**iOS面试总结**

事件的传递和响应：

触摸事件的传递是由下往上传递的 即由keyWindow来通过hitTest:withEvent:来判断它的子视图是不是那个点是不是在当前View里， 判断到最后一个是当前View里的时候，就是找到最合适的View。

触摸事件的响应式从上往下的，即找出最合适的View之后就从子View一直往父View来找出合适响应事件的 View，来响应相关的事件。如果当前的 View 有添加手势，那么直接响应相应的事件，不会继续向下寻找了，如果没有手势事件，那么会看其是否实现了如下的方法：

- (void)touchesBegan:(NSSet \*)touches withEvent:(UIEvent \*)event;

- (void)touchesMoved:(NSSet \*)touches withEvent:(UIEvent \*)event;

- (void)touchesEnded:(NSSet \*)touches withEvent:(UIEvent \*)event;

- (void)touchesCancelled:(NSSet \*)touches withEvent:(UIEvent \*)event;

如果有实现那么就由此 View 响应，如果没有实现，那么就会传递给他的下一个响应者（父view或者控制器）

为了方便，我们将

- (nullableUIView \*)hitTest:(CGPoint)point withEvent:(nullableUIEvent \*)event；称为方法A

- (BOOL)pointInside:(CGPoint)point withEvent:(nullableUIEvent \*)event；称为方法B

对view进行重写这两个方法后，就会发现，点击屏幕后，首先响应的是方法A；

如果方法A中，我们没有调用父类的这个方法，那就根据这个方法A的返回view，作为响应事件的view。（当然返回nil，就是这个view不响应）

如果方法A中，我们调用了父类的这个方法，也就是

[super hitTest:point withEvent:event];那这个时候系统就要调用方法B；通过这个方法的返回值，来判断当前这个view能不能响应消息。

如果方法B返回的是no，那就不用再去遍历它的子视图。方法A返回的view就是可以响应事件的view。

如果方法B返回的是YES，那就去遍历它的子视图。（就是上图我们描述的那样，找到合适的view返回，如果找不到，那就由方法A返回的view去响应这个事件。）

因此总结下来：

//返回一个view来响应事件 （我们如果不想影响系统的事件传递链，在这个方法内，最好调用父类的这个方法）

- (nullableUIView \*)hitTest:(CGPoint)point withEvent:(nullableUIEvent \*)event；

//返回的值可以用来判断是否继续遍历子视图（返回的根据是触摸的point是否在view的frame范围内）

- (BOOL)pointInside:(CGPoint)point withEvent:(nullableUIEvent \*)event；

**sizeToFit 和 sizeThatFits的使用区别**

sizeToFit:会计算出最优的 size 而且会改变自己的size

sizeThatFits:会计算出最优的 size 但是不会改变 自己的 size

**setNeedsLayout与layoutIfNeeded的区别**

setNeedsLayout

标记为需要重新布局，异步调用layoutIfNeeded刷新布局，不立即刷新，在下一轮runloop结束前刷新，对于这一轮runloop之内的所有布局和UI上的更新只会刷新一次，layoutSubviews一定会被调用。

layoutIfNeeded

如果有需要刷新的标记，立即调用layoutSubviews进行布局（如果没有标记，不会调用layoutSubviews）。

**关键点**

layoutIfNeeded不一定会调用layoutSubviews方法。

setNeedsLayout一定会调用layoutSubviews方法（有延迟，在下一轮runloop结束前）。

如果想在当前runloop中立即刷新，调用顺序应该是

[self setNeedsLayout];

[self layoutIfNeeded];

反之可能会出现布局错误的问题。

UIWebView特点:

1.加载速度慢；

2.内存占用多，内存优化困难；

3.如果内存占用过多，还可能因为占用过多被系统kill掉。

WKWebView特点：

1. 运行速度更快，占用内存更少。同样加载一个百度首页，UIWebView占用35兆左右，WKWebView仅仅需要补刀十兆。性能相差三倍不止。因此我们需要做的仅仅是果断抛弃UIWebView。

Strong 和copy修饰字符串有什么不同，

strong修饰只是在对对象引用计数加1 （无论可变不可变）是浅拷贝 copy修饰不可变对象的时候和strong是一样的是浅拷贝 修饰可变对象的时候是不仅引用计数+1 还对对象进行了深拷贝

