Microsoft®  
Visual Basic®  
语言规范

版本 11.0

Paul Vick、Lucian Wischik

Microsoft Corporation

本文档中包含的信息代表 Microsoft Corporation 对于所讨论问题在发布时的当前观点。由于 Microsoft 必须对不断变化的市场环境作出响应，因此不应将这些内容视作 Microsoft 一方的承诺，并且 Microsoft 不能保证所提供的任何信息在发布日期之后的准确性。

本语言规范仅供参考之用。Microsoft 对本文档中的信息不提供任何明确、隐含或法定的保证。

遵守所有适用的著作权法是用户的责任。未经 Microsoft Corporation 明确的书面许可，不得出于任何目的或以任何形式或任何手段（电子、机械、复印、记录或其他方法）复制或传播本文档的任何部分，也不得将其存储或引入检索系统。受版权法保护的权利不受此限制。

对于本文档中的主题，Microsoft 可能具有专利、专利申请、商标、版权或其他知识产权。除非 Microsoft 在任何书面许可协议中明确规定，否则提供本文档并不授予您这些专利、商标、版权或其他知识产权的任何许可证。

除非另有说明，否则此处描述的示例公司、组织、产品、域名、电子邮件地址、徽标、人员、地点和事件均属虚构，无意与任何真实的公司、组织、产品、域名、电子邮件地址、徽标、人员、地点和事件关联，也不应作此推测。

© 2012 Microsoft Corporation。保留所有权利。

Microsoft、MS-DOS、Visual Basic、Windows 2000、Windows 95、Windows 98、Windows ME、Windows NT、Windows XP、Windows Vista 和 Windows 是 Microsoft Corporation 在美国和/或其他国家/地区的注册商标或商标。

这里提到的实际的公司名称和产品名称可能是其各自所有者的商标。

目录

[1. 介绍 1](#_Toc327262933)

[1.1 文法表示法 1](#_Toc327262934)

[1.2 兼容性 1](#_Toc327262935)

[1.2.1 各种破坏兼容性的情况 2](#_Toc327262936)

[1.2.2 影响条件 2](#_Toc327262937)

[1.2.3 语言弃用 3](#_Toc327262938)

[2. 词法文法 5](#_Toc327262939)

[2.1 字符和行 5](#_Toc327262940)

[2.1.1 行终止符 5](#_Toc327262941)

[2.1.2 行继续符 5](#_Toc327262942)

[2.1.3 空白 7](#_Toc327262943)

[2.1.4 注释 7](#_Toc327262944)

[2.2 标识符 7](#_Toc327262945)

[2.2.1 类型字符 8](#_Toc327262946)

[2.3 关键字 9](#_Toc327262947)

[2.4 文本 10](#_Toc327262948)

[2.4.1 Boolean 文本 11](#_Toc327262949)

[2.4.2 Integer 文本 11](#_Toc327262950)

[2.4.3 Floating-Point 文本 12](#_Toc327262951)

[2.4.4 字符串文本 12](#_Toc327262952)

[2.4.5 字符文本 13](#_Toc327262953)

[2.4.6 日期文本 13](#_Toc327262954)

[2.4.7 Nothing 14](#_Toc327262955)

[2.5 分隔符 14](#_Toc327262956)

[2.6 运算符字符 14](#_Toc327262957)

[3. 预处理指令 15](#_Toc327262958)

[3.1 条件编译 15](#_Toc327262959)

[3.1.1 条件常量指令 16](#_Toc327262960)

[3.1.2 条件编译指令 17](#_Toc327262961)

[3.2 外部源指令 17](#_Toc327262962)

[3.3 区域指令 18](#_Toc327262963)

[3.4 外部校验和指令 18](#_Toc327262964)

[4. 一般概念 21](#_Toc327262965)

[4.1 声明 21](#_Toc327262966)

[4.1.1 重载和签名 21](#_Toc327262967)

[4.2 范围 22](#_Toc327262968)

[4.3 继承 23](#_Toc327262969)

[4.3.1 MustInherit 和 NotInheritable 类 24](#_Toc327262970)

[4.3.2 接口和多重继承 25](#_Toc327262971)

[4.3.3 隐藏 27](#_Toc327262972)

[4.4 实现 31](#_Toc327262973)

[4.4.1 实现方法 34](#_Toc327262974)

[4.5 多态性 35](#_Toc327262975)

[4.5.1 重写的方法 37](#_Toc327262976)

[4.6 可访问性 40](#_Toc327262977)

[4.6.1 构成类型 42](#_Toc327262978)

[4.7 类型和名称空间名称 42](#_Toc327262979)

[4.7.1 用于命名空间和类型的限定名称解析 43](#_Toc327262980)

[4.7.2 用于命名空间和类型的非限定名称解析 44](#_Toc327262981)

[4.8 变量 45](#_Toc327262982)

[4.9 泛型类型和方法 45](#_Toc327262983)

[4.9.1 类型形参 46](#_Toc327262984)

[4.9.2 类型约束 48](#_Toc327262985)

[4.9.3 类型形参差异 52](#_Toc327262986)

[5. 特性 55](#_Toc327262987)

[5.1 特性类 56](#_Toc327262988)

[5.2 特性块 57](#_Toc327262989)

[5.2.1 特性名称 58](#_Toc327262990)

[5.2.2 特性参数 59](#_Toc327262991)

[6. 源文件和命名空间 61](#_Toc327262992)

[6.1 程序启动和终止 61](#_Toc327262993)

[6.2 编译选项 62](#_Toc327262994)

[6.2.1 Option Explicit 语句 62](#_Toc327262995)

[6.2.2 Option Strict 语句 62](#_Toc327262996)

[6.2.3 Option Compare 语句 63](#_Toc327262997)

[6.2.4 整数溢出检查 63](#_Toc327262998)

[6.2.5 Option Infer 语句 63](#_Toc327262999)

[6.3 Imports 语句 64](#_Toc327263000)

[6.3.1 导入别名 65](#_Toc327263001)

[6.3.2 命名空间导入 67](#_Toc327263002)

[6.3.3 XML 命名空间导入 68](#_Toc327263003)

[6.4 命名空间 69](#_Toc327263004)

[6.4.1 命名空间声明 69](#_Toc327263005)

[6.4.2 命名空间成员 71](#_Toc327263006)

[7. 类型 73](#_Toc327263007)

[7.1 值类型和引用类型 73](#_Toc327263008)

[7.1.1 可以为 Null 的值类型 74](#_Toc327263009)

[7.2 接口实现 75](#_Toc327263010)

[7.3 基元类型 76](#_Toc327263011)

[7.4 枚举 77](#_Toc327263012)

[7.4.1 枚举成员 77](#_Toc327263013)

[7.4.2 枚举值 78](#_Toc327263014)

[7.5 类 79](#_Toc327263015)

[7.5.1 类基本规范 80](#_Toc327263016)

[7.5.2 类成员 81](#_Toc327263017)

[7.6 结构 81](#_Toc327263018)

[7.6.1 结构成员 82](#_Toc327263019)

[7.7 标准模块 83](#_Toc327263020)

[7.7.1 标准模块成员 84](#_Toc327263021)

[7.8 接口 84](#_Toc327263022)

[7.8.1 接口继承 85](#_Toc327263023)

[7.8.2 接口成员 86](#_Toc327263024)

[7.9 数组 87](#_Toc327263025)

[7.10 委托 89](#_Toc327263026)

[7.11 分部类型 90](#_Toc327263027)

[7.12 构造类型 91](#_Toc327263028)

[7.12.1 开放和封闭类型 92](#_Toc327263029)

[7.13 特殊类型 93](#_Toc327263030)

[8. 转换 95](#_Toc327263031)

[8.1 隐式和显式转换 95](#_Toc327263032)

[8.2 布尔转换 95](#_Toc327263033)

[8.3 数值转换 96](#_Toc327263034)

[8.4 引用转换 97](#_Toc327263035)

[8.4.1 引用差异转换 97](#_Toc327263036)

[8.4.2 匿名委托转换 98](#_Toc327263037)

[8.5 数组转换 98](#_Toc327263038)

[8.6 值类型转换 100](#_Toc327263039)

[8.6.1 可为 null 的值类型转换 103](#_Toc327263040)

[8.7 字符串转换 104](#_Toc327263041)

[8.8 扩大转换 104](#_Toc327263042)

[8.9 收缩转换 106](#_Toc327263043)

[8.10 类型形参转换 108](#_Toc327263044)

[8.11 用户定义的转换 109](#_Toc327263045)

[8.11.1 最具体的扩大转换 110](#_Toc327263046)

[8.11.2 最具体的收缩转换 110](#_Toc327263047)

[8.12 本机转换 111](#_Toc327263048)

[8.13 主导类型 111](#_Toc327263049)

[9. 类型成员 113](#_Toc327263050)

[9.1 接口方法实现 113](#_Toc327263051)

[9.2 方法 115](#_Toc327263052)

[9.2.1 常规、异步和迭代器方法声明 117](#_Toc327263053)

[9.2.2 外部方法声明 119](#_Toc327263054)

[9.2.3 重写方法 120](#_Toc327263055)

[9.2.4 共享方法 121](#_Toc327263056)

[9.2.5 方法参数 122](#_Toc327263057)

[9.2.5.1 值形参 122](#_Toc327263058)

[9.2.5.2 引用参数 123](#_Toc327263059)

[9.2.5.3 可选参数 125](#_Toc327263060)

[9.2.5.4 ParamArray 形参 125](#_Toc327263061)

[9.2.6 事件处理 126](#_Toc327263062)

[9.2.7 扩展方法 127](#_Toc327263063)

[9.2.8 分部方法 131](#_Toc327263064)

[9.3 构造函数 132](#_Toc327263065)

[9.3.1 实例构造函数 132](#_Toc327263066)

[9.3.2 共享构造函数 134](#_Toc327263067)

[9.4 事件 136](#_Toc327263068)

[9.4.1 自定义事件 138](#_Toc327263069)

[9.5 常量 140](#_Toc327263070)

[9.6 实例和共享变量 141](#_Toc327263071)

[9.6.1 只读变量 143](#_Toc327263072)

[9.6.2 WithEvents 变量 144](#_Toc327263073)

[9.6.3 变量初始值设定项 145](#_Toc327263074)

[9.6.3.1 常规初始值设定项 147](#_Toc327263075)

[9.6.3.2 对象初始值设定项 148](#_Toc327263076)

[9.6.3.3 数组大小初始值设定项 148](#_Toc327263077)

[9.6.4 System.MarshalByRefObject 类 149](#_Toc327263078)

[9.7 属性 149](#_Toc327263079)

[9.7.1 Get 访问器声明 154](#_Toc327263080)

[9.7.2 Set 访问器声明 155](#_Toc327263081)

[9.7.3 默认属性 156](#_Toc327263082)

[9.7.4 自动实现的属性 157](#_Toc327263083)

[9.7.5 迭代器属性 158](#_Toc327263084)

[9.8 运算符 159](#_Toc327263085)

[9.8.1 一元运算符 160](#_Toc327263086)

[9.8.2 二元运算符 160](#_Toc327263087)

[9.8.3 转换运算符 161](#_Toc327263088)

[9.8.4 运算符映射 162](#_Toc327263089)

[10. 语句 165](#_Toc327263090)

[10.1 控制流 165](#_Toc327263091)

[10.1.1 常规方法 165](#_Toc327263092)

[10.1.2 迭代器方法 166](#_Toc327263093)

[10.1.3 异步方法 167](#_Toc327263094)

[10.1.4 块和标签 169](#_Toc327263095)

[10.1.5 局部变量和参数 170](#_Toc327263096)

[10.2 局部声明语句 172](#_Toc327263097)

[10.2.1 隐式局部声明 174](#_Toc327263098)

[10.3 With 语句 175](#_Toc327263099)

[10.4 SyncLock 语句 175](#_Toc327263100)

[10.5 事件语句 176](#_Toc327263101)

[10.5.1 RaiseEvent 语句 177](#_Toc327263102)

[10.5.2 AddHandler 和 RemoveHandler 语句 178](#_Toc327263103)

[10.6 赋值语句 178](#_Toc327263104)

[10.6.1 常规赋值语句 179](#_Toc327263105)

[10.6.2 复合赋值语句 180](#_Toc327263106)

[10.6.3 Mid 赋值语句 181](#_Toc327263107)

[10.7 调用语句 181](#_Toc327263108)

[10.8 条件语句 182](#_Toc327263109)

[10.8.1 If...Then...Else 语句 182](#_Toc327263110)

[10.8.2 Select Case 语句 183](#_Toc327263111)

[10.9 循环 语句 184](#_Toc327263112)

[10.9.1 While...End While 和 Do...Loop 语句 185](#_Toc327263113)

[10.9.2 For...Next 语句 186](#_Toc327263114)

[10.9.3 For Each...Next 语句 188](#_Toc327263115)

[10.10 异常处理语句 191](#_Toc327263116)

[10.10.1 结构化的异常处理语句 191](#_Toc327263117)

[10.10.1.1 Finally 块 192](#_Toc327263118)

[10.10.1.2 Catch 块 192](#_Toc327263119)

[10.10.1.3 Throw 语句 193](#_Toc327263120)

[10.10.2 非结构化的异常处理语句 194](#_Toc327263121)

[10.10.2.1 Error 语句 194](#_Toc327263122)

[10.10.2.2 On Error 语句 194](#_Toc327263123)

[10.10.2.3 Resume 语句 195](#_Toc327263124)

[10.11 分支语句 196](#_Toc327263125)

[10.12 数组处理语句 197](#_Toc327263126)

[10.12.1 ReDim 语句 197](#_Toc327263127)

[10.12.2 Erase 语句 198](#_Toc327263128)

[10.13 Using 语句 198](#_Toc327263129)

[10.14 Await 语句 199](#_Toc327263130)

[10.15 yield 语句 199](#_Toc327263131)

[11. 表达式 201](#_Toc327263132)

[11.1 表达式的分类 201](#_Toc327263133)

[11.1.1 表达式的重新分类 202](#_Toc327263134)

[11.2 常量表达式 205](#_Toc327263135)

[11.3 后期绑定表达式 205](#_Toc327263136)

[11.4 简单表达式 207](#_Toc327263137)

[11.4.1 文本表达式 207](#_Toc327263138)

[11.4.2 带括号的表达式 207](#_Toc327263139)

[11.4.3 实例表达式 207](#_Toc327263140)

[11.4.4 简单名称表达式 207](#_Toc327263141)

[11.4.5 AddressOf 表达式 209](#_Toc327263142)

[11.5 类型表达式 209](#_Toc327263143)

[11.5.1 GetType 表达式 209](#_Toc327263144)

[11.5.2 *CommaList* ::= *Comma* | *CommaList* *Comma*TypeOf...Is 表达式 210](#_Toc327263145)

[11.5.3 Is 表达式 210](#_Toc327263146)

[11.5.4 GetXmlNamespace 表达式 211](#_Toc327263147)

[11.6 成员访问表达式 211](#_Toc327263148)

[11.6.1 相同的类型和成员名称 214](#_Toc327263149)

[11.6.2 默认实例 215](#_Toc327263150)

[11.6.2.1 默认实例和类型名称 216](#_Toc327263151)

[11.6.2.2 组类 216](#_Toc327263152)

[11.6.3 扩展方法收集 218](#_Toc327263153)

[11.7 字典成员访问表达式 221](#_Toc327263154)

[11.8 调用表达式 222](#_Toc327263155)

[11.8.1 重载的方法解析 222](#_Toc327263156)

[11.8.1.1 给定实参列表时成员/类型的特异性 229](#_Toc327263157)

[11.8.1.2 一般性 230](#_Toc327263158)

[11.8.1.3 泛型深度 231](#_Toc327263159)

[11.8.2 实参列表适用性 232](#_Toc327263160)

[11.8.3 为可选形参传递实参并选择实参 233](#_Toc327263161)

[11.8.4 条件方法 234](#_Toc327263162)

[11.8.5 类型实参推断 235](#_Toc327263163)

[11.9 索引表达式 237](#_Toc327263164)

[11.10 New 表达式 237](#_Toc327263165)

[11.10.1 对象创建表达式 238](#_Toc327263166)

[11.10.2 数组表达式 240](#_Toc327263167)

[11.10.2.1 数组创建表达式 240](#_Toc327263168)

[11.10.2.2 数组文本 241](#_Toc327263169)

[11.10.3 委托创建表达式 242](#_Toc327263170)

[11.10.4 匿名对象创建表达式 245](#_Toc327263171)

[11.11 强制转换表达式 247](#_Toc327263172)

[11.12 运算符表达式 248](#_Toc327263173)

[11.12.1 运算符的优先级和结合性 249](#_Toc327263174)

[11.12.2 对象操作数 249](#_Toc327263175)

[11.12.3 运算符解析 250](#_Toc327263176)

[11.13 算术运算符 252](#_Toc327263177)

[11.13.1 一元加运算符 252](#_Toc327263178)

[11.13.2 一元减运算符 253](#_Toc327263179)

[11.13.3 加法运算符 253](#_Toc327263180)

[11.13.4 减法运算符 254](#_Toc327263181)

[11.13.5 乘法运算符 255](#_Toc327263182)

[11.13.6 除法运算符 256](#_Toc327263183)

[11.13.7 Mod 运算符 258](#_Toc327263184)

[11.13.8 幂运算符 258](#_Toc327263185)

[11.14 关系运算符 259](#_Toc327263186)

[11.15 Like 运算符 261](#_Toc327263187)

[11.16 串联运算符 262](#_Toc327263188)

[11.17 逻辑运算符 263](#_Toc327263189)

[11.17.1 短路逻辑运算符 264](#_Toc327263190)

[11.18 移位运算符 266](#_Toc327263191)

[11.19 布尔表达式 267](#_Toc327263192)

[11.20 lambda 表达式 268](#_Toc327263193)

[11.20.1 闭包 272](#_Toc327263194)

[11.21 查询表达式 274](#_Toc327263195)

[11.21.1 范围变量 274](#_Toc327263196)

[11.21.2 可查询类型 275](#_Toc327263197)

[11.21.3 默认查询索引器 277](#_Toc327263198)

[11.21.4 From 查询运算符 277](#_Toc327263199)

[11.21.5 Join 查询运算符 278](#_Toc327263200)

[11.21.6 Let 查询运算符 279](#_Toc327263201)

[11.21.7 Select 查询运算符 280](#_Toc327263202)

[11.21.8 Distinct 查询运算符 281](#_Toc327263203)

[11.21.9 Where 查询运算符 281](#_Toc327263204)

[11.21.10 分区查询运算符 282](#_Toc327263205)

[11.21.11 Order By 查询运算符 283](#_Toc327263206)

[11.21.12 Group By 查询运算符 284](#_Toc327263207)

[11.21.13 Aggregate 查询运算符 286](#_Toc327263208)

[11.21.14 Group Join 查询运算符 287](#_Toc327263209)

[11.22 条件表达式 288](#_Toc327263210)

[11.23 XML 文本表达式 288](#_Toc327263211)

[11.23.1 词法规则 289](#_Toc327263212)

[11.23.2 嵌入式表达式 290](#_Toc327263213)

[11.23.3 XML 文档 290](#_Toc327263214)

[11.23.4 XML 元素 292](#_Toc327263215)

[11.23.5 XML 命名空间 294](#_Toc327263216)

[11.23.6 XML 处理指令 295](#_Toc327263217)

[11.23.7 XML 注释 295](#_Toc327263218)

[11.23.8 CDATA 节 295](#_Toc327263219)

[11.24 XML 成员访问表达式 296](#_Toc327263220)

[11.25 Await 运算符 297](#_Toc327263221)

[12. 文档注释 301](#_Toc327263222)

[12.1 文档注释格式 301](#_Toc327263223)

[12.2 建议的标记 301](#_Toc327263224)

[12.2.1 <c> 302](#_Toc327263225)

[12.2.2 <code> 302](#_Toc327263226)

[12.2.3 <example> 303](#_Toc327263227)

[12.2.4 <exception> 303](#_Toc327263228)

[12.2.5 <include> 303](#_Toc327263229)

[12.2.6 <list> 304](#_Toc327263230)

[12.2.7 <para> 304](#_Toc327263231)

[12.2.8 <param> 305](#_Toc327263232)

[12.2.9 <paramref> 305](#_Toc327263233)

[12.2.10 <permission> 305](#_Toc327263234)

[12.2.11 <remarks> 306](#_Toc327263235)

[12.2.12 <returns> 306](#_Toc327263236)

[12.2.13 <see> 306](#_Toc327263237)

[12.2.14 <seealso> 307](#_Toc327263238)

[12.2.15 <summary> 307](#_Toc327263239)

[12.2.16 <typeparam> 307](#_Toc327263240)

[12.2.17 <value> 307](#_Toc327263241)

[12.3 ID 字符串 308](#_Toc327263242)

[12.3.1 ID 字符串示例 308](#_Toc327263243)

[12.4 文档注释示例 312](#_Toc327263244)

[13. 语法摘要 317](#_Toc327263245)

[13.1 词法文法 317](#_Toc327263246)

[13.1.1 字符和行 317](#_Toc327263247)

[13.1.2 标识符 318](#_Toc327263248)

[13.1.3 关键字 319](#_Toc327263249)

[13.1.4 文本 319](#_Toc327263250)

[13.2 预处理指令 321](#_Toc327263251)

[13.2.1 条件编译 321](#_Toc327263252)

[13.2.2 外部源指令 322](#_Toc327263253)

[13.2.3 区域指令 322](#_Toc327263254)

[13.2.4 外部校验和指令 322](#_Toc327263255)

[13.3 句法文法 322](#_Toc327263256)

[13.3.1 特性 323](#_Toc327263257)

[13.3.2 源文件和命名空间 324](#_Toc327263258)

[13.3.3 类型 325](#_Toc327263259)

[13.3.4 类型成员 328](#_Toc327263260)

[13.3.5 语句 333](#_Toc327263261)

[13.3.6 表达式 338](#_Toc327263262)

[14. 更改列表 349](#_Toc327263263)

[14.1 版本 7.1 到版本 8.0 349](#_Toc327263264)

[14.1.1 主要更改 349](#_Toc327263265)

[14.1.2 次要更改 349](#_Toc327263266)

[14.1.3 澄清/勘误表 350](#_Toc327263267)

[14.1.4 其他 352](#_Toc327263268)

[14.2 版本 8.0（第二版本）到版本 8.0 353](#_Toc327263269)

[14.2.1 次要更改 353](#_Toc327263270)

[14.2.2 澄清/勘误表 353](#_Toc327263271)

[14.2.3 其他 354](#_Toc327263272)

[14.3 版本 8.0（第二版本）到版本 9.0 354](#_Toc327263273)

[14.3.1 主要更改 354](#_Toc327263274)

[14.3.2 次要更改 355](#_Toc327263275)

[14.3.3 澄清/勘误表 355](#_Toc327263276)

[14.3.4 其他 356](#_Toc327263277)

[14.4 版本 9.0 到版本 10.0 356](#_Toc327263278)

[14.4.1 主要更改 356](#_Toc327263279)

[14.4.2 次要更改 357](#_Toc327263280)

[14.4.3 澄清/勘误表 357](#_Toc327263281)

[14.5 版本 10.0 到版本 11.0 359](#_Toc327263282)

[14.5.1 主要更改 359](#_Toc327263283)

[14.5.2 次要更改 359](#_Toc327263284)

[14.5.3 澄清/勘误表 360](#_Toc327263285)

# 介绍

Microsoft® Visual Basic® 是针对 Microsoft .NET Framework 的高级编程语言。这种语言虽然在设计上简单易学，但也具备足够满足资深编程人员需求的强大功能。Visual Basic 编程语言的语法类似于英语，这使 Visual Basic 代码更为清晰易懂。这种语言尽可能使用有意义的单词或短语，而不是缩写词、首字母缩写词或特殊字符。外来语法或不需要的语法通常允许使用，但不是必需的。

Visual Basic 编程语言既可以是强类型语言，又可以是松散类型语言。松散类型将大部分类型检查负担都推迟到程序运行时进行。这不仅包括转换的类型检查，还包括方法调用的类型检查，这意味着方法调用的绑定可以推迟到运行时进行。在构建原型或开发速度比执行速度更为重要的其他程序时，这种做法非常有用。Visual Basic 编程语言还提供了强类型语义，这些语义在编译时执行所有类型检查并禁止方法调用的运行时绑定。这可以保证最佳性能并有助于确保类型转换正确。当构建执行速度和执行正确性非常重要的生产应用程序时，这非常有用。

本文档描述 Visual Basic 语言，目的是提供完整的语言说明，而不是语言教程或用户参考手册。

## 文法表示法

本规范描述两种文法：词法文法和句法文法。词法文法定义如何将字符组合成标记；句法文法定义如何将标记组合成 Visual Basic 程序。还有几种用于条件编译等预处理操作的辅助文法。

注意   本规范中的文法是可读文法，而非正式文法（即可由 LEX 或 YACC 使用的文法）。

所有文法均使用经过修改的 BNF 表示法，由结束符和非结束符名称组成的一组产生式组成。结束符名称表示一个或多个 Unicode 字符。每个非结束符名称都由一个或多个产生式定义。在产生式中，非结束符名称以斜体 显示，结束符名称以fixed-width type显示。普通字体的文本和尖括号中的文本是非正式结束符（例如，“< all Unicode characters >”）。每个文法都以非结束符 Start 开始。

大小写在 Visual Basic 程序中并不重要。为简单起见，所有结束符都将以标准大小写给出，但任意大小写均可。作为 ASCII 字符集的可打印元素的结束符由其相应的 ASCII 字符表示。当匹配结束符时，Visual Basic 还不区分宽度，从而允许全角 Unicode 字符与其相应半角 Unicode 字符相匹配，但必须以整个标记为基础。如果标记混合了半角和全角字符，将不会匹配。

一组产生式以非结束符名称开始，后跟两个冒号和一个等号。右侧包含一个结束符或非结束符产生式。非结束符可以有多个以竖线元符号 (|) 分隔的产生式。方括号元符号 ([]) 中的项是可选的。项后跟加号元符号 (+) 表示该项可发生一次或多次。

为增强可读性，可添加换行符或缩进，但它们不属产生式的范围。

## 兼容性

编程语言的一项重要功能是在不同语言版本之间的兼容性。如果新语言版本不接受旧语言版本的相同代码，或对代码的解释与旧版本不同，则会增加编程人员将代码从一个语言版本升级到另一个语言版本时的负担。因此，除非语言使用者有明显的巨大收益，否则必须保持版本间的兼容性。

不同 Visual Basic 语言版本间的更改通过以下策略进行管理。术语“语言”在本上下文中使用时仅指 Visual Basic 语言本身的句法和语义方面，而不包括任何作为 Microsoft.VisualBasic 命名空间（和子命名空间）的一部分提供的 .NET Framework 类。.NET Framework 中的所有类通过另一个单独的版本控制和兼容性策略进行管理，该策略不属于本文档的讨论范围。

### 各种破坏兼容性的情况

理想情况下，现有 Visual Basic 版本与所有 Visual Basic 未来版本应 100% 兼容。但在一些情况下，即使需要编程人员付出一些开销，也有必要破坏兼容性。这些情况有：

新警告。引入新警告本身并不破坏兼容性。但是，因为很多开发人员都打开“将警告视为错误”进行编译，所以在引入警告时必须倍加小心。

新关键字。当引入新的语言功能时，可能需要引入新关键字。应进行合理的努力，选择能将与用户标识符冲突的可能性减到最小的关键字，使用起作用的现有关键字。我们将就更新旧版本项目和转义任何新关键字提供帮助。

编译器 Bug。当编译器的行为与语言规范中记录的行为不一致时，可能需要修复编译器行为使其与记录的行为匹配。

规范 Bug。当编译器与语言规范一致，但语言规范明显错误时，可能需要更改语言规范和编译器行为。“明显错误”指记录的行为背离了大部分用户所期望的清晰明确的行为，并为用户产生非常不理想的行为。

规范多义性。当语言规范应说明特定情况下发生的情况但未予说明，并且编译器对该情况的处理不一致或明显错误（定义同前一点）时，可能需要阐明范并更正编译器行为。换言之，当规范说明了情况 a、b、d 和 e 下的行为，但未指出情况 c 下的任何行为，并且情况 c 下的编译器行为不正确时，可能需要记录情况 c 下的行为并更改编译器行为以便与之匹配。（注意，如果规范在某一情况下的行为有多义性，但编译器以没有明显错误的方式运行，则编译器行为成为事实规范。）

使运行时错误成为编译时错误。在保证代码在运行时 100% 失败的情况下（例如，用户代码中有明确的 Bug），最好添加捕获该情况的编译时错误。

规范省略。当语言规范没有特别允许或不允许某一特定情况，并且编译器以不适当的方式处理该情况（如果编译器行为明显错误，则属规范 Bug，而不是规范省略），可能需要阐明规范并更改编译器行为。除常用影响分析外，此类更改进一步限制为对更改影响非常小且开发人员获益非常高的情况。

新功能。一般情况下，引入新功能应该不更改现有语言规范部分或现有编译器行为。在引入新功能需要更改现有语言规范的情况下，仅当影响非常小且功能获益非常高时才可以进行兼容性破坏。

安全性。在异常情况下，出于安全问题可能必须破坏兼容性，如删除或修改本身不安全并对用户构成明显安全风险的功能。

以下情况不属于破坏兼容性的可接受原因：

不合需要或不满意的行为。追溯合理但不合需要或不满意的语言设计或编译器行为不是破坏向后兼容性的理由。而必须使用下面所讲的语言否决过程。

任何其他理由。否则，编译器行为保持向后兼容。

### 影响条件

当考虑兼容性破坏是否可接受时，有几个条件用来确定更改可能造成的影响。影响越大，接受兼容性破坏的可能性就越小。

这些条件是：

更改范围有多大？换句话说，有多少程序可能会受到影响？有多少用户可能会受到影响？编写受更改影响的代码的几率有多大？

有没有什么解决办法能够在更改前达到同样的行为？

更改有多明显？用户能立即得到有关发生更改的反馈吗？还是直接使他们的程序以不同方式执行？

在升级过程中能合理地解决更改吗？有没有可能编写一种工具，可以准确找出发生更改的情况并更改代码以解决更改？

社区对更改有什么反馈？

### 语言弃用

随着时间的推移，部分语言和编译器可能会弃用。如前面所讨论，破坏兼容性来删除这些弃用功能是不可接受的。而是必须遵守以下步骤：

假定 Visual Studio A 版本中存在一项功能，必须从用户社区就该功能弃用请求反馈，在做出任何最终弃用决定之前必须全力注意反馈。根据用户社区反馈，弃用过程可能保留或在任意点放弃。

必须带有警告弃用使用的编译器警告发布 Visual Studio 的完整版本（即不是单点版本）B。该警告必须默认开启并可关闭。弃用必须在产品文档和网站上明确记录。

必须带有不能关闭的编译器警告发布 Visual Studio 的完整版本 C。

随后必须发布 Visual Studio 的完整版本 D，弃用编译器警告转换为编译器错误。D 必须在版本 A 的主流支持阶段（编写本文时为 5 年）结束后发布。

最后，可发布删除了编译器错误的 Visual Studio E 版本。

不允许进行在此弃用框架内无法处理的更改。

# 词法文法

Visual Basic 程序的编译首先涉及到将原始的 Unicode 字符流转换为有序的词法标记集。因为 Visual Basic 语言不是自由格式语言，标记集随后进一步划分为一系列逻辑行。一个逻辑行的跨度为从流的开头或一个行终止符到前面没有行继续符的下一个行终止符或到流的末尾。

注意 随着在该语言 9.0 版本中引入 XML 文本表达式，Visual Basic 不再有独特的词法文法，也就是说，可在不考虑句法上下文的情况下对 Visual Basic 代码进行标记化。这是因为 XML 和 Visual Basic 有不同的词法规则，并且任意特定时间使用哪组词法规则取决于当时所处理的句法构造。本规范保留此词法文法节作为常规 Visual Basic 代码的词法规则指南。但是，就长期计划来说，词法规则将并入句法规则。

Start ::= [ LogicalLine+ ]

LogicalLine ::= [ LogicalLineElement+ ] [ Comment ] LineTerminator

LogicalLineElement ::= WhiteSpace | LineContinuation | Token

Token ::= Identifier | Keyword | Literal | Separator | Operator

## 字符和行

Visual Basic 程序由来自 Unicode 字符集的字符组成。

Character ::= < any Unicode character except a LineTerminator >

### 行终止符

Unicode 换行符字符划分逻辑行。

LineTerminator ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character > < Unicode linefeed character > |  
 < Unicode line separator character (0x2028) > |  
 < Unicode paragraph separator character (0x2029) >

### 行继续符

行继续符由一个空白字符后跟作为文本行中最后一个字符（空白除外）的下划线字符组成。行继续符使得一个逻辑行可以跨越多个物理行。虽然行继续符不是空白，但它们被视为空白。

下面的程序显示一些行继续符：

Module Test  
 Sub Print( \_  
 Param1 As Integer, \_  
 Param2 As Integer )  
  
 If (Param1 < Param2) Or \_  
 (Param1 > Param2) Then  
 Console.WriteLine("Not equal")  
 End If  
 End Function  
End Module

句法文法的某些位置允许使用隐式行继续符。在以下位置遇到行终止符时：

逗号 (,)、左括号 (()、左大括号 ({) 或左嵌入式表达式 (<%=) 之后

成员限定符（. 或 .@ 或 ...）之后，如果限定了某些内容（即不使用隐式 With 上下文）

右括号 ())、右大括号 (}) 或右嵌入式表达式 (%>) 之前

特性上下文中小于号 (<) 之后

特性上下文中大于号 (>) 之前

非文件级别特性上下文中大于号 (>) 之后

在查询运算符（Where、Order、Select 等）前后

表达式上下文中二元运算符（+、-、/、\* 等）之后

任意上下文中赋值运算符（=、:=、+=、-= 等）之后。

行终止符视为行继续符。例如，上例也可以如下编写：

Module Test  
 Sub Print(  
 Param1 As Integer,  
 Param2 As Integer)  
  
 If (Param1 < Param2) Or  
 (Param1 > Param2) Then  
 Console.WriteLine("Not equal")  
 End If  
 End Function  
End Module

只有在紧挨指定标记的之前或之后才能推断隐式行继续符。在行继续符之前或之后不能推断为隐式行继续符。例如：

Dim y = 10  
' Error: Expression expected for assignment to x  
Dim x = \_  
  
y

在条件编译上下文中将不能推断行继续符。

批注

最后这一条约束是必需的，因为不编译的条件编译块中的文本不必在句法上有效。因此，此类块中的文本可能会被条件编译语句意外“选中”，特别是将来在扩展该语言时。

LineContinuation ::= WhiteSpace \_ [ WhiteSpace+ ] LineTerminator

Comma ::= , [ LineTerminator ]

Period ::= . [ LineTerminator ]

OpenParenthesis ::= ( [ LineTerminator ]

CloseParenthesis ::= [ LineTerminator ] )

OpenCurlyBrace ::= { [ LineTerminator ]

CloseCurlyBrace ::= [ LineTerminator ] }

Equals ::= = [ LineTerminator ]

ColonEquals ::= : = [ LineTerminator ]

### 空白

空白只是用来分隔标记，其他情况下将被忽略。只包含空白的逻辑行将被忽略。

注意   行终止符不视为空白。

WhiteSpace ::=  
 < Unicode blank characters (class Zs) > |  
 < Unicode tab character (0x0009) >

### 注释

注释 以单引号字符或关键字 REM 开始。单引号字符是 ASCII 单引号字符、Unicode 左单引号字符或 Unicode 右单引号字符。注释可以从源行中的任意位置开始，物理行结尾结束注释。编译器会忽略注释开始和行终止符之间的字符。因此，不能使用行继续符将注释扩展到多行。

Comment ::= CommentMarker [ Character+ ]

CommentMarker ::= SingleQuoteCharacter | REM

SingleQuoteCharacter ::=  
 ' |  
 < Unicode left single-quote character (0x2018) > |  
 < Unicode right single-quote character (0x2019) >

## 标识符

标识符是名称。Visual Basic 标识符符合 Unicode 标准附录 15，但有一个例外：标识符可以以下划线（连接线）字符开始。如果标识符以下划线开始，它必须包含至少一个其他有效标识符字符来与行继续符区分。

规则标识符不能与关键字冲突，但转义标识符或具有类型字符的标识符可以。转义标识符是以方括号分隔的标识符。转义标识符遵循与规则标识符相同的规则，但转义标识符可以与关键字冲突并且可以没有类型字符。

此示例定义一个名为 class 的类，该类具有一个名为 shared 的共享方法，该类采用名为 boolean 的形参，然后调用该方法。

Class [class]  
 Shared Sub [shared]([boolean] As Boolean)  
 If [boolean] Then  
 Console.WriteLine("true")  
 Else  
 Console.WriteLine("false")  
 End If  
 End Sub  
End Class  
  
Module [module]  
 Sub Main()  
 [class].[shared](True)  
 End Sub  
End Module

标识符不区分大小写，因此只有大小写不同的两个标识符将被认为是同一个标识符。

注意   当比较标识符时，将使用 Unicode 标准一对一大小写映射，并将忽略任何区域设置特定的大小写映射。

Identifier ::=  
 NonEscapedIdentifier [ TypeCharacter ] |  
 Keyword TypeCharacter |  
 EscapedIdentifier

NonEscapedIdentifier ::= < IdentifierName but not Keyword >

EscapedIdentifier ::= [ IdentifierName ]

IdentifierName ::= IdentifierStart [ IdentifierCharacter+ ]

IdentifierStart ::=  
 AlphaCharacter |  
 UnderscoreCharacter IdentifierCharacter

IdentifierCharacter ::=  
 UnderscoreCharacter |  
 AlphaCharacter |  
 NumericCharacter |  
 CombiningCharacter |  
 FormattingCharacter

AlphaCharacter ::=  
 < Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >

NumericCharacter ::= < Unicode decimal digit character (class Nd) >

CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >

FormattingCharacter ::= < Unicode formatting character (class Cf) >

UnderscoreCharacter ::= < Unicode connection character (class Pc) >

IdentifierOrKeyword ::= Identifier | Keyword

### 类型字符

类型字符 表示前面标识符的类型。类型字符不视为标识符的一部分。如果声明包含类型字符，类型字符必须符合声明本身中指定的类型；否则，将引发编译时错误。如果声明省略类型（例如，如果未指定 As 子句），则隐式使用类型字符作为声明的类型。

标识符及其类型字符之间不能有任何空白。由于缺少合适的字符，因此没有适用于 Byte、SByte、UShort、Short、UInteger 或 ULong 的类型字符。

向概念上没有类型的标识符（例如，命名空间名称）或类型与类型字符的类型不匹配的标识符添加类型字符会导致编译时错误。

下例显示类型字符的用法：

' The follow line will cause an error: standard modules have no type.  
Module Test1#  
End Module  
  
Module Test2  
  
 ' This function takes a Long parameter and returns a String.  
 Function Func$(Param&)  
  
 ' The following line causes an error because the type character  
 ' conflicts with the declared type of Func and Param.  
 Func# = CStr(Param@)  
  
 ' The following line is valid.  
 Func$ = CStr(Param&)  
 End Function  
End Module

类型字符 ! 表示存在一个特殊问题，在该语言中，它既可用作类型字符，也可用作分隔符。为消除多义性，如果 ! 字符后的字符不能开始一个标识符，那它就是类型字符。如果能，则 ! 字符就是分隔符，而不是类型字符。

TypeCharacter ::=  
 IntegerTypeCharacter |  
 LongTypeCharacter |  
 DecimalTypeCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 StringTypeCharacter

IntegerTypeCharacter ::= %

LongTypeCharacter ::= &

DecimalTypeCharacter ::= @

SingleTypeCharacter ::= !

DoubleTypeCharacter ::= #

StringTypeCharacter ::= $

## 关键字

关键字是在语言构造中具有特殊含义的词。所有关键字都由语言保留，不能用作标识符（转义标识符除外）。

注意   EndIf、GoSub、Let、Variant 和 Wend 尽管不再在 Visual Basic 中使用，但仍保留为关键字。

Keyword ::= < member of keyword table >

|  |  |  |  |
| --- | --- | --- | --- |
| AddHandler | AddressOf | Alias | And |
| AndAlso | As | Boolean | ByRef |
| Byte | ByVal | Call | Case |
| Catch | CBool | CByte | CChar |
| CDate | CDbl | CDec | Char |
| CInt | Class | CLng | CObj |
| Const | Continue | CSByte | CShort |
| CSng | CStr | CType | CUInt |
| CULng | CUShort | Date | Decimal |
| Declare | Default | Delegate | Dim |
| DirectCast | Do | Double | Each |
| Else | ElseIf | End | EndIf |
| Enum | Erase | Error | Event |
| Exit | False | Finally | For |
| Friend | Function | Get | GetType |
| GetXmlNamespace | Global | GoSub | GoTo |
| Handles | If | Implements | Imports |
| In | Inherits | Integer | Interface |
| Is | IsNot | Let | Lib |
| Like | Long | Loop | Me |
| Mod | Module | MustInherit | MustOverride |
| MyBase | MyClass | Namespace | Narrowing |
| New | Next | Not | Nothing |
| NotInheritable | NotOverridable | Object | Of |
| On | Operator | Option | Optional |
| Or | OrElse | Overloads | Overridable |
| Overrides | ParamArray | Partial | Private |
| Property | Protected | Public | RaiseEvent |
| ReadOnly | ReDim | REM | RemoveHandler |
| Resume | Return | SByte | Select |
| Set | Shadows | Shared | Short |
| Single | Static | Step | Stop |
| String | Structure | Sub | SyncLock |
| Then | Throw | To | True |
| Try | TryCast | TypeOf | UInteger |
| ULong | UShort | Using | Variant |
| Wend | When | While | Widening |
| With | WithEvents | WriteOnly | Xor |

## 文本

文本是某一类型的特定值的文本表示。文本类型包括布尔值、整数、浮点数、字符串、字符和日期。

Literal ::=  
 BooleanLiteral |  
 IntegerLiteral |  
 FloatingPointLiteral |  
 StringLiteral |  
 CharacterLiteral |  
 DateLiteral |  
 Nothing

### Boolean 文本

True 和 False 是 Boolean 类型的文本，分别映射到 true 和 false 状态。

BooleanLiteral ::= True | False

### Integer 文本

Integer 文本可以是十进制（基数为 10）、十六进制（基数为 16）或八进制（基数为 8）。十进制 Integer 文本是十进制数字 (0-9) 字符串。十六进制文本是 &H 后跟十六进制数字（0-9、A-F）字符串。八进制文本是 &O 后跟八进制数字 (0-7) 字符串。十进制文本直接表示整数文本的十进制值，而八进制和十六进制文本表示整数文本的二进制值（因此，&H8000S 是 –32768，而不是溢出错误）。

文本的类型由其值或后跟的类型字符确定。如果未指定类型字符，则 Integer 类型范围内的值类型化为 Integer；Integer 范围外的值类型化为 Long。如果某一整数文本的类型大小不足以容纳整数文本，则会产生编译时错误。

批注

没有针对 Byte 的类型字符，因为最适合的字符是 B，而 B 是十六进制文本中的合法字符。

IntegerLiteral ::= IntegralLiteralValue [ IntegralTypeCharacter ]

IntegralLiteralValue ::= IntLiteral | HexLiteral | OctalLiteral

IntegralTypeCharacter ::=  
 ShortCharacter |  
 UnsignedShortCharacter |  
 IntegerCharacter |  
 UnsignedIntegerCharacter |  
 LongCharacter |  
 UnsignedLongCharacter |  
 IntegerTypeCharacter |  
 LongTypeCharacter

ShortCharacter ::= S

UnsignedShortCharacter ::= US

IntegerCharacter ::= I

UnsignedIntegerCharacter ::= UI

LongCharacter ::= L

UnsignedLongCharacter ::= UL

IntLiteral ::= Digit+

HexLiteral ::= & H HexDigit+

OctalLiteral ::= & O OctalDigit+

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

OctalDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

### Floating-Point 文本

Floating-Point 文本是一个 Integer 文本后跟一个可选的小数点（ASCII 句点字符）和 尾数，以及一个可选的以 10 为基数的指数。默认情况下，Floating-Point 文本为 Double 类型。如果指定 Single、Double 或 Decimal 类型的字符，则文本属于该类型。如果某一 Floating-Point 文本的类型大小不足以容纳 Floating-Point 文本，就会产生编译时错误。

注释：

值得注意的是，Decimal 数据类型可将尾随零编码到值中。当前，规范未对编译器是否识别 Decimal 文本中的尾随零提供任何说明。

FloatingPointLiteral ::=  
 FloatingPointLiteralValue [ FloatingPointTypeCharacter ] |  
 IntLiteral FloatingPointTypeCharacter

FloatingPointTypeCharacter ::=  
 SingleCharacter |  
 DoubleCharacter |  
 DecimalCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 DecimalTypeCharacter

SingleCharacter ::= F

DoubleCharacter ::= R

DecimalCharacter ::= D

FloatingPointLiteralValue ::=  
 IntLiteral . IntLiteral [ Exponent ] |  
 . IntLiteral [ Exponent ] |  
 IntLiteral Exponent

Exponent ::= E [ Sign ] IntLiteral

Sign ::= + | -

### 字符串文本

字符串文本是以 ASCII 双引号字符、Unicode 左双引号字符或 Unicode 右双引号字符开始并结束的零个或零个以上 Unicode 字符序列。在字符串中，两个双引号字符组成的序列是转义序列，表示字符串中的双引号。字符串常量属于 String 类型。

Module Test  
 Sub Main()  
  
 ' This prints out: ".  
 Console.WriteLine("""")  
  
 ' This prints out: a"b.  
 Console.WriteLine("a""b")  
  
 ' This causes a compile error due to mismatched double-quotes.  
 Console.WriteLine("a"b")  
 End Sub  
End Module

允许使用编译器用字符串文本替换常量字符串表达式。每个字符串文本不一定产生新的字符串实例。当根据使用二进制比较语义的字符串相等运算符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，这些字符串可能引用相同的字符串实例。例如，下面程序的输出可能返回 True，因为两个文本可引用同一字符串实例。

Module Test  
 Sub Main()  
 Dim a As Object = "he" & "llo"  
 Dim b As Object = "hello"  
 Console.WriteLine(a Is b)  
 End Sub  
End Module

StringLiteral ::=  
 DoubleQuoteCharacter [ StringCharacter+ ] DoubleQuoteCharacter

DoubleQuoteCharacter ::=  
 " |  
 < Unicode left double-quote character (0x201C) > |  
 < Unicode right double-quote character (0x201D) >

StringCharacter ::=  
 < Character except for DoubleQuoteCharacter > |  
 DoubleQuoteCharacter DoubleQuoteCharacter

### 字符文本

一个字符文本表示一个 Char 类型的 Unicode 字符。两个双引号字符是一个表示双引号字符的转义序列。

Module Test  
 Sub Main()  
  
 ' This prints out: a.  
 Console.WriteLine("a"c)  
  
 ' This prints out: ".  
 Console.WriteLine(""""c)  
 End Sub  
End Module

CharacterLiteral ::= DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C

### 日期文本

日期文本表示特定时刻，表示为 Date 类型的值。日期文本可以指定日期和时间，也可以只指定日期或只指定时间。如果忽略了日期值，将假定为公历 1 年 1 月 1 日。如果忽略了时间值，将假定为 12:00:00 AM。

若要避免日期值中年值的解释问题，年值不能为两位数。当表示第一世纪 AD/CE 中的日期时，必须指定前导零。

可以使用 24 小时值或 12 小时值指定时间值；省略 AM 或 PM 的时间值假定为 24 小时值。如果时间值省略了分钟，将默认使用文本 0。如果时间值省略了秒钟，将默认使用 0。如果同时省略分钟和秒钟，则必须指定 AM 或 PM。如果指定的日期值超出了 Date 类型的范围，将发生编译时错误。

下面的示例包含几个日期文本。

Dim d As Date  
  
d = # 8/23/1970 3:45:39AM #  
d = # 8/23/1970 # ' Date value: 8/23/1970 12:00:00AM.  
d = # 3:45:39AM # ' Date value: 1/1/1 3:45:39AM.  
d = # 3:45:39 # ' Date value: 1/1/1 3:45:39AM.  
d = # 13:45:39 # ' Date value: 1/1/1 1:45:39PM.  
d = # 1AM # ' Date value: 1/1/1 1:00:00AM.  
d = # 13:45:39PM # ' This date value is not valid.

DateLiteral ::= # [ Whitespace+ ] DateOrTime [ Whitespace+ ] #

DateOrTime ::=  
 DateValue Whitespace+ TimeValue |  
 DateValue |  
 TimeValue

DateValue ::=  
 MonthValue / DayValue / YearValue |  
 MonthValue – DayValue - YearValue

TimeValue ::=  
 HourValue : MinuteValue [ : SecondValue ] [ WhiteSpace+ ] [ AMPM ] |  
 HourValue [ WhiteSpace+ ] AMPM

MonthValue ::= IntLiteral

DayValue ::= IntLiteral

YearValue ::= IntLiteral

HourValue ::= IntLiteral

MinuteValue ::= IntLiteral

SecondValue ::= IntLiteral

AMPM ::= AM | PM

ElseIf ::= ElseIf | Else If

### Nothing

Nothing 是一个特殊文本；它没有类型，可以转换为包括类型参数在内的类型系统中的所有类型。当转换为某一特定类型时，它等效于该类型的默认值。

Nothing ::= Nothing

## 分隔符

以下 ASCII 字符为分隔符：

Separator ::= ( | ) | { | } | ! | # | , | . | : | ?

## 运算符字符

以下 ASCII 字符或字符序列表示运算符：

Operator ::=  
 & | \* | + | - | / | \ | ^ | < | = | >

# 预处理指令

对文件进行词法分析时，会发生几种源文件预处理。最重要的是条件编译，它确定句法文法处理的源；另外两种类型的指令（外部源指令和区域指令）提供关于源的元信息，但对编译没有任何作用。

## 条件编译

条件编译控制是否将逻辑行序列转换为实际代码。在条件编译开始时，所有逻辑行都已启用；但是，将行封闭在条件编译语句内会选择性地在文件内禁用这些行，导致它们在剩余编译过程中被忽略。

例如，程序

#Const A = True  
#Const B = False  
  
Class C  
  
#If A Then  
 Sub F()  
 End Sub  
#Else  
 Sub G()  
 End Sub  
#End If  
  
#If B Then  
 Sub H()  
 End Sub  
#Else  
 Sub I()  
 End Sub  
#End If  
  
End Class

产生与下面的程序完全相同的标记序列

Class C  
 Sub F()  
 End Sub  
  
 Sub I()  
 End Sub  
End Class

条件编译指令中允许的常量表达式是一般常量表达式的子集。

预处理器允许在每个标记之前和之后存在空格和显示行继续符。

Start ::= [ CCStatement+ ]

CCStatement ::=  
 CCConstantDeclaration |  
 CCIfGroup |  
 LogicalLine

CCExpression ::=  
 LiteralExpression |  
 CCParenthesizedExpression |  
 CCSimpleNameExpression |  
 CCCastExpression |  
 CCOperatorExpression |  
 CCConditionalExpression

CCParenthesizedExpression ::= ( CCExpression )

CCSimpleNameExpression ::= Identifier

CCCastExpression ::=   
 DirectCast ( CCExpression , TypeName ) |  
 TryCast ( CCExpression , TypeName ) |  
 CType ( CCExpression , TypeName ) |  
 CastTarget ( CCExpression )

CCOperatorExpression ::=  
 CCUnaryOperator CCExpression  
 CCExpression CCBinaryOperator CCExpression

CCUnaryOperator ::= + | - | Not

CCBinaryOperator ::= + | - | \* | / | \ | Mod | ^ | = | < > | < | > |  
 < = | > = | & | And | Or | Xor | AndAlso | OrElse | < < | > >

CCConditionalExpression ::=   
 If ( CCExpression , CCExpression , CCExpression ) |  
 If ( CCExpression , CCExpression )

### 条件常量指令

条件常量语句定义范围为源文件的单独条件编译声明空间中存在的常量。该声明空间的特别之处在于不需要显式声明条件编译常量，条件常量可以在条件编译指令中隐式定义。

在对条件编译常量赋值前，它的值为 Nothing。为条件编译常量赋值（必须为常量表达式）后，常量类型成为指派给它的值的类型。条件编译常量在整个源文件中可重新定义多次。

例如，以下代码仅输出字符串 about to print value 以及 Test 的值。

Module M1  
 Sub PrintValue(Test As Integer)  
  
#Const DebugCode = True  
  
#If DebugCode Then  
 Console.WriteLine("about to print value")  
#End If  
  
#Const DebugCode = False  
  
 Console.WriteLine(Test)  
  
#If DebugCode Then  
 Console.WriteLine("printed value")  
#End If  
  
 End Sub  
End Module

编译环境也可以在条件编译声明空间中定义条件常量。

CCConstantDeclaration ::= # Const Identifier = CCExpression LineTerminator

### 条件编译指令

条件编译指令控制条件编译，并且只能引用常量表达式和条件编译常量。单个条件编译组中的每个常量表达式都以文本顺序从头到尾进行计算并转换为 Boolean 类型，直到一个条件表达式计算为 True。如果有表达式不可转换为 Boolean，则会产生编译时错误。当计算条件编译常量表达式时，无论任何 Option 指令或编译环境设置如何，都会使用容许语义和二进制字符串比较。

封闭在条件编译组中的所有行（包括嵌套条件编译指令）都被禁用，但有两类行例外：一是位于包含 True 表达式的语句与该组的下一个条件语句之间的行；一是 Else 语句与 End If 语句之间的行（如果组中有 Else 并且所有表达式都计算为 False）。

在本例中，将不处理 Trace 条件编译指令中对 WriteToLog 的调用，因为周围的 Debug 条件编译指令计算为 False。

#Const Debug = False ' Debugging off  
#Const Trace = True ' Tracing on  
  
Class PurchaseTransaction  
 Sub Commit()  
  
#If Debug Then  
 CheckConsistency()  
#If Trace Then  
 WriteToLog(Me.ToString())  
#End If  
#End If  
 ...  
 End Sub  
End Class

CCIfGroup ::=  
 # If CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]  
 [ CCElseIfGroup+ ]  
 [ CCElseGroup ]  
 # End If LineTerminator

CCElseIfGroup ::=  
 # ElseIf CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]

CCElseGroup ::=  
 # Else LineTerminator  
 [ CCStatement+ ]

## 外部源指令

源文件可包含外部源指令，外部源指令指示源行和源外部的文本之间的映射。外部源指令对编译没有影响，而且不能嵌套。例如：

Module Test  
 Sub Main()  
  
#ExternalSource("c:\wwwroot\inetpub\test.aspx", 30)  
 Console.WriteLine("In test.aspx")  
#End ExternalSource  
  
 End Sub  
End Module

Start ::= [ ExternalSourceStatement+ ]

ExternalSourceStatement ::= ExternalSourceGroup | LogicalLine

ExternalSourceGroup ::=  
 # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator  
 [ LogicalLine+ ]  
 # End ExternalSource LineTerminator

## 区域指令

区域指令将源代码行分组，但对编译没有任何影响。在集成开发环境 (IDE) 中，可以折叠（隐藏）或展开（查看）整个组。区域可以嵌套。区域指令的特殊之处在于它们不能在方法体内开始和终止，并且它们应采用该程序的块结构。例如：

Module Test#Region "Startup code – do not edit"  
 Sub Main()  
 End Sub  
#End Region  
  
End Module

' Error due to Region directives breaking the block structure  
Class C  
#Region "Fred"  
End Class  
#End Region

Start ::= [ RegionStatement+ ]

RegionStatement ::= RegionGroup | LogicalLine

RegionGroup ::=  
 # Region StringLiteral LineTerminator  
 [ RegionStatement+ ]  
 # End Region LineTerminator

## 外部校验和指令

源文件可以包含外部校验和指令，该指令指示应对外部源指令中引用的文件发出什么校验和。在所有其他方面，外部源指令对编译没有任何作用。

外部校验和指令包含外部文件的文件名、与该文件关联的全局唯一标识符 (GUID) 和该文件的校验和。GUID 指定为“{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}”形式的字符串常量，其中 x 是十六进制数字。校验和指定为“xxxx…”形式的字符串常量，其中 x 是十六进制数字。校验和中的数字位数必须为偶数。

如果所有 GUID 和校验和值都完全匹配，则一个外部文件可以有多个关联的外部校验和指令。如果外部文件的文件名与正在编译的文件的名称匹配，将忽略该校验和，而使用编译器的校验和计算。

例如：

#ExternalChecksum("c:\wwwroot\inetpub\test.aspx", \_  
 "{12345678-1234-1234-1234-123456789abc}", \_  
 "1a2b3c4e5f617239a49b9a9c0391849d34950f923fab9484")  
  
Module Test  
 Sub Main()  
  
#ExternalSource("c:\wwwroot\inetpub\test.aspx", 30)  
 Console.WriteLine("In test.aspx")  
#End ExternalSource  
  
 End Sub  
End Module

Start ::= [ ExternalChecksumStatement+ ]

ExternalChecksumStatement ::=  
 # ExternalChecksum ( StringLiteral , StringLiteral , StringLiteral ) LineTerminator

# 一般概念

本章包含理解 Microsoft Visual Basic 语言语义所需要的大量概念。其中很多概念是 Visual Basic 程序员或 C/C++ 程序员所熟悉的，但是这些概念的精确定义可能不同。

## 声明

Visual Basic 程序是由命名的实体构成的。这些实体通过声明引入，用来表示程序的“意思”。

从较高层面来说，命名空间是用来组织其他实体的实体（如嵌套的命名空间和类型）。类型是用于描述值和定义可执行代码的实体。类型可以包含嵌套类型和类型成员。类型成员包括常量、变量、方法、运算符、属性、事件、枚举值和构造函数。

可包含其他实体的实体定义声明空间。通过声明或继承将实体引入声明空间；包含实体的声明空间称为实体的声明上下文。在声明空间中声明了一个实体，就等于定义了一个新声明空间，该空间可以包含进一步嵌套的实体声明；由此，程序中的声明将形成声明空间的层次结构。

除重载的类型成员之外，声明将同一类型的同名实体引入同一声明上下文的做法是无效的。另外，声明空间永远不能包含不同种类的同名实体；例如，声明空间永远不能包含名称相同的变量和方法。

批注

在其他语言中，可能可以创建包含同名的不同种类实体的声明空间（例如那些区分大小写和允许基于大小写的不同声明的语言）。在这种情况下，将最易于访问的实体视为与该名称绑定；如果最易于访问的实体不止一个类型，则该名称具有多义性。Public 比 Protected Friend 更易于访问，Protected Friend 比 Protected 或 Friend 更易于访问，Protected 或 Friend 比 Private 更易于访问。

命名空间的声明空间是“开放式”的，因此两个具有相同的完全限定名的命名空间声明将构成同一个声明空间。在下面的示例中，两个命名空间声明向同一声明空间提供成员，在这种情况下，声明了带完全限定名 Data.Customer 和 Data.Order 的两个类：

Namespace Data  
 Class Customer  
 End Class  
End Namespace  
  
Namespace Data  
 Class Order  
 End Class  
End Namespace

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。

### 重载和签名

在声明空间中声明同一类型的同名实体的唯一途径是通过重载。只能重载方法、运算符、实例构造函数和属性。

重载的类型成员必须具有唯一的签名。类型成员的签名由类型形参数目以及成员形参的数目和类型组成。转换运算符的签名中还包含运算符的返回类型。

以下实体不是成员签名的组成部分，因此不能进行重载：

类型成员的修饰符（例如，Shared 或 Private）。

形参的修饰符（例如，ByVal 或 ByRef）。

参数名。

方法或运算符的返回类型（除转换运算符之外）或属性的元素类型。

类型参数约束。

下面的示例演示了一个重载方法声明集及其签名。因为有若干个方法声明具有相同的签名，所以此声明无效。

Interface ITest  
 Sub F1() ' Signature is ().  
 Sub F2(x As Integer) ' Signature is (Integer).  
 Sub F3(ByRef x As Integer) ' Signature is (Integer).  
 Sub F4(x As Integer, y As Integer) ' Signature is   
 (Integer, Integer).  
 Function F5(s As String) As Integer ' Signature is (String).  
 Function F6(x As Integer) As Integer ' Signature is (Integer).  
 Sub F7(a() As String) ' Signature is (String()).  
 Sub F8(ParamArray a() As String) ' Signature is (String()).  
 Sub F9(Of T)() ' Signature is !1().  
 Sub F10(Of T, U)(x As T, y As U) ' Signature is !2(!1, !2)  
 Sub F11(Of U, T)(x As T, y As U) ' Signature is !2(!2, !1)  
 Sub F12(Of T)(x As T) ' Signature is !1(!1)  
 Sub F13(Of T As IDisposable)(x As T) ' Signature is !1(!1)  
End Interface

有效的做法是根据提供的类型参数，定义可以包含具有相同签名的成员的一般类型。重载解决方案规则用于尝试消除这些重载之间的歧义，但有时候歧义是无法消除的。例如：

Class C(Of T)  
 Sub F(x As Integer)  
 End Sub  
  
 Sub F(x As T)  
 End Sub  
  
 Sub G(Of U)(x As T, y As U)  
 End Sub  
  
 Sub G(Of U)(x As U, y As T)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As New C(Of Integer)  
 x.F(10) ' Calls C(Of T).F(Integer)  
 x.G(Of Integer)(10, 10) ' Error: Can't choose between overloads  
 End Sub  
End Module

## 范围

实体名的范围是所有声明空间的集合，在该范围内可以无限制引用该名称。一般来说，实体名的范围是其整个声明上下文；但实体声明也可能包含同名实体的嵌套声明。此种情况下，嵌套实体会隐藏外部实体，只能通过限定来访问被隐藏的实体。

嵌套导致的隐藏会在如下情况产生：名称空间中嵌套名称空间或类型时；在类型中嵌套其他类型时；在类型成员体中。声明嵌套导致的隐藏总是隐式发生，不需要明确的语法。

在下面的示例中，在 F 方法中，局部变量 i 隐藏了实例变量 i，但在 G 方法中，i 仍引用实例变量。

Class Test  
 Private i As Integer = 0  
  
 Sub F()  
 Dim i As Integer = 1  
 End Sub  
  
 Sub G()  
 i = 1  
 End Sub  
End Class

当内部范围中的名称隐藏外部范围中的名称时，它会隐藏该名称的所有重载匹配项。在下面的示例中，因为内层声明隐藏了 F 的所有外层匹配项，所以调用 F(1) 将调用在 Inner 中声明的 F。同样的原因，调用 F("Hello") 会出现错误。

Class Outer  
 Shared Sub F(i As Integer)  
 End Sub  
  
 Shared Sub F(s As String)  
 End Sub  
  
 Class Inner  
 Shared Sub F(l As Long)  
 End Sub  
  
 Sub G()  
 F(1) ' Invokes Outer.Inner.F.  
 F("Hello") ' Error.  
 End Sub  
 End Class  
End Class

## 继承

在继承关系中，一种类型（派生类型）是由其他类型（基类型）派生的，因此，该派生类型的声明空间将隐式包含可访问的非构造函数类型成员，以及该派生类型的基类型的嵌套类型。在下面的示例中，类 A 是 B 的基类，B 派生自 A。

Class A  
End Class  
  
Class B  
 Inherits A  
End Class

因为 A 未显式指定基类，所以其基类为隐式指定的 Object。

继承的一些重要性质为：

继承是可传递的。如果类型 C 派生自类型 B，并且类型 B 派生自类型 A，则类型 C 将继承类型 B 以及类型 A 所声明的类型成员。

派生类型只能扩展（而无法缩小）其基类型的范围。派生类型可以添加新的类型成员，并可以隐藏继承的类型成员，但它不能移除继承的类型成员的定义。

因为某类型的实例包含其基类型的所有类型成员，所以始终存在从派生类型到基类型的转换。

除 Object 类型外，所有类型都必须有基类型。因此，Object 是所有类型的最终基类型，所有类型都能转换为该类型。

不允许循环派生。即如果类型 B 派生自类型 A，则类型 A 直接或间接派生自类型 B 是错误的。

某个类型不可以直接或间接派生自其嵌套类型。

由于存在类间的循环依赖，下面的示例产生了编译时错误。

Class A  
 Inherits B  
End Class  
  
Class B  
 Inherits C  
End Class  
  
Class C  
 Inherits A  
End Class

由于 B 通过类 A 间接派生自其嵌套类 C，下面的示例也产生了编译时错误。

Class A  
 Inherits B.C  
End Class  
  
Class B  
 Inherits A  
  
 Public Class C  
 End Class   
End Class

因为 A 未派生自类 B，所以下面的示例不会产生错误。

Class A  
 Class B  
 Inherits A  
 End Class   
End Class

### MustInherit 和 NotInheritable 类

MustInherit 类是不完整的类型，只能作为基类型。MustInherit 类无法实例化，所以对其使用 New 运算符也会产生错误。可以声明 MustInherit 类的变量；对这些变量只能赋予 Nothing，或赋予派生自相应 MustInherit 类的某个类的值。

如果某个常规类派生自 MustInherit 类，该常规类必须重写所有继承的 MustOverride 成员。例如：

MustInherit Class A  
 Public MustOverride Sub F()  
End Class  
  
MustInherit Class B  
 Inherits A  
  
 Public Sub G()  
 End Sub  
End Class   
  
Class C  
 Inherits B  
  
 Public Overrides Sub F()  
 End Sub   
End Class

MustInherit 类 A 引入了 MustOverride 方法 F。类 B 引入其他方法 G，但不提供对 F 的实现。因此，类 B 也必须声明为 MustInherit。类 C 重写 F，并提供一个具体实现。由于类 C 中没有未处理的 MustOverride 成员，所以该类不必是 MustInherit。

NotInheritable 类不能派生其他类。NotInheritable 类主要用于防止发生意外的派生。

在此示例中，类 B 尝试从 NotInheritable 类 A 派生，因此会产生错误。类不能同时标记为 MustInherit 和 NotInheritable。

NotInheritable Class A  
End Class  
  
Class B  
 ' Error, a class cannot derive from a NotInheritable class.  
 Inherits A  
End Class

### 接口和多重继承

其他类型只派生自单一基类型，而接口则可以派生自多个基接口。因此，接口可以从不同的基接口继承同名的类型成员。在这种情况下，多重继承的名称在派生的接口中不可用，并且通过派生接口引用这些类型成员中的任何一个都会导致编译时错误（无论是签名还是重载）。必须转而通过基接口名称引用冲突的类型成员。

在下面的示例中，前两个语句会导致编译时错误，因为多重继承成员 Count 在接口 IListCounter 中不可用：

Interface IList  
 Property Count() As Integer  
End Interface  
  
Interface ICounter  
 Sub Count(i As Integer)  
End Interface  
  
Interface IListCounter  
 Inherits IList  
 Inherits ICounter   
End Interface   
  
Module Test  
 Sub F(x As IListCounter)  
 x.Count(1) ' Error, Count is not available.  
 x.Count = 1 ' Error, Count is not available.  
 CType(x, IList).Count = 1 ' Ok, invokes IList.Count.  
 CType(x, ICounter).Count(1) ' Ok, invokes ICounter.Count.  
 End Sub   
End Module

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。

当单个类型成员通过多个路径继承自同一基接口时，会将该类型成员视为只继承过一次。换言之，派生的接口只包含继承自特定基接口的每个类型成员的一个实例。例如：

Interface IBase  
 Sub F(i As Integer)  
End Interface  
  
Interface ILeft  
 Inherits IBase  
End Interface  
  
Interface IRight  
 Inherits IBase  
End Interface  
  
Interface IDerived  
 Inherits ILeft, IRight  
End Interface  
  
Class Derived  
 Implements IDerived  
  
 ' Only have to implement F once.  
 Sub F(i As Integer) Implements IDerived.F  
 End Sub  
End Class

如果通过继承层次结构在一条路径中隐藏 了某个类型成员名称，则将在所有路径中隐藏该名称。在下面的示例中，IBase.F 成员被 ILeft.F 成员隐藏，但未在 IRight 中隐藏：

Interface IBase  
 Sub F(i As Integer)  
End Interface   
  
Interface ILeft  
 Inherits IBase  
  
 Shadows Sub F(i As Integer)  
End Interface   
  
Interface IRight  
 Inherits IBase  
  
 Sub G()  
End Interface   
  
Interface IDerived  
 Inherits ILeft, IRight   
End Interface   
  
Class Test  
 Sub H(d As IDerived)  
 d.F(1) ' Invokes ILeft.F.  
 CType(d, IBase).F(1) ' Invokes IBase.F.  
 CType(d, ILeft).F(1) ' Invokes ILeft.F.  
 CType(d, IRight).F(1) ' Invokes IBase.F.  
 End Sub   
End Class

虽然 IBase.F 似乎未在通过 IRight 的访问路径中隐藏，调用 d.F(1) 时仍会选择 ILeft.F。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。

### 隐藏

派生类型通过重新声明继承的类型成员的名称来隐藏该名称。隐藏某个名称的操作并不会移除具有该名称的继承的类型成员；该操作只是使所有具有该名称的继承的类型成员在派生类中不可用。隐藏的声明可以是任何类型的实体。

可重载的实体可以选择两种隐藏形式。使用 Shadows 关键字指定按名称隐藏。按名称隐藏的实体可以隐藏基类中具有该名称的任何项，包括所有的重载。使用 Overloads 关键字指定按名称和签名隐藏。按名称和签名隐藏的实体隐藏具有该名称并与该实体具有相同签名的任何项。例如：

Class Base  
 Sub F()  
 End Sub  
  
 Sub F(i As Integer)  
 End Sub  
  
 Sub G()  
 End Sub  
  
 Sub G(i As Integer)  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 ' Only hides F(Integer).  
 Overloads Sub F(i As Integer)  
 End Sub  
  
 ' Hides G() and G(Integer).  
 Shadows Sub G(i As Integer)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As New Derived()  
  
 x.F() ' Calls Base.F().  
 x.G() ' Error: Missing parameter.  
 End Sub  
End Module

按名称和签名隐藏带 ParamArray 参数的方法时，只隐藏单个签名，而不是所有可能的扩展签名。即使隐藏方法的签名与被隐藏方法的未扩展签名相匹配时也是如此。以下示例

Class Base  
 Sub F(ParamArray x() As Integer)  
 Console.WriteLine("Base")  
 End Sub  
End Class  
  
Class Derived   
 Inherits Base  
  
 Overloads Sub F(x() As Integer)  
 Console.WriteLine("Derived")  
 End Sub  
End Class  
  
Module Test  
 Sub Main  
 Dim d As New Derived()  
 d.F(10)  
 End Sub  
End Module

输出 Base（即使 Derived.F 与 Base.F 的非扩展形式的签名相同）。

相反，带 ParamArray 参数的方法只隐藏与之具有相同签名的方法，而不是所有具有可能的扩展签名的方法。以下示例

Class Base  
 Sub F(x As Integer)  
 Console.WriteLine("Base")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Overloads Sub F(ParamArray x() As Integer)  
 Console.WriteLine("Derived")  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim d As New Derived()  
 d.F(10)  
 End Sub  
End Module

输出 Base（即使 Derived.F 的扩展形式与 Base.F 具有相同的签名）。

如果对未指定 Shadows 或 Overloads 的隐藏方法或属性声明了 Overrides，则该隐藏方法或属性采用 Overloads，否则采用 Shadows。如果重载实体集的一个成员指定了 Shadows 或 Overloads 关键字，则该实体集的所有成员都必须指定该关键字。不能同时指定 Shadows 和 Overloads 关键字。不能在标准模块中指定 Shadows 或 Overloads；标准模块中的成员隐式隐藏从 Object 继承的成员。

隐藏通过接口继承进行多重继承（因此不可用）的类型成员的名称是有效的，这样使该名称在派生接口中可用。

例如：

Interface ILeft  
 Sub F()  
End Interface  
  
Interface IRight  
 Sub F()  
End Interface  
  
Interface ILeftRight  
 Inherits ILeft, IRight  
  
 Shadows Sub F()  
End Interface  
  
Module Test  
 Sub G(i As ILeftRight)  
 i.F() ' Calls ILeftRight.F.  
 CType(i, ILeft).F() ' Calls ILeft.F.  
 CType(i, IRight).F() ' Calls IRight.F.  
 End Sub  
End Module

因为方法可以隐藏被继承方法，所以，类可以包含多个签名相同的 Overridable 方法。这不会造成多义性问题，因为只有派生程度最大的那个方法是可见的。在下面的示例中，C 和 D 类包含两个签名相同的 Overridable 方法：

Class A  
 Public Overridable Sub F()  
 Console.WriteLine("A.F")  
 End Sub   
End Class   
  
Class B  
 Inherits A  
  
 Public Overrides Sub F()  
 Console.WriteLine("B.F")  
 End Sub   
End Class   
  
Class C  
 Inherits B  
  
 Public Shadows Overridable Sub F()  
 Console.WriteLine("C.F")  
 End Sub   
End Class   
  
Class D  
 Inherits C  
  
 Public Overrides Sub F()  
 Console.WriteLine("D.F")  
 End Sub   
End Class   
  
Module Test  
 Sub Main()  
 Dim d As New D()  
 Dim a As A = d  
 Dim b As B = d  
 Dim c As C = d  
 a.F()  
 b.F()  
 c.F()  
 d.F()  
 End Sub   
End Module

示例中有两个 Overridable 方法：一个由类 A 引入，另一个由类 C 引入。类 C 引入的方法将隐藏从类 A 继承的方法。因此，类 D 中的 Overrides 声明会重写类 C 引入的方法，类 D 不可能重写类 A 引入的方法。此例产生输出：

B.F  
B.F  
D.F  
D.F

通过未隐藏 Overridable 方法的派生程度最小的类型来访问类 D 的实例，可以调用该被隐藏方法。

隐藏 MustOverride 方法是无效的，因为大多数情况下该操作会使类不可用。例如：

MustInherit Class Base  
 Public MustOverride Sub F()  
End Class  
  
MustInherit Class Derived  
 Inherits Base  
  
 Public Shadows Sub F()  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 ' Error: MustOverride method Base.F is not overridden.  
End Class

在本示例中，需要类 MoreDerived 来重写 MustOverride 方法 Base.F，但是因为类 Derived 隐藏了 Base.F，所以该操作是不可行的。无法为 Derived 声明有效的子代。

与隐藏外部范围中的名称不同，隐藏继承范围中的可访问名称会导致发出警告，如下面的示例所示：

Class Base  
 Public Sub F()  
 End Sub  
  
 Private Sub G()  
 End Sub   
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Sub F() ' Warning: shadowing an inherited name.  
 End Sub  
  
 Public Sub G() ' No warning, Base.G is not accessible here.  
 End Sub  
End Class

在类 Derived 中声明方法 F 会导致警告。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 类引入了该类的早期版本中不存在的方法 F，就可能发生上述情况。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 Shadows 或Overloads 修饰符可以消除因隐藏继承的名称导致的警告：

Class Base  
 Public Sub F()  
 End Sub   
End Class   
  
Class Derived  
 Inherits Base  
  
 Public Shadows Sub F() 'OK.  
 End Sub  
End Class

Shadows 修饰符指示要隐藏继承的成员。如果没有可隐藏的类型成员名称，指定 Shadows 或 Overloads 修饰符不会导致错误。

在声明一个新成员时，仅在该新成员的范围内隐藏继承的成员，如下面的示例所示：

Class Base  
 Public Shared Sub F()  
 End Sub   
End Class   
  
Class Derived  
 Inherits Base  
  
 Private Shared Shadows Sub F() ' Shadows Base.F in class Derived only.  
 End Sub   
End Class   
  
Class MoreDerived  
 Inherits Derived  
  
 Shared Sub G()  
 F() ' Invokes Base.F.  
 End Sub   
End Class

在上面的示例中，Derived 类的方法 F 的声明隐藏了继承自 Base 类的方法 F，但因为 Derived 类的新方法 F 具有 Private 访问类型，所以其作用域不会扩展到 MoreDerived 类。因此，MoreDerived.G 中调用 F() 是有效的并将调用 Base.F。对于重载的类型成员，将整个重载的类型成员集视为所有成员都具有最高访问级别，以便于进行隐藏。

Class Base  
 Public Sub F()  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Private Shadows Sub F()  
 End Sub  
  
 Public Shadows Sub F(i As Integer)  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 Public Sub G()  
 F() ' Error. No accessible member with this signature.  
 End Sub  
End Class

在此示例中，即使在 Derived 中声明 F() 时使用的是 Private 访问类型，声明重载 F(Integer) 时也使用 Public 访问类型。因此，为便于隐藏，将 Derived 中的名称 F 视为 Public，因此两种方法都隐藏 Base 中的 F。

## 实现

当某个类型声明其实现一个接口，并且该类型实现该接口的所有类型成员时，会存在一个实现关系。实现某个特定接口的类型可以转换为该接口。无法实例化接口，但声明接口变量是有效的；对这些变量所赋的值只能是实现该接口的类的值。例如：

Interface ITestable  
 Function Test(value As Byte) As Boolean  
End Interface  
  
Class TestableClass  
 Implements ITestable  
  
 Function Test(value As Byte) As Boolean Implements ITestable.Test  
 Return value > 128  
 End Function  
End Class  
  
Module Test  
 Sub F()  
 Dim x As ITestable = New TestableClass  
 Dim b As Boolean  
  
 b = x.Test(34)  
 End Sub  
End Module

即使某些方法不能从正在实现的派生接口直接访问，实现接口（具有多重继承的类型成员）的类型也必须仍然实现这些方法。例如：

Interface ILeft  
 Sub Test()  
End Interface  
  
Interface IRight  
 Sub Test()  
End Interface  
  
Interface ILeftRight  
 Inherits ILeft, IRight  
End Interface  
  
Class LeftRight  
 Implements ILeftRight  
  
 ' Has to reference ILeft explicitly.  
 Sub TestLeft() Implements ILeft.Test  
 End Sub  
  
 ' Has to reference IRight explicitly.  
 Sub TestRight() Implements IRight.Test  
 End Sub  
  
 ' Error: Test is not available in ILeftRight.  
 Sub TestLeftRight() Implements ILeftRight.Test  
 End Sub  
End Class

虽然 MustInherit 类必须提供对实现的接口的所有成员的实现；但是，通过将方法声明为 MustOverride，这些类可以延迟实现这些方法。例如：

Interface ITest  
 Sub Test1()  
 Sub Test2()  
End Interface  
  
MustInherit Class TestBase  
 Implements ITest  
  
 ' Provides an implementation.  
 Sub Test1() Implements ITest.Test1  
 End Sub  
  
 ' Defers implementation.  
 MustOverride Sub Test2() Implements ITest.Test2  
End Class  
  
Class TestDerived  
 Inherits TestBase  
  
 ' Have to implement MustOverride method.  
 Overrides Sub Test2()  
 End Sub  
End Class

某个类型可以选择重新实现其基类型实现的接口。要重新实现该接口，该类型必须显式声明实现该接口。重新实现接口的类型可以选择只重新实现部分（而不是全部）接口成员–任何未被重新实现的成员将继续使用基类型的实现。例如：

Class TestBase  
 Implements ITest  
  
 Sub Test1() Implements ITest.Test1  
 Console.WriteLine("TestBase.Test1")  
 End Sub  
  
 Sub Test2() Implements ITest.Test2  
 Console.WriteLine("TestBase.Test2")  
 End Sub  
End Class  
  
Class TestDerived  
 Inherits TestBase  
 Implements ITest ' Required to re-implement  
  
 Sub DerivedTest1() Implements ITest.Test1  
 Console.WriteLine("TestDerived.DerivedTest1")  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim Test As ITest = New TestDerived()  
 Test.Test1()  
 Test.Test2()  
 End Sub  
End Module

此示例输出：

TestDerived.DerivedTest1  
TestBase.Test2

如果派生类型实现的接口的基接口是通过该派生类型的基类型实现的，则派生类型可以选择仅实现该接口中尚未由基类型实现的类型成员。例如：

Interface IBase  
 Sub Base()  
End Interface  
  
Interface IDerived  
 Inherits IBase  
  
 Sub Derived()  
End Interface  
  
Class Base  
 Implements IBase  
  
 Public Sub Base() Implements IBase.Base  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
 Implements IDerived  
  
 ' Required: IDerived.Derived not implemented by Base.  
 Public Sub Derived() Implements IDerived.Derived  
 End Sub  
End Class

在基类型中使用可重写方法也可以实现接口方法。在这种情况下，派生类型也可以重写可重写的方法，从而更改接口的实现。例如：

Class Base  
 Implements ITest  
  
 Public Sub Test1() Implements ITest.Test1  
 Console.WriteLine("TestBase.Test1")  
 End Sub  
  
 Public Overridable Sub Test2() Implements ITest.Test2  
 Console.WriteLine("TestBase.Test2")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 ' Overrides base implementation.  
 Public Overrides Sub Test2()  
 Console.WriteLine("TestDerived.Test2")  
 End Sub  
End Class

### 实现方法

通过为方法提供 Implements 子句，类型可以为已实现的接口实现类型成员。两个类型成员必须具有相同数量的参数，参数的所有类型和修饰符也必须匹配（包括可选参数的默认值），返回类型必须匹配，方法参数的所有约束也必须匹配。例如：

Interface ITest  
 Sub F(ByRef x As Integer)  
 Sub G(Optional y As Integer = 20)  
 Sub H(Paramarray z() As Integer)  
End Interface  
  
Class Test  
 Implements ITest  
  
 ' Error: ByRef/ByVal mismatch.  
 Sub F(x As Integer) Implements ITest.F  
 End Sub  
  
 ' Error: Defaults do not match.  
 Sub G(Optional y As Integer = 10) Implements ITest.G  
 End Sub  
  
 ' Error: Paramarray does not match.  
 Sub H(z() As Integer) Implements ITest.H  
 End Sub  
End Class

如果类型成员符合上述所有标准，则单个方法可以实现任意数量的接口类型成员。例如：

Interface ITest  
 Sub F(i As Integer)  
 Sub G(i As Integer)  
End Interface  
  
Class Test

Implements ITest  
  
 Sub F(i As Integer) Implements ITest.F, ITest.G  
 End Sub  
End Class

实现常规接口的某个方法时，实现的方法必须提供与该接口的类型参数相对应的类型参数。例如：

Interface I1(Of U, V)   
 Sub M(x As U, y As List(Of V))   
End Interface  
  
Class C1(Of W, X)  
 Implements I1(Of W, X)  
  
 ' W corresponds to U and X corresponds to V  
 Public Sub M(x As W, y As List(Of X)) Implements I1(Of W, X).M  
 End Sub   
End Class  
  
Class C2  
 Implements I1(Of String, Integer)  
  
 ' String corresponds to U and Integer corresponds to V  
 Public Sub M(x As String, y As List(Of Integer)) \_  
 Implements I1(Of String, Integer).M  
 End Sub  
End Class

请注意，对于某些类型实参集来说，泛型接口可能是无法实现的。

Interface I1(Of T, U)  
 Sub S1(x As T)  
 Sub S1(y As U)  
End Interface  
  
Class C1  
 ' Unable to implement because I1.S1 has two identical signatures  
 Implements I1(Of Integer, Integer)  
End Class

## 多态性

多态性可以使方法或属性的实现多样化。使用多态性，相同的方法或属性可以执行不同的操作，具体取决于调用该方法或属性的实例的运行时类型。方法或属性的多态性称为可重写性。相反，不可重写的方法或属性的实现是固定的；不论是针对在其中声明方法或属性的类的实例调用方法或属性，还是针对派生类的实例调用方法或属性，实现都是相同的。当调用不可重写的方法或属性时，实例的编译时类型是决定因素。例如：

Class Base  
 Public Overridable Property X() As Integer  
 Get  
 End Get  
  
 Set  
 End Set  
 End Property  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Overrides Property X() As Integer  
 Get  
 End Get  
  
 Set  
 End Set  
 End Property  
End Class  
  
Module Test  
 Sub F()  
 Dim Z As Base  
  
 Z = New Base()  
 Z.X = 10 ' Calls Base.X  
 Z = New Derived()  
 Z.X = 10 ' Calls Derived.X  
 End Sub  
End Module

可重写方法也可以是 MustOverride，这意味着方法不提供方法体，必须将其重写。只有在 MustInherit 类中才允许使用 MustOverride。

在下面的示例中，Shape 类定义了可以绘制自身的几何形状对象的抽象表示形式：

MustInherit Public Class Shape  
 Public MustOverride Sub Paint(g As Graphics, r As Rectangle)  
End Class   
  
Public Class Ellipse  
 Inherits Shape  
  
 Public Overrides Sub Paint(g As Graphics, r As Rectangle)  
 g.drawEllipse(r)  
 End Sub   
End Class   
  
Public Class Box  
 Inherits Shape  
  
 Public Overrides Sub Paint(g As Graphics, r As Rectangle)  
 g.drawRect(r)  
 End Sub   
End Class

Paint 方法是 MustOverride，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。因为这些类不是 MustInherit 类，所以它们需要重写 Paint 方法并提供实际的实现。

基访问引用 MustOverride 方法是错误的，如下例所示：

MustInherit Class A  
 Public MustOverride Sub F()  
End Class  
  
Class B  
 Inherits A  
  
 Public Overrides Sub F()  
 MyBase.F() ' Error, MyBase.F is MustOverride.  
 End Sub   
End Class

MyBase.F() 调用会报告错误，因为它引用了 MustOverride 方法。

### 重写的方法

通过声明具有相同名称和签名的方法并用 Overrides 修饰符标记声明，类型可重写 继承的可重写方法。下面列出了对重写方法的附加要求。Overridable 方法声明用于引入新方法，而 Overrides 方法声明用于替换继承的方法实现。

重写方法可以声明为 NotOverridable，这样可防止在派生类型中对该方法进行进一步重写。实际上，NotOverridable 方法在任何进一步派生的类中都变为不可重写。

请看下面的示例：

Class A  
 Public Overridable Sub F()  
 Console.WriteLine("A.F")  
 End Sub  
  
 Public Overridable Sub G()  
 Console.WriteLine("A.G")  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Public Overrides NotOverridable Sub F()  
 Console.WriteLine("B.F")  
 End Sub  
  
 Public Overrides Sub G()  
 Console.WriteLine("B.G")  
 End Sub  
End Class  
  
Class C  
 Inherits B  
  
 Public Overrides Sub G()  
 Console.WriteLine("C.G")  
 End Sub  
End Class

在此示例中，类 B 提供两个 Overrides 方法：具有 NotOverridable 修饰符的方法 F 以及没有该修饰符的方法 G。通过使用 NotOverridable 修饰符可以防止类 C 进一步重写方法 F。

重写方法也可以声明为 MustOverride，即使该方法要重写的方法未声明为 MustOverride 也是如此。这需要包含类声明为 MustInherit，并且未声明为 MustInherit 的所有进一步派生的类都必须重写该方法。例如：

Class A  
 Public Overridable Sub F()  
 Console.WriteLine("A.F")  
 End Sub  
End Class  
  
MustInherit Class B  
 Inherits A  
  
 Public Overrides MustOverride Sub F()  
End Class

在该示例中，类 B 用 MustOverride 方法重写 A.F。这意味着派生自 B 的所有类都必须重写 F，除非这些类也声明为 MustInherit。

除非下列所有项对于一个重写方法皆为真，否则将会出现编译时错误：

1. 声明上下文包含一个单个可访问的继承方法，该方法具有与重写方法相同的签名和返回类型（如果存在）。
2. 正在被重写的继承的方法是可重写的。换言之，要重写的继承方法不是 Shared 或 NotOverridable。
3. 声明的方法的可访问域与被重写的继承方法的可访问域相同。有一种例外情况：如果要重写的方法在另一个程序集中，而重写方法对该程序集没有 Friend 访问权限，则必须由 Protected 方法重写 Protected Friend 方法。
4. 重写方法的形参与被重写方法的形参在 ByVal、ByRef、ParamArray 和 Optional 修饰符的使用上是一致的，包括为可选形参提供的值也是相同的。
5. 重写方法的类型参数与被重写方法的类型参数在类型约束方面必须相匹配。

重写基泛型类型中的方法时，重写方法必须提供与基类型形参对应的类型实参。例如：

Class Base(Of U, V)   
 Public Overridable Sub M(x As U, y As List(Of V))   
 End Sub  
End Class  
  
Class Derived(Of W, X)  
 Inherits Base(Of W, X)  
  
 ' W corresponds to U and X corresponds to V  
 Public Overrides Sub M(x As W, y As List(Of X))   
 End Sub   
End Class  
  
Class MoreDerived  
 Inherits Derived(Of String, Integer)  
  
 ' String corresponds to U and Integer corresponds to V  
 Public Overrides Sub M(x As String, y As List(Of Integer))  
 End Sub  
End Class

请注意，对于某些类型参数集来说，可能无法重写常规类中的可重写方法。如果方法声明为 MustOverride，意味着某些继承链是无法实现的。例如：

MustInherit Class Base(Of T, U)  
 Public MustOverride Sub S1(x As T)  
 Public MustOverride Sub S1(y As U)  
End Class  
  
Class Derived  
 Inherits Base(Of Integer, Integer)  
  
 ' Error: Can't override both S1's at once  
 Public Overrides Sub S1(x As Integer)  
 End Sub  
End Class

一个重写声明可以使用基访问来访问被重写的基方法，如下面的示例所示：

Class Base  
 Private x As Integer  
  
 Public Overridable Sub PrintVariables()  
 Console.WriteLine("x = " & x)  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Private y As Integer  
  
 Public Overrides Sub PrintVariables()  
 MyBase.PrintVariables()  
 Console.WriteLine("y = " & y)  
 End Sub  
End Class

在此示例中，类 Derived 中的调用 MyBase.PrintVariables() 调用了在类 Base 中声明的 PrintVariables 方法。基访问禁用可重写调用机制，仅仅将基方法视为不可重写的方法。如果对 Derived 中的调用写入 CType(Me, Base).PrintVariables()，则该调用将以递归方式调用在 Derived 中声明的 PrintVariables 方法，而不是调用在 Base 中声明的该方法。

仅当方法包含 Overrides 修饰符时，才能重写另一个方法。在所有其他情况下，与继承的方法具有相同签名的方法仅隐藏继承的方法，如下面的示例所示：

Class Base  
 Public Overridable Sub F()  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Overridable Sub F() ' Warning, shadowing inherited F().  
 End Sub  
End Class

在该示例中，类 Derived 中的 F 方法不包含 Overrides 修饰符，因此不重写 Base 类中的 F 方法。相反，Derived 类中的 F 方法隐藏类 Base 中的方法，并且由于该声明中没有包含 Shadows 或 Overloads 修饰符，因此会报出警告。

在下面的示例中，类 Derived 中的方法 F 隐藏继承自类 Base 的可重写方法 F：

Class Base  
 Public Overridable Sub F()  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Private Shadows Sub F() ' Shadows Base.F within Derived.  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 Public Overrides Sub F() ' Ok, overrides Base.F.  
 End Sub  
End Class

由于类 Derived 中的新方法 F 具有 Private 访问权限，它的范围只包括 Derived 的类体而没有延伸到类 MoreDerived。因此，类 MoreDerived 中的方法 F 的声明可以重写继承自类 Base 的方法 F。

当调用 Overridable 方法时，将根据实例的类型，调用实例方法的派生程度最大的实现（不论调用的方法来自基类还是派生类）。就类 R 而言 Overridable 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 Overridable 声明，则这是 M 派生程度最大的实现。

否则，如果 R 包含 M 的重写，则这是 M 的派生程度最重的实现。

否则，M 的派生程度最重的实现与 R 的直接基类的派生程度最重的实现相同。

## 可访问性

声明指定所声明实体的可访问性。实体的可访问性不会更改实体名称的范围。声明的可访问域是声明的实体在其中可以访问的所有声明空间的集合。

五个访问类型为 Public、Protected、Friend、Protected Friend 和 Private。Public 是对访问权限要求最低的访问类型，其他四个访问类型都是 Public 的子集。对访问权限要求最高的访问类型是 Private，其他四个访问类型都是 Private 的超集。

声明的访问类型通过可选访问修饰符来指定，这些修饰符可以是 Public、Protected、Friend、Private 或 Protected 和 Friend 的组合。如果未指定访问修饰符，则默认的访问类型取决于声明上下文；允许的访问类型也取决于声明上下文。

使用 Public 修饰符声明的实体具有 Public 访问类型。对 Public 实体的使用没有限制。

使用 Protected 修饰符声明的实体具有 Protected 访问类型。只能对类成员（包括常规类型成员和嵌套类）或标准模块和结构的 Overridable 成员（按定义必须继承自 System.Object 或 System.ValueType）指定 Protected 访问类型。派生类可以访问 Protected 成员，条件是该成员不是实例成员，或者访问是通过派生类的实例进行的。Protected 访问类型不是 Friend 访问类型的超集。

使用 Friend 修饰符声明的实体具有 Friend 访问类型。对于具有 Friend 访问类型的实体而言，只有在包含该实体声明的程序中，或在任何通过 System.Runtime.CompilerServices.InternalsVisibleToAttribute 特性指定了 Friend 访问类型的程序集中，才可以进行访问。

使用 Protected Friend 修饰符声明的实体兼有 Protected 和 Friend 访问类型。

使用 Private 修饰符声明的实体具有 Private 访问类型。Private 实体（包括所有嵌套实体）只能在其声明上下文中访问。

声明中的可访问性不取决于声明上下文的可访问性。例如，使用 Private 访问类型声明的类型可以包含具有 Public 访问类型的类型成员。

下面的代码示例演示了各种可访问域：

Public Class A  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
End Class  
  
Friend Class B  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
  
 Public Class C  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
 End Class  
  
 Private Class D  
 Public Shared X As Integer  
 Friend Shared Y As Integer  
 Private Shared Z As Integer  
 End Class  
End Class

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 的可访问域是包含它们的程序。

A.Z 的可访问域是 A.

