

32 | 热点问题答疑 (三)

戴铭 2019-05-23



00:00

09:35

讲述：冯永吉 大小：8.79M

你好，我是戴铭。

这是我们《iOS 开发高手课》专栏的第三期答疑文章，我将继续和你分享大家在学习前面文章时遇到的最普遍的问题。

今天，我在这段时间的留言问题中，挑选了几个 iOS 开发者普遍关注的问题，在这篇答疑文章里来做一个统一回复。

A/B 测试 SDK

@鼠辈同学在第 24 篇文章 [《A/B 测试：验证决策效果的利器》](#) 留言中问道：

最近一直在找一个好的 A/B 测试的 SDK，不知道作者之前用过什么好的 A/B 测试的 SDK（三方的，可以后台控制的）

我认为带后台功能的 A/B 测试 SDK 没什么必要，原因有二：

1. A/B 测试本身就是为业务服务的，需要对会影响产品决策的业务场景做大量定制化开发；
2. A/B 测试功能本身并不复杂，第三方后台定制化开发，成本也不会节省多少。

因此，我推荐后台功能自己来做，端上使用我在第 24 篇文章中提到的 SkyLab 就完全没有问题了。另外，SkyLab 也可以很方便地集成到你自己的后台中。

如何衡量性能监控的优劣？

@ RiverLi 同学在第 16 篇文章 [《性能监控：衡量 App 质量的那把尺》](#) 的评论区留言问到：

对于性能监控有没有衡量标准，如何衡量优劣？

我觉得，如果给所有 App 制定相同的衡量标准是不现实的，这样的标准，也是无法落地的。为什么这么说呢，很有可能由于历史原因或者 App 的特性决定了有些 App 的性能无法达到另一个 App 的标准。又或者说，有些 App 需要进行大量的重构，才能要达到另一个 App 的性能标准，而这些重构明显不是一朝一夕就能落地执行的。特别是业务还在快跑的情况下，你只能有针对性地去优化，而不是大量的重构。

回到性能监控的初衷，它主要是希望通过监控手段去发现突发的性能问题，这也是我们再做线上性能监控时需要重点关注的。

对于 App 运行普遍存在的性能问题，我们应该在上线前就设法优化完成。因为，线下的性能问题是可控的，而线上的性能问题往往是“摸不着”的，也正是这个原因，我们需要监控线上性能问题。

因此，**性能监控的标准一定是针对 App 线下的性能表现来制定的**。比如，你的 App 在线下连续 3 秒 CPU 占比都是在 70% 以下，那么 CPU 占比的监控值就可以设置为 3 秒内占比在 70% 以下。如果超过这个阈值就属于突发情况，就做报警处理，进行问题跟踪排查，然后有针对性地进行修复问题。

关于 WatchDog


我在第 13 篇文章中讲解如何用 RunLoop 原理去监控卡顿的时候，用到了 WatchDog 机制。Xqqq0 同学在文后留言中，希望我解释一下这个机制，并推荐一些相关的学习资料。

WatchDog 是苹果公司设计的一种机制，主要是为了避免 App 界面无响应造成用户无法操作，而强杀掉 App 进程。造成 App 界面无响应的原因种类太多，于是苹果公司采用了一刀切的做法：凡是主线程卡死一定的时间就会被 WatchDog 机制强杀掉。这个卡死时间，WatchDog 在启动时设置的是 20 秒，前台时设置的是 10 秒，后台时设置的是 10 分钟。

由于 WatchDog 强杀日志属于系统日志，所以你的 App 上线后需要自己来监控卡顿，这样才能够在 WatchDog 强杀之前捕获到 App 卡死的情况。关于这部分内容的详细讲解，你可以参考苹果公司关于[崩溃分析](#)的文档。

关于 iOS 崩溃

在专栏的第 12 篇文章 [《iOS 崩溃千奇百怪，如何全面监控？》](#) 后，(Jet) 黄仲平同学提了这么几个问题。考虑到这几个问题涉及知识点比较有代表性，所以我特意在今天这篇答疑文章中和你详细展开下。

 **(Jet)黄仲平**

戴老师好，我最近有一个困惑，就是我们公司计划做一个自己的“crash 监控系统”。其中有一个很重要的目标是要实现 crash 日志，能定位到责任人，然后进行快速响应。

目前我个人对这件事的解决思路如下：

1. 收集 crash 日志，并上传日志到服务器上；
2. 通过解析 git 库的提交日志，对代码文件与作者与进行关连并把日志提交到服务器；
3. 发生 crash 时，通过把错误堆栈信息与符号表 (dysm) 进行匹配，找到堆栈信息里含特定前缀的文件名。进行责任人匹配；

