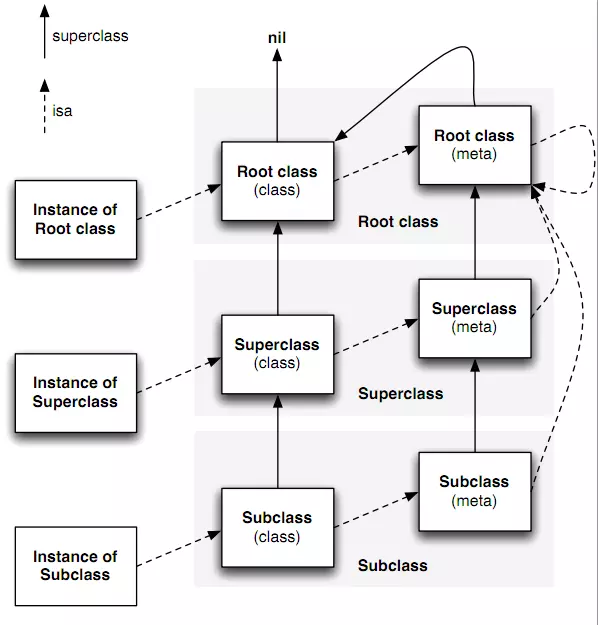
# iOS

## Runtime

1. 一个objc对象的isa的指针指向什么？有什么作用？

指向他的类对象,从而可以找到对象上的方法

详解：下图很好的描述了对象，类，元类之间的关系:



1. \_objc\_msgForward函数是做什么的，直接调用它将会发生什么？

\_objc\_msgForward是 IMP 类型，用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，\_objc\_msgForward会尝试做消息转发。

List itemresolveInstanceMethod:方法 (或resolveClassMethod:)。//解析实例方法

List itemforwardingTargetForSelector:方法 转发目标

List itemmethodSignatureForSelector:方法 方法签名

List itemforwardInvocation:方法 提出调用

List itemdoesNotRecognizeSelector: 方法 不识别

## 二、Runloop

1. 为什么 NSTimer 有时候不好使？

因为创建的 NSTimer 默认是被加入到了 defaultMode，所以当 Runloop 的 Mode 变化时，当前的 NSTimer 就不会工作了。

1. PerformSelector 的实现原理？

当调用 NSObject 的 performSelecter:afterDelay: 后，实际上其内部会创建一个 Timer 并添加到当前线程的 RunLoop 中。所以如果当前线程没有 RunLoop，则这个方法会失效。

当调用 performSelector:onThread: 时，实际上其会创建一个 Timer 加到对应的线程去，同样的，如果对应线程没有 RunLoop 该方法也会失效。

1. PerformSelector:afterDelay:这个方法在子线程中是否起作用？为什么？怎么解决？

不起作用，子线程默认没有 Runloop，也就没有 Timer。 解决的办法是可以使用 GCD 来实现：Dispatch\_after

4、RunLoop的Mode？

关于Mode首先要知道一个RunLoop 对象中可能包含多个Mode，且每次调用 RunLoop 的主函数时，只能指定其中一个 Mode(CurrentMode)。切换 Mode，需要重新指定一个 Mode 。主要是为了分隔开不同的 Source、Timer、Observer，让它们之间互不影响。

当RunLoop运行在Mode1上时，是无法接受处理Mode2或Mode3上的Source、Timer、Observer事件的

总共是有五种CFRunLoopMode:

kCFRunLoopDefaultMode：默认模式，主线程是在这个运行模式下运行

UITrackingRunLoopMode：跟踪用户交互事件（用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他Mode影响）

UIInitializationRunLoopMode：在刚启动App时第进入的第一个 Mode，启动完成后就不再使用

GSEventReceiveRunLoopMode：接受系统内部事件，通常用不到

kCFRunLoopCommonModes：伪模式，不是一种真正的运行模式，是同步Source/Timer/Observer到多个Mode中的一种解决方案

5、RunLoop的实现机制

对于RunLoop而言最核心的事情就是保证线程在没有消息的时候休眠，在有消息时唤醒，以提高程序性能。RunLoop这个机制是依靠系统内核来完成的（苹果操作系统核心组件Darwin中的Mach）。

RunLoop通过mach\_msg()函数接收、发送消息。它的本质是调用函数mach\_msg\_trap()，

到内核态；内核态中内核实现的mach\_msg()函数会完成实际的工作。

即基于port的source1，监听端口，端口有消息就会触发回调；而source0，要手动标记为待处理和手动唤醒RunLoop

[Mach消息发送机制](https://www.jianshu.com/p/a764aad31847)

