UIScrollView 实践经验

Allen 许帅 15 December 2014

`UIScrollView`（包括它的子类 `UITableView` 和 `UICollectionView`）是 iOS 开发中最常用也是最有意思的 UI 组件，大部分 App 的核心界面都是基于三者之一或三者的组合实现。`UIScrollView` 是 `UIKit` 中为数不多能响应滑动手势的 view，相比自己用 `UIPanGestureRecognizer` 实现一些基于滑动手势的效果，用 `UIScrollView` 的优势在于 bounce 和 decelerate 等特性可以让 App 的用户体验与 iOS 系统的用户体验保持一致。本文通过一些实例讲解 `UIScrollView` 的特性和实际使用中的经验。

iPhone 5 刚出来的时候，大部分不支持横屏的 App 都不需要做太多的适配工作，因为屏幕宽度没有变，table view 多个 cell 也不需要加 code。但是在 iPhone 6 和 iPhone 6 Plus 发布以后，多分辨率适配终于不再是 Android 开发的专利了。于是，从 iOS 6 起就存在的 Auto Layout 终于有了用武之地。

关于 Auto Layout 的基本用法不再赘述，可以参考 Ray Wenderlich 上的教程（Part 2）。但 `UIScrollView` 在 Auto Layout 是一个很特殊的 view，对于 `UIScrollView` 的 subview 来说，它的 leading/trailing/top/bottom space 是相对于 `UIScrollView` 的 contentSize 而不是 bounds 来确定的，所以当你尝试用 `UIScrollView` 和它 subview 的 leading/trailing/top/bottom 来互相决定大小的时候，就会出现「Has ambiguous scrollable content width/height」的 warning。正确的姿势是用 `UIScrollView` 外部的 view 或 `UIScrollView` 本身的 width/height 确定 subview 的尺寸，进而确定 `contentSize`。因为 `UIScrollView` 本身的 leading/trailing/top/bottom 变得不好用，所以我习惯的做法是在 `UIScrollView` 和它原来的 subviews 之间增加一个 content view，这样做的好处有：

- 不会在 storyboard 里留下 error/warning
- 为 subview 提供 leading/trailing/top/bottom，方便 subview 的布局
- 通过调整 content view 的 size（可以是 constraint 的 `IBOutlet`）来调整 `contentSize`
- 不需要 hard code 与屏幕尺寸相关的代码

- 更好地支持 rotation

Sample 中的 AutoLayout 演示了 `UIScrollView` + Auto Layout 的例子。

UIScrollViewDelegate

`UIScrollViewDelegate` 是 `UIScrollView` 的 delegate protocol，`UIScrollView` 有意思的功能都是通过它的 delegate 方法实现的。了解这些方法被触发的条件及调用的顺序对于使用 `UIScrollView` 是很有必要的，本文主要讲拖动相关的效果，所以 zoom 相关的方法跳过不提，拖动相关的 delegate 方法按调用顺序分别是：

```
- (void)scrollViewDidScroll:(UIScrollView *)scrollView
```

这个方法在任何方式触发 `contentOffset` 变化的时候都会被调用（包括用户拖动，减速过程，直接通过代码设置等），可以用于监控 `contentOffset` 的变化，并根据当前的 `contentOffset` 对其他 view 做出随动调整。

```
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
```

用户开始拖动 scroll view 的时候被调用。

```
- (void)scrollViewWillEndDragging:(UIScrollView *)scrollView
withVelocity:(CGPoint)velocity targetContentOffset:(inout CGPoint
*)targetContentOffset
```

该方法从 iOS 5 引入，在 didEndDragging 前被调用，当 willEndDragging 方法中 `velocity` 为 `CGPointZero`（结束拖动时两个方向都没有速度）时，didEndDragging 中的 `decelerate` 为 `NO`，即没有减速过程，willBeginDecelerating 和 didEndDecelerating 也就不会被调用。反之，当 `velocity` 不为 `CGPointZero` 时，scroll view 会以 `velocity` 为初速度，减速直到 `targetContentOffset`。值得注意的是，这里的 `targetContentOffset` 是个指针，没错，你可以改变减速运动的目的地，这在一些效果的实现时十分有用，实例章节中会具体提到它的用法，并和其他实现方式作比较。

```
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
```

```
willDecelerate:(BOOL)decelerate
```

