C++ 风格指南

命名约定

最重要的一致性规则是命名管理。命名的风格能让我们在不需要去查找类型声明的条件下快速地了解某个名字代表的含义：类型，变量，函数，常量，宏，等等。

命名规则具有一定随意性，但相比按个人喜好命名，一致性更重要。

1. 通用命名约定

函数命名，变量命名，文件命名要有描述性；少用缩写。

说明：尽可能使用描述性的命名，不要用只有项目开发者能理解的缩写，也不要通过砍掉几个字母来缩写单词。

```
int price_count_reader;    // 无缩写
int num_errors;            // "num" 是一个常见的写法
int num_dns_connections;   // 人人都知道 "DNS" 是什么

int n;                     // 毫无意义.
int nerr;                  // 含糊不清的缩写.
int n_comp_conns;          // 含糊不清的缩写.
int wgc_connections;       // 只有贵团队知道是什么意思.
int pc_reader;             // "pc" 有太多可能的解释了.
int cstmr_id;              // 删减了若干字母.
```

注意，一些特定的广为人知的缩写是允许的，例如用 `i` 表示迭代变量和用 `T` 表示模板参数。

模板参数的命名应当遵循对应的分类：类型模板参数应当遵循 :ref:`类型命名 <type-names>` 的规则，而非类型模板应当遵循 :ref:`变量命名 <variable-names>` 的规则.

命名约定

2. 文件命名

文件名要全部小写，可以包含下划线 ***(_)*** 或连字符 ***(-)***，依照项目的约定 **.** 如果没有约定，那么 ***"_"*** 更好。

可接受的文件命名示例 ***:***

my_useful_class.cpp

my-useful-class.cpp

myusefulclass.cpp

myusefulclass_test.cpp // _unittest 和 ***_regtest*** 已弃用 **.**

C++ 文件要以 ***.cpp/.cc(google)*** 结尾，头文件以 ***.h*** 结尾，专门插入文本的文件则以 ***.inc*** 结尾。

不要使用已经存在于 ***/usr/include*** 下的文件名，如 ***db.h*** 。

通常应尽量让文件名更加明确 ***http_server_logs.h*** 就比 ***logs.h*** 要好。定义类时文件名一般成对出现，如 ***foo_bar.h*** 和 ***foo_bar.cc*** ，对应于类 ***FooBar*** 。

命名约定

3. 类型命名

类型名称的每个单词首字母均大写，不包含下划线：***MyExcitingClass***, ***MyExcitingEnum***

所有类型命名 —— *类*，*结构体*，*类型定义* ***(typedef)***，*枚举*，*类型模板参数* —— 均使用相同约定，即以大写字母开始，每个单词首字母均大写，不包含下划线。例如：

```
// 类和结构体

class UrlTable { ...

class UrlTableTester { ...

struct UrlTableProperties { ...

// 类型定义

typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// using 别名

using PropertiesMap = hash_map<UrlTableProperties *, string>;

// 枚举

enum UrlTableErrors { ...
```

命名约定

4. 变量命名

变量 *(*包括函数参数*)* 和数据成员名一律小写，单词之间用下划线连接 *.* 类的成员变量以下划线结尾，但结构体的就不用，如 *: a_local_variable, a_struct_data_member, a_class_data_member_.*

- 普通变量命名

string table_name; // 好 **-** 用下划线

string tablename; // 好 **-** 全小写

string tableName; // 差 – 混合大小写

- 类数据成员

不管是静态的还是非静态的，类数据成员都可以和普通变量一样，但要接下划线 *.*

```
class TableInfo { // google
  ...
 private:
  string table_name_; // 好 - 后加下划线 .
  string tablename_;  // 好 .
  static Pool<TableInfo>* pool_; // 好 .
};
```

```
class TableInfo { // our
  ...
 private:
  string m_table_name; // 好 - 后加下划线 .
  string m_tablename;  // 好 .
  static Pool<TableInfo>* m_pool; // 好 .
};
```

