性能优化面试题

在性能优化中一个最具参考价值的属性是 FPS:Frames Per Second,其实就是屏幕刷新率，苹果的 iphone 推荐的刷新率是 60Hz，也就是说 GPU 每秒钟刷新屏幕 60 次，这每刷新一次就是一帧 frame，FPS 也就是每秒钟刷新多少帧画面。静止不变的页面 FPS 值是 0，这个值是没有参考意义的，只有当页面在执行动画或者滑动的时候，FPS 值才具有参考价值，FPS 值的大小体现了页面的流畅程度高低，当低于 45 的时候卡顿会比较明显。

图层混合：

每一个 layer 是一个纹理，所有的纹理都以某种方式堆叠在彼此的顶部。对于屏幕上的每一个像素，GPU 需要算出怎么混合这些纹理来得到像素 RGB 的值。

当 Sa = 0.5 时，RGB 值为(0.5, 0, 0)，可以看出，当两个不是完全不透明的 CALayer 覆盖在一起时,GPU 大量做这种复合操作，随着这中操作的越多，GPU 越忙碌，性能肯定会受到影响。

公式：

R = S + D * (1 – Sa)

结果的颜色是源色彩(顶端纹理)+目标颜色(低一层的纹理)*(1-源颜色的透明度)。

当 Sa = 1 时，R = S,GPU 将不会做任何合成，而是简单从这个层拷贝，不需要考虑它下方的任何东西(因为都被它遮挡住了)，这节省了 GPU 相当大的工作量。

一、入门级

1、用 ARC 管理内存
2、在正确的地方使用 reuseIdentifier
3、尽量把 views 设置为透明
4、避免过于庞大的 XIB
5、不要阻塞主线程

6、在 ImageViews 中调整图片大小。如果要在 UIImageView 中显示一个来自 bundle 的图片，你应保证图片的大小和 UIImageView 的大小相同。在运行中缩放图片是很耗费资源的，特别是 UIImageView 嵌套在 UIScrollView 中的情况下。如果图片是从远端服务加载的你不能控制图片大小，比如在下载前调整到合适大小的话，你可以在下载完成后，最好是用 background
thread，缩放一次，然后在 UIImageView 中使用缩放后的图片。

7、选择正确的 Collection。

- Arrays: 有序的一组值。使用 index 来 lookup 很快，使用 value lookup 很慢， 插入/删除很慢。
- Dictionaries: 存储键值对。 用键来查找比较快。
- Sets: 无序的一组值。用值来查找很快，插入/删除很快。

8、打开 gzip 压缩。app 可能大量依赖于服务器资源，问题是我们的目标是移动设备，因此你就不能指望网络状况有多好。减小文档的一个方式就是在服务端和你的 app 中打开 gzip。这对于文字这种能有更高压缩率的数据来说会有更显著的效用。
iOS 已经在 NSURLConnection 中默认支持了 gzip 压缩，当然 AFNetworking 这些基于它的框架亦然。

二、中级

1、重用和延迟加载(lazy load) Views

- 更多的 view 意味着更多的渲染，也就是更多的 CPU 和内存消耗，对于那种嵌套了很多 view 在 UIScrollView 里边的 app 更是如此。
- 这里我们用到的技巧就是模仿 UITableView 和 UICollectionView 的操作：不要一次创建所有的 subview，而是当需要时才创建，当它们完成了使命，把他们放进一个可重用的队列中。这样的话你就只需要在滚动发生时创建你的 views，避免了不划算的内存分配。

2、Cache, Cache, 还是 Cache!

- 一个极好的原则就是，缓存所需要的，也就是那些不大可能改变但是需要经常读取的东西。
- 我们能缓存些什么呢？一些选项是，远端服务器的响应，图片，甚至计算结果，比如 UITableView 的行高。
- NSCache 和 NSDictionary 类似，不同的是系统回收内存的时候它会自动删掉它的内容。

3、权衡渲染方法.性能能还是要 bundle 保持合适的大小。

4、处理内存警告.移除对缓存，图片 object 和其他一些可以重创建的 objects 的 strong references.

