|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | 1.入口 setImageWithURL:placeholderImage:options:  会先把 placeholderImage 显示，然后 SDWebImageManager 根据 URL 开始处理图片。  2.进入 SDWebImageManagerdownloadWithURL:delegate:options:userInfo:，  交给 SDImageCache 从缓存查找图片是否已经下载 queryDiskCacheForKey:delegate:userInfo:.  3.先从内存图片缓存查找是否有图片，  如果内存中已经有图片缓存，SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo: 到 SDWebImageManager。  4.SDWebImageManagerDelegate 回调 webImageManager:didFinishWithImage:  到 UIImageView+WebCache 等前端展示图片。  5.如果内存缓存中没有，生成 NSInvocationOperation  添加到队列开始从硬盘查找图片是否已经缓存。  6.根据 URLKey 在硬盘缓存目录下尝试读取图片文件。  这一步是在 NSOperation 进行的操作，所以回主线程进行结果回调 notifyDelegate:。  7.如果上一操作从硬盘读取到了图片，将图片添加到内存缓存中  （如果空闲内存过小，会先清空内存缓存）。  SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo:。  进而回调展示图片。  8.如果从硬盘缓存目录读取不到图片，  说明所有缓存都不存在该图片，需要下载图片，  回调 imageCache:didNotFindImageForKey:userInfo:。  9.共享或重新生成一个下载器 SDWebImageDownloader 开始下载图片。  10.图片下载由 NSURLConnection 来做，  实现相关 delegate 来判断图片下载中、下载完成和下载失败。  11.connection:didReceiveData: 中  利用 ImageIO 做了按图片下载进度加载效果。  12.connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。  13.图片解码处理在一个 NSOperationQueue 完成，  不会拖慢主线程 UI。如果有需要对下载的图片进行二次处理，  最好也在这里完成，效率会好很多。  14.在主线程 notifyDelegateOnMainThreadWithInfo:  宣告解码完成，  imageDecoder:didFinishDecodingImage:userInfo  回调给 SDWebImageDownloader。  15.imageDownloader:didFinishWithImage:  回调给 SDWebImageManager 告知图片下载完成。  16.通知所有的 downloadDelegates 下载完成，  回调给需要的地方展示图片。  17.将图片保存到 SDImageCache 中，  内存缓存和硬盘缓存同时保存。  写文件到硬盘也在以单独 NSInvocationOperation 完成，  避免拖慢主线程。  18.SDImageCache 在初始化的时候会注册一些消息通知，  在内存警告或退到后台的时候清理内存图片缓存  应用结束的时候清理过期图片。  19.SDWI 也提供了 UIButton+WebCache 和  MKAnnotationView+WebCache，方便使用。  20.SDWebImagePrefetcher 可以预先下载图片，  方便后续使用。 |

**iOS**面试题：腾讯一面以及参考思路：

**1** 使用了第三方库**,** 有看他们是怎么实现的吗？

例：SD、YY、AFN、MJ等！

**<1>.SD**为例：

**2.**强连通分量了解嘛？

概念：

有向图强连通分量：在有向图G中，如果两个顶点vi,vj间（vi>vj）有一条从vi到vj的有向路径，同时还有一条从vj到vi的有向路径，则称两个顶点强连通(strongly

