HOPECHART

C++编码规范

编号：HQ-C-33-05　　版本：C/1　　发布号：

编制：　　日期：2021.12.28

审核：　　日期：2021.12.29

批准：　　日期：2021.12.29

保密等级：□绝密 □机密 ■敏感 □公开

受控盖章区	作废盖章区
受控文件	

基本信息

文档名称	HQ-C-33-05 C++编码规范		
上级文档信息	HQ-B-33 软件控制程序		
文档属性	工作流程□ 技术规范■ 规章制度□	维护部门	研发中心

修改记录

序号	修改时间	修改人	主要修改	存档版本
1	2018.01.22	俞月卿	将“杭州鸿泉数字设备有限公司”改为“杭州鸿泉物联网技术股份有限公司”	B/1
2	2018.06.08	沈林强等	全部改版，重新制定 C++编码规范	B/2
3	2018.06.28	沈林强	允许“{”单独一行	B/3
4	2019.10.02	金丽娟 尤胜坤	根据质量手册过程识别修改进行梳理修订；修改注释规范要求；完善枚举项、结构体定义等	C/0
5	2021.12.23	陈玲	修改文件模版要求，增加文档属性	C/1

目录

1 目的.....1
2 范围.....1
3 职责.....1
4 定义.....1
5 流程图.....1
6 内容.....1
7 相关文件.....32
8 相关记录.....32
9 附录.....32

1 目的

为了改善软件的可读性，可以让开发人员尽快而彻底地理解新的代码，确保每个软件开发人员必须一致遵守编码规范，特制定本规范。

2 范围

本规定适用于指导本公司所有产品的C++编码、评审过程。

3 职责

3.1 C++开发工程师：按照本规范执行编码。

3.2 评审专家：负责进行代码规范评审。

4 定义

无

5 流程图

无

6 内容

6.1 头文件

6.1.1 头文件名的定义

a） 头文件以.h结尾。

b） 对于插入的文本，以.inc结尾，不能以-inc.h结尾。

6.1.2 防止多次编译的保护

使用条件宏指令保护。

格式：_<PREFIX>_<FILE>_H_

PREFIX：定义前缀，一般用工程名或者库名；全部字母大写

FILE：文件名；全部字母大写

例:

```
#ifndef _FOO_BAZ_H_
#define _FOO_BAZ_H_
...
```

```
#endif // _FOO_BAZ_H_
```

6.1.3 避免使用前置声明

尽可能避免使用前置声明。尽量使用#include 加头文件的方式。

如:

在文件 a.h 中定义 ClassA 和成员：

```
class TClassA {
    int a;
    int b;
};
```

在文件 b.h 中只声明 ClassA 是一个类：

```
class TClassA;
```

这种就是前置声明。

b.h 可以改成

```
#include "a.h"
```

6.1.4 内联函数

只有当函数只有 10 行甚至更少时才将其定义为内联函数。

6.1.5 #include 的路径及顺序

使用标准的头文件包含顺序可增强可读性，避免隐藏依赖，顺序如下：

相关头文件；

C 库

C++库

其他库的.h

本项目内的.h

说明：

1. 相关头文件的说明：比如有 2 个文件，a.h，a.cpp，那么 a.cpp 中第一个引用的头文件应该是 a.h

```
#include "a.h"
```

这个就是相关头文件。

注：

避免使用以"."， ".."开头的相对路径。如：

```
#include "../a.h"
```

6.2 作用域

6.2.1 命名空间

鼓励在.cpp、.cc 文件内使用匿名命名空间或 static 声明。使用具名的命名空间时，其名称可基于项目名或相对路径。禁止使用 using 指示（using-directive）。禁止使用内联命名空间（inline namespace）。

```
// .h 文件
namespace mynamespace {
// 所有声明都置于命名空间中
// 注意不要使用缩进
class TMyClass {
    public:
    ...
    void Foo();
};
} // namespace mynamespace

// .cpp 文件
namespace mynamespace {
// 函数定义都置于命名空间中
void TMyClass::Foo() {
    ...
}
} // namespace mynamespace
```

