

[google c++ style guide](#)

一. 背景

在大多数的 Google 开源项目中, c++是最主要的开发语言。每个 c++程序员都知道, c++语言有很强大的语言特性, 但是会导致程序的复杂度, 在编程的过程中容易出现 bug **【bug-prune】**, 并且难以维护 **【maintain】**。

这个文档通过详细的描述该做 **【dos】** 与不该做 **【don't s】** 来进行管理代码的复杂度。在兼顾代码库易维护 **【manageable】** 的同时, 保证 c++程序员在使用 c++特性的时候可以得心应手 **【productively】**。

风格 **【style】**, 也叫代码的可读性, 是用来管理的代码库的惯例 **【conventions】**。风格这个词有些不恰当 **【misnomer】**, 因为这些惯例不仅仅包括源码文件的格式。

大多数的 Google 开源项目都会遵循 **【conform】** 文档中描述的风格。

注意: 这个文档不是 c++的使用说明书, 我们默认读者对 c++足够熟悉

1.1 目的

为什么我们需要提供这个文档?

该文档需要提供 **【serve】** 一些主要的目的。找到每个单一规则的最原始的出发点。通过对每一个规则适用性 **【in place】** 进行详细的讨论, 并且得出相应的结论 **【decision】**。如果读者已经理解了每一个规则所要达到的目的, 那么在规则被遗弃, 或者某个规则被改变的时候, 读者也会理解这样做的意图。

正如你看到的这个文档的目的如下:

- 规则应起作用 **【pull their weight】**

遵守规则所带来的益处需足够的大, 从而可以证明让所有的工程师去记住这样的规则是正确的。规则大多数解释我们不用相应的特性, 而不是我们使用相应的特性。比如说 goto 违反 **【contravenes】** 了许多下面相应的规则, 但是 goto 已经被遗弃, 因此该文档将不会考虑讨论它。

- 从读者的角度进行优化

通常期望代码库 (大多数的子模块 **【individual components】** 提交到代码库中) 可以使用很长时间。因此, 大多数时间回去读相应的 code, 而不是去写。我们明确的 **【explicitly】** 去选择优化中等水平的软件工程师的读代码, 维护 **【maintain】**, 和 debug 代码的能力, 而不是去优化编写所述 code 的难度。“给读者留下线索”是这些准则的另外一个特别 **【particularly】** 目的。当代码库中有一段 **【snippet】** 精妙或者不寻常 **【unusual】**

的 code(比如说, 指针 ownership 的转换), 留下相应的文字提示【textual hints】是非常重要的(std::unique_ptr 表明了 ownership 转换的无二义性【unambiguously】)。

- 与现有的 code 保持一致性
- 在必要的情况下与 c++社区保持一致
- 避免奇怪的【surprising】或者危险的设计
- 避免设计出中等 engineer 难以维护的代码
- 注意【mindful】代码库的大小
- 在必要的情况下需要考虑优化

性能优化【performance optimization】有些时候是必要的, 尽管同时会与这个文档中一些准则会有冲突。

这个文档的意图是提供最大的指导, 通过合理的约束【reasonable restriction】。通常编程常识或者效果好的习惯【good taste】将被接受并且盛行【prevail】。我们特指的是在整个 google c++社区中的建立的管理, 而不是你的个人或者某个团队的偏好。如果在使用精妙的【clever】或者不寻常的观念【constructs】表示怀疑【skeptical】或者不情愿【reluctant】, 没有禁止【prohibition】并不以为这允许【proceed】, 通过自己的判断, 如果还不确信, 可以向 project leader 进行确认得到额外的解释。

1.2 c++版本

当前 code 主要面向于 c++17,而不是 c++2x 中的特性。随着时间的推移, 这个文档面向的 c++版本将会进行修改【advance】。

不要去使用非标准的扩展【extensions】。

在使用 c++14 和 c++17 中的特性之前, 应该认真考虑转换到其他环境时的便携性【portability】。

二. 头文件

通常, 每一个.cc 文件都应有相关的【associated】.h 文件。当然会有一些例外【exceptions】, 例如单元测试和仅仅含有 main 函数的.cc 文件。

正确的使用头文件将对代码可读性, 文件大小, 性能产生很大的影响【make a huge difference to】。

下面的规则将会带领你认识使用头文件所遇到的陷阱【pitfall】。

2.1 self-contained 头文件

2.2 头文件保护符【guard】

所有的头文件都应含有#define 保护, 用来防止多重引入【multiple inclusion】, 标识符的命名格式应为 <PROJECT>_<PATH>_<FILE>_H_.

为了保证唯一性【uniqueness】, 标识符的命名应根据项目源文件所在的全路径。例如, 文件/scr/bar/baz.h 位于项目 foo 下, 则该头文件应有下面的头文件保护符。

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif // FOO_BAR_BAZ_H_
```

说明: 第二种方法: #pragma once

2.3 前向声明

避免使用前向声明, 取而代之的是使用#include 引入需要的头文件。

2.4 内联函数

2.5 include 顺序

四. 类

4.1 构造函数的职责

4.2 隐式转换

4.3 可拷贝和可移动类型

4.4 结构和类

4.5 结构和 pairs 和 tuples

4.6 继承

4.7 操作符重载

4.8 存取控制

4.9 声明顺序

五. 函数

5.1 输出参数

5.2 短函数

5.3 引用参数

5.4 函数重载

5.5 默认参数

六. google 的奇巧

6.1 ownership 和智能指针

6.2 c++格式检查工具