connected)。如果有向图G的每两个顶点都强连通，称G是一个强连通图。有向图的极大强连通子图，称为强连通分量(strongly

connected components)。

定义：

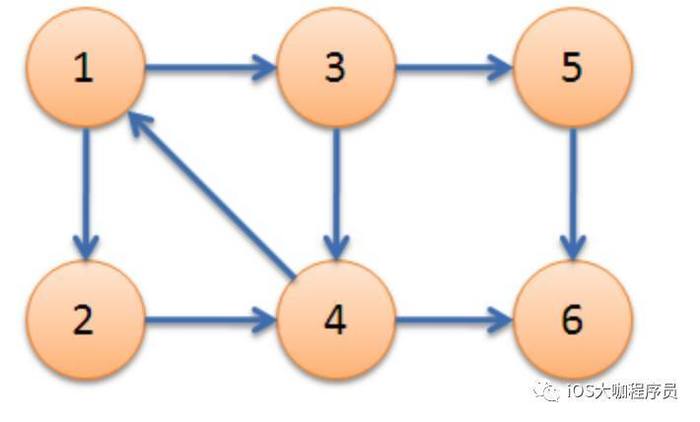
有向图强连通分量：

在有向图G中，如果两个顶点间至少存在一条路径，称两个顶点强连通（strongly connected）。

如果有向图G的每两个顶点都强连通，则称G是一个强连通图。

非强连通图有向图的极大强连通子图，成为强连通分量（strongly connected components）。

下图中，子图{1,2,3,4}为一个强连通分量，因为顶点1,2,3,4两两可达，{5}，{6}也分别是两个强连通分量。



直接根据定义，用双向遍历取交际的方法求强连通分量，时间复杂度为O（N^2+M）。更好的方法是Kosaraju算法或者Tarjan算法

两者的时间复杂度都是O（N+M）。本文介绍的是Tarjan算法。

算法原理：（**Tarjan**）

need-to-insert-img

Tarjan算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一颗子树。

搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以盘对栈顶到栈中的节点是否为一个强连通分量。

定义DFN（u）为节点u搜索的次序编号（时间戳）。Low（u）为u或者u的子树能够追溯到的最早的栈中的节点的次序号。

由定义可以得出：

Low（u）= Min { DFN（u）， Low（v）} （（u，v）为树枝边，u为v的父节点DFN（v），（u，v）为指向栈中节点的后向边（非横叉边））

当DFN（u）=Low（u）时，以u为根的搜索子树上所有节点是一个强连通分量。

**3.**遇到**tableView**卡顿嘛？会造成卡顿的原因大致有哪些？

可能造成**tableView**卡顿的原因有：

**1.**最常用的就是**cell**的重用，注册重用标识符

如果不重用cell时，每当一个cell显示到屏幕上时，就会重新创建一个新的cell

如果有很多数据的时候，就会堆积很多cell。

如果重用cell，为cell创建一个ID，每当需要显示cell 的时候，都会先去缓冲池中寻找可循环利用的cell，如果没有再重新创建cell

**2.**避免**cell**的重新布局

cell的布局填充等操作 比较耗时，一般创建时就布局好

如可以将cell单独放到一个自定义类，初始化时就布局好

**3.**提前计算并缓存**cell**的属性及内容

当我们创建cell的数据源方法时，编译器并不是先创建cell 再定cell的高度

而是先根据内容一次确定每一个cell的高度，高度确定后，再创建要显示的cell，滚动时，每当cell进入凭虚都会计算高度，提前估算高度告诉编译器，编译器知道高度后，紧接着就会创建cell，这时再调用高度的具体计算方法，这样可以方式浪费时间去计算显示以外的cell

**4.**减少**cell**中控件的数量

尽量使cell得布局大致相同，不同风格的cell可以使用不用的重用标识符，初始化时添加控件，

不适用的可以先隐藏

**5.**不要使用**ClearColor**，无背景色，透明度也不要设置为**0**

渲染耗时比较长

**6.**使用局部更新

如果只是更新某组的话，使用reloadSection进行局部更

**7.**加载网络数据，下载图片，使用异步加载，并缓存

**8.**少使用**addView** 给**cell**动态添加**view**

**9.**按需加载**cell**，**cell**滚动很快时，只加载范围内的**cell**

**10.**不要实现无用的代理方法，**tableView**只遵守两个协议

**11.**缓存行高：**estimatedHeightForRow**不能和**HeightForRow**里面的**layoutIfNeed**同时存在，这两者同时存在才会出现**“**窜动**”**的**bug**。所以我的建议是：只要是固定行高就写预估行高来减少行高调用次数提升性能。如果是动态行高就不要写预估方法了，用一个行高的缓存字典来减少代码的调用次数即可

**12.**不要做多余的绘制工作。在实现drawRect:的时候，它的rect参数就是需要绘制的区域，这个区域之外的不需要进行绘制。例如上例中，就可以用CGRectIntersectsRect、CGRectIntersection或CGRectContainsRect判断是否需要绘制image和text，然后再调用绘制方法。

**13.**预渲染图像。当新的图像出现时，仍然会有短暂的停顿现象。解决的办法就是在bitmap context里先将其画一遍，导出成UIImage对象，然后再绘制到屏幕；

**14.**使用正确的数据结构来存储数据。

**4.M**、**V**、**C**相互通讯规则你知道的有哪些？

**MVC** 是一种设计思想，一种框架模式，是一种把应用中所有类组织起来的策略，它把你的程序分为三块，分别是：

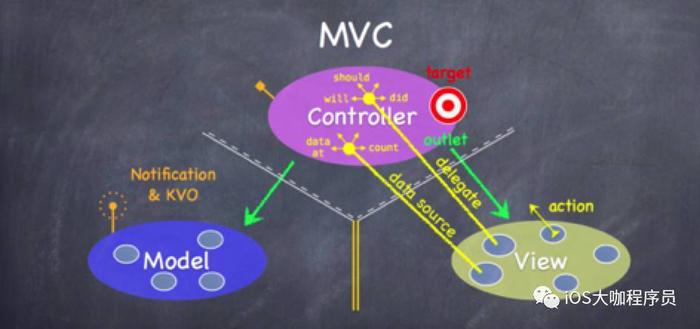
**M**（**Model**）：实际上考虑的是“什么”问题，你的程序本质上是什么，独立于 UI 工作。是程序中用于处理应用程序逻辑的部分，通常负责存取数据。

C（Controller）：控制你 Model 如何呈现在屏幕上，当它需要数据的时候就告诉 Model，你帮我获取某某数据；当它需要 UI 展示和更新的时候就告诉 View，你帮我生成一个 UI 显示某某数据，是 Model 和 View 沟通的桥梁。

