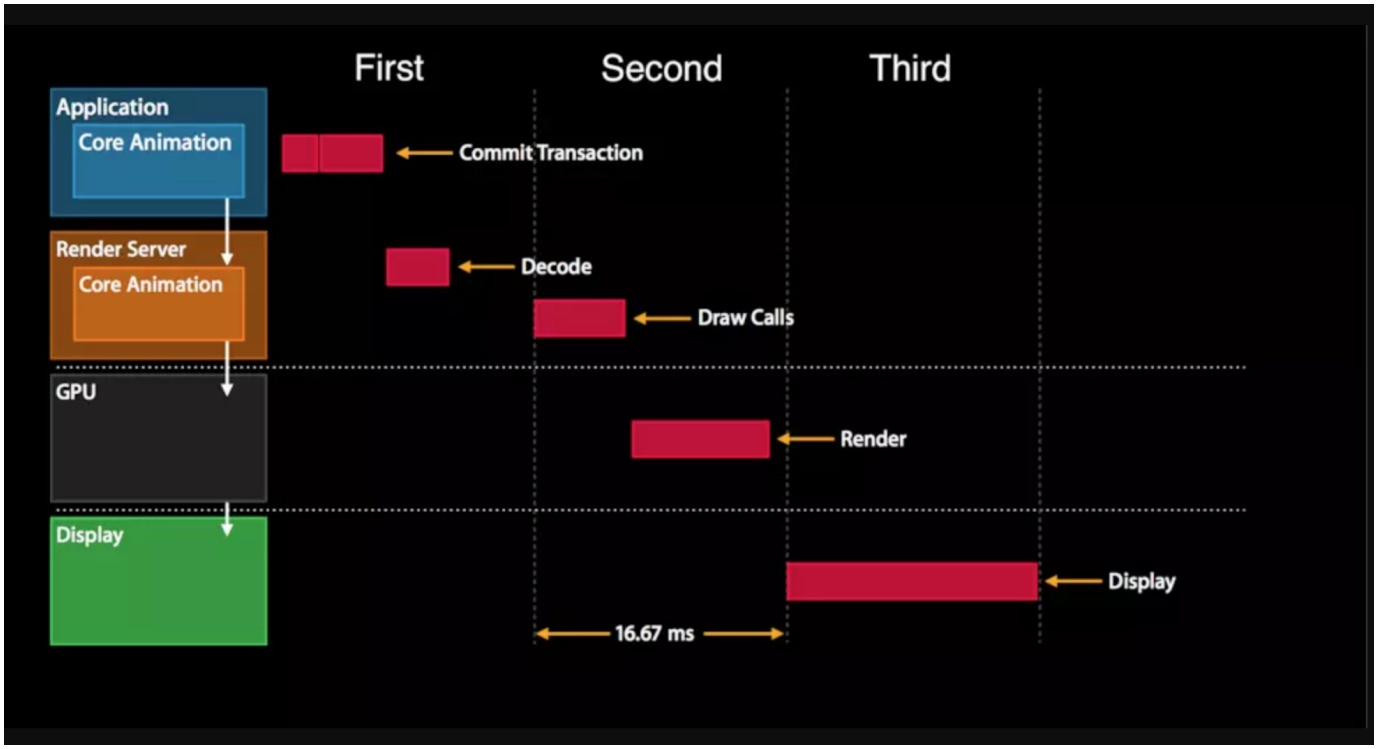
卡顿原理及监控

一、什么为卡顿，以及卡顿发生的原理。

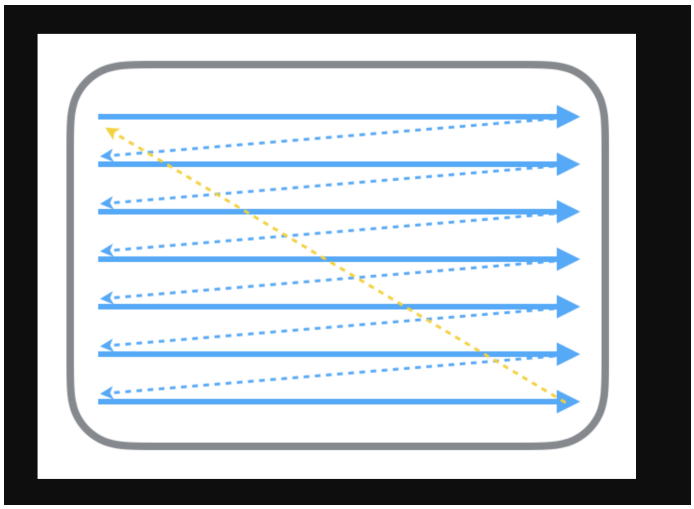
1、渲染机制



上图为WWDC2014讲述渲染模块所用的图，这个图非常清晰地讲述了整个渲染过程：

1. Application打包提交图层树并发送到渲染服务进程
2. 渲染服务进程对图层树进行反序列化得到渲染树，利用渲染树绘制位图，GPU合成位图
3. 最终由显示器将其显示出来

2、屏幕显示图像的原理



首先从过去的 CRT 显示器原理说起。CRT
的电子枪按照上面方式，从上到下一行行扫描，扫描完成后显示器就呈现一帧画面，随后电子枪回到初始位置继续下一次扫描。为了把显示器的显

