# 25 个增强 iOS 应用程序性能的提示和技巧

**初级**

1.使用 ARC 进行内存管理

2.在适当的情况下使用 reuseIdentifier

3.尽可能将 View 设置为不透明（Opaque）

4.避免臃肿的 XIBs

5.不要阻塞主线程

6.让图片的大小跟 UIImageView 一样

7.选择正确的集合

8.使用 GZIP 压缩

**中级：**

9.重用和延迟加载 View

10.缓存、缓存、缓存

11.考虑绘制

12.处理内存警告

13.重用花销很大的对象

14.使用 Sprite Sheets

15.避免重新处理数据

16.选择正确的数据格式

17.设置适当的背景图片

18.降低 Web 内容的影响

19.设置阴影路径

20.优化 TableView

21.选择正确的数据存储方式

高级

22.加速启动时间

23.使用 Autorelease Pool

24.缓存图片 — 或者不缓存

25.尽量避免 Date 格式化

# 25 个增强 iOS 应用程序性能的提示和技巧(初级篇)

**初级**

在开发过程中，下面这些初级技巧需要时刻注意：

1.使用 ARC 进行内存管理
2.在适当的情况下使用 reuseIdentifier
3.尽可能将 View 设置为不透明（Opaque）
4.避免臃肿的 XIBs
5.不要阻塞主线程
6.让图片的大小跟 UIImageView 一样
7.选择正确的集合
8.使用 GZIP 压缩

**1) 使用 ARC 进行内存管理**

ARC 是在 iOS 5 中发布的，它解决了最常见的内存泄露问题——也是开发者最容易健忘的。ARC 的全称是“Automatic Reference Counting”——自动引用计数，它会自动的在代码中做 retain/release 工作，开发者不用再手动处理。

下面是创建一个 View 通用的一些代码块：

```
1. UIView *view = [[UIView alloc] init];
2. // ...
3. [self.view addSubview:view];
4. [view release];
```

在上面代码结束的地方很容易会忘记调用 release。不过当使用 ARC 时，ARC 会在后台自动的帮你调用 release。

ARC 除了能避免内存泄露外，还有助于程序性能的提升：当程序中的对象不再需要的时候，ARC 会自动销毁对象。所以，你应该在工程中使用 ARC。

下面是学习 ARC 的一些资源：
苹果的官方文档

Matthijs Hollemans 的初级 ARC

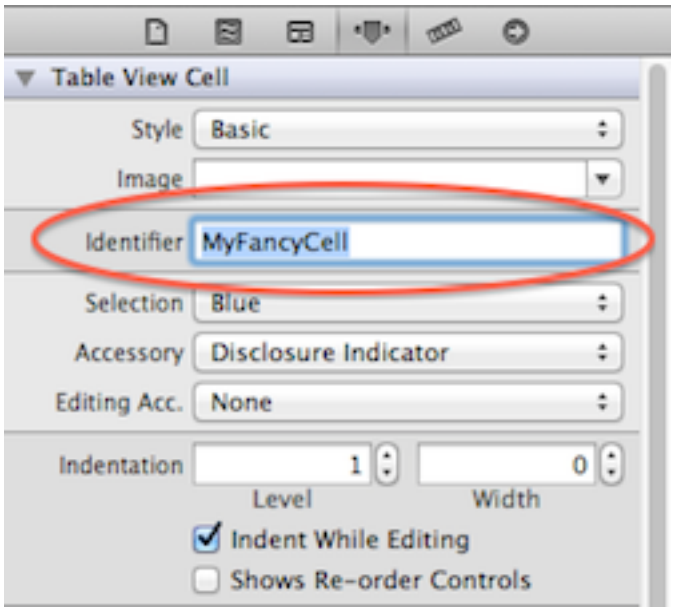
Tony Dahbura 的如何在 Cocos2D 2.X 工程中使用 ARC

如果你仍然不确定 ARC 带来的好处，那么看一些这篇文章：8 个关于 ARC 的神话——这能够让你相信你应该在工程中使用 ARC！

值得注意的是，ARC 并不能避免所有的内存泄露。使用 ARC 之后，工程中可能还会有内存泄露，不过引起这些内存泄露的主要原因是：block，retain 循环，对 CoreFoundation 对象（通常是 C 结构）管理不善，以及真的是代码没写好。

这里有一篇文章是介绍哪些问题是 ARC 不能解决的 — 以及如何处理这些问题。

**2) 在适当的情况下使用 reuseIdentifier**




在适当的情况使用 reuseIdentifier

在 iOS 程序开发中一个普遍性的错误就是没有正确的为 UITableViewCells、

UICollectionViewCells 和 UITableViewHeaderFooterViews 设置 reuseIdentifier。

为了获得最佳性能，当在 tableView:cellForRowAtIndexPath:方法中返回 cell 时，table view 的数据源一般会重用 UITableViewCell 对象。table view 维护着 UITableViewCell 对象的一个队列或者列表，这些数据源已经被标记为重用了。

**如果没有使用 reuseIdentifier 会发生什么？** 如果你在程序中没有使用 reuseIdentifier，table view 每次显示一个 row 时，都会配置一个全新的 cell。这其实是一个非常消耗资源的操作，并且会影响程序中 table view 滚动的效率。

