欧阳大哥2013 Lv3
2021年03月30日 阅读 1484

iOS疑难Crash的寄存器赋值追踪排查技术

我们会借助一些崩溃日志收集库来定位和排查线上的崩溃信息，但是有些崩溃堆栈所提供的信息有限又不是必现崩溃，很难直观排查出问题的所在。这里我给大家分享一个采用**寄存器赋值追踪的技术**来排查和分析崩溃日志的技巧。话不多说先看案例：


```
//符号化后的崩溃堆栈
Date/Time:       2021-03-25 04:35:38.211 +0800
OS Version:      iOS 10.3.2 (14F89)
Report Version:  104

Monitor Type:    Unix Signal
Exception Type:  EXC_CRASH (SIGABRT)
Exception Codes: 0x00000000 at 0x00000001808f1014
Crashed Thread:  27


Pthread id: 1313098
Thread 27 Crashed:
0   libsystem_kernel.dylib          __pthread_kill + 8
1   libsystem_pthread.dylib         pthread_kill + 112
2   libsystem_c.dylib               abort + 140
3   libsystem_malloc.dylib          szone_error + 420
4   libsystem_malloc.dylib          free_list_checksum_botch.295 + 36
5   libsystem_malloc.dylib          tiny_free_list_remove_ptr + 288
6   libsystem_malloc.dylib          tiny_free_no_lock + 684
7   libsystem_malloc.dylib          free_tiny + 472
8   CoreFoundation                  _CFRelease + 1228
//只有这句信息可以参考
9   testApp                        __99-[XXX fn:queue:]_block_invoke + 384
10  libdispatch.dylib               _dispatch_call_block_and_release + 24
11  libdispatch.dylib               _dispatch_client_callout + 16
12  libdispatch.dylib               _dispatch_queue_serial_drain + 928
13  libdispatch.dylib               _dispatch_queue_invoke + 884
14  libdispatch.dylib               _dispatch_root_queue_drain + 540
```

复制代码

```
//这是原始崩溃栈
Pthread id: 1313098
 Thread 27 Crashed:
0   libsystem_kernel.dylib          0x00000001808f1014 __pthread_kill + 8
1   libsystem_pthread.dylib         0x00000001809bb264 pthread_kill + 112
2   libsystem_c.dylib               0x00000001808659c4 abort + 140
3   libsystem_malloc.dylib          0x0000000180931828 szone_error + 420
4   libsystem_malloc.dylib          0x000000018093b74c 0x180924000 + 96076
5   libsystem_malloc.dylib          0x0000000180928994 0x180924000 + 18836
6   libsystem_malloc.dylib          0x000000018093ba00 0x180924000 + 96768
7   libsystem_malloc.dylib          0x000000018093c0c8 0x180924000 + 98504
8   CoreFoundation                  0x00000001818a701c 0x1817ca000 + 905244
9   testApp                         0x0000000103b9e5b8 __99-[XXX fn:queue:]_block_invoke + 384
10  libdispatch.dylib               0x00000001807ae9e0 0x1807ad000 + 6624
11  libdispatch.dylib               0x00000001807ae9a0 0x1807ad000 + 6560
12  libdispatch.dylib               0x00000001807bcad4 0x1807ad000 + 64212
13  libdispatch.dylib               0x00000001807b22cc 0x1807ad000 + 21196
14  libdispatch.dylib               0x00000001807bea50 0x1807ad000 + 72272
15  libdispatch.dylib               0x00000001807be7d0 0x1807ad000 + 71632
16  libsystem_pthread.dylib         0x00000001809b7100 _pthread_wqthread + 1096


Binary Images:
       0x1000b8000 -        0x108263fff +testApp arm64  <cb8cc0075ed4352b802c6c586b8a93d5> /va
```

从上面的崩溃信息大概可以看出这是一个GCD队列线程调用时发生了崩溃。其中崩溃的第9层显示是在一个[XXX fn:queue:] 方法内定义的block内部代码发生了崩溃，除此之外没有其它信息。崩溃方法的源代码定义如下：

复制代码

```
//这是一个简化代码
@interface TestObj:NSObject

-(NSString*)testString;

-(NSInteger)length;

@end

//代码片段
```

```
                if ([testObj length] != 0) {
                    NSString *suffix = [testObj testString];
                    const static int len = 4;
                    if (suffix.length > len) {
                        suffix = [suffix substringToIndex:len];
                    }
                }
            }
        });
}
```

