**.Net Framework**

目录

[泛型(Generic) 2](#_Toc319065692)

[什么是泛型 2](#_Toc319065693)

[泛型的优势: (示例: GenericAdvantage) 2](#_Toc319065694)

[CLR类库中提供的一些泛型: (示例: CLRGeneric). 2](#_Toc319065695)

[泛型类型: 3](#_Toc319065696)

[开放类型和封闭类型. (示例: OpenCloseGenericClass). 4](#_Toc319065697)

[泛型接口: 5](#_Toc319065698)

[泛型委托: 5](#_Toc319065699)

[泛型方法: (示例: GenericMethod) 5](#_Toc319065700)

[泛型的可验证性和约束. 6](#_Toc319065701)

[主要约束: (示例: PrimaryConstraint). 6](#_Toc319065702)

[次要约束: (示例: SecondaryConstraint). 7](#_Toc319065703)

[构造器约束: 8](#_Toc319065704)

[Framework 3.0 语法的提高: 9](#_Toc319065705)

[局部变量的隐式类型. (Implicitly typed local variables). 9](#_Toc319065706)

[语法 9](#_Toc319065707)

[类型初始化(Object Initializers) 10](#_Toc319065708)

[集合初始化 11](#_Toc319065709)

[Lambda表达式: 12](#_Toc319065710)

[回顾委托(delegate). 12](#_Toc319065711)

[匿名方法(Anonymous Methods) 14](#_Toc319065712)

[引入Lambda表达式(Lambda Expressions) 15](#_Toc319065713)

[比较Lambda表达式和匿名方法. 15](#_Toc319065714)

[怎么使用Lambda表达式: 15](#_Toc319065715)

[内置委托类型 16](#_Toc319065716)

[扩展方法(Extension Method) 18](#_Toc319065717)

[定义简单的扩展方法. 18](#_Toc319065718)

[内置的一些扩展方法. 20](#_Toc319065719)

[扩展方法的限制: 21](#_Toc319065720)

[匿名类型(Anonymous Type). 22](#_Toc319065721)

[使用匿名类型把数据集合到对象中. 22](#_Toc319065722)

[没有名称的类型. 23](#_Toc319065723)

[用匿名类型提高我们的示例 24](#_Toc319065724)

[匿名对象的限制. 24](#_Toc319065725)

# 泛型(Generic)

## 什么是泛型

泛型是CLR和编程语言提供的特殊机制, 它支持另一种形式的代码重用,即 “算法重用”.

简单的说就是: 在算法定义的时候并不知道所操作的类型, 在使用的时候才确定操作类型.

因为这个我们所操作的类型在定义时不知道, 所以我们需要定义一个或者多个类型参数以供使用时传入. 类型参数一般在<> 中间定义, 比如: <T>, <TKey, TValue>

## 泛型的优势: (示例: GenericAdvantage)

1. 类型安全:
2. 更加清晰的代码.
3. 更佳的性能.
4. 源代码保护.

## CLR类库中提供的一些泛型: (示例: CLRGeneric).

1. 泛型类型: List<T>, Comparer<T>, Dictionary<TKey, TValue> 等.

|  |
| --- |
| private List<int> \_lstInt;  private Comparer<int> \_comInt;  private Dictionary<int, string> dicInt; |

1. 泛型接口: IList<T>, IEnumerable<T>, ICollection<T>, IDictionary<TKey, TValue>等.

|  |
| --- |
| private IList<int> \_ilstInt;  private ICollection<int> \_iclInt;  private IEnumerable<int> \_ienumInt;  private IDictionary<int, string> \_idicInt; |

1. 泛型方法: Array.Sort<T>(), Array.BinarySearch<T>(). 等.

|  |
| --- |
| int[] arr = new int[1];  Array.Sort<int>(arr);  Array.BinarySearch(arr, 1); |

1. 泛型代理: Action<T>, Predicate<T>, Fun<T, TResult> 等.

|  |
| --- |
| private Action<int> \_act;  private Predicate<int> \_pre;  private Func<int, string> \_func; |

## 泛型类型:

泛型类型的定义

|  |
| --- |
| public class Node<T>  {  private T \_data;  private Node<T> \_next;  public Node(T data): this(data, null)  { }  public Node(T data, Node<T> next)  {  \_data = data;  \_next = next;  }  public override string ToString()  {  return \_data.ToString() + \_next ?? \_next.ToString();  }  } |

泛型类型的继承：

|  |
| --- |
| public class IntNode : Node<int>  {  public IntNode(int data): base(data)  { }  public IntNode(int data, IntNode next) : base(data, next)  { }  } |

泛型类型的使用：

|  |
| --- |
| Node<char> head = new Node<char>('C');  head = new Node<char>('b', head);  head = new Node<char>('a', head);  Console.WriteLine(head.ToString()); |

Note: 有的时候泛型会将开发人员弄糊涂. 因为代码中可能散布着大量的”<>”, 影响可读性, 有的时候大家会定义一个新的非泛型类型,它是一个派生, 并制定了所有的类型实参. 也可以用Using来实现.

