DKBA

华为技术有限公司内部技术规范

DKBA 2826-2011.5

C语言编程规范

2011年5月9日发布　　2011　　年5月9日实施

华为技术有限公司

Huawei Technologies Co., Ltd.

修订声明 Revision declaration

本规范拟制与解释部门：

本规范的相关系列规范或文件：

相关国际规范或文件一致性：

替代或作废的其它规范或文件：

相关规范或文件的相互关系：

规范号	主要起草部门专家	主要评审部门专家	修订情况
DKBAxxxx.x-xxxx.xx	PSST 质量部： 郭曙光 00121837 网络： 张伟 00118807 周灿 00056781 王晶 00041937 陈艺彪 00036913 IP 开发部： 薛治 00038309 核心网： 张小林 00058208 王德喜 00040674 李明胜 00042021 软件公司： 文　滔00119601 无线： 刘爱华 00162172 中研： 谭洪 00162654	PSST 质量部： 李重霄 00117374 郭永生 00120218 核心网： 张进柏 00120359 中研： 张建保 00116237 无线： 苏光牛 00118740 郑铭 00118617 陶永祥 00120482 软件公司： 周代兵 00120359 刘心红 00118478 朱文琦 00172539 网络： 王玎 00168059 黄维东 49827 IP 开发部： 饶远 00152313	

目　录Table of Contents

C语言编程规范

范　　围:

本规范适用于公司内使用　　C语言编码的所有软件。本规范自发布之日起生效，以后新编写的和修改的代码应遵守本规范。

简　　介：

本规范制定了编写　C语言程序的基本原则、规则和建议。从代码的清晰、简洁、可测试、安全、程序效率、可移植各个方面对　　C语言编程作出了具体指导。

0 规范制订说明

0.1 前言

为提高产品代码质量，指导广大软件开发人员编写出简洁、可维护、可靠、可测试、高效、可移植的代码，编程规范修订工作组分析、总结了我司的各种典型编码问题，并参考了业界编程规范近年来的成果，重新对我司 1999年版编程规范进行了梳理、优化、刷新，编写了本规范。

本规范将分为完整版和精简版，完整版将包括更多的样例、规范的解释以及参考材料 (what & why) ，而精简版将只包含规则部分 (what) 以便查阅。

在本规范的最后，列出了一些业界比较优秀的编程规范，作为延伸阅读参考材料。

0.2 代码总体原则

1、清晰第一

清晰性是易于维护、易于重构的程序必需具备的特征。 代码首先是给人读的，好的代码应当可以像文章一样发声朗诵出来。

目前软件维护期成本占整个生命周期成本的 40%~90% 根据业界经验， 维护期变更代码的成本， 小型系统是开发期的 5倍，大型系统（ 100万行代码以上）可以达到 100 倍。业界的调查指出，开发组平均大约一半的人力用于弥补过去的错误，而不是添加新的功能来帮助公司提高竞争力。

“ 程序必须为阅读它的人而编写，只是顺便用于机器执行。 ”—— Harold Abelson 和 Gerald Jay Sussman

“ 编写程序应该以人为本，计算机第二。 ”—— Steve McConnell

本规范通过后文中的原则 （如头优秀的代码可以自我解释， 不通过注释即可轻易读懂 / 头文件中适合放置接口的声明，不适合放置实现 / 除了常见的通用缩写以外，不使用单词缩写，不得使用汉语拼音）、规则（如防止局部变量与全局变量同名）等说明清晰的重要性。

一般情况下，代码的可阅读性高于性能，只有确定性能是瓶颈时，才应该主动优化。

2、简洁为美

简洁就是易于理解并且易于实现。 代码越长越难以看懂，也就越容易在修改时引入错误。写的代码越多，意味着出错的地方越多，也就意味着代码的可靠性越低。因此，我们提倡大家通过编写简洁明了的代码来提升代码可靠性。

废弃的代码 (没有被调用的函数和全局变量) 要及时清除，重复代码应该尽可能提炼成函数。

本规范通过后文中的原则（如文件应当职责单一 / 一个函数仅完成一件功能）、规则（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 50行）等说明简洁的重要性。

3、选择合适的风格，与代码原有风格保持一致

产品所有人共同分享同一种风格所带来的好处，远远超出为了统一而付出的代价。在公司已有编码规范的指导下， 审慎地编排代码以使代码尽可能清晰， 是一项非常重要的技能。 如果重构 / 修改其他风格的代码时，比较明智的做法是根据现有代码的现有风格继续编写代码 ，或者使用格式转换工具进行转

换成公司内部风格。

0.3 规范实施、解释

本规范制定了编写 C语言程序的基本原则、规则和建议。

本规范适用于公司内使用 C语言编码的所有软件。本规范自发布之日起生效，对以后新编写的和修改的代码应遵守本规范。

本规范由质量体系发布和维护。实施中遇到问题，可以到论坛 http://hi3ms.huawei.com/group/1735/threads.html 上讨论。

在某些情况下（如 BSP 软件）需要违反本文档给出的规则时，相关团队必须通过一个正式的流程来评审、决策规则违反的部分，个体程序员不得违反本规范中的相关规则。

0.4 术语定义

原则 ：编程时必须坚持的指导思想。

规则 ：编程时强制必须遵守的约定。

建议 ：编程时必须加以考虑的约定。

说明 ：对此原则 / 规则 / 建议进行必要的解释。

示例 ：对此原则 / 规则 / 建议从正、反两个方面给出例子。

延伸阅读材料 ：建议进一步阅读的参考材料。

1 头文件

背景

对于 C语言来说，头文件的设计体现了大部分的系统设计。不合理的头文件布局是编译时间过长的根因，不合理的头文件实际上不合理的设计。

术语定义：

依赖 ：本章节特指编译依赖。若 x.h 包含了 y.h ，则称作 x 依赖 y。依赖关系会进行传导，如 x.h 包含 y.h ，而y.h 又包含了 z.h ，则 x 通过 y依赖了 z。依赖将导致编译时间的上升。虽然依赖是不可避免的，也是必须的，但是不良的设计会导致整个系统的依赖关系无比复杂，使得任意一个文件的修改都要重新编译整个系统，导致编译时间巨幅上升。

在一个设计良好的系统中，修改一个文件，只需要重新编译数个，甚至是一个文件。

某产品曾经做过一个实验，把所有函数的实现通过工具注释掉，其编译时间只减少了不到 10%，究其原因，在于 A包含 B， B包含 C， C包含 D，最终几乎每一个源文件都包含了项目组所有的头文件，从而导致绝大部分编译时间都花在解析头文件上。

某产品更有一个“ 优秀实践 ”，用于将 .c 文件通过工具合并成一个比较大的 .c 文件，从而大幅度提高编译效率。其根本原因还是在于通过合并 .c 文件减少了头文件解析次数。但是，这样的“ 优秀实践 ”是对合理划分 .c 文件的一种破坏。

大部分产品修改一处代码，都得需要编译整个工程，对于 TDD之类的实践，要求对于模块级别的编译时间控制在秒级，即使使用分布式编译也难以实现，最终仍然需要合理的划分头文件、以及头文件之间的包含关系，从根本上降低编译时间。

《google C++ Style Guide 》1.2 头文件依赖 章节也给出了类似的阐述：

若包含了头文件 aa.h，则就引入了新的依赖：一旦 aa.h 被修改，任何直接和间接包含 aa.h 代码都会被重新编译。如果 aa.h 又包含了其他头文件如 bb.h，那么 bb.h 的任何改变都将导致所有包含了 aa.h 的代码被重新编译，在敏捷开发方式下，代码会被频繁构建，漫长的编译时间将极大的阻碍频繁构建。因此，我们倾向于减少包含头文件，尤其是在头文件中包含头文件，以控制改动代码后的编译时间。

合理的头文件划分体现了系统设计的思想，但是从编程规范的角度看，仍然有一些通用的方法，用来合理规划头文件。本章节介绍的一些方法，对于合理规划头文件会有一定的帮助。

原则 1.1 头文件中适合放置接口的声明，不适合放置实现。

说明：头文件是模块（ Module）或单元（ Unit ）的对外接口。头文件中应放置对外部的声明，如对外提供的函数声明、宏定义、类型定义等。

内部使用的函数（相当于类的私有方法）声明不应放在头文件中。

内部使用的宏、枚举、结构定义不应放入头文件中。

变量定义不应放在头文件中，应放在 .c 文件中。

变量的声明尽量不要放在头文件中，亦即尽量不要使用全局变量作为接口。变量是模块或单元的内部实现细节，不应通过在头文件中声明的方式直接暴露给外部，应通过函数接口的方式进行对外暴露。即使必须使用全局变量，也只应当在 .c 中定义全局变量，在 .h 中仅声明变量为全局的。

延伸阅读材料：《 C语言接口与实现》（ David R. Hanson 著 傅蓉 周鹏 张昆琪 权威 译 机械工业出版社 2004 年1月）（英文版： "C Interfaces and Implementations" ）

原则 1.2 头文件应当职责单一。

说明： 头文件过于复杂， 依赖过于复杂是导致编译时间过长的主要原因。很多现有代码中头文件过大，职责过多，再加上循环依赖的问题，可能导致为了在 .c 中使用一个宏，而包含十几个头文件。

示例：如下是某平台定义 WORK 类型的头文件：

```
#include <VXWORKS.H>
#include <KERNELLIB.H>
#include <SEMLIB.H>
#include <INTLIB.H>
#include <TASKLIB.H>
#include <MSGQLIB.H>
#include <STDARG.H>
#include <FIOLIB.H>
#include <STDIO.H>
#include <STDLIB.H>
#include <CTYPE.H>
#include <STRING.H>
#include <ERRNOLIB.H>
#include <TIMERS.H>
#include <MEMLIB.H>
#include <TIME.H>
#include <WDLIB.H>
#include <SYSLIB.H>
#include <TASKHOOKLIB.H>
#include <REBOOTLIB.H>
…
```

```
typedef    unsigned    short    WORD;
…
```

这个头文件不但定义了基本数据类型 WORD，还包含了 stdio.h syslib.h 等等不常用的头文件。如果工程中有 10000个源文件，而其中 100个源文件使用了 stdio.h 的printf ，由于上述头文件的职责过于庞大，而WORD又是每一个文件必须包含的，从而导致 stdio.h/syslib.h 等可能被不必要的展开了 9900次，大大增加了工程的编译时间。

原则 1.3 头文件应向稳定的方向包含。

说明：头文件的包含关系是一种依赖，一般来说，应当让不稳定的模块依赖稳定的模块，从而当不稳定的模块发生变化时，不会影响（编译）稳定的模块。

就我们的产品来说，依赖的方向应该是： 产品依赖于平台，平台依赖于标准库。 某产品线平台的代码中已经包含了产品的头文件，导致平台无法单独编译、发布和测试，是一个非常糟糕的反例。

除了不稳定的模块依赖于稳定的模块外，更好的方式是两个模块共同依赖于接口 ，这样任何一个模块的内部实现更改都不需要重新编译另外一个模块。在这里，我们假设接口本身是最稳定的。

延伸阅读材料：编者推荐开发人员使用 “依赖倒置 ”原则，即由使用者制定接口，服务提供者实现接口，更具体的描述可以参见《敏捷软件开发：原则、模式与实践》（ Robert C.Martin 著 邓辉 译 清华大学出版社 2003 年9月） 的第二部分 “敏捷设计 ”章节。

规则 1.1 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。

说明：如果一个 .c 文件不需要对外公布任何接口，则其就不应当存在，除非它是程序的入口，如 main 函数所在的文件。

现有某些产品中，习惯一个 .c 文件对应两个头文件，一个用于存放对外公开的接口，一个用于存放内部需要用到的定义、声明等，以控制 .c 文件的代码行数。编者不提倡这种风格。这种风格的根源在于源文件过大，应首先考虑拆分 .c 文件，使之不至于太大。另外，一旦把私有定义、声明放到独立的头文件中，就无法从技术上避免别人 include 之，难以保证这些定义最后真的只是私有的。

本规则反过来并不一定成立。有些特别简单的头文件， 如命令 ID定义头文件， 不需要有对应的 .c 存在。

示例：对于如下场景，如在一个 .c 中存在函数调用关系：

```
void   foo()
{
   bar();
}

void   bar()
{
   Do something;
}
```

必须在 foo 之前声明 bar ，否则会导致编译错误。

这一类的函数声明，应当在 .c 的头部声明，并声明为 static 的，如下：

```
static    void   bar();

void   foo()
{
   bar();
```

```
}

void   bar()
{
    Do something;
}
```

规则 1.2　禁止头文件循环依赖。

说明：头文件循环依赖，指　a.h 包含 b.h ，b.h 包含 c.h ，c.h 包含 a.h 之类导致任何一个头文件修改，都导致所有包含了　a.h/b.h/c.h　的代码全部重新编译一遍。而如果是单向依赖，如　a.h 包含 b.h ，b.h 包含 c.h ，而 c.h 不包含任何头文件，则修改　a.h 不会导致包含了　b.h/c.h　的源代码重新编译。

规则 1.3 .c/.h　文件禁止包含用不到的头文件。

说明：很多系统中头文件包含关系复杂，开发人员为了省事起见，可能不会去一一钻研，直接包含一切想到的头文件，　甚至有些产品干脆发布了一个　god.h ，其中包含了所有头文件，　然后发布给各个项目组使用，这种只图一时省事的做法，导致整个系统的编译时间进一步恶化，并对后来人的维护造成了巨大的麻烦。

规则 1.4　头文件应当自包含。

说明：简单的说，自包含就是任意一个头文件均可独立编译。如果一个文件包含某个头文件，还要包含另外一个头文件才能工作的话，就会增加交流障碍，给这个头文件的用户增添不必要的负担。

示例：

如果 a.h 不是自包含的，需要包含　b.h 才能编译，会带来的危害：

每个使用 a.h 头文件的 .c 文件，为了让引入的　a.h 的内容编译通过，都要包含额外的头文件　b.h 。

额外的头文件　b.h 必须在 a.h 之前进行包含，这在包含顺序上产生了依赖。

注意：该规则需要与“　.c/.h　文件禁止包含用不到的头文件”规则一起使用，不能为了让　a.h 自包含，而在 a.h 中包含不必要的头文件。　a.h 要刚刚可以自包含，　不能在 a.h 中多包含任何满足自包含之外的其他头文件。

