C# 学习笔记

1 Attribute 特性

2 生成器

3 Action

4 Lambda

5 NuGet 依赖包管理工具

6 单元测试框架 XUnit

SimpleUnitDemo.Services

SimpleUnitDemo.Tests

7 扩展方法

静态方法 参数 this 类型名 参数名

8 结构和类的区别

结构不支持实现继承

结构是值类型 继承自 System.ValueType

类派生自System.Object

9 ref 关键字

函数参数传递分为值传递和引用传递

引用类型通过引用传递，值类型通过值传递，ref强制值类型通过引用传递

10 virtual override abstract

抽象方法：纯虚方法

虚方法：virtual

重写方法：override

11 泛型

null 只能用于引用类型 0 用于值类型

default 关键字

T doc = default(T) 如果T是值类型，就初始化为0，如果是引用类型，就初始化为null。

泛型约束：

where T:class 必须是引用类型

where T:struct 必须是值类型

where T:new() T 必须有一个默认构造函数

where T:IFoo 必须实现接口IFoo

where T:Foo 必须派生自基类Foo

where T1:T2

IComparable

int CompareTo(object obj)

12 枚举

foreach

示例代码

foreach(var p in persons)

{

Console.WriteLine(p);

}

编译器翻译：

IEnumerator<Person> enumator = persons.GetEnumerator();

while (enumator.MoveNext())

{

Person p = enumator.Current;

Console.WriteLine(p);

}

yield语句 包含yield语句的方法是一个迭代块,迭代块必须声明为返回IEnumator或其接口，不能包含return 语句，可以包含yield return 语句

# 13 委托

C++ 函数指针，委托是类型安全的类，定义了返回类型和参数的类型

匿名方法：

Func<string,string> a = delegate(string param)

{

return “result”;

}

lambda 表达式

参数=>方法

de

# 14 params 可变参数

# 15 GenericArguments

GenericParameters

# 16 Attribute

特性（属性）

属性类派生自System.Attribute 至少有一个公共构造函数

属性类有两种类型的参数：

定位参数：必须指定的参数，属性类的构造函数参数

命名参数：可选，使用属性时必须使用参数的名字。通过包含非静态字段或属性来定义命名参数

例如 AttributeUsage

有一个定位参数AllowOn 指定可以将属性赋值给的程序元素（类，方法，属性，参数）

System.Attributes.AttributeTargets

AttributeTargets.Class

AllowMultiple 一个bool值，是否允许为一个程序元素指定多个属性。默认值false

Inherited: 我们可以使用这个属性来控制定制特性的继承规则。它标记了我们的特性能否被继承。

# 17 集合类字典 枚举

所有集合类都实现了IEnumerable和IEnumerator接口

可枚举接口

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

枚举器接口

public interface IEnumerator

{

object Current{get;}

bool MoveNext();

void Reset();

}

实现了IEnmerable<T>接口的集合，是强类型的。它为子对象的迭代提供类型更加安全的方式。

显式实现接口

显示实现继承非泛型IEnumerable接口的方法

IEnumerator IEnumerable.GetEnumerator()

{

}

关键字yield

# 18 属性

**自动属性3.0**

public class Test

{

public string Name

{

get;

set;

}

}

# 19 Object 比较

Equals

GetHashCode

实现了Equals必须实现GetHashCode

字典

键类型

如果两个对象的 Equals 比较结果相等，则每个对象的 GetHashCode 方法都必须返回同一个值。 但是，如果两个对象的比较结果不相等，则这两个对象的 GetHashCode 方法不一定返回不同的值。

GetHashCode()实现要求：

* 相同的对象应该总是返回相同的值。
* 不同的对象可以返回相同的值。

如果Equals 比较两个对象相等，则GetHashCode必须相等

如果不满足上述条件：某个类的实例放入字典后可能再也检索不到，或者会返回错误的项

A B 是相等的对象，但是HashCode不一致，经过散列后在一个桶里，这个时候可能A存在B的后面，那么性检索A，会得到B的值。

# 20 可访问性 accessibility

The protected internal accessibility means protected OR internal, not protected AND internal.

class Test

{

protected internal string name;

}

UNITY3D 脚本工程文件

* （1）Assembly-CSharp-firstpass

所有在Standard Assets，Pro Standard Assets或者 Plugins文件夹中的脚本会产生一个Assembly-CSharp-firstpass-vs.csproj文件，并且先编译

* （2）Assembly-CSharp-Editor-firstpass

所有在Standard Assets/Editor, Pro Standard Assets/Editor 或这Plugins/Editor文件夹中的脚本产生Assembly-CSharp-Editor-firstpass-vs.csproj工程，接着编译；

