您的潜力，我们的动力

Microsoft
微软（中国）有限公司

Visual Studio 2005 系列课程

跟我一起学 Visual Studio 2005
C# 语法篇(下）

徐长龙

vsts_china@hotmail.com

msdn

MSDN Webcasts

本系列课程简介

- 跟我一起学 Visual Studio 2005，本系列课程的目标是让各位对VS 2005有一个全面的认识与了解，从VS 2005的新功能的角度，全面实例介绍VS 2005的各方面新特性。
- 暂定以下九个主题，会根据大家的反馈不断进行调整。

1. C# 语法篇　　（今天的课程）
2. Win Form 编程篇
3. ASP.NET 2.0 篇
4. ADO.NET 2.0 篇
5. Crystal Report 篇
6. 智能设备编程篇
7. Office 编程篇
8. 部署篇
9. Team System 篇

今天的重点

- C#泛型(C# Generics)

前提

- 熟悉C# 1.1
- 熟悉Visual Studio .net 开发工具
- 2006年2月8日的课程《跟我一起学Visual Studio 2005（一）—C＃语法篇(上)》

- Level:200

C#泛型(C# Generics)

- 概述
- 什么是泛型？
- 如何使用泛型？
- 泛型约束
- 泛型和强制类型转换
- 继承和泛型
- 泛型方法
- 泛型委托
- 泛型和反射

泛型和强制类型转换

- C# 编译器只允许将泛型参数隐式强制转换到 **Object** 或约束指定的类型

```
interface ISomeInterface
{...}
class BaseClass
{...}
class MyClass<T> where T : BaseClass,ISomeInterface
{
   void SomeMethod(T t)
   {
      ISomeInterface obj1 = t;
      BaseClass      obj2 = t;
      object         obj3 = t;
   }
}
```

类型安全的，编译时检查

泛型和强制类型转换

- 编译器允许您将泛型参数显式强制转换到其他任何接口，但不能将其转换到类

```
interface ISomeInterface
{...}
class SomeClass
{...}
class MyClass<T>
{
  void SomeMethod(T t)
  {
    ISomeInterface obj1 = (ISomeInterface)t;  //Compiles
    SomeClass      obj2 = (SomeClass)t;       //Does not compile
  }
}
```

泛型和强制类型转换

- 但是，您可以使用临时的 Object 变量，将泛型参数强制转换到其他任何类型

```
class SomeClass
{...}
class MyClass<T>
{
  void SomeMethod(T t)
  {
    object temp = t;
    SomeClass obj = (SomeClass)temp;
  }
}
```

泛型和强制类型转换

- 不用说强制类型转换是危险的，因为如果为取代泛型参数而使用的类型实参不是派生自您要显式强制转换到的类型，则可能在运行时引发异常
- 解决强制类型转换的方法：使用is和as运算符

```
public class MyClass<T>
{
  public void SomeMethod(T t)
  {
    if(t is int) {...}
    if(t is LinkedList<int,string>) {...}
    string str = t as string;
    if(str != null){...}
    LinkedList<int,string> list = t as LinkedList<int,string>;
    if(list != null){...}
  }
}
```

如果泛型参数的类型是所查询的类型，则 is 运算符返回 true

如果这些类型兼容，则 as 将执行强制类型转换，否则将返回 null

DEMO

- 代码演示

继承和泛型

- 在从泛型基类派生时，可以提供类型实参，而不是基类泛型参数

```
public class BaseClass<T>
{...}
public class SubClass : BaseClass<int>
{...}
```

- 如果子类是泛型，而非具体的类型实参，则可以使用子类泛型参数作为泛型基类的指定类型

```
public class BaseClass<T>
{...}
public class SubClass<T> : BaseClass<T>
{...}
```

继承和泛型

- 在使用子类泛型参数时，必须在子类级别重复在基类级别规定的任何约束

A.派生约束

```
public class BaseClass<T>  where T : ISomeInterface
{...}
public class SubClass<T> : BaseClass<T> where T : ISomeInterface
{...}
```

