**Time Profiler**

可以查看多个线程里那些方法费时过多的方法。先将右侧Hide System Libraries打上勾，这样能够过滤信息。然后在Call Tree上会默认按照费时的线程进行排序，单个线程中会也会按照对应的费时方法排序，选择方法后能够通过右侧Heaviest Stack Trace里双击查看到具体的费时操作代码，从而能够有针对性的优化，而不需要在一些本来就不会怎么影响性能的地方过度优化。

**Allocations**

这里可以对每个动作的前后进行Generations，对比内存的增加，查看使内存增加的具体的方法和代码所在位置。具体操作是在右侧Generation Analysis里点击Mark Generation，这样会产生一个Generation，切换到其他页面或一段时间产生了另外一个事件时再点Mark Generation来产生一个新的Generation，这样反复，生成多个Generation，查看这几个Generation会看到Growth的大小，如果太大可以点进去查看相应占用较大的线程里右侧Heaviest Stack Trace里查看对应的代码块，然后进行相应的处理。

**Leak**

可以在上面区域的Leaks部分看到对应的时间点产生的溢出，选择后在下面区域的Statistics>Allocation Summary能够看到泄漏的对象，同样可以通过Stack Trace查看到具体对应的代码区域。

**开发时需要注意如何避免一些性能问题**

**NSDateFormatter**

通过Instruments的检测会发现创建NSDateFormatter或者设置NSDateFormatter的属性的耗时总是排在前面，如何处理这个问题呢，比较推荐的是添加属性或者创建静态变量，这样能够使得创建初始化这个次数降到最低。还有就是可以直接用C，或者这个NSData的Category来解决https://github.com/samsoffes/sstoolkit/blob/master/SSToolkit/NSData%2BSSToolkitAdditions.m

**UIImage**

这里要主要是会影响内存的开销，需要权衡下imagedNamed和imageWithContentsOfFile，了解两者特性后，在只需要显示一次的图片用后者，这样会减少内存的消耗，但是页面显示会增加Image IO的消耗，这个需要注意下。由于imageWithContentsOfFile不缓存，所以需要在每次页面显示前加载一次，这个IO的操作也是需要考虑权衡的一个点。

**页面加载**

如果一个页面内容过多，view过多，这样将长页面中的需要滚动才能看到的那个部分视图内容通过开启新的线程同步的加载。

**优化首次加载时间**

通过Time Profier可以查看到启动所占用的时间，如果太长可以通过Heaviest Stack Trace找到费时的方法进行改造。

**监控卡顿的方法**

还有种方法是在程序里去监控性能问题。可以先看看这个Demo，地址https://github.com/ming1016/DecoupleDemo。 这样在上线后可以通过这个程序将用户的卡顿操作记录下来，定时发到自己的服务器上，这样能够更大范围的收集性能问题。众所周知，用户层面感知的卡顿都是来自处理所有UI的主线程上，包括在主线程上进行的大计算，大量的IO操作，或者比较重的绘制工作。如何监控主线程呢，首先需要知道的是主线程和其它线程一样都是靠NSRunLoop来驱动的。可以先看看CFRunLoopRun的大概的逻辑

int32\_t \_\_CFRunLoopRun()

{

    \_\_CFRunLoopDoObservers(KCFRunLoopEntry);

    do

    {

        \_\_CFRunLoopDoObservers(kCFRunLoopBeforeTimers);

        \_\_CFRunLoopDoObservers(kCFRunLoopBeforeSources); //这里开始到kCFRunLoopBeforeWaiting之间处理时间是感知卡顿的关键地方

        \_\_CFRunLoopDoBlocks();

        \_\_CFRunLoopDoSource0(); //处理UI事件

        //GCD dispatch main queue

        CheckIfExistMessagesInMainDispatchQueue();

        //休眠前

        \_\_CFRunLoopDoObservers(kCFRunLoopBeforeWaiting);

        //等待msg

        mach\_port\_t wakeUpPort = SleepAndWaitForWakingUpPorts();

        //等待中

        //休眠后，唤醒

        \_\_CFRunLoopDoObservers(kCFRunLoopAfterWaiting);

        //定时器唤醒

        if (wakeUpPort == timerPort)

            \_\_CFRunLoopDoTimers();

        //异步处理

        else if (wakeUpPort == mainDispatchQueuePort)

            \_\_CFRUNLOOP\_IS\_SERVICING\_THE\_MAIN\_DISPATCH\_QUEUE\_\_()

        //UI，动画

        else

            \_\_CFRunLoopDoSource1();

        //确保同步

        \_\_CFRunLoopDoBlocks();

    } while (!stop && !timeout);

    //退出RunLoop

    \_\_CFRunLoopDoObservers(CFRunLoopExit);

}

根据这个RunLoop我们能够通过CFRunLoopObserverRef来度量。用GCD里的dispatch\_semaphore\_t开启一个新线程，设置一个极限值和出现次数的值，然后获取主线程上在kCFRunLoopBeforeSources到kCFRunLoopBeforeWaiting再到kCFRunLoopAfterWaiting两个状态之间的超过了极限值和出现次数的场景，将堆栈dump下来，最后发到服务器做收集，通过堆栈能够找到对应出问题的那个方法。