从源代码来看确实是在方法-[XXX fn:queue:]内调用了一个dispatch_async，然后block也是定义在方法内。不过依然没办法定位到是哪行代码发生了崩溃, 同时也不是必现的线上崩溃。

所以要想查明原因需要到汇编代码级别定位崩溃原因！！ 步骤如下：

1. 先下载可执行文件到本地或者从CI发布部门获取可执行的app包并解压。

2. 用系统自带的otool工具，进行代码的反汇编处理。下面的otool命令格式可以用来显示具体的函数或者方法的反汇编代码：

复制代码

```
otool "可执行文件路径"   -p "函数或者方法名" -V -t
```

otool命令中 -p 后面跟的是方法名或者函数名或者符号名。 这里需要注意的时因为系统编译时可能会在函数名或者符号名前多增加一个下划线_ 因此在指定符号名时需要多增加一个_。 -V 是表明打印函数对应的汇编代码。 -t 是表明打印代码段中的代码。

本例的问题使用otool如下：

复制代码

```
otool   "/Users/apple/Downloads/Payload/testApp.app/testApp" -p "___99-[XXX fn:queue:]_block_
```

汇编出来的局部代码如下：

复制代码

```
/Users/apple/Downloads/Payload/testApp.app/testApp:
(__TEXT,__text) section
___99-[XXX fn:queue:]_block_invoke:
```

```
0000000103ae655c<+292>  adrp    x8, 26695 ; 0x10a32d000
0000000103ae6560<+296>  ldr     x1, [x8, #0x940] ; Objc selector ref: testString
0000000103ae6564<+300>  bl      0x107fac9e8 ; Objc message: -[x0 testString]
0000000103ae6568<+304>  mov     x29, x29
0000000103ae656c<+308>  bl      0x107faca48 ; symbol stub for: _objc_retainAutoreleasedReturnV
0000000103ae6570<+312>  mov     x25, x0
0000000103ae6574<+316>  mov     x0, x26
0000000103ae6578<+320>  bl      0x107faca18 ; symbol stub for: _objc_release
0000000103ae657c<+324>  mov     x0, x25
0000000103ae6580<+328>  mov     x1, x22
0000000103ae6584<+332>  bl      0x107fac9e8 ; Objc message: -[x0 testString]  //这里是otool的bug
0000000103ae6588<+336>  cmp     x0, #0x5
0000000103ae658c<+340>  b.lo    0x103ae65bc
0000000103ae6590<+344>  adrp    x8, 26674 ; 0x10a318000
0000000103ae6594<+348>  ldr     x1, [x8, #0xb30] ; Objc selector ref: substringToIndex:
0000000103ae6598<+352>  mov     x0, x25
0000000103ae659c<+356>  mov     w2, #0x4
0000000103ae65a0<+360>  bl      0x107fac9e8 ; Objc message: -[x0 substringToIndex:]
0000000103ae65a4<+364>  mov     x29, x29
0000000103ae65a8<+368>  bl      0x107faca48 ; symbol stub for: _objc_retainAutoreleasedReturnV
0000000103ae65ac<+372>  mov     x22, x0
0000000103ae65b0<+376>  mov     x0, x25
0000000103ae65b4<+380>  bl      0x107faca18 ; symbol stub for: _objc_release
0000000103ae65b8<+384>  mov     x25, x22
```

从上述的崩溃信息可以看出崩溃的地址是0x0000000103b9e5b8。根据这个地址可以得出崩溃是在我们反汇编代码的倒数第二行。 `0x0000000103ae65b4<+380>`

> 上述的两个地址都不同，结论是如何得出的?

1. 线上的程序运行时程序时会在一个随机的基地址上加载，从崩溃的原始堆栈中最下面部分可以看到程序映像的基地址就是 0x1000b8000。因此：

复制代码

```
0x0000000103b9e5b8 - 0x1000b8000 = 0x0000000103ae65b8
```

2. 又因为一般程序崩溃的地址都有3个特征：

a. 崩溃堆栈层级中的非顶层地址都是函数调用指令的下一条地址也就是LR的值，所以真实的崩溃指令是第1步算出的结果再减去4也就是实际崩溃的地址是：0x0000000103ae65b4