AF2之前要 创建一个 常驻线程，AF3之后 又把常驻线程去掉了？

因为AF2用的是 NSURLconnection 他要求发送请求和接受回调在同一个线程 AF3之后改为NSURLSession系统允许在不同线程发送 和接受回调

webview和原生交互的过程（JavaScriptBridge）最终转化成stringByEvaluatingJavaScriptFromString方法

runtime如何实现weak变量的自动置nil？知道SideTable吗？

runtime 对注册的类会进行布局，对于 weak 修饰的对象会放入一个 hash 表中。 用 weak 指向的对象内存地址作为key，当此对象的引用计数为0的时候会 dealloc，假如 weak 指向的对象内存地址是a，那么就会以a为键， 在这个 weak表中搜索，找到所有以a为键的 weak 对象，从而设置为 nil。

更细一点的回答：

1.初始化时：runtime会调用objc\_initWeak函数，初始化一个新的weak指针指向对象的地址。

2.添加引用时：objc\_initWeak函数会调用objc\_storeWeak() 函数， objc\_storeWeak()的作用是更新指针指向，创建对应的弱引用表。

3.释放时,调用clearDeallocating函数。clearDeallocating函数首先根据对象地址获取所有weak指针地址的数组，然后遍历这个数组把其中的数据设为nil，最后把这个entry从weak表中删除，最后清理对象的记录。

**性能优化**

CPU方面

1.对象创建

不需要响应触摸事件的控件，用 CALayer比UIView 更加合适。

如果对象不涉及 UI 操作，则尽量放到后台线程去创建

如果对象可以复用，并且复用的代价比释放、创建新对象要小，那么这类对象应当尽量放到一个缓存池里复用。

2.对象的调整

尽量避免调整视图层次、添加和移除视图

避免重复多次刷新 (可用联合刷新机制 dispatch\_source\_t)

需要重绘的区域尽可能缩小

3.对象的销毁

放到后台线程,用block捕获机制 尤其的容器类

4.布局计算

cell的高度,及子控件frame的缓存,到模型了 预排版

例如聊天页面消息

5.文本的计算,文理渲染及图片的解码 尽量把消耗造作放到子线程执行

GPU方面

1.避免图层混合

尽量减少视图数量和层次 减少透明视图 不透明的opaque 为yes

当多个视图重叠在一起,如果结构复杂也会消耗CPU资源,如果存在透明,可能会触发颜色的混合,所以我们应该尽量减少视图的层次及数量,慎用透明.

2.临时转换

避免控件frame和图片大小不一致,可以让UI设计图片大小,也用用代码动态的修改

确保图片颜色格式被GPU支持

3.尽量避免离屏渲染

离屏渲染:当我们设置了某一些UI视图的图层属性时, 被GPU标记为在合成之前不能直接用于显示时,就会会触发离屏渲染,指的是在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染及合成操作.

**哪些会触发离屏渲染**

1.圆角(maskToBounds 一起使用)

2.图层蒙版

mask

3.阴影

showPath

4.光栅化

label.layer.shouldRasterize 缓存 0.1秒

为何要避免离屏渲染

离屏渲染会触发openGL多通道渲染管线,产生额外的开销.增加GPU的工作量,从而可能导致CPU和GPU工作总耗时超出了16.7毫秒,就会导致UI的卡顿和掉帧.

**电量优化：**

定位方面的优化

1、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

2、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

3、尽量降低定位精度，比如尽量不要使用精度最高的 kCLLocationAccuracyBest

4、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

5、尽量不要使用 startMonitoringSignificantLocationChanges，优先考虑 startMonitoringForRegion:

**网络优化：**

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

**Block：**

Block是封装了函数调用及函数调用环境的OC对象,底层是一个结构体,结构里里面有isa指针

代码块的函数指针Funcptr,及捕获的外部变量.

block捕获变量的规则是 auto变量是捕获值,static变量是捕获指针 全局变量不捕获

自动释放池 AutoreleasePool

主要由 双向链表和栈实现的

主要由AutoreleasePoolPage 对象

和

[obj autorelease]

AutoreleasePoolPage::push

AutoreleasePoolPage::pop

两个方法实现

AutoreleasePoolPage 为双向链表中的节点 携带left, right 指向AutoreleasePoolPage对象的指针和next 指向栈顶的指针

每当调用AutoreleasePoolPage::push 在next指针处插入一个nil 作为哨兵

每当调用AutoreleasePoolPage::pop则拿到AutoreleasePoolPage对象,一直出栈,出栈的autorelease对象调用release,直到遇到最近一个哨兵对象,并修改next指针指向