说明：

内联命名空间举例：

```
namespace X {
inline namespace Y {
void Foo();
}
}
```

注：

不用以“std”作为命名空间的名称

禁止使用 using-directive 来使得该命名空间里面的名称都可见。如：**using namespace** foo;

禁止在头文件中使用命名空间的别名。如：**namespace** baz = ::foo::bar::baz;

禁止使用内联命名空间

6.2.2 匿名命名空间

在.cpp、.cc 文件中定义一个不需要被外部引用的变量时，可以将它们放在匿名命名空间或声明为 **static**。但是不要在.h 文件中这么做。

```
namespace {
bool xxx(){
    ...
}
} // namespace
```

6.2.3 非成员函数、静态成员函数和全局函数

使用静态成员函数或命名空间内的非成员函数，尽量不要用裸的全局函数。将一系列函数直接置于命名空间中，不要用类的静态方法模拟出命名空间的效果，类的静态方法应当和类的实例或静态数据紧密相关。

例如用下面的

```
namespace myproject {
namespace foo_bar {
```

```
void Function1();
void Function2();
}
}
```

代替

```
namespace myproject {
class TFooBar {
public:
    static void Function1();
    static void Function2();
};
}
```

6.2.4 局部变量

将函数变量尽可能置于最小作用域内，并在变量声明时进行初始化。

下面的语句

```
int i;
i = f();
```

应该用下面的语句代替：

```
int i = f();
```

下面的语句

```
vector<int> v;
v.push_back(1);
v.push_back(2);
```

应该用下面的语句代替：

```
vector<int> v = {1, 2};
```

if, while 和 for 的变量应在这些语句中声明，如：

```
while(const char *p = strchr(str, '/'))
```

```
    str = p + 1;
```

例外：

如果变量是类，则应该在循环外面定义。

如：

```
for (int i = 0; i < 100000; ++i) {
    Foo f;
    f.DoSomething(i);
}
```

应该改成：

```
Foo f;
for (int i = 0; i < 100000; ++i) {
    f.DoSomething(i);
}
```

6.2.5 静态和全局变量

禁止定义静态储存周期非 POD(Plain Old Data)变量，禁止使用含有副作用的函数初始化 POD 全局变量，因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。

POD：原生数据类型，如 **int**，**char**，**float** 等，已经 POD 类型的指针、数组、结构体等。

6.3 类

6.3.1 构造函数的职责

在构造函数中可以进行各种初始化操作。

不要在构造函数中调用虚函数，也不要在无法报出错误时进行可能失败的初始化。

需要调用虚方法或者调用有可能出错的方法时，可以定义类似于 **bool** Init()这样的方法。

6.3.2 隐式类型转换

不要定义隐式类型转换。对于转换运算符和单参数构造函数，请使用 **explicit** 关键字。

```
class TFoo {
```

```
    explicit TFoo(int x, double y);
    ...
};
void Func(Foo f);
```

6.3.3 可拷贝类型和可移动类型

如果你的类型需要，就让它们支持拷贝/移动。否则，就把隐式产生的拷贝和移动函数禁用。

6.3.4 struct VS class

仅当只有数据成员时使用 struct，其它一概使用 class。

6.3.5 继承

使用组合常常比使用继承更合理。如果使用继承的话，定义为 public 继承。

```
class TA: public TB {
};
```

6.3.6 多重继承

真正需要用到多重实现继承的情况少之又少。只在以下情况才允许多重继承：最多只有一个基类是非抽象类；其它基类都是以 Interface 为后缀的纯接口类。

6.3.7 接口

接口是指满足特定条件的类，这些类以 Interface 为后缀（不强制）。

6.3.8 运算符重载

除少数特定环境外，不要重载运算符。也不要创建用户定义字面量。

6.3.9 存取控制

将所有数据成员声明为 private，除非是 static const 类型成员（遵循常量命名规则）或者事件回调（事件回调放到 public 中）。

6.3.10 声明顺序

将相似的声明放在一起，将 public 部分放在最前，之后是 protected 部分，最后是 private 部分。

```
class TA {
public:
    //...
```

```
protected:
    //...
private:
    //...
};
```

