Contents

[<C#入门经典> 1](#_Toc440974114)

[<C#与.NET4高级程序设计> 8](#_Toc440974115)

[<Window Form> 35](#_Toc440974116)

[<WPF 和XAML> 36](#_Toc440974117)

Visual Studio IDE

vs2008 快捷键

Ctrl+E,D ----格式化全部代码

Ctrl+E,F ----格式化选中的代码

CTRL + SHIFT + B生成解决方案

CTRL + F7 生成编译

CTRL + O 打开文件

CTRL + SHIFT + O打开项目

CTRL + SHIFT + C显示类视图窗口

F4 显示属性窗口

SHIFT + F4显示项目属性窗口

CTRL + SHIFT + E显示资源视图

F12 转到定义

CTRL + F12转到声明

CTRL + ALT + J对象浏览

CTRL + ALT + F1帮助目录

CTRL + F1 动态帮助

F1 帮助

SHIFT + F1当前窗口帮助

CTRL + ALT + F3帮助-搜索

SHIFT + ALT + ENTER全屏显示

CTRL + -向后定位

CTRL + SHIFT + -向前定位

CTRL + F4关闭文档窗口

CTRL + PAGE DOWN光标定位到窗口上方

CTRL + PAGE UP光标定位到窗口下方

CTRL + F6

CTRL + TAB下一个文档窗口

CTRL + SHIFT + F6

CTRL + SHIFT + TAB上一个文档窗口

ALT + F6下一个面板窗口

CTRL + K, CTRL + L取消remark

CTRL + K, CTRL + C注释选择的代码

CTRL + K, CTRL + U取消对选择代码的注释

CTRL + M, CTRL + O折叠代码定义

CTRL + M, CTRL + L展开代码定义

CTRL + DELETE删除至词尾

CTRL + BACKSPACE删除至词头

SHIFT + TAB取消制表符

CTRL + U转小写

CTRL + SHIFT + U转大写

CTRL + SHIFT + END选择至文档末尾

CTRL + SHIFT + HOME选择至文档末尾开始

SHIFT + END选择至行尾

SHIFT + HOME选择至行开始处

SHIFT + ALT + END垂直选择到最后尾

SHIFT + ALT + HOME垂直选择到最前面

CTRL + A全选

CTRL + W选择当前单词

CTRL + SHIFT + PAGE UP选择至本页前面

CTRL + SHIFT + PAGE DOWN选择至本页后面

CTRL + END文档定位到最后

CTRL + HOME文档定位到最前

CTRL + G转到…

CTRL + K, CTRL + P上一个标签

CTRL + K, CTRL + N下一个标签

ALT + F10调试-ApplyCodeChanges

CTRL + ALT+ Break停止调试

CTRL + SHIFT + F9 取消所有断点

CTRL + F9允许中断

CTRL + SHIFT + F5调试-重新开始

F5运行调试

CTRL + F5运行不调试

F10跨过程序执行

F11单步逐句执行

CTRL + J列出成员

CTRL + PAGE DOWN下一个视图

CTRL + B格式-粗体

CTRL + SHIFT + T格式-文字缩进

调试快捷键

F6: 生成解决方案

Ctrl+F6: 生成当前项目

F7: 查看代码

Shift+F7: 查看窗体设计器

F5: 启动调试

Ctrl+F5: 开始执行(不调试)

Shift+F5: 停止调试

Ctrl+Shift+F5: 重启调试

F9: 切换断点

Ctrl+F9: 启用/停止断点

Ctrl+Shift+F9: 删除全部断点

F10: 逐过程

Ctrl+F10: 运行到光标处

F11: 逐语句

编辑快捷键

Shift+Alt+Enter: 切换全屏编辑

Ctrl+B,T / Ctrl+K,K: 切换书签开关

Ctrl+B,N / Ctrl+K,N: 移动到下一书签

Ctrl+B,P: 移动到上一书签

Ctrl+B,C: 清除全部标签

Ctrl+I: 渐进式搜索

Ctrl+Shift+I: 反向渐进式搜索

Ctrl+F: 查找

Ctrl+Shift+F: 在文件中查找

F3: 查找下一个

Shift+F3: 查找上一个

Ctrl+H: 替换

Ctrl+Shift+H: 在文件中替换

Alt+F12: 查找符号(列出所有查找结果)

Ctrl+Shift+V: 剪贴板循环

Ctrl+左右箭头键: 一次可以移动一个单词

Ctrl+上下箭头键: 滚动代码屏幕，但不移动光标位置。

Ctrl+Shift+L: 删除当前行

Ctrl+M,M: 隐藏或展开当前嵌套的折叠状态

Ctrl+M,L: 将所有过程设置为相同的隐藏或展开状态

Ctrl+M,P: 停止大纲显示

Ctrl+E,S: 查看空白

Ctrl+E,W: 自动换行

Ctrl+G: 转到指定行

Shift+Alt+箭头键: 选择矩形文本

Alt+鼠标左按钮: 选择矩形文本

Ctrl+Shift+U: 全部变为大写

Ctrl+U: 全部变为小写

代码快捷键

Ctrl+J / Ctrl+K,L: 列出成员

Ctrl+Shift+空格键 / Ctrl+K,P: 参数信息

Ctrl+K,I: 快速信息

Ctrl+E,C / Ctrl+K,C: 注释选定内容

Ctrl+E,U / Ctrl+K,U: 取消选定注释内容

Ctrl+K,M: 生成方法存根

Ctrl+K,X: 插入代码段

Ctrl+K,S: 插入外侧代码

F12: 转到所调用过程或变量的定义

窗口快捷键

Ctrl+W,W: 浏览器窗口

Ctrl+W,S: 解决方案管理器

Ctrl+W,C: 类视图

Ctrl+W,E: 错误列表

Ctrl+W,O: 输出视图

Ctrl+W,P: 属性窗口

Ctrl+W,T: 任务列表

Ctrl+W,X: 工具箱

Ctrl+W,B: 书签窗口

Ctrl+W,U: 文档大纲

Ctrl+D,B: 断点窗口

Ctrl+D,I: 即时窗口

Ctrl+Tab: 活动窗体切换

Ctrl+Shift+N: 新建项目

Ctrl+Shift+O: 打开项目

Ctrl+Shift+S: 全部保存

Shift+Alt+C: 新建类

Ctrl+Shift+A: 新建项

VS2005的隐藏快捷键

这里我将会把一些无意中发现的VS2005中没有明确指出的快捷键共享出来，并不是所有的快捷键，或者常见的一些快捷键。

1、Ctrl+Space直接完成类或函数（本来这个并不算隐藏的快捷键，但是因为中文输入法抢占这个快捷键，所以。。。，替代的快捷键是Alt+Right）

2、Shift+Delete整行删除，并且将这一行放到剪贴板（这时候不能选中一段内容）

3、Shift+Insert粘贴，有点匪夷所思，Ctrl+V就可以了，大概是为了和Shift+Delete对应吧

4、Ctrl+Up，Ctrl+Down滚动编辑器，但尽量不移动光标，光标保证在可见范围内

5、Ctrl+BackSpace，Ctrl+Delete整词删除，有的时候很有用

6、Ctrl+Left，Ctrl+Right按整词移动光标（不算隐藏，和前面几条加起来就是Ctrl光标控制套件了）

7、Alt+Shift+F10打开执行改名，实现接口和抽象类的小窗口（还可以用Ctrl+.，不过有的中文输入法用到这个）

8、Shift+F9调试是打开QuickWatch，内容是当前光标所在处的内容

9、F12转跳到定义，很有用的快捷键

10、Shift+F12查找所有引用

VS2008快捷键大全

Ctrl+m+Crtr+o折叠所有大纲

Ctrl+M+Crtr+P: 停止大纲显示

Ctrl+K+Crtr+C: 注释选定内容

Ctrl+K+Crtr+U: 取消选定注释内容

Ctrl+J : 列出成员 智能感知

Shift+Alt+Enter: 切换全屏编辑

Ctrl+B,T / Ctrl+K,K: 切换书签开关

Ctrl+B,N / Ctrl+K,N: 移动到下一书签

Ctrl+B,P: 移动到上一书签

Ctrl+B,C: 清除全部标签

Ctrl+I: 渐进式搜索

Ctrl+Shift+I: 反向渐进式搜索

Ctrl+F: 查找

Ctrl+Shift+F: 在文件中查找

F3: 查找下一个

Shift+F3: 查找上一个

Ctrl+H: 替换

Ctrl+Shift+H: 在文件中替换

Alt+F12: 查找符号(列出所有查找结果)

Ctrl+Shift+V: 剪贴板循环

Ctrl+左右箭头键: 一次可以移动一个单词

Ctrl+上下箭头键: 滚动代码屏幕，但不移动光标位置。

Ctrl+Shift+L: 删除当前行

Ctrl+M,M: 隐藏或展开当前嵌套的折叠状态

Ctrl+M,L: 将所有过程设置为相同的隐藏或展开状态

Ctrl+E,S: 查看空白

Ctrl+E,W: 自动换行

Ctrl+G: 转到指定行

Shift+Alt+箭头键: 选择矩形文本

Alt+鼠标左按钮: 选择矩形文本

Ctrl+Shift+U: 全部变为大写

Ctrl+U: 全部变为小写

代码快捷键

Ctrl+Shift+空格键 / Ctrl+K,P: 参数信息

Ctrl+K,I: 快速信息

Ctrl+E,U / Ctrl+K,U: 取消选定注释内容

Ctrl+K,M: 生成方法存根

Ctrl+K,X: 插入代码段

Ctrl+K,S: 插入外侧代码

F12: 转到所调用过程或变量的定义

窗口快捷键

Ctrl+W,W: 浏览器窗口

Ctrl+W,S: 解决方案管理器

Ctrl+W,C: 类视图

Ctrl+W,E: 错误列表

Ctrl+W,O: 输出视图

trl+W,P: 属性窗口

Ctrl+W,T: 任务列表

Ctrl+W,X: 工具箱

Ctrl+W,B: 书签窗口

Ctrl+W,U: 文档大纲

Ctrl+D,B: 断点窗口

Ctrl+D,I: 即时窗口

Ctrl+Tab: 活动窗体切换

Ctrl+Shift+N: 新建项目

Ctrl+Shift+O: 打开项目

Ctrl+Shift+S: 全部保存

Shift+Alt+C: 新建类

Ctrl+Shift+A: 新建项

Shift+Alt+Enter: 切换全屏编辑

Ctrl+B,T / Ctrl+K,K: 切换书签开关

Ctrl+B,N / Ctrl+K,N: 移动到下一书签

Ctrl+B,P: 移动到上一书签

Ctrl+B,C: 清除全部标签

Ctrl+I: 渐进式搜索

Ctrl+Shift+I: 反向渐进式搜索

Ctrl+F: 查找

Ctrl+Shift+F: 在文件中查找

F3: 查找下一个

Shift+F3: 查找上一个

Ctrl+H: 替换

Ctrl+Shift+H: 在文件中替换

Alt+F12: 查找符号(列出所有查找结果)