**gcd和NSOperationQueue的区别**

1、GCD是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中，在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；而Operation作为一个对象，为我们提供了更多的选择；

2、在NSOperationQueue中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而GCD没法停止已经加入queue的block(其实是有的，但需要许多复杂的代码)；

3、NSOperation能够方便地设置依赖关系，我们可以让一个Operation依赖于另一个Operation，这样的话尽管两个Operation处于同一个并行队列中，但前者会直到后者执行完毕后再执行；

4、我们能将KVO应用在NSOperation中，可以监听一个Operation是否完成或取消，这样子能比GCD更加有效地掌控我们执行的后台任务；

5、在NSOperation中，我们能够设置NSOperation的priority优先级，能够使同一个并行队列中的任务区分先后地执行，而在GCD中，我们只能区分不同任务队列的优先级，如果要区分block任务的优先级，也需要大量的复杂代码；

6、我们能够对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将block任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。

触摸事件是**source0**还是**source1**

Source0事件：处理如UIEvent，CFSocket这类事件。需要手动触发。触摸事件其实是Source1接收系统事件后在回调 \_\_IOHIDEventSystemClientQueueCallback() 内触发的 Source0，Source0 再触发的 \_UIApplicationHandleEventQueue()。source0一定是要唤醒runloop及时响应并执行的，如果runloop此时在休眠等待系统的 mach\_msg事件，那么就会通过source1来唤醒runloop执行。

Source1事件：处理系统内核的mach\_msg事件。（推测CADisplayLink也是这里触发）。

**+（void）调用**

+load 方法是系统自动调用的，无需手动调用，系统自动为每一个类调用+load方法（如果有），所以也无需手动调用[super load]方法。

+load 方法按照[SuperClass load]->[Class load]->[ChildClass load]的顺序加载。

+load 方法是在所有类被加入到runtime以后调用的。

[ChildClass load]方法是按照Compile Sources的排列顺序加载的，但要遵循调用[ChildClass load]之前，必须先调用其[SuperClass load]方法。

在所有类的+load方法调用完以后再调用[Category load]方法，[Category load]的调用顺序完全按照Compile Sources排列顺序。

**id和instancetype 小结**

1.instancetype是一个上下文关键字，只允许在Objective-C方法的结果类型中使用，不能作为参数使用.能够做到返回类型检测。

2.id是一个参数类型，没法做返回类型检测,可以用作返回结果类型和作为参数使用，但是有时候我们也看到类似- (id)initWithxxx:写法，同样可以做到类型检测，那是因为编译器自动转换以alloc，init或new开头的方法并返回相关类型。

3.官方建议使用instancetype关键字作为返回调用类的实例（或该类的子类）的方法的返回类型。这些方法包括alloc，init和类工厂方法。使用instancetype而不是id,在适当的位置改进Objective-C代码中的类型安全性

**Category 在编译过后，是在什么时机与原有的类合并到一起的？**

\_objc\_init-> map\_images-> map\_images\_nolock-> 1 read\_images，这个方法会读取所有的类的相关信息。

->reMethodizeClass:，这个方法是重新方法化的意思。(内部调用attachCategories将方法列表和协议列表与原有类合并)

**atomic和nonatomic的对比**

1、atomic和nonatomic用来决定编译器生成的getter和setter是否为原子操作。

2、atomic：系统生成的 getter/setter 会保证 get、set 操作的完整性，不受其他线程影响。getter 还是能得到一个完好无损的对象（可以保证数据的完整性），但这个对象在多线程的情况下是不能确定的，

也就是说：如果有多个线程同时调用setter的话，不会出现某一个线程执行完setter全部语句之前，另一个线程开始执行setter情况，相当于函数头尾加了锁一样，每次只能有一个线程调用对象的setter方法，所以可以保证数据的完整性。

atomic所说的线程安全只是保证了getter和setter存取方法的线程安全，并不能保证整个对象是线程安全的。

3、nonatomic：就没有这个保证了，nonatomic返回你的对象可能就不是完整的value。因此，在多线程的环境下原子操作是非常必要的，否则有可能会引起错误的结果。但仅仅使用atomic并不会使得对象线程安全，我们还要为对象线程添加lock来确保线程的安全。

4、nonatomic的速度要比atomic的快。

5、atomic与nonatomic的本质区别其实也就是在setter方法上的操作不同