目录

[书写规则 1](#_Toc41054657)

[C#基本语法 2](#_Toc41054658)

[0.Preprocessor Directives & Comments 2](#_Toc41054659)

[1. Varaibles & Constants 2](#_Toc41054660)

[2. Conditional Statements, Switch Statements 4](#_Toc41054661)

[3. Loops 4](#_Toc41054662)

[4. Type Convertion/Casting 4](#_Toc41054663)

[5. Complex Variable Type 5](#_Toc41054664)

[6. Functions 6](#_Toc41054665)

[7. Delegate 7](#_Toc41054666)

[8.调试和错误处理 8](#_Toc41054667)

[C#OOP技术笔记 9](#_Toc41054668)

[一、基本概念 9](#_Toc41054669)

[二、定义类/接口 9](#_Toc41054670)

[三、定义类成员/接口成员 10](#_Toc41054671)

[四、集合，比较和转换 13](#_Toc41054672)

[五、泛型 15](#_Toc41054673)

[六、事件 16](#_Toc41054674)

[七、特性[attribute]与反射 17](#_Toc41054675)

[C#高级用法和话题 18](#_Toc41054676)

[lambda expression用法 18](#_Toc41054677)

[初始化器 18](#_Toc41054678)

[类型推理和匿名类型 19](#_Toc41054679)

[动态查找和动态类型/变量 19](#_Toc41054680)

[高级方法参数 19](#_Toc41054681)

[C#额外知识积累 20](#_Toc41054682)

[@符号的用法 20](#_Toc41054683)

[await和async与异步编程 20](#_Toc41054684)

# 书写规则

1.橙色代表不确定的个人推论

2.红色代表可能在新标准中更改的内容

3.绿色代表批注与评论

4.蓝色代表关键词keywords，指令中其他首字母大写的是具体编辑内容，如

const DataType DataName

5.绛色代表代码

# C#基本语法

## 0.Preprocessor Directives & Comments

①折叠代码区域

#region XXX

//XXX为代码块名称

#endregion

②导入外部资源或名称空间

使用using关键字，用法类似Java的import

③注释

类似C++

•对于comment blocks仍然沿用//或/\*…\*/的注释方式

## 1. Varaibles & Constants

类型 变量名称=初值

const 类型 变量名=初值

•与C++相同

•类型：int, string（UnityScript中是String）, void, bool

**①整型类型表**

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 别名 | 值域 | 后缀 |
| sbyte | System.Sbyte | -128~127 |  |
| byte | System.Byte | 0~255 |  |
| short | System.Int16 | -32768~32767 |  |
| ushort | System.UInt16 | 0~65535 |  |
| int | System.Int32 | -2147483648~2147483647 |  |
| uint | System.UInt32 | 0~4294967295 | u, U |
| long | System.Int64 | -9223372036854775808~9223372036854775807 | l, L |
| ulong | System.UInt64 | 0~18446744074709551615 | u, U, l, L, ul, uL, Ul, UL, lu, lU, Lu, LU |
| char | System.Char | Unicode, 0~65535 |  |
| bool | System.Boolean |  |  |
| string | System.String |  |  |

**②浮点数类型**

float/double形式：+/- m×2e

decimal形式：+/- m×10e

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类型 | 别名 | m  min | m  max | e  min | e  max | 近似最小 | 近似最大 | 后缀 |
| float | System.Single | 0 | 224 | -149 | 104 | 1.5×10-45 | 3.4×1038 | f, F |
| double | System.Double | 0 | 253 | -1075 | 970 | 5.0×10-324 | 1.7×10308 | d, D |
| decimal | System.Decimal | 0 | 296 | -26 | 0 | 1.0×10-28 | 7.9×1028 | m, M |

**③转义字符**

|  |  |  |
| --- | --- | --- |
| 转义序列 | 产生字符 | 字符的Unicode值 |
| \a | Sounds a beep | 0x0007 |
| \n | Newline | 0x000A |
| \t | Tab | 0x0009 |
| \b | Backspace | 0x0008 |
| \r | return to the start point of the current line | 0x000D |
| \f | move to the start point of next page | 0x000C |
| \v | jump vertically | 0x000B |
| \\ | Backslash, = “\” | 0x005C |
| \’ | Single quote | 0x0027 |
| \” | Double quote | 0x0022 |
| \0 or \000 | empty character(NULL) | 0x0000 |
| \? | Question mark |  |

**④数学运算符/赋值运算符**

|  |  |  |
| --- | --- | --- |
| 运算符 | 类别 | 作用 |
| + | 二元 | 求和，对于字符串起到连接作用 |
| - | 二元 | 其他运算符对于字符串无效 |
| \* | 二元 |  |
| / | 二元 |  |
| % | 二元 | 求余数 |
| + | 一元 | 正号 |
| - | 一元 | 取负 |
| ++, -- | 一元 | 自增自减，同C++ |
| = | 二元 | 同C++ |
| += | 二元 | 同C++ |
| -= | 二元 | 同C++ |
| \*= | 二元 | 同C++ |
| /= | 二元 | 同C++ |
| %= | 二元 | 同C++ |

**⑤布尔比较运算符/关系运算符**

|  |  |  |
| --- | --- | --- |
| 运算符 | 类别 | 作用 |
| ==, != | 二元 |  |
| <, <= | 二元 |  |
| >, >= | 二元 |  |
| ! | 一元 | 逻辑非 |
| &&, || | 二元 | 逻辑与，或 |
| &=, |=, ^= | 二元 | 布尔赋值运算 |

