有灵机器人

C 语言编程规范与风格指南

整理：陆巍

Youling Robotics

2023 年 12 月 14 日

前言

本指南的内容来自多家企业、组织。其中规范是必须遵守的，而风格则可按照个人喜好选择，但同一项目中个人编写的代码应只使用一种风格。

目录

第一章　注释

注释对于保持代码的可读性绝对重要。以下规则说明了应该注释的内容和位置。但请记住：虽然注释非常重要，但最好的代码是自文档化的。为类型和变量赋予合理的名称比使用晦涩难懂的名称要好得多，因为你必须通过注释来解释这些名称。

在写注释时，要为你的读者而写，即下一个需要理解你的代码的贡献者。然而现实中往往下一位读者就是你自己！

1.1　优秀的代码可以自我解释，不通过注释即可轻易读懂。

优秀的代码不写注释也可轻易读懂，注释无法把糟糕的代码变好，需要很多注释来解释的代码往往存在坏味道，需要重构。

这条规则所包含的内容实际上很多，这里只是简单的列出，细节则包含在编程中的方方面面。

1.2　注释风格

一律使用“///”（注意，是三条）来注释。即使内容跨越多行，也同样如此。可能大家在写大段注释时觉得这样不方便，是的，就是故意让人觉得不方便，以免把写注释当成写小说，注释要简洁。

1.3　一般规则

1.3.1　注释的内容要清楚、明了，含义准确，防止二义性。

1.3.2　注释解释代码难以直接表达的意图，而不是重复描述代码。

注释的目的是解释代码的设计意图、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，防止没必要的重复注释信息。

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

注释不是为了名词解释（what），而是说明用途（why）。

示例

如下注释纯属多余：

```
++i; /// increment i
if (receive_flag) /// if receive_flag is TRUE
```

如下这种无价值的注释不应出现（空洞的笑话，无关紧要的注释）：

```
/// 时间有限，现在是:04，根本来不及想为什么，也没人能帮我说清楚
```

如下的注释则给出了有用的信息：

```
/// 由于 xx 编号网上问题，在 xx 情况下，芯片可能存在写错误，此芯片进行写操作后，必须进行回读校
/// 验，如果回读不正确，需要再重复写-回读操作，最多重复三次，这样可以解决绝大多数网上应用时
/// 的写错误问题
int time = 0;
do {
  write_reg(some_addr, value);
  time++;
} while ((read_reg(some_addr) != value) && (time < 3));
```

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释，出彩的或复杂的代码块前要加注释：

```
/// Divide result by two, taking into account that x contains the carry from the add.
for (int i = 0; i < result->size(); i++)
{
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

1.3.3 修改代码时，维护代码周边的所有注释，以保证注释与代码的一致性。不再有用的注释要删除。

1.3.4 注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同。

1.3.5 对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

示例：

```
case CMD_FWD:
  ProcessFwd();
  /// now jump into case CMD_A
case CMD_A:
  ProcessA();
  break;
  /// 对于中间无处理的连续 case，已能较清晰说明意图，不强制注释。
switch (cmd_flag) {
case CMD_A:
case CMD_B:
    {
      ProcessCMD();
      break;
    }
    ...
}
```

1.3.6　避免在注释中使用缩写，除非是业界通用或子系统内标准化的缩写。

1.3.7　避免在一行代码或表达式的中间插入注释。

1.3.8　注释应考虑程序易读及外观排版的因素，使用的语言若是汉、英兼有的，建议多使用汉语，除非能用非常流利准确的英语表达。对于有外籍员工的，由产品确定注释语言。

1.3.9　标点、拼写和语法应按照所用语言的习惯进行书写。

1.3.10　注释格式采用 doxygen 可识别的格式。

1.3.11　TODO 待办注释

对那些临时的、短期的解决方案，或已经够好但并不完美的代码使用 TODO 注释。

这样的注释要使用全大写的字符串 TODO，后面括号（parentheses）里加上你的大名、邮件地址等，还可以加上冒号（colon）：目的是可以根据统一的 TODO 格式进行查找：

```
/// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
/// TODO(Zeke) change this to use relations.
```

如果加上是为了在“将来某一天做某事”，可以加上一个特定的时间（”Fix by November 2025”）或事件（”Remove this code when all clients can handle XML responses.”）。

1.4　文件注释

在每一个手工编写的文件开头加入文件注释。注释必须列出：版权说明、版本号、生成日期、作者姓名、内容、功能说明、修改日志等。格式如下：

```
/// ---------------------------------------------------------------------
/// @file 本文件的文件名
/// @brief 本文件实现功能的简述
/// @details 详细描述
/// @copyright 版权声明
/// @author 作者
/// @version 版本
/// @date 创建日期
/// 修改日志
/// ---------------------------------------------------------------------
```

示例：

```
/// ---------------------------------------------------------------------
/// @file config.hpp
```

```
/// @brief configuration file
/// @copyright Copyright (c) 2008-2020, by Google. All rights reserved.
/// @author Charles Martin
/// @version 1.0.0
/// @date May 20, 2009
/// ---------------------------------------------------------------------
```

对于开源项目，一般采用下面的方式来注释：

