C#

规范 2.0 版

2005 年 7 月

目录

19. C# 2.0 简介

C# 2.0 引入了几项语言扩展，其中包括泛型 (Generic)、匿名方法 (Anonymous Method)、迭代器 (Iterator)、分部类型 (Partial Type) 和可空类型 (Nullable Type)。

- 泛型可以让类、结构、接口、委托和方法按它们存储和操作的数据的类型进行参数化。泛型很有用，因为它们能提供更强的编译时类型检查，减少数据类型之间的显式转换，以及装箱操作和运行时的类型检查。
- 在需要委托值的地方，匿名方法允许以“内联”方式编写代码块。匿名方法类似于 Lisp 编程语言中的 lambda 函数。C# 2.0 支持创建“closure”，其中的匿名方法可以访问外层局部变量和参数。
- 迭代器是执行递增计算并产生一系列值的方法。迭代器使类型可以简便地指定 `foreach` 语句循环访问其元素的方式。
- 分部类型允许将类、结构和接口划分为多个部分，存储在不同的源文件中，以便于开发和维护。此外，分部类型允许将计算机生成的类型部分和用户编写的类型部分互相分开，以便更容易地扩充工具生成的代码。
- 可空类型表示可能未知的值。可空类型支持其基础类型的所有值以及一个附加的空状态。任何值类型均可作为可空类型的基础类型。可空类型支持与其基础类型相同的转换和运算符，另外还提供类似于 SQL 的空值传播。

本章将简要介绍这些新功能。之后，接下来的 5 章详细介绍这些功能的完整技术规范。最后一章介绍 C# 2.0 中引入的一些次要的扩展。

C# 2.0 中的语言扩展在设计上充分考虑并确保与现有代码的最大兼容性。例如，尽管 C# 2.0 在某些上下文中为单词 `where`、`yield` 和 `partial` 提供了特殊含义，但是这些单词仍可用作标识符。实际上，C# 2.0 没有添加新的关键字，因为这样的关键字可能与现有代码中的标识符冲突。

有关 C# 语言的最新信息以及如何为本文档提供反馈意见的说明，请访问“C# 语言主页”(http://msdn.microsoft.com/vcsharp/language)。

19.1 泛型

泛型可以让类、结构、接口、委托和方法按它们存储和操作的数据的类型进行参数化。使用过 Eiffel 或 Ada 泛型的用户或 C++ 模板的用户很快就能熟悉 C# 泛型，而且这些用户会发现 C# 泛型较之过去这些语言更加简便易用。

19.1.1 为什么要使用泛型？

如果没有泛型，通用数据结构可通过使用类型 `object` 实现任何数据类型的存储。例如，下面是一个简单的 `Stack` 类，它将数据存储在一个 `object` 数组中，它的两个方法 `Push` 和 `Pop` 分别使用 `object` 接受和返回数据：

```
public class Stack
{
   object[] items;
   int count;

   public void Push(object item) {...}

   public object Pop() {...}
}
```

虽然使用类型 object 使得 Stack 类非常灵活，但这种方式仍存在某些缺陷。例如，我们可以将任何类型的值（例如一个 Customer 实例）推入堆栈。但是，当从堆栈中检索某个值时，必须将 Pop 方法的结果显式强制转换回相应的类型，这样的代码编写起来颇为繁琐，而且在运行时执行的类型检查会造成额外的开销从而影响性能：

```
Stack stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();
```

再比如，当我们将一个值类型（例如 int）的值传递给 Push 方法时，则该值将自动被装箱。当以后检索该 int 时，必须使用显式类型强制转换将其取消装箱：

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

这样的装箱和取消装箱操作由于涉及动态内存分配和运行时类型检查而额外增加了性能开销。

上述 Stack 类还有一个潜在的问题就是我们无法对放到堆栈上的数据的种类施加限制。实际上，可能会发生这种情况：将一个 Customer 实例推入堆栈，而在检索到该实例之后却意外地将它强制转换为错误的类型：

```
Stack stack = new Stack();
stack.Push(new Customer());
string s = (string)stack.Pop();
```

虽然上面的代码错误使用了 Stack 类，但是从技术角度讲该代码可以视作是正确的，编译器不会报告编译时错误。这个问题在该代码被执行之前不会暴露出来，但在执行该代码时会引发 InvalidCastException。

显然，如果能够指定元素类型，Stack 类将能够从中受益。有了泛型，我们便可以做到这一点。

19.1.2 创建和使用泛型

泛型提供了一种新的创建类型的机制，使用泛型创建的类型将带有类型形参 ***(type parameter)***。下面的示例声明一个带有类型形参 T 的泛型 Stack 类。类型形参在 < 和 > 分隔符中指定并放置在类名后。Stack<T> 的实例的类型由创建时所指定的类型确定，该实例将存储该类型的数据而不进行数据类型转换。这有别于同 object 之间的相互转换。类型形参 T 只起占位符的作用，直到在使用时为其指定了实际类型。注意，这里的 T 用作内部项数组的元素类型、传递给 Push 方法的参数类型和 Pop 方法的返回类型：

```
public class Stack<T>
{
   T[] items;
   int count;

   public void Push(T item) {...}

   public T Pop() {...}
}
```

在使用泛型类 Stack<T> 时，将指定用于替换 T 的实际类型。在下面的示例中，指定了 int 作为 T 的类型实参 ***(type argument)***：

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

Stack<int> 类型称为构造类型 ***(constructed type)***。在 Stack<int> 类型中，出现的每个 T 都被替换为类型实参 int。在创建 Stack<int> 的实例后，items 数组的本机存储是 int[] 而不是 object[]。无疑，这比非泛型的 Stack 提供了更高的存储效率。同样，Stack<int> 的 Push 和 Pop 方法所操作的也是 int 类型的值。如果将其他类型的值推入堆栈则产生编译时错误。而且在检索值时也不再需要将它们显式强制转换为原始类型。

泛型提供了强类型机制，这意味着如果将一个 int 值推入 Customer 对象的堆栈将导致错误。正如 Stack<int> 仅限于操作 int 值一样，Stack<Customer> 仅限于操作 Customer 对象，编译器将对下面示例中的最后两行报告错误：

```
Stack<Customer> stack = new Stack<Customer>();
stack.Push(new Customer());
Customer c = stack.Pop();
stack.Push(3);                  // Type mismatch error
int x = stack.Pop();            // Type mismatch error
```

泛型类型声明可以含有任意数目的类型形参。上面的 Stack<T> 示例只有一个类型形参，而一个泛型 Dictionary 类可能具有两个类型形参，一个用于键的类型，一个用于值的类型：

```
public class Dictionary<K,V>
{
   public void Add(K key, V value) {...}

   public V this[K key] {...}
}
```

在使用上述 Dictionary<K,V> 时，必须提供两个类型实参：

```
Dictionary<string,Customer> dict = new Dictionary<string,Customer>();
dict.Add("Peter", new Customer());
Customer c = dict["Peter"];
```

19.1.3 泛型类型实例化

与非泛型类型类似，泛型类型的编译表示形式也是中间语言 (IL) 指令和元数据。当然，泛型类型的表示形式还要对类型形参的存在和使用进行编码。

在应用程序第一次创建构造泛型类型（例如 Stack<int>）的实例时，.NET 公共语言运行库的实时 (JIT) 编译器将泛型 IL 和元数据转换为本机代码，并在该过程中将类型形参替换为实际类型。然后，对该构造泛型类型的后续引用将使用相同的本机代码。从泛型类型创建特定构造类型的过程称为泛型类型实例化 ***(generic type instantiation)***。

对于值类型，.NET 公共语言运行库为每次泛型类型实例化单独创建专用的本机代码副本。而对于所有的引用类型，则共享该本机代码的单个副本（因为在本机代码级别，引用不过是具有相同表示形式的指针）。

19.1.4 约束

通常，泛型类的作用并不仅仅是根据类型形参存储数据。泛型类常常需要调用对象上的方法，对象的类型由类型形参给出。例如，Dictionary<K,V> 类中的 Add 方法可能需要使用 CompareTo 方法对键进行比较：

```
public class Dictionary<K,V>
{
   public void Add(K key, V value)
   {
      ...
      if (key.CompareTo(x) < 0) {...}     // Error, no CompareTo method
      ...
   }
}
```

由于为 K 指定的类型实参可以是任何类型，对于 key 参数，能够假设存在的成员仅限于类型 object 声明的成员，例如 Equals、GetHashCode 和 ToString；因此上面的示例将发生编译时错误。当然，我们可以将 key 参数强制转换为含有 CompareTo 方法的类型。例如，可以将 key 参数强制转换为 IComparable：

```
public class Dictionary<K,V>
{
   public void Add(K key, V value)
   {
      ...
      if (((IComparable)key).CompareTo(x) < 0) {...}
      ...
   }
}
```

虽然这种解决方案可行，但它需要在运行时动态检查类型，因此会增加开销。另外它还将错误报告推迟到运行时，当键未实现 IComparable 时引发 InvalidCastException。

为了提供更强的编译时类型检查并减少类型强制转换，C# 允许为每个类型形参提供一个可选的约束 ***(constraint)*** 列表。类型形参约束指定了一个要求，类型必须满足该要求才能用作该类型形参的实参。约束使用单词 where 进行声明，后跟一个类型形参和一个冒号，再跟着一个逗号分隔的列表，列表项可以是类类型、接口类型甚或类型形参（还可以是特殊引用类型、值类型和构造函数约束）。

为了让 Dictionary<K,V> 类能够确保键总是实现 IComparable，可在类声明中为类型形参 K 指定一个约束，如下所示：

```
public class Dictionary<K,V> where K: IComparable
{
   public void Add(K key, V value)
   {
      ...
      if (key.CompareTo(x) < 0) {...}
      ...
   }
}
```

有了这个声明后，编译器将确保为 K 提供的任何类型实参均为一个实现了 IComparable 的类型。不仅如此，此约束声明还避免了在调用 CompareTo 方法之前显式将键参数强制转换为 IComparable；满足此类型形参约束的类型的所有成员都可作为该类型形参类型的值直接使用。

对于一个给定的类型形参，作为约束的接口和类型形参的数目不受限制，但只能有一个类。每个受约束的类型形参具有单独的 where 子句。在下面的示例中，类型形参 K 具有两个接口约束，而类型形参 E 具有一个类类型约束和一个构造函数约束：

```
public class EntityTable<K,E>
   where K: IComparable<K>, IPersistable
   where E: Entity, new()
{
   public void Add(K key, E entity)
   {
      ...
      if (key.CompareTo(x) < 0) {...}
      ...
   }
}
```

在上面示例中，构造函数约束 new() 确保用作 E 的类型实参的类型具有无参数的公共构造函数，这样，泛型类便可以使用 new E() 创建该类型的实例。

需指出的是，类型形参约束应该谨慎使用。虽然它们提供更强的编译时类型检查，并在某些情况下改进了性能，但是它们也使泛型类型的使用受到限制。例如，泛型类 List<T> 可能约束 T 必须实现 IComparable，以便 List 的 Sort 方法能够对项进行比较。但是，即使在某些情形下 Sort 方法根本没有被调用，也会由于上述约束的存在而导致未实现 IComparable 的类型无法使用 List<T>。

19.1.5 泛型方法

在有些情况下，并不是整个类都需要某个类型形参，而是仅在某个特定方法中需要。通常，在创建采用泛型类型作为形参的方法时会遇到这种情况。例如，在使用上述 Stack<T> 类时，常见的使用模式可能是使用一行代码将多个值推入堆栈，我们可以很方便地编写一个方法在单个调用中完成此任务。对于特定的构造类型，例如 Stack<int>，该方法如下所示：

```
void PushMultiple(Stack<int> stack, params int[] values) {
   foreach (int value in values) stack.Push(value);
}
```

然后，可以使用此方法将多个 int 值推入 Stack<int>：

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

但是，上面的方法仅适用于特定的构造类型 Stack<int>。为了将其用于任何 Stack<T>，必须将该方法编写为一个泛型方法 ***(generic method)***。泛型方法在方法名后面的 < 和 > 分隔符中指定了一个或多个类型形参。这些类型形参可以在形参列表、返回类型和方法体内使用。泛型 PushMultiple 方法如下所示：

```
void PushMultiple<T>(Stack<T> stack, params T[] values) {
   foreach (T value in values) stack.Push(value);
}
```

使用此泛型方法，可以将多个项推入任何 Stack<T>。在调用泛型方法时，类型实参在方法调用中的尖括号中给出。例如：

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

此泛型 PushMultiple 方法较之前一个版本更具可重用性，因为它适用于任何 Stack<T>。但由于必须以类型实参的方式为该方法提供所需的 T，使其调用起来似乎没有前一版本方便。但是在许多情况下，编译器能够使用称为“类型推断”***(type inferencing)*** 的过程，从传递给方法的其他实参推断出正确的类型实参。在上面的示例中，由于第一个标准实参是 Stack<int> 类型，后续的几个实参为 int 类型，编译器能够推断出类型形参一定是 int。这样，可以在不指定类型形参的情况下调用泛型 PushMultiple 方法：

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

19.2 匿名方法

事件处理程序和其他回调方法通常仅通过委托机制进行调用，而从不直接进行调用。所以，我们目前还是只能将事件处理程序代码及回调代码置于特定的方法中，并显式为方法创建委托。而匿名方法***(anonymous method)*** 则不同，它允许与委托关联的代码以“内联”方式写入使用委托的位置，从而方便地将代码直接“绑定”到委托实例。除了这种便利之外，匿名方法还能够对包含它的函数成员的局部状态进行共享访问。而要使用具名方法实现同样的状态共享，需要将局部变量“提升”到手动编写的辅助类实例的字段中。

下面的示例显示一个包含列表框、文本框和按钮的简单输入窗体。单击该按钮时，一个包含文本框中所示文本的项将被添加到列表框中。

```
class InputForm: Form
{
   ListBox listBox;
   TextBox textBox;
   Button addButton;

   public MyForm() {
      listBox = new ListBox(...);
      textBox = new TextBox(...);
      addButton = new Button(...);

      addButton.Click += new EventHandler(AddClick);
   }

   void AddClick(object sender, EventArgs e) {
      listBox.Items.Add(textBox.Text);
   }
}
```

虽然在响应该按钮的 Click 事件时只执行了一条语句，但是必须将该语句提取到一个具有完整参数列表的单独方法中，并且必须手动创建一个引用该方法的 EventHandler 委托。使用匿名方法，该事件处理代码明显变得更简洁了：

```
class InputForm: Form
{
   ListBox listBox;
   TextBox textBox;
   Button addButton;

   public MyForm() {
      listBox = new ListBox(...);
      textBox = new TextBox(...);
      addButton = new Button(...);

      addButton.Click += delegate {
         listBox.Items.Add(textBox.Text);
      };
   }
}
```

匿名方法由关键字 delegate、可选的参数列表和包含在 { 和 } 分隔符中的语句列表组成。上述示例中的匿名方法没有使用委托提供的参数，因此可以省略参数列表。若要获得对参数的访问，该匿名方法可以包括参数列表：

```
addButton.Click += delegate(object sender, EventArgs e) {
   MessageBox.Show(((Button)sender).Text);
};
```

在上面的示例中，发生了从该匿名方法到 EventHandler 委托类型（Click 事件的类型）的隐式转换。能够进行该隐式转换是因为委托类型的参数列表和返回类型与匿名方法兼容。具体的兼容规则如下：

- 如果下列条件之一成立，则委托的参数列表与匿名方法兼容：
  - 匿名方法没有参数列表并且委托没有 out 参数。
  - 匿名方法包含的参数列表与委托的参数列表在数目、类型和修饰符方面都精确匹配。
- 如果下列条件之一成立，则委托的返回类型与匿名方法兼容：
  - 委托的返回类型为 void，并且匿名方法没有 return 语句或只有无表达式的 return 语句。
  - 委托的返回类型不为 void，并且与匿名方法中的所有 return 语句关联的表达式都可隐式转换为委托的返回类型。

委托的参数列表和返回类型都必须与匿名方法兼容才能进行从匿名方法到委托类型的隐式转换。

下面的示例使用匿名方法编写“内联”函数。匿名方法作为 Function 委托类型的参数来传递。

```
using System;

delegate double Function(double x);

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

   static double[] MultiplyAllBy(double[] a, double factor) {
      return Apply(a, delegate(double x) { return x * factor; });
   }

   static void Main() {
      double[] a = {0.0, 0.5, 1.0};

      double[] squares = Apply(a, delegate(double x) { return x * x; });

      double[] doubles = MultiplyAllBy(a, 2.0);
   }
}
```

Apply 方法将给定的 Function 应用于 double[] 的各元素，并返回含有结果的 double[]。在 Main 方法中，传递给 Apply 的第二个参数是与 Function 委托类型兼容的匿名方法。此匿名方法完成一个简单的操作，即返回其实参的平方，因此该 Apply 调用的结果是包含 a 中各个值的平方的 double[]。

MultiplyAllBy 方法返回一个 double[]，其中的值分别为实参数组 a 中的每个值乘以给定的 factor 后所得的结果。为了获得所需的结果，MultiplyAllBy 调用了 Apply 方法，并传递一个用来将实参 x 乘以 factor 的匿名方法。

其作用域包含某匿名方法的局部变量和参数称为该匿名方法的外层变量 ***(outer variable)***。在 MultiplyAllBy 方法中，a 和 factor 是传递给 Apply 的匿名方法的外层变量，并且由于该匿名方法引用 factor，我们称 factor 被该匿名方法捕获 ***(captured)***。通常，局部变量的生存期仅限于该变量所关联的代码块或语句的执行期间。但是，被捕获的外层变量的生存期将至少延长至引用匿名方法的委托可以被垃圾回收为止。

19.2.1 方法组转换

正如前一节所述，匿名方法可隐式转换为与之兼容的委托类型。C# 2.0 允许对方法组进行这种相同类型的转换，这样，几乎在所有情况下都可以省略显式的委托实例化。例如，语句

```
addButton.Click += new EventHandler(AddClick);
Apply(a, new Function(Math.Sin));
```

可改写为

```
addButton.Click += AddClick;
Apply(a, Math.Sin);
```

当使用较短的形式时，编译器自动推断要实例化的委托类型，而效果与使用较长的形式是一样的。

19.3 迭代器

C# foreach 语句用于循环访问可枚举 ***(enumerable)*** 集合的元素。为了成为可枚举的类型，集合必须具有返回枚举器 ***(enumerator)*** 的无参数的 GetEnumerator 方法。一般而言，枚举器不易实现，但是通过使用迭代器可以显著简化该任务。

迭代器 ***(iterator)*** 是一个产生 ***(yield)*** 有序值序列的语句块。迭代器与普通语句块的区别在于迭代器存在一个或多个 yield 语句：

- yield return 语句产生迭代的下一个值。
- yield break 语句指示迭代完成。

只要函数成员的返回类型是枚举器接口 ***(enumerator interface)*** 之一或可枚举接口 ***(enumerable interface)*** 之一，迭代器就可用作该函数成员的函数体：

- 枚举器接口为 System.Collections.IEnumerator 和从 System.Collections.Generic.IEnumerator<T> 构造的类型。
- 可枚举接口为 System.Collections.IEnumerable 和从 System.Collections.Generic.IEnumerable<T> 构造的类型。

迭代器并非一种成员，而是一种实现函数成员的手段，了解这一点很重要。通过迭代器实现的成员能够被其他可能使用也可能不使用迭代器实现的成员重写或重载。

下面的 Stack<T> 类使用一个迭代器实现其 GetEnumerator 方法。该迭代器以自顶向下的顺序枚举堆栈的元素。

```
using System.Collections.Generic;
public class Stack<T>: IEnumerable<T>
{
   T[] items;
   int count;
   public void Push(T data) {...}
```

```
    public T Pop() {...}

    public IEnumerator<T> GetEnumerator() {
       for (int i = count - 1; i >= 0; --i) {
          yield return items[i];
       }
    }
}
```

GetEnumerator 方法的存在使得 Stack<T> 成为可枚举类型，这样，我们就可以在 foreach 语句中使用 Stack<T> 的实例。下面的示例将值 0 至 9 推入一个整数堆栈，然后使用 foreach 循环按自顶向下的顺序显示这些值。

```
using System;

class Test
{
   static void Main() {
      Stack<int> stack = new Stack<int>();
      for (int i = 0; i < 10; i++) stack.Push(i);
      foreach (int i in stack) Console.Write("{0} ", i);
      Console.WriteLine();
   }
}
```

该示例的输出为：

```
9 8 7 6 5 4 3 2 1 0
```

foreach 语句隐式调用集合的无参数 GetEnumerator 方法获得枚举器。一个集合只能定义一个这样的无参数 GetEnumerator 方法，但是可以有多种枚举方式以及多种通过参数控制枚举的方式。在这种情况下，集合可以使用迭代器实现多个返回可枚举接口类型之一的属性或方法。例如，Stack<T> 可引入 IEnumerable<T> 类型的两个新属性 TopToBottom 和 BottomToTop：

```
using System.Collections.Generic;

public class Stack<T>: IEnumerable<T>
{
   T[] items;
   int count;

   public void Push(T data) {...}

   public T Pop() {...}

   public IEnumerator<T> GetEnumerator() {
      for (int i = count - 1; i >= 0; --i) {
         yield return items[i];
      }
   }

   public IEnumerable<T> TopToBottom {
      get {
         return this;
      }
   }

   public IEnumerable<T> BottomToTop {
      get {
         for (int i = 0; i < count; i++) {
            yield return items[i];
         }
      }
   }
}
```

TopToBottom 属性的 get 访问器直接返回 this，因为该堆栈本身是可枚举的。BottomToTop 属性返回一个使用 C# 迭代器实现的可枚举接口类型。下面的示例显示如何使用这两个属性以任一顺序枚举堆栈元素：

```
using System;

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

      foreach (int i in stack.TopToBottom) Console.Write("{0} ", i);
      Console.WriteLine();

      foreach (int i in stack.BottomToTop) Console.Write("{0} ", i);
      Console.WriteLine();
   }
}
```

当然，这些属性也可以在 foreach 语句之外使用。下面的示例将调用这些属性的结果传递给一个单独的 Print 方法。该示例还演示了一个用作 FromToBy 方法（该方法带有参数）的方法体的迭代器：

```
using System;
using System.Collections.Generic;

class Test
{
   static void Print(IEnumerable<int> collection) {
      foreach (int i in collection) Console.Write("{0} ", i);
      Console.WriteLine();
   }

   static IEnumerable<int> FromToBy(int from, int to, int by) {
      for (int i = from; i <= to; i += by) {
         yield return i;
      }
   }

   static void Main() {
      Stack<int> stack = new Stack<int>();
      for (int i = 0; i < 10; i++) stack.Push(i);
      Print(stack.TopToBottom);
      Print(stack.BottomToTop);
      Print(FromToBy(10, 20, 2));
   }
}
```

该示例的输出为：

```
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
10 12 14 16 18 20
```

泛型和非泛型可枚举接口仅含有一个成员，即一个不带参数并返回一个枚举器接口类型的 GetEnumerator 方法。可枚举接口起到“枚举器工厂”的作用。如果正确实现了可枚举接口，每次调用它们的 GetEnumerator 方法时，都会生成一个独立的枚举器。假设在两次调用 GetEnumerator 期间可枚举接口类型的内部状态没有改变，则所返回的两个枚举器应以相同的顺序产生相同的枚举值集合。即使枚举器的生存期重叠，这个结论也应该有效，如下面的代码示例所示：

```
using System;
using System.Collections.Generic;

class Test
{
   static IEnumerable<int> FromTo(int from, int to) {
      while (from <= to) yield return from++;
   }

   static void Main() {
      IEnumerable<int> e = FromTo(1, 10);
      foreach (int x in e) {
         foreach (int y in e) {
            Console.Write("{0,3} ", x * y);
         }
         Console.WriteLine();
      }
   }
}
```

上面的代码可输出整数 1 至 10 的简单乘法表。注意，FromTo 方法仅被调用一次来生成可枚举的 e。但是，e.GetEnumerator() 被调用多次（由 foreach 语句调用）以生成多个等效的枚举器。这些枚举器全都封装了在 FromTo 的声明中指定的迭代器代码。注意该迭代器代码将修改 from 参数。然而，这些枚举器独立工作，因为每个枚举器都会获得它自己的 from 和 to 参数的副本。枚举器之间的过渡状态的共享是若干常见细微缺陷之一，在实现可枚举接口和枚举器时应该避免。C# 迭代器旨在帮助避免这些问题，并以简单、直观的方法实现可靠的可枚举接口和枚举器。

19.4 分部类型

虽然在单个文件中维护某个类型的所有源代码是个很好的编程习惯，但是有时一个类型会变得非常大，在这种情况下，这种做法反而成为了一种不切实际的限制。此外，程序员经常使用源代码生成器产生应用程序的初始结构，然后修改结果代码。遗憾的是，当将来某个时候再次发出源代码时，现有修改将被改写。

分部类型 ***(Partial type)*** 允许将类、结构和接口划分为多个部分，存储在不同的源文件中，以便于开发和维护。此外，分部类型允许将计算机生成的类型部分和用户编写的类型部分互相分开，以便更容易地扩充工具生成的代码。

当通过多个部分来定义类型时，将使用新增的类型修饰符 partial。下面是一个分为两部分来实现的分部类示例。这里，第一部分是通过数据库映射工具由计算机生成的，而第二部分是手动编写的，所以，可将这两个部分分别存储在不同的源文件中：

```
public partial class Customer
{
   private int id;
   private string name;
   private string address;
   private List<Order> orders;

   public Customer() {
      ...
   }
}

public partial class Customer
{
   public void SubmitOrder(Order order) {
      orders.Add(order);
   }
```

```
    public bool HasOutstandingOrders() {
       return orders.Count > 0;
    }
}
```

当将上述两个部分一起编译时，结果代码与将该类编写为单个单元时相同：

```
public class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Order> orders;

    public Customer() {
       ...
    }

    public void SubmitOrder(Order order) {
       orders.Add(order);
    }

    public bool HasOutstandingOrders() {
       return orders.Count > 0;
    }
}
```

分部类型的所有部分必须一起编译，以使这些部分可在编译时被合并。特别指出的是，分部类型不允许对已经编译的类型进行扩展。

19.5 可空类型

在与数据库交互时，支持所有类型（包括值类型）的可空值是很重要的，而一直以来通用编程语言很少或没有提供这方面的支持。在没有直接语言支持的情况下，虽然也存在许多用于处理空值和值类型的方法，但都存在缺点。例如，一种方法是使用“特殊”值（例如将 –1 用于整数）指示空值，但是这种方法只有在能够确定未使用的值的情况下有效。另一种方法是在单独的字段或变量中维护布尔空值指示符，但是这种方法不是很适合参数和返回值。第三种方法是使用一组用户定义的可空类型，但是这仅适用于属于闭集的类型集合。C# 的可空类型 ***(nullable type)*** 通过为所有值类型的可空形式提供完整和集成的支持，解决了这个由来已久的问题。

可空类型是使用 `?` 类型修饰符来构造的。例如，`int?` 是预定义类型 `int` 的可空形式。可空类型的基础类型必须是非可空的值类型。

可空类型是一个组合了基础类型的值和布尔空值指示符的结构。可空类型的实例具有两个公共只读属性：`bool` 类型的 `HasValue` 和可空类型的基础类型的 `Value`。`HasValue` 对所有非空实例都为 true，对空实例为 false。当 `HasValue` 为 true 时，`Value` 属性返回所包含的值。当 `HasValue` 为 false 时，尝试访问 `Value` 属性将引发异常。

可从任何非可空值类型隐式转换到该类型的可空形式。此外，还可从 `null` 文本隐式转换到任何可空类型。在下面的示例中

```
int? x = 123;
int? y = null;

if (x.HasValue) Console.WriteLine(x.Value);
if (y.HasValue) Console.WriteLine(y.Value);
```

`int` 值 `123` 和 `null` 文本被隐式转换为可空类型 `int?`。该示例针对 `x` 输出 `123`，但是第二个 `Console.WriteLine` 未执行，因为 `y.HasValue` 为 false。

可空转换 ***(Nullable conversion)*** 和提升转换 ***(lifted conversion)*** 允许对非可空值类型进行操作的预定义转换和用户定义转换也能够用于这些类型的可空形式。同样，提升运算符 ***(lifted operator)*** 允许用于非可空值类型的预定义运算符和用户定义运算符也能够用于这些类型的可空形式。

对于从非可空值类型 S 到非可空值类型 T 的每个预定义转换，将自动存在从 S? 到 T? 的预定义可空转换。这种可空转换是基础转换的一种空传播 ***(null propagating)*** 形式：它将空的源值直接转换为空的目标值，但是对其他值则执行基础非空转换。此外还进一步提供了从 S 到 T? 和从 S? 到 T 的可空转换，后一种转换作为在源值为空时将引发异常的显式转换。

