

Coding Convention

For C#



August 4, 2014

Allen Lu

目录

[2 前言 4](#_Toc394914232)

[2.1 编写目的 4](#_Toc394914233)

[2.2 适用范围 4](#_Toc394914234)

[3 命名规范 5](#_Toc394914235)

[3.1 命名约定 5](#_Toc394914236)

[3.1.1 PascalCasing 5](#_Toc394914237)

[3.1.2 camelCasing 5](#_Toc394914238)

[3.1.3 UPPER\_CAPS 5](#_Toc394914239)

[3.1.4 私有变量的命名 5](#_Toc394914240)

[3.1.5 首字母缩写词的大小写 5](#_Toc394914241)

[3.1.6 复合词的大小写 6](#_Toc394914242)

[3.1.7 在带单位的值的变量后加上“\_camelCasing单位” 6](#_Toc394914243)

[3.1.8 不要使用匈牙利命名法 6](#_Toc394914244)

[3.1.9 使用英语语序命名标识符 6](#_Toc394914245)

[3.1.10 尽量不要使用单词缩写，除非它是众所周知的 7](#_Toc394914246)

[3.2 命名选择 7](#_Toc394914247)

[3.2.1 名字一定要能够表达出标识符的含意 7](#_Toc394914248)

[3.2.2 选择意义单一明确的名字 7](#_Toc394914249)

[3.2.3 使用不会产生歧义的名字 8](#_Toc394914250)

[3.2.4 命名要与使用者的期望相匹配 8](#_Toc394914251)

[3.2.5 为名字附加更多的信息 9](#_Toc394914252)

[3.2.6 不要卖弄风骚 9](#_Toc394914253)

[3.3 特定场景下的命名最佳实践 10](#_Toc394914254)

[3.3.1 命名空间 10](#_Toc394914255)

[3.3.2 要让接口的名字以字母I开头 10](#_Toc394914256)

[3.3.3 派生类的末尾使用基类名称 10](#_Toc394914257)

[3.3.4 泛型类型参数的命名 10](#_Toc394914258)

[3.3.5 枚举类型的命名 10](#_Toc394914259)

[3.3.6 要用动词和动词短语命名方法 11](#_Toc394914260)

[3.3.7 属性的命名 11](#_Toc394914261)

[3.3.8 事件的命名 11](#_Toc394914262)

[3.3.9 字段的命名 11](#_Toc394914263)

[4 注释 12](#_Toc394914264)

[4.1 注释约定 12](#_Toc394914265)

[4.1.1 类注释约定 12](#_Toc394914266)

[4.1.2 类属性注释约定 12](#_Toc394914267)

[4.1.3 方法注释约定 12](#_Toc394914268)

[4.1.4 代码间注释约定 12](#_Toc394914269)

[4.1.5 强制注释的约定 13](#_Toc394914270)

[4.1.6 常用注释标识的约定 13](#_Toc394914271)

[4.1.7 关于何时使用“///”和“//”的约定 14](#_Toc394914272)

[4.2 不需要的注释 14](#_Toc394914273)

[4.2.1 不要为了注释而注释 14](#_Toc394914274)

[4.2.2 不要用注释来粉饰糟糕的代码 15](#_Toc394914275)

[4.2.3 日志式注释 15](#_Toc394914276)

[4.2.4 个人签名 15](#_Toc394914277)

[4.2.5 位置标识 15](#_Toc394914278)

[4.2.6 注释掉的代码 16](#_Toc394914279)

[4.3 需要的注释 16](#_Toc394914280)

[4.3.1 记录你对代码有价值的见解 16](#_Toc394914281)

[4.3.2 为代码中的不足写注释 16](#_Toc394914282)

[4.3.3 对意料之中的疑问添加注释 16](#_Toc394914283)

[4.3.4 公布可能的陷阱 17](#_Toc394914284)

[4.3.5 对于代码块总结性地注释 17](#_Toc394914285)

[4.4 如何写好注释 18](#_Toc394914286)

[4.4.1 避免使用不明确的代词 18](#_Toc394914287)

[4.4.2 精确描述方法的行为 18](#_Toc394914288)

[4.4.3 用输入输出例子来说明特殊的情况 19](#_Toc394914289)

[4.4.4 适当的使用具名函数 19](#_Toc394914290)

[4.4.5 使用通用专有名词 19](#_Toc394914291)

[4.4.6 更新代码时记得更新注释 20](#_Toc394914292)

[4.4.7 只有能让别人读懂的注释才是合格的注释 20](#_Toc394914293)

[5 格式 21](#_Toc394914294)

[5.1 长度 21](#_Toc394914295)

[5.2 空格、空行 21](#_Toc394914296)

[5.3 换行 21](#_Toc394914297)

[5.4 Region的使用 22](#_Toc394914298)

[5.5 使用VS的Format Document功能 22](#_Toc394914299)

[6 Reference 23](#_Toc394914300)

# 前言

## 编写目的

为了保证大家编写出的程序都使用统一的风格，以方便阅读和后期维护。

编码规范对于程序员而言尤为重要，有以下几个原因：

1. 一个软件的生命周期中，80%的花费在于维护。
2. 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护。
3. 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码 。
4. 编码规范可以保证代码阅读者在阅读代码时产生尽可能少的歧义。
5. 编码规范可以使我们的代码统一、美观，让客户可以一眼看出我们是专业的团队。

