符像比较无符号整数一样比较两个操作数给出的地址。

18.5.8 sizeof 运算符

sizeof 运算符返回由给定类型的变量占用的字节数。被指定为 sizeof 的操作数的类型必须为 *unmanaged-type*（第 18.2 节）。

sizeof-expression:
    sizeof (*unmanaged-type*)

sizeof 运算符的结果是 int 类型的值。对于某些预定义类型，sizeof 运算符将产生如下表所示的常量值。

表达式	结果
sizeof(sbyte)	1
sizeof(byte)	1
sizeof(short)	2
sizeof(ushort)	2
sizeof(int)	4
sizeof(uint)	4
sizeof(long)	8
sizeof(ulong)	8
sizeof(char)	2
sizeof(float)	4
sizeof(double)	8
sizeof(bool)	1

对于所有其他类型，sizeof 运算符的结果是由实现定义的，并且属于值而不是常量。

一个结构所属的各个成员以什么顺序被装入该结构中，没有明确规定。

出于对齐的目的，在结构的开头、结构内以及结构的结尾处可以插入一些未命名的填充位。这些填充位的内容是不确定的。

当 sizeof 应用于具有结构类型的操作数时，结果是该类型变量所占的字节总数（包括所有填充位在内）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

18.6 fixed 语句

在不安全上下文中，*embedded-statement*（第 8 章）产生式允许使用一个附加结构即 `fixed` 语句，该语句用于“固定”可移动变量，从而使该变量的地址在语句的持续时间内保持不变。

embedded-statement:
    ...
    fixed-statement

fixed-statement:
    `fixed` `(` *pointer-type* *fixed-pointer-declarators* `)` *embedded-statement*

fixed-pointer-declarators:
    fixed-pointer-declarator
    fixed-pointer-declarators `,` *fixed-pointer-declarator*

fixed-pointer-declarator:
    identifier `=` *fixed-pointer-initializer*

fixed-pointer-initializer:
    `&` *variable-reference*
    expression

如上述产生式所述，每个 *fixed-pointer-declarator* 声明一个给定 *pointer-type* 的局部变量，并使用由相应的 *fixed-pointer-initializer* 计算的地址初始化该局部变量。在 `fixed` 语句中声明的局部变量的可访问范围仅限于：在该变量声明右边的所有 *fixed-pointer-initializer* 中，以及在该 `fixed` 语句的 *embedded-statement* 中。由 `fixed` 语句声明的局部变量被视为只读。如果嵌入语句试图修改此局部变量（通过赋值或 `++` 和 `--` 运算符）或者将它作为 `ref` 或 `out` 参数传递，则将出现编译时错误。

fixed-pointer-initializer 可以是下列之一：

- “`&`”标记，后接一个 *variable-reference*（第 5.3.3 节），它引用非托管类型 `T` 的可移动变量（第 18.3 节），前提是类型 `T*` 可以隐式转换为 `fixed` 语句中给出的指针类型。在这种情况下，初始值设定项将计算给定变量的地址，而 `fixed` 语句在生存期内将保证该变量的地址不变。
- 元素类型为非托管类型 `T` 的 *array-type* 的表达式，前提是类型 `T*` 可隐式转换为 `fixed` 语句中给出的指针类型。在这种情况下，初始值设定项将计算数组中第一个元素的地址，而 `fixed` 语句在生存期内将保证整个数组的地址保持不变。如果数组表达式为 null 或者数组具有零个元素，则 `fixed` 语句的行为由实现定义。
- `string` 类型的表达式，前提是类型 `char*` 可以隐式转换为 `fixed` 语句中给出的指针类型。在这种情况下，初始值设定项将计算字符串中第一个字符的地址，而 `fixed` 语句在生存期内将保证整个字符串的地址不变。如果字符串表达式为 null，则 `fixed` 语句的行为由实现定义。
- 引用可移动变量的固定大小缓冲区成员的 *simple-name* 或 *member-access*，前提是固定大小缓冲区成员的类型可以隐式转换为 `fixed` 语句中给出的指针类型。这种情况下，初始值设定项计算出指向固定大小缓冲区（第 18.7.2 节）第一个元素的指针，并且该固定大小缓冲区保证在 `fixed` 语句的持续时间内保留在某个固定地址。

对于每个由 *fixed-pointer-initializer* 计算的地址，`fixed` 语句确保由该地址引用的变量在 `fixed` 语句的生存期内不会被垃圾回收器重定位或者释放。例如，如果由 *fixed-pointer-initializer* 计算的地址引用对象的字段或数组实例的元素，`fixed` 语句将保证包含该字段或元素的对象实例本身也不会在该语句的生存期内被重定位或者释放。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

确保由 fixed 语句创建的指针在执行这些语句之后不再存在是程序员的责任。例如，当 fixed 语句创建的指针被传递到外部 API 时，确保 API 不会在内存中保留这些指针是程序员的责任。

固定对象可能导致堆中产生存储碎片（因为它们无法移动）。出于该原因，只有在绝对必要时才应当固定对象，而且固定对象的时间越短越好。

下面的示例

```
class Test
{
   static int x;
   int y;

   unsafe static void F(int* p) {
      *p = 1;
   }

   static void Main() {
      Test t = new Test();
      int[] a = new int[10];
      unsafe {
         fixed (int* p = &x) F(p);
         fixed (int* p = &t.y) F(p);
         fixed (int* p = &a[0]) F(p);
         fixed (int* p = a) F(p);
      }
   }
}
```

演示了 fixed 语句的几种用法。第一条语句固定并获取一个静态字段的地址，第二条语句固定并获取一个实例字段的地址，第三条语句固定并获取一个数组元素的地址。在这几种情况下，直接使用常规 & 运算符都是错误的，这是因为这些变量都属于可移动变量。

上面示例中的第四个 fixed 语句生成与第三个语句类似的结果。

此 fixed 语句示例将使用 string：

```
class Test
{
   static string name = "xx";

   unsafe static void F(char* p) {
      for (int i = 0; p[i] != '\0'; ++i)
         Console.WriteLine(p[i]);
   }

   static void Main() {
      unsafe {
         fixed (char* p = name) F(p);
         fixed (char* p = "xx") F(p);
      }
   }
}
```

在不安全上下文中，一维数组的数组元素按递增索引顺序存储，从索引 0 开始，到索引 Length – 1 结束。对于多维数组，数组元素按这样的方式存储：首先增加最右边维度的索引，然后是左边紧邻的维度，依此类推直到最左边。在获取指向数组实例 a 的指针 p 的 fixed 语句内，从 p 到 p + a.Length – 1 范围内的每个指针值均表示数组中的一个元素的地址。与此类似，从 p[0] 到 p[a.Length – 1] 范围内的变量表示实际的数组元素。已知数组的存储方式，可以将任意维度的数组都视为线性的。

例如：

```
using System;
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static void Main() {
      int[,,] a = new int[2,3,4];
      unsafe {
         fixed (int* p = a) {
            for (int i = 0; i < a.Length; ++i)  // treat as linear
               p[i] = i;
         }
      }
      for (int i = 0; i < 2; ++i)
         for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 4; ++k)
               Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
            Console.WriteLine();
         }
   }
}
```

生成以下输出：

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

在下面的示例中

```
class Test
{
   unsafe static void Fill(int* p, int count, int value) {
      for (; count != 0; count--) *p++ = value;
   }
   static void Main() {
      int[] a = new int[100];
      unsafe {
         fixed (int* p = a) Fill(p, 100, -1);
      }
   }
}
```

使用一条 fixed 语句固定一个数组，以便可以将该数组的地址传递给一个采用指针作为参数的方法。

在下面的示例中：

```
unsafe struct Font
{
   public int size;
   public fixed char name[32];
}
class Test
{
   unsafe static void PutString(string s, char* buffer, int bufSize) {
      int len = s.Length;
      if (len > bufSize) len = bufSize;
      for (int i = 0; i < len; i++) buffer[i] = s[i];
      for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
   }
   Font f;
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        unsafe static void Main()
        {
            Test test = new Test();
            test.f.size = 10;
            fixed (char* p = test.f.name) {
                PutString("Times New Roman", p, 32);
            }
        }
    }
```

一个固定语句用于固定一个结构的固定大小缓冲区，因此可以将该缓冲区的地址用作指针。

通过固定字符串实例产生的 `char*` 类型的值始终指向以 null 结尾的字符串。在获取指向字符串实例 `s` 的指针 `p` 的 fixed 语句内，从 `p` 到 `p + s.Length - 1` 范围内的指针值表示字符串中字符的地址，而指针值 `p + s.Length` 则始终指向一个 null 字符(值为 `'\0'` 的字符)。

通过固定指针修改托管类型的对象可能导致未定义的行为。例如，由于字符串是不可变的，因此程序员应确保指向固定字符串的指针所引用的字符不被修改。

这种字符串的自动空字符终止功能，大大方便了调用需要“C 风格”字符串的外部 API。但请注意，核心 C# 允许字符串实例包含空字符。如果字符串中存在此类空字符，则在将字符串视为空终止的 `char*` 时将出现截断。

18.7 固定大小缓冲区

固定大小缓冲区用于将“C 风格”的内联数组声明为结构的成员，且主要用于与非托管 API 交互。

18.7.1 固定大小缓冲区的声明

固定大小缓冲区 (fixed size buffer) 是一个成员，表示给定类型的变量的固定长度缓冲区的存储区。固定大小缓冲区声明引入了给定元素类型的一个或多个固定大小缓冲区。仅允许在结构声明中使用固定大小缓冲区，且只能出现在不安全上下文（第 18.1 节）中。

struct-member-declaration:
    ...
    fixed-size-buffer-declaration

fixed-size-buffer-declaration:
    attributes$_{opt}$ *fixed-size-buffer-modifiers*$_{opt}$ `fixed` *buffer-element-type*
        fixed-size-buffer-declarators `;`

fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-size-buffer-modifier *fixed-size-buffer-modifiers*

fixed-size-buffer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `unsafe`

buffer-element-type:
    type

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

fixed-size-buffer-declarators:
fixed-size-buffer-declarator
fixed-size-buffer-declarator , *fixed-size-buffer-declarators*

fixed-size-buffer-declarator:
identifier [*constant-expression*]