下面是一些可空转换示例。

```
int i = 123;
int? x = i;          // int --> int?
double? y = x;       // int? --> double?
int? z = (int?)y;    // double? --> int?
int j = (int)z;      // int? --> int
```

当源和目标类型均为非可空值类型时，用户定义的转换运算符将具有提升形式。源和目标类型将添加一个 ? 修饰符以创建提升形式。与预定义的可空转换类似，提升转换运算符也可传播空值。

当操作数类型和结果类型全都为非可空值类型时，非比较运算符具有提升形式。对于非比较运算符，每个操作数类型和结果类型将添加一个 ? 修饰符以创建提升形式。例如，接受两个 int 操作数并返回一个 int 的预定义 + 运算符的提升形式是接受两个 int? 操作数并返回一个 int? 的运算符。与提升转换类似，提升非比较运算符也进行空传播：如果提升运算符的任一操作数为空，则结果为空值。

下面的示例使用 + 提升运算符将两个 int? 值相加：

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x + y;
```

对 z 的赋值实际上等价于：

```
int? z = x.HasValue && y.HasValue ? x.Value + y.Value : (int?)null;
```

由于存在从非可空值类型到其可空形式的隐式转换，当只有一个操作数是可空类型时，提升运算符同样适用。下面的示例使用与上述示例相同的提升后的 + 运算符：

```
int? x = GetNullableInt();
int? y = x + 1;
```

如果 x 为空，则 y 被赋值为空。否则，y 被赋以 x 加一的值。

C# 的可空转换、提升转换和提升非比较运算符的空传播语义非常类似于 SQL 中的对应转换和运算符。但是，C# 的提升比较运算符产生标准的布尔结果而不是引入 SQL 的三值布尔逻辑。

当操作数类型均为非可空值类型并且结果类型为 bool 时，比较运算符（==、!=、<、>、<=、>=）具有提升形式。比较运算符的提升形式是通过向每个操作数类型添加一个 ? 修饰符（但是不添加到结果类型）构成的。== 和 != 运算符的提升形式将两个空值视为相等，并且空值不等于非空值。如果一个为空或者两个操作数都为空，<、>、<= 和 >= 运算符的提升形式返回 false。

当 == 或 != 运算符的操作数之一为 null 文本时，另一个操作数可以是任何可空类型，而不管基础值类型是否实际声明了该运算符。在没有为运算符 == 或 != 提供实现的情况下，操作数的 HasValue 属性的检查将被替换。此规则的效果在于，类似如下的语句

```
if (x == null) Console.WriteLine("x is null");
if (x != null) Console.WriteLine("x is non-null");
```

对于任何可空类型或引用类型的 x 都是允许的，从而为可为空值的所有类型执行空值检查提供一种公共的方法。

另外还提供了空合并运算符 ***(null coalescing operator)*** ??。如果 a 为非空，则 a ?? b 的结果为 a；否则结果为 b。从效果看，b 提供了要在 a 为空时使用的值。

当 a 为可空类型而 b 为非可空类型时，只要操作数类型之间存在适当的隐式转换，a ?? b 将返回非可空值。在下面的示例中

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x ?? y;
int i = z ?? -1;
```

x ?? y 的类型为 int?，但是 z ?? -1 的类型为 int。后一种运算特别方便，因为它从类型移除 ?，同时还提供要在为空值时使用的默认值。

空合并运算符也适用于引用类型。下面的示例

```
string s = GetStringValue();
Console.WriteLine(s ?? "Unspecified");
```

输出 s 的值，或在 s 为空值时输出 Unspecified。

20. 泛型

20.1 泛型类声明

泛型类声明是一种类的声明，它需要提供类型实参才能构成实际类型。

类声明可以有选择地定义类型形参：

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$ *class-base*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *class-body* ;$_{opt}$

只有提供了一个 *type-parameter-list*，才可以为这个类声明提供 *type-parameter-constraints-clauses*（第 20.7 节）。

提供了 *type-parameter-list* 的类声明是一个泛型类声明。此外，任何嵌套在泛型类声明或泛型结构声明中的类本身就是一个泛型类声明，因为必须为包含类型提供类型形参才能创建构造类型。

除了明确指出的地方外，泛型类声明与非泛型类声明遵循相同的规则。泛型类声明可嵌套在非泛型类声明中。

使用构造类型 ***(constructed type)***（第 20.4 节）引用泛型类。给定下面的泛型类声明

```
class List<T> {}
```

`List<T>`、`List<int>` 和 `List<List<string>>` 就是其构造类型的一些示例。使用一个或多个类型形参的构造类型（例如 `List<T>`）称为开放构造类型 ***(open constructed type)***。不使用类型形参的构造类型（例如 `List<int>`）称为封闭构造类型 ***(closed constructed type)***。

泛型类型可根据类型形参的数目进行“重载”；这就是说，同一命名空间或外层类型声明中的两个类型声明可以使用相同标识符，只要这两个声明具有不同数目的类型形参即可。

```
class C {}
class C<V> {}
struct C<U,V> {}     // Error, C with two type parameters defined twice
class C<A,B> {}      // Error, C with two type parameters defined twice
```

类型名称解析（第 20.9.1 节）、简单名称解析（第 20.9.5 节）和成员访问（第 20.9.6 节）过程中使用的类型查找规则均会考虑类型形参的数目。

泛型类声明的基接口必须满足第 20.3.1 节中所述的唯一性规则。

20.1.1 类型形参

类型形参可在类声明中提供。每个类型形参都是一个简单标识符，代表一个为创建构造类型而提供的类型实参的占位符。类型形参是将来提供的类型的形式占位符。而类型实参（第 20.5.1 节）是在创建构造类型时替换类型形参的实际类型。

type-parameter-list:
    < type-parameters >

type-parameters:
    attributes$_{opt}$ *type-parameter*
    type-parameters , *attributes*$_{opt}$ *type-parameter*

type-parameter:
    identifier

类声明中的每个类型形参在该类的声明空间（第 3.3 节）中定义一个名称。因此，它不能与另一个类型形参或该类中声明的成员具有相同的名称。类型形参不能与类型本身具有相同的名称。

类上的类型形参的作用域（第 3.7 节）包括 *class-base*、*type-parameter-constraints-clauses* 和 *class-body*。与类的成员不同，此作用域不会扩展到派生类。在其作用域中，类型形参可用作类型。

type:
    value-type
    reference-type
    type-parameter

由于类型形参可使用许多不同的实际类型实参进行实例化，因此类型形参具有与其他类型稍微不同的操作和限制。这包括：

- 不能直接使用类型形参声明基类或接口（第 20.1.3 节）。
- 类型形参上的成员查找规则取决于应用到该类型形参的约束（如果有）。这将在第 20.7.2 节中详细描述。
- 类型形参的可用转换取决于应用到该类型形参的约束（如果有）。这将在第 20.7.4 节中详细描述。
- 如果事先不知道由类型形参给出的类型是引用类型（第 20.7.4 节），不能将标识 `null` 转换为该类型。不过，可以改为使用默认值表达式（第 25.6 节）。此外，具有由类型形参给出的类型的值可以使用 == 和 != 与 `null` 进行比较（第 20.8.3 节），除非该类型形参具有值类型约束（第 20.7.4 节）。
- 仅当类型形参受 *constructor-constraint* 或值类型约束（第 20.7 节）的约束时，才能将 new 表达式（第 20.8.1 节）与类型形参联合使用。
- 不能在属性中的任何位置上使用类型形参。
- 不能在成员访问或类型名称中使用类型形参标识静态成员或嵌套类型（第 20.9.1、第 20.9.6 节）。
- 在不安全代码中，类型形参不能用作 *unmanaged-type*（第 18.2 节）。

作为类型，类型形参纯粹是一个编译时构造。在运行时，每个类型形参都绑定到一个运行时类型，运行时类型是通过向泛型类类型声明提供类型实参来指定的。因此，使用类型形参声明的变量的类型在运行时将是封闭构造类型（第 20.5.2 节）。涉及类型形参的所有语句和表达式的运行时执行都使用作为该形参的类型实参提供的实际类型。

20.1.2 实例类型

每个类声明都有一个关联的构造类型，即实例类型 ***(instance type)***。对于泛型类声明，实例类型是通过从该类型声明创建构造类型（第 20.4 节）来构成的，所提供的每个类型实参替换对应的类型形参。由于实例类型使用类型形参，因此只能在类型形参的作用域中使用该实例类型；也就是在类声明的内部。对于在类声明中编写的代码，实例类型为 `this` 的类型。对于非泛型类，实例类型就是所声明的类。下面显示几个类声明以及它们的实例类型：

```
class A<T>                    // instance type: A<T>
{
   class B {}                 // instance type: A<T>.B
   class C<U> {}              // instance type: A<T>.C<U>
}
class D {}                    // instance type: D
```

20.1.3 基规范

类声明中指定的基类可以是构造类类型（第 20.4 节）。基类本身不能是类型形参，但在其作用域中可以包含类型形参。

```
class Extend<V>: V {}         // Error, type parameter used as base class
```

泛型类声明不能使用 System.Attribute 作为直接或间接基类。

类声明中指定的基接口可以是构造接口类型（第 20.4 节）。基接口本身不能是类型形参，但在其作用域中可以包含类型形参。下面的代码演示类实现和扩展构造类型的方法：

```
class C<U,V> {}
interface I1<V> {}
class D: C<string,int>, I1<string> {}
class E<T>: C<int,T>, I1<T> {}
```

泛型类声明的基接口必须满足第 20.3.1 节中所述的唯一性规则。

如果类中的方法重写或实现基类或接口中的方法，则必须为特定类型提供相应的方法。下面的代码演示如何重写和实现方法。第 20.1.10 节对此做了进一步的解释。

```
class C<U,V>
{
   public virtual void M1(U x, List<V> y) {...}
}
interface I1<V>
{
   V M2(V);
}
class D: C<string,int>, I1<string>
{
   public override void M1(string x, List<int> y) {...}
   public string M2(string x) {...}
}
```

20.1.4 泛型类的成员

泛型类的所有成员都可以直接或作为构造类型的一部分使用任何包容类 (enclosing class) 中的类型形参。当在运行时使用特定的封闭构造类型（第 20.5.2 节）时，所出现的每个类型形参都被替换成为该构造类型提供的实际类型实参。例如：

```
class C<V>
{
   public V f1;
   public C<V> f2 = null;
```

```
   public C(V x) {
      this.f1 = x;
      this.f2 = this;
   }
}
class Application
{
   static void Main() {
      C<int> x1 = new C<int>(1);
      Console.WriteLine(x1.f1);     // Prints 1

      C<double> x2 = new C<double>(3.1415);
      Console.WriteLine(x2.f1);     // Prints 3.1415
   }
}
```

在实例函数成员中，类型 this 是包含这些成员的声明的实例类型（第 20.1.2 节）。

除了使用类型形参作为类型以外，泛型类声明中的成员与非泛型类的成员遵循相同的规则。下面几小节将讨论适用于特定种类的成员的附加规则。

20.1.5 泛型类中的静态字段

泛型类声明中的静态变量在相同封闭构造类型（第 20.5.2 节）的所有实例之间共享，但是不会在不同封闭构造类型的实例之间共享。不管静态变量的类型是否涉及任何类型形参，这些规则都适用。

例如：

```
class C<V>
{
   static int count = 0;

   public C() {
      count++;
   }

   public static int Count {
      get { return count; }
   }
}


class Application
{
   static void Main() {
      C<int> x1 = new C<int>();
      Console.WriteLine(C<int>.Count);    // Prints 1

      C<double> x2 = new C<double>();
      Console.WriteLine(C<int>.Count);    // Prints 1

      C<int> x3 = new C<int>();
      Console.WriteLine(C<int>.Count);    // Prints 2
   }
}
```

20.1.6 泛型类中的静态构造函数

泛型类中的静态构造函数用于初始化静态字段，并为从该泛型类声明创建的每个不同封闭构造类型执行其他初始化操作。泛型类型声明的类型形参处于作用域中，并且可在静态构造函数的函数体中使用。

新的封闭构造类类型在第一次发生下列任一情况时进行初始化：

- 创建该封闭构造类型的实例。
- 引用该封闭构造类型的任何静态成员。

为了初始化新的封闭构造类类型，需要先为该特定的封闭构造类型创建一组新的静态字段（第 20.1.5 节）。将其中的每个静态字段初始化为默认值（第 5.2 节）。下一步，为这些静态字段执行静态字段初始值设定项（第 10.4.5.1 节）。最后，执行静态构造函数。

由于静态构造函数只为每个封闭构造类类型执行一次，因此对于无法通过约束（第 20.6.6 节）在编译时进行检查的类型形参来说，此处是进行运行时检查的方便位置。例如，下面的类型使用静态构造函数检查类型实参是否为一个枚举：

```
class Gen<T> where T: struct
{
   static Gen() {
      if (!typeof(T).IsEnum) {
         throw new ArgumentException("T must be an enum");
      }
   }
}
```

20.1.7 访问受保护成员

在泛型类声明中，通过从该泛型类构造的任何类类型的实例，可以对继承的受保护实例成员进行访问。具体而言，第 3.5.3 节中指定的用于访问 `protected` 和 `protected internal` 实例成员的规则通过下面针对泛型的规则进行了扩充：

- 在泛型类 `G` 中，如果 `E` 的类型是从 `G` 构造的类类型或从 `G` 构造的类类型继承的类类型，则使用 `E.M` 形式的 *primary-expression* 访问继承的受保护实例成员 `M` 是允许的。

在下面的示例中

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

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

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

20.1.8 泛型类中的重载

泛型类声明中的方法、构造函数、索引器和运算符可以被重载。虽然声明的签名必须唯一，但是在替换类型实参时可能会导致出现完全相同的签名。在这样的情况下，重载解析的附加规则将挑选最明确的成员。

下面的示例根据此规则演示有效和无效的重载：

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
   int F1(U u);               // Overload resulotion for G<int>.F1
   int F1(int i);             // will pick non-generic

   void F2(I1<U> a);          // Valid overload
   void F2(I2<U> a);
}

class G2<U,V>
{
   void F3(U u, V v);         // Valid, but overload resolution for
   void F3(V v, U u);         // G2<int,int>.F3 will fail

   void F4(U u, I1<V> v);     // Valid, but overload resolution for
   void F4(I1<V> v, U u);     // G2<I1<int>,int>.F4 will fail

   void F5(U u1, I1<V> v2);   // Valid overload
   void F5(V v1, U u2);

   void F6(ref U u);          // valid overload
   void F6(out V v);
}
```

20.1.9 形参数组方法和类型形参

可以在形参数组的类型中使用类型形参。例如，给定下面的声明

```
class C<V>
{
   static void F(int x, int y, params V[] args);
}
```

对该方法的如下展开形式的调用：

```
C<int>.F(10, 20);
C<object>.F(10, 20, 30, 40);
C<string>.F(10, 20, "hello", "goodbye");
```

完全对应于：

```
C<int>.F(10, 20, new int[] {});
C<object>.F(10, 20, new object[] {30, 40});
C<string>.F(10, 20, new string[] {"hello", "goodbye"} );
```

20.1.10 重写和泛型类

和往常一样，泛型类中的函数成员可以重写基类中的函数成员。在确定被重写的基成员时，必须通过替换类型实参来确定基类的成员，如第 20.5.4 节所述。一旦确定了基类的成员，重写规则就与非泛型类相同。

下面的示例演示重写规则如何在存在泛型的情况下起作用：

```
abstract class C<T>
{
   public virtual T F() {...}

   public virtual C<T> G() {...}

   public virtual void H(C<T> x) {...}
}
```

```
class D: C<string>
{
   public override string F() {...}          // Ok
   public override C<string> G() {...}       // Ok
   public override void H(C<T> x) {...}      // Error, should be C<string>
}
class E<T,U>: C<U>
{
   public override U F() {...}               // Ok
   public override C<U> G() {...}            // Ok
   public override void H(C<T> x) {...}      // Error, should be C<U>
}
```

20.1.11 泛型类中的运算符

泛型类声明可以定义运算符，所遵循的规则与非泛型类声明相同。运算符声明中使用类声明的实例类型（第 20.1.2 节）的方式必须与运算符的正常使用规则类似，具体如下：

- 一元运算符必须以该实例类型的单个参数为操作对象。一元的 ++ 和 -- 运算符必须返回该实例类型或从该实例类型派生的类型。
- 二元运算符的参数中必须至少有一个属于该实例类型。
- 转换运算符的形参类型或返回类型必须属于该实例类型。

下面演示泛型类中的有效运算符声明的一些示例：

```
class X<T>
{
   public static X<T> operator ++(X<T> operand) {...}
   public static int operator *(X<T> op1, int op2) {...}
   public static explicit operator X<T>(T value) {...}
}
```

对于从源类型 `S` 转换到目标类型 `T` 的转换运算符，在应用第 10.9.3 节中指定的规则时，与 `S` 或 `T` 关联的任何类型形参都被视为与其他类型没有继承关系的唯一类型，并忽略对那些类型形参的所有约束。

在下面的示例中

```
class C<T> {...}
class D<T>: C<T>
{
   public static implicit operator C<int>(D<T> value) {...}    // Ok
   public static implicit operator C<string>(D<T> value) {...} // Ok
   public static implicit operator C<T>(D<T> value) {...}      // Error
}
```

前两个运算符声明是允许的，因为根据第 10.9.3 节，`T` 和 `int` 以及 `string` 分别被视为没有关系的唯一类型。但是，第三个运算符是错误的，因为 `C<T>` 是 `D<T>` 的基类。

对于某些类型实参，可以声明这样的运算符，即这些运算符指定了已经作为预定义转换而存在的转换。在下面的示例中

```
struct Convertible<T>
{
   public static implicit operator Convertible<T>(T value) {...}
```

```
    public static explicit operator T(Convertible<T> value) {...}
}
```

当把类型 object 指定为 T 的类型实参时，第二个运算符将声明一个已经存在的转换（存在从任何类型到类型 object 的隐式转换，因此也存在显式转换）。

在两个类型之间存在预定义转换的情况下，这些类型之间的任何用户定义的转换将被忽略。具体而言：

- 如果存在从类型 S 到类型 T 的预定义隐式转换（第 6.1 节），则从 S 到 T 的所有用户定义的转换（隐式或显式）将被忽略。
- 如果存在从类型 S 到类型 T 的预定义显式转换（第 6.2 节），则从 S 到 T 的所有用户定义的显式转换将被忽略。但是，仍然会考虑从 S 到 T 的用户定义的隐式转换。

对于除 object 以外的所有类型，上面的 Convertible<T> 类型声明的运算符都不会与预定义的转换发生冲突。例如：

```
void F(int i, Convertible<int> n) {
   i = n;                          // Error
   i = (int)n;                     // User-defined explicit conversion
   n = i;                          // User-defined implicit conversion
   n = (Convertible<int>)i;        // User-defined implicit conversion
}
```

但是对于类型 object，除了下面这个特例之外，预定义的转换将在其他所有情况下隐藏用户定义的转换：

```
void F(object o, Convertible<object> n) {
   o = n;                          // Pre-defined boxing conversion
   o = (object)n;                  // Pre-defined boxing conversion
   n = o;                          // User-defined implicit conversion
   n = (Convertible<object>)o;     // Pre-defined unboxing conversion
}
```

20.1.12 泛型类中的嵌套类型

泛型类声明可以包含嵌套的类型声明。包容类的类型形参可以在嵌套类型中使用。嵌套类型声明可以包含仅适用于该嵌套类型的附加类型形参。

泛型类声明中包含的每个类型声明都隐式地是泛型类型声明。在编写对嵌套在泛型类型中的类型的引用时，必须指定其包容构造类型（包括其类型实参）。但是可在外层类中不加限定地使用嵌套类型；在构造嵌套类型时可以隐式地使用外层类的实例类型。下面的示例演示三种不同的引用从 Inner 创建的构造类型的正确方法；前两种方法是等效的：

```
class Outer<T>
{
   class Inner<U>
   {
      public static void F(T t, U u) {...}
   }

   static void F(T t) {
      Outer<T>.Inner<string>.F(t, "abc");    // These two statements have
      Inner<string>.F(t, "abc");             // the same effect

      Outer<int>.Inner<string>.F(3, "abc");  // This type is different

      Outer.Inner<string>.F(t, "abc");       // Error, Outer needs type arg
   }
}
```

嵌套类型中的类型形参可以隐藏外层类型中声明的成员或类型形参，但这是一种不好的编程风格：

```
class Outer<T>
{
   class Inner<T>    // Valid, hides Outer's T
   {
      public T t;    // Refers to Inner's T
   }
}
```

20.1.13 应用程序入口点

应用程序入口点方法（第 3.1 节）不能在泛型类声明中。

20.2 泛型结构声明

结构声明可以定义类型形参及其关联的约束：

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ `struct` *identifier* *type-parameter-list*$_{opt}$ *struct-interfaces*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *struct-body* `;`$_{opt}$

用于泛型类声明的规则同样适用于泛型结构声明，但第 11.3 节中指出的情况例外。

20.3 泛型接口声明

接口声明可以定义类型形参及其关联的约束：

interface-declaration:
attributes$_{opt}$ *interface-modifiers*$_{opt}$ `interface` *identifier* *type-parameter-list*$_{opt}$
interface-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *interface-body* `;`$_{opt}$

除了下面指出的情况外，泛型接口声明与非泛型接口声明遵循相同的规则。

接口声明中的每个类型形参在该接口的声明空间（第 3.3 节）中定义一个名称。接口上的类型形参的作用域（第 3.7 节）包括 *interface-base*、*type-parameter-constraints-clauses* 和 *interface-body*。在其作用域中，类型形参可用作类型。适用于类上的类型形参的限制（第 20.1.1 节）也适用于接口上的类型形参。

泛型接口中的方法与泛型类中的方法遵循相同的重载规则（第 20.1.8 节）。

20.3.1 所实现接口的唯一性

泛型类型声明所实现的接口必须对所有可能的构造类型都保持唯一。如果没有此规则，则无法确定要为某些构造类型调用的正确方法。例如，假设允许以如下形式声明某个泛型类：

```
interface I<T>
{
   void F();
}

class X<U,V>: I<U>, I<V>                // Error: I<U> and I<V> conflict
{
   void I<U>.F() {...}
   void I<V>.F() {...}
}
```

如果允许这样，则无法确定要在下面的情况下执行的代码：

```
I<int> x = new X<int,int>();
x.F();
```

为了确定泛型类型声明的接口列表是否有效，将执行以下步骤：

- 假设 L 是泛型类、结构或接口声明 C 中直接指定的接口列表。
- 将已经在 L 中的接口的所有基接口添加到 L。
- 移除 L 中的所有重复接口。
- 在将类型实参替换到 L 中之后，如果从 C 创建的任何可能的构造类型导致 L 中的两个接口完全相同，则 C 的声明无效。在确定所有可能的构造类型时不考虑约束声明。

在上面的类声明 X 中，接口列表 L 由 I<U> 和 I<V> 组成。该声明无效，因为任何 U 和 V 属于相同类型的构造类型都将导致这两个接口成为完全相同的类型。

可以将不同继承级别指定的接口进行统一：

```
interface I<T>
{
   void F();
}
class Base<U>: I<U>
{
   void I<U>.F() {…}
}
class Derived<U,V>: Base<U>, I<V>   // Ok
{
   void I<V>.F() {…}
}
```

虽然 Derived<U,V> 同时实现了 I<U> 和 I<V>，但是此代码是有效的。代码

```
I<int> x = new X<int,int>();
x.F();
```

调用 Derived 中的方法，因为 Derived<int,int> 实际重新实现了 I<int>（第 13.4.4 节）。

20.3.2 显式接口成员实现

本质上，显式接口成员实现使用构造接口类型的方式与使用简单接口类型相同。像通常那样，显式接口成员实现必须通过指示要实现的接口的 *interface-type* 来进行限定。该类型可以是简单接口或构造接口，如下例所示：

```
interface IList<T>
{
   T[] GetElements();
}
interface IDictionary<K,V>
{
   V this[K key];
   void Add(K key, V value);
}
class List<T>: IList<T>, IDictionary<int,T>
{
   T[] IList<T>.GetElements() {...}
   T IDictionary<int,T>.this[int index] {...}
   void IDictionary<int,T>.Add(int index, T value) {...}
}
```

20.4 泛型委托声明

委托声明可以定义类型形参及其关联的约束：

delegate-declaration:
　　attributes$_{opt}$　*delegate-modifiers*$_{opt}$　delegate　*return-type*　*identifier*　*type-parameter-list*$_{opt}$
　　　　(　*formal-parameter-list*$_{opt}$　)　*type-parameter-constraints-clauses*$_{opt}$　;

除了下面指出的地方以外，泛型委托声明与非泛型委托声明遵循相同的规则。泛型委托声明中的每个类型形参在与该委托声明关联的特殊声明空间（第 3.3 节）中定义一个名称。委托声明中的类型形参的作用域（第 3.7 节）包括 *return-type*、*formal-parameter-list* 和 *type-parameter-constraints-clauses*。

与其他泛型类型声明一样，必须提供类型实参才能创建构造委托类型。构造委托类型的形参类型和返回类型是通过将委托声明中的每个类型形参替换为构造委托类型的对应类型实参来创建的。结果返回类型和形参类型用于确定哪些方法与构造委托类型兼容。例如：

```
delegate bool Predicate<T>(T value);

class X
{
   static bool F(int i) {...}

   static bool G(string s) {...}

   static void Main() {
      Predicate<int> p1 = F;
      Predicate<string> p2 = G;
   }
}
```

注意上面的 Main 方法中的两个赋值等效于下面的长格式：

```
static void Main() {
   Predicate<int> p1 = new Predicate<int>(F);
   Predicate<string> p2 = new Predicate<string>(G);
}
```

允许短格式的原因是存在第 21.9 节中描述的方法组转换。

20.5 构造类型

泛型类型声明本身表示未绑定的泛型类型 ***(unbound generic type)***，它通过应用类型实参 ***(type argument)*** 被用作构成许多不同类型的“蓝图”。类型实参编写在紧跟在泛型类型声明的名称后面的尖括号（< 和 >）中。至少使用一个类型实参命名的类型称为构造类型 ***(constructed type)***。构造类型可以在语言中能够出现类型名的大多数地方使用。未绑定的泛型类型只能在 *typeof-expression*（第 20.8.2 节）中使用。

type-name:
　　namespace-or-type-name

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

构造类型还可以在表达式中用作简单名称（第 20.9.5 节）或在访问成员时使用（第 20.9.6 节）。

在计算 *namespace-or-type-name* 时，仅考虑具有正确数目的类型形参的泛型类型。因此，可以使用同一个标识符标识不同的类型，前提是那些类型具有不同数目的类型形参。当在同一程序中混合使用泛型和非泛型类时，这是很有用的：

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

用于在 *namespace-or-type-name* 式中查找名称的详细规则将在第 20.9 节中说明。这些式子中的多义性解析将在第 20.6.5 节中说明。

即使未直接指定类型形参，*type-name* 也可以标识构造类型。当某个类型嵌套在泛型类声明中，并且包含该类型的声明的实例类型被隐式用于名称查找（第 20.1.12 节）时，就会出现这种情况：

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

在不安全代码中，构造类型不能用作 *unmanaged-type*（第 18.2 节）。

20.5.1 类型实参

类型实参列表中的每个实参都只是一个 *type*。

type-argument-list:
    < type-arguments >

type-arguments:
    type-argument
    type-arguments , type-argument

type-argument:
    type

类型实参可以是构造类型或类型形参。在不安全代码（第 18 章）中，*type-argument* 不可以是指针类型。每个类型实参都必须满足对应的类型形参上的所有约束（第 20.7.1 节）。

20.5.2 开放和封闭类型

所有类型都可归类为开放类型 ***(open type)*** 或封闭类型 ***(closed type)***。开放类型是包含类型形参的类型。更明确地说：

- 类型形参定义开放类型。
- 当且仅当数组元素类型是开放类型时，该数组类型才是开放类型。
- 当且仅当构造类型的一个或多个类型实参为开放类型时，该构造类型才是开放类型。当且仅当构造的嵌套类型的一个或多个类型实参或其包含类型的类型实参为开放类型时，该构造的嵌套类型才是开放类型。

封闭类型是不属于开放类型的类型。

在运行时，泛型类型声明中的所有代码都在一个封闭构造类型的上下文中执行，这个封闭构造类型是通过将类型实参应用该泛型声明来创建的。泛型类型中的每个类型形参都绑定到特定的运行时类型。所有语句和表达式的运行时处理都始终使用封闭类型，开放类型仅出现在编译时处理过程中。

每个封闭构造类型都有自己的静态变量集，任何其他封闭构造类型都不会共享这些变量。由于开放类型在运行时并不存在，因此不存在与开放类型关联的静态变量。如果两个封闭构造类型是从相同的未绑定泛型类型构造的，并且它们的对应类型实参属于相同类型，则这两个封闭构造类型是相同类型。

20.5.3 构造类型的基类和接口

与简单类类型一样，构造类类型具有直接基类。如果泛型类声明未指定基类，则基类为 `object`。如果泛型类声明中指定了基类，则构造类型的基类将通过把基类声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument* 来获得。给定下面的泛型类声明

