# 1、大厂常问iOS面试题--组件化篇

**1.组件化有什么好处？**

* 业务分层、解耦，使代码变得可维护；
* 有效的拆分、组织日益庞大的工程代码，使工程目录变得可维护；
* 便于各业务功能拆分、抽离，实现真正的功能复用；
* 业务隔离，跨团队开发代码控制和版本风险控制的实现；
* 模块化对代码的封装性、合理性都有一定的要求，提升开发同学的设计能力；
* 在维护好各级组件的情况下，随意组合满足不同客户需求；（只需要将之前的多个业务组件模块在新的主App中进行组装即可快速迭代出下一个全新App）

**2.你是如何组件化解耦的？**

* 分层

基础功能组件：按功能分库，不涉及产品业务需求，跟库Library类似，通过良好的接口拱上层业务组件调用；不写入产品定制逻辑，通过扩展接口完成定制；

基础UI组件：各个业务模块依赖使用，但需要保持好定制扩展的设计

业务组件：业务功能间相对独立，相互间没有Model共享的依赖；业务之间的页面调用只能通过UIBus进行跳转；业务之间的逻辑Action调用只能通过服务提供；

* 中间件：target-action，url-block，protocol-class

**3.为什么CTMediator方案优于基于Router的方案？**

Router的缺点：

* 在组件化的实施过程中，注册URL并不是充分必要条件。组件是不需要向组件管理器注册URL的，注册了URL之后，会造成不必要的内存常驻。注册URL的目的其实是一个服务发现的过程，在iOS领域中，服务发现的方式是不需要通过主动注册的，使用runtime就可以了。另外，注册部分的代码的维护是一个相对麻烦的事情，每一次支持新调用时，都要去维护一次注册列表。如果有调用被弃用了，是经常会忘记删项目的。runtime由于不存在注册过程，那就也不会产生维护的操作，维护成本就降低了。 由于通过runtime做到了服务的自动发现，拓展调用接口的任务就仅在于各自的模块，任何一次新接口添加，新业务添加，都不必去主工程做操作，十分透明。
* 在iOS领域里，一定是组件化的中间件为openURL提供服务，而不是openURL方式为组件化提供服务。如果在给App实施组件化方案的过程中是基于openURL的方案的话，有一个致命缺陷：非常规对象(不能被字符串化到URL中的对象，例如UIImage)无法参与本地组件间调度。 在本地调用中使用URL的方式其实是不必要的，如果业务工程师在本地间调度时需要给出URL，那么就不可避免要提供params，在调用时要提供哪些params是业务工程师很容易懵逼的地方。
* 为了支持传递非常规参数，蘑菇街的方案采用了protocol，这个会侵入业务。由于业务中的某个对象需要被调用，因此必须要符合某个可被调用的protocol，然而这个protocol又不存在于当前业务领域，于是当前业务就不得不依赖public Protocol。这对于将来的业务迁移是有非常大的影响的。

CTMediator的优点：

* 调用时，区分了本地应用调用和远程应用调用。本地应用调用为远程应用调用提供服务。
* 组件仅通过Action暴露可调用接口，模块与模块之间的接口被固化在了Target-Action这一层，避免了实施组件化的改造过程中，对Business的侵入，同时也提高了组件化接口的可维护性。
* 方便传递各种类型的参数。

**4.基于CTMediator的组件化方案，有哪些核心组成？**

* CTMediator中间件：集成就可以了
* 模块Target\_%@：模块的实现及提供对外的方法调用Action\_methodName，需要传参数时，都统一以NSDictionary\*的形式传入。
* CTMediator+%@扩展：扩展里声明了模块业务的对外接口，参数明确，这样外部调用者可以很容易理解如何调用接口。

# 2、大厂常问iOS面试题--Runtime篇

**1.Category 的实现原理？**

* Category 实际上是 Category\_t的结构体，在运行时，新添加的方法，都被以倒序插入到原有方法列表的最前面，所以不同的Category，添加了同一个方法，执行的实际上是最后一个。
* Category 在刚刚编译完的时候，和原来的类是分开的，只有在程序运行起来后，通过 Runtime ，Category 和原来的类才会合并到一起。

**2.isa指针的理解，对象的isa指针指向哪里？isa指针有哪两种类型？**

* isa 等价于 is kind of

实例对象的 isa 指向类对象

类对象的 isa 指向元类对象

元类对象的 isa 指向元类的基类

* isa 有两种类型

纯指针，指向内存地址

NON\_POINTER\_ISA，除了内存地址，还存有一些其他信息

**3.Objective-C 如何实现多重继承？**

Object-c的类没有多继承,只支持单继承,如果要实现多继承的话，可使用如下几种方式间接实现

* 通过组合实现

A和B组合，作为C类的组件

* 通过协议实现

C类实现A和B类的协议方法

* 消息转发实现

forwardInvocation:方法

**4.runtime 如何实现 weak 属性？**

weak 此特质表明该属性定义了一种「非拥有关系」(nonowning relationship)。为这种属性设置新值时，设置方法既不持有新值（新指向的对象），也不释放旧值（原来指向的对象）。

runtime 对注册的类，会进行内存布局，从一个粗粒度的概念上来讲，这时候会有一个 hash 表，这是一个全局表，表中是用 weak 指向的对象内存地址作为 key，用所有指向该对象的 weak 指针表作为 value。当此对象的引用计数为 0 的时候会 dealloc，假如该对象内存地址是 a，那么就会以 a 为 key，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。

runtime 如何实现 weak 属性具体流程大致分为 3 步：

* 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（指针可能原来指向着其他对象，这时候需要将该 weak 指针与旧对象解除绑定，会调用到 weak\_unregister\_no\_lock），如果指针指向的新对象非空，则创建对应的弱引用表，将 weak 指针与新对象进行绑定，会调用到 weak\_register\_no\_lock。在这个过程中，为了防止多线程中竞争冲突，会有一些锁的操作。
* 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。

**5.讲一下 OC 的消息机制**

* OC中的方法调用其实都是转成了objc\_msgSend函数的调用，给receiver（方法调用者）发送了一条消息（selector方法名）
* objc\_msgSend底层有3大阶段，消息发送（当前类、父类中查找）、动态方法解析、消息转发

**6.runtime具体应用**

* 利用关联对象（AssociatedObject）给分类添加属性
* 遍历类的所有成员变量（修改textfield的占位文字颜色、字典转模型、自动归档解档）
* 交换方法实现（交换系统的方法）
* 利用消息转发机制解决方法找不到的异常问题
* KVC 字典转模型