**⑥位操作运算符**

|  |  |  |
| --- | --- | --- |
| &, |, ^,~ | 二元 | 按位与，或，异或，非 |
| >>, << | 二元 | var1=var2<<var3, 将var2的二进制值，左移var3位，赋给var1 |
| >>=, <<= | 二元 | 位操作赋值运算 |

**⑦运算符优先级**

|  |
| --- |
| ++, --(前缀), +, -(一元), (), !, ~ |
| \*, /, % |
| +, -(二元) |
| <<, >> |
| <, >, <=, >= |
| ==, != |
| & |
| ^ |
| | |
| && |
| || |
| ? X:Y |
| =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, != |
| ++, --（后缀） |

## 2. Conditional Statements, Switch Statements

if(…)…

else…

•条件语句与逻辑运算符的使用与C++一致

switch(X)

{

case Y1:

…

break;

case Y2:

…

break;

default:

…

break;

}

•switch语句如果无break，C#诊断为非法，但C++中则可利用此执行多个case语句

X ? Y:Z

•三元运算符同C++

## 3. Loops

•while、do…while和for（传统c98式）循环用法与C++一致

•循环中断语句

1) break

2) continue

3) goto

goto语句是C#中添加的指令，类似FORTRAN，为指令添加标签label作为位置标记，之后用goto转移；但是这种结构不利于代码清晰和阅读

4) return

## 4. Type Convertion/Casting

(类型) 变量

•C#采用C的转换类型格式

checked((类型) 变量)

unchecked((类型) 变量)

•checked对表达式进行溢出检查，如果溢出将报错；unchecked使编译器对表达式不进 行溢出检查，溢出串位的情况不报错；可以通过对IDE设置使编译软件自行添加溢出 检查

•System.Convert执行显示转换

|  |  |
| --- | --- |
| 命令 | 结果 |
| Convert.ToBoolean(X) | bool |
| Convert.ToByte(X) | byte |
| Convert.ToChar(X) | char |
| Convert.ToDecimal(X) | decimal |
| Convert.ToDouble(X) | double |
| Convert.ToInt16(X) | short |
| Convert.ToInt32(X) | int |
| Convert.ToInt64(X) | long |
| Convert.ToSByte(X) | sbyte |
| Convert.ToSingle(X) | float |
| Convert.ToString(X) | string |
| Convert.ToUInt16(X) | ushort |
| Convert.ToUInt32(X) | uint |
| Convert.ToUInt64(X) | ulong |

注：1）这些指令总会进行溢出检查；

2）将字符串转换为数值必须采用标准格式，例：-1.2451e-24，+0.1415E+23，最后有一 空格

•字符串与数字相加时自动转换数字为字符

e.g. “Hello”+1 结果为 “Hello1”

C++中实现这个结果需要借助stringstream等

## 5. Complex Variable Type

*5.1 Enumeration*

enum 类型名:基本类型

{枚举值1=数值1, 枚举值2…}

•基本类型是枚举变量的具体数值类型，默认为int，可以设为byte, sbyte, short, ushort, uint, long, ulong

•可以选择性的为枚举值手动赋值，默认从0开始，依次+1，道理同C++

•枚举值和各种整型的转换必须显式进行，反之亦然，即便是与基本类型之间转换

变量名=类型名.枚举值

•为枚举变量赋值，必须加类型名和成员符号

字符串变量=Convert.ToString(枚举变量)

•将枚举值转换为字符串不能使用(string)强制类型，因为后者只能将枚举值对应的基本类型数值转化为字符串而不是枚举值

枚举变量=(枚举类型)Enum.Parse(typeof(枚举类型), 字符串变量)

•将字符串转换为枚举值按照以上方法，注意字符串书写必须完全符合枚举值

*5.2 Struct*

struct 结构体名

{

访问属性 类型 变量名

}

*5.3 Arrays*

类型 [] 数组名

类型 [,] 数组名

类型 [][] 数组名

•声明数组的语法与C++不一样，声明时不能在中括号内指定维数，只能在初始化后被 自动确定；对于一般的数组，一旦初始化后维度也就确定了。

•第二种是传统多维数组的定义方式，也是矩形数组，第三种是锯齿形数组，代替了原 有的C++传统多维数组

类型 [] 数组名 = {….}

类型 [] 数组名 = new 类型[元素个数]{….}

类型 [,] 数组名 = new 类型[元素维度,维度]

•使用后一种方式初始化时，大括号内的元素数量必须与new后面中括号内的数字相同

•初始化可以和声明语句分开，使用new可以不指定具体元素取值，取默认值

•这两种一般的初始化方法只适合一维或多维数组，对于锯齿形数组的初始化需要特殊 处理（p100/119）

foreach(类型 变量名 in 数组名){…}

•这种遍历方法类似c++11中的for循环，用一个相同类型的变量遍历每个数组元素

•这种遍历方法是只读的，因此如果要改变元素的值仍需使用for循环或其他方式

## 6. Functions

*6.1 参数数组*

返回类型 函数名(参数类型1 参数名1, 参数类型2 参数名2,…, params 类型[] 数组名)

例：int SumVals(params int[] vals)

函数名(参数1, 参数2,…, 数组元素值1, 数组元素值2,…)

•C#独有添加的参数数组，是一个特殊的参数，每个函数最多只能有一个参数数组，且 必须在参数列表的最后；定义时，参数数组按照普通数组参数处理，但调用时，需按照 元素一一传值，且不限个数只限类型。

