复用的精妙 - UITableView 复用技术原理分析

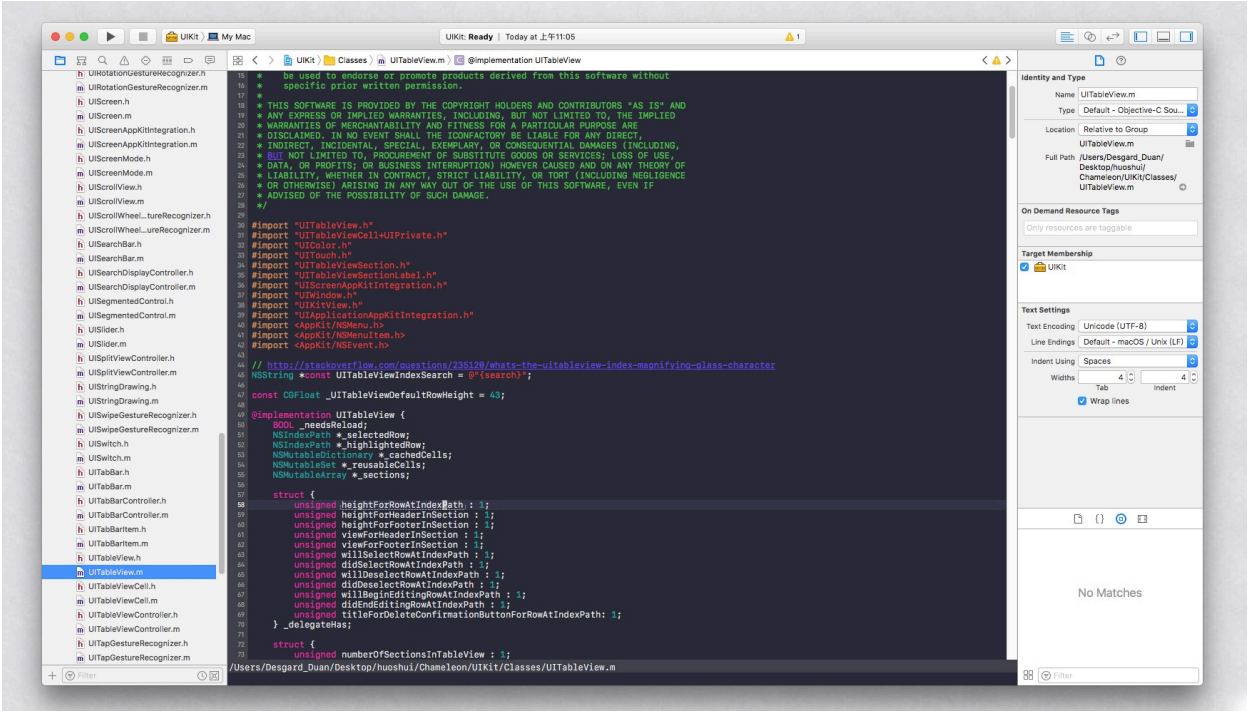
作者：冬瓜

在现在很多公司的 app 中，许多展示页面为了多条数据内容，而采用 `UITableView` 来设计页面。在滑动 `UITableView` 的时候，并不会因为数据量大而产生卡顿的情况，这正是因为其**复用机制**的特点。但是其复用机制是如何实现的？我决定来探索一番。

Chameleon PROJECT

Chameleon 是我长期以来一直关注的一个项目。接触过 macOS 开发的人肯定多少有写了解。（虽然这个项目在三年以前就已经停更，但是在原理上还是有很高的参考价值。）*Chameleon* 用于将 iOS 的功能迁移到 macOS 上，并且在其中为 macOS 实现了一套与 iOS UIKit 同名的框架，并且其代码都为开源。由于 *Chameleon* 属于对苹果早期源码的逆向工程项目，所以我们可以据此来对 iOS 一些闭源库展开学习和思路的借鉴。

> *Chameleon* 所迁移的 iOS 版本为 `3.2`，如今已经没有人使用，所以其代码和思路我们只能用来了解。例如在 iOS 8 之后推出的根据 `autoLayout` 自动计算 `cell` 高度的功能，在其中无法体现。