规则 1.5　总是编写内部　#include　保护符（ #define　保护）。

说明：多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取阻止头文件内容被包含多于一次的机制。

通常的手段是为每个文件配置一个宏，当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。

所有头文件都应当使用　#define　防止头文件被多重包含，命名格式为　FILENAME_H 为了保证唯一性，更好的命名是　PROJECTNAME_PATH_FILENAME_H

注：没有在宏最前面加上　“_"，即使用 FILENAME_H 代替 _FILENAME_H，是因为一般以　"_" 和 ”__" 开头的标识符为系统保留或者标准库使用，　在有些静态检查工具中，　若全局可见的标识符以　"_" 开头会给出告警。

定义包含保护符时，应该遵守如下规则：

1）保护符使用唯一名称；

2）不要在受保护部分的前后放置代码或者注释。

示例：假定 VOS工程的 timer 模块的 timer.h ，其目录为 VOS/include/timer/timer.h, 应按如下方式保护：

```
#ifndef VOS_INCLUDE_TIMER_TIMER_H
#define  VOS_INCLUDE_TIMER_TIMER_H
...
#endif
```

也可以使用如下简单方式保护 ：

```
#ifndef  TIMER_H
#define  TIMER_H
 ..
#endif
```

例外情况：头文件的版权声明部分以及头文件的整体注释部分（如阐述此头文件的开发背景、使用注意事项等）可以放在保护符 (#ifndef XX_H) 前面。

规则 1.6 禁止在头文件中定义变量。

说明：在头文件中定义变量，将会由于头文件被其他 .c 文件包含而导致变量重复定义。

规则 1.7 只能通过包含头文件的方式使用其他 .c 提供的接口，禁止在 .c 中通过 extern 的方式使用外部函数接口、变量。

说明：若 a.c 使用了 b.c 定义的 foo() 函数，则应当在 b.h 中声明 extern int foo(int input) ；并在 a.c 中通过 #include <b.h> 来使用 foo 。禁止通过在 a.c 中直接写 extern int foo(int input); 来使用 foo ，后面这种写法容易在 foo 改变时可能导致声明和定义不一致。

规则 1.8 禁止在 extern "C" 中包含头文件。

说明：在 extern "C" 中包含头文件，会导致 extern "C" 嵌套， Visual Studio 对 extern "C" 嵌套层次有限制，嵌套层次太多会编译错误。

在 extern "C" 中包含头文件，可能会导致被包含头文件的原有意图遭到破坏。例如，存在 a.h 和 b.h 两个头文件：

a.h

```
#ifndef  A_H__
#define  A_H__

#ifdef  __cplusplus
void  foo(  int  );
#define  a(value) foo(value)
#else
void  a(  int  )
#endif

#endif  /* A_H__ */
```

b.h

```
#ifndef  B_H__
#define  B_H__

#ifdef  __cplusplus
extern  "C"  {
#endif

#include  "a.h"
      void  b();

#ifdef  __cplusplus
}
#endif

#endif  /* B_H__ */
```

使用 C++预处理器展开 b.h ， 将会得到

```
extern    "C"  {
        void   foo(   int   );
        void   b();
}
```

按照 a.h 作者的本意，函数 foo 是一个 C++自由函数，其链接规范为 "C++" 。但在 b.h 中，由于 #include "a.h" 被放到了 extern "C" { } 的内部，函数 foo 的链接规范被不正确地更改了。

示例：错误的使用方式：

```
extern    “C”
{
#include    “xxx.h   ”
...
}
```

正确的使用方式：

```
#include    “xxx.h   ”
extern    “C”
{
...
}
```

建议 1.1 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。为方便外部使用者，建议每一个模块提供一个 .h ，文件名为目录名。

说明：需要注意的是，这个 .h 并不是简单的包含所有内部的 .h ，它是为了模块使用者的方便，对外整体提供的模块接口。

以Google test （简称 GTest）为例， GTest作为一个整体对外提供 C++单元测试框架， 其1.5 版本的 gtest 工程下有 6个源文件和 12个头文件。但是它对外只提供一个 gtest.h ，只要包含 gtest.h 即可使用 GTest 提供的所有对外提供的功能，使用者不必关系 GTest内部各个文件的关系，即使以后 GTest 的内部实现改变了，比如把一个源文件 c拆成两个源文件，使用者也不必关心，甚至如果对外功能不变，连重新编译都不需要。

对于有些模块， 其内部功能相对松散， 可能并不一定需要提供这个 .h ，而是直接提供各个子模块或者 .c 的头文件。

比如产品普遍使用的 VOS，作为一个大模块，其内部有很多子模块，他们之间的关系相对比较松散，就不适合提供一个 vos.h 。而 VOS的子模块，如 Memory（仅作举例说明，与实际情况可能有所出入），其内部实现高度内聚， 虽然其内部实现可能有多个 .c 和 .h ，但是对外只需要提供一个 Memory.h 声明接口。

建议 1.2 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。

说明：降低接口使用者的编写难度。

建议 1.3 头文件不要使用非习惯用法的扩展名，如 .inc 。

说明： 目前很多产品中使用了 .inc 作为头文件扩展名， 这不符合 c语言的习惯用法。 在使用 .inc 作为头文件扩展名的产品，习惯上用于标识此头文件为私有头文件。但是从产品的实际代码来看，这一条并没有被遵守，一个 .inc 文件被多个 .c 包含比比皆是。本规范不提倡将私有定义单独放在头文件中，具

体见 规则 1.1 。

除此之外，使用 .inc 还导致 source insight 、Visual stduio 等 IDE 工具无法识别其为头文件，导致很多功能不可用，如“跳转到变量定义处”。虽然可以通过配置，强迫 IDE识别 .inc 为头文件，但是有些软件无法配置，如 Visual Assist 只能识别 .h 而无法通过配置识别 .inc 。

建议 1.4 同一产品统一包含头文件排列方式。

说明：常见的包含头文件排列方式：功能块排序、文件名升序、稳定度排序。

示例 1：

以升序方式排列头文件可以避免头文件被重复包含，如：

```
#include <a.h>
#include <b.h>
#include <c/d.h>
#include <c/e.h>
#include <f.h>
```

示例 2：

以稳定度排序，建议将不稳定的头文件放在前面，如把产品的头文件放在平台的头文件前面，如下：

```
#include <product.h>
#include <platform.h>
```

相对来说， product.h 修改的较为频繁，如果有错误，不必编译 platform.h 就可以发现 product.h 的错误，可以部分减少编译时间。

2 函数

背景

函数设计的精髓：编写整洁函数，同时把代码有效组织起来。

整洁函数要求：代码简单直接、不隐藏设计者的意图、用干净利落的抽象和直截了当的控制语句将函数有机组织起来。

代码的有效组织包括：逻辑层组织和物理层组织两个方面。逻辑层，主要是把不同功能的函数通过某种联系组织起来，主要关注模块间的接口，也就是模块的架构。物理层，无论使用什么样的目录或者名字空间等，需要把函数用一种标准的方法组织起来。例如：设计良好的目录结构、函数名字、文件组织等，这样可以方便查找。

原则 2.1 一个函数仅完成一件功能。

说明：一个函数实现多个功能给开发、使用、维护都带来很大的困难。

将没有关联或者关联很弱的语句放到同一函数中，会导致函数职责不明确，难以理解，难以测试和改动。

案例：realloc 。在标准 C语言中， realloc 是一个典型的不良设计。 这个函数基本功能是重新分配内存，但它承担了太多的其他任务： 如果传入的指针参数为 NULL就分配内存， 如果传入的大小参数为 0就释放内存，如果可行则就地重新分配，如果不行则移到其他地方分配。如果没有足够可用的内存用来完成重新分配（扩大原来的内存块或者分配新的内存块），则返回 NULL，而原来的内存块保持不变。这个函数不易扩展，容易导致问题。例如下面代码容易导致内存泄漏：

```
char *buffer = ( char *)malloc(XXX_SIZE);
.....
```

buffer = (char *)realloc(buffer, NEW_SIZE);

如果没有足够可用的内存用来完成重新分配，函数返回为 NULL，导致 buffer 原来指向的内存被丢失。

延伸阅读材料：《敏捷软件开发：原则、模式与实践》 第八章，单一职责原则 (SRP)

原则 2.2 重复代码应该尽可能提炼成函数。

说明：重复代码提炼成函数可以带来维护成本的降低。

重复代码是我司不良代码最典型的特征之一。在“ 代码能用就不改 ”的指导原则之下，大量的烟囱式设计及其实现充斥着各产品代码之中。新需求增加带来的代码拷贝和修改，随着时间的迁移，产品中堆砌着许多类似或者重复的代码。

项目组应当使用代码重复度检查工具，在持续集成环境中持续检查代码重复度指标变化趋势，并对新增重复代码及时重构。当一段代码重复两次时，即应考虑消除重复，当代码重复超过三次时，应当立刻着手消除重复。

一般情况下，可以通过提炼函数的形式消除重复代码。

示例：

```
UC ccb_aoc_process( )
{
   ... ...
      struct    AOC_E1_E7 aoc_e1_e7;
   aoc_e1_e7.aoc  = 0;
   aoc_e1_e7.e[0] = 0;
   ... ... //aoc_e1_e7.e[i]                从到赋值，下同
aoc_e1_e7.e[6] = 0;
aoc_e1_e7.tariff_rate = 0;
... ...

   if   (xxx)
{
        if   (xxx)
   {
            aoc_e1_e7.e[0] = 0;
        ... ...
        aoc_e1_e7.e[6] = 0;
        aoc_e1_e7.tariff_rate = 0;
      }
   ... ...
}
   else   if   (xxx)
{
        if   (xxx)
   {
            aoc_e1_e7.e[0] = 0;
        ... ...
        aoc_e1_e7.e[6] = 0;
```

```
        aoc_e1_e7.tariff_rate = 0;
        }
ccb_caller_e1 = aoc_e1_e7.e[0];
... ...
ccb_caller_e7 = aoc_e1_e7.e[6];
ccb_caller_tariff_rate = aoc_e1_e7.tariff_rate;
... ...
}
... ...

    if   (xxx)
{
          if   (xxx)
   {
                 if   (xxx)
         {
                    aoc_e1_e7.e[0] = 0;
              ... ...
              aoc_e1_e7.e[6] = 0;
              aoc_e1_e7.tariff_rate = 0;
                }
       ... ...
}
        else    if   (xxx)
{
                if   (xxx)
   {
        aoc_e1_e7.e[0] = 0;
       ... ...
       aoc_e1_e7.e[6] = 0;
       aoc_e1_e7.tariff_rate = 0;
             }
   ccb_caller_e1 = aoc_e1_e7.e[0];
   ... ...
   ccb_caller_e7 = aoc_e1_e7.e[6];
          ccb_caller_tariff_rate = aoc_e1_e7.tariff_rate;
          ... ...
             }
             return    1;
        }
        else
        {
             return    0;
```

```
        }
    }
```

刺鼻的代码坏味充斥着这个函数。 红色字体的部分是简单的代码重复， 粗体字部分是代码结构的重复，将重复部分提炼成一个函数即可消除重复。

规则 2.1 避免函数过长，新增函数不超过 50行（非空非注释行）。

说明：本规则仅对新增函数做要求，对已有函数修改时，建议不增加代码行。

过长的函数往往意味着函数功能不单一，过于复杂（参见原则 2.1 ：一个函数只完成一个功能）。

函数的有效代码行数，即 NBNC(非空非注释行）应当在 [1 ， 50] 区间。

例外：某些实现算法的函数，由于算法的聚合性与功能的全面性，可能会超过 50行。

延伸阅读材料： 业界普遍认为一个函数的代码行不要超过一个屏幕，避免来回翻页影响阅读；一般的代码度量工具建议都对此进行检查，例如 Logiscope 的函数度量： "Number of Statement" （函数中的可执行语句数）建议不超过 20行， QAC建议一个函数中的所有行数（包括注释和空白行）不超过 50行。

规则 2.2 避免函数的代码块嵌套过深，新增函数的代码块嵌套不超过 4层。

说明：本规则仅对新增函数做要求，对已有的代码建议不增加嵌套层次。

函数的代码块嵌套深度指的是函数中的代码控制块（例如： if 、 for 、 while 、 switch 等）之间互相包含的深度。每级嵌套都会增加阅读代码时的脑力消耗，因为需要在脑子里维护一个“ 栈 ”（比如，进入条件语句、进入循环 ,, ）。应该做进一步的功能分解，从而避免使代码的阅读者一次记住太多的上下文。优秀代码参考值： [1, 4] 。

示例：如下代码嵌套深度为 5。

```
void  serial (      void  )
{
      if   (!Received)
   {
      TmoCount = 0;
            switch   (Buff)
      {
            case  AISGFLG:
                  if   ((TiBuff.Count > 3)
            && ((TiBuff.Buff[0] == 0xff) || (TiBuf.Buff[0] == CurPa.ADDR)))
         {
            Flg7E =                false  ;
            Received =                true  ;
         }
                  else
         {
            TiBuff.Count = 0;
            Flg7D =                false  ;
            Flg7E =                true  ;
         }
                  break ;
            default   :
                  break ;
      }
   }
}
```

规则 2.3 可重入函数应避免使用共享变量；若需要使用，则应通过互斥手段（关中断、信号量）对其加以保护。

说明：可重入函数是指可能被多个任务并发调用的函数。在多任务操作系统中，函数具有可重入性是多个任务可以共用此函数的必要条件。共享变量指的全局变量和 static 变量。

编写 C语言的可重入函数时，不应使用 static 局部变量， 否则必须经过特殊处理， 才能使函数具有可重入性。

示例：函数 square_exam 返回 g_exam平方值。那么如下函数不具有可重入性。

```
int  g_exam;
unsigned  int  example(  int  para )
{
      unsigned  int  temp;

   g_exam = para;        //  （** ）
   temp = square_exam ( );

      return  temp;
}
```

