**概述**

因为VS的C#编辑环境提供了自动格式纠正，我们的规范可以与微软的格式保持一致，这些规则不赘述，着重讲述“自动格式”之外的内容。

本文参考了：[(G1261)C++编码管理办法](http://wiki/pages/viewpage.action?pageId=28412071)、[(G1115)LPC编码管理办法](http://wiki/pages/viewpage.action?pageId=51482073)、[C sharp 编程规范](http://wiki/pages/viewpage.action?pageId=511844199)

在选取规范时采取的原则：

· 容易理解和执行

· 尽量大众化

**关于自动格式**

在VS环境下编码，只要注意不要破坏自动格式（包括缩进、空格等），顺着给定格式写即可。微软的格式与我们的要求是相同的，这套格式要求的具体说明见：[(G1115)LPC编码管理办法#排版要求](#%28G1115%29LPC%E7%BC%96%E7%A0%81%E7%AE%A1%E7%90%86%E5%8A%9E%E6%B3%95-%E6%8E%92%E7%89%88%E8%A6%81%E6%B1%82)

在Unity的环境下，需要配置编码格式为微软格式。

wps264A.tmp.png

配置方法（点此展开）：

**命名规则**

1. 变量、参数、字段：

标识符的首字母小写，而每个后面连接的单词的首字母都大写（骆驼命名法）。例如：backColor

2. 方法、属性、类、接口、事件、命名空间、文件名：

标识符首字母和后面连接的单词的首字母都大写（Pascal命名法）。例如：BackColor

3. 文件名要和类名相同，一般情况下一个类一个文件。

4. 常量：全用大写字母，单词间以下划线分割。如：GROUP\_SIZE

5. 缩写：约定俗成的缩写可以使用，如：msg，proc，dlg等，但并非一目了然的缩写要谨慎使用，定义处必须有注释说明。

6. 含义准确：注意用词准确，契合真正的含义。

对函数的命名特别注意动词的准确性，比如：CheckXXX、GetXXX不应对数据做任何修改。

7. 避免不同层次间的变量重名。

8. int count;

9. ...

10. void MyMethod()

11. {

12.     if (condition)

13.     {

14.         int count = 0;  // 避免

15.         ...

16.     }

17. }

18. 在业务逻辑里，如果遇到策划设计的具有中国特色的词汇，宁可用汉语拼音，也不要用难以理解的英语翻译。

**函数**

1. 避免函数有太多的参数，参数个数尽量控制在5个以内。如果参数太多，在使用时容将参数类型或顺序搞错。

2. 函数的功能要单一，不要设计多用途的函数。

3. 函数的规模尽量限制在200行以内（不包括注释和空行）。

4. 尽量避免函数带有"记忆"功能。相同的输入应当产生相同的输出。带有"记忆" 功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种"记忆状态 "。这样的函数既不易理解又不利于测试和维护。在C/C++语言中，函数的 static 局部变量是函数的"记忆"存储器。建议尽量少用static局部变量，除非必需。

5. 功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并 到上级函数中，而不必单独存在。

说明：模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函 数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

6. 减少函数本身或函数间的递归调用。

说明：递归调用特别是函数间的递归调用（如A->B->C->A），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

7. 要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

8. 调用其它函数一定要检查返回值是否出错。同时，自身出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

**安全**

1. 用ASSERT来检查程序正常运行时不应发生但在调测时有可能发生的非法情况；用ASSERT确认函数的参数；用ASSERT保证没有定义的特性或功能不被使用。

ASSERT应放在函数入口处。

2. int Function(object a, object b)

3. {

4.     ASSERT(a != null);

5.     ASSERT(b != null);

6.     ....

