# 蛮牛主办《刀塔传奇》开源项目

# C#编码规范

肖红 2015/01/09草拟

注意：该文档来着百度，内容可以经过讨论修改

# 1前言

## 1.1编码规范的重要性

在软件工程领域，源程序风格统一标志着可读性，可维护性，可重用性，是软件工程的重要组成部分。

编码规范对于程序员而言非常重要：

1. 规范的代码促进团队合作
2. 规范的代码可以降低维护成本
3. 规范的代码有助于代码审查
4. 养成规范代码的习惯，有助于程序员自身的成长

## 1.2规范制定原则

1.方便代码的交流和维护

2.不影响编码的效率，不与大众习惯冲突。

3.使代码更加美观，阅读更方便。

4.使代码的逻辑更清晰，更易于理解。

## 1.3术语定义

Pascal 大小写(火箭式)

将标识符的首字母和后面连接的每个单词首字母都大写。例如

BackColor

Camel 大小写(骆驼式)

标识符首字母小写，后面连接的单词的首字母都大写。例如

backColor

过去很多代码中全部采取下划线连接单词的方式已经过时了。当然需要匈牙利命名法进行界定的m\_和p\_等还是可以使用下划线。

# 2文件

## 2.1文件命名

1.文件名应该**能够反映它所对应的类名**。不能包含空格、标点、特殊符号，不使用预留关键字。

2.文件名遵从Pascal命名法，无特殊情况，扩展名小写。

3. 使用同一通用的文件扩展名： C#类 .cs

例如:ExampleClassName

IExampleInterface

MyNamespace

特殊情况，已有的遗留代码可暂时不遵守这个规则(以前的代码包层次已经固定，类似UI这种认为大写字母才能代表其意义的)。我们应该注意到开源代码基本都遵守以上3条要求，可参考PureMVC代码。

## 2.2文件注释

版权和版本的声明位于头文件和定义文件的开头（参见示例1-1）：

|  |
| --- |
| /\*\*  \* Copyright (c) 2014,龙谷网络科技有限公司  \*All rights reserved.  \*  \* 文件名称：FileName.cs  \* 简 述：*简要描述本文件的内容*  \* 创建标识：Lorry 2014/06/09  \* 修改标识：RonyZ2014/06/11  \* 修改描述：添加XX属性，修正XX功能。  \*/ |

示例1-1 版权和版本的声明

 文件功能描述只需简诉，具体详情在类的注释中描述。

标识有创建或修改人员的名字拼音或英文名加日期组成。如：

Lxt 2014/06/09

一天内有多个修改时，只需要添加一个修改标识和描述，但是需要在所有的代码修改处加上修改标识的注释。

# 3代码外观

## 文件结构

文件是基本上应该按以下顺序构成。不过也有人提出变量应该在靠近其使用方法处声明。

1. 版权声明（暂时没有可以最后一次性添加）
2. 使用命名空间的语句  
   using System;   
   using System.Threading;  
   …
3. 命名空间声明（）
4. 类或接口声明
5. 静态变量(static var )
6. 静态属性(static get set)
7. 静态方法(static function)
8. 构造函数
9. 变量（var）
10. 重载属性(override get set)
11. 属性(get set)
12. 重载方法(override function)
13. 方法(function)
14. 重载事件处理函数
15. 事件处理函数
16. 帮助类（一般都是内部用数据结构类VO或者存储管理类）。

## 3.1列宽

代码列宽控制在100个字符左右。

也不要写过于长的一句代码。

（以标准19寸显示器来说，一行最好不要超过100个字符，不然IDE中分屏显示无法看到两边完整的代码）。

为什么要看两屏代码？咕~~(╯﹏╰)b，多编程你就知道了。

## 3.2换行

当表达式超出规定列宽，遵循以下规则进行换行

1. 在逗号后换行
2. 在操作符前换行
3. 规则1优于规则2.

当以上规则会导致代码混乱的时候自己采取灵活的换行规则。

## 3.3缩进

缩进应该是每行一个Tab（4个空格），不要在代码中使用Tab字符。

Visual Studio.Net设置：工具->选项->文本编辑器->C#- >制表符->插入空格

## 3.4空行

空行是为了将逻辑上相关联的代码分块，以便提高代码的可阅读性。

有些代码规范中，会有要求空两行的情况，不过我个人不太认同，所以空行这里先不做强制要求。**以下仅供建议**

**使用两个空行的情况**

1. 接口和类定义之间
2. 枚举和类的定义之间
3. 类与类定义之间（但是一个文件中要避免放置多个类）

**使用一个空行的情况**

1. 方法与方法之间
2. 方法中不同的逻辑块之间
3. 方法中的返回语句与其他语句之间
4. 属性与方法，属性与字段，方法与字段之间。
5. 注释与它注释的语句间不空行

## 3.5空格

在以下情况中要使用到空格

1. 关键字和左括号“(”应该用空格隔开。  
   如：while (true)  
   **注意：**方法名和左括号“(”之间不要使用空格，这样有助于辨认代码中方法调用和关键字。
2. 多个参数用逗号隔开，每个逗号后面应该加一个空格。
3. 除了. 之外，所有的二元操作符都应使用空格与他们的操作数隔开。一元操作符、++及—与操作数间不需要空格。如：  
   a += c + d;  
   a = (a + b) / (c \* d);  
   while (d++ = s++)  
   {  
    n++;  
   }  
   PrintSize(“size is ” + size + “\n”);
4. 语句中表达式之间用空格隔开，如  
   for (expr1; expr2; expr3)

