卡顿就是在应用使用过程中出现界面不响应或者界面渲染粘滞的情况，而应用界面的渲染以及事件响应是在主线程完成的，出现卡顿的原因可以归结为主线程阻塞。

开发过程中，造成主线程阻塞的原因可能是：

1. 主线程进行大量I/O操作
2. 主线程进行大量计算
3. 大量UI绘制
4. 主线程在等锁：主线程需要获得锁A，但是当前某个子线程持有这个锁A，导致主线程不得不等待子线程完成任务。

无论是何种原因引起的卡顿，都会造成非常不好的用户体验。因此，开发人员需要解决用户遇到的卡顿。

然而，卡顿和普通的bug不一样，卡顿的发生通常和机型、系统、用户操作路径有密切关系。因此，当用户上报卡顿时，开发、测试人员很难复现用户的问题。因此，要求开发人员在代码中监控卡顿，当卡顿发生时，能立马监听到且知道当前程序在做什么，并将日志上传。这样，开发人员才能解决问题。

上面说了，卡顿主要是主线程阻塞。因此，监控卡顿，其实就是卡顿发生时找到主线程在干什么，这样就能找到哪些操作耗时。在iOS中，线程消息处理是依赖于NSRunLoop来驱动的，因此需要从RunLoop入手，来查看主线程在做什么操作。

在介绍如何监控卡顿之前，先解决一个问题：卡顿的标准是什么？

卡顿标准

实际上，iOS中卡顿并没有一个严格的标准。iOS设备的标准刷新频率是60fps，但实际上，fps在55~60之间时，用户的感知并不明显。因此，不能简单的用fps来衡量卡顿。

现在已经有很多库能够监控卡顿，每个库的标准可能都不一样，这里介绍一种简单的策略。

iOS设备在1s内刷新60次，平均16.7s刷新一次。假设连丢3帧，也就是50ms左右，可以以此为衡量值。当然，上面已经说了，fps在55~60之间,用户感知并不明显，因此连丢3帧，并不能判定为卡顿。

不过，可以将50ms当做一个衡量值，如果连续几次都连丢3帧，那么可以判定为卡顿。可以设置为5次，也就是说，连续5次，每次都丢3帧，则定义此时发生了卡顿。

发生卡顿后，需要获得当前所有线程的调用栈，上报到服务端。关于如何获取所有线程调用栈，可以参考这篇文章。

监控卡顿

上面已经提到了RunLoop，这篇文章不会详细介绍RunLoop。这里简单说下和本文有关的RunLoop知识点。

RunLoop的整个工作流程如下：

```
1. 进入runLoop // kCFRunLoopEntry, 会发送通知
2. 通知将要处理Timer  // kCFRunLoopBeforeTimers , 会发送通知
3. 通知将要处理Source0  // kCFRunLoopBeforeSources, 会发送通知
4. 处理Source0
5. 如果有Source1，跳转到11
6. 通知即将休眠，BeforeWaiting  // kCFRunLoopBeforeWaiting, 即将休眠, 会发送通知
7. wait
8. 通知afterWaiting   // kCFRunLoopAfterWaiting, 从休眠中唤醒, 会发送通知
9. 处理timer
10. 处理dispatch到main_queue的block
11. 处理Source1
12. 回到第2步开始
13. 退出   // kCFRunLoopExit,会发送通知
```

在一些特殊的时间点，如kCFRunLoopEntry、kCFRunLoopBeforeSource会发送通知，通知Observer。

因此，非常明显的一点，我们可以创建一个Observer，来获得主线程RunLoop状态的变化。

上面已经说了，iOS中事件、消息处理是依赖于RunLoop的，那么在上面的工作流程中，RunLoop在哪些阶段是处理事件消息的呢?

从上面的流程可以看到：kCFRunLoopBeforeSources后会处理事件、kCFRunLoopAfterWaiting之后会处理事件。因此，需要监听kCFRunLoopBeforeSources、kCFRunLoopAfterWaiting这两个状态。

接下来就明了了，不间断的监控主线程RunLoop状态，当主线程RunLoop状态是上述两个状态时，记录下时间，看看在这两个状态上花费了多久。如果超过了50ms，则记录一次，连续5次之后，收集卡顿日志上报。

