读本篇文章之前，需要先阅读《Mach-O文件格式》、《函数调用栈》这两篇文章。

工作中其实和线程调用栈打交道的机会挺多，使用Xcode调试时就可以看到当前程序的所有线程调用栈。当应用程序发布出去时，偶尔我们也想知道用户某时刻的线程调用栈，以方便解决问题。比如发生崩溃时、发生卡顿时，如果此时有用户的线程调用栈，开发者解决问题会容易很多。

iOS获取线程调用栈

iOS系统如何获取线程调用栈呢？首先想到的是系统API：

```
Thread.callstackSymbols
```

遗憾的是，该API只能获取到当前线程的调用栈。有经验的开发者肯定知道，发生崩溃、卡顿的原因可能并不是当前线程，很有可能是其他子线程某个操作引起了问题。因此，仅仅获取当前线程的调用栈还不能满足需求。

既然系统API不行，那就要想起他的办法。考虑一下，获取线程调用栈需要哪几步？

获取线程调用栈步骤拆分

上面说了，需求是要获取到所有线程的调用栈。因此第一步，我们需要获取到当前程序内的所有线程，这一步是没有任何疑问的。

获取到线程后，需要获取该线程的调用栈。根据《函数调用栈》这篇文章所述，只要获取到线程**最顶部**的函数，以及esp和ebp指针，就可以递归的获取到所有函数的调用关系，也就得到了该线程的调用栈。

注意上一步得到的起始都是一个地址，开发者仅仅是根据一个地址肯定是不能解决问题的。而是应该将地址解析成对应的符号、对应的字符串。实际上，一个App的运行依赖于多个镜像文件，包含应用的可执行文件、系统动态库、framework中的二进制文件等。比如Foundation、ImageIOKit、libdispatch.dylibv 都是单独的镜像。因此，需要确定上一步得到的地址对应的是哪个镜像。

确定镜像后，读取该镜像文件（文件格式是Mach-O），还是根据地址，和符号表、字符串表进行匹配，确定符号名称。

好了，总结一下上述步骤：

1. 获取所有线程
2. 获取对应线程最顶层函数，以及esp、ebp指针
3. 根据地址定位镜像
4. 根据地址、镜像文件，定位符号

是不是很简单？下面，看一下每一步到底是怎么实现的。

获取所有线程

Mach内核提供了获取所有线程的接口，如下：

```
kern_return_t task_threads
(
    task_inspect_t target_task,
    thread_act_array_t *act_list,
    mach_msg_type_number_t *act_listCnt
);
```

该函数的作用是：

```
target_task 任务中的所有线程保存在 act_list 数组中，数组中包含 act_listCnt 个线程
```

task_threads函数中有个参数是target_task，代表当前的任务。Mach内核同样提供了接口获取当前的任务，如下：

```
使用mach_task_self()获取当前进程标记 target_task
```

至此，已经可以获取到所有线程了。

完整代码如下：

```
thread_act_array_t threads;
mach_msg_type_number_t thread_count = 0;
const task_t this_task = mach_task_self();

kern_return_t kr = task_threads(this_task, &threads,
&thread_count);
if(kr != KERN_SUCCESS) {
    return @"Fail to get information of all threads";
}
```

所有线程都被保存到了threads数组中。

获取esp、ebp指针

获取某个线程最顶层的函数，以及esp、ebp指针，所依赖的仍然是Mach内核暴露出的接口。函数是：

```
kern_return_t thread_get_state
(
    thread_act_t target_act,   //目标线程，通过task_threads接口来获取
    thread_state_flavor_t flavor,  //线程状态类型，如
[ARM/x86]_THREAD_STATE64
    thread_state_t old_state,  //线程状态信息，可获取线程调用栈寄存器信息
    mach_msg_type_number_t *old_stateCnt  //线程状态信息成员数目
);
```