## 3.6括号 () 参考3.5空格

## 3.7花括号 {}

1. 左花括号”{”放于关键字或方法名的下一行，并与之对齐。如：  
   if (condition)  
   {  
   }  
     
   public int Add(int x, int y)  
   {  
   }
2. 左花括号要和右花括号对齐。
3. 通常情况下左花括号“{”单独成行，不与任何语句并列。
4. if、while、do语句后一定要使用{},即使{}中只有一条语句或为空(！！)。如  
   if (somevalue == 1)  
   {  
   somevalue = 2;  
   }
5. 在右花括号“}”后面建议添加一个注释，便于找到与之对应的{ 。如  
   while (1)  
   {  
    if (valid)  
    {  
    }//if valid  
    else  
    {  
    }//not valid  
   }//end forever

# 4注释

注释是软件可读性的具体表现。程序注释量一般占程序编码量的20%，软件工程要求不少于20%，可以一段程序的前面加一段注释，具有明确的处理逻辑。

良好的可读性是注释的金科玉律，对于开发者来说，基本原则应该是：让代码自己描述自己。虽然有人怀疑这是不喜欢注释的程序员所倡导的一场运动，但是无需注释的代码有很大的好处，这些代码更加容易理解，以至于让注释变得没有必要。

但因为我们生活在英语非母语的国家，要做到代码自注释这种最高境界不管对自己还是共同工作的伙伴，要求非常高。在未达到最高境界时，请老老实实的写额外注释。

对于游戏的编写，最好能够在类注释上额外写明，此class到底是为了完成哪一部分的策划案而实现的，这样对于代码审核、Bug查找、维护有莫大的好处。

## 概述：

1. 修改代码时，总是使代码周围的注释保持最新。
2. 在每个例程的开始，提供标准的注释样本以指示例程的用途、假设和限制很有帮助。  
   注释样本应该是解释它为什么存在和可以做什么的简短介绍。
3. 避免在代码行的末尾添加注释；行尾注释使代码更难阅读。不过在批注变量声明时，行尾注释是合适的；请将所有行尾注释用制表位对齐。（基于MonoDevelop对中文的支持还是放弃行尾注释吧！）
4. 避免杂乱的注释，如一整行星号，应该使用空白将注释同代码分开。
5. 避免在块注释的周围加上印刷框，这样看起来可能很漂亮，但是难于维护。
6. 在代码上传之前，移除所有临时或无关的注释，以避免在日后的维护工作中产生混乱。
7. 如果需要用注释来解释复杂的代码节，请检查代码以确定是否应该重写它。  
   尽一切可能不注释难以理解的代码，而应该重写它。  
   尽管一般不应该为了使代码更简单(便于使用)而牺牲性能，但必须保持性能和可维护性之间的平衡。
8. 在编写注释时使用完整的句子。注释应该阐明代码，而不应该增加多义性。
9. 在编写代码时就注释，因为以后很可能没有时间这样做。另外，如果有机会复查已编写的代码，在今天看来很明显的东西六周以后或许就不明显了。
10. 避免多余的或不适当的注释，如幽默的非主要的备注。
11. 使用注释来解释代码的意图。它们不应作为代码的联机翻译。
12. 注释代码中不十分明显的任何内容。
13. 为了防止问题反复出现，对错误修复和解决方法代码总是使用注释，尤其是在团队环境中。
14. 对由循环和逻辑分支组成的代码使用注释。这些是帮助源代码读者的主要方面。
15. 在整个应用程序中，使用具有一致的标点和结构的统一样式来构造注释。
16. 用空白将注释同注释分隔符分开。在没有颜色提示的情况下查看注释时，这样做会使注释很明显且容易被找到。
17. 在所有的代码修改处加上[修改标识](#修改标识)的注释。
18. 为了层次清晰，在闭合的右花括号后注释该闭合所对应的起点。  
     namespace Langchao.Procument.Web  
    {  
    } // namespace Langchao.Procument.Web

不强求，但是当代码比较长，特别是有多重嵌套时，这样做会更加清晰，便于阅读。

## 4.1文档型注释

该类注释采用.Net已定义好的Xml标签来标记，在声明接口、类、方法、属性、字段都应该使用该类注释，以便代码完成后直接生成代码文档，让别人更好的了解代码的实现和接口。如

///<summary>MyMethod is a method in the MyClass class.

///<para>Here's how you could make a second paragraph in a description.

///<see cref="System.Console.WriteLine"/>

///for information about output statements.