UITableView 的初始化方法

当我们定义一个 `UITableView` 对象的时候，需要对这个对象进行初始化。最常用的方法莫过于 `-(id)initWithFrame:(CGRect)frame style:(UITableViewStyle)theStyle`。下面跟着这个初始化入口，逐渐来分析代码：

```
- (id)initWithFrame:(CGRect)frame style:(UITableViewStyle)theStyle {
    if ((self=[super initWithFrame:frame])) {
        // 确定 TableView 的 Style
        _style = theStyle;
        // 要点一：Cell 缓存字典
        _cachedCells = [[NSMutableDictionary alloc] init];
        // 要点二：Section 缓存 Mutable Array
        _sections = [[NSMutableArray alloc] init];
        // 要点三：复用 Cell Mutable Set
        _reusableCells = [[NSMutableSet alloc] init];

        // 一些关于 Table View 的属性设置
        self.separatorColor = [UIColor colorWithRed:.88f green:.88f
blue:.88f alpha:1];
        self.separatorStyle = UITableViewCellSeparatorStyleSingleLine;
        self.showsHorizontalScrollIndicator = NO;
        self.allowsSelection = YES;
        self.allowsSelectionDuringEditing = NO;
        self.sectionHeaderHeight = self.sectionFooterHeight = 22;
        self.alwaysBounceVertical = YES;

        if (_style == UITableViewStylePlain) {
            self.backgroundColor = [UIColor whiteColor];
        }
        // 加入 Layout 标记，进行手动触发布局设置
        [self _setNeedsReload];
    }
    return self;
}
```

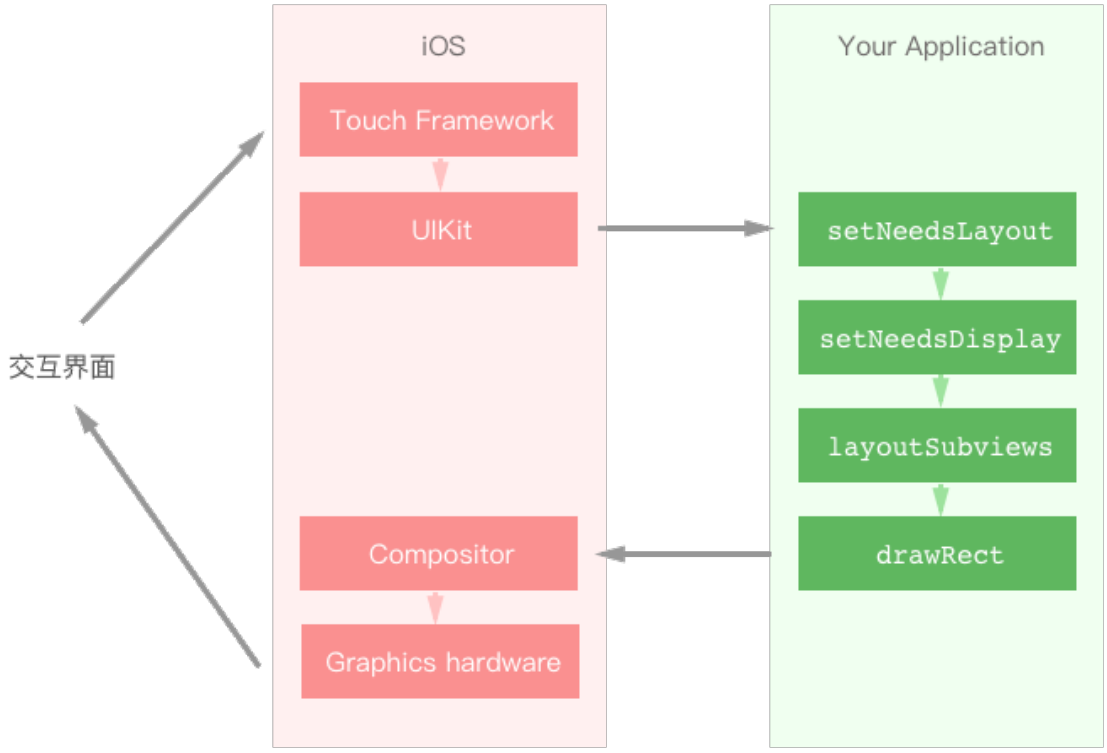
在初始化代码中就看到了重点，`_cachedCells`、`_sections` 和 `_reusableCells` 无疑是复用的核心成员。