B.构造函数约束

```
public class BaseClass<T>  where T : new()
{
   public T SomeMethod()
   {
      return new T();
   }
}
public class SubClass<T> : BaseClass<T> where T : new()
{...}
```

继承和泛型

- 基类可以定义其签名使用泛型参数的虚拟方法。在重写它们时，子类必须在方法签名中提供相应的类型

```
public class BaseClass<T>
{
  public virtual T SomeMethod()
  {...}
}
public class SubClass: BaseClass<int>
{
  public override int SomeMethod()
  {...}
}
```

- 如果该子类是泛型类，则它还可以在重写时使用它自己的泛型参数

```
public class SubClass<T>: BaseClass<T>
{
  public override T SomeMethod()
  {...}
}
```

继承和泛型

- 您可以定义泛型接口、泛型抽象类，甚至泛型抽象方法。这些类型的行为像其他任何泛型基类型一样

```
public interface ISomeInterface<T>
{
  T SomeMethod(T t);
}
public abstract class BaseClass<T>
{
  public abstract T SomeMethod(T t);
}
public class SubClass<T> : BaseClass<T>
{
  public override T SomeMethod(T t)
  {...}
}
```

继承和泛型

- 泛型抽象方法和泛型接口有一种有趣的用法。在 C# 2.0 中，不能对泛型参数使用诸如 + 或 += 之类的运算符。例如，以下代码无法编译，因为 C# 2.0 不具有运算符约束

```
public class Calculator<T>
{
  public T Add(T arg1,T arg2)
  {
    return arg1 + arg2;//Does not compile
  }
  //Rest of the methods
}
```

- 但是，我们通过定义泛型操作，使用抽象方法（最好使用接口）进行补偿。由于抽象方法的内部不能具有任何代码，因此可以在基类级别指定泛型操作，并且在子类级别提供具体的类型和实现

继承和泛型

A.泛型抽象方法

```
public abstract class BaseCalculator<T>
{
   public abstract T Add(T arg1,T arg2);
   public abstract T Subtract(T arg1,T arg2);
   public abstract T Divide(T arg1,T arg2);
   public abstract T Multiply(T arg1,T arg2);
}
public class MyCalculator : BaseCalculator<int>
{
   public override int Add(int arg1, int arg2)
   {
      return arg1 + arg2;
   }
   //Rest of the methods
}
```

B.泛型接口

```
public interface ICalculator<T>
{
   T Add(T arg1,T arg2);
   //Rest of the methods
}
public class MyCalculator : ICalculator<int>
{
   public int Add(int arg1, int arg2)
   {
      return arg1 + arg2;
   }
   //Rest of the methods
}
```

泛型接口更精简的代码实现！

DEMO

- 代码演示

泛型方法

- 在 C# 2.0 中，方法可以定义特定于其执行范围的泛型参数

```
public class MyClass<T>
{
   public void MyMethod<X>(X x)
   {...}
}
```

- 即使包含类根本不使用泛型，您也可以定义方法特定的泛型参数

```
public class MyClass
{
   public void MyMethod<T>(T t)
   {...}
}
```

- 注意：该功能仅适用于方法。属性或索引器只能使用在类的作用范围中定义的泛型参数。

泛型方法

- 在调用定义了泛型参数的方法时，您可以提供要在调用场所使用的类型

```
MyClass obj = new MyClass();
obj.MyMethod<int>(3);
```

- 因此，当调用该方法时，C# 编译器将足够聪明，从而基于传入的参数的类型推断出正确的类型，并且它允许完全省略类型规范

```
MyClass obj = new MyClass();
obj.MyMethod(3);
```

该功能称为泛型推理

泛型方法

- 请注意，编译器无法只根据返回值的类型推断出类型

```
public class MyClass
{
  public T MyMethod<T>()
  {}
}
MyClass obj = new MyClass();
int number = obj.MyMethod();//Does not compile
```