编码规范的核心出发点是

1. 让其他人能够快速，准确的知道代码的作用并且确保不会出现歧义（其他人也包括一段时间后的自己）
2. 让我们的代码看着更加清晰、美观、专业、统一。

## 适用范围

因为我们是外包项目组，如果客户方给出他们的代码规范就使用客户方的代码规范；否则需遵循本规范。

# 命名规范

任何标识符的名字都应该可以简单、清楚、正确的表示出该标识符的作用。

同时我们要将尽可能多的信息装入到标识符的名字当中去，以便读代码的人可以快速的读懂代码。

## 命名约定

我们在命名标识符时（包括参数，常量，变量），应使用单词的首字母大小写来区分一个标识符中的多个单词，如UserName.

### PascalCasing

PascalCasing包含一到多个单词，每一个单词第一个字母大写，其余字母均小写。例如：HelloWorld、SetName等。

除了参数、变量、常量外，所有命名空间名称、类、函数、接口、属性、事件、枚举等名称的命名，使用 Pascal 风格。

### camelCasing

camelCasing包含一到多个单词，第一个单词首字母小写，其余单词首字母大写。例如：name、productId等。

参数与变量的命名使用camelCasing.

### UPPER\_CAPS

UPPER\_CAPS包含一到多个单词，每个单词的所有字母都大写，单词与单词之间用”\_”连接，该风格目前在c#中只用于const常量。

如：public const string DEFAULT\_PAGE = "default.aspx";

### 私有变量的命名

Private 的私有变量使用下划线”\_”+camelCasing的大小写规则，以便快速确认该变量的作用域。

如: private int \_userId;

### 首字母缩写词的大小写

首字母缩写词是由一个短语的首字母组成的，如Xml(ExtensibleMarkuLaguage),IO(Input and Output)。它和单词缩写是有区别的，单词缩写仅仅是把一个单词的长度变短。

1. 把两个字母的首字母缩写词全部大写，除非它是camelCasing的第一个单词。

using System.IO;

public void StartIO(Stream ioStream)

1. 由三个或以上的字母组成的首字母缩写词，只有第一个字母大写，如Xml,Html.除非首字母是camelCasing标识符的第一个单词。

using System.Xml;

public void ProcessXmlNode(XmlNode xmlNode)

### 复合词的大小写

不要把复合词中的首字母大写。复合词要当成一个单词来处理。

如endpoint, callback,metadata,namespace等都是正确的写法

### 在带单位的值的变量后加上“\_camelCasing单位”

将单位加入标识符命名中，可以使使用者快速准确的知道传人数据的单位，减少错误的发生。

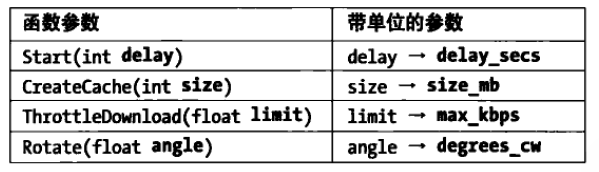
如public void CreateCache(int cacheSize)

传入的数据是bytes, KB, MB 还是GB？

改成public void CreateCache(int cacheSize\_mb)

一目了然，并且会减少调用者传入错误数据的可能。

其他一些没有单位的函数参数以及带单位的版本。



### 不要使用匈牙利命名法

匈牙利命名法是指用小写形式的数据类型缩写来作为变量名的前缀。如：strName,intCount。

这种命名法在C和C++时代很流行，可以帮助程序员记住自己的类型。

但在C#中需要禁用，除非你有足够的理由，因为：

1. C#都是强类型的，现在的IDE(如Visual Studio)可以自动的检测出当前变量的类型以及类型错误



1. 开发初期经常需要修改变量的类型，使用匈牙利命名法维护很困难。

### 使用英语语序命名标识符

人在阅读代码时，能更快的理解符合其阅读习惯的命名。

如VerticalAlignment比AlignmentVertical能让人更快的知道该变量的含意。

### 尽量不要使用单词缩写，除非它是众所周知的

## 命名选择

### 名字一定要能够表达出标识符的含意

标识符名字必须要表达出该标识符的意义，绝对不可以使用无意义的v1,v2…vn之类的命名。

public static void CloneChars(char[] cl1, char[] cl2)

{

for (var i = 0; i < cl1.Count(); i++)

{

cl2[i] = cl1[i];

}

}

代码的调用者不看这函数是无法知道cl1还是cl2是要拷贝的char数组，他必须进到这个函数去看完整个逻辑才可以调用。而且在看的过程中cl2[i] = cl1[i]; 也需要他花几秒钟来思考是做什么的。

如果改成有意义的名字： source 和target那么这个方法调用者一看名字就知道使用方法了。

public static void CloneChars(char[] source, char[] target)

{

for (var i = 0; i < source.Count(); i++)

{

target[i] = source[i];

