正确使用Block避免Cycle Retain和Crash

Posted by tanqisen Apr 19th, 2013

> *本文只介绍了MRC时的情况，有些细节不适用于ARC。比如MRC下__block不会增加引用计数，但ARC会，ARC下必须用__weak指明不增加引用计数；ARC下block内存分配机制也与MRC不一样，所以文中的一些例子在ARC下测试结果可能与文中描述的不一样*

Block简介

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、词法作用域以内的外部变量的值。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

1. Block执行的代码，这是在编译的时候已经生成好的；
2. 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）。

Block基本语法

```
1  // 声明一个Block变量
```

```
long (^sum) (int, int) = nil;
// sum是个Block变量，该Block类型有两个int型参数，返回类型是long。

// 定义Block并赋给变量sum
sum = ^ long (int a, int b) {
  return a + b;
};

// 调用Block：
long s = sum(1, 2);
```

定义一个实例函数，该函数返回Block：

```
- (long (^)(int, int)) sumBlock {
    int base = 100;
    return [[ ^ long (int a, int b) {
      return base + a + b;
    } copy] autorelease];
  }

// 调用Block
[self sumBlock](1,2);
```

是不是感觉很怪？为了看的舒服，我们把Block类型typedef一下

```
typedef long (^BlkSum)(int, int);

- (BlkSum) sumBlock {
    int base = 100;
    BlkSum blk = ^ long (int a, int b) {
      return base + a + b;
    }
    return [[blk copy] autorelease];
}
```

Block在内存中的位置

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。

```
BlkSum blk1 = ^ long (int a, int b) {
  return a + b;
};
NSLog(@"blk1 = %@", blk1);// blk1 = <__NSGlobalBlock__: 0x47d0>


int base = 100;
BlkSum blk2 = ^ long (int a, int b) {
  return base + a + b;
};
NSLog(@"blk2 = %@", blk2); // blk2 = <__NSStackBlock__: 0xbfffddf8>
```

```
12
13  BlkSum blk3 = [[blk2 copy] autorelease];
14  NSLog(@"blk3 = %@", blk3); // blk3 = <__NSMallocBlock__: 0x902fda0>
```

为什么blk1类型是NSGlobalBlock，而blk2类型是NSStackBlock？blk1和blk2的区别在于，blk1没有使用Block以外的任何外部变量，Block不需要建立局部变量值的快照，这使blk1与函数没有任何区别，从blk1所在内存地址0x47d0猜测编译器把blk1放到了text代码段。blk2与blk1唯一不同是的使用了局部变量base，在定义（注意是定义，不是运行）blk2时，局部变量base当前值被copy到栈上，作为常量供Block使用。执行下面代码，结果是203，而不是204。

```
1   int base = 100;
2   base += 100;
3   BlkSum sum = ^ long (int a, int b) {
4     return base + a + b;
5   };
6   base++;
7   printf("%ld",sum(1,2));
```

在Block内变量base是只读的，如果想在Block内改变base的值，在定义base时要用`__block`修饰：`__block int base = 100;`。

```
1   __block int base = 100;
2   base += 100;
3   BlkSum sum = ^ long (int a, int b) {
4     base += 10;
5     return base + a + b;
6   };
7   base++;
8   printf("%ld\n",sum(1,2));
9   printf("%d\n",base);
```

输出将是214,211。Block中使用`__block`修饰的变量时，将取变量此刻运行时的值，而不是定义时的快照。这个例子中，执行`sum(1,2)`时，base将取`base++`之后的值，也就是201，再执行Block`base+=10; base+a+b`，运行结果是214。执

行完Block时，base已经变成211了。

Block的copy、retain、release操作

不同于NSObjec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]`，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。

Block对不同类型的变量的存取

基本类型

- 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;
BlkSum sum = ^ long (int a, int b) {
  // base++; 编译错误，只读
  return base + a + b;
};
base = 0;
printf("%ld\n",sum(1,2)); // 这里输出是103，而不是3
```

- static变量、全局变量。如果把上个例子的base改成全局的、或static。

Block就可以对他进行读写了。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;
BlkSum sum = ^ long (int a, int b) {
  base++;
  return base + a + b;
};
base = 0;
printf("%d\n", base);
printf("%ld\n",sum(1,2)); // 这里输出是3，而不是103
printf("%d\n", base);
```