大致逻辑为：

1、通知观察者 RunLoop 即将启动。

2、通知观察者即将要处理Timer事件。

3、通知观察者即将要处理source0事件。

4、处理source0事件。

5、如果基于端口的源(Source1)准备好并处于等待状态，进入步骤9。

6、通知观察者线程即将进入休眠状态。

7、将线程置于休眠状态，由用户态切换到内核态，直到下面的任一事件发生才唤醒线程。

一个基于 port 的Source1 的事件(图里应该是source0)。

一个 Timer 到时间了。

RunLoop 自身的超时时间到了。

被其他调用者手动唤醒。

8、通知观察者线程将被唤醒。

9、处理唤醒时收到的事件。

如果用户定义的定时器启动，处理定时器事件并重启RunLoop。进入步骤2。

如果输入源启动，传递相应的消息。

如果RunLoop被显示唤醒而且时间还没超时，重启RunLoop。进入步骤2

1. 通知观察者RunLoop结束
2. autoreleasePool 在何时被释放？

App启动后，苹果在主线程 RunLoop 里注册了两个 Observer，其回调都是 \_wrapRunLoopWithAutoreleasePoolHandler()。

第一个 Observer 监视的事件是 Entry(即将进入Loop)，其回调内会调用 \_objc\_autoreleasePoolPush() 创建自动释放池。其 order 是 -2147483647，优先级最高，保证创建释放池发生在其他所有回调之前。

第二个 Observer 监视了两个事件： BeforeWaiting(准备进入休眠) 时调用\_objc\_autoreleasePoolPop() 和 \_objc\_autoreleasePoolPush() 释放旧的池并创建新池；Exit(即将退出Loop) 时调用 \_objc\_autoreleasePoolPop() 来释放自动释放池。这个 Observer 的 order 是 2147483647，优先级最低，保证其释放池子发生在其他所有回调之后。

在主线程执行的代码，通常是写在诸如事件回调、Timer回调内的。这些回调会被 RunLoop 创建好的 AutoreleasePool 环绕着，所以不会出现内存泄漏，开发者也不必显示创建 Pool 了。

1. 什么是异步绘制？

异步绘制，就是可以在子线程中提前把需要绘制的图形处理好。将准备好的图像数据直接返给主线程使用，这样可以降低主线程的压力。

异步绘制的过程

要通过系统的 [view.delegate displayLayer:] 这个入口来实现异步绘制。

A 代理负责生成对应的 Bitmap

B 设置该 Bitmap 为 layer.contents 属性的值。

1. 利用 runloop 解释一下页面的渲染的过程？

当我们调用 [UIView setNeedsDisplay] 时，这时会调用当前 View.layer 的 [view.layer setNeedsDisplay]方法。

这等于给当前的 layer 打上了一个脏标记，而此时并没有直接进行绘制工作。而是会到当前的 Runloop 即将休眠，也就是 beforeWaiting 时才会进行绘制工作。

紧接着会调用 [CALayer display]，进入到真正绘制的工作。CALayer 层会判断自己的 delegate 有没有实现异步绘制的代理方法 displayLayer:，这个代理方法是异步绘制的入口，如果没有实现这个方法，那么会继续进行系统绘制的流程，然后绘制结束。

CALayer 内部会创建一个 Backing Store，用来获取图形上下文。接下来会判断这个 layer 是否有 delegate。

如果有的话，会调用 [layer.delegate drawLayer:inContext:]，并且会返回给我们 [UIView DrawRect:] 的回调，让我们在系统绘制的基础之上再做一些事情。

如果没有 delegate，那么会调用 [CALayer drawInContext:]。

以上两个分支，最终 CALayer 都会将位图提交到 Backing Store，最后提交给 GPU。

至此绘制的过程结束。

## 三、KVC

1. KVC(Key-value coding)

KVC就是指iOS的开发中，可以允许开发者通过Key名直接访问对象的属性，或者给对象的属性赋值。而不需要调用明确的存取方法。这样就可以在运行时动态地访问和修改对象的属性。而不是在编译时确定，这也是iOS开发中的黑魔法之一。很多高级的iOS开发技巧都是基于KVC实现的