}

}

### 选择意义单一明确的名字

在命名时要使用专业的单词，避免使用“空洞”的单词

如： class BinaryTree

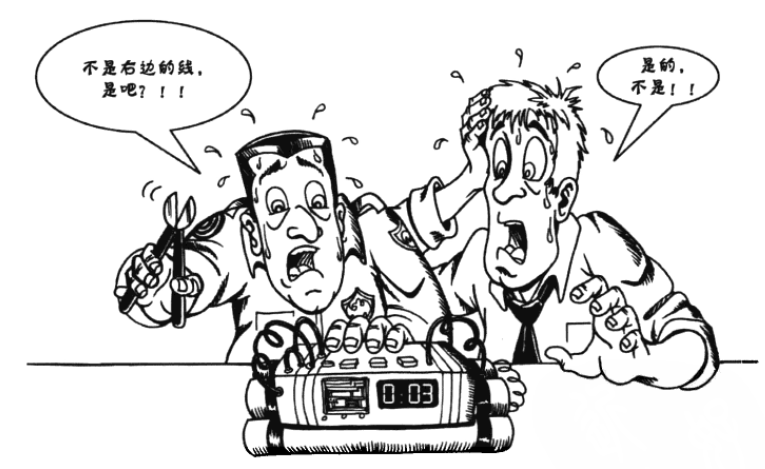
{

public int Size()

看到这行代码你想到Size会返回什么，树的高度，节点数还是树在内存中的空间？

我们可以使用更单一明确的词来告诉读者这个方法的具体含义，如Height,NodesNum,Memory\_Bytes

### 使用不会产生歧义的名字



在给标识符命名时，一定不能产生歧义，代码中的很多错误都是由于命名时的歧义造成的。例如：

public const int CART\_TOO\_BIG\_LIMIT = 10;

if (ShoppingCart.Count() >= CART\_TOO\_BIG\_LIMIT)

{

LogError("Too many items in cart.");

}

这段代码有个很经典的”大小差一缺陷“。在判断购物车物品上限时，我是应该使用 “>“还是应该使用”>=”,我是无法从代码中判断出来的，所以这个地方很容易出现bug.如果我们换成MAX\_ITEMS\_IN\_CART, 那我马上就可以判定出这里要使用“>“。

### 命名要与使用者的期望相匹配

有些名字之所以会让人误解是因为带吗阅读者对它们有先入为主的印象，就算你本意并非如此。这种情况下，你最好是选用一个与使用者期望所匹配的名字。

如很多程序员都习惯了把Get开始的方法当作“轻量级访问器“，他只是简单的返回成员变量。

大家看到以下的代码

class BinaryTree

{

public int GetNodesCount()

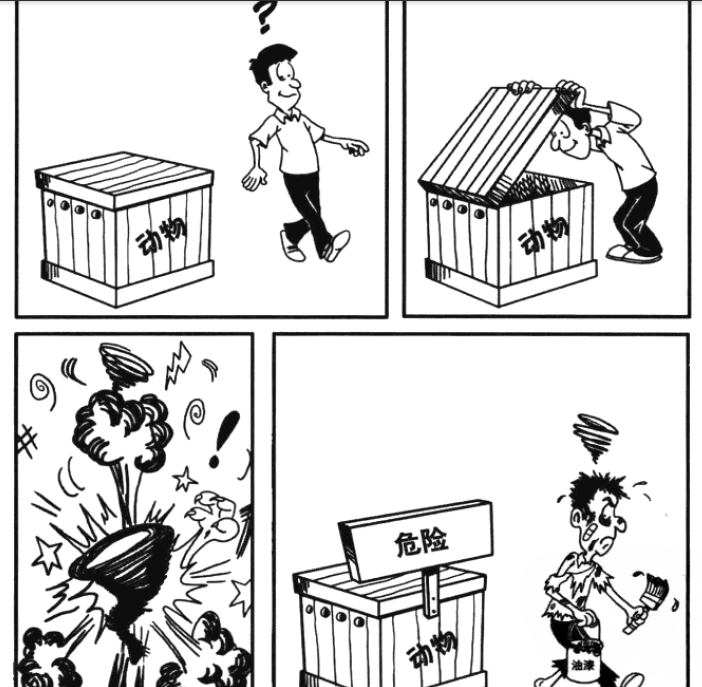
会以为只是返回内部private int \_nodesCount; 私有变量的访问器。

但如果实际你的代码可能是一个非常耗时的代码，内部实现是广度优先遍历所有的树节点，还要去数据库查找父节点和子节点的关系，然后累加。

那么这么一个耗时的方法可能由于你的命名，导致了被调用者反复多次的调用，导致整个系统性能下降。

如果你将命名改为ComputeNodesCount那么调用者就会知道这是个耗时的操作，需要缓存调用结果并减少调用。

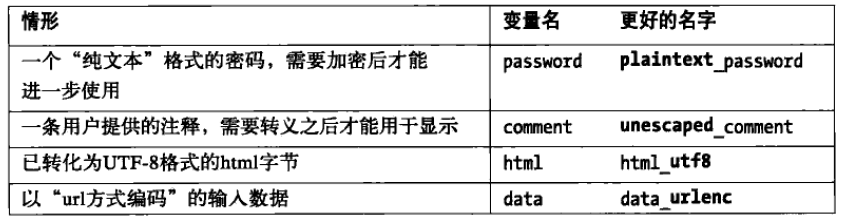
### 为名字附加更多的信息



一个变量名就像一个小注释，尽管空间不大，但不管你在命中挤进任何额外的信息，每次有人看到命名时都会看到这些信息。

例子：当你从网页接收了请求的表单，里面可能还有不安全的代码，如注入语句等，这时你在命名时需要体现该数据不安全，可以使用unsafeFormData,当调用完安全检查方法后可以将其改为 safeFormData = HandleUnsafeData(unsafeFormData).这样代码阅读者就知道可以放心的使用该变量了。

