[1. C# versions and main features 1](#_Toc471301096)

[2. 关键字 1](#_Toc471301097)

[2.1. ReadOnly 1](#_Toc471301098)

[2.1-1. 定义 1](#_Toc471301099)

[2.1-2. 是字段才有的关键字，属性不能使用 1](#_Toc471301100)

[2.1-3. 只读字段只能在构造函数或者变量声明时赋值 1](#_Toc471301101)

[2.1-4. ReadOnly Field 也可以赋值很多次 1](#_Toc471301102)

[2.1-5. 用readonly 标识的集合仍可以添加和修改元素 2](#_Toc471301103)

[2.1-6. 用readonly标识的引用类型字段仍可以修改引用类型的值 3](#_Toc471301104)

[2.1-7. 针对struct， 内嵌的属性自然是不能更改的 3](#_Toc471301105)

[2.1-8. No readonly reference instance 4](#_Toc471301106)

[2.1-9. ReadOnlyCollection 5](#_Toc471301107)

[2.1-10. ReadOnly和Const 区别 5](#_Toc471301108)

[2.2. Using 语句 5](#_Toc471301109)

[2.2-1. Using 多余的命名空间并不会影响运行时的性能 5](#_Toc471301110)

[3. 语法 7](#_Toc471301111)

[3.1. Int 和 Char的互相转换 7](#_Toc471301112)

[3.1-1. Int to char 7](#_Toc471301113)

[3.1-2. Char to int 7](#_Toc471301114)

[3.2. Const 和 Static 9](#_Toc471301115)

[3.2-1. Const 和 static 不能同时使用 9](#_Toc471301116)

[3.2-2. Const 成员只能用类名访问，不能用instance 访问 10](#_Toc471301117)

[3.2-3. Static或者const 在类内部访问时，要么不加限定符，要么只能加类名 10](#_Toc471301118)

[3.2-4. Static Readonly 可以和const 起到类似的作用，又带来了灵活性 10](#_Toc471301119)

[3.2-5. Static readonly 不能在实例构造函数里初始化 10](#_Toc471301120)

[3.2-6. Const 赋值时，右侧必须是字面常量 11](#_Toc471301121)

[3.3. Static constructor： 静态构造函数 12](#_Toc471301122)

[3.3-1. 静态构造函数和实例构造函数可以共存 12](#_Toc471301123)

[3.3-2. 静态构造函数不允许接收参数 13](#_Toc471301124)

[3.3-3. 静态构造函数前面不允许有访问修饰符 13](#_Toc471301125)

[4. Class 14](#_Toc471301126)

[4.1. 互相引用或者递归引用自身时的堆栈溢出 14](#_Toc471301127)

[4.1-1. Class在构造的时候互联引用，堆栈溢出 14](#_Toc471301128)

[4.1-2. 递归引用自身，堆栈溢出 15](#_Toc471301129)

[4.1-3. 引用自身的第二种写法，仍然堆栈溢出 15](#_Toc471301130)

[4.1-4. 引用自身的正确写法 16](#_Toc471301131)

[4.2. Sealed 有什么用： Design 问题 16](#_Toc471301132)

[5. Struct 17](#_Toc471301133)

[5.1. 定义 17](#_Toc471301134)

[5.2. Struct 可以实现interface 18](#_Toc471301135)

[5.3. Struct 不允许有Protected 的修饰符 18](#_Toc471301136)

[5.4. Struct 不能显式声明无参的构造函数 19](#_Toc471301137)

[5.5. Struct 不允许实例字段的初始化，static 字段可以 20](#_Toc471301138)

[5.6. Struct 可以不使用new ，但作为局部变量时必须先初始化 20](#_Toc471301139)

[5.7. Struct作为全局变量时，编译器允许不初始化就是用 22](#_Toc471301140)

[5.8. 使用new 关键字生成的struct， 编译器认为其已经初始化过了 22](#_Toc471301141)

[5.9. 不管用不用new， struct 作为值类型都是生成在栈上 22](#_Toc471301142)

[5.10. Struct 作为引用对象的filed 时，struct 各字段不能单独赋值 22](#_Toc471301143)

[5.11. Class 和 struct的区别 25](#_Toc471301144)

[5.11-1. 从使用场景的角度 25](#_Toc471301145)

[5.11-2. 从语法的角度 25](#_Toc471301146)

[6. Interface 26](#_Toc471301147)

[6.1. Interface 中的方法都不需要加public 关键字，也不能加 26](#_Toc471301148)

[6.2. 对于 接口继承的情况， 实现类可以只写子接口，也可以把子接口和接口都写上 26](#_Toc471301149)

[6.3. 几个常见的接口 27](#_Toc471301150)

[6.3-1. IEnumerable 27](#_Toc471301151)

[6.3-2. IEnumerator 28](#_Toc471301152)

[6.3-3. 以Array类为例看IEnumerable 和IEnumerator 28](#_Toc471301153)

[6.3-4. ICollection 29](#_Toc471301154)

[6.3-5. ICollection的SyncRoot 和IsSynchronized 29](#_Toc471301155)

[6.3-6. IList 33](#_Toc471301156)

[6.3-7. IComparable 的CompareTo 必须返回int类型的值 33](#_Toc471301157)

[6.3-8. Comparison<T> delegate 34](#_Toc471301158)

[6.3-9. IComparer 和IComparer<T> 接口 34](#_Toc471301159)

[6.3-10. IComparable 接口 和IComparable<T> 接口 35](#_Toc471301160)

[6.3-11. Int32的排序接口实例 36](#_Toc471301161)

[7. Enum 37](#_Toc471301162)

[7.1. 枚举在IL内部被转换为int32 的值类型 37](#_Toc471301163)

[7.1-1. 不过在Class Enum 内部应该用的uLong 37](#_Toc471301164)

[7.2. 枚举的Tostring() 显示自己的名称 38](#_Toc471301165)

[7.3. FlagsAtribute 38](#_Toc471301166)

[7.4. 枚举内部实现 39](#_Toc471301167)

[7.4-1. 枚举继承自System.Enum 类 39](#_Toc471301168)

[7.4-2. Values 和 names 数组 40](#_Toc471301169)

[7.4-3. Tostring() 40](#_Toc471301170)

[7.5. C#编译器不允许直接继承System.Enum 41](#_Toc471301171)

[7.6. 可以把一个不存在的value 值强制转化为Enum 42](#_Toc471301172)

[7.7. Enum 项目可以赋负值 43](#_Toc471301173)

[8. Collections 44](#_Toc471301174)

[8.1. C#没有List类 44](#_Toc471301175)

[8.2. Array, ArrayList, List<T> 44](#_Toc471301176)

[8.2-1. 这三个其实都不能叫链表，而都应该叫数组 45](#_Toc471301177)

[8.2-2. Array 是抽象类 45](#_Toc471301178)

[8.2-3. Array的Sort 和 List 的Sort 45](#_Toc471301179)

[8.2-4. Sort时可能会报错 46](#_Toc471301180)

[8.3. List<T>的排序问题 47](#_Toc471301181)

[8.3-1. IList并没有实现排序的接口,List<T>也没有 47](#_Toc471301182)

[8.3-2. 通过Linq的扩展方法实现List<T>的排序 47](#_Toc471301183)

[8.3-3. 使用List<T>的sort方法 49](#_Toc471301184)

[8.3-4. 使用s1.age.CompareTo(s2.age) 而不是s1.age >= s2.age 52](#_Toc471301185)

[9. IO 52](#_Toc471301186)

[9.1. 默认的Encoding为UTF8 52](#_Toc471301187)

[9.1-1. UTF8 BOM 是什么 52](#_Toc471301188)

[9.2. 默认的BufferSize 为1024 53](#_Toc471301189)

[9.3. End of File 54](#_Toc471301190)

[9.3-1. ReadFile 遇到end 时返回0 而不是-1 54](#_Toc471301191)

[9.3-2. End of File 总是单独读的，只有所有字节读完了才会遇到end of file 54](#_Toc471301192)

[9.4. Read 的时候，Byte[] 即使入参，也是出参 55](#_Toc471301193)

[9.5. Read 自己有一个Buffer 55](#_Toc471301194)

[9.6. Encoding 是类而不是枚举 56](#_Toc471301195)

[9.6-1. 主要作用是把字符char， 字符串string 和byte 数组间转换 56](#_Toc471301196)

[9.6-2. Stream 只能read，write byte 数组 57](#_Toc471301197)

[9.6-3. ASCII， UTF8等是Encoding类的static 成员 59](#_Toc471301198)

[9.6-4. Encoding 类都有一个相关的Encoder 和Decoder类 59](#_Toc471301199)

[9.6-5. 系统预制的Encoding 类 59](#_Toc471301200)

[9.6-6. 拿到其他的Encoding 类 60](#_Toc471301201)

[9.6-7. GBK 其实用的是GB2312 63](#_Toc471301202)

[9.6-8. EncodingInfo 64](#_Toc471301203)

[10. 关键类和结构 65](#_Toc471301204)

[10.1. Convert 65](#_Toc471301205)

[11. C# 异步编程 66](#_Toc471301206)

[11.1. Thread 和 TheadPool 66](#_Toc471301207)

[11.1-1. 一个进程内部只有一个ThreadPool实例 66](#_Toc471301208)

[11.1-2. 前台线程和后台线程 66](#_Toc471301209)

[11.2. TPL和TAP 区别 68](#_Toc471301210)

[11.3. TPL Task Parallel Library 68](#_Toc471301211)

[11.3-1. TPL的历史 68](#_Toc471301212)

[11.3-2. System.Threading.Tasks 68](#_Toc471301213)

[11.3-3. TPL 与直接使用Thread的区别： TPL基于ThreadPool 68](#_Toc471301214)

[11.3-4. Task 就实现了IThreadPoolWorkItem 接口 69](#_Toc471301215)

[11.3-5. Task<TResult> : Task 70](#_Toc471301216)

[11.3-6. Task 重载的几个Run方法都是Static 方法，且返回Task 类型的对象 70](#_Toc471301217)

[11.3-7. CancellationToken 是个Struct 71](#_Toc471301218)

[11.3-8. Action 和 Func<TResult> 72](#_Toc471301219)

[11.3-9. Task 执行之后拿到它的执行结果:ContinueWith 74](#_Toc471301220)

[11.3-10. Task.Run 不在需要调用Task.Start 74](#_Toc471301221)

[11.3-11. Task.Factory 和 TaskFactory 其实是一回事 77](#_Toc471301222)

[11.3-12. Parallel 类 77](#_Toc471301223)

[11.4. Async 和 await 79](#_Toc471301224)

[11.4-1. Async 在函数定义时用， await 在函数调用时用 79](#_Toc471301225)

[11.4-2. 没有async 关键字的时候，不能直接在定义为Task<TResult> 的方法中返回TResult 类型 79](#_Toc471301226)

[11.4-3. 使用了async 关键字后，可以直接返回TResult 方法 79](#_Toc471301227)

[11.4-4. 使用了async关键字后，反而不能返回Task 类型了 79](#_Toc471301228)

[11.4-5. 使用async 不使用await 没错误，会有warning 80](#_Toc471301229)

[11.4-6. 在调用async 方法时，还必须使用Task 而不是TResult 来接收返回值 80](#_Toc471301230)

[11.4-7. 能用async 修饰的函数，其返回类型只能为void ， Task 或者 Task<TResult> 81](#_Toc471301231)

[11.4-8. Await 只能出现在用async 修饰的方法内部 81](#_Toc471301232)

[11.4-9. Await 之后只能跟Task 或者 Task<TResult> 对象 81](#_Toc471301233)

[11.4-10. TaskAwaiter 和 TaskAwaiter<TResult> 82](#_Toc471301234)

[11.4-11. Async 和await 设计的线程 82](#_Toc471301235)

[11.4-12. 使用async 被C#编译器加了很多代码： AsyncStateMachineAttribute 83](#_Toc471301236)

[11.5. APM 异步编程模型 84](#_Toc471301237)

[11.5-1. History 84](#_Toc471301238)

[11.5-2. IAsyncResult 84](#_Toc471301239)

[11.5-3. AsyncCallBack delegate 85](#_Toc471301240)

[11.5-4. FileStream 为例 86](#_Toc471301241)

[11.6. EAP 87](#_Toc471301242)

[11.7. TPL：TaskFactory.FromAsync与委托的异步调用 87](#_Toc471301243)

[12. C# 网络编程 90](#_Toc471301244)

# C# versions and main features

<c# version & features.docx>

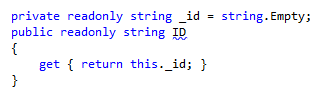
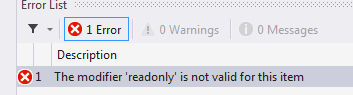
# 关键字

## ReadOnly

### 定义

The readonly keyword is a modifier that you can use on **fields**. When a field declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the **declaration or in a constructor** in the same class.

### 是字段才有的关键字，属性不能使用

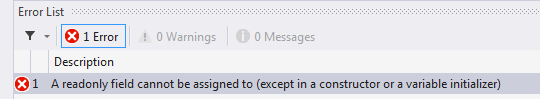
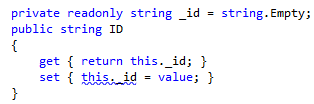
 

因为属性本质上是方法，方法谈不上只读还是可读写。属性是对字段（Field）的封装，只有字段，也就是数据才有只读，可读写的概念。

方法通过是否设置set 方法来实现对字段是否可读写的封装

### 只读字段只能在构造函数或者变量声明时赋值

否则编译器就会报错： A readonly field cannot be assigned to (except in a constructor or a variable initializer)



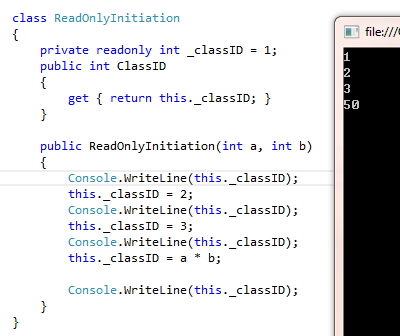
所以只读字段对应 的属性也不能有set方法

* For a instance field, in the instance constructors of the class
* for a static field, in the static constructor of the class

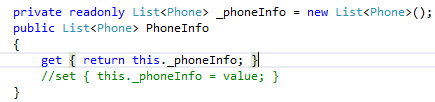
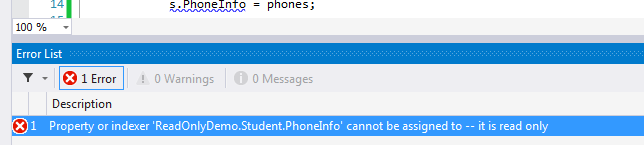
### ReadOnly Field 也可以赋值很多次

只要是这种赋值发生在field 声明或者在构造函数内部就可以。Readonly 强调的是一旦对象初始化完成，就不能再修改。而不是说只能进行一次赋值。

下面这段代码完全没问题

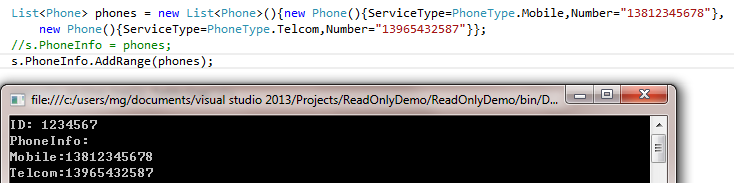


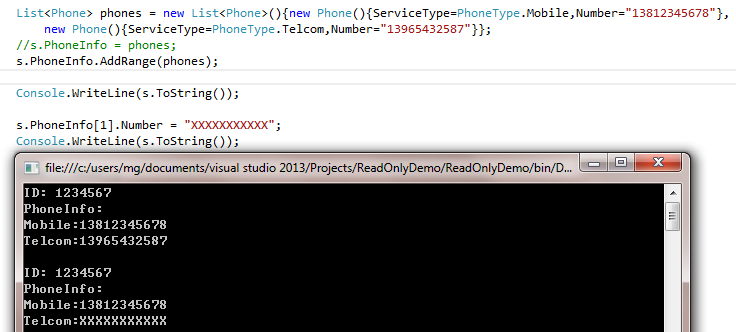
### 用readonly 标识的集合仍可以添加和修改元素

给只读集合赋值会报错。

但是给集合添加对象不会出错

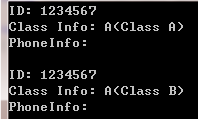
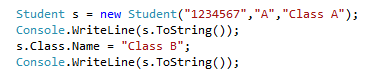
还可以改变集合里面每个元素的属性



还可以对collection的每一个item重新赋值



### 用readonly标识的引用类型字段仍可以修改引用类型的值

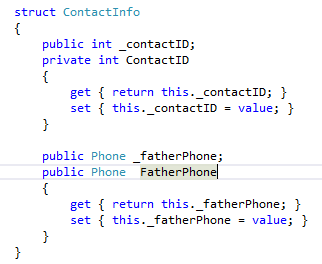
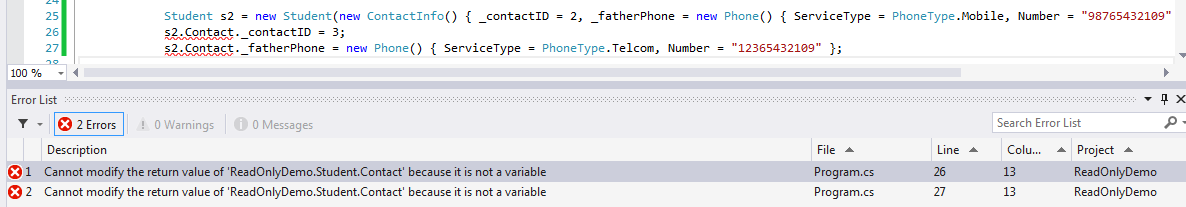
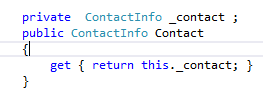


属性里面的内容仍然可以修改

### 针对struct， 内嵌的属性自然是不能更改的

这是和boxing， un-boxing机制相关的。

而struct本身如果用readonly 修饰，自然是不能改的

### No readonly reference instance

there's a big difference between making the field readonly and making the object itself immutable.

<http://stackoverflow.com/questions/681287/how-to-make-a-reference-type-property-readonly>

C#本身并不存在一种机制可以让object itself 本身不能被改变。有一种替代的方案是immutable 类型。但是.net 框架并没有提供一种现成的类或者attribute 可以很方便的声明一个immutable 的类型。所有的机制都需要开发者自己去完成

### ReadOnlyCollection

命名空间在System.Collections.ObjectModel， 而不是System.Collections.Generic 或者System.Collectios;



该类型没有add 或者insert 方法，所以无法直接往里面添加元素。

### ReadOnly和Const 区别

A const field can only be initialized at the declaration of the field

A readonly field can be initialized either at the declaration or in a constructor

readonly fields can have different values depending on the constructor used.

while a const field is a compile-time constant, the readonly field can be used for runtime constants （readonly 的使用场景是运行时常量）

比如系统开始运行的时刻：在系统一经运行开始就不能再更改，但是又不是一个编译时可以知道的常量。



## Using 语句

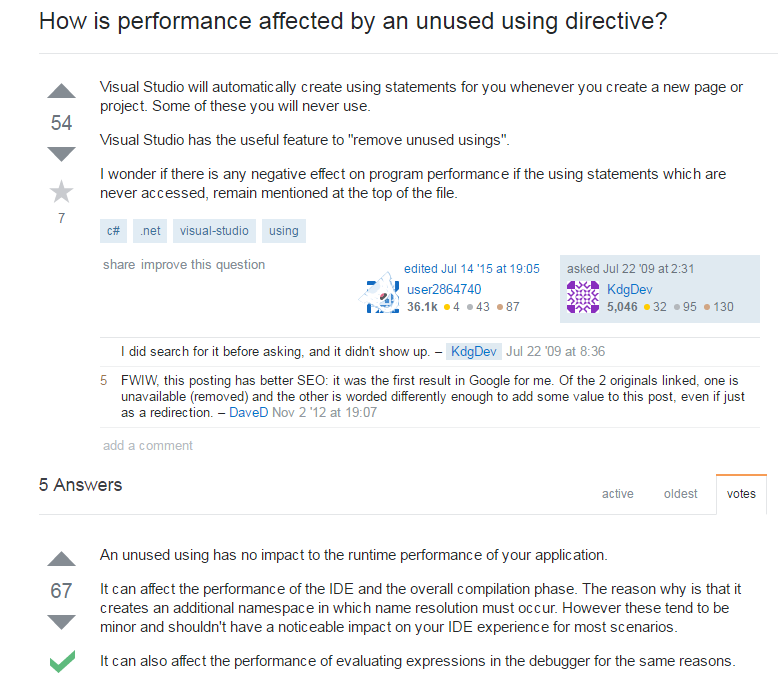
### Using 多余的命名空间并不会影响运行时的性能

影响的是编译阶段的性能和智能提示的性能，并不影响运行时的性能。Java里面的Import也是如此。不过还是有一些理由不引入不用的命名空间。

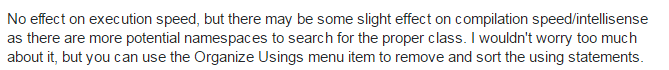
避免冲突的可能性

更清晰干净的代码

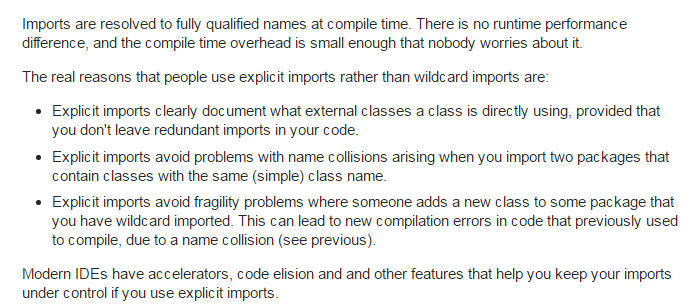
**.NET**

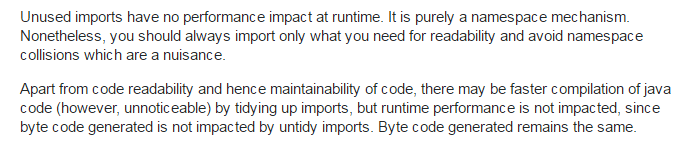






**Java**

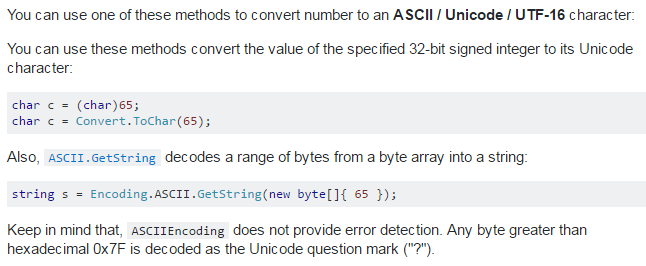




# 语法

## Int 和 Char的互相转换

### Int to char



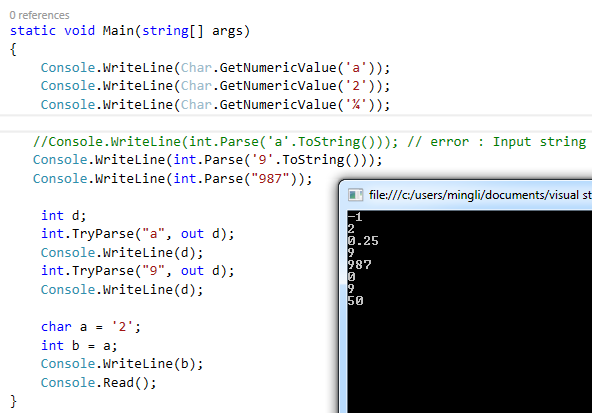
### Char to int

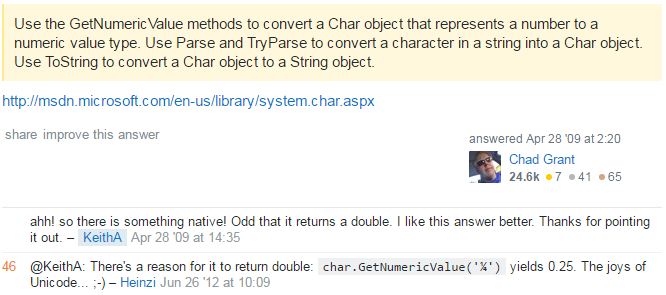
其实这里面分两种情况 这个char 本身是数字形式的，

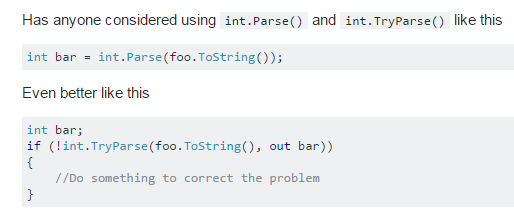
比如’1’, ‘2’, ‘9’ 等，那么它转化为int 就是对应的转化为 1， 2， 9