在用户结束拖动后被调用，decelerate 为 YES 时，结束拖动后会有减速过程。注，在 didEndDragging 之后，如果有减速过程，scroll view 的 dragging 并不会立即置为 NO，而是要等到减速结束之后，所以**这个 dragging 属性的实际语义更接近 scrolling**。

```
- (void)scrollViewWillBeginDecelerating:(UIScrollView
*)scrollView
```

减速动画开始前被调用。

```
- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
```

减速动画结束时被调用，这里有一种特殊情况：当一次减速动画尚未结束的时候再次 drag scroll view，didEndDecelerating 不会被调用，**并且这时 scroll view 的 `dragging` 和 `decelerating` 属性都是 `YES`**。新的 dragging 如果有加速度，那么 willBeginDecelerating 会再一次被调用，然后才是 didEndDecelerating；如果没有加速度，虽然 willBeginDecelerating 不会被调用，但前一次留下的 didEndDecelerating 会被调用，所以连续快速滚动一个 scroll view 时，delegate 方法被调用的顺序（不含 didScroll）可能是这样的：

```
scrollViewWillBeginDragging:
scrollViewWillEndDragging: withVelocity: targetContentOffset:
scrollViewDidEndDragging: willDecelerate:
scrollViewWillBeginDecelerating:
scrollViewWillBeginDragging:
scrollViewWillEndDragging: withVelocity: targetContentOffset:
scrollViewDidEndDragging: willDecelerate:
scrollViewWillBeginDecelerating:
...
scrollViewWillBeginDragging:
scrollViewWillEndDragging: withVelocity: targetContentOffset:
scrollViewDidEndDragging: willDecelerate:
scrollViewWillBeginDecelerating:
scrollViewDidEndDecelerating:
```

虽然很少有因为这个导致的 bug，但是你需要知道这种很常见的用户操作会导致的中间状态。例如你尝试在 UITableViewDataSource 的

`tableView:cellForRowAtIndexPath:` 方法中基于 tableView 的 dragging 和 decelerating 属性判断是在用户拖拽还是减速过程中的话可能会误判（见例 1）。

Sample 中的 Delegate 简单输出了一些 Log，你可以快速了解这些方法的调用顺序。

实例

下面通过一些实例，更详细地演示和描述以上各 delegate 方法的用途。

1. Table View 中图片加载逻辑的优化

虽然这种优化方式在现在的机能和网络环境下可能看似不那么必要，但在我最初看到这个方法是的 09 年（印象中是 Tweetie 作者在 08 年写的 Blog，可能有误），遥想 iPhone 3G/3GS 的机能，这个方法为多图的 table view 的性能带来很大的提升，也成了我的秘密武器。而现在，在移动网络环境下，你依然值得这么做来为用户节省流量。

先说一下原文的思路：

- 当用户手动 drag table view 的时候，会加载 cell 中的图片；
- 在用户快速滑动的减速过程中，不加载过程中 cell 中的图片（但文字信息还是会被加载，只是减少减速过程中的网络开销和图片加载的开销）；
- 在减速结束后，加载所有可见 cell 的图片（如果需要的话）；

问题 1：

前面提到，刚开始拖动的时候，`dragging` 为 `YES`，`decelerating` 为 `NO`；decelerate 过程中，`dragging` 和 `decelerating` 都为 `YES`；decelerate 未结束时开始下一次拖动，`dragging` 和 `decelerating` 依然都为 `YES`。所以无法简单通过 table view 的 `dragging` 和 `decelerating` 判断是在用户拖动还是减速过程。

解决这个问题很简单，添加一个变量如 `userDragging`，在

willBeginDragging 中设为 YES，didEndDragging 中设为 NO。那么 tableView: cellForRowAtIndexPath: 方法中，是否 load 图片的逻辑就是：

```
if (!self.userDragging && tableView.decelerating) {
    cell.imageView.image = nil;
} else {
    // code for loading image from network or disk
}
```

问题 2：

这么做的话，decelerate 结束后，屏幕上的 cell 都是不带图片的，解决这个问题也不难，你需要一个形如 loadImageForVisibleCells 的方法，加载可见 cell 的图片：

```
- (void)loadImageForVisibleCells
{
    NSArray *cells = [self.tableView visibleCells];
    for (GLImageCell *cell in cells) {
        NSIndexPath *indexPath = [self.tableView indexPathForCell:cell];
        [self setupCell:cell withIndexPath:indexPath];
    }
}
```

