BAT 内部面试资料（绝密）

目录

14.swift 中 struct 和 class 的区别

15.在一个 HTTPS 连接的网站里，输入账号密码点击登录后，到服务器返回这个请求前，中间经历了什么

16.什么时候用 delegate，什么时候用 Notification?

17.什么是 KVO 和 KVC?

18.KVC 的底层实现？

19.KVO 的底层实现？

20.ViewController 生命周期

21.方法和选择器有何不同？

22.你是否接触过 OC 中的反射机制？简单聊一下概念和使用

23.调用方法有两种方式：

24.如何对 iOS 设备进行性能测试？

25.开发项目时你是怎么检查内存泄露？

26.什么是懒加载？

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

28.什么是谓词？

29.isa 指针问题

30.如何访问并修改一个类的私有属性？

31.一个 objc 对象的 isa 的指针指向什么？有什么作用？

32.下面的代码输出什么？

33.写一个完整的代理，包括声明、实现

34.isKindOfClass、isMemberOfClass、selector 作用分别是什么

35.delegate 和 notification 的区别

36.block 反向传值

37.block 的注意点

38.BAD_ACCESS 在什么情况下出现？

39.lldb（gdb）常用的控制台调试命令？

40.你一般是怎么用 Instruments 的？

41.iOS 中常用的数据存储方式有哪些？

42.iOS 的沙盒目录结构是怎样的？

43.iOS 多线程技术有哪几种方式？

44.GCD 与 NSOperation 的区别：

45.写出使用 GCD 方式从子线程回到主线程的方法代码

46.调用代码使 APP 进入后台，达到点击 Home 键的效果。（私有 API）

47.获取 UIWebView 的高度

48.设置 UILable 的行间距 和 计算带行间距的高度

49.禁止程序运行时自动锁屏

50.CocoaPods pod install/pod update 更新慢的问题

51.修改 textFieldplaceholder 字体颜色和大小

52.禁止 textField 和 textView 的复制粘贴菜单

53.三级页面隐藏系统 tabbar 1、单个处理

54.取消系统的返回手势

55.百度坐标跟火星坐标相互转换

56.添加 pch 文件的步聚

57.关于 Masonry

58.UIWebView 在 IOS9 下底部出现黑边解决方式

59.数组逆序遍历

60.把时间字符串 2015-04-10 格式化日期转为 NSDate 类型

61.远程推送原理是什么??

62.http 和 scoket 通信的区别?

63.import、include 和@class 有什么区别

64.举出 5 个以上你所熟悉的 ios sdk 库有哪些和第三方库有哪些?

65.ViewController 的 loadView, viewDidLoad,viewWillAppear,viewDidUnload,dealloc、init 分别是在什么时 66.候调用的?在自定义 ViewController 的时候这几个函数里面应该做什么工作?

66.简述你对 UIView、UIWindow 和 CALayer 的理解

67.为什么很多内置类如 UITableViewController 的 delegate 属性都是 assign 而不是 retain 的?

68.简述 NotificationCenter、KVC、KVO、Delegate?并说明它们之间的区别?

69.线程与进程的区别和联系?

70.简述多线程的作用以及什么地方会用到多线程?OC 实现多线程的方法有哪些?谈谈多线程安全问题的几种解决方案?何为线程同步,如何实现的?分线程回调主线程方法是什么,有什么作用?

71.Objective-C 如何对内存管理的,说说你的看法和解决方法?