具体实现

监控主线程RunLoop状态，需要一个观察者，创建观察者代码如下：

```
//! 创建一个观察者
    CFRunLoopObserverContext context = {0,(__bridge
void*)self,NULL,NULL};

    // 第一个参数用于分配observer对象的内存
    // 第二个参数用于设置observer所要关注的事件
    // 第三个参数用于标识该observer是在第一次进入RunLoop时执行还是每次进
入RunLoop时都执行
    // 第四个参数用于设置该observer的优先级
    // 第五个参数用于设置该observer的回调函数
    // 第六个参数用于设置该observer的运行环境
    runLoopObserver = CFRunLoopObserverCreate(kCFAllocatorDefault,

kCFRunLoopAllActivities,
                                              YES,
                                              0,

&runLoopObserverCallBack,
                                              &context);
```

runLoopObserverCallBack是一个函数，用于处理当RunLoop状态发生改变时的逻辑，如下：

```
static void runLoopObserverCallBack(CFRunLoopObserverRef observer,
CFRunLoopActivity activity, void *info){
    LagMonitor *lagMonitor = (__bridge LagMonitor*)info;
    lagMonitor->runLoopActivity = activity;

    dispatch_semaphore_t semaphore = lagMonitor->dispatchSemaphore;
    dispatch_semaphore_signal(semaphore);
}
```

信号量的作用待会再说。

创建好观察者后，需要将该观察者添加到主线程runloop的commonModes模式下，只有这样，才能获得主线程runloop的状态，代码如下：

```
 //! 将观察者添加到主线程runloop的common模式下的观察中
    CFRunLoopAddObserver(CFRunLoopGetMain(), runLoopObserver,
kCFRunLoopCommonModes);
```

因为要监控主线程卡顿，因此监控程序肯定不能运行在主线程中。因为主线程发生卡顿时，监控程序也被阻塞了。所以需要开启一个子线程，在子线程中开启一个持续的循环，定时监控主线程的RunLoop状态。伪代码如下：

```
dispatch_async(dispatch_get_global_queue(0,0),^{
    while(YES) {
        // 是一个死循环，一直不间断监控
    }
});
```

因为有多线程，涉及到线程同步的问题。而且，我们的程序也要求，当有一个线程正在获取主线程状态时，其他线程需要等待，否则的话就得不到主线程RunLoop处理事件所花费的时间。也就是说，我们需要一个同步锁。

iOS中有很多种类型的锁，开发中，通常使用dispatch_semaphore_t来实现锁的效果，因为效率高。这也是为何 接收到RunLoop状态改变的回调函数中有信号量处理的代码。

上面说了，如果主线程RunLoop处理时间超过了50ms，则记录一次。借助于dispatch_semaphore_t的API，可以不用计算时间差，API如下：

```
long st = dispatch_semaphore_wait(semaphore,
dispatch_time(DISPATCH_TIME_NOW, 50*NSEC_PER_MSEC));
```

dispatch_semaphore_wait,如果成功，返回0；如果超时，返回非0。

也就是说，如果st不为0，说明RunLoop处理某个事件超过了50ms，进行一次记录。当超过5次后，记录线程栈日志。

代码实现

下面看一下完整的代码实现：

```
// 开始监听
- (void)startMonitor {
    if (observer) {
        return;
    }

    // 创建信号
    semaphore = dispatch_semaphore_create(0);

    // 注册RunLoop状态观察
    CFRunLoopObserverContext context = {0,(__bridge
void*)self,NULL,NULL};
    observer = CFRunLoopObserverCreate(kCFAllocatorDefault,
                                       kCFRunLoopAllActivities,
                                       YES,
                                       0,
                                       &runLoopObserverCallBack,
                                       &context);
    CFRunLoopAddObserver(CFRunLoopGetMain(), observer,
kCFRunLoopCommonModes);

    // 在子线程监控时长
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        while (YES) {
            long st = dispatch_semaphore_wait(semaphore,
dispatch_time(DISPATCH_TIME_NOW, 50*NSEC_PER_MSEC));
            if (st != 0) {  // 信号量超时了 - 即 runloop 的状态长时间没有
发生变更,长期处于某一个状态下
                if (!observer) {
                    timeoutCount = 0;
                    semaphore = 0;
                    activity = 0;
                    return;
                }
                // 判断是否
是kCFRunLoopBeforeSources、kCFRunLoopAfterWaiting这两个状态
                if (activity == kCFRunLoopBeforeSources || activity
== kCFRunLoopAfterWaiting) {    // 记录次数
                    if (++timeoutCount < 5) {
                        continue;
                    }

                    // 收集线程栈日志
                }
```

```
                }
                timeoutCount = 0;
            }
        });
    }
```

总结

以上就是iOS卡顿监控的实现方案。开发中还会遇到的一个崩溃是卡死崩溃，iOS系统没有提供卡死崩溃监控，需要开发者自己实现。其实现方案和卡顿监控类似，只不过是加了一些限制条件，获得了更多的操作日志，不再细说。