The interview questions

1: 讲讲你对atomic & nonatomic的理解

答：nonatomic：非原子操作，决定编译器生成的setter getter是否是原子操作,不会为setter方法加锁。系统自动生成的 getter/setter 方法不一样。如果自己写 getter/setter ，那 atomic/nonatomic/retain/assign/copy 这些关键字只起提示作用，写不写都一样。线程不安全，如有两个线程访问同一个属性，会出现无法预料的结果.
atomic 和 nonatomic 用来决定编译器生成的 getter 和 setter 是否为原子操作。在多线程环境下，原子操作是必要的，否则有可能引起错误的结果。
对于 atomic 的属性， atomic 表示多线程安全， atomic 意为操作是原子的，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。
比如，线程 A 的 getter 方法运行到一半，线程 B 调用了 setter ： 那么线程 A 的 getter 还是能得到一个完好无损的对象。意味着只有一个线程访问实例变量 (生成的 setter 和 getter 方法是一个原子操作) 而 nonatomic 就没有这个保证了。所以， nonatomic 的速度要比 atomic 快 ; 不过 atomic 可并不能保证线程安全。如果线程 A 调了 getter ，与此同时线程 B 、线程 C 都调了 setter—— 那最后线程 A get 到的值， 3 种都有可能：可能是 B 、 C set 之前原始的值，也可能是 B set 的值，也可能是 C set 的值。同时，最终这个属性的值，可能是 B set 的值，也有可能是 C set 的值 .

> atomic 是线程安全的 , 需要消耗大量的资源，至少在当前的存取器上是安全的。它是一个默认的特性，但是很少使用，因为比较影响效率 , 加了 atomic ， setter 函数会变成下面这样：
> if (property != newValue) {
> [property release];
> property = [newValue retain];
> }
> 假设有一个 atomic 的属性 "name" ，如果线程 A 调 [self setName:@"A"] ，线程 B 调 [self setName:@"B"] ，线程 C 调 [self name] ，那么所有这些不同线程上的操作都将依次顺序执行 —— 也就是说，如果一个线程正在执行 getter/setter ，其他线程就得等待。因此，属性 name 是读 / 写安全的。
> 但是，如果有另一个线程 D 同时在调 [name release] ，那可能就会 crash ，因为 release 不受 getter/setter 操作的限制。也就是说，这个属性只能说是读 / 写安全的，但并不是线程安全的，因为别的线程还能进行读写之外的其他操作。线程安全需要开发者自己来保证。
> 如果 name 属性是 nonatomic 的，那么上面例子里的所有线程 A 、 B 、 C 、 D 都可以同时执行，可能导致无法预料的结果。如果是 atomic 的，那么 A 、 B 、 C 会串行，而 D 还是并行的。
> //nonatomic如下解释：
> //@property(nonatomic, retain) UITextField *userName;*
> *//系统生成的代码如下：*

> - *(UITextField*) userName {
> return userName;
> }
> - (void) setUserName:(UITextField *)userName {*
> *[userName retain];*
> *[userName release];*
> *userName = userName;*
> *}*
> *//atomic如下解释：*
> *//@property(retain) UITextField* userName;
> *//*系统生成的代码如下：
> - (UITextField *) userName {*
> *UITextField* retval = nil;
> *@synchronized(self) {*
> *retval = [[userName retain] autorelease];*
> *}*
> *return retval;*
> *}*
> *- (void) setUserName:(UITextField*)userName {
> @synchronized(self) {
> [userName release];
> userName = [userName_ retain];
> }
> }*
> // atomic 会加一个锁来保障线程安全，并且引用计数会 +1，来向调用者保证这个对象会一直存在。假如不这样做，如有另一个线程调 setter，可能会出现线程问题，导致引用计数降到 0，原来那个对象就会被释放掉；
> @synthesize 的语义：如果没有手动实现 setter 和 getter 方法，编译器会自动添加这两个方法。
> 【强调合成】
> @dynamic 的语义：告知编译器 , 属性的 setter 与 getter 方法由自己实现，不需要自动生成。
> 【对于 readonly 的属性只需要提供 geter 】；
> 当没有用 @dynamic 修饰属性的时候，编译器默认是实现了 getter 和 setter 方法。还顺便插入了实例变量

atomic 修饰的属性是绝对安全的吗？为什么?
不是，所谓的安全只是局限于 Setter、Getter 的访问器方法而言的，你对它做 Release 的操作是不会受影响的。这个时候就容易崩溃了。

2: 被 weak 修饰的对象在被释放的时候会发生什么？是如何实现的?

答：`weak` 修饰的对象是弱引用，意味着它不会对所引用的对象的生命周期造成影响。如果所引用的对象没有被其他任何强引用所引用，那么在垃圾回收的时候它就会被回收。

这是通过在内存管理的基础数据结构中，如垃圾回收系统的内存表（side table）中维护一个引用计数器实现的。当一个对象的引用计数器为 0 时，它就会被回收。对于强引用，引用计数器加 1，而对于弱引用，引用计数器不会改变。

weak
weak表示指向但不拥有该对象。其修饰的对象引用计数不会增加。无需手动设置，该对象会自行在内存中销毁。 __weak(assign) 修饰表明一种关系“非拥有关系”。
弱引用，不决定对象的存亡。即使一个对象被持有无数个弱引用，只要没有强引用指向它，那么还是会被销毁；在一个对象被释放后，weak会自动将指针指向nil，而assign则不会，向nil发送消息时不会导致崩溃的，所以assign就会导致野指针的错误unrecognized selector sent to instance。
若附有weak 修饰符的变量所引用的对象被废弃，则将nil赋值给该变量。

> 假设变量obj附加strong修饰符且对象被赋值。
> {
> // 声明一个weak指针
> id weak obj1 = obj;
> }
> 模拟编译器编译后的代码：
> id obj1;
> objc_initWeak(&obj1, obj);
> objc_release(obj);
> objc_destroyWeak(&obj1);
> 通过objc_initWeak 函数初始化附有weak修饰符的变量：
> *| 编译器的模拟代码 /*
> *id obj1;*
> *obj1 = 0;*
> *objc_storeWeak(&obj1, obj);*
> *objc_storeWeak函数把第二参数的赋值对象的地址作为键值，将第一参数的附有weak修饰符的变量的地址注册到weak表中。*
> *如果第二参数为0，则把变量的地址从weak表中删除。*
> *weak 表与引用计数表相同，作为散列表被实现。*
> *如果使用weak表，将废弃对象的地址作为键值进行检索，就能高速地获取对应的附有weak修饰符的变量的地址。*
> *另外，由于一个对象可同时赋值给多个附有weak修饰符的变量中，所以对于一个键值，可注册多个变量的地址。 在变量作用域结束时通过 objc_destroyWeak函数释放该变量：*
> */ 编译器的模拟代码 /*
> *objc_storeWeak(&obj1, 0);*

释放对象时，废弃谁都不持有的对象的同时，程序的动作是怎样的呢？下面我们来跟踪观察。对象将通过objc_release函数释放。

（1）objc_release

（2）因为引用计数为0所以执行dealloc

（3）objc_rootDealloc

（4）object_dispose

（5）objc_destructInstance

（6）objc_clear_deallocating 对象被废弃时最后调用的objc_clear_deallocating

函数的动作如下：

（1）从weak表中获取废弃对象的地址为键值的记录。

（2）将包含在记录中的所有附有weak修饰符变量的地址，赋值为nil。

（3）从weak表中删除该记录。

（4）从引用计数表中删除废弃对象的地址为键值的记录。

根据以上步骤，前面说的如果附有weak修饰符的变量所引用的对象被废弃，则将nil赋值给该变量这一功能即被实现。

由此可知，如果大量使用附有weak修饰符的变量，则会消耗相应的CPU资源。

良策是只在需要避免循环引用时使用weak修饰符。

以上就是一个 weak 指针从初始化到被置为nil 的全过程，在写这篇文章之前我一直有疑惑，如果是objc_clear_deallocating函数进行了weak指针置为nil的操作，那objc_destroyWeak函数是干嘛的?

我反复推敲，想起来文中早已说明了用途“在变量作用域结束时通过 objc_destroyWeak函数释放该变量”，也就是说objc_destroyWeak函数是在weak指针被置为nil后，用来将weak释放掉。

weak立即释放对象 使用weak修饰符时，以下源代码会引起编译器警告。

```
{
id weak obj = [[NSObject alloc] init];
}
```

因为该源代码将自己生成并持有的对象赋值给附有weak修饰符的变量中，所以自己不能持有该对象，这时会被释放并被废弃，因此会引起编译器警告：warning: Assigning retained object to weak variable; object will be released after assignment 编译器如何处理该源代码呢?

```
/编译器的模拟代码/
id obj；
id tmp = objc_msgSend(NSObject, @selector(alloc));
objc_msgSend(tmp, @selector(init));
objc_initweak(&obj, tmp);
objc_destroyWeak(&object)；
```

虽然自己生成并持有的对象通过objc_initWeak函数被赋值给附有weak修饰符的变量中，但编译器判断其没有持有者，故该对象立即通过objc_release函数被释放和废弃。

这样一来，nil就会被赋值给引用废弃对象的附有weak修饰符的变量中。

> *下面我们通过NSLog函数来验证一下:*
> *id weak obj= [[NSObject alloc] init];*
> *NSLog(@"obj=%@"，obj);*
> *以下为该源代码的输出结果，其中用%@输出nil。*
> *obj =（null）*
> *如上所述，以下源代码会引起编译器警告。*
> *id weak obj= [[NSObject alloc] init];*
> *这是由于编译器判断生成并持有的对象不能继续持有。*
> *附有unsafe_unretained修饰符的变量又如何呢?*
> *id unsafe_unretained obj=[[NSObject alloc] init]; 与weak修饰符完全相同,*
> *编译器判断生成并持有的对象不能继续持有，从而发出警告： Assigning retained object to unsafe_unretained variable;*
> *object will be released after assignment 该源代码通过编译器转换为以下形式。*
> */编译器的模拟代码/*
> *id obj = objc_msgSend(NSObject, @selector(alloc))；*
> *objc_msgSend(obj,@selector(init))；*
> *objc_release(obj);*
> *objc_release函数立即释放了生成并持有的对象，这样该对象的悬垂指针被赋值给变量obj中。*
> *那么如果最初不赋值变量又会如何呢?*
> *下面的源代码在MRC时必定会发生内存泄漏。*
> *[[NSObject alloc] init]；*
> *由于源代码不使用返回值的对象，所以编译器发出警告。*
> *warning：expression result unused [-Wunused-value] [[NSObject alloc] init]；*
> *可像下面这样通过向void型转换来避免发生警告。*
> *（void)[[NSObject alloc] init]；*
> *不管是否转换为void，该源代码都会转换为以下形式*
> */ 编译器的模拟代码 |*
> id tmp = objc_msgSend(NSObject, @selector(alloc));
> objc_msgSend(tmp, @selector(init))；
> objc_release(tmp)；
> 在调用了生成并持有对象的实例方法后，该对象被释放。
> 看来“由编译器进行内存管理”这句话应该是正确的。

3: block 用什么修饰？strong 可以?

答：block的本质是oc对象，可以通过获取class得知最终继承自NSObject。block分为全局block保存在数据区，栈block保存在栈区，堆block保存在堆区。
block如果捕获了自动变量就是栈block，但是在arc环境下系统会对栈block进行copy操作，拷贝到堆区。在mrc下就是栈block，需要手动调用 copy操作拷贝到栈区。block要想修改变量就需要使用block。用block修饰的变量会被包装成对象。编译成c++文件后可以查看。在编译后的结构

体中会持有被修饰的变量。如果是强类型结构体中也是强类型，如果是弱类型结构体中也是弱类型。而且结构体会有两个函数指针copy和dispose。block编译后如果捕获后的变量需要block来管理内存，那么 block的desc成员中也会有copy和dispose指针。**在修改被block修饰的方法中会通过forwarding指针去取，forwarding指针指向自己栈上的block如果被拷贝到堆上，栈上的forwarding会指向堆上的结构体。这样保障能够访问到正确的值**。然后内存管理方面主要是copy函数和 dispose函数。循环引用的解决手段就是打破强引用改为弱引用。(备注：这个很多地方有讲，block的本质和底层可以看objective-c高级编程这本书很详细)

4: block 为什么能够捕获外界变量？ block 做了什么事?

答："block内部也有个isa指针。block是封装了函数调用以及函数调用环境的OC对象。block内部有isa指针，以及包含指向方法实现的地址，对于局部自动变量，是值传递，对于局部静态变量，是指针传递，全局变量直接访问，**block可以用于解决block内部无法修改auto变量值的问题，编译器会将**block变量包装成一个对象。
"如何防止block里的对象被提前释放了
在block内部使用外部对象时，可以使用block修饰符声明变量，这样可以将变量的生命周期与block的生命周期绑定。这样在block内部使用的变量就不会被提前释放。另外，可以将外部对象的引用添加到一个strong变量中，然后在block中使用strong变量，这样可以保证引用的对象不会被提前释放。（vc你销毁不了 vc只能在主线程同步销毁 arc下托管给系统处理的 mrc下你异步销毁唯一结果就是崩溃；vc走不走dealloc 取决于他所有持有的对象有没有走dealloc，找自己持有的所有非系统对象是不是没走deadlock 系统的不会有问题 就看你自定义对象就行
iOS 比如你进某个页面会从服务器拉个结果，这个结果是全局都在用的，也就是说你进去一次就会刷新一次这个结果，这个结果在其他页面也会用到，假设你进去这个vc触发了拉取的动作 但你很快又退出了，那这个请求的动作怎么办？他回来的结果给谁处理？(关于block引用问题)
这个问题可以通过在请求结束时，判断当前页面是否仍然在显示，如果不是，可以不进行处理。或者，在请求时，将需要处理请求的对象，比如说其他的VC，设置为block的弱引用，在请求结束后，只有在该VC仍然存在时才进行处理。）

"block有什么作用

_在block内修改某局部变量需加block, MRC 环境下block在使用过程中不会对原来值进行copy，可以直接修改该变量 ，ARC环境下会对原值进行copy，内存地址也发生变化;

block可以直接修改 全局和静态变量 ，不会copy该变量的值.
不加_block, MRC 和 ARC block中都是对（原来指针的copy），也就是有两个不同的指针，指向同一个对象。
_使用_block ,MRC环境block中不会对原来的指针进行copy，所以可以更改属性，也可以更改对象本身 ； 而ARC环境则是对原对象的copy，内存地址也发生变化
block所起到的作用就是只要观察到该变量被 block 所持有,就将"外部变量"在栈中的内存地址放到了堆中。进而在block内部也可以修改外部变量的值,
一、block 的三种类型
block 三种类型:全局 block，堆 block、栈 block。

全局 block(NSGlobalBlock)：没有访问外界局部变量的 block 就是全局 block，存储在全局区。
堆 block(NSMallocBlock)：对栈 block 进行 copy 操作返回的就是堆 block，存储在堆区。
栈 block(NSStackBlock)：访问了外界普通局部变量的 block 就是栈 block，存储在栈区。
二、block 建议用 copy 而不用 retain/strong 的原因
block 本质上是一个OC对象，内部有个 isa 指针，可以用 retain/strong/copy 等修饰词修饰。但是 block 在创建的时候内存默认分配在栈上，而不是堆上的。所以它的作用域仅限创建时候的作用域内，当你在该作用域外调用该 block 时，程序就会崩溃

1. 将外部变量作为 __block 声明的变量在 block 中使用。这告诉编译器外部变量的值应该被复制到 block 中。
2. 将外部变量作为弱引用使用，使用 weakSelf 等形式声明 weak 变量，以便 block 不保留对该对象的强引用。这确保了在 block 执行结束后，外部对象可以被正确回收。

5: 谈谈你对事件的传递链和响应链的理解

答：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.大致的过程 initial view –> super view –> .....–> view controller –> window –> Application这里呢 UIResponder是所有可以响应事件的类的基类，其中包括最常见的UIView和UIViewController甚至是UIApplication，通过Hit-Test过程找到这个initial view，当用户点击了手机屏幕时，UIApplication就会调用UIWindow的hitTest: withEvent:方法。这个方法最终返回一个UiView。
也就是我们要找到的那个最前的view响应链中是有 controller 的，因为 controller 继承自 UIResponder。UIApplication –> UIWindow –>递归找到最合适处理的控件 –> 控件调用 touches 方法 –> 判断是否实现 touches 方法 –> 没有实现默认会将事件传递给上一个响应者 –> 找到上一个响应者 –> 找不到方法作废

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。
手势识别 当上面的 _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。

6: 谈谈 KVC 以及 KVO 的理解

答：就是为对象添加一个观察者“Observer”，当其属性值发生改变时，就会调用"observeValueForKeyPath:"方法，为我们提供一个“对象值改变了！”的时机进行一些操作。
Key-Value Obersver，即键值观察。它是观察者模式的一种衍生。基本思想是，对目标对象的某属性添加观察，当该属性发生变化时，会自动的通知观察者。这里所谓的通知是触发观察者对象实现的KVO的接口方法。
KVO是解决model和view同步的好法子。另外，KVO的优点是当被观察的属性值改变时是会自动发送通知的，这比通知中心需要post通知来说，简单了许多。
KVO:当指定的对象的属性被修改了，允许对象接收到通知的机制。
利用RuntimeAPI动态生成一个子类，并且让instance对象的isa指向这个全新的子类，当修改instance对象的属性时，会调用Foundation的，_NSSetXXXValueAndNotify函数，此函数的内部实现为 调用willChangeValueForKey，调用父类(原来)的setter实现，调用
didChangeValueForKey:，
didChangeValueForKey:内部会调用observer的
observeValueForKeyPath:ofObject:change:context:方法。
Apple使用了isa混写（isa-swizzling）来实现KVO。
当观察对象A时，KVO机制动态创建一个新的名为NSKVONotifying_A的新类，该类集成自对象A的本类，且KVO为NSKVONotifying_A重写观察属性的setter方法，setter方法会负责在调用元setter方法之前和之后，通知所有观察对象属性值的更改情况。（备注：isa混写（isa-swizzling）isa:is a kind of ; swizzling: 混合，搅合）
1、NSKVONotifying_A类剖析：在这个过程，被观察对象的isa指针从指向原来的A类，被KVO机制修改为指向系统创建的自NSKVONotifying_A类，来实现当前类属性值改变的监听；所以当我们从应用层面来看，完全没有意识到有新的类出现，这是系统“隐瞒”了对KVO的底层想实现过程，让我们误以为还是原来的类。但是此时如果我们创建一个新的名为“NSKVONotifying_A”的类，就会发现系统运行到注册KVO的那段代码时程序就崩溃，因为系统在注册监听的时候动态创建了名为NSKVONotifying_A的中间类，并指向这个中间类了（isa指针的作用：每个对象都有isa

指针，指向该对象的类，他告诉Runtime系统这个对象的类是什么。所以对象注册为观察者时，isa指针指向新子类，那么这个被观察的对象就神奇地变成新子类的对象（或实例）了。）因而在该对象上对setter的调用就会调用已重写的setter，从而激活键值通知机制。
2、子类setter方法剖析：KVO的键值观察通知依赖与NSObject的两个方法：willChangeValueForKey:和didChangeValueForKey:，
在存取数值的前后分别调用2个方法：被观察属性发生改变之前，willChangeValueForkey:被调用，通知系统该keyPath的属性值即将变更；
当改变发生后，didChangeValueForkey:被调用，通知系统该keyPath的属性值已经变更；
之后，observeValueForKey:ofObject:context:也会被调用。且重写观察属性的setter方法这种继承方式的注入是在运行时而不是编译时实现的。
KVO方法列表中包含：setValueclass-deallocisKVOA-self.p isa addobserve 指向NSKVONotifying_A 动态生成这个子类self.p isa removeobserve 指向A 类临时帮我们实现KVO默默付出 依靠setValue方法 成员变量并没有set方法 watchpoint set variable self->p>name 系统set方法内部调用willChangeValueForkey、didChangeValueForkey实现KVO监听成员变量不会执行set方法didChangeValueForkey 调用nitifyforkey了
KVC的实现原理
答：俗称“键值编码”，通过一个key来访问某个属性；一种间接访问对象属性的机制，甚至可以通过KVC来访问对象的私有属性！修改textField的placeholder也是通过KVC修改的，KVC对多种数据类型的支持，KVC在某种程度上提供了替代存取方法（访问器方法）的方案，不过存取方法终究是个好东西，以至于只要有可能，KVC也尽可能先尝试使用存取方法访问属性。
当使用KVC访问属性时，它内部其实做了很多事：
1.首先查找有无，set，is等property属性对应的存取方法，若有，则直接使用这些方法;
2.若无，则继续查找，get，set等方法，若有就使用；
3.若查询不到以上任何存取方法，则尝试直接访问实例变量，
4.若连该成员变量也访问不到，则会在下面方法中抛出异常。之所以提供这两个方法，就是让你在因访问不到该属性而程序即将崩掉前，供你重写，在内做些处理，防止程序直接崩掉。
5.利用KVC即键值编码来给对象的私有属性赋值。6.如何手动触发KVO，valueForUndefinedKey:和setValue:forUndefinedKey:方法。

7: 讲一下对KVC和KVO的了解，KVC是否会调用setter方法?

答：KVC (Key-Value Coding) 和 KVO (Key-Value Observing) 是 iOS 开发中常用的两种机制。

- KVC (Key-Value Coding) 是一种用于访问对象属性的机制，它可以通过字符串来访问对象的属性，而不需要使用属性的 setter 和 getter 方法。KVC 可以通过 valueForKey: 和 setValue:forKey: 方法来访问和修改对象的属性。
- KVO (Key-Value Observing) 是一种用于监听对象属性变化的机制。当一个对象的属性发生变化时，KVO 会自动通知监听者，使用者可以在对应的方法里进行相应的处理。

使用 KVC 和 KVO 可以使得代码更加简洁，同时使得代码更加灵活，能够让程序更加高效地访问和监听对象的属性变化。

注意：使用 KVO 时需要在监听对象销毁时取消监听，否则会造成内存泄露。

8: 苹果是如何实现 autoreleasepool的？为什么这么设计?

