Block编程值得注意的那些事儿

1，block 在实现时就会对它引用到的它所在方法中定义的栈变量进行一次只读拷贝，然后在 block 块内使用该只读拷贝。

如下代码:

```
- (void)testAccessVariable
{
NSInteger outsideVariable = 10;
//__block NSInteger outsideVariable = 10;
NSMutableArray * outsideArray = [[NSMutableArray alloc] init];

void (^blockObject)(void) = ^(void){
NSInteger insideVariable = 20;
NSLog(@"  > member variable = %d", self.memberVariable);
NSLog(@"  > outside variable = %d", outsideVariable);
NSLog(@"  > inside variable = %d", insideVariable);

[outsideArray addObject:@"AddedInsideBlock"];
```

```
};

outsideVariable = 30;
self.memberVariable = 30;

blockObject();

NSLog(@"  > %d items in outsideArray",
[outsideArray count]);
}
```

输出结果为:

```
> member variable = 30
> outside variable = 10
> inside variable = 20
> 1 items in outsideArray
```

注意到没? outside 变量的输出值为10，虽然outside变量在定义 block 之后在定义 block 所在的方法 testAccessVariable 中被修改为 20 了。这里的规则就是: blockObject 在实现时会对 outside 变量进行只读拷贝，在 block 块内使用该只读拷贝。因此这里输出的是拷贝时的变量值

10。如果，我们想要让 blockObject 修改或同步使用 outside 变量就需要用 __block 来修饰 outside 变量。

```
__block NSInteger outsideVariable = 10;
```

注意：

a)，在上面的 block 中，我们往 outsideArray 数组中添加了值，但并未修改 outsideArray 自身，这是允许的，因为拷贝的是 outsideArray 自身。

b)，对于 static 变量，全局变量，在 block 中是有读写权限的，因为在 block 的内部实现中，拷贝的是指向这些变量的指针。

c)， __block 变量的内部实现要复杂许多，__block 变量其实是一个结构体对象，拷贝的是指向该结构体对象的指针。

2，非内联（inline） block 不能直接访问 self，只能通过将 self 当作参数传递到 block 中才能使用，并且此时的 self 只能通过 setter 或 getter 方法访问其属性，不能使用句点式方法。但内联 block 不受此限制。

```
 typedef NSString* (^IntToStringConverter)(id self,
NSInteger paramInteger);
 - (NSString *)
convertIntToString:(NSInteger)paramInteger

usingBlockObject:(IntToStringConverter)paramBlock
Object
 {
 return paramBlockObject(self, paramInteger);
 }

 typedef NSString*
(^IntToStringInlineConverter)(NSInteger
paramInteger);
 - (NSString *)
convertIntToStringInline:(NSInteger)paramInteger

usingBlockObject:(IntToStringInlineConverter)para
mBlockObject
 {
 return paramBlockObject(paramInteger);
 }
```

```
 IntToStringConverter independentBlockObject = ^(id
self, NSInteger paramInteger) {
 KSLog(@" >> self %@, memberVariable %d", self, [self
memberVariable]);

 NSString *result = [NSString
stringWithFormat:@"%d", paramInteger];
 KSLog(@" >> independentBlockObject %@", result);
 return result;
 };

 - (void)testAccessSelf
 {
 // Independent
 //
 [self convertIntToString:20
usingBlockObject:independentBlockObject];

 // Inline
 //
```

```
 IntToStringInlineConverter inlineBlockObject =
^(NSInteger paramInteger) {
 KSLog(@" >> self %@, memberVariable %d", self,
self.memberVariable);

 NSString *result = [NSString
stringWithFormat:@"%d", paramInteger];
 KSLog(@" >> inlineBlockObject %@", result);
 return result;
 };
 [self convertIntToStringInline:20
usingBlockObject:inlineBlockObject];
 }
```

3，使用 weak–strong dance 技术来避免循环引用

在第二条中，我提到内联 block 可以直接引用 self，但是要非常小心地在 block 中引用 self。因为在一些内联 block 引用 self，可能会导致循环引用。如下例所示：

```
 @interface KSViewController ()
 {
```

```
id _observer;
}

@end

@implementation KSViewController

- (void)viewDidLoad
{
[super viewDidLoad];
// Do any additional setup after loading the view,
typically from a nib.

KSTester * tester = [[KSTester alloc] init];
[tester run];

_observer = [[NSNotificationCenter defaultCenter]
addObserverForName:@"TestNotificationKey"
object:nil queue:nil usingBlock:^(NSNotification
*n) {
NSLog(@"%@", self);
}];
```

```
}

- (void)dealloc
{
if (_observer) {
[[NSNotificationCenter defaultCenter]
removeObserver:_observer];
}
}
```

在上面代码中，我们添加向通知中心注册了一个观察者，然后在 dealloc 时解除该注册，一切看起来正常。但这里有两个问题：

a）在消息通知 block 中引用到了 self，在这里 self 对象被 block retain，而 _observer 又 retain 该 block的一份拷贝，通知中心又持有 _observer。因此只要 _observer 对象还没有被解除注册，block 就会一直被通知中心持有，从而 self 就不会被释放，其 dealloc 就不会被调用。而我们却又期望在 dealloc 中通过 removeObserver 来解除注册以消除通知中心对 _observer/block 的 retain。

b）同时，_observer 是在 self 所在类中定义赋值，因此是被 self retain 的，这样就形成了循环引用。

上面的过程 a）值得深入分析一下：

苹果官方文档中对 addObserverForName:object:queue:usingBlock: 中的 block 变量说明如下：

The block is copied by the notification center and (the copy) held until the observer registration is removed.