7. }

8. 尽量用乘法或其它方法代替除法，特别是浮点运算中的除法，但是要注意精度溢出的问题。整数乘除法在保证不溢出的情况下，先除后乘。

9. 保持单线程逻辑，除了调用异步I/O接口之外，不能使用多线程。

10. 协程尽量不用。在必须要用的场合，将逻辑注释清楚。

11. 不要"捕捉了异常却什么也不做"。要么不捕捉，捕捉了就要妥善处理（至少要完整的报出来）。

如果程序崩溃，至少可以发现问题。如果隐藏了一个异常，会让问题更隐蔽。

12. 尽量避免强制类型转换。如果不得不转换，尽量使用as关键字。

13. 对所有的用户输入，必须进行合法性检查。

14. 程序与环境或状态发生关系时（文件能否逻辑锁定、打印机是否联机等），必须主动去处理发生的意外事件。

对于明确的错误，要有明确的容错代码提示用户，使用Try Throw Catch。

15. 使throw而不是return错误值的方式报错。

16. 除非为了和其它语言进行互动，否则绝不要使用不安（unsafe）的代码。

17. 不能手动修改任何机器生成的代码（除非只是测试）。

**效率**

1. 除内核级代码，禁止追求程序的效率而牺牲可靠性、健壮性、可读性。

业务逻辑、脚本级代码一般不需要过分考虑运行效率，可读性反而更重要。

2. 在优化程序的效率时，应当先找出限制效率的"瓶颈"，不要在无关紧要之处优化。

3. 先提高算法级优化，再提高编码级优化。

4. 不要追求短小的代码，因为较短的代码并不意味是高效的代码。

5. 在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数。如果循环体内存在逻辑判断，并且循环次 数很大，宜将逻辑判断移到循环体的外面。

6. for (row = 0; row < 100; row++)

7. {

8.     for (col = 0; col < 5; col++)

9.     {

10.         sum += a[row][col];

11.     }

12. }

13. 可以改为如下方式，以提高效率：

14. for (col = 0; col < 5; col++)

15. {

16.     for (row = 0; row < 100; row++)

17.     {

18.         sum += a[row][col];

19.     }

20. }

21. 生成和构建一个长的字符串时，一定要使用StringBuilder，而不用string。

22. 尽可能不用Hashtable，可用Dictionary替代。Dictionary也尽量不要使用。

**注释**

|  |  |
| --- | --- |
|  | **注释不是负担，而是工具**  新手一般有这样的问题：如果一段代码自己比较轻松完成，这时一般记得写注释。但如果编码花费了很多脑力，可能就自动省略了写注释这个步骤……但事实是：  1. 可以轻松完成的代码，没有注释的必要。  2. 代码越难写，越需要注释让人读懂。  注释的作用是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码。  在编码过程中，还可以帮助程序员整理思路，要逐步养成习惯：  1. 编码的同时就完善注释：不能依靠事后补充，一则可能更“没有时间”、二则起不到整理思路的作用。  2. 先写注释再编码：注释如同作文的提纲。  3. 越费脑力的代码，越要认真考虑注释。 |

wps265A.tmp.png

1. 不要“直译”代码。其言自明的代码可以不注释。

|  |  |
| --- | --- |
|  | **错误示范：如下注释没有任何意义**  // 如果ob.is\_monster()返回真，那么返回  if (ob.is\_monster())      return; |

wps265B.tmp.png

2. 变量的定义和分支语句（条件分支、循环语句等），除了其言自明的情况，必须写注释。

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

3. 微妙的逻辑、较长的算式、复杂的模块，必须在注释中说明思路或目的。

4. // 将水的渲染层级调低，不会挡住其它效果

5. water.GetComponentInChildren<MeshRenderer>().sortingOrder = -1;

6.

7. // 计算这个船与准心的角度偏差

8. angle = Mathf.Rad2Deg \* Mathf.Acos(Vector3.Dot(from2enemy, from2target) / (enemyDistance \* targetMag));

9. 对于复杂逻辑用到的关键变量应该有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

10. 如果代码必须用复杂注释才能解释，要考虑是否应该重写代码。

11. 代码跟注释要保持同步，修改代码需同时修改注释。

12. 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。

13. 代码提交前，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。

14. 当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

15. while (id > 0)

16. {

17.     ob = GetObjectById(id);

18.     if (ob != null && ob.active)

19.     {

20.         ...