此函数若被多个线程调用的话，其结果可能是未知的，因为当（ **）语句刚执行完后，另外一个使用本函数的线程可能正好被激活，那么当新激活的线程执行到此函数时，将使 g_exam赋于另一个不同的 para 值，所以当控制重新回到“ temp = square_exam ( )”后，计算出的 temp 很可能不是预想中的结果。此函数应如下改进。

```
int  g_exam;
unsigned  int  example(  int  para )
{
      unsigned  int  temp;

   [  申请信号量操作  ]      //  若申请不到 “信号量 ”，说明另外的进程正处于
   g_exam = para;            //  给g_exam赋值并计算其平方过程中（即正在使用此
   temp = square_exam( );          //  信号），本进程必须等待其释放信号后，才可继
   [  释放信号量操作  ]      //  续执行。其它线程必须等待本线程释放信号量后
      //  才能再使用本信号。
      return  temp;
}
```

规则 2.4 对参数的合法性检查，由调用者负责还是由接口函数负责，应在项目组 / 模块内应统一规定。缺省由调用者负责。

说明：对于模块间接口函数的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查， 结果就遗漏了合法性检查这一必要的处理过程， 造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查， 这种情况虽不会造成问题， 但产生了冗余代码，降低了效率。

示例：下面红色部分的代码在每一个函数中都写了一次，导致代码有较多的冗余。如果函数的参数比较多，而且判断的条件比较复杂（比如：一个整形数字需要判断范围等），那么冗余的代码会大面积充斥着业务代码。

```
void  PidMsgProc(MsgBlock *Msg)
{
   MsgProcItem *func = NULL;
```

```
if (Msg == NULL)
{
    return;
}
... ...
GetMsgProcFun(Msg, &func);

func(Msg);
        return   ;
}

int   GetMsgProcFun(MsgBlock *Msg, MsgProcItem **func)
{
    if (Msg == NULL)
   {
       return 1;
   }

   ... ...
   *func = VOS_NULL_PTR;

       for   (Index = 0; Index < NELEM(g_MsgProcTable); Index++)
   {
            if   ((g_MsgProcTable[Index].FlowType == Msg->FlowType)
          && (g_MsgProcTable[Index].Status == Msg->Status)
          && (g_MsgProcTable[Index].MsgType == Msg->MsgType))
       {
          *func = &(g_MsgProcTable[Index]);
                return    0;
       }
   }

       return    1;
}

int   ServiceProcess(      int    CbNo, MsgBlock *Msg)
{
   if ( Msg == NULL)
   {
       return 1;
   }

   ... ...
       //    业务处理代码
   ... ...

       return    0;
}
```

规则 2.5　对函数的错误返回码要全面处理。

说明：一个函数（标准库中的函数 / 第三方库函数 / 用户定义的函数）能够提供一些指示错误发生的方法。这可以通过使用错误标记、特殊的返回数据或者其他手段，不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

示例：下面的代码导致宕机

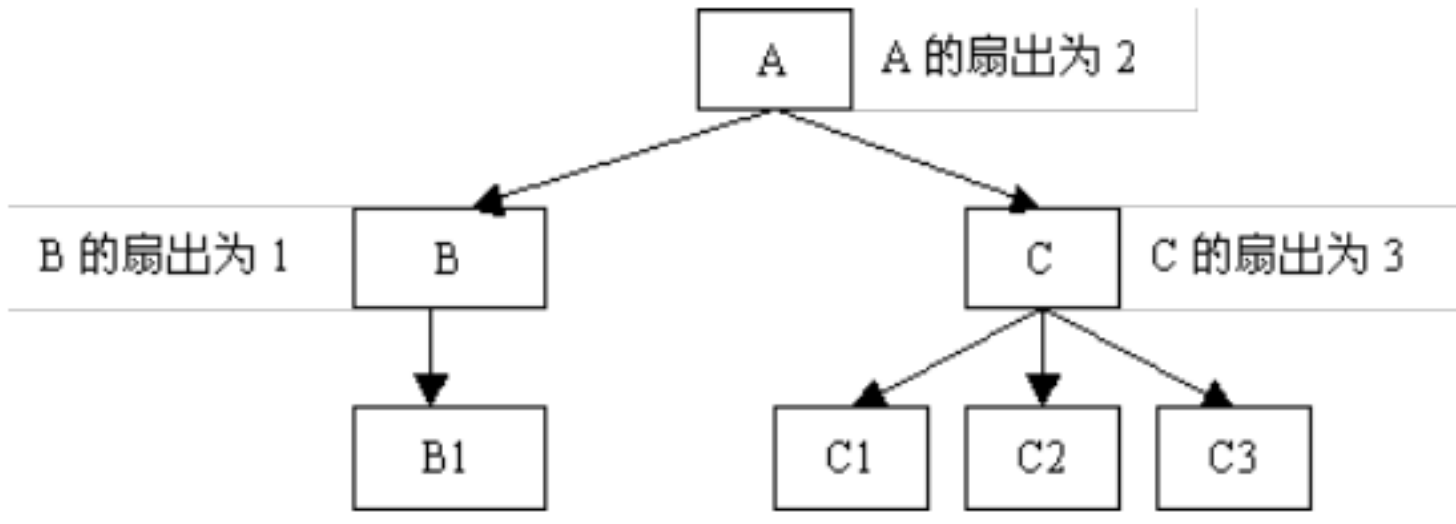
```
    FILE *fp = fopen( "./writeAlarmLastTime.log","r");
  if    (fp == NULL)
  {
        return  ;
    }
    char  buff[128] =  "" ;
    fscanf(fp,  "%s", buff); /*  读取最新的告警时间；由于文件 writeAlarmLastTime.log 为空，导致buff 为空 */
    fclose(fp);
    long  fileTime = getAlarmTime(buff);  /*  解析获取最新的告警时间； getAlarmTime 函数未检查buff 指针，导致宕机 */
```

正确写法：

```
    FILE *fp = fopen( "./writeAlarmLastTime.log","r");
  if    (fp == NULL)
  {
        return  ;
    }
    char  buff[128] =  "" ;
    if  (fscanf(fp,  "%s",buff) == EOF)  //  检查函数 fscanf 的返回值，确保读到数据
    {
        fclose(fp);
        return  ;
    }
    fclose(fp);
    long  fileTime = getAlarmTime(buff);  //  解析获取最新的告警时间；
```

规则 2.6 设计高扇入，合理扇出（小于 7）的函数。

说明：扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。



扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，例如：总是 1，表明函数的调用层次可能过多，这样不利于程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力。通常函数比较合理的扇出（调度函数除外）通常是 3~5。

扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。

扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

延伸阅读材料：扇入（ Fan-in ）和扇出（ Fan-out ）是 Henry 和 Kafura 在 1981 年引入，用来说明模块间的耦合（ coupling ），后面人们扩展到函数 / 方法、模块 / 类、包等。

The Fan-in (Informational fan-in) metric measures the fan-in of a module. The fan-in of a module A is the number of modules that pass control into module A.

The Fan-out metric measures the number of the number of modules that are called by a given module.

规则 2.7 废弃代码（没有被调用的函数和变量 ）要及时清除。

说明：程序中的废弃代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

建议 2.1 函数不变参数使用 const 。

说明：不变的值更易于理解 / 跟踪和分析，把 const 作为默认选项，在编译时会对其进行检查，使代码更牢固 / 更安全。

示例： C99 标准 7.21.4.4 中 strncmp 的例子，不变参数声明为 const 。

```
int strncmp( const char *s1, const char *s2, register size_t n)
{
    register unsigned char u1, u2;

    while (n-- > 0)
  {
    u1 = ( unsigned char ) *s1++;
    u2 = ( unsigned char ) *s2++;

        if (u1 != u2)
    {
            return u1 - u2;
    }

        if (u1 == '\0' )
        {
            return 0;
        }
      }

    return 0;
}
```

延伸阅读： pc-lint 8.0 的帮助材料（ pc-lint.pdf ） 11.4 const Checking

建议 2.2　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。

说明：带有内部“存储器”的函数的功能可能是不可预测的，　　因为它的输出可能取决于内部存储器　　（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在　　C语言中，函数的　static　局部变量是函数的内部存储器，有可能使函数的功能不可预测，然而，当某函数的返回值为指针类型时，则必须是 static　的局部变量的地址作为返回值，若为　　auto 类，则返回为错针。

示例：如下函数，其返回值（即功能）是不可预测的。

```
unsigned　int　integer_sum(　　unsigned　int　base )
{
    unsigned　　int　index;
   static unsigned int sum = 0;                //　注意，是 static　类型的。
                                               //　若改为 auto 类型，则函数即变为可预测。
    for　(index = 1; index <= base; index++)
   {
    sum += index;
   }

    return　　sum;
}
```

延伸阅读材料：　erlang　语言中关于　dirty　的概念，函数式语言的优势

建议 2.3　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。函数在使用输入参数之前，应进行有效性检查。

示例：下面的代码导致宕机

```
hr = root_node->get_first_child(&log_item);                    // list.xml　　为空，导致读出　log_item　为空
…..
hr = log_item->get_next_sibling(&media_next_node);                  // log_item　　为空，导致宕机
```

正确写法：确保读出的内容非空。

```
hr = root_node->get_first_child(&log_item);
…..
if　(log_item == NULL)　　//　确保读出的内容非空
{
    return　　retValue;
}
hr = log_item->get_next_sibling(&media_next_node);
```

建议 2.4　函数的参数个数不超过　　5个。

说明：函数的参数过多，会使得该函数易于受外部（其他部分的代码）变化的影响，从而影响维护工作。函数的参数过多同时也会增大测试的工作量。

函数的参数个数不要超过　　5个，如果超过了建议拆分为不同函数。

建议 2.5　除打印类函数外，不要使用可变长参函数。

说明：可变长参函数的处理过程比较复杂容易引入错误，而且性能也比较低，使用过多的可变长参函

数将导致函数的维护难度大大增加。

建议 2.6　在源文件范围内声明和定义的所有函数，除非外部可见，否则应该增加　　static　关键字。
说明：如果一个函数只是在同一文件中的其他地方调用，那么就用　　static　声明。使用 static　确保只是在声明它的文件中是可见的，并且避免了和其他文件或库中的相同标识符发生混淆的可能性。
建议定义一个　STATIC宏，在调试阶段，将　STATIC定义为 static　，版本发布时，改为空，以便于后续的打热补丁等操作。

```
#ifdef　_DEBUG
#define STATIC static
#else
#define　STATIC
#endif
```

3　标识符命名与定义

3.1　通用命名规则

目前比较使用的如下几种命名风格：
unix like　风格：单词用小写字母，每个单词直接用下划线　　,_?分割，例如　text_mutex　，kernel_text_address　。
Windows风格：大小写字母混用，单词连在一起，每个单词首字母大写。不过　　Windows风格如果遇到大写专有用语时会有些别扭，　例如命名一个读取　RFC文本的函数，　命令为 ReadRFCText，看起来就没有　unix like　的read_rfc_text　清晰了。
匈牙利命名法是　计算机程序设计中的一种命名规则，用这种方法命名的变量显示了其数据类型。　　匈牙利命名主要包括三个部分：基本类型、一个或更多的前缀、一个限定词。　　这种命令法最初在　20世纪 80 年代的微软公司广泛使用，　并在 win32API 和 MFC库中广泛的使用，　但 匈牙利命名法存在较多的争议，　例如：.NET Framework , 微软新的软件开发平台，　除了接口类型一般不适用匈牙利命名法。　.NET Framework 指导方针建议程序员不要用匈牙利命名法，但是没有指明不要用系统匈牙利命名法还是匈牙利应用命名法，或者是两者都不要用。与此对比，　Java 的标准库中连接口类型也不加前缀。　(来源 http://zh.wikipedia.org/wiki/%E5%8C%88%E7%89%99%E5%88%A9%E5%91%BD%E5%90%8D%E6%B3%95)
匈牙利命名法更多的信息见　http://en.wikipedia.org/wiki/Hungarian_notation　。
标识符的命名规则历来是一个敏感话题，　典型的命名风格如　unix 风格、 windows 风格等等，　从来无法达成共识。实际上，各种风格都有其优势也有其劣势，而且往往和个人的审美观有关。我们对标识符定义主要是为了让团队的代码看起来尽可能统一，有利于代码的后续阅读和修改，产品可以根据自己的实际需要指定命名风格，规范中不再做统一的规定。

原则 3.1 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：尽可能给出描述性名称，不要节约空间，让别人很快理解你的代码更重要。
示例：好的命名：

```
int    error_number;
int    number_of_completed_connection;
```

不好的命名：使用模糊的缩写或随意的字符：

```
int    n;
int    nerr;
int    n_comp_conns;
```

原则 3.2　除了常见的通用缩写以外，不使用单词缩写，不得使用汉语拼音。

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，常用单词的缩写必须统一。协议中的单词的缩写与协议保持一致。对于某个系统使用的专用缩写应该在注视或者某处做统一说明。

示例：一些常见可以缩写的例子：

argument　可缩写为　arg
buffer　可缩写为　buff
clock　可缩写为　clk
command 可缩写为　cmd
compare　可缩写为　cmp
configuration　可缩写为　cfg
device　可缩写为　dev
error　可缩写为　err
hexadecimal　可缩写为　hex
increment　可缩写为　inc 、
initialize　可缩写为　init
maximum 可缩写为　max
message 可缩写为　msg
minimum 可缩写为　min
parameter　可缩写为　para
previous　可缩写为　prev
register　可缩写为　reg
semaphore 可缩写为　sem
statistic　可缩写为　stat
synchronize　可缩写为　sync
temp 可缩写为　tmp

规则 3.1　产品 / 项目组内部应保持统一的命名风格。

说明： Unix like 和 windows like 风格均有其拥趸，产品应根据自己的部署平台，选择其中一种，并在产品内部保持一致。

例外：即使产品之前使用匈牙利命名法，新代码也不应当使用。

建议 3.1　用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

示例：

add/remove　　begin/end　　create/destroy

insert/delete	first/last	get/release
increment/decrement	put/get	add/delete
lock/unlock	open/close	min/max
old/new	start/stop	next/previous
source/target	show/hide	send/receive
source/destination	copy/paste	up/down

建议 3.2　尽量避免名字中出现数字编号，除非逻辑上的确需要编号。

示例：如下命名，使人产生疑惑。

```
#define    EXAMPLE_0_TEST_
#define    EXAMPLE_1_TEST_
```

应改为有意义的单词命名

```
#define    EXAMPLE_UNIT_TEST_
#define    EXAMPLE_ASSERT_TEST_
```

