**FSAPM SDK 开发文档**

**修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **日期** | **概要** | **编写人** |
| V1.0.0 | 2019-01-18 | 完成初稿 |  |
| V1.0.0 | 2019-02-19 | 添加HTTPDNS |  |
| V1.0.0 | 2019-02-22 | 细节完善 |  |
|  |  |  |  |

1. **概述**

APM 的全称是 Application performance management，即应用性能管理，通过对应用的可靠性、稳定性等方面的监控，进而达到快速修复问题、提高用户体验的目的。将在以下几个方面对数据进行监控，包括 CPU 占有率，内存使用情况，磁盘使用情况，FPS，冷启动时间，卡顿，闪退，闪退防护，HTTPDNS，网络流量等。

1. **系统**

## 设备信息

**实现功能：**

* 设备型号
* 设备系统名称
* 设备系统版本
* 设备启动时间
* 设备信息

## 设备硬编码

通过设备机器名称（如“iPhone1,1”）硬编码方式输出相关数据。数据来自[维基百科](https://zh.wikipedia.org/wiki/IOS%E8%AE%BE%E5%A4%87%E5%88%97%E8%A1%A8#iPhone_%E5%9E%8B%E8%99%9F)以及[theiphonewiki](https://www.theiphonewiki.com/wiki/List_of_iPhones)。

**实现功能：**

* 设备名称
* CPU 名称
* CPU 频率
* 协处理器 名称
* 电池容量
* 电池电压
* 屏幕尺寸
* 屏幕 PPI

1. **磁盘**

**实现功能：**

* 磁盘总空间
* 磁盘使用空间
* 磁盘可用空间
* 文件大小
* 目录大小

1. **CPU**

任务（task）是一种容器（container）对象，虚拟内存空间和其他资源都是通过这个容器对象管理的，这些资源包括设备和其他句柄。严格地说，**Mach** 的任务并不是其他操作系统中所谓的进程，因为 **Mach** 作为一个微内核的操作系统，并没有提供“进程”的逻辑，而只是提供了最基本的实现。不过在 BSD 的模型中，这两个概念有1：1的简单映射，每一个 BSD 进程（也就是 OS X 进程）都在底层关联了一个 Mach 任务对象。

上面引用的是《OS X and iOS Kernel Programming》对 Mach task 的描述，Mach task 可以看作一个机器无关的 thread 执行环境的抽象一个 task 包含它的线程列表。内核提供了 task\_threads API 调用获取指定 task 的线程列表，然后可以通过 thread\_info API 调用来查询指定线程的信息。

task\_threads 将 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 个条目。

thread\_info 查询 flavor 指定的 thread 信息，将信息返回到长度为 thread\_info\_outCnt 字节的 thread\_info\_out 缓存区中，

**代码实现：**

#import <mach/mach.h>

#import <assert.h>

+ (CGFloat)appCpuUsage {

kern\_return\_t kr;

task\_info\_data\_t tinfo;

mach\_msg\_type\_number\_t task\_info\_count;

task\_info\_count = TASK\_INFO\_MAX;

kr = task\_info(mach\_task\_self(), MACH\_TASK\_BASIC\_INFO, (task\_info\_t)tinfo, &task\_info\_count);

if (kr != KERN\_SUCCESS) {

return -1;

}

thread\_array\_t thread\_list;

mach\_msg\_type\_number\_t thread\_count;

thread\_info\_data\_t thinfo;

mach\_msg\_type\_number\_t thread\_info\_count;

thread\_basic\_info\_t basic\_info\_th;

// get threads in the task

kr = task\_threads(mach\_task\_self(), &thread\_list, &thread\_count);

if (kr != KERN\_SUCCESS) {

return -1;

}

long total\_time = 0;

long total\_userTime = 0;

CGFloat total\_cpu = 0;

int j;

// for each thread

for (j = 0; j < (int)thread\_count; j++) {

thread\_info\_count = THREAD\_INFO\_MAX;

kr = thread\_info(thread\_list[j], THREAD\_BASIC\_INFO,

(thread\_info\_t)thinfo, &thread\_info\_count);

if (kr != KERN\_SUCCESS) {

return -1;