当调用setValue：属性值forKey：@”name“的代码时，，底层的执行机制如下：

A、程序优先调用set<Key>:属性值方法，代码通过setter方法完成设置。注意，这里的<key>是指成员变量名，首字母大小写要符合KVC的命名规则，下同

B、如果没有找到setName：方法，KVC机制会检查+ (BOOL)accessInstanceVariablesDirectly方法有没有返回YES，默认该方法会返回YES，如果你重写了该方法让其返回NO的话，那么在这一步KVC会执行setValue：forUndefinedKey：方法，不过一般开发者不会这么做。所以KVC机制会搜索该类里面有没有名为<key>的成员变量。无论该变量是在类接口处定义，还是在类实现处定义，也无论用了什么样的访问修饰符，只在存在以<key>命名的变量，KVC都可以对该成员变量赋值。

C、如果该类即没有set<key>：方法，也没有\_<key>成员变量，KVC机制会搜索\_is<Key>的成员变量。

D、和上面一样，如果该类即没有set<Key>：方法，也没有\_<key>和\_is<Key>成员变量，KVC机制再会继续搜索<key>和is<Key>的成员变量。再给它们赋值。

如果上面列出的方法或者成员变量都不存在，系统将会执行该对象的setValue：forUndefinedKey：方法，默认是抛出异常。

即如果没有找到Set<Key>方法的话，会按照\_key，\_iskey，key，iskey的顺序搜索成员并进行赋值操作。

如果开发者想让这个类禁用KVC，那么重写+ (BOOL)accessInstanceVariablesDirectly方法让其返回NO即可，这样的话如果KVC没有找到set<Key>:属性名时，会直接用setValue：forUndefinedKey：方法。

当调用valueForKey：@”name“的代码时，KVC对key的搜索方式不同于setValue：属性值 forKey：@”name“，其搜索方式如下：

A、首先按get<Key>,<key>,is<Key>的顺序方法查找getter方法，找到的话会直接调用。如果是BOOL或者Int等值类型， 会将其包装成一个NSNumber对象。

B、如果上面的getter没有找到，KVC则会查找countOf<Key>,objectIn<Key>AtIndex或<Key>AtIndexes格式的方法。如果countOf<Key>方法和另外两个方法中的一个被找到，那么就会返回一个可以响应NSArray所有方法的代理集合(它是NSKeyValueArray，是NSArray的子类)，调用这个代理集合的方法，或者说给这个代理集合发送属于NSArray的方法，就会以countOf<Key>,objectIn<Key>AtIndex或<Key>AtIndexes这几个方法组合的形式调用。还有一个可选的get<Key>:range:方法。所以你想重新定义KVC的一些功能，你可以添加这些方法，需要注意的是你的方法名要符合KVC的标准命名方法，包括方法签名。

C、如果上面的方法没有找到，那么会同时查找countOf<Key>，enumeratorOf<Key>,memberOf<Key>格式的方法。如果这三个方法都找到，那么就返回一个可以响应NSSet所的方法的代理集合，和上面一样，给这个代理集合发NSSet的消息，就会以countOf<Key>，enumeratorOf<Key>,memberOf<Key>组合的形式调用。

D、如果还没有找到，再检查类方法+(BOOL)accessInstanceVariablesDirectly,如果返回YES(默认行为)，那么和先前的设值一样，会按\_<key>,\_is<Key>,<key>,is<Key>的顺序搜索成员变量名，这里不推荐这么做，因为这样直接访问实例变量破坏了封装性，使代码更脆弱。如果重写了类方法+ (BOOL)accessInstanceVariablesDirectly返回NO的话，那么会直接调用valueForUndefinedKey:方法，默认是抛出异常。

2、KVO (Key-value observing)

KVO是观察者模式的另一实现。

使用了isa混写(isa-swizzling)来实现KVO

使用setter方法改变值KVO会生效，使用setValue:forKey即KVC改变值KVO也会生效，因为KVC会去调用setter方法

那么通过直接赋值成员变量会触发KVO吗？

不会，因为不会调用setter方法，需要加上

willChangeValueForKey和didChangeValueForKey方法来手动触发才行

1. 属性关键字

1.读写权限：readonly,readwrite(默认)

2.原子性: atomic(默认)，nonatomic。atomic读写线程安全，但效率低，而且不是绝对的安全，比如如果修饰的是数组，那么对数组的读写是安全的，但如果是操作数组进行添加移除其中对象的还，就不保证安全了。