问题 3：

这个问题可能不容易被发现，在减速过程中如果用户开始新的拖动，当前屏幕的 cell 并不会被加载（前文提到的调用顺序问题导致），而且问题 1 的方案并不能解决问题 3，因为这些 cell 已经在屏上，不会再次经过 cellForRowAtIndexPath 方法。虽然不容易发现，但解决很简单，只需要在 scrollViewWillBeginDragging: 方法里也调用一次 loadImageForVisibleCells 即可。

再优化

上述方法在那个年代的确提升了 table view 的 performance，但是你会发现在减速过程最后最慢的那零点几秒时间，其实还是会让人等得有些心急，尤其

如果你的 App 只有图片没有文字。在 iOS 5 引入了 `scrollViewWillEndDragging: withVelocity: targetContentOffset:` 方法后，配合 `SDWebImage`，我尝试再优化了一下这个方法以提升用户体验：

- 如果内存中有图片的缓存，减速过程中也会加载该图片
- 如果图片属于 `targetContentOffset` 能看到的 cell，正常加载，这样一来，快速滚动的最后一屏出来的的过程中，用户就能看到目标区域的图片逐渐加载
- 你可以尝试用类似 fade in 或者 flip 的效果缓解生硬的突然出现（尤其是像本例这样只有图片的 App）

核心代码：

```
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
{
    self.targetRect = nil;
    [self loadImageForVisibleCells];
}

- (void)scrollViewWillEndDragging:(UIScrollView *)scrollView withVelocity:(
{
    CGRect targetRect = CGRectMake(targetContentOffset->x, targetContentOff
    self.targetRect = [NSValue valueWithCGRect:targetRect];
}

- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    self.targetRect = nil;
    [self loadImageForVisibleCells];
}
```

是否需要加载图片的逻辑：

```
BOOL shouldLoadImage = YES;
if (self.targetRect && !CGRectIntersectsRect([self.targetRect CGRectValue],
    SDImageCache *cache = [manager imageCache];
    NSString *key = [manager cacheKeyForURL:targetURL];
    if (![cache imageFromMemoryCacheForKey:key]) {
        shouldLoadImage = NO;
    }
}
if (shouldLoadImage) {
```

```
    // load image
}
```

更值得高兴的是，通过判断是否 `nil`，`targetRect` 同时起到了原来 `userDragging` 的作用。本例完整的代码见 Sample 中的 LazyLoad

2. 分页的几种实现方式

利用 UIScrollView 有多种方法实现分页，但是各自的效果和用途不尽相同，其中方法 2 和方法 3 的区别也正是一些同类 App 在模仿 Glow 的首页 Bubble 翻转效果时跟 Glow 体验上的的差距所在（但愿他们不会看到本文并且调整他们的实现方式）。本例通过三种方法实现相似的一个场景，你可以通过安装到手机上来感受三种实现方式的不同用户体验。为了区分每个例子的重点，本例没有重用机制，重用相关内容见例 3。