}

basic\_info\_th = (thread\_basic\_info\_t)thinfo;

if (!(basic\_info\_th->flags & TH\_FLAGS\_IDLE)) {

total\_time = total\_time + basic\_info\_th->user\_time.seconds + basic\_info\_th->system\_time.seconds;

total\_userTime = total\_userTime + basic\_info\_th->user\_time.microseconds + basic\_info\_th->system\_time.microseconds;

total\_cpu = total\_cpu + basic\_info\_th->cpu\_usage / (float)TH\_USAGE\_SCALE \* kMaxPercent;

}

}

kr = vm\_deallocate(mach\_task\_self(), (vm\_offset\_t)thread\_list, thread\_count \* sizeof(thread\_t));

assert(kr == KERN\_SUCCESS);

return total\_cpu;

}

**实现功能：**

* CPU架构
* CPU核数
* 系统CPU使用率
* 当前应用CPU使用率

**注意：由于安全性考虑，苹果已经禁止访问内核变量来获取CPU频率。现实现方法是通过硬编码方式获取CPU频率，新机发布需更新。**

1. **内存**

mach\_task\_basic\_info` 结构体存储了 Mach task 的内存使用信息，其中 `resident\_size` 就是应用使用的物理内存大小，`virtual\_size` 是虚拟内存大小。

与获取 CPU 占用率类似，在调用 task\_info API 时，target\_task 参数传入的是 `mach\_task\_self()`，表示获取当前的 Mach task，另外 flavor 参数传的是 `MACH\_TASK\_BASIC\_INFO`，使用这个类型会返回 `mach\_task\_basic\_info` 结构体，表示返回 target\_task 的基本信息，比如 task 的挂起次数和驻留页面数量。

**代码实现：**

/\*\*

获取APP内存使用量

@return byte

\*/

+ (unsigned long long)getAppRAMUsage {

struct mach\_task\_basic\_info info;

mach\_msg\_type\_number\_t count = MACH\_TASK\_BASIC\_INFO\_COUNT;

kern\_return\_t kr = task\_info(mach\_task\_self(), MACH\_TASK\_BASIC\_INFO, (task\_info\_t)&info, &count);

if (kr != KERN\_SUCCESS) {

return 0;

}

return info.resident\_size;

}

/\*\*

获取内存使用结构体

@return fs\_system\_ram\_usage

\*/

+ (fs\_system\_ram\_usage)getSystemRamUsageStruct {

vm\_statistics64\_data\_t vmStats;

mach\_msg\_type\_number\_t infoCount = HOST\_VM\_INFO\_COUNT;

kern\_return\_t kr = host\_statistics(mach\_host\_self(),

HOST\_VM\_INFO,

(host\_info\_t)&vmStats,

&infoCount);

fs\_system\_ram\_usage system\_memory\_usage = {0, 0, 0};

if (kr != KERN\_SUCCESS) {

return system\_memory\_usage;

}

system\_memory\_usage.used\_size = (vmStats.active\_count + vmStats.wire\_count + vmStats.inactive\_count) \* vm\_kernel\_page\_size;

system\_memory\_usage.available\_size = (vmStats.free\_count) \* vm\_kernel\_page\_size;

system\_memory\_usage.total\_size = [NSProcessInfo processInfo].physicalMemory;

return system\_memory\_usage;

}

**实现功能：**

* 系统总内存
* 系统使用内存
* 系统可用内存
* 当前应用内存使用量

1. **启动时间**

## 冷启动

t(App总启动时间) = t1(main()之前的加载时间) + t2(main()之后的加载时间)

t1 = 系统先读取App的可执行文件（Mach-O文件），从里面获得dyld的路径，然后加载dyld，dyld去初始化运行环境，开启缓存策略，加载程序相关依赖库(其中也包含我们的可执行文件)，并对这些库进行链接，最后调用每个依赖库的初始化方法，在这一步，runtime被初始化。当所有依赖库的初始化后，轮到最后一位(程序可执行文件)进行初始化，在这时runtime会对项目中所有类进行类结构初始化，然后调用所有的load方法。最后dyld返回main函数地址，main函数被调用，我们便来到了熟悉的程序入口。可归纳以下几点：

