我们都是菜鸟，但是我们一直都在努力！

25条提高iOS App性能的建议和技巧

当我们开发iOS应用时，好的性能对我们的App来说是很重要的。你的用户也希望如此，但是如果你的app表现的反应迟钝或者很慢就会让你得到不好的评论。

然而，由于IOS设备的限制有时很难工作得很正确。我们开发时有很多需要我们记住这些容易忘记的决定对性能的影响。

这是为什么我写这篇文章的原因。这篇文章用备忘录的形式集合了25个技巧和诀窍可以用来提高你的app性能。所以耐心的阅读来给你未来的App一个很不错的提高。

Note:在优化代码之前，必须保证有个需要解决的问题！不要陷入"pre-optimizing(预优化)"你的代码。勤 用Instruments分析你的代码，发现任何一个需要提高的地方。Matt Galloway写了一个使用Instruments优化代码的的教程

以下这些技巧分为三个不同那个的级别---基础，中级，高级。

基础

这些技巧你要总是想着实现在你开发的App中。

1. 用ARC去管理内存（Use ARC to Manage Memory）

2.适当的地方使用reuseIdentifier（Use a reuseIdentifier Where Appropriate）

3.尽可能设置视图为不透明（Set View as Opaque When Possible）

4.避免臃肿的XIBs（Avoid Fat XiBs）

5.不要阻塞主进程（Don't Block the Main Thread）

6.调整图像视图中的图像尺寸（Size Images to Image Views）

7.选择正确集合（Choose the Correct Collection）

8.启用Gzip压缩（Enable GZIP Compression）

中级

这些技巧是当你遇到更复杂的情况的时候使用。

9. 重用和延迟加载视图（Reuse and Lazy Load Views）

10.缓存，缓存，缓存（Cache,Cache,Cache）

11.考虑绘图（Consider Drawing）

12.处理内存警告（Handle Memory Warnings）

13.重用大开销对象（Reuse Expensive Objects）

14.使用精灵表（Use Sprite Sheets ）

15.避免重复处理数据（Avoid Re-Processing Data）

16.选择正确的数据格式（Choose the Right Data Format）

17.适当的设置背景图片（Set Background Images Appropriately）

18.减少你的网络占用（Reduce Your Web Footprint）

19.设置阴影路径（Set the Shadow Path ）

20.你的表格视图Optimize Your Table Views）

21.选择正确的数据存储方式（Choose Correct Data Storage Option）

高级

这些技巧你应该只在你很积极认为它们能解决这个问题，而且你觉得用它们很舒适的时候使用。

22.加速启动时间（Speed up Launch Time ）

23.使用自动释放池（Use AutoRelease Pool）

24.缓存图像（Cache Images-Or not ）

25.尽可能避免日期格式化器（Avoid Date Formatters Where Possible）

没有其他的，一起去看看这些技巧吧！

基础的性能提升

1）用ARC去管理内存

ARC是伴随IOS5 一起发布的，它用来消除常见的的内存泄漏。

ARC是"Automatic Reference Counting"的缩写。它自动管理你代码中的retain/release循环，这样你就不必手动做这事儿了。

下面这段代码展示了创建一个view的常用代码

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

这里极其容易忘记在代码结束的地方调用release，ARC将会自动的，底层的为你做这些工作。

除了帮助你你避免内存泄漏，ARC还能保证对象不再使用时立马被回收来提高你的性能。你应该在你的工程里多用ARC。

这里是一些学习更多关于ARC的非常棒的资源

- Apple’s official documentation 苹果的官方文档。
- Matthijs Hollemans’s Beginning ARC in iOS Tutorial
- Tony Dahbura’s How To Enable ARC in a Cocos2D 2.X Project
- 如果你还是不确信ARC的好处，看看这篇文章 eight myths about ARC 说服你为什么用ARC。

值得注意的是ARC不能消除所有的内存泄漏。你依然有可能内存泄漏，这主要可能是由于blocks(块)，引用循环，CoreFoundation对象管理不善（通常是C结构体，或者是确实很糟糕的代码）。

2）适当的地方使用reuseIdentifier

在app开发中的一个常见的为UITableViewCells，UICollectionViewCells，UITableViewHeaderFooterViews设置一个正确的reuseIdentifier(重用标识)。

为了最大化性能，一个tableView的数据源一般应该重用UITableViewCell对象，当它在tableView:cellForRowAtIndexPath:中分配数据给cells的时候。一个表视图维护了一个UITableViewCell对象的队列或者列表，这些对象已被数据源标记为重用。

如果你不用reuseIdentifier 会怎么样呢?