代码跟踪

我们先来查看一下 `_setNeedsReload` 方法中做了什么：

```
- (void)_setNeedsReload {
    _needsReload = YES;
    [self setNeedsLayout];
}
```

首先先对 `_needsReload` 进行标记，之后调用了 `setNeedsLayout` 方法。对于 `UIView` 的 `setNeedsLayout` 方法，在调用后 *Runloop* 会在即将到来的周期中来检测 `displayIfNeeded` 标记，如果为 `YES` 则会进行 `drawRect` 视图重绘。作为 Apple *UIKit* 层中的基础 Class，在属性变化后都会进行一次视图重绘的过程。这个属性过程的变化即为对象的初始化加载以及手势交互过程。这也就是官方文档中的 *The Runtime Interaction Model*。



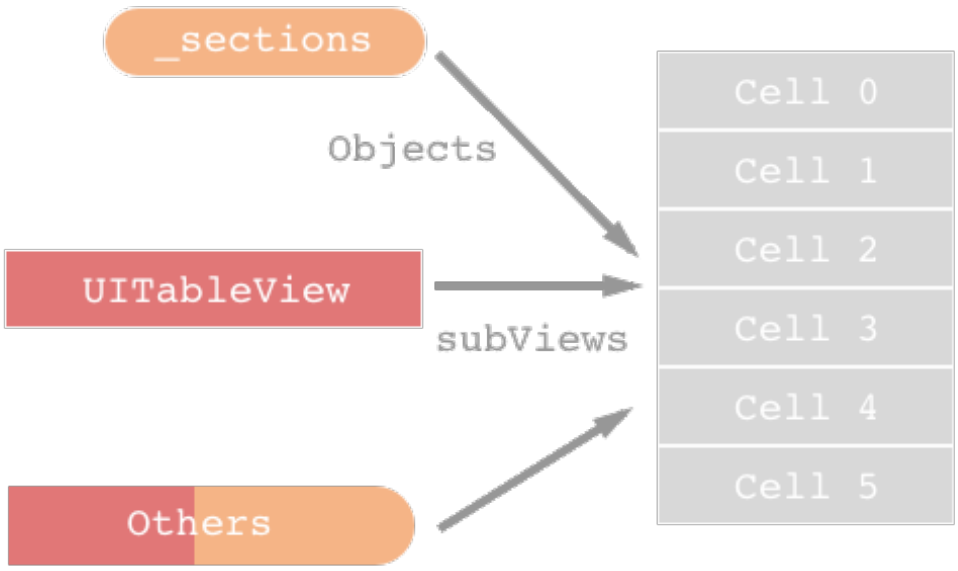
当 *Runloop* 到来时，开始重绘过程即调用 `layoutSubViews` 方法。在 `UITableView` 中这个方法已经被重写过：

```
- (void)layoutSubviews {
    // 会在初始化的末尾手动调用重绘过程
    // 并且 UITableView 是 UIScrollView 的继承，会接受手势
    // 所以在滑动 UITableView 的时候也会调用
    _backgroundView.frame = self.bounds;
    // 根据标记确定是否执行数据更新操作
    [self _reloadDataIfNeeded];
    // 布局入口
    [self _layoutTableView];
    [super layoutSubviews];
}
```

接下来我们开始查看 `_reloadDataIfNeeded` 以及 `reloadData` 方法：

```
- (void)_reloadDataIfNeeded {
    // 查询 _needsReload 标记
    if (_needsReload) {
        [self reloadData];
    }
}

- (void)reloadData {
    // 清除之前的缓存并删除 Cell
    // makeObjectsPerformSelector 方法值都进行调用某个方法
    [[_cachedCells allValues]
makeObjectsPerformSelector:@selector(removeFromSuperview)];
    // 复用 Cell Set 也进行删除操作
    [_reusableCells
makeObjectsPerformSelector:@selector(removeFromSuperview)];
    [_reusableCells removeAllObjects];
    [_cachedCells removeAllObjects];

    // 删除选择的 Cell
    _selectedRow = nil;
    // 删除被高亮的 Cell
    _highlightedRow = nil;

    // 更新缓存中状态
    [self _updateSectionsCache];
    // 设置 Size
    [self _setContentSize];

    _needsReload = NO;
}
```

