

# block产生的内存泄漏以及解决方案（以及扩展）



作者 JoanKing (/u/8fed18ed70c9) +关注

2016.07.12 00:04 字数 3347 阅读 1410 评论 2 喜欢 31

(/u/8fed18ed70c9)

前言：

在ARC（自动引用技术）前，Objective-c都是手动来分配释放 释放 计数内存，其过程非常复杂。

ARC技术推出后，貌似世界和平了很多，但是其实ARC并不等同于Java或者C#中的垃圾回收，ARC计数只是在XCode在编译的时候自动帮我们加上了释放 计数+1 计数-1.

内存泄露例子：

然而在一些特殊的情况下，内存泄露依然存在，而且防不慎防，这里讲一下Objective-C中Block计数是如何产生内存泄露的，如下代码

.h中

```
typedef void (^CompletionBlock)(NSString *aStr);
@interface B : NSObject
@property (copy) CompletionBlock completionBlock;
@property (copy) NSString *str;
@end
```

.m中

```
@implementation B
-(id)init{
    self = [super init];
    if(self){
        self.str = @"init string value";
    }
    return self;
}

-(void)doAction
{
    __block B *b1 = self;
    self.completionBlock = ^(NSString *aStr){
        b1.str = aStr;
    };
    self.completionBlock(@"new string value");
}

-(void)dealloc{
    NSLog(@"dealloc B");
}
@end
```

main函数中

```
B *b = [[B alloc] init];
[b doAction];
b = nil;//这句有和无其实无所谓
```



上面的程序看似没有问题，但是实际上对象b永远无法释放，原因在于doAction函数，这个函数里面有一个block函数名为completionBlock，也就是一个函数指针。这个函数指针在调用的时候有使用一个对象，也就是self对象。但是这个block隐形的做了一件事情——将self引用计数+1了，因此这个时候self对象（也就是main函数中的b对象）的引用计数是2，这个时候即使我执行了b=nil，也无法释放，因为b=nil只是将计数减1了，而真正释放的唯一条件是引用计数为0。这就是所谓的Block的循环引用。

### 如何解决：

所以在使用block技术的时候，需要格外小心。有几个解决方法

approach 1: 让block里面的self的引用计数不要+1，这个时候做法是将" **block B \*b1 = self;**"这一行改为，" weak \_\_block B \*b1 = self;"，表示说“我block里面虽然会用到self，但是别担心，我不会讲引用计数+1的”

approach 2: 在doAction函数内存的最后一行添加 self.completionBlock=nil；因为block内部将self计数+1了，但是如果这个block自己先消亡，那么与之相关的一切都讲消亡（当然对于引用计数大于1的对象，不会消亡，只会计数减1）。

### 附加：

**PS：**开发中，几乎每个.m文件都会用到block技术，但是从未发现和在意这个内存泄露问题，这并不是XCode编译时的优化，而是我们所用到的Block技术（例如AFNetwork GCD Animation）中的block都是匿名Block——即，用完自动释放。如果有一天不用匿名block就需要注意这个问题了。

例如下面的例子中，虽然使用了Block，但是没有泄露，是因为这是一个匿名的Block（即匿名函数指针）

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(1 * NSEC_PER_SEC)), dispatch_queue_t) {
    self.view.backgroundColor = [UIColor redColor];
};
```

## 2.关于block的扩展总结

Block简介（copy一段）

Block作为C语言的扩展，并不是高新技术，和其他语言的闭包或lambda表达式是一回事。需要注意的是由于Objective-C在iOS中不支持GC机制，使用Block必须自己管理内存，而内存管理正是使用Block坑最多的地方，错误的内存管理 要么导致return cycle内存泄漏要么内存被提前释放导致crash。Block的使用很像函数指针，不过与函数最大的不同是：Block可以访问函数以外、词法作用域以外的外部变量的值。换句话说，Block不仅实现函数的功能，还能携带函数的执行环境。

可以这样理解，Block其实包含两个部分内容

Block执行的代码，这是在编译的时候已经生成好的；

一个包含Block执行时需要的所有外部变量值的数据结构。Block将使用到的、作用域附近到的变量的值建立一份快照拷贝到栈上。

Block与函数另一个不同是，Block类似ObjC的对象，可以使用自动释放池管理内存（但Block并不完全等同于ObjC对象，后面将详细说明）。

### 2.2Block基本语法

基本语法在本文就不赘述了，同学们自学。

### 2.3Block的类型与内存管理



根据Block在内存中的位置分为三种类型NSGlobalBlock, NSStackBlock, NSMallocBlock。

NSGlobalBlock：类似函数，位于text段；

NSStackBlock：位于栈内存，函数返回后Block将无效；

NSMallocBlock：位于堆内存。

**1、NSGlobalBlock如下，我们可以通过是否引用外部变量识别，未引用外部变量即为NSGlobalBlock，可以当做函数使用。**