示过程和系统的视频控制器进行同步，显示器（或者其他硬件）会用硬件时钟产生一系列的定时信号。当电子枪换到新的一行，准备进行扫描时，显示器会发出一个水平同步信号（horizonal synchronization），简称
HSync；而当一帧画面绘制完成后，电子枪回复到原位，准备画下一帧前，显示器会发出一个垂直同步信号（vertical synchronization），简称VSync。显示器通常以固定频率进行刷新，这个刷新率就是 VSync 信号产生的频率。尽管现在的设备大都是液晶显示屏了，但原理仍然没有变。

iOS 上完成图形的显示实际上是 CPU、GPU 和显示器协同工作的结果，具体来说，CPU
负责计算显示内容，包括视图的创建、布局计算、图片解码、文本绘制等，CPU 完成计算后会将计算内容提交给 GPU，GPU
进行变换、合成、渲染后将渲染结果提交到帧缓冲区，当下一次垂直同步信号（简称 V-Sync）到来时，最后显示到屏幕上

3、卡顿发生的原因：

搞清楚了 iPhone 的屏幕显示原理后，下面来看看在 iPhone 上为什么会出现卡顿现象，上文已经提及在图像真正在屏幕显示之前，CPU 和 GPU 需要完成自身的任务，而如果他们完成的时间错过了下一次 V-Sync
的到来（通常是1000/60=16.67ms），这样就会出现显示屏还是之前帧的内容，这就是界面卡顿的原因。不难发现，无论是 CPU 还是 GPU 引起错过 V-Sync 信号，都会造成界面卡顿。

参考博客：

iOS 保持界面流畅的技巧：https://blog.ibireme.com/2015/11/12/smooth_user_interfaces_for_ios/

页面间跳转的性能优化(二)：https://www.jianshu.com/p/92532c2b1d55

参考书籍：
《计算机图形学(第三版)》

二、卡顿监控方案

方案一（监控RunLoop状态检测超时）

假如在滚动过程中发生了卡顿现象，那么RunLoop必然会保持kCFRunLoopBeforeSources(处理消息之前)或者kCFRunLoopAfterWaiting(处理消息之后)这两个状态之一。

首先需要注册RunLoop的监听回调，保存RunLoop状态；其次，通过创建子线程循环监听主线程RunLoop的状态来检测是否存在停留卡顿现象。

缺陷：（只能检测滚动状态，因为非滚动状态主线程发生卡顿runloop会处于before waiting状态，并且不卡顿静止时也会处于before waiting状态）

方案二（标记位检测线程超时）

这套卡顿监控方案大致思路为：创建一个子线程进行循环检测，每次检测时设置标记位为YES，然后派发任务到主线程中将标记位设置为NO。接着子线程沉睡超时阈值时长，判断标志位是否成功设置成NO。如果没有说明主线程发生了卡顿，无法处理派发任务（如果主线程此时处于阻塞状态，那么我们给它派发的任务必定不能立即执行到）。

方案三 （CADisplayLink监控）

第三种方案采用CADisplayLink的方式来处理。思路是每个屏幕刷新周期派发标记位设置任务到主线程中，如果多次超出16.7ms的刷新阈值，即可看作是发生了卡顿.

技术难点：

1、在监测到主线程发生卡顿的时候打印出主线程的函数调用栈

通过callstackSymbols 方法

- 我们知道NSThread有一个类方法callstackSymbols可以获取当前线程的调用栈。通过使用dispatch_async或performSelectorOnMainThread等方法，回到主线程并获取调用栈。(行不通)