如果你用，你的tableview每显示一行将会配置一个全新的cell。这是非常费事的操作而且绝对会影响你app滚动的性能。

自从引进了iOS6，你应该为header and footer 视图设置reuseIdentifiers，就像在 UICollectionView’s cells 和 supplementary views（补充视图）一样。

使用reuseIdentifiers，当你的数据源要求提供一个新的cell给tableview的时候调用这个方

```
NSString *CellIdentifier = @"Cell";


UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];
```

3）可能的时候设置视图为不透明

如果你有不透明视图（opaque views）--也就是说，没有透明度定义的视图，你应该设置他们的opaque属性为YES。

为什么？ 这会允许系统以最优的方式绘制你的views。这是一个简单的属性可以在Interface Builder 和代码中设置。

苹果的文档 Apple documentation中有对这个属性的描述

这个属性提供了一个提示给图系统如何对待这个视图。如果设置为YES，绘制系统将会把这个视图视为完全不透明。这样允许系统优化一些绘制操作和提高性能。如果设置为NO，绘图系统会复合这个视图和其他的内容，这个属性的默认值是YES

在相对静态的屏幕上，设置opaque属性不会有什么大问题。尽管如此，如果你的视图是嵌入在一个scrollView，或者是一个复杂的动画的一部分，不设置这个属性绝对会影响你的程序的性能。

你也可以使用Debug\Color olor Blended Layers选项 在你的模拟器中形象化的看见没有设置为不透明（opaque）的视图.你的目标应该是尽可能多的设置视图为透明。

4） 避免臃肿的XIB文件

故事板，由iOS5引进，很快的替代XIBs。尽管如此，XIBs在一下情况下依然是很有用的。如果你需要在IOS5之前版本的设备上运行或者你想自定义重用的视图，那么你确实不能避免使用它们。

如果你专注使用XIBs，那么让它们尽量的简单。尝试为一个试图控制器创建一个XIB，如果可能的话，把一个视图控制器的视图分层管理在单独的XIBs中。

注意当你加载一个XIB到内存的时候，它所有的内容都会载入内存，包括所有的图片。如果你有视图但不是要立即使用，那你就浪费了珍贵的内存。值得注意的是这不会发生在故事板中，因为故事版只会在需要的时候实例化一个视图控制器。

当你载入一个xib，所有的图像文件会被缓存，如果是开发OSX，那么音频文件也会被缓存。

Apple's documentation 如是说：

当你载入一个包含了图和声音资源引用的nib文件时，nib加载代码读取实际的图片文件和音频文件到内存中并缓存它。在OS X中，图片和音频资源被存储在已命名的缓存 中这样你可以在之后需要的时候访问它们。在iOS中，只有图片资源被缓存，访问图片，你使用NSImage或者UIImage的imageNamed:方法来访问，具体使用取决于你 的平台。

显然这也发生在使用故事板的时候。尽管如此，我还不能找到这种说法的证据。如果你知道，请给我留言。

想学习更多关于故事板的更多内容吗?看看Matthijs Hollemans的 Beginning Storyboards in iOS 5 Part 1and Part 2.

5）不要阻塞主进程

你永远不应该在主线程中做任何繁重的工作。这是因为UIKIt的所有工作都在主线程中进行，比如绘画，管理触摸，和响应输出。

你的app的所有工作都在主线程上进行就会有阻塞主线程的风险，你的app会表现的反应迟钝。这是在App Store里获一星评论的快速途径!（作者卖萌..）

阻塞主线程最多的情况就是发生在你的app进行I/O操作,包括牵扯到任何需要读写外部资源的任务，比如读取磁盘或者网络

你可以异步的执行网络任务使用NSURLConnection中的这个方法：

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

或者使用第三方框架比如 AFNetworking.