阿里 *MNN* 中，使用 *mTableName*

命名约定

4. 类型命名

- 结构体变量

不管是静态的还是非静态的，结构体数据成员都可以和普通变量一样，不用像类那样接下划线*：*

```
struct UrlTableProperties {
  string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

命名约定

5. 常量命名

声明为 ***constexpr*** 或 ***const*** 的变量，或在程序运行期间其值始终保持不变的，命名时以 ***"k"*** 开头，大小写混合。例如：

const int kDaysInAWeek = 7;

所有具有静态存储类型的变量都应当以此方式命名。对于其他存储类型的变量，如自动变量等，这条规则是可选的。如果不采用这条规则，就按照一般的变量命名规则。

6. 函数命名

常规函数使用大小写混合，取值和设值函数则要求与变量名匹配：***MyExcitingFunction()***, ***MyExcitingMethod()***, ***my_exciting_member_variable()***, ***set_my_exciting_member_variable()***.

一般来说，函数名的每个单词首字母大写（即 "驼峰变量名" 或 "帕斯卡变量名"），没有下划线。对于首字母缩写的单词，更倾向于将它们视作一个单词进行首字母大写（例如，写作 ***StartRpc()*** 而非 ***StartRPC()***).

AddTableEntry()

DeleteUrl()

OpenFileOrDie()

取值和设值函数的命名与变量一致。一般来说它们的名称与实际的成员变量对应，但并不强制要求。例如 ***int count()*** 与 ***void set_count(int count)***.

命名约定

7. 命名空间命名

命名空间以小写字母命名．最高级命名空间的名字取决于项目名称．要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突。

顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字。命名空间中的代码，应当存放于和命名空间的名字匹配的文件夹或其子文件夹中。

8. 枚举命名

枚举的命名应当和 :ref:`常量 <constant-names>` 或 :ref:`宏 <macro-names>` 一致***：kEnumName*** 或是 ***ENUM_NAME.***

单独的枚举值应该优先采用 :ref:`常量 <constant-names>` 的命名方式．但 :ref:`宏 <macro-names>` 方式的命名也可以接受．枚举名 ***UrlTableErrors (***以及 ***AlternateUrlTableErrors)*** 是类型，所以要用大小写混合的方式．

```
enum UrlTableErrors { // 推荐
  kOK = 0,
  kErrorOutOfMemory,
  kErrorMalformedInput,
};
enum AlternateUrlTableErrors { // 我们现在使用的
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

2009 年 ***1*** 月之前，我们一直建议采用 :ref:`宏 <macro-names>` 的方式命名枚举值．由于枚举值和宏之间的命名冲突，直接导致了很多问题．由此，这里改为优先选择常量风格的命名方式．新代码应该尽可能优先使用常量风格．但是老代码没必要切换到常量风格，除非宏风格确实会产生编译期问题．

命名约定

9. 宏命名

通常不应该使用宏。如果不得不用，其命名像枚举命名一样全部大写，使用下划线***：***

#define ROUND(x) ...

#define PI_ROUNDED 3.0

10. 命名规则的特例

如果你命名的实体与已有 ***C/C++*** 实体相似，可参考现有命名策略

bigopen(): 函数名，参照 ***open()*** 的形式、

uint: typedef

bigpos: struct 或 ***class,*** 参照 ***pos*** 的形式

sparse_hash_map: STL 型实体***；*** 参照 ***STL*** 命名约定

LONGLONG_MAX: 常量***，*** 如同 ***INT_MAX***

注释

注释虽然写起来很痛苦，但对保证代码可读性至关重要．下面的规则描述了如何注释以及在哪儿注释．当然也要记住：注释固然很重要，但最好的代码应当本身就是文档．有意义的类型名和变量名，要远胜过要用注释解释的含糊不清的名字．