5、重用大开销对象

6、一些 objects 的初始化很慢，比如 NSDateFormatter 和 NSCalendar。然而，你又不可避免地需要使用它们，比如从 JSON 或者 XML 中解析数据。想要避免使用这个对象的瓶颈你就需要重用他们，可以通过添加属性到你的 class 里或者创建静态变量来实现。

7、避免反复处理数据.在服务器端和客户端使用相同的数据结构很重要。

8、选择正确的数据格式.解析 JSON 会比 XML 更快一些，JSON 也通常更小更便于传输。从 iOS5 起有了官方内建的 JSON deserialization 就更加方便使用了。但是 XML 也有 XML 的好处，比如使用 SAX 来解析 XML 就像解析本地文件一样，你不需像解析 json 一样等到整个文档下载完成才开始解析。当你处理很大的数据的时候就会极大地减低内存消耗和增加性能。

9、正确设定背景图片

- 全屏背景图，在 view 中添加一个 UIImageView 作为一个子 View
- 只是某个小的 view 的背景图，你就需要用 UIColor 的 colorWithPatternImage 来做了，它会更快地渲染也不会花费很多内存：

10、减少使用 Web 特性。想要更高的性能你就要调整下你的 HTML 了。第一件要做的事就是尽可能移除不必要的 javascript，避免使用过大的框架。能只用原生 js 就更好了。尽可能异步加载例如用户行为统计 script 这种不影响页面表达的 javascript。注意你使用的图片，保证图片的符合你使用的大小。

11、Shadow Path 。CoreAnimation 不得不先在后台得出你的图形并加好阴影然后才渲染，这开销是很大的。使用 shadowPath 的话就避免了这个问题。使用 shadow path 的话 iOS 就不必每次都计算如何渲染，它使用一个预先计算好的路径。但问题是自己计算 path 的话可能在某些 View 中比较困难，且每当 view 的 frame 变化的时候你都需要去 update shadow path.

12、优化 Table View

- 正确使用 reuseIdentifier 来重用 cells
- 尽量使所有的 view opaque，包括 cell 自身
- 避免渐变，图片缩放，后台选人
- 缓存行高
- 如果 cell 内现实的内容来自 web，使用异步加载，缓存请求结果
- 使用 shadowPath 来画阴影
- 减少 subviews 的数量
- 尽量不适用 cellForRowAtIndexPath:，如果你需要用到它，只用-一次然后缓存结果
- 使用正确的数据结构来存储数据
- 使用 rowHeight, sectionFooterHeight 和 sectionHeaderHeight 来设定固定的高，不要请求 delegate

13、选择正确的数据存储选项

- NSUserDefaults 的问题是什么？虽然它很 nice 也很便捷，但是它只适用于小数据，比如一些简单的布尔型的设置选项，再大点你就要考虑其它方式了
- XML 这种结构化档案呢？总体来说，你需要读取整个文件到内存里去解析，这样是很不经济的。使用 SAX 又是一个很麻烦的事情。
- NSCoding？不幸的是，它也需要读写文件，所以也有以上问题。
- 在这种应用场景下，使用 SQLite 或者 Core Data 比较好。使用这些技术你用特定的查询语句就能只加载你需要的对象。
- 在性能层面来讲，SQLite 和 Core Data 是很相似的。他们的不同在于具体使用方法。
- Core Data 代表一个对象的 graph model，但 SQLite 就是一个 DBMS。
- Apple 在一般情况下建议使用 Core Data，但是如果你有理由不使用它，那么就去使用更加底层的 SQLite 吧。
- 如果你使用 SQLite，你可以用 FMDB 这个库来简化 SQLite 的操作，这样你就不用花很多经历了解 SQLite 的 C API 了。

三、高级

1、加速启动时间。快速打开 app 是很重要的，特别是用户第一次打开它时，对 app 来讲，第一印象太太太重要了。你能做的就是使它尽可能做更多的异步任务，比如加载远端或者数据库数据，解析数据。避免过于庞大的 XIB，因为他们是在主线程上加载的。所以尽量使用没有这个问题的 Storyboards 吧！一定要把设备从 Xcode 断开来测试启动速度