Ctrl+Shift+V: 剪贴板循环

Ctrl+左右箭头键: 一次可以移动一个单词

Ctrl+上下箭头键: 滚动代码屏幕，但不移动光标位置。

Ctrl+Shift+L: 删除当前行

Ctrl+M,M: 隐藏或展开当前嵌套的折叠状态

Ctrl+M,L: 将所有过程设置为相同的隐藏或展开状态

Ctrl+M,P: 停止大纲显示

Ctrl+E,S: 查看空白

Ctrl+E,W: 自动换行

Ctrl+G: 转到指定行

Shift+Alt+箭头键: 选择矩形文本

Alt+鼠标左按钮: 选择矩形文本

Ctrl+Shift+U: 全部变为大写

Ctrl+U: 全部变为小写

# <C#入门经典>

字符串中的每对花括号都是一个占位符，包含列表中每个变量的内容。占位符的总数应等于列表中指定的变量数。

逐字指定的字符串：@"C:\Temp\MyDir\MyFile.doc" 等价于 C:\\Temp\\MyDir\\MyFile.doc

@"A short list:

item 1

item 2"

引用参数与值参数：

static void showDouble(ref int val)

{

val \*= 2 ;

}

int myNumber = 5 ;

showDouble(ref myNumber) ;

用作ref参数的变量有两个限制：因为函数可能会改变引用参数的值，所以必须在函数调用中使用“非常量”变量。同时该变量必须已初始化。

委托：函数指针

delegate double processDelegate(double param1, double param2) ;

static double multiply(double param1, double param1)

{

return param1 \* param2 ;

}

static double divide(double param1, double param1)

{

return param1 / param2 ;

}

static void executeFunction(processDelegate process) ;

{

if (input == "M") process = multiply ;

else

process = divide ;

process(2.2, 3.3) ;

}

输出调试信息：

1.output窗口

using System.Diagnostics ;

Debug.WriteLine(string.Format("",)) ;

Trace.WriteLine(string.Format("",)) ;

Debug.WriteLineIf() ;

Trace.WriteLineIf() ;

2.调试输出写入日志文件。

数组与集合的主要区别：集合通常实现额外的功能，例如add() and remove() 方法可添加和删除集合中的项。而集合通常有一个Item属性，它根据对象的索引返回该对象。

类修饰符

项目内访问修饰符：

internal 默认

abstract 抽象类

sealed 不能派生

跨项目访问修饰符：（即各个项目之间引用）

public

public abstract

public sealed

在一个solution中，类库项目编译为.dll程序集

solution -> add new project -> VC# -> Class Library, 添加类，编译成.dll文件

主项目引用.dll文件： Reference -> add reference -> ClassLibrary -> debug -> .dll

类内成员修饰符（包括字段，属性和方法）

public

private

protected 同C++

internal 成员只能由定义它的项目中的代码访问

protect internal只能由项目中派生类的代码来访问

定义字段

readonly 构造函数赋值

static

const 静态常量，直接字段赋值

定义方法

virtual 子类重写，一定要用override

abstract 同C++纯虚概念

override 重写基类方法，子类需要用override，当然不用override也行，也是隐藏基类方法，但会产生警告

override sealed 同java final概念

extern 同c声明

定义属性

状态（私有字段）访问器，同时对状态进行额外的操作，比如数据有效性验证

private int nValue ;

public int NValue

{

get {return nValue ; }

set {nValue = value ; }

}

常规方法：

//使用new，隐藏基类实现代码(隐藏不想要的继承成员），通过基类还是调用基类方法

public class Base1

{

public void DoSomething()

{}

}

public class Derived1 : Base1

{

public new void DoSomething()

{}

}

//使用override，使用多态性，通过基类可以调用子类方法

//欲在子类调用基类方法，可以用base.DoSomething()

public class Base2

{

public virtual void DoSomething()

{}

}

public class Derived2 : Base2

{

public override void DoSomething()

{}

}

注意：关键字base(指父类，因为单继承，java里用super)和this指代对象实例，所以不能用于类静态方法和变量

同C++,C#内可以嵌套类，嵌套类一般作为辅助类,是个私有类

部分类定义 （把类的定义放在多个文件中），对windows应用程序隐藏与窗体布局相关的代码有很大的作用。

1.可以把字段属性构造函数放在一个文件中，而把方法放在另一个文件中

2.分别实现各个接口

public partial class MyClass : IMyInterface1

{}

public partial class MyClass : IMyInterface2

{}

和如下等价

public partial class MyClass : IMyInterface1, IMyInterface2

{}

第11章 集合、比较和转换

//注意，多态是不需要类型转换，非多态，基类调用派生类新方法，需要类型转换，同C++ down\_cast

简单数组：

Animal[] animalArr = new Animal[2] ;

animalArr[0] = new Cow() ;

animalArr[1] = new Chicken() ;

((Cow)animalArr[0]).Milk() ;

((Chicken)animalArr[1].LayEgg() ;

复杂数组：

ArrayList集合是System.Object对象的集合（通过多态性），对于非多态，同样需要类型转换

ArrayList animalArrayList = new ArrayList() ;

animalArrayList.Add(new Cow()) ;

animalArrayList.Add(new Chicken()) ;

定义集合：从CollectionBase中派生，或自己实现集合接口，例如IEnumerable, ICollection and IList,一般需要为集合定义一个索引器，以使用collection[index]语法来访问集合成员

using System.Collections ;

public class Animals : CollectionBase

{

public void Add(Animal animal)

{

List.Add(animal) ;

}

public void Remove(Animal animal)

{

List.Remove(animal) ;

}

public Animal this[int index]

{

get { return (Animal)List[index] ; }

set { List[index] = value ; }

}

}

定义字典：

public class Animals : DictionaryBase

{

public void Add(string id, Animal animal)

{

Dictionary.Add(id, animal) ;

}

public void Remove(string id, Animal animal)

{

Dictionary.Remove(id) ;

}

public Animal this[string id]

{

get {return (Animal)Dictionary[id] ; }

set { Dictionary[id] = value ; }

}

}

迭代器

IEnumerable and IEnumerator 接口负责foreach循环，是一个代码块，按顺序提供了要在foreach循环中的使用的所有值

如果要迭代一个类，可使用方法GetEnumberator(),其返回类型是IEnumberator

如果要迭代一个类成员，例如一个方法，则使用IEnumerable

public static IEnumerable SimpleList()

{

yield return "string 1" ;

yield return "string 2" ;

yield return "string 3" ;

}

foreach (string item in SimpleList()

{

Console.WriteLine(item) ;

}

public new IEnumberator GetEnumerator()

{

foreach(object animal in Dictionary.Value)

{

yield return (Animal)animal ;

}

}

foreach (Animal animal in animalCollection)

{

Console.WriteLine("New {0} object added to custom collection, " +

"Name = {1}", animal.ToString(), animal.Name()) ;

}

类型比较：

if (obj.GetType() == typeof(Class))

{}

或用is 判断对象是否是某类型

public void Check(object param1)

{

if (param1 is ClassA)

{}

else (param1 is ClassB)

{}

}

类比较：常用于集合排序等功能

//ArrayList的方法Sort()，不带参数是使用默认的比较方式，即调用类实现接口IComparable的函数public void CompareTo(object obj)

//带参数，即调用类实现接口IComparer的函数public int Compare(object x, object y)

class Person : IComparable

{

public string Name {set; get;}

public int Age {set; get;}

public Person(string name, int age)

{

Name = name ;

Age = age ;

}

public int CompareTo(object obj)

{

if (obj is Person)

{

Person otherPerson = obj as Person ;

return this.Age - otherPerson.Age ;

}

}

}

public class PersonComparerName : IComparer

{

public static IComparer Default = new PersonComparerName() ;

public int Compare(object x, object y)

{

if (x is Person && y is Person)

{

return Comparer.Default.Compare( ((Person)x).Name, ((Person)y).Name) ;

}

}

}

static void test()

{

ArrayList list = new ArrayList() ;

list.Add(new Person("Jim", 20) ;

list.Add(new Person("Bob", 25) ;

list.Sort() ;

list.Sort(PersonComparerName.Default) ;

}

类型转换

重载转换运算符：显式explicit和隐式implicit转换（同C++类型转换运算符）

public calss ConvClass1

{

public int Val {set ; get ; }

public static implicit operator ConvClass2(ConvClass1 op1)

{

ConvClass2 returnVal = new ConvClass2() ;

returnVal.Val = op1.Val ;

return returnVal ;

}

}

as运算符把一种类型转换为指定的引用类型，如果不能进行转换，as运算符就返回null值

所以在C#应用程序中常见如下代码结构：

public void MilkCow(Animal animal)

{

Cow cow = animal as Cow ;

if (cow != null)

{

cow.Milk() ;

}

else

{

Console.WriteLine("") ;

}

}

XML: Extensible Markup Language(可扩展标记语言,XML)

XML在.NET中执行大量的任务，包括描述应用程序的配置，在Web服务之间传输信息等。

XML是一种以简单文本格式存储数据的方式，这意味着它可以被任何计算机读取。XML是在Internet上传输数据的绝佳格式，目前.NET中的大多数应用程序都以某种形式使用XML，例如存储配置细节的.config文件，WPF中使用的XAML文件等。甚至Office2007引入的新文档格式都基于XML，

XML实际上不是语言，而是定义语言的标准（称为XML应用），XML分析程序(parser)读取XML文档，并分析其中各个元素，它们会拒绝任何包含非法XML的文档

XML语法：

<?xml version=”1.0” encoding=”utf-8”?> //xml声明定义xml版本

//根节点，支持XSD模式，且将XML文档与XSD模式文件关联,XML名称空间xsi

<books xmlns:xsi=”[http://www.w3.org/2001/XMLSchema-instance” xsi:noNamespaceSchemaLocation=](http://www.w3.org/2001/XMLSchema-instance)”c:\XMLConver.xsd”>

<book title=”my love story” > //XML特性在元素的开始标记内

<author>qzlin</author> //XML元素

</book>

</books>

XML元素 = 开始标记< author > + 元素中的数据qzlin + 结束标记 </author>

元素可以嵌套

XML特性在元素的开始标记内如title=”my love story”

XML模式用于定义XML文档的结构，需要与第三方交换信息时，模式特别有用

格式良好并有效的XML：

1. 有且只有一个根元素
2. 每一个元素都有结束标记
3. 没有重叠元素（所有子元素必须完全嵌套在父元素内）
4. 所有特性必须放在引号内

程序中的XML：

.NET Framework为创建和处理XML提供一系列类，可以使用XML存储应用程序配置，把数据保存到磁盘上，在线上传信息等

Using System.Xml;

XmlDocument document = new XmlDocument() ;

Document.Load(@”file path”) ;

XmlElement element = document.DocumentElement ; //XML文档根节点

//遍历,修改，插入，删除，选择节点等见类方法

XmlNode,XmlDocument,XmlElement,XmlAttribute,XmlText,XmlComment,XmlNodeList

# <C#与.NET4高级程序设计>

CIL： common intermediate language公共中间语言

CLR: 公共语言运行库

CTS: 公共类型系统 （完整描述运行库所支持的所有可能的数据类型和编程结构）

CLS: 公共语言规范 （定义一个让所有.NET语言都支持的公共类型和编程结构的子集）

JIT： just in time, 即时编译

.NET可以理解为一个运行库环境和一个全面的基础类库，运行库层是CLR,其主要作用是为我们定位，加载和管理.NET类型，同时也负责一些低层细节的工作，如内存管理，应用托管，处理线程，安全检查等。

基础类库

数据库访问

GUI桌面API

安全

远程处理API

线程

文件输入/输出

Web API

其他

CLR

在.NET运行库下执行的代码称为托管代码(managed code)，不能直接在.NET运行库承载(host)的代码称为非托管代码，可以通过C++/CLR来封装标准C/C++.