```
{
//create a NSGlobalBlock
float (^sum)(float, float) = ^(float a, float b){

    return a + b;
};

NSLog(@"block is %@", sum); //block is <__NSGlobalBlock__: 0x47d0>
}
```

**2、NSStackBlock如下：**

```
{
NSArray *testArr = @"1", @"2";

void (^TestBlock)(void) = ^{

    NSLog(@"testArr :%@", testArr);
};

NSLog(@"block is %@", ^{

    NSLog(@"test Arr :%@", testArr);
});
//block is <__NSStackBlock__: 0xbfffdac0>
//打印可看出block是一个 NSStackBlock，即在栈上，当函数返回时block将无效

NSLog(@"block is %@", TestBlock);
//block is <__NSMallocBlock__: 0x75425a0>
//上面这句在非arc中打印是 NSStackBlock，但是在arc中就是NSMallocBlock
//即在arc中默认会将block从栈复制到堆上，而在非arc中，则需要手动copy。
}
```

**3、NSMallocBlock只需要对NSStackBlock进行copy操作就可以获取，但是retain操作就不行,会在下面说明**

Block的copy、retain、release操作（还是copy一段）

不同于NSObject的copy、retain、release操作：

- Block\_copy与copy等效，Block\_release与release等效；
- 对Block不管是retain、copy、release都不会改变引用计数retainCount，retainCount始终是1；
- NSGlobalBlock：retain、copy、release操作都无效；
- NSStackBlock：retain、release操作无效，必须注意的是，NSStackBlock在函数返回后，Block内存将被回收。即使retain也没用。容易犯的错误是[[mutableAarry addObject:stackBlock]，（补：在arc中不用担心此问题，因为arc中会默认将实例化的block拷贝到堆上）在函数出栈后，从mutableAarry中取到的stackBlock已经被回收，变成了野指针。正确的做法是先将stackBlock copy到堆上，然后加入数组：[mutableAarry addObject:[stackBlock copy] autorelease]]。支持copy，copy之后生成新的NSMallocBlock类型对象。
- NSMallocBlock支持retain、release，虽然retainCount始终是1，但内存管理器中仍然会增加、减少计数。copy之后不会生成新的对象，只是增加了一次引用，类似



retain;

- 尽量不要对Block使用retain操作。

### 3.Block对外部变量的存取管理

基本数据类型

#### 1、局部变量

局部自动变量，在Block中只读。Block定义时copy变量的值，在Block中作为常量使用，所以即使变量的值在Block外改变，也不影响他在Block中的值。

```
{
    int base = 100;
    long (^sum)(int, int) = ^ long (int a, int b) {

        return base + a + b;
    };

    base = 0;
    printf("%ld\n", sum(1, 2));
    // 这里输出是103，而不是3，因为块内base为拷贝的常量 100
}
```

#### 2、STATIC修饰符的全局变量

因为全局变量或静态变量在内存中的地址是固定的，Block在读取该变量值的时候是直接从其所在内存读出，获取到的是最新值，而不是在定义时copy的常量。

```
{
    static int base = 100;
    long (^sum)(int, int) = ^ long (int a, int b) {
        base++;
        return base + a + b;
    };

    base = 0;
    printf("%ld\n", sum(1, 2));
    // 这里输出是4，而不是103，因为base被设置为了0
    printf("%d\n", base);
    // 这里输出1， 因为sum中将base++了
}
```

#### 3、\_\_BLOCK修饰的变量

Block变量，被\_\_block修饰的变量称作Block变量。基本类型的Block变量等效于全局变量、或静态变量。

注：BLOCK被另一个BLOCK使用时，另一个BLOCK被COPY到堆上时，被使用的BLOCK也会被COPY。但作为参数的BLOCK是不会发生COPY的

### OBJC对象

block对于objc对象的内存管理较为复杂，这里要分static global local block变量分析、还要分非arc和arc分析

#### 非ARC中的变量

先看一段代码(非arc)