*6.2 参数引用*

C#中的引用使用方法和C++不一样，使用ref关键字来作为引用传递参数给函数。

ref的特点是有进有出，即在传递参数之前就已经对它进行赋值，在传入方法体时，是将该数的地址传了进来，如果对其进行相应的赋值操作，直接改的是地址里的值，所以，当该方法执行完，该数的值也就跟着改变了。

代码如下：

static void Main(string[] args)

{

int i = 10;

FF(ref i); //调用

Console.WriteLine(i);

}

static void FF(ref int i) //定义

{

i = i + 1;

}

*6.3 输出参数*

而out与ref的唯一区别是，在方法接收参数后，对它进行初始化（如果没有初始化，将会报错的），其余的用法都和ref一样。当然也可以传送已经初始化过的参数给out形参，但是这个初始化值在函数调用后势必会改变。

代码如下：

static void Main(string[] args)

{

int i ;

FF(out i);

Console.WriteLine(i);

}

static void FF(out int i)

{

i=10; //如果没有这个赋值，系统将会报错的

i = i + 1;

}

*6.4 表达式体方法expression-bodied method*

执行仅一行代码的的函数可使用lambda函数来定义：

static double Multiply (double myVal1, double myVal2)

{

return myVal1 \* myVal2;

}

可写成

static double Multiply (double myVal1, double myVal2) => myVal1 \* myVal2;

方法可以无返回类型void，如果有则编译器会根据函数体内容自行推导返回值。

## 7. Delegate

委托的功能简单来说类似于函数指针，但是其真正用途在于事件和事件处理机制中的动态函数类型绑定。所以学习委托的使用和事件机制分不开。

·委托实际上也是一个类。

①委托的声明：

delegate 返回类型 委托名(参数列表);

·注意不需要函数体

·委托名类似函数名，但是并不是函数名，而是一种类型

②委托变量的声明：

委托名 委托变量名;

③委托变量的初始化：

委托变量 = new 委托名(函数名)

或

委托变量 = 函数名

委托变量 +=或-= 函数名

·函数名是实际定义的真正函数，且具备和委托类型一致的返回类型与参数列表。因此这实际上是将委托变量变成了该函数的引用

·后一种初始化方法的函数名不带括号，相较于第一种方法实际上是靠编译器自行根据语境解读指令的目的

·最后一种语法表明一个委托可以引发多个方法的调用，可以使用+=按次序添加方法，或-=删除方法。

④委托变量的使用：

委托变量名(实参列表)

委托变量名.Invoke(实参列表)

·第一种用法是和函数名或函数指针一样的使用方式

·委托变量及其委托类型可以作为普通函数参数传递给其他函数

·第一种方式实际上是通过第二种来实现的，二者在源程序层面等价

## 8.调试和错误处理

1）C#代码中输出调试信息

Debug.WriteLine(string content, string category) //在Output窗口中输出<category>: <content>

Trace.WriteLine(…) //用法类似上一个，但是可以用于release版本

·这些WriteLine方法不同于Console.WriteLine()，因为不支持格式输出（也不输出在控制台上）。可以使用string.Format()先处理格式输出再由它们输出。

2）跟踪点与断点（Visual Studio通用工具）

二者均只能用于Debug模式。

·跟踪点的用处完全类似于Debug.WriteLine()，但不修改代码。跟踪点可以输出该处日志信息、变量的值等。跟踪点算是断点的特殊形式，可设置跟踪点是否引发中断。

·断点可以设置中断条件（如布尔条件和命中次数），而非只能逢点必停。

·二者一般都在Debug|Windows|Breakpoints窗口中查看和配置。

3）判定方法Assert

Debug.Assert(bool condition, string dialog, string output)

//当condition==false时触发提示，弹出dialog对话框，输出output到Output窗口

Trace.Assert(…)

二者的性质和1）中的两个方法差不多。assertion触发时可以选择终止abort、中断retry和忽略ignore。

4）单步执行（Visual Studio通用工具）

Step Into

Step Over

Step Out

Set Next Statement：可选择下一个要执行的指令行，一般用黄色箭头标识

5）Immediate和Command窗口（Visual Studio通用工具）

Immediate窗口可执行和源程序并行的逻辑，在这里也可以查看程序信息和赋值。

Command窗口可视为DOS模式的VS操作窗口。

·两个窗口的功能都可以在其他GUI功能下实现，主要面向DOS模式编程习惯。

6）Call Stack窗口（Visual Studio通用工具）

略

7）错误处理

try catch块也称结构化异常处理structured exception handling，SHE

try

{…}

catch (<exceptionType> e) when (filterIsTrue)

{

…

}

finally

{

…

}

·catch可携带的参数和C++中用法类似，when语句（C#6）表示异常过滤，当为true时才执行该catch块。注意，任何时候同一个try catch结构最多只有一个catch块执行。

·finally包含始终会执行的代码，如未发生异常、在catch块异常处理之后、未处理异常上抛之前。finally块和catch块二者至少有一种存在。

·在Debug|Exceptions中可以通过VS配置异常的更多功能。

**11. Coding Convention**

camel casing: 除了第一个字母外，所有词首字母大写（capitalized）e.g. myVariable

Pascal casing: 包括第一个字母，所有词首字母大写e.g. MyVariable

C#使用两种方法的结合，根据是class, interface, variable, resource来决定采用哪种命名方式

# C#OOP技术笔记

## 一、基本概念

1.接口interface

•接口是把公共实例（非静态）方法和属性组合起来，以封装特定功能的一个集合。定义了接口之后，可以在类的定义中使用它，使得该类具有接口包含的数据和方法。

•接口不能单独存在，不能实例化，也不能包含实现其成员的任何代码，只能在接口中定义成员，具体的实现过程要在调用接口的类中进行（调用接口的类也必须执行定义过程，但可以使用“空”代码实现）。