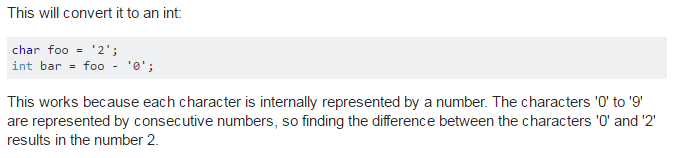
如果用’a’ , 因为’a’ 并不是一个有效的number (0~9)， 所以’a’ 不能转换为int，强转则报错

另一种情况是取的字符的ASCII 编码， 这时不仅字符是不是’1’ 到’9’ 都可以。‘a’也可以。而且’2’ 也不在得到十进制的2，而是得到50(0x32)



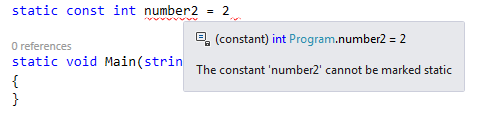






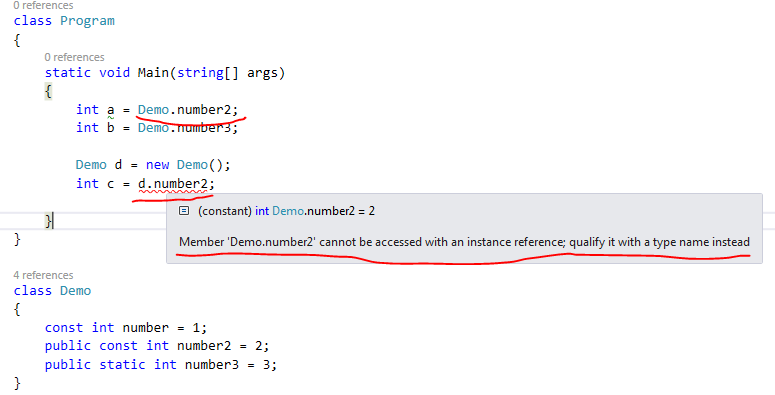
## Const 和 Static

### Const 和 static 不能同时使用

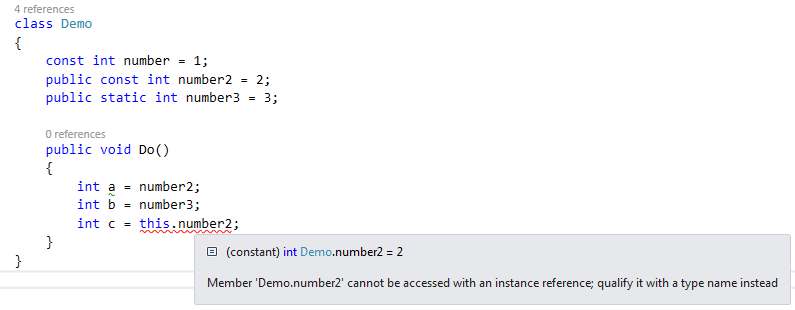


因为const 隐含的就是static 了，如果再允许 static 和 const 连用。可能会带来迷惑： static const 和const 有什么区别么?

### Const 成员只能用类名访问，不能用instance 访问



### Static或者const 在类内部访问时，要么不加限定符，要么只能加类名



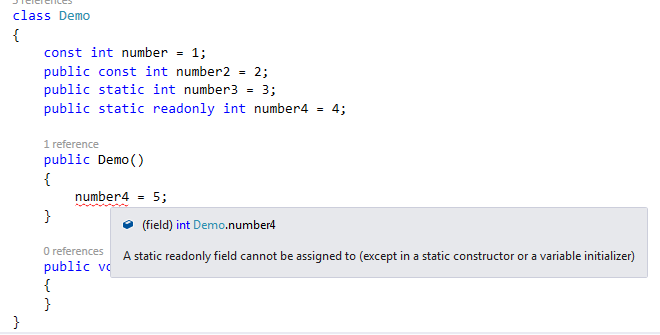
### Static Readonly 可以和const 起到类似的作用，又带来了灵活性

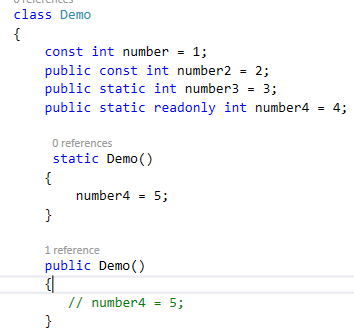
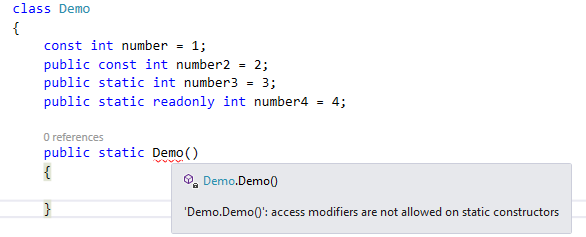
Const 在编译时刻就用字面字符代替了，优点是性能好，缺点是一点改了值，就必须重新编译。在const 值跨dll 使用时，会产生问题。因为如果const 值变了，另一个dll 即使不编译也不会产生编译错误，但运行时就麻烦了

Static readonly 即能起到const 的常量作用，又不会导致跨dll 使用时的问题，当然其性能比const 要稍差。一般跨dll 使用时用static readonly

### Static readonly 不能在实例构造函数里初始化

只能在静态构造函数里初始化



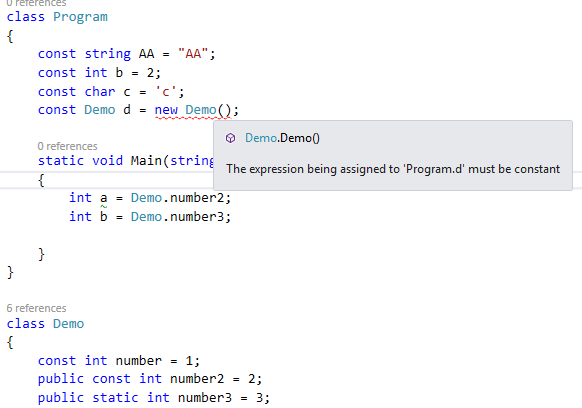
 

注意 Static constructor 前面不允许有访问符

### Const 赋值时，右侧必须是字面常量

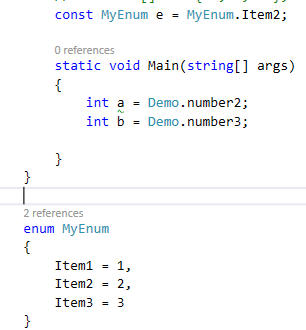
String ， int， char ,enum可以定义成常量

自己定义的类型不行



自己声明的类显然不能定义为常量，byte 数组也不行，

Enum 可以

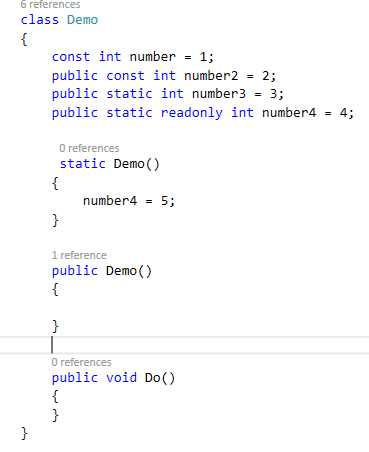


## Static constructor： 静态构造函数

用来初始化static 成员。

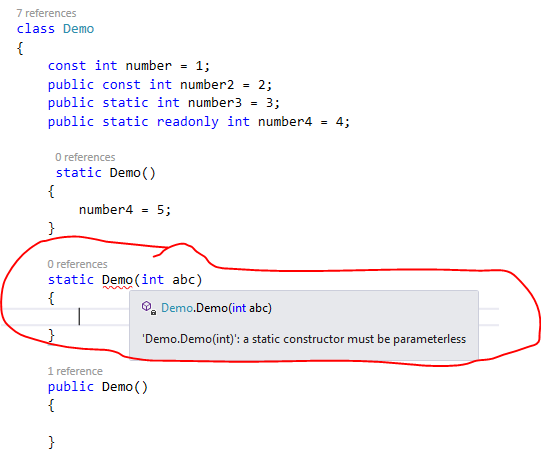
A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed **once only**. It is called automatically **before the first instance is created or any static members are referenced.**

### 静态构造函数和实例构造函数可以共存



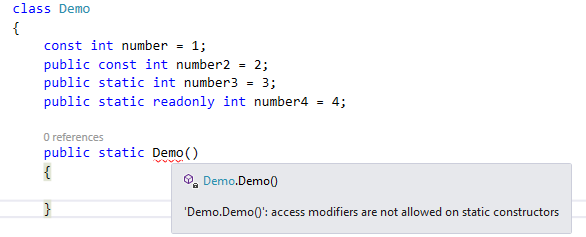
### 静态构造函数不允许接收参数

因为它是由CLR 自动调用的，所以CLR显然不知道该如何传参。



### 静态构造函数前面不允许有访问修饰符

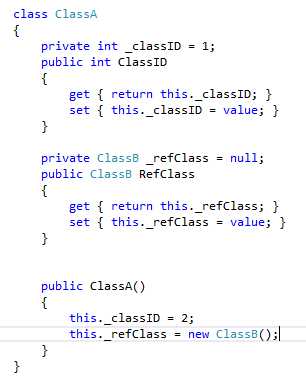
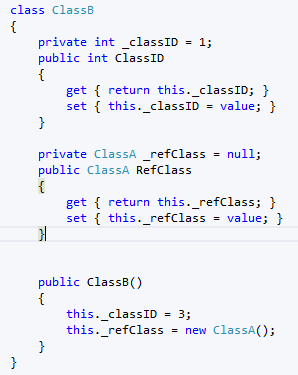
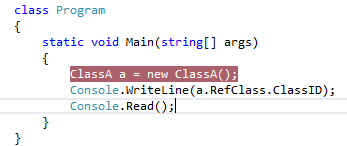
原因可能还是由于CLR必须要自动调用这个静态构造函数。加成private 就麻烦了。

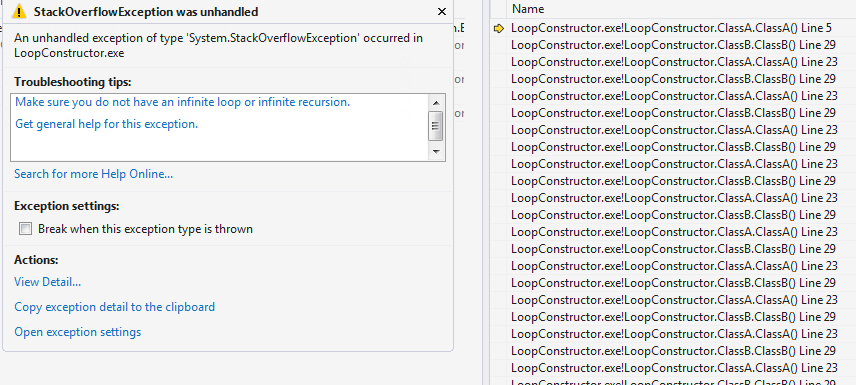


# Class

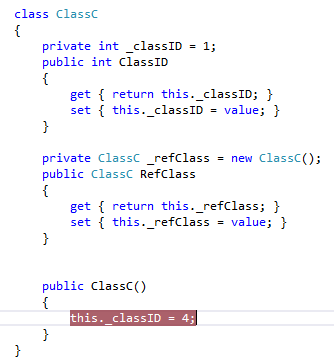
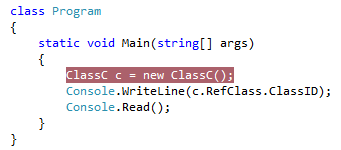
## 互相引用或者递归引用自身时的堆栈溢出

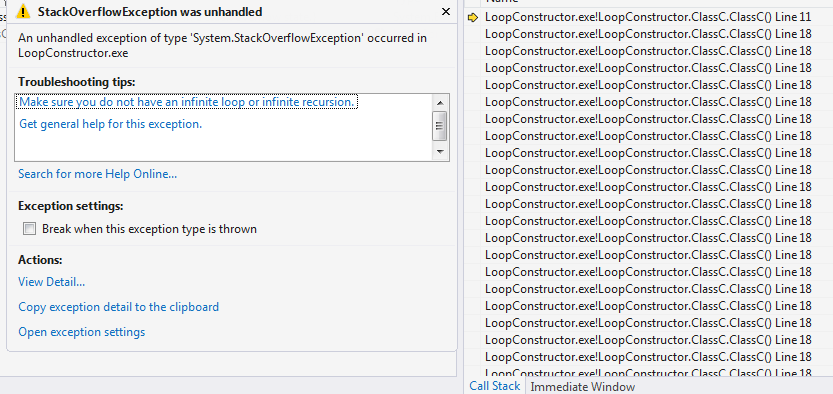
### Class在构造的时候互联引用，堆栈溢出

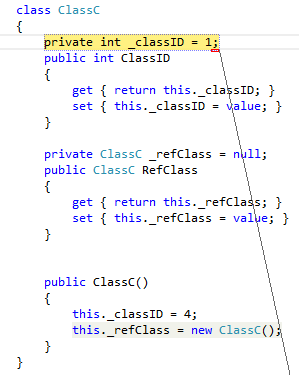
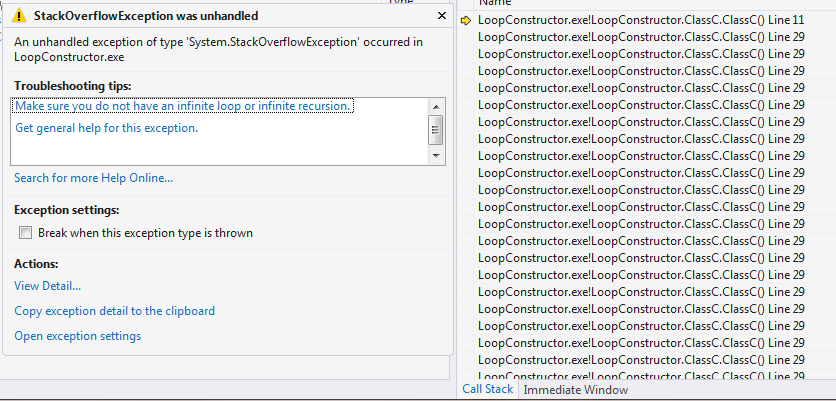


### 递归引用自身，堆栈溢出

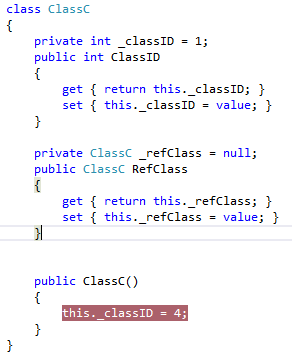
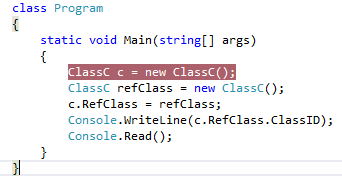
 



### 引用自身的第二种写法，仍然堆栈溢出

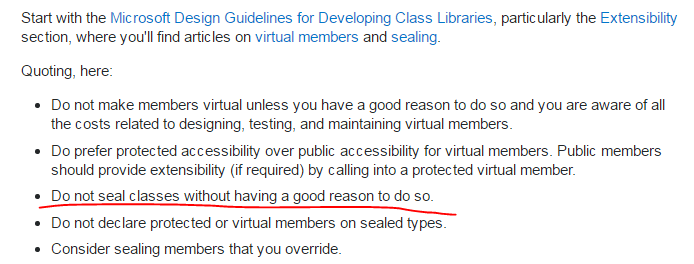
### 引用自身的正确写法

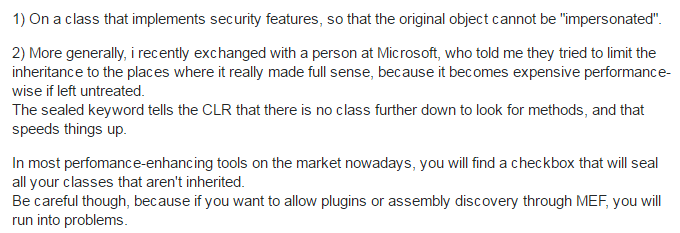
## Sealed 有什么用： Design 问题

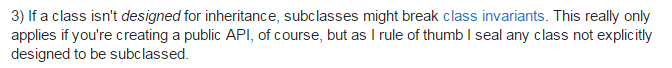
The explicit purpose is to prevent a class from being inherited. If I make a class Car that inherits from class Giraffe I'm saying that cars are giraffes and that may have implications the writer of Giraffe never coded for resulting in system instability. I'm not saying someone can't offer an alternative implementation, they just can't paste it over an existing one that wasn't designed to account for that

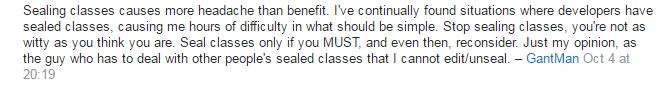
1. 这时一个类设计时的问题
2. 根据msdn的建议，能不用seal 就不用



第三： seal 对于类库的developer 可能更需要留意下



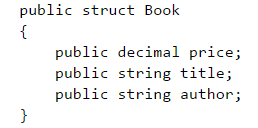




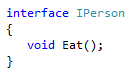
# Struct

## 定义

A struct type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory。

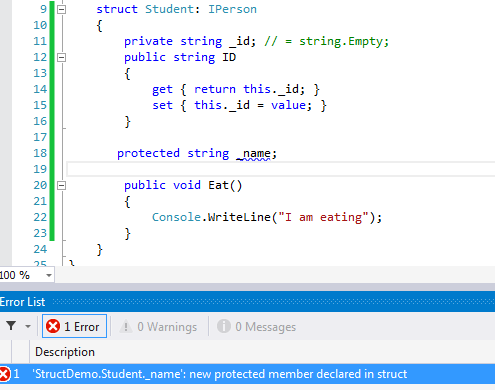


## Struct 可以实现interface

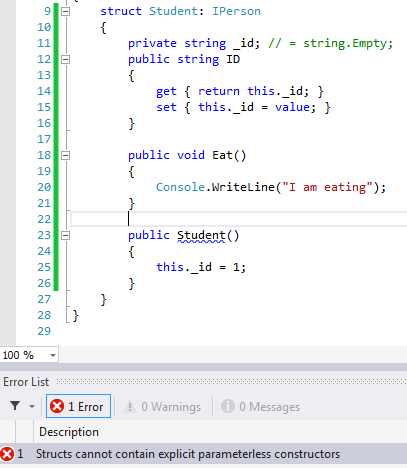
 

## Struct 不允许有Protected 的修饰符

因为Struct 不允许相互继承，所以Protected 也就没有了意义。不过private，public 没问题。

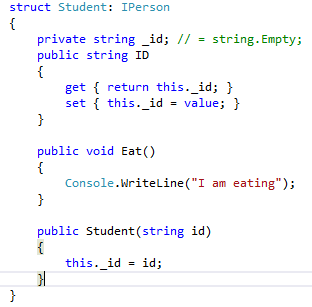


## Struct 不能显式声明无参的构造函数

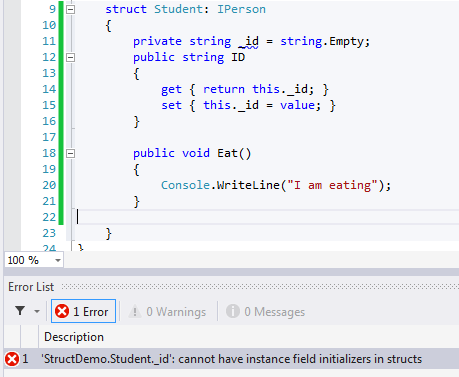


编译器保证struct 必定有一个无参的构造函数（应该是当不使用new 关键字生成struct时调用的）。在这个构造函数里，所有的值类型被初始化为默认值，所有的引用类型被初始化为null。