- 当方法定义它自己的泛型参数时，它还可以定义这些类型的约束

```
public class MyClass
{
  public void SomeMethod<T>(T t) where T : IComparable<T>
  {...}
}
```

- 但是，您无法为类级别泛型参数提供方法级别约束。类级别泛型参数的所有约束都必须在类作用范围中定义

泛型方法

- 在重写定义了泛型参数的虚拟方法时，子类方法必须重新定义该方法特定的泛型参数

```
public class BaseClass
{
   public virtual void SomeMethod<T>(T t)
   {...}    }
public class SubClass : BaseClass
{
   public override void SomeMethod<T>(T t)
   {...}    }
```

- 子类实现不能重复在基础方法级别出现的约束

```
public class BaseClass
{
   public virtual void SomeMethod<T>(T t) where T : new()
   {...}    }
public class SubClass : BaseClass
{
   public override void SomeMethod<T>(T t)
   {...}    }
```

注意：方法重写不能定义没有在基础方法中出现的新约束

泛型方法

- 此外，如果子类方法调用虚拟方法的基类实现，则它必须指定要代替泛型基础方法类型参数使用的类型实参。您可以自己显式指定它，或者依靠类型推理（如果可用）

```
public class BaseClass
{
  public virtual void SomeMethod<T>(T t)
  {...}
}
public class SubClass : BaseClass
{
  public override void SomeMethod<T>(T t)
  {
    base.SomeMethod<T>(t);
    base.SomeMethod(t);
  }
}
```

泛型方法—泛型静态方法

- C# 允许定义使用泛型参数的静态方法。但是，在调用这样的静态方法时，您需要在调用场所为包含类提供具体的类型

```
public class MyClass<T>
{
   public static T SomeMethod(T t)
   {...}
}
int number = MyClass<int>.SomeMethod(3);
```

泛型方法—泛型静态方法

- 静态方法可以定义方法特定的泛型参数和约束，就像实例方法一样。在调用这样的方法时，您需要在调用场所提供方法特定的类型

```
public class MyClass<T>
{
   public static T SomeMethod<X>(T t,X x)
   {..}
}
int number = MyClass<int>.SomeMethod<string>(3,"AAA");
```

- 或者类型推理（如果可能）

```
int number = MyClass<int>.SomeMethod(3,"AAA");
```

泛型方法—泛型静态方法

- 泛型静态方法遵守施加于它们在类级别使用的泛型参数的所有约束。就像实例方法一样，您可以为静态方法定义的泛型参数提供约束

```
public class MyClass
{
  public static T SomeMethod<T>(T t) where T : IComparable<T>
  {...}
}
```

- C# 中的运算符只是静态方法而已，并且 C# 允许您为自己的泛型重载运算符

```
LinkedList<int,string> list1 = new LinkedList<int,string>();
LinkedList<int,string> list2 = new LinkedList<int,string>();
 ...
LinkedList<int,string> list3 = list1+list2;
```

泛型方法—泛型静态方法

```
public class LinkedList<K,T>
{
  public static LinkedList<K,T> operator+(LinkedList<K,T> lhs,
                                          LinkedList<K,T> rhs)
  {
    return concatenate(lhs,rhs);
  }
  static LinkedList<K,T> concatenate(LinkedList<K,T> list1,
                                     LinkedList<K,T> list2)
  {
    LinkedList<K,T> newList = new LinkedList<K,T>();
    Node<K,T> current;
    current = list1.m_Head;
    while(current != null)
    {
      newList.AddHead(current.Key,current.Item);
      current = current.NextNode;
    }
    current = list2.m_Head;
    while(current != null)
    {
      newList.AddHead(current.Key,current.Item);
      current = current.NextNode;
    }
    return newList;
  }
  //Rest of LinkedList
}
```