```
class B<U,V> {...}
class G<T>: B<string,T[]> {...}
```

构造类型 `G<int>` 的基类将是 `B<string,int[]>`。

类似地，构造类、结构和接口类型具有一组显式的基接口。显式基接口是通过接受泛型类型声明上的显式基接口声明，并将基接口声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument* 来构成的。

像通常那样，所构成类型的所有基类和基接口的集合是通过递归地获取直接基类和接口的基类和接口来构成的。例如，给定下面的泛型类声明：

```
class A {...}
class B<T>: A {...}
class C<T>: B<IComparable<T>> {...}
class D<T>: C<T[]> {...}
```

`D<int>` 的基类是 `C<int[]>`、`B<IComparable<int[]>>`、`A` 和 `object`。

20.5.4 构造类型的成员

构造类型的非继承成员是通过将成员声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument* 来获得的。替换过程基于类型声明的语义含义，并不只是文本替换。

例如，给定下面的泛型类声明

```
class Gen<T,U>
{
   public T[,] a;
   public void G(int i, T t, Gen<U,T> gt) {...}
   public U Prop { get {...} set {...} }
   public int H(double d) {...}
}
```

构造类型 `Gen<int[],IComparable<string>>` 具有以下成员：

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
```

```
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

泛型类声明 Gen 中的成员 a 的类型是“T 的二维数组”，因此上面的构造类型中的成员 a 的类型是“int 的一维数组的二维数组”，或 int[,][]。

构造类型的继承成员以类似的方式获得。首先，确定直接基类的所有成员。如果基类本身是构造类型，则这可能涉及当前规则的递归应用。然后，通过将成员声明中的每个 *type-parameter* 替换为构造类型的对应 *type-argument* 来转换每个继承的成员。

```
class B<U>
{
   public U F(long index) {...}
}
class D<T>: B<T[]>
{
   public T G(string s) {...}
}
```

在上面的示例中，构造类型 D<int> 具有一个非继承的成员 public int G(string s)，该成员是通过将类型形参 T 替换为类型实参 int 来获得的。D<int> 还有一个从类声明 B 继承的成员。这个继承的成员是通过下面的方式确定的，即先通过用 T[] 替换 U（从而产生 public T[] F(long index)）确定构造类型 B<T[]> 的成员。然后，用类型实参 int 替换类型形参 T，从而产生继承的成员 public int[] F(long index)。

20.5.5 构造类型的可访问性

当构造类型 $C<T_1, \ldots, T_N>$ 的所有组件 C、T_1、…、T_N 都可访问时，则该构造类型是可访问的。更准确地说，构造类型的可访问域是未绑定的泛型类型的可访问域和类型实参的可访问域的交集。

20.5.6 转换

构造类型与非泛型类型遵循相同的转换规则（第 6 章）。在应用这些规则时，必须按照第 20.5.3 节所述确定构造类型的基类和接口。

除第 6 章中描述的转换之外，构造引用类型之间不存在特殊转换。尤其是，与数组类型不同，构造引用类型不表现出“协变”转换。这意味着即使 B 是从 A 派生的，类型 List<B> 也没有到 List<A> 的转换（隐式或显式）。同样地，也不存在从 List<B> 到 List<object> 的转换。

这种情况的原因很简单：如果允许到 List<A> 的转换，那显然可以将类型为 A 的值存储到该列表中。但是这样将违背下面的不变性，即 List<B> 类型的列表中的每个对象应始终为 B 类型的值，否则在赋值到集合类中时可能会发生意外错误。

转换的行为和运行时类型检查如下所示：

```
class A {...}
class B: A {...}
class Collection {...}
class List<T>: Collection {...}
class Test
{
   void F() {
      List<A> listA = new List<A>();
      List<B> listB = new List<B>();
```

```
        Collection c1 = listA;           // Ok, List<A> is a Collection
        Collection c2 = listB;           // Ok, List<B> is a Collection
        List<A> a1 = listB;              // Error, no implicit conversion
        List<A> a2 = (List<A>)listB;     // Error, no explicit conversion
    }
}
```

20.5.7 using 别名指令

using 别名可以命名封闭构造类型，但是不能命名未提供类型实参的泛型类型声明。例如：

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
}
namespace N2
{
    using W = N1.A;         // Error, cannot name generic type
    using X = N1.A.B;       // Error, cannot name generic type
    using Y = N1.A<int>;    // Ok, can name closed constructed type
using Z<T> = N1.A<T>;   // Error, using alias cannot have type parameters
}
```

20.5.8 属性

用作属性实参表达式的 *typeof-expression*（第 20.8.2 节）可引用非泛型类型、封闭构造类型或未绑定的泛型类型，但是不能引用开放类型。

```
class A: Attribute
{
    public A(Type t) {...}
}
class G<T>
{
    [A(typeof(T))] T t;              // Error, open type in attribute
}
class X
{
    [A(typeof(List<int>))] int x;    // Ok, closed constructed type
    [A(typeof(List<>))] int y;       // Ok, unbound generic type
}
```

20.5.9 数组和泛型 IList 接口

一维数组 `T[]` 实现了接口 `System.Collections.Generic.IList<T>`（缩写为 `IList<T>`）及其基接口。相应地，存在从 `T[]` 到 `IList<T>` 及其基接口的隐式转换。此外，如果存在从 `S` 到 `T` 的隐式引用转换，则 `S[]` 实现 `IList<T>`，并且存在从 `S[]` 到 `IList<T>` 及其基接口的隐式引用转换（第 6.1.4 节）。如果存在从 `S` 到 `T` 的显式引用转换，则存在从 `S[]` 到 `IList<T>` 及其基接口的显式引用转换（第 6.2.3 节）。例如：

```
using System.Collections.Generic;

class Test
{
   static void Main() {
      string[] sa = new string[5];
      object[] oa1 = new object[5];
      object[] oa2 = sa;

      IList<string> lst1 = sa;                  // Ok
      IList<string> lst2 = oa1;                 // Error, cast needed
      IList<object> lst3 = sa;                  // Ok
      IList<object> lst4 = oa1;                 // Ok

      IList<string> lst5 = (IList<string>)oa1;  // Exception
      IList<string> lst6 = (IList<string>)oa2;  // Ok
   }
}
```

赋值操作 `lst2 = oa1` 将产生编译时错误，因为从 `object[]` 到 `IList<string>` 的转换是显式转换，不是隐式转换。强制转换 `(IList<string>)oa1` 将导致在运行时引发异常，因为 `oa1` 引用 `object[]` 而不是 `string[]`。但是，强制转换 `(IList<string>)oa2` 不会导致引发异常，因为 `oa2` 引用 `string[]`。

如果存在从 `S[]` 到 `IList<T>` 的隐式或显式引用转换，则也存在从 `IList<T>` 及其基接口到 `S[]` 的显式引用转换（第 6.2.3 节）。

当数组类型 `S[]` 实现 `IList<T>` 时，所实现的接口的有些成员可能会引发异常。该接口的实现的确切行为不在本规范讨论的范围之内。

20.6 泛型方法

泛型方法是在声明中包括了类型形参的方法。泛型方法可以在类、结构或接口声明中声明，这些类、结构或接口本身可以是泛型或非泛型。如果在泛型类型声明中声明泛型方法，则方法体可以同时引用该方法的类型形参和包含该方法的声明的类型形参。

class-member-declaration:
...
generic-method-declaration

struct-member-declaration:
...
generic-method-declaration

interface-member-declaration:
...
interface-generic-method-declaration

泛型方法是通过在方法名后面放置类型形参列表进行声明的：

generic-method-declaration:
generic-method-header method-body

generic-method-header:
attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type member-name type-parameter-list*
(*formal-parameter-list*$_{opt}$ *)* *type-parameter-constraints-clauses*$_{opt}$

interface-generic-method-declaration:
attributes$_{opt}$ **new**$_{opt}$ *return-type identifier type-parameter-list*
(*formal-parameter-list*$_{opt}$ *)* *type-parameter-constraints-clauses*$_{opt}$ *;*

泛型方法声明的 *type-parameter-list* 和 *type-parameter-constraints-clauses* 与在泛型类型声明中具有相同的语法和用途。方法的 *type-parameter* 作用于整个 *method-declaration* 范围，并且可在整个该范围中用于构成 *return-type*、*method-body* 和 *type-parameter-constraints-clauses*（但是不包括 *attributes*）中的类型。

方法类型形参的名称不能与相同方法中的普通形参的名称相同。

下面的示例查找数组中满足给定的测试委托的第一个元素（如果有）。（有关泛型委托的描述见第 20.4 节。）

```
public delegate bool Test<T>(T item);

public class Finder
{
   public static T Find<T>(T[] items, Test<T> test) {
      foreach (T item in items) {
         if (test(item)) return item;
      }
      throw new InvalidOperationException("Item not found");
   }
}
```

泛型方法不能声明为 `extern`。其他所有方法修饰符在泛型方法上都有效。

20.6.1 泛型方法签名

为了对签名进行比较，任何类型形参约束子句都被忽略，方法的类型形参的名称也被忽略，但是泛型类型形参的数目是要比较的，另外还需比较的有类型形参的序号位置（按从左向右的顺序）。下面的示例显示此规则如何影响方法签名：

```
class A {}

class B {}

interface IX
{
   T F1<T>(T[] a, int i);     // Error, both declarations have the same
   void F1<U>(U[] a, int i);  // signature because return type and type
                              // parameter names are not significant

   void F2<T>(int x);         // Ok, the number of type parameters is part
   void F2(int x);            // of the signature

   void F3<T>(T t) where T: A;   // Error, constraints are not
   void F3<T>(T t) where T: B;   // considered in signatures
}
```

20.6.2 虚泛型方法

泛型方法可以使用 `abstract`、`virtual` 和 `override` 修饰符进行声明。在匹配方法以进行重写或实现接口时，将使用第 20.6.1 节中描述的签名匹配规则。如果某个泛型方法重写基类中声明的泛型方法，或者是基接口中某个方法的显式接口成员实现，则该方法不能指定任何 *type-parameter-constraints-clauses*。在这些情况下，该方法的类型形参从被重写或被实现的方法继承约束。例如：

```
abstract class Base
{
   public abstract T F<T,U>(T t, U u) where U: T;

   public abstract T G<T>(T t) where T: IComparable;
}
```

```
interface I
{
   bool M<T>(T a, T, b) where T: class;
}
class Derived: Base, I
{
   // A "Y: X" constraint is implicitly inherited,
   // so y is implicitly convertible to type X
   public override X F<X,Y>(X x, Y y) {
      return y;
   }
   // This method is in error because an override
   // cannot include a where clause
   public override T G<T>(T t) where T: IComparable {
      ...
   }
   // A "U: class" constraint is implicitly inherited,
   // so a and b can be compared using the reference
   // type equality operators
   bool I.M<U>(U a, U b)
   {
      return a == b;
   }
}
```

F 的重写是有效的，因为类型形参名允许不相同。在 Derived.F 中，类型形参 Y 隐式地具有从 Base.F 继承的约束 Y: X。G 的重写无效，因为重写不允许指定类型形参约束。Derived 中的显式方法实现 I.M 隐式地从该接口方法继承 U: class 约束。

当泛型方法隐式地实现接口方法时，为每个方法类型形参提供的约束必须在两个声明中是等效的（在将任何接口类型形参替换为相应的类型实参之后），其中方法的类型形参按序号位置从左到右进行标识。例如：

```
interface I<A,B,C>
{
   void F<T>(T t) where T: A;
   void G<T>(T t) where T: B;
   void H<T>(T t) where T: C;
}
class C: I<object,C,string>
{
   public void F<T>(T t) {...}                   // Ok
   public void G<T>(T t) where T: C {...}        // Ok
   public void H<T>(T t) where T: string {...}  // Error
}
```

方法 C.F<T> 隐式地实现 I<object,C,string>.F<T>。在此例中，C.F<T> 不需要（也不允许）指定约束 T: object，因为 object 是所有类型形参上的隐式约束。方法 C.G<T> 隐式地实现 I<object,C,string>.G<T>，因为在将接口类型形参替换为对应的类型实参之后，该约束与接口中的约束匹配。方法 C.H<T> 的约束是错误的，因为密封类型（在此例中为 string）不能用作约束。省略该约束也是错误的，因为需要对隐式接口方法实现的约束进行匹配。因此，隐式地实现 I<object,C,string>.H<T> 是不可能的。此接口方法只能使用显式接口成员实现来实现：

```
class C: I<object,C,string>
{
   ...
   public void H<U>(U u) where U: class {...}
```

```
   void I<object,C,string>.H<T>(T t) {
      string s = t;  // Ok
      H<T>(t);
   }
}
```

在此例中，该显式接口成员实现调用严格具有更弱约束的公共方法。注意，虽然 T: string 约束无法在源代码中表示，但从 t 到 s 的赋值是有效的，因为 T 继承该约束。

20.6.3 调用泛型方法

泛型方法调用可以显式指定类型实参列表，或者可以省略类型实参列表并依赖类型推断确定类型实参。方法调用（包括泛型方法调用）的确切编译时处理将在第 20.9.7 节中描述。调用泛型方法时如果不带类型实参列表，类型推断将按照第 20.6.4 节中的描述进行。

下面的示例显示如何在类型推断之后以及在将类型实参替换到形参列表中之后进行重载解析：

```
class Test
{
   static void F<T>(int x, T y) {
      Console.WriteLine("one");
   }

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

   static void Main() {
      F<int>(5, 324);       // Ok, prints "one"
      F<byte>(5, 324);      // Ok, prints "two"
      F<double>(5, 324);    // Error, ambiguous

      F(5, 324);            // Ok, prints "one"
      F(5, 324L);           // Error, ambiguous
   }
}
```

20.6.4 类型实参推断

当不指定类型实参调用泛型方法时，类型推断 ***(type inference)*** 过程将尝试为该调用推断类型实参。类型推断的存在允许使用更方便的语法调用泛型方法，并使得程序员不必指定多余的类型信息。例如，给定下面的方法声明：

```
class Chooser
{
   static Random rand = new Random();

   public static T Choose<T>(T first, T second) {
      return (rand.Next(2) == 0)? first: second;
   }
}
```

可以在不显式指定类型实参的情况下调用 Choose 方法：

```
int i = Chooser.Choose(5, 213);              // Calls Choose<int>
string s = Chooser.Choose("foo", "bar");     // Calls Choose<string>
```

借助于类型推断，可通过传递给方法的实参来确定类型实参为 int 和 string。

类型推断作为方法调用的编译时处理的一部分进行（第 20.9.7 节），并且在调用的重载解析步骤之前进行。当在方法调用中指定了特定的方法组，并且没有在方法调用中指定类型实参时，将会对该方法组中的每个泛型方法应用类型推断。如果类型推断成功，则使用推断出的类型实参确定用于后续重载解析的实参的类型。如果重载决择选择一个泛型方法作为要调用的方法，则使用推断出的类型实参作为用于调用的实际类型实参。如果特定方法的类型推断失败，则该方法不参与重载决策。类型推断失败本身不会导致编译时错误。但是，如果随后的重载解析无法找到任何适用的方法，则它通常会导致编译时错误。

如果所提供的实参的数目与方法中的形参的数目不同，则推断立即失败。否则，将首先对提供给方法的每个常规实参进行独立的类型推断。假设此实参具有类型 A，对应的形参具有类型 P。将按下列步骤来关联 A 和 P，并由此进行类型推断：

- 如果下列任何条件成立，则不能从该实参推断出任何结论（但是类型推断成功）：
  - P 不涉及任何方法类型形参。
  - 实参为 null 文本。
  - 实参为匿名方法。
  - 实参为方法组。
- 只要一个条件为真，则重复下列步骤：
  - 如果 P 是数组类型，A 是具有相同秩的数组类型，则分别将 A 和 P 替换为 A 和 P 的元素类型。
  - 如果 P 是从 IEnumerable<T>、ICollection<T> 或 IList<T>（全都在 System.Collections.Generic 命名空间中）构造的类型，并且 A 是一维数组类型，则分别将 A 和 P 替换为 A 和 P 的元素类型。
- 如果 P 是数组类型（意味着前一步未能将 A 与 P 关联），则该泛型方法的类型推断失败。
- 如果 P 是方法类型形参，则此实参的类型推断成功，并且 A 是为该类型形参推断出的类型。
- 否则，P 一定是构造类型。对于出现在 P 中的每个方法类型形参 M_x，如果只能确切确定一个类型 T_x，使得将每个 M_x 替换为每个 T_x 将产生可通过标准隐式转换将 A 转换到的类型，则此实参的推断成功，并且每个 T_x 就是为每个 M_x 推断出的类型。为了进行类型推断，方法类型形参约束（如有）将被忽略。对于给定的 M_x，如果不存在 T_x 或存在多个 T_x，则泛型方法的类型推断失败（仅当 P 为泛型接口类型，并且 A 实现了该接口的多个构造版本时，才会出现存在多个 T_x 的情况）。

如果使用上述算法能成功处理所有方法实参，则从实参得出的所有推断都将存储在池中。如果下面两个条件都成立，则认为给定的泛型方法和实参列表的类型推断已成功：

- 方法的每个类型形参都有一个为它推断出的类型实参（简而言之，该推断集是完整的 ***(complete)***）。
- 对于每个类型形参，该类型形参的所有推断都推断出相同类型实参（简而言之，该推断集是一致的 ***(consistent)***）。

如果泛型方法是使用形参数组（第 10.5.1.4 节）声明的，则首先对正常形式的方法执行类型推断。如果类型推断成功，并且结果方法适用，则该方法能够以其正常形式进行重载解析。否则，将对该方法的展开形式执行类型推断（第 7.4.2.1 节）。

20.6.5 语法多义性

simple-name（第 20.9.5 节）和 *member-access*（第 20.9.6 节）的表述形式可能引起表达式的语法多义性。例如，语句：

```
F(G<A,B>(7));
```

可解释为用两个实参 `G < A` 和 `B > (7)` 调用 `F`。或者，也可以将它解释为用一个实参调用 `F`，该实参是使用两个类型实参和一个常规实参对泛型方法 `G` 的调用。

如果某个标记序列可被分析（在上下文中）为以 *type-argument-list*（第 20.5.1 节）结尾的 *simple-name*（第 20.9.5 节）、*member-access*（第 20.9.6 节）或 *pointer-member-access*（第 18.5.2 节），则会检查紧跟在结束 `>` 标记后面的标记。如果它是下列标记之一

```
(  )  ]  :  ;  ,  .  ?  ==  !=
```

则将 *type-argument-list* 保留为 *simple-name*、*member-access* 或 *pointer-member-access* 的一部分，并丢弃该标记序列的其他任何可能的分析。否则，*type-argument-list* 不视为 *simple-name*、*member-access* 或 *pointer-member-access* 的一部分，即使不存在该标记序列的其他可能的分析。注意，在分析 *namespace-or-type-name* 中的 *type-argument-list*（第 20.9.1 节）时，不应用这些规则。语句

```
F(G<A,B>(7));
```

将（按照此规则）被解释为使用一个实参对 `F` 进行调用，该实参是使用两个类型实参和一个常规实参对泛型方法 `G` 的调用。语句

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

都被解释为使用两个实参调用 `F`。语句

```
x = F < A > +y;
```

将被解释为小于运算符、大于运算符和一元加运算符，如同语句 `x = (F < A) > (+y)`，而不是在带 *type-argument-list* 的 *simple-name* 后面跟着一个一元加运算符。在语句

```
x = y is C<T> + z;
```

中，标记 `C<T>` 被解释为带 *type-argument-list* 的 *namespace-or-type-name*。

20.6.6 通过委托使用泛型方法

可以创建引用泛型方法声明的委托实例。委托创建表达式（包括引用泛型方法的委托创建表达式）的具体编译过程将在第 21.10 节中描述。

在通过委托调用泛型方法时使用的类型实参是在实例化委托时确定的。类型实参可以通过 *type-argument-list* 显式给出，或者通过类型推断（第 20.6.4 节）来确定。如果使用类型推断，则在推断过程中使用委托的形参类型作为实参类型。委托的返回类型不用于推断。下面的示例展示了两种向委托实例化表达式提供类型实参的方法：

```
delegate int D(string s, int i);
delegate int E();
class X
{
   public static T F<T>(string s, T t) {...}
   public static T G<T>() {...}
```

```
   static void Main() {
      D d1 = new D(F<int>);      // Ok, type argument given explicitly
      D d2 = new D(F);           // Ok, int inferred as type argument

      E e1 = new E(G<int>);      // Ok, type argument given explicitly
      E e2 = new E(G);           // Error, cannot infer from return type
   }
}
```

每当使用泛型方法创建委托实例时，都会在创建委托实例时给出或推断类型实参，并且不会在调用委托时提供 *type-argument-list*。

20.6.7 不能是泛型的成员

属性、事件、索引器、运算符、构造函数和析构函数本身不能具有类型形参。但是，它们可以出现在泛型类型中并使用包容类型中的类型形参。

20.7 约束

泛型类型和方法声明可以通过包括 *type-parameter-constraints-clause* 指定类型形参约束。

type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
    `where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
    primary-constraint
    secondary-constraints
    constructor-constraint
    primary-constraint `,` *secondary-constraints*
    primary-constraint `,` *constructor-constraint*
    secondary-constraints `,` *constructor-constraint*
    primary-constraint `,` *secondary-constraints* `,` *constructor-constraint*

primary-constraint:
    class-type
    `class`
    `struct`

secondary-constraints:
    interface-type
    type-parameter
    secondary-constraints `,` *interface-type*
    secondary-constraints `,` *type-parameter*

constructor-constraint:
    `new` `(` `)`

每个 *type-parameter-constraints-clause* 包括标记 `where`，后面跟着类型形参的名称，再跟着一个冒号和该类型形参的约束列表。每个类型形参最多只能有一个 `where` 子句，并且 `where` 子句可以按任何顺序列出。与属性访问器中的 `get` 和 `set` 标记一样，`where` 标记不是关键字。

`where` 子句中给出的约束列表可以包括下面的任何组件（按如下顺序）：单个主要约束、一个或多个次要约束、构造函数约束、`new()`。

主要约束可以是类类型或引用类型约束 ***(reference type constraint)*** `class` 或值类型约束 ***(value type constraint)*** `struct`。次要约束可以是 *type-parameter* 或 *interface-type*。

引用类型约束指定用于类型形参的类型实参必须是引用类型。所有类类型、接口类型、委托类型、数组类型和已知将是引用类型（将在下面定义）的类型形参都满足此约束。

值类型约束指定用于类型形参的类型实参必须是非可空值类型。所有非可空结构类型、枚举类型和具有值类型约束的类型形参都满足此约束。注意，虽然可空类型（第 24.1 节）被分类为值类型，但是不满足值类型约束。具有值类型约束的类型形参还不能具有 *constructor-constraint*。

指针类型从不允许作为类型实参，并且不被视为满足引用类型或值类型约束。

如果约束是类类型、接口类型或类型形参，则该类型指定用于该类型形参的每个类型实参必须支持的最低“基类型”。每当使用构造类型或泛型方法时，都会在编译时根据类型形参上的约束检查类型实参。所提供的类型实参必须从为该类型形参给出的约束派生或者实现所有这些约束。

class-type 约束必须满足下列规则：

- 该类型必须是类类型。
- 该类型一定不能为 `sealed`。
- 该类型一定不能是下列类型之一：`System.Array`、`System.Delegate`、`System.Enum` 或 `System.ValueType`。
- 该类型一定不能是 `object`。由于所有类型都派生自 `object`，允许这样的约束没有任何作用。
- 给定的类型形参至多只能有一个约束可以是类类型。

指定为 *interface-type* 约束的类型必须满足下列规则：

- 该类型必须是接口类型。
- 不能在给定的 `where` 子句中多次指定某个类型。

在任一情况下，该约束都可以包括关联的类型或方法声明的任何类型形参作为构造类型的组成部分，并且可以包括被声明的类型。

指定为类型形参约束的任何类或接口类型至少与声明的泛型类型或方法具有相同的可访问性（第 3.5.4 节）。

指定为 *type-parameter* 约束的类型必须满足下列规则：

- 该类型必须是类型形参。
- 不能在给定的 `where` 子句中多次指定某个类型。

此外，类型形参的依赖关系图中一定不能存在循环，其中依赖性是通过下列方式定义的传递关系：

- 如果类型形参 `T` 用作类型形参 `S` 的约束，则 `S` 依赖 ***(depend on)*** `T`。
- 如果类型形参 `S` 依赖类型形参 `T`，并且 `T` 依赖类型形参 `U`，则 `S` 依赖 ***(depend on)*** `U`。

根据这个关系，如果类型形参依赖自身（直接或间接），则会产生编译时错误。

相互依赖的类型形参之间的任何约束都必须一致。如果类型形参 `S` 依赖类型形参 `T`，则：

- `T` 一定不能具有值类型约束。否则，`T` 被有效地密封，使得 `S` 将被强制为与 `T` 相同的类型，从而消除了使用这两个类型形参的需要。

- 如果 S 具有值类型约束，则 T 一定不能具有 *class-type* 约束。
- 如果 S 具有 *class-type* 约束 A，T 具有 *class-type* 约束 B，则必须存在从 A 到 B 的标识转换或隐式引用转换或者从 B 到 A 的隐式引用转换。
- 如果 S 还依赖类型形参 U，并且 U 具有 *class-type* 约束 A，T 具有 *class-type* 约束 B，则必须存在从 A 到 B 的标识转换或隐式引用转换或者从 B 到 A 的隐式引用转换。

S 具有值类型约束而 T 具有引用类型约束是有效的。这实际上将 T 限制到类型 System.Object、System.ValueType、System.Enum 和任何接口类型。

如果类型形参的 where 子句包括构造函数约束（具有 new() 形式），则可以使用 new 运算符创建该类型的实例（第 20.8.1 节）。用于具有构造函数约束的类型形参的任何类型实参必须具有公共的无参数构造函数（任何值类型都隐式地存在此构造函数），或者是具有值类型约束或构造函数约束的类型形参（有关详细信息请参见第 20.7.1 节）。

下面是约束的示例：

```
interface IPrintable
{
   void Print();
}

interface IComparable<T>
{
   int CompareTo(T value);
}

interface IKeyProvider<T>
{
   T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>
   where K: IComparable<K>
   where V: IPrintable, IKeyProvider<K>, new()
{
   ...
}
```

下面的示例是错误的，因为它将导致类型形参的依赖关系发生循环：

```
class Circular<S,T>
   where S: T
   where T: S           // Error, circularity in dependency graph
{
   ...
}
```

下面的示例演示其他无效情况：

```
class Sealed<S,T>
   where S: T
   where T: struct      // Error, T is sealed
{
   ...
}

class A {...}
```

```
class B {...}

class Incompat<S,T>
   where S: A, T
   where T: B           // Error, incompatible class-type constraints
{
   ...
}

class StructWithClass<S,T,U>
   where S: struct, T
   where T: U
   where U: A           // Error, A incompatible with struct
{
   ...
}
```

类型形参 T 的有效基类 ***(effective base class)*** 定义如下：

- 如果 T 没有主要约束或类型形参约束，则其有效基类为 object。
- 如果 T 具有值类型约束，则其有效基类为 System.ValueType。
- 如果 T 具有 *class-type* 约束 C，但是没有 *type-parameter* 约束，则其有效基类为 C。
- 如果 T 没有 *class-type* 约束，但是有一个或多个 *type-parameter* 约束，则其有效基类为其 *type-parameter* 约束的有效基类集中被包含程度最大的类型（第 6.4.2 节）。一致性规则确保存在这样的被包含程度最大的类型。
- 如果 T 同时具有 *class-type* 约束和一个或多个 *type-parameter* 约束，则其有效基类为包含 T 的 *class-type* 约束及其 *type-parameter* 约束的有效基类的集合中被包含程度最大的类型（第 6.4.2 节）。一致性规则确保存在这样的被包含程度最大的类型。
- 如果 T 具有引用类型约束，但是没有 *class-type* 约束，则其有效基类为 object。

类型形参 T 的有效接口集 ***(effective interface set)*** 定义如下：

- 如果 T 没有 *secondary-constraints*，则其有效接口集为空。
- 如果 T 具有 *interface-type* 约束，但是没有 *type-parameter* 约束，则其有效接口集为其 *interface-type* 约束的集合。
- 如果 T 没有 *interface-type* 约束，但是具有 *type-parameter* 约束，则其有效接口集为其 *type-parameter* 约束的有效接口集的并集。
- 如果 T 同时具有 *interface-type* 约束和 *type-parameter* 约束，则其有效接口集为其 *interface-type* 约束集和其 *type-parameter* 约束的有效接口集的并集。