带参数的构造函数是允许的

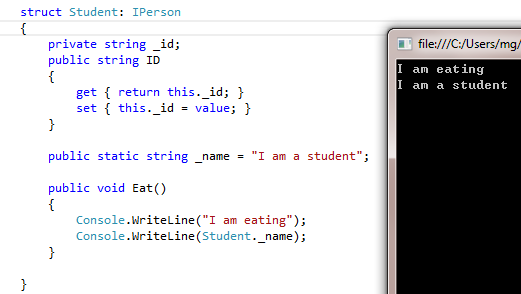


## Struct 不允许实例字段的初始化，static 字段可以



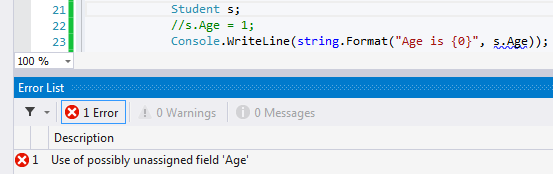
所有的初始化都在默认构造函数里自动做

the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to null. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

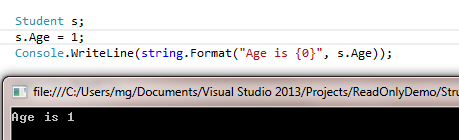


## Struct 可以不使用new ，但作为局部变量时必须先初始化

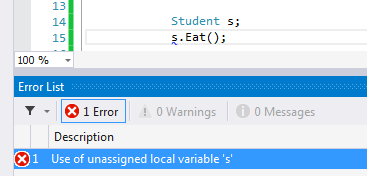
否则会报未初始化错误，虽然其实编译器会调用struct的默认构造函数进行初始化。



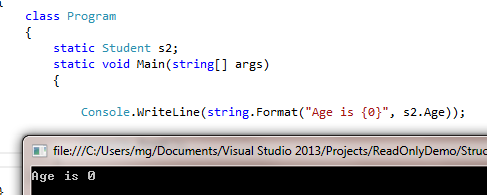
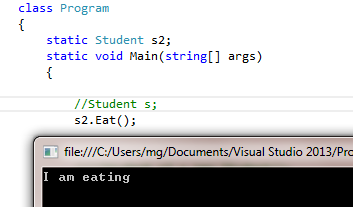
在使用时先赋一下值就可以了



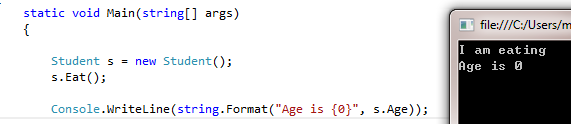
也不允许调用方法， 此时只能用new 关键字了



## Struct作为全局变量时，编译器允许不初始化就是用

## 使用new 关键字生成的struct， 编译器认为其已经初始化过了



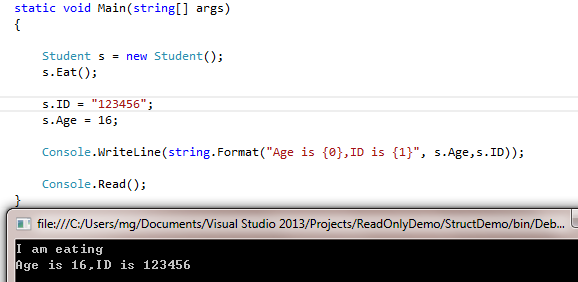
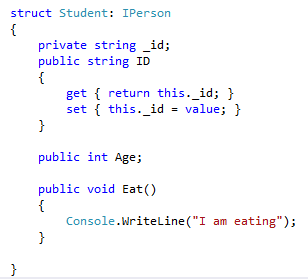
## 不管用不用new， struct 作为值类型都是生成在栈上

当然这是单独使用struct的情况。

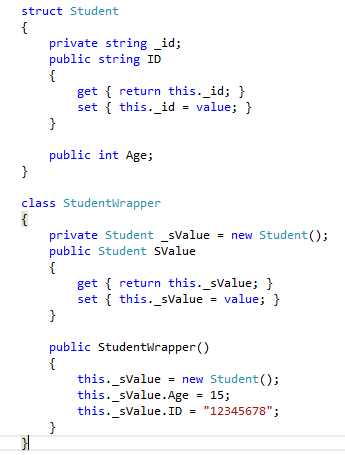
如果struct 作为别的一个引用类型的字段，那么它的值是生成在堆上的。

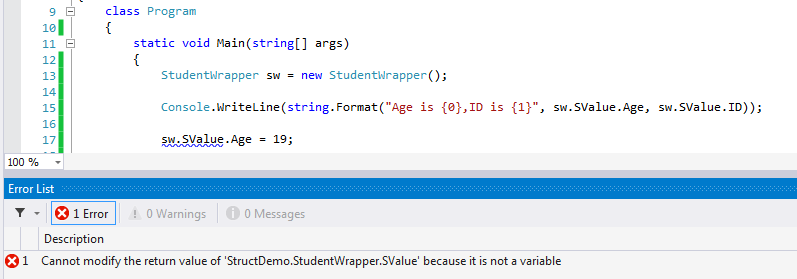
## Struct 作为引用对象的filed 时，struct 各字段不能单独赋值

当struct 单独使用时，其每个字段是可以单独赋值的。

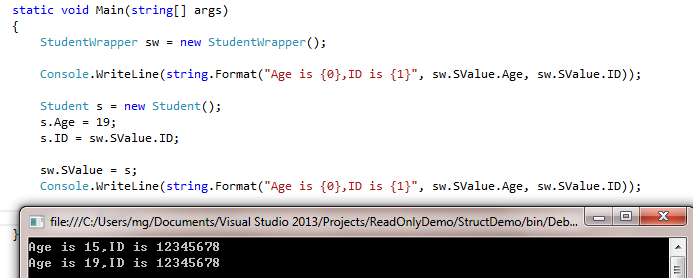
 

但是当struct 作为引用对象的field 时，就不能给struct的各个字段单位赋值了。 这是由于.net 的boxing-unboxing机制造成的。由于student是一个值类型，sw.SValue是在栈上复制了一份sw 的sValue的完整拷贝。此时sw.SValue 和 sw在堆中的sValue 的值已经完全没有关系了。直接读是没问题的，但当试图修改时，显然修改不能体现在堆上，所以编译器在编译阶段就直接出错了。



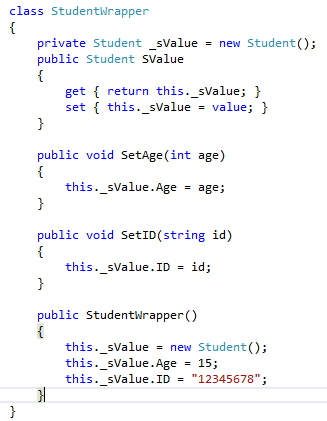


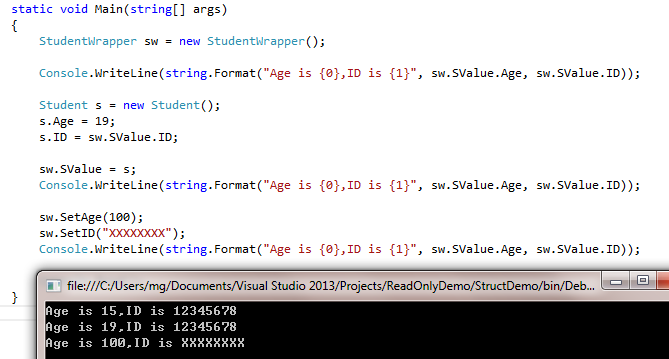
解决方案一： 整体赋值



解决方案二： 使用set 函数

根据编译起的错误可知，报的错误是StudentWrapper.SValue 的返回值不是一个变量所以不能更改，那我不使用这个返回值不就行了？ 而在StudentWrapper的构造函数里，我们看到是可以对SValue 的各个field 单独赋值的。所以我们可以考虑使用类似set 函数的方法





## Class 和 struct的区别

### 从使用场景的角度

Structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead.（这是微软的建议：https://msdn.microsoft.com/en-us/library/ah19swz4.aspx）

struct 有它自己的使用场景，如果你把它当成了class 来用，那为什么不一开始就定义成class得了

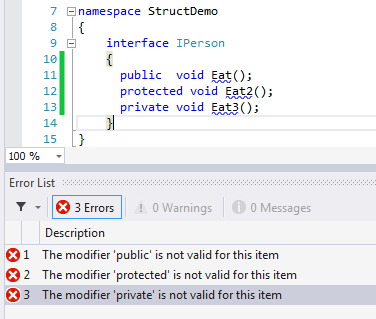
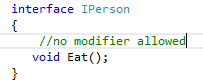
### 从语法的角度

1. Class 必须要使用new 关键字生成实例。 Struct 则可以使用new 也可以不用，就如同定义一个int 类型一样
2. Class 可以显式声明不带参数的构造函数，Struct 不可以。
3. Class 可以继承别的Class，Struct 不支持彼此间的继承

# Interface

## Interface 中的方法都不需要加public 关键字，也不能加

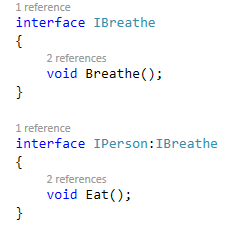
事实上interface 中的方法不允许加任何的修饰符。只能用默认的就是public

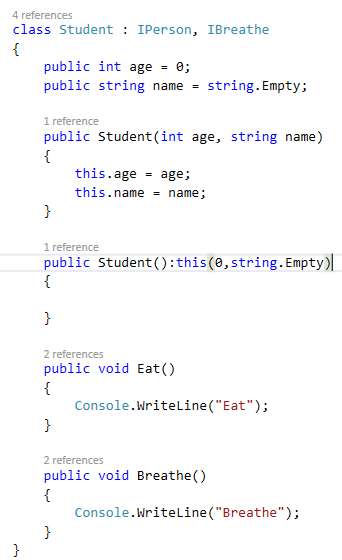
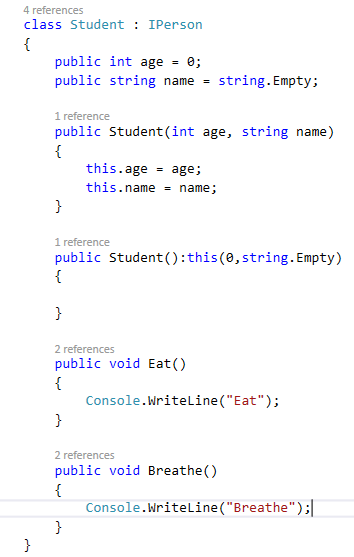
 

## 对于 接口继承的情况， 实现类可以只写子接口，也可以把子接口和接口都写上

一下两种写法编译器都是支持的，执行结果也完全一样。

至于为什么要把两个都写上，有人觉得这样更清晰。微软自己的.NET 类库里就有大量这样的写法



比如





网上也有人问，不过也没有太权威的说法

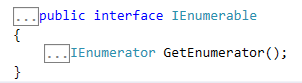
[Why collections classes in C# (like ArrayList) inherit from multiple interfaces if one of these interfaces inherits from the remaining?](http://stackoverflow.com/questions/1023375/why-collections-classes-in-c-sharp-like-arraylist-inherit-from-multiple-interf)

[Why ArrayList implement IList, ICollection, IEnumerable?](http://stackoverflow.com/questions/1164757/why-arraylist-implement-ilist-icollection-ienumerable)

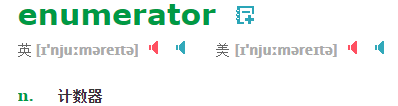
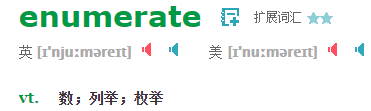
## 几个常见的接口

### IEnumerable

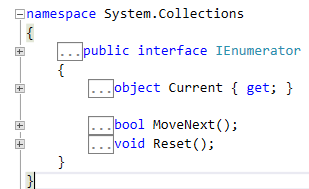
里面只有一个方法，那就是返回一个IEnumerator对象



注意其发音：

### IEnumerator



只有一个属性和两个方法，

属性返回当前对象

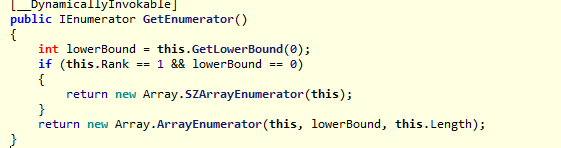
方法是移到下一个对象，重置

### 以Array类为例看IEnumerable 和IEnumerator

Array 本身实现了IEnumerable 接口



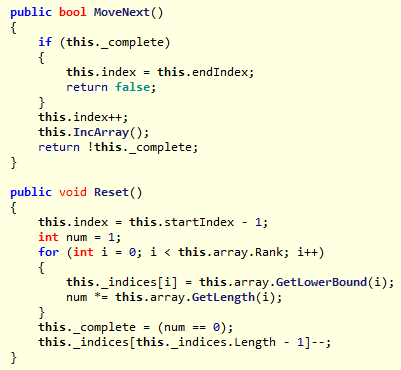
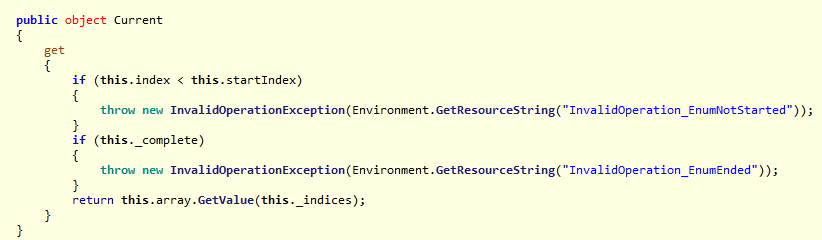
Array通过一个内部类ArrayEnumerator来实现enumerating的功能。



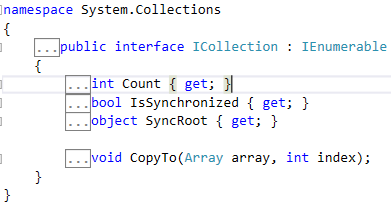
注意，因为enumerator 需要使用内部的一些数据，所以要么自己本身就做自己的enumerator，即不仅实现IEnumerable接口， 还同时实现IEnumerator接口。要么 要把this对象，或者至少是自己的一部分数据传递给enumerator。 Array用的是后一种。

ArrayEnumerator 实现了IEnumerator 接口，并提供了相应的实现



### ICollection



有三个属性和一个方法。

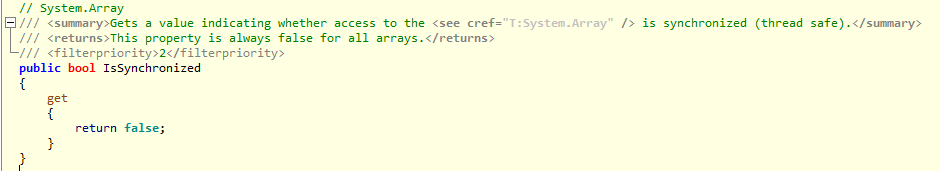
返回集合个数， 把集合拷贝到数组中。

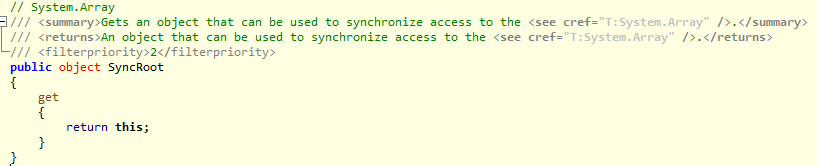
太奇怪了，居然连增，删，查的功能都没有？

### ICollection的SyncRoot 和IsSynchronized

目的是为了在多线程的访问中实现同步

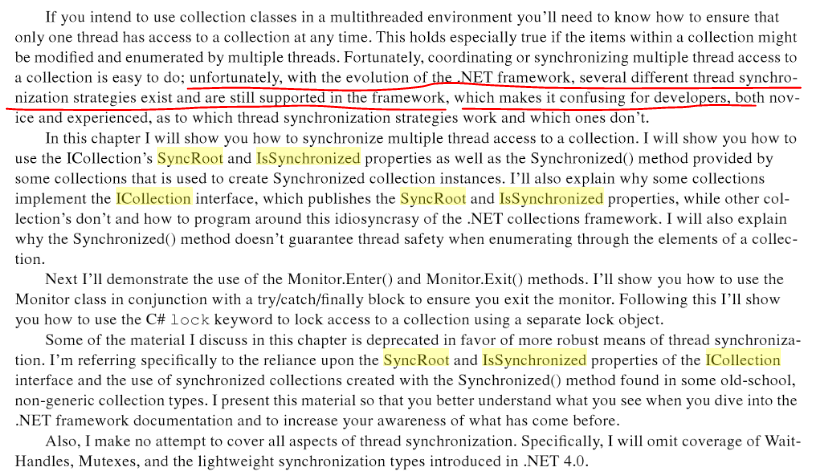
Array 类中是这样实现的



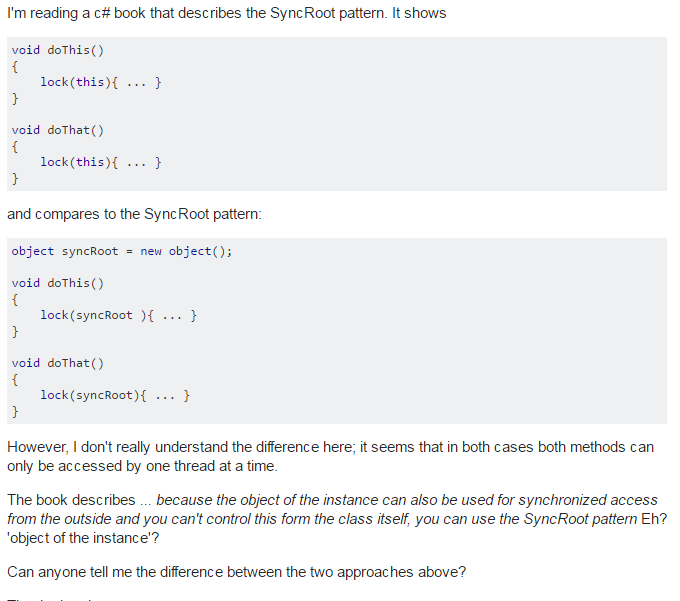


不过这个机制好像有点老了，.NET提供了好几个同步的策略，现在比较推崇的是通过lock或者Monitor.Enter

在《C# collections : a detailed presentation》中曾有过说明

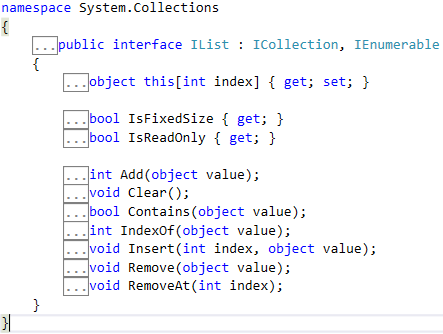


还有就是下面这个例子[What's the use of the SyncRoot pattern?](http://stackoverflow.com/questions/728896/whats-the-use-of-the-syncroot-pattern) 说明直接lock(this) 不太好，因为this对象有可能被别的什么场景lock住用来做线程同步，从而会影响到你当前这个场景的同步化。理想的是为每个场景设置一个private的对象来做同步用，不会跟别的场景相冲突。





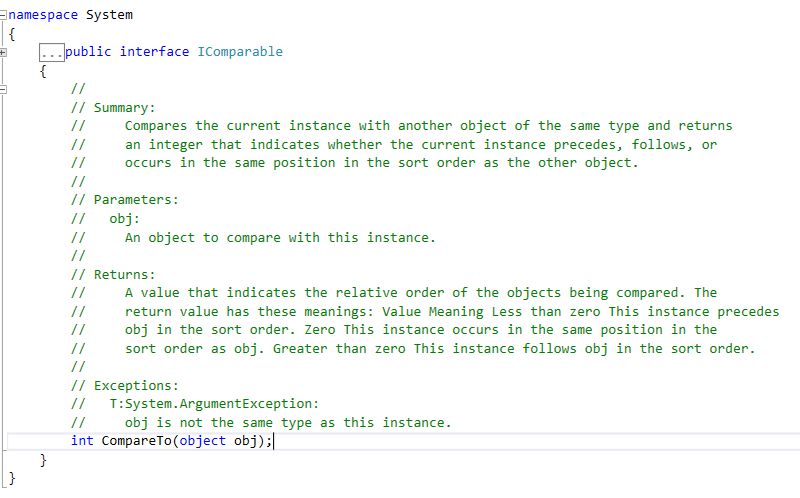
### IList



这个终于有Add，Remove，RemoveAt，Contains，IndexOf 增删查的功能了

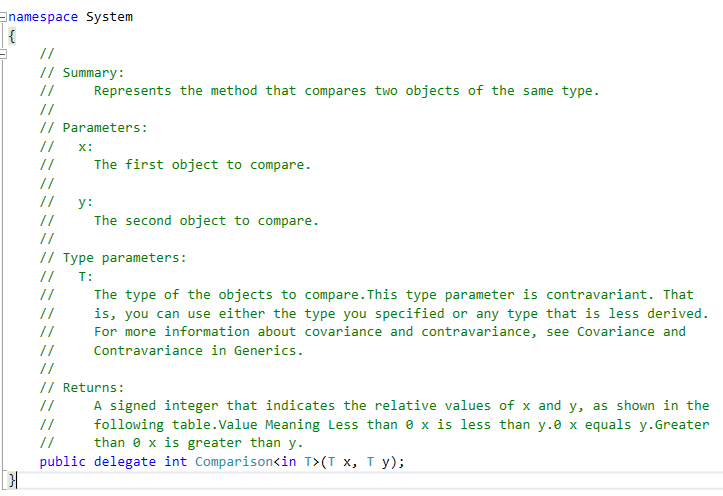
而且还定义了一个[] index 索引符

### IComparable 的CompareTo 必须返回int类型的值



里面只有一个方法，就是CompareTo。 注意这里的返回值是int 型（包含三种值，大于0， 等于0， 小于0），不能简单的用bool 型，因为转换为int之后只剩0 和1了。