答：由若干个AutoreleasePoolPage以双向链表的形式组合而成，对应数据结构中的parent指针和child指针,并且autorelease还存着对象的地址和内部成员变量地址；AutoreleasePool是按线程一一对应的；
之所以用双链表设计，是因为每张链表都是表头尾相连接，每创建page，会在会在首部创建一个哨兵对象作为标记，最外层page的顶端会有一个next指针，next指针初始值指向该page中可以存储对象地址的开始位置，当满的时候，就会在链表的顶端指向下一张表；
删除对象地址也就是释放某一个对象时会根据链表去查找链表所在的节点位置，这样删除节点，链表依然是连续的，说一下链表删除核心思想，其主要是找到指定位置节点的前一个节点，找到指定位置节点的下一个节点，将指定位置节点的前一个节点指向指定位置节点的下一个节点，删除指定位置节点。虽然这么做增加删除节点复杂了点但是提高了检索速度，双向查找节点值的时候方便，可进可退。

备注：通知使用单链表数据结构：没有增容问题,插入一个开辟一个空间，用链表不用数组，利于增删改查，增删。而且 查找方面，数组基于角标查找才是 O(1)，数组在内存中是按照顺序存储的，每个元素出现顺序与我们可见的index，也就是索引一一对应。这就意味着，只要知道其中一个元素地址，便能在O(1)时间内随机访问任意一个元素。基于元素查找 跟 链表没区别。数组扩容问题，其实关系不大，只是没必要，如果数组有一方面优于链表，肯定选择数组，因为数组对cpu友好。实际上，你实在找不到在 通知这一方面 数组能比链表好。而这种能力以链表为代表的非顺序存储模型是无法具备的。我们都知道链表某个节点引用了前驱或后继节点，因此它只能一次移动一步，像蜗牛似的爬到想要访问的节点。如果已知节点和待查找的节点中间间隔N个元素，它就得移动N步。综上，数组能一步到位，链表N步到位，结论：数组随机访问能力强于链表。

9: 谈谈你对 FRP (函数响应式) 的理解，延伸一下 RxSwift 或者 RAC ?

答：FRP (Functional Reactive Programming) 是一种编程范式，其中函数和数据流是基本元素。在 FRP 中，函数对于输入流的响应是基于时间的，这种响应是可观察的。

RxSwift 是一个用于 iOS 和 macOS 开发的 FRP 框架。它基于 ReactiveX（Rx）规范，提供了一组用于操作数据流的操作符。

RAC (ReactiveCocoa) 是一个用于 iOS 和 macOS 开发的 FRP 框架。它提供了一组用于操作数据流的操作符。

总的来说，FRP 是一种编程思想，而 RxSwift 和 RAC 则是具体的实现方式

10:property的作用是什么，有哪些关键词，分别是什么含义?

答：关键字有strong，weak，assign，copy，nonatomic，atomic，
readonly,readwrite,getter，setter。
strong：释放旧对象将旧对象的值赋予输入对象，再提高输入对象的索引计数为1，常使用在继承自NSObject的类
weak：weak不增加对对象的引用计数，也不持有对象，因此不能决定对象的释放。它比assign多了一个功能，当对象消失后自动把指针变成nil
assign：简单赋值，不更改索引计数，适用于基础数据类型（NSInteger CGFloat）和C数据类型（int float double char 等）简单数据类型。
copy：建立一个索引计数为1 的对象然后释放旧对象 对NSString它指出，在赋值时使用传入值的一份拷贝，拷贝工作由copy方法执行，此属性只对那些实行了NSCopying协议的对象类型有效。
nonatomic：非原子性访问对于属性赋值的时候不加锁，多线程并发访问会提高性能，如果不加此属性则默认是两个访问方法都为原子型事务访问。
atomic：和 nonatomic用来决定编译器生成的getter和setter是否为原子操作，atomic 设置成员变量的@property属性时 默认为是atomic 提供线程安全。在多线程环境下，原子操作是必要的否则会引起错误的结果。
readonly：此标记说明属性是只读的
readwrite：此标记说明属性会被当成读写的 这也是默认的属性
getter：指定 get 方法，并需要实现这个方法。必须返回与声明类型相同的变量，没有参数
setter：指定 set 方法，并需要实现这个方法。带一个与声明类型相同的参数，没有返回值（返回空值）

11: 父类的property是如何查找的?

答：在 iOS 中，父类的 property 是在子类中继承过来的。当一个子类对象访问其父类的 property 时，会直接在父类中查找该 property。如果父类中没有该 property，则会在父类的父类中查找该 property，以此类推。如果在所有父类中都找不到该 property，则会抛出一个异常。

在 iOS 中，类继承采用单继承模型，即一个类只能有一个父类，而不像python中那样可以有多个父类。

12: NSArray、NSDictionary应该如何选关键词?

答：NSArray 是一种有序集合，它可以存储多个对象，并按照添加顺序维护元素的顺序。在使用 NSArray 时，关键词主要包括：objectAtIndex:、count、addObject:、removeObject:、containsObject:等。
NSDictionary 是一种无序集合，它可以存储多个键值对。在使用 NSDictionary 时，关键词主要包括：objectForKey:、setObject:forKey:、count、allKeys、allValues等。

总的来说，选择使用 NSArray 或 NSDictionary 取决于你的需求，如果需要按照添加顺序维护元素的顺序，可以使用 NSArray，如果需要根据键来查询元素，可以使用 NSDictionary。

13: copy和muteCopy有什么区别，深复制和浅复制是什么意思，如何实现深复制?

答：1.不可变类型(不管是集合还是非集合),copy结果，不产生新对象，浅拷贝；
不可变类型(不管是集合还是非集合),mutableCopy结果，产生新对象，深拷贝.
2.可变类型(不管是集合还是非集合),copy结果，产生新对象，深拷贝；
可变类型(不管是集合还是非集合),mutableCopy结果，产生新对象，深拷贝.
3.对不可变类型（NString、NSArray、NSSet），要用copy修饰；
4.可变类型（NSMutableString、NSMutableArray、NSMutableSet）,要用strong修饰;
5.用copy还是strong修饰一个属性时，与深拷贝浅拷贝不要混为一谈了，是两码事。

15: RunLoop 的作用是什么？它的内部工作机制了解么？

答：https://mp.weixin.qq.com/s/CTFWeNg6sZueKz8UkZEe2Q

16: ios 中少用NSLog

答：推荐使用其他的日志框架，如 CocoaLumberjack，XCGLogger 等

1. 影响性能：在 Release 模式下， NSLog 不会被编译，但是在 Debug 模式下，它会不断写入日志，这会消耗系统资源，影响应用程序的性能。
2. 安全性问题：使用 NSLog 记录的日志信息可能包含敏感信息，如用户的隐私数据，密码，设备信息等。如果这些日志被黑客获取，将对用户隐私和应用程序安全带来严重威胁。
3. 难以管理：随着应用程序的不断扩大， NSLog 的日志量也会随之增加，如果没有一个良好的日志管理机制，很容易使应用程序的日志混乱不堪。

17: 说一下对GCD的了解，它有那些方法，分别是做什么用的?

答：GCD (Grand Central Dispatch) 是 Apple 在 iOS 和 macOS 中提供的一种用于管理多线程和并发任务的技术。它是基于 C 语言的，使用起来简单高效。

常用的 GCD 方法包括：

- dispatch_async: 异步执行一个任务，该任务会在新的线程中执行。
- dispatch_sync: 同步执行一个任务，该任务会在当前线程中执行。
- dispatch_after: 延迟执行一个任务，该任务会在给定的时间后执行。
- dispatch_group_async: 将一个任务添加到一个任务组中，可以在所有任务都完成后进行回调。
- dispatch_barrier_async: 在并发队列中插入一个栅栏，在栅栏之前提交的任务和之后提交的任务可以并行执行，但是栅栏之前的任务必须等待栅栏之后的任务完成。
- dispatch_apply: 并发执行一个循环任务，可以对一组数据进行并发处理。

使用 GCD 可以使得代码更加简洁，同时能够更好地利用多核 CPU 的优势，提高程序的性能。

注意：在使用 GCD 时需要注意线程安全的问题。

18、ARC和MRC的区别，iOS是如何管理引用计数的，什么情况下引用计数加1什么情况引用计数减一?

答：ARC (Automatic Reference Counting) 和 MRC (Manual Reference Counting) 是 iOS 中用于管理内存的两种机制。

- MRC 是手动引用计数。在 MRC 中，开发者需要手动管理对象的引用计数，使用 retain、release、autorelease 等方法来管理对象的生命周期。
- ARC 是自动引用计数。在 ARC 中，编译器会自动管理对象的引用计数。开发者不需要手动管理对象的生命周期，编译器会根据对象使用情况自动释放对象。

在 iOS 中，系统使用引用计数来管理对象的生命周期。当一个对象的引用计数变为 0 时，系统会自动释放该对象。

引用计数加1的情况有以下几种：

- 当对象被创建时，引用计数为1
- 当对象被赋值给一个变量时，引用计数加1
- 当对象被添加到一个集合中时，引用计数加1
- 当对象被 retain 方法所持有时，引用计数加1

引用计数减1的情况有以下几种：

- 当对象的引用计数为0时，系统会自动释放该对象
- 当对象被重新赋值给一个变量时，引用计数减1
- 当对象被从一个集合中删除时，引用计数减1
- 当对象被 release 或 autorelease 方法所释放时，引用计数减1
- 在 ARC 中，编译器会自动管理对象的引用计数。开发者不需要手动调用 retain、release、autorelease 等方法来管理对象的生命周期，编译器会根据对象使用情况自动释放对象。但是开发者可以使用 weak 和 unowned 关键字来处理循环引用问题。

  总结：使用ARC可以减少代码编写量，不需要手动管理对象的生命周期，但是开发者需要注意循环引用的问题。

19、在MRC下执行[object autorelease]会发生什么，autorelease是如何实现的?

答：在 MRC 下，执行 [object autorelease] 会将对象添加到自动释放池中。在自动释放池被销毁之前，该对象会保留在内存中。当自动释放池销毁时，会自动调用该对象的 release 方法，减少该对象的引用计数。

自动释放池是由 NSAutoreleasePool 类实现的。在程序运行时，会自动创建一个自动释放池，当程序运行到主线程的 runloop 的即将结束时，会自动销毁该自动释放池，并释放其中所有被 autorelease 方法所持有的对象。

autorelease 方法实现原理如下：

- 当对象被 autorelease 方法所持有时，会调用 retain 方法，将对象的引用计数加1。
- 将对象添加到当前线程的自动释放池中。
- 当自动释放池被销毁时，会自动调用该对象的 release 方法，减少该对象的引用计数。

20、CoreAnimation是如何绘制图像的，动画过程中的frame能否获取到?

答：CoreAnimation是iOS中图像绘制的核心动画框架，它的绘制过程是通过动画层（CALayer）和动画类（CAAnimation）来实现的。动画过程中，每一帧都会调用绘制函数，在绘制函数中会使用图层树结构来进行绘制。

动画过程中的frame能够获取到，通过对动画层的presentationLayer属性，可以获取当前显示在屏幕上的图层，从而获取到当前帧的frame值。

如果需要在动画过程中实时获取frame值，可以通过Core Animation的代理方法实现，或者使用KVO监听动画过程中的变化。

21、OC 中的三大特性怎么理解?

答：Objective-C (OC) 是一种面向对象的编程语言,用于开发 iOS 和 macOS 应用. 下面是 OC 中三大特性的简要说明:

1. 动态类型: OC 是一种动态类型语言,这意味着在运行时可以检查对象的类型并执行相应的操作.
2. 消息传递: OC 是基于消息传递的语言,这意味着对象之间通信是通过发送消息来实现的. 对象的行为是由接收到的消息来决定的.
3. 多继承: OC 支持多继承, 意味着一个类可以从多个父类继承方法和属性,但是由于 OC 中没有接口的概念，实际上是通过协议来实现的.

22、OC如何实现多继承？（其实借助于消息转发，protocol和类别都可以间接实现多继承。）

答：在 Objective-C 中，没有直接支持多继承的语法，但是可以通过消息转发、协议和类别来间接实现多继承的效果。

1. 消息转发

Objective-C 支持动态消息转发，当一个对象收到一个无法识别的消息时，会调用其转发机制，在转发机制中可以重定向消息到另一个对象上。这样就可以让一个类“继承”另一个类的方法。

1. 协议

Objective-C 支持协议，它是一种特殊的接口，可以让一个类实现多个协议。通过实现多个协议，可以让一个类“继承”多个类的方法。

1. 类别

Objective-C 支持类别，它是一种特殊的接口，可以让一个类“扩展”另一个类的方法。通过多个类别，可以让一个类“继承”多个类的方法。

总之,在 Objective-C 中，没有直接支持多继承的语法，但是可以通过消息转发、协议和类别来间接实现多继承的效果。消息转发可以让一个类重定向消息到另一个类上，协议和类别可以让一个类实现多个接口。这样就可以让一个类同时具有多个类的特性。

23、对设计模式有什么了解，讲一下其中一种是如何使用的?

答：设计模式是软件工程中一种代码重用的方法，它是对软件开发中常见问题的抽象和模板化的解决方案。常用的设计模式有23种,如单例模式，工厂模式，装饰器模式，模板方法模式等.

1. 单例模式（Singleton Pattern）: 保证一个类仅有一个实例，并且提供一个全局访问点。
2. 观察者模式（Observer Pattern）: 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。
3. 适配器模式（Adapter Pattern）: 将一个类的接口转换成客户希望的另外一个接口，使得原本不兼容的两个类可以一起工作。
4. 工厂模式（Factory Pattern）: 定义一个用于创建对象的接口，让子类决定实例化哪一个类。
5. 装饰模式（Decorator Pattern）: 动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。
6. 桥接模式（Bridge Pattern）: 将抽象部分与它的实现部分分离，使它们都可以独立地变化。
7. 策略模式（Strategy Pattern）: 定义一系列算法，把它们一个个封装起来，并且使它们可以相互替换。

其中之一的设计模式是单例模式。

单例模式是一种常用的设计模式，它确保一个类只有一个实例，并提供一个全局访问点。

使用单例模式的好处是可以避免对象的重复创建和资源浪费，并且可以在系统中方便地访问它。

使用方法:

- 私有化构造器
- 提供静态的访问方法

```
@implementation Singleton

+ (Singleton *)sharedInstance
{
    static Singleton *sharedSingleton = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
                                  sharedSingleton = [[self alloc] init];
                });
        return sharedSingleton;
}

- (id)init {
        if (self = [super init]) {
                // 进行初始化
        }
        return self;
}
- (void)someMethod {
        // 方法实现
}
@end
```

上面这段代码是一个简单的单例模式的实现示例，使用dispatch_once来确保在多线程环境下单例的唯一性。在这个例子中，可以通过调用 [Singleton sharedInstance] 来获取单例对象的实例。

单例模式是一种常用的设计模式,可以避免对象的重复创建和资源浪费,并且可以在系统中方便地访问它.

策略模式是一种行为型设计模式，它定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

策略模式的意义：(testSingature_N/ViewController/test13)

1. 将算法封装到独立的类中，使得它们可以相互替换。
2. 避免多重条件转移语句。
3. 提高算法的保密性和安全性。

24、有没有哪个开源库让你用的很舒服，讲一下让你舒服的地方?

答：

25、pod install 与 pod update的区别?

答：每一次运行pod install命令后，都会去下载安装新的库，并且会修改Podfile.lock文件中记录的库的版本。Podfile.lock文件是用来追踪和锁定这些库的版本的。
运行pod install后，它仅仅只能解决Podfile.lock中没有列出来的依赖关系。
在Podfile.lock中列出的那些库，也仅仅只是去下载Podfile.lock中指定的版本，并不会去检查最新的版本。
没有在Podfile.lock中列出的那些库，会去检索Podfile中指定的版本，比如pod 'myPod', '~>1.2'
当你运行了 pod update PODNAME命令，CocoaPods将不会考虑Podfile.lock中列出的版本，而直接去查找该库的新版本。它将更新到这个库尽可能新的版本，只要符合Podfile中的版本限制要求。
如果使用pod update命令不带库名称参数，CocoaPods将会去更新Podfile中每一个库的尽可能新的版本。
使用pod update仅仅只是去更新指定库的版本（或者全部库）。
提交你的Podfile.lock文件:
提醒一下，即使你一向不**commit**你的库文件到你的共享仓库，你也应该总是**commit & push**到你的Podfile.lcok文件中。
否则，就会破坏掉pod install的整个设计逻辑，造成Podfile.lock文件无法锁定你已经安装的库。

26、NSNotification原理，如何处理对象释放问题？

答：通知原理解析，通知底层探究
NSNotificationCenter 内部根据每个通知名称name都引用着一个数组（这个数组应该是弱引用数组NSHashTable），数组里存着每个注册者的target-action，当发通知时（[[NSNotificationCenter defaultCenter] postNotificationName:@"target" object:nil];），会根据NotificationName找到对应的数组，然后便利数组里的target-action，达到批量分发的功能。
对于Observation持有observer：持有的是weak类型指针，当observer释放时observer会置nil，nil对象performSelector不再会崩溃。 name和Observation是映射关系，observer和sel包含在Observation结构体中，NSNotification维护了全局对象表NCTable结构，结构体里包含GSIMapTable表的结构，用于存储Observation。

27、多个线程，操作同一个变量，会有什么后果?

答：

```
@property (nonatomic, strong) NSString bookName;
Data p1 = [[Data alloc]init];
dispatch_queue_t queue = dispatch_queue_create("temp",
DISPATCH_QUEUE_CONCURRENT);
for(int i = 1;i<10000;i++)
{
dispatch_async(queue, ^{
[p1 setBookName:[NSString stringWithFormat:@"dddddd:%d",i]];
```

});
}
崩溃-> 0x10961b928 <+8>: testb $0x4, 0x20(%rax)
bookName的声明中指定属性nonatomic，表示为非线程安全的，set方法没有上锁。多线程同时进入到set方法进行操作时会引发问题，只要改换为atomic即可避免该问题,那为什么使用nonatomic就不行？多线程同时进入set方法执行时为什么会出错？原因也很简单，因为在ARC下编译器自动生成的set方法中，需要先对之前持有的对象进行release释放操作，如果此时没有加锁，就有可能发生多个线程执行同一段代码，导致release语句被执行了多次，从而导致对同一块内存多次发送release消息，导致的对已经回收的内存进行重复回收，即会报错。
如果修改为：就不会报错
[p1 setBookName:[NSString stringWithFormat:@"1234 %d",i]];
苹果当中为了节省内存和提高执行效率，提出的一种Tagged Pointer的概念，在8个字节之内能存放的范围之内，系统就会以Tagged Pointer的方式生成指针，如果 8 字节承载不了就会用普通的方式来生成普通指针。这个TaggedPointer与普通指针不同，普通指针是一个变量，存放在栈上，其内容是一块内存块的基地址，所以说指针指向一个内存块，但TaggedPointer并不是一个真正的对象，实际上他的内容就是真正的值，而不是内存地址，所以他本质上就是一个普通变量，也不持有某个内存块。所以在set方法中，对这样的并没有持有内存块的普通变量进行赋值是不需要提前release的，是完全没有问题的，所以不会报错。

NSOperationQueue最大并发量是多少

答：创建操作时，应使用 NSOperationQueueDefaultMaxConcurrentOperationCount 操作系统决定最大操作数-它会根据可用的硬件和系统资源来决定。该数字当然会因设备而异

28、view生命周期

答：view的创建：loadView 视图控制器(UIViewController)及其子类,无论是手写代码还是storyboard、xib肯定会调用loadView方法。其它的视图不会调用比如UIButton，UILabel等，因为他们不是视图控制器。下面是视图控制器被创建时会被调用的其它方法： Storyboard/XIB会调用的方法:• initWithCoder • awakeFromNib：此时frame还没有完成。 手写代码调用的代码(必须是UIView比如自定义MDDButton : UIButton) • initWithCoder initWithFrame，创建时init会被调用此方法(可以继承UIView，做下测试)，不过frame为0.除非显示调用此方法，frame才会有值，比如：[[MDDButton alloc] initWithFrame:CGRectMake(10, 10, 100, 40)];这样显示的调用frame不为0。 view采用懒加载的方式，只有用到view时才会被创建，即才会被调用 loadView ——>viewDidLoad这一系列函数 iOS6及以后，内存警告时系统会回收ViewController的View的CALayer里的BitMap（CABackingStore类型，它的内容是直接用于渲染到屏幕，它是View消耗内存的大户）。view和calayer占的内存极少， 数量级也就在byte和kbyte之间，所以系统只回收了BitMap，但是这里所谓的回收只是给BitMap占用的内存打了一个volatile标记表明这部分内存是可能随时被其它数据占用,平时没内存警告时正在使用的内存标记为In use，完全被释放回收的标记为Not in use。概括起来也就是说：iOS6及以后的内存警告时，系统会给用于渲染视图的数据(BitMap)内存打一个volatile， ViewController的View的架子结构并不会回收，当View再次被

访问时，虽然View的架子结构会用重建，但触发drawRect来渲染界面时，如果view对应的BitMap数据内存没有被占用则会被View的drawRect方法直接渲染出来且内存被标记为in use，从而这块内存又可以独享了；如果已被其它数据占用，那么BitMap必须要重建。所以可以看到整个重建过程不再是由loadView来做的，它是通过对view的访问来触发的。但是，请注意， 如果说在iOS6及以后ViewController的loadView方法只会被调用一次，这种说法是不完全准确的。因为：如果在didReceiveMemoryWarning里把ViewController的View也回收了([self.view removeFromSuperview];self.view = nil;)，那么当再次有对View访问时，loadView会被调用以进行完全最彻底的重建(想想也是，ViewController的View都没了，不调loadView来重建那怎么办呢)。 viewDidLoad 加载到内存完成后会调用此函数，在视图切换中，只要控制器不从内存中移除此方法就不会被调用。一般在此方法中添加一些子控件，设置视图的初始属性等等，类似初始化。 viewWillAppear 即将加载到窗口时调用此方法。一般在此方法做一些较为耗时的。这样就可以先显示基本的视图，呈现给用户(让用户感觉不是那么卡)，然后再显示比较耗时的。以免显示一个白屏给用户。 viewDidAppear 视图已经加载到窗口时调用。
- viewWillDisappear－视图即将消失、被覆盖或是隐藏时调用；
- viewDidDisappear－视图已经消失、被覆盖或是隐藏时调用；
- viewVillUnload－当内存过低时，需要释放一些不需要使用的视图时，即将释放时调用；
- viewDidUnload－当内存过低，释放一些不需要的视图时调用。

布局 我们能看到手机上的视图都是UIView还有它的子UIView，当然不能杂乱无章的显示。要进行布局，父UIView需要布局、排列这些子UIView。UIView提供了layoutSubviews方法来处理。需要注意的是layoutSubviews方法由系统来调用，不能程序员来手动调用。可以调用setNeedsLayout方法进行标记，以保证在UI下个刷屏循环中系统会调用layoutSubviews。或者调用layoutIfNeeded直接请求系统调用layoutSubviews。 layoutSubviews的被调用的时机：
- addSubview会触发layoutSubviews，比如viewA add viewB，第一次添加A和B的layoutSubviews都会被调用，而第二次(viewA已经有了viewB)只调用viewB的
- view的Frame变化会触发layoutSubviews
- 滚动一个UIScrollView会触发layoutSubviews
- 旋转Screen会触发父UIView上的layoutSubviews
- 改变transform属性时，当然frame也会变
- 处于key window的UIView才会调用(程序同一时间只有一个window为keyWindow，可以简单理解为显示在最前面的window为keywindow) 最后总结一句话就是，有必要时才会调用，比如设置Frame值没有变化，是不会被调用的，很明显没有必要

29、typeof 和 __typeof,typeof 的区别?