B.Z、B.D、B.D.X 和 B.D.Y 的可访问域是 B，包括 B.C 和 B.D。

B.C.Z 的可访问域是 B.C。

B.D.Z 的可访问域是 B.D。

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 Public 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。

必须通过派生类型的实例才能访问 Protected 实例成员，这样不相关的类型就无法访问彼此的受保护成员。例如：

Class User  
 Protected Password As String  
End Class  
  
Class Employee  
 Inherits User  
End Class  
  
Class Guest  
 Inherits User  
  
 Public Function GetPassword(u As User) As String  
 ' Error: protected access has to go through derived type.  
 Return U.Password  
 End Function  
End Class

在上面的示例中，只有使用 Guest 的实例对受保护的 Password 字段进行限定时，Guest 类才能访问该字段。这能防止 Guest 访问 Employee 对象的 Password 字段（否则将该类强制转换为 User 就能访问该字段）。

为便于在泛型类型中访问 Protected 成员，声明上下文包含类型形参。这意味着，具有一个类型实参集的派生类型不能访问具有不同类型实参集的派生类型的 Protected 成员。例如：

Class Base(Of T)  
 Protected x As T  
End Class  
  
Class Derived(Of T)  
 Inherits Base(Of T)  
  
 Public Sub F(y As Derived(Of String))  
 ' Error: Derived(Of T) cannot access Derived(Of String)'s   
 ' protected members  
 y.x = "a"  
 End Sub  
End Class

批注

C# 语言（可能还有其他语言）允许泛型类型访问 Protected 成员，而不论提供的是哪些类型实参。在设计包含 Protected 成员的泛型类时，请务必注意这一点。

AccessModifier ::= Public | Protected | Friend | Private | Protected Friend

### 构成类型

声明的构成类型是由声明所引用的类型。例如，常量类型、方法的返回类型以及构造函数的参数类型都是构成类型。声明的构成类型的可访问域必须与声明本身的可访问域相同，或前者必须是后者的超集。例如：

Public Class X  
 Private Class Y  
 End Class  
  
 ' Error: Exposing private class Y outside of X.  
 Public Function Z() As Y  
 End Function  
  
 ' Valid: Not exposing outside of X.  
 Private Function A() As Y  
 End Function  
End Class  
  
Friend Class B  
 Private Class C  
 End Class  
  
 ' Error: Exposing private class Y outside of B.  
 Public Function D() As C  
 End Function  
End Class

## 类型和名称空间名称

很多语言需要制定名称空间或类型，方法是通过使用名称空间或类型的名称的限定形式。限定的名称包含一系列的由句号分隔的标识符；句号右侧的标识符在声明空间中进行解析，声明空间由句号左侧的标识符来指定。

名称空间或类型的完全限定名称是一个限定名称，该名称包含所有包含的名称空间和类型的名称。换言之，命名空间或类型的完全限定名是 N.T，其中 T 是实体的名称，N 是包含实体的完全限定名。

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（在同一行内作为注释）。

Class A ' A.  
End Class  
  
Namespace X ' X.  
 Class B ' X.B.  
 Class C ' X.B.C.  
 End Class  
 End Class  
  
 Namespace Y ' X.Y.  
 Class D ' X.Y.D.  
 End Class  
 End Namespace   
End Namespace   
  
Namespace X.Y ' X.Y.  
 Class E ' X.Y.E.  
 End Class  
End Namespace

可看到已在源代码中的两个不同位置声明命名空间 X.Y，但这两个分部声明仅构成单个名为 X.Y 的命名空间，其中同时包含类 D 和类 E。

在某些情况下，限定名可能以关键字 Global 开头。该关键字表示未命名的最外层命名空间，在声明隐藏外层命名空间的情况下很有用。在此情况下，Global 关键字允许“转义”到最外层命名空间。例如：

Namespace NS1  
 Class System  
 End Class  
  
 Module Test  
 Sub Main()  
 ' Error: Class System does not contain Int32  
 Dim x As System.Int32  
  
  
 ' Legal, binds to System in outermost namespace  
 Dim y As Global.System.Int32  
 End Sub  
 End Module  
End Namespace

在上面的示例中，第一个方法调用是无效的，因为标识符 System 绑定到类 System，而不是绑定到命名空间 System。访问 System 命名空间的唯一途径是使用 Global 转到最外层命名空间。Imports 语句或 Namespace 声明中不能使用 Global。

因为其他语言可以引入与该语言中的关键字匹配的类型或命名空间，Visual Basic 在关键字后跟一个句号时，就认为该关键字是限定名称的一部分。将以这种方式使用的关键字视为标识符。例如，限定标识符 X.Default.Class 是一个有效的限定标识符，而 Default.Class 不是有效的限定标识符。

### 用于命名空间和类型的限定名称解析

给定一个 N.R(Of A) 形式的限定命名空间名或类型名，其中 R 是限定名中最右端的标识符，A 是可选类型实参列表，下面的步骤说明如何确定限定名引用的命名空间或类型：

1. 使用用于限定或非限定名称解析的规则解析 N。
2. 如果 N 的解析失败或解析为类型形参，则会发生编译时错误。
3. 否则，如果 R 与 N 中某一命名空间的名称匹配并且未提供类型实参，或者 R 与 N 中的可访问类型（该类型拥有与类型实参同等数目的类型形参）的相匹配，则限定名将引用该命名空间或类型。
4. 否则，如果 N 包含一个或多个标准模块，而 R 与仅存在于一个模块的可访问类型（如果有实参，其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则该限定名引用该类型。如果 R 与多个标准模块中的可访问类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则会发生编译时错误。
5. 其他情况下，将发生编译时错误。

注意   此解析进程假设解析命名空间或类型名称时，类型成员不隐藏命名空间或类型。

### 用于命名空间和类型的非限定名称解析

给定一个非限定名 R(Of A)，其中 A 是可选类型实参列表，下面的步骤说明如何确定非限定名引用的命名空间或类型：

1. 如果 R 与当前方法的类型形参的名称相匹配，并且没有提供类型实参，则非限定的名将引用该类型形参。
2. 对于每个包含名称引用的嵌套类型，顺序是从最内层类型到最外层类型：
3. 如果 R 与当前类型中的类型形参的名称相匹配，并且没有提供类型实参，则非限定的名将引用该类型形参。
4. 否则，如果 R 与可访问嵌套类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则该非限定名引用该类型。
5. 对于包含名称引用的每个嵌套的命名空间，顺序是从最内层的命名空间到最外层命名空间：
6. 如果 R 与当前命名空间中的嵌套命名空间的名称相匹配，并且未提供类型实参列表，则该非限定名引用该嵌套命名空间。
7. 否则，如果 R 与可访问类型（该类型与类型实参拥有同等数目的类型形参）的名称相匹配，如果当前命名空间中存在这样的类型，则非限定名将引用该类型。
8. 否则，如果命名空间包含一个或多个可访问标准模块，并且 R 与可访问嵌套类型（该类型与类型实参拥有同等数目的类型形参）的名称相匹配，如果这样的嵌套类型只在一个标准模块中存在，则非限定名将引用该嵌套类型。如果 R 与多个标准模块中的可访问嵌套类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则会发生编译时错误。
9. 如果源文件具有一个或多个导入别名，并且 RR 与其中一个导入别名的名称相匹配，则该非限定名引用该导入别名。如果提供了类型实参列表，则将发生编译时错误。
10. 如果包含名称引用的源文件具有一个或多个导入：
11. 如果 R 与仅存在于一个导入中的可访问类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则该非限定名引用该类型。如果 R 与多个导入中的可访问类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，而这些类型各不相同，则会发生编译时错误。
12. 否则，如果未提供类型实参列表，并且 R 与仅存在于一个导入中的具有可访问类型的命名空间的名称相匹配，则该非限定名引用该命名空间。如果未提供类型实参列表，而 R 与多个导入中的具有可访问类型的命名空间的名称相匹配，并且这些命名空间不是同一个命名空间，则会发生编译时错误。
13. 否则，如果导入中包含一个或多个可访问标准模块，并且 R 与可访问嵌套类型（该类型与类型实参拥有相同数目的类型形参）的名称相匹配，并且该类型存在于一个模块中，则非限定名将引用该类型。如果 R 与多个标准模块中的可访问嵌套类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则会发生编译时错误。
14. 如果编译环境定义一个或多个导入别名，并且 R 与其中一个别名的名称相匹配，则该非限定名引用该导入别名。如果提供了类型实参列表，则将发生编译时错误。
15. 如果编译环境定义一个或多个导入，则：
16. 如果 R 与仅存在于一个导入中的可访问类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则该非限定名引用该类型。如果 R 与多个导入中的可访问类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则会发生编译时错误。
17. 否则，如果未提供类型实参列表，并且 R 与仅存在于一个导入中的具有可访问类型的命名空间的名称相匹配，则该非限定名引用该命名空间。如果未提供类型实参列表，并且 R 与多个导入中的具有可访问类型的命名空间的名称相匹配，则会发生编译时错误。
18. 否则，如果导入中包含一个或多个可访问标准模块，并且 R 与可访问嵌套类型（该类型与类型实参拥有相同数目的类型形参）的名称相匹配，并且该类型存在于一个模块中，则非限定名将引用该类型。如果 R 与多个标准模块中的可访问嵌套类型（其类型形参数目与类型实参（如果有）数目相同）的名称相匹配，则会发生编译时错误。
19. 其他情况下，将发生编译时错误。

注意   此解析进程假设解析命名空间或类型名称时，类型成员不隐藏命名空间或类型。

一般来说，在特定的命名空间中，一个名称只能出现一次。但由于可以跨多个 .NET 程序集声明命名空间，因此可能会遇到两个程序集使用同一完全限定名来定义一个类型的情况。在此情况下，当前源文件集中声明的类型将优先于外部 .NET 程序集中声明的类型。否则，名称将出现无法消除的多义性。

## 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。因为 Visual Basic 是一种类型安全的语言，程序中的每个变量都具有一个类型，并且该语言保证存储在变量中的值总是具有合适的类型。在引用变量之前，变量总是初始化为其类型的默认值。访问未初始化的内存是不可行的。

## 泛型类型和方法

类型（除标准模块和枚举类型之外）和方法可以声明类型形参，该形参是在声明该类型的实例或调用该方法之后才能提供的类型。包含类型形参的类型和方法也各自称为泛型类型和泛型方法，因为必须以泛型方式写入该类型或方法，写入时不必包含由使用该类型或方法的代码提供的类型特定信息。

批注

此时，虽然方法和委托可以是泛型的，但属性、事件和运算符本身不能是泛型的。但它们可以使用来自包含的类中的类型形参。

从泛型类型或方法的角度来说，类型形参是占位符类型，将在使用类型或方法时用实际类型对其进行填充。使用类型或方法时，类型或方法中的类型实参将取代类型形参。例如，泛型堆栈类可以按如下方式实现：

Public Class Stack(Of ItemType)  
 Protected Items(0 To 99) As ItemType  
 Protected CurrentIndex As Integer = 0  
  
 Public Sub Push(data As ItemType)  
 If CurrentIndex = 100 Then  
 Throw New ArgumentException("Stack is full.")  
 End If  
  
 Items(CurrentIndex) = Data  
 CurrentIndex += 1  
 End Sub  
  
 Public Function Pop() As ItemType  
 If CurrentIndex = 0 Then  
 Throw New ArgumentException("Stack is empty.")  
 End If  
  
 CurrentIndex -= 1  
 Return Items(CurrentIndex + 1)   
 End Function  
End Class

使用 Stack(Of ItemType) 类的声明必须为类型形参 ItemType 提供类型实参。在该类中使用 ItemType 时即会填充该类型：

Option Strict On  
  
Module Test  
 Sub Main()  
 Dim s1 As New Stack(Of Integer)()  
 Dim s2 As New Stack(Of Double)()  
  
 s1.Push(10.10) ' Error: Stack(Of Integer).Push takes an Integer  
 s2.Push(10.10) ' OK: Stack(Of Double).Push takes a Double  
 Console.WriteLine(s2.Pop().GetType().ToString()) ' Prints: Double  
 End Sub  
End Module

### 类型形参

可以在声明类型或方法时提供类型形参。每个类型形参都是一个标识符，该标识符是类型实参的占位符，该类型实参在创建构造类型或方法时提供。相反，类型实参是实际的类型，用于在使用泛型类型或方法时取代类型形参。

类型或方法声明中的每个类型形参会在该类型或方法的声明空间中定义一个名称。因此，类型形参不能与其他类型形参、类型成员、方法形参或局部变量同名。一个类型或方法的类型形参的作用域是整个类型或方法。因为类型形参的作用域是整个类型声明，所以嵌套类型可以使用外部类型形参。这也表示在访问泛型类型的嵌套类型时，必须始终指定类型形参：

Public Class Outer(Of T)  
 Public Class Inner  
 Public Sub F(x As T)  
 ...  
 End Sub  
 End Class  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As New Outer(Of Integer).Inner()  
 ...  
 End Sub  
End Module

与类中的其他成员不同，类型形参是不能继承的。类型中的类型形参只能由其简单名称引用；换言之，类型形参不能使用包含它们的类型名进行限定。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(Of T)  
 Class Inner(Of T)  
 Public t1 As T ' Refers to Inner's T  
 End Class  
End Class

可以根据类型或方法所声明的类型形参的数目（或元数）对类型和方法进行重载。例如，下面的声明是合法的：

Module C  
 Sub M()  
 End Sub  
  
 Sub M(Of T)()  
 End Sub  
  
 Sub M(Of T, U)()  
 End Sub  
End Module  
  
Structure C(Of T)  
 Dim x As T  
End Structure  
  
Class C(Of T, U)  
End Class

重载类型时，将始终根据指定的类型实参的数目对重载进行匹配。在同一个程序中结合使用泛型和非泛型类时这是有用的：

Class Queue   
End Class   
   
Class Queue(Of T)  
End Class  
  
Class X  
 Dim q1 As Queue ' Non-generic queue  
 Dim q2 As Queue(Of Integer) ' Generic queue  
End Class

方法重载解决方案部分介绍针对类型形参进行方法重载的规则。

类型形参在所在的声明之内视为完整类型。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下述其他类型稍微不同的操作和限制：

不能直接使用类型形参声明基类或接口。

类型形参的成员查找规则取决于应用到该类型形参的约束（如果有）。

类型形参的可用转换取决于应用到类型形参的约束（如果有）。

如果没有 Structure 约束，可以使用 Is 和 IsNot 将类型形参所表示的类型的值与 Nothing 进行比较。

仅当类型形参受 New 或 Structure 约束的约束时，才能在 New 表达式中使用类型形参。

GetType 表达式的属性异常中不能使用类型形参。

类型形参可以作为其他泛型类型和泛型参数的类型实参。以下示例是一个扩展 Stack(Of ItemType) 类的泛型类型：

Class MyStack(Of ItemType)  
 Inherits Stack(Of ItemType)  
  
 Public ReadOnly Property Size() As Integer  
 Get  
 Return CurrentIndex  
 End Get  
 End Property  
End Class

当声明向 MyStack 提供一个类型实参时，也会向 Stack 提供该类型实参。

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，该运行时类型是通过向泛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非泛型类型或特定的构造类型。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。

TypeParameterList ::=  
 OpenParenthesis Of TypeParameters CloseParenthesis

TypeParameters ::=  
 TypeParameter |  
 TypeParameters Comma TypeParameter

TypeParameter ::=  
 [ VarianceModifier ] Identifier [ TypeParameterConstraints ]

VarianceModifier ::=  
 In | Out

### 类型约束

因为类型实参可以是类型系统中的任何类型，所以一个泛型类型或泛型方法不能对类型形参作任何假设。因此，类型形参的成员可视为类型 Object 的成员，因为所有类型都是从 Object 派生的。

对于 Stack(Of ItemType) 这样的集合来说，这不是特别重要的限制，但有时泛型类型可能希望对将作为类型实参提供的类型作出假设。可将类型约束应用于类型形参，以限制哪些类型可以作为类型形参提供，以便允许泛型类型或方法对类型形参作出更多假设。

Public Class DisposableStack(Of ItemType As IDisposable)  
 Implements IDisposable  
  
 Private \_items(0 To 99) As ItemType  
 Private \_currentIndex As Integer = 0  
  
 Public Sub Push(data As ItemType)  
 ...  
 End Sub  
  
 Public Function Pop() As ItemType  
 ...  
 End Function  
  
 Private Sub Dispose() Implements IDisposable.Dispose  
 For Each item As IDisposable In \_items  
 If item IsNot Nothing Then  
 item.Dispose()  
 End If  
 Next item  
 End Sub  
End Class

在此示例中，DisposableStack(Of ItemType) 将其类型形参约束为仅限于实现接口 System.IDisposable 的类型。因此，它可以实现 Dispose 方法，用于处理队列中留下的所有对象。

类型约束必须是特殊约束 Class、Structure 或 New 之一，或者必须是符合以下条件的类型 T：

T 是一个类、一个接口或一个类型形参。

T 不是 NotInheritable。

T 不是以下特殊类型之一，也不是以下特殊类型的派生类型：System.Array、System.Delegate、System.MulticastDelegate、System.Enum 或 System.ValueType。

T 不是 Object。所有类型都派生自 Object，这种约束（如果允许）是没有效果的。

T 的可访问性不低于要声明的类型或方法的可访问性。

可以为单个类型形参指定多个类型约束，方法是将类型约束括在大括号 ({}) 中。对于给定的类型形参，只有一个类型约束可以是类。将 Structure 特殊约束与命名类约束或 Class 特殊约束组合使用是错误的。

Class ControlFactory(Of T As {Control, New})  
 ...  
End Class

类型约束可以使用包含类型或包含类型的任何类型形参。在下面的示例中，类型约束要求提供的类型实参使用自身作为类型实参来实现一个泛型接口：

Class Sorter(Of V As IComparable(Of V))  
 ...  
End Class

特殊类型约束 Class 将提供的类型实参约束为任何引用类型。

批注

特殊类型约束 Class 可由接口来满足。而结构可以实现接口。因此，约束 (Of T As U, U As Class) 可以通过“T”结构（该结构不满足 Class 特殊约束）及其实现的接口“U”（该接口满足 Class 特殊约束）来满足。

特殊类型约束 Structure 将提供的类型实参约束为 System.Nullable(Of T) 以外的任何值类型。

批注

结构约束不允许 System.Nullable(Of T)，因此，不能将 System.Nullable(Of T) 作为类型实参提供给自身。

特殊类型约束 New 要求提供的类型实参必须具有可访问的无形参构造函数，并且不能声明为 MustInherit。例如：

Class Factory(Of T As New)  
 Function CreateInstance() As T  
 Return New T()  
 End Function  
End Class

类类型约束要求提供的类型实参必须是该类型或继承自该类型。接口类型约束要求提供的类型实参必须实现该接口。类型形参约束要求提供的类型实参必须派生自或用于实现针对匹配的类型形参提供的所有绑定。例如：

Class List(Of T)  
 Sub AddRange(Of S As T)(collection As IEnumerable(Of S))  
 ...  
 End Sub  
End Class

在此示例中，将 AddRange 的类型形参 S 约束为 List 的类型形参 T。这意味着，List(Of Control) 会将 AddRange 的类型形参约束为 Control 类型或从其继承的任何类型。

类型形参约束“Of S As T”的解析方式是将 T 的所有约束添加到 S 而不是使用特殊约束（Class、Structure、New）。循环约束（例如 Of S As T, T As S）是错误的用法。类型形参约束本身拥有 Structure 约束是错误的。添加约束之后，有几种可能出现的特殊情况：

如果存在多个类约束，则将派生程度最重的类视为约束。如果一个或多个类约束没有继承关系，则该约束不可满足，是错误的约束。

类型形参将 Structure 特殊约束与命名类约束或 Class 特殊约束组合使用是错误的用法。类约束可以是 NotInheritable，这种情况下，不接受该约束的任何派生类型，使用派生类型是错误的用法。

该类型可为以下特殊类型之一或继承自以下特殊类型：System.Array、System.Delegate、System.MulticastDelegate、System.Enum 或 System.ValueType。在这种情况下，只接受该类型或从其继承的类型。如果类型形参约束为这些类型之一，则只能使用 DirectCast 运算符所允许的转换。例如：

MustInherit Class Base(Of T)  
 MustOverride Sub S1(Of U As T)(x As U)  
End Class  
  
Class Derived  
 Inherits Base(Of Integer)  
  
 ' The constraint of U must be Integer, which is normally not allowed.  
 Overrides Sub S1(Of U As Integer)(x As U)  
 Dim y As Integer = x ' OK  
 Dim z As Long = x ' Error: Can't convert  
 End Sub  
End Class

另外，由于存在上述松弛法之一，约束为值类型的类型形参无法调用针对该值类型定义的任何方法。例如：

Class C1(Of T)  
 Overridable Sub F(Of G As T)(x As G)  
 End Sub  
End Class  
  
Class C2  
 Inherits C1(Of IntPtr)  
  
 Overrides Sub F(Of G As IntPtr)(ByVal x As G)  
 ' Error: Cannot access structure members  
 x.ToInt32()  
 End Sub  
End Class

如果进行替换后约束最后成为数组类型，则也会允许任何协变的数组类型。例如：

Module Test  
 Class B  
 End Class  
  
 Class D  
 Inherits B  
 End Class  
  
 Function F(Of T, U As T)(x As U) As T  
 Return x  
 End Function  
  
 Sub Main()  
 Dim a(9) As B  
 Dim b(9) As D  
  
 a = F(Of B(), D())(b)  
 End Sub  
End Module

将使用类或接口约束的类型形参视为与该类或接口约束具有相同的成员。如果类型形参包含多个约束，则将类型形参视为这些约束的所有成员的并集。如果多个约束中存在同名的成员，则按以下顺序隐藏这些成员：类约束隐藏接口约束中的成员，接口约束隐藏 System.ValueType 中的成员（如果指定了 Structure 约束），后一种约束隐藏 Object 中的成员。如果同名的成员出现在多个接口约束中，则该成员将不可用（与出现在多个接口继承中情况相同），并且类型形参必须强制转换为所需的接口。例如：

Class C1  
 Sub S1(x As Integer)  
 End Sub  
End Class  
  
Interface I1  
 Sub S1(x As Integer)  
End Interface  
  
Interface I2  
 Sub S1(y As Double)  
End Interface  
  
Module Test  
 Sub T1(Of T As {C1, I1, I2})()  
 Dim a As T  
 a.S1(10) ' Calls C1.S1, which is preferred  
 a.S1(10.10) ' Also calls C1.S1, class is still preferred  
 End Sub  
  
 Sub T2(Of T As {I1, I2})()  
 Dim a As T  
 a.S1(10) ' Error: Call is ambiguous between I1.S1, I2.S1  
 End Sub  
End Module

当作为类型实参提供类型形参时，该类型形参必须满足匹配的类型形参的约束。

Class Base(Of T As Class)  
End Class  
  
Class Derived(Of V)  
 ' Error: V does not satisfy the constraints of T  
 Inherits Base(Of V)  
End Class

约束类型形参的值可用于访问实例成员，包括该约束中指定的实例方法。

Interface IPrintable  
 Sub Print()  
End Interface  
  
Class Printer(Of V As IPrintable)  
 Sub PrintOne(v1 As V)  
 V1.Print()  
 End Sub  
End Class

TypeParameterConstraints ::=  
 As Constraint |  
 As OpenCurlyBrace ConstraintList CloseCurlyBrace

ConstraintList ::=  
 ConstraintList *Comma* Constraint |  
 Constraint

Constraint ::= TypeName | New | Structure | Class

### 类型形参差异

接口或委托类型声明中的类型形参可以选择指定差异修饰符。包含差异修饰符的类型形参限制如何在接口或委托类型中使用类型形参，但允许泛型接口或委托类型转换为包含与该差异兼容的类型实参的其他泛型类型。例如：

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As IEnumerable(Of Derived) = ...  
  
 ' OK, as IEnumerable(Of Base) is variant compatible  
 ' with IEnumerable(Of Derived)  
 Dim y As IEnumerable(Of Base) = x  
 End Sub  
End Module

对具有类型形参（包含差异修饰符）的泛型接口有以下限制：

这些接口不能包含指定形参列表的事件声明（但允许自定义事件声明或具有委托类型的事件声明）。

这些接口不能包含嵌套的类、结构或枚举类型。

批注

进行这些限制是因为嵌套在泛型类型中的类型隐式复制其父类型的泛型参数。对于嵌套的类、结构或枚举类型，其类型形参中不能包含差异标识符。对于带形参列表的事件声明，如果看似在 In 位置（即形参类型）使用的类型实际上在 Out 位置（即事件类型）使用，则生成的嵌套委托类可能会令人混淆。

类型形参（使用Out 修饰符声明）为协变。非正式地讲，协变类型形参只能在输出位置（即正在由接口或委托类型返回的值）使用，不能在输入位置中使用。在以下条件下，类型 T 被视为有效协变：

T 是一个类、结构或枚举类型。

T 是非泛型委托或接口类型。

T 是一个数组类型，并且该类型的元素类型是有效协变的。

T 是类型形参，并且未声明为 Out 类型形参。

T 是采用类型实参 A1,…,An 的构造接口或委托类型 X(Of P1,…,Pn)，性质如下：

如果 Pi 声明为Out 类型形参，则 Ai 是有效协变的。

如果 Pi 声明为In 类型形参，则 Ai 是有效逆变的。

在接口或委托类型中，以下项必须是有效协变的：

接口的基接口。

函数或委托类型的返回类型。

具有 Get 访问器的属性的类型。

任意 ByRef 参数的类型。

例如：

Delegate Function D(Of Out T, U)(x As U) As T  
  
Interface I1(Of Out T)  
End Interface  
  
Interface I2(Of Out T)  
 Inherits I1(Of T)  
  
 ' OK, T is only used in an Out position  
 Function M1(x As I1(Of T)) As T  
  
 ' Error: T is used in an In position  
 Function M2(x As T) As T  
End Interface

类型形参（使用In 修饰符声明）为逆变。非正式地讲，逆变类型形参只能在输入位置（即正在传入接口或委托类型的值）使用，而不能在输出位置使用。在以下条件下，类型 T 被视为有效逆变：

T 是一个类、结构或枚举类型。

T 是非泛型委托或接口类型。

T 是一个数组类型，并且其元素类型是有效逆变的。

T 是类型形参，并且未声明为In 类型形参。

T 是采用类型实参 A1,…,An 的构造接口或委托类型 X(Of P1,…,Pn)，性质如下：

如果 Pi 声明为 Out 类型形参，则 Ai 是有效逆变的。

如果 Pi 声明为 In 类型形参，则 Ai 是有效协变的。

在接口或委托类型中，以下项必须是有效逆变的：

参数的类型。

针对方法类型形参的类型约束。

具有 Set 访问器的属性的类型。

事件的类型。

例如：

Delegate Function D(Of T, In U)(x As U) As T  
  
Interface I1(Of In T)  
End Interface  
  
Interface I2(Of In T)  
 ' OK, T is only used in an In position  
 Sub M1(x As I1(Of T))  
  
 ' Error: T is used in an Out position  
 Function M2() As T  
End Interface

如果类型必须是有效逆变和协变的（如同时具有 Get 和 Set 访问器的属性，或具有 ByRef 参数的属性），则不能使用变化类型形参。

注意 Out 不是保留字。

批注

协变和逆变会引起“菱形多义性问题”。考虑下列代码：

类 C  
 Implements IEnumerable(Of String)  
 Implements IEnumerable(Of Exception)  
  
 Public Function GetEnumerator1() As IEnumerator(Of String) \_  
 Implements IEnumerable(Of String).GetEnumerator  
 Console.WriteLine(“string”)  
 End Function  
  
 Public Function GetEnumerator2() As IEnumerator(Of Exception) \_  
 Implements IEnumerable(Of Execption).GetEnumerator  
 Console.WriteLine(“exception”)  
 End Function  
End Class

Dim c As IEnumerable(Of Object) = New C  
c.GetEnumerator()

可通过两种方式将类 C 转换为 IEnumerable(Of Object)，通过从 IEnumerable(Of String) 进行的协变转换和通过从 IEnumerable(Of Exception) 进行的协变转换。CLR 不指定 c.GetEnumerator() 将调用这两种方法中的哪一种。通常，只要类声明用两个具有公共超类型的不同泛型实参（例如，在此示例中，String 和 Exception 具有公共超类型 Object）来实现协变接口，或类声明用两个具有公共子类型的不同泛型实参来实现逆变接口，就可能引发歧义性。编译器将针对此类声明发出警告。

# 特性

Visual Basic 语言使程序员能够在声明上指定修饰符，用以表示有关所声明实体的信息。例如，向类方法附加修饰符 Public、Protected、Friend、Protected Friend 或 Private 可以指定其可访问性。

除了定义修饰符以外，Visual Basic 语言还使程序员可以创建新的修饰符（称为特性），并在声明新实体时使用它们。这些新修饰符通过特性类的声明进行定义，然后通过特性块分配给实体。

注意   可在运行时通过 .NET Framework 的反射 API 检索特性。这些 API 不属于本规范的讨论范围。

例如，框架可以定义 Help 特性，该特性可置于类和方法等程序元素上，用以提供从程序元素到文档的映射，如下例所示：

<AttributeUsage(AttributeTargets.All)> \_  
Public Class HelpAttribute  
 Inherits Attribute  
  
 Public Sub New(urlValue As String)  
 Me.UrlValue = urlValue  
 End Sub  
  
 Public Topic As String  
 Private UrlValue As String  
  
 Public ReadOnly Property Url() As String  
 Get  
 Return UrlValue  
 End Get  
 End Property  
End Class

此示例定义了一个名为 HelpAttribute（或简称 Help）的特性类，该特性类具有一个位置形参 (UrlValue) 和一个命名实参 (Topic)。

下一示例演示该特性的多种用法：

<Help("http://www.example.com/.../Class1.htm")> \_  
Public Class Class1  
 <Help("http://www.example.com/.../Class1.htm", Topic:="F")> \_  
 Public Sub F()  
 End Sub  
End Class

下面的示例检查 Class1 是否具有 Help 特性；如果存在该特性，则写出关联的 Topic 和 Url 值。

Module Test  
 Sub Main()  
 Dim type As Type = GetType(Class1)  
 Dim arr() As Object = \_  
 type.GetCustomAttributes(GetType(HelpAttribute), True)  
  
 If arr.Length = 0 Then  
 Console.WriteLine("Class1 has no Help attribute.")  
 Else  
 Dim ha As HelpAttribute = CType(arr(0), HelpAttribute)  
 Console.WriteLine("Url = " & ha.Url & ", Topic = " & ha.Topic)  
 End If  
 End Sub  
End Module

## 特性类

特性类 是派生自 System.Attribute 的非泛型类，因此不是 MustInherit。特性类可以具有 System.AttributeUsage 特性，以便声明特性所适用的对象、是否可在声明中多次使用以及是否被继承。下面的示例定义一个名为 SimpleAttribute 的特性类，该特性类可放在类声明和接口声明上：

<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Interface)> \_  
Public Class SimpleAttribute  
 Inherits System.Attribute  
End Class

下面的示例演示 Simple 特性的几种用法。虽然该特性类名为 SimpleAttribute，但使用此特性时可忽略 Attribute 后缀，从而使其名称缩短为 Simple：

<Simple> Class Class1  
End Class  
  
<Simple> Interface Interface1  
End Interface

如果该特性缺少 System.AttributeUsage，则可将该特性放置在任何目标上（等效于 AttributeTargets.All）。System.AttributeUsage 特性具有一个变量初始值设定项 AllowMultiple，该初始值设定项用于指定是否可为给定的声明多次指定所标示的特性。如果特性的 AllowMultiple 为 True，则为多次性特性类，可在一个声明上指定多次。如果特性的 AllowMultiple 为 False 或未指定，则为一次性特性类，在一个声明上最多只能指定一次。

下面的示例定义一个名为 AuthorAttribute 的多次性特性类：

<AttributeUsage(AttributeTargets.Class, AllowMultiple:=True)> \_  
Public Class AuthorAttribute  
 Inherits System.Attribute  
  
 Private \_Value As String  
  
 Public Sub New(value As String)  
 Me.\_Value = value  
 End Sub  
  
 Public ReadOnly Property Value() As String  
 Get  
 Return \_Value  
 End Get  
 End Property  
End Class

该示例演示一个使用两次 Author 特性的类声明：

<Author("Maria Hammond"), Author("Ramesh Meyyappan")> \_  
Class Class1  
End Class

System.AttributeUsage 特性具有一个公共实例变量 Inherited，该变量指定基类型上指定的特性是否也由从该基类型派生的类型继承。如果未初始化 Inherited 公共实例变量，则使用默认值 False。属性和事件不继承特性，但由属性和事件定义的方法则继承特性。接口不继承特性。

如果一次性特性既是继承的，又指定在派生类型上，则在派生类型上指定的特性将重写继承的特性。如果多次性特性既是继承的，又指定在派生类型上，则将在派生类型上同时指定这两个特性。例如：

<AttributeUsage(AttributeTargets.Class, AllowMultiple:=True, \_  
 Inherited:=True) > \_  
Class MultiUseAttribute  
 Inherits System.Attribute  
  
 Public Sub New(value As Boolean)  
 End Sub  
End Class  
  
<AttributeUsage(AttributeTargets.Class, Inherited:=True)> \_  
Class SingleUseAttribute  
 Inherits Attribute  
  
 Public Sub New(value As Boolean)  
 End Sub  
End Class  
  
<SingleUse(True), MultiUse(True)> Class Base  
End Class  
  
' Derived has three attributes defined on it: SingleUse(False),  
' MultiUse(True) and MultiUse(False)  
<SingleUse(False), MultiUse(False)> \_  
Class Derived  
 Inherits Base  
End Class

特性的位置参数由特性类的公共构造函数的参数定义。位置形参必须为 ByVal，不能指定 ByRef。公共实例变量和属性由特性类的公共读写属性或实例变量定义。可用在位置参数及公共实例变量和属性中的类型限定为特性类型。如果类型满足以下条件之一，则为特性类型：

除 Date 和 Decimal 之外的任何基元类型。

Object 类型。

System.Type 类型。

枚举类型，但该类型及其中的嵌套类型（如果有）应具有 Public 可访问性。

此列表中上述某一类型的一维数组。

## 特性块

特性在特性块中指定。每个特性块由尖括号（“<>”）分隔；在一个特性块或多个特性块中，可用逗号分隔列表指定多个特性。指定特性的顺序无关紧要。例如，特性块 <A, B>、<B, A>、<A> <B> 和 <B> <A> 都是等效的。

不能在特性不支持的声明类型上指定特性，在特性块中不能多次指定一次性特性。下面的示例会导致错误，因为它试图在接口 Interface1 上使用 HelpString 并在 Class1 的声明上多次使用该特性。

<AttributeUsage(AttributeTargets.Class)> \_  
Public Class HelpStringAttribute  
 Inherits System.Attribute  
  
 Private InternalValue As String  
  
 Public Sub New(value As String)  
 Me.InternalValue = value  
 End Sub  
  
 Public ReadOnly Property Value() As String  
 Get  
 Return InternalValue  
 End Get  
 End Property  
End Class  
  
' Error: HelpString only applies to classes.  
<HelpString("Description of Interface1")> \_  
Interface Interface1  
 Sub Sub1()  
End Interface  
  
' Error: HelpString is single-use.  
<HelpString("Description of Class1"), \_  
 HelpString("Another description of Class1")> \_  
Public Class Class1  
End Class

特性由一个可选的修饰符、一个特性名称、一个可选的位置参数列表及变量/属性初始值设定项组成。如果没有参数或初始值设定项，则可以省略括号。如果特性具有修饰符，则必须放在源文件顶部的特性块中。

如果源文件顶部包含的特性块为将包含该源文件的程序集或模块指定特性，则该特性块中的每个特性都必须以 Assembly 或 Module 修饰符加冒号作为前缀。

Attributes ::=  
 AttributeBlock |  
 Attributes AttributeBlock

AttributeBlock ::= [ LineTerminator ] < AttributeList [ LineTerminator ] > [ LineTerminator ]

AttributeList ::=  
 Attribute |  
 AttributeList *Comma* Attribute

Attribute ::=  
 [ AttributeModifier : ] SimpleTypeName [ OpenParenthesis [ AttributeArguments ] CloseParenthesis ]

AttributeModifier ::= Assembly | Module

### 特性名称

特性的名称指定特性类。按照惯例，特性类名称带有后缀 Attribute。使用特性时可以包含或省略此后缀。因此，与特性标识符相对应的特性类的名称不是该标识符本身，就是连接在一起的限定标识符和 Attribute。当编译器对特性名称进行解析时，它会向该名称附加 Attribute 并尝试执行查找。如果该查找失败，编译器会尝试在不附加该后缀的情况下尝试执行查找。例如，使用特性类 SimpleAttribute 时可以省略 Attribute 后缀，从而将该名称缩短为 Simple：

<Simple> Class Class1  
End Class  
  
<Simple> Interface Interface1  
End Interface

上面的示例在语义上与下面的示例是等效的：

<SimpleAttribute> Class Class1  
End Class  
  
<SimpleAttribute> Interface Interface1  
End Interface

一般来说，最好使用带后缀 Attribute 的特性名称。下面的示例演示两个分别名为 T 和 TAttribute 的特性类。

<AttributeUsage(AttributeTargets.All)> \_  
Public Class T  
 Inherits System.Attribute  
End Class  
  
<AttributeUsage(AttributeTargets.All)> \_  
Public Class TAttribute  
 Inherits System.Attribute  
End Class  
  
' Refers to TAttribute.  
<T> Class Class1   
End Class  
  
' Refers to TAttribute.  
<TAttribute> Class Class2   
End Class

特性块 <T> 和特性块 <TAttribute> 都引用名为 TAttribute 的特性类。除非移除类 TAttribute 的声明，否则不能将 T 用作特性。

### 特性参数

特性的形参可采用两种形式：位置实参和实例变量/属性初始值设定项。特性的任何位置参数都必须位于实例变量/属性初始值设定项之前。位置参数由常量表达式、一维数组创建表达式或 GetType 表达式组成。实例变量/属性初始值设定项由标识符（相当于后跟冒号加等号的关键字）组成，以常量表达式或 GetType 表达式结尾。

假定有一个特性具有特性类 T、位置参数列表 P 和实例变量/属性初始值设定项列表 N，则以下这些步骤可确定这些参数是否有效：

按照编译时处理步骤编译 New T(P) 形式的表达式。这会生成编译时错误，或导致确定 T 上最适用于参数列表的构造函数。

如果步骤 1 中确定的构造函数包含非特性类型的参数，或在目标站点不可访问，则会发生编译时错误。

对于 N 中的每个实例变量/属性初始值设定项 Arg：

让 Name 成为实例变量/属性初始值设定项 Arg 的标识符。

Name 必须在类型为特性类型的 T 上标识非 Shared 的可写 Public 实例变量或无形参属性。如果 T 没有这样的实例变量或属性，则会发生编译时错误。

例如：

<AttributeUsage(AttributeTargets.All)> \_  
Public Class GeneralAttribute  
 Inherits Attribute  
  
 Public Sub New(x As Integer)  
 End Sub  
  
 Public Sub New(x As Double)  
 End Sub  
  
 Public y As Type  
  
 Public Property z As Integer  
 Get  
 End Get  
  
 Set  
 End Set  
 End Property  
End Class  
  
' Calls the first constructor.  
<General(10, z:=30, y:=GetType(Integer))> \_  
Class C1  
End Class  
  
' Calls the second constructor.  
<General(10.5, z:=10)> \_  
Class C2  
End Class

不能在特性实参中的任何位置使用类型形参。不过，可以使用构造类型：

<AttributeUsage(AttributeTargets.All)> \_  
Class A   
 Inherits System.Attribute   
   
 Public Sub New(t As Type)  
 End Sub   
End Class  
  
Class List(Of T)   
 ' Error: attribute argument cannot use type parameter  
 <A(GetType(T))> Dim t1 As T   
  
 ' OK: closed type  
 <A(GetType(List(Of Integer)))> Dim y As Integer  
End Class

AttributeArguments ::=  
 AttributePositionalArgumentList |  
 AttributePositionalArgumentList Comma VariablePropertyInitializerList |  
 VariablePropertyInitializerList

AttributePositionalArgumentList ::=  
 [ AttributeArgumentExpression ] |  
 AttributePositionalArgumentList Comma [ AttributeArgumentExpression ]

VariablePropertyInitializerList ::=  
 VariablePropertyInitializer |  
 VariablePropertyInitializerList Comma VariablePropertyInitializer

VariablePropertyInitializer ::=  
 IdentifierOrKeyword ColonEquals AttributeArgumentExpression

AttributeArgumentExpression ::=  
 ConstantExpression |  
 GetTypeExpression |  
 ArrayExpression

# 源文件和命名空间

一个 Visual Basic 程序由一个或更多源文件组成。在编译程序时，将一起处理所有源文件；这样，源文件就可能以循环方式相互依赖，没有任何前向声明要求。程序文本中声明的文本顺序通常不重要。

一个源文件由可选的 option 语句、import 语句和属性集组成，它们的后面跟有命名空间主体。每个特性都必须具有 Assembly 或 Module 修饰符，特性应用于通过编译生成的 .NET 程序集或模块。源文件的主体用作全局命名空间的隐式命名空间声明，表示源文件顶层的所有声明都位于全局命名空间中。例如：

FileA.vb:

Class A  
End Class

FileB.vb:

Class B  
End Class

两个源文件为全局命名空间提供成员，在此情况下将声明具有完全限定名 A 和 B 的两个类。由于两个源文件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。

注意 编译环境可能会重写命名空间声明，向声明中隐式放置源文件。

除非另有说明，否则可以通过行结束符或冒号终止 Visual Basic 程序中的语句。

Start ::=  
 [ OptionStatement+ ]  
 [ ImportsStatement+ ]  
 [ AttributesStatement+ ]  
 [ NamespaceMemberDeclaration+ ]

StatementTerminator ::= LineTerminator | :

AttributesStatement ::= Attributes StatementTerminator

## 程序启动和终止

当执行环境执行指定的方法（称为程序的“入口点”）时程序启动发生。这个入口点方法（必须始终命名为 Main）必须共享，不能包含在泛型类型中，不能具有异步修饰符，并且必须具有下列签名之一：

Sub Main()  
Sub Main(args() As String)  
Function Main() As Integer  
Function Main(args() As String) As Integer

入口点方法的可访问性并不重要。如果程序中包含一个以上适宜入口点，则编译环境必须将其中一个指定为入口点。其他情况下，将发生编译时错误。如果不存在入口点方法，编译环境可能还会创建一个入口点方法。

程序开始执行时，如果入口点具有形参，则由执行环境提供的实参包含命令行实参，在程序中表示为字符串。如果入口点的返回类型为 Integer，则从函数返回的值将作为程序的结果返回到执行环境。

在所有其他方面，入口点方法与其他方法的表现方式相同。当执行离开由执行环境进行的入口点方法调用时，程序将终止。

## 编译选项

源文件可以使用 option 语句在源代码中指定编译选项。Option 语句仅适用于它所在的源文件，并且在一个源文件中，每个类型的 Option 语句只能出现一次。例如：

Option Strict On  
Option Compare Text  
Option Strict Off ' Not allowed, Option Strict is already specified.  
Option Compare Text ' Not allowed, Option Compare is already specified.

共有 4 个编译选项：严格类型语义、显式声明语义、比较语义和局部变量类型推断语义。如果源文件不包含任何特定 Option 语句，则编译环境确定将使用哪个特定语义集。除上述四个编译选项之外，还有一个编译选项，即“整数溢出检查”，此选项只能通过编译环境指定。

OptionStatement ::=  
 OptionExplicitStatement |  
 OptionStrictStatement |  
 OptionCompareStatement |  
 OptionInferStatement

### Option Explicit 语句

Option Explicit 语句确定是否可以隐式声明局部变量。该语句后面可带关键字 On 或 Off；如果不指定这两个关键字，则使用默认值 On。如果文件中未指定该语句，则由编译环境确定使用哪一个选项。

注意   Explicit 和 Off 不是保留字。

Option Explicit Off  
  
Module Test  
 Sub Main()  
 x = 5 ' Valid because Option Explicit is off.  
 End Sub  
End Module

在此示例中，通过向局部变量 x 赋值对它进行了隐式声明。x 的类型为 Object。

OptionExplicitStatement ::= Option Explicit [ OnOff ] StatementTerminator

OnOff ::= On | Off

### Option Strict 语句

Option Strict 语句确定对 Object 的转换和操作是否由严格或许可类型语义控制，以及在没有指定 As 子句的情况下是否将类型隐式类型化为 Object。该语句后面可带关键字 On 或 Off；如果不指定这两个关键字，则使用默认值 On。如果文件中未指定该语句，则由编译环境确定使用哪一个选项。

注意   Strict 和 Off 不是保留字。

Option Strict On  
  
Module Test  
 Sub Main()  
 Dim x ' Error, no type specified.  
 Dim o As Object  
 Dim b As Byte = o ' Error, narrowing conversion.  
  
 o.F() ' Error, late binding disallowed.  
 o = o + 1 ' Error, addition is not defined on Object.  
 End Sub  
End Module

在严格语义下，不允许出现以下情况：

不带显式强制转换运算符的收缩转换。

后期绑定。

对 TypeOf...Is、Is 和 IsNot 之外的类型 Object 进行运算。

在不具有推断类型的声明中省略 As 子句。

OptionStrictStatement ::= Option Strict [ OnOff ] StatementTerminator

### Option Compare 语句

Option Compare 语句确定字符串比较的语义。字符串比较可使用二进制比较（对每个字符的二进制 Unicode 值进行比较）或文本比较（使用当前区域性对每个字符的词法含义进行比较）来进行。如果文件中未指定该语句，则由编译环境对使用的比较类型进行控制。

注意   Compare、Binary 和 Text 不是保留字。

Option Compare Text  
  
Module Test  
 Sub Main()  
 Console.WriteLine("a" = "A") ' Prints True.  
 End Sub  
End Module

在此示例中，字符串比较是使用忽略大小写差异的文本比较完成的。如果指定了 Option Compare Binary，则会输出 False。

OptionCompareStatement ::= Option Compare CompareOption StatementTerminator

CompareOption ::= Binary | Text

### 整数溢出检查

可以选择在运行时检查或不检查整数运算的溢出状况。如果检查溢出情况且发生整数运算溢出，则会引发 System.OverflowException 异常。如果不检查溢出状况，则整数运算溢出不会引发异常。编译环境将确定该选项是 on 还是 off。

### Option Infer 语句

Option Infer 语句用于确定没有 As 子句的局部变量声明是否具有推断类型或使用 Object。该语句后面可带关键字 On 或 Off；如果不指定这两个关键字，则使用默认值 On。如果文件中未指定该语句，则由编译环境确定使用哪一个选项。

注意   Infer 和 Off 不是保留字。

Option Infer On  
  
Module Test  
 Sub Main()  
 ' The type of x is Integer  
 Dim x = 10  
  
 ' The type of y is String  
 Dim y = "abc"  
 End Sub  
End Module

OptionInferStatement ::= Option Infer [ OnOff ] StatementTerminator

## Imports 语句

Imports 语句可将实体的名称导入源文件，这样，无需限定即可引用这些名称，该语句也可导入命名空间供 XML 表达式使用。

如果源文件包含 Imports 语句，则可以在成员声明内直接引用给定命名空间中的类型，如下面的示例所示：

Imports N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace   
  
Namespace N3  
 Class B  
 Inherits A  
 End Class  
End Namespace

这里，源文件中直接提供了命名空间 N1.N2 的类型成员，这样，类 N3.B 从类 N1.N2.A 派生而来。

Imports 语句必须在所有 Option 语句之后，但在所有类型声明之前。编译环境还可以定义隐式 Imports 语句。

通过 Imports 语句，可以在源文件中使用相应名称，而不用在全局命名空间的声明空间内声明任何内容。由 Imports 语句导入的名称作用域包括源文件所包含的命名空间成员声明。具体地说，Imports 语句的范围既不包括其他 Imports 语句，也不包括其他源文件。Imports 语句不能相互引用。

在此示例中，最后一个 Imports 语句有错误，因为它不能引用第一个导入别名。

Imports R1 = N1 ' OK.  
Imports R2 = N1.N2 ' OK.  
Imports R3 = R1.N2 ' Error: Can't refer to R1.  
  
Namespace N1.N2  
End Namespace

注意   出现在 Imports 语句中的命名空间或类型名称始终被视为是完全限定的。也就是说，命名空间或类型名称中最左侧的标识符始终在全局命名空间中进行解析，其余的解析按照常规名称解析规则进行。这是该语言中应用这种规则的唯一位置；该规则可确保名称不能从限定完全隐藏。在没有该规则的情况下，如果全局命名空间中的名称在某个特定源文件中隐藏，则将无法以限定方式指定该命名空间中的任何名称。

在此示例中，Imports 语句始终引用全局 System 命名空间，而不是源文件中的类。

Imports System ' Imports the namespace, not the class.  
  
Class System  
End Class

ImportsStatement ::= Imports ImportsClauses StatementTerminator

ImportsClauses ::=  
 ImportsClause |  
 ImportsClauses Comma ImportsClause

ImportsClause ::=  
 AliasImportsClause |  
 MembersImportsClause |  
 XMLNamespaceImportsClause

### 导入别名

导入别名定义了命名空间或类型的别名。

Imports A = N1.N2.A  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class B  
 Inherits A  
 End Class  
End Namespace

这里，源文件中的 A 是 N1.N2.A 的别名，这样，类 N3.B 从类 N1.N2.A 派生而来。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：

Imports R = N1.N2  
  
Namespace N3  
 Class B  
 Inherits R.A  
 End Class  
End Namespace

导入别名的标识符在全局命名空间的声明空间内必须是唯一的（不仅仅是定义了导入别名的源文件中的全局命名空间声明），即使它没有在全局命名空间的声明空间中声明名称。

批注

模块中的声明不会将名称引入到包含声明空间中。因此，模块中的声明可以具有与导入别名相同的名称，即使可以在包含声明空间中访问该声明的名称。

' Error: Alias A conflicts with typename A  
Imports A = N3.A  
  
Class A  
End Class  
  
Namespace N3  
 Class A  
 End Class  
End Namespace

这里，全局命名空间中已包含成员 A，因此，导入别名使用该标识符是一个错误。同样，相同源文件中的两个或更多导入别名以相同名称声明别名也是一个错误。

一个导入别名可以为任何命名空间或类型创建别名。对一个命名空间或类型进行访问时，无论用它的别名，还是用它的声明名称，结果是完全相同的。

Imports R1 = N1  
Imports R2 = N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class B  
 Private a As N1.N2.A  
 Private b As R1.N2.A  
 Private c As R2.A  
 End Class  
End Namespace

这里，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。

导入指定正在为其创建别名的命名空间或类型的确切名称。这必须是该命名空间或类型的精确的完全限定名称：不将一般规则用于限定名称解析（例如，允许通过派生类访问基类的成员）。

如果导入别名指向这些规则无法解析的类型或命名空间，则会忽略导入语句（并且编译器会发出警告）。

而且，不能引用开放式泛型类型 – 所有泛型类型必须提供有效的类型实参，并且所有类型实参都必须可由上述规则来解析。泛型类型的任何错误绑定均为错误。

Imports A = G ' 错误：因为 G 是开放式泛型类型  
Imports B = G(Of Integer) ' 正确  
Imports C = Derived.Nested ' 警告：Derived.Nested 本身不是类型  
Imports D = G(Of Derived.Nested) ' 错误：找不到 Derived.Nested  
  
Class G(Of T) : End Class  
  
Class Base  
 Class Nested : End Class  
End Class  
  
Class Derived : Inherits Base  
End Class  
  
Module Module1  
 Sub Main()  
 Dim x As C ' 错误：“C”定义错误  
 Dim y As Derived.Nested ' 正确  
 End Sub  
End Module

源文件中的声明可能会隐藏导入别名。

Imports R = N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class R  
 End Class  
  
 Class B  
 Inherits R.A ' Error, R has no member A  
 End Class  
End Namespace

在上面的示例中，在 B 的声明中对 R.A 进行引用会引起错误，因为 R 引用的是 N3.R 而不是 N1.N2。

导入别名可使别名在特定源文件中使用，但不会向基础声明空间提供任何新成员。换言之，导入别名不是可传递的，而只会影响它所在的源文件。

File1.vb:

Imports R = N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace

File2.vb:

Class B  
 Inherits R.A ' Error, R unknown.  
End Class

在上面的示例中，因为引入 R 的导入别名的范围仅限于包含该导入别名的源文件中的声明，所以，在第二个源文件中，R 是未知的。

AliasImportsClause ::=   
 Identifier Equals TypeName

### 命名空间导入

通过命名空间导入，可以导入命名空间或类型的所有成员，从而允许该命名空间或类型的每个成员的标识符没有限定地使用。对于类型而言，命名空间导入仅允许访问类型的共享成员而无需限定类名。尤其是，它允许没有限定地使用枚举类型的成员。例如：

Imports Colors  
  
Enum Colors  
 Red  
 Green  
 Blue  
End Enum  
  
Module M1  
 Sub Main()  
 Dim c As Colors = Red  
 End Sub  
End Module

与导入别名不同的是，命名空间导入对它所导入的名称没有限制，并且可以导入标识符已在全局命名空间中进行声明的命名空间和类型。源文件中的导入别名和声明将隐藏通过常规导入所导入的名称。

在下面的示例中，A 引用的是 N3 命名空间中成员声明内的 N3.A 而不是 N1.N2.A。

Imports N1.N2  
  
Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N3  
 Class A  
 End Class  
  
 Class B  
 Inherits A  
 End Class  
End Namespace

当一个以上导入的命名空间包含同名成员（并且该名称没有被导入别名或声明所隐藏）时，对该名称的引用将是不明确的，会引起编译时发生错误。

Imports N1  
Imports N2  
  
Namespace N1  
 Class A  
 End Class  
End Namespace   
  
Namespace N2  
 Class A  
 End Class  
End Namespace   
  
Namespace N3  
 Class B  
 Inherits A ' Error, A is ambiguous.  
 End Class  
End Namespace

在上面的示例中，N1 和 N2 都包含成员 A。由于 N3 将二者都导入，因此引用 N3 中的 A 会引起编译时发生错误。在这种情况下，通过限定对 A 的引用或引入导入别名来选择特定 A，可以解决这一冲突，如下例所示：

Imports N1  
Imports N2  
Imports A = N1.A  
  
Namespace N3  
 Class B  
 Inherits A ' A means N1.A.  
 End Class  
End Namespace

只能导入命名空间、类、结构、枚举类型和标准模块。

MembersImportsClause ::=  
 TypeName

### XML 命名空间导入

XML 命名空间导入为编译单元包含的非限定 XML 表达式定义命名空间或默认命名空间。例如：

Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 ' db namespace is "http://example.org/database"  
 Dim x = <db:customer><db:Name>Bob</></>  
  
 Console.WriteLine(x.<db:Name>)  
 End Sub  
End Module

只能为特定导入集定义 XML 命名空间（包括默认命名空间）一次。例如：

Imports <xmlns:db="http://example.org/database-one">  
' Error: namespace db is already defined  
Imports <xmlns:db="http://example.org/database-two">

XMLNamespaceImportsClause ::=  
 < XMLNamespaceAttributeName [ XMLWhitespace ] *Equals* [ XMLWhitespace ] XMLNamespaceValue >

XMLNamespaceValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter

## 命名空间

Visual Basic 程序是使用命名空间进行组织的。命名空间在内部对程序进行组织，并对程序元素向其他程序公开的方式进行组织。

与其他实体不同的是，命名空间是无终止的，并且可在同一程序内或多个程序之间多次进行声明，每个声明都向相同命名空间提供成员。在下面的示例中，两个命名空间声明向同一声明空间提供成员，声明了带完全限定名 N1.N2.A 和 N1.N2.B 的两个类。

Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N1.N2  
 Class B  
 End Class  
End Namespace

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同名成员的声明，就将出现错误。

存在一个没有任何名称且它的嵌套命名空间和类型始终可以无限定地进行访问的全局命名空间。在该全局命名空间中声明的命名空间成员的范围是整个程序文本。如果类型或命名空间是在完全限定名为 N 的命名空间中声明的，其作用域是命名空间完全限定名以 N 开始的每一个命名空间或 N 本身的程序文本。（请注意，编译器在默认情况下可以选择将声明置于特定命名空间中。这并不会改变仍存在一个全局、未命名的命名空间这一事实。）

在此示例中，类 B 可以看到类 A，因为在概念上，B 的命名空间 N1.N2.N3 嵌套在命名空间 N1.N2 中。

Namespace N1.N2  
 Class A  
 End Class  
End Namespace  
  
Namespace N1.N2.N3  
 Class B  
 Inherits A  
 End Class  
End Namespace

### 命名空间声明

命名空间声明有三种形式。

第一种形式以关键字 Namespace 开头，后跟相关的命名空间名称。如果相对命名空间名称是限定的，则可将命名空间声明视为在对应于限定名中的每个名称的命名空间声明中在词法上是嵌套的。例如，以下两个命名空间在语义上等效：

Namespace N1.N2  
 Class A  
 End Class  
  
 Class B  
 End Class  
End Namespace   
  
Namespace N1  
 Namespace N2  
 Class A  
 End Class  
  
 Class B  
 End Class  
 End Namespace  
End Namespace

第二种形式以关键字 Namespace Global 开头。这种形式被视为其所有成员声明都按词法放在全局未命名的命名空间中，而不考虑编译环境提供的任何默认值。这种形式的命名空间声明不能按词法嵌套在任何其他命名空间声明中。

第三种形式以关键字 Namespace Global 开头，后跟限定的标识符 N。这种形式视同第一种形式“Namespace N”的命名空间声明，将其按词法放在全局未命名的命名空间中，而不考虑编译环境提供的任何默认值。这种形式的命名空间声明不能按词法嵌套在任何其他命名空间声明中。

Namespace Global ' Puts N1.A in the global namespace  
 Namespace N1  
 Class A  
 End Class  
 End Namespace  
End Namespace  
  
Namespace Global.N1 ' Equivalent to the above  
 Class A  
 End Class  
End Namespace  
  
Namespace N1 ' May or may not be equivalent to the above,  
 Class A ' depending on defaults provided by the  
 End Class ' compilation environment  
End Namespace

在处理命名空间的成员时，声明特定成员的位置并不重要。如果两个程序在同一命名空间中定义了同名实体，则尝试在该命名空间中解析该名称会引起不明确性错误。

命名空间根据定义为 Public，因此命名空间声明不能包含任何访问修饰符。

NamespaceDeclaration ::=  
 Namespace NamespaceName StatementTerminator  
 [ NamespaceMemberDeclaration+ ]  
 End Namespace StatementTerminator

NamespaceName ::=   
 RelativeNamespaceName |  
 Global |  
 Global . RelativeNamespaceName

*RelativeNamespaceName ::=*  
 Identifier |  
 *Relative*NamespaceName Period IdentifierOrKeyword

### 命名空间成员

命名空间成员只能是命名空间声明和类型声明。类型声明可具有 Public 或 Friend 访问。类型的默认访问为 Friend 访问。

NamespaceMemberDeclaration ::=  
 NamespaceDeclaration |  
 TypeDeclaration

TypeDeclaration ::=  
 ModuleDeclaration |  
 NonModuleDeclaration

NonModuleDeclaration ::=  
 EnumDeclaration |  
 StructureDeclaration |  
 InterfaceDeclaration |  
 ClassDeclaration |  
 DelegateDeclaration

# 类型

Visual Basic 的类型包括值类型和引用类型两大类。基元类型（除字符串外）、枚举和结构都是值类型。类、字符串、标准模块、接口、数组和委托都是引用类型。

每个类型都有默认值，默认值是在初始化时赋给该类型变量的值。

TypeName ::=  
 ArrayTypeName |  
 NonArrayTypeName

NonArrayTypeName ::=  
 SimpleTypeName |  
 NullableTypeName

SimpleTypeName ::=  
 QualifiedTypeName |  
 BuiltInTypeName

QualifiedTypeName ::=  
 Identifier [ TypeArguments ] |  
 Global Period IdentifierOrKeyword [ TypeArguments ] |  
 QualifiedTypeName Period IdentifierOrKeyword [ TypeArguments ]

TypeArguments ::=  
 OpenParenthesis Of TypeArgumentList CloseParenthesis

TypeArgumentList ::=  
 TypeName |  
 TypeArgumentList Comma TypeName

BuiltInTypeName ::= Object | PrimitiveTypeName

TypeModifier ::= AccessModifier | Shadows

IdentifierModifiers ::= [ *NullableNameModifier* ] [ ArrayNameModifier ]

## 值类型和引用类型

尽管值类型和引用类型在声明语法和用法上类似，它们的语义却是完全不同的。

引用类型存储于运行时堆中；只能通过引用该存储来访问这些类型。因为引用类型总是通过引用访问的，所以其生命周期由 .NET Framework 管理。将跟踪对特定实例的未完成引用，只有不再保留任何引用时才会销毁该实例。引用类型的变量包含对如下值的引用：该类型的值、派生程度更重的类型的值或 null 值。null 值不引用任何项，null 值只能被赋值。对引用类型的变量进行赋值时会创建引用的副本，而不是引用值的副本。对于引用类型的变量，默认值为 null 值。

值类型直接存储在数组或其他类型的堆栈中；只能直接访问对这些类型的存储。因为值类型直接存储在变量中，所以这些类型的生命周期由其所在变量的生命周期所决定。当销毁包含值类型实例的位置时，也会销毁该值类型实例。总是直接访问值类型；不能创建对值类型的引用。禁止这样的引用就不会引用已经销毁的值类型实例。因为值类型始终是 NotInheritable，所以值类型变量始终包含值类型值。因此，值类型的值不能为 null 值，也不能引用一个派生程度更重的类型的对象。对值类型的变量赋值时，会创建所赋的值的一个副本。对于值类型的变量来说，默认值是将该类型的每个变量成员初始化为其默认值的结果。

下面的示例说明引用类型与值类型之间的差异：

Class Class1  
 Public Value As Integer = 0  
End Class  
  
Module Test  
 Sub Main()  
 Dim val1 As Integer = 0  
 Dim val2 As Integer = val1  
 val2 = 123  
 Dim ref1 As Class1 = New Class1()  
 Dim ref2 As Class1 = ref1  
 ref2.Value = 123  
 Console.WriteLine("Values: " & val1 & ", " & val2)  
 Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)  
 End Sub  
End Module

该程序的输出为：

Values: 0, 123  
Refs: 123, 123

对局部变量 val2 的赋值不影响局部变量 val1，因为这两个局部变量都是值类型（Integer 类型），每个值类型局部变量都有自己的存储位置。相反，赋值语句 ref2.Value = 123; 对 ref1 和 ref2 引用的对象都会产生影响。

请注意，在 .NET Framework 类型系统中，尽管结构、枚举和基元类型（除 String 外）是值类型，它们全部继承自引用类型。结构和基元类型继承自引用类型 System.ValueType，后者又继承自 Object。枚举类型继承自引用类型 System.Enum，后者又继承自 System.ValueType。

### 可以为 Null 的值类型

对于值类型来说，可以将 ? 修饰符添加到类型名称，以表示该类型的可以为 null 的版本。可以为 null 的值类型可以包含与该类型的不可为 null 的版本相同的值，也可以包含 null 值。因此，对于可以为 null 的值类型，将 Nothing 赋予该类型的变量会将该变量的值设置为 null 值，而不是该值类型的零值。例如：

Dim x As Integer = Nothing  
Dim y As Integer? = Nothing  
  
' Prints zero  
Console.WriteLine(x)  
' Prints nothing (because the value of y is the null value)  
Console.WriteLine(y)

通过将可以为 null 的类型修饰符应用到变量名，也可以将变量声明为可以为 null 的值类型的变量。为清楚起见，在同一个声明中，对变量名和类型名同时应用可以为 null 的类型修饰符是无效的。因为可以为 null 的类型是通过使用类型 System.Nullable(Of T) 实现的，所以类型 T? 等同于类型 System.Nullable(Of T)，这两个名称可以互换使用。可以为 null 的类型不能再使用 ? 修饰符；因此，不能声明类型 Integer?? 或 System.Nullable(Of Integer)?。

可以为 null 的值类型 T? 具有 System.Nullable(Of T) 的成员，以及从基础类型 T 提升 到类型 T? 的所有运算符或转换。从基础类型提升的副本运算符和转换，在大多数情况下会用可以为 null 的值类型替代不可以为 null 的值类型。这样，很多应用于 T 的转换和运算符也可以应用于 T?。

NullableTypeName ::= NonArrayTypeName ?

NullableNameModifier ::= ?

## 接口实现

结构和类声明可以声明，它们通过一个或多个 Implements 子句实现了一组接口类型。在 Implements 子句中指定的所有类型都必须是接口，并且这些类型必须实现接口的所有成员。例如：

Interface ICloneable  
 Function Clone() As Object  
End Interface   
  
Interface IComparable  
 Function CompareTo(other As Object) As Integer  
End Interface   
  
Structure ListEntry  
 Implements ICloneable, IComparable  
  
 ...  
  
 Public Function Clone() As Object Implements ICloneable.Clone  
 ...  
 End Function   
  
 Public Function CompareTo(other As Object) As Integer \_  
 Implements IComparable.CompareTo  
 ...  
 End Function   
End Structure

实现接口的类型还隐式实现接口的所有基接口。即使类型未在 Implements 子句中显式列出所有基接口，也是如此。在此示例中，TextBox 结构实现 IControl 和 ITextBox。

Interface IControl  
 Sub Paint()  
End Interface   
  
Interface ITextBox  
 Inherits IControl  
  
 Sub SetText(text As String)  
End Interface   
  
Structure TextBox  
 Implements ITextBox  
  
 ...  
  
 Public Sub Paint() Implements ITextBox.Paint  
 ...  
 End Sub  
  
 Public Sub SetText(text As String) Implements ITextBox.SetText  
 ...  
 End Sub  
End Structure

声明一个类型实现了该类型的接口，并不意味着在该类型的声明空间中声明了任何内容。因此，使用同名的方法实现两个接口是有效的。

虽然类型可以包含其作用域内的类型形参，但类型本身不能实现类型形参。

Class C1(Of V)  
 Implements V ' Error, can't implement type parameter directly  
 Implements IEnumerable(Of V) ' OK, not directly implementing  
  
 ...  
End Class

可以使用不同的类型实参多次实现泛型接口。然而，如果提供的类型形参（无论带有何种类型约束）可以与接口的其他实现重叠，则泛型类型无法使用类型形参实现泛型接口。例如：

Interface I1(Of T)  
End Interface  
  
Class C1  
 Implements I1(Of Integer)  
 Implements I1(Of Double) ' OK, no overlap  
End Class  
  
Class C2(Of T)  
 Implements I1(Of Integer)  
 Implements I1(Of T) ' Error, T could be Integer  
End Class

TypeImplementsClause ::= Implements *Type*Implements StatementTerminator

TypeImplements ::=  
 NonArrayTypeName |  
 *Type*Implements Comma NonArrayTypeName

## 基元类型

基元类型通过关键字进行标识，这些关键字是 System 命名空间中的预定义类型的别名。基元类型与类型别名没有任何区别：写入保留字 Byte 与写入 System.Byte 是完全相同的。基元类型也称为*内部类型*。

因为基元类型的别名为常规类型，所以每个基元类型都包含成员。例如，Integer 具有 System.Int32 中声明的成员。可将文本视为其对应类型的实例。

基元类型与其他结构类型的不同之处在于前者允许某些附加运算符：

基元类型允许通过写入文本来创建值。例如，123I 是 Integer 类型的文本。

可以声明基元类型的常量。

当表达式的操作数都是基元类型常量时，编译器可以在编译时计算表达式。这样的表达式称为常量表达式。

Visual Basic 定义以下基元类型：

整数值类型 Byte（1 字节无符号整数）、SByte（1 字节有符号整数）、UShort（2 字节无符号整数）、Short（2 字节有符号整数）、UInteger（4 字节无符号整数）、Integer（4 字节有符号整数）、ULong（8 字节无符号整数）和 Long（8 字节有符号整数）。这些类型分别映射到 System.Byte、System.SByte、System.UInt16、System.Int16、System.UInt32、System.Int32、System.UInt64 和 System.Int64。整数类型的默认值等于文本 0。

浮点值类型 Single（4 字节浮点）和 Double（8 字节浮点）。这些类型分别映射到 System.Single 和 System.Double。浮点类型的默认值等于文本 0。

Decimal 类型（16 字节小数值）映射到 System.Decimal。小数类型的默认值等于文本 0D。

Boolean 值类型，表示真值，通常是关系或逻辑运算的结果。其文本是 System.Boolean 类型。Boolean 类型的默认值相当于文本 False。

Date 值类型，表示日期和/或时间，映射到 System.DateTime。Date 类型的默认值相当于文本 # 01/01/0001 12:00:00AM #。

Char 值类型，表示单个 Unicode 字符，映射到 System.Char。Char 类型的默认值相当于常量表达式 ChrW(0)。

String 引用类型，表示 Unicode 字符序列，映射到 System.String。String 类型的默认值为 null 值。

PrimitiveTypeName ::= NumericTypeName | Boolean | Date | Char | String

NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal

IntegralTypeName ::= Byte | SByte | UShort | Short | UInteger | Integer | ULong | Long

FloatingPointTypeName ::= Single | Double

## 枚举

枚举是继承自 System.Enum 的值类型，枚举用符号表示基元整型的值集。对于类型 E 的枚举，默认值为表达式 CType(0, E) 生成的值。

枚举的基础类型必须是整数类型，并且可以表示枚举中定义的所有枚举值。如果指定基础类型，则基础类型必须是 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long，或这些类型在 System 命名空间中的对应类型之一。如果未显式指定基础类型，则默认基础类型为 Integer。

下面的示例声明基础类型为 Long 的枚举：

Enum Color As Long  
 Red  
 Green  
 Blue  
End Enum

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 Integer 范围内的值，或者保留此选项供将来使用。

EnumDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Enum Identifier [ As NonArrayTypeName ] StatementTerminator  
 EnumMemberDeclaration+  
 End Enum StatementTerminator

### 枚举成员

枚举成员是在枚举中声明的枚举值，这些成员继承自 System.Enum 类。

枚举成员的作用域是枚举声明体。这表示在枚举声明之外，始终应对枚举成员进行限定（除非该类型是通过命名空间导入明确导入到命名空间中的）。

当省略常量表达式值时，枚举成员声明的声明顺序非常重要。枚举成员只隐式具有 Public 访问类型；不能对枚举成员声明使用访问修饰符。

EnumMemberDeclaration ::= [ Attributes ] Identifier [ Equals ConstantExpression ] StatementTerminator

### 枚举值

枚举成员列表中的枚举值声明为常量，这些常量转换成基础枚举类型，并且每当需要常量时就会显示这些类型。使用 = 的枚举成员定义对关联的成员赋予常量表达式指示的值。常量表达式必须计算为可隐式转换为基础类型的整数类型，且该表达式必须在可由基础类型表示的值的范围之内。在下面的示例中，因为在严格语义下，常量值 1.5、2.3 和 3.3 不会隐式转换为基础整数类型 Long，所以会出现错误。

Option Strict On  
  
Enum Color As Long  
 Red = 1.5  
 Green = 2.3  
 Blue = 3.3  
End Enum

多个枚举成员可以共享同一关联值，如下所示：

Enum Color  
 Red  
 Green  
 Blue  
 Max = Blue  
End Enum

该示例中的枚举有两个枚举成员（Blue 和 Max）具有同一关联值。

如果枚举中的第一个枚举器值定义没有初始值设定项，则对应常量的值为 0。不包含初始值设定项的枚举值定义对枚举器赋的值为前一个枚举值加 1。这样增加后的值必须在该基础类型可表示的值的范围内。

Enum Color  
 Red  
 Green = 10  
 Blue  
End Enum   
  
Module Test  
 Sub Main()  
 Console.WriteLine(StringFromColor(Color.Red))  
 Console.WriteLine(StringFromColor(Color.Green))  
 Console.WriteLine(StringFromColor(Color.Blue))  
 End Sub  
  
 Function StringFromColor(c As Color) As String  
 Select Case c  
 Case Color.Red  
 Return String.Format("Red = " & CInt(c))  
  
 Case Color.Green  
 Return String.Format("Green = " & CInt(c))  
  
 Case Color.Blue  
 Return String.Format("Blue = " & CInt(c))  
  
 Case Else  
 Return "Invalid color"  
 End Select  
 End Function  
End Module

上述示例输出枚举值及其关联值。输出为：

Red = 0  
Green = 10  
Blue = 11

产生这些值的原因如下：

枚举值 Red 自动赋值 0（因为该枚举值没有初始值设定项，并且是第一个枚举值成员）。

枚举值 Green 显式赋值 10。

枚举值 Blue 自动赋予比前一个枚举值大 1 的值。

常量表达式不能直接或间接使用其自身的关联枚举值的值（即不允许常量表达式的循环）。因为 A 和 B 的声明是循环的，所以下面的示例无效。

Enum Circular  
 A = B  
 B  
End Enum

A 显式地依赖于 B，B 隐式地依赖于 A。

## 类

类是一种数据结构，它可以包含数据成员（常量、变量和事件）、函数成员（方法、属性、索引器、运算符和构造函数）和嵌套类型。类是引用类型。下面的示例显示包含每种成员的类：

Class AClass  
 Public Sub New()  
 Console.WriteLine("Constructor")  
 End Sub  
  
 Public Sub New(value As Integer)  
 MyVariable = value  
 Console.WriteLine("Constructor")  
 End Sub  
  
 Public Const MyConst As Integer = 12  
 Public MyVariable As Integer = 34  
  
 Public Sub MyMethod()  
 Console.WriteLine("MyClass.MyMethod")  
 End Sub  
  
 Public Property MyProperty() As Integer  
 Get  
 Return MyVariable  
 End Get  
  
 Set (value As Integer)  
 MyVariable = value  
 End Set  
 End Property  
  
 Default Public Property Item(index As Integer) As Integer  
 Get  
 Return 0  
 End Get  
  
 Set (value As Integer)  
 Console.WriteLine("Item(" & index & ") = " & value)  
 End Set  
 End Property  
  
 Public Event MyEvent()  
  
 Friend Class MyNestedClass  
 End Class   
End Class

下面的示例显示这些成员的用法：

Module Test  
  
 ' Event usage.  
 Dim WithEvents aInstance As AClass  
  
 Sub Main()  
 ' Constructor usage.  
 Dim a As AClass = New AClass()  
 Dim b As AClass = New AClass(123)  
  
 ' Constant usage.  
 Console.WriteLine("MyConst = " & AClass.MyConst)  
  
 ' Variable usage.  
 a.MyVariable += 1  
 Console.WriteLine("a.MyVariable = " & a.MyVariable)  
  
 ' Method usage.  
 a.MyMethod()  
  
 ' Property usage.  
 a.MyProperty += 1  
 Console.WriteLine("a.MyProperty = " & a.MyProperty)  
 a(1) = 1  
  
 ' Event usage.  
 aInstance = a  
 End Sub   
  
 Sub MyHandler() Handles aInstance.MyEvent  
 Console.WriteLine("Test.MyHandler")  
 End Sub   
End Module

有两个类特定的修饰符，即 MustInherit 和 NotInheritable。同时指定这两个修饰符是无效的。

ClassDeclaration ::=  
 [ Attributes ] [ ClassModifier+ ] Class Identifier [ TypeParameterList ] StatementTerminator  
 [ ClassBase ]  
 [ TypeImplementsClause+ ]  
 [ ClassMemberDeclaration+ ]  
 End Class StatementTerminator

ClassModifier ::= TypeModifier | MustInherit | NotInheritable | Partial

### 类基本规范

类声明可以包含基本类型规范，这些规范定义类的直接基本类型。如果类声明中未显式指定基类型，则隐式指定的直接基类型是 Object。例如：

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class

虽然基本类型可以包含其作用域内的类型形参，但基本类型本身不可以是类型形参。

Class C1(Of V)   
End Class  
  
Class C2(Of V)  
 Inherits V ' Error, type parameter used as base class   
End Class  
  
Class C3(Of V)  
 Inherits C1(Of V) ' OK: not directly inheriting from V.  
End Class

类只能从 Object 和类派生。从 System.ValueType、System.Enum、System.Array、System.MulticastDelegate 或 System.Delegate 派生类是无效的。不能从 System.Attribute 或其派生类派生泛型类。

每个类包含一个直接基类，不允许循环派生。不能从 NotInheritable 类派生，基类的可访问域必须与类本身的可访问域相同，或者是其超集。

ClassBase ::= Inherits NonArrayTypeName StatementTerminator

### 类成员

类成员包含由类成员声明引入的成员，以及从该类的直接基类继承的成员。

类成员声明可具有 Public、Protected、Friend、Protected Friend 或 Private 访问。当类成员声明不包含访问修饰符时，声明默认为 Public 访问类型（若是变量声明，则默认为 Private 访问类型）。

类成员的作用域是进行成员声明的类体以及该类的约束列表（如果它是泛型类并且具有约束）。如果成员具有 Friend 访问类型，其作用域将扩展到同一程序内所有派生类的类体，或具有 Friend 访问类型的任何程序集，如果成员具有 Public、Protected 或 Protected Friend 访问类型，其作用域将扩展到任何程序内任何派生类的类体。

ClassMemberDeclaration ::=  
 NonModuleDeclaration |  
 EventMemberDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

## 结构

结构 是继承自 System.ValueType 的值类型。结构与类很相似，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，结构不需要堆分配。

对于类，两个变量可以引用同一对象，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。在结构中，每个变量都有自己的非 Shared 数据副本，因此，对一个变量进行的操作不会影响另一个变量，如下例所示：

Structure Point  
 Public x, y As Integer  
  
 Public Sub New(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub  
End Structure

基于上述声明，下面的代码输出值 10：

Module Test  
 Sub Main()  
 Dim a As New Point(10, 10)  
 Dim b As Point = a  
  
 a.x = 100  
 Console.WriteLine(b.x)  
 End Sub  
End Module

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。

StructureDeclaration ::=  
 [ Attributes ] [ StructureModifier+ ] Structure Identifier [ TypeParameterList ]  
 StatementTerminator  
 [ TypeImplementsClause+ ]  
 [ StructMemberDeclaration+ ]  
 End Structure StatementTerminator

StructureModifier ::= TypeModifier | Partial

### 结构成员

结构的成员是结构成员声明引入的成员，或继承自 System.ValueType 的成员。每个结构都隐式具有 Public 无参数实例构造函数，用于生成结构默认值。因此，结构类型声明不能声明无参数实例构造函数。但允许结构类型声明参数化实例构造函数，如下面的示例所示：

Structure Point  
 Private x, y As Integer  
  
 Public Sub New(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub  
End Structure

如果进行上述声明，则下面两个语句都创建 x 和 y 初始化为零的 Point。

Dim p1 As Point = New Point()  
Dim p2 As Point = New Point(0, 0)

因为结构直接包含其字段值（而不是引用这些值），所以结构不能包含直接或间接引用自身的字段。例如，下面的代码是无效的：

Structure S1  
 Dim f1 As S2  
End Structure  
  
Structure S2  
 ' This would require S1 to contain itself.  
 Dim f1 As S1  
End Structure

通常，结构成员声明只能具有 Public、Friend 或 Private 访问，但重写继承自 Object 的成员时，也可以使用 Protected 和 Protected Friend 访问。当结构成员声明不包含访问修饰符时，该声明默认为 Public 访问类型。结构声明的成员的作用域是进行声明的结构体以及该结构的约束（如果它是泛型结构并且具有约束）。

StructMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

## 标准模块

标准模块是一种类型，其成员隐式为 Shared，其作用域是标准模块的包含命名空间的声明空间，而不仅是标准模块声明自身。永远不能实例化标准模块。声明标准模块类型的变量是错误的。

标准模块的成员具有两个完全限定名，其中一个不包含标准模块名，另一个包含标准模块名。一个命名空间中可能有多个标准模块定义一个具有特定名称的成员；在其中任何一个模块之外非限定地引用该名称会产生多义性。例如：

Namespace N1  
 Module M1  
 Sub S1()  
 End Sub  
  
 Sub S2()  
 End Sub  
 End Module  
  
 Module M2  
 Sub S2()  
 End Sub  
 End Module  
  
 Module M3  
 Sub Main()  
 S1() ' Valid: Calls N1.M1.S1.  
 N1.S1() ' Valid: Calls N1.M1.S1.  
 S2() ' Not valid: ambiguous.  
 N1.S2() ' Not valid: ambiguous.  
 N1.M2.S2() ' Valid: Calls N1.M2.S2.  
 End Sub  
 End Module  
End Namespace

模块只能在命名空间中声明，且不可以嵌套在其他类型中。标准模块不能实现接口，它们隐式从 Object 派生，只有 Shared 构造函数。

ModuleDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Module Identifier StatementTerminator  
 [ ModuleMemberDeclaration+ ]  
 End Module StatementTerminator

### 标准模块成员

标准模块的成员是其成员声明引入的成员，或从 Object 继承的成员。标准模块可以具有除实例构造函数外的任何类型的成员。所有标准模块类型成员都隐式为 Shared。

通常，标准模块成员声明只能具有 Public、Friend 或 Private 访问，但重写继承自 Object 的成员时，可以指定 Protected 和 Protected Friend 访问修饰符。当标准模块声明不包含访问修饰符时，声明默认为 Public 访问类型（若是变量声明，则默认为 Private 访问类型）。

如上所述，标准模块成员的作用域是包含标准模块声明的声明。继承自 Object 的成员不包含在此特定作用域中；这些成员没有作用域，必须始终用模块名进行限定。如果成员具有 Friend 访问类型，其作用域只扩展到在同一程序或具有 Friend 访问类型的程序集中声明的命名空间成员。

ModuleMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration

## 接口

接口是引用类型，由其他类型实现接口类型，以保证它们支持某些方法。接口从不会直接创建，也没有实际的表示形式 — 其他类型必须转换为接口类型。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。

下面的示例显示的接口包含默认属性 Item、事件 E、方法 F 和属性 P：

Interface IExample  
 Default Property Item(index As Integer) As String  
  
 Event E()  
  
 Sub F(value As Integer)  
  
 Property P() As String  
End Interface

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  
 Sub Paint()  
End Interface   
  
Interface ITextBox  
 Inherits IControl  
  
 Sub SetText(text As String)  
End Interface   
  
Interface IListBox  
 Inherits IControl  
  
 Sub SetItems(items() As String)  
End Interface   
  
Interface IComboBox  
 Inherits ITextBox, IListBox   
End Interface

类和结构可以实现多个接口。在下面的示例中，类 EditBox 派生自类 Control，并实现 IControl 和 IDataBound：

Interface IDataBound  
 Sub Bind(b As Binder)  
End Interface   
  
Public Class EditBox  
 Inherits Control  
 Implements IControl, IDataBound  
  
 Public Sub Paint() Implements IControl.Paint  
 ...  
 End Sub  
  
 Public Sub Bind(b As Binder) Implements IDataBound.Bind  
 ...  
 End Sub  
End Class

InterfaceDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Interface Identifier [ TypeParameterList ] StatementTerminator  
 [ InterfaceBase+ ]  
 [ InterfaceMemberDeclaration+ ]  
 End Interface StatementTerminator

### 接口继承

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。如果接口声明没有显式基接口，则类型不会有基接口，因为接口不会继承 Object（尽管接口确实可以扩大转换为 Object）。在下面的示例中，IComboBox 的基接口为 IControl、ITextBox 和 IListBox。

Interface IControl  
 Sub Paint()  
End Interface   
  
Interface ITextBox  
 Inherits IControl  
  
 Sub SetText(text As String)  
End Interface   
  
Interface IListBox  
 Inherits IControl  
  
 Sub SetItems(items() As String)  
End Interface   
  
Interface IComboBox  
 Inherits ITextBox, IListBox   
End Interface

接口继承其基接口的所有成员。换言之，上面的 IComboBox 接口继承成员 SetText 和 SetItems 以及 Paint。

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。

如果某个接口在基接口的可传递闭包中出现多次，则该接口只有一次将其成员提供给派生接口。对于多次定义的基接口，实现派生接口的类型只需要实现一次该基接口的方法。在下面的示例中，只需实现一次 Paint，即使该类实现 IComboBox 和 IControl 也是如此。

Class ComboBox  
 Implements IControl, IComboBox  
  
 Sub SetText(text As String) Implements IComboBox.SetText  
 End Sub  
  
 Sub SetItems(items() As String) Implements IComboBox.SetItems  
 End Sub  
  
 Sub Print() Implements IComboBox.Paint  
 End Sub  
End Class

Inherits 子句不影响其他 Inherits 子句。在下面的示例中，IDerived 必须使用 IBase 限定 INested 的名称。

Interface IBase  
 Interface INested  
 Sub Nested()  
 End Interface  
  
 Sub Base()  
End Interface  
  
Interface IDerived  
 Inherits IBase, INested ' Error: Must specify IBase.INested.  
End Interface

基接口的可访问域必须与接口本身的可访问域相同，或者前者是后者的超集。

InterfaceBase ::= Inherits InterfaceBases StatementTerminator

InterfaceBases ::=  
 NonArrayTypeName |  
 InterfaceBases Comma NonArrayTypeName

### 接口成员

接口的成员包含由该接口的成员声明引入的成员，以及从该接口的基接口继承的成员。尽管接口不从 Object 继承成员，因为每个实现接口的类或结构都不继承 Object，Object 的成员（包括扩展方法）仍被视为接口的成员，接口不需要强制转换为 Object 即可直接调用这些成员。例如：

Interface I1  
End Interface  
  
Class C1  
 Implements I1  
End Class  
  
Module Test  
 Sub Main()  
 Dim i As I1 = New C1()  
 Dim h As Integer = i.GetHashCode()  
 End Sub  
End Module

Members of an interface with the same name as members of Object implicitly shadow Object members. Only nested types, methods, properties, and events may be members of an interface. Methods and properties may not have a body. Interface members are implicitly Public and may not specify an access modifier. The scope of a member declared in an interface is the interface body in which the declaration occurs, plus the constraint list of that interface (if it is generic and has constraints).InterfaceMemberDeclaration ::=  
 NonModuleDeclaration |  
 InterfaceEventMemberDeclaration |  
 InterfaceMethodMemberDeclaration |  
 InterfacePropertyMemberDeclaration

## 数组

数组是一种引用类型，它包含通过索引来访问的变量，这些索引和数组中变量的顺序是一一对应的。数组中包含的变量（也称为数组元素）必须均为同一类型，将该类型称为数组的元素类型。创建数组实例时会产生数组元素，销毁数组实例时数组元素就会消失。数组中的每个元素都将初始化为其类型的默认值。类型 System.Array 是所有数组类型的基类型，不能进行实例化。每个数组类型都继承 System.Array 类型所声明的成员，并且可转换为该基类型（以及 Object）。包含元素 T 的一维数组类型还实现接口 System.Collections.Generic.IList(Of T) 和 IReadOnlyList(Of T)；如果 T 是引用类型，则该数组类型也为任何 U（它具有来自 T 的扩大引用转换）实现 IList(Of U) 和 IReadOnlyList(Of U)。

数组有一个“秩”，它确定和每个数组元素关联的索引个数。数组的秩决定了数组的维数。例如，秩为一的数组称为单维数组，秩大于一的数组称为多维数组。

下面的示例创建整数值的单维数组，初始化数组元素，然后输出每个元素：

Module Test  
 Sub Main()  
 Dim arr(5) As Integer  
 Dim i As Integer  
  
 For i = 0 To arr.Length - 1  
 arr(i) = i \* i  
 Next i  
  
 For i = 0 To arr.Length - 1  
 Console.WriteLine("arr(" & i & ") = " & arr(i))  
 Next i  
 End Sub  
End Module

该程序输出如下：

arr(0) = 0  
arr(1) = 1  
arr(2) = 4  
arr(3) = 9  
arr(4) = 16  
arr(5) = 25

数组的每个维都有一个关联的长度。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。维度的长度确定了该维度的索引的有效范围：如果维度长度为 N，其索引范围为零至 N – 1。如果某个维度的长度为零，则该维度没有有效的索引。数组中的元素总数是数组中各维度长度的乘积。如果某数组的任何维度为零，则称该数组为空。数组的元素类型可以是任何类型。

通过将修饰符添加到现有类型名称来指定数组类型。修饰符包含左括号、零个或更多逗号集和右括号。修饰后的类型即数组的元素类型，维数是逗号的个数加一。如果指定了多个修饰符，则该数组的元素类型是个数组。从左至右读取修饰符，最左边的修饰符是最外层的数组。在下面的示例中

Module Test  
 Dim arr As Integer(,)(,,)()  
End Module

arr 的元素类型是 Integer 的一维数组的三维数组的二维数组。

通过将数组类型修饰符或数组大小初始化修饰符应用到变量名称，也可以将变量声明为数组类型。在这种情况下，数组元素类型是在声明中提供的类型，并且数组维度由变量名修饰符确定。为清楚起见，在同一个声明中，对变量名和类型名同时应用一个数组类型修饰符是无效的。

下面的示例演示各种局部变量声明，这些声明使用的数组类型以 Integer 为元素类型：

Module Test  
 Sub Main()  
 Dim a1() As Integer ' Declares 1-dimensional array of integers.  
 Dim a2(,) As Integer ' Declares 2-dimensional array of integers.  
 Dim a3(,,) As Integer ' Declares 3-dimensional array of integers.  
  
