安富莱C语言编码规范

1.1 文件与目录

（1） 文件及目录的命名规定可用的字符集是[A-Z；a-z；0-9；._-]。

（2） 源文件名后缀用小写字母 .c 和.h。

（3） 文件的命名要准确清晰地表达其内容，同时文件名应该精练，防止文件名过长而造成使用不便。在文件名中可以适当地使用缩写。

以下提供两种命名方式以供参考：

- 各程序模块的文件命名开头 2 个消协字母代表本模块的功能：

  如：主控程序为 mpMain.c，mpDisp.c …

- 不写模块功能标识：

  如：主控程序为 Main.c，Disp.c …

（4） 一个软件包或一个逻辑组件的所有头文件和源文件建议放在一个单独的目录下，这样有利于查找并使用相关的文件，有利于简化一些编译工具的设置。

（5） 对于整个项目需要的公共头文件，应存放在一个单独的目录下（例如：myProject/include）下，可避免其他编写人引用时目录太过分散的问题。

（6） 对于源码文件中的段落安排，我们建议按如下的顺序排列：

- 文件头注释
- 防止重复引用头文件的设置
- #include 部分
- #define 部分
- enum 常量声明
- 类型声明和定义，包括 struct、union、typedef 等
- 全局变量声明
- 文件级变量声明

- 全局或文件级函数声明
- 函数实现。按函数声明的顺序排列
- 文件尾注释

（7） 在引用头文件时，不要使用绝对路径。如果使用绝对路径，当需要移动目录时，必须修改所有相关代码，繁琐且不安全；使用相对路径，当需要移动目录时，只需修改编译器的某个选项即可。例如：

```
#include “/project/inc/hello.h”      /* 不应使用绝对路径 */
#include “../inc/hello.h”           /* 可以使用相对路径 */
```

（8） 在引用头文件时，使用 <> 来引用预定义或者特定目录的头文件，使用 “” 来引用当前目录或者路径相对于当前目录的头文件。

```
#include <stdio.h>          /* 标准头文件 */
#include <projdefs.h>       /* 工程指定目录头文件 */
#include “global.h”         /* 当前目录头文件 */
#include “inc/config.h”     /* 路径相对于当前目录的头文件 */
```

（9） 为了防止头文件被重复引用，应当用 ifndef/define/endif 结构产生预处理块。

```
#ifndef __DISP_H        /* 文件名前名加两个下划线“__”，后面加 “_H”
#define __DISP_H
...
...
#endif
```

（10）头文件中只存放“声明”而不存放“定义”，通过这种方式可以避免重复定义。

```
/* 模块 1 头文件： module1.h */
extern int a = 5;               /* 在模块 1 的 .h 文件中声明变量 */
```

```
/* 模块 1 实现文件：module1.c */
uint8_t g_ucPara;    /* 在模块 1 的 .h 文件中定义全局变量 g_ucPara */
```

（11）如果其它模块需要引用全局变量 g_ucPara，只需要在文件开头包含 module1.h

```
/* 模块 2 实现文件：module2.c */
#include “module1.h”      /* 在模块 2 中包含模块 1 的 .h 文件 */
```

```
......
g_ucPara = 0;
......
```

（12）对于文件的长度没有非常严格的要求，但应尽量避免文件过长。一般来说，文件长度应尽量保持在 1000 行之内。

1.2 排版

（1） 程序块要采用缩进风格编写，缩进的空格数为 4 个。

（2） 相对独立的程序块之间、变量说明之后必须加空行。

```
void DemoFunc(void)
{
    uint8_t i;
                    <---- 局部变量和语句间空一行
    /* 功能块 1 */
    for (i = 0; i < 10; i++)
    {
        ...
    }
                    <---- 不同的功能块间空一行
    /* 功能块 2 */
    for (i = 0; i < 20; i++)
    {
        ...
    }
}
```