建议 3.3　标识符前不应添加模块、项目、产品、部门的名称作为前缀。

说明：很多已有代码中已经习惯在文件名中增加模块名，这种写法类似匈牙利命名法，导致文件名不可读，并且带来带来如下问题：

第一眼看到的是模块名，而不是真正的文件功能，阻碍阅读；

文件名太长；

文件名和模块绑定，　不利于维护和移植。　若 foo.c 进行重构后，　从 a模块挪到 b模块，若 foo.c 中有模块名，则需要将文件名从　a_module_foo.c 改为 b_module_foo.c

建议 3.4　平台 / 驱动等适配代码的标识符命名风格保持和平台　/ 驱动一致。

说明：涉及到外购芯片以及配套的驱动，这部分的代码变动（包括为产品做适配的新增代码），应该保持原有的风格。

建议 3.5　重构 / 修改部分代码时，应保持和原有代码的命名风格一致。

说明：根据源代码现有的风格继续编写代码，有利于保持总体一致。

3.2　文件命名规则

建议 3.6　文件命名统一采用小写字符。

说明：因为不同系统对文件名大小写处理会不同（如　MS的 DOS、Windows系统不区分大小写，但是　Linux 系统则区分），所以代码文件命名建议统一采用全小写字母命名。

3.3　变量命名规则

规则 3.2　全局变量应增加　“g_”前缀。

规则 3.3　静态变量应增加　“s_”前缀。

说明：增加 g_前缀或者 s_前缀，原因如下：
首先，全局变量十分危险，通过前缀使得全局变量更加醒目，促使开发人员对这些变量的使用更加小心。
其次，从根本上说，应当尽量不使用全局变量，增加 g_和 s_前缀，会使得全局变量的名字显得很丑陋，从而促使开发人员尽量少使用全局变量。

规则 3.4 禁止使用单字节命名变量，但允许定义 i 、 j 、 k作为局部循环变量。

建议 3.7 不建议使用匈牙利命名法。
说明：变量命名需要说明的是变量的含义，而不是变量的类型。在变量命名前增加类型说明，反而降低了变量的可读性；更麻烦的问题是，如果修改了变量的类型定义，那么所有使用该变量的地方都需要修改。
匈牙利命名法源于微软，然而却被很多人以讹传讹的使用。而现在即使是微软也不再推荐使用匈牙利命名法。历来对匈牙利命名法的一大诟病，就是导致了变量名难以阅读，这和本规范的指导思想也有冲突，所以本规范特意强调，变量命名不应采用匈牙利命名法，而应想法使变量名为一个有意义的词或词组，方便代码的阅读。

建议 3.8 使用名词或者形容词＋名词方式命名变量。

3.4 函数命名规则

建议 3.9 函数命名应以函数要执行的动作命名，一般采用动词或者动词＋名词的结构。
示例：找到当前进程的当前目录

```
DWORD GetCurrentDirectory( DWORD BufferLength, LPTSTR Buffer );
```

建议 3.10 函数指针除了前缀，其他按照函数的命名规则命名。

3.5 宏的命名规则

规则 3.5 对于数值或者字符串等等常量的定义，建议采用全大写字母，单词之间加下划线 **,_?**的方式命名（枚举同样建议使用此方式定义）。
示例：

```
#define  PI_ROUNDED 3.14
```

规则 3.6 除了头文件或编译开关等特殊标识定义，宏定义不能使用下划线 **,_?**开头和结尾。
说明：一般来说， ?_?开头、结尾的宏都是一些内部的定义， ISO/IEC 9899 （俗称 C99）中有如下的描述（ 6.10.8 Predefined macro names ）：
None of these macro names （这里上面是一些内部定义的宏的描述） , nor the identifier defined, shall be the subject of a #define or a #undef preprocessing directive.
Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

延伸阅读材料：《代码大全第　　2版》（ Steve　McConnell　著　金戈 / 汤凌 / 陈硕 / 张菲　译　电子工业出版社　2006年3月）＂第11章变量命的力量　＂。

4 变量

原则 4.1　一个变量只有一个功能，不能把一个变量用作多种用途。

说明：一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

示例：具有两种功能的反例

```
WORD DelRelTimeQue(void )
{
        WORD Locate;

        Locate = 3;
        Locate = DeleteFromQue(Locate); /* Locate          具有两种功能：位置和函数      DeleteFromQue 的返回值  */
        return    Locate;
}
```

正确做法：使用两个变量

```
WORD  DelRelTimeQue( void )
{
        WORD Ret;
        WORD Locate;

        Locate    =  3;
        Ret       =  DeleteFromQue(Locate);

        return    Ret;
}
```

原则 4.2　结构功能单一；不要设计面面俱到的数据结构。

说明：相关的一组信息才是构成一个结构体的基础，结构的定义应该可以明确的描述一个对象，而不是一组相关性不强的数据的集合。

设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

示例：如下结构不太清晰、合理。

```
typedef     struct    STUDENT_STRU
{
        unsigned    char  name[32];      /* student's name */
        unsigned    char  age;           /* student's age */
        unsigned    char  sex;           /* student's sex, as follows */
```

```
        /* 0 - FEMALE; 1 - MALE */
        unsigned    char  teacher_name[32];       /* the student teacher's name */
        unsigned    char  teacher_sex;            /* his teacher sex */
} STUDENT;
```

若改为如下，会更合理些。

```
typedef     struct     TEACHER_STRU
{
        unsigned    char  name[32];      /* teacher name */
        unsigned    char  sex;           /* teacher sex, as follows */
        /* 0 - FEMALE; 1 - MALE */
        unsigned    int    teacher_ind;       /* teacher index */
} TEACHER;

typedef     struct     STUDENT_STRU
{
        unsigned    char  name[32];            /* student's name */
        unsigned    char  age;                 /* student's age */
        unsigned    char  sex;                 /* student's sex, as follows */
        /* 0 - FEMALE; 1 - MALE */
        unsigned    int     teacher_ind;       /* his teacher index */
} STUDENT;
```

原则 4.3　不用或者少用全局变量。

说明：单个文件内部可以使用　　static 的全局变量，可以将其理解为类的私有成员变量。

全局变量应该是模块的私有数据，不能作用对外的接口使用，使用　　static 类型定义，可以有效防止外部文件的非正常访问，建议定义一个　　STATIC 宏，在调试阶段，将　STATIC 定义为 static ，版本发布时，改为空，以便于后续的打补丁等操作。

```
#ifdef     _DEBUG
#define STATIC static
#else
#define    STATIC
#endif
```

直接使用其他模块的私有数据，将使模块间的关系逐渐走向“ 剪不断理还乱 ”的耦合状态，这种情形是不允许的。

规则 4.1　防止局部变量与全局变量同名。

说明：尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

规则 4.2　通讯过程中使用的结构，必须注意字节序。

说明：通讯报文中，字节序是一个重要的问题，我司设备使用的　　cpu 类型复杂多样，大小端、　　32 位/64 位的处理器也都有，如果结构会在报文交互过程中使用，必须考虑字节序问题。

由于位域在不同字节序下，表现看起来差别更大，所以更需要注意。
对于这种跨平台的交互，数据成员发送前，都应该进行主机序到网络序的转换；接收时，也必须进行网络序到主机序的转换。

规则 4.3　严禁使用未经初始化的变量作为右值。
说明：坚持建议　4.3 （在首次使用前初始化变量，初始化的地方离使用的地方越近越好。）可以有效避免未初始化错误。

建议 4.1　构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的全局变量，防止多个不同模块或函数都可以修改、创建同一全局变量的现象。
说明：降低全局变量耦合度。

建议 4.2 使用面向接口编程思想，通过　API 访问数据：如果本模块的数据需要对外部模块开放，应提供接口函数来设置、获取，同时注意全局数据的访问互斥。
说明：避免直接暴露内部数据给外部模型使用，是防止模块间耦合最简单有效的方法。
定义的接口应该有比较明确的意义，比如一个风扇管理功能模块，　有自动和手动工作模式，　那么设置、查询工作模块就可以定义接口为　SetFanWorkMode　，GetFanWorkMode　；查询转速就可以定义为 GetFanSpeed　；风扇支持节能功能开关，可以定义　EnabletFanSavePower　等等。

建议 4.3　在首次使用前初始化变量，初始化的地方离使用的地方越近越好。
说明：未初始化变量是　C和 C++程序中错误的常见来源。在变量首次使用前确保正确初始化。
在较好的方案中，变量的定义和初始化要做到亲密无间。
示例：

```
// 不可取的初始化：无意义的初始化
int   speedup_factor      = 0;
if   (condition)
{
   speedup_factor = 2;
}
else
{
   speedup_factor = -1;
}

// 不可取的初始化：初始化和声明分离
int   speedup_factor;
if   (condition)
{
   speedup_factor = 2;
}
else
{
   speedup_factor = -1;
}

// 较好的初始化：使用默认有意义的初始化
```

```
int    speedup_factor = -1;
if    (condition)
{
    speedup_factor = 2;
}

//  较好的初始化使用     ?: 减少数据流和控制流的混合
int    speedup_factor = condition?2:-1;

//  较好的初始化：使用函数代替复杂的计算流
    int    speedup_factor = ComputeSpeedupFactor()            ;
```

建议 4.4　明确全局变量的初始化顺序，避免跨模块的初始化依赖。

说明：系统启动阶段，使用全局变量前，要考虑到该全局变量在什么时候初始化，使用全局变量和初始化全局变量，两者之间的时序关系，谁先谁后，一定要分析清楚，不然后果往往是低级而又灾难性的。

建议 4.5　尽量减少没有必要的数据类型默认转换与强制转换。

说明：当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

```
    char   ch;
    unsigned    short    int   exam;

    ch = -1;
    exam = ch;     //     编译器不产生告警，此时       exam为0xFFFF。
```

5 宏、常量

规则 5.1　用宏定义表达式时，要使用完备的括号。

说明：因为宏只是简单的代码替换，不会像函数一样先将参数计算后，再传递。

示例：如下定义的宏都存在一定的风险

```
    #define    RECTANGLE_AREA(a, b) a * b
    #define    RECTANGLE_AREA(a, b) (a * b)
    #define    RECTANGLE_AREA(a, b) (a) * (b)
```

正确的定义应为：

```
    #define    RECTANGLE_AREA(a, b) ((a) * (b))
```

这是因为：

如果定义 #define RECTANGLE_AREA(a, b) a * b　　或#define RECTANGLE_AREA(a, b) (a * b)

则 c/RECTANGLE_AREA(a,b) 将扩展成 c/a * b，c 与b 本应该是除法运算，结果变成了乘法运算，造成错误。

如果定义 #define RECTANGLE_AREA(a, b) (a) * (b)

则 RECTANGLE_AREA(c + d, e + f)　将扩展成：　(c + d * e + f), d　　与 e 先运算，造成错误。

规则 5.2　将宏所定义的多条表达式放在大括号中。

说明：更好的方法是多条语句写成　　do while(0) 的方式。

示例：看下面的语句，只有宏的第一条表达式被执行。

```
#define    FOO(x) \
        printf(     "arg is %d\n"       , x); \
        do_something_useful(x);
```

为了说明问题，下面　for 语句的书写稍不符规范

```
for    (blah = 1; blah < 10; blah++)
        FOO(blah)
```

用大括号定义的方式可以解决上面的问题：

```
#define    FOO(x) { \
        printf(     "arg is %s\n"       , x); \
        do_something_useful(x); \
}
```

但是如果有人这样调用：

```
if    (condition == 1)
        FOO(10);
else
        FOO(20);
```

那么这个宏还是不能正常使用，所以必须这样定义才能避免各种问题：

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

用 do-while(0)　方式定义宏，完全不用担心使用者如何使用宏，也不用给使用者加什么约束。

规则 5.3　使用宏时，不允许参数发生变化。

示例：如下用法可能导致错误。

```
#define    SQUARE(a) ((a) * (a))
int    a = 5;
int    b;
b = SQUARE(a++);    //    结果： a = 7 ，即执行了两次增。
```

正确的用法是：

```
b = SQUARE(a);
a++;    //    结果： a = 6 ，即只执行了一次增。
```

同时也建议即使函数调用，也不要在参数中做变量变化操作，因为可能引用的接口函数，在某个版本升级后，变成了一个兼容老版本所做的一个宏，结果可能不可预知。

规则 5.4　不允许直接使用魔鬼数字。

说明：使用魔鬼数字的弊端：代码难以理解；如果一个有含义的数字多处使用，一旦需要修改这个数值，代价惨重。

使用明确的物理状态或物理意义的名称能增加信息，并能提供单一的维护点。

解决途径：

对于局部使用的唯一含义的魔鬼数字，可以在代码周围增加说明注释，也可以定义局部 const 变量，变量命名自注释。

对于广泛使用的数字，必须定义 const 全局变量 / 宏；同样变量 / 宏命名应是自注释的。

0作为一个特殊的数字，作为一般默认值使用没有歧义时，不用特别定义。

建议 5.1　除非必要，应尽可能使用函数代替宏。

说明：宏对比函数，有一些明显的缺点：

宏缺乏类型检查，不如函数调用检查严格。

宏展开可能会产生意想不到的副作用，如 #define SQUARE(a) (a) * (a) 这样的定义，如果是 SQUARE(i++)，就会导致 i 被加两次；如果是函数调用 double square(double a) {return a * a;} 则不会有此副作用。

以宏形式写的代码难以调试难以打断点，不利于定位问题。

宏如果调用的很多，会造成代码空间的浪费，不如函数空间效率高。

示例：下面的代码无法得到想要的结果：

```
#define    MAX_MACRO(a, b)   ((a) > (b) ? (a) : (b))
int    MAX_FUNC(int    a,    int    b) {
        return    ((a) > (b) ? (a) : (b));
}
int    testFunc()
{
        unsigned    int    a = 1;
        int    b = -1;
    printf(        "MACRO: max of a and b is: %d\n"        , MAX_MACRO(++a, b));
    printf(        "FUNC : max of a and b is: %d\n"        , MAX_FUNC(a, b));
        return    0;
}
```

上面宏代码调用中，结果是 (a < b) ，所以 a 只加了一次，所以最终的输出结果是：

MACRO: max of a and b is: -1

FUNC : max of a and b is: 2

建议 5.2　常量建议使用 const 定义代替宏。

说明：‘尽量用编译器而不用预处理’，因为 #define 经常被认为好象不是语言本身的一部分。看下面的语句：

```
#define    ASPECT_RATIO 1.653
```

编译器会永远也看不到 ASPECT_RATIO 这个符号名，因为在源码进入编译器之前，它会被预处理程序去掉，于是 ASPECT_RATIO 不会加入到符号列表中。如果涉及到这个常量的代码在编译时报错，就会很令人费解，因为报错信息指的是 1.653，而不是 ASPECT_RATIO 。如果 ASPECT_RATIO 不是在你自己写的头文件中定义的，你就会奇怪 1.653 是从哪里来的，甚至会花时间跟踪下去。这个问题也会出现在符号调试器中，因为同样地，你所写的符号名不会出现在符号列表中。

解决这个问题的方案很简单：不用预处理宏，定义一个常量：

```
const double ASPECT_RATIO = 1.653;
```

这种方法很有效，但有两个特殊情况要注意。首先，定义指针常量时会有点不同。因为常量定义一般是放在头文件中（许多源文件会包含它），除了指针所指的类型要定义成 const 外，重要的是指针也经常要定义成 const 。例如，要在头文件中定义一个基于 char* 的字符串常量，你要写两次 const ：

```
const char * const authorName = "Scott Meyers" ;
```

延伸阅读材料：关于 const 和指针的使用，这里摘录两段 ISO/IEC 9899 （俗称 C99 ）的描述：

The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value".

