Vong

WWDC 2018：理解崩溃以及崩溃日志

2019-02-22 | 原创

> 本文首发于掘金。

人非圣贤，孰能无过。每个人在写代码的时候，或多或少都会犯错，那么如何调试、找出问题所在呢？让我们跟随苹果工程师一起了解一下崩溃是如何产生以及如何解决它们的吧。

1. 基础知识

崩溃是什么？崩溃是当应用想要做某件事的时候，被意外终止。

1.1 崩溃为什么会发生

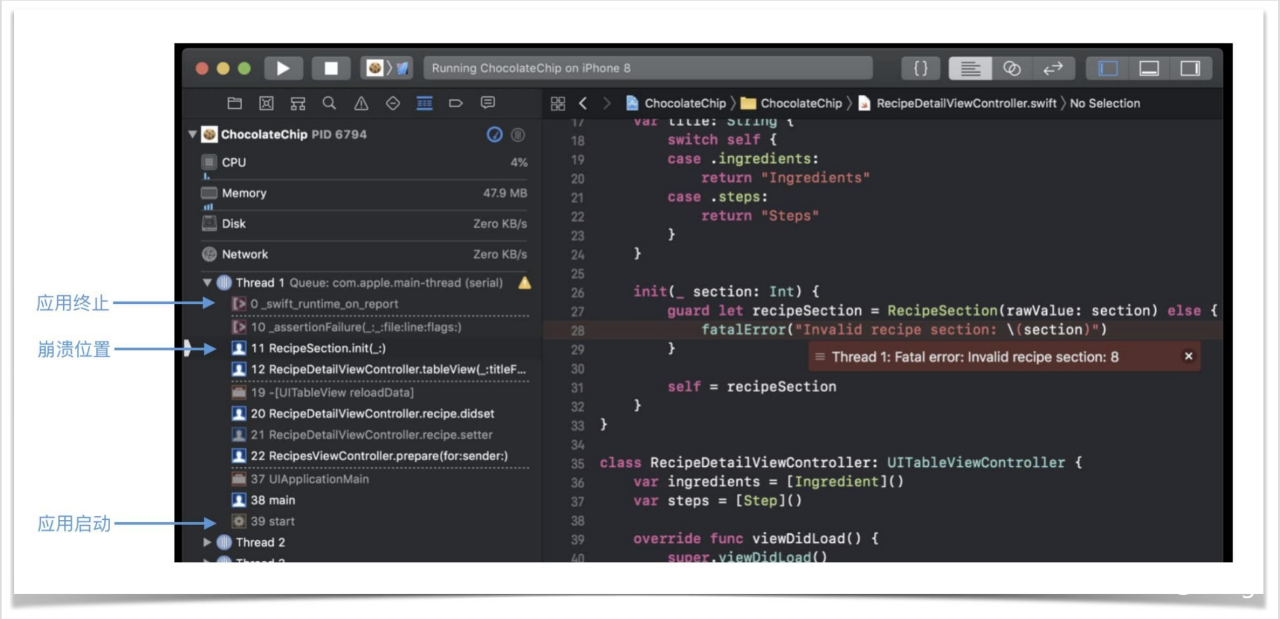
主要是以下几方面原因

- CPU 无法执行的代码。
- 被操作系统“强杀”，系统为了用户体验，会强制终止掉那些卡顿时间过长或者内存消耗过高的应用。
- 编程语言为了防止错误发生而触发的崩溃，如 `NSArray` 或者 `Swift.Array` 越界
- 开发者为了防止错误发生而触发的崩溃，比如一些非空判断的断言