72.内存管理的几条原则时什么？按照默认法则.哪些方法生成的对象需要手动释放？在和 property 结合的时候怎样有效的避免内存泄露？

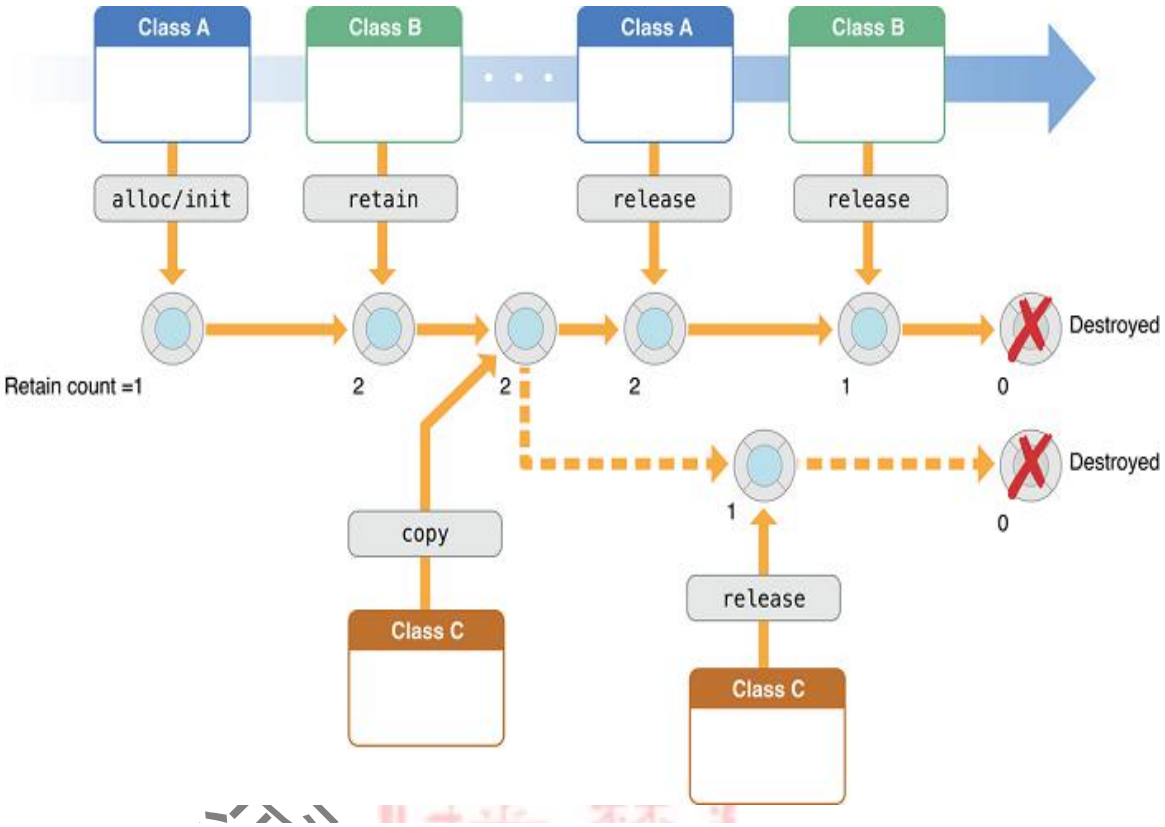
73.What is Singleton?（单例是什么）

74.对象是什么时候被释放的？

75.什么情况下会发生内存泄漏和内存溢出？

一. 讲一下你对 ios 内存管理的理解

在 Objective-C 的内存管理中，其实就是引用计数(reference count)的管理。内存管理就是在程序需要时程序员分配一段内存空间，而当使用完之后将它释放。如果程序员对内存资源使用不当，有时不仅会造成内存资源浪费，甚至会导致程序 crach。



1. 引用计数(Reference Count)