某种支持.NET的语言编写的.NET源代码

对应的.NET编译器

\*.dll或\*.exe程序集(CIL,元数据和清单)

基础类库

(mscorlib.dll等)

.NET执行引擎(mscoree.dll)

类加载

JIT

特定平台的指令

执行成员

当用户程序引用一个程序集，要使用它时，mscoree.dll将首先自动加载，然后由它负责将需要的程序集导入内存，运行时引擎负责许多任务，首要的任务是负责解析程序集的位置，并通过读取其中包含的元数据，在二进制文件中发现所请求的类型。接着，CLR在内存中为类型布局，将关联的CIL编译成特定平台的指令，执行所有需要的安全检查，然后运行当前的代码。

.NET命令空间

System

在System内，你将会发现很多有的类型，可以用来处理内建数据，数学计算，随机数的产生，环境变量，垃圾收集器以及一些常见的异常和特性

System.Collections

System.Collections.Generic

这些命名空间定义了一些集合容器类型，还有一些基类型和接口，使你有可能构建自定义的收集器

System.IO

这些命名空间定义了许多处理文件I/O，数据压缩和端口操作的类型

System.Reflection

这些命名空间定义了一些类型，支持运行时类型发现与类型的动态创建

System.Runtime.InteropServices

这些命名空间提供了一些设施，使得.NET类型可以与“非托管”代码交互（例如基于C的DLL和COM服务器），或反过来

System.Drawing

System.Windows.Forms

这些命名空间空义了使用.NET原始UI工具包（Windows Forms）来构建桌面应用程序所用到的类型

System.Windows

System.Windows.Controls

System.Windows.Shapes

命名空间是一些表示WPF UI工具包的几个命名空间的根

System.Linq

System.Xml.Linq

System.Data.DataSetExtensions

这些命名空间定义了针对 LINQ API编程时用到的类型

System.Web

这个命名空间用来构建ASP.NET Web应用程序

System.ServiceModel

这个命名空间用来通过 WCF API构建分布式应用程序

System.Threading

System.Threading.Tasks

这个命名空间定义了可以用来构建多线程应用程序（将工作负载到多个CPU）上的类型

System.Xml

这个以XML为中心的命名空间包括众多用于与XML数据交互的类型

常用工具：ildasm.exe, reflector探索程序集

Class view, object browser, refactor,

class designer, class designer toolbox（模型驱动开发）

定义main()方法的类叫做应用程序对象，一个可执行程序可以有多个应用程序对象（在执行单元测试的时候可能有用），但是我们必须通过命令行编译编译器的/main选项或通过位于项目属性Application选项卡内的Startup Object下拉列表框来通知编译器将哪个main()方法用作入口点。

System.Environment.GetCommandLineArgs();//获取应用程序名称和命令行参数

System.Environment.ProcessorCount;//获取系统内核数,还有很多操作系统的细节

String and StringBuilder（注意using System.Text）的区别:

String修改时会产生副本

StringBuilder，直接修改对象内部的字符数据，而不是获取按修改后格式的数据副本。

CLR不能应用窄化运算，所有窄化转换会导致编译器错误，因为C#是类型安全的，因为对于宽数据转换为窄数据，需要显示强制转换

隐式类型本地变量var variableName = initialValue;

编译器会根据本地数据点的初值来自动推断实际的数据类型。

隐式类型只能用于方法或属性范围内的本地变量，且声明时需要分配初始值

隐式类型，只有在定义LINQ查询的返回数据时才应该使用，因为LINQ技术使用了查询表达式，它可以根据表达式本身的格式产生动态创建的结果集，由于在某些情况下根本无法显式定义查询的返回类型，这时隐式类型就非常有用了。

参数修饰符params:这个参数修饰符允许将一组可变数量的参数作为单独的逻辑参数进行传递，方法可能有一个params修饰符，而且必须是方法的最后一个参数，事实上你不会经常使用params修饰符，但要知道的是，基础类库中的许多方法都使用了这个C#语言特性.

参数修饰符out, ref (二者都可以在不同作用域改变变量的值)

带有输出参数Out的方法有义务在退出这个方法之前，必须给参数赋值

Ref参数必须在它们被传递给方法之前初始化，因为在传递一个对已存在变量的引用。

Static void Add(int x, int y, out int ans)

{

ans = x + y ;

}

static void SwapStrings(ref string s1, ref string s2)

{

String tempStr = s1 ;

S1 = s2 ;

S2 = tempStr ;

}

Static void main()

{

Int ans ;

Add(1, 1, out ans) ;

String s1 = “Flip” ;

String s2 = “Flop” ;

SwapStrings(ref s1, ref s2) ;

}

一维数组：int[] iArr = new int[256] ;

二维数组：int[,] iMat = new int[512, 512] ;

交错数组：int[][] myJagArr = new int[9][] ; (如九九乘法表)

For (int i=0; i<9; ++i) myJagArr[i] = new int[i+1] ;

浅复制：两个独立的结构，每一个都包含指向内存中同一个对象的引用

深复制：内部引用的状态完全复制到一个新对象中，需要实现ICloneable接口。

按值传递引用类型，被调用者可能改变对象的状态数据的值，但不能改变所引用的对象。如

Void SendAPersonByValue(Person p) //等价于C++ Person& const p

{

p.PersonAge = 20 ; //引用同一对象，可以改变对象状态

p = new Person(“Nikki”, 99) ; //无法对引用重新赋值新的对象

}

按引用传递引用类型，被调用者可能改变对象的状态数据，也可能改变所有引用的对象

Void SendAPersonByValue(ref Person p) //等价于C++ Person& p

{

p.PersonAge = 20 ; //可以改变对象状态

p = new Person(“Nikki”, 99) ; //引用赋新的对象

}

可空类型 ?, 如int?等价于System.Nullable<int>,主要用于涉及数据库编程时，可空数据类型可能特别有用，因为一个数据表中的列可能有意是空的。如未定义

??操作符

//从GetIntFromDatabase()返回的值为null时，将本地变量赋值为100

Int myData = dr.GetIntFromDatabase() ?? 100 ;

等价于如下代码：

Int? moreData = dr.GetIntFromDatabase() ;

If (!moreData.HasValue)

moreData = 100 ;

由构造函数通常会验证传入的参数来强制各种业务规则，所以在类的构造函数集合中经常会找到冗余的验证逻辑。解决方法如下：

让一个接受最多参数个数的构造函数做“主构造函数”，并且实现必须的验证逻辑，其余的构造函数可以使用this关键字把传入的参数转发给主构造函数，并且提供所有必须的其他参数。如：

Class Motorcycle

{

Int driverIntensity ;

String driverName ;

Public Motocycle() {}

Public Motorcycle(int intensity)

: this(intensity, “”)

{}

//主构造函数完成所有实际工作

Public Motorcycle(int intensity, string name)

{

If (intensity > 10) intensity = 10 ;

driverIntensity = intensity ;

driverName = name ;

}

}

采用可选参数方式：允许我们对传入参数提供默认值

Class Motorcycle

{

Public Motorcycle(int intensity=0, string name=””)

{

If (intensity >10 ) intensity = 10 ;

driverIntensity = 10 ;

driverName = name ;

}

}

Void MakeSomeBikes()

{

Motocycle m1 = new Motorcycle() ;

Motocycle m2 = new Motorcycle(name:”Tiny”) ;

Motocycle m3 = new Motorcycle(7) ;

}

静态类等价于c++ namespace,主要用作工具类比如System.Math, System.Console, System.Environment, System.GC

属性：

封装概念的核心是：对象的内部数据不应该从对象实例直接访问，如果调用者想改变对象的状态，就要使用传统的setter and getter方法。.NET语言提供使用属性来强制数据封装状态数据，属性总是映射到实际中的setter and getter方法。

如：

Class Employee

{

Private string empName ;

Public string Name //注意属性名可以和对应字段名不同，只要数据类型相同

{

Get {return empName} ;

Set

{

If (value.Length >15) Console.Writeline(“Error!”)

Else empName = value ;

}

}

}

一旦创建了属性，对调用者来说好像是获取或设置公共数据点，但在背后会调用相应的get or set块来保持封装。

属性可以结合C#内部操作符进行使用。

Employee joe = new Employee() ;

Joe.empName += “Zhou” ;

尽量使用属性，而不需要对私有字段的直接操作。通常情况下类的构造函数会接收传入参数，检查有效数据，然后赋值给内部私有字段。会显得特别复杂，可以验证数据有效性留给属性去做，除了更新构造函数使用属性分配值外，在整个类实现中一直使用属性也是不错的做法，这可以确保业务规则总是被强制检查，在多数情况下，唯一需要直接使用私有数据的情况是在属性内部。

如果用ildasm.exe打开程序集，将看到每一个属性实际上被映射到由CLR内部调用的隐藏的get\_XXX()/set\_XXX()方法。

C#喜欢用属性来封装数据，句法实体与传统的访问方法，修改方法对的使用目的是相同的，属性的优点是，对象的用户可以只使用一个命名项就能操作内部数据。

大多数C#都在set作用域中包含业务逻辑，但仅仅设置并直接返回私有字段的情况也很多，因此可以采用，自动属性语法：

Class Car

{

Public strinig PetName {get; set;}

}

自动属性语法，自动生成的私有返回字段的名称在C#代码库中是不可见的，查看它的唯一方法是使用ildasm.exe这样的工具。

对象初始化语法：只是使用默认构造函数创建类变量并设置各个属性状态数据

如：

Class Point

{

Public int X {get; set;}

Public int Y {get; set;}

Public Point(int x, int y)

{

X = x ;

Y = y ;

}

Public Point() {}

}

Void main()

{

//通过手动设置各个属性

Point pt0 = New Point() ;

Pt0.X = 10 ;

Pt0.Y = 10 ;

//通过自定义构造函数

Point pt1 = new Point(10, 10) ;

//使用对象初始化语法

Point pt2 = new Point{X=10, Y=10} ;

}

const常量数据，类的常量字段是隐式静态的。

Readonly只读字段，同常量数据一样，在赋初始后不可改变，然而和常量不同的是，赋给只读字段的值可以在运行时决定（但只能在构造函数里赋值），常常用于直到运行时才知道字段值（可能我们需要读取外部文件来获得值），并且希望之后值不会被改变

Namespace ConstData

{

Class MyMathClass

{

Public const double PI = 3.14 ;