```
/// @copyright Copyright 2023 Willard Lu
///
/// Use of this source code is governed by an MIT-style
/// license that can be found in the LICENSE file or at
/// https://opensource.org/licenses/MIT.
```

可以根据实际需要，添加各种说明，例如版本号。

1.5　函数注释

函数声明处注释描述函数功能，定义处描述函数实现。

1.5.1　函数声明

几乎每个函数声明的前面都应该有注释，用来描述函数的功能和使用方法。只有当函数简单且明显时，这些注释才可以省略。注释使用描述式（”Opens the file”）而非指令式（”Open the file”）；注释只是为了描述函数而不是告诉函数做什么。通常，注释不会描述函数如何实现，那是定义部分的事情。

函数声明处注释的内容：

1）inputs（输入）及 outputs（输出）；

2）如果函数分配了空间，需要由调用者释放；

3）参数是否可以为 NULL；

4）是否存在函数使用的性能隐忧（performance implications）；

5）如果函数是可重入的（re-entrant），其同步前提（synchronization assumptions）是什么？

示例：

```
/// @brief 在光标的当前位置打印字符
///
/// 如果字符是换行符 ('\n')，光标应该移动到下一行 (必要时滚动)。如果字符是一个回车
↪  ('\r'),
```

```
/// 光标应该立即重置为当前行的开头，从而导致任何未来的输出覆盖该行上的任何现有输出。
/// 如果遇到退格符 ('\b')，应该擦除前面的字符 (在其上写一个空格并将光标移回一列)。
///
/// @param ch 要打印的字符
/// @return 输入字符
int PutByte(char ch);
```

注意，不要过于啰嗦，也不要说完全显而易见的事情。

1.5.2　函数定义

如果某个函数的工作方式有些麻烦，则该函数定义应有一个解释性注释。例如，在定义注释中，可能会描述使用的任何编码技巧，概述所经历的步骤，或者解释为什么选择了以自己的方式实现功能而不是使用可行的替代方法。

注意，您不应该只是在.h 文件或其他地方重复函数声明中给出的注释。简单地概括一下函数的功能是可以的，但是注释的重点应该是它是如何实现的。

1.6　变量注释

通常变量名本身足以很好说明变量用途, 特定情况下, 需要额外注释说明。

1.6.1　全局变量（常量）

全局变量要有较详细的注释，包括对其功能、取值范围以及存取时注意事项等的说明。例如：

```
/// The ErrorCode when SCCP translate
/// Global Title failure, as follows    变量作用、含义
/// 0 -SUCCESS    1 -GT Table error
/// 2 -GT error Others -no use          变量取值范围
/// only function SCCPTranslate() in
/// this modual can modify it, and other
/// module can visit it through call
/// the function GetGTTransErrorCode()  使用方法
BYTE g_GTTranErrorCode;
```

1.7　实现注释

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

1.7.1　代码前注释

出彩的或复杂的代码块前要加注释，如：

```
/// Divide result by two, taking into account that x
/// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

1.7.2　行注释

比较隐晦的地方要在行尾加入注释，可以在代码之后空两格加行尾注释，如：

```
/// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
return; /// Error already logged.
```

注意，有两块注释描述这段代码，当函数返回时注释提及错误已经被记入日志。

前后相邻几行都有注释，可以适当调整使之可读性更好：

```
...
DoSomething();                    /// Comment here so the comments line up.
DoSomethingElseThatIsLonger();    /// Comment here so there are two spaces
                                  /// between the code and the comment.
...
```

1.7.3　函数参数注释

当函数参数的含义不明显时，请考虑以下补救措施之一：

- 如果参数是一个字面常量，并且在多个函数调用中以默认相同的方式使用相同的常量，那么应该使用一个命名常量来显式地表示约束，并确保它是有效的。
- 考虑更改函数签名，以用枚举参数替换布尔参数。这将使参数值能够自我描述。
- 对于具有多个配置选项的函数，可以考虑定义单个类或结构来保存所有选项，并传递一个实例。这种方法有几个优点。在调用点上，选项是通过名称引用的，这澄清了它们的含义。它还减少了函数的参数计数，这使得函数调用更容易读写。另一个好处是，当您添加另一个选项时，您不必更改调用点。

- 将大型或复杂的嵌套表达式替换为命名变量。
- 作为最后的选择，请使用注释来澄清调用点上的参数含义。

考虑下面的例子:

```
/// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

与:

```
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

1.7.4　不要做的事

不要陈述显而易见的事情。特别是，不要逐字地描述代码的功能，除非此行为对精通 c++ 的读者来说并不明显。相反，应该提供更高级的注释来描述为什么代码要这样做，或者使代码自描述。

比较下面两段代码中的注释:

```
/// Find the element in the vector.  <-- Bad: obvious!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

```
/// Process "element" unless it was already processed.
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

自描述代码不需要注释。来自上面例子的注释是显而易见的:

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```