固定大小缓冲区声明可包括一组特性（第 17 节）、一个 new 修饰符（第 10.2.2 节）、四个访问修饰符（第 10.2.3 节）的一个有效组合和一个 unsafe 修饰符（第 18.1 节）。这些特性和修饰符适用于由固定大小缓冲区声明所声明的所有成员。同一个修饰符在一个固定大小缓冲区声明中出现多次是一个错误。

固定大小缓冲区声明不允许包含 static 修饰符。

固定大小缓冲区声明的缓冲区元素类型指定了由该声明引入的缓冲区的元素类型。缓冲区元素类型必须为下列预定义类型之一：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 bool。

缓冲区元素类型后接一个固定大小缓冲区声明符的列表，该列表中的每个声明符引入一个新成员。固定大小缓冲区声明符由一个用于命名成员的标识符以及标识符后面由 [和] 标记括起来的常量表达式所组成。该常量表达式表示在由该固定大小缓冲区声明符引入的成员中的元素数量。该常量表达式的类型必须可隐式转换为类型 int，并且该值必须是非零的正整数。

固定大小缓冲区的元素保证在内存中按顺序放置。

声明多个固定大小缓冲区的固定大小缓冲区声明相当于单个固定大小缓冲区的带有相同特性和元素类型的多个声明。例如

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

相当于

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
   public fixed int z[100];
}
```

18.7.2 表达式中的固定大小缓冲区

固定大小缓冲区成员的成员查找（第 7.3 节）过程与字段的成员查找完全相同。

可使用 *simple-name*（第 7.5.2 节）或 *member-access*（第 7.5.4 节）在表达式中引用固定大小缓冲区。

当固定大小缓冲区成员作为简单名称被引用时，其效果与 this.I 形式的成员访问相同，其中 I 为固定大小缓冲区成员。

在 E.I 形式的成员访问中，如果 E 为结构类型，并且在该结构类型中通过 I 的成员查找标识了一个固定大小成员，则如下计算并归类 E.I：

- 如果表达式 E.I 不属于不安全上下文，则发生编译时错误。
- 如果 E 归类为值类别，则发生编译时错误。
- 否则，如果 E 为可移动变量(第 18.3 节)并且表达式 E.I 不是 *fixed-pointer-initializer*（第 18.6 节），则发生编译时错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 否则，E 引用固定变量，并且该表达式的结果为指向 E 中的固定大小缓冲区成员 I 的第一个元素的指针。结果为类型 S*，其中 S 为 I 的元素类型，并且归类为值。

可使用指针操作从第一个元素开始访问固定大小缓冲区的后续元素。与访问数组不同，访问固定大小缓冲区的元素是不安全操作，并且不进行范围检查。

下面的示例声明并使用了一个包含固定大小缓冲区成员的结构。

```
unsafe struct Font
{
   public int size;
   public fixed char name[32];
}

class Test
{
   unsafe static void PutString(string s, char* buffer, int bufSize) {
      int len = s.Length;
      if (len > bufSize) len = bufSize;
      for (int i = 0; i < len; i++) buffer[i] = s[i];
      for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
   }

   unsafe static void Main()
   {
      Font f;
      f.size = 10;
      PutString("Times New Roman", f.name, 32);
   }
}
```

18.7.3 明确赋值 检查

固定大小缓冲区不接受明确赋值检查（第 5.3 节），并且为了对结构类型变量进行明确赋值检查，忽略固定大小缓冲区成员。

当包含固定大小缓冲区成员的最外层结构变量为静态变量、类实例的实例变量或数组元素时，该固定大小缓冲区的元素自动初始化为其默认值（第 5.2 节）。而在所有其他情况下，固定大小缓冲区的初始内容未定义。

18.8 堆栈分配

在不安全上下文中，局部变量声明（第 8.5.1 节）可以包含一个从调用堆栈中分配内存的堆栈分配初始值设定项。

local-variable-initializer:
    ...
    stackalloc-initializer

stackalloc-initializer:
    stackalloc *unmanaged-type* [*expression*]

上述产生式中，*unmanaged-type* 表示将在新分配的位置中存储的项的类型，而 *expression* 则指示这些项的数目。合在一起，它们指定所需的分配大小。由于堆栈分配的大小不能为负值，因此将项的数目指定为计算结果为负值的 *constant-expression* 属于编译时错误。

stackalloc T[E] 形式的堆栈分配初始值设定项要求 T 必须为非托管类型（第 18.2 节），E 必须为 int 类型的表达式。该构造从调用堆栈中分配 E * sizeof(T) 个字节，并返回一个指向新分配的块的、类型 T* 的指针。如果 E 为负值，则其行为是未定义的。如果 E 为零，则不进行任何分配，并且返回的指针由实现定义。如果没有足够的内存以分配给定大小的块，则引发 System.StackOverflowException。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

新分配的内存的内容是未定义的。

在 catch 或 finally 块（第 8.10 节）中不允许使用堆栈分配初始值设定项。

无法显式释放利用 stackalloc 分配的内存。在函数成员的执行期间创建的所有堆栈分配内存块都将在该函数成员返回时自动丢弃。这对应于 alloca 函数，它是通常存在于 C 和 C++ 实现中的一个扩展。

在下面的示例中

```
using System;

class Test
{
   static string IntToString(int value) {
      int n = value >= 0? value: -value;
      unsafe {
         char* buffer = stackalloc char[16];
         char* p = buffer + 16;
         do {
            *--p = (char)(n % 10 + '0');
            n /= 10;
         } while (n != 0);
         if (value < 0) *--p = '-';
         return new string(p, 0, (int)(buffer + 16 - p));
      }
   }

   static void Main() {
      Console.WriteLine(IntToString(12345));
      Console.WriteLine(IntToString(-999));
   }
}
```

在 IntToString 方法中使用了 stackalloc 初始值设定项，以在堆栈上分配一个 16 个字符的缓冲区。此缓冲区在该方法返回时自动丢弃。

18.9 动态内存分配

除 stackalloc 运算符外，C# 不提供其他预定义构造来管理那些不受垃圾回收控制的内存。这些服务通常是由支持类库提供或者直接从基础操作系统导入的。例如，下面的 Memory 类阐释了可以如何从 C# 访问基础操作系统的有关堆处理的各种函数：

```
using System;
using System.Runtime.InteropServices;

public unsafe class Memory
{
   // Handle for the process heap. This handle is used in all calls to the
   // HeapXXX APIs in the methods below.

   static int ph = GetProcessHeap();

   // Private instance constructor to prevent instantiation.

   private Memory() {}

   // Allocates a memory block of the given size. The allocated memory is
   // automatically initialized to zero.

   public static void* Alloc(int size) {
      void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
      if (result == null) throw new OutOfMemoryException();
      return result;
   }
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        // Copies count bytes from src to dst. The source and destination
        // blocks are permitted to overlap.
        public static void Copy(void* src, void* dst, int count) {
           byte* ps = (byte*)src;
           byte* pd = (byte*)dst;
           if (ps > pd) {
              for (; count != 0; count--) *pd++ = *ps++;
           }
           else if (ps < pd) {
              for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
           }
        }
        // Frees a memory block.
        public static void Free(void* block) {
           if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
        }
        // Re-allocates a memory block. If the reallocation request is for a
        // larger size, the additional region of memory is automatically
        // initialized to zero.
        public static void* ReAlloc(void* block, int size) {
           void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
           if (result == null) throw new OutOfMemoryException();
           return result;
        }
        // Returns the size of a memory block.
        public static int SizeOf(void* block) {
           int result = HeapSize(ph, 0, block);
           if (result == -1) throw new InvalidOperationException();
           return result;
        }
        // Heap API flags
        const int HEAP_ZERO_MEMORY = 0x00000008;
        // Heap API functions
        [DllImport("kernel32")]
        static extern int GetProcessHeap();
        [DllImport("kernel32")]
        static extern void* HeapAlloc(int hHeap, int flags, int size);
        [DllImport("kernel32")]
        static extern bool HeapFree(int hHeap, int flags, void* block);
        [DllImport("kernel32")]
        static extern void* HeapReAlloc(int hHeap, int flags,
           void* block, int size);
        [DllImport("kernel32")]
        static extern int HeapSize(int hHeap, int flags, void* block);
     }
```

以下给出一个使用 Memory 类的示例：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
class Test
{
   static void Main() {
      unsafe {
         byte* buffer = (byte*)Memory.Alloc(256);
         try {
            for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
            byte[] array = new byte[256];
            fixed (byte* p = array) Memory.Copy(buffer, p, 256);
         }
         finally {
            Memory.Free(buffer);
         }
         for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
      }
   }
}
```

此示例通过 Memory.Alloc 分配了 256 字节的内存，并且使用从 0 增加到 255 的值初始化该内存块。它然后分配一个具有 256 个元素的字节数组并使用 Memory.Copy 将内存块的内容复制到此字节数组中。最后，使用 Memory.Free 释放内存块并将字节数组的内容输出到控制台上。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

A. 文档注释

C# 提供一种机制，使程序员可以使用含有 XML 文本的特殊注释语法为他们的代码编写文档。在源代码文件中，可以使用特定形式的注释来指导工具从这些注释及其后的源代码元素生成 XML。使用这类语法的注释称为文档注释 (documentation comment)。这些注释后面必须紧跟用户定义的类型（如类、委托或接口）或者成员（如字段、事件、属性或方法）。该 XML 生成工具称为***文档生成器***。\t "See documentation comment" （此生成器可以但不需要是 C# 编译器本身。）文档生成器生成的输出称为***文档文件***。\t "See documentation comment" \b 文档文件用作***文档查看器***（\t "See documentation comment" \b \b 一个用于生成类型信息及其关联文档的某种可视显示的工具）的输入。