```

@interface MyClass : NSObject {
    NSObject* _instanceObj;
}
@end

@implementation MyClass

    NSObject* __globalObj = nil;

- (id) init {
    if (self = [super init]) {
        _instanceObj = [[NSObject alloc] init];
    }
    return self;
}

- (void) test {
    static NSObject* __staticObj = nil;
    __globalObj = [[NSObject alloc] init];
    __staticObj = [[NSObject alloc] init];

    NSObject* localObj = [[NSObject alloc] init];
    __block NSObject* blockObj = [[NSObject alloc] init];

    typedef void (^MyBlock)(void) ;
    MyBlock aBlock = ^{
        NSLog(@"%@", __globalObj);
        NSLog(@"%@", __staticObj);
        NSLog(@"%@", _instanceObj);
        NSLog(@"%@", localObj);
        NSLog(@"%@", blockObj);
    };
    aBlock = [[aBlock copy] autorelease];
    aBlock();

    NSLog(@"%d", [__globalObj retainCount]);
    NSLog(@"%d", [__staticObj retainCount]);
    NSLog(@"%d", [_instanceObj retainCount]);
    NSLog(@"%d", [localObj retainCount]);
    NSLog(@"%d", [blockObj retainCount]);
}
@end

int main(int argc, char *argv[]) {
    @autoreleasepool {
        MyClass* obj = [[MyClass alloc] init] autorelease;
        [obj test];
        return 0;
    }
}

```

执行结果为1 1 1 2 1。

**globalObj**和**staticObj**在内存中的位置是确定的，所以Block copy时不会retain对象。

**\_instanceObj**在Block copy时也没有直接retain **\_instanceObj**对象本身，但会retain self。所以在Block中可以直接读写**\_instanceObj**变量。

**localObj**在Block copy时，系统自动retain对象，增加其引用计数。

**blockObj**在Block copy时也不会retain。

## ARC中的变量测试

由于arc中没有retain，retainCount的概念。只有强引用和弱引用的概念。当一个变量没有\_\_strong的指针指向它时，就会被系统释放。因此我们可以通过下面的代码来测试。



```
代码片段1(globalObject全局变量)
NSString *__globalString = nil;

- (void)testGlobalObj
{
    __globalString = @"1";
    void (^TestBlock)(void) = ^{

        NSLog(@"string is :%@", __globalString); //string is :( null)
    };

    __globalString = nil;

    TestBlock();
}

- (void)testStaticObj
{
    static NSString *__staticString = nil;
    __staticString = @"1";

    printf("static address: %p\n", &__staticString);    //static address: 0x6a8c

    void (^TestBlock)(void) = ^{

        printf("static address: %p\n", &__staticString); //static address: 0x6a8c

        NSLog(@"string is : %@", __staticString); //string is :( null)
    };

    __staticString = nil;

    TestBlock();
}

- (void)testLocalObj
{
    NSString *__localString = nil;
    __localString = @"1";

    printf("local address: %p\n", &__localString); //local address: 0xbfffd9c0

    void (^TestBlock)(void) = ^{

        printf("local address: %p\n", &__localString); //local address: 0x71723e4

        NSLog(@"string is : %@", __localString); //string is : 1
    };

    __localString = nil;

    TestBlock();
}

- (void)testBlockObj
{
    __block NSString *_blockString = @"1";

    void (^TestBlock)(void) = ^{

        NSLog(@"string is : %@", _blockString); // string is :( null)
    };

    _blockString = nil;

    TestBlock();
}

- (void)testWeakObj
{
    NSString *__localString = @"1";

