

13 | 如何利用 RunLoop 原理去监控卡顿？

戴铭 2019-04-09



你好，我是戴铭。今天，我来和你说说如何监控卡顿。

卡顿问题，就是在主线程上无法响应用户交互的问题。如果一个 App 时不时地就给你卡一下，有时还长时间无响应，这时你还愿意继续用它吗？所以说，卡顿问题对 App 的伤害是巨大的，也是我们必须重点解决的一个问题。

现在，我们先来看一下导致卡顿问题的几种原因：

复杂 UI、图文混排的绘制量过大；

在主线程上做网络同步请求；

在主线程做大量的 IO 操作；

运算量过大，CPU 持续高占用；

死锁和主子线程抢锁。

那么，我们如何监控到什么时候会出现卡顿呢？是要监视 FPS 吗？

以前，我特别喜欢一本叫作《24 格》的杂志，它主要是介绍的是动画片制作的相关内容。那么，它为啥叫 24 格呢？这是因为，动画片中 1 秒钟会用到 24 张图片，这样肉眼看起来就是流畅的。

FPS 是一秒显示的帧数，也就是一秒内画面变化数量。如果按照动画片来说，动画片的 FPS 就是 24，是达不到 60 满帧的。也就是说，对于动画片来说，24 帧时虽然没有 60 帧时流畅，但也已经是连贯的了，所以并不能说 24 帧时就算是卡住了。

由此可见，简单地通过监视 FPS 是很难确定是否会出现卡顿问题了，所以我就果断弃了通过监视 FPS 来监控卡顿的方案。

那么，我们到底应该使用什么方案来监控卡顿呢？

RunLoop 原理

对于 iOS 开发来说，监控卡顿就是要去找到主线程上都做了哪些事儿。我们都知道，线程的消息事件是依赖于 NSRunLoop 的，所以从 NSRunLoop 入手，就可以知道主线程上都调用了哪些方法。我们通过监听 NSRunLoop 的状态，就能够发现调用方法是否执行时间过长，从而判断出是否会出现卡顿。

所以，我推荐的监控卡顿的方案是：通过监控 RunLoop 的状态来判断是否会出现卡顿。

RunLoop 是 iOS 开发中的一个基础概念，为了帮助你理解并用好这个对象，接下来我会先和你介绍一下它可以做哪些事儿，以及它为什么可以做成这些事儿。

RunLoop 这个对象，在 iOS 里由 CFRunLoop 实现。简单来说，RunLoop 是用来监听输入源，进行调度处理的。这里的输入源可以是输入设备、网络、周期性或者延迟时间、异步回调。RunLoop 会接收两种类型的输入源：一种是来自另一个线程或者来自不同应用的异步消息；另一种是来自预订时间或者重复间隔的同步事件。


RunLoop 的目的是，当有事件要去处理时保持线程忙，当没有事件要处理时让线程进入休眠。所以，了解 RunLoop 原理不光能够运用到监控卡顿上，还可以提高用户的交互体验。通过将那些繁重而不紧急会大量占用 CPU 的任务（比如图片加载），放到空闲的 RunLoop 模式里执行，这样就可以避开在 UITrackingRunLoopMode 这个 RunLoop 模式时是执行。UITrackingRunLoopMode 是用户进行滚动操作时会切换到的 RunLoop 模式，避免在这个 RunLoop 模式执行繁重的 CPU 任务，就能避免影响用户交互操作上体验。

接下来，我就通过 CFRunLoop 的源码来跟你分享下 RunLoop 的原理吧。

第一步

通知 observers：RunLoop 要开始进入 loop 了。紧接着就进入 loop。代码如下：


```
1 // 通知 observers
2 if (currentMode->_observerMask & kCFRunLoopEntry )
3     __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopEntry);
```

 复制代码

```
4 // 进入 loop
5 result = __CFRunLoopRun(r1, currentMode, seconds, returnAfterSourceHandled, prev
6
```


第二步

开启一个 do while 来保活线程。通知 Observers：RunLoop 会触发 Timer 回调、Source0 回调，接着执行加入的 block。代码如下：