///</para>

///<seealsocref="MyClass.Main"/>

///</summary>

public static void MyMethod(int Int1)

{

}

## 4.2类c注释（as3中也有）

该类注释用于

1 不再使用的代码。

2 临时测试屏蔽某些代码。

用法

/\*

[修改标识]

[修改原因]

. . . (the source code )

\*/

## 4.3单行注释

该类注释用于

1 方法内的代码注释。如变量的声明、代码或代码段的解释。注释示例：

// 注释语句

privateint number;

或

private int number; // 注释语句

2 方法内变量的声明或花括号后的注释, 注释示例：

if ( 1 == 1) // always true

{

statement;

} // always true

## 4.4注释标签

|  |  |  |
| --- | --- | --- |
| 标签 | 用法 | 作用 |
| <c> | <c>*text*</c>  *text* 希望将其指示为代码的文本。 | 为您提供了一种将说明中的文本标记为代码的方法。使用 [<code>](vclrfcode.htm) 将多行指示为代码 |
| <para> | <para>*content*</para>  content段落文本。 | 用于诸如 [<remarks>](vclrfremarks.htm) 或 [<returns>](vclrfreturns.htm) 等标记内，使您得以将结构添加到文本中。 |
| <param> | <param name='*name*'>*description*</param>  name 为方法参数名。将此名称用单引号括起来 (' ')。 | 应当用于方法声明的注释中，以描述方法的一个参数。 |
| <paramref> | <paramref name="*name*"/>  *name*  要引用的参数名。将此名称用双引号括起来 (" ")。 | <paramref> 标记为您提供了一种指示词为参数的方法。可以处理 XML 文件，从而用某种独特的方法格式化该参数。 |
| <see> | <see cref="*member*"/>  cref = "*member*"对可以通过当前编译环境进行调用的成员或字段的引用。编译器检查到给定代码元素存在后，将 *member* 传递给输出 XML 中的元素名。必须将 *member* 括在双引号 (" ") 中。 | 使您得以从文本内指定链接。使用 [<seealso>](vclrfseealso.htm) 指示希望在“请参阅”一节中出现的文本。 |
| <seealso> | <seealsocref="*member*"/>  cref = "*member*"对可以通过当前编译环境进行调用的成员或字段的引用。编译器检查到给定代码元素存在后，将 *member* 传递给输出 XML 中的元素名。必须将 *member* 括在双引号 (" ") 中 | 使您得以指定希望在“请参阅”一节中出现的文本。使用 [<see>](vclrfsee.htm) 从文本 |
| <example> | <example>*description*</example>  *description*  代码示例的说明。 | 使用 <example> 标记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>](vclrfcode.htm) 标记的使用。 |
| <code> | <code>*content*</code>  content 为希望将其标记为代码的文本。 | 记为您提供了一种将多行指示为代码的方法。使用[<c>](vclrfc.htm)指示应将说明中的文本标记为代码 |
| <summary> | <summary>*description*</summary>  此处description 为对象的摘要。 | 应当用于描述类型成员。使用 [<remarks>](vclrfremarks.htm) 以提供有关类型本身的信息。 |
| <exception> | <exception cref="*member*">*description*</exception>  cref = "*member*"对可从当前编译环境中获取的异常的引用。编译器检查到给定异常存在后，将 *member* 转换为输出 XML 中的规范化元素名。必须将 *member* 括在双引号 (" ") 中。  *description* 说明。 | <exception> 标记使您可以指定类能够引发的异常。 |
| <include> | <include file='*filename*' path='*tagpath*[@*name*="*id*"]' />  filename 包含文档的文件名。该文件名可用路径加以限定。将 *filename* 括在单引号中 (' ')。  *Tagpath：filename* 中指向标记名的标记路径。将此路径括在单引号中 (' ')。  *name* 注释前边的标记中的名称说明符；名称具有一个 *id*。  *id*  位于注释之前的标记的 id。将此 id 括在双引号中 (" ")。 | <include> 标记使您得以引用描述源代码中类型和成员的另一文件中的注释。这是除了将文档注释直接置于源代码文件中之外的另一种可选方法。  <include> 标记使用 XML XPath 语法。有关自定义 <include> 使用的方法，请参阅 XPath 文档。 |
| <list> | <list type="bullet" | "number" | "table">  <listheader>  <term>*term*</term>  <description>*description*</description>  </listheader>  <item>  <term>*term*</term>  <description>*description<*/description>  </item>  </list>  *term*定义的项，该项将在 *text* 中定义。  *description*目符号列表或编号列表中的项或者 *term* 的定义。 | <listheader>块用于定义表或定义列表中的标题行。定义表时，只需为标题中的项提供一个项。  列表中的每一项用 <item> 块指定。创建定义列表时，既需要指定 *term* 也需要指定 *text*。但是，对于表、项目符号列表或编号列表，只需为 *text* 提供一个项。  列表或表所拥有的 <item> 块数可以根据需要而定。 |
| <permission> | <permission cref="*member*">*description*</permission>  cref = "*member*" 对可以通过当前编译环境进行调用的成员或字段的引用。编译器检查到给定代码元素存在后，将 *member* 转换为输出 XML 中的规范化元素名。必须将 *member* 括在双引号 (" ") 中。  *description*成员的访问的说明。 | <permission> 标记使您得以将成员的访问记入文档。**System.Security.PermissionSet** 使您得以指定对成员的访问。 |
| <remarks> | <remarks>*description*</remarks>  *description* 成员的说明。 | <remarks> 标记是可以指定有关类或其他类型的概述信息的位置。[<summary>](vclrfsummary.htm) 是可以描述该类型的成员的位置。 |
| <returns> | <returns>*description*</returns>  *description* 返回值的说明。 | <returns> 标记应当用于方法声明的注释，以描述返回值。 |
| <value> | <value>*property-description*</value>  *property-description* 属性的说明。 | <value> 标记使您得以描述属性。请注意，当在 Visual Studio .NET 开发环境中通过代码向导添加属性时，它将会为新属性添加 [<summary>](vclrfsummary.htm) 标记。然后，应该手动添加 <value> 标记以描述该属性所表示的值。 |
|  |  |  |

# 5声明（申明）

## 每行声明数

一行只建议作一个声明，并按字母顺序排列。如

int level; //推荐

int size; //推荐

int x, y; //不推荐

## 初始化

**建议在变量声明时就对其做初始化。**

## 位置

变量在最接近实用时做声明。如：

voidMyMethod()

{

int int1 = 0; // beginning of method block

if (condition)

{

int int2 = 0; // beginning of "if" block

...