### 开放类型和封闭类型. (示例: OpenCloseGenericClass).

开放类型: 具有泛型类型参数的类型称开放类型, Dictionary<TKey,TValue>就是一个开放类型，以下的子类也是开放类型。

|  |
| --- |
| public class DictionaryStringKey<TValue>: Dictionary<string, TValue>  {  } |

CLR禁止构造开发类型的任何实例. 以下代码会报错：“无法创建 GenericLib.DictionaryStringKey`1[TValue] 的实例，因为 Type.ContainsGenericParameters 为 True”。

|  |
| --- |
| object o = Activator.CreateInstance(typeof(DictionaryStringKey<>)); |

封闭类型: 为所有类型参数传递的都是实际数据类型：

|  |
| --- |
| public class DictionaryStringKeyValue : Dictionary<string, string>  {  } |

## 泛型接口:

定义泛型接口.

|  |
| --- |
| public interface INode<T>  {  } |

## 泛型委托:

定义泛型委托:

|  |
| --- |
| public delegate TResult CallMe<in T, out TResult>(T arg) |

注意：建议尽量使用在FCL中[内置委托类型](#_内置委托类型)：Action, Predicate 和Func.

## 泛型方法: (示例: GenericMethod)

泛型方法的定义.

|  |
| --- |
| public TOutput Converter<TOutput>(T input)  {  TOutput result = (TOutput)Convert.ChangeType(input, typeof(TOutput));  return result;  }  public static void swap<T>(ref T o1, ref T o2)  {  T temp = o1;  o1 = o2;  o2 = temp;  } |

泛型方法的使用：

|  |
| --- |
| int n1 = 1;  int n2 = 2;  Console.WriteLine("n1 = {0}, n2 = {1}", n1, n2);  swap<int>(ref n1, ref n2);  Console.WriteLine("n1 = {0}, n2 = {1}", n1, n2); |

类型推断: 编译器会在调用一个泛型方法时自动判断(或者推断)要使用的类型：

|  |
| --- |
| int n1 = 1;  int n2 = 2;  Console.WriteLine("n1 = {0}, n2 = {1}", n1, n2);  swap(ref n1, ref n2);  Console.WriteLine("n1 = {0}, n2 = {1}", n1, n2); |

## 泛型的可验证性和约束.

约束: 约束的作用是限制能制定成泛型实参的类型数量, 通过限制类型的数量, 我们可以对那些类型执行更多的操作.

### 主要约束: (示例: PrimaryConstraint).

类型参数可以指定零个或者一个主要约束, 主要约束可以是一个应用类型.

主要约束的定义：

|  |
| --- |
| public class PrimaryConstraintOfStream<T>  where **T: Stream**  {  public void CloseStream(T stream)  {  stream.Close();  }  } |

|  |
| --- |
| public class PrimaryConstraintOfClass<T>  where **T : class**  {  public void M()  {  T temp = null;  }  } |

|  |
| --- |
| public class PrimaryConstraintOfStruct<T>  where **T : struct**  {  public void M()  {  T t = new T();  }  } |

### 次要约束: (示例: SecondaryConstraint).

类型参数可以指定零个或者多个次要约束, 次要约束有2种.

接口约束: 指定一个接口约束时, 向编译器承诺一个指定的类型实参是实现了接口的一个类型. 因为支持指定多个接口约束, 所以类型实参必须实现所有的接口.

|  |
| --- |
| public class SecondaryConstraintOfInterface<T>  where **T: IComparable, IList**  {  public void M()  {  Object o = null;  T test = default(T);  test.CompareTo(o);  test.Clear();  }  } |

类型参数约束: 是在指定的类型实参之间, 必须存在一个关系. 一个类型实参可以指定零个或者多个参数约束.

|  |
| --- |
| public static List<TBase> ConvertList<T, TBase>(IList<T> list)  where **T: TBase**  {  List<TBase> lstBase = new List<TBase>(list.Count);  foreach (T t in list)  {  lstBase.Add(t);  }  return lstBase;  } |

类型参数约束的应用：

|  |
| --- |
| List<string> ls = new List<string>();  IList<object> lo = ConvertList<string, object>(ls);  IList<IComparable> lc = ConvertList<string, IComparable>(ls);  IList<IComparable<string>> lcs = ConvertList<string, IComparable<string>>(ls); |