**V**（**View**）：Controller 的手下，是 Controller 要使用的类，用于构建视图，通常是根据 Model 来创建视图的。

要了解 **MVC** 如何工作，首先需要了解这三个模块间如何通信。

MVC通信规则



**Controller to Model**

可以直接单向通信。Controller 需要将 Model 呈现给用户，因此需要知道模型的一切，还需要有同 Model 完全通信的能力，并且能任意使用 Model 的公共 API。

**Controller to View**

可以直接单向通信。Controller 通过 View 来布局用户界面。

**Model to View**

永远不要直接通信。Model 是独立于 UI 的，并不需要和 View 直接通信，View 通过 Controller 获取 Model 数据

**View to Controller**

View 不能对 Controller 知道的太多，因此要通过间接的方式通信。

Target

action。首先 Controller 会给自己留一个 target，再把配套的 action 交给 View 作为联系方式。那么 View

接收到某些变化时，View 就会发送 action 给 target 从而达到通知的目的。这里 View 只需要发送

action，并不需要知道 Controller 如何去执行方法。

代理。有时候 View 没有足够的逻辑去判断用户操作是否符合规范，他会把判断这些问题的权力委托给其他对象，他只需获得答案就行了，并不会管是谁给的答案。

DataSoure。View 没有拥有他们所显示数据的权力，View 只能向 Controller 请求数据进行显示，Controller 则获取 Model 的数据整理排版后提供给 View。

Model 访问 Controller

同样的 Model 是独立于 UI 存在的，因此无法直接与 Controller 通信，但是当 Model 本身信息发生了改变的时候，会通过下面的方式进行间接通信。

**Notification & KVO**一种类似电台的方法，**Model** 信息改变时会广播消息给感兴趣的人，只要 **Controller** 接收到了这个广播的时候就会主动联系 **Model**，获取新的数据并提供给 **View**。

从上面的简单介绍中我们来简单概括一下 MVC 模式的优点。

1.低耦合性

2.有利于开发分工

3.有利于组件重用

4.可维护性

**5.NStimer**准吗？谈谈你的看法？如果不准该怎样实现一个精确的**NSTimer?**

**1.**不准

**2.**不准的原因如下：

1、NSTimer加在main runloop中，模式是NSDefaultRunLoopMode，main负责所有主线程事件，例如UI界面的操作，复杂的运算，这样在同一个runloop中timer就会产生阻塞。

2、模式的改变。主线程的 RunLoop 里有两个预置的 Mode：kCFRunLoopDefaultMode 和 UITrackingRunLoopMode。

当你创建一个 Timer 并加到 DefaultMode 时，Timer 会得到重复回调，但此时滑动一个ScrollView时，RunLoop 会将 mode 切换为 TrackingRunLoopMode，这时 Timer 就不会被回调，并且也不会影响到滑动操作。所以就会影响到NSTimer不准的情况。

PS:DefaultMode 是 App 平时所处的状态，rackingRunLoopMode 是追踪 ScrollView 滑动时的状态。

方法一：

1、在主线程中进行NSTimer操作，但是将NSTimer实例加到main runloop的特定mode（模式）中。避免被复杂运算操作或者UI界面刷新所干扰。

self.timer = [NSTimer timerWithTimeInterval:1 target:self selector:@selector(showTime) userInfo:nil repeats:YES];

[[NSRunLoop currentRunLoop] addTimer:self.timer forMode:NSRunLoopCommonModes];

2、在子线程中进行NSTimer的操作，再在主线程中修改UI界面显示操作结果；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | - (void)timerMethod2 {  NSThread \*thread = [[NSThread alloc] initWithTarget:self selector:@selector(newThread) object:nil];  [thread start];  }  - (void)newThread  {  @autoreleasepool  {  [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(showTime) userInfo:nil repeats:YES];  [[NSRunLoop currentRunLoop] run];  }  } |

总结：

一开始的时候系统就为我们将主线程的main runloop隐式的启动了。

在创建线程的时候，可以主动获取当前线程的runloop。每个子线程对应一个runloop

方法二：

使用示例

使用mach内核级的函数可以使用mach\_absolute\_time()获取到CPU的tickcount的计数值，可以通过”mach\_timebase\_info”函数获取到纳秒级的精确度 。然后使用mach\_wait\_until(uint64\_t deadline)函数，直到指定的时间之后，就可以执行指定任务了。

关于数据结构mach\_timebase\_info的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | struct mach\_timebase\_info {uint32\_t numer;uint32\_t denom;};  #include  #include  static const uint64\_t NANOS\_PER\_USEC = 1000ULL;  static const uint64\_t NANOS\_PER\_MILLISEC = 1000ULL \* NANOS\_PER\_USEC;  static const uint64\_t NANOS\_PER\_SEC = 1000ULL \* NANOS\_PER\_MILLISEC;  static mach\_timebase\_info\_data\_t timebase\_info;  static uint64\_t nanos\_to\_abs(uint64\_t nanos) {  return nanos \* timebase\_info.denom / timebase\_info.numer;  }  void example\_mach\_wait\_until(int seconds)  {  mach\_timebase\_info(&timebase\_info);  uint64\_t time\_to\_wait = nanos\_to\_abs(seconds \* NANOS\_PER\_SEC);  uint64\_t now = mach\_absolute\_time();  mach\_wait\_until(now + time\_to\_wait);  } |