2.继承

•C#中，类只能直接派生于一个基类。接口也具有继承的功能，但是接口可以继承多个基接口，而类可以支持多个接口。

•密封sealed：密封的类不能用作基类，所以没有派生类。

•C#中，所有的对象都有一个共同的基类object，.Net Framework中，这个基类是System.Object类的别名

•当某一个类作为函数参数时，其派生类均可作为该函数参数，因为函数一定调用的是这些派生类的基类成员，而派生类对象可以用于给基类对象赋值

3.多态性

•C#中没有指针，使用多态性虚函数的办法是，可以用派生类对象为基类对象赋值，而后用该基类对象调用虚函数即可调用派生类对象的函数。也就是用基类对象来代替C++里的基类指针。

•多态性不仅仅局限于同一父类的派生类之间，只要子类或孙子类在继承层次结构中有一个相同的类，即可使用。

•接口同样具有多态性，可以建立接口类型的变量（不能当做实例化来理解），然后将该变量当做基类对象类似处理。

4.对象之间的关系

•包含关系：一个类包含另一个类

•集合关系：基本上是对象数组

5.结构体类型struct

•结构体类型和普通类的差别在于结构体是值类型，而类是引用类型。在用结构体为结构体赋值时，传递值；用类为类赋值时，传递指针或者是关联为引用；C#中的指针被封装在不安全代码操作级别中。

## 二、定义类/接口

1. 类的定义

public/(internal) abstract/sealed class 类名:基类名, 接口1, 接口2…

{}

•C#中，类型本身有修饰符，不仅仅是类的成员，public代表其他项目中的代码可以访问该类，internal代表只有当前项目中的代码才能访问（是默认的设定）。abstract/sealed代表抽象类或密封类，即只能继承或禁止继承。

•C#中派生类的可访问性不能高于基类，如基类是内部internal的，那么派生类不能是public

•C#中基类只能有一个，接口可以支持多个；声明时，基类必须在最前（如果有基类）

2. 接口的定义

public/(internal) interface 接口名: 基接口1, 基接口2…

{}

•接口名称一般以大写字母I开头。

•接口不能指定abstract/sealed属性，虽然像类，但是不是类，同时也不继承System.Object，不过为了方便，System.Object的成员也可以通过接口类型的变量来访问

3. 接口和抽象类

•抽象类和接口虽然都不能实例化，但是都可以定义变量，用来实现多态性

•接口中定义的成员一般都是公共的，因为接口倾向于外部使用，在派生类中的作用也往往是和外部调用相关；而抽象类成员可以是任意访问属性的，它们是派生类的基础类别。

•接口不能包含字段，构造析构函数，静态成员和常量

4. 深度复制和浅度复制

•浅度复制（shallow copy）即一般意义上的拷贝，使用System.Object中的MemberwiseClone()方法，该方法是受保护的，需要自定义公共方法予以调用。

•深度复制（deep copy）可以将引用成员复制成为新的实例的方法，浅度复制只能让引用成员指向源对象中相同的成员所指对象；使用ICloneable接口，对复制过程进行定义

5. 部分类定义

public/… partial class 类型名..

•可以在多个文件中对同一个类进行定义，每个文件中定义其一个部分，最终效果是组合后的类，每个部分类都必须用partial关键字进行定义

•一个类的部分如果支持一个接口，那么也会应用于整个类，同理对基类也一样

## 三、定义类成员/接口成员

•C#中的访问限制符有：public，private（默认），internal，protected四种，和类的访问限制符含义类似

•static可以声明静态成员，const可以修饰常量，常量默认为静态成员，不能再用static修饰

•类的作用域运算符仍然被成员运算符代替，同UnityScript，比如对于静态成员的调用

1. 定义字段

•定义字段和C++方法类似，可以使用readonly关键字，使得其只能在构造函数中赋值或定义中就地初始化。

public/(private)/internal/protected readonly 类型 变量名 = 初值

2. 定义方法

•定义方法使用的特殊关键字有：virtual（虚函数），abstract（相当于C++纯虚函数），override（用于在派生类中重写虚函数），extern（方法定义在外部）

•对方法也可以使用sealed关键字来终止其被派生类重写的特性

public override sealed 类型 函数名()

3. 定义属性

public/… 类型名 属性变量名

{

public/protected/… get

{

return …} //get块必须返回一个和属性同数据类型的值

public/protected/… set

{

… = value} //用来赋值的语句，使用关键字value表示用户提供的属性值

} = initialValue; //可选，用于默认初始化属性值

•get和set称为访问器，其访问性不能和属性的访问性冲突

•属性的使用可以用来更改字段的值，从而加强封装使用的效果

•属性可以使用virtual,override,abstract关键字，但是字段不行，属性类似于数据和方法的中间体

•属性在代码中的使用方式和普通的字段数据成员一样，当为属性变量赋值时，调用set块中的语句，当调取属性变量中的值时，调用get块中的语句（当然两个块的访问性决定了读写操作的可行性）

①基于表达式的属性

类似于基于表达式的方法，可以把属性的定义减少为一行，此时属性相当于只有get访问器。如：

private int myDoubledInt = 5; //字段

public int MyDoubleIntProp => (myDoubledInt \* 2); //只能get为字段2倍的属性

②自动属性

即由编译器自动默认实现字段定义和访问器实现的属性：

public int MyIntProp { get; set } = 9; //带有初始化器的自动属性

自动属性无需定义字段，编译器自行命名和定义一个对应的私有字段。其中set访问器可省略以声明不可用，但get访问器必须有。

4. 构造函数

•基本的构造函数方法和调用顺序等准则和C++一致，不同之处在派生类的构造函数

•默认构造函数为数值提供初值0，false提供给bool变量

•使用构造函数构造对象时，必须前加new运算符，如new className(…)；一般用class声明对象时，无须这么做，和C++一致

public构造函数名(总参数表列): base(参数表列)