### 构造器约束:

一个类型参数可以指定零个或者一个构造器约束. 指定构造器约束相当于向编译器承诺一个指定的类型实参是实现了公共无参的一个非抽象类型.

|  |
| --- |
| public class StructConstraint<T>  where **T : new()**  {  public T NewT()  {  return new T();  }  public TOut NewTout<TOut>()  where TOut : new()  {  return new TOut();  }  } |

泛型类型变量设为默认值. Default(T),

|  |
| --- |
| public TOut NewTout<TOut>()  where TOut  {  return **default(TOut);**  } |

# Framework 3.0 语法的提高:

.net framework 3.0 在语法上提供了很多的改进, 主要的改进包括:

* 局部变量的隐式类型. (Implicitly typed local variables).
* 类的初始化. (Object Initializers)
* Lambda表达式. (Lambda Expression)
* 扩展方法. (Extension Methods)
* 匿名类型. (Anonymous Type).

我们将定义ProcessData类供后面的示例使用

|  |
| --- |
| class ProcessData  {  public Int32 Id { get; set; }  public Int64 Memory { get; set; }  public String Name { get; set; }  } |

# 局部变量的隐式类型. (Implicitly typed local variables).

示例：**ImplicitlyType**

在C# 3.0 中提供一个关键字 var. 该关键字允许你声明一个不需要指定特定类型的变量. 编译器会在编译时根据给该变量赋值时的表达式来确定变量类型.

## 语法

var 关键字非常容易使用, 在var后面跟变量名以及赋值的表达式, 让我们比较一下使用var 的隐式类型的代码和不使用var:

|  |
| --- |
| var i = 12;  var s = "Hello";  var d = 1.0;  var numbers = new int[] { 1, 2, 3 };  var o = new object();  var dic = new Dictionary<int, string>(); |

|  |
| --- |
| int i = 12;  string s = "Hello";  double d = 1.0;  int[] numbers = new int[] { 1, 2, 3 };  object o = new object();  Dictionary<int, string> dic = new Dictionary<int, string>(); |

以上的隐式类型和非隐式类型在使用时是一样的, 没有区别.

使用隐式类型, 我们不需要写2遍指定的类型, 编译器会给该变量指定类型, 并且这个类型是强类型.

在定义一个[匿名类型(anonymous type)](#_匿名类型.)的变量中, 我们只能使用var来定义变量.

# 类型初始化(Object Initializers)

示例：**ObjectInitializer**

类型初始化允许我们在一个语句中给一个或者多个字段或者属性赋值. 这可以用来定义所有的类. 让我们回顾一下类型初始化的发展: 首先, C#提供很简单的方式初始化主要类型和数组.

|  |
| --- |
| int i = 12;  string s = "Hello";  string[] names = new string[] { "Linq", "in", "Action" }; |

但是我们没有简单的方式去构造或者初始化其他的类型. 我们需要写一下代码.

|  |
| --- |
| ProcessData data = new ProcessData();  data.Id = 123;  data.Name = "LinqInAction";  data.Memory = 1024 \* 1024; |

在C# 3.0 后 ,我们可以使用类型初始化来简化这种类的构造和初始化:

|  |
| --- |
| ProcessData data = new ProcessData {  Id = 123, Name = "LinqInAction", Memory = 1024 \* 1024 }; |

上面2段使用或者不使用类型初始化的代码, 都会编译成同一个中间代码, 实现的功能是相同的.

如果该类提供了带参数的构造函数, 我们同样可以使用类型初始化. 以下 2段代码的效果是一样的:

|  |
| --- |
| throw new Exception("message") { Source = "Linq In Action" }; |

|  |
| --- |
| var exception = new Exception("Message");  exception.Source = "Linq In Action";  throw exception; |

# 集合初始化

C#3.0 提供集合的初始化, 只要该集合是继承System.Collections.IEnumerable和提供了合适的Add 方法.

|  |
| --- |
| List<int> lstInt = new List<int> { 1, 2, 3, 4, 5 }; |

以上代码实现的效果如下:

|  |
| --- |
| List<int> lstInt = new List<int>();  lstInt.Add(1);  lstInt.Add(2);  lstInt.Add(3);  lstInt.Add(4);  lstInt.Add(5); |

类型初始化和集合初始化在同一段代码中混合使用非常有用.你可以使用初始化写出更精简的代码, 如下:

|  |
| --- |
| var lstProcesses = new List<ProcessData> {  new ProcessData { Id = 1, Name = "devenv" },  new ProcessData { Id = 2, Name = "firefox" }  }; |

|  |
| --- |
| ProcessData process;  List<ProcessData> lstProcesses = new List<ProcessData>();  process = new ProcessData();  process.Id = 1;  process.Name = "devenv";  lstProcesses.Add(process);  process = new ProcessData();  process.Id = 2;  process.Name = "firefox";  lstProcesses.Add(process); |

类型初始化的优势:

* 我们给只有一个构造函数的类提供初始化.
* 我们不需要给简单的类提供构造函数.
* 我们不需要为了初始化类的不同属性而提供很多个构造函数

# Lambda表达式:

示例：**Lambda**

Lambda表达式来源于Lambda计算, 很多其他的语言都支持Lambda表达式. 下面让我们根据C#的语言的发展历程来一步一步揭开Lambda表达式的面纱.

