iOS 开发代码规范文档

IOS 开发代码规范文档 1
一、原则 1
二、布局 2
1.文件布局 2
2、基本格式 3
3、对齐 4
4、空行空格 6
5、断行 8
三、注释 8
四、命名规则 11
1、基本规则 11
2、资源命名 12
五、变量、常量、宏与类型 12
1、变量、常量以及宏 13
2、类型 14
六、表达式与语句 14
七、函数、方法、接口 18
八、头文件 19
九、可靠性 20
1、内存使用 20
2、指针使用 20
3、类 21
十、断言与错误处理 22
十一、其它补充 22

一、原则

【原则 1.1】首先是为人编写程序，其次才是计算机。
说明：这是软件开发的基本要点，软件的生命周期贯穿产品的开发、测试、生产、用户使用、版本升级和后期维护等长期过程，只有易读、易维护的软件代码才具有生命力。

【原则 1-2】保持代码的简单清晰，避免过分的编程技巧。
说明：简单就是最美。保持代码的简单化是软件工程师的基本要求，不要过分追求技巧，否则会减低程序的可读性。

【原则 1-3】编程时首先达到正确性，其次考虑效率。
说明：编程首先考虑的是满足正确性、健壮性、可维护性、可移植性等质量因素，最后才考虑程序的效率和资源占用。

【原则 1-4】编写代码时要考虑到代码的可测试性。
说明：不可以测试的代码是无法保证质量的，开发人员要牢记这一天来设计、编码。实现设计功能的同时，要提供可以测试、验证的方法。

【原则 1-5】函数(方法)是为一特定的功能而编写，不是万能工具箱。
说明：方法是一个处理单元，是有特定的功能的，所以应该很好地规划方法，不能是所有的东西都放在一个方法里实现。

【原则 1-6】鼓励多加注释。

【原则 1-7】内存空间在哪分配在哪释放。

二、布局

程序布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性、可维护性。更重要的是，统一的程序布局和编程风格，有助于提高整个项目的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。因此，统一的、良好的程序布局和编程风格不仅仅是个人主观美学上的或是形式上的问题，而且会涉及到产品质量，涉及到个人编程能力的提高，必须引起大家重视。

1.文件布局

【规范 2-1-1】遵循统一的布局顺序来编写头文件。
说明：以下内容某些不需要，可以忽略。但是需要的要保持该次序。

正例：

1、文件引用
2、类引用
3、宏定义
4、常量
5、类型定义
6、类定义

【规范 2-1-2】遵循统一的布局顺序来编写实现文件。
说明：以下内容某些不需要，可以忽略。但是需要的要保持该次序。

正例：
1、文件引用
2、宏定义
3、常量
4、类型定义
5、全局变量
6、分类
7、类特性
8、类实现:生命周期、Action、Delegate、通知和 KVO、其它私有方法、重写、懒加载等

【规范 2-1-4】包含标准库头文件用尖括号<>，包含非标准库头文件用双引号“”。
正例：

```
#import <stdio.h>
#import “heads.h”
```

2、基本格式

【规范 2-2-1】if/else/else if/for/while/do 等语句花括号紧跟其后，不论执行语句有多少都需要加花括号{}。
说明：这样防止书写错误，也易于阅读。

正例：

```
if (a = 2) {
   NSLog(@”Test.”);
}
```

反例：下面的代码执行语句竟跟 if 的条件之后，而且没有加{}，违反规则。

```
if (a = 2) NSLong(@”Test.”);
```

【规范 2-2-2】定义指针类型的变量，*应放在变量前面。

正例：

```
float *testBuffer;
```

反例：

```
faloat*     testBuffer;
```

【建议 2-2-3】源程序中关系较为紧密的代码应尽可能相邻。
说明：这样便于程序阅读和查找。

正例：

```
length = 10;
width = 5;          // 矩形的长与宽关系较密切，放在一起。
stringCaption = @“Test”;
```