参数target_act就是上一步获取到的单个线程。

thread_state_t 中就包含了esp、ebp指针。thread_state_t的定义如下：

```
_STRUCT_X86_THREAD_STATE64
{
    __uint64_t  __rax;
    __uint64_t  __rbx;
    __uint64_t  __rcx;
    __uint64_t  __rdx;
    __uint64_t  __rdi;
    __uint64_t  __rsi;
    __uint64_t  __rbp;  //帧指针
    __uint64_t  __rsp;  //栈指针
    __uint64_t  __r8;
    __uint64_t  __r9;
    __uint64_t  __r10;
    __uint64_t  __r11;
    __uint64_t  __r12;
    __uint64_t  __r13;
    __uint64_t  __r14;
    __uint64_t  __r15;
    __uint64_t  __rip;  //当前线程指令地址
    __uint64_t  __rflags;
    __uint64_t  __cs;
    __uint64_t  __fs;
    __uint64_t  __gs;
};
```

实际上在编码过程中，获取esp、ebp指针使用的结构体是_STRUCT_MCONTEXT,_STRUCT_MCONTEXT有一个属性__ss就是thread_state_t类型。

看一下_STRUCT_MCONTEXT的定义：

```
#if defined(__x86_64__)
    _STRUCT_MCONTEXT ctx;
    mach_msg_type_number_t count = x86_THREAD_STATE64_COUNT;
    thread_get_state(thread, x86_THREAD_STATE64,
(thread_state_t)&ctx.__ss, &count);

    uint64_t pc = ctx.__ss.__rip;
    uint64_t sp = ctx.__ss.__rsp;
    uint64_t fp = ctx.__ss.__rbp;
#elif defined(__arm64__)
    _STRUCT_MCONTEXT ctx;
    mach_msg_type_number_t count = ARM_THREAD_STATE64_COUNT;
    thread_get_state(thread, ARM_THREAD_STATE64,
(thread_state_t)&ctx.__ss, &count);

    uint64_t pc = ctx.__ss.__pc;
    uint64_t sp = ctx.__ss.__sp;
    uint64_t fp = ctx.__ss.__fp;
#endif
```

获取到所有线程后，对于每一个线程，可以用thread_get_state方法获取线程的所有信息，信息填充在 _STRUCT_MCONTEXT类型的结构体中。在 _STRUCT_MCONTEXT结构体中，存储了当前线程**最顶部的栈指针和帧指针**，利用这些信息，可以获取到所有线程的调用栈。

完整代码如下：

```
bool fillThreadStateIntoMachineContext(thread_t thread,
_STRUCT_MCONTEXT *machineContext) {
    mach_msg_type_number_t state_count = BS_THREAD_STATE_COUNT;
    kern_return_t kr = thread_get_state(thread, BS_THREAD_STATE,
(thread_state_t)&machineContext->__ss, &state_count);
    return (kr == KERN_SUCCESS);
}
```

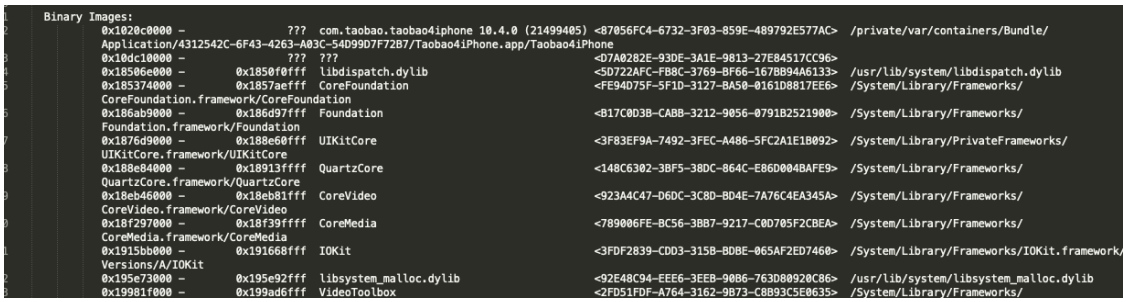
栈帧信息保存到了machineContext中。

对应到镜像文件

上面已经提到了镜像文件，这里再说一下。

一个App顺利运行依赖于很多的系统库，这些库，包括可执行文件本身，在系统看来都是镜像文件。比如Foundation、ImageIOKit、libdispatch.dylibv 都是单独的镜像。

