ios平台编程规范

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 归档日期 | 版本描述 | 作者 | 审批人 |
| V1.0 | 2015-11-10 | 初稿整理 | 林建洪 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 目 录

目 录 3

1.1 背景知识 4

1.1.1 名词解释 4

1.1.2 语言特性 4

1.2 目的 4

1.3 规范与注意事项 4

1.3.1 代码格式化 4

1.3.2 禁用 goto 9

1.3.3 一个函数应该只有一个出口 9

1.3.4 常量 10

1.3.5 慎用全局变量 11

1.3.6 函数的行数限制约定 11

1.3.7 第一时间为变量赋初值 12

1.3.8 初始化函数 12

1.3.9 析构函数 12

1.3.10 成员变量 13

1.3.11 临时对象 13

1.3.12 属性 14

1.3.13 命名规范 15

1.3.14 定时器 16

1.3.15 performSelector 17

1.3.16 关于 delegate 17

1.3.17 关于 NSArray/NSSet/NSDictionary 等集合类 18

1.3.18 私有方法的处理 18

1.3.19 关于 retain/release 21

1.4 推荐注释风格 23

1.5 编后语 24

## 背景知识

Objective-C 是扩充 C 的面向对象编程语言，是苹果公司推荐的Mac编程语言，唯一可以为 iPhone/iPad 编程的语言。

### 名词解释

* NS - NextStep (NeXT.Inc开发的操作系统，后被苹果买下)。由于Objective-C没有命名空间机制，因此MacOS X的类别和函数以NS作为前缀。
* Cocoa - 使用 Objective-C 语言编写的用于 MacOS X 开发的类库。

### 语言特性

* Objective-C 不支持多重继承
* 动态性非常强、非常灵活，但增加了运行时出错的可能
* Objective-C 2.0开始支持属性和垃圾回收。

## 目的

本文档主要介绍使用 Objective-C 在项目开发过程中的注意事项以及编码规范，旨在提升代码的可读性与可维护性，同时减少代码出错的机会。

由于 Objective-C 完全兼容 C/C++ 语法，为了完整性，本文档也包括了部分 C++ 编码规范的内容。

本文档假定阅读者已经具备 Objective-C 以及 Cocoa 编程的基本能力。

## 规范与注意事项

### 代码格式化

这里讲的代码格式化规范与 C++ 编码规范一致，另外习惯的养成很重要，不应该将代码编写完成后再格式化代码，而是应该养成良好的编程习惯，每输入一个字符都是符合格式化规范的。

题外话：作为程序员，首先应该严格要求自己养成优良的编程习惯，然后再修炼自身的编程思想，最终提升设计水平。只有具备了扎实的基本功才有可能练就成为出色的架构设计师。

#### 空格的使用

* 等号、双等号、大于号、小于号以及普通运算符的两边都需要使用一个空格，示例代码如下：

int count = [m\_viewList count];

* ++、-- 符号两侧不使用空格，示例代码如下：

for (int i = 0; i < count; ++i)

{

}

vector<int>::iterator it = numList.begin();

for (; it != numList.end(); ++it)

{

}

这里的 ++ 写在 it 的前面或后面对程序的运行结果无任何影响，it++ 可能被实现为 it = it + 1; 这种方式会有新的临时对象产生，因此推荐使用 ++it 的方式，该规则并非必须遵守。

* 括号（包括小括号与中括号）本身是一种分隔符，因此括号两侧通常不使用空格，但关键字 if、for 后面要使用空格：

if (a > b)

{

}

* 分号后面如果不是换行符，则需要使用空格。
* 大括号后面如果不是换行符或分号，则需要使用空格。
* 函数声明中，+/- 号与括号之间需要使用空格，括号与函数名之间不使用空格：

- (id)init;

+ (id)standardInstance;

* +、-、\*、/等运算符两侧，需要使用空格：

int value = (width / 2) + (wd \* 3 / 4);

* 函数参数的冒号两边不使用空格，逗号后面需要使用空格：

NSString\* strNode = [NSString stringWithFormat:@"Node\_%d", index];