    __weak NSString *weakString = __localString;

    printf("weak address: %p\n", &weakString); //weak address: 0xbfffd9c4
    printf("weak str address: %p\n", weakString); //weak str address: 0x684c

    void (^TestBlock)(void) = ^{

        printf("weak address: %p\n", &weakString); //weak address: 0x7144324
        printf("weak str address: %p\n", weakString); //weak str address: 0x684c

        NSLog(@"string is : %@", weakString); //string is :1
    };

    __localString = nil;
```



```
TestBlock();  
}
```

由以上几个测试我们可以得出：

- 1、只有在使用local变量时，block会复制指针，且强引用指针指向的对象一次。其它如全局变量、static变量、block变量等，block不会拷贝指针,只会强引用指针指向的对象一次。
- 2、即时标记了为**weak**或**unsafe\_unretained**的local变量。block仍会强引用指针对象一次。（这个不太明白，因为这种写法可在后面避免循环引用的问题）

#### 循环引用retain cycle

循环引用指两个对象相互强引用了对方，即retain了对方，从而导致谁也释放不了谁的内存泄露问题。如声明一个delegate时一般用assign而不能用retain或strong，因为你一旦那么做了，很大可能引起循环引用。在以往的项目中，我几次用动态内存检查发现了循环引用导致的内存泄露。

这里讲的是block的循环引用问题，因为block在拷贝到堆上的时候，会retain其引用的外部变量，那么如果block中如果引用了他的宿主对象，那很有可能引起循环引用，如：

```
self.myblock = ^{  
    [self doSomething];  
};
```

为测试循环引用，写了些测试代码用于避免循环引用的方法，如下，（只有arc的，懒得做非arc测试了）



```
- (void)dealloc
{

NSLog(@"no cycle retain");
}

- (id)init
{
    self = [super init];
    if (self) {

#ifdef TestCycleRetainCase1

        //会循环引用
        self.myblock = ^{

            [self doSomething];
        };
#elif TestCycleRetainCase2

        //会循环引用
        __block TestCycleRetain *weakSelf = self;
        self.myblock = ^{

            [weakSelf doSomething];
        };
#elif TestCycleRetainCase3

        //不会循环引用
        __weak TestCycleRetain *weakSelf = self;
        self.myblock = ^{

            [weakSelf doSomething];
        };
#elif TestCycleRetainCase4

        //不会循环引用
        __unsafe_unretained TestCycleRetain *weakSelf = self;
        self.myblock = ^{

            [weakSelf doSomething];
        };
#endif

        NSLog(@"myblock is %@", self.myblock);
    }
    return self;
}

- (void)doSomething
{
    NSLog(@"do Something");
}

int main(int argc, char *argv[]) {
    @autoreleasepool {
        TestCycleRetain* obj = [[TestCycleRetain alloc] init];
        obj = nil;
        return 0;
    }
}
```

经过上面的测试发现，在加了**weak**和**unsafe\_unretained**的变量引入后，TestCycleRetain方法可以正常执行dealloc方法，而不转换和用**block**转换的变量都会引起循环引用。

因此防止循环引用的方法如下： `unsafe_unretained TestCycleRetain *weakSelf = self;`

喜欢的话记得点赞！！！ 谢谢





希望您能加入我们的iOS开发交流QQ群:584599353


(/u/8fed18ed70c9)

+ 关注

喜欢，您就打赏！

赞赏支持

♡ 喜欢 (/sign\_in) | 31



更多分享

(http://cwb.assets.jianshu.io/notes/images/4670901




登录 (/sign\_in) 后发表评论

2条评论

只看作者

按喜欢排序 按时间正序 按时间倒序




夜空\_守望者 (/u/777a32e47403)

2楼 · 2016.07.13 14:54

(/u/777a32e47403)

很清晰,又让自己重新认识了block

👍 赞    💬 回复



Thebloodelves (/u/cd648456ae0d)

3楼 · 2016.10.20 20:35

(/u/cd648456ae0d)

很清晰

👍 赞    💬 回复

被以下专题收入，发现更多相似内容

iOS 开发之路

首页投稿

程序员

iOS OC ...

iOS Dev...

iOS分享的demo

面试

iOS入的那些坑

iOS开发攻城...

QB@IO NIC

iOS 开发

IOS三人行

iOS备忘录

程序

iOS进阶

阅读更多...

↑

🔗