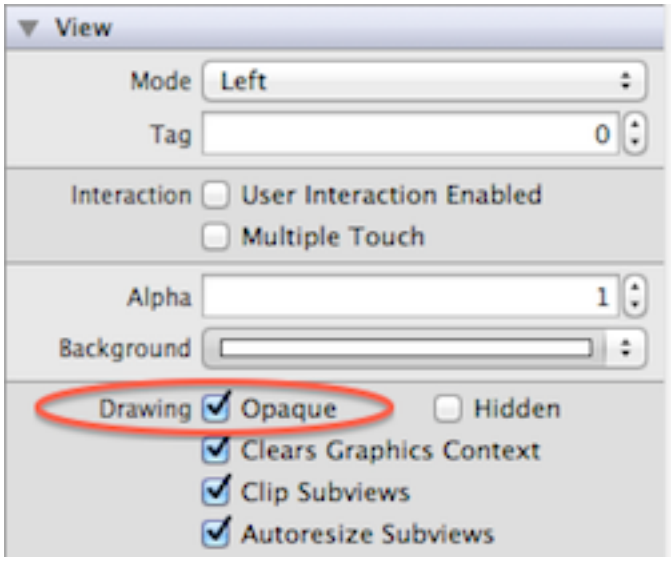
自 iOS 6 以来，你可能还希望 header 和 footer views，以及 UICollectionView 的 cell 和 supplementary views。

为了使用 reuseIdentifiers，在 table view 请求一个新的 cell 时，在数据源中调用下面的方法：

```
1. static NSString *CellIdentifier = @"Cell";
2. UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:C
   ellIdentifier forIndexPath:indexPath];
```

如果 table view 维护的 UITableViewCell 队列或列表中有可用的 cell，则从队列从移除一个已经存在的 cell，如果没有的话，就从之前注册的 nib 文件或类中创建一个新的 cell。如果没有可以重用的 cell，并且没有注册 nib 文件或类，tableview 的 dequeueReusableCellWithIdentifier:方法会返回一个 nil。

**3) 尽可能将 View 设置为不透明（Opaque）**




尽量将 view 设置为 Opaque

如果 view 是不透明的，那么应该将其 opaque 属性设置为 YES。为什么要这样做呢？这样设置可以让系统以最优的方式来绘制 view。opaque 属性可以在 Interface Builder 或代码中设置。

苹果的官方文档对 opaque 属性有如下解释：

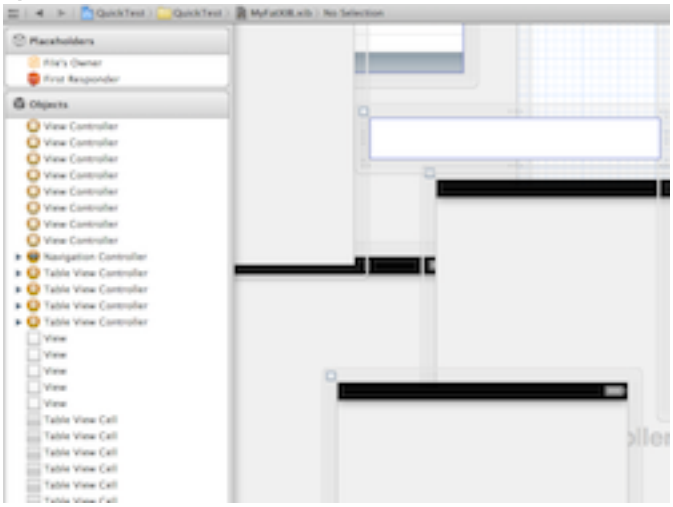
This property provides a hint to the drawing system as to how it should treat the view. If set to YES, the drawing system treats the view as fully opaque, which allows the drawing system to optimize some drawing operations and improve performance. If set to NO, the drawing system composites the view normally with other content. The default value of this property is YES.

（opaque 属性提示绘制系统如何处理 view。如果 opaque 设置为 YES，绘图系统会将 view 看为完全不透明，这样绘图系统就可以优化一些绘制操作以提升性能。如果设置为 NO，那么绘图系统结合其它内容来处理 view。默认情况下，这个属性是 YES。）

如果屏幕是静止的，那么这个 opaque 属性的设置与否不是一个大问题。但是，如果 view 是嵌入到 scroll view 中的，或者是复杂动画的一部分，不将设置这个属性的话肯定会影响

程序的性能！可以通过模拟器的 Debug\Color Blended Layers 选项来查看哪些 view 没有设置为不透明。为了程序的性能，尽可能的将 view 设置为不透明！

**4) 避免臃肿的 XIBs**



避免臃肿的 XIBs

在 iOS 5 中开始使用 Storyboards，并且将替代 XIBs。不过在有些情况下 XIBs 仍然有用。如果你的程序需要运行在装有 iOS 5 之前版本的设备上，或者要自定义可重用的 view，那么是避免不了要使用 XIBs 的。

如果必须要使用 XIBs 的话，尽量让 XIBs 文件简单。并且每个 view controller 对于一个 XIB 文件，如果可以的话，把一个 view controller 的 view 不同的层次单独分到一个 XIBs 文件中。

（注意：当把一个 XIB 文件加载到内存时，XIB 文件中的所有内容都将被加载到内存中，包括图片。如果有一个 view 还不立即使用的话，就会造成内存的浪费。而这在 storyboard 中是不会发生的，因为 storyboard 还在需要的时候才实例化一个 view controller。）

当加载 XIB 时，所有涉及到的图片都将被缓存，并且如果是开发的程序是针对 OS X 的话，声音文件也会被加载。苹果的官方文档这样说：

When you load a nib file that contains references to image or sound resources, the nib-loading code reads the actual image or sound file into memory and and caches it. In OS X, image and sound resources are stored in named caches so that you can access them later if needed. In iOS, only image resources are stored in named caches. To access images, you use the imageNamed: method of NSImage or UIImage, depending on your platform.

