UITableView 是开发中经常会用到的控件，UITableView 中由一个个的 UITableViewCell 组成。对于一些简单的 UITableViewCell ，快速滑动时没什么问题。然而对于比较复杂的 UITableViewCell ，比如微博的 cell，cell 里面包含的内容非常多，有图片（图片数量不固定，不同数量对应的布局也不同），有文案（文案的长度不固定），还有很多小的控件，这种复杂的 cell 快速滑动起来时，可能会有卡顿的感觉。那么对于一些元素比较多的，布局比较复杂的 UITableViewCell，应该如何来优化呢?

fps

在开始介绍UITableView性能优化之前，先了解一下fps的概念。

fps 全称 frame per second。可以理解为每秒钟的帧率，也即每秒钟屏幕刷新的次数。每秒钟帧数越多，所显示的动作就越流畅。iOS 设备每秒钟刷新的频率是60，通常情况下，iOS 应用fps保持在50-60之间不会感觉到卡顿，否则可能会感觉到卡顿。 Wiki 中 fps的定义如下：

> Frame rate (expressed in frames per second or FPS) is the frequency (rate) at which consecutive images called frames are displayed in an animated display. The term applies equally to film and video cameras, computer graphics, and motion capture systems. Frame rate may also be called the frame frequency, and be expressed in hertz.

可以将fps作为衡量app是否流畅的工具。那么，如何检测fps呢?

检测fps

检测fps有两种方法，一种是使用Xcode自带的工具，另一种是使用CADisplayLink实现。

Xcode-Instruments

使用Xcode-Instruments中的 Core Animation Frames Per Second可以看到fps的值，调试App时，启动Instruments，操作App，fps的值可以实时的显示出来。

这种方法的缺点是必须启动Instruments,相对来说没那么方便；优点是比较准确。

CADisplayLink

CADisplayLink是系统提供的定时器，其特点是和屏幕刷新频率相同。即屏幕刷新一次，定时器执行一次。理想情况下，1秒钟内，屏幕刷新60次，定时器也应该执行60次。当屏幕丢帧，屏幕不刷新，定时器也不执行。根据此原理，可以创建CADisplayLink，因为是定时器，所以同样需要添加到RunLoop中，提供target和selector。在selector中设置计数器，每执行一次，计数器增加一次。统计1秒钟内计数器的值，基本上就是屏幕刷新的次数，也就是fps值。

使用CADisplayLink检测fps的示例代码如下：

```
CADisplayLink *link = [CADisplayLink displayLinkWithTarget:self
selector:@selector(linkValueAdd)];
self.link = link;
[link addToRunLoop:[NSRunLoop mainRunLoop]
forMode:NSRunLoopCommonModes];

- (void)linkValueAdd
{
    if(self.lastTime == 0){
        self.lastTime = self.link.timestamp;
    }
    self.count++;
    NSTimeInterval diff = self.link.timestamp - self.lastTime;
    if(diff > 1){
        self.lastTime = self.link.timestamp;
        self.count = 0;
    }
}
```

使用CADisplayLink的缺点是值不是非常精准，只是一个参考值；优点是不用启动Instruments，可以随时查看。

CPU && GPU

iOS系统中，图像显示到屏幕上，主要依赖于CPU和GPU工作。CPU主要负责计算，GPU主要负责渲染，当CPU或者GPU负载过大，来不及计算、渲染时，就会丢帧，也就产生了卡顿。

因此，优化UITableView，主要从CPU和GPU两个方面来进行优化。

工具

优化UITableView需要有针对性，找到哪些代码耗费时间，对GPU的压力较大，有针对性的优化。主要使用的工具是Instruments。Instruments中的TimeProfile可以看到代码中每个方法的执行时间，找到耗时的方法，针对性优化；Instruments中的CoreAnimation可以看到图层相关的内容，找到耗费GPU的地方，针对性优化。

CPU优化

以下列出一些针对CPU的优化措施，有一些是很小的tip，不要感觉是小tip就不在意，积少成多，把每个小tip改了，就可以得到一个流畅的UITableView。

高度计算优化

由于cell中内容不同，高度也不同，且每次滑动UITableView，都需要计算很多cell的高度，因此计算cell高度是一块比较耗时的操作。

优化计算高度的操作有：

1. 计算cell高度时，变量能计算一次就计算一次，节省时间，如屏幕宽度，_kScreenWidth = [UIScreen mainScreen].bounds.size.width; 使用_kScreenWidth，而不是每次都使用 [UIScreen mainScreen].bounds.size.width 。
2. 避免动态添加控件、图层。在初始化cell的时候，可以将所有的图层都创建好，在cell中通过hidden属性来控制显示还是隐藏。因为单纯的显示隐藏操作要比创建节省很多时间。
3. 可以缓存cell的高度，以及cell内部控件的frame信息，确保只计算一次，不会重复计算（该方法是以空间换时间）
4. 动态计算cell高度时，可以使用runloop，在空闲时计算高度