1. 注释风格

使用 ***//*** 或 ***/* */,*** 统一就好。

// 或 ***/* */*** 都可以；但 ***//*** 更 常用．要在如何注释及注释风格上确保统一。

2. 文件注释

在每一个文件开头加入版权公告．（文件注释描述了该文件的内容．如果一个文件只声明，或实现，或测试了一个对象，并且这个对象已经在它的声明处进行了详细的注释，那么就没必要再加上文件注释．除此之外的其他文件都需要文件注释。）

每个文件都应该包含许可证引用。为项目选择合适的许可证版本***.(*** 比如***, Apache 2.0, BSD, LGPL, GPL)***

如果你对原始作者的文件做了重大修改***,*** 请考虑删除原作者信息。

如果一个 ***.h*** 文件声明了多个概念，则文件注释应当对文件的内容做一个大致的说明，同时说明各概念之间的联系。一到两行的文件注释就足够了，对于每个概念的详细文档应当放在各个概念中，而不是文件注释中。

不要在 ***.h*** 和 ***.cc*** 之间复制注释。

注释

3. 类注释

每个类的定义都要附带一份注释***,*** 描述类的功能和用法***,*** 除非它的功能相当明显***.***

```
// Iterates over the contents of a GargantuanTable.
// Example:
//    GargantuanTableIterator* iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
//    delete iter;
class GargantuanTableIterator {
  ...
};
```

类注释应当为读者理解如何使用与何时使用类提供足够的信息，同时应当提醒读者在正确使用此类时应当考虑的因素。如果类有任何同步前提，请用文档说明。如果该类的实例可被多线程访问，要特别注意文档说明多线程环境下相关的规则和常量使用。

如果你想用一小段代码演示这个类的基本用法或通常用法，放在类注释里也非常合适。

如果类的声明和定义分开了***(***例如分别放在了 ***.h*** 和 ***.cc*** 文件中***),*** 此时，描述类用法的注释应当和接口定义放在一起，描述类的操作和实现的注释应当和实现放在一起。

注释

4. 函数注释

函数声明处的注释描述函数功能***;***定义处的注释描述函数实现。

函数声明处注释的内容***:***

函数的输入输出***.***

对类成员函数而言***:*** 函数调用期间对象是否需要保持引用参数***,*** 是否会释放这些参数***.***

函数是否分配了必须由调用者释放的空间***.***

参数是否可以为空指针***.***

是否存在函数使用上的性能隐患***.***

如果函数是可重入的***,*** 其同步前提是什么***?***

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

但也要避免罗罗嗦嗦，或者对显而易见的内容进行说明。下面的注释就没有必要加上 “否则返回 ***false”***，因为已经暗含其中了:

```
// Returns true if the table cannot hold any more entries.
bool IsTableFull();
```

注释函数重载时，注释的重点应该是函数中被重载的部分，而不是简单的重复被重载的函数的注释。多数情况下，函数重载不需要额外的文档，因此也没有必要加上注释。

注释构造／析构函数时，应当注明的是注明构造函数对参数做了什么以及析构函数清理了什么***.*** 如果都是些无关紧要的内容***,*** 直接省掉注释***.*** 析构函数前没有注释是很正常的***.***

如果函数使用了一些技巧，说明实现的大致步骤或解释如此实现的理由。

避免重复注释。

注释

5. 变量注释

通常变量名本身足以很好说明变量用途。某些情况下，也需要额外的注释说明。

每个类数据成员 ***(*** 也叫实例变量或成员变量 ***)*** 都应该用注释说明用途。如果有非变量的参数不能够用类型与变量名明确表达，则应当加上注释 ***(*** 例如特殊值，数据成员之间的关系，生命周期等 ***)*** 。如果变量类型与变量名已经足以描述一个变量，那么就不再需要加上注释。

特别地，如果变量可以接受 ***NULL*** 或 ***-1*** 等警戒值，须加以说明。 比如 ***:***

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

- 全局变量

和数据成员一样，所有全局变量也要注释说明含义及用途，以及作为全局变量的原因。比如 ***:***

```
// The total number of tests cases that we run through in this regression test.
const int kNumTestCases = 6;
```

注释

6. 实现注释

- 代码前注释

  对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释。
  巧妙或复杂的代码段前要加注释。比如：

```
// 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;
}
```

