C++注释规范

版本： 1.0

制定部门：　质量管理部

目录

1 说明

本文档用于规范 C++ 代码中注释的编写。 规范中提出的多数注释格式都来源于文档生成工具 doxygen，所以遵从本规范进行注释的 C++ 代码都可以使用 doxygen 生成美观一致的代码文档。

同时另一方面， 美观绝非衡量文档质量的唯一标准。 文档内容准确与否， 是否充分，以及语言组织是否清晰流畅，这些都是决定一份文档质量的重要标准。遗憾的是，这些标准当中有不少需要通过主观加以判断，很难进行明确的规范。

所以我们将尽可能的提供明确的评判标准，同时，本规范中也不可避免的提出了一些比较主观的注释要求或是建议， 这些要求或是建议多数都来自于众多先驱多年的开发经验。遵循它们不仅有助于生成一份美观的代码文档。更重要，依照这些要求和建议来编写注释， 能够有效的帮助开发者在早期就反省自己设计的合理性，同时也为编写单元测试提供更多的帮助。

2 注释种类

2.1 重复代码

重复性注释只是用不同文字把代码的工作又描述一次。 他除了给读者增加阅读量外， 没有提供更多信息。

2.2 解释代码

解释性注释通常用于解释复杂、 敏感的代码块。 在这些场合他们能派上用场， 但通常正是因为代码含混不清， 才体现出这类注释的价值。 如果代码过于复杂而需要解释， 最好是改进代码，而不是添加注释。使代码清晰后再使用概述性注释或者意图性注释。

2.3 代码标记

标记性注释并非有意留在代码中， 他提醒开发者某处的工作未做完。 在实际工作中， 我们经常会使用这些注释作为程序骨架的占位符，或是已知 bug 的标记。

2.4 概述代码

概述性注释是这么做的： 将若干代码行的意思以一两句话说出来。 这种注释比重复性注释强多了， 因为读者读注释能比读代码更快。 概述性注释对于要修改你代码的其他人来说尤

其有用。

2.5 代码意图说明

意图性注释用来指出要解决的问题，而非解决的方法。意图性注释和概述性注释没有明显的界限，其差异也无足轻重，都是非常有效的注释。

2.6 传达代码无法表达的信息

某些信息不能通过代码来表达，但又必须包含在源代码中。这种注释包括版权声明、作者、日期、版本号等杂项信息；与代码设计有关的注意事项；优化注记；和 doxygen 要求的注释等等。

3 注释原则

编写一份好的注释，并不比编写一段好的代码更容易，以至于我们不得不特别强调编写注释应该遵循的三个原则：

3.1 站在读者的立场编写注释

我们的代码将会面对很多不同的读者。其中一类是代码复审者。他们希望看到的是准确的注释说明，小心地编写注释避免出现可笑的错误是最重要的。

接下来的读者是代码的用户，这部分人大多并不关心代码中使用了何等绝妙的高超技术。事实上，代码能干什么，以及如何正确的使用更令他们感兴趣。所以当我们在对那些可能面向用户的地方注释时，多多考虑一下用户的实际需要并非多余。

最后一类，怎么说呢，作为代码的维护者，他们整天要和成打的代码打交道，渴望能够用最少的时间找到目标代码。所以，如果代码中有一些提纲挈领的注释，可以供他们用作节约代码阅读时间的索引，毫无疑问，这会令大家都工作得更加开心。

3.2 注释无法取代良好的编程风格

引用《代码大全》中的观点，在代码层文档中起主要作用的因素并非注释，而是良好的编程风格。编程风格包括良好的程序结构、直率易懂的方法、有意义的变量名和函数名、具名常量、清晰的布局、以及最低复杂度的控制流及数据结构。关于命名方面的规范参见《 C++ 命名规范》。

