# 文档前言

## 基本描述

俗话说的好：“好记性，不如烂笔头”。说明的一个道理就是纪录的重要性，哪怕是一个记忆天才，纪录在其他地方，可以随时看到也是非常重要的。何况我是如此一个平凡的人啊！谦虚学习，踏踏实实，不能好高骛远，天道酬勤才是王道。

本文档旨在纪录开发过程中遇到的一些重要点、难点，也能帮助我纪录我的成长过程。

少波，好好纪录，这是很重要的一个事情，不要觉得我写了一些知道的东西，没什么价值，磨刀不误砍柴工。

## 内容概括

本文包括如下内容：iOS开发、设计模式、数据结构、java服务器开发、前段开发等，若有新技术的添加会及时的添加到文档中。

IOS开发部分

# Ios代码书写规范

## 他人代码规范1

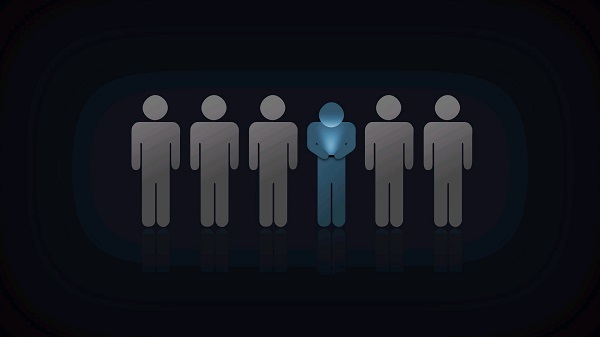
<http://www.cocoachina.com/ios/20160324/15727.html>

iOS开发编码建议与编程经验（持续更新中）

2016-03-24 06:02 编辑： suiling 分类：[iOS开发](http://www.cocoachina.com/ios/) 来源：[乞力马扎罗的雪的博客](http://blog.csdn.net/chenyufeng1991/article/details/50261901)

50 11427

[iOS开发](http://www.cocoachina.com/cms/tags.php?/iOS%E5%BC%80%E5%8F%91/)[编程经验](http://www.cocoachina.com/cms/tags.php?/%E7%BC%96%E7%A8%8B%E7%BB%8F%E9%AA%8C/)[编码建议](http://www.cocoachina.com/cms/tags.php?/%E7%BC%96%E7%A0%81%E5%BB%BA%E8%AE%AE/)

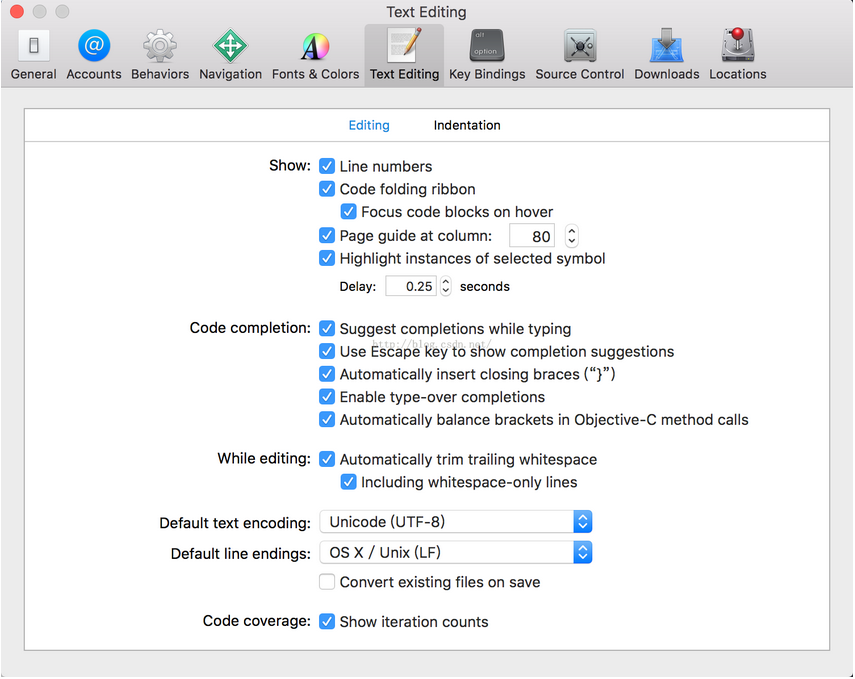


作者：乞力马扎罗的雪（[GitHub](https://github.com/chenyufeng1991)） [原文](http://blog.csdn.net/chenyufeng1991/article/details/50261901)

在开发过程中，我们不仅要去看别人的代码，也要让别人看我们的代码。那么，有一个良好的编码习惯将会非常重要。下面将会罗列使用Objective-C来开发iOS的编码建议。

【1】任意函数长度不得超过50行。（其实很容易就超过50行，这就要考虑代码抽取了。）

【2】任意行代码不能超过80字符。（其实也很容易超过80字符，可以考虑多行显示，比如有多个参数时，可以每个参数放一行。）可以在Xcode中设置超过80个字符的提醒，选中“Page guide at column”.设置完之后就会在代码80个字符处有一条竖线。



【3】在每个方法的定义前留白一行，也就是在方法和方法之间留空一行。

【4】功能相近的方法要放在一起，并推荐使用#pragma mark - \*\*\*来导航代码，切分代码块。这样可以方便函数的查找。并且可以使用快捷键control+6 来快速查找方法的位置。

【5】二元运算符和参数之间要有一个空格，如赋值号=左右各留一个空格。

|  |  |
| --- | --- |
| 1 | self.myString = @"235423rew523452345"; |

【6】一元运算符和参数之间不放置空格，比如！非运算符，&按位与，|按位或。

|  |  |
| --- | --- |
| 1  2 | BOOL isOpen = **true**;  BOOL isClose = !isOpen; |

【7】强制类型转换和参数之间不放置空格。

|  |  |
| --- | --- |
| 1 | NSString \*str3 = (NSString\*)self.myString; |

【8】长的变量值应该拆分为多行。尤其体现在使用数组或者字典。以下也分别是快速声明数组@[]和字典@{}的方法。

|  |  |
| --- | --- |
| 1  2  3  4  5 | NSArray \*array = @[@"111",                       @"2222222222",                       @"3333333",                       @"wwwwwwwwwwww"                       ]; |
| 1  2  3  4 | @"age":@"20",                         @"gender":@"female",                         @"isMarried":@"false"                         }; |

【9】尽量使用有意义的名字命名，拒绝使用i,j等无意义字符命名。类的命名首字母大写，其他变量的命名首字符小写，并使用驼峰式分割单词。

【10】尽量减少在代码中直接使用数字常量，而使用宏定义等方式。如：MAX\_NUMBER\_PHONE替代8等等。这样我们搜索也比较方便。

【11】尽量减少代码中的重复计算，比如代码中多处要使用屏幕宽度，然后计算：[[UIScreenmainScreen] bounds].size.width ,很多次，闲得很繁琐，代码也冗长。不如直接宏定义：

|  |  |
| --- | --- |
| 1 | #define SCREEN\_WIDTH ([[UIScreen mainScreen] bounds].size.width) |

