返回目录:全网各大厂 iOS 面试题-题集大全

iOS 开发中常见的内存问题有哪些？

内存问题主要包括两个部分，一个是 iOS 中常见循环引用导致的内存泄露 ，另外就是大量数据加载及使用导致的内存警告。

mmap

虽然苹果并没有明确每个 App 在运行期间可以使用的内存最大值，但是有开发者进行了实验和统计，一般在占用系统内存超过 20% 的时候会有内存警告，而超过 50% 的时候，就很容易 Crash 了，所以内存使用率还是尽量要少，对于数据量比较大的应用，可以采用分步加载数据的方式，或者采用 mmap 方式。mmap 是使用逻辑内存对磁盘文件进行映射，中间只是进行映射没有任何拷贝操作，避免了写文件的数据拷贝。 操作内存就相当于在操作文件，避免了内核空间和用户空间的频繁切换，能够提供高性能的写入速度。此外，mmap 可以保持数据的一致性，即使在对应的用户进程崩溃后，内存映射的文件仍然可以落盘。参见：mmap 实现数据一致性 。因为，用户进程崩溃后，内核会托管 mmap 的交换区，保证对应的数据能够存盘。sqlite 里也使用 mmap 提高性能防止丢数据。

循环引用

循环引用是 iOS 开发中经常遇到的问题，尤其对于新手来说是个头疼的问题。循环引用对 App 有潜在的危害，会使内存消耗过高，性能变差和 Crash 等，iOS 常见的内存主要以下三种情况：

1）Delegate

代理协议是一个最典型的场景，需要你使用弱引用来避免循环引用。ARC 时代，需要将代理声明为 weak 是一个即好又安全的做法：

```
@property (nonatomic, weak) id <MyCustomDelegate> delegate;
```

2）block

Block 的循环引用，主要是发生在 ViewController 中持有了 block，比如：

```
@property (nonatomic, copy) LFCallbackBlock callbackBlock;
```

同时在对 callbackBlock 进行赋值的时候又调用了 ViewController 的方法，比如：

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

就会发生循环引用，因为：ViewController -> 强引用了 callback -> 强引用了 ViewController，解决方法也很简单：

```
__weak __typeof(self) weakSelf = self;
self.callbackBlock = ^{
  [weakSelf doSomething];
}];
```

原因是使用 MRC 管理内存时，Block 的内存管理需要区分是 Global(全局)、Stack(栈)还是 Heap(堆)，而在使用了 ARC 之后，苹果自动会将所有原本应该放在栈中的 Block 全部放到堆中。全局的 Block 比较简单，凡是没有引用到 Block 作用域外面的参数的 Block 都会放到全局内存块中，在全局内存块的 Block 不用考虑内存管理问题。(放在全局内存块是为了在之后再次调用该 Block 时能快速反应，当然没有调用外部参数的 Block 根本不会出现内存管理问题)。

所以 Block 的内存管理出现问题的，绝大部分都是在堆内存中的 Block 出现了问题。默认情况下，Block 初始化都是在栈上的，但可能随时被收回，通过将 Block 类型声明为 copy 类型，这样对 Block 赋值的时候，会进行 copy 操作，copy 到堆上，如果里面有对 self 的引用，则会有一个强引用的指针指向 self，就会发生循环引用，如果采用 weakSelf，内部不会有强类型的指针，所以可以解决循环引用问题。

那是不是所有的 block 都会发生循环引用呢？其实不然，比如 UIView 的类方法 Block 动画，NSArray 等的类的遍历方法，也都不会发生循环引用，因为当前控制器一般不会强引用一个类。

此外，还有一种情况是在 self.callbackBlock 中使用了 ivar，也会造成循环引用。因为对 ivar 变量的直接访问还是会依赖 self 的编译地址再进行偏移。

3）NSTimer

NSTimer 我们开发中会用到很多，比如下面一段代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.myTimer = [NSTimer scheduledTimerWithTimeInterval:1 target:
self selector:@selector(doSomeThing) userInfo:nil repeats:YES];
}
- (void)doSomeThing {
}
- (void)dealloc {
     [self.timer invalidate];
     self.timer = nil;
}
```

这是典型的循环引用，因为 timer 会强引用 self，而 self 又持有了 timer，所有就造成了循环引用。那有人可能会说，我使用一个 weak 指针，比如：

```
    __weak typeof(self) weakSelf = self;
    self.myTimer = [NSTimer scheduledTimerWithTimeInterval:1 target:wea
kSelf selector:@selector(doSomeThing) userInfo:nil repeats:YES];
```

但是其实并没有用，因为不管是 weakSelf 还是 strongSelf，最终在 NSTimer 内部都会重新生成一个新的指针指向 self，这是一个强引用的指针，结果就会导致循环引用。那怎么解决呢？主要有如下三种方式：

3.1）使用中间类

创建一个继承 NSObject 的子类 MyTimerTarget，并创建开启计时器的方法。