如果你在做任何大开销的操作(比如执行一个耗时的计算，或者读写磁盘)使用Grand Central Dispatch（GCD）或者 NSOperations 和 NSOperationQueues.

使用GCD的模板如下代码所示：

```
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

    });
```

```
});
```

这里为什么dispatch_async 嵌套在第一个的里面？这是因为任何UIKit相关的代码都必须在主线程上执行。

对NSOperation和GCD的详情感兴趣?看看Ray Wenderlich's Multithreading and Grand Central Dispatch on iOS for Beginners 教程,和 Soheil Azarpour's How To Use NSOperations and NSOperationQueues 教程。

6）调整图像视图中的图像尺寸

如果你用UIImageView呈现app束中的图片时，确认图片和UIImageView的尺寸相同。缩放图片会非常的耗时，特别是当你的UIImageView被嵌入UIScrollView。

如果图片是从远程服务器上下载的，有时你没法控制图片尺寸，或者你不能在服务器上在下载之前缩放它。在这些情况下你可以在图片下载完成后手动缩放一次，最好是在后台进程中。然在UIImageView中使用调整尺寸之后的图片。

7）选择正确集合

学着怎么在手头工作中使用最合适的类或对象是写出高效代码的基本。当时用集合是(collections)，这个说法特别对。

可喜的是在苹果开发者文档（ Collections Programming Topics）中有详细解释可用类之间的关系，还有解释各个类的适用情况。这个文档是每个使用集合的人的必读文档。

这是一个最常见的集合类型的快速简介:

- Arrays:有序的值的列表，用index快速查找，通过值查找慢，insert/delete操作慢。
- Dictionaries:存储键/值对.用index快速查找。
- Sets: 无序的值列表。通过值快速查找，insert/delete快。

8）启用Gzip压缩

大量和持续增长的app依赖从远端服务器或者外部APIs获取的外部数据。某些时候你可能会开发一些需要下载XML，JSON，HTML或者其他文本格式的应用。

问题是移动设备不能保证网络环境，用户可能一分钟在边缘网络，下一分钟又是3G网络，无论什么情况下，你不想你的用户一直等待。

一个减少文件大小并加速下载的网络资源的方法是同时在你的服务器和客户端上使用GZIP压缩，对于文本数据这种有高比率压缩的数据来说非常有用。

好消息是iOS早已默认支持GZIP压缩，如果你是使用NSURLConnection或者建立在这之上的框架比如AFNetworking。更好的消息是一切云服务提供商像 Google App Engine早已发送压缩之后的响应数据。

这里有一篇文章great article about GZIP compression 介绍如何在你的Apache或IIS服务器上启用GZIP。

中级性能提升

好的，当谈到优化你的代码时，你应该很自信你已经初级的方法已经完全掌握了。但有时候有的问题的解决方法并不是那么显而易见，它由你app的结构和代码决定，尽管如此，在正确的上下文中，它们可能是没有价值的。

9）重用和延迟加载视图

越多的视图就有越多的绘图操作，最终意味着更多的CPU和内存开销。这说得特别对如果你的app嵌入很多视图在UIScrollView时。

管理这个的技巧是去模拟UITableView 和 UICollectionView的行为:不要一次创建所有的子视图，而是在需要的时候创建，然后把他们假如重用队列中。

这样，你只需要在视图浮动时配置你的视图，避免昂贵的资源分配开销。

视图创建的时机问题也同样适用于你app的其他地方。试想当你点击一个button时呈现一个视图的情景。至少有两种方法:

1.屏幕第一次载入时创建视图并隐藏它。当你需要的时候，显示出来。

2.需要呈现的时候一次创建视图并显示它。

每种方法都有各自的优缺点

使用第一种方法，你消耗了更多内存因为从创建开始到它释放前你都保持了它的内存，然而，当你点击button的时候，你的app会表现得响应快速因为它只需要更改视图的可视化属性。