为了解释引用计数，我们做一个类比：员工在办公室使用灯的情景。

当第一个人进入办公室时，他需要使用灯，于是开灯，引用计数为 1

当另一个人进入办公室时，他也需要灯，引用计数为 2；每当多一个人进入办公室时，引用计数加 1

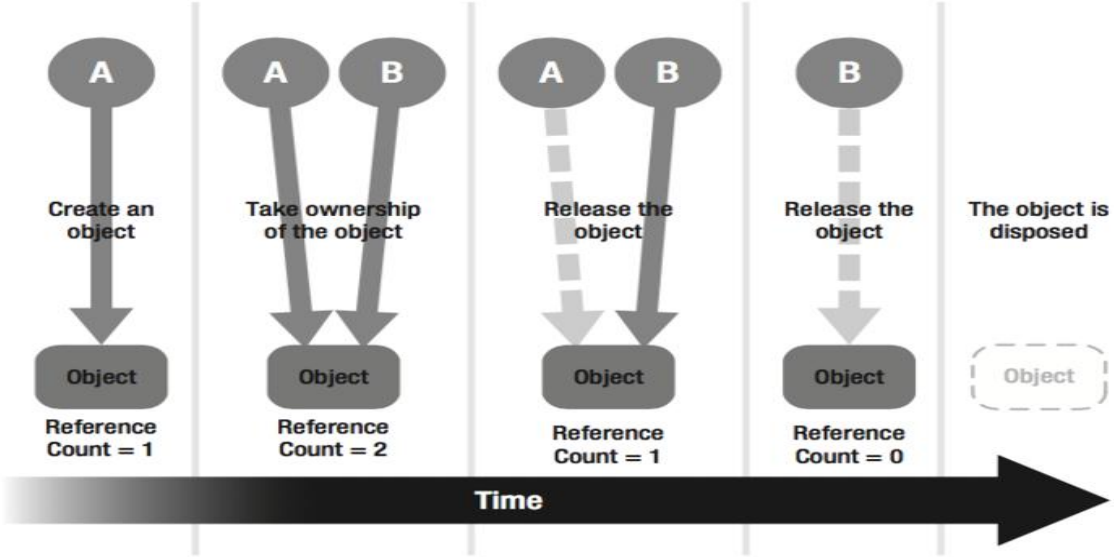
当有一个人离开办公室时，引用计数减 1，当引用计数为 0 时，也就是最后一个人离开办公室时，他不再需要使用灯，关灯离开办公室。

2. 内存管理规则

从上面员工在办公室使用灯的例子，我们对比一下灯的动作与 Objective-C 对象的动作有什么相似之处：

灯的动作	Objective-C对象的动作
开灯	创建一个对象并获取它的所有权(ownership)
使用灯	获取对象的所有权
不使用灯	放弃对象的所有权
关灯	释放对象

因为我们是通过引用计数来管理灯，那么我们也可以通过引用计数来管理使用 Objective-C 对象。



而 Objective-C 对象的动作对应有哪些方法以及这些方法对引用计数有什么影响？

Objective-C对象的动作	Objective-C对象的方法
1. 创建一个对象并获取它的所有权	alloc/new/copy/mutableCopy (RC = 1)
2. 获取对象的所有权	retain (RC + 1)
3. 放弃对象的所有权	release (RC - 1)
4. 释放对象	dealloc (RC = 0，此时会调用该方法)

当你 alloc 一个对象 objc，此时 RC=1；在某个地方你又 retain 这个对象 objc，此时 RC 加 1，也就是 RC=2；由于调用 alloc/retain 一次，对应需要调用 release 一次来释放对象 objc，所以你需要 release 对象 objc 两次，此时 RC=0；而当 RC=0 时，系统会自动调用 dealloc 方法释放对象。