输出结果是`0 4 1`，表明Block外部对base的更新会影响Block中的base的取值，同样Block对base的更新也会影响Block外部的base值。

- 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。

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。

```
void foo() {
  int base = 100;
  BlkSum blk = ^ long (int a, int b) {
    return  base + a + b;
  };
  NSLog(@"%@", blk); // <__NSStackBlock__: 0xbfffdb40>
  bar(blk);
}

void bar(BlkSum sum_blk) {
  NSLog(@"%@",sum_blk); // 与上面一样，说明作为参数传递时，并不会发生copy

  void (^blk) (BlkSum) = ^ (BlkSum sum) {
    NSLog(@"%@",sum);     // 无论blk在堆上还是栈上，作为参数的Block不会发生copy。
    NSLog(@"%@",sum_blk); // 当blk copy到堆上时，sum_blk也被copy了一分到堆上上。
  };
  blk(sum_blk); // blk在栈上

  blk = [[blk copy] autorelease];
  blk(sum_blk); // blk在堆上
}
```

ObjC对象，不同于基本类型，Block会引起对象的引用计数变化。

先看下面代码

```
@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。

非ObjC对象，如GCD队列dispatch_queue_t。Block copy时并不会自动增加他

的引用计数，这点要非常小心。

Block中使用的ObjC对象的行为

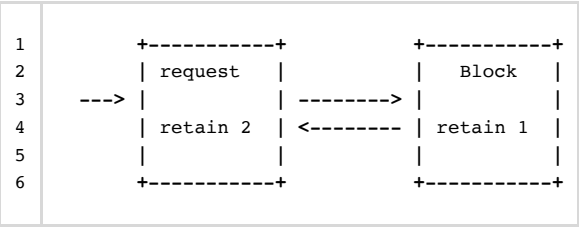
```
@property (nonatomic, copy) void(^myBlock)(void);

MyClass* obj = [[[MyClass alloc] init] autorelease];
self.myBlock = ^ {
  [obj doSomething];
};
```

对象obj在Block被copy到堆上的时候自动retain了一次。因为Block不知道obj什么时候被释放，为了不在Block使用obj前被释放，Block retain了obj一次，在Block被释放的时候，obj被release一次。

retain cycle

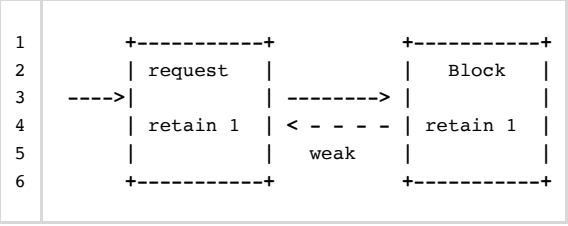
retain cycle问题的根源在于Block和obj可能会互相强引用，互相retain对方，这样就导致了retain cycle，最后这个Block和obj就变成了孤岛，谁也释放不了谁。比如：

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setCompletionBlock:^{
  NSString* string = [request responseString];
}];
```



解决这个问题的办法是使用弱引用打断retain cycle：

```
__block ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setCompletionBlock:^{
  NSString* string = [request responseString];
}];
```



request被持有者释放后。request 的retainCount变成0,request被dealloc，request释放持有的Block，导致Block的retainCount变成0，也被销毁。这样这两个对象内存都被回收。

```
      +-----------+           +-----------+
      | request   |           |   Block   |
 --X->|           | ----X---> |           |
      | retain 0  | < - - - - | retain 0  |
      |           |   weak    |           |
      +-----------+           +-----------+
```

与上面情况类似的陷阱：

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

这里self和myBlock循环引用，解决办法同上：

```
__block MyClass* weakSelf = self;
self.myBlock = ^ {
  [weakSelf doSomething];
};
```

```
@property (nonatomic, retain) NSString* someVar;

self.myBlock = ^ {
  NSLog(@"%@", _someVer);
};
```

这里在Block中虽然没直接使用self，但使用了成员变量。在Block中使用成员变量，retain的不是这个变量，而会retain self。解决办法也和上面一样。