第一个问题：以上是我的解决思路。不知道戴老师是否也是这种思路？

第二个问题：在上面的第一步日志收集时是通过文中“handleSignalException”方法来收集？


还是通过 PLCCrashReporter 来进行收集？

第三个问题：dysm 在解析堆栈信息时的工作原理有无相关文章 进行介绍；

望老师百忙之中帮忙答复，不胜感激。

——— 写于 2019年04月24日

引自：iOS开发高手课
12 | iOS 崩溃千奇百怪，如何全面监控？



识别二维码打开原文
「极客时间」App

关于实现崩溃问题自动定位到人，我认为通过堆栈信息来匹配到人是没有问题的。关于实现方法的问题，也就是第一个问题，你可以先做个映射表，每个类都能够对应到一个负责人，当获取到崩溃堆栈信息时，根据映射表就能够快速定位到人了。

对于第二个问题关于日志的收集方法，我想说的是 PLCCrashReporter 就是用 handleSignalException 方法来收集的。

第三个关于 dSYM 解析堆栈信息工作原理的问题，也不是很复杂。dSYM 会根据线程中方法调用栈的指针，去符号表里找到这些指针所对应的符号信息进行解析，解析完之后就能够展示出可读的方法调用栈。

接下来，**我来和你说说通过堆栈匹配到人的具体实现的问题。**

第一步，通过 task_threads 获取当前所有的线程，遍历所有线程，通过 thread_info 获取各个线程的详细信息。

第二步，遍历线程，每个线程都通过 thread_get_state 得到 machine context 里面函数调用栈的指针。

thread_get_state 获取函数调用栈指针的具体实现代码如下：

```
1 _STRUCT_MCONTEXT machineContext; // 线程栈里所有的栈指针
2 // 通过 thread_get_state 获取完整的 machineContext 信息，包含 thread 状态信息
3 const uint32_t state_count = smThreadStateCountByCPU();
4 kern_return_t kr = thread_get_state(thread, smThreadStateByCPU(), (thread_state_
5
```

获取到的这些函数调用栈，需要一个栈结构体来保存。

第三步，创建栈结构体。创建后通过栈基地址指针获取到当前帧栈地址，然后往前查找函数调用帧地址，并将它们保存到创建的栈结构体中。具体代码如下：

```
1 // 为通用回溯设计结构支持栈地址由小到大，地址里存储上个栈指针的地址
2 typedef struct SMStackFrame {
3     const struct SMStackFrame *const previous;
4     const uintptr_t return_address;
5 } SMStackFrame;
6
7 SMStackFrame stackFrame = {0};
8 // 通过栈基址指针获取当前帧栈地址
9 const uintptr_t framePointer = smMachStackBasePointerByCPU(&machineContext);
10 if (framePointer == 0 || smMemCopySafely((void *)framePointer, &stackFrame, size
11     return @"Fail frame pointer";
12 }
13 for (; i < 32; i++) {
14     buffer[i] = stackFrame.return_address;
15     if (buffer[i] == 0 || stackFrame.previous == 0 || smMemCopySafely(stackFrame
16         break;
17     }
18 }
19
```

第四步，根据获取到的栈帧地址，找到对应的 image 的游标，从而能够获取 image 的更多信息。代码如下：

```
1 // 初始化保存符号结果的结构体 DL_info
2 info->dli_fname = NULL;
3 info->dli_fbase = NULL;
4 info->dli_sname = NULL;
5 info->dli_saddr = NULL;
6
7 // 根据地址获取是哪个 image
8 const uint32_t idx = smDyldImageIndexFromAddress(address);
9 if (idx == ULINK_MAX) {
10     return FALSE;
11 }
12
```

第五步，在知道了是哪个 image 后，根据 Mach-O 文件的结构，要想获取符号表所在的 segment，需要先找到 Mach-O 里对应的 Header。通过 _dyld_get_image_header 方法，我们可以找到 mach_header 结构体。然后，使用 _dyld_get_image_vmaddr_slide 方法，我们就能够获取虚拟内存地址 slide 的数量。而动态链接器就是通过添加 slide 数量到 image 基地址，以实现将 image 映射到未占用地址的进程虚拟地址空间来加载 image 的。具体实现代码如下：

```
1 /*
2  Header
3  -----
4  Load commands
5  Segment command 1 -----|
6  Segment command 2 -----|
7  -----|
8  Data -----|
9  Section 1 data |segment 1 <----|
10 Section 2 data | <----|
11 Section 3 data | <----|
12 Section 4 data |segment 2
13 Section 5 data |
14 ... |
15 Section n data |
16 */
17 /*-----Mach Header-----*/
18 // 根据 image 的序号获取 mach_header
19 const struct mach_header* machHeader = _dyld_get_image_header(idx);
20
21 // 将 header 的名字和 machHeader 记录到 DL_info 结构体里
22 info->dli_fname = _dyld_get_image_name(idx);
23 info->dli_fbase = (void*)machHeader;
24
25 // 返回 image_index 索引的 image 的虚拟内存地址 slide 的数量
26 // 动态链接器就是通过添加 slide 数量到 image 基地址，以实现将 image 映射到未占用地址的进程
27 const uintptr_t imageVMAddressSlide = (uintptr_t)_dyld_get_image_vmaddr_slide(ic
28
```