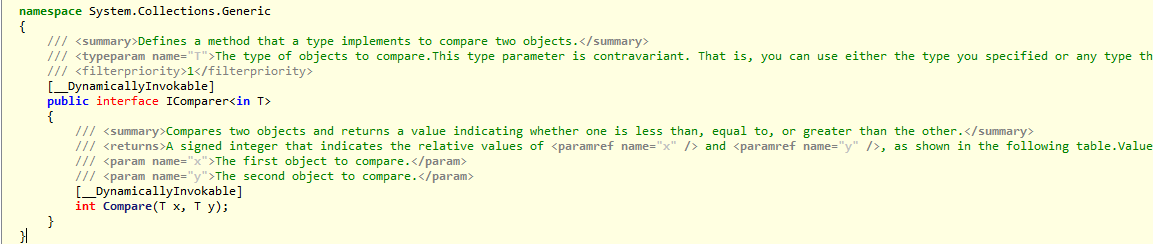
### Comparison<T> delegate



这个代理没有发现非泛型的定义

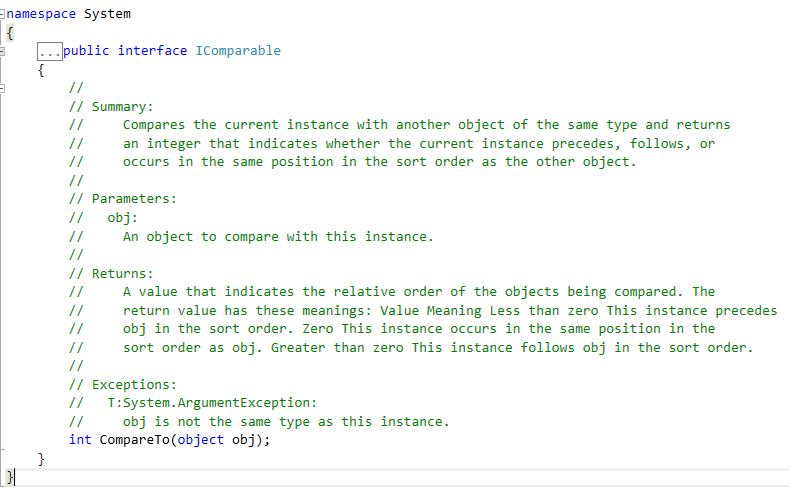
### IComparer 和IComparer<T> 接口

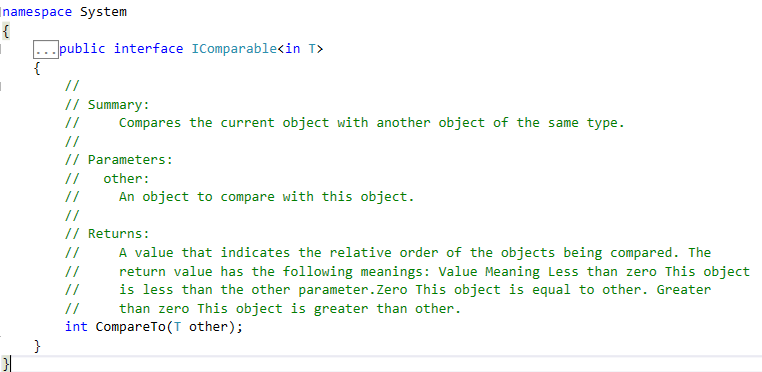




接受两个待比较的对象，所以这个接口纯粹是一个比较算法的封装，待比较的对象不需要实现该接口，往往是一个新的单纯的类，封装了一个比较算法

### IComparable 接口 和IComparable<T> 接口



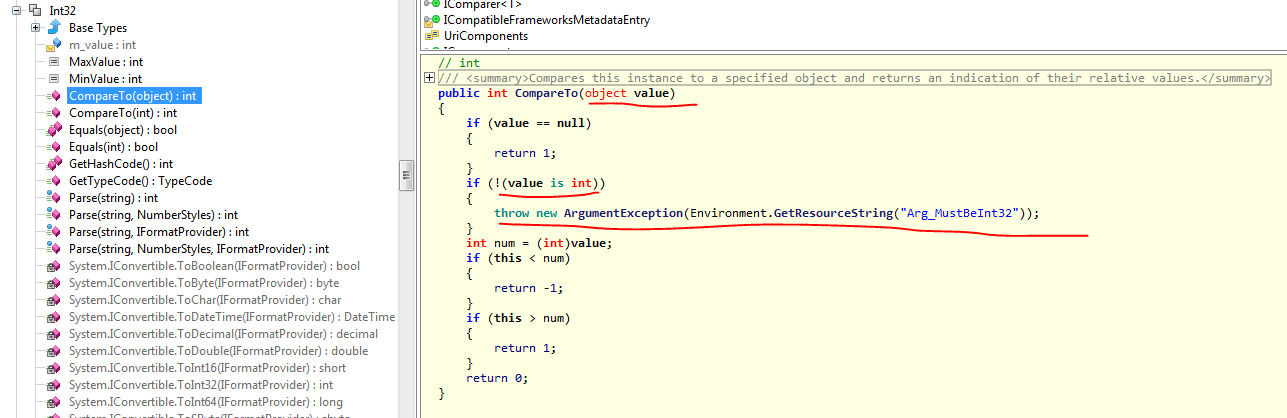


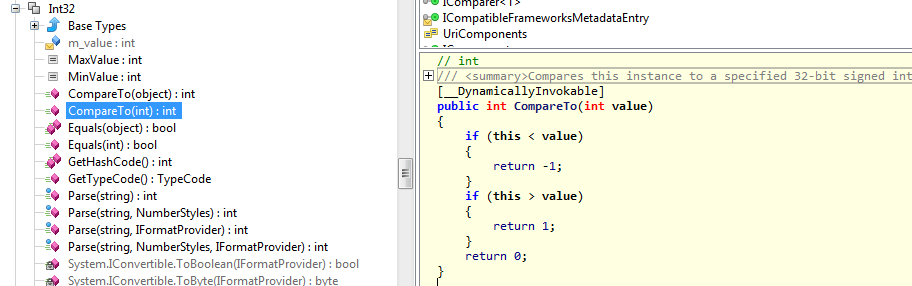
接收一个入参，所以另个比较对象肯定是自己，所以这两个接口是要比较的对象自己实现的接口

### Int32的排序接口实例

实现了IComparable 和 IComparable<int>

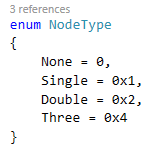


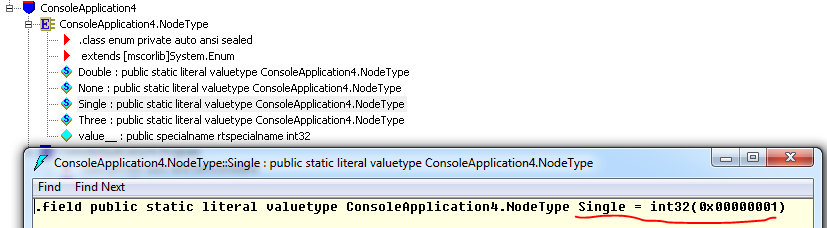




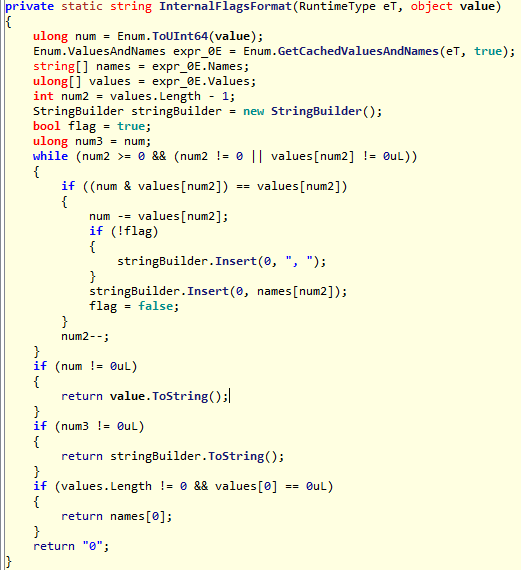
# Enum

## 枚举在IL内部被转换为int32 的值类型

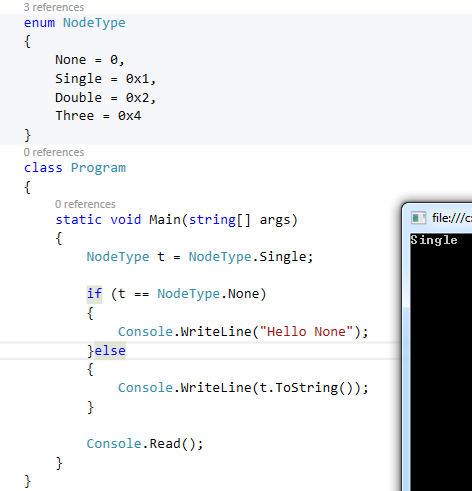




### 不过在Class Enum 内部应该用的uLong



## 枚举的Tostring() 显示自己的名称

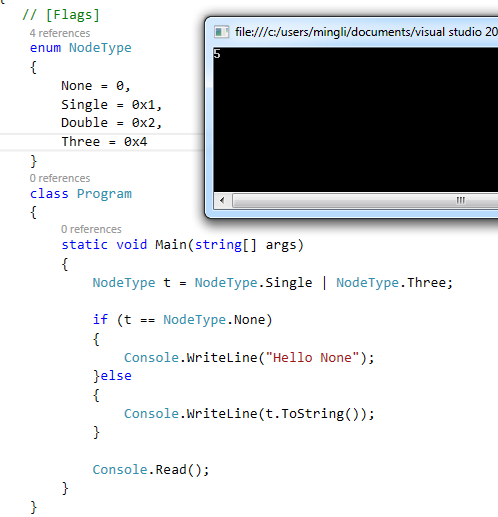
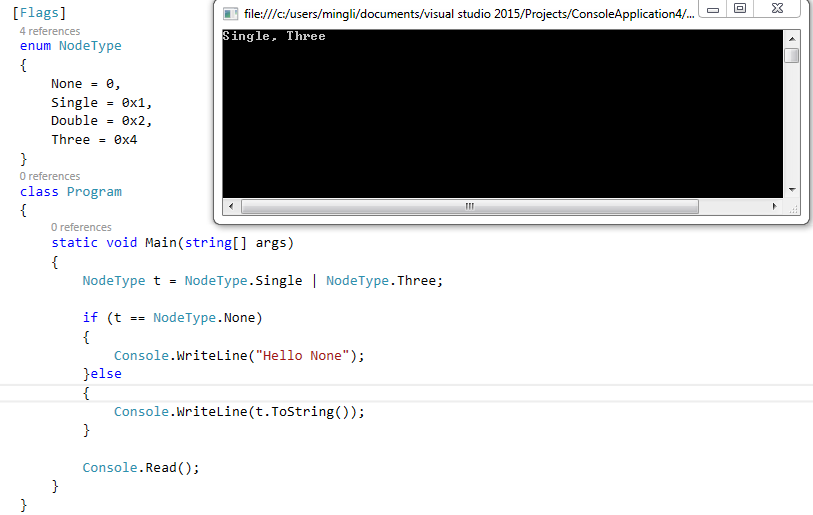


## FlagsAtribute

用来指示Enum是否可以被视为一组bit 位的flag，相互间可以与或者或。其实由于Enum 内部被实现为int32，无论加不加这个标志，都是可以与或的。加的目的是更加清晰，尤其是Tostring的时候。

对于组合的enum, 加flag的时候，显示的是名字的组合

不加flag的时候，显示的是int值， 因为找不到该值对应的name 标签

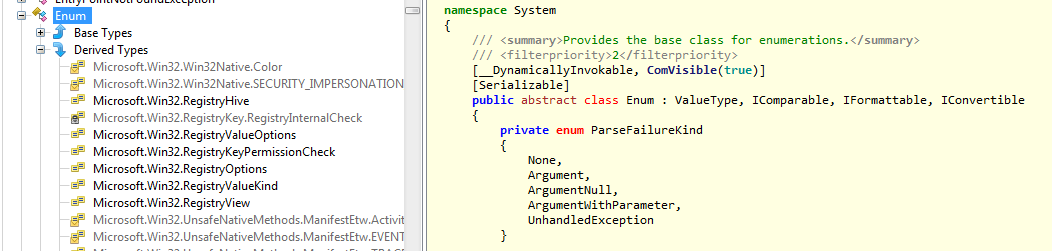
 

原因是在Enum 的tostring 里，会根据Enum是否加了Flags属性标签来采用不同的方法

## 枚举内部实现

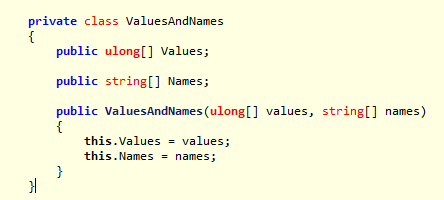
### 枚举继承自System.Enum 类

注意是一个abstract 类， 不能直接实例化，必须通过子类继承



### Values 和 names 数组

Enum 有一个values 数组和一个 string 数组来存储 值和name



### Tostring()

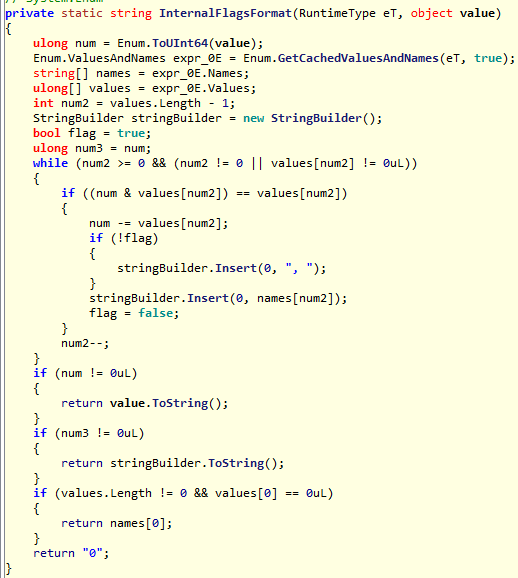
会根据是否实现flag 来用不同的方法来tostring。





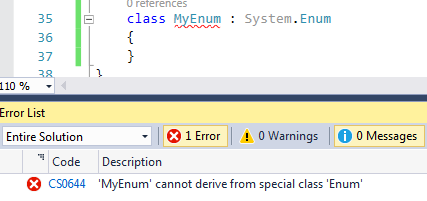
没有flag的时候，通过value 找到name，有则显示，没有则直接显示value 值

有flag的时候，会把那么一个个拼起来

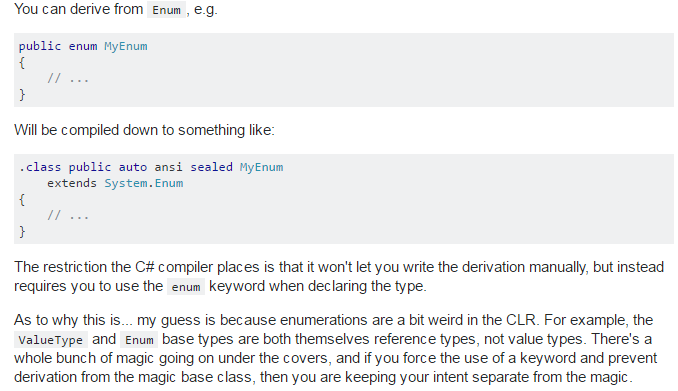


## C#编译器不允许直接继承System.Enum

System.Enum 是个抽象类，必须通过子类继承才能实例化。但是C#编译器不允许直接继承。

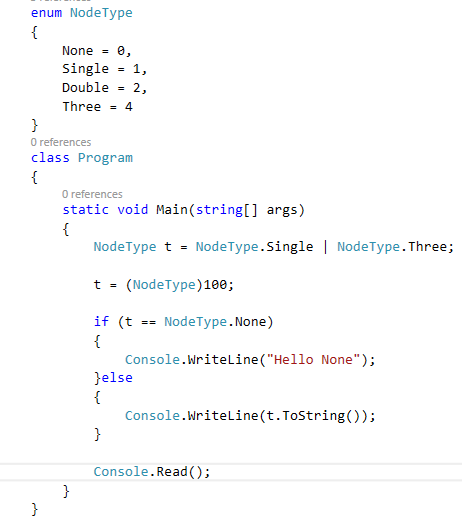


继承是编译器内部隐式做的：



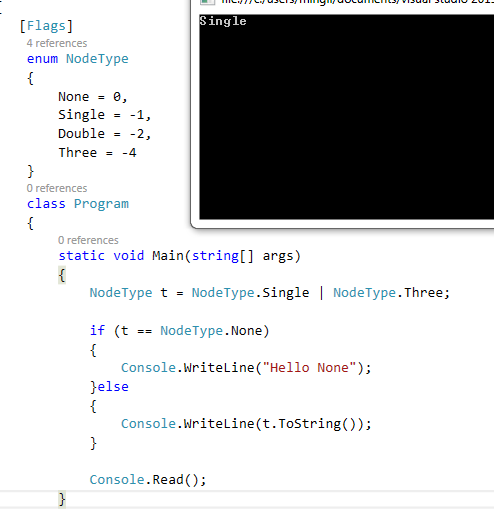
## 可以把一个不存在的value 值强制转化为Enum

编译和运行并不会报错， 这其实和几个enum 项目做与是一个效果



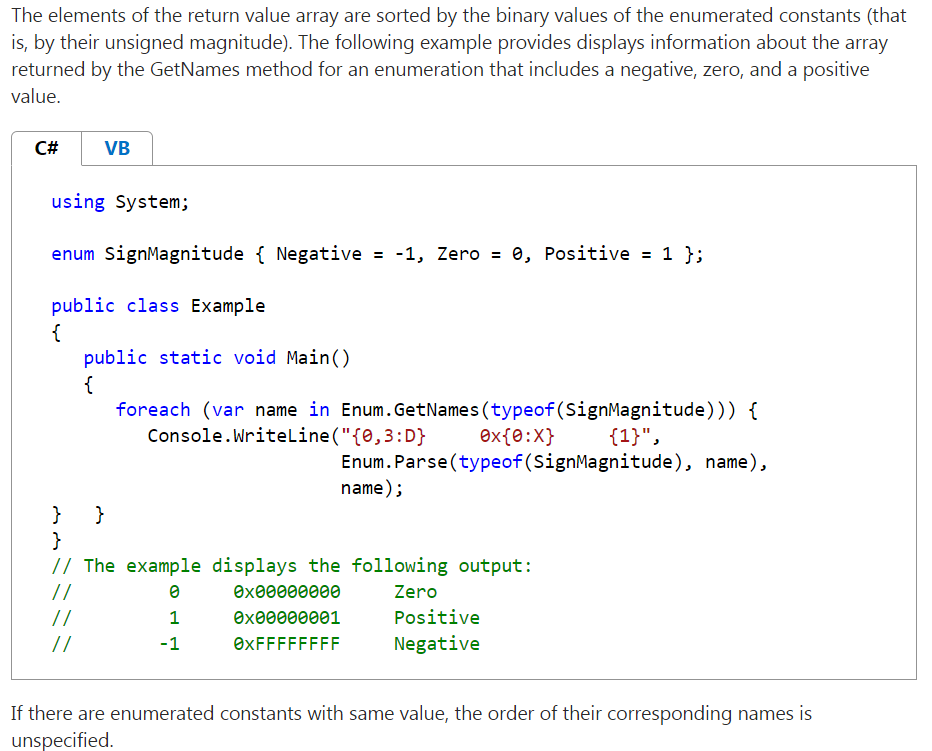
## Enum 项目可以赋负值

如果你愿意的话，虽然结果怪怪的，可能是有符号数转成无符号数的转换问题。



微软给的官方例子中就有一个把枚举项设置为-1的

<https://msdn.microsoft.com/en-us/library/system.enum.getnames.aspx>



# Collections

## C#没有List类

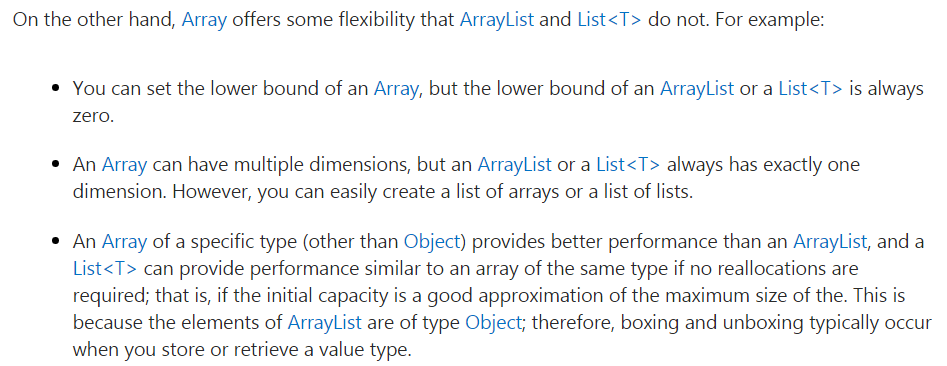
C#根本就没有List 类， 只有List<T> 泛型类，或者IList 接口



## Array, ArrayList, List<T>

ArrayList 只处理Object 类型的对象，所以可能会对值类型进行不必要的拆装箱





### 这三个其实都不能叫链表，而都应该叫数组

虽然名字里有List，但内部都是用数组来实现的，和链表LinkedList 还是有本质区别的

### Array 是抽象类

不能直接new，不过Array有几个static 方法CreateInstance 可以用来创建一个Array对象。

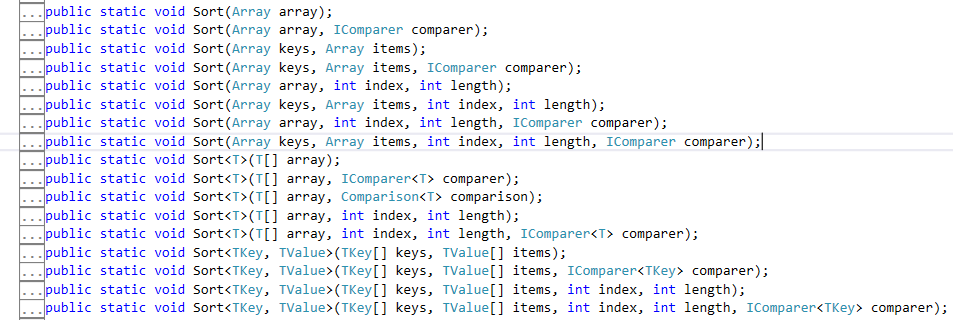


### Array的Sort 和 List 的Sort

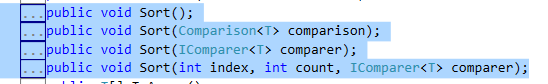
从形式上看，可以直接Sort： 要求集合对象本身提供了IComparable 接口