方法三：直接使用**GCD**替代！

**iOS**面试题：腾讯二面以及参考思路：

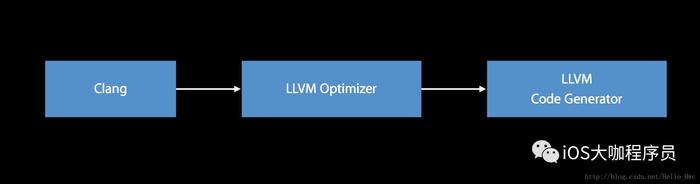
**1** 编译过程做了哪些事情？

1.C++,Objective C都是编译语言。编译语言在执行的时候，必须先通过编译器生成机器码，机器码可以直接在CPU上执行，所以执行效率较高。

iOS开发目前的常用语言是：Objective和Swift。二者都是编译语言，换句话说都是需要编译才能执行的。二者的编译都是依赖于Clang + LLVM. OC和Swift因为原理上大同小异，知道一个即可！

**iOS**编译

不管是OC还是Swift，都是采用Clang作为编译器前端，LLVM(Low level vritual machine)作为编译器后端。所以简单的编译过程如图



编译器前端

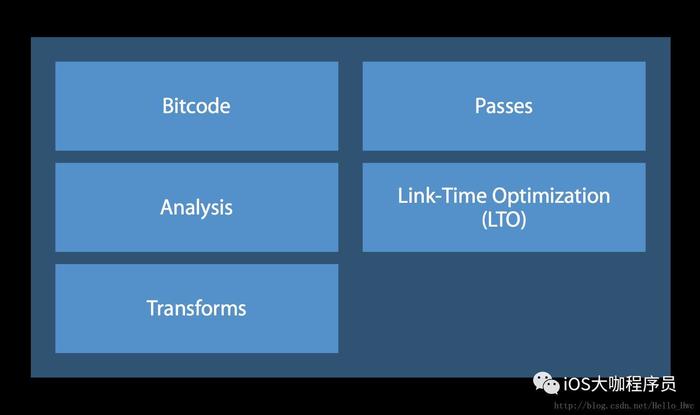
编译器前端的任务是进行：语法分析，语义分析，生成中间代码(intermediate representation )。在这个过程中，会进行类型检查，如果发现错误或者警告会标注出来在哪一行。



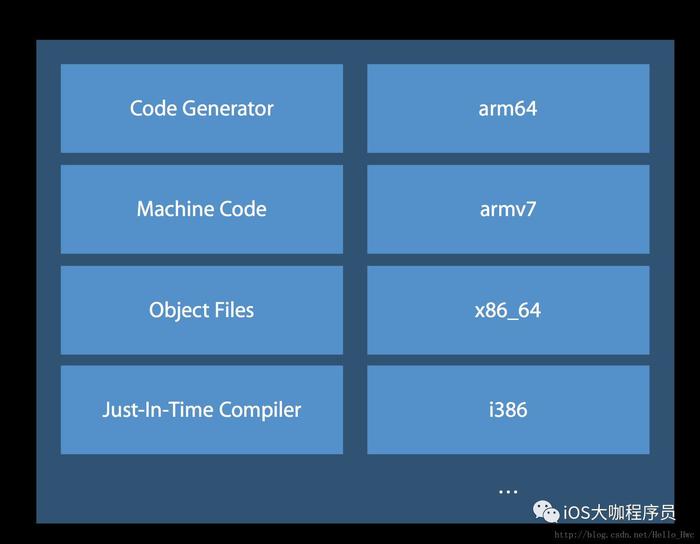
编译器后端

编译器后端会进行机器无关的代码优化，生成机器语言，并且进行机器相关的代码优化。iOS的编译过程，后端的处理如下

**LVVM**优化器会进行**BitCode**的生成，链接期优化等等

****

**LLVM**机器码生成器会针对不同的架构，比如**arm64**等生成不同的机器码。

****

执行一次**XCode build**的流程

当你在XCode中，选择build的时候（快捷键command+B），会执行如下过程

编译信息写入辅助文件，创建编译后的文件架构(name.app)

处理文件打包信息，例如在debug环境下



执行CocoaPod编译前脚本

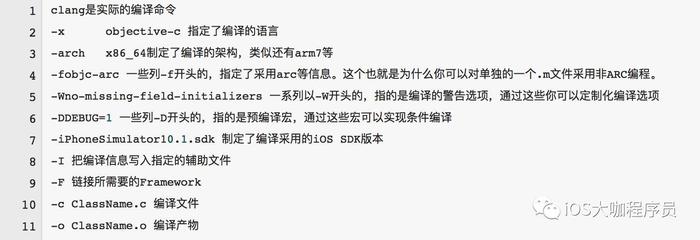
例如对于使用CocoaPod的工程会执行CheckPods Manifest.lock

