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(上）

徐长龙
vsts_china@hotmail.com

MSDN Webcasts

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

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

Microsoft Visual Studio 2005
msdn
MSDN Webcasts

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

今天的重点

- C#泛型(C# Generics)

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

前提

- 熟悉C# 1.1
- 熟悉Visual Studio .net 开发工具

- Level:200

C#泛型(C# Generics)

- 概述
- 什么是泛型？
- 如何使用泛型？
- 泛型约束

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

C#泛型概述

- 先看一个通用的数据结构示例：

结构类:

```
public class Stack
{
   object[] m_Items;
   public void Push(object item)
   {...}
   public object Pop()
   {...}
}
```

实例:

```
Stack stack = new Stack();
stack.Push(1);
stack.Push(2);
int number = (int)stack.Pop();
```

C#泛型概述

通用数据结构示例

```
public class Stack
{
  readonly int m_Size;
  int m_StackPointer = 0;
  object[] m_Items;
  public Stack():this(100)
  {}
  public Stack(int size)
  {
    m_Size = size;
    m_Items = new object[m_Size];
  }
  public void Push(object item)
  {
    if(m_StackPointer >= m_Size)
      throw new StackOverflowException();
    m_Items[m_StackPointer] = item;
    m_StackPointer++;
  }
```

```
 public object Pop()
  {
    m_StackPointer–;
    if(m_StackPointer >= 0)
    {
      return m_Items[m_StackPointer];
    }
    else
    {
      m_StackPointer = 0;
      throw new InvalidOperationException(
              "Cannot pop an empty stack");
    }
  }
}
```

C#泛型概述

- 基于object解决方案存在的问题
  - 性能问题：
    - 值类型Push时要装箱处理，Pop时要取消装箱处理，造成更多的垃圾碎片，增加垃圾收集的负担
    - 引用类型也有强制转换的开销
  - 类型安全问题（更为严重）
    - 编译时任务类型都可以转换成object，无法保证运行时的类型安全

```
Stack stack = new Stack();
stack.Push("1");
string number = (string)stack.Pop();
```

```
Stack stack = new Stack();
stack.Push("test");
//This compiles, but is not type safe,
//and will throw an exception:
int number = (int)stack.Pop();
```

C#泛型概述

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

- 解决性能和类型安全的方法：
  - 编写特定类型的数据结构

```
public class IntStack
{
   int[] m_Items;
   public void Push(int item){...}
   public int Pop(){...}
}

IntStack stack = new IntStack();
stack.Push(1);
int number = stack.Pop();
```

```
public class StringStack
{
   string[] m_Items;
   public void Push(string item){...}
   public string Pop(){...}
}

StringStack stack = new StringStack();
stack.Push("1");
string number = stack.Pop();
```

C#泛型概述

- 同样严重的新的问题又产生了：
  - 影响工作效率
  - 代码冗余，重用率不高
  - 一个数据结构变更，要将所有类型的数据结构做相应的修改
  - 为了提供不可预知的数据类型的支持，还是要提供object类型接口，类型安全的问题又会出现

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

什么是泛型？

- 通过泛型可以定义类型安全类，而不会损害类型安全、性能或工作效率

```
public class Stack
{
   object[] m_Items;
   public void Push(object item)
   {...}
   public object Pop()
   {...}
}
```

```
public class Stack <T>
{
   T[] m_Items;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
```

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

什么是泛型？

- 可以使用任何类型来声明和实例化。
- 声明和实例化时都必须用一个特定的类型来代替一般类型 T

```
public class Stack
{
   object[] m_Items;
   public void Push(object item)
   {...}
   public object Pop()
   {...}
}
```

```
public class Stack <T>
{
   T[] m_Items;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
```

```
Stack stack = new Stack();
stack.Push(1);
int number = (int)stack.Pop();
```

```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

什么是泛型？

- 通用数据结构类采用泛型实现

```
public class Stack<T>
{
  readonly int m_Size;
  int m_StackPointer = 0;
  T[] m_Items;
  public Stack():this(100)
  {}
  public Stack(int size)
  {
    m_Size = size;
    m_Items = new T[m_Size];
  }
  public void Push(T item)
  {
    if(m_StackPointer >= m_Size)
      throw new StackOverflowException();
    m_Items[m_StackPointer] = item;
    m_StackPointer++;
  }
```