 Dim a4 As Integer() ' Declares 1-dimensional array of integers.  
 Dim a5 As Integer(,) ' Declares 2-dimensional array of integers.  
 Dim a6 As Integer(,,) ' Declares 3-dimensional array of integers.  
  
 ' Declare 1-dimensional array of 2-dimensional arrays of integers   
 Dim a7()(,) As Integer  
 ' Declare 2-dimensional array of 1-dimensional arrays of integers.  
 Dim a8(,)() As Integer  
  
 Dim a9() As Integer() ' Not allowed.  
 End Sub  
End Module

数组类型名修饰符扩展到后面跟随的所有括号组之后。这表示当类型名后面允许跟有扩在括号中的实参集时，不能为数组类型名指定实参。例如：

Module Test  
 Sub Main()  
 ' This calls the Integer constructor.  
 Dim x As New Integer(3)  
  
 ' This declares a variable of Integer().  
 Dim y As Integer()  
  
 ' This gives an error.  
 ' Array sizes can not be specified in a type name.  
 Dim z As Integer()(3)  
 End Sub  
End Module

在最后一种情况下，(3) 解释为类型名的一部分，而不是解释为构造函数实参集。

ArrayTypeName ::= NonArrayTypeName ArrayTypeModifiers

ArrayTypeModifiers ::= ArrayTypeModifier+

ArrayTypeModifier ::= OpenParenthesis [ RankList ] CloseParenthesis

RankList ::=  
 Comma |  
 RankList Comma

ArrayNameModifier ::=  
 ArrayTypeModifiers |  
 ArraySizeInitializationModifier

## 委托

委托是一种引用类型，它引用类型的 Shared 方法或对象的实例方法。在其他语言中，与委托最接近的是函数指针，但函数指针只能引用 Shared 函数，而委托既可以引用 Shared，也可以引用实例方法。在引用实例方法时，委托不仅存储对方法入口点的引用，还存储对调用方法时所用对象实例的引用。

委托声明不能包含 Handles 子句、Implements 子句、方法体或 End 构造。委托声明的参数列表不能包含 Optional 或 ParamArray 参数。返回类型和参数类型的可访问域必须与委托本身的可访问域相同，或者前者是后者的超集。

委托的成员是从类 System.Delegate 继承的成员。委托也定义了以下方法：

带两个形参的构造函数，这两个形参分别是 Object 和 System.IntPtr 类型。

与委托具有相同签名的 Invoke 方法。

其签名为委托签名的 BeginInvoke 方法，其中有三点不同。第一，返回类型更改为 System.IAsyncResult。第二，两个附加参数添加到参数列表的末尾：第一个的类型为 System.AsyncCallback，第二个的类型为 Object。最后，所有 ByRef 形参都更改为 ByVal。

返回类型与委托相同的 EndInvoke 方法。方法的形参只是作为 ByRef 形参的委托参数，其顺序与在委托签名中的顺序相同。除这些形参之外，形参列表的末尾还有附加的 System.IAsyncResult 类型的形参。

定义和使用委托有三个步骤：声明、实例化和调用。

委托是使用委托声明语法声明的。下面的示例声明一个名为 SimpleDelegate 的委托，该委托不接受任何实参：

Delegate Sub SimpleDelegate()

下面的示例创建一个 SimpleDelegate 实例，然后立即调用它：

Module Test  
 Sub F()  
 System.Console.WriteLine("Test.F")  
 End Sub   
  
 Sub Main()  
 Dim d As SimpleDelegate = AddressOf F  
 d()  
 End Sub   
End Module

实例化方法的委托后立即通过委托调用方法没有什么意义，因为直接调用方法会更简单一些。委托会在使用其匿名之后发挥作用。下一个示例显示一个重复调用 SimpleDelegate 实例的 MultiCall 方法：

Sub MultiCall(d As SimpleDelegate, count As Integer)  
 Dim i As Integer  
  
 For i = 0 To count - 1  
 d()  
 Next i  
End Sub

对于 MultiCall 方法来说，SimpleDelegate 的目标方法是什么、此方法的可访问性如何，或该方法是 Shared 还是非共享的，这些都不重要。重要的是目标方法的签名与 SimpleDelegate 兼容。

DelegateDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Delegate MethodSignature StatementTerminator

MethodSignature ::= SubSignature | FunctionSignature

## 分部类型

类和结构声明可以是分部声明。分部声明可能能够（也可能不能）完全描述声明内的声明类型。相反，类型的声明可以分布到程序内的多个分部声明中；分部类型不能跨程序边界进行声明。分部类型声明针对声明指定 Partial 修饰符。然后，程序内某个类型的具有相同完全限定名称的任何其他声明，将在编译时与分部声明进行合并，形成单个类型声明。例如，下面的代码声明单个类 Test，其成员为 Test.C1 和 Test.C2。

a.vb:

Public Partial Class Test  
 Public Sub S1()  
 End Sub  
End Class

b.vb:

Public Class Test  
 Public Sub S2()  
 End Sub  
End Class

合并分部类型声明时，至少有一个声明必须有 Partial 修饰符，否则会产生编译时错误。

批注

尽管在多个分部声明中可以只对其中一个声明指定 Partial，最好还是对所有分部声明都指定该修饰符。如果只有一个分部声明可见，另有一个或多个分部声明隐藏（如扩展工具生成的代码时），也可以不对可见声明指定 Partial 修饰符，而对隐藏声明指定该修饰符。

只有类和结构可以使用分部声明进行声明。合并分部声明时要考虑类型的元数：如果两个类名称相同，但具有不同数目的类型形参，则不能同时对这两个类进行分部声明。分部声明可以指定特性、类修饰符、Inherits 语句或 Implements 语句。在编译时，所有分部声明将结合在一起，用作类型声明的一部分。如果特性、修饰符、基、接口或类型成员之间存在冲突，则会产生编译时错误。例如：

Public Partial Class Test1  
 Implements IDisposable  
End Class  
  
Class Test1  
 Inherits Object  
 Implements IComparable  
End Class  
  
Public Partial Class Test2  
End Class  
  
Private Partial Class Test2  
End Class

上一个示例声明一个类型 Test1，该类型为 Public，继承自 Object，并实现 System.IDisposable 和 System.IComparable。Test2 的分部声明会产生编译时错误，因为其中一个声明将 Test2 声明为 Public，另一个声明将 Test2 声明为 Private。

包含类型形参的分部类型可以为类型形参声明约束和差异，但是每个分部声明的约束和差异必须匹配。因此，约束和差异是特定的，因为它们不会像其他修饰符一样自动合并：

Partial Public Class List(Of T As IEnumerable)  
End Class  
  
' Error: Constraints on T don't match  
Class List(Of T As IComparable)  
End Class

使用多个分部声明声明类型不会影响类型内的名称查找规则。因此，分部类型声明可以使用在其他分部类型声明中声明的成员，或可以实现在其他分部类型声明中声明的接口上的方法。例如：

Public Partial Class Test1  
 Implements IDisposable  
  
 Private IsDisposed As Boolean = False  
End Class  
  
Class Test1  
 Private Sub Dispose() Implements IDisposable.Dispose  
 If Not IsDisposed Then  
 ...  
 End If  
 End Sub  
End Class

嵌套的类型也可以具有分部声明。例如：

Public Partial Class Test  
 Public Partial Class NestedTest  
 Public Sub S1()  
 End Sub  
 End Class  
End Class  
  
Public Partial Class Test  
 Public Partial Class NestedTest  
 Public Sub S2()  
 End Sub  
 End Class  
End Class

分部声明内的初始值设定项仍以声明的顺序执行；但是，在不同的分部声明中产生的初始值设定项没有确定的执行顺序。

## 构造类型

泛型类型声明本身并不表示具体的类型。只是通过应用类型实参形成很多不同类型时，可以将泛型类型声明用作“蓝图”。应用了类型实参的泛型类型称为构造类型。构造类型中的类型实参必须始终满足与之匹配的类型形参上的约束。

即使未直接指定类型形参，类型名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时，就会出现这种情况：

Class Outer(Of T)   
 Public Class Inner   
 End Class  
  
 ' Type of i is the constructed type Outer(Of T).Inner  
 Public i As Inner   
End Class

当泛型类型和所有类型实参可访问时，构造类型 C(Of T1,…,Tn) 是可访问的。例如，如果泛型类型 C 为 Public，并且所有类型实参 T1,…,Tn 都为 Public，则构造类型为 Public。但如果类型名或其中一个类型实参为 Private，则构造类型的可访问性为 Private。如果构造类型的一个类型实参是 Protected，另一个类型实参是 Friend，则构造类型只能在所在程序集或访问类型为 Friend 的任何程序集中的类或其子类中访问。换言之，构造类型的可访问域是其构成部分的可访问域的交集。

批注

构造类型的可访问域是其构成部分的可访问域的交集，这一点在定义新的可访问级别时会产生有趣的副作用。如果构造类型包含 Protected 元素和 Friend 元素，则只有在同时 可以访问 Friend 和 Protected 成员的上下文中，才能访问此构造类型。不过，在本语言中，无法表示此可访问性级别，因为 Protected Friend 可访问性表示实体可在能访问 Friend 或 Protected 成员的上下文中进行访问。

通过用提供的类型实参替代泛型类型中每次出现的类型形参，可以确定基、实现的接口和构造类型的成员。

### 开放和封闭类型

如果一个构造类型的一个或多个类型实参是该类型所在的类型或方法的类型形参，则此构造类型称为开放类型。这是因为该类型的有些类型形参仍然是未知的，所以该类型的实际形状尚不是完全已知的。相反，其类型实参都不是类型形参的泛型类型称为封闭类型。封闭类型的形状总是完全已知的。例如：

Class Base(Of T, V)  
End Class  
  
Class Derived(Of V)  
 Inherits Base(Of Integer, V)  
End Class  
  
Class MoreDerived  
 Inherits Derived(Of Double)  
End Class

构造类型 Base(Of Integer, V) 是开放类型，因为尽管已提供类型形参 T，但为类型形参 U 提供了另一个类型形参。因此，该类型的完全形状还是未知的。不过，构造类型 Derived(Of Double) 是封闭类型，因为继承层次结构中的所有类型形参均已提供。

开放类型的定义如下：

类型形参是开放类型。

当数组元素类型是开放类型时，该数组类型也是开放类型。

当构造类型的一个或多个类型实参为开放类型时，该构造类型是开放类型。

封闭类型是不属于开放类型的类型。

因为程序入口点不能是泛型类型，所以运行时使用的所有类型都是封闭类型。

## 特殊类型

.NET Framework 包含大量由 .NET Framework 和 Visual Basic 语言特殊对待的类：

System.Void 类型在 .NET Framework 中表示 void 类型，只能在 GetType 表达式中直接引用。

类型 System.RuntimeArgumentHandle、System.ArgIterator 和 System.TypedReference 都可以包含指向堆栈的指针，因此，不会出现在 .NET Framework 堆中。因此，不能将这些类型用作数组元素类型、返回类型、字段类型、泛型类型实参、可以为 null 的类型、ByRef 形参类型、要转换为 Object 或 System.ValueType 的值的类型、对 Object 或 System.ValueType 的实例成员进行调用的目标，也不能提升到闭包。

# 转换

转换是将一个值从一个类型更改为另一个类型的过程。例如，Integer 类型的值可以转换为 Double 类型的值，Derived 类型的值可以转换为 Base 类型的值（假定 Base 和 Derived 都是类，并且 Derived 继承自 Base）。转换可能不要求值本身发生改变（如下第二个示例所示），或者它们可能要求值的表示形式发生显著变化（如下第一个示例所示）。

转换可以是扩大转换或收缩转换。扩大转换就是从一个类型转换到另一个类型，后者的值域即使不大于原始类型的值域，也至少与它一样大。扩大转换绝不会失败。收缩转换是从一个类型转换为另一个类型，而目标类型的值范围小于原类型的值范围或者完全无关，因此进行转换时（例如，从 Integer 转换为 String）必须额外小心。可能伴有信息丢失的收缩转换可能会失败。

所有类型都定义有标识转换（即从某个类型转换到自身类型）和默认值转换（即从 Nothing 进行转换）。

## 隐式和显式转换

转换可以是隐式转换或显式转换。进行隐式转换无需任何特殊语法。在下面的示例中，Integer 值隐式转换为 Long 值：

Module Test  
 Sub Main()  
 Dim intValue As Integer = 123  
 Dim longValue As Long = intValue  
  
 Console.WriteLine(intValue & " = " & longValue)  
 End Sub  
End Module

另一方面，显式转换需要强制转换运算符。尝试在不使用强制转换运算符的情况下执行显式转换会引起编译时错误。下面的示例使用显式转换将 Long 值转换为 Integer 值。

Module Test  
 Sub Main()  
 Dim longValue As Long = 134  
 Dim intValue As Integer = CInt(longValue)  
  
 Console.WriteLine(longValue & " = " & intValue)  
 End Sub  
End Module

隐式转换集取决于编译环境和 Option Strict 语句。如果使用的是严格语义，则只有收缩转换才会隐式发生。如果使用的是许可类型语义，则所有扩大和收缩转换（换言之即所有转换）都可隐式发生。

## 布尔转换

Boolean 不是数值类型，但它就像枚举类型一样，可与数值类型相互进行收缩转换。文本 True 可转换为文本 255（Byte 类型）、65535（UShort 类型）、4294967295（UInteger 类型）、18446744073709551615（ULong 类型）和表达式 -1（SByte、Short、Integer、Long、Decimal、Single 和 Double 类型）。文本 False 可转换为文本 0。数值零转换为文本 False。所有其他数值转换为文本 True。

存在从布尔值到字符串的收缩转换，将转换为 System.Boolean.TrueString 或 System.Boolean.FalseString。也存在从字符串到布尔值的收缩转换：如果该字符串等于 TrueString 或 FalseString（在当前区域性中，不区分大小写），则该转换将使用适当的值；否则，该转换尝试将字符串分析为数值类型（如果可能，分析为十六进制或八进制；否则分析为浮点），并使用上面的规则；如果上述转换都不成功，则将引发 System.InvalidCastException。

## 数值转换

类型 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Decimal、Single 和 Double 与所有枚举类型之间都可进行数值转换。进行转换时，可将枚举类型视为是它们的基础类型。当转换到一个枚举类型时，无需源值即可符合枚举类型中所定义的值集。例如：

Enum Values  
 One  
 Two  
 Three  
End Enum  
  
Module Test  
 Sub Main()  
 Dim x As Integer = 5  
  
 ' OK, even though there is no enumerated value for 5.  
 Dim y As Values = CType(x, Values)  
 End Sub  
End Module

数值转换在运行时进行处理，如下所示：

对于从一个数值类型到一个更宽数值类型的转换，只是简单地将数值转换到更宽的类型。从 UInteger、Integer、ULong、Long 或 Decimal 到 Single 或 Double 的转换将舍入为最接近的 Single 或 Double 值。虽然此转换可能导致精度损失，但决不会影响数值大小。

如果从一个整型转换为另一个整型，或从 Single、Double 或 Decimal 转换为整型，结果取决于是否启用整数溢出检查：

如果检查整数溢出：

当源是一个整数类型时，如果源参数在目标类型的范围之内，转换将会成功。如果源实参在目标类型的范围之外，则转换会引发 System.OverflowException 异常。

如果源是 Single、Double 或 Decimal 类型，源值将舍入为最接近的整数值，该整数值即是转换结果。如果源值同样地接近于两个整数值，则该源值将舍入到最低有效位位置是一个偶数的值。如果得到的整数值在目标类型的范围之外，则会引发 System.OverflowException 异常。

如果没有检查整数溢出：

如果源是整数类型，则转换始终成功，转换时将简单地丢弃源值的最高有效位。

如果源是 Single、Double 或 Decimal 类型，则转换一定会成功，转换时只是会将源值舍入为最接近的整数值。如果源值同样地接近于两个整数值，则该源值将始终舍入到最低有效位位置是一个偶数的值。

对于从 Double 到 Single 的转换，Double 值将舍入为最近的 Single 值。如果 Double 值过小，无法表示为 Single 值，则结果为正零或负零。如果 Double 值过大，无法表示为 Single 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 Single 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。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。

对于从 Double 到 Single 的转换，Double 值将舍入为最近的 Single 值。如果 Double 值过小，无法表示为 Single 值，则结果为正零或负零。如果 Double 值过大，无法表示为 Single 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。

## 引用转换

可以将引用类型转换成基类型，反之亦然。如果所转换的值是 null 值、派生类型本身或派生程度更大的类型，则从基类型到派生程度更大的类型的转换仅在运行时才会成功。

可以在类和接口类型与任何接口类型之间相互进行强制转换。如果涉及的实际类型具有继承或实现关系，则在一个类型与一个接口类型之间的转换仅在运行时才会成功。因为接口类型始终包含从 Object 派生而来的类型的实例，所以接口类型与 Object 之间也始终可以相互进行强制转换。

批注

在 NotInheritable 类与它不实现的接口之间进行相互转换不会引起错误，因为表示 COM 类的类可能具有到运行时才明确的接口实现。

如果引用转换在运行时失败，则会引发 System.InvalidCastException 异常。

### 引用差异转换

泛型接口或委托可能具有变量类型参数，这些参数允许在类型的兼容变量之间进行转换。因此在运行时，从一个类类型或接口类型到在变量上与所继承或实现的接口类型兼容的一个接口类型的转换将会成功。同样，可以在委托类型与变量兼容委托类型之间相互进行强制转换。例如，委托类型

Delegate Function F(Of In A, Out R)(a As A) As R

将允许从 F(Of Object, Integer) 到 F(Of String, Integer) 的转换。也就是说，接受 Object 的委托 F 可以安全地用作接受 String 的委托 F。在调用该委托时，目标方法将需要一个对象，而一个字符串就是一个对象。

在以下情况下，泛型委托或接口类型 S(Of S1,…,SN) 称为与泛型接口或委托类型 T(Of T1,…,TN) 变体兼容：

S 和 T 由同一泛型类型 U(Of U1,…,UN) 构造而成。

对于每个类型形参 UX：

如果声明的类型形参没有变化，则 SX 和 TX 必须为同一类型。

如果类型形参声明为 In，则从 SSX 到 TX 必须存在扩大的标识、默认值、引用、数组或类型形参转换。

如果类型形参声明为 Out，则从 STX 到 SX 必须存在扩大的标识、默认值、引用、数组或类型形参转换。

从一个类转换到一个带有变量类型参数的泛型接口时，如果该类实现一个以上变量兼容接口，则在不存在非变量转换时，转换将是不明确的。例如：

Class Base  
End Class  
  
Class Derived1  
 Inherits Base  
End Class  
  
Class Derived2  
 Inherits Base  
End Class  
  
Class OneAndTwo  
 Implements IEnumerable(Of Derived1)  
 Implements IEnumerable(Of Derived2)  
End Class  
  
Class BaseAndOneAndTwo  
 Implements IEnumerable(Of Base)  
 Implements IEnumerable(Of Derived1)  
 Implements IEnumerable(Of Derived2)  
End Class  
  
Module Test  
 Sub Main()  
 ' Error: conversion is ambiguous  
 Dim x As IEnumerable(Of Base) = New OneAndTwo()  
  
 ' OK, will pick up the direct implementation of IEnumerable(Of Base)  
 Dim y as IEnumerable(Of Base) = New BaseAndOneAndTwo()  
 End Sub  
End Module

### 匿名委托转换

在没有目标类型的上下文中将已归类为 lambda 方法的表达式重新归类为值时（例如，Dim x = Function(a As Integer, b As Integer) a + b），或其中的目标类型不是委托类型时，所得到的表达式的类型是一个匿名委托类型，等效于 lambda 方法的签名。此匿名委托类型具有到任何兼容委托类型的转换：以匿名委托类型的 Invoke 方法作为形参、可以使用委托创建表达式创建的任何委托类型，都是兼容委托类型。例如：

' Anonymous delegate type similar to Func(Of Object, Object, Object)  
Dim x = Function(x, y) x + y  
  
' OK because delegate type is compatible  
Dim y As Func(Of Integer, Integer, Integer) = x

请注意：System.Delegate 和 System.MulticastDelegate 类型本身不是委托类型（即使所有委托类型都从其中继承）。还请注意，从匿名委托类型到兼容委托类型的转换不是引用转换。

## 数组转换

数组是引用类型，除了为引用类型定义的转换之外，数组还可以进行几种特殊转换。

对于任何两个类型 A 和 B，如果它们都是引用类型或不知道是值类型的类型参数，并且如果 A 具有到 B 的引用、数组或类型参数转换，则存在从类型 A 的数组到具有相同秩的类型 B 的数组的转换。这种关系称为数组协变。数组协变尤其意味着，元素类型为 B 的数组的元素实际上可能是元素类型为 A 的数组的元素，前提是 A 和 B 都为引用类型，并且 B 具有到 A 的引用转换或数组转换。在下面的示例中，第二个 F 调用导致引发 System.ArrayTypeMismatchException 异常，原因是 b 的实际元素类型是 String，而不是 Object：

Module Test  
 Sub F(ByRef x As Object)  
 End Sub  
  
 Sub Main()  
 Dim a(10) As Object  
 Dim b() As Object = New String(10) {}  
 F(a(0)) ' OK.  
 F(b(1)) ' Not allowed: System.ArrayTypeMismatchException.  
 End Sub  
End Module

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。

Module Test  
 Sub Fill(array() As Object, index As Integer, count As Integer, \_  
 value As Object)  
 Dim i As Integer  
  
 For i = index To (index + count) - 1  
 array(i) = value  
 Next i  
 End Sub  
  
 Sub Main()  
 Dim strings(100) As String  
  
 Fill(strings, 0, 101, "Undefined")  
 Fill(strings, 0, 10, Nothing)  
 Fill(strings, 91, 10, 0)  
 End Sub  
End Module

在此示例中，方法 Fill 中对 array(i) 的赋值隐式包括运行时检查，它可确保由变量 value 引用的对象是 Nothing 或是与数组 array 的实际元素类型兼容的类型的实例。在方法 Main 中，方法 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 异常。发生该异常的原因是，Integer 不能存储在 String 数组中。

如果数组元素类型之一是具有运行时值类型的类型形参，则会引发 System.InvalidCastException 异常。例如：

Module Test  
 Sub F(Of T As U, U)(x() As T)  
 Dim y() As U = x  
 End Sub  
  
 Sub Main()  
 ' F will throw an exception because Integer() cannot be  
 ' converted to Object()  
 F(New Integer() { 1, 2, 3 })  
 End Sub  
End Module

在枚举类型的数组与枚举类型的基础类型的数组或具有相同基础类型的其他枚举类型的数组之间也存在转换，前提是这些数组的秩相同。

Enum Color As Byte  
 Red  
 Green  
 Blue  
End Enum  
  
Module Test  
 Sub Main()  
 Dim a(10) As Color  
 Dim b() As Integer  
 Dim c() As Byte  
  
 b = a ' Error: Integer is not the underlying type of Color  
 c = a ' OK  
 a = c ' OK  
 End Sub  
End Module

在此示例中，Color 数组与 Color 的基础类型 Byte 的数组相互转换。但是，转换为 Integer 数组时会发生错误，因为 Integer 不是 Color 的基础类型。

以下任何条件为真时，秩为 1 的 A() 类型数组还可转换为 System.Collections.Generic 中的集合接口类型 IList(Of B)、IReadOnlyList(Of B)、ICollection(Of B)、IReadOnlyCollection(Of B) 和 IEnumerable(Of B)：

A 和 B 都是引用类型，或是未已知为值类型的类型形参；并且 A 具有到 B 的扩大引用、数组或类型形参转换；或

A 和 B 都是相同基础类型的枚举类型；或者

A 和 B 之一是枚举类型，另一个为其基础类型。

具有任何秩的任何 A 类型数组还可转换为 System.Collections 中的非泛型集合接口类型 IList、ICollection 和 IEnumerable。

可以使用 For Each 或通过直接调用 GetEnumerator 方法来迭代生成的接口。将秩为 1 数组转换为 IList 或 ICollection 的泛型或非泛型形式后，也可通过索引获取元素。将秩为 1 的数组转换为 IList 的泛型或非泛型形式后，也可按索引设置元素，并遵循前面所述的相同运行时数组协变检查。其他所有接口方法的行为由 VB 语言规范定义，依赖于基础运行时。

## 值类型转换

可以将一个值类型值转换为它的基引用类型之一或转换为它通过称为“装箱”的进程实现的接口类型。将一个值类型值装箱之后，会将该值从它所在的位置复制到 .NET Framework 堆上。然后返回对该堆上此位置的引用，可以将该引用存储在一个引用类型变量中。该引用也称为值类型的装箱实例。装箱实例具有与引用类型（而不是值类型）相同的语义。

可以通过一个称为“取消装箱”的进程将装箱值类型转换回它们的原始值类型。在将装箱值类型取消装箱之后，可将该值从堆复制到一个变量位置。从此刻起，它的行为就像是一个值类型。在将值类型取消装箱时，该值必须是一个 null 值，或者是该值类型的一个实例。否则会引发 System.InvalidCastException 异常。如果该值是枚举类型的一个实例，则也可以将该值取消装箱到该枚举类型的基础类型或具有相同基础类型的其他枚举类型。null 值被视为文本 Nothing。

为了很好地支持可以为 null 的值类型，在执行装箱和拆箱时，将对值类型 System.Nullable(Of T) 进行特殊处理。对类型 Nullable(Of T) 的值进行装箱会产生类型 T 的装箱值（如果该值的 HasValue 属性为 True）或 Nothing 值（如果该值的 HasValue 属性为 False）。将类型 T 的值拆箱为 Nullable(Of T) 会生成 Nullable(Of T) 的实例，其 Value 属性为装箱值，HasValue 属性为 True。值 Nothing 可以拆箱为任何 T 的 Nullable(Of T)，产生 HasValue 属性为 False 的值。由于装箱值类型的行为类似于引用类型，因此可以创建对同一个值的多个引用。这对于基元类型和枚举类型来说并不重要，因为这些类型的实例是不可变的。也就是说，不可能修改这些类型的装箱实例，因此无法观察到存在对同一个值的多个引用的情况。

另一方面，如果结构的实例变量是可以访问的，或者它的方法或属性将修改其实例变量，则这些结构可能是可变的。如果使用一个对装箱结构的引用来修改结构，则对该装箱结构的所有引用都将看到变化。因为可能获得意外的结果，因此在将类型化为 Object 的值从一个位置复制到另一个位置时，将自动在堆上克隆装箱值类型，而不是仅复制它们的引用。例如：

Class Class1  
 Public Value As Integer = 0  
End Class  
  
Structure Struct1  
 Public Value As Integer  
End Structure  
  
Module Test  
 Sub Main()  
 Dim val1 As Object = New Struct1()  
 Dim val2 As Object = val1  
  
 val2.Value = 123  
  
 Dim ref1 As Object = New Class1()  
 Dim ref2 As Object = ref1  
  
 ref2.Value = 123  
  
 Console.WriteLine("Values: " & val1.Value & ", " & val2.Value)  
 Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)  
 End Sub  
End Module

该程序的输出为：

Values: 0, 123  
Refs: 123, 123

向局部变量 val2 的字段赋值不会影响局部变量 val1 的字段，因为在将装箱的 Struct1 赋给 val2 时，会创建该值的一个副本。相反，赋值语句 ref2.Value = 123 对 ref1 和 ref2 引用的对象都会产生影响。

批注

类型化为 System.ValueType 的装箱结构不进行结构复制，因为 System.ValueType 无法进行后期绑定。

此规则有一个例外，就是将在赋值时复制装箱值类型。如果将一个装箱值类型引用存储在其他类型内，则将不会复制内部引用。例如：

Structure Struct1  
 Public Value As Object  
End Structure  
  
Module Test  
 Sub Main()  
 Dim val1 As Struct1  
 Dim val2 As Struct1  
  
 val1.Value = New Struct1()  
 val1.Value.Value = 10  
  
 val2 = val1  
 val2.Value.Value = 123  
 Console.WriteLine("Values: " & val1.Value.Value & ", " & \_  
 val2.Value.Value)  
 End Sub  
End Module

该程序的输出为：

Values: 123, 123

这是因为在复制值时将不复制内部装箱值。因此，val1.Value 和 val2.Value 都引用同一装箱值类型。

批注

不复制内部装箱值类型是 .NET 类型系统的限制，这是为了确保每当所复制的 Object 类型的值开销极大时，将对所有内部装箱值类型进行复制。

如上文所述，只能将装箱值类型取消装箱为它们的原始类型。但是，类型化为 Object 的装箱基元类型有特殊的处理方式。可以将它们转换为它们可以转换到的任何其他基元类型。例如：

Module Test  
 Sub Main()  
 Dim o As Object = 5  
 Dim b As Byte = CByte(o) ' Legal  
 Console.WriteLine(b) ' Prints 5  
 End Sub  
End Module

通常，装箱的 Integer 值 5 不能拆箱为 Byte 变量。但是，因为 Integer 和 Byte 是基元类型并且存在转换关系，所以可以进行这一转换。

请务必注意，将一个值类型转换为一个接口不同于约束为接口的泛型参数。当访问约束类型形参的接口成员（或调用 Object 的方法）时，不会像将值类型转换为接口以及对接口成员进行访问时那样发生装箱。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：

Interface ICounter  
 Sub Increment()  
 ReadOnly Property Value() As Integer  
End Interface  
  
Structure Counter  
 Implements ICounter  
  
 Dim \_value As Integer  
  
 Property Value() As Integer Implements ICounter.Value  
 Get  
 Return \_value  
 End Get  
 End Property  
  
 Sub Increment() Implements ICounter.Increment  
 value += 1  
 End Sub  
End Structure  
  
Module Test  
 Sub Test(Of T As ICounter)(x As T)  
 Console.WriteLine(x.value)  
 x.Increment() ' Modify x  
 Console.WriteLine(x.value)  
 CType(x, ICounter).Increment() ' Modify boxed copy of x  
 Console.WriteLine(x.value)  
 End Sub  
  
 Sub Main()  
 Dim x As Counter  
 Test(x)  
 End Sub  
End Module

对 Increment 的第一个调用修改变量 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

### 可为 null 的值类型转换

值类型 T 可以与可以为 null 的类型 T? 进行相互转换。如果要转换的值是 Nothing，则从 T? 到 T 的转换会引发 System.InvalidOperationException 异常。此外，如果 T 具有到 S 的内部转换，则 T? 具有到类型 S 的转换。如果 S 是值类型，则 T? 和 S? 之间存在以下内部转换：

从 T? 到 S? 的相同分类（收缩或扩大）转换。

从 T 到 S? 的相同分类（收缩或扩大）转换。

从 S? 到 T 的收缩转换。

例如，因为存在从 Integer 到 Long 的内部扩大转换，所以存在从 Integer? 到 Long? 的内部扩大转换：

Dim i As Integer? = 10  
Dim l As Long? = i

从 T? 转换为 S? 时，如果 T? 的值为 Nothing，则 S? 的值将为 Nothing。从 S? 转换为 T 或从 T? 转换为 S 时，如果 T? 或 S? 的值为 Nothing，则会引发 System.InvalidCastException 异常。

对可以为 null 的值类型 T? 装箱时，结果是类型 T 的装箱值，而不是类型 T? 的装箱值，这是基础类型 System.Nullable(Of T) 的行为造成的。相反，当拆箱为可以为 null 的值类型 T? 时，该值将由 System.Nullable(Of T) 进行包装，并且 Nothing 将拆箱为类型 T? 的 null 值。例如：

Dim i1? As Integer = Nothing  
Dim o1 As Object = i1  
  
Console.WriteLine(o1 Is Nothing) ' Will print True  
o1 = 10  
i1 = CType(o1, Integer?)  
Console.WriteLine(i1) ' Will print 10

此行为的一个副作用是，可以为 null 的值类型 T? 看起来将实现 T 的所有接口，因为将值类型转换为接口需要对该类型进行装箱。因此，T? 可转换为 T 可以转换为的所有接口。但请务必注意，可以为 null 的值类型 T? 不会出于泛型约束检查或反射的目的而实际实现 T 的接口。例如：

Interface I1  
End Interface  
  
Structure T1  
 Implements I1  
 ...  
End Structure  
  
Module Test  
 Sub M1(Of T As I1)(ByVal x As T)  
 End Sub  
  
 Sub Main()  
 Dim x? As T1 = Nothing  
 Dim y As I1 = x ' Valid  
 M1(x) ' Error: x? does not satisfy I1 constraint  
 End Sub  
End Module

## 字符串转换

将 Char 转换为 String 会生成一个字串符，该字符的值即是该字符串的第一个字符。将 String 转换为 Char 会生成一个字符，该字符的值即是字符串的第一个字符。将 Char 数组转换为 String 会生成一个字符串，字符串中的字符即是数组的元素。将 String 转换为 Char 数组会生成一个字符数组，其中的元素是字符串的字符。

String 与 Boolean、Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Decimal、Single、Double、Date 之间的精确相互转换不属于本规范的讨论范围，这些转换的实现都依赖于各自在细节上的独特之处。字符串转换始终会考虑运行时环境的当前区域性。因此，必须在运行时执行这些转换。

## 扩大转换

扩大转换永远不会溢出，但可能会伴有精度损失。下列转换是扩大转换：

标识/默认转换

从一个类型到其本身。

从为 lambda 方法重新分类而生成的匿名委托类型到任何具有相同签名的委托类型。

从文本 Nothing 到某个类型。

数值转换

从 Byte 到 UShort、Short、UInteger、Integer、ULong、Long、Decimal、Single 或 Double。

从 SByte 到 Short、Integer、Long、Decimal、Single 或 Double。

从 UShort 到 UInteger、Integer、ULong、Long、Decimal、Single 或 Double。

从 Short 到 Integer、Long、Decimal、Single 或 Double。

从 UInteger 到 ULong、Long、Decimal、Single 或 Double。

从 Integer 到 Long、Decimal、Single 或 Double。

从 ULong 到 Decimal、Single 或 Double。

从 Long 到 Decimal、Single 或 Double。

从 Decimal 到 Single 或 Double。

从 Single 到 Double。

从文本 0 到一个枚举类型。

批注

从 0 到任何枚举类型的转换是扩大转换，用于简化测试标志。例如，如果 Values 是值为 One 的枚举类型，则可以通过 (v And Values.One) = 0 来测试 Values 类型的变量 v。

从一个枚举类型到它的基础数值类型或到它的基础数值类型扩大转换到的一个数值类型。

从 ULong、Long、UInteger、Integer、UShort、Short、Byte 或 SByte 类型的常量表达式到较窄的类型，前提是常量表达式的值在目标类型的范围之内。

注意 从 UInteger 或 Integer 到 Single、从 ULong 或 Long 到 Single 或 Double、或从 Decimal 到 Single 或 Double 的转换可能存在精度损失，但绝不会影响到数量级。其他扩大数值转换决不会丢失任何信息。

引用转换

从引用类型到基类型。

从引用类型到接口类型，前提是该类型实现该接口或一个变量兼容接口。

从接口类型到 Object。

从接口类型到变量兼容接口类型。

从委托类型到变量兼容委托类型。

注意：这些规则暗含其他的许多引用转换。例如，匿名委托是继承自 System.MulticastDelegate 的引用类型；数组类型是继承自 System.Array 的引用类型；匿名类型是继承自 System.Object 的引用类型。

匿名委托转换

从为 lambda 方法重新分类而生成的匿名委托类型到任何较宽的委托类型。

数组转换

从元素类型为 SE 的数组类型 S 到元素类型为 TE 的数组类型 T（前提是以下所列条件均成立）：

S 和 T 只有元素类型不同。

SE 和 TE 都是引用类型，或者都是已知将为引用类型的类型形参。

从 SE 到 TE 存在扩大引用、数组或类型形参转换。

从枚举的元素类型为 SE 的数组类型 S 到元素类型为 TE 的数组类型 T（前提是以下所列条件均成立）：

S 和 T 只有元素类型不同。

TE 是 SE 的基础类型。

从具有枚举元素类型 SE 的秩为 1 的数组类型 S 到 System.Collections.Generic.IList(Of TE)、IReadOnlyList(Of TE)、ICollection(Of TE)、IReadOnlyCollection(Of TE) 和 IEnumerable(Of TE)，前提是下面的一个条件成立：

SE 和 TE 都是引用类型，或是已知为引用类型的类型形参，并且从 SE 到 TE 存在扩大的引用、数组或类型形参转换；或

TE 是 SE 的基础类型；或

TE 与 SE 相同。值类型转换

从值类型到基类型。

从值类型到该类型所实现的接口类型。

可为 null 的值类型转换

从类型 T 到类型 T?。

从类型 T? 到类型 S?，其中从类型 T 到类型 S 存在扩大转换。

从类型 T 到类型 S?，其中从类型 T 到类型 S 存在扩大转换。

从类型 T? 到类型 T 所实现的接口类型。

字符串转换

从 Char 到 String。

从 Char() 到 String。

类型形参转换

从类型形参到 Object。

从类型参数到接口类型约束或与接口类型约束兼容的任何接口变量。

从类型参数到由类约束实现的接口。

从类型参数到与由类约束实现的接口兼容的接口变量。

从类型参数到类约束或该类约束的基类型。

从类型形参 T 到类型形参约束 TX 或 TX 可扩大转换为的任何类型。

## 收缩转换

收缩转换是这样一些转换：它们不能保证总是成功，知道有可能丢失信息，并且转换所跨的类型域显著不同以致于值得使用收缩表示法。下列转换属于收缩转换：

布尔转换

从 Boolean 到 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Decimal、Single 或 Double。

从 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Decimal、Single 或 Double 到 Boolean。

数值转换

从 Byte 到 SByte。

从 SByte 到 Byte、UShort、UInteger 或 ULong。

从 UShort 到 Byte、SByte 或 Short。

从 Short 到 Byte、SByte、UShort、UInteger 或 ULong。

从 UInteger 到 Byte、SByte、UShort、Short 或 Integer。

从 Integer 到 Byte、SByte、UShort、Short、UInteger 或 ULong。

从 ULong 到 Byte、SByte、UShort、Short、UInteger、Integer 或 Long。

从 Long 到 Byte、SByte、UShort、Short、UInteger、Integer 或 ULong。

从 Decimal 到 Byte、SByte、UShort、Short、UInteger、Integer、ULong 或 Long。

从 Single 到 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long 或 Decimal。

从 Double 到 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Decimal 或 Single。

从数值类型到枚举类型。

从一个枚举类型到它的基础数值类型收缩转换到的一个数值类型。

从一个枚举类型到另一个枚举类型。

引用转换

从引用类型到派生程度更大的类型。

从类类型到接口类型，前提是该类类型不实现该接口类型或与它兼容的接口类型变量。

从接口类型到类类型。

从一个接口类型到另一个接口类型，前提是这两个类型之间没有继承关系，并且它们不是变量兼容的。

匿名委托转换

从为 lambda 方法重新分类而生成的匿名委托类型到任何较窄的委托类型。

数组转换

从元素类型为 SE 的数组类型 S 到元素类型为 TE 的数组类型 T（前提是以下所列条件均成立）：

S 和 T 只有元素类型不同。

SE 和 TE 都是引用类型，或者都不是已知将为值类型的类型形参。

从 SE 到 TE 存在收缩引用、数组或类型形参转换。

从枚举的元素类型为 SE 的数组类型 S 到枚举的元素类型为 TE 的数组类型 T（前提是以下所列条件均成立）：

S 和 T 只有元素类型不同。

SE 是 TE 的基础类型，或者它们是基础类型相同的不同枚举类型。

从具有枚举元素类型 SE 的秩为 1 的数组类型 S 到 IList(Of TE)、IReadOnlyList(Of TE)、ICollection(Of TE)、IReadOnlyCollection(Of TE) 和 IEnumerable(Of TE)，前提是下面的一个条件成立：

SE 和 TE 都是引用类型，或是已知为引用类型的类型形参，并且从 SE 到 TE 存在缩小的引用、数组或类型形参转换；或

SE 是 TE 的基础类型，或者它们是基础类型相同的不同枚举类型。

值类型转换

从引用类型到派生程度更大的值类型。

从一个接口类型到一个值类型，前提是该值类型实现该接口类型。

可为 null 的值类型转换

从类型 T? 到类型 T。

从类型 T? 到类型 S?，其中从类型 T 到类型 S 存在收缩转换。

从类型 T 到类型 S?，其中从类型 T 到类型 S 存在收缩转换。

从类型 S? 到类型 T，其中从类型 S 到类型 T 存在转换。

字符串转换

从 String 到 Char。

从 String 到 Char()。

从 String 到 Boolean 和从 Boolean 到 String。

在 String 和 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Decimal、Single 或 Double 之间的转换。

从 String 到 Date 和从 Date 到 String。

类型形参转换

从 Object 到类型形参。

从类型参数到接口类型，前提是该类型参数未约束到该接口或未约束到实现该接口的类。

从接口类型到类型参数。

从类型参数到类约束的派生类型。

从类型形参 T 到类型形参约束 TX 可收缩转换为的任何类型。

## 类型形参转换

类型参数转换由对它们所实施的约束（如果有）确定。类型形参 T 始终可以转换为其自身、可与 Object 相互转换，也可以与任何接口类型相互转换。请注意，如果类型 T 是运行时的值类型，则从 T 到 Object 或接口类型的转换将是装箱转换，而从 Object 或接口类型到 T 的转换将是取消装箱转换。具有类约束 C 的类型形参定义了类型形参与 C（及其基类）之间的附加转换。具有类型形参约束 TX 的类型形参 T 定义了到 TX（以及 TX 可转换为的任何类型）的转换。

元素类型是具有接口约束 I 的类型形参的数组与元素类型是 I 的数组具有相同的协变数组转换，前提是该类型形参还具有 Class 或类约束（因为只有引用数组元素类型才可以是协变的）。元素类型是具有类约束 C 的类型形参的数组与元素类型是 C 的数组具有相同的协变数组转换。

以上转换规则不允许从无约束的类型参数到非接口类型的转换，这可能有点奇怪。其原因是为了防止在这些转换的语义上产生混淆。例如，请考虑下面的声明：

Class X(Of T)  
 Public Shared Function F(t As T) As Long   
 Return CLng(t) ' Error, explicit conversion not permitted  
 End Function  
End Class

如果允许从 T 转换到 Integer，则人们很容易认为 X(Of Integer).F(7) 将返回 7L。但结果不是这样，因为仅当在编译时已经知道类型将是数值时，才会考虑数值转换。为了使语义清楚，必须将上面的示例改写为：

Class X(Of T)  
 Public Shared Function F(t As T) As Long  
 Return CLng(CObj(t)) ' OK, conversions permitted  
 End Function  
End Class

## 用户定义的转换

内部转换是由语言定义的转换（即本规范中所列转换），而用户定义的转换是通过重载 CType 运算符定义的转换。在两个类型之间进行转换时，如果没有适用的内部转换，则将会考虑用户定义的转换。如果有一个对于源和目标类型来说最具体的用户定义转换，则将使用该用户定义转换。否则将发生编译时错误。最具体的转换是指，该转换的操作数“最接近”于源类型，并且它的结果类型“最接近”于目标类型。在确定要使用的用户定义转换时，将使用最具体的扩大转换；如果没有最具体的扩大转换，则使用最具体的收缩转换。如果没有最具体的收缩转换，则该转换未定义，将发生编译时错误。

以下各节介绍如何确定最具体的转换。它们使用了下列术语：

如果存在从类型 A 到类型 B 的内部扩大转换，并且 A 和 B 都不是接口，则称 A 包含于 B，B 包含 A。

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。直观地讲，包含程度最大的类型是集中的“最大”类型，每个其他类型均可通过扩大转换而转换为该类型。

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。直观地讲，被包含程度最大的类型是集中的“最小”类型，该类型可通过收缩转换而转换为每个其他类型。

在为类型 T? 收集候选用户定义转换时，将使用由 T 定义的用户定义转换运算符。如果要转换到的类型也是可以为 null 的值类型，则会提升 T 的所有仅涉及不可以为 null 的值类型的用户定义转换。从 T 到 S 的转换运算符提升为从 T? 到 S? 的转换，并通过将 T? 转换为 T 对其进行计算，如有必要，然后再计算从 T 到 S 的用户定义转换运算符，必要时再将 S 转换为 S?。但是，如果要转换的值为 Nothing，则提升转换运算符直接转换为类型化为 S? 的 Nothing 值。例如：

Structure S  
 ...  
End Structure  
  
Structure T  
 Public Shared Widening Operator CType(ByVal v As T) As S  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim x As T?  
 Dim y As S?  
  
 y = x ' Legal: y is still null  
 x = New T()  
 y = x ' Legal: Converts from T to S  
 End Sub  
End Module

在解析转换时，用户定义的转换运算符始终优先于提升转换运算符。例如：

Structure S  
 ...  
End Structure  
  
Structure T  
 Public Shared Widening Operator CType(ByVal v As T) As S  
 ...  
 End Operator  
  
 Public Shared Widening Operator CType(ByVal v As T?) As S?  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim x As T?  
 Dim y As S?  
  
 y = x ' Calls user-defined conversion, not lifted conversion  
 End Sub  
End Module

在运行时，计算用户定义的转换可包含最多三个步骤：

首先，如果有必要，使用一个内部转换将值从源类型转换到操作数类型。

然后，调用用户定义的转换。

最后，如果有必要，使用一个内部转换将用户定义转换的结果转换到目标类型。

请务必注意，用户定义转换的计算决不会包含一个以上用户定义的转换运算符。

### 最具体的扩大转换

确定两个类型之间的最具体的用户定义扩大转换运算符是通过下列步骤完成的：

首先，收集所有候选转换运算符。候选转换运算符是源类型中的所有用户定义的扩大转换运算符以及目标类型中的所有用户定义的扩大转换运算符。

然后，从该集中删除所有不适用的转换运算符。一个转换运算符在以下条件下适用于一个源类型和目标类型：存在从源类型到操作数类型的内部扩大转换运算符，并且存在从该运算符的结果到目标类型的内部扩大转换运算符。如果没有适用的转换运算符，则没有最具体的扩大转换。

然后将确定适用的转换运算符的最具体源类型：

如果任何转换运算符直接从源类型进行转换，则该源类型就是最具体的源类型。

否则，最具体的源类型是转换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，则没有最具体的扩大转换。

然后将确定适用的转换运算符的最具体目标类型：

如果任何转换运算符直接转换到目标类型，则该目标类型就是最具体的目标类型。

否则，最具体的目标类型是转换运算符的合并目标类型集中包含程度最大的类型。如果找不到包含程度最大的类型，则没有最具体的扩大转换。

如果一个转换运算符从最具体的源类型精确转换到最具体的目标类型，则该转换运算符就是最具体的转换运算符。如果这种运算符存在一个以上，则将没有最具体的扩大转换。

### 最具体的收缩转换

确定两个类型之间的最具体的用户定义收缩转换运算符是通过下列步骤完成的：

首先，收集所有候选转换运算符。候选转换运算符是源类型中的所有用户定义的转换运算符以及目标类型中的所有用户定义的转换运算符。

然后，从该集中删除所有不适用的转换运算符。一个转换运算符在以下条件下适用于一个源类型和目标类型：存在从源类型到操作数类型的内部转换运算符，并且存在从该运算符的结果到目标类型的内部转换运算符。如果没有适用的转换运算符，则没有最具体的收缩转换。

然后将确定适用的转换运算符的最具体源类型：

如果任何转换运算符直接从源类型进行转换，则该源类型就是最具体的源类型。

否则，如果任何转换运算符从包含该源类型的类型进行转换，则最具体的源类型是这些转换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，则没有最具体的收缩转换。

否则，最具体的源类型是转换运算符的合并源类型集中包含程度最大的类型。如果找不到包含程度最大的类型，则没有最具体的收缩转换。

然后将确定适用的转换运算符的最具体目标类型：

如果任何转换运算符直接转换到目标类型，则该目标类型就是最具体的目标类型。

否则，如果任何转换运算符转换到由该目标类型所包含的类型，则最具体的目标类型是这些转换运算符的合并源类型集中包含程度最大的类型。如果找不到包含程度最大的类型，则没有最具体的收缩转换。

否则，最具体的目标类型是转换运算符的合并目标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，则没有最具体的收缩转换。

如果一个转换运算符从最具体的源类型精确转换到最具体的目标类型，则该转换运算符就是最具体的转换运算符。如果这种运算符存在一个以上，则将没有最具体的收缩转换。

## 本机转换

有一些转换属于“本机转换”，因为它们是由 .NET Framework 以本机方式提供支持的。这些转换可以通过使用 DirectCast 和 TryCast 转换运算符以及其他特殊行为进行优化。属于本机转换的转换包括：标识转换、默认转换、引用转换、数组转换、值类型转换和类型参数转换。

## 主导类型

对于给定类型集，在某些情况下（如类型推断）经常需要确定该集的主导类型。类型集的主导类型可通过先将一个或更多其他类型没有隐式转换到的任何类型删除来加以确定。如果此时没有剩下任何类型，则没有主导类型。主导类型就是其余类型的被包含程度最大的类型。如果有一个以上被包含程度最大的类型，则将没有主导类型。

# 类型成员

类型成员定义存储位置和可执行代码。它们可以是方法、构造函数、事件、常量、变量和属性。

## 接口方法实现

方法、事件和属性可以实现接口成员。若要实现接口成员，成员声明应指定 Implements 关键字并列出一个或多个接口成员。实现接口成员的方法和属性是隐式 NotOverridable，除非它们声明为 MustOverride、Overridable 或重写其他成员。如果实现接口成员的成员为 Shared，则会发生错误。成员的可访问性对其实现接口成员的能力没有任何影响。

若要使接口实现有效，包含类型的实现列表必须为包含兼容成员的接口指定名称。兼容成员是其签名与实现成员的签名相匹配的成员。如果实现的是泛型接口，则在检查兼容性时 Implements 子句中提供的类型参数将替换到签名中。例如：

Interface I1(Of T)  
 Sub F(x As T)  
End Interface  
  
Class C1  
 Implements I1(Of Integer)  
  
 Sub F(x As Integer) Implements I1(Of Integer).F  
 End Sub  
End Class  
  
Class C2(Of U)  
 Implements I1(Of U)  
  
 Sub F(x As U) Implements I1(Of U).F  
 End Sub  
End Class

如果使用委托类型声明的事件实现的是接口事件，则兼容事件是其基础委托类型为同一类型的事件。否则，该事件使用来自它所实现的接口事件的委托类型。如果这样的事件实现的是多个接口事件，则所有接口事件都必须具有相同的基础委托类型。例如：

Interface ClickEvents  
 Event LeftClick(x As Integer, y As Integer)  
 Event RightClick(x As Integer, y As Integer)  
End Interface  
  
Class Button  
 Implements ClickEvents  
  
 ' OK. Signatures match, delegate type = ClickEvents.LeftClickHandler.  
 Event LeftClick(x As Integer, y As Integer) \_  
 Implements ClickEvents.LeftClick  
  
 ' OK. Signatures match, delegate type = ClickEvents.RightClickHandler.  
 Event RightClick(x As Integer, y As Integer) \_  
 Implements ClickEvents.RightClick  
End Class  
  
Class Label  
 Implements ClickEvents  
  
 ' Error. Signatures match, but can't be both delegate types.  
 Event Click(x As Integer, y As Integer) \_  
 Implements ClickEvents.LeftClick, ClickEvents.RightClick  
End Class

使用类型名称、句点和标识符指定实现列表中的接口成员。类型名称必须是实现列表中的接口或实现列表中接口的基接口，而标识符必须是指定接口的成员。单个成员可以实现多个匹配的接口成员。

Interface ILeft  
 Sub F()  
End Interface  
  
Interface IRight  
 Sub F()  
End Interface  
  
Class Test  
 Implements ILeft, IRight  
  
 Sub F() Implements ILeft.F, IRight.F  
 End Sub  
End Class

如果所实现的接口成员因多重接口继承而在所有显式实现的接口中不可用，则实现成员必须显式引用该成员在其中可用的基接口。例如，如果 I1 和 I2 包含成员 M，而 I3 继承自 I1 和 I2，则实现 I3 的类型也会实现 I1.M 和 I2.M。如果接口隐藏多重继承成员，则实现类型必须实现这些继承成员及隐藏它们的成员。

Interface ILeft  
 Sub F()  
End Interface  
  
Interface IRight  
 Sub F()  
End Interface  
  
Interface ILeftRight  
 Inherits ILeft, IRight  
  
 Shadows Sub F()  
End Interface  
  
Class Test  
 Implements ILeftRight  
  
 Sub LeftF() Implements ILeft.F  
 End Sub  
  
 Sub RightF() Implements IRight.F  
 End Sub  
  
 Sub LeftRightF() Implements ILeftRight.F  
 End Sub  
End Class

如果所实现的接口成员的包含接口是泛型，则必须提供与所实现的接口相同的类型参数。例如：

Interface I1(Of T)  
 Function F() As T  
End Interface  
  
Class C1  
 Implements I1(Of Integer)  
 Implements I1(Of Double)  
  
 Function F1() As Integer Implements I1(Of Integer).F  
 End Function  
  
 Function F2() As Double Implements I1(Of Double).F  
 End Function  
  
 ' Error: I1(Of String) is not implemented by C1  
 Function F3() As String Implements I1(Of String).F  
 End Function  
End Class  
  
Class C2(Of U)  
 Implements I1(Of U)  
  
 Function F() As U Implements I1(Of U).F  
 End Function  
End Class

ImplementsClause ::= [ Implements ImplementsList ]

ImplementsList ::=  
 InterfaceMemberSpecifier |  
 ImplementsList Comma InterfaceMemberSpecifier

InterfaceMemberSpecifier ::= NonArrayTypeName Period IdentifierOrKeyword

## 方法

方法包含程序的可执行语句。具有一个可选参数列表和一个可选返回值的方法不是共享方法，就是非共享方法。共享方法通过类或类的实例进行访问。非共享方法也称为实例方法，通过类的实例进行访问。下面的示例演示 Stack 类，该类具有多个共享方法（Clone 和 Flip）及多个实例方法（Push、Pop 和 ToString）：

Public Class Stack  
 Public Shared Function Clone(s As Stack) As Stack  
 ...  
 End Function  
  
 Public Shared Function Flip(s As Stack) As Stack  
 ...  
 End Function  
  
 Public Function Pop() As Object  
 ...  
 End Function  
  
 Public Sub Push(o As Object)  
 ...  
 End Sub   
  
 Public Overrides Function ToString() As String  
 ...  
 End Function   
End Class   
  
Module Test  
 Sub Main()  
 Dim s As Stack = New Stack()  
 Dim i As Integer  
  
 While i < 10  
 s.Push(i)  
 End While  
  
 Dim flipped As Stack = Stack.Flip(s)  
 Dim cloned As Stack = Stack.Clone(s)  
  
 Console.WriteLine("Original stack: " & s.ToString())  
 Console.WriteLine("Flipped stack: " & flipped.ToString())  
 Console.WriteLine("Cloned stack: " & cloned.ToString())  
 End Sub  
End Module

方法可以重载，这意味着只要具有唯一签名，多个方法就可以同名。方法的签名由其形参的数量和类型组成。具体而言，方法的签名不包含返回类型或形参修饰符，如 Optional、ByRef 或 ParamArray。下的示例演示一个具有多个重写的类：

Module Test  
 Sub F()  
 Console.WriteLine("F()")  
 End Sub   
  
 Sub F(o As Object)  
 Console.WriteLine("F(Object)")  
 End Sub  
  
 Sub F(value As Integer)  
 Console.WriteLine("F(Integer)")  
 End Sub   
  
 Sub F(a As Integer, b As Integer)  
 Console.WriteLine("F(Integer, Integer)")  
 End Sub   
  
 Sub F(values() As Integer)  
 Console.WriteLine("F(Integer())")  
 End Sub   
  
 Sub G(s As String, Optional s2 As String = 5)  
 Console.WriteLine("G(String, Optional String")  
 End Sub  
  
 Sub G(s As String)  
 Console.WriteLine("G(String)")  
 End Sub  
  
  
 Sub Main()  
 F()  
 F(1)  
 F(CType(1, Object))  
 F(1, 2)  
 F(New Integer() { 1, 2, 3 })  
 G("hello")  
 G("hello", "world")  
 End Sub  
End Module

该程序的输出为：

F()  
F(Integer)  
F(Object)  
F(Integer, Integer)  
F(Integer())  
G(String)  
G(String, Optional String)

批注

只有可选形参不同的重载可用于库的“版本控制”。例如，库的 v1 版可能包含具有以下可选形参的函数：

Sub fopen(fileName As String, Optional accessMode as Integer = 0)

然后该库的 v2 版要添加另一个可选形参“password”，并希望在不损害到源代码兼容性（这样，以前面向 v1 版的应用程序就可以重新编译）且不损害二进制代码兼容性（这样，以前引用 v1 版的应用程序现在无需重新编译就可引用 v2 版）的情况下实现。以下将是 v2 版的内容：

Sub fopen(file As String, mode as Integer)  
 Sub fopen(file As String, Optional mode as Integer = 0, \_  
 Optional pword As String = "")

请注意，公共 API 中的可选形参不符合 CLS。但是，至少可以通过 Visual Basic 、C# 4 和 F# 使用它们。

MethodMemberDeclaration ::= MethodDeclaration | ExternalMethodDeclaration

InterfaceMethodMemberDeclaration ::= InterfaceMethodDeclaration

### 常规、异步和迭代器方法声明

有两种类型的方法：子例程和函数；前者不返回值，后者返回值。仅当方法在接口中进行了定义或具有 MustOverride 修饰符时，才能忽略其主体和 End 构造。如果未在函数中指定返回类型并且使用的是严格语义，则会发生编译时错误；否则，返回类型是隐式 Object 或是该方法的类型字符的类型。方法的返回类型和参数类型的可访问域必须与方法本身的可访问域相同或者是其超集。

**常规方法**是不带 Async 和 Iterator 修饰符的方法。它可能是子例程或者是函数。10.1.1 一节详细说明了调用常规方法时将发生的情况。

**迭代器方法**是带有 Iterator 修饰符，但没有 Async 修饰符的方法。它必须是函数，其返回类型必须是 IEnumerator、IEnumerable 或 IEnumerator(Of T)，或者，对于某些 T 必须为 IEnumerable(Of T)，并且不能包含 ByRef 形参。10.1.2 一节详细说明了调用迭代器方法时将发生的情况。

**异步方法**是带有 Async 修饰符，但没有 Iterator 修饰符的方法。它必须是子例程，或者是返回类型为 Task（或者，对于某些 T 为 Task(Of T)）的函数，并且不能包含 ByRef 形参。10.1.3 一节详细说明了调用异步方法时将发生的情况。

如果某个方法不是这三种方法之一，就会发生编译时错误。子例程和函数声明的特殊之处在于，其开始和结尾语句必须都从逻辑行的开头处开始。此外，非 MustOverride 子例程或函数声明的主体也必须位于逻辑行的开头。例如：

Module Test  
 ' Illegal: Subroutine doesn’t start the line  
 Public x As Integer : Sub F() : End Sub  
  
 ' Illegal: First statement doesn’t start the line  
 Sub G() : Console.WriteLine("G")  
 End Sub  
  
 ' Illegal: End Sub doesn’t start the line  
 Sub H() : End Sub  
End Module

MethodDeclaration ::=  
 SubDeclaration |  
 MustOverrideSubDeclaration |  
 FunctionDeclaration |  
 MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=  
 InterfaceSubDeclaration |  
 InterfaceFunctionDeclaration

SubSignature ::= Sub Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]

FunctionSignature ::= Function Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] [ As [ Attributes ] TypeName ]

SubDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] SubSignature [ HandlesOrImplements ] LineTerminator  
 Block  
 End Sub StatementTerminator

MustOverrideSubDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ SubSignature [ HandlesOrImplements ]  
 StatementTerminator

InterfaceSubDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] SubSignature StatementTerminator

FunctionDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] FunctionSignature [ HandlesOrImplements ]  
 LineTerminator  
 Block  
 End Function StatementTerminator

MustOverrideFunctionDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ FunctionSignature [ HandlesOrImplements ]  
 StatementTerminator

InterfaceFunctionDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] FunctionSignature StatementTerminator

ProcedureModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads |  
 Partial |  
 Iterator |  
 Async

MustOverrideProcedureModifier ::= ProcedureModifier | MustOverride

InterfaceProcedureModifier ::= Shadows | Overloads

HandlesOrImplements ::= HandlesClause | ImplementsClause

### 外部方法声明

外部方法声明引入了一种新方法，该方法的实现在程序外部提供。由于外部方法声明不提供实际的实现，因此它没有方法体或 End 构造。外部方法是隐式共享的，可能没有类型参数，因而可能无法处理事件或实现接口成员。如果未在所使用的函数和严格语义中指定返回类型，则会发生编译时错误。否则，返回类型是隐式 Object 或该方法的类型字符的类型。外部方法的返回类型和参数类型的可访问域必须与外部方法本身的可访问域相同或者是其超集。

外部方法的库子句声明指定实现该方法的外部文件的名称。可选的别名子句是一个字符串，该字符串指定该方法在外部文件中的数字序号（以 # 字符作为前缀）或名称。也可以指定单个字符集修饰符，以控制用于在调用外部方法期间封送字符串的字符集。Unicode 修饰符用于将所有字符串封送为 Unicode 值，Ansi 修饰符用于将所有字符串封送为 ANSI 值，而 Auto 修饰符用于根据基于方法名称或指定别名的 .NET Framework 规则封送字符串。如果未指定修饰符，则默认为 Ansi。

如果指定了 Ansi 或 Unicode，则在不加修改的情况下在外部文件中查找方法名称。如果指定 Auto，则方法名称查找因平台而异。如果平台被视为 ANSI 平台（例如，Windows 95、Windows 98、Windows ME），则在不进行修改的情况下查找方法名称。如果查找失败，则会追加 A 并重试查找。如果平台被视为 Unicode 平台（例如，Windows NT、Windows 2000、Windows XP），则追加 W 并查找名称。如果查找失败，则会去掉 W 并重试查找。例如：

Module Test  
 ' All platforms bind to "ExternSub".  
 Declare Ansi Sub ExternSub Lib "ExternDLL" ()  
  
 ' All platforms bind to "ExternSub".  
 Declare Unicode Sub ExternSub Lib "ExternDLL" ()  
  
 ' ANSI platforms: bind to "ExternSub" then "ExternSubA".  
 ' Unicode platforms: bind to "ExternSubW" then "ExternSub".  
 Declare Auto Sub ExternSub Lib "ExternDLL" ()  
End Module

要传递给外部方法的数据类型将根据 .NET Framework 数据封送约定进行封送，但有一个例外。按值（即 ByVal x As String）传递的字符串变量将封送为 OLE Automation BSTR 类型，并且在外部方法中对 BSTR 所做的更改将重新反映在字符串参数中。这是因为外部方法中的类型 String 是可变的，因而这是一种模拟该行为的特殊封送。按引用（即 ByRef x As String）传递的字符串参数将作为指针封送到 OLE 自动化 BSTR 类型。通过对形参指定 System.Runtime.InteropServices.MarshalAsAttribute 特性，可以重写这些特殊行为。

该示例演示如何使用外部方法：

Class Path  
 Declare Function CreateDirectory Lib "kernel32" ( \_  
 Name As String, sa As SecurityAttributes) As Boolean  
 Declare Function RemoveDirectory Lib "kernel32" ( \_  
 Name As String) As Boolean  
 Declare Function GetCurrentDirectory Lib "kernel32" ( \_  
 BufSize As Integer, Buf As String) As Integer  
 Declare Function SetCurrentDirectory Lib "kernel32" ( \_  
 Name As String) As Boolean  
End Class

ExternalMethodDeclaration ::=  
 ExternalSubDeclaration |  
 ExternalFunctionDeclaration

ExternalSubDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Sub Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ]   
 StatementTerminator

ExternalFunctionDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Function Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ]  
 StatementTerminator

ExternalMethodModifier ::= AccessModifier | Shadows | Overloads

CharsetModifier ::= Ansi | Unicode | Auto

LibraryClause ::= Lib StringLiteral

AliasClause ::= Alias StringLiteral

### 重写方法

Overridable 修饰符指示方法是可重写的。Overrides 修饰符指示方法重写具有相同签名的基类型可重写方法。NotOverridable 修饰符指示不能进一步重写可重写方法。MustOverride 修饰符指示必须在派生类中重写方法。

这些修饰符的某些组合是无效的：

Overridable 和 NotOverridable 相互排斥，不能组合使用。

MustOverride 隐含着 Overridable（因此不能指定它）并且不能和 NotOverridable 组合使用。

NotOverridable 不能和 Overridable 或 MustOverride 组合使用，而必须和 Overrides 组合使用。

Overrides 隐含着 Overridable（因此不能指定它）并且不能和 MustOverride 组合使用。

此外，可重写方法还存在其他限制：

MustOverride 方法不能包含方法体或 End 构造，不能重写其他方法，并且只能出现在 MustInherit 类中。

如果方法指定 Overrides，但没有要重写的匹配基方法，则会发生编译时错误。重写方法不能指定 Shadows。

如果重写方法的可访问域与被重写方法的可访问域不同，则前一方法不能重写后一方法。有一种例外情况：在没有 Friend 访问的另一个程序集中重写 Protected Friend 方法的方法必须指定 Protected（而非 Protected Friend）。

Private 方法不能是 Overridable、NotOverridable 或 MustOverride，它们也不能重写其他方法。

NotInheritable 类中的方法不能声明为 Overridable 或 MustOverride。

下面的示例阐释可重写方法与不可重写方法之间的区别：

Class Base  
 Public Sub F()  
 Console.WriteLine("Base.F")  
 End Sub  
  
 Public Overridable Sub G()  
 Console.WriteLine("Base.G")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Shadows Sub F()  
 Console.WriteLine("Derived.F")  
 End Sub  
  
 Public Overrides Sub G()  
 Console.WriteLine("Derived.G")  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim d As Derived = New Derived()  
 Dim b As Base = d  
  
 b.F()  
 d.F()  
 b.G()  
 d.G()  
 End Sub  
End Module

在下面的示例中，类 Base 引入了方法 F 和 Overridable 方法 G。类 Derived 引入了新方法 F，因而会隐藏继承的 F，此外还会重写继承的方法 G。该示例产生如下输出：

Base.F  
Derived.F  
Derived.G  
Derived.G

注意，语句 b.G() 调用 Derived.G，而不调用 Base.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 Derived），而不是该实例的编译时类型（即 Base）。

### 共享方法

Shared 修饰符指示方法是共享方法。共享方法不对类型的特定实例执行操作，可直接从类型进行调用，而不用通过类型的特定实例。不过，使用实例限定共享方法是有效的。在共享方法中引用 Me、MyClass 或 MyBase 是无效的。共享方法不能是 Overridable、NotOverridable 或 MustOverride，它们也不能重写方法。标准模块和接口中定义的方法不能指定 Shared，因为它们已隐含了 Shared。

在结构或类中未使用 Shared 修饰符声明的方法是实例方法。实例方法用于对类型的指定实例执行操作。实例方法只能通过类型的实例进行调用，可通过 Me 表达式引用实例。

下面的示例阐释访问共享成员和实例成员的规则：

Class Test  
 Private x As Integer  
 Private Shared y As Integer  
  
 Sub F()  
 x = 1 ' Ok, same as Me.x = 1.  
 y = 1 ' Ok, same as Test.y = 1.  
 End Sub  
  
 Shared Sub G()  
 x = 1 ' Error, cannot access Me.x.  
 y = 1 ' Ok, same as Test.y = 1.  
 End Sub  
  
 Shared Sub Main()  
 Dim t As Test = New Test()  
  
 t.x = 1 ' Ok.  
 t.y = 1 ' Ok.  
 Test.x = 1 ' Error, cannot access instance member through type.  
 Test.y = 1 ' Ok.  
 End Sub  
End Class

方法 F 演示在实例函数成员中，可使用标识符同时访问实例成员和共享成员。方法 G 演示在共享函数成员中，通过标识符访问实例成员是错误的。方法 Main 演示在成员访问表达式中，实例成员必须通过实例进行访问，但共享成员可通过类型或实例进行访问。

### 方法参数

形参 是一种变量，可用于将信息传入方法及从方法中传出信息。方法的参数由方法的参数列表定义，该列表由一个或多个以逗号分隔的参数组成。如果没有为要使用的参数和严格语义指定类型，则会发生编译时错误。否则，默认类型为 Object 或形参类型字符的类型。当一个形参包含 As 子句时，即使在宽松语义下，所有形参也都必须指定类型。

形参由修饰符 ByVal、ByRef、Optional 和 ParamArray 分别指定为值、引用、可选或形参数组形参。未指定 ByRef 或 ByVal 的形参默认为 ByVal。

参数名称的范围为整个方法体，并且始终是可公开访问的。方法调用会为形参创建特定于该调用的副本，该调用的实参列表会向新创建的形参赋予值或变量引用。由于外部方法声明和委托声明没有主体，因此允许形参列表中包含重复的形参名称，但建议不要这样做。

标识符后面可跟有可以为 null 的名称修饰符 ? 来指示该形参可以为 null，也可跟有数组名称修饰符来指示该形参是数组。它们可以组合使用，例如“ByVal x?() As Integer”。不允许使用显式数组界限；此外，如果存在可以为 null 的名称修饰符，则必须存在 As 子句。

ParameterList ::=  
 Parameter |  
 ParameterList Comma Parameter

Parameter ::=  
 [ Attributes ] [ ParameterModifier+ ] ParameterIdentifier [ As TypeName ]  
 [ *Equals* ConstantExpression ]

ParameterModifier ::= ByVal | ByRef | Optional | ParamArray

ParameterIdentifier ::= Identifier IdentifierModifiers

#### 值形参

值形参r用显式 ByVal 修饰符进行声明。如果使用了 ByVal 修饰符，则不能指定 ByRef 修饰符。值形参在调用该形参所属的成员时开始存在，并用调用中给定的实参的值初始化。在返回该成员时，值参数就不再存在。

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给出的实参。

在将某一实参的值传入方法时使用值形参，对该形参所做的修改不会影响原实参。值形参引用其自己的变量，该变量不同于对应实参的变量。此变量通过复制其对应参数的值进行初始化。下面的示例演示方法 F，该方法具有名为 p 的值形参：

Module Test  
 Sub F(p As Integer)  
 Console.WriteLine("p = " & p)  
 p += 1  
 End Sub   
  
 Sub Main()  
 Dim a As Integer = 1  
  
 Console.WriteLine("pre: a = " & a)  
 F(a)  
 Console.WriteLine("post: a = " & a)  
 End Sub  
End Module

该示例产生如下输出，即使修改了值形参 p 也是如此：

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

#### 引用参数

引用形参是用 ByRef 修饰符声明的形参。如果指定了 ByRef 修饰符，则不能使用 ByVal 修饰符。引用形参不创建新的存储位置。相反，引用形参表示的恰是在方法或构造函数调用中作为实参给出的那个变量。因此，引用参数的值总是与基础变量相同。

引用形参以两种模式使用：*别名* 或通过*复制再反向复制*。

**别名。**当形参充当调用方提供的实参的别名时，将使用引用形参。引用形参本身不定义变量，而是引用其对应实参的变量。对引用形参所做修改会直接并立即影响其对应实参。下面的示例演示具有两个引用形参的方法 Swap：

Module Test  
 Sub Swap(ByRef a As Integer, ByRef b As Integer)  
 Dim t As Integer = a  
 a = b  
 b = t  
 End Sub   
  
 Sub Main()  
 Dim x As Integer = 1  
 Dim y As Integer = 2  
  
 Console.WriteLine("pre: x = " & x & ", y = " & y)  
 Swap(x, y)  
 Console.WriteLine("post: x = " & x & ", y = " & y)  
 End Sub   
End Module

该程序的输出为：

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

对于类 Main 中方法 Swap 的调用，a 表示 x，b 表示 y。因此，该调用具有交换 x 和 y 的值的效果。

在采用引用参数的方法中，多个名称可能表示同一存储位置：

Module Test  
 Private s As String  
  
 Sub F(ByRef a As String, ByRef b As String)  
 s = "One"  
 a = "Two"  
 b = "Three"  
 End Sub  
  
 Sub G()  
 F(s, s)  
 End Sub  
End Module

在该示例中，G 中方法 F 的调用同时为 a 和 b 传递了对 s 的引用。因此，对于该调用，名称 s、aa 和 b 引用同一存储位置，并且这三个赋值都会修改实例变量 s。

**复制再反向复制。**如果要传递给引用形参的变量的类型与该引用形参的类型不兼容，或者将非变量（例如属性）作为实参传递给引用形参，或者调用是后期绑定的，则可能会将一个临时变量分配并传递给引用形参。所传入的值将在调用方法之前复制到此临时变量，并将在方法返回时复制回原始变量（如果有且可写入）。因此，引用参数可能不一定包含对所传入变量的具体存储的引用，并且只有在方法退出时对引用参数所做的任何更改才会反映在该变量中。例如：

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module Test  
 Sub F(ByRef b As Base)  
 b = New Base()  
 End Sub  
  
 Property G() As Base  
 Get  
 End Get  
 Set  
 End Set  
 End Property  
  
 Sub Main()  
 Dim d As Derived  
  
 F(G) ' OK.  
 F(d) ' Throws System.InvalidCastException after F returns.  
 End Sub  
End Module

在首次调用 F 时，将创建一个临时变量，将属性 G 的值赋予该变量，并将该值传入 F。从 F 返回时，临时变量中的值将赋回 G 的属性。在第二次调用时，将创建另一个临时变量，将 d 的值赋予该变量，并将该值传入 F。从 F 返回时，会将临时变量中的值强制转换回该变量的类型 Derived 并将该值赋予 d。由于所传回的值不能强制转换为 Derived，因此会在运行时引发异常。

#### 可选参数

可选形参是用 Optional 修饰符声明的。形参表中位于可选形参后面的形参也必须是可选的；如果不对后随形参指定 Optional 修饰符，则会触发编译时错误。某些可为 null 的类型 *T?* 或不可为 null 的类型 *T* 的可选形参必须指定常量表达式 *e*，以便在未指定实参的情况下作为默认值使用。如果 *e* 的计算结果为类型 Object 的 Nothing，则形参类型的默认值将用作该形参的默认值。否则，CType(e, T) 必须是一个常量表达式，并且将用作该形参的默认值。

形参初始值设定项只对可选形参有效。初始化将始终作为调用表达式的一部分执行，而不是在方法体自身中执行。

Module Test  
 Sub F(x As Integer, Optional y As Integer = 20)  
 Console.WriteLine("x = " & x & ", y = " & y)  
 End Sub  
  
 Sub Main()  
 F(10)  
 F(30,40)  
 End Sub  
End Module

该程序的输出为：

x = 10, y = 20  
x = 30, y = 40

不能在委托或事件声明中以及在 lambda 表达式中指定可选形参。

#### ParamArray 形参

ParamArray 形参用 ParamArray 修饰符声明。如果存在 ParamArray 修饰符，则必须指定 ByVal 修饰符，并且其他形参不能使用 ParamArray 修饰符。ParamArray 形参的类型必须是一维数组，并且该形参必须是形参列表中的最后一个形参。

ParamArray 形参表示不确定数目的 ParamArray 类型的形参。在方法自身中，ParamArray 形参将被视为其声明类型，没有特殊的语义。ParamArray 形参是隐式可选的，默认值为 ParamArray 类型的一维空数组。

ParamArray 允许在方法调用中通过以下两种方式之一指定形参：

为 ParamArray 指定的形参可以是其类型扩大为 ParamArray 类型的单个表达式。在此情况下，ParamArray 的作用与值形参完全相同。

或者，该调用也可以为 ParamArray 指定零个或多个形参，其中每个形参都是其类型可隐式转换为 ParamArray 的元素类型的表达式。在此情况下，该调用会创建一个 ParamArray 类型的实例（该实例的长度与实参个数相对应），用给定的实参值初始化该数组实例的元素，并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之外，ParamArray 与同一类型的值形参完全等效，如下面的示例所示。

Module Test  
 Sub F(ParamArray args() As Integer)  
 Dim i As Integer  
  
 Console.Write("Array contains " & args.Length & " elements:")  
 For Each i In args  
 Console.Write(" " & i)  
 Next i  
 Console.WriteLine()  
 End Sub  
  
 Sub Main()  
 Dim a As Integer() = { 1, 2, 3 }  
  
 F(a)  
 F(10, 20, 30, 40)  
 F()  
 End Sub  
End Module

该示例产生如下输出

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。第二次调用 F 时，会自动创建一个具有给定元素值的四元素数组，并将该数组实例作为值形参传递。同样，第三次调用 F 时，会创建一个零元素数组，并将该数组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 Integer() {10, 20, 30, 40})  
F(New Integer() {})

不能在委托或事件声明中指定 ParamArray 形参。

### 事件处理

方法可以通过声明方式在实例或共享变量中处理由对象引发的事件。若要处理事件，方法声明应指定 Handles 关键字并列出一个或多个事件。Handles 列表中的事件由以句点分隔的两个标识符指定：

第一个标识符必须是包含类型中指定 WithEvents 修饰符或者 MyBase、MyClass 或 Me 关键字的实例或共享变量；否则会发生编译时错误。此变量包含的对象将引发由此方法处理的事件。

第二个标识符必须指定第一个标识符的类型的成员。该成员必须是事件，并可以共享。如果为第一个标识符指定了共享变量，则事件必须是共享的，否则会导致错误。

如果语句 AddHandler E, AddressOf M 有效，则处理程序方法 M 也会视为事件 E 的有效事件处理程序。但与 AddHandler 语句不同的是，无论是否使用严格语义，显式事件处理程序都允许使用不带参数的方法处理事件：

Option Strict On  
  
Class C1  
 Event E(x As Integer)  
End Class  
  
Class C2  
 withEvents C1 As New C1()  
  
 ' Valid  
 Sub M1() Handles C1.E  
 End Sub  
  
 Sub M2()  
 ' Invalid  
 AddHandler C1.E, AddressOf M1  
 End Sub  
End Class

一个成员可以处理多个匹配的事件，并且多个方法可以处理一个事件。方法的可访问性对其处理事件的能力没有任何影响。下面的示例演示一个方法如何处理事件：

Class Raiser  
 Event E1()  
  
 Sub Raise()  
 RaiseEvent E1  
 End Sub  
End Class  
  
Module Test  
 WithEvents x As Raiser  
  
 Sub E1Handler() Handles x.E1  
 Console.WriteLine("Raised")  
 End Sub  
  
 Sub Main()  
 x = New Raiser()  
 x.Raise()  
 x.Raise()  
 End Sub  
End Module

这会输出如下结果：

Raised  
Raised

类型继承由其基类提供的所有事件处理程序。派生类型不能以任何方式改变它从其基类型继承的事件映射，但可以向事件添加其他处理程序。

HandlesClause ::= [ Handles EventHandlesList ]

EventHandlesList ::=  
 EventMemberSpecifier |  
 EventHandlesList Comma EventMemberSpecifier

EventMemberSpecifier ::=  
 Identifier Period IdentifierOrKeyword |  
 MyBase Period IdentifierOrKeyword |  
 MyClass Period IdentifierOrKeyword |  
 Me Period IdentifierOrKeyword

### 扩展方法

可使用扩展方法从类型声明外部向类型添加方法。扩展方法是应用了 System.Runtime.CompilerServices.ExtensionAttribute 特性的方法。它们只能在标准模块中声明，并且必须至少有一个参数，该参数指定方法扩展的类型。例如，以下扩展方法扩展类型 String：

Imports System.Runtime.CompilerServices  
  
Module StringExtensions  
 <Extension> \_  
 Sub Print(s As String)  
 Console.WriteLine(s)  
 End Sub  
End Module

批注

虽然 Visual Basic 要求在标准模块中声明扩展方法，但是其他语言（如 C#）可能允许在其他类型中声明此类方法。只要方法遵循此处概述的其他约定并且包含类型不是开放式泛型类型且不能实例化，Visual Basic 就会将其识别为扩展方法。

在调用扩展方法时，会在其中调用该方法的实例传递给第一个参数。第一个形参不能声明为 Optional 或 ParamArray。任何类型（包括类型参数）都可以显示为扩展方法的第一个参数。例如，以下各方法分别扩展类型 Integer()、任何实现 System.Collections.Generic.IEnumerable(Of T) 的类型以及所有类型：

Imports System.Runtime.CompilerServices  
  
Module Extensions  
 <Extension> \_  
 Sub PrintArray(a() As Integer)  
 ...  
 End Sub  
  
 <Extension> \_  
 Sub PrintList(Of T)(a As IEnumerable(Of T))  
 ...  
 End Sub  
  
 <Extension> \_  
 Sub Print(Of T)(a As T)  
 ...  
 End Sub  
End Module

如上述示例所示，接口也可以扩展。接口扩展方法提供方法的实现，因此实现已定义扩展方法的接口的类型仍然仅实现最初由该接口声明的成员。例如：

Imports System.Runtime.CompilerServices  
  
Interface IAction  
 Sub DoAction()  
End Interface  
  
Module IActionExtensions   
 <Extension> \_  
 Public Sub DoAnotherAction(i As IAction)   
 i.DoAction()  
 End Sub  
End Module  
  
Class C  
 Implements IAction  
  
 Sub DoAction() Implements IAction.DoAction  
 ...  
 End Sub  
  
 ' ERROR: Cannot implement extension method IAction.DoAnotherAction  
 Sub DoAnotherAction() Implements IAction.DoAnotherAction  
 ...  
 End Sub  
End Class

扩展方法也可以在其类型形参上包含类型约束，而且可以像非扩展泛型方法一样推断类型实参：

Imports System.Runtime.CompilerServices  
  
Module IEnumerableComparableExtensions  
 <Extension> \_  
 Public Function Sort(Of T As IComparable(Of T))(i As IEnumerable(Of T)) \_  
 As IEnumerable(Of T)   
 ...  
 End Function  
End Module

此外，还可以在所扩展的类型中通过隐式实例表达式访问扩展方法：

Imports System.Runtime.CompilerServices  
  
Class C1  
 Sub M1()  
 Me.M2()  
 M2()  
 End Sub  
End Class  
  
Module C1Extensions  
 <Extension> \_  
 Sub M2(c As C1)  
 ...  
 End Sub  
End Module

对于可访问性而言，也可以将扩展方法视为在其中声明它们的标准模块的成员；除了因其声明上下文而具有的访问权限之外，它们对所扩展的类型的成员没有额外的访问权限。

仅当标准模块方法位于范围内时，扩展方法才可用。否则，原始类型不会显示为已扩展。例如：

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Namespace N1  
 Module C1Extensions  
 <Extension> \_  
 Sub M1(c As C1)  
 ...  
 End Sub  
 End Module  
End Namespace  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Error: c has no member named "M1"  
 c.M1()  
 End Sub  
End Module

如果在只有类型上的扩展方法可用时引用该类型，仍会产生编译时错误。

请务必注意，扩展方法被视为在其中绑定成员的所有上下文中的类型的成员，如强类型 For Each 模式。例如：

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Class C1Enumerator  
 ReadOnly Property Current() As C1  
 Get  
 ...  
 End Get  
 End Property  
  
 Function MoveNext() As Boolean  
 ...  
 End Function  
End Class  
  
Module C1Extensions  
 <Extension> \_  
 Function GetEnumerator(c As C1) As C1Enumerator  
 ...  
 End Function  
End Module  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Valid  
 For Each o As Object In c  
 ...  
 Next o  
 End Sub  
End Module

也可以创建引用扩展方法的委托。因此，下面的代码：

Delegate Sub D1()  
  
Module Test  
 Sub Main()  
 Dim s As String = "Hello, World!"  
 Dim d As D1  
  
 d = AddressOf s.Print  
 d()  
 End Sub  
End Module

大致等效于：

Delegate Sub D1()  
  
Module Test  
 Sub Main()  
 Dim s As String = "Hello, World!"  
 Dim d As D1  
  
 d = CType([Delegate].CreateDelegate(GetType(D1), s, \_  
 GetType(StringExtensions).GetMethod("Print")), D1)  
 d()  
 End Sub  
End Module

批注

Visual Basic 通常会插入一个针对实例方法调用的检查，如果对其调用该方法的实例为 Nothing，则会导致发生 System.NullReferenceException。对于扩展方法，没有插入此检查的有效方式，因此扩展方法将需要显式检查 Nothing。

此外，在将值类型作为 ByVal 实参传递给类型化为接口的形参时，将对该值类型装箱。这意味着扩展方法的副作用将适用于结构的副本，而不是原始结构。尽管该语言对扩展方法的第一个实参不加限制，但建议不要使用扩展方法来扩展值类型，或者在扩展值类型时将第一个形参传递给 ByRef 以确保副作用应用于原始值。

### 分部方法

分部方法是一种指定签名而不是指定方法体的方法。方法体只能由具有相同名称和签名的另一个方法声明提供，最可能由类型的另一个分部声明提供。例如：

a.vb:

' Designer generated code  
Public Partial Class MyForm  
 Private Partial Sub ValidateControls()  
 End Sub  
  
 Public Sub New()  
 ' Initialize controls  
 ...  
  