 复制代码

```
1 // 通知 Observers RunLoop 会触发 Timer 回调
2 if (currentMode->_observerMask & kCFRunLoopBeforeTimers)
3     __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeTimers);
4 // 通知 Observers RunLoop 会触发 Source0 回调
5 if (currentMode->_observerMask & kCFRunLoopBeforeSources)
6     __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeSources);
7 // 执行 block
8 __CFRunLoopDoBlocks(runloop, currentMode);
9
```

接下来，触发 Source0 回调，如果有 Source1 是 ready 的状态的话，就会跳转到 handle_msg 去处理消息。代码如下：

 复制代码

```
1 if (MACH_PORT_NULL != dispatchPort ) {
2     Boolean hasMsg = __CFRunLoopServiceMachPort(dispatchPort, &msg)
3     if (hasMsg) goto handle_msg;
4 }
5
```

第三步

回调触发后，通知 Observers：RunLoop 的线程将进入休眠（sleep）状态。代码如下：

 复制代码

```
1 Boolean poll = sourceHandledThisLoop || (0ULL == timeout_context->termTSR);
2 if (!poll && (currentMode->_observerMask & kCFRunLoopBeforeWaiting)) {
3     __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeWaiting);
4 }
5
```

第四步

进入休眠后，会等待 mach_port 的消息，以再次唤醒。只有在下面四个事件出现时才会被再次唤醒：

- 基于 port 的 Source 事件；
- Timer 时间到；
- RunLoop 超时；
- 被调用者唤醒。


等待唤醒的代码如下：

 复制代码

```
1 do {
2     __CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer), &livePort) {
3         // 基于 port 的 Source 事件、调用者唤醒
4         if (modeQueuePort != MACH_PORT_NULL && livePort == modeQueuePort) {
5             break;
6         }
7         // Timer 时间到、RunLoop 超时
8         if (currentMode->_timerFired) {
9             break;
10        }
11    } while (1);
12
```

第五步

唤醒时通知 Observer：RunLoop 的线程刚刚被唤醒了。代码如下：

 复制代码

```
1 if (!poll && (currentMode->_observerMask & kCFRunLoopAfterWaiting))
2     __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopAfterWaiting);
3
```

第六步


RunLoop 被唤醒后就要开始处理消息了：

如果是 Timer 时间到的话，就触发 Timer 的回调；

如果是 dispatch 的话，就执行 block；

如果是 source1 事件的话，就处理这个事件。

消息执行完后，就执行加到 loop 里的 block。代码如下：

 复制代码

```
1 handle_msg:
2 // 如果 Timer 时间到，就触发 Timer 回调
3 if (msg-is-timer) {
4     __CFRunLoopDoTimers(runloop, currentMode, mach_absolute_time())
5 }
6 // 如果 dispatch 就执行 block
7 else if (msg_is_dispatch) {
8     __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__(msg);
9 }
10
11 // Source1 事件的话，就处理这个事件
12 else {
13     CFRunLoopSourceRef source1 = __CFRunLoopModeFindSourceForMachPort(runloop, c
14     sourceHandledThisLoop = __CFRunLoopDoSource1(runloop, currentMode, source1,
15     if (sourceHandledThisLoop) {
16         mach_msg(reply, MACH_SEND_MSG, reply);
17     }
18 }
19
```