}

}

不过也有一个例外

for (inti = 0; i<maxLoops; i++)

{

...

}

应避免不同层次间的变量重名，如

int count;

...

voidMyMethod()

{

if (condition)

{

int count = 0; // 避免

...

}

...

}

## 类和接口的声明

1 方法名与其后的左括号间没有任何空格。（关键字后有）

2 左花括号“{”出现在声明的下行并与之对齐，单独成行。

3方法间用一个空行隔开。

# 命名规范

## 6.1命名概述

名称应该说明“什么”而不是“如何”。

通过避免使用公开基础实现（它们会发生改变）的名称，可以保留抽象层（用于简化复杂性）。

例如，可以使用 GetNextStudent()，而不是 GetNextArrayElement()。

命名原则：

1. 选择正确名称时的困难可能表明需要进一步分析或定义项的目的。
2. 使名称足够长以便有一定的意义，并且足够短以避免冗长。
3. 唯一名称在编程上仅用于将各项区分开。
4. 表现力强的名称是为了帮助人们阅读；因此，提供人们可以理解的名称是有意义的。

不过，请确保选择的名称符合适用语言的规则和标准。

以下几点是推荐的命名方法:

1、避免容易被主观解释的难懂的名称，如：方法名AnalyzeThis()，或者属性名xxK8。这样的名称会导致多义性。

2、在类属性的名称中包含类名是多余的，如Book.BookTitle。而是应该使用Book.Title。

3、只要合适，在变量名的末尾或开头加计算限定符（Avg、Sum、Min、Max、Index）。

4、在变量名中使用互补对，如 min/max、begin/end 和 open/close。

5、布尔变量名应该包含Is，这意味着 Yes/No 或 True/False 值，如fileIsFound。

6、在命名状态变量时，避免使用诸如Flag的术语。状态变量不同于布尔变量的地方是它可以具有两个以上的可能值。不是使用documentFlag，而是使用更具描述性的名称，如documentFormatType。（**此项只供参考**）

7、即使对于可能仅出现在几个代码行中的生存期很短的变量，仍然使用有意义的名称。仅对于短循环索引使用单字母变量名，如i或j。

可能的情况下，**尽量不要使用原义数字或原义字符串，如For i = 1 To 7。**

**而是使用命名常数，如For i = 1 To NUM\_DAYS\_IN\_WEEK以便于维护和理解**。

## 6.2大小写规则

大写

标识符中的所有字母都大写。仅对于由两个或者更少字母组成的标识符使用该约定。

例如：

System.IO

System.Web.UI

下表汇总了大写规则，并提供了不同类型的标识符的示例

|  |  |  |
| --- | --- | --- |
| 标识符 | 大小写 | 示例 |
| 类 | Pascal | AppDomain |
| 枚举类型 | Pascal | ErrorLevel |
| 枚举值 | Pascal | FatalError |
| 事件 | Pascal | ValueChange |
| 异常类 | Pascal | WebException  注意 总是以 Exception 后缀结尾。 |
| 只读的静态字段 | Pascal | RedValue |
| 接口 | Pascal | IDisposable  注意 总是以 I 前缀开始。 |
| 方法 | Pascal | ToString |
| 命名空间 | Pascal | System.Drawing |
| 属性 | Pascal | BackColor |
| 公共实例字段 | Pascal | RedValue  注意 很少使用。属性优于使用公共实例字段。 |
| 受保护的实例字段 | Camel | redValue  注意 很少使用。属性优于使用受保护的实例字段。 |
| 私有的实例字段 | Camel | redValue |
| 参数 | Camel | typeName |
| 方法内的变量 | Camel | backColor |
|  |  |  |

## 6.3缩写

能避免就避免！

代码清晰永远比减少打字重要，何况现在开发环境的智能感应系统已经flash builder有Alt+/ 来提示，vs.net有VA。

但是也要记住一些标准化的通用缩写：

auto, impl, info, num, min, max, nav(navigation), regexp(regular expression), util(utility)等等。

如果有其他缩写，我们需要内部商议通过并加入内部缩写对照表（稍后建立）。

现在已有的包括 dup(duplicate)，mgr(manager)，but(button)等等。

为了避免混淆和保证跨语言交互操作，请遵循有关区缩写的使用的下列规则：

1 不要将缩写或缩略形式用作标识符名称的组成部分。例如，使用GetWindow，而不要使用GetWin。

2不要使用计算机领域中未被普遍接受的缩写。

3 在适当的时候，使用众所周知的缩写替换冗长的词组名称。例如，用 UI 作为 User Interface 缩

写，用 OLAP 作为 On-line Analytical Processing 的缩写。

4 使用缩写时，对于超过两个字符长度的缩写请使用 Pascal 大小写。例如，使用HtmlButton或HTMLButton。

但是，仅有两个字符的缩写应当大写。如，System.IO，而不是System.Io。

5尽量不要在参数名称中使用缩写。

## 6.4命名空间

1、命名空间时的一般性规则是使用公司名称，后跟技术名称和可选的功能与设计，如下所示。

CompanyName.TechnologyName[.Feature][.Design]