- **typeof** () 和 __typeof() 是 C语言 的编译器特定扩展，因为标准 C 不包含这样的运算符。 标准 C 要求编译器用双下划线前缀语言扩展（这也是为什么你不应该为自己的函数，变量等做这些)
- typeof() 与前两者完全相同的，只不过去掉了下划线，同时现代的编译器也可以理解。
  所以这三个意思是相同的，但没有一个是标准C，不同的编译器会按需选择符合标准的写法。

30、iOS内存管理方式

retain：持有，对原对象引用计数加1，强引用。ARC中使用strong。
copy：拷贝，复制一个对象并创建strong关联，引用计数为1 ，原来对象计数不变。
assign：赋值，不涉及引用计数的变化，弱引用。ARC中对象不使用assign,但原始类型(BOOL、int、float)仍然可以使用,assign修饰的对象，当对象释放之后，即引用计数为0时，指针变量并不会同时置为nil，全局变量就是变为野指针，不知道指向哪，再向该对象发消息，非常容易崩溃。因此，当属性类型是对象时，不要使用assign，会带来一些风险。
weak：赋值（ARC），比assign多了一个功能，对象释放后把指针置为nil，避免了野指针,weak只能用来修饰对象，不能用来修饰基本数据类型。
strong：持有（ARC），等同于retain。
在你打开ARC时，你是不能使用retain、release、autorelease 操作的，原先需要手动添加的用来处理内存管理的引用计数的代码可以自动地由编译器完成了，需要在对象属性上使用weak 和 strong, 其中strong就相当于retain属性，而weak相当于assign，基础类型只需声明非原子锁即可。
其他非对象类型（int、char、float、double、struct、enum 等）则存放在栈区，由系统进行分配和释放，对象的成员变量进行赋值后，会从栈区复制一份到堆区。
代码区：用于存放程序的代码，即 CPU 执行的机器指令，并且是只读的。
* 全局区 / 静态区：它主要存放静态数据、全局数据和常量。分为未初始化全局区（BSS 段）、初始化全局区：（数据段）。程序结束后由系统释放。
* 数据段：用于存放可执行文件中已经初始化的全局变量，也就是用来存放静态分配的变量和全局变量。
* BSS 段：用于存放程序中未初始化的全局变量。
* 常量区：用于存储已经初始化的常量。程序结束后由系统释放。
* 栈区（Stack）：用于存放程序 临时创建的变量、存放函数的参数值、局部变量等。从上往下，地址是连续的，由编译器自动分配释放。
* 堆区（Heap）：用于存放alloc分配的对象，copy之后的block变量（copy后其实是一个对象）等，从下往上，是链表结构，地址不连续的，由程序员分配和释放。
对象的创建和释放：创建对象种时需要申请内存，使用完对象后要及时释放。在 Objective-C 中使用自动引用计数 (ARC) 可以解决大部分内存管理问题。但在某些特殊情况下，如循环引用等，仍然需要手动管理内存。
图像缓存：在 iOS 开发中，图像是一种常见的内存占用者。为了避免内存暴涨，可以使用图像缓存技术，例如 NSCache 和 SDWebImage。
注意循环引用：循环引用是一常见的内存泄漏原因。为了避免循环引用，需要正确使用 weak 和 unowned 关键字，在必要的时候手动管理内存。
定期检查内存使用情况：通过 Xcode 的 Instrument 工具可以定期检查应用的内存使用情况，快速发现内存泄漏和暴涨的问题，并及时解决。

Tagged Pointer（小对象）
Tagged Pointer 专门用来存储小的对象，例如 NSNumber 和 NSDate
Tagged Pointer 指针的值不再是地址了，而是真正的值。所以，实际上它不再是一个对象了，它

只是一个披着对象皮的普通变量而已。所以，它的内存并不存储在堆中，也不需要 malloc 和 free
在内存读取上有着 3 倍的效率，创建时比以前快 106 倍
objc_msgSend 能识别 Tagged Pointer，比如 NSNumber 的 intValue 方法，直接从指针提取数据
使用 Tagged Pointer 后，指针内存储的数据变成了 Tag + Data，也就是将数据直接存储在了指针中
NONPOINTER_ISA （指针中存放与该对象内存相关的信息） 苹果将 isa 设计成了联合体，在 isa 中存储了与该对象相关的一些内存的信息，原因也如上面所说，并不需要 64 个二进制位全部都用来存储指针。

isa 的结构：
// x86_64 架构
struct {
uintptr_t nonpointer : 1; // 0:普通指针，1:优化过，使用位域存储更多信息
uintptr_t has_assoc : 1; // 对象是否含有或曾经含有关联引用
uintptr_t has_cxx_dtor : 1; // 表示是否有C++析构函数或OC的dealloc
uintptr_t shiftcls : 44; // 存放着 Class、Meta-Class 对象的内存地址信息
uintptr_t magic : 6; // 用于在调试时分辨对象是否未完成初始化
uintptr_t weakly_referenced : 1; // 是否被弱引用指向
uintptr_t deallocating : 1; // 对象是否正在释放
uintptr_t has_sidetable_rc : 1; // 是否需要使用 sidetable 来存储引用计数
uintptr_t extra_rc : 8; // 引用计数能够用 8 个二进制位存储时，直接存储在这里
};

// arm64 架构
struct {
uintptr_t nonpointer : 1; // 0:普通指针，1:优化过，使用位域存储更多信息
uintptr_t has_assoc : 1; // 对象是否含有或曾经含有关联引用
uintptr_t has_cxx_dtor : 1; // 表示是否有C++析构函数或OC的dealloc
uintptr_t shiftcls : 33; // 存放着 Class、Meta-Class 对象的内存地址信息
uintptr_t magic : 6; // 用于在调试时分辨对象是否未完成初始化
uintptr_t weakly_referenced : 1; // 是否被弱引用指向
uintptr_t deallocating : 1; // 对象是否正在释放
uintptr_t has_sidetable_rc : 1; // 是否需要使用 sidetable 来存储引用计数
uintptr_t extra_rc : 19; // 引用计数能够用 19 个二进制位存储时，直接存储在这里
};

这里的 has_sidetable_rc 和 extra_rc，has_sidetable_rc 表明该指针是否引用了 sidetable 散列表，之所以有这个选项，是因为少量的引用计数是不会直接存放在 SideTables 表中的，对象的引用计数会先存放在 extra_rc 中，当其被存满时，才会存入相应的 SideTables 散列表中，

SideTables 中有很多张 SideTable，每个 SideTable 也都是一个散列表，而引用计数表就包含在 SideTable 之中。

散列表（引用计数表、弱引用表）

引用计数要么存放在 isa 的 extra_rc 中，要么存放在引用计数表中，而引用计数表包含在一个叫 SideTable 的结构中，它是一个散列表，也就是哈希表。而 SideTable 又包含在一个全局的 StripeMap 的哈希映射表中，这个表的名字叫 SideTables。

当一个对象访问 SideTables 时：

首先会取得对象的地址，将地址进行哈希运算，与 SideTables 中 SideTable 的个数取余，最后得到的结果就是该对象所要访问的 SideTable

在取得的 SideTable 中的 RefcountMap 表中再进行一次哈希查找，找到该对象在引用计数表中对应的位置

如果该位置存在对应的引用计数，则对其进行操作，如果没有对应的引用计数，则创建一个对应的 size_t 对象，其实就是一个 uint 类型的无符号整型

弱引用表也是一张哈希表的结构，其内部包含了每个对象对应的弱引用表 weak_entry_t，而 weak_entry_t 是一个结构体数组，其中包含的则是每一个对象弱引用的对象所对应的弱引用指针。

线程锁

答：循环锁NSRecursiveLock，条件锁NSConditionLock，分布式锁NSDistributedLock

Runtime相关问题

1: 什么是 isa，isa 的作用是什么?

答：isa 是 Objective-C 中继承和多态的基础。
isa 是 Objective-C 中一个特殊的指针，用来表示一个对象的类信息。它是存储在每个对象中的，用来标识对象的类型。

isa 的作用是在运行时动态确定对象的类型，以便于消息转发（Message Forwarding）、多态（Polymorphism）和内存管理（Memory Management）等功能的实现。

2: 一个实例对象的isa 指向什么？类对象指向什么？元类isa 指向什么?

答：一个实例对象的 `isa` 指针指向该实例对象的类对象。类对象的 `isa` 指针指向该类的元类。元类的 `isa` 指针指向基类的元类，一直到根元类 `NSObject`。

3: objc中类方法和实例方法有什么本质区别和联系?

答：类方法：类方法是作用在类上的方法，可以通过类名直接调用，不需要创建实例对象。类方法通常用于实现工厂方法、创建单例对象等。类方法的声明方式是在方法前面加上 + 号。

实例方法：实例方法是作用在实例对象上的方法，必须通过创建的实例对象调用，访问对象的内部数据等。实例方法的声明方式是在方法前面加上 - 号。

联系：类方法和实例方法都是类的成员，可以直接访问类的其他成员（包括类变量、实例变量、其他方法等），但是对于其他类的访问权限要遵守封装原则。

4: load 和 initialize 的区别?

答：load是main函数之前调用的，initialize是类第一次接收到消息的时候调用的，每一个类只会initialize一次(如果子类没有实现+initialize，会调用父类的+initialize（所以父类的+initialize可能会被调用多次）,如果分类实现了+initialize，就覆盖类本身的+initialize调用)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。

5: _objc_msgForward 函数是做什么的？直接调用会发生什么问题?

答：_objc_msgForward 函数是 Objective-C 的消息转发机制的一部分，在 Objective-C 的运行时中被调用。当程序试图调用一个没有实现的方法时，运行时系统会调用 _objc_msgForward 函数来对该方法进行动态转发，以便能够在运行时进行实现。

直接调用 _objc_msgForward 函数是不安全的，因为该函数是内部 Objective-C 运行时实现的一部分，具体的实现和调用方法可能会有所变化。如果直接调用，则会导致程序崩溃或不可预期的行为。

6: Url输入到浏览器后都做了哪些工作?

答：

1. 解析URL：浏览器会分析URL，并将其分成主机名、协议和资源路径等部分。
2. 构建DNS请求：浏览器会请求DNS服务器，解析URL对应的主机名为IP地址。

3. 建立连接：浏览器根据解析出的IP地址与Web服务器建立连接，连接的方式取决于协议，例如HTTP或HTTPS。
4. 发送请求：浏览器向Web服务器发送请求，请求内容包括资源的路径、请求方式（例如GET或POST）等。
5. 接收响应：Web服务器向浏览器返回响应，响应内容可以是HTML、图像、音频等。
6. 解析内容：浏览器对返回的内容进行解析，并将其呈现在用户界面上。
7. 缓存：浏览器会根据响应头中的Cache-Control字段决定是否缓存内容，以便于下次快速访问。

7: 能否想象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?

答：在 iOS 开发中，编译后生成的类是不能添加实例变量的。这是因为编译后生成的类已经经过了编译器静态检查，已经固定了类的内部结构和实现。

然而，在运行时，可以通过 Objective-C 运行时系统向运行时创建的类中添加实例变量。但是，在实际开发中，这种做法是不推荐的，因为它不是安全的，并且容易导致不可预知的问题。如果需要为类添加实例变量，应该使用继承或关联对象。

8: 谈谈你对切面编程的理解

答：是面向对象编程的一种补充。AOP的目的是将程序中的横切关注点（cross-cutting concerns）从核心业务逻辑代码中分离出来。这些横切关注点包括日志记录，安全性，事务管理等。使用AOP的目的是使代码结构更加清晰，提高代码的可维护性和可读性。

AOP通过使用切面（aspect）来管理这些横切关注点。切面是一个独立的组件，其中定义了要在程序中增加的行为。切面可以通过在方法执行前，方法执行后或方法执行过程中插入代码来实现。这些代码称为通知（advice），它们可以在方法执行时被自动调用。

通过使用AOP，开发人员可以更好地管理应用程序的复杂性，并且可以通过提高代码的重用性来提高开发效率。在很多情况下，使用AOP可以帮助避免在业务代码中包含过多的关注点，并且可以使代码结构更加清晰，使开发人员能够更容易地维护和扩展代码。

9: 消息转发机制实现过程?

答：当调用一个NSObject对象不存在的方法时，并不会马上抛出异常，而是会经过多层转发，层层调用对象的-resolveInstanceMethod: , -forwardingTargetForSelector: , -methodSignatureForSelector: ,-forwardInvocation:等方法，其中最后-forwardInvocation：是会有一个NSInvocation对象，这个NSInvocation对象保存了这个方法调用的所有信息，包括Selector名，参数和返回值类型，最重要的是有所有参数值，可以从这个NSInvocation对象里拿到调用的所有参数值。若想令类能理解某条消息，我们必须实现对应的方法才行，但是在编译期向类发送了其无法理解的消息并不会报错，因为在运行期可以继续向类中添加方法，所以编译器

在编译时还不确定类中到底会不会有某个方法的实现。当对象接收到无法解读的消息后，就会启动“消息转发”机制，程序可由此过程告诉对象应该如何处理未知消息。消息转发分为两大阶段，第一阶段先征询接收者所属的类，看其是否能动态添加方法，以处理当前这个“未知的选择子”，这叫做“动态方法解析”。如果运行期系统已经把第一阶段执行完了，那么接收者自己就无法再以动态新增方法的手段来响应包含该选择的消息了。此时运行期系统会请求接收者以其他手段来处理与消息相关的方法调用。这又细分为两小步。首先请接收者看看有没有其他对象可以处理这条消息，若有，则运行期会把消息转给那个对象，于是消息转发过程结束，一切如常。若没有“备援的接收者”，则启动完整的消息转发机制，运行期会把与消息有关的全部细节都封装到NSInvocation对象中，再给接收者最后一次机会，令其设法解决当前还未处理的这条消息。1.动态方法解析对象在无法解读消息会首先调用所属类的下列类方法：+ (BOOL) resolveInstanceMethod:(SEL)selector参数为那个未知的选择子，返回值表示这个类能否新增一个实例方法处理此选择子。假如尚未实现的方法不是实例方法而是类方法则运行期会调用另一个方法：+ (BOOL) resolveClassMethod:(SEL)selector。使用这种方法的前提是：相关方法的实现代码已经写好，只等着运行的时候动态插入到类里面就可以了。此方案常用来实现@dynamic属性。2.备援接收者当前接收者还有第二次机会能处理未知的选择子，这一步中，运行期会问它：能不能把这条消息转发给其他接收者来处理。与该步骤对应的处理方法：- (id) forwardingTargetForSelector:(SEL)selector方法参数代表未知的选择子，若当前接收者能找到备援对象，则将其返回，若找不到就返回nil。通过此方案我们可以用“组合”来模拟出“多重继承”的某些特性（因为OC属于单继承，一个字类只能继承一个基类）。在一个对象内部，可能还有一系列其他对象，该对象可能由此方法将能够处理某选择子的相关内部对象返回，这样的话，在外界看来，好像该对象亲自处理了这些消息。3.完整的消息转发如果转发算法已经到了这一步的话，那么唯一能做的就是启用完整的消息转发机制了。首先创建NSInvocation对象，把与尚未处理的那条消息有关的全部细节都封装与其中。此对象包含选择子、目标（target）、参数。在出发NSInvocation对象时“消息派发系统”将亲自出马，把消息指派给目标对象。- (void) forwardInvocation:(NSInvocation *)invocation这个方法可以实现很简单：只需改变调用目标，使消息在新目标上得以调用即可，然而这样实现出来的方法与“备援接收者”方案所实现的方法等效，所以很少有人采用这么简单的实现方式。比较有用的实现方式为：在出发消息前，现已某种方式改变消息内容，比如追加另外一个参数，或者改换选择子等等。实现了此方法若发现某调用操作不应由本类处理，则需调用超类的同名方法。这样的话继承体系中的每个类都有机会处理此调用请求，直至NSObject。如果最后调用了NSObject类的方法，那么该方法还会继续调用“doesNotRecognizeSelector:”以抛出异常，此异常表明选择子最终未能得道处理。

10: 调用runtime registerClass方法，创建出来了几个类?

答：实例对象、类对象、runtime，成员变量保存在class_rw_t->class_ro_t
实例对象 本质 objc结构题就是一个对象ISA从objc_object继承过来的
class_ro_t存储了当前类在编译期就已经确定的属性、方法以及遵循的协议，里面是没有分类的方法的。那些运行时添加的方法将会存储在运行时生成的class_rw_t中。

```
// 可读可写
struct class_rw_t {
```

// Be warned that Symbolication knows the layout of this structure.
uint32_t flags;
uint32_t version;
const class_ro_t ro;
// 指向只读的结构体,存放类初始信息
| 这三个都是二位数组，是可读可写的，包含了类的初始内容、分类的内容。
methods中，存储 method_list_t ----> method_t 二维数组，
method_list_t --> method_t 这三个二位数组中的数据有一部分是从class_ro_t中合并过来的。 |
method_array_t methods;
// 方法列表（类对象存放对象方法，元类对象存放类方法）
property_array_t properties;
// 属性列表 protocol_array_t protocols;
//协议列表 Class firstSubclass; Class nextSiblingClass;
}
rw什么时候创建的： 运行时创建的 copy进来 可读写的 属性 方法 协议 脏内存 数据昂贵 一直存在
ro什么时候创建的： 编译的时候创建的 只读状态 成员变量 可移除 保持清洁的数据 永远不可改变 ISA 实例对象 类对象 指向元类 根元类 指向自己 SEL 方法编号 IMP 函数指针 保存了方法的地址 Method 参数类型描述字符串 设计元类的原类 (复用消息通道，类方法也可以放在Class里，但发送消息时，需要增加一个参数)
是实例 类对象方法 发送消息 消息发送 消息转发 runtime 是查找的元类 类方法
https://www.jianshu.com/p/da200b79a6af

11: 用runtime做过什么事情？runtime中的方法交换是如何实现的?

答：用 Runtime 做的事情有很多，主要有以下几种：

1. 动态创建类和对象：通过 Runtime 可以动态创建类和对象，实现在运行时动态添加属性和方法。
2. 消息转发：当调用一个对象的方法时，如果这个对象没有实现这个方法，Runtime 会自动转发这个消息。
3. 方法交换：通过 Runtime 可以交换两个方法的实现，实现在运行时修改类的行为。
4. 关联对象：Runtime 提供了一种机制，可以在运行时为对象添加额外的数据。
5. 动态类型：在运行时，可以动态地改变类型，而不需要重新编译程序。
6. 动态绑定：在运行时，可以动态地绑定对象，而不需要重新编译程序。
7. 动态加载：在运行时，可以动态地加载类，而不需要重新编译程序。
8. 动态调用：在运行时，可以动态地调用方法，而不需要重新编译程序。
9. 动态分配：在运行时，可以动态地分配内存，而不需要重新编译程序。
10. 动态编译：在运行时，可以动态地编译代码，而不需要重新编译程序。

方法交换是通过 Runtime 的函数 method_exchangeImplementations() 实现的。这个函数会将两个方法的实现互相交换。

例如： Method method1 = class_getInstanceMethod([self class], @selector(method1));
Method method2 = class_getInstanceMethod([self class], @selector(method2));
method_exchangeImplementations(method1, method2);

这个例子会交换 self 类的 method1 和 method2 的实现。

但是需要注意，在进行方法交换时，需要确保交换的方法在同一个类中，并且调用的时候不会造成死循环。

12:runtime 如何实现 weak 属性?

答：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。
Runtime会维护一个Weak表,用于维护指向对象的所有weak指针。
Weak表是一个哈希表,其key为所指对象的指针,vaue为Weak指针的地址数组。
具体过程如下:
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(0)函数,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n,最后把这个enty从Weak表中删除,最后清理对象的记录。
当weak引用指向的对象被释放时，又是如何去处理weak指针的呢?

当释放对象时，其基本流程如下：
1、调用objc_release
2、因为对象的引用计数为0，所以执行dealloc
3、在dealloc中，调用了objc_rootDealloc函数
4、在objc_rootDealloc中，调用了object_dispose函数
5、调用objc_destructInstance
6、最后调用objc_clear_deallocating,

详细过程如下：
a. 从weak表中获取废弃对象的地址为键值的记录
b. 将包含在记录中的所有附有 weak修饰符变量的地址，赋值为 nil
c. 将weak表中该记录删除
d. 从引用计数表中删除废弃对象的地址为键值的记录

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。

13:runtime如何通过selector找到对应的IMP地址?

答：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。

14:简述下Objective-C中调用方法的过程

答：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(+ (void)load 方法：这是在类加载到内存中时调用的方法，可以用来执行一些预处理工作。(void)initialize 方法：这是在类的第一次调用之前调用的方法，可以用来初始化类的状态。(void)dealloc 方法：这是在对象被销毁之前

调用的方法，可以用来做一些清理工作。)，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。

首先，程序中会调用某个对象的方法，使用如下的语法：

[object method];

然后，Objective-C 的运行时系统会检查 object 所属的类，看是否有名为 method 的方法。

如果类中存在该方法，运行时系统就会调用该方法，并将执行流程交给方法体。

如果类中不存在该方法，运行时系统就会尝试在父类中寻找该方法。如果父类中也不存在该方法，运行时系统就会继续在父类的父类中查找，直到找到该方法或者查找到了根类为止。

如果在整个继承链中都找不到该方法，程序就会抛出一个异常，终止程序的执行。

如果找到了该方法，方法体就会被执行，直到方法执行完毕为止。

15:Objc为什么要设计消息发送机制，直接调函数有什么缺点?