GPU优化

以下列出一些针对GPU的优化，GPU优化主要是从渲染的角度考虑：

1. 尽量减少图层混合，可以使用Xcode中的CoreAnimation查看，红色图层为混合，绿色图层为不混合。绿色越多越好。
2. 对于一些不需要触摸事件的元素，可以用ASDK的图层合成技术预先绘制为一张图。
3. 异步绘制。通过重写drawRect方法，使用Core Graphics框架中的API进行异步绘制，提高效率。另外drawRect中大量的绘制操作也会造成内存的增长，可以使用CAShapeLayer来代替。
4. view尽量不设置透明度，可以设置默认背景色，减少图层混合
5. 尽量减少控件，能用一个控件的，不要用多个控件
6. ix. 画圆角的操作，可以通过贝塞尔曲线+Core Graphics框架设置圆角

```
- (void)setImageCircularEdge:(UIImageView *)imageView {

    //开始对imageView进行画图
    UIGraphicsBeginImageContextWithOptions(imageView.bounds.size,
NO, 1.0);
    //使用贝塞尔曲线画出一个圆形图
    [[UIBezierPath bezierPathWithRoundedRect:imageView.bounds
cornerRadius:imageView.frame.size.width] addClip];
    [imageView drawRect:imageView.bounds];
    imageView.image = UIGraphicsGetImageFromCurrentImageContext();
    //结束画图
    UIGraphicsEndImageContext();
}
```

也可以通过贝塞尔曲线+CAShapeLayer设置圆角：

```
- (void)setImageCircularEdge2:(UIImageView *)imageView {

    UIBezierPath *maskPath = [UIBezierPath
bezierPathWithRoundedRect:imageView.bounds
byRoundingCorners:UIRectCornerAllCorners
cornerRadii:imageView.bounds.size];
    CAShapeLayer *maskLayer=[[CAShapeLayer alloc] init];
    //设置大小
    maskLayer.frame = imageView.bounds;
    //设置图形样子
    maskLayer.path = maskPath.CGPath;
    imageView.layer.mask = maskLayer;
}
```

也可以直接在图片上覆盖一个内部透明圆的图片。目的都是避免离屏渲染

策略上优化

除CPU、GPU外，还有一些通用的优化方式，可以归类为策略上的优化，如下：

1. 分页加载数据，避免一次获取的数据太多，需要渲染的数据太多
2. 预先异步请求数据。滑动时为避免卡顿，不必要等待数据全部显示完时再请求数据，而是快要显示完时，预先请求数据，减少用户等待时间，给用户一个流畅的体验
3. 将数据保存到本地的操作放到子线程
4. 使用timeProfiler查看消耗时间的代码，针对耗时较长的操作可以确认是否可以放到子线程
5. 尽量使用轻量级对象代替重量对象，比如说CALayer要比UIView轻许多。当不需要响应触摸事件的时候，使用CALayer会更合适。

相关知识点

UIView和CALayer的关系

两者的区别和联系如下：

1. UIView继承自UIResponder，CALayer继承自NSObject。因此，UIView可以响应事件，CALayer不能响应事件
2. UIView主要是对显示内容的管理，CALayer主要是对内容的绘制
3. UIView是CALayer的代理（CALayerDelegate）
4. 每个UIView内部都有一个CALayer提供绘制和显示。实际上，UIView的frame等属性，都是CALayer提供的。
5. UIView和CALayer的结构类似，都是树状结构，UIView可以有subview，CALayer也可以有subLayer

离屏渲染

什么是离屏渲染

通常情况下，iOS系统中，GPU只有一块缓存区（当前屏幕的缓存区）用于渲染。但是，在一些特殊操作，如设置圆角时，GPU在当前的缓存区不能完成渲染，就会另外开辟一块缓存区来渲染，这就是离屏渲染。

离屏渲染耗性能主要有两方面原因：

1. 开辟一块新的缓存区本身就非常耗性能
2. 在新旧缓存区之间切换耗性能

离屏渲染是怎么引起的

设置圆角，如UIImageView、UIButton设置圆角(CornerRadius 和 masksToBounds 一起使用时)，会引起离屏渲染。