•C#中不再使用基类构造函数名，而是使用base关键字调用基类构造函数

public 构造函数名(参数表列): this(参数表列)

•C#中额外增加的构造函数初始化器，意义是调用被声明的构造函数前，调用其他的该类的构造函数（显然必须在此构造函数外事先再声明至少一个构造函数，通过this后的参数表列决定调用哪一个）

①静态构造函数

static 构造函数名()

·静态构造函数一个程序中只执行一次，区别于一般的实例构造函数，它没有访问修饰符和任何参数，且一个类中只有一个。静态构造函数可用于初始化静态成员。

·不同于C++，C#中可以对非常量静态成员进行就地初始化，也称type initializer。但是它的执行时机非常不固定和复杂（未必在程序最初，甚至只要在访问任何静态字段前均可，即可能在已经访问了其他该类成员之后）。如果一个类中有静态构造函数（空的也行），那么type initializer将严格在访问任何成员前执行，但也未必是程序伊始。

5. 隐藏基类方法

new public/… 类型 函数名

•在不使用虚函数和多态性时，即不重写override基类方法时，如果想要改动基类的方法，可以重新在派生类中定义基类的方法，只需在前面加new关键字。因为没有使用多态性，如果试图用基类对象调用派生类的方法，仍会调用基类的该方法（虽然被隐藏但是仍然可以调用）

6. 调用被重写或隐藏的基类方法

•在C#的类体代码中，base和this是两个常用的特殊关键字，base一般用来代表基类的相关事物，this则代表类本体的相关事物。除了在构造函数中的应用外，base和this还可以分别代表基类和本类的对象实例（基类的对象实例指包含在该派生类对象中的基类对象实例），因此二者不能用于静态成员的定义中。

•通过使用base关键字，可以调用基类的成员，尤其是被重写的方法，以此可以达到增添基类方法内容的目的，而不是完全从零重写

•而this作为本类对象实例，可以看做是C++this->指针的代替，只是要用 “.”代替 “->”

7. 定义接口成员

•接口成员的定义特点

①所有成员都是公共的，因此不能使用任何访问限制符，包括get/set前

②接口成员不能包含代码体

③不能定义字段成员

④不能包含static,virtual,abstract,sealed等其他限制符

⑤不能有其他类class或接口定义成员（即不能嵌套，但是接口可以作为类的成员，而不仅仅是被类支持继承）

•接口中的属性定义，可以通过是否加入get/set关键字来设定访问性，而get/set后的代码块包括括号都必须省略（不能在接口中实现，接口中也没有字段成员）

•接口也可以使用new隐藏基接口的方法

8. 在类中实现接口

•实现的接口成员必须全部是public的类成员（因为接口就是要被外部调用的方法和属性集），可以使用virtual,abstract修饰类中被实现的接口成员，但是不能使用static和const

•基类如果支持某接口，那么派生类也支持

•显式实现接口成员：一般意义上的实现接口成员看似和定义普通成员类似，这也是所谓的隐式实现接口成员；而如果在定义时，在成员名前加接口名限定，那么就是显式实现接口，此时该成员只能通过接口来访问，而不能以类的对象实例来访问

•在类的实现中，如果接口的属性成员缺少get/set中的一个存取器，则可以在类中为接口的属性成员添加缺少的属性存取器，但是前提是添加的存取器可访问性必须是非公共的（因为可访问性必须低于接口中属性存取器的可访问性，也就是公共的）

9. 部分类成员

•部分类中可以定义部分方法，即方法的声明和实现出现在两个文件的类的两部分定义中，每个部分方法都需要用partial在最前修饰

•部分方法可以是静态的，但是总是私有的，且不能有返回值，使用的任何参数不能为out，但可以是ref参数，也不能使用virtual,abstract,override,new,sealed,extern修饰，具体的应用是在编译过程中。

10. 为原始类型扩展方法

C#中可以为既定义的类型在不修改其代码的情况下进行外部扩展。使用一个静态类负责此工作，在其中的静态方法的签名中，第一个参数必须为被扩展的原始类型，且前加this关键字。之后便可以如普通成员函数一样调用该方法，调用时第一个参数忽略。

public static class Extends

{

public static object ToJson(this string Json)

{…}

}

调用举例：(string)sss.ToJson();

## 四、集合，比较和转换

1. 集合

•C#中的数组是System.Array类的实例，它们只是集合类(Collection Classes)中的一种类型，虽然集合类似于对象数组，但是并不同于数组，功能更多更全

•System.Collections命名空间中，有几个接口提供了基本的集合功能

①IEnumerable 可以迭代集合中的项，支持foreach语句

②ICollection（继承于IEnumerable）可以获取集合中项的个数，可以把项复制到一 个 简单的数组类型中

③IList（继承于前两者）提供了集合的项列表，允许访问这些项，并提供其他一些相 关的基本功能

④IDictionary（继承于最前两者）类似于前者，但提供了可通过键值而不是索引访问 的项列表

•System.Array类实现了IList，ICollection，IEnumerable，但是不支持IList的一些高级功能，System.Collections.ArrayList是一个高级集合类型，也是一个已经定义好，并且可以使用的类，类似于C++中的容器，不过不对元素类型有要求（因为基本类型是System.Object）