三元操作符的使用示范代码：

int value = (a > b) ? a : b;

#### 大括号的使用

每一个大括号应该新起一行，像这样:

if (a > b)

{

}

而不要这样：

if (a > b) {

}

这里要特别说明的是，Xcode 向导产生的代码，可能是大括号与代码同在一行的（应该有设置可以修改），如果是系统产生的，我们不必去修改，但是新写的代码，则要按规范去写。

条件成立的 return 语句，需新起一行，同时需要使用大括号：

if (index < 0)

{

return;

}

而不要这样写：

if (index < 0)

return;

更不要这样写：

if (index < 0) return;

大括号的后面需要有分隔符，可能是换行符或空格。下例中同一行中大括号后还有其它内容，则加一个空格：

do

{

} while (false);

#### 段落结构

在适当的时候应该添加空行，每一个代码块都不应该太大：

m\_backwardButton = [[UIButton alloc] initWithFrame:rectBack]; m\_backwardButton.exclusiveTouch = YES;

m\_backwardButton.tag = BACKWARD;

[m\_imageView addSubview:m\_backwardButton];

m\_forwardButton = [[UIButton alloc] initWithFrame:rectForward]; m\_forwardButton.exclusiveTouch = YES;

m\_forwardButton.tag = FORWARD;

[m\_imageView addSubview:m\_forwardButton];

如以上代码，在每一个业务处理单元后空一行再开始第二个代码块的编写，以保持结构清晰。

不要写成这样：

m\_backwardButton = [[UIButton alloc] initWithFrame:rectBack]; m\_backwardButton.exclusiveTouch = YES;

m\_backwardButton.tag = BACKWARD;

[m\_imageView addSubview:m\_backwardButton];

m\_forwardButton = [[UIButton alloc] initWithFrame:rectForward]; m\_forwardButton.exclusiveTouch = YES;

m\_forwardButton.tag = FORWARD;

[m\_imageView addSubview:m\_forwardButton];

以上代码将非常难以阅读。

通常，一对大括号结束后，应该空一行再开始编写新的代码，例如：

if (nil == view)

{

return;

}

[self addSubview:view];

而不要这样写：

if (nil == view)

{

return;

}

[self addSubview:view];

当大括号后面紧跟的代码是另一个条件分支时，则不能有空行间隔：

if (index < 0)

{

}

else if (index >= count)

{

}

else 语句前不可以插入空行。

#### switch/case 语句的使用

switch 语句要注意几点：

* 每一个 case 都应该要有一个 break，若特殊需要，明确不要 break，则需写注释
* 必须要有 default 分支，哪怕 default 分支什么也不做，也要写上 break
* 每一个分支都使用大括号括起来
* 避免出现两行之间相差两个缩进位的情况

示例代码如下：

switch (direction)

{

case DIR\_LEFT:

{

break;

}

case DIR\_RIGHT:

{

break;

}

default:

{

break;

}

}

#### 缩进层级

缩进不应该超过3级，如下代码是可以的：

for (int i = 0; i < rows; ++i)

{

for (int j = 0; j < cols; ++j)

{

if ((i > (rows / 2)) && (j > (cols / 2)))

{

}

}

}

像以下的代码就应该考虑重构了：

for (int i = 0; i < rows; ++i)

{

for (int j = 0; j < cols; ++j)

{

if ((i > (rows / 2)) && (j > (cols / 2)))

{

if (0 == i)

{

}

}

}

}

通常，每两行代码之间应该最多相差一个缩进位。

### 禁用 goto

goto 是结构化程序设计坚决反对的，可以使用 do{ } while (false) + break 来完成 goto 的功能。 参考下一节示例代码。

### 一个函数应该只有一个出口

一个 return 语句便是一个出口，这里并不是要求每一个函数只允许在结尾处有一个 return 语句，而是不允许在函数逻辑处理的中间部分使用 return，在函数头部有一到两个检测参数合法性的判断并 return 的情况是允许的。