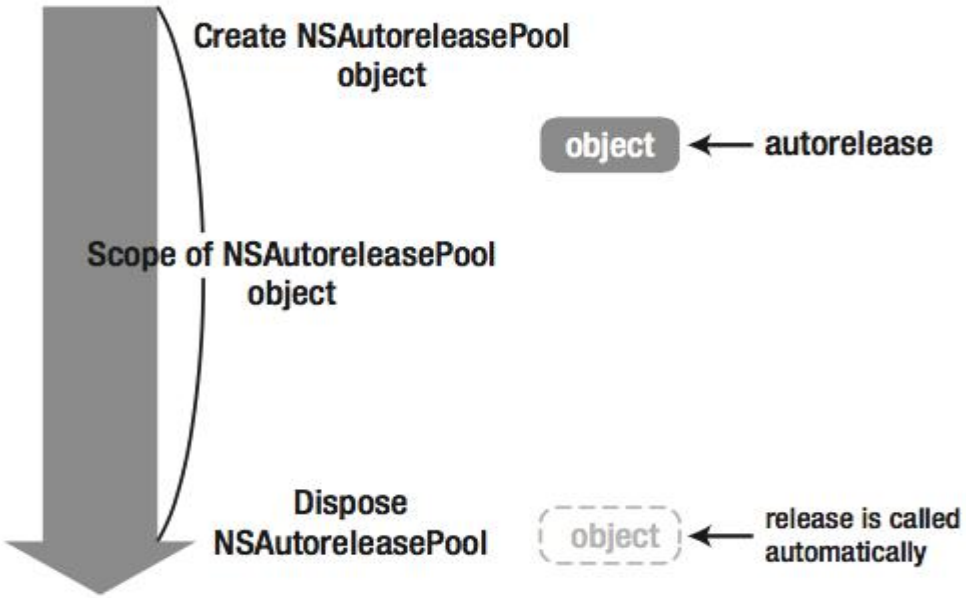
3. Autorelease Pool

在开发中，我们常常都会使用到局部变量，局部变量一个特点就是当它超过作用域时，就会自动释放。而 autorelease pool 跟局部变量类似，当执行代码超过 autorelease pool 块时，所有放在 autorelease pool 的对象都会自动调用 release。它的工作原理如下：

创建一个 NSAutoreleasePool 对象

在 autorelease pool 块的对象调用 autorelease 方法

释放 NSAutoreleasePool 对象



4. ARC 管理方法

iOS/OS X 内存管理方法有两种：手动引用计数(Manual Reference Counting)和自动引用计数(Automatic Reference Counting)。

自动引用计数(Automatic Reference Counting)简单来说，它让编译器来代替程序员来自动加入 retain 和 release 方法来持有和放弃对象的所有权。

在 ARC 内存管理机制中，id 和其他对象类型变量必须是以下四个 ownership qualifiers 其中一个来修饰：
所以在管理 Objective-C 对象内存的时候，你必须选择其中一个，下面会用一些列子来逐个解释它们的含义以及如何选择它们。

```
strong:被它修饰的变量持有对象的所有权(默认，如果不指定其他，编译器就默认加入)

weak: 被它修饰的变量都不持有对象的所有权，而且当变量指向的对象的 RC 为 0 时，变量设置为 nil。

unsafe_unretained:被它修饰的变量都不持有对象的所有权，但当变量指向的对象的 RC 为 0 时，变量并不设置为 nil，而是继续保存对象的地址；这样的话，对象有可能已经释放，但继续访问，就会造成非法访问(Invalid Access)。

autoreleasing:相比之前的创建、使用和释放 NSAutoreleasePool 对象，现在你只需要将代码放在 @autoreleasepool 块即可。你也不需要调用 autorelease 方法了，只需要用__autoreleasing 修饰变量即可。
```

5.Property(属性)

Property modifier	Ownership qualifier
strong	__strong
retain	__strong
copy	__strong
weak	__weak
assign	__unsafe_unretained
unsafe_unretained	__unsafe_unretained