反例：

```
length = 10;
stringCaption = @“Test”;
width = 5;
```

3、对齐

【规则 2-3-1】禁止使用 TAB 键，必须使用空格进行缩进。缩进为 4 个空格。
说明：消除不同编辑器对 TAB 处理的差异，有的代码编辑器可以设置用空格代替 TAB 键。

【规则 2-3-2】程序的分解符‘{’和‘}’要紧跟方法等其后并且位于同一列，同时与引用它们的语句左对齐。{}之间的代码块使用缩进规则对齐。
说明：这样使代码便于阅读，并且方便注释。

正例：

```
- (void)function:(NSInteger)i {
    while (condition) {
        doSomething();     // 与{ }缩进 4 格
    }
}
```

反例：

```
- (void)function:(NSInteger)i
{
    while (condition)
    {
```

```
            DoSomething();
        }
    }
```

【规则 2-3-3】数组、结构体、矩阵等如果在定义时初始化，按照数组的矩阵结构分行书写。

正例：

```
NSInteger numbers[4][3] = {1, 1, 1,
                           2, 4, 8,
                           3, 9, 27,
                           4, 16, 64};

NSArray *array = @[ @“1”,
                    @“2”,
                    @“3”];
```

【规则 2-3-4】相关的赋值语句等号对齐。

正例：

```
    tPDBRes.wHead     = 0;
    tPDBRes.wTail     = wMaxNumOfPDB - 1;
    tPDBRes.wFree     = wMaxNumOfPDB;
    tPDBRes.wAddress  = wPDBAddr;
    tPDBRes.wSize     = wPDBSize;
```

【规则 2-3-5】所有@property 不要写成员变量和@synthesize，书写格式如下：

正例：

```
    @property (nonatomic, strong) Schedule *schedule;
```

反例：

```
    @property(nonatomic,strong) Schedule*schedule;
```

【建议 2-3-6】在 switch 语句中，每一个 case 分支和 default 在使用{}括起来，{}中的内容需要缩进。

说明：使程序可读性更好。

正例：

```
    switch (iCode) {
        case 1: {
            doSomething();      // 缩进 4 格
            break;
        }
        case 2: {                   // 每一个 case 分支和 default 要用{}括起来
            doOtherThing();
```

```
            break;
        }
          ...                          // 其它 case 分支
        default: {
            doNothing();
            break;
        }
    }
```

4、空行空格

【规则 2-4-1】函数(方法)块之间使用一个空行分隔。

说明：空行起着分隔程序段落的作用。适当的空行可以使程序的布局更加清晰。

正例：

```
    - (void)key {
      [key 实现代码];
    }
// 空一行
    - (void)test {
        [test 实现代码];
    }
```

反例：

```
    - (void)test {
        [Hey 实现代码];
    }
    - (void)key {
        [key 实现代码];
    }
// 两个函数的实现是两个逻辑程序块，应该用空行加以分隔。
```

【规则 2-4-2】一元操作符如”!”、”~”、”++”、”--”、”*”、”&”(地址运算符)等前后不加空格。

正例：

```
    !value
    ~value
    ++count
    *stringSource
    &sum
    number[i] = 5;
    box.width
    box->width
```

【规则 2-4-3】多元运算符和它们的操作数之间至少需要一个空格。

正例：

```
value = oldValue;
total + fValue
number += 2;
```

【规则 2-4-4】关键字之后要留空格。
说明：if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。

【规则 2-4-5】函数名之后不要留空格。
说明：函数名后紧跟左括号‘(’，以与关键字区别。

【规则 2-4-6】方法名与形参不能留空格，返回类型与方法标识符有一个空格。
说明：方法名后紧跟’:’，然后紧跟形参，返回类型’-‘与’(‘之间有一个空格。

正例：

```
- (BOOL)test:(NSInteger)i {
   // Return YES for supported orientations.
   return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

【规则 2-4-7】‘(’向后紧跟‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。‘,’之后要留空格。‘;’不是行结束符号时后面要留空格。

正例：

```
for (NSInteger i = 0; i < 10; i++) {
   doSomething(width, iHeight);
}
```

【规则 2-4-8】注释符与注释内容之间要有空格进行分隔，单行注释一个空格，多行注释两个空格。

正例：