例如：

namespace Langchao.Procurement//浪潮公司的采购单管理系统

namespace Langchao.Procurement.DataRules//浪潮公司的采购单管理系统的业务规则模块

**现在我们使用 namespace Weedong.BX //维动科技，暗黑西游项目 black xy**

2、命名空间使用Pascal大小写，用逗号分隔开。

3、TechnologyName指的是该项目的英文缩写，或软件名。

4、命名空间和类不能使用同样的名字。

例如，有一个类被命名为Debug后，就不要再使用Debug作为一个名称空间名。

## 6.5类

1、使用 Pascal 大小写。

2、用名词或名词短语命名类。

3、使用全称避免缩写，除非缩写已是一种公认的约定，如URL、HTML

4、不要使用类型前缀，如在类名称上对类使用 C 前缀。例如，使用类名称FileStream，而不是CFileStream。

5、不要使用下划线字符( \_ )。

6、有时候需要提供以字母 I 开始的类名称，虽然该类不是接口。只要 I 是作为类名称组成部分的整个单词的第一个字母，这便是适当的。

例如，类名称IdentityStore是适当的。在适当的地方，使用复合单词命名派生的类。派生类名称的第二个部分应当是基类的名称。例如，ApplicationException对于从名为 Exception 的类派生的类是适当的名称，因为ApplicationException是一种Exception。

但在应用该规则时请进行合理的判断。例如，Button 对于从 Control 派生的类是适当的名称。尽管按钮是一种控件，但是将 Control 作为类名称的一部分将使名称不必要地加长。

public class FileStream

public class Button

public class String

## 6.6接口

以下规则概述接口的命名指南：

1、用名词或名词短语，或者描述行为的形容词命名接口。例如，接口名称IComponent使用描述性

名词。接口名称 ICustomAttributeProvider使用名词短语。名称IPersistable使用形容词。

2、使用 Pascal 大小写。

3、少用缩写。

4、给接口名称加上字母 I 前缀，以指示该类型为接口。在定义类/接口对（其中类是接口的标准

实现）时使用相似的名称。两个名称的区别应该只是接口名称上有字母 I 前缀。

5、不要使用下划线字符 (\_)。

6、当类是接口的标准执行时，定义这一对类/接口组合就要使用相似的名称。两个名称的不同之处

只是接口名前有一个I前缀。

以下是正确命名的接口的示例。

public interface IServiceProvider

public interface IFormatable

以下代码示例阐释如何定义IComponent接口及其标准实现 Component 类。

public interface IComponent

{

// Implementation code goes here.

}

public class Component: IComponent

{

// Implementation code goes here.

}

## 6.7属性(Attribute)

应该总是将后缀Attr（或Attribute）添加到自定义属性类。以下是正确命名的属性类的示例。

public class ObsoleteAttribute

{

}

## 6.8枚举(Enum)

枚举 (Enum) 值类型从Enum类继承。以下规则概述枚举的命名指南：

1对于Enum类型和值名称使用 Pascal 大小写。

2少用缩写。

3不要在Enum类型名称上使用Enum后缀。

4对大多数Enum类型使用单数名称，但是对作为位域的Enum类型使用复数名称。

5总是将FlagsAttribute添加到位域Enum类型。

## 6.9参数

以下规则概述参数的命名指南：

1、使用描述性参数名称。参数名称应当具有足够的描述性，在大多数情况下,应该可以通过参数的名称及类型确定它的含义。

2、对参数名称使用 Camel 大小写。

3、使用描述参数含义的名称，而不要使用描述参数类型的名称。开发工具将提供有关参数类型的信息（现在智能提示已经很强了）。因此，通过描述意义，可以更好地使用参数的名称。少用基于类型的参数名称，仅在适合的地方使用。

4、不要使用保留的参数。如果保留的参数是专用参数，可以在未来的版本中公开它们；如果在类库的未来版本中需要更多的数据，请为方法添加新的重载。

5、不要给参数名称加匈牙利命名法的前缀。

以下是正确命名的参数的示例。

Type GetType(string typeName)

string Format(string format, args() As object)

## 6.10方法(函数)

以下规则概述方法的命名指南：

1使用动词或动词短语命名方法。

2使用 Pascal 大小写。

3以下是正确命名的方法的实例。

RemoveAll()

GetCharArray()

Invoke()

## 6.11属性(property)

以下规则概述属性的命名指南,属性其实就是返回/设置 指定变量的函数：

1使用名词或名词短语命名属性。

2使用 Pascal 大小写。

3不要使用匈牙利语表示法。

4考虑用与属性的基础类型相同的名称创建属性。

例如，如果声明名为 Color 的属性，则属性的类型同样应该是 Color。

请参阅本主题中后面的示例。

以下代码示例阐释正确的属性命名。

public class SampleClass

{

public Color BackColor

{

// Code for Get and Set accessors goes here.

}

}

以下代码示例阐释提供其名称与类型相同的属性。

publicenum Color

{

// Insert code for Enum here.