示例说明: 我们将引用: System.Diagnostics 命名空间, 使用一下代码获取所有系统运行的进程

|  |
| --- |
| Process[] arrProcess = Process.GetProcesses(); |

## 回顾委托(delegate).

我们需要显示内存中满足一定条件的进程ID, 名称, 以及所占内存, 我们定义了一个方法DisplayProcesses, 并且我们把筛选条件硬编码在方法中:

|  |
| --- |
| static void DisplayProcesses()  {  List<ProcessData> processes = new List<ProcessData>();  foreach (Process process in Process.GetProcesses())  {  // 筛选条件  if (**process.WorkingSet64 >= 20 \* 1024 \* 1024**)  {  processes.Add(new ProcessData {  Id = process.Id, Name = process.ProcessName, Memory = process.WorkingSet64 });  }  }  // Print out the list of processes to the console  } |

WorkingSet64是系统为该进程分配的物理内存, 所以以上的函数显示的是所有物理内存大于20M的进程.

为了使该函数更具有通用化, 我们需要把筛选条件作为参数传给这个方法, 而不是使用硬编码, 在C#2.0或者之前, 我们很容易想到使用委托(delegate). 委托是一个类型, 存储指向函数的指针.

我们的筛选器需要提供一个Process对象作为参数, 返回一个布尔型来标识这个Process是否满足一定的规则, 以下是定义的委托:

|  |
| --- |
| public delegate bool FilterDelegate(Process process); |

我们可以使用C#2.0 提供的内置委托对象Predicate<T>来代替我们自定义委托对象. 以下是Predicate<T> 的定义:

|  |
| --- |
| public delegate bool Predicate<T>(T obj); |

Predicate<T>返回布尔值”true”或者”false”. 提供一个泛型参数, 所以我们在使用时必须指定泛型的具体类型, 在我们这个示例中, 我们使用的委托类型是Predicate<Process>.

我们将修改DisplayPrecesses方法,提供一个Predicate<Process>参数.

|  |
| --- |
| static void DisplayProcesses(**Predicate<Process> match**)  {  List<ProcessData> processes = new List<ProcessData>();  foreach (Process process in Process.GetProcesses())  {  if (**match(process))** // 筛选条件  {  processes.Add(new ProcessData {  Id = process.Id, Name = process.ProcessName,  Memory = process.WorkingSet64 });  }  }  // Print out the list of processes to the console  } |

现在我们DisplayProcesses可以支持任何的筛选条件, 筛选函数只要满足参数是Process返回值是布尔值就行,

|  |
| --- |
| static bool Filter(Process process)  {  return process.WorkingSet64 > 20 \* 1024 \* 1024;  } |

我们只要把筛选函数作为参数传给DisplayProcesses方法就行.

|  |
| --- |
| DisplayProcesses(Filter); |

## 匿名方法(Anonymous Methods)

委托存在于C#1.0. 在C#2.0 中, 我们可以使用匿名方法对委托进行赋值. 匿名方法允许你写更少的代码和不为一个方法命名.

是想一下匿名方法, 我们可以不需要为筛选条件声明一个函数Filter, 我们可以直接写筛选条件.

|  |
| --- |
| DisplayProcesses(delegate(Process process)  {  return process.WorkingSet64 >= 20 \* 1024 \* 1024;  }); |

C#2.0 给List<T> 和 Array类提供了一些支持匿名方法作为参数的方法. 这些方法包含: Find, FindAll和ForEach. 我们可以使用更少的方法去操作列表和数组.

示例: 我们可以利用Find找到visualStudio的进程.

|  |
| --- |
| Process visualStudio = Array.Find(Process.GetProcesses(),  delegate(Process p){  return p.ProcessName == "devenv";  }); |

## 引入Lambda表达式(Lambda Expressions)

C# 3.0 可以使用Lambda表达式来代替匿名方法, 上述的例子可以改成:

|  |
| --- |
| DisplayProcesses(process => process.WorkingSet64 >= 20 \* 1024 \* 1024); |

我们可以看到代码是非常的简洁. 这个代码可以阅读为: “给一个process, 如果分配的物理内存大于等于20MB, 则返回true”. 我们可以看到使用Lambda表达式时, 我们不需要提供类型, 因为这个类型信息是重复的, 编译器可以根据方法签名推断出这个参数的类型.