2、使用 Autorelease Pool。NSAutoreleasePool`负责释放 block 中的 autoreleased objects。一般情况下它会自动被 UIKit 调用。但是有些状况下你也需要手动去创建它。假如你创建很多临时对象，你会发现内存一直在减少直到这些对象被 release 的时候。这是因为只有当 UIKit 用光了 autorelease pool 的时候 memory 才会被释放。消息是你可以在你自己的@autoreleasepool 里创建临时的对象来避免这个行为。

3、选择是否缓存图片。常见的从 bundle 中加载图片的方式有两种，一个是用 imageNamed，二是用 imageWithContentsOfFile，第一种比较常见一点。

4、避免日期格式转换。如果你要用 NSDateFormatter 来处理很多日期格式，应该小心以待。就像先前提到的，任何时候重用 NSDateFormatters 都是一个好的实践。如果你可以控制你所处理的日期格式，尽量选择 Unix 时间戳。你可以方便地从时间戳转换到 NSDate:

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

这样会比用 C 来解析日期字符串还快！需要注意的是，许多 web API 会以微秒的形式返回时间戳，因为这种格式在 javascript 中更方便使用。记住用 dateFromUnixTimestamp 之前除以 1000 就好了。

平时你是如何对代码进行性能优化的？

- 利用性能分析工具检测，包括静态 Analyze 工具，以及运行时 Profile 工具，通过 Xcode 工具栏中 Product->Profile 可以启动,
- 比如测试程序启动运行时间，当点击 Time Profiler 应用程序开始运行后.就能获取到整个应用程序运行消耗时间分布和百分比.为了保证数据分析在统一使用场景真实需要注意一定要使用真机,因为此时模拟器是运行在 Mac 上，而 Mac 上的 CPU 往往比 iOS 设备要快。
- 为了防止一个应用占用过多的系统资源，开发 iOS 的苹果工程师门设计了一个“看门狗”的机制。在不同的场景下，“看门狗”会监测应用的性能。如果超出了该场景所规定的运行时间，“看门狗”就会强制终结这个应用的进程。开发者们在 crashlog 里面，会看到诸如 0x8badf00d 这样的错误代码。

优化 Table View

- 正确使用 reuseIdentifier 来重用 cells
- 尽量使所有的 view opaque，包括 cell 自身
- 如果 cell 内现实的内容来自 web，使用异步加载，缓存请求结果
  减少 subviews 的数量
- 尽量不适用 cellForRowAtIndexPath:，如果你需要用到它，只用一次然后缓存结果
- 使用 rowHeight, sectionFooterHeight 和 sectionHeaderHeight 来设定固定的高，不要请求 delegate

UIImage 加载图片性能问题

- imagedNamed 初始化
- imageWithContentsOfFile 初始化
- imageNamed 默认加载图片成功后会内存中缓存图片,这个方法用一个指定的名字在系统缓存中查找并返回一个图片对象.如果缓存中没有找到相应的图片对象,则从指定地方加载图片然后缓存对象，并返回这个图片对象.
- imageWithContentsOfFile 则仅只加载图片,不缓存.
- 加载一张大图并且使用一次，用 imageWithContentsOfFile 是最好,这样 CPU 不需要做缓存节约时间.
- 使用场景需要编程时，应该根据实际应用场景加以区分，UIimage 虽小，但使用元素较多问题会有所凸显.
    - 不要在 viewWillAppear 中做费时的操作：viewWillAppear: 在 view 显示之前被调用，出于效率考虑，方法中不要处理复杂费时操作；在该方法设置 view 的显示属性之类的简单事情，比如背景色，字体等。否则，会明显感觉到 view 有卡顿或者延迟。
    - 在正确的地方使用 reuseIdentifier：table view 用 tableView:cellForRowAtIndexPath:为 rows 分配 cells 的时候，它的数据应该重用自 UITableViewCell。
    - 尽量把 views 设置为透明：如果你有透明的 Views 你应该设置它们的 opaque 属性为 YES。系统用一个最优的方式渲染这些 views。这个简单的属性在 IB 或者代码里都可以设定。
    - 避免过于庞大的 XIB：尽量简单的为每个 Controller 配置一个单独的 XIB，尽可能把一个 View Controller 的 view 层次结构分散到单独的 XIB 中去, 当你加载一个引用了图片或者声音资源的 nib 时，nib 加载代码会把图片和声音文件写进内存。
    - 不要阻塞主线程：永远不要使主线程承担过多。因为 UIKit 在主线程上做所有工作，渲染，管理触摸反应，回应输入等都需要在它上面完成,大部分阻碍主进程的情形是你的 app 在做一些牵涉到读写外部资源的 I/O 操作，比如存储或者网络。