3.引用计数：

retain/strong：它俩都是强引用，除了某些情况下不一样，其他的时候也是可以通用的。主要区别是在修饰block属性的时候，相信大家都知道要用copy吧，block在创建的时候它的内存是默认分配在栈(stack)上，而不是堆(heap)上，所以他的生命周期会随着函数的结束而结束，当你在该作用域外调用该block时, 程序就会crash，而copy之后会放在堆里面。strong在修饰block的时候就相当于copy，而retain修饰block的时候就相当于assign，这样block会出现提前被释放掉的危险。

assign：修饰基本数据类型，修饰对象类型时，不改变其引用计数，会产生悬垂指针，修饰的对象在被释放后，assign指针仍然指向原对象内存地址，如果使用assign指针继续访问原对象的话，就可能会导致内存泄漏或程序异常

weak：不改变被修饰对象的引用计数，所指对象在被释放后，weak指针会自动置为nil

copy：分为深拷贝和浅拷贝

浅拷贝：对内存地址的复制，让目标对象指针和原对象指向同一片内存空间会增加引用计数

深拷贝：对对象内容的复制，开辟新的内存空间

可变对象的copy和mutableCopy都是深拷贝

不可变对象的copy是浅拷贝，mutableCopy是深拷贝

copy方法返回的都是不可变对象

@property (nonatomic, copy) NSMutableArray \* array;这样写有什么影响？

因为copy方法返回的都是不可变对象，所以array对象实际上是不可变的，如果对其进行可变操作如添加移除对象，则会造成程序crash

## iOS UI相关面试题

一、UIView与CALayer

<单一职责原则>

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

二、事件传递与视图响应链 :

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

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

如果事件一直传递到UIAppliction还是没处理，那就会忽略掉

1. 图像显示原理

1.CPU:输出位图

2.GPU :图层渲染，纹理合成

3.把结果放到帧缓冲区(frame buffer)中

4.再由视频控制器根据vsync信号在指定时间之前去提取帧缓冲区的屏幕显示内容

5.显示到屏幕上

CPU工作

1.Layout: UI布局，文本计算

2.Display: 绘制

3.Prepare: 图片解码

4.Commit：提交位图

GPU渲染管线(OpenGL)

顶点着色，图元装配，光栅化，片段着色，片段处理

四、UI卡顿掉帧原因

iOS设备的硬件时钟会发出Vsync（垂直同步信号），然后App的CPU会去计算屏幕要显示的内容，之后将计算好的内容提交到GPU去渲染。随后，GPU将渲染结果提交到帧缓冲区，等到下一个VSync到来时将缓冲区的帧显示到屏幕上。也就是说，一帧的显示是由CPU和GPU共同决定的。

一般来说，页面滑动流畅是60fps，也就是1s有60帧更新，即每隔16.7ms就要产生一帧画面，而如果CPU和GPU加起来的处理时间超过了16.7ms，就会造成掉帧甚至卡顿。

五、滑动优化方案

CPU：

把以下操作放在子线程中

1.对象创建、调整、销毁

2.预排版（布局计算、文本计算、缓存高度等等）

3.预渲染（文本等异步绘制，图片解码等）

GPU:

纹理渲染，视图混合

一般遇到性能问题时，考虑以下问题：

是否受到CPU或者GPU的限制？

是否有不必要的CPU渲染？

是否有太多的离屏渲染操作？

是否有太多的图层混合操作？

是否有奇怪的图片格式或者尺寸？

是否涉及到昂贵的view或者效果？

view的层次结构是否合理？

1. UI绘制原理

异步绘制：

[self.layer.delegate displayLayer: ]

代理负责生成对应的bitmap

设置该bitmap作为该layer.contents属性的值

1. 离屏渲染

On-Screen Rendering:当前屏幕渲染，指的是GPU的渲染操作是在当前用于显示的屏幕缓冲区中进行

Off-Screen Rendering:离屏渲染，分为CPU离屏渲染和GPU离屏渲染两种形式。GPU离屏渲染指的是GPU在当前屏幕缓冲区外新开辟一个缓冲区进行渲染操作