答：消息发送机制对于直接调函数
主要是多了一个消息动态解析以及消息转发
消息动态解析可以在运行时动态的去添加实现调用，更加灵活
消息转发，则是更加灵活的去更改方法的调用对象，以及方法的执行。
总的来说我感觉消息发送机制是运行时的基础。更加灵活方便吧

网络&多线程问题

1: HTTP的缺陷是什么?

答：HTTP是一种应用层协议，主要用于客户端和服务器之间进行数据交换。它的缺陷包括：

1. 无状态：HTTP是一种无状态协议，不能保持客户端与服务器间的状态信息。
2. 不安全：HTTP是一种明文传输协议，数据易被窃取或篡改。
3. 无控制：HTTP并没有提供流量控制和拥塞控制机制，可能会导致网络阻塞。
4. 效率低下：HTTP需要频繁建立和销毁连接，带来了额外的开销和时间浪费。
5. 缺少认证机制：HTTP并不提供认证机制，因此不能保证客户端和服务器间的安全性。

2: 谈谈三次握手，四次挥手！为什么是三次握手，四次挥手?

答：三次握手和四次挥手是指建立或关闭一个 TCP 连接的步骤。

三次握手：

1. 客户端发送一个带有SYN标志的数据包，表示客户端请求建立连接。

2. 服务器收到请求后，发送一个带有SYN/ACK标志的数据包，表示服务器同意建立连接。
3. 客户端收到服务器的应答后，发送一个带有ACK标志的数据包，表示客户端已经确认建立连接。

四次挥手：

1. 客户端发送一个带有FIN标志的数据包，表示客户端请求关闭连接。
2. 服务器收到请求后，发送一个带有ACK标志的数据包，表示服务器已经确认收到关闭请求。
3. 服务器再发送一个带有FIN标志的数据包，表示服务器也请求关闭连接。
4. 客户端收到服务器的关闭请求后，发送一个带有ACK标志的数据包，表示客户端已经确认关闭连接。

为什么是三次握手、四次挥手？因为每次都需要两个数据包互相确认，才能确定连接已经建立或已经关闭。

3: socket 连接和 Http 连接的区别

答：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，Socket本身并不是协议，而是一个调用接口（API），通过Socket，才能使用TCP/IP协议。
http连接：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：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是连接断开，比如说：服务器端或客户端主机down了，网络故障，或者两者之间长时间没有数据传输，网络防火墙可能会断开该连接以释放网络资源。
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。
HTTP 这个应用层协议是以 TCP 为基础来传输数据的。当你想访问一个资源（资源在网络中就是一个 URL）时，你需要先解析这个资源的 IP 地址和端口号，从而和这个 IP 和端口号所在的服务器建立 TCP 连接，然后 HTTP 客户端发起服务请求（GET）报文，服务器对服务器的请求报文做出响应，等到不需要交换报文时，客户端会关闭连接

4: HTTPS，安全层除了SSL还有，最新的？参数握手时首先客户端要发什么额外参数

答：OS 中的 HTTPS 除了 SSL (Secure Sockets Layer)，还有 TLS (Transport Layer Security)，它是 SSL 的替代品。

在 HTTPS 的安全握手中，首先客户端会发送一个额外的参数，称为客户端支持的 TLS 版本，以及客户端支持的加密算法。服务器可以根据客户端发送的参数，选择一个合适的加密算法，并返回给客户端一个证书，以确认服务器的身份。客户端收到服务器的证书后，就可以确定服务器的身份，并开始加密数据传输。

5: 什么时候POP网络，有了 Alamofire 封装网络 URLSession为什么还要用Moya？

答：POP网络是指符合Protocol Oriented Programming（面向协议编程）思想的网络层设计方案。

Alamofire是一个iOS平台的HTTP网络库，它封装了URLSession，使用方便，但是Moya的出现还是有原因的。

Moya是一个基于Alamofire的网络层封装，它引入了面向协议编程（Protocol Oriented Programming）的思想，在对网络请求的处理和管理上更加灵活，使用起来更加简便，而且更加适合复杂的项目。

6: 如何实现 dispatch_once

答：

```
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
    // 只会执行一次的代码
});
```

7: 能否写一个读写锁？谈谈具体的分析

答：这个场景用于我之前写的IM单聊的时候用到，实现方式是在写入数据的时候其实有很多读取的操作，读写很有可能是同时进行的，我主要控制读写互斥，首先进行读取锁加锁，锁住修改数据源的过程，在加锁的情况下进行写入锁的变更；当数据读取完后，就对写入锁解锁，解锁后写入操作就可以正常操作了，实现读与写的互斥，并行读 串行写的逻辑。

8: 什么时候会出现死锁？如何避免？

答：答：造成死锁的四个必要条件通常被称为“死锁的四个条件”，这四个条件是：互斥条件、请求与保持条件、不剥夺条件、循环等待条件。
互斥条件：指进程对所分配到的资源进行排它性控制，即在一段时间内某资源仅为一个进程所占用。
请求与保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其他进程占有，此时请求进程阻塞，但又对自己已获得的资源保持不放。
不剥夺条件: 指进程已获得的资源在未使用完之前不能强行剥夺，只能由获得该资源的进程自行释放。

循环等待条件: 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占有的资源，P1正在等待P2占有的资源，...，而Pn正在等待已被P0占有的资源。

9: 有哪几种锁？各自的原理？它们之间的区别是什么？

答：

1. @synchronized：是 Objective-C 语言内置的锁，它是一种简单的互斥锁，适用于单线程环境下的同步控制。
2. NSLock：是 Foundation 框架中的锁，它是一种互斥锁，也适用于单线程环境下的同步控制。
3. pthread_mutex：是 POSIX 线程库中的锁，是一种互斥锁，适用于多线程环境下的同步控制。
4. dispatch_semaphore：是 Grand Central Dispatch（GCD）框架中的锁，是一种信号量锁，适用于多线程环境下的同步控制。

它们之间的主要区别：

- @synchronized 和 NSLock 都是互斥锁，pthread_mutex 和 dispatch_semaphore 都是信号量锁，互斥锁和信号量锁在同步控制时的行为方式是不同的。
- @synchronized 和 NSLock 都是在单线程环境下使用，pthread_mutex 和 dispatch_semaphore 都是在多线程环境下使用。
- 在性能上，pthread_mutex 和 dispatch_semaphore 较 @synchronized 和 NSLock 更强大。

10、网络请求的缓存处理，NSCache原理，LRU MRU 缓存原理，SKU SPU组合算法探究

答：iOS中主要有三种缓存处理方式：NSCache、LRU (Least Recently Used) 和 MRU (Most Recently Used)。

NSCache是一种内存缓存，它可以加快应用程序的性能，避免重复请求网络数据。它采用LRU算法淘汰不常用的数据，最常用的数据可以保留在缓存中，加快读取速度。

```
#import <Foundation/Foundation.h>

@interface LRUCache : NSObject

@property (nonatomic, assign) NSInteger capacity;

- (id)get:(id)key;
- (void)put:(id)key value:(id)value;
```

```
@end

@interface LRUCache ()

@property (nonatomic, strong) NSMutableDictionary *cache;
@property (nonatomic, strong) NSMutableDictionary *keys;

@end

@implementation LRUCache
// 使用两个字典来维护缓存内容和数据的使用顺序。一个字典保存缓存数据，另一个字典保存数据使用的顺序，每次查询数据或添加新数据时，程序会更新数据的使用顺序。
- (instancetype)initWithCapacity:(NSInteger)capacity {
    if (self = [super init]) {
        _capacity = capacity;
        _cache = [NSMutableDictionary dictionary];
        _keys = [NSMutableDictionary dictionary];
    }
    return self;
}

- (id)get:(id)key {
    if (_cache[key]) {
        [_keys removeObjectForKey:key];
        [_keys setObject:key forKey:key];
        return _cache[key];
    }
    return nil;
}

- (void)put:(id)key value:(id)value {
    if (_cache[key]) {
        [_cache removeObjectForKey:key];
        [_keys removeObjectForKey:key];
    }
    if (_cache.count >= _capacity) {
        id firstKey = [_keys allKeys][0];
        [_cache removeObjectForKey:firstKey];
        [_keys removeObjectForKey:firstKey];
    }
    [_cache setObject:value forKey:key];
    [_keys setObject:key forKey:key];
}
@end
```

LRU算法是缓存淘汰策略的一种，它按照最近使用的顺序来判断数据的重要性，最近使用的数据会被留在缓存中，最久未使用的数据会被淘汰。

```
//用字典存储缓存数据，然后维护一个链表，链表的首位为最常用数据，末尾为最不常用数据，在读取数据的时候将数据移到链表的最前面，当缓存空间满的时候把链表的末尾元素删除即可。
@interface LRUCache : NSObject

@property (nonatomic, assign) NSInteger capacity;
@property (nonatomic, strong) NSMutableDictionary *cache;
@property (nonatomic, strong) NSMutableArray *keys;

-(id)objectForKey:(id)key;
-(void)setObject:(id)object forKey:(id)key;
@end

@implementation LRUCache

- (instancetype)initWithCapacity:(NSInteger)capacity {
    self = [super init];
    if (self) {
        _capacity = capacity;
        _cache = [NSMutableDictionary dictionary];
        _keys = [NSMutableArray array];
    }
    return self;
}
- (id)objectForKey:(id)key {
    if (!key) {
        return nil;
    }
    id object = [_cache objectForKey:key];
    if (object) {
        [_keys removeObject:key];
        [_keys insertObject:key atIndex:0];
    }
    return object;
}
- (void)setObject:(id)object forKey:(id)key {
    if (!key) {
        return;
    }
    [_cache setObject:object forKey:key];
    [_keys removeObject:key];
    [_keys insertObject:key atIndex:0];
    if (_keys.count > _capacity) {
        id keyToRemove = [_keys lastObject];
```

```
        [_cache removeObjectForKey:keyToRemove];
        [_keys removeLastObject];
    }
}
@end
```

MRU算法是缓存淘汰策略的一种，它按照最近使用的顺序来判断数据的重要性，最近使用的数据会被淘汰，最久未使用的数据会被留在缓存中。

SPU组合算法是一种更加灵活的缓存算法，它结合了LRU和MRU算法的优点，在缓存数据的时候能更好地保证数据的有效性和读取速度。

11、post和get区别,应用层协议里的 GET 和 POST 有啥区别

答：get方法，应用于 等幂操作 的请求；会被缓存且是主动行为；可以收藏书签；可以保留历史记录，有缓存；获取数据；get的querystring（仅支持urlencode编码）；querystring 是url的一部分get、post都可以带上；get请求长度最多1024kb；请求参数会被完整保留在浏览器历史记录里；
post方法，应用于 非等幂操作 的请求；POST 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改；提交表单、数据等，没有历史记录且安全；querystring 是url的一部分get、post都可以带上；post的参数是放在body（支持多种编码）；post对请求数据没有限制；post在浏览器中返回的时候，会有再次提交请求的提示；POST中的参数不会被保留；

12、如何切换到指定线程?

答： performSelector:onThread:withObject:waitUntilDone:

13、为什么说子线程不能更新UI呢?

答：UI操作涉及到渲染访问视图的各种对象属性,异步操作会存在各种读写问题,虽可以通过加锁解决,但是这种另辟蹊径的方式会耗费大量的资源导致运行速度耗损；程序加载完景象就会进入main函数，此时UIApplication初始化主线程进行用户的事件，例如点击 手势操作等传递在主线程进行；所有的用户事件只能在主线程进行响应，渲染则需要30帧左右同步在屏幕上等待Sync垂直信号到来进行更新；非主线程异步操作的情况下不能保证实现同步更新机制。
两个异步线程同时做某些操作 可能发生不被预期的情况：
同时设置一个视图的背景图片很有可能因为当前图片被释放了两次导致应用程序的crash;
同时设置一个视图的背景颜色很有可能渲染树显示的是颜色属性A，而此时的视图的逻辑树储存的是颜色属性B； 同时操作一个视图的属性结构比如线程A循环便利所有的子视图，而此时线程B中将某个子视图直接删除，这样不仅会导致A的错乱，还可能导致应用程序的崩溃；
将大部分视图绘制方法属性改为了线程安全，但仍强烈建议将UI操作放在主线程，原因是是属性图存在读写，不可能全部改写为线程安全。
UI更新一定要在主线程实现的原因：

系统默认是没有加锁的，多个线程被允许同时访问更新同一个UI控件，也就是所谓的Ui访问时候系统当中的UI控件都不是安全的，在多线程中变得不可控；
规定只能在主线程访问UI，相当于是一种UI访问加的锁；
多线程更多的是算力，而非不断的进行UI的更新，保证处理UI的事件都在串行队列中进行。

14、AFNetworking原理

答：AFNetworking是一个 Objective-C 的 HTTP 网络库，它提供了强大的网络请求功能。
AFNetworking 底层是基于 Apple 官方提供的 Foundation Networking 和 System Configuration 框架，并在此基础上进行了高层封装。
AFNetworking 的核心是AFURLConnectionOperation 类，这个类封装了一个 NSURLConnection 对象，提供了简单易用的接口。它支持网络请求的各种方式，如 GET、POST、PUT、DELETE、HEAD等。
AFNetworking 还支持文件上传、文件下载、后台下载、数据的缓存等功能，并对网络状态的监测和管理，有更好的处理。
AFNetworking 框架的优点：

- 简单易用，API 接口清晰明了；
- 支持同步异步网络请求；
- 支持多种请求方式；
- 支持文件下载和文件上传；
- 支持网络状态的监测和管理；
- 对常见错误的提示；
- 支持队列，可以控制请求的顺序。
- AFNetworking 框架的缺点：
- 1 过于依赖回调：AFNetworking使用回调函数处理响应数据，这不利于代码的可读性和可维护性。
- 2 缺少可扩展性：AFNetworking的扩展性不足，无法适应特殊的网络请求需求。

- 3 无法处理错误：AFNetworking没有统一的错误处理方式，需要开发者自己实现错误处理。

- 4 可能存在线程安全问题：AFNetworking使用回调函数和多线程，有可能存在线程安全问题。

- 5 无法跟踪请求状态：AFNetworking不能直接跟踪请求状态，需要开发者手动实现。

- 弥补：
  1。Alamofire：这是 AFNetworking 的 Swift 版本，已经发布了许多版本，且具有更好的代码结构和语法简洁性。
  2。Moya：这是一个更加简洁的网络请求库，在简洁和灵活性方面比 AFNetworking 更胜一筹。

  3。Network：这是一个受欢迎的网络请求库，在代码简洁性和高效性方面比 AFNetworking 有很大的提高

数据结构问题

1: 数据结构的存储一般常用的有几种？各有什么特点？

答：数据结构的存储一般常用的有以下几种：

1. 链式存储：每个节点都存有前驱节点和后继节点的指针，实现了对数据的逻辑连续性。
2. 数组存储：用一个连续的存储区存储数据，每个元素的位置确定，可以快速的随机访问元素。
3. 顺序存储：是数组存储的一种特殊形式，即在静态数组的基础上加上一个有限制的插入和删除操作，常用来实现栈和队列。
4. 矩阵存储：对于二维数据，矩阵存储是一种常用的存储方式，可以索引二维数据的行列元素。
5. 树形存储：将数据组织为树形结构，可以实现数据的逻辑划分，常用来实现高效的查询。

这几种存储方式各有特点，在数据的查询、插入、删除等操作的性能上有很大的差异。在具体应用中，需要根据实际情况选择合适的存储方式。

2: 集合结构 线性结构 树形结构 图形结构

答：数据结构分为四类：集合结构、线性结构、树形结构、图形结构。

1. 集合结构：集合结构是数据结构的基础，它把元素当作整体来看待。常用的集合结构有数组和链表。
2. 线性结构：线性结构以线性的方式组织数据，常见的线性结构有栈、队列、单向链表和双向链表。
3. 树形结构：树形结构是一种分层数据结构，每个节点可以有多个子节点，但每个子节点只有一个父节点，常见的树形结构有二叉树、红黑树和平衡树。
4. 图形结构：图形结构以图的形式组织数据，数据之间可以有多种关系，常见的图形结构有邻接矩阵和邻接表。

3: 单向链表 双向链表 循环链表

答：

4: 数组和链表区别

答：1.存储方式：数组是连续的内存空间，链表是链接的结点 2.插入/删除操作：数组在插入/删除时需要移动大量的元素，而链表只需要修改指针。 3.随机访问：数组支持随机访问，而链表不支持。 4.内存空间：数组需要预先分配空间，而链表动态分配内存。

5: 堆、栈和队列

答：

6: 输入一棵二叉树的根结点，求该树的深度?

答：

7: 输入一棵二叉树的根结点，判断该树是不是平衡二叉树?

答：

算法问题

1: 时间复杂度

答：

2: 空间复杂度

答：

3: 常用的排序算法

答：

4: 字符串反转

答：

5: 链表反转（头差法）

答：

6: 有序数组合并

答：

7: 查找第一个只出现一次的字符（Hash查找）

答：

8: 查找两个子视图的共同父视图

答：

9: 无序数组中的中位数(快排思想)

答：

10: 给定一个整数数组和一个目标值，找出数组中和为目标值的两个数

答：暴力法：遍历每个数，并且对于每个数，检查是否存在另一个数与其相加为目标值。如果存在，则返回两个数的索引。

哈希表法：首先创建一个字典，其中的键是数组中的数，值是数组中的数的索引。然后遍历数组，并且对于每个数，检查是否存在另一个数（目标值减去当前数）存在于字典中。如果存在，则返回这两个数的索引。

两种方法的复杂度均为O(n)，但哈希表法的空间复杂度比暴力法更高，因为它需要一个字典来存储数字和索引之间的映射。

```
- (NSArray *)twoSum:(NSArray *)nums target:(NSInteger)target {
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    for (int i = 0; i < [nums count]; i++) {
        NSInteger complement = target - [nums[i] integerValue];
        if (dict[@(complement)]) {
            return @[@(i), dict[@(complement)]];
        }
        dict[@([nums[i] integerValue])] = @(i);
    }
```

```
    return nil;
}
```