}

Class Program

{

Public readonly double PI ; //只能在构造函数中为只读字段赋值

Static void main()

{

Double area = MyMathClass.PI \* 4\*4 ;

}

}

}

继承

Sealed: 防止继承,同java final

Sealed class MiniVan : Car

{}

Class SalesPerson : Employee

{

Public override sealed void GiveBonus(float amout)

{}

}

包含/委托编程:委托就是增加公共成员到包含类，以便使用被包含对象的功能。其实就是限制型代理模式

Shadowing（成员投影）：C#提供了逻辑上和方法重写相对的功能，正式地说，如果派生类定义的成员和定义在基类中的成员一致，派生类就投影了父类的版本。其实就是隐藏基类成员，说明is-a关系不好，建议用has-a关系，当然，想用，可以通过显示声明派生类型的实现故意设计为隐藏父类的版本

Class ThreeDCircle : Circle

{

Public new void Draw()

{}

}

子类可以隐式转换为父类，若父类需要转换为子类，用如下语法

1. 强制转换

Object obj = new ThreeDCircle() ;

ThreeDCircle circle = (ThreeDCircle) obj ;

1. As 返回null or 非null

ThreeDCircle circle = obj as ThreeDCircle ;

If (circle == null)

1. Is 返回true/false

If (obj is ThreeDCircle)

{}

结构化异常处理：

Bug:由程序员一方引起的错误，比如内存泄漏

用户错误：由运行程序的用户引起，比如没处理错误输入，用户在文本框中输入格式非法的字符串

异常：异常往往是运行时的非正常情况，在编程时很难估计到，异常可能包括试图连接一个已经不存在的数据库，打开已被破坏的XML文件，连接当前处于离线状态的机器等。

.NET中，异常解释为bug,用户错误输入和运行时错误

系统级异常和应用程序级异常

.NET基础类库提供了几百个预定义的接口类型，由各种类和结构实现，自定义类和结构完全可以实现这些预定义的接口，以支持对象克隆，对象枚举和对象排序等的高级行为

接口类型和抽象基类的区别：

抽象基类中的抽象成员只应用到派生类型，因此我们不能在不同的层次结构中对类型进行配置，让它们支持相同的多态接口。

接口具有较高级别的多态性，派生于接口类型可以不具有相同的父类，即独自继承体系，但我们可以通过 接口类型对它们进行多态处理

抽象基类除一组抽象成员，还可以有构造函数，字段数据，非抽象成员（具体实现），而接口只能包含抽象成员，属性，事件及索引器

多接口，需要通过显式接口实现解决命名冲突,如下

Public interface IDrawToForm

{

Void Draw() ;

}

Public interface IDrawToMemory

{

Void Draw() ;

}

Public interface IDrawToPrinter

{

Void Draw() ;

}

Class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter

{

Void IDrawToForm.Draw() {}

Void IDrawToMemory.Draw() {}

Void IDrawToPrinter.Draw() {}

}

Static void main()

{

Octagon oct = new Octagon() ;

((IDrawToForm) oct).Draw() ;

((IDrawToMemory) oct).Draw() ;

((IDrawToPrinter) oct).Draw() ;

}

.NET标准接口

构建可枚举类型 IEnumerable, IEnumerator

主要用来支持foreach，遍历可枚举类型

//系统提供的接口

Public interface IEnumerable

{

IEnumerator GetEnumerator() ;

}

Public interface IEnumerator

{

Bool MoveNext() ;

Object Current {get;}

Void Reset() ;

}

Using System.Collections ;

Public class Garage : IEnumerable

{

Private Car[] carArray = new Car[4] ;

Public Garage()

{

carArray[0] = new Car(“FeeFee”, 200, 0) ;

…

}

Public IEnumerable GetEnumerator()

{

//这里有个技巧，因为System.Array,和其他许多类型已经实现可枚举接口，直接复用即可，一般情况下，这个方法的实现只是交给保存子对象的内部成员

Return carArray.GetEnumerator() ;

}

//扩展知识：实际上还可以用yield构建迭代器方法，来实现可枚举

Public IEnumerable GetEnumerator()

{

Foreach (Car c in carArray)

{

Yield return c ;

}

}

//实现带参数的迭代器

Public IEnumerable GetTheCars(bool ReturnReversed)

{

If (ReturnReversed)

For (int I = carArray.Length; I != 0; I --)

Yield return carArray[i-1] l

Else

Foreach (Car c in carArray)

Yield return c ;

}

}

Static void main()

{

Garage carLot = new Garage() ;

//常规使用

Foreach (Car c in carLot)

{}

//带参数使用

Foreach (Car in carLot.GetTheCars(true))

{}

}

构建可克隆对象ICloneable

//系统提供的接口

Public interface ICloneable

{

Object Clone() ;

}

Public class Point : ICloneable

{

Public int X {get; set;}

Public int Y{get; set;}

Public Point(int xPos, int yPos) { X = xPos; Y = yPos ; }

Public object Clone()

{ return new Point(this.X, this.Y); }

}

Static void main()

{

Point p1 = new Point(100, 100) ;

Point p2 = p1 ; //p2与p1引用同一个对象

Point p3 = (Point) p1.Clone() ; //p3与p1不同对象

}

构建可比较的对象IComparable

允许对象可基于某些特定键值进行排序

//系统提供的接口

Public interface IComparable

{

Int CompareTo(object o) ;

}

public class Car : IComparable

{

//第一种方法

Int IComparable.ComareTo(object obj)

{

Car temp = obj as Car ;

If (temp != null)

//数据类型，string类型等实现了IComparable，可以直接复用

Return this. CarID.CompareTo(temp.CarID) ;

}

//第二种方法

扩展知识：也可以通过IComparer接口辅助类中实现

//系统提供接口

Interface IComparer

{

Int Compare(object o1, object o2) ;

}

Public class PetNameComparer : IComparer

{

Int IComparer.Compare(object o1, object o2)

{

Car t1 = o1 as Car ;

Car t2 = o2 as Car ;

If (t1 != null && t2 != null)

Return String.Compare(t1.PetName, t2.PetName) ;

}

}

public static IComparer SortByPetName

{

Get ｛ return (IComparer) new PetNameComparer() ; ｝

}

｝

Static void main()

{

Garage carLot = new Garage() ;

Array.Sort(carLot) ; //使用IComparable为它们排序

Array.Sort(carLot, Car.SortByPetName) ;

}

任何使用.NET2.0或更高版本创建的项目都应该放弃旧的非泛型类，而使用相应的泛型类，为什么？

1. 性能问题

Public class ArrayList : object, IList, ICollection, IEnumerable, ICloneable

{

Public virtual int Add(object value) ;

…

}

非泛型 集合操作的都是System.Object,所有东西都可以当成System.Object,所以非泛型集合可以包含值类型数据或引用类型，但如果要在程序中将值类型赋给引用类型，需要对值类型进行装箱，同样将引用类型赋给值类型要进行拆箱操作

Static void WorkWithArrayList()

{

ArrayList myInts = new ArrayList() ;

myInts.Add(10) ; //将值类型赋给object，值类型将自动装箱

int I = (int)myInts[0] ; //将object转换回栈数据时，会发生拆箱

}

步骤如下：

1. 必须在托管堆上分配一个新对象
2. 基于栈数据的值必须被转移到新分配的内存位置
3. 在拆箱时，保存在堆对象中的值必须转移回栈
4. 堆上无用的对象（最后）会被回收
5. 类型安全问题

由于System.Collections中的大多数类所操作的都是System.Object,因此它们可以容纳任何类型，如下

Static void ArrayListRandomObjects()

{

//ArrayList可以保存任何类型

ArrayList allMyObjects = new ArrayList() ;

allMyObjects.Add(true) ;

allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0))) ;

allMyObjects.Add(3.14) ;

}

大多数情况下，需要一个类型安全的容器来操作特定的数据类型

Using System.Collections.Generic ;

Dictionary<TKey, TValue> 名/值对

List<T> 动态改变大小的顺序列表

LinkedList<T> 双向链表

Queue<T> 先入先出队列

SortedDictionary<TKey, TValue> 排序的名称/值对

SortedSet<T> 排序的不重复对象集合

Stack<T> 后入先出堆栈

//集合初始化语法同对象初始化语法

List<int> myGenericList = new List<int> {0, 1, 2, 3} ;

//创建自定义泛型方法

Static void Swap<T>(ref T a, ref T b)

{

T temp = a;

a = b ;

b = temp ;

}

//使用时可省略类型参数，因为编译器会基于成员参数推断类型参数

Swap(ref b1, ref b2) ;

不过，建议加上类型参数，好的编程风格Swap<bool>(ref b1, ref b2);

//创建自定义泛型结构和类

public struct Point<T>

{

public T xPos { get; set;}

public T yPos { get ; set;}

Public Point()

{

xPos = default(T) ; //表示一个类型参数的默认值

yPos = default(T) ;

}

}

正常情况下，构建泛型类层次结构的机会微乎其微，所以一般继承自一个泛型必须指定一个类型参数

Public class MyList<T>

{

Private List<T> listOfData = new List<T> ;

Public virtual void Insert(T data) ;

}

Public class MyStringList : MyList<String>

{

//注意：在创建泛型方法时，如果对类型参数应用任何C#操作符（+,-,\*，==等）都将产生编译器错误

Public override void Insert(string data) {} ;

}

//泛型类型参数的约束

//派生自object, 并且包含项为实现了IDrawable的类，并且必须支持默认构造函数

Public class MyGenericClass<T> where T : class, IDrawable, new()

{}

委托，事件和Lambda

在.NET平台下，委托类型用来定义和响应应用程序中的回调。同C++函数指针不同，.NET委托是内置支持多路广播和异步方法调用的对象。

Window API经常使用C语言风格的函数指针来创建称为回调函数的实体。然而有个问题，它们除了原始内存地址外无法表示其他信息，而理想中的回调应该包含更多类型安全信息。如参数的数量与类型，所指向的方法的返回值等。因此经常成为bug，崩溃和其他运行时灾难的源头。

因此在.NET平台上，用委托delegate来完成，委托是一个类型安全的对象，它指向程序中另一个以后会被调用的方法（或多个方法）

public delegate string MyOtherDelegate(out bool a, ref bool b, int c) ;

//委托类型后台代码

sealed class MyOtherDelegate : System.MulticastDelegate

{

//同步方式调用委托对象维护的每个方法,也就是说，必须等待调用完成才能继续执行

Public string Invoke(out bool a, ref bool b, int c) ;

//在第二个执行线程上异步调用当前方法，用于多线程，当第二个执行线程是调用 比较耗时的方法

Public IAsyncResult BeginInvoke(out bool a, ref bool b, int c, AsyncCallback db, object state) ;

Public string EndInvoke(out bool a, ref bool b, IAsyncResult result) ;

}

Public abstract class System.MulticastDelegate : system.Delegate

{

//返回所指向的方法列表

Public sealed override Delegate[] GetInvocationList() ;

