

6. 命名约定

最重要的一致性规则是命名管理。命名风格快速获知名字代表是什么东东：类型？变量？函数？常量？宏 ... ？甚至不需要去查找类型声明。我们大脑中的模式匹配引擎可以非常可靠的处理这些命名规则。

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，所以不管你怎么想，规则总归是规则。

6.1. 通用命名规则

Tip

函数命名，变量命名，文件命名要有描述性；少用缩写。

尽可能给有描述性的命名，别心疼空间，毕竟让代码易于新读者理解很重要。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” 本来就常见
int num_dns_connections;  // 人人都知道 “DNS” 是啥
```

Warning

```
int n;                    // 莫名其妙。
int nerr;                 // 怪缩写。
int n_comp_conns;        // 怪缩写。
int wgc_connections;     // 只有贵团队知道是啥意思。
int pc_reader;           // “pc” 有太多可能的解释了。
int cstmr_id;            // 有删减若干字母。
```

6.2. 文件命名

Tip

文件名要全部小写，可以包含下划线 () 或连字符 ()。按项目约定来。如果并没有项目约定，“_”更好。

可接受的文件命名：

```
* my_useful_class.cc
* my-useful-class.cc
* myusefulclass.cc
* muusefulclass_test.cc // ``_unittest`` 和 ``_regtest`` 已弃用。
```

C++ 文件要以 `.cc` 结尾，头文件以 `.h` 结尾。专门插入文本的文件则以 `.inc` 结尾，参见:ref:self-contained headers。

不要使用已经存在于 `/usr/include` 下的文件名 (Yang.Y 注：即编译器搜索系统头文件的路径)，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`。

内联函数必须放在 `.h` 文件中。如果内联函数比较短，就直接放在 `.h` 中。

6.3. 类型命名

Tip

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 —— 均使用相同约定。例如：

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// enums
enum UrlTableErrors { ...
```

6.4. 变量命名

Tip

变量名一律小写，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：

举例：

```
string table_name; // 可 - 用下划线。
string tablename; // 可 - 全小写。
```

Warning

```
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 {
    ...
private:
    string table_name_; // 可 - 尾后加下划线。
    string tablename_; // 可。
    static Pool<TableInfo>* pool_; // 可。
};
```

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;
}
```

结构体与类的讨论参考 [结构体 vs. 类](#) 一节。

全局变量：

对全局变量没有特别要求，少用就好，但如果你要用，可以用 `g_` 或其它标志作为前缀，以便更好的区分局部变量。

6.5. 常量命名

Tip

在全局或类里的常量名称前加 `k`：kDaysInAWeek。且除去开头的 `k` 之外每个单词开头字母均大写。

所有编译时常量，无论是局部的，全局的还是类中的，和其他变量稍微区别一下。`k` 后接大写字母开头的单词：

```
const int kDaysInAWeek = 7;
```

这规则适用于编译时的局部作用域常量，不过要按变量规则来命名也可以。

6.6. 函数命名

Tip

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()`.

常规函数:

函数名的每个单词首字母大写，没有下划线。

如果您的某函数出错时就要直接 `crash`，那么就在函数名加上 `OrDie`。但这函数本身必须集成在产品代码里，且平时也可能会出错。

`AddTableEntry()`

`DeleteUrl()`

`OpenFileOrDie()`

取值和设值函数:

取值（**Accessors**）和设值（**Mutators**）函数要与存取的变量名匹配。这儿摘录一个类，`num_entries_` 是该类的实例变量:

```
class MyClass {
public:
    ...
    int num_entries() const { return num_entries_; }
    void set_num_entries(int num_entries) { num_entries_ = num_entries; }

private:
    int num_entries_;
};
```

其它非常短小的内联函数名也可以用小写字母，例如。如果你在循环中调用这样的函数甚至都不用缓存其返回值，小写命名就可以接受。

6.7. 名字空间命名

Tip

名字空间用小写字母命名，并基于项目名称和目录结构: `google_awesome_project`.

关于名字空间的讨论和如何命名，参考 [名字空间](#) 一节。

6.8. 枚举命名

Tip

枚举的命名应当和 **常量** 或 **宏** 一致: `kEnumName` 或是 `ENUM_NAME` .

单独的枚举值应该优先采用 **常量** 的命名方式. 但 **宏** 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 {
    kOK = 0,
    kErrorOutOfMemory,
    kErrorMalformedInput,
};

enum AlternateUrlTableErrors {
    OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2,
};
```

2009 年 1 月之前, 我们一直建议采用 **宏** 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.

6.9. 宏命名

Tip

你并不打算:ref:使用宏 `<preprocessor-macros>`, 对吧? 如果你一定要用, 像这样命名: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN` .