1.2 崩溃长什么样子

1.2.1 调试器里

当我们连接着 `Xcode` 进行调试的时候，遇到崩溃，大概长这个样子。



当连着调试器的时候，我们能够拿到崩溃现场的一些调用栈以及对应的方法，当没有连着调试器的时候，系统会将崩溃日志存储到磁盘当中。

1.2.2 崩溃日志里

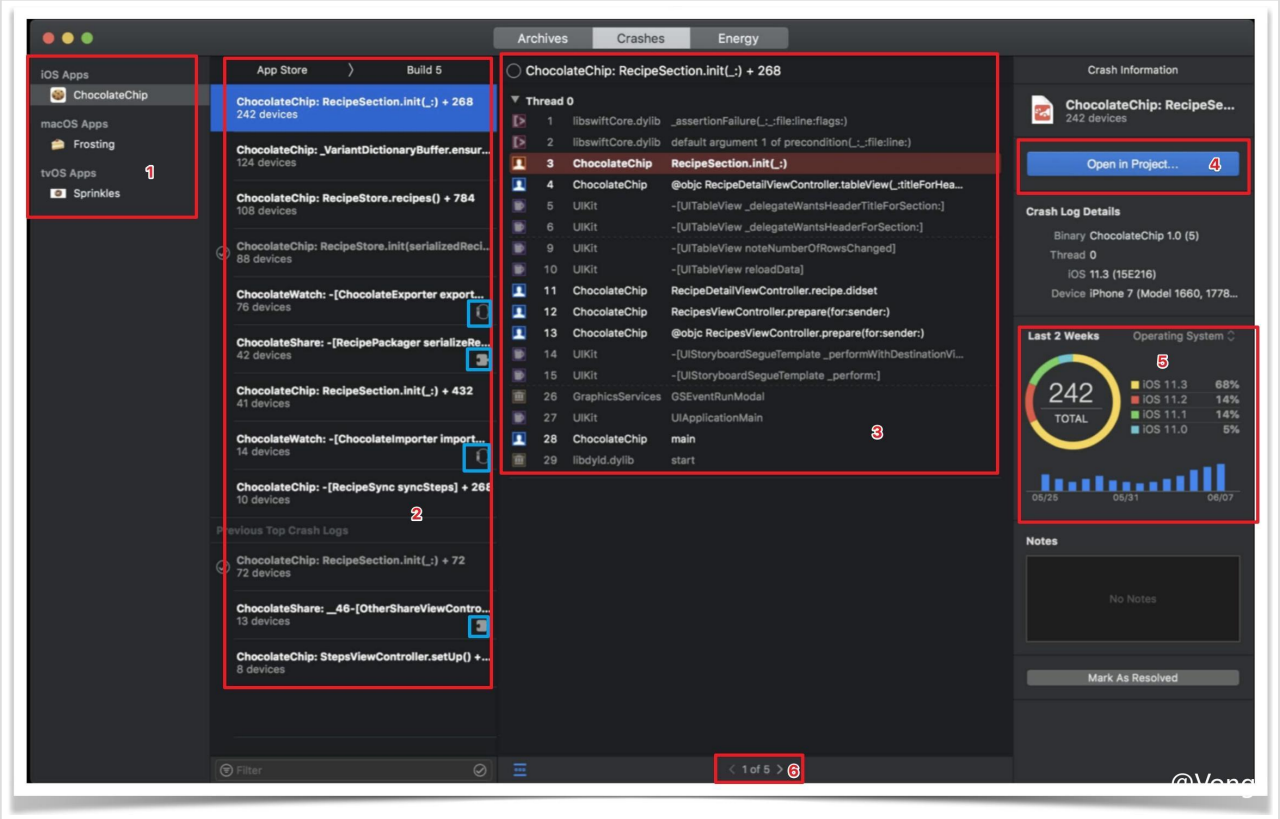
通常情况下， `release` 模式的应用的崩溃日志是没有符号化的，日志内记录的都是地址。我们可以通过 `Xcode` 来将崩溃日志进行符号化，解析出对应文件名、方法名以及对应崩溃在第几行。

1.3 获取崩溃日志

获取崩溃日志的方式很多，我们先来了解一下如何通过 `Xcode Organizer` 来获取从 `TestFlight` 或 `App Store` 下载的应用的崩溃日志。