…

}

Public abstract class System.Delegate : ICloneable, ISerializable

{

//与函数列表交互的方法，其实+=, -=调用如下方法

Public static Delegate Combine(Delegate a, Delegate b) ;

Public static Delegate Remove(Delegate source, Delegate value) ;

//返回静态方法的详细信息

Public MethodInfo Method {get;}

//返回委托维护的方法的对象

Public object Target {get ;}

}

.NET委托内置支持多路广播，也就是说，一个委托对象可以维护一个可调用方法的列表而不只是单独一个方法,给一个委托对象添加多个方法时，不用直接分配，重载+=操作符即可。

协变与逆变

只有接口和委托类型能声明为协变量或逆变性

协变只能用于返回类及相关继承体系的方法，不能用于更改数据的方法

//接口协变

Interface IRetrieveWrapper<out T>

{

T GetData() ;

}

Class Wrapper<T> : IRetrieveWrapper<T>

{

Private T storedData ;

T IRetrieveWrapper<T>.GetData()

{

Return this.storedData ;

}

}

Wrapper<string> stringWrapper = new Wrapper<string>() ;

IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper ;

//以下是合理的，因为协变表明只会返回类及相关继承体系

IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper ;

//委托协变

Public delegate Car ObtainVehicleDelegate() ;

Public static Car GetBasicCar() { return new Car(); }

Public static SportsCar GetSportsCar() { return new SportsCar() ; }

Static void main()

{

ObtainVehicleDelegate targetA = new ObtainVehicleDelegate(GetBasicCar) ;

Car c = targetA() ; //标准用法

//委托协变才允许返回继承体系的方法

ObtainVehicleDelegate target = new ObtainVehicleDelegate(GetSportsCar) ;

SportsCar sc = (SportsCar) target() ;

}

逆变性：允许使用泛型接口，通过子类来引用父类的对象。

Public interface IComparer<in T>

{

Int Compare(T x, T y) ;

}

Class ObjectComparer : IComparer<Object>

{

Int Comparer<object>.Compare<Object x, Object y)

{

Int xHash = x.GetHashCode() ;

Int yHash = y.GetHashCode() ;

If (xHash == yHash) return 0 ;

If (xHash < yHash) return -1 ;

Return 1 ;

}

}

Object x = new Object() ;

Object y = new Object() ;

ObjectComparer objectComparer = new ObjectComparer() ;

IComparer<Object> objectComparator = objectComparer ;

Int result = objectComparator(x, y) ;

//以下是合理的

IComparer<String> stringComparator = objectComparer ;

事件与响应函数：同Qt里的信号和槽概念

定义事件步骤：

首先：定义一个委托类型，它包含在事件触发时将要调用的方法

其次：通过C# event关键字用相关委托声明这个事件

例子：

Public Class Car

{

//微软推荐的事件模式，委托的第一个参数是object,第二个参数是派生自System.EventArgs的子类型，object表示一个对发送事件的对象Car的引用，EventArgs则表示与该事件相关的信息，一般用来传递自定义数据

Public class CarEventArgs : EventArgs

{

Public readonly string msg ;

Public CarEventArgs(string message)

{ mes = message ; }

}

//定义委托类型

Public delegate void CarEngineHandler(object sender, CarEventArgs e) ;

//用event关键字用CarEngineHandler声明事件（声明信号一样），注意事件必须是委托类型

Public event CarEngineHandler AboutToBlow ;

//发起事件 （同发起信号一样）

Public void Accelerate(int delta)

{

…

AboutToBlow(this, “Careful buddy! Gonna blow!”) ;

…

}

}

//事件（回调函数）实现，同槽实现，***该回调函数一般是留给用户实现***

Public static void CarAboutToBlow(object sender, CarEventARgs e)

{

If (sender is Car)

{

Car c = (Car) sender ;

console.writeline(“{0} says: {1}”, sender, e.msg);

}

}

Static void main()

{

Car c = new Car() ;

c.AboutToBow += CarAboutToBlow ; //注册事件处理程序

c.Accelerate(10) ; //调用事件处理程序

}

实际上上面的代码过于复杂，一般常用如下的方法进行简化

Public Class Car

{

Public class CarEventArgs : EventArgs

{

Public readonly string msg ;

Public CarEventArgs(string message)

{ mes = message ; }

}

//泛型EventHandler<T>实现定义委托类型和事件声明，默认第一个参数是object,第二个参数是T类型

Public event EventHandler<CarEventArgs> Exploded ;

//发起事件 （同发起信号一样）

Public void Accelerate(int delta)

{

…

AboutToBlow(this, “Careful buddy! Gonna blow!”) ;

…

}

}

Static void main()

{

Car c = new Car() ;

//事件（回调函数）实现，同槽实现，***该回调函数一般是留给用户实现***

//注册事件处理程序,采用匿名方法实现

c.AboutToBow += delegate(object sender, Car.CarEventArgs e)

{

If (sender is Car)

{

Car c = (Car) sender ;

console.writeline(“{0} says: {1}”, sender, e.msg);

}

};

//注册事件处理程序,采用Lambda表达式实现，实际上内部是采用匿名

c.AboutToBow += (sender, e) => console.writeline(“{0} says: {1}”, sender, e.msg);

c.Accelerate(10) ; //调用事件处理程序

}

索引器方法，操作符重载，自定义类型转换，扩展方法，分部方法，匿名类型及不安全代码

//索引操作符[]访问包含在一个标准数组中的各个子项，索引器看上去和任何C#属性声明很相似。

Public class PeopleCollection : IEnumerable

{

Private ArrayList arPeople = new ArrayList() ;

Public Person this[int index]

{

Get { return (Person) arPeople[index] ; }

Set { arPeople.Insert(index, value) ; }

}

Private Dictionary<string, Person> listPeople = new Dictionary<string, Person>() ;

Public Person this[string name]

{

Get {return (Person) listPeople[name] ; }

Set {listPeople[name] = value ; }

}

Private int [,] my2DArray = new int[10, 10] ;

Public int this[int row, int col]

{}

}

Public class Point

{

//重载操作符，通常仅在构建实用工具类型时才有用。

//如果类型重载二元操作符，则简写赋值操作符(+=,-=)等将自动重载,C#要求，如果重载<,则必须重载>。<=与>=同理

Public static Point operator+ (Point p1, Point p2)

{

Return new Point(p1.x+p2.x, p1.y+p2.y) ;

}

//自定义类型转换

Public static explicity operator Point(Point3D pt) //显式类型转换

{

Return new Point(pt.x, pt.y) ;

}

Public static implicity operator Point(Pointf pt) //隐式类型转换

{

Return new Point(pt.x, pt.y) ;

}

}

//扩展方法：当需要为类型添加功能但并不拥有类型的已有代码时，当需要使类型支持一系列成员（为了实现多态）但不能改进类型的原始定义时，扩展方法能很好地帮助你解决问题，使用扩展方法，你可以为预编译的类型添加功能，同时这些方法将独立分开存放

定义扩展方法的限制：

1. 扩展方法必须 定义在静态类里，每个扩展方法也必须声明为静态的
2. 所有扩展方法都需要使用this关键字对第一个参数（并且仅对第一个参数）进行修饰
3. 每一个扩展方法只可以被内存中正确的实例调用，或者通过其所处的静态类被调用。

微软公司推荐把扩展方法放在独立的程序集，这样做只是为了减少编程环境的混乱

Static class MyExtensions

{

//本方法允许任何对象显示它所处的程序集

Public static void DisplayDefinigAssembly(this object obj)

{…}

//每一个Int32拥有一个方法Foo()

Public static void Foo(this int i)

{..}

//重载方法Foo()来接收字符串参数

Public static void Foo(this int , string msg)

{..}

//扩展接口方法,说明任何实现该接口的类现在都有该扩展方法

Public static int Subtract(this IBasicMath itf, int x, int y) ;

}

Static void main()

{

Int myInt = 123455 ;

myInt.DisplayDefiningAssembly() ;

myInt.Foo() ;

myInt.Foo(“Ints?”) ;

//使用如上扩展方法实际上调用静态类静态扩展方法

MyExtensions.DisplayDefineAssembly(myInt) ;

MyExtensions.Foo(myInt) ;

MyExtensions.Foo(myInt, ‘Ints?”) ;

}

//分部类（允许跨多个代码文件实现类型的语法），分部方法（允许方法原型和方法实现分开），分部方法常用来替代条件预编译，和实现轻量级事件回调

1. 分部方法的限制：
2. 分部方法只能定义在分部类中，
3. 分部方法必须返回void
4. 分部方法可以是静态的或实例级别的
5. 分部方法可以有参数，但不能具有out修饰符
6. 分部方法是隐式私有的

Partial class CarLocator

{

Public bool CarAvailableInZipCode(string zipCode)

{

…

//编译器会根据分部方法是否已经实现还是空的签名来决定方法是否应该放到程序集中，如果没有方法体的话，所有方法的使用痕迹（调用、元数据描述以及原型）都会在编译的时候去除

在某些情况下，C#分部方法是条件代码编译的强类型版本形式（通过#if, #elif, #else预处理指令），然而主要的区别是，如果分部方法没有提供实现的话，它会在编译周期（不管编译设置了什么）中被完全忽略。

使用分部方法来定义轻量级事件是很常见的，这个技术使得类设计者可以提供方法挂钩，就像事件处理程序一样，开发要员可以选择实现或不实现，根据命令约定，这样的轻量级事件处理方法使用On前缀

OnZipCodeLookup(zipCode) ;

…

}

//轻量级事件处理程序

Partial void OnZipCodeLookup(string make) ;

}

//用户代码

Partial class CarLocator

{

Partial void OnZipCodeLookup(string make)

{}

}

//匿名类型var，有时候你可能需要定义类来封装一些相关数据，但并不需要任何关联的方法、事件和其他自定义的功能。

Static void main()

{

//构建一个匿名对象myCar,虽然没有强类型定义，但C#编译器会在编译时自动生成名称唯一的类,需要通过对象初始化语法定义一系列来表示要封装的各个数据，所有的匿名类型都自动继承System.Object

Var myCar = new { Color = “Bright Pink”, Make = “Saab”, CurrentSpeed = 55} ;

//不安全代码，指针类型

Unsafe

{

Int myInt = 0;

Int\* ptrInt = & myInt ;

Char\* p =stackalloc char[256] ;

For (int k = 0; k < 256 ; ++ k)

P[k] = (char) k ;

Int size = sizeof(short) ;

}

}

Linq to Object

数据随处可见，如XML文件，关系数据库，内存中的集合，基元数组等。使用Linq，可以构建与数据库SQL查询类似的表达式，但Linq查询可以用于多种数据存储，甚至与关系型数据完全无关的存储

Linq相关的特性：隐式类型本地变量，对象/集合初始化语法，Lambda表达式，扩展方法和匿名类型

Linq查询返回的序列只有在编译时才能确定其数据类型，由于只有在应用程序编译之后才能知道其实际的数据类型，所以显然无法地声明变量，所以用隐式类型本地变量

关系数据： System.Data.dll, System.Data.SqlClient.dll

XML文档数据：System.Xml.dll

元数据表：System.Reflection命名空间

对象集合：System.Array, System.Collection, System.Collections.Generic

Linq语言级集成查询API的意图是提供一种统一且对称的方式，让程序员在广义的数据上得到和操作“数据”，通过使用Linq，我们能够在C#编程语言内直接创建被称为查询表达式（query expression）的实体，这些查询表达式是基于许多查询操作符(query operator)的，而且是有意设计成类似SQL表达式的