## 比较Lambda表达式和匿名方法.

在C#2.0中引入的匿名方法运行我们在委托出现的地方直接写代码, 匿名方法的语法基本和平时的一样. 相比的, Lambda表达式语法更加简洁, 更有表达力.

Lambda表达式提供以下优势:

* Lambda表达式可以忽略参数类型. 允许省略.
* Lambda表达式提供代码块或者表达式体. 而匿名方法只提供代码块.
* Lambda的表达式体可以转换为表达式树(Expression Tree), MSDN: <http://msdn.microsoft.com/zh-cn/library/bb397951.aspx> .

Lambda表达式引入了新的语法.

## 怎么使用Lambda表达式:

在C#中, 一个Lambda表达式的书写方式是: 写一个参数列表, 紧跟着 => 关键字, 最后是一个表达式或者代码块.

|  |
| --- |
| Process => process.WorkingSet64 >= 20 \* 1024 \* 1024  (传入参数) (Lambda操作符) (表达式或者代码块) |

*注意: => 是Lambda表达式的操作符, 不要和 >= 或者 <= 混淆*

Lambda操作符可以被读成”转化为”(goes to). 操作符左边是任何传入参数, 右边是计算用的表达式和代码块.

我们有2种Lambda表达式, 操作符右边是表达式的我们称作表达式Expression Lambda, 在操作符右边的是代码块的称为Statement Lambda.

请参考以下示例来属性Lambda表达式:

|  |
| --- |
| // 隐式类型, 表达式体  x => x + 1; // Implicitly typed, expression body  // 隐式类型, 代码体  x => { return x + 1; }; // Implicitly typed, statement body  // 显式类型, 表达式提  (int x) => x + 1; // Explicitly typed, expression body  // 显式类型, 代码提  (int x) => { return x + 1; }; // Explicitly typed, statement body  // 多个参数  (x, y) => x \* y; // Multiple parameters  // 没有参数, 表达式体  () => 1; // No parameters, expression body  // 没有参数, 代码体  () => Console.WriteLine(); // No parameters, statement body |

*注意: Lambda表达式的参数可以是显式类型或者隐式类型.*

## 内置委托类型

我们可以看到, Lambda表达式最后是编译成委托的. C#给我们提供了很多委托使我们更方便的使用Lambda表达式.

System.Core.dll提供了一下几组委托供用户使用:

Action<>:

|  |
| --- |
| public delegate void Action();  public delegate void Action<T1 >(T arg);  public delegate void Action<T1, T2 >(T1 arg1, T2 arg2);  public delegate void Action<T1, T2, T3 >(T1 arg1, T2 arg2, T3 arg3);  public delegate void Action<T1, T2, T3, T4>(T1 arg1, T2 arg2, T3 arg3, T4 arg4); |

*注意: 在VS2010中, Action的参数支持到16个.*

Predicate<>:

|  |
| --- |
| public delegate bool Predicate<T>(T obj); |

Func<>:

|  |
| --- |
| public delegate TResult Func<TResult>();  public delegate TResult Func<T, TResult>(T1 arg1);  public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);  public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2, T3 arg3);  public delegate TResult Func<T1, T2, T3, T4, TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4); |

*注意: 在VS2010中, Func的参数支持到16个.*

Lambda表达式编译成委托必须满足以下条件:

* Lambda表达式包含的参数个数必须和委托包含的参数个数相同,
* Lambda表达式中的参数必须可以隐式转换为对应的委托的参数.
* Lambda表达式中的返回值必须可以隐式转换为委托的返回值.

为了更好的理解Lambda表达式和委托的结合, 我们提供了一下是的示例:

|  |
| --- |
| // lambda 表达式不包含参数  Func<DateTime> getDateTime = () => DateTime.Now;  // lambda 表达式: 包含隐式类型的参数  Action<string> printImplicit = s => Console.WriteLine(s);  // lambda 表达式: 包含显式类型的参数  Action<string> printExplicit = (string s) => Console.WriteLine(s);  // lambda 表达式: 包含2个隐式类型的参数  Func<int, int, int> sumInts = (x, y) => x + y;  // Predicate<T> 和 Func<T, Boolean> 是等价的 (但是不兼容)  Predicate<int> equalsOne1 = x => x == 1;  Func<int, bool> equalsOne2 = x => x == 1;  // Lambda表达式包含表达式体.  Func<int, int> incInt = x => x + 1;  Func<int, double> incIntAsDouble = x => x + 1;    // Lambda表达式包含显式类型参数以及代码体.  Func<int, int, int> comparer = (int x, int y) =>  {  if (x > y) return 1;  if (x < y) return -1;  return 0;  }; |

# 扩展方法(Extension Method)

示例：**ExtensionMethod**

扩展方法是C#3.0 提供的新的语法, 可以给已经定义好的类型增加新的方法.