```
public T Pop()
 {
   m_StackPointer–;
   if(m_StackPointer >= 0)
   {
     return m_Items[m_StackPointer];
   }
   else
   {
     m_StackPointer = 0;
     throw new InvalidOperationException(
             "Cannot pop an empty stack");
   }
 }
}
```

编程模型的优点在于，内部算法和数据操作保持不变，而实际数据类型可以在使用时指定。

泛型是如何实现的？

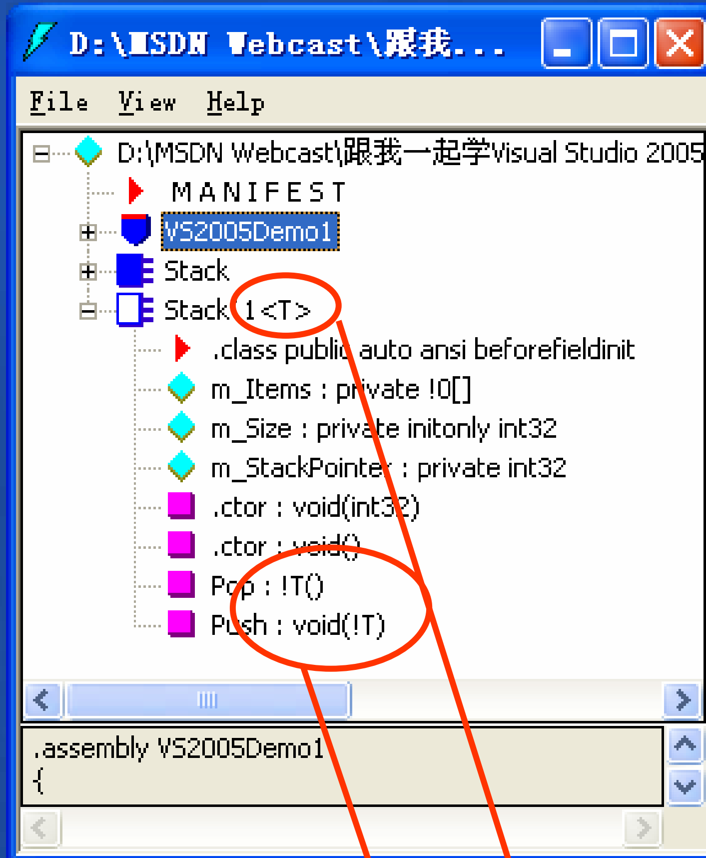
您的潜力，我们的动力
Microsoft
微软(中国)有限公司

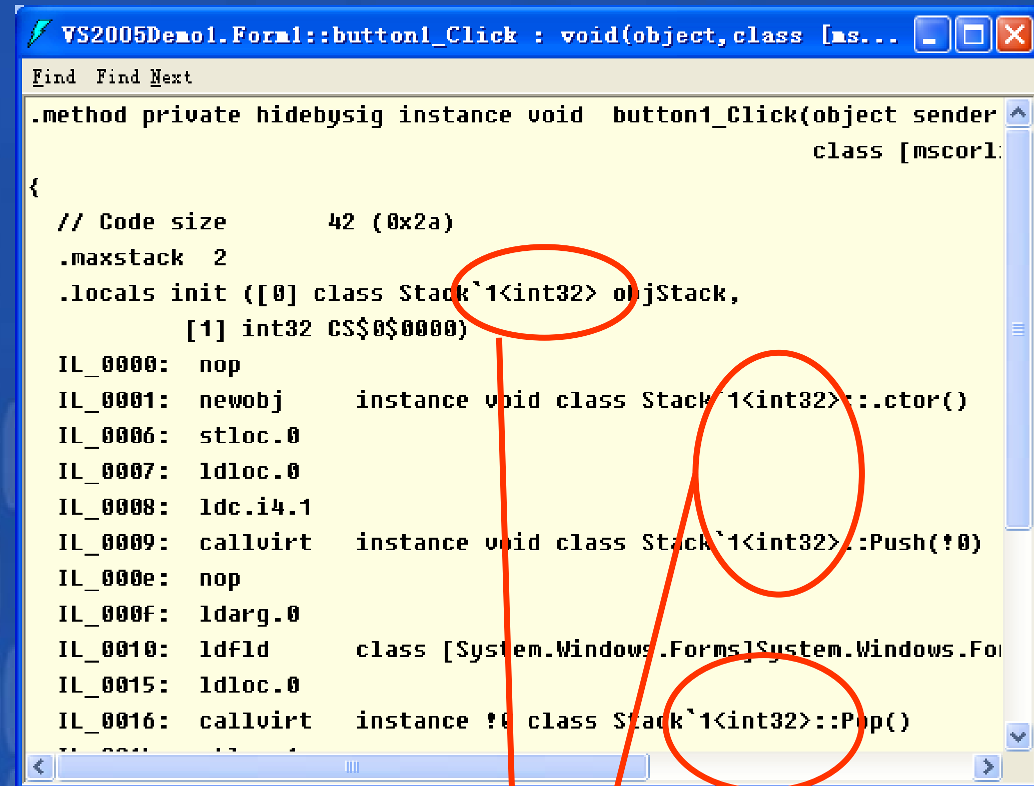
- 在 .NET 2.0 中，泛型在 IL（中间语言）和 CLR 本身中具有本机支持