核心Linq程序集：

System.Core.dll -> Linq to Object

System.Data.DataSetExtensions.dll -> Linq to DataSet

System.Xml.Linq.dll -> Linq to XML

//Linq查询用于原始数组, 集合对象,还可以用于任何实现了IEnumerable<T>的类型

注意：Linq查询用于非泛型集合时，需要转换成兼容于IEnumerable<T>类型

Int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8} ;

Var subset = from I in numbers where I <10 select I ; //隐式类型

Foreach (var I in subset)

{}

//非泛型类型可以包含任何类型的项，还记得参数是object吗？实际上用OfType<T>可以初步筛选数据类型

ArrayList myCars = new ArrayList() { …} ;

Var myCarsEnum = myCars.OfType<Car>() ;

//整个Linq表达式用圆括号括起来，就将强制转换为正确的实际类型来调用Enumerable的扩展方法

List<int> subsetAsListOfInts = (from I in numbers where I <10 select i).ToList<int>() ;

函数返回Linq查询的结果，不能用Var，（因为Var不能用来定义类和结构的参数，返回值或字段）,所以函数返回可以由IEnumerable<T>类型，或Array来表示

Static IEnumerable<string> GetStringSubset()

{

String[] = colors = {…} ;

IEnumerable<string> theRedColrs = from c in colors where c.Contains(“Red”) select c ;

Return theRedColors ;

}

Static Array GetProjectedSubset()

{…

Return nameDesc.ToArray() ;

}

Static void main()

{

Array objs = GetProjectedSubset() ;

Foreach (object o in objs)

{}

}

除了Linq查询操作符from, in, where, select, join, on, equals, into, orderby, ascending, descending, group, by，还可以用Systems.Linq.Enumerable类提供的扩展方法，如Reverse<>(), ToArray<>(), ToList<>(), Distinct<>(), Union<>(), Intersect<>(), Except<>(), Concat<>()，Count<>(), Sum<>(), Min<>(), Max<>()等

List<string> myCars = new List<String> {…} ;

List<string yourCars = new List<String> {…} ;

Var carDiff = (from c in myCars select c).Except(from c2 in yourCars select c2) ;

投影 新数据类型：从现有的数据源投影出新的数据形式，可以定义一个select语句，动态生成一个新的匿名类型

Var nameDesc = from p in products select new { p.Name, p.Description} ;

C# using关键字可以用于创建类型完全限定名的别名

Using bfHome = System.Rutime.Serialization.Formatters.Binary ;

bfHome.BinaryFormatter b = new bfHome.BinaryFormatter() ; //尽量少用，否则会导致代码库混乱

创建新项目时，会自动默认命名空间名就是项目名，可以在Project -> Properties- >Application里修改

版本号+可选的公钥值 使得一个程序集的不同版本在同一台机器上能够共处而不产生冲突，正式来讲，提供了公钥信息的程序集被称为强命名

私有程序集与调用它的客户端应用程序处于同一个目录下（也可以在其子目录下）。References引用时，会将引用的dll复制到项目文件夹的子目录\bin\debug下,由于.NET运行库在查找被引用的程序集时并不查询系统注册表，因此我们可以把项目可执行文件和引用程序集一起放到机器上的某个位置，然后运行应用程序，这就是所谓的Xcopy部署，卸载（复制）排他性使用私有程序集的应用程序非常容易，只需直接删除（复制）应用程序文件夹就可以了，不需要像使用COM应用程序那样，担心许多遗留在注册表的设置。因此一台机器上很有可能会有同一个私有程序集的副本在不同目录下多次出现的情况。

通过References引用时，则隐式的加载外部程序集，可以在清单的.assembly extern标记来解析程序集位置(显式的加载，需要编程调用System.Reflection.Assembly.Load(“”) ;

\*.dll和\*.exe放在一起，可以直接运行，但如果将\*.dll放在下一层目录里哪MyLibraries，需要使用配置文件，\*.exe.config，是个xml格式的文件，有个privatePath属性修改一下查找地址即可。

共享程序集则是为了一让同一台机器上的大部分应用程序都可以使用它而存在的一种代码库，它被部署在全局程序集缓存(GAC)的特定目录中，

在部署程序集到GAC前，必须赋予它一个强名称，强名称用于标识给定.NET二进制文件的发行者，发行者可以是一个独立程序员，或公司，强名称在.NET中的作用好比全局唯一标识符GUID在COM中的作用。

强名称 =　程序集的友好名称（程序集名称）+版本号[AssemblyVersion]+公钥值[AssemblyKeyFile]+用于本地化的可选区域性标识[AssemblyCulture]+嵌入的数字签名

为程序集赋予强名称方法：properties->Signing->new->myKey.snk

安装强命名程序集：

1. 打开VS2008 Command Prompt，将目录定位到　\*.dll所有目录
2. Gacutil –I \*.dll

共享程序集不同版本可以共存，所在在需要的时候，只要修改应用程序的配置文件\*.config(使用\*.config文件可以忽略清单中记录的版本，并使客户端程序绑定到指定版本的共享程序集)，然后原有客户程序就可以动态定向到某一特定版本（以调用新功能），这一切并不需要重新编译和部署

发行者策略程序集：发行者策略允许程序集的发行者（用户、用户的部门、用户的公司）在安装相关的最新版本程序集到GAC的同时，把一个\*.config文件的二进制版本也安装到GAC。这样做的好处是客户端应用程序目录不需要包含\*.config文件。CLR会读取当前客户端程序的清单，尝试在GAC在查找被清求的版本。但如果CLR找到一个发行者策略程序集，它会读取其中嵌入的XML数据，在GAC级别执行请求重定向。

\*.config,客户端配置文件中，我们还能够增加基于具体应用程序的自定义信息，System.Configuration命名空间提供了一组类型供开发人员读取\*.config文件的自定义数据,这个自定义的配置信息必须放在<appSettings>元素里，<appSettings>元素可包含任意多个<add>元素（用于定义键值对）供开发员以编程方式获取

.NET程序集的格式：

Windows文件首部

Windows文件首部使程序集可以被Windows系列操作加载和操作，这些首部信息标识了应用程序将以什么类型（是基于控制台、基于图形用户界面还是\*.dll代码库）驻留于Windows操作系统中。（大多数程序员不关心）

CLR文件首部

CLR文件首部定义了多个标记，它们使得运行库可以了解到托管文件的布局（大多数程序员不关心）

CIL代码

CIL代码是独立于平台和CPU的中间语言，在运行时，程序集内部的CIL代码才被（实时的JIT编译器）编译成特定平台和CPU的指令。

类型元数据

这些元数据完整地描述了程序集内含类型和引用外部类型的格式。描述程序集包含的每一个类型组成（成员名称、实现的接口、基类、构造函数等），由于程序集的信息被如此详细地记录下来，CLR并不需要访问Windows系统注册表来解析程序集的位置。

程序集清单（manifest程序集元数据）

描述程序集自身各种信息（名称、版本、需要的外部程序集等）的元数据块blob

可选的嵌入资源

如应用程序图标、图像文件、声音片段或者字符串表

类型元数据：完整地描述类型（类、接口、结构、枚举和委托），许多.NET技术如对象序列化,.NET远程处理、XML Web服务以及WCF都需要这个能力在运行时发现类型格式，另外，跨语言互操作、编译器服务以及集成开发环境的智能感知能力都依赖于对类型的具体描述。

使用dldasm.exe并按Ctrl+M打开\*.dll or \*.exe程序集，打开类型元数据，.NET元数据是一组清晰地标记了所有的类型定义（TypeDefs）和被引用类型(TypeRefs)的表。

TypeDef #3

TypeDefName: 类型名

Flags: [Public]

Extends: System.Object

Filed #2 字段

…

Method #1 方法

Property #1 属性

TypeRef #1 引用类型

…

Assembly 描述程序集自身

…

AssemblyRef #2 描述外部程序集

…

User Strings 字符串字面量

反射reflection是一个运行库发现的过程，使用反射服务，可以通过编程使用一个友好的对象模型得到与通过ildasm.exe显示的相同的元数据信息，如类型的列表，包括类型定义的方法、字段、属性和事件。也可以动态发现一组给定类型支持的接口、方法的参数和其他相关细节（基类、命名空间、清单数据等）

SportsCar sc = new SportsCar() ;

Type t = sc.GetType() ; //使用对象得到Type类

Type t = typeof(SportsCar) ; //使用类型得到Type类

Type t = Type.GetType(“CarLibrary.SportsCar”) ; //使用字符串得到Type类

//使用Type类成员获取类型元数据信息

Var methodNames = from n in t.GetMethods() select n.Name ;

Var fieldNames = from n in t.GetFileds() select f.Name ;

Var propNames = from p in t.GetProperties() select p.Name ;

Var ifaces = from I in t.GetInterfaces() select I ;

动态加载程序集：在运行时以编程的方式动态载入程序集，即使那些程序集没有记录在程序清单中。

//动态加载程序集，并获取程序集自身的属性。

Using system.Reflection ;

Assembly asm = Assembly.Load(“CarLibrary”) ;

Type[] types = asm.GetTypes() ;

//动态加载共享程序集

Assembly asm = Assembly.Load(@”CarLibrary, Version=1.0.0.0, PublicKeyToken=null, Culture=”””) ;

晚期绑定：动态类型创建并在运行时调用其成员。而不需要在编译时知道它存在的技术。当建立晚期绑定到外部程序集类型的应用程序时，因为没有设置该程序集的引用，因此调用程序清单没有直接列出这个程序集。(晚期绑定的要点是试图创建编译时未知的对象)

Assembly asm = Assembly.Load(“CarLibrary”) ; //加载程序集

Type minivan = asm.GetType(“CarLibrary.miniVan”) ; //得到类型元数据，

//-------------------------实际上下面这块代码可以用dynamic实现更简单------------------------//

Object obj = Activator.CreateInstance(minivan) ; //运行时创建实例

MethodInfo mi = minivan.GetMethod(“TurboBoost”) ; //得到方法信息

mi.Invoke(obj, null) ; //调用方法,第一个参数是this对象,null表示没有参数

MethodInfo mi = minivan.GetMethod(“TurnOnRadio”) ;

mi.Invoke(obj, new object[] {true, 2}) ; //两个输入参数的方法的调用

//------------------------------------------------------------------------------------------------------------------------//

Dynamic obj = Activator.CreateInstance(minivan) ;

Obj.TurboBoost() ;

Obj.TurnOnRadio(true, 2) ;

.NET平台允许程序员使用特性(attribute)把更多的元数据嵌入到程序集中。特性就是用于类型（比如类、接口、结构等）、成员（比如属性、方法等）、程序集或模块的代码注解。