使用第二种方法会有相反的效果，在需要的时候创建视图，消耗更少的内存，但当button被点击时应用会表现得不那么响应快速。

10）缓存，缓存，缓存

在开发应用时的一个伟大的经验是"Cache what matters"--也就是说那些不大会改变但会平凡被访问的东西。

你能缓存些什么呢？缓存的候选项有远程服务器的响应，图片，已计算过的值（比如UITableView的行高）。

NSURLConnection 根据处理的Http头缓存资源到磁盘或者内存中，你甚至可以手动创建一个NSURLRequest值加载缓存过的值。

这里有一段很棒的代码，用在任何时候你需要针对一个不大会改变的图片创建一个NSURLRequest。

```
+ (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;

}
```

如果想知道更多关于Http caching，NSURLCache，NSURLConnection等内容，请阅读the NSURLCache entry

注意，你可以通过NSURLConnection获取取一个URL请求，AFNetworking也可以。有了这个技巧这样你不用改变任何你的网络代码。

如果要缓存不牵扯到HTTP请求的其他东西，NSCache是很好的选择。

NSCache像NSDictionary，但是当系统需要回收内存的时候会自动的移除内容。

对HTTP Cache感兴趣并想学更多的内容？推荐阅读这篇文章best-practices document on HTTP caching

11）考虑绘图

在IOS中有很多方法可以制作拥有很棒外观的buttons，你可以是由全尺寸的图像，也可以使用调整尺寸之后的图像，或者你用CALayer，CoreGraphics，甚至OpenGL手动的它们。

当然，每种途径都有不同的复杂度级别和不同的性能，这篇文章非常值得一读post about iOS graphics performance here,这是Apple UIKit团队成员Andy Matuschak发表的文章，里面对各种方法有一些非常棒的见解和对性能的权衡。

使用预渲染图片更快，因为iOS不用创建一张图像和绘制图形到屏幕上(图像已经处理好了)。问题是你需要全部把这些图片放进应用束里,增加它的尺寸。那就是为什么使用可调整尺寸的图片是那么好:你通过移除”浪费了的“图片空间来节约空间。你也不需要为不同的元素生成不同的图片。（例如 buttons）

尽管如此，用图片你会失去代码调整你图片的能力，需要一次又一次的生成它们然后把它们加入到应用中。这是个缓慢的过程。另外一点如果你有动画或者很多张稍微变化的图片（例如 颜色叠加），你需要加很多的图片增加了应用束的大小。

总结一下，你需要想对你来说最重要的是什么：绘图性能还是app的大笑.通常两个都很重要，所以你会在一个工程里使用这两种方法。

12）处理内存警告

当系统内存低的时候iOS会通知所有的正在运行的app,关于低内存警告的处理苹果官方文档 official Apple documentation描述：

如果你的应用收到这个警告，它必须尽可能多的释放内存。最好的方法是移除对缓存，图像对象，和其他稍后要创建的对象的强引用。

幸运的是，UIKit提供了一些方法去接收低内存警告:

- 实现App代理中的applicationDidReceiveMemoryWarning:方法。
- 重载你自定义UIViewController子类中的didReceiveMemoryWarning方法。
- 注册接收UIApplicationDidReceiveMemoryWarningNotification的通知

一旦收到这些警告，你的处理方法必须立刻响应并释放不必要的内存。

举例，如果视图当前不可见，UIViewController的默认行为是清除这些视图；子类可以通过清除额外的数据结构来补充父类的默认行为。一个应用程序维护一个图片的缓存，没有在屏幕上的图片都会被释放。

一旦收到内存警告，释放可能的全部内存是很重要的，否则你就有让你的app被系统杀死的的风险。

尽管如此，开始扑杀对象释放内存的时候要小心，因为你需要保证它们会在之后重新创建。当你开发app的时候，用你的模拟器上的模拟内存警告功能测试这种情况。

13）重用大开销对象

有的对象的初始化非常慢--NSDateFormatter 和 NSCalendar是两个例子，但是你不能避免使用它们，当你从 JSON/XML响应中解析日期时。

避免使用这些对象时的性能瓶颈，试着尽可能的重用这些对象。你可以加入你的类中成为一个属性，也可以创建为静态变量。

注意如果你选择了第二种方式，这个对象在app运行的时候会一直保持在内存里，像单例一样。

下面这段代码演示了NSDateFomatter作为一个属性的lazy加载，第一次被调用然后创建它，之后就使用已创建在的实例

```
// in your .h or inside a class extension

@property (nonatomic, strong) NSDateFormatter *formatter;



// inside the implementation (.m)

// When you need, just use self.formatter

- (NSDateFormatter *)formatter {

    if (! _formatter) {

        _formatter = [[NSDateFormatter alloc] init];

        _formatter.dateFormat = @"EEE MMM dd HH:mm:ss Z yyyy"; // twitter date format

    }

    return _formatter;

}
```