编译各个.m文件，使用CompileC和clang命令。

编译各个**.m**文件，使用**CompileC**和**clang**命令。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 1.CompileC ClassName.o ClassName.m normal x86\_64 objective-c com.apple.compilers.llvm.clang.1\_0.compiler  2.export.US-ASCII  3.export PATH="..."  4.clang-x objective-c -arch x86\_64 -fmessage-length=0 -fobjc-arc...  -Wno-missing-field-initializers ... -DDEBUG=1 ... -isysroot  iPhoneSimulator10.1.sdk -fasm-blocks ... -I 上文提到的文件 -F 所需要的Framework-iquote 所需要的Framework ... -c ClassName.c -o ClassName.o |

通过这个编译的命令，我们可以看到

****

**2.**字典大致实现原理；

一：字典原理

NSDictionary（字典）是使用hash表来实现key和value之间的映射和存储的

方法：- (void)setObject:(id)anObject forKey:(id)aKey;

Objective-C中的字典NSDictionary底层其实是一个哈希表

二：哈希原理

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

给定表M，存在函数f(key)，对任意给定的关键字值key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表M为哈希(Hash）表，函数f(key)为哈希(Hash) 函数。

哈希概念:哈希表的本质是一个数组，数组中每一个元素称为一个箱子(bin)，箱子中存放的是键值对。

三：哈希存储过程

**1.**根据 **key** 计算出它的哈希值 **h**。

**2.**假设箱子的个数为 **n**，那么这个键值对应该放在第 **(h % n)** 个箱子中。

**3.**如果该箱子中已经有了键值对，就使用开放寻址法或者拉链法解决冲突。

在使用拉链法解决哈希冲突时，每个箱子其实是一个链表，属于同一个箱子的所有键值对都会排列在链表中。

哈希表还有一个重要的属性: 负载因子(load factor)，它用来衡量哈希表的空/满程度，一定程度上也可以体现查询的效率，计算公式为:

负载因子 **=** 总键值对数 **/** 箱子个数

负载因子越大，意味着哈希表越满，越容易导致冲突，性能也就越低。因此，一般来说，当负载因子大于某个常数(可能是 1，或者 0.75 等)时，哈希表将自动扩容。

哈希表在自动扩容时，一般会创建两倍于原来个数的箱子，因此即使 key 的哈希值不变，对箱子个数取余的结果也会发生改变，因此所有键值对的存放位置都有可能发生改变，这个过程也称为重哈希(rehash)。

哈希表的扩容并不总是能够有效解决负载因子过大的问题。假设所有 key 的哈希值都一样，那么即使扩容以后他们的位置也不会变化。虽然负载因子会降低，但实际存储在每个箱子中的链表长度并不发生改变，因此也就不能提高哈希表的查询性能。

基于以上总结，细心的朋友可能会发现哈希表的两个问题**:**

**1.**如果哈希表中本来箱子就比较多，扩容时需要重新哈希并移动数据，性能影响较大。

**2.**如果哈希函数设计不合理，哈希表在极端情况下会变成线性表，性能极低。

**3.block**和函数指针的理解；

相似点：

函数指针和Block都可以实现回调的操作，声明上也很相似，实现上都可以看成是一个代码片段。

函数指针类型和Block类型都可以作为变量和函数参数的类型。（typedef定义别名之后，这个别名就是一个类型）

不同点：

函数指针只能指向预先定义好的函数代码块（可以是其他文件里面定义，通过函数参数动态传入的），函数地址是在编译链接时就已经确定好的。

Block本质是Objective-C对象，是NSObject的子类，可以接收消息。

函数里面只能访问全局变量，而Block代码块不光能访问全局变量，还拥有当前栈内存和堆内存变量的可读性（当然通过\_\_block访问指示符修饰的局部变量还可以在block代码块里面进行修改）。

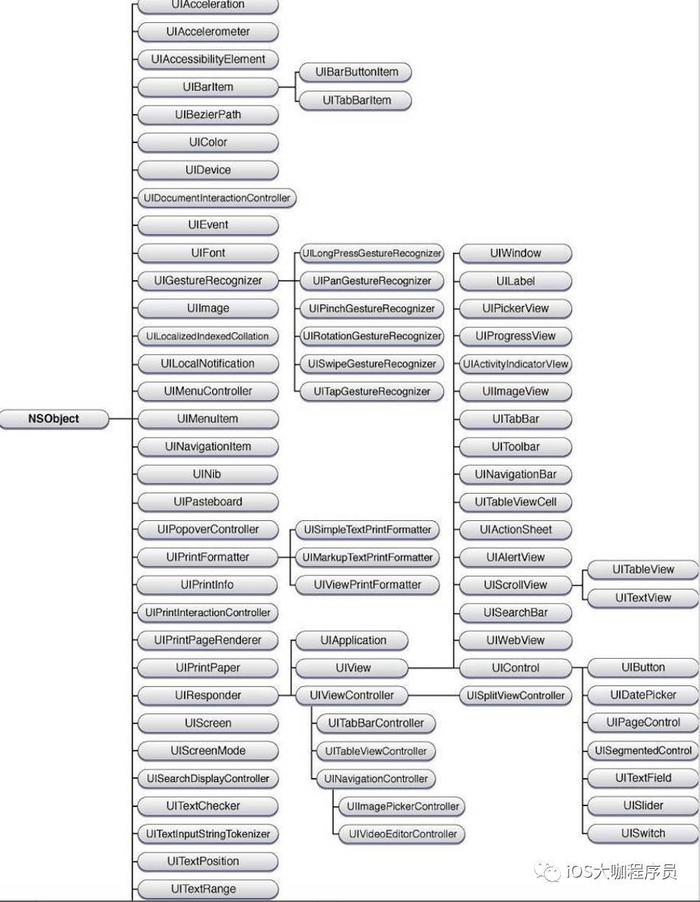
从内存的角度看，函数指针只不过是指向代码区的一段可执行代码，而block实际上是程序运行过程中在栈内存动态创建的对象，可以向其发送copy消息将block对象拷贝到堆内存，以延长其生命周期。