如果类型形参具有引用类型约束，或其有效基类不为 object 或 System.ValueType，则该类型形参将视为一个引用类型 ***(known to be a reference type)***。

受约束的类型形参类型的值可用于访问约束所暗示的实例成员。在下面的示例中

```
interface IPrintable
{
   void Print();
}
```

```
class Printer<T> where T: IPrintable
{
   void PrintOne(T x) {
      x.Print();
   }
}
```

可直接在 x 上调用 IPrintable 的方法，因为 T 被约束为始终实现 IPrintable。

20.7.1 满足约束

每当引用构造类型或泛型方法时，都会根据泛型类型或方法上声明的类型形参约束对所提供的类型实参进行检查。对于每个 where 子句，将根据每个约束检查与命名的类型形参相对应的类型实参 A，如下所示：

- 如果约束为类类型、接口类型或类型形参，则假设 C 表示该约束，并用所提供的类型实参替换出现在该约束中的任何类型形参。若要满足该约束，必须可通过下列方式之一将类型 A 转换为类型 C：
  - 标识转换（第 6.1.1 节）
  - 隐式引用转换（第 6.1.4 节）
  - 装箱转换（第 6.1.5 节）-- 前提是类型 A 为非可空值类型。
  - 从类型形参 A 到 C 的隐式引用、装箱或类型形参转换（第 20.7.4 节）。
- 如果约束为引用类型约束 (class)，则类型 A 必须满足下列条件之一：
  - A 为接口类型、类类型、委托类型或数组类型。注意，System.ValueType 和 System.Enum 是满足此约束的引用类型。
  - A 是已知为引用类型的类型形参（第 20.7 节）。
- 如果约束为值类型约束 (struct)，则类型 A 必须满足下列条件之一：
  - A 为结构类型或枚举类型，但不是可空类型。注意，System.ValueType 和 System.Enum 是不满足此约束的引用类型。
  - A 为具有值类型约束的类型形参（第 20.7 节）。
- 如果约束为构造函数约束 new()，则类型 A 一定不能为 abstract，并且必须具有公共无参数构造函数。如果下列条件之一成立，则满足此条件：
  - A 为值类型，因为所有值类型都具有公共默认构造函数（第 4.1.2 节）。
  - A 为具有构造函数约束的类型形参（第 20.7 节）。
  - A 为具有值类型约束的类型形参（第 20.7 节）。
  - A 是不为 abstract 并且包含显式声明的无参数 public 构造函数的类。
  - A 不为 abstract，并且具有默认构造函数（第 10.10.4 节）。

如果给定的类型实参未满足一个或多个类型形参的约束，则会发生编译时错误。

由于类型形参未被继承，因此约束也从不被继承。在下面的示例中，D 需要指定其类型形参 T 上的约束，以便 T 满足基类 B<T> 所施加的约束。相反，类 E 不需要指定约束，因为对于任何 T，List<T> 都实现 IEnumerable。

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

20.7.2 类型形参上的成员查找

在由类型形参 T 给出的类型中，成员查找的结果取决于为 T 指定的约束（如果有）。如果 T 没有 *class-type*、*interface-type* 或 *type-parameter* 约束，则 T 上的成员查找与 object 上的成员查找返回相同的成员集。否则，在成员查找的第一阶段（第 20.9.2 节），将考虑 T 的有效基类中的所有成员和 T 的有效接口集的每个接口中的所有成员。在为这其中每个类型执行第一阶段的成员查找之后，将对结果进行组合，然后从组合的结果中移除隐藏成员。

在泛型出现之前，成员查找始终返回仅在类中声明的成员集或仅在接口中声明的成员集，可能还会返回类型 object。类型形参上的成员查找与此有所不同。当类型形参同时具有除 object 以外的有效基类和非空有效接口集时，成员查找可以返回这样的成员集，即其中有些成员是在类中声明的，另外的成员则是在接口中声明的。下面的附加规则将处理这种情况。

- 正如第 20.9.2 节中所述，在成员查找期间，除 object 以外的在类中声明的成员会隐藏接口中声明的成员。
- 在方法（第 7.5.5.1 节）和索引器（第 7.5.6.2 节）的重载解析期间，如果在除 object 以外的类中声明了任何适用的成员，则接口中声明的所有成员都将从所考虑的成员集中移除。

仅当在绑定同时具有除 object 以外的有效基类和非空有效接口集的类型形参时，这些规则才有效。非正式地讲，类类型约束中定义的成员优先于接口约束中定义的成员。

20.7.3 类型形参和装箱

当结构类型重写从 System.Object 继承的虚方法（如 Equals、GetHashCode 或 ToString）时，通过该结构类型的实例进行的虚方法调用不会导致装箱。即使将该结构用作类型形参，并且通过类型形参类型的实例进行调用，情况也是如此。例如：

```
using System;
struct Counter
{
   int value;
   public override string ToString() {
      value++;
      return value.ToString();
   }
}
class Program
{
   static void Test<T>() where T: new() {
      T x = new T();
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
   }
   static void Main() {
      Test<Counter>();
   }
}
```

该程序的输出为：

```
1
2
3
```

虽然让 ToString 具有副作用是一种不好的做法，但是该示例证明了 x.ToString() 的三个调用没有发生装箱。

类似地，在受约束的类型形参上访问成员时，从来不会隐式地进行装箱。例如，假设接口 ICounter 包含可用于修改值的方法 Increment。如果将 ICounter 用作约束，则会用对在其上调用 Increment 的变量（从来不是装箱的副本）的引用调用 Increment 方法的实现。

```
using System;

interface ICounter
{
   void Increment();
}

struct Counter: ICounter
{
   int value;

   public override string ToString() {
      return value.ToString();
   }

   void ICounter.Increment() {
      value++;
   }
}

class Program
{
   static void Test<T>() where T: ICounter, new() {
      T x = new T();
      Console.WriteLine(x);
      x.Increment();               // Modify x
      Console.WriteLine(x);
      ((ICounter)x).Increment();   // Modify boxed copy of x
      Console.WriteLine(x);
   }

   static void Main() {
      Test<Counter>();
   }
}
```

对 Increment 的第一个调用修改变量 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
```

20.7.4 涉及类型形参的转换

给定的类型形参 T 存在下列隐式转换：

- 从 T 到其有效基类 C、从 T 到 C 的任何基类，以及从 T 到 C 实现的任何接口。在运行时，如果 T 为值类型，转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。

- 从 T 到 T 的有效接口集中的接口类型 I 和从 T 到 I 的任何基接口。在运行时，如果 T 为值类型，转换将作为装箱转换执行。否则，转换将作为隐式引用转换或标识转换执行。
- 从 T 到类型形参 U（假定 T 依赖 U）。在运行时，如果 T 为值类型，U 为引用类型，则转换将作为装箱转换执行。否则，如果 T 和 U 都是值类型，则 T 和 U 必须是相同类型，并且不执行任何转换。否则，如果 T 是引用类型，则 U 一定也是引用类型，并且转换将作为隐式引用转换或标识转换执行。
- 从空类型到 T（假定 T 已知为引用类型）。

如果 T 已知为引用类型，则上面的转换全都归类为隐式引用转换（第 6.1.4 节）。如果 T 已知不为引用类型，则上文中前两个项目符号中描述的转换被归类为装箱转换（第 6.1.5 节）。

给定的类型形参 T 存在下列显式转换：

- 从 T 的有效基类 C 到 T 和从 C 的任何基类到 T。在运行时，如果 T 为值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从任何接口类型到 T。在运行时，如果 T 为值类型，则转换将作为取消装箱转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从 T 到任何 *interface-type* I，前提是还不存在从 T 到 I 的隐式转换。在运行时，如果 T 为值类型，则转换将先作为装箱转换执行，然后作为显式引用转换执行。否则，转换将作为显式引用转换或标识转换执行。
- 从类型形参 U 到 T，前提是 T 依赖 U。在运行时，如果 T 为值类型，U 为引用类型，则转换将作为取消装箱转换执行。否则，如果 T 和 U 都是值类型，则 T 和 U 必须是相同类型，并且不执行任何转换。否则，如果 T 是引用类型，则 U 一定也是引用类型，并且转换将作为显式引用转换或标识转换执行。

如果 T 已知为引用类型，则上面的转换全都归类为显式引用转换（第 6.2.3 节）。如果 T 已知不为引用类型，则上面前两个项目符号中描述的转换被归类为取消装箱转换（第 6.2.4 节）。

上面的规则不允许从未受约束的类型形参到非接口类型的直接显式转换，这可能有点奇怪。其原因是为了防止混淆，并使得此类转换的语义更清楚。例如，请考虑下面的声明：

```
class X<T>
{
   public static long F(T t) {
      return (long)t;            // Error
   }
}
```

如果允许从 t 到 int 的直接显式转换，人们很可能以为 X<int>.F(7) 将返回 7L。但结果不是这样，因为仅当在编译时已知类型将是数字时，才会考虑标准数字转换。为了使语义清楚，必须将上面的示例改写为：

```
class X<T>
{
   public static long F(T t) {
      return (long)(object)t;    // Ok, but will only work when T is long
   }
}
```

此代码现在可以正常编译，但是在运行时执行 X<int>.F(7) 将引发异常，因为不能将装箱的 int 直接转换为 long。

20.8 表达式和语句

有些表达式和语句的运算针对泛型进行了修改。本节将详细描述这些变化。

20.8.1 对象创建表达式

对象创建表达式的类型可以是类型形参。当在对象创建表达式中指定类型形参作为类型时，必须同时满足下面两个条件，否则将发生编译时错误：

- 实参列表必须为空。
- 必须已经为类型形参指定了值类型约束或构造函数约束。

对象创建表达式的执行是通过创建类型形参所绑定到的运行时类型的实例并调用该类型的默认构造函数来完成的。运行时类型可以是引用类型或值类型。

20.8.2 typeof 运算符

`typeof` 运算符用于获得类型的 `System.Type` 对象。

typeof-expression:
    `typeof ( type )`
    `typeof ( unbound-type-name )`
    `typeof ( void )`

unbound-type-name:
    identifier generic-dimension-specifier$_{opt}$
    identifier `::` *identifier generic-dimension-specifier*$_{opt}$
    unbound-type-name `.` *identifier generic-dimension-specifier*$_{opt}$

generic-dimension-specifier:
    `<` *commas*$_{opt}$ `>`

commas:
    `,`
    commas `,`

typeof-expression 的第一种形式由 `typeof` 关键字后跟带括号的 *type* 组成。这种形式的表达式的结果是给定类型的 `System.Type` 对象。任何给定的类型都只有一个 `System.Type` 对象。这意味着对于类型 `T`，`typeof(T) == typeof(T)` 总是为 true。

typeof-expression 的第二种形式由 `typeof` 关键字后跟带括号的 *unbound-type-name* 组成。*unbound-type-name* 与 *type-name*（第 3.8 节）非常类似，只不过 *unbound-type-name* 包含 *generic-dimension-specifier*，而 *type-name* 包含 *type-argument-list*。当 *typeof-expression* 的操作数是同时满足 *unbound-type-name* 和 *type-name* 的语法的标记序列，即当它既不包含 *generic-dimension-specifier* 也不包含 *type-argument-list* 时，该标记序列被视为是一个 *type-name*。*unbound-type-name* 的含义按下述步骤确定：

- 通过将每个 *generic-dimension-specifier* 替换为与 *type-argument* 具有相同数目的逗号和关键字 `object` 的 *type-argument-list*，从而将标记序列转换为 *type-name*。
- 计算结果 *type-name*，同时忽略所有类型形参约束。
- *unbound-type-name* 解析为与结果构造类型关联的未绑定的泛型类型（第 20.5 节）。

typeof-expression 的结果是所产生的未绑定泛型类型的 `System.Type` 对象。

typeof-expression 的第三种形式由 typeof 关键字后跟带括号的 void 关键字组成。这种形式的表达式的结果是一个表示“类型不存在”的 System.Type 对象。这种通过 typeof(void) 返回的类型对象与为任何类型返回的类型对象截然不同。这种特殊类型对象在如下类库中很有用：这些类库允许在语言中反射到方法，其中那些方法希望有一种用 System.Type 的实例表示任何方法（包括 void 方法）的返回类型的途径。

typeof 运算符可以在类型形参上使用。结果为绑定到该类型形参的运行时类型的 System.Type 对象。typeof 运算符还可以在构造类型或未绑定的泛型类型上使用（第 20.5 节）。未绑定的泛型类型的 System.Type 对象与实例类型的 System.Type 对象不同。实例类型在运行时始终是封闭构造类型，因此其 System.Type 对象取决于正在使用的运行时类型实参，而未绑定的泛型类型没有类型实参。

下面的示例

```
class X<T>
{
   public static void PrintTypes() {
      Console.WriteLine(typeof(T));
      Console.WriteLine(typeof(X<T>));
      Console.WriteLine(typeof(X<X<T>>));
      Console.WriteLine(typeof(X<>));
   }
}

class M
{
   static void Main() {
      X<int>.PrintTypes();
   }
}
```

产生下列输出：

```
System.Int32
x`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

注意 typeof(X<>) 的结果与类型实参无关，但是 typeof(X<T>) 的结果却取决于类型实参。

20.8.3 引用相等运算符

如果 T 已知为引用类型（第 20.7 节），可以使用引用类型相等运算符（第 7.9.6 节）比较类型形参 T 的值。

引用相等运算符的使用有点宽松，以允许一个实参为类型形参 T 的类型，另一个实参为 null，前提是 T 没有值类型约束（第 20.7 节）。在运行时，如果 T 为值类型，则比较结果为 false。

下面的示例检查未受约束的类型形参类型的实参是否为 null。

```
class C<T>
{
   void F(T x) {
      if (x == null) throw new ArgumentNullException();
      ...
   }
}
```

虽然 T 可能表示值类型，但是 x == null 构造是允许的，当 T 为值类型时，结果只是被定义为 false。

20.8.4 is 运算符

is 运算符支持开放类型。在 e is T 形式的运算中，如果 e 或 T 的编译时类型为开放类型，则始终对 e 和 T 的运行时类型执行动态类型检查。

有关对 is 运算符的更改的更多详细信息，请参见第 24.3.5 节。

20.8.5 as 运算符

只有当 T 已知为引用类型时（第 20.7 节），as 运算符才可在右侧用于类型形参 T。需要这个限制是因为值 null 可能作为该运算符的结果返回。

```
class X
{
   public T F<T>(object o) where T: Attribute {
      return o as T;          // Ok, T has a class constraint
   }

   public T G<T>(object o) {
      return o as T;          // Error, unconstrained T
   }
}
```

有关对 as 运算符的更改的更多详细信息，请参见第 24.3.6 节。

20.8.6 异常语句

当用于开放类型时，throw（第 8.9.5 节）和 try（第 8.10 节）语句的一般规则将适用：

- throw 语句可用于其类型由类型形参给出的表达式，前提是该类型形参具有 System.Exception（或其子类）作为其有效基类（第 20.7 节）。
- catch 子句中命名的类型可以是类型形参，前提是该类型形参具有 System.Exception（或其子类）作为其有效基类（第 20.7 节）。

20.8.7 lock 语句

lock 语句可用于其类型由类型形参给出的表达式，前提是该类型形参已知为引用类型（第 20.7 节）。

20.8.8 using 语句

using 语句（第 8.13 节）遵循以下一般规则：表达式必须可隐式转换为 System.IDisposable。如果类型形参受 System.IDisposable 约束，则该类型的表达式可用于 using 语句。

20.8.9 foreach 语句

在如下形式的 foreach 语句中

```
foreach (ElementType element in collection) statement
```

如果 collection 表达式是一个未实现集合模式但是只针对一个类型 T 实现了构造接口 System.Collections.Generic.IEnumerable<T> 的类型，则该 foreach 语句的展开形式为：

```
IEnumerator<T> enumerator = ((IEnumerable<T>)(collection)).GetEnumerator();
try {
   while (enumerator.MoveNext()) {
      ElementType element = (ElementType)enumerator.Current;
      statement;
   }
}
finally {
   enumerator.Dispose();
}
```

20.9 查找规则的修改

泛型修改了用于查找和绑定名称的一些基本规则。下面几小节将在考虑泛型的情况下重述所有基本名称查找规则。

20.9.1 命名空间和类型名称

下列内容取代第 3.8 节：

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

namespace-name:
　　namespace-or-type-name

type-name:
　　namespace-or-type-name

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

qualified-alias-member 式是在第 25.3.1 节中定义的。

namespace-name 是引用一个命名空间的 *namespace-or-type-name*。根据如下所述的解析过程，*namespace-name* 的 *namespace-or-type-name* 必须引用一个命名空间，否则将发生编译时错误。*namespace-name* 中不能存在任何类型实参（第 20.5.1 节），只有类型才能具有类型实参。

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

namespace-or-type-name 具有四种形式之一：

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

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

namespace-or-type-name 的含义按下述步骤确定：

- 如果 *namespace-or-type-name* 的形式为 `I` 或 `I<A1, ..., AK>`：
  - 如果 `K` 为零，*namespace-or-type-name* 出现在泛型方法声明体中（第 20.6 节），并且如果该声明包含名为 `I` 的类型形参（第 20.1.1 节），则 *namespace-or-type-name* 引用该类型形参。
  - 否则，如果 *namespace-or-type-name* 出现在类型声明体中，则对于每个实例类型 `T`（第 20.1.2 节），从该类型声明的实例类型开始，并对每个包容类或结构声明（如果有）的实例类型继续如下过程：
    - 如果 `K` 为零，并且 `T` 的声明包含名为 `I` 的类型形参，则 *namespace-or-type-name* 引用该类型形参。
    - 否则，如果 `T` 包含具有名称 `I` 且有 `K` 个类型形参的嵌套的可访问类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。如果存在多个这样的类型，则选择在派生程度最大的类型中声明的类型。请注意，在确定 *namespace-or-type-name* 的含义时，将忽略非类型成员（常量、字段、方法、属性、索引器、运算符、实例构造函数、析构函数和静态构造函数）和具有不同数目的类型形参的类型成员。
  - 否则，对于每个命名空间 `N`（从出现 *namespace-or-type-name* 的命名空间开始，继续到每个包容命名空间（如果有），到全局命名空间结束），对下列步骤进行计算，直到找到实体：
    - 如果 `K` 为零，并且 `I` 为 `N` 中的命名空间的名称，则：
      - 如果出现 *namespace-or-type-name* 的位置包含在 `N` 的命名空间声明中，并且该命名空间声明包含将名称 `I` 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *namespace-or-type-name* 是不明确的，并将发生编译时错误。
      - 否则，*namespace-or-type-name* 引用 `N` 中名为 `I` 的命名空间。
    - 否则，如果 `N` 包含一个具有名称 `I` 且有 `K` 个类型形参的可访问类型，则：
      - 如果 `K` 为零，并且出现 *namespace-or-type-name* 的位置包含在 `N` 的命名空间声明中，并且该命名空间声明包含将名称 `I` 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *namespace-or-type-name* 是不明确的，并将发生编译时错误。
      - 否则，*namespace-or-type-name* 引用利用给定类型实参构造的该类型。
    - 否则，如果出现 *namespace-or-type-name* 的位置包含在 `N` 的命名空间声明中：
      - 如果 `K` 为零，并且该命名空间声明包含一个将名称 `I` 与一个导入的命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *namespace-or-type-name* 引用该命名空间或类型。
      - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间恰好包含一个具有名称 `I` 且有 `K` 个类型形参的类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。
      - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间包含多个具有名称 `I` 且有 `K` 个类型形参的类型，则 *namespace-or-type-name* 是不明确的，并将导致发生错误。
  - 否则，*namespace-or-type-name* 未定义，并将导致发生编译时错误。

- 否则，*namespace-or-type-name* 的形式为 N.I 或 N.I<A1, …, AK>。N 首先被解析为 *namespace-or-type-name*。如果对 N 的解析不成功，则发生编译时错误。否则，N.I 或 N.I<A1, …, AK> 按如下方式进行解析：
  - 如果 K 为零，N 引用一个命名空间，并且 N 包含名为 I 的嵌套命名空间，则 *namespace-or-type-name* 引用该嵌套命名空间。
  - 否则，如果 N 引用一个命名空间，并且 N 包含一个具有名称 I 且有 K 个类型形参的可访问类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。
  - 否则，如果 N 引用一个（可能是构造的）类或结构类型，并且 N 包含一个具有名称 I 且有 K 个类型形参的嵌套可访问类型，则 *namespace-or-type-name* 引用利用给定类型实参构造的该类型。如果存在多个这样的类型，则选择在派生程度最大的类型中声明的类型。
  - 否则，N.I 是无效的 *namespace-or-type-name*，并将发生编译时错误。

如果下列条件成立，则允许 *namespace-or-type-name* 引用静态类（第 25.2 节）

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

20.9.2 成员查找

下列内容取代第 7.3 节：

成员查找是用于确定名称在类型上下文中的含义的过程。成员查找可以作为表达式中计算 *simple-name*（第 20.9.5 节）或 *member-access*（第 20.9.6 节）的过程的一部分进行。

成员查找不仅考虑成员的名称，而且考虑该成员具有的类型形参的数目以及该成员是否可访问。对成员查找来说，泛型方法和嵌套泛型类型具有的类型形参数目就是在它们各自的声明中所指定的数目，其他所有成员则具有零个类型形参。

类型 T 中的具有 K 个类型形参的名称 N 的成员查找过程如下：

- 首先确定名为 N 的可访问成员的集合：
  - 如果 T 是类型形参，则该集合是被指定为 T 的主要约束和次要约束（第 20.7 节）的每个类型中名为 N 的可访问成员的集合与 object 中名为 N 的可访问成员的集合的并集。
  - 否则，该集合由 T 中所有名为 N 的可访问（第 3.5 节）成员（包括继承的成员）和 object 中名为 N 的可访问成员构成。如果 T 为构造类型，则按照第 20.5.4 节中的描述通过替换类型实参获得成员集合。包含 override 修饰符的成员不包括在此集合中。
- 下一步，如果 K 为零，则移除声明中包含类型形参的所有嵌套类型。如果 K 不为零，则移除所有具有不同数目的类型形参的成员。注意，当 K 为零时，将不会移除具有类型形参的方法，因为类型推断过程（第 20.6.4 节）也许能够推断类型实参。
- 然后，从该集合中移除被其他成员隐藏的成员。对于该集合中的每个成员 S.M（其中 S 是声明了成员 M 的类型），应用下面的规则：
  - 如果 M 是一个常量、字段、属性、事件或枚举成员，则从该集合中移除在 S 的基类型中声明的所有成员。

  - 如果 M 是一个类型声明，则从该集合中移除在 S 的基类型中声明的所有非类型，并从该集合中移除与在 S 的基类型中声明的 M 具有相同数目的类型形参的所有类型声明。
  - 如果 M 是方法，则从该集合移除在 S 的基类型中声明的所有非方法成员。
- 然后，从该集合中移除被类成员隐藏的接口成员。仅当 T 为类型形参，并且 T 同时具有除 object 以外的有效基类和非空有效接口集（第 20.7 节）时，此步骤才会产生效果。对于该集合中的每个成员 S.M（其中 S 是声明了成员 M 的类型），如果 S 是除 object 以外的类声明，则应用下面的规则：
  - 如果 M 是一个常量、字段、属性、事件、枚举成员或类型声明，则从该集合中移除在接口声明中声明的所有成员。
  - 如果 M 是一个方法，则从该集合中移除在接口声明中声明的所有非方法成员，并从该集合中移除与在接口声明中声明的 M 具有相同签名的所有方法。
- 最后，在移除隐藏成员之后，按下述规则确定查找的结果：
  - 如果该集合由单个非方法成员组成，则此成员即为查找的结果。
  - 否则，如果该集合只包含方法，则这组方法为查找的结果。
  - 否则，该查找是不明确的，将会发生编译时错误。

对于非类型形参和接口的类型中的成员查找，以及严格单一继承的接口（继承链中的每个接口都只有零个或一个直接基接口）中的成员查找，这些查找规则的效果就相当于派生成员隐藏具有相同名称或签名的基成员。这种单一继承查找从来不会是多义的。有关多重继承接口中的成员查找可能引起的多义性的介绍详见第 13.2.5 节。

20.9.3 适用函数成员

下面这一段已从第 7.4.2.1 节中移除：

- 如果声明函数成员的类、结构或接口已经包含另一个与展开形式具有相同签名的适用函数成员，则该展开形式不适用。

因此，即使存在具有相同签名的已声明的成员，函数成员的展开形式可能仍然适用。潜在的多义性将通过下一小节中描述的附加规则来解决。

20.9.4 更好的函数成员

下列内容取代第 7.4.2.2 节：

给定一个带实参类型序列 $\{A_1, A_2, \ldots, A_N\}$ 的实参列表 A 和带形参类型 $\{P_1, P_2, \ldots, P_N\}$ 和 $\{Q_1, Q_2, \ldots, Q_N\}$ 的两个适用函数成员 M_P 和 M_Q，在进行展开和类型实参替换之后，如果下列条件成立，则 M_P 被定义为比 M_Q 更好的函数成员 ***(better function member)***：

- 对于每个实参，从 A_X 到 P_X 的隐式转换不会比从 A_X 到 Q_X 的隐式转换差，并且
- 至少对于一个实参，从 A_X 到 P_X 的转换比从 A_X 到 Q_X 的转换更好。

在形参类型序列 $\{P_1, P_2, \ldots, P_N\}$和 $\{Q_1, Q_2, \ldots, Q_N\}$ 完全相同的情况下，则应用下列附加规则以便确定更好的函数成员：

- 如果 M_P 是非泛型方法而 M_Q 是泛型方法，则 M_P 比 M_Q 好。

- 否则，如果 M_P 在正常形式下适用，M_Q 有一个 params 数组并且仅在其展开形式下适用，则 M_P 比 M_Q 好。
- 否则，如果 M_P 具有比 M_Q 更少的已声明形参，则 M_P 比 M_Q 好。如果两个方法都有 params 数组，并且都仅在它们的展开形式下适用，就可能出现这种情况。
- 否则，如果 M_P 具有比 M_Q 更明确的形参类型，则 M_P 比 M_Q 好。假设 $\{R_1, R_2, \ldots, R_N\}$ 和 $\{S_1, S_2, \ldots, S_N\}$ 表示 M_P 和 M_Q 的未实例化和未展开的形参类型。如果对于每个形参，R_X 都不比 S_X 更不明确，并且至少对于一个形参，R_X 比 S_X 更明确，则 M_P 的形参类型比 M_Q 的形参类型更明确：
  - 类型形参不如非类型形参明确。
  - 递归地，如果某个构造类型至少有一个类型实参更明确，并且没有类型实参比另一个构造类型（两者具有相同数目的类型实参）中的对应类型实参更不明确，则某个构造类型比另一个构造类型更明确。
  - 如果一个数组类型的元素类型比另一个数组类型的元素类型更明确，则第一个数组类型比第二个数组类型（具有相同的维数）更明确。
- 否则，两个都不是更好的方法。

20.9.5 简单名称

下列内容取代第 7.5.2 节：

simple-name 由一个标识符后跟可选的类型实参列表构成：

simple-name:
identifier *type-argument-list*$_{opt}$

simple-name 的形式为 I 或 I<A_1, ..., A_K>，其中 I 是单个标识符，<A_1, ..., A_K>是可选的 *type-argument-list*。当未指定 *type-argument-list* 时，可将 K 视为零。*simple-name* 的计算和分类方式如下：

- 如果 K 为零，*simple-name* 在一个 *block* 内出现，并且该 *block*（或包容 *block*）的局部变量声明空间（第 3.3 节）包含一个名为 I 的局部变量、形参或常量，则 *simple-name* 引用该局部变量、形参或常量，并被归类为变量或值。
- 如果 K 为零，并且 *simple-name* 出现在泛型方法声明体中，如果该声明包含名为 I 的类型形参，则 *simple-name* 引用该类型形参。
- 否则，对于每个实例类型 T（第 20.1.2 节），从直接的包容类型声明的实例类型开始，对每个包容类或结构声明（如果有）的实例类型进行如下过程：
  - 如果 K 为零，并且 T 的声明包含名为 I 的类型形参，则 *simple-name* 引用该类型形参。
  - 否则，如果在 T 中对具有 K 个类型实参的 I 进行成员查找（第 20.9.2 节）得到匹配项：
    - 如果 T 为直接包容类或结构类型的实例类型，并且该查找标识了一个或多个方法，则结果是一个具有 this 的关联实例表达式的方法组。如果指定了类型实参列表，则将在调用泛型方法时使用它（第 20.6.3 节）。
    - 否则，如果 T 为直接包容类或结构类型的实例类型，如果查找标识出一个实例成员，并且引用发生在实例构造函数、实例方法或实例访问器的 *block* 内，则结果与 this.I 形式的成员访问（第 20.9.6 节）相同。仅当 K 为零时才会发生这种情况。