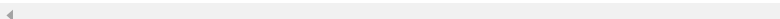


第七步

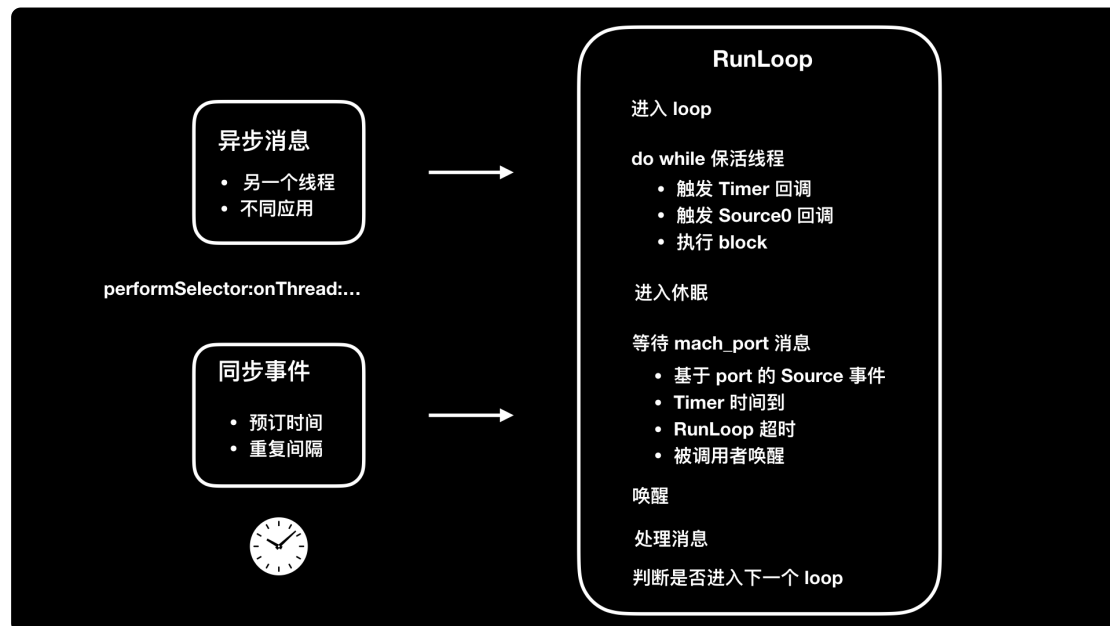
根据当前 RunLoop 的状态来判断是否需要走下一个 loop。当被外部强制停止或 loop 超时，就不继续下一个 loop 了，否则继续走下一个 loop。代码如下：

 复制代码

```
1 if (sourceHandledThisLoop && stopAfterHandle) {
2     // 事件已处理完
3     retVal = kCFRunLoopRunHandledSource;
4 } else if (timeout) {
5     // 超时
6     retVal = kCFRunLoopRunTimedOut;
7 } else if (__CFRunLoopIsStopped(runloop)) {
8     // 外部调用者强制停止
9     retVal = kCFRunLoopRunStopped;
10 } else if (__CFRunLoopModeIsEmpty(runloop, currentMode)) {
11     // mode 为空，RunLoop 结束
12     retVal = kCFRunLoopRunFinished;
13 }
14
```



整个 RunLoop 过程，我们可以总结为如下所示的一张图片。



这里只列出了 CFRunLoop 的关键代码，你可以点击[这个链接](#)查看完整代码。

loop 的六个状态

通过对 RunLoop 原理的分析，我们可以看出在整个过程中，loop 的状态包括 6 个，其代码定义如下：

```
1 typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
2     kCFRunLoopEntry, // 进入 loop
3     kCFRunLoopBeforeTimers, // 触发 Timer 回调
4     kCFRunLoopBeforeSources, // 触发 Source0 回调
5     kCFRunLoopBeforeWaiting, // 等待 mach_port 消息
6     kCFRunLoopAfterWaiting, // 接收 mach_port 消息
7     kCFRunLoopExit, // 退出 loop
8     kCFRunLoopAllActivities // loop 所有状态改变
9 }
10
```

复制代码

如果 RunLoop 的线程，进入睡眠前方法的执行时间过长而导致无法进入睡眠，或者线程唤醒后接收消息时间过长而无法进入下一步的话，就可以认为是线程受阻了。如果这个线程是主线程的话，表现出来的就是出现了卡顿。


所以，如果我们要利用 RunLoop 原理来监控卡顿的话，就是要关注这两个阶段。RunLoop 在进入睡眠之前和唤醒后的两个 loop 状态定义的值分别是 kCFRunLoopBeforeSources 和

kCFRunLoopAfterWaiting，也就是要触发 Source0 回调和接收 mach_port 消息两个状态。

接下来，我们就一起分析一下，如何对 loop 的这两个状态进行监听，以及监控的时间值如何设置才合理。

如何检查卡顿？

要想监听 RunLoop，你就首先需要创建一个 CFRunLoopObserverContext 观察者，代码如下：


 复制代码

```
1 CFRunLoopObserverContext context = {0, (__bridge void*)self, NULL, NULL};
2 runLoopObserver = CFRunLoopObserverCreate(kCFAllocatorDefault, kCFRunLoopAllActiv
3
```