【12】合理使用约定俗成的缩略词：

* alloc:分配；
* alt：轮流，交替；
* app:应用程序；
* calc：计算；
* dealloc：销毁、析构；
* func：函数、方法；
* horiz：水平的；
* info：信息；
* init：初始化；
* max：最大的；
* min：最小的；
* msg：消息；
* nib：Interface Builder；
* rect：矩形；
* temp：暂时的；
* vert：垂直的；

【13】宏定义全部字母大写。

【14】函数长度不要超过50行，小函数比大函数可读性更强。函数的参数不宜过多，零元函数最好，一元函数也不错，高于三元的函数虚重构。

【15】合理范围内使用链式编程:

|  |  |
| --- | --- |
| 1 | NSString \*myName = [[NSString alloc] init]; |

但是嵌套不宜超过3层，超过3层需进行重构。

【16】函数调用时所有参数在同一行。如果参数过多，则可以每行一个参数，每个参数以冒号对齐。

【17】对传入参数的保护或者说是否为空的判断，尽量不要使用if(!obj),而使用NSAssert断言来处理。NSAssert是系统定义的宏。

|  |  |
| --- | --- |
| 1 | NSAssert(myName != nil, @"myName参数为空"); |

* 如果条件判断为真，则程序继续执行。
* 如果判断条件为假，则抛出异常，异常内容为后面定义的字符串。

【18】方法参数名前一般使用"an","the","new"来进行修饰。如：

|  |  |
| --- | --- |
| 1 | -(void)setPersonInfo:(NSString\*)theID theName:(NSString\*)theName theAge:(NSInteger\*)theAge |

【19】if-else超过四层的时候，就要考虑重构，多层的if-else结构很难维护。

【20】当需要一定条件才执行某项操作时，最左边的应该是最重要的代码，不要将最重要的代码内嵌到if中。如良好的风格是：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | - (void) someMethod {  **if**(![someOther boolValue]) {  **return**;    }  //最重要的代码写在这里；  } |

反面教材:

|  |  |
| --- | --- |
| 1  2  3  4  5 | - (void) someMethod {  **if**([someOther boolValue]) {       //重要代码；    }  } |

【21】所有的逻辑块都使用{}花括号包围，就算只是一行代码。

【22】明确指定构造函数，并有适当的注释。

【23】不要在init方法中把变量或者说属性初始化为0或者nil，因为没有必要。

【24】UIView的子类初始化的时候，不要进行任何的布局操作。布局操作应该在layoutSubviews里面做；需要重新布局的时候调用setNeedsLayout，而不要直接调用layoutSubviews。

【25】保持公共API简单，也就是保持.h文件简单。放在.h中声明的函数都是会被公开的，如果根本就没必要对其他类公开，再不要在.h中声明。OC中的方法都是公有方法，没有私有方法一说。

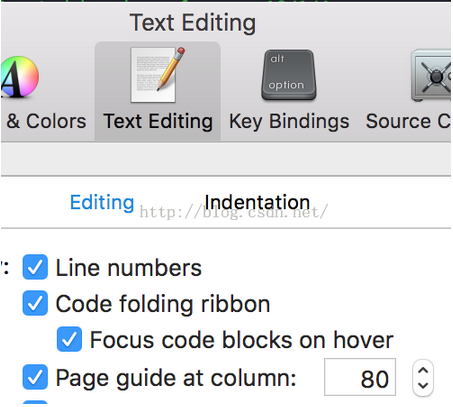
【26】一个文件只实现一个类。同一个文件中不要有多个类。

【27】Protocol单独用一个文件来创建，尽量不要与相关类混在一个文件中。

【28】在类定义中使用到自己定义类的时候，尽量不要在头文件中引入自己定义类的头文件，使用@class替代。而在实现文件中引入头文件。

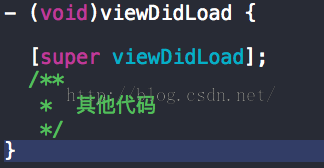
【29】布局时尽量使用相对布局，比如使用子View在父View中的相对位置。

【30】代码折叠，这个可能是关于开发效率的，我也写在编码规范中，因为这个很有用。Xcode7默认没有开启代码折叠，如果你的方法体行数很长，看起来会很不方便，此时你就可以把方法“收起来”，一个类中的结构就会很清晰。开启方法如下：Xcode菜单-->Preferences-->Text Editing-->勾选Code folding ribbon.如图：



【31】推荐方法的第一个花括号直接跟在方法体后，而不是另起一行，这样可以减少代码行。

【32】推荐方法体中的第一行留空，最后一行不留空，这样一个方法就会比较清晰。如图：



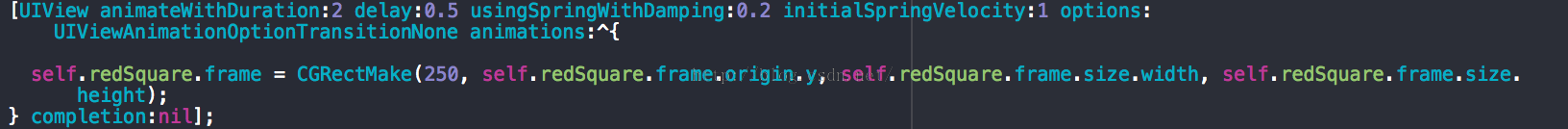
但是如果该花括号里面又是一个if，for之类的带花括号的语句块，那么上述的第一行可以不留空。

同样，如果花括号内第一行是注释的话，第一行也可以不留空。注释也起到了分隔代码的作用，看起来比较清晰。

再者，如果花括号内只有一行代码，第一行可以不留空。

【33】block中第一行也要留空，同方法体中的第一行留空，使代码清晰。

【34】代表类方法和实例方法的"+"加号,"-"减号后需要一个空格。这是一个非常小的细节，系统默认的方法都是这样的，我们自己声明或者实现一个方法的时候也需要这样：



【35】这一条有点像编程经验了，就是为解决某个问题估算时间。比如要开发某个功能、调试某个bug、给自己一个时间限制，如果在这期间不能解决问题，那么就去寻求帮助。这既是给自己一个压力，也为了不浪费时间。虽然，这一条其实很难做到，我往往由于不甘心而无限拖延时间去解决问题。