建议：

1. 将对外开放的类型定义放入一个 public 段中
2. 将属性和事件回调成员放入一个 public 段中

如：

```
class TA {
public:
    enum TEnum{kEnumA, kEnumB};
    //...
public:
int Count{return count_;}
void SetCount(int value){count_ = value;}
//...
public:
    //...
protected:
    //...
private:
    //...
};
```

6.4 函数

6.4.1 参数顺序

函数的参数顺序为：输入参数在前，后跟输出参数。

输入参数通常是值参或者 const 引用。新加的值参也应该加入到输出参数之前，不应该放到输出参数之后。

对于即输入又输出的参数，放输入参数之后,输出参数之前。

6.4.2 编写简短函数名

我们倾向于编写简短、凝练的函数名。

6.4.3 引用参数

所有按引用传递的参数必须加上 const。

函数参数列表中，所有引用参数都必须是 const：

```
void Foo(const string &in, string *out);
```

6.4.4 函数重载

若要用好函数重载，最好能让读者一看调用点（call site）就胸有成竹，不用花心思猜测调用的重载函数到底是哪一种。该规则适用于构造函数。

```
class TMyClass {
public:
    void Analyze(const string &text);
    void Analyze(const char *text, int text_len);
};
```

如果您打算重载一个函数，可以试试在函数名里加上参数信息。例如用 AppendString()和 AppendInt()等，而不是一口气重载多个 Append()。

6.4.5 缺省参数

只允许在非虚函数中使用缺省参数,且必须保证缺省参数的值始终一致。缺省参数与函数重载遵循同样的规则。一般情况下建议使用函数重载。

6.4.6 函数返回类型后置语法

只有在常规写法（返回类型前置）不便于书写或不便于阅读时使用返回类型后置语法。

在函数名前使用 auto 关键字，在参数列表之后后置返回类型：

```
auto foo(int x) -> int;
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

6.4.7 所有权与智能指针

动态分配出的对象最好有单一且固定的所有主，并通过智能指针传递所有权。

6.5 其他 C++特性

6.5.1 右值引用

只在定义移动构造函数与移动赋值操作时使用右值引用。不要使用 std::forward。

右值引用是一种只能绑定到临时对象的引用的一种，其语法与传统的引用语法相似。

例如，**void** f(string &&s); 声明了一个其参数是一个字符串的右值引用的函数。

6.5.2 变长数组和 alloca()

禁止使用变长数组和**alloca**()。

改用更安全的分配器（**allocator**），就像 **std**::**vector** 或 **std**::unique_ptr<T[]>。

6.5.3 友元

允许合理使用友元类及友元函数。

6.5.4 异常

不要使用 C++异常。

6.5.5 运行时类型识别

禁止使用 RTTI。

RTTI 允许程序员在运行时识别 C++类对象的类型。它通过使用 typeid 或者 dynamic_cast 完成。

6.5.6 类型转换

使用 C++的类型转换，如 static_cast<>()。不要使用 **int** y = (**int**)x 或 **int** y = **int**(x) 等转换方式。

不要使用 C 风格类型转换。而应该使用 C++ 风格。

- static_cast：和 C 风格转换相似可做值的强制转换，或指针的父类到子类的明确的向上转换；
- const_cast：移除 **const** 属性；
- reinterpret_cast：指针类型和整型或其他指针间不安全的相互转换，仅在你对所做一切了然于心时使用；
- dynamic_cast：除测试外不要使用，除单元测试外，如果你需要在运行时确定类型信息，说明设计有缺陷。

6.5.7 流

只在记录日志时使用流。流用来替代 printf()和 scanf()。

6.5.8 前置自增和自减

对于简单数值（非对象），两者都可以；对于迭代器和其他模板对象使用前缀形式(++i)的自增、自减运算符。

6.5.9 const 用法

强烈建议在任何可以使用的情况下都要使用 **const**。此外有时改用 C++11 推出的 **constexpr** 更好。

约定使用 **const int** *foo 而不是使用 **int const** *foo;

6.5.10 constexpr 用法

在 C++11 里，用 **constexpr** 来定义真正的常量，或实现常量初始化。

6.5.11 整型

C++内建整型中，唯一用到的是 int，如果程序中需要不同大小的变量，可以使用<stdint.h>中的精确宽度（precise-width）的整型，如 int16_t。

公司将自定义一个类型头文件，用于各种基本类型的定义。

写代码时，尽量使用有符号的类型。

6.5.12 64 位下的可移植性

代码应该对 64 位和 32 位系统友好。处理打印、比较、结构体对齐时应切记。

6.5.13 预处理宏

使用宏时要非常谨慎，尽量以内联函数、枚举和常量代替之。

使用宏应尽可能遵守：

- 尽量不要在 .h 文件中定义宏；
- 在马上要使用时才进行 **#define**，使用后要立即 **#undef**；
- 不要只是对已经存在的宏使用**#undef**，选择一个不会冲突的名称；
- 不要试图使用展开后会导致 C++构造不稳定的宏， 不然也至少要附上文档说明其行为；
- 不建议用 ##处理函数、类和变量的名字。

6.5.14 0,nullptr 和 NULL

整数用 0，实数用 0.0，指针用 nullptr 或 NULL，字符（串）用'\0'。

6.5.15 sizeof

尽可能用 sizeof(varname)代替 sizeof(type)。