```
/**  注释内容  */
// 注释内容
```

反例：

```
/*注释内容*/
//注释内容
```

5、断行

【规则 2-5-1】长表达式(超过 80 列)要在低优先级操作符处拆分成新行，操作符放在新行之首(以变突出操作符)。拆分出的新行要进行适当的缩进，使排版整齐。
说明：条件表达式的续行在第一个条件处对齐。
for 循环语句的续行在初始化条件语句处对齐。
函数调用和函数声明的续行在第一个参数处对齐。
赋值语句的续行应在赋值号处对齐。

正例：

```
// 条件表达式的续行在第一个条件处对齐
if ((format == CH_A_Format_M)
   && (iOfficeType == CH_BSC_M)) {
       doSomething();
}

// 函数声明的续行在第一个参数处对齐
- (void)reportStatusCheckPara:(NSString *)string
                       number:(NSInteger)number;

// 赋值语句的续行应在赋值号处对齐
totalBill = totalBill + faCustomerPurchases[1]
            + salesTax(faCustomerPurchases[2]);
```

【规则 2-5-2】函数(方法)申明时，类型与名称不允许分行书写。

三、注释

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

【规则 3-1】OC 语言的注释符号为多行注释“/* … */”和单行注释“//”.

【建议 3-2】不管多行还是单行，全用注释符“/* … */”。

【规则 3-3】一般情况下，源程序有效注释量在 30%以上。
说明：注释的原则是有助于对程序的阅读理解，注释不宜太多也不能太少，注释

语言必须准确、易懂、简洁。有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息。

【规则 3-4】注释使用中文。
说明：对于特殊要求的可以使用英文注释，如工具不支持中文或国际化版本时。

【规则 3-5】文件头部必须进行注释。
说明：注释必须列出：版权信息、内容摘要、版本号、作者、完成日期、修改信息等。修改记录部分建议在代码做了大修改之后添加修改记录。备注：文件名称，内容摘要，作者等部分一定要写清楚。

正例：
下面是文件头部的中文注释：

```
/************************************************************************
*
* Copyright © 2018 年 广州绯泛信息科技有限公司. All rights reserved.
*
* 文件名称： // 文件名
* 项目名称： // 项目名
* 作　　者： // 输入作者名字及单位
* 当前版本： // 输入当前版本
* 创建时间： // 创建时间
* 完成日期： // 输入完成日期，例：2016 年 03 月 22 日
* 内容摘要： // 简要描述本文件的内容，包括主要模块、函数及其功能的说明
* 其它说明： // 其它内容的说明


* 修改记录 1：// 修改历史记录，包括修改日期、修改者及修改内容
*　　修改日期：
*　　版 本 号：//版本号
*　　修 改 人：
*　　修改内容：//修改原因以及修改内容说明
* 修改记录 2：…
************************************************************************
*/
```

设置路径：Xcode ▸ Contents ▸ Developer ▸ Platforms ▸ iPhoneOS.platform ▸ Developer ▸ Library ▸ Xcode ▸ Templates ▸ File Templates ▸ Source ▸ Cocoa Touch Class.xctemplate；

【规则 3-6】方法头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、访问和修改的表、修改信息等，出了函数名称和功能描述必须描述

外，其它部分描述建议写。
说明：注释必须列出：函数名称、功能描述、输入参数、输出参数、返 回 值、修改信息等。备注：方法名称、功能描述要正确描述。

正例：

```
/**********************************************************************
* 方法名称：  // 方法名称
* 功能描述：  // 方法功能、性能等的描述
* 输入参数：  // 输入参数说明，包括每个参数的作用、取值说明及参数间关系
* 输出参数：  // 对输出参数的说明。
* 返 回 值：  // 方法返回值的说明
* 其它说明：  // 其它说明
**********************************************************************/
```

【规则 3-7】注释应与其描述的代码相近，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。
说明：在使用缩写时或之前，应对缩写进行必要的说明。

正例：

如下书写比较结构清晰

```
/**   获得子系统索引   */
iSubSysIndex = aData[iIndex].iSysIndex;