}

public class Control

{

public Color Color

{

get

{

// Insert code here.

}

set

{

// Insert code here.

}

}

}

以下代码示例不正确，原因是 Color 属性是 Integer 类型的。

publicenum Color

{

// Insert code for Enum here.

}

public class Control

{

publicint Color

{

// Insert code here

}

}

在不正确的示例中，不可能引用 Color 枚举的成员。Color.Xxx将被解释为访问一个成员，

该成员首先获取 Color 属性（ C# 中为int类型）的值，然后再访问该值的某个成员（该成

员必须是 System.Int32 的实例成员）。

## 6.12事件

以下规则概述事件的命名指南：

1、对事件处理程序名称使用EventHandler后缀。

2、指定两个名为 sender 和 e 的参数，sender 参数表示引发事件的对象，sender 参数始终是object 类型的，即使在可以使用更为特定的类型时也如此。与事件相关联的状态封装在名为 e 的事件类的实例中，对 e 参数类型使用适当而特定的事件类。

3、用EventArgs后缀命名事件参数类。

4、考虑用动词命名事件。

5、使用动名词（动词的“ing”形式）创建表示事件前的概念的事件名称，用过去式表示事

件后。例如，可以取消的 Close 事件应当具有 Closing 事件和 Closed 事件。不要使用

BeforeXxx/AfterXxx命名模式。

6、不要在类型的事件声明上使用前缀或者后缀。例如，使用 Close，而不要使用OnClose。

7、通常情况下，对于可以在派生类中重写的事件，应在类型上提供一个受保护的方法（称为

OnXxx）。此方法只应具有事件参数 e，因为发送方总是类型的实例。

以下示例阐释具有适当名称和参数的事件处理程序。

public delegate void MouseEventHandler(object sender, MouseEventArgs e);

以下示例阐释正确命名的事件参数类。

public class MouseEventArgs : EventArgs

{

int x;

int y;

publicMouseEventArgs(int x, int y)

{

this.x = x;

this.y = y;

}

publicint X

{

get

{

return x;

}

}

publicint Y

{

get

{

return y;

}

}

}

## 6.13常量（const）

以下规则概述常量的命名指南：

所有单词大写，多个单词之间用 "\_" 隔开。如

publicconst string PAGE\_TITLE = "Welcome";

## 6.14字段

以下规则概述字段的命名指南：

1、private、protected 使用 Camel 大小写。

2、public 使用 Pascal 大小写。

3、拼写出字段名称中使用的所有单词。仅在开发人员一般都能理解时使用缩写。字段名称不

要使用大写字母。下面是正确命名的字段的示例。

classSampleClass

{

stringurl;

stringdestinationUrl;

}

4、不要对字段名使用匈牙利语表示法。**好的名称描述语义而非类型**。

5、不要对字段名或应用前缀。具体说来，不要对字段名称应用前缀来区分。例如，应用 g\_ 前缀是不正确的。

6、对预定义对象实例使用公共静态只读字段。如果存在对象的预定义实例，则将它们声明为对象本身的公共静态只读字段。使用 Pascal 大小写，原因是字段是公共的。下面的代码

示例阐释公共静态只读字段的正确使用。

publicstruct Color

{

public static readonly Color Red = new Color(0x0000FF);

public Color(intrgb)

{

// Insert code here.

}

public Color(byte r, byte g, byte b)

{

// Insert code here.

}

public byte RedValue

{

get

{

// Insert code here return Color;

}

}

}

## 6.15静态字段

以下规则概述静态字段的命名指南：

1、使用名词、名词短语或者名词的缩写命名静态字段。

2、使用 Pascal 大小写。

3、对静态字段名称使用匈牙利命名法表示法前缀。（唯一建议用匈牙利命名法的地方）

4、建议尽可能使用静态属性而不是公共静态字段。

## 6.16集合

集合是一组组合在一起的类似的类型化对象，如哈希表、查询、堆栈、字典和列表，集合命名建议用复数。

## 6.17措词

避免使用与常用的 .NET 框架命名空间重复的类名称。例如，不要将以下任何名称用作类名称：

System、Collections、Forms 或 UI。有关 .NET 框架命名空间的列表，请参阅类库。另外，避免使用和以下关键字冲突的标识符。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AddHandler | AddressOf | Alias | And | Ansi |
| As | Assembly | Auto | Base | Boolean |
| ByRef | Byte | ByVal | Call | Case |
| Catch | CBool | CByte | Cchar | CDate |
| CDec | CDbl | Char | Cint | Class |
| CLng | CObj | Const | Cshort | CSng |
| CStr | CType | Date | Decimal | Declare |
| Default | Delegate | Dim | Do | Double |
| Each | Else | ElseIf | End | Enum |
| Erase | Error | Event | Exit | ExternalSource |
| False | Finalize | Finally | Float | For |
| Friend | Function | Get | GetType | Goto |
| Handles | If | Implements | Imports | In |
| Inherits | Integer | Interface | Is | Let |
| Lib | Like | Long | Loop | Me |
| Mod | Module | MustInherit | MustOverride | MyBase |
| MyClass | Namespace | New | Next | Not |
| Nothing | NotInheritable | NotOverridable | Object | On |
| Option | Optional | Or | Overloads | Overridable |
| Overrides | ParamArray | Preserve | Private | Property |
| Protected | Public | RaiseEvent | ReadOnly | ReDim |
| Region | REM | RemoveHandler | Resume | Return |
| Select | Set | Shadows | Shared | Short |
| Single | Static | Step | Stop | String |
| Structure | Sub | SyncLock | Then | Throw |
| To | True | Try | TypeOf | Unicode |
| Until | volatile | When | While | With |
| WithEvents | WriteOnly | Xor | Eval | extends |
| instanceof | package | var |  |  |