当在函数体中间需要 return 的时候，推荐使用 do{ } while (false) + break 来解决。

示例代码如下：

*// 只是一个示范性编码，其实现的功能并无实际意义。*

BOOL isUTF8XmlFile(const char\* fileName)

{

if ((nil == fileName) || !isFileExist(fileName))

{

return NO; *// 函数头部，在进入逻辑处理之前，可以使用return。*

}

BOOL isUTF8Xml = NO;

char\* data = nil;

FILE\* fp = nil;

do

{

fp = fopen(fileName);

if (nil == fp)

{

break;

}

int length = getFileLength(fp);

data = new char[length + 1];

fread(data, length, 1, fp);

data[length] = '\0';

if (!strstr(data, "<?xml "))

{

break;

}

const char\* p = strstr(data, "encoding=");

if (nil == p)

{

break;

}

if (0 == stricmp(p, "encoding=\"UTF-8\""))

{

isUTF8Xml = YES;

}

} while (false);

*// 释放前面分配的内存*

if (fp)

{

fclose(fp);

}

if (data)

{

delete [] data;

}

return isUTF8Xml;

}

上面代码中的 do/while 不是循环，只是起到流程控制的作用，假如将代码中的 break 都换成 return 的话，那么内存以及文件资源就得不到释放，早期的代码可能使用 goto 语句来释放资源，但是那不满足结构化编程的原则，这里禁用 goto。

### 常量

代码中比较语句将常量写在前面是一种较好的习惯（不作强制要求），常量放前面可以防止将 "==" 写成 "=" 而导致程序出现BUG的情况。例如，如果将：

if (NO == br)

{

}

写成了这样：

if (br = NO)

{

}

条件判断将永远不会成立，并且将 br 的值改变成了 NO，可能会产生不可预料的问题，自然就成了BUG。但如果这样写：

if (NO = a)

{

}

编译器立即报错，无法编译，因此这个习惯可以避免一些意外的错误发生。

Objective-C 中的 BOOL 类似于 VC 中的 BOOL，是一个宏定义，标识真假的常量使用 YES 或 NO，而不要使用 TRUE 和 FALSE 。C++ 代码中才使用 TURE/FALSE。若类型定义为 bool，则使用 true 或 false。推荐使用大写的 BOOL/YES/NO。

### 慎用全局变量

全局变量的缺点是容易出现野指针的情况，另外若程序段使用了全局变量，那么该段代码通常不是线程安全的，一旦放在多线程环境中就会出现问题，而且出现问题的时候很难快速定位到问题代码。全局变量的使用也会给程序的维护带来麻烦。

### 函数的行数限制约定

* 一个函数的长度不应该超过80行
* 当函数长度超过80行后，应该将内部一些复杂的逻辑提炼出来，形成新的函数，然后调用之，微型重构工作也应该无处不在，而不是等项目完成后再来重构。
* C++ 编程规范中对每一行代码的长度有不能超过80个字符的规定，由于 Objective-C 的语法特性，导致代码可能会更长，因此这里未做强行规定，但是也不能无限制的增加每一行代码的长度，建议超过100字符时做折行处理，处理时请以结构清晰为原则。可在 Xcode -> Preferences -> Text Editing 中的 Page guide at column 中设置每行100字符宽的提示线，如果一行的代码超过提示线，则应该进行折行处理。
* 一个类不应该将很多复杂的逻辑揉合到一起来实现，在 C++ 中，通常一个类的实现代码超过800行就被认为该类是一个比较复杂的类，Objective-C 的源码文件可能会更长一些。我们约定当 .mm 文件超过1500行时，要考虑将这个文件进行拆分，可以使用类目的方法来分离功能代码。如果逻辑过于复杂，则应该考虑从设计上将一些内部可以独立的逻辑提炼出来，形成新的类，以减轻单一类的复杂度。

### 第一时间为变量赋初值

由于 C 语言只能在函数开始处将所有要使用的变量都声明好，因此很多代码中出现不为变量赋初值的情况，这在 C++ 编码规范中是极力反对的，C++ 规范要求变量不要过早定义，而是在第一次使用时再定义，保证每一个变量的生命期最小化。 示例代码：