DEMO

- 代码演示

泛型委托

- 在某个类中定义的委托可以利用该类的泛型参数

```
public class MyClass<T>
{
   public delegate void GenericDelegate(T t);
   public void SomeMethod(T t)
   {...}
}
```

- 在为包含类指定类型时，也会影响到委托

```
MyClass<int> obj = new MyClass<int>();
MyClass<int>.GenericDelegate del;

del = new MyClass<int>.GenericDelegate(obj.SomeMethod);
del(3);
```

泛型委托

- C# 2.0 使您可以将方法引用的直接分配转变为委托变量

```
MyClass<int> obj = new MyClass<int>();
MyClass<int>.GenericDelegate del;

del = obj.SomeMethod;
```

- 我将把该功能称为委托推理。编译器能够推断出您分配到其中的委托的类型，查明目标对象是否具有采用您指定的名称的方法，并且验证该方法的签名匹配。然后，编译器创建所推断出的参数类型（包括正确的类型而不是泛型参数）的新委托，并且将新委托分配到推断出的委托中

泛型委托

- 像类、结构和方法一样，委托也可以定义泛型参数

```
public class MyClass<T>
{
   public delegate void GenericDelegate<X>(T t,X x);
}
```

- 在类的作用范围外部定义的委托可以使用泛型参数。在该情况下，在声明和实例化委托时，必须为其提供类型实参

```
public delegate void GenericDelegate<T>(T t);

public class MyClass
{
   public void SomeMethod(int number)
   {...}
}

MyClass obj = new MyClass();
GenericDelegate<int> del;
del = new GenericDelegate<int>(obj.SomeMethod);
del(3);
```

泛型委托

- 另外，还可以在分配委托时使用委托推理

```
MyClass obj = new MyClass();
GenericDelegate<int> del;

del = obj.SomeMethod;
```

- 当然，委托可以定义约束以伴随它的泛型参数

```
public delegate void MyDelegate<T>(T t) where T : IComparable<T>;
```

- 委托级别约束只在使用端实施（在声明委托变量和实例化委托对象时），类似于在类型或方法的作用范围中实施的其他任何约束

泛型委托

- 泛型委托对于事件尤其有用。您可以精确地定义一组有限的泛型委托（只按照它们需要的泛型参数的数量进行区分），并且使用这些委托来满足所有事件处理需要

```
public delegate void GenericEventHandler<S,A>(S sender,A args);
public class MyPublisher
{
  public event GenericEventHandler<MyPublisher,EventArgs> MyEvent;
  public void FireEvent()
  {
    MyEvent(this,EventArgs.Empty);
  }
}
public class MySubscriber<A> //Optional: can be a specific type
{
  public void SomeMethod(MyPublisher sender,A args)
  {...}
}
MyPublisher publisher = new MyPublisher();
MySubscriber<EventArgs> subscriber = new MySubscriber<EventArgs>();
publisher.MyEvent += subscriber.SomeMethod;
```

泛型委托

- 使用名为 GenericEventHandler 的泛型委托，它接受泛型发送者类型和泛型参数。显然，如果您需要更多的参数，则可以简单地添加更多的泛型参数，但是我希望模仿按如下方式定义的 .NET EventHandler 来设计 GenericEventHandler

```
public void delegate EventHandler(object sender,EventArgs args);
```

- 与 EventHandler 不同，GenericEventHandler 是类型安全的，因为它只接受 MyPublisher 类型的对象（而不是纯粹的 Object）作为发送者。实际上，.NET 已经在 System 命名空间中定义了泛型的 EventHandler

```
public void delegate EventHandler(object sender,A args) where A : EventArgs;
```

