**iOS 保持界面流畅的技巧**

1.演示项目

2.屏幕显示图像的原理

3.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

4.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

5.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

6.如何评测界面的流畅度

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）

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

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

**对象的创建**

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**

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**

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

**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**

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

**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

**文本渲染**

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

**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 CGBitmapContext 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

**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**

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

**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

**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

**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：**

**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。**

**ASDK 异步并发操作,** 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

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。

**微博 Demo 性能优化技巧**

**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。

**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。

**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**

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

**其他增加的优化**

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**。**

**如何评测界面的流畅度**

对于 CPU 的卡顿，它可以通过内置的 CADisplayLink 检测出来