1. **代码通用格式**
2. 每行代码不多于80个字符；
3. 使用空格，而不是制表符(Tab)来缩进（QT中Tab=4个字符），每次缩进4个字符；
4. 指针符号\*，引用符号&写在靠近类型的位置；
5. 花括号的位置，使用Allman风格，另起一行，代码会更清晰；

for (auto i = 0; i < 100; i++)

{

printf("%d\n", i);

}

1. if、for、while等语句就算只有一行，也要强制使用花括号；

//永远不要省略花括号

if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)

goto fail;

//需要写成：

if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)

{

goto fail;

}

1. **注释格式（便于使用doxygen等第三方工具导出代码说明文档）**
2. 注释命令说明

| **注释命令** | **描述** |
| --- | --- |
| @brief | 概要信息 |
| @details | 详细描述 |
| @par | 开始一个段落，段名自定义 |
| @param | 标记参数意义 |
| @return | 描述返回意义 |
| @retval | 描述返回值意义 |
| @todo | 对将要做的事情进行注释 |
| @bug | 缺陷，链接到所有缺陷汇总的缺陷列表 |
| @since | 通常用来说明从什么版本、时间写此部分代码 |
| @pre | 用来说明代码项的前提条件 |
| @post | 用来说明代码项之后的使用条件 |
| @code | 在注释中开始说明一段代码，直到@endcode命令 |
| @endcode | 注释中代码段的结束 |
| @fn | 函数说明 |
| @include | 包含文件 |
| @var | 标注变量 |
| @enum | 标注枚举 |
| @struct | 标注结构体 |
| @class | 标注类 |
| @note | 描述一些注意事项 |
| @addtogroup | 添加到一个组 |
|  |  |
| //! 描述内容 | 简要描述（在变量、方法上方） |
| /\*! 描述内容 \*/ | 补充描述（在变量、方法上方） |
| /\*!  \param a 描述内容  \return b 描述  \*/ | 对函数的参数 a 描述，返回值 b 描述 |
| /\*!< 枚举值描述. \*/ | 每个枚举值描述 |
| ///< 变量描述 | 单个变量注释 |

1. C/C++在头文件（.h）需进行版权、版本、作者等声明。

版权和版本的声明位于头文件和.cpp文件的开头（参见示例），主要内容有：

1. 版权信息
2. 文件名称，摘要
3. 当前版本号，作者，日期

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Copyright (C), 2010-2019, GOSUN CL

\*

\* @file 文件名(包含后缀名)

\* @brief 简要描述本文件的内容

\* @author 作者

\* @date 输入日期

\* @version 版本号

\*

\* （若本次是第一个定型版本，可以省略下列Note内容）

\*

\*----------------------------------------------------------------------------

\* @note 历史版本 修改人员 修改日期 修改内容

\* @note

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

///以下是例子，供参考///

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Copyright (C), 2010-2019, GOSUN CL

\*

\* @file CLUtility.h

\* @brief 各种通用小功能/通用文件操作

\* 1. 小功能：新增 正则匹配/查找下一个字符/文件夹（全部）创建

\* 2. 重新封装: CECDateTime/CECString/CECFile/CECLogFile/

\* 3. 新增功能模块：CECRunLog(运行日志)/CECReadIni(ini文件读取)