下表给出了更多需要给名字附加额外信息的例子



### 不要卖弄风骚

使用最常用，众所周知的单词。不要在代码命名时卖弄你的学识，要让你的代码快速准确的表达出你的想法才是真正的牛人。

如public static string ConvertXml2Html (string sourcePath)

有些人在看到这个方法的时候怎么想也想不明白这个2是做什么用的，是把一个Xml文件变成两个Html?

熟悉英语文化的人可能知道这是To的俚语表达。如果你不能保证所有阅读你代码的人都知道2是To的缩写。那么请使用ConvertXmlToHtml命名。

## 特定场景下的命名最佳实践

### 命名空间

1. 要使用PascalCasing,并用点号来分隔名字空间中的各个部分。

如Microsof.Office.PowerPoint

1. 要用公司名作为命名空间的前缀，这样就可以避免与另外一家公司使用相同的名字。
2. 要用稳定的，与版本无关的产品名称作为命名空间的第二层
3. 不要使用公司的组织架构来决定命名空间的层次结构，因为内部组织结构经常改变。
4. 不要用相同的名字来命名命名空间和该空间内的类型。

例如，不要先将命名空间命名为Debug,然后又在该空间中提供Debug类。大部分编译器包括VS要求用户在这样的类型前加上完整的限定符。

### 要让接口的名字以字母I开头

如IComponet,IDisposable 大家一看就知道是接口。

同时要确保如果一个类是一个接口的标准实现，那么这个类和接口应该只差一个”I“前缀。

### 派生类的末尾使用基类名称

例如，从 Stream 继承的 Framework 类型以 Stream 结尾，从 Exception 继承的类型以 Exception 结尾。

### 泛型类型参数的命名

1. 使用描述性的名字来命名泛型类型参数，并且在前面加上T前缀

如下面都是很好的命名

public delegate TOutput Converter<TInput, TOutput>(TInput from);

1. 如果只有一个类型参数，可以只用一个字母T来表示泛型

public class Nullable<T>

public class List<T>

1. 如果泛型参数有约束，那么需要在泛型类型参数名中需要显示出该约束

public interface ISessionChannel<TSession> where TSession:ISession

### 枚举类型的命名

1. 要用单数名词而不是复数命名枚举类型,如要用ConsoleColor而不是ConsoleColors

public enum ConsoleColor

{

Red,

Yellow,

Blue

}

1. 不要给枚举类型加”Enum“、”Flag”等后缀。

ColorEnum,ColorFlag都不好，因为本身就是枚举，再加上就是没有意义的重复 。

### 要用动词和动词短语命名方法

### 属性的命名

1. 要用名词、名词短语或形容词来命名属性
2. 要用描述集合中具体内容的短语的复数形式来命名属性集合，而不要用短语的单数形式加”List“、”Array”或”Collection“后缀

class BinaryTree

{

//Good Naming

public NodeCollection Nodes { get; set; }

//Bad Naming

public NodeCollection NodesCollection { get; set; }

1. 要用肯定性的短语命名布尔属性。最好在前面选择性的加入”Is“、”Can“、”Has“等前缀。

CanSeek比CantSeek和Seekable都更准确和容易理解。

### 事件的命名

1. 要用动词或动词短语命名事件

如： Clicked、Painting、DroppedDown 等等

1. 要用现在进行时(ing)和过去式(ed)来赋予事件发生之前和之后的概念。而不是使用Before和After.

如窗口关闭前发生的close事件应该命名为Closing,而在窗口关闭之后发生的应该命名为Closed.

### 字段的命名

1. 禁止使用实例的公有字段和受保护字段，请使用属性代替。

Tips:在VisualStudio中输入”prop”可快速创建外部可修改的属性，输入”propg”可快速创建不允许外部修改的属性。如：

//prop

public int NodesCount { get; private set; }

//propg

public List<BinaryNode> Nodes { get; set; }

1. 一般只使用静态字段
2. 要使用名词、名词短语或形容词命名字段
3. 不要给字段加前缀如“g\_”、”s\_”来表示静态字段。因为字段和属性是非常相似的，所以要遵循相同的命名规范。

# 注释

注释毫无疑问是让别人以最快速度了解你代码的最快途径，但写注释的目的绝不仅仅是”解释代码做了什么“，更重要的尽量帮助代码阅读者对代码了解的和作者一样多。

当你写代码时，你脑海里会有很多有价值的信息，但当其他人读你代码时，这些信息已经丢失，他们所见到的只是眼前代码。

## 注释约定

如果IDE提供注释格式，则尽量使用IDE提供的格式，否则使用”//”来注释。类、属性和方法的注释在Visual Studio中都使用输入”///”自动生成的格式。

### 类注释约定

/// <summary>

/// 类说明

/// </summary>

public class BinaryTree

### 类属性注释约定

/// <summary>

/// 属性说明

/// </summary>

public int NodesCount { get; private set; }

### 方法注释约定

/// <summary>

/// 方法说明

/// </summary>

/// <param name="parentNode">参数说明</param>

/// <returns>返回值说明</returns>

public int ComputeChildNodesCount(BinaryNode parentNode)

### 代码间注释约定

1. 单行注释，注释行数<3行时使用

//单行注释

1. 多行注释，2<注释行数<=10时使用

/\*多行注释1

多行注释2

多行注释3\*/

1. 注释块，10<注释行数时使用，用50个\*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 代码块注释1

\* 代码块注释2

\* ......

