# Google C#编程规范

# 目录

# 1.目的

当团队的所有开发人员都在同一个代码上工作时，也就是代码集体拥有的情况，大家都不希望别人改变代码的外观以适应他们自己的风格。因此，通过在项目之初建立一个编码标准，便于提高代码的可读性和可维护性，便于检查、减少出错概率，有助于项目组成员间交流，保证一致性、统一性。就可以增加团队的生产率和沟通效果。

# 2.适用范围

1. 本规范主要以C#为开发语言的规范；
2. 由于本规范是为撰写程序而设计，所以适用于一切有关程序撰写的工作事项。对于具体的每个项目，可能需要对之进行裁剪和补存。
3. 适用人员：软件工程专业的学生；
4. 适用产品：以C#编写的程序。

# 3.命名规范

### 1，命名的基本约定

像C#这样的面向对象语言允许开发人员采用叙述性的方式较灵活地命名类、方法、字段等。对于这些类型的事物，如果可以更清楚地描述事物的含义，应该使用完整的单词，不要使用缩略语，要尽可能的详细的命名，如果一定要使用，就谨慎使用。同时，应该保留一个标准缩写的列表，并且在使用时保持一致。 对常见缩略词，两个字母的缩写要采用统一大小写的方式（示例：ioStream，getIOStream）；多字母缩写采用首字母大写，其他字母小写的方式（示例： getHtmlTag）；避免使用长名字（最好不超过 15 个字母）。 避免使用相似或者仅在大小写上有区别的名字。

  .NET有两种主要类型的字母大小写形式：Pascal和camel。对于Pascal字母大小写形式来说，所创建名称的第一个单词的第一个字母是大写的，该名称中的后续单词也是如此，例如ThisIsPascalCase。而对于camel字母大小写形式来说，所创建名称的第一个单词的第一个字母是小写的，而该名称中的后续单词使用大写的字母，例如thisIsCamelCase。

下表描述了不同类型标识符的大小写规则：

|  |  |  |
| --- | --- | --- |
| 标识符 | 大小写 | 示例 |
| 命名空间 | Pascal | namespace Com.Techstar.ProductionCenter |
| 类型 | Pascal | public class DevsList |
| 接口 | Pascal | public interface ITableModel |
| 方法 | Pascal | public void UpdateData() |
| 属性 | Pascal | Public intLength{…} |
| 事件 | Pascal | public event EventHandler Changed; |
| 私有字段 | Camel | private string fieldName; |
| 非私有字段 | Pascal | public string FieldName； |
| 枚举值 | Pascal | FileMode{Append} |
| 参数 | Camel | public void UpdateData(string fieldName) |
| 局部变量 | Camel | string fieldName; |

### 2.各种标示符类型的命名约定

　1>  程序集命名

　　实验室名称（Lab）+ 项目名称 + 模块名称（可选），例如：

　　　　中心服务器程序集：Lab.SeverCenter；

　　　　中心服务器业务逻辑程序集：Lab.SeverCenter.Business；

　　2>  命名空间命名

　　采用和程序集命名相同的方式：实验室名称（Lab）+ 项目名称 + 模块名称。另外，一般情况下建议命名空间和目录结构相同。例如：

　　　　中心服务器：Lab.SeverCenter；

　　　　中心服务器下的用户控件：Lab.SeverCenter.UserControl；

　　　　中心服务器业务逻辑：Lab.SeverCenter.Business；

　　　　中心服务器数据访问：Lab.SeverCenter.Data；

　　3>  程序集和DLL

　　l  大多数情况下，程序集包含全部或部分可重用库，且它包含在单个动态链接库(DLL) 中。

　　l  一个程序集可拆分到多个DLL 中，但这非常少见，在此准则中也没有说明。

　　l  程序集和DLL 是库的物理组织，而命名空间是逻辑组织，其构成应与程序集的组织无关。

　　l  命名空间可以且经常跨越多个程序集。可以考虑如下模式命名DLL：

      　　  <Company>.<Component>.dll

   　　     例：Lab.SeverCenter.dll

　　4> 类和接口命名

　　l  类的名字要用名词；

　　l  避免使用单词的缩写，除非它的缩写已经广为人知，如HTTP。

　　l  接口的名字要以字母I开头。保证对接口的标准实现名字只相差一个“I”前缀，例如对IComponent接口的标准实现为Component；

　　l  泛型类型参数的命名：命名要为T或者以T开头的描述性名字，例如：

　　　　public class List<T>

　　　　public class MyClass<Tsession>

　　l  对同一项目的不同命名空间中的类，命名避免重复。避免引用时的冲突和混淆；

　　5> 方法命名

　　l  第一个单词一般是动词；

　　l  如果方法返回一个成员变量的值，方法名一般为Get+成员变量名，如若返回的值是bool变量，一般以Is作为前缀。另外，如果必要，考虑用属性来替代方法；

　　l  如果方法修改一个成员变量的值，方法名一般为：Set + 成员变量名。同上，考虑用属性来替代方法。

　　6> 变量命名

　　l  按照使用范围来分，我们代码中的变量的基本上有以下几种类型，类的公有变量；类的私有变量（受保护同公有）；方法的参数变量；方法内部使用的局部变量。　　　　这些变量的命名规则基本相同，见标识符大小写对照表。区别如下：