同样要记住设置一个NSDateFormatter的日期格式几乎跟创建一个新的一样慢。因此，如果在你的应用中你频繁需要处理多个日期格式，你的代码应该获利于初始化创建，重用，多个NSDateFormatter对象。

14） 使用精灵表

你是一个游戏开发者吗？精灵表是你的好朋友之一.精灵表让绘制比标准屏幕绘制方法更快速，消耗更少的内存。

这里有两个很棒的精灵表使用的教程

1. How To Use Animations and Sprite Sheets in Cocos2D
2. How to Create and Optimize Sprite Sheets in Cocos2D with Texture Packer and Pixel Formats

第二个教程详细覆盖了像素格式，它可以对游戏性能有一个可衡量的影响。

如果对精灵表还不是很熟悉，一个很好的介绍 SpriteSheets – The Movie, Part 1and Part 2. 这些视频的作者是Andreas Löw，一个最流行的创建精灵表的工具Texture Packer的创建者。

除了使用精灵表之外，之前已经说到的内容也可以用在游戏上.举个例子，如果你的游戏有很多精灵，比如在标准的敌人或炮弹射击游戏，你可以重用精灵表额如是每次重新创建它们。

15）避免重复处理数据

很多app调用函数获取远程服务器上的数据.这些数据通常是通过JSON 或者 XML格式来传输。非常重要的是在请求和接收数据的时候努力在两端使用相同的数据结构。

理由?在内存中操纵数据以合适你的数据结构是非常昂贵的。

比如,如果你需要在表格视图中显示数据，最好请求和接收数据是数组的格式,以避免任何中间操纵数据,使其适合你在app中使用的数据结构

相似的，如果你的应用程序依赖于访问特定值的键,那么你可能会想要请求和接收一个键/值对的字典

通过第一次就获取正确格式的数据,在自己的应用程序中你就会避免很多的重复处理工作,使数据符合你的选择的结构。

16）选择正确的数据格式

你可以有很多方法从web 服务中传递数据到你的app中

JSON 是一种通常比XML小且解析更快的格式，它的传输的内容也比较小。自iOS5起，内置的JSON解析很好用 built-in JSON deserialization

尽管如此，XML的一个优势当你使用SAXparsing方法时，你可以传输过程中读取它，在面的非常大的数据时，你不必像JSON一样在数据下载完之后才开始读取。

17）适当的设置背景图片

像iOS编码的其他工作一样，至少有两种不同方式去替换你视图的背景图片。

1. 你可以设置你的视图的背景颜色为UIColor的colorWithPatternImage创建的颜色。
2. 你可以添加一个UIImageView子试图给View

如果你有全尺寸的背景图片，你绝对要用UIImageView，因为UIColor的colorWithPatternImage是重复的创建小的模式图片，在这种情况下用UIImageView方式会节约很多内存。

```
// You could also achieve the same result in Interface Builder


 UIImageView *backgroundView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"background"]];


[self.view addSubview:backgroundView];
```

尽管如此，如果你计划用模式图片背景，你应该是用UIColor的colorWithPatternImage。它更快一些，而且这种情况不会使用很多内存。

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

18）减少你的网络占用

UIWebView 是非常游泳的.它非常容易用来显示web内容，甚至创建你app的视窗。这些都是标准UIKit 空间很难做到的。

尽管如此，你可能注意你可以用在你的app中的UIWebView组件并没有Apple的Safari app快。这是Webkit's的Nitro引擎的限制使用。JIT compilation.