- 编译泛型类时，就像编译其他类一样，泛型仅保留一个占位符
- 而用特定类型实例化泛型代码，编译时会将泛型替换为实例化的特定类型

泛型是如何实现的？

您的潜力，我们的动力

Microsoft 微软(中国)有限公司





泛型占位符

特定类型（整型）

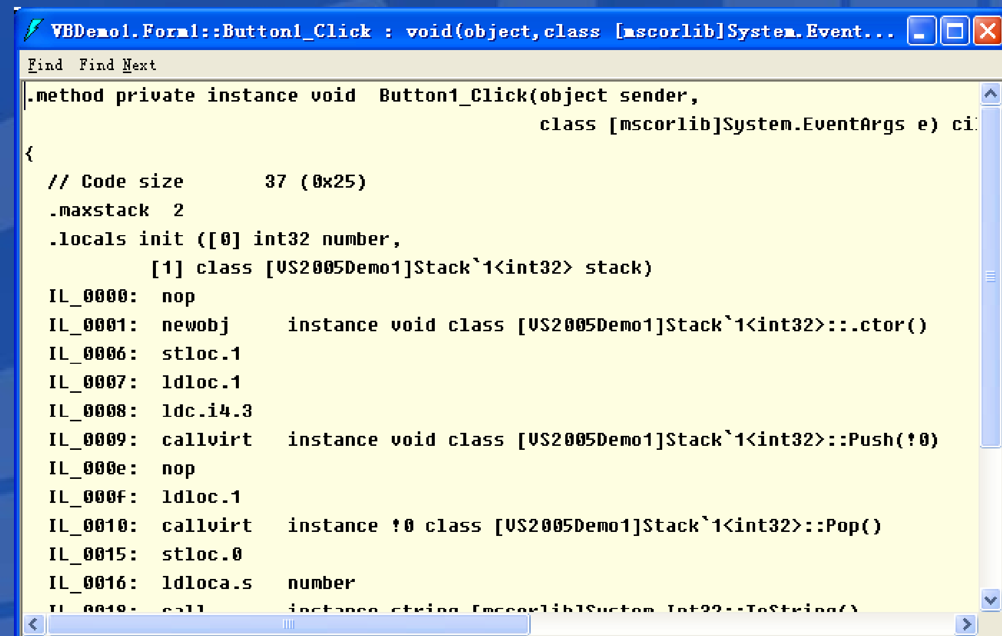
泛型的好处

- 一次性地开发、测试和部署代码，通过任何类型（包括将来的类型）来重用它
- 编译器支持和类型安全
- 不会强行对值类型进行装箱和取消装箱，或者对引用类型进行向下强制类型转换，所以性能得到显著提高
  - 值类型，性能通常会提高 **200%**
  - 引用类型，在访问该类型时，可以预期性能最多提高 **100%**（当然，整个应用程序的性能可能会提高，也可能不会提高）

应用泛型

- 因为 IL 和 CLR 为泛型提供本机支持，所以大多数符合 CLR 的语言都可以利用一般类型