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of any object pointed to by ptr_to_constant shall not be modified through that pointer,but ptr_to_constant itself may be changed to point to another object. Similarly, the contents of the intpointed to by constant_ptrmay be modified, but constant_ptritself shall always point to the same location.

The declaration of the constant pointer constant_ptr may be clarified by including a definition for the type "pointer to int".

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares constant_ptras an object that has type "const-qualified pointer to int".

建议 5.3 宏定义中尽量不使用 return 、goto 、continue 、break 等改变程序流程的语句。

说明：如果在宏定义中使用这些改变流程的语句，很容易引起资源泄漏问题，使用者很难自己察觉。

示例：在某头文件中定义宏 CHECK_AND_RETURN：

```
#define CHECK_AND_RETURN(cond, ret) { if (cond == NULL_PTR) { return ret;}}
```

然后在某函数中使用 (只说明问题，代码并不完整):

```
pMem1 = VOS_MemAlloc(...);
CHECK_AND_RETURN(pMem1 , ERR_CODE_XXX)
pMem2 = VOS_MemAlloc(...);
CHECK_AND_RETURN(pMem2, ERR_CODE_XXX) /* 此时如果 pMem2==NULL_PTR，则 pMem1未释放函数就返回了，造成内存泄漏。 */
```

所以说，类似于 CHECK_AND_RETURN 这些宏，虽然能使代码简洁，但是隐患很大，使用须谨慎。

6 质量保证

原则 6.1　代码质量保证优先原则

（1）正确性，指程序要实现设计要求的功能。

（2）简洁性，指程序易于理解并且易于实现。

（3）可维护性，指程序被修改的能力，包括纠错、改进、新需求或功能规格变化的适应能力。

（4）可靠性，指程序在给定时间间隔和环境条件下，按设计要求成功运行程序的概率。

（5）代码可测试性，指软件发现故障并隔离、定位故障的能力，以及在一定的时间和成本前提下，进行测试设计、测试执行的能力。

（6）代码性能高效，指是尽可能少地占用系统资源，包括内存和执行时间。

（7）可移植性，指为了在原来设计的特定环境之外运行，对系统进行修改的能力。

（8）个人表达方式　/ 个人方便性，指个人编程习惯。

原则 6.2　要时刻注意易混淆的操作符。

说明：包括易混淆和的易用错操作符

1、易混淆的操作符

C语言中有些操作符很容易混淆，编码时要非常小心。

赋值操作符　“= ”　　逻辑操作符　“== ”

关系操作符　“< ”　　位操作符　"<<"

关系操作符　“> ”　　位操作符　“>> ”

逻辑操作符　“|| ”　　位操作符　"|"

逻辑操作符　“&& ”　位操作符　"&"

逻辑操作符　"!"　　位操作符　“~ ”

2、易用错的操作符

(1)　除操作符　"/"

当除操作符　“/ ”的运算量是整型量时，运算结果也是整型。

如:1/2=0

(2) 求余操作符　"%"

求余操作符　"%"的运算量只能是整型。

如： 5%2=1，而 5.0%2是错误的。

(3)　自加、自减操作符　“++ ”、“-- ”

示例 1

```
k = 5;
x = k++;
```

执行后，　x = 5 ，　k = 6

示例 2

```
k = 5;
x = ++k;
```

执行后，　x = 6 ，　k = 6

示例 3

k = 5;

x = k--;

执行后， x = 5 ， k = 4

示例 4

k = 5;

x = --k;

执行后， x = 4 ， k = 4

原则 6.3 必须了解编译系统的内存分配方式，特别是编译系统对不同类型的变量的内存分配规则，如局部变量在何处分配、静态变量在何处分配等。

原则 6.4 不仅关注接口，同样要关注实现。

说明：这个原则看似和“面向接口”编程思想相悖，但是实现往往会影响接口，函数所能实现的功能，除了和调用者传递的参数相关，往往还受制于其他隐含约束，如：物理内存的限制，网络状况，具体看“抽象漏洞原则”。

延伸阅读材料：

http://local.joelonsoftware.com/mediawiki/index.php?title=Chinese_%28Simplified%29&oldid=9699

规则 6.1 禁止内存操作越界。

说明：内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

示例：使用 itoa （）将整型数转换为字符串时：

char TempShold[10] ;

itoa(ProcFrecy,TempShold, 10); /* 数据库刷新间隔设为值 1073741823 时，系统监控后台 coredump，监控前台抛异常。 */

TempShold是以‘ \0 ’结尾的字符数组，只能存储 9个字符，而 ProcFrecy 的最大值可达到 10位，导致符数组 TempShold越界。

正确写法：一个 int （32 位）在－ 2147483647 ～2147483648 之间，将数组 TempShold设置成 12 位。

char TempShold[12] ;

itoa(ProcFrecy,TempShold,10);

坚持下列措施可以避免内存越界：

数组的大小要考虑最大情况，避免数组分配空间不够。

避免使用危险函数 sprintf /vsprintf/strcpy/strcat/gets 操作字符串，使用相对安全的函数 snprintf/strncpy/strncat/fgets 代替。

使用 memcpy/memset 时一定要确保长度不要越界

字符串考虑最后的’ \0 ’， 确保所有字符串是以’ \0 ’结束

指针加减操作时，考虑指针类型长度

数组下标进行检查

使用时 sizeof 或者 strlen 计算结构 / 字符串长度，避免手工计算

延伸阅读材料： 《公司常见软件编程低级错误：内存越界 .ppt 》

规则 6.2　禁止内存泄漏。

说明： 内存和资源　（包括定时器　/ 文件句柄　/Socket/　队列 / 信号量　/GUI 等各种资源）　泄漏是常见的错误。

示例：异常出口处没有释放内存

```
MsgDBDEV = (PDBDevMsg)GetBuff(    sizeof   ( DBDevMsg ), __LINE__);
if   (MsgDBDEV == NULL)
{
        return   ;
}
MsgDBAppToLogic = (LPDBSelfMsg)GetBuff(         sizeof   (DBSelfMsg), __LINE__ );
if   ( MsgDBAppToLogic == NULL )
{
        return   ;  //MsgDB_DEV指向的内存丢失
}
```

坚持下列措施可以避免内存泄漏：

异常出口处检查内存、定时器　　/ 文件句柄　/Socket/　队列 / 信号量　/GUI 等资源是否全部释放

删除结构指针时，必须从底层向上层顺序删除

使用指针数组时，确保在释放数组时，数组中的每个元素指针是否已经提前被释放了

避免重复分配内存

小心使用有　return　、 break 语句的宏，确保前面资源已经释放

检查队列中每个成员是否释放

延伸阅读材料：　　《公司常见软件编程低级错误：内存泄漏　　.ppt 》

规则 6.3　禁止引用已经释放的内存空间。

说明：在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块，而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

示例：一个函数返回的局部自动存储对象的地址，导致引用已经释放的内存空间

```
int   * foobar (      void  )
{
        int   local_auto = 100;
        return    &local_auto;
}
```

坚持下列措施可以避免引用已经释放的内存空间：

内存释放后，把指针置为　　NULL；使用内存指针前进行非空判断。

耦合度较强的模块互相调用时，　　一定要仔细考虑其调用关系，　　防止已经删除的对象被再次使用。

避免操作已发送消息的内存。

自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象（具有更大作用域的对象或者静态对象或者从一个函数返回的对象）

延伸阅读材料：　　《公司常见软件编程低级错误：野指针　　.ppt 》

规则 6.4　编程时，要防止差　1错误。

说明： 此类错误一般是由于把　　“<= ”误写成　“< ”或 “>= ”误写成　“> ”等造成的，　由此引起的后果，　　很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查。

使用变量时要注意其边界值的情况。

示例：如 C语言中字符型变量，有效值范围为 -128 到127。故以下表达式的计算存在一定风险。

```
char  ch = 127;
int   sum = 200;
ch += 1;    // 127  为ch的边界值，再加将使  ch上溢到 -128 ，而不是 128
sum += ch;   //  故sum的结果不是 328，而是 72。
```

规则 6.5 所有的 if ... else if 结构应该由 else 子句结束 ；switch 语句必须有 default 分支。

建议 6.1 函数中分配的内存，在函数退出之前要释放。

说明：有很多函数申请内存，保存在数据结构中，要在申请处加上注释，说明在何处释放。

建议 6.2 if 语句尽量加上 else 分支，对没有 else 分支的语句要小心对待。

建议 6.3 不要滥用 goto 语句。

说明： goto 语句会破坏程序的结构性，所以除非确实需要，最好不使用 goto 语句。

可以利用 goto 语句方面退出多重循环；同一个函数体内部存在大量相同的逻辑但又不方便封装成函数的情况下，譬如反复执行文件操作，对文件操作失败以后的处理部分代码（譬如关闭文件句柄，释放动态申请的内存等等），一般会放在该函数体的最后部分，再需要的地方就 goto 到那里，这样代码反而变得清晰简洁。实际也可以封装成函数或者封装成宏，但是这么做会让代码变得没那么直接明了。

示例：

```
int  foo(  void )
{
    char * p1 = NULL;
    char * p2 = NULL;
    char * p3 = NULL;
    int   result = -1;

  p1 = (    char  *)malloc(0x100);
    if  (p1 == NULL)
  {
        goto  Exit0;
  }
  strcpy(p1,      "this is p1"    );

  p2 = (    char  *)malloc(0x100);
    if  (p2 == NULL)
  {
        goto  Exit0;
  }
  strcpy(p2,      "this is p2"    );

  p3 = (    char  *)malloc(0x100);
    if  (p3 == NULL)
  {
        goto  Exit0;
  }
  strcpy(p3,      "this is p3"    );
```

```
    result = 0;

Exit0:
    free(p1);                   // C  标准规定可以  free  空指针
    free(p2);
    free(p3);

        return   result;
}
```

建议 6.4 时刻注意表达式是否会上溢、下溢。

示例：如下程序将造成变量下溢。

```
    unsigned char size ;
    …
    while   (size-- >= 0)          //    将出现下溢
    {
        ... // program code
    }
```

当 size 等于 0时，再减不会小于 0，而是 0xFF，故程序是一个死循环。应如下修改。

```
    char   size;     //    从 unsigned char      改为 char
    …
    while   (size-- >= 0)
    {
        ... // program code
    }
```

7 程序效率

原则 7.1 在保证软件系统的正确性、简洁、可维护性、可靠性及可测性的前提下，提高代码效率。

本章节后面所有的规则和建议，都应在不影响前述可读性等质量属性的前提下实施。

说明：不能一味地追求代码效率，而对软件的正确、简洁、可维护性、可靠性及可测性造成影响。

产品代码中经常有如下代码：

```
    int   foo()
    {
        if  ( 异常条件 )
      {
            异常处理 ;
            return   ERR_CODE_1;
      }
        if  ( 异常条件）
      {
            异常处理 ;
            return   ERR_CODE_2;
```

```
    }
        正常处理 ;
        return    SUCCESS;
}
```

这样的代码看起来很清晰，而且也避免了大量的 if else 嵌套。但是从性能的角度来看，应该把执行概率较大的分支放在前面处理，由于正常情况下的执行概率更大，若首先考虑性能，应如下书写：

```
int    foo()
{
    if  ( 满足条件 )
    {
        正常处理 ;
        return    SUCCESS;
    }
    else  if  ( 概率比较大的异常条件    )
    {
        异常处理 ;
        return    ERR_CODE_1;
    }
    else
    {
        异常处理 ;
        return    ERR_CODE_2;
    }
}
```

除非证明 foo 函数是性能瓶颈，否则按照本规则， 应优先选用前面一种写法 。

以性能为名，使设计或代码更加复杂，从而导致可读性更差，但是并没有经过验证的性能要求（比如实际的度量数据和目标的比较结果）作为正当理由，本质上对程序没有真正的好处。无法度量的优化行为其实根本不能使程序运行得更快。

记住：让一个正确的程序更快速，比让一个足够快的程序正确，要容易得太多。大多数时候，不要把注意力集中在如何使代码更快上，应首先关注让代码尽可能地清晰易读和更可靠。

原则 7.2 通过对数据结构、程序算法的优化来提高效率。

建议 7.1 将不变条件的计算移到循环体外。

说明：将循环中与循环无关，不是每次循环都要做的操作，移到循环外部执行。

示例一：

```
for  ( int   i = 0; i < 10; i++ )
{
    sum += i;
    back_sum = sum;
}
```

对于此 for 循环来说语句 “back_Sum = sum; ”没必要每次都执行，只需要执行一次即可，因此可以改为：

```
for  ( int   i = 0; i < 10; i++ )
{
    sum += i;
}
```

```
back_sum = sum;
```

示例二：

```
for (    _UL i = 0; i < func_calc_max(); i++)
{
        //process;
}
```

函数 func_calc_max()　没必要每次都执行，只需要执行一次即可，因此可以改为：

```
_UL max = func_calc_max();
for (    _UL i = 0; i < max; i++)
{
        //process;
}
```

建议 7.2　对于多维大数组，避免来回跳跃式访问数组成员。

示例：多维数组在内存中是从最后一维开始逐维展开连续存储的。　　下面这个对二维数组访问是以　　SIZE_B 为步长跳跃访问，到尾部后再从头（第二个成员）开始，依此类推。局部性比较差，当步长较大时，可能造成 cache 不命中，反复从内存加载数据到　　cache。应该把 i 和 j 交换。

```
...
for (int       i = 0; i < SIZE_B; i++)
{
        for (int       j = 0; j < SIZE_A; j++)
        {
            sum += x[j][i];
        }
}
...
```

上面这段代码，在　SIZE_B　数值较大时，效率可能会比下面的代码低　　：

```
...
for (int       i = 0; i < SIZE_B; i++)
{
        for (int       j = 0; j < SIZE_A; j++)
        {
            sum +=      x[ i ][  j ] ;
        }
}
...
```

建议 7.3　创建资源库，以减少分配对象的开销。

说明：例如，使用线程池机制，避免线程频繁创建、销毁的系统调用；使用内存池，对于频繁申请、释放的小块内存，一次性申请一个大块的内存，当系统申请内存时，从内存池获取小块内存，使用完

毕再释放到内存池中，避免内存申请释放的频繁系统调用.

