声明：本文翻译自《iOS performance optimization》，原文作者 Khang Vo。翻译本文纯属为了技术交流的目的，并不具有任何的商业性质，也不得利用本文内容进行商业盈利。欢迎转载，但是希望转载的时候加上出处连接，谢谢。译者联系方式 setipro@163.com，如果有 iOS 开发之类的问题，欢迎一起讨论，谢谢。另，由于本人翻译经验不多，如果翻译不妥或者理解不到位的地方，希望各位朋友海涵，可以发信到上述邮箱，我会及时地根据大家的反馈，对翻译稿做及时地修改，谢谢！

第二个例子

当你在一台老式设备上运行一个含有 TableView 的应用，而每个 Cell 上又由很多的子视图(subView)组成的时候，对 Cell 的绘制代码进行定制化将有助于性能的提升。对于 iPhone4 及其以前的设备，这个性能优化技巧做带来的效果是显著的。

在这个例子中，我将把应用程序中的 Cell 变得更加复杂，每个 Cell 含有的子视图数量达到了十个之多，包括各种图片，文字等等。因此，你会看到在某些真正应用程序（比如我们要模拟的 Facebook 应用）的滑动性能将会被 Cell 复杂的子视图结构严重影响。我要测试的应用程序界面如图 3-5 所示。

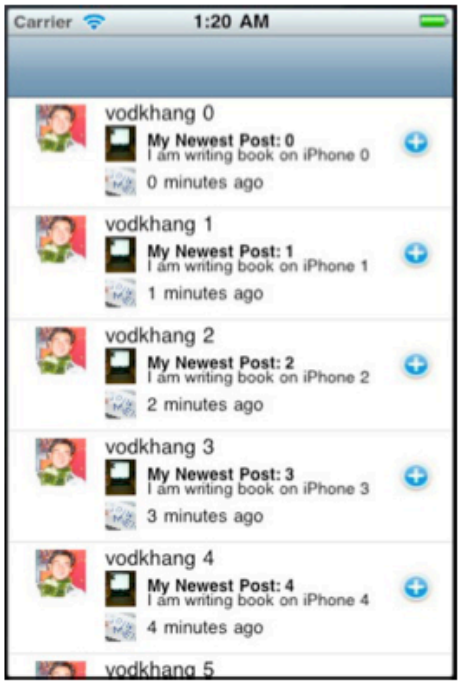

Figure 3–5. *The second example application*

每一个 Cell 都由一个用户头像图片，一个用户名，还有一个附带图片、标题和内容的状态构成。测试结果如表 3-5 所示。

Table 3–5. *Benchmarks Results After Reusing Images*

Test Number	Preparation Time (Seconds)	General Time (Frames per second)
1	0.004199	44
2	0.004388	52
3	0.004237	47
4	0.004245	54
5	0.004197	51
6	0.004232	43

表 3-6 显示了对 Cell 的绘制代码进行定制化之后的测试结果

Table 3–6. *Benchmark Results with Custom Drawing Code*

Test Number	Preparation Time (Seconds)	General Time (Frames per second)
1	0.004652	53
2	0.004764	53
3	0.004568	43
4	0.005015	54
5	0.004743	56
6	0.004743	60

从表 3-5 和表 3-6 的数据对比可以看出，个性化的渲染代码对渲染时间有了一个显著地提升。对一个有着复杂子视图结构的应用而言，这种性能已经足够好到不用再进行什么优化了。

Figure 3–6. *The cell*

图 3-6 中的 TableViewCell 有四个图片，外加一个不同背景颜色的子视图。当你想在已有视图中简单快速的创建一个拥有不同背景颜色的视图组件的时候，子视

图通常是个不错的选择。但是这种方法在滚动视图的时候可能会导致一些性能问题，因此你需要想办法避免这种解决方案。

现在我们来看看新方案的源码，在新方案中，我将对 Cell 进行个性化绘制，而不是使用子视图的方案。马上你会看到我是如何实现这种技术的，然后我会总结不同技术方案的优缺点。样例代码源自工程 DrawingCellViewController。下面是主要的代码段。

For UITableViewController:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"CellIdentifier";

    CustomDrawingTableViewCell *cell = (CustomDrawingTableViewCell *) [self.tableView

    dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {

        cell = [[CustomDrawingTableViewCell alloc]

        initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];

    }

    [cell updateMyCell];

    return cell;

    }
```

好，如你所见，`UITableViewController` 的主要代码并没有太大的改动。这段代码和标准的 `UITableViewCell` 代码的主要区别在于你初始化 `Cell` 的方式。

```
[[CustomDrawingTableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
```

对比另外一段代码

```
[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
```

在定制化的 `UITableViewCell（CustomDrawingTableViewCell）`中

```
 (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString

*)reuseIdentifier {

    if (self = [super initWithStyle:UITableViewCellStyleDefault reuseIdentifier:reuseIdentifier]) {