（当加载一个 nib 文件时，也会将 nib 文件涉及到的图片或声音资源加载到内存中，nib-loading 代码会将实际的图片或声音文件读取到内存中，并一直缓存着。在 OS X 中，图片和声音资源都存储在命名缓存中，这样之后如果需要的话，可以对其进行访问。在 iOS 中，只有图片资源被存储到命名缓存中。要访问图片的话，使用 NSImage 或 UIImage（根据不同的系统）的 imageNamed:方法即可。）

显然，在使用 storyboard 时也会发生类似的缓存操作；不过我没有找到相关内容的任何资料。想要学习 storyboard 的更多知识吗？可以看看 Matthijs Hollemans 写的 iOS 5 中：初级 Storyboard Part 1 和 Part2。

**5) 不要阻塞主线程**

永远都不要在主线程做繁重的任务。因为 UIKit 的左右任务都在主线程中进行，例如绘制、触摸管理和输入响应。

在主线程做所有任务的风险是：如果你的代码阻塞了主线程，那么程序将出现反应迟钝。这回招致用户在 App Store 上对程序的差评！

在执行 I/O 操作中，大多数情况下都会祖塞主线程，这些操作需要从读写外部资源，例如磁盘或者网络。

关于网络操作可以使用 NSURLConnection 的如下方法，以异步的方式来执行：

```
+ (void)sendAsynchronousRequest:(NSURLRequest *)request queue:(NSOperationQueue *)queue completionHandler:(void (^)(NSURLResponse*, NSData*, NSError*))handler
```

或者使用第三方框架，例如 AFNetworking。

如果你需要做一些其它类型开销很大的操作(例如执行一个时间密集型的计算或者对磁盘进行读写)，那么就使用 GCD（Grand Central Dispatch），或 NSOperations 和 NSOperationQueues。

下面的代码是使用 GCD 的一个模板：

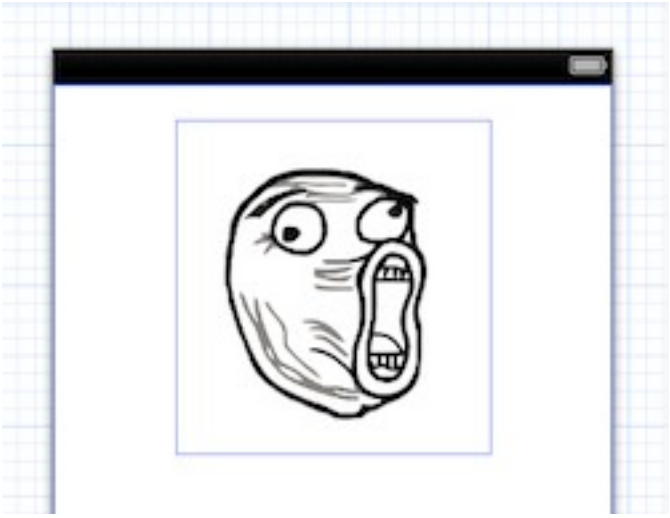
```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // switch to a background thread and perform your expensive operation

    dispatch_async(dispatch_get_main_queue(), ^{
        // switch back to the main thread to update your UI

    });
});
```

如上代码，为什么在第一个 dispatch_async 里面还嵌套了一个 dispatch_async 呢？这是因为关于 UIKit 相关的代码需要在主线程里面执行。
可以看看 Ray Wenderlich 中的教程：iOS 中多线程和 GCD—初级，以及 Soheil Azarpour 的如何使用 NSOperations 和 NSOperationQueues 教程。

### 6) 让图片的大小跟 UIImageView 一样

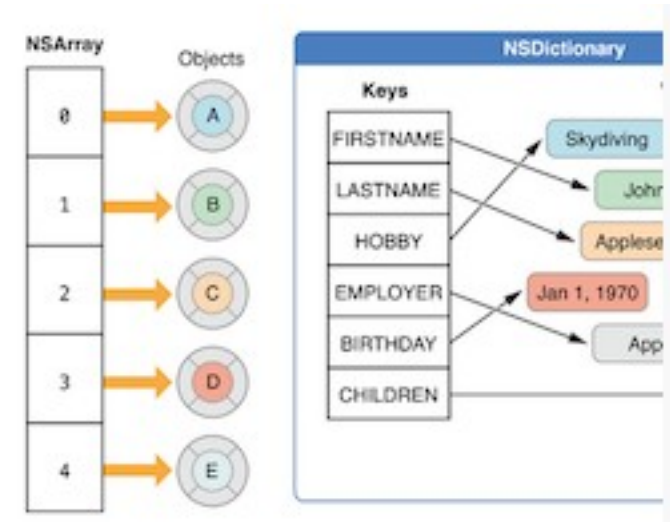


确保图片和 UIImageView 大小一致

如果需要将程序 bundle 中的图片显示到 UIImageView 中，请确保图片和 UIImageView 的大小是一样的。因为图片的缩放非常耗费资源，特别是将 UIImageView 嵌入到 UIScrollView 中。

如果是从远程服务中下载图片，有时候你控制不了图片的尺寸，或者在下载之前无法在服务器上进行图片的缩放。这种情况，当图片下载完之后，你可以手动进行图片的缩放——做好是在后台线程中！——然后再在 UIImageView 中使用缩放过的图片。

**7) 选择正确的集合**




选择正确的集合

学习使用最适合的类或对象是编写高效代码的基础。特别是在处理集合数据时，尤为重要。

苹果的官网上有一篇文章：集合编程主题(Collections Programming Topics)——详细的介绍了在集合数据中可以使用的类，以及什么情况下使用哪个类。在使用集合时，每个开发者都应该阅读一下这个文档。