**7.runtime如何通过selector找到对应的IMP地址？**

每一个类对象中都一个对象方法列表（对象方法缓存）

* 类方法列表是存放在类对象中isa指针指向的元类对象中（类方法缓存）。
* 方法列表中每个方法结构体中记录着方法的名称,方法实现,以及参数类型，其实selector本质就是方法名称,通过这个方法名称就可以在方法列表中找到对应的方法实现。
* 当我们发送一个消息给一个NSObject对象时，这条消息会在对象的类对象方法列表里查找。
* 当我们发送一个消息给一个类时，这条消息会在类的Meta Class对象的方法列表里查找。

**8.简述下Objective-C中调用方法的过程**

Objective-C是动态语言，每个方法在运行时会被动态转为消息发送，即：objc\_msgSend(receiver, selector)，整个过程介绍如下：

* objc在向一个对象发送消息时，runtime库会根据对象的isa指针找到该对象实际所属的类
* 然后在该类中的方法列表以及其父类方法列表中寻找方法运行
* 如果，在最顶层的父类（一般也就NSObject）中依然找不到相应的方法时，程序在运行时会挂掉并抛出异常unrecognized selector sent to XXX
* 但是在这之前，objc的运行时会给出三次拯救程序崩溃的机会，这三次拯救程序奔溃的说明见问题《什么时候会报unrecognized selector的异常》中的说明。

**9.load和initialize的区别**

两者都会自动调用父类的，不需要super操作，且仅会调用一次（不包括外部显示调用).

* load和initialize方法都会在实例化对象之前调用，以main函数为分水岭，前者在main函数之前调用，后者在之后调用。这两个方法会被自动调用，不能手动调用它们。
* load和initialize方法都不用显示的调用父类的方法而是自动调用，即使子类没有initialize方法也会调用父类的方法，而load方法则不会调用父类。
* load方法通常用来进行Method Swizzle，initialize方法一般用于初始化全局变量或静态变量。
* load和initialize方法内部使用了锁，因此它们是线程安全的。实现时要尽可能保持简单，避免阻塞线程，不要再使用锁。

**10.怎么理解Objective-C是动态运行时语言。**

* 主要是将数据类型的确定由编译时,推迟到了运行时。这个问题其实浅涉及到两个概念,运行时和多态。
* 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。
* 多态:不同对象以自己的方式响应相同的消息的能力叫做多态。
* 意思就是假设生物类(life)都拥有一个相同的方法-eat;那人类属于生物,猪也属于生物,都继承了life后,实现各自的eat,但是调用是我们只需调用各自的eat方法。也就是不同的对象以自己的方式响应了相同的消 息(响应了eat这个选择器)。因此也可以说,运行时机制是多态的基础.

# 3. 大厂常问iOS面试题--Runloop篇

**1.Runloop 和线程的关系？**

* 一个线程对应一个 Runloop。
* 主线程的默认就有了 Runloop。
* 子线程的 Runloop 以懒加载的形式创建。
* Runloop 存储在一个全局的可变字典里，线程是 key ，Runloop 是 value。

**2.RunLoop的运行模式**

* RunLoop的运行模式共有5种，RunLoop只会运行在一个模式下，要切换模式，就要暂停当前模式，重写启动一个运行模式
* - kCFRunLoopDefaultMode, App的默认运行模式，通常主线程是在这个运行模式下运行
* - UITrackingRunLoopMode, 跟踪用户交互事件（用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他Mode影响）
* - kCFRunLoopCommonModes, 伪模式，不是一种真正的运行模式
* - UIInitializationRunLoopMode：在刚启动App时第进入的第一个Mode，启动完成后就不再使用
* - GSEventReceiveRunLoopMode：接受系统内部事件，通常用不到

**3.runloop内部逻辑？**

* 实际上 RunLoop 就是这样一个函数，其内部是一个 do-while 循环。当你调用 CFRunLoopRun() 时，线程就会一直停留在这个循环里；直到超时或被手动停止，该函数才会返回。

RunLoop

* 内部逻辑：
  1. 通知 Observer 已经进入了 RunLoop
  2. 通知 Observer 即将处理 Timer
  3. 通知 Observer 即将处理非基于端口的输入源（即将处理 Source0）
  4. 处理那些准备好的非基于端口的输入源（处理 Source0）
  5. 如果基于端口的输入源准备就绪并等待处理，请立刻处理该事件。转到第 9 步（处理 Source1）
  6. 通知 Observer 线程即将休眠
  7. 将线程置于休眠状态，直到发生以下事件之一
     + 事件到达基于端口的输入源（port-based input sources）(也就是 Source0)
     + Timer 到时间执行
     + 外部手动唤醒
     + 为 RunLoop 设定的时间超时
  8. 通知 Observer 线程刚被唤醒（还没处理事件）
  9. 处理待处理事件
     + 如果是 Timer 事件，处理 Timer 并重新启动循环，跳到第 2 步
     + 如果输入源被触发，处理该事件（文档上是 deliver the event）
     + 如果 RunLoop 被手动唤醒但尚未超时，重新启动循环，跳到第 2 步

**4.autoreleasePool 在何时被释放？**

* App启动后，苹果在主线程 RunLoop 里注册了两个 Observer，其回调都是 \_wrapRunLoopWithAutoreleasePoolHandler()。
* 第一个 Observer 监视的事件是 Entry(即将进入Loop)，其回调内会调用 \_objc\_autoreleasePoolPush() 创建自动释放池。其 order 是 -2147483647，优先级最高，保证创建释放池发生在其他所有回调之前。
* 第二个 Observer 监视了两个事件： BeforeWaiting(准备进入休眠) 时调用\_objc\_autoreleasePoolPop() 和 \_objc\_autoreleasePoolPush() 释放旧的池并创建新池；Exit(即将退出Loop) 时调用 \_objc\_autoreleasePoolPop() 来释放自动释放池。这个 Observer 的 order 是 2147483647，优先级最低，保证其释放池子发生在其他所有回调之后。
* 在主线程执行的代码，通常是写在诸如事件回调、Timer回调内的。这些回调会被 RunLoop 创建好的 AutoreleasePool 环绕着，所以不会出现内存泄漏，开发者也不必显示创建 Pool 了。