**4.**一般开始做一个项目，你的架构是如何思考的？

[参考文章一](https://link.jianshu.com/?t=https://casatwy.com/iosying-yong-jia-gou-tan-viewceng-de-zu-zhi-he-diao-yong-fang-an.html)

[参考文章二](https://link.jianshu.com/?t=https://casatwy.com/iosying-yong-jia-gou-tan-wang-luo-ceng-she-ji-fang-an.html)

**5.**你了解的**UIKit**结构？

****

这题应该是在你回答的基础之上，进行二次提问，主要还是看你的基础！

**iOS**面试题：腾讯三面

前言：

面试题均来源于自己群内成员面试后，口述而成，面试题和iOS程序员本身技术水平没任何关联，无论你能否全部答出，都不要对自己产生任何正面或消极的评价。

12月更新参考思路~

[腾讯一面以及参考思路](https://www.jianshu.com/p/0e9e7486e1a7)

[腾讯二面以及参考思路](https://www.jianshu.com/p/dd17bdcff9f7)

如果以上题目都能回答七八，请call Me,BAT需要你！

腾讯三面：

**1.OC**你了解的锁有哪些？在你回答基础上进行二次提问；

追问一：自旋和互斥对比？

追问二：使用以上锁需要注意哪些？

追问三：用**C/OC/C++**，任选其一，实现自旋或互斥？口述即可！

**2.**内存泄漏可能会出现的几种原因，聊聊你的看法？

追问一：非**OC**对象如何处理？

追问二：地图类内存若泄漏，如何处理？

追问三：若常用框架出现内存泄漏如何处理？

**3.**容错处理你们一般是怎么做的？

**4.**项目开始容错处理没做？如何防止拦截潜在的崩溃？

技术

基础

为什么说**Objective-C**是一门动态的语言？

讲一下**MVC**和**MVVM**，**MVP**？

为什么代理要用**weak**？代理的**delegate**和**dataSource**有什么区别？**block**和代理的区别**?**

属性的实质是什么？包括哪几个部分？属性默认的关键字都有哪些？**@dynamic**关键字和**@synthesize**关键字是用来做什么的？

属性的默认关键字是什么？

**NSString**为什么要用**copy**关键字，如果用**strong**会有什么问题？（注意：这里没有说用**strong**就一定不行。使用**copy**和**strong**是看情况而定的）

如何令自己所写的对象具有拷贝功能**?**

可变集合类和不可变集合类的 **copy** 和 **mutablecopy**有什么区别？如果是集合是内容复制的话，集合里面的元素也是内容复制么？

为什么**IBOutlet**修饰的**UIView**也适用**weak**关键字？

**nonatomic**和**atomic**的区别？**atomic**是绝对的线程安全么？为什么？如果不是，那应该如何实现？

**UICollectionView**自定义**layout**如何实现？

用**StoryBoard**开发界面有什么弊端？如何避免？

进程和线程的区别？同步异步的区别？并行和并发的区别？

线程间通信？

**GCD**的一些常用的函数？（**group**，**barrier**，信号量，线程同步）

如何使用队列来避免资源抢夺？

数据持久化的几个方案（**fmdb**用没用过）

说一下**AppDelegate**的几个方法？从后台到前台调用了哪些方法？第一次启动调用了哪些方法？从前台到后台调用了哪些方法？

**NSCache**优于**NSDictionary**的几点？

知不知道**Designated Initializer**？使用它的时候有什么需要注意的问题？

实现**description**方法能取到什么效果？

**objc**使用什么机制管理对象内存？

中级

**Block**

**block**的实质是什么？一共有几种**block**？都是什么情况下生成的？

为什么在默认情况下无法修改被**block**捕获的变量？ **\_\_block**都做了什么？

模拟一下循环引用的一个情况？**block**实现界面反向传值如何实现？

**Runtime**

**objc**在向一个对象发送消息时，发生了什么？

什么时候会报**unrecognized selector**错误？**iOS**有哪些机制来避免走到这一步？

能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？

**runtime**如何实现**weak**变量的自动置**nil**？

给类添加一个属性后，在类结构体里哪些元素会发生变化？

**RunLoop**

**runloop**是来做什么的？**runloop**和线程有什么关系？主线程默认开启了**runloop**么？子线程呢？

**runloop**的**mode**是用来做什么的？有几种**mode**？

为什么把**NSTimer**对象以**NSDefaultRunLoopMode**（**kCFRunLoopDefaultMode**）添加到主运行循环以后，滑动**scrollview**的时候**NSTimer**却不动了？