 ValidateControls()  
 End Sub   
End Class

b.vb:

Public Partial Class MyForm  
 Public Sub ValidateControls()  
 ' Validation logic goes here  
 ...  
 End Sub  
End Class

在此示例中，类 MyForm 的分部声明声明了没有实现的分部方法 ValidateControls。分部声明中的构造函数调用了该分部方法，即使文件中未提供主体也是如此。然后，MyForm 的另一个分部声明提供了该方法的实现。

无论是否已提供主体，都可以调用分部方法；如果未提供方法主体，则将忽略该调用。例如：

Public Class C1  
 Private Partial Sub M1()  
 End Sub  
  
 Public Sub New()  
 ' Since no implementation is supplied, this call will not be made.  
 M1()  
 End Sub  
End Class

也将忽略作为参数传入分部方法调用的任何表达式，不会对其进行计算。

批注

这表明分部方法是一种十分有效的方式，可提供在两种分部类型上定义的行为，因为分部方法在未使用时不会产生开销。

分部方法声明必须声明为 Private，并且必须始终是其主体中没有语句的子例程。分部方法本身不能实现接口方法，但提供其主体的方法可以。

只能有一个方法为分部方法提供主体。为分部方法提供主体的方法必须与该分部方法具有相同的签名，对任何类型参数的相同约束、相同的声明修饰符以及相同的参数和类型参数名称。分部方法及提供其主体的方法上的特性将合并在一起，因为这些特性是这两个方法的参数。同样，这两个方法处理的事件列表也将合并在一起。例如：

Class C1  
 Event E1()  
 Event E2()  
  
 Private Partial Sub S() Handles Me.E1  
 End Sub  
  
 ' Handles both E1 and E2  
 Private Sub S() Handles Me.E2  
 ...  
 End Sub  
End Class

## 构造函数

构造函数是用于控制初始化的专用方法。它们在启动程序后或创建类型实例后运行。与其他成员不同的是，构造函数不能被继承，不会在类型的声明空间中引入名称。构造函数只能由对象创建表达式或 .NET Framework 进行调用；它们决不能直接被调用。

注意 构造函数对行放置的限制与子例程的相同。开始和结束语句以及代码块都必须位于逻辑行的开头。

ConstructorMemberDeclaration ::=  
 [ Attributes ] [ ConstructorModifier+ ] Sub New  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Sub StatementTerminator

ConstructorModifier ::= AccessModifier | Shared

### 实例构造函数

实例构造函数用于初始化类型实例，在创建实例时由 .NET Framework 运行。构造函数的参数列表与方法的参数列表遵循相同的规则。实例构造函数可以被重载。

引用类型中的所有构造函数都必须调用另一个构造函数。如果该调用是显式的，则它必须是构造函数方法体中的第一个语句。该语句可以调用类型的另一个实例构造函数（例如，Me.New(...) 或 MyClass.New(...)）；或者，如果该类型不是结构，则可以调用其基类型的实例构造函数（例如，MyBase.New(...)）。构造函数对自身的调用是无效的。如果构造函数省略对另一个构造函数的调用，则隐式使用 MyBase.New()。如果没有无参数基类型构造函数，则会发生编译时错误。由于只有调用基类构造函数后才会将 Me 视为已构造，因此构造函数调用语句的参数不能隐式或显式引用 Me、MyClass 或 MyBase。

如果构造函数的第一个语句采用 MyBase.New(...) 形式，则该构造函数会隐式执行由类型中所声明的实例变量的变量初始值设定项指定的初始化。这对应于一个赋值序列，它们会在调用直接基类型构造函数后立即执行。这种排序确保了在执行任何访问该实例的语句之前，所有基实例变量都已按照它们的变量初始值设定项进行了初始化。例如：

Class A  
 Protected x As Integer = 1  
End Class  
  
Class B  
 Inherits A  
  
 Private y As Integer = x  
  
 Public Sub New()  
 Console.WriteLine("x = " & x & ", y = " & y)  
 End Sub  
End Class

当使用 New B() 创建 B 的实例时，产生如下输出：

x = 1, y = 1

y 的值为 1，因为变量初始值设定项是在调用基类构造函数后执行的。变量初始值设定项按它们出现在类型声明中的文本顺序执行。

如果某类型仅声明 Private 构造函数，则通常不能从该类型派生其他类型或创建该类型的实例；唯的一例外是嵌套在该类型中的类型。Private 构造函数通常用在仅包含 Shared 成员的类型中。

如果类型不包含实例构造函数声明，则会自动提供默认构造函数。默认构造函数只是调用直接基类型的无参数构造函数。如果直接基类型没有可访问的无参数构造函数，则会发生编译时错误。除非默认构造函数的已声明访问类型为 MustInherit（在这种情况下默认构造函数为 Protected），否则该类型为 Public。

批注

MustInherit 类型的默认构造函数的默认访问为 Protected，因为不能直接创建 MustInherit 类。因此，将该默认构造函数设为 Public 是没有意义的。

在下面的示例中，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：

Class Message  
 Dim sender As Object  
 Dim text As String  
End Class

因而，该示例完全等效于下面的示例：

Class Message  
 Dim sender As Object  
 Dim text As String  
  
 Sub New()  
 End Sub  
End Class

如果将默认构造函数发出到标有 Microsoft.VisualBasic.CompilerServices.DesignerGeneratedAttribute 特性的设计器生成类，则它将在调用基构造函数后调用方法 Sub InitializeComponent()（如果该方法存在）。

批注

这允许设计器生成的文件（例如，由 WinForms 设计器创建的那些文件）忽略设计器文件中的构造函数。这使程序员可以自行指定该构造函数（如果他们选择这么做）。

### 共享构造函数

共享构造函数用于初始化类型的共享变量；它们在程序开始执行后但在对类型成员进行任何引用之前创建。共享构造函数指定 Shared 修饰符，除非它位于标准模块中（标准模块中隐含使用 Shared 修饰符）。

与实例构造函数不同，共享构造函数具有隐式公共访问，没有参数并且不能调用其他构造函数。在其第一个语句之前，共享构造函数会隐式执行由类型中所声明的共享变量的变量初始值设定项指定的初始化。这对应于一个赋值序列，它们会在进入该构造函数后立即执行。这些变量初始值设定项按它们出现在类型声明中的文本顺序执行。

下面的示例演示 Employee 类，该类具有一个用于初始化共享变量的共享构造函数：

Imports System.Data  
  
Class Employee  
 Private Shared ds As DataSet  
  
 Shared Sub New()  
 ds = New DataSet()  
 End Sub  
  
 Public Name As String  
 Public Salary As Decimal  
End Class

每个封闭式泛型类型都存在一个单独的共享构造函数。由于每个封闭式类型的共享构造函数仅执行一次，因此它适合对无法在编译时通过约束检查的类型参数执行运行时检查。例如，下面的类型使用共享构造函数检查类型参数是 Integer 还是 Double：

Class EnumHolder(Of T)  
 Shared Sub New()   
 If Not GetType(T).IsEnum() Then  
 Throw New ArgumentException("T must be an enumerated type.")  
 End If  
 End Sub  
End Class

具体在何时运行共享构造函数主要依赖于实现，但如果显式定义了共享构造函数，则会提供多种保证：

共享构造函数在访问类型的任何静态字段之前运行。

共享构造函数在首次调用类型的任何静态方法之前运行。

共享构造函数在首次调用类型的任何构造函数之前运行。

上述保证不适用于为共享初始值设定项隐式创建共享构造函数的情况。下面示例的输出是不确定的，原因是未定义共享构造函数的具体加载排序及相应的执行：

Module Test  
 Sub Main()  
 A.F()  
 B.F()  
 End Sub  
End Module  
  
Class A  
 Shared Sub New()  
 Console.WriteLine("Init A")  
 End Sub  
  
 Public Shared Sub F()  
 Console.WriteLine("A.F")  
 End Sub  
End Class  
  
Class B  
 Shared Sub New()  
 Console.WriteLine("Init B")  
 End Sub  
  
 Public Shared Sub F()  
 Console.WriteLine("B.F")  
 End Sub  
End Class

输出可能为以下任一项：

Init A  
A.F  
Init B  
B.F

或

Init B  
Init A  
A.F  
B.F

相反，下面的示例会产生可预测输出。请注意，类 A 的 Shared 构造函数从不执行，即使类 B 从其派生也是如此：

Module Test  
 Sub Main()  
 B.G()  
 End Sub  
End Module  
  
Class A  
 Shared Sub New()  
 Console.WriteLine("Init A")  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Shared Sub New()  
 Console.WriteLine("Init B")  
 End Sub  
  
 Public Shared Sub G()  
 Console.WriteLine("B.G")  
 End Sub  
End Class

输出为：

Init B  
B.G

也可以构造循环依赖关系，从而允许带有变量初始值设定项的 Shared 变量处于其默认值状态，如下面的示例所示：

Class A  
 Public Shared X As Integer = B.Y + 1  
End Class  
  
Class B  
 Public Shared Y As Integer = A.X + 1  
  
 Shared Sub Main()  
 Console.WriteLine("X = " & A.X & ", Y = " & B.Y)  
 End Sub  
End Class

这会产生如下输出：

X = 1, Y = 2

为了执行 Main 方法，系统首先加载类 B。接着，类 B 的 Shared 构造函数计算 Y 的初始值，这会因引用 A.X 的值而导致递归加载类 A。类 A 的 Shared 构造函数进而计算 X 的初始值，并在执行此操作期间获取 Y 的默认值零。因而，A.X 会初始化为 1。随后会完成加载 A 的过程，从而重新计算 Y 的初始值，结果将变为 2。

在类 A 中加载 Main 方法后，该示例会产生如下输出：

X = 2, Y = 1

请避免在 Shared 变量初始值设定项中使用循环引用，因为通常无法确定加载包含这些引用的类的顺序。

## 事件

事件用于通知特定匹配项的代码。事件声明由一个标识符、一个委托类型或参数列表以及可选的 Implements 子句组成。如果指定了委托类型，则该委托类型不能有返回类型。如果指定了参数列表，则该列表不能包含 Optional 或 ParamArray 参数。参数类型和/或委托类型的可访问域必须与事件本身的可访问域相同或者是其超集。事件可通过指定 Shared 修饰符进行共享。

除了添加到类型的声明空间的成员名称之外，事件声明还需要隐式声明其他几个成员。如果指定一个名为 X 的事件，则需要将下列成员添加到声明空间中：

如果声明形式为方法声明，则会引入一个名为 XEventHandler 的嵌套委托类。嵌套的委托类与方法声明相匹配，并与事件具有相同的可访问性。参数列表中的特性应用于委托类的参数。

类型化为委托的 Private 实例变量，名为 XEvent。

名为 add\_X 和 remove\_X 的两个方法，它们不能被调用、重写或重载。

如果类型尝试声明与上述任一名称相匹配的名称，则会导致发生编译时错误，因而出于名称绑定目的将忽略隐式 add\_X 和 remove\_X 声明。不能重写或重载任何引入的成员，但可以在派生类型中隐藏它们。例如，以下类声明

Class Raiser  
 Public Event Constructed(i As Integer)  
End Class

与以下声明等效

Class Raiser  
 Public Delegate Sub ConstructedEventHandler(i As Integer)  
  
 Protected ConstructedEvent As ConstructedEventHandler  
  
 Public Sub add\_Constructed(d As ConstructedEventHandler)  
 ConstructedEvent = \_  
 CType( \_  
 [Delegate].Combine(ConstructedEvent, d), \_  
 Raiser.ConstructedEventHandler)  
 End Sub  
  
 Public Sub remove\_Constructed(d As ConstructedEventHandler)  
 ConstructedEvent = \_  
 CType( \_  
 [Delegate].Remove(ConstructedEvent, d), \_  
 Raiser.ConstructedEventHandler)  
 End Sub  
End Class

在不指定委托类型的情况下声明事件是最简单且最紧凑的语法，但缺点是需要为每个事件声明新的委托类型。例如，在下面的示例中，将创建三个隐藏的委托类型，即使所有这三个事件具有相同的参数列表也是如此：

Public Class Button  
 Public Event Click(sender As Object, e As EventArgs)  
 Public Event DoubleClick(sender As Object, e As EventArgs)  
 Public Event RightClick(sender As Object, e As EventArgs)  
End Class

在下面的示例中，多个事件只使用同一委托 EventHandler：

Public Delegate Sub EventHandler(sender As Object, e As EventArgs)  
  
Public Class Button  
 Public Event Click As EventHandler  
 Public Event DoubleClick As EventHandler  
 Public Event RightClick As EventHandler  
End Class

只能通过下列两种方式之一处理事件：静态方式或动态方式。以静态方式处理事件更简单，只需 WithEvents 变量和 Handles 子句。在下面的示例中，类 Form1 以静态方式处理对象 Button 的事件 Click：

Public Class Form1  
 Public WithEvents Button1 As New Button()  
  
 Public Sub Button1\_Click(sender As Object, e As EventArgs) \_  
 Handles Button1.Click  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
End Class

以动态方式处理事件是比较复杂，因为必须用代码显式连接和断开连接事件。语句 AddHandler 为事件添加处理程序，而语句 RemoveHandler 移除事件的处理程序。下一示例演示 Form1 类，该类将 Button1\_Click 添加为 Button1 的 Click 事件的事件处理程序：

Public Class Form1  
 Public Sub New()  
 ' Add Button1\_Click as an event handler for Button1's Click event.  
 AddHandler Button1.Click, AddressOf Button1\_Click  
 End Sub   
  
 Private Button1 As Button = New Button()  
  
 Sub Button1\_Click(sender As Object, e As EventArgs)  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
  
 Public Sub Disconnect()  
 RemoveHandler Button1.Click, AddressOf Button1\_Click  
 End Sub   
End Class

在方法 Disconnect 中将移除该事件处理程序。

EventMemberDeclaration ::=  
 RegularEventMemberDeclaration |  
 CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType [ ImplementsClause ]  
 StatementTerminator

InterfaceEventMemberDeclaration ::=  
 [ Attributes ] [ InterfaceEventModifiers+ ] Event Identifier ParametersOrType StatementTerminator

ParametersOrType ::=  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] |  
 As NonArrayTypeName

EventModifiers ::= AccessModifier | Shadows | Shared

InterfaceEventModifiers ::= Shadows

### 自定义事件

如上一节所述，事件声明隐式定义一个字段、一个 add\_ 方法和一个 remove\_ 方法，用于跟踪事件处理程序。但在某些情况下，可能需要提供用于跟踪事件处理程序的自定义代码。例如，如果某一个类定义了四十个事件，而在任何时候将只处理其中几个事件，则使用哈希表而不是四十个字段来跟踪每个事件的事务处理程序可能会更加有效。使用自定义事件可以显式定义 add\_X 和 remove\_X 方法，这可为事件处理程序启用自定义存储。

声明自定义事件的方式与声明指定委托类型的事件的方式基本相同，只是前一种方式中必须在 Event 关键字之前添加关键字 Custom。自定义事件声明包含三个声明：AddHandler 声明、RemoveHandler 声明和 RaiseEvent 声明。其中任何声明都不能具有任何修饰符，但它们可以具有特性。例如：

Class Test  
 Private Handlers As EventHandler  
  
 Public Custom Event TestEvent As EventHandler  
 AddHandler(value As EventHandler)  
 Handlers = CType([Delegate].Combine(Handlers, value), \_  
 EventHandler)  
 End AddHandler  
  
 RemoveHandler(value as EventHandler)  
 Handlers = CType([Delegate].Remove(Handlers, value), \_  
 EventHandler)  
 End RemoveHandler  
  
 RaiseEvent(sender As Object, e As EventArgs)  
 Dim TempHandlers As EventHandler = Handlers  
  
 If TempHandlers IsNot Nothing Then  
 TempHandlers(sender, e)  
 End If  
 End RaiseEvent  
 End Event  
End Class

AddHandler 和 RemoveHandler 声明采用一个 ByVal 形参，该形参必须属于事件的委托类型。在执行 AddHandler 或 RemoveHandler 语句（或 Handles 子句自动处理事件）时，将调用相应的声明。RaiseEvent 声明与事件委托采用相同的参数，该声明在执行 RaiseEvent 语句时调用。必须提供所有这些声明，并将其视为子例程。

注意 AddHandler、RemoveHandler 和 RaiseEvent 声明对行放置的限制与子例程相同。开始和结束语句以及代码块都必须位于逻辑行的开头。

批注

在 Microsoft Visual Basic 11.0 中，对于在使用 /target:winmdobj 编译的文件中声明的事件，或先在这样的文件中的接口中声明，再在其他地方实现的事件，其处理方式略有不同。

1.用于生成 winmd 的外部工具通常只允许某些委托类型，如 System.EventHandler(Of T) 或 System.TypedEventHandle(Of T, U)，而不允许其他类型。

2.XEvent 字段的类型为 System.Runtime.InteropServices.WindowsRuntime.EventRegistrationTokenTable(Of T)，其中 T 是委托类型。

3.AddHandler 访问器将返回 System.Runtime.InteropServices.WindowsRuntime.EventRegistrationToken，RemoveHandler 访问器则使用类型相同的单个形参。

下面是此类自定义事件的一个示例。

Imports System.Runtime.InteropServices.WindowsRuntime

Public NotInheritable Class ClassInWinMD  
 Private XEvent As EventRegistrationTokenTable(Of EventHandler(Of Integer))

Public Custom Event X As EventHandler(Of Integer)  
 AddHandler(handler As EventHandler(Of Integer))  
 Return EventRegistrationTokenTable(Of EventHandler(Of Integer)).  
 GetOrCreateEventRegistrationTokenTable(XEvent).  
 AddEventHandler(handler)  
 End AddHandler

RemoveHandler(token As EventRegistrationToken)  
 EventRegistrationTokenTable(Of EventHandler(Of Integer)).  
 GetOrCreateEventRegistrationTokenTable(XEvent).  
 RemoveEventHandler(token)  
 End RemoveHandler

RaiseEvent(sender As Object, i As Integer)  
 Dim table = EventRegistrationTokenTable(Of EventHandler(Of Integer)).  
 GetOrCreateEventRegistrationTokenTable(XEvent).  
 InvocationList  
 If table IsNot Nothing Then table(sender, i)  
 End RaiseEvent  
 End Event  
End Class除了添加到类型的声明空间的成员名称之外，自定义事件声明还需要隐式声明其他几个成员。如果指定一个名为 X 的事件，则需要将下列成员添加到声明空间中：

名为 add\_X 的方法，与 AddHandler 声明相对应。

名为 remove\_X 的方法，与 RemoveHandler 声明相对应。

名为 fire\_X 的方法，与 RaiseEvent 声明相对应。

如果类型尝试声明与上述任一名称相匹配的名称，则会导致发生编译时错误，因而出于名称绑定目的将全部忽略隐式声明。不能重写或重载任何引入的成员，但可以在派生类型中隐藏它们。

注意 Custom 不是保留字。

CustomEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Custom Event Identifier As TypeName [ ImplementsClause ]  
 StatementTerminator  
 EventAccessorDeclaration+  
 End Event StatementTerminator

EventAccessorDeclaration ::=  
 AddHandlerDeclaration |  
 RemoveHandlerDeclaration |  
 RaiseEventDeclaration

AddHandlerDeclaration ::=  
 [ Attributes ] AddHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End AddHandler StatementTerminator

RemoveHandlerDeclaration ::=  
 [ Attributes ] RemoveHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RemoveHandler StatementTerminator

RaiseEventDeclaration ::=  
 [ Attributes ] RaiseEvent OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RaiseEvent StatementTerminator

## 常量

常量是作为类型成员的固定值。常量是隐式共享的。如果声明包含 As 子句，则该子句指定该声明所引入的成员的类型。如果省略类型，则会推断出常量类型。常量的类型只能为基元类型或 Object。如果常量类型化为 Object 并且没有类型字符，则常量的实际类型将为常量表达式的类型。否则，常量的类型为该常量的类型字符的类型。

下面的示例演示一个名为 Constants 的类，该类有两个公共常量：

Class Constants  
 Public Const A As Integer = 1  
 Public Const B As Integer = A + 1  
End Class

常量可通过类进行访问，如下面的示例所示，该示例输出 Constants.A 和 Constants.B 的值。

Module Test  
 Sub Main()  
 Console.WriteLine(Constants.A & ", " & Constants.B)  
 End Sub   
End Module

声明多个常量的常量声明等效于单个常量的多个声明。下面的示例在一个声明语句中声明三个常量。

Class A  
 Protected Const x As Integer = 1, y As Long = 2, z As Short = 3  
End Class

此声明与下列声明等效：

Class A  
 Protected Const x As Integer = 1  
 Protected Const y As Long = 2  
 Protected Const z As Short = 3  
End Class

常量类型的可访问域必须与常量本身的可访问域相同或者是其超集。常数表达式必须产生常量类型的值或可隐式转换为常量类型的类型的值。常量表达式不能是循环的；也就是说，不能用常量自身来定义常量。

编译器会自动按适当的顺序来计算各个常量声明。在下面的示例中，编译器依次计算 Y、Z 和 X，并分别生成值 10、11 和 12。

Class A  
 Public Const X As Integer = B.Z + 1  
 Public Const Y As Integer = 10  
End Class  
  
Class B  
 Public Const Z As Integer = A.Y + 1  
End Class

如果需要使用常量值的符号名称，但在常量声明中不允许该值类型，或者如果常量表达式无法在编译时计算该值，则可能会改用只读变量。

ConstantMemberDeclaration ::=  
 [ Attributes ] [ ConstantModifier+ ] Const ConstantDeclarators StatementTerminator

ConstantModifier ::= AccessModifier | Shadows

ConstantDeclarators ::=  
 ConstantDeclarator |  
 ConstantDeclarators Comma ConstantDeclarator

ConstantDeclarator ::= Identifier [ As TypeName ] Equals ConstantExpression StatementTerminator

## 实例和共享变量

实例或共享变量是可存储信息的类型的成员。如果未指定任何修饰符，则必须指定 Dim 修饰符，但在其他情况下可以省略该修饰符。单个变量声明可包含多个变量声明符；每个变量声明符都引入了一个新的实例或共享成员。

如果指定了初始值设定项，则变量声明符只能声明一个实例或共享变量：

Class Test  
 Dim a, b, c, d As Integer = 10 ' Invalid: multiple initialization  
End Class

此限制不适用于对象初始值设定项：

Class Test  
 Dim a, b, c, d As New Collection() ' OK  
End Class

用 Shared 修饰符声明的变量为共享变量。无论创建多少个类型实例，共享变量都只标识一个存储位置。共享变量在程序开始执行时开始存在，在程序终止时不复存在。

共享变量仅在特定的封闭式泛型类型实例之间共享。例如，程序：

Class C(Of V)   
 Shared InstanceCount As Integer = 0  
  
 Public Sub New()   
 InstanceCount += 1   
 End Sub  
   
 Public Shared ReadOnly Property Count() As Integer   
 Get  
 Return InstanceCount  
 End Get  
 End Property  
End Class  
  
Class Application   
 Shared Sub Main()   
 Dim x1 As New C(Of Integer)()  
 Console.WriteLine(C(Of Integer).Count)  
  
 Dim x2 As New C(Of Double)()   
 Console.WriteLine(C(Of Integer).Count)  
  
 Dim x3 As New C(Of Integer)()   
 Console.WriteLine(C(Of Integer).Count)  
 End Sub  
End Class

输出如下结果：

1  
1  
2

未用 Shared 修饰符声明的变量称为实例变量。类的每个实例都为该类的所有实例变量包含一个单独副本。引用类型的实例变量在创建该类型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 Finalize 方法时不复存在。值类型的实例变量与它所属的变量具有完全相同的生存期。换言之，当值类型的变量开始存在或不复存在时，该值类型的实例变量也随之存在或消失。

如果声明符包含 As 子句，则该子句会指定该声明所引入的成员的类型。如果省略该类型并且使用的是严格语义，则会发生编译时错误。否则，成员的类型是隐式 Object 或成员的类型字符的类型。

注意   该语法不存在歧义：如果声明符省略类型，则它会始终使用以下声明符的类型。

实例或共享变量类型或数组元素类型的可访问域必须与该实例或共享变量本身的可访问域相同或者是其超集。

下面的示例演示 Color 类，该类具有分别名为 redPart、greenPart 和 bluePart 的内部实例变量：

Class Color  
 Friend redPart As Short  
 Friend bluePart As Short  
 Friend greenPart As Short  
  
 Public Sub New(red As Short, blue As Short, green As Short)  
 redPart = red  
 bluePart = blue  
 greenPart = green  
 End Sub  
End Class

VariableMemberDeclaration ::=  
 [ Attributes ] VariableModifier+ VariableDeclarators StatementTerminator

VariableModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 ReadOnly |  
 WithEvents |  
 Dim

VariableDeclarators ::=  
 VariableDeclarator |  
 VariableDeclarators Comma VariableDeclarator

VariableDeclarator ::=  
 VariableIdentifiers As ObjectCreationExpression |  
 VariableIdentifiers [ As TypeName ] [ Equals Expression ]

VariableIdentifiers ::=  
 VariableIdentifier |  
 VariableIdentifiers Comma VariableIdentifier

VariableIdentifier ::= Identifier IdentifierModifiers

### 只读变量

当实例或共享变量声明包含 ReadOnly 修饰符时，该声明引入的变量赋值只能作为声明的一部分出现，或者出现在同一个类的构造函数中。具体来说，在下列情况下只允许只读实例或共享变量的赋值：

在引入实例或共享变量（通过在声明中包含变量初始值设定项）的变量声明中。

对于实例变量，在包含变量声明的类的实例构造函数中。只能以非限定方式或通过 Me 或 MyClass 访问该实例变量。

对于共享变量，在包含共享变量声明的类的共享构造函数中。

如果需要使用常量值的符号名称，但在常量声明中不允许该值类型，或者如果常量表达式无法在编译时计算该值，则共享只读变量十分有用。

下面是第一个此类应用程序的示例，其中颜色共享变量声明为 ReadOnly 以防被其他程序更改：

Class Color  
 Friend redPart As Short  
 Friend bluePart As Short  
 Friend greenPart As Short  
  
 Public Sub New(red As Short, blue As Short, green As Short)  
 redPart = red  
 bluePart = blue  
 greenPart = green  
 End Sub   
  
 Public Shared ReadOnly Red As Color = New Color(&HFF, 0, 0)  
 Public Shared ReadOnly Blue As Color = New Color(0, &HFF, 0)  
 Public Shared ReadOnly Green As Color = New Color(0, 0, &HFF)  
 Public Shared ReadOnly White As Color = New Color(&HFF, &HFF, &HFF)  
End Class

常量和只读共享变量具有不同的语义。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共享变量时，要等到运行时才获取该共享变量的值。请考虑以下包含两个单独程序的应用程序。

file1.vb:

Namespace Program1  
 Public Class Utils  
 Public Shared ReadOnly X As Integer = 1  
 End Class  
End Namespace

file2.vb:

Namespace Program2  
 Module Test  
 Sub Main()  
 Console.WriteLine(Program1.Utils.X)  
 End Sub  
 End Module  
End Namespace

Program1 和 Program2 命名空间表示两个单独编译的程序。由于变量 Program1.Utils.X 声明为 Shared ReadOnly，因此由 Console.WriteLine 语句输出的值在编译时是未知的，而是在运行时获取。因此，如果更改了 X 的值并重新编译 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，则该值不会受到 Program1 中更改的影响，除非重新编译 Program2。

### WithEvents 变量

通过使用 WithEvents 修饰符声明引发某些事件的某个实例或共享变量，类型可以声明处理由该实例或共享变量引发的这些事件。例如：

Class Raiser  
 Public Event E1()  
  
 Public Sub Raise()  
 RaiseEvent E1  
 End Sub  
End Class  
  
Module Test  
 Private WithEvents x As Raiser  
  
 Private Sub E1Handler() Handles x.E1  
 Console.WriteLine("Raised")  
 End Sub  
  
 Public Sub Main()  
 x = New Raiser()  
 End Sub  
End Module

在此示例中，方法 E1Handler 用于处理事件 E1，该事件由存储在实例变量 x 中的类型 Raiser 的实例引发。

WithEvents 修饰符会导致用前导下划线重命名该变量，并用执行事件挂钩的同名属性替换该变量。例如，如果该变量的名称为 F，则会将其重命名为 \_F 并隐式声明属性 F。如果该变量的新名称与另一个声明之间存在冲突，则会报告编译时错误。应用于该变量的任何特性都会转给重命名的变量。

由 WithEvents 声明创建的隐式属性负责挂钩和解除挂钩相关事件处理程序。在为该变量赋值后，该属性首先在当前位于该变量中的实例上调用事件的 remove 方法（如果存在现有事件处理程序，则对其解除挂钩）。接下来在执行赋值后，该属性在变量中的新实例上调用事件的 add 方法（挂钩新的事件处理程序）。下面的代码与上述标准模块 Test 的代码等效：

Module Test  
 Private \_x As Raiser  
  
 Public Property x() As Raiser  
 Get  
 Return \_x  
 End Get  
  
 Set (Value As Raiser)  
 ' Unhook any existing handlers.  
 If \_x IsNot Nothing Then  
 RemoveHandler \_x.E1, AddressOf E1Handler  
 End If  
  
 ' Change value.  
 \_x = Value  
  
 ' Hook-up new handlers.  
 If \_x IsNot Nothing Then  
 AddHandler \_x.E1, AddressOf E1Handler  
 End If  
 End Set  
 End Property  
  
 Sub E1Handler()  
 Console.WriteLine("Raised")  
 End Sub  
  
 Sub Main()  
 x = New Raiser()  
 End Sub  
End Module

如果实例或共享变量已类型化为结构，则将该变量声明为 WithEvents 是无效的。此外，也不能在结构中指定 WithEvents，并且不能组合使用 WithEvents 和 ReadOnly。

### 变量初始值设定项

类中的实例和共享变量声明以及结构中的实例变量声明（而不是共享变量声明）可包括变量初始值设定项。对于 Shared 变量，变量初始值设定项相当于启动程序之后但首次引用 Shared 变量之前执行的赋值语句。对于实例变量，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。结构不能具有实例变量初始值设定项，因为其无参数构造函数无法修改。

请看下面的示例：

Class Test  
 Public Shared x As Double = Math.Sqrt(2.0)  
 Public i As Integer = 100  
 Public s As String = "Hello"  
End Class  
  
Module TestModule  
 Sub Main()  
 Dim a As New Test()  
  
 Console.WriteLine("x = " & Test.x & ", i = " & a.i & ", s = " & a.s)  
 End Sub  
End Module

该示例产生如下输出：

x = 1.4142135623731, i = 100, s = Hello

在加载类时会执行对 x 的赋值，而在创建类的新实例时会执行对 i 和 s 的赋值。

将变量初始值设定项视为自动插入类型的构造函数块的赋值语句是十分有用的。下面的示例包含几个实例变量初始值设定项。

Class A  
 Private x As Integer = 1  
 Private y As Integer = -1  
 Private count As Integer  
  
 Public Sub New()  
 count = 0  
 End Sub  
  
 Public Sub New(n As Integer)  
 count = n  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Private sqrt2 As Double = Math.Sqrt(2.0)  
 Private items As ArrayList = New ArrayList(100)  
 Private max As Integer  
  
 Public Sub New()  
 Me.New(100)  
 items.Add("default")  
 End Sub  
  
 Public Sub New(n As Integer)  
 MyBase.New(n - 1)  
 max = n  
 End Sub  
End Class

该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。

Class A  
 Private x, y, count As Integer  
  
 Public Sub New()  
 MyBase.New ' Invoke object() constructor.  
 x = 1 ' This is a variable initializer.  
 y = -1 ' This is a variable initializer.  
 count = 0  
 End Sub  
  
 Public Sub New(n As Integer)  
 MyBase.New ' Invoke object() constructor.   
 x = 1 ' This is a variable initializer.  
 y = - 1 ' This is a variable initializer.  
 count = n  
 End Sub  
End Class  
  
Class B  
 Inherits A  
  
 Private sqrt2 As Double  
 Private items As ArrayList  
 Private max As Integer  
  
 Public Sub New()  
 Me.New(100)   
 items.Add("default")  
 End Sub  
  
 Public Sub New(n As Integer)  
 MyBase.New(n - 1)   
 sqrt2 = Math.Sqrt(2.0) ' This is a variable initializer.  
 items = New ArrayList(100) ' This is a variable initializer.  
 max = n  
 End Sub  
End Class

在执行任何变量初始值设定项之前，所有变量都会初始化为其类型的默认值。例如：

Class Test  
 Public Shared b As Boolean  
 Public i As Integer  
End Class  
  
Module TestModule  
 Sub Main()  
 Dim t As New Test()  
 Console.WriteLine("b = " & Test.b & ", i = " & t.i)  
 End Sub  
End Module

由于在加载类会自动将 b 初始化为其默认值，而在创建类的实例时也会自动将 i 初始化为其默认值，因此上述代码会产生以下输出：

b = False, i = 0

每个变量初始值设定项必须产生该常量类型的值或可隐式转换为该变量类型的类型的值。变量初始值设定项可以是循环的，也可以引用将在其后进行初始化的变量，在后一情况下，被引用变量的值是其默认值以便使用该初始值设定项。这种初始值设定项的值是不确定的。

变量初始值设定项有三种形式：常规初始值设定项、数组大小初始值设定项和对象初始值设定项。前一种形式显示在等号（跟在类型名称后面）之后，后两种形式是声明自身的一部分。在任何特定声明中，只能使用一种初始值设定项形式。

#### 常规初始值设定项

常规初始值设定项是一个可隐式转换为变量类型的表达式。它显示在等号（跟在类型名称后面）之后，并且必须归类为一个值。例如：

Module Test  
 Dim x As Integer = 10  
 Dim y As Integer = 20  
  
 Sub Main()  
 Console.WriteLine("x = " & x & ", y = " & y)  
 End Sub  
End Module

此程序产生如下输出：

x = 10, y = 20

如果变量声明具有初始值设定项，则一次只能声明一个变量。例如：

Module Test  
 Sub Main()  
 ' OK, only one variable declared at a time.  
 Dim x As Integer = 10, y As Integer = 20  
  
 ' Error: Can’t initialize multiple variables at once.  
 Dim a, b As Integer = 10  
 End Sub  
End Module

#### 对象初始值设定项

对象初始值设定项是使用对象创建表达式（替代了类型名称）指定的。对象初始值设定项相当于将对象创建表达式的结果赋予变量的常规初始值设定项。因此

Module TestModule  
 Sub Main()  
 Dim x As New Test(10)  
 End Sub  
End Module

相当于

Module TestModule  
 Sub Main()  
 Dim x As Test = New Test(10)  
 End Sub  
End Module

对象初始值设定项中的括号始终解释为构造函数的参数列表，而决不会解释为数组类型修饰符。具有对象初始值设定项的变量名称不能具有数组类型修饰符或可以为 null 的类型修饰符。

#### 数组大小初始值设定项

数组大小初始值设定项是变量名称上的修饰符，提供一组由表达式指示的维度上限。这些上限表达式必须归类为值，并且必须能够隐式转换为 Integer。上限绑定集相当于指定了上限的数组创建表达式的变量初始值设定项。从数组大小初始值设定项推断出数组类型的维度的数目。因此

Module Test  
 Sub Main()  
 Dim x(5, 10) As Integer  
 End Sub  
End Module

相当于

Module Test  
 Sub Main()  
 Dim x As Integer(,) = New Integer(5, 10) {}  
 End Sub  
End Module

所有上限都必须等于或大小 -1，并且所有维度都必须具有指定上限。如果被初始化的数组的元素类型本身是数组类型，则数组类型的修饰符会转至数组大小初始值设定项的右侧。例如

Module Test  
 Sub Main()  
 Dim x(5,10)(,,) As Integer  
 End Sub  
End Module

声明一个局部变量 x，其类型为 Integer 类型的三维数组的二维数组，在第一个维度中初始化为界限为 0..5 的数组，在第二个维度中初始化为界限为 0..10 的数组。不能使用数组大小初始值设定项初始化其类型为数组的变量的元素。

包含数组大小初始值设定项的变量声明不能在其类型或常规初始值设定项上包含数组类型修饰符。

ArraySizeInitializationModifier ::=  
 OpenParenthesis BoundList CloseParenthesis [ ArrayTypeModifiers ]

BoundList::=  
 Bound |  
 BoundList Comma Bound

Bound ::=  
 Expression |  
 0 To Expression

### System.MarshalByRefObject 类

从类 System.MarshalByRefObject 派生的类使用代理（即通过引用）而不是通过复制（即通过值）跨上下文边界进行封送处理。这意味着这种类的实例可能不是真正的实例，而可能只是跨上下文边界封送变量访问和方法调用的存根。

因此，不能创建对在这种类上定义的变量的存储位置的引用。这意味着，如果变量类型化为从 System.MarshalByRefObject 给引用参数，并且不能访问类型化为值类型的变量的方法和变量。相反，Visual Basic 会将在这种类上定义的变量视为属性（因为这些属性与属性具有相同的限制）。

此规则有一个例外：用 Me 隐式或显式限定的成员不存在上述限制，因为 Me 可始终保证该成员为实际对象，而不是代理。

## 属性

属性是变量的自然扩展；这两者都是具有关联类型的命名成员，而且访问变量和属性的语法是相同的。然而，与变量不同，属性不表示存储位置。相反，属性有访问器，这些访问器指定在读取或写入属性的值时要执行的语句。

属性是用属性声明定义的。属性的第一部分与字段声明类似。第二部分包含 Get 访问器和/或 Set 访问器。在下面的示例中，Button 类定义了 Caption 属性。

Public Class Button  
 Private captionValue As String  
  
 Public Property Caption() As String  
 Get  
 Return captionValue  
 End Get  
  
 Set (Value As String)  
 captionValue = value  
 Repaint()  
 End Set  
 End Property  
  
 ...  
End Class

下面的示例根据上述 Button 类演示如何使用 Caption 属性：

Dim okButton As Button = New Button()  
  
okButton.Caption = "OK" ' Invokes Set accessor.  
Dim s As String = 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。

如果没有为属性指定类型并且使用的是严格语义，则会发生编译时错误；否则，属性的类型是隐式 Object 或是属性的类型字符的类型。属性声明可以包含 Get 访问器和/或 Set 访问器；前者用于检索属性值，后者用于存储属性值。由于属性隐式声明方法，因此可以用与方法相同的修饰符来声明属性。如果在接口中定义属性或使用 MustOverride 修饰符定义属性，则必须省略属性体和 End 构造，否则会发生编译时错误。

索引参数列表构成了属性的签名，因此可以在索引参数（而不是在属性类型）上重载属性。索引参数列表与常规方法的相同。不过，其中任何参数都不能用 ByRef 修饰符修饰，也不能命名为 Value（这是 Set 访问器中为隐式值参数保留的名称）。

可按如下所示声明属性：

如果属性未指定属性类型修饰符，则该属性必须同时具有 Get 访问器和 Set 访问器。该属性被视为读写属性。

如果属性指定 ReadOnly 修饰符，则该属性必须具有 Get 访问器，但不能具有 Set 访问器。该属性被视为只读属性。将只读属性作为赋值目标会导致编译时错误。

如果属性指定 WriteOnly 修饰符，则该属性必须具有 Set 访问器，但不能具有 Get 访问器。该属性被视为只写属性。除了作为赋值的目标或方法的参数之外，在表达式中引用只写属性会导致编译时错误。

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。下面的示例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：

Class A  
 Private nameValue As String  
  
 ' Error, contains a duplicate member name.  
 Public ReadOnly Property Name() As String   
 Get  
 Return nameValue  
 End Get  
 End Property  
  
 ' Error, contains a duplicate member name.  
 Public WriteOnly Property Name() As String   
 Set (Value As String)  
 nameValue = value  
 End Set  
 End Property  
End Class

由于在同一个类中不能声明两个同名成员，此示例将导致编译时错误。

默认情况下，属性的 Get 和 Set 访问器的可访问性与属性本身的可访问性相同。不过，Get 和 Set 访问器也可以指定与属性不同的可访问性。在该情况下，访问器的可访问性要比属性的可访问性受到更多的限制，并且只能有一个访问器具有与属性不同的可访问性级别。访问类型被视为受到或多或少的限制，如下所示：

Private 的限制性高于 Public、Protected Friend、Protected 或 Friend。

Friend 的限制性高于 Protected Friend 或 Public。

Protected 的限制性高于 Protected Friend 或 Public。

Protected Friend 的限制性高于 Public。

如果属性的一个访问器可访问，而另一个访问器不可访问，则该属性将被视为只读或只写。例如：

Class A  
 Public Property P() As Integer  
 Get  
 ...  
 End Get  
  
 Private Set (Value As Integer)  
 ...  
 End Set  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim a As A = New A()  
  
 ' Error: A.P is read-only in this context.  
 a.P = 10  
 End Sub  
End Module

当派生类型隐藏着某一属性时，该派生属性会隐藏与读写有关的隐藏属性。在下面的示例中，B 中的 P 属性隐藏 A 中 P 属性的读与写：

Class A  
 Public WriteOnly Property P() As Integer  
 Set (Value As Integer)  
 End Set  
 End Property  
End Class  
  
Class B  
 Inherits A  
  
 Public Shadows ReadOnly Property P() As Integer  
 Get  
 End Get  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As B = New B  
  
 B.P = 10 ' Error, B.P is read-only.  
 End Sub  
End Module

返回类型或参数类型的可访问域必须与属性本身的可访问域相同或者是其超集。属性只能有一个 Set 访问器和一个 Get 访问器。

除了在声明和调用语法方面存在差异之外，Overridable、NotOverridable、Overrides、MustOverride 和 MustInherit 属性与 Overridable、NotOverridable、Overrides、MustOverride 和 MustInherit 方法具有完全相同的行为。在重写属性时，被重写的属性必须属于同一类型（读写、只读、只写）。Overridable 属性不能包含 Private 访问器。

在下面的示例中，X 是 Overridable 只读属性，Y 是 Overridable 读写属性，Z 是 MustOverride 读写属性。

MustInherit Class A  
 Private \_y As Integer  
  
 Public Overridable ReadOnly Property X() As Integer  
 Get  
 Return 0  
 End Get  
 End Property  
  
 Public Overridable Property Y() As Integer  
 Get  
 Return \_y  
 End Get  
 Set (Value As Integer)  
 \_y = value  
 End Set  
 End Property  
  
 Public MustOverride Property Z() As Integer  
End Class

由于 Z 是 MustOverride，因此包含类 A 必须声明为 MustInherit。

相比之下，从类 A 派生的类如下所示：

Class B  
 Inherits A  
  
 Private \_z As Integer  
  
 Public Overrides ReadOnly Property X() As Integer  
 Get  
 Return MyBase.X + 1  
 End Get  
 End Property  
  
 Public Overrides Property Y() As Integer  
 Get  
 Return MyBase.Y  
 End Get  
 Set (Value As Integer)  
 If value < 0 Then  
 MyBase.Y = 0  
 Else  
 MyBase.Y = Value  
 End If  
 End Set  
 End Property  
  
 Public Overrides Property Z() As Integer  
 Get  
 Return \_z  
 End Get  
 Set (Value As Integer)  
 \_z = Value  
 End Set  
 End Property  
End Class

此处，属性 X、Y 和 Z 的声明重写基属性。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。属性 X 的 Get 访问器和属性 Y 的 Set 访问器使用 MyBase 关键字访问继承的属性。属性 Z 的声明重写 MustOverride 属性，因此类 B 中没有未处理的 MustOverride 成员，并且允许 B 作为常规类。

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：

Imports System.IO  
  
Public Class ConsoleStreams  
 Private Shared reader As TextReader  
 Private Shared writer As TextWriter  
 Private Shared errors As TextWriter  
  
 Public Shared ReadOnly Property [In]() As TextReader  
 Get  
 If reader Is Nothing Then  
 reader = Console.In  
 End If  
 Return reader  
 End Get  
 End Property  
  
 Public Shared ReadOnly Property Out() As TextWriter  
 Get  
 If writer Is Nothing Then  
 writer = Console.Out  
 End If  
 Return writer  
 End Get  
 End Property  
  
 Public Shared ReadOnly Property [Error]() As TextWriter  
 Get  
 If errors Is Nothing Then  
 errors = Console.Error  
 End If  
 Return errors  
 End Get  
 End Property  
End Class

ConsoleStreams 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、输出和错误信息报告。通过将这些成员作为属性公开，ConsoleStreams 类可以将它们的初始化延迟到它们被实际使用时。例如，在 ConsoleStreams.Out.WriteLine("hello, world") 中首次引用 Out 属性时，将初始化输出设备的基础 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。

PropertyMemberDeclaration ::=  
 RegularPropertyMemberDeclaration |  
 MustOverridePropertyMemberDeclaration |  
 AutoPropertyMemberDeclaration

PropertySignature ::=  
 Property Identifier [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ]

RegularPropertyMemberDeclaration ::=  
 [ Attributes ] [ PropertyModifier+ ] PropertySignature [ ImplementsClause ] LineTerminator  
 PropertyAccessorDeclaration+  
 End Property StatementTerminator

MustOverridePropertyMemberDeclaration ::=  
 [ Attributes ] MustOverridePropertyModifier+ PropertySignature [ ImplementsClause ]  
 StatementTerminator

AutoPropertyMemberDeclaration ::=  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ] [ Equals Expression ] [ ImplementsClause ] LineTerminator |  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 As [ Attributes ] New [ NonArrayTypeName  
 [ OpenParenthesis [ ArgumentList ] CloseParenthesis ] ] [ ObjectCreationExpressionInitializer ]  
 [ ImplementsClause ] LineTerminator

InterfacePropertyMemberDeclaration ::=  
 [ Attributes ] [ InterfacePropertyModifier+ ] PropertySignature StatementTerminator

AutoPropertyModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads

PropertyModifier ::=  
 AutoPropertyModifier |  
 Default |  
 ReadOnly |  
 WriteOnly |  
 Iterator

MustOverridePropertyModifier ::= PropertyModifier | MustOverride

InterfacePropertyModifier ::=  
 Shadows |  
 Overloads |  
 Default |  
 ReadOnly |  
 WriteOnly

PropertyAccessorDeclaration ::= PropertyGetDeclaration | PropertySetDeclaration

### Get 访问器声明

Get 访问器 (getter) 通过使用属性 Get 声明进行声明。属性 Get 声明由关键字 Get 后跟语句块组成。如果有一个名为 P 的属性，则 Get 访问器声明会用与该属性相同的修饰符、类型和参数列表隐式声明一个名为 get\_P 的方法。如果类型包含具有该名称的声明，则会导致发生编译时错误，但出于名称绑定目的而忽略隐式声明。

在 Get 访问器体的声明空间中将隐式声明一个与该属性同名的特殊局部变量，该变量表示该属性的返回值。该局部变量在用于表达式时会具有特殊的名称解析语义。如果在一个上下文（此上下文应为一个归类为方法组的表达式，如调用表达式）中使用该局部变量，则名称将解析为函数而非局部变量。例如：

ReadOnly Property F(i As Integer) As Integer  
 Get  
 If i = 0 Then  
 F = 1 ' Sets the return value.  
 Else  
 F = F(i - 1) ' Recursive call.  
 End If  
 End Get  
End Property

使用括号会导致出现不明确的情形（如 F(1)，其中 F 是类型为一维数组的属性）。在所有不明确的情形中，名称都将解析为函数而非局部变量。例如：

ReadOnly Property F(i As Integer) As Integer()  
 Get  
 If i = 0 Then  
 F = new Integer(2) { 1, 2, 3 }  
 Else  
 F = F(i – 1) ' Recursive call, not index.  
 End If  
 End Get  
End Property

当控制流离开 Get 访问器体时，局部变量的值将传递回调用表达式。由于调用 Get 访问器在概念上等效于读取变量的值，因此这被视为不良的 Get 访问器编程风格，具有明显的副作用，如下面的示例所示：

Class Counter  
 Private Value As Integer  
  
 Public ReadOnly Property NextValue() As Integer  
 Get  
 Value += 1  
 Return Value  
 End Get  
 End Property  
End Class

NextValue 属性的值取决于该属性以前被访问的次数。因此，访问该属性会产生明显示的副作用，该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 访问器总应编写为仅返回存储在变量中的值。实际上，Get 访问器通常会通过访问多个变量或调用方法来计算属性的值。但是，正确设计的 Get 访问器不会执行任何导致对象的状态发生可见变化的操作。

注意 Get 访问器对行放置的限制与子例程相同。开始和结束语句以及代码块都必须位于逻辑行的开头。

PropertyGetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Get LineTerminator  
 [ Block ]  
 End Get StatementTerminator

### Set 访问器声明

Set 访问器 (setter) 通过使用属性 set 声明进行声明。属性 set 声明由关键字 Set、可选参数列表和语句块组成。假设有一个名为 P 的属性，则 setter 声明会用与该属性相同的修饰符和参数列表隐式声明一个名为 set\_P 的方法。如果类型包含具有该名称的声明，则会导致发生编译时错误，但出于名称绑定目的而忽略隐式声明。

如果指定了参数列表，则该列表必须具有一个成员，该成员不能包含 ByVal 以外的修饰符，并且其类型必须与属性的类型相同。参数表示所设置的属性值。如果省略参数，则将隐式声明一个名为 Value 的参数。

注意 Set 访问器对行放置的限制与子例程相同。开始和结束语句以及代码块都必须位于逻辑行的开头。

PropertySetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Set [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Set StatementTerminator

### 默认属性

指定修饰符 Default 的属性称为默认属性。允许属性的任何类型都可以具有默认属性，包括接口。引用默认属性时无需用属性名限定实例。因此，如果指定以下类

Class Test  
 Public Default ReadOnly Property Item(i As Integer) As Integer  
 Get  
 Return i  
 End Get  
 End Property  
End Class

则代码

Module TestModule  
 Sub Main()  
 Dim x As Test = New Test()  
 Dim y As Integer  
  
 y = x(10)  
 End Sub  
End Module

相当于

Module TestModule  
 Sub Main()  
 Dim x As Test = New Test()  
 Dim y As Integer  
  
 y = x.Item(10)  
 End Sub  
End Module

将某一属性声明为 Default 后，在继承层次结构中以该名称重载的所有属性都将成为默认属性，无论它们是否已声明为 Default。如果基类按另一个名称声明了默认属性，则在派生类中声明某属性为 Default 时不需要任何其他修饰符，如 Shadows 或 Overrides 等。这是因为默认属性没有标识或签名，因此无法被隐藏或重载。例如：

Class Base  
 Public ReadOnly Default Property Item(i As Integer) As Integer  
 Get  
 Console.WriteLine("Base = " & i)  
 End Get  
 End Property  
End Class  
  
Class Derived  
 Inherits Base  
  
 ' This hides Item, but does not change the default property.  
 Public Shadows ReadOnly Property Item(i As Integer) As Integer  
 Get  
 Console.WriteLine("Derived = " & i)  
 End Get  
 End Property  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 ' This declares a new default property, but not Item.  
 ' This does not need to be declared Shadows  
 Public ReadOnly Default Property Value(i As Integer) As Integer  
 Get  
 Console.WriteLine("MoreDerived = " & i)  
 End Get  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As MoreDerived = New MoreDerived()  
 Dim y As Integer  
 Dim z As Derived = x  
  
 y = x(10) ' Calls MoreDerived.Value.  
 y = x.Item(10) ' Calls Derived.Item  
 y = z(10) ' Calls Base.Item  
 End Sub  
End Module

此程序会产生如下输出：

MoreDerived = 10  
Derived = 10  
Base = 10

在类型中声明的所有默认属性都必须具有相同的名称，并且为了清楚起见，必须指定 Default 修饰符。由于没有索引参数的默认属性会导致在分配包含类的实例时出现不明确情况，因此默认属性必须具有索引参数。此外，如果一个以特定名称重载的属性包含 Default 修饰符，则以该名称重载的所有属性也都必须指定该修饰符。默认属性不能为 Shared，并且该属性必须至少有一个访问器不能为 Private。

### 自动实现的属性

如果属性省略任何访问器的声明，则系统会自动提供该属性的实现，除非该属性是在接口中声明的或已声明为 MustOverride。只能自动实现不带参数的读/写属性；否则，会发生编译时错误。

自动实现的属性 x，即使是重写另一个属性的属性，也会引入一个与该属性具有相同类型的专用局部变量 \_x。如果该局部变量的名称与另一个声明之间存在冲突，则会报告编译时错误。自动实现的属性的 Get 访问器会返回该局部变量的值，该属性的 Set 访问器会设置该局部变量的值。例如，以下声明：

Public Property x() As Integer

大致等效于：

Private \_x As Integer  
Public Property x() As Integer  
 Get  
 Return \_x  
 End Get  
 Set (value As Integer)  
 \_x = value  
 End Set  
End Property

与变量声明一样，自动实现的属性也可以包含初始值设定项。例如：

Public Property x() As Integer = 10  
Public Shared Property y() As New Customer() With { .Name = "Bob" }

批注

在初始化自动实现的属性时，将通过该属性而不是基础字段进行初始化。因此，重写属性可根据需要截获初始化。

自动实现的属性允许数组初始值设定项，但无法显式指定数组界限。例如：

' Valid  
Property x As Integer() = {1, 2, 3}  
Property y As Integer(,) = {{1, 2, 3}, {12, 13, 14}, {11, 10, 9}}  
  
' Invalid  
Property x4(5) As Short

### 迭代器属性

*迭代器属性*是带有 Iterator 修饰符的属性。使用它的原因与使用迭代器方法 (‎10.1.2) 的原因相同，即使用它可以方便地生成 For Each 语句可使用的序列。迭代器属性的 Get 访问器与迭代器方法的解释相同。

迭代器属性必须具有显式 Get 访问器，其类型必须是 IEnumerator、IEnumerable 或 IEnumerator(Of T)；或者，对于某些 T，必须为 IEnumerable(Of T)。

下面是迭代器属性的一个示例：

Class Family  
 Property Daughters As New List(Of String) From {"Beth", "Diane"}  
 Property Sons As New List(Of String) From {"Abe", "Carl"}

ReadOnly Iterator Property Children As IEnumerable(Of String)  
 Get  
 For Each name In Daughters : Yield name : Next  
 For Each name In Sons : Yield name : Next  
 End Get  
 End Property  
End Class

Module Module1  
 Sub Main()  
 Dim x As New Family  
 For Each c In x.Children  
 Console.WriteLine(c) ' prints Beth, Diane, Abe, Carl  
 Next  
 End Sub  
End Module

## 运算符

运算符是为包含类定义现有 Visual Basic 运算符含义的方法。在通过表达式向类应用运算符时，会将运算符编译为对该类中定义的运算符方法的调用。为类定义运算符也称为重载运算符。不能重载已存在的运算符；实际上，这主要适用于转换运算符。例如，不能重载派生类到基类的转换：

Class Base  
End Class  
  
Class Derived  
 ' Cannot redefine conversion from Derived to Base,  
 ' conversion will be ignored.  
 Public Shared Widening Operator CType(s As Derived) As Base  
 ...  
 End Operator  
End Class

运算符也可以按该词的常规意义进行重载：

Class Base  
 Public Shared Widening Operator CType(b As Base) As Integer  
 ...  
 End Operator  
  
 Public Shared Narrowing Operator CType(i As Integer) As Base  
 ...  
 End Operator  
End Class

运算符声明不会将名称显式添加到包含类型的声明空间；但它们会隐式声明以字符“op\_”开头的对应方法。以下各部分列出了与每个运算符相对应的方法名称。

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。所有运算符声明都存在某些限制：

运算符声明必须始终为 Public 和 Shared。在采用 Public 修饰符的上下文中可以省略该修饰符。

运算符的形参不能声明为 ByRef、Optional 或 ParamArray。

至少有一个操作数或返回值的类型必须是包含运算符的类型。

没有为运算符定义的函数返回变量。因此，必须使用 Return 语句从运算符体返回值。

这些限制的唯一例外是可以为 null 的值类型。由于可以为 null 的值类型没有实际的类型定义，因此值类型可以为该类型的可以为 null 版本声明用户定义的运算符。在确定类型是否可以声明特定的用户定义运算符时，请首先将 ? 修饰符从声明中涉及的所有类型中去除，以便进行有效性检查。此宽松规则不适用于 IsTrue 和 IsFalse 运算符的返回类型；它们必须仍返回 Boolean，而不是 Boolean?。

运算符声明不能修改运算符的优先级和关联性。

注意 运算符对行放置的限制与子例程的相同。开始和结束语句以及代码块都必须位于逻辑行的开头。

OperatorDeclaration ::=  
 [ Attributes ] [ OperatorModifier+ ] Operator OverloadableOperator OpenParenthesis ParameterList CloseParenthesis  
 [ As [ Attributes ] TypeName ] LineTerminator  
 [ Block ]  
 End Operator StatementTerminator

OperatorModifier ::= Public | Shared | Overloads | Shadows | Widening | Narrowing

OverloadableOperator ::=  
 + | - | \* | / | \ | & | Like | Mod | And | Or | Xor | ^ | < < | > > |  
 = | < > | > | < | > = | < = | Not | IsTrue | IsFalse | CType

### 一元运算符

可以重载下列一元运算符：

一元加号运算符 +（对应的方法：op\_UnaryPlus）

一元减号运算符 -（对应的方法：op\_UnaryNegation）

逻辑 Not 运算符（对应的方法：op\_OnesComplement）

IsTrue 和 IsFalse 运算符（对应的方法：op\_True 和 op\_False）

所有重载的一元运算符都必须采用包含类型的单个参数，可以返回任何类型，但 IsTrue 和 IsFalse 除外，这两者必须返回 Boolean。如果包含类型是泛型类型，则类型形参必须与包含类型的类型形参相匹配。例如，

Structure Complex  
 ...  
  
 Public Shared Operator +(v As Complex) As Complex  
 Return v  
 End Operator  
End Structure

如果类型重载 IsTrue 或 IsFalse 中的一个运算符，则另一个运算符也必须重载。如果只重载其中一个，则会导致编译时错误。

注意   IsTrue 和 IsFalse 不是保留字。

### 二元运算符

可以重载下列二元运算符：

加法 +、减法 -、乘法 \*、除法 /、整数除法 \、取模 Mod 和指数 ^ 运算符（对应的方法：op\_Addition、op\_Subtraction、op\_Multiply、op\_Division、op\_IntegerDivision、op\_Modulus 和 op\_Exponent）

关系运算符 =、<>、<、>、<=、>=（对应方法：op\_Equality、op\_Inequality、op\_LessThan、op\_GreaterThan、op\_LessThanOrEqual、op\_GreaterThanOrEqual）

注意虽然可以重载相等运算符，但不能重载赋值运算符（仅用在赋值语句中）。

Like 运算符（对应的方法：op\_Like）

串联运算符 &（对应的方法：op\_Concatenate）

逻辑 And、Or 和 Xor 运算符（对应的方法：op\_BitwiseAnd、op\_BitwiseOr 和 op\_ExclusiveOr）

移位运算符 << 和 >>（对应方法：op\_LeftShift、op\_RightShift）

所有重载的二元运算符都必须采用包含类型作为形参之一。如果包含类型是泛型类型，则类型形参必须与包含类型的类型形参相匹配。移位运算符进一步限制了此规则，要求第一个形参属于包含类型；第二个形参必须始终为 Integer 类型。

下列二元运算符必须成对声明：

运算符 = 和运算符 <>

运算符 > 和运算符 <

运算符 >= 和运算符 <=

如果声明了上述运算符对中的一个，则也必须用匹配的形参和返回类型声明另一个，否则会导致编译时错误。

批注

要求成对声明关系运算符的目的是在重载运算符中尽量确保最低限度的逻辑一致性。

与关系运算符相反，强烈建议不要同时重载除法和整数除法运算符，但这样做不会出错。

批注

通常，这两种除法应是完全不同的：支持除法的类型要么是整数（在这种情况下应支持 \），要么不是整数（在这种情况下应支持 /）。我们考虑过将同时定义这两个运算符的做法报为错误，但由于这两个运算符的语言通常不会像 Visual Basic 那样区分这两种除法，因此我们认为允许该做法是最安全的，但强烈建议不要这样做。

不能直接重载复合赋值运算符。相反，重载对应的二元运算符时，复合赋值运算符将使用重载的运算符。例如：

Structure Complex  
 ...  
  
 Public Shared Operator +(x As Complex, y As Complex) \_  
 As Complex  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim c1, c2 As Complex  
 ' Calls the overloaded + operator  
 c1 += c2  
 End Sub  
End Module

### 转换运算符

转换运算符定义类型之间的新转换。这些新转换称为用户定义的转换。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）。转换必须归类为扩大转换或收缩转换。包含 Widening 关键字的转换运算符声明引入一种用户定义的扩大转换（对应的方法：op\_Implicit）。包含 Narrowing 关键字的转换运算符声明引入一种用户定义的收缩转换（对应的方法：op\_Explicit）。

一般来说，用户定义的扩大转换应设计为绝不会引发异常，也绝不会丢失信息。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收缩转换。在下面的示例中：

Structure Digit  
 Dim value As Byte  
  
 Public Sub New(value As Byte)  
 if value < 0 OrElse value > 9 Then Throw New ArgumentException()  
 Me.value = value  
 End Sub  
  
 Public Shared Widening Operator CType(d As Digit) As Byte  
 Return d.value  
 End Operator  
  
 Public Shared Narrowing Operator CType(b As Byte) As Digit  
 Return New Digit(b)  
 End Operator  
End Structure

从 Digit 到 Byte 的转换是扩大转换，因为它绝不会引发异常或丢失信息，而从 Byte 到 Digit 的转换是收缩转换，因为 Digit 只能表示 Byte 的可能值的子集。

与可重载的所有其他类型成员不同，转换运算符的签名包含转换的目标类型。这是唯一一种由返回类型参与签名的类型成员。不过，转换运算符的扩大或收缩分类不是运算符签名的一部分。因此，类或结构不能用相同的源和目标类型同时声明扩大转换运算符和收缩转换运算符。

用户定义的转换运算符必须在包含类型之间转换；例如，类 C 可以定义从 C 到 Integer 以及从 Integer 到 C 的转换，但不能定义从 Integer 到 Boolean 的转换。如果包含类型是泛型类型，则类型形参必须与包含类型的类型形参相匹配。此外，也不能重新定义内部（即非用户定义的）转换。因此，在下列情况下类型不能声明转换：

源类型与目标类型相同。

源类型和目标类型不是定义转换运算符的类型。

源类型或目标类型是接口类型。

源类型和目标类型通过继承相关（包括 Object）。

这些规则的唯一例外是可以为 null 的值类型。由于可以为 null 的值类型没有实际的类型定义，因此值类型可以为该类型的可以为 null 的版本声明用户定义的转换。在确定类型是否可以声明特定的用户定义转换时，请首先将 ? 修饰符从声明中涉及的所有类型中去除，以便进行有效性检查。因此，以下声明是有效的，因为 S 可定义从 S 到 T 的转换：

Structure T  
 ...  
End Structure  
  
Structure S  
 Public Shared Widening Operator CType(ByVal v As S?) As T  
 ...  
 End Operator  
End Structure

以下声明是无效的，因为结构 S 不能定义从 S 到 S 的转换：

Structure S  
 Public Shared Widening Operator CType(ByVal v As S) As S?  
 ...  
 End Operator  
End Structure

### 运算符映射

由于 Visual Basic 支持的运算符集可能不会与 .NET Framework 中其他语言支持的运算符集完全相同，因此某些运算符在定义或使用时会专门映射到其他运算符。具体包括：

定义整数除法运算符将自动定义调用整数除法运算符的常规除法运算符（仅对其他语言可用）。

对于区分逻辑和按位运算符的其他语言，重载Not、And 和 Or 运算符将只重载按位运算符。

在区分逻辑和按位运算符的语言（也就是对 Not、And 和 Or 分别使用 op\_LogicalNot、op\_LogicalAnd 和 op\_LogicalOr 的语言）中，只重载逻辑运算符的类会将其逻辑运算符映射到 Visual Basic 逻辑运算符。如果同时重载逻辑和按位运算符，则将只使用按位运算符。

对于区分有符号和无符号移位运算符的其他语言，重载 << 和 >> 运算符将只重载有符号运算符。

只重载无符号移位运算符的类会将无符号移位运算符映射到对应的 Visual Basic 移位运算符。如果同时重载无符号和有符号移位运算符，则将只使用有符号移位运算符。

# 语句

语句表示可执行代码。

Statement ::=  
 LabelDeclarationStatement |  
 LocalDeclarationStatement |  
 WithStatement |  
 SyncLockStatement |  
 EventStatement |  
 AssignmentStatement |  
 InvocationStatement |  
 ConditionalStatement |  
 LoopStatement |  
 ErrorHandlingStatement |  
 BranchStatement |  
 ArrayHandlingStatement |  
 UsingStatement |  
 AwaitStatement |  
 YieldStatement

批注

Microsoft Visual Basic 编译器仅允许使用以关键字或标识符开头的语句。因此，举例来说，允许调用语句“Call (Console).WriteLine”，但不允许调用语句“(Console).WriteLine”。

## 控制流

*控制流*是在其中执行语句和表达式的序列。执行顺序取决于特定语句或表达式。

例如，计算加法运算符（11.3 一节）时，首先计算左操作数，然后计算右操作数，最后再计算运算符本身。对于块的执行顺序（10.1.2 一节），首先执行第一个子语句，然后依次逐个执行块中的语句。

*控制点*概念隐含在此顺序中，它将是下一个执行的运算。当某个方法被调用时，它会创建该方法的*实例*。方法实例由其自己的形参和局部变量的副本及其自己的控制点组成。

### 常规方法

下面是一个常规方法的示例

Function Test() As Integer  
 Console.WriteLine("hello")  
 Return 1  
End Sub

Dim x = Test() ' invokes the function, prints "hello", assigns 1 to x

当某个常规方法被调用时，

1. 首先，将创建该方法特定于该调用的实例。此实例包括该方法的所有形参和局部变量的副本。
2. 然后，其所有形参都将初始化为提供的值，其所有局部变量则初始化为其类型的默认值。
3. 对于 Function，还会初始化一个称为*函数返回变量* 的隐式局部变量，其名称是该函数的名称，其类型是该函数的返回类型，其初始值是其类型的默认值。
4. 然后，将在方法体的第一条语句设置该方法实例的控制点，并且方法体立即开始从该处执行（10.1.4 一节）。

当控制流到达标记结尾的 End Function 或 End Sub，或通过显式的 Return 或 Exit 语句正常退出方法体时，控制流将返回到该方法实例的调用方。如果存在函数返回变量，则调用的结果就是此变量的最终值。

当控制流由于未经处理的异常退出方法体时，该异常将传播给调用方。

控制流退出后，不再存在对该方法实例的任何有效引用。如果该方法实例仅包含对其局部变量或形参副本的引用，则它们可能被当作垃圾回收。

### 迭代器方法

迭代器方法是生成序列的一种简便方法，For Each 语句可以使用此类序列。迭代器方法使用 Yield 语句（10.15 一节）来提供序列的元素。（没有 Yield 语句的迭代器方法将产生空序列。）下面是迭代器方法的一个示例：

Iterator Function Test() As IEnumerable(Of Integer)  
 Console.WriteLine("hello")  
 Yield 1  
 Yield 2  
End Function

Dim en = Test()  
For Each x In en ' prints "hello" before the first x  
 Console.WriteLine(x) ' prints "1" and then "2"  
Next

调用返回类型为 IEnumerator(Of T) 的迭代器方法时，

1. 首先，将创建该迭代器方法特定于该调用的实例。此实例包括该方法的所有形参和局部变量的副本。
2. 然后，其所有形参都将初始化为提供的值，其所有局部变量则初始化为其类型的默认值。
3. 还会初始化一个称为 *iterator current 变量*的隐式局部变量，其类型为 T，其初始值为其类型的默认值。
4. 然后，将在方法体的第一个语句处设置该方法实例的控制点。
5. 然后，将创建与此方法实例相关联的*迭代器对象*。该迭代器对象将实现声明的返回类型，并会有如下所述的行为。
6. 然后，控制流将在调用方*立即* 恢复，调用结果是迭代器对象。请注意，控制上的这个转移并不会退出迭代器方法实例，因此不会导致 finally 处理程序执行。该方法实例仍然由迭代器对象引用，并且，只要存在对该迭代器对象的有效引用，就不会被作为垃圾回收。

当访问迭代器对象的 Current 属性时，将返回该调用的*当前变量*。

当调用迭代器对象的 MoveNext 方法时，调用不会创建新的方法实例。而会使用现有的方法实例（及其控制点、局部变量和形参），即首次调用迭代器方法时创建的实例。控制流将在该方法实例的控制点处恢复执行，并如常继续执行迭代器方法体。

当调用迭代器对象的 Dispose 方法时，又会使用现有方法实例。控制流将在该方法实例的控制点处恢复执行，但是其行为立即就像下一个操作是一个 Exit Function 语句一样。

仅当对迭代器对象的 MoveNext 或 Dispose 先前的所有调用都已返回到其调用方时，对该迭代器对象的 MoveNext 或 Dispose 调用的行为才符合以上描述。如果那些调用尚未返回，则其行为不确定。

当控制流到达标记结尾的 End Function，或通过显式 Return 或 Exit 语句正常退出迭代器方法体时，它必须在对迭代器对象的 MoveNext 或 Dispose 函数调用的上下文中退出以恢复迭代器方法实例，并且它已使用首次调用该迭代器方法时创建的方法实例。该实例的控制点将留在 End Function 语句处，且控制流将在调用方恢复；如果它已通过调用 MoveNext 恢复，则将向调用方返回 False 值。

当控制流由于未经处理的异常而退出迭代器方法体时，异常将传播到调用方，这又将通过调用 MoveNext 或 Dispose 实现。

对于迭代器函数的其他可能的返回类型，

1. 对于某些 T，当调用返回类型为 IEnumerable(Of T) 的迭代器方法时，将首先创建该方法中所有形参的一个实例（特定于该迭代器方法的调用），并且这些形参都将用提供的值初始化。调用的结果是一个实现返回类型的对象。如果此对象的 GetEnumerator 方法被调用，则它将针对 GetEnumerator 调用创建该方法中所有形参和局部变量的一个特定实例。它会将这些形参初始化为已保存的值，并继续执行以上迭代器方法。
2. 当调用返回类型为非泛型接口 IEnumerator 的迭代器方法时，行为与 IEnumerator(Of Object) 完全相同。
3. 当调用返回类型为非泛型接口 IEnumerable 的迭代器方法时，行为与 IEnumerable(Of Object) 完全相同。

### 异步方法

使用异步方法可以方便地执行长时间运行的工作，并可避免一些问题，例如阻塞应用程序的 UI。“异步”的英文简写是 *Async*，就是说异步方法的调用方将立即恢复其执行，但是异步方法实例将在以后的某个时间最终完成。按照惯例，异步方法名称跟有后缀“Async”。

Async Function TestAsync() As Task(Of String)  
 Console.WriteLine("hello")  
 Await Task.Delay(100)  
 Return "world"  
End Function

Dim t = TestAsync() ' prints "hello"  
Console.WriteLine(Await t) ' prints "world"

批注

异步方法*不* 在后台线程中运行。它们允许方法通过 Await 操作符将自身挂起，并安排自身在对某个事件进行响应时恢复运行。

当调用某个异步方法时，

1. 首先将创建该异步方法特定于该调用的实例。此实例包括该方法的所有形参和局部变量的副本。
2. 然后，其所有形参都将初始化为提供的值，其所有局部变量则初始化为其类型的默认值。
3. 对于某些 T 的返回类型为 Task(Of T) 的异步方法，还会初始化一个名为*任务返回变量* 的隐式局部变量，该变量类型为 T，初始值为 T 的默认值。
4. 如果该异步方法对于某个 T 是返回类型为 Task 或 Task(Of T) 的 Function，则将隐式创建与当前调用相关联的一个该类型的对象。该对象称为*异步对象*，表示以后将通过执行该异步方法的实例完成的工作。当控制权在此异步方法实例的调用方恢复时，该调用方将接收此异步对象作为其调用结果。
5. 然后，将在异步方法体的第一条语句设置该实例的控制点，并且立即开始从该处执行方法体（10.1.2 一节）。

**恢复委托和当前调用方**

如 11.25“Await 运算符”一节中详细介绍的那样，执行 Await 表达式将能够挂起方法实例的控制点，并让控制流转向其他位置。控制流可稍后通过调用*恢复委托* 在同一实例的控制点恢复。请注意，此挂起操作执行时并不会退出异步方法，因此不会导致 finally 处理程序执行。该方法实例将仍由恢复委托和 Task 或 Task(Of T) 结果（如果有）引用，且只要存在对委托或结果的有效引用，就不会被作为垃圾回收。

批注

假定语句“Dim x = Await WorkAsync()”相当于以下内容的简短语法形式有助于理解：

Dim temp = WorkAsync().GetAwaiter()  
 If Not temp.IsCompleted Then  
 temp.OnCompleted(resumptionDelegate)  
 Return [task]  
 CONT: ' *resumptionDelegate* 调用在此恢复  
 End If  
 Dim x = temp.GetResult()

下面该方法实例的*当前调用方* 定义为初始调用方或恢复委托的调用方中较新的那一个。

当控制流到达标记结尾的 End Sub 或 End Function，或通过显式的 Return 或 Exit 语句，或由于未经处理的异常而退出异步方法体时，实例控制点将设置到方法结尾。之后的行为取决于异步方法的返回类型。

1. 对于返回类型为 Task 的 Async Function：
   1. 如果控制流由于未经处理的异常而退出，异步对象的状态将设置为 TaskStatus.Faulted，且其 Exception.InnerException 属性将设置为该异常（例外情况：某些 OperationCanceledException 之类由实现定义的异常将它更改为 TaskStatus.Canceled）。控制流将在当前调用方中恢复。
   2. 否则，异步对象状态将设置为 TaskStatus.Completed。控制流将在当前调用方中恢复。

批注  
  
任务的全部意义和异步方法之所以有趣的原因是，当任务变为“完成”状态时，等待它的所有方法都将很快能够执行它们的恢复委托，即它们将不再被阻塞。

1. 对于某些 T，返回类型为 Task(Of T) 的 Async Function：行为就如上述行为，不同的是，在非异常情况下，异步对象的 Result 属性也将设置为任务返回变量的最终值。
2. 对于 Async Sub：
   1. 如果控制流由于未经处理的异常退出，则该异常将通过某种特定于实现的方式传播到环境。控制流将在当前调用方中恢复。
   2. 否则，控制流将只是在当前调用方中恢复。

批注

Async Sub 存在某些特定于 Microsoft 的行为。

如果 SynchronizationContext.Current 在调用开头是 Nothing，则 Async Sub 中任何未经处理的异常都会发送到 Threadpool。

如果 SynchronizationContext.Current 在调用开头不是 Nothing，则在该方法开始之前将对该上下文调用 OperationStarted()，并在该方法结束之后调用 OperationCompleted()。此外，任何未经处理的异常都将被发送，以在同步上下文中重新引发。

这就意味着，在 UI 应用程序中，对于在 UI 线程中调用的 Async Sub，未能处理的所有异常都将重新发送到该 UI 线程。

批注

人们通常认为可变结构是不好的做法，异步方法和迭代器方法都不支持此类结构。具体来说，结构中每次调用异步或迭代器方法时，都将隐式对在调用时复制的该结构的*副本* 执行操作。因此，举例来说，

Structure S  
 Dim x As Integer  
 Async Sub Mutate()  
 x = 2  
 End Sub  
 End Structure

Dim s As New S With {.x = 1}  
 s.Mutate()  
 Console.WriteLine(s.x) ’ 输出“1”

### 块和标签

一组可执行语句称为一个语句块。一个语句块的执行过程将从该语句块中的第一个语句开始。在执行完一个语句后，除非语句将执行过程转移到其他位置或发生异常，否则会按照词法顺序执行下一个语句。

在语句块中，逻辑行上的语句划分并不重要（标签声明语句除外）。标签是一个标识符，用于标识语句块中可用作分支语句（如 GoTo）的目标的特定位置。

标签声明语句必须出现在逻辑行的开头处，而标签可以是标识符或整数文本。由于标签声明语句和调用语句都可以由单个标识符构成，因此始终将位于逻辑行开头处的单个标识符视为一个标签声明语句。标签声明语句的后面必须总跟有一个冒号，即使在同一逻辑行上没有任何后跟语句也是如此。

标签具有自己的声明空间，并不影响其他标识符。以下是一个有效示例，其中将名称变量 x 同时用作参数和标签。

Function F(x As Integer) As Integer  
 If x >= 0 Then  
 GoTo x  
 End If  
 x = -x  
x:   
 Return x  
End Function

标签的范围其所属方法的主体。

为了提高可读性，在本规范中，将涉及多个子语句的语句产生式视为单个产生式，即使这些子语句中的每个子语句自身可以位于一个标记的行上也是如此。

Block ::= [ Statements+ ]

LabelDeclarationStatement ::= LabelName :

LabelName ::= Identifier | IntLiteral

Statements ::=  
 [ Statement ] |  
 Statements : [ Statement ]

### 局部变量和参数

10.1.1-10.1.3 这几节详细介绍了如何以及何时创建方法实例及方法的局部变量和形参的副本。此外，如 10.9 一节所述，每次进入循环体时，都将制作该循环内声明的每个局部变量的新副本，因此该方法实例现在包含其局部变量的此副本，而不是前一个副本。

所有局部变量都将初始化为其类型的默认值。局部变量和参数始终是可公开访问的。在某个局部变量声明之前的文本位置中引用该局部变量是错误的，如以下示例所示：

Class A  
 Private i As Integer = 0  
  
 Sub F()  
 i = 1  
 Dim i As Integer ' Error, use precedes declaration.  
 i = 2  
 End Sub  
  
 Sub G()  
 Dim a As Integer = 1  
 Dim b As Integer = a ' This is valid.  
 End Sub  
End Class

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，这会导致出现错误，因为它在文本上位于该变量的声明之前。在 G 方法中，后面的变量声明引用的是同一局部变量声明内的前一变量声明中声明的局部变量。

方法中的每个块都会为局部变量创建一个声明空间。名称可通过方法体中的局部变量声明引入此声明空间，并可通过方法的参数列表引入最外层块的声明空间。块不允许通过嵌套来隐藏名称：在块中声明某个名称之后，无法在任何嵌套块中重新声明该名称。

因此，在下面的示例中，F 和 G 方法都存在错误，因为名称 i 已在外部块中声明，不能在内部块中再次声明。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。

Class A  
 Sub F()  
 Dim i As Integer = 0  
 If True Then  
 Dim i As Integer = 1  
 End If  
 End Sub  
  
 Sub G()  
 If True Then  
 Dim i As Integer = 0  
 End If  
 Dim i As Integer = 1  
 End Sub   
  
 Sub H()  
 If True Then  
 Dim i As Integer = 0  
 End If  
 If True Then  
 Dim i As Integer = 1  
 End If  
 End Sub  
  
 Sub I()   
 For i As Integer = 0 To 9  
 H()  
 Next i  
   
 For i As Integer = 0 To 9  
 H()  
 Next i  
 End Sub   
End Class

当方法是一个函数时，会使用与表示函数返回值的方法相同的名称在方法体的声明空间内隐式声明一个特殊局部变量。该局部变量在用于表达式时会具有特殊的名称解析语义。如果在一个上下文（此上下文应为一个归类为方法组的表达式，如调用表达式）中使用此局部变量，则名称将解析为函数而非局部变量。例如：

Function F(i As Integer) As Integer  
 If i = 0 Then  
 F = 1 ' Sets the return value.  
 Else  
 F = F(i - 1) ' Recursive call.  
 End If  
End Function

使用括号会导致指代不明（例如 F(1)，其中 F 是一个返回类型为一维数组的函数）；在所有指代不明确的情形中，名称都将解析为函数而非局部变量。例如：

Function F(i As Integer) As Integer()  
 If i = 0 Then  
 F = new Integer(2) { 1, 2, 3 }  
 Else  
 F = F(i - 1) ' Recursive call, not an index.  
 End If  
End Function

当控制流离开方法体时，局部变量的值将传递回调用表达式。如果方法是一个子例程，则没有此类隐式局部变量，并且控制只返回调用表达式。

## 局部声明语句

局部声明语句声明的是新的局部变量、局部常量或静态变量。局部变量和局部常量 等效于其范围为方法的实例变量和常量，并且二者的声明方式相同。静态变量与 Shared 变量类似，并且是使用 Static 修饰符声明的。

静态变量是在方法调用中保留其值的局部变量。在非共享方法中声明的静态变量是基于实例的：包含方法的类型的每个实例都具有自己的静态变量副本。在 Shared 方法中声明的静态变量是基于类型的；所有实例仅有一个静态变量副本。局部变量在每个项进入方法中时将初始化为其类型的默认值，而静态变量仅在初始化类型或类型实例时才初始化为其类型的默认值。静态变量不能在结构或泛型方法中声明。

局部变量、局部常量或静态变量总是具有公共可访问性，且无法指定可访问性修饰符。如果局部声明语句中未指定任何类型，则将执行下列步骤来确定局部声明的类型：

如果声明具有一个类型字符，则该类型字符的类型为局部声明的类型。

如果局部声明是一个局部常量，或者局部声明是一个带有初始值设定项的局部变量并且正在使用局部变量类型推理，则从该初始值设定项的类型推断出局部声明的类型。如果该初始值设定项引用局部声明，则将出现编译时错误。（使用初始值设定项需要局部常量。）

如果未使用严格语义，则局部声明语句的类型隐式为 Object。

其他情况下，将发生编译时错误。

如果在具有数组大小或数组类型标识符的局部声明语句中未指定任何类型，则局部声明的类型为一个具有指定秩的数组，且先前的步骤用于确定数组的元素类型。如果使用了局部变量类型推断，则初始值设定项的类型必须为一个具有与局部声明语句相同的数组形状（即数组类型修饰符）的数组类型。请注意，推断出的元素类型仍可能会是数组类型。例如：

Option Infer On  
  
Module Test  
 Sub Main()  
 ' Error: initializer is not an array type  
 Dim x() = 1  
  
 ' Type is Integer()  
 Dim y() = New Integer() {}  
  
 ' Type is Integer()()  
 Dim z() = New Integer()() {}  
  
 ' Type is Integer()()()

Dim a()() = New Integer()()() {}  
  
 ' Error: initializer does not have same array shape  
 Dim b()() = New Integer(,)() {}  
 End Sub  
End Module

如果在具有可为 null 的类型修饰符的局部声明语句中未指定任何类型，则局部声明的类型为推断出的类型的可为 null 的版本或推断出的类型（如果它本身就是一个可为 null 的值类型）。如果推断出的类型不是一个可为 null 的值类型，则将发生编译时错误。如果将一个可为 null 的类型修饰符和一个数组大小或数组类型修饰符同时置于一个不带任何类型的局部声明语句中，则可为 null 的类型修饰符将视为可应用于数组的元素类型，且先前的步骤用于确定该元素类型。

局部声明语句中的变量初始值设定项等效于置于声明的文本位置的赋值语句。因此，如果执行过程分支到局部声明语句中，则不会执行变量初始值设定项。如果多次执行局部声明语句，则会执行变量初始值设定项相同的次数。静态变量仅在第一次时执行其初始值设定项。如果在初始化一个静态变量时发生异常，则将此静态变量视为是使用其类型的默认值初始化的。

下面的示例显示初始值设定项的用法：

Module Test  
 Sub F()  
 Static x As Integer = 5  
  
 Console.WriteLine("Static variable x = " & x)  
 x += 1  
 End Sub  
  
 Sub Main()  
 Dim i As Integer  
  
 For i = 1 to 3  
 F()  
 Next i  
  
 i = 3  
label:  
 Dim y As Integer = 8  
  
 If i > 0 Then  
 Console.WriteLine("Local variable y = " & y)  
 y -= 1  
 i -= 1  
 GoTo label  
 End If  
 End Sub  
End Module

此程序输出：

Static variable x = 5  
Static variable x = 6  
Static variable x = 7  
Local variable y = 8  
Local variable y = 8  
Local variable y = 8

静态局部变量初始值设定项是线程安全的，它在初始化过程中将受到保护以防止发生异常。如果一个静态局部变量初始值设定项发生异常，则此静态局部变量将具有自己的默认值且不会进行初始化。静态局部变量初始值设定项

Module Test  
 Sub F()  
 Static x As Integer = 5  
 End Sub  
End Module

相当于

Imports System.Threading  
Imports Microsoft.VisualBasic.CompilerServices  
  
Module Test  
 Class InitFlag  
 Public State As Short  
 End Class  
  
 Private xInitFlag As InitFlag = New InitFlag()  
  
 Sub F()  
 Dim x As Integer  
  
 If xInitFlag.State <> 1 Then  
 Monitor.Enter(xInitFlag)  
 Try  
 If xInitFlag.State = 0 Then  
 xInitFlag.State = 2  
 x = 5  
 Else If xInitFlag.State = 2 Then  
 Throw New IncompleteInitialization()  
 End If  
 Finally  
 xInitFlag.State = 1  
 Monitor.Exit(xInitFlag)  
 End Try  
 End If  
 End Sub  
End Module

局部变量、局部常量和静态变量的范围为声明它们的语句块。静态变量的特别之处在于其名称在整个方法中只能使用一次。例如，使用同一名称指定两个静态变量声明是无效的，即使这两个变量位于不同的块中也是如此。

LocalDeclarationStatement ::= LocalModifier VariableDeclarators StatementTerminator

LocalModifier ::= Static | Dim | Const

### 隐式局部声明

除局部声明语句之外，局部变量也可以通过使用它来对它进行隐式声明。一个使用未解析为其他内容的名称的名称表达式会按照该名称来声明局部变量。例如：

Option Explicit Off  
  
Module Test  
 Sub Main()  
 x = 10  
 y = 20  
 Console.WriteLine(x + y)  
 End Sub  
End Module

隐式局部声明只能在可接受归类为一个变量的表达式的表达式上下文中出现。此规则有一个例外，即当局部变量是函数调用表达式、索引表达式或成员访问表达式的目标时，将无法对其进行隐式声明。

隐式局部变量将被视为它们在包含方法的开头处被声明。因此，其范围始终为整个方法体，即使在 lambda 表达式内部声明也是如此。以下列代码为例：

Option Explicit Off   
  
Module Test  
 Sub Main()  
 Dim x = Sub()  
 a = 10  
 End Sub  
 Dim y = Sub()  
 Console.WriteLine(a)  
 End Sub  
  
 x()  
 y()  
 End Sub  
End Module

将输出值 10。如果未将任何类型字符附加到变量名，则隐式局部变量将被类型化为 Object；否则，该变量的类型将为类型字符的类型。局部变量类型推断不用于隐式局部变量。

如果编译环境或 Option Explicit 指定显式局部声明，则必须对所有局部变量进行显式声明，而不允许进行隐式变量声明。

## With 语句

利用 With 语句，可以多次引用某表达式的成员，而无需多次指定该表达式。必须将表达式归类为一个值，并且在项进入块后，计算一次表达式。在 With 语句块中，会将以句点或感叹号开始的成员访问表达式或字典访问表达式视为其前面带有 With 表达式来进行计算。例如：

Structure Test  
 Public x As Integer  
  
 Function F() As Integer  
 Return 10  
 End Sub  
End Structure  
  
Module TestModule  
 Sub Main()  
 Dim y As Test  
  
 With y  
 .x = 10  
 Console.WriteLine(.x)  
 .x = .F()  
 End With  
 End Sub  
End Module

从 With 语句块的外部分支到该块是无效的。

WithStatement ::=  
 With Expression StatementTerminator  
 [ Block ]  
 End With StatementTerminator

## SyncLock 语句

利用 SyncLock 语句，可在一个表达式上同步多个语句，从而确保多个执行线程不会同时执行同一语句。必须将表达式归类为一个值，并且在项进入块后，计算一次表达式。在进入 SyncLock 块后，将对指定的表达式调用 Shared 方法 System.Threading.Monitor.Enter，此方法会一直阻塞，直到执行线程对该表达式返回的对象拥有排他锁。SyncLock 语句中的表达式类型必须为引用类型。例如：

Class Test  
 Private count As Integer = 0  
  
 Public Function Add() As Integer  
 SyncLock Me  
 count += 1  
 Add = count  
 End SyncLock  
 End Function  
  
 Public Function Subtract() As Integer  
 SyncLock Me  
 count -= 1  
 Subtract = count  
 End SyncLock  
 End Function  
End Class

上述示例对类 Test 的特定实例进行同步，以确保不会一次为某个特定实例在 count 变量中增加或减去多个执行线程。

SyncLock 块由 Try 语句隐式包含，该语句的 Finally 块会对该表达式调用 Shared 方法 System.Threading.Monitor.Exit。这将确保锁被释放（即使引发异常时也是如此）。因此，从 SyncLock 块的外部分支到该块是无效的，Resume 和 Resume Next 会将 SyncLock 块视为一个语句。上面的示例与下面的代码等效：

Class Test  
 Private count As Integer = 0  
  
 Public Function Add() As Integer  
 Try  
 System.Threading.Monitor.Enter(Me)  
  
 count += 1  
 Add = count  
 Finally  
 System.Threading.Monitor.Exit(Me)  
 End Try  
 End Function  
  
 Public Function Subtract() As Integer  
 Try  
 System.Threading.Monitor.Enter(Me)  
  
 count -= 1  
 Subtract = count  
 Finally  
 System.Threading.Monitor.Exit(Me)  
 End Try  
 End Function  
End Class

SyncLockStatement ::=  
 SyncLock Expression StatementTerminator  
 [ Block ]  
 End SyncLock StatementTerminator

## 事件语句

RaiseEvent、AddHandler 和 RemoveHandler 语句会引发事件并动态处理事件。

EventStatement ::=  
 RaiseEventStatement |  
 AddHandlerStatement |  
 RemoveHandlerStatement

### RaiseEvent 语句

RaiseEvent 语句会通知事件处理程序已发生某个特定事件。RaiseEvent 语句中的简单名称表达式将被解释为对 Me 的成员查找。因此，RaiseEvent x 解释为 RaiseEvent Me.x。表达式的结果必须归为在类本身中定义的事件的事件访问类别；在基类型上定义的事件不能用在 RaiseEvent 语句中。

程序使用提供的参数（如果有），将 RaiseEvent 语句作为对事件委托的 Invoke 方法的调用进行处理。如果委托的值为 Nothing，则不会引发任何异常。如果没有参数，则可以省略括号。例如：

Class Raiser  
 Public Event E1(Count As Integer)  
  
 Public Sub Raise()  
 Static RaiseCount As Integer = 0  
  
 RaiseCount += 1  
 RaiseEvent E1(RaiseCount)  
 End Sub  
End Class  
  
Module Test  
 Private WithEvents x As Raiser  
  
 Private Sub E1Handler(Count As Integer) Handles x.E1  
 Console.WriteLine("Raise #" & Count)  
 End Sub  
  
 Public Sub Main()  
 x = New Raiser  
 x.Raise() ' Prints "Raise #1".  
 x.Raise() ' Prints "Raise #2".  
 x.Raise() ' Prints "Raise #3".  
 End Sub  
End Module

上面的类 Raiser 等效于：

Class Raiser  
 Public Event E1(Count As Integer)  
  
 Public Sub Raise()  
 Static RaiseCount As Integer = 0  
 Dim TemporaryDelegate As E1EventHandler  
  
 RaiseCount += 1  
  
 ' Use a temporary to avoid a race condition.  
 TemporaryDelegate = E1Event  
 If Not TemporaryDelegate Is Nothing Then  
 TemporaryDelegate.Invoke(RaiseCount)  
 End If  
 End Sub  
End Class

RaiseEventStatement ::= RaiseEvent IdentifierOrKeyword [ OpenParenthesis  
 [ ArgumentList ] CloseParenthesis ] StatementTerminator

### AddHandler 和 RemoveHandler 语句

虽然大多数事件处理程序会通过 WithEvents 变量自动挂钩，但在运行时可能需要动态添加和移除事件处理程序。AddHandler 和 RemoveHandler 语句可执行此操作。

每个语句均采用两个参数：第一个参数必须是归类为一个事件访问的表达式，第二个参数必须是归类为一个值的表达式。第二个参数的类型必须是与事件访问关联的委托类型。例如：

Public Class Form1  
 Public Sub New()  
 ' Add Button1\_Click as an event handler for Button1's Click event.  
 AddHandler Button1.Click, AddressOf Button1\_Click  
 End Sub   
  
 Private Button1 As Button = New Button()  
  
 Sub Button1\_Click(sender As Object, e As EventArgs)  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
  
 Public Sub Disconnect()  
 RemoveHandler Button1.Click, AddressOf Button1\_Click  
 End Sub  
End Class

假定有一个事件 E，此语句会对实例调用相关的 add\_E 或 remove\_E 方法，以便添加或移除作为该事件的处理程序的委托。因此，上面的代码等效于：

Public Class Form1  
 Public Sub New()  
 Button1.add\_Click(AddressOf Button1\_Click)  
 End Sub   
  
 Private Button1 As Button = New Button()  
  
 Sub Button1\_Click(sender As Object, e As EventArgs)  
 Console.WriteLine("Button1 was clicked!")  
 End Sub  
  
 Public Sub Disconnect()  
 Button1.remove\_Click(AddressOf Button1\_Click)  
 End Sub  
End Class

AddHandlerStatement ::= AddHandler Expression Comma Expression StatementTerminator

RemoveHandlerStatement ::= RemoveHandler Expression Comma Expression StatementTerminator

## 赋值语句

赋值语句会将表达式的值赋给变量。赋值的类型有多种。

AssignmentStatement ::=  
 RegularAssignmentStatement |  
 CompoundAssignmentStatement |  
 MidAssignmentStatement

### 常规赋值语句

简单的赋值语句会将表达式的结果存储在变量中。赋值运算符左侧的表达式必须归类为一个变量或属性访问，而赋值运算符右侧的表达式必须归类为一个值。表达式的类型必须能隐式转换为变量或属性访问的类型。

如果赋予的变量是引用类型的数组元素，则将执行运行时检查以确保表达式与数组元素类型兼容。在下面的示例中，最后一个赋值会导致引发 System.ArrayTypeMismatchException，这是因为 ArrayList 的实例不能存储在 String 数组的元素中。

Dim sa(10) As String  
Dim oa As Object() = sa  
oa(0) = Nothing ' This is allowed.  
oa(1) = "Hello" ' This is allowed.  
oa(2) = New ArrayList() ' System.ArrayTypeMismatchException is thrown.