# 语句

## 7.1每行一个语句

每行最多包含一个语句。如

a++; //推荐

b--; //推荐

a++; b--; //不推荐

## 7.2复合语句

复合语句是指包含"父语句{子语句;子语句;}"的语句，使用复合语句应遵循以下几点

1子语句要缩进。

2左花括号“{”在复合语句父语句的下一行并与之对齐，单独成行。

3 即使只有一条子语句要不要省略花括号“ {}”。如

while(d+ = s++)

{

n++;

}

## 7.3 return 语句

return语句中不使用括号，除非它能使返回值更加清晰。如

return;

returnmyDisk.size();

return (size ? size : defaultSize);

## 7.4 if、 if-else、if else-if 语句

if、 if-else、if else-if 语句使用格式

if (condition)

{

statements;

}

if (condition)

{

statements;

}

else

{

statements;

}

if (condition)

{

statements;

}

else if (condition)

{

statements;

}

else

{

statements;

}

## 7.5for、foreach语句

for 语句使用格式

for (initialization; condition; update)

{

statements;

}

空的 for 语句（所有的操作都在initialization、condition 或 update中实现）使用格式

for (initialization; condition; update); // update user id

foreach语句使用格式

foreach (object obj in array)

{

statements;

}

**注意**1在循环过程中不要修改循环计数器。

2对每个空循环体给出确认性注释。

## 7.6 while 语句

while 语句使用格式

while (condition)

{

statements;

}

空的 while 语句使用格式

while (condition);

## 7.7do - while 语句

do - while 语句使用格式

do

{

statements;

} while (condition);

## 7.8switch - case 语句

switch - case 语句使用格式

switch (condition)

{

case 1:

statements;

break;

case 2:

statements;

break;

default:

statements;

break;

}

注意：

1、语句switch中的每个case各占一行。

2、语句switch中的case按字母顺序排列。

3、为所有switch语句提供default分支。  
 4、所有的非空 case 语句必须用 break; 语句结束。

## 7.9 try - catch 语句

try - catch 语句使用格式

try

{

statements;

}

catch (ExceptionClass e)

{

statements;

}

finally

{

statements;

}

## 7.10 using 块语句

using 块语句使用格式

using (object)

{

statements;

}

## 7.11 goto语句

goto语句使用格式

goto Label1:

statements;

Lable1:

statements;

# 8 控件命名规则

## 8.1 命名方法

控件名简写+英文描述，英文描述首字母大写

## 8.2 主要控件名简写对照表

控件名 简写 控件名 简写

Label lbl TextBox txt

Button btn LinkButton lnkbtn

ImageButton imgbtn DropDownList ddl

ListBox lst DataGrid dg

DataList dl CheckBox chk

CheckBoxList chklsRadioButton rdo

RadioButtonList rdolt Image img

Panel pnl Calender cld

AdRotator ar Table tbl

RequiredFieldValidator rfv CompareValidator cv

RangeValidator rv RegularExpressionValidator rev

ValidatorSummary vs CrystalReportViewer rptvew

# 9 其他

## 9.1 表达式

1 避免在表达式中用赋值语句

2避免对浮点类型做等于或不等于判断

## 9.2 类型转换

1尽量避免强制类型转换。

2如果不得不做类型转换，尽量用显式方式。

## 9.3 其他建议

使用C#预定义的类型，而不是使用System命名空间中的别名

采用object 而不是 Object

采用 string 而不是 String

采用int而不是 Int32

对泛型类型采用大写字母。

//正确方法

public class LinkedList<K, T>

{…}

//避免使用：

public class LinkedList<KeyType, DataType>

{…}

编码实践

1. 避免同一个文件夹中放置多个类
2. 避免在一个文件内写多余500行的代码
3. 避免写超过25行代码的方法
4. 避免写超过5个参数的方法（使用结构）
5. 注释时，在注释符号后加入一个空格，便于阅读
6. 不要手动修改任何机器生成的代码
7. 尽量使用partial classes特性，以提高可维护性。
8. 注释应该只说明一些前提假设，算法的内部信息等内容。（不要写废注释，注意变量可读性，少见的单词添加注释依然是必要的，要添加到英文命名表中）
9. 避免对只读变量使用const关键字，应该直接使用readonly关键字。
10. 避免利用返回值作为函数的错误代码
11. 避免自定义异常类
12. 避免friend assemblies，因为会增加程序之间的耦合（不过有些时候还是必要的）
13. 避免显示指定枚举值
14. 避免使用三元操作符，因为大部分时候不好理解。
15. 总是使用以0为基数的数组。
16. 总是使用一个for循环显示的初始化一个引用成员的数组。  
    public class MyClass  
    {}  
    constintArraySize=100;  
    MyClass[] array=new MyClass[ArraySize];  
    for (int index = 0; index <array.Length; index ++)  
    {  
     array[index] = new MyClass();  
    }
17. 避免利用返回的Boolean值作为条件语句，把返回值赋给一个局部变量，然后再检测。
18. 使用属性来代替public 或 protected类型的成员变量
19. 不要使用继承下来的new操作符，使用override关键字覆写new的实现
20. 避免显示类型转换，使用 as 关键字安全转换到另一个类型。
21. 在调用一个代理前，总是检测它是否为null
22. 不要提供public 的成员变量。改用Event Accessor
23. 接口和类中仿佛和属性的比应该在2:1左右
24. 避免只有一个成员的接口
25. 努力保证一个接口有3~5个成员
26. 不要让一个接口中的成员数量超过20个（12是更加实际的限制）
27. 避免在接口中包含事件
28. 当使用抽象类时，提供一个接口。
29. 从不假设一个类型支持某个接口，总在使用前询问  
    （用as ImyInterface然后判断null的方式）
30. 在switch语句中的default中提供一个断言debug.Assert(false);
31. 除了在一个构造函数中调用其他的构造函数之外，不要使用this关键字
32. 不要使用base关键字访问基类成员，除非你在调用一个基类构造函数时，要……