二. KVO 实现原理

KVO 基本原理：

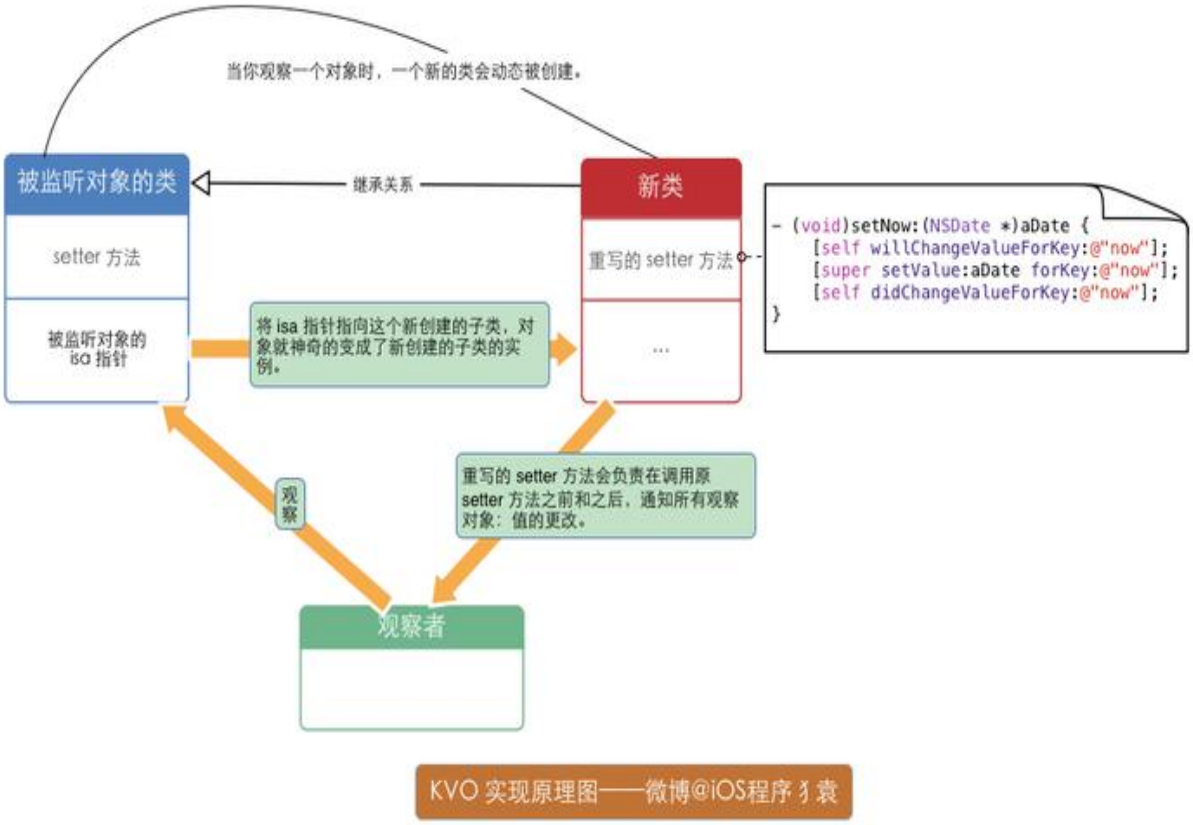
1.KVO 是基于 runtime 机制实现的

2.当某个类的属性对象第一次被观察时，系统就会在运行期动态地创建该类的一个派生类，在这个派生类中重写基类中任何被观察属性的 setter 方法。派生类在被重写的 setter 方法内实现真正的通知机制

3.如果原类为 Person，那么生成的派生类名为 NSKVONotifying_Person

4.每个类对象中都有一个 isa 指针指向当前类，当一个类对象的第一次被观察，那么系统会偷偷将 isa 指针指向动态生成的派生类，从而在给被监控属性赋值时执行的是派生类的 setter 方法

5.键值观察通知依赖于 NSObject 的两个方法: willChangeValueForKey: 和 didChangevlueForKey:; 在一个被观察属性发生改变之前，willChangeValueForKey:一定会被调用，这就 会记录旧的值。而当改变发生后，didChangeValueForKey:会被调用，继而 observeValueForKey:ofObject:change:context: 也会被调用。

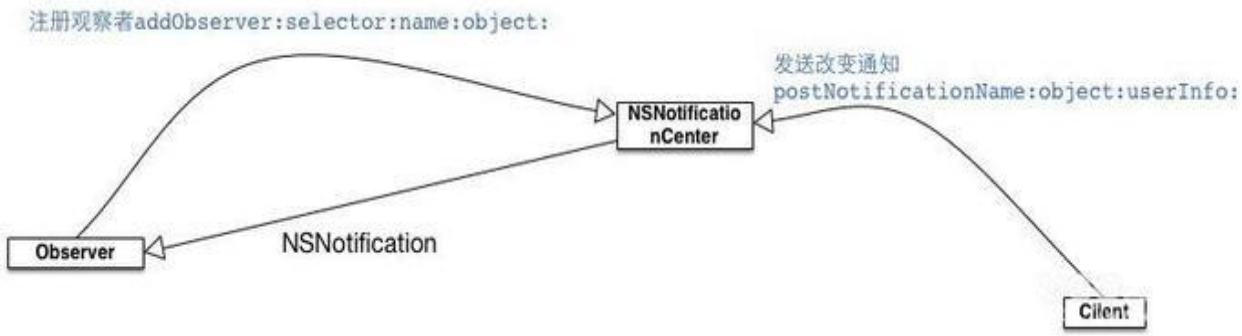


三. 观察者模式

观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。

在 iOS 中典型的观察者模式是:NSNotificationCenter 和 KVO。

1. NSNotificationCenter



观察者 Observer，通过 NSNotificationCenter 的 addObserver:selector:name:object 接口来注册对某一类型通知感兴趣。在注册时候一定要注意，NSNotificationCenter 会对观察者进行引用计数+1 的操作，我们在程序中释放观察者的时候，一定要去从 center 中将其移除。