第六步，计算 ASLR（地址空间布局随机化）偏移量。

ASLR 是一种防范内存损坏漏洞被利用的计算机安全技术，想详细了解 ASLR 的话，你可以看看它的[Wiki 页面](#)。

通过 ASLR 偏移量可以获取 segment 的基地址，segment 定义 Mach-O 文件中的字节范围以及动态链接器加载应用程序时这些字节映射到虚拟内存中的地址和内存保护属性。所以，segment 总是虚拟内存页对齐。

```
1 /*-----ASLR 偏移量 -----*/
2 // https://en.wikipedia.org/wiki/Address_space_layout_randomization
3 const uintptr_t addressWithSlide = address - imageVMAddressSlide;
4 // 通过 ASLR 偏移量可以获取 segment 的基地址
5 // segment 定义 Mach-O 文件中的字节范围以及动态链接器加载应用程序时这些字节映射到虚拟内存中
6 const uintptr_t segmentBase = smSegmentBaseOfImageIndex(idx) + imageVMAddressSlid
7 if (segmentBase == 0) {
8     return false;
9 }
10 }
11
```

第七步，遍历所有 segment，查找目标地址在哪个 segment 里。

除了 __TEXT segment 和 __DATA segment 外，还有 __LINKEDIT segment。__LINKEDIT segment 里包含了动态链接器使用的原始数据，比如符号、字符串、重定位表项。LC_SYMTAB 描述的是，__LINKEDIT segment 里查找的字符串在符号表的位置。有了符号表里字符串的位置，就能找到目标地址对应的字符串，从而完成函数调用栈地址的符号化。

这个过程的详细实现代码如下：

```
1 /*-----Mach Segment-----*/
2 // 地址最匹配的 symbol
3 const nlistByCPU* bestMatch = NULL;
4 uintptr_t bestDistance = 0;
5 uintptr_t cmdPointer = smCmdFirstPointerFromMachHeader(machHeader);
6 if (cmdPointer == 0) {
7     return false;
8 }
9 // 遍历每个 segment 判断目标地址是否落在该 segment 包含的范围里
10 for (uint32_t iCmd = 0; iCmd < machHeader->ncmds; iCmd++) {
11     const struct load_command* loadCmd = (struct load_command*)cmdPointer;
12     /*----- 目标 Image 的符号表 -----*/
13     // Segment 除了 __TEXT 和 __DATA 外还有 __LINKEDIT segment，它里面包含动态链接器信息
14     // LC_SYMTAB 描述了 __LINKEDIT segment 内查找字符串和符号表的位置
15     if (loadCmd->cmd == LC_SYMTAB) {
16         // 获取字符串和符号表的虚拟内存偏移量。
17         const struct symtab_command* symtabCmd = (struct symtab_command*)cmdPoir
18         const nlistByCPU* symbolTable = (nlistByCPU*)(segmentBase + symtabCmd->st
19         const uintptr_t stringTable = segmentBase + symtabCmd->stroff;
20
21         for (uint32_t iSym = 0; iSym < symtabCmd->nsyms; iSym++) {
22             // 如果 n_value 是 0，symbol 指向外部对象
23             if (symbolTable[iSym].n_value != 0) {
24                 // 给定的偏移量是文件偏移量，减去 __LINKEDIT segment 的文件偏移量获得字
25                 uintptr_t symbolDistance = symbolTable[iSym].n_value;
26                 uintptr_t currentDistance = addressWithSlide - symbolBase;
27                 // 寻找最小的距离 bestDistance，因为 addressWithSlide 是某个方法的指
28                 // 离 addressWithSlide 越近的函数入口越匹配
29                 if ((addressWithSlide >= symbolBase) && (currentDistance <= best
30                     bestMatch = symbolTable + iSym;
31                     bestDistance = currentDistance;
32                 }
33             }
34         }
35         if (bestMatch != NULL) {
36             // 将虚拟内存偏移量添加到 __LINKEDIT segment 的虚拟内存地址可以提供字符串和
37             info->dli_saddr = (void*)(bestMatch->n_value + imageVMAddressSlide);
38             info->dli_sname = (char*)((intptr_t)stringTable + (intptr_t)bestMatc
39             if (*info->dli_sname == '_') {
40                 info->dli_sname++;
41             }
42             // 所有的 symbols 的已经被处理好了
43             if (info->dli_saddr == info->dli_fbase && bestMatch->n_type == 3) {
44                 info->dli_sname = NULL;
45             }
46             break;
47         }
48     }
49     cmdPointer += loadCmd->cmdsize;
50 }
51
```