**5.GCD 在Runloop中的使用？**

* GCD由 子线程 返回到 主线程,只有在这种情况下才会触发 RunLoop。会触发 RunLoop 的 Source 1 事件。

**6.AFNetworking 中如何运用 Runloop?**

* AFURLConnectionOperation 这个类是基于 NSURLConnection 构建的，其希望能在后台线程接收 Delegate 回调。为此 AFNetworking 单独创建了一个线程，并在这个线程中启动了一个 RunLoop：
* + (void)networkRequestThreadEntryPoint:(id)\_\_unused object {
* @autoreleasepool {
* [[NSThread currentThread] setName:@"AFNetworking"];
* NSRunLoop \*runLoop = [NSRunLoop currentRunLoop];
* [runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];
* [runLoop run];
* }
* }
* + (NSThread \*)networkRequestThread {
* static NSThread \*\_networkRequestThread = nil;
* static dispatch\_once\_t oncePredicate;
* dispatch\_once(&oncePredicate, ^{
* \_networkRequestThread = [[NSThread alloc] initWithTarget:self selector:@selector(networkRequestThreadEntryPoint:) object:nil];
* [\_networkRequestThread start];
* });
* return \_networkRequestThread;
* }
* RunLoop 启动前内部必须要有至少一个 Timer/Observer/Source，所以 AFNetworking 在 [runLoop run] 之前先创建了一个新的 NSMachPort 添加进去了。通常情况下，调用者需要持有这个 NSMachPort (mach\_port) 并在外部线程通过这个 port 发送消息到 loop 内；但此处添加 port 只是为了让 RunLoop 不至于退出，并没有用于实际的发送消息。
* - (void)start {
* [self.lock lock];
* if ([self isCancelled]) {
* [self performSelector:@selector(cancelConnection) onThread:[[self class] networkRequestThread] withObject:nil waitUntilDone:NO modes:[self.runLoopModes allObjects]];
* } else if ([self isReady]) {
* self.state = AFOperationExecutingState;
* [self performSelector:@selector(operationDidStart) onThread:[[self class] networkRequestThread] withObject:nil waitUntilDone:NO modes:[self.runLoopModes allObjects]];
* }
* [self.lock unlock];
* }
* 当需要这个后台线程执行任务时，AFNetworking 通过调用 [NSObject performSelector:onThread:..] 将这个任务扔到了后台线程的 RunLoop 中。

**7.PerformSelector 的实现原理？**

* 当调用 NSObject 的 performSelecter:afterDelay: 后，实际上其内部会创建一个 Timer 并添加到当前线程的 RunLoop 中。所以如果当前线程没有 RunLoop，则这个方法会失效。
* 当调用 performSelector:onThread: 时，实际上其会创建一个 Timer 加到对应的线程去，同样的，如果对应线程没有 RunLoop 该方法也会失效。

**8.PerformSelector:afterDelay:这个方法在子线程中是否起作用？**

* 不起作用，子线程默认没有 Runloop，也就没有 Timer。可以使用 GCD的dispatch\_after来实现

**9.事件响应的过程？**

* 苹果注册了一个 Source1 (基于 mach port 的) 用来接收系统事件，其回调函数为 \_\_IOHIDEventSystemClientQueueCallback()。
* 当一个硬件事件(触摸/锁屏/摇晃等)发生后，首先由 IOKit.framework 生成一个 IOHIDEvent 事件并由 SpringBoard 接收。这个过程的详细情况可以参考这里。SpringBoard 只接收按键(锁屏/静音等)，触摸，加速，接近传感器等几种 Event，随后用 mach port 转发给需要的 App 进程。随后苹果注册的那个 Source1 就会触发回调，并调用 \_UIApplicationHandleEventQueue() 进行应用内部的分发。
* \_UIApplicationHandleEventQueue() 会把 IOHIDEvent 处理并包装成 UIEvent 进行处理或分发，其中包括识别 UIGesture/处理屏幕旋转/发送给 UIWindow 等。通常事件比如 UIButton 点击、touchesBegin/Move/End/Cancel 事件都是在这个回调中完成的。

**10.手势识别的过程？**

* 当 \_UIApplicationHandleEventQueue() 识别了一个手势时，其首先会调用 Cancel 将当前的 touchesBegin/Move/End 系列回调打断。随后系统将对应的 UIGestureRecognizer 标记为待处理。
* 苹果注册了一个 Observer 监测 BeforeWaiting (Loop即将进入休眠) 事件，这个 Observer 的回调函数是 \_UIGestureRecognizerUpdateObserver()，其内部会获取所有刚被标记为待处理的 GestureRecognizer，并执行GestureRecognizer 的回调。
* 当有 UIGestureRecognizer 的变化(创建/销毁/状态改变)时，这个回调都会进行相应处理。

**11.CADispalyTimer和Timer哪个更精确**

CADisplayLink 更精确

* iOS设备的屏幕刷新频率是固定的，CADisplayLink在正常情况下会在每次刷新结束都被调用，精确度相当高。
* NSTimer的精确度就显得低了点，比如NSTimer的触发时间到的时候，runloop如果在阻塞状态，触发时间就会推迟到下一个runloop周期。并且 NSTimer新增了tolerance属性，让用户可以设置可以容忍的触发的时间的延迟范围。
* CADisplayLink使用场合相对专一，适合做UI的不停重绘，比如自定义动画引擎或者视频播放的渲染。NSTimer的使用范围要广泛的多，各种需要单次或者循环定时处理的任务都可以使用。在UI相关的动画或者显示内容使用 CADisplayLink比起用NSTimer的好处就是我们不需要在格外关心屏幕的刷新频率了，因为它本身就是跟屏幕刷新同步的。

# 4. 大厂常问iOS面试题--多线程篇

**1.进程与线程**

* 进程：

1.进程是一个具有一定独立功能的程序关于某次数据集合的一次运行活动，它是操作系统分配资源的基本单元.

2.进程是指在系统中正在运行的一个应用程序，就是一段程序的执行过程,我们可以理解为手机上的一个app.

3.每个进程之间是独立的，每个进程均运行在其专用且受保护的内存空间内，拥有独立运行所需的全部资源

* 线程

1.程序执行流的最小单元，线程是进程中的一个实体.

2.一个进程要想执行任务,必须至少有一条线程.应用程序启动的时候，系统会默认开启一条线程,也就是主线程

* 进程和线程的关系

1.线程是进程的执行单元，进程的所有任务都在线程中执行

2.线程是 CPU 分配资源和调度的最小单位

3.一个程序可以对应多个进程(多进程),一个进程中可有多个线程,但至少要有一条线程

4.同一个进程内的线程共享进程资源

**2.什么是多线程？**

* 多线程的实现原理：事实上，同一时间内单核的CPU只能执行一个线程，多线程是CPU快速的在多个线程之间进行切换（调度），造成了多个线程同时执行的假象。
* 如果是多核CPU就真的可以同时处理多个线程了。
* 多线程的目的是为了同步完成多项任务，通过提高系统的资源利用率来提高系统的效率。

**3.多线程的优点和缺点**

* 优点:

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

* 缺点:

开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果开启大量的线程，会占用大量的内存空间，降低程序的性能