（3） 较长的语句或函数过程参数（>80 字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

```
    if ((ucParam1 == 0) && (ucParam2 == 0) && (ucParam3 == 0)
        || (ucParam4 == 0))        <---- 长表达式需要换行书写
    {
        ......
    }
```

（4） 不允许把多个短语句写在一行中，即一行只写一条语句。

```
rect.length = 0;  rect.width = 0;  <---- 不正确的写法

rect.length = 0;                   <---- 正确的写法
rect.width = 0;
```

（5） 对齐使用 TAB 键，1 个 TAB 对应 4 个字符位。

（6） 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case 语句下的情况处理语句也要遵从语句缩进要求。

（7） 程序块的分界符（如大括号‘{’和‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及 if、for、do、while、switch、case 语句中的程序都要采用如上的缩进方式。对于与规则不一致的现存代码，应优先保证同一模块中的风格一致性。

```
for (...) {                        <---- 不规范的写法
    ... /* program code */
}

for (...)
{                                  <---- 规范的写法
    ... /* program code */
}

if (...)
    {                              <---- 不规范的写法
    ... /* program code */
    }

if (...)
{                                  <---- 规范的写法
    ... /* program code */
}
```

（8） 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。

说明：采用这种松散方式编写代码的目的是使代码更加清晰。

由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(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。

示例：

- 逗号、分号只在后面加空格。

```
int_32 a, b, c;
```

- 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;
```

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

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

- "->"、"."前后不加空格。

```
p->id = pid;      /* "->"指针前后不加空格 */
```

- 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，函数名与其后

的括号之间不加空格，以与保留字区别开。

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

1.3 注释

（1） 一般情况下，源程序有效注释量必须在 20%以上。

说明：注释的原则是有助于对程序的阅读理解，在该加的地方都加，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

（2） 在文件的开始部分，应该给出关于文件版权、内容简介、修改历史等项目的说明。

具体的格式请参见如下的说明。在创建代码和每次更新代码时，都必须在文件的历史记录中标注版本号、日期、作者、更改说明等项目。其中的版本号的格式为两个数字字符和一个英文字母字符。数字字符表示大的改变，英文字符表示小的修改。如果有必要，还应该对其它的注释内容也进行同步的更改。注意：注释第一行星号要求为 76 个,结尾行星号为 1 个。

```
/*****************************************************************************
* Copyright (C), 2010-2011,武汉汉升汽车传感系统有限责任公司
* 文件名: main.c
* 内容简述:
*
* 文件历史:
* 版本号  日期          作者      说明
* 01a     2010-07-29    王江河    创建该文件
* 01b     2010-08-20    王江河    改为可以在字符串中发送回车符
* 02a     2010-12-03    王江河    增加文件头注释
*/
```

（3） 对于函数，在函数实现之前，应该给出和函数的实现相关的足够而精练的注释信息。内容包括本函数功能介绍，调用的变量、常量说明，形参说明，特别是全局、全程或静态变量（慎用静态变量），要求对其初值，调用后的预期值作详细的阐述。具体的书写格式和包含的各项内容请参见如下的例子。

示例：

下面这段函数的注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/*********************************************************************************
*   函数名: SendToCard()
*   功   能: 向读卡器发命令，如果读卡器进入休眠，则首先唤醒它
*   输   入: 全局变量 gaTxCard[]存放待发的数据
*            全局变量 gbTxCardLen 存放长度
*   输   出: 无
*/
```

（4） 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

（5） 注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：错误的注释不但无益反而有害。注释主要阐述代码做了什么（What），或者如果有必要的话，阐述为什么要这么做（Why），注释并不是用来阐述它究竟是如何实现算法（How）的。

（6） 避免在注释中使用缩写，特别是非常用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

（7） 注释应与其描述的代码靠近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

示例：如下例子不符合规范。

例 1：不规范的写法

```
/* 获取复本子系统索引和网络指示器 */
                                <---- 不规范的写法，此处不应该空行
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

例 2：不规范的写法