- 行注释
  比较隐晦的地方要在行尾加入注释。 在行尾空两格进行注释。 比如：

```
// 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， 这里用了两段注释分别描述这段代码的作用，和提示函数返回时错误已经被记入日志。

  如果你需要连续进行多行注释，可以使之对齐获得更好的可读性：

```
DoSomething();                    // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Two spaces between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
std::vector<string> list{
                     // Comments in braced lists describe the next element...
                     "First item",
                     // .. and should be aligned appropriately.
"Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

注释

6. 实现注释

- 函数参数注释

如果函数参数的意义不明显，考虑用下面的方式进行弥补*：*

- 如果参数是一个字面常量，并且这一常量在多处函数调用中被使用，用以推断它们一致，你应当用一个常量名让这一约定变得更明显，并且保证这一约定不会被打破。
- 考虑更改函数的签名，让某个***bool***类型的参数变为***enum***类型，这样可以让这个参数的值表达其意义。
- 如果某个函数有多个配置选项，你可以考虑定义一个类或结构体以保存所有的选项，并传入类或结构体的实例。这样的方法有许多优点，例如这样的选项可以在调用处用变量名引用，这样就能清晰地表明其意义。同时也减少了函数参数的数量，使得函数调用更易读也易写。除此之外，以这样的方式，如果你使用其他的选项，就无需对调用点进行更改。
- 用具名变量代替大段而复杂的嵌套表达式。
- 万不得已时，才考虑在调用点用注释阐明参数的意义。

比如下面的示例的对比*：*

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

和

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

- 不允许的行为

不要描述显而易见的现象，永远不要用自然语言翻译代码作为注释，除非即使对深入理解***C++***的读者来说代码的行为都是不明显的。要假设读代码的人***C++***水平比你高，即便他／她可能不知道你的用意*：*你所提供的注释应当解释代码为什么要这么做和代码的目的，或者最好是让代码自文档化。

比较这样的注释*：*

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

和这样的注释*：*

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

注释

7. 标点，拼写和语法

注意标点，拼写和语法***;***写的好的注释比差的要易读的多。

注释的通常写法是包含正确大小写和结尾句号的完整叙述性语句。大多数情况下，完整的句子比句子片段可读性更高。短一点的注释，比如代码行尾注释，可以随意点，但依然要注意风格的一致性。

8. TODO 注释

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

TODO 注释要使用全大写的字符串 ***TODO***，在随后的圆括号里写上你的名字，邮件地址，***bug ID***，或其它身份标识和与这一 ***TODO*** 相关的 ***issue***。主要目的是让添加注释的人 ***(***也是可以请求提供更多细节的人***)*** 可根据规范的 ***TODO*** 格式进行查找。 添加 ***TODO*** 注释并不意味着你要自己来修正，因此当你加上带有姓名的 ***TODO*** 时，一般都是写上自己的名字。

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.

// TODO(Zeke) change this to use relations.

// TODO(bug 12345): remove the "Last visitors" feature
```

如果加 ***TODO*** 是为了在 “将来某一天做某事”，可以附上一个非常明确的时间 “***Fix by November 2005”)***，或者一个明确的事项 ***(“Remove this code when all clients can handle XML responses.”).***

8. 弃用注释

通过弃用注释（***DEPRECATED comments***）以标记某接口点已弃用。

您可以写上包含全大写的 ***DEPRECATED*** 的注释，以标记某接口为弃用状态。注释可以放在接口声明前，或者同一行。

在 ***DEPRECATED*** 一词后，在括号中留下您的名字，邮箱地址以及其他身份标识。

弃用注释应当包涵简短而清晰的指引，以帮助其他人修复其调用点。在 ***C++*** 中，你可以将一个弃用函数改造成一个内联函数， 这一函数将调用新的接口。

标记接口为 ***DEPRECATED*** 并不会让大家不约而同地弃用，您还得亲自主动修正调用点（***callsites***），或是找个帮手。