* 系统先读取App的可执行文件（Mach-O文件，自身App的所有.o文件的集合）
* 加载dyld（动态链接编辑器）
* Load dylibs: 加载动态库（包括所依赖的所有动态库）
* Rebase && Bind
* Objc SetUp: 初始化Objective C Runtime
* Initializers



进入main方法前操作

t2 = main方法执行之后到AppDelegate类中的- (BOOL)Application:(UIApplication \*)Application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions方法执行结束前这段时间，主要是构建第一个界面，并完成渲染展示。

**代码实现：**

static uint64\_t loadTime;

static uint64\_t applicationRespondedTime = -1;

static mach\_timebase\_info\_data\_t timebaseInfo;

static inline NSTimeInterval MachTimeToSeconds(uint64\_t machTime) {

return ((machTime / 1e9) \* timebaseInfo.numer) / timebaseInfo.denom;

}

@implementation XXStartupMeasurer

+ (void)load {

loadTime = mach\_absolute\_time();

mach\_timebase\_info(&timebaseInfo);

@autoreleasepool {

\_\_block id<NSObject> obs;

obs = [[NSNotificationCenter defaultCenter] addObserverForName:UIApplicationDidFinishLaunchingNotification

object:nil queue:nil

usingBlock:^(NSNotification \*note) {

dispatch\_async(dispatch\_get\_main\_queue(), ^{

applicationRespondedTime = mach\_absolute\_time();

NSLog(@"StartupMeasurer: it took %f seconds until the app could respond to user interaction.", MachTimeToSeconds(applicationRespondedTime - loadTime));

});

[[NSNotificationCenter defaultCenter] removeObserver:obs];

}];

}

}

1. **FPS**

FPS 是测量用于保存、显示动态视频的信息数量，每秒钟帧数愈多，所显示的动作就会愈流畅，一般应用只要保持 FPS 在 50-60，应用就会给用户流畅的感觉，反之，用户则会感觉到卡顿。

**代码实现：**

@implementation YYFPSLabel {

CADisplayLink \*\_link;

NSUInteger \_count;

NSTimeInterval \_lastTime;

}

- (id)init {

self = [super init];

if( self ){

\_link = [CADisplayLink displayLinkWithTarget:[YYWeakProxy proxyWithTarget:self] selector:@selector(tick:)];

[\_link addToRunLoop:[NSRunLoop mainRunLoop] forMode:NSRunLoopCommonModes];

}

return self;

}

- (void)dealloc {

[\_link invalidate];

}

- (void)tick:(CADisplayLink \*)link {

if (\_lastTime == 0) {

\_lastTime = link.timestamp;

return;

}

\_count++;

NSTimeInterval delta = link.timestamp - \_lastTime;

if (delta < 1) return;

\_lastTime = link.timestamp;

float fps = \_count / delta;

\_count = 0;

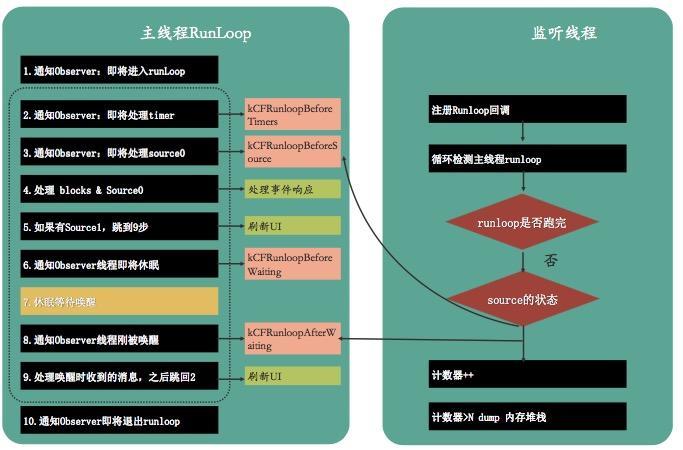
}

由于 CADisplayLink 的 `frameInterval` 值默认为 1，代表与帧刷新同步，这样可以通过 CADisplayLink 一个时间段内方法刷新次数算出屏幕 FPS 值