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

{

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

    object->activity = activity;

}

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

    SMLagMonitor \*lagMonitor = (\_\_bridge SMLagMonitor\*)info;

    lagMonitor->runLoopActivity = activity;

    dispatch\_semaphore\_t semaphore = lagMonitor->dispatchSemaphore;

    dispatch\_semaphore\_signal(semaphore);

}

- (void)endMonitor {

    if (!runLoopObserver) {

        return;

    }

    CFRunLoopRemoveObserver(CFRunLoopGetMain(), runLoopObserver, kCFRunLoopCommonModes);

    CFRelease(runLoopObserver);

    runLoopObserver = NULL;

}

- (void)beginMonitor {

    if (runLoopObserver) {

        return;

    }

    dispatchSemaphore = dispatch\_semaphore\_create(0); //Dispatch Semaphore保证同步

    //创建一个观察者

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

    runLoopObserver = CFRunLoopObserverCreate(kCFAllocatorDefault,

                                              kCFRunLoopAllActivities,

                                              YES,

                                              0,

                                              &runLoopObserverCallBack,

                                              &context);

    //将观察者添加到主线程runloop的common模式下的观察中

    CFRunLoopAddObserver(CFRunLoopGetMain(), runLoopObserver, kCFRunLoopCommonModes);

    //创建子线程监控

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

        //子线程开启一个持续的loop用来进行监控

        while (YES) {

            long semaphoreWait = dispatch\_semaphore\_wait(dispatchSemaphore, dispatch\_time(DISPATCH\_TIME\_NOW, 30\*NSEC\_PER\_MSEC));

            if (semaphoreWait != 0) {

                if (!runLoopObserver) {

                    timeoutCount = 0;

                    dispatchSemaphore = 0;

                    runLoopActivity = 0;

                    return;

                }

                //两个runloop的状态，BeforeSources和AfterWaiting这两个状态区间时间能够检测到是否卡顿

                if (runLoopActivity == kCFRunLoopBeforeSources || runLoopActivity == kCFRunLoopAfterWaiting) {

                    //出现三次出结果

                    if (++timeoutCount 3) {

                        continue;

                    }

                    //将堆栈信息上报服务器的代码放到这里

                } //end activity

            }// end semaphore wait

            timeoutCount = 0;

        }// end while

    });

}

有时候造成卡顿是因为数据异常，过多，或者过大造成的，亦或者是操作的异常出现的，这样的情况可能在平时日常开发测试中难以遇到，但是在真实的特别是用户受众广的情况下会有人出现，这样这种收集卡顿的方式还是有价值的。

**堆栈dump的方法**

第一种是直接调用系统函数获取栈信息，这种方法只能够获得简单的信息，没法配合dSYM获得具体哪行代码出了问题，类型也有限。这种方法的主要思路是signal进行错误信号的获取。代码如下

static int s\_fatal\_signals[] = {

    SIGABRT,

    SIGBUS,

    SIGFPE,

    SIGILL,

    SIGSEGV,

    SIGTRAP,

    SIGTERM,

    SIGKILL,

};

static int s\_fatal\_signal\_num = sizeof(s\_fatal\_signals) / sizeof(s\_fatal\_signals[0]);

void UncaughtExceptionHandler(NSException \*exception) {

    NSArray \*exceptionArray = [exception callStackSymbols]; //得到当前调用栈信息

    NSString \*exceptionReason = [exception reason];       //非常重要，就是崩溃的原因

    NSString \*exceptionName = [exception name];           //异常类型

}

void SignalHandler(int code)

{

    NSLog(@"signal handler = %d",code);

}

void InitCrashReport()

{

    //系统错误信号捕获

    for (int i = 0; i signal(s\_fatal\_signals[i], SignalHandler);

    }

    //oc未捕获异常的捕获

    NSSetUncaughtExceptionHandler(&UncaughtExceptionHandler);

}

int main(int argc, char \* argv[]) {

    @autoreleasepool {

        InitCrashReport();

        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));

    }

}

使用PLCrashReporter的话出的报告看起来能够定位到问题代码的具体位置了。

NSData \*lagData = [[[PLCrashReporter alloc]

                                          initWithConfiguration:[[PLCrashReporterConfig alloc] initWithSignalHandlerType:PLCrashReporterSignalHandlerTypeBSD symbolicationStrategy:PLCrashReporterSymbolicationStrategyAll]] generateLiveReport];

PLCrashReport \*lagReport = [[PLCrashReport alloc] initWithData:lagData error:NULL];

NSString \*lagReportString = [PLCrashReportTextFormatter stringValueForCrashReport:lagReport withTextFormat:PLCrashReportTextFormatiOS];

//将字符串上传服务器

NSLog(@"lag happen, detail below:

%@",lagReportString);