苹果是如何实现**Autorelease Pool**的？

类结构

**isa**指针？（对象的**isa**，类对象的**isa**，元类的**isa**都要说）

类方法和实例方法有什么区别？

介绍一下分类，能用分类做什么？内部是如何实现的？与该类原有方法的名称相同的时候，为什么原有方法会失效？

运行时能增加成员变量么？能增加属性么？如果能，如何增加？如果不能，为什么？

**objc**中向一个**nil**对象发送消息将会发生什么？（返回值是对象，是标量，结构体）

高级

**UITableview**的优化方法（缓存高度，异步绘制，减少层级，**hide**，避免离屏渲染）

有没有用过运行时，用它都能做什么？（交换方法，创建类，给新创建的类增加方法，改变**isa**指针）

看过哪些第三方框架的源码？都是如何实现的？（如果没有，问一下多图下载的设计）

**SDWebImage**的缓存策略？

**AFN**为什么添加一条常驻线程？

**KVO**的使用？实现原理？（为什么要创建子类来实现）

**KVC**的使用？实现原理？（**KVC**拿到**key**以后，是如何赋值的？知不知道集合操作符，能不能访问私有属性，能不能直接访问**\_ivar**）

项目

有已经上线的项目么？

项目里哪个部分是你完成的？（找一个亮点问一下如何实现的）

开发过程中遇到过什么困难，是如何解决的？

学习

遇到一个问题完全不能理解的时候，是如何帮助自己理解的？举个例子？

有看书的习惯么？最近看的一本是什么书？有什么心得？

有没有使用一些笔记软件？会在多平台同步以及多渠道采集么？（如果没有，问一下是如何复习知识的）

有没有使用清单类，日历类的软件？（如果没有，问一下是如何安排，计划任务的）

平常看博客么？有没有自己写过？（如果写，有哪些收获？如果没有写，问一下不写的原因）

**Objective-C** 中创建线程的方法是什么？如果在主线程中执行代码，方法是什么？如果想延时执行代码、方法又是什么？

答：线程创建有三种方法：使用**NSThread**创建、使用**GCD**的**dispatch**、使用子类化的**NSOperation,**然后将其加入**NSOperationQueue;**在主线程执行代码，方法是**performSelectorOnMainThread**，如果想延时执行代码可以用**performSelector:onThread:withObject:waitUntilDone:**

**Category**（类别）、 **Extension**（扩展）和继承的区别

区别：

分类有名字，类扩展没有分类名字，是一种特殊的分类。

分类只能扩展方法（属性仅仅是声明，并没真正实现），类扩展可以扩展属性、成员变量和方法。

继承可以增加，修改或者删除方法，并且可以增加属性。

我们说的**OC**是动态运行时语言是什么意思？

答：主要是将数据类型的确定由编译时，推迟到了运行时。简单来说**,** 运行时机制使我们直到运行时才去决定一个对象的类别**,**以及调用该类别对象指定方法。

为什么我们常见的**delegate**属性都用是**weak**而不是**retain/strong**？

答：是为了防止**delegate**两端产生不必要的循环引用。

**@property (nonatomic, weak) id<UITableViewDelegate> delegate;**

什么时候用**delete**，什么时候用**Notification**？

**Delegate(**委托模式**)**：**1**对**1**的反向消息通知功能。

**Notification(**通知模式**)**：只想要把消息发送出去，告知某些状态的变化。但是并不关心谁想要知道这个。

什么是 **KVO** 和 **KVC**？

**1). KVC(Key-Value-Coding)**：键值编码是一种通过字符串间接访问对象的方式（即给属性赋值）

举例说明：

**stu.name = @"**张三**" //** 点语法给属性赋值

**[stu setValue:@"**张三**" forKey:@"name"]; //** 通过字符串使用**KVC**方式给属性赋值

**stu1.nameLabel.text = @"**张三**";**

**[stu1 setValue:@"**张三**" forKey:@"nameLabel.text"]; //** 跨层赋值

**2). KVO(key-Value-Observing)**：键值观察机制他提供了观察某一属性变化的方法，极大的简化了代码。

**KVO**只能被**KVC**触发，包括使用**setValue:forKey:**方法和点语法。

**//** 通过下方方法为属性添加**KVO**观察

**-(void)addObserver:(NSObject \*)observer forKeyPath:(NSString \*)keyPath options:(NSKeyValueObservingOptions)options context:(nullable void \*)context;**

**//** 当被观察的属性发送变化时，会自动触发下方方法

**(void)observeValueForKeyPath:(NSString \*)keyPath ofObject:(id)object change:(NSDictionary \*)change context:(void \*)context{}**