\*

\* @author gu

\* @date 2019-07-18

\* @version V1.1.0 20190718

\*----------------------------------------------------------------------------

\* @note 历史版本 修改人员 修改日期 修改内容

\* @note v1.0 gu 2018-2-15 1.创建

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

1. 函数声明

每个函数（除了构造跟析构可以不用），都应该在函数上方进行注释，注释内容要与实际内容含义保持一致。

PS：如果函数功能或输入参数等信息发生变化，需要及时修改注释信息

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @brief 描述该函数主要功能

\*

\* @param[in] 参数名（与函数里的参数一致） 参数的描述 in：表示输入类型

\* @param[out] 参数名（与函数里的参数一致） 参数的描述 out：表示输出类型

\* @param[in,out] 参数名（与函数里的参数一致） 参数的描述 in,out：表示既是输入又是输出

\*

\* @return 返回值的类型说明

\* @retval 返回值的意义

\* @author （若与文件申明作者是不一样的，需要填写；否则，可以省略此项）

\*

\* （若本次是第一个定型版本，可以省略下列Note内容）

\*----------------------------------------------------------------------------

\* @note 历史版本 修改人员 修改日期 修改内容

\* @note v1.0 gu 2019-5-15 1.创建

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

///以下是例子，供参考///

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @brief 对外接口，用于new创建对象（实例CRunLog）

\*

\* @param[in] strPath 存放路径（完整）

\* @param[in] nMaxDay 最多存放多少天

\* @param[in] nLenDay 每天多少长度（字节）

\*

\* @return CRunLog的实例，指针

\*----------------------------------------------------------------------------

\* @note 历史版本 修改人员 修改日期 修改内容

\* @note v1.0 gu 2019-5-15 1.创建

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

1. **命名约定**
2. **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。

所有类型命名，包括类, 结构体, 类型定义 (typedef), 枚举, 类型模板参数等均使用此命名规则，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 { ...

1. **普通变量命名**
2. 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.

如: a\_local\_variable, a\_struct\_data\_member

1. 类的成员变量以下划线结尾, 但结构体的就不用, 如: a\_class\_data\_member\_.

class TableInfo {

...

private:

string table\_name\_; // 好 - 后加下划线.

string tablename\_; // 好.

static Pool<TableInfo>\* pool\_; // 好.

};

struct UrlTableProperties {

string name;

int num\_entries;

static Pool<UrlTableProperties>\* pool;};

1. 坚决不能用以下命名方式，此方式是标准库中声明标识符以供内部使用的方式，编译器可能会预先定义这种非标准的标识符，如果程序员使用了这种标识符，其行为是未定义的。
2. 包含两个连续下划线的标识符
3. 以一个下划线加一个大写字母开头的标识符
4. 顶级命名空间内的以一个下划线开头的标识符
5. **静态成员**

类中尽量不要出现静态变量。类中的静态变量不用加任何前缀。文件中的静态变量统一加s\_前缀，并尽可能的详细命名。比如

static ColorTransformStack s\_colorTransformStack; // 对static ColorTransformStack s\_stack; // 错（太简略）

1. **全局变量**

不要使用全局变量。真的没有办法，加上前缀 g\_，并尽可能的详细命名。比如

Document g\_currentDocument;

1. **函数命名**

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。

函数名整体上应该是个动词，或者是形容词(返回bool的函数)，不要是名词。

teacherNames(); // 错（这个是总体是名词）

GetTeacherNames(); // 对

无论是全局函数，静态函数，私有的成员函数，都不强制加前缀。

1. **命名空间**

命名空间名字，使用小写加下划线的形式；

namespace lua\_wrapper;

使用小写加下划线，而不要使用骆驼命名法。可以方便跟类型名字区分开来。比如

lua\_wrapper::GetField(); // GetField是命令空间lua\_wrapper的函数

LuaWrapper::GetField(); // GetField是类型LuaWrapper的静态函数

1. **宏命名**

不建议使用宏，除非真的需要。宏的名字全部大写，中间使用下划线相连。

头文件的防御宏定义

#ifndef \_\_COCOS2D\_FLASDK\_H\_\_

#define \_\_COCOS2D\_FLASDK\_H\_\_

....