建议 7.4　将多次被调用的　“小函数　”改为 inline　函数或者宏实现。

说明：　如果编译器支持　inline　，可以采用　inline　函数。否则可以采用宏。

在做这种优化的时候一定要注意下面　　inline　函数的优点：其一编译时不用展开，代码　　SIZE小。其二可以加断点，易于定位问题，例如对于引用计数加减的时候。其三函数编译时，编译器会做语法检查。

三思而后行。

8 注释

原则 8.1　优秀的代码可以自我解释，不通过注释即可轻易读懂。

说明：优秀的代码不写注释也可轻易读懂，注释无法把糟糕的代码变好，需要很多注释来解释的代码往往存在坏味道，需要重构。

示例：注释不能消除代码的坏味道：

```
/*　判断 m是否为素数　*/
/*　返回值：　：　是素数，　：　不是素数　*/
int　p(　int　m)
{
        int　k = sqrt(m);
        for　(　int　i = 2; i <= k; i++)
                if　(m % i == 0)
                        break；　/*　发现整除，表示　m不为素数，结束遍历　*/
        /*　遍历中没有发现整除的情况，返回　　*/
        if　(i > k)
                return　1;
        /*　遍历中没有发现整除的情况，返回　　*/
        else
                return　0;
}
```

重构代码后，不需要注释：

```
int　IsPrimeNumber(　int　num)
{
        int　sqrt_of_num = sqrt (num);
        for　(　int　i = 2; i <= sqrt_of_num; i++)
    {
                if　(num % i == 0)
        {
                        return　FALSE;
        }
    }
        return　TRUE;
}
```

原则 8.2　注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：有歧义的注释反而会导致维护者更难看懂代码，正如带两块表反而不知道准确时间。

示例：注释与代码相矛盾，注释内容也不清楚，前后矛盾。

```
/*  上报网管时要求故障  ID 与恢复 ID 相一致 */
/*  因此在此由告警级别获知是不是恢复  ID */
/*  若是恢复 ID 则设置为 ClearId ，否则设置为 AlarmId */
if  (CLEAR_ALARM_LEVEL != RcData.level)
{
   SetAlarmID(RcData.AlarmId);
}
else
{
   SetAlarmID(RcData.ClearId);
}
```

正确做法：修改注释描述如下：

```
/*  网管达成协议：上报故障  ID 与恢复 ID 由告警级别确定，若是清除级别，  ID 设置为 ClearId ，否
则设为 AlarmId 。 */
```

原则 8.3 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，而不是重复描述代码。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

注释不是为了名词解释（ what），而是说明用途（ why）。

示例：如下注释纯属多余。

```
++i;  /* increment i */
if  (receive_flag)  /* if receive_flag is TRUE */
```

如下这种无价值的注释不应出现（空洞的笑话，无关紧要的注释）。

```
/*  时间有限，现在是 :04 ，根本来不及想为什么，也没人能帮我说清楚  */
```

而如下的注释则给出了有用的信息：

```
/*  由于 xx 编号网上问题，在 xx 情况下，芯片可能存在写错误，此芯片进行写操作后，必须进行回读校
验，如果回读不正确，需要再重复写  - 回读操作，最多重复三次，这样可以解决绝大多数网上应用时的
写错误问题 */
int  time = 0;
do
{
   write_reg(some_addr, value);
   time++;
}  while  ((read_reg(some_addr) != value) && (time < 3));
```

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释，出彩的或复杂的代码块前要加注释，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;
```

```
}
```

规则 8.1　修改代码时，维护代码周边的所有注释，以保证注释与代码的一致性。不再有用的注释要删除。
说明：不要将无用的代码留在注释中，随时可以从源代码配置库中找回代码；即使只是想暂时排除代码，也要留个标注，不然可能会忘记处理它。

规则 8.2　文件头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者姓名、工号、内容、功能说明、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。
说明：通常头文件要对功能和用法作简单说明，源文件包含了更多的实现细节或算法讨论。
版权声明格式：　Copyright　？ Huawei Technologies Co., Ltd. 1998-2011. All rights reserved.
1998-2011 根据实际需要可以修改，　1998 是文件首次创建年份，而　2011是最新文件修改年份。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/************************************************
  Copyright　？ Huawei Technologies Co., Ltd. 1998-2011. All rights reserved.
  File name:  //　　文件名
  Author:　　ID ：　Version:　Date: //　　作者、工号、版本及完成日期
  Description:  //　　用于详细说明此程序文件完成的主要功能，与其他模块
          //　　或函数的接口，输出值、取值范围、含义及参数间的控
          //　　制、顺序、独立或依赖等关系
  Others:　//　　其它内容的说明
  History:　//　　修改历史记录列表，每条修改记录应包括修改日期、修改
          //　　者及修改内容简述
    1. Date:
      Author:　　ID:
      Modification:
    2. ...
************************************************/
```

规则 8.3　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。
说明：重要的、复杂的函数，提供外部使用的接口函数应编写详细的注释。

规则 8.4　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。
示例：

```
/* The ErrorCode when SCCP translate */
/* Global Title failure, as follows */                    /*  变量作用、含义  */
/* 0    ― SUCCESS  1  ― GT Table error */
/* 2    ― GT error  Others      ― no use  */          /*  变量取值范围  */
/* only  function  SCCPTranslate() in */
/* this modual can modify it,  and  other */
/* module can visit it through call */
```

```
/* the  function GetGTTransErrorCode() */                    /*    使用方法  */
BYTE g_GTTranErrorCode;
```

规则 8.5　注释应放在其代码上方相邻位置或右方，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，且与下方代码缩进相同。

示例：

```
/* active statistic task number */
#define    MAX_ACT_TASK_NUMBER 1000
#define    MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

可按如下形式说明枚举　　/ 数据 / 联合结构。

```
/* sccp interface with sccp user primitive message name */
enum SCCP_USER_PRIMITIVE
{
    N_UNITDATA_IND,      /* sccp notify sccp user unit data come */
    N_NOTICE_IND,      /* sccp notify user the No.7 network can not transmission this message */
    N_UNITDATA_REQ,    /* sccp user's unit data transmission request*/
};
```

规则 8.6　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_FWD:
    ProcessFwd();
        /* now jump into case CMD_A */
case  CMD_A:
    ProcessA();
        break ;
     //  对于中间无处理的连续　　case，已能较清晰说明意图，不强制注释。
switch    (cmd_flag)
{
case  CMD_A:
case  CMD_B:
   {
      ProcessCMD();
            break ;
   }
      ……
}
```

规则 8.7　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。

规则 8.8　同一产品或项目组统一注释风格。

建议 8.1　避免在一行代码或表达式的中间插入注释。

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

建议 8.2　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，除非能用非常流利准确的英文表达。对于有外籍员工的，由产品确定注释语言。

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。

建议 8.3　文件头、函数头、全局常量变量、类型定义的注释格式采用工具可识别的格式。

说明：采用工具可识别的注释格式，例如　　doxygen 格式，方便工具导出注释形成帮助文档。

以doxygen 格式为例，文件头，函数和全部变量的注释的示例如下：

文件头注释：

```
/**
*   @file               （本文件的文件名    eg ： mib.h ）
*   @brief              （本文件实现的功能的简述）
*   @version 1.1            （版本声明）
*   @author             （作者，    eg：张三）
*   @date               （文件创建日期，      eg ： 2010年12月15日）
*/
```

函数头注释：

```
/**
 *@ Description:      向接收方发送    SET请求
* @param req -        指向整个  SNMP SET 请求报文  .
 * @param ind -       需要处理的   subrequest     索引 .
 * @return     成功 ： SNMP_ERROR_SUCCESS;失败 ：  SNMP_ERROR_COMITFAIL
 */

Int commit_set_request(Request *req, int ind);
```

全局变量注释：

```
/**      模拟的  Agent MIB */
agentpp_simulation_mib * g_agtSimMib;
```

函数头注释建议写到声明处。并非所有函数都必须写注释，建议针对这样的函数写注释：重要的、复杂的函数，提供外部使用的接口函数。

延伸阅读材料：

1、《代码大全第　2版》（ Steve　McConnell　著　金戈 / 汤凌 / 陈硕 / 张菲　译　电子工业出版社　 2006 年3月）"第32章自说明代码　"。

2、《代码整洁之道》（　Robert C.Martin　　著　韩磊　译　人民邮电出版社　2010　年1月）第四章 "注释 "。

3、《敏捷软件开发：原则、模式与实践》（　　Robert C.Martin　　著　邓辉　译　清华大学出版社　2003 年9月）"第5章重构 "。

4、《 Doxygen中文手册》（　http://hi3ms.huawei.com/group/1735/files.html　　）。

9 排版与格式

规则 9.1　程序块采用缩进风格编写，每级缩进为　　4个空格。

说明：当前各种编辑器　　/IDE 都支持 TAB键自动转空格输入，需要打开相关功能并设置相关功能。

编辑器 /IDE 如果有显示　TAB的功能也应该打开，方便及时纠正输入错误。

IDE向导生成的代码可以不用修改。

宏定义、编译开关、条件预处理语句可以顶格（或使用自定义的排版方案，但产品　　/ 模块内必须保持一致）。

规则 9.2　相对独立的程序块之间、变量说明之后必须加空行。

示例：如下例子不符合规范。

```
if    (!valid_ni(ni))
{
        // program code
    ...
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

应如下书写

```
if    (!valid_ni(ni))
{
        // program code
    ...
}

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

规则 9.3　一条语句不能过长，如不能拆分需要分行写。一行到底多少字符换行比较合适，产品可以自行确定。

说明：对于目前大多数的　　PC来说，　132比较合适（　80/132 是VTY常见的行宽值）；对于新　　PC宽屏显示器较多的产品来说，可以设置更大的值。

换行时有如下建议：

　　换行时要增加一级缩进，使代码可读性更好；

　　低优先级操作符处划分新行；换行时操作符应该也放下来，放在新行首；

　　换行时建议一个完整的语句放在一行，不要根据字符数断行

示例：

```
if   ((temp_flag_var == TEST_FLAG)
    &&(((temp_counter_var - TEST_COUNT_BEGIN) % TEST_COUNT_MODULE) >= TEST_COUNT_THRESHOLD))
{
        // process code
}
```

规则 9.4　多个短语句（包括赋值语句）不允许写在同一行内，即一行只写一条语句。

示例：

```
int    a = 5;      int    b= 10;       //  不好的排版
```

较好的排版

```
int    a = 5;
int    b= 10;
```

规则 9.5 if　、for 、do、while 、case、switch 、default　等语句独占一行。

说明：执行语句必须用缩进风格写，属于　　if 、for 、do、while 、case、switch 、default　等下一个缩进级别；

一般写 if 、for 、do、while 等语句都会有成对出现的　　,{} ?，对此有如下建议可以参考：

if 、for 、do、while 等语句后的执行语句建议增加成对的　　,{} ?；

如果 if/else　配套语句中有一个分支有　　,{} ?，那么令一个分支即使一行代码也建议增加　　,{} ?；

添加 ,{?的位置可以在 if 等语句后，　也可以独立占下一行；　独立占下一行时，　可以和 if 在一个缩进级别，也可以在下一个缩进级别；但是如果　　if 语句很长，或者已经有换行，建议　　,{?使用独占一行的写法。

规则 9.6　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。

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。

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++;                         // "++","--"       与内容之间
```

(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)
```

建议 9.1 注释符（包括 **,/* ?,// ?,*/ ?**）与注释内容之间要用一个空格进行分隔。

说明：这样可以使注释的内容部分更清晰。

现在很多工具都可以批量生成、删除 '//' 注释，这样有空格也比较方便统一处理。

建议 9.2 源程序中关系较为紧密的代码应尽可能相邻。

10 表达式

规则 10.1 表达式的值在标准所允许的任何运算次序下都应该是相同的。

说明：除了少数操作符（函数调用操作符 ( ) 、&& || 、?: 和 , （逗号）） 之外，子表达式所依据的运算次序是未指定的并会随时更改。注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

将复合表达式分开写成若干个简单表达式，明确表达式的运算次序，就可以有效消除非预期副作用。

1、自增或自减操作符

示例：

```
x = b[i] + i++;
```

b[i] 的运算是先于还是后于 i ++ 的运算，表达式会产生不同的结果，把自增运算做为单独的语句，可以避免这个问题。

```
x = b[i] + i;
i ++;
```