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

**4.多线程的 并行 和 并发 有什么区别？**

* 并行：充分利用计算机的多核，在多个线程上同步进行
* 并发：在一条线程上通过快速切换，让人感觉在同步进行

**5.iOS中实现多线程的几种方案，各自有什么特点？**

* NSThread 面向对象的，需要程序员手动创建线程，但不需要手动销毁。子线程间通信很难。
* GCD c语言，充分利用了设备的多核，自动管理线程生命周期。比NSOperation效率更高。
* NSOperation 基于gcd封装，更加面向对象，比gcd多了一些功能。

**6.多个网络请求完成后执行下一步**

* 使用GCD的dispatch\_group\_t

创建一个dispatch\_group\_t

每次网络请求前先dispatch\_group\_enter,请求回调后再dispatch\_group\_leave，enter和leave必须配合使用，有几次enter就要有几次leave，否则group会一直存在。

当所有enter的block都leave后，会执行dispatch\_group\_notify的block。

NSString \*str = @"http://xxxx.com/";

NSURL \*url = [NSURL URLWithString:str];

NSURLRequest \*request = [NSURLRequest requestWithURL:url];

NSURLSession \*session = [NSURLSession sharedSession];

dispatch\_group\_t downloadGroup = dispatch\_group\_create();

for (int i=0; i<10; i++) {

dispatch\_group\_enter(downloadGroup);

NSURLSessionDataTask \*task = [session dataTaskWithRequest:request completionHandler:^(NSData \* \_Nullable data, NSURLResponse \* \_Nullable response, NSError \* \_Nullable error) {

NSLog(@"%d---%d",i,i);

dispatch\_group\_leave(downloadGroup);

}];

[task resume];

}

dispatch\_group\_notify(downloadGroup, dispatch\_get\_main\_queue(), ^{

NSLog(@"end");

});

* 使用GCD的信号量dispatch\_semaphore\_t

dispatch\_semaphore信号量为基于计数器的一种多线程同步机制。如果semaphore计数大于等于1，计数-1，返回，程序继续运行。如果计数为0，则等待。dispatch\_semaphore\_signal(semaphore)为计数+1操作,dispatch\_semaphore\_wait(sema, DISPATCH\_TIME\_FOREVER)为设置等待时间，这里设置的等待时间是一直等待。

创建semaphore为0，等待，等10个网络请求都完成了，dispatch\_semaphore\_signal(semaphore)为计数+1，然后计数-1返回

NSString \*str = @"http://xxxx.com/";

NSURL \*url = [NSURL URLWithString:str];

NSURLRequest \*request = [NSURLRequest requestWithURL:url];

NSURLSession \*session = [NSURLSession sharedSession];

dispatch\_semaphore\_t sem = dispatch\_semaphore\_create(0);

for (int i=0; i<10; i++) {

NSURLSessionDataTask \*task = [session dataTaskWithRequest:request completionHandler:^(NSData \* \_Nullable data, NSURLResponse \* \_Nullable response, NSError \* \_Nullable error) {

NSLog(@"%d---%d",i,i);

count++;

if (count==10) {

dispatch\_semaphore\_signal(sem);

count = 0;

}

}];

[task resume];

}

dispatch\_semaphore\_wait(sem, DISPATCH\_TIME\_FOREVER);

dispatch\_async(dispatch\_get\_main\_queue(), ^{

NSLog(@"end");

});

**7.多个网络请求顺序执行后执行下一步**

* 使用信号量semaphore

每一次遍历，都让其dispatch\_semaphore\_wait(sem, DISPATCH\_TIME\_FOREVER)，这个时候线程会等待，阻塞当前线程，直到dispatch\_semaphore\_signal(sem)调用之后

NSString \*str = @"http://www.jianshu.com/p/6930f335adba";

NSURL \*url = [NSURL URLWithString:str];

NSURLRequest \*request = [NSURLRequest requestWithURL:url];

NSURLSession \*session = [NSURLSession sharedSession];

dispatch\_semaphore\_t sem = dispatch\_semaphore\_create(0);

for (int i=0; i<10; i++) {

NSURLSessionDataTask \*task = [session dataTaskWithRequest:request completionHandler:^(NSData \* \_Nullable data, NSURLResponse \* \_Nullable response, NSError \* \_Nullable error) {

NSLog(@"%d---%d",i,i);

dispatch\_semaphore\_signal(sem);

}];

[task resume];

dispatch\_semaphore\_wait(sem, DISPATCH\_TIME\_FOREVER);

}

dispatch\_async(dispatch\_get\_main\_queue(), ^{

NSLog(@"end");

});

**8.异步操作两组数据时, 执行完第一组之后, 才能执行第二组**

* 这里使用dispatch\_barrier\_async栅栏方法即可实现
* dispatch\_queue\_t queue = dispatch\_queue\_create("test", DISPATCH\_QUEUE\_CONCURRENT);
* dispatch\_async(queue, ^{
* NSLog(@"第一次任务的主线程为: %@", [NSThread currentThread]);
* });
* dispatch\_async(queue, ^{
* NSLog(@"第二次任务的主线程为: %@", [NSThread currentThread]);
* });
* dispatch\_barrier\_async(queue, ^{
* NSLog(@"第一次任务, 第二次任务执行完毕, 继续执行");
* });
* dispatch\_async(queue, ^{
* NSLog(@"第三次任务的主线程为: %@", [NSThread currentThread]);
* });
* dispatch\_async(queue, ^{
* NSLog(@"第四次任务的主线程为: %@", [NSThread currentThread]);
* });

**9.多线程中的死锁？**

死锁是由于多个线程（进程）在执行过程中，因为争夺资源而造成的互相等待现象，你可以理解为卡主了。产生死锁的必要条件有四个：

* 互斥条件 ： 指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
* 请求和保持条件 ： 指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
* 不可剥夺条件 ： 指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
* 环路等待条件 ： 指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

最常见的就是 同步函数 + 主队列 的组合，本质是队列阻塞。

dispatch\_sync(dispatch\_get\_main\_queue(), ^{

NSLog(@"2");

});

NSLog(@"1");

