关于一些 iOS 面试问题的解答

2015年04月26日

这篇 post 主要是对知乎上 iOS程序员的问题列表 的回答, 也算是对自己已有的知识进行整理.

如果你对本篇 post 中的回答有所疑问, 可以在下面留言. 如果有问题, 我一定会修改的 :-)

问题以及回答

1. 什么是 ARC? (ARC 是为了解决什么问题而诞生的?)

ARC 是 Automatic Reference Counting 的缩写, 即自动引用计数. 这是苹果在 iOS5 中引入的内存管理机制. Objective-C 和 Swift 使用 ARC 追踪和管理应用的内存使用. 这一机制使得开发者无需键入 `retain` 和 `release`, 这不仅能够降低程序崩溃和内存泄露的风险, 而且可以减少开发者的工作量, 能够大幅度提升程序的**流畅性**和**可预测性**. 但是 ARC 不适用于 Core Foundation 框架中, 仍然需要手动管理内存.

---

2. 以下 keywords 有什么区别: `assign vs`

`weak`, `__block vs __weak`

`assign` 和 `weak` 是用于在声明属性时, 为属性指定内存管理的语义.

- `assign` 用于简单的赋值, 不改变属性的引用计数, 用于 Objective-C 中的 `NSInteger`, `CGFloat` 以及 C 语言中 `int`, `float`, `double` 等数据类型.
- `weak` 用于对象类型, 由于 `weak` 同样不改变对象的引用计数且不持有对象实例, 当该对象废弃时, 该弱引用自动失效并且被赋值为 `nil`, 所以它可以用于避免两个强引用产生的**循环引用**导致内存无法释放的问题.

`__block` 和 `__weak` 之间的却是确实极大的, 不过它们都用于修饰变量.

- 前者用于指明当前声明的变量在被 block 捕获之后, 可以在 block 中改变变量的值. 因为在 block 声明的同时会截获该 block 所使用的全部自动变量的值, 而这些值只在 block 中**只具有"使用权"而不具有"修改权"**. 而 `__block` 说明符就为 block 提供了变量的修改权.
- 后者是**所有权修饰符**, 什么是所有权修饰符? 这里涉及到另一个问题, 因为在 ARC 有效时, id 类型和对象类型同 C 语言中的其他类型不同, 必须附加所有权修饰符. 所有权修饰符一种有 4 种:
  - __strong
  - __weak
  - __unsafe_unretained
  - __autorelease
- `__weak` 与 `weak` 的区别只在于, 前者用于变量的声明, 而后者用于属性的声明.

---

3. `__block` 在 ARC 和非 ARC 下含义一样吗?

`__block` **在 ARC 下捕获的变量会被 block retain**, 这样可能导致循环引用, 所以必须要使用弱引用才能解决该问题. 而在非 ARC 下, 可以直接使用 `__block` 说明符修饰变量, 因为在非 ARC 下, block 不会 retain 捕获的变量.

---

4. 使用 `nonatomic` 一定是线程安全的吗?

`nonatomic` 的内存管理语义是**非原子**的, 非原子的操作本来就是线程不安全的, 而 `atomic` 的操作是原子的, 但是**并不意味着它是线程安全的**, 它会增加正确的几率, 能够更好的避免线程的错误, 但是它仍然是线程不安全的.

当使用 `nonatomic` 的时候, 属性的 setter 和 getter 操作是非原子的, 所以当多个线程同时对某一属性进行读和写的操作, 属性的最终结果是不能预测的.

当使用 `atomic` 时, 虽然对属性的读和写是原子的, 但是仍然可能出现线程错误: 当线程 A 进行写操作, 这时其他线程的读或写操作会因为该操作的进行而等待. 当 A 线程的写操作结束后, B 线程进行写操作, 然后当 A 线程进行读操作时, 却获得了在 B 线程中的值, 这就破坏了线程安全, 如果有线程 C 在 A 线程读操作前 release 了该属性, 那么还会导致程序崩溃. 所以仅仅使用 `atomic` 并不会使得线程安全, 我们还需要为线程添加 `lock` 来确保线程的安全.

