iOScarsh分析

一、通过查看友盟统计奔溃日志分析  
二、通过xcode自带的xcode—window---organizer—carsh分析

三、通过命令行解析

1、找到dysm文件  
2、找出carsh文件  
3、执行命令

3.1dwarfdump --uuid /Users/hy001/Downloads/沃阅读.app.dSYM

3.2dwarfdump --arch=arm64 --lookup 0x104db0000 /Users/hy001/Downloads/沃阅读.app.dSYM/Contents/Resources/DWARF/沃阅读  
 3.3关于dsym文件的查找打包时需要设置  
 然后找到对应的archives找到dysms文件夹

**四、Crash分析**

以上图为例，大部分字段都是不言而喻的，下面列举一些有用处的。（在官方文档都有解释，这里做归纳与翻译）

**Incident Identifier**： 报告的唯一标识符，两份报告决不会共享同一个事件标识符。  
**CrashReporter Key**：每个设备的匿名标识符，来自同一设备的两个报告将包含相同的值。  
**Process**：很明显是我们的进程名称。  
**Date/Time 与 Launch Time**：报告生成时间与程序开始运行时间  
**Exception Type**：异常类型  
**Exception Note**：不属于异常类型的附加信息，如果这个字段包含SIMULATED(不是崩溃)，那这个进程不是崩溃的，而是在系统的请求下被杀死，通常是看门狗机制起了作用（APP内一段时间内无法响应用户的操作，会被系统kill）。

由下到上为最后调用的顺序

四、错误类型分析

##### 内存访问不良[EXC\_BAD\_ACCESS // SIGSEGV // SIGBUS]

通常用于访问了不该访问的内存导致或者尝试以不允许的方式访问内存（例如只读属性，比如在上一篇我们提到的通过KVO更改只读属性），并且" Exception SubType"字段会包含kern\_return\_t来描述错误和未正确访问的内存地址。  
下面是官方给出的建议：

* 如果objc\_msgSend或objc\_release接近崩溃线程的Backtraces(堆栈信息回溯)的顶部，则该进程可能试图向释放的对象发送消息，可以使用[Zombie Instrument](https://link.jianshu.com/?t=https%3A%2F%2Fdeveloper.apple.com%2Flibrary%2Fcontent%2Fdocumentation%2FDeveloperTools%2FConceptual%2FInstrumentsUserGuide%2FEradicatingZombies.html" \t "_blank)来分析应用程序，以更好地了解此次崩溃的情况。
* 如果gpus\_ReturnNotPermittedKillClient接近崩溃线程的Backtraces的顶部，则该进程被终止，因为它尝试在后台使用OpenGL ES或Metal进行渲染。

##### 异常退出[EXC\_CRASH // SIGABRT]

该进程异常退出，此异常类型崩溃的最常见原因是向对象发送了无法识别的消息，比如上文中向NSArray发送了addObject消息。  
另外如果App Extensions需要太多时间来初始化（看门狗机制），那么App Extensions将终止于此异常类型，如果扩展因启动时挂起而死亡，则生成的崩溃报告的**Exception Subtype**将会是LAUNCH\_HANG，由于扩展没有main函数，任何花在初始化上的时间都会在+load扩展库和相关库中的静态构造函数和方法中，你应该尽可能多地推迟这项工作。

##### 资源限制[EXC\_RESOURCE]

该过程超出了资源消耗限制，这是来自操作系统的通知，该进程正在使用太多的资源,确切的资源列在**Exception SubType**字段中。如果**Exception Note**字段包含NON-FATAL CONDITION，则即使生成崩溃报告，该进程也不会被终止。

* 异常子类型MEMORY表示该进程已超过系统施加的内存限制。
* 异常子类型WAKEUPS表示进程中的线程每秒被唤醒的次数过多，这迫使CPU醒来的频率很高，并且消耗电池寿命。

## Trace Trap [EXC\_BREAKPOINT // SIGTRAP]

这个异常是为了让一个附加的调试器有机会在执行过程中的某个特定时刻中断进程，我们可以在代码里面添加\_\_builtin\_trap()方法手动触发这个异常。如果没有附加调试器，则会生成crash文件。  
较低级别的库（例如libdispatch）会在遇到致命错误时捕获该进程。  
swift代码在遇到下面2种情况的时候也会抛出这个异常：