　　　　a)   类的公有变量按通常的方式命名，无特殊要求；

　　　　b)   类的私有变量采用两种方式均可：采用加“m”前缀，例如mWorkerName;

　　　　c)   方法的参数变量采用camalString，例如workerName；

　　l  方法内部的局部变量采用camalString，例如workerName。

　　l  不要用\_或&作为第一个字母；

　　l  尽量要使用短而且具有意义的单词；

　　l  单字符的变量名一般只用于生命期非常短暂的变量：i,j,k,m,n一般用于integer；c,d,e一般用于characters；s用于string

　　l  如果变量是集合，则变量名要用复数。例如表格的行数，命名应为：RowsCount；

　　l  命名组件要采用匈牙利命名法，所有前缀均应遵循同一个组件名称缩写列表

### 3  组件名称缩写列表

　　缩写的基本原则是取组件类名各单词的第一个字母，如果只有一个单词，则去掉其中的元音，留下辅音。缩写全部为小写。

|  |  |  |
| --- | --- | --- |
| 组件类型 | 缩写 | 例子 |
| Label | Lbl | lblNote |
| TextBox | Txt | txtName |
| Button | Btn | btnOK |
| ImageButton | Ib | ibOK |
| LinkButton | Lb | lbJump |
| HyperLink | Hl | hlJump |
| DropDownList | Ddl | ddlList |
| CheckBox | Cb | cbChoice |
| CheckBoxList | Cbl | cblGroup |
| RadioButton | Rb | rbChoice |
| RadioButtonList | Rbl | rblGroup |
| Image | Img | imgBeauty |
| Panel | Pnl | pnlTree |
| TreeView | Tv | tvUnit |
| WebComTable | Wct | wctBasic |
| ImageDateTimeInput | Dti | dtiStart |
| ComboBox | Cb | cbList |
| MyImageButton | Mib | mibOK |
| WebComm.TreeView | Tv | tvUnit |
| PageBar | Pb | pbMaster |

# 4．缩进

我们可以将Visual Studio 配置为使用制表符或者空格来进行缩进，同时还可设置缩进的字符单位。日常受用的规范使用4个空格的缩进单位。

    代码的行长对于可读性也非常重要。试着将每一行的代码控制在80个字符之内。当每行的字符不超过80个，而又需要换行时，请遵循下述规则：

    ·在操作符前断开

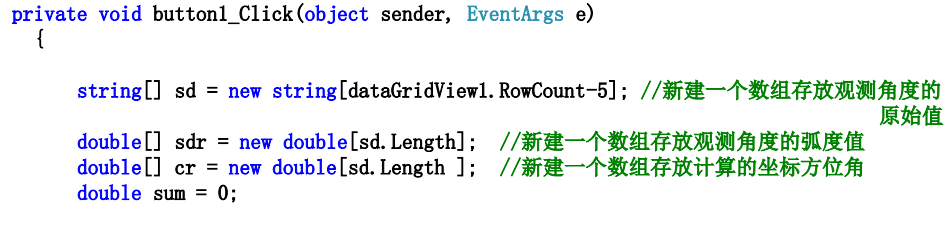
    ·在逗号后断开

·对齐括号

# 5．声明

在声明类字段、实例字段或者局部变量时，每行只声明一个。而当在一起进行几个声明时，需对准字段或者变量的名称。对于局部变量来说，应该在声明变量时就进行初始化，除非您在初始化变量时，还需要执行一些其他的动作，例如计算。

例如：

者方法的顶部，这样就可以使未来对声明的查看变得更容易。该规则的一个例外就是for循环内局部变量的声明和初始化。

例如：

for (inti = 0; i<d.Length; i++) //将度分秒存入这个双精度数值数组中

d[i] = Convert.ToDouble(ss[i]);

double sign = d[0] >= 0.0 ? 1.0:-1.0;//判断角度值是否为负值

double rad =0;

if (d.Length == 1) //根据数组的长度进行判断计算

rad = Math.Abs(d[0]) \* Math.PI / 180; //将度取绝对值，并转换为弧度

else if (d.Length == 2) rad = (Math.Abs(d[0]) + d[1] / 60) \* Math.PI /180; //将度取绝对值，分化为度，并最终转换为弧度

else rad = (Math.Abs(d[0])+ d[1]/60+d[2]/60/60)\*Math.PI/180;