太长，不想阅读(TLDR)？下面是常见集合类型的一个简介：

•数组：是一个值按顺序排列的一个列表。根据索引可以快速查找，不过根据值进行查找就比较慢，另外插入和删除也比较慢。

•字典：存储键/值对。根据键可以快速查找。

•Sets：是一个值无序排列的列表，根据值可以快速查找，另外插入和删除也比较快。

**8) 使用 GZIP 压缩**


使用 GZIP 压缩

越来越多的程序依赖于外部数据，这些数据一般来自远程服务器或者其它的外部 APIs。有时候你需要开发一个程序来下载一些数据，这些数据可以是 XML，JSON，HTML 或者其它一些文本格式。

问题是在移动设备上的网络是不确定的。用户的设备可能在 EDGE 网络一分钟，然后接着又在 3G 网络中。不管在什么情况下，都不要让用户等待。

有一个可以优化的选择：使用 GZIP 对网络传输中的数据进行压缩，这样可以减小文件的大小，并加快下载的速度。压缩对于文本数据特别有用，因为文本具有很高的压缩比。

iOS 中，如果使用 NSURLConnection，那么默认情况下已经支持 GZIP 压缩了，并且基于 NSURLConnection 的框架页支持 GZIP 压缩，如 AFNetworking。甚至有些云服务提供商已经提供发送经压缩过的响应内容，例如 Google App Engine。

这里有一篇关于 GZIP 压缩很好的文章，介绍了如何在 Apache 活 IIS 服务器中开启支持 GZIP 压缩。

# 25 个增强 iOS 应用程序性能的提示和技巧--中级篇

在性能优化时，当你碰到一些复杂的问题，应该注意和使用如下技巧：

9.重用和延迟加载 View

10.缓存、缓存、缓存

11.考虑绘制

12.处理内存警告

13.重用花销很大的对象

14.使用 Sprite Sheets

15.避免重新处理数据

16.选择正确的数据格式

17.设置适当的背景图片

18.降低 Web 内容的影响

19.设置阴影路径

20.优化 TableView

21.选择正确的数据存储方式

**中级性能提升**

现在，在进行代码优化时，你已经能够完成一些初级性能优化了。但是下面还有另外一些优

化方案，虽然可能不太明显（取决于程序的架构和相关代码），但是，如果能够正确的利用好这些方案，那么它们对性能的优化将非常明显！

**9) 重用和延迟加载 View**

程序界面中包含更多的 view，意味着界面在显示的时候，需要进行更多的绘制任务；也就意味着需要消耗更多的 CPU 和内存资源。特别是在一个 UIScrollView 里面加入了许多 view。

这种情况的管理技巧可以参考 UITableView 和 UICollectionView 的行为：不要一次性创建所有的 subview，而是在需要的时候在创建 view，并且当 view 使用完毕时候将它们添加到重用队列中。

这样就可以仅在 UIScrollView 滚动的时候才配置 view，以此可以避免分配创建 view 的带来的成本——这可能是非常耗资源的。

现在有这样的一个问题：在程序中需要显示的 view 在什么时机创建（比如说，当用户点击某个按钮，需要显示某个 view）。这里有两种可选方法：
在屏幕第一次加载以及隐藏的时候，创建 view；然后在需要的时候，再把 view 显示出来。
直到需要显示 view 的时候，才创建并显示 view。

每种方法都有各自的优点和缺点。第一种方法需要消耗更多的内容，因为创建出来的 view 一直占据着内存，直到 view 被 release 掉。不过，使用这种方法，当用户点击按钮时，程序会很快的显示出 view，因为只需要修改一下 view 的可见性即可。而第二种方法则产生相反的效果；当需要的时候猜创建 view，这会消耗更少的内存；不过，当用户点击按钮的时候，不会立即显示出 view。

**10) 缓存、缓存、缓存**

在开发程序时，一个重要的规则就是“缓存重要的内容”——这些内容一般不会改变，并且访问的频率比较高。

可以缓存写什么内容呢？比如远程服务器的响应内容，图片，甚至是计算结果，比如 UITableView 的行高。

NSURLConnection 根据 HTTP 头的处理过程，已经把一些资源缓存到磁盘和内存中了。你甚至可以手动创建一个 NSURLRequest ，让其只加载缓存的值。
下面的代码片段一般用在为图片创建一个 NSURLRequest：

```
1. + (NSMutableURLRequest *)imageRequestWithURL:(NSURL *)url {
```

```
    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];

    request.cachePolicy = NSURLRequestReturnCacheDataElseLoad; // this will make sure the request always returns the cached image
    request.HTTPShouldHandleCookies = NO;
    request.HTTPShouldUsePipelining = YES;
    [request addValue:@"image/*" forHTTPHeaderField:@"Accept"];

    return request;
}
```

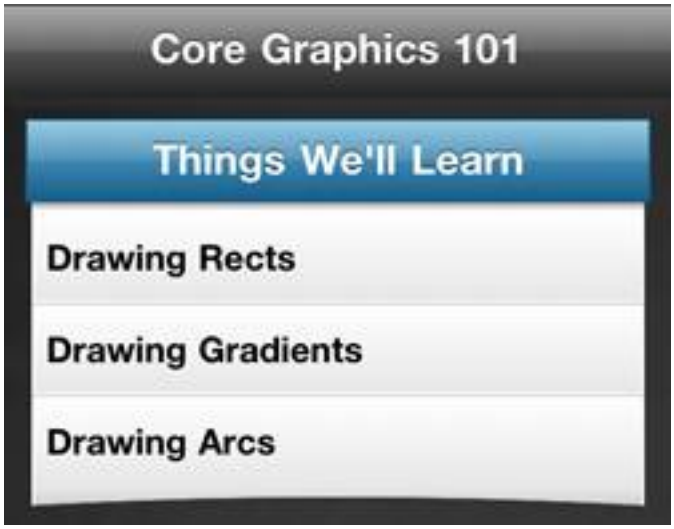
注意：你可以使用 NSURLConnection 抓取一个 URL 请求，但是同样可以使用 AFNetworking 来抓取，这种方法不用修改所有网络相关的代码——这是一个技巧！