如果将赋值运算符左侧的表达式归类为一个变量，则赋值表达式会将值存储在变量中。如果将该表达式归类为一个属性访问，则赋值语句会将该属性访问转换为对属性的 Set 访问器的调用，并用值替换值参数。例如：

Module Test  
 Private PValue As Integer  
  
 Public Property P As Integer  
 Get  
 Return PValue  
 End Get  
  
 Set (Value As Integer)  
 PValue = Value  
 End Set  
 End Property  
  
 Sub Main()  
 ' The following two lines are equivalent.  
 P = 10  
 set\_P(10)  
 End Sub  
End Module

如果将变量或属性访问的目标类型化为一个值类型，而不将其归类为一个变量，则将发生编译时错误。例如：

Structure S  
 Public F As Integer  
End Structure  
  
Class C  
 Private PValue As S  
  
 Public Property P As S  
 Get  
 Return PValue  
 End Get  
  
 Set (Value As S)  
 PValue = Value  
 End Set  
 End Property  
End Class  
  
Module Test  
 Sub Main()  
 Dim ct As C = New C()  
 Dim rt As Object = new C()  
  
 ' Compile-time error: ct.P not classified as variable.  
 ct.P.F = 10  
  
 ' Run-time exception.  
 rt.P.F = 10  
 End Sub  
End Module

Note   赋值的语义取决于要对其赋值的变量或属性的类型。如果要对其赋值的变量是值类型，则赋值会将表达式的值复制到变量中。如果要对其赋值的变量是引用类型，则赋值会将引用（而非值本身）复制到变量中。如果变量的类型为 Object，则赋值语义取决于值的类型在运行时是值类型还是引用类型。

批注

对于内部类型（如 Integer 和 Date）来说，引用和值赋值语义是相同的，因为这些类型是不可变的。因此，语言可以将装箱内部类型上的引用赋值随意用作一种优化。从值的角度看，结果是相同的。

由于等号字符 (=) 既可用于赋值，又可用于相等比较，因此，在某些情况下（例如 x = y.ToString()，将无法确定是简单赋值还是调用语句。在所有此类情况下，赋值语句将优先于相等运算符。这意味着，上述示例表达式将被解释为 x = (y.ToString())，而不是 (x = y).ToString()。

RegularAssignmentStatement ::= Expression Equals Expression StatementTerminator

### 复合赋值语句

复合赋值语句采用的格式为 V OP= E（其中，OP 为有效的二元运算符）。赋值运算符左侧的表达式必须归类为一个变量或属性访问，而赋值运算符右侧的表达式必须归类为一个值。复合赋值语句等效于语句 V = V OP E，差异在于复合赋值运算符左侧的变量只能计算一次。下面的示例说明了此差异：

Module Test  
 Function GetIndex() As Integer  
 Console.WriteLine("Getting index")  
 Return 1  
 End Function  
  
 Sub Main()  
 Dim a(2) As Integer  
  
 Console.WriteLine("Simple assignment")  
 a(GetIndex()) = a(GetIndex()) + 1  
  
 Console.WriteLine("Compound assignment")  
 a(GetIndex()) += 1  
 End Sub  
End Module

表达式 a(GetIndex()) 对于简单赋值将计算两次；对于复合赋值，则只计算一次，因此代码将输出：

Simple assignment  
Getting index  
Getting index  
Compound assignment  
Getting index

CompoundAssignmentStatement ::= Expression CompoundBinaryOperator [ LineTerminator ]  
 Expression StatementTerminator

CompoundBinaryOperator ::= ^ = | \* = | / = | \ = | + = | - = | & = | < < = | > > =

### Mid 赋值语句

Mid 赋值语句可将一个字符串赋于另一个字符串中。此赋值左侧的语法与调用函数 Microsoft.VisualBasic.Strings.Mid 的语法相同。第一个参数为赋值的目标，必须归类为一个变量或属性访问（其类型既要能隐式转换为 String，又要能从 String 隐式转换）。第二个参数为从 1 开始的起始位置（此位置与目标字符串中应开始赋值的位置相对应），必须归类为一个其类型必须能隐式转换为 Integer 的值。第三个参数是一个可选参数，它是要从右侧值赋到目标字符串中的字符数，必须归类为一个其类型能够隐式转换为 Integer 的值。右侧是源字符串，必须归类为一个其类型能够隐式转换为 String 的值。右侧将被截断到长度参数（如果已指定）并替换左侧字符串中的字符（从起始位置开始）。如果右侧字符串包含的字符数少于第三个参数指定的字符数，则仅复制右侧字符串中的字符。

下面的示例显示 ab123fg：

Module Test  
 Sub Main()  
 Dim s1 As String = "abcdefg"  
 Dim s2 As String = "1234567"  
  
 Mid$(s1, 3, 3) = s2  
 Console.WriteLine(s1)  
 End Sub  
End Module

注意   Mid 不是保留字。

MidAssignmentStatement ::=  
 Mid [ $ ] OpenParenthesis Expression Comma Expression [ Comma Expression ] CloseParenthesis  
 Equals Expression StatementTerminator

## 调用语句

调用语句会调用前面带有可选关键字 Call 的方法。处理调用语句的方式与处理函数调用表达式的方式相同，但有如下所述的一些不同之处。调用表达式必须归类为一个值或 void。通过计算调用表达式得到的任何值都将被丢弃。

如果省略 Call 关键字，则调用表达式必须以标识符或关键字开头，或在 With 块中以 . 开头。因此，举例来说，“Call 1.ToString()”是有效语句，但“1.ToString()”则无效。（请注意，在表达式上下文中，调用表达式也无需以标识符开头。例如，“Dim x = 1.ToString()”是有效语句）。

在调用语句与调用表达式之间还有另一个区别：如果调用语句包含实参列表，则这将始终是该调用的实参列表。下面的示例说明了差异：

Module Test  
 Sub Main()  
 Call {Function() 15}(0)  
 ' error: (0) is taken as argument list, but array is not invokable  
  
 Call ({Function() 15}(0))  
 ' valid, since the invocation statement has no argument list  
  
 Dim x = {Function() 15}(0)  
 ' valid as an expression, since (0) is taken as an array-indexing  
  
 Call f("a")  
 ' error: ("a") is taken as argument list to the invocation of f  
  
 Call f()("a")  
 ' valid, since () is the argument list for the invocation of f  
  
 Dim y = f("a")  
 ' valid as an expression, since f("a") is interpreted as f()("a")  
 End Sub  
  
 Sub f() As Func(Of String,String)  
 Return Function(x) x  
 End Sub  
End Module

InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator

## 条件语句

利用条件语句，可以在运行时根据所计算的表达式有条件地执行语句。

ConditionalStatement ::= IfStatement | SelectStatement

### If...Then...Else 语句

If...Then...Else 语句是基本的条件语句。If...Then...Else 语句中的每个表达式都必须是 ‎11.19 一节中所述的布尔表达式。（注意：这并不要求表达式为布尔值类型）。如果 If语句中的布尔表达式为 true，则将执行由 If 块包含的语句。如果布尔表达式为 false，则将计算每个 ElseIf 表达式。如果某个 ElseIf 表达式的计算结果为 true，则执行相应的块。如果任何表达式的计算结果都不为 true，并且有一个 Else 块，则执行此 Else 块。在执行完一个块后，执行过程将传递到 If...Then...Else 语句的结尾。

If语句的行版本具有一组在 If 表达式为 True 时要执行的语句和一组在表达式为 False 时要执行的可选语句。例如：

Module Test  
 Sub Main()  
 Dim a As Integer = 10  
 Dim b As Integer = 20  
  
 ' Block If statement.  
 If a < b Then  
 a = b  
 Else  
 b = a  
 End If  
  
 ' Line If statement  
 If a < b Then a = b Else b = a  
 End Sub  
End Module

IfStatement ::= BlockIfStatement | LineIfThenStatement

BlockIfStatement ::=  
 If BooleanExpression [ Then ] StatementTerminator  
 [ Block ]  
 [ ElseIfStatement+ ]  
 [ ElseStatement ]  
 End If StatementTerminator

ElseIfStatement ::=  
 ElseIf BooleanExpression [ Then ] StatementTerminator  
 [ Block ]

ElseStatement ::=  
 Else StatementTerminator  
 [ Block ]

LineIfThenStatement ::=  
 If BooleanExpression Then Statements [ Else Statements ] StatementTerminator

If 语句的行版本绑定的紧密程度小于“:”，并且它的 Else 绑定到语法所允许的在词法上最接近的前方 If。例如，下面的两个版本是等效的：

If True Then \_  
If True Then Console.WriteLine("a") Else Console.WriteLine("b") \_  
Else Console.WriteLine("c") : Console.WriteLine("d")  
  
If True Then  
 If True Then  
 Console.WriteLine("a")  
 Else  
 Console.WriteLine("b")  
 End If  
 Console.WriteLine("c") : Console.WriteLine("d")  
End If

行 If 语句内允许使用除标签声明语句之外的所有语句，包括块语句。不过，它们不能将行结束符用作语句结束符（但在多行 lambda 表达式中除外）。例如：

' Allowed, since it uses : instead of LineTerminator to separate statements  
If b Then With New String("a"(0),5) : Console.WriteLine(.Length) : End With  
  
' Disallowed, since it uses a LineTerminator  
If b then With New String("a"(0), 5)  
 Console.WriteLine(.Length)  
 End With

' Allowed, since it only uses LineTerminator inside a multi-line lambda  
If b Then Call Sub()  
 Console.WriteLine("a")  
 End Sub.Invoke()

### Select Case 语句

Select Case 语句基于表达式的值执行语句。该表达式必须归类为一个值。在执行 Select Case语句时，首先会计算 Select 表达式，然后再按文中声明的顺序计算 Case 语句。计算得出 True 的第一个 Case 语句得以执行其代码块。如果没有 Case 语句的计算结果为 True，并且有一个 Case Else 语句，则将执行该代码块。在执行完一个块后，执行过程将传递到 Select 语句的结尾。

一个 Case 块的执行过程不允许“贯穿”到下一个切换节。这可以防止当 Case 终止语句被意外忽略时其他语言中通常会出现的一类 Bug。下面的示例阐释了这一点：

Module Test  
 Sub Main()  
 Dim x As Integer = 10  
  
 Select Case x  
 Case 5  
 Console.WriteLine("x = 5")  
 Case 10  
 Console.WriteLine("x = 10")  
 Case 20 - 10  
 Console.WriteLine("x = 20 - 10")  
 Case 30  
 Console.WriteLine("x = 30")  
 End Select  
 End Sub  
End Module

代码将输出：

x = 10

虽然 Case 10 和 Case 20 - 10 选择的是同一个值，但执行的是 Case 10，因为按文本顺序它位于 Case 20 - 10 之前。当到达下一个 Case 时，执行过程将在 Select 语句后继续。

Case 子句可以采用两种形式。一种形式由一个可选 Is 关键字、一个比较运算符和一个表达式构成。此表达式将转换为 Select 表达式的类型；如果此表达式不能隐式转换为 Select 表达式的类型，则将发生编译时错误。如果 Select 表达式为 E，比较运算符为 Op，且 Case 表达式为 E1，则按 E OP E1 形式计算此事例。运算符必须对两个表达式的类型均有效，否则将发生编译时错误。

另一种形式由一个可后跟关键字 To 的表达式和另一个表达式构成。这两个表达式都将转换为 Select 表达式的类型；如果任一表达式不能隐式转换为 Select 表达式的类型，则将发生编译时错误。如果 Select 表达式为 E，第一个 Case 表达式为 E1 且第二个 Case 表达式为 E2，则按 E = E1（如果没有指定 E2）或 (E >= E1) And (E <= E2) 条件计算此 Case。运算符必须对这两个表达式的类型均有效，否则将发生编译时错误。

SelectStatement ::=  
 Select [ Case ] Expression StatementTerminator  
 [ CaseStatement+ ]  
 [ CaseElseStatement ]  
 End Select StatementTerminator

CaseStatement ::=  
 Case CaseClauses StatementTerminator  
 [ Block ]

CaseClauses ::=  
 CaseClause |  
 CaseClauses Comma CaseClause

CaseClause ::=  
 [ Is [ LineTerminator ] ] ComparisonOperator [ LineTerminator ] Expression |  
 Expression [ To Expression ]

ComparisonOperator ::= = | < > | < | > | > = | < =

CaseElseStatement ::=  
 Case Else StatementTerminator  
 [ Block ]

## 循环 语句

利用循环语句，可以重复执行循环体中的语句。

每次进入一个循环体，都会在该循环体中声明所有局部变量的全新副本，这些变量都初始化为其先前的值。对循环体中变量的任何引用都将使用最近制作的副本。这段代码提供了一个示例：

Module Test  
 Sub Main()  
 Dim lambdas As New List(Of Action)  
 Dim x = 1

For i = 1 To 3  
 x = i  
 Dim y = x  
 lambdas.Add(Sub() Console.WriteLine(x & y))  
 Next

For Each lambda In lambdas  
 lambda()  
 Next  
 End Sub  
End Module

代码产生的输出如下：

31 32 33

执行循环体时，将使用变量的最新副本。例如，语句 Dim y = x 引用的是 y 的最新副本和 x 的初始副本。当创建 lambda 时，它将记住创建它时的最新变量副本。因此，每个 lambda 都使用 x 的相同共享副本，但使用 y 的不同副本。在程序结尾，当程序执行 lambda 时，所有 lambda 共同指向的 x 共享副本现在是其最终值 3。

Note that if there are no lambdas or LINQ expressions, then it's impossible to know that a fresh copy is made on loop entry. Indeed, compiler optimizations will avoid making copies in this case. Note too that it's illegal to GoTo into a loop that contains lambdas or LINQ expressions.LoopStatement ::=  
 WhileStatement |  
 DoLoopStatement |  
 ForStatement |  
 ForEachStatement

### While...End While 和 Do...Loop 语句

While 或 Do 循环语句基于布尔表达式进行循环。只要布尔表达式的计算结果为 true，While 循环语句就会进行循环；Do 循环语句可以包含更复杂的条件。可以将表达式置于 Do 关键字或 Loop 关键字的后面，但不能将表达式同时置于这两个关键字的后面。将按 ‎11.19 一节所述计算布尔表达式。（注意：这并不要求表达式为布尔值类型）。此外，不指定任何表达式也是有效的；在此情况下，循环将永远不会退出。如果将一个表达式置于 Do 的后面，则每次迭代时将先计算此表达式，然后再执行循环块。如果将一个表达式置于 Loop 的后面，则每次迭代时将先执行循环块，然后再计算此表达式。因此，将表达式置于 Loop 的后面所生成的循环数会比将表达式置于 Do 的后面所生成的循环数多一个。下面的示例演示这一行为：

Module Test  
 Sub Main()  
 Dim x As Integer  
  
 x = 3  
 Do While x = 1  
 Console.WriteLine("First loop")  
 Loop  
  
 Do  
 Console.WriteLine("Second loop")  
 Loop While x = 1  
 End Sub  
End Module

代码产生的输出如下：

Second Loop

如果是第一次循环，则会先计算条件，然后再执行循环。如果是第二次循环，则会先执行循环，然后再计算条件。条件表达式必须以 While 关键字或 Until 关键字为前缀。如果条件的计算结果为 false，则前一个关键字会中断循环；如果条件的计算结果为 true，则后一个关键字会中断循环。

注意   Until 不是保留字。

WhileStatement ::=  
 While BooleanExpression StatementTerminator  
 [ Block ]  
 End While StatementTerminator

DoLoopStatement ::= DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=  
 Do [ WhileOrUntil BooleanExpression ] StatementTerminator  
 [ Block ]  
 Loop StatementTerminator

DoBottomLoopStatement ::=  
 Do StatementTerminator  
 [ Block ]  
 Loop WhileOrUntil BooleanExpression StatementTerminator

WhileOrUntil ::= While | Until

### For...Next 语句

1. For...Next 语句基于一组边界进行循环。For 语句会指定一个循环控制变量、一个下限表达式、一个上限表达式和一个可选步长值表达式。循环控制变量既可通过一个标识符后跟可选的 As 子句来指定，也可通过表达式来指定。根据以下规则，循环控制变量引用该 For...Next 语句专用的新局部变量，或者引用预先存在的变量，或者引用表达式。如果循环控制变量是一个带有 As 子句的标识符，则该标识符定义具有在 As 子句中所指定的类型的新局部变量，其范围为整个 For 循环。
2. 如果循环控制变量是一个不带 As 子句的标识符，则将首先使用简单名称解析规则（请参见 ‎11.4.4 节）来解析该标识符，除非此标识符的出现本身不会引起创建隐式局部变量（10.2.1 节）。
   1. 如果此解析成功，并且结果归类为变量，则该循环控制变量就是该预先存在的变量。
   2. 如果解析失败，或者，如果解析成功且结果归类为一个类型，则：
      1. 如果正在使用局部变量类型推断，则该标识符定义新的局部变量，其类型将通过界限和步骤表达式来推断出来，其范围为整个 For 循环；
      2. 如果没有使用局部变量类型推断，而是使用隐式局部声明，则会创建一个隐式局部变量，其范围为整个方法（10.2.1 节），并且该循环控制变量引用此预先存在的变量；
      3. 如果既不使用局部变量类型推断，也不使用隐式局部声明，则出现错误。
   3. 如果解析成功，但结果既未归类为类型，也未归类为变量，则出现错误。
3. 如果循环控制变量是一个表达式，则必须将此表达式归类为一个变量。

循环控制变量不能由另一个封闭 For...Next 语句使用。For 语句的循环控制变量类型决定了迭代的类型，它必须是下列类型之一：

1. Byte, SByte, UShort, Short, UInteger, Integer, ULong, Long, Decimal, Single, Double
2. 一个枚举类型
3. Object
4. 一个具有以下运算符的类型 T，其中 B 是一个可在布尔表达式中使用的类型：

Public Shared Operator >= (op1 As T, op2 As T) As B  
Public Shared Operator <= (op1 As T, op2 As T) As B  
Public Shared Operator - (op1 As T, op2 As T) As T  
Public Shared Operator + (op1 As T, op2 As T) As T

界限表达式和步长表达式必须能够隐式转换为循环控制变量的类型且必须归类为值。在编译时，通过在下限、上限和步长表达式类型之间选择最宽的类型来推断循环控制变量的类型。如果两个类型之间不存在扩大转换，则将发生编译时错误。

在运行时，如果循环控制变量的类型为 Object，则按照编译时的相同方式推断迭代的类型，但以下两种情况例外。第一种例外情况是，如果界限表达式和步长表达式都属于整型且没有最宽类型，则将推断出包含所有三种类型的最宽类型。第二种例外情况是，如果推断出的循环控制变量的类型为 String，则将改为推断 Double。如果在运行时无法确定循环控制类型，或者无法将任何表达式转换为循环控制类型，则将发生 System.InvalidCastException。若在循环的开头选择了一个循环控制类型，则在整个迭代中将使用该类型，而不考虑对循环控制变量中的值所做的更改。

For 语句必须通过匹配的 Next 语句结束。不带变量的 Next 语句将与最内部的开始 For 语句匹配，而带一个或多个循环控制变量的 Next 语句则按从左至右的方式与对应于每个变量的 For 循环匹配。如果一个变量此时所匹配的 For 循环不是嵌套最深的循环，则将发生编译时错误。

在循环的开头，将按照文本顺序计算这三个表达式，并会将下限表达式赋给循环控制变量。如果省略步长值，则文本 1 将隐式转换为循环控制变量的类型。只会在循环的开头计算这三个表达式。

在每个循环的开头，将比较控制变量以查看此变量是大于结束点（如果步长表达式为正）还是小于结束点（如果步长表达式为负）。如果比较结果为真，则 For 循环将终止；否则将执行循环块。如果循环控制变量不是基元类型，则比较运算符由表达式 step >= step – step 是 true 还是 false 来确定。在 Next 语句中，会将控制变量与步长值相加，并且执行过程会返回到循环的顶部。

请注意，*不* 是在循环块的每个迭代上都会创建循环控制变量的新副本。就这一点而言，For 语句不同于 For Each（10.9.3 节）。

从循环的外部分支到 For 循环是无效的。

ForStatement ::=  
 For LoopControlVariable Equals Expression To Expression [ Step Expression ] StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

LoopControlVariable ::=  
 Identifier [ IdentifierModifiers As TypeName ] |  
 Expression

NextExpressionList ::=  
 Expression |  
 NextExpressionList Comma Expression

### For Each...Next 语句

For Each...Next 语句基于表达式中的元素进行循环。For Each 语句指定一个循环控制变量和一个枚举器表达式。循环控制变量既可通过一个标识符后跟可选的 As 子句来指定，也可通过表达式来指定。与 For...Next 语句 (10.9.2) 的规则相同，该循环控制变量引用此 For...Next 语句专用的新局部变量，或者引用预先存在的变量，或者引用表达式。

枚举器表达式必须归类为一个值，并且其类型必须为集合类型或 Object。如果枚举器表达式的类型为 Object，则所有处理都将推迟到运行时。否则，必须存在从集合的元素类型到循环控制变量的类型的转换。

循环控制变量不能由另一个封闭 For Each 语句使用。For Each 语句必须通过匹配的 Next 语句来结束。不带循环控制变量的 Next 语句与最内部的开始 For Each 语句匹配。带一个或多个循环控制变量的 Next 语句将按照从左至右的方式与具有相同循环控制变量的 For Each 循环匹配。如果一个变量此时所匹配的 For Each 循环不是嵌套最深的循环，则将发生编译时错误。

在以下情况下，称类型 C 为集合类型：

1. 如果满足以下所有条件：
   1. C 包含一个可访问实例以及带有签名 GetEnumerator() 且返回类型 E 的共享或扩展方法。
   2. E 包含一个可访问实例以及带有签名 MoveNext() 和返回类型 Boolean 的共享或扩展方法。
   3. E 包含一个具有 getter 的名为 Current 的可访问实例属性。此属性的类型为集合类型的元素类型。
2. 此类型实现接口 System.Collections.Generic.IEnumerable(Of T)，在此情况下，该集合的元素类型被视为 T。
3. 此类型实现接口 System.Collections.IEnumerable，在此情况下，该集合的元素类型被视为 Object。

以下是可枚举的类的示例：

Public Class IntegerCollection  
 Private integers(10) As Integer  
  
 Public Class IntegerCollectionEnumerator  
 Private collection As IntegerCollection  
 Private index As Integer = -1  
  
 Friend Sub New(c As IntegerCollection)  
 collection = c  
 End Sub  
  
 Public Function MoveNext() As Boolean  
 index += 1  
  
 Return index <= 10  
 End Function  
  
 Public ReadOnly Property Current As Integer  
 Get  
 If index < 0 OrElse index > 10 Then  
 Throw New System.InvalidOperationException()  
 End If  
  
 Return collection.integers(index)  
 End Get  
 End Property  
 End Class  
  
 Public Sub New()  
 Dim i As Integer  
  
 For i = 0 To 10  
 integers(i) = I  
 Next i  
 End Sub  
  
 Public Function GetEnumerator() As IntegerCollectionEnumerator  
 Return New IntegerCollectionEnumerator(Me)  
 End Function  
End Class

在循环开始之前计算枚举数表达式。如果表达式的类型不符合设计模式，则将该表达式强制转换为 System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable(Of T)。如果表达式类型实现泛型接口，则在编译时优先选择泛型接口，而在运行时优先选择非泛型接口。如果表达式类型多次实现泛型接口，则语句是不明确的，并将发生编译时错误。

批注

在后期绑定示例中优先选择非泛型接口，这是因为选择泛型接口将意味着对接口方法的所有调用将涉及类型参数。由于在运行时无法获知匹配类型，因此必须使用后期绑定调用来进行所有此类调用。此类调用的速度比调用非泛型接口的速度要慢一些，因为可以使用编译时调用来调用非泛型接口。

对生成的值调用 GetEnumerator，并将函数的返回值存储在一个临时变量中。然后，在每个迭代的开始处对该临时变量调用 MoveNext。如果返回 False，则循环终止。否则，该循环的每个迭代都按如下方式执行：

1. 如果循环控制变量确定出新的局部变量（而不是预先存在的变量），则为此局部变量创建一个全新副本。对于当前迭代，循环块内的所有引用都将引用此副本。
2. 将会检索 Current 属性，并将其强制为该循环控制变量的类型（无论此转换是隐式转换还是显式转换），然后将其指派给该循环控制变量。
3. 循环块执行。

批注

语言版本 10.0 和 11.0 之间在行为方面略有不同。在 11.0 以前的版本中，*不会* 为循环的每个迭代都创建全新迭代变量。只有在 lambda 或 LINQ 表达式捕获到迭代变量并随后在循环结束后调用时，才能观察到此差别。

Dim lambdas As New List(Of Action)  
For Each x In {1,2,3}  
 lambdas.Add(Sub() Console.WriteLine(x)  
Next  
lambdas(0).Invoke()  
lambdas(1).Invoke()  
lambdas(2).Invoke()

在 Visual Basic 10.0 之前，这会在编译时产生警告，并打印“3”三次。这是因为，该循环的所有迭代仅共享一个变量“x”，并且所有三个 lambda 都捕获到相同的“x”，到这些 lambda 执行时，该变量随后会保持数字 3。

从 Visual Basic 11.0 开始，它会输出“1、2、3”。这是因为，每个 lambda 都会捕获不同的变量“x”。

批注

迭代的当前元素将转换为循环控制变量的类型，即使此转换是显式的也是如此，这是因为语句中没有用来引入转换运算符的合适位置。在处理现在已过时的类型 System.Collections.ArrayList 时，这会变得十分麻烦，因为它的元素类型为 Object。这将需要在许多循环中进行强制转换，有时这样做并不理想。

有趣的是，泛型支持创建强类型的集合 System.Collections.Generic.List(Of T)，这可能会让我们重新考虑此设计点，但出于兼容性原因，目前无法对此进行更改。

当到达 Next 语句时，执行过程会返回到循环的顶部。如果在 Next 关键字后指定了变量，则该变量必须与 For Each 后的第一个变量相同。例如，考虑以下代码：

Module Test  
 Sub Main()  
 Dim i As Integer  
 Dim c As IntegerCollection = New IntegerCollection()  
  
 For Each i In c  
 Console.WriteLine(i)  
 Next i  
 End Sub  
End Module

此代码与以下代码等效：

Module Test  
 Sub Main()  
 Dim i As Integer  
 Dim c As IntegerCollection = New IntegerCollection()  
  
 Dim e As IntegerCollection.IntegerCollectionEnumerator  
  
 e = c.GetEnumerator()  
 While e.MoveNext()  
 i = e.Current  
  
 Console.WriteLine(i)  
 End While  
 End Sub  
End Module

如果枚举器的类型 E 实现 System.IDisposable，则在退出循环时，将通过调用 Dispose 方法来释放该枚举器。这将确保释放由枚举器占用的资源。如果包含 For Each 语句的方法不使用非结构化错误处理，则使用在 Finally 中调用的 Dispose 方法，将 For Each 语句包装在 Try 语句中以确保清除。

注意   System.Array 类型是一个集合类型，由于所有数组类型都派生自 System.Array，因此 For Each 语句中允许有任何数组类型表达式。对于一维数组，For Each 语句将按照递增索引顺序（以索引 0 开头，并以将索引长度减 1 得到的数结尾）枚举数组元素。对于多维数组，首先会增大最右侧维度的索引。

例如，下面的代码将输出 1 2 3 4：

Module Test  
 Sub Main()  
 Dim x(,) As Integer = { { 1, 2 }, { 3, 4 } }  
 Dim i As Integer  
  
 For Each i In x  
 Console.Write(i & " ")  
 Next i  
 End Sub  
End Module

从 For Each 语句块的外部分支到该块是无效的。

ForEachStatement ::=  
 For Each LoopControlVariable In [ LineTerminator ] Expression StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

## 异常处理语句

Visual Basic 支持结构化的异常处理和非结构化的异常处理。一个方法只能使用一种样式的异常处理，不过可以在结构化的异常处理中使用 Error 语句。如果一个方法同时使用两种样式的异常处理，则将发生编译时错误。

ErrorHandlingStatement ::=  
 StructuredErrorStatement |  
 UnstructuredErrorStatement

### 结构化的异常处理语句

结构化的异常处理是一类用于处理错误的方法，它通过声明在其中处理特定异常的显式块来处理错误。结构化的异常处理是通过 Try 语句来执行的。

例如：

Module Test  
 Sub ThrowException()  
 Throw New Exception()  
 End Sub  
  
 Sub Main()  
 Try  
 ThrowException()  
 Catch e As Exception  
 Console.WriteLine("Caught exception!")  
 Finally  
 Console.WriteLine("Exiting try.")  
 End Try  
 End Sub  
End Module

Try 语句由三类块构成：try 块、catch 块和 finally 块。try 块是包含要执行的语句的语句块。catch 块是处理异常的语句块。finally 块是包含在退出 Try 语句时要运行的语句（不管是否已发生并处理异常）的语句块。Try 语句只能包含一个 try 块和一个 finally 块，并且必须包含至少一个 catch 块或 finally 块。将执行过程显式转移到 try 块中是无效的，除非是从同一语句中的 catch 块转移执行过程。

StructuredErrorStatement ::=  
 ThrowStatement |  
 TryStatement

TryStatement ::=  
 Try StatementTerminator  
 [ Block ]  
 [ CatchStatement+ ]  
 [ FinallyStatement ]  
 End Try StatementTerminator

#### Finally 块

当执行离开 Try 语句的任何部分时，总是会执行 Finally 块。执行 Finally 块时无需进行显式操作；当执行过程离开 Try 语句时，系统将自动执行 Finally 块，然后将执行过程转移到其预期目标。无论执行通过以下何种方式离开 Try 语句，都会执行 Finally 块：通过 Try 块的结尾、通过 Catch 块的结尾，通过 Exit Try 语句，通过 GoTo 语句，或通过不处理引发的异常。

请注意，异步方法中的 Await 表达式和迭代器方法中的 Yield 语句可能导致控制流在异步或迭代器方法实例中挂起，然后在其他某个方法实例中继续。不过，这只是执行挂起，不涉及退出相应的异步方法或迭代器方法实例，因此不会导致执行 Finally 块。

将执行过程显式转移到 Finally 块中是无效的；此外，将执行过程转移到 Finally 块的外部也是无效的（除非通过异常）。

FinallyStatement ::=  
 Finally StatementTerminator  
 [ Block ]

#### Catch 块

如果在处理 Try 块时发生异常，则会按文中顺序检查每个 Catch 语句，以确定它是否处理异常。Catch 子句中指定的标识符表示已引发的异常。如果标识符包含一个 As 子句，则将标识符视为是在 Catch 块的局部声明空间中声明的。否则，标识符必须是包含块中已定义的局部变量（而非静态变量）。

不带标识符的 Catch 子句将会捕获派生自 System.Exception 的所有异常。带标识符的 Catch 子句只会捕获其类型与标识符的类型相同或派生自标识符的类型的异常。类型必须为 System.Exception，或派生自 System.Exception 的类型。在捕获到派生自 System.Exception 的异常时，会将对异常对象的引用存储到由函数 Microsoft.VisualBasic.Information.Err 返回的对象中。

带 When 子句的 Catch 子句只在表达式的计算结果为 True 时捕获异常；表达式的类型必须为 ‎11.19 一节所述的布尔表达式。When 子句只有在检查异常的类型之后才会应用，表达式可以引用表示异常的标识符，如以下示例所示：

Module Test  
 Sub Main()  
 Dim i As Integer = 5  
  
 Try  
 Throw New ArgumentException()  
 Catch e As OverflowException When i = 5  
 Console.WriteLine("First handler")  
 Catch e As ArgumentException When i = 4  
 Console.WriteLine("Second handler")  
 Catch When i = 5  
 Console.WriteLine("Third handler")  
 End Try  
  
 End Sub  
End Module

此示例输出：

Third handler

如果 Catch 子句处理异常，则执行将转移到 Catch 块。在 Catch 块的结尾，执行过程将转移到 Try 语句后面的第一个语句。Try 语句不会处理 Catch 块中引发的任何异常。如果没有任何 Catch 子句处理异常，则执行过程将转移到由系统确定的位置。

将执行过程显式转移到 Catch 块中是无效的。

批注

通常会在引发异常之前对 When 子句中的筛选器进行计算。例如，下面的代码将输出“Filter, Finally, Catch”。

Sub Main()  
 Try  
 Foo()  
 Catch ex As Exception When F()  
 Console.WriteLine("Catch")  
 End Try  
 End Sub

Sub Foo()  
 Try  
 Throw New Exception  
 Finally  
 Console.WriteLine("Finally")  
 End Try  
 End Sub

Function F() As Boolean  
 Console.WriteLine("Filter")  
 Return True  
 End Function

不过，异步和迭代器方法会导致在执行任何外部筛选器之前执行其内部的所有 finally 块。例如，如果以上代码具有“Async Sub Foo()”，则输出将是“Finally, Filter, Catch”。

CatchStatement ::=  
 Catch [ Identifier [ As NonArrayTypeName ] ] [ When BooleanExpression ] StatementTerminator  
 [ Block ]

#### Throw 语句

Throw 语句会引发异常，此异常由派生自 System.Exception 类型的实例来表示。如果表达式未归类为一个值或不是派生自 System.Exception 的类型，则将发生编译时错误。如果在运行时此表达式的计算结果为 null 值，则引发 System.NullReferenceException 异常。

只要没有干涉的 finally 块，Throw 语句就可以省略 Try 语句的 catch 块中的表达式。在此情况下，语句将重新引发 catch 块中当前正在处理的异常。例如：

Sub Test(x As Integer)  
 Try  
 Throw New Exception()  
 Catch  
 If x = 0 Then  
 Throw ' OK, rethrows exception from above.  
 Else  
 Try  
 If x = 1 Then  
 Throw ' OK, rethrows exception from above.  
 End If  
 Finally  
 Throw ' Invalid, inside of a Finally.  
 End Try  
 End If  
 End Try  
End Sub

ThrowStatement ::= Throw [ Expression ] StatementTerminator

### 非结构化的异常处理语句

非结构化的异常处理是一类用于处理错误的方法，它通过指示语句在发生异常时进行分支来处理错误。非结构化异常处理是使用三种语句实现的：Error 语句、On Error 语句和 Resume 语句。例如：

Module Test  
 Sub ThrowException()  
 Error 5  
 End Sub  
  
 Sub Main()  
 On Error GoTo GotException  
  
 ThrowException()  
 Exit Sub  
  
GotException:  
 Console.WriteLine("Caught exception!")  
 Resume Next  
 End Sub  
End Module

当方法使用非结构化的异常处理时，将为捕获所有异常的整个方法创建一个结构化异常处理程序。（请注意，在构造函数中，此处理程序不会扩展到构造函数开头处对 New 的调用。）之后，方法会跟踪最新的异常处理程序位置和已引发的最新异常。在进入该方法后，异常处理程序位置和异常都设置为 Nothing。当在使用非结构化异常处理的方法中引发异常时，会将对异常对象的引用存储到由函数 Microsoft.VisualBasic.Information.Err 返回的对象中。

迭代器或异步方法中不允许使用非结构化错误处理语句。

UnstructuredErrorStatement ::=  
 ErrorStatement |  
 OnErrorStatement |  
 ResumeStatement

#### Error 语句

Error 语句会引发包含 Visual Basic 6 异常编号的 System.Exception 异常。表达式必须归类为一个值，并且其类型必须能够隐式转换为 Integer。

ErrorStatement ::= Error Expression StatementTerminator

#### On Error 语句

On Error 语句会修改最近的异常处理状态。可按以下四种方式之一使用此语句：

On Error GoTo -1 会将最近异常重置为 Nothing。

On Error GoTo 0 会将最近异常处理程序位置重置为 Nothing。

On Error GoTo LabelName 可创建标签作为最新的异常处理程序位置。此语句不能在包含 lambda 或查询表达式的方法中使用。

On Error Resume Next 会建立 Resume Next 行为以作为最近异常处理程序位置。

OnErrorStatement ::= On Error ErrorClause StatementTerminator

ErrorClause ::=  
 GoTo - 1 |  
 GoTo 0 |  
 GoToStatement |  
 Resume Next

#### Resume 语句

Resume 语句将执行过程返回到导致最新异常的语句。如果指定 Next 修饰符，则执行过程将返回到已在导致最新异常的语句后执行的语句。如果指定标签名，则执行过程将返回到标签。

由于 SyncLock 语句包含一个隐式结构化错误处理块，因此 Resume 和 Resume Next 具有针对 SyncLock 语句中发生的异常的特殊行为。Resume 会将执行返回到 SyncLock 语句的开头，而 Resume Next 会将执行返回到紧跟 SyncLock 语句的下一个语句。例如，考虑以下代码：

Class LockClass  
End Class  
  
Module Test  
 Sub Main()  
 Dim FirstTime As Boolean = True  
 Dim Lock As LockClass = New LockClass()  
  
 On Error GoTo Handler  
  
 SyncLock Lock  
 Console.WriteLine("Before exception")  
 Throw New Exception()  
 Console.WriteLine("After exception")  
 End SyncLock  
  
 Console.WriteLine("After SyncLock")  
 Exit Sub  
  
Handler:  
 If FirstTime Then  
 FirstTime = False  
 Resume  
 Else  
 Resume Next  
 End If  
 End Sub  
End Module

此代码将输出以下结果。

Before exception  
Before exception  
After SyncLock

在第一次通过 SyncLock 语句时，Resume 会将执行返回到 SyncLock 语句的开头。在第二次通过 SyncLock 语句时，Resume Next 会将执行返回到 SyncLock 语句的结尾。SyncLock 语句中不允许有 Resume 和 Resume Next。

在所有情况下，在执行 Resume 语句时，最新异常将设置为 Nothing。如果在执行 Resume 语句时没有发生最新异常，则该语句将引发一个 System.Exception 异常，其中包含 Visual Basic 错误号 20（无错误继续执行）。

ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator

ResumeClause ::= Next | LabelName

## 分支语句

分支语句可修改方法中的执行流。分支语句分为以下六类：

1. GoTo 语句促使执行过程转移到方法中的指定标签。如果在 lambda 或 LINQ 表达式中捕获到 Try、Using、SyncLock、With、For 或 For Each 块的本地变量，则不允许 GoTo 到该块中。
2. Exit 语句将执行过程转移到紧跟在指定类型的直接包含块语句结尾的下一个语句。如果该块为方法块，则控制流将按照 10.1 节、10.2 节、10.3 节所述退出该方法。如果 Exit 语句未包含在语句中指定类型的块中，则将发生编译时错误。
3. Continue 语句将执行过程转移到指定类型的直接包含块循环语句的结尾。如果 Continue 语句未包含在语句中指定类型的块中，则将发生编译时错误。
4. Stop 语句会导致发生调试器异常。
5. End 语句可终止程序。终结器在程序关闭前运行，但不会执行任何当前正在执行的 Try 语句中的 finally 块。此语句不能在不可执行的程序（如 DLL）中使用。
6. 不含表达式的 Return 语句等效于 Exit Sub 或 Exit Function 语句。含有表达式的 Return 语句只允许在作为函数的常规方法中使用，或者在作为函数且部分 T 的返回类型为 Task(Of T) 的异步方法中使用。该表达式必须归类为一个值，该值可隐式转换为*函数返回变量*（对于常规方法）或隐式转换为*任务返回变量*（对于异步方法）。其行为是计算它的表达式，然后将其存储在返回变量中，然后执行隐式 Exit Function 语句。

BranchStatement ::=  
 GoToStatement |  
 ExitStatement |  
 ContinueStatement |  
 StopStatement |  
 EndStatement |  
 ReturnStatement

GoToStatement ::= GoTo LabelName StatementTerminator

ExitStatement ::= Exit ExitKind StatementTerminator

ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try

ContinueStatement ::= Continue ContinueKind StatementTerminator

ContinueKind ::= Do | For | While

StopStatement ::= Stop StatementTerminator

EndStatement ::= End StatementTerminator

ReturnStatement ::= Return [ Expression ] StatementTerminator

## 数组处理语句

以下两种语句可简化数组的使用：ReDim 语句和 Erase 语句。

ArrayHandlingStatement ::=  
 RedimStatement |  
 EraseStatement

### ReDim 语句

ReDim 语句可实例化新数组。此语句中的每个子句都必须归类为一个变量或属性访问（其类型是数组类型或 Object）并后跟一个数组边界列表。边界的数量必须与变量的类型保持一致；Object 允许有任意数量的边界。在运行时，将使用指定界限为每个表达式实例化一个数组（按从左至右的顺序），然后将其赋给变量或属性。如果变量类型为 Object，则维数为指定的维数，且数组元素类型为 Object。如果给定维数在运行时与变量或属性不兼容，则将发生编译时错误。例如：

Module Test  
 Sub Main()  
 Dim o As Object  
 Dim b() As Byte  
 Dim i(,) As Integer  
  
 ' The next two statements are equivalent.  
 ReDim o(10,30)  
 o = New Object(10, 30) {}  
  
 ' The next two statements are equivalent.  
 ReDim b(10)  
 b = New Byte(10) {}  
  
 ' Error: Incorrect number of dimensions.  
 ReDim i(10, 30, 40)  
 End Sub  
End Module

如果指定 Preserve 关键字，则表达式也必须归类为一个值，并且每个维度的新大小必须与现有数组的大小相同（最右侧的维度除外）。现有数组中的值将复制到新的数组中：如果新的数组较小，则丢弃现有值；如果新的数组较大，则额外元素将实例化为数组的元素类型的默认值。例如，考虑以下代码：

Module Test  
 Sub Main()  
 Dim x(5, 5) As Integer  
  
 x(3, 3) = 3  
  
 ReDim Preserve x(5, 6)  
 Console.WriteLine(x(3, 3) & ", " & x(3, 6))  
 End Sub  
End Module

此代码将输出以下结果：

3, 0

如果现有数组引用在运行时是一个 null 值，则不给出任何错误。如果维度的大小发生更改（最右侧的维度除外），则将引发 System.ArrayTypeMismatchException 异常。

注意   Preserve 不是保留字。

RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator

RedimClauses ::=  
 RedimClause |  
 RedimClauses Comma RedimClause

RedimClause ::= Expression ArraySizeInitializationModifier

### Erase 语句

Erase 语句可将语句中指定的每个数组变量或属性设置为 Nothing。此语句中的每个表达式都必须归类为一个变量或属性访问（其类型是数组类型或 Object）。例如：

Module Test  
 Sub Main()  
 Dim x() As Integer = New Integer(5) {}  
  
 ' The following two statements are equivalent.  
 Erase x  
 x = Nothing  
 End Sub  
End Module

EraseStatement ::= Erase EraseExpressions StatementTerminator

EraseExpressions ::=  
 Expression |  
 EraseExpressions Comma Expression

## Using 语句

若在执行收集时未找到对实例的任何有效引用，则垃圾回收器将自动释放类型的实例。如果一个类型占据了特别有价值的短缺资源（如数据库连接或文件句柄），则可以清除不再使用的类型的特定实例，而无需等到下一次垃圾回收时再进行清除。若要提供一种用于在垃圾回收之前释放资源的简便方法，类型可以实现 System.IDisposable 接口。实现该接口的类型将公开一个 Dispose 方法，然后可以通过调用此方法来强制立即释放宝贵的资源，如下所示：

Module Test  
 Sub Main()  
 Dim x As DBConnection = New DBConnection("...")  
  
 ' Do some work  
 ...  
  
 x.Dispose() ' Free the connection  
 End Sub  
End Module

Using 语句会自动完成以下过程：获取资源，执行一组语句，然后释放资源。此语句可采用两种格式：在一种格式中，资源是一个局部变量，此变量被声明为语句的一部分并被视为常规局部变量声明语句；在另一种格式中，资源是表达式的结果。

如果资源是局部变量声明语句，则局部变量声明的类型必须为能够隐式转换为 System.IDisposable 的类型。声明的局部变量是只读的（其作用范围为 Using 语句块）且必须包含初始值设定项。如果资源是一个表达式的结果，则该表达式必须归类为一个值，且其类型必须为能够隐式转换为 System.IDisposable 的类型。在此语句的开头，仅计算表达式一次。

Using 块由 Try 语句隐式包含，该语句的 finally 块会对该资源调用 IDisposable.Dispose 方法。这将确保资源被释放（即使引发异常时也是如此）。因此，从 Using 块的外部分支到该块是无效的，Resume 和 Resume Next 会将 Using 块视为一个语句。如果资源为 Nothing，则不会调用 Dispose。因此，示例：

Using f As C = New C()  
 ...  
End Using

等效于：

Dim f As C = New C()  
Try  
 ...  
Finally  
 If f IsNot Nothing Then  
 f.Dispose()  
 End If  
End Try

带有局部变量声明语句的 Using 语句一次可以获取多个资源，该语句与嵌套的 Using 语句等效。例如，以下形式的 Using 语句：

Using r1 As R = New R(), r2 As R = New R()  
 r1.F()  
 r2.F()  
End Using

等效于：

Using r1 As R = New R()  
 Using r2 As R = New R()  
 r1.F()  
 r2.F()  
 End Using  
End Using

UsingStatement ::=  
 Using UsingResources StatementTerminator  
 [ Block ]  
 End Using StatementTerminator

UsingResources ::= VariableDeclarators | Expression

## Await 语句

await 语句的语法与 await 运算符表达式 (11.25) 相同，只允许包含在也允许 await 表达式的方法中，并且行为也与 await 运算符表达式相同。

不过，可以将该语句归类为值或 void。通过计算 await 运算符表达式得到的任何值都将被丢弃。

AwaitStatement ::= AwaitOperatorExpression StatementTerminator

## yield 语句

Yield 语句与 10.1.2 节中描述的迭代器方法相关。

如果包含 Yield 的直接封闭方法或 lambda 表达式具有 Iterator 修饰符，并且 Yield 出现在该 Iterator 修饰符后面，则 Yield 是一个保留字；它在其他位置不是保留字。Yield 在预处理器指令中也不是保留字。只允许将 yield 语句在方法或 lambda 表达式体中作为保留字使用。在直接封闭方法或 lambda 中，Catch 或 Finally 块体以及 SyncLock 语句体中不能出现 yield 语句。

yield 语句采取单一表达式的形式，必须归类为一个值，并且其类型可隐式转换为其封闭迭代器方法的*迭代器当前变量*（10.1.2 节）的类型。

只有在对迭代器对象调用 MoveNext 方法时，控制流才会到达 Yield 语句。（这是因为，只有在对迭代器对象调用 MoveNext 或 Dispose 方法时，迭代器方法才会执行其语句；并且，Dispose 方法只在不允许有 Yield 的 Finally 块中执行代码）。

执行 Yield 语句时，将对其表达式进行计算，并将结果存储在与该迭代器对象关联的迭代器方法实例的*迭代器当前变量* 中。值 True 将返回到 MoveNext 的调用方，并且，此实例的控制点会停止前进，直到对迭代器对象下一次调用 MoveNext。

YieldStatement ::= Yield *Expression* StatementTerminator

# 表达式

表达式由一系列运算符和操作数构成，用于指定值的计算或用于表示变量或常量。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。

Expression ::=  
 SimpleExpression |  
 TypeExpression |  
 MemberAccessExpression |  
 DictionaryAccessExpression |  
 InvocationExpression |  
 IndexExpression |  
 NewExpression |  
 CastExpression |  
 OperatorExpression |  
 ConditionalExpression |  
 LambdaExpression |  
 QueryExpression |  
 XMLLiteralExpression |  
 XMLMemberAccessExpression

## 表达式的分类

每个表达式均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成员访问的左侧。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成员访问的左侧。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。它是一组根据同一名称进行重载的方法。方法组可以有关联的目标表达式和关联的类型参数列表。

方法指针。它表示方法的位置。方法指针可以有关联的目标表达式和关联的类型参数列表。

lambda 方法。它是一个匿名方法。

属性组。它是一组根据同一名称进行重载的属性。属性组可以有关联的目标表达式。

属性访问。每个属性访问都有关联的类型，即该属性的类型。属性访问可以有关联的目标表达式。

后期绑定访问。它表示推迟到运行时的方法或属性访问。后期绑定访问可以有关联的目标表达式和关联的类型参数列表。后期绑定访问的类型始终为 Object。

事件访问。每个事件访问都有关联的类型，即该事件的类型。事件访问可以有关联的目标表达式。事件访问可以作为 RaiseEvent、AddHandler 和 RemoveHandler 语句的第一个实参出现。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。

数组文本。它表示类型尚未确定的数组的初始值。

Void。当表达式是对子例程的调用或者是没有结果的 await 运算符表达式时，就会出现这种情况。归类为 void 的表达式仅在调用语句或 await 语句的上下文中有效。

默认值。仅文本 Nothing 会产生此分类。

表达式的最终结果通常是一个值或变量，而表达式的其他类别会用作只能在特定上下文中使用的中间值。

请注意，可以在要求表达式类型具有某些特征（如为引用类型、值类型、派生自某个类型等）的语句和表达式中使用其类型为一个类型参数的表达式，但前提是对该类型参数施加的约束满足这些特征。

### 表达式的重新分类

一般来说，在所需分类不同于表达式的分类的上下文中使用该表达式时，将发生编译时错误 — 例如，在尝试将一个值赋给文本时。但在许多情况下，可以通过重新分类这一过程来更改表达式的分类。

如果重新分类成功完成，则该重新分类要定性为扩大转化或收缩转换。除非另行说明，否则此列表中的所有重新分类均为扩大转换。

可以对以下类型的表达式进行重新分类：

1. 可以将变量重新归类为一个值。获取存储在变量中的值。
2. 可以将方法组重新归类为一个值。方法组表达式被解释为带有关联目标表达式和类型形参列表以及空括号的调用表达式（也就是说，f 解释为 f()，而 f(Of Integer) 解释为 f(Of Integer)()）。此重新分类可能会导致表达式进一步被重新归类为 void。
3. 可以将方法指针重新归类为一个值。此重新分类只能在其目标类型已知的转换上下文中出现。方法指针表达式将解释为带有关联类型参数列表的适当类型的委托实例化表达式的参数。例如：

Delegate Sub D(i As Integer)  
  
Module Test  
 Sub F(i As Integer)  
 End Sub  
  
 Sub Main()  
 Dim del As D  
  
 ' The next two lines are equivalent.  
 del = AddressOf F  
 del = New D(AddressOf F)  
 End Sub  
End Module

1. 可以将 lambda 方法重新归类为一个值。如果此种重新分类在其目标类型已知的转换上下文中出现，则会出现以下两种重新分类之一：
2. 如果目标类型是委托类型，则 lambda 方法将解释为适当类型的委托构造表达式的参数。
3. 如果目标类型是 System.Linq.Expressions.Expression(Of T) 且 T 为委托类型，则对 lambda 方法的解释是将其视为在 T 的委托构造表达式中使用然后转换为表达式树。

如果委托没有 ByRef 形参，则只能将异步或迭代器 lambda 方法解释为委托构造表达式的实参。

如果从该委托的任何形参类型向对应的 lambda 形参类型的转换是收缩转换，则将重新分类判断为收缩转换；否则，重新分类就是扩大转换。

批注

lambda 方法和表达式树之间的准确转换在各个版本的编译器之间可能不是固定的，并且超出了本规范的讨论范围。对于 Microsoft Visual Basic 11.0，所有 lambda 表达式都可以转换为表达式树，但具有以下限制：

1. 只有不带 ByRef 参数的单行 lambda 表达式才可以转换为表达式树。在单行 Sub lambda 中，只有调用语句才可以转换为表达式树。
2. 如果用前面的字段初始值设定项初始化后面的字段初始值设定项（例如，New With {.a=1, .b=.a}），则匿名类型表达式不能转换为表达式树。
3. 如果在某个字段初始值设定项中使用正在初始化的当前对象的成员（例如 New C1 With {.a=1, .b=.Method1()}），则对象初始值设定项表达式不能转换为表达式树。
4. 多维数组创建表达式只有在显式声明自身元素类型的情况下，才能转换为表达式树。
5. 无法将后期绑定表达式转换为表达式树。
6. 当变量或字段通过引用传递到调用表达式但其类型不与 ByRef 形参完全相同时，或者通过引用传递属性时，一般的 VB 语义是通过引用来传递实参的副本，然后将其最终值复制回变量或字段或属性中。在表达式树中，不会发生反向复制。

所有这些限制也适用于嵌套 lambda 表达式。

如果目标类型未知，则 lambda 方法将被解释为一个匿名委托类型（带有与 lambda 方法相同的签名）的委托实例化表达式的参数。如果使用严格语义并省略任一参数的类型，则将发生编译时错误；否则，Object 将替换任何缺少的参数类型。例如：

Module Test  
 Sub Main()  
 ' Type of x will be equivalent to Func(Of Object, Object, Object)  
 Dim x = Function(a, b) a + b  
  
 ' Type of y will be equivalent to Action(Of Object, Object)  
 Dim y = Sub(a, b) Console.WriteLine(a + b)  
 End Sub  
End Module

1. 可以将属性组重新归类为一个属性访问。属性组表达式将被解释为一个带有空括号的索引表达式（也就是说，f 将被解释为 f()）。
2. 可以将属性访问重新归类为一个值。属性访问表达式将被解释为属性的 Get 访问器的调用表达式。如果属性没有 getter，则将发生编译时错误。
3. 可以将后期绑定访问重新归类为后期绑定方法或后期绑定属性。在可以将后期绑定访问重新归类为方法访问和属性访问的情况下，优先选择将后期绑定访问重新归类为属性访问。
4. 可以将后期绑定访问重新归类为一个值。
5. 可以将数组文本重新归类为值。将按如下方式确定值的类型：
6. 如果在目标类型已知且目标类型为数组类型的转换上下文中发生重新分类，则将数组文本重新分类为类型 T() 的值。如果目标类型为 System.Collections.Generic.IList(Of T)、IReadOnlyList(Of T)、ICollection(Of T)、IReadOnlyCollection(Of T) 或 IEnumerable(Of T)，并且数组文本具有一级嵌套，则将数组文本重新分类为类型 T() 的值。
7. 否则，将数组文本重新分类为这样一个值：其类型是秩为所用嵌套级别数的数组，其元素类型由初始值设定项中元素的主导类型确定；如果无法确定主导类型，则使用 Object。例如：

' x Is GetType(Double(,,))  
Dim x = { { { 1, 2.0 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } }.GetType()  
  
' y Is GetType(Integer())  
Dim y = { 1, 2, 3 }.GetType()  
  
' z Is GetType(Object())  
Dim z = { 1, "2" }.GetType()  
  
' Error: Inconsistent nesting  
Dim a = { { 10 }, { 20, 30 } }.GetType()

批注

语言版本 9.0 和 10.0 之间在行为上有些许更改。10.0 以前，数组元素初始值设定项不会影响局部变量类型推断，但现在则相反。因此，在 9.0 版语言中，Dim a() = { 1, 2, 3 } 会推断 a 的类型为 Object()，而在 10.0 版语言中，则推断为 Integer()。

然后，此重新分类会将数组文本重新解释为数组创建表达式。因此，以下示例：

Dim x As Double = { 1, 2, 3, 4 }  
Dim y = { "a", "b" }

等效于：

Dim x As Double = New Double() { 1, 2, 3, 4 }  
Dim y = New String() { "a", "b" }

如果从元素表达式到数组元素类型的任何转换是收缩转换，则将重新分类判断为收缩转换；否则，将其判断为扩大转换。

1. 可以将默认值 Nothing 重新归类为值。在目标类型已知的上下文中，结果将是该目标类型的默认值。在目标类型未知的上下文中，结果将是类型 Object 的 null 值。

无法对命名空间表达式、类型表达式、事件访问表达式或 void 表达式进行重新分类。可以同时执行多种重新分类。例如：

Module Test  
 Sub F(i As Integer)  
 End Sub  
  
 ReadOnly Property P() As Integer  
 Get  
 End Get  
 End Sub  
  
 Sub Main()  
 F(P)  
 End Property  
End Module

在此示例中，首先将属性组表达式 P 从属性组重新归类为属性访问，然后再将其从属性访问重新归类为一个值。执行最少次数的重新分类以在上下文中实现有效分类。

## 常量表达式

常量表达式是一个可以在编译时对其完全求值的表达式。常量表达式的类型可以是 Byte、SByte、UShort、Short、UInteger、Integer、ULong、Long、Char、Single、Double、Decimal、Date、Boolean、String、Object 或任何枚举类型。常量表达式中允许下列构造：

文本（包括 Nothing）。

对常量类型成员或常量局部变量的引用。

对枚举类型的成员的引用。

带括号的子表达式。

强制转换表达式（前提是目标类型为以上列出的类型之一）。对于此规则而言，与 String 进行强制转换是一个例外，只允许针对 null 值执行此类操作，因为 String 转换应始终在运行时的执行环境的当前区域文化中执行。请注意，常量强制转换表达式只能使用内部转换。

+、– 和 Not 一元运算符，前提是操作数和结果属于上面列出的类型。

The +、–、\*、^、Mod、/、\、<<、>>、&、And、Or、Xor、AndAlso、OrElse、=、<、>、<>、<= 和 => 二进制运算符，前提是每个操作数和结果均属于上面列出的类型。

条件运算符If，前提是每个操作数和结果均为上面列出的类型。

以下是运行时函数：

Microsoft.VisualBasic.Strings.ChrW

Microsoft.VisualBasic.Strings.Chr，如果常量值介于 0 和 128 之间

Microsoft.VisualBasic.Strings.AscW，如果常量字符串不为空

Microsoft.VisualBasic.Strings.Asc，如果常量字符串不为空

常量表达式中不允许下列构造：

通过 With 上下文的隐式绑定。

整型（ULong、Long、UInteger、Integer、UShort、Short、SByte 或 Byte）的常量表达式可以隐式转换为范围更窄的整型，而类型 Double 的常量表达式可以隐式转换为 Single，前提是常量表达式的值在目标类型的范围内。无论使用的是宽松语义还是严格语义，都允许这些收缩转换。

ConstantExpression ::= Expression

## 后期绑定表达式

当成员访问表达式或索引表达式的目标属于类型 Object 时，可以将对表达式的处理推迟到运行时进行。按此方式推迟处理的这一做法称作后期绑定。利用后期绑定，可以按无类型方式使用 Object 变量，这样，对成员的所有解析都基于变量中的值的实际运行时类型。如果编译环境或 Option Strict 指定严格语义，则后期绑定将导致编译时错误。在执行后期绑定时（包括为进行重载解析），非公共成员将被忽略。请注意，与早期绑定情况不同，调用或访问后期绑定的 Shared 成员将导致在运行时计算调用目标。如果表达式是在 System.Object 上定义的成员的调用表达式，则不会发生后期绑定。

一般来说，后期绑定访问是在运行时通过查找表达式的实际运行时类型上的标识符来解析的。如果后期绑定成员查找在运行时失败，则会引发 System.MissingMemberException 异常。由于后期绑定成员查找是独立于关联目标表达式的运行时类型执行的，并且对象的运行时类型绝不会是一个接口。因此，无法在后期绑定成员表达式中访问接口成员。

The arguments to a late-bound member access are evaluated in the order they appear in the member access expression: not the order in which parameters are declared in the late-bound member. The following example illustrates this difference:Class C

Public Sub f(ByVal x As Integer, ByVal y As Integer)

End Sub

End Class

Module Module1  
 Sub Main()

Console.Write("Early-bound: ")

Dim c As C = New C

c.f(y:=t("y"), x:=t("x"))

Console.Write(vbCrLf & "Late-bound: ")

Dim o As Object = New C

o.f(y:=t("y"), x:=t("x"))

End Sub

Function t(ByVal s As String) As Integer

Console.Write(s)

Return 0

End Function

End Module

此代码显示：

Early-bound: xy

Late-bound: yx

由于后期绑定重载解析是基于参数的运行时类型执行的，因此表达式可能会产生不同的结果，具体取决于表达式是在编译时还是运行时计算的。下面的示例说明了此差异：

Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module Test  
 Sub F(b As Base)  
 Console.WriteLine("F(Base)")  
 End Sub  
  
 Sub F(d As Derived)  
 Console.WriteLine("F(Derived)")  
 End Sub  
  
 Sub Main()  
 Dim b As Base = New Derived()  
 Dim o As Object = b  
  
 F(b)  
 F(o)  
 End Sub  
End Module

此代码显示：

F(Base)  
F(Derived)

## 简单表达式

简单表达式包括文本、带括号的表达式、实例表达式或简单名称表达式。

SimpleExpression ::=  
 LiteralExpression |  
 ParenthesizedExpression |  
 InstanceExpression |  
 SimpleNameExpression |  
 AddressOfExpression

### 文本表达式

文本表达式的计算结果为文本表示的值。文本表达式将归类为一个值（文本 Nothing 除外，它将归类为一个默认值）。

LiteralExpression ::= Literal

### 带括号的表达式

带括号的表达式由一个用括号括起来的表达式构成。带括号的表达式将归类为一个值，而包含的表达式必须归类为一个值。带括号的表达式的计算结果为括号内的表达式的值。

ParenthesizedExpression ::= OpenParenthesis Expression CloseParenthesis

### 实例表达式

实例表达式 为关键字 Me。该表达式只能在非共享方法、构造函数或属性访问器体中使用。它被归类为一个值。.关键字 Me 表示包含正在执行的方法或属性访问器的类型的实例。如果一个构造函数显式调用其他构造函数（9.3 节），则只有在该构造函数调用之后，才可使用 Me，因为尚未构造该实例。

InstanceExpression ::= Me

### 简单名称表达式

简单名称表达式由一个标识符以及后跟的可选类型参数列表构成。在解析名称并对其分类时，遵循以下“简单名称解析规则”：

1. 将从最里层的封闭块开始解析和分类，然后再继续向外对每个外部封闭块（如果有）进行解析和分类。如果标识符与局部变量、静态变量、局部常量、方法类型参数或参数的名称匹配，则标识符将引用匹配的实体。

如果标识符与局部变量、静态变量、常量局部变量匹配并且提供了类型参数列表，则将发生编译时错误。如果标识符与方法类型参数匹配并且提供了类型参数列表，则不会发生任何匹配且解析将继续。如果标识符与局部变量匹配，匹配的局部变量为隐式函数或 Get 访问器返回局部变量，且表达式是调用表达式、调用语句或 AddressOf 表达式的一部分，则不会发生任何匹配，并将继续进行解析。

表达式将归类为一个变量（如果表达式是局部变量、静态变量或参数）。表达式将归类为一个类型（如果表达式是方法类型参数）。表达式将归类为一个值（如果表达式是常量局部变量）。

1. 对于包含表达式的每个嵌套类型，顺序是从最内层的嵌套类型到最外层的嵌套类型（如果对类型中的标识符进行查找时得到与可访问成员匹配的项）：
2. 如果匹配的类型参数是一个类型参数，则结果为匹配的类型参数并将归类为一个类型。如果提供了类型参数列表，则不会发生任何匹配且解析将继续。
3. 否则，如果类型是最里层的封闭类型且查找标识非共享类型成员，则结果与形式为 Me.E(Of A) 的成员访问相同，其中 E 为标识符，而 A 为类型参数列表（如果有）。
4. 否则，结果与形式为 T.E(Of A) 的成员访问完全相同，其中 T 为包含匹配成员的类型，E 为标识符，而 A 为类型参数列表（如果有）。在此情况下，标识符引用非共享成员是错误的。
5. 对于每个嵌套的命名空间（顺序是从最内层的命名空间到最外层的命名空间），执行以下操作：
6. 如果命名空间包含一个带有给定名称的可访问类型，并且该类型具有类型参数列表中提供的所有类型参数（如果有），则标识符将引用该类型并被归类为一个类型。
7. 否则，如果未提供任何类型参数列表，且命名空间包含带有给定名称的命名空间成员，则标识符将引用该命名空间并归类为一个命名空间。
8. 否则，如果命名空间包含一个或多个可访问的标准模块，并且对标识符进行成员名称查找时正好在一个标准模块中得到一个可访问的匹配项，则结果与形式为 M.E(Of A) 的成员访问完全相同，其中 M 为包含匹配成员的标准模块，E 为标识符，而 A 为类型参数列表（如果有）。如果标识符与多个标准模块中的可访问的类型成员匹配，则将发生编译时错误。
9. 如果源文件具有一个或多个导入别名，并且标识符与其中一个别名的名称相匹配，则标识符将引用该命名空间或类型。如果提供了类型实参列表，则将发生编译时错误。
10. 如果包含名称引用的源文件具有一个或多个导入：
11. 如果标识符与正好一个导入中的可访问类型（类型形参数目与类型实参列表（如果有）中提供的实参数目相同）或类型成员的名称相匹配，则标识符将引用该类型或类型成员。如果标识符与多个导入中的可访问类型（类型形参数目与类型实参列表（如果有）中提供的实参数目相同）或可访问类型成员的名称相匹配，则将发生编译时错误。
12. 否则，如果未提供类型实参列表，并且标识符正好与一个导入中具有可访问类型的命名空间的名称相匹配，则标识符将引用该命名空间。如果未提供类型参数列表，并且标识符与多个导入中具有可访问类型的命名空间的名称相匹配，则将发生编译时错误。
13. 否则，如果导入包含一个或多个可访问的标准模块，并且对标识符进行成员名称查找时正好在一个标准模块中得到一个可访问的匹配项，则结果与形式为 M.E(Of A) 的成员访问完全相同，其中 M 为包含匹配成员的标准模块，E 为标识符，而 A 为类型参数列表（如果有）。如果标识符与多个标准模块中的可访问的类型成员匹配，则将发生编译时错误。
14. 如果编译环境定义了一个或多个导入别名，并且标识符与其中一个别名的名称相匹配，则标识符将引用该命名空间或类型。如果提供了类型实参列表，则将发生编译时错误。
15. 如果编译环境定义一个或多个导入，则：
16. 如果标识符与正好一个导入中的可访问类型（类型形参数目与类型实参列表（如果有）中提供的实参数目相同）或类型成员的名称相匹配，则标识符将引用该类型或类型成员。如果标识符与多个导入中的可访问类型（类型形参数目与类型实参列表（如果有）中提供的实参数目相同）或类型成员的名称相匹配，则将发生编译时错误。
17. 否则，如果未提供类型实参列表，并且标识符正好与一个导入中具有可访问类型的命名空间的名称相匹配，则标识符将引用该命名空间。如果未提供类型参数列表，并且标识符与多个导入中具有可访问类型的命名空间的名称相匹配，则将发生编译时错误。
18. 否则，如果导入包含一个或多个可访问的标准模块，并且对标识符进行成员名称查找时正好在一个标准模块中得到一个可访问的匹配项，则结果与形式为 M.E(Of A) 的成员访问完全相同，其中 M 为包含匹配成员的标准模块，E 为标识符，而 A 为类型参数列表（如果有）。如果标识符与多个标准模块中的可访问的类型成员匹配，则将发生编译时错误。
19. 否则，标识符给定的名称是未定义的。

未定义的简单名称表达式会导致编译时错误。

一般来说，在特定的命名空间中，一个名称只能出现一次。但由于可以跨多个 .NET 程序集声明命名空间，因此可能会遇到两个程序集使用同一完全限定名来定义一个类型的情况。在此情况下，当前源文件集中声明的类型将优先于外部 .NET 程序集中声明的类型。否则，名称将出现无法消除的多义性。

SimpleNameExpression ::= Identifier [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

### AddressOf 表达式

AddressOf 表达式用于生成一个方法指针。此表达式由 AddressOf 关键字和一个必须归类为方法组或后期绑定访问的表达式构成。方法组无法引用构造函数。

结果将归类为一个方法指针，此方法指针具有与方法组相同的关联目标表达式和类型参数列表（如果有）。

AddressOfExpression ::= AddressOf Expression

## 类型表达式

类型表达式 是 GetType 表达式、TypeOf...Is 表达式、Is 表达式或 GetXmlNamespace 表达式。

TypeExpression ::=  
 GetTypeExpression |  
 TypeOfIsExpression |  
 IsExpression |  
 GetXmlNamespaceExpression

### GetType 表达式

GetType 表达式由关键字 GetType 和类型名称构成。类型表达式将归类为一个值，并且此表达式的值是表示类型的反射 (System.Type)) 类。如果表达式是一个类型形参，则表达式将返回 System.Type 对象，此对象与在运行时为该类型形数提供的类型实参相对应。

GetType 表达式中的类型名称具有两大特别之处：

类型名称允许为 System.Void，它是 Visual Basic 语言中唯一可引用此类型名称的位置。

类型名称可以是带有省略的类型参数的构造泛型类型。这会允许 GetType 表达式返回与泛型类型本身对应的 System.Type 对象。

下面的示例演示了 GetType 表达式：

Module Test  
 Sub Main()  
 Dim t As Type() = { GetType(Integer), GetType(System.Int32), \_  
 GetType(String), GetType(Double()) }  
 Dim i As Integer  
  
 For i = 0 To t.Length – 1  
 Console.WriteLine(t(i).Name)  
 Next i  
 End Sub  
End Module

得到的输出为：

Int32  
Int32  
String  
Double[]

GetTypeExpression ::= GetType OpenParenthesis GetTypeTypeName CloseParenthesis

GetTypeTypeName ::=  
 TypeName |  
 *Qualified*OpenTypeName

QualifiedOpenTypeName ::=  
 Identifier [ TypeArityList ] |  
 Global Period IdentifierOrKeyword [ TypeArityList ] |  
 QualifiedOpenTypeName Period IdentifierOrKeyword [ TypeArityList ]

*TypeArityList* ::= ( Of [ *CommaList* ] )

### CommaList ::= Comma | CommaList CommaTypeOf...Is 表达式

TypeOf...Is 表达式用于检查一个值的运行时类型是否与给定类型兼容。第一个操作数必须归类为一个值，而不能是重新归类的 lambda 方法，并且它必须属于引用类型或无约束的类型参数类型。第二个操作数必须是类型名称。表达式的结果将归类为一个值，并且是一个 Boolean 值。如果操作数的运行时类型具有到该类型的标识转换、默认转换、引用转换、数组转换、值类型转换或类型参数转换，则表达式的计算结果为 True；否则表达式的计算结果为 False。如果表达式的类型和特定类型之间不存在任何转换，则将发生编译时错误。

TypeOfIsExpression ::= TypeOf Expression Is [ LineTerminator ] TypeName

### Is 表达式

Is 或 IsNot 表达式用于执行引用相等比较。这两个表达式都必须归类为一个值，并且二者的类型必须是引用类型、无约束的类型参数类型或可为 null 的值类型。但如果其中一个表达式的类型是无约束的类型参数类型或可以为 null 的值类型，则另一个表达式必须为文本 Nothing。

结果将归类为一个值并类型化为 Boolean。如果这两个值都引用同一实例或都为 Nothing，则 Is 运算的计算结果为 True；否则为 False。如果这两个值都引用同一实例或都为 Nothing，则 IsNot 运算的计算结果为 False；否则为 True。

IsExpression ::=  
 Expression Is [ LineTerminator ] Expression |  
 Expression IsNot [ LineTerminator ] Expression

### GetXmlNamespace 表达式

GetXmlNamespace 表达式由关键字 GetXmlNamespace 和通过源文件或编译环境声明的 XML 命名空间的名称构成。XML 命名空间表达式归类为一个值，并且此表达式的值为表示该 XML 命名空间的 System.Xml.Linq.XNamespace 的实例。如果该类型不可用，则将发生编译时错误。

例如：

Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 Dim db = GetXmlNamespace(db)  
  
 ' The following are equivalent  
 Dim customer1 = \_  
 New System.Xml.Linq.XElement(db + "customer", "Bob")  
 Dim customer2 = <db:customer>Bob</>  
 End Sub  
End Module

括号内的所有内容将视为命名空间名称的一部分，因此针对内容（如空格）的 XML 规则将适用。例如：

Imports <xmlns:db-ns="http://example.org/database">  
  
Module Test  
 Sub Main()  
  
 ' Error, XML name expected  
 Dim db1 = GetXmlNamespace( db-ns )  
  
 ' Error, ')' expected  
 Dim db2 = GetXmlNamespace(db \_  
 )  
  
 ' OK.  
 Dim db3 = GetXmlNamespace(db-ns)  
 End Sub  
End Module

也可以省略 XML 命名空间表达式，在此情况下，表达式将返回表示默认 XML 命名空间的对象。

GetXmlNamespaceExpression ::= GetXmlNamespace OpenParenthesis [ XMLNamespaceName ]  
 CloseParenthesis

## 成员访问表达式

成员访问表达式可用于访问实体的成员。E.I(Of A) 形式的成员访问（其中 E 为表达式、非数组类型名称、关键字 Global 或省略，而 I 为带有可选类型实参列表 A 的标识符）的计算和分类方式如下：

1. 如果省略 E，则来自直接包含 With 语句的表达式将替换 E，并且将执行成员访问。如果不存在包含 With 语句，则将发生编译时错误。
2. 如果 E 归类为一个命名空间或 E 为关键字 Global，则将在指定的命名空间的上下文中执行成员查找。如果 I 为该命名空间的一个可访问成员的名称且该成员具有类型参数列表中（如果有）提供的相同数量的类型参数，则结果为该成员。结果将归类为一个命名空间或类型，具体取决于该成员。其他情况下，将发生编译时错误。
3. 如果 E 是一个类型或一个归类为类型的表达式，则将在指定类型的上下文中执行成员查找。如果 I 是 E 的一个可访问成员的名称，则 E.I 的计算和分类方式如下：
4. 如果 I 是关键字 New，并且 E 不是枚举，则将发生编译时错误。
5. 如果 I 标识一个具有类型参数列表（如果有）中提供的相同数量的类型参数的类型，则结果为该类型。
6. 如果 I 标识一个或多个方法，则结果为一个方法组，而且此方法组带有关联的类型参数列表，但不带关联的目标表达式。
7. 如果 I 标识一个或多个属性，并且未提供类型参数列表，则结果为一个不带关联的目标表达式的属性组。
8. 如果 I 标识一个共享变量，并且未提供类型参数列表，则结果为一个变量或一个值。如果变量是只读的，并且引用发生在声明变量的类型的共享构造函数外部，则结果为共享变量 I 在 E 中具有的值。否则，结果为 E 中的共享变量 I。
9. 如果 I 标识一个共享事件，并且未提供类型参数列表，则结果为一个不带关联的目标表达式的事件访问。
10. 如果 I 标识一个常量，并且未提供类型参数列表，则结果为该常量的值。
11. 如果 I 标识一个枚举成员，并且未提供类型参数列表，则结果为该枚举成员的值。
12. 否则，E.I 是无效成员引用，并且会出现编译时错误。
13. 如果 E 归类为一个变量或值（其类型为 T），则将在 T 的上下文中执行成员查找。如果 I 是 T 的一个可访问成员的名称，则 E.I 的计算和分类方式如下：
14. 如果 I 是关键字 New，E 是 Me、MyBase 或 MyClass，并且未提供类型实参，则结果为表示类型 E 的实例构造函数的方法组，此方法组带有 E 的关联目标表达式，并且不带类型实参列表。其他情况下，将发生编译时错误。
15. 如果 I 标识一个或多个方法（包括扩展方法）且 T 不为 Object，则结果为一个方法组，此方法组带有关联的类型参数列表和 E 的关联的目标表达式。
16. 如果 I 标识一个或多个属性，并且未提供类型参数，则结果为一个带有 E 的关联目标表达式的属性组。
17. 如果 I 标识一个共享变量或实例变量，并且未提供类型参数，则结果为一个变量或一个值。如果变量是只读的，并且引用发生在声明变量的类的构造函数（对应于变量是共享变量还是实例变量）的外部，则结果为变量 I 在 E 引用的对象中具有的值。如果 T 为引用类型，则结果为 E 引用的对象中的变量 I。在相反的情况下，如果 T 为一个值类型，并且表达式 E 归类为一个变量，则结果为一个变量；否则结果为一个值。
18. 如果 I 标识一个事件，并且未提供类型参数，则结果为一个带有 E 的关联目标表达式的事件访问。
19. 如果 I 标识一个常量，并且未提供类型参数，则结果为该常量的值。
20. 如果 I 标识一个枚举成员，并且未提供类型参数，则结果为该枚举成员的值。
21. 如果 T 为 Object，则结果为一个归类为后期绑定访问的后期绑定成员查找，它带有关联的类型参数列表和 E 的关联的目标表达式。
22. 否则，E.I 是无效成员引用，并且会出现编译时错误。

MyClass.I(Of A) 形式的成员访问等效于 Me.I(Of A)，但在其上访问的所有成员都按不可重写成员的处理。因此，所访问的成员将不受正在访问该成员的值的运行时类型影响。

MyBase.I(Of A) 形式的成员访问等效于 CType(Me, T).I(Of A)，其中，T 是包含成员访问表达式的类型的直接基类型。在其上进行的所有方法调用均按不可重写方法处理。这种形式的成员访问也称为*基访问*。

下面的示例演示 Me、MyBase 和 MyClass 之间的相关方式：

Class Base  
 Public Overridable Sub F()  
 Console.WriteLine("Base.F")  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Public Overrides Sub F()  
 Console.WriteLine("Derived.F")  
 End Sub  
  
 Public Sub G()  
 MyClass.F()  
 End Sub  
End Class  
  
Class MoreDerived  
 Inherits Derived  
  
 Public Overrides Sub F()  
 Console.WriteLine("MoreDerived.F")  
 End Sub  
  
 Public Sub H()  
 MyBase.F()  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As MoreDerived = new MoreDerived()  
  
 x.F()  
 x.G()  
 x.H()  
 End Sub  
  
End Module

此代码将输出：

MoreDerived.F  
Derived.F  
Derived.F

当成员访问表达式以关键字 Global 开头时，此关键字表示未命名的最外层命名空间，它在声明隐藏封闭命名空间时很有用。在此情况下，Global 关键字允许“转义”到最外层命名空间。例如：

Class System  
End Class  
  
Module Test  
 Sub Main()  
 ' Error: Class System does not contain Console  
 System.Console.WriteLine("Hello, world!")   
  
  
 ' Legal, binds to System in outermost namespace  
 Global.System.Console.WriteLine("Hello, world!")   
 End Sub  
End Module

在上面的示例中，第一个方法调用是无效的，因为标识符 System 绑定到类 System，而不是绑定到命名空间 System。只能通过使用 Global 转义到最外层命名空间来访问 System 命名空间。

如果正在访问的成员是共享成员，则句点左侧的任何表达式都是多余的，除非成员访问执行了后期绑定，否则不会计算这些表达式。例如，考虑以下代码：

Class C  
 Public Shared F As Integer = 10  
End Class  
  
Module Test  
 Public Function ReturnC() As C  
 Console.WriteLine("Returning a new instance of C.")  
 Return New C()  
 End Function  
  
 Public Sub Main()  
 Console.WriteLine("The value of F is: " & ReturnC().F)  
 End Sub  
End Module

此代码将输出 The value of F is: 10，因为无需调用函数 ReturnC 即可提供 C 的实例来访问共享成员 F。

MemberAccessExpression ::=  
 [ MemberAccessBase ] Period IdentifierOrKeyword  
 [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

MemberAccessBase ::=  
 Expression |  
 NonArrayTypeName |  
 Global |  
 MyClass |  
 MyBase

### 相同的类型和成员名称

使用成员的类型名称来命名成员的情况很常见。但在此情况下，不方便进行名称隐藏：

Enum Color  
 Red  
 Green  
 Yellow  
End Enum  
  
Class Test  
 ReadOnly Property Color() As Color  
 Get  
 Return Color.Red  
 End Get  
 End Property  
  
 Shared Function DefaultColor() As Color  
 Return Color.Green ' Binds to the instance property!  
 End Function  
End Class

在上一示例中，DefaultColor 中的简单名称 Color 绑定到实例属性，而不是绑定到类型。由于无法在共享成员中引用实例成员，因此这通常会导致出现错误。

不过，此时可通过一个特殊规则来访问类型。如果成员访问表达式的基表达式是一个简单名称且绑定到具有相同类型名称的常量、字段、属性、局部变量或参数，则基表达式可以引用成员或类型。由于在上述任一情况下可访问的成员都是相同的，因此绝不会导致出现歧义。

### 默认实例

在某些情况下，派生自常用基类的类通常或始终只具有一个实例。例如，用户界面上显示的大多数窗口在任何时候都只会在屏幕上显示一个实例。为了简化对这些类型的类的使用，Visual Basic 会自动生成这些类的默认实例，以便为每个类提供一个易于引用的实例。

默认实例总是针对一系列类型（而非某个特定类型）创建的。因此，将为派生自 Form 的所有类创建默认实例，而不是为派生自 Form 的类 Form1 创建默认实例。这意味着，不必将派生自基类的各个类专门标记为具有默认实例。

一个类的默认实例由编译器生成的属性表示，此属性可返回该类的默认实例。作为类的成员生成的属性称作组类，用于管理对派生自特定基类的所有类的默认实例所执行的分配和销毁操作。例如，可以在 MyForms 类中收集派生自 Form 的类的所有默认实例属性。如果表达式 My.Forms 返回组类的实例，则以下代码将访问派生类 Form1 和 Form2 的默认实例：

Class Form1  
 Inherits Form  
 Public x As Integer  
End Class  
  
Class Form2  
 Inherits Form  
 Public y As Integer  
End Class  
  
Module Main  
 Sub Main()  
 My.Forms.Form1.x = 10  
 Console.WriteLine(My.Forms.Form2.y)  
 End Sub  
End Module

在首次引用默认实例之前，不会创建默认实例；若获取表示默认实例的属性，则会导致创建默认实例（如果尚未创建默认实例或已将默认实例设置为 Nothing）。为了能够测试默认实例是否存在，当默认实例是 Is 或 IsNot 运算符的目标时，将不会创建默认实例。这样便能够测试默认实例是 Nothing 还是其他一些引用，而不会导致创建默认实例。

默认实例旨在使从具有默认实例的类的外部引用默认实例变得轻松。若从定义默认实例的类中使用默认实例，则可能会导致对引用了哪个实例（即默认实例或当前实例）造成混淆。例如，以下代码只修改默认实例中的值 x，即使该值是从另一个实例中调用的。因此，此代码将输出值 5 而不是 10：

Class Form1  
 Inherits Form  
  
 Public x As Integer = 5  
  
 Public Sub ChangeX()  
 Form1.x = 10  
 End Sub  
End Class  
  
Module Main  
 Sub Main()  
 Dim f As Form1 = New Form1()  
 f.ChangeX()  
 Console.WriteLine(f.x)  
 End Sub  
End Module

为了防止出现此类混淆，从默认实例的类型的实例方法中引用默认实例将被视为无效。

#### 默认实例和类型名称

也可以通过默认实例的类型名称来直接访问默认实例。在此情况下，在不允许类型名称的任何表达式上下文中，表达式 E（其中 E 表示带有默认实例的类的完全限定名）将更改为 E’（其中 E’ 表示获取默认实例属性的表达式）。例如，如果派生自 Form 的类的默认实例允许通过类型名称来访问默认实例，则以下代码等效于上一示例中的代码：

Module Main  
 Sub Main()  
 Form1.x = 10  
 Console.WriteLine(Form2.y)  
 End Sub  
End Module

这同时还意味着，可通过其类型名称访问的默认实例也可通过其类型名称进行赋值。例如，以下代码将 Form1 的默认实例设置为 Nothing：

Module Main  
 Sub Main()  
 Form1 = Nothing  
 End Sub  
End Module

请注意，E.I 的含义（其中 E 表示一个类，而 I 表示一个共享成员）不会发生更改。此类表达式仍会直接访问类实例中的共享成员，并且不会引用默认实例。

#### 组类

Microsoft.VisualBasic.MyGroupCollectionAttribute 特性指示一系列默认实例的组类。此特性具有四个参数：

参数 TypeToCollect 指定组的基类。不带开放类型参数（派生自具有此名称的类型）的所有可实例化的类（与类型参数无关）将自动具有默认实例。

参数 CreateInstanceMethodName 指定要在组类中调用的方法，通过调用此方法可在默认实例属性中创建新实例。

参数 DisposeInstanceMethodName 指定要在组类中调用的方法，通过调用此方法可释放默认实例属性（如果将值 Nothing 赋给默认实例属性）。

参数 DefaultInstanceAlias 指定表达式 E’ 替代类名（如果默认实例可通过其类型名称直接访问）。如果此参数是 Nothing 或空字符串，则该组类型的默认实例将无法通过其类型名称直接访问。

批注

在 Visual Basic 语言的所有当前实现中，将忽略 DefaultInstanceAlias 参数（在编译器提供的代码中除外）。

通过使用逗号在前三个参数中将类型和方法的名称分隔开，可以将多个类型收集到同一个组中。每个参数中的项数必须相同，并且各个列表元素是按顺序匹配的。例如，以下特性声明将派生自 C1、C2 或 C3 的类型收集到一个组中：

<Microsoft.VisualBasic.MyGroupCollection("C1, C2, C3", \_  
 "CreateC1, CreateC2, CreateC3", \_  
 "DisposeC1, DisposeC2, DisposeC3", "My.Cs")>  
Public NotInheritable Class MyCs  
 …  
End Class

创建方法的签名必须采用 Shared Function <Name>(Of T As {New, <Type>})(Instance Of T) As T 的形式。释放方法的形式必须为 Shared Sub <Name>(Of T As <Type>)(ByRef Instance Of T)。因此，上一部分中的示例的组类可以按以下方式声明：

<Microsoft.VisualBasic.MyGroupCollection("Form", "Create", \_  
 "Dispose", "My.Forms")> \_  
Public NotInheritable Class MyForms  
 Private Shared Function Create(Of T As {New, Form}) \_  
 (Instance As T) As T  
 If Instance Is Nothing Then  
 Return New T()  
 Else  
 Return Instance  
 End If  
 End Function  
  
 Private Shared Sub Dispose(Of T As Form)(ByRef Instance As T)  
 Instance.Close()  
 Instance = Nothing  
 End Sub  
End Class

如果源文件声明了一个派生类 Form1，则生成的组类将等效于：

<Microsoft.VisualBasic.MyGroupCollection("Form", "Create", \_  
 "Dispose", "My.Forms")> \_  
Public NotInheritable Class MyForms  
 Private Shared Function Create(Of T As {New, Form}) \_  
 (Instance As T) As T  
 If Instance Is Nothing Then  
 Return New T()  
 Else  
 Return Instance  
 End If  
 End Function  
  
 Private Shared Sub Dispose(Of T As Form)(ByRef Instance As T)  
 Instance.Close()  
 Instance = Nothing  
 End Sub  
  
 Private m\_Form1 As Form1  
  
 Public Property Form1() As Form1  
 Get  
 Return Create(m\_Form1)  
 End Get  
 Set (Value As Form1)  
 If Value IsNot Nothing AndAlso Value IsNot m\_Form1 Then  
 Throw New ArgumentException( \_  
 "Property can only be set to Nothing.")  
 End If  
 Dispose(m\_Form1)  
 End Set  
 End Property  
End Class

### 扩展方法收集

可以通过收集当前上下文中可用的带有名称 I 的所有扩展方法，收集成员访问表达式 E.I 的扩展方法：

1. 首先，检查每个包含表达式的嵌套类型，顺序是从最内层的类型到最外层的类型。
2. 再检查每个嵌套的命名空间，顺序是从最内层的命名空间到最外层的命名空间。
3. 紧接着，检查源文件中的导入。
4. 然后，检查由编译环境定义的导入。

仅当存在从目标表达式类型到扩展方法的第一个参数的类型的扩大本机转换时，才收集扩展方法。与常规的简单名称表达式绑定不同，搜索将收集所有扩展方法；在找到一个扩展方法后，收集过程不会停止。例如：

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class

Namespace N1  
 Module N1C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, x As Integer)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2  
 Module N2C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, y As Double)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2.N3  
 Module Test  
 Sub Main()  
 Dim x As New C1()  
  
 ' Calls N1C1Extensions.M1  
 x.M1(10)  
 End Sub  
 End Module  
End Namespace

在此示例中，即使 N2C1Extensions.M1 位于 N1C1Extensions.M1 的前面，这两者也被视为是扩展方法。在收集完所有扩展方法后，将对这些方法进行扩充。扩充采用扩展方法调用的目标并将其应用于扩展方法调用，这将生成新的方法签名并会删除第一个参数（由于该参数已指定）。例如：

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M(x As Integer, y As Integer)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M(x As Integer, y As Double)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim v As Integer = 10  
  
 ' The curried method signatures considered are:  
 ' Ext1.M(y As Integer)  
 ' Ext2.M(y As Double)  
 v.M(10)  
 End Sub  
End Module

在上面的示例中，将 v 应用于 Ext1.M 所得到的扩充结果为方法签名 Sub M(y As Integer)。

扩充不仅会删除扩展方法的第一个参数，而且会删除属于第一个参数的类型的任何方法类型参数。在扩充带有方法类型参数的扩展方法时，类型推断将应用于第一个参数，并且得到的结果对于推断出的任何类型参数都是一样的。如果类型推断失败，则忽略方法。例如：

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M(Of T, U)(x As T, y As U)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M(Of T)(x As T, y As T)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim v As Integer = 10  
  
 ' The curried method signatures considered are:  
 ' Ext1.M(Of U)(y As U)  
 ' Ext2.M(y As Integer)  
 v.M(10)  
 End Sub  
End Module

在上面的示例中，将 v 应用于 Ext1.M 所得到的扩充结果为方法签名 Sub M(Of U)(y As U)，这是因为，类型形参 T 被推断为扩充的结果，并且此结果现在是不变的。由于类型参数 U 未被推断为扩充的一部分，因此该参数仍将为开放参数。同样，由于类型形参 T 是作为将 v 应用于 Ext2.M 所得到的结果推断出来的，因此，形参 y 的类型变为 Integer 并保持不变。而不会将它推断为任何其他类型。在扩充签名时，还将应用所有约束（New 约束除外）。如果约束未得到满足或依赖于一个未被推断为扩充的一部分的类型，则忽略扩展方法。例如：

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M1(Of T As Structure)(x As T, y As Integer)  
 End Sub  
  
 <Extension> \_  
 Sub M2(Of T As U, U)(x As T, y As U)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim s As String = "abc"  
  
 ' Error: String does not satisfy the Structure constraint  
 s.M1(10)  
  
 ' Error: T depends on U, which cannot be inferred  
 s.M2(10)  
 End Sub  
End Module

批注

对扩展方法进行扩充的主要原因之一是，通过扩充可让查询表达式在计算查询模式方法的参数之前推断迭代的类型。由于大多数查询模式方法使用的是 lambda 表达式（这些表达式自身需要类型推断），因此这样做可大大地简化查询表达式的计算过程。

与一般的接口继承不同，只要用于扩展两个互不关联的接口的扩展方法所具有的扩充签名不同，就可以使用这些扩展方法：

Imports System.Runtime.CompilerServices  
  
Interface I1  
End Interface  
  
Interface I2  
End Interface  
  
Class C1  
 Implements I1, I2  
End Class  
  
Module I1Ext  
 <Extension> \_  
 Sub M1(i As I1, x As Integer)  
 End Sub  
  
 <Extension> \_  
 Sub M2(i As I1, x As Integer)  
 End Sub  
End Module  
  
Module I2Ext  
 <Extension> \_  
 Sub M1(i As I2, x As Integer)  
 End Sub  
  
 <Extension> \_  
 Sub M2(I As I2, x As Double)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim c As New C1()  
  
 ' Error: M is ambiguous between I1Ext.M1 and I2Ext.M1.  
 c.M1(10)  
  
 ' Calls I1Ext.M2  
 c.M2(10)  
 End Sub  
End Module

最后请记住，在执行后期绑定时不考虑扩展方法，这一点很重要：

Module Test  
 Sub Main()  
 Dim o As Object = …  
  
 ' Ignores extension methods  
 o.M1()  
 End Sub  
End Module

## 字典成员访问表达式

字典成员访问表达式用于查找集合的成员。字典成员访问表达式采用的形式为 E!I，其中 E 是归类为一个值的表达式，而 I 是标识符。表达式的类型必须具有由一个 String 参数进行索引的默认属性。字典成员访问表达式 E!I 将转换为表达式 E.D("I")，其中 D 是 E 的默认属性。例如：

Class Keys  
 Public ReadOnly Default Property Item(s As String) As Integer  
 Get  
 Return 10  
 End Get  
 End Property   
End Class  
  
Module Test  
 Sub Main()  
 Dim x As Keys = new Keys()  
 Dim y As Integer  
 ' The two statements are equivalent.  
 y = x!abc  
 y = x("abc")  
 End Sub  
End Module

如果未用任何表达式指定感叹号，则假定使用来自直接包含 With 语句的表达式。如果不存在包含 With 语句，则将发生编译时错误。

DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword

## 调用表达式

调用表达式由一个调用目标和一个可选参数列表构成。目标表达式必须归类为一个方法组或值（其类型是委托类型）。如果目标表达式是一个其类型为委托类型的值，则调用表达式的目标将成为此委托类型的 Invoke 成员的方法组，而目标表达式将成为此方法组的关联的目标表达式。