或者通过IComparer 或者Comparision<T> 代理来做

**Array**



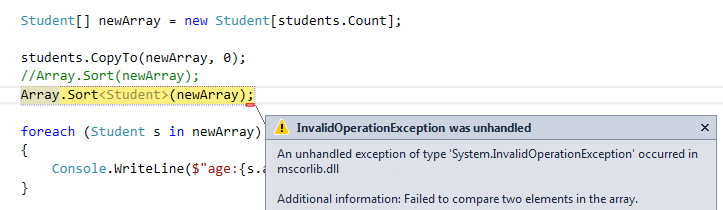
**List**



### Sort时可能会报错

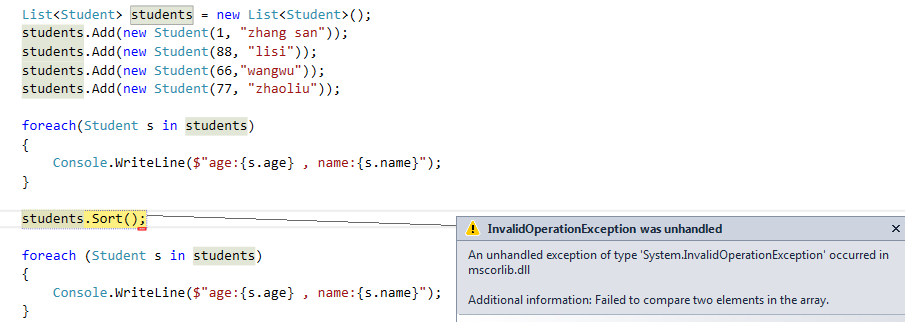
Sort时，如果对象本身没有实现IComparable接口，又没有给sort传递IComparer 或者 comparision ，那么 运行时会报错，编译没问题

**Array**



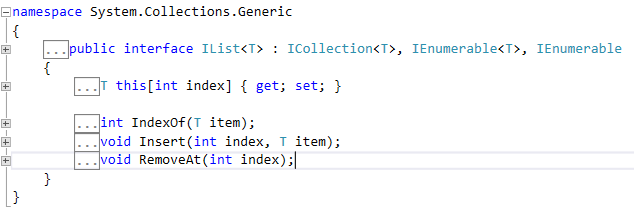


**List**



## List<T>的排序问题

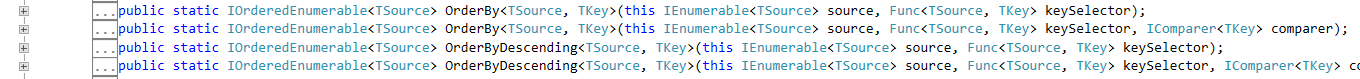
### IList并没有实现排序的接口,List<T>也没有





List<T> 对象本身也没有实现排序接口

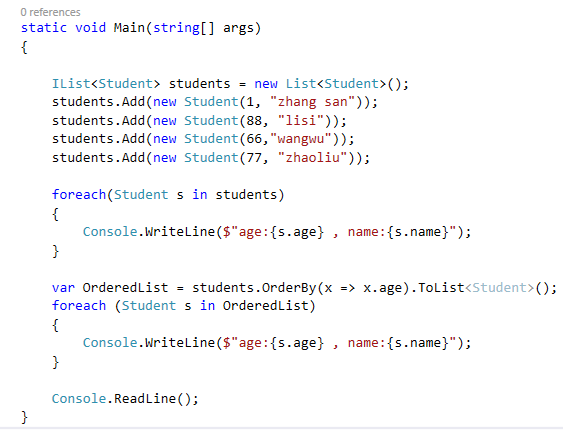
### 通过Linq的扩展方法实现List<T>的排序

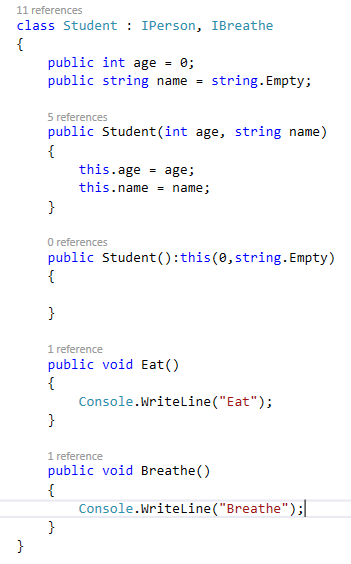


实现的有点重，好多人不推崇使用Linq

代码如下



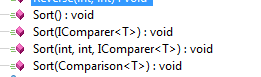




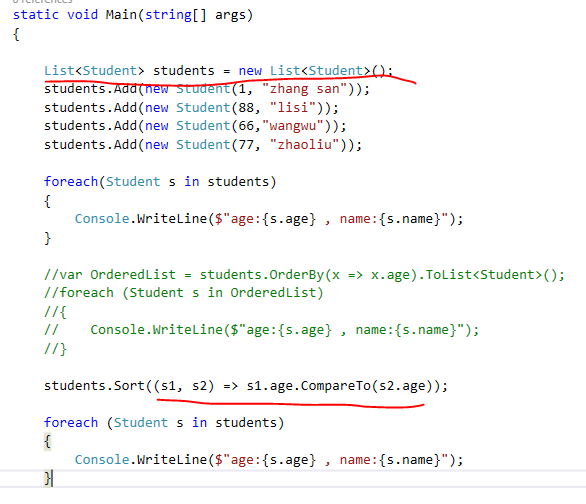
### 使用List<T>的sort方法

注意，IList<T>是没有sort方法的，只有List<T>有定义：

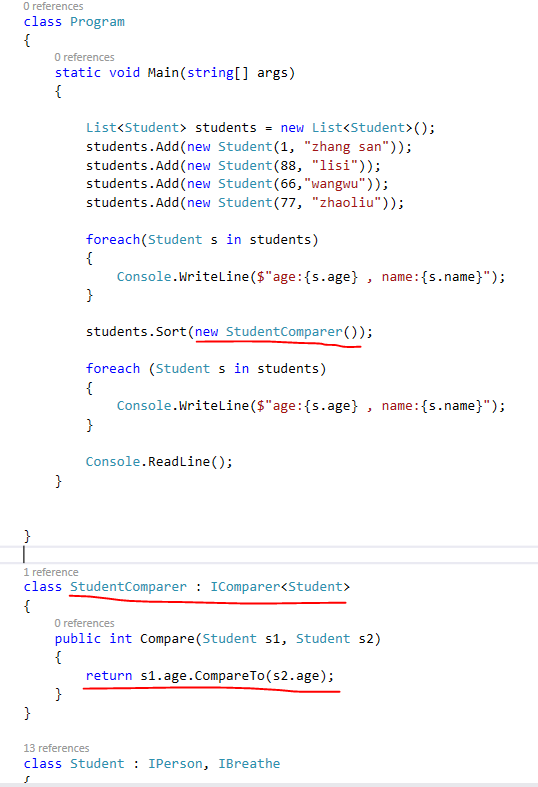
定义如下：



所以 写法上不能是再是，而必须是



或者下面这样



### 使用s1.age.CompareTo(s2.age) 而不是s1.age >= s2.age

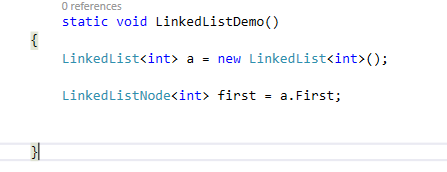
因为这里的compare 包含三种情况，大于，等于和小于。所以返回值是一个int，分别用大于0，等于0，小于0来表示

CompareTo 是正式IComparable 接口定义的方法，其返回值是int，而integer类型也实现了该方法，所以应该用s1.age.CompareTo(s2.age) 而不用s1.age >= s2.age

## LinkedList<T> 是真的用双向链表实现的链表

每个节点的类型都是LinkedListNode<T>

且没有非泛型版本



操作原语为Add 和 Remove

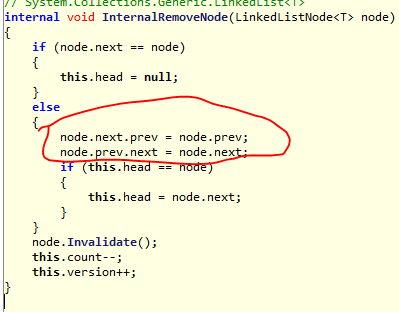
AddFirst，AddLast，RemoveFirst，RemoveLast ： 处理最头和最最尾的元素

AddAfter，AddBefore，Remove(T)

在中间add元素需要处理四个属性值

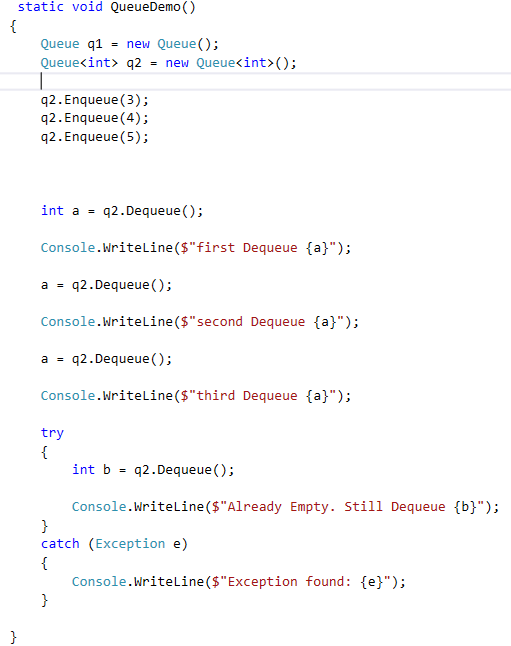
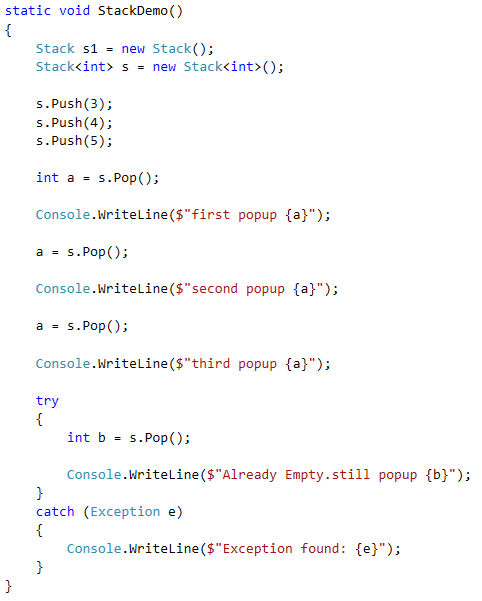


而remove元素一般只需要处理两个属性值



## Stack 和 Queue

### Stack 用的原语是Pop和Push，Queue用的是Enqueue和Dequeue



### Stack 和Queue都有泛型和非泛型两个版本

现在应该用不上非泛型的版本了

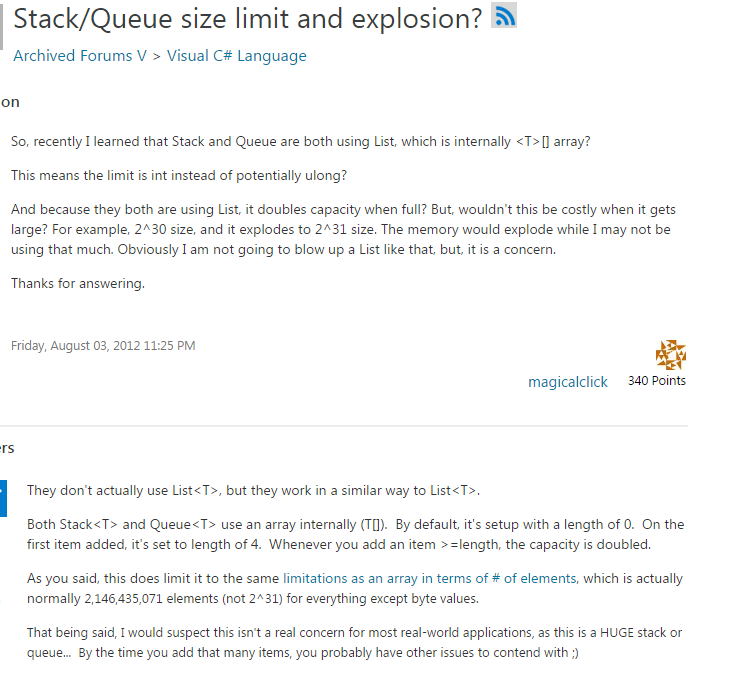
### Stack和Queue 只能设置初始size，不支持最大size

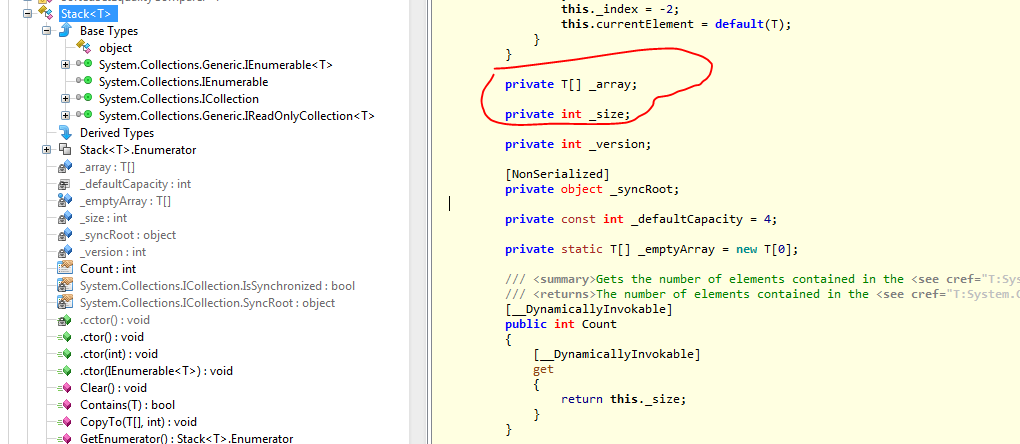
如果持续往里面加入元素，栈和队列的长度会以2的倍数递增

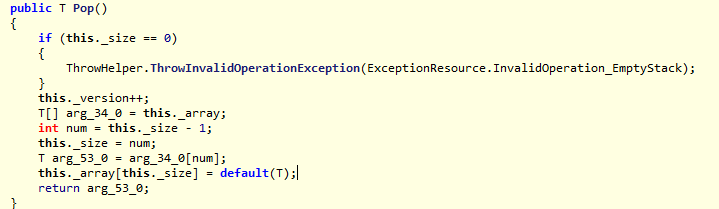
如果想限制最大容量，需要自己写一个子类



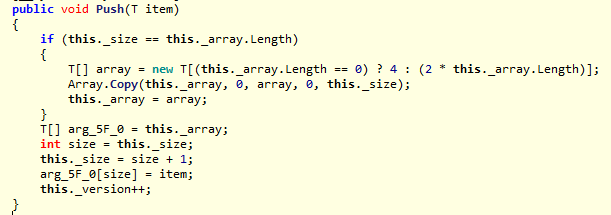
### Stack和Queue 内部用数组来实现的







### 初始长度为4

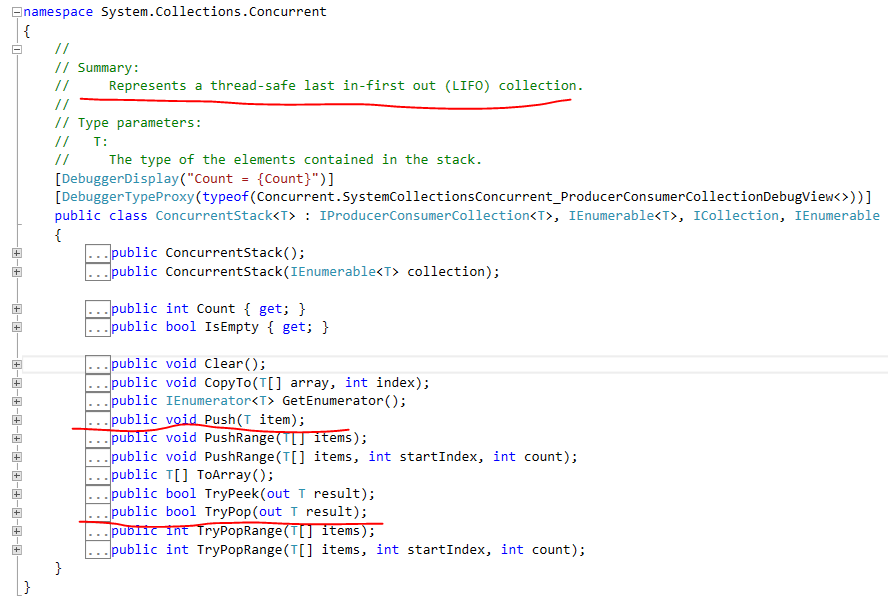


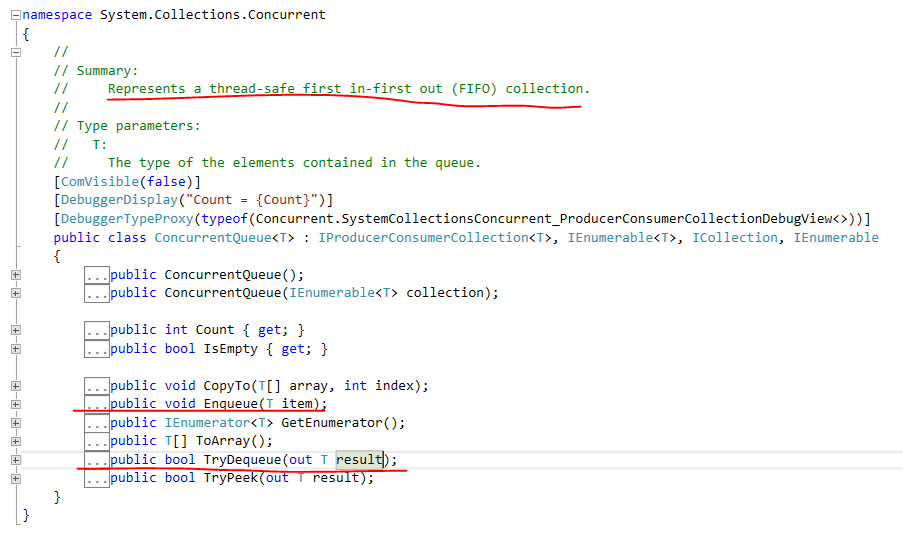
### ConcurrentStack和ConcurrentQueue

这两个只有泛型的版本，没有非泛型的版本

引用的命名空间为 using System.Collections.Concurrent;

只有Push方法，没有Pop方法了，全改为tryPop和tryDequeue了。可能是因为Push的时候不抛异常，而空栈或者空队列Pop的时候会抛异常吧。在多线程环境下，还是不抛异常好些



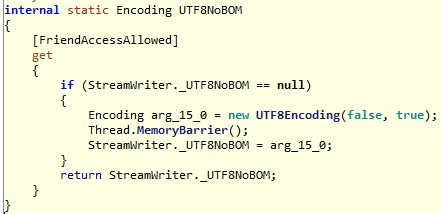


# IO

## 默认的Encoding为UTF8

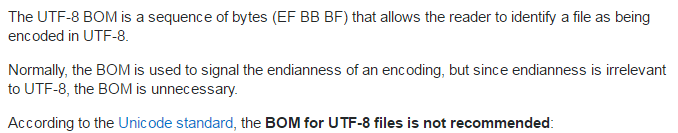
以StreamWriter 为例，默认的Encoding 是StreamWriter.UTF8NoBOM

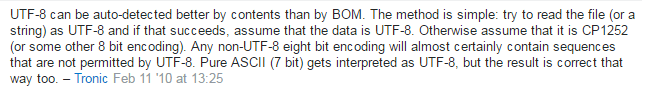
而UTF8NoBOM



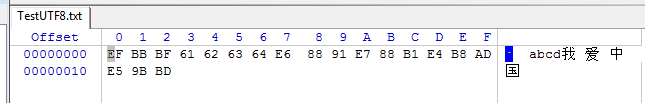
### UTF8 BOM 是什么

Byte Order Mark





下面是以UTF-8 格式存的abcd我爱中国文件。可以看到刚开始的三个字节是EFBB BF



而同一段文本存为ANSI格式的时候





存的时候说会Lost，其实是真的Lost了，因为所有的字符都被存成了？，是真正的？，字节都丢了。所以，字符和编码是两个方面。一方面，字符的编码是什么，另一方面，每个编码格式都有自己的字符集。不在这个字符集内的字符是无法用这种编码格式编码的，那就会用一个自己字符集内的有效的字符的编码来代替那个无法编码的字符，所以信息丢失的时候是不可逆的