此规范推荐了一组在文档注释中使用的标记，但是这些标记不是必须使用的，如果需要也可以使用其他标记，只要遵循“格式良好的 XML”规则即可。

A.1 介绍

具有特殊格式的注释可用于指导某个工具根据这些注释 和它们后面的源代码元素生成 XML。这类注释是以三个斜杠 (///) 开始的单行注释，或者是以一个斜杠和两个星号 /**) 开始的分隔注释。这些注释后面必须紧跟它们所注释的用户定义类型（如类、委托或接口）或者成员（如字段、事件、属性或方法）。特性节（第 17.2 节）被视为声明的一部分，因此，文档注释必须位于应用到类型或成员的特性之前。

语法：

single-line-doc-comment:
    /// *input-characters*$_{opt}$

delimited-doc-comment:
    /** *delimited-comment-text*$_{opt}$ */

在 *single-line-doc-comment* 中，如果当前 *single-line-doc-comment* 旁边的每 *single-line-doc-comment* 上的 /// 字符后接有 *whitespace* 字符，则此 *whitespace* 字符不包括在 XML 输出中。

在 *delimited-doc-comment* 中，如果第二行上的第一个非 *whitespace* 字符是一个 *asterisk*，并且在 *delimited-doc-comment* 内的每行开头都重复同一个由可选 *whitespace* 字符和 *asterisk* 字符组成的样式，则该重复出现的样式所含的字符不包括在 XML 输出中。此样式中，可以在 *asterisk* 字符之前或之后包括 *whitespace* 字符。

示例：

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
   /// <summary>method <c>draw</c> renders the point.</summary>
   void draw() {…}
}
```

文档注释内的文本必须根据 XML 规则 (http://www.w3.org/TR/REC-xml) 设置正确的格式。如果 XML 不符合标准格式，将生成警告，并且文档文件将包含一条注释，指出遇到错误。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

尽管开发人员可自由创建自己的标记集，但第 A.2 节中定义有建议的标记集。某些建议的标记具有特殊含义：

- `<param>` 标记用于描述参数。如果使用这样的标记，文档生成器必须验证指定参数是否存在以及文档注释中是否描述了所有参数。如果此验证失败，文档生成器将发出警告。
- `cref` \b 特性可以附加到任意标记，以提供对代码元素的引用。文档生成器必须验证此代码元素是否存在。如果验证失败，文档生成器将发出警告。查找在 `cref` 特性中描述的名称时，文档生成器必须根据源代码中出现的 `using` 语句来考虑命名空间的可见性。对于泛型代码元素，不能使用正常的泛型语法（即“`List<T>`”），因为该语法会生成无效的 XML。可以使用大括号代替方括号（即“`List{T}`”），也可以使用 XML 转义语法（即“`List<T>`”）。
- `<summary>` 标记旨在标出可由文档查看器显示的有关类型或成员的额外信息。
- `<include>` 标记表示应该包含的来自外部 XML 文件的信息。

注意，文档文件并不提供有关类型和成员的完整信息（例如，它不包含任何关于类型的信息）。若要获得有关类型或成员的完整信息，必须协同使用文档文件与对实际涉及的类型或成员的反射调用。

A.2 建议的标记

依据 XML 的规则，文档生成器必须接受并处理任何有效标记。下列标记提供了用户文档中常用的功能。（当然，也可能有其他标记。）

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

标记	章节	用途
<c>	A.2.1	将文本设置为类似代码的字体
<code>	A.2.2	将一行或多行源代码或程序输出设置为某种字体
<example>	A.2.3	表示所含的是示例
<exception>	A.2.4	标识方法可能引发的异常
<include>	A.2.5	包括来自外部文件的 XML
<list>	A.2.6	创建列表或表
<para>	A.2.7	用于将结构添加到文本中
<param>	A.2.8	描述方法或构造函数的参数
<paramref>	A.2.9	确认某个单词是参数名
<permission>	A.2.10	描述成员的安全性和访问权限
<remark>	A.2.11	描述有关类型的更多信息
<returns>	A.2.12	描述方法的返回值
<see>	A.2.13	指定链接
<seealso>	A.2.14	生成“请参见”项
<summary>	A.2.15	描述类型或类型的成员
<value>	A.2.16	描述属性
<typeparam>		描述泛型类型形参
<typeparamref>		标识某个单词是类型形参名称

A.2.1 <c> \t "See <c>" \b

此标记提供一种机制以指示用特殊字体（如用于代码块的字体）设置说明中的文本段落。对于实际代码行，请使用 <code>（第 A.2.2 节）。

语法：

```
<c>text</c>
```

示例：

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
public class Point
{
   // ...
}
```

A.2.2 <code> \t "See <code>" \b

此标记用于将一行或多行源代码或程序输出设置为某种特殊字体。对于叙述中较小的代码段，请使用 <c>（第 A.2.1 节）。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

语法：

```
<code>source code or program output</code>
```

示例：

```
/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// <example>For example:
/// <code>
///   Point p = new Point(3,5);
///   p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

A.2.3 <example> \t "See <example>" \b

此标记用于在注释中插入代码示例，以说明如何使用所关联的方法或其他库成员。通常，此标记是同标记 <code>（第 A.2.2 节）一起使用的。

语法：

```
<example>description</example>
```

示例：

有关示例，请参见 <code> (§ A.2.2)。

A.2.4 <exception> \t "See <exception>" \b

此标记提供了一种用于记录方法可以引发的异常的方式。

语法：

```
<exception cref="member">description</exception>
```

其中

cref="*member*"

成员的名称。文档生成器检查给定成员是否存在，并将 *member* 转换为文档文件中的规范化元素名称。

description

对引发异常的情况的描述。

示例：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
public class DataBaseOperations
{
   /// <exception cref="MasterFileFormatCorruptException"></exception>
   /// <exception cref="MasterFileLockedOpenException"></exception>
   public static void ReadRecord(int flag) {
      if (flag == 1)
         throw new MasterFileFormatCorruptException();
      else if (flag == 2)
         throw new MasterFileLockedOpenException();
      // …
   }
}
```

A.2.5 <include>

此标记允许包含来自源代码文件外部的 XML 文档的信息。外部文件必须是格式良好的 XML 文档，还可以将 XPath 表达式应用于该文档来指定应包含该 XML 文档中的哪些 XML 文本。然后用从外部文档中选定的 XML 来替换 <include> 标记。

语法：

```
<include file="filename" path="xpath" />
```

其中

`file="filename"`

外部 XML 文件的文件名。该文件名是相对于包含 include 标记的文件进行解释的（确定其完整路径名）。

`path="xpath"`

XPath 表达式，用于选择外部 XML 文件中的某些 XML。

示例：

如果源代码包含了如下声明：

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { … }
```

并且外部文件“docs.xml”含有以下内容：

```
<?xml version="1.0"?>
<extradoc>
   <class name="IntList">
      <summary>
         Contains a list of integers.
      </summary>
   </class>
   <class name="StringList">
      <summary>
         Contains a list of integers.
      </summary>
   </class>
</extradoc>
```

则输出的文档将与源代码中包含以下内容时一样：

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { … }
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

A.2.6 <list> \t "See <list>" \b

此标记用于创建项列表或项表。它可以包含 <listheader> 块以定义表或定义列表的标头行。（定义表时，只需提供一项作为标题中的 *term*。）

列表中的每一项都用一个 <item> 块来指定。创建定义列表时，必须同时指定 *term* 和 *description*。但对于表、项目符号列表或编号列表，只需指定 *description*。

语法：

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
   …
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

其中

term

要定义的术语，其定义位于 *description* 中。

description

是项目符号列表或编号列表中的项，或者是 *term* 的定义。

示例：

```
public class MyClass
{
   /// <summary>Here is an example of a bulleted list:
   /// <list type="bullet">
   /// <item>
   /// <description>Item 1.</description>
   /// </item>
   /// <item>
   /// <description>Item 2.</description>
   /// </item>
   /// </list>
   /// </summary>
   public static void Main () {
      // ...
   }
}
```

A.2.7 <para> \t "See <para>" \b

此标记用于其他标记内，如 <summary>（第 A.2.11 节）或 <returns>（第 A.2.12 节），用于将结构添加到文本中。

语法：

```
<para>content</para>
```

其中

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

content

段落文本。

示例：

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trvial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
   // ...
}
```

A.2.8 <param> \t "See <param>" \b

该标记用于描述方法、构造函数或索引器的参数。

语法：

```
<param name="name">description</param>
```

其中

name

参数名。

description

参数的描述。

示例：

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

A.2.9 <paramref> \t "See <paramref>" \b

该标记表示某单词是一个参数。可以对生成文档文件进行处理，以便用某种独特的方法来设置该参数的格式。

语法：

```
<paramref name="name"/>
```

其中

name

参数名。

示例：

```
/// <summary>This constructor initializes the new Point to
///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
public Point(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

A.2.10 <permission> \t "See <permission>" \b

该标记用于将成员的安全性和可访问性记入文档。

语法：

```
<permission cref="member">description</permission>
```

其中

`cref="`*member*`"`

成员的名称。文档生成器检查给定的代码元素是否存在，并将*member*转换为文档文件中的规范化元素名称。

description

对成员的访问属性的说明。

示例：

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>
public static void Test() {
   // ...
}
```

A.2.11 <remark> \t "See <remarks>" \b

该标记用于指定类型的额外信息。（使用 <summary>（第 A.2.15 节）描述类型本身及类型的成员。）

语法：

```
<remark>description</remark>
```

其中

description

备注文本。

示例：

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remark>Uses polar coordinates</remark>
public class Point
{
   // ...
}
```

A.2.12 <returns> \t "See <returns>" \b

该标记用于描述方法的返回值。

语法：

```
<returns>description</returns>
```

其中

description

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

返回值的说明。

示例：

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///   without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
   return "(" + X + "," + Y + ")";
}
```