/**   代码段 1 注释   */
[代码段 1];

/**   代码段 2 注释   */
[代码段 2];
```

反例 1：

如下例子注释与描述的代码相隔太远。

```
/**   获得子系统索引   */

iSubSysIndex = aData[iIndex].iSysIndex;
```

反例 2：

如下例子注释不应放在所描述的代码下面。

```
iSubSysIndex = aData[iIndex].iSysIndex;

/**   获得子系统索引   */
```

反例 3：

如下例子，显得代码与注释过于紧凑。

```
/**   代码段 1 注释   */
[代码段 1];
/**   代码段 2 注释   */
[代码段 2];
```

【规则 3-8】全局变量要有详细的注释，包括对其功能、取值范围、访问信息及访问时注意事项等的说明。

正例：

```
/**   变量作用说 变量值说明   */
NSInteger tranErrorCode;
```

【规则 3-9】注释与索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。

正例：

```
如下注释结构比较清晰
- (int)doSomething {
   /**   代码段 1 注释   */
   [代码段 1];

   /**   代码段 2 注释   */
   [代码段 2];
}
```

【建议 3-10】尽量避免在注释中使用缩写，特别是不常用的缩写。
说明：在使用缩写时，应对缩写进行必要的说明。

四、命名规则

1、基本规则

好的命名规则能极大地增加可读性和可维护性。同时，对于一个有上百个人共同完成的大项目来说，统一命名约定也是一项必不可少的内容。本章对程序中的所有标识符（包括变量名、常量名、函数名、类名、结构体名、宏定义等）的命名做出约定。

【规则 4-1】标识符要采用英文单词或其组合，便于记忆和阅读，切记使用汉语

拼音来命名。
说明：标识符应当直观且可以拼读，可望文知义，避免使人产生误解。程序中的英文单词一般不要太复杂，用词应当准确。

【规则 4-2】严格禁止使用连续的下划线，下划线也不能出现在标识符头或结尾。建议不使用下划线分割单词。
说明：这样做的目的是为了使程序易读。因为 variable_name 和 variable__name 很难区分，下划线符号‘_’若出现在标识符头或结尾，容易与不带下划线‘_’的标识符混淆。

【规则 4-3】程序中不要出现仅靠大小写区分的相似的标识符。

【规则 4-4】宏、常量和变量名都要使用正规的命名规则，宏用下划线‘_’分割单词，常量和变量使用小驼峰命名，预编译开发的定义使用下划线‘_’开始。

【规则 4-5】程序中局部变量不要与全局变量重名。
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 4-6】方法名用小写字母开头的单词组合而成。
说明：方法名力求清晰、明了，通过方法名就能够判断方法的主要功能。方法名中不同意义字段之间不要用下划线连接，而要把每个字段的首字母大写以示区分。方法命名采用大小写字母结合的形式，但专有名词不受限制。

【建议 4-7】尽量避免名字中出现数字编号，如 value1、value2 等，除非逻辑上的确需要这样的符号。

【建议 4-8】标识符前最好不加项目、产品、部门的标识。
说明：这样做的目的是为了代码的可重用性。

2、资源命名

- 字符串：已使用类类名简写开头。
- 图片：已使用类类名简写开头。

五、变量、常量、宏与类型

变量、常量和数据类型是程序编写的基础，它们的正确使用直接关系到程序

设计的成败，变量包括全局变量、局部变量和静态变量，常量包括数据常量和指针常量，类型包括系统的数据类型和自定义数据类型。本章主要说明变量、常量与类型使用时必须遵循的规则和一些需注意的建议，关于它们的命名，参见命名规则。

1、变量、常量以及宏

【规则 5-1-1】一个变量有且只有一个功能，尽量不要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。

【规则 5-1-2】循环语句与判断语句中，不允许对其它变量进行计算与赋值。
说明：循环语句只完成循环控制功能，if 语句只完成逻辑判断功能，不能完成计算赋值功能。

正例：

