

# iOS 保持界面流畅的技巧

[ibireme](#)

这篇文章会非常详细的分析 iOS 界面构建中的各种性能问题以及对应的解决思路，同时给出一个开源的微博列表实现，通过实际的代码展示如何构建流畅的交互。

Index

演示项目

屏幕显示图像的原理

卡顿产生的原因和解决方案

CPU 资源消耗原因和解决方案

GPU 资源消耗原因和解决方案

AsyncDisplayKit

ASDK 的由来

ASDK 的资料

ASDK 的基本原理

ASDK 的图层预合成

ASDK 异步并发操作

RunLoop 任务分发

微博 Demo 性能优化技巧

预排版

预渲染

异步绘制

全局并发控制

更高效的异步图片加载

其他可以改进的地方

如何评测界面的流畅度

演示项目

在开始技术讨论前，你可以先下载我写的 Demo 跑到真机上体验一下：<https://github.com/ibireme/YYKit>。Demo 里包含一个微博的 Feed 列表、发布视图，还包含一个 Twitter 的 Feed 列表。为了公平起见，所有

界面和交互我都从官方应用原封不动的抄了过来，数据也都是从官方应用抓取的。你也可以自己抓取数据替换掉 Demo 中的数据，方便进行对比。尽管官方应用背后的功能更多更为复杂，但不至于会带来太大的交互性能差异。

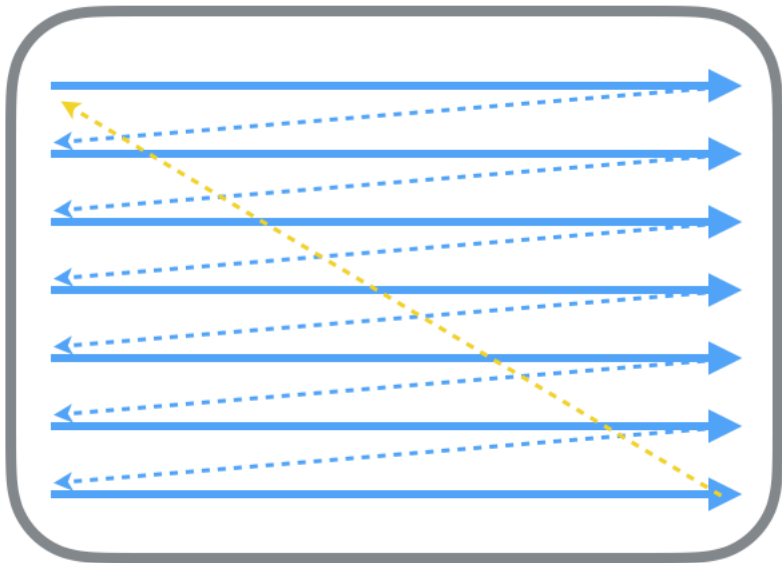




这个 Demo 最低可以运行在 iOS 6 上，所以你可以把它跑到老设备上体验一下。在我的测试中，即使在 iPhone 4S 或者 iPad 3 上，Demo 列表在快速滑动时仍然能保持 50~60 FPS 的流畅交互，而其他诸如微博、朋友圈等应用的列表视图在滑动时已经有很严重的卡顿了。

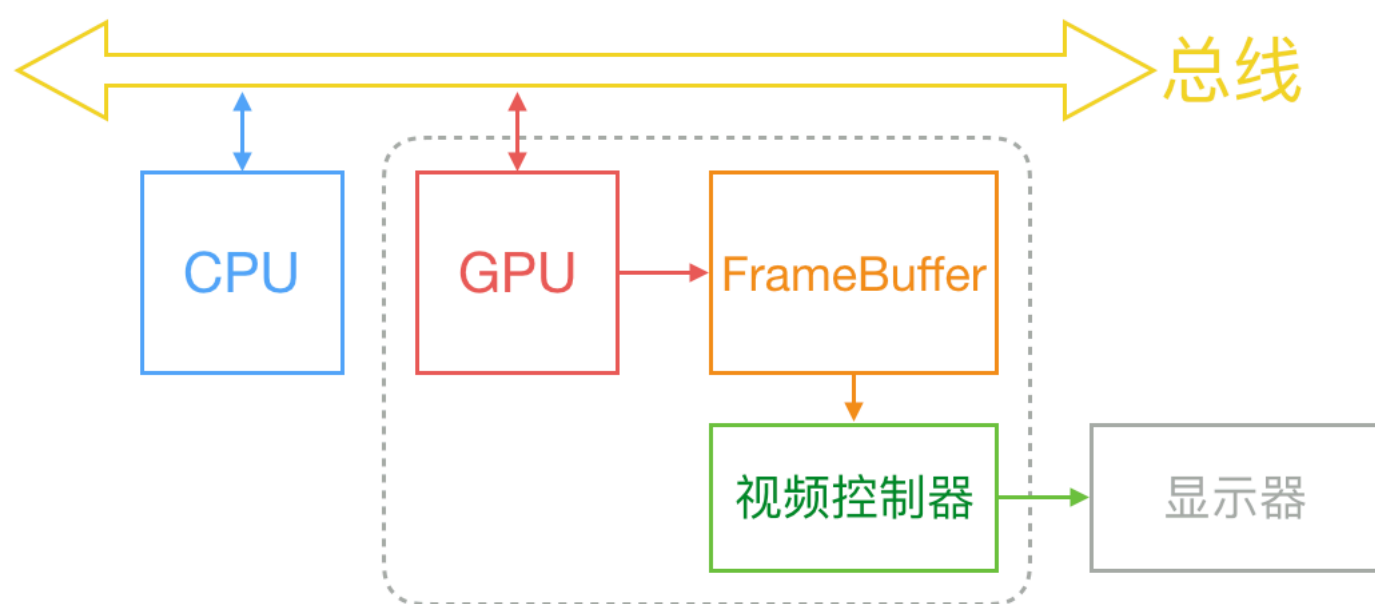
微博的 Demo 有大约四千行代码，Twitter 的只有两千行左右代码，第三方库只用到了 YYKit，文件数量比较少，方便查看。好了，下面是正文。

### 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（horizontal 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。



通常来说，计算机系统中 CPU、GPU、显示器是以上面这种方式协同工作的。CPU 计算好显示内容提交到 GPU，GPU 渲染完成后将渲染结果放入帧缓冲区，随后视频控制器会按照 VSync 信号逐行读取帧缓冲区的数据，经过可能的数模转换传递给显示器显示。

