IM UI性能优化之异步绘制

2016年7月5日

重构完Socket之后，最近我们也开始针对IM的UI做了优化，这次的优化我们主要是参考了YYKit对于性能方面的优化，前期我的另一个小伙伴西兰花也对AsyncDisplayKit做了调研，不过这个库理解起来确实要费一番功夫，由于YYkit的核心思路基本上都是学习AsyncDisplayKit的，相信YYkit这个库大家都已经很熟悉了，不过可能还没有看过这个库，那下面我做一个简单的介绍

YYKit的作者是郭曜源，YYKit实际上是将它那些单独的iOS组件整合在了一起，类似于集合一样组成功能比较全面的组件，你也可以根据自己业务的需要单独使用其中的某些部分

0x00 前期准备

我们首先阅读了郭曜源在对界面流畅性方面的见解，里面提到了 `异 步绘 制` ，但是文字表述毕竟是抽象的，然后我们简单看了下他的YYText和YYAsyncLayer组件，看完之后实际上对如何使用他的YYAsyncLayer这个组件来实现异步绘制还是有点模糊的，后来我们直接看他的微博demo，我们逐渐理清了他是如何实现异步绘制以及几个性能优化方面的点

因为YYLabel Async Display里面加了是否异步绘制开关，所以我们直接用这个例子作为对比，首先我们来看下异步绘制的效果，开始的时候我们关闭异步绘制的开关，你会发现FPS瞬间掉到6了，屏幕滚动开始非常卡，但是打开开关之后，滚动时虽然FPS还是会掉到30-40，但是滑动的流畅度比之前要好很多，感觉这异步绘制的效果杠杠的好啊，那我们一定要看看他是怎么做的了

0x01 分析

其实整一个性能优化关键的点及流程有三个：

1.数据源的异步处理

当我们获取到数据源的时候，我们需要对数据源进行计算处理，计算出UI绘制所需要的属性比如宽高、颜色等等，而且这些计算要异步去做，否则会卡住主线程，等这些数据源计算完成之后，再去处理绘制，但是如果数据源过大，计算的耗时还是在的，所以会有较长时间的等待时间，此时我们需要考虑加上等待的友好处理

2.采用更轻量级的绘制

在绘制时，对于不需要响应触摸事件的控件，我们应该尽量避免创建UIView对象，取而代之的是使用更为轻量的CALayer，并且对于一个layer包含多个subLayer的情况时，我们可以通过图层预合成的方法，将多个subLayer合成渲染成一张图片，通过上述的处理，不仅能减少CPU在创建UIKit对象的消耗，还能减少GPU在合成和渲染上的消耗，内存的占用也会少很多

3.异步绘制

我们将使用 `YYAsyncLayer` 组件实现异步绘制

0x02 YYAsyncLayer介绍

前面两个优化点，平时在做的时候可能也都会去做，但是异步绘制这个该怎么去实现呢？我们直接来看下 `YYAsyncLayer` 的代码

`YYAsyncLayer` 组件里面一共包含了三个类：`YYAsyncLayer` 、`YYSentinel` 、`YYTransaction`

`YYAsyncLayer` 类是我们主要用的类，它是CALayer的子类，是用来异步渲染layer内容

`YYSentinel` 类是用来给线程安全计数的，用于在多线程处理的场景

`YYTransaction` 类是利用runloop在休眠前的空闲时间来触发你预设的方法

因为我们没有用到 `YYTransaction` 类，所以我们直接将 `YYAsyncLayer` 、`YYSentinel` 合成一个类，并做了混淆，这样可以少引用一个库

我们首先来看 `YYAsyncLayer` 的头文件

`YYAsyncLayer` 类只有一个 `displaysAsynchronously` 属性，就是设置渲染是否是异步执行的

```
@property BOOL displaysAsynchronously;
```

然后还有个代理方法，这个代理方法的触发时机是在layer的内容需要更新的时候，此时你有个新的绘制任务，然后返回的是个 `YYAsyncLayerDisplayTask` 对象

```
- (YYAsyncLayerDisplayTask *)newAsyncDisplayTask;
```

`YYAsyncLayerDisplayTask` 类只有三个block，即将绘制、绘制中、绘制完成

```
@property (nullable, nonatomic, copy) void (^willDisplay)(CALayer *layer);
@property (nullable, nonatomic, copy) void (^display)(CGContextRef context, CGSize
@property (nullable, nonatomic, copy) void (^didDisplay)(CALayer *layer, BOOL finis
```

看到实现文件里面，触发这个代理的方法是 `- setNeedsDisplay` 方法，就是当layer需要更新内容的时候，它会向代理发起一个异步绘制的请求，将内容的渲染放到后台队列去做，所以我们在使用 `YYAsyncLayer` 类时，我们需要重写 `+ layerClass` 方法，返回 `YYAsyncLayer` 类，否则会直接调用 `CALayer` 的方法，不会触发代理

```
- (void)setNeedsDisplay {
    [self _cancelAsyncDisplay];
    [super setNeedsDisplay];
}

- (void)display {
    super.contents = super.contents;
    [self _displayAsync:_displaysAsynchronously];
}

#pragma mark - Private

- (void)_displayAsync:(BOOL)async {
    __strong id<YYAsyncLayerDelegate> delegate = self.delegate;
    YYAsyncLayerDisplayTask *task = [delegate newAsyncDisplayTask];
   // ...
}
```

在 `- _displayAsync` 方法里面主要分成三部分：

如果没有设置display回调，layer的内容会被清空

```
if (!task.display) {
    if (task.willDisplay) task.willDisplay(self);
    self.contents = nil;
    if (task.didDisplay) task.didDisplay(self, YES);
    return;
}
```