更多关于 HTTP 缓存, NSURLCache, NSURLConnection 以及相关的内容，那么看一下 NSHipster 中的 the NSURLCache entry。

如果你需要缓存的内容没涉及到 HTTP 请求，那使用 NSCache。NSCache 的外观和行为与 NSDictionary 类似，但是，当系统需要回收内存时，NSCache 会自动的里面存储的内容。Mattt Thompson 在 NSHipster 上写了一篇关于 NSCache 非常不错的文章。

更多关于 HTTP 缓存的内容，建议读一下 Google 的这篇文章：best-practices document on HTTP caching。

**11) 考虑绘制**




在 iOS 中制作漂亮的按钮有多种方法。可以使用全尺寸图片，可缩放图片，或使用 CALayer, CoreGraphics， 甚至是 OpenGL 来手动测量和绘制按钮。

这些方法的复杂程度不同，性能也有区别。这篇关于 iOS 中图形性能的文章值得一读。其中 Andy Matuschak（曾经是苹果的 UIKit 小组的组员）对这篇文章的评论中，对于不同的方法及其性能权衡有非常好的一个见解。

简单来说，使用预渲染图片技术是最快的，因为 iOS 中不用等到在屏幕上显示的时候才创建图形和对形状进行绘制（图片已经创建好了!）。这样带来的问题是需要把所有的图片都放到程序 bundle 中，从而增加了程序的大小。因此使用可伸缩图片在这里将排上用场了：可以移除“浪费”空间的图片——iOS 可以重复利用。并且针对不同的元素（例如按钮）不需要创建不同的图片。

不过，使用图片的话会失去代码对图片的控制能力，进而针对不同的程序，就需要重复的生成每一个需要的图片，并反复的放到每个程序中。这个处理过程一般会比较慢。另外一点就是如果你需要一个动画，或者许多图片都要进行轻微的调整（比如多个颜色的覆盖），那么需要在程序中加入许多图片，进而增加了程序 bundle 的大小。

总的来说，要考虑一下什么才是最重要的：绘制性能还是程序大小。一般来说都重要，所以在同一个工程中，应该两种都应考虑。

**12) 处理内存警告**

当系统内存偏低时，iOS 会通知所有在运行的程序。苹果的官方文档中介绍了如何处理低内存警告：

If your app receives this warning, it must free up as much memory as possible. The best way to do this is to remove strong references to caches, image objects, and other data objects that can be recreated later.

如果程序收到了低内存警告，在程序中必须尽量释放内存。最佳方法就是移除强引用的涉及到的缓存，图片对象，以及其它可以在之后使用时还可以重新创建的数据对象。

UIKit 中提供了如下几种方法来接收低内存（low-memory）警告：
实现 app delegate 中的 applicationDidReceiveMemoryWarning: 方法。
在 UIViewController 子类中重写(Override)didReceiveMemoryWarning 方法。
在通知中心里面注册 UIApplicationDidReceiveMemoryWarningNotificatio 通知。

在收到以上任意的警告时，需要立即释放任何不需要的内存。

例如，UIViewController 的默认情况是清除掉当前不可见的 view；在 UIViewController 的子类中，可以清除一些额外的数据。程序中不没有显示在当前屏幕中的图片也可以 release 掉。

当收到低内存警告时，尽量释放内存是非常重要的。否则，运行中的程序有可能会被系统杀掉。

不过，在清除内存时要注意一下：确保被清除的对象之后还可以被创建出来。另外，在开发程序的时候，请使用 iOS 模拟器中的模拟内存警告功能对程序进行测试！

**13) 重用花销很大的对象**


有些对象的初始化非常慢——比如 NSDateFormatter 和 NSCalendar。不过有时候可以避免使用这些对象，例如在解析 JSON/XML 中的日期时。

当使用这些对象时，为了避免性能上的瓶颈，可以尝试尽量重用这些对象——在类中添加一个属性或者创建一个静态变量。

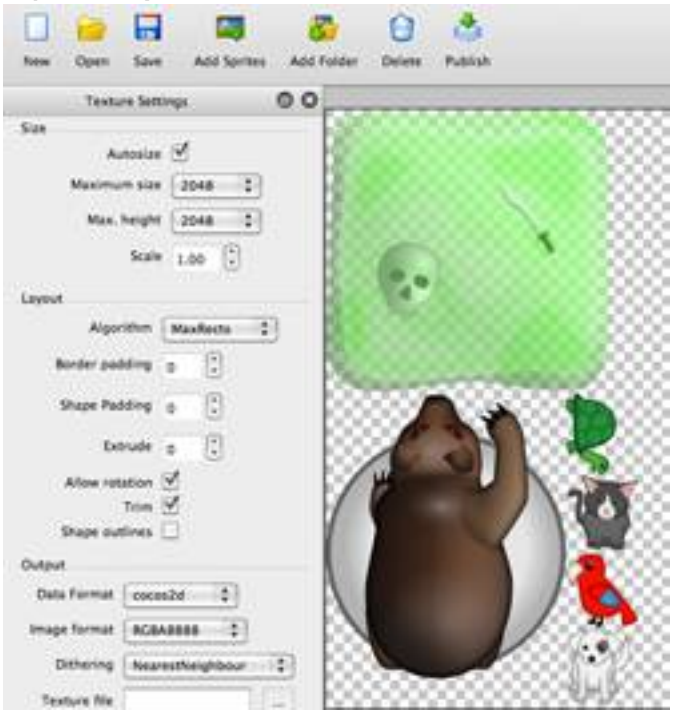
注意，如果使用静态变量的话，对象会在程序运行的时候一直存在，就像单例一样。

下面的代码演示创建一个延迟加载的日期格式属性。第一次调用属性的时候，会创建一个新的日期格式。之后再调用的话，会返回已经创建好的实例对象：

```
1. // in your .h or inside a class extension
2. @property (nonatomic, strong) NSDateFormatter *formatter;
3.
4. // inside the implementation (.m)
5. // When you need, just use self.formatter
6. - (NSDateFormatter *)formatter {
7.     if (! _formatter) {
8.         _formatter = [[NSDateFormatter alloc] init];
9.         _formatter.dateFormat = @"EEE MMM dd HH:mm:ss Z yyyy"; // twit
   ter date format
10.     }
11.     return _formatter;
12. }
```