当 `reloadData` 方法被触发时，`UITableView` 默认为在这个 `UITableView` 中的数据将会全部发生变化。测试之前遗留下的缓存列表以及复用列表全部都丧失了利用性。为了避免出现**悬挂指针**的情况（有可能某个 cell 被其他的视图进行了引用），我们需要对每个 cell 进行 `removeFromSuperview` 处理，这个处理即针对于容器 `UITableView`，又对其他的引用做出保障。然后我们更新当前 tableView 中的两个缓存容器，`_reusableCells` 和 `_cachedCells`，以及其他需要重置的成员属性。



需要解除所有的持有关系

最关键的地方到了，缓存状态的更新方法 `_updateSectionsCache`，其中涉及到数据如何存储、如何复用的操作：

```
- (void)_updateSectionsCache {
    // 使用 dataSource 来创建缓存容器
    // 如果没有 dataSource 则放弃重用操作
    // 在这个逆向工程中并没有对 header 进行缓存操作，但是 Apple 的 UIKit 中一定也做到了
    // 真正的 UIKit 中应该会获取更多的数据进行存储，并实现了 TableView 中所有视图的复用

    // 先移除每个 Section 的 Header 和 Footer 视图
    for (UITableViewSection *previousSectionRecord in _sections) {
        [previousSectionRecord.headerView removeFromSuperview];
        [previousSectionRecord.footerView removeFromSuperview];
    }

    // 清除旧缓存，对容器进行初始化操作
    [_sections removeAllObjects];

    if (_dataSource) {
        // 根据 dataSource 计算高度和偏移量
        const CGFloat defaultRowHeight = _rowHeight ?: _UITableViewDefaultRowHeight;
        // 获取 Section 数目
        const NSInteger numberOfSections = [self numberOfSections];
        for (NSInteger section=0; section<numberOfSections; section++) {
```

```
            const NSInteger numberOfRowsInSection = [self
numberOfRowsInSection:section];

            UITableViewSection *sectionRecord = [[UITableViewSection alloc]
init];
            sectionRecord.headerTitle =
_dataSourceHas.titleForHeaderInSection? [self.dataSource tableView:self
titleForHeaderInSection:section] : nil;
            sectionRecord.footerTitle =
_dataSourceHas.titleForFooterInSection? [self.dataSource tableView:self
titleForFooterInSection:section] : nil;

            sectionRecord.headerHeight =
_delegateHas.heightForHeaderInSection? [self.delegate tableView:self
heightForHeaderInSection:section] : _sectionHeaderHeight;
            sectionRecord.footerHeight =
_delegateHas.heightForFooterInSection ? [self.delegate tableView:self
heightForFooterInSection:section] : _sectionFooterHeight;

            sectionRecord.headerView = (sectionRecord.headerHeight > 0 &&
_delegateHas.viewForHeaderInSection)? [self.delegate tableView:self
viewForHeaderInSection:section] : nil;
            sectionRecord.footerView = (sectionRecord.footerHeight > 0 &&
_delegateHas.viewForFooterInSection)? [self.delegate tableView:self
viewForFooterInSection:section] : nil;

            // 先初始化一个默认的 headerView , 如果没有直接设置 headerView 就直接更
换标题
            if (!sectionRecord.headerView && sectionRecord.headerHeight > 0
&& sectionRecord.headerTitle) {
                sectionRecord.headerView = [UITableViewSectionLabel
sectionLabelWithTitle:sectionRecord.headerTitle];
            }

            // Footer 也做相同的处理
            if (!sectionRecord.footerView && sectionRecord.footerHeight > 0
&& sectionRecord.footerTitle) {
                sectionRecord.footerView = [UITableViewSectionLabel
sectionLabelWithTitle:sectionRecord.footerTitle];
            }

            if (sectionRecord.headerView) {
                [self addSubview:sectionRecord.headerView];
            } else {
                sectionRecord.headerHeight = 0;
            }

            if (sectionRecord.footerView) {
                [self addSubview:sectionRecord.footerView];
```

```
            } else {
                sectionRecord.footerHeight = 0;
            }

            // 为高度数组动态开辟空间
            CGFloat *rowHeights = malloc(numberOfRowsInSection *
sizeof(CGFloat));
            // 初始化总高度
            CGFloat totalRowsHeight = 0;

            for (NSInteger row=0; row<numberOfRowsInSection; row++) {
                // 获取 Cell 高度，未设置则使用默认高度
                const CGFloat rowHeight =
_delegateHas.heightForRowAtIndexPath? [self.delegate tableView:self
heightForRowAtIndexPath:[NSIndexPath indexPathForRow:row inSection:section]]
: defaultRowHeight;
                // 记录高度
                rowHeights[row] = rowHeight;
                // 总高度统计
                totalRowsHeight += rowHeight;
            }

            sectionRecord.rowsHeight = totalRowsHeight;
            [sectionRecord setNumberOfRows:numberOfRowsInSection
withHeights:rowHeights];
            free(rowHeights);

            // 缓存高度记录
            [_sections addObject:sectionRecord];
        }
    }
}
```