在最简单的情况下，帧缓冲区只有一个，这时帧缓冲区的读取和刷新都都会有比较大的效率问题。为了解决效率问题，显示系统通常会引入两个缓冲区，即双缓冲机制。在这种情况下，GPU 会预先渲染好一帧放入一个缓冲区内，让视频控制器读取，当下一帧渲染好后，GPU 会直接把视频控制器的指针指向第二个缓冲器。如此一来效率会有很大的提升。

双缓冲虽然能解决效率问题，但会引入一个新的问题。当视频控制器还未读取完成时，即屏幕内容刚显示一半时，GPU 将新的一帧内容提交到帧缓冲区并把两个缓冲区进行交换后，视频控制器就会把新的一帧数据的下半段显示到屏幕上，造成画面撕裂现象，如下图：

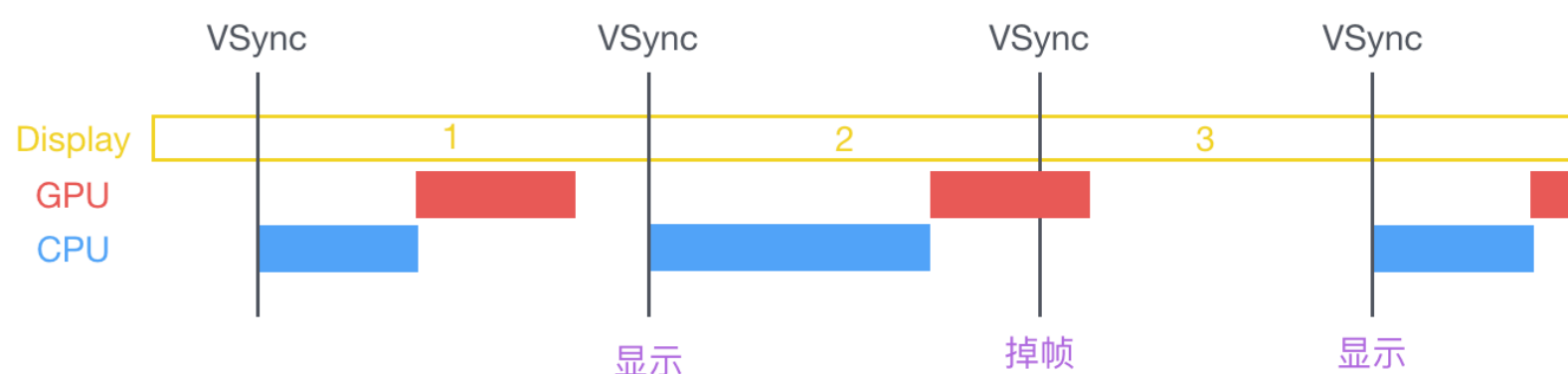




为了解决这个问题，GPU 通常有一个机制叫做垂直同步（简写也是 V-Sync），当开启垂直同步后，GPU 会等待显示器的 VSync 信号发出后，才进行新的一帧渲染和缓冲区更新。这样能解决画面撕裂现象，也增加了画面流畅度，但需要消费更多的计算资源，也会带来部分延迟。

那么目前主流的移动设备是什么情况呢？从网上查到的资料可以知道，iOS 设备会始终使用双缓存，并开启垂直同步。而安卓设备直到 4.1 版本，Google 才开始引入这种机制，目前安卓系统是三缓存+垂直同步。

## 卡顿产生的原因和解决方案



在 VSync 信号到来后，系统图形服务会通过 CADisplayLink 等机制通知 App，App 主线程开始在 CPU 中计算显示内容，比如视图的创建、布局计算、图片解码、文本绘制等。随后 CPU 会将计算好的内容提交到 GPU 去，由 GPU 进行变换、合成、渲染。随后 GPU 会把渲染结果提交到帧缓冲区去，等待下一次 VSync 信号到来时显示到屏幕上。由于垂直同步的机制，如果在一个 VSync 时间内，CPU 或者 GPU 没有完成内容提交，则那一帧就会被丢弃，等待下一次机会再显示，而这时显示屏会保留之前的内容

不变。这就是界面卡顿的原因。

从上面的图中可以看到，CPU 和 GPU 不论哪个阻碍了显示流程，都会造成掉帧现象。所以开发时，也需要分别对 CPU 和 GPU 压力进行评估和优化。

## CPU 资源消耗原因和解决方案

### 对象创建

对象的创建会分配内存、调整属性、甚至还有读取文件等操作，比较消耗 CPU 资源。尽量用轻量的对象代替重量的对象，可以对性能有所优化。比如 CALayer 比 UIView 要轻量许多，那么不需要响应触摸事件的控件，用 CALayer 显示会更加合适。如果对象不涉及 UI 操作，则尽量放到后台线程去创建，但可惜的是包含有 CALayer 的控件，都只能在主线程创建和操作。通过 Storyboard 创建视图对象时，其资源消耗会比直接通过代码创建对象要大非常多，在性能敏感的界面里，Storyboard 并不是一个好的技术选择。

尽量推迟对象创建的时间，并把对象的创建分散到多个任务中去。尽管这实现起来比较麻烦，并且带来的优势并不多，但如果有能力做，还是要尽量尝试一下。如果对象可以复用，并且复用的代价比释放、创建新对象要小，那么这类对象应当尽量放到一个缓存池里复用。

### 对象调整

对象的调整也经常是消耗 CPU 资源的地方。这里特别说一下 CALayer：CALayer 内部并没有属性，当调用属性方法时，它内部是通过运行时 `resolveInstanceMethod` 为对象临时添加一个方法，并把对应属性值保存到内部的一个 Dictionary 里，同时还会通知 delegate、创建动画等等，非常消耗资源。UIView 的关于显示相关的属性（比如 `frame/bounds/transform`）等实际上都是 CALayer 属性映射来的，所以对 UIView 的这些属性进行调整时，消耗的资源要远大于一般的属性。对此你在应用中，应该尽量减少不必要的属性修改。

当视图层次调整时，UIView、CALayer 之间会出现很多方法调用与通知，所以在优化性能时，应该尽量避免调整视图层次、添加和移除视图。

## 对象销毁

对象的销毁虽然消耗资源不多，但累积起来也是不容忽视的。通常当容器类持有大量对象时，其销毁时的资源消耗就非常明显。同样的，如果对象可以放到后台线程去释放，那就挪到后台线程去。这里有个小 Tip：把对象捕获到 block 中，然后扔到后台队列去随便发送个消息以避免编译器警告，就可以让对象在后台线程销毁了。

```
NSArray *tmp = self.array;  
self.array = nil;
```

```
NSArray *tmp = self.array;  
  
self.array = nil;  
  
dispatch_async(queue, ^{  
    [tmp class];  
});
```

