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！

第三章（上） 对 UITableview 的性能进行优化

在本章，你将逐步完成如下任务：
1. 在一个真正的例子中去运用你在第二章学到的工具去进行测试。
2. 在例子中，逐步地对其滚动性能进行优化
3. 使用如下技术对 UITableView 的性能进行优化
（1）用一些基本的优化技巧对 UITableView 的 Cell 性能进行简单地优化
（2）在 Cell 中直接绘制视图
（3）对一些需要展示编辑或者重排序动画的 Cell 进行优化
（4）还有一些其他的基本开发技巧

iPhone 应用通常以列表的格式来展示数据。Apple 为初级开发者提供了非常好用的工具:UITableView 和 UITableViewCell。如果开发者只是想在 Cell 中的左边展示一张小图片，在 Cell 的中间显示一些文本信息，那么这个默认的苹果控件会运行的很好。但是！当开发者想在 Cell 上做更多的定制化工作，比如在 Cell 中展示两到三张图片，或者在不同的位置添加文本信息的时候，那么你的麻烦就来了。你迟早会陷入由于滚动速度不稳定而带来的性能问题，这点在一些老设备，比如 iPhone 3G 上尤为明显。

示例介绍

在这个练习中，我会对两个方面进行性能测试：
（1）从 TableView 队列中弹出一个 Cell 或者直接创建一个 Cell 的速度，或者将 Cell 返回给操作系统的速度
（2）操作系统对 Cell 进行渲染以及展示的速度

第一个指标可以简单地通过使用 NSLog 来进行测试;而第二项指标比较复杂，只能通过 CoreAnimation 这个工具进行测试。为了进一步说明，我将举两个不同的例子来进行讲解。第一个例子的 Cell 只包含一个图片和一个文本信息块；另外一个 TableView 的 Cell 则包含多复杂的子视图。通过这两个例子，你将会对 UITableView 不同的优化方法有一个了解。

在本章的结尾，我将会列出一些其他重要的知识。由于时间所限，我没有对这些知识在本章进行深入地探讨。这些知识所相关的错误并不常见，但是如果开发者过于粗心以至于犯下了其中的一个错误，那么他可能会为调试，找出问题的所在而花上一整天的时间。我希望能让您有足够的知识和技能来应对各种情况。有时，性能优化的工作可以如此的简单，以至于你只需要在你的代码中做些微小的改动。但是，在某些情况，比如我们在第二个例子中所看到的，你需要对整个模块的代码进行重写才能改善应用的性能。我希望在我对两个例子进行讲解之后，您能能对程序的框架有一个清晰的设计思路，然后在一开始就能进行正确的决策，而不用再在后来对相关代码模块进行重写。

重温相关测试工具

在本章，你将使用 CoreAnimation 来对 iPhone OS 的渲染性能进行测试。这将帮你判断问题是出在显示方面的处理还是计算方面的处理上。鉴于第二章已经对这个工具有所介绍，因此本章只做一个简单的回顾。

图 3-1 显示了 CoreAnimation 工具在运行应用时需要进行观测的三个部分。图 3-2 显示了相关性能指数。

Figure 3–1. *Main parts in the CoreAnimation tool*

Table

	Frames Per Second
0	0
1	27
2	0
3	0
4	4
5	13
6	15
7	24
8	32
9	49
10	33

Figure 3–2. *Recent display performance*

第一个例子

第一个例子将会为你展示对一个逐步对 UITableView 进行性能调优的过程。最开始的相关代码包含了我从许多开发者那里收集到的很多性能方面的错误。在这个优化过程中，你将发现在每进行一步优化之后，性能都会有所提升。

第一个例子的介绍

如图 3-3 所示，一个十分常见的需求，开发一个 UITableView，每一个 Cell 中有一个图片和一个文本块。我会带着您浏览一下本示例的相关代码。让我们看一下一些常见的应用，比如 Facebook 的应用；应用需要展示一个用户的图片，还有一个用户分享的图片。这个应用还需要在 Cell 中展示其他的小图标。在第一个测试中，请将工程名取为 SlowPerformanceTableView。