我们发现在 `_updateSectionsCache` 更新缓存状态的过程中对 `_sections` 中的数据全部清除。之后缓存了更新后的所有 Section 数据。那么这些数据有什么利用价值呢？继续来看**布局更新**操作。

```
- (void)_layoutTableView {
    // 在需要渲染时放置需要的 Header 和 Cell
    // 缓存所有出现的单元格，并添加至复用容器
    // 之后那些不显示但是已经出现的 Cell 将会被复用

    // 获取容器视图相对于父类视图的尺寸及坐标
    const CGSize boundsSize = self.bounds.size;
    // 获取向下滑动偏移量
    const CGFloat contentOffset = self.contentOffset.y;
    // 获取可视矩形框的尺寸
    const CGRect visibleBounds =
CGRectMake(0,contentOffset,boundsSize.width,boundsSize.height);
    // 表高纪录值
```

```
    CGFloat tableHeight = 0;
    // 如果有 header 则需要额外计算
    if (_tableHeaderView) {
        CGRect tableHeaderFrame = _tableHeaderView.frame;
        tableHeaderFrame.origin = CGPointZero;
        tableHeaderFrame.size.width = boundsSize.width;
        _tableHeaderView.frame = tableHeaderFrame;
        tableHeight += tableHeaderFrame.size.height;
    }

    // availableCell 记录当前正在显示的 Cell
    // 在滑出显示区之后将添加至 _reusableCells
    NSMutableDictionary *availableCells = [_cachedCells mutableCopy];
    const NSInteger numberOfSections = [_sections count];
    [_cachedCells removeAllObjects];

    // 滑动列表，更新当前显示容器
    for (NSInteger section=0; section<numberOfSections; section++) {
        CGRect sectionRect = [self rectForSection:section];
        tableHeight += sectionRect.size.height;
        if (CGRectIntersectsRect(sectionRect, visibleBounds)) {
            const CGRect headerRect = [self rectForHeaderInSection:section];
            const CGRect footerRect = [self rectForFooterInSection:section];
            UITableViewSection *sectionRecord = [_sections
objectAtIndex:section];
            const NSInteger numberOfRows = sectionRecord.numberOfRows;

            if (sectionRecord.headerView) {
                sectionRecord.headerView.frame = headerRect;
            }

            if (sectionRecord.footerView) {
                sectionRecord.footerView.frame = footerRect;
            }

            for (NSInteger row=0; row<numberOfRows; row++) {
                // 构造 indexPath 为代理方法准备
                NSIndexPath *indexPath = [NSIndexPath indexPathForRow:row
inSection:section];
                // 获取第 row 个坐标位置
                CGRect rowRect = [self rectForRowAtIndexPath:indexPath];
                // 判断当前 Cell 是否与显示区域相交
                if (CGRectIntersectsRect(rowRect,visibleBounds) &&
rowRect.size.height > 0) {
                    // 首先查看 availableCells 中是否已经有了当前 Cell 的存储
                    // 如果没有，则请求 tableView 的代理方法获取 Cell
                    UITableViewCell *cell = [availableCells
objectForKey:indexPath] ?: [self.dataSource tableView:self
cellForRowAtIndexPath:indexPath];
```

```
                    // 由于碰撞检测生效，则按照逻辑需要更新 availableCells 字典
                    if (cell) {
                        // 获取到 Cell 后，将其进行缓存操作
                        [_cachedCells setObject:cell forKey:indexPath];
                        [availableCells removeObjectForKey:indexPath];
                        cell.highlighted = [_highlightedRow
isEqual:indexPath];
                        cell.selected = [_selectedRow isEqual:indexPath];
                        cell.frame = rowRect;
                        cell.backgroundColor = self.backgroundColor;
                        [cell _setSeparatorStyle:_separatorStyle
color:_separatorColor];
                        [self addSubview:cell];
                    }
                }
            }
        }
    }

    // 将已经退出屏幕且定义 reuseIdentifier 的 Cell 加入可复用 Cell 容器中
    for (UITableViewCell *cell in [availableCells allValues]) {
        if (cell.reuseIdentifier) {
            [_reusableCells addObject:cell];
        } else {
            [cell removeFromSuperview];
        }
    }

    // 不能复用的 Cell 会直接销毁，可复用的 Cell 会存储在 _reusableCells

    // 确保所有的可用（未出现在屏幕上）的复用单元格在 availableCells 中
    // 这样缓存的目的之一是确保动画的流畅性。在动画的帧上都会对显示部分进行处理，重新计算
可见 Cell。
    // 如果直接删除掉所有未出现在屏幕上的单元格，在视觉上会观察到突然消失的动作
    // 整体动画具有跳跃性而显得不流畅

    // 把在可视区的 Cell（但不在屏幕上）已经被回收为可复用的 Cell 从视图中移除
    NSArray* allCachedCells = [_cachedCells allValues];
    for (UITableViewCell *cell in _reusableCells) {
        if (CGRectIntersectsRect(cell.frame,visibleBounds) && !
[allCachedCells containsObject: cell]) {
            [cell removeFromSuperview];
        }
    }

    if (_tableFooterView) {
        CGRect tableFooterFrame = _tableFooterView.frame;
        tableFooterFrame.origin = CGPointMake(0,tableHeight);
        tableFooterFrame.size.width = boundsSize.width;
```

```
        _tableFooterView.frame = tableFooterFrame;
    }
}
```