NSString\* strNode = [NSString stringWithFormat:@"Node\_%d", index];

不要这样写：

NSString\* strNode;

strNode = [NSString stringWithFormat:@"Node\_%d", index];

可以这样写：

NSString\* strNode = nil;

strNode = [NSString stringWithFormat:@"Node\_%d", index];

### 初始化函数

- (id)init

{

self = [super init];

if (self)

{

*// Do Something*

}

return self;

}

初始化代码应该在 self 有效的情况下进行，否则运行过程极可能会出现崩溃的情况。

若特殊情况下，需要在 [super init] 成功的情况下还返回空，那么请调用 [self release]，再返回 nil，否则会内存泄漏。

成员变量不必做初始化，编译器会将所有成员变量赋空值，当然如果期望初值不是空值，就在 [super init] 成功的条件分支里面做初始化工作。

C++ 代码中的成员变量则必须要在构造函数中初始化，除非这些变量本身是类对象。

### 析构函数

- (void)dealloc

{

*// Do Something*

[super dealloc];

}

* 所有 Objective-C 类成员变量都应该在析构函数中 release
* 最好将清理工作代码写在 [super dealloc] 之前
* 永远不要调用 dealloc 函数
* 推荐将 dealloc 函数放在 @synthesize 之后，这样可以对照着将成员变量及属性释放，而不会遗漏。

### 成员变量

* 成员变量不允许使用 autorelease
* 若成员变量只在一个函数中会用到，那么请将它从成员变量中删除，直接用临时变量。临时对象推荐使用 autorelease，不用关心释放时机。
* 成员变量在非析构函数中 release 后一定要立即置空
* 成员变量一定要在析构函数（dealloc）中 release

- (void)dealloc

{

[m\_webView release];

m\_webView = nil;

[m\_addButton release];

m\_addButton = nil;

[super dealloc];

}

由于 Objective-C 中，向一个 nil 指针发送消息是安全的，因此不必做空指针检测。

对于便捷构造方法创建的对象，赋值给成员变量时，必须 retain，例如：

m\_backImage = [UIImage imageNamed:@"background.png"];

[m\_backImage retain];

特例：当 Timer 作为成员变量的时候，推荐使用便捷构造方法创建 Timer，那么该 Timer 本身是 autorelease 的，因此对于 Timer 类型的成员变量，便捷构造方法创建后不强制要求使用 retain，因为 Timer 有一个特殊的方法 invalidate 需要调用。

### 临时对象

- (id)initWithFrame:(CGRect)rect

{

self = [super initWithFrame:rect];

if (self)

{

UIView\* testView = [[UIView alloc] initWithFrame:self.bounds];

[self addSubview:testView];

[testView release];

}

return self;

}

临时的UIView对象（testView ）当添加到 superview 中以后，可以立即 release，因为 addSubview 会为其增加引用计数，当 superview 析构时，或显示调用 removeFromSuperview 时，其引用计数会自动减1，因此不用担心释放时机问题。

临时对象也可以使用 autorelease，代码会更加简洁：

- (id)initWithFrame:(CGRect)rect

{

self = [super initWithFrame:rect];

if (self)

{

UIView\* testView = [[[UIView alloc] initWithFrame:self.bounds] autorelease];

[self addSubview:testView];

}

return self;

}

### 属性

以下为页面参数类定义：

@interface WebParam : NSObject

{

}

@property (retain) NSString\* url; *// 页面URL* @property (retain) NSString\* title; *// 页面标题* @property (nonatomic, retain) UIImage\* icon; *// 站点图标*

* 当属性需要在多线程中使用时，不能使用 nonatomic，否则可能会出现异常。
* 当属性指定了 retain 时，必须在析构中 release，否则会内存泄漏：

- (void)dealloc

{

[url release];

[title release];

[icon release];

[super dealloc];