11:5个节点找到中间节点

你可以用快慢指针的方法来找到链表的中间节点。快指针每次向后走两个节点，慢指针每次向后走一个节点。当快指针到达链表末尾时，慢指针恰好在中间节点处。

```python
// 这是一种 O(n) 时间复杂度和 O(1) 空间复杂度的方法。
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def findMiddleNode(head):
    slow = head
    fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

架构设计问题

1: 设计模式是为了解决什么问题的?

答：1). MVC模式：Model View Control，把模型 视图 控制器 层进行解耦合编写。
2). MVVM模式：Model View ViewModel 把模型 视图 业务逻辑 层进行解耦和编写。
3). 单例模式：通过static关键词，声明全局变量。在整个进程运行期间只会被赋值一次。
4). 观察者模式：KVO是典型的观察者模式，观察某个属性的状态，状态发生变化时通知观察者。
5). 委托模式：代理+协议的组合。实现1对1的反向传值操作。
6). 工厂模式：通过一个类方法，批量的根据已有模板生产对象。

2: 看过哪些第三方框架的源码，它们是怎么设计的?

答：SDWebImage是一款非常流行的图片加载和缓存的开源库。它的不足在于：

1. 内存占用过大：SDWebImage 使用内存缓存策略，容易造成内存占用过多的问题，对于对内存较敏感的应用程序尤其如此。
2. 缓存策略非常简单：SDWebImage 没有使用复杂的缓存策略，对于复杂的缓存需求需要自己编写代码实现。
3. 没有提供接口进行定制：SDWebImage 没有提供相应的接口进行定制，对于更复杂的图片加载需求需要自己进行扩展。

为了优化 SDWebImage 的性能，可以：

1. 使用磁盘缓存：可以结合磁盘缓存和内存缓存，减小内存占用的同时进一步优化图片加载的速度。
2. 实现自定义缓存策略：可以通过继承现有缓存策略实现自定义缓存策略。
3. 扩展SDWebImage：可以通过扩展 SDWebImage 来满足更多的图片加载需求。

3: 可以说几个重构的技巧么？你觉得重构适合什么时候来做?

答：

1. 提炼函数：将一段复杂的代码抽取出来，单独放在一个函数里
2. 重命名：给变量，函数，类名等重新命名，使代码更清晰易懂
3. 移除无用代码：删除没有被使用到的代码
4. 提炼类：将某些逻辑放在一个类里，并且将相关的代码移到该类中
5. 合并重复代码：把重复代码合并为一个函数

重构适合在以下情况进行：

1. 软件维护：在软件维护过程中，可以通过重构来使代码更容易维护
2. 代码优化：在代码运行效率不足时，可以通过重构来优化代码
3. 新功能开发：在新功能开发过程中，可以通过重构来帮助代码重构和重组。
4. 增加代码可读性：当代码难以理解和维护时，可以通过重构来增加代码的可读性。

4: 开发中常用架构设计模式你怎么选型?

答：

1. 项目的需求：项目的规模、复杂度和性能要求等因素都应该考虑在内。
2. 可维护性：代码结构要清晰，方便后期维护。
3. 可扩展性：要考虑到项目可能需要扩展或增加新功能。
4. 技术选型：框架、工具和第三方库等技术的选择要认真考虑。
5. 解耦：通过模块分离、接口隔离等方式，使代码变得可重复使用、维护性高。
6. 稳定性：项目架构应该尽可能保证代码稳定性和正确性。
7. 灵活性：架构应该允许灵活的扩展，使项目能适应新的需求变化。
8. 可读性：代码结构、命名等应该具有较高的可读性，方便他人阅读和维护。
9. 可扩展性：架构应该尽可能的支持项目的扩展。

常用的架构设计模式包括：MVC，MVVM，VIPER，Clean Architecture 等。在实际应用中，选择哪种模式取决于实际的需求和因素。

重构适合在以下情况使用：

1. 代码混乱：代码难以维护，结构不清晰，考虑重构。
2. 新需求：如果需要添加新的功能，原代码可能不适用，重构是一种合理的解决方案。
3. 代码不可读：代码很难阅读和理解，重构可以提高代码可读性。
4. 性能问题：如果代码存在性能问题，重构可以提高代码性能。

5: 怎么区分组件化和模块化？怎么区分框架与架构？你是如何组件化解耦的？

答：
1、组件多是半成品（甚至可以是代码片段）强调的是重用。一般作为框架的最底层。独立性强以供模块使用。
2、模块多是指定业务代码进行的封装。可独立运行，以界面或者功能粒度进行划分。一般作为框架的业务层。通过接口等手段通信。目的是降低业务耦合。
3、举例：一个登录的UI界面呈现出来的可以是一个登录模块，而登录模块有按钮、输入框等组件构成（也可以是一个）。
4、它俩一般是一起呈现，一个叫做组件\模块。多个就成了组件\模块化
1、架构是一种抽象，框架是一种具体的实现。架构（动词）是框架（名词）的指导。
2、框架是指以具体的软件实现某种或多种特定功能需求（强调先通用在专用，项目里边都会对第三方框架做二次封装）。架构是思想（强调先大局在局部）。
3、框架是抽象的解决方案（关注大局忽略细节的实现，因为还强调通用性所以多是半成品）
4、两者都是为解决软件开发越来越复杂而采取的策略手段。

1. 请说说你对 iOS 架构的理解。

   答：

2. 请描述 MVC，MVP 和 MVVM 架构模式的区别。
   答：

3. 请说说你最喜欢的 iOS 架构模式，以及原因。
   答：我最喜欢的 iOS 架构模式是MVC（Model-View-Controller）模式，它将应用程序的功能分为三个部分：模型（Model）、视图（View）和控制器（Controller）。这种模式可以有效地将应用程序的功能分离，使得程序的维护和开发更加容易，同时也可以提高程序的可扩展性和可维护性。

4. 请说说如何使用协议和委托进行架构设计。
   答：

5. 请说说怎样使用依赖注入提高代码可测试性。
   答：

6. 请说说怎样使用网络层封装，以提高代码复用性。
   答：网络层封装的目的是把网络请求的逻辑从应用的业务逻辑中分离出来，避免业务代码繁琐的网络请求细节。

   常见的做法是：

   1. 设计一个网络层抽象类，定义网络请求的公共接口，例如请求方式，请求参数等。
   2. 为不同的请求实现不同的网络请求类，继承网络层抽象类，实现自己的请求逻辑。
   3. 通过工厂模式或者其他方法来统一生成请求实例，统一处理请求的响应结果。

7. 请说说如何使用 Core Data 管理数据。
   答：

8. 请说说你使用过的 iOS 数据存储技术，以及它们的优劣。
   答：

9. 常见的 iOS 数据存储技术有：

   - Core Data：支持对象图模型，提供了高级查询、数据迁移等功能，适用于大型数据存储；但是涉及到 Core Data 的实现机制，可能不够直观，对于一些简单的存储需求，使用起来会显得很冗余。
   - SQLite：灵活的 SQL 查询语言，同样适用于大型数据存储；但是需要手动编写比较多的 SQL 语句，并且不支持对象图模型。
   - NSUserDefaults：简单易用，适用于存储少量的配置信息和状态；但是不适用于大型数据存储。
   - NSKeyedArchiver：实现了对象图模型的存储，适用于存储一些对象的状态；但是存储的对象必须遵循 NSCoding 协议，不能存储数据库查询结果。

   在实际项目中，选择哪种存储技术，取决于项目的实际需求，可以根据存储数据的复杂度、数据结构、读写频率等因素来进行选择。

10. 请说说如何使用 iOS 动画技术进行用户体验改进。
    答：

11. 请说说如何使用多线程技术优化 iOS 应用的性能
    答：

性能优化问题

1: tableView 有什么好的性能优化方案？为什么tableView需要数据源来实现协议方法而不是直接把数据通过属性传给tableView?

答：TableView 的性能优化方案如下：

1. 尽量减少 Cell 数量：可以使用复用机制，避免创建过多的 Cell
2. 使用较快的数据源方法：使用基于数组的数据源方法，如 cellForRowAtIndexPath
3. 缓存 Cell 高度：避免在每次滚动时重新计算高度
4. 减少图像大小：使用较小的图像，避免加载过大的图像
5. 异步加载图像：使用异步加载图像，避免加载图像造成 UI 卡顿
6. 使用 Autolayout：使用 Autolayout 管理布局，避免重复计算布局
7. 预加载数据：在 TableView 即将显示前预加载数据，避免滚动时加载数据造成卡顿
8. 使用批量操作：使用批量操作，避免对 TableView 进行多次单独的操作。
    1. 可重用性：TableView的设计使得其中的行可以很方便的复用，可以有效减少内存的使用和提高性能。
    2. 数据管理：TableView可以方便的管理大量的数据，并根据需要显示出必要的数据。
    3. 视觉效果：TableView可以通过自定义cell、分组、索引等功能，实现漂亮的视觉效果。
    4. 用户体验：TableView提供了良好的用户体验，可以让用户方便的浏览数据列表并进行操作。

TableView需要数据源来实现协议方法是因为这样可以更好地控制tableView的数据和样式。数据源通过协议方法与tableView进行通信，告诉tableView如何显示数据，例如提供行数，单元格的高度等。数据源还可以维护自己的数据，并在tableView需要数据时返回。这种设计模式可以提高代码的可读性和可维护性。

2: 界面卡顿和检测你都是怎么处理?

答：

1. 界面卡顿：

- 使用 Time Profiler 工具定位卡顿代码
- 减少 UI 线程执行的任务数量
- 使用多线程编程
- 缓存图像，避免加载图像时造成卡顿
- 优化动画，避免过多的动画造成卡顿

1. 检测线程卡住：

- 使用 Debugger 工具进行实时调试
- 设置断点，监视线程状态
- 在卡住的线程中打印 Call Stack，从而定位到卡住代码
- 使用 NSLog 或 printf 等方法记录日志，方便定位问题

```
//引入Performance Monitoring工具：Performance Monitoring工具可以帮助我们诊断应用程序的卡顿和性能问题，例如，主线程卡顿、帧数、内存使用等。
//通过代码追踪线程卡顿：我们可以通过在代码中加入log语句来确定哪个线程卡住了，再通过debug调试检测代码，从而定位卡顿的代码。
// 如果线程卡住，我们可以通过查看log语句的打印情况来确定是否有线程卡住，从而定位问题。
- (void)checkThreadBlock {
    NSTimeInterval begin = [NSDate timeIntervalSinceReferenceDate];
    NSTimeInterval end = 0;
    while ((end - begin) < 10) {
        NSLog(@"run loop");
        end = [NSDate timeIntervalSinceReferenceDate];
    }
}
```

3: 谈谈你对离屏渲染的理解?

答：GPU把渲染好的的内容存放到离屏渲染缓冲区中，在离屏渲染缓冲区（OffscreenBuffer）中进一步做一些处理后，再提交到帧缓冲区（FrameBuffer）中。
一般都是系统去触发，例如对layer层相关处理：包括圆角、阴影、mask等等。iOS系统扁平化后出现的高斯模糊也是利用离屏渲染方式 亦或 是一种主动行为，是为了提高复用的效率。通常是

设置layer的shouldRasterize属性来实现。开启后，会将layer作为位图保存下来，下次直接与其他内容进行混合。这个保存的位置就是OffscreenBuffer中。这样下次需要再次渲染的时候，就可以直接拿来使用了。
触发离屏渲染的原因：
1、一个UIView上包含了多个图层并且为subviews设置内容(背景颜色、图片)，开启了圆角绘制(layer.cornerRadis)
2、需要进行裁剪的 layer (layer.masksToBounds / view.clipsToBounds)
3、设置了组透明度为 YES，并且透明度不为 1 的 layer (layer.allowsGroupOpacity/ layer.opacity)
4、添加了投影的 layer (layer.shadow*)
5、采用了光栅化的 layer (layer.shouldRasterize)
6、绘制了文字的 layer (UILabel, CATextLayer, Core Text 等)
7、为UIView的layer.content设置内容，同时设置背景颜色，会触发离屏渲染

发生原因解释：
APP在渲染的时候大概是使用画家算法： 绘制的过程首先绘制距离较远的场景，然后用绘制距离较近的场景覆盖较远的部分。
在没有设置圆角和maskToBounds的情况下进行图层绘制，一个图层被绘制进FramBuffer后就被丢弃了，接着绘制下一个图层，但是如果开启圆角和maskToBounds时，就会单独开辟另外一片区域OffScreenBuffer用来保存中间的状态，最终在OffScreenBuffer完成渲染，等待图层需要被显示的时候，然后从OffScreenBuffer给到FramBuffer进行显示。

```
    // 不会产生离屏渲染的方案
    let imageView = UIImageView()
    imageView.frame = .init(x: 100, y: 100, width: 60, height: 60)
    // 设置背景颜色
    imageView.backgroundColor = .red
    // 设置圆角
    imageView.layer.cornerRadius = 30
    // 关闭 masksToBounds
    imageView.layer.masksToBounds = false
    view.addSubview(imageView)
// 亦或
    let imageView = UIImageView()
    imageView.frame = .init(x: 100, y: 200, width: 60, height: 60)
    imageView.backgroundColor = .red
    imageView.layer.cornerRadius = 30
    imageView.layer.masksToBounds = true
    view.addSubview(imageView)
// 亦或
    let imageView = UIImageView()
    imageView.frame = .init(x: 100, y: 400, width: 60, height: 60)
    imageView.layer.cornerRadius = 30
```

```
    imageView.layer.masksToBounds = true
    imageView.image = UIImage(named: "avatar.jpg")
    view.addSubview(imageView)
```

4: 如何降低APP包的大小

答：

1. 图片优化：使用适当大小的图片，压缩图片质量，删除不必要的图片。
2. 文件移除：删除不必要的资源，如字体、音频、视频等。
3. 使用静态库：将常用的第三方库打包成静态库，以避免重复引入。
4. 文件冗余：删除重复的文件，如多余的图标。
5. 编译优化：设置编译选项，如优化代码和缩小符号表。
6. 代码优化：优化代码，减少不必要的代码。

以下是一个图片优化的示例代码：

```
#import <UIKit/UIKit.h>

@interface UIImage (Compression)

- (UIImage *)compressImageWithMaxLength:(NSUInteger)maxLength;

@end

@implementation UIImage (Compression)

- (UIImage *)compressImageWithMaxLength:(NSUInteger)maxLength {
    CGFloat compression = 1.0f;
    NSData *data = UIImageJPEGRepresentation(self, compression);
    while (data.length > maxLength && compression > 0) {
        compression -= 0.1f;
        data = UIImageJPEGRepresentation(self, compression);
    }
    return [UIImage imageWithData:data];
}

@end
  // 调用
  UIImage *image = [UIImage imageNamed:@"example.jpg"];
image = [image compressImageWithMaxLength:100 * 1024];
```

5: 日常如何检查内存泄露?

答：

1. 使用 Xcode 工具：在 Xcode 中使用「Debug Memory Graph」和「Allocations」工具来分析内存使用情况。
2. 使用 Instruments：使用「Leaks」模板来检测内存泄漏。
3. 添加断点：在代码中设置断点并使用「Debug Navigator」监视内存使用情况。
4. 使用内存警告：在 Xcode 设置「Edit Scheme」，选择「Diagnostics」并启用「Enable Zombie Objects」。
5. 使用第三方工具：使用如「LeakEye」等第三方内存泄漏检测工具。
6. 在ARC（Automatic Reference Counting）模式下，以下情况可能导致内存泄漏：
    1. 循环引用：当两个对象互相引用时，ARC不能正确管理其引用计数，导致内存泄漏。
    2. 未被使用的闭包：如果闭包被强制执行并引用了一个对象，但没有被正确释放，那么对象就会被泄漏。
    3. 没有释放的单例模式：如果使用单例模式创建对象，并且在整个生命周期内一直保留，那么对象就会被泄漏。
    4. 未正确使用的指针：如果指针没有正确释放，或者释放的不是正确的对象，那么就会造成内存泄漏。

6: APP启动时间应从哪些方面优化?

答：
1.准备编译 （选择build system, building targets）
2.Build target
1.创建了几个文件夹，写入辅助文件：将项目的文件结构对应表、将要执行的脚本、项目依赖库的文件结构对应表写成文件,写入Entitlements.plist文件 ，方便后面使用；并且创建一个 .app 包，后面编译后的文件都会被放入包中；
2.运行预设脚本： Build Phases / Cocoapods 会预设一些脚本。

```
#import <mach/mach_time.h>
// 该代码计算出 app 启动时间（以纳秒为单位），并使用 NSLog 将其记录到控制台。
int main(int argc, char *argv[]) {
    uint64_t start = mach_absolute_time();
    @autoreleasepool {
        int retVal = UIApplicationMain(argc, argv, nil, @"AppDelegate");
        return retVal;
    }
    uint64_t end = mach_absolute_time();
    uint64_t elapsed = end - start;
    mach_timebase_info_data_t timebase;
    mach_timebase_info(&timebase);
    uint64_t elapsedNano = elapsed * timebase.numer / timebase.denom;
```

```
    NSLog(@"elapsed time: %llu ns", elapsedNano);
}
```

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'YourTargetName' do
  # Comment the next line if you're not using Swift and don't want to use
dynamic frameworks
  use_frameworks!
# 在这段代码中，「post_install」预设脚本会在安装完「CocoaPods」依赖后自动运行，并在
「OTHER_SWIFT_FLAGS」中添加「-Xfrontend -debug-time-function-bodies」标志，这样就
能够在「Xcode」控制台中记录函数体执行时间，从而评估启动时间。

需要注意的是，这段代码只是一个示例，根据项目需求可能需要进行调整。
  # Pods for YourTargetName
  pod 'YourPodName'

  post_install do |installer|
    installer.pods_project.targets.each do |target|
        target.build_configurations.each do |config|
            config.build_settings['OTHER_SWIFT_FLAGS'] = '-Xfrontend -debug-
time-function-bodies'
        end
    end
  end
end
```

3.编译类文件：针对每一个文件进行编译，生成可执行文件 Mach-O，这过程 LLVM 的完整流程，前端、优化器、后端；
4.链接文件：将项目中的多个可执行文件合并成一个文件；
5.拷贝资源文件：将项目中的资源文件拷贝到目标包；
6.编译 Asset 文件：我们的图片如果使用 Assets.xcassets 来管理图片，那么这些图片将会被编译成机器码，除了 icon 和 launchImage；
7.编译XIB文件， storyboard 文件：storyboard 文件也是会被编译的；
8.链接 storyboard 文件：将编译后的 storyboard 文件链接成一个文件；
9.生成DSYM文件
10.运行 Cocoapods 脚本：将在编译项目之前已经编译好的依赖库和相关资源拷贝到包中。(如果有bugly脚本 还会执行)
11.生成 .app 包
12.将 Swift 标准库拷贝到包中（如果有swift）
13.对包进行签名
14.完成打包

数据库索引的索引下推?

答：索引下推是一种数据库优化技术，在数据库执行SQL语句时，会自动评估是否可以使用索引代替全表扫描来完成查询，从而提高查询效率。

以MySQL数据库为例，可以使用以下代码创建索引并证明索引下推的效果：

1. 创建表：

```
sqlCopy code
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255),
    age INT,
    created_at DATETIME
);
```

1. 创建索引：

```
scssCopy code
CREATE INDEX idx_users_age ON users(age);
```

1. 插入数据：

```
scssCopy code
INSERT INTO users(name, age, created_at) VALUES('John', 25, NOW());
INSERT INTO users(name, age, created_at) VALUES('Jane', 30, NOW());
```

1. 执行查询语句：

```
sqlCopy code
EXPLAIN SELECT * FROM users WHERE age > 27;
```

1. 查询语句的执行计划中，如果使用了索引，则可以看到"Using index"的字样，说明索引下推成功。

```
sqlCopy code
+----+-------------+-------+-------+--------------+--------+--------+----
---+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref
| rows | Extra       |
+----+-------------+-------+-------+--------------+--------+--------+----
---+------+-------------+
|  1 | SIMPLE      | users | range | idx_users_age | idx_users_age | 4
```

```
| NULL |   1 | Using index |
+----+-------------+-------+-------+---------------+---------+---------+----
---+------+-------------+
```

LLVM 编译过程

- 预处理
- 词法分析
- 语法分析
- 生成 AST
- 静态分析
- 生成 LLVM IR
- 编译器优化
- Bitcode （可选）
- 生成汇编
- 生成目标文件
- 生成可执行文件

void *_Nullable NSMapGet(NSMapTable* _Nonnull, const void * _Nullable): map table argument is NULL**

runtime时系统创建了多少个全局对象 数据结构是什么 作用是什么?

答：系统创建了多个全局对象。这些对象是用于管理类的元数据、消息转发、方法调用等运行时行为的。

数据结构可能包括类元数据缓存、方法列表、消息分发表等。

作用是为了提高程序的运行效率，使用元数据缓存可以避免重复加载类的元数据，方法列表可以帮助快速找到方法的入口，消息分发表可以快速找到正确的方法来处理消息。

面试题：
1.用什么集合，几个集合区别是什么
2.java类型有什么，我说是数据类型， 八大数据类型 他还是问，string是什么类型， 我说是引用类型，接着问 string 和stringbuffer stringbuilding区别是什么
3.消息队列
4.mq
5.sql优化

面试题A：
1.重写和重载分别是什么，区别是什么
2.hashMap的底层远离

3.创建线程的几个方法是什么，区别是什么?

答：iOS中创建线程的几种方法有：NSThread、pthread、NSOperation、GCD。

- NSThread：通过封装了pthread API的NSThread类实现，比较简单易用。
- pthread：是C语言中的多线程库，底层实现，编程难度高。
- NSOperation：是面向对象的线程抽象，是基于pthread的封装，比较易用。
- GCD：是基于C语言的多线程技术，通过队列和任务的方式实现，编程难度低。

代码示例：

- NSThread：

```
objectivecCopy code
NSThread *thread = [[NSThread alloc] initWithTarget:self
selector:@selector(run) object:nil];
[thread start];

- (void)run {
    NSLog(@"NSThread");
}
```

- pthread：

```
cppCopy code
pthread_t thread;
pthread_create(&thread, NULL, run, NULL);
pthread_join(thread, NULL);

void *run(void *param) {
    NSLog(@"pthread");
    return NULL;
}
```

- NSOperation：

```
objectivecCopy code
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
NSInvocationOperation *operation = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(run) object:nil];
[queue addOperation:operation];

- (void)run {
    NSLog(@"NSOperation");
}
```

- GCD：

```
cCopy code
dispatch_queue_t queue = dispatch_queue_create("com.test.queue", NULL);
dispatch_async(queue, ^{
    NSLog(@"GCD");
});
```

4.linux命令中用什么打开文件， 有几种
5.vi命令中用什么搜索
6.linux中用什么查询
面试题B
1.java多线程，
https://www.runoob.com/java/java-multithreading.html
2.事务管理，
https://blog.csdn.net/zhaohong_bo/article/details/90443545
3.Zookeeper
https://www.cnblogs.com/areyouready/p/10014947.html
4.Redis，
https://www.runoob.com/redis/redis-intro.html
5.java集合类的用法，如:HashMap的用法，
https://baijiahao.baidu.com/s?id=1635677223045847111&wfr=spider&for=pc
6.数据库分区分表，
https://blog.csdn.net/qq_28289405/article/details/80576614
7.索引优化 ，
https://blog.csdn.net/yangxingpa/article/details/81477607
8.单点登录的问题
https://www.cnblogs.com/ywlaker/p/6113927.html
面试题C
springcloud的五大组件是什么
面试题d
1.消息队列
2.redius
3.集合有哪些，hashmap,hashset 区别
4.springboot 注解 ，核心注解是什么，核心注解是哪几个组合
5.Arreylist linkedlist 区别
6.spring,springbuilder,springbuffer 区别
7.字符串的方法(忘了具体怎么问，)就是回答。append 等，
8.用过哪些中间价
9.sql优化，数据库优化做过那
10.分页

11.分库分表
12.ioc,aop是什么
15:30
面试题E：
1.抽象类和接口的区别是什么
2.异常都有几种，都是怎么处理的
3.== 和equals什么区别

面试题A：
1.重写和重载分别是什么，区别是什么
2.hashMap的底层远离
3.创建线程的几个方法是什么，区别是什么
4.linux命令中用什么打开文件， 有几种
5.vi命令中用什么搜索
6.linux中用什么查询
面试题B
1.java多线程，2.事务管理，
3.zookeeper，
4.Redis，
5.java集合类的用法，如:hashmap的用法，
6.数据库分区分表，
7.索引优化 ，
8.单点登录的问题
面试题C
springcloud的五大组件是什么
面试题d
1.消息队列
2.redius
3.集合有哪些，hashmap,hashset 区别
4.springboot 注解 ，核心注解是什么，核心注解是哪几个组合
5.Arreylist linkedlist 区别
6.spring,springbuilder,springbuffer 区别
7.字符串的方法(忘了具体怎么问，)就是回答。append 等，
8.用过哪些中间价
9.sql优化，数据库优化做过那
10.分页
11.分库分表
12.ioc,aop是什么
面试题E：
1.抽象类和接口的区别是什么
2.异常都有几种，都是怎么处理的

3.== 和equals什么区别
面试题F：
1.SpringCloud具有以下特点：
做到了良好的开箱即用，可扩展性机制覆盖
包括服务发现和服务注册
服务地址路由
软负载均衡
断路器
分布式消息传递
2.springboot的注解， 关于参数的都有哪个
3.l
面试题G:
1.重载与重写的区别?
2.string、stringbuffer、springbuilder的区别?
3.说说你知道的集合（一般会问）
4.Arraylist与linkedlist的区别？（问的较多）
5.hashtable与hashmap的区别？（问的较多）
面试题F：
1.控制器view的加载过程
2.应用程序的启动流程
3.事件传递与响应的完整过程
4.XIB文件的构成分为哪三个图标？都具有什么功能?
什么是指针、指针地址
什么是cpu：CPU是中央处理器，是电脑内超大规模的集成电路，主控电脑的运算核心与控制核心，可以处理与解释计算机软件的数据。一个成功的CPU不仅有助于降低内部整体计算的成本，还可以通过向外输出给客户更高性价比的选择。
runtime
1、OC的动态性就是由runtime来支撑和实现的，方法的调用 其实都是转化为 objc_msgSend函数的调用
数据结构：objc_object，objc_class，isa，class_data_bits_t，cache_t，method_t对象，类对象，元类对象消息传递
2、利用关联函数给分类添加属性
3、遍历类的成员变量以及方法
4、交换系统方法
5、利用消息转发机制解决方法找不到的问题
首先判断消息接收者是否为nil 如果为空直接退出
1、去自己类的方法缓存列表中查找该方法如果找到则执行该方法，resolveInstanceMethod
class
2、否则去自己类的方法列表中查找该方法 如果找到先缓存该方法然后再执行，
3、否则去自己父类的方法缓存列表中查找该方法 如果找到先缓存该方法到