虽然本篇规范讨论的重点并非编程风格，但是无论什么时候，我们都应该明白，对于精心编写的代码而言，注释不过是美丽衣裳上的小饰物而已。

3.3 好注释能在更高抽象层次上解释我们想干什么

在编写注释这一问题上，我们经常犯的一个错误，就是将代码已经清楚说明的东西换种说法再写一次。 尤其是如果我们为代码添加中文注释的时候，简单的等同于将英文译作中文， 那么这样的注释能够给他人带来的好处微乎其微，更多时候是徒增阅读负担以及维护工作量。

再次重申，好注释不是重复代码，或是解释代码。它们会让其意图清晰。注释应该能在更高抽象层次上解释我们想干什么。

4 规范细则

4.1 文件注释规范

代码文件注释是典型的用于传达代码无法表达信息类的注释， 所以没有太多值得讨论的地方。一份文件注释至少需要包括以下内容：

1) 代码文件的文件名，占据注释的第一行并用 @file 加以标记
2) 文件内容的简要描述，占据单独一行并用 @brief 加以标记
3) 文件作者的公司 email 地址，占据单独一行并用 @author 加以标记
4) 文件活动年份[1]，占据单独一行并用 @date 加以标记

也可以包含以下一些可选内容：

1) 文件内容的详细描述，紧接简要描述之后并与其以一空行隔开
2) 文件版本， 占据单独一行并用 @version 加以标记， 使用 maj.min 格式表示， 其中 maj 为主版本号， min 为次版本号。

文件注释需要放到每个代码文件的起始位置。 这里提供了一个完整的格式示例， 多数情况下可以完全照搬其格式。

```
/**
 * @file Shape.h
 * @brief 包含图形基类 Shape 的定义
 *
 * 本示例文件中包含了类 Shape 的定义，以及此处省去的十余字 …
 *
 * @author wenruiyun@ms.com
 * @version 1.0
 * @date 2006
*/
```

请注意，正如我们《 C++编码规范》中提到的。在 C++世界中，我们不应该使用 C 风格的/* … */块注释，而应该使用 C++风格的 //行注释。是的，我们应该毫不动摇的恪守这一条

[1] 所谓“文件活动年份”，可以把它理解为文件从创建年份到最后一次维护的年份，比如 2004-2006。

款。但是，由于我们希望借助 doxygen 帮助我们生成漂亮的文档，如果我们在对文件注释的风格选取上毫不通融的话，doxygen 会为我们生成一大片可笑的空白：

```
00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011 #pragma once
00012
00013 #include "utility.h"
00014
00018 namespace MyGraph
00019 {
```

为了避免这样的尴尬场面，我们只好破例留下了这段 C 风格文件注释。好在这是唯一必须使用 C 风格注释才能令 doxygen 满意的地方，跨过文件注释之后，我们仍然应该虔诚的遵循 C++的行注释风格。

4.2 名字空间注释规范

名字空间是一个好东西，编写名字空间注释的要点是，不要试图把这个名字空间介绍的面面俱到。简单的介绍设置这个名字空间的意图或是完全不要注释都是不错的选择。

如果决定要为名字空间添加注释，有一点非常重要： 无论如何不要试图给同一个名字空间提供两份不同的注释，不管这些注释是否位于同一个代码文件中。一个有效的做法是，一开始不用给名字空间添加任何注释，先用 doxygen 生成文档，发现哪个名字空间缺少文档，再挑选一个最合适的代码文件进行名字空间注释。下面是一个名字空间注释的格式示例：

```
/// @brief 包含一系列图形元素
///
///  在 MyGraph 名字空间中，包含一系列图形元素，所有的这些图形元素都派生于
/// Shape 类
namespace MyGraph
{
```