}

当我们在类声明中将属性再次定义（不是必须的，建议参考1.3.18.2方式实现），目的是为了调试时可以方便的看到这些属性的值（当未显示声明的时候，编译器会自动生成同名成员变量）：

@interface WebParam : NSObject

{

NSString\* url;

NSString\* title;

UIImage \* icon;

}

但请不要把这些属性直接当作成员来访问，访问时必须使用属性访问方法：

NSLog(@"%@", self.url);

而不要这样写：

NSLog(@"%@", url);

否则当他人阅读代码的时候会误认为是临时变量，而且假如这样写：

url = inputUrl;

则产生了内存泄漏，因为 url 是 retain 的，先前的 url 得不到释放。如果这样写：

self.url = inputUrl;

这样就不会产生内存泄漏了。

* 当在 init、dealloc 或 setter 与 getter 以外的方法中访问自身属性变量的时候，请一定要记得使用属性存取方法，而不要直接使用成员变量访问方法。

### 命名规范

#### 成员变量

成员变量的命名不必严格按照匈牙利命名规则，但是一定要使用 m\_ 作为前缀，前缀后面的第一个字母要小写，之后连接的每个单词首字母大写。例如：

UIView\* m\_webView;

UIButton\* m\_addButton;

UIImage\* m\_backgroundImage;

命名遵循直观易读的原则，让人顾名思义，一看就知道是什么用途的。

#### 属性

属性的命名按照成员变量的命名原则，去掉m\_前缀即可。例如：

@property (retain) UIButton\* addButton; *// 添加按钮*

@property (nonatomic, assign) BOOL active; *// 是否活动页面*

@property (nonatomic, readonly) BOOL canForward; *// 是否可以前进*

#### 临时变量

临时变量的命名可以参照属性的命名规则，由于我们要求对自身属性的访问，一定要加上 “self.”，因此从名称完全可以很容易区分出到底是成员变量、临时变量以及属性。

功能特殊的临时变量也可以使用最简洁的命名方式，比如：

int i = 0;

#### 函数

函数的命名第一个单词首字母小写，之后连接的单词首字母大写，例如：

- (void)clicked:(id)sender;

- (void)clearButtonClicked:(id)sender;

- (void)setBackgroundImage:(UIImage\*)image;

- (void)setActive:(BOOL)active;

- (BOOL)active;

获取一些属性的函数直接使用属性名称命名，而不必写成 getActive 的方式，这样将与内置属性的访问方式一致；如果获取的属性是 BOOL 类型，可以使用 isActive 的方式。

### 定时器

需要循环使用（指定了 repeat 参数为 YES）的 Timer，必须要在合适的时机调用 invalidate 方法，否则会出现内存泄漏，在使用类的析构函数中调用 Timer 的 invalidate 方法为时已晚，因为 timer 会对其传递的目标 object 增加引用计数，若不调用 invalidate，使用类根本得不到析构。 对于指定了 repeat 参数为 NO 的 Timer，则可以不调用 invalidate 方法。

[NSTimer scheduledTimerWithTimeInterval:0.1 target:self selector: @selector(timerCallback:) userInfo:nil repeats:NO];

### performSelector

[self performSelector:@selector(fun:) withObject:nil afterDelay:1.0];

上面的代码可能导致运行时出现野指针而崩溃，因为1秒之后 self 可能已经析构。解决方法是将 self 作为 object 参数传进去：

[self performSelector:@selector(fun:) withObject:self afterDelay:1.0];

performSelector会为withObject的参数增加引用计数，并在目标函数执行完毕之后将其 release。

由于fun方法可能本身需要更多的参数，这时我们可以使用字典，将 self 以及其他参数都添加到字典对象中，再将字典对象作为 Object 传入即可。

NSDictionary\* dict = [[NSDictionary alloc] initWithObjectsAndKeys:self, @"Object", nil];

[self performSelector:@selector(fun:) withObject:dict afterDelay:1.0]; [dict release];