## 默认的BufferSize 为1024

仍然以StreamWriter 为例





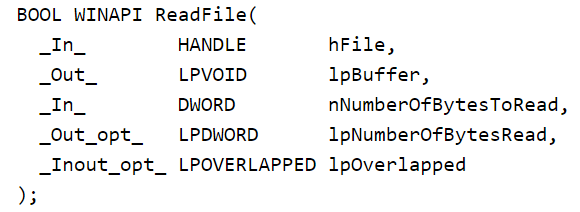
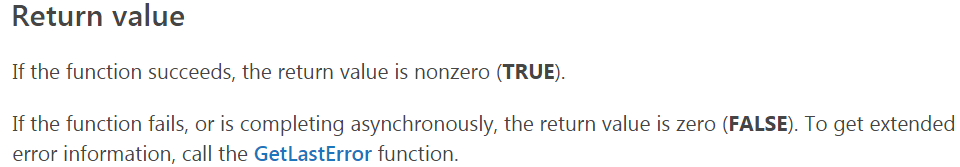
## End of File

### ReadFile 遇到end 时返回0 而不是-1

C# 中所有FileStream的Read 最后都其实是转换为对Win32的ReadFile API的调用



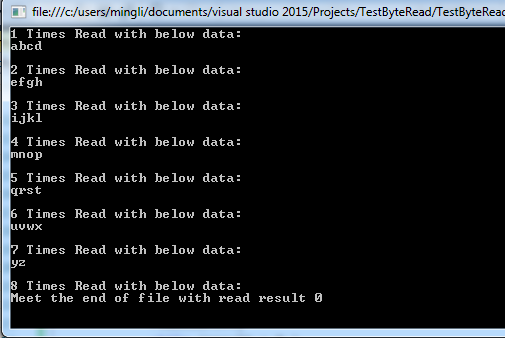
而在Win32定义中，ReadFile 只会返回False（0）和True（NonZero,其实是读到的字节数）

### End of File 总是单独读的，只有所有字节读完了才会遇到end of file

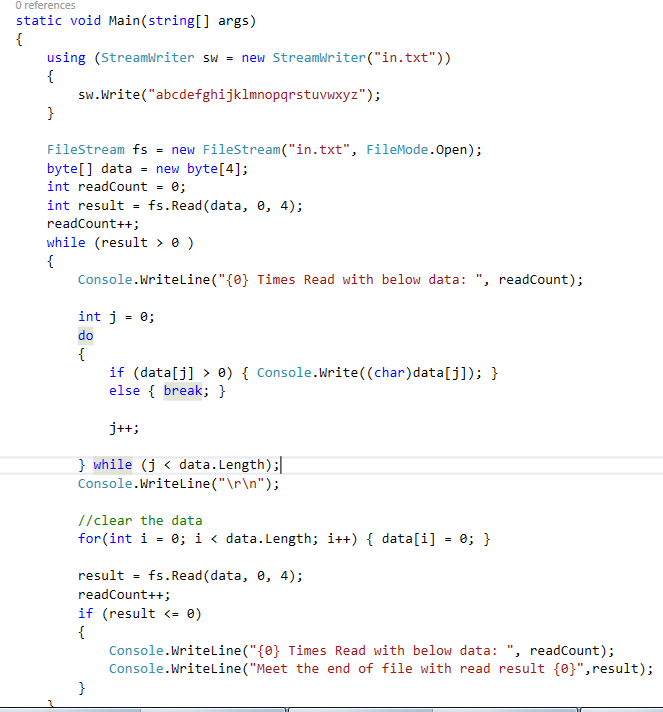
[end of file.cs](IO/end%20of%20file.cs)

以读26个英文字母为例，如果一次读4个字符，一共需要读8次，第7次其实只返回两个字符。第八次读时会遇到end of file



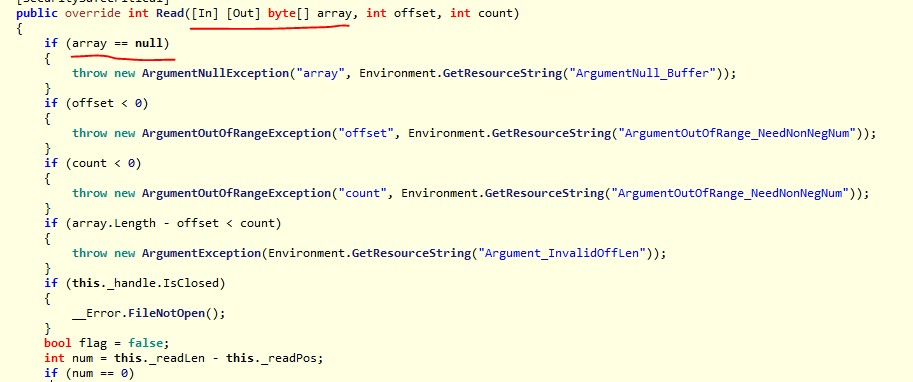
而不是第7次读的时候，读两个字符，然后遇到end of file

End of file 绝对不会和有效的字符混在一块返回。所以，要么读到字符，哪怕是一个byte，要么遇到end of file



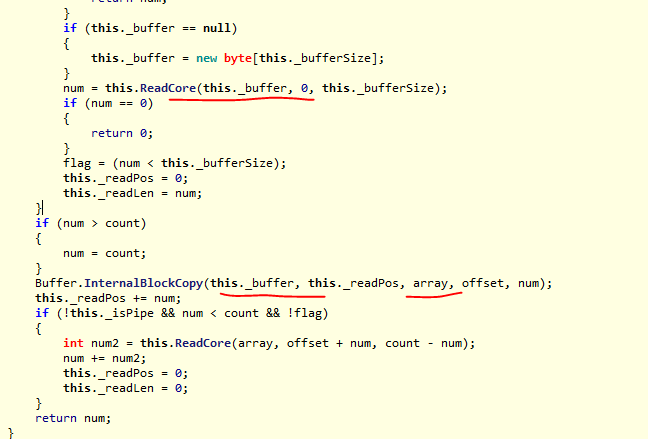
## Read 的时候，Byte[] 即使入参，也是出参

所以一定要初始化

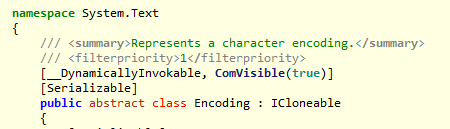


## Read 自己有一个Buffer

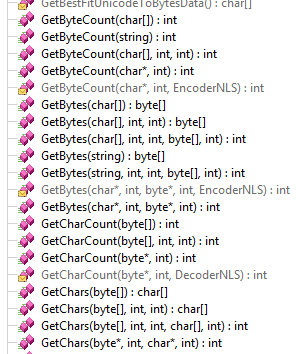
以FileStream的Read 为例，其实有一种场景是先把内容read 到自己的Buffer内部，然后再Copy到传递进来的Byte[]中，而不是一开始就直接使用传递进来的Byte[]

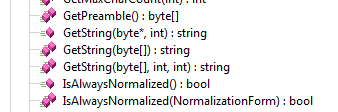


## Encoding 是类而不是枚举



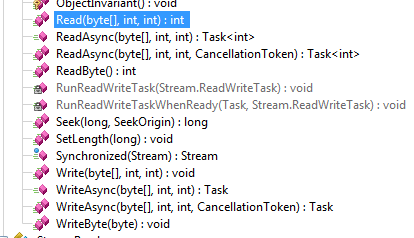
### 主要作用是把字符char， 字符串string 和byte 数组间转换



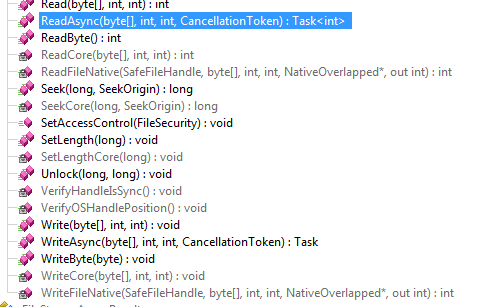


### Stream 只能read，write byte 数组

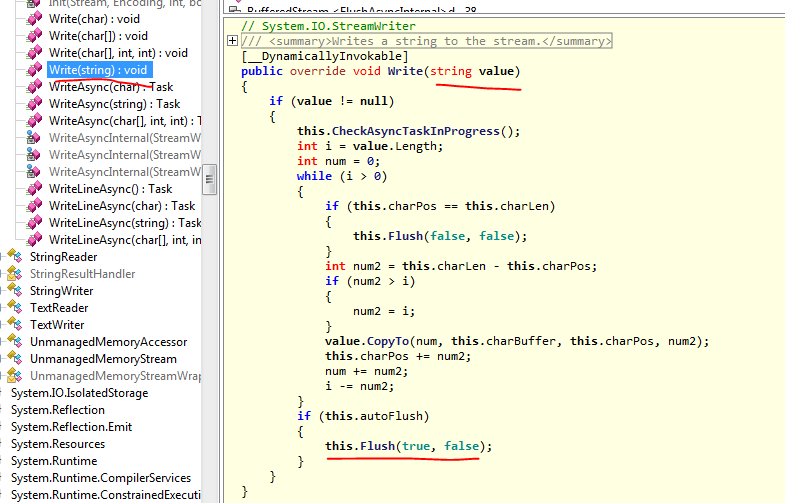
Stream 类是只能read ，write byte 数组的

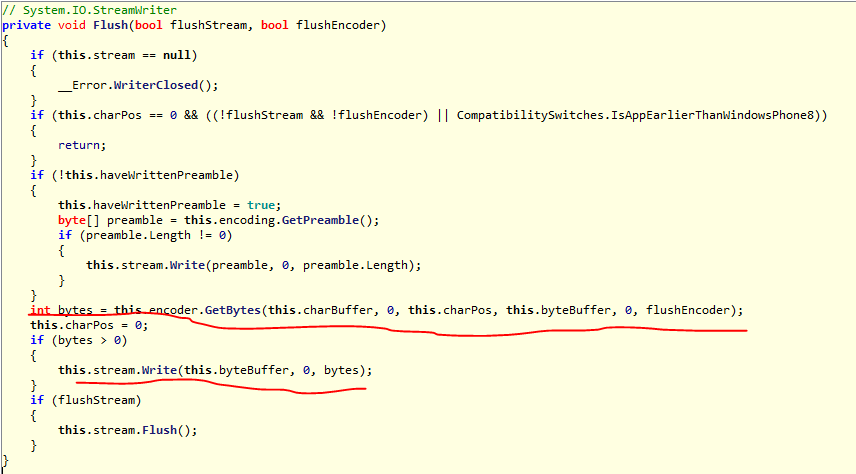


其子类如FileSteam 也是只能处理Byte[]



所以，我们用的封装类，如StreamWriter 和StreamReader， 其能处理string类型的，那么内部一定会要经过Encoding 转化为byte数组的。所以，对于特殊的中文，一定要显示的指定其encoding





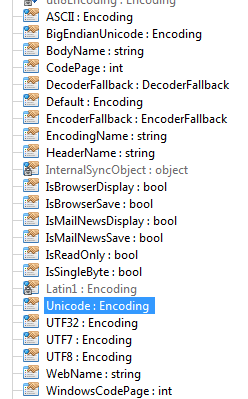
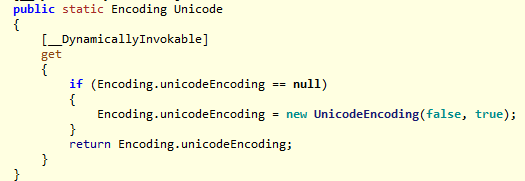
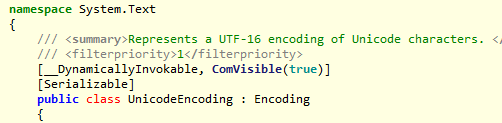
所以，我们生成streamwriter对象时，要么显示指定encoding，否则默认就用utf-8.

当然，utf-8 是可以处理中文的，不过解码的时候，也就是streamreader的时候，是要根据编码时用的来个encoder 来选择用什么来解码



### ASCII， UTF8等是Encoding类的static 成员

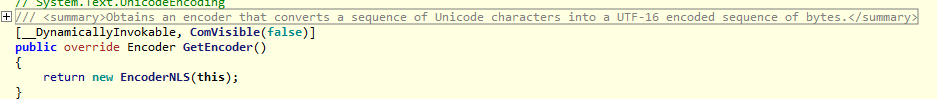
其本身也是一个Encoding 类

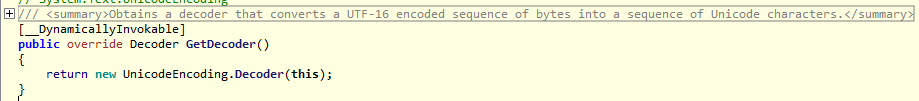
  

### Encoding 类都有一个相关的Encoder 和Decoder类

Encoder 是把字符序列转换为byte 序列，Decoder 是把Byte序列转换为字符序列





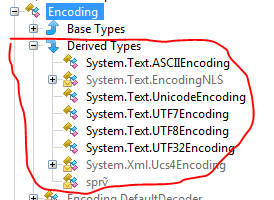


### 系统预制的Encoding 类

Encoding 是一个abstract 类，所以不能直接new

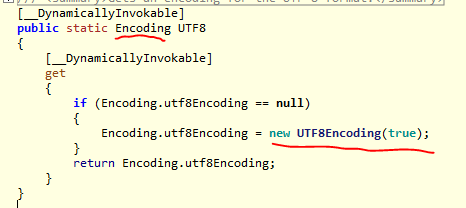


想new 的话怎么办，那只能new 它的子类了。下面这些都是可以直接new的子类



还有就是Encoding 类其实已经帮我们声明好了一些static 属性，就是这些子类的实例，可以直接拿来用，以UTF8 为例。 其实Encoding 的static 属性也是通过new 子类来实现的。

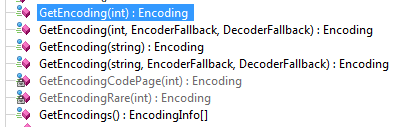
需要注意的是，Encoding 所有的static 属性返回的都是Encoding 类型的变量，所以你可以用var ，Encoding 类型的变量来接。如果想用子类类型来接，要经过类型转换

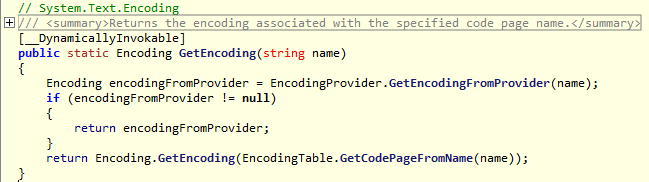


### 拿到其他的Encoding 类

那其他的一些字符集的编解码怎么办呢，就比如中文常用的GBK 或者GB2312

Encoding 还提供了一些通过codepage id 或者 code page名称来取得特定Encoding的方法



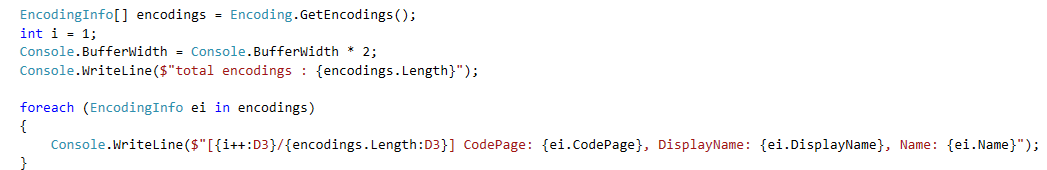


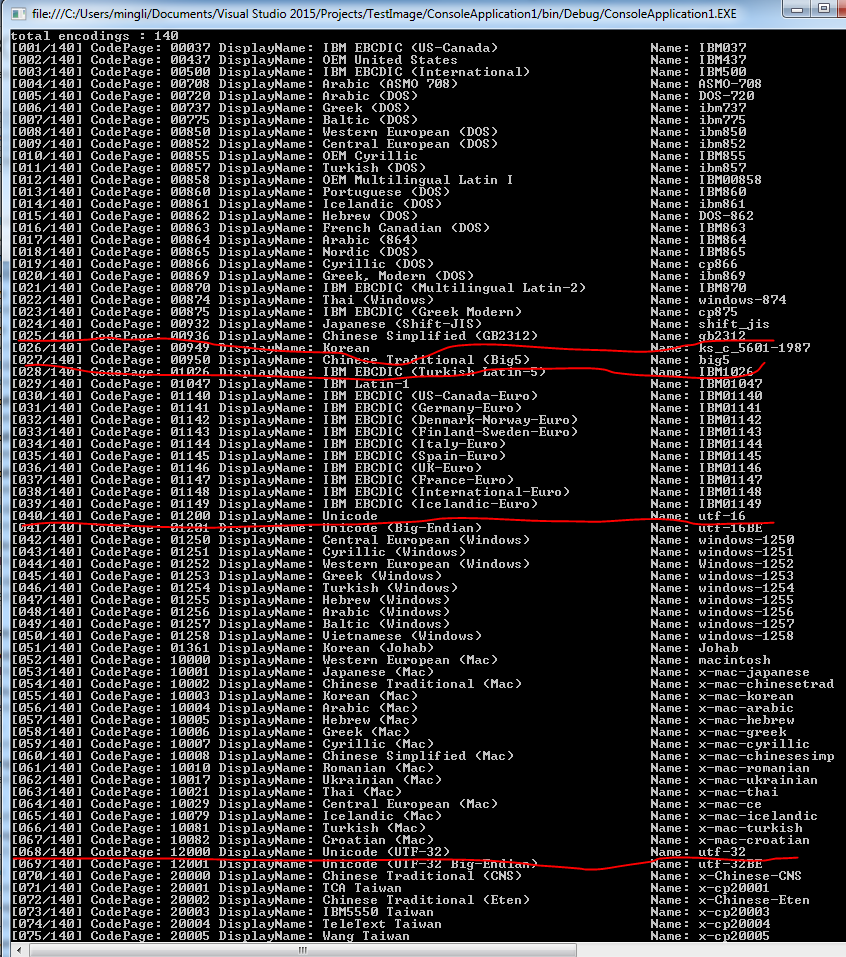
类似于

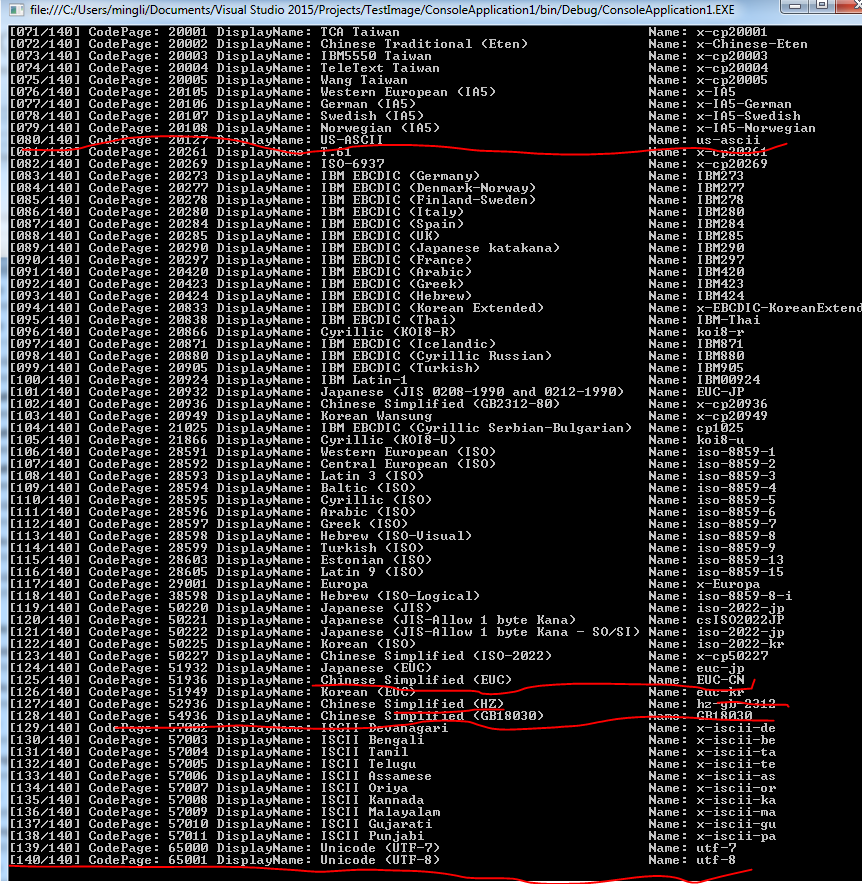




Encoding 类还提供了一个GetEncodings的方法来枚举所有支持的codepage

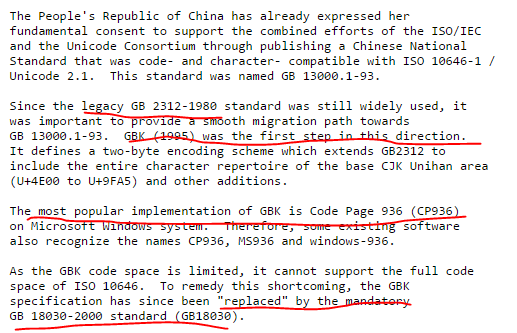






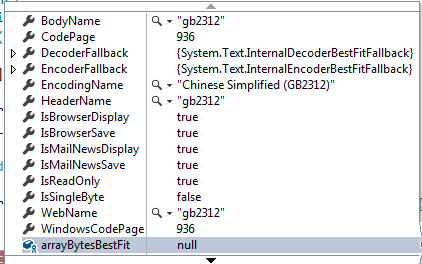
### GBK 其实用的是GB2312

上面的例子里， 我们并没有看到name 为GBK的Encoding，但是传入GBK来获取Encoding 又能够成功，原因是什么呢？GB2312 是80年的标准，95年改为GBK了，不过在windows操作系统内仍然叫GBK