```
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* 获取复本子系统索引和网络指示器 */  <---- 不规范的写法，应该在语句前注释
```

例 3：规范的写法

```
/* 获取复本子系统索引和网络指示器 */
```

```
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

例 4：不规范的写法，显得代码过于紧凑

```
/* code one comments */
program code one
/* code two comments */            <---- 不规范的写法，两段代码之间需要加空行
program code two
```

例 5：规范的写法

```
/* code one comments */
program code one

/* code two comments */
program code two
```

（8） 注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

例 1：如下例子，排版不整齐，阅读稍感不方便。

```
void example_fun( void )
{
/* code one comments */    <---- 不规范的写法，注释和代码应该相同的缩进
    CodeBlock One

        /* code two comments */  <---- 不规范的写法，注释和代码应该相同的缩进
    CodeBlock Two
}
```

例 2：正确的布局。

```
void example_fun( void )
{
    /* code one comments */
    CodeBlock One

    /* code two comments */
    CodeBlock Two
```

```
}
```

（9） 对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

（10）对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

示例（注意斜体加粗部分）：

```
case CMD_DOWN:
    ProcessDown();
    break;

case CMD_FWD:
    ProcessFwd();

    if (...)
    {
        ...
        break;
    }
    else
    {
        ProcessCFW_B();   /* now jump into case CMD_A  */
    }

case CMD_A:
    ProcessA();
    break;
...
```

（11）注释格式尽量统一，建议使用“/* …… */”，因为 C++注释“//”并不被所有 C 编译器支持。

（12）注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非

能非常流利准确的用英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

标识符命名

（13）标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

说明：较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

示例：如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp;

flag 可缩写为 flg;

statistic 可缩写为 stat;

increment 可缩写为 inc;

message 可缩写为 msg;

（14）命名中若使用特殊约定或缩写，则要有注释说明。

说明：应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

（15）自己特有的命名风格，要自始至终保持一致，不可来回变化。

说明：个人的命名风格，在符合所在项目组或产品组的命名规则的前提下，才可使用。（即命名规则中没有规定到的地方才可有个人命名风格）。

（16）对于变量命名，禁止取单个字符（如 i、j、k…），建议除了要有具体含义外，还能表明其变量类型、数据类型等，但 i、j、k 作局部循环变量是允许的。

说明：变量，尤其是局部变量，如果用单个字符表示，很容易敲错（如i写成j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

（17）命名规范必须与所使用的系统风格保持一致，并在同一项目中统一，比如采用 UNIX 的全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的 m_和 g_，其后加上大小写混排的方式是允许的。

示例： Add_User不允许，add_user、AddUser、m_AddUser允许。

（18）除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：如下命名，使人产生疑惑。

```
uint8_t dat01;
void Set00(uint_8 c);
```

应改为有意义的单词命名。

```
uint8_t ucWidth;
void SetParam(uint_8 _ucValue);
```

（19）在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常量之前加上“模块”标识等。

（20）除了编译开关/头文件等特殊应用，应避免使用_EXAMPLE_TEST_之类以下划线开始和结尾的定义。

1.4 可读性

（1） 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例：下列语句中的表达式

正确的写法：

```
word = (high << 8) | low;
if ((a | b) && (a & c))
if ((a | b) < (c & d))
```

错误的写法：

```
word = high << 8 | low;
if (a | b && a & c)
if (a | b < c & d)          /* 造成了判断条件出错 */
```

（2） 避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

示例：如下的程序可读性差。

```
if (Trunk[index].trunk_state == 0)    <---- 不规范的写法，应使用有意义的标识
{
    Trunk[index].trunk_state = 1;     <---- 不规范的写法，应使用有意义的标识

    ...  /* program code  */
}
```

应改为如下形式。

```
enum trunk_state_e
{
    TRUNK_IDLE = 0,
    TRUNK_BUSY = 1
};
```

```
if (Trunk[index].trunk_state == TRUNK_IDLE)
{
    Trunk[index].trunk_state = TRUNK_BUSY;
    ...  /* program code  */
}
```