2.1 pagingEnabled

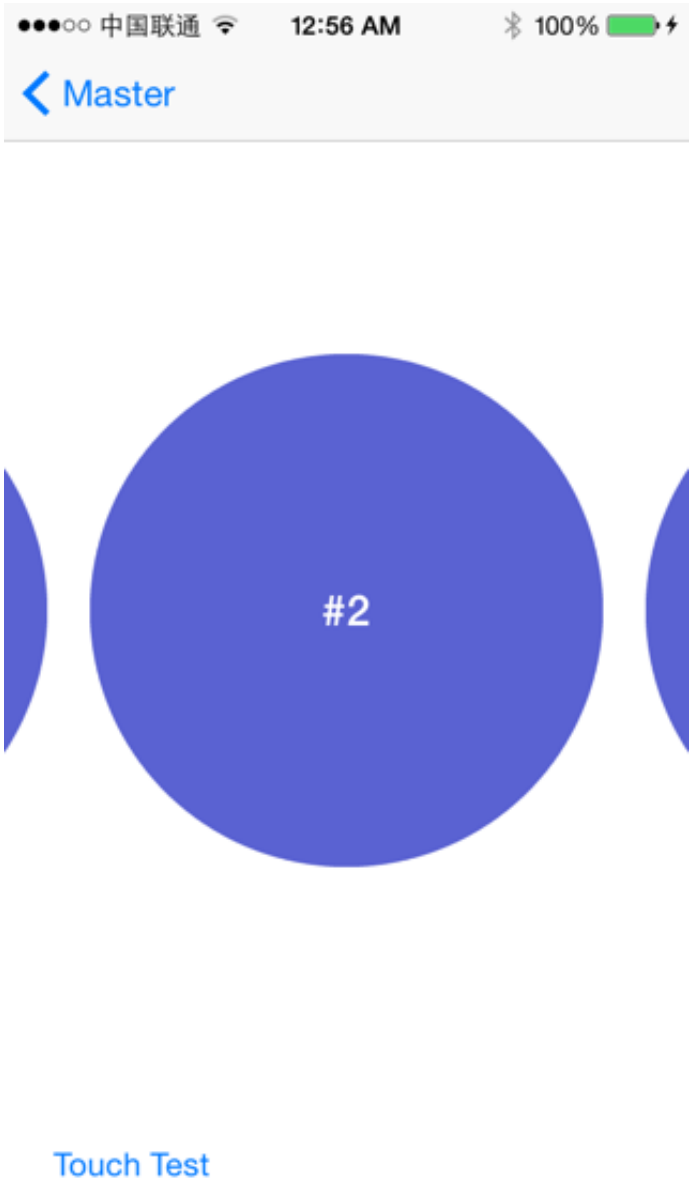
这是系统提供的分页方式，最简单，但是有一些局限性：

- 只能以 frame size 为单位翻页，减速动画阻尼大，减速过程不超过一页
- 需要一些 hacking 实现 bleeding 和 padding（即页与页之间有 padding，在当前页可以看到前后页的部分内容）

Sample 中 Pagination 有简单实现 bleeding 和 padding 效果的代码，主要的思路是：

- 让 scroll view 的宽度为 page 宽度 + padding，并且设置 `clipsToBounds` 为 `NO`
- 这样虽然能看到前后页的内容，但是无法响应 touch，所以需要另一个覆盖期望的可触摸区域的 view 来实现类似 touch bridging 的功能

适用场景：上述局限性同时也是这种实现方式的优点，比如一般 App 的引导页（教程），Calendar 里的月视图，都可以用这种方法实现。



2.2 Snap

这种方法就是在 didEndDragging 且无减速动画，或在减速动画完成时，snap 到一个整数页。核心算法是通过当前 contentOffset 计算最近的整数页及其对应的 contentOffset，通过动画 snap 到该页。这个方法实现的效果都有个通病，就是最后的 snap 会在 decelerate 结束以后才发生，总感觉很突兀。