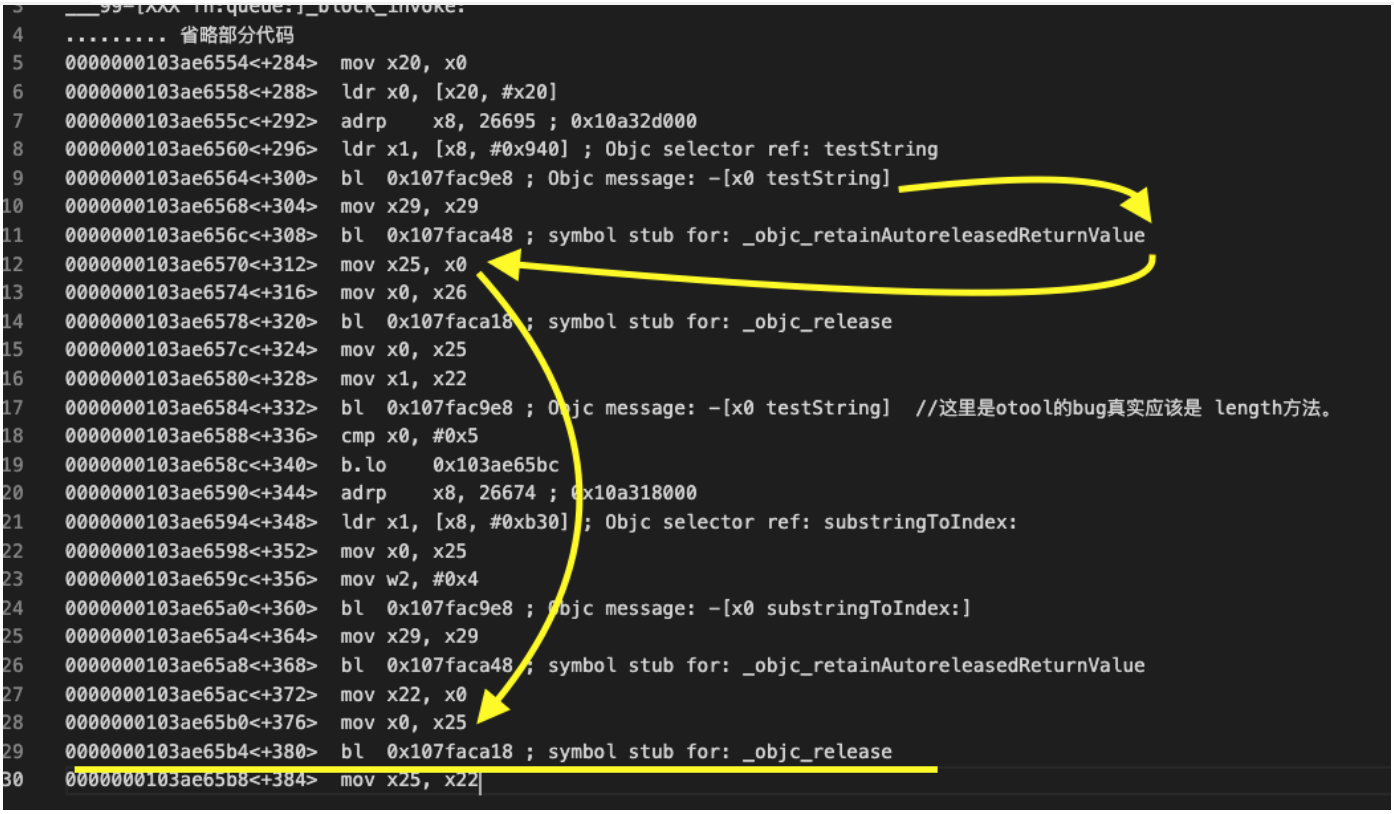
c . 如果崩溃信息出现在最顶层即无内存访问也无函数调用的指令时，这种崩溃一般是触发了brk断点指令，或者产生了其他一些无法可判断的原因了。前者比较好定位，后者就很难了。

从上面汇编代码倒数第二行<+384>处可以看出是一个对象调用了_objc_release进行释放时导致了崩溃。而_objc_release内部可能会调用_CFRelease方法，这也就是在上面崩溃信息堆栈中__99-[XXX fn:queue:]_block_invoke + 384的上面是_CFRelease函数了。