1.3.1 Organizer Window

先来看一下下面这张图：



> 下面数字 1~6 分别代表图中标注的 1~6

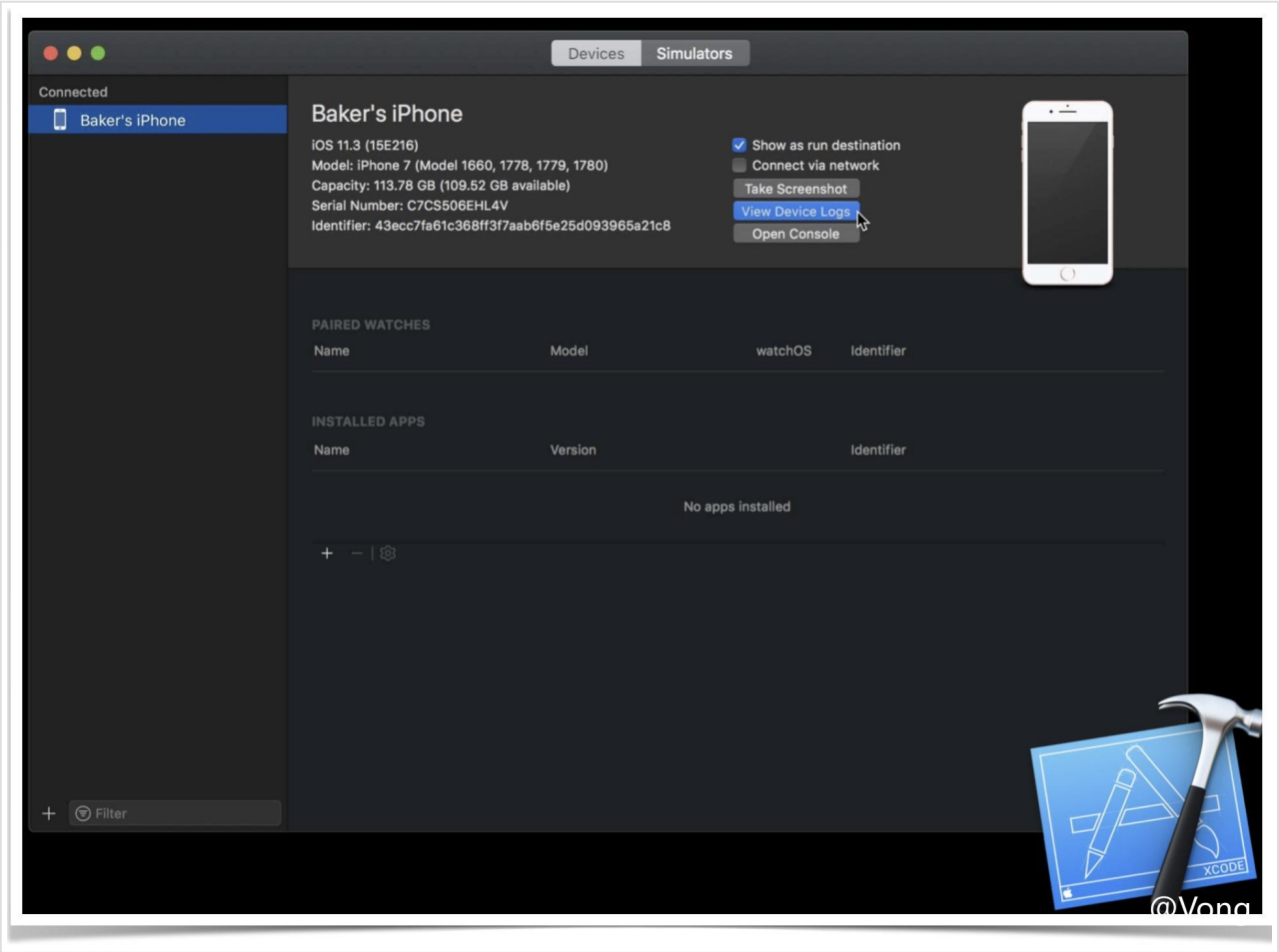
- 1.可以看到所有平台发布在 App Store 或者 TestFlight 上的应用。
- 2.崩溃日志列表，可以看到对应影响的设备数以及对应的平台、扩展（extension），如图中蓝色框标注的位置。
- 3.崩溃所在调用栈及崩溃位置的高亮。
- 4.在对应工程中打开崩溃所在的文件，并跳转到指定位置，方便追踪问题。
- 5.最近数据分析，包含系统和机型两个维度。
- 6.在崩溃数较多时，支持翻页。

PS：上面6个只是简单介绍了一下主题部分，剩余的可以自行探索使用。比如搜索、对单个日志做一些笔记、以及将已修复的崩溃标记为已解决等等。

那么如何才能在 Organizer 中获取对应的崩溃日志呢？很简单，只需要做到下面几步

- 在 Xcode 中登录已付费的开发者帐号。
- 上传应用到 App Store 或 TestFlight 时，一并上传符号文件。
- 打开 Xcode Organizer 窗口，选中 Crashes tab(快捷键: Cmd+Shift+6)。

1.3.2 Devices Window



连接上设备，打开 Xcode，使用快捷键 Cmd+Shift+2 来打开 Devices Window，选中对应设备，然后选择 View Device Logs，即可查看当前设备磁盘上的所有崩溃文件，找到应用对应的日志即可展开分析。

> 有些时候，获取到的崩溃日志并没有符号化。这个时候需要自己做一些额外操作，这里可以参考我之前在**知识小集**分享过的一个小 tip——iOS快速解析崩溃日志。

1.3.3 其它途径

- Xcode 的自动化测试（得到的是已符号化的日志）
- Mac 自带的 Console 应用，获取 Mac 或者模拟器的崩溃日志
- iOS设备可通过这种操作获取，打开【设置】->【隐私】->【分析】->【分析数据】拿到对应的未符号化的崩溃日志，然后通过系统自带的分享即可传输到对应的设备上进行分析。

1.4 符号化最佳实践

- 上传应用的符号文件，以便苹果后台可以直接符号化崩溃日志，最终得以在 Xcode Organizer 的 Crashes tab 中呈现。
- 保留应用归档文件，以便做本地符号化，只要有归档文件在， Xcode 会自动进行符号化。
- 在 Xcode Organizer 的 Archive tab 为已开启 bitcode 的应用下载 dSYM 文件。