        CGRect subFrame = CGRectMake(0.0, 0.0,

        self.contentView.bounds.size.width, self.contentView.bounds.size.height);

        drawingView = [[CustomDrawingView alloc] initWithFrame: subFrame];

        drawingView.autoresizingMask = UIViewAutoresizingFlexibleWidth |

        UIViewAutoresizingFlexibleHeight;

        [self.contentView addSubview:drawingView];

       return self;
```

}

接下来是最重要的部分：如何在视图内绘制文本，图像和控件。

CustomDrawingView.m

```
- (void)drawRect:(CGRect)rect {
    self.backgroundColor = [UIColor whiteColor];
    // Drawing code.
    [self.userName drawInRect:CGRectMake(70,0, 95, 21) withFont:userNameFont
    lineBreakMode:UILineBreakModeTailTruncation alignment:UIBaselineAdjustmentAlignBaselines];
    // Drawing Image
    [self.avatarImage drawInRect:CGRectMake(20, 5, 36, 34)];
    // Drawing button
    [self.button drawInRect:CGRectMake(50, 5, 36, 34)];
 }
```

简而言之，在 UITableViewController 中构造一个定制化 UITableViewCell 的方法和之前是很相似的；你只需要判断从队列中弹出的元素是否为空，如果是空的话就构造一个新的元素。在新元素的初始化方法中，你必须在 Cell 中添加一个子视图。在这个子视图中，你需要覆盖 drawRect 方法，然后使用 drawInRect 方法来绘制文本或者图像。

个性化的绘制代码之所以能比加载 nib 文件或者直接添加子视图这两种方式性能更优越的原因就在于 GPU(图形处理器)将会负责运行个性化的绘制代码。GPU 渲染和显示 UI 的速度极快；因此，个性化的绘制代码只绘制具有复杂子视图结构的最佳方案。

[**注意**：谨记，要把 CustomDrawingView 的背景颜色设置为白色。默认背景颜色是黑色。]

您能从这些例子中学到什么？

从前面列举的两个例子来看，你需要记住一些基本知识。

（1）使用 ReuseIdentifier。这将提升你的程序性能。

（2）尝试着去减少 Cell 的准备时间，尤其是从网络或者文件加载图片素材的时候。这可以在最短的时间内来展示图片。

（3）如果你的应用程序的 Cell 使用了大量的子视图结构，那么就需要考虑自己通过代码去手动绘制 Cell。这将会让 GPU 来对整个过程进行提速。

[**警告**：测试结果如你所见，fps 的数据变得更好了，已经接近了最理想的 60。但是，如果采用这种方法，你就不能利用 InterfaceBuilder 来构建你的 UI 了。你需要去计算绘制的位置，并且将信息更新放到 drawRect 中去。很快，你就会

发现维护程序变得非常困难。因此，谨慎使用 drawRect 方法，并且避免过度优化。]

其他的相关技术

就对 TableView 的滚动性能就行优化这个话题，我们已经讨论了一些重要的技术。还有一些你不经常用到的小技巧，但是我也会这在里介绍一下它们。如果你能理解这些概念，你就在其他的例子能使用这些技术。

缓存高度

因为 TableView 在创建一个新的 Cell 的时候需要知道 Cell 的高度信息，所以你要对这些信息进行缓存。如果这些 Cell 的高度是固定的，那么你就无需对这类事情担心。但是，如果那些高度不是固定的，那么你就需要确保 Cell 高度的计算速度足够的快。

尝试如下代码：

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath
 {
     return 80;
 }
```

尽量避免如下的代码:

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
for (int i = 0; i < 100; i++) {
// find the smallest possible height for the row
}
return smallestHeight; }
```

当操作系统需要渲染 Cell 或者在动画中需要对 Cell 进行编辑或者重排序的话，系统就会对第一段代码进行多次调用。但是如果使用第二段代码的话，每当操作系统需要了解 Cell 的高度，就需要进行将近一百次的计算。

透明度

如果可能的话，尽量让 UITableViewCell 的所有子视图和图层都设置为不透明状态。当一个视图是透明的状态时，iOS 就需要在一个像素点上绘制两次或者更多次，因为那个点同时属于多个子视图。绘制的过程是比较耗费时间的。

通过 InterfaceBuilder 可以简单的进行这种设置。开发者应当多检查检查相关设置，以确保所有的子视图都是不透明的，图 3-7 显示了如果将 Cell 的子视图设置为不透明。

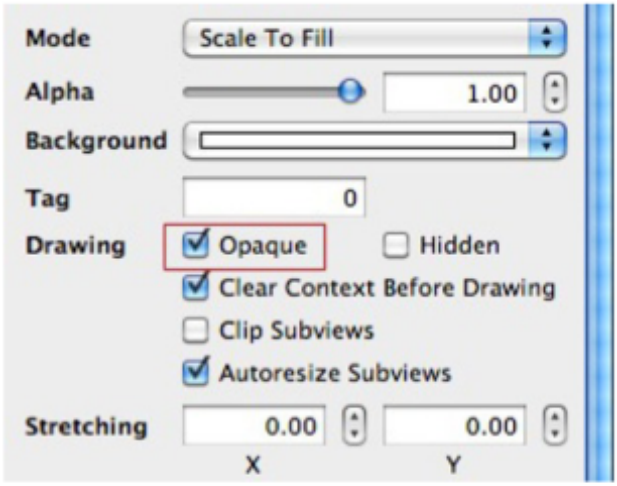

Figure 3–7. *Set opaque for the subviews*

对相应的代码来说，我们也可以通过代码来进行相关的设置，设置方法如下：