// 什么也不会打印，直接报错

**10.GCD执行原理？**

* GCD有一个底层线程池，这个池中存放的是一个个的线程。之所以称为“池”，很容易理解出这个“池”中的线程是可以重用的，当一段时间后这个线程没有被调用胡话，这个线程就会被销毁。注意：开多少条线程是由底层线程池决定的（线程建议控制再3~5条），池是系统自动来维护，不需要我们程序员来维护（看到这句话是不是很开心？） 而我们程序员需要关心的是什么呢？我们只关心的是向队列中添加任务，队列调度即可。
* 如果队列中存放的是同步任务，则任务出队后，底层线程池中会提供一条线程供这个任务执行，任务执行完毕后这条线程再回到线程池。这样队列中的任务反复调度，因为是同步的，所以当我们用currentThread打印的时候，就是同一条线程。
* 如果队列中存放的是异步的任务，（注意异步可以开线程），当任务出队后，底层线程池会提供一个线程供任务执行，因为是异步执行，队列中的任务不需等待当前任务执行完毕就可以调度下一个任务，这时底层线程池中会再次提供一个线程供第二个任务执行，执行完毕后再回到底层线程池中。
* 这样就对线程完成一个复用，而不需要每一个任务执行都开启新的线程，也就从而节约的系统的开销，提高了效率。在iOS7.0的时候，使用GCD系统通常只能开5--8条线程，iOS8.0以后，系统可以开启很多条线程，但是实在开发应用中，建议开启线程条数：3--5条最为合理。

# 6. 大厂常问iOS面试题--性能优化篇

**1.造成tableView卡顿的原因有哪些？**

* 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。

**2.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

* iOS 保持界面流畅的技巧

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

**3.APP启动时间应从哪些方面优化？**

App启动时间可以通过xcode提供的工具来度量，在Xcode的Product->Scheme-->Edit Scheme->Run->Auguments中，将环境变量DYLD\_PRINT\_STATISTICS设为YES，优化需以下方面入手

* dylib loading time

核心思想是减少dylibs的引用

合并现有的dylibs（最好是6个以内）

使用静态库

* rebase/binding time

核心思想是减少DATA块内的指针

减少Object C元数据量，减少Objc类数量，减少实例变量和函数（与面向对象设计思想冲突）

减少c++虚函数

多使用Swift结构体（推荐使用swift）

* ObjC setup time

核心思想同上，这部分内容基本上在上一阶段优化过后就不会太过耗时

initializer time

* 使用initialize替代load方法

减少使用c/c++的attribute((constructor))；推荐使用dispatch\_once() pthread\_once() std:once()等方法

推荐使用swift

不要在初始化中调用dlopen()方法，因为加载过程是单线程，无锁，如果调用dlopen则会变成多线程，会开启锁的消耗，同时有可能死锁

不要在初始化中创建线程

**4.如何降低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（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

**5.如何检测离屏渲染与优化**

* 检测，通过勾选Xcode的Debug->View Debugging-->Rendering->Run->Color Offscreen-Rendered Yellow项。
* 优化，如阴影，在绘制时添加阴影的路径

**6.怎么检测图层混合**

1、模拟器debug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：

* 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
* 如无特殊需要，不要设置低于1的alpha值
* 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

**7.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：

Laek 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。

# 7. socket 连接和 Http 连接的区别？

**http 是基于 socket 之上的。socket 是一套完整的 tcp,udp协议的接口。**

* HTTP协议：简单对象访问协议，对应于应用层，HTTP协议是基于TCP连接的。
  + tcp协议：对应于传输层。
  + ip协议：对应于网络层。

**TCP/IP是传输层协议，主要解决数据如何在网络中传输，而HTTP协议是应用层协议，主要解决如何包装数据。**

Socket是对TCP/IP 协议的封装，它本身不是协议，而是一个调用接口，通过Socket，我们才能使用TCP/IP协议。

* http连接：就是所谓的短连接，即客户端向服务器端发送一次请求，服务器端响应后连接即会断掉。
* socket连接：就是所谓的长连接，理论上客户端和服务器端一旦建立起连接将不会主动断掉，但是由于各种环境因素可能会使连接断开。

http是客户端用http协议进行请求，发送请求时候需要封装http请求头，并绑定请求的数据，服务器一般有web服务器配合。http请求方式为客户端主动发起请求，服务器才能给响应，一次请求完毕后则断开连接以节省资源。服务器不能主动给客户端响应。iPhone主要使用的类是NSUrlConnection。socket是客户端跟服务器直接使用socket“套接字”进行拼接，并没有规定连接后断开，所以客户端和服务器可以保持连接，双方都可以主动发送数据。一般在游戏开发或者股票开发这种即时性很强的并且保持发送数据量比较大的场合使用。主要类是CFSocketRef。

* UDP：是用户数据报协议：主要用在实时性要求高以及对质量相对较弱的地方，但面对现在高质量的线路容易丢包。
* TCP：是传输控制协议，是面向连接的，，运行环境必然要求其可靠性不可丢失包有良好的拥塞控制机制。

# 9. load 和 initialize 的区别？

+load

* 1、只要程序启动就会将所有类的代码加载到内存中（在main函数执行之前）, 放到代码区(无论该类有没有被使用到都会被调用)
* 2、+load方法会在当前类被加载到内存的时候调用, 有且仅会调用一次
* 3、当父类和子类都实现+load方法时, 会先调用父类的+load方法, 再调用子类的+load方法
* 4、先加载原始类，再加载分类的+load方法
* 5、当子类未实现+load方法时，不会调用父类的+load方法
* 6、多个类都实现+load方法，+load方法的调用顺序，与Compile Sources中出现的顺序一致

[load方法在Apple官方文档中的描述](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.apple.com%2Fdocumentation%2Fobjectivec%2Fnsobject%2F1418815-load%3FpreferredLanguage%3Docc)

＋initialize

* 1、当类第一次被使用的时候就会调用(创建类对象的时候)
* 2、initialize方法在整个程序的运行过程中只会被调用一次, 无论你使用多少次这个类都只会调用一次
* 3、initialize用于对某一个类进行一次性的初始化
* 4、先调用父类的initialize再调用子类的initialize
* 5、当子类未实现initialize方法时，会把父类的实现继承过来调用一遍，再次之前父类的initialize方法会被优先调用一次
* 6、当有多个Category都实现了initialize方法，会覆盖类中的方法，只执行一个(会执行Compile Sources 列表中最后一个Category 的initialize方法)

# 10. 谈谈你对事件的传递链和响应链的理解?

* 一：响应者链  
  UIResponser包括了各种Touch message 的处理，比如开始，移动，停止等等。常见的 UIResponser 有 UIView及子类，UIViController,APPDelegate，UIApplication等等。

回到响应链，响应链是由UIResponser组成的，那么是按照哪种规则形成的。

* A： 程序启动  
  UIApplication会生成一个单例，并会关联一个APPDelegate。APPDelegate作为整个响应链的根建立起来，而``UIApplication会将自己与这个单例链接，即UIApplication的nextResponser(下一个事件处理者)为APPDelegate`。
* B：创建UIWindow  
  程序启动后，任何的UIWindow被创建时，UIWindow内部都会把nextResponser设置为UIApplication单例。UIWindow初始化rootViewController,rootViewController的nextResponser会设置为UIWindow
* C：UIViewController初始化  
  loadView, VC的view的nextResponser会被设置为VC.
* D：addSubView  
  addSubView操作过程中，如果子subView不是VC的View,那么subView的nextResponser会被设置为superView。如果是VC的View,那就是 subView -> subView.VC ->superView如果在中途，subView.VC被释放，就会变成subView.nextResponser = superView