通知中心 NSNotificationCenter，通知的枢纽。

被观察的对象，通过 postNotificationName:object:userInfo:发送某一类型通知，广播改变。

通知对象 NSNotification，当有通知来的时候，Center 会调用观察者注册的接口来广播通知，同时传递存储着更改内容的 NSNotification 对象。

2. KVO

KVO 的全称是 Key-Value Observer，即键值观察。是一种没有中心枢纽的观察者模式的实现方式。一个主题对象管理所有依赖于它的观察者对象，并且在自身状态发生改变的时候主动通知观察者对象。

注册观察者

```
[object addObserver:self forKeyPath:property options:NSKeyValueObservingOptionNew
context:]。
```

更改主题对象属性的值，即触发发送更改的通知。

在制定的回调函数中，处理收到的更改通知。

注销观察者[object removeObserver:self forKeyPath:property]。

四. 如果让你实现 NSNotificationCenter，讲一下思路

NSNotificationCenter 是一个单例

NSNotificationCenter 内部使用可变字典 NSMutableDictionary 来存储，以通知名称 postName 作为 key，以数组 NSAray 作为值，该数组存储着每个观察者的信息：观察者对象、观察者处理方法、通知名称等。

当发送通知时，以通知名称为 key 去获取相应的观察者信息数组，然后遍历这个数组，取出观察者对象和相对应处理方法，进行实例方法调用。

五. 如果让你实现 GCD 的线程池，讲一下思路

线程池包含如下 8 个部分:

线程池管理器（ThreadPoolManager）:用于创建并管理线程池，是一个单例

工作线程（WorkThread）: 线程池中线程

任务接口（Task）:每个任务必须实现的接口，以供工作线程调度任务的执行。

任务队列:用于存放没有处理的任务。提供一种缓冲机制。

corePoolSize 核心池的大小:默认情况下，在创建了线程池后，线程池中的线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到 corePoolSize 后，就会把到达的任务放到缓存队列当中；

maximumPoolSize 线程池最大线程数:它表示在线程池中最多能创建多少个线程；

存活时间 keepAliveTime:表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于 corePoolSize 时，keepAliveTime 才会起作用，这是如果一个线程空闲的时间达到 keepAliveTime，则会终止直到线程池中的线程数不大于 corePoolSize.

具体流程:

当通过任务接口向线程池管理器中添加任务时，如果当前线程池管理器中的线程数目小于 corePoolSize，则每来一个任务，就会通过线程池管理器创建一个线程去执行这个任务；

如果当前线程池中的线程数目大于等于 corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

如果当前线程池中的线程数目达到 maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize 时，如果某线程空闲时间超过 keepAliveTime，线程将被终止，直至线程池中的线程数目不大于 corePoolSize；

六. Category 的实现原理，以及 Category 为什么只能加方法不能加实例变量。

```
// 从 `class_data_bits_t `调用 `data` 方法，将结果从 `class_rw_t `强制转换为 `class_ro_t `指针const class_ro_t *ro = (const class_ro_t *)cls->data();

// 初始化一个 `class_rw_t` 结构体class_rw_t *rw = (class_rw_t *)calloc(sizeof(class_rw_t), 1);

// 设置`结构体 ro` 的值以及 `flag`

rw->ro = ro;

// 最后设置正确的` data`。rw->flags = RW_REALIZED|RW_REALIZING;

cls->setData(rw);
```

运行时加载的时候 class_ro_t 里面的方法、协议、属性等内容赋值给 class_rw_t，而 class_rw_t 里面没有用来存储相关变量的数组，这样的结构也就注定实例变量是无法在运行期进行填充.