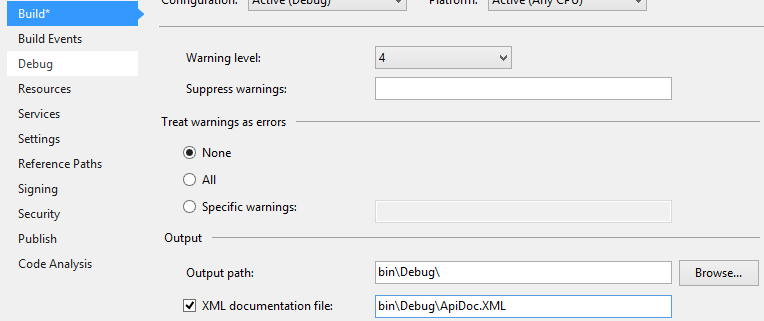
\* 代码块注释10

\* 代码块注释11

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 强制注释的约定

1. 以下三种情况我们需要在所有的类、类属性和方法都必须按照上述格式编写注释
2. 客户方对代码注释重视程度较高
3. 我们需要提供代码注释自动生成的API文档。



1. 目前编写的是公共核心模块
2. 如果客户方没有对注释特殊要求，那么按照下文中讨论的只在需要的地方加注释。不要加无谓的注释。

### 常用注释标识的约定

这里约定下以后团队常用几种注释标识及含义：

//**TODO: 我还没有处理的事情**

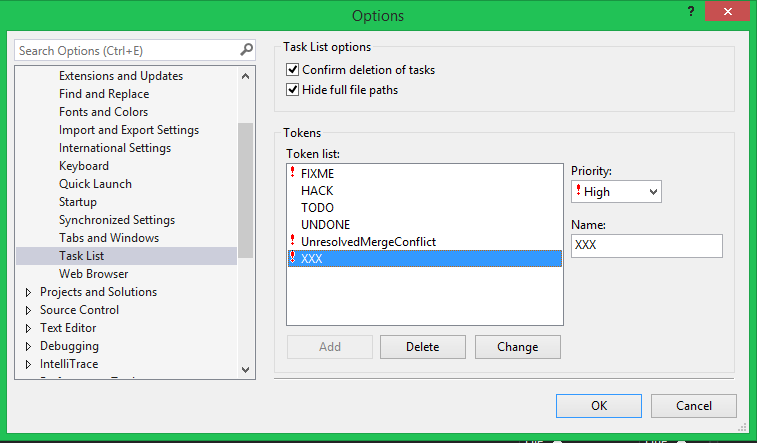
//FIXME: 已知的问题

//HACK: 对一个问题不得不采用比较粗糙的解决方案

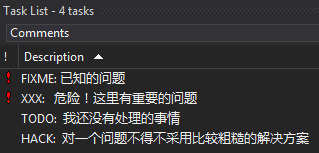
//XXX: 危险！这里有重要的问题

请团队成员自行在Visual Studio中配置FIXME和XXX为高优先级的Comments.

Steps: Tools->Options->Environment->Task List->Tokens->Add->OK

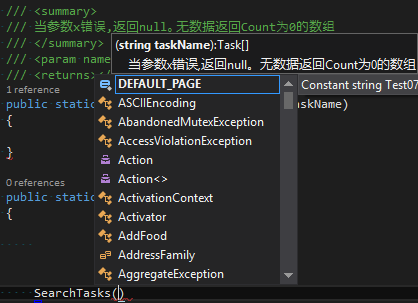


配置完成后，我们就能在Task List（Ctrl+w,t）窗口中的Comments选项中看到代码中存在的任务了。



### 关于何时使用“///”和“//”的约定

1. 对于需要让调用者知道的信息，使用“///”注释，以便让调用者能在调用时看到。



1. 对于代码内部实现细节，需要维护者知道的注释，使用“//”。减少调用者阅读时间。

## 不需要的注释

阅读注释会占用阅读真实代码的时间，并且每条注释都会占用屏幕上的空间。所以我们约定所加的注释必须是有意义的注释，否则不要浪费时间和空间。

区别要不要写注释的核心思想就是:不要为那些能快速从代码本身就推断的事实写注释。

### 不要为了注释而注释

有些人可能以前的公司对于注释要求很高，如“何时写注释”章节中的要求。所以很多人为了写注释而注释。

再没有特殊要求的情况下我们要禁止写下面这种没有意义的注释。

/// <summary>

/// The class definition for Account

/// </summary>

public class BinaryTree

{

/// <summary>

/// Total counts of the nodes

/// </summary>

public int NodesCount { get; private set; }

/// <summary>

/// All the nodes in the tree

/// </summary>

public List<BinaryNode> Nodes { get; set; }

/// <summary>

/// Insert a node to the tree

/// </summary>

/// <param name="node">the node you want insert into the tree</param>

public void InsertNode(BinaryNode node)