实际上在系统生成的崩溃日志中，就有很多的镜像。



如上图的日志中，出现的镜像有 Foundation、UIKitCore、IOKit等。

每个镜像文件都对应一个地址范围，且每个镜像文件都是Mach-O格式的文件。

上一步根据栈帧指针得到的是方法的地址，首先需要判断该方法属于哪个镜像文件。

判断逻辑很简单，只需要判断该方法地址位于某个镜像文件的地址范围内即可。问题是，如果获取有多少个镜像文件，以及每个镜像文件的地址范围是多少?

dyld提供了镜像相关的接口，可以获取镜像数量、名称、地址，接口如下：

```
uint64_t count = _dyld_image_count();//image数量
const struct mach_header *header =
_dyld_get_image_header(index);//image mach-o header
const char *name = _dyld_get_image_name(index);//image name
uint64_t slide = _dyld_get_image_vmaddr_slide(index);//ALSR偏移地址
```

获取到镜像后，循环对比地址就可以。完整代码如下：

```
static uint32_t imageIndexContainingAddress(const uintptr_t
address)
{
    const uint32_t imageCount = _dyld_image_count();
    const struct mach_header* header = 0;

    for(uint32_t iImg = 0; iImg < imageCount; iImg++)
    {
```

```
        header = _dyld_get_image_header(iImg);
        if(header != NULL)
        {
            // Look for a segment command with this address within
its range.
            uintptr_t addressWSlide = address -
(uintptr_t)_dyld_get_image_vmaddr_slide(iImg);
            uintptr_t cmdPtr = firstCmdAfterHeader(header);
            if(cmdPtr == 0)
            {
                continue;
            }
            for(uint32_t iCmd = 0; iCmd < header->ncmds; iCmd++)
            {
                const struct load_command* loadCmd = (struct
load_command*)cmdPtr;
                if(loadCmd->cmd == LC_SEGMENT)
                {
                    const struct segment_command* segCmd = (struct
segment_command*)cmdPtr;
                    if(addressWSlide >= segCmd->vmaddr &&
                       addressWSlide < segCmd->vmaddr + segCmd-
>vmsize)
                    {
                        return iImg;
                    }
                }
                else if(loadCmd->cmd == LC_SEGMENT_64)
                {
                    const struct segment_command_64* segCmd =
(struct segment_command_64*)cmdPtr;
                    if(addressWSlide >= segCmd->vmaddr &&
                       addressWSlide < segCmd->vmaddr + segCmd-
>vmsize)
                    {
                        return iImg;
                    }
                }
                cmdPtr += loadCmd->cmdsize;
            }
        }
    }
    return UINT_MAX;
}
```