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,  0), ^{
// 选择一个子线程来执行耗时操作
dispatch_async(dispatch_get_main_queue(), ^{
// 返回主线程更新UI
});
});
```

    - 在 Image Views 中调整图片大小

    如果要在 UIImageView 中显示一个来自 bundle 的图片，你应保证图片的大小和 UIImageView 的大小相同。在运行中缩放图片是很耗费资源的.

讲讲你用 Instrument 优化动画性能的经历吧（别问我什么是 Instrument）

Apple的instrument为开发者提供了各种template去优化app性能和定位问题。很多公司都在赶feature，并没有充足的时间来做优化，导致不少开发者对instrument不怎么熟悉，但这里面其实涵盖了非常完整的计算机基础理论知识体系，memory，disk，network，thread，cpu，gpu等等，顺藤摸瓜去学习，是一笔巨大的知识财富。
动画性能只是其中一个template，重点还是理解上面问题当中CPU GPU如何配合工作的知识。

facebook 启动时间优化

1. 瘦身请求依赖
2.UDP 启动请求先行缓存
3.队列串行化处理启动响应

四、光栅化

光栅化是将几何数据经过一系列变换后最终转换为像素，从而呈现在显示设备上的过程，光栅化的本质是坐标变换、几何离散化

我们使用 UITableView 和 UICollectionView 时经常会遇到各个 Cell 的样式是一样的，这时候我们可以使用这个属性提高性能：

```
cell.layer.shouldRasterize=YES;

cell.layer.rasterizationScale=[[UIScreenmainScreen]scale];
```

五、日常如何检查内存泄露？

目前我知道的方式有以下几种

- Memory Leaks
- Alloctions
- Analyse
- Debug Memory Graph
- MLeaksFinder

泄露的内存主要有以下两种：

- `Lak Memory` 这种是忘记 `Release` 操作所泄露的内存。
- `Abandon Memory` 这种是循环引用，无法释放掉的内存。

上面所说的五种方式，其实前四种都比较麻烦，需要不断地调试运行，第五种是腾讯阅读团队出品，效果好一些

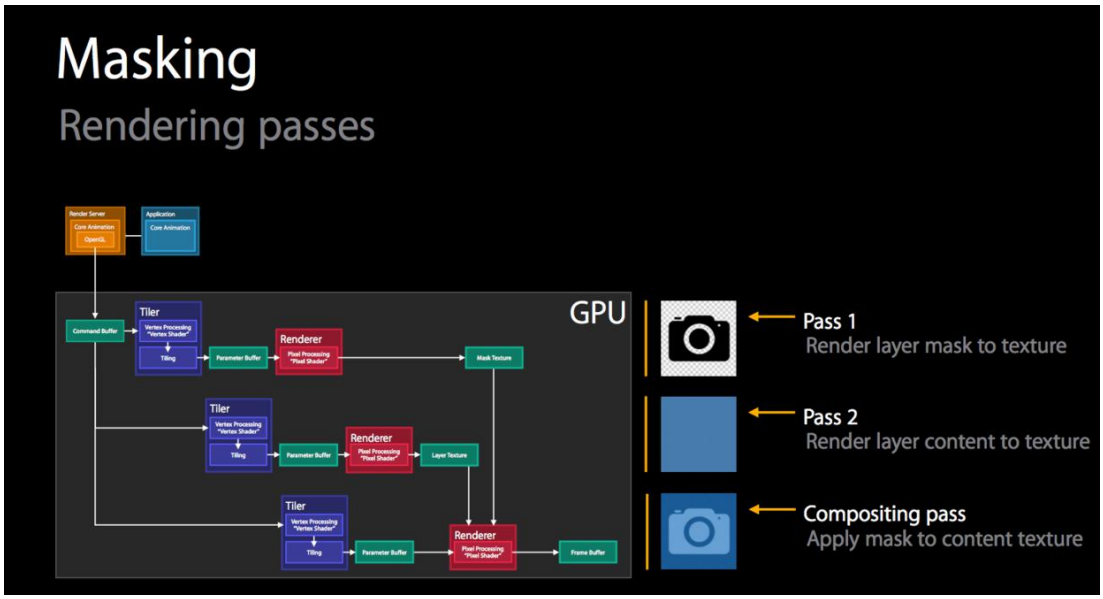
六、如何高性能的画一个圆角？

视图和圆角的大小对帧率并没有什么卵影响，数量才是伤害的核心输出