自己类中，然后执行该方法
4、否则去自己父类的方法列表中查找该方法如果找到先缓存该方法到自己类中，
然后执行该方法，自己没有能力处理这个方法 将此方法转交给别人类处理 执行消息转发机制。
1、首先调用 forwardingTargetForSelector:方法
在此方法中可以返回一个可以处理消息的对象，
如果对象存在则直接给对象转发消息，
如果对象不存在否则进入第2步 调用方法签名函数
2、调用methodSignatureForSelector:方法（方法签名函数）在此方法中：
如果返回值为空 则直接调用 调用doesNotRecognizeSelector:方法
然后程序报错：unrecognized selector sent to instance 经典错误
如果返回值不为空 则直接进入第3步 调用 forwardInvocation方法
3、forwardInvocation：方法
开发人员可以在此方法中返回一个对象来转发此消息
UIView为CALayer提供内容，以及负责处理触摸等事件，参与响应链
CALayer负责显示内容contents
事件响应链
1、点击屏幕-> UIApplication -> UIWindow -> hittest -> pointInside -> subviews -> 遍历view -> subview hittest->hit != nil 执行事件
runloop
1、全局的Dictionary，key 是 pthread_t， value 是 CFRunLoopRef
2、线程和 RunLoop 之间是一一对应的，其关系是保存在一个全局的 Dictionary 里。线程刚创建时并没有 RunLoop，如果你不主动获取，那它一直都不会有。RunLoop 的创建是发生在第一次获取时，RunLoop 的销毁是发生在线程结束时
3、一个 RunLoop 包含若干个 Mode，每个 Mode 又包含若干个 Source/Timer/Observer
4、Source1：port通信，系统事件捕捉；source0只有一个回调函数 没有port
5、主线程的RunLoop是默认开启的，所以程序在开启后，会一直运行，不会退出。其他线程的RunLoop如果需要开启，就手动开启
6、 即将进入Loop 即将处理 Timer 即将处理 Source 即将进入休眠 刚从休眠中唤醒 即将退出Loop
weak
weak表示指向但不拥有该对象。其修饰的对象引用计数不会增加。无需手动设置，该对象会自行在内存中销毁weak表有两个参数，分别是赋值对象的地址作为键值 weak修饰符的变量的地址注册到weak表中，如果对象地址为0，就把变量的地址从weak表中删除weak 策略在属性所指的对象销毁时，系统会将 weak 修饰的属性对象的指针指 向 nil，在 OC 给 nil 发消息是不会有什么问题的; 如果使用 assign 在属性所指 的对象销毁时，属性对象指针还指向原来的对象，由于对象已经被销毁，这时候就产生了野指针，如果这时候在给此对象发送消息，很容造成程序奔溃
assigin 可以用于修饰非 OC 对象,而 weak 必须用于 OC 对象
进程
进程和线程的关系
进程是操作系统的基本单元，也是一段程序的执行过程，是独立运行的

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，一个进程要想执行任务,必须至少有一条线程.
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
7、**KVC**
KVC全称（key-Value coding）称呼为键值编码，在iOS开发中。允许开发者通过key名直接访问对象的属性，或者给对象的属性赋值。需要调用明确的存取方法，这样就可以在运行时动态访问和修改对象的属性，而不是在编译时确定。
首先查找有无，set，is等property属性对应的存取方法，若有，则直接使用这些方法;
2.若无，则继续查找，get，set等方法，若有就使用；
3.若查询不到以上任何存取方法，则尝试直接访问实例变量，
4.若连该成员变量也访问不到，则会在下面方法中抛出异常。之所以提供这两个方法，就是让你在因访问不到该属性而程序即将崩掉前，供你重写，在内做些处理，防止程序直接崩掉。
5.利用KVC即键值编码来给对象的私有属性赋值。
KVO KVO是观察者模式的另一实现,使用了isa混写(isa-swizzling)来实现KVO,当其属性值发生改变时，
利用RuntimeAPI动态生成一个子类NSKVONotifying_A，并且让instance对象的isa指向这个全新的子类
调用willChangeValueForKey，调用父类(原来)的setter实现，调用didChangeValueForKey:内部会调用observer的observeValueForKeyPath:ofObject:change:context:方法。
AutoreleasePool
objc_autoreleasePoolPush：
把当前next位置置为nil，即哨兵对象,然后next指针指向下一个可入栈位置，
AutoreleasePool的多层嵌套，即每次objc_autoreleasePoolPush，实际上是不断地向栈中插入哨兵对象。
objc_autoreleasePoolPop:
根据传入的哨兵对象找到对应位置。
给上次push操作之后添加的对象依次发送release消息。
回退next指针到正确的位置。
10.分类 **声明私有方法，分解体积大的类文件，把framework的私有方法公开，运行时决议，可以为系统类添加分类 。然后会递归调用所有类的 load 方法，可以添加实例方法，类方法，协议，属性（添加getter和setter方法，并没有实例变量，添加实例变量需要用关联对象）+load这一切都是在main函数之前执行的。**
4.如果工程里有两个分类A和B，两个分类中有一个同名的方法，哪个方法最终生效?
取决于分类的编译顺序，最后编译的那个分类的同名方法最终生效，而之前的都会被覆盖掉(这里并不是真正的覆盖，因为其余方法仍然存在，只是访问不到，因为在动态添加类的方法的时候是倒序遍历方法列表的，而最后编译的分类的方法会放在方法列表前面，访问的时候就会先被访问到，同理如果声明了一个和原类方法同名的方法，也会覆盖掉原类的方法)。
6.分类能添加成员变量吗?

不能。只能通过关联对象(objc_setAssociatedObject)来模拟实现成员变量，但其实质是关联内容，所有对象的关联内容都放在同一个全局容器哈希表中:AssociationsHashMap,由AssociationsManager统一管理。
扩展** 声明私有属性，声明方法（没什么意义），声明私有成员变量 ,编译时决议，只能以声明的形式存在，多数情况下寄生在宿主类的.m中，不能为系统类添加扩展。

5**、block*
*Block不允许修改外部变量的值，这里所说的外部变量的值，指的是栈中指针的内存地址。**block所起到的作用就是只要观察到该变量被 block 所持有，就将"外部变量"在栈中的内存地址放到了堆中，这里拷贝的是指向该结构体对象的指针。这样就可以在block内部就可以修改外部变量的值。***
//对于auto age是捕获到内部，把外部age的值存起来，而对于static height，是把外部变量的指针保存起来,
//对于static height值， block在访问static变量（局部变量）时，block内部会捕获到外部变量的地址值，后面修改外部static变量值的时候，（通过指针变量所指向的内存的值）地址访问到的是最新修改后的值。
局部变量被编译成值形式，而静态变量被编成指针形式，全局变量并未截获
而block修饰的变量也是以指针形式
//思考：为什么会出现这两种情况？原因很简单，因为auto是自动变量，出了作用域后会自动销毁的，如果我们不保留他的指针，就会存在访问野指针的情况，所以去要捕获他并把值保存起来。
//局部变量需要跨函数访问，而且随时可能会被销毁所以需要提前捕获。
//全局变量随时都可以访问，且一直存在，所以不需要捕获。
//auto 值传递
//static 指针传递
//全局变量 直接访问
//成员变量与属性在block中调用时，需要被捕获吗?
//因为调用成员变量与属性时，需要通过self来调用，而self对block来说是局部变量，所以是需要捕获的。clang编译后
block_impl结构体为
***struct** block_impl {*
void isa;//isa指针，所以说Block是对象
int Flags;
int Reserved;
void *FuncPtr;//函数指针
};

备注：

https://blog.csdn.net/MinggeQingchun/article/details/117471901

iOS8.0 至 iOS15.0 版本变化?

答：在iOS8里面，官方提供了新的类UIAlertController来替换UIActionSheet 和 UIAlertView。
定位功能使用改变，定位开启定位和始终开启定位，还有info.plist文件新增定位字段
在iOS5.0时时可以跳转到系统的设置页的，跳转到系统设置里自己App的页面，但是在5.1之后就不可以

如何看待重构?

答：二次求导就变成一次求导函数，依赖注入、tableview、代理在安卓实现过程
如果重复三遍就写成工具

动态链接到虚拟内存和machO文件及物理内存之间的地址映射转换吗?

动态链接是指在程序运行期间通过链接库文件将函数和变量等符号引用与符号定义进行连接的过程。在 macOS 系统中，程序的链接是基于 Mach-O 格式进行的，该格式定义了程序在虚拟内存中的布局，包括代码段、数据段、BSS 段等。虚拟内存是操作系统提供的一种地址空间，每个进程都有自己的虚拟地址空间，虚拟地址与物理地址之间的映射关系由操作系统的内存管理单元（MMU）维护。

当程序中的某个符号被引用时，动态链接器会在已加载的库文件中查找符号定义，并将其地址填充到符号引用处。由于虚拟地址空间与物理地址空间之间的映射关系是动态变化的，因此需要进行地址映射转换。这个过程由操作系统内核中的虚拟内存管理单元（VMM）负责，通过页表实现虚拟地址到物理地址的映射。

简单来说，动态链接到虚拟内存和 Mach-O 文件及物理内存之间的地址映射转换由操作系统的 VMM 和 MMU 实现。

Clang:

C 语言

```
int a,b;
a = 2;
b = 10;
bool d = a > b ? : b;
//a > b ? : b 等价于 a > b ? a > b : b;
NSLog(@"%d",d);//10
int x = 3;
int y = ++x ? ++x : 3;
NSLog(@"%d",y);//10

int xx = 0;
int yy = ++xx ? : 3;
```

NSLog(@"%d",yy);//10

xcode的编译流程做了啥?

答：https://www.jianshu.com/p/14612abdeb26 https://objccn.io/issue-6-1/

推荐一些 Code Review 工具

(https://www.zhihu.com/question/41089988/answer/544543949)
1、Crucible：Atlassian 内部代码审查工具；
2、Gerrit：Google 开源的 git 代码审查工具；
3、GitHub：程序员应该很熟悉了，上面的 "Pull Request" 在代码审查这里很好用；
4、LGTM：可用于 GitHub 和 Bitbucket 的 PR 代码安全漏洞和代码质量审查辅助工具；
5、Phabricator：Facebook 开源的 git/mercurial/svn 代码审查工具；
6、PullRequest：GitHub pull requests 代码审查辅助工具；
7、Pull Reminders：GitHub 上有 PR 需要你审核，该插件自动通过 Slack 提醒你；
8、Reviewable：基于 GitHub pull requests 的代码审查辅助工具；
9、Sider：GitHub 自动代码审查辅助工具；
10、Upsource：JetBrain 内部部署的 git/mercurial/perforce/svn 代码审查工具。

开发过程中遇到的最大的困难, 难点在哪里, 怎么解决的,

遇到一个崩溃， bugly也不显示具体位置，然后 通过符号表， 地址偏移，逆向工程，查到了一个sdk的问题， 然后 我通过代码注入的方式，帮他修复了bug。

- 我参与过的项目中，最能证明我能力的事情之一是我在一个应用中实现了一个性能瓶颈的优化。这个应用是一款社交应用，在某些情况下，它会变得非常缓慢。我使用Instruments工具来分析问题，发现是一个高频率的数据库操作导致的。我使用CoreData的子线程上下文来优化数据库操作，并使用GCD来管理线程，这样大大提高了应用的性能。
- 另外一件事是我在一个项目中引入了MVVM架构，使得团队的开发效率大大提高。这个项目是一款金融应用，在之前的MVC架构下，我们发现代码难以维护和扩展。我提出使用MVVM架构来

flutter 事件循环

答：Flutter 使用事件循环来控制应用程序的流程。

Flutter 的事件循环主要由两部分组成：

- Flutter 引擎
- Dart 虚拟机

Flutter 引擎负责渲染界面、处理输入事件、调度动画等。Dart 虚拟机负责处理 Dart 代码的执行。

事件循环的流程如下：

1. Flutter 引擎首先执行一个循环，在该循环中会处理所有的输入事件、动画等。
2. Flutter 引擎会将处理完的事件传递给 Dart 虚拟机，Dart 虚拟机会根据事件调用对应的 Dart 代码进行处理。
3. Dart 代码执行完成后，Flutter 引擎会根据 Dart 代码的执行结果来更新界面。
4. 更新完界面后，Flutter 引擎会将新的界面渲染到屏幕上。
5. 一次事件循环结束后，等待下一次事件处理。

   这样的事件循环机制保证了 Flutter 程序的流程总是在控制之下，能够及时响应用户的输入并更新界面。

Dart 是单线程还是多线程

答：单线程

Stream feature 的区别

答：Flutter 中 Stream 是一种异步事件的处理机制，它可以用来处理来自不同来源的事件，如网络请求、定时器、用户输入等。

Stream 有两种不同的实现方式：

- Single-subscription stream
- Broadcast stream

Single-subscription stream 也称为单订阅流，在这种情况下，一个流只能被监听一次，并且只有一个订阅者能够接收到事件。

Broadcast stream 也称为广播流，在这种情况下，一个流可以被多次监听，并且多个订阅者都可以接收到事件。

总结：

- Single-subscription stream 只能被订阅一次，只能有一个订阅者接收事件
- Broadcast stream 可以被多次订阅，多个订阅者都可以接收事件

在使用Stream的时候，需要根据需要来选择使用 Single-subscription stream 或是 Broadcast stream.

isolate有了解吗

答：Flutter 中 isolate 是一种用于跨线程执行代码的机制。它可以在单独的线程中执行 Dart 代码，而不会影响主线程的性能。

一个 isolate 拥有自己独立的内存空间，不会与其他 isolate 共享数据。这意味着在 isolate 中运行的代码是安全的，不会被其他 isolate 中的代码干扰。

Flutter 中有两种方式来创建 isolate:

- 使用 dart:isolate 库中的 Isolate.spawn() 方法
- 使用 Flutter 提供的 IsolateNameServer.lookup() 方法

isolate 主要用于处理计算密集型任务，如图像处理、大量数据的计算等，或是长时间的后台任务。使用 isolate 可以有效地提高应用程序的性能，并避免主线程的阻塞。

Stateless Widget和Stateful Widget区别

答：Flutter 中有两种类型的 Widget：StatelessWidget 和 StatefulWidget。

StatelessWidget 是不可变的，它的状态不会改变。当 StatelessWidget 的父 Widget 的状态发生变化时，Flutter 会重新构建该 Widget。

StatefulWidget 是可变的，它的状态可以改变。当 StatefulWidget 的状态发生变化时，Flutter 只会对该 Widget 进行部分重绘。

总结：

- StatelessWidget 是不可变的，状态不会改变，在父widget状态改变时会被重构。
- StatefulWidget 是可变的，状态可以改变，在状态改变时只会进行部分重绘。
- 一般来说，如果 Widget 的状态不需要改变，使用 StatelessWidget 更加简单和高效，如果需要改变状态，则使用 StatefulWidget 更加合适。

  另外, StatelessWidget 不能调用 setState方法，而StatefulWidget可以调用setState进行状态更新

选择正确的 Widget 类型，可以提高应用程序的性能和易用性。

StatefulWidget 的生命周期

答：

- createState: 创建并返回 State 对象
- initState: 初始化 State 对象，在每次 build 方法之前调用
- didChangeDependencies: 当 State 对象依赖变化时调用
- build: 绘制 Widget
- didUpdateWidget: 当 Widget 的配置变化时调用
- dispose: 释放资源
- createState: 当第一次构建时调用，返回一个新的 State 对象
- initState: 当第一次构建完成后调用，可以在这里进行一些初始化工作
- didChangeDependencies: 当 State 对象所依赖的 InheritedWidget 发生变化时调用
- build: 绘制 Widget，返回当前 State 对象对应的 Widget
- didUpdateWidget: 当 Widget 的配置发生变化时调用
- dispose: 释放资源，在移除该 Widget 时调用，在这里可以进行一些清理工作。
- 还有一个方法 setState, 这个方法用于更新状态,当状态改变时调用setState，会重新调用build方法进行更新。
- 总结一下，生命周期方法主要有:

    - createState: 创建并返回 State 对象
    - initState: 初始化 State 对象
    - didChangeDependencies: 当 State 对象依赖变化时调用
    - build: 绘制 Widget
    - didUpdateWidget: 当 Widget 的配置变化时调用
    - dispose: 释放资源
    - setState: 用于更新状态

- 在开发中我们可以在这些生命周期方法中执行特定的操作，例如：

    - 初始化：在 initState 方法中进行初始化操作
    - 数据请求：在 didChangeDependencies 方法中进行数据请求
    - 清理：在 dispose 方法中进行资源释放等清理操作。

    这样做能够更好的组织代码，提高代码的可读性和可维护性。

介绍下widget、state、context

答：Widget是Flutter中的基本单元，用于描述应用程序的UI。在Flutter中，所有的东西都是一个widget。

State是Flutter中的状态管理器。它维护着一些可变的数据，并且当这些数据发生改变时，可以触发UI重构。

Context是Flutter中的上下文管理器。它为widget提供了额外的信息，比如主题、语言等。

Widget和element和RenderObject之间的关系

答：Widget, Element 和 RenderObject是Flutter中用于构建UI的三种不同类型的对象。

Widget是用来描述UI元素的对象，它表示了一个可渲染的部件。它是一个不可变对象，可以被挂载到一个"widget tree"上。

Element 是 Widget的一个实例，它是一个可变对象。 Element 中包含着Widget的配置信息，并且与一个RenderObject一一对应。当需要重绘时，会根据Element中的信息来重新构建对应的RenderObject.

RenderObject是一个可渲染的对象，它是最终会在屏幕上呈现的对象。RenderObject并不直接对应一个Element，而是通过RenderBox来渲染。通常，RenderObject是用来渲染一组相关的内容，而RenderBox则是用来渲染单个元素。

简单来说： Widget是一个描述性的对象，Element是一个具体实例化的对象，RenderObject是最终渲染的对象。

flutter是如何实现多任务并行的?

答：Isolate 是 Dart 中的一种并发单元，它可以独立于其他 Isolate 运行，并且每个 Isolate 都有自己的内存空间。Flutter 在渲染界面、执行 I/O 操作等场景中使用 Isolates 来实现多任务并行。

在 Flutter 中，可以使用 `dart:isolate` 库中的 `Isolate` 类和 `Isolate.spawn` 方法来创建和启动新的 Isolate。

举个例子:

```
  static void _longRunningTask() {
    // do long running task here
  }

  void _startTask() {
    Isolate.spawn(_longRunningTask, null);
  }
```

这样可以在 _startTask 方法中调用 Isolate.spawn 方法来启动一个新的 Isolate，并在其中执行 _longRunningTask 方法。

Flutter 框架也提供了一些高级的并发抽象,如 `Future` 和 `Stream`,这些抽象可以让开发者更加简单地实现多任务并行,更加简单易用。

总之，Flutter 使用 Isolates 来实现多任务并行，开发者可以使用 `dart:isolate` 库中的 API或者是 Flutter 提供的高级并发抽象来实现多任务并行。

在 Flutter 中，通常使用 Future 和 Stream 来实现多任务并行。

Future 是一种异步编程模型，可以在将来的某个时刻获取一个值，或者是在某个操作完成后获取结果。

Stream 是一种异步数据源，可以在将来的某个时刻发出一系列事件。

举个例子:

```
Future<String> fetchData() async {
    // Simulate a network request
    return Future.delayed(Duration(seconds: 2),
() => 'Data from the internet');
}

void someFunction() {
fetchData().then((data) {
// Do something with the data
```

```
print(data);
});
}
```

在这个例子中,我们调用了fetchData() 这个异步函数，并在它返回结果之后使用then() 方法来处理返回的数据。

Stream 也是一样,可以使用 StreamBuilder 或者是 Listenable 等组件来监听一个Stream,并在数据变化时进行响应。

总之,Flutter 提供了丰富的并发抽象来让开发者更加简单地实现多任务并行,使用Future,Stream等抽象可以避免复杂的线程管理,提高开发效率.

1、介绍一下项目中flutter的运用?

答：在项目中使用 Flutter 时，可以考虑以下几点：

- 选择正确的 Widget 类型，避免不必要的重绘
- 使用 isolate 处理计算密集型任务
- 使用 Stream 进行异步事件处理

OC 与 Flutter 交互需要注意以下几点：

- 需要使用 methodChannel 进行通信
- 需要确保 iOS 和 Flutter 端使用相同的 method channel 名称
- 需要在 iOS 端使用 flutter_viewcontroller 进行渲染
- 需要在 Swift 端转换数据类型，以便与 Flutter 端进行交互
  在 Swift 代码中，可以使用 `FlutterMethodCall` 的 invokeMethod 函数向 Flutter 端传递 Int 值。首先，在 Flutter 端通过 `MethodChannel` 注册接收函数，其中参数是 Int 类型，然后在 Swift 代码中调用 invokeMethod 函数，并将 Int 值作为参数传入。

  ```
  // Swift 代码:
  let channel = FlutterMethodChannel(name: "test_channel",
  binaryMessenger: controller)
  channel.invokeMethod("testMethod", arguments: 123)
  ```

  ```
  //Flutter 代码:
  MethodChannel('test_channel').setMethodCallHandler((call) {
  ```

```
  if (call.method == 'testMethod') {
    int data = call.arguments;
    // 处理传递的 Int 值
  }
});
```

注意:

- 需要在 iOS 端使用 FlutterViewController 进行渲染， 使用 FlutterViewController 来渲染 flutter 页面，或者使用 FlutterEngine 来渲染 flutter 页面。

- 需要确保 iOS 和 Flutter 端使用相同的 method channel 名称， 为了能在 iOS 与 flutter 之间进行通信,需要在 iOS 端和 flutter 端使用相同的 method channel 名称。

- 需要在 Swift 端转换数据类型，Swift 与 Flutter 之间的数据交互可能需要进行类型转换。例如, Swift 中的 String 和 Flutter 中的 String 是不同类型的，在交互时需要进行转换。

  在项目中使用Flutter 与原生语言交互时,需要注意这些细节,以便正确的实现交互并保证项目的正常运行.

2、flutter引擎方面?