```
    // MyTimerTarget.h
    #import <Foundation/Foundation.h>
    @interface MyTimerTarget : NSObject
    + (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)interva
l target:(id)target selector:(SEL)selector userInfo:(id)userInfo repeat
s:(BOOL)repeats;
    @end
    // MyTimerTarget.m
    #import "MyTimerTarget.h"
    @interface MyTimerTarget ()
    @property (assign, nonatomic) SEL outSelector;
    @property (weak, nonatomic) id outTarget;
    @end
    @implementation MyTimerTarget
    + (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)interva
l target:(id)target selector:(SEL)selector userInfo:(id)userInfo repeat
s:(BOOL)repeats {
        MyTimerTarget *timerTarget = [[MyTimerTarget alloc] init];
        timerTarget.outTarget = target;
        timerTarget.outSelector = selector;
        NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:interv
al target:timerTarget selector:@selector(timerSelector:) userInfo:userI
nfo repeats:repeats];
        return timer;
    }
    - (void)timerSelector:(NSTimer *)timer {
        if (self.outTarget && [self.outTarget respondsToSelector:self.o
utSelector]) {
            [self.outTarget performSelector:self.outSelector withObject:
timer.userInfo];
        } else {
            [timer invalidate];
        }
    }
    @end
    // 调用方
    @property (strong, nonatomic) NSTimer *myTimer;
    - (void)viewDidLoad {
```

```
        [super viewDidLoad];
        self.myTimer = [MyTimerTarget scheduledTimerWithTimeInterval:1
target:self selector:@selector(doSomething) userInfo:nil repeats:YES];
    }
    - (void)doSomeThing {
    }
    - (void)dealloc {
        NSLog(@"MyViewController dealloc");
    }
```

VC 强引用 timer，因为 timer 的 target 是 MyTimerTarget 实例，所以 timer 强引用 MyTimerTarget 实例，而 MyTimerTarget 实例弱引用 VC，解除循环引用。这种方案 VC 在退出时都不用管 timer，因为自己释放后自然会触发 `timerSelector:`中的 `[timer invalidate]`逻辑，timer 也会被释放。

3.2）使用类方法

我们还可以对 NSTimer 做一个 category，通过 block 将 timer 的 target 和 selector 绑定到一个类方法上，来实现解除循环引用。

```
    // NSTimer+MyUtil.h
    #import <Foundation/Foundation.h>
    @interface NSTimer (MyUtil)
    + (NSTimer *)MyUtil_scheduledTimerWithTimeInterval:(NSTimeInterval)
interval block:(void(^)())block repeats:(BOOL)repeats;
    @end
    // NSTimer+MyUtil.m
    #import "NSTimer+MyUtil.h"
    @implementation NSTimer (MyUtil)
    + (NSTimer *)MyUtil_scheduledTimerWithTimeInterval:(NSTimeInterval)
interval block:(void(^)())block repeats:(BOOL)repeats {
        return [self scheduledTimerWithTimeInterval:interval target:sel
f selector:@selector(MyUtil_blockInvoke:) userInfo:[block copy] repeats:
repeats];
    }
    + (void)MyUtil_blockInvoke:(NSTimer *)timer {
        void (^block)() = timer.userInfo;
        if (block) {
            block();
        }
    }
    @end
    // 调用方
    @property (strong, nonatomic) NSTimer *myTimer;
    - (void)viewDidLoad {
        [super viewDidLoad];
        self.myTimer = [NSTimer MyUtil_scheduledTimerWithTimeInterval:1
 block:^{
            NSLog(@"doSomething");
```

```
    } repeats:YES];
}
- (void)dealloc {
    if (_myTimer) {
        [_myTimer invalidate];
    }
    NSLog(@"MyViewController dealloc");
}
```

这种方案下，VC 强引用 timer，但是不会被 timer 强引用，但有个问题是 VC 退出被释放时，如果要停掉 timer 需要自己调用一下 timer 的 invalidate 方法。

3.2）使用 weakProxy

创建一个继承 NSProxy 的子类 MyProxy，并实现消息转发的相关方法。NSProxy 是 iOS 开发中一个消息转发的基类，它不继承自 NSObject。因为他也是 Foundation 框架中的基类, 通常用来实现消息转发, 我们可以用它来包装 NSTimer 的 target, 达到弱引用的效果。

```
// MyProxy.h
#import <Foundation/Foundation.h>
@interface MyProxy : NSProxy
+ (instancetype)proxyWithTarget:(id)target;
@end
// MyProxy.m
#import "MyProxy.h"
@interface MyProxy ()
@property (weak, readonly, nonatomic) id weakTarget;
@end
@implementation MyProxy
+ (instancetype)proxyWithTarget:(id)target {
    return [[MyProxy alloc] initWithTarget:target];
}
- (instancetype)initWithTarget:(id)target {
    _weakTarget = target;
    return self;
}
- (void)forwardInvocation:(NSInvocation *)invocation {
    SEL sel = [invocation selector];
    if (_weakTarget && [self.weakTarget respondsToSelector:sel]) {
        [invocation invokeWithTarget:self.weakTarget];
    }
}
- (NSMethodSignature *)methodSignatureForSelector:(SEL)sel {
    return [self.weakTarget methodSignatureForSelector:sel];
}
- (BOOL)respondsToSelector:(SEL)aSelector {
    return [self.weakTarget respondsToSelector:aSelector];
}
```