### 不要用注释来粉饰糟糕的代码

写注释常见的动机之一就是试图来使糟糕的代码能让别人看懂。对于这种“拐杖式注释”，我们不需要，我们要做的是把代码改的能够更具有”自我说明性“。

记住：“好代码>坏代码+好注释”

如下面这段函数的注释

//Enforce limits on the reply as stated in the request

//such as the number of items returned, or total byte size,etc.

public void CleanReply(Request request,Reply reply)

既然知道这个函数名会让人很难读懂，那么为什么不直接改好名字呢？这样所有调用这个函数的地方都能很快速知道这个函数的作用，不用再跟进来看函数的作用。

public void EnforceLimitsFromRequestOnReply(Request request,Reply reply)

### 日志式注释

有人喜欢在每次编辑代码时，都在模块开始处加一条注释。这类注释就像是一种记录每次修改的日志。在很久以前这种记录对于维护还有意义。但是对于现在的源码控制来说，这些记录完全是冗余的，需要完全废除。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* July-29-2014:Fix **Bug-12345: Add new method to calculate nodes count**

\* July-20-2014:Fix **Bug-11111: Add Insert new node method**

\* ......

\* July-20-2014:Task-00001: Create BinaryTree class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 个人签名

//Added By XXXX

有人认为这种注释有助于不了解这段代码含意的人和他讨论。事实上确是这条注释放在那一年复一年，后来的代码和原作者写的源码越来越不一样，和XXXX也越来越没关系。

重申一下，TFS里都能看到这类信息，不要加在代码里。

### 位置标识

//AddNodePlace1

//AddNodePlace2

有人喜欢在代码注释里加入位置标识以方便他查找代码的位置。

现在的IDE都集成了这些功能，如VS中可以使用Bookmark(Ctrl+b,t)。

不要将这类注释加到代码中。

### 注释掉的代码

直接把代码注释掉是非常令人讨厌的做法。

其他人不敢删掉这些代码。他们会想代码依然在这一定是有原因的，而且这段代码很重要，不能删除。而且每个阅读代码的人都会去看一下这些被注释掉的代码中是否有他们需要注意的信息。

这些注释掉的代码会堆积在一起，散发着腐烂的恶臭。

## 需要的注释

### 记录你对代码有价值的见解

你应该在代码中加入你对代码这段代码有价值的见解注释。

如： //出乎意料的是，对于这些数据用二叉树比哈希表要快40%

//哈希运算的代价比左右比要大的多

这段注释会告诉读者一些重要的性能信息，防止他们做无谓的优化。

### 为代码中的不足写注释

代码始终在演进，并且在代码中肯定会有不足。

要把这些不足记录下来以便后来人完善。

如当代码需要改进时：

//**TODO:尝试优化算法**

如当代码没有完成时：

//**TODO:处理JPG以外的图片格式**

你应该随时把代码将来该如何改动的想法用注释的方式记录下来。这种注释给读者带来对代码质量和当前状态的宝贵见解，甚至会给他们指出如何改进代码的方向。

### 对意料之中的疑问添加注释

当别人读你的代码的时候，有些部分可能让他们有这样的疑问：“为什么要这样写？”你的工作就是要给这些部分加上注释。

如：

// 因为Connection的创建很耗费资源和时间，而且需要多线程访问，

// 所以使用多线程单例模式

public static Connection Instance

{

get

{

if(\_instance==null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Connection();

}

}

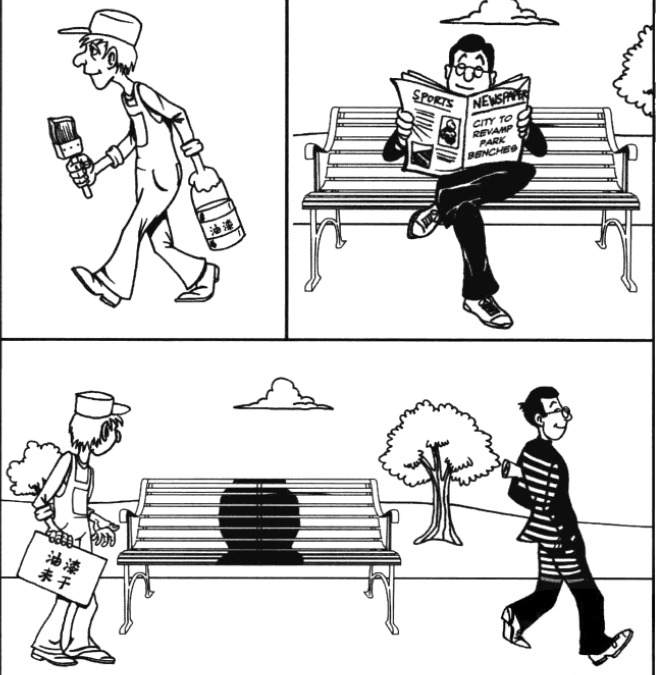
}

return \_instance;