另外，还需要记住的是在设置 NSDateFormatter 的日期格式时，同样跟创建新的一个 NSDateFormatter 实例对象时一样慢！因此，在程序中如果需要频繁的处理日期格式，那么对 NSDateFormatter 进行重用是非常好的。

**14) 使用 Sprite Sheets**




使用 sprite sheets

你是一个游戏开发者吗？是的话那么 sprite sheets 是最佳选择之一。使用 Sprite sheets 跟常用的绘制方法比起来，绘制更快，并且消耗更少的内存。

下面是两个非常不错的 sprite sheets 教程：

如何在 Cocos2D 中使用动画和 Sprite Sheets

如何在 Cocos2D 中使用纹理包（Texture Packer）和像素格式来创建并优化 Sprite Sheets 。（第二个教程详细的介绍了像素格式——在游戏中可以衡量性能的影响）

如果还不熟悉 sprite sheets，可以看看这里的介绍：**SpriteSheets – 视频, Part 1** 和 Part 2 。这两个视频的作者是 Andreas Löw，他是纹理包(Texture Packer)的创建者，纹理包是创建 sprite sheets 的重要工具。

除了使用 sprite sheets 外，这里还介绍了一些用于游戏开发中的技巧，例如，如果你有很多 sprite（比如射击类游戏中），那么可以重用 sprite，而不用每次都创建 sprite。

**15) 避免重新处理数据**

许多程序都需要从远程服务器中获取数据，以满足程序的需求。这些数据一般是 JSON 或 XML 格式。在请求和接收数据时，使用相同的数据结构非常重要。

为什么呢？在内存中把数据转换为适合程序的数据格式是需要付出额外代价的。

例如，如果你需要在 table view 中显示一些数据，那么请求和接收的数据格式最好是数组格式的，这样可以避免一些中间操作——将数据转换为适合程序使用的数据结构。

类似的，如果程序是根据键来访问具体的值，那么最好请求和接收一个键/值对字典。

在第一时间获得的数据就是所需要格式的，可以避免将数据转换为适合程序的数据格式带来的额外代价。

**16) 选择正确的数据格式**


选择正确的数据格式

将数据从程序传到网络服务器中有多种方法，其中使用的数据格式基本都是 JSON 和 XML。你需要做的就是在程序中选择正确的数据格式。

JSON 的解析速度非常快，并且要比 XML 小得多，也就意味着只需要传输更少数据。并且在 iOS5 之后，已经有内置的 JSON 反序列化 API 了，所以使用 JSON 是很容易的。

不过 XML 也有它自己的优势：如果使用 SAX 方法来解析 XML，那么可以边读 XML 边解析，并不用等到全部的 XML 获取到了才开始解析，这与 JSON 是不同的。当处理大量数据时，这种方法可以提升性能并减少内存的消耗。

**17) 设置适当的背景图片**

在 iOS 编码中，跟别的许多东西类似，这里也有两种方法来给 view 设置一个背景图片：

1.可以使用 UIColor 的 colorWithPatternImge 方法来创建一个颜色，并将这个颜色设置为 view 的背景颜色。
2.可以给 view 添加一个 UIImageView 子视图。

如果你有一个全尺寸的背景图片，那么应该使用 UIImageView，因为 UIColor 的 colorWithPatternImge 方法是用来创建小图片的——该图片会被重复使用。此时使用 UIImageView 会节省很多内存。

```
1. // You could also achieve the same result in Interface Builder
2. UIImageView *backgroundView = [[UIImageView alloc] initWithImage:[UIIm
   age imageNamed:@"background"]];
3. [self.view addSubview:backgroundView];
```

不过，如果你计划用小图片当做背景，那么应该使用 UIColor 的 colorWithPatternImge 方法。这种情况下绘制速度会很快，并且不会消耗大量的内存。

```
1. self.view.backgroundColor = [UIColor colorWithPatternImage:[UIImage im
   ageNamed:@"background"]];
```

**18) 降低 Web 内容的影响**

UIWebView 非常有用。用它可以很容易的显示 web 内容，甚至可以构建 UIKit 空间难以显示的内容。

不过，你可以能已经注意到程序中使用的 UIWebView 组建没有苹果的 Safari 程序快。这是因为 JIT 编译限制了 WebKit 的 Nitro 引擎的使用。

因此为了获得更加的性能，需要调整一下 HTML 的大小。首先就是尽量的摆脱 JavaScript，并避免使用大的矿建，例如 jQuery。有时候使用原始的 JavaScript 要比别的框架快。

另外，尽量的异步加载 JavaScript 文件——特别是不直接影响到页面行为时，例如分析脚本。

最后——让使用到的图片，跟实际需要的一样大小。如之前提到的，尽量使用 sprite sheets，以此节省内存和提升速度。

更多相关信息，可以看一下：WWDC 2012 session #601 – 在 iOS 中优化 UIWebView 和网站中的 Web 内容。

**19) 设置阴影路径**

如果需要在 view 活 layer 中添加一个阴影，该如何处理呢？大多数开发者首先将 QuartzCore 框架添加到工程中，然后添加如下代码：

```
1. #import &lt;QuartzCore/QuartzCore.h&gt;
2.
3. // Somewhere later ...
4. UIView *view = [[UIView alloc] init];
5.
6. // Setup the shadow ...
7. view.layer.shadowOffset = CGSizeMake(-1.0f, 1.0f);
8. view.layer.shadowRadius = 5.0f;
9. view.layer.shadowOpacity = 0.6;
```