1）自定义集合

•ArrayList是一个已经被定义好的集合，用户也可以自己定义集合，有一些支持各种接口的基类可以使用（它们实现了接口的部分方法但仍需用户实现其他的方法），比如System.Collections.CollectionBase，用来定义类似ArrayList的集合；还有System.Collections.DictionaryBase，用来定义关键字值集合，前者用索引查找元素，类似数组，后者用键值查找元素，类似C++STL的map

·重载索引符indexer[]，和一般的C#重载运算符语法方式不同，它看上去既是一个属性又是一个使用方括号[]的方法。

public Animal this [int animalIndex]

{

get { return …; }

set { … = value; }

}

另外，可见int animalIndex是可以显式替换为其他类型，甚至包括嵌套数组类型

2）迭代器

·迭代器的核心实际上就是一个方法或代码块，基于表面理解，它将一系列返回值按序存储并构建成一个新集合，而后依次在foreach循环中访问。实际上，迭代器返回的序列确实是集合，但可实现迭代器的不局限于集合对象，如一个可以计算并返回素数的类，它不存储素数集合但可以通过迭代器计算素数序列。

·有两种实体可以使用迭代器，对象和方法，它们均可出现在foreach循环的in之后。

被迭代的对象的类需要实现GetEnumerator()方法（或被迭代的对象实现了IEnumerable接口），该方法返回类型IEnumerator；

被迭代的方法需要返回IEnumerable类型，即返回匿名的实现了IEnumerable的对象；

·在GetEnumerator()或被迭代方法中，使用：

yield return <value>;

来依次返回内容，其类型必须和foreach中in之前的变量类型匹配。

使用：

yield break;

可以中断某次的foreach循环。

·原理——

foreach结构的编译工作本质是：首先必须保证in关键字后是一个IEnumerable接口的对象，可将其视为一个容器；调用其GetEnumerator()方法获取一个IEnumerator接口的对象，这就是迭代器（容器对象不必须实现IEnumerable，但必须有GetEnumerator方法）；IEnumerator对象的构建由编译器根据方法内yield语句来进行，简单来说IEnumerator就是一个可以保存当前位置的状态机，其成员方法MoveNext()重新包装了原方法以逐步方式实现其过程。

2. 对象比较

1）类型比较

①GetType()与typeof()

System.Object中的GetType()可以返回一个对象的具体类型，它可以和typeof()的返回结果作相同性比较：

myObj.GetType() == typeof(MyComplexClass)

②ToString()

System.Object中的ToString()可以返回对象类型的字符串表示。

③is运算符

用于检测类型，返回bool：

<operand> is <type>

若type是一个类或接口，operand继承、实现或可封箱（boxing，把值类型转换为System.Object类型或由值类型实现的接口类型，封箱后的对象包含源值类型变量的一个副本的引用）到type，则返回true

若type是一个值类型（结构体），operand可拆箱（unboxing，反过程）到type，则返回true

2）值比较

①使用IComparable和IComparer接口（.NET Framework标准）

·IComparable在要比较的对象的类中实现，可以比较该对象和另一个对象。该接口中的CompareTo(Object x)用以实现比较功能，其返回值为int。

·IComparer在一个单独的类中实现，用于比较任意两个对象。该接口中的Compare()方法接收两个Object对象，也返回int。这个接口也是C#既定义集合类中排序方式设置的接口，如ArrayList.Sort()。

·System.Collections.Comparer类上提供了IComparer接口的默认实现，使用Comparer.Default静态成员获取实例。该对象可以对简单类型和支持IComparable接口的任意类型进行比较。除此之外，可以通过System.Globalization.CultureInfo对象来设置语言种类以及其变体CaseInsensitiveComparer进行不区分大小写的字符串比较。

②运算符重载operator overloading

用法示例：

public static 返回类型 operator + (类型 op1, 类型 op2)

运算符重载函数必须为static，且参数数目和运算符的操作数相同，这不同于C++。

注意事项：

·不能重载赋值运算符=和+=等类似运算符，但可以重载+等运算符。

·不能重载&&和||，但可以重载&和|，前者实际上使用后者实现

·部分运算符必须成对重载，如>和<

·重载true和false（二者是为运算符）可以让对象直接作为条件表达式

·Object.Equals(object op1)和Object.GetHashCode()也是可用于比较的基类方法，如果重载了==和!=等相同性检测运算符，需要考虑是否需要重写override这些基本方法

3. 转换

1）重载转换运算符

C#的转换运算符不限于基本类型，而且可以定义隐式或显式的两种

public class ConvClass1

{…

public static implicit/explicit operator ConvClass2(ConvClass1 op1)

{…

return …;

}

}

注意必须是公共静态不声明返回类型的成员函数，返回类型由运算符名称确定，这和C++一样。另外，隐式和显式声明应该匹配常见的类型转换惯例，如果是会造成精度损失或异常的转换应该声明为显式。

2）as运算符

用于将operand转换为type类型：

<operand> as <type>

·仅限于以下几种情况：

operand的类型是type

operand可以隐式转换为type类型，包括存储派生类对象的基类引用转换为派生类型引用

operand可以封箱到type类型

除此之外，表达式结果为null，包括将基类对象直接转换为派生类。直接使用显式类型转换进行这种转换在C#中会引发异常，而as运算符可以更好的处理这种情况。

## 五、泛型

虽然泛型类似于C++模板，但是泛型的实例化是在运行时进行的，而非编译时。另外，由于所有类型都继承自Object，因而很多库类型和接口都是有泛型和非泛型两种变体。非泛型显然需要依赖多态性进行访问，是弱类型的。比如集合类的非泛型在System.Collections下，而泛型在System.Collections.Generic下。