  - 否则，结果与 `T.I` 或 `T.I<A1, ..., AK>` 形式的成员访问（第 20.9.6 节）相同。在此情况下，*simple-name* 引用实例成员将发生编译时错误。

- 否则，对于每个命名空间 `N`，从出现 *simple-name* 的命名空间开始，继续到每个包容命名空间（如果有），到全局命名空间结束，对下列步骤进行计算，直到找到实体：
  - 如果 `K` 为零，并且 `I` 为 `N` 中的命名空间的名称，则：
    - 如果出现 *simple-name* 的位置包含在 `N` 的命名空间声明中，并且该命名空间声明包含将名称 `I` 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *simple-name* 是不明确的，并将发生编译时错误。
    - 否则，*simple-name* 引用 `N` 中名为 `I` 的命名空间。
  - 否则，如果 `N` 包含一个具有名称 `I` 且有 `K` 个类型形参的可访问类型，则：
    - 如果 `K` 为零，并且出现 *simple-name* 的位置包含在 `N` 的命名空间声明中，并且该命名空间声明包含将名称 `I` 与某个命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *simple-name* 是不明确的，并将发生编译时错误。
    - 否则，*namespace-or-type-name* 引用利用给定类型实参构造的该类型。
  - 否则，如果出现 *simple-name* 的位置包含在 `N` 的命名空间声明中：
    - 如果 `K` 为零，并且该命名空间声明包含一个将名称 `I` 与一个导入的命名空间或类型关联的 *extern-alias-directive* 或 *using-alias-directive*，则 *simple-name* 引用该命名空间或类型。
    - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间恰好包含一个具有名称 `I` 且有 `K` 个类型形参的类型，则 *simple-name* 引用利用给定类型实参构造的该类型。
    - 否则，如果该命名空间声明的 *using-namespace-directive* 导入的命名空间包含多个具有名称 `I` 且有 `K` 个类型形参的类型，则 *simple-name* 是不明确的，并将导致发生错误。

  注意这整个步骤与 *namespace-or-type-name*（第 3.8 和 20.9.1 节）的处理中相应的步骤完全对应。

- 否则，*simple-name* 未定义，将发生编译时错误。

20.9.6 成员访问

下列内容取代第 7.5.4 节：

member-access 的组成部分包括：一个 *primary-expression*、一个 *predefined-type* 或 *qualified-alias-member*，后跟一个“`.`”标记，一个 *identifier* 和一个 *type-argument-list*（可选）。

member-access:
primary-expression `.` *identifier* *type-argument-list*$_{opt}$
predefined-type `.` *identifier* *type-argument-list*$_{opt}$
qualified-alias-member `.` *identifier*

predefined-type: 下列之一

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

qualified-alias-member 产生式是在第 25.3.1 节中定义的。

member-access 的形式为 `E.I` 或 `E.I<A1, ..., AK>`，其中 `E` 是基本表达式，`I` 是单个标识符，`<A1, ..., AK>` 是可选的 *type-argument-list*。当未指定 *type-argument-list* 时，可将 `K` 视为零。*member-access* 的计算和分类方式如下：

- 如果 `K` 为零，`E` 是命名空间，并且 `E` 包含名为 `I` 的嵌套命名空间，则结果为该命名空间。
- 否则，如果 `E` 为命名空间，并且 `E` 包含具有名称为 `I` 且有 `K` 个类型形参的可访问类型，则结果为利用给定类型实参构造的该类型。
- 如果 `E` 是一个 *predefined-type* 或一个被归类为类型的 *primary-expression*，`E` 不是类型形参，并且在 `E` 中对具有 `K` 个类型实参的 `I` 进行成员查找（第 20.9.2 节）得到匹配项，则 `E.I` 的计算和分类方式如下：
  - 如果 `I` 标识一个类型，则结果为使用给定类型实参构造的该类型。
  - 如果 `I` 标识一个或多个方法，则结果为没有关联的实例表达式的方法组。如果指定了类型实参列表，则会在调用泛型方法时使用它（第 20.6.3 节）。
  - 如果 `I` 标识一个 `static` 属性，则结果为没有关联的实例表达式的属性访问。
  - 如果 `I` 标识一个 `static` 字段，则：
    - 如果该字段为 `readonly`，并且引用发生在声明该字段的类或结构的静态构造函数外，则结果为一个值，即 `E` 中静态字段 `I` 的值。
    - 否则，结果为一个变量，即 `E` 中的静态字段 `I`。
  - 如果 `I` 标识一个 `static` 事件，则：
    - 如果引用发生在声明该事件的类或结构内，并且该事件不是用 *event-accessor-declarations*（第 10.7 节）声明的，则完全将 `I` 视为静态字段来处理 `E.I`。
    - 否则，结果为没有关联的实例表达式的事件访问。
  - 如果 `I` 标识一个常量，则结果为一个值，即该常量的值。
  - 如果 `I` 标识枚举成员，则结果为一个值，即该枚举成员的值。
  - 否则，`E.I` 是无效成员引用，并且发生编译时错误。
- 如果 `E` 是类型为 `T` 的属性访问、索引器访问、变量或值，并且在 `T` 中对具有 `K` 个类型实参的 `I` 进行成员查找（第 20.9.2 节）时得到匹配项，则 `E.I` 的计算和分类方式如下：
  - 首先，如果 `E` 为属性访问或索引器访问，则获取该属性访问或索引器访问的值（第 7.1.1 节），并将 `E` 重新归类为值。
  - 如果 `I` 标识一个或多个方法，则结果为具有 `E` 的关联实例表达式的方法组。如果指定了类型实参列表，则会在调用泛型方法时使用它（第 20.6.3 节）。
  - 如果 `I` 标识实例属性，则结果为具有 `E` 的关联实例表达式的属性访问。
  - 如果 `T` 为 *class-type* 并且 `I` 标识该 *class-type* 的一个实例字段，则：
    - 如果 `E` 的值为 `null`，则引发 `System.NullReferenceException`。
    - 否则，如果该字段为 `readonly`，并且引用发生在声明该字段的类的实例构造函数外，则结果为一个值，即 `E` 引用的对象中字段 `I` 的值。

    - 否则，结果为一个变量，即 E 引用的对象中的字段 I。
  - 如果 T 为 *struct-type* 并且 I 标识该 *struct-type* 的实例字段，则：
    - 如果 E 为值，或者如果该字段为 readonly，并且引用发生在声明该字段的结构的实例构造函数外，则结果为一个值，即 E 给定的结构实例中字段 I 的值。
    - 否则，结果为一个变量，即 E 给定的结构实例中的字段 I。
  - 如果 I 标识实例事件，则：
    - 如果引用发生在声明该事件的类或结构内，并且该事件不是用 *event-accessor-declarations*（第 10.7 节）声明的，则完全将 I 视为实例字段来处理 E.I。
    - 否则，结果为具有 E 的关联实例表达式的事件访问。
- 否则，E.I 是无效成员引用，并且发生编译时错误。

20.9.7 方法调用

下面的内容取代第 7.5.5.1 节中描述方法调用的编译时处理的部分：

M(A) 形式（其中 M 是方法组并且可能包括 *type-argument-list*，A 是可选的 *argument-list*）的方法调用的编译时处理包括以下步骤：

- 构造方法调用的候选方法集。对于与方法组 M 关联的每个方法 F：
  - 如果 F 是非泛型的，则在满足以下条件时，F 是候选方法：
    - M 没有类型实参列表，并且
    - 对 A 来说，F 是适用的（第 7.4.2.1 节）。
  - 如果 F 是泛型的，并且 M 没有类型实参列表，则在满足以下条件时，F 是候选方法：
    - 类型推断（第 20.6.4 节）成功，为该调用推断出一个类型实参列表，并且
    - 一旦使用推断出的类型实参替换对应的方法类型形参，则 F 的形参列表中的所有构造类型都满足它们的约束（第 20.7.1 节），并且对 A 来说，F 的形参列表是适用的（第 7.4.2.1 节）。
  - 如果 F 是泛型的，并且 M 包含类型实参列表，则在满足以下条件时，F 是候选方法：
    - F 具有的方法类型形参数目与类型实参列表中提供的数目相同，并且
    - 一旦使用类型实参替换对应的方法类型形参，则 F 的形参列表中的所有构造类型都满足它们的约束（第 20.7.1 节），并且对 A 来说，F 的形参列表是适用的（第 7.4.2.1 节）。
- 候选方法集被减少到仅包含派生程度最大的类型中的方法：对于该集合中的每个方法 C.F（其中 C 是声明了方法 F 的类型），将从该集合中移除在 C 的基类型中声明的所有方法。此外，如果 C 是 object 以外的类类型，则从该集合中移除在接口类型中声明的所有方法。（仅当该方法组是具有除 object 以外的有效基类和非空有效接口集的类型形参上的成员查找的结果时，后一条规则才有效。）
- 如果结果候选方法集为空，则不存在适用的方法，并发生编译时错误。如果候选方法并非全都在同一类型中声明，则该方法调用是多义的，并发生编译时错误。（只有对于具有多个直接基接口的接口中的方法的调用，如第 13.2.5 节所述，或者对于类型形参上的方法的调用，后一种情况才会发生。）

- 候选方法集的最佳方法是使用第 7.4.2 节的重载解析规则确定的。如果无法确定单个最佳方法，则该方法调用是多义的，并发生编译时错误。在执行重载解析时，将在使用类型实参（提供或推断出的）替换对应的方法类型形参之后考虑泛型方法的参数。
- 所选最佳方法的最终验证按如下方式执行：
  - 该方法在方法组的上下文中进行验证：如果该最佳方法是静态方法，则方法组必须是从 *simple-name* 或通过某个类型从 *member-access* 产生的。如果该最佳方法为实例方法，则方法组必须是从 *simple-name*、通过某个变量或值从 *member-access* 或从 *base-access* 产生的。如果两个要求都不满足，则发生编译时错误。
  - 如果该最佳方法是泛型方法，则根据泛型方法上声明的约束（第 20.7.1 节）检查类型实参（提供或推断出的）。如果任何类型实参不满足类型形参上的对应约束，则会发生编译时错误。

通过以上步骤在编译时选定并验证方法之后，将根据第 7.4.3 节中描述的函数成员调用规则处理实际的运行时调用。

20.10 右移语法变化

用于泛型的语法使用 < 和 > 字符分隔类型形参和类型实参（类似于 C++ 中用于模板的语法）。构造类型有时会嵌套（例如在 `List<Nullable<int>>` 中），但是这样的构造存在细微的语法问题：词法语法将把此构造中的最后两个字符组合为 >>（右移运算符），而不是产生句法语法将需要的两个 > 标记。虽然一种可能的解决办法是在两个 > 字符之间加入一个空格，但是这样难处理也容易混淆，并且无论如何也没有增加程序的清晰性。

为了允许这些自然结构，以及为了维持简单的词法语法，>> 和 >>= 标记已从词法语法中移除，并代之以 *right-shift* 和 *right-shift-assignment* 产生式。（还要注意，词法语法增添了两个新的标记 ?? 和 ::，它们的用途分别在第 24.3.9 节和第 25.3 节中描述。）

operator-or-punctuator: 下列之一

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

right-shift:

```
> >
```

right-shift-assignment:

```
> >=
```

与句法语法中的其他产生式不同，*right-shift* 和 *right-shift-assignment* 产生式中的标记之间不允许有任何种类的字符（即使空白也不允许）。

下面的产生式被修改为使用 *right-shift* 和 *right-shift-assignment*：

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression *right-shift* *additive-expression*

assignment-operator:
=
+=
-=
*=
/=
%=
&=
|=
^=
<<=
right-shift-assignment

overloadable-binary-operator:
+
-
*
/
%
&
|
^
<<
right-shift
==
!=
>
<
>=
<=

21. 匿名方法

21.1 匿名方法表达式

anonymous-method-expression 定义匿名方法 ***(anonymous method)***，将对该表达式进行计算，得到引用该方法的具体值：

primary-no-array-creation-expression:
　　...
　　anonymous-method-expression

anonymous-method-expression:
　　`delegate` *anonymous-method-signature*$_{opt}$ *block*

anonymous-method-signature:
　　`(` *anonymous-method-parameter-list*$_{opt}$ `)`

anonymous-method-parameter-list:
　　anonymous-method-parameter
　　anonymous-method-parameter-list `,` *anonymous-method-parameter*

anonymous-method-parameter:
　　parameter-modifier$_{opt}$ *type* *identifier*

anonymous-method-expression 属于一种具有特殊转换规则的值（第 21.3 节）。这个值没有类型，但是可隐式转换为与之兼容的委托类型。

anonymous-method-expression 为参数、局部变量和常量定义了一个新的声明空间，并为标签（第 3.3 节）定义了一个新的声明空间。

21.2 匿名方法签名

可选的 *anonymous-method-signature* 为匿名方法定义了形参的名称和类型。匿名方法参数的作用域为 *block*。如果匿名方法的参数名与作用域包含该 *anonymous-method-expression* 的本地变量、本地常量或参数的名称相匹配，则为编译时错误。

如果 *anonymous-method-expression* 有一个 *anonymous-method-signature*，则兼容的委托类型集合将局限于那些具有相同参数类型和修饰符且顺序相同的委托类型（第 21.3 节）。与方法组转换（第 21.9 节）不同，匿名方法参数类型的逆变不受支持。如果 *anonymous-method-expression* 没有 *anonymous-method-signature*，则兼容的委托类型集将局限于那些没有 `out` 参数的委托类型。

注意，*anonymous-method-signature* 不可包含属性或参数数组。不过，*anonymous-method-signature* 可与参数列表中包含参数数组的委托类型兼容。

21.3 匿名方法转换

anonymous-method-expression 属于无类型的值。*anonymous-method-expression* 可用在 *delegate-creation-expression*（第 21.10 节）中。*anonymous-method-expression* 的所有其他有效用法均取决于此处定义的隐式转换。

存在从 *anonymous-method-expression* 到任何兼容 ***(compatible)*** 委托类型的隐式转换（第 6.1 节）。如果 D 为委托类型，并且 A 为 *anonymous-method-expression*，则当且仅当满足下面两个条件时，D 与 A 兼容。

- 首先，D 的参数类型必须与 A 兼容：
  - 如果 A 不包含 *anonymous-method-signature*，则 D 可以有零个或更多个任意类型的参数，只要 D 的所有参数都没有 out 参数修饰符。
  - 如果 A 有一个 *anonymous-method-signature*，则 D 必须有相同数目的参数，且 A 的每个参数都必须与 D 的对应参数的类型相同，并且 A 的每个参数是否有 ref 或 out 修饰符也必须与 D 的对应参数匹配。D 的最后一个参数是否为 *parameter-array* 与 A 和 D 的兼容性无关。
- 其次，D 的返回类型必须与 A 兼容。对于这些规则，均认为 A 不包含任何其他匿名方法的 *block*。
  - 如果 D 声明了 void 返回类型，则 A 中包含的任何 return 语句都不能指定表达式。
  - 如果 D 声明了 R 返回类型，则 A 中包含的任何 return 语句都必须指定一个可隐式转换（第 6.1 节）为 R 的表达式。而且，A 的 *block* 的结束点必须是不可到达的。

除了隐式转换为兼容的委托类型外，*anonymous-method-expression* 不能转换为任何其他类型，即使是 object 类型。

下面的示例演示了这些规则：

```
delegate void D(int x);

D d1 = delegate { };                    // Ok
D d2 = delegate() { };                  // Error, signature mismatch
D d3 = delegate(long x) { };            // Error, signature mismatch
D d4 = delegate(int x) { };             // Ok
D d5 = delegate(int x) { return; };     // Ok
D d6 = delegate(int x) { return x; };   // Error, return type mismatch

delegate void E(out int x);

E e1 = delegate { };                    // Error, E has an out parameter
E e2 = delegate(out int x) { x = 1; }; // Ok
E e3 = delegate(ref int x) { x = 1; }; // Error, signature mismatch

delegate int P(params int[] a);

P p1 = delegate { };                    // Error, end of block reachable
P p2 = delegate { return; };            // Error, return type mismatch
P p3 = delegate { return 1; };          // Ok
P p4 = delegate { return "Hello"; };    // Error, return type mismatch
P p5 = delegate(int[] a) {              // Ok
   return a[0];
};
P p6 = delegate(params int[] a) {       // Error, params modifier
   return a[0];
};
P p7 = delegate(int[] a) {              // Error, return type mismatch
   if (a.Length > 0) return a[0];
   return "Hello";
};

delegate object Q(params int[] a);

Q q1 = delegate(int[] a) {              // Ok
   if (a.Length > 0) return a[0];
   return "Hello";
};
```

21.4 匿名方法块

anonymous-method-expression 的 *block* 遵循以下规则：

- 如果匿名方法包括签名，则签名中指定的参数可在 *block* 中使用。如果匿名方法没有签名，则它可转换为带有参数的委托类型（第 21.3 节），但是这些参数在 *block* 中不可访问。
- 除了在最近的包容匿名方法签名（如果存在）中指定的 `ref` 或 `out` 参数外，*block* 访问其他 `ref` 或 `out` 参数将导致编译时错误。
- 当 `this` 的类型为结构类型时，*block* 访问 `this` 将导致编译时错误。无论访问是显式（如 `this.x`）还是隐式（如 `x`，其中 `x` 是该结构的实例成员），此规则都成立。此规则仅仅是禁止此类访问，但是不影响成员查找是否能找到该结构的成员。
- *block* 可访问匿名方法的外层变量（第 21.5 节）。访问外层变量将引用计算 *anonymous-method-expression* 时处于活动状态的变量实例（第 21.6 节）。
- *block* 包含目标在 *block* 之外或所包含的匿名方法的 *block* 之内的 `goto` 语句、`break` 语句或 `continue` 语句时将导致编译时错误。
- *block* 内的 `return` 语句从最近的包容匿名方法调用中返回控制，而不是从包容函数成员中返回。`return` 语句中指定的表达式必须与最近的包容 *anonymous-method-expression* 所转换为的委托类型兼容（第 21.3 节）。

除了通过计算和调用 *anonymous-method-expression* 之外，并未明确指定是否还有任何其他方法可执行匿名方法的 *block*。具体而言，编译器可选择通过合成一个或多个命名方法或类型来实现匿名方法。任何此类合成元素的名称必须位于保留给编译器使用的空间中（即，名称必须包含两个连续的下划线字符）。

21.5 外层变量

作用域包括 *anonymous-method-expression* 的任何局部变量、值参数或参数数组都称为该 *anonymous-method-expression* 的外层变量 ***(outer variable)***。在类的实例函数成员中，`this` 值被视为值参数，并且是该函数成员内包含的所有 *anonymous-method-expression* 的外层变量。

21.5.1 捕获的外层变量

当某个外层变量由匿名方法引用时，称为该外层变量被匿名方法捕获 ***(captured)***。通常，局部变量的生存期仅限于该变量所关联的块或语句的执行期（第 5.1.7 节）。但是，被捕获的外层变量的生存期将至少延长至引用匿名方法的委托可以被垃圾回收为止。

在下面的示例中

```
using System;

delegate int D();

class Test
{
   static D F() {
      int x = 0;
      D result = delegate { return ++x; }
      return result;
   }
```

```
   static void Main() {
      D d = F();
      Console.WriteLine(d());
      Console.WriteLine(d());
      Console.WriteLine(d());
   }
}
```

局部变量 x 由匿名方法捕获，并且 x 的生存期至少延长至从 F 返回的委托可以被垃圾回收为止（这要到程序的最后才会发生）。由于对匿名方法的每次调用都对同一个 x 实例进行操作，因此该示例的输出为：

```
1
2
3
```

当局部变量或值参数由匿名方法捕获时，该局部变量或参数不再被视作固定变量（第 18.3 节），而是被视作可移动变量。因此，任何使用被捕获外层变量的地址的 unsafe 代码必须首先使用 fixed 语句固定该变量。

21.5.2 局部变量实例化

当执行过程进入局部变量作用域时，该变量视为被实例化 ***(instantiated)***。例如，当调用下面的方法时，局部变量 x 被实例化和初始化三次，每一次对应于循环的一轮迭代。

```
static void F() {
   for (int i = 0; i < 3; i++) {
      int x = i * 2 + 1;
      ...
   }
}
```

但是，如果将 x 声明移到循环之外，则 x 将只实例化一次：

```
static void F() {
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      ...
   }
}
```

通常，我们无法确切知道局部变量实例化的频率，因为实例化的生存期不是连续的，有可能每次实例化都只使用同一存储位置。但是，当匿名方法捕获到局部变量时，实例化的效果就会变得很明显。

下面的示例

```
using System;

delegate void D();

class Test
{
   static D[] F() {
      D[] result = new D[3];
      for (int i = 0; i < 3; i++) {
         int x = i * 2 + 1;
         result[i] = delegate { Console.WriteLine(x); };
      }
      return result;
   }
```

```
   static void Main() {
      foreach (D d in F()) d();
   }
}
```

产生下列输出：

```
1
3
5
```

但是，当 x 的声明移到循环外时：

```
static D[] F() {
   D[] result = new D[3];
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      result[i] = delegate { Console.WriteLine(x); };
   }
   return result;
}
```

输出为：

```
5
5
5
```

注意上面第二个 F 版本中创建的三个委托，根据相等运算符，它们将相等（第 21.7 节）。此外请注意，允许（但不要求）编译器将三次实例化优化为一个委托实例（第 21.6 节）。

匿名方法委托可共享某些捕获的变量，但是又具有其他变量的不同实例。例如，如果 F 更改为

```
static D[] F() {
   D[] result = new D[3];
   int x = 0;
   for (int i = 0; i < 3; i++) {
      int y = 0;
      result[i] = delegate { Console.WriteLine("{0} {1}", ++x, ++y); };
   }
   return result;
}
```

则这三个委托将捕获 x 的同一个实例，但捕获 y 的不同实例，并且输出为：

```
1 1
2 1
3 1
```

不同的匿名方法可捕获一个外层变量的同一个实例。在下面的示例中：

```
using System;
delegate void Setter(int value);
delegate int Getter();
```

```
class Test
{
   static void Main() {
      int x = 0;
      Setter s = delegate(int value) { x = value; };
      Getter g = delegate { return x; };
      s(5);
      Console.WriteLine(g());
      s(10);
      Console.WriteLine(g());
   }
}
```

两个匿名方法捕获局部变量 x 的同一个实例，并且它们因此可通过该变量进行“通信”。该示例的输出为：

```
5
10
```

21.6 匿名方法计算

anonymous-method-expression 的运行时计算生成一个委托实例，该实例引用匿名方法以及所捕获的、在计算时处于活动状态的外层变量的集合（可能为空）。当调用从 *anonymous-method-expression* 生成的委托时，将执行匿名方法体。使用委托引用的被捕获外层变量集合执行方法体中的代码。

从 *anonymous-method-expression* 生成的委托的调用列表只包含一个项。该委托的确切目标对象和目标方法并未指定。具体而言，没有指定该委托的目标对象是 null、包容函数成员的 this 值，还是某个其他对象。

允许（但不要求）以相同的被捕获外层变量实例集合（可能为空集）计算语义上相同的 *anonymous-method-expression* 以返回相同的委托实例。此处所用的术语“语义上相同”表示，无论何种情况，只要给定相同的参数，匿名方法的执行都将产生相同的结果。此规则允许优化如下面这样的代码。

```
delegate double Function(double x);

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

   static void F(double[] a, double[] b) {
      a = Apply(a, delegate(double x) { return Math.Sin(x); });
      b = Apply(b, delegate(double y) { return Math.Sin(y); });
      ...
   }
}
```

由于两个匿名方法委托有着相同的被捕获外层变量集合（空集），并且这两个匿名方法语义上相同，所以允许编译器使这两个委托引用同一个目标方法。实际上，允许编译器从这两个匿名方法表达式返回同一个委托实例。

21.7 委托实例相等性

下面的规则决定由相等运算符（第 7.9.8 节）和匿名方法委托实例的 Object.Equals 方法所生成的结果：

- 以相同的被捕获外层变量实例的集合（可能为空集）计算语义上相同的 *anonymous-method-expression* 时，允许（但不是要求）所生成的委托实例相等。
- 计算语义上不同的 *anonymous-method-expression* 或者使用不同的被捕获外层变量实例的集合进行计算时，所生成的委托实例永远不等。

21.8 明确赋值

匿名方法的参数的明确赋值状态与命名方法的参数的明确赋值状态相同。即，引用参数和值参数初始是明确赋值的，而输出参数初始时未赋值。此外，在匿名方法正常返回之前，输出参数必须明确赋值（第 5.1.6 节）。

当控制转移到 *anonymous-method-expression* 的 *block* 时，外层变量 *v* 的明确赋值状态与 *v* 在 *anonymous-method-expression* 之前的明确赋值状态相同。即，外层变量的明确赋值是从 *anonymous-method-expression* 的上下文中继承的。在 *anonymous-method-expression* 的 *block* 内，明确赋值与在常规程序块（第 5.3.3 节）内的方式相同。

变量 *v* 在 *anonymous-method-expression* 之后的明确赋值状态与其在 *anonymous-method-expression* 之前的明确赋值状态相同。

下面的示例

```
delegate bool Filter(int i);

void F() {
   int max;

   // Error, max is not definitely assigned
   Filter f = delegate(int n) { return n < max; }

   max = 5;
   DoWork(f);
}
```

生成一个编译时错误，因为在声明匿名方法的位置，max 尚未明确赋值。下面的示例

```
delegate void D();

void F() {
   int n;
   D d = delegate { n = 1; };

   d();

   // Error, n is not definitely assigned
   Console.WriteLine(n);
}
```

也生成一个编译时错误，因为匿名方法中对 n 的赋值并不影响匿名方法之外的 n 的明确赋值状态。

21.9 方法组转换

与第 21.3 节中所述的隐式匿名方法转换类似，也存在从方法组（第 7.1 节）到兼容的委托类型的隐式转换（第 6.1 节）。对于给定的委托类型 D 和可归类为方法组的表达式 E，如果下述条件成立则存在从 E 到 D 的隐式转换：E 至少包含一个方法，该方法能够以其正常形式（第 7.4.2.1 节）应用于类型和修饰符与 D 的参数类型和修饰符匹配的实参列表。

从方法组 E 转换到委托类型 D 的编译时应用在下面的部分中描述。注意，存在从 E 到 D 的隐式转换并不保证该转换的编译时应用会成功和不会出错。

- 对于 E(A) 形式的方法调用（第 20.9.7 节），仅选择一个方法 M，并进行以下修改：
  - D 的参数类型和修饰符（ref 或 out）用作实参列表 A 的实参类型和修饰符。
  - 所考虑的候选方法仅为那些可以其正常形式（第 7.4.2.1 节）加以应用的方法，而不是那些只能以其展开形式应用的方法。
  - 如果第 20.9.7 节的算法产生错误，则会发生编译时错误。否则，该算法将生成一个与 D 有着相同数目参数的最佳方法 M。
- 选定的方法 M 必须与委托类型 D 一致（如下所述），否则将发生编译时错误。
- 如果选定的方法 M 是实例方法，则与 E 关联的实例表达式确定委托的目标对象。
- 转换的结果是类型 D 的值，即一个引用选定方法和目标对象的新创建的委托。

如果下面的所有条件都成立，则方法 M 与委托类型 D 一致 ***(consistent)***：

- D 和 M 有着相同数目的参数，并且 D 中的每一个参数都具有与 M 中的对应参数相同的修饰符。
- 对于每一个值参数（没有 ref 或 out 修饰符的参数），都存在从 D 中的参数类型到 M 中的对应参数类型的标识转换（第 6.1.1 节）或隐式引用转换（第 6.1.4 节）。
- 对于每个 ref 或 out 参数，D 中的参数类型与 M 中的参数类型相同。
- 存在从 M 的返回类型到 D 的返回类型的标识或隐式引用转换。

上述委托一致性规则取代了前面的第 15.1 节中所述的委托兼容性规则，并且比后者更为宽松。

下面的示例演示方法组转换：

```
delegate string D1(object o);
delegate object D2(string s);
delegate string D3(int i);
class Test
{
   static string F(object o) {...}
   static void G() {
      D1 d1 = F;        // Ok
      D2 d2 = F;        // Ok
      D3 d3 = F;        // Error
   }
}
```

对 d1 的赋值隐式将方法组 F 转换为 D1 类型的值。以前，必须编写 new D1(F) 这样的代码来生成委托值。虽然这仍是允许的，但不再是必需的。

对 d2 的赋值演示了委托给具有派生程度较小（逆变）的参数类型和派生程度较大（协变）的返回类型的方法的可能性。从直观上看，这是安全的并且没有开销，因为作为参数和返回值传递的引用仅被视作派生程度较小的类型的引用。