2. 分析奔溃日志

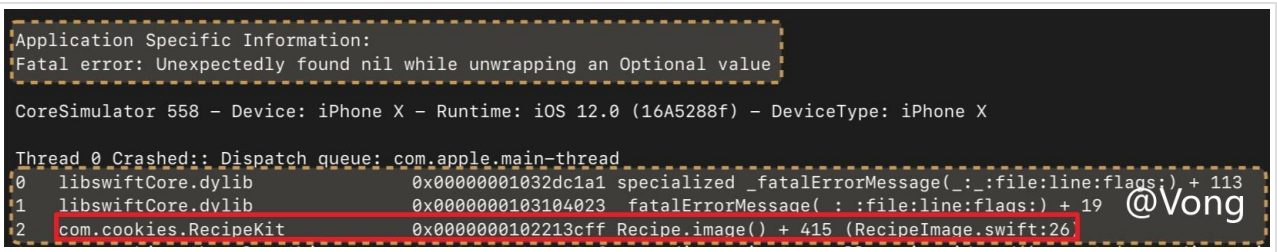
2.1 崩溃日志的组成

- 崩溃摘要，主要记录一些基本信息，比如机型、系统版本、崩溃时间等
- 崩溃原因
- 崩溃信息（这一部分在真机上处于隐私原因，一般都是不可见的，在模拟器和 MacOS 上可见）
- 崩溃线程的调用栈
- 崩溃发生时，其它线程的调用栈
- 寄存器状态
- 已加载的可执行二进制文件

2.2 如何分析

首先从崩溃原因中的崩溃类型开始

如上图的崩溃类型为 EXC_BAD_INSTRUCTION ，它代表 CPU 尝试在执行一段不存在或无效的代码，而导致进行被“杀死”。



然后我们可以找到崩溃线程的调用栈的前几行，结合崩溃信息（如果有的话）进一步分析。找到崩溃栈中第一处二进制名为应用名称所在那一行，进到对应文件对应的代码行数进行查看（如上图中标红的那一行），然后进一步分析。上图中的崩溃可以很明显看出其原因是对 nil 进行了强制解包。

2.3 断言和先决条件导致的崩溃

断言和先决条件的意义在于当错误发生时，强制终止当前进程。

上述提到的对 `nil` 强制解包导致的崩溃是断言和先决条件中的一种。而它们还包含下面几种情况：

- 数据越界访问
- 算术溢出
- 未捕获的异常
- 代码中的自定义断言

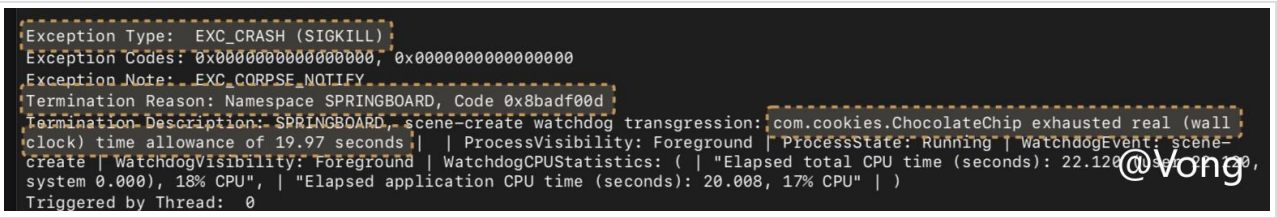
2.4 操作系统“杀死”应用导致的崩溃

某些情况下，系统处于保护目的，会将一些异常的应用“杀死”。以下几种场景可能触发系统将应用“杀死”：

- 看门狗事件，主线程长时间无响应
- 设备过度发烫
- 内存消耗殆尽
- 非法的应用签名

以上几种场景导致的崩溃，其崩溃日志可以在上面提到的 `Device Window` 中查看， `Organizer Window` 并不一定能够收集到这些日志。更多细节可以参考苹果的这个技术讲座 Understanding and Analyzing Application Crash Reports。

先来看一个关于看门狗的例子。



上面的崩溃类型为 `EXC_CRASH (SIGKILL)`， `SIGKILL` 一般代表的是系统终止了进程的运行，这种信号无法被应用捕获，进而也就无法处理。终止原因为 `Namespace SPRINGBOARD, Code 0x8badf00d` ，如果你有查看上面提到的关于崩溃日志的讲座，你应该会知道 `Code 0x8badf00d` 代表什么。从终止描述中来看，是由于启动时长超过了 19.97 秒。

> 这次总算知道为什么看门狗对应的 `code` 是 `0x8badf00d` 了，从这次苹果工程师的发音上来看，这个 `code` 的发音同 `ate bad food` 。