`atomic` 都不是一定线程安全的, `nonatomic` 就更不必多说了.

5. ~~描述一个你遇到过的 retain cycle 例子.~~

6. `+ (void)load;` 和 `+ (void)initialize;` 有什么用处?

当类对象被引入项目时, runtime 会向每一个类对象发送 `load` 消息. `load` 方法还是非常的神奇的, 因为它会在**每一个类甚至分类**被引入时仅调用一次, 调用的顺序是父类优先于子类, 子类优先于分类. 而且 `load` 方法不会被类自动继承, 每一个类中的 `load` 方法都不需要像 `viewDidLoad` 方法一样调用父类的方法. 由于 `load` 方法会在类被 `import` 时调用一次, 而这时往往是改变类的行为的最佳时机. 我在 DKNightVersion 中使用 `method swizlling` 来修改原有的方法时, 就是在分类 `load` 中实现的.

`initialize` 方法和 `load` 方法有一些不同, 它虽然也会在整个 runtime 过程中调用一次, 但是它是在**该类的第一个方法执行之前**调用, 也就是说 `initialize` 的调用是**惰性**的, 它的实现也与我们在平时使用的惰性初始化属性时基本相同. 我在实际的项目中并没有遇到过必须使用这个方法的情况, 在该方法中主要做**静态变量的设置**并用于**确保在实例初始化前某些条件必须满足**.

7. 为什么其他语言里叫函数调用, Objective-C 中是给对象发送消息 (谈下对

runtime 的理解)

我们在其他语言中比如说: C, Python, Java, C++, Haskell ... 中提到函数调用或者方法调用(面向对象). 函数调用是在编译期就已经决定了会调用哪个函数(方法), 编译器在编译期就能检查出函数的执行是否正确.

然而 Objective-C(ObjC) 是一门动态的语言, 整个 ObjC 语言都是尽可能的将所有的工作推迟到运行时才决定. 它基于 runtime 来工作, runtime 就是 ObjC 的灵魂, 其核心就是消息发送`objc_msgSend`.

> *What makes Objective-C truly powerful is its runtime.*

所有的消息都会在运行时才会确定, `[obj message]` 在运行时会被转化为 `objc_msgSend(id self, SEL cmd, ...)` 来执行, 它会在运行时从**选择子表中寻找对应的选择子**并将选择子与实现进行绑定. 而如果没有找到对应的实现, 就会进入类似黑魔法的消息转发流程. 调用 `+(BOOL)resolveInstanceMethod:(SEL)aSelector` 方法, 我们可以在这个方法中**为类动态地生成方法**.

我们几乎可以使用 runtime 魔改 Objective-C 中的一切: `class` `property` `object` `ivar` `method` `protocol`, 而下面就是它的主要应用:

- 内省
- 为分类动态的添加属性
- 使用方法调剂修改原有的方法实现
- ...

---

8. 什么是 Method Swizzling?

`method swizzling` 实际上就是一种在运行时动态修改原有方法实现的技术, 它实际上是基于 ObjC runtime 的特性, 而 `method swizzling` 的核心方法就是 `method_exchangeImplementations(SEL origin, SEL swizzle)`. 使用这个方法就可以在运行时动态地改变原有的方法实现, 在 DKNigtVersion(为 iOS 应用添加夜间模式) 中能够看到大量 `method swizzling` 的使用, 方法的调用时机就是在上面提到的 `load` 方法中, 不在 `initialize` 方法中改变方法实现的原因是 **`initialize` 可能会被子类所继承并重新执行最终导致无限递归**, 而 `load` 并不会被继承.

---

9. `UIView` 和 `CALayer` 有什么关系?

看到这个问题不禁想到大一在网易面试时的经历, 当时的两位面试官就问了我这么一个问题, `UIView`和 `CALayer` 是什么关系, 为什么要这么设计? 我已经忘记了当时是怎么回答的. 隐约记得当时说每一个 `UIView` 都会对应一个 `CALayer` 至于为什么这样, 当时的我实在是太弱无法回答出来了.