2 、函数参数

说明：函数参数通常从右到左压栈，但函数参数的计算次序不一定与压栈次序相同。

示例：

```
x = func( i++, i);
```

应该修改代码明确先计算第一个参数：

```
i++;
x = func(i, i);
```

3、函数指针

说明：函数参数和函数自身地址的计算次序未定义。

示例：

```
p->task_start_fn(p++);
```

求函数地址 p与计算 p++无关，结果是任意值。必须单独计算 p++：

```
p->task_start_fn(p);
p++;
```

4 、函数调用

示例：

```
int g_var = 0;
```

```
int    fun1()
{
    g_var += 10;
        return    g_var;
}

int    fun2()
{
    g_var += 100;
        return    g_var;
    }
    int    x = fun1() + fun2();
```

编译器可能先计算 fun1() ，也可能先计算 fun2() ，由于 x的结果依赖于函数 fun1()/fun2() 的计算次序（fun1()/fun2() 被调用时修改和使用了同一个全局变量），则上面的代码存在问题。

应该修改代码明确 fun1/ fun2 的计算次序：

```
int    x = fun1();
x = x + fun2();
```

5、嵌套赋值语句

说明：表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。

示例：

```
x = y = y = z / 3;
x = y = y++;
```

6、 volatile 访问

说明：限定符 volatile 表示可能被其它途径更改的变量，例如硬件自动更新的寄存器。编译器不会优化对 volatile 变量的读取。

示例：下面的写法可能无法实现作者预期的功能：

```
/* volume 变量被定义为 volatile 类型 */
UINT16 x = ( volume << 3 ) | volume;          /* 在计算了其中一个子表达式的时候， volume 的值可能已经被其它程序或硬件改变，导致另外一个子表达式的计算结果非预期，可能无法实现作者预期的功能
*/
```

建议 10.1 函数调用不要作为另一个函数的参数使用，否则对于代码的调试、阅读都不利。

说明：如下代码不合理， 仅用于 说明当函数作为参数时，由于参数压栈次数不是代码可以控制的，可能造成未知的输出：

```
int    g_var;
int    fun1()
{
    g_var += 10;
        return    g_var;
}

int    fun2()
{
```

```
    g_var += 100;
        return    g_var;
}
int    main(  int    argc,     char  *argv[],       char   *envp[])
{
    g_var = 1;
    printf(         "func1: %d, func2: %d\n"          , fun1(), fun2());

    g_var = 1;
    printf(         "func2: %d, func1: %d\n"          , fun2(), fun1());
}
```

上面的代码，使用断点调试起来也比较麻烦，阅读起来也不舒服，所以不要为了节约代码行，而写这种代码。

建议 10.2　赋值语句不要写在　if 等语句中，或者作为函数的参数使用。

说明：因为　if 语句中，会根据条件依次判断，如果前一个条件已经可以判定整个条件，则后续条件语句不会再运行，所以可能导致期望的部分赋值没有得到运行。

示例：

```
int    main(  int    argc,     char  *argv[],       char   *envp[])
{
        int    a = 0;
        int    b;
        if    ((a == 0) || ((b = fun1()) > 10))
    {
        printf(             "a: %d\n"     , a);
    }
    printf(         "b: %d\n"     , b);
}
```

作用函数参数来使用，参数的压栈顺序不同可能导致结果未知。

看如下代码，能否一眼看出输出结果会是什么吗？好理解吗？

```
int    g_var;
int    main(  int    argc,     char  *argv[],       char   *envp[])
{
    g_var = 1;
    printf(         "set 1st: %d, add 2nd: %d\n"            , g_var = 10, g_var++);
    g_var = 1;
    printf(         "add 1st: %d, set 2nd: %d\n"            , g_var++, g_var = 10);
}
```

建议 10.3　用括号明确表达式的操作顺序，避免过分依赖默认优先级。

说明：使用括号强调所使用的操作符，防止因默认的优先级与设计思想不符而导致程序出错；同时使得代码更为清晰可读，然而过多的括号会分散代码使其降低了可读性。下面是如何使用括号的建议。

1. 一元操作符，不需要使用括号

```
x = ~a;                    /*    一元操作符，不需要括号      */
x = -a;            /*    一元操作符，不需要括号      */
```

2. 二元以上操作符，如果涉及多种操作符，则应该使用括号

```
x = a + b + c;                    /*    操作符相同，不需要括号      */
```

```
x = f ( a + b, c )                  /*   操作符相同，不需要括号     */
if   (a && b && c)       /*   操作符相同，不需要括号     */
x = (a * 3) + c + d;              /*   操作符不同，需要括号     */
x = ( a == b ) ? a : ( a                –b );         /*   操作符不同，需要括号     */
```

3 . 即使所有操作符都是相同的，如果涉及类型转换或者量级提升，也应该使用括号控制计算的次序

以下代码将 3个浮点数相加：

```
/* 除了逗号 (,) ，逻辑与 (&&) ，逻辑或 (||) 之外， C标准没有规定同级操作符是从左还是从右开始计算，以上表达式存在种计算次序： f4 = (f1 + f2) + f3 或 f4 = f1 + (f2 + f3) ，浮点数计算过程中可能四舍五入，量级提升，计算次序的不同会导致 f4 的结果不同，以上表达式在不同编译器上的计算结果可能不一样，建议增加括号明确计算顺序 */
f4 =  f1 + f2 + f3;
```

.

建议 10.4 赋值操作符不能使用在产生布尔值的表达式上。

说明： 如果布尔值表达式需要赋值操作， 那么赋值操作必须在操作数之外分别进行。这可以帮助避免 = 和 = = 的混淆，帮助我们静态地检查错误。

示例：

```
x = y;
if   (x != 0)
{
   foo ();
}
```

不能写成：

```
if   (( x = y ) != 0)
{
   foo ();
}
```

或者更坏的

```
if   (x = y)
{
   foo ();
}
```

11 代码编辑、编译

规则 11.1 使用编译器的最高告警级别，理解所有的告警，通过修改代码而不是降低告警级别来消除所有告警。

说明：编译器是你的朋友，如果它发出某个告警，这经常说明你的代码中存在潜在的问题。

规则 11.2 在产品软件（项目组）中，要统一编译开关、静态检查选项以及相应告警清除策略。

说明： 如果必须禁用某个告警，应 尽可能单独局部禁用，并且编写一个清晰的注释，说明为什么屏蔽。

某些语句经编译 / 静态检查产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

规则 11.3 本地构建工具（如 PC-Lint ）的配置应该和持续集成的一致。

说明：两者一致，避免经过本地构建的代码在持续集成上构建失败。

规则 11.4 使用版本控制（配置管理）系统，及时签入通过本地构建的代码，确保签入的代码不会影响构建成功。

说明：及时签入代码降低集成难度。

建议 11.1 要小心地使用编辑器提供的块拷贝功能编程。

12 可测性

原则 12.1 模块划分清晰，接口明确，耦合性小，有明确输入和输出，否则单元测试实施困难。

说明：单元测试实施依赖于：

模块间的接口定义清楚、完整、稳定；

模块功能的有明确的验收条件（包括：预置条件、输入和预期结果）；

模块内部的关键状态和关键数据可以查询，可以修改；

模块原子功能的入口唯一；

模块原子功能的出口唯一；

依赖集中处理：和模块相关的全局变量尽量的少，或者采用某种封装形式。

规则 12.1 在同一项目组或产品组内， 要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明。

说明：本规则是针对项目组或产品组的。代码至始至终只有一份代码，不存在开发版本和测试版本的说法。测试与最终发行的版本是通过编译开关的不同来实现的。并且编译开关要规范统一。统一使用编译开关来实现测试版本与发行版本的区别，一般不允许再定义其它新的编译开关。

规则 12.2 在同一项目组或产品组内，调测打印的日志要有统一的规定。

说明：统一的调测日志记录便于集成测试，具体包括：

统一的日志分类以及日志级别；

通过命令行、网管等方式可以配置和改变日志输出的内容和格式；

在关键分支要记录日志，日志建议不要记录在原子函数中，否则难以定位；

调试日志记录的内容需要包括文件名 / 模块名、代码行号、函数名、被调用函数名、错误码、错误发生的环境等。

规则 12.3 使用断言记录内部假设。

说明：断言是对某种内部模块的假设条件进行检查，如果假设不成立，说明存在编程、设计错误。断言可以对在系统中隐藏很深，用其它手段极难发现的问题进行定位，从而缩短软件问题定位时间，提高系统的可测性。

规则 12.4 不能用断言来检查运行时错误。

说明：断言是用来处理内部编程或设计是否符合假设；不能处理对于可能会发生的且必须处理的情况要写防错程序，而不是断言。如某模块收到其它模块或链路上的消息后， 要对消息的合理性进行检查，此过程为正常的错误检查，不能用断言来实现。

断言的使用是有条件的。 断言只能用于程序内部逻辑的条件判断， 而不能用于对外部输入数据的判断 ，因为在网上实际运行时，是完全有可能出现外部输入非法数据的情况。

建议 12.1 为单元测试和系统故障注入测试准备好方法和通道。

13 安全性

代码的安全漏洞大都是由代码缺陷导致，但不是所有代码缺陷都有安全风险。理解安全漏洞产生的原理和如何进行安全编码是减少软件安全问题最直接有效的办法。

原则 13.1 对用户输入进行检查。

说明：不能假定用户输入都是合法的，因为难以保证不存在恶意用户，即使是合法用户也可能由于误用误操作而产生非法输入。用户输入通常需要经过检验以保证安全，特别是以下场景：

用户输入作为循环条件

用户输入作为数组下标

用户输入作为内存分配的尺寸参数

用户输入作为格式化字符串

用户输入作为业务数据（如作为命令执行参数、拼装 sql 语句、以特定格式持久化）

这些情况下如果不对用户数据做合法性验证，很可能导致 DOS、内存越界、格式化字符串漏洞、命令注入、 SQL注入、缓冲区溢出、数据破坏等问题。

可采取以下措施对用户输入检查：

用户输入作为数值的，做数值范围检查

用户输入是字符串的，检查字符串长度

用户输入作为格式化字符串的，检查关键字 “ %”

用户输入作为业务数据，对关键字进行检查、转义

13.1 字符串操作安全

规则 13.1 确保所有字符串是以 NULL结束。

说明：C语言中 ?0 ?作为字符串的结束符， 即NULL结束符。 标准字符串处理函数 （如 strcpy() 、strlen() ）依赖 NULL结束符来确定字符串的长度。没有正确使用 NULL结束字符串会导致缓冲区溢出和其它未定义的行为。

为了避免缓冲区溢出，常常会用相对安全的限制字符数量的字符串操作函数代替一些危险函数。如：

用 strncpy() 代替 strcpy()

用 strncat() 代替 strcat()

用 snprintf() 代替 sprintf()

用 fgets() 代替 gets()

这些函数会截断超出指定限制的字符串，但是要注意它们并不能保证目标字符串总是以 NULL结尾。如果源字符串的前 n个字符中不存在 NULL字符，目标字符串就不是以 NULL结尾。

示例：

```
char a[16];
```

```
strncpy(a,  "0123456789abcdef"  ,  sizeof  (a));
```

上述代码存在安全风险：在调用 strncpy() 后，字符数组 a中的字符串是没有 NULL结束符的，也没有空间存放 NULL结束符。

正确写法：截断字符串，保证字符串以 NULL结束。

```
char  a[16];
strncpy(a,  "0123456789abcdef"  ,  sizeof  (a) - 1 );
a[ sizeof  (a) - 1] =  '\0'  ;
```

规则 13.2 不要将边界不明确的字符串写到固定长度的数组中。

说明：边界不明确的字符串（如来自 gets() 、getenv() 、scanf() 的字符串），长度可能大于目标数组长度，直接拷贝到固定长度的数组中容易导致缓冲区溢出。

示例：

```
char  buff[256];
char  *editor = getenv(  "EDITOR");
if  (editor != NULL)
{
   strcpy(buff, editor);
}
```

上述代码读取环境变量 "EDITOR"的值，如果成功则拷贝到缓冲区 buff 中。而从环境变量获取到的字符串长度是不确定的，把它们拷贝到固定长度的数组中很可能导致缓冲区溢出。

正确写法：计算字符串的实际长度，使用 malloc 分配指定长度的内存

```
char  *buff;
char  *editor = getenv(  "EDITOR");
if  (editor != NULL)
{
   buff = malloc(strlen(editor) + 1);
      if  (buff != NULL)
   {
      strcpy(buff, editor);
   }
}
```

13.2 整数安全

C99标准定义了整型提升（ integer promotions ）、整型转换级别（ integer conversion rank ）以及普通算术转换（ usual arithmetic conversions ）的整型操作。不过这些操作实际上也带来了安全风险。

规则 13.3 避免整数溢出。

说明：当一个整数被增加超过其最大值时会发生整数上溢，被减小小于其最小值时会发生整数下溢。

带符号和无符号的数都有可能发生溢出。

示例 1：有符号和无符号整数的上溢和下溢

```
int  i;
unsigned  int  j;

i = INT_MAX;   // 2,147,483,647
i++;
```

```
printf(     "i = %d\n"     , i);         // i=-2,147,483,648

j = UINT_MAX;     // 4,294,967,295;
j++;
printf(     "j = %u\n"     , j);         // j = 0

i = INT_MIN;       // -2,147,483,648;
i--;
printf(     "i = %d\n"     , i);         // i = 2,147,483,647

   j = 0;
   j--;
   printf(     "j = %u\n"     , j);         // j = 4,294,967,295
```

示例 2：整数下溢导致报文长度异常

```
   /*    报文长度减去    FSM 头的长度   */
   unsigned    int    length;

   length -= FSM_HDRLEN ;
```

处理过短报文时，length 的长度可能小于 FSM_HDRLEN，减法的结果小于。由于 length 是无符号数，结果返回了一个很大的数。

正确写法：增加长度检查

```
   if    (length < FSM_HDRLEN )
   {
         return    VOS_ERROR;
   }
   length -= FSM_HDRLEN ;
```

规则 13.4 避免符号错误。

说明：有时从带符号整型转换到无符号整型会发生符号错误，符号错误并不丢失数据，但数据失去了原来的含义。

带符号整型转换到无符号整型，最高位（ high-order bit ）会丧失其作为符号位的功能。如果该带符号整数的值非负，那么转换后值不变；如果该带符号整数的值为负，那么转换后的结果通常是一个非常大的正数。