```
TStruct data, *point_data;
```

用下面的方法：

```
memset(&data, 0, sizeof(data));
memset(point_data, 0, sizeof(*point_data));
```

不要用下面的方法：

```
memset(&data, 0, sizeof(TStruct));
memset(point_data, 0, sizeof(TStruct));
```

6.5.16 auto

用 auto 绕过烦琐的类型名，只要可读性好就继续用，别用在局部变量之外的地方。

```
sparse_hash_map<string, int>::iterator iter = m.find(val);
auto iter = m.find(val);
```

6.5.17 列表初始化

可以用列表初始化。

```
struct TPoint {
    int x;
    int y;
};
Point p = {1, 2};
```

6.5.18 Lambda 表达式

适当使用 lambda 表达式。别用默认 lambda 捕获，所有捕获都要显式写出来。

Lambda 表达式是创建匿名函数对象的一种简易途径，常用于把函数当参数传，例如：

```
std::sort(v.begin(), v.end(), [](int x, int y) {
    return Weight(x) < Weight(y);
});
```

6.5.19 模板编程

不要使用复杂的模板编程。

尽量不用模板。

模板编程指的是利用 c++模板实例化机制是图灵完备性，可以被用来实现编译时刻的类型判断的一系列编程技巧。

6.5.20 C++11

尽量用 C++11 前的版本。

6.6 命名约定

6.6.1 通用命名规则

函数命名，变量命名，文件命名要有描述性；少用缩写。

```
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
```

上面的比较好理解，下面的就不好理解了。

```
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.
```

6.6.2 文件命名

文件名要全部小写，可以包含下划线（_），依照项目的约定。

文件命名示例：

my_useful_class.cpp

useful.cpp

C++文件要以.cpp 或者.cc 尾，头文件以.h 结尾。专门插入文本的文件则以.inc 结尾。

6.6.3 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。

```
// 类和结构体
class TUrlTable { ...
class TUrlTableTester { ...
```

```
struct TUrlTableProperties { ...
// 类型定义
typedef hash_map<TUrlTableProperties *, string> TPropertiesMap;
// using 别名
using PropertiesMap = hash_map<TUrlTableProperties *, string>;
// 枚举
Enum TUrlTableErrors { ...
```

6.6.4 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，但是回调函数变量用大小写方式，中间不带下划线。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;   // 不要用
string tableName;   // 不要用
```

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

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：

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

6.6.5 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;
```

6.6.6 函数命名

常规函数使用大小写混合，如：

MyExcitingFunction(),

MyExcitingMethod(),

取值和设值函数则要求与变量名匹配，且按照变量命名规则执行：