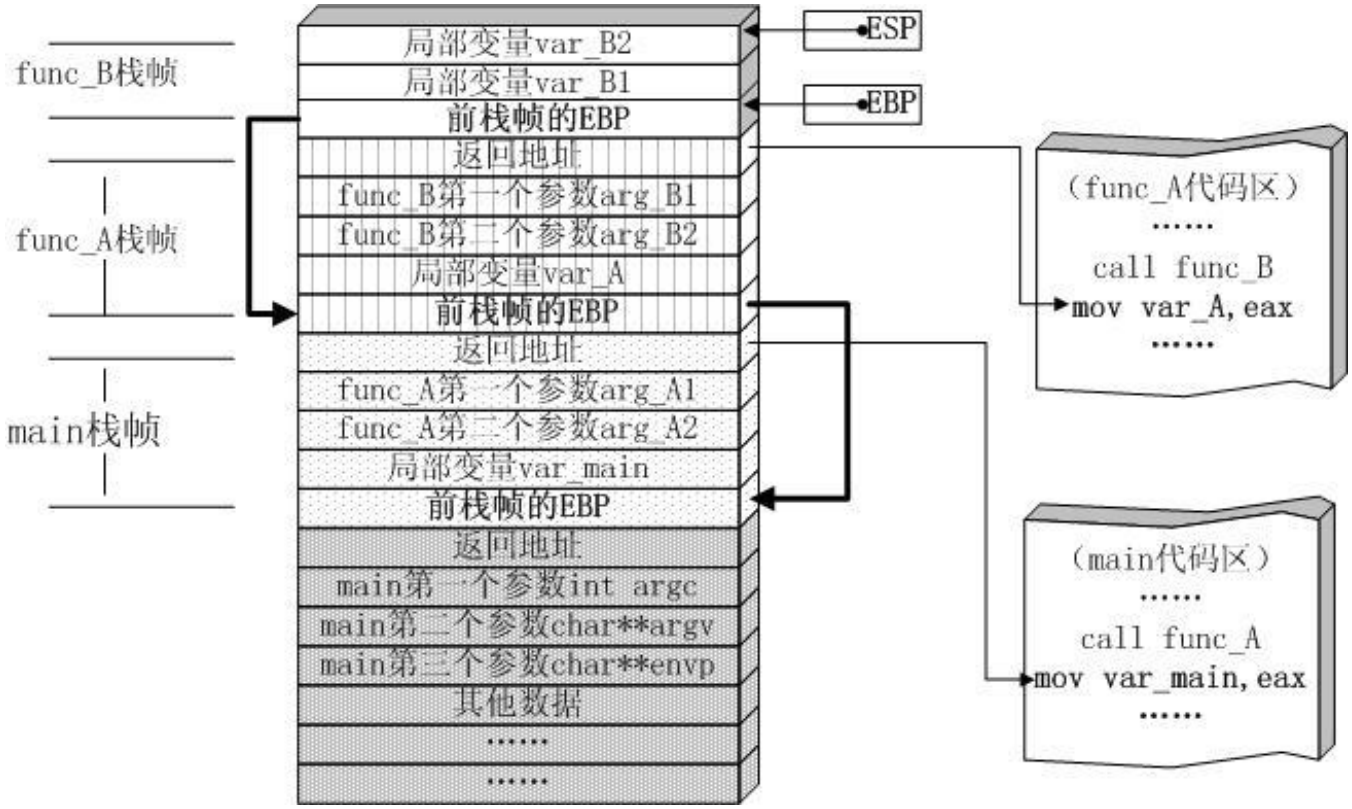
然后就需要另一种方式

先了解一下函数调用栈：

- ESP：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。
- EBP：基址指针寄存器(extended base pointer)，也可以叫帧指针寄存器，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。
- 函数栈帧：ESP和EBP之间的内存空间为当前栈帧，EBP标识了当前栈帧的底部，ESP标识了当前栈帧的顶部。
- EIP：指令寄存器(extended instruction pointer)，其内存放着一个指针，该指针永远指向下一条待执行的指令地址。

函数调用大致包括以下几个步骤：

- 参数入栈：将参数从右向左依次压入系统栈中
- 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行
- 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处
- 栈帧调整：具体包括
- 保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）
- 将当前栈帧切换到新栈帧。（将EBP指向ESP的位置，更新栈帧底部）
- 给新栈帧分配空间。（把ESP减去所需空间的大小，抬高栈顶）



结论：当前ebp指向的存储地址中，存储了上一次 ebp的值。这样的话，只需要知道当前的当前栈帧的EBP，就能知道上一个栈帧的EBP和返回地址，从而递归的获取函数调用栈。

首先系统提供了task_threads方法，可以获取到所有的线程(最底层的 mach 线程)

对于每一个线程(最底层的 mach 线程)，可以用thread_get_state方法获取它的所有信息，信息填充在_STRUCT_MCONTEXT类型的参数中

在_STRUCT_MCONTEXT类型的结构体中，存储了当前线程的栈指针Stack Pointer (esp)和最顶部栈帧的帧指针Frame Pointer(ebp)，从而获取到了整个线程的调用栈

2、符号解析

1. 根据 Frame Pointer（ebp）找到函数调用的地址
2. 找到 Frame Pointer （ebp）属于哪个镜像文件
3. 找到镜像文件的符号表
4. 在符号表中找到函数调用地址对应的符号名

参考博客：

卡顿监控：http://sindrilin.com/apm/performance/2017/03/24/%E5%8D%A1%E9%A1%BF%E6%A3%80%E6%B5%8B.html

获取任意线程调用栈的那些事：https://github.com/bestswifter/blog/blob/master/articles/objc-thread-backtrace.md

开源第三方：https://github.com/zixun/ANREye

函数调用过程中栈到底是怎么压入和弹出的？：https://www.zhihu.com/question/22444939

函数调用--函数栈：https://www.cnblogs.com/rain-lei/p/3622057.html

三、解决卡顿

1、CPU

- 耗时代码，阻塞主线程

2、GPU

- 纹理的渲染
- 视图的混合
- 图形的生成

3、平衡CPU与GPU的压力

参考博客：

iOS 保持界面流畅的技巧：https://blog.ibireme.com/2015/11/12/smooth_user_interfaces_for_ios/

UIKit 性能调优实战讲解：https://github.com/bestswifter/blog/blob/master/articles/uikit-optimization.md

离屏渲染：https://blog.csdn.net/qq_29846663/article/details/68960512