A.2.13 <see> \t "See <see>" \b

此标记用于在文本内指定链接。（使用 <seealso>（第 A.2.14 节）指定将在“请参见”部分中出现的文本。）

语法：

```
<see cref="member"/>
```

其中

`cref="member"`

成员的名称。文档生成器检查给定的代码元素是否存在，并将 *member* 更改为所生成的文档文件中的元素名称。

示例：

```
/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}

/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

A.2.14 <seealso> \t "See <seealso>" \b

该标记用于生成将列入“请参见”部分的项。（使用 <see>（第 A.2.13 节）可指定文本中的链接。）

语法：

```
<seealso cref="member"/>
```

其中

`cref="member"`

成员的名称。文档生成器检查给定的代码元素是否存在，并将 *member* 更改为所生成的文档文件中的元素名称。

示例：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
/// <summary>This method determines whether two Points have the same
///   location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
   // ...
}
```

A.2.15 <summary> \t "See <summary>" \b

可以用此标记描述类型或类型的成员。使用 <remark>（第 A.2.11 节）描述类型本身。

语法：

```
<summary>description</summary>
```

其中

description

类型或成员的摘要。

示例：

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

A.2.16 <value> \t "See <value>" \b

该标记用于描述属性。

语法：

```
<value>property description</value>
```

其中

property description

属性的说明。

示例：

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
   get { return x; }
   set { x = value; }
}
```

A.2.17 <typeparam>

该标记用于描述类、结构、接口、委托或方法的泛型类型参数。

语法：

```
<typeparam name="name">description</typeparam>
```

其中

name

类型参数名。

description

类型参数的描述。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

示例：

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

A.2.18 <typeparamref>

该标记表示某单词是一个类型参数。这样，生成文档文件后经适当处理，可以用某种独特的方法来格式化该类型参数。

语法：

```
<typeparamref name="name"/>
```

其中

name

类型参数名。

示例：

```
/// <summary>This method fetches data and returns a list of <typeparamref
name=”T”> ”/>”> .</summary>
/// <param name="string">query to execute</param>

public List<T> FetchData<T>(string query) {
   ...
}
```

A.3 处理文档文件

文档生成器会为源代码中标记有文档注释的每个元素生成一个ID 字符串。该ID 字符串唯一地标识源元素。文档查看器利用此ID 字符串来标识该文档所描述的对应的元数据/反射项。

文档文件不是源代码的层次化表现形式；而是为每个元素生成的ID 字符串的一维列表。

A.3.1 ID 字符串格式

文档生成器在生成ID 字符串时遵循下列规则：

- 不在字符串中放置空白。
- 字符串的第一部分通过单个字符后跟一个冒号来标识所记录的成员的种类。定义以下几种成员：

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

字符	说明
E	事件
F	字段
M	方法（包括构造函数、析构函数和运算符）
N	命名空间
P	属性（包括索引器）
T	类型（如类、委托、枚举、接口和结构）
!	错误字符串；字符串的其他部分提供有关错误的信息。例如，文档生成器对无法解析的链接生成错误信息。

- 字符串的第二部分是元素的完全限定名，从命名空间的根开始。元素的名称、其封闭类型和命名空间都以句点分隔。如果项名本身含有句点，则将用 # (U+0023) 字符替换。（假定任何元素的名称内都不包含此字符。）
- 对于带有参数的方法和属性，接下来是用括号括起来的参数列表。对于那些不带参数的方法和属性，则省略括号。多个参数以逗号分隔。每个参数的编码都与 CLI 签名相同，如下所示：
  - 参数由其基于完全限定名的文档名称来表示，并做如下修改：

    表示泛型类型的实参附加了一个“’”字符，后接类型形参个数

    具有 `out` 或 `ref` 修饰符的参数在其类型名后接有 `@` 符。对于由值传递或通过 `params` 传递的参数没有特殊表示法。

    数组参数表示为 [*lowerbound* : *size* , … , *lowerbound* : *size*]，其中逗号数量等于秩减去一，而下限和每个维的大小（如果已知）用十进制数表示。如果未指定下限或大小，则将其省略。如果省略了某个特定维的下限及大小，则“:”也将被省略。交错数组由每个级别一个“[]”来表示。

    指针类型为非 void 的参数用类型名后面跟一个 * 的形式来表示。void 指针用类型名 `System.Void` 表示。

    引用在类型上定义的泛型类型形参的实参使用“`”字符进行编码，后接类型形参从零开始的索引。

    引用在方法中定义的泛型类型形参的实参使用双反引号“``”，而不使用用于类型的“`”。

    引用构造泛型类型的参数使用该泛型类型进行编码，后面依次跟“{”、逗号分隔的类型参数列表以及“}”。

A.3.2 ID 字符串示例

下列各个示例分别演示一段 C# 代码以及为每个可以含有文档注释的源元素生成的 ID 字符串：

- 类型用它们的完全限定名来表示，并使用泛型信息进行扩充：

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

namespace Acme
{
   interface IProcess {...}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
   struct ValueType {...}
   class Widget: IProcess
   {
      public class NestedClass {...}
      public interface IMenuItem {...}
      public delegate void Del(int i);
      public enum Direction { North, South, East, West }
   }
   class MyList<T>
   {
      class Helper<U,V> {...}
   }
}
"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
”T:Acme.MyList`1”
”T:Acme.MyList`1.Helper`2”
```

- 字段用它们的完全限定名来表示：

```
namespace Acme
{
   struct ValueType
   {
      private int total;
   }
   class Widget: IProcess
   {
      public class NestedClass
      {
         private int value;
      }
      private string message;
      private static Color defaultColor;
      private const double PI = 3.14159;
      protected readonly double monthlyAverage;
      private long[] array1;
      private Widget[,] array2;
      private unsafe int *pCount;
      private unsafe float **ppValues;
   }
}
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

- 构造函数。

```
namespace Acme
{
   class Widget: IProcess
   {
      static Widget() {...}

      public Widget() {...}

      public Widget(string s) {...}
   }
}

"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- 析构函数。

```
namespace Acme
{
   class Widget: IProcess
   {
      ~Widget() {...}
   }
}

"M:Acme.Widget.Finalize"
```

- 方法。

```
namespace Acme
{
   struct ValueType
   {
      public void M(int i) {...}
   }

   class Widget: IProcess
   {
      public class NestedClass
      {
         public void M(int i) {...}
      }

      public static void M0() {...}
      public void M1(char c, out float f, ref ValueType v) {...}
      public void M2(short[] x1, int[,] x2, long[][] x3) {...}
      public void M3(long[][] x3, Widget[][,,] x4) {...}
      public unsafe void M4(char *pc, Color **pf) {...}
      public unsafe void M5(void *pv, double *[][,] pd) {...}
      public void M6(int i, params object[] args) {...}
   }

   class MyList<T>
   {
      public void Test(T t) { }
   }

   class UseList
   {
      public void Process(MyList<int> list) { }
      public MyList<T> GetValues<T>(T inputValue) { return null; }
   }
}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
”M:Acme.MyList`1.Test(`0)”
”M:Acme.UseList.Process(Acme.MyList{System.Int32})”
”M:Acme.UseList.GetValues``(``0)”
```

- 属性和索引器。

```
namespace Acme
{
   class Widget: IProcess
   {
      public int Width { get {...} set {...} }
      public int this[int i] { get {...} set {...} }
      public int this[string s, int i] { get {...} set {...} }
   }
}
```

```
"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- 事件。

```
namespace Acme
{
   class Widget: IProcess
   {
      public event Del AnEvent;
   }
}
```

```
"E:Acme.Widget.AnEvent"
```

- 一元运算符。

```
namespace Acme
{
   class Widget: IProcess
   {
      public static Widget operator+(Widget x) {...}
   }
}
```

```
"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

所使用的一元运算符函数名称的完整集合如下：op_UnaryPlus、op_UnaryNegation、op_LogicalNot、op_OnesComplement、op_Increment、op_Decrement、op_True 和 op_False。

- 二元运算符。

```
namespace Acme
{
   class Widget: IProcess
   {
      public static Widget operator+(Widget x1, Widget x2) {...}
   }
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

所使用的二元运算符函数名称的完整集合如下：op_Addition、op_Subtraction、op_Multiply、op_Division、op_Modulus、op_BitwiseAnd、op_BitwiseOr、op_ExclusiveOr、op_LeftShift、op_RightShift、op_Equality、op_Inequality、op_LessThan、op_LessThanOrEqual、op_GreaterThan 和 op_GreaterThanOrEqual。

- 转换运算符具有一个尾随“~”，然后再跟返回类型。

```
namespace Acme
{
   class Widget: IProcess
   {
      public static explicit operator int(Widget x) {...}
      public static implicit operator long(Widget x) {...}
   }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