每一个 `UIView` 的身后对应一个 `Core Animation` 框架中的 `CALayer`. 每一个 `UIView` 都是 `CALayer` 的代理.

> *Many of the methods you call on UIView simply delegate to the layer*

在 iOS 上当你处理一个又一个的 `UIView` 时, 实际上是在操作 `CALayer`. 尽管有的时候你并不知道 (直接操作 `CALayer` 并不会对效率有着显著的提升).

`UIView` 实际上就是对 `CALayer` 的轻量级的封装. `UIView` 继承自 `UIResponder`, 用来处理来自用户的事件; `CALayer` 继承自 `NSObject` 主要用于处理图层的渲染和动画. 这么设计有以下几个原因:

- 你可以通过操作 `UIView` 在一个更高的层级上处理与用户的交互, 触摸, 点击, 拖拽等事件, 这些都是在 `UIKit` 这个层级上完成的.
- `UIView` 和 `NSView(AppKit)` 的实现极其不同, 而使用 `Core Animation` 可以实现底层代码地重用, 在 Mac 和 iOS 平台上都使用着近乎相同的 `Core Animation` 代码, 这样我们可以对这个层级进行抽象在两种平台上产生 `UIKit` 和 `AppKit` 用于不同平台的框架.

使用 `CALayer` 的唯一原因大概是便于移植到不同的平台, 如果仅仅使用 `Core Animation` 层级进行开发, 处理用户的交互时间需要写更多的代码.

10. 如何高性能的给 `UIImageView` 加个圆角? (不准说 `layer.cornerRadius`!)

一般情况下给 UIImageView 或者说 UIKit 的控件添加圆角都是改变 `clipsToBounds` 和 `layer.cornerRadius`, 这样大约两行代码就可以解决这个问题. 但是, 这样使用这样的方法会**强制 Core Animation 提前渲染屏幕的离屏绘制**, 而离屏绘制就会为性能带来负面影响.

我们也可以使用另一种比较复杂的方式来为图片添加圆角, 这里就用到了贝塞尔曲线.

```objectivec
UIImageView *imageView = [[UIImageView alloc] initWithFrame
imageView.center = CGPointMake(200, 300);
UIImage *anotherImage = [UIImage imageNamed:@"image"];
UIGraphicsBeginImageContextWithOptions(imageView.bounds.siz
[[UIBezierPath bezierPathWithRoundedRect:imageView.bounds
                                cornerRadius:50] addClip];
[anotherImage drawInRect:imageView.bounds];
imageView.image = UIGraphicsGetImageFromCurrentImageContext
UIGraphicsEndImageContext();
[self.view addSubview:imageView];
```

在这里使用了贝塞尔曲线"切割"个这个图片, 给 `UIImageView` 添加了的圆角.

---

11. 使用 `drawRect:` 有什么影响?

这个问题对于我来说确实有些难以回答, 我记得我在我人生的第一个 iOS 项目 SportJoin 中曾经使用这个方法来绘制图形, 但是具体怎么做的, 我已经忘记了.

这个方法的主要作用是根据传入的 `rect` 来绘制图像 参见文档. 这个方法的默认实现没有做任何事情, 我们**可以**在这个方法中使用 `Core Graphics` 和 `UIKit` 来绘制视图的内容.

这个方法的调用机制也是非常特别. 当你调用 `setNeedsDisplay` 方法时, UIKit 将会把当前图层标记为 dirty, 但**还是会显示原来的内容**, 直到下一次的视图渲染周期, 才会为标记为 dirty 的图层重新建立 Core Graphics 上下文, 然后将内存中的数据恢复出来, 再使用 CGContextRef 进行绘制.