* 一个非可选类型的值为nil
* 错误的强制类型转换，如把NSString转换为NSDate等等。

## Illegal Instruction [EXC\_BAD\_INSTRUCTION // SIGILL]

程序尝试的去执行一个非法的或者没有定义的指令。如果配置的函数地址有问题，进程在执行函数跳转的时候，就会发生这个crash。

## Quit [SIGQUIT]

该进程在具有管理其生命周期的权限的另一进程的请求下终止。 SIGQUIT并不意味着进程崩溃，但是可以说明该进程存在一些问题。  
比如在[iOS](https://so.csdn.net/so/search?q=iOS&spm=1001.2101.3001.7020" \t "_blank)中，第三方键盘应用可能在在其他APP中被唤起，但是如果键盘应用需要很长的时间去加载，则会被强制退出。

## Killed [SIGKILL]

进程被系统强制结束，通过查看Termination Reason找到crash信息。

## Guarded Resource Violation

## Resource Limit

进程使用的资源超出的限制。这个一个系统的通知，我们可以在Exception Subtype里面查看相关信息。

## Other Exception Types

一些crash report记录的exception信息可能并不是描述型的，可能是一个十六进制的代码，比如00000020，下面列出了几种常见的异常代码：

* 0xbaaaaaad：该code表示这个crash文件是系统的stackshot，并不是crash report，可以通过按住home+voice按键生成；
* 0xbad22222：一个VoIP应用启动恢复的次数太频繁；
* 0x8badf00d： 看门狗定时器超时，上面有讲到，一般是因为APP启动的时间过长或者响应系统事件事件超时导致；比如在主线程进行网络请求，主线程会一直卡住知道网络回调回来；
* 0xc00010ff ： 当操作系统响应thermal事件的时候，会强制的kill进程。
* 0xdead10cc： 进程在suspend期间保持在文件锁或sqlite数据库锁。

#### 4. Mach异常和Unix信号的分类

**常见的Mach异常**

* EXC\_CRASH: 进程异常退出(SIGABORT) 或者 watch dog超时杀死App(SIGKILL)
* EXC\_BREAKPOINT (SIGTRAP)
* EXC\_BAD\_ACCESS :内存访问无效
* EXC\_BAD\_INSTRUCTION:线程试图访问非法/无效的指令或将无效的参数（操作数）传递给指令
* EXC\_ARITMETHIC:除以0或整数溢出/下溢引发的异常
* EXC\_SYSCALL 和 EXC\_MACH\_SYSCALL:应用程序访问内核服务（如文件I/O）或网络访问时发出
* 其他Mach异常定义在mach/exception\_types.h中。与处理器相关的异常定义在mach/(i386,ppc,...)/exception.h中  
  在开发中最常见的异常应该是EXC\_BAD\_ACCESS，就比如这次追踪到的

**Unix信号**

信号处理函数可以通过 signal() 系统调用来设置。如果没有为一个信号设置对应的处理函数，就会使用默认的处理函数，否则信号就被进程截获并调用相应的处理函数。在没有处理函数的情况下，程序可以指定两种行为：忽略这个信号 SIG\_IGN 或者用默认的处理函数 SIG\_DFL 。但是有两个信号是无法被截获并处理的： SIGKILL、SIGSTOP 。

**Signal信号类型：**

* SIGABRT--程序中止命令中止信号
* SIGALRM--程序超时信号
* SIGFPE--程序浮点异常信号
* SIGILL--程序非法指令信号
* SIGHUP--程序终端中止信号
* SIGINT--程序键盘中断信号
* SIGKILL--程序结束接收中止信号
* SIGTERM--程序kill中止信号
* SIGSTOP--程序键盘中止信号
* SIGSEGV--程序无效内存中止信号
* SIGBUS--程序内存字节未对齐中止信号
* SIGPIPE--程序Socket发送失败中止信号
* **二、防护纲要**
* 1、容器越界、非空防护  
  2、unrecognized selector 崩溃  
  3、NSTimer  
  4、KVO crash  
  5、Bad Access crash （野指针）

#### 1、容器越界、非空防护

[方法交换的原理](https://www.jianshu.com/p/adcad5533321)对容器进行crash预处理。

2 unrecognized selector 崩溃  
处理 建立一个专门用于消息拦截的类，每次发生unrecognized selector的时候，都将消息转发给这个类，那么我们就可以统一处理了。