结合前面文件注释的例子我们可以看到，@brief 概述项之后，必须有一个空行用于分割随后的详细描述。本文中所有的概述和详述项都遵循这一规则，后面将不再累述。

4.3 类定义注释规范

类定义注释主要有以下几个用途：

- 说明该类的设计方法。有的信息通过编码细节“逆向工程”是难以获得的，如果概述性注释能够提供这些信息，将会特别有用。在其中应该说明类的设计思路、总体设计方法和曾考虑过后又放弃的其他方案等等。
- 类使用指南。这主要是针对代码用户所做的注释，当看到一个类定义时，用户往往第一个念头就是想要知道这个类可以用来做什么，以及怎么才能正确地做到。添加这样的注释，可以有效地帮助用户进入一些代码作者认为正确地使用场景，而不是任凭用户自行想象，然后使用错误的方式来操作这个类。
- 说明类不变项。类确保从调用者的视角来看，该条件总是为真。在类各个成员函数内部处理过程中，不变项不一定会保持，但在函数退出，控制权返回到调用者时，不变项必须为真。

请注意，要想达到以上这些目标，使用重复代码的注释方式是不可能做到的。下面是我们一个类注释格式及演示如何达到第 2、3 两个个目标的例子：

```
/// @brief 提供对 MyGraph 中所有图形元素公共特性的抽象
///
///  具备可显示，可移动等特征的图形基类，所有其他的图形元素都派生自该类型。
///  MyGraph 的用户多数情况下都会使用那些具体的 Shape 派生类，而不是 Shape 。
/// \n\n
///  当调用 @ref [2] Shape::Shape " 构造函数 " 创建出一个图形对象时，
///  该对象并不会马上显示到显示设备上，而是需要调用该对象的 @ref draw
///  方法后图形才会得以显示。不过需要注意的是，@ref draw 方法只是负责将
///  图形的最新状态绘制出来，并不负责清除这个图形上一次在显示设备上的残留
///  图像。因此在大多数情况下，你可能都需要首先调用 @ref Device::clean
///  方法进行清除操作。图形还支持移动操作，调用 @ref moveTo 方法，
///  就可以将图形移动到目标区域，注意该方法同时还具备图形拉伸的功能。
///  如果希望获得当前图形所在的位置，则可以通过调用 @ref getPosition
///  达到这一目的。
/// \n\n
///  继承自 Shape 基类的派生类可以通过重写 @ref prepareDevice 和
/// @ref drawOnDevice 方法自定义各自不同的图形绘制行为
///
/// @invariant 无论在任何情况下，图形必然不会超出
///  (-1600, -1600, 1600, 1600) 这一范围
class Shape
{
```

请特别注意例子中注释的最后部分，使用 @invariant 指出的不变项，这一承诺将对类所有的方法有效。因此，单元测试的编写者将会非常乐意对这一承诺加以验证。

那么，是不是所有的类，我们都需要为其编写如此详尽细致的说明注释呢？事实上完全

---

[2] @ref 是 doxygen 中用于引用其他主题的特殊标签，本例中多处出现的引用都会指向一些方法说明。

没有必要。 所谓使用指南， 只有在别人有使用的需求且存在疑问， 需要这些信息加以解答时才具有实际的意义。 所以， 对于大多数为了优化代码结构， 或是为了实施某些设计模式而衍生出来， 并不真正直接向外界提供功能的内部实现类而言， 都不需要编写示例中那样太过详细的指南性质注释。这些类更需要的是说明该类设计方法的注释。

大致上我们认为只要满足下面列举的一些条件，就应该编写较为详细的指南性说明注释：

- 概要设计中明确提出的接口类
- 会被多人或是多处共享的工具类
- 虽是局部使用，但结构复杂以至于很容易引起错误理解以至于需要特别说明

4.4 数据声明注释规范

数据声明包括类数据成员、具名常量和位标志。

变量声明的注释应给出变量名未表达出来的各种信息。 研究表明， 对数据进行注释比对使用数据的过程做注释更为重要（ SDC 1982）。下面是对数据进行注释的一些准则：

- 注释数值的单位。 例如某个数值变量表示长度， 请指明长度单位是英寸、 英尺、米、千米还是毫米。不能认为单位是不言自明的——新手不知道，工作于另一个系统的人也不知道。
- 说明数值的允许范围。如果变量值有一个期望范围，就应该明确的说明这个范围。

