编程规范

♪4 总体原则

- 清晰，易于维护、易于重构。
- 简洁，易于理解，并且易于实现。
- 风格统一，代码整体风格保持统一。
- 通用性，遵循业界通用的编程规范。

♪4 命名

- 使用驼峰风格进行命名，此风格大小写字母混用，不同单词间通过单词首字母大写来分开，具体规则如下：

类型	命名风格	形式
函数，自定义的类型	大驼峰，或带有模块前缀的大驼峰	AaaBbb, XXX_AaaBbb
局部变量，函数参数，宏参数，结构体成员，联合体成员	小驼峰	aaaBbb
全局变量	带'g_'前缀的小驼峰	g_aaaBbb
宏，枚举值	全大写并下划线分割	AAA_BBB
内核头文件中防止重复包含的宏变量	带'_LOS'前缀和'H'后缀，中间为大写模块名，以下划线分割	_LOS_MODULE_H

- 全局函数、全局变量、宏、类型名、枚举名的命名，应当准确描述并全局唯一。
- 在能够准确表达含义的前提下，局部变量，或结构体、联合体的成员变量，其命名应尽可能简短。

♪4 排版与格式

- 程序块采用缩进风格编写，使用空格而不是制表符（'\t'）进行缩进，每级缩进为4个空格。
- 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如->），后不应加空格
  采用这种松散方式编写代码的目的是使代码更加清晰。
  在已经非常清晰的语句中没有必要再留空格，如括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C语言中括号已经是最清晰的标志了。在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。
  正确示例：
  1、逗号、分号只在后面加空格。

```
int a, b, c;
```

2、比较操作符，赋值操作符"="、"+="，算术操作符"+"、"%"，逻辑操作符"&&"、"||"，位域操作符"<<"、"^"等双目操作符的前后加空格。

```
if (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

3、"!"、"~"、"++"、"--"、"&"(地址操作符)等单目操作符前后不加空格。

```
*p = 'a'; // 内容操作"*"与内容之间
flag = !is_empty; // 非操作"!"与内容之间
p = &mem; // 地址操作"&" 与内容之间
i++;
```

4、"->"、"."前后不加空格。

```
p->id = pid; // "->"指针前后不加空格
```

5、if、for、while、switch等与后面的括号间应加空格,使if等关键字更为突出、明显。

```
if (a >= b && c > d)
```

6、注释符(包括/**/、//)与注释内容之间要用一个空格进行分隔。

- 采用K&R风格作为大括号换行风格,即函数左大括号另起一行放行首,并独占一行,其他左大括号跟随语句放行末,右大括号独占一行,除非后面跟着同一语句的剩余部分,如if语句的else/else if或者分号,比如:

```
struct MyType {   // 左大括号跟随语句放行末,前置1个空格
    ...
};                // 右大括号后面紧跟分号
```

```
int Foo(int a)
{                 // 函数左大括号独占一行,放行首
    if (a > 0) {  // 左大括号跟随语句放行末,前置1个空格
        ...
    } else {      // 右大括号、"else"、以及后续的左大括号均在同一行
        ...
    }             // 右大括号独占一行
    ...
}
```

- 条件、循环语句使用大括号,比如:

```
if (objectIsNotExist) {       // 单行条件语句也加大括号
    return CreateNewObject();
}
```

```
while (condition) {} // 即使循环体是空,也应使用大括号
```

```
while (condition) {
    continue;         // continue表示空逻辑,使用大括号
}
```

- case/default语句相对switch缩进一层,缩进风格如下:

```
switch (var) {
    case 0:             // 缩进一层
        DoSomething1(); // 缩进一层
        break;
    case 1:
        DoSomething2();
        break;
    default:
        break;
}
```

- 一行只写一条语句。
- 一条语句不能过长,建议不超过120个字符,如不能缩短语句则需要分行写。
- 换行时将操作符留在行末,新行进行同类对齐或缩进一层,比如:

```
// 假设下面第一行不满足行宽要求
if (currentValue > MIN &&  // 换行后,布尔操作符放在行末
    currentValue < MAX) {  // 与(&&)操作符的两个操作数同类对齐
    DoSomething();
    ...
}
```

```
// 假设下面的函数调用不满足行宽要求,需要换行
ReturnType result = FunctionName(paramName1,
                                 paramName2,
                                 paramName3); // 保持与上方参数对齐
```

```
ReturnType result = VeryVeryVeryLongFunctionName( // 写入第1个参数后导致过长,直接换行
    paramName1, paramName2, paramName3);          // 换行后,4空格缩进一层
```

```
// 每行的参数代表一组相关性较强的数据结构,放在一行便于理解,此时可理解性优先于格式排版要求
int result = DealWithStructLikeParams(left.x, left.y,    // 表示一组相关参数
                                      right.x, right.y); // 表示另外一组相关参数
```

- 声明定义函数时,函数的返回类型以及其他修饰符,与函数名同行。
- 指针类型"*"应该靠右跟随变量或者函数名,比如:

```
int *p1;  // Good:右跟随变量,和左边的类型隔了1个空格
int* p2;  // Bad:左跟随类型
int*p3;   // Bad:两边都没空格
int * p4; // Bad:两边都有空格
```

当"*"与变量或函数名之间有其他修饰符,无法跟随时,此时也不要跟随修饰符,比如:

```
char * const VERSION = "V100";     // Good:当有const修饰符时,"*"两边都有空格
int Foo(const char * restrict p); // Good:当有restrict修饰符时,"*"两边都有空格
```

- 根据上下内容的相关程度,合理安排空行,但不要使用连续3个或更多空行。
- 编译预处理的"#"统一放在行首,无需缩进。嵌套编译预处理语句时,"#"可以进行缩进,比如:

```
#if defined(__x86_64__) && defined(__GCC_HAVE_SYNC_COMPARE_AND_SWAP_16) // 位于行首,不缩进
    #define ATOMIC_X86_HAS_CMPXCHG16B 1                                 // 缩进一层,区分层次,便于阅读
#else
    #define ATOMIC_X86_HAS_CMPXCHG16B 0
#endif
```

♪⁴ 注释

- 注释的内容要清楚、明了,含义准确,防止注释二义性。
- 在代码的功能、意图层次上进行注释,即注释解释代码难以直接表达的意图,而不是仅仅重复描述代码。
- 函数声明处注释描述函数功能、性能及用法,包括输入和输出参数、函数返回值、可重入的要求等;定义处详细描述函数功能和实现要点,如实现的简要步骤、实现的理由、设计约束等。
- 全局变量要有较详细的注释,包括对其功能、取值范围以及存取时注意事项等的说明。
- 避免在注释中使用缩写,除非是业界通用或子系统内标准化的缩写。
- 文件头部要进行注释,建议注释列出:版权说明、版本号、生成日期、作者姓名、功能说明、与其它文件的关系、修改日志等。
- 注释风格要统一,建议优先选择/* */的方式,注释符与注释内容之间要有1空格,单行、多行注释风格如下:

```
/* 单行注释 */
```

```
/*
 * 多行注释
 * 第二行
 */
```

- 注释应放在其代码上方或右方。
  上方的注释，与代码行之间无空行，保持与代码一样的缩进。右边的注释，与代码之间至少相隔1个空格。如果有多条右置注释，上下对齐会更加美观，比如：

```
#define A_CONST 100          // 此处两行注释属于同类
#define ANOTHER_CONST 200    // 可保持左侧对齐
```

♪4 宏

- 代码片段使用宏隔离时，统一通过#ifdef的方式，例如：