因此，通知中心会拷贝 block 并持有该拷贝直到解除 _observer 的注册。在 ARC 中，在被拷贝的 block 中无论是直接引用 self 还是通过引用 self 的成员变量间接引用 self，该 block 都会 retain self。

这两个问题，可以用 weak–strong dance 技术来解决。该技术在 WWDC 中介绍过：2011 WWDC Session #322 (Objective-C Advancements in Depth)

```
__weak KSViewController * wself = self;
_observer = [[NSNotificationCenter defaultCenter]
```

```
 addObserverForName:@"TestNotificationKey"
 object:nil queue:nil usingBlock:^(NSNotification
*n) {
 KSViewController * sself = wself;
 if (sself) {
 NSLog(@"%@", sself);
 }
 else {
 NSLog(@"<self> dealloc before we could run this
code.");
 }
 }];
```

下面来分析为什么该手法能够起作用。

首先，在 block 之前定义对 self 的一个弱引用 wself，因为是弱引用，所以当 self 被释放时 wself 会变为 nil；然后在 block 中引用该弱应用，考虑到多线程情况，通过使用强引用 sself 来引用该弱引用，这时如果 self 不为 nil 就会 retain self，以防止在后面的使用过程中 self 被释放；然后在之后的 block 块中使用该强引用 sself，注意在使用前要对 sself 进行了 nil 检测，因为多线程环境下在用弱引用 wself 对强引用 sself 赋值时，弱引用 wself 可能已经为 nil 了。

通过这种手法，block 就不会持有 self 的引用，从而打破了循环引用。

扩展：其他还需要注意避免循环引用的地方

与此类似的情况还有 NSTimer。苹果官方文档中提到"Note in particular that run loops retain their timers, so you can release a timer after you have added it to a run loop."，同时在对接口

```
+ (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)seconds target:(id)target selector:(SEL)aSelector userInfo:(id)userInfo repeats:(BOOL)repeats
```

的 target 说明文档中提到：

The object to which to send the message specified by aSelector when the timer fires. The target object is retained by the timer and released when the timer is invalidated.

结合这两处文档说明，我们就知道只要重复性 timer 还没有被 invalidated，target 对象就会被一直持有而不会被释放。因此当你使用 self 当作 target 时，你就不能期望在 dealloc 中 invalidate timer，因为在 timer 没有被invalidate 之前，dealloc 绝不会被调用。因此，需要找个合适的时机和地方来 invalidate timer，但绝不是在 dealloc 中。

4，block 内存管理分析

block 其实也是一个 NSObject 对象，并且在大多数情况下，block 是分配在栈上面的，只有当 block 被定义为全局变量或 block 块中没有引用任何 automatic 变量时，block 才分配在全局数据段上。 __block 变量也是分配在栈上面的。

在 ARC 下，编译器会自动检测为我们处理了 block 的大部分内存管理，但当将 block 当作方法参数时候，编译器不会自动检测，需要我们手动拷贝该 block 对象。幸运的是，Cocoa 库中的大部分名称中包含”usingBlock“的接口以及 GCD 接口在其接口内部已经进行了拷贝操作，不需要我们再手动处理了。但除此之外的情况，就需要我们手动干预了。

```
- (id) getBlockArray
{
int val = 10;
return [[NSArray alloc] initWithObjects:
^{ KSLog(@"  > block 0:%d", val); },   // block on
the stack
^{ KSLog(@"  > block 1:%d", val); },   // block on
the stack
nil];

//   return [[NSArray alloc] initWithObjects:
//           [^{ KSLog(@"  > block 0:%d", val); }
copy],   // block copy to heap
//           [^{ KSLog(@"  > block 1:%d", val); }
copy],   // block copy to heap
//           nil];
}

- (void)testManageBlockMemory
{
id obj = [self getBlockArray];
typedef void (^BlockType)(void);
```

```
 BlockType blockObject = (BlockType)[obj
objectAtIndex:0];
 blockObject();
 }
```

执行上面的代码中，在调用 testManageBlockMemory 时，程序会 crash 掉。因为从 getBlockArray 返回的 block 是分配在 stack 上的，但超出了定义 block 所在的作用域，block 就不在了。正确的做法（被屏蔽的那段代码）是在将 block 添加到 NSArray 中时先 copy 到 heap 上，这样就可以在之后的使用中正常访问。

在 ARC 下，对 block 变量进行 copy 始终是安全的，无论它是在栈上,还是全局数据段,还是已经拷贝到堆上。对栈上的 block 进行 copy 是将它拷贝到堆上；对全局数据段中的 block 进行 copy 不会有任何作用；对堆上的 block 进行 copy 只是增加它的引用记数。

如果栈上的 block 中引用了__block 类型的变量，在将该 block 拷贝到堆上时也会将 __block 变量拷贝到堆上如果该 __block 变量在堆上还没有对应的拷贝的话，否则就增加堆上对应的拷贝的引用记数。