```
int Count() const {return count_;}
void SetCount(int value){count_ = vlaue;}
```

对于缩写的字母也是如此，比如：

StartRpc();

而不用

StartRPC();

6.6.7 回调函数变量：

```
typedef struct TDrvCanTag TDrvCan;
typedef void (*TDrvCanOnSent)(TDrvCan *sender);
struct TDrvCanTag
```

```
{
    TDrvCanOnSent OnSent;    // 发送完成事件
    ...
};
```

6.6.8 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。

注意：不使用缩写作为名称的规则同样适用于命名空间。命名空间中的代码极少需要涉及命名空间的名称，因此没有必要在命名空间中使用缩写。

6.6.9 枚举命名

枚举的类型名称统一以“T”开头，以“Tag”结尾。

枚举类型的值在 c++标准中规定的以“k”开始的基础上，规定后续跟枚举类型的重定义名称（除最前面的“T”外），后续跟值的名称。比如：kDrvUartParityNone。

```
typedef enum TDrvUartParityTag TDrvUartParity;
enum TDrvUartParityTag
{
    kDrvUartParityNone,
    kDrvUartParityEven,
    kDrvUartParityOdd,
};
```

整个值的名称控制在 24 个字符以内，如果整个值的名称过长，可以缩短枚举类型名称，但前提是尽可能不跟其它枚举类型冲突。如下所示，原 kDrvCanBaudRate250k 缩短为 kCanBaudRate250k：

```
typedef enum TDrvCanBaudRateTag TDrvCanBaudRate;
enum TDrvCanBaudRateTag
{
    kCanBaudRate250k,
```

```
    kCanBaudRate500k,
};
```

6.6.10 结构体和枚举名称重定义

为了使用方便，可以对结构或者枚举进行重定义，重定义格式：

```
typedef struct TxxxTag Txxx;
typedef enum TxxxTag Txxx;
```

重定义语句必须在实体定义之前，如果没有必要，两者中间不要插入其它内容，如下所示：

```
typedef enum TDrvCanBaudRateTag TDrvCanBaudRate;
enum TDrvCanBaudRateTag
{
    ...
};
```

两者之间必须插入内容的情况，如下所示：

```
typedef struct TDrvCanTag TDrvCan;
typedef void (*TDrvCanOnSent)(TDrvCan *sender);
struct TDrvCanTag
{
    TDrvCanOnSent OnSent_;
    ...
};
```

6.6.11 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：MY_MACRO_THAT_SCARES_SMALL_CHILDRE。

6.7 注释

6.7.1 注释风格

单行注释：

```
/// 前置注释
///< 后置注释
```

说明：这里的’/’和’<’后面需要空一格空格，前置注释为注释后置项，后置注释为注释前置项，具体可以参考枚举和结构体的注释例子。

多行注释：

```
/**
 多行注释
 */
```

6.7.2 文件注释

```
/**
 * @file XXXXXX.h
 * @brief XXXXXX...
 * @note XXXXXX...
 * @author XXXXXX...
 * @date YYYY-MM-DD
 * @version X.X
 *
 * @par 修改日志
 * <table>
 * <tr><th>Date        <th>Version             <th>Author
<th>Description
 * <tr><td>YYYY-MM-DD  <td>X.X                 <td>XXXXXX...
<td>XXXXXX...
 * <tr><td>YYYY-MM-DD  <td>X.X                 <td>XXXXXX...
   <td> 1.XXXXXX...   \n
        2.XXXXXX...   \n
```

```
        3.XXXXXX...
 * @copyright 杭州鸿泉物联网技术股份有限公司
 */
```

说明：

1.修改日志可分单行和多行编写，若是“类”，则需要写明注意事项，例如多线程是否安全，是否可重入。

2.file 字段必须与文件同名(不含路径，必需项,例如 rfc_string.h)

3.brief 为文件说明

4.note 为注意事项

5.author 为作者

6.7.3 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
 * @par 示例
 * @code
    GargantuanTableIterator* iter = table->NewIterator();
     for (iter->Seek("foo"); !iter->done(); iter->Next()) {
         process(iter->key(), iter->value());
     }

     delete iter;
 * @endcode
*/
class TGargantuanTableIterator {
    ...
};
```