对于数组与字典等容器对象，会自动为添加进去的对象增加引用计数，从中移除的时候引用计数减1，因此不必担心容器内对象野指针的问题。若是在线程中，字典对象最好不要使用 autorelease，因为线程不一定存在自动回收池。

performSelectorOnMainThread 将有可能出现同样的问题，除非设置 waitUntilDone 为 YES。

### 关于 delegate

当我们在编写一个功能模块的时候，可能需要通知使用者发生了什么事件，这时候禁止直接去调用使用者的方法，而是使用 delegate 模式。

需要说明的是，我们可能会在一个 delegate 协议中设计很多事件通知接口，但是不能假定使用者一定会关心所有事件，也就是不要求使用者实现 delegate 协议定义的所有方法。那么我们在使用 delegate 进行回调的时候，应该使用 respondsToSelector 方法检测使用者是否实现了该方法，然后再调用。示例代码如下：

if (self.delegate && [(NSObject\*)self.delegate respondsToSelector:@selector(drawLayer:)])

{

[(id<LayerDelegate>)self.delegate drawLayer:layer];

}

通常，一个控件（或子窗口）不会向它的父窗口（superview）发送消息，而是向其 delegate 发送消息。例如 UIButton 使用 addTarget 来指定接收消息的 delegate 对象与方法，而不要求 delegate 是其 superview。

对于delegate的使用，只允许调用其 delegate 协议约定的方法，而不应该访问其他方法，例如 retain 与 release 方法，否则极容易出现内存泄漏。因此也不应该为 delegate 属性指定 retain 参数：

@property (nonatomic, assign) id<Delegate> delegate; *// 回调通知接口*

**Delegate的使用规则：**

* delegate 属性不应该指定retain参数；
* delegate 赋值配对原则：为一个对象指定了 delegate，就一定要为其置空；
* delegate 属性的值只能由使用者来修改，不要试图修改自身 delegate 属性的值；
* 不要调用自身 delegate 属性的 retain/release 等方法，只能调用协议方法。

### 关于 NSArray/NSSet/NSDictionary 等集合类

集合类会自动为加入的对象增加引用计数，当移除对象或自身析构时会将其对象引用计数减1。

从效率与安全性因素考虑，遍历集合类的元素时，推荐使用快速迭代器，例如：

for (WebPage\* page in m\_pageList)

{

}

而不要使用获取count然后根据下标获取元素的方法：

int pageCount = [m\_pageList count];

for (int i = 0; i < pageCount; ++i)

{

}

对于字典，其快速迭代器返回的并非目标对象，而是字典中的 Key（NSString\* 类型）：

for (NSString\* key in m\_controllerDict)

{

id<IUCController> contoller = [m\_controllerDict valueForKey:key];

}

如果要简单遍历时使用如下方法：

for (id<IUCController> contoller in [m\_controllerDict allValues])

{

}

### 私有方法的处理

#### Category的使用

内部使用的私有方法，不要随意暴露到接口头文件里面，因为 Objectiv-C 不支持 C++ 的 private 方法。变通的处理方法是使用类目（Category），将扩展的类目直接写在类的 .mm 文件里面，代码位于类实现代码的前面即可，示例：

*//*

*// UIImageCache.h*

@interface UICacheImage : UIImage

{

}

+ (UICacheImage\*)cacheImageWithFile:(NSString \*)path;

@end

该类中，我们打算实现两个私有方法：

- (int)fileSize;

- (NSString\*)fileName;

既然是私有方法，那么我们并不希望放到接口头文件里面去声明，干扰使用者阅读，因此编写了 UICacheImage 的一个私有类目 UICacheImage (Private)。代码如下：

*//*

*// UIImageCache.mm*

@interface UICacheImage (Private)

- (int)fileSize;

- (NSString\*)fileName;

@end

@implementation UICacheImage (Private)

- (int)fileSize

{

return 0;

}

- (NSString\*)fileName

{

return @"";

}

@end

#pragma mark UICacheImage

@implementation UICacheImage

+ (UICacheImage\*)cacheImageWithFile:(NSString \*)path

{

return nil;