参数是函数的地址，返回地址是对应镜像的索引。

定位符号

镜像本身也是Mach-O格式的文件。获取到镜像文件后，就可以读取该镜像文件，获的其符号表信息、字符串表信息（包含符号数量、符号大小、字符串数量、字符串大小）。

首先是得到符号表地址、字符串表地址，代码如下：

```
//获取Mach-O Header
const struct mach_header* header = _dyld_get_image_header(index);
//通过header遍历Load Commands获取_LINKEDIT 及 LC_SYMTAB
for(uint32_t iCmd = 0; iCmd < header->ncmds; iCmd++)
{
        const struct load_command* loadCmd = (struct
load_command*)cmdPtr;
    if(loadCmd->cmd == LC_SYMTAB){
      symtabCmd = loadCmd;
    } else if(loadCmd->cmd == LC_SEGMENT_64) {
        const struct segment_command_64* segmentCmd = (struct
segment_command_64*)cmdPtr;
        if(strcmp(segmentCmd->segname, SEG_LINKEDIT) == 0)
        {
            linkeditSegment = segmentCmd;
        }
    }
}

//基址 = 偏移量 + _LINKEDIT段虚拟地址 - _LINKEDIT段文件偏移地址
uintptr_t linkeditBase = (uintptr_t)slide + linkeditSegment->vmaddr
- linkeditSegment->fileoff;
//符号表的地址 = 基址 + 符号表偏移量
const nlist_t *symbolTable = (nlist_t *)(linkeditBase + symtabCmd-
>symoff);
//字符串表的地址 = 基址 + 字符串表偏移量
char *stringTab = (char *)(linkeditBase + symtabCmd->stroff);
//符号数量
uint32_t symNum = symtabCmd->nsyms;
```

遍历符号表，首先要从load_commands中定位到符号表的位置，而symtab_command并没有给我们一个绝对的位置信息，只有一个stroff和symoff，也就是字符串表偏移量和符号表偏移量，所以我们还需要找出其真正的内存地址。而我们可以从LC_SEGMENT(__LINKEDIT)段中获取到绝对位置vmaddr和偏移量fileoff。

LC_SEGMENT(__LINKEDIT)和LC_SYMTAB结合，就可以获取到符号表、字符串表的位置。

符号表、字符串表的数据结构够可以参考《Mach-O文件格式》这篇文章的介绍。

得到符号表后，还需要定位符号。上述查找符号是获取的真正的符号内存地址和函数名，而通过函数调用栈获取的是函数内部执行指令的地址，不过该地址与真正的函数地址偏离不大，因此可以通过遍历符号的内存地址与调用栈函数地址比较得到离符号内存地址最近的最佳匹配符号，即是当前调用栈的符号。完整代码如下：

```
const uintptr_t imageVMAddrSlide =
(uintptr_t)_dyld_get_image_vmaddr_slide(idx);
const uintptr_t addressWithSlide = address -
imageVMAddrSlide;//address为调用栈内存地址
//遍历符号需找最佳匹配符号
for(uint32_t iSym = 0; iSym < symtabCmd->nsyms; iSym++)
{
    // If n_value is 0, the symbol refers to an external object.
    if(symbolTable[iSym].n_value != 0)
    {
        uintptr_t symbolBase = symbolTable[iSym].n_value;//获取符号的
内存地址(函数指针)
        uintptr_t currentDistance = addressWithSlide - symbolBase;
        if((addressWithSlide >= symbolBase) &&
        (currentDistance <= bestDistance))
        {
            bestMatch = symbolTable + iSym;//最佳匹配符号地址
            bestDistance = currentDistance;//调用栈内存地址与当前符号内
存地址距离
        }
    }
}

if(bestMatch != NULL)
{
    info->dli_saddr = (void*)(bestMatch->n_value +
imageVMAddrSlide);
    if(bestMatch->n_desc == 16)
    {
        // This image has been stripped. The name is meaningless,
and
        // almost certainly resolves to "_mh_execute_header"
        info->dli_sname = NULL;
    }
    else
    {
        //获取符号名
        info->dli_sname = (char*)((intptr_t)stringTable +
(intptr_t)bestMatch->n_un.n_strx);
        if(*info->dli_sname == '_')
        {
            info->dli_sname++;
        }
    }
}
```

总结

以上就是获取iOS线程调用栈的方法。其实真正的把问题拆分开来，获取调用栈也不难，绝大多数用到的都是系统API，只需要按照步骤一步步来就可以。

难点在于如何获取这些步骤。首先需要了解Mach-O文件的格式，知道可以根据Mach-O文件获取信息，如加载命令、符号表等；其次是要了解函数调用栈，函数调用栈是获取所有线程调用栈的基础，只有了解了函数调用栈的原理，才明白为何要这样做；最后是要知道有相关的API，无论是Mach内核还是dyld，这些函数日常开发中都很少用到，多数存在于系统源码中，这就要求对源码有一定的了解。