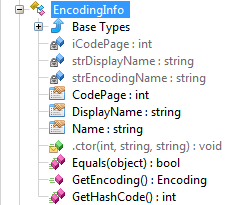
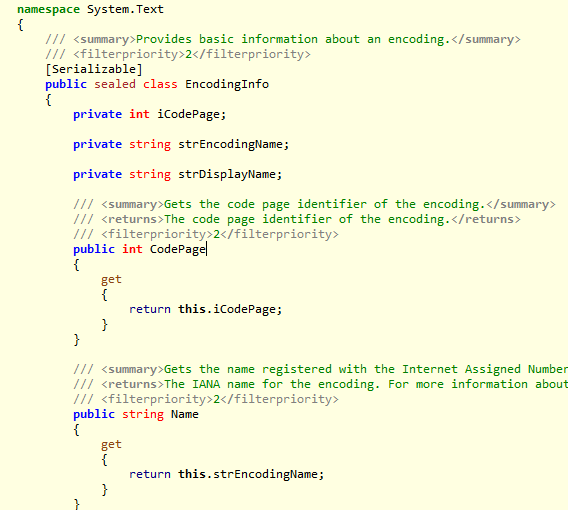


生成GBK的Encoding， 其实起属性里是gb2312



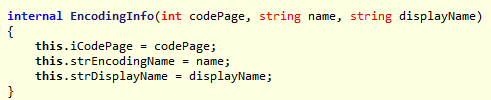
### EncodingInfo

我们在调用Encoding.GetEncodings()时，用到了EncodingInfo类。



该类也比较简单，包含三个属性，一个构造函数，和一个获取对应的Encoding对象的方法。

该类虽是public的，但是其构造函数确实internal的，无法在外面new





看来EncodingInfo的使用场景也就是通过Encoding.GetEncodings() 获取EncodingInfo，再通过EncodingInfo获取对象的Encoding 对象

# 关键类和结构

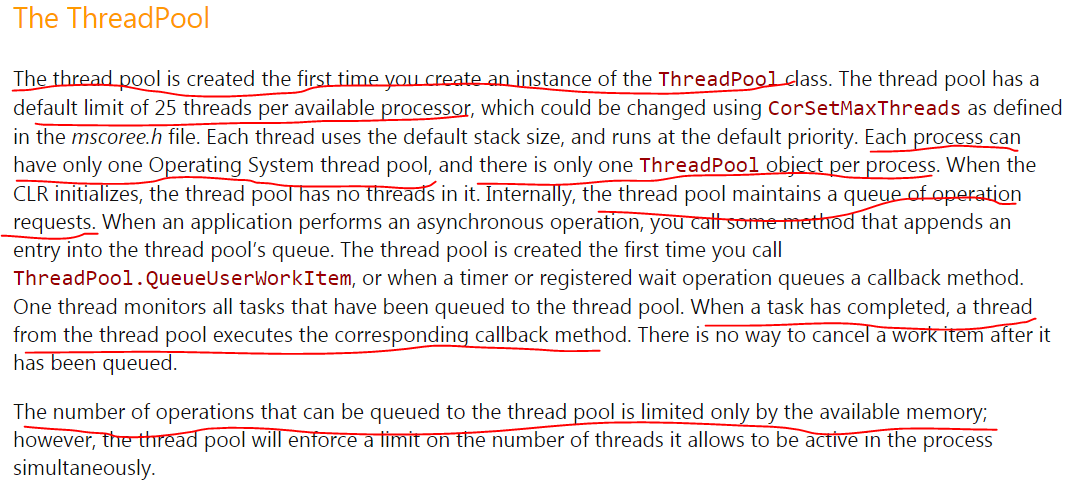
## Convert

# C# 异步编程

## Thread 和 TheadPool

### 一个进程内部只有一个ThreadPool实例

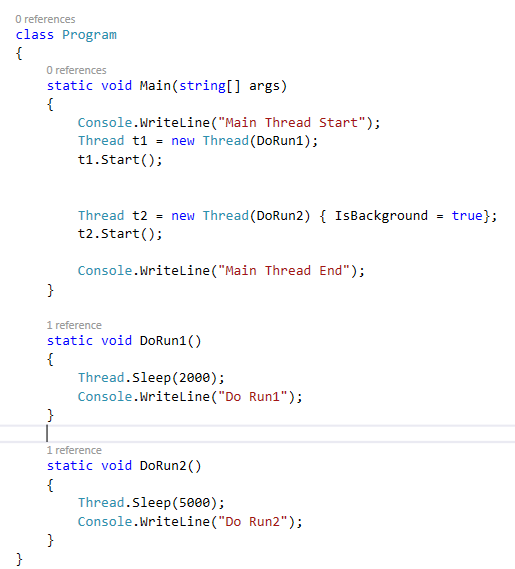
https://www.codeproject.com/articles/32374/thread-basics-and-the-clr-s-thread-pool



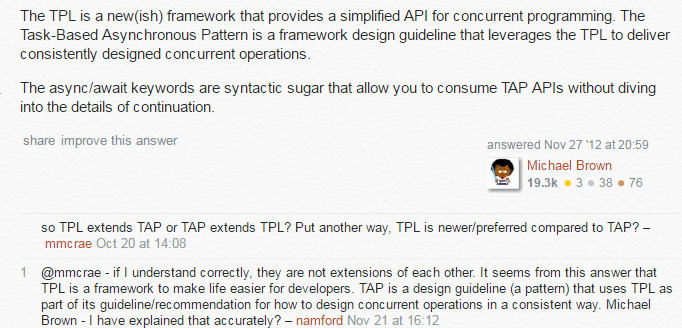
### 前台线程和后台线程

默认情况下，实例化一个Thread创建的是前台线程，只要有前台线程在运行，应用程序的进程就一直处于运行状态，以控制台应用程序为例，在Main方法中实例化一个Thread，这个Main方法就会等待Thread线程执行完毕才退出。而对于后台线程，应用程序将不考虑其是否执行完毕，只要应用程序的主线程和前台线程执行完毕就可以退出，退出后所有的后台线程将被自动终止。

下面的例子， Do Run1 可以打印出来， Do Run2 打印不出来



## TPL和TAP 区别



## TPL Task Parallel Library

### TPL的历史

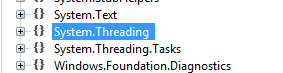
C# 4.0 时引入TPL （C#4.0 对应.net framework 4.0， vs2010）

C# 4.5 丰富了TPL API

C# 5.0 引入async 和 await 关键字 (对应 .net framework 4.5 , vs 2012)

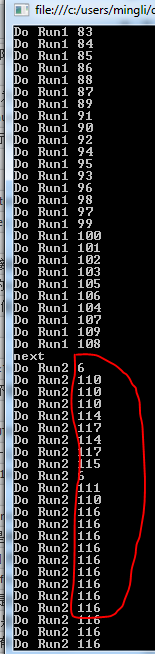
### System.Threading.Tasks

主要命名空间是System.Threading.Tasks, 在mscorlib.dll 中



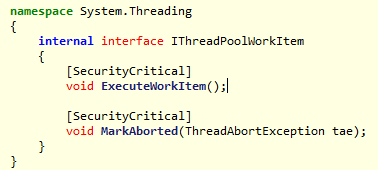
### TPL 与直接使用Thread的区别： TPL基于ThreadPool

Task底层是使用线程池的，而Thread每次实例化都会创建一个新的线程。

### Task 就实现了IThreadPoolWorkItem 接口



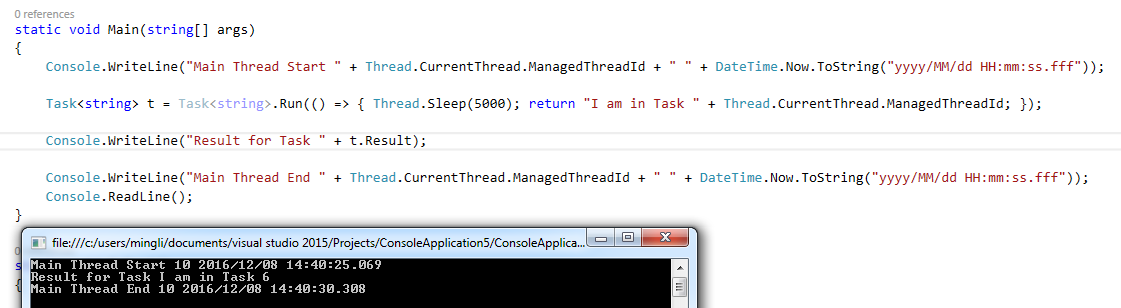


在TPL 类库中搜索ThreadPool得到的结果：

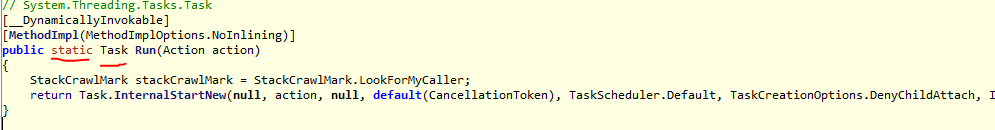


### Task<TResult> : Task

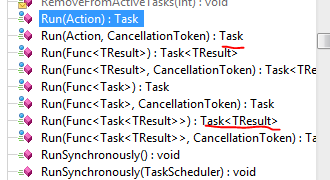
 继承自Task，这两个之间的最大不同是Task<TResult>可以有一个返回值，对result的调用会阻塞当前调用线程，知道结果返回，效果跟调用task.Wait()是一样的。



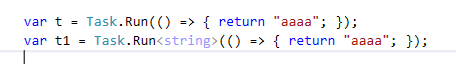
### Task 重载的几个Run方法都是Static 方法，且返回Task 类型的对象



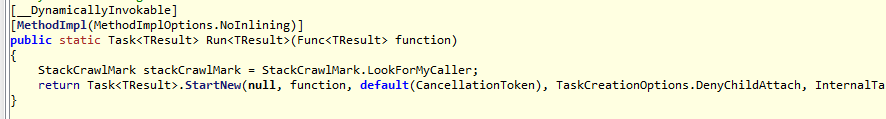
如果有返回值，当然也返回Task<TResult> 类型的对象



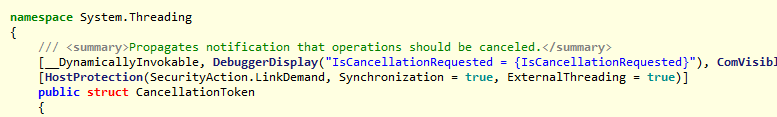
注意其实下面两种写法都支持，就是方法名称中是否包含泛型的返回类型

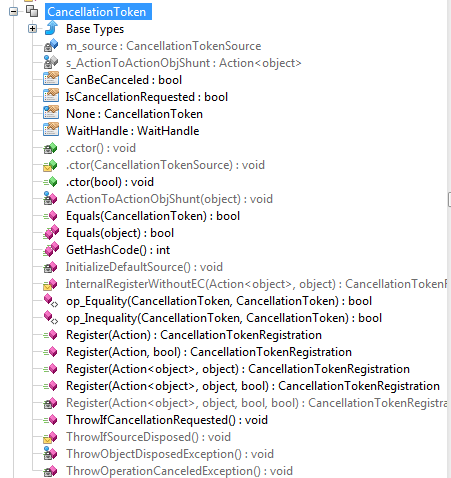


因为定义的时候是包含的



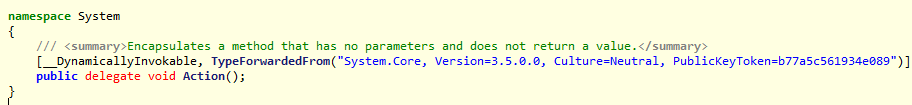
### CancellationToken 是个Struct



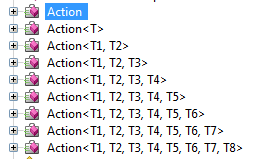


### Action 和 Func<TResult>

既没有入参，也没有返回值

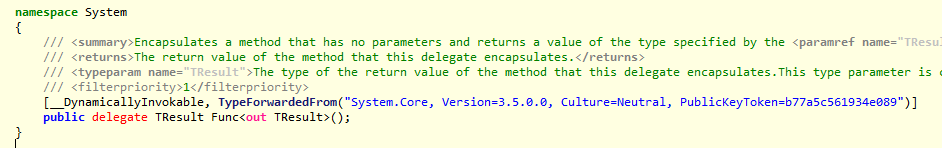


其实Action 主要想强调是没有返回值，它还是定义了好几个有入参的形式的

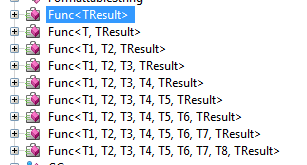




Function 则一定有返回值，不一定有入参



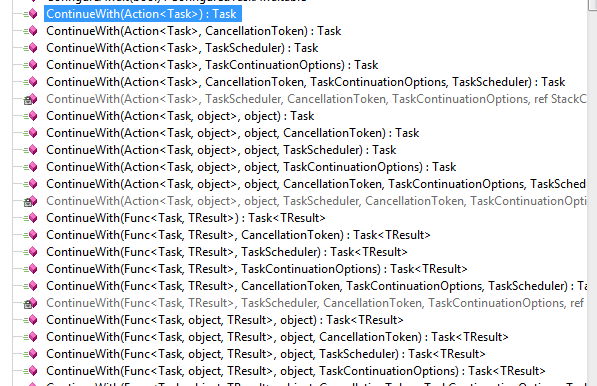
Func 也定义了多个重载方法，不仅有返回值，还包含多个入参



有返回值，又有入参的情况下，返回值类型在最后面定义



### Task 执行之后拿到它的执行结果:ContinueWith



ContinueWith 是Task的非static 方法， 同时返回值也是Task类型，这样设计可能是为了编程时的管道的那种写法吧。

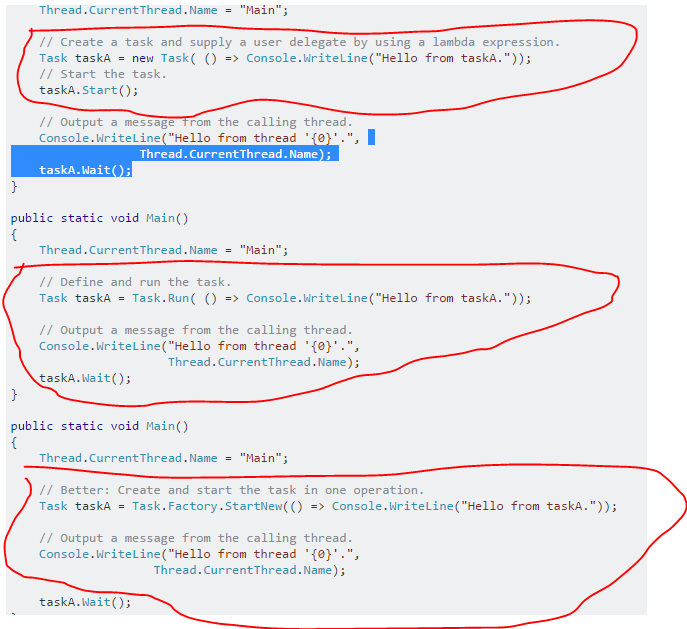
入参是一个delegate 类型， delegate的入参是Task ，也就是执行任务的那个Task对象，可以对该对象访问Result属性拿到执行的结果

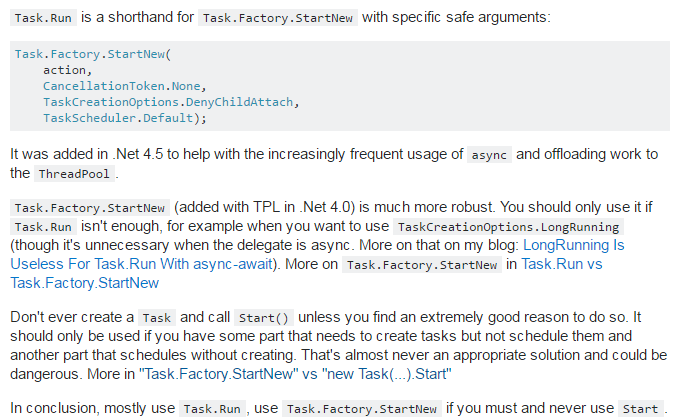
### Task.Run 不在需要调用Task.Start

只有用构造函数new出来的Task 才需要显示的调用Start() 方法

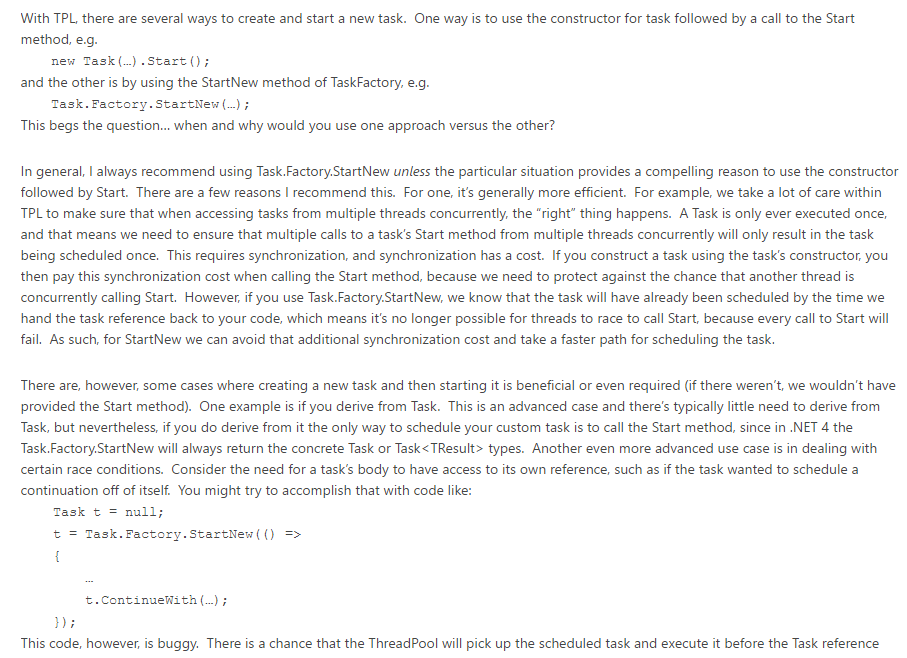
一下三个方法在功能上是相同的，但是性能上稍有不同

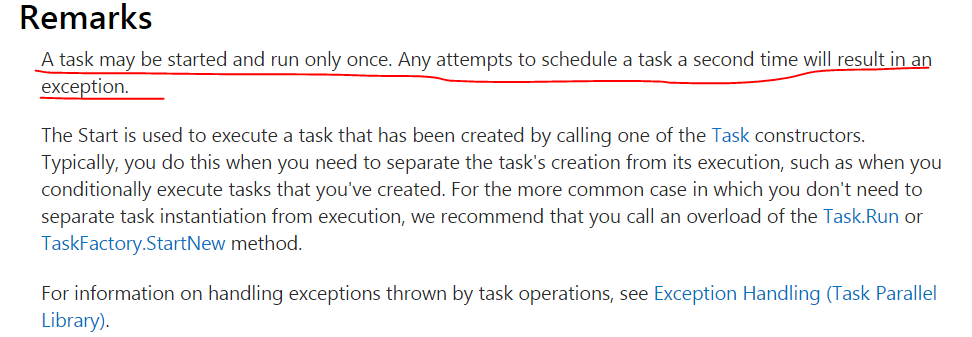
[Regarding usage of Task.Start() , Task.Run() and Task.Factory.StartNew()](http://stackoverflow.com/questions/29693362/regarding-usage-of-task-start-task-run-and-task-factory-startnew)





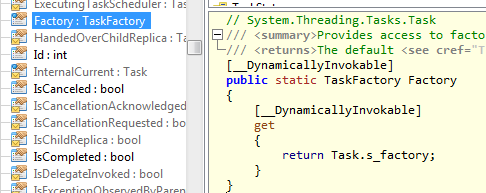
三者在性能上的差异：["Task.Factory.StartNew" vs "new Task(…).Start"](https://blogs.msdn.microsoft.com/pfxteam/2010/06/13/task-factory-startnew-vs-new-task-start/)





[Task.Start Method ()](https://msdn.microsoft.com/en-us/library/dd270682(v=vs.110).aspx)

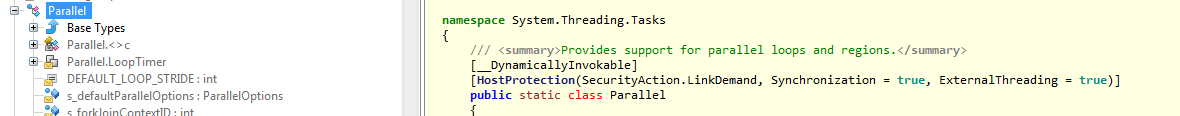
### Task.Factory 和 TaskFactory 其实是一回事



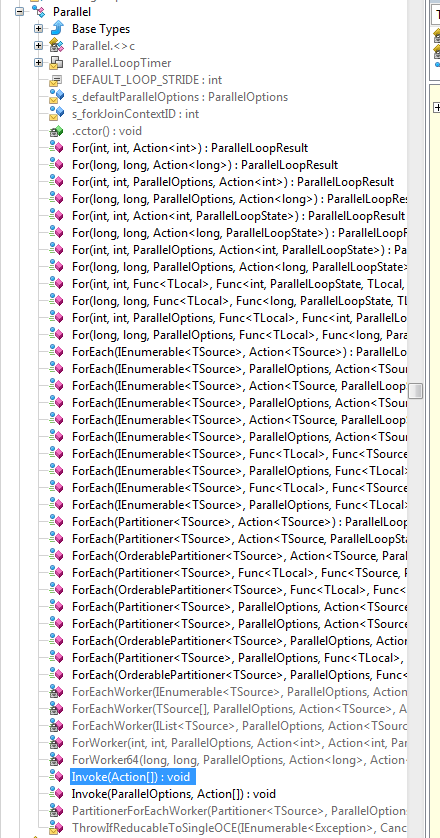


### Parallel 类

[C#并行编程-Parallel](http://www.cnblogs.com/woxpp/p/3925094.html)