参数列表包含两个部分：位置参数和命名参数。位置参数是表达式，它必须位于任何命名参数的前面。命名参数由一个可匹配关键字的标识符以及后跟的 := 和表达式构成。

如果方法组只包含一个可访问的方法（包括实例方法和扩展方法），并且此方法未使用任何参数且方法本身是一个函数，则方法组将解释为带有空参数列表的调用表达式，而结果将用作带有提供的参数列表的调用表达式的目标。例如：

Class C1  
 Function M1() As Integer()  
 Return New Integer() { 1, 2, 3 }  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Prints 3  
 Console.WriteLine(c.M1(2))  
 End Sub  
End Module

否则，重载解析将应用于方法以便为给定参数列表选择最适用的方法。如果此最适用的方法是一个函数，则调用表达式的结果将归类为一个类型化为函数的返回类型的值。如果此最适用的方法是一个子例程，则结果将归类为 void。如果最适用的方法是一个没有主体的分部方法，则忽略调用表达式，并且结果将归类为 void。

对于早期绑定调用表达式，将按照在目标方法中声明相应形参的顺序来计算实参。对于后期绑定成员访问表达式，将按照它们在成员访问表达式中出现的顺序对它们进行计算。请参见 ‎11.3 一节的“后期绑定表达式”。

InvocationExpression ::= Expression [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]

ArgumentList ::=  
 PositionalArgumentList |  
 PositionalArgumentList Comma NamedArgumentList | NamedArgumentList

PositionalArgumentList ::=  
 [ Expression ] |  
 PositionalArgumentList Comma [ Expression ]

NamedArgumentList ::=  
 IdentifierOrKeyword ColonEquals Expression |  
 NamedArgumentList Comma IdentifierOrKeyword ColonEquals Expression

### 重载的方法解析

实际上，用于确定重载决策的规则将查找与提供的实际实参“最接近”的重载。如果有一个方法的形参类型与实参类型匹配，则此方法显然是最接近的。除此之外，如果一个方法的所有参数类型的宽度都小于或等于另一个方法的参数类型的宽度，则前者比后者更接近。如果其中任一方法的形参宽度都大于另一个方法的形参宽度，则无法确定哪一个方法更接近实参。

注意   重载解析不考虑方法的预期返回类型。

另请注意，由于存在命名参数语法，实参和形参的顺序可能会不同。

假定有一个方法组，可使用以下步骤来确定该组中最适用于参数列表的方法。如果在执行某个特定步骤后，组中未保留任何成员，则将发生编译时错误。如果组中仅保留一个成员，则该成员为最适用的成员。这些步骤是：

1. 首先（如果未提供类型实参）对具有类型形参的方法应用类型推断。如果对某个方法成功应用类型推断，则为该特定方法使用推断出的类型实参。如果对某个方法应用类型推断失败，则从组中清除该方法。
2. 然后从组中清除不可访问的或不适用于 (11.8.2) 参数列表的所有成员。
3. 接下来，如果一个或多个实参为 AddressOf 或 lambda 表达式，则如下计算每个这种实参的*委托松弛级别*。如果 N 中的最差（最低）委托松弛级别差于 M 中的最低委托松弛级别，则从集中清除 N。  
     
   委托松弛级别如下：
4. *(1) 错误* *委托松弛级别* - 如果不能将 AddressOf 或 lambda 转换为委托类型。  
     
   *(2) 返回类型或形参的收缩转换委托松弛* - 如果实参为 AddressOf 或带有已声明类型的 lambda，并且从其返回类型到委托返回类型的转换是收缩转换；或者，如果实参为常规 lambda，并且从其任何返回表达式到委托返回类型的转换是收缩转换；或者，如果该实参是异步 lambda 且委托返回类型是 Task(Of T)，并且从其任何返回表达式到 T 的转换是收缩转换；或者，如果该实参是迭代器 lambda 以及委托返回类型 IEnumerator(Of T) 或 IEnumerable(Of T)，并且从其任何 yield 操作数到 T 的转换为收缩转换。  
     
   *(3) 至不带签名的委托的扩大委托松弛* - 如果委托类型为 System.Delegate 或 System.MultiCastDelegate 或 System.Object。  
     
   *(4) 放置返回或实参委托松弛* - 如果实参为 AddressOf 或带有已声明返回类型的 lambda，并且该委托类型没有返回类型；或者，如果实参为带有一个或更多返回表达式的 lambda，并且该委托类型无返回类型；或者，如果该实参为 AddressOf 或不带形参的 lambda，并且该委托类型具有形参。  
     
   *(5) 返回类型的扩大委托松弛* - 如果实参为 AddressOf 或带有已声明返回类型的 lambda，并且存在从其返回类型到该委托的返回类型的扩大转换；或者，如果实参是常规 lambda，其中，从所有返回表达式到该委托返回类型的转换为扩大转换或具有至少一个扩大转换的标识；或者，如果实参为异步 lambda 且委托为 Task(Of T) 或 Task，并且从所有返回表达式到 T/Object 的转换分别为扩大转换或具有至少一个扩大转换的标识；或者，如果实参为迭代器 lambda 且委托为 IEnumerator(Of T) 或 IEnumerable(Of T) 或 IEnumerator 或 IEnumerable，并且从所有返回表达式到 T/Object 的转换是扩大转换或具有至少一个扩大转换的标识。  
     
   *(6) 标识委托松弛* - 如果实参为 AddressOf 或与该委托精确匹配的 lambda，不带扩大或收缩转换、形参放置、返回或生成。接下来，如果集的某些成员不需要将收缩转换应用于任何实参，则清除执行这种转换的所有成员。例如：

Sub f(x As Object)  
End Sub

Sub f(x As Short)  
End Sub

Sub f(x As Short())  
End Sub

f("5") ' picks the Object overload, since String->Short is narrowing  
f(5) ' picks the Object overload, since Integer->Short is narrowing  
f({5}) ' picks the Object overload, since Integer->Short is narrowing  
f({}) ' a tie-breaker rule subsequent to [3] picks the Short() overload

1. 下一步，将基于收缩如下执行清除。（请注意，如果 Option Strict 为 On，则需要收缩的所有成员都已被判断为不适用 (11.8.2) 并由步骤 2 移除。）
   1. 如果该集的某些实例成员只需要实参表达式类型为 Object 的收缩转换，则清除所有其他成员。
   2. 如果该集包含多个需要仅从 Object 收缩的成员，则调用目标表达式会重新分类为后期绑定方法访问（如果包含该方法组的类型是接口，或者任何适用的成员是扩展成员，则会发生错误）。
   3. 如果存在任何仅需要从数字文本收缩的候选项，则按下面的步骤，在所有剩余候选项中选择最特定的候选项。如果入选项仅需要从数字文本收缩，则将会作为重载决策的结果来选择该项；否则就是一个错误。

批注

使用此规则的理由是，当程序是松散类型化程序（即，将大多数或所有变量声明为 Object）时，如果来自 Object 的许多转换都是收缩转换，执行重载解析会很困难。与其让重载解析在多数情况下失败（要求对方法调用的参数进行强类型化），不如将对要调用的适当重载方法的解析推迟到运行时进行。这将成功执行松散类型化调用，而无需其他强制转换。

但不幸的是，这会带来副作用，即，执行后期绑定调用需要将调用目标强制转换为 Object。对于结构值来说，这意味着必须将该值装箱为临时值。如果最后调用的方法尝试更改结构的字段，则所作更改将在此方法返回后丢失。

此特殊规则不包括接口，这是因为后期绑定始终对运行时类或结构类型的成员进行解析，而这些成员具有的名称可能与其实现的接口的成员的名称不同。

1. 接下来，如果任何实例方法仍保留在不需要收缩的集中，则从该集中清除所有扩展方法。例如：

Imports System.Runtime.CompilerServices  
  
Class C3  
 Sub M1(d As Integer)  
 End Sub  
End Class  
  
Module C3Extensions  
 <Extension> \_  
 Sub M1(c3 As C3, c As Long)  
 End Sub  
  
 <Extension> \_  
 Sub M1(c3 As C3, c As Short)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim c As New C3()  
 Dim sVal As Short = 10  
 Dim lVal As Long = 20  
  
 ' Calls C3.M1, since C3.M1 is applicable.  
 c.M1(sVal)  
  
 ' Calls C3Extensions.M1 since C3.M1 requires a narrowing conversion  
 c.M1(lVal)  
 End Sub  
End Module

批注

如果有适用的实例方法，则将忽略扩展方法，以确保添加导入（这可能会将新的扩展方法引入到范围中）不会导致对现有实例方法的调用被重新绑定到扩展方法。如果为一些扩展方法（如在接口和/或类型参数上定义的扩展方法）给定扩大范围，则通过此方式绑定到扩展方法会更为安全。

1. 接下来，对于该集的任何两个成员 M 和 N，如果 M 在实参列表方面比 N 更加*特定*（11.8.1.1 节），则从该集中清除 N。如果组中保留了多个成员，并且根据参数列表这些成员并不同等具体，则将发生编译时错误。
2. 否则，如果给定了任意两个成员 M 和 N，则按顺序应用以下附加规则：

If M does not have a ParamArray parameter but N does, or if both do but M passes fewer arguments into the ParamArray parameter than N does, then eliminate N from the set. For example:Module Test  
 Sub F(a As Object, ParamArray b As Object())  
 Console.WriteLine("F(Object, Object())")  
 End Sub  
  
 Sub F(a As Object, b As Object, ParamArray c As Object())  
 Console.WriteLine("F(Object, Object, Object())")  
 End Sub  
  
 Sub G(Optional a As Object = Nothing)  
 Console.WriteLine("G(Object)")  
 End Sub  
  
 Sub G(ParamArray a As Object())  
 Console.WriteLine("G(Object())")  
 End Sub Sub Main()  
 F(1)  
 F(1, 2)  
 F(1, 2, 3)  
 G()  
 End Sub  
End Module

上面的示例产生如下输出：

F(Object, Object())  
F(Object, Object, Object())  
F(Object, Object, Object())  
G(Object)

批注

当一个类声明带有 paramarray 参数的方法时，同时包括与常规方法相同的一些展开形式，这种情况很常见。通过这种方式，可以避免分配数组实例（若调用带有 paramarray 参数的方法的展开形式，则无法避免）。

1. 如果与 N 相比，M 是在派生程度更大的类型中声明的，则从组中清除 N。例如：

Class Base  
 Sub F(Of T, U)(x As T, y As U)  
 End Sub  
End Class  
  
Class Derived  
 Inherits Base  
  
 Overloads Sub F(Of T, U)(x As U, y As T)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim d As New Derived()  
  
 ' Calls Derived.F  
 d.F(10, 10)  
 End Sub  
End Module

此规则也适用于在其上定义扩展方法的类型。例如：

Imports System.Runtime.CompilerServices  
  
Class Base  
End Class  
  
Class Derived  
 Inherits Base  
End Class  
  
Module BaseExt  
 <Extension> \_  
 Sub M(b As Base, x As Integer)  
 End Sub  
End Module  
  
Module DerivedExt  
 <Extension> \_  
 Sub M(d As Derived, x As Integer)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim b As New Base()  
 Dim d As New Derived()  
  
 ' Calls BaseExt.M  
 b.M(10)  
  
 ' Calls DerivedExt.M   
 d.M(10)  
 End Sub  
End Module

1. 如果 M 和 N 都是扩展方法，M 的目标类型为一个类或结构，而 N 的目标类型为一个接口，则从组中清除 N。例如：

Imports System.Runtime.CompilerServices  
  
Interface I1  
End Interface  
  
Class C1  
 Implements I1  
End Class  
  
Module Ext1  
 <Extension> \_  
 Sub M(i As I1, x As Integer)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M(c As C1, y As Integer)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim c As New C1()  
  
 ' Calls Ext2.M, because Ext1.M is hidden since it extends  
 ' an interface.  
 c.M(10)  
  
 ' Calls Ext1.M  
 CType(c, I1).M(10)  
 End Sub  
End Module

1. 如果 M 和 N 是扩展方法，并且 M 和 N 的目标类型在类型形参替换后相同，而 M 在类型形参替换前的目标类型不包含类型形参，但 N 的目标类型包含类型形参，则 M 的目标类型的类型形参少于 N 的目标类型的类型形参，则从该集中清除 N。例如：

Imports System.Runtime.CompilerServices  
  
Module Module1  
 Sub Main()  
 Dim x As Integer = 1  
 x.f(1) ' Calls first "f" extension method

Dim y As New Dictionary(Of Integer, Integer)  
 y.g(1) ' Ambiguity error  
 End Sub

<Extension()> Sub f(x As Integer, z As Integer)  
 End Sub

<Extension()> Sub f(Of T)(x As T, z As T)  
 End Sub  
  
 <Extension()> Sub g(Of T)(y As Dictionary(Of T, Integer), z As T)  
 End Sub

<Extension()> Sub g(Of T)(y As Dictionary(Of T, T), z As T)  
 End Sub  
End Module

1. 在替换类型实参之前，如果 M 不如 N *泛型化*（11.8.1.2 节），则从该集中清除 N。
2. 如果 M 不是扩展方法，而 N 是扩展方法，则从该集中清除 N。
3. 如果 M 和 N 是扩展方法，并且在 N 之前发现了 M（11.6.3 节），则从该集中清除 N。例如：

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Namespace N1  
 Module N1C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, x As Integer)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2  
 Module N2C1Extensions  
 <Extension> \_  
 Sub M1(c As C1, y As Integer)  
 End Sub  
 End Module  
End Namespace  
  
Namespace N1.N2.N3  
 Module Test  
 Sub Main()  
 Dim x As New C1()  
  
 ' Calls N2C1Extensions.M1  
 x.M1(10)  
 End Sub  
 End Module  
End Namespace

如果在同一个步骤中找到了多个扩展方法，则这些扩展方法是不明确的。该调用总是会产生歧义：使用包含扩展方法的标准模块的名称，并按照常规成员来调用扩展方法。例如：

Imports System.Runtime.CompilerServices  
  
Class C1  
End Class  
  
Module C1ExtA  
 <Extension> \_  
 Sub M(c As C1)  
 End Sub  
End Module  
  
Module C1ExtB  
 <Extension> \_  
 Sub M(c As C1)  
 End Sub  
End Module  
  
Module Main  
 Sub Test()  
 Dim c As New C1()  
  
 C1.M() ' Ambiguous between C1ExtA.M and BExtB.M  
 C1ExtA.M(c) ' Calls C1ExtA.M  
 C1ExtB.M(c) ' Calls C1ExtB.M  
 End Sub  
End Module

1. 如果 M 和 N 都需要使用类型推理来生成类型参数，其中 M 不需要确定其任何类型参数（即，每个将被推断为同一个类型的类型参数）的主导类型，而 N 需要，则从组中清除 N。

批注

此规则可确保在以前的版本（在这些版本中，若为一个类型参数推断多个类型，则将导致发生错误）中成功进行的重载解析仍产生相同的结果。

1. 如果正在执行重载解析以从 AddressOf 表达式解析委托-创建表达式的目标，且委托和 M 都是函数，而 N 是一个子例程，则从组中清除 N。同样，如果委托和 M 都是子例程，而 N 是函数，则从该集中清除 N。
2. 如果 M 没有使用任何可选形参默认值来代替显式实参，但 N 使用了可选形参默认值，则从该集中清除 N。
3. 在替换类型实参之前，如果与 N 相比 M 具有*更大的泛型深度*（11.8.1.3 节），则从该集中清除 N。
4. 其他情况下，此调用将具有多义性，并将发生编译时错误。

#### 给定实参列表时成员/类型的特异性

给定实参列表 A 时，如果成员 M 和 N 的签名相同，或者 M 中的每个形参类型与 N 中的对应形参类型相同，则将成员 M 视为与 N 同等特定。

批注

这两个成员会因扩展方法导致在方法组中结束，并带有相同的签名。这两个成员也可能因类型参数或参数数组展开导致具有同等具体性，但不具有相同的签名。

如果成员 M 和 N 的签名不同，并且 M 中的至少一个形参类型比 N 中的一个形参类型更加特定，且 N 中没有形参类型比 M 中的一个形参类型更加特定，则将成员 M 视为比 N 更加特定。假设有一对匹配实参 Aj 的形参 Mj 和 Nj，如果满足下列条件之一，则将 Mj 的类型视为比 Nj 的类型更具体：

1. 已存在从 Mj 的类型到 Nj 的类型的扩大转换，或

批注

请注意，由于在此情况下对形参类型进行的比较与实际的实参无关，因此不考虑从常量表达式到值所适合的数值类型的扩大转换。

1. Aj 为文本 0，Mj 为数值类型且 Nj 为枚举类型，或

批注

由于文本 0 扩大为任何枚举类型，因此该规则是必需的。枚举类型会扩大为其基础类型，这意味着在默认情况下，对 0 的重载解析将首选使用枚举类型，而不是数值类型。我们收到了许多指明此行为不够直观的反馈。

1. Mj 和 Nj 都是数值类型，并且在下面的列表中，Mj 排在 Nj 前面  
   Byte、SByte、Short、UShort、Integer、UInteger、Long、ULong、Decimal、Single、Double

批注

由于具有特定大小的有符号数值类型和无符号数字类型之间只具有收缩转换，因此有关数值类型的规则是有用的。上述规则将中断这两个类型之间的联系，而改为使用更“普通”的数值类型。在对扩大为具有特定大小的有符号数值类型和无符号数值类型的某个类型（例如，适合于这两个类型的数值文本）执行重载解析时，这样做特别重要。

1. Mj 和 Nj 是委托函数类型，且 Mj 的返回类型比 Nj 的返回类型更加特定。如果 Aj 归类为 lambda 方法，且 Mj 或 Nj 是 System.Linq.Expressions.Expression(Of T)，则该类型（假定是委托类型）的类型实参将替换正在比较的类型。
2. Mj 与 Aj 的类型相同，而 Nj 则不然。

批注

有一点需要注意，上述规则在 C# 中和 Visual Basic 中会稍有不同。在 C# 中，此规则要求委托函数类型在与返回类型进行比较之前具有相同的参数列表；而在 Visual Basic 中，此规则没有此项要求。

#### 一般性

按照以下规则确定成员 M 的一般性低于成员 N 的一般性：

1. 如果在每对匹配的参数 Mj 和 Nj 中，就方法的类型参数而言，Mj 的一般性低于或等于 Nj 的一般性，且至少有一个 Mj 的一般性较低。
2. 否则，如果在每对匹配的参数 Mj 和 Nj 中，就类型的类型参数而言，Mj 的一般性低于或等于 Nj 的一般性，且至少有一个 Mj 的一般性较低，则 M 的一般性低于 N 的一般性。

如果参数 M 和参数 N 的类型 Mt 和 Nt 都引用类型参数或都不引用类型参数，则认为两个参数的一般性相等。如果 Mt 不引用类型参数，而 Nt 引用类型参数，则将参数 M 的一般性视为低于 N 的一般性。

例如：

Class C1(Of T)  
 Sub S1(Of U)(x As U, y As T)  
 End Sub  
  
 Sub S1(Of U)(x As U, y As U)  
 End Sub  
  
 Sub S2(x As Integer, y As T)  
 End Sub  
  
 Sub S2(x As T, y As T)  
 End Sub  
End Class  
  
Module Test  
 Sub Main()  
 Dim x As C1(Of Integer) = New C1(Of Integer)  
  
 x.S1(10, 10) ' Calls S1(U, T)  
 x.S2(10, 10) ' Calls S2(Integer, T)  
 End Sub  
End Module

扩充过程中不变的扩展方法类型参数将视为类型的类型参数，而不是方法的类型参数。例如：

Imports System.Runtime.CompilerServices  
  
Module Ext1  
 <Extension> \_  
 Sub M1(Of T, U)(x As T, y As U, z As U)  
 End Sub  
End Module  
  
Module Ext2  
 <Extension> \_  
 Sub M1(Of T, U)(x As T, y As U, z As T)  
 End Sub  
End Module  
  
Module Test  
 Sub Main()  
 Dim i As Integer = 10  
  
 i.M1(10, 10)  
 End Sub  
End Module

#### 泛型深度

如果对于每对匹配的形参 Mj 和 Nj，Mj 的*泛型深度* 都大于或等于 Nj 的泛型深度，且至少一个 Mj 具有更大的泛型深度，则会确定成员 M 具有比成员 N 更大的泛型深度。泛型深度的定义如下：

1. 类型形参以外的任何项都具有比类型形参更大的泛型深度；
2. 以递归方式，如果至少一个类型实参具有比另一构造类型中的对应类型实参更大的泛型深度，并且没有类型实参具有比对应类型实参更小的深度，则一个构造类型就会具有比另一构造类型（具有相同数目的类型实参）更大的泛型深度。
3. 如果一个数组类型的元素类型具有比另一个数组类型的元素类型更大的泛型深度，则第一个数组类型的泛型深度大于第二个数组类型（维数相同）的泛型深度。

例如：

Module Test

Sub f(Of T)(x As Task(Of T))  
 End Sub

Sub f(Of T)(x As T)  
 End Sub

Sub Main()  
 Dim x As Task(Of Integer) = Nothing  
 f(x) ' Calls the first overload  
 End Sub  
End Module

### 实参列表适用性

如果一个方法可以通过使用两个参数列表来进行调用，则此方法适用于一组类型参数、位置参数和命名参数。实参列表按如下方式针对形参列表进行匹配：

1. 首先，将每个位置实参按顺序与一列方法形参进行匹配。如果位置实参数多于形参数，且最后一个形参不是 paramarray，则方法不适用。否则，paramarray 形参将与 paramarray 元素类型的形参一起展开以匹配位置实参的数量。如果省略将进入到形参数组中的位置实参，则该方法不适用。
2. 然后，将每个命名实参与具有给定名称的形参进行匹配。如果其中某个命名实参无法匹配，而匹配了一个 paramarray 形参或匹配了一个实参（此实参已经与另一个位置实参或命名实参匹配），则方法不适用。
3. 接下来，如果已指定类型实参，则针对类型形参列表匹配这些类型实参。如果这两个列表具有的元素数量不相同，则除非类型实参列表为空，否则方法不适用。如果类型实参列表为空，则类型推断用于尝试推断类型实参列表。如果类型推断失败，则该方法不适用。否则，类型实参将替代位置形参填在签名中。如果不匹配的形参不是可选形参，则该方法不适用。
4. 如果实参表达式不能隐式转换为它们所匹配的形参的类型，则该方法不适用。
5. 如果形参是 ByRef，且不存在从形参类型到实参类型的隐式转换，则该方法不适用。
6. 如果类型实参违反了方法约束（包括步骤 3 中的推断类型实参），则该方法不适用。例如：

Module Module1  
 Sub Main()  
 f(Of Integer)(New Exception)  
 ' 选取第一个重载（收缩转换），  
 ' 因为第二个重载（扩大转换）违反约束   
 End Sub  
  
 Sub f(Of T)(x As IComparable)  
 End Sub

Sub f(Of T As Class)(x As Object)  
 End Sub  
End Module

如果一个实参表达式与一个 paramarray 形参匹配，并且此实参表达式的类型能够转换为该 paramarray 形参的类型和 paramarray 元素类型，则该方法适用于其展开形式和未展开形式，但有以下两种例外情况。如果从实参表达式的类型到 paramarray 类型的转换是收缩转换，则该方法仅适用于其展开形式。如果实参表达式为文本 Nothing，则只有该方法的未扩展形式适用。例如：

Module Test  
 Sub F(ParamArray a As Object())  
 Dim o As Object  
  
 For Each o In a  
 Console.Write(o.GetType().FullName)  
 Console.Write(" ")  
 Next o  
 Console.WriteLine()  
 End Sub  
  
 Sub Main()  
 Dim a As Object() = { 1, "Hello", 123.456 }  
 Dim o As Object = a  
  
 F(a)  
 F(CType(a, Object))  
 F(o)  
 F(CType(o, Object()))  
 End Sub  
End Module

上面的示例产生如下输出：

System.Int32 System.String System.Double  
System.Object[]  
System.Object[]  
System.Int32 System.String System.Double

在 F 的第一个和最后一个调用中，F 的一般形式将适用，这是因为存在从实参类型到形参类型（二者都属于类型 Object()）的扩大转换，并且实参是作为常规值形参传递的。在第二个和第三个调用中，F 的一般形式将不适用，这是因为不存在从实参类型到形参类型的扩大转换（从 Object 到 Object() 的转换是收缩转换）。但 F 的扩展形式将适用，并且调用会创建一个元素 Object()。可使用给定实参值初始化数组的单个元素（它本身是对 Object() 的引用）。

### 为可选形参传递实参并选择实参

如果形参是一个值形参，则匹配的实参表达式必须归类为一个值。此值将转换为形参的类型，并在运行时作为形参传入。如果形参是一个引用形参且匹配的实参表达式归类为一个变量（其类型与形参的类型相同），则对变量的引用将在运行时作为形参传入。

否则，如果匹配的实参表达式归类为一个变量、值或属性访问，则分配形参的类型的临时变量。在运行时进行方法调用之前，实参表达式将重新归类为一个值，再转换为形参的类型，然后被赋给临时变量。紧接着，对临时变量的引用将作为形参传入。在计算方法调用之后，如果实参表达式归类为一个变量或属性访问，则临时变量将赋给变量表达式或属性访问表达式。如果属性访问表达式不具有 Set 访问器，则不进行此赋值。

对于未提供实参的可选形参，编译器按如下所述选择实参。在所有情况下，编译器会在泛型类型替换后对形参类型进行测试。

* 1. 如果可选形参具有特性 System.Runtime.CompilerServices.CallerLineNumber，并且调用是来自源代码中的某个位置，且表示该位置的行号的数字文本可内部转换为该形参类型，则将使用该数字文本。如果该调用跨多行，则选择要使用哪一行将取决于具体实现。
  2. 如果可选形参具有特性 System.Runtime.CompilerServices.CallerFilePath，并且调用是来自源代码中的某个位置，且表示该位置的文件路径的字符串文本可内部转换为该形参类型，则将使用该字符串文本。文件路径的格式取决于具体实现。
  3. 如果可选形参具有特性 System.Runtime.CompilerServices.CallerMemberName，并且该调用位于类型成员体内或位于应用于该类型成员的任何部分的某个特性中，且表示该成员名称的字符串文本可内部转换为该形参类型，则使用该字符串文本。对于作为属性访问器或自定义事件处理程序一部分的调用，所使用的成员名称是该属性或事件本身的名称。对于作为运算符或构造函数一部分的调用，将使用取决于具体实现的名称。

如果上面几项都不适用，则将使用可选形参的默认值（如果未提供默认值，则为 Nothing）。如果上面多项适用，则选择要使用哪一项将取决于具体实现。

批注

CallerLineNumber 和 CallerFilePath 特性对于日志记录十分有用。CallerMemberName 对于实现 INotifyPropertyChanged 十分有用。以下是几个示例。

Sub Log(msg As String,  
 <CallerFilePath> Optional file As String = Nothing,  
 <CallerLineNumber> Optional line As Integer? = Nothing)  
 Console.WriteLine("{0}:{1} - {2}", file, line, msg)  
End Sub

WriteOnly Property p As Integer  
 Set(value As Integer)  
 Notify(\_p, value)  
 End Set  
End Property

Private \_p As Integer

Sub Notify(Of T As IEquatable(Of T))(ByRef v1 As T, v2 As T,  
 <CallerMemberName> Optional prop As String = Nothing)  
 If v1 IsNot Nothing AndAlso v1.Equals(v2) Then Return  
 If v1 Is Nothing AndAlso v2 Is Nothing Then Return  
 v1 = v2  
 RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(prop))  
End Sub

批注

除了上面的可选形参外，如果某些其他可选形参是从元数据导入的（即从 DLL 引用导入），则 Microsoft Visual Basic 也会识别这些可选形参。从元数据导入时，Visual Basic 还会将参数 <Optional> 视为可选参数的标志：这样，有可能导入具有可选参数但没有默认值的声明，即使这不能使用 Optional 关键字表示。

1. 如果可选形参具有特性 Microsoft.VisualBasic.CompilerServices.OptionCompareAttribute，且数字文本 1 或 0 可转换为形参类型，那么，如果 Option Compare Text 有效，则编译器将文本 1 用作实参；如果 Optional Compare Binary 有效，则将文本 0 用作实参。
2. 如果可选形参具有特性 System.Runtime.CompilerServices.IDispatchConstantAttribute，且其类型为 Object，并且它未指定默认值，则编译器使用实参 New System.Runtime.InteropServices.DispatchWrapper(Nothing)。
3. 如果可选形参具有特性 System.Runtime.CompilerServices.IUnknownConstantAttribute，且其类型为 Object，并且它未指定默认值，则编译器使用实参 New System.Runtime.InteropServices.UnknownWrapper(Nothing)。
4. 如果可选形参的类型为 Object，并且它未指定默认值，则编译器使用实参 System.Reflection.Missing.Value。

### 条件方法

如果调用表达式引用的目标方法是一个子例程（此子例程不是接口的成员），而且该方法具有一个或多个 System.Diagnostics.ConditionalAttribute 特性，则根据源文件中此时定义的条件编译常量来计算表达式。该特性的每个实例均指定一个字符串，用于命名条件编译常量。每个条件编译常量被视为条件编译语句的一部分进行计算。如果此常量的计算结果为 True，则在运行时通常会计算表达式。如果此常量的计算结果为 False，则根本不会计算表达式。

在查找该特性时，将检查可重写方法的派生程度最大的声明。

注意   该特性对于函数或接口方法无效，当对任一类方法指定该特性时，该特性将被忽略。因此，条件方法只会在调用语句中出现。

### 类型实参推断

当不指定类型实参而调用具有类型形参的方法时，类型实参推断用于为该调用推断类型实参。在能够推断类型实参的情况下，这将允许更常用的语法用于调用具有类型形参的方法。例如，给定下面的方法声明：

Module Util  
 Function Choose(Of T)(b As Boolean, first As T, second As T) As T  
 If b Then  
 Return first  
 Else  
 Return second  
 End If  
 End Function  
End Class

可以在不显式指定类型实参的情况下调用 Choose 方法：

' calls Choose(Of Integer)  
Dim i As Integer = Util.Choose(True, 5, 213)  
' calls Choose(Of String)  
Dim s As String = Util.Choose(False, "a", "b")

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 Integer 和 String。

类型实参推理会在对实参列表中的 lambda 方法或方法指针执行表达式重新分类之前发生，这是因为对这两类表达式进行重新分类可能需要获知形参的类型。给定一组实参 (A1, A2, …, AN)、一组匹配的形参 (P1, P2, …, PN) 和一组方法类型形参 (T1, T2, …, TN)，首先按如下方式收集实参和方法类型形参之间的依赖项：

如果 AN 是文本 Nothing，则不生成任何依赖项。

如果 AN 是 lambda 方法，且 PN 的类型是构造委托类型或 System.Linq.Expressions.Expression(Of T)（其中 T 是构造委托类型），

如果将从对应的形参 PN 来推断 lambda 方法形参的类型，且形参的类型依赖于方法类型形参 TN，则 AN 依赖于 TN。

如果指定了 lambda 方法形参的类型，且对应的形参 PN 的类型依赖于方法类型形参 TN，则 TN 依赖于 AN。

如果 PN 的返回类型依赖于方法类型形参 TN，则 TN 依赖于 AN。

在 AN 是方法指针且 PN 的类型是构造委托类型的情况下：

如果 PN 的返回类型依赖于方法类型形参 TN，则 TN 依赖于 AN。

如果 PN 是构造类型且 PN 的类型依赖于方法类型形参 TN，则 TN 依赖于 AN。

否则，不生成任何依赖项。

在收集依赖项后，将清除任何不具有依赖项的实参。如果任何方法类型形参都不具有传出依赖项（即方法类型形参不依赖于实参），则类型推断将失败。否则，将剩余的实参和方法类型形参组合到强连接的组件中。强连接的组件是一组实参和方法类型形参，该组件中的任何元素均可通过它在其他元素上的依赖项到达。

然后，按拓扑顺序对强连接的组件进行排序和处理：

在强类型组件只包含一个元素的情况下：

如果该元素标记为已完成，则跳过它。

如果该元素是一个实参，则将此实参中的类型提示添加到依赖于此实参的方法类型形参，并将该元素标记为已完成。如果此实参是一个具有仍需要推断类型的形参的 lambda 方法，则为这些形参的类型推断 Object。

如果该元素是一个方法类型形参，则将此方法类型形参推断为实参类型提示中的主导类型，并将该元素标记为已完成。如果已对类型提示设置数组元素约束，则仅考虑给定类型的数组之间的有效转换（即协变和内部数组转换）。如果已对类型提示设置泛型实参约束，则仅考虑标识转换。如果无法选择主导类型，则推断将失败。如果任何 lambda 方法实参类型依赖于该方法类型形参，则将此类型传播给 lambda 方法。

如果强类型组件包含多个元素，则该组件包含一个循环。

对于作为该组件中元素的每个方法类型形参而言，如果方法类型形参依赖于未标记为已完成的实参，则将该依赖项转换为一个在推断过程结束时将检查的断言。

在确定强类型组件时重新启动推断过程。

如果对所有方法类型形参成功执行类型推断，则将检查已更改为断言的所有依赖项。如果实参的类型能够隐式转换为方法类型形参的推断类型，则断言成功。如果断言失败，则类型实参推断将失败。

假设为实参 A 给定实参类型 TA，并为形参 P 给定形参类型 TP，则按如下方式生成类型提示：

如果 TP 不涉及任何方法类型形参，则不会生成提示。

如果 TP 和 TA 是具有相同秩的数组类型，则将 TA 和 TP 分别替换为 TA 和 TP 的元素类型，然后在设置数组元素限制的情况下重新启动此过程。

如果 TP 是方法类型形参，则将 TA 添加为具有当前限制的类型提示（如果有）。

如果 A 是 lambda 方法，且 TP 是构造委托类型或 System.Linq.Expressions.Expression(Of T)（其中 T 是构造委托类型），则对于每个 lambda 方法形参类型 TL 以及对应的委托形参类型 TD，将 TA 替换为 TL，将 TP 替换为 TD，并不受限制地重新启动进程。然后，将 TA 替换为该 lambda 方法的返回类型，并且：

如果 A 是常规 lambda 方法，则将 TP 替换为该委托类型的返回类型；

如果 A 是异步 lambda 方法，并且对于某些 T，该委托类型的返回类型的形式为 Task(Of T)，则将 TP 替换为该 T；

如果 A 是迭代器 lambda 方法，并且对于某些 T，该委托类型的返回类型的形式为 IEnumerator(Of T) 或 IEnumerable(Of T)，则将 TP 替换为该 T。

下一步，不受任何限制地重新启动进程。

如果 A 是方法指针且 TP 是构造委托类型，则使用 TP 的形参类型来确定哪个方法指针最适用于 TP。如果有一个最适用的方法，则将 TA 替换为该方法的返回类型，并将 TP 替换为该委托类型的返回类型，并不受限制地重新启动进程。

否则，TP 必须为构造类型。假定 TG 为 TP 的泛型类型，

如果 TA 为 TG、继承自 TG 或刚好实现类型 TG 一次，则对于每个匹配的类型实参（TA 中的 TAX 和 TP 中的 TPX），将 TA 替换为 TAX，并将 TP 替换为 TPX，然后在设置泛型实参限制的情况下重新启动此过程。

否则，对泛型方法进行的类型推断将失败。

类型推断成功本身不保证方法是适用的。

## 索引表达式

索引表达式会产生数组元素或将属性组重新归类到属性访问中。索引表达式由（按先后顺序）表达式、左括号、索引实参列表和右括号构成。索引表达式的目标必须归类为一个属性组或值。按以下方式处理索引表达式：

如果目标表达式归类为一个值并且其类型不是数组类型、Object 或 System.Array，则该类型必须具有默认属性。为表示该类型的所有默认属性的属性组建立索引。虽然在 Visual Basic 中声明无参数默认属性是无效的，但其他语言可能允许声明此类属性。因此，允许对不带有实参的属性建立索引。

如果此表达式产生一个数组类型的值，则实参列表中的实参数必须与该数组类型的秩相同，并且不能包含命名实参。如果在运行时有任何索引无效，则会引发 System.IndexOutOfRangeException 异常。每个表达式都必须能够隐式转换为类型 Integer。索引表达式的结果为指定索引处的变量并被归类为一个变量。

如果该表达式归类为一个属性组，则重载解析用于确定是否有一个属性适用于索引实参列表。如果属性组只包含一个具有 Get 访问器的属性，并且该访问器未使用任何实参，则属性组将被解释为一个带有空实参列表的索引表达式。结果将用作当前索引表达式的目标。如果没有适用的属性，则将发生编译时错误。否则，表达式将产生一个属性访问，该属性访问带有属性组的关联的目标表达式（如果有）。

如果该表达式归类为一个后期绑定属性组或一个值（其类型为 Object 或 System.Array），则对索引表达式的处理将推迟到运行时进行，并且索引是后期绑定的。该表达式将产生一个类型化为 Object 的后期绑定属性访问。关联的目标表达式为目标表达式（如果它是一个值）或属性组的关联的目标表达式。在运行时，按以下方式处理该表达式：

如果该表达式归类为一个后期绑定属性组，则该表达式可能会产生一个方法组、属性组或值（如果成员是一个实例或共享变量）。如果结果是一个方法组或属性组，则对组执行重载解析以确定实参列表的正确方法。如果重载决策失败，则会引发 System.Reflection.AmbiguousMatchException 异常。然后，将结果作为属性访问或调用进行处理，并返回相应结果。如果调用属于子例程，则结果为 Nothing。

如果目标表达式的运行时类型是一个数组类型或 System.Array，则索引表达式的结果为指定索引处的变量的值。

否则，该表达式的运行时类型必须具有默认属性，并对表示此类型的所有默认属性的属性组建立索引。如果此类型不具有默认属性，则将引发 System.MissingMemberException 异常。

IndexExpression ::= Expression OpenParenthesis [ ArgumentList ] CloseParenthesis

## New 表达式

New 运算符用于创建类型的新实例。有四种形式的 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（不具有与对象创建表达式不同的语法）用于创建委托类型的新实例。

匿名对象创建表达式用于创建匿名类类型的新实例。

New 表达式归类为一个值，并且结果为该类型的新实例。

NewExpression ::=  
 ObjectCreationExpression |  
 ArrayExpression |  
 AnonymousObjectCreationExpression

### 对象创建表达式

对象创建表达式用于创建类类型或结构类型的新实例。对象创建表达式的类型必须是一个类类型、结构类型或带有 New 约束的类型参数，但不能是 MustInherit 类。假定有一个形式为 New T(A) 的对象创建表达式，其中 T 为类类型或结构类型，A 为可选实参列表，则重载解析将确定要调用的 T 的正确构造函数。带有 New 约束的类型形参将被视为具有一个无参数的构造函数。如果没有可调用的构造函数，则将发生编译时错误；否则，该表达式将导致使用选定构造函数创建 T 的新实例。如果没有参数，则可以省略括号。

一个实例的分配位置取决于该实例是一个类类型还是值类型。类类型的 New 实例是在系统堆上创建的，而值类型的新实例是直接在堆栈上创建的。

对象创建表达式可以选择在构造函数实参的后面指定一个成员初始值设定项的列表。这些成员初始值设定项以关键字 With 作为前缀，并将初始值设定项列表视为在 With 语句的上下文中进行解释。例如，假定有以下类：

Class Customer  
 Dim Name As String  
 Dim Address As String  
End Class

代码：

Module Test  
 Sub Main()  
 Dim x As New Customer() With { .Name = "Bob Smith", \_  
 .Address = "123 Main St." }  
 End Sub  
End Module

大致等效于：

Module Test  
 Sub Main()  
 Dim x, \_t1 As Customer  
  
 \_t1 = New Customer()  
 With \_t1  
 .Name = "Bob Smith"  
 .Address = "123 Main St."  
 End With  
  
 x = \_t1  
 End Sub  
End Module

每个初始值设定项必须指定一个要赋值的名称，该名称必须是正在构建的类型的非 ReadOnly 实例变量或属性；如果正在构建的类型为 Object，则成员访问将不是后期绑定的。初始值设定项不能使用 Key 关键字。类型中的每个成员只能被初始化一次。但初始值设定项表达式可以相互引用。例如：

Module Test  
 Sub Main()  
 Dim x As New Customer() With { .Name = "Bob Smith", \_  
 .Address = .Name & " St." }  
 End Sub  
End Module

由于将按照从左至右的顺序对初始值设定项进行赋值，因此如果一个初始值设定项引用尚未进行初始化的成员，则该初始值设定项将获知构造函数后面的实例变量所运行的任何值：

Module Test  
 Sub Main()  
 ' The value of Address will be " St." since Name has not been  
 ' assigned yet.  
 Dim x As New Customer() With { .Address = .Name & " St." }  
 End Sub  
End Module

初始值设定项可以嵌套：

Class Customer  
 Dim Name As String  
 Dim Address As Address  
 Dim Age As Integer  
End Class  
  
Class Address  
 Dim Street As String  
 Dim City As String  
 Dim State As String  
 Dim ZIP As String  
End Class  
  
Module Test  
 Sub Main()  
 Dim c As New Customer() With { \_  
 .Name = "John Smith", \_  
 .Address = New Address() With { \_  
 .Street = "23 Main St.", \_  
 .City = "Peoria", \_  
 .State = "IL", \_  
 .ZIP = "13934" }, \_  
 .Age = 34 }  
 End Sub  
End Module

如果正在创建的类型是一个集合类型并具有名为 Add 的实例方法（包括扩展方法），则对象创建表达式可以指定一个以关键字 From 为前缀的集合初始值设定项。对象创建表达式不能同时指定成员初始值设定项和集合初始值设定项。集合初始值设定项中的每个元素都将作为一个实参传递给 Add 函数的调用。例如：

Dim list = New List(Of Integer)() From { 1, 2, 3, 4 }

等效于：

Dim list = New List(Of Integer)()  
list.Add(1)  
list.Add(2)  
list.Add(3)

如果元素是集合初始值设定项本身，则子集合初始值设定项的每个元素将作为一个实参传递给 Add 函数。例如，以下：

Dim dict = Dictionary(Of Integer, String) From { { 1, "One" },{ 2, "Two" } }

等效于：

Dim dict = New Dictionary(Of Integer, String)  
dict.Add(1, "One")  
dict.Add(2, "Two")

始终会进行此扩展，并且仅扩展一层；此后，将把子初始值设定项视为数组文本。例如：

' Error: List(Of T) does not have an Add method that takes two parameters.  
Dim list = New List(Of Integer())() From { { 1, 2 }, { 3, 4 } }  
  
' OK, this initializes the dictionary with (Integer, Integer()) pairs.  
Dim dict = New Dictionary(Of Integer, Integer())() From \_  
 { { 1, { 2, 3 } }, { 3, { 4, 5 } } }

ObjectCreationExpression ::=  
 New NonArrayTypeName [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]   
 [ ObjectCreationExpressionInitializer ]

ObjectCreationExpressionInitializer ::= ObjectMemberInitializer | ObjectCollectionInitializer

ObjectMemberInitializer ::=  
 With OpenCurlyBrace FieldInitializerList CloseCurlyBrace

FieldInitializerList ::=  
 FieldInitializer |  
 FieldInitializerList Comma FieldInitializer

FieldInitializer ::= [ [ Key ] . IdentifierOrKeyword Equals ] Expression

ObjectCollectionInitializer ::= From CollectionInitializer

CollectionInitializer ::= OpenCurlyBrace [ CollectionElementList ] CloseCurlyBrace

CollectionElementList ::=  
 CollectionElement |  
 CollectionElementList Comma CollectionElement

CollectionElement ::=  
 *Expression* |  
 *CollectionInitializer*

### 数组表达式

数组表达式用于创建数组类型的新实例。有两种类型的数组表达式：数组创建表达式和数组文本。

#### 数组创建表达式

如果提供了数组大小初始化修饰符，则通过从数组大小初始化参数列表中删除每个参数来派生结果数组类型。每个参数的值确定新分配的数组实例中相应维度的上限。如果表达式包含非空集合初始值设定项，则参数列表中的每个参数必须为常量，并且表达式列表指定的秩和维度长度必须匹配集合初始值设定项的秩和维度长度。

Dim a() As Integer = New Integer(2) {}  
Dim b() As Integer = New Integer(2) { 1, 2, 3 }  
Dim c(,) As Integer = New Integer(1, 2) { { 1, 2, 3 } , { 4, 5, 6 } }  
  
' Error, length/initializer mismatch.  
Dim d() As Integer = New Integer(2) { 0, 1, 2, 3 }

如果未提供数组大小初始化修饰符，则类型名称必须是数组类型，并且集合初始值设定项必须为空或具有与指定数组类型的秩相同的嵌套层数。最里面的嵌套层中的所有元素必须能够隐式转换为数组的元素类型且必须归类为值。每个嵌套的集合初始值设定项中的元素数必须始终与同一级别的其他集合的大小保持一致。各维度长度从集合初始值设定项的每个对应嵌套层数中的元素数推断出。如果集合初始值设定项为空，则每个维度的长度为零。

Dim e() As Integer = New Integer() { 1, 2, 3 }  
Dim f(,) As Integer = New Integer(,) { { 1, 2, 3 } , { 4, 5, 6 } }  
  
' Error: Inconsistent numbers of elements!  
Dim g(,) As Integer = New Integer(,) { { 1, 2 }, { 4, 5, 6 } }  
  
' Error: Inconsistent levels of nesting!  
Dim h(,) As Integer = New Integer(,) { 1, 2, { 3, 4 } }

集合初始值设定项的最外面的嵌套层对应于数组的最左边的维度，而最里面的嵌套层对应于最右边的维度。示例：

Dim array As Integer(,) = \_  
 { { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7 }, { 8, 9 } }

等效于下面的内容：

Dim array(4, 1) As Integer  
  
array(0, 0) = 0: array(0, 1) = 1  
array(1, 0) = 2: array(1, 1) = 3  
array(2, 0) = 4: array(2, 1) = 5  
array(3, 0) = 6: array(3, 1) = 7  
array(4, 0) = 8: array(4, 1) = 9

如果集合初始值设定项为空（也就是说，它包含大括号但不包含初始值设定项列表），并且正在初始化的数组的维度的界限未知，则空集合初始值设定项表示一个具有指定大小的数组实例，其中所有元素已初始化为元素类型的默认值。如果正在初始化的数组的维度的界限未知，则空集合初始值设定项表示一个其包含的所有维度大小都为零的数组实例。

在一个数组实例的整个生存期内，该数组实例的秩和每个维度的长度都是常量。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。

#### 数组文本

数组文本表示一个数组，该数组的元素类型、秩和边界是结合表达式上下文和集合初始值设定项这二者的情况推断出来的。在 ‎11.1.1 一节“表达式的重新分类”中对此进行了说明。例如：

' array of integers  
Dim a = {1, 2, 3}

' array of shorts  
Dim b = {1S, 2S, 3S}

' array of shorts whose type is taken from the context  
Dim c As Short() = {1, 2, 3}

' array of type Integer(,)  
Dim d = {{1, 0}, {0, 1}}

' jagged array of rank ()()  
Dim e = {({1, 0}), ({0, 1})}

' error: inconsistent rank  
Dim f = {{1}, {2, 3}}

' error: inconsistent rank  
Dim g = {1, {2}}

数组文本中集合初始值设定项的格式与要求与数组创建表达式中集合初始值设定项的格式与要求完全相同。

批注

数组文本本身不创建数组，而是将表达式重新归类为一个可以导致创建数组的值。例如，由于没有从 Integer() 到 Short() 的转换，因此不能进行 CType(new Integer() {1,2,3}, Short()) 转换；但由于表达式 CType({1,2,3},Short()) 先将数组文本重新分类为数组创建表达式 New Short() {1,2,3}，因此可使用此表达式。

ArrayExpression ::= *ArrayCreationExpression* | *ArrayLiteralExpression*

*ArrayCreationExpression*  ::=  
 New NonArrayTypeName ArrayNameModifier CollectionInitializer

ArrayLiteralExpression ::=  
 CollectionInitializer

### 委托创建表达式

委托创建表达式用于创建委托类型的新实例。委托创建表达式的参数必须是一个归类为方法指针或 lambda 方法的表达式。

如果此参数是一个方法指针，则在该方法指针所引用的方法中，必须有一个方法适用于委托类型的签名。在以下情况下，方法 M 适用于委托类型 D：

M 不是 Partial，或者具有主体。

M 和 D 都是函数，或者 D 是子例程。

M 和 D 具有相同的参数数。

M 的每个参数类型都可以从 D 的对应参数类型的类型进行转换，并且这些参数类型的修饰符（即，ByRef 和 ByVal）相匹配。

M 的返回类型（如果有）可以转换为 D 的返回类型。

如果方法指针引用了一个后期绑定访问，则将该后期绑定访问假定为一个具有与委托类型相同的参数数的函数。

如果未使用严格语义且方法指针只引用了一个方法，但由于该方法不具有参数而委托类型具有参数导致该方法不适用，则将该方法视为适用并忽略参数或返回值。例如：

Delegate Sub F(x As Integer)  
  
Module Test  
 Sub M()  
 End Sub  
  
 Sub Main()  
 ' Valid  
 Dim x As F = AddressOf M  
 End Sub  
End Module

批注

由于扩展方法的存在，导致只能在未使用严格语义时才能放宽上述原则。由于仅在常规方法不适用时才考虑扩展方法，因此不带参数的实例方法可以隐藏带参数的扩展方法以便实现委托构造。

如果方法指针所引用的多个方法都适用于委托类型，则重载解析用于在这些候选方法中选择最适用的方法。为了进行重载解析，委托形参的类型将用作实参的类型。如果这些候选方法中没有最适用的方法，则将发生编译时错误。在下面的示例中，将通过引用第二个 Square 方法的委托来初始化局部变量，这是因为该方法更适用于 DoubleFunc 的签名和返回类型。

Delegate Function DoubleFunc(x As Double) As Double  
  
Module Test  
 Function Square(x As Single) As Single  
 Return x \* x  
 End Function   
  
 Function Square(x As Double) As Double  
 Return x \* x  
 End Function  
  
 Sub Main()  
 Dim a As New DoubleFunc(AddressOf Square)  
 End Sub  
End Module

如果第二个 Square 方法不存在，则会选择第一个 Square 方法。如果编译环境或 Option Strict 指定了严格语义，则当方法指针所引用的最具体的方法窄于委托签名时，会发生编译时错误。在以下情况下，方法 M 将被视为窄于委托类型 D：

M 的参数类型可以扩大转换到 D 的对应参数类型。

或者，M 的返回类型（如果有）可以收缩转换到 D 的返回类型。

如果类型参数与方法指针关联，则仅考虑与类型参数一样多的方法。如果没有类型参数与方法指针关联，则在对泛型方法匹配签名时使用类型推断。与其他一般类型推断不同，在推断类型参数时将使用委托的返回类型，但在确定最低泛型重载使仍不会考虑返回类型。下面的示例展示了两种向委托创建表达式提供类型参数的方法：

Delegate Function D(s As String, i As Integer) As Integer  
Delegate Function E() As Integer  
  
Module Test  
 Public Function F(Of T)(s As String, t1 As T) As T  
 End Function  
   
 Public Function G(Of T)() As T  
 End Function  
  
 Sub Main()  
 Dim d1 As D = AddressOf f(Of Integer) ' OK, type arg explicit  
 Dim d2 As D = AddressOf f ' OK, type arg inferred  
  
 Dim e1 As E = AddressOf g(Of Integer) ' OK, type arg explicit  
 Dim e2 As E = AddressOf g ' OK, infer from return  
 End Sub  
End Module

在上面的示例中，已使用泛型方法来实例化非泛型委托类型。也可以使用泛型方法来创建构造委托类型的实例。例如：

Delegate Function Predicate(Of U)(u1 As U, u2 As U) As Boolean  
  
Module Test  
 Function Compare(Of T)(t1 As List(of T), t2 As List(of T)) As Boolean  
 ...  
 End Function  
   
 Sub Main()  
 Dim p As Predicate(Of List(Of Integer))  
 p = AddressOf Compare(Of Integer)  
 End Sub  
End Module

如果委托创建表达式的参数是一个 lambda 方法，则该 lambda 方法必须适用于委托类型的签名。在以下情况下，lambda 方法 L 适用于委托类型 D：

如果 L 和 D 具有相同数目的参数。（如果 L 不具有参数，则忽略 D 的参数。）

L 的每个参数类型都可以转换为 D 的对应参数类型的类型，并且这些参数类型的修饰符（即，ByRef 和 ByVal）相匹配。

如果 D 是函数，则 L 的返回类型可以转换为 D 的返回类型。（如果 D 是子例程，则忽略 L 的返回值。）

如果省略 L 的某个参数的参数类型，则推断 D 中对应参数的类型；如果 L 的参数具有数组或可以为 null 的名称修饰符，则将发生编译时错误。一旦 L 的所有参数类型可用，则推断 lambda 方法中的表达式的类型。例如：

Delegate Function F(x As Integer, y As Long) As Long  
  
Module Test  
 Sub Main()  
 ' b inferred to Integer, c and return type inferred to Long  
 Dim a As F = Function(b, c) b + c  
  
 ' e and return type inferred to Integer, f inferred to Long  
 Dim d As F = Function(e, f) e + CInt(f)  
 End Sub  
End Module

在某些情况下，委托签名未与 lambda 方法或方法签名完全匹配，此时 .NET Framework 可能不支持本机委托创建。在此情况下，lambda 方法表达式用于匹配这两个方法。例如：

Delegate Function IntFunc(x As Integer) As Integer  
  
Module Test  
 Function SquareString(x As String) As String  
 Return CInt(x) \* CInt(x)  
 End Function   
  
 Sub Main()  
 ' The following two lines are equivalent  
 Dim a As New IntFunc(AddressOf SquareString)  
 Dim b As New IntFunc( \_  
 Function(x As Integer) CInt(SquareString(CStr(x))))  
 End Sub  
End Module

委托创建表达式的结果是一个委托实例，该委托实例引用与方法指针表达式中的关联的目标方法（如果有）相匹配的方法。如果目标表达式类型化一个值类型，则该值类型将复制到系统堆上，这是因为委托只能指向堆上某个对象的方法。在委托的整个生存期内，委托所引用的方法和对象保持不变。换句话说，在创建委托后，将无法更改其目标或对象。

### 匿名对象创建表达式

带有成员初始值设定项的对象创建表达式也可以完全省略类型名称。在此情况下，根据作为表达式的一部分进行初始化的成员的类型和名称来构造匿名类型。例如：

Module Test  
 Sub Main()  
 Dim Customer = New With { .Name = "John Smith", .Age = 34 }  
  
 Console.WriteLine(Customer.Name)  
 End Sub  
End Module

匿名对象创建表达式创建的类型是一个类，该类不具有名称、直接从 Object 继承并且具有一组属性，这些属性具有与成员初始值设定项列表中赋值的成员相同的名称。每个属性的类型将通过使用与局部变量类型推理相同的规则来进行推断。生成的匿名类型还会重写 ToString，返回所有成员及其值的字符串表示形式。（字符串的准确格式不属于本规范的讨论范围）。

默认情况下，匿名类型生成的属性是可读写的。可以使用 Key 修饰符将匿名类型属性标记为只读。Key 修饰符指定字段可用于唯一标识匿名类型表示的值。除了使属性只读外，它还造成匿名类型重写 Equals 和 GetHashCode，并实现接口 System.IEquatable(Of T)（填入 T 的匿名类型）。按如下方式定义这些成员：

Function Equals(obj As Object) As Boolean 和 Function Equals(val As T) As Boolean 的实现方式是验证两个实例是否为同一类型，然后使用 Object.Equals 比较每个 Key 成员。如果所有 Key 成员均相等，则 Equals 返回 True，否则 Equals 返回 False。

Function GetHashCode() As Integer 按以下方式实现：如果 Equals 对于该匿名类型的两个实例均为 true，则 GetHashCode 将返回相同的值。哈希处理首先从一个种子值开始，然后依次为每个 Key 成员执行以下计算：将哈希值乘以 31 后再加上 Key 成员的哈希值（由 GetHashCode 提供）；但前提是该成员不是引用类型或可以为 null 的值类型（其值为 Nothing）。

例如，以下语句中创建的类型：

Dim zipState = New With { Key .ZipCode = 98112, .State = "WA" }

创建与下面类似的类（虽然具体实现方式可能不同）：

Friend NotInheritable Class $Anonymous1  
 Implements IEquatable(Of $Anonymous1)  
  
 Private ReadOnly \_zipCode As Integer  
 Private \_state As String  
  
 Public Sub New(zipCode As Integer, state As String)  
 \_zipCode = zipcode  
 \_state = state  
 End Sub  
  
 Public ReadOnly Property ZipCode As Integer  
 Get  
 Return \_zipCode  
 End Get  
 End Property  
  
 Public Property State As String  
 Get  
 Return \_state  
 End Get  
 Set (value As Integer)  
 \_state = value  
 End Set  
 End Property  
  
 Public Overrides Function Equals(obj As Object) As Boolean  
 Dim val As $Anonymous1 = TryCast(obj, $Anonymous1)  
 Return Equals(val)  
 End Function  
  
 Public Overloads Function Equals(val As $Anonymous1) As Boolean \_  
 Implements IEquatable(Of $Anonymous1).Equals  
  
 If val Is Nothing Then   
 Return False  
 End If  
  
 If Not Object.Equals(\_zipCode, val.\_zipCode) Then   
 Return False  
 End If  
  
 Return True  
 End Function  
  
 Public Overrides Function GetHashCode() As Integer  
 Dim hash As Integer = 0  
  
 hash = hash Xor \_zipCode.GetHashCode()  
  
 Return hash  
 End Function  
  
 Public Overrides Function ToString() As String  
 Return "{ Key .ZipCode = " & \_zipCode & ", .State = " & \_state & " }"  
 End Function  
End Class

为了简化从另一个类型的字段创建匿名类型这一情况，在以下情况下可以直接从表达式推断字段类型：

简单名称表达式 x 推断名称 x。

成员访问表达式 x.y 推断名称 y。

字典查找表达式 x!y 推断名称 y。

不带参数的调用或索引表达式 x() 推断名称 xx。

XML 成员访问表达式 x.<y>, x...<y>, x.@y 推断名称 y。

作为成员访问表达式 x.<y>.z 的目标的 XML 成员访问表达式推断名称 z。

作为不带参数的调用或索引表达式 x.<y>.z() 的目标的 XML 成员访问表达式推断名称 z。

作为调用或索引表达式 x.<y>(0) 的目标的 XML 成员访问表达式推断名称 y。

初始值设定项将解释为对推断名称分配表达式。例如，下面的初始值设定项是等效的：

Class Address  
 Public Street As String  
 Public City As String  
 Public State As String  
 Public ZIP As String  
End Class  
  
Class C1  
 Sub Test(a As Address)  
 Dim cityState1 = New With { .City = a.City, .State = a.State }  
 Dim cityState2 = New With { a.City, a.State }  
 End Sub  
End Class

如果推断的成员名称与类型的现有成员（如 GetHashCode）发生冲突，则将发生编译时错误。与常规成员初始值设定项不同，匿名对象创建表达式不允许成员初始值设定项具有循环引用或在完成初始化之前引用成员。例如：

Module Test  
 Sub Main()  
 ' Error: Circular references  
 Dim x = New With { .a = .b, .b = .a }  
  
 ' Error: Referring to .b before it has been assigned to  
 Dim y = New With { .a = .b, .b = 10 }  
  
 ' Error: Referring to .a before it has been assigned to  
 Dim z = New With { .a = .a }  
 End Sub  
End Module

如果同一方法中出现两个匿名类创建表达式并且产生同一结果形状（在属性顺序、属性名称和属性形状全都匹配的情况下），则这两个表达式将同时引用同一匿名类。带初始值设定项的实例或共享成员变量的方法范围是在其中初始化变量的构造函数。

批注

编译器可以选择将匿名类型进一步地统一（如在程序集级别上），但此时不能依赖此操作。

AnonymousObjectCreationExpression ::=  
 New ObjectMemberInitializer

## 强制转换表达式

强制转换表达式用于将表达式强制转换为给定类型。特定的强制转换关键字会将表达式强制转换为基元类型。CType、TryCast 和 DirectCast 这三个常规强制转换关键字会将表达式强制转换为一个类型。

DirectCast 和 TryCast 具有特殊行为。为此，它们仅支持本机转换。此外，TryCast 表达式中的目标类型不能为值类型。当使用 DirectCast 或 TryCast 时，将不考虑用户定义的转换运算符。

批注

DirectCast 和 TryCast 所支持的转换集是受限制的，因为它们实现的是“本机 CLR”转换。DirectCast 用于提供“unbox”指令的功能，而 TryCast 用于提供“isinst”指令的功能。由于它们会映射到 CLR 指令上，因此如果支持的转换不受 CLR 直接支持，则将无法实现二者的预期用途。

DirectCast 会按照与 CType 不同的方式转换类型化为 Object 的表达式。在转换类型为 Object 的表达式（其运行时类型为基元值类型）时，如果指定的类型不同于表达式的运行时类型，则 DirectCast 将引发 System.InvalidCastException 异常；如果表达式的计算结果为 Nothing，则将引发 System.NullReferenceException 异常。

批注

如上所述，当表达式的类型为 Object 时，DirectCast 将直接映射到 CLR 指令“unbox”上。相反，CType 将变成对运行时帮助器的调用来执行转换，以便支持基元类型之间的转换。在此情况下，当 Object 表达式转换为基元值类型，且实际实例的类型与目标类型相匹配时，DirectCast 执行转换的速度明显要快于 CType。

当 TryCast 转换表达式时，如果表达式无法转换为目标类型，将不会引发异常。相反，如果表达式在运行时无法转换，则 TryCast 将产生 Nothing。例如：

Interface ITest  
 Sub Test()  
End Interface  
  
Module Test  
 Sub Convert(o As Object)  
 Dim i As ITest = TryCast(o, ITest)  
  
 If i IsNot Nothing Then  
 i.Test()  
 End If  
 End Sub  
End Module

批注

如上所述，TryCast 将直接映射到 CLR 指令“isinst”上。通过将类型检查和转换合并到一个操作中，可使得执行 TryCast 所产生的开销比执行 TypeOf…Is 再执行 CType 所产生的开销要小一些。

如果不存在从表达式的类型到指定类型的转换，则将发生编译时错误。否则，表达式将归类为一个值，并且结果为转换所产生的值。

CastExpression ::=  
 DirectCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 TryCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CType OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CastTarget OpenParenthesis Expression CloseParenthesis

CastTarget ::=  
 CBool | CByte | CChar | CDate | CDec | CDbl | CInt | CLng | CObj | CSByte | CShort |  
 CSng | CStr | CUInt | CULng | CUShort

## 运算符表达式

运算符有两种。一元运算符带一个操作数并使用前缀表示法（如 –x）。二元运算符带两个操作数并使用中缀表示法（如 x + y）。为某个特定类型定义的运算符将会产生该类型，但关系运算符除外，此类运算符始终产生 Boolean。运算符的操作数必须始终归类为一个值；运算符表达式的结果将归类为一个值。

OperatorExpression ::=  
 ArithmeticOperatorExpression |  
 RelationalOperatorExpression |  
 LikeOperatorExpression |  
 ConcatenationOperatorExpression |  
 ShortCircuitLogicalOperatorExpression |  
 LogicalOperatorExpression |  
 ShiftOperatorExpression |  
 AwaitOperatorExpression

### 运算符的优先级和结合性

当表达式包含多个二元运算符时，运算符的优先级控制各二元运算符的计算顺序。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。下表按优先级的降序顺序列出了二元运算符：

|  |  |
| --- | --- |
| 类别 | 运算符 |
| 基本 | 所有非运算符表达式 |
| Await | Await |
| 幂 | ^ |
| 一元求反 | +, - |
| 乘法 | \*, / |
| 整除 | \ |
| 取模 | Mod |
| 加减 | +, - |
| 串联 | & |
| 移位 | <<, >> |
| 关系 | =, <>, <, >, <=, >=, Like, Is, IsNot |
| 逻辑“非” | Not |
| 逻辑“与” | And, AndAlso |
| 逻辑 OR | Or, OrElse |
| 逻辑 XOR | Xor |

当表达式包含的两个运算符具有相同优先级时，运算符的顺序关联性控制运算的执行顺序。所有的二元运算符都是向左关联的，这表示从左向右执行运算。优先级和顺序关联性都可以用带括号的表达式控制。

### 对象操作数

除了每个运算符支持的常规类型之外，所有运算符还支持 Object 类型的操作数。对应用于 Object 操作数的运算符的处理方式与对 Object 值执行方法调用的方式类似：可能选择后期绑定方法调用，在此情况下，将由操作数的运行时类型（而非编译时类型）确定运算的有效性和类型。如果编译环境或 Option Strict 指定了严格语义，则用于 Object 类型操作数的任何运算符都将导致编译时错误（TypeOf...Is、Is 和 IsNot 运算符除外）。

在运算符解析确定应以后期绑定方式执行运算的情况下，如果操作数的运行时类型是该运算符支持的类型，则运算的结果为对操作数类型应用运算符所得到的结果。值 Nothing 在二元运算符表达式中被视为另一个操作数的类型的默认值。在一元运算符表达式中，或者当二元运算符表达式中的两个操作数都为 Nothing 时，运算的类型为 Integer 或为运算符的唯一结果类型（如果运算符不产生 Integer）。然后，运算的结果将始终强制转换回 Object。如果操作数类型不具有有效运算符，则将引发 System.InvalidCastException 异常。运行时将执行转换，这与它们是隐式的还是显式的无关。

如果数值二元运算的结果将引发溢出异常（不管整数溢出检查是已启用还是已禁用），则结果类型将提升为下一个更宽的数值类型（如果可能）。例如，考虑以下代码：

Module Test  
 Sub Main()  
 Dim o As Object = CObj(CByte(2)) \* CObj(CByte(255))  
  
 Console.WriteLine(o.GetType().ToString() & " = " & o)  
 End Sub  
End Module

此代码将输出以下结果：

System.Int16 = 512

如果没有更宽的数值类型可用来容纳数值，则将引发 System.OverflowException 异常。

### 运算符解析

假定有一个运算符类型和一组操作数，运算符解析将确定对操作数使用哪一个运算符。在解析运算符时，将首先考虑用户定义的运算符，操作步骤如下：

首先，收集所有候选运算符。候选运算符包括源类型中的特定运算符类型的所有用户定义的运算符和目标类型中的特定类型的所有用户定义的运算符。如果源类型和目标类型相关，则只考虑常用运算符一次。

然后，对运算符和操作数应用重载解析以选择最具体的运算符。对于二元运算符，这种情况可能会导致后期绑定调用。

在为类型 T? 收集候选运算符时，将改用类型 T 的运算符。此外，还会提升 T 的任何用户定义的运算符（这些运算符只涉及不可以为 null 的值类型）。提升的运算符将使用任何值类型的可以为 null 的版本，但 IsTrue 和 IsFalse 的返回类型除外（它们必须是 Boolean）。计算提升的运算符的方式如下：将操作数转换为其不可为 null 的版本，再计算用户定义的运算符，然后将结果类型转换为其可为 null 的版本。如果任一操作数为 Nothing，则表达式的结果为 Nothing 的值（该值类型化为结果类型的可以为 null 的版本）。例如：

Structure T  
 ...  
End Structure  
  
Structure S  
 Public Shared Operator +(ByVal op1 As S, ByVal op2 As T) As T  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim x As S?  
 Dim y, z As T?  
  
 ' Valid, as S + T = T is lifted to S? + T? = T?  
 z = x + y   
 End Sub  
End Module

如果运算符是二元运算符，并且某个操作数是引用类型，则也会提升运算符，只不过对运算符进行任何绑定会导致发生错误。例如：

Structure S1  
 Public F1 As Integer  
  
 Public Shared Operator +(left As S1, right As String) As S1  
 ...  
 End Operator  
End Structure  
  
Module Test  
 Sub Main()  
 Dim a? As S1  
 Dim s As String  
  
 ' Error: '+' is not defined for S1? and String  
 a = a + s  
 End Sub  
End Module

批注

由于我们需要考虑是否应在将来版本中增加 null 传播引用类型，而在此情况下，这两个类型之间的二元运算符的行为将发生更改，因此需要设置此规则。

与转换一样，用户定义的运算符始终优先于提升的运算符。

在解析重载的运算符时，Visual Basic 中定义的类和其他语言中定义的类之间可能会有差异：

在其他语言中，Not、And 和 Or 可以作为逻辑运算符和按位运算符进行重载。在从外部程序集导入之后，对于这些运算符而言，任一形式都将作为一个有效重载被接受。但对于定义逻辑运算符和按位运算符的类型而言，只会考虑按位实现。

在其他语言中，>> 和 << 可以同时作为有符号运算符和无符号运算符进行重载。在从外部程序集导入之后，任一形式都将作为一个有效重载被接受。但对于定义有符号运算符和无符号运算符的类型而言，只会考虑有符号实现。

如果没有最特定于操作数的用户定义的运算符，则将考虑内部运算符。如果未针对操作数定义内部运算符，并且有任何一个操作数的类型为 Object，则该运算符将解析为后期绑定；否则会发生编译时错误。

批注  
  
在 Visual Basic 的早期版本中，如果有且仅有一个 Object 类型的操作数，而没有适用的用户定义运算符，也没有适用的内部运算符，则会发生错误。从 Visual Basic 11 起，运算符解析为后期绑定。例如：

Module Module1  
 Sub Main()  
 Dim p As Object = Nothing  
 Dim U As New Uri("http://www.microsoft.com")  
 Dim j = U \* p ' is now resolved late-bound  
 End Sub  
End Module

具有某个内部运算符的类型 T 也会为 T? 定义相同的运算符。T? 的运算符的结果将与 T 的相同，只不过当任一操作数为 Nothing 时，该运算符的结果将为 Nothing（即，传播 null 值）。为了解析运算的类型，将执行以下操作：从任何具有 ? 的操作数中删除它，再确定运算的类型，然后将 ? 添加到运算的类型（如果任何操作数是可为 null 的值类型）。例如：

Dim v1? As Integer = 10  
Dim v2 As Long = 20  
  
' Type of operation will be Long?  
Console.WriteLine(v1 + v2)

每个运算符列出了它所针对的内部类型和对给定操作数类型执行的运算的类型。内部运算的类型的结果遵循以下一般规则：

如果所有操作数都属于同一类型，并且为该类型定义了运算符，则不会发生任何转换并会使用该类型的运算符。

通过以下步骤转换任何操作数（其类型不是针对运算符定义的）并对新类型解析运算符：

将操作数转换为针对运算符和操作数定义的下一个最宽的类型，并且能够隐式转换到该类型。

如果没有此种类型，则将操作数转换为针对运算符和操作数定义的下一个最窄的类型，并且能够隐式转换到该类型。

如果没有此种类型或无法执行转换，则将发生编译时错误。

否则，操作数将转换为更宽的操作数类型，并使用该类型的运算符。如果更窄的操作数类型无法隐式转换为更宽的运算符类型，则将发生编译时错误。

尽管存在上述一般规则，但运算符结果表中还是列出了大量特殊情况。

注意 由于格式方面的原因，运算符类型表中的预定义名称已缩写为相应名称的前两个字符。因此，“By”表示 Byte，“UI”表示 UInteger，“St”表示 String，依此类推。“Err”表示没有为给定的操作数类型定义运算。

## 算术运算符

\*、/、\、^、Mod、+ 和 – 运算符都是算术运算符。

可以用比运算的结果类型更高的精度来执行浮点算术运算。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，才能使硬件架构用“较低”的精度执行浮点运算。Visual Basic 采取的是允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）。

ArithmeticOperatorExpression ::=  
 UnaryPlusExpression |  
 UnaryMinusExpression |  
 AdditionOperatorExpression |  
 SubtractionOperatorExpression |  
 MultiplicationOperatorExpression |  
 DivisionOperatorExpression |  
 ModuloOperatorExpression |  
 ExponentOperatorExpression

### 一元加运算符

为 Byte、SByte、UShort、Short、UInteger Integer、ULong、Long、Single、Double 和 Decimal 类型定义了一元加运算符。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Sh | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |

UnaryPlusExpression ::= + Expression

### 一元减运算符

为以下类型定义了一元减运算符：

SByte、Short、Integer 和 Long。通过从零中减去操作数来计算结果。如果启用整数溢出检查并且操作数的值是 SByte、Short、Integer 或 Long 的最大负值，则将引发 System.OverflowException 异常。否则，如果操作数的值是 SByte、Short、Integer 或 Long 的最大负值，则结果为同一个值并且不会报告溢出。

Single 和 Double。结果是操作数的值（其符号相反），包括值 0 和 Infinity。如果操作数为 NaN，则结果也为 NaN。

Decimal.通过从零中减去操作数来计算结果。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |

UnaryMinusExpression ::= - Expression

### 加法运算符

该加法运算符计算两个操作数的和。为以下类型定义了加法运算符：

Byte、SByte、UShort、Short、UInteger、Integer、ULong 和 Long。如果启用整数溢出检查并且得到的和不在结果类型的范围之内，则将引发 System.OverflowException 异常。否则，将不会报告溢出，并会丢弃结果的所有有效高序位。

Single 和 Double。根据 IEEE 754 算术运算法则计算和。

Decimal.如果结果值太大，无法用十进制格式表示，则将引发 System.OverflowException 异常。如果结果值太小，无法用十进制格式表示，则结果为 0。

String.两个 String 操作数将串连在一起。

注意 System.DateTime 类型定义了重载的加法运算符。由于 System.DateTime 等效于内部 Date 类型，因此这些运算符也可用于 Date 类型。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | Si | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | Si | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Si | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | St | Err | St | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | St | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

AdditionOperatorExpression ::= Expression + [ LineTerminator ] Expression

### 减法运算符

该减法运算符从第一个操作数中减去第二个操作数。为以下类型定义了减法运算符：

Byte、SByte、UShort、Short、UInteger、Integer、ULong 和 Long。如果启用整数溢出检查并且得到的差不在结果类型的范围之内，则将引发 System.OverflowException 异常。否则，将不会报告溢出，并会丢弃结果的所有有效高序位。

Single 和 Double。根据 IEEE 754 算术运算法则计算差。

Decimal.如果结果值太大，无法用十进制格式表示，则将引发 System.OverflowException 异常。如果结果值太小，无法用十进制格式表示，则结果为 0。

注意 System.DateTime 类型定义了重载的减法运算符。由于 System.DateTime 等效于内部 Date 类型，因此这些运算符也可用于 Date 类型。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | Si | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | Si | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Si | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

SubtractionOperatorExpression ::= Expression - [ LineTerminator ] Expression

### 乘法运算符

乘法运算符计算两个操作数的乘积。为以下类型定义了乘法运算符：

Byte、SByte、UShort、Short、UInteger、Integer、ULong 和 Long。如果启用整数溢出检查并且得到的积不在结果类型的范围之内，则将引发 System.OverflowException 异常。否则，将不会报告溢出，并会丢弃结果的所有有效高序位。

Single 和 Double。根据 IEEE 754 算术运算法则计算乘积。

Decimal.如果结果值太大，无法用十进制格式表示，则将引发 System.OverflowException 异常。如果结果值太小，无法用十进制格式表示，则结果为 0。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | Si | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | Si | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Si | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

MultiplicationOperatorExpression ::= Expression \* [ LineTerminator ] Expression

### 除法运算符

除法运算符计算两个操作数之商。有两类除法运算符：常规（浮点）除法运算符和整除运算符。

为以下类型定义了常规除法运算符：

Single 和 Double。根据 IEEE 754 算法法则计算商。

Decimal.如果右操作数的值为零，则引发 System.DivideByZeroException 导常。如果结果值太大，无法用十进制格式表示，则将引发 System.OverflowException 异常。如果结果值太小，无法用十进制格式表示，则结果为零。在进行任何舍入之前，结果的小数位数是与首选小数位数最接近的小数位数，这将保留等于准确结果的结果。首选小数位数是第一个操作数的小数位数与第二个操作数的小数位数之间的较小者。

根据一般运算符解析规则，在 Byte、Short、Integer 和 Long 等类型的操作数之间执行常规除法会导致两个操作数都转换为类型 Decimal。不过，在对除法运算符（在任一类型不为 Decimal 的情况下）执行运算符解析时，Double 被视为窄于 Decimal。由于 Double 除法比 Decimal 除法更高效，因此将遵循此约定。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Do | Do | Do | Do | Do | Do | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| SB |  | Do | Do | Do | Do | Do | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| By |  |  | Do | Do | Do | Do | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Do | Do | Do | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| US |  |  |  |  | Do | Do | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | Do | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | Do | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Do | Do | De | Si | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | Do | De | Si | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | Si | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Si | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

为 Byte、SByte、UShort、Short、UInteger、Integer、ULong 和 Long 定义了整数除法运算符。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，并且结果的绝对值是小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。如果左操作数是 SByte、Short、Integer 或 Long 的最大负数，且右操作数为 –1，则将发生溢出；如果启用整数溢出检查，则将引发 System.OverflowException 异常。否则，不会报告溢出，并且结果将改为左操作数的值。

批注

由于无符号类型的两个操作数总是为零或正值，因此结果始终为零或负值。由于表达式的结果总是小于或等于两个操作数中的较大者，因此不可能会发生溢出。由于未对两个无符号整数之间的整除执行整数溢出检查，因此，结果为左操作数的类型。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| In |  |  |  |  |  | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Lo |  |  |  |  |  |  |  | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UL |  |  |  |  |  |  |  |  | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| De |  |  |  |  |  |  |  |  |  | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Lo | Lo | Err | Err | Lo | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Lo | Err | Err | Lo | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Lo | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

DivisionOperatorExpression ::=  
 FPDivisionOperatorExpression |  
 IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= Expression / [ LineTerminator ] Expression

IntegerDivisionOperatorExpression ::= Expression \ [ LineTerminator ] Expression

### Mod 运算符

Mod（取模）运算符计算两个操作数相除后的余数。为以下类型定义了 Mod 运算符：

Byte、SByte、UShort、Short、UInteger、Integer、ULong 和 Long。 x Mod y 的结果是表达式 x – (x \ y) \* y 的值。如果 y 为零，则引发 System.DivideByZeroException 异常。取模运算符从不导致溢出。

Single 和 Double。根据 IEEE 754 算术运算法则计算余数。

Decimal.如果右操作数的值为零，则引发 System.DivideByZeroException 导常。如果结果值太大，无法用十进制格式表示，则将引发 System.OverflowException 异常。如果结果值太小，无法用十进制格式表示，则结果为零。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Sh | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | Si | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | Si | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Si | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ModuloOperatorExpression ::= Expression Mod [ LineTerminator ] Expression

### 幂运算符

幂运算符计算第一个操作数的第二个操作数次幂。为类型 Double 定义了幂运算符。根据 IEEE 754 算术运算法则计算值。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| SB |  | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| By |  |  | Do | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Do | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| US |  |  |  |  | Do | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | Do | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | Do | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Do | Do | Do | Do | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | Do | Do | Do | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | Do | Do | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Do | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Do | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ExponentOperatorExpression ::= Expression ^ [ LineTerminator ] Expression

## 关系运算符

关系运算符用于将值与其他值进行比较。比较运算符包括 =、<>、<、>、<= 和 >=。所有关系运算符都会产生 Boolean 值。

关系运算符具有以下一般含义：

= 运算符测试两个操作数是否相等。

<> 运算符测试两个操作数是否不相等。

< 运算符测试第一个操作数是否小于第二个操作数。

> 运算符测试第一个操作数是否大于第二个操作数。

<= 运算符测试第一个操作数是否小于或等于第二个操作数。

>= 运算符测试第一个操作数是否大于或等于第二个操作数。

为以下类型定义了关系运算符：

Boolean.运算符对两个操作数的真假值进行比较。True 被视为小于 False，这与它们的数值相符。

Byte、SByte、UShort、Short、UInteger、Integer、ULong 和 Long。该运算符比较两个整型操作数的数值。

Single 和 Double。这些运算符根据 IEEE 754 标准法则比较操作数。

Decimal.这些运算符比较两个 decimal 类型操作数的数值。

Date.这些运算符返回两个日期/时间值的比较结果。

Char.这些运算符返回两个 Unicode 值的比较结果。

String.这些运算符返回使用二进制比较或文本比较来比较两个值所得到的结果。编译环境和 Option Compare 语句将确定使用哪一类比较。二进制比较用于确定每个字符串中的每个字符的数值型 Unicode 值是否相等。文本比较根据中 .NET Framework 当前使用的区域性来执行 Unicode 文本比较。在执行字符串比较时，null 值等效于字符串文本 ""。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Bo | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Bo | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| In |  |  |  |  |  | In | Lo | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | De | Si | Do | Err | Err | Do | Ob |
| Lo |  |  |  |  |  |  |  | Lo | De | De | Si | Do | Err | Err | Do | Ob |
| UL |  |  |  |  |  |  |  |  | UL | De | Si | Do | Err | Err | Do | Ob |
| De |  |  |  |  |  |  |  |  |  | De | Si | Do | Err | Err | Do | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Si | Do | Err | Err | Do | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Do | Err | Err | Do | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Da | Err | Da | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Ch | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

RelationalOperatorExpression ::=  
 Expression = [ LineTerminator ] Expression |  
 Expression < > [ LineTerminator ] Expression |  
 Expression < [ LineTerminator ] Expression |  
 Expression > [ LineTerminator ] Expression |  
 Expression < = [ LineTerminator ] Expression |  
 Expression > = [ LineTerminator ] Expression

## Like 运算符

为 String 类型定义了 Like 运算符，用于确定字符串是否与给定模式相匹配。第一个操作数是正在匹配的字符串，第二个操作数是要与之匹配的模式。模式由 Unicode 字符构成。以下字符序列具有特殊的含义：

字符 ? 与任何单个字符相匹配。

字符 \* 与零个或多个字符相匹配。

字符 # 与任何单个数字 (0–9) 相匹配。

用方括号括起的字符列表 ([ab...]) 与列表中的任何单个字符相匹配。

用方括号括起的并以感叹号为前缀的字符列表 ([!ab...]) 与不在字符列表中的任何单个字符相匹配。

用连字符 (-) 分隔的字符列表中的两个字符指定一系列 Unicode 字符，这些字符以第一个字符开始并以第二个字符结束。如果第二个字符在排序顺序中未位于第一个字符的后面，则将发生运行时异常。在字符列表的开头或结尾出现的连字符指定其本身。

注意   若要与左方括号 ([)、问号 (?)、数字号 (#) 和星号 (\*) 这些特殊字符匹配，必须用方括号将它们括起。右方括号 (]) 不能在组中用来与自身匹配，但它可用在组外作为单个字符。可以将字符序列 [] 视为字符串文本 ""。

另请注意，字符列表的字符比较和排序取决于正在使用的比较的类型。如果使用的是二元比较，则根据数值 Unicode 值进行字符比较和排序。如果使用的是文本比较，则根据 .NET Framework 中正使用的当前区域设置来进行字符比较和排序。

在某些语言中，字母表中的特殊字符表示两种不同的字符，反之亦然。例如，有几种语言使用字符 æ 来表示字符 a 和 e（当这两个字符一起出现时），而字符 ˆ 和 O 可用于表示字符 Ô。在使用文本比较时，Like 运算符会识别此类区域性等效项。在此情况下，在模式或字符串中出现的单个特殊字符将匹配其他字符串中等效的双字符序列。与此类似，括在方括号内的模式中的单个特殊字符（独立存在、在列表中或在某个范围内）匹配字符串中等效的双字符序列，反之亦然。

在 Like 表达式中，如果两个操作数都是 Nothing，或者一个操作数可以内部转换为 String 而另一个操作数为 Nothing，则将 Nothing 视为空字符串文本 ""。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | St | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| SB |  | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| By |  |  | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| Sh |  |  |  | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| US |  |  |  |  | St | St | St | St | St | St | St | St | St | St | St | Ob |
| In |  |  |  |  |  | St | St | St | St | St | St | St | St | St | St | Ob |
| UI |  |  |  |  |  |  | St | St | St | St | St | St | St | St | St | Ob |
| Lo |  |  |  |  |  |  |  | St | St | St | St | St | St | St | St | Ob |
| UL |  |  |  |  |  |  |  |  | St | St | St | St | St | St | St | Ob |
| De |  |  |  |  |  |  |  |  |  | St | St | St | St | St | St | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | St | St | St | St | St | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | St | St | St | St | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | St | St | St | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | St | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

LikeOperatorExpression ::= Expression Like [ LineTerminator ] Expression

## 串联运算符

为所有内部类型定义了串联运算符，包括内部值类型的可为 null 的版本和 System.DBNull。同样也为上述类型与被视为 Nothing 字符串的 System.DBNull 之间的串联定义了此运算符。串联运算符将其所有操作数都转换为 String；在表达式中，所有到 String 的转换均被视为扩大转换，而不管是否使用了严格语义。System.DBNull 值将转换为类型化为 String 的文本 Nothing。此外，其值为 Nothing 的可为 null 的值类型将转换为类型化为 String 的文本 Nothing，而不是引发运行时错误。

串联运算符会产生一个字符串，该字符串由两个操作数按从左至右的顺序串联而成。值 Nothing 将被视为空字符串文本 ""。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | St | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| SB |  | St | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| By |  |  | St | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| Sh |  |  |  | St | St | St | St | St | St | St | St | St | St | St | St | Ob |
| US |  |  |  |  | St | St | St | St | St | St | St | St | St | St | St | Ob |
| In |  |  |  |  |  | St | St | St | St | St | St | St | St | St | St | Ob |
| UI |  |  |  |  |  |  | St | St | St | St | St | St | St | St | St | Ob |
| Lo |  |  |  |  |  |  |  | St | St | St | St | St | St | St | St | Ob |
| UL |  |  |  |  |  |  |  |  | St | St | St | St | St | St | St | Ob |
| De |  |  |  |  |  |  |  |  |  | St | St | St | St | St | St | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | St | St | St | St | St | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | St | St | St | St | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | St | St | St | Ob |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | St | St | Ob |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | St | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ConcatenationOperatorExpression ::= Expression & [ LineTerminator ] Expression

## 逻辑运算符

And、Not、Or 和 Xor 运算符称作逻辑运算符，计算方式如下：

对于 Boolean 类型：

对该类型的两个操作数执行逻辑 And 运算。

对该类型的操作数执行逻辑 Not 运算。

对该类型的两个操作数执行逻辑 Or 运算。

对该类型的两个操作数执行逻辑互斥 Or 运算。

对于 Byte、SByte、UShort、Short、UInteger、Integer、 ULong、Long 和所有枚举类型，将对两个操作数的二进制表示形式的每个位执行指定运算：

And：如果两个位都为 1，则结果位为 1；否则结果位为 0。

Not：如果位为 0，则结果位为 1；否则结果位为 1。

Or：如果任一位为 1，则结果位为 1；否则结果位为 0。

Xor：如果任一位为 1 而不是两个位都为 1，则结果位为 1；否则结果位为 0（也就是说，1 Xor 0 = 1，1 Xor 1 = 0）。

在为类型 Boolean? 提升逻辑运算符 And 和 Or 时，这两个运算符将扩展为包含三值 Boolean 逻辑，如下所示：

如果两个操作数都为 true，则 And 的计算结果为 true；如果其中一个操作数为 false，则计算结果为 false；否则为 Nothing。

如果任一操作数为 true，则 Or 的计算结果为 true；如果两个操作数都为 false，则计算结果为 false；否则为 Nothing。

例如：

Module Test  
 Sub Main()  
 Dim x?, y? As Boolean  
  
 x = Nothing  
 y = True   
  
 If x Or y Then  
 ' Will execute  
 End If  
 End Sub  
End Module

批注

理想情况下，将使用针对可在布尔表达式中使用的任何类型（即，实现 IsTrue 和 IsFalse 的类型）的三值逻辑来提升逻辑运算符 And 和 Or，其方式与在可用于布尔表达式的任何类型之间使用 AndAlso 和 OrElse 短路运算符一样。遗憾的是，由于三值提升仅适用于 Boolean?，因此，若用户定义的类型需要三值逻辑，则必须通过为其可为 null 的版本定义 And 和 Or 运算符，手动实现这一功能。

这些运算不可能产生溢出。虽然枚举类型运算符对枚举类型的基础类型执行按位运算，但返回值为枚举类型。

Not 运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |

And、Or、Xor 运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Bo | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Bo | Ob |
| SB |  | SB | Sh | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| By |  |  | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Sh |  |  |  | Sh | In | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| US |  |  |  |  | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| In |  |  |  |  |  | In | Lo | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UI |  |  |  |  |  |  | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Lo |  |  |  |  |  |  |  | Lo | Lo | Lo | Lo | Lo | Err | Err | Lo | Ob |
| UL |  |  |  |  |  |  |  |  | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |
| De |  |  |  |  |  |  |  |  |  | Lo | Lo | Lo | Err | Err | Lo | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Lo | Lo | Err | Err | Lo | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Lo | Err | Err | Lo | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Lo | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

LogicalOperatorExpression ::=  
 Not Expression |  
 Expression And [ LineTerminator ] Expression |  
 Expression Or [ LineTerminator ] Expression |  
 Expression Xor [ LineTerminator ] Expression

### 短路逻辑运算符

AndAlso 和 OrElse 运算符是 And 和 Or 逻辑运算符的短路版本。由于其短路行为的存在，导致如果在计算第一个操作数后获知运算符结果，则在运行时不计算第二个操作数。

按以下方式计算短路逻辑运算符：

如果 AndAlso 运算中的第一个操作数的计算结果为 False，或从其 IsFalse 运算符返回 True，则该表达式返回其第一个操作数。否则，计算第二个操作数并对得到的两个结果执行逻辑 And 运算。

如果 OrElse 运算中的第一个操作数的计算结果为 True，或从其 IsTrue 运算符返回 True，则该表达式返回其第一个操作数。否则，计算第二个操作数并对得到的两个结果执行逻辑 Or 运算。

为类型 Boolean 或任何重载以下运算符的类型 T 定义了 AndAlso 和 OrElse 运算符：

Public Shared Operator IsTrue(op As T) As Boolean  
Public Shared Operator IsFalse(op As T) As Boolean

同时还重载相应的 And 或 Or 运算符：

Public Shared Operator And(op1 As T, op2 As T) As T  
Public Shared Operator Or(op1 As T, op2 As T) As T

在计算 AndAlso 或 OrElse 运算符时，仅计算第一个操作数一次，而不计算第二个操作数或者只计算一次。例如，考虑以下代码：

Module Test  
 Function TrueValue() As Boolean  
 Console.Write(" True")  
 Return True  
 End Function  
  
 Function FalseValue() As Boolean  
 Console.Write(" False")  
 Return False  
 End Function  
  
 Sub Main()  
 Console.Write("And:")  
 If FalseValue() And TrueValue() Then  
 End If  
 Console.WriteLine()  
  
 Console.Write("Or:")  
 If TrueValue() Or FalseValue() Then  
 End If  
 Console.WriteLine()  
  
 Console.Write("AndAlso:")  
 If FalseValue() AndAlso TrueValue() Then  
 End If  
 Console.WriteLine()  
  
 Console.Write("OrElse:")  
 If TrueValue() OrElse FalseValue() Then  
 End If  
 Console.WriteLine()  
 End Sub  
End Module

此代码将输出以下结果：

And: False True  
Or: True False  
AndAlso: False  
OrElse: True

在 AndAlso 和 OrElse 操作符的提升形式中，如果第一个操作数是 null Boolean?，则将对第二个操作数进行计算，但结果始终为 null Boolean?。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| SB |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| By |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| Sh |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| US |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| In |  |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| UI |  |  |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| Lo |  |  |  |  |  |  |  | Bo | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| UL |  |  |  |  |  |  |  |  | Bo | Bo | Bo | Bo | Err | Err | Bo | Ob |
| De |  |  |  |  |  |  |  |  |  | Bo | Bo | Bo | Err | Err | Bo | Ob |
| Si |  |  |  |  |  |  |  |  |  |  | Bo | Bo | Err | Err | Bo | Ob |
| Do |  |  |  |  |  |  |  |  |  |  |  | Bo | Err | Err | Bo | Ob |
| Da |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err | Err |
| Ch |  |  |  |  |  |  |  |  |  |  |  |  |  | Err | Err | Err |
| St |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Bo | Ob |
| Ob |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Ob |

ShortCircuitLogicalOperatorExpression ::=  
 Expression AndAlso [ LineTerminator ] Expression |  
 Expression OrElse [ LineTerminator ] Expression

## 移位运算符

二元运算符 << 和 >> 用于执行移位运算。为 Byte、SByte、UShort、Short、UInteger、Integer、 ULong 和 Long 类型定义了运算符。与其他二元运算符不同，移位运算的返回类型将按照运算符为仅带左操作数的一元运算符的情况来确定。右操作数的类型必须能够隐式转换为 Integer，它不用于确定运算的结果类型。

<< 运算符可使第一个操作数中的位向左移动由移位量指定的位数。超出结果类型范围的高序位被放弃，空出的低序数位位置用 0 填充。

>> 运算符可使第一个操作数中的数位向右移动由移位量指定的位数。低序位被放弃，空出的高序数位位置设置为 0（如果左操作数为正）或 1（如果左操作数为负）。如果左操作数的类型为 Byte、UShort、UInteger 或 ULong，则空出的高序位用 0 填充。

移位运算符将按照第二个操作数所指定的移位量来移动第一个操作数的基础表示形式的数位。如果第二个操作数的值大于第一个操作数中的数位数或为负，则按照 RightOperand And SizeMask 形式来计算移位量，其中 SizeMask 为：

|  |  |
| --- | --- |
| LeftOperand 类型 | SizeMask |
| Byte, SByte | 7 (&H7) |
| UShort, Short | 15 (&HF) |
| UInteger, Integer | 31 (&H1F) |
| ULong, Long | 63 (&H3F) |

如果移位量为零，则运算的结果等于第一个操作数的值。这些运算不可能产生溢出。

运算类型：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bo | SB | By | Sh | US | In | UI | Lo | UL | De | Si | Do | Da | Ch | St | Ob |
| Sh | SB | By | Sh | US | In | UI | Lo | UL | Lo | Lo | Lo | Err | Err | Lo | Ob |

ShiftOperatorExpression ::=  
 Expression < < [ LineTerminator ] Expression |  
 Expression > > [ LineTerminator ] Expression

## 布尔表达式

可以对布尔表达式进行测试以查看它为 true 还是为 false。在以下情况（按优先顺序）中，可以在布尔表达式中使用类型 T：

T 是 Boolean 或 Boolean?