特性是派生自System.Attribute的类

特性导致嵌入的元数据

直到被其他代理反射、特性才发挥使用

特性在C#中用方括号来应用。

Public sealed class VehicleDescriptionAttribute : System.Attribute

{

Public string Description {get; set;}

}

[serializableAttribute] //serializable

[ObsoleteAttribute(“use another vehicle!”)] //Obsolete

[VehicleDescriptionAttribute(Description=”the old gray mare”)

Public class HorseAndbuggy

{}

//程序集级别特性,强制所有在程序集中的公共类型符合CLS,实现上放置程序集级别使用的特性存于文件AssemblyInfo.cs，也可以通过properties->AssemblyInfo来查看和修改

[assemble : CLSCompliant(true)]

Type t = typeof(HorseAndbuggy) ;

Object[] customAtts = t.GetCustomAttributes(flase) ; //获取horseAndbuggy所有特性(早期绑定)

插件技术：产品必须可通过使用第三方工具进行扩展

当应用程序开发时，各种“钩子”被插入，允许其他软件提供商在IDE开发环境中插入自定义模块。步骤如下：

1. 首先，可扩展的应用程序必须提供一些输入手段，允许用户指定被插入的模块（比如一个对话框或命令行标志）。这需要动态加载
2. 其次，为了插入到环境中，可扩展的应用程序必须要确定模块是否支持正确的功能（比如一组需要的接口）。这需要反射
3. 最后，可扩展的应用程序必须获取一个需要的基础架构的引用（例如接口类型）并调用成员触发底层功能。这经常需要晚期绑定。

如果可扩展的应用程序预编程为查询指定的接口，则它可以在运行时确定类型是否可以被激活，一旦验证测试通过，类型便可以支持额外的接口，为它们的功能提供多种结构。

接口定义

Namespace CommonSnappableTypes

{

//接口为可被可扩展应用程序使用的所有插件提供了一个多态接口，为接近实际例子，想像有这样一组接口，它们可以允许插件产生脚本代码，可以在应用程序工具箱中生成图像或整合到承载程序的主菜单中

Public interface IAppFunctioinality

{

Void DoIt() ;

}

[AttributeUsage(AttributeTargets.Class)] //限制特性使用，仅使用于类中

Public sealed class CompanyInfoAttribute : Syste.Attribute

{

Public string CompanyName {get; set;} ;

Public string CompanyUrl {get; set;} ;

}

}

可扩展应用程序

Private void snapInModule\_Click(object sender, EventArgs e)

{

//允许用户加载一个程序集

OpenFileDialog dlg = new OpenFileDialog() ;

If (dlg.ShowDialog() == DialogResult.OK)

{

Assembly theSnapInAsm = Assembly.LoadFrom(dlg.FileName) ;

//选择程序集中所有的IAppFunctionality接口兼容的类

Var theClassTypes = from t in theSnapInAsm.GetTypes()

Where t.IsClass && (t.GetInterface(“IAppFunctionality”) != null)

Select t ;

//创建对象，并调用方法

Foreach (Type t in theClassType)

{

IAppFunctionality itfApp = (IAppFunctionality)theSnapInAsm.CreateInstance(t.FullName, true) ;

itfApp.DoIt() ;

}

}

//构建插件

Namespace SnapIn

{

[CompanyInfo(CompanyName=”My Company”, CompanyUrl=[www.MyCompany.com](http://www.MyCompany.com))]

Public class Moule : IAppFunctionality

{

Void IAppFunctionality.DoIt()

{…}

}

}

//启动进程

processStartInfo startInfo = ProcessStartInfo(“IExplore.exe”, [www.facebook.com](http://www.facebook.com)) ;

startInfo.WindowStyle = ProcessWindowStyle.Maximized ;

Process ieProc = Process.Start(startInfo) ; // 启动进程

…

ieProcess.Kill() ; //结束进程

文件输入输出和对象序列化

# <Window Form>

Properties

Anchor： 指定在用户重新设置窗口的大小时控件该如何响应

Dock： 指定控件应停靠在容器的边框上。

Event

两部分：

InitializeComponent()里订阅事件 + 事件处理程序

Button：主要用于执行3类任务

1. 用某种状态关闭对话框如OK和Cancel按钮
2. 给对话框上输入的数据执行操作，例如输入一些搜索条件后，单击Search
3. 打开另一个对话框或应用程序如Help按钮

对Button控件的处理，通常是在窗体上添加控件，双击它，给Click事件添加代码，对于大多数应用程序来说足够了

Label and LinkLabel： label在窗体上显示文本，通常不添加任何事件处理代码。LinkLabel，如果希望用户可以单击它，进入文本中显示的网页，就需要添加其他代码。

TextBox: .NET framework内置了两个基本控件来提取用户输入的文本，TextBox and RichTextBox，获取文本框内的字符串，可以用文本框名字.Text; 可以用事件Validating有效性验证文本框不输入无效的字符或只能输入某个范围内的数值。事件KeyPress验证输入的字符是否在一定范围内

RadioButton and CheckBox: RadioButton常放入GroupBox里，作为一个逻辑单元，RadioButton按钮会自动改变自己的状态，以使组框中只选中一个选项。CheckBox一般用于希望用户可以选择一个或多个选项，二者均可以用Checked属性获得哪个选项或将Checked设置为true表示选中

ListBox and CheckedListBox: 列表框ListBox用于显示一组字符串，一般用于显示多个选项。CheckedListBox派生于ListBox，一般用于选择多个选项

ListBox.Items包含列表框中的所有选项集合，使用这个集合的属性可以增加或删除选项。

CheckedListBox.CheckedItems包含选中的选项集合，CheckedListBox.CheckOnClick若为true，则选项就会在用户单击它时改变它的状态。

ListView列表视图通常用于显示数据，用户可以对这些数据和显示方式进行某些控制，还可以把包含在控件中的数据显示为列和行（像网格那样），或者显示为一列，或者显示为图标表示。最常用的列表视图就是用于导航计算机中文件夹的视图。

ListView.Items.Add(ListViewItems)，列表视图中的选项ListViewItem包含要显示的信息如文本和图标的索引ListViewItem.SubItems.Add(ListViewSubItem)，ListViewSubItem表示列表视图中的一列。

ListView.Columns.Add(ColumnHeader)，列表视图列标题ColumnHeader，在ListView控件处于Details模式下时，为要显示的列提供一个标题。

在窗体Form中，通过ImageList控件存储其他控件（在窗体中）使用的图像。在设计期间，点击Form1,从工具箱里选择ImageList,拖拉Form中，给ImageList命名，同时点击ImageList的Images属性，插入图像。窗体中的其他控件想使用ImageList，是通过ImageList的索引。

为了允许用户通过双击ListView中的选项来浏览文件夹，需要订阅ItemActivate事件。ListView.View可以设置为View.LargeIcon, View.SmallIcon, View.List, View.Tile

TabControl控件，可以把对话框组织为合乎逻辑的部分，以便根据控件顶部的选项卡来访问，TabControl包含TabPages，工作方式与GroupBox类似。可以在TabPages属性中添加任意数据量的选项，再把要显示的控件拖放到各个页面上。

TabControl的属性一般用于控制TabPage对象窗口的外观，特别是显示的选项卡的外观。

# <WPF 和XAML>

传统图形化桌面应用程序 WPF

构建带控件的表单 Windows forms System.Windows.Forms.dll WPF

2D 图形支持 GDI+ System.Drawing.dll WPF

3D图形支持 DirectX API WPF

对流视频的支持 Windows Media Player API WPF

对流文档的支持 编程操作PDF文件 WPF

WPF优点：

将许多之前无关的桌面技术（2D图形，3D图形，窗口和控件开发等）整合到一个统一的编程模型中。

代码文件方式，将Windows应用程序的界面外观与驱动它们的编程逻辑清晰地相互分离。使用XAML，可以利用标记来定义一个应用程序的UI，即不仅可以定义简单的UI元素（按钮，网络，列表框等），而且还可以定义二维和三维图形数据，动画，数据绑定逻辑以及多媒体功能（如视频回放）。WPI在呈现图形数据时不使用GDI，所有呈现操作如2D，3D图形，动画，UI组件呈现的按钮和下拉列表框都使用DirectX API。WPF应用程序还能使用丰富的图形服务（模糊效果、抗锯齿、透明度等），并且避免了直接使用DirectX API进行编程所带来的复杂性。

一般来说，XAML文件包含描述窗口外观的标记，而C#代码文件包含实现逻辑，例如，Window的XAML文件会描述整个布局系统及其内部的控件，指定不同事件处理程序的名称。而相关的C#文件则包含这些事件处理程序的实现逻辑和应用程序所需的自定义代码

WPF程序集

PresentationCore.dll WPF GUI层基础的类型

PresentationFramework.dll WPF控件，Application,Window类以及对交互的二维几何图形的支持，及在运行时读写XAML文档

System.Xaml.dll 运行时对XAML文档进行编程

WindowsBase.dll WPF API 基础结构的核心类型，如WPF线程类型，安全类型，各种类型转换器以及对领带属性和路由事件的支持

WPF核心命名空间

System.Windows WPF根命名空间，所有WPF桌面项目所需要的核心类如Application and Window

System.Windows.Controls 构建菜单系统，工具帮助以及众多布局管理器的多种类型

System.Windows.Media 以媒体为主的命名空间的根空间，包括动画，三维显示，文本显示以及其他多媒体用途的类型

System.Windows.Shapes 响应鼠标输入的交互式二维图形

WPF = System.Windows .Window+ System.Windows .Application(全局应用程序对象)

Application.Current ; //全局应用程序对象（同MFC）

//访问 Application对象的线程所创建的每个窗口

WindowCollection winCol = Application.Current.Windows ;

//建立和获取整个WPF应用程序中可以访问的数据（如窗口，对话框等）

Application.Current.Properties[“GodMode”] = false ; //采用名称值对形式，在某回调函数

If ( (bool) Application.Current.Properties[“GodMode”]) //在别的回调函数里使用

System.Windows.Controls.ContentControl 内容控件

提供承载内容，就是一组通过Content属性放在控件表面的对象集合，一个内容控件能够容纳非常多的UI元素，而不只限于简单的字符串数据

System.Windows.UIElement

提供一组可以允许其子类获取焦点并处理输入请求的事件。比如负责拖放，鼠标移动，键盘输入等操作事件

System.Windows.Controls.Control控件

提供多种属性，设置控件大小，透明度，焦点切换顺序逻辑，光标显示，背景颜色等。

System.Windows.FrameworkElement

支持情节提要（在动画中使用），数据绑定以及命名成员（通过Name属性），获取派生类型定义的资源，创建派生类型的整体的大小。

//在\*.xaml文件中建立的标记会被转化为一个直接映射到相关的.NET命名空间中的一个完整对象模型，XAML文件中的每个元素都代表某个.NET命名空间中的一个类型名称如Button,Window,Application，一个起始元素范围内的特性将映射到这个特定类型的特性如Height,Width等和事件Startup,Click等