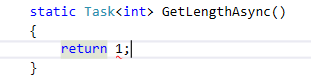
## Async 和 await

[.NET 中的async/await 异步编程](http://blog.jobbole.com/85787/)

### Async 在函数定义时用， await 在函数调用时用

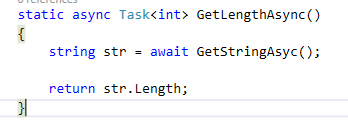
一个是定义关键字，一个是调用关键字

### 没有async 关键字的时候，不能直接在定义为Task<TResult> 的方法中返回TResult 类型



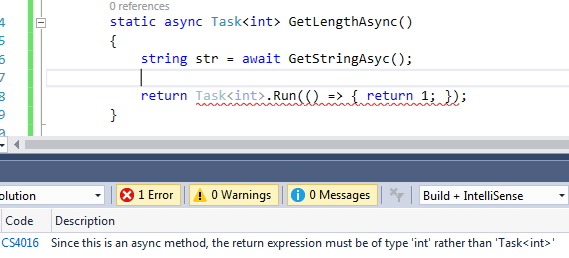


### 使用了async 关键字后，可以直接返回TResult 方法



### 使用了async关键字后，反而不能返回Task 类型了

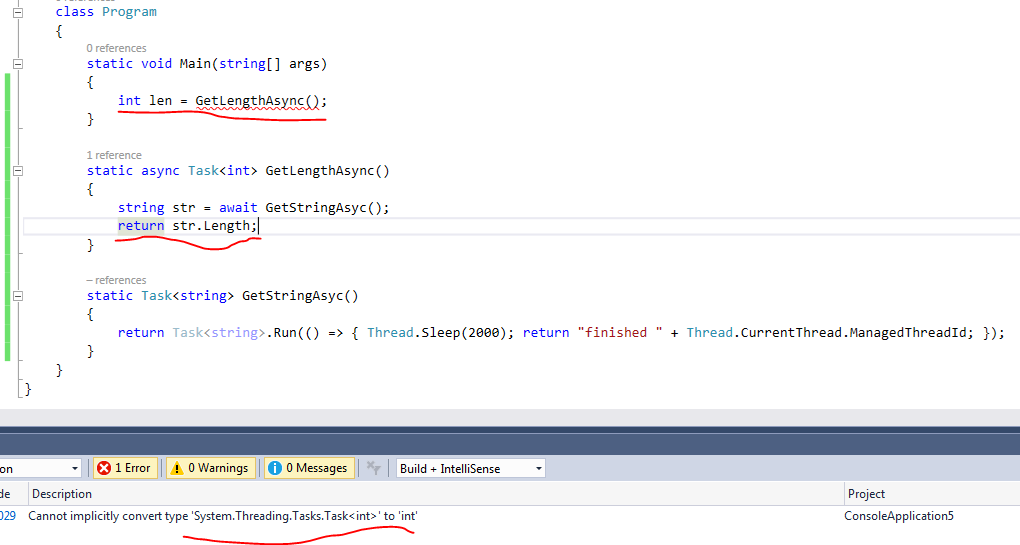
Fuck，这C#编译器他妈就是把程序员当玩具了，想怎么玩就怎么玩？



### 使用async 不使用await 没错误，会有warning



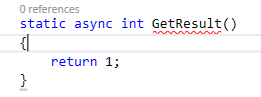
### 在调用async 方法时，还必须使用Task 而不是TResult 来接收返回值



正确写法：

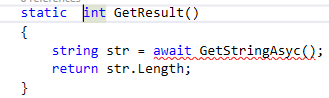


### 能用async 修饰的函数，其返回类型只能为void ， Task 或者 Task<TResult>



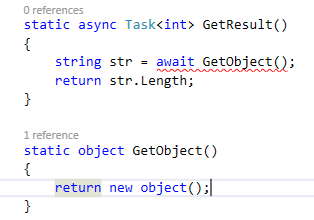


### Await 只能出现在用async 修饰的方法内部





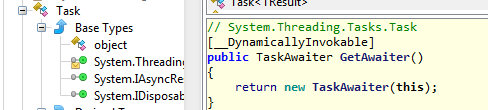
### Await 之后只能跟Task 或者 Task<TResult> 对象

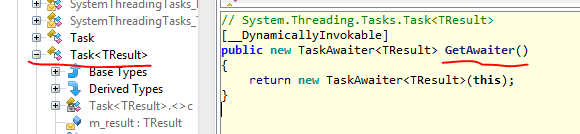




Await 虽然常表现为后面跟对方法的调用，但await 真正的作用对象时一个Task 或者Task<Result>对象， 其实await 相当于是调用task.wait ，task的生成是在其他线程生成的。

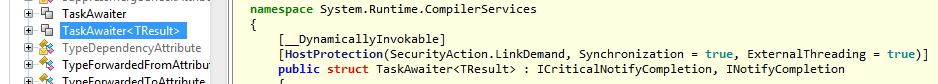
只有Task 或者Task<TResult> 才定义的有GetAWaiter()

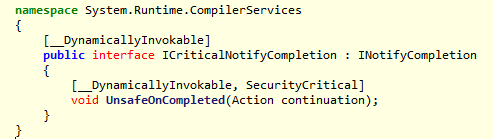


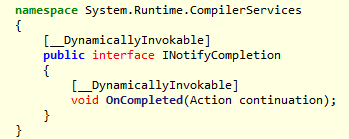


### TaskAwaiter 和 TaskAwaiter<TResult>







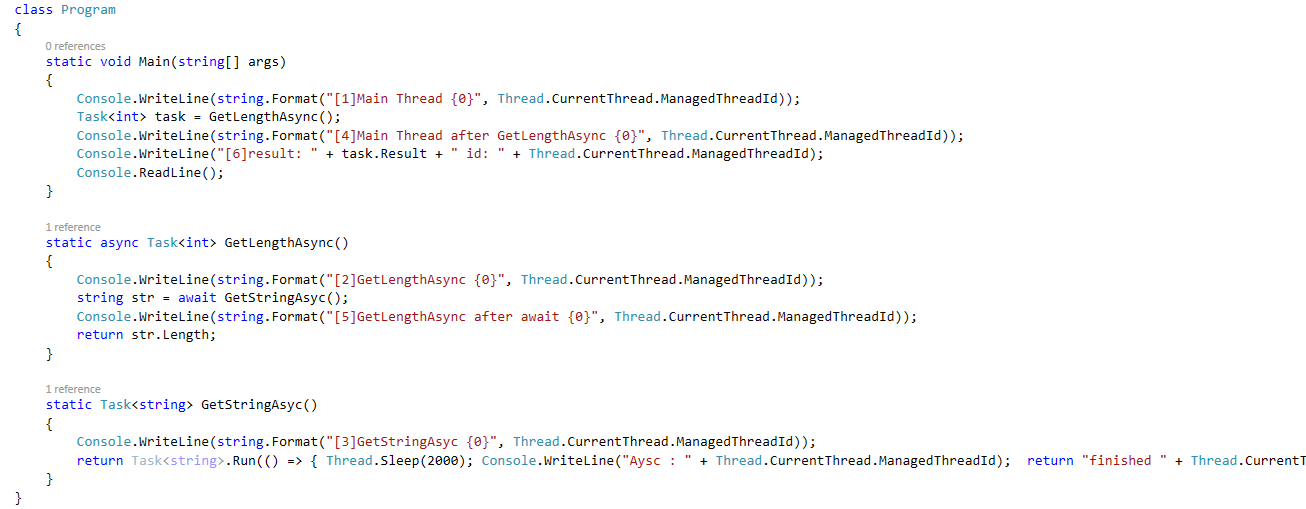


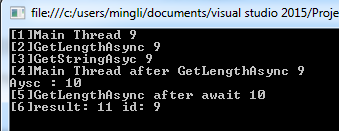
### Async 和await 设计的线程

在遇到async 中的await 关键字后，程序调到async 方法的调用语句之后继续执行，还是原来的线程

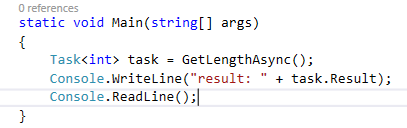
Await 后跟方法调用时（此方法调用必须返回Task 之类的），并非整个方法都是用新线程执行， 其实还是来线程，想象new 一个Task对象的语句，这个new 的语句肯定是在老线程上执行的，只不过task封装的那个方法才是在新线程执行。

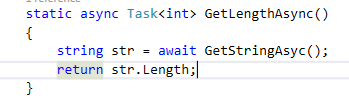
当await 修饰的方法返回后，先接着await语句的下面开始执行（此位置应该在async 修饰的方法内部），线程同执行task封装的方法线程是同一个。

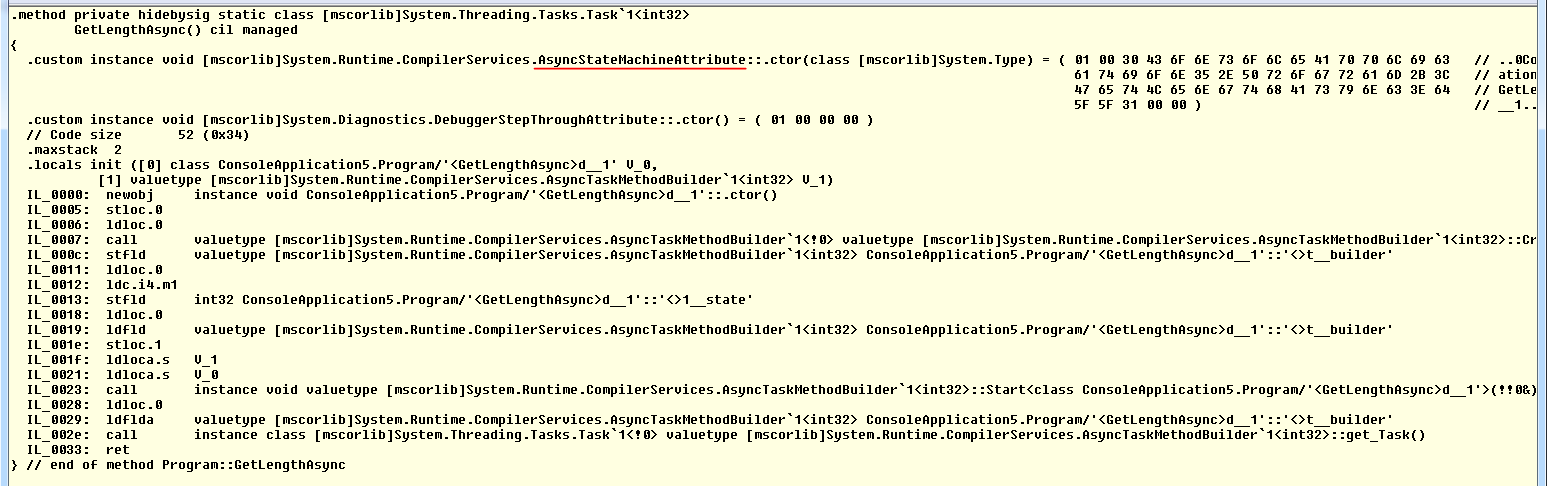




### 使用async 被C#编译器加了很多代码： AsyncStateMachineAttribute







## APM 异步编程模型

演进历史 APM(.NET 1.0) 🡪 EAP(.NET 2.0) 🡪 TAP(.NET 4.0)🡪async/await(.NET 4.6)

[异步编程模型(APM)](http://www.cnblogs.com/zhili/archive/2013/05/10/APM.html)

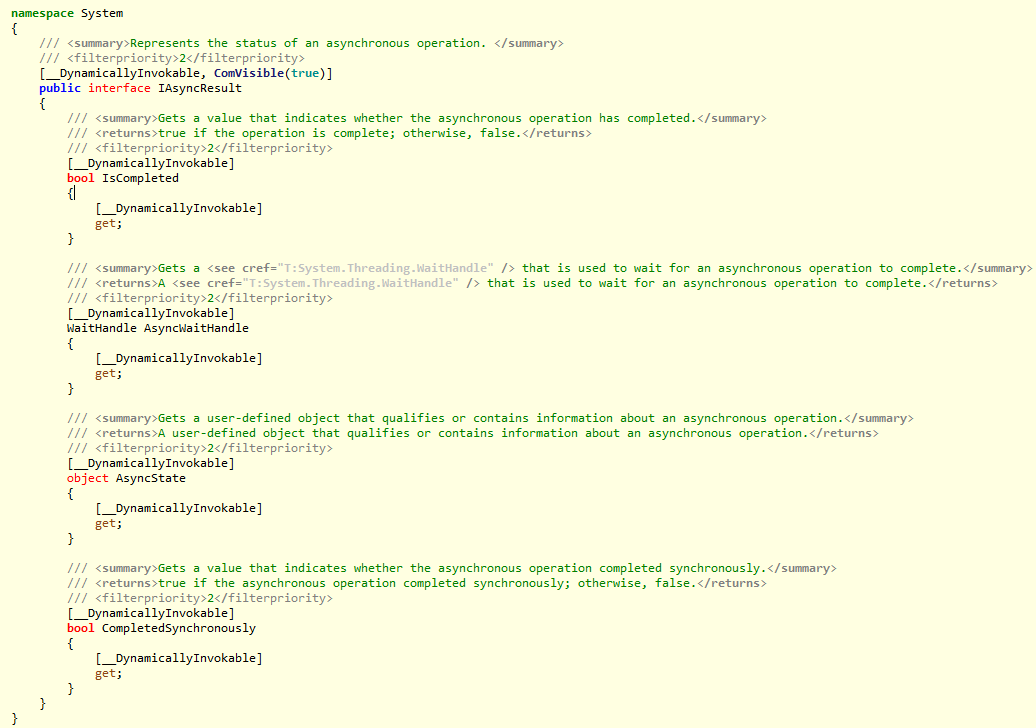
### History

APM即异步编程模型的简写（Asynchronous Programming Model），.NET肯定会经常看到和使用以BeginXXX和EndXXX类似的方法，其实你在使用这些方法的时候，你就在使用异步编程模型来编写程序。

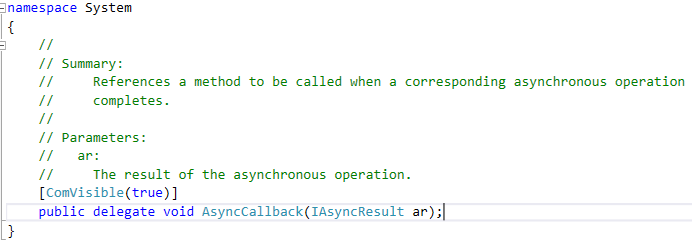
异步编写模型是一种模式，该模式允许用更少的线程去做更多的操作，.NET Framework很多类也实现了该模式，同时我们也可以自定义类来实现该模式，（也就是在自定义的类中实现返回类型为IAsyncResult接口的BeginXXX方法和EndXXX方法），另外委托类型也定义了BeginInvoke和EndInvoke方法，并且我们使用WSDL.exe和SvcUtil.exe工具来生成Web服务的代理类型时，也会生成使用了APM的BeginXxx和EndXxx方法。

### IAsyncResult

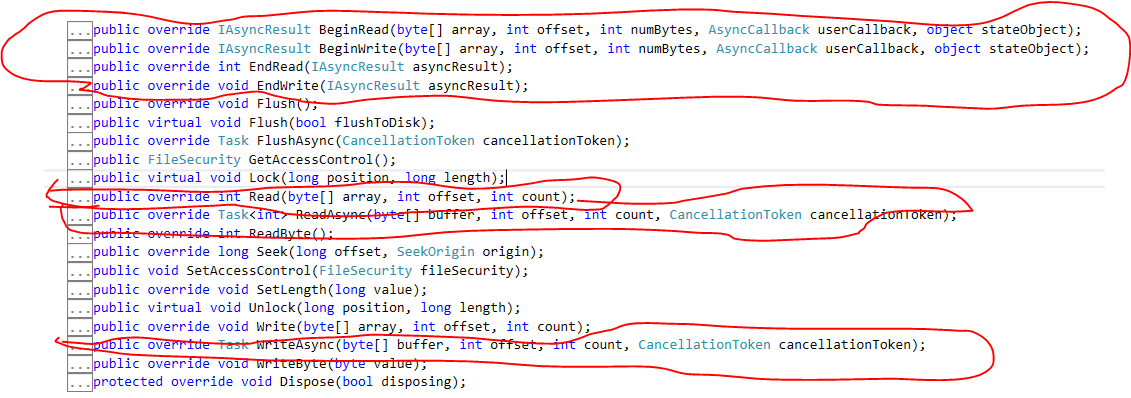
在mscorlib.dll 的 System 命名空间中定义，所以需要时不需要特别引用



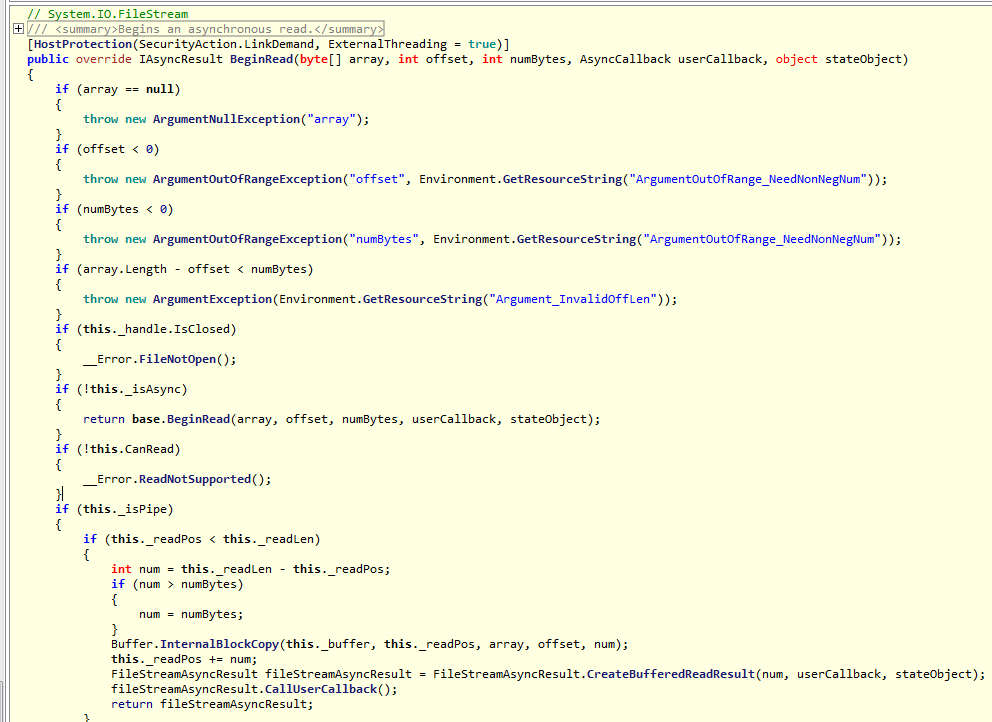
### AsyncCallBack delegate



### FileStream 为例



2.0 版本的BeginRead 和4.0版本的BeginRead是非常不同的





## EAP

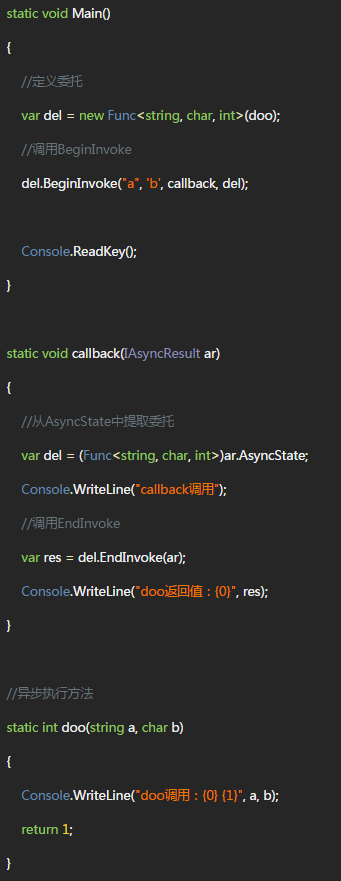
演进历史 APM(.NET 1.0) 🡪 EAP(.NET 2.0) 🡪 TAP(.NET 4.0)🡪async/await(.NET 4.6)

## TPL：TaskFactory.FromAsync与委托的异步调用

[TPL：TaskFactory.FromAsync与委托的异步调用](http://web.seoluo.com/aspnet_7360.html)

这个例子比较清楚的说明了为什么IAsyncResult 里面要定义一个AsyncState 的Object对象，以及它怎么赋值。

其实AsyncState 对象就是在上面调用其BeginXXXX 方法的那个对象， 因为在callback方法里面，需要显示的调用其EndXXXX方法才能拿到最终的结果，那么调用EndXXXX 时，去哪找对象的reference呢，这时候就需要用到IAsyncResult 中的AysncState属性。而IAsyncResult 是作为结果传入回调函数的。 当然，如果你保留一个调用对象的global reference 也能实现，但是代码直接耦合度大了。





[C# 5.0 新特性——Async和Await使异步编程更简单](http://www.cnblogs.com/zhili/archive/2013/05/15/Csharp5asyncandawait.html)

[[C# 网络编程系列]专题九：实现类似QQ的即时通信程序](http://www.cnblogs.com/zhili/archive/2012/09/23/QQ_P2P.html)

[基于任务的异步模式](http://www.cnblogs.com/zhili/archive/2013/05/13/TAP.html)

[[你必须知道的异步编程]C# 5.0 新特性——Async和Await使异步编程更简单](http://www.cnblogs.com/zhili/archive/2013/05/15/Csharp5asyncandawait.html)

# C# 网络编程

程序集在System.dll 中， namespace 为System.net.sockets

类可以分为三组 Socket 是基础类，当然你也可以直接使用Socket 来编程

TcpClient 和 TcpListener 是在socket 基础上对TCP协议的分装

UdpClient 是在socket 基础上对UDP协议的封装。注意没有udpserver类。 Dupclient 即可以接受本机IP 和远端IP，分别对应着server和client