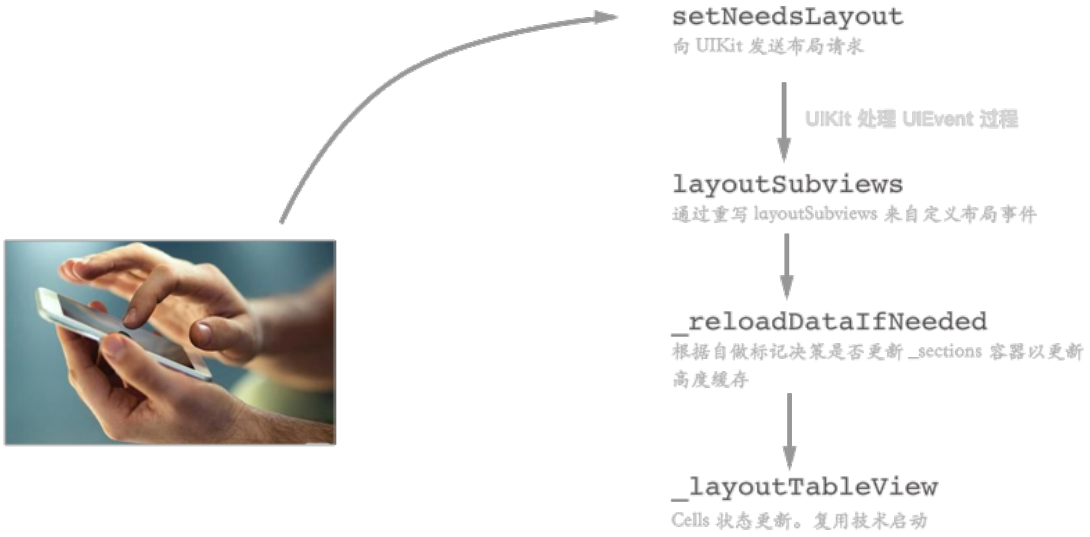
`CGRectIntersectsRect` 方法用于检测两个 Rect 的碰撞情况。如下图所示：