12. ASIHttpRequest 或者 SDWebImage 里面给 UIImageView 加载图片的逻辑是什么样的?

我曾经阅读过 SDWebImage 的源代码, 就在这里对如何给 UIImageView 加载图片做一个总结吧, SDWebImage 中为 UIView 提供了一个分类叫做 WebCache, 这个分类中有一个最常用的接口, `sd_setImageWithURL:placeholderImage:`, 这个分类同时提供了很多类似的方法, 这些方法最终会调用一个同时具有 `option` `progressBlock` `completionBlock` 的方法, 而在这个类最终被调用的方法首先会检查是否传入了 `placeholderImage` 以及对应的参数, 并设置 `placeholderImage`.

然后会获取 `SDWebImageManager` 中的单例调用一个 `downloadImageWithURL:...` 的方法来获取图片, 而这个 manager 获取图片的过程有大体上分为两部分, 它首先会在 `SDWebImageCache` 中寻找图片是否有对应的缓存, 它会以 url 作为数据的索引先在内存中寻找是否有对应的缓存, 如果缓存未命中就会在磁盘中利用 MD5 处理过的 key 来继续查询对应的数据, 如果找到了, 就会把磁盘中的缓存备份到内存中.

然而, 假设我们在内存和磁盘缓存中都没有命中, 那么 manager 就会调用它持有的一个 `SDWebImageDownloader` 对象的方法 `downloadImageWithURL:...` 来下载图片, 这个方法会在执行的过程中调用另一个方法 `addProgressCallback:andCompletedBlock:fotURL:createCallback:` 来存储下载过程中和下载完成的回调, 当回调块是第一次添加的时候, 方法会实例化一个 `NSMutableURLRequest` 和 `SDWebImageDownloaderOperation`, 并将后者加入 downloader 持有的下载队列开始图片的异步下载.

而在图片下载完成之后, 就会在主线程设置 image, 完成整个图像的异步下载和配置.

---

13. 设计一个简单的图片内存缓存器 (包含移除策略)

待我阅读完 path 开源的 FastImageCache的源代码就来回答.

---

14. ~~讲讲你用Instrument优化动画性能的经历~~

---

15. `loadView` 的作用是什么?

> *This method loads or creates a view and assigns it to the `view` property.*

`loadView` 是 `UIViewController` 的实例方法, 我们永远不要直接调用这个方法 `[self loadView]`. 这在苹果的官方文档中已经明确的写出了. `loadView` 会在获取视图控制器的 `view` 但是却得到 `nil` 时被调用.

`loadView` 的具体实现会做下面两件事情中的一件:

1. 如果你的视图控制器关联了一个 storyboard, 那么它就会加载 storyboard 中的视图.

2. 如果试图控制器没有关联的 storyboard, 那么就会创建一个空的视图, 并分配给 `view` 属性

如果你需要覆写 `loadView` 方法:

1. 你需要创建一个根视图.
2. 创建并初始化 `view` 的子视图, 调用 `addSubview:` 方法将它们添加到父视图上.
3. 如果你使用了自动布局, 提供足够的约束来保证视图的位置.
4. 将根视图分配给 `view` 属性.

永远不要在这个方法中调用 `[super loadView]`.

---

16. `viewWillLayoutSubviews` 的作用是什么?

`viewWillLayoutSubviews` 方法会在视图的 bounds 改变时, 视图会调整子视图的位置, 我们可以在视图控制器中覆写这个方法在视图放置子视图前做出改变, 当屏幕的方向改变时, 这个方法会被调用.

---

17. GCD 里面有哪几种 Queue? 背后的线程模型是什么样的?

GCD 中 Queue 的种类还要看我们怎么进行分类, 如果根据同一时间内处理的操作数分类的话, GCD 中的 Queue 分为两类

1. Serial Dispatch Queue
2. Concurrent Dispatch Queue

一类是串行派发队列, 它只使用一个线程, 会等待当前执行的操作结束后才会执行下一个操作, 它按照追加的顺序进行处理. 另一类是并行派发队列, 它同时使用多个线程, 如果当前的线程数足够, 那么就不会等待正在执行的操作, 使用多个线程同时执行多个处理.