我们约定， 可以在一行以内容纳的数据声明注释， 都应该使用行尾注释法， 下面是一个变量声明注释的格式示例：

```
int    currentPosition_;       ///<  以像素为单位，取值范围    (-1600, -1600, 1600, 1600)
```

注意 ///之后的 <小于号，这是行尾注释必须遵循的格式。

一行无法容纳的数据声明注释，则不采用行尾注释法。下面是一个多行注释的例子：

```
/// statusFlags 位含义如下
/// MSB 0   错误检测：  1=有，  0=无
/// 1-2   错误类型：  0=语法，  1=警告，  2=服务器，  3=致命
/// 3   保留
/// 4   打印状态：  1=准备好，  0=没有准备好
/// …
/// 14   没有使用  (应该为零  )
/// LSB 15-32   没有使用  (应该为零  )
int statusFlags;
```

4.5 函数注释规范

为一个函数编写注释，一般而言需要说明以下几个部分的信息：

a) 函数的概要描述，使用 @brief 加以标识。注意如果一个函数无法用一两句话说清

楚，就有必要考虑我们到底想让函数做什么。 要是不便做出简短的说明， 往往意味着设计还嫌不足，这时就应该反思一下是否需要修正设计了。原则上除了简单的 get/set 访问器，其余所有的函数都应该附上概要说明。

b) 通过函数名无法传达的信息， 紧接概要描述之后， 并与其以一空行隔开。 如果函数名和概要描述已经足够说明这个函数的一切， 那么完全可以忽略这一部分。 但有些时候我们希望告诉读者一些名称和标准功能之外的函数行为特征时， 就可以将这些内容通过这种方式表达出来。 这些内容很可能会描述一些函数精度局限性； 是否以及如何对某一全局数据进行了修改；所使用算法的来源等等。

c) 参数的描述，每个参数注释都使用 @param 加以标识。事实上，参数注释的内容和前面介绍的数据注释非常类似， 它们都用于给出名称未能表达的信息， 比如单位等信息。 只不过， 对于函数注释而言， 有专门的前置条件用于描述参数取值的范围限定。所以在参数注释中就不需要专门指出其取值范围。注意为参数加上输入 /输出标记是必要的，这一指示通过紧接参数名的 [in]/[out] 来体现。

d) 前置条件描述，在最后一个参数注释之后使用 @pre 加以标识。说到前置条件，这在契约式编程中是很重要的概念。 它是函数与其调用者之间制定的契约之一。 前置条件规定了作为函数的调用者， 必须遵循的一些调用规则， 如果调用者违反的这些规则， 那么函数将不会正确执行。 正如前面提到， 前置条件可以用于描述参数的取值范围。 事实上， 前置条件可以约定的规则远不止这一点， 例如作为一个数据库数据检索函数，它可以在其前置条件中指出必须首先保证数据库连接是打开的等等。
前置条件不是函数注释中必须的选项， 但是我们认为作为契约式编程实践的重要体现，尽量明确的为自己的函数构筑起防线是非常值得的事情。

e) 返回值描述，在前置条件之后使用 @return 加以标识。对于返回值，我们需要以比较简短的语言说明它的含义， 同时也要注意是否存在单位等需要特别说明的附加信息。返回值描述比较特殊的一个地方在于， 如果返回值的值本身有一个确定的取值范围，并且应该分项说明的话，可以使用 @retval 标记，对每个可能返回的值加以说明。如果函数没有返回值或是返回值为 void，则可以忽略本项内容。

f) 后置条件描述，在返回值描述之后使用 @post 加以标识。与前置条件相对应，后置条件用于向函数调用者承诺函数保证会做到的事情， 以及函数执行完成时世界的状态。一个函数具有后条件这一事实同时也意味着它会结束：不可能出现无限循环。
同前置条件一样，后置条件同样不是必须的选项，但这并不意味着其可有可无。