```
label.layer.cornerRadius = 5
label.layer.masksToBounds = true
```

首先上面的方式是不可取的，会触发离屏渲染。

- 如果能够只用 `cornerRadius` 解决问题，就不用优化。
- 如果必须设置 `masksToBounds`，可以参考圆角视图的数量，如果数量较少（一页只有几个）也可以考虑不用优化。
- `UIImageView` 的圆角通过直接截取图片实现，其它视图的圆角可以通过 `Core Graphics` 画出圆角矩形实现。



七、如何提升 tableview 的流畅度？

本质上是降低 CPU、GPU 的工作，从这两个大的方面去提升性能。

- CPU：对象的创建和销毁、对象属性的调整、布局计算、文本的计算和排版、图片的格式转换和解码、图像的绘制
- GPU：纹理的渲染

卡顿优化在 CPU 层面

- 尽量用轻量级的对象，比如用不到事件处理的地方，可以考虑使用 CALayer 取代 UIView
- 不要频繁地调用 UIView 的相关属性，比如 frame、bounds、transform 等属性，尽量减少不必要的修改
- 尽量提前计算好布局，在有需要时一次性调整对应的属性，不要多次修改属性
- Autolayout 会比直接设置 frame 消耗更多的 CPU 资源
- 图片的 size 最好刚好跟 UIImageView 的 size 保持一致
- 控制一下线程的最大并发数量
- 尽量把耗时的操作放到子线程
    - 文本处理（尺寸计算、绘制）
    - 图片处理（解码、绘制）

卡顿优化在 GPU 层面

- 尽量避免短时间内大量图片的显示，尽可能将多张图片合成一张进行显示
- GPU 能处理的最大纹理尺寸是 4096x4096，一旦超过这个尺寸，就会占用 CPU 资源进行处理，所以纹理尽量不要超过这个尺寸
- 尽量减少视图数量和层次
- 减少透明的视图（alpha<1），不透明的就设置 opaque 为 YES
- 尽量避免出现离屏渲染

1. 预排版，提前计算

在接收到服务端返回的数据后，尽量将 CoreText 排版的结果、单个控件的高度、cell 整体的高度提前计算好，将其存储在模型的属性中。需要使用时，直接从模型中往外取，避免了计算的过程。

尽量少用 UILabel，可以使用 CALayer 。避免使用 AutoLayout 的自动布局技术，采取纯代码的方式

2. 预渲染，提前绘制

例如圆形的图标可以提前在，在接收到网络返回数据时，在后台线程进行处理，直接存储在模型数据里，回到主线程后直接调用就可以了

避免使用 CALayer 的 Border、corner、shadow、mask 等技术，这些都会触发离屏渲染。

3. 异步绘制

4. 全局并发线程

5. 高效的图片异步加载

八、如何优化 APP 的电量？

程序的耗电主要在以下四个方面：

- CPU 处理
- 定位
- 网络
- 图像

优化的途径主要体现在以下几个方面：

- 尽可能降低 CPU、GPU 的功耗。
- 尽量少用 定时器。
- 优化 I/O 操作。
    - 不要频繁写入小数据，而是积攒到一定数量再写入
    - 读写大量的数据可以使用 Dispatch_io ，GCD 内部已经做了优化。
    - 数据量比较大时，建议使用数据库
- 网络方面的优化
    - 减少压缩网络数据 （XML -> JSON -> ProtoBuf），如果可能建议使用 ProtoBuf。
    - 如果请求的返回数据相同，可以使用 NSCache 进行缓存
    - 使用断点续传，避免因网络失败后要重新下载。
    - 网络不可用的时候，不尝试进行网络请求
    - 长时间的网络请求，要提供可以取消的操作
    - 采取批量传输。下载视频流的时候，尽量一大块一大块的进行下载，广告可以一次下载多个
- 定位层面的优化
    - 如果只是需要快速确定用户位置，最好用 CLLocationManager 的 requestLocation 方法。定位完成后，会自动让定位硬件断电
    - 如果不是导航应用，尽量不要实时更新位置，定位完毕就关掉定位服务