T可以扩大转换为Boolean

T可以扩大转换为Boolean?

T 定义了两个伪运算符：IsTrue 和 IsFalse。

T 可以收缩转换为 Boolean?，而不涉及从 Boolean 到 Boolean? 的转换。

T 可以收缩转换为 Boolean。

批注

需要值得注意的是，如果禁用了 Option Strict，则将接受可以收缩转换为 Boolean 的表达式，并且不会发生编译时错误，但如果 IsTrue运算符存在，则语言仍会优先选择该运算符。这是因为 Option Strict 只会更改语言接受的内容和不接受的内容，但绝不会更改表达式的实际含义。因此，IsTrue 必须总是优先于收缩转换，这与 Option Strict 无关。

例如，下面的类不定义到 Boolean 的扩大转换。因此，在 If 语句中使用该类会导致调用 IsTrue 运算符。

Class MyBool  
 Public Shared Widening Operator CType(b As Boolean) As MyBool  
 ...  
 End Operator  
  
 Public Shared Narrowing Operator CType(b As MyBool) As Boolean  
 ...  
 End Operator  
  
 Public Shared Operator IsTrue(b As MyBool) As Boolean  
 ...  
 End Operator  
  
 Public Shared Operator IsFalse(b As MyBool) As Boolean  
 ...  
 End Operator  
End Class  
  
Module Test  
 Sub Main()  
 Dim b As New MyBool  
  
 If b Then Console.WriteLine("True")  
 End Sub  
End Module

在将一个布尔表达式类型化为或转换为 Boolean 或 Boolean? 之后，如果值为 True，则该表达式为 true，否则表达式为 false。

否则，布尔表达式将调用 IsTrue 运算符，并在该运算符返回 True 时返回 True；否则该表达式为 false（但从不调用 IsFalse 运算符）。

在下面的示例中，Integer 收缩转换到 Boolean，因此，一个 null Integer? 可以收缩转换到 Boolean?（产生一个 null Boolean）和 Boolean（引发异常）。收缩转换到 Boolean? 为首选，因此，作为布尔表达式的“i”值为 False。

Dim i As Integer? = Nothing  
If i Then Console.WriteLine()

BooleanExpression ::= Expression

## lambda 表达式

lambda 表达式定义了一个称作 lambda 方法的匿名方法。利用 lambda 方法，可以将“内联”方法轻松传递到采用委托类型的其他方法。示例：

Module Test  
 Delegate Function IntFunc(x As Integer) As Integer  
  
 Sub Apply(a() As Integer, func As IntFunc)  
 For index As Integer = 0 To a.Length - 1  
 a(index) = func(a(index))  
 Next index  
 End Sub  
  
 Sub Main()  
 Dim a() As Integer = { 1, 2, 3, 4 }  
  
 Apply(a, Function(x As Integer) x \* 2)  
  
 For Each value In a  
 Console.Write(value & " ")  
 Next value  
 End Sub  
End Module

将输出：

2 4 6 8

lambda 表达式以可选修饰符 Async 或 Iterator 开头，后跟关键字 Function 或 Sub 和一个形参列表。lambda 表达式中的形参不能声明为 Optional 或 ParamArray，并且不能具有特性。与常规方法不同，省略 lambda 方法的参数类型不会自动推断 Object。相反，在对 lambda 方法进行重新分类后，将从目标类型推断省略的参数类型和 ByRef 修饰符。在上一示例中，lambda 表达式已编写为Function(x) x \* 2，并且当 lambda 方法用于创建 IntFunc 委托类型的实例时，该表达式会将 x 的类型推断为 Integer。与局部变量推断不同，如果 lambda 方法参数省略类型但具有数组或可为 null 的名称修饰符，则将发生编译时错误。

**正则 lambda 表达式**是不带 Async 和 Iterator 修饰符的表达式。

**迭代器 lambda 表达式**是带有 Iterator 修饰符，但没有 Async 修饰符的表达式。它必须是一个函数。将其重新归类为值时，只能将其重新归类为委托类型的值，其返回类型为 IEnumerator、IEnumerable 或 IEnumerator(Of T)（或者，对于某些 T 为 IEnumerable(Of T)），并且没有 ByRef 形参。

**异步 lambda 表达式**是带有 Async 修饰符，但没有 Iterator 修饰符的表达式。异步 sub lambda 只能重新归类为 sub 委托类型的值，不带 ByRef 形参。异步函数 lambda 只能重新归类为函数委托类型的值，其返回类型为 Task（或者，对于某些 T 为 Task(Of T)），并且没有 ByRef 形参。

Lambda 表达式可以为单行或多行。单行 Function lambda 表达式包含一个表示从 lambda 方法返回的值的表达式。单行 Sub lambda 表达式包含单个不带结束 StatementTerminator 的语句。例如：

Module Test Sub Do(a() As Integer, action As Action(Of Integer))  
 For index As Integer = 0 To a.Length - 1  
 action(a(index))  
 Next index  
 End Sub  
  
 Sub Main()  
 Dim a() As Integer = { 1, 2, 3, 4 }  
  
 Do(a, Sub(x As Integer) Console.WriteLine(x))  
 End Sub  
End Module

单行 lambda 构造的绑定紧密程度要小于所有其他表达式和语句。例如，“Function() x + 5”等效于“Function() (x+5)””而非“(Function() x) + 5”。为避免多义性，一个单行 Sub lambda 表达式不能含有 Dim 语句或标签声明语句。此外，单行 Sub lambda 表达式若不放在括号中，则后面不能紧跟冒号“:”、成员访问运算符“.”、字典成员访问运算符“!”或左括号“(”。该表达式不能含有任何块语句（With、SyncLock, If…EndIf、While、For、Do、Using），也不能含有 OnError 和 Resume。

批注

在 lambda 表达式“Function(i) x=i”中，主体解释为一个*表达式*（用于测试 x 和 i 是否相等）。但在 lambda 表达式“Sub(i) x=i”中，主体被解释为一个语句（它将 i 赋给 x）。

多行 lambda 表达式包含一个语句块，并且必须以适当的 End 语句（即 End Function 或 End Sub）结束。与常规方法一样，多行 lambda 方法的 Function 或 Sub 语句和 End 语句必须在其自己的行上。例如：

' Error: Function statement must be on its own line!  
Dim x = Sub(x As Integer) : Console.WriteLine(x) : End Sub  
  
' OK  
Dim y = Sub(x As Integer)  
 Console.WriteLine(x)  
 End Sub

多行 Function lambda 表达式可以声明一个返回类型，但不能为该类型设置特性。如果多行 Function lambda 表达式不声明返回类型，但返回类型可从使用该 lambda 表达式的上下文推断出来，则将使用该返回类型。否则，该函数的返回类型会计算如下。

1. 在常规 lambda 表达式中，返回类型是该语句块所有 Return 语句中的表达式的主导类型。
2. 在异步 lambda 表达式中，返回类型是 Task(Of T)，其中，T 是该语句块所有 Return 语句中的表达式的主导类型。
3. 在迭代器 lambda 表达式中，返回类型是 IEnumerable(Of T)，其中，T 是该语句块所有 Yield 语句中的表达式的主导类型。

例如：

Function f(min As Integer, max As Integer) As IEnumerable(Of Integer)  
 If min > max Then Throw New ArgumentException()  
 Dim x = Iterator Function()  
 For i = min To max  
 Yield i  
 Next  
 End Function

' infers x to be a delegate with return type IEnumerable(Of Integer)  
 Return x()  
End Function

在任何情况下，如果没有 Return（个别为 Yield）语句或这些语句之间没有主导类型，并且使用了严格语义，则将发生编译时错误；否则，主导类型隐式为 Object。

请注意，返回类型是从所有 Return 语句计算的，即使这些语句是不可访问的。例如：

' Return type is Double  
Dim x = Function()  
 Return 10  
 Return 10.50  
 End Function

不存在隐式返回变量，因为不存在该变量的名称。

多行 lambda 表达式中的语句块具有以下限制：

不允许有 On Error 和 Resume 语句，但允许有 Try 语句。

不能在多行 lambda 表达式中声明静态局部变量。

虽然多行 lambda 表达式的语句块中应用了一般分支规则，但不能分支到该语句块的内部或外部。例如：

Label1:  
Dim x = Sub()  
 ' Error: Cannot branch out  
 GoTo Label1  
  
 ' OK: Wholly within the lamba.  
 GoTo Label2:  
 Label2:  
 End Sub  
  
' Error: Cannot branch in  
GoTo Label2

lambda 表达式大致等效于包含类型上声明的匿名方法。初始示例大致等效于：

Module Test  
 Delegate Function IntFunc(x As Integer) As Integer  
  
 Sub Apply(a() As Integer, func As IntFunc)  
 For index As Integer = 0 To a.Length - 1  
 a(index) = func(a(index))  
 Next index  
 End Sub  
  
 Function $Lambda1(x As Integer) As Integer  
 Return x \* 2  
 End Function  
  
 Sub Main()  
 Dim a() As Integer = { 1, 2, 3, 4 }  
  
 Apply(a, AddressOf $Lambda1)  
  
 For Each value In a  
 Console.Write(value & " ")  
 Next value  
 End Sub  
End Module

LambdaExpression ::=  
 SingleLineLambda |  
 MultiLineLambda

SingleLineLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Expression |  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Statement

MultiLineLambda ::=  
 MultiLineFunctionLambda |  
 MultiLineSubLambda

MultiLineFunctionLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] [ As TypeName ] LineTerminator  
 Block  
 End Function

MultiLineSubLambda ::=  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] LineTerminator  
 Block  
 End Sub

LambdaModifier ::=  
 Async |  
 Iterator

### 闭包

Lambda 表达式可以访问范围内的所有变量，包括包含方法和 lambda 表达式中定义的局部变量或参数。当一个 lambda 表达式引用局部变量或参数时，该 lambda 表达式会捕获引用到闭包中的变量。闭包是一个存在于堆（而非堆栈）上的对象，在捕获一个变量时，对该变量的所有引用都将重定向到闭包。这使 lambda 表达式能够继续引用局部变量和参数，即使在完成包含方法之后也是如此。例如：

Module Test  
 Delegate Function D() As Integer  
  
 Function M() As D  
 Dim x As Integer = 10  
 Return Function() x  
 End Function  
  
 Sub Main()  
 Dim y As D = M()  
  
 ' Prints 10  
 Console.WriteLine(y())  
 End Sub  
End Module

大致等效于：

Module Test  
 Delegate Function D() As Integer  
  
 Class $Closure1  
 Public x As Integer  
  
 Function $Lambda1() As Integer  
 Return x  
 End Function  
 End Class  
  
 Function M() As D  
 Dim c As New $Closure1()  
 c.x = 10  
 Return AddressOf c.$Lambda1  
 End Function  
  
 Sub Main()  
 Dim y As D = M()  
  
 ' Prints 10  
 Console.WriteLine(y())  
 End Sub  
End Module

每当闭包进入声明一个局部变量的块中时，闭包都会捕获该局部变量的新副本，不过此新副本是用先前副本的值（如果有）初始化的。例如：

Module Test  
 Delegate Function D() As Integer  
  
 Function M() As D()  
 Dim a(9) As D  
  
 For i As Integer = 0 To 9  
 Dim x  
 a(i) = Function() x  
 x += 1  
 Next i  
  
 Return a  
 End Function  
  
 Sub Main()  
 Dim y() As D = M()  
  
 For i As Integer = 0 To 9  
 Console.Write(y(i)() & " ")  
 Next i  
 End Sub  
End Module

输出

1 2 3 4 5 6 7 8 9 10

而不是输出

9 9 9 9 9 9 9 9 9 9

由于在进入块时必须初始化闭包，因此不允许从一个具有闭包的块的外部通过 GoTo 跳转到该块中，但允许通过 Resume 跳转到具有闭包的块中。例如：

Module Test  
 Sub Main()  
 Dim a = 10  
  
 If a = 10 Then  
L1:  
 Dim x = Function() a  
  
 ' Valid, source is within block  
 GoTo L2  
L2:  
 End If  
  
 ' ERROR: target is inside block with closure  
 GoTo L1  
 End Sub  
End Module

由于以下内容无法捕获到闭包中，因此它们无法在 lambda 表达式中出现：

引用参数。

实例表达式（Me、MyClass、MyBase）（如果 Me 的类型不是类）。

匿名类型创建表达式的成员（如果 lambda 表达式是该表达式的一部分）。例如：

' Error: Lambda cannot refer to anonymous type field  
Dim x = New With { .a = 12, .b = Function() .a }

实例构造函数中的 ReadOnly 实例变量或共享构造函数中的 ReadOnly 共享变量，其中的变量在非值上下文中使用。例如：

Class C1  
 ReadOnly F1 As Integer  
  
 Sub New()  
 ' Valid, doesn’t modify F1  
 Dim x = Function() F1  
  
 ' Error, tries to modify F1  
 Dim f = Function() ModifyValue(F1)  
 End Sub  
  
 Sub ModifyValue(ByRef x As Integer)  
 End Sub  
End Class

## 查询表达式

查询表达式是这样一类表达式，它会对可查询集合的元素应用一系列查询运算符。 例如，以下表达式采用一个 Customer 对象集合，并返回华盛顿州 (WA) 的所有客户的名称：

Dim names = \_  
 From cust In Customers \_  
 Where cust.State = "WA" \_  
 Select cust.Name

查询表达式必须以 From 或 Aggregate 运算符开头，并以任何查询运算符结束。查询表达式的结果归类为一个值；该表达式的结果类型取决于表达式中最后一个查询运算符的结果类型。

QueryExpression ::=   
 FromOrAggregateQueryOperator |  
 QueryExpression QueryOperator

FromOrAggregateQueryOperator ::= FromQueryOperator | AggregateQueryOperator

*JoinOrGroupJoinQueryOperator ::= JoinQueryOperator* | GroupJoinQueryOperator

QueryOperator ::=  
 FromQueryOperator |  
 AggregateQueryOperator |  
 SelectQueryOperator |  
 DistinctQueryOperator |  
 WhereQueryOperator |  
 OrderByQueryOperator |  
 PartitionQueryOperator |  
 LetQueryOperator |  
 GroupByQueryOperator |  
 JoinOrGroupJoinQueryOperator

### 范围变量

某些查询运算符会引入一类称作范围变量的特殊变量。范围变量不是真正的变量；相反，它们表示对输入集合的查询计算过程中的各个值。范围变量的范围是从引入查询运算符到查询表达式的结尾或到隐藏它们的查询运算符（如 Select）。例如，在以下查询中

Dim waCusts = \_  
 From cust As Customer In Customers \_  
 Where cust.State = "WA"

From 查询运算符引入一个类型化为 Customer 的范围变量 cust，此变量表示 Customers 集合中的每个客户。后面的 Where 查询运算符接下来引用筛选器表达式中的范围变量 cust，从而确定是否从结果集合中筛选出单个客户。

有两类范围变量：集合范围变量和表达式范围变量。集合范围变量从正在查询的集合的元素获取其值。集合范围变量声明中的集合表达式必须归类为一个其类型可查询的值。如果省略集合范围变量的类型，则会将它推断为集合表达式的元素类型或 Object（如果集合表达式不具有元素类型，即只定义 Cast 方法）。如果集合表达式是不可查询的（即，无法推断集合的元素类型），则将发生编译时错误。

表达式范围变量是这样一个范围变量，其值是通过表达式而非集合来计算的。在下面的示例中，Select查询运算符引入一个名为 cityState 的表达式范围变量，该变量是从两个字段计算的：

Dim cityStates = \_  
 From cust As Customer In Customers \_  
 Select cityState = cust.City & "," & cust.State \_  
 Where cityState.Length() < 10

虽然表达式范围变量可能是一个不确定的值，但不要求此变量引用另一个范围变量。分配给一个表达式范围变量的表达式必须归类为一个值，并且必须能够隐式转换为该范围变量的类型（如果已给定）。

只有在 Let 运算符中，才可以指定表达式范围变量的类型。如果是其他运算符，或者如果未指定类型，则将使用局部变量类型推理来确定范围变量的类型。

范围变量必须遵循用于声明有关隐藏的局部变量的规则。因此，范围变量无法隐藏封闭方法或另一个范围变量中的局部变量或参数的名称（除非查询运算符专门隐藏范围内的所有当前范围变量）。

CollectionRangeVariableDeclarationList ::=  
 CollectionRangeVariableDeclaration |  
 CollectionRangeVariableDeclarationList Comma CollectionRangeVariableDeclaration

CollectionRangeVariableDeclaration ::=   
 Identifier [ As TypeName ] In [ LineTerminator ] Expression

ExpressionRangeVariableDeclarationList ::=  
 ExpressionRangeVariableDeclaration |  
 ExpressionRangeVariableDeclarationList Comma ExpressionRangeVariableDeclaration

ExpressionRangeVariableDeclaration ::=   
 [ Identifier [ As TypeName ] Equals ] Expression

### 可查询类型

可通过将查询表达式转换为对集合类型的已知方法的调用来实现查询表达式。这些定义完善的方法定义可查询集合的元素类型以及对集合执行的查询运算符的结果类型。每个查询运算符都会指定其通常转换为的方法，尽管特定转换与实现相关。在规范中，按照与如下格式类似的一般格式提供方法：

Function Select(selector As Func(Of T, R)) As CR

以下规则将应用于方法：

方法必须是集合类型的实例或扩展成员，并且必须是可访问的。

方法可以是泛型的，前提是可以推断所有类型参数。

方法可以进行重载，在此情况下，将使用重载解析来确定要使用的准确方法。

可以使用另一个委托类型来替代委托 Func 类型，前提是前者具有与匹配的 Func 类型相同的签名（包括返回类型）。

可以使用类型 System.Linq.Expressions.Expression(Of D) 来替代委托 Func 类型，前提是 D 是具有与匹配的 Func 类型相同的签名（包括返回类型）的委托类型。

类型 T 表示输入集合的元素类型。一个集合类型定义的所有方法都必须具有相同的输入元素类型，才能使该集合类型成为可查询的集合类型。

对于执行联接的查询运算符，类型 S 表示第二个输入集合的元素类型。

对于具有一组用作键的范围变量的查询运算符，类型 K 表示键类型。

类型 N 表示用作数值类型的类型（虽然此类型可能仍为用户定义的类型且不是内部数值类型）。

类型 B 表示一个可在布尔表达式中使用的类型。

类型 R 表示结果集合的元素类型（如果查询运算符产生结果集合）。R 依赖于查询运算符结束时范围内的范围变量数。如果有一个范围变量位于范围内，则 R 为该范围变量的类型。在下面的示例中

Dim custNames = \_  
 From c In Customers \_  
 Select c.Name

查询的结果将是一个具有 String 元素类型的集合类型。如果有多个范围变量位于范围内，则 R 为一个将范围内的所有范围变量作为 Key 字段包含的匿名类型。在下面的示例中：

Dim custAndOrderNames = \_  
 From c In Customers, o In c.Orders \_  
 Select Name = c.Name, ProductName = o.ProductName

查询的结果将为具有一个匿名类型的元素类型的集合类型，该匿名类型具有一个 String 类型的名为 Name 的只读属性和一个 String 类型的名为 ProductName 的只读属性。

在查询表达式内，产生的用于包含范围变量的匿名类型都是透明的，这意味着始终能够以非限定方式使用范围变量。例如，在上一示例中，可以按照非限定方式在 Select 查询运算符中访问范围变量 c 和 o，即使输入集合的元素类型为匿名类型也是如此。

类型 CX 表示一个集合类型（不一定是输入集合类型），其元素类型为某个类型 X。

可查询集合类型必须满足下列条件之一（按照优先选择顺序）：

它必须定义一致的 Select 方法。

它必须具有下列方法之一

Function AsEnumerable() As CT  
Function AsQueryable() As CT

可以调用这些方法来获取可查询集合。如果同时提供这两个方法，则 AsQueryable 优先于 AsEnumerable。

它必须具有一个方法

Function Cast(Of T)() As CT

通过将此方法与范围变量的类型一起调用可产生可查询集合。

由于将独立于实际方法调用来确定集合的元素类型，因此无法确定特定方法的适用性。因此，在确定集合的元素类型时，如果存在与已知方法相匹配的实例方法，则将忽略与已知方法相匹配的任何扩展方法。

查询运算符转换将按照查询运算符在表达式中的出现顺序进行。集合对象不必实现所有查询运算符所需的所有方法，但每个集合对象必须至少支持 Select 查询运算符。如果所需的方法不存在，则将发生编译时错误。在绑定已知方法名称时，为了在接口和扩展方法绑定中实现多重继承，将忽略非方法名称，但隐藏语义仍将应用。例如：

Class Q1  
 Public Function [Select](selector As Func(Of Integer, Integer)) As Q1  
 End Function  
End Class  
  
Class Q2  
 Inherits Q1  
  
 Public [Select] As Integer  
End Class  
  
Module Test  
 Sub Main()  
 Dim qs As New Q2()  
  
 ' Error: Q2.Select still hides Q1.Select  
 Dim zs = From q In qs Select q  
 End Sub  
End Module

### 默认查询索引器

每个可查询集合类型（其元素类型为 T 且尚不具有默认属性）将视为具有以下一般形式的默认属性：

Public ReadOnly Default Property Item(index As Integer) As T  
 Get  
 Return Me.ElementAtOrDefault(index)  
 End Get  
End Property

只能使用默认属性访问语法来引用默认属性，而不能按照名称来引用默认属性。例如：

Dim customers As IEnumerable(Of Customer) = ...  
Dim customerThree = customers(2)  
  
' Error, no such property  
Dim customerFour = customers.Item(4)

如果集合类型不具有 ElementAtOrDefault 成员，则将发生编译时错误。

### From 查询运算符

From 查询运算符引入一个集合范围变量，此变量表示要查询的集合的单个成员。例如，以下查询表达式：

From c As Customer In Customers ...

可视为等效于

For Each c As Customer In Customers  
 ...  
Next c

若一个 From 查询运算符声明多个集合范围变量或不是查询表达式中的第一个 From 查询运算符，则每个新的集合范围变量将交叉联接到现有的范围变量集。结果是根据联接的集合中所有元素的叉积来评估查询。例如，以下表达式：

From c In Customers \_  
From e In Employees \_  
...

可视为等效于：

For Each c In Customers  
 For Each e In Employees  
 ...  
 Next e  
Next c

并且完全等效于：

From c In Customers, e In Employees ...

可以在后一个 From 查询运算符中使用上一个查询运算符中引入的范围变量。例如，在以下查询表达式中，第二个 From 查询运算符引用第一个范围变量的值：

From c As Customer In Customers \_  
From o As Order In c.Orders \_  
Select c.Name, o

如果集合类型包含下面的一个或两个方法，则仅支持一个 From 查询运算符或多个 From 查询运算符中的多个范围变量：

Function SelectMany(selector As Func(Of T, CR)) As CR  
Function SelectMany(selector As Func(Of T, CS), \_  
 resultsSelector As Func(Of T, S, R)) As CR

代码

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = From x In xs, y In ys ...

通常会转换为

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = \_  
 xs.SelectMany( \_  
 Function(x As Integer) ys, \_  
 Function(x As Integer, y As Integer) New With {x, y})...

注意   From 不是保留字。

FromQueryOperator ::=  
 [ LineTerminator ] From [ LineTerminator ] CollectionRangeVariableDeclarationList

### Join 查询运算符

Join 查询运算符用于将现有范围变量与新的集合范围变量联接在一起，从而产生一个集合，该集合的元素已基于一个相等表达式联接在一起。例如：

Dim customersAndOrders = \_  
 From cust In Customers \_  
 Join ord In Orders On cust.ID Equals ord.CustomerID

此相等表达式受到比常规相等表达式更多的限制：

这两个表达式都必须归类为一个值。

这两个表达式都必须至少引用一个范围变量。

Join 查询运算符中声明的范围变量必须由某一个表达式引用，并且该表达式不得引用任何其他范围变量。

在这两个表达式的类型不是同一类型的情况下：

如果为两个类型定义了相等运算符，而且两个表达式都能够隐式转换为该类型且该类型不是 Object，则将两个表达式都转换为该类型。

否则，如果存在一个这两个表达式都能够隐式转换为的主导类型，则将两个表达式都转换为该类型。

其他情况下，将发生编译时错误。

使用哈希值（即，通过调用 GetHashCode()）而非相等运算符来比较表达式以提高效率。Join 查询运算符可以在同一运算符中执行多重联接或相等条件。Join 查询运算符仅在集合类型包含方法时受支持：

Function Join(inner As CS, \_  
 outerSelector As Func(Of T, K), \_  
 innerSelector As Func(Of S, K), \_  
 resultSelector As Func(Of T, S, R)) As CR

代码

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = From x In xs \_  
 Join y In ys On x Equals y \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = \_  
 xs.Join( \_  
 ys, \_  
 Function(x As Integer) x, \_  
 Function(y As Integer) y, \_  
 Function(x As Integer, y As Integer) New With {x, y})...

注意   Join、On 和 Equals 不是保留字。

JoinQueryOperator ::=  
 [ LineTerminator ] Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList

JoinConditionList ::=  
 JoinCondition |  
 JoinConditionList And [ LineTerminator ] JoinCondition

JoinCondition ::= Expression Equals [ LineTerminator ] Expression

### Let 查询运算符

Let 查询运算符引入一个表达式范围变量。这将允许计算一次将在后面的查询运算符中多次使用的中间值。例如：

Dim taxedPrices = \_  
 From o In Orders \_  
 Let tax = o.Price \* 0.088 \_  
 Where tax > 3.50 \_  
 Select o.Price, tax, total = o.Price + tax

可视为等效于：

For Each o In Orders  
 Dim tax = o.Price \* 0.088  
 ...  
Next o

Let 查询运算符仅在集合类型包含方法时受支持：

Function Select(selector As Func(Of T, R)) As CR

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Let y = x \* 10 \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Select(Function(x As Integer) New With {x, .y = x \* 10})...

LetQueryOperator ::=  
 [ LineTerminator ] Let [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Select 查询运算符

Select 查询运算符与 Let 查询运算符的相同之处是，二者都引入表达式范围变量；但 Select 查询运算符将隐藏当前可用的范围变量，而不是添加到这些变量中。此外，始终使用局部变量类型推理规则来推断由 Select 查询运算符引入的表达式范围变量的类型；不能指定显式类型，如果无法推断类型，则将发生编译时错误。例如，在以下查询中：

Dim smiths = \_  
 From cust In Customers \_  
 Select name = cust.name \_  
 Where name.EndsWith("Smith")

Where 查询运算符只能访问由 Select 运算符引入的 name 范围变量；如果 Where 运算符已尝试引用 cust，则可能已发生编译时错误。

Select 查询运算符可以使用与匿名类型对象创建表达式相同的规则推断范围变量的名称，而不是显式指定范围变量的名称。例如：

Dim custAndOrderNames = \_  
 From cust In Customers, ord In cust.Orders \_  
 Select cust.name, ord.ProductName \_  
 Where name.EndsWith("Smith")

如果未提供范围变量的名称并且无法推断名称，则将发生编译时错误。如果 Select 查询运算符仅包含一个表达式，则当无法推断范围变量的名称时不会发生任何错误，但范围变量是无名称的。例如：

Dim custAndOrderNames = \_  
 From cust In Customers, ord In cust.Orders \_  
 Select cust.Name & " bought " & ord.ProductName \_  
 Take 10

如果 Select 查询运算符无法确定是为范围变量分配名称还是选择相等表达式，则优先选择名称分配。例如：

Dim badCustNames = \_  
 From c In Customers \_  
 Let name = "John Smith" \_  
 Select name = c.Name ' Creates a range variable named "name"

Dim goodCustNames = \_  
 From c In Customers \_  
 Let name = "John Smith" \_  
 Select match = (name = c.Name)

Select 查询运算符中的每个表达式都必须归类为一个值。Select 查询运算符仅在集合类型包含方法时受支持：

Function Select(selector As Func(Of T, R)) As CR

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Select x, y = x \* 10 \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Select(Function(x As Integer) New With {x, .y = x \* 10})...

SelectQueryOperator ::=  
 [ LineTerminator ] Select [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Distinct 查询运算符

Distinct 查询运算符用于将集合中的值限制为带不同值的值，这可以通过比较元素类型是否相等来确定。例如，以下查询：

Dim distinctCustomerPrice = \_  
 From cust In Customers, ord In cust.Orders \_  
 Select cust.Name, ord.Price \_  
 Distinct

只会为每个不同的客户名称与订单价格对返回一个行，即使客户具有多个同一价格的订单也是如此。Distinct 查询运算符仅在集合类型包含方法时受支持：

Function Distinct() As CT

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Distinct \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = xs.Distinct()...

注意   Distinct 不是保留字。

DistinctQueryOperator ::= [ LineTerminator ] Distinct [ LineTerminator ]

### Where 查询运算符

Where 查询运算符用于将集合中的值限制为满足某个给定条件的值。Where 查询运算符使用一个将为每组范围变量值进行计算的布尔表达式；如果此布尔表达式的值为 true，则值将出现在输出集合中；否则将跳过值。例如，以下查询表达式：

From cust In Customers, ord In Orders \_  
Where cust.ID = ord.CustomerID \_  
...

可视为等效于嵌套的循环

For Each cust In Customers  
 For Each ord In Orders  
 If cust.ID = ord.CustomerID Then  
 ...  
 End If  
 Next ord  
Next cust

Where 查询运算符仅在集合类型包含方法时受支持：

Function Where(predicate As Func(Of T, B)) As CT

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Where x < 10 \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Where(Function(x As Integer) x < 10)...

注意   Where 不是保留字。

WhereQueryOperator ::= [ LineTerminator ] Where [ LineTerminator ] BooleanExpression

### 分区查询运算符

Take 查询运算符会产生集合中的前 n 个元素。在与 While 修饰符一起使用时，Take 运算符会产生集合的前 n 个满足布尔表达式的元素。Skip 运算符用于跳过集合的前 n 个元素并返回集合的其余元素。在与 While 修饰符一起使用时，Skip 运算符将跳过集合的前 n 个满足布尔表达式的元素，然后返回集合的其余元素。Take 或 Skip 查询运算符中的表达式必须归类为一个值。

Take 查询运算符仅在集合类型包含方法时受支持：

Function Take(count As N) As CT

Skip 查询运算符仅在集合类型包含方法时受支持：

Function Skip(count As N) As CT

Take While 查询运算符仅在集合类型包含方法时受支持：

Function TakeWhile(predicate As Func(Of T, B)) As CT

Skip While 查询运算符仅在集合类型包含方法时受支持：

Function SkipWhile(predicate As Func(Of T, B)) As CT

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Skip 10 \_  
 Take 5 \_  
 Skip While x < 10 \_  
 Take While x > 5 \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Skip(10). \_  
 Take(5). \_  
 SkipWhile(Function(x) x < 10). \_  
 TakeWhile(Function(x) x > 5)...

注意   Take 和 Skip 不是保留字。

PartitionQueryOperator ::=   
 [ LineTerminator ] Take [ LineTerminator ] Expression |  
 [ LineTerminator ] Take While [ LineTerminator ] BooleanExpression |  
 [ LineTerminator ] Skip [ LineTerminator ] Expression |  
 [ LineTerminator ] Skip While [ LineTerminator ] BooleanExpression

### Order By 查询运算符

Order By 查询运算符用于对范围变量中出现的值进行排序。Order By 查询运算符使用的表达式将指定应用于对迭代变量进行排序的键值。里如，以下查询将返回按价格排序的产品：

Dim productsByPrice = \_  
 From p In Products \_  
 Order By p.Price \_  
 Select p.Name

可以将排序标记为 Ascending，在此情况下，较小的值将排在较大的值之前；也可以将排序标记为 Descending，在此情况下，较小的值将排在较大的值之后。如果未指定任何排序顺序，则默认排序顺序为 Ascending。例如，以下查询将返回按价格排序的产品（价格最高的产品排在第一位）：

Dim productsByPriceDesc = \_  
 From p In Products \_  
 Order By p.Price Descending \_  
 Select p.Name

Order By 查询运算符可以指定多个表达式进行排序，在此情况下，将按照某种嵌套方式对集合进行排序。例如，以下查询将先后按照省/市/自治区、每个省/市/自治区内的城市和每个城市内的邮政编码对客户进行排序：

Dim customersByLocation = \_  
 From c In Customers \_  
 Order By c.State, c.City, c.ZIP \_  
 Select c.Name, c.State, c.City, c.ZIP

Order By 查询运算符中的表达式必须归类为一个值。Order By 查询运算符仅在集合类型包含以下一个或两个方法时受支持：

Function OrderBy(keySelector As Func(Of T, K)) As CT  
Function OrderByDescending(keySelector As Func(Of T, K)) As CT

返回类型 CT 必须为一个已排序的集合。已排序的集合是一个包含以下一个或两个方法的集合类型：

Function ThenBy(keySelector As Func(Of T, K)) As CT  
Function ThenByDescending(keySelector As Func(Of T, K)) As CT

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Order By x Ascending, x Mod 2 Descending \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.OrderBy(Function(x) x).ThenByDescending(Function(x) x Mod 2)...

批注

由于查询运算符只是将语法映射到实现特定查询运算的方法，因此顺序保留不是由语言来指示，而是通过运算符的自身实现来确定。这与用户定义的运算符非常类似，即在为用户定义的数值类型重载加法运算符时无法执行与加法类似的任何运算。当然，为了保留可预见性，建议不要实现与用户预期不符的某些运算。

注意   Order 和 By 不是保留字。

OrderByQueryOperator ::= [ LineTerminator ] Order By [ LineTerminator ] OrderExpressionList

OrderExpressionList ::=  
 OrderExpression |  
 OrderExpressionList Comma OrderExpression

OrderExpression ::=  
 Expression [ Ordering ]

Ordering ::= Ascending | Descending

### Group By 查询运算符

Group By 查询运算符用于根据一个或多个表达式对范围内的范围变量进行分组，然后根据这些分组产生新的范围变量。例如，以下查询将按照 State 对所有客户进行分组，然后计算每个组所含客户的数目和平均年龄：

Dim averageAges = \_  
 From cust In Customers \_  
 Group By cust.State \_  
 Into Count(), Average(cust.Age)

Group By 查询运算符有三个子句：可选的 Group 子句、By 子句和 Into 子句。Group 子句具有与 Select 查询运算符相同的语法和作用，只是该子句仅影响 Into 子句（而非 By 子句）中的范围变量。例如：

Dim averageAges = \_  
 From cust In Customers \_  
 Group cust.Age By cust.State \_  
 Into Count(), Average(Age)

By 子句将声明在分组运算中用作键值的表达式范围变量。利用 Into 子句，可以声明用于对 By 子句形成的每个组进行聚合计算的表达式范围变量。在 Into 子句中，只能为表达式范围变量分配一个作为聚合函数的方法调用的表达式。聚合函数是组的集合类型（它可能不一定是原始集合的集合类型）的函数，它与以下任一方法类似：

Function <name>() As <type>  
Function <name>(selector As Func(Of T, R)) As R

如果聚合函数使用了委托参数，则调用表达式可以具有一个必须归类为值的参数表达式。此参数表达式可以使用范围内的范围变量；在对聚合函数的调用中，这些范围变量表示正在形成的组中的值，而不是集合中的所有值。例如，在本节的最初的示例中，Average 函数将计算每个州内的客户的平均年龄，而不是计算所有客户的平均年龄。

所有集合类型可视为自身已定义聚合函数 Group，此聚合函数不使用任何参数并且只返回组。集合类型可以提供的其他标准聚合函数包括：

Count 和 LongCount，这两个聚合函数将返回组中的元素计数或组中满足布尔表达式的元素计数。Count 和 LongCount 仅在集合类型包含下列方法之一时受支持：

Function Count() As N  
Function Count(selector As Func(Of T, B)) As N  
Function LongCount() As N  
Function LongCount(selector As Func(Of T, B)) As N

Sum，此聚合函数将返回跨组中所有元素的表达式的和。Sum 仅在集合类型包含下列方法之一时受支持：

Function Sum() As N  
Function Sum(selector As Func(Of T, N)) As N

Min，此聚合函数将返回跨组中所有元素的表达式的最小值。Min 仅在集合类型包含下列方法之一时受支持：

Function Min() As N  
Function Min(selector As Func(Of T, N)) As N

Max，此聚合函数将返回跨组中所有元素的表达式的最大值。Max 仅在集合类型包含下列方法之一时受支持：

Function Max() As N  
Function Max(selector As Func(Of T, N)) As N

Average，此聚合函数将返回跨组中所有元素的表达式的平均值。Average 仅在集合类型包含下列方法之一时受支持：

Function Average() As N  
Function Average(selector As Func(Of T, N)) As N

Any，此聚合函数将确定组是否包含成员或布尔表达式对于组中的任何元素是否都为 true。Any 将返回一个可在布尔表达式中使用的值，并且仅在集合类型包含下列方法之一时受支持：

Function Any() As B  
Function Any(predicate As Func(Of T, B)) As B

All，此聚合函数将确定布尔表达式对于组中的所有元素是否都为 true。All 将返回一个可在布尔表达式中使用的值，并且仅在集合类型包含方法时受支持：

Function All(predicate As Func(Of T, B)) As B

在 Group By 查询运算符之后，先前在范围内的范围变量将会隐藏，而由 By 和 Into 子句引入的范围变量将变为可用。Group By 查询运算符仅在集合类型包含方法时受支持：

Function GroupBy(keySelector As Func(Of T, K), \_  
 resultSelector As Func(Of K, CT, R)) As CR

Group 子句中的范围变量声明仅在集合类型包含以下方法时受支持：

Function GroupBy(keySelector As Func(Of T, K), \_  
 elementSelector As Func(Of T, S), \_  
 resultSelector As Func(Of K, CS, R)) As CR

代码

Dim xs() As Integer = ...  
Dim zs = From x In xs \_  
 Group y = x \* 10, z = x / 10 By evenOdd = x Mod 2 \_  
 Into Sum(y), Average(z) \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.GroupBy( \_  
 Function(x As Integer) x Mod 2, \_  
 Function(x As Integer) New With {.y = x \* 10, .z = x / 10}, \_  
 Function(evenOdd, group) New With { \_  
 evenOdd, \_  
 .Sum = group.Sum(Function(e) e.y), \_  
 .Average = group.Average(Function(e) e.z)})...

注意   Group、By 和 Into 不是保留字。

GroupByQueryOperator ::=  
 [ LineTerminator ] Group [ [ LineTerminator ] ExpressionRangeVariableDeclarationList ]  
 [ LineTerminator ] By [ LineTerminator ] ExpressionRangeVariableDeclarationList   
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Aggregate 查询运算符

Aggregate 查询运算符将执行与 Group By 运算符相同的功能，只是此查询运算符允许对已形成的组进行聚合。由于组已形成，因此 Aggregate 查询运算符的 Into 子句不会隐藏范围内的范围变量（通过此方式，Aggregate 与 Let 更加类似，而 Group By 与 Select 更加类似）。例如，以下查询将计算由华盛顿州的客户下达的所有订单的总数：

Dim orderTotals = \_  
 From cust In Customers \_  
 Where cust.State = "WA" \_  
 Aggregate order In cust.Orders \_  
 Into Sum(order.Total)

此查询的结果是元素类型为一个匿名类型的集合，该匿名类型具有一个类型化为 Customer 的名为 cust 的属性，以及一个类型化为 Integer 的名为 Sum 的属性。

与 Group By 不同，Aggregate 子句和 Into 子句之间可以放入其他查询运算符。在 Aggregate 子句和 Into 子句的结尾之间，可以使用范围内的所有范围变量（包括 Aggregate 子句声明的范围变量）。例如，以下查询将计算华盛顿州的客户在 2006 年之前下达的所有订单的总数：

Dim orderTotals = \_  
 From cust In Customers \_  
 Where cust.State = "WA" \_  
 Aggregate order In cust.Orders \_  
 Where order.Date <= #01/01/2006# \_  
 Into Sum = Sum(order.Total)

Aggregate 运算符还可用于开始查询表达式。在此情况下，查询表达式的结果将为一个由 Into 子句计算的值。例如，以下查询将计算 2006 年 1 月 1 日之前下达的所有订单的总数：

Dim ordersTotal = \_  
 Aggregate order In Orders \_  
 Where order.Date <= #01/01/2006# \_  
 Into Sum(order.Total)

此查询的结果是一个 Integer 值。Aggregate 查询运算符总是可用（但必须同时提供聚合函数才能使表达式有效）。代码

Dim xs() As Integer = ...  
Dim zs = \_  
 Aggregate x In xs \_  
 Where x < 5 \_  
 Into Sum()

通常会转换为

Dim xs() As Integer = ...  
Dim zs = \_  
 xs.Where(Function(x) x < 5).Sum()

注意   Aggregate 和 Into 不是保留字。

AggregateQueryOperator ::=  
 [ LineTerminator ] Aggregate [ LineTerminator ] CollectionRangeVariableDeclaration   
 [ QueryOperator+ ]   
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

### Group Join 查询运算符

Group Join 查询运算符用于将 Join 和Group By 查询运算符的功能合并到一个运算符中。Group Join 将根据从元素中提取的匹配键来联接两个集合，从而将联接右侧的与联接左侧的某个特定元素匹配的所有元素组合在一起。因此，该运算符将产生一组分层结果。例如，以下查询会产生包含单个客户名称、一个由所有客户订单构成的组和所有这些订单的总数的元素：

Dim custsWithOrders = \_  
 From cust In Customers \_  
 Group Join order In Orders On cust.ID Equals order.CustomerID \_   
 Into Orders = Group, OrdersTotal = Sum(order.Total) \_  
 Select cust.Name, Orders, OrdersTotal

此查询的结果是元素类型为一个匿名类型的集合，该匿名类型具有三个属性：类型化为 String 的 Name、类型化为集合（其元素类型为 Order）的 Orders 以及类型化为 Integer 的 OrdersTotal。Group Join 查询运算符仅在集合类型包含方法时受支持：

Function GroupJoin(inner As CS, \_  
 outerSelector As Func(Of T, K), \_  
 innerSelector As Func(Of S, K), \_  
 resultSelector As Func(Of T, CS, R)) As CR

代码

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = From x In xs \_  
 Group Join y in ys On x Equals y \_  
 Into g = Group \_  
 ...

通常会转换为

Dim xs() As Integer = ...  
Dim ys() As Integer = ...  
Dim zs = \_  
 xs.GroupJoin( \_  
 ys, \_  
 Function(x As Integer) x, \_  
 Function(y As Integer) y, \_  
 Function(x, group) New With {x, .g = group})...

注意   Group、Join 和 Into 不是保留字。

GroupJoinQueryOperator ::=  
 [ LineTerminator ] Group Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

## 条件表达式

条件 If 表达式用于测试表达式并返回值。但与 IIF 运行时函数不同，条件表达式仅在必要时计算其操作数。例如，如果 c 的值为 Nothing，则表达式 If(c Is Nothing, c.Name, "Unknown") 将不会引发异常。条件表达式具有两种形式：一种形式使用两个操作数，另一种形式使用三个操作数。

如果提供了三个操作数，则所有三个表达式都必须归类为值，并且第一个操作数必须为布尔表达式。如果该表达式的结果为 true，则第二个表达式将为运算符的结果；否则第三个表达式将为运算符的结果。该表达式的结果类型为第二个表达式的类型和第三个表达式的类型之间的主导类型。如果没有主导类型，则将发生编译时错误。

如果提供了两个操作数，则这两个操作数都必须归类为值，并且第一个操作数必须为引用类型或可为 null 的值类型。然后，将按照计算表达式 If(x IsNot Nothing, x, y) 的方式来计算表达式 If(x, y)，但有以下两种例外情况。第一种例外情况是，仅计算第一个表达式一次；第二个例外情况是，如果第二个操作数的类型是不可为 null 的值类型，而第一个操作数的类型是可为 null 的值类型，则在确定针对表达式的结果类型的主导类型时，将从第一个操作数的类型中删除 ?。例如：

Module Test  
 Sub Main()  
 Dim x?, y As Integer  
 Dim a?, b As Long  
  
 a = If(x, a) ' Result type: Long?  
 y = If(x, 0) ' Result type: Integer  
 End Sub  
End Module

对于该表达式的两种形式，如果操作数为 Nothing，则其类型不用于确定主导类型。对于表达式 If(<expression>, Nothing, Nothing)，将 Object 视为主导类型。

ConditionalExpression ::=   
 If OpenParenthesis BooleanExpression Comma Expression Comma Expression CloseParenthesis |  
 If OpenParenthesis Expression Comma Expression CloseParenthesis

## XML 文本表达式

XML 文本表达式表示 XML（可扩展标记语言）1.0 值。XML 文本表达式的结果是一个类型化为 System.Xml.Linq 命名空间中的某个类型的值。如果该命名空间中的类型不可用，则 XML 文本表达式将导致发生编译时错误。这些值将通过从 XML 文本表达式转换的构造函数调用来生成。例如，以下代码：

Dim book As System.Xml.Linq.XElement = \_  
 <book title="My book"></book>

大致等效于代码：

Dim book As System.Xml.Linq.XElement = \_  
 New System.Xml.Linq.XElement( \_  
 "book", \_  
 New System.Xml.Linq.XAttribute("title", "My book"))

XML 文本表达式可以采用以下形式：XML 文档、XML 元素、XML 处理指令、XML 注释或 CDATA 节。

批注

此规范仅包含用于描述 Visual Basic 语言行为的足够的 XML 说明。有关 XML 的更多信息，请参见 [http://www.w3.org/TR/REC-xml/](http://vstfdevdiv:8080/web/wi.aspx)。

XMLLiteralExpression ::=  
 XMLDocument |  
 XMLElement | XMLProcessingInstruction |  
 XMLComment |  
 XMLCDATASection

### 词法规则

将使用 XML 的词法规则（而非常规 Visual Basic 代码的词法规则）来解释 XML 文本表达式。一般来说，这两组规则的不同之处包括：

空白在 XML 中是有效的。因此，XML 文本表达式的语法会显式指明允许空白的位置。除非空白出现在元素内的字符数据的上下文中，否则不保留空白。例如：

' The following element preserves no whitespace  
Dim e1 = \_  
 <customer>  
 <name>Bob</>  
 </>  
  
' The following element preserves all of the whitespace  
Dim e2 = \_  
 <customer>  
 Bob  
 </>

XML 行尾空白将按照 XML 规范正常化。

XML 区分大小写。关键字必须是大小写完全匹配的，否则将发生编译时错误。

在 XML 中，将行终止符视为空白。因此，XML 文本表达式中不需要行继续符。

XML 不接受全角字符。如果使用全角字符，则将发生编译时错误。

XMLCharacter ::=  
 < Unicode tab character (0x0009) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character (0x000D) > |  
 < Unicode characters 0x0020 – 0xD7FF > |  
 < Unicode characters 0xE000 – 0xFFFD > |  
 < Unicode characters 0x10000 – 0x10FFFF >

XMLString ::= XMLCharacter+

XMLWhitespace ::= XMLWhitespaceCharacter+

XMLWhitespaceCharacter ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode space character (0x0020) > |  
 < Unicode tab character (0x0009) >

XMLNameCharacter ::= XMLLetter | XMLDigit | . | - | \_ | : | XMLCombiningCharacter | XMLExtender

XMLNameStartCharacter ::= XMLLetter | \_ | :

XMLName ::= XMLNameStartCharacter [ XMLNameCharacter+ ]

XMLLetter ::=   
 < Unicode character as defined in the Letter production of the XML 1.0 specification >

XMLDigit ::=  
 < Unicode character as defined in the Digit production of the XML 1.0 specification >

XMLCombiningCharacter ::=  
 < Unicode character as defined in the CombiningChar production of the XML 1.0 specification >

XMLExtender ::=  
 < Unicode character as defined in the Extender production of the XML 1.0 specification >

### 嵌入式表达式

XML 文本表达式可以包含嵌入式表达式。嵌入式表达式是一个 Visual Basic 表达式，可以对该表达式进行计算并用来在嵌入式表达式所在位置填入一个或多个值。例如，以下代码将置入字符串 John Smith 作为 XML 元素的值：

Dim name as String = "John Smith"  
Dim element As System.Xml.Linq.XElement = <customer><%= name %></customer>

表达式可以嵌入到许多上下文中。例如，下面的代码示例产生一个名为 customer 的元素：

Dim name As String = "customer"  
Dim element As System.Xml.Linq.XElement = <<%= name %>>John Smith</>

可使用嵌入式表达式的每个上下文都会指定将要接受的类型。当处于嵌入式表达式的表达式部分的上下文中时，针对 Visual Basic 代码的一般词法规则仍将适用，因此必须使用行继续符：

' Visual Basic expression uses line continuation, XML does not  
Dim element As System.Xml.Linq.XElement = \_  
 <<%= name & \_  
 name %>>John   
 Smith</>

XMLEmbeddedExpression ::=  
 < % = [ LineTerminator ] Expression [ LineTerminator ] % >

### XML 文档

XML 文档会产生类型化为 System.Xml.Linq.XDocument 的值。与 XML 1.0 规范不同，XML 文本表达式中的 XML 文档需要指定 XML 文档序言；不带 XML 文档序言的 XML 文本表达式将被解释为其单个实体。例如：

Dim doc As System.Xml.Linq.XDocument = \_  
 <?xml version="1.0"?>  
 <?instruction?>  
 <customer>Bob</>  
  
Dim pi As System.Xml.Linq.XProcessingInstruction = \_  
 <?instruction?>

XML 文档可以包含一个其类型可为任何类型的嵌入式表达式；但在运行时，该对象必须满足 XDocument 构造函数的要求，否则将发生运行时错误。

与常规 XML 不同，XML 文档表达式不支持 DTD（文档类型声明）。此外，由于 XML 文档表达式的编码始终与源文件自身的编码相同，因此将忽略编码特性（如果已提供）。

批注

虽然编码特性将被忽略，但它仍是有效特性，以便能够继续在源代码中包含任何有效的 Xml 1.0 文档。

XMLDocument ::=  
 XMLDocumentPrologue [ XMLMisc+ ] XMLDocumentBody [ XMLMisc+ ]

XMLDocumentPrologue ::=  
 < ? xml XMLVersion [ XMLEncoding ] [ XMLStandalone ] [ XMLWhitespace ] ? >

XMLVersion ::=  
 XMLWhitespace version [ XMLWhitespace ] = [ XMLWhitespace ] XMLVersionNumberValue

XMLVersionNumberValue ::=   
 SingleQuoteCharacter 1 . 0 SingleQuoteCharacter |  
 DoubleQuoteCharacter 1 . 0 DoubleQuoteCharacter

XMLEncoding ::=  
 XMLWhitespace encoding [ XMLWhitespace ] = [ XMLWhitespace ] XMLEncodingNameValue

XMLEncodingNameValue ::=   
 SingleQuoteCharacter XMLEncodingName SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLEncodingName DoubleQuoteCharacter

XMLEncodingName ::= XMLLatinAlphaCharacter [ XMLEncodingNameCharacter+ ]

XMLEncodingNameCharacter ::=  
 XMLUnderscoreCharacter |  
 XMLLatinAlphaCharacter |  
 XMLNumericCharacter |  
 XMLPeriodCharacter |  
 XMLDashCharacter

XMLLatinAlphaCharacter ::=  
 < Unicode Latin alphabetic character (0x0041-0x005a, 0x0061-0x007a) >

XMLNumericCharacter ::= < Unicode digit character (0x0030-0x0039) >

XMLHexNumericCharacter ::=  
 XMLNumericCharacter |  
 < Unicode Latin hex alphabetic character (0x0041-0x0046, 0x0061-0x0066) >

XMLPeriodCharacter ::= < Unicode period character (0x002e) >

XMLUnderscoreCharacter ::= < Unicode underscore character (0x005f) >

XMLDashCharacter ::= < Unicode dash character (0x002d) >

XMLStandalone ::=  
 XMLWhitespace standalone [ XMLWhitespace ] = [ XMLWhitespace ] XMLYesNoValue

XMLYesNoValue ::=   
 SingleQuoteCharacter XMLYesNo SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLYesNo DoubleQuoteCharacter

XMLYesNo ::= yes | no

XMLMisc ::=  
 XMLComment |  
 XMLProcessingInstruction |  
 XMLWhitespace

XMLDocumentBody ::= XMLElement | XMLEmbeddedExpression

### XML 元素

XML 元素会产生类型化为 System.Xml.Linq.XElement 的值。与常规 XML 不同，XML 元素可以省略结束标记中的名称，并且当前嵌套最深的元素将被封闭。例如：

Dim name = <name>Bob</>

XML 元素中的特性声明会产生类型化为 System.Xml.Linq.XAttribute 的值。特性值将按照 XML 规范正常化。当某个特性的值为 Nothing 时，将不会创建此特性，因此不必检查特性值表达式是否有 Nothing。例如：

Dim expr = Nothing  
  
' Throws null argument exception  
Dim direct = New System.Xml.Linq.XElement( \_  
 "Name", \_  
 New System.Xml.Linq.XAttribute("Length", expr))  
  
' Doesn't throw exception, the result is <Name/>  
Dim literal = <Name Length=<%= expr %>/>

XML 元素和特性可以在以下位置包含嵌入式表达式：

元素的名称，在此情况下，嵌入式表达式必须为一个能够隐式转换为 System.Xml.Linq.XName 的类型的值。例如：

Dim name = <<%= "name" %>>Bob</>

特素的名称，在此情况下，嵌入式表达式必须为一个能够隐式转换为 System.Xml.Linq.XName 的类型的值。例如：

Dim name = <name <%= "length" %>="3">Bob</>

元素的特性的值，在此情况下，嵌入式表达式可以是任何类型的值。例如：

Dim name = <name length=<%= 3 %>>Bob</>

元素的特性，在此情况下，嵌入式表达式可以是任何类型的值。例如：

Dim name = <name <%= new XAttribute("length", 3) %>>Bob</>

元素的内容，在此情况下，嵌入式表达式可以是任何类型的值。例如：

Dim name = <name><%= "Bob" %></>

如果嵌入式表达式的类型为 Object()，则将数组作为一个参数数组传递给 XElement 构造函数。

XMLElement ::=  
 XMLEmptyElement |  
 XMLElementStart XMLContent XMLElementEnd

XMLEmptyElement ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitepace ] / >

XMLElementStart ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitespace ] >

XMLElementEnd ::=  
 < / > |  
 < / XMLQualifiedName [ XMLWhitespace ] >

XMLContent ::=  
 [ XMLCharacterData ] [ XMLNestedContent [ XMLCharacterData ] ]+

XMLCharacterData ::=  
 < Any XMLCharacterDataString that does not contain the string "]]>" >

XMLCharacterDataString ::=  
 < Any Unicode character except < or & >+

XMLNestedContent ::=  
 XMLElement |  
 XMLReference |  
 XMLCDATASection |  
 XMLProcessingInstruction |  
 XMLComment |  
 XMLEmbeddedExpression

XMLAttribute ::=  
 XMLWhitespace XMLAttributeName [ XMLWhitespace ] = [ XMLWhitespace ] XMLAttributeValue |  
 XMLWhitespace XMLEmbeddedExpression

XMLAttributeName ::=  
 XMLQualifiedNameOrExpression |  
 XMLNamespaceAttributeName

XMLAttributeValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter |  
 XMLEmbeddedExpression

XMLAttributeDoubleQuoteValueCharacter ::=   
 < Any XMLCharacter except <, &, or DoubleQuoteCharacter > |  
 XMLReference

XMLAttributeSingleQuoteValueCharacter ::=  
 < Any XMLCharacter except <, &, or SingleQuoteCharacter > |  
 XMLReference

XMLReference ::= XMLEntityReference | XMLCharacterReference

XMLEntityReference ::=  
 & XMLEntityName ;

XMLEntityName ::= lt | gt | amp | apos | quot

XMLCharacterReference ::=  
 & # XMLNumericCharacter+ ; |  
 & # x XMLHexNumericCharacter+ ;

### XML 命名空间

依据 XML 命名空间 1.0 规范，XML 元素可以包含 XML 命名空间声明。有关定义命名空间 xml 和 xmlns 的限制将被强制执行，并且将发生编译时错误。XML 命名空间声明的值中不能包含嵌入式表达式；提供的值必须是非空字符串文本。例如：

' Declares a valid namespace  
Dim customer = <db:customer xmlns:db="http://example.org/database">Bob</>  
  
' Error: xmlns cannot be re-defined  
Dim bad1 = <elem xmlns:xmlns="http://example.org/namespace"/>  
  
' Error: cannot have an embedded expression  
Dim bad2 = <elem xmlns:db=<%= "http://example.org/database" %>>Bob</>

批注

此规范仅包含用于描述 Visual Basic 语言行为的足够的 XML 命名空间说明。有关 XML 命名空间的更多信息，请参见 [http://www.w3.org/TR/REC-xml-names/](http://vstfdevdiv:8080/web/wi.aspx)。

可以使用命名空间名称来限定 XML 元素和特性名称。命名空间在常规 XML 中是绑定的，只不过可将在文件级别声明的任何命名空间导入视为是在一个包含此声明的上下文中声明的，而该上下文自身包含在由编译环境声明的任何命名空间导入中。如果找不到命名空间名称，则将发生编译时错误。例如：

Imports System.Xml.Linq  
Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 ' Binds to the imported namespace above.  
 Dim c1 = <db:customer>Bob</>  
  
 ' Binds to the namespace declaration in the element  
 Dim c2 = \_  
 <db:customer xmlns:db="http://example.org/database-other">Mary</>  
  
 ' Binds to the inner namespace declaration  
 Dim c3 = \_  
 <database xmlns:db="http://example.org/database-one">  
 <db:customer xmlns:db="http://example.org/database-two">Joe</>  
 </>  
  
 ' Error: namespace db2 cannot be found  
 Dim c4 = \_  
 <db2:customer>Jim</>  
 End Sub  
End Module

元素中声明的 XML 命名空间不适用于嵌入式表达式内部的 XML 文本。例如：

' Error: Namespace prefix 'db' is not declared  
Dim customer = \_  
 <db:customer xmlns:db="http://example.org/database">  
 <%= <db:customer>Bob</> %>  
 </>

批注

这是因为嵌入式表达式可以是任何内容，包括函数调用。如果函数调用包含 XML 文本表达式，则无法确定程序员是希望应用还是忽略 XML 命名空间。

XMLNamespaceAttributeName ::=  
 XMLPrefixedNamespaceAttributeName |  
 XMLDefaultNamespaceAttributeName

XMLPrefixedNamespaceAttributeName ::=  
 xmlns : XMLNamespaceName

XMLDefaultNamespaceAttributeName ::=  
 xmlns

XMLNamespaceName ::= XMLNamespaceNameStartCharacter [ XMLNamespaceNameCharacter+ ]

XMLNamespaceNameStartCharacter ::=  
 < Any XMLNameCharacter except : >

XMLNamespaceNameCharacter ::= XMLLetter | \_

XMLQualifiedNameOrExpression ::= XMLQualifiedName | XMLEmbeddedExpression

XMLQualifiedName ::=  
 XMLPrefixedName |  
 XMLUnprefixedName

XMLPrefixedName ::= XMLNamespaceName : XMLNamespaceName

XMLUnprefixedName ::= XMLNamespaceName

### XML 处理指令

XML 处理指令会产生类型化为 System.Xml.Linq.XProcessingInstruction 的值。XML 处理指令不能包含嵌入式表达式，因为它们在处理指令中是有效语法。

XMLProcessingInstruction ::=  
 < ? XMLProcessingTarget [ XMLWhitespace [ XMLProcessingValue ] ] ? >

XMLProcessingTarget ::=  
 < Any XMLName except a casing permutation of the string "xml" >

XMLProcessingValue ::=  
 < Any XMLString that does not contain the string "?>" >

### XML 注释

XML 注释会产生类型化为 System.Xml.Linq.XComment 的值。XML 注释不能包含嵌入式表达式，因为它们在注释中是有效语法。

XMLComment ::=  
 < ! - - [ XMLCommentCharacter+ ] - - >

XMLCommentCharacter ::=  
 < Any XMLCharacter except dash (0x002D) > |  
 - < Any XMLCharacter except dash (0x002D) >

### CDATA 节

CDATA 节会产生类型化为 System.Xml.Linq.XCData 的值。CDATA 节不能包含嵌入式表达式，因为它们在 CDATA 节中是有效语法。

XMLCDATASection ::=  
 < ! [ CDATA [ [ XMLCDATASectionString ] ] ] >

XMLCDATASectionString ::=  
 < Any XMLString that does not contain the string "]]>" >

## XML 成员访问表达式

XML 成员访问表达式用于访问 XML 值的成员。有以下三类 XML 成员访问表达式：

元素访问，其中 XML 名称跟在一个点后面。例如：

Dim customer = \_  
 <customer>  
 <name>Bob</>  
 </>  
Dim customerName = customer.<name>.Value

元素访问映射到函数：

Function Elements(name As System.Xml.Linq.XName) As \_  
 System.Collections.Generic.IEnumerable(Of \_  
 System.Xml.Linq.XNode)

因此，上面的示例等效于：

Dim customerName = customer.Elements("name").Value

特性访问，其中 Visual Basic 标识符跟在一个点和一个 @ 符号后面，或 XML 名称跟在一个点和一个 @ 符号后面。例如：

Dim customer = <customer age="30"/>  
Dim customerAge = customer.@age

特性访问映射到函数：

Function AttributeValue(name As System.Xml.Linq.XName) as String

因此，上面的示例等效于：

Dim customerAge = customer.AttributeValue("age")

批注

当前未在任何程序集中定义 AttributeValue 扩展方法（以及相关的扩展属性 Value）。如果需要扩展成员，则将在所产生的程序集中自动定义它们。

子代访问，其中 XML 名称跟在三个点后面。例如：

Dim company = \_  
 <company>  
 <customers>  
 <customer>Bob</>  
 <customer>Mary</>  
 <customer>Joe</>  
 </>  
 </>  
Dim customers = company...<customer>

子代访问映射到函数：

Function Descendents(name As System.Xml.Linq.XName) As \_  
 System.Collections.Generic.IEnumerable(Of \_  
 System.Xml.Linq.XElement)

因此，上面的示例等效于：

Dim customers = company.Descendants("customer")

XML 成员访问表达式的基表达式必须是一个值，并且必须属于以下类型：

对于元素访问或子代访问，为 System.Xml.Linq.XContainer 或派生类型，或者为 System.Collections.Generic.IEnumerable(Of T) 或派生类型，其中 T 是 System.Xml.Linq.XContainer 或派生类型。

对于特性访问，为 System.Xml.Linq.XElement 或派生类型，或者为 System.Collections.Generic.IEnumerable(Of T) 或派生类型，其中 T 是 System.Xml.Linq.XElement 或派生类型。

XML 成员访问表达式中的名称不能为空。可以使用由导入定义的任何命名空间对这些名称进行命名空间限定。例如：

Imports <xmlns:db="http://example.org/database">  
  
Module Test  
 Sub Main()  
 Dim customer = \_  
 <db:customer>  
 <db:name>Bob</>  
 </>  
 Dim name = customer.<db:name>  
 End Sub  
End Module

在 XML 成员访问表达式中，点的后面不能有空格，或者尖括号和名称之间不能有空格。例如：

Dim customer = \_  
 <customer age="30">  
 <name>Bob</>  
 </>  
' All the following are error cases  
Dim age = customer.@ age  
Dim name = customer.< name >  
Dim names = customer...< name >

如果 System.Xml.Linq 命名空间中的类型不可用，则 XML 文本表达式将导致发生编译时错误。

XMLMemberAccessExpression ::=  
 Expression . [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] IdentifierOrKeyword |  
 Expression . . . [ LineTerminator ] < XMLQualifiedName >

## Await 运算符

await 运算符与在 10.1.3 节中描述的异步方法相关。

如果包含 Await 的直接封闭方法或 lambda 表达式具有 Async 修饰符，并且 Await 出现在该 Async 修饰符后面，则 Await 是一个保留字；它在其他位置不是保留字。Yield 在预处理器指令中也不是保留字。await 运算符只能在方法体或 lambda 表达式中作为保留字使用。在直接封闭方法或 lambda 中，Catch 或 Finally 块体、SyncLock 语句体及查询表达式中不得使用 await 表达式。

await 运算符采用单一表达式，此表达式必须分类为值，并且其类型必须是*可等待* 类型或者是 Object。如果其类型为 Object，则会将所有处理一直推迟到运行时。如果以下所有条件均为 true，则称类型 C 是可等待的：

1. C 包含一个名为 GetAwaiter 的可访问实例或扩展方法，此实例或扩展方法没有实参，并且会返回某个类型 E；
2. E 包含一个名为 IsCompleted 的可读实例或扩展属性，此实例或扩展属性没有实参，其类型为 Boolean；
3. E 包含一个名为 GetResult 的可访问实例或扩展方法，此实例或扩展方法无实参；
4. E 实现 System.Runtime.CompilerServices.INotifyCompletion 或 ICriticalNotifyCompletion。

如果 GetResult 是 Sub，则会将 await 表达式分类为 void。否则，将 await 表达式分类为值，其类型是 GetResult 方法的返回类型。

以下是可等待的类的示例：

Class MyTask(Of T)  
 Function GetAwaiter() As MyTaskAwaiter(Of T)  
 Return New MyTaskAwaiter With {.m\_Task = Me}  
 End Function

...  
End Class

Structure MyTaskAwaiter(Of T)  
 Implements INotifyCompletion  
  
 Friend m\_Task As MyTask(Of T)

ReadOnly Property IsCompleted As Boolean  
 Get  
 Return m\_Task.IsCompleted  
 End Get  
 End Property

Sub OnCompleted(r As Action) Implements INotifyCompletion.OnCompleted  
 ' r is the "resumptionDelegate"  
 Dim sc = SynchronizationContext.Current  
 If sc Is Nothing Then  
 m\_Task.ContinueWith(Sub() r())  
 Else  
 m\_Task.ContinueWith(Sub() sc.Post(Sub() r(), Nothing))  
 End If  
 End Sub

Function GetResult() As T  
 If m\_Task.IsCanceled Then Throw New TaskCanceledException(m\_Task)  
 If m\_Task.IsFaulted Then Throw m\_Task.Exception.InnerException  
 Return m\_Task.Result  
 End Function  
End Structure

批注

建议库作者在与调用 OnCompleted 时相同的 SynchronizationContext 上遵循他们调用延续委托的模式。此外，不应在 OnCompleted 方法内同步执行恢复委托（因为这样会导致堆栈溢出），是应对该委托进行排队以便随后执行。

当控制流到达 Await 运算符时，行为如下。

1. 调用 await 操作数的 GetAwaiter 方法。此调用的结果称为 *awaiter*。
2. 对 awaiter 的 IsCompleted 属性进行检索。如果结果为 true，则：
   1. 调用 awaiter 的 GetResult 方法。如果 GetResult 是函数，则 await 表达式的值是此函数的返回值。
3. 如果 IsCompleted 属性不为 true，则：
   1. 对 awaiter 调用 ICriticalNotifyCompletion.UnsafeOnCompleted（如果 awaiter 的类型 E 实现 ICriticalNotifyCompletion）或 INotifyCompletion.OnCompleted（其他情况）。在这两种情况下，它都会传递与该异步方法的当前实例关联的*恢复委托*。
   2. 当前异步方法实例的控制点挂起，控制流在*当前调用方*（在 10.1.3 节中定义）中恢复。
   3. 如果随后调用恢复委托，
      1. 则恢复委托首先将 System.Threading.Thread.CurrentThread.ExecutionContext 恢复到调用 OnCompleted 时的状态，
      2. 然后在异步方法实例的控制点处恢复控制流（请参见 10.1.3 节），
      3. 在此处，它会调用 awaiter 的 GetResult 方法（如上面的 2.1 所述）。

如果 await 操作数的类型为 Object，则将此行为一直推迟到运行时：

* 步骤 1 是通过调用不带实参的 GetAwaiter() 完成的；因此，它在运行时可能会绑定到采用可选形参的实例方法。
* 步骤 2 是通过检索不带实参的 IsCompleted() 属性并尝试进行向 Boolean 的内部转换完成的。
* 步骤 3.1 是通过尝试 TryCast(awaiter, ICriticalNotifyCompletion) 完成的，如果失败，则尝试 DirectCast(awaiter, INotifyCompletion)。

在 3.1 中传递的恢复委托只能调用一次。如果对其调用一次以上，则行为将不确定。

AwaitOperatorExpression ::= Await Expression

# 文档注释

文档注释是源中经过特殊格式设置的注释，可以通过分析这些注释来创建有关这些注释将附加到的代码的文档。文档注释的基本格式为 XML。在编译带文档注释的代码时，编译器可以选择发出一个表示源中的文档注释总数的 XML 文件。然后，其他工具可以使用此 XML 文件来创建打印文档或联机文档。

本章介绍了文档注释和用于文档注释的建议的 XML 标记。

## 文档注释格式

文档注释是一类特殊的注释，它以三个单引号 ''' 开头。这些注释后面必须紧跟它们所记录的类型（如类、委托或接口）或者类型成员（如字段、事件、属性或方法）。分部方法声明的文档注释将替换为提供其主体的方法的文档注释（如果有）。可以将所有相邻的文档注释追加在一起来生成一个文档注释。如果 ''' 字符后面跟有一个空格，则串联中不会包含此空格字符。例如：

''' <remarks>  
''' Class <c>Point</c> models a point in a two-dimensional plane.  
''' </remarks>  
Public Class Point   
 ''' <remarks>  
 ''' Method <c>Draw</c> renders the point.  
 ''' </remarks>  
 Sub Draw()  
 End Sub  
End Class

依据 [http://www.w3.org/TR/REC-xml](http://vstfdevdiv:8080/web/wi.aspx)，文档注释必须是格式正确的 XML。如果 XML 的格式不正确，则将生成警告，并且文档文件将包含一条注释，指出遇到了错误。

尽管开发人员可自由创建他们自己的标记集，但下一节定义了建议的标记集。某些建议的标记具有特殊含义：

<param> 标记用于描述参数。<param> 标记指定的参数必须存在，并且文档注释必须描述类型成员的所有参数。如果上述任一条件未得到满足，则编译器将发出警告。

cref 特性可以附加到任意标记，以提供对代码元素的引用。该代码元素必须存在；在编译时，编译器会将名称替换为表示成员的 ID 字符串。如果该代码元素不存在，则编译器将发出警告。在查找 cref 特性中描述的名称时，编译器将考虑在包含源文件中出现的 Imports 语句。

<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。

请注意，文档文件不提供有关类型和成员的完整信息，而仅提供文档注释中包含的内容。若要获取有关类型或成员的更多信息，文档文件必须与实际类型或成员上的反射一起使用。

## 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：

<c> 将文本设置为类似代码的字体

<code> 将一行或多行源代码或程序输出设置为类似代码的字体

<example> 标识示例

<exception> 标识方法可能引发的异常

<include> 包含外部 XML 文档

<list> 创建列表或表

<para> 用于将结构添加到文本中

<param> 描述方法或构造函数的参数

<paramref> 确认某个单词是参数名

<permission> 描述成员的安全性和访问权限

<remarks> 描述类型

<returns> 描述方法的返回值

<see> 指定链接

<seealso> 生成“请参见”项

<summary> 描述类型的成员

<typeparam> 描述类型参数

<value> 描述属性

### <c>

此标记指定说明中的文本段落应使用与代码块所使用的字体类似的字体。（对于实际代码行，请使用 <code>）。

语法：

<c>text to be set like code</c>

示例：

''' <remarks>  
 ''' Class <c>Point</c> models a point in a two-dimensional plane.  
 ''' </remarks>  
 Public Class Point   
 End Class

### <code>

此标记指定一行或多行源代码或程序输出应使用等宽字体。（对于较小的代码段，请使用 <c>。）

语法：

<code>source code or program output</code>

示例：

''' <summary>  
 ''' This method changes the point's location by the given x- and   
 ''' y-offsets.  
 ''' <example>  
 ''' For example:  
 ''' <code>  
 ''' Dim p As Point = New Point(3,5)  
 ''' p.Translate(-1,3)  
 ''' </code>  
 ''' results in <c>p</c>'s having the value (2,8).  
 ''' </example>  
 ''' </summary>  
 Public Sub Translate(x As Integer, y As Integer)  
 Me.x += x  
 Me.y += y  
 End Sub

### <example>

此标记可让注释中的代码示例说明如何使用元素。通常，这还将涉及到标记 <code> 的使用。

语法：

<example>description</example>

示例：

有关示例，请参见 <code>。

### <exception>

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>

示例：

Public Module DataBaseOperations  
 ''' <exception cref="MasterFileFormatCorruptException"></exception>  
 ''' <exception cref="MasterFileLockedOpenException"></exception>  
 Public Sub ReadRecord(flag As Integer)  
 If Flag = 1 Then  
 Throw New MasterFileFormatCorruptException()  
 ElseIf Flag = 2 Then  
 Throw New MasterFileLockedOpenException()  
 End If  
 ' ...  
 End Sub  
 End Module

### <include>

此标记用于包含来自格式良好的外部 XML 文档的信息。可将 XPath 表达式应用于 XML 文档，以指定应从该文档包含的 XML。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">

示例：

如果源代码包含了与以下声明类似的声明：

''' <include file="docs.xml" path="extra/class[@name="IntList"]/\*" />

并且外部文件 docs.xml 具有以下内容

<?xml version="1.0"?>  
 <extra>  
 <class name="IntList">  
 <summary>  
 Contains a list of integers.  
 </summary>  
 </class>  
 <class name="StringList">  
 <summary>  
 Contains a list of strings.  
 </summary>  
 </class>  
 </extra>

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>

### <list>

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词条。）

列表中的每一项都用一个 <item> 块来指定。创建定义列表时，必须同时指定词条和说明。但对于表、项目符号列表或编号列表，只需指定说明。

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>

示例：

Public Class TestClass  
 ''' <remarks>  
 ''' Here is an example of a bulleted list:  
 ''' <list type="bullet">  
 ''' <item>  
 ''' <description>Item 1.</description>  
 ''' </item>  
 ''' <item>  
 ''' <description>Item 2.</description>  
 ''' </item>  
 ''' </list>  
 ''' </remarks>  
 Public Shared Sub Main()  
 End Sub  
 End Class

### <para>

此标记用于其他标记内，如 <remarks> 或 <returns>，用于将结构添加到文本中。

语法：

<para>content</para>

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 Shared Sub Main()  
 End Sub

### <param>

此标记用于描述方法、构造函数或索引属性的参数。

语法：

<param name="name">description</param>

示例：

''' <summary>  
 ''' This method changes the point's location to the given  
 ''' coordinates.  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new y-coordinate.</param>  
 Public Sub Move(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub

### <paramref>

此标记指示某个单词是一个参数。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"/>

示例：

''' <summary>  
 ''' This constructor initializes the new Point to  
 ''' (<paramref name="x"/>,<paramref name="y"/>).  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new Point's x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new Point's y-coordinate.</param>  
 Public Sub New(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub

### <permission>

此标记用于记录成员的安全可访问性。

语法：

<permission cref="member">description</permission>

示例：

''' <permission cref="System.Security.PermissionSet">Everyone can  
 ''' access this method.</permission>  
 Public Shared Sub Test()  
 End Sub

### <remarks>

此标记可指定有关类型的概述信息。（使用 <summary> 可描述类型的成员。）

语法：

<remarks>description</remarks>

示例：

''' <remarks>  
 ''' Class <c>Point</c> models a point in a two-dimensional plane.  
 ''' </remarks>  
 Public Class Point   
 End Class

### <returns>

此标记用于描述方法的返回值。

语法：

<returns>description</returns>

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, training, or embedded whitespace.  
 ''' </returns>  
 Public Overrides Function ToString() As String  
 Return "(" & x & "," & y & ")"  
 End Sub

### <see>

此标记用于在文本内指定链接。（使用 <seealso> 指定将在“请参见”部分中出现的文本。）

语法：

<see cref="member"/>

示例：

''' <summary>  
 ''' This method changes the point's location to the given  
 ''' coordinates.  
 ''' </summary>  
 ''' <see cref="Translate"/>  
 Public Sub Move(x As Integer, y As Integer)  
 Me.x = x  
 Me.y = y  
 End Sub  
  
 ''' <summary>  
 ''' This method changes the point's location by the given x- and  
 ''' y-offsets.  
 ''' </summary>  
 ''' <see cref="Move"/>  
 Public Sub Translate(x As Integer, y As Integer)  
 Me.x += x  
 Me.y += y  
 End Sub

### <seealso>

此标记用于生成将列入“请参见”部分的项。（使用 <see> 可指定文本中的链接。）

语法：

<seealso cref="member"/>

示例：

''' <summary>  
 ''' This method determines whether two Points have the same location.  
 ''' </summary>  
 ''' <seealso cref="operator=="/>  
 ''' <seealso cref="operator!="/>  
 Public Overrides Function Equals(o As Object) As Boolean  
 ' ...  
 End Function

### <summary>

此标记用于描述类型成员。（使用 <remarks> 可描述类型本身。）

语法：

<summary>description</summary>

示例：

''' <summary>  
 ''' This constructor initializes the new Point to (0,0).  
 ''' </summary>  
 Public Sub New()  
 Me.New(0,0)  
 End Sub

### <typeparam>

此标记用于描述类型参数。

语法：

<typeparam name="name">description</typeparam>

示例：

''' <typeparam name="T">  
 ''' The base item type. Must implement IComparable.  
 ''' </typeparam>  
 Public Class ItemManager(Of T As IComparable)  
 End Class

### <value>

此标记用于描述属性。

语法：

<value>property description</value>

示例：

''' <value>  
 ''' Property <c>X</c> represents the point's x-coordinate.  
 ''' </value>  
 Public Property X() As Integer  
 Get  
 Return \_x  
 End Get  
 Set (Value As Integer)  
 \_x = Value  
 End Set  
 End Property

## ID 字符串

在生成文档文件时，编译器将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，用于唯一地标识该元素。此 ID 字符串可供外部工具用来识别已编译的程序集中与文档注释对应的元素。

ID 字符串可按如下方式生成：

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。以下各类成员将通过其后面括号内的相应字符来定义：事件 (E)、字段 (F)、包含构造函数和运算符的方法 (M)、命名空间 (N)、属性 (P) 和类型 (T)。感叹号 (!) 指示在生成 ID 字符串时发生了错误，并且该字符串的其余部分提供了有关该错误的信息。

字符串的第二部分是元素的完全限定名（从全局命名空间开始）。元素的名称、其封闭类型和命名空间都以句点分隔。如果项的名称本身包含句点，则用井号 (#) 替换这些句点。（假定任何元素的名称内都不包含此字符。）带类型参数的类型的名称的结尾是一个反引号 (`)，接着是一个表示该类型具有的类型参数数的数字。请务必记住，由于嵌套类型可以访问其所属类型的类型参数，因此嵌套类型隐式包含其所属类型的类型参数，在此情况下，嵌套类型所属类型的类型参数总数中将计入这些类型。

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。每个参数的编码与 CLI 签名相同，如下所示：参数由其完全限定名表示。例如，Integer 变为 System.Int32，String 变为 System.String，Object 变为 System.Object，依此类推。带 ByRef 修饰符的参数的类型名称后有一个“@”符号。具有 ByVal、Optional 或 ParamArray 修饰符的实参没有特殊表示法。作为数组的参数表示为 [lowerbound:size, …, lowerbound:size]，其中逗号数为秩减 1 得到的数，而每个维度的下限和大小（如果已知）用十进制表示。如果未指定下限或大小，则将其省略。如果省略了特定维度的下限和大小，则“:”也将被省略。每个级别的嵌套数组都由一个“[]”表示。

### ID 字符串示例

以下每个示例均演示一个 VB 代码段以及从每个可具有文档注释的源元素产生的 ID 字符串：

类型用其完全限定名来表示。

Enum Color  
 Red  
 Blue  
 Green  
End Enum  
  
Namespace Acme  
 Interface IProcess  
 End Interface  
  
 Structure ValueType  
 ...  
 End Structure  
  
 Class Widget  
 Public Class NestedClass  
 End Class  
  
 Public Interface IMenuItem  
 End Interface  
  
 Public Delegate Sub Del(i As Integer)  
  
 Public Enum Direction  
 North  
 South  
 East  
 West  
 End Enum  
 End Class  
End Namespace  
  
"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"

字段用其完全限定名来表示。

Namespace Acme  
 Structure ValueType  
 Private total As Integer  
 End Structure  
  
 Class Widget  
 Public Class NestedClass  
 Private value As Integer  
 End Class  
  
 Private message As String  
 Private Shared defaultColor As Color  
 Private Const PI As Double = 3.14159  
 Protected ReadOnly monthlyAverage As Double  
 Private array1() As Long  
 Private array2(,) As Widget  
 End Class  
End Namespace  
  
"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"

构造函数。

Namespace Acme  
 Class Widget  
 Shared Sub New()  
 End Sub  
  
 Public Sub New()  
 End Sub  
  
 Public Sub New(s As String)  
 End Sub  
 End Class  
End Namespace  
  
"M:Acme.Widget.#cctor"  
"M:Acme.Widget.#ctor"  
"M:Acme.Widget.#ctor(System.String)"

方法。

Namespace Acme  
 Structure ValueType  
 Public Sub M(i As Integer)  
 End Sub  
 End Structure  
  
 Class Widget  
 Public Class NestedClass  
 Public Sub M(i As Integer)  
 End Sub  
 End Class  
  
 Public Shared Sub M0()  
 End Sub  
  
 Public Sub M1(c As Char, ByRef f As Float, \_  
 ByRef v As ValueType)  
 End Sub  
  
 Public Sub M2(x1() As Short, x2(,) As Integer, \_  
 x3()() As Long)  
 End Sub  
  
 Public Sub M3(x3()() As Long, x4()(,,) As Widget)  
 End Sub  
  
 Public Sub M4(Optional i As Integer = 1)  
  
 Public Sub M5(ParamArray args() As Object)  
 End Sub  
 End Class  
End Namespace  
  
"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.Int32)”  
“M:Acme.Widget.M5(System.Object[])"

属性。

Namespace Acme  
 Class Widget  
 Public Property Width() As Integer  
 Get  
 End Get  
 Set (Value As Integer)  
 End Set  
 End Property  
  
 Public Default Property Item(i As Integer) As Integer  
 Get  
 End Get  
 Set (Value As Integer)  
 End Set  
 End Property  
  
 Public Default Property Item(s As String, \_  
 i As Integer) As Integer  
 Get  
 End Get  
 Set (Value As Integer)  
 End Set  
 End Property  
 End Class  
End Namespace  
  
"P:Acme.Widget.Width"  
"P:Acme.Widget.Item(System.Int32)"  
"P:Acme.Widget.Item(System.String,System.Int32)"

事件

Namespace Acme  
 Class Widget  
 Public Event AnEvent As EventHandler  
 Public Event AnotherEvent()  
 End Class  
End Namespace  
  
"E:Acme.Widget.AnEvent"  
"E:Acme.Widget.AnotherEvent"

运算符。

Namespace Acme  
 Class Widget  
 Public Shared Operator +(x As Widget) As Widget  
 End Operator  
  
 Public Shared Operator +(x1 As Widget, x2 As Widget) As Widget  
 End Operator  
 End Class  
End Namespace  
  
"M:Acme.Widget.op\_UnaryPlus(Acme.Widget)"  
"M:Acme.Widget.op\_Addition(Acme.Widget,Acme.Widget)"

转换运算符具有一个后跟返回类型的尾随“~”。

Namespace Acme  
 Class Widget  
 Public Shared Narrowing Operator CType(x As Widget) As \_  
 Integer  
 End Operator  
  
 Public Shared Widening Operator CType(x As Widget) As Long  
 End Operator  
 End Class  
End Namespace  
  
"M:Acme.Widget.op\_Explicit(Acme.Widget)~System.Int32"  
"M:Acme.Widget.op\_Implicit(Acme.Widget)~System.Int64"

## 文档注释示例

以下示例演示 Point 类的源代码：

Namespace Graphics  
 ''' <remarks>  
 ''' Class <c>Point</c> models a point in a two-dimensional  
 ''' plane.  
 ''' </remarks>  
 Public Class Point  
 ''' <summary>  
 ''' Instance variable <c>x</c> represents the point's x-coordinate.  
 ''' </summary>  
 Private \_x As Integer  
  
 ''' <summary>  
 ''' Instance variable <c>y</c> represents the point's y-coordinate.  
 ''' </summary>  
 Private \_y As Integer  
  
 ''' <value>  
 ''' Property <c>X</c> represents the point's x-coordinate.  
 ''' </value>  
 Public Property X() As Integer  
 Get  
 Return \_x  
 End Get  
 Set(Value As Integer)  
 \_x = Value  
 End Set  
 End Property  
  
 ''' <value>  
 ''' Property <c>Y</c> represents the point's y-coordinate.  
 ''' </value>  
 Public Property Y() As Integer  
 Get  
 Return \_y  
 End Get  
 Set(Value As Integer)  
 \_y = Value  
 End Set  
 End Property  
  
 ''' <summary>  
 ''' This constructor initializes the new Point to (0,0).  
 ''' </summary>  
 Public Sub New()  
 Me.New(0, 0)  
 End Sub  
  
 ''' <summary>  
 ''' This constructor initializes the new Point to  
 ''' (<paramref name="x"/>,<paramref name="y"/>).  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new Point's  
 ''' x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new Point's  
 ''' y-coordinate.</param>  
 Public Sub New(x As Integer, y As Integer)  
 Me.X = x  
 Me.Y = y  
 End Sub  
  
 ''' <summary>  
 ''' This method changes the point's location to the given  
 ''' coordinates.  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the new x-coordinate.</param>  
 ''' <param name="y"><c>y</c> is the new y-coordinate.</param>  
 ''' <see cref="Translate"/>  
 Public Sub Move(x As Integer, y As Integer)  
 Me.X = x  
 Me.Y = y  
 End Sub  
  
 ''' <summary>  
 ''' This method changes the point's location by the given x- and  
 ''' y-offsets.  
 ''' <example>  
 ''' For example:  
 ''' <code>  
 ''' Dim p As Point = New Point(3, 5)  
 ''' p.Translate(-1, 3)  
 ''' </code>  
 ''' results in <c>p</c>'s having the value (2,8).  
 ''' </example>  
 ''' </summary>  
 ''' <param name="x"><c>x</c> is the relative x-offset.</param>  
 ''' <param name="y"><c>y</c> is the relative y-offset.</param>  
 ''' <see cref="Move"/>  
 Public Sub Translate(x As Integer, y As Integer)  
 Me.X += x  
 Me.Y += y  
 End Sub  
  
 ''' <summary>  
 ''' This method determines whether two Points have the same  
 ''' location.  
 ''' </summary>  
 ''' <param name="o"><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 op\_Equality"/>  
 ''' <seealso cref="Operator op\_Inequality"/>  
 Public Overrides Function Equals(o As Object) As Boolean  
 If o Is Nothing Then  
 Return False  
 End If  
 If o Is Me Then  
 Return True  
 End If  
 If Me.GetType() Is o.GetType() Then  
 Dim p As Point = CType(o, Point)  
 Return (X = p.X) AndAlso (Y = p.Y)  
 End If  
 Return False  
 End Function  
  
 ''' <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 Overrides Function ToString() As String  
 Return "(" & X & "," & Y & ")"  
 End Function  
  
 ''' <summary>  
 ''' This operator determines whether two Points have the  
 ''' same location.  
 ''' </summary>  
 ''' <param name="p1"><c>p1</c> is the first Point to be compared.  
 ''' </param>  
 ''' <param name="p2"><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="op\_Inequality"/>  
 Public Shared Operator =(p1 As Point, p2 As Point) As Boolean  
 If p1 Is Nothing OrElse p2 Is Nothing Then  
 Return False  
 End If  
 If p1.GetType() Is p2.GetType() Then  
 Return (p1.X = p2.X) AndAlso (p1.Y = p2.Y)  
 End If  
 Return False  
 End Operator  
  
 ''' <summary>  
 ''' This operator determines whether two Points have the  
 ''' same location.  
 ''' </summary>  
 ''' <param name="p1"><c>p1</c> is the first Point to be comapred.  
 ''' </param>  
 ''' <param name="p2"><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="op\_Equality"/>  
 Public Shared Operator <>(p1 As Point, p2 As Point) As Boolean  
 Return Not p1 = p2  
 End Operator  
  
 ''' <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 Shared Sub Main()  
 ' class test code goes here  
 End Sub  
 End Class  
End Namespace

以下是为类 Point 给定源代码（如上所示）后产生的输出：

<?xml version="1.0"?>  
<doc>  
 <assembly>  
 <name>Point</name>  
 </assembly>  
 <members>  
 <member name="T:Graphics.Point">  
 <remarks>Class <c>Point</c> models a point in a  
 two-dimensional plane. </remarks>  
 </member>  
 <member name="F:Graphics.Point.x">  
 <summary>Instance variable <c>x</c> represents the point's  
 x-coordinate.</summary>  
 </member>  
 <member name="F:Graphics.Point.y">  
 <summary>Instance variable <c>y</c> represents the point's  
 y-coordinate.</summary>  
 </member>  
 <member name="M:Graphics.Point.#ctor">  
 <summary>This constructor initializes the new Point to  
 (0,0).</summary>  
 </member>  
 <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">  
 <summary>This constructor initializes the new Point to  
 (<paramref name="x"/>,<paramref name="y"/>).</summary>  
 <param><c>x</c> is the new Point's x-coordinate.</param>  
 <param><c>y</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>x</c> is the new x-coordinate.</param>  
 <param><c>y</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>x</c> is the relative x-offset.</param>  
 <param><c>y</c> is the relative y-offset.</param>  
 <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>  
 </member>  
 <member name="M:Graphics.Point.Equals(System.Object)">  
 <summary>This method determines whether two Points have the  
 same location.</summary>  
 <param><c>o</c> is the object to be compared to the current  
 object.</param>  
 <returns>True if the Points have the same location and they  
 have the exact same type; otherwise, false.</returns>  
 <seealso cref=  
 "M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"  
 />  
 <seealso cref=  
 "M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"  
 />  
 </member>  
 <member name="M:Graphics.Point.ToString">  
 <summary>Report a point's location as a string.</summary>  
 <returns>A string representing a point's location, in the form  
 (x,y), without any leading, training, or embedded  
 whitespace.</returns>  
 </member>  
 <member name=  
 "M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)">  
 <summary>This operator determines whether two Points have the  
 same location.</summary>  
 <param><c>p1</c> is the first Point to be compared.</param>  
 <param><c>p2</c> is the second Point to be compared.</param>  
 <returns>True if the Points have the same location and they  
 have the exact same type; otherwise, false.</returns>  
 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>  
 <seealso cref=  
 "M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"  
 />  
 </member>  
 <member name=  
 "M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)">  
 <summary>This operator determines whether two Points have the  
 same location.</summary>  
 <param><c>p1</c> is the first Point to be compared.</param>  
 <param><c>p2</c> is the second Point to be compared.</param>  
 <returns>True if the Points do not have the same location and  
 the exact same type; otherwise, false.</returns>  
 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>  
 <seealso cref=  
 "M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"  
 />  
 </member>  
 <member name="M:Graphics.Point.Main">  
 <summary>This is the entry point of the Point class testing  
 program.  
 <para>This program tests each method and operator, and  
 is intended to be run after any non-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>

# 语法摘要

本节概括介绍了 Visual Basic 语言文法。有关如何阅读文法的信息，请参见“文法表示法”。

## 词法文法

Start ::= [ LogicalLine+ ]

LogicalLine ::= [ LogicalLineElement+ ] [ Comment ] LineTerminator

LogicalLineElement ::= WhiteSpace | LineContinuation | Token

Token ::= Identifier | Keyword | Literal | Separator | Operator

### 字符和行

Character ::= < any Unicode character except a LineTerminator >

LineTerminator ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character > < Unicode linefeed character > |  
 < Unicode line separator character (0x2028) > |  
 < Unicode paragraph separator character (0x2029) >

LineContinuation ::= WhiteSpace \_ [ WhiteSpace+ ] LineTerminator

Comma ::= , [ LineTerminator ]

OpenParenthesis ::= ( [ LineTerminator ]

CloseParenthesis ::= [ LineTerminator ] )

OpenCurlyBrace ::= { [ LineTerminator ]

CloseCurlyBrace ::= [ LineTerminator ] }

Equals ::= = [ LineTerminator ]

ColonEquals ::= : = [ LineTerminator ]

WhiteSpace ::=  
 < Unicode blank characters (class Zs) > |  
 < Unicode tab character (0x0009) >

Comment ::= CommentMarker [ Character+ ]

CommentMarker ::= SingleQuoteCharacter | REM

SingleQuoteCharacter ::=  
 ' |  
 < Unicode left single-quote character (0x2018) > |  
 < Unicode right single-quote character (0x2019) >

### 标识符

Identifier ::=  
 NonEscapedIdentifier [ TypeCharacter ] |  
 Keyword TypeCharacter |  
 EscapedIdentifier

NonEscapedIdentifier ::= < IdentifierName but not Keyword >

EscapedIdentifier ::= [ IdentifierName ]

IdentifierName ::= IdentifierStart [ IdentifierCharacter+ ]

IdentifierStart ::=  
 AlphaCharacter |  
 UnderscoreCharacter IdentifierCharacter

IdentifierCharacter ::=  
 UnderscoreCharacter |  
 AlphaCharacter |  
 NumericCharacter |  
 CombiningCharacter |  
 FormattingCharacter

AlphaCharacter ::=  
 < Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >

NumericCharacter ::= < Unicode decimal digit character (class Nd) >

CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >

FormattingCharacter ::= < Unicode formatting character (class Cf) >

UnderscoreCharacter ::= < Unicode connection character (class Pc) >

IdentifierOrKeyword ::= Identifier | Keyword

TypeCharacter ::=  
 IntegerTypeCharacter |  
 LongTypeCharacter |  
 DecimalTypeCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 StringTypeCharacter

IntegerTypeCharacter ::= %

LongTypeCharacter ::= &

DecimalTypeCharacter ::= @

SingleTypeCharacter ::= !