## 布局计算

视图布局的计算是 App 中最为常见的消耗 CPU 资源的地方。如果能在后台线程提前计算好视图布局、并且对视图布局进行缓存，那么这个地方基本就不会产生性能问题了。

不论通过何种技术对视图进行布局，其最终都会落到对 `UIView.frame/bounds/center` 等属性的调整上。上面也说过，对这些属性的调整非常消耗资源，所以尽量提前计算好布局，在需要时一次性调整好对应属性，而不要多次、频繁的计算和调整这些属性。

## Autolayout

Autolayout 是苹果本身提倡的技术，在大部分情况下也能很好的提升开发效率，但是 Autolayout 对于复杂视图来说常常会产生严重的性能问题。随着视图数量的增长，Autolayout 带来的 CPU 消耗会呈指数级上升。具体数据可以看这个文章：<http://pilky.me/36/>。如果你不想手动调整 `frame` 等属性，你可以用一些工具方法替代（比如常见的 `left/right/top/bottom/width/height` 快捷属性），或者使用

UIKit、AsyncDisplayKit 等框架。

## 文本计算

如果一个界面中包含大量文本（比如微博微信朋友圈等），文本的宽高计算会占用很大一部分资源，并且不可避免。如果你对文本显示没有特殊要求，可以参考下 UILabel 内部的实现方式：用 [NSAttributedString boundingRectWithSize:options:context:] 来计算文本宽高，用 - [NSAttributedString drawWithRect:options:context:] 来绘制文本。尽管这两个方法性能不错，但仍旧需要放到后台线程进行以避免阻塞主线程。

如果你用 CoreText 绘制文本，那就可以先生成 CoreText 排版对象，然后自己计算了，并且 CoreText 对象还能保留以供稍后绘制使用。

## 文本渲染

屏幕上能看到的所有文本内容控件，包括 UIWebView，在底层都是通过 CoreText 排版、绘制为 Bitmap 显示的。常见的文本控件（UILabel、UITextView 等），其排版和绘制都是在主线程进行的，当显示大量文本时，CPU 的压力会非常大。对此解决方案只有一个，那就是自定义文本控件，用 TextKit 或最底层的 CoreText 对文本异步绘制。尽管这实现起来非常麻烦，但其带来的优势也非常大，CoreText 对象创建好后，能直接获取文本的宽高等信息，避免了多次计算（调整 UILabel 大小时算一遍、UILabel 绘制时内部再算一遍）；CoreText 对象占用内存较少，可以缓存下来以备稍后多次渲染。

## 图片的解码

当你用 UIImage 或 CGImageSource 的那几个方法创建图片时，图片数据并不会立刻解码。图片设置到 UIImageView 或者 CALayer.contents 中去，并且 CALayer 被提交到 GPU 前，CGImage 中的数据才会得到解码。这一步是发生在主线程的，并且不可避免。如果想要绕开这个机制，常见的做法是在后台线程先把图片绘制到 CGContext 中，然后从 Bitmap 直接创建图片。目前常见的网络图片库都自带这个功能。

## 图像的绘制



图像的绘制通常是指用那些以 CG 开头的方法把图像绘制到画布中，然后从画布创建图片并显示这样一个过程。这个最常见的地方就是 [UIView drawRect:] 里面了。由于 CoreGraphic 方法通常都是线程安全的，所以图像的绘制可以很容易的放到后台线程进行。一个简单异步绘制的过程大致如下（实际情况会比这个复杂得多，但原理基本一致）：

- (void)display {

```
- (void)display {
    dispatch_async(backgroundQueue, ^{
        CGContextRef ctx = CGContextCreate(...);
        // draw in context...
        CGImageRef img = CGContextCreateImage(ctx);
        CFRelease(ctx);
        dispatch_async(mainQueue, ^{
            layer.contents = img;
        });
    });
}
```

## GPU 资源消耗原因和解决方案

相对于 CPU 来说，GPU 能干的事情比较单一：接收提交的纹理（Texture）和顶点描述（三角形），应用变换（transform）、混合并渲染，然后输出到屏幕上。通常你所能看到的内容，主要也就是纹理（图片）和形状（三角模拟的矢量图形）两类。

## 纹理的渲染

所有的 Bitmap，包括图片、文本、栅格化的内容，最终都要由内存提交到显存，绑定为 GPU Texture。不论是提交到显存的过程，还是 GPU 调整和渲染 Texture 的过程，都要消耗不少 GPU 资源。当在较短时间显示大量图片时（比如 TableView 存在非常多的图片并且快速滑动时），CPU 占用率

很低，GPU 占用非常高，界面仍然会掉帧。避免这种情况的方法只能是尽量减少在短时间内大量图片的显示，尽可能将多张图片合成为一张进行显示。

当图片过大，超过 GPU 的最大纹理尺寸时，图片需要先从 CPU 进行预处理，这对 CPU 和 GPU 都会带来额外的资源消耗。目前来说，iPhone 4S 以上机型，纹理尺寸上限都是 4096x4096，更详细的资料可以看这里：[iosres.com](http://iosres.com)。所以，尽量不要让图片和视图的大小超过这个值。

## 视图的混合 (Composing)

当多个视图（或者说 CALayer）重叠在一起显示时，GPU 会首先把他们混合到一起。如果视图结构过于复杂，混合的过程也会消耗很多 GPU 资源。为了减轻这种情况的 GPU 消耗，应用应当尽量减少视图数量和层次，并在不透明的视图里标明 opaque 属性以避免无用的 Alpha 通道合成。当然，这也可以用上面的方法，把多个视图预先渲染为一张图片来显示。

## 图形的生成。

CALayer 的 border、圆角、阴影、遮罩 (mask)，CASHarpLayer 的矢量图形显示，通常会触发离屏渲染 (offscreen rendering)，而离屏渲染通常发生在 GPU 中。当一个列表视图中出现大量圆角的 CALayer，并且快速滑动时，可以观察到 GPU 资源已经占满，而 CPU 资源消耗很少。这时界面仍然能正常滑动，但平均帧数会降到很低。为了避免这种情况，可以尝试开启 CALayer.shouldRasterize 属性，但这会把原本离屏渲染的操作转嫁到 CPU 上去。对于只需要圆角的某些场合，也可以用一张已经绘制好的圆角图片覆盖到原本视图上面来模拟相同的视觉效果。最彻底的解决办法，就是把需要显示的图形在后台线程绘制为图片，避免使用圆角、阴影、遮罩等属性。