}

}

### 公布可能的陷阱



当为一个函数或者类写注释时，可以这样的问自己:”这段代码有什么出人意料的地方吗？会不会被无用？“。基本上说就是你需要未雨绸缪，预料到别人使用你代码时可能遇到的问题。如：

//XXX: 因为调用外部邮件服务器发送邮件，所以耗时较长，请使用异步方法调用以防止UI卡死。

public void SendEmail(string to, string subject, string body)

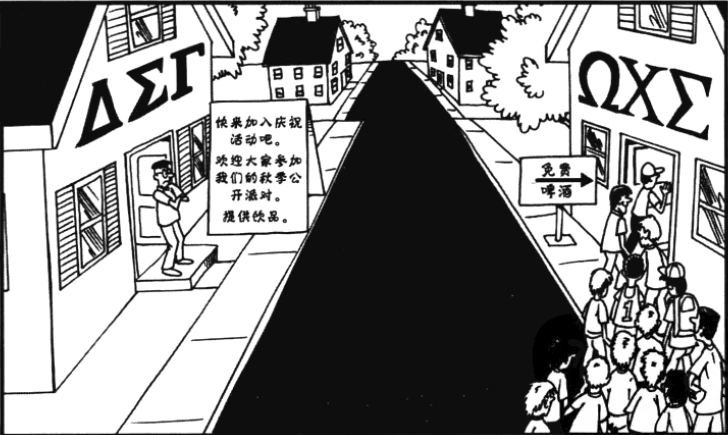
### 对于代码块总结性地注释

对于代码块的总结性注释可以使读者在深入细节之前就能得到该代码块的主旨，甚至有时候都可以直接跳过该代码块，从而可以快速准确的把握代码。

如读者看到：//下面代码使用了二分查找算法来快速的根据用户Id找到相应用户

那么他就可以快速理解下面代码的逻辑，否则自己看二分查找还是要用些时间的。

## 如何写好注释



### 避免使用不明确的代词

有些情况下，”it”, “this”等代词指代很容易产生歧义，最安全的方式是不要使用将所有可能产生歧义的代词替换成实际指代的词。

如：//Insert the data into the cache,but check if it's too big first.

”it”是指”data“还是”cache“? 在读完剩下的代码前谁也不知道指代的是谁。那还要注释做什么？替换成要指代的词后读者就可以直接了当的知道接下来的代码要做什么了。

//Insert the data into the cache,but check if the data is too big first.

### 精确描述方法的行为

注释一定要精确的描述方法的行为。避免由于注释不准确而造成的误调用。

如你写了一个方法统计文件中的行数

//Return the number of lines in this file

public long CountLinesInFile(string fileName)

上面的注释不是很精确，因为有很多定义行的方式，下面几种情况这个方法的返回值无法根据注释快速的判断出来。

1. “”(空文件)——0或1行？
2. “hello”­­——0或1行？
3. “hello\n”­­——1或2行？
4. “hello\n\r world\r”­­——2、3或4行？

假设该方法的实现是统计换行符的（\n）的个数，下面的注释就要比原来的注释更好些。

//Count how many newline symbols('\n') are this file

这条注释包含更多的信息。读者可以知道如果没有换行符，这个函数会返回0。读者还知道回车符（\r）会被忽略。

### 用输入输出例子来说明特殊的情况

对于注释来讲，一个精挑细选的例子比千言万语还要有效，而且更加直白有效，阅读速度更快。

如： /// <summary>

/// Remove the suffix/prefix of charsToRemove from the input source

/// </summary>

public string StripPrefixAndSuffix(string source, string charsToRemove)

这条注释不是很精确，因为它不能回答下面的问题

1. 是只有按charsToRemove中顺序的字符才会被移除，还是无序的charsToRemove也会被移除？
2. 如果在开头和结尾有多个charsToRemove会怎样？

而一个好例子就可以简单直白的回答这些问题：

/// <summary>

/// Example: StripPrefixAndSuffix("abbayabbazbaba","ab") returns "yababz"

/// </summary>

### 适当的使用具名函数

假设你看见这样的函数调用：

ConnectMailServer(100,false);

因为直接传入的是数值和bool值，使得这个调用很难理解，在这种情况下可以使用C#的具名函数以便代码阅读者快速知道这两个参数的含意。

假设函数是这样的：public void ConnectMailServer(int timeout\_s, bool useEncryption)

那我们可以这样使用具名函数:

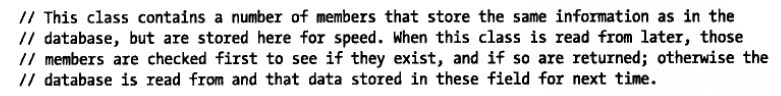
ConnectMailServer(timeout\_s: 100, useEncryption: false);

代码阅读者一眼就能看出这两个参数的作用。

### 使用通用专有名词

代码的阅读者和调用者都是程序员，程序员间有大量的专有名词可以用来描述一些模式和解决方案，使用这些词，可以简单明了的阐述你代码的含意。

假设你原来的注释是这样的：



那么你完全可以简单的说：

//This class acts as a **caching layer** to the database.