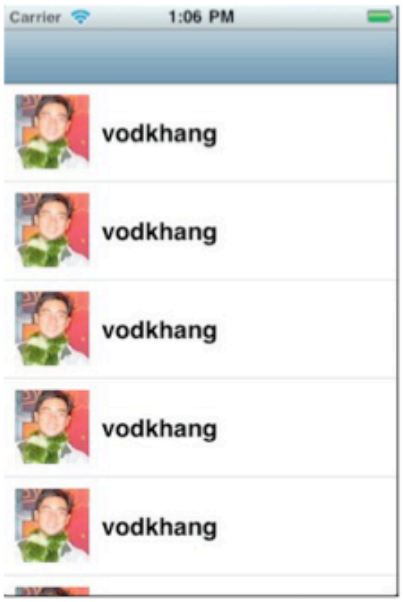

Figure 3–3. *The application for the first example*

测试标准

在开始工作之前，你需要了解你的最终目标；在本例中，最终目标就是在运行这个应用的时候，用户能在进行滚动操作的时候得到良好的用户体验。所以，运行一个普通的，没有进行定制化的 UITableViewCell(Cell 中会对图片进行简单的加载)，得到如 3-1 表所示结果。

Table 3–1. *Results from Running the Test on the Example*

Test Number	Preparation Time (Seconds)	General Time (Frames per second)
1	0.000200	55
2	0.000209	56
3	0.000200	55
4	0.000226	54
5	0.000217	60
6	0.000376	59

使用 CoreAnimation 对每秒渲染的帧数进行测试，最理想的数据是 60fps(数字越高，意味着更好的性能)。对一个标准的 UITableViewCell 来讲，通常的渲染速度是 55-60fps；这就是我们的优化目标之一。另外一个目标就是确保 Cell 的准备时间足够的短。减少 Cell 的准备时间是相对简单的，因此我会在第一个例子中首先专注于此。

初始测试数据

在第一个例子中，我对一个具有 6 个 Cell 的例子进行了最初的基准此时，并随机得到了一些测试结果。通过运用 NSLog 和 CoreAnimation 工具，我们得到的测试结果如表 3-2 所示。

Table 3–2. *Results from Initial Benchmark on the First Example*

Test Number	Preparation Time (Seconds)	General Time (Frames per second)
1	0.013282	38
2	0.012456	31
3	0.013496	43
4	0.013560	37
5	0.013815	50
6	0.013090	20

所以，从测试结果中可以看出，每准备一个 Cell 需要耗费 10 毫秒左右的时间。因为上述巨大的延时，主要的测试指标（fps）大幅度地下降了。因此，我最初的目标就是减少 Cell 的准备时间。
在使用[UIImage imageNamed:name]和[[UIImage alloc] initWithContentsOfFile:name]之间，有一些需要注意的地方。我会在稍后解释他们之间的区别，以及你需要用 imageNamed 来代替 initWithContentsOfFile 的原因。

复用 UITableViewCell

对 UITableView 进行优化通常是比较简单的；你需要做的只是核实你是否运用正确的方式来复用 UITableViewCell。对于 iOS 来说，创建一个新的 UITableViewCell 是比较耗费 CPU 的资源的。因此，如果每当用户对 UITableView 进行上下滚动，CPU 就要不停地创建新的 Cell 的话，那么整体的性能就会下降。Apple 对这个问题的标准解决方法就是当 Cell 移出屏幕后对其进行复用。

标准的 TableView Cell

对于一个标准的 UITableViewCell 而言，标准的代码会运行的很好，并且具有高速的滚动性能。

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

static NSString *CellIdentifier = @"Cell";

UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier]; if (cell == nil) {

cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault

reuseIdentifier:CellIdentifier]; }
```

注意以下两段代码的使用：

reuseIdentifier:CellIdentifier and dequeueReusableCellWithIdentifier:CellIdentifier.

这两部分代码会帮助您正确地复用 UITableViewCell。有两种方式来创建一个新的定制化的 Cell，通过使用 InterfaceBuilder 或者直接通过调用 addSubview: 方法来以代码的方式进行创建。

通过 Interface Builder 对 TableViewCell 进行定制

在使用 InterFaciBuilder 的过程中，开发者通常忘记对 identifier 属性进行设置，而设置是很简单的。打开 xib 文件，找到第一栏，对第一行进行更改即可，如图 3-4 所示。

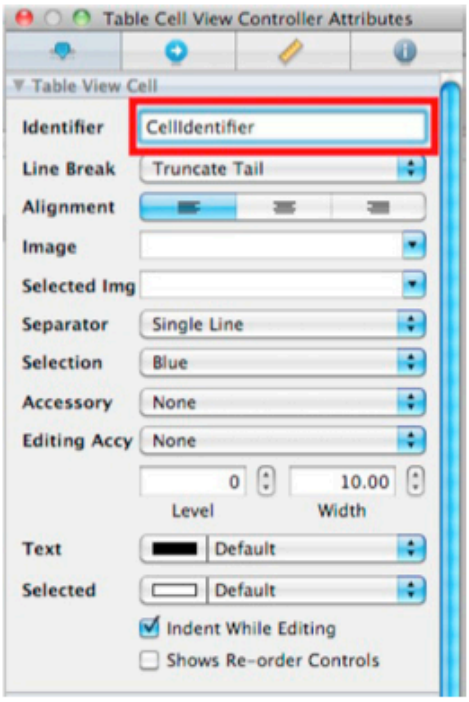

Figure 3–4. *Setting the Reuse Identifier for the Cell*

现在，在 cell 的初始化的代码中，你需要使用完全一样的 identifier。

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

static NSString *CellIdentifier = @"CellIdentifier"; // must match the one in InterfaceBuilder

UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier]; if (cell == nil)

{

cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];

}

通过代码来定制 TableView

如果你没有使用 IB 来创建定制的 Cell，那么你可以在你的定制类 ReuseTableViewCell 中返回它。

TableCellViewController.h

@interface TableCellViewController : UITableViewCell { }

@end

TableCellViewController.m #import "TableCellViewController.h"

@implementation TableCellViewController

- (NSString *)reuseIdentifier { return @"CellIdentifier";

} @end

[**注意**：上述两种不同的方法主要区别于你加载和初始化 UITableView 的方式不同。为了确保你能理解和区分这两个例子，我将为你展示一些主要的代码段。]

从 Nib 文件中加载 Cell

首先，你需要能把 nib 文件从文件系统加载到内存，并且将其从 UITableView 中解析出来的代码。