```
do {
   [处理语句];
  input = getChar();
} while (input == 0);
```

反例：

```
 do {
     [处理语句];
  } while (input = getChar());
```

【规则 5-1-3】宏定义中如果包含表达式或变量，表达式和变量必须用小括号括起来。
说明：在宏定义中，对表达式和变量使用括号，可以避免可能发生的计算错误。

正例：

```
#define    kHandle(A, B)     (( A ) / ( B ))
```

反例：

```
#define    kHandle(A, B)     (A / B)
```

【规则 5-1-4】宏名命名规则。全部大写，单词之间用‘_’隔开，内部使用小写 k 开头。

正例：

```
#define    kBUTTON_WIDTH       (NSInteger)320
```

反例：

```
#define    kButton_width          (NSInteger)320
```

【规则 5-1-5】宏常量要指定类型。
说明：不同的编译器，默认类型不一样。

正例：

```
#define    kBUTTON_WIDTH       (NSInteger)320
```

反例：

```
#define    kBUTTON_WIDTH       320
```

【建议 5-1-6】对于全局变量通过统一的函数访问。
说明：可以避免访问全局变量时引起的错误。

【建议 5-1-7】最好不要在语句块内声明局部变量。

2、类型

【建议 5-2-1】结构体是针对一种事物的抽象，功能要单一，不要设计面面俱到的数据结构体。
说明：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。结构体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体中。

六、表达式与语句

表达式是语句的一部分，它们是不可分割的。表达式和语句虽然看起来比较简单，但使用时隐患比较多。本章归纳了正确使用表达式和 if、for、while、switch 等基本语句的一些规则与建议。

【规则 6-1】一条语句只完成一个功能。
说明：复杂的语句阅读起来，难于理解，并容易隐含错误。变量定义时，一行只定义一个变量。

正例：

```
int help;
int base;
```

```
int result;
help = base;
result = help + getValue(&base);
```

反例：

```
/**  一行定义多个变量  */
int base, result;
/**  一条语句实现多个功能，base 有两种用途。  */
result = base + getValue(&base);
```

【规则 6-2】在表达式中使用括号，使表达式的运算顺序更清晰。

说明：由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

正例：

```
if (((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))
```

反例：

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
```

【规则 6-3】避免表达式中的附加功能，不要编写太复杂的复合表达式。

说明：带附加功能的表达式难于阅读和维护，它们常常导致错误。对于一个好的编译器，下面两种情况效果是一样的。

正例 ：

```
var[1] = var[2] + var[3];
var[4]++;
result = var[1] + var[4];
var[3]++;
```

反例：

```
result = (var[1] = var[2] + var[3]++) + ++var[4];
```

【规则 6-4】不可将布尔变量和逻辑表达式直接与 false、ture 或者 1、0 进行比较。

说明：ture 和 false 的定义值是和语言环境相关的，且可能会被重定义的。

正例：

```
设 testFlag  是布尔类型的变量
if (testFlag)       //  表示 flag 为真
if (!testFlag)      //  表示 flag 为假
```

反例：

设 testFlag 是布尔类型的变量
if (testFlag == TRUE)
if (testFlag == 1)
if (testFlag == FALSE)
if (testFlag == 0)

【规则 6-5】在条件判断语句中，当整数变量与 0 比较时，不可模仿布尔变量的风格，应当将整型变量用“==”或“!=”直接与 0 比较。

正例：
```
if (value == 0)
if (value != 0)
```

反例：
```
/**  会让人误解 value 是布尔变量  */
if (value)
if (!value)
```

【规则 6-6】不可将浮点变量用“==”或“!=”与任何数字比较。
说明：无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该转化成“>=”或“<=”形式。

正例：
```
if ((result >= -epsinon) && (result <= epsinon))
```

反例：
```
/**  隐含错误的比较  */
if (result == 0.0)
```

【规则 6-7】应当将指针变量用“==”或“！=”与 nil 比较。
说明：指针变量的零值是“空”（记为 NULL），即使 NULL 的值与 0 相同，但是两者意义不同。

正例：
```
/**  head 与 NULL 显式比较，强调 head 是指针变量  */
if (head == nil)
if (head != nil)
```