}

@end

如此一来，就可以在类的实现方法中使用私有方法了，而不必将这些私有方法声明在类的接口头文件中。

#### 关于 Category 与 Extension

Category 是 Objective-C 2.0 之前很广泛的用法，Category 的设计本身是为了给已有的类增加方法的，所以除非我们写 implementation 时专门写一个 Private 的实现体，否则编译器是不会去检查这个函数是否有实现体，万一漏写了，编译器不会报告 warning，等真正 call 到这个函数时，出现崩溃。

而且Category也无法新增加成员变量或者 property。（property 可以声明，但是无法 synthesize，只不过相当于增加了两个函数而已）

为此，Objective-C 2.0中增加了 Extension 机制，语法跟 Category 类似，仅仅只是在括号中不用写名称。

@interface BookmarkAgent()

@property(nonatomic, retain) NSString\* baseURL;

- (void)loadBookmarkData;

@end

但是，这个语法的原理却完全不一样，在 Extension 中声明的方法要求必须在主实现体中实现，否则编译器会 warning，帮助查错。

在 Extension 中新增加的 property 可以 synthesize 出成员变量：

@synthesize baseURL = m\_baseURL; *// 头文件中无须写m\_baseURL的声明*

这样就可以增加一些头文件看不见的私有变量了，甚至是对外只可读的 property，在 Extension 中也可以重新声明为可写的，进一步减少对成员变量的依赖。

尽管语法上两者几乎一致，Extension 是比 Category 更加强大的功能，建议大家在写实现文件时可以使用（当然，如果实现体是在两个不同的文件的情况，还是只能用 Category）。 参考：[[1]](http://stackoverflow.com/questions/4685679/minutia-on-objective-c-categories-and-extensions)

PS：如果使用 Apple LLVM 3.0 编译器，甚至可以在 Extension 中直接增加成员变量：

@interface AddLinkController ()

{

int m\_privateFlag;

}

@end

但是我们现在的工程仍然是用 GCC 4.2 编译的，这样的语法暂时还是编不过的。

### 关于 retain/release

Objective-C 中，retain 与 release 需要遵循配对调用的原则，同时也应该遵循谁 retain，谁就要负责 release（“谁分配，谁释放”原则），既然是引用计数，千万不要在一个类中将一个对象 retain，而在另一个类中 release。

以下的代码从功能和逻辑上看都没有任何错误，也没有内存泄漏：

@implementation PageView

- (void)fun

{

UIImage\* image = [[UIImage alloc] initWithContentsOfFile:IMAGE\_FILE]; [self.property setPageIcon:image]; *// 疑惑：image是否需要release?*

}

@end

@implementation WebProperty

- (void)setPageIcon:(UIImage\*)image

{

self.icon = image;

[image release];

}

@end

但是该代码违背了“谁分配，谁释放”的原则，当阅读到上例中加亮的代码时，阅读者会担心 image 会不会泄漏，然后去猜测 WebProperty:setIcon 函数到底会不会去释放传入的 image，增加了代码维护的难度。因此规范规定不允许出现这样的代码，正确的写法如下：

@implementation PageView

- (void)fun

{

UIImage\* image = [[UIImage alloc] initWithContentsOfFile:IMAGE\_FILE];

[self.property setPageIcon:image];

[image release];*// 我分配的image，我负责释放，不会产生疑惑*

}

@end

@implementation WebProperty

- (void)setPageIcon:(UIImage\*)image

{

self.icon = image;

}

@end

这样的代码不会给阅读者带来困难，“谁分配，谁释放”，假如 setIcon 的实现者不处理传入的 image，外部释放的时候 image 则析构，假如 setIcon 的实现者需要维持 image 的生命期，那么它应该将其 retain，用完之后再 release，比如在析构中 release，那么我们外部的 release 也只是起到将引用计数减一的作用，而不会导致对象析构。对象的分配者不必关心使用者是否正确处理该对象，这便是引用计数的设计精髓所在（引用计数机制本就是为了解决这类问题而设计的）。