## AsyncDisplayKit

AsyncDisplayKit 是 Facebook 开源的一个用于保持 iOS 界面流畅的库，我从中学到了很多东西，所以下面我会花较大的篇幅来对其进行介绍和分析。

## ASDK 的由来



ASDK 的作者是 Scott Goodson ([Linkedin](#)), 他曾经在苹果工作, 负责 iOS 的一些内置应用的开发, 比如股票、计算器、地图、钟表、设置、Safari 等, 当然他也参与了 UIKit framework 的开发。后来他加入 Facebook 后, 负责 Paper 的开发, 创建并开源了 AsyncDisplayKit。目前他在 Pinterest 和 Instagram 负责 iOS 开发和用户体验的提升等工作。



ASDK 自 2014 年 6 月开源, 10 月发布 1.0 版。目前 ASDK 即将要发布 2.0 版。

V2.0 增加了更多布局相关的代码, ComponentKit 团队为此贡献很多。现在 Github 的 master 分支上的版本是 V1.9.1, 已经包含了 V2.0 的全部内容。

## ASDK 的资料

想要了解 ASDK 的原理和细节, 最好从下面几个视频开始:

2014.10.15 [NSLondon - Scott Goodson - Behind AsyncDisplayKit](#)

2015.03.02 [MCE 2015 - Scott Goodson - Effortless Responsiveness with AsyncDisplayKit](#)

2015.10.25 [AsyncDisplayKit 2.0: Intelligent User Interfaces - NSSpain 2015](#)

前两个视频内容大同小异, 都是介绍 ASDK 的基本原理, 附带介绍 POP 等其他项目。

后一个视频增加了 ASDK 2.0 的新特性的介绍。

除此之外，还可以到 Github Issues 里看一下 ASDK 相关的讨论，下面是几个比较重要的内容：

[关于 Runloop Dispatch](#)

[关于 ComponentKit 和 ASDK 的区别](#)

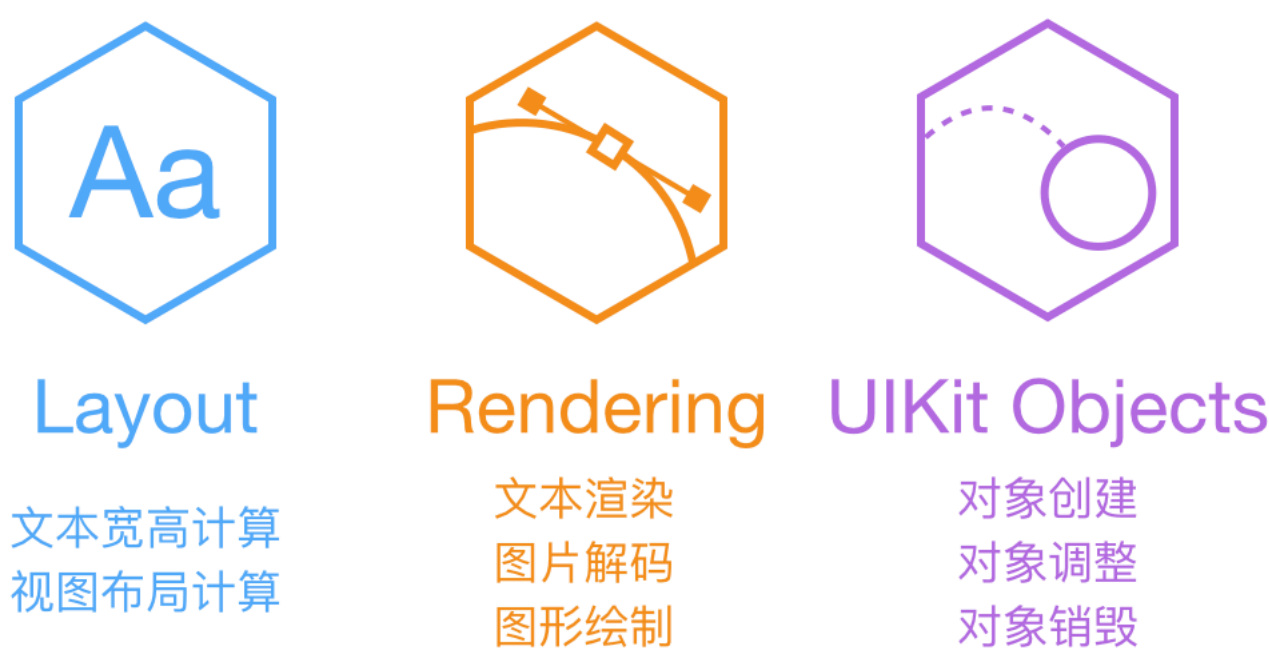
[为什么不支持 Storyboard 和 Autolayout](#)

[如何评测界面的流畅度](#)

之后，还可以到 Google Groups 来查看和讨论更多内容：

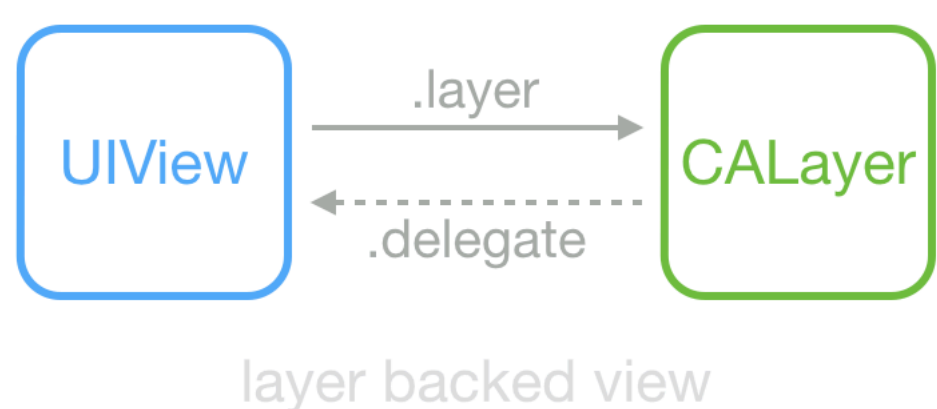
<https://groups.google.com/forum/#!forum/asyncdisplaykit>