反例：
```
/**  容易让人误解 head 是整型变量  */
if (head == 0)
if (head != 0)
```
或者

```
/**   容易让人误解 head 是布尔变量   */
if (head)
if (!head)
```

【规则 6-8】在 switch 语句中，每一个 case 分支必须使用 break 结尾，最后一个分支必须是 default 分支。
说明：避免漏掉 break 语句造成程序错误。同时保持程序简洁。
对于多个分支相同处理的情况可以共用一个 break，但是要用注释加以说明。

【规则 6-9】不可再 for 循环体内修改循环变量，防止 for 循环失去控制。

【建议 6-10】循环嵌套次数不大于 3 次。

【建议 6-11】do while 语句和 while 语句仅使用一个条件。
说明：保持程序简洁。如果需要判断的条件较多，建议用临时布尔变量先计算是否满足条件。

【建议 6-12】如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。
说明：下面两个示例中，反例比正例多执行了 num-1 次逻辑判断。并且由于前者总要进行逻辑判断，使得编译器不能对循环进行优化处理，降低了效率。如果 num 非常大，最好采用正例的写法，可以提高效率。

```
const int num = 100000;
```
正例:
```
if (condition) {
   for (i = 0; i < num; i++) {
      doSomething();
   }
} else {
   for (i = 0; i < num; i++) {
      doOtherthing();
   }
}
```

反例:
```
for (i = 0; i < num; i++) {
   if (condition) {
      doSomething();
   } else {
      doOtherthing();
```

```
    }
}
```

【建议 6-13】for 语句的循环控制变量的取值采用“半开半闭区间”写法。
说明：这样做更能适应 OC 语言数组的特点，OC 语言的下标属于一个“半开半闭区间”。

正例：

```
NSArray * scoreArray = @[@"1", @"2", @"3"];
for (NSInteger i = 0; i < [scoreArray count]; i++) {
  NSLog(@“%@”, scoreArray[i]);
}
```

反例：

```
NSArray * scoreArray = @[@"1", @"2", @"3"];
for (NSInteger i = 0; i <= [scoreArray count] - 1; i++) {
  NSLog(@“%@”, scoreArray[i]);
}
```

相比之下，正例的写法更加直观，尽管两者的功能是相同的。

七、函数、方法、接口

【规则 7-1】方法不能为多个目的服务。

正例：

```
- (BOOL)test1:(NSInteger)i;
- (BOOL)test2:(NSString *)i;
- (BOOL)test3:(byte)i;
```

反例：

```
- (BOOL)test:(id)i;
```

【规则 7-2】在组件接口中应该尽量少使用外部定义的类型(重用、减少耦合)。

【建议 7-3】避免函数有太多的参数，参数个数尽量控制在 5 个以内。
说明：如果参数太多，在使用时容易将参数类型或顺序搞错，而且调用的时候也不方便。如果参数的确比较多，而且输入的参数相互之间的关系比较紧密，不妨把这些参数定义成一个对象，然后把对象当成参数输入。

【规则 7-4】对于有返回值的函数(方法)，每一个分支都必须有返回值。

说明：为了保证对被调用函数返回值的判断，有返回值的函数中的每一个退出点都需要有返回值。

【规则 7-5】对输入参数的正确性和有效性进行检查。
说明：很多程序错误是由非法参数引起的，我们应该充分理解并正确处理来防止此类错误。

【建议 7-6】函数(方法)的功能要单一，不要设计多用途的函数(方法)。
说明：多用途的函数往往通过在输入参数中有一个控制参数，根据不同的控制参数产生不同的功能。这种方式增加了函数之间的控制耦合性，而且在函数调用的时候，调用相同的一个函数却产生不同的效果，降低了代码的可读性，也不利于代码调试和维护。