A.4 示例

A.4.1 C# 源代码

下面的示例演示一个 Point 类的源代码：

```
namespace Graphics
{

/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
/// </summary>
public class Point
{

   /// <summary>Instance variable <c>x</c> represents the point's
   ///   x-coordinate.</summary>
   private int x;

   /// <summary>Instance variable <c>y</c> represents the point's
   ///   y-coordinate.</summary>
   private int y;

   /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
   public int X
   {
      get { return x; }
      set { x = value; }
   }

   /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
   public int Y
   {
      get { return y; }
      set { y = value; }
   }

   /// <summary>This constructor initializes the new Point to
   ///   (0,0).</summary>
   public Point() : this(0,0) {}
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
/// <summary>This constructor initializes the new Point to
///   (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param><c>xor</c> is the new Point's x-coordinate.</param>
/// <param><c>yor</c> is the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
   X = xor;
   Y = yor;
}

/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <param><c>xor</c> is the new x-coordinate.</param>
/// <param><c>yor</c> is the new y-coordinate.</param>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}

/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// <example>For example:
/// <code>
///   Point p = new Point(3,5);
///   p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
/// <param><c>xor</c> is the relative x-offset.</param>
/// <param><c>yor</c> is the relative y-offset.</param>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}

/// <summary>This method determines whether two Points have the same
///   location.</summary>
/// <param><c>o</c> is the object to be compared to the current object.
/// </param>
/// <returns>True if the Points have the same location and they have
///   the exact same type; otherwise, false.</returns>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
   if (o == null) {
      return false;
   }

   if (this == o) {
      return true;
   }

   if (GetType() == o.GetType()) {
      Point p = (Point)o;
      return (X == p.X) && (Y == p.Y);
   }
   return false;
}

/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///   without any leading, training, or embedded whitespace.</returns>
public override string ToString() {
   return "(" + X + "," + Y + ")";
}
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        /// <summary>This operator determines whether two Points have the same
        ///   location.</summary>
        /// <param><c>p1</c> is the first Point to be compared.</param>
        /// <param><c>p2</c> is the second Point to be compared.</param>
        /// <returns>True if the Points have the same location and they have
        ///   the exact same type; otherwise, false.</returns>
        /// <seealso cref="Equals"/>
        /// <seealso cref="operator!="/>
        public static bool operator==(Point p1, Point p2) {
           if ((object)p1 == null || (object)p2 == null) {
              return false;
           }

           if (p1.GetType() == p2.GetType()) {
              return (p1.X == p2.X) && (p1.Y == p2.Y);
           }

           return false;
        }

        /// <summary>This operator determines whether two Points have the same
        ///   location.</summary>
        /// <param><c>p1</c> is the first Point to be compared.</param>
        /// <param><c>p2</c> is the second Point to be compared.</param>
        /// <returns>True if the Points do not have the same location and the
        ///   exact same type; otherwise, false.</returns>
        /// <seealso cref="Equals"/>
        /// <seealso cref="operator=="/>
        public static bool operator!=(Point p1, Point p2) {
           return !(p1 == p2);
        }

        /// <summary>This is the entry point of the Point class testing
        /// program.
        /// <para>This program tests each method and operator, and
        /// is intended to be run after any non-trvial maintenance has
        /// been performed on the Point class.</para></summary>
        public static void Main() {
           // class test code goes here
        }
     }
     }
```

A.4.2 生成的 XML

以下是文档生成器根据给定类 Point 的源代码(如上所示)所产生的输出：

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-dimensional
            plane.
            </summary>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>

        <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
            <summary>This method changes the point's location to
            the given coordinates.</summary>
            <param><c>xor</c> is the new x-coordinate.</param>
            <param><c>yor</c> is the new y-coordinate.</param>
            <see
cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
        </member>

        <member
            name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
            <summary>This method changes the point's location by
            the given x- and y-offsets.
            <example>For example:
            <code>
            Point p = new Point(3,5);
            p.Translate(-1,3);
            </code>
            results in <c>p</c>'s having the value (2,8).
            </example>
            </summary>
            <param><c>xor</c> is the relative x-offset.</param>
            <param><c>yor</c> is the relative y-offset.</param>
            <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
        </member>

        <member name="M:Graphics.Point.Equals(System.Object)">
            <summary>This method determines whether two Points have the same
            location.</summary>
            <param><c>o</c> is the object to be compared to the current
            object.
            </param>
            <returns>True if the Points have the same location and they have
            the exact same type; otherwise, false.</returns>
            <seealso
      cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
            <seealso
      cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
        </member>

        <member name="M:Graphics.Point.ToString">
            <summary>Report a point's location as a string.</summary>
            <returns>A string representing a point's location, in the form
            (x,y),
            without any leading, training, or embedded whitespace.</returns>
        </member>
```

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

```
            <member
             name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
                 <summary>This operator determines whether two Points have the
                 same
                 location.</summary>
                 <param><c>p1</c> is the first Point to be compared.</param>
                 <param><c>p2</c> is the second Point to be compared.</param>
                 <returns>True if the Points have the same location and they have
                 the exact same type; otherwise, false.</returns>
                 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
                 <seealso
           cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
            </member>

            <member
             name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
                 <summary>This operator determines whether two Points have the
                 same
                 location.</summary>
                 <param><c>p1</c> is the first Point to be compared.</param>
                 <param><c>p2</c> is the second Point to be compared.</param>
                 <returns>True if the Points do not have the same location and
                 the
                 exact same type; otherwise, false.</returns>
                 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
                 <seealso
           cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
            </member>

            <member name="M:Graphics.Point.Main">
                 <summary>This is the entry point of the Point class testing
                 program.
                 <para>This program tests each method and operator, and
                 is intended to be run after any non-trvial maintenance has
                 been performed on the Point class.</para></summary>
            </member>

            <member name="P:Graphics.Point.X">
                 <value>Property <c>X</c> represents the point's
                 x-coordinate.</value>
            </member>