对 d3 的赋值演示了如何允许委托和方法的参数和返回类型仅对引用类型存在不同。

与所有其他隐式和显式转换一样，强制转换运算符可用于显式执行方法组转换。因此，示例

```
object obj = new EventHandler(myDialog.OkClick);
```

可改写为

```
object obj = (EventHandler)myDialog.OkClick;
```

方法组和匿名方法表达式可能影响重载解析，但不会参与类型推断。有关详细信息，请参见第 20.6.4 节。

21.10 委托创建表达式

委托创建表达式（第 7.5.10.3 节）得到了扩展以允许实参为归类为方法组的表达式、归类为匿名方法的表达式，或者为委托类型的值。

new D(E) 形式（其中 D 是 *delegate-type*，E 是 *expression*）的 *delegate-creation-expression* 的编译时处理包括以下步骤：

- 如果 E 为方法组，则必须存在从 E 到 D 的方法组转换（第 21.9 节），并且以与该转换相同的方式处理委托创建表达式。
- 如果 E 为匿名方法，则必须存在从 E 到 D 的匿名方法转换（第 21.3 节），并且以与该转换相同的方式处理委托创建表达式。
- 如果 E 为委托类型的值，则 E 的方法签名必须与 D 一致（第 21.9 节），并且结果为对新创建的 D 类型的委托的引用，该委托引用与 E 相同的调用列表。如果 E 与 D 不一致，则会发生编译时错误。

21.11 实现示例

本节从标准 C# 构造的角度描述可能的匿名方法实现方法。此处所描述的实现基于 Microsoft C# 编译器所使用的相同原理，但决非强制性的实现方式，也不是唯一可能的实现方式。

本节的其余部分提供了多个代码示例，其中包含具有不同特点的匿名方法。对于每个示例，提供了到仅使用标准 C# 构造的代码的相应转换。在这些示例中，假定标识符 D 表示下面的委托类型：

```
public delegate void D();
```

匿名方法的最简单形式是不捕获外层变量的形式：

```
class Test
{
   static void F() {
      D d = delegate { Console.WriteLine("test"); };
   }
}
```

这可转换为引用编译器生成的静态方法的委托实例化，匿名方法的代码位于该静态方法中：

```
class Test
{
   static void F() {
      D d = new D(__Method1);
   }
```

```
   static void __Method1() {
      Console.WriteLine("test");
   }
}
```

在下面的示例中，匿名方法引用 this 的实例成员：

```
class Test
{
   int x;

   void F() {
      D d = delegate { Console.WriteLine(x); };
   }
}
```

这可转换为包含该匿名方法代码的、编译器生成的实例方法：

```
class Test
{
   int x;

   void F() {
      D d = new D(__Method1);
   }

   void __Method1() {
      Console.WriteLine(x);
   }
}
```

在此示例中，匿名方法捕获一个局部变量：

```
class Test
{
   void F() {
      int y = 123;
      D d = delegate { Console.WriteLine(y); };
   }
}
```

局部变量的生存期现在必须至少延长为匿名方法委托的生存期。这可通过将局部变量“提升”为编译器生成的类的字段来实现。局部变量的实例化（第 21.5.2 节）则对应于为编译器生成的类创建实例的过程，而访问局部变量的过程则对应于访问编译器生成的类的实例中的字段。而且，匿名方法变为编译器生成类的实例方法：

```
class Test
{
   void F() {
      __Locals1 __locals1 = new __Locals1();
      __locals1.y = 123;
      D d = new D(__locals1.__Method1);
   }

   class __Locals1
   {
      public int y;

      public void __Method1() {
         Console.WriteLine(y);
      }
   }
}
```

最后，下面的匿名方法捕获 this 以及两个具有不同生存期的局部变量：

```
class Test
{
   int x;

   void F() {
      int y = 123;
      for (int i = 0; i < 10; i++) {
         int z = i * 2;
         D d = delegate { Console.WriteLine(x + y + z); };
      }
   }
}
```

此处，将为捕获局部变量的每一个语句块分别创建一个编译器生成的类，这样不同块中的局部变量可以有独立的生存期。__Locals2（对应于内层语句块的编译器生成类）的实例包含局部变量 z 和引用 __Locals1 的实例的字段。__Locals1（对应于外层语句块的编译器生成类）的实例包含局部变量 y 和引用包容函数成员的 this 的字段。使用这些数据结构，可以通过 __Local2 的实例到达所有被捕获外层变量，并且匿名方法的代码因此可实现为该类的实例方法。

```
class Test
{
   void F() {
      __Locals1 __locals1 = new __Locals1();
      __locals1.__this = this;
      __locals1.y = 123;
      for (int i = 0; i < 10; i++) {
         __Locals2 __locals2 = new __Locals2();
         __locals2.__locals1 = __locals1;
         __locals2.z = i * 2;
         D d = new D(__locals2.__Method1);
      }
   }

   class __Locals1
   {
      public Test __this;
      public int y;
   }

   class __Locals2
   {
      public __Locals1 __locals1;
      public int z;

      public void __Method1() {
         Console.WriteLine(__locals1.__this.x + __locals1.y + z);
      }
   }
}
```

22. 迭代器

22.1 迭代器块

迭代器块是一个产生有序值序列的 *block*（第 8.2 节）。迭代器块与普通语句块的区别在于迭代器块存在一条或多条 `yield` 语句。

- `yield return` 语句产生迭代的下一个值。
- `yield break` 语句指示迭代完成。

迭代器块可用作 *method-body*、*operator-body* 或 *accessor-body*，只要对应的函数成员的返回类型为枚举器接口（第 22.1.1 节）之一或可枚举接口（第 22.1.2 节）之一。

迭代器块并不是 C# 语法中的一个特别的元素。它们受到多方面的限制，并且对于函数成员声明的语义有主要影响；然而，从语法角度来说，它们只是块。

当使用迭代器块实现函数成员时，为该函数成员的形参列表指定任何 `ref` 或 `out` 参数将产生编译时错误。

迭代器块中出现 `return` 语句时也会产生编译时错误（但是允许 `yield return` 语句）。

迭代器块包含不安全的上下文（第 18.1 节）时将导致编译时错误。迭代器块总是定义安全的上下文，即使其定义嵌套在不安全的上下文中也如此。

22.1.1 枚举器接口

枚举器接口 ***(enumerator interface)*** 为非泛型接口 `System.Collections.IEnumerator` 和泛型接口 `System.Collections.Generic.IEnumerator<T>` 的所有实例化。为简洁起见，本章中将这些接口分别表示为 `IEnumerator` 和 `IEnumerator<T>`。

22.1.2 可枚举接口

可枚举接口 ***(enumerable interface)*** 为非泛型接口 `System.Collections.IEnumerable` 和泛型接口 `System.Collections.Generic.IEnumerable<T>` 的所有实例化。为简洁起见，本章中将这些接口分别表示为 `IEnumerable` 和 `IEnumerable<T>`。

22.1.3 产生类型

迭代器块产生一系列值，所有值的类型均相同。此类型称为迭代器块的产生类型 ***(yield type)***。

- 如果迭代器块用于实现返回 `IEnumerator` 或 `IEnumerable` 的函数成员，则其产生类型为 `object`。
- 如果迭代器块用于实现返回 `IEnumerator<T>` 或 `IEnumerable<T>` 的函数成员，则其产生类型为 `T`。

22.1.4 this 访问

在类的实例成员的迭代器块中，表达式 this 归类为一个值。该值的类型就是其所在的类，并且该值是一个引用，它引用的对象就是为其调用该成员的对象。

在结构的实例成员的迭代器块中，表达式 this 归类为一个变量。该变量的类型就是其所在的结构。该变量表示为其调用该成员的结构的副本。结构的实例成员的迭代器块中的 this 变量的行为与结构类型的值参数完全一样。注意，这与非迭代器函数成员体内的行为不同。

22.2 枚举器对象

如果返回枚举器接口类型的函数成员是使用迭代器块实现的，调用该函数成员不会立即执行迭代器块中的代码。而是先创建并返回一个枚举器对象 ***(enumerator object)***。此对象封装了在迭代器块中指定的代码，并且在调用该枚举器对象的 MoveNext 方法时执行该迭代器块中的代码。枚举器对象具有下列特点：

- 它实现了 IEnumerator 和 IEnumerator<T>，其中 T 为迭代器块的产生类型。
- 它实现了 System.IDisposable。
- 它以传递给该函数成员的参数值（如果存在）和实例值的副本进行初始化。
- 它有四种可能的状态：运行前 ***(before)***、运行中 ***(running)***、挂起 ***(suspended)*** 和运行后 ***(after)***，并且初始状态为运行前 ***(before)*** 状态。

枚举器对象通常是编译器生成的枚举器类的一个实例，它封装了迭代器块中的代码，并实现了枚举器接口，但也可能实现其他方法。如果枚举器类由编译器生成，则该类将直接或间接嵌套在包含该函数成员的类中，它将具有私有可访问性，并且它将具有一个供编译器使用的保留名称（第 2.4.2 节）。

枚举器对象可实现除上面指定的那些接口以外的其他接口。

下面的各节将描述由枚举器对象所提供的 IEnumerable 和 IEnumerable<T> 接口实现的 MoveNext、Current 和 Dispose 成员的确切行为。

注意，枚举器对象不支持 IEnumerator.Reset 方法。调用此方法将导致引发 System.NotSupportedException。

22.2.1 MoveNext 方法

枚举器对象的 MoveNext 方法封装了迭代器块的代码。调用 MoveNext 方法将执行迭代器块中的代码，并相应设置枚举器对象的 Current 属性。MoveNext 执行的具体操作取决于调用 MoveNext 时的枚举器对象的状态：

- 如果枚举器对象的状态为运行前 ***(before)***，则调用 MoveNext 会：
  - 将状态更改为运行中 ***(running)***。
  - 将迭代器块的参数（包括 this）初始化为实参值以及初始化该枚举器对象时所保存的实例值。
  - 从头开始执行迭代器块，直到执行被中断（如后文所述）。
- 如果枚举器对象的状态为运行中 ***(running)***，则调用 MoveNext 的结果不确定。
- 如果枚举器对象的状态为挂起 ***(suspended)***，则调用 MoveNext 会：
  - 将状态更改为运行中 ***(running)***。

  - 将所有局部变量和参数（包括 this）的值恢复为迭代器块的执行上次挂起时保存的值。注意，这些变量所引用对象的内容可能自上次调用 MoveNext 之后已经发生更改。
  - 恢复执行紧跟在引起执行挂起的 `yield return` 语句后面的迭代器块，并一直继续，直到执行中断（如后文所述）。
- 如果枚举器对象的状态为执行后 ***(after)***，则调用 `MoveNext` 将返回 `false`。

当 `MoveNext` 执行迭代器块时，可以用四种方法中断执行：通过 `yield return` 语句、通过 `yield break` 语句、遇到迭代器块的结束处，以及通过引发并传播到迭代器块之外的异常。

- 当遇到 `yield return` 语句时（第 22.4 节）：
  - 计算该语句中给出的表达式，隐式转换为产生类型，并赋给枚举器对象的 `Current` 属性。
  - 迭代器体的执行被挂起。所有局部变量和参数（包括 `this`）的值被保存，此 `yield return` 语句的位置也被保存。如果 `yield return` 语句在一个或多个 `try` 块内，则与之关联的 `finally` 块此时不会执行。
  - 枚举器对象的状态更改为挂起 ***(suspended)***。
  - `MoveNext` 方法向其调用方返回 `true`，指示迭代成功前进至下一个值。
- 当遇到 `yield break` 语句时（第 22.4 节）：
  - 如果 `yield break` 语句在一个或多个 `try` 块内，则执行与之关联的 `finally` 块。
  - 枚举器对象的状态更改为运行后 ***(after)***。
  - `MoveNext` 方法向其调用方返回 `false`，指示迭代完成。
- 当遇到迭代器体的结束处时：
  - 枚举器对象的状态更改为运行后 ***(after)***。
  - `MoveNext` 方法向其调用方返回 `false`，指示迭代完成。
- 当引发异常并传播到迭代器块之外时：
  - 通过异常传播机制执行迭代器体内的相应 `finally` 块。
  - 枚举器对象的状态更改为运行后 ***(after)***。
  - 异常继续传播至 `MoveNext` 方法的调用方。

22.2.2 Current 属性

枚举器对象的 `Current` 属性受迭代器块中的 `yield return` 语句的影响。

当枚举器对象处于挂起 ***(suspended)*** 状态时，`Current` 的值为上一次调用 `MoveNext` 时设置的值。当枚举器对象处于运行前 ***(before)***、运行中 ***(running)*** 或运行后 ***(after)*** 状态时，访问 `Current` 的结果不确定。

对于产生类型不是 `object` 的迭代器块，通过枚举器对象的 `IEnumerable` 实现来访问 `Current` 的结果对应于通过枚举器对象的 `IEnumerator<T>` 实现来访问 `Current` 并将该结果强制转换为 `object`。

22.2.3 Dispose 方法

`Dispose` 方法用于通过使枚举器对象变为运行后 ***(after)*** 状态来清除迭代。

- 如果枚举器对象的状态为运行前 ***(before)***，则调用 Dispose 将把状态更改为运行后 ***(after)***。
- 如果枚举器对象的状态为运行中 ***(running)***，则调用 Dispose 的结果不确定。
- 如果枚举器对象的状态为挂起 ***(suspended)***，则调用 Dispose 将：
  - 将状态更改为运行中 ***(running)***。
  - 执行所有 finally 块，就好像最后执行的 yield return 语句为 yield break 语句一样。如果这导致引发异常，并且异常传播到迭代器体之外，则枚举器对象的状态设置为运行后 ***(after)***，并且将异常传播到 Dispose 方法的调用方。
  - 将状态更改为运行后 ***(after)***。
- 如果枚举器对象的状态为运行后 ***(after)***，则调用 Dispose 没有任何作用。

22.3 可枚举对象

如果返回可枚举接口类型的函数成员是使用迭代器块实现的，调用该函数成员不会立即执行迭代器块中的代码。而是先创建并返回一个可枚举对象 ***(enumerable object)***。可枚举对象的 GetEnumerator 方法返回一个封装有迭代器块中指定的代码的枚举器对象，当调用该枚举器对象的 MoveNext 方法时，将执行迭代器块中的代码。可枚举对象具有下列特点：

- 它实现了 IEnumerable 和 IEnumerable<T>，其中 T 为迭代器块的产生类型。
- 它以传递给该函数成员的参数值（如果存在）和实例值的副本进行初始化。

可枚举对象通常是编译器生成的可枚举类的实例，它封装了迭代器块中的代码，并实现了可枚举接口，但也可能实现其他方法。如果可枚举类由编译器生成，则该类将直接或间接嵌套在包含该函数成员的类中，它将具有私有可访问性，并且它将具有供编译器使用的保留名称（第 2.4.2 节）。

可枚举对象可实现除上面指定的那些接口以外的其他接口。具体而言，可枚举对象还可实现 IEnumerator 和 IEnumerator<T>，从而使其既可作为可枚举对象，也可作为枚举器对象。在该类型的实现中，首次调用可枚举对象的 GetEnumerator 方法时，将返回可枚举对象本身。对可枚举对象的 GetEnumerator 的后续调用（如果存在），将返回可枚举对象的副本。因此，每个返回的枚举器都有自己的状态，一个枚举器中的更改不会影响其他枚举器。

22.3.1 GetEnumerator 方法

可枚举对象实现了 IEnumerable 和 IEnumerable<T> 接口的 GetEnumerator 方法。这两种 GetEnumerator 方法的实现是相同的，都是获取并返回一个可用的枚举器对象。枚举器对象是以初始化该可枚举对象时保存的实例值和参数值进行初始化的，此外，枚举器对象函数如第 22.2 节所述。

22.4 yield 语句

yield 语句用在迭代器块中，作用是向枚举器对象产生一个值，或者通知迭代的结束。

embedded-statement:
    ...
    yield-statement

yield-statement:
    yield return *expression* ;
    yield break ;

为了确保与现有程序的兼容性，yield 不是一个保留字，并且 yield 仅在直接位于 return 或 break 关键字之前时才有特殊意义。在其他上下文中，yield 可用作标识符。

yield 语句可出现的位置存在几个限制，如下所述。

- 如果 yield 语句（包括两种形式）出现在 *method-body*、*operator-body* 或 *accessor-body* 之外，则会引起编译时错误。
- 如果 yield 语句（包括两种形式）出现在匿名方法之内，则会引起编译时错误。
- 如果 yield 语句（包括两种形式）出现在 try 语句的 finally 子句之内，则会引起编译时错误。
- 如果 yield return 语句出现在包含 catch 子句的 try 语句内的任何位置，则会引起编译时错误。

下面的示例演示 yield 语句的有效用法和无效用法。

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
   try {
      yield return 1;      // Ok
      yield break;         // Ok
   }
   finally {
      yield return 2;      // Error, yield in finally
      yield break;         // Error, yield in finally
   }

   try {
      yield return 3;      // Error, yield return in try...catch
      yield break;         // Ok
   }
   catch {
      yield return 4;      // Error, yield return in try...catch
      yield break;         // Ok
   }

   D d = delegate {
      yield return 5;      // Error, yield in an anonymous method
   };
}

int MyMethod() {
   yield return 1;         // Error, wrong return type for an iterator block
}
```

yield return 语句中的表达式的类型必须能够隐式转换（第 6.1 节）为迭代器块的产生类型（第 22.1.3 节）。

yield return 语句的执行方式如下：

- 计算该语句中给出的表达式，隐式转换为产生类型，并赋给枚举器对象的 Current 属性。
- 迭代器块的执行被挂起。如果 yield return 语句在一个或多个 try 块内，则与之关联的 finally 块此时不会执行。
- 枚举器对象的 MoveNext 方法向其调用方返回 true，指示枚举器对象成功前进到下一项。

下次调用枚举器对象的 MoveNext 方法时将从上次挂起的地方恢复迭代器块的执行。

yield break 语句的执行方式如下：

- 如果 yield break 语句包含在一个或多个具有关联 finally 块的 try 块中，则控制首先将转移到最内层 try 语句的 finally 块。当（如果）控制到达某个 finally 块的结束点时，控制就转移到下一层 try 语句的 finally 块。此过程不断重复，直到执行完所有层中的 try 语句的 finally 块。
- 控制返回给迭代器块的调用方。这是枚举器对象的 MoveNext 方法或 Dispose 方法。

由于 yield break 语句无条件地将控制转移到别处，因此永远无法到达 yield break 语句的结束点。

22.4.1 明确赋值

对于具有以下形式的 yield return 语句 *stmt*：

```
yield return expr ;
```

- 变量 *v* 在 *expr* 的开头处与在 *stmt* 的开头处具有相同的明确赋值状态。
- 如果变量 *v* 在 *expr* 的结尾处明确赋值，则它在 *stmt* 的结束点也明确赋值；否则，它在 *stmt* 的结束点不明确赋值。

22.5 实现示例

本节从标准 C# 构造的角度描述迭代器可能的实现。此处所描述的实现基于 Microsoft C# 编译器所使用的相同原理，但决非是强制性的实现方式，也不是唯一可能的实现方式。

下面的 Stack<T> 类使用一个迭代器实现其 GetEnumerator 方法。该迭代器以自顶向下的顺序枚举堆栈的元素。

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
   T[] items;
   int count;

   public void Push(T item) {
      if (items == null) {
         items = new T[4];
      }
      else if (items.Length == count) {
         T[] newItems = new T[count * 2];
         Array.Copy(items, 0, newItems, 0, count);
         items = newItems;
      }
      items[count++] = item;
   }

   public T Pop() {
      T result = items[--count];
      items[count] = default(T);
      return result;
   }

   public IEnumerator<T> GetEnumerator() {
      for (int i = count - 1; i >= 0; --i) yield return items[i];
   }
}
```

GetEnumerator 方法可转换为编译器生成的枚举器类的实例化，该类封装了迭代器块中的代码，如下所示。

```
class Stack<T>: IEnumerable<T>
{
   ...
   public IEnumerator<T> GetEnumerator() {
      return new __Enumerator1(this);
   }

   class __Enumerator1: IEnumerator<T>, IEnumerator
   {
      int __state;
      T __current;
      Stack<T> __this;
      int i;

      public __Enumerator1(Stack<T> __this) {
         this.__this = __this;
      }

      public T Current {
         get { return __current; }
      }

      object IEnumerator.Current {
         get { return __current; }
      }

      public bool MoveNext() {
         switch (__state) {
            case 1: goto __state1;
            case 2: goto __state2;
         }
         i = __this.count - 1;
      __loop:
         if (i < 0) goto __state2;
         __current = __this.items[i];
         __state = 1;
         return true;
      __state1:
         --i;
         goto __loop;
      __state2:
         __state = 2;
         return false;
      }

      public void Dispose() {
         __state = 2;
      }

      void IEnumerator.Reset() {
         throw new NotSupportedException();
      }
   }
}
```

在前面的转换中，迭代器块中的代码转换为状态机，并置于枚举器类的 MoveNext 方法中。此外，局部变量 i 转换为枚举器对象中的字段，这样它就可以在 MoveNext 的多次调用之间继续存在。

下面的示例打印整数 1 到 10 的简单乘法表。该示例中的 FromTo 方法使用迭代器实现，并且返回一个可枚举对象。

```
using System;
using System.Collections.Generic;

class Test
{
   static IEnumerable<int> FromTo(int from, int to) {
      while (from <= to) yield return from++;
   }

   static void Main() {
      IEnumerable<int> e = FromTo(1, 10);
      foreach (int x in e) {
         foreach (int y in e) {
            Console.Write("{0,3} ", x * y);
         }
         Console.WriteLine();
      }
   }
}
```

FromTo 方法可转换为编译器生成的可枚举类的实例化，该类封装了迭代器块中的代码，如下所示。

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
   ...

   static IEnumerable<int> FromTo(int from, int to) {
      return new __Enumerable1(from, to);
   }

   class __Enumerable1:
      IEnumerable<int>, IEnumerable,
      IEnumerator<int>, IEnumerator
   {
      int __state;
      int __current;
      int __from;
      int from;
      int to;
      int i;

      public __Enumerable1(int __from, int to) {
         this.__from = __from;
         this.to = to;
      }

      public IEnumerator<int> GetEnumerator() {
         __Enumerable1 result = this;
         if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
            result = new __Enumerable1(__from, to);
            result.__state = 1;
         }
         result.from = result.__from;
         return result;
      }

      IEnumerator IEnumerable.GetEnumerator() {
         return (IEnumerator)GetEnumerator();
      }

      public int Current {
         get { return __current; }
      }
```

```
        object IEnumerator.Current {
           get { return __current; }
        }

        public bool MoveNext() {
           switch (__state) {
           case 1:
              if (from > to) goto case 2;
              __current = from++;
              __state = 1;
              return true;
           case 2:
              __state = 2;
              return false;
           default:
              throw new InvalidOperationException();
           }
        }

        public void Dispose() {
           __state = 2;
        }

        void IEnumerator.Reset() {
           throw new NotSupportedException();
        }
     }
  }
```

可枚举类同时实现了可枚举接口和枚举器接口，使其既可作为可枚举对象，也可作为枚举器对象。首次调用 GetEnumerator 方法时将返回该可枚举对象本身。对可枚举对象的 GetEnumerator 的后续调用（如果存在），将返回可枚举对象的副本。因此，每个返回的枚举器都有自己的状态，一个枚举器中的更改不会影响其他枚举器。Interlocked.CompareExchange 方法用于确保线程安全的操作。

from 和 to 参数转换为可枚举类中的字段。因为 from 是在迭代器块中修改的，所以引入附加 __from 字段以保存提供给每个枚举器中的 from 的初始值。

如果在 __state 为 0 时调用 MoveNext 方法，该方法将引发 InvalidOperationException。这可防止在未事先调用 GetEnumerator 的情况下将可枚举对象用作枚举器对象。

下面的示例演示一个简单的树类。Tree<T> 类使用迭代器实现其 GetEnumerator 方法。迭代器按照中缀顺序枚举树的元素。

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
   T value;
   Tree<T> left;
   Tree<T> right;

   public Tree(T value, Tree<T> left, Tree<T> right) {
      this.value = value;
      this.left = left;
      this.right = right;
   }

   public IEnumerator<T> GetEnumerator() {
      if (left != null) foreach (T x in left) yield x;
      yield value;
      if (right != null) foreach (T x in right) yield x;
   }
}
```

```
class Program
{
   static Tree<T> MakeTree<T>(T[] items, int left, int right) {
      if (left > right) return null;
      int i = (left + right) / 2;
      return new Tree<T>(items[i],
         MakeTree(items, left, i - 1),
         MakeTree(items, i + 1, right));
   }

   static Tree<T> MakeTree<T>(params T[] items) {
      return MakeTree(items, 0, items.Length - 1);
   }

   // The output of the program is:
   // 1 2 3 4 5 6 7 8 9
   // Mon Tue Wed Thu Fri Sat Sun

   static void Main() {
      Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
      foreach (int i in ints) Console.Write("{0} ", i);
      Console.WriteLine();

      Tree<string> strings = MakeTree(
         "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
      foreach (string s in strings) Console.Write("{0} ", s);
      Console.WriteLine();
   }
}
```

GetEnumerator 方法可转换为编译器生成的枚举器类的实例化，该类封装了迭代器块中的代码，如下所示。

```
class Tree<T>: IEnumerable<T>
{
   ...

   public IEnumerator<T> GetEnumerator() {
      return new __Enumerator1(this);
   }

   class __Enumerator1 : IEnumerator<T>, IEnumerator
   {
      Node<T> __this;
      IEnumerator<T> __left, __right;
      int __state;
      T __current;

      public __Enumerator1(Node<T> __this) {
         this.__this = __this;
      }

      public T Current {
         get { return __current; }
      }

      object IEnumerator.Current {
         get { return __current; }
      }

      public bool MoveNext() {
         try {
            switch (__state) {

            case 0:
               __state = -1;
               if (__this.left == null) goto __yield_value;
               __left = __this.left.GetEnumerator();
               goto case 1;
```

```
            case 1:
               __state = -2;
               if (!__left.MoveNext()) goto __left_dispose;
               __current = __left.Current;
               __state = 1;
               return true;

            __left_dispose:
               __state = -1;
               __left.Dispose();

            __yield_value:
               __current = __this.value;
               __state = 2;
               return true;

            case 2:
               __state = -1;
               if (__this.right == null) goto __end;
               __right = __this.right.GetEnumerator();
               goto case 3;

            case 3:
               __state = -3;
               if (!__right.MoveNext()) goto __right_dispose;
               __current = __right.Current;
               __state = 3;
               return true;

            __right_dispose:
               __state = -1;
               __right.Dispose();

            __end:
               __state = 4;
               break;

            }
         }
         finally {
            if (__state < 0) Dispose();
         }
         return false;
      }

      public void Dispose() {
         try {
            switch (__state) {

            case 1:
            case -2:
               __left.Dispose();
               break;

            case 3:
            case -3:
               __right.Dispose();
               break;

            }
         }
         finally {
            __state = 4;
         }
      }
```

```
            void IEnumerator.Reset() {
               throw new NotSupportedException();
            }
         }
      }
```

foreach 语句中使用的编译器生成的临时变量被提升为枚举器对象的 __left 和 __right 字段。枚举器对象的 __state 字段得到了妥善的更新，以便在引发异常时正确地调用正确的 Dispose() 方法。注意，不可能使用简单的 foreach 语句写入转换后的代码。

23. 分部类型

23.1 分部声明

当通过多个部分来定义类型时，将使用新增的类型修饰符 partial。为确保与现有程序的兼容性，此修饰符有别于其他修饰符：与 get 和 set 一样，它不是一个关键字，并且其后必须紧跟下列关键字之一：class、struct 或 interface。

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ partial$_{opt}$ class *identifier* *type-parameter-list*$_{opt}$
class-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *class-body* ;$_{opt}$

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ partial$_{opt}$ struct *identifier* *type-parameter-list*$_{opt}$
struct-interfaces$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *struct-body* ;$_{opt}$

interface-declaration:
attributes$_{opt}$ *interface-modifiers*$_{opt}$ partial$_{opt}$ interface *identifier* *type-parameter-list*$_{opt}$
interface-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *interface-body* ;$_{opt}$

分部类型声明的每个部分都必须包含 partial 修饰符，并且其声明必须与其他部分位于同一命名空间。partial 修饰符的出现指示其他位置可能还有类型声明的其他部分，但是这些其他部分并非必须存在；如果只有一个类型声明，包含 partial 修饰符也是有效的。