## 性能优化建议

1. 在处理枚举时，对于数组有特别的优化，所以，当枚举的集合是一个数组时，性能会好些
2. 在计算密集型的程序中，结构的创建仍然比类要高效的多，另外，属性和字段的访问其性能基本相当。
3. 移除代码中的任何字符串连接，因为这会给GC留下大量垃圾。
4. 用简单的“for”循环代替“foreach”循环。由于某些原因，每个“foreach”循环的每次迭代会生成24字节的垃圾内存。一个简单的循环迭代10次就可以留下240字节的垃圾内存。
5. 用“if (go.CompareTag (“Enemy”)”来代替“if (go.tag == “Enemy”)”。在一个内部循环调用对象分配的标签属性以及拷贝额外内存，这是一个非常糟糕的做法。
6. 对象库很棒，我们为所有动态游戏对象制作和使用库，这样在游戏运行时间内不会动态分配任何东西，不需要的时候所有东西反向循环到库中。
7. 不使用LINQ命令，因为它们一般会分配中间缓器，而这很容易生成垃圾内存。
8. 在这一情景中四处移动对象要求来自脚本代码的调用进入引擎代码，这样我们就会在游戏玩法代码的一个帧中缓存某一对象的转换需求，并一次仅向引擎发送一个请求，以便减少调用开销。这种模式也适用于其他相似的地方，而不仅局限于移动和旋转对象。
9. 将引用本地缓存到元件中会减少每次在一个游戏对象中使用“GetComponent”获取一个元件引用的需求，这是调用本地引擎代码的另一个例子。
10. 将物理模拟时间步设置到最小化状态。在我们的项目中就不可以将让它低于16毫秒。
11. 减少角色控制器移动命令的调用。移动角色控制器会同步发生，每次调用都会耗损极大的性能。我们的做法是缓存每帧的移动请求，并且仅运用一次。
12. 修改代码以免依赖“ControllerColliderHit”回调函数。
13. 面对性能更弱的设备，要用skinned mesh代替physics cloth。cloth参数在运行表现中发挥重要作用，如果你肯花些时间找到美学与运行表现之间的平衡点，就可以获得理想的结果。
14. 在物理模拟过程中不要使用ragdolls，只有在必要时才让它生效。
15. 要谨慎评估触发器的“onInside”回调函数，在我们的项目中，我们尽量在不依赖它们的情况下模拟逻辑。
16. 使用层次而不是标签。我们可以轻松为对象分配层次和标签，并查询特定对象，但是涉及碰撞逻辑时，层次至少在运行表现上会更有明显优势。更快的物理计算和更少的无用分配内存是使用层次的基本原因。
17. 千万不要使用Mesh对撞机。
18. 最小化碰撞检测请求（例如ray casts和sphere checks），尽量从每次检查中获得更多信息。
19. AI逻辑（例如能见度检查等）会生成大量物理查询。可以让AI更新循环设置低于图像更新循环，以减少CPU负荷。
20. 使用细节层次的定制关卡将远处的敌人AI关闭。
21. 移动平台和障碍，当它们远去时其物理碰撞机也会关闭。
22. Unity内置的“动画挑选”系统可以用来关闭未被渲染对象的动画。
23. 所有关卡内的粒子系统也可以使用同样的禁用机制。
24. 共享游戏对象材料，令其在Unity中处于静止状态，可以让它们绑定在一起，由此产生的简化绘图调用是呈现良好移动运行性能的重要元素。
25. 方形纹理以及两者功率的合理压缩是必不可少的步骤。
26. 我们的动画师尽力让不同角色共享动画。
27. 不同设备要使用不同的纹理大小，尤其是UI和大型背景中的纹理。
28. 我们要确保未使用的资产不会载入内存。我们必须迟一点在项目中找到仅被一个预制件实例引用，并且从未完全载入内存中实例化的资产。
29. 我们应该重建一些资产的生周期管理。例如，调整主菜单资产的加载/卸载时间，或者关卡资产、游戏音乐的有效期限。
30. 每个关卡都要有根据其动态对象需求而量身定制的特定对象库，并根据最小内存需求来优化。对象库可以灵活一点，在开发过程中包含大量对象，但知道游戏对象需求后就要具体一点。
31. 保持声音文件在内存的压缩状态也是必要之举。