我们使用一个现实场景来解释这个问题：当一个用点击屏幕上的一个按钮，这个过程具体发生了什么。

* 1. 用户触摸屏幕，系统硬件进程会获取到这个点击事件，将事件简单处理封装后存到系统中，由于硬件检测进程和当前App进程是两个进程，所以进程两者之间传递事件用的是端口通信。硬件检测进程会将事件放到APP检测的那个端口。
* 2.APP启动主线程RunLoop会注册一个端口事件，来检测触摸事件的发生。当事件到达，系统会唤起当前APP主线程的RunLoop。来源就是App主线程事件，主线程会分析这个事件。
* 3.最后，系统判断该次触摸是否导致了一个新的事件, 也就是说是否是第一个手指开始触碰，如果是，系统会先从响应网中 寻找响应链。如果不是，说明该事件是当前正在进行中的事件产生的一个Touch message， 也就是说已经有保存好的响应链
* 二：事件传递链

通过两种方法来做这个事情。

// 先判断点是否在View内部，然后遍历subViews

- (nullable UIView \*)hitTest:(CGPoint)point withEvent:(nullable UIEvent \*)event;

//判断点是否在这个View内部

- (BOOL)pointInside:(CGPoint)point withEvent:(nullable UIEvent \*)event; // default returns YES if point is in bounds

* A: 流程
* 1：先判断该层级是否能够响应（1.alpha>0.01 2.userInteractionEnabled == YES 3.hidden = NO）
* 2：判断改点是否在view内部,
* 3：如果在那么遍历子view继续返回可响应的view，直到没有。
* B：常见问题
* 父view设置为不可点击，子view可以点击吗
* 不可以，hit test 到父view就截止了
* 子view设置view不可点击不影响父类点击
* 同父view覆盖不影响点击
* 手势对responder方法的影响
* C：实际用法
* 点一一个圆形控件，如何实现只点击圆形区域有效，重载pointInside。此时可将外部的点也判断为内部的点，反之也可以。
* 事件响应链在复杂功能界面进行不同控件间的通信，简便某些场景下优于代理和block

# 11. block 为什么能够捕获外界变量？ \_\_block做了什么事？

研究Block的捕获外部变量就要除去函数参数这一项，下面一一根据这4种变量类型的捕获情况进行分析。

* 自动变量
* 静态变量
* 静态全局变量
* 全局变量

首先 全局变量global\_i和 静态全局变量static\_global\_j 的值增加，以及它们被Block捕获进去，这一点很好理解，因为是全局的，作用域很广，所以Block捕获了它们进去之后，在Block里面进行++操作，Block结束之后，它们的值依旧可以得以保存下来。

struct \_\_main\_block\_impl\_0 {

struct \_\_block\_impl impl;

struct \_\_main\_block\_desc\_0\* Desc;

\_\_Block\_byref\_a\_0 \*a; // by ref

\_\_main\_block\_impl\_0(void \*fp, struct \_\_main\_block\_desc\_0 \*desc, \_\_Block\_byref\_a\_0 \*\_a, int flags=0) : a(\_a->\_\_forwarding) {

impl.isa = &\_NSConcreteStackBlock;

impl.Flags = flags;

impl.FuncPtr = fp;

Desc = desc;

}

};

\_\_main\_block\_impl\_0结构体 就是这样把自动变量捕获进来的。也就是说，在执行 Block 语法的时候，Block 语法表达式所使用的自动变量的值是被保存进了Block的结构体实例中，也就是 Block 自身中。

这里值得说明的一点是，如果Block外面还有很多自动变量，静态变量，等等，这些变量在Block里面并不会被使用到。那么这些变量并不会被Block捕获进来，也就是说并不会在构造函数里面传入它们的值。

Block捕获外部变量仅仅只捕获Block闭包里面会用到的值，其他用不到的值，它并不会去捕获。

# 13. 如何关闭默认的KVO的默认实现，KVO的实现原理？

所谓的“手动触发”是区别于“自动触发”：

* 自动触发是指类似这种场景：在注册 KVO 之前设置一个初始值，注册之后，设置一个不一样的值，就可以触发了。

想知道如何手动触发，必须知道自动触发 KVO 的原理：

键值观察通知依赖于 NSObject 的两个方法: willChangeValueForKey:和 didChangevlueForKey:。在一个被观察属性发生改变之前， willChangeValueForKey: 一定会被调用，这就 会记录旧的值。而当改变发生后， didChangeValueForKey: 会被调用，继而 observeValueForKey:ofObject:change:context: 也会被调用。如果可以手动实现这些调用，就可以实现“手动触发”了。

当你观察一个对象时，一个新的类会被动态创建。这个类继承自该对象的原本的类，并重写了被观察属性的 setter 方法。重写的 setter 方法会负责在调用原 setter 方法之前和之后，通知所有观察对象：值的更改。最后通过 isa 混写（isa-swizzling） 把这个对象的 isa 指针 ( isa 指针告诉 Runtime 系统这个对象的类是什么 ) 指向这个新创建的子类，对象就神奇的变成了新创建的子类的实例。

如下所示：

# 14、 系统对象的 copy 与 mutableCopy 方法？

不管是集合类对象（NSArray、NSDictionary、NSSet ... 之类的对象），还是非集合类对象（NSString, NSNumber ... 之类的对象），接收到copy和mutableCopy消息时，都遵循以下准则：

* 1. copy 返回的是不可变对象（immutableObject）；如果用copy返回值调用mutable对象的方法就会crash。
  2. mutableCopy 返回的是可变对象（mutableObject）。

**一、非集合类对象的copy与mutableCopy**  
在非集合类对象中，对不可变对象进行copy操作，是指针复制，mutableCopy操作是内容复制；  
对可变对象进行copy和mutableCopy都是内容复制。用代码简单表示如下：

NSString \*str = @"hello word!";

NSString \*strCopy = [str copy] // 指针复制，strCopy与str的地址一样

NSMutableString \*strMCopy = [str mutableCopy] // 内容复制，strMCopy与str的地址不一样

NSMutableString \*mutableStr = [NSMutableString stringWithString: @"hello word!"];

NSString \*strCopy = [mutableStr copy] // 内容复制

NSMutableString \*strMCopy = [mutableStr mutableCopy] // 内容复制

**二、集合类对象的copy与mutableCopy (同上)**  
在集合类对象中，对不可变对象进行copy操作，是指针复制，mutableCopy操作是内容复制；  
对可变对象进行copy和mutableCopy都是内容复制。但是：集合对象的内容复制仅限于对象本身，对集合内的对象元素仍然是指针复制。(即单层内容复制)