分部类型的所有部分必须一起编译，以使这些部分可在编译时被合并。特别指出的是，分部类型不允许对已经编译的类型进行扩展。

可使用 partial 修饰符在多个部分中声明嵌套类型。通常，其包含类型也使用 partial 声明，并且嵌套类型的每个部分均在该包含类型的不同部分中声明。

不允许使用 partial 修饰符声明委托或枚举。

23.1.1 属性

分部类型的属性是通过组合每个部分的属性（不指定顺序）来确定的。如果一个属性放置在多个部分中，则相当于多次在该类型上指定此属性。例如，下面的两个部分：

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

相当于下面的声明：

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

类型参数的属性以类似的方式进行组合。

23.1.2 修饰符

当分部类型声明指定了可访问性（public、protected、internal 和 private 修饰符）时，它必须与所有其他指定了可访问性的部分一致。如果某个分部类型的任何部分都未规定可访问性，则该类型将获得相应的默认可访问性（第 3.5.1 节）。

如果嵌套类型的一个或多个分部声明包含 new 修饰符，则在嵌套类型对继承的成员进行了隐藏（第 3.7.1.2 节）的情况不会发出警告。

如果某个类的一个或多个分部声明包含 abstract 修饰符，则该类被视为抽象类（第 10.1.1.1 节）。否则，该类被视为非抽象类。

如果某个类的一个或多个分部声明包含 sealed 修饰符，则该类被视为密封类（第 10.1.1.2 节）。否则，该类被视为非封闭类。

注意，一个类不能既是抽象类又是密封类。

当 unsafe 修饰符用于某个分部类型声明时，只有该特定部分才被视为不安全的上下文（第 18.1 节）。

23.1.3 类型参数和约束

如果在多个部分中声明泛型类型，则每个部分都必须声明类型参数。每个部分都必须有相同数目的类型参数，并且每个类型参数按照顺序有相同的名称。

当分部泛型类型声明包含约束（where 子句）时，该约束必须与包含约束的所有其他部分一致。具体而言，包含约束的每个部分都必须有针对相同的类型参数集的约束，并且对于每个类型参数，主要、次要和构造函数约束集都必须等效。如果两个约束集包含相同的成员，则它们等效。如果某个分部泛型类型的任何部分都未指定类型参数约束，则该类型参数被视为无约束。

下面的示例

```
partial class Dictionary<K,V>
   where K: IComparable<K>
   where V: IKeyProvider<K>, IPersistable
{
   ...
}

partial class Dictionary<K,V>
   where V: IPersistable, IKeyProvider<K>
   where K: IComparable<K>
{
   ...
}

partial class Dictionary<K,V>
{
   ...
}
```

是正确的，因为包含约束的那些部分（前两个）实际上分别对相同的类型参数集指定了相同的主要、次要和构造函数约束集。

23.1.4 基类

当一个分部类声明包含基类说明时，它必须与包含基类说明的所有其他部分一致。如果某个分部类的任何部分都不包含基类说明，则基类将为 System.Object（第 10.1.2.1 节）。

23.1.5 基接口

在多个部分中声明的类型的基接口集是每个部分中指定的基接口的并集。一个特定基接口在每个部分中只能指定一次，但是允许多个部分指定相同的基接口。任何给定基接口的成员只能有一个实现。

在下面的示例中

```
partial class C: IA, IB {...}
partial class C: IC {...}
partial class C: IA, IB {...}
```

类 C 的基接口集为 IA、IB 和 IC。

通常，每个部分都提供了在该部分上声明的接口的实现；但这不是必需的。一个部分可能提供在另一个部分上声明的接口的实现：

```
partial class X
{
   int IComparable.CompareTo(object o) {...}
}
partial class X: IComparable
{
   ...
}
```

23.1.6 成员

在多个部分中声明的类型的成员是每个部分中声明的成员的并集。所有部分的类型声明主体共享相同的声明空间（第 3.3 节），并且每个成员的范围（第 3.7 节）都扩展到所有部分的主体。任何成员的可访问性域总是包含包容类型的所有部分；在一个部分中声明的 private 成员可从其他部分随意访问。在类型的多个部分中声明同一个成员将引起编译时错误，除非该成员是带有 partial 修饰符的类型。

```
partial class A
{
   int x;                  // Error, cannot declare x more than once
   partial class Inner     // Ok, Inner is a partial type
   {
      int y;
   }
}
partial class A
{
   int x;                  // Error, cannot declare x more than once
   partial class Inner     // Ok, Inner is a partial type
   {
      int z;
   }
}
```

虽然类型中的成员的顺序对于 C# 代码无关紧要，但是在与其他语言或环境交互时，这可能很重要。在这些情况下，没有对分为多个部分声明的类型内的成员顺序进行定义。

23.2 名称绑定

虽然可扩展类型的每个部分都必须在同一命名空间中声明，但是这些部分通常在不同的命名空间声明下编写。因此，每个部分可能存在不同的 using 指令（第 9.3 节）。当解释一个部分内的简单名称（第 7.5.2 节）时，只考虑包容该部分的命名空间定义的 using 指令。这可能会导致同一标识符在不同部分中具有不同的含义：

```
namespace N
{
   using List = System.Collections.ArrayList;
   partial class A
   {
      List x;           // x has type System.Collections.ArrayList
   }
}
namespace N
{
   using List = Widgets.LinkedList;
   partial class A
   {
      List y;           // y has type Widgets.LinkedList
   }
}
```

24. 可空类型

24.1 可空类型

可空类型属于值类型：

value-type:
　　struct-type
　　enum-type

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

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

non-nullable-value-type:
　　type

在可空类型中的 ? 修饰符之前指定的类型称为该可空类型的基础类型 ***(underlying type)***。可空类型的基础类型可以是任何非可空值类型或任何限制为非可空值类型的类型参数（即任何具有 `struct` 约束的类型参数）。可空类型的基础类型不能是可空类型或引用类型。例如，`int??` 和 `string?` 是无效类型。

可空类型可以表示其基础类型的所有值和一个额外的空值。

语法 `T?` 是 `System.Nullable<T>` 的简写形式，这两种形式可以互换使用。

24.1.1 成员

可空类型 `T?` 的实例有两个公共只读属性：

- 类型为 `bool` 的 `HasValue` 属性
- 类型为 `T` 的 `Value` 属性

`HasValue` 为 true 的实例称为非空。非空实例包含一个已知值，可通过 `Value` 返回该值。

`HasValue` 为 false 的实例称为空。空实例有一个不确定的值。尝试读取空实例的 `Value` 将导致引发 `System.InvalidOperationException`。

除了默认构造函数之外，每个可空类型 `T?` 都有一个具有类型为 `T` 的单个参数的公共构造函数。例如，给定一个类型为 `T` 的值 `x`，调用形如

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

的构造函数将创建 `T?` 的非空实例，其 `Value` 属性为 `x`。

实际上，根本没有必要显式调用可空类型的构造函数，因为从 `T` 到 `T?` 的隐式转换提供了同样的功能。

24.1.2 默认值

可空类型的默认值是一个 HasValue 属性为 false 且 Value 属性不确定的实例。默认值也称为可空类型的空值 ***(null value)***。从 null 文本到任何可空类型存在隐式转换，这种转换产生该类型的空值。

24.1.3 值类型约束

值类型约束（写作关键字 struct）将类型参数限制为非可空值类型（第 20.7 节）。具体而言，如果类型参数是用 struct 约束声明的，则用于该类型参数的类型实参可以是可空类型之外的任何值类型。

System.Nullable<T> 类型指定 T 的值类型约束。因此，禁止使用 T?? 和 Nullable<Nullable<T>> 形式的递归构造类型。

24.2 转换

后面各节中使用的术语如下：

- 术语包装 ***(wrapping)*** 表示将 T 类型的值打包在类型 T? 的实例中的过程。可通过计算表达式 new T?(x)，将 T 类型的值 x 包装为类型 T?。
- 术语解包 ***(unwrapping)*** 表示获取包含在类型 T? 的实例中的类型为 T 的值的过程。通过计算表达式 x.Value，可将类型 T? 的值 x 解包为类型 T。尝试解包空实例将导致引发 System.InvalidOperationException。

24.2.1 null 文本转换

从 null 文本到任何可空类型存在隐式转换。这种转换产生给定可空类型的空值（第 24.1.2 节）。

24.2.2 可空转换

可空转换 ***(Nullable conversion)*** 还允许将对非可空值类型执行的预定义转换操作也用于这些类型的可空形式。对于从非可空值类型 S 转换为非可空值类型 T 的每一个预定义的隐式或显式转换（第 6.1.1 节、第 6.1.2 节、第 6.1.3 节、第 6.1.6 节、第 6.2.1 节和第 6.2.2 节），存在下列可空转换：

- 从 S? 到 T? 的隐式或显式转换。
- 从 S 到 T? 的隐式或显式转换。
- 从 S? 到 T 的显式转换。

可空转换本身属于隐式或显式转换。

某些可空转换属于标准转换，可作为用户定义转换的一部分进行。具体而言，所有隐式可空转换都属于标准隐式转换（第 6.3.1 节），那些满足第 6.3.2 节的要求的显式可空转换则属于标准显式转换。

基于从 S 到 T 的基础转换计算可空转换的过程如下：

- 如果可空转换是从 S? 到 T?：
  - 如果源值为空（HasValue 属性为 false），则结果为 T? 类型的空值。
  - 否则，转换计算过程为从 S? 解包为 S，然后进行从 S 到 T 的基础转换，最后从 T 包装为 T?。
- 如果可空转换是从 S 到 T?，则转换计算过程为从 S 到 T 的基础转换，然后从 T 包装为 T?。
- 如果可空转换是从 S? 到 T，则转换计算过程为从 S? 解包为 S，然后进行从 S 到 T 的基础转换。

24.2.3 装箱和取消装箱转换

在装箱和取消装箱转换中，可空类型的行为与其他值类型不同。可空类型的值的装箱结果为空引用或对基础类型的装箱值的引用。反之，空引用或对给定类型的装箱值的引用在取消装箱后将成为该类型的可空形式的值。例如，int? 的装箱结果为空引用或对装箱的 int 的引用，并且空引用或对装箱的 int 的引用在取消装箱后为 int?。

可空类型 T? 与 T 具有相同的装箱转换（第 6.1.5 节）目标类型集。从可空类型 T? 进行的装箱转换的过程如下：

- 如果源值为空（HasValue 属性为 false），则结果为目标类型的空引用。
- 否则，结果为对经过源值解包和装箱后所产生的装箱 T 的引用。

可空类型 T? 与 T 具有相同的取消装箱转换（第 6.2.4 节）源类型集。转换为可空类型 T? 的取消装箱的过程如下：

- 如果源为空引用，则结果为 T? 类型的空值。
- 如果源为对装箱的 T 的引用，则结果为对源进行取消装箱和包装后产生的 T?。
- 否则，引发 System.InvalidCastException。

上述规则实际统一了可空类型和引用类型的空表示形式 -- 可空类型的空值在装箱后变为空引用，空引用在取消装箱后则变为可空类型的空值。这些规则还保证装箱值的运行时类型始终是非可空值类型，并且任何情况下都不会成为可空类型。

因为可空类型 T? 与 T 具有相同的装箱转换目标类型集，所以从 T? 到由 T 实现的任何接口都存在装箱转换，并且从由 T 实现的任何接口都存在到 T? 的取消装箱转换。但是，任何情况下可空类型都不满足接口约束，即使基础类型实现了该特定接口。

下面的示例演示涉及可空类型的几个装箱和取消装箱操作。

```
int i = 123;

int? x = 456;
int? y = null;

object o1 = i;          // o1 = reference to boxed int 123
object o2 = x;          // o2 = reference to boxed int 456
object o3 = y;          // o3 = null

int i1 = (int)o1;       // i1 = 123
int i2 = (int)o2;       // i2 = 456
int i3 = (int)o3;       // Error, System.NullReferenceException

int? ni1 = (int?)o1;    // ni1 = 123
int? ni2 = (int?)o2;    // ni2 = 456
int? ni3 = (int?)o3;    // ni3 = null
```

24.2.4 允许的用户定义转换

规定允许的用户定义转换运算符声明的规则已经修改，现在，也允许结构声明转换结构类型可空形式的转换运算符。下列限制替代了第 6.4.1 节和第 10.9.3 节中所列的限制。

仅当下列所有条件都成立时才允许类或结构声明从源类型 S 到目标类型 T 的转换，其中 S_0 和 T_0 是从 S 和 T 移除尾随 ? 修饰符（如果有）得到的类型：

- S_0 和 T_0 的类型不同。
- S_0 和 T_0 之一是声明了该运算符的类类型或结构类型。
- S_0 和 T_0 都不是 *interface-type*。
- 除用户定义的转换之外，不存在从 S 到 T 或从 T 到 S 的转换。

24.2.5 用户定义转换的计算

对用户定义转换的计算（第 6.4.2 节）进行了如下修改以支持可空类型：

- 在确定用户定义的转换运算符针对的类型集之前，先从源和目标类型移除尾随 ? 修饰符（如果有）。例如，从类型 S? 转换为 T? 时，将认为用户定义的转换运算符所针对的类型集由 S 和 T 组成。
- 当源和目标类型都为可空类型时，相应的转换运算符集不仅包括用户定义的转换运算符，也包括提升的转换运算符（第 24.2.6 节）。
- 在确定最具体的转换运算符时，用户定义的转换运算符优先于提升的转换运算符。

24.2.6 提升的转换

给定一个从非可空值类型 S 到非可空值类型 T 的用户定义转换运算符，存在从 S? 到 T? 的提升转换运算符 ***(lifted conversion operator)***。这个提升转换运算符执行从 S? 到 S 的解包，接着是从 S 到 T 的用户定义转换，然后是从 T 到 T? 的包装，空值的 S? 直接转换为空值的 T? 除外。

提升的转换运算符与其基础用户定义转换运算符具有相同的隐式或显式类别。

24.2.7 用户定义的隐式转换

下面的内容替换第 6.4.3 节。

从类型 S 到类型 T 的用户定义的隐式转换的处理过程如下：

- 确定从 S 和 T 移除尾随 ? 修饰符（如果有）所得到的类型 S_0 和 T_0。
- 查找类型集 D，将针对该类型集考虑用户定义的转换运算符。此集合由 S_0（如果 S_0 是类或结构）、S_0 的基类（如果 S_0 是类）和 T_0（如果 T_0 是类或结构）组成。
- 查找适用的转换运算符集合 U。此集合由用户定义的隐式转换运算符和提升的（第 24.2.6 节）隐式转换运算符（如果 S 和 T 都为可空）组成，这些运算符是由 D 中的类或结构声明的，用于从包含 S 的类型转换为被 T 包含的类型。如果 U 为空，则转换未定义，并发生编译时错误。
- 查找 U 中的运算符的最具体的源类型 S_x：
  - 如果 U 中的所有运算符都从 S 转换，则 S_x 为 S。
  - 否则，S_x 是 U 中的运算符的源类型合集中被包含程度最大的类型。如果无法恰好找到一个被包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
- 查找 U 中的运算符的最具体的目标类型 T_x：
  - 如果 U 中的所有运算符都转换为 T，则 T_x 为 T。
  - 否则，T_x 是 U 中的运算符的目标类型合集中包含程度最大的类型。如果无法恰好找到一个包含程度最大的类型，则转换是不明确的，并且发生编译时错误。

- 查找最具体的转换运算符：
  - 如果 U 中正好含有一个从 S_x 转换到 T_x 的用户定义转换运算符，则这就是最具体的转换运算符。
  - 否则，如果 U 恰好包含一个从 S_x 转换到 T_x 的提升转换运算符，则这就是最具体的转换运算符。
  - 否则，转换是不明确的，并发生编译时错误。
- 最后，应用转换：
  - 如果 S 不是 S_x，则执行从 S 到 S_x 的标准隐式转换。
  - 调用最具体的转换运算符，以从 S_x 转换到 T_x。
  - 如果 T_x 不是 T，则执行从 T_x 到 T 的标准隐式转换。

24.2.8 用户定义的显式转换

下面的内容替换第 6.4.4 节。

从类型 S 到类型 T 的用户定义的显式转换处理过程如下：

- 确定从 S 和 T 移除尾随 ? 修饰符（如果有）所得到的类型 S_0 和 T_0。
- 查找类型集 D，将针对该类型集考虑用户定义的转换运算符。此集合由 S_0（如果 S_0 为类或结构）、S_0 的基类（如果 S_0 为类）、T_0（如果 T_0 为类或结构）和 T_0 的基类（如果 T_0 为类）组成。
- 查找适用的转换运算符集合 U。此集合由用户定义的隐式或显式转换运算符和提升的（第 24.2.6 节）隐式或显式转换运算符（如果 S 和 T 都为可空）组成，这些运算符是由 D 中的类或结构声明的，用于从包含 S 或被 S 包含的类型转换为包含 T 或被 T 包含的类型。如果 U 为空，则转换未定义，并发生编译时错误。
- 查找 U 中的运算符的最具体的源类型 S_x：
  - 如果 U 中的所有运算符都从 S 转换，则 S_x 为 S。
  - 否则，如果 U 中的所有运算符都从包含 S 的类型转换，则 S_x 是这些运算符的源类型合集中被包含程度最大的类型。如果无法恰好找到一个被包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
  - 否则，S_x 是 U 中的运算符的合并源类型集中包含程度最大的类型。如果无法恰好找到一个包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
- 查找 U 中的运算符的最具体的目标类型 T_x：
  - 如果 U 中的所有运算符都转换为 T，则 T_x 为 T。
  - 否则，如果 U 中的所有运算符都转换为被 T 包含的类型，则 T_x 在这些运算符的合并目标类型集中是包含程度最大的类型。如果无法恰好找到一个包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
  - 否则，T_x 是 U 中的运算符的目标类型合集中被包含程度最大的类型。如果无法恰好找到一个被包含程度最大的类型，则转换是不明确的，并且发生编译时错误。
- 查找最具体的转换运算符：
  - 如果 U 中正好含有一个从 S_x 转换到 T_x 的用户定义转换运算符，则这就是最具体的转换运算符。

- 否则，如果 U 恰好包含一个从 S_x 转换到 T_x 的提升转换运算符，则这就是最具体的转换运算符。
- 否则，转换是不明确的，并发生编译时错误。

- 最后，应用转换：
  - 如果 S 不是 S_x，则执行从 S 到 S_x 的标准显式转换。
  - 调用最具体的转换运算符，以从 S_x 转换到 T_x。
  - 如果 T_x 不是 T，则执行从 T_x 到 T 的标准显式转换。

24.3 表达式

24.3.1 提升运算符

提升运算符 ***(Lifted operator)*** 允许操作非可空值类型的预定义运算符及用户定义运算符亦可用于这些类型的可空形式。提升运算符是根据符合某些要求的预定义和用户定义运算符构造而成的，如下所述：

- 对于一元运算符

  ```
  +   ++   -   --   !   ~
  ```

  如果操作数和结果类型都为非可空值类型，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到操作数和结果类型构造而成的。如果操作数为空，则提升运算符产生一个空值。否则，提升运算符对该操作数进行解包，应用基础运算符，并包装结果。

- 对于二元运算符

  ```
  +   -   *   /   %   &   |   ^   <<   >>
  ```

  如果操作数和结果类型都为非可空值类型，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到每个操作数和结果类型构造的。如果一个操作数为空或两个操作数都为空，则提升运算符产生一个空值（bool? 类型的 & 和 | 运算符除外，如第 24.3.8 节所述）。否则，提升运算符对这些操作数进行解包，应用基础运算符，并包装结果。

- 对于相等运算符

  ```
  ==   !=
  ```

  如果两个操作数类型都为非可空值类型，并且结果类型为 bool，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到每个操作数类型构造的。该提升运算符认为两个空值相等，空值不等于任何非空值。如果两个操作数都为非空，则提升运算符对这两个操作数进行解包，并应用基础运算符以产生 bool 结果。

- 对于关系运算符

  ```
  <   >   <=   >=
  ```

  如果两个操作数类型都为非可空值类型，并且结果类型为 bool，则存在运算符的提升形式。该提升形式是通过将一个 ? 修饰符添加到每个操作数类型构造的。如果一个操作数为空或两个操作数都为空，则提升运算符产生 false 值。否则，提升运算符对这些操作数进行解包，并应用基础运算符以产生 bool 结果。

24.3.2 允许的用户定义运算符

控制允许的用户定义运算符声明的规则已经修改，允许结构也可以声明结构类型的可空形式的运算符。下面的限制取代了第 10.9.1 节和第 10.9.2 节中所列的限制，其中 T 表示包含运算符声明的类或结构类型：

- 一元 +、-、! 或 ~ 运算符必须具有单个 T 或 T? 类型的参数，并且可以返回任何类型。
- 一元 ++ 或 -- 运算符必须具有单个 T 或 T? 类型的参数并且必须返回相同类型。
- 一元 true 或 false 运算符必须具有单个 T 或 T? 类型的参数并且必须返回类型 bool。
- 二元非移位运算符必须带有两个参数，其中至少有一个必须为类型 T 或 T?，并且可返回其中的任一类型。
- 二元 << 或 >> 运算符必须带有两个参数，其中第一个必须具有类型 T 或 T?，第二个必须具有类型 int 或 int?，并且可返回其中的任一类型。

由于上述规则，未提升运算符和提升运算符可能具有相同的签名。这种情况下，未提升运算符优先，如第 24.3.3 节所述。

24.3.3 运算符重载解析

一元和二元运算符重载解析（第 7.2.3 节和第 7.2.4 节）规则已作如下修改，以支持提升运算符：

- 从操作数类型移除尾随 ? 修饰符（如果有），以确定在哪些类型中查找用户定义的运算符声明。例如，如果操作数为类型 X? 和 Y?，则通过检查 X 和 Y 确定候选运算符集。
- 在确定候选用户定义运算符（第 7.2.5 节）集时，在类型中声明的运算符的提升形式被视为也是由该类型声明的。
- 运算符提升适用于预定义运算符，预定义运算符的提升形式本身被视为预定义运算符。
- 在选择单个最佳运算符时，如果两个运算符的签名相同，则未提升的运算符优于提升的运算符。

24.3.4 相等操作符和空

== 和 != 运算符允许一个操作数为可空类型的值，另一个为 null 文本，即使操作中不存在预定义或用户定义的运算符（未提升或提升形式）。

对于下面某个形式的操作

```
x == null    null == x    x != null    null != x
```

其中 x 为可空类型的表达式，如果运算符重载解析（第 7.2.4 节和第 §24.3.3 节）未能找到适用的运算符，则改为从 x 的 HasValue 属性计算结果。具体而言，前两种形式转换为 !x.HasValue，后两种形式转换为 x.HasValue。

24.3.5 is 运算符

is 运算符（第 7.9.9 节）已扩展为支持可空类型。e is T 形式的操作（其中 e 为表达式，T 为类型）在类型实参替换了所有类型参数之后，将进行如下计算：

- 如果 e 的类型是引用类型或可空类型，并且 e 的值为空，则结果为 false。
- 否则，根据下列规则让 D 表示 e 的动态类型：

- 如果 e 的类型为引用类型，则 D 为 e 引用的实例的运行时类型。
- 如果 e 的类型为可空类型，则 D 为该可空类型的基础类型。
- 如果 e 的类型为非可空值类型，则 D 为 e 的类型。

- 该操作的结果取决于 D 和 T，具体如下：
  - 如果 T 为引用类型，那么，在以下情况下结果为 true：D 和 T 为相同类型，或者 D 为引用类型并且存在从 D 到 T 的隐式引用转换，或者 D 为值类型并且存在从 D 到 T 的装箱转换。
  - 如果 T 为可空类型，那么，当 D 为 T 的基础类型时结果为 true。
  - 如果 T 为非可空值类型，那么，如果 D 和 T 为相同类型，则结果为 true。
  - 否则，结果为 false。

24.3.6 as 运算符

as 运算符（第 7.9.10 节）已扩展为支持可空类型。在 e as T 形式的操作中，e 必须为表达式，T 必须为引用类型、已知为引用类型的类型参数或可空类型。此外，下列条件中必须至少有一条成立，否则会发生编译时错误：

- 存在从类型 e 到 T 的标识转换（第 6.1.1 节）、隐式引用转换（第 6.1.4 节）、装箱转换（第 6.1.5 节、第 24.2.3 节）、显式引用转换（第 6.2.3 节）或取消装箱转换（第 6.2.4 节、第 24.2.3 节）。
- e 或 T 的类型为开放类型。
- e 为 null 文本。

e as T 操作产生与下面的操作相同的结果

```
e is T ? (T)(e) : (T)null
```

不同的是 e 只计算一次。编译器应该优化 e as T 以最多执行一次动态类型检查，而不是上面的扩展隐含的两次动态类型检查。

24.3.7 复合赋值

复合赋值操作（第 7.13.2 节）支持提升运算符。因为复合赋值 x *op*= y 计算为 x = x *op* y 或 x = (T)(x *op* y)，所以现有计算规则隐含了提升运算符，不需要更改这些规则。

24.3.8 bool? 类型

可空布尔类型 bool? 可表示三个值 true、false 和 null，概念上类似于 SQL 中的布尔表达式的三值类型。为了确保针对 bool? 操作数的 & 和 | 运算符产生的结果与 SQL 的三值逻辑一致，提供了下列预定义运算符：

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

下表列出了这些运算符对 true、false 和 null 值的所有组合所产生的结果。

x	Y	x & y	x | y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

24.3.9 空合并运算符

?? 运算符称为空合并运算符。

null-coalescing-expression:
conditional-or-expression
conditional-or-expression ?? null-coalescing-expression

C# 词法语法中新增了 ?? 标记。经过更新的 *operator-or-punctuator* 词法语法产生形式请参见第 20.10 节。

a ?? b 形式的空合并表达式要求 a 为可空类型或引用类型。如果 a 为非空，则 a ?? b 的结果为 a；否则，结果为 b。仅当 a 为空时，该操作才计算 b。

空合并运算符为右结合运算符，表示操作从右向左进行组合。例如，a ?? b ?? c 形式的表达式按 a ?? (b ?? c) 计算。概括地说，E_1 ?? E_2 ?? ... ?? E_N 形式的表达式返回第一个非空操作数，如果所有操作数都为 null，则返回 null。

表达式 a ?? b 的类型取决于两个操作数类型之间有哪些隐式转换可用。按照优先级顺序，a ?? b 的类型为 A_0、A 或 B，其中 A 的类型为 a，B 的类型为 b，A_0 为从 A 移除尾随 ? 修饰符所得到的类型。具体而言，a ?? b 的处理过程如下：

- 如果 A 不是可空类型或引用类型，则会发生编译时错误。
- 如果 A 为可空类型并且存在从 b 到 A_0 的隐式转换，则结果类型为 A_0。在运行时，首先计算 a。如果 a 不为空，则 a 解包为类型 A_0，这即是结果。否则，计算 b 并转换为类型 A_0，这即是结果。
- 否则，如果存在从 b 到 A 的隐式转换，则结果类型为 A。在运行时，首先计算 a。如果 a 不为空，则 a 即是结果。否则，计算 b 并转换为类型 A，这即是结果。
- 否则，如果存在从 A_0 到 B 的隐式转换，则结果类型为 B。在运行时，首先计算 a。如果 a 不为空，则 a 解包为类型 A_0（除非 A 和 A_0 类型相同）并转换为类型 B，这即是结果。否则，计算 b 并且 b 作为结果。
- 否则，a 和 b 不兼容，并发生编译时错误。

25. 其他功能

25.1 属性访问器的可访问性

有时，组件设计要求某个属性对于 `set` 访问器和 `get` 访问器应具有不同的可访问性。最常见的情况是，设计者希望允许任何代码都可调用 `get` 访问器，但是限制 `set` 访问器只能由子代类型（`protected` 访问）或同一程序内的类型（`internal` 访问）调用。通过本节介绍的语言扩展功能，我们可以在 C# 中实现这类设计。

25.1.1 访问器声明

属性访问器和索引器访问器的语法已进行了如下的修改，增加了一个可选的 *accessor-modifier*：

get-accessor-declaration:
　　attributes$_{opt}$　*accessor-modifier*$_{opt}$　`get`　*accessor-body*

set-accessor-declaration:
　　attributes$_{opt}$　*accessor-modifier*$_{opt}$　`set`　*accessor-body*

accessor-modifier:
```
    protected
    internal
    private
    protected internal
    internal protected
```

accessor-modifier 的使用具有下列限制：

- *accessor-modifier* 不可用在接口中或显式接口成员实现中。
- 对于没有 `override` 修饰符的属性或索引器，仅当该属性或索引器同时带有 `get` 和 `set` 访问器时，才允许使用 *accessor-modifier*，并且只能用于其中的一个访问器。
- 对于包含 `override` 修饰符的属性或索引器，访问器必须匹配被重写的访问器的 *accessor-modifier*（如果存在）。
- *accessor-modifier* 声明的可访问性的限制性必须严格高于属性或索引器本身所声明的可访问性。准确地说：
  - 如果属性或索引器声明了 `public` 可访问性，则可使用任何 *accessor-modifier*。
  - 如果属性或索引器声明了 `protected internal` 可访问性，则 *accessor-modifier* 可为 `internal`、`protected` 或 `private` 之一。
  - 如果属性或索引器声明了 `internal` 或 `protected` 可访问性，则 *accessor-modifier* 必须为 `private`。
  - 如果属性或索引器声明了 `private` 可访问性，则任何 *accessor-modifier* 都不可使用。