## ASDK 的基本原理

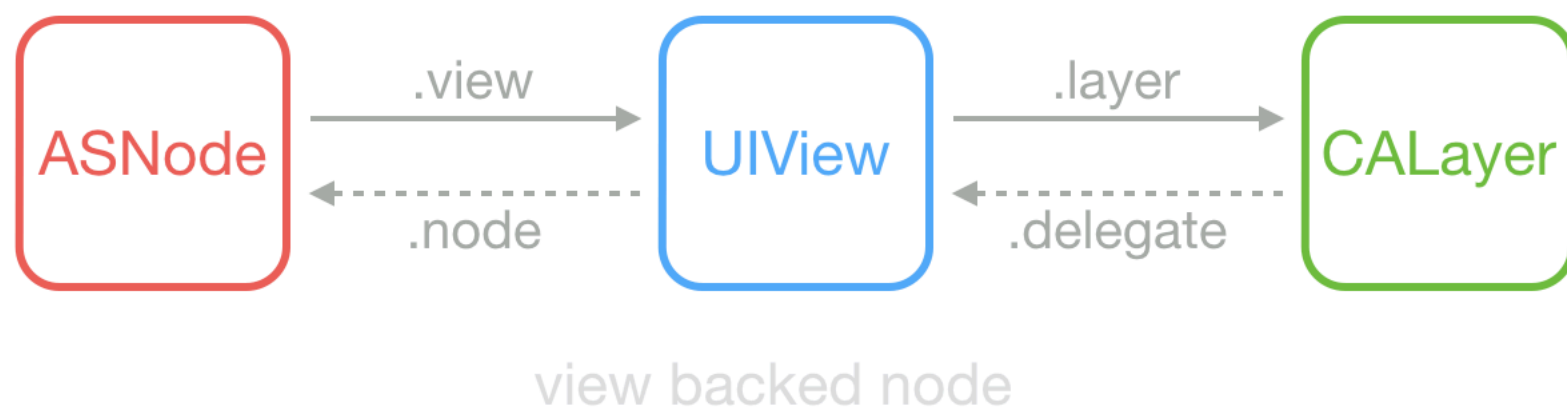


ASDK 认为，阻塞主线程的任务，主要分为上面这三大类。文本和布局的计算、渲染、解码、绘制都可以通过各种方式异步执行，但 UIKit 和 Core Animation 相关操作必需在主线程进行。ASDK 的目标，就是尽量把这些任务从主线程挪走，而挪不走的，就尽量优化性能。

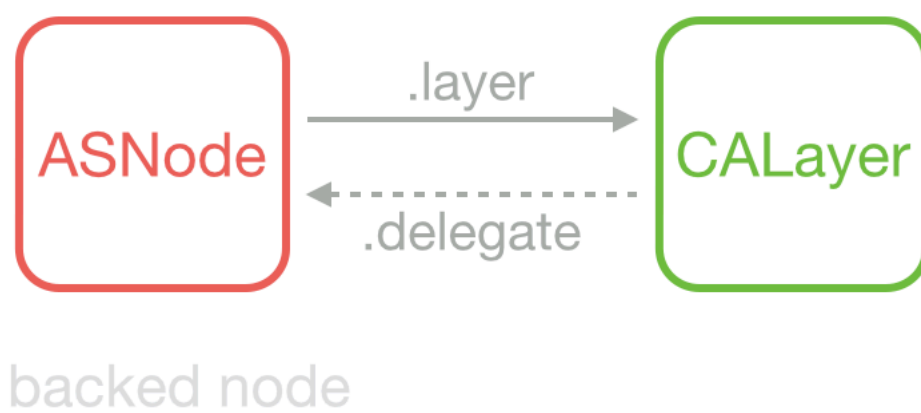
为了达成这一目标，ASDK 尝试对 UIKit 组件进行封装：



这是常见的 UIView 和 CALayer 的关系：View 持有 Layer 用于显示，View 中大部分显示属性实际是从 Layer 映射而来；Layer 的 delegate 在这里是 View，当其属性改变、动画产生时，View 能够得到通知。UIView 和 CALayer 不是线程安全的，并且只能在主线程创建、访问和销毁。



ASDK 为此创建了 ASDisplayNode 类，包装了常见的视图属性（比如 frame/bounds/alpha/transform/background-color/superNode/subNodes 等），然后它用 UIView->CALayer 相同的方式，实现了 ASNode->UIView 这样一个关系。



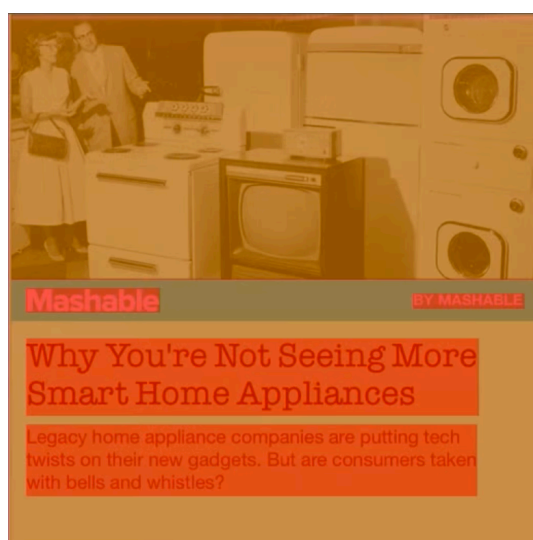
当不需要响应触摸事件时，ASDisplayNode 可以被设置为 layer backed，即 ASDisplayNode 充当了原来 UIView 的功能，节省了更多资源。

与 UIView 和 CALayer 不同，ASDisplayNode 是线程安全的，它可以在后台线程创建和修改。Node 刚创建时，并不会在内部新建 UIView 和 CALayer，直到第一次在主线程访问 view 或 layer 属性时，它才会在内部生成对应的对象。当它的属性（比如 frame/transform）改变后，它并不会立刻同步到其持有的 view 或 layer 去，而是把被改变的属性保存到内部的一个中间变量，稍后在需要时，再通过某个机制一次性设置到内部的 view 或 layer。



通过模拟和封装 UIView/CALayer，开发者可以把代码中的 UIView 替换为 ASNode，很大的降低了开发和学习成本，同时能获得 ASDK 底层大量的性能优化。为了方便使用，ASDK 把大量常用控件都封装成了 ASNode 的子类，比如 Button、Control、Cell、Image、ImageView、Text、TableView、CollectionView 等。利用这些控件，开发者可以尽量避免直接使用 UIKit 相关控件，以获得更完整的性能提升。