如果你已经对 `UITableView` 的缓存机制有所了解，那么你在阅读完代码之后会对其有更深刻的认识。如果看完代码还是一头雾水，那么请继续看下面的分析。

Cell 复用场景三个阶段

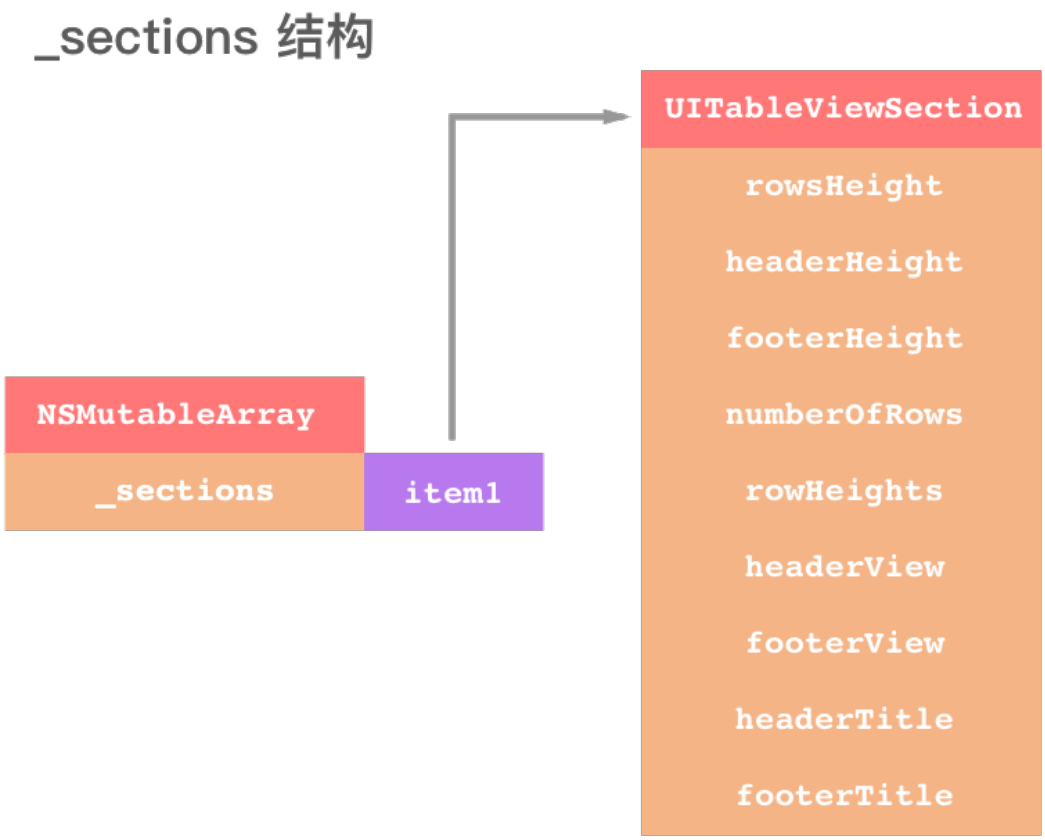
布局方法触发阶段

在用户触摸屏幕后，硬件报告触摸时间传递至 `UIKit` 框架，之后 `UIKit` 将触摸事件打包成 `UIEvent` 对象，分发至指定视图。这时候其视图就会做出相应，并调用 `setNeedsLayout` 方法告诉视图及其子视图需要进行布局更新。此时，`setNeedsLayout` 被调用，也就变为 Cell 复用场景的入口。



缓存 Cell 高度信息阶段

当视图加载后，由 `UIKit` 调用布局方法 `layoutSubviews` 从而进入**缓存 Cell 高度阶段** `_updateSectionsCache`。在这个阶段，通过代理方法 `heightForRowAtIndexPath:` 获取每一个 Cell 的高度，并将高度信息缓存起来。这其中的高度信息由 `UITableViewSection` 的一个实例 `sectionRecord` 进行存储，其中以 section 为单位，存储每个 section 中各个 Cell 的高度、Cell 的数量、以及 section 的总高度、footer 和 header 高度这些信息。这一部分的信息采集是为了在 Cell 复用的核心部分，Cell 的 Rect 尺寸与 tableView 尺寸计算边界情况建立数据基础。