## 定义简单的扩展方法.

让我们回到之前显式系统进程的示例, 现在我们有一个新的需求: 计算一组进程所占用的内存量. 我们可以使用之前所定义的ProcessData. 我们可以定义一个以一组进程作为参数的今天方法, 这个方法循环所有的进程, 计算总内存量:

|  |
| --- |
| static Int64 TotalMemory(IEnumerable<ProcessData> processes)  {  Int64 result = 0;  foreach (var process in processes)  result += process.Memory;  return result;  } |

所以我们可以使用这个静态方法得到内存总量:

|  |
| --- |
| Console.WriteLine("Total memory: {0} MB",  TotalMemory(processes) / 1024 / 1024); |

在C#3.0以后, 我们可以做的提高就是把这个静态方法转换为扩展方法.

首先定义一个扩展方法:

|  |
| --- |
| static Int64 TotalMemory(**this** IEnumerable<ProcessData> processes)  {  Int64 result = 0;  foreach (var process in processes)  result += process.Memory;  return result;  } |

相比较前面定义的静态方法, 这个方法多了一个关键字**this**. This关键字提示编译器该方法是扩展方法, 所扩展的类型是IEnumerable<ProcessData>.

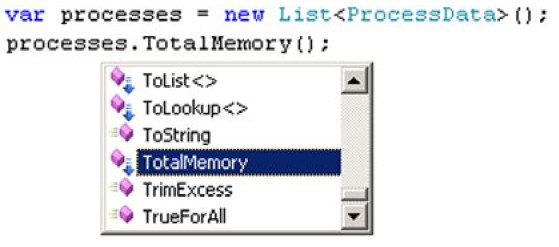
***注意: 在C#中, 扩展方法必须定义在静态类中, 扩展方法支持任意数量的参数, 但是第一个参数必须加this关键字, 用来指明所扩展的类型.***

我们可以在任何IEnumerable<ProcessData>类型的实例中使用扩展方法.

|  |
| --- |
| Console.WriteLine("Total memory: {0} MB",  processes.TotalMemory() / 1024 / 1024); |

我们怎么做到扩展的呢, 其实这个类型并没有做任何改变. 编译器把这个代码编译成了静态方法, 和我们之前使用的静态方法是相同的.

虽然看起来扩展方法并没有做任何改变, 但是在写代码中, 智能感知会提供这个扩展方法.



一个蓝色的向下箭头标志用来标识扩展方法. 现在我们写代码时很容易发现TotalMemory方法. 所以扩展方法在智能感知上比类的静态方法更有优势.

扩展方法的另一个优势是可以很简单的使用链锁操作.

我们有以下的需求:

1. 从ProcessData列表中筛选出一些ProcessData.
2. 计算出总内存量.
3. 把总内存量从B转换为MB.

如果使用静态方法:

|  |
| --- |
| BytesToMegaBytes(TotalMemory(FilterSomeProcess(processes))); |

可以看出这段代码写法和执行顺序是相反的(从后面执行到前面的), 这样更加难以阅读和理解.

而相比较使用扩展方法:

|  |
| --- |
| Processes.FilterSomeProcess().TotalMemory().BytesToMegaBytes(); |

这个代码的写法和执行顺序是一致的, 这样是不是更容易阅读和理解.

## 内置的一些扩展方法.

C# 3.0 提供了很多扩展方法供使用. 在这里我们选取System.Linq.Enumerable类中的一些扩展方法做一些简单的介绍(这些扩展方法扩展的类是IEnumerable<T>).

OrderByDescending:

该扩展方法是用来做逆序排序的, 方法的参数是一个Func<T, TResult>, 现在我们需要对进程列表按照使用的内存进行倒序排序.

|  |
| --- |
| IEnumerable<ProcessData> orderProcesses = processes.**OrderByDescending(process => process.Memory)**; |

Take:

当我们需要获取内存最大的2个进程时, 我们需要用Take扩展函数, 该函数用来获取列表中的前几项.

|  |
| --- |
| IEnumerable<ProcessData> orderProcesses = processes.OrderByDescending(process => process.Memory).**Take(2)**; |

Sum:

我们想要计算一下获取的最大2个进程的总内存, 这时我们可以使用另一个内置的扩展方法: Sum, Sum就像我们之前定义的TotalMomery:

|  |
| --- |
| var totalMemory = processes.OrderByDescending(process => process.Memory).Take(2).**Sum(process => process.Memory)**; |

## 扩展方法的限制:

示例：**ExtensionMethodConflict**

扩展方法一个非常重要的问题就是和实例方法的竞争: 扩展方法比实例方法更低的优先级, 也就是说当同时存在扩展方法和实例方法时, 会调用实例方法, 扩展方法不能隐藏实例方法. 让我们举个例子说明一下:

|  |
| --- |
| class Class1  {  }  class Class2  {  public void Method1(string s)  {  Console.WriteLine("Class2.Method1");  }  }  class Class3  {  public void Method1(object o)  {  Console.WriteLine("Class3.Method1");  }  }  class Class4  {  public void Method1(int i)  {  Console.WriteLine("Class4.Method1");  }  }  static class Extensions  {  static public void Method1(this object o, int i)  {  Console.WriteLine("Extensions.Method1");  }  static void Main()  {  new Class1().Method1(12); // Extensions.Method1 is called  new Class2().Method1(12); // Extensions.Method1 is called  new Class3().Method1(12); // Class3.Method1 is called  new Class4().Method1(12); // Class4.Method1 is called  }  } |

运行后的结果是:

* Extensions.Method1.
* Extensions.Method2.
* Class3.Method1.
* Class4.Method2.

从这里可以看出, 当有实例方法满足参数条件时, 会直接调用实例方法, 当所有的实例代码都不满足签名后, 才会调用扩展方法.

扩展方法比起实例方法还有很多其他的限制, 比如扩展方法不能访问非公共成员. 当我们使用扩展方法把所有方法集中写在一个其他的静态类中时, 它也影响了我们的可读性, 所以只有当我们不合适写实例方法时, 才建议使用静态方法.

# 匿名类型(Anonymous Type).

示例：**AnonymousType**

和对象初始化的语法相似, 我们可以创建匿名类型. 他可以在不定义类的情况下, 把数据分组到一个对象. 我们将先用示例介绍一下匿名类型, 以及匿名类型为什么是真实存在的类型, 最后介绍一下匿名类型的限制.

## 使用匿名类型把数据集合到对象中.

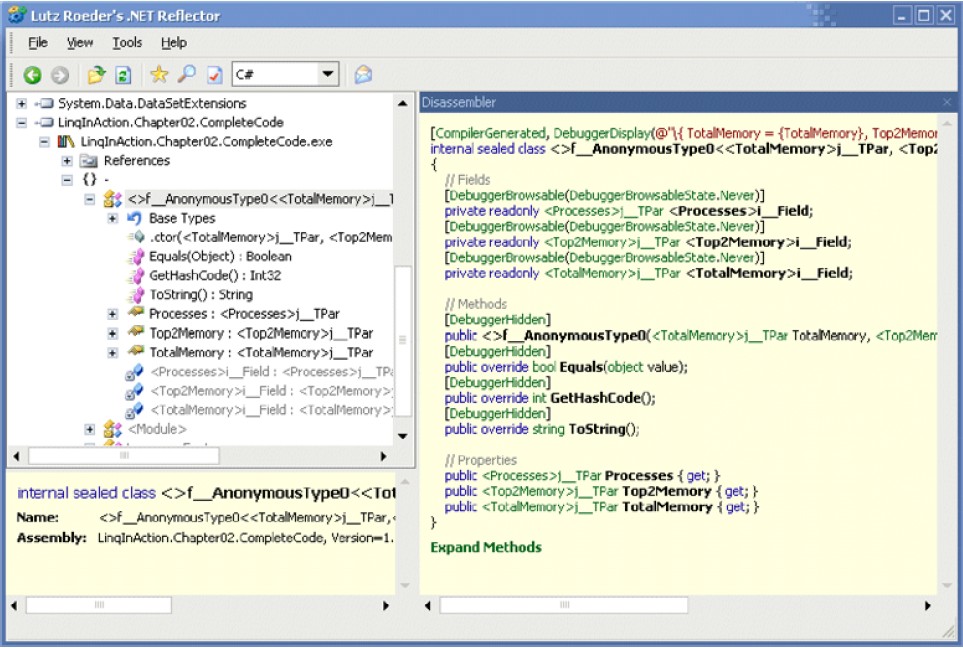
我们需要把数据结合到一个对象中, 定义一个特定的类型比较繁琐.