1）可空类型

System.Nullable<int> var1;

int? var1; //另一种写法

声明一个用法类似int（或其他基本类型）的对象，但可以赋值null

·??运算符（空接合运算符null coalescing operator）

以下两种写法等价，??运算符用于给可空类型对象赋予一个非空默认值。

op1 ?? op2

op1 == null ? op2 : op1

·?.运算符（Elvis运算符或空条件运算符）

以下代码会在orders为null时返回null给count，即简化空值检测。

int? count = customer.orders?.Count();

2）常用泛型集合类

①List<T>

②Dictionary<K,V>

·Add(key, val)

添加条目

·Keys和Values

是可以进行foreach迭代的属性

·KeyValuePair<K,V>

是容器中的元素（条目）类型

3）定义泛型类型

·default关键字

default(T)

可以返回类型T的“默认值”，根据T是值类型或引用类型，分别返回0或null

·约束类型constraining

用法示例：

class MyGenericClass<T1, T2> : MyBaseClass, IMyInterface where T1 : constraint11, constraint12 where T2 : constraint2 {…}

约束限制了泛型可实例化的情形，可用的约束如下：

|  |  |
| --- | --- |
| 约束 | 定义 |
| struct | 类型必须是值类型 |
| class | 类型必须是引用类型 |
| *base-class* | 类型必须是base-class或继承自它，这个约束名是自定义的 |
| *interface* | 类似上一个 |
| new() | 类型必须有一个公共的无参数构造函数 |

注意：

new()约束必须是类型的最后一个约束；

base-class约束可以使用泛型类型参数，即两个类型必须是继承关系，此时称为裸类型约束naked type constraint；

如果派生泛型类型，则子类的约束只能比父类更严格

·泛型类型转换运算符

类似于常规类型转换运算符

4）定义泛型方法

举例：

public T2 GetDefault<T2>() where T2 : T1 {}

5) 定义泛型委托

举例：

public delegate T1 MyDelegate<T1, T2> (T2 op1, T2 op2) where T1: T2;

6）变体variance

与多态性类似（可能底层实现是基于多态性的），是对泛型接口类型在同类族的类型参数之间转换的技术。作为高级语言中的较高级，C#如何实现一些复杂的编程辅助技术，只能通过查看中间代码、标准文档了解。

·协变covariance

举例：

public interface IMethaneProducer<out T> {…}

协变类型参数只能用作方法的返回值或属性的get访问器。协变的类型参数可以向上转换，如IEnumerable<Derivative>可以转换为IEnumerable<Base>

·抗变contravariance

public interface IMethaneProducer<in T> {…}

抗变类型参数只能用作方法参数，不能作返回类型。抗变类型参数可以向下转换，如IComparer<Base>可以转换为IComparer<Derivative>

## 六、事件

事件类似于异常，都由对象引发，并且由对应的代码处理。但是事件的不适用try catch块结构，而是使用订阅subscribe。

·事件和委托变量的用法几乎完全一致（除了声明语法区别）。事件基于委托，可以猜测事件是.NET框架的要求，而委托是C#层面的机制。

1. 订阅事件

·订阅事件使用特殊的语法如下，一个事件可以有任意多的订阅方法，只要满足委托的类型要求。当事件引发时，会依次调用每个处理程序。

事件 += new 委托类型名(处理方法名);

事件 += 处理方法名; //即省略委托类型，但这种方法会导致程序可读性降低

事件 += delegate(参数签名) {方法体}; //匿名方法订阅，最好不要调用本地局部变量，否则会使其实际成为外部变量而持续占用资源

·注意，未有任何订阅的事件可以通过赋值符号=检测为null，可以通过!= null的语法来检测订阅。

·不带有()的函数名在C#术语下称为方法组。

2. 定义事件

·定义事件前需要首先定义其处理方法的委托类型！

event 委托类型名 事件名;

·事件处理方法中最常见的两个参数就是引发事件的对象的引用，如object；和事件传送的参数。.NET提供了两个标准委托类型EventHandler和EventHandler<T>泛型，二者返回类型为void，后者允许指定要使用的事件传送参数类型。另外，.NET还提供了System.EventArgs用于派生事件传送参数对象。当使用EventHandler委托类型定义事件时，仍需要传递EventArgs.Empty作为实参值。

·定义事件最好使用void类型的委托，如果有返回值，那么在多个订阅并存的情况下，只允许访问最后一个返回值。

3. 引发事件

·用法就像是调用事件名为名称的方法

事件名(委托参数);

## 七、特性[attribute]与反射

1）特性

特性是添加在程序元素（方法、数据成员等）上的额外信息，.NET既定义的特性可以定制编译过程（取决于编译器支持），但是用户也可以自定义特性实现各种功能（但是一般无法干预编译过程）。特性本质上也是类或对象，所有特性均继承自System.Attribute，且定义名称必须以Attribute结尾（调用时可省略Attribute后缀，编译器会自动加上该结尾以查找，不省略也可以）。