既然是因为对象调用_objc_release导致的内存释放异常。那么就需要继续追踪是哪个对象调用了_objc_release。

根据arm系统的函数调用ABI规则，以及从倒数第三行<+376>的汇编代码中可以看出是执行了一个 x0 = x25的操作，也就是x0对象是从x25赋值而来的。这时候我们就可以利用寄存器赋值追踪的技巧，继续往上查看x25又是在哪里被赋值。 往上的代码可以看出在<+312>处的指令执行了 x25 = x0的赋值操作，而x0的结果是上一条指令调用_objc_retainAutoreleasedReturnValue函数返回的结果。而_objc_retainAutoreleasedReturnValue的入参又是上一条指令<+300>处的[x0 testString] 方法返回的结果。到这里为止我就可以从源代码中推断出是[testObj testString] 返回的结果对象在释放时导致了崩溃了。

那接下来你就可以仔细查查源代码[testObj testString]的方法哪里有问题了。并最终定位出异常原因。



> 小贴士：在arm64位系统中，函数的第一个参数用x0寄存器保存，OC方法调用的对象也是用x0寄存器保存，函数和方法的返回结果也是用x0寄存器保存。

文章分类 iOS 文章标签 iOS

欧阳大哥2013 Lv3 美团
获得点赞 1,807 · 获得阅读 136,969

关注

输入评论...

最后是什么原因导致野指针了？多线程多次release对象导致的crash，[[testObj testString] copy]后面加一个copy可以解决这个问题么？

1月前 回复

bxyfighting

同问

5天前

usleep iOS @ 家里蹲

现实中遇到了各种没有工程代码的崩溃堆栈 😅

1月前 回复

用户9408937701142

优秀，能不能来一波汇编教程 😄

1月前 回复

好东西

😭😭 这也太秀了吧

1月前 回复

相关推荐

goyohol 11小时前

UILabel的自适应 及 富文本

UILabel的自适应 UILabel的自动换行 NSLineBreakMode枚举： 几种UILabel高度 对应的效果： 效果：装不下 (装不完) 效果：装下，剩一点点空间 效果：装下，多出了很多

点赞 1 iOS

村村村村村村村村村长 9小时前

PINCache源码解读

PINCache源码逻辑学习, 主要是学习框架设计思路, 接口设计, 以及内部具体实现, 不同的数据结构对效率的影响..等等, 如果没有好的学习方式 , 源码肯定是最直接的答案

掘金酱 2天前

6月更文挑战 | 这一次，豪华大礼祝你扶稳Flag！

在6月1日～6月30日内，在掘金社区更新原创文章，就有机会获得奖品，更文天数越多，奖品越丰厚，Gopro、iwatch等大奖等你拿～

28 33 后端 前端

CoderStar 23小时前

iOS编译简析

前言 一般的编译器都是由三部分构成.从源码到机器码基本上都要经过这三部分. 编译器前端(FrontEnd): 词法分析，语法分析，语义分析，将源代码抽象为语法树 AST，继而生成中间代码 IR。 优化

3 评论 iOS

goyohol 9小时前

App上架

因为翻出了以前总结的截图，就索性 就把“App上架”的截图 也一起拿出来，总结一篇 App上架的文章~

1 1 iOS

goyohol 11小时前

iOS10之后 权限设置

iOS10之后，苹果对权限问题限制得更加严厉了。 之前在APP里面还可以跳转到手机的设置页面，而现在不可以了！！ 足以说明🍎对APP权限问题的重视！ （其实还蛮 讨人烦的~） 若要了解跳 “iOS 10

点赞 评论 iOS

猿大鑫 1天前

能否与安卓、iOS抗衡？鸿蒙OS+华为P40第一时间上手体验

由内测转为了开发者....

1 评论 iOS

goyohol 11小时前

呼吸按钮 及 警告忽略(粗暴)

呼吸效果：传说中的“呼吸__灯__效果 ”，美观又起到了提示效果 gif图 只有单一的“呼”(或者“吸”)效果，没有对应的“吸”(或许“呼”)效果！见谅~~ 亮度到达最高值，瞬间变为最低值(个人觉得会

点赞 评论 iOS

huangjinsheng 3天前

Objective-C KVC 底层原理探究

我们都知道苹果对于KVC(Key-Value-Coding)部分的源码是不开源的，因此我们不能从源码中直接了解KVC的底层实现原理，但我们可以从苹果的官方文档中去学习并探索其底层实现原理，记住当没有开

3 评论 iOS

goyohol 11小时前

UIWebView的使用

WebView的几种使用方法： 首先：配置Plist文件 可以进行网络请求 或者：Source Code写入HTML格式 否则:你懂得~~😂 使用StoryBoard拖3个WebView，平分了整个屏

点赞 评论 iOS