另外的一种分类方式如下:

1. Main Dispatch Queue
2. Global Dispatch Queue
3. Custom Dispatch Queue

主线程只有一个, 它是一个串行的进程. 所有追加到 Main Dispatch Queue 中的处理都会在 RunLoop 在执行. Global Dispatch Queue 是所有应用程序都能使用的并行派发队列, 它有 4 个执行优先级 High, Default, Low, Background. 当然我们也可以使用 `dispatch_queue_create` 创建派发队列.

---

18. Core Data 或者 sqlite 的读写是分线程的吗? 死锁如何解决?

Core Data 和 sqlite 这两个我还真没深入用过, 我只在小的玩具应用上使用过 Core Data, 但是发现这货实在是太难用了, 我就果断放弃了,

sqlite 我也用过, 每次输入 SQL 语句的时候我多想吐槽, 写一些简单的还好, 复杂的就直接 Orz 了. 所以我一般会使用 levelDB 对进行数据的持久存储.

数据库读取操作一般都是多线程的, 在对数据进行读取的时候, 我们要确保当前的状态不会被修改, 所以加锁, 防止由于线程竞争而出现的错误. 在 Core Data 中使用并行的最重要的规则是: **每一个 `NSManagedObjectContext` 必须只从创建它的进程中访问**.

---

19. http 的 POST 和 GET 有什么区别?

根据 HTTP 协议的定义 GET 类型的请求是幂等的, 而 POST 请求是有副作用的, 也就是说 GET 用于获取一些资源, 而 POST 用于改变一些资源, 这可能会创建新的资源或更新已有的资源.

POST 请求比 GET 请求更加的安全, 因为你不会把信息添加到 URL 上的查询字符串上. 所以使用 GET 来收集密码或者一些敏感信息并不是什么好主意.

最后, POST 请求比 GET 请求也可以传输更多的信息.

---

20. 什么是 Binary search tree, 它的时间复杂度是多少?

二叉搜索树是一棵以二叉树来组织的, 它搜索的时间复杂度 $O(h)$ 与树的高度成正比, 最坏的运行时间是 $\Theta(\lg n)$.

参考资料

- iOS: ARC和非ARC下使用Block属性的问题
- Which is threadsafe atomic or non atomic?
- NSObject +load and +initialize - What do they do?
- Objective-C Class Loading and Initialization
- Objective-C: init vs initialize
- Dynamic Tips & Tricks With Objective-C
- Method Swizzling
- What are the differences between a UIView and a CALayer?
- iOS Brownbag: View vs. Layers (including Clock Demo)
- UILabel layer cornerRadius negatively impacting performance
- UIView Class Reference
- 绘制像素到屏幕上
- Proper use of loadView and viewDidLoad with UIViewController without nibs/xibs
- UIViewController Class Reference
- Core Data & Threads
- Making Core Data Thread-safe
- What is the difference between POST and GET?
- When do you use POST and when do you use GET?
- Objective-C高级编程-iOS与OS-X多线程和内存管理

Draveness

继续阅读此作者的更多文章。

分享此博文

前一篇

iOS 源代码分析----Masonry

Masonry 是 Objective-C 中用于自动布局的第三方框架, 我们一般使用它来代替冗长, 繁琐的 AutoLayout 代码. Masonry 的使用还是很简洁的: [button mas_makeConstraints:^(MASConstraintMaker *make) { make.centerX.equalTo(self.…

后一篇

Swift 类构造器的使用

这几天在使用 Swift 重写原来的一个运动社交应用 SportJoin. 为什么要重写呢? 首先因为实在找不到设计师给我作图; 其次, 原来写的代码太烂了我也闲不下来, 想找一些项目做, 所以只好将原来的代码重写了. 原来的代码大约是一年半以前写的, 我现在真的不想吐槽当时写的代码有多烂, 有一句话怎么说来着: 程序员连自己写的源代码都不想读, 怎么可能看别人写的源代码! 每半年获得的知识相当于之前获得的全部知识的总和. 个人觉得这句话还是蛮有道理的. 反正对于我来说, 每过一段的时间回过头来看自己写的代码都感觉有很大的重构空间,…

Proudly published with **Ghost** & **GhostChina**