    - 尽量降低定位精度，比如尽量不要使用精度最高的 kCLLocationAccuracyBest
    - 需要后台定位时，尽量设置 pausesLocationUpdatesAutomatically 为 YES，如果用户不太可能移动的时候系统会自动暂停位置更新
    - 尽量不要使用 startMonitoringSignificantLocationChanges，优先考虑 startMonitoringForRegion:
- 硬件检测优化

- 用户移动、摇晃、倾斜设备时，会产生动作(motion)事件，这些事件由加速度计、陀螺仪、磁力计等硬件检测。在不需要检测的场合，应该及时关闭这些硬件

九、如何有效降低 APP 包的大小？

降低包大小需要从两方面着手

可执行文件

- 编译器优化
  - Strip Linked Product、Make Strings Read-Only、Symbols Hidden by Default 设置为 YES
  - 去掉异常支持，Enable C++ Exceptions、Enable Objective-C Exceptions 设置为 NO， Other C Flags 添加 -fno-exceptions
- 利用 AppCode 检测未使用的代码：菜单栏 -> Code -> Inspect Code
- 编写 LLVM 插件检测出重复代码、未被调用的代码

资源

资源包括 图片、音频、视频 等

- 优化的方式可以对资源进行无损的压缩
- 去除没有用到的资源

十、什么是 离屏渲染？什么情况下会触发？该如何应对？

离屏渲染就是在当前屏幕缓冲区以外，新开辟一个缓冲区进行操作。

离屏渲染出发的场景有以下：

- 圆角 （maskToBounds 并用才会触发）
- 图层蒙版
- 阴影
- 光栅化

为什么要避免离屏渲染？

CPU GPU 在绘制渲染视图时做了大量的工作。离屏渲染发生在 GPU 层面上，会创建新的渲染缓冲区，会触发 OpenGL 的多通道渲染管线，图形上下文的切换会造成额外的开销，增加 GPU 工作量。如果 CPU GPU 累计耗时 16.67 毫秒还没有完成，就会造成卡顿掉帧。

圆角属性、蒙层遮罩 都会触发离屏渲染。指定了以上属性，标记了它在新的图形上下文中，在未愈合之前，不可以用于显示的时候就出发了离屏渲染。

- 在 OpenGL 中，GPU 有 2 种渲染方式
  - On-Screen Rendering：当前屏幕渲染，在当前用于显示的屏幕缓冲区进行渲染操作
  - Off-Screen Rendering：离屏渲染，在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作
- 离屏渲染消耗性能的原因
  - 需要创建新的缓冲区
  - 离屏渲染的整个过程，需要多次切换上下文环境，先是从当前屏幕（On-Screen）切换到离屏（Off-Screen）；等到离屏渲染结束以后，将离屏缓冲区的渲染结果显示到屏幕上，又需要将上下文环境从离屏切换到当前屏幕
- 哪些操作会触发离屏渲染？
  - 光栅化，layer.shouldRasterize = YES
  - 遮罩，layer.mask
  - 圆角，同时设置 layer.masksToBounds = YES、layer.cornerRadius 大于 0
  - 考虑通过 CoreGraphics 绘制裁剪圆角，或者叫美工提供圆角图片
  - 阴影，layer.shadowXXX，如果设置了 layer.shadowPath 就不会产生离屏渲染

十一、如何检测离屏渲染？

1、模拟器 debug-选中 color Offscreen - Renderd 离屏渲染的图层高亮成黄 可能存在性能问题

2、真机 Instrument-选中 Core Animation-勾选 Color Offscreen-Rendered Yellow

离屏渲染的触发方式

设置了以下属性时，都会触发离屏绘制：

1、layer.shouldRasterize（光栅化）

光栅化概念：将图转化为一个个栅格组成的图象。

光栅化特点：每个元素对应帧缓冲区中的一像素。

2、masks（遮罩）

3、shadows（阴影）

4、edge antialiasing（抗锯齿）