DEMO

- 代码演示

泛型和反射

- 在 .NET 2.0 中，扩展了反射以支持泛型参数。类型 Type 现在可以表示带有特定类型实参（称为绑定类型）或未指定（未绑定）类型的泛型。像 C# 1.1 中一样，您可以通过使用 typeof 运算符或者通过调用每个类型支持的 GetType() 方法来获得任何类型的 Type。

```
LinkedList<int,string> list = new LinkedList<int,string>();

Type type1 = typeof(LinkedList<int,string>);
Type type2 = list.GetType();
Debug.Assert(type1 == type2);
```

- typeof 和 GetType() 都可以对泛型参数进行操作

```
public class MyClass<T>
{
  public void SomeMethod(T t)
  {
    Type type = typeof(T);
    Debug.Assert(type == t.GetType());
  }
}
```

泛型和反射

- 此外，typeof 运算符还可以对未绑定的泛型进行操作

```
public class MyClass<T>
{}
Type unboundedType = typeof(MyClass<>);
Trace.WriteLine(unboundedType.ToString());
//Writes: MyClass`1[T]
```

所追踪的数字 1 是所使用的泛型类型的泛型类型参数的数量

- 请注意空 <> 的用法。要对带有多个类型参数的未绑定泛型类型进行操作，请在 <> 中使用“,”

```
public class LinkedList<K,T>
{...}
Type unboundedList = typeof(LinkedList<,>);
Trace.WriteLine(unboundedList.ToString());
//Writes: LinkedList`2[K,T]
```

泛型和反射

- Type 类中添加了新的方法和属性，用于提供有关该类型的泛型方面的反射信息

```
public abstract class Type : //Base types
{
   public virtual Type[] GetGenericArguments();
   public virtual Type[] GetGenericParameterConstraints();
   public virtual Type GetGenericTypeDefinition();
   public virtual int GenericParameterPosition{get;}
   public virtual bool IsGenericType { get; }
   public virtual bool IsGenericParameter{get;}
   public virtual bool IsGenericTypeDefinition{get;}
   //Rest of the members
}
```

泛型和反射


```
LinkedList<int, string> list = new LinkedList<int, string>();

Type boundedType = list.GetType();
Trace.WriteLine(boundedType.ToString());
//Writes: LinkedList`2[System.Int32,System.String]

Debug.Assert(boundedType.IsGenericType);

Type[] parameters = boundedType.GetGenericArguments();

Debug.Assert(parameters.Length == 2);
Debug.Assert(parameters[0] == typeof(int));
Debug.Assert(parameters[1] == typeof(string));

Type unboundedType = boundedType.GetGenericTypeDefinition();
Debug.Assert(unboundedType == typeof(LinkedList<,>));
Trace.WriteLine(unboundedType.ToString());
//Writes: LinkedList`2[K,T]
```

泛型和反射

- 与 Type 类似，MethodInfo 和它的基类 MethodBase 具有反射泛型方法信息的新成员

```
LinkedList<int,string> list = new LinkedList<int,string>();
Type type = list.GetType();
MethodInfo methodInfo = type.GetMethod("AddHead");
object[] args = {1,"AAA"};
methodInfo.Invoke(list,args);
```

晚期绑定

属性和泛型

- C# 2.0 不允许定义泛型属性

```
//Does not compile:
public class SomeAttribute<T> : Attribute
{...}
```

- 属性类可以使用泛型类，也可以定义泛型方法

```
public class SomeAttribute : Attribute
{
  void SomeMethod<T>(T t)
  {...}
  LinkedList<int,string> m_List = new LinkedList<int,string>();
}
```

泛型集合

- System.Collections 中的数据结构全部都是基于 Object 的，基于Object的方案性能较差和缺少类型安全。.NET 2.0 在 System.Collections.Generic 命名空间中引入了一组泛型集合。例如，有泛型的Stack<T>类和泛型的Queue<T>类。Dictionary<K,T> 数据结构等效于非泛型的 HashTable，并且还有一个有点像 SortedList 的 SortedDictionary<K,T>类。类 List<T> 类似于非泛型的 ArrayList

```
//Does not compile:
public class SomeAttribute<T> : Attribute
{...}
```