答：Flutter的渲染引擎是基于Skia的。Skia是一个跨平台的2D图形库，它提供了丰富的图形接口。Flutter通过使用Skia作为渲染引擎，可以在多个平台上提供一致的图形效果。

Flutter的热重载功能可以在开发过程中实时地看到代码更改的效果。当开发者修改代码时，Flutter会在不重启应用程序的情况下重新加载代码，并在UI上实时显示修改的效果。

Flutter还提供了一个Dart虚拟机，可以在多个平台上运行Dart代码。Flutter应用程序是由Dart代码驱动的，并且Flutter框架本身也是由Dart代码编写的。

3、flutter 三棵树以及联系?

答：Flutter中的三棵树指的是框架的三个层次：widget树，元素树和渲染树。

- widget树定义了用户界面的结构和布局，由widget组成，它是用户界面的基本构建块。
- 元素树是从widget树创建的，表示用户界面的有状态元素。元素是框架创建的运行时对象，用于管理widget树中widget的状态和布局。
- 渲染树是从元素树创建的，表示用户界面的视觉元素。渲染树由框架使用，将用户界面绘制到屏幕上。

这三棵树是密切相关的，并协同工作来在Flutter中构建和呈现用户界面。widget树用于定义用户界面的结构和布局，元素树用于管理状态和布局，渲染树用于将用户界面呈现给用户。

4、封装一个组件的方式?

答：在 Flutter 中封装一个组件需要以下几步：

1. 创建一个新的 Dart 类，继承自 `StatefulWidget` 或 `StatelessWidget`。
2. 在新类中定义组件的属性和状态。
3. 实现组件的 `build` 方法，返回组件的布局。
4. 在其他地方使用这个组件。

下面是一个简单的例子：

```
class MyButton extends StatelessWidget {
  final String text;
  final Function onPressed;

  MyButton({this.text, this.onPressed});

  @override
  Widget build(BuildContext context) {
    return RaisedButton(
      child: Text(text),
      onPressed: onPressed,
    );
  }
}
```

在上面的例子中，我们创建了一个名为 `MyButton` 的组件。这个组件有两个参数：`text` 和 `onPressed`。在 `build` 方法中，我们返回了一个 `RaisedButton` 组件，将参数传给了它。

现在可以在其他地方使用这个组件了。

```
MyButton(text: 'Click me', onPressed: () {
  print('Button clicked');
});
```

这样就可以通过调用 MyButton来使用这个组件了,可以根据实际需求来添加属性和方法,使用起来更加灵活。

6、原生的内存管理?

答：Flutter是一个跨平台的应用程序框架，它使用Dart语言来编写应用程序，并在各种平台上通过渲染器来呈现UI。由于Flutter是使用Dart语言编写的，因此内存管理主要是通过Dart语言来实现。

Dart语言使用了垃圾回收来管理内存，它会自动检测并回收不再使用的内存。这意味着，在Dart中，开发人员不需要显式地释放内存，也不需要考虑内存泄漏的问题。

然而,在Flutter中,需要特别关注的是在页面销毁时释放掉不用的资源和回收内存，例如监听器，动画和定时器等。如果不释放这些资源，它们可能会导致内存泄漏。

此外,Flutter有一些工具来监控和优化应用程序的内存使用情况，如Flutter DevTools和Flutter Memory Inspection。

7、Dart 是如何实现多任务并行的?

答：Dart通过使用Isolates来实现多任务并行。Isolates是独立的Dart运行时环境，它们可以在不同的线程上运行，并且互相之间没有任何共享状态。

使用Isolates可以在多个线程上并行执行多个任务，每个Isolate都有自己的堆和栈，并且运行在自己的线程上。这样可以有效的降低多线程竞争的问题。

Isolates之间可以通过消息传递来进行通信，如果需要在Isolates之间共享数据，可以使用共享内存或者文件。

Dart中主要是使用Future和Isolate来实现多任务并行， Future是用来处理异步任务返回值的对象，Isolate是独立的Dart运行时环境，它们可以在不同的线程上运行，并且互相之间没有任何共享状态。

8、await的原理?

答：`await` 是 Dart 中的一个关键字，用于等待一个异步操作的完成。

在使用 `await` 等待一个异步操作时，程序会暂停当前函数的执行，等待该操作完成。完成后，程序会继续执行当前函数。

它可以用来等待一个Future 对象的返回值,可以在异步操作的回调函数中使用.

举个例子:

```
Future<String> fetchData() async {
  // Simulate a network request
  return Future.delayed(Duration(seconds: 2), () => 'Data from the
internet');
}
```

```
Future<void> someFunction() async {
  String data = await fetchData();
  print(data);
}
```

在这个例子中,使用await关键字等待fetchData()返回值,并在返回之后继续执行其他操作.

总之,await 是 Dart 中用来等待异步操作的关键字,可以让代码更加简洁易读,使用await可以避免回调地狱的情况,使得代码更加结构化.

在 Flutter 中，使用 await 可以让你在等待异步操作完成时不必打断程序的流程。这使得代码更加简洁易读，而且还能保证应用程序在等待异步操作完成时仍然能够响应用户输入等事件。

另外,await 只能在 async 函数中使用,可以通过 async/await 来更好的管理异步操作.

9、future的原理?

答：Flutter中的Future是用来处理异步操作的对象。它表示一个异步操作的未来结果。当异步操作完成时，Future对象会将结果作为参数传递给回调函数。

当你调用Future对象的then()方法时，它会立即返回，并将回调函数存储起来，等待异步操作完成。当异步操作完成时，Future对象会调用存储的回调函数，并将结果作为参数传递给它。

可以使用async/await语法来简化异步操作的代码，使其看起来像同步代码。 async函数返回Future对象，而await关键字用于等待Future对象的结果。

简单来说，Future是用来处理异步任务返回值的对象，比如网络请求，文件读写等异步操作会返回一个Future对象，可以使用then()或async/await来处理这个异步操作的返回值。

10、原生HashMap？

答：在 Flutter 中，可以使用 dart:collection 中的 `HashMap` 来实现原生的 HashMap 功能。

`HashMap` 是一种键值对的数据结构，其中键是唯一的。插入、查询、删除等操作的时间复杂度都是 O(1)。

举个例子:

```
var map = HashMap<String, int>();
map['first'] = 1;
map['second'] = 2;
print(map['first']); // 1
```

在上面的例子中,我们创建了一个 HashMap 对象,并使用了花括号语法来添加键值对,可以通过key来获取对应的值.

AFNetwork 是一个 iOS 平台上的网络库,如果想在 flutter 中使用AFNetwork,需要通过使用 platform channel 来实现.

Platform channel 是 Flutter 提供的一种跨平台通信机制，可以在 Dart 代码中调用原生代码，并在原生代码中调用 Dart 代码。

首先,在 Dart 代码中创建一个 MethodChannel 对象,并在原生代码中对应注册对应的方法.

然后,在 Dart 代码中调用 invokeMethod() 方法,传递参数给原生代码.

原生代码收到参数后,使用AFNetwork进行网络请求,并将结果返回给Dart代码.首先,在 Dart 代码中创建一个 MethodChannel 对象

```
static const platform = const MethodChannel('com.example.afnetwork');
```

然后在原生代码中注册对应的方法

```
Future<String> makeNetworkRequest(String url) async {
    final String result = await platform.invokeMethod('makeNetworkRequest',
{'url': url});
    return result;
}
```

在原生代码中实现对应的方法

```
@override
    Future<dynamic> onMethodCall(MethodCall call, Result result) {
        switch (call.method) {
            case "makeNetworkRequest":
                String url = call.arguments["url"];
                AFHTTPSessionManager *manager = [AFHTTPSessionManager
manager];
                [manager GET:url parameters:nil progress:nil
success:^(NSURLSessionTask *task, id responseObject) {
                    result.success(responseObject);
                } failure:^(NSURLSessionTask *operation, NSError *error) {
                    result.error("Error", error.localizedDescription,
error);
                }];
                break;
            default:
                result.notImplemented();
                break;
        }
    }
```

最后,在Dart代码中调用 makeNetworkRequest() 方法,传递参数给原生代码

```
String data = await
makeNetworkRequest("https://jsonplaceholder.typicode.com/posts");
```

上面的代码仅是示例代码,在实际使用中需要根据具体情况进行修改,这里只是给出了思路.

总之,使用platform channel 可以在flutter中使用AFNetwork进行网络请求,需要在Dart代码中调用原生代码,在原生代码中使用AFNetwork进行网络请求,并将结果返回给Dart代码.

总之,flutter 不支持直接使用AFNetwork进行网络请求,需要通过platform channel 来实现,即在Dart代码中调用原生代码,使用AFNetwork进行网络请求,并将结果返回给Dart代码.

总之, `HashMap` 是 Flutter 中用来实现原生 HashMap 功能的类,可以使用 dart:collection 中的 `HashMap` 来实现键值对数据结构,并进行插入,查询,删除等操作.

11、原生第三方库 okhttp等?

答：Flutter 支持使用原生第三方库，如 okhttp 等。

通过使用 Dart 的 `dart:ffi` 和 `package:ffi` 库可以轻松地在 Flutter 中调用原生代码。

首先，需要在 `pubspec.yaml` 中配置依赖，如：

```
dependencies:
  okhttp: ^3.12.1
```

然后在 dart 代码中使用 import 导入 okhttp 库

```
import 'package:okhttp/okhttp.dart' as okhttp;
```

接着就可以使用 okhttp 库中的 API 来进行网络请求了.

需要注意的是,如果需要使用原生第三方库，需要先在pubspec.yaml 中配置依赖,然后在dart代码中导入,最后就可以使用了.
总之,flutter支持使用原生第三方库，如 okhttp，可以通过使用 Dart 的 `dart:ffi` 和 `package:ffi` 库来在 Flutter 中调用原生代码。使用步骤为配置依赖，导入库，使用库中的 API。这样可以让 Flutter 应用程序能够使用现有的原生第三方库来实现各种功能。

12、flutter setstate刷新机制?

答：Flutter的setState函数用于在更新组件状态时刷新UI。它接受一个回调函数，在回调函数中修改组件的状态，然后调用setState函数来通知Flutter框架进行重绘。当setState被调用时，Flutter会调用组件的build方法来重新构建UI，并在新的UI上显示更新后的状态。

13、flutter动画 自定义view?

答：在 Flutter 中，可以使用 AnimatedWidget 和 AnimatedBuilder 来创建自定义动画。

AnimatedWidget 是一种特殊类型的小部件，它可以在动画期间自动重建自身。它需要一个 Animation 对象作为构造函数的参数，该对象包含动画运行时的信息。使用 AnimatedWidget 可以很容易地在动画期间更新部件的属性。

AnimatedBuilder 是一种更灵活的动画建造器，它允许我们在动画期间重建任何小部件。它接受一个动画构建器函数作为参数，该函数接收当前动画运行时的信息并返回要重建的小部件。

还可以使用AnimationController来控制动画的进度，以及Tween来定义动画的起始值和结束值。

另外还有很多flutter内置的动画组件，如：FadeTransition,ScaleTransition,SizeTransition,RotateTransition,AnimatedPositioned,AnimatedPadding 等，可以根据需求使用。

14、flutter 状态管理 bloc?

答：BLoC (业务逻辑组件) 是一种常用于Flutter状态管理的设计模式。它由Google提出，基于可观察对象和流的概念。

在 BLoC 模式中，应用程序的状态存储在独立的 BLoC 组件中，而不是在单个顶级组件中。这样的好处是将应用程序的业务逻辑和界面逻辑分开，使得应用程序更易于维护和测试。

BLoC 组件通常由两个部分组成：事件处理器和状态流。事件处理器处理来自用户界面的事件，并将它们转换为状态更改。状态流是一个可观察对象，它将状态更改发布到用户界面以进行更新。

BLoC 模式可以使用第三方库如 bloc 或 flutter_bloc来实现。 使用这些库可以减少手写代码，并使实现变得更简单。

Swift知识

1、Swift和Objective-C有什么区别?

1）Swift是强类型（静态）语言，有类型推断，Objective-C弱类型（动态）语言
2）Swift面向协议编程，Objective-C面向对象编程
3）Swift注重值类型，Objective-C注重引用类型
4）Swift支持泛型，Objective-C只支持轻量泛型（给集合添加泛型）
5）Swift支持静态派发（效率高）、动态派发（函数表派发、消息派发）方式，Objective-C支持动态派发（消息派发）方式
6）Swift支持函数式编程（高阶函数）
7）Swift的协议不仅可以被类实现，也可以被Struct和Enum实现

8）Swift有元组类型、支持运算符重载
9）Swift支持命名空间
10）Swift支持默认参数
11）Swift比Objective-C代码更简洁
在 Swift 中，函数是一种引用类型，因此将函数赋值给常量相当于将该函数的地址存储在常量中，而不是将函数的内容复制到常量中。因此，即使将函数赋值给常量，也仍然可以对该函数进行修改。
在 Swift 中，函数是一种引用类型，因此将函数赋值给常量相当于将该函数的地址存储在常量中，而不是将函数的内容复制到常量中。因此，即使将函数赋值给常量，也仍然可以对该函数进行修改；结构体是值类型，常量持有的是结构体的一份拷贝，不能对其修改。而引用类型的常量仅仅是常量指向的内存地址不能变，但其本身的值是可以修改的。

2、讲述讲Swift的派发机制

1）函数的派发机制：静态派发（直接派发）、函数表派发、消息派发
2）Swift派发机制总结：

Swift中所有ValueType（值类型：Struct、Enum）使用直接派发；
Swift中协议的Extensions使用直接派发，初始声明函数使用函数表派发；
Swift中Class中Extensions使用直接派发，初始声明函数使用函数表派发，dynamic修饰的函数使用消息派发；
Swift中NSObject的子类用@nonobjc或final修饰的函数使用直接派发，初始声明函数使用函数表派发，dynamic修饰的Extensions使用消息派发；
3）Swift中函数派发查看方式: 可将Swift代码转换为SIL（中间码）
swiftc -emit-silgen -O example.swift

3、Swift如何显示指定派发方式?

添加final关键字的函数使用直接派发
添加static关键字函数使用直接派发
添加dynamic关键字函数使用消息派发
添加@objc关键字的函数使用消息派发
@dynamic 和 @objc组合修饰函数使用直接派发
extension 扩展的直接是静态派发
添加@inline关键字的函数会告诉编译器可以使用直接派发

4、Struct和Class的区别?

1）Struct不支持继承，Class支持继承
2）Struct是值类型，Class是引用类型
3）Struct使用let创建不可变，Class使用let创建可变
4）Struct无法修改自身属性值，函数需要添加mutating关键字

5）Struct不需要deinit方法，因为值类型不关系引用计数，Class需要deinit方法
6）Struct初始化方法是基于属性的
需要一种拥有动态特性（或者说多态特性）而且还得效率高速度快的一种数据结构，struct这个东西就被定制出来了， 动态性：struct+协议 效率高：栈存储
优点：值类型，速度快、变量 方法 下标基本都能满足需求； 缺点： 不能继承了

5、Swift中的常量和Objective-C中的常量有啥区别?

Objective-C中的常量(const)是编译期决定的，Swift中的常量(let)是运行时确定的

6、?，??的区别

？用来声明可选值，如果变量未初始化则自动初始化nil；在操作可选值时，如果可选值是nil则不响应后续的操作；使用as?进行向下转型操作；
?? 用来判断左侧可选值非空（not nil）时返回左侧值可选值，左侧可选值为空（nil）则返回右侧的值。

? : Optional其实是个enum，里面有None和Some两种类型。其实所谓的nil就是Optional.None， 非nil就是Optional.Some， 然后会通过Some(T)包装（wrap）原始值，这也是为什么在使用Optional的时候要拆包（从enum里取出来原始值）的原因。这里是enum Optional的定义（对于Optional 值，不能直接进行操作，否则会报错）

7、Swift中mutating的作用

Swift中协议是可以被Struct和Enum实现的，mutating关键字是为了能在被修饰的函数中修改Struct或Enum的变量值，对Class完全透明。

8、Set（集合类型）的使用场景

Set存储值类型相同、无序、去重

9、final关键词的用法

final关键词的作用：它修饰的类、方法、变量是不能被继承或重写的，编译器会报错。另外，通过它可以显示的指定函数的派发机制。

10、lazy关键词的用法

lazy关键词的作用：指定延时加载（懒加载），懒加载存储属性只会在首次使用时才会计算初始值属性。懒加载属性必须声明（var）为变量，因为常量属性（let）初始化之前会有值。
lazy修饰的属性非线程安全的。**1.struct是值类型，class是引用类型。**

值类型的变量直接包含它们的数据，对于值类型都有它们自己的数据副本，因此对一个变量操作不可能影响另一个变量。

引用类型的变量存储对他们的数据引用，因此后者称为对象，因此对一个变量操作可能影响另一个变量所引用的对象。

2.二者的本质区别：

struct是深拷贝，拷贝的是内容；class是浅拷贝，拷贝的是指针。

3.property的初始化不同：

class 在初始化时不能直接把 property 放在 默认的constructor 的参数里，而是需要自己创建一个带参数的constructor；而struct可以，把属性放在默认的constructor 的参数里。

4.变量赋值方式不同：

struct是值拷贝；class是引用拷贝。

5.immutable变量：

swift的可变内容和不可变内容用var和let来甄别，如果初始为let的变量再去修改会发生编译错误。struct遵循这一特性；class不存在这样的问题。

6.mutating function：

struct 和 class 的差别是 struct 的 function 要去改变 property 的值的时候要加上 mutating，而 class 不用。

7.继承：

struct不可以继承，class可以继承。

8.struct比class更轻量：

struct分配在栈中，class分配在堆中。

open与public的区别

public:可以别任何人访问，但是不可以被其他module复写和继承。

open:可以被任何人访问，可以被继承和复写。

swift中struct作为数据模型

优点:

1.安全性:因为 Struct 是用值类型传递的，它们没有引用计数。

2.内存:由于他们没有引用数，他们不会因为循环引用导致内存泄漏。

速度:值类型通常来说是以栈的形式分配的，而不是用堆。因此他们比 Class 要快很多!

3.拷贝:Objective-C 里拷贝一个对象,你必须选用正确的拷贝类型（深拷贝、浅拷贝）,而值类型的拷贝则非常轻松!

4.线程安全:值类型是自动线程安全的。无论你从哪个线程去访问你的 Struct ，都非常简单。

缺点:

1.Objective-C与swift混合开发：OC调用的swift代码必须继承于NSObject。

2.继承：struct不能相互继承。

3.NSUserDefaults：Struct 不能被序列化成 NSData 对象

Swift代码复用的方式有哪些?

1.继承

2.在swift 文件里直接写方法，相当于一个全局函数

3.extension 给类直接扩展方法

Array、Set、Dictionary 三者的区别

Set:是用来存储相同类型、没有确定顺序、且不重复的值的集

Array:是有序数据的集

Dictionary:是无序的键值对的集

map、filter、reduce 的作用

map : 映射 ， 将一个元素根据某个函数 映射 成另一个元素（可以是同类型，也可以是不同类型）

filter : 过滤 ， 将一个元素传入闭包中，如果返回的是false ， 就过滤掉

reduce ： 先映射后融合 ， 将数组中的所有元素映射融合在一起。

map 与 flatmap 的区别

map:作用于数组中的每个元素，闭包返回一个变换后的元素，接着将所有这些变换后的元素组成一个新的数组。

flatMap:功能和map类似，区别在于flatMap可以去nil,能够将可选类型(optional)转换为非可选类型(non-optionals),把数组中存有数组的数组 一同打开变成一个新的数组(数组降维)

copy on write

写时复制：每当数组被改变，它首先检查它对存储缓冲区 的引用是否是唯一的，或者说，检查数组本身是不是这块缓冲区的唯一拥有者。如果是，那么 缓冲区可以进行原地变更;也不会有复制被进行。不过，如果缓冲区有一个以上的持有者 (如本 例中)，那么数组就需要先进行复制，然后对复制的值进行变化，而保持其他的持有者不受影响。

在Swift提供一个函数isKnownUniquelyReferenced，能检查一个类的实例是不是唯一的引用，如果是，说明实例没有被共享

eg: isKnownUniquelyReferenced(&object)

获取当前代码的函数名和行号

函数名:#function

行号:#line

文件名:#file

guard、defer、where

defer:defer所声明的 block 会在当前代码执行退出后被调用，如果有多个 defer, 那么后加入的先执行

guard:可以理解为拦截，凡是不满足 guard 后面条件的，都不会再执行下面的代码。

where:在Swift语法里where关键字的作用跟SQL的where一样， 即附加条件判断。where关键字可以用在集合遍历、switch/case、协议中； Swift3时if let和guard场景的where已经被Swift4的逗号取代

String 与 NSString 的关系与区别

String为Swift的Struct结构，值类型；NSString为OC对象，引用类型，能够互相转换

throws 和 rethrows 的用法与作用

throws 用在函数上, 表示这个函数会抛出错误.

有两种情况会抛出错误, 一种是直接使用 throw 抛出, 另一种是调用其他抛出异常的函数时, 直接使用 try xx 没有处理异常.

```
enum DivideError: Error {
    case EqualZeroError;
}
func divide(_ a: Double, _ b: Double) throws -> Double {
    guard b != Double(0) else {
        throw DivideError.EqualZeroError
    }
    return a / b
}
func split(pieces: Int) throws -> Double {
    return try divide(1, Double(pieces))
}
```

rethrows 与 throws 类似, 不过只适用于参数中有函数, 且函数会抛出异常的情况, rethrows 可以用 throws 替换, 反过来不行

```
func processNumber(a: Double, b: Double, function: (Double, Double) throws -
> Double) rethrows -> Double {
    return try function(a, b)
}
```

try?和 try!

这两个都用于处理可抛出异常的函数, 使用这两个关键字可以不用写 do catch.

区别在于, try? 在用于处理可抛出异常函数时, 如果函数抛出异常, 则返回 nil, 否则返回函数返回值的可选值

而 try! 则在函数抛出异常的时候崩溃, 否则则返会函数返回值, 相当于(try? xxx)!

associatedtype 的作用

简单来说就是 protocol 使用的泛型

Swift 下的 KVO , KVC

KVO, KVC 都是Objective-C 运行时的特性, Swift 是不具有的, 想要使用, 必须要继承 NSObject, 自然, 继承都没有的结构体也是没有 KVO, KVC 的.