XAML可以描述任何程序集中的任何类型，只要它是一个非抽象类型，并且包含默认构造函数

<Window x:Class=”Ex0.MainWindow”

Xmlns=<http://schmas.microsoft.com/winfx/2006/xaml/presentation>

Xmlns=<http://schemas.microsoft.com/winfx/2006/xaml>

Xmlns:CorLib=”clr-namespace:System; assembly=”mscorlib”//引用外部类型

Title=”demo” Height=”256” Width=”512” WindowStartupLocation=”CenterScreen”>

//使用复杂数据隐式设置Content属性，WPF控件的内容模型只能设置一次Content属性，所以几乎在所有情况下，分配给Content属性的“单一内容”实际上是一个布局管理器，如DockPanel,Grid,Cavas or StackPanel。在布局管理器内，可以对控件进行任意组合，甚至可以内嵌另一个布局管理器

<StackPanel>

<Label Content =”{x:Static CorLib:Environment.OSVersion}”/> //字符串

<ListBox Width=”200” Height=”50”>

<ListBox.ItemsSource> //属性元素语法

<x:Array Type=”CorLib:String”> //标记扩展

<CorLib:String>Sun Kil</CorLib:String>

<CorLib:String>Sun Kil</CorLib:String>

</x:Array>

</ListBox.ItemsSource>

</ListBox>

</StackPanel>

</Window>

<Application x:Class=”Ex0.MyApp”

Xmlns=<http://schmas.microsoft.com/winfx/2006/xaml/presentation>

Xmlns=<http://schemas.microsoft.com/winfx/2006/xaml>

StartupUri=”MainWindow.xaml”

Exit=”AppExit”>

</Application>

Xmlns:x 建立一个XML命名空间声明

X:Class 使用命名空间x的class特性指定C#类的名称

FAQ of C#

1. How to extend method for existing class?

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

// binary tree

namespace BinaryTree

{

public class Tree<TItem> where TItem : IComparable<TItem>

{

public TItem NodeData { get; set; }

public Tree<TItem> LeftTree { get; set; }

public Tree<TItem> RightTree { get; set; }

public Tree(TItem nodeValue)

{

this.NodeData = nodeValue;

this.LeftTree = null;

this.RightTree = null;

}

public void Insert(TItem newItem)

{

TItem currentNodeValue = this.NodeData;

if (currentNodeValue.CompareTo(newItem) > 0)

{

if (this.LeftTree == null)

{

this.LeftTree = new Tree<TItem>(newItem);

}

else

{

this.LeftTree.Insert(newItem);

}

}

else

{

if (this.RightTree == null)

{

this.RightTree = new Tree<TItem>(newItem) ;

}

else

{

this.RightTree.Insert(newItem);

}

}

}

public void WalkTree()

{

if (this.LeftTree != null)

{

this.LeftTree.WalkTree();

}

Console.WriteLine(this.NodeData.ToString());

if (this.RightTree != null)

{

this.RightTree.WalkTree();

}

}

}

}

// add extend method to binary tree

using BinaryTree;

static class ExtendMethod4Tree

{

public static void InsertIntoTree<TItem>(this Tree<TItem> tree, params TItem[] data) where TItem : IComparable<TItem>

{

if (data.Length == 0)

{

throw new ArgumentException("Must provide at least one data value");

}

foreach (TItem datum in data)

{

tree.Insert(datum);

}

}

}

// application

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World");

Tree<int> tree = new Tree<int>(10);

tree.Insert(5);

tree.Insert(11);

tree.WalkTree();

Console.WriteLine();

tree.InsertIntoTree('X', 'A', 'M', 'Z', 'Z', 'N');

tree.WalkTree();

Console.ReadKey();

}

}

How to understand covariance and contravariance?

// out is to denote T can only be as return

interface IRetrieveWrapper<out T>

{

T GetData() ;

}

class Wrapper<T> : IRetrieveWrapper<T>

{

private T storedData ;

T IRetrieveWrapper<T>.GetData()

{

return this.storedData ;

}

public Wrapper(T data)

{

this.storedData = data ;

}

static public void test()

{

Wrapper<string> stringWrapper = new Wrapper<string>("Hello") ;

IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper ;

Console.WriteLine("stored value is {0}", retrievedStringWrapper.GetData()) ;

//Covariance preserves [assignment compatibility](http://blogs.msdn.com/ericlippert/archive/2009/11/30/what-s-the-difference-between-covariance-and-assignment-compatibility.aspx)

// derived type is assgined to based type as generic type argument

IRetrieveWrapper<object> retrievedOjbectWrapper = stringWrapper ;

Console.WriteLine("stored value is {0}", retrievedOjbectWrapper.GetData()) ;

}

}

// in is to denote T can only be based type or derived type

interface IComparer<in T>

{

int Compare(T x, T y);

}

class ObjectComparer : IComparer<Object>

{

int IComparer<object>.Compare(Object x, Object y)

{

int xHash = x.GetHashCode();

int yHash = y.GetHashCode();

if (xHash == yHash) return 0;

if (xHash < yHash) return -1;

return 1;

}

static public void test()

{

ObjectComparer objectComparer = new ObjectComparer();

IComparer<Object> objectComparator = objectComparer;

Console.WriteLine("result is {0}", objectComparator.Compare(new Object(), new Object()));

// based type is is assgined to derived type as generic type argument

IComparer<String> stringComparator = objectComparer;

Console.WriteLine("result is {0}", stringComparator.Compare("qzlin", "lin qizhong"));

}

}

How to implement Enumerator?

Commonly, the programmer simply use iterator (yield) to implement class with the function of Ieumerable, then this class could be traversed via foreach and LINQ

class EnumeratorTest<T> : IEnumerable<T>

{

private List<T> data = new List<T>();

public void FillList(params T[] items)

{

foreach (T datum in items)

{

data.Add(datum);

}

}

IEnumerator<T> IEnumerable<T>.GetEnumerator()

{

foreach (T datum in data)

{

yield return datum;

}

}

public IEnumerable<T> Reverse

{

get

{

for (int i = data.Count - 1; i >= 0; --i)

{

yield return data[i];

}

}

}

public static void test()

{

EnumeratorTest<string> bc = new EnumeratorTest<string>();

bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc)

{

Console.WriteLine(word);

}

foreach (string word in bc.Reverse)

{

Console.WriteLine(word);

}

}

}

How to override operator?

class Complex

{

public float Real { get; set; }

public float Imaginary { get; set; }

public Complex(float real, float imag)

{

this.Real = real;

this.Imaginary = imag;

}

// different to C++, both left hand side and right hand side are required for binary operator

// override operator require public + static

public static Complex operator +(Complex lhs, Complex rhs)

{

return new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);

}

// implicit and explicit conversion

public static implicit operator Complex(float from)

{

return new Complex(from, 0.0f);

}

public static explicit operator float(Complex from)

{

return from.Real;

}

public static void test()

{

Complex A = new Complex(1, 1);

Complex Z = A + 1.0f;

float real = (float)Z;

}

}

Static class, anonymous class?

Static class as container for tool method, such as Math.Sin(), Math.Sqrt()

Anonymous class only contains public field, all fields must be initialized, not be static and no include method

public static class Math

{

public static double Sin(double x) { }

public static double Cos(double x) { }

...

}

//anonymous class, for LINQ

var myAnonymousObject = new {Name="Zhou Jing", Age=44};

the difference of array parameter and parameter array?

static class Util

{

public static int Min1(int[] paramList)

{

if (paramList == null || paramList.Length == 0)

{

throw new ArgumentException("not enough parameter");

}

int currentMin = paramList[0];

foreach (int i in paramList)

{

if (i < currentMin) currentMin = i;

}

return currentMin;

}

public static int Min2(params int[] paramList)

{

// ditto

}

public static void test()

{

// array params require array initialized manually

int[] array = new int[2];

array[0] = 1;

array[1] = 2;

int min = Util.Min1(array);

// params array save code line

int m = Util.Min2(1, 2);

}

}

Extension method?

Allow to add extra static method to extend exsiting type

static class Util

{

// this is to denote extension method for int type

public static int Negate(this int i)

{

return -i;

}

// if class Util is in the same scope, then extension method could be called directly

public static void test()

{

int x = 591;

int y = x.Negate();

}

}

Delegate is like function declaration, but is type-safe

//other tools supply each API

// code like following is hard code,

// Controller type is closely connected to external API

// external API is inside class Controller

class Contoller

{

private FoldingMachine folder = new FoldingMachine();

private WeldingMachine welder = new WeldingMachine();

private PaintingMachine painter = new PaintingMachine();

public void ShutDown()

{

folder.StopFolding();

welder.FinishWelding();

painter.PaintOff();

}

}

// use delegate to disconnect

// external API is outside class Controller

class Controller

{

public delegate void stopMachineryDelegate(); // like function declaration

private stopMachineryDelegate stopMachinery;

public void Add(stopMachineryDelegate stopMehtod)

{

this.stopMachinery += stopMethod;

}

public void Remove(stopMachineryDelegate stopMethod)

{

this.stopMachinery -= stopMethod;

}

}

static class TestClass

{

public static void test()

{

// hard code, closely connect

Controller controller = new Controller();

controller.ShutDown();

// disconnect

Controller controller = new Controller();

FoldingMachine folder = new FoldingMachine();

WeldingMachine welder = new WeldingMachine();

PaintingMachine painter = new PaintingMachine();

controller.Add(folder.StopFolding);

controller.Add(welder.FinishWelding);

controller.Add(painter.PaintOff);

controller.ShutDown();

}

}

Lambda includes para list and method body (expression block)

() => {Math.Sin(0); Math.Cos(0);}

(object sender, CarEventArgs e) =>

{

if (sender is Car)

{

Car c = (Car) sender ;

fun(c, e) ;

}

}

//compiler transfer lambda method into anomounous

delegate(object sender, CarEventArgs e)

{

if (sender is Car)

{

Car c = (Car) sender ;

fun(c, e) ;

}

}

Event mechenism

Event type must be delegate

// event source class includes event declaration and event generation

class TemperatureMonitor

{

// event declaration

public delegate void StopMachineryDelegate();

public event StopMachineryDelegate MachineOverheating;

// event generation

private void Notify()

{

if (this.MachineOverheating != null)

{

this.MachineOverheating();

}

}

}

class TestClass

{

static void main()

{

TemperatureMonitor temp = new TemperatureMonitor();

// event subscriber

temp.MachineOverheating += () => StopFolding(0);

temp.MachineOverheating += FinishWelding;

}

void StopFolding(int x) { }

void FinishWelding() { }

}

//WPF GUI Event

// hide code

// event declaration

public delegate void RoutedEventHander(object sender, RoutedEventArgs e) ;

public class ButtonBase : ...

{

public event RoutedEventHander Click ;

...

}

public class Button : ButtonBase

{

}

// event generation

public partial class Example : Window, IComponentConnector

{

internal Button okay ;

void Connect()

{

this.okay.Click += new RoutedEventHander(this.okayClick) ;

}

}

// user code

// event subscriber

public partial class Example : Window

{

private void okayClick(object sender, RoutedEventArgs args)

{

// Click event implement

}

}