2.3 修改 targetContentOffset

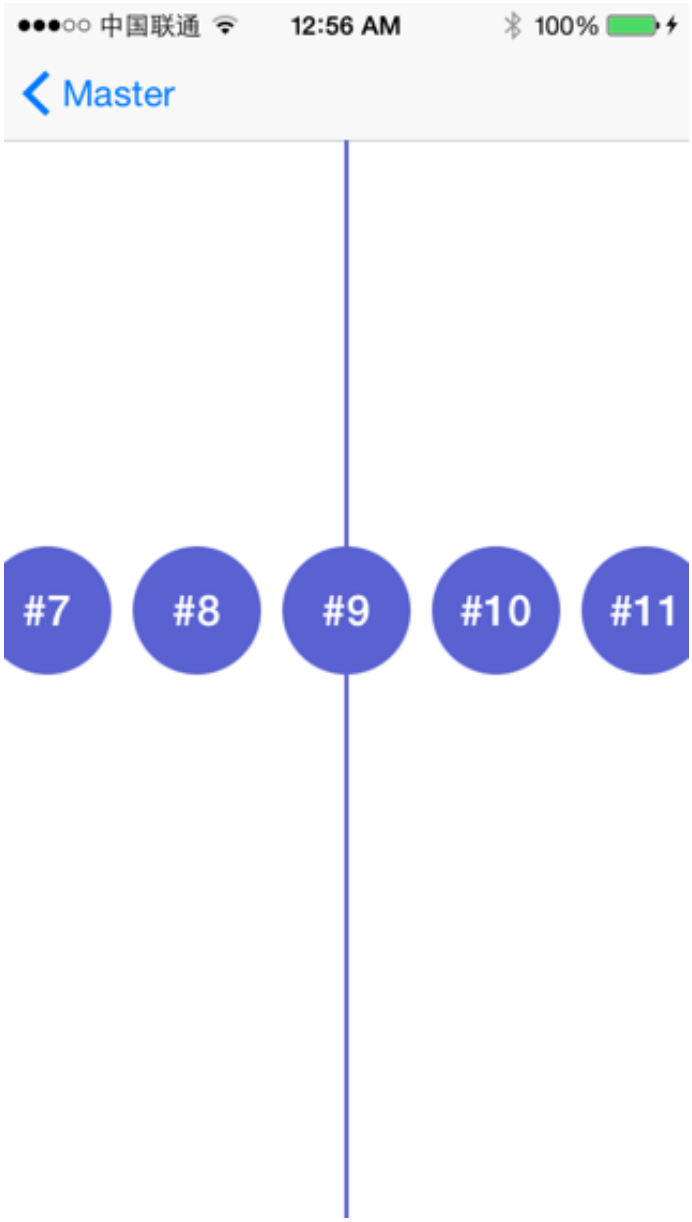
通过修改 `scrollViewWillEndDragging: withVelocity: targetContentOffset:` 方法中的 `targetContentOffset` 直接修改目标 offset 为整数页位置。其中核心代码：

```
- (CGPoint)nearestTargetOffsetForOffset:(CGPoint)offset
{
    CGFloat pageSize = BUBBLE_DIAMETER + BUBBLE_PADDING;
    NSInteger page = roundf(offset.x / pageSize);
    CGFloat targetX = pageSize * page;
    return CGPointMake(targetX, offset.y);
}

- (void)scrollViewWillEndDragging:(UIScrollView *)scrollView withVelocity:(
{
    CGPoint targetOffset = [self nearestTargetOffsetForOffset:*targetConten
    targetContentOffset->x = targetOffset.x;
    targetContentOffset->y = targetOffset.y;
}
```

适用场景：方法 2 和 方法 3 的原理近似，效果也相近，适用场景也基本相同，但方法 3 的体验会好很多，snap 到整数页的过程很自然，或者说用户完全感知不到 snap 过程的存在。这两种方法的减速过程流畅，适用于一屏有多页，但需要按整数页滑动的场景；也适用于如图表中自动 snap 到整数天的场景；还适用于每页大小不同的情况下 snap 到整数页的场景（不做举例，自行发挥，其实只需要修改计算目标 offset 的方法）。

完整代码参见 Pagination



3. 重用

大部分的 iOS 开发应该都清楚 `UITableView` 的 cell 重用机制，这种重用机制减少了内存开销也提高了 performance，`UIScrollView` 作为 `UITableView` 的父类，在很多场景中也很适合应用重用机制（其实不只是 `UIScrollView`，任何场景中会反复出现的元素都应该适当地引入重用机制）。

你可以参照 `UITableView` 的 cell 重用机制，总结重用机制如下：

- 维护一个重用队列
- 当元素离开可见范围时，`removeFromSuperview` 并加入重用队列（enqueue）
- 当需要加入新的元素时，先尝试从重用队列获取可重用元素（dequeue）并且从重用队列移除

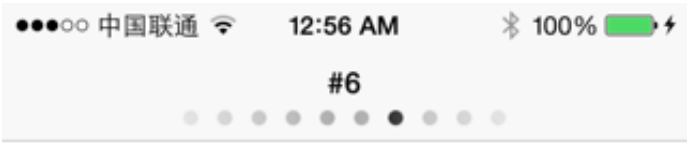
- 如果队列为空，新建元素
- 这些一般都在 `scrollViewDidScroll:` 方法中完成

实际使用中，需要注意的点是：

- 当重用对象为 view controller 时，记得 `addChildeViewController`
- 当 view 或 view controller 被重用但其对应 model 发生变化的时候，需要及时清理重用前留下的内容
- 数据可以适当做缓存，在重用的时候尝试从缓存中读取数据甚至之前的状态（如 table view 的 contentOffset），以得到更好的用户体验
- 当 on screen 的元素数量可确定的时候，有时候可以提前 init 这些元素，不会在 scroll 过程中遇到因为 init 开销带来的卡顿（尤其是以 view controller 为重用对象的时候）