【36】由于提到编程经验，就不得不提到版本控制。务必去学会SVN或者Git，就算你是独立开发，也要学会控制自己的代码，当然，你要经常备份你的代码。

## 枚举的写法：

学习官方写法

# 网络部分

## 实现离线断点下载功能（2016年4月19日星期二）

### 基本描述：

实现大文件的离线重新下载，在程序运行过程可以暂停、继续以及在程序app下一次启动后，可以继续下载，上次下载的文件进行保存，下次从上次下载完成的部分继续下载。

基本的实际案例：百度音乐下载、喜马拉雅音乐下载、百度云文件的下载。

其实本人认为，网络部分断点续传算是难度和复杂度都比较高的部分。

### 解决方案

#### 解决方案理论

用户发送下载文件请求（如下载音乐、大文件等），当用户首次发送下载文件，将请求头部设置为请求的文件二进制位置为0，在下载过程中不断将下载的二进制文件存储在本地，并且记录下载量，一旦用户暂停下载，app记录下最后下载完成的数量以及下载请求的位置，从多少字节开始下载。服务器端则根据客户端的请求，从head区域的文件请求位置，开始传输二进制数据，发送给客户端，在完成总的文件下载后，客户端取消任何关于本文件的下载信息。

当前，一般的网络处理，都会用到AFNetworking 网络请求库，但是此框架对实现离线形式（重启app，从上次的下载点继续下载）的断点续传支持不好。因此我此处使用到的是apple系统api原生的网络请求方式。

#### 解决实践

1. 用到NSURLSession ->session，调用含有代理设置的初始化方法:

+ (NSURLSession \*)sessionWithConfiguration:(NSURLSessionConfiguration \*)configuration delegate:(nullable id <NSURLSessionDelegate>)delegate delegateQueue:(nullable NSOperationQueue \*)queue;

2. 使用NSMutableRequest->request，设置头部文件为：

//设置表头信息

NSString \*rang = [NSString stringWithFormat:”@bytes=%zd-”,currrentSize];

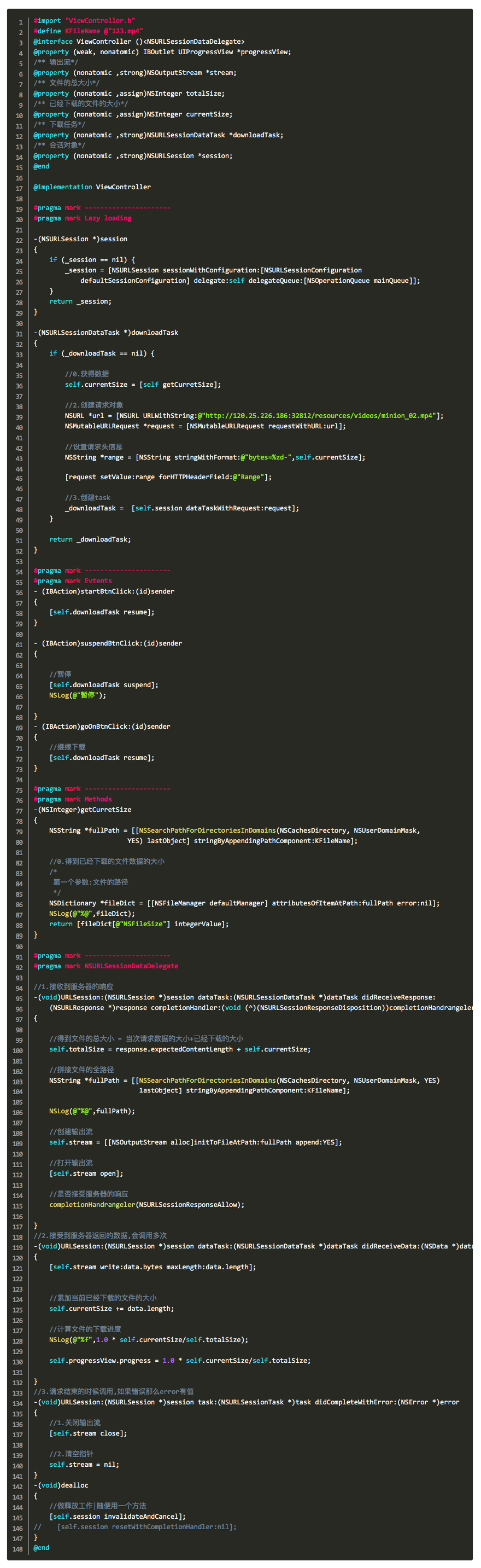
[request setValue:rang forHTTPHeanderField:”@Range”];

3.调用创建task方法

//创建task

NSURLSessionDataTask \*task = [session dataTaskWithRequest:request];

下图是参考别人写的，我觉的很好，直接贴上来，如果觉的图片不清楚，可以通过放大文档的形式查看，如下图：



# 多线程

## NSThread

是三种方法里面相对轻量级的，但需要管理线程的生命周期、同步、加锁问题，这会导致一定的性能开销

### 优缺点

1.优点：NSThread比其他两种多线程方案较轻量级，更直观地控制线程对象

2.缺点：需要自己管理线程的生命周期，线程同步。线程同步对数据的加锁会有一定的系统开销

## Cocoa Operations(NSOperationQueue)

是基于OC实现的，NSOperation以面向对象的方式封装了需要执行的操作，不必关心线程管理、同步等问题。NSOperation是一个抽象基类，iOS提供了两种默认实现：NSInvocationOperation和NSBlockOperation，当然也可以自定义NSOperation

### 注意点：

1. 运行中的任务不能够取消，只能够取消还没有执行的任务。
2. 分为两个队列，主线程队列和新线程队列。
3. 在NSOperationQueue中, 可以指定各个NSOperation之间的依赖关系.
4. 主队列只是串行执行
5. operation的executing和finished状态量需要用willChangeValueForKey/didChangeValueForKey来触发KVO消息.

### 与gcd对比

NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

GCD 只支持FIFO 的队列， 而NSOperationQueue可以调整队列的执行顺序。（通过调整权重）

NSOperationQueue可以在Operation间设置依赖关系，而GCD不可以。 如果一个Operation依赖另一个Operation所产生的数据的化，你可以设置一个Operation依赖于另一个Operation来实现， NSOperationQueue可以根据依赖关系，可以以正确的顺序执行Queue中的Operation。

NSOperationQueue支持KVO。 这就意味着你可以观察任务的状态属性。

以上几点并不是说在任何处理多任务时一定要选择NSOperationQueue， 因为NSOperationQueue在执行速度上会比GCD慢。

## GCD（grand gentral dispatch）

简称GCD，iOS4才开始支持)：提供了一些新特性、运行库来支持多核并行编程，它的关注点更高：如何在多个cpu上提升效率

/\*

### GCD的优势－－－－－－－－

GCD是苹果公司为多核的并行运算提出的解决方案

GCD会自动利用更多的CPU内核（比如双核、四核）

GCD会自动管理线程的生命周期（创建线程、调度任务、销毁线程）

程序员只需要告诉GCD想要执行什么任务，不需要编写任何线程管理代码

### 任务和队列－－－－－－－－－

GCD中有2个核心概念

（1）任务：执行什么操作

（2）队列：用来存放任务

队列－－－－－－－－－

GCD的队列分为2大类型

（1）并发队列（concurrent dispatch queue）

可以让多个任务并发（同时）执行（自动开启多个线程同时执行任务）并发功能，只有在异步(dispatch\_async)函数下才有效。

（2）串行队列 (serial dispatch queue)