说明：类中的注释中需要说明类的功能，示例代码等。

6.7.4 结构体注释

```
/**
 * XXXXXX...(结构体名注释)
 */
typedef struct TStringStructTag TStringStruct;
struct TStringStructTag
{
    char *value1;       ///< (后置注释)XXXXXX...
    int value2;         ///< (后置注释)XXXXXX...
    int value3;         ///< (后置注释)XXXXXX...
    char value4;        ///< (后置注释)XXXXXX...

    /// (前置备注结构体成员变量,)XXXXXX...
    char *test_value;
};
```

说明：

结构体名的注释采用多行注释,/**开头 */结尾，成员变量的注释用前置注释或后置注释的方式，“()”中的内容只是在文档中说明意义，如果在变量的末尾加注释统称后置注释，在变量的前面加注释统称前置注释。

6.7.5 枚举注释

```
/**
 * XXXXXX...(枚举名注释)
 */
typedef enum TStringEnumTag TStringEnum;
enum TStringEnumTag
{
    kStringEnumTest0    = 0,     ///< (后置备注)XXXXXX...
    kStringEnumTest1    = 1,     ///< (后置备注)XXXXXX...

    /// (前置备注)XXXXXX...
```

```
    kStringEnumTest2    = 2,
};
```

说明：

枚举名的注释采用多行注释,/**开头 */结尾，枚举项的注释用前置注释或后置注释的方式，“()”中的内容只是在文档中说明意义，如果在枚举项的末尾加注释统称后置注释，在枚举项的前面加注释统称前置注释。

6.7.6 变量注释

```
/// (前置注释)XXXXX...
extern unsigned int g_string_test_value1;
extern unsigned int g_string_test_value2; ///< (后置注释)XXXXXX...
```

6.7.7 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，需要注明多线程是否安全，是否可重入。

函数声明：

函数声明处注释的内容：

函数的输入输出。

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？

是否支持多线程等

示例：

```
    /**
     * 简要说明
     *
     * 详细说明（非必选项，上面的空行只在有需要详细说明的情况下才需要）
     * @param param1: 参数 1
     * @param param2: 参数 2
     * @todo XXXXXX...(非必选项，待办事项)
```

```
    * @note XXXXXX...(非必选项，注意事项)
    * @bug XXXXXX...(非必选项，Bug 事项)
    * @since XXXXXX...(非必选项，自从)
    * @pre XXXXXX...(非必选项，前置条件)
    * @post XXXXXX...(非必选项，后置条件)
    * @deprecated XXXXXX...(非必选项，弃用)
    * @par 代码示例 (非必选项,结合 code 与 endcode)
    * @code
        XXXXXX...
    * @endcode
    * @return
    * @see (非必选项，参见)
    */
   int Function2(int param1, int param2);
```

说明：以上的文件注释过程中“()”中的是文档中的说明参数的含义。

6.7.8 回调函数注释

```
/**
 * 数据收发完成事件的函数类型
 * @param sender: 事件的产生者
 * @param server_req: 服务器是否有数据请求发送
 * @return
 */
typedef void (*TDrvXcuSpiDoComplete)(void *sender, Bool server_req);
```

6.7.9 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。

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。

```
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;
```

全局变量定义分 2 种，一种全局变量仅在.cpp 文件中可以见，另外一种为整个工程中可见。

仅在当前.cpp 文件中的全局变量以“g_”开头，以“_”结尾，变量类型必须为“static”，如下所示：

```
static int g_count_;
```

整个工程中可见的全局变量以“g_”开头，如：

```
int g_count;
```

需要在头文件中导出 extern int g_count;

6.7.10 实现注释

对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释。

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.
```

6.7.11 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。

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很大帮助。

6.7.12 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(kl@gmail.com) : Use a "*" here for concatenation operator.
// TODO(Zeke) : change this to use relations.
// TODO(bug 12345) : remove the "Last visitors" feature
```