将创建好的观察者 runLoopObserver 添加到主线程 RunLoop 的 common 模式下观察。然后，创建一个持续的子线程专门用来监控主线程的 RunLoop 状态。

一旦发现进入睡眠前的 kCFRunLoopBeforeSources 状态，或者唤醒后的状态 kCFRunLoopAfterWaiting，在设置的时间阈值内一直没有变化，即可判定为卡顿。接下来，我们就可以 dump 出堆栈的信息，从而进一步分析出具体是哪个方法的执行时间过长。

开启一个子线程监控的代码如下：

 复制代码

```
1 // 创建子线程监控
2 dispatch_async(dispatch_get_global_queue(0, 0), ^{
3     // 子线程开启一个持续的 loop 用来进行监控
4     while (YES) {
5         long semaphoreWait = dispatch_semaphore_wait(dispatchSemaphore, dispatch
6         if (semaphoreWait != 0) {
7             if (!runLoopObserver) {
8                 timeoutCount = 0;
9                 dispatchSemaphore = 0;
10                runLoopActivity = 0;
11                return;
12            }
13            //BeforeSources 和 AfterWaiting 这两个状态能够检测到是否卡顿
14            if (runLoopActivity == kCFRunLoopBeforeSources || runLoopActivity ==
15                // 将堆栈信息上报服务器的代码放到这里
16            } //end activity
17        } // end semaphore wait
18        timeoutCount = 0;
19    } // end while
20 });
21
```

代码中的 `NSEC_PER_SEC`，代表的是触发卡顿的时间阈值，单位是秒。可以看到，我们把这个阈值设置成了 3 秒。那么，这个 3 秒的阈值是从何而来呢？这样设置合理吗？

其实，触发卡顿的时间阈值，我们可以根据 WatchDog 机制来设置。WatchDog 在不同状态下设置的不同时间，如下所示：

启动 (Launch) : 20s;

恢复 (Resume) : 10s;

挂起 (Suspend) : 10s;

退出 (Quit) : 6s;

后台 (Background) : 3min (在 iOS 7 之前，每次申请 10min；之后改为每次申请 3min，可连续申请，最多申请到 10min)。


通过 WatchDog 设置的时间，我认为可以把启动的阈值设置为 10 秒，其他状态则都默认设置为 3 秒。总的原则就是，要小于 WatchDog 的限制时间。当然了，这个阈值也不用小得太多，原则就是要优先解决用户感知最明显的体验问题。

如何获取卡顿的方法堆栈信息？

子线程监控发现卡顿后，还需要记录当前出现卡顿的方法堆栈信息，并适时推送到服务端供开发者分析，从而解决卡顿问题。那么，在这个过程中，如何获取卡顿的方法堆栈信息呢？

获取堆栈信息的一种方法是直接调用系统函数。这种方法的有点在于，性能消耗小。但是，它只能够获取简单的信息，也没有办法配合 dSYM 来获取具体是哪行代码出了问题，而且能够获取的信息类型也有限。这种方法，因为性能比较好，所以适用于观察大盘统计卡顿情况、而不是想要找到卡顿原因的场景。

直接调用系统函数方法的主要思路是：用 signal 进行错误信息的获取。具体代码如下：

 复制代码

```
1 static int s_fatal_signals[] = {
2     SIGABRT,
3     SIGBUS,
4     SIGFPE,
5     SIGILL,
6     SIGSEGV,
7     SIGTRAP,
8     SIGTERM,
9     SIGKILL,
10 };
11
12 static int s_fatal_signal_num = sizeof(s_fatal_signals) / sizeof(s_fatal_signals)
13
14 void UncaughtExceptionHandler(NSException *exception) {
15     NSArray *exceptionArray = [exception callStackSymbols]; // 得到当前调用栈信息
16     NSString *exceptionReason = [exception reason];          // 非常重要，就是崩溃的原因
17     NSString *exceptionName = [exception name];              // 异常类型
```



```

18 }
19
20 void SignalHandler(int code)
21 {
22     NSLog(@"signal handler = %d",code);
23 }
24
25 void InitCrashReport()
26 {
27     // 系统错误信号捕获
28     for (int i = 0; i < s_fatal_signal_num; ++i) {
29         signal(s_fatal_signals[i], SignalHandler);
30     }
31
32     //oc 未捕获异常的捕获
33     NSSetUncaughtExceptionHandler(&UncaughtExceptionHandler);
34 }
35
36 int main(int argc, char * argv[]) {
37     @autoreleasepool {
38         InitCrashReport();
39         return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
40