#endif

1. **枚举命名**

尽量使用 0x11 风格 enum，枚举值采用首字母大写方式或全部大写, 使用下划线，例如：

**enum** UrlTableErrors {

OK = 0,

ErrorOutOfMemory,

ErrorMalformedInput,};

**enum** AlternateUrlTableErrors {

OK = 0,

OUT\_OF\_MEMORY = 1,

MALFORMED\_INPUT = 2,};

enum class ColorType : uint8\_t

{

Black,

While,

Red,

}

有些时候，需要使用0x11之前的enum风格，这种情况下，每个枚举值，都需要带上类型信息，用下划线分割。比如

enum HttpResult

{

HttpResult\_OK = 0,

HttpResult\_Error = 1,

HttpResult\_Cancel = 2,

}

1. **纯C风格的接口**

假如我们需要结构里面的内存布局精确可控，有可能需要编写一些纯C风格的结构和接口。这个时候，接口前面应该带有模块或者结构的名字，中间用下划线分割。比如

struct HSBColor

{

float h;

float s;

float b;

};

struct RGBColor

{

float r;

float g;

float b;

}

RGBColor color\_hsbToRgb(HSBColor hsb);

HSBColor color\_rgbToHsb(RGBColor rgb);

这里，color 就是模块的名字。这里的模块，充当 C++ 中命名空间的作用。

1. **代码文件**
2. #define保护

所有的头文件，都应该使用#define来防止头文件被重复包含。命名的格式为：

\_\_<模块>\_<文件名>\_H\_\_

很多时候，模块名字都跟命名空间对应。比如

#ifndef \_\_GEO\_POINT\_H\_\_

#define \_\_GEO\_POINT\_H\_\_

namespace geo

{

class Point

{

.....

};

}

#endif

1. #include的顺序

C++代码使用#include来引入其它的模块的头文件。尽可能，按照模块的稳定性顺序来排列#include的顺序。按照稳定性从高到低排列。比如：

#include <map>

#include <vector>

#include <boost/noncopyable.hpp>

#include "cocos2d.h"

#include "json.h"

#include "FlaSDK.h"

#include "support/TimeUtils.h"

#include "Test.h"

上面例子中。#include的顺序，分别是C++标准库，boost库，第三方库，我们自己写的跟工程无关的库，工程中比较基础的库，应用层面的文件。

但有一个例外，就是 .cpp中，对应的.h文件放在第一位。比如geo模块中的, Point.h 跟 Point.cpp文件，Point.cpp中的包含

#include "geo/Point.h"

#include <cmath>

这里，将 #include "geo/Point.h"，放到第一位，之后按照上述原则来排列#include顺序。理由下一条规范来描述。

1. 尽可能减少对头文件的依赖

代码文件中，每多出现一次#include包含, 就会多一层依赖。比如，有A，B类型，各自有对应的.h文件和.cpp文件。

当A.cpp包含了A.h, A.cpp就依赖了A.h，当A.h被修改的时候，A.cpp就需要重修编译。

若B.cpp 包含了B.h, B.h包含了A.h, 这个时候。B.cpp虽然没有直接包含A.h, 但也间接依赖于A.h。当A.h修改了，B.cpp也需要重修编译。

当在头文件中，出现不必要的包含，就会生成不必要的依赖，引起连锁反应，使得编译时间大大被拉长。

使用前置声明，而不是直接#include，可以显著地减少依赖数量。

1. #include中的头文件，尽量使用全路径，或者相对路径

路径的起始点，为工程文件代码文件的根目录。

#include "ui/home/HomeLayer.h"

#include "ui/home/HomeCell.h"

#include "support/MathUtils.h"

不要直接包含：

#include "HomeLayer.h"

#include "HomeCell.h"

#include "MathUtils.h"

也可以使用相对路径。比如

#include "../MathUtil.h"

#include "./home/HomeCell.h"

1. **作用域**

作用域，表示某段代码或者数据的生效范围。作用域越大，修改代码时候影响区域也就越大，原则上，作用域越小越好。

1. 全局变量

禁止使用全局变量。若真需要使用全局变量。变量使用g\_开头。

全局变量在项目的任何地方都可以访问。两个看起来没有关系的函数，一旦访问了全局变量，就会产生无形的依赖。使用全局变量，基本上都是怕麻烦，贪图方便。比如：

funA -> funB -> funC -> funD

上图表示调用顺序。当funD需要用到funA中的某个数据。正确的方式，是将数据一层层往下传递。但因为这样做，需要修改几个地方，修改的人怕麻烦，直接定义出全局变量。这样做，当然是可以快速fix bug。但funA跟funD就引入无形的依赖，从接口处看不出来。