# 6．语句

    C#中有多种类型的语句。每一行代码包含的内容不应该超过一个语句。

    对于if、if-else、和if else-if else 语句来说，总是使用大括号，而对于for、foreach、while、和do-while语句来说，也总是使用大括号（大括号两部分都应该独占一行）。

在return语句中，一般不要使用括号，除非为了使返回值更加明显。

下面是使用switch语句的格式：

switch (expression)

{

    case constant-expression:

    statements;

    break;

    default:

    statements;

    break;

}

而try-catch-finally语句则采用下述格式：

try

{

    statements;

}

catch (exception)

{

    statements;

}

finally

{

    statements;

}

If的语句格式：

if ()

else if ()

else7.    代码注释

# 7     代码注释

## .1    代码注释约定

所有的方法和函数都应该以描述这段代码的功能的一段简明注释开始（方法是干什么）。这种描述不应该包括执行过程细节（它是怎么做的），因为这常常是随时间而变的，而且这种描述会导致不必要的注释维护工作，甚至更糟—成为错误的注释。代码本身和必要的嵌入注释将描述实现方法。

当参数的功能不明显且当过程希望参数在一个特定的范围内时，也应描述传递给过程的参数。被过程改变的函数返回值和全局变量，特别是通过引用参数的那些，也必须在每个过程的起始处描述它们。

## 2     模块头部注释规范

以一个物理文件为单元的都需要有模块头部注释规范，例如：C#中的.cs文件

用于每个模块开头的说明，主要包括：（粗体字为必需部分，其余为可选部分）

1.文件名称(File Name)：此文件的名称

2.功能描述(Description)：此模块的功能描述与大概流程说明

3.数据表(Tables)：所用到的数据表，视图，存储过程的说明，如关系比较复杂，则应说明哪些是可擦写的，哪些表为只读的。

4.作者(Author)：

5.日期(Create Date)：

6.参考文档(Reference)(可选)：该档所对应的分析文档，设计文檔。

7.引用(Using) (可选)﹕开发的系统中引用其它系统的Dll、对象时，要列出其对应的出处，是否与系统有关﹙不清楚的可以不写﹚，以方便制作安装档。

8.修改记录(Revision History)：若档案的所有者改变，则需要有修改人员的名字、修改日期及修改理由。

9.分割符：\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* (前后都要)

## 3     方法注释规范

　　1>  C# 提供一种机制，使程序员可以使用含有XML 文本的特殊注释语法为他们的代码编写文档。在源代码文件中，具有某种格式的注释可用于指导某个工具根　　　　      据这些注释和它们后面的源代码元素生成XML。具体应用当中，类、接口、属性、方法必须有<Summary>节，另外方法如果有参数及返回值，则必须有　　　　　　   <Param>及<Returns>节。示例如下：

　　　　/// <summary>

　　　　/// …

　　　　/// </summary>

　　　　/// <param name=””></param>

　　　　/// <returns></returns>

　　2>  事件不需要头注解，但包含复杂处理时（如：循环/数据库操作/复杂逻辑等），应分割成单一处理函数，事件再调用函数。

　　3>  所有的方法必须在其定义前增加方法注释。

　　4>  方法注释采用 /// 形式自动产生XML标签格式的注释。

|  |  |  |
| --- | --- | --- |
| 标记 | 说明 | 备注 |
| <c> | 提供了一种将说明中的文本标记为代码的方法 |  |
| <code> | 提供了一种将多行指示为代码的方法 |  |
| <example> | 可以指定使用方法或其他库成员的示例。一般情况下，这将涉及到<code>标记的使用。 |  |
| <exception> | 对可从当前编译环境中获取的异常的引用。 |  |
| <include> | 得以引用描述源代码中类型和成员的另一文件中的注释。 |  |
| <list> | 用于定义表或定义列表中的标题行。 |  |
| <para> | 用于诸如<summary>、<remarks>或<returns>等标记内，使您得以将结构添加到文本中。 |  |
| <param> | 应当用于方法声明的注释中，以描述方法的一个参数。 |  |
| <paramref> | 提供了一种指示词为参数的方法。 |  |
| <permission> | 得以将成员的访问记入文档。 |  |
| <remarks> | 用于添加有关某个类型的信息，从而补充由<summary>所指定的信息。 |  |
| <returns> | 应当用于方法声明的注释，以描述返回值。 |  |
| <see> | 得以从文本内指定链接。 |  |
| <seealso> | 对可以通过当前编译环境进行调用的成员或字段的引用。 |  |
| <summary> | 应当用于描述类型或类型成员。 |  |
| <value> | 得以描述属性。 |  |