**3、NSTimer防护**

NSTimer使用频率很高，但每次使用后，都要在dealloc之前将定时器释放掉，否则会引起循环引用，甚至崩溃。因为timer被target强引用的同时自身也被target所持有，如果不在dealloc之前释放定时器，target对象也将无法释放。  
针对这个情况，我们想到的是解除target和timer之间互相引用的环，但又不影响业务逻辑的开展。那么如何才能解除这个环呢？

**1、timer被target引用**

@property (nonatomic, weak) NSTimer \*timer;

target依然被timer强引用，导致dealloc方法不执行，循环引用仍然存在

**2、target被timer引用**

\_\_weak typeof(self) weakSelf = self;

self.timer = [NSTimer scheduledTimerWithTimeInterval:1 target:weakSelf selector:@selector(test) userInfo:nil repeats:YES];

对target进行弱引用以后，并不能改变target指向的内存地址，例如上述的self指向地址S，弱引用后，weakSelf仍然指向地址S，当我们使用weakSelf传给定时器的时候，相当于是将地址S传过去，scheduledTimerWithTimeInterval方法内部仍然强引用地址S，dealloc方法仍然无法被执行，循环引用仍然存在。

**3、timer与target直接加入一个桥阶层**

桥阶层

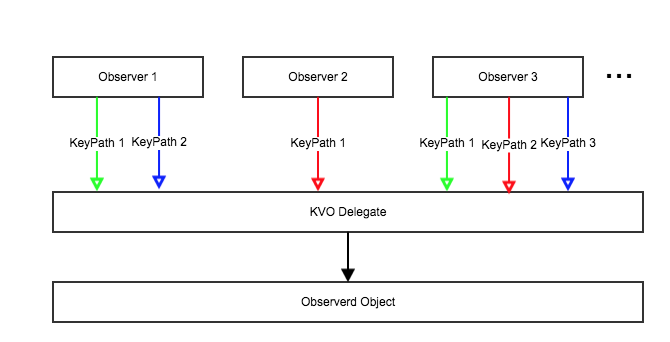
image.png

image.png

通过方法交换，在scheduledTimerWithTimeInterval方法中，新建一个subTarget对象，并将target，selector，timer，targetClass等参数传递给subTarget，subTarget内部对这些参数进行弱引用持有，并实现一个中转方法给原始scheduledTimerWithTimeInterval调用。这样，每当定时器触发时，target对象强引用了timer，timer强引用subTarget，并向subTarget调用中转方法，由subTarget的中转方法再将方法分发给原始target，而原始target则被subTarget弱引用。  
此时，当target被需要被释放时，由于没有被timer强引用，dealloc会被调用，同时会释放subTarget和timer。由于timer及时被释放了，回调函数在target被释放后不会再执行，就不会再引起野指针等异常内存地址访问的问题。

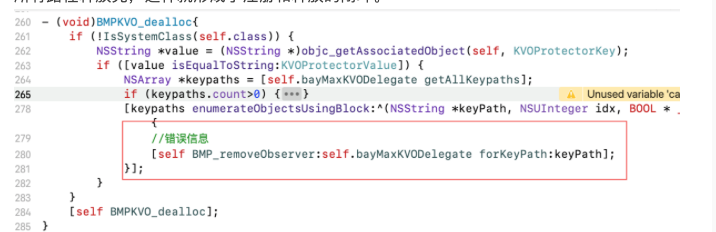
#### 4、KVO crash防护

从上述内容我们可知，KVO的注册与释放彼此依赖，与NSTimer的防护非常相似，如果能够打破这个环结构，那么就可以避免KVO的崩溃了。我们参考NSTimer的思路，在注册与释放环节增加中间桥接层，让注册与依赖不强相关，而分别于桥接层强相关，在对象释放的时候，只需要操作桥接层即可，即便注册和释放不成对，也不会引起KVO的crash。



如上图我们可以知道，在观察者与被观察者之间的这层桥接层（KVO delegate）负责记录两者之间的关系，对观察者而言，KVO delegate是被观察者，对被观察者而言，KVO delegate是观察者。

  
KVO delegate有一个属性用于记录观察路径等信息，当对象被释放时，会将KVO delegate的所有路径释放完，这样就形成了注册和释放的闭环。



#### 5、Bad Access crash （野指针）

野指针或者异常内存地址访问是经常遇到，并且是很难处理的问题，如果能够对这一块的崩溃进行提前规避掉，将大大降低崩溃率。所以只能平常写代码时多注意。