**注意：值得注意的是基于CADisplayLink实现的 FPS 在生产场景中只有指导意义，不能代表真实的 FPS，因为基于CADisplayLink实现的 FPS 无法完全检测出当前 Core Animation 的性能情况，它只能检测出当前 RunLoop 的帧率。**

1. **卡顿**

监控卡顿,最直接就是找到主线程.我们知道一个线程的消息事件处理都是依赖于NSRunLoop来驱动,所以要知道线程正在调用什么方法,就需要从NSRunLoop来入手。发现NSRunLoop调用方法主要就是在kCFRunLoopBeforeSources和kCFRunLoopBeforeWaiting之间,还有kCFRunLoopAfterWaiting之后,也就是如果我们发现这两个时间内耗时太长,那么就可以判定出此时主线程卡顿。



RunLoop监听卡顿

**代码实现：**

static void runLoopObserverCallBack(CFRunLoopObserverRef observer, CFRunLoopActivity activity, void \*info)

{

MyClass \*object = (\_\_bridge MyClass\*)info;

// 记录状态值

object->activity = activity;

// 发送信号

dispatch\_semaphore\_t semaphore = moniotr->semaphore;

dispatch\_semaphore\_signal(semaphore);

}

- (void)registerObserver

{

CFRunLoopObserverContext context = {0,(\_\_bridge void\*)self,NULL,NULL};

CFRunLoopObserverRef observer = CFRunLoopObserverCreate(kCFAllocatorDefault,

kCFRunLoopAllActivities,

YES,

0,

&runLoopObserverCallBack,

&context);

CFRunLoopAddObserver(CFRunLoopGetMain(), observer, kCFRunLoopCommonModes);

// 创建信号

semaphore = dispatch\_semaphore\_create(0);

// 在子线程监控时长

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

while (YES)

{

// 假定连续5次超时50ms认为卡顿(当然也包含了单次超时250ms)

long st = dispatch\_semaphore\_wait(semaphore, dispatch\_time(DISPATCH\_TIME\_NOW, 50\*NSEC\_PER\_MSEC));

if (st != 0)

{

if (activity==kCFRunLoopBeforeSources || activity==kCFRunLoopAfterWaiting)

{

if (++timeoutCount < 5)

continue;

NSLog(@"好像有点儿卡哦");

}

}

timeoutCount = 0;

}

});

}

1. **Crash防护**

利用Objective-C语言的动态特性，采用AOP(Aspect Oriented Programming) 面向切面编程的设计思想，做到无痕植入。能够自动在app运行时实时捕获导致app崩溃的破环因子，然后通过特定的技术手段去化解这些破坏因子，使app免于崩溃，照样可以继续正常运行，为app的持续运转保驾护航。

实现思路来自《[大白健康系统--iOS APP运行时Crash自动修复系统](https://neyoufan.github.io/2017/01/13/ios/BayMax_HTSafetyGuard/)》。

具体实现引用了[JJException库](https://github.com/jezzmemo/JJException)以及其[实现原理](https://github.com/jezzmemo/JJException/blob/master/JJExceptionPrinciple.md)

**实现功能：**

* Unrecognized selector crash
* KVO crash
* NSNotification crash
* NSTimer crash
* Container crash（数组越界，插nil等）
* NSString crash （字符串操作的crash）
* Bad Access crash （野指针）

1. **Crah记录**

开发 iOS 应用，解决 Crash 问题始终是一个难题。Crash分为两种，一种是由 EXC\_BAD\_ACCESS 引起的，原因是访问了不属于本进程的内存地址，有可能是访问已被释放的内存；另一种是未被捕获的 Objective-C 异常（NSException），导致程序向自身发送了 SIGABRT 信号而崩溃。

对于系统 Crash 而引起的程序异常退出，可以通过UncaughtExceptionHandler 机制捕获；也就是说在程序中 catch 以外的内容，被系统自带的错误处理而捕获。我们要做的就是用自定义的函数替代该 ExceptionHandler 即可。

**代码实现：**

#pragma mark -uncaughtCrashHandler 异常处理

/\*\*

开启捕捉闪退handler

\*/

void fs\_installUncaughtCrashHandler() {

if (isSetUncaughtHandler) {

return;