　5>  在公用类库中的公用方法需要在一般方法的注释后添加作者、日期及修改记录信息，统一采用XML标签的格式加注，标签如下：

　　　　<Author></Author>作者

　　　　<CreateDate></CreateDate>建立日期

　　　　<RevisionHistory>修改记录

　　　　　　<ModifyBy></ModifyBy>    修改作者

　　　　　　<ModifyDate></ModifyDate>   修改日期

　　　　　　<ModifyReason></ModifyReason>   修改理由

　　　　　　<ModifyBy></ModifyBy>    修改作者

　　　　　　<ModifyDate></ModifyDate>   修改日期

　　　　　　<ModifyReason></ModifyReason>   修改理由

　　　　　　<ModifyBy></ModifyBy>    修改作者

　　　　　　<ModifyDate></ModifyDate>   修改日期

　　　　　　<ModifyReason></ModifyReason>   修改理由

　　　　</RevisionHistory>

　　　　<LastModifyDate></LastModifyDate>最后修改日期

　　6>  一个代码文件如果是由一人编写，则此代码文件中的方法无需作者信息，非代码文件作者在此文件中添加方法时必须要添加作者、日期等注释。

　　7>  修改任何方法，必须要添加修改记录的注释

## 4     代码行注释规范

　　1>  如果处理某一个功能需要很多行代码实现，并且有很多逻辑结构块，类似此种代码应该在代码开始前添加注释，说明此块代码的处理思路及注意事项等

　　2>  注释从新行增加，与代码开始处左对齐

3>  双斜线与注释之间以空格分开，

4> 如果语句块（比如循环和条件分枝的代码块）代码太长，嵌套太多，则在其结束“｝”要加上注释，标志对应的开始语句。如果分支条件逻辑比较复杂，也要加上注释.

5>保持注释的简洁，不是任何代码都需要注释的，过多的注释反而会影响代码的可读性。

´ 注释**不要**包括其他的特殊字符.

# 8. 类型设计规范

**要**确保每个类型由一组定义明确，相互关联的成员组成，而不仅仅是一些无关功能的随 机集合；

## 8.1. 类型和命名空间

**要**用命名空间把类型组织成相关域的层次结构。例如：

界面层：Techstar.ProductionCenter；

业务逻辑层：Techstar.ProductionCenter.Business；

数据访问层：Techstar.ProductionCenter.Data；

´ **避免**过深的命名空间；

´ **避免**太多的命名空间；

## 8.2. 类型和接口的选择

1. **要**优先采用类而不是接口。

接口的缺点在于语义变化时改变困难。注意接口并不是协定，把协定和实现分开并非一 定用接口实现，用基类和抽象类同样可以表达；

2. **建议**使用抽象类而不是接口来解除协定与实现间的偶合；

3. **要**定义接口，来实现类似多重继承的效果；

精心定义接口的标志是一个接口只做一件事情。关键是接口的协定需要保持不变， 如果一个接口包含太多功能，那么这个胖接口产生变化的机会就会大得多。

## 8.3. 抽象类设计：

´ **不要**在抽象类中定义公有的或内部受保护的构造函数。因为抽象类无法实例化，所以这 种设计会误导用户；

**要**为抽象类定义受保护的构造函数或内部构造函数；

## 8.4. 静态类设计

静态类是一个只包含静态成员的类，它提供了一种纯面向对象设计和简单性之间的一个权衡，广泛用来提供类似于全局变量或一些通用功能。

**要**少用静态类。静态类应该仅用作辅助类；

´ **避免**把静态类当作杂物箱。每个静态类都应该有其明确目的；

**不要**在静态类中声明或覆盖实例成员；

## 8.5. 枚举设计

**要**用枚举来加强那些表示值的集合的参数，属性以及返回值的类型性；

**要**优先使用枚举而不是静态常量。例如：

//不好的写法

public static class Color

{

public static int Red = 0;

public static int Green = 1;

public static int Blue = 2;

}

//好的写法

public enum Color

{

Red,

Green,

Blue

}

´ **不要**把枚举用于开放的场合，例如操作系统的版本，朋友的名字等；

´ **枚举最后一个值不要加逗号；**

´ 枚举中**不要**提供为了今后使用而保留的枚举值；