单件可以看做全局变量的变种。最优先的方式，应该将数据从接口中传递，其次封装单件，再次使用函数操作静态数据，最糟糕就是使用全局变量。

1. 类的成员变量
2. 类的成员变量，只能够是private或者public, 不要设置成protected。protected的数据看似安全，实际只是一种错觉。
3. 数据只能通过接口来修改访问，不要直接访问。这样的话，在接口中设置个断点就可以调试知道什么时候数据被修改。另外改变类的内部数据表示，也可以维持接口的不变，而不影响全局。
4. 系统头文件里将宏名、变量名、内部函数名用 \_ 开头就是为了避免与用户用的名字冲突，我们写程序时一定不要用 \_ 开头的名字，以免与系统头文件的名字冲突，产生奇怪的问题。
5. 绝大多数情况，数据都应该设置成私有private, 变量加后缀\_。比如：

class Data

{private:

const uint8\_t\* bytes\_;

size\_t size\_;

}

1. 公有的数据，通常出现在C风格的结构中，或者一些数据比较简单，并很常用的类，public数据不要加后缀。

class Point

{public:

Point(float x\_, float y\_) : x(x\_), y(y\_)

{

}

.....

float x;

float y;

}

注意，我们在构造函数，使用 x\_ 的方式表示传入的参数，防止跟 x 来重名。

1. 局部变量

局部变量真正需要使用的时候才定义，一行定义一个变量，并且一开始就给它一个合适的初始值。

（在函数最前面定义变量，变量就在整个函数都可见，作用域越大，就越容易被误修改。）

1. 命名空间

C++中，尽量不要出现全局函数，应该放入某个命名空间当中。命名空间将全局的作用域细分，可有效防止全局作用域的名字冲突。

比如：

namespace json

{

class Value

{

....

}

}

namespace splite

{

class Value

{

...

}

}

两个命名空间都出现了Value类。外部访问时候，使用 json::Value, splite::Value来区分。

1. 文件作用域

详见

1. 头文件中不要出现using namespace ...

头文件，很可能被多个文件包含。当某个头文件出现了 using namespace ... 的字样，所有包含这个头文件的文件，都简直看到此命令空间的全部内容，就有可能引起冲突。

// Test.h

#include <string>using namespace std;

class Test

{public:

Test(const string& name);

};

这个时候，只要包含了Test.h, 就都看到std的所有内容。正确的做法，是头文件中，将命令空间写全。将 string, 写成 std::string, 这里不要偷懒。

1. **类**
2. 让类的接口尽可能小

设计类的接口时，不要想着接口以后可能有用就先加上，而应该想着接口现在没有必要，就直接去掉。这里的接口，你可以当成类的成员函数。添加接口是很容易的，但是修改，去掉接口会会影响较大。

接口小，不单指成员函数的数量少，也指函数的作用域尽可能小。

比如，

class Test

{public:

void funA();

void funB();

void funC();

void funD();

};

假如，funD 其实是可以使用 funA, funB, funC 来实现的。这个时候，funD，就不应该放到Test里面。可以将funD抽取出来。funD 只是一个封装函数，而不是最核心的。

void Test\_funD(Test\* test);

编写类的函数时候，一些辅助函数，优先采用 Test\_funD 这样的方式，将其放到.cpp中，使用匿名空间保护起来，外界就就不用知道此函数的存在，那些都只是实现细节。

当不能抽取独立于类的辅助函数，先将函数，变成private, 有必要再慢慢将其提出到public。 不要觉得这函数可能有用，一下子就写上一堆共有接口。

再强调一次，如无必要，不要加接口。

从作用域大小，来看

* 独立于类的函数，比类的成员函数要好
* 私有函数，比共有函数要好
* 非虚函数，比虚函数要好

1. 声明顺序

类的成员函数或者成员变量，按照使用的重要程度，从高到低来排列。

比如，使用类的时候，用户更关注函数，而不是数据，所以成员函数应该放到成员变量之前。 再比如，使用类的时候，用户更关注共有函数，而不是私有函数，所以public，应该放在private前面。