让任务一个接一个执行

### 补充说明－－－－－－－－－

4个术语：同步 异步 并发 串行

同步和异步决定是不是要开启新线程

同步：在当前线程中执行，不具备开启新线程的能力

异步：在新线程中执行，具备开启新线程的能力

并发和串行决定任务的执行方式

并发：多个任务并发（同时）执行

串行：一个任务执行后，再执行下一个任务

### 串行队列

GCD中获得串行队列有2种途径

（1）使用dispatch\_queue\_create 函数创建

dispatch\_queue\_t dispatch\_queue\_create(const char \*label, dispatch\_queue\_attr\_t attr); // 队列名称， 队列属性，一般用NULL即可

示例：

dispatch\_queue\_t queue = dispatch\_queue\_create("wendingding", NULL); // 创建

dispatch\_release(queue); // 非ARC需要释放手动创建的队列

（2）使用主队列（跟主线程相关联的队列）

主队列是GCD自带的一种特殊的串行队列,放在主队列中的任务，都会放到主线程中执行

使用dispatch\_get\_main\_queue()获得主队列

示例：

dispatch\_queue\_t queue = dispatch\_get\_main\_queue();

### 并发队列－－－－－－－

GCD默认已经提供了全局的并发队列，供整个应用使用，不需要手动创建

使用dispatch\_get\_global\_queue函数获得全局的并发队列

dispatch\_queue\_t dispatch\_get\_global\_queue(dispatch\_queue\_priority\_t priority,unsigned long flags); // 此参数暂时无用，用0即可

示例：

这个参数是留给以后用的，暂时用不上，传个0。

第一个参数为优先级，这里选择默认的。获取一个全局的默认优先级的并发队列。

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0); // 获得全局并发队列

说明：全局并发队列的优先级

#define DISPATCH\_QUEUE\_PRIORITY\_HIGH 2 // 高

#define DISPATCH\_QUEUE\_PRIORITY\_DEFAULT 0 // 默认（中）

#define DISPATCH\_QUEUE\_PRIORITY\_LOW (-2) // 低

#define DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND INT16\_MIN // 后台

### 各种队列执行的效果

### 小结

说明：同步函数不具备开启线程的能力，无论是什么队列都不会开启线程；异步函数具备开启线程的能力，开启几条线程由队列决定（串行队列只会开启一条新的线程，并发队列会开启多条线程）。

同步函数

（1）并发队列：不会开线程

（2）串行队列：不会开线程

异步函数

（1）并发队列：能开启N条线程

（2）串行队列：开启1条线程

### 补充：

凡是函数中，各种函数名中带有create\copy\new\retain等字眼，都需要在不需要使用这个数据的时候进行release。

GCD的数据类型在ARC的环境下不需要再做release。

CF（core Foundation）的数据类型在ARC环境下还是需要做release。

异步函数具备开线程的能力，但不一定会开线程

### GCD调度组

有这么一个需要，分别执行2个耗时的异步操作，等2个异步操作都执行完毕后在回到主线程执行操作。如果想要快速高效地实现这个需求，可以考虑使用调度组。 调度组的创建代码如下：

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_group\_async(dispatch\_group\_t group, dispatch\_queue\_t queue, ^(void)block)

第一个参数group是一个调度组，queue是一个异步队列，这里如果使用同步队列意义就不大了，不过还是能达到效果，第三个参数是一个执行代码块，任务都添加到该代码块里面。 往调度组里面添加任务并获得通知的代码如下

// 群组－统一监控一组任务

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_queue\_t q = dispatch\_get\_global\_queue(0, 0);

// 添加任务

// group 负责监控任务，queue 负责调度任务

dispatch\_group\_async(group, q, ^{

[NSThread sleepForTimeInterval:1.0];

NSLog(@"任务1 %@", [NSThread currentThread]);

});

dispatch\_group\_async(group, q, ^{

NSLog(@"任务2 %@", [NSThread currentThread]);

});

dispatch\_group\_async(group, q, ^{

NSLog(@"任务3 %@", [NSThread currentThread]);

});

// 监听所有任务完成 － 等到 group 中的所有任务执行完毕后，"由队列调度 block 中的任务异步执行！"

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

// 修改为主队列，后台批量下载，结束后，主线程统一更新UI

NSLog(@"OK %@", [NSThread currentThread]);

});

NSLog(@"come here");

GCD调度组中还有一个添加任务的函数： dispatch\_group\_enter(dispatch\_group\_t group) dispatch\_group\_leave(dispatch\_group\_t group) 这两个函数要配对出现 例如：

// 群组－统一监控一组任务

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_queue\_t q = dispatch\_get\_global\_queue(0, 0);

// 1> 入组 -> 之后的 block 会被 group 监听

// dispatch\_group\_enter 一定和 dispatch\_group\_leave 要配对出现

dispatch\_group\_enter(group);

dispatch\_async(q, ^{

NSLog(@"task1 %@", [NSThread currentThread]);

// block 的末尾，所有任务执行完毕后，添加一个出组

dispatch\_group\_leave(group);

});

// 再次入组

dispatch\_group\_enter(group);

dispatch\_async(q, ^{

[NSThread sleepForTimeInterval:1.0];

NSLog(@"task2 %@", [NSThread currentThread]);

// block 的末尾，所有任务执行完毕后，添加一个出组

dispatch\_group\_leave(group);

});

// 群组结束

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

NSLog(@"OVER");

});

NSLog(@"come here");

代码执行的效果与刚才那种方式相同！ GCD中还有一个函数

dispatch\_group\_wait(dispatch\_group\_t group, dispatch\_time\_t timeout)

该函数是个阻塞式的等待，当我们将第二个参数设置为DISPATCH\_TIME\_FOREVER时，群组任务不执行完，后续代码就会无法执行，也就是说只要任务不执行完come here就不会打印。

### dispatch\_apply

执行某个代码片段若干次。

dispatch\_apply(10, globalQ, ^(size\_t index) {

// do sth. 10 times

});

### dispatch\_once

它可以保证整个应用程序生命周期中某段代码只被执行一次！

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

// code to be executed once

});

举例：

＋(MyInstance\*)shareMyInstance（）

｛

static MyInstance \*inst=nil;

static dispatch\_once\_t p;

dispatch\_once(&p,^{

inst=[[MyInstance alloc]init];

});

return inst;

｝

### dispatch\_after

有时候我们需要等个几秒钟然后做个动画或者给个提示，这时候可以用dispatch\_after这个函数：

double delayInSeconds = 2.0;

dispatch\_time\_t popTime = dispatch\_time(DISPATCH\_TIME\_NOW, delayInSeconds \* NSEC\_PER\_SEC);

dispatch\_after(popTime, dispatch\_get\_main\_queue(), ^(void){

// code to be executed on the main queue after delay

});