示例：符号错误绕过长度检查

```
   #define    BUF_SIZE 10
   int    main(  int    argc,    char * argv[])
   {
        int          length;
        char         buf[BUF_SIZE];

        if (    argc != 3)
      {
            return    -1;
      }
      length = atoi(argv[1]);              //   如果 atoi  返回的长度为负数
```

```
    if (    length < BUF_SIZE)        //    len 为负数，长度检查无效
  {
    memcpy(buf, argv[2],              length);      /*    带符号的 len 被转换为 size_t 类型的无符号整数，负值被解释为一个极大的正整数。        memcpy()调用时引发 buf 缓冲区溢出    */
    printf(           "Data copied\n"      );
  }
    else
  {
    printf(           "Too many data\n"      );
  }
```

正确写法 1：将 len 声明为无符号整型

```
#define    BUF_SIZE 10

int    main(  int    argc,     char * argv[])
{
        unsigned    int    length;
        char  buf[BUF_SIZE];

        if    (argc != 3)
    {
            return    -1;
    }

    length = atoi(argv[1]);

        if    (length < BUF_SIZE)
    {
        memcpy(buf, argv[2], length);
        printf(           "Data copied\n"      );
    }
        else
    {
        printf(           "Too much data\n"      );
        }
        return    0;
}
```

正确写法 2：增加对 len 的更有效的范围校验

```
#define    BUF_SIZE 10

int    main(  int    argc,     char * argv[])
{
        int    length;
        char  buf[BUF_SIZE];

        if    (argc != 3)
    {
            return    -1;
    }
```

```
length = atoi(argv[1]);

    if    ((length > 0)  && (length < BUF_SIZE))
{
   memcpy(buf, argv[2], length);
   printf(            "Data copied\n"      );
}
    else
{
   printf(            "Too much data\n"      );
    }
        return    0;
  }
```

规则 13.5 ：避免截断错误。

说明：将一个较大整型转换为较小整型，并且该数的原值超出较小类型的表示范围，就会发生截断错误，原值的低位被保留而高位被丢弃。截断错误会引起数据丢失。

使用截断后的变量进行内存操作，很可能会引发问题。

示例：

```
  int    main(  int    argc,     char * argv[])
  {
        unsigned    short    total = strlen(argv[1]) + strlen(argv[2]) + 1;
        char * buffer = (       char *)malloc(total);

      strcpy(buffer, argv[1]);
      strcat(buffer, argv[2]);
      free(buffer);

        return    0;
  }
```

示例代码中 total 被定义为 unsigned short ，相对于 strlen() 的返回值类型 size_t （通常为 unsigned long ）太小。如果攻击者提供的两个入参长度分别为 65500和36， unsigned long 的65500+36+1会被取模截断， total 的最终值是 （65500+36+1）%65536= 1。malloc() 只为 buff 分配了 1字节空间， 为strcpy() 和strcat() 的调用创造了缓冲区溢出的条件。

正确写法：将涉及到计算的变量声明为统一的类型， 并检查计算结果。

```
  int    main(  int    argc,     char * argv[])
  {
      size_t total = strlen(argv[1]) + strlen(argv[2]) + 1;
        if    ((total <= strlen(argv[1])) || (total <= strlen(argv[2])))
      {
            /* handle error */
            return    -1;
        }

      char * buffer = (       char *)malloc(total);
    strcpy(buffer, argv[1]);
      strcat(buffer, argv[2]);
      free(buffer);
```

```
        return    0;
}
```

13.3 格式化输出安全

规则 13.6 ：确保格式字符和参数匹配。

说明：使用格式化字符串应该小心，确保格式字符和参数之间的匹配，保留数量和数据类型。格式字符和参数之间的不匹配会导致未定义的行为。大多数情况下，不正确的格式化字符串会导致程序异常终止。

示例：

```
char   *error_msg =       "Resource not available to user."                ;
int    error_type = 3;
/*    格式字符和参数的类型不匹配       */
printf(      "Error (type %s): %d\n"           , error_type, error_msg);

/*    格式字符和参数的数量不匹配       */
printf(      "Error: %s\n"       );
```

格式化字符串在编码时会大量使用，容易 copy-paste 省事，这就容易出现不匹配的错误。

规则 13.7 避免将用户输入作为格式化字符串的一部分或者全部。

说明： 调用格式化 I/O 函数时， 不要直接或者间接将用户输入作为格式化字符串的一部分或者全部。 攻击者对一个格式化字符串拥有部分或完全控制，存在以下风险：进程崩溃、查看栈的内容、改写内存、甚至执行任意代码。

示例 1：

```
char   input[1000];
if    (fgets(input,          sizeof   (input) - 1, stdin) == NULL)
{
        /* handle error */
}
input[    sizeof   (input)-1] =         '\0'    ;
printf(input);
```

上述代码 input 直接来自用户输入，并作为格式化字符串直接传递给 printf() 。当用户输入的是“ %s%s%s%s%s%s%s%s%s%s”，就可能触发无效指针或未映射的地址读取。格式字符 %s 显示栈上相应参数所指定的地址的内存。这里 input 被当成格式化字符串， 而没有提供参数， 因此 printf() 读取栈中任意内存位置，指导格式字符耗尽或者遇到一个无效指针或未映射地址为止。

正确做法：给 printf() 传两个参数，第一个参数为 ”%s”，目的是将格式化字符串确定下来；第二个参数为用户输入 input 。

```
char   input[1000];
if    (fgets(input,          sizeof   (input)-1, stdin) == NULL)
{
        /* handle error */
}
input[    sizeof   (input)-1] =         '\0'    ;
printf(      “%s”, input);
```

示例 2：

```
void   check_password(    char   *user,      char   *password)
{
        if    (strcmp(password(user), password) != 0)
    {
            char   *msg = malloc(strlen(user) + 100);
            if    (!msg)
        {
                /* handle error condition */
        }
        sprintf(msg,              "%s login incorrect"          , user);
        fprintf(STDERR, msg);
        syslog(LOG_INFO, msg);
        free(msg);
    }
     /* …*/
}
```

上述代码检查给定用户名及其口令是否匹配，当不匹配时显示一条错误信息，并将错误信息写入日志中。同样的，如果 user 为 ”%s%s%s%s%s%s%s%s%s%s%s“，经过格式化函数 sprintf() 的拼装后， msg 指向的字符串为 ”%s%s%s%s%s%s%s%s%s%s%s%s login incorrect “，在 fprintf() 调用中， msg 将作为 fprintf() 的格式化字符串，可能引发如同示例 1一样的问题。而且， syslog() 函数也一样存在格式化字符串的问题。

正确做法：格式化字符串由代码确定，未经检查过滤的用户输入只能作为参数。

```
void   check_password(    char   *user,      char   *password)
{
        if    (strcmp(password(user), password) != 0)
    {
            char   *msg = malloc(strlen(user) + 100);
            if    (!msg)
        {
                /* handle error condition */
        }
        sprintf(msg,              "%s password incorrect"        , user);
        fprintf(stderr,                "%s" , user);
        syslog(LOG_INFO,             "%s" , msg);
        free(msg);
    }
}
```

13.4 文件I/O安全

规则 13.8 避免使用 strlen() 计算二进制数据的长度。

说明： strlen() 函数用于计算字符串的长度，它返回字符串中第一个 NULL结束符之前的字符的数量。

因此用 strlen() 处理文件 I/O 函数读取的内容时要小心，因为这些内容可能是二进制也可能是文本。

示例：

```
char   buf[BUF_SIZE + 1];
```

```
if  (fgets(buf,  sizeof  (buf), fp) == NULL)
{
    /* handle error */
}
buf[strlen(buf) - 1] =  '\0'  ;
```

上述代码试图从一个输入行中删除行尾的换行符（ \n ）。如果 buf 的第一个字符是 NULL, strlen(buf) 返回 0, 这时对 buf 进行数组下标为 [-1] 的访问操作将会越界。

正确做法：在不能确定从文件读取到的数据的类型时，不要使用依赖 NULL结束符的字符串操作函数。

```
char  buf[BUF_SIZE + 1];
char  *p;
if  (fgets(buf,  sizeof  (buf), fp))
{
  p = strchr(buf,  '\n'  );
    if  (p)
  {
    *p =  '\0'  ;
  }
}
else
{
    /* handle error condition */
}
```

规则 13.9 使用 int 类型变量来接受字符 I/O 函数的返回值。

说明：字符 I/O 函数 fgetc() 、 getc() 和 getchar() 都从一个流读取一个字符，并把它以 int 值的形式返回。如果这个流到达了文件尾或者发生读取错误， 函数返回 EOF。fputc() 、putc() 、putchar() 和 ungetc() 也返回一个字符或 EOF。

如果这些 I/O 函数的返回值需要与 EOF 进行比较，不要将返回值转换为 char 类型。因为 char 是有符号 8 位的值， int 是32位的值。如果 getchar() 返回的字符的 ASCII 值为 0xFF，转换为 char 类型后将被解释为 EOF。因为这个值被有符号扩展为 0xFFFFFFFF（EOF 的值）执行比较。

示例：

```
char  buf[BUF_SIZE];
char  ch;
int  i = 0;
while  ( (ch = getchar()) !=  '\n'  && ch != EOF )
{
    if  ( i < BUF_SIZE - 1 )
  {
    buf[i++] = ch;
  }
}
buf[i] =  '\0'  ;  /* terminate NTBS */
```

正确做法：使用 int 类型的变量接受 getchar() 的返回值。

```
char  buf[BUF_SIZE];
int  ch;
int  i = 0;
while  (((ch = getchar()) !=  '\n'  ) && ch != EOF)
{
    if  (i < BUF_SIZE - 1)
```

```
    {
        buf[i++] = ch;
    }
  }
  buf[i] =      '\0'    ;   /* terminate NTBS */
```

对于 sizeof(int)　　== sizeof(char)　　的平台，用　int　接收返回值也可能无法与　　EOF区分，这时要用　feof() 和 ferror()　　检测文件尾和文件错误。

13.5 其它

规则 13.10　防止命令注入。

说明： C99函数 system()　通过调用一个系统定义的命令解析器（如　　UNIX的 shell　，　Windows的 CMD.exe）来执行一个指定的程序　　/ 命令。类似的还有　POSIX的函数　popen()　。

如果 system()　的参数由用户的输入组成,　　恶意用户可以通过构造恶意输入,　　改变 system()　调用的行为。

示例：

```
  system(sprintf(      "any_exe %s"   , input));
```

如果恶意用户输入参数：

```
happy; useradd attacker
```

最终 shell　将字符串“　any_exe　happy; useradd attacker　　”解释为两条独立的命令：

正确做法：使用　POSIX函数 execve()　代替 system().

```
  void   secuExec (    char  *input)
  {
      pid_t pid;
      char  * const   args[] = {      ""  , input, NULL};
      char  * const   envs[] = {NULL};
      pid = fork();
      if   (pid == -1)
    {
      puts(      "fork error"      );
    }
    else      if   (pid == 0)
    {
            if   (execve(    "/usr/bin/any_exe"       , args, envs) == -1)
      {
        puts(          "Error executing any_exe"       );
      }
    }
      return   ;
}
```

Windows环境可能对　execve()　的支持不是很完善，建议使用　　Win32 API CreateProcess()　　代替 system()　。

14　单元测试

规则 14.1　在编写代码的同时，或者编写代码前，编写单元测试用例验证软件设计　　/ 编码的正确。

建议 14.1　单元测试关注单元的行为而不是实现，避免针对函数的测试。

说明：应该将被测单元看做一个被测的整体，根据实际资源、进度和质量风险，权衡代码覆盖、打桩工作量、补充测试用例的难度、被测对象的稳定程度等，一般情况下建议关注模块 / 组件的测试，尽量避免针对函数的测试。尽管有时候单个用例只能专注于对某个具体函数的测试，但我们关注的应该是函数的行为而不是其具体实现细节。

15　可移植性

规则 15.1　不能定义、重定义或取消定义标准库 / 平台中保留的标识符、宏和函数。

建议 15.1　不使用与硬件或操作系统关系很大的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。

说明：使用标准的数据类型，有利于程序的移植。

示例：如下例子（在 DOS下BC3.1 环境中），在移植时可能产生问题。

```
void  main()
{
      register    int  index;    //  寄存器变量

   _AX = 0x4000;       // _AX  是BC3.1提供的寄存器 “伪变量 ”
   ...        // program code
}
```

建议 15.2　除非为了满足特殊需求，避免使用嵌入式汇编。

说明：程序中嵌入式汇编，一般都对可移植性有较大的影响。

16　业界编程规范

本次编程规范整理的原则是求精不求全，主要针对华为当前编码上的突出问题，所以在全面性上不免有所欠缺。业界一些公司、组织也发布了一些编程规范，对编程语言的缺陷、使用风险都有很好的描述，这里做一些简单的推介，有兴趣的同学可以在平时学习中可以参考，提高自己的编程能力。

google C++ 编程指南

目标：

增强代码一致性，创建通用的、必需的习惯用语和模式可以使代码更加容易理解

C++是一门包含大量高级特性的巨型语言， 某些情况下， 我们会限制甚至禁止使用某些特性使代码简化，避免可能导致的各种问题

包含的内容：头文件、命名规则、注释、语言特性的使用规则、编码格式

特点：强调理解基础上的遵循，一个规则通常明确说明其优点、缺点，并举很多例子，让读者在理解的基础上遵循， 不像规章制度那样生硬和抽象， 实际上读起来更像一个教程。 比如： 禁止使用 C++异常，花了一页纸的篇幅来解释使用和不使用的优缺点，非常容易理解

推荐语：读起来非常舒服，抛开编程规范，拿来作为理解学习 C++也是不错的

推荐度：★★★★★

汽车业 C语言使用规范 (MISRA)

目标：因为编译器、编程人员理解、 C语言本等原因，完全放开使用 C语言存在一些风险，因此制定这个规范的目标为了促进 C语言的最为安全的使用而定义一些规则。

特点： 规则都是针对的是 C语言本身缺陷或容易被误解的点， 如： 自动变量如果不初始化就使用会出现随机值、不同类型数据赋值容易出现的隐式转换；没有包含诸如注释、变量名、编码格式等统一编程风格的内容。

推荐语：对 C的缺点了如指掌，可以帮助更好的掌握 C语言，避免编程陷阱，提高程序可靠性。

推荐度：★★★★