具体规范

* 按照 public, protected, private 的顺序分块。那一块没有，就直接忽略。

每一块中，按照下面顺序排列

* typedef，enum，struct，class 定义的嵌套类型
* 常量
* 构造函数
* 析构函数
* 成员函数,含静态成员函数
* 数据成员,含静态数据成员

.cpp 文件中，函数的实现尽可能给声明次序一致。

1. 继承

优先使用组合，而不是继承。

继承主要用于两种场合：实现继承，子类继承了父类的实现代码。接口继承，子类仅仅继承父类的方法名称。

我们不提倡实现继承，实现继承的代码分散在子类跟父亲当中，理解起来变得很困难。通常实现继承都可以采用组合来替代。

规则：

* 继承应该都是 public
* 假如父类有虚函数，父类的析构函数为 virtual
* 假如子类覆写了父类的虚函数，应该显式写上 override

 比如：

// swf/Definition.h

class Definition

{public:

virtual ~Definition() {}

virtual void parse(const uint8\_t\* bytes, size\_t len) = 0;

};

// swf/ShapeDefinition.h

class ShapeDefinition : public Definition

{public:

ShapeDefinition() {}

virtual void parse(const uint8\_t\* bytes, size\_t len) override;

private:

Shape \_shape;

};

Definition\* p = new ShapeDefinition();

....delete p;

上面的例子，使用父类的指针指向子类，假如父类的析构函数不为virtual, 就只会调用父类的Definition的释放函数，引起子类独有的数据不能释放。所有需要加上virtual。

另外子类覆写的虚函数写上，override的时候，当父类修改了虚函数的名字，就会编译错误。从而防止，父类修改了虚函数接口，而忘记修改子类相应虚函数接口的情况。

1. **函数**
2. 编写短小的函数

函数尽可能的短小，凝聚，功能单一。

只要某段代码，可以用某句话来描述，尽可能将这代码抽取出来，作为独立的函数，就算那代码只有一行。最典型的就是C++中的max, 实现只有一句话。

template <typename T>

inline T max(T a, T b)

{

return a > b ? a : b;

}

* 将一段代码抽取出来，作为一个整体，一个抽象，就不用纠结在细节之中。
* 将一个长函数，切割成多个短小的函数。每个函数中使用的局部变量，作用域也会变小。
* 短小的函数，更容易复用，从一个文件搬到另一个文件也会更容易。
* 短小的函数，因为内存局部性，运行起来通常会更快。
* 短小的函数，也容易阅读，调试。

1. 函数的参数尽可能少，原则上不超过5个
2. 函数参数顺序

参数顺序按照传入参数，传出参数的顺序排列

1. 函数的传出参数，使用指针，而不要使用引用

比如：

bool loadFile(const std::string& filePath, ErrorCode\* code); // 对bool loadfile(const std::string& filePath, ErrorCode& code); // 错

因为当使用引用的时候，使用函数的时候会变成

ErrorCode code;if (loadFile(filePath, code))

{

...

}

而使用指针，调用的时候，会是

ErrorCode code;if (loadFile(filePath, &code))

{

...

}

这样从，&code的方式可以很明显的区分，传入，传出参数。试比较

doFun(arg0, arg1, arg2); // 错

doFun(arg0, &arg1, &arg2); // 对

1. **其他**
2. const

建议，尽可能多使用const

C++中，const是个很重要的关键字，应用了const之后，就不可以随便改变变量的数值了，不小心改变了编译器会报错，就容易找到错误的地方。只要你觉得有不变的地方，就用const来修饰；

1. 不要注释代码，代码不使用就直接删掉

要想看之前的代码，可以通过版本控制工具；