### dispatch\_time

或者说同步地等待一段时间看是否结束：

dispatch\_time\_t time = dispatch\_time(DISPATCH\_TIME\_NOW, 1ull \* NSEC\_PER\_SEC);

dispatch\_group\_wait(group, time);

### dispatch\_barrier\_async

通过dispatch\_barrier\_async函数提交的任务会等它前面的任务执行结束才开始，然后它后面的任务必须等它执行完毕才能开始。

dispatch\_async(concurrentQ, blk0);

dispatch\_async(concurrentQ, blk1);

dispatch\_barrier\_async(concurrentQ, blk\_barrier);

dispatch\_async(concurrentQ, blk2);

\*/

## 同步锁（NSLock）

## @ synchronized

使用方法；

在防止多个线程访问的对象和代码处使用如下：

@synchronized (\_elements) {

[\_elements addObject:element];

}

”synchronized block” 与 [\_lock lock] 和 [\_lock unlock] 效果相同。你可以把它当成是锁住 self，仿佛 self 就是个 NSLock。锁在左括号 { 后面的任何代码运行之前被获取到，在右括号 } 后面的任何代码运行之前被释放掉。这爽就爽在妈妈再也不用担心我忘记调用 unlock 了！

其实上面的\_elements可以设置为当前的类对象，self，关键是通过@synchronized关键字会将过好内的代码进行加锁，等代码执行完成后才允许其他线程访问这些代码。

# NSRunLoop(较难)

暂放

# Block

## Block简介

Block是一种比较特殊的数据类型。它可以保存一段代码，在合适的时候取出来调用。

## Block的定义

Block的定义格式

返回值类型(^block变量名)(形参列表) = ^(形参列表) {

};

调用Block保存的代码

block变量名(实参);

默认情况下,Block内部不能修改外面的局部变量

Block内部可以修改使用\_\_block修饰的局部变量

Block的模式

1.无参数无返回值的Block

2.有参数无返回值的Block

3.有参数有返回值的Block

## Block的修饰

ARC情况下

1.如果用copy修饰Block，该Block就会存储在堆空间。则会对Block的内部对象进行强引用，导致循环引用。内存无法释放。

解决方法：

新建一个指针(\_\_weak typeof(Target) weakTarget = Target )指向Block代码块里的对象，然后用weakTarget进行操作。就可以解决循环引用问题。

2.如果用weak修饰Block，该Block就会存放在栈空间。不会出现循环引用问题。

MRC情况下

用copy修饰后，如果要在Block内部使用对象，则需要进行(\_\_block typeof(Target) blockTarget = Target )处理。在Block里面用blockTarget进行操作。

## Block简单用法举例

### 无参数无返回值的Block

/\*\*

\* 无参数无返回值的Block

\*/

-(void)func1{

/\*\*

\* void ：就是无返回值

\* emptyBlock：就是该block的名字

\* ()：这里相当于放参数。由于这里是无参数，所以就什么都不写

\*/

void (^emptyBlock)() = ^(){

NSLog(@"无参数,无返回值的Block");

};

emptyBlock();

}

### 有参数无返回值的Block

/\*\*

\* 调用这个block进行两个参数相加

\*

\* @param int 参数A

\* @param int 参数B

\*

\* @return 无返回值

\*/

void (^sumBlock)(int ,int ) = ^(int a,int b){

NSLog(@"%d + %d = %d",a,b,a+b);

};

/\*\*

\* 调用这个sumBlock的Block，得到的结果是20

\*/

sumBlock(10,10);

### 有参数有返回值的Block

/\*\*

\* 有参数有返回值

\*

\* @param NSString 字符串1

\* @param NSString 字符串2

\*

\* @return 返回拼接好的字符串3

\*/

NSString\* (^logBlock)(NSString \*,NSString \*) = ^(NSString \* str1,NSString \*str2){

return [NSString stringWithFormat:@"%@%@",str1,str2];

};

//调用logBlock,输出的是 我是Block

NSLog(@"%@", logBlock(@"我是",@"Block"));

### Block结合typedef使用

自己定义一个Block类型，用定义的类型去创建Block，更加简单便捷。  
这里举例一个Block回调修改上一下界面的背景颜色。  
ViewController1 控制器1，ViewController2 控制器2  
控制器1跳转到控制器2，然后在控制器2触发事件回调修改控制器1的背景颜色为红色。

ViewController2的实现

#import <UIKit/UIKit.h> @interface ViewController2 : UIViewController /\*\* \* 定义了一个changeColor的Block。这个changeColor必须带一个参数，这个参数的类型必须为id类型的 \* 无返回值 \* @param id \*/ typedef void(^changeColor)(id); /\*\* \* 用上面定义的changeColor声明一个Block,声明的这个Block必须遵守声明的要求。 \*/ @property (nonatomic, copy) changeColor backgroundColor; @end

-(void)touchesBegan:(NSSet<UITouch \*> \*)touches withEvent:(UIEvent \*)event{ //声明一个颜色 UIColor \*color = [UIColor redColor]; //用刚刚声明的那个Block去回调修改上一界面的背景色 self.backgroundColor(color); }

ViewController1的实现

-(void)touchesBegan:(NSSet<UITouch \*> \*)touches withEvent:(UIEvent \*)event{ ViewController2 \*vc =[[ViewController2 alloc]init]; // 回调修改颜色 vc.backgroundColor = ^(UIColor \*color){ self.view.backgroundColor = color; }; [self.navigationController pushViewController:vc animated:YES]; }

# 内存管理

## 检测内存泄漏

**Xcode结合Leaks检测内存泄露**

字数329 阅读503 评论1 喜欢18

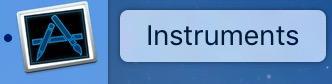
Collection/Bookmark/Share for width under 768px

**内存很重要！内存很重要！内存很重要！**

重要的事情要说三遍。虽然现在项目大多数是ARC模式，Xcode会帮我们管理内存，但并不意味着这样就不会存在内存泄露的现象。Xcode里面有一个工具Instruments，这个工具可以检测项目的性能。