复用 Cell 的核心处理阶段

我们要关注三个存储容器的变化情况：

- `NSMutableDictionary` 类型 **_cachedCells**：用来存储当前屏幕上所有 Cell 与其对应的 indexPath。以键值对的关系进行存储。
- `NSMutableDictionary` 类型 **availableCells**：当列表发生滑动的时候，部分 Cell 从屏幕移出，这个容器会对 `_cachedCells` 进行拷贝，然后将屏幕上此时的 Cell 全部去除。即最终取出所有退出屏幕的 Cell。
- `NSMutableSet` 类型 **_reusableCells**：用来收集曾经出现过此时未出现在屏幕上的 Cell。当再出滑入主屏幕时，则直接使用其中的对象根据 `CGRectIntersectsRect` Rect 碰撞试验进行复用。

在整个核心复用阶段，这三个容器都充当着很重要的角色。我们给出以下的场景实例，例如下图的一个场景，图 ① 为页面刚刚载入的阶段，图 ② 为用户向下滑动一个单元格时的状态：

当到状态 ② 的时候，我们发现 `_reusableCells` 容器中，已经出现了状态 ① 中已经退出屏幕的 Cell 0。而当我们重新将 Cell 0 滑入界面的时候，在系统 `addView` 渲染阶段，会直接将 `_reusableCells` 中的 Cell 0 立即取出进行渲染，从而代替创建新的实例再进行渲染，简化了时间与性能上的开销。

UITableView 的其他细节优化

复用容器数据类型 `NSMutableSet`

在三个重要的容器中，只有 `_reusableCells` 使用了 `NSMutableSet`。这是因为我们在每一次对于 `_cachedCells` 中的 Cell 进行遍历并在屏幕上渲染时，都需要在 `_reusableCells` 进行一次扫描。而且当一个页面反复的上下滑动时，`_reusableCells` 的检索复杂度是相当庞大的。为了确保这一情况下滑动的流畅性，Apple 在设计时不得不将检索复杂度最小化。并且这个复杂度要是非抖动的，不能给体验造成太大的不稳定性。

在 C++ 的 STL 标准库中也有 `multiset` 数据类型，其中实现的方法是通过构建**红黑树**来实现。因为红黑树具有高效检索的性质，这也是 `set` 的一个普遍特点。也许是 `NSMutableSet` 是 *Foundation* 框架的数据结构，构造其主要目的是为了更快的检索。所以 `NSMutableSet` 的实现并没有使用红黑树，而是暴力的使用 **Hash 表**实现。从 *Core Foundation* 中的 CFSet.c 可以清晰的看见其底层实现。在很久之前的 Cocoa Dev 的提问帖中也能发现答案。

高度缓存容器 _sections

在每次布局方法触发阶段，由于 Cell 的状态发生了变化。在对 Cell 复用容器的修改之前，首先要做的一件事是以 Section 为单位对所有的 Cell 进行缓存高度。从这里可以看出 `UITableView` 设计师的细节。 Cell 的高度在 `UITableView` 中充当着十分重要的角色，一下列表是需要使用高度的方法：

- `- (CGFloat)_offsetForSection:(NSInteger)index`：计算指定 Cell 的滑动偏移量。

- `- (CGRect)rectForSection:(NSInteger)section`：返回某个 Section 的整体 Rect。
- `- (CGRect)rectForHeaderInSection:(NSInteger)section`：返回某个 Header 的 Rect。
- `- (CGRect)rectForFooterInSection:(NSInteger)section`：返回某个 Footer 的 Rect。
- `- (CGRect)rectForRowAtIndexPath:(NSIndexPath *)indexPath`：返回某个 Cell 的 Rect。
- `- (NSArray *)indexPathsForRowsInRect:(CGRect)rect`：返回 Rect 列表。
- `- (void)_setContentSize`：根据高度计算 `UITableView` 中实际内容的 Size。

一次有拓展性的源码研究

在阅读完 Chameleon 工程中的 `UITableView` 源码，进一步可以去查看 `FDTemplateLayoutCell` 的优化方案。Apple 的工程师对于细节的处理和方案值得各位开发者细细寻味。多探求、多阅读以写出更优雅的代码。😄