```
Dim stack As Stack(Of Integer)
stack = new Stack(Of Integer)
stack.Push(3)
Dim number As Integer
number = stack.Pop()
```



您的潜力，我们的动力
Microsoft
微软(中国)有限公司

在结构中使用泛型

- 轻量级的结构中使用泛型

```
public struct Point<T>
{
   public T X;
   public T Y;
}
```

```
Point<int> point;
point.X = 1;
point.Y = 2;
```

```
Point<double> point;
point.X = 1.2;
point.Y = 3.4;
```

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

Default关键字

- 假设您不希望在堆栈为空时引发异常，而是希望返回堆栈中存储的类型的默认值
  - 值类型返回0（整型、枚举和结构）
  - 引用类型返回null
- 如果是基于object，则可以简单的返回null

```
public T Pop()
 {
   m_StackPointer–;
   if(m_StackPointer >= 0)
   {
     return m_Items[m_StackPointer];
   }
   else
   {
     m_StackPointer = 0;
     throw new InvalidOperationException(
            "Cannot pop an empty stack");
   }
 }
```

```
public T Pop()
{
  m_StackPointer–;
  if(m_StackPointer >= 0)
  {
    return m_Items[m_StackPointer];
  }
  else
  {
    m_StackPointer = 0;
    return default(T);
  }
}
```

多个泛型

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

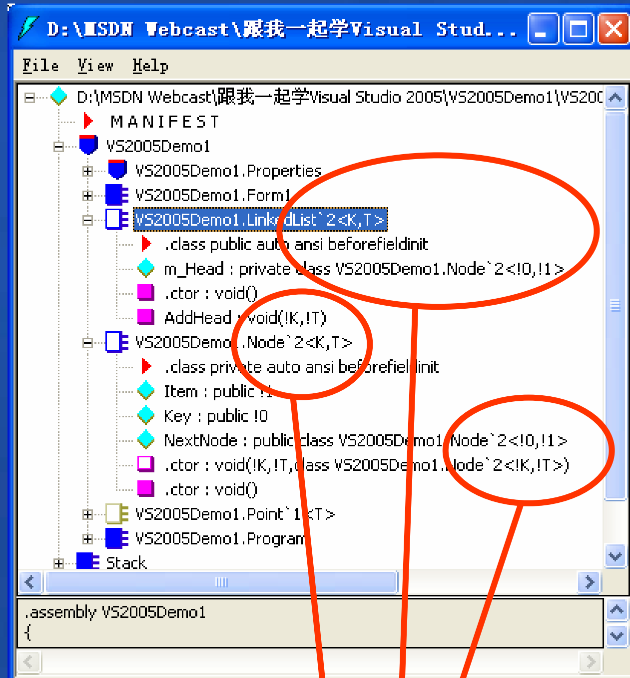
- 单个类型可以定义多个泛型

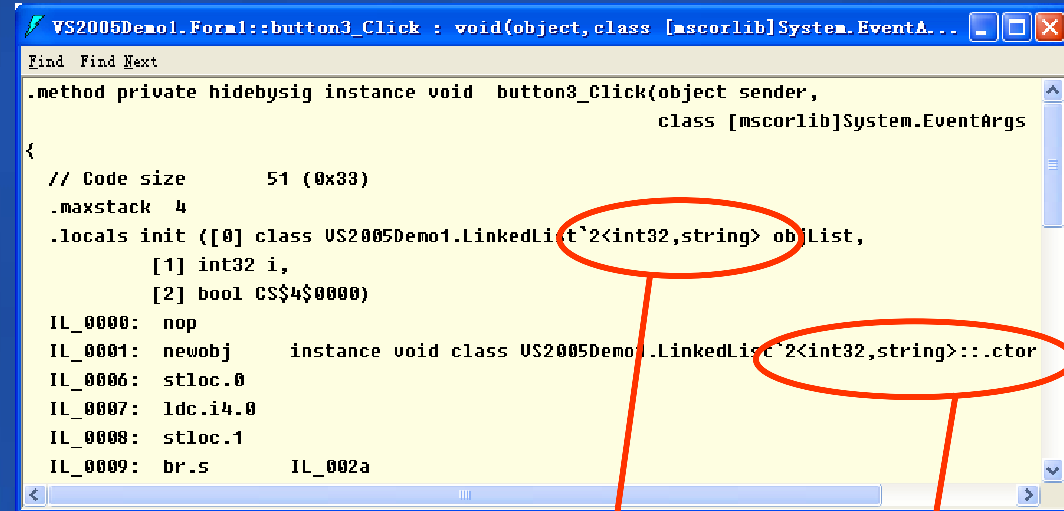
```
class Node<K,T>
{
  public K Key;
  public T Item;
  public Node<K,T> NextNode;
  public Node()
  {
    Key      = default(K);
    Item     = defualt(T);
    NextNode = null;
  }
  public Node(K key,T item,Node<K,T> nextNode)
  {
    Key      = key;
    Item     = item;
    NextNode = nextNode;
  }
}
```