```

另一种方法是，直接用 PLCrashReporter 这个开源的第三方库来获取堆栈信息。这种方法的特点是，能够定位到问题代码的具体位置，而且性能消耗也不大。所以，也是我推荐的获取堆栈信息的方法。

具体如何使用 PLCrashReporter 来获取堆栈信息，代码如下所示：

 复制代码

```

1 // 获取数据
2 NSData *lagData = [[PLCrashReporter alloc]
3                     initWithConfiguration:[PLCrashReporte
4 // 转换成 PLCrashReport 对象
5 PLCrashReport *lagReport = [[PLCrashReport alloc] initWithData:lagData error:NUL
6 // 进行字符串格式化处理
7 NSString *lagReportString = [PLCrashReportTextFormatter stringValueForCrashRepor
8 // 将字符串上传服务器
9 NSLog(@"lag happen, detail below: \n %@",lagReportString);
10

```

搜集到卡顿的方法堆栈信息以后，就是由开发者来分析并解决卡顿问题了。

在今天这篇文章中，我们用到的从监控卡顿到收集卡顿问题信息的完整代码，你都可以点击[这个链接](#)查看。

小结

今天我给你介绍了使用 RunLoop 监控卡顿的方案，我还跟你说了下 RunLoop 的原理，希望能够帮助你更好的理解 RunLoop 监控卡顿的方案。

读到这里你可能会想，为什么要将卡顿监控放到线上做呢？其实这样做主要是为了能够更大范围的收集问题，如果仅仅通过线下收集卡顿的话，场景无法被全面覆盖。因为，总有一些卡顿问题，是由于少数用户的数据异常导致的。

而用户反馈的卡顿问题往往都是说在哪个页面卡住了，而具体是执行哪个方法时卡主了，我们是从不得知的。在碰到这样问题时，你一定会感觉手足无措，心中反问一百遍：“我怎么在这个页面不卡，测试也不卡，就你卡”。而且，通过日志我们也很难查出个端倪。这时候，线上监控卡顿的重要性就凸显出来了。

有时，某个问题看似对 App 的影响不大，但如果这个问题在某个版本中爆发起来了就会变得难以收场。所以，你需要对这样的问题进行有预见性的监控，一方面可以早发现、早解决，另一方面在遇到问题时能够快速定位原因，不至于过于被动。要知道，面对问题的响应速度往往是评判基础设施建设优劣的一个重要的标准。

以上就是我们今天的内容了。接下来，我想请你回顾一下你都碰到过哪些卡顿问题，又是如何解决的呢？

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

©



一手资源 同步更新 加微信 sucl2015

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言(5)



Keep-Moving

PLCrashReporter怎么和卡顿检测结合起来呢？我理解它是收集崩溃信息的，但卡顿又不是一定会崩溃



1 2019-04-09



鹏哥

老师，请问下，如果我在用户滑动界面的时候不去加载图片，等停止滑动的时候再去加载图片，这个场景用runloop或者scrollview的代理来实现，和使用sdwebimage异步下载图片有什么区别，这几种方式貌似都没有影响用户滑动体验！



2019-04-09



大官人

一直跟着听，受益匪浅，学到很多也了解很多，如果只听不做，感觉都会了，都知道，一动手，卡住了……，动手才会思考，



2019-04-09



Geek_cc73f2

老师你好，从文章真的受益匪浅，以前只会用第三方的，现在能从本质看出端倪，另外能不能从源头给一些建议呢，比如怎么避免卡顿，哪些操作容易卡顿，然后怎么处理呢，感谢



2019-04-09



Ceezy

对于那个 GCD 的 dispatchPort，我看源码貌似是 GCD 提交给主线程的任务，这一事件源能称为 source1 吗？



2019-04-09