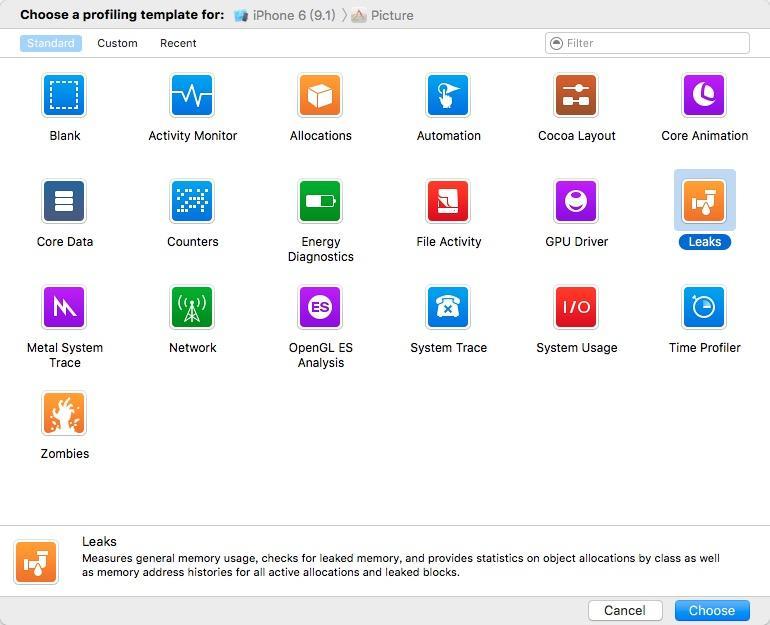
**Xcode结合Leaks检测内存泄露代码**

1.command+i 开启 Instruments工具



启动.png

2.选择Leaks，点choose



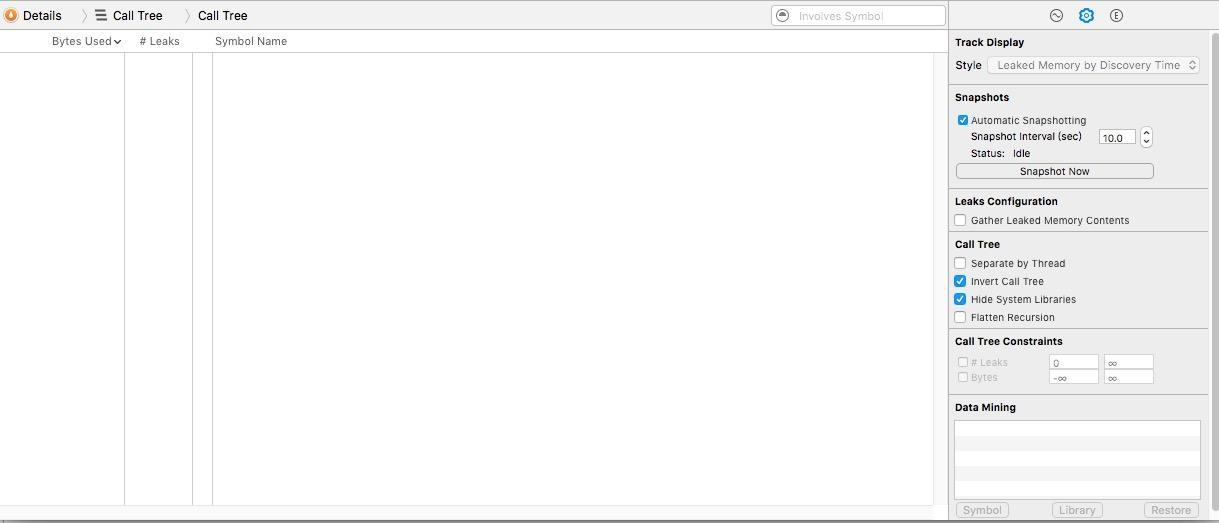
Leaks.png

3.选择Leaks，然后在Leaks那里选择**Call Tree**



xxxx.png

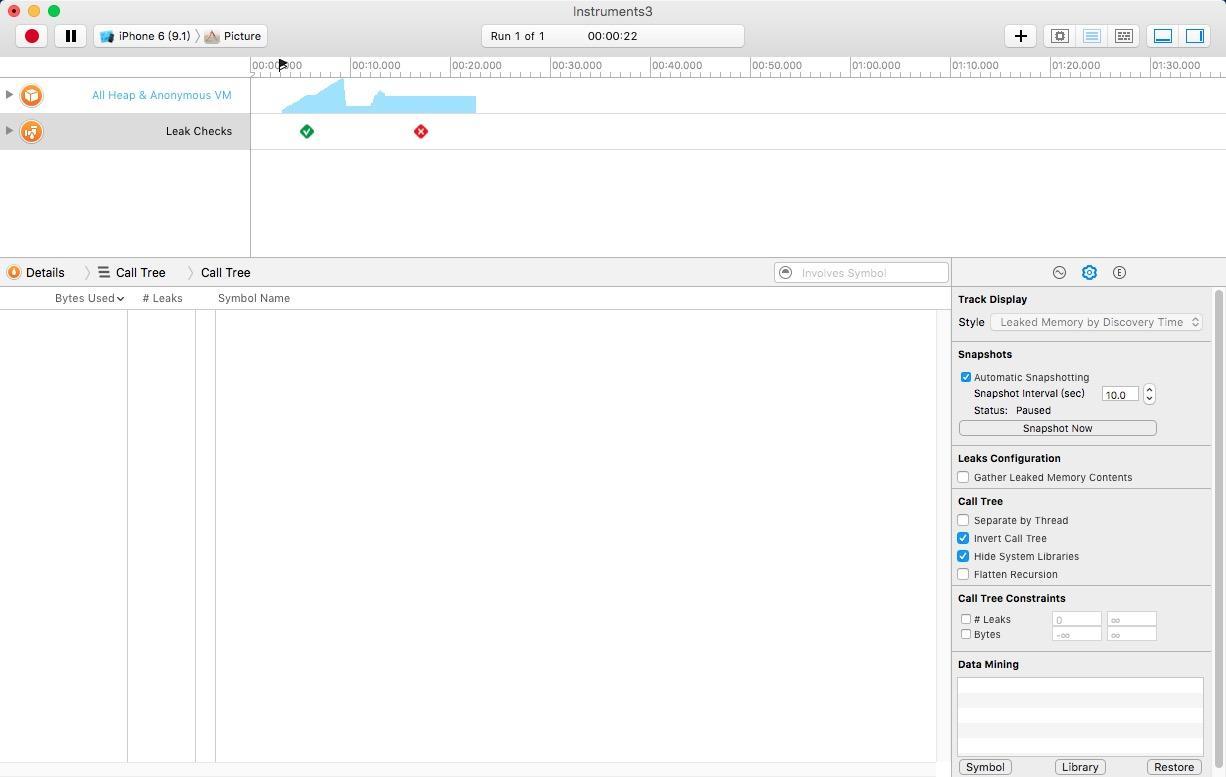
4.选择好Call Tree 之后选择设置，长得和齿轮一样



xxxxx.png

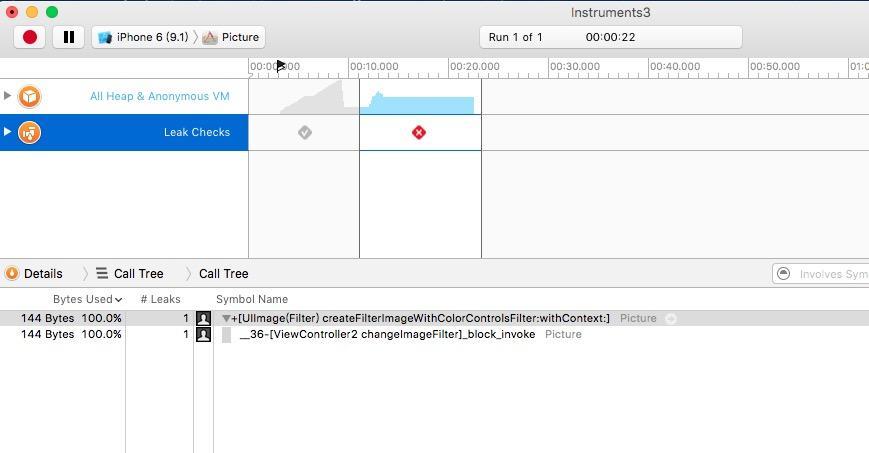
把Invert Call Tree 和 Hide system Libraries勾选