## ASDK 的图层预合成



有时一个 layer 会包含很多 sub-layer，而这些 sub-layer 并不需要响应触摸事件，也不需要进行动画和位置调整。ASDK 为此实现了一个被称为 pre-composing 的技术，可以把这些 sub-layer 合成渲染为一张图片。开发时，ASNode 已经替代了 UIView 和 CALayer；直接使用各种 Node 控件并设置为 layer backed 后，ASNode 甚至可以通过预合成来避免创建内部的 UIView 和 CALayer。

通过这种方式，把一个大的层级，通过一个大的绘制方法绘制到一张图上，性能会获得很大提升。CPU 避免了创建 UIKit 对象的资源消耗，GPU 避免了多张 texture 合成和渲染的消耗，更少的 bitmap 也意味着更少的内存占

用。

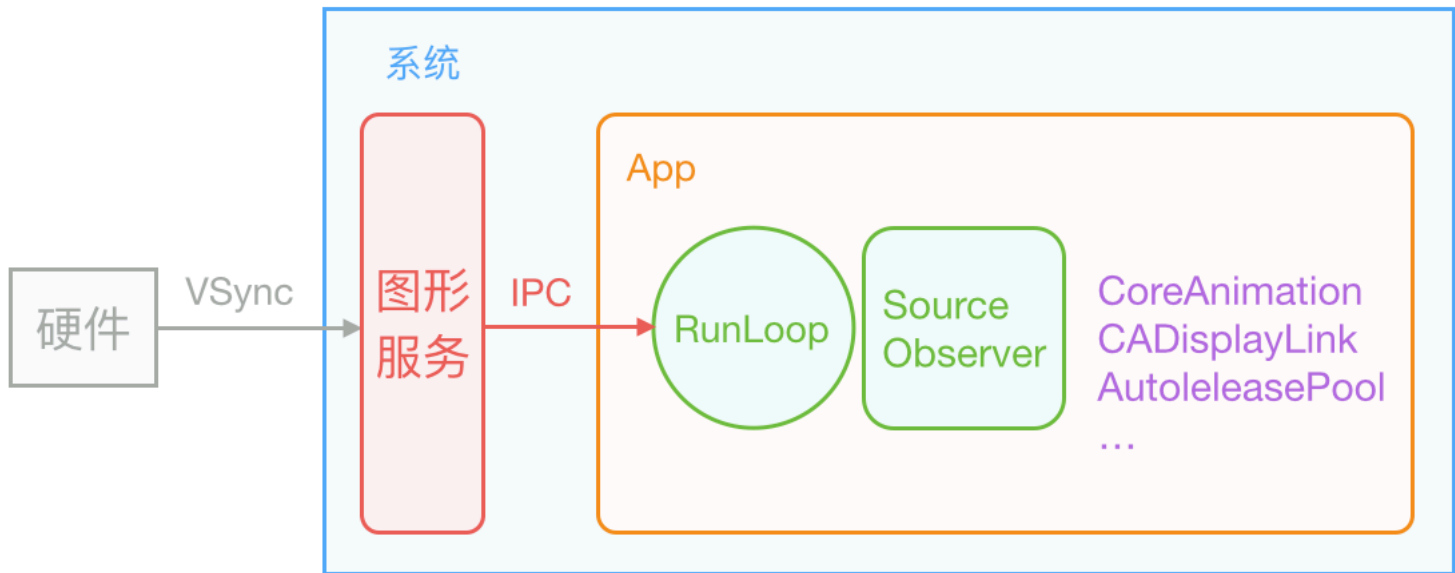
## ASDK 异步并发操作



自 iPhone 4S 起，iDevice 已经都是双核 CPU 了，现在的 iPad 甚至已经更新到 3 核了。充分利用多核的优势、并发执行任务对保持界面流畅有很大作用。ASDK 把布局计算、文本排版、图片/文本/图形渲染等操作都封装成较小的任务，并利用 GCD 异步并发执行。如果开发者使用了 ASNode 相关的控件，那么这些并发操作会自动在后台进行，无需进行过多配置。

## RunLoop 任务分发

RunLoop work distribution 是 ASDK 比较核心的一个技术，ASDK 的介绍视频和文档中都没有详细展开介绍，所以这里我会多做一些分析。如果你对 Runloop 还不太了解，可以看一下我之前的文章 [深入理解RunLoop](#)，里面会对 ASDK 也有所提及。



iOS 的显示系统是由 VSync 信号驱动的，VSync 信号由硬件时钟生成，每秒钟发出 60 次（这个值取决设备硬件，比如 iPhone 真机上通常是 59.97）。iOS 图形服务接收到 VSync 信号后，会通过 IPC 通知到 App

内。App 的 Runloop 在启动后会注册对应的 CFRunLoopSource 通过 mach\_port 接收传过来的时钟信号通知，随后 Source 的回调会驱动整个 App 的动画与显示。

Core Animation 在 RunLoop 中注册了一个 Observer，监听了 BeforeWaiting 和 Exit 事件。这个 Observer 的优先级是 2000000，低于常见的其他 Observer。当一个触摸事件到来时，RunLoop 被唤醒，App 中的代码会执行一些操作，比如创建和调整视图层级、设置 UIView 的 frame、修改 CALayer 的透明度、为视图添加一个动画；这些操作最终都会被 CALayer 捕获，并通过 CATransaction 提交到一个中间状态去（CATransaction 的文档略有提到这些内容，但并不完整）。当上面所有操作结束后，RunLoop 即将进入休眠（或者退出）时，关注该事件的 Observer 都会得到通知。这时 CA 注册的那个 Observer 就会在回调中，把所有的中间状态合并提交到 GPU 去显示；如果此处有动画，CA 会通过 DisplayLink 等机制多次触发相关流程。

ASDK 在此处模拟了 Core Animation 的这个机制：所有针对 ASNode 的修改和提交，总有些任务是必需放入主线程执行的。当出现这种任务时，ASNode 会把任务用 ASAsyncTransaction(Group) 封装并提交到一个全局的容器去。ASDK 也在 RunLoop 中注册了一个 Observer，监视的事件和 CA 一样，但优先级比 CA 要低。当 RunLoop 进入休眠前、CA 处理完事件后，ASDK 就会执行该 loop 内提交的所有任务。具体代码见这个文件：[ASAsyncTransactionGroup](#)。