如果要重载属性存取方法，而且该属性是制定了 retain 参数的，应该这样写：

@property (retain) NSString\* icon;

- (void)setIcon:(UIImage\*)image

{

if (icon == image)

{

return;

}

[icon release];

icon = [image retain];

}

为了使用方便，通常可以使用 autorelease 来替代 release，autorelease 将延缓 release 的时机，autorelease 通常会在下一个 RunLoop 中调用对象的 release 方法。

Q：哪些对象需要 release，哪些对象不需要 release？

A：

1. alloc 返回的对象需要 release；
2. 名称包含create的函数返回的对象需要release；
3. 显示调用retain的对象需要release；
4. copy和mutableCopy返回的对象需要release；
5. new方法返回的对象需要release；
6. 便捷构造方法创建的对象不需要release。

我们在设计函数的时候也要注意这些规则：如果我们提供如下类似

- (id)icon;

- (NSString\*)fileName;

这类方法的时候，应该是不需要使用者 release 的，我们可以这样写：

- (id)icon

{

UIImage\* image = [[UIImage alloc] initWithContentsOfFile:IMAGE\_FILE];

return [image autorelease];

}

或这样写：

- (id)icon

{

return [[[UIImage alloc] initWithContentsOfFile:IMAGE\_FILE] autorelease];

}

当使用者拿到该方法返回的对象，使用后不需要 release，假如使用者需要用成员变量来记录这个对象，那么它应该将其 retain，析构中对这个成员调用 release。如果记录了对象指针而不 retain，则会出现野指针而导致程序崩溃的情况。

autorelease 的真实意义并非自动释放，而是延时释放的意思。千万不要把 autorelease 跟自动回收联系起来。iOS 5 据说已经支持垃圾回收，但是我们不要使用该特性。

## 推荐注释风格

* 文件头注释：

*/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* File : ThumbnailView.h*

*\**

*\* Description : A UIView shows webpages' thumbnail matrix*

*\**

*\* Creation : 2013/11/12*

*\* Author : zihong*

*\* History :*

*\* Creation, 2013/11/12, zihong Create the file \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*/*

* 对属性、变量声明的注释：

@property (nonatomic, readonly) BOOL isHomePage; *// 是否首页*

@property (nonatomic, readonly) BOOL canForward; *// 是否可以前进* @property (nonatomic, readonly) BOOL canBackward; *// 是否可以后退*  @property (nonatomic, assign) BOOL active; *// 是否活动窗口*

* 注释可以和代码处于同一行，也可以在代码的上一行：

*// 添加新页面*

- (void)addNewPage: (UIPageView\*)page ;

* 凡是提供给使用者调用的接口，一定要添加适当的注释，原则是让使用者能快速理解接口的设计与参数的意义。
* 内部使用的成员变量也应该都有合适的注释说明。
* 实现文件中，根据需要提供注释。由于 Objective-C 的源码文件通常会比较大，代码行较多，推荐使用 #pragma mark XXX 将代码不同的处理段分隔开，方便在编辑器中快速定位到需要查看的代码。例如在页面管理器中有很多业务逻辑，我们可以将同一逻辑的处理代码放在一起，然后使用如下标记来分隔：

#pragma mark 页面管理

#pragma mark 页面动画

#pragma mark 页面信息保存

#pragma mark 页面抓图

#pragma mark 页面恢复

#pragma mark UCWebViewDelegate Methods

#pragma mark -

#pragma mark - 会在XCode索引列表中添加一条分割线，让分类更加清晰。

如此一来，无论该文件中代码多么复杂，都可以在编辑器中快速定位到指定的代码段，而不会给人找不到北的感觉。

## 编后语

规范是为了让大家能写出统一风格、可读性强、易维护的代码，规范是死的，其中的条款也并非每一条都要百分百的遵守，一些是严格要求，一些是推荐的方法。另外每一个项目都可能有该项目特有的规范，不必拿本文档去对比。

祝大家享受编程的乐趣，而不必在痛苦中挣扎！