所以为了获得最佳的性能，你需要调整你的HTML。第一件事是尽可能多的避免Javascript，包括避免大的框架比如jQuery。有时使用vanilla Javascript取代依赖的框架会快很多。

随时随地遵循异步加载Javascript文件的实践。特别当它们不直接影响到页面表现的时候，比如分析脚本。

最后，总是要意识到你在用的图片，保持图片的正确尺寸。正如这个教程前面所提到的，利用精灵表的优势来节约内存和提高速度。

想要获取更多的信息，看看WWDC 2012 session #601 – Optimizing Web Content in UIWebViews and Websites on iOS.

19） 设置阴影路径

你需要给视图或者layer添加一个阴影,你应该怎么做?

大多数开发者是添加 QuartzCore框架到工程中,然后写如下代码:

```
01
#import <QuartzCore/QuartzCore.h>
02

03
 // Somewhere later ...
04
```

```
UIView *view = [[UIView alloc] init];


 // Setup the shadow ...


view.layer.shadowOffset = CGSizeMake(-1.0f, 1.0f);


view.layer.shadowRadius = 5.0f;


view.layer.shadowOpacity = 0.6;
```

看起来非常简单,是吧?

不好的是这个方法有一个问题。核心动画必须要先做一幕动画确定视图具体形状之后才渲染阴影，这是非常费事的操作。

这里有个替代方法让系统更好的渲染，设置阴影路径:

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

如果你想知道这个内容的更多技巧，Mark Pospesel 写过一篇post about shadowPath.

设置阴影路径，iOS不需要总是计算如何绘制阴影。而是用已经计算好的的路径。坏消息是它依赖与你的视图格式，你是视图可能很难计算这个路径。另一个问题是你需要在每次视图的框架改变时更新阴影路径。

20） 优化你的表格视图

表格视图需要快速的滚动，如果不能，用户能确切注意到很滞后。

为了让你的表格视图流畅的滚动，保证你实现了下列的建议。

- 通过正确的reuseIdentifier重用cells
- 尽量多的设置views 为不透明，包括cell本身。
- 避免渐变，图像缩放，屏幕以外的绘制。
- 如果行高不总是一样，缓存它们。
- 如果cell显示的内容来自网络，确保异步和缓存。
- 使用shadowPath来建立阴影。
- 减少子视图的数目。
- cellForRowAtIndexPath:中做尽量少的工作，如果需要做相同的工作，那么只做一次并缓存结果。
- 使用适当的数据结构存储你要的信息，不同的结构有对于不同的操作有不同的代价。
- 使用rowHeight，sectionFooterHeight，sectionHeaderHeight为常数，而不是询问代理。

21) 选择正确的数据存储方式

当要存储和读取大数据的时候你的选择是什么?

你有一些选项，包括:

- 使用 NSUserDefaults存储它们。
- 存储在结构化文件中，XML，JSON，Plist格式中。
- 是用NSCoding打包?
- 存储在本地数据库,如SQLite
- 使用NSData

NSUserDefaults有什么问题呢？虽然说NSUserDefaults是好而且简单，它确实很好只有当你有很少的数据要存(像你的等级，或者音量是开还是关)。一旦你接触大数据，会有更好的其他选择。

保存在结构化文件中也可能有问题。一般的，在解析之前，你需要加载整个文件到内存中，这是非常耗时的操作。你可以使用SAX去处理XML文件，但是那是一个复杂的作法。同时你加载了全部的对象进内存，其中有你想要的也有不想要的。

那么NSCoding怎么样呢?不幸的是，它也同样要读写文件,跟上面说的方法有同样的问题。

你最好的解决方法是使用SQLite或者 Core Data. 通过这些技术，你可以执行特定的查询只加载需要的对象,避免强力搜索方法来检索数据。性能方面，SQLite和Core Data 非常接近。

SQLite 和 Core Data最大的不同就是它们的使用方法。Core Data呈现为一个对象图模型，但是SQLite是一个传统的DBMS(数据库管理系统).通常Apple建议你用Core Data，但是除非你有特殊的原因不让你你会想避开它，使用更低级的SQLite。

如果在你的app中使用SQLite，一个方便的库 FMDB 允许你使用SQLite而不用专研SQLite的C API。

高级性能技巧

寻找一些精英的方式去成为十足的代码忍者？这些高级性能技巧可以合适的时候使用让你的app运行得尽可能的高效。

22）加速启动时间

App的启动时间非常重要，特别是第一次启动的时候。第一影响意味着太多了!