```
    @end
    // 调用方
    @property (strong, nonatomic) NSTimer *myTimer;
    - (void)viewDidLoad {
        [super viewDidLoad];
        self.myTimer = [NSTimer scheduledTimerWithTimeInterval:1 target:
[MyProxy proxyWithTarget:self] selector:@selector(doSomething) userInfo:
nil repeats:YES];
    }
    - (void)dealloc {
        if (_myTimer) {
            [_myTimer invalidate];
        }
        NSLog(@"MyViewController dealloc");
    }
```

上面的代码中，了解一下消息转发的过程就可以知道 -forwardInvocation: 是会有一个 NSInvocation 对象，这个 NSInvocation 对象保存了这个方法调用的所有信息，包括 Selector 名，参数和返回值类型，最重要的是有所有参数值，可以从这个 NSInvocation 对象里拿到调用的所有参数值。这时候我们把转发过来的消息和 weakTarget 的 selector 信息做对比，然后转发过去即可。

这里需要注意的是，在调用方的 dealloc 中一定要调用 timer 的 invalidate 方法，因为如果这里不清理 timer，这个调用方 dealloc 被释放后，消息转发就找不到接收方了，就会 crash。

3.3）使用 GCD timer

GCD 提供的定时器叫 dispatch_source_t。使用方式如下：

```
    // 调用方
    @property (strong, nonatomic) dispatch_source_t myGCDTimer;
    - (void)viewDidLoad {
        [super viewDidLoad];
        dispatch_source_t timer = dispatch_source_create(DISPATCH_SOURC
E_TYPE_TIMER, 0, 0, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_D
EFAULT, 0));
        if (timer) {
            self.myGCDTimer = timer;
            dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0),
 1 * NSEC_PER_SEC, 1ull * NSEC_PER_SEC);
            dispatch_source_set_event_handler(timer, ^ {
                NSLog(@"doSomething");
            });
            dispatch_resume(timer);
        }
    }
    - (void)dealloc {
        if (_myGCDTimer) {
```

```
            dispatch_cancel(_myGCDTimer);
        }
        NSLog(@"MyViewController dealloc");
    }
```

更多详情见：NSTimer 循环引用解决方案

其他内存问题

- NSNotification addObserver 之后，记得在 dealloc 里面添加 remove。
- 动画的 repeat count 无限大，而且也不主动停止动画，基本就等于无限循环了。
- forwardingTargetForSelector 返回了 self。

高性能地使用内存的建议

- 熟读 ARC 机制原理 。
- 使用 `weak` 修饰替换 `unsafe_unretain`。
- 小心方法中的 self，在 Objective-C 的方法中隐含的 self 是 `__unsafed_unretain` 的。
- 使用 Autorelease Pool 来降低循环中的内存峰值，避免 OOM。
- 要处理 Memory Warning。
- 需要在收到内存警告的时候释放的缓存类数据，在选用数据结构时，用 NSCache 代替 NSDictionary，使用 NSPurgableData 代替 NSData。在其他常见的操作系统上，由于局部性原理，OS 会将不常用的内存页面写回磁盘，频繁的写磁盘会缩短磁盘或闪存的生命，iOS 为了提升闪存的生命周期，所以没有交换空间，取而代之的是内存压缩技术，iOS 将不常用到的 dirty 页面压缩以减少页面占用量，在再次访问到的时候重新解压缩。这些都在操作系统层面实现，对进程无感知。倘若在使用 NSDictionary 的时候收到内存警告，然后去释放这个 NSDictionary，如果占据的内存过大，很可能在内存解压的过程中造成内存压力更大而导致 App 就被 JetSam 给 Kill 掉了，如果你的内存只是缓存或者是可重建的数据，就把 NSCache 当初 NSDictionary 用。同理 NSPurableData 也是。
- UITableView/UICollectionView 的重用不单单是 cell 重用，cell 使用的子 view 也要重用。
- `[UIImage imageNamed:]` 适合于 UI 界面中的贴图的读取，较大的资源文件应该尽量避免使用。
- WKWebView 是跨进程通信的，不会占用我们的 APP 使用的物理内存量。
- try、catch、finally 一定要清理资源。
- 对大的内存对象进行懒加载，但是要注意线程安全。

关于 iOS 内存管理更多的内容，参见 iOS Memory Deep Dive。

内存解决思路

- 通过 Instruments 来查看 leaks。

- 集成 Facebook 开源的 FBRetainCycleDetector。
- 集成 MLeaksFinder。

更多信息参加：iOS App 稳定性指标及监测

返回目录:全网各大厂 iOS 面试题-题集大全