应当尽量避免的则是GPU离屏渲染

GPU离屏渲染何时会触发呢？

圆角（当和maskToBounds一起使用时）、图层蒙版、阴影，设置

layer.shouldRasterize ＝ YES

为什么要避免GPU离屏渲染？

GPU需要做额外的渲染操作。通常GPU在做渲染的时候是很快的，但是涉及到offscreen-render的时候情况就可能有些不同，因为需要额外开辟一个新的缓冲区进行渲染，然后绘制到当前屏幕的过程需要做onscreen跟offscreen上下文之间的切换，这个过程的消耗会比较昂贵，涉及到OpenGL的pipeline跟barrier，而且offscreen-render在每一帧都会涉及到，因此处理不当肯定会对性能产生一定的影响。另外由于离屏渲染会增加GPU的工作量，可能会导致CPU+GPU的处理时间超出16.7ms，导致掉帧卡顿。所以可以的话应尽量减少offscreen-render的图层

**五、多线程**

1. 进程：

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

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

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

二、 线程

1.程序执行流的最小单元，线程是进程中的一个实体.

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

三、 进程和线程的关系

1.线程是进程的执行单元，进程的所有任务都在线程中执行

2.线程是 CPU 分配资源和调度的最小单位

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

4.同一个进程内的线程共享进程资源

四、多进程

打开mac的活动监视器，可以看到很多个进程同时运行

进程是程序在计算机上的一次执行活动。当你运行一个程序，你就启动了一个进程。显然，程序是死的(静态的)，进程是活的(动态的)。

进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程，它们就是处于运行状态下的操作系统本身;所有由用户启动的进程都是用户进程。进程是操作系统进行资源分配的单位。

进程又被细化为线程，也就是一个进程下有多个能独立运行的更小的单位。在同一个时间里，同一个计算机系统中如果允许两个或两个以上的进程处于运行状态，这便是多进程。

1. 多线程

1.同一时间，CPU只能处理1条线程，只有1条线程在执行。多线程并发执行，其实是CPU快速地在多条线程之间调度（切换）。如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象

2.如果线程非常非常多，CPU会在N多线程之间调度，消耗大量的CPU资源，每条线程被调度执行的频次会降低（线程的执行效率降低）

3.多线程的优点:

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

4.多线程的缺点:

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

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

1. NSOperationQueue的优点

NSOperationQueue的优点：

NSOperation、NSOperationQueue 是苹果提供给我们的一套多线程解决方案。实际上 NSOperation、NSOperationQueue 是基于 GCD 更高一层的封装，完全面向对象。但是比 GCD 更简单易用、代码可读性也更高。

1、可以添加任务依赖，方便控制执行顺序

2、可以设定操作执行的优先级

3、任务执行状态控制:isReady,isExecuting,isFinished,isCancelled

如果只是重写NSOperation的main方法，由底层控制变更任务执行及完成状态，以及任务退出

如果重写了NSOperation的start方法，自行控制任务状态

系统通过KVO的方式移除isFinished==YES的NSOperation

1. 可以设置最大并发量

七、自旋锁与互斥锁

自旋锁：

是一种用于保护多线程共享资源的锁，与一般互斥锁（mutex）不同之处在于当自旋锁尝试获取锁时以忙等待（busy waiting）的形式不断地循环检查锁是否可用。当上一个线程的任务没有执行完毕的时候（被锁住），那么下一个线程会一直等待（不会睡眠），当上一个线程的任务执行完毕，下一个线程会立即执行。

在多CPU的环境中，对持有锁较短的程序来说，使用自旋锁代替一般的互斥锁往往能够提高程序的性能。

互斥锁：

当上一个线程的任务没有执行完毕的时候（被锁住），那么下一个线程会进入睡眠状态等待任务执行完毕，当上一个线程的任务执行完毕，下一个线程会自动唤醒然后执行任务。

总结：

自旋锁会忙等: 所谓忙等，即在访问被锁资源时，调用者线程不会休眠，而是不停循环在那里，直到被锁资源释放锁。

　　互斥锁会休眠: 所谓休眠，即在访问被锁资源时，调用者线程会休眠，此时cpu可以调度其他线程工作。直到被锁资源释放锁。此时会唤醒休眠线程。

优缺点：

自旋锁的优点在于，因为自旋锁不会引起调用者睡眠，所以不会进行线程调度，CPU时间片轮转等耗时操作。所有如果能在很短的时间内获得锁，自旋锁的效率远高于互斥锁。

缺点在于，自旋锁一直占用CPU，他在未获得锁的情况下，一直运行－－自旋，所以占用着CPU，如果不能在很短的时 间内获得锁，这无疑会使CPU效率降低。自旋锁不能实现递归调用。