最大的事情是保证你的App开始尽量的快，尽量的多的执行异步任务，不如网络请求，数据库访问，或者数据解析。

尽量避免臃肿的XIBs，因为你在主线程中加载。但是在故事板中不会有这个问题，所以尽量用它们。

Note: 监察人不会运行你的app在Xcode调试中，所以确保测试启动性能时断开与Xcode的连接。

23）使用自动释放池

NSAutoreleasePool负责释放在代码块中的自动释放对象。通常，它是被UIKit自动调用的。但是也有一些场景我们需要手动创建NSAutoreleasePools。

举个例子，如果你创建太多的临时对象在你的代码中，你会注意到你的内存用量会增加直到对象被释放掉。问题是内存只有在UIKit排空

(drains)自动释放池的时候才能被释放，这意味着内存被占用的时间超过了需要。

好消息是你可以在你的@autoreleasepool段中创建临时对象来避免上述情况。代码如下所示。

```
NSArray *urls = <# An array of file URLs #>;
for (NSURL *url in urls) {

    @autoreleasepool {

        NSError *error;

        NSString *fileContents = [NSString
stringWithContentsOfURL:url

                                  encoding:NSUTF8StringEncod
ing error:&error];

        /* Process the string, creating and autoreleasing more
objects. */

    }

}
```

在每次迭代之后会自动释放所有的对象。

你可以阅读更多关于NSAutoreleasePool的内容Apple’s official documentation.

24）缓存图像

这里有两种方法去加载app束中的Image,第一个常见的方式是用imageNamed. 第二个是使用imageWithContentsOfFile

为什么会有两种方法,它们有效率吗?

imageNamed 在载入时有缓存的优势。文档 documentation for imageNamed是这样解释的:

这个方法看起来在系统缓存一个图像对象并指定名字，如果存在则返回对象，如果匹配图像的对象不在缓存中，这个方法会从指定的文件中加载数据，并缓存它，然后返回结果对象。

作为替代,imageWithContendsOfFile 简单的载入图像并不会缓存。

这两个方法的的演示片段如下:

```
UIImage *img = [UIImage imageNamed:@"myImage"]; // caching


// or
UIImage *img = [UIImage imageWithContentsOfFile:@"myImage"]; // no caching
```

如果你加载只使用一次大图片，那就不需要缓存。这种情况imageWithContendsOfFile会非常好,这种方式不会浪费内存来缓存图片。什么时候使用哪一种呢?

然而，imageNamed 对于要重用的图片来说是更好的选择，这种方法节约了经常的从磁盘加载图片的时间。

25) 尽可能避免日期格式化器

如果你要用NSDateFormatter来解析日期数据，你就得小心对待了。之前提到过，尽量的重用NSDateFormatters总是一个好的想法。

然而，如果你需要更快的速度，你可以使用C代替NSDateFormatter来解析日期。 Sam Soffes写了一篇 blog post about this topic来说明如何用代码来解析 ISO-8601日期串。尽管如此，你可以很容易的修改他的代码例子来适应你的特殊需求。

噢，听起来很棒，但是你相信有更好的办法吗?

如果你能控制你所处理日期的格式，尽可能的选择使用 Unix timestamps。Unix时间戳是简单的整数代表从某个起始时间点开始到现在的秒数。这个起始点通常是1970年1月1日 UTC 00:00:00。

你可以容易的把时间戳转换为NSDate，如下面所示:

```
1
- (NSDate*)dateFromUnixTimestamp:(NSTimeInterval)timestamp {
2

3
  return [NSDate dateWithTimeIntervalSince1970:timestamp];
4

5
}
```

这甚至比C函数更快

注意，很多WEB APIs返回时间戳是毫秒，因为这对于javascript最终来使用和处理数据是非常常见的。只要记住将这个时间戳除以1000再传递给dateFromUnixTimestamp方法即可。