DoubleTypeCharacter ::= #

StringTypeCharacter ::= $

### 关键字

Keyword ::= < member of keyword table in 2.3 >

### 文本

Literal ::=  
 BooleanLiteral |  
 IntegerLiteral |  
 FloatingPointLiteral |  
 StringLiteral |  
 CharacterLiteral |  
 DateLiteral |  
 Nothing

BooleanLiteral ::= True | False

IntegerLiteral ::= IntegralLiteralValue [ IntegralTypeCharacter ]

IntegralLiteralValue ::= IntLiteral | HexLiteral | OctalLiteral

IntegralTypeCharacter ::=  
 ShortCharacter |  
 UnsignedShortCharacter |  
 IntegerCharacter |  
 UnsignedIntegerCharacter |  
 LongCharacter |  
 UnsignedLongCharacter |  
 IntegerTypeCharacter |  
 LongTypeCharacter

ShortCharacter ::= S

UnsignedShortCharacter ::= US

IntegerCharacter ::= I

UnsignedIntegerCharacter ::= UI

LongCharacter ::= L

UnsignedLongCharacter ::= UL

IntLiteral ::= Digit+

HexLiteral ::= & H HexDigit+

OctalLiteral ::= & O OctalDigit+

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

OctalDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

FloatingPointLiteral ::=  
 FloatingPointLiteralValue [ FloatingPointTypeCharacter ] |  
 IntLiteral FloatingPointTypeCharacter

FloatingPointTypeCharacter ::=  
 SingleCharacter |  
 DoubleCharacter |  
 DecimalCharacter |  
 SingleTypeCharacter |  
 DoubleTypeCharacter |  
 DecimalTypeCharacter

SingleCharacter ::= F

DoubleCharacter ::= R

DecimalCharacter ::= D

FloatingPointLiteralValue ::=  
 IntLiteral . IntLiteral [ Exponent ] |  
 . IntLiteral [ Exponent ] |  
 IntLiteral Exponent

Exponent ::= E [ Sign ] IntLiteral

Sign ::= + | -

StringLiteral ::=  
 DoubleQuoteCharacter [ StringCharacter+ ] DoubleQuoteCharacter

DoubleQuoteCharacter ::=  
 " |  
 < Unicode left double-quote character (0x201C) > |  
 < Unicode right double-quote character (0x201D) >

StringCharacter ::=  
 < Character except for DoubleQuoteCharacter > |  
 DoubleQuoteCharacter DoubleQuoteCharacter

CharacterLiteral ::= DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C

DateLiteral ::= # [ Whitespace+ ] DateOrTime [ Whitespace+ ] #

DateOrTime ::=  
 DateValue Whitespace+ TimeValue |  
 DateValue |  
 TimeValue

DateValue ::=  
 MonthValue / DayValue / YearValue |  
 MonthValue - DayValue - YearValue

TimeValue ::=  
 HourValue : MinuteValue [ : SecondValue ] [ WhiteSpace+ ] [ AMPM ] |  
 HourValue [ WhiteSpace+ ] AMPM

MonthValue ::= IntLiteral

DayValue ::= IntLiteral

YearValue ::= IntLiteral

HourValue ::= IntLiteral

MinuteValue ::= IntLiteral

SecondValue ::= IntLiteral

AMPM ::= AM | PM

ElseIf ::= ElseIf | Else If

Nothing ::= Nothing

Separator ::= ( | ) | { | } | ! | # | , | . | : | ?

Operator ::=  
 & | \* | + | - | / | \ | ^ | < | = | >

## 预处理指令

### 条件编译

Start ::= [ CCStatement+ ]

CCStatement ::=  
 CCConstantDeclaration |  
 CCIfGroup |  
 LogicalLine

CCExpression ::=  
 LiteralExpression |  
 CCParenthesizedExpression |  
 CCSimpleNameExpression |  
 CCCastExpression |  
 CCOperatorExpression |  
 CCConditionalExpression

CCParenthesizedExpression ::= ( CCExpression )

CCSimpleNameExpression ::= Identifier

CCCastExpression ::=   
 DirectCast ( CCExpression , TypeName ) |  
 TryCast ( CCExpression , TypeName ) |  
 CType ( CCExpression , TypeName ) |  
 CastTarget ( CCExpression )

CCOperatorExpression ::=  
 CCUnaryOperator CCExpression |  
 CCExpression CCBinaryOperator CCExpression

CCUnaryOperator ::= + | - | Not

CCBinaryOperator ::= + | - | \* | / | \ | Mod | ^ | = | < > | < | > |  
 < = | > = | & | And | Or | Xor | AndAlso | OrElse | < < | > >

CCConditionalExpression ::=   
 If ( CCExpression , CCExpression , CCExpression ) |  
 If ( CCExpression , CCExpression )

CCConstantDeclaration ::= # Const Identifier = CCExpression LineTerminator

CCIfGroup ::=  
 # If CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]  
 [ CCElseIfGroup+ ]  
 [ CCElseGroup ]  
 # End If LineTerminator

CCElseIfGroup ::=  
 # ElseIf CCExpression [ Then ] LineTerminator  
 [ CCStatement+ ]

CCElseGroup ::=  
 # Else LineTerminator  
 [ CCStatement+ ]

### 外部源指令

Start ::= [ ExternalSourceStatement+ ]

ExternalSourceStatement ::= ExternalSourceGroup | LogicalLine

ExternalSourceGroup ::=  
 # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator  
 [ LogicalLine+ ]  
 # End ExternalSource LineTerminator

### 区域指令

Start ::= [ RegionStatement+ ]

RegionStatement ::= RegionGroup | LogicalLine

RegionGroup ::=  
 # Region StringLiteral LineTerminator  
 [ RegionStatement+ ]  
 # End Region LineTerminator

### 外部校验和指令

Start ::= [ ExternalChecksumStatement+ ]

ExternalChecksumStatement ::=  
 # ExternalChecksum ( StringLiteral , StringLiteral , StringLiteral ) LineTerminator

## 句法文法

AccessModifier ::= Public | Protected | Friend | Private | Protected Friend

TypeParameterList ::=  
 OpenParenthesis Of TypeParameters CloseParenthesis

TypeParameters ::=  
 TypeParameter |  
 TypeParameters Comma TypeParameter

TypeParameter ::=  
 [ VarianceModifier ] Identifier [ TypeParameterConstraints ]

VarianceModifier ::=  
 In | Out

TypeParameterConstraints ::=  
 As Constraint |  
 As OpenCurlyBrace ConstraintList CloseCurlyBrace

ConstraintList ::=  
 ConstraintList *Comma* Constraint |  
 Constraint

Constraint ::= TypeName | New | Structure | Class

### 特性

Attributes ::=  
 AttributeBlock |  
 Attributes AttributeBlock

AttributeBlock ::= [ LineTerminator ] < AttributeList [ LineTerminator ] > [ LineTerminator ]

AttributeList ::=  
 Attribute |  
 AttributeList *Comma* Attribute

Attribute ::=  
 [ AttributeModifier : ] SimpleTypeName [ OpenParenthesis [ AttributeArguments ] CloseParenthesis ]

AttributeModifier ::= Assembly | Module

AttributeArguments ::=  
 AttributePositionalArgumentList |  
 AttributePositionalArgumentList Comma VariablePropertyInitializerList |  
 VariablePropertyInitializerList

AttributePositionalArgumentList ::=  
 AttributeArgumentExpression |  
 AttributePositionalArgumentList Comma AttributeArgumentExpression

VariablePropertyInitializerList ::=  
 VariablePropertyInitializer |  
 VariablePropertyInitializerList Comma VariablePropertyInitializer

VariablePropertyInitializer ::=  
 IdentifierOrKeyword ColonEquals AttributeArgumentExpression

AttributeArgumentExpression ::=  
 ConstantExpression |  
 GetTypeExpression |  
 ArrayExpression

### 源文件和命名空间

Start ::=  
 [ OptionStatement+ ]  
 [ ImportsStatement+ ]  
 [ AttributesStatement+ ]  
 [ NamespaceMemberDeclaration+ ]

StatementTerminator ::= LineTerminator | :

AttributesStatement ::= Attributes StatementTerminator

OptionStatement ::=  
 OptionExplicitStatement |  
 OptionStrictStatement |  
 OptionCompareStatement |  
 OptionInferStatement

OptionExplicitStatement ::= Option Explicit [ OnOff ] StatementTerminator

OnOff ::= On | Off

OptionStrictStatement ::= Option Strict [ OnOff ] StatementTerminator

OptionCompareStatement ::= Option Compare CompareOption StatementTerminator

CompareOption ::= Binary | Text

OptionInferStatement ::= Option Infer [ OnOff ] StatementTerminator

ImportsStatement ::= Imports ImportsClauses StatementTerminator

ImportsClauses ::=  
 ImportsClause |  
 ImportsClauses Comma ImportsClause

ImportsClause ::=  
 AliasImportsClause |  
 MembersImportsClause |  
 XMLNamespaceImportsClause

AliasImportsClause ::=   
 Identifier Equals TypeName

MembersImportsClause ::=  
 TypeName

XMLNamespaceImportsClause ::=  
 < XMLNamespaceAttributeName [ XMLWhitespace ] *Equals* [ XMLWhitespace ] XMLNamespaceValue >

XMLNamespaceValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter

NamespaceDeclaration ::=  
 Namespace NamespaceName StatementTerminator  
 [ NamespaceMemberDeclaration+ ]  
 End Namespace StatementTerminator

NamespaceName ::=   
 RelativeNamespaceName |  
 Global |  
 Global . RelativeNamespaceName

*RelativeNamespaceName ::=*  
 Identifier |  
 *Relative*NamespaceName Period IdentifierOrKeyword

NamespaceMemberDeclaration ::=  
 NamespaceDeclaration |  
 TypeDeclaration

TypeDeclaration ::=  
 ModuleDeclaration |  
 NonModuleDeclaration

NonModuleDeclaration ::=  
 EnumDeclaration |  
 StructureDeclaration |  
 InterfaceDeclaration |  
 ClassDeclaration |  
 DelegateDeclaration

### 类型

TypeName ::=  
 ArrayTypeName |  
 NonArrayTypeName

NonArrayTypeName ::=  
 SimpleTypeName | NullableTypeName

SimpleTypeName ::=  
 QualifiedTypeName |  
 BuiltInTypeName

QualifiedTypeName ::=  
 Identifier [ TypeArguments ] |  
 Global Period IdentifierOrKeyword [ TypeArguments ] |  
 QualifiedTypeName Period IdentifierOrKeyword [ TypeArguments ]

TypeArguments ::=  
 OpenParenthesis Of TypeArgumentList CloseParenthesis

TypeArgumentList ::=  
 TypeName |  
 TypeArgumentList Comma TypeName

BuiltInTypeName ::= Object | PrimitiveTypeName

TypeModifier ::= AccessModifier | Shadows

IdentifierModifiers ::= [ *NullableNameModifier* ] [ ArrayNameModifier ]

NullableTypeName ::= NonArrayTypeName ?

NullableNameModifier ::= ?

TypeImplementsClause ::= Implements *Type*Implements StatementTerminator

TypeImplements ::=  
 NonArrayTypeName |  
 *Type*Implements Comma NonArrayTypeName

PrimitiveTypeName ::= NumericTypeName | Boolean | Date | Char | String

NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal

IntegralTypeName ::= Byte | SByte | UShort | Short | UInteger | Integer | ULong | Long

FloatingPointTypeName ::= Single | Double

EnumDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Enum Identifier [ As NonArrayTypeName ] StatementTerminator  
 EnumMemberDeclaration+  
 End Enum StatementTerminator

EnumMemberDeclaration ::= [ Attributes ] Identifier [ Equals ConstantExpression ] StatementTerminator

ClassDeclaration ::=  
 [ Attributes ] [ ClassModifier+ ] Class Identifier [ TypeParameterList ] StatementTerminator  
 [ ClassBase ]  
 [ TypeImplementsClause+ ]  
 [ ClassMemberDeclaration+ ]  
 End Class StatementTerminator

ClassModifier ::= TypeModifier | MustInherit | NotInheritable | Partial

ClassBase ::= Inherits NonArrayTypeName StatementTerminator

ClassMemberDeclaration ::=  
 NonModuleDeclaration |  
 EventMemberDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

StructureDeclaration ::=  
 [ Attributes ] [ StructureModifier+ ] Structure Identifier [ TypeParameterList ]  
 StatementTerminator  
 [ TypeImplementsClause+ ]  
 [ StructMemberDeclaration+ ]  
 End Structure StatementTerminator

StructureModifier ::= TypeModifier | Partial

StructMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration |  
 OperatorDeclaration

ModuleDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Module Identifier StatementTerminator  
 [ ModuleMemberDeclaration+ ]  
 End Module StatementTerminator

ModuleMemberDeclaration ::=  
 NonModuleDeclaration |  
 VariableMemberDeclaration |  
 ConstantMemberDeclaration |  
 EventMemberDeclaration |  
 MethodMemberDeclaration |  
 PropertyMemberDeclaration |  
 ConstructorMemberDeclaration

InterfaceDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Interface Identifier [ TypeParameterList ] StatementTerminator  
 [ InterfaceBase+ ]  
 [ InterfaceMemberDeclaration+ ]  
 End Interface StatementTerminator

InterfaceBase ::= Inherits InterfaceBases StatementTerminator

InterfaceBases ::=  
 NonArrayTypeName |  
 InterfaceBases Comma NonArrayTypeName

InterfaceMemberDeclaration ::=  
 NonModuleDeclaration |  
 InterfaceEventMemberDeclaration |  
 InterfaceMethodMemberDeclaration |  
 InterfacePropertyMemberDeclaration

ArrayTypeName ::= NonArrayTypeName ArrayTypeModifiers

ArrayTypeModifiers ::= ArrayTypeModifier+

ArrayTypeModifier ::= OpenParenthesis [ RankList ] CloseParenthesis

RankList ::=  
 Comma |  
 RankList Comma

ArrayNameModifier ::=  
 ArrayTypeModifiers |  
 ArraySizeInitializationModifier

DelegateDeclaration ::=  
 [ Attributes ] [ TypeModifier+ ] Delegate MethodSignature StatementTerminator

MethodSignature ::= SubSignature | FunctionSignature

### 类型成员

ImplementsClause ::= [ Implements ImplementsList ]

ImplementsList ::=  
 InterfaceMemberSpecifier |  
 ImplementsList Comma InterfaceMemberSpecifier

InterfaceMemberSpecifier ::= NonArrayTypeName Period IdentifierOrKeyword

MethodMemberDeclaration ::= MethodDeclaration | ExternalMethodDeclaration

InterfaceMethodMemberDeclaration ::= InterfaceMethodDeclaration

MethodDeclaration ::=  
 SubDeclaration |  
 MustOverrideSubDeclaration |  
 FunctionDeclaration |  
 MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=  
 InterfaceSubDeclaration |  
 InterfaceFunctionDeclaration

SubSignature ::= Sub Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]

FunctionSignature ::= Function Identifier [ TypeParameterList ]  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] [ As [ Attributes ] TypeName ]

SubDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] SubSignature [ HandlesOrImplements ] LineTerminator  
 Block  
 End Sub StatementTerminator

MustOverrideSubDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ SubSignature [ HandlesOrImplements ]  
 StatementTerminator

InterfaceSubDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] SubSignature StatementTerminator

FunctionDeclaration ::=  
 [ Attributes ] [ ProcedureModifier+ ] FunctionSignature [ HandlesOrImplements ]  
 LineTerminator  
 Block  
 End Function StatementTerminator

MustOverrideFunctionDeclaration ::=  
 [ Attributes ] MustOverrideProcedureModifier+ FunctionSignature  
 [ HandlesOrImplements ] StatementTerminator

InterfaceFunctionDeclaration ::=  
 [ Attributes ] [ InterfaceProcedureModifier+ ] FunctionSignature StatementTerminator

ProcedureModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads |  
 Partial |  
 Iterator |  
 Async

MustOverrideProcedureModifier ::= ProcedureModifier | MustOverride

InterfaceProcedureModifier ::= Shadows | Overloads

HandlesOrImplements ::= HandlesClause | ImplementsClause

ExternalMethodDeclaration ::=  
 ExternalSubDeclaration |  
 ExternalFunctionDeclaration

ExternalSubDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Sub Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ] StatementTerminator

ExternalFunctionDeclaration ::=  
 [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Function Identifier  
 LibraryClause [ AliasClause ] [ OpenParenthesis [ ParameterList ] CloseParenthesis ] [ As [ Attributes ] TypeName ]  
 StatementTerminator

ExternalMethodModifier ::= AccessModifier | Shadows | Overloads

CharsetModifier ::= Ansi | Unicode | Auto

LibraryClause ::= Lib StringLiteral

AliasClause ::= Alias StringLiteral

ParameterList ::=  
 Parameter |  
 ParameterList Comma Parameter

Parameter ::=  
 [ Attributes ] [ ParameterModifier+ ] ParameterIdentifier [ As TypeName ]  
 [ *Equals* ConstantExpression ]

ParameterModifier ::= ByVal | ByRef | Optional | ParamArray

ParameterIdentifier ::= Identifier IdentifierModifiers

HandlesClause ::= [ Handles EventHandlesList ]

EventHandlesList ::=  
 EventMemberSpecifier |  
 EventHandlesList Comma EventMemberSpecifier

EventMemberSpecifier ::=  
 Identifier Period IdentifierOrKeyword |  
 MyBase Period IdentifierOrKeyword |  
 Me Period IdentifierOrKeyword

ConstructorMemberDeclaration ::=  
 [ Attributes ] [ ConstructorModifier+ ] Sub New  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Sub StatementTerminator

ConstructorModifier ::= AccessModifier | Shared

EventMemberDeclaration ::=  
 RegularEventMemberDeclaration |  
 CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType [ ImplementsClause ]  
 StatementTerminator

InterfaceEventMemberDeclaration ::=  
 [ Attributes ] [ InterfaceEventModifiers+ ] Event Identifier ParametersOrType StatementTerminator

ParametersOrType ::=  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ] |  
 As NonArrayTypeName

EventModifiers ::= AccessModifier | Shadows | Shared

InterfaceEventModifiers ::= Shadows

CustomEventMemberDeclaration ::=  
 [ Attributes ] [ EventModifiers+ ] Custom Event Identifier As TypeName [ ImplementsClause ]  
 StatementTerminator  
 EventAccessorDeclaration+  
 End Event StatementTerminator

EventAccessorDeclaration ::=  
 AddHandlerDeclaration |  
 RemoveHandlerDeclaration |  
 RaiseEventDeclaration

AddHandlerDeclaration ::=  
 [ Attributes ] AddHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End AddHandler StatementTerminator

RemoveHandlerDeclaration ::=  
 [ Attributes ] RemoveHandler OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RemoveHandler StatementTerminator

RaiseEventDeclaration ::=  
 [ Attributes ] RaiseEvent OpenParenthesis ParameterList CloseParenthesis LineTerminator  
 [ Block ]  
 End RaiseEvent StatementTerminator

ConstantMemberDeclaration ::=  
 [ Attributes ] [ ConstantModifier+ ] Const ConstantDeclarators StatementTerminator

ConstantModifier ::= AccessModifier | Shadows

ConstantDeclarators ::=  
 ConstantDeclarator |  
 ConstantDeclarators Comma ConstantDeclarator

ConstantDeclarator ::= Identifier [ As TypeName ] Equals ConstantExpression StatementTerminator

VariableMemberDeclaration ::=  
 [ Attributes ] VariableModifier+ VariableDeclarators StatementTerminator

VariableModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 ReadOnly |  
 WithEvents |  
 Dim

VariableDeclarators ::=  
 VariableDeclarator |  
 VariableDeclarators Comma VariableDeclarator

VariableDeclarator ::=  
 VariableIdentifiers As ObjectCreationExpression |  
 VariableIdentifiers [ As TypeName ] [ Equals Expression ]

VariableIdentifiers ::=  
 VariableIdentifier |  
 VariableIdentifiers Comma VariableIdentifier

VariableIdentifier ::= Identifier IdentifierModifiers

ArraySizeInitializationModifier ::=  
 OpenParenthesis BoundList CloseParenthesis [ ArrayTypeModifiers ]

BoundList::=  
 Bound |  
 BoundList Comma Bound

Bound ::=  
 Expression |  
 0 To Expression

PropertyMemberDeclaration ::=  
 RegularPropertyMemberDeclaration |  
 MustOverridePropertyMemberDeclaration |  
 AutoPropertyMemberDeclaration

PropertySignature ::=  
 Property Identifier [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ]

RegularPropertyMemberDeclaration ::=  
 [ Attributes ] [ PropertyModifier+ ] PropertySignature [ ImplementsClause ] LineTerminator  
 PropertyAccessorDeclaration+  
 End Property StatementTerminator

MustOverridePropertyMemberDeclaration ::=  
 [ Attributes ] MustOverridePropertyModifier+ PropertySignature [ ImplementsClause ]  
 StatementTerminator

AutoPropertyMemberDeclaration ::=  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 [ As [ Attributes ] TypeName ] [ Equals Expression ] [ ImplementsClause ] LineTerminator |  
 [ Attributes ] [ AutoPropertyModifier+ ] Property Identifier  
 [ OpenParenthesis [ ParameterList ] CloseParenthesis ]  
 As [ Attributes ] New [ NonArrayTypeName  
 [ OpenParenthesis [ ArgumentList ] CloseParenthesis ] ] [ ObjectCreationExpressionInitializer ]  
 [ ImplementsClause ] LineTerminator

InterfacePropertyMemberDeclaration ::=  
 [ Attributes ] [ InterfacePropertyModifier+ ] PropertySignature StatementTerminator

AutoPropertyModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads

PropertyModifier ::=  
 AutoPropertyModifier |  
 Default |  
 ReadOnly |  
 WriteOnly |  
 Iterator

MustOverridePropertyModifier ::= PropertyModifier | MustOverride

InterfacePropertyModifier ::=  
 Shadows |  
 Overloads |  
 Default |  
 ReadOnly |  
 WriteOnly

PropertyAccessorDeclaration ::= PropertyGetDeclaration | PropertySetDeclaration

PropertyGetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Get LineTerminator  
 [ Block ]  
 End Get StatementTerminator

PropertySetDeclaration ::=  
 [ Attributes ] [ AccessModifier ] Set [ OpenParenthesis [ ParameterList ] CloseParenthesis ] LineTerminator  
 [ Block ]  
 End Set StatementTerminator

OperatorDeclaration ::=  
 [ Attributes ] [ OperatorModifier+ ] Operator OverloadableOperator OpenParenthesis ParameterList CloseParenthesis  
 [ As [ Attributes ] TypeName ] LineTerminator  
 [ Block ]  
 End Operator StatementTerminator

OperatorModifier ::= Public | Shared | Overloads | Shadows | Widening | Narrowing

OverloadableOperator ::=  
 + | - | \* | / | \ | & | Like | Mod | And | Or | Xor | ^ | < < | > > |  
 = | < > | > | < | > = | < = | Not | IsTrue | IsFalse | CType

### 语句

Statement ::=  
 LabelDeclarationStatement |  
 LocalDeclarationStatement |  
 WithStatement |  
 SyncLockStatement |  
 EventStatement |  
 AssignmentStatement |  
 InvocationStatement |  
 ConditionalStatement |  
 LoopStatement |  
 ErrorHandlingStatement |  
 BranchStatement |  
 ArrayHandlingStatement |  
 UsingStatement |  
 AwaitStatement |  
 YieldStatement

Block ::= [ Statements+ ]

LabelDeclarationStatement ::= LabelName :

LabelName ::= Identifier | IntLiteral

Statements ::=  
 [ Statement ] |  
 Statements : [ Statement ]

LocalDeclarationStatement ::= LocalModifier VariableDeclarators StatementTerminator

LocalModifier ::= Static | Dim | Const

WithStatement ::=  
 With Expression StatementTerminator  
 [ Block ]  
 End With StatementTerminator

SyncLockStatement ::=  
 SyncLock Expression StatementTerminator  
 [ Block ]  
 End SyncLock StatementTerminator

EventStatement ::=  
 RaiseEventStatement |  
 AddHandlerStatement |  
 RemoveHandlerStatement

RaiseEventStatement ::= RaiseEvent IdentifierOrKeyword [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]  
 StatementTerminator

AddHandlerStatement ::= AddHandler Expression Comma Expression StatementTerminator

RemoveHandlerStatement ::= RemoveHandler Expression Comma Expression StatementTerminator

AssignmentStatement ::=  
 RegularAssignmentStatement |  
 CompoundAssignmentStatement |  
 MidAssignmentStatement

RegularAssignmentStatement ::= Expression Equals Expression StatementTerminator

CompoundAssignmentStatement ::= Expression CompoundBinaryOperator [ LineTerminator ]  
 Expression StatementTerminator

CompoundBinaryOperator ::= ^ = | \* = | / = | \ = | + = | - = | & = | < < = | > > =

MidAssignmentStatement ::=  
 Mid [ $ ] OpenParenthesis Expression Comma Expression [ Comma Expression ] CloseParenthesis  
 Equals Expression StatementTerminator

InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator

ConditionalStatement ::= IfStatement | SelectStatement

IfStatement ::= BlockIfStatement | LineIfThenStatement

BlockIfStatement ::=  
 If BooleanExpression [ Then ] StatementTerminator  
 [ Block ]  
 [ ElseIfStatement+ ]  
 [ ElseStatement ]  
 End If StatementTerminator

ElseIfStatement ::=  
 ElseIf BooleanExpression [ Then ] StatementTerminator  
 [ Block ]

ElseStatement ::=  
 Else StatementTerminator  
 [ Block ]

LineIfThenStatement ::=  
 If BooleanExpression Then Statements [ Else Statements ] StatementTerminator

SelectStatement ::=  
 Select [ Case ] Expression StatementTerminator  
 [ CaseStatement+ ]  
 [ CaseElseStatement ]  
 End Select StatementTerminator

CaseStatement ::=  
 Case CaseClauses StatementTerminator  
 [ Block ]

CaseClauses ::=  
 CaseClause |  
 CaseClauses Comma CaseClause

CaseClause ::=  
 [ Is [ LineTerminator ] ] ComparisonOperator [ LineTerminator ] Expression |  
 Expression [ To Expression ]

ComparisonOperator ::= = | < > | < | > | > = | < =

CaseElseStatement ::=  
 Case Else StatementTerminator  
 [ Block ]

LoopStatement ::=  
 WhileStatement |  
 DoLoopStatement |  
 ForStatement |  
 ForEachStatement

WhileStatement ::=  
 While BooleanExpression StatementTerminator  
 [ Block ]  
 End While StatementTerminator

DoLoopStatement ::= DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=  
 Do [ WhileOrUntil BooleanExpression ] StatementTerminator  
 [ Block ]  
 Loop StatementTerminator

DoBottomLoopStatement ::=  
 Do StatementTerminator  
 [ Block ]  
 Loop WhileOrUntil BooleanExpression StatementTerminator

WhileOrUntil ::= While | Until

ForStatement ::=  
 For LoopControlVariable Equals Expression To Expression [ Step Expression ] StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

LoopControlVariable ::=  
 Identifier [ IdentifierModifiers As TypeName ] |  
 Expression

NextExpressionList ::=  
 Expression |  
 NextExpressionList Comma Expression

ForEachStatement ::=  
 For Each LoopControlVariable In [ LineTerminator ] Expression StatementTerminator  
 [ Block ]  
 [ Next [ NextExpressionList ] StatementTerminator ]

ErrorHandlingStatement ::=  
 StructuredErrorStatement |  
 UnstructuredErrorStatement

StructuredErrorStatement ::=  
 ThrowStatement |  
 TryStatement

TryStatement ::=  
 Try StatementTerminator  
 [ Block ]  
 [ CatchStatement+ ]  
 [ FinallyStatement ]  
 End Try StatementTerminator

FinallyStatement ::=  
 Finally StatementTerminator  
 [ Block ]

CatchStatement ::=  
 Catch [ Identifier [ As NonArrayTypeName ] ] [ When BooleanExpression ] StatementTerminator  
 [ Block ]

ThrowStatement ::= Throw [ Expression ] StatementTerminator

UnstructuredErrorStatement ::=  
 ErrorStatement |  
 OnErrorStatement |  
 ResumeStatement

ErrorStatement ::= Error Expression StatementTerminator

OnErrorStatement ::= On Error ErrorClause StatementTerminator

ErrorClause ::=  
 GoTo - 1 |  
 GoTo 0 |  
 GoToStatement |  
 Resume Next

ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator

ResumeClause ::= Next | LabelName

BranchStatement ::=  
 GoToStatement |  
 ExitStatement |  
 ContinueStatement |  
 StopStatement |  
 EndStatement |  
 ReturnStatement

GoToStatement ::= GoTo LabelName StatementTerminator

ExitStatement ::= Exit ExitKind StatementTerminator

ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try

ContinueStatement ::= Continue ContinueKind StatementTerminator

ContinueKind ::= Do | For | While

StopStatement ::= Stop StatementTerminator

EndStatement ::= End StatementTerminator

ReturnStatement ::= Return [ Expression ] StatementTerminator

ArrayHandlingStatement ::=  
 RedimStatement |  
 EraseStatement

RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator

RedimClauses ::=  
 RedimClause |  
 RedimClauses Comma RedimClause

RedimClause ::= Expression ArraySizeInitializationModifier

EraseStatement ::= Erase EraseExpressions StatementTerminator

EraseExpressions ::=  
 Expression |  
 EraseExpressions Comma Expression

UsingStatement ::=  
 Using UsingResources StatementTerminator  
 [ Block ]  
 End Using StatementTerminator

UsingResources ::= VariableDeclarators | Expression

AwaitStatement ::= AwaitOperatorExpression StatementTerminator

YieldStatement ::= Yield Expressions StatementTerminator

### 表达式

Expression ::=  
 SimpleExpression |  
 TypeExpression |  
 MemberAccessExpression |  
 DictionaryAccessExpression |  
 InvocationExpression |  
 IndexExpression |  
 NewExpression |  
 CastExpression |  
 OperatorExpression |  
 ConditionalExpression |  
 LambdaExpression |  
 QueryExpression |  
 XMLLiteralExpression |  
 XMLMemberAccessExpression

ConstantExpression ::= Expression

SimpleExpression ::=  
 LiteralExpression |  
 ParenthesizedExpression |  
 InstanceExpression |  
 SimpleNameExpression |  
 AddressOfExpression

LiteralExpression ::= Literal

ParenthesizedExpression ::= OpenParenthesis Expression CloseParenthesis

InstanceExpression ::= Me

SimpleNameExpression ::= Identifier [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

AddressOfExpression ::= AddressOf Expression

TypeExpression ::=  
 GetTypeExpression |  
 TypeOfIsExpression |  
 IsExpression |  
 GetXmlNamespaceExpression

GetTypeExpression ::= GetType OpenParenthesis GetTypeTypeName CloseParenthesis

GetTypeTypeName ::=  
 TypeName |  
 *Qualified*OpenTypeName

QualifiedOpenTypeName ::=  
 Identifier [ TypeArityList ] |  
 Global Period IdentifierOrKeyword [ TypeArityList ] |  
 QualifiedOpenTypeName Period IdentifierOrKeyword [ TypeArityList ]

*TypeArityList* ::= ( Of [ *CommaList* ] )

CommaList ::=  
 Comma |  
 CommaList Comma

TypeOfIsExpression ::= TypeOf Expression Is [ LineTerminator ] TypeName

IsExpression ::=  
 Expression Is [ LineTerminator ] Expression |  
 Expression IsNot [ LineTerminator ] Expression

GetXmlNamespaceExpression ::= GetXmlNamespace OpenParenthesis [ XMLNamespaceName ]  
 CloseParenthesis

MemberAccessExpression ::=  
 [ MemberAccessBase ] *Period* IdentifierOrKeyword  
 [ OpenParenthesis Of TypeArgumentList CloseParenthesis ]

MemberAccessBase ::=  
 Expression |  
 BuiltInTypeName |  
 Global |  
 MyClass |  
 MyBase

DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword

InvocationExpression ::= Expression [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]

ArgumentList ::= PositionalArgumentList |  
 PositionalArgumentList Comma NamedArgumentList |  
 NamedArgumentList

PositionalArgumentList ::=  
 [ Expression ] |  
 PositionalArgumentList Comma [ Expression ]

NamedArgumentList ::=  
 IdentifierOrKeyword ColonEquals Expression |  
 NamedArgumentList Comma IdentifierOrKeyword ColonEquals Expression

IndexExpression ::= Expression OpenParenthesis [ ArgumentList ] CloseParenthesis

NewExpression ::=  
 ObjectCreationExpression |  
 ArrayExpression |  
 AnonymousObjectCreationExpression

AnonymousObjectCreationExpression ::=  
 New ObjectMemberInitializer

ObjectCreationExpression ::=  
 New NonArrayTypeName [ OpenParenthesis [ ArgumentList ] CloseParenthesis ]  
 [ ObjectCreationExpressionInitializer ]

ObjectCreationExpressionInitializer ::= ObjectMemberInitializer | ObjectCollectionInitializer

ObjectMemberInitializer ::=  
 With OpenCurlyBrace FieldInitializerList CloseCurlyBrace

FieldInitializerList ::=  
 FieldInitializer |  
 FieldInitializerList , FieldInitializer

FieldInitializer ::= [ [ Key ] . IdentifierOrKeyword = ] Expression

ObjectCollectionInitializer ::= From CollectionInitializer

CollectionInitializer ::= OpenCurlyBrace [ CollectionElementList ] CloseCurlyBrace

CollectionElementList ::=  
 CollectionElement |  
 CollectionElementList Comma CollectionElement

CollectionElement ::=  
 *Expression* |  
 CollectionInitializer

ArrayExpression ::=  
 *ArrayCreationExpression* |  
 *ArrayLiteralExpression*

ArrayCreationExpression ::=  
 New NonArrayTypeName ArrayNameModifier CollectionInitializerArrayLiteralExpression ::= CollectionInitializer

CastExpression ::=  
 DirectCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 TryCast OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CType OpenParenthesis Expression Comma TypeName CloseParenthesis |  
 CastTarget OpenParenthesis Expression CloseParenthesis

CastTarget ::=  
 CBool | CByte | CChar | CDate | CDec | CDbl | CInt | CLng | CObj | CSByte | CShort |  
 CSng | CStr | CUInt | CULng | CUShort

OperatorExpression ::=  
 ArithmeticOperatorExpression |  
 RelationalOperatorExpression |  
 LikeOperatorExpression |  
 ConcatenationOperatorExpression |  
 ShortCircuitLogicalOperatorExpression |  
 LogicalOperatorExpression |  
 ShiftOperatorExpression |  
 AwaitOperatorExpression

ArithmeticOperatorExpression ::=  
 UnaryPlusExpression |  
 UnaryMinusExpression |  
 AdditionOperatorExpression |  
 SubtractionOperatorExpression |  
 MultiplicationOperatorExpression |  
 DivisionOperatorExpression |  
 ModuloOperatorExpression |  
 ExponentOperatorExpression

UnaryPlusExpression ::= + Expression

UnaryMinusExpression ::= - Expression

AdditionOperatorExpression ::= Expression + [ LineTerminator ] Expression

SubtractionOperatorExpression ::= Expression - [ LineTerminator ] Expression

MultiplicationOperatorExpression ::= Expression \* [ LineTerminator ] Expression

DivisionOperatorExpression ::=  
 FPDivisionOperatorExpression |  
 IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= Expression / [ LineTerminator ] Expression

IntegerDivisionOperatorExpression ::= Expression \ [ LineTerminator ] Expression

ModuloOperatorExpression ::= Expression Mod [ LineTerminator ] Expression

ExponentOperatorExpression ::= Expression ^ [ LineTerminator ] Expression

RelationalOperatorExpression ::=  
 Expression = [ LineTerminator ] Expression |  
 Expression < > [ LineTerminator ] Expression |  
 Expression < [ LineTerminator ] Expression |  
 Expression > [ LineTerminator ] Expression |  
 Expression < = [ LineTerminator ] Expression |  
 Expression > = [ LineTerminator ] Expression

LikeOperatorExpression ::= Expression Like [ LineTerminator ] Expression

ConcatenationOperatorExpression ::= Expression & [ LineTerminator ] Expression

LogicalOperatorExpression ::=  
 Not Expression |  
 Expression And [ LineTerminator ] Expression |  
 Expression Or [ LineTerminator ] Expression |  
 Expression Xor [ LineTerminator ] Expression

ShortCircuitLogicalOperatorExpression ::=  
 Expression AndAlso [ LineTerminator ] Expression |  
 Expression OrElse [ LineTerminator ] Expression

ShiftOperatorExpression ::=  
 Expression < < [ LineTerminator ] Expression |  
 Expression > > [ LineTerminator ] Expression

BooleanExpression ::= Expression

LambdaExpression ::=  
 SingleLineLambda |  
 MultiLineLambda

SingleLineLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Expression |  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Statement

MultiLineLambda ::=  
 MultiLineFunctionLambda |  
 MultiLineSubLambda

MultiLineFunctionLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] [ As TypeName ] LineTerminator  
 Block  
 End Function

MultiLineSubLambda ::=  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] LineTerminator  
 Block  
 End Sub

LambdaModifier ::=  
 Async |  
 Iterator

QueryExpression ::=   
 FromOrAggregateQueryOperator |  
 QueryExpression QueryOperator

FromOrAggregateQueryOperator ::= FromQueryOperator | AggregateQueryOperator

JoinOrGroupJoinQueryOperator := JoinQueryOperator | GroupJoinQueryOperator

QueryOperator ::=  
 FromQueryOperator |  
 AggregateQueryOperator |  
 SelectQueryOperator |  
 DistinctQueryOperator |  
 WhereQueryOperator |  
 OrderByQueryOperator |  
 PartitionQueryOperator |  
 LetQueryOperator |  
 GroupByQueryOperator | *JoinOr*GroupJoinQueryOperator

CollectionRangeVariableDeclarationList ::=  
 CollectionRangeVariableDeclaration |  
 CollectionRangeVariableDeclarationList Comma CollectionRangeVariableDeclaration

CollectionRangeVariableDeclaration ::=   
 Identifier [ As TypeName ] In [ LineTerminator ] Expression

ExpressionRangeVariableDeclarationList ::=  
 ExpressionRangeVariableDeclaration |  
 ExpressionRangeVariableDeclarationList Comma ExpressionRangeVariableDeclaration

ExpressionRangeVariableDeclaration ::=   
 Identifier [ As TypeName ] Equals Expression

FromQueryOperator ::=  
 [ LineTerminator ] From [ LineTerminator ] CollectionRangeVariableDeclarationList

JoinQueryOperator ::=  
 [ LineTerminator ] Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList

JoinConditionList ::=  
 JoinCondition |  
 JoinConditionList And [ LineTerminator ] JoinCondition

JoinCondition ::= Expression Equals [ LineTerminator ] Expression

LetQueryOperator ::=  
 [ LineTerminator ] Let [ LineTerminator ] ExpressionRangeVariableDeclarationList

SelectQueryOperator ::=  
 [ LineTerminator ] Select [ LineTerminator ] ExpressionRangeVariableDeclarationList

DistinctQueryOperator ::=  
 [ LineTerminator ] Distinct [ LineTerminator ]

WhereQueryOperator ::=   
 [ LineTerminator ] Where [ LineTerminator ] BooleanExpression

PartitionQueryOperator ::=   
 [ LineTerminator ] Take [ LineTerminator ] Expression |  
 [ LineTerminator ] Take While [ LineTerminator ] BooleanExpression |  
 [ LineTerminator ] Skip [ LineTerminator ] Expression |  
 [ LineTerminator ] Skip While [ LineTerminator ] BooleanExpression

OrderByQueryOperator ::=  
 [ LineTerminator ] Order By [ LineTerminator ] OrderExpressionList

OrderExpressionList ::=  
 OrderExpression |  
 OrderExpressionList Comma OrderExpression

OrderExpression ::=  
 Expression [ Ordering ]

Ordering ::= Ascending | Descending

GroupByQueryOperator ::=  
 [ LineTerminator ] Group [ [ LineTerminator ] ExpressionRangeVariableDeclarationList ]  
 [ LineTerminator ] By [ LineTerminator ] ExpressionRangeVariableDeclarationList  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

AggregateQueryOperator ::=  
 [ LineTerminator ] Aggregate [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ QueryOperator+ ]  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

GroupJoinQueryOperator ::=  
 [ LineTerminator ] Group Join [ LineTerminator ] CollectionRangeVariableDeclaration  
 [ JoinOrGroupJoinQueryOperator ] [ LineTerminator ] On [ LineTerminator ] JoinConditionList  
 [ LineTerminator ] Into [ LineTerminator ] ExpressionRangeVariableDeclarationList

ConditionalExpression ::=   
 If OpenParenthesis BooleanExpression Comma Expression Comma Expression CloseParenthesis |  
 If OpenParenthesis Expression Comma Expression CloseParenthesis

XMLLiteralExpression ::=  
 XMLDocument |  
 XMLElement | XMLProcessingInstruction |  
 XMLComment |  
 XMLCDATASection

XMLCharacter ::=  
 < Unicode tab character (0x0009) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode carriage return character (0x000D) > |  
 < Unicode characters 0x0020 – 0xD7FF > |  
 < Unicode characters 0xE000 – 0xFFFD > |  
 < Unicode characters 0x10000 – 0x10FFFF >

XMLString ::= XMLCharacter+

XMLWhitespace ::= XMLWhitespaceCharacter+

XMLWhitespaceCharacter ::=  
 < Unicode carriage return character (0x000D) > |  
 < Unicode linefeed character (0x000A) > |  
 < Unicode space character (0x0020) > |  
 < Unicode tab character (0x0009) >

XMLNameCharacter ::= XMLLetter | XMLDigit | . | - | \_ | : | XMLCombiningCharacter | XMLExtender

XMLNameStartCharacter ::= XMLLetter | \_ | :

XMLName ::= XMLNameStartCharacter [ XMLNameCharacter+ ]

XMLLetter ::=   
 < Unicode character as defined in the Letter production of the XML 1.0 specification >

XMLDigit ::=  
 < Unicode character as defined in the Digit production of the XML 1.0 specification >

XMLCombiningCharacter ::=  
 < Unicode character as defined in the CombiningChar production of the XML 1.0 specification >

XMLExtender ::=  
 < Unicode character as defined in the Extender production of the XML 1.0 specification >

XMLEmbeddedExpression ::=  
 < % = [ LineTerminator ] Expression [ LineTerminator ] % >

XMLDocument ::=  
 XMLDocumentPrologue [ XMLMisc+ ] XMLDocumentBody [ XMLMisc+ ]

XMLDocumentPrologue ::=  
 < ? xml XMLVersion [ XMLEncoding ] [ XMLStandalone ] [ XMLWhitespace ] ? >

XMLVersion ::=  
 XMLWhitespace version [ XMLWhitespace ] = [ XMLWhitespace ] XMLVersionNumberValue

XMLVersionNumberValue ::=   
 SingleQuoteCharacter 1 . 0 SingleQuoteCharacter |  
 DoubleQuoteCharacter 1 . 0 DoubleQuoteCharacter

XMLEncoding ::=  
 XMLWhitespace encoding [ XMLWhitespace ] = [ XMLWhitespace ] XMLEncodingNameValue

XMLEncodingNameValue ::=   
 SingleQuoteCharacter XMLEncodingName SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLEncodingName DoubleQuoteCharacter

XMLEncodingName ::= XMLLatinAlphaCharacter [ XMLEncodingNameCharacter+ ]

XMLEncodingNameCharacter ::=  
 XMLUnderscoreCharacter |  
 XMLLatinAlphaCharacter |  
 XMLNumericCharacter |  
 XMLPeriodCharacter |  
 XMLDashCharacter

XMLLatinAlphaCharacter ::=  
 < Unicode Latin alphabetic character (0x0041-0x005a, 0x0061-0x007a) >

XMLNumericCharacter ::= < Unicode digit character (0x0030-0x0039) >

XMLHexNumericCharacter ::=  
 XMLNumericCharacter |  
 < Unicode Latin hex alphabetic character (0x0041-0x0046, 0x0061-0x0066) >

XMLPeriodCharacter ::= < Unicode period character (0x002e) >

XMLUnderscoreCharacter ::= < Unicode underscore character (0x005f) >

XMLDashCharacter ::= < Unicode dash character (0x002d) >

XMLStandalone ::=  
 XMLWhitespace standalone [ XMLWhitespace ] = [ XMLWhitespace ] XMLYesNoValue

XMLYesNoValue ::=   
 SingleQuoteCharacter XMLYesNo SingleQuoteCharacter |  
 DoubleQuoteCharacter XMLYesNo DoubleQuoteCharacter

XMLYesNo ::= yes | no

XMLMisc ::=  
 XMLComment |  
 XMLProcessingInstruction |  
 XMLWhitespace

XMLDocumentBody ::= XMLElement | XMLEmbeddedExpression

XMLElement ::=  
 XMLEmptyElement |  
 XMLElementStart XMLContent XMLElementEnd

XMLEmptyElement ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitepace ] / >

XMLElementStart ::=  
 < XMLQualifiedNameOrExpression [ XMLAttribute+ ] [ XMLWhitespace ] >

XMLElementEnd ::=  
 < / > |  
 < / XMLQualifiedName [ XMLWhitespace ] >

XMLContent ::=  
 [ XMLCharacterData ] [ XMLNestedContent [ XMLCharacterData ] ]+

XMLCharacterData ::=  
 < Any XMLCharacterDataString that does not contain the string "]]>" >

XMLCharacterDataString ::=  
 < Any Unicode character except < or & >+

XMLNestedContent ::=  
 XMLElement |  
 XMLReference |  
 XMLCDATASection |  
 XMLProcessingInstruction |  
 XMLComment |  
 XMLEmbeddedExpression

XMLAttribute ::=  
 XMLWhitespace XMLAttributeName [ XMLWhitespace ] = [ XMLWhitespace ] XMLAttributeValue |  
 XMLWhitespace XMLEmbeddedExpression

XMLAttributeName ::=  
 XMLQualifiedNameOrExpression |  
 XMLNamespaceAttributeName

XMLAttributeValue ::=  
 DoubleQuoteCharacter [ XMLAttributeDoubleQuoteValueCharacter+ ] DoubleQuoteCharacter |  
 SingleQuoteCharacter [ XMLAttributeSingleQuoteValueCharacter+ ] SingleQuoteCharacter |  
 XMLEmbeddedExpression

XMLAttributeDoubleQuoteValueCharacter ::=   
 < Any XMLCharacter except <, &, or DoubleQuoteCharacter > |  
 XMLReference

XMLAttributeSingleQuoteValueCharacter ::=  
 < Any XMLCharacter except <, &, or SingleQuoteCharacter > |  
 XMLReference

XMLReference ::= XMLEntityReference | XMLCharacterReference

XMLEntityReference ::=  
 & XMLEntityName ;

XMLEntityName ::= lt | gt | amp | apos | quot

XMLCharacterReference ::=  
 & # XMLNumericCharacter+ ; |  
 & # x XMLHexNumericCharacter+ ;

XMLNamespaceAttributeName ::=  
 XMLPrefixedNamespaceAttributeName |  
 XMLDefaultNamespaceAttributeName

XMLPrefixedNamespaceAttributeName ::=  
 xmlns : XMLNamespaceName

XMLDefaultNamespaceAttributeName ::=  
 xmlns

XMLNamespaceName ::= XMLNamespaceNameStartCharacter [ XMLNamespaceNameCharacter+ ]

XMLNamespaceNameStartCharacter ::=  
 < Any XMLNameCharacter except : >

XMLNamespaceNameCharacter ::= XMLLetter | \_

XMLQualifiedNameOrExpression ::= XMLQualifiedName | XMLEmbeddedExpression

XMLQualifiedName ::=  
 XMLPrefixedName |  
 XMLUnprefixedName

XMLPrefixedName ::= XMLNamespaceName : XMLNamespaceName

XMLUnprefixedName ::= XMLNamespaceName

XMLProcessingInstruction ::=  
 < ? XMLProcessingTarget [ XMLWhitespace [ XMLProcessingValue ] ] ? >

XMLProcessingTarget ::=  
 < Any XMLName except a casing permutation of the string "xml" >

XMLProcessingValue ::=  
 < Any XMLString that does not contain the string "?>" >

XMLComment ::=  
 < ! - - [ XMLCommentCharacter+ ] - - >

XMLCommentCharacter ::=  
 < Any XMLCharacter except dash (0x002D) > |  
 - < Any XMLCharacter except dash (0x002D) >

XMLCDATASection ::=  
 < ! [ CDATA [ [ XMLCDATASectionString ] ] ] >

XMLCDATASectionString ::=  
 < Any XMLString that does not contain the string "]]>" >

XMLMemberAccessExpression ::=  
 Expression . [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] < XMLQualifiedName > |  
 Expression . [ LineTerminator ] @ [ LineTerminator ] IdentifierOrKeyword |  
 Expression . . . [ LineTerminator ] < XMLQualifiedName >

AwaitOperatorExpression ::= Await Expression

# 更改列表

以下各节列出了对各个版本之间的规范所做的更改。每次更改后都列出受影响的节。

## 版本 7.1 到版本 8.0

此更新包括了对语言版本 7.1 和 语言版本 8.0 之间所做的语言更改。

### 主要更改

* 现在，在一个声明之前允许使用多个特性块，即，允许 <a> <b> 而不只是允许 <a,b>。[5.2, 6]
* 添加了 Continue 语句。[2.3, 10.11]
* 添加了 Using 语句。[2.3, 10, 10.13]
* 添加了 IsNot 运算符。[2.3, 11.5.3]
* 添加了允许在全局命名空间中使用绑定的 Global 限定符。[2.3, 4.7, 11.6]
* 添加了 XML 文档注释。[12]
* 允许派生类重新实现由它们的基类实现的接口。[4.4, 4.4.1]
* 添加了 TryCast 运算符。[2.3, 11.11]
* 特性可以具有类型化为 Object 或一维数组的参数。[5.1, 5.2.2]
* 添加了语言兼容性上的一节 [1.2, 1.2.1, 1.2.2, 1.2.3]
* 添加了运算符重载。[2.3, 4.1.1, 7.5.2, 7.6.1, 9.8, 9.8.1, 9.8.2, 9.8.3, 10.8.2, 10.9.2, 11.17.1, 11.11, 8.11, 8.11.1, 8.11.2, 11.12.3]
* 添加了伪运算符 IsTrue 和 IsFalse。[11.19, 10.8.1, 10.9.1, 10.10.1.2]
* 添加了无符号整数类型。[2.2.1, 2.3, 2.4.2, 7.3, 11.11, 7.4, 8.2, 8.3, 8.7, 8.8, 8.9, 10.9.2, 11.2, 11.12.3, 11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18, 11.8.1]
* 添加了自定义事件声明。[9.4.1]
* 属性访问器可以指定限制性高于它们的包含属性的访问级别限制性。[9.7, 9.7.1, 9.7.2, 9.7.3]
* 添加了分部类型。[2.3, 7.5, 7.6, 7.11]
* 添加了默认实例。[11.6.2, 11.6.2.1, 11.6.2.2]
* 添加了泛型类型和方法。[2.3, 2.4.7, 4.1.1, 4.4.1, 4.5.1, 4.6, 4.7.1, 4.7.2, 4.9, 4.9.1, 4.9.2, 5.1, 5.2.2, 6.1, 6.3.1, 6.3.2, 7, 7.2, 7.5, 7.5.1, 7.6, 7.8, 7.8.1, 7.9, 7.11, 7.12, 7.12.1, 8.6, 8.10, 9.1, 9.2.1, 9.2.2, 9.3.2, 9.4, 9.6, 9.8.1, 9.8.2, 9.8.3, 10.2, 10.9.3, 10.10.1.2, 11.1, 11.4.4, 11.4.5, 11.5.1, 11.5.2, 11.5.3, 11.6, 11.6.2, 11.8, 11.8.2, 11.8.5, 11.10.1, 11.10.2, 11.10.3, 12.2.16, 12.3]

### 次要更改

* 二进制字符串比较始终用于条件编译。否则，由于文本字符串比较依赖于运行时区域性，因此字符串比较将不再工作。[3.1.2]
* 类型不能继承自其中直接包含或间接包含的类型。还阐明了这些示例。[4.3]
* 更改了文法和规范，以便枚举类型可以使用基本类型的 System 等效项作为基础类型。[7.4]
* 现在，允许在枚举类型的数组和枚举的基础类型的数组之间进行转换。[8.5, 8.8.8.9]
* 当重写方法时，可以用 MustOverride 方法来重写它，使其成为抽象的。[4.5.1]
* 类型成员可以使用 Handles 在自己的类中处理事件。[9.2.6]
* 声明为 Overrides 的方法和属性现在采用 Overloads，这比采用 Shadows 更符合逻辑。[4.3.3]
* 允许字段和局部变量使用 As New 语法一次初始化多个变量。[9.6]
* 移除了内部 Catch 块不能分支到外部 Try 块中的限制。[10.10.1.2]
* 类不能从 System.MulticastDelegate 继承。[7.5.1]
* 在结构中共享的变量可以有初始值。[9.6.3]
* 添加了一个规则，此规则是：当针对文本 0 执行重载解决方案时数值类型优先于枚举类型。[11.8.1]
* 数组大小初始值可以显式声明下限为零。[9.6.3.3]
* 添加了外部校验和指令。[3.4]
* 允许字段上的数组大小的初始值为非常量的表达式。[9.6.3.3]
* 现在，将有类型字符的关键字视为标识符。[2.2]
* 与其类型具有相同名称的常量、字段、属性、局部变量和参数可解释为成员或以成员查找为目的的类型。[11.6.1]
* 添加了讨论有关由 .NET Framework 限制的类型的章节，并将有关 System.Void 的讨论移动到此章节中。[7, 7.13]
* 当处理定义相同的完全限定名的项目和引用程序集时，项目的类型是首选的，否则名称是不明确的。[4.7.2, 11.4.4]
* On Error 语句不会扩展到构造函数开头处对 New 的调用。[10.10.2]
* 如果包含类型是一个接口，则不允许解析对后期绑定调用的重载调用。[11.8.1]
* 标准模块中不允许有 Overloads 和 Shadows。[4.3.3]
* 当在接口中查找时，通过层次结构隐藏在某一路径中的名称将通过层次结构隐藏在所有路径中。以前，我们会给一个不明确的错误。[4.3.2]
* 当通过导入进行绑定时，类型和类型成员优先级高于命名空间。[4.7.2, 11.6]
* 更改类型的默认属性不再要求 Shadows。[9.7.3]
* 取消保留上下文关键字 Assembly、Ansi、Auto、Preserve、Unicode 和 Until。[2.3]

### 澄清/勘误表

* 添加了全宽/半宽仅在完整标记基础上工作的备注（例如，您不能将它混合到标记中。）[1.1]
* 在此语言中有些位置允许常规类型名称，但不允许数组类型名称。增加了对要将这些位置调用出来的语法的阐述。[5.2, 7, 7.2, 7.5.1, 7.8.1, 7.9, 9.1, 9.2.5, 10.10.1.2, 11.10.1, 11.10.2, 11.10.3]
* 此规范错误地说明了冒号在语句级别上只可以用作分隔符。实际上，差不多可在任何位置使用它们。阐明了语法和规范。[6, 6.2.1, 6.2.2, 6.2.3, 6.3, 6.4.1, 7.2, 7.4, 7.4.1, 7.5, 7.5.1, 7.6, 7.7, 7.8, 7.8.1, 9.2.1, 9.2.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.7.1, 9.7.2, 10]
* 作为以前项目符号的一部分，对它们自身标上语句标签并修正语法。[10, 10.1]
* 收缩了针对要更加准确的委托声明的语法的范围。[7.10]
* 数组协变还包括除接口之外的接口实现。[8.5]
* 当实现接口或处理事件时，句号后面的标识符可以匹配关键字。[9.1, 9.2.6]
* 将 MustOverride 声明从常规的方法和属性声明中分离出来。[9.2.1, 9.7]
* 有关变量声明的文法是错误的 – Dim 是一个常规修饰符，而不是一个单独的元素。[9.6]
* 指出如何将 WithEvents 字段的特性传输到基础的合成成员。[9.6.2]
* 如果没有侦听器，则调用事件将不会引发异常。[10.5.1]
* 对 RaiseEvent 节进行了扩展，介绍如何避免在引发事件时发生争用。[10.5.1]
* 澄清了以下问题：RaiseEvent 采用一个标识符作为参数，而不是采用一个异常。[10.5.1]
* 介绍了 Mid 赋值的边际情形。[10.6.3]
* If 行后面的语句不是可选的。[10.8.1]
* 扩展了 Do…Loop 语法，使其更加明确。[10.9.1]
* For 循环中的循环控制变量可以是数组类型。[10.9.2]
* 在数组创建表达式中不可选大小修饰符。[11.10.2]
* 属性返回类型可以具有特性 [9.7]
* 为了语法清晰，将事件、属性和方法声明的接口版本进行拆分。[7.8.2, 9.2, 9.2.1, 9.4, 9.7]
* MyClass 和 MyBase 不能独立存在，已将其移动到限定表达式产生式。[11.4.3, 11.6]
* TypeOf…Is 是运算符优先级的主类别的一部分。[11.12.1]
* 一个 Handles 子句可以具有多个标识符，但只有两个标识符是合法的。[9.2.6]
* 条件编译只支持常量表达式的子集。[3.1]
* 将对 System.Monitor 的引用更正为引用 System.Threading.Monitor。[10.4]
* 声明了编译器在默认情况下可以将声明置于特定命名空间中。[6.4]
* 在 Finally 块内部不会重新引发异常（不带实参的 Throw）。[10.10.1.3]
* 扩大考虑从类型转换到它自己。[8.8]
* 尝试比以前更好地澄清 DirectCast 的确切行为。[11.11]
* For Each 语句中的集合的元素类型不必具有到循环控制变量/表达式的隐式转换：转换可以是任意形式的。[10.9.3]
* 声明了小数除法行为。[11.13.6]
* 澄清了以下问题：在针对 & 运算符将 Nothing 转换到空字符串时，不考虑重载运算符。另外还表明了相同行为适用于 Like 运算符。[11.15, 11.16]
* 澄清了以下问题：包含 Object 参数的操作可能会导致出现结果并非 Integer 的情况。[11.12.2]
* 添加了显式运算符类型表。[11.12.3, 11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18]
* 尝试以使“最特定的”规则更加清晰为目的。[11.8.1]
* 澄清了以下问题：通过名称和签名隐藏 ParamArray 方法只会隐藏相应的签名，即使隐藏方法与 ParamArray 方法的未展开签名相匹配也是如此。[4.3.3]
* 已将重载决策过程中有关优先选择较少的参数数组匹配项的规则前移，以匹配编译器（和必需）的行为。[11.8.1, 11.8.2]
* 不能合并数组大小初始值和数组元素初始值。[9.6.3.3]
* 当指定数组创建表达式上的边界并提供数组元素初始值时，边界必须使用常量表达式来指定。[9.6.3.4]
* 添加了装箱和拆箱的讨论，以及到装箱类型设计的抗锯齿的限制。[8.6]
* 尝试澄清一条有关枚举类型和带有 Object 循环控制变量的 For 循环的模糊规则。[10.9.2]
* 澄清了以下问题：在 Object 循环过程中损坏循环控制变量并不会改变循环的类型。[10.9.2]
* 指明委托和外部方法可以使用重复的参数名称。[9.2.5]
* 接口具有到 Object 的扩大转换。[8.4, 8.8]
* 接口不从 Object 继承。[7.8.1]
* Object 是引用类型。它不是一个类型，此类型“不是引用类型，也不是值 类型。”[7]
* 指明了 System.ValueType 和 System.Enum 在值类型层次结构中的位置。[7.1]
* 指出在作为 Object 装箱时允许的基元类型转换。[8.6]
* 展开了委托的成员的解释。[7.10]
* 对有关隐式局部变量的讨论进行了扩充。[10.1.1, 10.2.1]
* 声明了静态初始值的工作原理。[10.2]
* 请注意，用名称绑定来忽略合成名称。[9.4, 9.4.1, 9.7.1, 9.7.2]
* 在 Try…Catch 块中捕获的异常将其异常存储在 Err 对象中。[10.10.1.2]
* 将此标识符转换的存在调用出来。[8]
* 声明了如何在表达式上下文中处理后期绑定访问。[11.1, 11.1.1, 11.3, 11.6, 11.8.1, 11.9]
* 更正共享构造函数执行时的规则。[9.3.2]

### 其他

* 从“Microsoft Visual Basic .NET”到“Microsoft Visual Basic”更改了对语言的名称的引用。语言自己的官方名称是简单的“Visual Basic”。
* 为了清楚起见，将多标记的标点符号和运算符（例如 <= 或 >>=）移到了词法文法部分。[2.5, 2.6, 5.2.2, 10.6.2, 10.8.2, 11.14]
* 删除了 NamespaceOrTypeName 产生式，因为实际上并不需要此产生式。[4.7, 6.3.1, 6.3.2, 7]
* 删除了局部变量产生式，因为它们是多余的。[10.2]
* 将所有只包含访问修饰符和 Shadows 的产生式合并为单个产生式。[7, 7.4, 7.5, 7.6, 7.7, 7.8, 7.10]
* 随着命令行的默认响应文件的出现，所有项目默认情况下都将导入 System，因此从所有示例中将它删除了。[计数太多。]
* 将预处理语句产生式的后缀从 Element 更改为 Statement，将条件编译语句的前缀从 Conditional 更改为 CC。[3.1, 3.1.1, 3.1.2, 3.2, 3.3]
* 更正了 Resume 语句节中的示例。[10.10.2.3]
* 将 Protected Friend 添加到修饰符产生式中。[4.6]
* 将一些示例添加到数组创建表达式。[11.10.2]

## 版本 8.0（第二版本）到版本 8.0

此更新涵盖非语言更改，它只包含更正和勘误表。

### 次要更改

* 现在允许对不引发事件的类型使用 WithEvents。[9.6.2]
* 将在编译时（而非运行时）捕获 ReDim 中维数的错误。[10.12.1]
* 常量表达式不能通过 With 上下文进行绑定。[11.2]
* Optional 参数不能类型化为无约束的类型参数。[9.2.5.3]
* MustInherit 类的默认构造函数是 Protected，而不是 Public。[9.3.1]
* 如果表达式绝不会是指定类型的表达式，则 TypeOf…Is 将发出一个编译时错误。[11.5.2]
* 如果将默认构造函数发出到设计器生成的类中，则它将调用 InitializeComponent。[9.3.1]
* 不允许通过类型约束访问值类型的成员。[4.9.2]
* 添加了 System.Nullable(Of T) 的特殊装箱/拆箱行为。[8.6]
* 一个方法可以是比另一个方法更适用的，即使参数类型不转换为每一个其他的类型，以便至少一个参数类型是更适用的。[11.8.1]
* 如果指定了 Structure 约束，则搜索 System.ValueType 有无成员。[4.9.2]
* 不允许循环结构定义。[7.6.1]

### 澄清/勘误表

* 更新了 SubSignature 和 FunctionSignature 以包括 Sub 和 Function 关键字，这样它们就会出现在委托产生式中。[9.2.1]
* 将丢失的 InvocationExpression 添加到 Expression 产生式。[11]
* 移除了重复的 VariableInitializer 产生式。[9.6.3.4]
* 删除了以下错误的表述：现在，Catch 语句可以将 Object 指定为 catch 块中的异常类型。[10.10.1.2]
* 更正了有关泛型类型和 Protected 成员之间的互交的表述。[4.6]
* 更正了 TypeArityList 产生式。[11.5.1]
* 声明了当装箱值类型不在任何类型中时不复制它，不仅仅是装箱值类型。[8.6]
* 更正了 ForEachStatement 产生式。[10.9.3]
* 更正了 ConversionOperatorModifier 产生式。[9.8.3]
* 更正了可能要包含泛型参数的 MemberAccessExpression 产生式。[11.6]
* 使用 # 而不是 @ 指定外部方法偏移量。[9.2.2]
* 移除了不必要的 RegularInitializer 产生式。[9.6.3, 9.6.3.1]
* 将运算符句法产生式折叠到一个单独的产生式中，因为实际语法比暗示更具灵活性。[9.8, 9.8.1, 9.8.2, 9.8.3]
* 声明了语言中的布尔表达式的使用。[10.8.1, 10.9.1, 10.9.2, 10.10.1.2]
* 指明 Protected 构造函数只可以从类型的构造函数或派生类型的构造函数调用。[4.6]
* 指明只可以由编译器生成代码创建默认实例。[11.6.2.2]
* 为简单起见，简化了 Imports 语法产生式。[6.3, 6.3.1, 6.3.2]
* 有关不在堆上的特殊类型的规则包括对 Object 和 System.ValueType 的实例方法的调用。[7.13]
* 澄清了以下问题：在进行名称隐藏时不会扩展 ParamArray 签名。[4.3.3]
* EnumDeclaration 生成一个 NonArrayTypeName，而不仅仅是一个 QualifiedName。[7.4]
* 声明了在 Visual Basic 的内部和外部映射重载运算符的方法。[9.8.1, 9.8.2, 9.8.4]
* 声明了绑定以区分大小写语言的方式工作的原理，和只由大小写区分的声明。[4.1]

### 其他

* 为了正确性和必要时更新来查看所有代码示例。

## 版本 8.0（第二版本）到版本 9.0

此更新涵盖在语言版本 8.0 和 9.0 之间所做的语言更改。

### 主要更改

* 添加了扩展方法。[9.2.7, 11.6, 11.8]
* 添加了分部方法。[9.2.1, 9.2.8, 11.8, 11.10.3, 12.1]
* 将成员初始化添加到对象创建表达式。[9.6, 9.6.3.2, 9.6.3.4, 11.10.1]
* 添加了局部类型推理。[6.2, 6.2.2, 6.2.5, 10.2, 10.2.1, 10.9.2, 10.9.3, 10.13]
* 添加了匿名类型构造。[11.10.1, 11.10.4]
* 添加了 Lambda 表达式和闭合。[7.13, 10.10.2.2, 11, 11.1, 11.1.1, 11.5.2, 11.8.1, 11.8.5, 11.10.3, 11.20, 11.20.1]
* 添加了表达式目录树。[11.1.1, 11.8.1, 11.8.5]
* 允许了要在签名上创建不完全匹配的委托的委托创建表达式。[11.10.3]
* 添加了可以为 Null 的值类型。[2.5, 7, 7.1.1, 7.13, 8.6.1, 8.8, 8.9, 8.11, 9.2.5, 9.6, 9.6.3.2, 9.8, 9.8.3, 10.2, 10.9.2, 11.5.3, 11.10.3, 11.12.3, 11.16, 11.17, 11.19, 11.20]
* 添加了查询表达式。[11, 11.21, 11.21.1, 11.21.2, 11.21.3, 11.21.4, 11.21.5, 11.21.6, 11.21.7, 11.21.8, 11.21.9, 11.21.10, 11.21.11, 11.21.12, 11.21.13, 11.21.14]
* 添加了条件运算符。[3.1, 11, 11.2, 11.22]
* 添加了 XML 命名空间导入。[6.3, 6.3.3, 11.5, 11.5.4]
* 添加了 XML 文本表达式。[2, 2.3, 11, 11.23, 11.23.1, 11.23.2, 11.23.3, 11.23.4, 11.23.5, 11.23.6, 11.23.7, 11.23.8]
* 添加了 XML 成员访问表达式。[11, 11.10.4, 11.24]

### 次要更改

* 添加了声明本地转换的方法的一节。[8.12, 11.11]
* 此语言现在考虑 InternalsVisibleToAttribute 特性。[4.5.1, 4.6, 7.5.2, 7.7.1, 7.12, 9.2.3]
* 现在，组类可以从多个类型（包括泛型类型）中收集。[11.6.2.2]
* Object 的成员可在接口的实例上得到，但由接口的成员隐式隐藏。[7.8.2]
* 现在，可基于重载决策中的返回类型比较委托类型的特异性。[11.8.1]
* 类型参数（无论是否为推断的）必须满足方法的约束才能适用于重载决策。[11.8.2]
* 扩展类型参数推断以对要解析的类型参数允许多个推断。[8.13, 11.8.5]
* 可以从 AddressOf 表达式的返回类型推断类型参数。[11.8.5]

### 澄清/勘误表

* 更改了要成为“目标”的方法/属性组的目标的术语，而不是更加混乱的“关联实例表达式”。[10.6.1, 11.1, 11.1.1, 11.3, 11.4.5, 11.6, 11.8, 11.9, 11.10.3]
* 声明了重载解决方案之前执行类型推断。[11.8.1]
* 声明了 Paramarray 如何参与重载解决方案。[11.8.1, 11.8.2]
* 添加了缺少的以下规则：在重载决策中，派生程度较大的方法优先于具有相同签名的派生程度较小的方法。[11.8.1]
* 固定了重载解决方案中选择“较小泛型”方法。[11.8.1]
* 澄清了以下问题：常规初始值设定项一次只可以初始化一个变量。[9.6.3.1]
* 固定了要在类型参数的存在中具有正确行为的简单名称和限定名称绑定。[4.7.1, 4.7.2, 11.4.4, 11.6]
* 删除了不能对声明了 MustOverride 的方法创建委托的错误规则。[11.10.3]
* 澄清了有关后期绑定和 AddressOf 的术语。[11.4.5]
* 固定了类型参数推断中数组的处理中的错误。[11.8.5]
* 声明了使用构造类型的类型参数推断。[11.8.5]
* 声明了引用转换周围的规则。[8.4]
* 声明/更正了对待类型参数约束的规则。[8.8, 8.9, 8.10, 8.12]
* 声明了带有对象初始值的变量名称不能有数组类型修饰符。[9.6.3.2]
* 声明了串联运算符的行为。[11.16]
* 声明/更正了布尔表达式的行为。[11.19]
* 更正了针对 Boolean 值的运算符的错误结果类型。[11.13.1, 11.13.2, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.18]
* 将 IsNot 添加到运算符优先级表中，它具有与 Is 相同的优先级。[11.12.1]
* 更正了 CatchStatement 产生式以指明 As 子句是可选的。[10.10.1.2]
* 更正了 MemberAccessExpression 产生式，因此该句点不是可选的。[11.6]
* 更正了条件编译表达式的语法。[3.1]
* 返工了类型约束一节，以满足类型约束的方式声明和允许数组协变。[4.9.2]
* 更正了以下问题：不能对带有 Option Strict 的 Object 使用 = 和 <>，但可使用 IsNot。[6.2.2]
* 指明可以在常量表达式中使用 Date。[11.2]

### 其他

* 删除了不需要的 DelegateCreationExpression 产生式。[11.10, 11.10.3]
* 将术语“预定义”更改为“内部”。[8.11, 8.11.1, 8.11.2, 11.12.3, 11.15]
* 删除了示例中对“foo”、“bar”和“baz”的所有引用。

## 版本 9.0 到版本 10.0

### 主要更改

* 添加了自动实现的属性。[9.7, 9.7.4]
* 允许在某些上下文中使用隐式行继续。[2.1.2, 4.9.1, 4.9.2, 5.2, 5.2.2, 6.3, 6.3.1, 6.3.3, 7.2, 7.4.1, 7.8.1, 7.9, 7.12, 9.1, 9.2.1, 9.2.2, 9.2.5, 9.2.6, 9.3, 9.4, 9.4.1, 9.5, 9.6, 9.6.3.3, 9.6.3.4, 9.7, 9.7.2, 9.8, 10.5.1, 10.5.2, 10.6.1, 10.6.3, 10.8.2, 10.9.2, 10.9.3, 10.12.1, 10.12.2, 11.4.2, 11.4.4, 11.5.1, 11.5.2, 11.5.3, 11.5.4, 11.6, 11.8, 11.9, 11.10.1, 11.11, 11.13.3, 11.13.4, 11.13.5, 11.13.6, 11.13.7, 11.13.8, 11.14, 11.15, 11.16, 11.17, 11.17.1, 11.18, 11.20, 11.21.1, 11.21.4, 11.21.5, 11.21.6, 11.21.7, 11.21.8, 11.21.9, 11.21.10, 11.21.11, 11.21.12, 11.21.13, 11.21.14, 11.22, 11.23.2]
* 添加了集合初始值设定项和数组文本。[9.6、9.6.3、9.6.3.3、9.6.3.4（已删除）、9.7、11.1、11.1.1、11.10、11.10.1、11.10.2、11.10.4、11.10.5]
* 添加了可包含语句和单行语句 lambda 的多行 lambda 表达式。[11.20]
* 添加了泛型变化。[4.9.1, 4.9.3, 7.11, 8.4.1, 8.8, 8.9]

### 次要更改

* 当绑定类型名称时，忽略不可访问的类型。[4.7.1, 4.7.2, 11.4.4]
* 不再要求结构至少具有一个实例变量。[7.6.1]
* 现在允许可选参数为结构、无约束的类型参数和引用类型。[9.2.5.3]
* 将对方法的返回值使用括号的特殊应用进一步限定为只有一个可访问的方法的情况。[11.8]
* 如果正在使用局部类型推理，并且标识符或者未定义，或者解析为一个类型，则带有该标识符的循环控制变量将定义一个新的局部标识符。[10.9.2, 10.9.3, 11.4.4]

### 澄清/勘误表

* 将多标记的标点符号和运算符移回到句法文法中，因为空格规则适用于它们。[2.5, 2.6, 3.1, 5.2.2, 9.8, 10.6.2, 10.8.2, 11.8, 11.14, 11.18]
* 更正了 Select 语句中的比较运算符的产生式。[10.8.2]
* 固定了针对属性的产生式中的错误数目。[9.7]
* 固定了针对日期文本的产生式中的错误。[2.4.6]
* 固定了 XML 字符引用产生式中的错误。[11.23.4]
* 更正了属性访问器可以与其包含属性具有相同访问级别的断言。[9.7]
* 修复了关键字表中 Wend 和 Variant 的拼写。[2.3]
* 声明了方法指针的分类和 lambda 方法发生在进行转换的任意位置。[11.1.1]
* 更正了有关不带上下文类型的 lambda 方法将成为 Func 的断言，这些方法实际上将成为可转换为类似 Func 的类型的匿名 lambda 类型。[11.1.1]
* 移除了错误的 XMLFragment 和 XmlNames 产生式。[11.23, 11.23.1]
* 更正了 XMLAttribute 产生式，使之从空格开始。[11.23.4]
* 声明了 lambda 表达式中的隐式声明局部变量在包含此表达式的方法中进行声明，而不是声明表达式自己。[10.2.1]
* 更正了类型名称产生式，因此正常处理泛型类型名称。[4.7, 6.4.1, 7, 7.12, 9.2.6, 11.5.1]
* 清除了转换说明和分类。[8, 8.1, 8.3, 8.4, 8.5, 8.8, 8.9, 8.12, 11.5.2]
* 更正了数组协变不再对继承和实现进行工作。[8.5, 8.8, 8.9]
* 修复了 XMLEncodingNameCharacter 产生式以包括短划线字符。[11.23.3]
* 声明了字符串限定可能出现（但可以不出现）。[2.4.4]
* 添加了有关匿名委托转换的章节。[8.4.2, 8.8, 8.9]
* 定义了短语“内部转换”。[8.11]
* 删除了以下错误表述：由条件编译排除的行一定是在词法上有效的。[3.1]
* 声明了重写方法的返回类型和重写方法必须符合。[4.5.1]
* 删除了以下的错误断言：Protected 构造函数只可以从类型的构造函数中或派生类型的构造函数中进行调用。[4.6]
* 澄清了结构可以满足 New 约束。[4.9.1]
* 澄清了其元素类型具有扩大标识或引用转换的类型的数组实现 IList(Of T)。[7.9]
* 声明了可以将装箱枚举的类型拆卸到它们的基础类型。[8.6]
* 澄清了以下问题：只有内部转换才可用于可选参数值。[9.2.5.3]
* 澄清了以下问题：如果名称不解析为任何其他内容，则隐式进行局部声明。[10.2.1]
* 更正了参数列表的文法。[5.2.2, 11.8]
* 澄清了实参列表匹配中 IDispatchConstantAttribute 和 IUnknownConstantAttribute 的用法。[11.8.2.]
* 更正了当在委托创建表达式中匹配签名时参数类型的转换方向。[11.10.3]
* 试图声明语言中 null 值的位置。[7.1, 7.3, 8.4, 8.6, 10.10.1.3, 10.12.1, 11.1, 11.1.1, 11.4.1, 11.8.2, 11.14]
* TryCast 中的表达式的类型不必是值类型。[11.11]
* Else If 允许有空格。[2.5, 3.2, 10.8.1]
* 预处理器允许有空白。[3.1]
* 将在运行时计算常量字符串表达式。[2.4.4]
* 澄清了 Region 指令的嵌套。[3.3]
* 澄清了以下问题：实现方法必须与返回类型匹配。[4.4.1]
* 澄清了以下问题：枚举类型不能声明类型参数。[4.9]
* 澄清了不允许类型约束的哪些不一致组合。[4.9.2]
* 特性类不需要具有显式 AttributeUsage。[5.1]
* 位置特性参数可以是 Optional 或 ParamArray。[5.2]
* 可将 IdentifierModifiers ? 和 () 进行组合。[7, 9.2.5, 9.6, 10.9.2]
* 委托声明可以具有特性和修饰符。[7.10]
* 嵌套分步类型声明可在非嵌套容器中出现。[7.12]
* 澄清了内部布尔转换。[8.2]
* Handles 子句允许零参数委托松弛，并在指定事件时允许 MyClass 和 MyBase。[9.2.6, 11.4.3]
* 语句必须以关键字或标识符开头。[10]
* 始终将推理用于缺少显式类型的局部常量声明。[10.2]
* 澄清了布尔表达式的处理。[10.8.1, 10.9.1, 10.10.1.2, 11.19]
* 将多个 Next 语句组合为一个。[10.9.2, 10.9.3]
* 集合模式与扩展方法相匹配。[10.9.3]
* 表达式“Nothing”具有“默认值”分类，可以重新分类。[11.1.1, 11.4.1]
* 在哪些类型的 lambda 可转换为表达式树方面存在限制。[11.1.1]
* Object 对于常量表达式是有效类型。常量中允许强制转换为 null 字符串和从 null 字符串进行强制转换。澄清了以下问题：针对常量进行的运算必须生成有效的常量类型。[11.2]
* 澄清了调用参数的计算顺序。[11.3, 11.8]
* TypeArityList 需要括号和“Of”。[11.5.1]
* 成员访问表达式可处理可以为 null 的类型。[11.6]
* E 是枚举时允许使用 E.New。[11.6]
* 在使用 Option Strict On 时不允许对 Object 使用相等和不相等运算符。[11.12.2]
* 澄清了 AndAlso 和 OrElse 对可以为 null 的布尔值的行为。[11.17.1]
* Join 和 GroupJoin 查询运算符允许使用嵌套 Join 和 GroupJoin 查询运算符。[11.21, 11.21.5, 11.21.14]
* 只有 Let 查询运算符才可以指定其表达式范围变量的类型。[11.21]
* “Group By”查询运算符的“Group”子句是可选的。[11.21.12]

## 版本 10.0 到版本 11.0

### 主要更改

* 添加了对异步方法和迭代器方法以及 Await 和 Yield 关键字的支持 [6.1，9.2.1，9.7.5，10，10.1，10.1.1-10.1.3，10.10.1.1，10.10.1.2，10.10.2，10.11，10.14，10.15，11.1，11.1.1，11.8.5，11.12.1，11.20，11.25]
* 添加了可选形参上对 CallerLineNumber、CallerFilePath 和 CallerMemberName 特性的支持；澄清了 Microsoft 对某些可选形参的另一种特殊处理 [11.8.2]
* 现在可以显式声明命名空间以包括在全局命名空间中 [6.4.1]

### 次要更改

* 向重载解析添加了新的优选规则：如果一个重载在调用时不需要替入任何可选形参默认值，而另一个重载需要，则优先采用第一个重载 [11.8.1]
* For Each 现在可在每个迭代循环上创建迭代变量的全新副本 [10.9.3]
* 编译器现在允许声明只有可选形参不同的多个方法（只要它们的签名不同即可）[9.2]
* 向重载解析添加了新的优选规则：在两个适用的泛型重载中，优先采用泛型类型形参“更深”的重载 [11.8.1.3]
* 添加了对 IReadOnlyList 和 IReadOnlyCollection 的支持 [7.9，8.8，8.9，11.1.1]
* 事件在 winmd 文件中具有略微不同的代码生成 [9.4.1]
* Object 和其他类型之间的二元运算符解析（如果没有内部转换或用户定义转换）现在将是后期绑定而不是错误 [11.12.3]

### 澄清/勘误表

* 更正了重载解析规则：将“适用性”重命名为“特异性”；如果重载候选为精确匹配，则可选择该候选而不必考虑其他重载解析规则；更正了内部数值类型之间的特异性优先级规则；澄清了 Option Strict 在重载解析中的使用；更正了使用后期绑定的规则；澄清了收缩实例方法不会排除扩展方法；添加了对委托松弛级别的大量讨论，这些委托松弛级别实际上在重载解析开始时执行，而不是作为最终的优选规则；澄清了对两个扩展方法候选之间的类型形参优选规则的描述；更正了有关对 AddressOf 表达式与委托类型之间的 Sub/Function 进行优选的描述；澄清了适用性也可判断泛型约束 [11.8.1，11.8.2]
* 更正了有关在重载解析中各数值类型之间特异性比较的规则，并描述了有关形参类型特异性的其他最终优选规则 [11.8.1.1]
* 澄清了 FieldInitializer 中的“Key”关键字只允许用在匿名对象创建表达式中 [11.10.1]
* 修复了匿名对象表达式的代码示例 [11.10.4]
* 澄清了带有实参列表的调用语句始终会作为该调用的实参处理，并且调用语句必须以一个标识符开头 [10.7]
* 澄清了集合模式可以绑定到共享方法/属性，并且集合初始值设定项可以绑定到共享 Add 方法 [10.9.3，11.10.1]
* 澄清了“For *标识符* =”和“For Each *标识符* In”语句中的名称查找规则 [10.9.2，10.9.3]
* 澄清了嵌套命名空间基于“逻辑”而不是基于“词法”[4.7]，并澄清了限定和非限定类型/命名空间名称查找规则 [4.7.1，4.7.2]
* 更正了扩展方法查找规则：在封闭命名空间中进行搜索前，编译器在导入的命名空间中搜索扩展方法 [11.6.3]
* 澄清了可将泛型类型约束到对它们进行定义的类型内的嵌套类型 [7.5.2，7.6.1，7.8.2]
* 澄清了 Imports 别名规则：它们引用类型或命名空间名称本身，而不是使用规则来进行限定名称解析；它们不能是开放式泛型类型 [6.3.1]
* 澄清了重载解析规则：如果某重载候选是精确匹配，则选择该候选，而不必考虑其他重载解析规则 [11.8.1]
* 澄清了基于基础枚举类型向 IList、IReadOnlyList、ICollection、IReadOnlyCollection、IEnumerable 进行数组转换的规则 [7.9，8.5，8.8，8.9]
* 描述了由协变和逆变所引入的多义性 [4.9.3]
* 澄清了将 lambda 转换为非委托类型（包括 Object、Delegate 和 MulticastDelegate）的规则 [8.4.2]
* 澄清了扩大转换和收缩转换的列表 [8.8，8.9]
* 澄清了方法可以同时为 MustOverride 和 Overrides [9.2.3]
* 更正了 GetTypeExpression 的语法 [11.5.1]
* 更正了 MemberAccessBase 的语法：它不包括 MyBase 或 Myclass [11.6]
* 更正了重载解析中内部数值类型之间的适用性优先级规则 [11.8.1]
* 允许在 [xml.@attr](mailto:xml.@attr) 中的句点后使用隐式行继续符 [11.24]
* 澄清了不能对事件调用 add\_X 和 remove\_X 方法 [9.4]
* 更正了有关重载解析中数值类型适用性比较的规则 [11.8.1.1]
* 更正了匿名类型示例 [11.10.4]
* 澄清了与 IsTrue 和 IsFalse 有关的短路逻辑运算符的行为 [11.17.1]
* 澄清了在循环内创建局部变量的精确时间 [10.1.5，10.9，10.9.2，10.9.3]
* 澄清了“方法签名”的定义 -- 不包括方法名称 [4.1.1]
* 澄清了通过别名或复制再反向复制来使用 ByRef 实参的规则 [9.2.5.2]
* 常量类型成员不使用类型推断 [9.5]
* 指明了某些块中不允许包含 GoTo 的情况 [10.11]
* 澄清了运算符解析可以后期绑定的规则 [11.12.2，11.12.3]
* 澄清了可以为 null 的可选形参的类型规则 [9.2.5.3]
* 澄清了只有 Me（而不是 MyClass 或 MyBase）才是有效实例表达式；MyClass 和 MyBase 只能用在成员访问表达式和 Handles 子句中 [9.2.6，11.4.3，11.6]
* 澄清了扩大转换/收缩转换判断适用于表达式重新分类，具体而言，具有较窄形参类型的 lambda 重新分类即为收缩转换，具有较窄数组元素类型的数组文本重新分类也是如此 [11.1.1]