上面这种方法有一个问题，Core Animation 在渲染阴影效果之前，必须通过做一个离屏(offscreen)才能确定 view 的形状，而这个离屏操作非常耗费资源。下面方法可以更容易地让系统进行阴影渲染：设置阴影路径！

```
1. view.layer.shadowPath = [[UIBezierPath bezierPathWithRect:view.bounds]
    CGPath];
```

通过设置阴影路径，iOS 就不用总是再计算该如何绘制阴影了。只需要使用你预先计算好的路径即可。有一点不好的是，根据 view 的格式，自己可能很难计算出路径。另外一个问题就是当 view 的 frame 改变时，必须每次都更新一下阴影路径。

如果你想了解更多相关信息，可参看 Mark Pospesel 的一篇文章：shadowPath。

**20) 优化 TableView**

Table views 需要快速的滚动——如果不能的话，用户会感觉到停顿。为了让 table view 平滑的滚动，确保遵循了如下建议：

1.设置正确的 reuseIdentifer 以重用 cell。

2.尽量将 view 设置为不透明，包括 cell 本身。

3.避免渐变，图像缩放以及离屏绘制。

4.如果 row 的高度不相同，那么将其缓存下来。

5.如果 cell 显示的内容来此网络，那么确保这些内容是通过异步来获取的。

6.使用 shadowPath 来设置阴影。

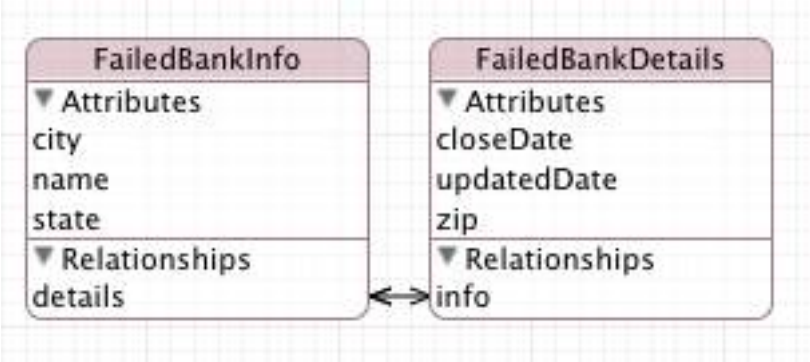
7.减少 subview 的数量。

8.在 cellForRowAtIndexPath:中尽量做更少的操作。如果需要做一些处理，那么最好做过一次之后，就将结果缓存起来。

9.使用适当的数据结构来保存需要的信息。不同的结构会带来不同的操作代价。

10.使用 rowHeight, sectionFooterHeight 和 sectionHeaderHeight 来设置一个恒定 高度，而不要从 delegate 中获取。

21) 选择正确的数据存储方式




选择正确的数据存储方式

当需要存储和读取大量的数据时，该如何选择存储方式呢？有如下选择：

1.使用 NSUserDefaults 进行存储

2.保存为 XML，JSON 或 Plist 格式的文件

3.利用 NSCoding 进行归档

4.存储到一个本地数据库，例如 SQLite。

5.使用 Core Data.

使用 NSUserDefaults 有什么问题呢？虽然 NSUserDefaults 很好并且容易，不过只只针对

于存储小量数据（比如你的级别，或者声音是开或关）。如果要存储大量的数据，最好选择别的存储方式。

大量数据保存为结构化的文件也可能会带来问题。一般，在解析这些结构数据之前，需要将内容全部加载到内存中，这是很消耗资源的。虽然可以使用 SAX 来处理 XML 文件，但是这有点复杂。另外，加载到内存中的所有对象，不一定全部都需要用到。

那么使用 NSCoding 来保存大量数据怎么样呢？因为它同样是对文件进行读写，因此依然存在上面说的问题。

要保存大量的数据，最好使用 SQLite 或 Core Data。通过 SQLite 或 Core Data 可以进行具体的查询——只需要获取并加载需要的数据对象——避免对数据进行不合理的搜索。在性能方面，SQLite 和 Core Data 差不大。

SQLite 和 Core Data 最大的区别实际上就是用法上。Core Data 代表一个对象模型，而 SQLite 只是一个 DBMS。一般，苹果建议使用 Core Data，不过如果你有特殊的原因不能使用 Core Data 的话，可以使用低级别的 SQLite。

在程序中，如果选择使用 SQLite，这里有个方便的库 FMDB ：可以利用该库操作 SQLite 数据库，而不用深入使用 SQLite C API。

# 25 个增强 iOS 应用程序性能的提示和技巧--高级篇

**高级**

当且仅当下面这些技巧能够解决问题的时候，才使用它们：

22.加速启动时间

23.使用 Autorelease Pool

24.缓存图片 — 或者不缓存

25.尽量避免 Date 格式化

**高级性能提升**

寻找一些高明的方法，让自己变为一个全代码忍者？下面这些高级的性能优化技巧可以在适当的时候让程序尽可能的高效运行！

### 22) 加速启动时间

能快速的启动程序非常重要，特别是在用户第一次启动程序时。第一映像对程序来说非常重要！

让程序尽量快速启动的方法就是尽量以异步方式执行任务，例如网络请求，数据访问或解析。

另外，避免使用臃肿的 XIBs，因为 XIB 的加载是在主线程中进行的。但是记住 storyboard 没有这样的问题——所以如果可以的话就使用 storyboard 吧！