5、group opacity（不透明）

6、复杂形状设置圆角等

7、渐变

8、drawRect

例如我们日程经常打交道的 TableViewCell,因为 TableViewCell 的重绘是很频繁的（因为 Cell 的复用）,如果 Cell 的内容不断变化,则 Cell 需要不断重绘,如果此时设置了 cell.layer 可光栅化。则会造成大量的离屏渲染,降低图形性能。

如果将不在 GPU 的当前屏幕缓冲区中进行的渲染都称为离屏渲染，那么就还有另一种特殊的“离屏渲染”方式：CPU 渲染。如果我们重写了 drawRect 方法，并且使用任何 Core Graphics 的技术进行了绘制操作，就涉及到了 CPU 渲染。整个渲染过程由 CPU 在 App 内同步地完成，渲染得到的 bitmap 最后再交由 GPU 用于显示。

现在摆在我们面前得有三个选择：当前屏幕渲染、离屏渲染、CPU 渲染，该用哪个呢？这需要根据具体的使用场景来决定。

尽量使用当前屏幕渲染

鉴于离屏渲染、CPU 渲染可能带来的性能问题，一般情况下，我们要尽量使用当前屏幕渲染。

离屏渲染 VS CPU 渲染

由于 GPU 的浮点运算能力比 CPU 强，CPU 渲染的效率可能不如离屏渲染；但如果仅仅是实现一个简单的效果，直接使用 CPU 渲染的效率又可能比离屏渲染好，毕竟离屏渲染要涉及到缓冲区创建和上下文切换等耗时操作

UIButton 的 masksToBounds = YES 又设置 setImage、setBackgroundImage、[button setBackgroundColor:[UIColor colorWithPatternImage:[UIImage imageNamed:@"btn_selected"]]];

下发生离屏渲染，但是[button setBackgroundColor:[UIColor redColor]];是不会出现离屏渲染的

关于 UIImageView,现在测试发现(现版本: iOS10),在性能的范围之内,给 UIImageView 设置圆角是不会触发离屏渲染的,但是同时给 UIImageView 设置背景色则肯定会触发.触发离屏渲染跟 png.jpg 格式并无关联

日常我们使用 layer 的两个属性，实现圆角

imageView.layer.cornerRaidus = CGFloat(10);

imageView.layer.masksToBounds = YES;

这样处理的渲染机制是 GPU 在当前屏幕缓冲区外新开辟一个渲染缓冲区进行工作，也就是离屏渲染，这会给我们带来额外的性能损耗。如果这样的圆角操作达到一定数量，会触发缓冲区的频繁合并和上下文的的频繁切换，性能的代价会宏观地表现在用户体验上——掉帧

十二、怎么检测图层混合？

1、模拟器 debug- 选中 color blended layers 红色区域表示图层发生了混合

2、Instrument-选中 Core Animation-勾选 Color Blended Layers

避免图层混合：

1、确保控件的 opaque 属性设置为 true，确保 backgroundColor 和父视图颜色一致且不透明

2、如无特殊需要，不要设置低于 1 的 alpha 值

3、确保 UIImage 没有 alpha 通道

UILabel 图层混合解决方法：

iOS8 以后设置背景色为非透明色并且设置 label.layer.masksToBounds=YES 让 label 只会渲染她的实际 size 区域，就能解决 UILabel 的图层混合问题

iOS8 之前只要设置背景色为非透明的就行

为什么设置了背景色但是在 iOS8 上仍然出现了图层混合呢？

UILabel 在 iOS8 前后的变化，在 iOS8 以前，UILabel 使用的是 CALayer 作为底图层，而在 iOS8 开始，UILabel 的底图层变成了_UILabelLayer，绘制文本也有所改变。在背景色的四周多了一圈透明的边，而这一圈透明的边明显超出了图层的矩形区域，设置图层的 masksToBounds 为 YES 时，图层将会沿着 Bounds 进行裁剪 图层混合问题解决了