参考:ref:预处理宏 `<preprocessor-macros>`; 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
```

6.10. 命名规则的特例

Tip

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 相似实体; 参照 STL 命名约定

`LONGLONG_MAX`:

常量, 如同 `INT_MAX`

译者 (acgtyrant) 笔记

1. 感觉 Google 的命名约定很高明, 比如写了简单的类 `QueryResult`, 接着又可以直接定义一个变量 `query_result`, 区分度很好; 再次, 类内变量以下划线结尾, 那么就可以直接传入同名的形参, 比如 `TextQuery::TextQuery(std::string word) : word_(word) {}`, 其中 `word_` 自然是类内私有成员。

7. 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, 但最好的代码本身应该是自文档化. 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.

你写的注释是给代码读者看的: 下一个需要理解你的代码的人. 慷慨些吧, 下一个人可能就是你!

7.1. 注释风格

Tip

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.

7.2. 文件注释

Tip

在每一个文件开头加入版权公告, 然后是文件内容描述.

法律公告和作者信息:

每个文件都应该包含以下项, 依次是:

- 版权声明 (比如, `Copyright 2008 Google Inc.`)
- 许可证. 为项目选择合适的许可证版本 (比如, Apache 2.0, BSD, LGPL, GPL)
- 作者: 标识文件的原始作者.

如果你对原始作者的文件做了重大修改, 将你的信息添加到作者信息里. 这样当其他人对该文件有疑问时可以知道该联系谁.

文件内容:

紧接着版权许可和作者信息之后, 每个文件都要用注释描述文件内容.

通常, `.h` 文件要对所声明的类的功能和用法作简单说明. `.cc` 文件通常包含了更多的实现细节或算法技巧讨论, 如果你感觉这些实现细节或算法技巧讨论对于理解 `.h` 文件有帮助, 可以将该注释挪到 `.h`, 并在 `.cc` 中指出文档在 `.h`.

不要简单的在 `.h` 和 `.cc` 间复制注释. 这种偏离了注释的实际意义.

7.3. 类注释

Tip

每个类的定义都要附带一份注释, 描述类的功能和用法.

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

如果你觉得已经在文件顶部详细描述了该类, 想直接简单的来上一句“完整描述见文件顶部”也不打紧, 但务必确保有这类注释.

如果类有任何同步前提, 文档说明之. 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.

7.4. 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。

函数声明：

注释位于声明之前，对函数功能及用法进行描述。注释使用叙述式（“Opens the file”）而非指令式（“Open the file”）；注释只是为了描述函数，而不是命令函数做什么。通常，注释不会描述函数如何工作。那是函数定义部分的事情。

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。
- 如果函数分配了空间，需要由调用者释放。
- 参数是否可以 `NULL`。
- 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？

举例如下：

```
// 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，或做些显而易见的说明。下面的注释就没有必要加上 “returns false otherwise”，因为已经暗含其中了：

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

注释构造/析构函数时，切记读代码的人知道构造/析构函数是干啥的，所以 “destroys this object” 这样的注释是没有意义的。注明构造函数对参数做了什么（例如，是否取得指针所有权）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。

函数定义：

每个函数定义时要用注释说明函数功能和实现要点。比如说说你用的编程技巧，实现的大致步骤，或解释如此实现的理由，为什么前半部分要加锁而后半部分不需要。

不要从 `.h` 文件或其他地方的函数声明处直接复制注释。简要重述函数功能是可以的，但注释重点要放在如何实现上。

7.5. 变量注释

Tip

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。
类数据成员：

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。如果变量可以接受 `NULL` 或 `-` 等警戒值，须加以说明。比如：

private:

```
// Keeps track of the total number of entries in the table.  
// Used to ensure we do not go over the limit. -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。比如：

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

7.6. 实现注释

Tip

对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释。
代码前注释：

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(); // Comment here so there are 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.
}
```

NULL, true/false, 1, 2, 3...:

向函数传入 `NULL`, 布尔值或整数时, 要注释说明含义, 或使用常量让代码望文知意. 例如, 对比:

Warning

```
bool success = CalculateSomething(interesting_value,
                                  10,
                                  false,
                                  NULL); // What are these arguments??
```

和:

```
bool success = CalculateSomething(interesting_value,
                                  10,    // Default base value.
                                  false,  // Not the first time we're calling this.
                                  NULL);  // No callback.
```

或使用常量或描述性变量:

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
                                  kDefaultBaseValue,
                                  kFirstTimeCalling,
                                  null_callback);
```

不允许:

注意 永远不要用自然语言翻译代码作为注释。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：

Warning

```
// 现在，检查 b 数组并确保 i 是否存在，  
// 下一个元素是 i+1。  
...           // 天哪，令人崩溃的注释。
```

7.7. 标点，拼写和语法

Tip

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语句。短一点的注释（如代码行尾注释）可以随意点，依然要注意风格的一致性。完整的语句可读性更好，也可以说明该注释是完整的，而不是一些不成熟的想法。

虽然被别人指出该用分号时却用了逗号多少有些尴尬，但清晰易读的代码还是很重要的。正确的标点，拼写和语法对此会有所帮助。

7.8. TODO 注释

Tip

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大名，邮件地址，或其它身份标识。冒号是可选的。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。

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

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.”）。

7.9. 弃用注释

Tip

通过弃用注释（**DEPRECATED** comments）以标记某接口点（interface points）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，留下您的名字，邮箱地址以及括号补充。

仅仅标记接口为 `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。

修正好的代码应该不会再涉及弃用接口点了，着实改用新接口点。如果您不知从何下手，可以找标记弃用注释的当事人一起商量。

译者 (YuleFox) 笔记

1. 关于注释风格，很多 C++ 的 coders 更喜欢行注释，C coders 或许对块注释依然情有独钟，或者在文件头大段大段的注释时使用块注释；
2. 文件注释可以炫耀你的成就，也是为了捅了篓子别人可以找你；
3. 注释要言简意赅，不要拖沓冗余，复杂的东西简单化和简单的东西复杂化都是要被鄙视的；
4. 对于 Chinese coders 来说，用英文注释还是用中文注释，it is a problem，但不管怎样，注释是为了让别人看懂，难道是为了炫耀编程语言之外的你的母语或外语水平吗；
5. 注释不要太乱，适当的缩进才会让人乐意看。但也没有必要规定注释从第几列开始（我自己写代码的时候总喜欢这样），UNIX/LINUX 下还可以约定是使用 tab 还是 space，个人倾向于 space；
6. TODO 很不错，有时候，注释确实是为了标记一些未完成的或完成的不尽如人意的地方，这样一搜索，就知道还有哪些活要干，日志都省了。