```
public class LinkedList<K,T>
{
  Node<K,T> m_Head;
  public LinkedList()
  {
    m_Head = new Node<K,T>();
  }
  public void AddHead(K key,T item)
  {
    Node<K,T> newNode = new Node<K,T>
              (key,item,m_Head.NextNode);
    m_Head.NextNode = newNode;
  }
}
```

多个泛型

您的潜力，我们的动力
Microsoft
微软(中国)有限公司





泛型占位符

特定类型（整形，字符）

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

泛型别名

- 在文件头部使用using为特定类型取别名
- 别名作用范围是整个文件

```
using List = LinkedList<int,string>;

class ListClient
{
   static void Main(string[] args)
   {
      List list = new List();
      list.AddHead(123,"AAA");
   }
}
```

msdn
MSDN Webcasts

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

泛型约束-概述

- 为什么要泛型约束？
- 先看以下示例：

```
public class LinkedList<K,T>
{
  T Find(K key)
  {...}
  public T this[K key]
  {
    get{return Find(key);}
  }
}
```

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

list.AddHead(123,"AAA");
list.AddHead(456,"BBB");
string item = list[456];
Debug.Assert(item == "BBB");
```

泛型约束-概述

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

- Find方法实现

```
T Find(K key)
{
  Node<K,T> current = m_Head;
  while(current.NextNode != null)
  {
    if(current.Key == key) //Will not compile
      break;
    else

      current = current.NextNode;
  }
  return current.Item;
}
```

因为编译器不知道 K（实例化时的实际类型）是否支持 == 运算符。例如，默认情况下，结构不提供这样的实现。

```
public interface IComparable
{
  int CompareTo(object obj);
}
```

```
if(current.Key.CompareTo(key) == 0)
```

```
if(((IComparable)(current.Key)).CompareTo(key) == 0)
```

msdn
MSDN Webcasts

泛型约束-派生约束

- Find方法的解决方法
  - 泛型增加一个派生约束(Derivation Constraints)

```
public class LinkedList<K,T> where K : IComparable
{
  T Find(K key)
  {
    Node<K,T> current = m_Head;
    while(current.NextNode != null)
    {
      if(current.Key.CompareTo(key) == 0)
        break;
      else
        current = current.NextNode;
    }
    return current.Item;
  }
  //Rest of the implementation
}
```

1.Where关键字

2.K:IComparable表示K只接受实现IComparable接口的类型。

尽管如此，还是无法避免传入值类型的K所带来的装箱问题。System.Collections.Generic 命名空间定义了泛型接口IComparable<T>:

```
public interface IComparable<T>
{
  int CompareTo(T other);
  bool Equals(T other);
}
```

泛型约束-派生约束

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

- 在C#2.0中，所有的派生约束必须放在类的实际派生列表之后，如：

```
public class LinkedList<K,T> : IEnumerable<T> where K : IComparable<K>
{...}
```