我们可以在C#中使用匿名类型:

|  |
| --- |
| var results = new {  TotalMemory = processes.TotalMemory()/1024/1024,  Top2Memory = top2Memory,  Processes = processes}; |

## 没有名称的类型.

匿名类型是没有名称的类型, 但是他也是类型. 编译器会为他创建真实的类型. 我们的results变量指向代码自动创建的类的一个实例. 这个类保护三个属性: TotalMemory, Top2Memory 和 Processes. 属性的类型在初始化时被推断出来. 下图说明了程序集为我们创建的匿名类型.



上图是我们使用反编译工具对示例编译的匿名类型进行反编译.

注意, 编译器会把2个具有顺序的属性名称和属性类型的匿名类型编译成同一个类型. 如果我们有如下代码, 则编译器只会编译成一个类型.

|  |
| --- |
| var v1 = new { Person = "Suzie", Age = 32, CanCode = true };  var v2 = new { Person = "Barney", Age = 28, CanCode = false }; |

在上面的代码片段执行后, v1和v2会是同一个类的不同实例.

如果我们运行下面的代码段, v3会是另一个类的实例, 因为属性的顺序不同.

|  |
| --- |
| var v3 = new { Age = 28, Person = "Bill", CanCode = false }; |

## 用匿名类型提高我们的示例

这些看起来很好, 我们可以利用匿名方法去掉ProcessData对象. 让我们从新回到我们想要做的, 显示进程方法DisplayProcesses, 这次我们要用匿名方法代替ProcessData类.

|  |
| --- |
| static void DisplayProcesses(Func<Process, Boolean> match)  {  var processes = new List<Object>();  foreach (var process in Process.GetProcesses())  {  if (match(process))  {  processes.Add( new { process.Id, Name=process.ProcessName,  Memory=process.WorkingSet64 } );  }  }  // 显示对象.  } |

这样做最大的好处就是我们不需要定义ProcessData类. 这个使得匿名类型是解决迅速和简单的解决临时结果的好办法, 我们不需要为了临时的结果定义一个类型.

但是匿名对象也有很多限制.

## 匿名对象的限制.

有一个问题时我们新的代码已经去掉了ProcessData类, 我们不能使用TotalMemory方法因为他是定义在ProcessData对象上. 当我们使用了匿名类型, 我们就不能在这个定义匿名类型的方法以为使用强类型. 这个意思指我们要给一个方法传入一个匿名类型的对象, 只能使用Object定义参数类型, 不能使用跟精确的类型. 反射是在定义匿名类型的方法以为使用匿名类型的唯一方法.

同样的, 匿名类型不能是一个方法的返回, 除非这个对象的返回值得类型是Object. 这就是为什么匿名类型只能是临时的数据, 不能像普通类型一样作为方法的签名.

但是, 上面说的也不是全部正确的, 我们可以使用泛型方法让匿名类型作为方法的返回值, 让我们试一下以下的方法.

|  |
| --- |
| static T ReturnAGeneric<T>(Func<T> creator)  {  return creator();  } |

ReturnAGeneric方法的返回值是一个泛型, 我们可以不用显式指出T类型参数的具体类型, 这个可以根据creator参数的签名推断出返回值的类型. 现在让我们试一下怎么调用ReturnAGeneric.

|  |
| --- |
| var result = ReturnAGeneric(() => new { Time = DateTime.Now, AString = "Abc" }); |

因为该方法以一个返回匿名类型的函数作为参数. ReturnAGeneric返回这个实例, 但是ReturnAGeneric没有定义一个Object作为返回值, 而是使用一个泛型. 这就是为什么result变量是一个强类型. 这个变量包含了一个时间类型的Time属性, 和一个字符串类型Astring属性.

我们ReturnAGeneric方法看起来没什么用处, 但是在Linq的使用中, 这个扩展的功能非常有用.

对于匿名方法, 我们还有注意在C#中, 匿名类型的实例是不变的. 指的是一旦我们创建了匿名方法的实例, 他的字段和属性都是固定不变的. 如果我们仔细看了一下匿名类型的反编译结果, 我们可以看到所有的属性只有getters但是没有setters. 唯一给属性和内部字段赋值的方法就是通过构造函数, 当我们使用类的初始化语法来初始化匿名类型时, 构造函数会被自动调用, 这些属性的值会一次性被设置.

因为匿名类型的实例是不可变的, 他们具有固定的Hash代码, 如果对象不能被修改, 那么他的Hash代码是不会被修改的(除非他内部的一个字段的Hask代码不固定). 这个在Hash 表和数据绑定的场景非常有用.

你会疑问为什么匿名类型在C#中被定义为不可变的, 这个可能在很多场景限制了匿名类型的使用. 这个是在避免客户端修改的功能语言中使用, 对象不能被修改在一致性时更有优势. 不可变的匿名对象让.Net更接近功能性编程世界, 可以用来为状态的快照和无副作用的代码.