根据之前 `displaysAsynchronously` 属性设置判断，如果是同步绘制的话，实际上的操作就是在调用完 `display` block之后，将sublayer合成一张图作为layer的内容

```
[self increase];
if (task.willDisplay) task.willDisplay(self);
UIGraphicsBeginImageContextWithOptions(self.bounds.size,self.opaque,self.contentsS
CGContextRef context = UIGraphicsGetCurrentContext();
task.display(context, self.bounds.size, ^{return NO;});
UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
self.contents = (__bridge id)(image.CGImage);
if (task.didDisplay) task.didDisplay(self, YES);
```

而异步渲染的处理和同步渲染大同小异，第一，多了一个 `BOOL (^isCancelled)()` block，这个block的好处是，在 `display` block调用绘制前，可以通过判断 `isCancelled` 布尔值的值来停止绘制，减少性能上的消耗，以及避免出现线程阻塞的情况，比如TableView快速滑动的时候，就可以通过这样的判断，来避免不必要的绘制，提升滑动的流畅性，第二，将上面同步的绘制处理放到了异步去做，绘制方式是一样的

```
if (task.willDisplay) task.willDisplay(self);
int32_t value = self.value;
BOOL (^isCancelled)() = ^BOOL() {
    return value != self.value;
};
CGSize size = self.bounds.size;
BOOL opaque = self.opaque;
CGFloat scale = self.contentsScale;
if (size.width < 1 || size.height < 1) {
    CGImageRef image = (__bridge_retained CGImageRef)(self.contents);
    self.contents = nil;
    if (image) {
        dispatch_async(FIMAsyncLayerGetReleaseQueue(), ^{
            CFRelease(image);
        });
    }
    if (task.didDisplay) task.didDisplay(self, YES);
    return;
}

dispatch_async(FIMAsyncLayerGetDisplayQueue(), ^{
    if (isCancelled()) return;
    UIGraphicsBeginImageContextWithOptions(size, opaque, scale);
    CGContextRef context = UIGraphicsGetCurrentContext();
    task.display(context, size, isCancelled);
    if (isCancelled()) {
        UIGraphicsEndImageContext();
```

```
        dispatch_async(dispatch_get_main_queue(), ^{
            if (task.didDisplay) task.didDisplay(self, NO);
        });
        return;
    }
    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    if (isCancelled()) {
        dispatch_async(dispatch_get_main_queue(), ^{
            if (task.didDisplay) task.didDisplay(self, NO);
        });
        return;
    }
    dispatch_async(dispatch_get_main_queue(), ^{
        if (isCancelled()) {
            if (task.didDisplay) task.didDisplay(self, NO);
        } else {
            self.contents = (__bridge id)(image.CGImage);
            if (task.didDisplay) task.didDisplay(self, YES);
        }
    });
});
```

这个异步的队列也是自己创建的，在预设了一个队列最大值之后，通过获取运行该进程的系统处于激活状态的处理器数量来创建队列，使得绘制的效率达到最高

```
static dispatch_queue_t FIMAsyncLayerGetDisplayQueue() {
#define MAX_QUEUE_COUNT 16
    static int queueCount;
    static dispatch_queue_t queues[MAX_QUEUE_COUNT];
    static dispatch_once_t onceToken;
    static int32_t counter = 0;
    dispatch_once(&onceToken, ^{
        queueCount = (int)[NSProcessInfo processInfo].activeProcessorCount;
        queueCount = queueCount < 1 ? 1 : queueCount > MAX_QUEUE_COUNT ? MAX_QUEUE_
        if ([UIDevice currentDevice].systemVersion.floatValue >= 8.0) {
            for (NSUInteger i = 0; i < queueCount; i++) {
                dispatch_queue_attr_t attr = dispatch_queue_attr_make_with_qos_class(DISPA
                queues[i] = dispatch_queue_create("com.ibireme.FIMkit.render", att
            }
        } else {
            for (NSUInteger i = 0; i < queueCount; i++) {
                queues[i] = dispatch_queue_create("com.ibireme.FIMkit.render", DISPATC
                dispatch_set_target_queue(queues[i], dispatch_get_global_queue(DISPATCH
            }
        }
    });
    int32_t cur = OSAtomicIncrement32(&counter);
    if (cur < 0) cur = -cur;
    return queues[(cur) % queueCount];
#undef MAX_QUEUE_COUNT
}
```

0x03 补充

在 `文 本` 的实现上，我们更加推荐使用CoreText，CoreText对象占用的内存少，而且适用于文本排版复杂的情况，虽然在实现上较为复杂，但是所带来的好处远远要多

在渲染 `图片` 时，我们应该在后台把图片绘制到 `CGBitmapContext` 中，然后从 `Bitmap` 直接创建图片，因为如果使用原来ImageView读取Image的方式是，在创建Image或者CGImageSource对象时，图片数据并不会立即解码，而是等到设置到ImageView或者layer.contents，layer被提交到GPU之前，才解码，并且这些操作都是在主线程进行，是相当耗性能的，所以我们应该用推荐的方式去绘制，而且AFNetworking在对图片处理的时候也是这么做的

0x04 简单实现demo

对于上述优化点，我实现了一个简单的CoreText demo，可以看一下这个demo做进一步了解~

0x05 相关推荐阅读

iOS 保持界面流畅的技巧

iOS 事件处理机制与图像渲染过程&version=12000110&lang=zh_CN&nettype=WIFI&fontScale=100&pass_ticket=JE1V0DIEopWtscTKwaYEiHN6qmvNRu9O60t4vUkn3Ek%3D)

PREV NEXT