            <member name="P:Graphics.Point.Y">
                 <value>Property <c>Y</c> represents the point's
                 y-coordinate.</value>
            </member>
        </members>
    </doc>
```


版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

B. 语法

此附录是主文档中描述的词法和语法以及不安全代码的语法扩展的摘要。这里，各语法产生式是按它们在主文档中出现的顺序列出的。

B.1 词法文法

input:
    input-section$_{opt}$

input-section:
    input-section-part
    input-section input-section-part

input-section-part:
    input-elements$_{opt}$ *new-line*
    pp-directive

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

input-element:
    whitespace
    comment
    token

B.1.1 行结束符

new-line:
    Carriage return character (U+000D)
    Line feed character (U+000A)
    Carriage return character (U+000D) followed by line feed character (U+000A)
    Next line character (U+0085)
    Line separator character (U+2028)
    Paragraph separator character (U+2029)

B.1.2 注释

comment:
    single-line-comment
    delimited-comment

single-line-comment:
    // *input-characters*$_{opt}$

input-characters:
    input-character
    input-characters input-character

input-character:
    Any Unicode character except a *new-line-character*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

new-line-character:
Carriage return character (U+000D)
Line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

delimited-comment:
/* *delimited-comment-text*$_{opt}$ *asterisks* /

delimited-comment-text:
delimited-comment-section
delimited-comment-text *delimited-comment-section*

delimited-comment-section:
/
asterisks$_{opt}$ *not-slash-or-asterisk*

asterisks:
*
asterisks *

not-slash-or-asterisk:
Any Unicode character except / or *

B.1.3 空白

whitespace:
Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

B.1.4 标记

token:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

B.1.5 Unicode 字符转义序列

unicode-escape-sequence:
\u *hex-digit* *hex-digit* *hex-digit* *hex-digit*
\U *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit*

B.1.6 标识符

identifier:
available-identifier
@ *identifier-or-keyword*

available-identifier:
An *identifier-or-keyword* that is not a *keyword*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

identifier-or-keyword:
identifier-start-character *identifier-part-characters*$_{opt}$

identifier-start-character:
letter-character
_ (the underscore character U+005F)

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

identifier-part-character:
letter-character
decimal-digit-character
connecting-character
combining-character
formatting-character

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

combining-character:
A Unicode character of classes Mn or Mc
A *unicode-escape-sequence* representing a character of classes Mn or Mc

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

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

formatting-character:
A Unicode character of the class Cf
A *unicode-escape-sequence* representing a character of the class Cf

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

B.1.7 关键字

keyword: one of

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

B.1.8 文本

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

boolean-literal:
`true`
`false`

integer-literal:
decimal-integer-literal
hexadecimal-integer-literal

decimal-integer-literal:
decimal-digits *integer-type-suffix*$_{opt}$

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*$_{opt}$
`0X` *hex-digits* *integer-type-suffix*$_{opt}$

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

real-literal:
    decimal-digits . *decimal-digits exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
    . *decimal-digits exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
    decimal-digits exponent-part real-type-suffix$_{opt}$
    decimal-digits real-type-suffix

exponent-part:
    e *sign*$_{opt}$ *decimal-digits*
    E *sign*$_{opt}$ *decimal-digits*

sign: one of
    + -

real-type-suffix: one of
    F f D d M m

character-literal:
    ' *character* '

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

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

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

hexadecimal-escape-sequence:
    \x *hex-digit hex-digit*$_{opt}$ *hex-digit*$_{opt}$ *hex-digit*$_{opt}$

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

regular-string-literal:
    " *regular-string-literal-characters*$_{opt}$ "

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-escape-sequence

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

verbatim-string-literal:
@" *verbatim-string-literal-characters*$_{opt}$ "

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:
""

null-literal:
null

B.1.9 运算符和标点符号

operator-or-punctuator: one of

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

right-shift:
>|>

right-shift-assignment:
>|>=

B.1.10 预处理指令

pp-directive:
pp-declaration
pp-conditional
pp-line
pp-diagnostic
pp-region
pp-pragma

conditional-symbol:
Any *identifier-or-keyword* except `true` or `false`

pp-expression:
whitespace$_{opt}$ *pp-or-expression whitespace*$_{opt}$

pp-or-expression:
pp-and-expression
pp-or-expression whitespace$_{opt}$ || *whitespace*$_{opt}$ *pp-and-expression*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

pp-and-expression:
    pp-equality-expression
    pp-and-expression *whitespace*$_{opt}$ `&&` *whitespace*$_{opt}$ *pp-equality-expression*

pp-equality-expression:
    pp-unary-expression
    pp-equality-expression *whitespace*$_{opt}$ `==` *whitespace*$_{opt}$ *pp-unary-expression*
    pp-equality-expression *whitespace*$_{opt}$ `!=` *whitespace*$_{opt}$ *pp-unary-expression*

pp-unary-expression:
    pp-primary-expression
    `!` *whitespace*$_{opt}$ *pp-unary-expression*

pp-primary-expression:
    `true`
    `false`
    conditional-symbol
    `(` *whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$ `)`

pp-declaration:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `define` *whitespace* *conditional-symbol* *pp-new-line*
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `undef` *whitespace* *conditional-symbol* *pp-new-line*

pp-new-line:
    whitespace$_{opt}$ *single-line-comment*$_{opt}$ *new-line*

pp-conditional:
    pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*

pp-if-section:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `if` *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-elif-sections:
    pp-elif-section
    pp-elif-sections *pp-elif-section*

pp-elif-section:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `elif` *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-else-section:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `else` *pp-new-line* *conditional-section*$_{opt}$

pp-endif:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endif` *pp-new-line*

conditional-section:
    input-section
    skipped-section

skipped-section:
    skipped-section-part
    skipped-section *skipped-section-part*

skipped-section-part:
    skipped-characters$_{opt}$ *new-line*
    pp-directive

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

skipped-characters:
    whitespace$_{opt}$ *not-number-sign* *input-characters*$_{opt}$

not-number-sign:
    Any *input-character* except #

pp-diagnostic:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `error` *pp-message*
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `warning` *pp-message*

pp-message:
    new-line
    whitespace *input-characters*$_{opt}$ *new-line*

pp-region:
    pp-start-region *conditional-section*$_{opt}$ *pp-end-region*

pp-start-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `region` *pp-message*

pp-end-region:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `endregion` *pp-message*

pp-line:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `line` *whitespace* *line-indicator* *pp-new-line*

line-indicator:
    decimal-digits *whitespace* *file-name*
    decimal-digits
    `default`
    `hidden`

file-name:
    `"` *file-name-characters* `"`

file-name-characters:
    file-name-character
    file-name-characters *file-name-character*

file-name-character:
    Any *input-character* except "

pp-pragma:
    whitespace$_{opt}$ `#` *whitespace*$_{opt}$ `pragma` *whitespace* *pragma-body* *pp-new-line*

pragma-body:
    pragma-warning-body

pragma-warning-body:
    `warning` *whitespace* *warning-action*
    `warning` *whitespace* *warning-action* *whitespace* *warning-list*

warning-action:
    `disable`
    `restore`

warning-list:
    decimal-digits
    warning-list *whitespace*$_{opt}$ `,` *whitespace*$_{opt}$ *decimal-digits*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

B.2 句法文法

B.2.1 基本概念

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier type-argument-list$_{opt}$
    namespace-or-type-name `.` *identifier type-argument-list*$_{opt}$
    qualified-alias-member

B.2.2 类型

type:
    value-type
    reference-type
    type-parameter

value-type:
    struct-type
    enum-type

struct-type:
    type-name
    simple-type
    nullable-type

simple-type:
    numeric-type
    `bool`

numeric-type:
    integral-type
    floating-point-type
    `decimal`

integral-type:
    `sbyte`
    `byte`
    `short`
    `ushort`
    `int`
    `uint`
    `long`
    `ulong`
    `char`

floating-point-type:
    `float`
    `double`

nullable-type:
    non-nullable-value-type `?`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

non-nullable-value-type:
    type

enum-type:
    type-name

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

class-type:
    type-name
    `object`
    `dynamic`
    `string`

interface-type:
    type-name

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

rank-specifier:
    `[` *dim-separators*$_{opt}$ `]`

dim-separators:
    `,`
    dim-separators `,`

delegate-type:
    type-name

type-argument-list:
    `<` *type-arguments* `>`

type-arguments:
    type-argument
    type-arguments `,` *type-argument*

type-argument:
    type

type-parameter:
    identifier

B.2.3 变量

variable-reference:
    expression

B.2.4 表达式

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

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

argument:
    argument-name$_{opt}$ *argument-value*

argument-name:
    identifier :

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

primary-expression:
    primary-no-array-creation-expression
    array-creation-expression

primary-no-array-creation-expression:
    literal
    simple-name
    parenthesized-expression
    member-access
    invocation-expression
    element-access
    this-access
    base-access
    post-increment-expression
    post-decrement-expression
    object-creation-expression
    delegate-creation-expression
    anonymous-object-creation-expression
    typeof-expression
    checked-expression
    unchecked-expression
    default-value-expression
    anonymous-method-expression

simple-name:
    identifier *type-argument-list*$_{opt}$

parenthesized-expression:
    (*expression*)

member-access:
    primary-expression . *identifier* *type-argument-list*$_{opt}$
    predefined-type . *identifier* *type-argument-list*$_{opt}$
    qualified-alias-member . *identifier*

predefined-type: one of
    `bool` `byte` `char` `decimal` `double` `float` `int` `long`
    `object` `sbyte` `short` `string` `uint` `ulong` `ushort`

invocation-expression:
    primary-expression (*argument-list*$_{opt}$)

element-access:
    primary-no-array-creation-expression [*argument-list*]

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

this-access:
    this

base-access:
    base . *identifier*
    base [*argument-list*]

post-increment-expression:
    primary-expression ++

post-decrement-expression:
    primary-expression --

object-creation-expression:
    new *type* (*argument-list*$_{opt}$) *object-or-collection-initializer*$_{opt}$
    new *type* *object-or-collection-initializer*

object-or-collection-initializer:
    object-initializer
    collection-initializer

object-initializer:
    { *member-initializer-list*$_{opt}$ }
    { *member-initializer-list* , }

member-initializer-list:
    member-initializer
    member-initializer-list , *member-initializer*

member-initializer:
    identifier = *initializer-value*

initializer-value:
    expression
    object-or-collection-initializer

collection-initializer:
    { *element-initializer-list* }
    { *element-initializer-list* , }

element-initializer-list:
    element-initializer
    element-initializer-list , *element-initializer*

element-initializer:
    non-assignment-expression
    { *expression-list* }

expression-list:
    expression
    expression-list , *expression*

array-creation-expression:
    new *non-array-type* [*expression-list*] *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
    new *array-type* *array-initializer*
    new *rank-specifier* *array-initializer*

delegate-creation-expression:
    new *delegate-type* (*expression*)

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

anonymous-object-creation-expression:
    `new` *anonymous-object-initializer*

anonymous-object-initializer:
    `{` *member-declarator-list*$_{opt}$ `}`
    `{` *member-declarator-list* `,` `}`