5.最后点左上角的红点Record，你就会发现模拟器启动了，然后随便你操作app。如果出现了红色的叉叉，就说明存在内存泄露。



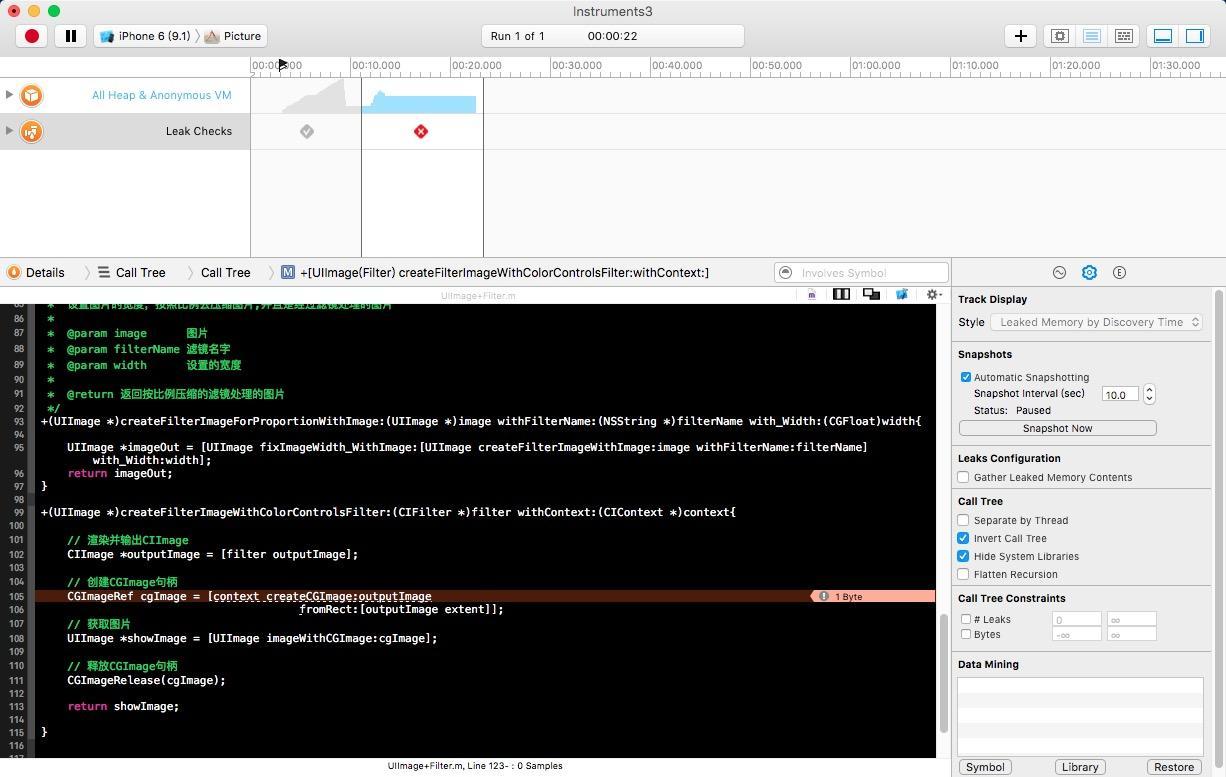
Bug.png

6.选中红色叉叉的区域,在Details里面就可以知道泄露的代码具体在哪一行。双击那段代码就可以去到泄露的位置。



findBug.png

7.找到泄露的代码，也可以一直点进去看。



Bug11.png

8.结束啦，找到泄露的地方，那就改去咯。

##### 内存管理

堆：需要手动释放。  
栈：系统自动管理。

##### 所谓内存管理,就是对内存进行管理,涉及的操作有:

分配内存 :比如创建一个对象,会增加内存占用  
清除内存 :比如销毁一个对象,能减小内存占用

##### 内存管理的管理范围

任何继承了NSObject的对象  
对其他非对象类型无效(int、char、float、double、struct、enum等)

##### 只有OC对象才需要进行内存管理的本质原因

OC对象存放于堆里面  
OC基础类型一般放在栈里面(栈内存会被系统自动回收)

##### 引用计数器的常见操作

给对象发送一条retain消息,可以使引用计数器值+1（retain方法返回对象本身）  
给对象发送一条release消息,可以使引用计数器值-1  
给对象发送retainCount消息,可以获得当前的引用计数器值

##### 需要注意的是: release并不代表销毁\回收对象,仅仅是计数器-1。

##### 空指针\野指针

僵尸对象  
已经被销毁的对象(不能再使用的对象)

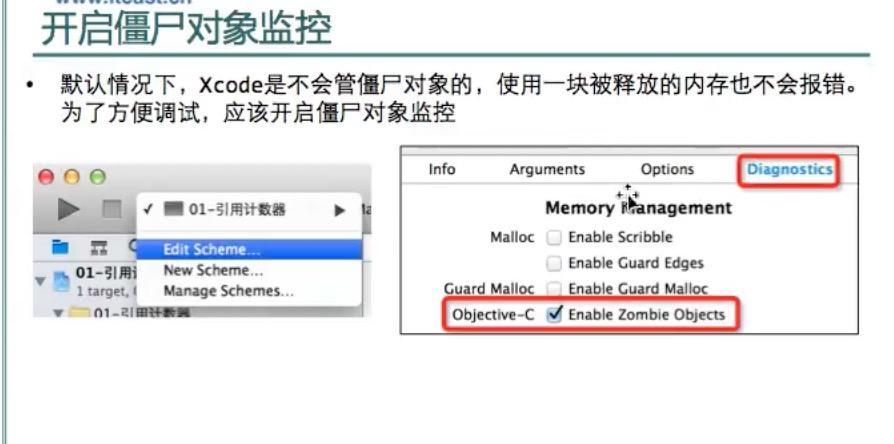
野指针  
指向僵尸对象(不可用内存)的指针  
给野指针发消息会报**EXC\_BAD\_ACCESS**错误

空指针  
没有指向存储空间的指针(里面存的是nil,也就是0)  
给空指针发消息是没有任何反应的。  
为了避免野指针错误的常见办法  
在对象被销毁之后,将指向对象的指针变为空指针