通过这种机制，ASDK 可以在合适的机会把异步、并发的操作同步到主线程去，并且能获得不错的性能。

## 其他

ASDK 中还有封装很多高级的功能，比如滑动列表的预加载、V2.0 添加的新的布局模式等。ASDK 是一个很庞大的库，它本身并不推荐你把整个 App 全部都改为 ASDK 驱动，把最需要提升交互性能的地方用 ASDK 进行优化就足够了。

## 微博 Demo 性能优化技巧

我为了演示 YYKit 的功能，实现了微博和 Twitter 的 Demo，并为它们做了

不少性能优化，下面就是优化时用到的一些技巧。

## 预排版

当获取到 API JSON 数据后，我会把每条 Cell 需要的数据都在后台线程计算并封装为一个布局对象 CellLayout。CellLayout 包含所有文本的 CoreText 排版结果、Cell 内部每个控件的高度、Cell 的整体高度。每个 CellLayout 的内存占用并不多，所以当生成后，可以全部缓存到内存，以供稍后使用。这样，TableView 在请求各个高度函数时，不会消耗任何多余计算量；当把 CellLayout 设置到 Cell 内部时，Cell 内部也不用再计算布局了。

对于通常的 TableView 来说，提前在后台计算好布局结果是非常重要的一个性能优化点。为了达到最高性能，你可能需要牺牲一些开发速度，不要用 Autolayout 等技术，少用 UILabel 等文本控件。但如果你对性能的要求并不那么高，可以尝试用 TableView 的预估高度的功能，并把每个 Cell 高度缓存下来。这里有个来自百度知道团队的开源项目可以很方便的帮你实现这一点：[FDTemplateLayoutCell](#)。

## 预渲染

微博的头像在某次改版中换成了圆形，所以我也跟进了一下。当头像下载下来后，我会在后台线程将头像预先渲染为圆形并单独保存到一个 ImageCache 中去。

对于 TableView 来说，Cell 内容的离屏渲染会带来较大的 GPU 消耗。在 Twitter Demo 中，我为了图省事用到了不少 layer 的圆角属性，你可以在低性能的设备（比如 iPad 3）上快速滑动一下这个列表，能感受到虽然列表并没有较大的卡顿，但是整体的平均帧数降了下来。用 Instrument 查看时能够看到 GPU 已经满负荷运转，而 CPU 却比较清闲。为了避免离屏渲染，你应当尽量避免使用 layer 的 border、corner、shadow、mask 等技术，而尽量在后台线程预先绘制好对应内容。

## 异步绘制

我只在显示文本的控件上用到了异步绘制的功能，但效果很不错。我参考 ASDK 的原理，实现了一个简单的异步绘制控件。这块代码我单独提取出



来，放到了这里：[YYAsyncLayer](#)。YYAsyncLayer 是 CALayer 的子类，当它需要显示内容（比如调用了 [layer setNeedDisplay]）时，它会向 delegate，也就是 UIView 请求一个异步绘制的任务。在异步绘制时，Layer 会传递一个 BOOL(^isCancelled)() 这样的 block，绘制代码可以随时调用该 block 判断绘制任务是否已经被取消。

当 TableView 快速滑动时，会有大量异步绘制任务提交到后台线程去执行。但是有时滑动速度过快时，绘制任务还没有完成就可能已经被取消了。如果这时仍然继续绘制，就会造成大量的 CPU 资源浪费，甚至阻塞线程并造成后续的绘制任务迟迟无法完成。我的做法是尽量快速、提前判断当前绘制任务是否已经被取消；在绘制每一行文本前，我都会调用 isCancelled() 来进行判断，保证被取消的任务能及时退出，不至于影响后续操作。




































目前有些第三方微博客户端（比如 VVebo、墨客等），使用了一种方式来避免高速滑动时 Cell 的绘制过程，相关实现见这个项

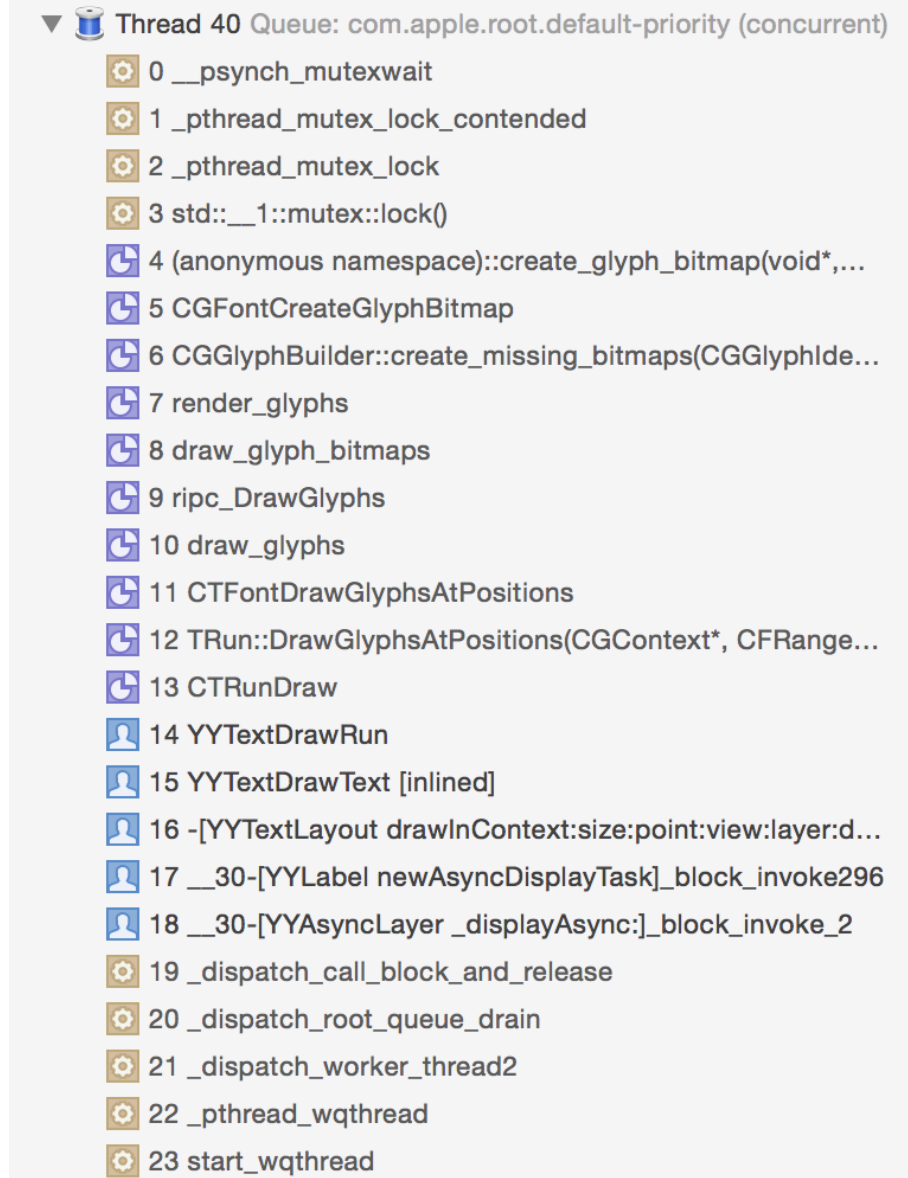
目：[VVeboTableViewDemo](#)。它的原理是，当滑动时，松开手指后，立刻计算出滑动停止时 Cell 的位置，并预先绘制那个位置附近的几个 Cell，而忽略当前滑动中的 Cell。这个方法比较有技巧性，并且对于滑动性能来说提升也很大，唯一的缺点就是快速滑动中会出现大量空白内容。如果你不想实现比较麻烦的异步绘制但又想保证滑动的流畅性，这个技巧是个不错的选择。