```
view.opaque = YES;
```

避免使用图形特效

尽量在 UIImage 中避免使用类似渐变效果这种特效。在对 CoreAnimation 进行相关的设置后，对你的应用检查特效和相关的问题，如图 3-8 和 3-9 所示。

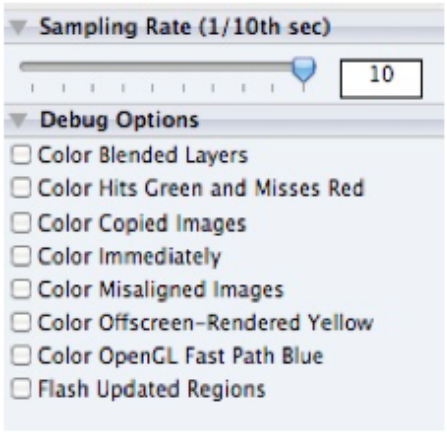

Figure 3–8. *CoreAnimation options to set up debug options*

Figure 3–9. CoreAnimation Instrument with color blended

编辑/重排的性能

在前面的一些章节中，我为你展示了如何直接进行绘制，通过这种方法，你可以对你的应用性能进行大幅度的优化。然后，这种绘制的方法在动画以及重排的性能方面会造成一些问题。

当你使用子视图技术的时候，动画会变得更快，并且 UIKit 不需要在动画的过程中进行重绘工作。因此，在使用子视图技术而非直接绘制技术的时候，相关处理速度会变得更快速。如果你在需要有动画的时候使用直接用代码绘制的方式的话，你需要再次对相关的视图进行二次绘制来适配新的视图。这对编写以及维护代码来讲，会让相关的工作变得更为繁杂。

因此，在对 UITableView 进行优化的时候，需要仔细地权衡。在没有太多的子视图结构，或者需要有相关的动画效果的时候，我还是推荐使用子视图技术。这么做之后，虽然可能程序会变得稍慢，但是相关性能也足够好了。

本章总结

通过学习一些例子以及研究相关代码，你已经学到了一些性能优化技巧。

（1）使用 NSLog 和 CoreAnimation 进行测试。通过一个真实的例子，我让你了解了如何使用这两种工具来有效地对相关指标进行测试，以能够迅速地了解到问题本质，了解我们在每一步的优化之后都取得了哪些进展。

（2）适当地复用 Cell。这是相关性能优化的第一步，也是最重要的移步。这很容易实现，但是很多应用并没有这么做。因此，如果你有相关的性能问题，请多检查一下相关的部分是否已经做了这种优化。

（3）正确地缓存、复用图像和数据。另外一个重要的优化步骤就是，在返回或者显示一个 Cell 的时候，减少加载数据和图像的时间。

（4）减少逻辑计算时间。并不是只有 I/O 过程才会减慢或者阻塞 UI 线程；任何一种数据处理都有可能会有这种效果。因此，你需要尽量减少这一类数据处理。

（5）设置为不透明。这个小问题通常发生在开发者在视图中添加元素的时候。如果他们没有把每个视图都设置为非透明状态的话，那么渲染的时候就要对同一个点进行多次渲染。

（6）对高度进行缓存。这是开发者通常犯的另外一个小错误。每当需要一个新的 Cell 的时候，有两个主要方法要被调用。

（7）避免使用图形特效。在 Cell 上，有越多的图形特效，那么渲染的过程就会越缓慢。所以，你也应该对这点进行相关的测试。你应该使用 CoreAnimation 来检查每个 UI 组件的渲染情况。

（8）编辑/重排的性能。通过使用代码绘制的技术来对滚动性能进行优化这种方式会在对 Cell 进行编辑或者有 Cell 相关的动画的时候造成一些问题。因此 UIKit 和动画框架已经对子视图进行相关的优化了。如果你的 Cell 是自己绘制的，那么这些对子视图的相关优化就会失效。

练习

A.理论

1.创建一个检查清单，以保证你在执行必要的，基本的优化步骤的时候，能让 UITableViewCell 的相关性能变得更好。

B.实践

1.写一个小应用，来查看在有编辑操作和动画的时候，drawRect 是如何运行的。

2.实践在"复用图像"这一小节提到的练习。尝试着使用多线程技术来从文件中加载图像。

3.尝试分别使用以下三种方式来创建一个定制化的 UITableViewCell：

（1）InterfaceBuilder

(2) 通过子视图技术（Subview）的相关代码

(3) 调用绘制代码