NSArray \*arr = @[@[@"a", @"b"], @[@"c", @"d"];

NSArray \*copyArr = [arr copy]; // 指针复制

NSMutableArray \*mCopyArr = [arr mutableCopy]; //单层内容复制

NSMutableArray \*array = [NSMutableArray arrayWithObjects:[NSMutableString stringWithString:@"a"],@"b",@"c",nil];

NSArray \*copyArr = [mutableArr copy]; // 单层内容复制

NSMutableArray \*mCopyArr = [mutableArr mutableCopy]; // 单层内容复制

**【总结一句话】**：  
只有对不可变对象进行copy操作是指针复制（浅复制），其它情况都是内容复制（深复制）！

# 16. 谈下Objective C都有哪些锁机制，你一般用哪个？

1）NSLock

iOS中对于资源抢占的问题可以使用同步锁NSLock来解决，使用时把需要加锁的代码（以后暂时称这段代码为”加锁代码“）放到NSLock的lock和unlock之间，一个线程A进入加锁代码之后由于已经加锁，另一个线程B就无法访问，只有等待前一个线程A执行完加锁代码后解锁，B线程才能访问加锁代码。

2）@synchronized代码块

使用@synchronized解决线程同步问题相比较NSLock要简单一些，日常开发中也更推荐使用此方法。

3）使用GCD解决资源抢占问题

在GCD中提供了一种信号机制，也可以解决资源抢占问题（和同步锁的机制并不一样）。

4）扩展--控制线程通信

由于线程的调度是透明的，程序有时候很难对它进行有效的控制，为了解决这个问题iOS提供了NSCondition来控制线程通信(同前面GCD的信号机制类似)。

5）iOS中的其他锁

NSRecursiveLock ：递归锁，有时候“加锁代码”中存在递归调用，递归开始前加锁，递归调用开始后会重复执行此方法以至于反复执行加锁代码最终造成死锁，这个时候可以使用递归锁来解决。使用递归锁可以在一个线程中反复获取锁而不造成死锁，这个过程中会记录获取锁和释放锁的次数，只有最后两者平衡锁才被最终释放。

NSDistributedLock：分布锁，它本身是一个互斥锁，基于文件方式实现锁机制，可以跨进程访问。

pthread\_mutex\_t：同步锁，基于C语言的同步锁机制，使用方法与其他同步锁机制类似。

# 17. 数据库建表的时候索引有什么用？

创建索引可以大大提高系统的性能。  
第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。  
第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。  
第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。  
第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。  
第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

# 18. 线程与进程的区别和联系?

 一个程序至少要有进城,一个进程至少要有一个线程.

 进程:资源分配的最小独立单元,进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

 线程:进程下的一个分支,是进程的实体,是CPU调度和分派的基本单元,它是比进程更小的能独立运行的基本单位,线程自己基本不拥有系统资源,只拥有一点在运行中必不可少的资源(程序计数器、一组寄存器、栈)，但是它可与同属一个进程的其他线程共享进程所拥有的全部资源。

 进程和线程都是由操作系统所体会的程序运行的基本单元，系统利用该基本单元实现系统对应用的并发性。

 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

 但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

# 19. 堆和栈的区别？

 栈区(stack)由编译器自动分配释放 ,存放方法(函数)的参数值, 局部变量的值等，栈是向低地址扩展的数据结构，是一块连续的内存的区域。即栈顶的地址和栈的最大容量是系统预先规定好的。

 堆区(heap)一般由程序员分配释放, 若程序员不释放,程序结束时由OS回收，向高地址扩展的数据结构，是不连续的内存区域，从而堆获得的空间比较灵活。

 碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出.

 分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

 分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的。

 全局区(静态区)(static),全局变量和静态变量的存储是放在一块 的,初始化的全局变量和静态变量在一块区域, 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后有系统释放。

 文字常量区—常量字符串就是放在这里的。程序结束后由系统释放。

 程序代码区—存放函数体的二进制代码

# 20. 键路径(keyPath)、键值编码（KVC）、键值观察（KVO）？

**键路径**  
在一个给定的实体中,同一个属性的所有值具有相同的数据类型。  
键-值编码技术用于进行这样的查找—它是一种间接访问对象属性的机制。 - 键路径是一个由用点作分隔符的键组成的字符串,用于指定一个连接在一起的对象性质序列。第一个键的性质是由先前的性质决定的,接下来每个键的值也是相对于其前面的性质。  
键路径使您可以以独立于模型实现的方式指定相关对象的性质。通过键路径,您可以指定对象图中的一个任意深度的路径,使其指向相关对象的特定属性。

**键值编码KVC**  
键值编码是一种间接访问对象的属性使用字符串来标识属性，而不是通过调用存取方法，直接或通过实例变量访问的机制，非对象类型的变量将被自动封装或者解封成对象，很多情况下会简化程序代码；  
KVC的缺点：一旦使用 KVC 你的编译器无法检查出错误,即不会对设置的键、键路径进行错误检查,且执行效率要低于合成存取器方法和自定的 setter 和 getter 方法。因为使用 KVC 键值编码,它必须先解析字符串,然后在设置或者访问对象的实例变量。

**键值观察KVO**

键值观察机制是一种能使得对象获取到其他对象属性变化的通知 ，极大的简化了代码。  
实现 KVO 键值观察模式,被观察的对象必须使用 KVC 键值编码来修 改它的实例变量,这样才能被观察者观察到。因此,KVC是KVO的基础。

**Demo**  
比如自定义的一个button

[self addObserver:self forKeyPath:@"highlighted" options:0 context:nil];

#pragma mark KVO

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

{

if ([keyPath isEqualToString:@"highlighted"] ) {

[self setNeedsDisplay];

}

}

对于系统是根据keypath去取的到相应的值发生改变，理论上来说是和kvc机制的道理是一样的。

KVC机制通过key找到value的原理

* 当通过KVC调用对象时，比如：[self valueForKey:@”someKey”]时，程序会自动试图通过下面几种不同的方式解析这个调用。
* 首先查找对象是否带有someKey这个方法，如果没找到，会继续查找对象是否带有someKey这个实例变量（iVar），如果还没有找到，程序会继续试图调用 -(id) valueForUndefinedKey:这个方法。如果这个方法还是没有被实现的话，程序会抛出一个NSUndefinedKeyException异常错误。
* 补充：KVC查找方法的时候，不仅仅会查找someKey这个方法，还会查找getsomeKey这个方法，前面加一个get，或者\_someKey以\_getsomeKey这几种形式。同时，查找实例变量的时候也会不仅仅查找someKey这个变量，也会查找\_someKey这个变量是否存在。
* 设计valueForUndefinedKey:方法的主要目的是当你使用-(id)valueForKey方法从对象中请求值时，对象能够在错误发生前，有最后的机会响应这个请求。