自旋锁：atomic、OSSpinLock、dispatch\_semaphore\_t

互斥锁：pthread\_mutex、@ synchronized、NSLock、NSConditionLock 、NSCondition、NSRecursiveLock

1. **内存管理相关**

**说一下对 retain,copy,assign,weak,\_Unsafe\_Unretain 关键字的理解。**

Strong

Strong 修饰符表示指向并持有该对象，其修饰对象的引用计数会加1。该对象只要引用计数不为0就不会被销毁。当然可以通过将变量强制赋值 nil 来进行销毁。

Weak

weak 修饰符指向但是并不持有该对象，引用计数也不会加1。在 Runtime 中对该属性进行了相关操作，无需处理，可以自动销毁。weak用来修饰对象，多用于避免循环引用的地方。weak 不可以修饰基本数据类型。

assign

assign主要用于修饰基本数据类型，

例如NSInteger，CGFloat，存储在栈中，内存不用程序员管理。assign是可以修饰对象的，但是会出现问题。

copy

copy关键字和 strong类似，copy 多用于修饰有可变类型的不可变对象上 NSString,NSArray,NSDictionary上。

\_\_unsafe\_unretain

\_\_unsafe\_unretain 类似于 weak ，但是当对象被释放后，指针已然保存着之前的地址，被释放后的地址变为 僵尸对象，访问被释放的地址就会出问题，所以说他是不安全的。

\_\_autoreleasing

将对象赋值给附有 \_\_autoreleasing修饰的变量等同于 ARC 无效时调用对象的 autorelease 方法,实质就是扔进了自动释放池。

**使用自动引用计数应遵循的原则**

不能使用 retain、release、retainCount、autorelease。

不可以使用 NSAllocateObject、NSDeallocateObject。

必须遵守内存管理方法的命名规则。

不需要显示的调用 Dealloc。

使用 @autoreleasePool来代替 NSAutoreleasePool。

不可以使用区域 NSZone。

对象性变量不可以作为 C语言的结构体成员。

显示转换 id和 void\*。

**能不能简述一下 Dealloc 的实现机制**

Dealloc 的实现机制是内容管理部分的重点，把这个知识点弄明白，对于全方位的理解内存管理的只是很有必要。

1.Dealloc 调用流程

1.首先调用 \_objc\_rootDealloc()

2.接下来调用 rootDealloc()

3.这时候会判断是否可以被释放，判断的依据主要有5个，判断是否有以上五种情况

NONPointer\_ISA

weakly\_reference

has\_assoc

has\_cxx\_dtor

has\_sidetable\_rc

4-1.如果有以上五中任意一种，将会调用 object\_dispose()方法，做下一步的处理。

4-2.如果没有之前五种情况的任意一种，则可以执行释放操作，C函数的 free()。

5.执行完毕。

object\_dispose() 调用流程。

1.直接调用 objc\_destructInstance()。

2.之后调用 C函数的 free()。

3.objc\_destructInstance() 调用流程

1.先判断 hasCxxDtor，如果有 C++ 的相关内容，要调用 object\_cxxDestruct() ，销毁 C++ 相关的内容。

2.再判断 hasAssocitatedObjects，如果有的话，要调用 object\_remove\_associations()，销毁关联对象的一系列操作。

3.然后调用 clearDeallocating()。

4.执行完毕。

4.clearDeallocating() 调用流程。

1.先执行 sideTable\_clearDellocating()。

2.再执行 weak\_clear\_no\_lock,在这一步骤中，会将指向该对象的弱引用指针置为 nil。

3.接下来执行 table.refcnts.eraser()，从引用计数表中擦除该对象的引用计数。

4.至此为止，Dealloc 的执行流程结束。

**编程中的六大设计原则？**

1.单一职责原则

通俗地讲就是一个类只做一件事

CALayer：动画和视图的显示。

UIView：只负责事件传递、事件响应。

2.开闭原则

对修改关闭，对扩展开放。

要考虑到后续的扩展性，而不是在原有的基础上来回修改

3.接口隔离原则

使用多个专门的协议、而不是一个庞大臃肿的协议

UITableviewDelegate

UITableViewDataSource

4.依赖倒置原则

抽象不应该依赖于具体实现、具体实现可以依赖于抽象。

调用接口感觉不到内部是如何操作的

5.里氏替换原则

父类可以被子类无缝替换，且原有的功能不受任何影响

例如 KVO

6.迪米特法则

一个对象应当对其他对象尽可能少的连接，实现高聚合、低耦合