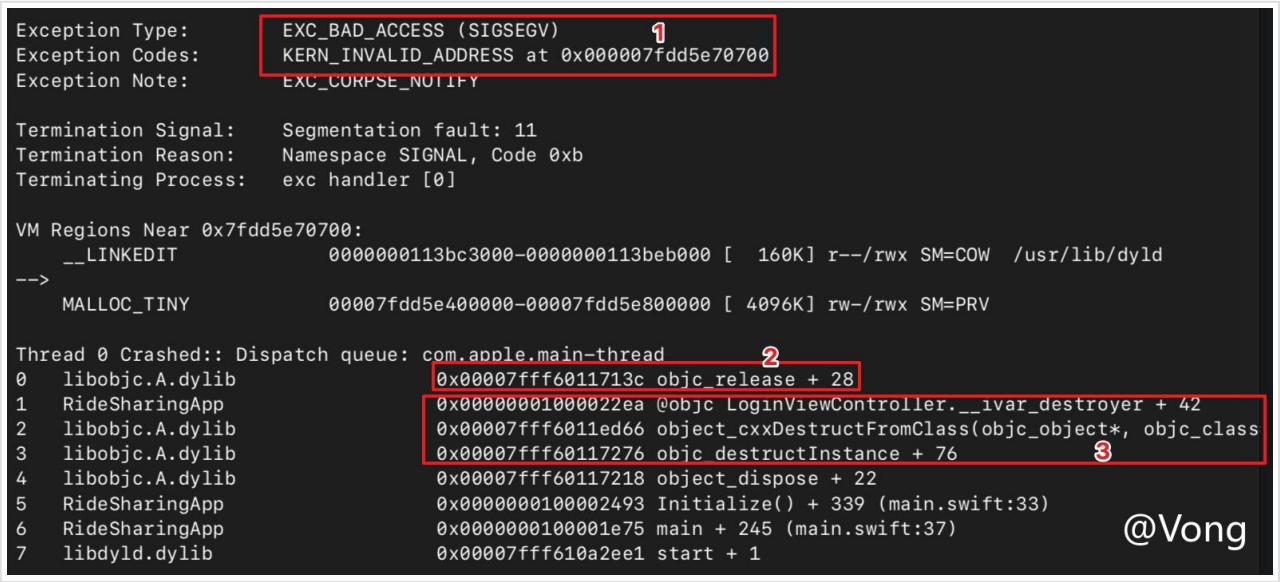
2.4.1 如何避免启动超时

应用审核被拒的比较常见的原因就包含启动超时这一项。那么如何来避免这种情况发生呢？苹果工程师给了我们这些建议：

- 在真机上测试，因为看门狗在模拟器以及调试阶段是被禁用的
- 在低性能设备上测试，高性能设备响应肯定会快，无法体现出真实效果

2.4.2 如何避免内存问题

常见的内存错误包含：过度释放、野指针（访问已释放对象）、内存访问越界（比如 C 数组）。我们还是通过一个日志来分析一下具体问题。



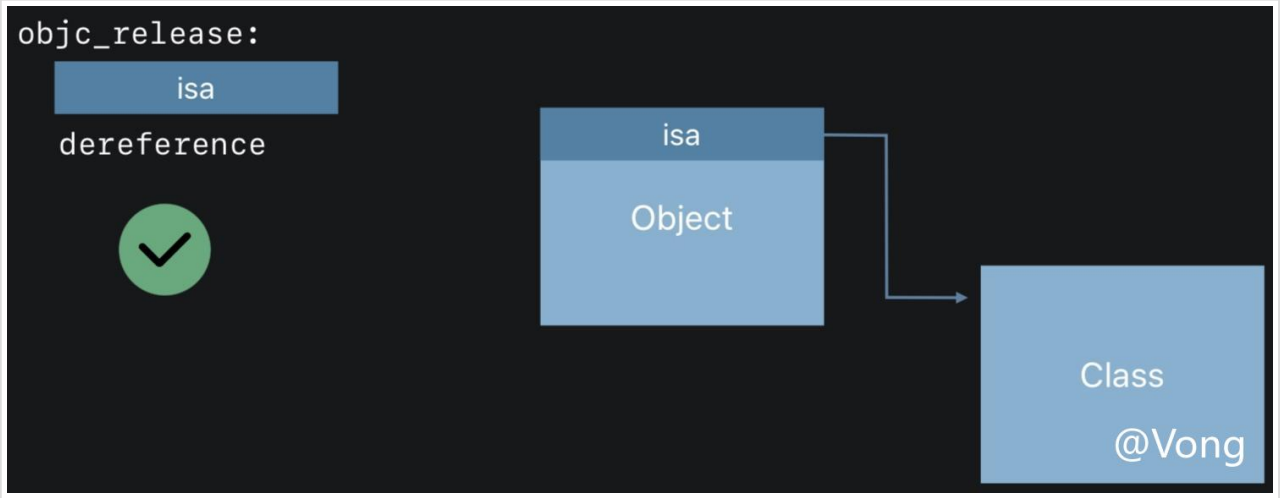
由上图中标注的1，我们知道崩溃类型为 EXC_BAD_ACCESS(SIGSEGV)，这种类型崩溃主要是有两种情况导致：

- 对只读的内存地址进行写操作
- 访问不存在的内存地址

通过崩溃栈中的 objc_release 、 object_dispose 等，我们更加确定这是由于内存问题导致的崩溃。我们通过这几个线索可以知道， LoginViewController 实例在调用 deinit 方法销毁相关属性的时候，发生了内存问题，进而导致崩溃的产生。