下面是一个包含了上述所有内容的成员函数注释示例：

```
/// @brief 移动图形到参数指定的位置并返回指示移动成功与否的标志
///
/// 图形移动到参数指定的矩形区域中后，还将根据目标矩形的大小进行拉伸，
/// 注意调用 moveTo 并不会马上引发图形的重绘，如果希望显示图形移动后的
/// 效果，还需要调用对象的 @ref draw 方法
/// @param[in] newPosition 形状占据的矩形区域，单位为像素
/// @pre 移动的目标位置被限制在 (-1600, -1600, 1600, 1600) 这一范围之内
/// @return 一个标志移动操作是否正确完成的整型值
/// @retval 0 移动成功
/// @retval 1 显示设备不支持的位置
```

```
/// @retval 2 ...
/// @post   移动后的图形位置将位于参数指定的位置并进行了适当拉伸
int moveTo(const Rect & newPosition);
```

4.6 代码标记注释规范

在代码编写过程中，经常需要在尚未完成的代码骨架中加入一些警示标记，或是当发现程序某处存在 bug 时加以标注以备随后修正。糟糕的是，不同的程序员可能会使用不同的方式来表达他们的意图，这导致维护难度增加以及很难使用工具检测。所以需要我们以一种统一且使用工具容易检测的方式来进行代码标记。

标注计划中需要完成的代码，我们可以在准备添加代码的地方使用 @todo 标记，这样不但在 doxygen 生成的文档中会醒目的表现出来，而且很多编辑器也能够识别这一标记，在特定的位置提示你将要进行的工作。类似的，为了指示已知 bug，我们可以使用 @bug 标记。

下面是这些标注格式的示例：

```
/// @todo   在此处添加文本过滤代码

/// @bug BUG 号 BUG 描述
```

5 FAQ

5.1 枚举值需要注释吗？

事实上，枚举值本身就是对一些原本是字面常量值的特殊注释。下面是我们经常可以看到的枚举注释：

```
enum Color
{
        RED = 0,     ///<  红色
        GREEN,     ///<  绿色
        BLUE,        ///<  蓝色
};
```

显然，这样画蛇添足地重复代码的注释毫无意义。请记住，枚举值本身就是说明，取一个合理的好名字远胜于添加一段无谓的注释。与之类似的还有 typedef，在是否需要注释这个问题上，他们具有类似的特征。

5.2 前置条件、后置条件和不变式有必要注释出来吗？

其实，采用注释来表现程序契约纯属无奈之举。原因很简单，C++语言本身并不支持程序契约的表达。可是经验证明，按合约设计的程序，在交付后产生问题的可能性远远小于没

有明确约束的。

尽管借助 C++ 编译器无法执行这些契约，但是至少我们可以通过编写注释，以一种语言提醒的方式对这一缺陷加以弥补。有一个可行的途径总好过一无所有不是吗。

5.3 写注释太耗时间怎么办？

有效注释并不费时。注释太多并不比注释太少好，

注释占用太多时间通常归因于两点。

一是注释风格耗时或枯燥乏味。我们在制定规范时对此进行了充分考虑，在编写便利和符合 doxygen 要求之间作了很多平衡。

二是因为不容易想出程序干什么的描述，所以写注释太难。这通常意味着还没有真正的理解程序。“写注释”所占用的时间其实都用在了更好地理解程序上面，不管写不写注释，这些时间注定是要花的。

5.4 有效的注释是指什么？

参考前文中提到的注释分类，我们认为，对于已经发布的代码，只允许有三种类型的注释：代码无法表达的信息、意图性注释和概述性注释。

特别需要注意的是，重复代码性质的注释，无论如何都不能称为好注释。编写这样的注释只会给人带来阅读和维护的负担。

参考书目

[McConnell04] Steve McConnell. CODE COMPLETE (2nd Edition) (Microsoft Press 2006)

[Hunt-Thomas04] A.Hunt and D.Thomas. The Pragmatic Programmer (Addison Wesley 2004)

[Doxygen06] Doxygen manual 1.4.7 (Doxygen 2006)

参考工具

[Doxygen] Doxygen 1.4.7