**KVC** 和 **KVO** 的 **keyPath** 可以是属性、实例变量、成员变量。

**KVC**的底层实现？

当一个对象调用**setValue**方法时，方法内部会做以下操作：

**1).** 检查是否存在相应的**key**的**set**方法，如果存在，就调用**set**方法。

**2).** 如果**set**方法不存在，就会查找与**key**相同名称并且带下划线的成员变量，如果有，则直接给成员变量属性赋值。

**3).** 如果没有找到**\_key**，就会查找相同名称的属性**key**，如果有就直接赋值。

**4).** 如果还没有找到，则调用**valueForUndefinedKey:**和**setValue:forUndefinedKey:**方法。

这些方法的默认实现都是抛出异常，我们可以根据需要重写它们。

**KVO**的底层实现？

**KVO**基于**runtime**机制实现。

**ViewController**生命周期

按照执行顺序排列：

**1.initWithCoder**：通过**nib**文件初始化时触发。

**2.awakeFromNib**：**nib**文件被加载的时候，会发生一个**awakeFromNib**的消息到**nib**文件中的每个对象。

**3.loadView**：开始加载视图控制器自带的**view**。

**4.viewDidLoad**：视图控制器的**view**被加载完成。

**5.viewWillAppear**：视图控制器的**view**将要显示在**window**上。

**6.updateViewConstraints**：视图控制器的**view**开始更新**AutoLayout**约束。

**7.viewWillLayoutSubviews**：视图控制器的**view**将要更新内容视图的位置。

**8.viewDidLayoutSubviews**：视图控制器的**view**已经更新视图的位置。

**9.viewDidAppear**：视图控制器的**view**已经展示到**window**上。

**10.viewWillDisappear**：视图控制器的**view**将要从**window**上消失。

**11.viewDidDisappear**：视图控制器的**view**已经从**window**上消失。

方法和选择器有何不同？

**selector**是一个方法的名字，方法是一个组合体，包含了名字和实现。

你是否接触过**OC**中的反射机制？简单聊一下概念和使用

**1). class**反射

通过类名的字符串形式实例化对象。

**Class class = NSClassFromString(@"student");**

**Student \*stu = [[class alloc] init];**

将类名变为字符串。

**Class class =[Student class];**

**NSString className = NSStringFromClass(class);**

**2). SEL**的反射

通过方法的字符串形式实例化方法。

**SEL selector = NSSelectorFromString(@"setName");**

**[stu performSelector:selector withObject:@"Mike"];**

将方法变成字符串。

**NSStringFromSelector(@selector(setName:));**

调用方法有两种方式：

**1).** 直接通过方法名来调用。**[person show];**

**2).** 间接的通过**SEL**数据来调用 **SEL aaa = @selector(show); [person performSelector:aaa];**

如何对**iOS**设备进行性能测试？

答： **Profile-> Instruments ->Time Profiler**

开发项目时你是怎么检查内存泄露？

**1).** 静态分析 **analyze**。

**2). instruments**工具里面有个**leak**可以动态分析。

什么是懒加载？

答：懒加载就是只在用到的时候才去初始化。也可以理解成延时加载。

我觉得最好也最简单的一个例子就是**tableView**中图片的加载显示了**,** 一个延时加载**,** 避免内存过高**,**一个异步加载**,**避免线程堵塞提高用户体验。

类变量的 **@public**，**@protected**，**@private**，**@package** 声明各有什么含义？

**@public** 任何地方都能访问**;**

**@protected** 该类和子类中访问**,**是默认的**;**

**@private** 只能在本类中访问**;**

**@package** 本包内使用**,**跨包不可以。

什么是谓词？

谓词就是通过**NSPredicate**给定的逻辑条件作为约束条件**,**完成对数据的筛选。

**//**定义谓词对象**,**谓词对象中包含了过滤条件**(**过滤条件比较多**)**

**NSPredicate \*predicate = [NSPredicate predicateWithFormat:@"age<%d",30];**

**//**使用谓词条件过滤数组中的元素**,**过滤之后返回查询的结果

**NSArray \*array = [persons filteredArrayUsingPredicate:predicate];**

**isa**指针问题

**isa**：是一个**Class** 类型的指针**.** 每个实例对象有个**isa**的指针**,**他指向对象的类**,**而**Class**里也有个**isa**的指针**,** 指向**meteClass(**元类**)**。元类保存了类方法的列表。当类方法被调用时**,**先会从本身查找类方法的实现**,**如果没有**,**元类会向他父类查找该方法。同时注意的是**:**元类**(meteClass)**也是类**,**它也是对象。元类也有**isa**指针**,**它的**isa**指针最终指向的是一个根元类**(root meteClass)**。根元类的**isa**指针指向本身**,**这样形成了一个封闭的内循环。

如何访问并修改一个类的私有属性？

**1).** 一种是通过**KVC**获取。

**2).** 通过**runtime**访问并修改私有属性。

一个**objc**对象的**isa**的指针指向什么？有什么作用？

答：指向他的类对象,从而可以找到对象上的方法。