```
- (UITableViewCell *)cellWithTableView:(UITableView *)tableView cellIdentifier:(NSString *)cellIdentifier
nibName:(NSString *)nibName {
UITableViewCell *textCell = [tableView dequeueReusableCellWithIdentifier:cellIdentifier];
if (textCell == nil) {
NSArray *topLevelObjects = [[NSBundle mainBundle] loadNibNamed:nibName
owner:nil options:nil];
for (id currentObject in topLevelObjects) {
if ([currentObject isKindOfClass:[UITableViewCell class]]) {
textCell = (UITableViewCell *)currentObject;
break; }
} }
return textCell; }
```

你可以调用 cellWithTableView:cellIdentifier:nibName:这个方法，来从 TableViewController 的 nib 文件中加载 TableView。相关代码如下所示:

```
ReuseTableViewCell *cell = (ReuseTableViewCell *) [self getCellWithTableView:tableView
cellIdentifier:CellIdentifier nibName:@"ReuseTableViewCell"];
```

这段代码并不是完美的;你需要自己对相关的 nib 文件来进行修改以确保程序能够良好的运行。然而，我的目的是为了让你了解两种方法的不同，因此我不会涉及更多的相关细节。

以代码的方式加载 Cell

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdentifier {
self = [super initWithStyle:style reuseIdentifier:reuseIdentifier]; if (self != nil) {
UIImageView *imageView = [[UIImageView alloc] initWithFrame:CGRectMake(20, 20, 30, 30)];
[self.contentView addSubview:imageView]; }
 return self; }
```

一个小练习：你需要自己创建一个新的工程，并且创建一些定制化的 Cell，然后你应该对各个相关细节进行检查，以确保你正确地复用了 Cell。

再次运行基准测试

在复用了 Cell 之后，你可以对滚动性能再次进行基准测试。如你在表 3-3 所看到的，在你正确地复用 Cell 之后，滚动性能有了成倍的提升。

Table 3–3. *Results from Benchmark After Reusing the Cell*

Test Number	Preparation Time (Seconds)	General Time (Frames per second)
1	0.007310	40
2	0.006888	35
3	0.006996	48
4	0.006934	42
5	0.006902	55
6	0.006863	25

结果显示您的思路是正确的；然后，当前的性能还是不够好。您通常需要让准备的时间达到 0.0006-0.0001 这个区间;这也是一个标准的 UITableViewCell 所应有的性能指数，正如我在第一节所提到的一样。因此，在接下来我将教您如何去复用图片，而不是每次需要图片的时候都去创建一个新的。

这就是我们为什么要复用 cell 的原因。操作系统创建一个新的 Cell 到内存即消耗时间，有消耗内存。这也是为什么 tableview 将每一个滑出屏幕的 cell 都进行队列缓存的原因。如果你复用了 Cell，操作系统就没有必要在创建一个新的出来；它只需要取出以往的 Cell，对其属性进行一些修改，然后再次显示那个 Cell 即可。整个过程要比重新创建一个 Cell 要快上很多。

图像复用

在图像显示方面，通常的问题就出来加载时间上，无论是通过文件读写的方式还是通过网络交互的方式，加载时间通常都会很长。当 iOS 不能返回 Cell 来对 UI 进行渲染的时候，这些加载过程就会影响到用户的滑动体验。