例 2 中的场景很适合以 view 为重用单位，本例新增一个以 view controller 为重用对象的例子，该例子同时演示了联动效果，具体见下个例子。

完整代码参见 Reuse



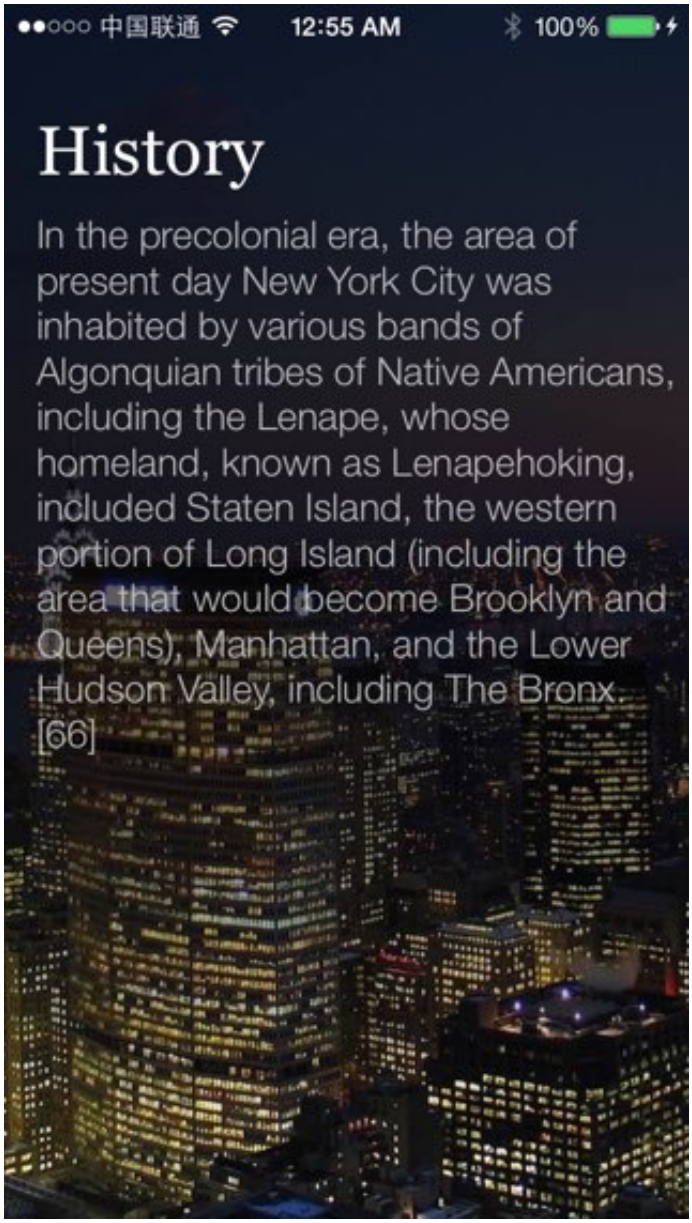
Page #6

Instance #0

4. 联动/视差滚动

上一个例子里 main scroll view 和 title view 里的 scroll view 就是一个联动的例子，所谓联动，就是当 A 滚动的时候，在 `scrollViewDidScroll:` 里根据 A 的 `contentOffset` 动态计算 B 的 `contentOffset` 并设给 B。同样对于非 scroll view 的 C，也可以动态计算 C 的 frame 或是 transform（Glow 的气泡为例）实现视差滚动或者其他高级动画，这在现在许多应用的引导页面里会被用到。

联动/视差滚动部分原理上其实比较简单，不再赘述，写了个简单的例子 Parallax。



写在最后

不知不觉就写了很多关于 `UIScrollView` 的内容，其实还有很多可写，由于时间关系只好停笔。在我看来，`UIScrollView` 就好像提供了一个跳脱二维空间束缚的途径，如果你有足够的想象力，它能帮你实现更丰富的跳出平面束缚的用户体验。本来还准备写一个综合性的例子，但是由于时间关系还没完成，后面有时间会继续更新。

此外，例子中可能会有错误或可以改进的地方，欢迎在 GitHub 直接提 Issue 或 PR。

iOS engineer at Glow, Inc., a great fan of Steve Jobs, Apple and Pixar.