泛型集合

System.Collections.Generic	System.Collections
Comparer<T>	Comparer
Dictionary<K,T>	HashTable
LinkedList<T>	-
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack
ICollection<T>	ICollection
IComparable<T>	System.IComparable
IDictionary<K,T>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IList<T>	IList

DEMO

- 代码演示

泛型无法完成的工作

- 在 .NET 2.0 下，您不能定义泛型 Web 服务，即使用泛型类型参数的 Web 方法。原因是没有哪个 Web 服务标准支持泛型服务。
- 您还不能在服务组件上使用泛型类型。原因是泛型不能满足 COM 可见性要求，而该要求对于服务组件而言是必需的（就像您无法在 COM 或 COM+ 中使用 C++ 模板一样）

小结

- 泛型和强制类型转换
- 继承和泛型
- 泛型方法
- 泛型委托
- 泛型和反射
- 泛型集合

您的潜力，我们的动力

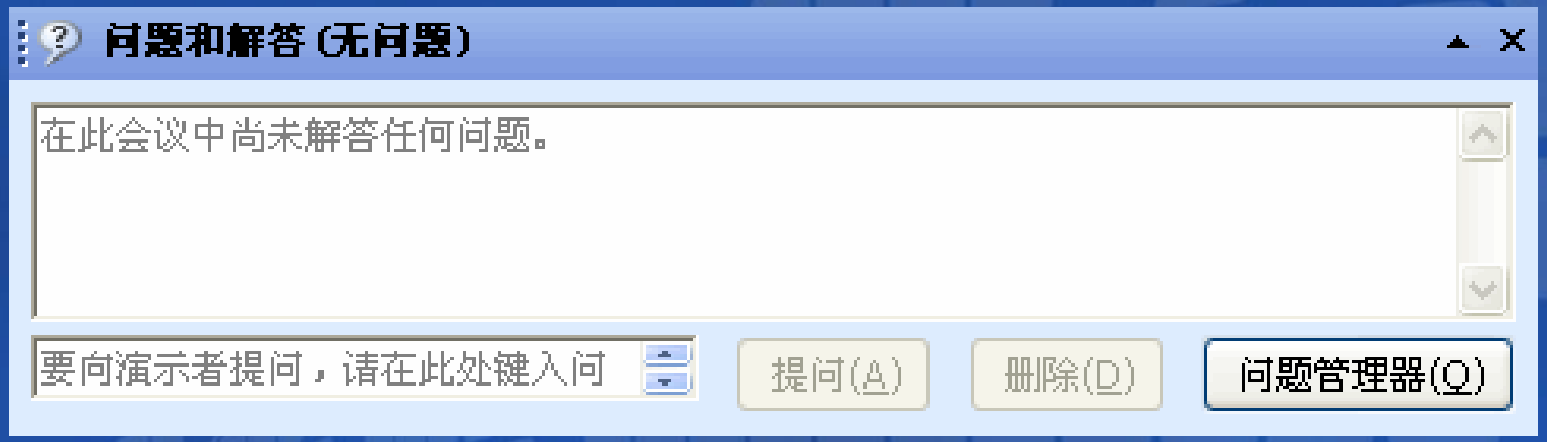
Microsoft
微软（中国）有限公司

获取更多MSDN资源

- MSDN中文网站
  http://www.microsoft.com/china/msdn
- MSDN中文网络广播
  http://www.msdnwebcast.com.cn
- MSDN Flash
  http://www.microsoft.com/china/newsletter/case/msdn.aspx
- MSDN开发中心
  http://www.microsoft.com/china/msdn/DeveloperCenter/default.mspx

Question & Answer

- 如需提出问题，请单击“提问”按钮并在随后显示的浮动面板中输入问题内容。一旦完成问题输入后，请单击“提问”按钮。




您的潜力，我们的动力
Microsoft
微软（中国）有限公司
Microsoft
msdn
MSDN Webcasts