在本节，请参考名为 ReuseImageViewController 的工程。我先解释一下，在这些工程中我为什么没有用[UIImage imageNamed:@""]方法。imageNamed 方法做了一项很重要的工作：它将所加载的图像在内存中进行了缓存，当你再次调用这个图像的时候，就能直接从内存进行复用了。这个方法的问题就在于它只能从你的 Bundle 中加载图像--换而言之，它只能加载那些在源代码中就有的图像。你不能通过这个方法来从网络来加载图像。通常，你必须使用[[UIImage alloc]

initWithContentsOfFile:@""]或者[[UIImage alloc] initWithData:Data]这两个方法。但是在使用这两个方法的时候，操作系统通常不会在内存中自动地进行缓存操作。

因此，我希望你能通过在内存中创建一个小的 dictionary 来对图像进行缓存操作（请参见第四章）。另外一个处理文件加载问题的途径就是使用多线程（请参见第六章）。通过使用这些技术，你可以将繁重的处理工作从主线程中分离出来。在当前例子中，我不会使用多线程，因为那有可能让你在同时遇到一大堆的新概念。在本章的结尾，你可以自己来进行实践练习。

下面是在 NSDictionary 中储存文件的主要代码段（请不要在你的应用中这么做，因为这将导致内存警告）

```
// Code to store the image in the dictionary - (UIImage *)imageWithName:(NSString *)name {

if ([self.imageDictionary objectForKey:name]) { return [self.imageDictionary objectForKey:name];

}

UIImage *image = [[UIImage alloc] initWithContentsOfFile:name]; [self.imageDictionary setObject:image

forKey:name];

return image;

}
```

接下来是提取图像数据的代码

```
// Customize the appearance of table view cells.

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

static NSString *CellIdentifier = @"CellIdentifier"; ReuseTableViewCell *cell = (ReuseTableViewCell *) [self

getCellWithTableView:tableView cellIdentifier:CellIdentifier nibName:@"ReuseTableViewCell"];

NSString *avatarFile = [NSString stringWithFormat:@"a0"];

NSString *avatarName = [[NSBundle mainBundle] pathForResource:avatarFile ofType:@"jpeg"];

cell.avatar.image = [self imageWithName:avatarName];

cell.userName.text = [NSString stringWithFormat:@"hi here: %d", indexPath.row];

// Configure the cell. return cell;

}
```

在对代码进行更新后，你可以再次进行基准测试。正如你在表 3-4 可以看到的，测试结果变得更好了。平均的运行时间现在降到了 0.002 秒，而渲染性能指标则几乎提升到 60fps 了。相比于之前的 ReuseTableViewCell，程序有了更好地性能表现。

Table 3–4. *Results from Benchmark After Reusing Images*

Test Number	Preparation Time (Seconds)	General Time (Frames per second)
1	0.002314	54
2	0.002233	59
3	0.002313	55
4	0.002307	49
5	0.002323	60
6	0.002307	55

漂亮！现在 fps 数据已经接近 60 了，而同时，准备时间指数也变得更好了。如果你的应用达到了这种水平，你就不用再为滑动方面的性能问题担心了。通常来讲，这种性能对于一个包含多个子视图的 Cell 已经足够了。这很好，因为你不需要在一开始的时候做很多的工作。但是，如果在进行了这些优化工作之后，你的应用滑动性能还是有问题，你就需要做更多的复杂工作来得到更好的性能了。

正如第一章和第二章所言，你需要注意避免过度优化。为了一些小的性能提升而做过多的优化工作是不值得的。因此，如果现在你的应用还是在滑动性能方面有问题的话，那么就需要看看第二个例子了，在第二个例子中，我们将使用 UITableView 的绘制技术。

减少准备的时间

通常，我会通过缓存图像的方式来达到复用图像的目的，并且，我想减少初始化的过程。当操作系统需要为 TableView 渲染一个 Cell 的时候，系统通常会调用如下方法

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

// Initialize and return the Cell here

}
```

因此，如果你在此等待了过长的时间，那么 UI 的渲染过程就会被阻塞；应用既不能显示新的内容，也不能做其他的工作。这就是为什么用户会看到滚动永远的停滞在一个地方的原因之一。

为了让这个过程尽可能的快，你可以去除相关的逻辑部分，延迟计算，并且对可复用的图像和数据进行缓存。另外一个方法就是通过设置默认显示的数据和图像来对 Cell 进行复用。同样，你也可以使用多线程技术来获取图像或者数据，

在稍后对相应的 Cell 进行填充。从用户的角度来讲，这种方式会让他们有如丝绸般的滑动体验，图片被迅速地加载。（第三章上半部分到此翻译完成）。