如果需要在代码中的某个位置设置标签,可以用 DONE

```
// DONE(fox) : xxx
```

6.8 格式

6.8.1 行长度

每一行代码字符数不超过 120 列。

包含长路径的**#include** 语句可以超出 120 列。

函数尽量控制在 60 行以内。

头文件保护可以无视该原则。

6.8.2 非 ASCII 字符

尽量不使用非 ASCII 字符，使用时尽量使用 UTF-8 编码。

6.8.3 空格还是制表位

只使用空格，每次缩进 4 个空格。

6.8.4 常规符合左右是否有空格的约定

除特殊说明外，一般的约定如下：

1. {: 前面留一个空格，后面不需要。
2. (,), [,], }: 左右不留空格。
3. =, ==, +, -, *, /, %, &, |, ^, &&, ||, >, <, >=, <=, !=: 左右各留一个空格。
4. 运算符和=连用是，如+=: 左右各留一个空格。
5. !, ~： 后面不跟空格。

6.8.5 作为块分隔符的{}放置位置

建议尽量用方案二

- 方案一

{放置在结构、类、枚举、函数、语句等名称所在行的后面，如下所示：

```
typedef struct TStructTag TStruct;
struct TStructTag {  // {放在这一行
    ...
};                   // }单独一行，位置与前面定义开始的地方对齐。
```

- 方案二

{放置在结构、类、枚举、函数、语句等名称所在行的下一行，如下所示：

```
struct TStructTag
{  // {单独一行
    ...
};                   // }单独一行，位置与前面定义开始的地方对齐。

int Count(){return count_;}  // 仅允许这样的函数把}放置在后面。
TStruct a = {1, 2};          // 或者这样的语句可以把}放置在后面。
```

6.8.6 函数声明与定义

返回类型和函数名在同一行，参数也尽量放在同一行，如果放不下就对形参分行，分行方式与函数调用一致。

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
    Type par_name2, Type par_name3) {
    DoSomething();
    ...
}
```

或者

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
Type par_name2, Type par_name3)
{
    DoSomething();
    ...
}
```

6.8.7 Lambda 表达式

Lambda 表达式对形参和函数体的格式化和其他函数一致；捕获列表同理，表项用逗号隔开。

6.8.8 函数调用

要么一行写完函数调用，要么在圆括号里对参数分行，要么参数另起一行且缩进四格。如果没有其它顾虑的话，尽可能精简行数，比如把多个参数适当地放在同一行里。

```
bool retval = DoSomething(averyveryveryverylongargument1,
    argument2, argument3);
```

6.8.9 列表初始化格式

您平时怎么格式化函数调用，就怎么格式化列表初始化。

```
// 一行列表初始化示范.
return {foo, bar};
functioncall({foo, bar});
pair<int, int> p{foo, bar};
// 当不得不断行时.
SomeType variable{
```

```
        "This is too long to fit all in one line"};  // 字符串过长，因此无法放在同一行.
    MyType m = {  // 注意了，您可以在 { 前断行.
        superlongvariablename1,
        superlongvariablename2,
        {short, interior, list},
        {interiorwrappinglist,
        interiorwrappinglist2}
    };
```

6.8.10 条件语句

建议尽量用方案二

倾向于不在圆括号内使用空格。关键字 if 和 else 另起一行。

```
if (condition) {    // if 后面空一格，圆括号里没有空格.
    ...             // 4 空格缩进.
} else if (...) {  // }后空一格，else 与 if 的右括号同一行.
    ...
} else {
    ...
}
```

或者

```
if (condition) // if 后面空一格，圆括号里没有空格.
{
    ...             // 4 空格缩进.
}
else if (...)
{
    ...
}
else
```

```
{
    ...
}
```

注意：condition 应使用 bool 型，如果是整型的话可以使用 param != 0 得到 bool 型；指针可以用 p != NULL 得到 bool 型。