·所有特性的定义本身也用一个特性System.AttributeUsage来标记，同时配置本特性的一些性质，如适用范围等。

[AttributeUsage(AttributeTargets.xxx, AllowMultiple=false, Inherited=false)]

public class AttributeNameAttribute: Attribute

AttributeTargets是一个枚举类型，用以指定特性的适用元素

AllowMultiple可选参数指定同一个特性是否可以重复应用于同一程序元素

Inherited可选参数设置本特性是否可以被自动延伸到派生类、重写方法等衍生元素上

·特性调用时，主要语法结构类似于调用其构造函数，除此之外，还有类似于指定命名参数的语法来配置公共属性或字段。这里的语法一般不适用于其他地方。

[AttributeName(firstArg, …, Prop/FieldName = Value)]

2）反射

①基于System.Type类

该类用于存储和查看与一个类型有关的全部基本信息。获得一个类型的Type类对象使用System.Object.GetType()或typeof()运算符。

Type的常用属性、方法和功能：

②基于System.Reflection.Assembly类

Assembly代表一个程序集，通过加载程序集之后使用各种属性和方法，可以查阅到该程序集中的各种元数据信息。

# C#高级用法和话题

## lambda expression用法

lambda表达式由三个部分组成：（未类型化的参数列表）、=>、函数体。表达式返回一个匿名函数的委托。它的实现依赖于编译器的类型推导和自动构建方法，不过也可以明确声明参数类型和返回值及其类型。lambda表达式存在的意义就在于特定条件下简化编程。

1）lambda表达式参数

·可以声明参数类型或不声明，但是不能混合。

·无参数的lambda表达式使用()空括号表示参数

() => Math.PI

·单独一个隐式类型参数可以省略其外部括号

param1 => param1 \* param1

2）lambda表达式语句体

可以使用一般方法的书写方式，但是lambda表达式语句体的本质特色是单行表达式可以自行推导出返回值，而不用return显式声明。

3）用法

①实现.NET框架定义的委托

lambda表达式可用于任意委托，但是有几个框架内既定义的委托，常常用于引导程序员赋予其lambda方法。

·System.Action：无参，void

·System.Action<>：最多8个参数，void

·System.Func<>：最多8个参数，非void

## 初始化器

1）对象初始化器

<ClassName> <objectName> = new <ClassName>(…)

{

<propertyOrField1> = <value1>,

<propertyOrField2> = <value2>,

…

}

即在声明对象时，可以在调用构造函数之后对任意的可访问且可写属性进行赋值（等效于先构造后访问修改的多条语句）。如果调用默认构造函数，则括号可省略。对象初始化器可以嵌套，即在value处调用另一对象的初始化器。

·进一步而言，new <ClassName>在默认构造函数和其他无歧义条件下可省略。

2）集合初始化器

将上述初始化器中的属性或字段改为依次书写的集合元素即可（不用赋值=表达式），同样的，对自定义类型元素也可嵌套初始化器。

·事实上，编译器只是在构建完容器对象后依次调用Add()方法来添加每个元素，如果集合类中没有定义Add()，则会报错。

①索引初始化器（C#6）

new Dictionary<int, string>()

{

[1] = “a”,

[2] = “b”,

…

};

可以使用上述示例的方法对支持索引器的类型进行初始化。

## 类型推理和匿名类型

1）类型推理

var <varName> = <value>;

用法类似C++中的auto。

2）匿名类型

var <varName> = new

{

<propertyOrField1> = <value1>,

<propertyOrField2> = <value2>,

…

}

匿名类型一般用于创建纯数据型类型，即只包含（只读）属性，无法后续修改。匿名类型实际上有一个不合命名法的名称，无法在程序中被用户调用。

var <varName> = new [] {…}

·也可以用来声明一个匿名类型的数组类型，在{}中依次使用上一个句式声明匿名类型对象，注意所有匿名类型对象的属性必须相同，即所有匿名类型是一致的。

## 动态查找和动态类型/变量

dynamic是动态类型的关键字，它和var不同，后者是编译期间可推导的，而dynamic是运行时才绑定的类型。动态查找功能由DLR动态语言运行库支持。

·dynamic可作为类型用于声明变量、方法参数和返回值等。

·动态查找功能在和动态类型语言对接时才应使用，其他时候应尽量保持C#的强类型化特点。

## 高级方法参数

1）可选参数

也称带默认值的参数，用法和C++中一样。

2）Optional特性

[Optional] <paraType> <paraName>

也是可选参数，但是不能提供默认值

3）命名参数

对已经定义的可选参数方法，可以使用命名参数调用来打破一些限制，比如可选参数的顺序。但是必须参数仍必须在靠前的位置。命名参数的语法仅限调用时，定义时的语法不做修改。

MyMethod(requiredParaValue, optionalPara: optionalParaValue);

optionalPara是形参的名称。

# C#额外知识积累

## @符号的用法

①字符串处理

C#中字符串常量可以以@开头声明，这样的优点是转义序列“不”被处理，按“原样”输出，即我们不需要对转义字符加上 \ （反斜扛），就可以轻松coding。如，

string filePath = @"c:\Docs\Source\a.txt" // rather than "c:\\Docs\\Source\\a.txt"

·如要在一个用@引起来的字符串中包括一个双引号，就需要使用两对双引号了。

这时候你不能使用\来转义引号了，因为在这里\的转义用途已经被@“屏蔽”掉了。如，

@"""Ahoy!"" cried the captain." // 输出为： "Ahoy!" cried the captain.

②标识符命名

在C#规范中，@可以作为标识符（类名、变量名、方法名等）的第一个字符，以允许C#中的保留关键字作为自己定义的标识符。注意，@虽然出现在标识符中，但不作为标识符本身的一部分。

因此，可以定义一个名为class的类，声明时写为@class。这样，对于跨语言的移植带来了便利。因为，某个单词在C#中作为保留关键字，但是在其他语言中也许不是。

## await和async与异步编程