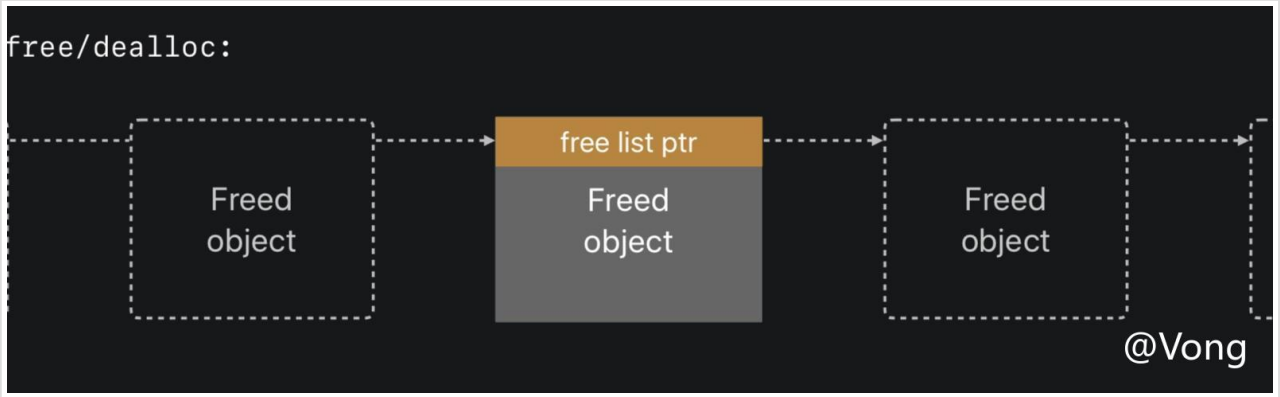
我们回到日志的第一部分中的 Exception Codes ， 苹果的工程师说可以根据经验以及日志中的相关信息得出结论，对应的 BAD_ADDRESS 为 0x7fdd5e70700 。原因是 0x7fdd5e70700 刚好在日志中的这一段 MALLOC_TINY 00007fdd5e400000-00007fdd5e800000 地址范围内。

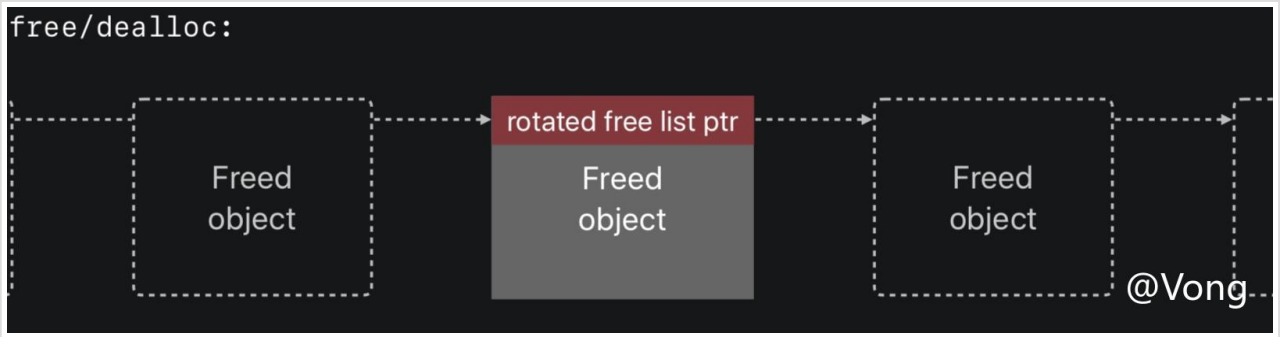
一些关于内存及释放的基础



Objective-C 对象以及一些 Swift 对象的内存布局如图，当一个对象有效（未释放）时以 isa 开始， isa 指向它所属的类。 objc_release 主要是读取对象的 isa 指针，然后将 isa 指针解除对 Class 的引用。

正常情况下，一切都能照常工作。如果对象已经被释放，会发生什么呢？ free 函数调用后，会将对象删除，并且将其插入到包含了其它已释放对象组成的链表中，同时将之前 isa 区域指向链表中下一个已释放对象。





当之前的 isa 内存区域被写入成 rotated free list 指针时，意味着访问这个地址返回的将是一个无效的内存地址，进而导致崩溃。所以当 objc_release 去解除 isa 引用时，访问到的是 rotated free list ，所以崩溃就发生了。

所以可以分析出，肯定是在释放某个属性时，该属性已经被释放。我们能知道具体是哪个属性导致的么？答案是肯定的。

目前从崩溃的那一行来看， `__ivar_destroyer` 是编译器帮我们自动生成的函数，所以我们无从知晓具体是哪一行导致的问题。我们只知道这个类有如图三个属性：