程序员一看caching layer就明白这个类是做什么的了。而且就算新手不明白，你的注释也很难给他讲明白的，还不如给他个关键词，让他自己去Google或向别人请教。这样的效果比你的注释会好的多。

### 更新代码时记得更新注释

再好的注释也会随着内容的更改而变得越来越没有意义，有时候甚至会对读者造成误导，产生不必要的bug。所以在更改代码后，记得要更新所更改代码的注释，使其表达最新代码的含意。

### 只有能让别人读懂的注释才是合格的注释

当自己不确定自己的注释是否合格时，请周围的同事读下你的注释，看他读完注释后说出的想法是否是你想要表达的，是否有信息遗漏和误解等。

# 格式

格式的统一使用可以使代码清晰、美观、方便阅读。为了不影响编码效率，在此只作如下规定：

## 长度

1. 一个文件最好不要超过500行(除IDE自动生成的类)。
2. 一个文件必须只有一个命名空间，严禁将多个命名空间放在一个文件里。
3. 一个文件最好只有一个类。
4. 如果超过500行，考虑拆分类或者使用Partial 类将类按照功能拆分。
5. 一个方法的代码最好不要超过50行，如果超过考虑将里面的逻辑封装成函数。

## 空格、空行

空行的使用以使代码清晰为为基本原则。空行影响程序的运行，但可以使代码看起来清晰，增加可读性，因此可以适当的使用。

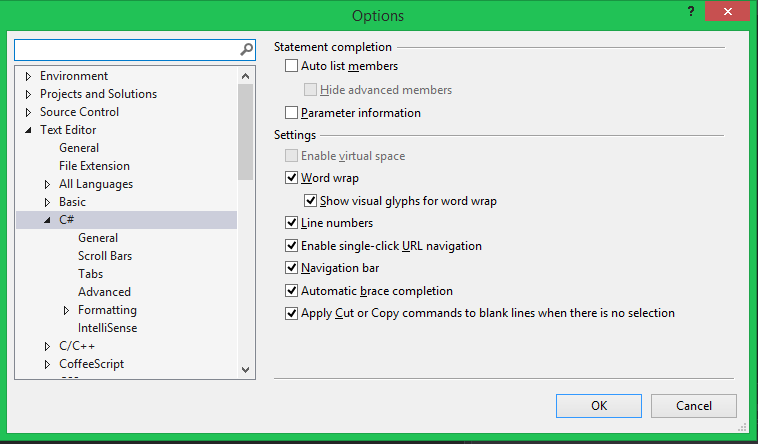
1. 方法与方法之间有两个空行
2. 函数内部变量声明与函数内部逻辑之间有一个空行。
3. 函数内部一个逻辑完成后要有一个空行，然后再写下一个逻辑

## 换行

换行掌握的原则是不要使一行代码特别长，以方便读者快速阅读。

1. 请团队所有成员开启Visual Studio自动换行功能。

Tools->Options->Text Editor->C#->Check the Word wrap CheckBox



1. 在不产生严重歧义或增加阅读难度的情况下，尽量控制在80列以内，不能超过120列，以方便阅读和代码的打印。
2. 另外换行的原则是，在逗号后，操作符前换行，换行与首行要保持一个单位的缩进。
3. “{”和“}”必须单独在一行上，且上下对齐。
4. If,while,do,try-catch-finaly 等语句后必须有{},即使里面只有一条语句或语句为空，且遵循“{”和“}”必须单独在一行上上的规定。

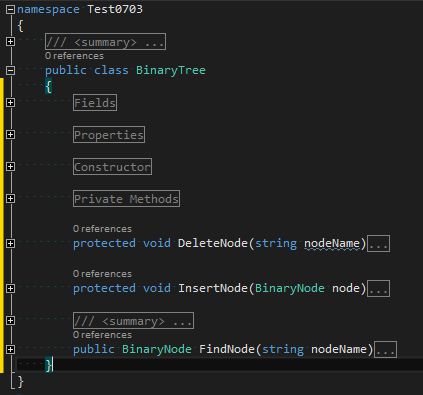
## Region的使用

Region的使用可以明确代码块的范围并提供相应注释，因此提倡大家尽量使用。

同时Region也可以用来组织代码结构，我们使用以下Region约定来使组织类：

1. 一个类应该用Region划分出Fields、Properties、Constructor、Private Methods和 Protected & Public Methods 五个区域。并按照上述顺序从上到下排列。
2. 其中Fields、Properties、Constructor、Private Methods必须放在Region。而Protected & Public Methods 必须放在外面。Protected & Public Methods 是代码阅读者查找最频繁的内容，这样做可以方便代码阅读者使用Ctrl+M, O时能够最快速度定位到自己想要查找的内容。

一个好的类结构第一层展开时应该如下图所示：



## 使用VS的Format Document功能

一个类或者方法编写完成后，必须使用Visual Studio 自带的“Format Document”的功能进行排版后才能Check In.

# Reference

1. [C# Coding Conventions](http://msdn.microsoft.com/en-us/library/ff926074.aspx)
2. NET设计规范约定惯用法与模式(第2版)
3. 编写可读性代码的艺术
4. 重构—改善既有代码的设计
5. 高效程序员的45个习惯
6. 代码整洁之道