- 通常，只须在需要的级别定义约束。比如：在Node节点定义IComparable<K>约束是没有意义的。如果一定要在Node上定义IComparable<K>约束，则LinkedList上也要定义此约束

```
class Node<K,T> where K : IComparable<K>
{...}
```

```
public class LinkedList<KeyType,DataType>
            where KeyType : IComparable<KeyType>
{
   Node<KeyType,DataType> m_Head;
}
```

msdn
MSDN Webcasts

泛型约束-派生约束

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

- 一个泛型参数上约束多个接口（彼此用逗号分隔）

```
public class LinkedList<K,T> where K : IComparable<K>,IConvertible
{...}
```

- 在一个约束中最多只能使用一个基类，这是因为 **C#** 不支持实现的多重继承
- 约束的基类不能是密封类或静态类，并且由编译器实施这一限制
- 不能将 **System.Delegate** 或 **System.Array** 约束为基类
- 可以同时约束一个基类以及一个或多个接口，但是该基类必须首先出现在派生约束列表中

```
public class LinkedList<K,T> where K : MyBaseClass, IComparable<K>
{...}
```

泛型约束-派生约束

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

- C# 允许您将另一个一般类型参数指定为约束

```
public class MyClass<T,U> where T : U
{...}
```

- 定义自己的基类或接口进行泛型约束

```
public interface IMyInterface
{...}
public class MyClass<T> where T : IMyInterface
{...}
MyClass<IMyInterface> obj = new MyClass<IMyInterface>();
```

```
public class MyOtherClass
{...}
public class MyClass<T> where T : MyOtherClass
{...}
MyClass<MyOtherClass> obj = new MyClass<MyOtherClass>();
```

泛型约束-派生约束

- 自定义的接口或基类必须与泛型参数具有一致的可见性

正确的可见性

```
public class MyBaseClass
{}
internal class MySubClass<T> where T : MyBaseClass
{}
```

颠倒的可见性

```
internal class MyBaseClass
{}
public class MySubClass<T> where T : MyBaseClass
{}
```

泛型约束-构造函数约束

- 假设您要在一般类的内部实例化一个新的一般对象。问题在于，**C#** 编译器不知道客户端将使用的类型实参是否具有匹配的构造函数，因而它将拒绝编译实例化行。
- 为了解决该问题，**C#** 允许约束一般类型参数，以使其必须支持公共默认构造函数。这是使用 **new()** 约束完成的。

```
class Node<K,T> where T : new()
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node()
   {
      Key      = default(K);
      Item      = new T();
      NextNode = null;
   }
}
```

泛型约束-构造函数约束

- 可以将构造函数约束与派生约束组合起来，前提是构造函数约束出现在约束列表中的最后

```
public class LinkedList<K,T> where K : IComparable<K>,new()
{...}
```

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

泛型约束-引用/值类型约束

- 可以使用 struct 约束将一般类型参数约束为值类型（例如，int、bool 和 enum），或任何自定义结构

```
public class MyClass<T> where T : struct
{...}
```

- 同样，可以使用 class 约束将一般类型参数约束为引用类型（类）

```
public class MyClass<T> where T : class
{...}
```

泛型约束-引用/值类型约束

- 不能将引用/值类型约束与基类约束一起使用，因为基类约束涉及到类
- 不能使用结构和默认构造函数约束，因为默认构造函数约束也涉及到类
- 虽然您可以使用类和默认构造函数约束，但这样做没有任何价值
- 可以将引用/值类型约束与接口约束组合起来，前提是引用/值类型约束出现在约束列表的开头

DEMO

- 代码演示

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

小结

- 概述
- 什么是泛型？
- 如何使用泛型？
- 泛型约束
  - 派生约束
  - 构造函数约束
  - 引用/值类型约束

您的潜力，我们的动力
Microsoft®
微软(中国)有限公司

下次课程预览

- 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
- Partial
- Edit&Continue (E&C)...

获取更多MSDN资源

您的潜力，我们的动力
Microsoft
微软(中国)有限公司

- **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

您的潜力，我们的动力

Microsoft®
微软（中国）有限公司

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。

问题和解答 (无问题)

在此会议中尚未解答任何问题。

要向演示者提问，请在此处键入问

提问(A)　删除(D)　问题管理器(Q)

您的潜力，我们的动力

Microsoft®

微软(中国)有限公司

Microsoft®

MSDN Webcasts