【建议 7-7】函数(方法)体的规模不能太大，尽量控制在 200 行代码之内。
说明：冗长的函数不利于调试，可读性差。

【规则 7-8】避免设计多参数函数(方法)。

八、头文件

【规则 8-1】如果不是确定需要，应该尽量避免头文件包含其它的头文件。
说明：头文件中应避免包含其它不相关的头文件，一次头文件包含就相当于一次代码拷贝。

【规则 8-2】申明成员类，应该引用该类申明，而不是包含该类的头文件。

正例：

```
@class SubClassName；
@interface ClassName : NSObject
@end
```

反例：

```
#import “SubClassName.h”；

@interface ClassName : NSObject
@end
```

九、可靠性

为保证代码的可靠性，编程时请遵循如下基本原则，优先级递减：
- 正确性，指程序要实现设计要求的功能。
- 稳定性、安全性，指程序稳定、可靠、安全。
- 可测试性，指程序要方便测试。
- 规范/可读性，指程序书写风格、命名规则等要符合规范。
- 全局效率，指软件系统的整体效率。
- 局部效率，指某个模块/子模块/函数的本身效率。
- 个人表达方式/个人方便性，指个人编程习惯。

1、内存使用

【规则 9-1-1】防止内存操作越界。
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。

【规则 9-1-2】必须对动态申请的内存做有效检查，并进行初始化；动态内存的释放必须和分配成对以防止内存泄露，释放后内存指针置为 nil。
说明：对嵌入式系统，通常内存是有限的，内存的申请可能会失败，如果不检查就对该指针进行操作，可能出现异常，而且这种异常不是每次都出现，比较难定位。指针释放后，该指针可能还是指向原有的内存块，可能不是，变成一个野指针，一般用户不会对它再操作，但用户失误情况下对它的操作可能导致程序崩溃。

【规则 9-1-3】变量在使用前应初始化，防止未经初始化的变量被引用。
说明：不同的编译系统，定义的变量在初始化前其值是不确定的。有些系统会初始化为 0，而有些不是。

2、指针使用

【规则 9-2-1】指针类型变量必须初始化为 nil。

【规则 9-2-2】指针不要进行复杂的逻辑或算术操作。
说明：指针加一的偏移，通常由指针的类型确定，如果通过复杂的逻辑或算法操

作，则指针的位置就很难确定。

【规则 9-2-3】减少指针的数据类型的强制类型转化。

【规则 9-2-4】对变量进行赋值时，必须对其值进行合法性检查，防止越界等现象发生。
说明：尤其对全局变量赋值时，应进行合法性检查，以提高代码的可靠性、稳定性。

【规则 9-2-5】在哪申请在哪释放(非 ARC)。

【规则 9-2-6】非初始化方法中的 alloc 操作之前必须要 nil 判断。

【建议 9-2-7】全局指针释放后置为 nil 值。

3、类

【规则 9-3-1】在编写派生类的赋值时，注意不要忘记对基类的成员变量重新赋值。
说明：除非在派生类中调用基类的赋值函数，否则基类变量不会自动被赋值。

正例：

```
- (void)viewDidLoad {
   [super viewDidLoad];
}
```

【规则 9-3-2】私有方法应该在实现文件中申明。

正例：

```
@interface ClassName ()
- (void)test;
@end

@implementation
- (void)test {
}
@end
```

十、断言与错误处理

断言是对某种假设条件进行检查(可理解为若条件成立则无动作，否则应报告)。它可以快速发现并定位软件问题，同时对系统错误进行自动报警。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。在实际应用时，可根据具体情况灵活地设计断言。

【规则 10-1】整个软件系统应该采用统一的断言。

【规则 10-2】正式软件产品中应把断言及其它调试代码去掉(即把有关的调试开发关掉)。
说明：加快软件运行速度。

【建议 10-3】使用断言检查函数输入参数的有效性、合法性。
说明：检查函数的输入参数是否合法，如输入参数为指针，则可用断言检查该指针是否为空，如输入参数为索引，则检查索引是否在值域范围内。

十一、其它补充

【规则 11-1】避免过多直接使用直接数。

正例：
```
viewBounds.size.height = VIEW_BOUNDS_HEIGHT;
```
反例：
```
viewBounds.size.height = 150;
height = 150;
```

【规则 11-2】addObject 之前要非空判断。