注意：在利用 Xcode 进行调试时，watchdog 不会运行，所在设备中测试程序启动性能时，不要将设备连接到 Xcode。

### 23) 使用 Autorelease Pool

NSAutoreleasePool 负责释放一个代码块中的自动释放对象。一般都是由 UIKit 来创建的。不过有些情况下需要手动创建 NSAutoreleasePool。

例如，如果在代码中创建了大量的临时对象，你将注意到内存使用量在增加，直到这些对象被释放。问题是只有当 UIKit 耗尽了 autorelease pool，这些对象才会被释放，也就是说当不再需要这些对象之后，这些对象还在内存中占据着资源。

不过这个问题完全可以避免：在@autoreleasepool 代码块中创建临时对象，如下代码：

```
1. NSArray *urls = &lt;# An array of file URLs #&gt;;
2. for (NSURL *url in urls) {
3.     @autoreleasepool {
4.         NSError *error;
5.         NSString *fileContents = [NSString stringWithContentsOfURL:url

6.                                                 encoding:NSUTF8StringEncoding
    error:&amp;error];
7.         /* Process the string, creating and autoreleasing more objects
   . */
8.     }
9. }
```

当每次迭代完之后，都会释放所有的 autorelease 对象。

关于 NSAutoreleasePool 的更多内容可以阅读苹果的官方文档。

**24) 缓存图片--或者不缓存**

iOS 中从程序 bundle 中加载 UIImage 一般有两种方法。

第一种比较常见：imageNamed。

第二种方法很少使用：imageWithContentsOfFile

为什么有两种方法完成同样的事情呢？imageNamed 的优点在于可以缓存已经加载的图片。

苹果的文档中有如下说法：

This method looks in the system caches for an image object with the specified name and returns that object if it exists. If a matching image object is not already in the cache, this method loads the image data from the specified file, caches it, and then returns the resulting object.

这种方法会在系统缓存中根据指定的名字寻找图片，如果找到了就返回。如果没有在缓存中找到图片，该方法会从指定的文件中加载图片数据，并将其缓存起来，然后再把结果返回。

而 imageWithContentsOfFile 方法只是简单的加载图片，并不会将图片缓存起来。这两个方法的使用方法如下：

```
1. UIImage *img = [UIImage imageNamed:@"myImage"]; // caching
2. // or
3. UIImage *img = [UIImage imageWithContentsOfFile:@"myImage"]; // no cac
   hing
```

那么该如何选择呢？

如果加载一张很大的图片，并且只使用一次，那么就不需要缓存这个图片。这种情况 imageWithContentsOfFile 比较合适——系统不会浪费内存来缓存图片。

然而，如果在程序中经常需要重用的图片，那么最好是选择 imageNamed 方法。这种方法可以节省出每次都从磁盘加载图片的时间。

**25) 尽量避免 Date 格式化**

如果有许多日期需要使用 NSDateFormatter，那么需要小心对待了。如之前（重用花销很大的对象）所提到的，无论什么时候，都应该尽量重用 NSDateFormatters。

然而，如果你需要更快的速度，那么应该使用 C 来直接解析日期，而不是 NSDateFormatter。Sam Soffes 写了一篇文章，其中提供了一些解析 ISO-8601 格式日期字符的串代码。你只需要简单的调整一下其中的代码就可以满足自己特殊的需求了。

这听起来不错把——不过，你相信这还有更好的一个办法吗？

如果你自己能控制处理日期的格式，那么可以选择 Unix timestamps（http://en.wikipedia.org/wiki/Unix_time）。Unix timestamps 是一个简单的整数，代表了从新纪元时间（epoch）开始到现在已经过了多少秒，通常这个新纪元参考时间是 00:00:00 UTC on 1 January 1970。

你可以很容易的见这个时间戳转换为 NSDate，如下所示：

```
1. - (NSDate*)dateFromUnixTimestamp:(NSTimeInterval)timestamp {
2.     return [NSDate dateWithTimeIntervalSince1970:timestamp];
3. }
```

上面这个方法比 C 函数还要快！

注意：许多网络 APIs 返回的时间戳都是毫秒，因此需要注意的是在将这个时间戳传递给 dateFromUnixTimestamp 之前需要除以 1000。

**何去何从？**

强烈建议对程序性能优化感兴趣的读者看看下面列出来的 WWDC 视频。在看视频之前，你需要注册一个 Apple ID（注册后就可以观看所有 WWDC2012 的视频）：

#406: Adopting Automatic Reference Counting
#238: iOS App Performance: Graphics and Animations
#242: iOS App Performance: Memory
#235: iOS App Performance: Responsiveness
#409: Learning Instruments
#706: Networking Best Practices
#514: OpenGL ES Tools and Techniques
#506: Optimizing 2D Graphics and Animation Performance
#601: Optimizing Web Content in UIWebViews and Websites on iOS
#225: Up and Running: Making a Great Impression with Every Launch

下面这些视频来自 WWDC 2011，也非常有用：

#308: Blocks and Grand Central Dispatch in Practice
#323: Introducing Automatic Reference Counting
#312: iOS Performance and Power Optimization with Instruments
#105: Polishing Your App: Tips and tricks to improve the responsiveness and performance
#121: Understanding UIKit Rendering

这里还有更多相关视频，大多数来自 iOS 5 技术讲座：
Optimizing App Performance with Instruments
Understanding iOS View Compositing

基于“Your iOS App Performance Hitlist”视频，Ole Begemann 写了一篇文章。苹果还提供了一篇非常好的文章：性能优化。其中提供的技巧和提示对程序性能提升很有帮助。