## 全局并发控制

当我用 concurrent queue 来执行大量绘制任务时，偶尔会遇到这种问题：



- ▶  Thread 34 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 35 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 36
- ▶  Thread 37 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 38 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 39
- ▶  Thread 40 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 41 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 42 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 43
- ▶  Thread 44 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 45
- ▶  Thread 46
- ▶  Thread 47
- ▶  Thread 48
- ▶  Thread 49
- ▶  Thread 50 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 51 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 52
- ▶  Thread 53
- ▶  Thread 54 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 55 Queue: ClassicURLConnection (serial)
- ▶  Thread 56 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 57 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 58
- ▶  Thread 59 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 60 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 61
- ▶  Thread 62 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 63
- ▶  Thread 64 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 65
- ▶  Thread 66 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 67 Queue: com.apple.root.default-priority (concurrent)
- ▶  Thread 68 Queue: com.apple.root.default-priority (concurrent)



大量的任务提交到后台队列时，某些任务会因为某些原因（此处是 CGFont 锁）被锁住导致线程休眠，或者被阻塞，concurrent queue 随后会创建新的线程来执行其他任务。当这种情况变多时，或者 App 中使用了大量 concurrent queue 来执行较多任务时，App 在同一时刻就会存在几十个线程同时运行、创建、销毁。CPU 是用时间片轮转来实现线程并发的，尽管 concurrent queue 能控制线程的优先级，但当大量线程同时创建运行销毁时，这些操作仍然会挤占掉主线程的 CPU 资源。ASDK 有个 Feed 列表的 Demo: [SocialAppLayout](#)，当列表内 Cell 过多，并且非常快速的滑动时，界面仍然会出现少量卡顿，我谨慎的猜测可能与这个问题有关。

使用 concurrent queue 时不可避免会遇到这种问题，但使用 serial queue 又不能充分利用多核 CPU 的资源。我写了一个简单的工具 [YYDispatchQueuePool](#)，为不同优先级创建和 CPU 数量相同的 serial queue，每次从 pool 中获取 queue 时，会轮询返回其中一个 queue。我把 App 内所有异步操作，包括图像解码、对象释放、异步绘制等，都按优先级不同放入了全局的 serial queue 中执行，这样尽量避免了过多线程导致的性能问题。

## 更高效的异步图片加载

SDWebImage 在这个 Demo 里仍然会产生少量性能问题，并且有些地方不能满足我的需求，所以我自己实现了一个性能更高的图片加载库。在显示简单的单张图片时，利用 `UIView.layer.contents` 就足够了，没必要使用 `UIImageView` 带来额外的资源消耗，为此我在 `CALayer` 上添加了 `setImageWithURL` 等方法。除此之外，我还把图片解码等操作通过 `YYDispatchQueuePool` 进行管理，控制了 App 总线程数量。

## 其他可以改进的地方

上面这些优化做完后，微博 Demo 已经非常流畅了，但在我的设想中，仍然有一些进一步优化的技巧，但限于时间和精力我并没有实现，下面简单列一下：

列表中有不少视觉元素并不需要触摸事件，这些元素可以用 ASDK 的图层合成技术预先绘制为一张图。

再进一步减少每个 Cell 内图层的数量，用 `CALayer` 替换掉 `UIView`。

目前每个 Cell 的类型都是相同的，但显示的内容却各部一样，比如有的 Cell 有图片，有的 Cell 里是卡片。把 Cell 按类型划分，进一步减少 Cell 内不必要的视图对象和操作，应该能有一些效果。

把需要放到主线程执行的任务划分为足够小的块，并通过 `RunLoop` 来进行调度，在每个 `Loop` 里判断下一次 `VSync` 的时间，并在下次 `VSync` 到来前，把当前未执行完的任务延迟到下一个机会去。这个只是我的一个设想，并不一定能实现或起作用。

## 如何评测界面的流畅度

最后还是要提一下，“过早的优化是万恶之源”，在需求未定，性能问题不明显时，没必要尝试做优化，而要尽量正确的实现功能。做性能优化时，也最好是走修改代码 -> Profile -> 修改代码这样一个流程，优先解决最值得优化的地方。

如果你需要一个明确的 FPS 指示器，可以尝试一下 [KMCGeigerCounter](#)。

对于 CPU 的卡顿，它可以通过内置的 CADisplayLink 检测出来；对于 GPU 带来的卡顿，它用了 1x1 的 SKView 来进行监视。这个项目有两个小问题：SKView 虽然能监视到 GPU 的卡顿，但引入 SKView 本身就会对 CPU/GPU 带来额外的一点的资源消耗；这个项目在 iOS 9 下有一些兼容问题，需要稍作调整。

我自己也写了个简单的 FPS 指示器：[FPSLabel](#) 只有几十行代码，仅用到了 CADisplayLink 来监视 CPU 的卡顿问题。虽然不如上面这个工具完善，但日常使用没有太大问题。

最后，用 Instruments 的 GPU Driver 预设，能够实时查看到 CPU 和 GPU 的资源消耗。在这个预设内，你能查看到几乎所有与显示有关的数据，比如 Texture 数量、CA 提交的频率、GPU 消耗等，在定位界面卡顿的问题时，这是最好的工具。