```
#ifdef LOSCFG_XXX
...
#endif
```

- 定义宏时，要使用完备的括号，比如：

```
#define SUM(a, b) a + b       // 不符合本条要求
#define SUM(a, b) ((a) + (b)) // 符合本条要求
```

  但是也要避免滥用括号，比如单独的数字或标识符加括号毫无意义：

```
#define SOME_CONST  100          // 单独的数字无需括号
#define ANOTHER_CONST   (-1)     // 负数需要使用括号
#define THE_CONST   SOME_CONST   // 单独的标识符无需括号
```

- 包含多条语句的函数式宏的实现语句必须放在do-while(0)中，例如：

```
#define FOO(x) do { \
    (void)printf("arg is %d\n", (x)); \
    DoSomething((x)); \
} while (0)
```

- 禁止宏调用参数中出现预编译指令。
- 宏定义不以分号结尾。

♪4 头文件

- 设计原则
  - 头文件应当职责单一。
  - 一个模块通常包含多个.c文件，建议放在同一个目录下，目录名即为模块名；如果一个模块包含多个子模块，则建议每一个子模块提供一个对外的.h，文件名为子模块名。
  - 建议每一个.c文件应有一个同名.h文件，用于声明需要对外公开的接口。
  - 头文件中适合放置接口的声明，不适合放置实现。
  - 不要在头文件中定义变量。
  - 禁止头文件循环依赖，循环依赖指a.h包含b.h，b.h包含c.h，c.h包含a.h。
  - 头文件应当自包含，即任意一个头文件均可独立编译，但同时也要避免包含用不到的头文件。
  - 头文件必须用#define保护，防止重复包含，比如内核中统一使用以下宏定义保护：

```
#ifndef _LOS_<MODULE>_H  // 比如 _LOS_TASK_H
#define _LOS_<MODULE>_H
...
#endif
```

- 禁止通过声明的方式引用外部函数接口、变量，只能通过包含头文件的方式使用其他模块或文件提供的接口。
- 禁止在 extern "C" 中包含头文件。
- 按照合理的顺序包含头文件：
  1. 源文件对应的头文件
  2. C标准库
  3. 需要包含的OS其他头文件

♪4 变量

- 一个变量只有一个功能，不要把一个变量用作多种用途。
- 防止局部变量与全局变量同名。
- 不用或者少用全局变量。
- 定义函数的局部变量时，控制变量的占用空间，避免因占用过多栈空间导致程序运行失败。比如需要一个大数组，可以通过动态分配内存的方式来避免栈空间占用过大。
- 在首次使用前初始化变量。
- 指向资源句柄或描述符的变量，在资源释放后立即赋予新值，包括指针、socket描述符、文件描述符以及其它指向资源的变量。
- 禁止将局部变量的地址返回到其作用域以外，下面是一个错误示例：

```
int *Func(void)
{
    int localVar = 0;
    ...
    return &localVar;  // 错误
}
void Caller(void)
{
    int *p = Func();
    ...
    int x = *p;        // 程序产生未定义行为
}
```

正确代码示例：

```
int Func(void)
{
    int localVar = 0;
    ...
    return localVar;
}
void Caller(void)
{
    int x = Func();
    ...
}
```

- 如果要使用其他模块的变量，应尽量避免直接对变量进行访问，而是通过统一的函数封装或者宏封装的方式，比如mutex模块中：

```
// 私有头文件中引入全局变量，但要避免直接使用
extern LosMuxCB *g_allMux;
// 通过GET_MUX的方式对g_allMux进行访问
#define GET_MUX(muxID)  (((LosMuxCB *)g_allMux) + GET_MUX_INDEX(muxID))
```

♪⁴ 函数

- 重复代码应该尽可能提炼成函数。
- 避免函数过长，新增函数不超过 40-50 行。
- 内联函数要尽可能短，避免超过 10 行（非空非注释）。
- 避免函数的代码块嵌套过深。
- 函数应避免使用全局变量、静态局部变量和I/O操作，不可避免的地方应集中使用。