21.         ... // 此处省略50行

22.         ...

23.     } // end if (ob != null && ob.active)

24. } // end while (id > 0)

25. 注释用中文。

26. 除了特别多行的情况可以用/\* \*/，一般一律使用//。

27. //后面需要且只需要跟一个空格。

28. 所有文件头部必须有注释，并且本工程内的其他文件的头部注释应该保持一致

在任何文件中：至少包括：作者，创建时间，本文件负责的内容。

29. // BugProducer.cs

30. // Created by lij, 2016.7.5

31. // 文件概要说明

32. 注释必须与其上面的代码用空行隔开。

|  |  |
| --- | --- |
|  | **错误示范：如下例子，显得代码过于紧凑**  // 第一段代码的注释  第一段代码  // 第二段代码的注释  第二段代码 |

wps266C.tmp.png

33. 正确示范：

34. // 第一段代码的注释

35. 第一段代码

36.

37. // 第二段代码的注释

38. 第二段代码

**排版**

1. 每个函数定义结束之后都必须加一行空行。

2. 在一个函数体内，逻辑上密切相关的语句作为一个段落(statements)，语句之间不加空行，段落间则必须有一空行。

3. <空行>

4. while (condition)

5. {

6.     statements1;

7.     <空行>

8.     if (condition)

9.     {

10.         statements2;

11.     }

12.     else

13.     {

14.  statements3;

15.     }

16.     <空行>

17.     statements4;

18. }

19. 在定义变量时，邻近变量都垂直对齐，相应的注释也对齐。

20. class Wnd

21. {

22.    Rect            mRectWnd;              // Wnd所占的区域

23.    bool            mShow;                 // 是否显示图片

24.    bool            mEnable;               // 是否可用（有时可能需要 显示图片，但是产可用）

25.    Point           mPosition;             // 窗口左上角在画布上的位置

26.

27.    LONG\_TYPE\_DEFINITION    mVarx;         // 长变量，不必和上面一批对齐

28.    LONG\_TYPE\_DEFINITION    mVary;         // 注释

29.    LONG\_TYPE\_DEFINITION    mVarz;         // 注释

30. };

31. 空函数内部不加空行。

32. public void MyClass()

33. {

34. }

35. 不得有连续两行以上的空行。

36. 每行尾不得有空格。

37. 把引用的系统的namespace和自定义或第三方的用一个换行把它们分开。

**其它**

1. 所有为了测试目的而临时增加的代码必须用 ////---- 加标记，在提交时删除这些带标记的代码

2. 所有BUG排查代码及待优化代码 必须用 // TODO

3. 变量与false、0或空的比较：

bool类型的写法：

4. if (b)

5.     ...

6. if (!b)

7.     ...

非bool类型的写法

if (a == 0)

    ...

if (object != null)

    ...

8. 代码中意义不明显的数值要定义为常量。

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。

9. 如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。

10. const float RADIUS = 100;

11. const float DIAMETER = RADIUS \* 2;

12. 一行只建议作一个声明。

13. int level;   // 推荐

14. int size;    // 推荐

15. int x, y;    // 不推荐

16. 必要时使用Enum，不要用数字或字符串来指示离散值。

17. 每行一个语句。

18. 避免采用多赋值语句，如：x = y = z;

19. 使用括号清晰地表达算术表达式和逻辑表达式的运算顺序。

如：将 x=a\*b/c\*d 写成 x=(a\*b/c)\*d，可避免阅读者误解为x=(a\*b)/(c\*d)。

20. 目录结构中要反应出namespace的层次。

21. 可重用性要求

o 重复使用的完成相对独立功能的算法或代码应抽象为类。

o 类应考虑OO思想，减少外界联系，考虑独立性或封装性。

o 避免让你的代码依赖于运行在某个特定地方的程序集。