（3） 不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

示例：如下表达式，考虑不周就可能出问题，也较难理解。

```
    * stat_poi ++ += 1;

    * ++ stat_poi += 1;
```

应分别改为如下。

```
    *stat_poi += 1;
    stat_poi++;     /* 此二语句功能相当于“ * stat_poi ++ += 1; ”*/

    ++ stat_poi;
    *stat_poi += 1; /* 此二语句功能相当于“ * ++ stat_poi += 1; ”*/
```

1.5 变量、结构、常量、宏

（1） 为了方便书写及记忆，变量类型采用如下重定义：

```
typedef unsigned char       uint8_t;
typedef unsigned short      uint16_t;
typedef unsigned long int   uint32_t;

typedef signed char         int8_t;
typedef signed short        int16_t;
typedef signed long int     int32_t;

#define     __IO    volatile
```

（2） 常见类型的前缀

- 对于一些常见类型的变量，应在其名字前标注表示其类型的前缀。前缀用小写字母表示。前缀的使用请参照下列表格中说明。

变量类型	前缀	举例	变量类型	前缀	举例
uint8_t	**uc**	ucSum	int8_t	**c**	cByte
uint16_t	**us**	usParaWord	int16_t	**s**	sParaWord
uint32_t	**ul**	ulParaWord	int32_t	**l**	lParaWord
uint8_t *	**ucp**	ucpWrite	int8_t *	**cp**	cpWrite
uint16_t *	**usp**	uspWrite	int16_t *	**sp**	spWrite
uint32_t *	**ulp**	ulpWrite	int32_t *	**lp**	lpWrite
uint8_t 数组	**uca**	ucaNum[5]	int8_t 数组	**ca**	caNum[5]
uint16_t 数组	**usa**	usaNum[5]	int16_t 数组	**sa**	saNum[5]
uint32_t 数组	**ula**	ulaNum[5]	int32_t 数组	**la**	laNum[5]
结构体	**t**	tParam			

- 对于几种变量类型组合，前缀可以迭加。

（3） 变量作用域的前缀

为了清晰的标识变量的作用域，减少发生命名冲突，应该在变量类型前缀之前再加上表示变量作用域的前缀，并在变量类型前缀和变量作用域前缀之间用下划线 ‘-’ 隔开。

具体的规则如下：

- 对于全局变量（global variable），在其名称前加“g”和变量类型符号前缀。

```
uint32_t g_ulParaWord;
uint8_t g_ucByte;
```

- 对于静态变量（static variable），在其名称前加“s”和变量类型符号前缀。

```
static uint32_t s_ulParaWord;
static uint8_t s_ucByte;
```

- 函数内部等局部变量前不加作用域前缀。
- 对于常量，当可能发生作用域和名字冲突问题时，以上几条规则对于常量同样适用。注意，虽然常量名的核心部分全部大写，但此时常量的前缀仍然用小写字母，以保持前缀的一致性。

（4） 对于结构体命名类型，表示类型的名字，所有名字以小写字母“tag”开头，之后每个英文单词的第一个字母大写（包括第一个单词的第一个字母），其他字母小写，结尾_T 标识。单词之间不使用下划线分隔,结构体变量以 t 开头。

```
/* 结构体命名类型名 */

typedef struct tagBillQuery_T

{
```

```
    ...
}BillQuery_T;


/* 结构体变量定义 */
BillQuery_T tBillQuery;
```

1.1 对于枚举定义全部采用大写，结尾_E 标识。

```
typedef enum
{
    KB_F1 = 0,          /* F1 键代码 */
    KB_F2,              /* F2 键代码 */
    KB_F3               /* F3 键代码 */
}KEY_CODE_E;
```