如果一个访问器带有 *accessor-modifier*，则该访问器的可访问域（第 3.5.2 节）通过使用 *accessor-modifier* 所声明的可访问性来确定。如果访问器没有 *accessor-modifier*，则该访问器的可访问域根据属性或索引器所声明的可访问性来确定。

25.1.2 访问器的使用

accessor-modifier 存在与否对于成员查找（第 7.3 节）或重载解析（第 7.4.2 节）毫无影响。属性或索引器的修饰符始终直接确定所绑定到的属性或索引器，并不考虑访问的上下文。

一旦选择了某个特定的属性或索引器，则所涉及的特定访问器的可访问域将用来确定该访问器的使用是否有效：

- 如果作为值（第 7.1.1 节）使用，则 `get` 访问器必须存在并且可访问。
- 如果作为简单赋值的目标使用（第 7.13.1 节），则 `set` 访问器必须存在并且可访问。
- 如果作为复合赋值的目标（第 7.13.2 节）或作为 `++` 或 `--` 运算符的目标（第 7.5.9 节、第 7.6.5 节）使用，则 `get` 访问器和 `set` 访问器都必须存在并且可访问。

在下面的示例中，属性 `A.Text` 被属性 `B.Text` 隐藏，甚至是在只调用 `set` 访问器的上下文中。相比之下，属性 `B.Count` 对于类 `M` 不可访问，因此改用可访问的属性 `A.Count`。

```
class A
{
   public string Text {
      get { return "hello"; }
      set { }
   }

   public int Count {
      get { return 5; }
      set { }
   }
}

class B: A
{
   private string text = "goodbye";
   private int count = 0;

   new public string Text {
      get { return text; }
      protected set { text = value; }
   }

   new protected int Count {
      get { return count; }
      set { count = value; }
   }
}

class M
{
   static void Main() {
      B b = new B();
      b.Count = 12;           // Calls A.Count set accessor
      int i = b.Count;        // Calls A.Count get accessor
      b.Text = "howdy";       // Error, B.Text set accessor not accessible
      string s = b.Text;      // Calls B.Text get accessor
   }
}
```

25.1.3 重写和接口实现

如果属性或索引器声明为 override，则进行重写的代码必须能够访问被重写的访问器。此外，属性或索引器本身以及访问器所声明的可访问性都必须与被重写的成员和访问器所声明的可访问性相匹配。例如：

```
public class B
{
   public virtual int P {
      protected set {...}
      get {...}
   }
}

public class D: B
{
   public override int P {
      protected set {...}        // Must specify protected here
      get {...}                  // Must not have a modifier here
   }
}
```

用来实现接口的访问器不能含有 *accessor-modifier*。如果仅使用一个访问器实现接口，则另一个访问器可以用 *accessor-modifier* 声明：

```
public interface I
{
   string Prop { get; }
}

public class C: I
{
   public Prop {
      get { return "April"; }    // Must not have a modifier here
      internal set {...}         // Ok, because I.Prop has no set accessor
   }
}
```

25.2 静态类

通常，不进行实例化且仅包含静态成员的类声明为具有私有构造函数的密封类。例如，.NET Framework 类库中的 System.Console 和 System.Environment 就属于这种类。虽然“具有私有构造函数的密封类”的设计模式可防止对这种类进行实例化或创建其子类，但是仍可将该类用作变量或参数的类型，并且可以使用该类声明实例成员。然而，此做法并无用处，因为该类类型的变量或参数几乎没有意义（它们只能为空），并且类中的实例成员不可访问（不存在用于访问这些成员的实例）。

静态类实现了“具有私有构造函数的密封类”设计模式，并且对与该模式逻辑关联的限制提供了更强的检查机制。

25.2.1 静态类声明

当类声明包含 static 修饰符时，所声明的类即为静态类 ***(static class)***。

class-declaration:
　attributes$_{opt}$ *class-modifiers*$_{opt}$ partial$_{opt}$ class *identifier* *type-parameter-list*$_{opt}$
　　class-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *class-body* ;$_{opt}$

class-modifiers:
　class-modifier
　class-modifiers *class-modifier*

class-modifier:
    new
    public
    protected
    internal
    private
    abstract
    sealed
    static

静态类声明受到以下限制：

- 静态类不可包含 sealed 或 abstract 修饰符。但是，注意，因为无法实例化静态类或从静态类派生，所以静态类的行为就好像既是密封的又是抽象的。
- 静态类不可包括 *class-base* 规范（第 10.1.2 节），并且不能显式指定基类或所实现接口的列表。静态类隐式从 object 类型继承。
- 静态类只能包含静态成员（第 10.2.5 节）。注意，常数和嵌套类型归为静态成员。
- 静态类不能含有声明的可访问性为 protected 或 protected internal 的成员。

违反上述任何限制都将导致编译时错误。

静态类没有实例构造函数。静态类中不能声明实例构造函数，并且对于静态类也不提供任何默认实例构造函数（第 10.10.4 节）。

静态类的成员并不会自动成为静态的，成员声明中必须显式包含一个 static 修饰符（常数和嵌套类型除外）。当一个类嵌套在一个静态的外层类中时，除非该类显式包含 static 修饰符，否则该嵌套类不是静态类。

25.2.2 引用静态类类型

如果下列条件成立，则允许 *namespace-or-type-name*（第 20.9.1 节）引用静态类：

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

如果下列条件成立，则允许 *primary-expression*（第 7.5 节）引用静态类：

- *primary-expression* 为 E.I 形式的 *member-access*（第 7.5.4 节）中的 E。

在任何其他上下文中，引用静态类将导致编译时错误。例如，将静态类用作基类、成员的构成类型（第 10.2.4 节）、泛型类型变量或类型参数约束，这些都是错误。同样，静态类不可用于数组类型、指针类型、new 表达式、强制转换表达式、is 表达式、as 表达式、sizeof 表达式或默认值表达式。

25.3 命名空间别名限定符

当类型或命名空间添加到程序集时，这些类型或命名空间的名称可能与相关程序中已经使用的名称发生冲突。例如，请看下面的两个程序集：

程序集 a1.dll：

```
namespace System.IO
{
   public class Stream {...}
```

```
   public class FileStream: Stream {...}
   ...
}
```

程序集 a2.dll：

```
namespace MyLibrary.IO
{
   public class EmptyStream: Stream {...}
}
```

以及下面引用这两个程序集的程序：

```
using System.IO;
using MyLibrary.IO;

class Program
{
   static void Main() {
      Stream s = new EmptyStream();
      ...
   }
}
```

如果在将来某个时候，名为 System.IO.EmptyStream 的类型添加到了 a1.dll 中，则上述程序中对 EmptyStream 的引用将变得不明确，从而发生编译时错误。

某种程度上，通过定义 using 别名并显式限定类型引用，可以防止与版本相关的代码破坏。例如，如果将上面的程序改成

```
using SIO = System.IO;
using MIO = MyLibrary.IO;

class Program
{
   static void Main() {
      SIO.Stream s = new MIO.EmptyStream();
      ...
   }
}
```

那么，引入一个名为 System.IO.EmptyStream 的类型将不会导致错误。但是，即使采用这种显式限定方法，还是存在由于引入新类型或新成员而导致错误的情况。例如，在上面的程序中，如果被引用的程序集引入了一个名为 MIO 的顶级命名空间，则可能发生多义性错误。

命名空间别名限定符 ***(namespace alias qualifier)*** :: 让类型名称的查找不受引入的新类型和新成员的影响成为可能。命名空间别名限定符总是出现在两个标识符之间，这两个标识符分别称为左标识符和右标识符。与普通的 . 标识符不同，:: 限定符的左标识符仅作为 extern 或 using 别名进行查找。

在下面的示例中

```
using SIO = System.IO;
using MIO = MyLibrary.IO;

class Program
{
   static void Main() {
      SIO::Stream s = new MIO::EmptyStream();
      ...
   }
}
```

在解析类型名称 SIO::Stream 和 MIO::EmptyStream 时，仅将 SIO 和 MIO 作为 extern 或 using 别名进行查找（第 25.4 节和第 9.3.1 节）。由于 extern 或 using 别名的范围不会扩展到定义别名的源文件之外，所以被引用的程序集不可能引入任何会影响解析的实体。这样，通过 :: 限定符为所有被引用的命名空间定义 using 别名并引用这些空间的成员，便可以防止将来由于版本控制而造成代码被破坏。

:: 限定符要求左标识符为标识符 global（如下文所述）或引用命名空间的 extern 或 using 别名。如果别名引用了某个类型，则会发生编译时错误。

当 :: 限定符的左标识符为标识符 global 时，则搜索全局命名空间（也是唯一的全局命名空间）以查找右标识符。例如：

```
class Program
{
   static void Main() {
      global::System.IO.Stream s = new global::MyLibrary.IO.EmptyStream();
      ...
   }
}
```

类似于将 :: 与 extern 和 using 别名一起使用，将 :: 与 global 标识符一起使用保证了名称查找不会受到引入新的类型和成员的影响。注意，标识符 global 仅在用作 :: 限定符的左标识符时才有特殊意义。它不是关键字，并且它本身不是别名。

25.3.1 限定的别名成员

qualified-alias-member 定义如下：

qualified-alias-member:
identifier :: *identifier* *type-argument-list*$_{opt}$

C# 词法语法中新增了 :: 标记。经过更新的 *operator-or-punctuator* 词法语法产生形式请参见第 20.10 节。

qualified-alias-member 可用作 *namespace-or-type-name*（第 20.9.1 节）或用作 *member-access*（第 20.9.6 节）中的左操作数。

qualified-alias-member 具有下列两种形式之一：

- N::I<A_1, …, A_K>，其中 N 和 I 表示标识符，<A_1, …, A_K> 为类型参数列表。（K 总是至少为 1。）
- N::I，其中 N 和 I 表示标识符。（在此情况下，K 视作 0。）

如果使用此表示法，*qualified-alias-member* 的含义按下面的过程确定：

- 如果 N 为标识符 global，则搜索全局命名空间以查找 I：
  - 如果全局命名空间包含名为 N 的命名空间，并且 K 为 0，则 *qualified-alias-member* 即表示该命名空间。
  - 否则，如果该全局命名空间包含名为 I 的非泛型类型，并且 K 为 0，则 *qualified-alias-member* 即表示该类型。
  - 否则，如果该全局命名空间包含名为 I 的带有 K 个类型参数的类型，则 *qualified-alias-member* 即表示使用给定的类型变量构造的该类型。
  - 否则，*qualified-alias-member* 即为未定义，并发生编译时错误。

- 否则，从直接包含 *qualified-alias-member* 的命名空间声明（第 9.2 节）开始（如果存在），持续处理每一个包含它的命名空间声明（如果存在），最后在含有 *qualified-alias-member* 的编译单元结束，在这一过程中将计算下列步骤直至找到某个实体：
  - 如果命名空间声明或编译单元包含将 N 与某个类型相关联的 *using-alias-directive*，则 *qualified-alias-member* 为未定义，并发生编译时错误。
  - 否则，如果命名空间声明或编译单元包含将 N 与某个命名空间相关联的 *extern-alias-directive* 或 *using-alias-directive*，则：
    - 如果与 N 关联的命名空间包含名为 I 的命名空间，并且 K 为 0，则 *qualified-alias-member* 即表示该命名空间。
    - 否则，如果与 N 关联的命名空间包含名为 I 的非泛型类型，并且 K 为 0，则 *qualified-alias-member* 即表示该类型。
    - 否则，如果与 N 关联的命名空间包含名为 I 的带有 K 个类型参数的类型，则 *qualified-alias-member* 即表示使用给定的类型变量构造的该类型。
    - 否则，*qualified-alias-member* 即为未定义，并发生编译时错误。
- 否则，*qualified-alias-member* 即为未定义，并发生编译时错误。

注意，将命名空间别名限定符与引用某个类型的别名一起使用将导致编译时错误。另请注意，如果标识符 N 为 global，则在全局命名空间中执行查找，即使存在将 global 与某个类型或命名空间关联的 using 别名。

25.3.2 别名的唯一性

在 C# 2.0 中，每个编译单元和命名空间体对于 extern 别名和 using 别名有着单独的声明空间。因此，虽然 extern 别名或 using 别名的名称在直接包含它们的编译单元或命名空间体中声明的 extern 别名和 using 别名集中必须唯一，但是允许别名与类型或命名空间同名，只要它仅与 :: 限定符连用。

在下面的示例中

```
namespace N
{
   public class A {}
   public class B {}
}
namespace N
{
   using A = System.IO;
   class X
   {
      A.Stream s1;         // Error, A is ambiguous
      A::Stream s2;        // Ok
   }
}
```

名称 A 在第二个命名空间体中有两种可能的含义，因为类 A 和 using 别名 A 都在范围中。因此，在限定名 A.Stream 中使用的 A 是不确定的，并会导致发生编译时错误。但是，将 A 与 :: 限定符连用则不是错误，因为将 A 只作为命名空间别名进行查找。

25.4 Extern 别名

直到现在，C# 只支持一个命名空间层次结构，来自被引用的程序集的类型以及当前程序置于其中。因为这种设计，不可能使用来自不同程序集的相同完全限定名来引用类型，当为多个类型分别赋予了相同的名称，或者当程序需要引用同一个程序集的多个版本时，会发生这种情况。extern 别名使得在这样的情况下创建和引用不同的命名空间层次结构成为可能。

请看下列两个程序集：

程序集 a1.dll：

```
namespace N
{
   public class A {}
   public class B {}
}
```

程序集 a2.dll：

```
namespace N
{
   public class B {}
   public class C {}
}
```

以及下面的程序：

```
class Test
{
   N.A a;          // Ok
   N.B b;          // Error
   N.C c;          // Ok
}
```

如果使用下面的命令行编译该程序：

```
csc /r:a1.dll /r:a2.dll test.cs
```

包含在 a1.dll 和 a2.dll 中的类型全部置于全局命名空间层次结构 ***(global namespace hierarchy)*** 中，并且因为两个程序集中都存在类型 N.B 而发生错误。使用 extern 别名则可以将包含在 a1.dll 和 a2.dll 中的类型置于不同的命名空间层次结构中。

下面的程序声明并使用两个 extern 别名 X 和 Y，每个别名代表根据包含在一个或多个程序集中的类型创建的不同命名空间层次结构的根。

```
extern alias X;
extern alias Y;

class Test
{
   X::N.A a;
   X::N.B b1;
   Y::N.B b2;
   Y::N.C c;
}
```

该程序声明存在 extern 别名 X 和 Y，但这些别名的实际定义在该程序的外部。如果使用命令行编译器，则在使用 /r 选项引用程序集时进行定义。在 Visual Studio 中，则通过在项目引用的一个或多个程序集的 Aliases 属性中包含别名来进行定义。命令行

```
csc /r:X=a1.dll /r:Y=a2.dll test.cs
```

将 extern 别名 X 定义为由 a1.dll 中的类型构成的命名空间层次结构的根，而将 Y 定义为由 a2.dll 中的类型构成的命名空间层次结构的根。同名的 N.B 类现在可分别通过 X.N.B 和 Y.N.B 引用，或者使用命名空间别名限定符通过 X::N.B 和 Y::N.B 引用。如果没有为程序声明的 extern 别名提供外部定义，则会发生错误。

一个 extern 别名可包含多个程序集，并且一个特定程序集可包含在多个 extern 别名中。例如，假定有以下程序集

程序集 a3.dll：

```
namespace N
{
   public class D {}
   public class E {}
}
```

命令行

```
csc /r:X=a1.dll /r:X=a3.dll /r:Y=a2.dll /r:Y=a3.dll test.cs
```

将 extern 别名 X 定义为由 a1.dll 和 a3.dll 中的类型构成的命名空间层次结构的根，而将 Y 定义为 a2.dll 和 a3.dll 中的类型构成的命名空间层次结构的根。因为存在此定义，所以通过 X::N.D 和 Y::N.D 都可以引用 a3.dll 中的类 N.D。

程序集可放置在全局命名空间层次结构中，即使它还包含在一个或多个 extern 别名中。例如，命令行

```
csc /r:a1.dll /r:X=a1.dll /r:Y=a2.dll test.cs
```

将程序集 a1.dll 同时放置在全局命名空间层次结构以及以 extern 别名 X 作为根的命名空间层次结构中。然后，类 N.A 可通过 N.A 或 X::N.A 引用。

通过将标识符 global 与命名空间别名限定符连用（如 global::System.IO.Stream），可以确保总是从全局命名空间层次结构的根开始查找。

using 指令可引用直接包容该指令的同一命名空间声明或编译单元中定义的 extern 别名。例如：

```
extern alias X;
using X::N;
class Test
{
   A a;           // X::N.A
   B b;           // X::N.B
}
```

25.4.1 Extern 别名指令

extern-alias-directive 引入了一个作为命名空间层次结构别名的标识符。

compilation-unit:
    extern-alias-directives$_{opt}$ *using-directives*$_{opt}$ *global-attributes*$_{opt}$
        namespace-member-declarations$_{opt}$

namespace-body:
    { *extern-alias-directives*$_{opt}$ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ }

extern-alias-directives:
    extern-alias-directive
    extern-alias-directives *extern-alias-directive*

extern-alias-directive:
    extern alias *identifier* ;

extern-alias-directive 的 *identifier* 在直接包含它的编译单元或命名空间体中声明的 extern 别名和 using 别名集中必须是唯一的。第 25.3.2 节对此有进一步描述。如果该 *identifier* 为单词 global 则会发生编译时错误。

extern-alias-directive 的范围扩展到直接包含它的编译单元或命名空间体内的所有 *using-directives*、*global-attributes* 和 *namespace-member-declarations*。在包含 *extern-alias-directive* 的编译单元或命名空间体中，由 *extern-alias-directive* 引入的标识符可用于引用具有别名的命名空间层次结构。

与 *using-alias-directive*（第 9.3.1 节）类似，*extern-alias-directive* 使别名可在特定的编译单元或命名空间体中使用，但是它不会为基础声明空间提供任何新成员。换言之，*extern-alias-directive* 是不可传递的，它仅影响它在其中出现的编译单元或命名空间体。

解析 *using-alias-directive* 所引用的 *namespace-or-type-name* 不受 *using-alias-directive* 本身以及直接包含它的编译单元或命名空间体内的其他 *using-directive* 的影响，但是可能受到直接包含它的编译单元或命名空间体内的 *extern-alias-directives* 的影响。换言之，对 *using-alias-directive* 的 *namespace-or-type-name* 的解析就好像在直接包含该指令的编译单元或命名空间体中并没有 *using-directive*，但是有正确的 *extern-alias-directive* 集合。在下面的示例中

```
namespace N1.N2 {}

namespace N3
{
   extern alias E;
   using R1 = E.N;      // OK
   using R2 = N1;       // OK
   using R3 = N1.N2;    // OK
   using R4 = R2.N2;    // Error, R2 unknown
}
```

最后一个 *using-alias-directive* 导致编译时错误，原因是它不受第一个 *using-alias-directive* 的影响。第一个 *using-alias-directive* 不会产生错误，因为 extern 别名 E 的范围包括 *using-alias-directive*。

25.5 Pragma 指令

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

pp-directive:
    ...
    pp-pragma

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

pragma-body:
    pragma-warning-body

C# 2.0 提供 #pragma 指令以控制编译器警告。此语言将来的版本可能包含更多的 #pragma 指令。为了确保与其他 C# 编译器的互操作性，Microsoft C# 编译器对于未知的 #pragma 指令不会发出编译错误；但是这类指令确实会生成警告。

25.5.1 Pragma warning

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

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

warning-action:
    disable
    restore

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

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

#pragma warning disable 指令禁用所有警告或指定警告。

#pragma warning restore 指令将所有警告或指定警告恢复为在编译单元的开始处有效的状态。注意，如果在外部禁用了某条特定警告（例如使用命令行编译器的 /nowarn 选项），则 #pragma warning restore（无论是针对所有警告还是特定警告）不会重新启用该警告。

下面的示例演示如何使用 #pragma warning 临时禁用在引用已过时的成员时出现的警告。

```
using System;

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

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

25.6 默认值表达式

默认值表达式用于获取某个类型的默认值（第 5.2 节）。通常，默认值表达式用于类型参数，因为可能并不知道类型参数是值类型还是引用类型。（不存在从 null 文本到类型参数的转换，除非类型参数已知为引用类型。）

primary-no-array-creation-expression:
    ...
    default-value-expression

default-value-expression:
    default (*type*)

如果 *default-value-expression* 中的 *type* 在运行时为引用类型，则结果将转换到该类型的 null。如果 *default-value-expression* 中的 *type* 在运行时为值类型，则结果将为该 *value-type* 的默认值（第 4.1.2 节）。

如果类型为引用类型或者是已知为引用类型的类型参数（第 20.7 节），则 *default-value-expression* 为常数表达式（第 7.15 节）。此外，如果类型为以下任一值类型，则 *default-value-expression* 为常数表达式：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool 或任何枚举类型。

25.7 条件属性类

以 Conditional 属性修饰的属性类（第 17.1 节）称为条件属性类 ***(conditional attribute class)***。Conditional 属性通过测试条件编译符号来指示条件。条件属性类的属性规范（第 17.2 节）既可以包括在内，也可省略，具体取决于此符号是否是在规范所在位置定义的。如果定义了该符号，则属性规范包括在内；否则省略属性规范。

下面的示例

```
#define DEBUG

using System;
using System.Diagnostics;

[Conditional("DEBUG")]
public class TestAttribute : Attribute {}

[Test]
class C {}
```

将属性类 TestAttribute 声明为条件属性类，并随后将 Test 属性应用于 Class1。由于定义了条件编译符号 DEBUG，如果在运行时检索应用于 Class1 的属性，将导致 TestAttribute 类被实例化。如果未定义符号 DEBUG，则在运行时检索应用于 Class1 的属性不会对 TestAttribute 类进行实例化。

注意包含或排除条件属性类的属性专用化是由该专用化所在位置的条件编译符号控制的，这一点很重要。在下面的示例中

文件 test.cs：

```
using System;
using System.Diagnostics;

[Conditional(“DEBUG”)]
public class TestAttribute : Attribute {}
```

文件 class1.cs：

```
#define DEBUG

[Test]            // TestAttribute is included
class Class1 {}
```

文件 class2.cs：

```
#undef DEBUG

[Test]            // TestAttribute is excluded
class Class2 {}
```

类 Class1 和 Class2 都用属性 Test 进行了修饰，而该属性是基于是否定义了 DEBUG 的条件属性。因为此符号是在 Class1 而不是在 Class2 的上下文中定义的，所以在 Class1 处定义的 Test 属性规范被包括在内，而在 Class2 处定义的 Test 属性规范则省略。

25.8 固定大小缓冲区

固定大小缓冲区用于将“C 样式”的内联数组声明为结构的成员。固定大小缓冲区主要用来与非托管 API 交互。固定大小缓冲区是不安全的功能，并且只能在不安全上下文（第 18.1 节）中声明。

25.8.1 固定大小缓冲区的声明

固定大小缓冲区 ***(fixed size buffer)*** 是一个成员，表示给定类型的变量的固定长度缓冲区的存储区。固定大小缓冲区声明引入了给定元素类型的一个或多个固定大小缓冲区。仅允许在结构声明中使用固定大小缓冲区，且只能出现在不安全上下文（第 18.1 节）中。

struct-member-declaration:
    ...
    fixed-size-buffer-declaration

fixed-size-buffer-declaration:
    attributes$_{opt}$ *fixed-size-buffer-modifiers*$_{opt}$ `fixed` *buffer-element-type*
        fixed-sized-buffer-declarators `;`

fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-sized-buffer-modifier *fixed-size-buffer-modifiers*

fixed-size-buffer-modifiers:
    `new`
    `public`
    `protected`
    `internal`
    `private`
    `unsafe`

buffer-element-type:
    type

fixed-sized-buffer-declarators:
    fixed-sized-buffer-declarator
    fixed-sized-buffer-declarator *fixed-sized-buffer-declarators*

fixed-sized-buffer-declarator:
    identifier `[` *const-expression* `]`

固定大小缓冲区声明可包括一组属性（第 17 节）、一个 `new` 修饰符（第 10.2.2 节）、四个访问修饰符（第 10.2.3 节）的一个有效组合和一个 `unsafe` 修饰符（第 18.1 节）。这些属性和修饰符适用于由固定大小缓冲区声明所声明的所有成员。同一个修饰符在一个固定大小缓冲区声明中出现多次是一个错误。

固定大小缓冲区声明不允许包含 `static` 修饰符。

固定大小缓冲区声明的缓冲区元素类型指定了由该声明引入的缓冲区的元素类型。缓冲区元素类型必须为下列预定义类型之一：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double` 或 `bool`。

缓冲区元素类型后跟一个固定大小缓冲区声明符的列表，该列表中的每个声明符引入一个新成员。固定大小缓冲区声明符由一个用于命名成员的标识符以及标识符后面由 `[` 和 `]` 标记括起来的常数表达式所组成。该常数表达式表示在由该固定大小缓冲区声明符引入的成员中的元素数量。该常数表达式的类型必须可隐式转换为类型 `int`，并且该值必须是非零的正整数。

固定大小缓冲区的元素保证在内存中按顺序放置。

声明多个固定大小缓冲区的固定大小缓冲区声明相当于带有相同属性和元素类型的多个固定大小缓冲区的声明。例如

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

相当于

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
   public fixed int z[100];
}
```

25.8.2 表达式中的固定大小缓冲区

固定大小缓冲区成员的成员查找（第 7.3 节）过程与字段的成员查找完全相同。

可使用 *simple-name*（第 7.5.2 节）或 *member-access*（第 7.5.4 节）在表达式中引用固定大小缓冲区。

当固定大小缓冲区成员作为简单名称被引用时，其效果与 `this.I` 形式的成员访问相同，其中 `I` 为固定大小缓冲区成员。

在 `E.I` 形式的成员访问中，如果 `E` 为结构类型，并且在该结构类型中通过 `I` 的成员查找标识了一个固定大小成员，则如下计算并归类 `E.I`：

- 如果表达式 `E.I` 不属于不安全上下文，则发生编译时错误。
- 如果 `E` 归类为值类别，则发生编译时错误。
- 否则，如果 `E` 为可移动变量（第 18.3 节）并且表达式 `E.I` 不是 *fixed-pointer-initializer*（第 18.6 节），则发生编译时错误。
- 否则，`E` 引用固定变量，并且该表达式的结果为指向 `E` 中的固定大小缓冲区成员 `I` 的第一个元素的指针。结果为类型 `S*`，其中 `S` 为 `I` 的元素类型，并且归类为值。

可使用指针操作从第一个元素开始访问固定大小缓冲区的后续元素。与访问数组不同，访问固定大小缓冲区的元素是不安全操作，并且不进行范围检查。

下面的示例声明并使用了一个包含固定大小缓冲区成员的结构。

```
unsafe struct Font
{
   public int size;
   public fixed char name[32];
}

class Test
{
   unsafe static void PutString(string s, char* buffer, int bufSize) {
      int len = s.Length;
      if (len > bufSize) len = bufSize;
      for (int i = 0; i < len; i++) buffer[i] = s[i];
      for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
   }
```

```
   unsafe static void Main()
   {
      Font f;
      f.size = 10;
      PutString("Times New Roman", f.name, 32);
   }
}
```

25.8.3 Fixed 语句

fixed 语句（第 18.6 节）扩展为允许 *fixed-pointer-initializer* 成为引用可移动变量的固定大小缓冲区成员的 *simple-name* 或 *member-access*。此形式的固定指针初始值设定项产生一个指向固定大小缓冲区（第 25.8.2 节）第一个元素的指针，并且该固定大小缓冲区保证在 fixed 语句的持续时间内保留在此固定地址。

例如：

```
unsafe struct Font
{
   public int size;
   public fixed char name[32];
}

class Test
{
   unsafe static void PutString(string s, char* buffer, int bufSize) {
      int len = s.Length;
      if (len > bufSize) len = bufSize;
      for (int i = 0; i < len; i++) buffer[i] = s[i];
      for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
   }

   Font f;

   unsafe static void Main()
   {
      Test test = new Test();
      test.f.size = 10;
      fixed (char* p = test.f.name) {
         PutString("Times New Roman", p, 32);
      }
   }
}
```

25.8.4 明确赋值检查

固定大小缓冲区不接受明确赋值检查（第 5.3 节），并且为了对结构类型变量进行明确赋值检查，忽略固定大小缓冲区成员。

当包含固定大小缓冲区成员的最外层结构变量为静态变量、类实例的实例变量或数组元素时，该固定大小缓冲区的元素自动初始化为其默认值（第 5.2 节）。而在所有其他情况下，固定大小缓冲区的初始内容未定义。