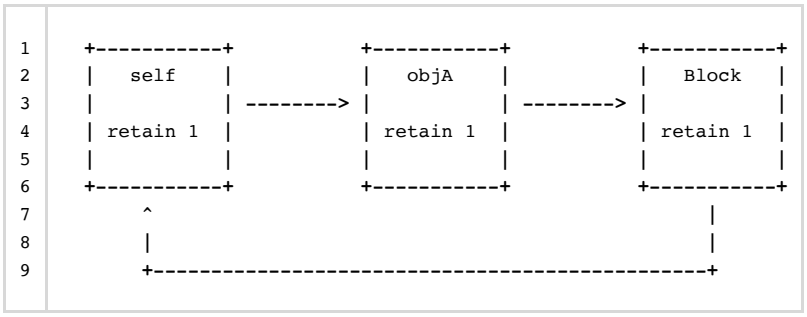
```
@property (nonatomic, retain) NSString* someVar;

__block MyClass* weakSelf = self;
self.myBlock = ^ {
  NSLog(@"%@", self.someVer);
};
```

或者

```
NSString* str = _someVer;
self.myBlock = ^ {
  NSLog(@"%@", str);
};
```

retain cycle不只发生在两个对象之间，也可能发生在多个对象之间，这样问题更复杂，更难发现

```
ClassA* objA = [[[ClassA alloc] init] autorelease];
  objA.myBlock = ^{
    [self doSomething];
  };
  self.objA = objA;
```



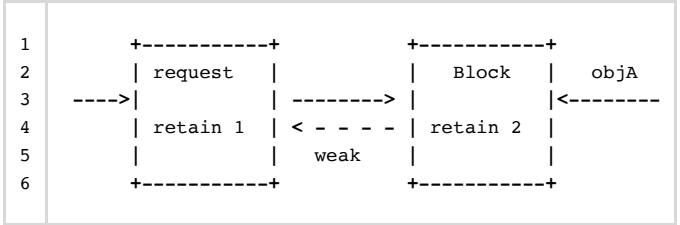
解决办法同样是用__block打破循环引用

```
ClassA* objA = [[[ClassA alloc] init] autorelease];

MyClass* weakSelf = self;
objA.myBlock = ^{
  [weakSelf doSomething];
};
self.objA = objA;
```

> *注意：MRC中__block是不会引起retain；但在ARC中__block则会引起retain。ARC中应该使用__weak或__unsafe_unretained弱引用。__weak只能在iOS5以后使用。*

Block使用对象被提前释放

看下面例子，有这种情况，如果不只是request持有了Block，另一个对象也持有了Block。



这时如果request 被持有者释放。

```
      +-----------+           +-----------+
      | request   |           |   Block   |   objA
 --X->|           | --------> |           |<--------
      | retain 0  | < - - - - | retain 1  |
      |           |   weak    |           |
      +-----------+           +-----------+
```

这时request已被完全释放，但Block仍被objA持有，没有释放，如果这时触发了Block，在Block中将访问已经销毁的request，这将导致程序crash。为了避免这种情况，开发者必须要注意对象和Block的生命周期。

另一个常见错误使用是，开发者担心retain cycle错误的使用__block。比如

```
__block kkProducView* weakSelf = self;
dispatch_async(dispatch_get_main_queue(), ^{
  weakSelf.xx = xx;
});
```

将Block作为参数传给dispatch_async时，系统会将Block拷贝到堆上，如果Block中使用了实例变量，还将retain self，因为dispatch_async并不知道self会在什么时候被释放，为了确保系统调度执行Block中的任务时self没有被意外释放掉，dispatch_async必须自己retain一次self，任务完成后再release self。但这里使用__block，使dispatch_async没有增加self的引用计数，这使得在系统在调度执行Block之前，self可能已被销毁，但系统并不知道这个情况，导致Block被调度执行时self已经被释放导致crash。

```
// MyClass.m
- (void) test {
  __block MyClass* weakSelf = self;
  double delayInSeconds = 10.0;
  dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, (int64_t)(delayInSeconds * NSEC_PER_SEC));
  dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
    NSLog(@"%@", weakSelf);
});

// other.m
MyClass* obj = [[[MyClass alloc] init] autorelease];
[obj test];
```

这里用dispatch_after模拟了一个异步任务，10秒后执行Block。但执行Block的时候MyClass* obj已经被释放了，导致crash。解决办法是不要使用__block。