（5） 常量、宏、模版的名字应该全部大写。如果这些名字由多个单词组成，则单词之间用下划线分隔。宏指所有用宏形式定义的名字，包括常量类和函数类；常量也包括枚举中的常量成员。

```
#define LOG_BUF_SIZE      8000
```

（6） 不推荐使用位域。

1.6 函数

（1） 函数的命名规则。每一个函数名前缀需包含模块名，模块名为小写，与函数名区别开。

如：uartReceive(串口接收)

备注：对于非常简单的程序，可以不加模块名。

（2） 函数的的形参需另启一行，在后面给予说明，形参都以下划线_开头，已示与普通变量进行区分，对于没有形参为空的函数(void)括号紧跟函数后面。

```
/*********************************************************************************
* 函数名：uartConvUartBaud
* 功   能：波特率转换
* 输   入： _ulBaud ： 波特率
* 输   出：无
* 返   回：uint32- 转换后的波特率值
*/
```

```
uint32_t uartConvUartBaud(uint32_t _ulBaud)
{
    uint32_t ulBaud;

    ulBaud = ulBaud * 2;     /* 计算波特率 */
    ......
    return ulBaud;
}
```

（3） 一个函数仅完成一件功能。

（4） 函数名应准确描述函数的功能。避免使用无意义或含义不清的动词为函数命名。使用动宾词组为执行某操作的函数命名。

说明：避免用含义不清的动词如process、handle等为函数命名，因为这些动词并没有说明要具体做什么。

示例：参照如下方式命名函数。

```
void PrintRecord(uint32_t _RecInd);
int32 InputRecord(void);
uint8_t GetCurrentColor(void);
```

（5） 检查函数所有参数输入的有效性。

说明：如果约定由调用方检查参数输入，则应使用assert()之类的宏，来验证所有参数输入的有效性。

（6） 检查函数所有非参数输入的有效性，如数据文件、公共变量等。

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入之前，应进行必要的检查。

（7） 防止将函数的参数作为工作变量。

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

（8） 避免设计五个以上参数函数，不使用的参数从接口中去掉。

说明：目的减少函数间接口的复杂度，复杂的参数可以使用结构传递。

（9） 在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。

说明：因为数据类型转换或多或少存在危险。

（10） 避免使用 BOOL 参数。

说明：原因有二，其一是BOOL参数值无意义，TURE/FALSE的含义是非常模糊的，在调用时很难知道该参数到底传达的是什么意思；其二是BOOL参数值不利于扩充。还有NULL也是一个无意义的单词。

（11） 函数的返回值要清楚、明了。除非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。

（12） 防止把没有关联的语句放到一个函数中。

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，否则这种构造将产生随机内聚的函数。

示例：如下函数就是一种随机内聚。

```
void InitVar( void )
{
    Rect.length = 0;
    Rect.width = 0; /* 初始化矩形的长与宽 */

    Point.x = 10;
    Point.y = 10;   /* 初始化“点”的坐标 */
}
```

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

```
void InitRect( void )
{
    Rect.length = 0;
    Rect.width = 0; /* 初始化矩形的长与宽 */
}

void InitPoint( void )
{
    Point.x = 10;
    Point.y = 10;   /* 初始化“点”的坐标 */
}
```

（13） 减少函数本身或函数间的递归调用。

说明：递归调用特别是函数间的递归调用（如A->B->C->A），影响程序的可理解性；递归调用一般

都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

（14） 改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。优化函数结构时，要遵守以下原则：

- 能影响模块功能的实现。
- 仔细考查模块或函数出错处理及模块的性能要求并进行完善。
- 通过分解或合并函数来改进软件结构。
- 考查函数的规模，过大的要进行分解。
- 降低函数间接口的复杂度。
- 不同层次的函数调用要有较合理的扇入、扇出。
- 函数功能应可预测。
- 提高函数内聚。（单一功能的函数内聚最高）

说明：对初步划分后的函数结构应进行改进、优化，使之更为合理。