}

isSetUncaughtHandler = YES;

oldUncaughtExceptionHandler = NSGetUncaughtExceptionHandler();

NSSetUncaughtExceptionHandler (p\_uncaughtExceptionHandler);

signal(SIGABRT, p\_uncaughtSignalHandler);

signal(SIGILL, p\_uncaughtSignalHandler);

signal(SIGSEGV, p\_uncaughtSignalHandler);

signal(SIGFPE, p\_uncaughtSignalHandler);

signal(SIGBUS, p\_uncaughtSignalHandler);

signal(SIGPIPE, p\_uncaughtSignalHandler);

}

/\*\*

异常处理回调

@param exception NSException

\*/

- (void)p\_uncaughtExceptionHandler:(NSException \*)exception {

if (!\_hashTable) {

return;

}

FSCrashInfo \*info = [[FSCrashInfo alloc] init];

info.date = [NSDate date];

info.signal = kFSCrashException;

info.exception = exception;

info.name = exception.name;

info.reason = exception.reason;

info.callBackTrace = [exception.callStackSymbols componentsJoinedByString:@"\n"];

for (id<FSCrashMonitorDelegate> delegate in \_hashTable) {

if ([delegate respondsToSelector:@selector(crashMonitor:didCatchExceptionInfo:)]) {

[delegate crashMonitor:self didCatchExceptionInfo:info];

}

}

}

/\*\*

异常信号处理回调

@param signal 信号类型

\*/

- (void)p\_uncaughtSignalHandler:(int)signal {

if (!\_hashTable) {

return;

}

NSMutableArray \*callStackSymbols = [[NSThread callStackSymbols] mutableCopy];

if (callStackSymbols.count > 2) {

[callStackSymbols removeObjectsAtIndexes:[NSIndexSet indexSetWithIndexesInRange:NSMakeRange(0, 3)]];

}

FSCrashInfo \*info = [[FSCrashInfo alloc]init];

info.date = [NSDate date];

info.signal = signal;

info.name = [self signalName:signal];

info.reason = [NSString stringWithFormat:@"Signal %@ crash.",

[self signalName:signal]];

info.callBackTrace = [callStackSymbols componentsJoinedByString:@"\n"];

for (id<FSCrashMonitorDelegate>delegate in \_hashTable) {

if ([delegate respondsToSelector:@selector(crashMonitor:didCatchExceptionInfo:)]) {

[delegate crashMonitor:self didCatchExceptionInfo:info];

}

}

[self stop];

kill(getpid(), SIGKILL);

}

**注意：此模块只是对未防护的闪退信息进行记录，用来定位问题，并不能屏蔽闪退**

1. **网络流量**

## HTTPDNS

应用内自动解析DNS，降低DNS请求带来的延迟，其次预防了DNS被劫持。应用将DNS映射文件打包到APP中，客户端启动时从服务器获取最新的DNS映射。此外客户端将自己的位置信息诸如IP地址，国家码等加入映射文件的请求参数中，服务器就可以根据客户端所处的位置不同，下发距离其物理位置最近的Server IP地址，从而减小整体网络请求的延迟，实现一定程度的服务器动态部署DNS。当映射IP地址持续请求失败，上报失败IP地址。

## 系统网络

**实现功能：**

* 移动网络IP
* WIfi网络IP

## 流量监控

在每一个 HTTP 请求开始时，URL 加载系统创建一个合适的 NSURLProtocol 对象处理对应的 URL 请求，而我们需要做的就是写一个继承自 NSURLProtocol 的类，并通过 - registerClass: 方法注册我们的协议类，然后 URL 加载系统就会在请求发出时使用我们创建的协议对象对该请求进行处理。我们在请求以及响应时记录数据从而实现了对网络流量的监控。

**注意：需要注意的是 NSURLProtocol 只能拦截 UIURLConnection、NSURLSession 和 UIWebView 中的请求，对于 WKWebView 中发出的网络请求无能为力.**

**实现功能：**

* 系统网络流量
* 应用网络流量
* 首包时间
* 响应时间
* 网络状态
* 上行流量
* 下行流量

1. **Q&A**