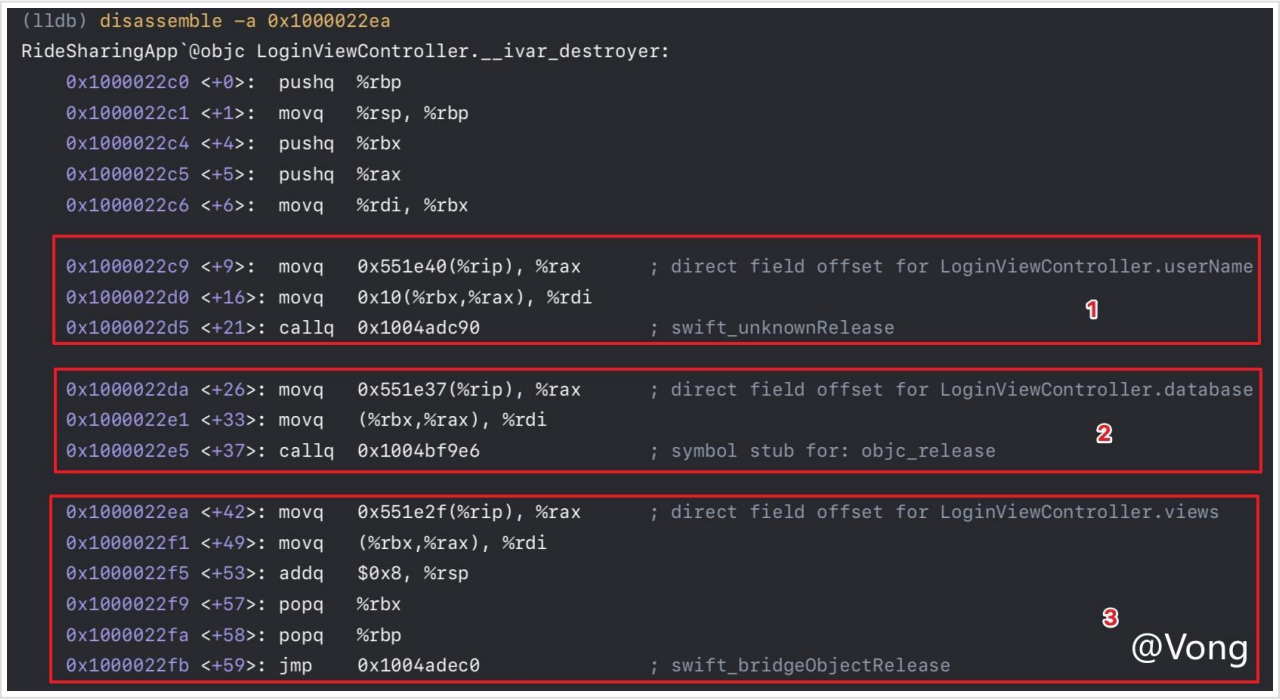
```
// LoginViewController.swift

class LoginViewController: UIViewController {
    var userName: String
    var database: DatabaseProxy
    var views: [UIView]
    ...
}
```


但是从 `@objc LoginViewController.__ivar_destroyer + 42` 可以获取到一些信息， `+42` 代表着汇编里面的该函数的偏移量。我们可以对 `__ivar_destroyer` 函数进行反汇编，然后看偏移量为42对应获取的是哪个属性，在 Xcode 中可以使用 `lldb` 调试。

```
(lldb) command script import lldb.macosx.crashlog
"crashlog" and "save_crashlog" command installed...
...
(lldb) crashlog /Users/…/RideSharingApp-2018-05-24-1.crash
...
Thread[0] EXC_BAD_ACCESS (SIGSEGV) (0x000007fdd5e70700)
[ 0] 0x00007fff6011713c libobjc.A.dylib`objc_release + 28
[ 1] 0x00000001000022ea RideSharingApp`@objc LoginViewController.__ivar_destroyer + 42
[ 2] 0x00007fff6011ed66 libobjc.A.dylib`object_cxxDestructFromClass + 127
[ 3] 0x00007fff60117276 libobjc.A.dylib`objc_destructInstance + 76
[ 4] 0x00007fff60117218 libobjc.A.dylib`object_dispose + 22
[ 5] 0x0000000100002493 RideSharingApp`Initialize (main.swift:33)
[ 6] 0x0000000100001e75 RideSharingApp`main (main.swift:37)
[ 7] 0x00007fff610a2ee1 libdyld.dylib`start + 1
...
(lldb)
```


断点后分别输入上图中黄色字的命令，分别为 `command script import lldb.macosx.crashlog` ， `crashlog /Users/.../RideSharingApp-2018-05-24-1.crash` ，后面的路径需要替换成你的崩溃日志路径。 Xcode 会自动检索二进制文件以及对应的 dSYM 文件，然后符号化显示在 `lldb` 控制台中。然后我们找到崩溃处的地址，执行如下命令，即可得到对应的反汇编代码：



我们不需要理解每一行汇编的意思，每行后面的注释可以帮助我们理解，根据注释可以知道 1、2、3 处代码分别代表着 `userName` 、 `database` 、 `views` 的释放。回到上面提到的 `+42` ，我们找到第3处的第一行，有一点需要注意的是大部分情况下汇编的偏移地址是返回地址，所以调用 `objc_release` 是在上一行。所以可以判断出是在释放 `database` 时出现了问题。虽然我们目前还不知道具体问题所在，但是可以通过这些信息缩小查找问题的范围，可以查找使用到 `database` 的地方，来找到真正的问题所在。