##### delloc方法注意细节

**当一个对象的引用计数器值为0时**  
这个对象即将被销毁，其占用的内存被系统回收  
系统会自动给对象发送一条dealloc消息  
(因此,从dealloc方法有没有被调用,就可以判断出对象是否被销毁)  
**dealloc方法的重写**  
一般会重写dealloc方法,在这里释放相关资源,dealloc就是对象的遗言  
一旦重写了dealloc方法,就必须调用[super dealloc],并且放在最后面调用  
使用注意  
**不能直接调用dealloc方法**  
一旦对象被回收了,它占用的内存就不再可用,坚持使用会导致程序崩溃（野指针错误）

##### Xcode开启僵尸对象的监控



截图1.png

##### 苹果官方规定的内存管理原则

谁创建谁release:如果你通过alloc、new或[mutable]copy来创建一个对象，那么你必须调用release或autorelease。  
谁retain谁release:只要你调用了retain，就必须调用一次release。  
总结一下就是  
**有加就有减**  
**曾经让对象的计数器+1，就必须在最后让对象计数器-1**

##### set方法的内存管理

- (void)setCar:(Car\*)car { if(car!= \_car) { // 对当前正在使用的车（旧车）做一次release [\_car release]; // 对新车做一次retain操作 \_car=[car retain]; } }

##### dealloc方法内存管理

- (void)dealloc { //当人不在了，代表不用车了 // 对车做一次release操作 [\_car release]; //调用dealloc方法时候一定要调用父类的dealloc的方法 [super dealloc]; }

##### @property参数修饰

##### 控制set方法的内存管理

retain：release旧值，retain新值（用于OC对象）  
assign：直接赋值，不做任何内存管理(默认，用于非OC对象类型)  
copy ：release旧值，copy新值（一般用于NSString\*,block）

##### 控制需不需生成set方法

readwrite：同时生成set方法和get方法（默认）  
readonly ：只会生成get方法

##### 多线程管理

atomic ：性能低（默认）  
nonatomic：性能高

##### 控制set方法和get方法的名称

setter：设置set方法的名称，一定有个冒号:  
getter：设置get方法的名称

##### 强引用strong与弱引用weak的广义区别：

强引用**strong**也就是我们通常所讲的引用，其存亡直接决定了所指对象的存亡。如果不存在指向一个对象的引用，并且此对象不再显示列表中，则此对象会被从内存中释放。  
弱引用**weak**除了不决定对象的存亡外，其他与强引用相同。即使一个对象被持有无数个若引用，只要没有强引用指向他，那麽其还是会被清除。没办法，还是 “强哥” 有面子。

**简单讲strong等同retain**  
ARC时候用strong,MRC时候用return  
weak比assign多了一个功能，当对象消失后自动把指针变成nil，好处不言而喻。所以，我觉得在delegate时候用weak会好过assign。

##### 至于block的话，就要结合考虑。（weak，copy，assign）这三个结合情况，在下一篇的介绍。

**\_\_weak** ,**\_\_strong** ,**\_\_unsafe\_unretained**, **\_\_autoreleasing**用来修饰变量.  
(\_\_strong) 是缺省的关键词。  
(\_\_weak) 声明了一个可以自动 nil 化的弱引用。  
(\_\_unsafe\_unretained) 声明一个弱应用，但是不会自动nil化，也就是说，如果所指向的内存区域被释放了，这个指针就是一个野指针了。  
(\_\_autoreleasing) 用来修饰一个函数的参数，这个参数会在函数返回的时候被自动释放。

###### @class和#import的区别

作用上的区别  
**#import**会包含引用类的所有信息(内容), 包括引用类的变量和方法  
**@class**仅仅是告诉编译器有这么一个类,具体这个类里有什么信息, 完全不知道。

效率上的区别  
如果有上百个头文件都#import了同一个文件，或者这些文件依次被#import,那么一旦最开始的头文件稍有改动，后面引用到这个文件的所有类都需要重新编译一遍,编译效率非常低  
相对来讲，使用@class方式就不会出现这种问题了

##### MRC循环retian

循环retain的场景  
比如A对象retain了B对象，B对象retain了A对象

循环retain的弊端  
这样会导致A对象和B对象永远无法释放

循环retain的解决方案  
当两端互相引用时，应该一端用retain、一端用assign

**同理，在ARC下，如果两个对象相互引用，并且用strong修饰，那两个对象都无法释放。解决方案：其中一个对象用weak修饰，一个用strong修饰。那就可以让两个对象都可以释放。**

##### nil Nil NULL NSNull 之间的区别

**nil** 是对objective c id 对象赋空值  
**Nil**: 表示对类进行赋空值  
**NULL**: 用于对非对象指针赋空值，比如C指针  
**NSNull** 对于像NSArray这样的类型，nil或NULL不能做为加到其中的Object，如果定义了一个NSArray，为其分配了内存，又想设置其中的内容为空，则可以用[NSNULL null】返回的对象来初始化NSArray中的内容

# 数据结构

# 设计模式

## 设计原则

### 单一职责原则

一个类只负责一个功能领域中相应的职责

### 开闭原则

一个类或者软件实体尽量实现对扩展开发，对修改关闭

### 里氏代换

所有引用基类（父类）的地方，都能够使用子类的对象

### 依赖倒转

具体依赖抽象，而不是抽象依赖具体。

### 接口隔离

### 合成复用

### 迪米特法则

## 创建型

# 行为型

## 策略模式

### 组成成分：

1. 抽象策略类
2. 具体策略类
3. 策略上下文

# 牛人技术博客地址

1. 苹果源码<http://opensource.apple.com/source/>
2. <http://www.henishuo.com/ios-block-memory-cycle/?utm_source=tuicool&utm_medium=referral>
3. <http://www.cnblogs.com/mjios/p/4409853.html>
4. <https://blog.cnbluebox.com/blog/2014/07/01/cocoashen-ru-xue-xi-nsoperationqueuehe-nsoperationyuan-li-he-shi-yong/>
5. <http://www.jianshu.com/p/c47c24ab1e76>
6. <http://weibo.com/5612984599/profile?rightmod=1&wvr=6&mod=personinfo&is_hot=1#_loginLayer_1464445659010>
7. <http://casatwy.com/zen-yao-mian-shi-jia-gou-shi.html>
8. <http://blog.cnbluebox.com>
9. <http://my.oschina.net/u/566401/blog/219568>
10. <http://blog.sunnyxx.com/archives/page/3/>
11. <http://my.oschina.net/caijunrong/blog/512372>
12. <http://www.jianshu.com/users/256fb15baf75/latest_articles>
13. <http://www.jianshu.com/p/8f16613861fa>
14. <http://www.jianshu.com/users/0a7ce85a2708/latest_articles>
15. <http://www.jianshu.com/p/2113ffe54b30>
16. <http://www.jianshu.com/collection/6930cb8ad81b>
17. <http://www.cnblogs.com/zy1987/>

# 面试题集锦

1. iOS面试必看，最全梳理<http://www.cocoachina.com/ios/20160323/15770.html>