* （3）Assembly-CSharp

所有在Assets/Editor外面的, 并且不在(1),(2)中的脚本文件（一般这些脚本就是我们自己写的非编辑器扩展的脚本）会产生Assembly-CSharp-vs.csproj工程，被编译；

* （4）Assembly-CSharp-Editor

所以在Assets/Editor中的脚本产生一个Assembly-CSharp-Editor-vs.csproj工程，被编译。

之所有这样建立工程并按此顺序编译，也是因为DLL间存在的依赖关系所决定的。

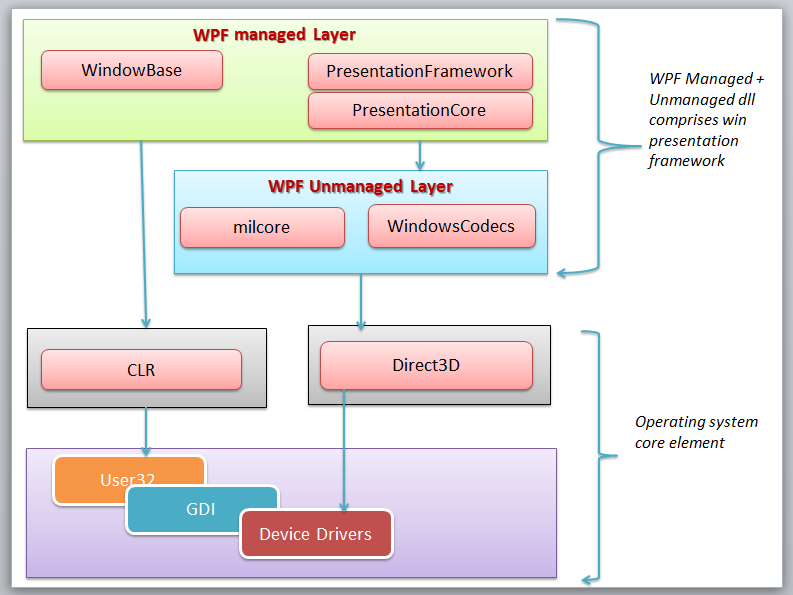
# Uconsole

# 第二章：WPF

DPI: dot per inch 每英寸的像素数，也叫屏幕密度 160DPI

DIP(DP) Density-independent pixel 与屏幕密度有关

Density-independent pixel (dp)   
A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent way.   
The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen. At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use. The conversion of dp units to screen pixels is simple: px = dp \* (dpi / 160). For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels. You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.



**ExpressionBuilder** 是最好的生成XAML的工具。

为了区分它们， WPF 使用 Preview\*\*\* 这样的风格命名隧道事件，使用只有 \*\*\* 来命名冒泡事件。举个例子: IsPreviewMouseDown 事件通过隧道传播整个可视化树，而 MouseDown 事件使用冒泡的方式。意思就是: 发生 Mounse Down 事件时，监听 IsPreviewMouseDown 事件的，最外层的元素最先被调用，监听 MouseDown 事件的，最里面的元素最先被调用。

WPF窗口继承自ContentControl

**ContentControl**：一个ContentControl 可以包含单个的子内容。窗口继承自ContentControl ，所以每个窗口都只能包含一个子元素。比如：Windows， Button等。

**HeaderedContentControl**：它基本上和 ContentControl 相同，不过它包含了一个多出来的内容的标题。例如： GroupBox， Expander 都是 HeaderedContentControl 。

**ItemsControl**：ItemsControl 可以包含多个内容。因此，你可以在 ItemsControl 中放入很多任意的元素。例如： ListBox , ListView 。

**HeaderedItemsControl**：每个 Collection 都包含了一个特别的标题内容。HeaderedItemsControl 是一个包含了每个都有单独标题的元素的复杂元素。 TreeView 就是一个 HeaderedItemsControl 。

* **Auto**：大小通过放入的元素来决定，这是默认的。
* **\*(星号)**：如果使用\*，意思就是将会使用比例来决定大小。2\*表示是1\*的两倍。所以如果你像创建宽度为2:1的两列，你应该指定宽度为 2\*和1\*。
* **Absolute**：你可以定义宽高的绝对大小。意思就是如果你把高度定义为100，它将相应地使用它。

**MVVM**

The interesting thing to me about MVVM (Model-View-ViewModel), is that the Model should know about nothing else, the ViewModel should only know about the Model (not the View), and the View should only know about the ViewModel (and not the Model).

**Model**: This can be really simple, the goal here is for the ViewModel not to have to do any of the business logic.

**ViewModel**: This should essentially delegate everything to the Model except for exposing data for the View.

**View**: This should just bind to the ViewModel and make stuff look pretty.

Data Binding

WPF Control DataContext ItemsSource

INotifyPropertyChanged Interface

ObserableCollection<T>