6.8.11 循环和开关选择语句

switch 语句可以使用大括号分段， 以表明 cases 之间不是连在一起的。在单语句循环里，括号可用可不用。空循环体应使用{}或 continue。

```
switch (var) {
case 0: {    // 0 空格缩进
    ...      // 4 空格缩进
             // 如果没有变量定义，可以去掉这对{}

    }
    break;
    // 此处空一行
default: {
    // 如果 default 应该永远执行不到，这里为空，并且可以去掉这对{}
    }
    break;
}
```

或者

```
switch (var)
{
case 0:
    {
    ...      // 4 空格缩进
             // 如果没有变量定义，可以去掉这对{}
```

```
        }
        break;
        // 此处空一行
    default:
        {
        // 如果 default 应该永远执行不到，这里为空，并且可以去掉这对{}
        }
        break;
    }
```

6.8.12 指针和引用表达式

句点或箭头前后不要有空格。指针/地址操作符（*，&）之后不能有空格。

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

在声明指针变量或参数时，星号与变量名紧挨：

```
char *c;
const string &str;
```

6.8.13 布尔表达式

如果一个布尔表达式超过标准行宽，断行方式要统一一下。

```
if (this_one_thing > this_other_thing
    && a_third_thing == a_fourth_thing
    && yet_another && last_one) {
    ...
}
```

6.8.14 函数返回值

不要在 return 表达式里加上非必须的圆括号。

```
return result;  // 返回值很简单，没有圆括号.
```

```
// 可以用圆括号把复杂表达式圈起来，改善可读性.
return (some_long_condition
    && another_condition);
```

6.8.15 变量及数组初始化

用=初始化。

```
int x = 3;
int x(3); 不用这种初始化
int x{3}; 不用这种初始化
string name = "Some Name";
string name("Some Name"); 不用这种初始化，除非这个类有这样的构造函数
string name{"Some Name"}; 不用这种初始化
```

6.8.16 预处理指令

预处理指令不要缩进，从行首开始。

```
// 指令从行首开始
    if (lopsided_score) {
#if DISASTER_PENDING      // 正确 - 从行首开始
        DropEverything();
# if NOTIFY               // 不要在#后跟空格
        NotifyClient();
# endif
#endif
        BackToNormal();
    }
```

6.8.17 类格式

访问控制块的声明依次序是 public:, protected:, private:, 每个不缩进。

```
class TMyClass : public TOtherClass {
public:
    TMyClass();  // 标准的 4 空格缩进
    explicit TMyClass(int var);
```

```
    ~TMyClass() {}

    void SomeFunction();
    void SomeFunctionThatDoesNothing() {}

    void SetSomeVar(int var) {some_var_ = var;}
    int SomeVar() const {return some_var_;}

private:
    bool SomeInternalFunction();
    int some_var_;
    int some_other_var_;
};
```

6.8.18 构造函数初始值列表

构造函数初始化列表放在同一行或按四格缩进并排多行。

```
// 如果所有变量能放在同一行:
TMyClass::TMyClass(int var) : some_var_(var) {
    DoSomething();
}
// 如果不能放在同一行,
// 必须置于冒号后，并缩进 4 个空格
TMyClass::TMyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
```

尽量放下面。

```
    DoSomething();
}
// 如果初始化列表需要置于多行，将每一个成员放在单独的一行
// 并逐行对齐
TMyClass::TMyClass(int var)
```

```
    : some_var_(var),              // 4 space indent
      some_other_var_(var + 1) {  // lined up
  DoSomething();
}
// 右大括号 } 可以和左大括号 { 放在同一行
// 如果这样做合适的话
TMyClass::TMyClass(int var)
: some_var_(var) {
}
```

6.8.19 命名空间格式化

命名空间内容不缩进。

```
namespace {
void Foo() {  // 正确. 命名空间内没有额外的缩进.
    ...}
}  // namespace
```

声明嵌套命名空间时，每个命名空间都独立成行。

```
namespace foo {
namespace bar {
```

6.8.20 水平留白

水平留白的使用根据在代码中的位置决定。永远不要在行尾添加没意义的留白。

6.8.21 空行

以功能子块为单位，进行空行，不要有超过 2 行的空行。

7 相关文件

无

8 相关记录

无

9 附录

无