2.4.2 日志分析总结

- 理解崩溃日志产生的原因
- 检查崩溃栈信息
- 使用反汇编帮我们找到更多线索来分析 `bad address` 问题

2.4.3 常见内存错误

- objc_msgSend 或者 retain/release 崩溃

- 无法识别的方法异常

- abort() inside malloc/free

2.5 日志分析建议

- 不要只关注崩溃发生的那一行代码，多查看一下和崩溃相关的代码，比如上面那个崩溃代码并不是真正导致 bug 出现的原因
- 查看所有调用栈，不要只关注崩溃所在线程的调用栈，非崩溃线程调用栈可以帮助我们查看崩溃时应用所处状态
- 多查看一些崩溃日志，有些时候很多崩溃日志都是崩溃在同一个地方，但是某些崩溃日志会包含更多的信息
- 使用 Xcode 提供的工具来复现内存问题，比如 Address Sanitizer 或者 Zombies

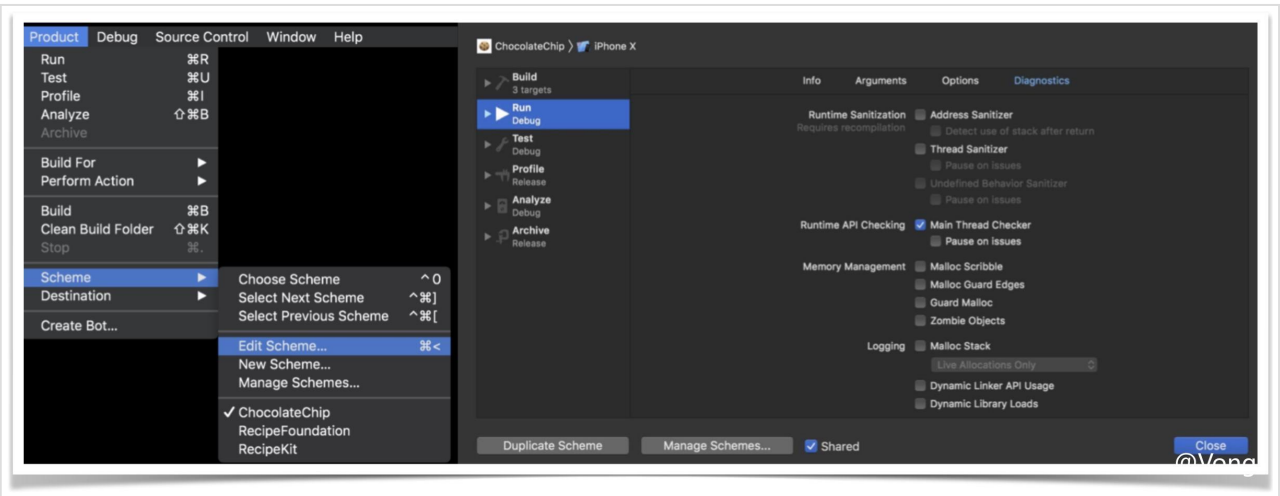
3. 多线程问题

3.1 崩溃日志中多线程问题的一些“症状”

- 最难复现和诊断的一类 bug
- 多线程问题通常会引起内存竞争
- 多个线程执行着相似代码
- 同一个 bug 可能会有不同的崩溃日志

3.2 使用 Thread Sanitizer 检测多线程问题

多线程问题即使我们拿到日志大概率情况下也无法分析问题所在，即使连着 Xcode 调试也不一定能够稳定复现，即使运气好能复现也可能分析不出具体问题。所以我们可以借助 Xcode 提供的工具来帮我们分析，这个工具就是 Thread Sanitizer 。通过快捷键 Cmd+shift+, ，然后选则 Diagnostics tab，勾选 Thread Sanitizer 即可。如下图所示



- 可稳定复现多线程 bug
- 在模拟器下也可进行

- 只查找当前正在执行的代码的问题

3.3 实用建议

在创建 GCD Queue 、(NS)OperationQueue 、(NS)Thread 时，使用自定义名称，方便后续调试以及崩溃日志内查看。

```
let queue = DispatchQueue(label: "com.example.myapp.networking")

let operationQueue = OperationQueue()
operationQueue.name = "Networking OperationQueue"

let thread = Thread(...)
thread.name = "Networking Thread"
```

3.4 额外建议

- 使用真机测试
- 尝试复现，从用户处拿到崩溃日志后根据调用栈尝试去复现问题
- 使用工具来查找难以复现的 bug，下面两个工具的更多使用方式可以参考 WWDC 2016 Session 412 Thread Sanitizer and Static Analysis
  - 使用 Address Sanitizer 来查看内存问题
  - 使用 Thread Sanitizer 来查看多线程问题

Buy Me A Cup Of Coffee!

赏

WWDC

‹ 时光的河如海流 WWDC 2018：使用日志框架测量性能 ›

显示 Gitment 评论