# 21. TCP和UDP的区别于联系

* TCP为传输控制层协议，为面向连接、可靠的、点到点的通信；
* UDP为用户数据报协议，非连接的不可靠的点到多点的通信；
* TCP侧重可靠传输，UDP侧重快速传输。

# 22. AFNetworking 底层原理分析

AFNetworking主要是对NSURLSession和NSURLConnection(iOS9.0废弃)的封装,其中主要有以下类:  
1). AFHTTPRequestOperationManager：内部封装的是 NSURLConnection, 负责发送网络请求, 使用最多的一个类。(3.0废弃)  
2). AFHTTPSessionManager：内部封装是 NSURLSession, 负责发送网络请求,使用最多的一个类。  
3). AFNetworkReachabilityManager：实时监测网络状态的工具类。当前的网络环境发生改变之后,这个工具类就可以检测到。  
4). AFSecurityPolicy：网络安全的工具类, 主要是针对 HTTPS 服务。  
5). AFURLRequestSerialization：序列化工具类,基类。上传的数据转换成JSON格式  
(AFJSONRequestSerializer).使用不多。  
6). AFURLResponseSerialization：反序列化工具类;基类.使用比较多:  
7). AFJSONResponseSerializer; JSON解析器,默认的解析器.  
8). AFHTTPResponseSerializer; 万能解析器; JSON和XML之外的数据类型,直接返回二进制数据.对服务器返回的数据不做任何处理.  
9). AFXMLParserResponseSerializer; XML解析器;

# 23. 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

**为什么要有离屏渲染?**

大家高中物理应该学过显示器是如何显示图像的：需要显示的图像经过CRT电子枪以极快的速度一行一行的扫描，扫描出来就呈现了一帧画面，随后电子枪又会回到初始位置循环扫描，形成了我们看到的图片或视频。

为了让显示器的显示跟视频控制器同步，当电子枪新扫描一行的时候，准备扫描的时发送一个水平同步信号(HSync信号)，显示器的刷新频率就是HSync信号产生的频率。然后CPU计算好frame等属性，将计算好的内容交给GPU去渲染，GPU渲染好之后就会放入帧缓冲区。然后视频控制器会按照HSync信号逐行读取帧缓冲区的数据，经过可能的数模转换传递给显示器，就显示出来了。具体的大家自行查找资料或询问相关专业人士，这里只参考网上资料做一个简单的描述。

离屏渲染的代价很高，想要进行离屏渲染，首选要创建一个新的缓冲区，屏幕渲染会有一个上下文环境的一个概念，离屏渲染的整个过程需要切换上下文环境，先从当前屏幕切换到离屏，等结束后，又要将上下文环境切换回来。这也是为什么会消耗性能的原因了。

由于垂直同步的机制，如果在一个 HSync 时间内，CPU 或者 GPU 没有完成内容提交，则那一帧就会被丢弃，等待下一次机会再显示，而这时显示屏会保留之前的内容不变。这就是界面卡顿的原因。

**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

# 24. UIWindow，UIView，CALayer的区别

**1. UIWindow**

@interface UIWindow : UIView

@property(nonatomic) UIWindowLevel windowLevel; // default = 0.0

@property(nonatomic,readonly,getter=isKeyWindow) BOOL keyWindow;

- (void)becomeKeyWindow; // override point for subclass. Do not call directly

- (void)resignKeyWindow; // override point for subclass. Do not call directly

- (void)makeKeyWindow;

- (void)makeKeyAndVisible; // convenience. most apps call this to show the main window and also make it key. otherwise use view hidden property

@property(nullable, nonatomic,strong) UIViewController \*rootViewController NS\_AVAILABLE\_IOS(4\_0); // default is nil

@end

继承自UIView，是一种特殊的 UIView，通常在一个app中只会有一个keyUIWindow。

iOS程序启动完毕后，创建的第一个视图控件就是UIWindow，接着创建控制器的view，最后将控制器的view添加到UIWindow上，于是控制器的view就显示在屏幕上了

主要作用是提供一个区域用来显示UIView；将事件分发给UIView；与UIViewController一起处理屏幕的旋转事件。

**2. UIView**

@interface UIView : UIResponder <NSCoding, UIAppearance, UIAppearanceContainer, UIDynamicItem, UITraitEnvironment, UICoordinateSpace, UIFocusItem, UIFocusItemContainer, CALayerDelegate>

@property(nonatomic,readonly,strong) CALayer \*layer;

@end

@interface UIResponder : NSObject <UIResponderStandardEditActions>

继承自UIResponder，间接继承自NSObject，主要是用来构建用户界面的，并且可以响应事件。

对于UIView，侧重于对内容的显示管理；其实是相对于CALayer的高层封装。

**3. CALayer**

@interface CALayer : NSObject <NSSecureCoding, CAMediaTiming>

直接继承自NSObject，所以不能响应事件

其实就是一个图层，UIView之所以能显示在屏幕上，主要是它内部有一个CALayer对象。在创建UIView时，它内部会自动创建一个图层，当UIView需要显示在屏幕上的时候，会调用drawRect:方法进行绘图，并且会将所有内容绘制到自己的图层上，绘图完毕后，系统会将图层拷贝到屏幕上，这样完成UIView的显示。

* layer给view提供了基础设施，使得绘制内容和呈现更高效动画更容易、更低耗
* layer不参与view的事件处理、不参与响应链

# 25、weak修饰的释放则自动被置为nil的实现原理

 Runtime维护着一个Weak表，用于存储指向某个对象的所有Weak指针

 Weak表是Hash表，Key是所指对象的地址，Value是Weak指针地址的数组

 在对象被回收的时候，经过层层调用，会最终触发下面的方法将所有Weak指针的值设为nil。\* runtime源码，objc-weak.m 的 arr\_clear\_deallocating 函数

 weak指针的使用涉及到Hash表的增删改查，有一定的性能开销.

# 26、HTTPS的加密原理

 服务器端用非对称加密(RSA)生成公钥和私钥

 然后把公钥发给客户端, 服务器则保存私钥

 客户端拿到公钥后, 会生成一个密钥, 这个密钥就是将来客户端和服务器用来通信的钥匙

 然后客户端用公钥对密钥进行加密, 再发给服务器

 服务器拿到客户端发来的加密后的密钥后, 再使用私钥解密密钥, 到此双方都获得通信的钥匙