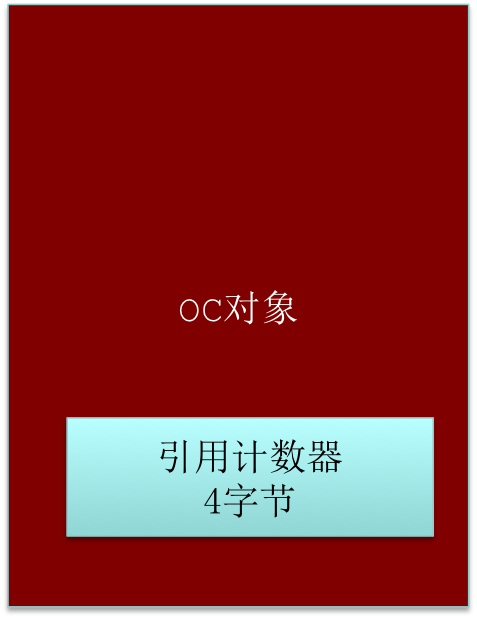
# 基本原理

## 什么是内存管理

* 移动设备的内存极其有限，每个app所能占用的内存是有限制的
* 当app所占用的内存较多时，系统会发出内存警告，这时得回收一些不需要再使用的内存空间。比如回收一些不需要使用的对象、变量等
* 管理范围：任何继承了NSObject的对象，对其他基本数据类型（int、char、float、double、struct、enum等）无效

## 对象的基本结构



* 每个OC对象都有自己的引用计数器，是一个整数，表示“对象被引用的次数”，即有多少人正在使用这个OC对象
* 每个OC对象内部专门有4个字节的存储空间来存储引用计数器

## 引用计数器的作用

* 当使用alloc、new或者copy创建一个新对象时，新对象的**引用计数器**默认就是1
* 当一个对象的引用计数器值为0时，对象占用的内存就会被系统回收。换句话说，如果对象的计数器不为0，那么在整个程序运行过程，它占用的内存就不可能被回收，除非整个程序已经退出

## 引用计数器的操作

* 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**
* 可以给对象发送retainCount消息获得当前的引用计数器值

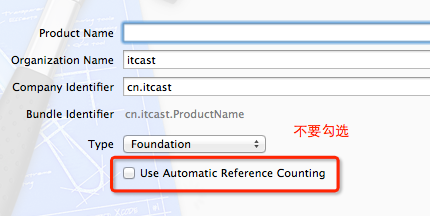
## 对象的销毁

* 当一个对象的引用计数器值为0时，那么它将被销毁，其占用的内存被系统回收
* 当一个对象被销毁时，系统会自动向对象发送一条dealloc消息
* 一般会重写dealloc方法，在这里释放相关资源，dealloc就像对象的遗言
* 一旦重写了dealloc方法，就必须调用[super dealloc]，并且放在最后面调用
* **不要直接调用**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的设置

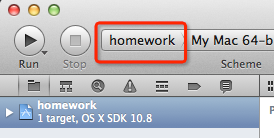
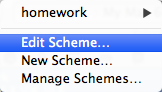
## 取消ARC

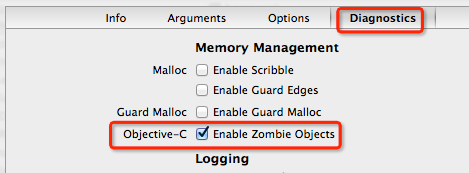
要想手动调用retain、release等方法，在创建项目的时候不要勾选ARC



## 开启僵尸对象监控

默认情况下，Xcode是不会管僵尸对象的，使用一块被释放的内存也不会报错。为了方便调试，应该开启僵尸对象监控



# 内存管理原则

## 原则分析

* QQ堂开房间原理：只要房间还有人在用，就不会解散



* 只要还有人在用某个对象，那么这个对象就不会被回收
* 只要你想用这个对象，就让对象的计数器+1
* 当你不再使用这个对象时，就让对象的计数器-1



## 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
* 换句话说，不是你创建的，就不用你去[auto]release

## 谁retain，谁release

* 只要你调用了retain，无论这个对象是如何生成的，你都要调用release

## 总结

* 有始有终，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

如果你有个OC对象类型的成员变量，就必须管理这个成员变量的内存。比如有个Book \*\_book

## set方法的实现

- (void)setBook:(Book \*)book{

if (book != \_book) {

[\_book release];

\_book = [book retain];

}

}

## dealloc方法的实现

- (void)dealloc {

[\_book release];

[super dealloc];

}

# @property参数

## 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 \*）

## 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

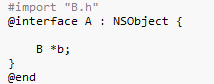
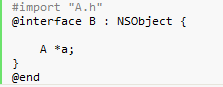
* setter ： 设置set方法的名称，一定有个冒号:
* getter ： 设置get方法的名称

# 循环引用

## @class

* 使用场景

对于循环依赖关系来说，比方A类引用B类，同时B类也引用A类

这种代码编译会报错。当使用@class在两个类相互声明，就不会出现编译报错

* 用法概括

使用 @class 类名; 就可以引用一个类，说明一下它是一个类

* 和#import的区别
* #import方式会包含被引用类的所有信息，包括被引用类的变量和方法；@class方式只是告诉编译器在A.h文件中 B \*b 只是类的声明，具体这个类里有什么信息，这里不需要知道，等实现文件中真正要用到时，才会真正去查看B类中信息
* 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#improt,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，这样的效率也是可想而知的，而相对来 讲，使用@class方式就不会出现这种问题了
* 在.m实现文件中，如果需要引用到被引用类的实体变量或者方法时，还需要使用#import方式引入被引用类

## 循环retain

* 比如A对象retain了B对象，B对象retain了A对象
* 这样会导致A对象和B对象永远无法释放

## 解决方案

* 当两端互相引用时，应该一端用retain、一端用assign

# autorelease

## autorelease

* 给某个对象发送一条autorelease消息时，就会将这个对象加到一个自动释放池中
* 当自动释放池销毁时，会给池子里面的所有对象发送一条release消息
* 调用autorelease方法时并不会改变对象的计数器，并且会返回对象本身
* autorelease实际上只是把对release的调用延迟了，对于每一次autorelease，系统只是把该对象放入了当前的autorelease pool中，当该pool被释放时，该pool中的所有对象会被调用Release

## 自动释放池的创建

* ios 5.0后

@autoreleasepool

{

// ....

}

* ios 5.0前

NSAutoreleasePool \*pool = [[NSAutoreleasePool alloc] init];

// .....

[pool release]; // 或[pool drain];

* 在程序运行过程中，可以创建多个自动释放池，它们是以栈的形式存在内存中
* OC对象只需要发送一条autorelease消息,就会把这个对象添加到最近的自动释放池中(栈顶的释放池)

## 应用实例

* 跟release的对比
* 以前:

Book \*book = [[Book alloc] init];

[book release];

* 现在:

Book \*book = [[[Book alloc] init] autorelease];

// 不要再调用[book release];

* 一般可以为类添加一个快速创建对象的类方法

+ (id)book {

return [[[Book alloc] init] autorelease];

}

外界调用[Book book]时，根本不用考虑在什么时候释放返回的Book对象

## 规律

* 一般来说,除了alloc、new或copy之外的方法创建的对象都被声明了autorelease
* 比如下面的对象都已经是autorelease的，不需要再release

NSNumber \*n = [NSNumber numberWithInt:100];

NSString \*s = [NSString stringWithFormat:@"jack"];

NSString \*s2 = @"rose";