member-declarator-list:
    member-declarator
    member-declarator-list `,` *member-declarator*

member-declarator:
    simple-name
    member-access
    identifier `=` *expression*

typeof-expression:
    `typeof` `(` *type* `)`
    `typeof` `(` *unbound-type-name* `)`
    `typeof` `(` `void` `)`

unbound-type-name:
    identifier *generic-dimension-specifier*$_{opt}$
    identifier `::` *identifier* *generic-dimension-specifier*$_{opt}$
    unbound-type-name `.` *identifier* *generic-dimension-specifier*$_{opt}$

generic-dimension-specifier:
    `<` *commas*$_{opt}$ `>`

commas:
    `,`
    commas `,`

checked-expression:
    `checked` `(` *expression* `)`

unchecked-expression:
    `unchecked` `(` *expression* `)`

default-value-expression:
    `default` `(` *type* `)`

unary-expression:
    primary-expression
    `+` *unary-expression*
    `-` *unary-expression*
    `!` *unary-expression*
    `~` *unary-expression*
    pre-increment-expression
    pre-decrement-expression
    cast-expression

pre-increment-expression:
    `++` *unary-expression*

pre-decrement-expression:
    `--` *unary-expression*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

cast-expression:
    (*type*) *unary-expression*

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*

shift-expression:
    additive-expression
    shift-expression << *additive-expression*
    shift-expression *right-shift* *additive-expression*

relational-expression:
    shift-expression
    relational-expression < *shift-expression*
    relational-expression > *shift-expression*
    relational-expression <= *shift-expression*
    relational-expression >= *shift-expression*
    relational-expression is *type*
    relational-expression as *type*

equality-expression:
    relational-expression
    equality-expression == *relational-expression*
    equality-expression != *relational-expression*

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*

conditional-and-expression:
    inclusive-or-expression
    conditional-and-expression && *inclusive-or-expression*

conditional-or-expression:
    conditional-and-expression
    conditional-or-expression || *conditional-and-expression*

null-coalescing-expression:
    conditional-or-expression
    conditional-or-expression ?? *null-coalescing-expression*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

conditional-expression:
    null-coalescing-expression
    null-coalescing-expression `?` *expression* `:` *expression*

lambda-expression:
    anonymous-function-signature `=>` *anonymous-function-body*

anonymous-method-expression:
    `delegate` *explicit-anonymous-function-signature*$_{opt}$ *block*

anonymous-function-signature:
    explicit-anonymous-function-signature
    implicit-anonymous-function-signature

explicit-anonymous-function-signature:
    `(` *explicit-anonymous-function-parameter-list*$_{opt}$ `)`

explicit-anonymous-function-parameter-list:
    explicit-anonymous-function-parameter
    explicit-anonymous-function-parameter-list `,` *explicit-anonymous-function-parameter*

explicit-anonymous-function-parameter:
    anonymous-function-parameter-modifier$_{opt}$ *type* *identifier*

anonymous-function-parameter-modifier:
    `ref`
    `out`

implicit-anonymous-function-signature:
    `(` *implicit-anonymous-function-parameter-list*$_{opt}$ `)`
    implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:
    implicit-anonymous-function-parameter
    implicit-anonymous-function-parameter-list `,` *implicit-anonymous-function-parameter*

implicit-anonymous-function-parameter:
    identifier

anonymous-function-body:
    expression
    block

query-expression:
    from-clause *query-body*

from-clause:
    `from` *type*$_{opt}$ *identifier* `in` *expression*

query-body:
    query-body-clauses$_{opt}$ *select-or-group-clause* *query-continuation*$_{opt}$

query-body-clauses:
    query-body-clause
    query-body-clauses *query-body-clause*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

query-body-clause:
    from-clause
    let-clause
    where-clause
    join-clause
    join-into-clause
    orderby-clause

let-clause:
    `let` *identifier* `=` *expression*

where-clause:
    `where` *boolean-expression*

join-clause:
    `join` $type_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression*

join-into-clause:
    `join` $type_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression* `into` *identifier*

orderby-clause:
    `orderby` *orderings*

orderings:
    ordering
    orderings `,` *ordering*

ordering:
    expression $ordering\text{-}direction_{opt}$

ordering-direction:
    `ascending`
    `descending`

select-or-group-clause:
    select-clause
    group-clause

select-clause:
    `select` *expression*

group-clause:
    `group` *expression* `by` *expression*

query-continuation:
    `into` *identifier* *query-body*

assignment:
    unary-expression *assignment-operator* *expression*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

assignment-operator:
    =
    +=
    -=
    *=
    /=
    %=
    &=
    |=
    ^=
    <<=
    right-shift-assignment

expression:
    non-assignment-expression
    assignment

non-assignment-expression:
    conditional-expression
    lambda-expression
    query-expression

constant-expression:
    expression

boolean-expression:
    expression

B.2.5 语句

statement:
    labeled-statement
    declaration-statement
    embedded-statement

embedded-statement:
    block
    empty-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    try-statement
    checked-statement
    unchecked-statement
    lock-statement
    using-statement
    yield-statement

block:
    { *statement-list*$_{opt}$ }

statement-list:
    statement
    statement-list statement

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

empty-statement:
    ;

labeled-statement:
    identifier : *statement*

declaration-statement:
    local-variable-declaration ;
    local-constant-declaration ;

local-variable-declaration:
    local-variable-type *local-variable-declarators*

local-variable-type:
    type
    var

local-variable-declarators:
    local-variable-declarator
    local-variable-declarators , *local-variable-declarator*

local-variable-declarator:
    identifier
    identifier = *local-variable-initializer*

local-variable-initializer:
    expression
    array-initializer

local-constant-declaration:
    const *type* *constant-declarators*

constant-declarators:
    constant-declarator
    constant-declarators , *constant-declarator*

constant-declarator:
    identifier = *constant-expression*

expression-statement:
    statement-expression ;

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression

selection-statement:
    if-statement
    switch-statement

if-statement:
    if (*boolean-expression*) *embedded-statement*
    if (*boolean-expression*) *embedded-statement* else *embedded-statement*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

switch-statement:
    `switch` `(` *expression* `)` *switch-block*

switch-block:
    `{` *switch-sections*$_{opt}$ `}`

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` `:`

iteration-statement:
    while-statement
    do-statement
    for-statement
    foreach-statement

while-statement:
    `while` `(` *boolean-expression* `)` *embedded-statement*

do-statement:
    `do` *embedded-statement* `while` `(` *boolean-expression* `)` `;`

for-statement:
    `for` `(` *for-initializer*$_{opt}$ `;` *for-condition*$_{opt}$ `;` *for-iterator*$_{opt}$ `)` *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*

foreach-statement:
    `foreach` `(` *local-variable-type* *identifier* `in` *expression* `)` *embedded-statement*

jump-statement:
    break-statement
    continue-statement
    goto-statement
    return-statement
    throw-statement

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

break-statement:
    `break` `;`

continue-statement:
    `continue` `;`

goto-statement:
    `goto` *identifier* `;`
    `goto case` *constant-expression* `;`
    `goto default` `;`

return-statement:
    `return` *expression*$_{opt}$ `;`

throw-statement:
    `throw` *expression*$_{opt}$ `;`

try-statement:
    `try` *block catch-clauses*
    `try` *block finally-clause*
    `try` *block catch-clauses finally-clause*

catch-clauses:
    specific-catch-clauses general-catch-clause$_{opt}$
    specific-catch-clauses$_{opt}$ *general-catch-clause*

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause

specific-catch-clause:
    `catch` `(` *class-type identifier*$_{opt}$ `)` *block*

general-catch-clause:
    `catch` *block*

finally-clause:
    `finally` *block*

checked-statement:
    `checked` *block*

unchecked-statement:
    `unchecked` *block*

lock-statement:
    `lock` `(` *expression* `)` *embedded-statement*

using-statement:
    `using` `(` *resource-acquisition* `)` *embedded-statement*

resource-acquisition:
    local-variable-declaration
    expression

yield-statement:
    `yield return` *expression* `;`
    `yield break` `;`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

B.2.6 命名空间

compilation-unit:
    extern-alias-directives$_{opt}$ *using-directives*$_{opt}$ *global-attributes*$_{opt}$
        namespace-member-declarations$_{opt}$

namespace-declaration:
    `namespace` *qualified-identifier* *namespace-body* `;`$_{opt}$

qualified-identifier:
    identifier
    qualified-identifier `.` *identifier*

namespace-body:
    `{` *extern-alias-directives*$_{opt}$ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ `}`

extern-alias-directives:
    extern-alias-directive
    extern-alias-directives *extern-alias-directive*

extern-alias-directive:
    `extern alias` *identifier* `;`

using-directives:
    using-directive
    using-directives *using-directive*

using-directive:
    using-alias-directive
    using-namespace-directive

using-alias-directive:
    `using` *identifier* `=` *namespace-or-type-name* `;`

using-namespace-directive:
    `using` *namespace-name* `;`

namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations *namespace-member-declaration*

namespace-member-declaration:
    namespace-declaration
    type-declaration

type-declaration:
    class-declaration
    struct-declaration
    interface-declaration
    enum-declaration
    delegate-declaration

qualified-alias-member:
    identifier `::` *identifier* *type-argument-list*$_{opt}$

B.2.7 类

class-declaration:
    attributes$_{opt}$ *class-modifiers*$_{opt}$ `partial`$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$
        class-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *class-body* `;`$_{opt}$

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

class-modifiers:
    class-modifier
    class-modifiers *class-modifier*

class-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `abstract`
    `sealed`
    `static`

type-parameter-list:
    `<` *type-parameters* `>`

type-parameters:
    $attributes_{opt}$ *type-parameter*
    type-parameters `,` $attributes_{opt}$ *type-parameter*

type-parameter:
    identifier

class-base:
    `:` *class-type*
    `:` *interface-type-list*
    `:` *class-type* `,` *interface-type-list*

interface-type-list:
    interface-type
    interface-type-list `,` *interface-type*

type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
    `where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
    primary-constraint
    secondary-constraints
    constructor-constraint
    primary-constraint `,` *secondary-constraints*
    primary-constraint `,` *constructor-constraint*
    secondary-constraints `,` *constructor-constraint*
    primary-constraint `,` *secondary-constraints* `,` *constructor-constraint*

primary-constraint:
    class-type
    `class`
    `struct`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

secondary-constraints:
    interface-type
    type-parameter
    secondary-constraints , *interface-type*
    secondary-constraints , *type-parameter*

constructor-constraint:
    new ( )

class-body:
    { *class-member-declarations*$_{opt}$ }

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

constant-declaration:
    attributes$_{opt}$ *constant-modifiers*$_{opt}$ 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*

field-declaration:
    attributes$_{opt}$ *field-modifiers*$_{opt}$ *type variable-declarators* ;

field-modifiers:
    field-modifier
    field-modifiers field-modifier

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

field-modifier:
    new
    public
    protected
    internal
    private
    static
    readonly
    volatile

variable-declarators:
    variable-declarator
    variable-declarators , *variable-declarator*

variable-declarator:
    identifier
    identifier = *variable-initializer*

variable-initializer:
    expression
    array-initializer

method-declaration:
    method-header *method-body*

method-header:
    $attributes_{opt}$ $method\text{-}modifiers_{opt}$ $\texttt{partial}_{opt}$ *return-type* *member-name* $type\text{-}parameter\text{-}list_{opt}$
        ($formal\text{-}parameter\text{-}list_{opt}$) $type\text{-}parameter\text{-}constraints\text{-}clauses_{opt}$

method-modifiers:
    method-modifier
    method-modifiers *method-modifier*

method-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern

return-type:
    type
    void

member-name:
    identifier
    interface-type . *identifier*

method-body:
    block
    ;

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

formal-parameter-list:
    fixed-parameters
    fixed-parameters , *parameter-array*
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters , *fixed-parameter*

fixed-parameter:
    attributes$_{opt}$ *parameter-modifier*$_{opt}$ *type* *identifier* *default-argument*$_{opt}$

default-argument:
    = *expression*

parameter-modifier:
    ref
    out
    this

parameter-array:
    attributes$_{opt}$ params *array-type* *identifier*

property-declaration:
    attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* { *accessor-declarations* }

property-modifiers:
    property-modifier
    property-modifiers *property-modifier*

property-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern

member-name:
    identifier
    interface-type . *identifier*

accessor-declarations:
    get-accessor-declaration *set-accessor-declaration*$_{opt}$
    set-accessor-declaration *get-accessor-declaration*$_{opt}$

get-accessor-declaration:
    attributes$_{opt}$ *accessor-modifier*$_{opt}$ get *accessor-body*

set-accessor-declaration:
    attributes$_{opt}$ *accessor-modifier*$_{opt}$ set *accessor-body*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

accessor-modifier:
    protected
    internal
    private
    protected internal
    internal protected

accessor-body:
    block
    ;

event-declaration:
    attributes$_{opt}$ *event-modifiers*$_{opt}$ event *type variable-declarators* ;
    attributes$_{opt}$ *event-modifiers*$_{opt}$ event *type member-name* { *event-accessor-declarations* }

event-modifiers:
    event-modifier
    event-modifiers event-modifier

event-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern

event-accessor-declarations:
    add-accessor-declaration remove-accessor-declaration
    remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:
    attributes$_{opt}$ add *block*

remove-accessor-declaration:
    attributes$_{opt}$ remove *block*

indexer-declaration:
    attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* { *accessor-declarations* }

indexer-modifiers:
    indexer-modifier
    indexer-modifiers indexer-modifier

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

indexer-modifier:
    new
    public
    protected
    internal
    private
    virtual
    sealed
    override
    abstract
    extern

indexer-declarator:
    type this [*formal-parameter-list*]
    type interface-type . this [*formal-parameter-list*]

operator-declaration:
    attributes$_{opt}$ *operator-modifiers operator-declarator operator-body*

operator-modifiers:
    operator-modifier
    operator-modifiers operator-modifier

operator-modifier:
    public
    static
    extern

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*)

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

overloadable-binary-operator:
+
-
*
/
%
&
|
^
<<
right-shift
==
!=
>
<
>=
<=

conversion-operator-declarator:
`implicit operator` *type* `(` *type identifier* `)`
`explicit operator` *type* `(` *type identifier* `)`

operator-body:
block
;

constructor-declaration:
attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator constructor-body*

constructor-modifiers:
constructor-modifier
constructor-modifiers constructor-modifier

constructor-modifier:
`public`
`protected`
`internal`
`private`
`extern`

constructor-declarator:
identifier `(` *formal-parameter-list*$_{opt}$ `)` *constructor-initializer*$_{opt}$

constructor-initializer:
`: base (` *argument-list*$_{opt}$ `)`
`: this (` *argument-list*$_{opt}$ `)`

constructor-body:
block
;

static-constructor-declaration:
attributes$_{opt}$ *static-constructor-modifiers identifier* `(` `)` *static-constructor-body*

static-constructor-modifiers:
`extern`$_{opt}$ `static`
`static extern`$_{opt}$

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

static-constructor-body:
    block
    ;

destructor-declaration:
    attributes$_{opt}$ `extern`$_{opt}$ `~` *identifier* `(` `)` *destructor-body*

destructor-body:
    block
    ;

B.2.8 结构

struct-declaration:
    attributes$_{opt}$ *struct-modifiers*$_{opt}$ `partial`$_{opt}$ `struct` *identifier* *type-parameter-list*$_{opt}$
        struct-interfaces$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *struct-body* `;`$_{opt}$

struct-modifiers:
    struct-modifier
    struct-modifiers *struct-modifier*

struct-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

struct-interfaces:
    `:` *interface-type-list*

struct-body:
    `{` *struct-member-declarations*$_{opt}$ `}`

struct-member-declarations:
    struct-member-declaration
    struct-member-declarations *struct-member-declaration*

struct-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration

B.2.9 数组

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

non-array-type:
    type

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

rank-specifier:
    `[` *dim-separators*$_{opt}$ `]`

dim-separators:
    `,`
    dim-separators `,`

array-initializer:
    `{` *variable-initializer-list*$_{opt}$ `}`
    `{` *variable-initializer-list* `,` `}`

variable-initializer-list:
    variable-initializer
    variable-initializer-list `,` *variable-initializer*

variable-initializer:
    expression
    array-initializer

B.2.10 接口

interface-declaration:
    attributes$_{opt}$ *interface-modifiers*$_{opt}$ `partial`$_{opt}$ `interface`
        identifier *variant-type-parameter-list*$_{opt}$ *interface-base*$_{opt}$
        type-parameter-constraints-clauses$_{opt}$ *interface-body* `;`$_{opt}$

interface-modifiers:
    interface-modifier
    interface-modifiers *interface-modifier*

interface-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`

variant-type-parameter-list:
    `<` *variant-type-parameters* `>`

variant-type-parameters:
    attributes$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*
    variant-type-parameters `,` *attributes*$_{opt}$ *variance-annotation*$_{opt}$ *type-parameter*

variance-annotation:
    `in`
    `out`

interface-base:
    `:` *interface-type-list*

interface-body:
    `{` *interface-member-declarations*$_{opt}$ `}`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

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

interface-method-declaration:
attributes$_{opt}$ `new`$_{opt}$ *return-type identifier type-parameter-list*
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

interface-property-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type identifier* `{` *interface-accessors* `}`

interface-accessors:
attributes$_{opt}$ `get ;`
attributes$_{opt}$ `set ;`
attributes$_{opt}$ `get ;` *attributes*$_{opt}$ `set ;`
attributes$_{opt}$ `set ;` *attributes*$_{opt}$ `get ;`

interface-event-declaration:
attributes$_{opt}$ `new`$_{opt}$ `event` *type identifier* `;`

interface-indexer-declaration:
attributes$_{opt}$ `new`$_{opt}$ *type* `this [` *formal-parameter-list* `] {` *interface-accessors* `}`

B.2.11 枚举

enum-declaration:
attributes$_{opt}$ *enum-modifiers*$_{opt}$ `enum` *identifier enum-base*$_{opt}$ *enum-body* `;`$_{opt}$

enum-base:
`:` *integral-type*

enum-body:
`{` *enum-member-declarations*$_{opt}$ `}`
`{` *enum-member-declarations* `, }`

enum-modifiers:
enum-modifier
enum-modifiers enum-modifier

enum-modifier:
`new`
`public`
`protected`
`internal`
`private`

enum-member-declarations:
enum-member-declaration
enum-member-declarations `,` *enum-member-declaration*

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

enum-member-declaration:
attributes$_{opt}$ *identifier*
attributes$_{opt}$ *identifier* = *constant-expression*

B.2.12 委托

delegate-declaration:
attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type*
identifier variant-type-parameter-list$_{opt}$
`(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

delegate-modifiers:
delegate-modifier
delegate-modifiers delegate-modifier

delegate-modifier:
`new`
`public`
`protected`
`internal`
`private`

B.2.13 特性

global-attributes:
global-attribute-sections

global-attribute-sections:
global-attribute-section
global-attribute-sections global-attribute-section

global-attribute-section:
`[` *global-attribute-target-specifier attribute-list* `]`
`[` *global-attribute-target-specifier attribute-list* `,` `]`

global-attribute-target-specifier:
global-attribute-target `:`

global-attribute-target:
`assembly`
`module`

attributes:
attribute-sections

attribute-sections:
attribute-section
attribute-sections attribute-section

attribute-section:
`[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`
`[` *attribute-target-specifier*$_{opt}$ *attribute-list* `,` `]`

attribute-target-specifier:
attribute-target `:`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

attribute-target:
    `field`
    `event`
    `method`
    `param`
    `property`
    `return`
    `type`

attribute-list:
    attribute
    attribute-list `,` *attribute*

attribute:
    attribute-name *attribute-arguments*$_{opt}$

attribute-name:
    type-name

attribute-arguments:
    `(` *positional-argument-list*$_{opt}$ `)`
    `(` *positional-argument-list* `,` *named-argument-list* `)`
    `(` *named-argument-list* `)`

positional-argument-list:
    positional-argument
    positional-argument-list `,` *positional-argument*

positional-argument:
    argument-name$_{opt}$ *attribute-argument-expression*

named-argument-list:
    named-argument
    named-argument-list `,` *named-argument*

named-argument:
    identifier `=` *attribute-argument-expression*

attribute-argument-expression:
    expression

B.3 不安全代码的语法扩展

class-modifier:
    ...
    `unsafe`

struct-modifier:
    ...
    `unsafe`

interface-modifier:
    ...
    `unsafe`

delegate-modifier:
    ...
    `unsafe`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

field-modifier:
    ...
    unsafe

method-modifier:
    ...
    unsafe

property-modifier:
    ...
    unsafe

event-modifier:
    ...
    unsafe

indexer-modifier:
    ...
    unsafe

operator-modifier:
    ...
    unsafe

constructor-modifier:
    ...
    unsafe

destructor-declaration:
    attributes$_{opt}$ extern$_{opt}$ unsafe$_{opt}$ ~ *identifier* ( ) *destructor-body*
    attributes$_{opt}$ unsafe$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*

static-constructor-modifiers:
    extern$_{opt}$ unsafe$_{opt}$ static
    unsafe$_{opt}$ extern$_{opt}$ static
    extern$_{opt}$ static unsafe$_{opt}$
    unsafe$_{opt}$ static extern$_{opt}$
    static extern$_{opt}$ unsafe$_{opt}$
    static unsafe$_{opt}$ extern$_{opt}$

embedded-statement:
    ...
    unsafe-statement
    fixed-statement

unsafe-statement:
    unsafe *block*

type:
    ...
    pointer-type

pointer-type:
    unmanaged-type *
    void *

unmanaged-type:
    type

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

primary-no-array-creation-expression:
    ...
    pointer-member-access
    pointer-element-access
    sizeof-expression

unary-expression:
    ...
    pointer-indirection-expression
    addressof-expression

pointer-indirection-expression:
    `*` *unary-expression*

pointer-member-access:
    primary-expression `->` *identifier type-argument-list*$_{opt}$

pointer-element-access:
    primary-no-array-creation-expression `[` *expression* `]`

addressof-expression:
    `&` *unary-expression*

sizeof-expression:
    `sizeof` `(` *unmanaged-type* `)`

fixed-statement:
    `fixed` `(` *pointer-type fixed-pointer-declarators* `)` *embedded-statement*

fixed-pointer-declarators:
    fixed-pointer-declarator
    fixed-pointer-declarators `,` *fixed-pointer-declarator*

fixed-pointer-declarator:
    identifier `=` *fixed-pointer-initializer*

fixed-pointer-initializer:
    `&` *variable-reference*
    expression

struct-member-declaration:
    ...
    fixed-size-buffer-declaration

fixed-size-buffer-declaration:
    attributes$_{opt}$ *fixed-size-buffer-modifiers*$_{opt}$ `fixed` *buffer-element-type*
        fixed-size-buffer-declarators `;`

fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-size-buffer-modifier fixed-size-buffer-modifiers

fixed-size-buffer-modifier:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `unsafe`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

buffer-element-type:
    type

fixed-size-buffer-declarators:
    fixed-size-buffer-declarator
    fixed-size-buffer-declarator *fixed-size-buffer-declarators*

fixed-size-buffer-declarator:
    identifier `[` *constant-expression* `]`

local-variable-initializer:
    ...
    stackalloc-initializer

stackalloc-initializer:
    `stackalloc` *unmanaged-type* `[` *expression* `]`

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。

C. 参考资料

Unicode 联合会。The Unicode Standard, Version 3.0（Unicode 标准 3.0 版）。Addison-Wesley，Reading，Massachusetts，2000，ISBN 0-201-616335-5。

IEEE。IEEE Standard for Binary Floating-Point Arithmetic（二进制浮点算术运算 IEEE 标准）。ANSI/IEEE 标准 754-1985。在 http://www.ieee.org 上提供。

ISO/IEC.*C*++。ANSI/ISO/IEC 14882:1998。

版权所有 © Microsoft Corporation 1999-2012。保留所有权利。