目录

[书写规则 1](#_Toc4954761)

[C#基本语法 1](#_Toc4954762)

[0.Preprocessor Directives & Comments 1](#_Toc4954763)

[1. Varaibles & Constants 1](#_Toc4954764)

[2. Conditional Statements, Switch Statements 3](#_Toc4954765)

[3. Loops 4](#_Toc4954766)

[4. Type Convertion/Casting 4](#_Toc4954767)

[5. Complex Variable Type 5](#_Toc4954768)

[6. Functions 6](#_Toc4954769)

[7. Delegate 7](#_Toc4954770)

[8.调试和错误处理 7](#_Toc4954771)

[C#OOP技术笔记 7](#_Toc4954772)

[一、基本概念 7](#_Toc4954773)

[二、定义类/接口 8](#_Toc4954774)

[三、定义类成员/接口成员 9](#_Toc4954775)

[四、集合，比较和转换 10](#_Toc4954776)

# 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

•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(p115/134)

*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;

}

## 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。

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

·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

④委托变量的使用：

委托变量名(实参列表)

·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

## 8.调试和错误处理

**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

•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

}

•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）

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）

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。

## 四、集合，比较和转换

•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，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

待续

## 五、事件