KVC：Swift 下的 KVC 用起来很简单, 只要继承 NSObject 就行了

KVO：KVO 就稍微麻烦一些了,由于 Swift 为了效率, 默认禁用了动态派发, 因此想用 Swift 来实现 KVO, 除了继承NSObject，还需要将想要观测的对象标记为 dynamic

@objc

OC 是基于运行时，遵循了 KVC 和动态派发，而 Swift 是静态语言，为了追求性能，在编译时就已经确定，而不需要在运行时的，在 Swift 类型文件中，为了解决这个问题，需要暴露给 OC 使用的任何地方（类，属性，方法等）的生命前面加上 @objc 修饰符

如果用 Swift 写的 class 是继承 NSObject 的话， Swift 会默认自动为所有非 private 的类和成员加上@objc

自定义下标获取

实现 subscript 即可, 索引除了数字之外, 其他类型也是可以的

```
extension AnyList {
    subscript(index: Int) -> T{
        return self.list[index]
    }

    subscript(indexString: String) -> T?{
        guard let index = Int(indexString) else {
            return nil
        }
        return self.list[index]
    }
}
```

lazy 的作用

懒加载, 当属性要使用的时候, 才去完成初始化

一个类型表示选项，可以同时表示有几个选项选中（类似 UIViewAnimationOptions），用什么类型表示

需要实现OptionSet, 一般使用 struct 实现. 由于 OptionSet 要求有一个不可失败的init(rawValue:)构造器, 而 枚举无法做到这一点(枚举的原始值构造器是可失败的, 而且有些组合值, 是没办法用一个枚举值表示的)

```
struct SomeOption: OptionSet {
    let rawValue: Int
    static let option1 = SomeOption(rawValue: 1 << 0)
    static let option2 =  SomeOption(rawValue:1 << 1)
    static let option3 =  SomeOption(rawValue:1 << 2)
}
let options: SomeOption = [.option1, .option2]
```

static和class的区别

static 可以在类、结构体、或者枚举中使用。而 class 只能在类中使用。

static 可以修饰存储属性，static 修饰的存储属性称为静态变量(常量)。而 class 不能修饰存储属性。

static 修饰的计算属性不能被重写。而 class 修饰的可以被重写。

static 修饰的静态方法不能被重写。而 class 修饰的类方法可以被重写。

class 修饰的计算属性被重写时，可以使用 static 让其变为静态属性。

class 修饰的类方法被重写时，可以使用 static 让方法变为静态方法

class关键字指定的类方法可以被子类重写，但是用static关键字指定的类方法是不能被子类重写的

mutating

mutating用于函数的最前面,用于告诉编译器这个方法会改变自身.

swift增强了结构体和枚举的使用,结构体和枚举也可以有构造方法和实例方法,但结构体和枚举是值类型,如果我们要在实例方法中修改当前类型的实例属性的值或当前对象实例的值,必须在func前面添加mutating关键字,表示当前方法将会将会修改它相关联的对象实例的实例属性值.

协议中,当在结构体或者枚举实现协议方法时,若对自身属性作修改,需要将协议的方法声明为mutating,对类无影响.

在扩展中,同样若对自身属性作修改,需要将方法声明为mutating

你如何使用swift进行内存管理的?

答：Swift 使用自动引用计数（ARC）来管理内存，并且在大多数情况下会自动帮助开发者处理内存管理，但仍需要开发者注意以下几点：

1. 避免循环引用：当两个对象互相持有引用时，就会产生循环引用，导致无法被回收。
2. 使用弱引用和无主引用：在需要防止循环引用的地方使用 weak 或者 unowned 关键字。//"无主引用" 是一种不强制对象关系的引用。
3. 使用 nil 解除引用：明确地将对象设置为 nil 可以立刻释放内存。
4. 注意类的生命周期：确保类在生命周期结束时释放相关资源。

swift多线程

1.Thread

```
//方式1：使用类方法，不需要手动启动线程
Thread.detachNewThreadSelector(#selector(ViewController.downloadImage),
toTarget: self, with: nil)

//方式2：实例方法-便利构造器，需调用start()启动线程
let myThread = Thread(target: self, selector: #selector(thread), object:
nil)
myThread.start()
```

线程同步：线程同步方法通过锁来实现，每个线程都只用一个锁，这个锁与一个特定的线程关联。

```
//定义两个线程
var thread1:Thread?
var thread2:Thread?
//定义两个线程条件，用于锁住线程
let condition1 = NSCondition()
let condition2 = NSCondition()

override func viewDidLoad() {
   super.viewDidLoad()
   thread2 = Thread(target: self, selector:
#selector(ViewController.method2), object: nil)
   thread1 = Thread(target: self, selector:
#selector(ViewController.method1), object: nil)
   thread1?.start()
}

//定义两方法，用于两个线程调用
func method1(sender:AnyObject){
   for i in 0 ..< 10 {
      print("Thread 1 running (i)")
      sleep(1)
      if i == 2 {
         thread2?.start() //启动线程2
         //本线程（thread1）锁定
         condition1.lock()
         condition1.wait()
         condition1.unlock()
      }
   }
   print("Thread 1 over")
   //线程2激活
   condition2.signal()
}
```

```
//方法2
func method2(sender:AnyObject){
    for i in 0 ..< 10 {
        print("Thread 2 running (i)")
        sleep(1)
        if i == 2 {
            //线程1激活
            condition1.signal()
            //本线程（thread2）锁定
            condition2.lock()
            condition2.wait()
            condition2.unlock()
         }
    }
    print("Thread 2 over")
}
```

2.Operation和OperationQueue

2.1 直接用定义好的子类：BlockOperation。

```
let operation = BlockOperation(block: { [weak self] in
    // 执行代码return
})
//创建一个NSOperationQueue实例并添加operation
let queue = OperationQueue()
queue.addOperation(operation)
```

2.2 继承Operation

把Operation子类的对象放入OperationQueue队列中，一旦这个对象被加入到队列，队列就开始处理这个对象，直到这个对象的所有操作完成，然后它被队列释放。

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        //创建线程对象
        let downloadImageOperation = DownloadImageOperation()
        //创建一个OperationQueue实例并添加operation
        let queue = OperationQueue()
        queue.addOperation(downloadImageOperation)
    }
}
class DownloadImageOperation: Operation {
```

```
    override func main(){
        let imageUrl = "http://hangge.com/blog/images/logo.png"
        let data = try! Data(contentsOf: URL(string: imageUrl)!)
        print(data.count)
    }
}
```

2.3 其他方法

```
//队列设置并发数
queue.maxConcurrentOperationCount = 5
//队列取消所有线程操作
queue.cancelAllOperations()
//给operation设置回调
operation.completionBlock = { () -> Void in
    print("--- operation.completionBlock ---")
}
```

3.GCD

3.1 创建队列

```
//------自己创建队列
//创建串行队列
let serial = DispatchQueue(label: "serialQueue1")
//创建并行队列
let concurrent = DispatchQueue(label: "concurrentQueue1", attributes:
.concurrent)

//------系统global队列
//Global Dispatch Queue有4个执行优先级:
//.userInitiated  高
//.default  正常
//.utility  低
//.background 非常低的优先级（这个优先级只用于不太关心完成时间的真正的后台任务）
let globalQueue = DispatchQueue.global(qos: .default)

//------系统主线程队列
//因为主线程只有一个，所有这自然是串行队列。一起跟UI有关的操作必须放在主线程中执行。
let mainQueue = DispatchQueue.main
```

3.2 添加任务到队列

async异步追加Block块（async函数不做任何等待）

```
DispatchQueue.global(qos: .default).async {
    //处理耗时操作的代码块...

    //操作完成，调用主线程来刷新界面
    DispatchQueue.main.async {
        print("main refresh")
    }
}
```

sync同步追加Block块

```
//同步追加Block块，与上面相反。在追加Block结束之前，sync函数会一直等待，等待队列前面的所有任务完成后才能执行追加的任务。
//添加同步代码块到global队列
//不会造成死锁，但会一直等待代码块执行完毕
DispatchQueue.global(qos: .default).sync {
    print("sync1")
}
print("end1")


//添加同步代码块到main队列
//会引起死锁
//因为在主线程里面添加一个任务，因为是同步，所以要等添加的任务执行完毕后才能继续走下去。但是新添加的任务排在
//队列的末尾，要执行完成必须等前面的任务执行完成，由此又回到了第一步，程序卡死
DispatchQueue.main.sync {
    print("sync2")
}
```

3.3 暂停或者继续队列

这两个函数是异步的，而且只在不同的blocks之间生效，对已经正在执行的任务没有影响。suspend()后，追加到Dispatch Queue中尚未执行的任务在此之后停止执行。而resume()则使得这些任务能够继续执行。

```
//创建并行队列
let conQueue = DispatchQueue(label: "concurrentQueue1", attributes:
.concurrent)
//暂停一个队列
conQueue.suspend()
//继续队列
conQueue.resume()
```

3.4 asyncAfter 延迟调用

asyncAfter 并不是在指定时间后执行任务处理，而是在指定时间后把任务追加到queue里面。因此会有少许延迟。注意，我们不能（直接）取消我们已经提交到 asyncAfter 里的代码。

```
//延时2秒执行
DispatchQueue.global(qos: .default).asyncAfter(deadline: DispatchTime.now()
+ 2.0) {
    print("after!")
}
```

如果需要取消正在等待执行的Block操作，我们可以先将这个Block封装到DispatchWorkItem对象中，然后对其发送cancle，来取消一个正在等待执行的block。

```
//将要执行的操作封装到DispatchWorkItem中
let task = DispatchWorkItem { print("after!") }
//延时2秒执行
DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + 2, execute:
task)
//取消任务
task.cancel()
```

3.5 DispatchWorkItem

DispatchWorkItem可以将任务封装成DispatchWorkItem对象。

可以调用workItem的`perform()`函数执行任务，也可以将workItem追加到DispatchQueue或DispatchGroup中。以上所有传block的地方都可换成DispatchWorkItem对象。DispatchQueue还可以使用`notify`函数观察workItem中的任务执行结束，以及通过`cancel()`函数取消任务。

另外，workItem也可以像DispatchGroup一样调用`wait()`函数等待任务完成。需要注意的是，追加workItem的队列或调用`perform()`所在的队列不能与调用`workItem.wait()`的队列是同一个队列，否则会出现线程死锁。

```
// 初始化方法
init(qos: DispatchQoS, flags: DispatchWorkItemFlags, block: () -> Void)

let workItem = DispatchWorkItem.init {
      print("执行任务")
}
```

3.6 DispatchQueue.concurrentPerform

sync函数和Dispatch Group的关联API。会按指定次数异步执行任务，并且会等待指定次数的任务全部执行完成，即会阻塞线程。建议在子线程中使用。

```
DispatchQueue.global().async {
     DispatchQueue.concurrentPerform(iterations: 5) { (i) in
         print("执行任务(i+1)")
     }
     print("任务执行完成")
}
```

3.7 DispatchSemaphore

信号量:用于控制访问资源的数量。比如系统有两个资源可以被利用，同时有三个线程要访问，只能允许两个线程访问，第三个会等待资源被释放后再访问。
信号量的初始化方法：`DispatchSemaphore.init(value: Int)`，value表示允许访问资源的线程数量，当value为0时对访问资源的线程没有限制。
信号量配套使用`wait()`函数与`signal()`函数控制访问资源。

wait函数会阻塞当前线程直到信号量计数大于或等于1，当信号量大于或等于1时，将信号量计数-1, 然后执行后面的代码。signal()函数会将信号量计数+1

信号量是GCD同步的一种方式。前面介绍过的`DispatchWorkItemFlags.barrier`是对queue中的任务进行批量同步处理，sync函数是对queue中的任务单个同步处理，而DispatchSemaphore是对queue中的某个任务中的某部分（某段代码）同步处理。此时将`DispatchSemaphore.init(value: Int)`中的参数value传入1。

```
var arr = [Int]()
let semaphore = DispatchSemaphore.init(value: 1) // 创建信号量，控制同时访问资源的线程数为1
for i in 0...100 {
    DispatchQueue.global().async {
        /*
        其他并发操作
        */
        semaphore.wait() // 如果信号量计数>=1,将信号量计数减1；如果信号量计数<1，阻塞线程直到信号量计数>=1
        arr.append(i)
        semaphore.signal() // 信号量计加1
        /*
        其他并发操作
        */
     }
}
```

什么是optional类型，它是用来解决什么问题的?

答案：optional类型被用来表示任何类型的变量都可以表示缺少值。在Objective-C中，引用类型的变量是可以缺少值得，并且使用nil作为缺少值。基本的数据类型如int 或者float没有这种功能。

Swift用optional扩展了在基本数据类型和引用类型中缺少值的概念。一个optional类型的变量，在任何时候都可以保存一个值或者为nil。

问题2- Swift 1.0 or later

在Swfit中,什么时候用结构体，什么时候用类?

答案：一直都有这样的争论：到底是用类的做法优于用结构体，还是用结构体的做法优于类。函数式编程倾向于值类型，面向对象编程更喜欢类。

在Swift 中，类和结构体有许多不同的特性。下面是两者不同的总结：
类支持继承，结构体不支持。
类是引用类型，结构体是值类型

并没有通用的规则决定结构体和类哪一个更好用。一般的建议是使用最小的工具来完成你的目标，但是有一个好的经验是多使用结构体，除非你用了继承和引用语义。
注意：在运行时，结构体的在性能方面更优于类，原因是结构体的方法调用是静态绑定，而类的方法调用是动态实现的。这就是尽可能得使用结构体代替类的又一个好的原因。

问题3- Swift 1.0 or later

什么是泛型？泛型是用来解决什么问题的?

答案：泛型是用来使类型和算法安全的工作的一种类型。在Swift中，在函数和数据结构中都可以使用泛型，例如类、结构体和枚举。

泛型一般是用来解决代码复用的问题。常见的一种情况是，你有一个函数，它带有一个参数，参数类型是A，然而当参数类型改变成B的时候，你不得不复制这个函数。

例如，下面的代码中第二个函数就是复制第一个函数——它仅仅是用String类型代替了Integer类型。

```
func areIntEqual(x: Int, _ y: Int) -> Bool {
  return x == y
}

func areStringsEqual(x: String, _ y: String) -> Bool {
  return x == y
}
```

```
areStringsEqual("ray", "ray") // true
areIntEqual(1, 1) // true
```

Objective-C开发人员可能想到用NSObject类来解决这个问题，代码如下：

```
import Foundation

func areTheyEqual(x: NSObject, _ y: NSObject) -> Bool {
  return x == y
}

areTheyEqual("ray", "ray") // true
areTheyEqual(1, 1) // true
```

这个代码会按照预期的方式工作,但是它在编译时不安全。它允许字符串和整数相比较,像这样:

```
areTheyEqual(1, "ray")
```

应用程序不会崩溃,但是允许字符串和整数相比较可能不是预想的结果。

通过采用泛型,可以合并这两个函数为一个并同时保持类型安全。下面是代码实现:

```
func areTheyEqual(x: T, _ y: T) -> Bool {
  return x == y
}

areTheyEqual("ray", "ray")
areTheyEqual(1, 1)
```

上面的例子是测试两个参数是否相等，这两个参数的类型受到约束都必须遵循Equatable协议。上面的代码达到预想的结果,并且防止了传递不同类型的参数。

问题4- Swift 1.0 or later

哪些情况下你不得不使用隐式拆包？说明原因。

答案：对optional变量使用隐式拆包最常见的原因如下：

1、对象属性在初始化的时候不能nil,否则不能被初始化。典型的例子是Interface Builder outlet类型的属性，它总是在它的拥有者初始化之后再初始化。在这种特定的情况下，假设它在Interface Builder中被正确的配置——outlet被使用之前，保证它不为nil。

2、解决强引用的循环问题——当两个实例对象相互引用，并且对引用的实例对象的值要求不能为nil时候。在这种情况下，引用的一方可以标记为unowned,另一方使用隐式拆包。

建议：除非必要，不要对option类型使用隐式拆包。使用不当会增加运行时崩溃的可能性。在某些情况下,崩溃可能是有意的行为,但有更好的方法来达到相同的结果,例如,通过使用fatalError( )函数。

问题5- Swift 1.0 or later

对一个optional变量拆包有多少种方法？并在安全方面进行评价。

答案：

- 强制拆包！ 操作符——不安全
- 隐式拆包变量声明——大多数情况下不安全
- 可选绑定——安全
- 自判断链接（optional chaining）——安全
- nil coalescing 运算符（空值合并运算符）——安全
- Swift 2.0 的新特性 guard 语句——安全
- Swift 2.0 的新特性optional pattern（可选模式） ——安全（@Kametrixom支持）

中级

问题1- Swift 1.0 or later

Swift 是面向对象编程语言还是函数式编程语言?

答案：Swift是一种混合编程语言，它包含这两种编程模式。它实现了面向对象的三个基本原则:

- 封装
- 继承
- 多态

说道Swift作为一种函数式编程语言，我们就不得不说一下什么是函数式编程。有很多不同的方法去定义函数式编程语言，但是他们表达的意义相同。

最常见的定义来自维基百科：...它是一种编程规范...它把电脑运算当做数学函数计算，避免状态改变和数据改变。

很难说Swift是一个成熟的函数式语言，但是它已经具备了函数式语言的基础。

问题2- Swift 1.0 or later

下面的功能特性都包含在Swift中吗?

1、泛型类

2、泛型结构体

3、泛型协议

答案：

- Swift 包含1和2特性。泛型可以在类、结构体、枚举、全局函数或者方法中使用。
- 3是通过typealias部分实现的。typealias不是一个泛型类型,它只是一个占位符的名字。它通常是作为关联类型被引用，只有协议被一个类型引用的时候它才被定义。

问题3- Swift 1.0 or later

在Objective-C中，一个常量可以这样定义：

```
const int number = 0;
```

类似的Swift是这样定义的：

```
let number = 0
```

两者之间有什么不同吗？如果有，请说明原因。

答案：const常量是一个在编译时或者编译解析时被初始化的变量。通过let创建的是一个运行时常量，是不可变得。它可以使用stattic 或者dynamic关键字来初始化。谨记它的的值只能被分配一次。

问题4- Swift 1.0 or later

声明一个静态属性或者函数，我们常常使用值类型的static修饰符。下面就是一个结构体的例子：

```
struct Sun {
static func illuminate() {}
}
```

对类来说,使用static 或者class修饰符，都是可以的。它们使用后的效果是一样的，但是本质上是不同的。能解释一下为什么不同吗?

答案：

static修饰的属性或者修饰的函数都不可以重写。但是使用class修饰符，你可以重写属性或者函数。

当static在类中应用的时候，static就成为class final的一个别名。

例如，在下面的代码中，当你尝试重写illuminate()函数时，编译器就会报错：

```
class Star {
  class func spin() {}
  static func illuminate() {}
}

class Sun : Star {
  override class func spin() {
    super.spin()
  }
  override static func illuminate() { // error: class method overrides a
'final' class method
    super.illuminate()
  }
}
```

问题5- Swift 1.0 or later

你能通过extension(扩展)保存一个属性吗？请解释一下原因。

答案：不能。扩展可以给当前的类型添加新的行为，但是不能改变本身的类型或者本身的接口。如果你添加一个新的可存储的属性，你需要额外的内存来存储新的值。扩展并不能实现这样的任务。

高级

问题1- Swift 1.2

在Swift1.2版本中，你能解释一下用泛型来声明枚举的问题吗？拿下面代码中Either枚举来举例说明吧，它有两个泛型类型的参数T和V，参数T在关联值类型为left情况下使用，参数V在关联值为rihgt情况下使用，代码如下：

```
enum Either{
  case Left(T)
  case Right(V)
}
```

提示：验证上面的条件，需要在Xcode工程里面，而不是在Playgroud中。同时注意，这个问题跟Swift1.2相关，所以Xcode的版本必须是6.4以上。

答案：上面的代码会出现编译错误：

```
unimplemented IR generation feature non-fixed multi-payload enum layout
```

问题是T的内存大小不能确定前期,因为它依赖于T类型本身,但enum情况下需要一个固定大小的有效载荷。

最常用的解决方法是讲泛类型用引用类型包装起来，通常称为box,代码如下：

```
class Box{
  let value: T
  init(_ value: T) {
    self.value = value
  }
}
enum Either{
  case Left(Box)
  case Right(Box)
}
```

这个问题在Swift1.0及之后的版本出现，但是Swift2.0的时候，被解决了。

问题2- Swift 1.0 or later

闭包是引用类型吗?

答案：闭包是引用类型。如果一个闭包被分配给一个变量,这个变量复制给另一个变量,那么他们引用的是同一个闭包，他们的捕捉列表也会被复制。

问题3- Swift 1.0 or later

UInt类型是用来存储无符号整型的。下面的代码实现了一个有符号整型转换的初始化方法：

```
init(_ value: Int)
```

然而，在下面的代码中，当你给一个负值的时候，它会产生一个编译时错误：

```
let myNegative = UInt(-1)
```

我们知道负数的内部结构是使用二进制补码的正数，在保持这个负数内存地址不变的情况下，如何把一个负整数转换成一个无符号的整数?

答案：使用下面的初始化方法：

```
UInt(bitPattern: Int)
```

问题4- Swift 1.0 or later

描述一种在Swift中出现循环引用的情况，并说明怎么解决。

答案：循环引用出现在当两个实例对象相互拥有强引用关系的时候，这会造成内存泄露，原因是这两个对像都不会被释放。只要一个对象被另一个对象强引用，那么该对象就不能被释放，由于强引用的存在，每个对象都会保持对方存在。

解决这个问题的方法是，用weak或者unowned引用代替其中一个的强引用，来打破循环引用。

问题5- Swift 2.0 or later

Swift2.0 增加了一个新的关键字来实现递归枚举。下面的例子是一个枚举类型，它在Node条件下有两个相关联的值类型T和List：

```
enum List{
 case Node(T, List)
}
```

什么关键字可以实现递归枚举?

答案：indirect 关键值可以允许递归枚举，代码如下：

```
enum List{
 indirect case Cons(T, List)
}
```

问题6- Swift嵌套函数在哪存的什么地方

答：Swift中的嵌套函数是存储在调用它的外部函数中的. 它们只能在外部函数的内部访问,并不能在外部函数外部访问.(嵌套函数只能在它所嵌套在的外部函数中被调用和使用,不能在外部函数外部被直接调用或使用. 外部函数必须先被调用才能使嵌套函数可用. 例如,如果有一个名为 outerFunction() 的外部函数和一个名为 nestedFunction() 的嵌套函数,那么在 outerFunction() 外部调用 nestedFunction() 是不允许的,但是在 outerFunction() 内部调用 nestedFunction() 是允许的。)