七. swift 中 struct 和 class 的区别

swift 中，class 是引用类型，struct 是值类型。值类型在传递和赋值时将进行复制，而引用类型则只会使用引用对象的一个"指向"。所以他们两者之间的区别就是两个类型的区别。

class 有这几个功能 struct 没有的：

class 可以继承，这样子类可以使用父类的特性和方法

类型转换可以在 runtime 的时候检查和解释一个实例的类型

可以用 deinit 来释放资源

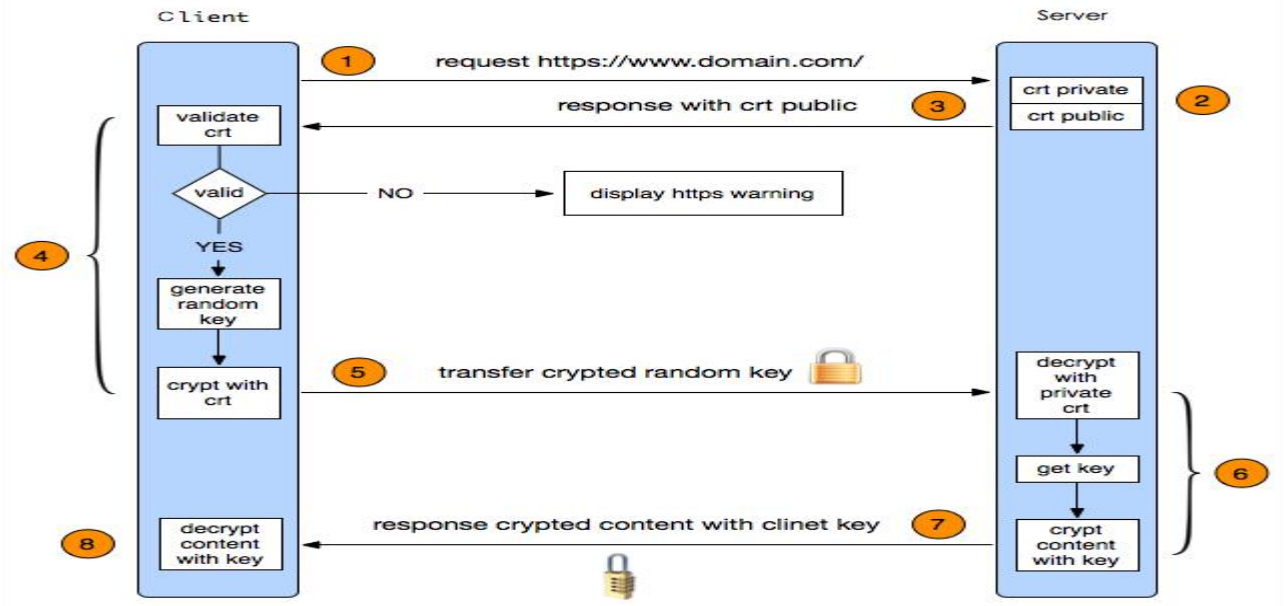
一个类可以被多次引用

struct 也有这样几个优势：

结构较小，适用于复制操作，相比于一个 class 的实例被多次引用更加安全。

无须担心内存 memory leak 或者多线程冲突问题

八.在一个 HTTPS 连接的网站里，输入账号密码点击登录后，到服务器返回这个请求前，中间经历了什么



- 客户端打包请求。包括 url，端口，你的账号密码等等。账号密码登陆应该用的是 Post 方式，所以相关的用户信息会被加载到 body 里面。这个请求应该包含三个方面：网络地址，协议，资源路径。注意，这里是 HTTPS，就是 HTTP + SSL / TLS，在 HTTP 上又加了一层处理加密信息的模块（相当于是个锁）。这个过程相当于是客户端请求钥匙。

- 服务器接受请求。一般客户端的请求会先发送到 DNS 服务器。 DNS 服务器负责将你的网络地址解析成 IP 地址，这个 IP 地址对应网上一台机器。这其中可能发生 Hosts Hijack 和 ISP failure 的问题。过了 DNS 这一关，信息就到了服务器端，此时客户端会和服务器的端口之间建立一个 socket 连接，socket 一般都是以 file descriptor 的方式解析请求。这个过程相当于是服务器端分析是否要向客户端发送钥匙模板。

- 服务器端返回数字证书。服务器端会有一套数字证书（相当于是个钥匙模板），这个证书会先发送给客户端。这个过程相当于是服务器端向客户端发送钥匙模板。

- 客户端生成加密信息。根据收到的数字证书（钥匙模板），客户端会生成钥匙，并把内容锁上，此时信息已经加密。这个过程相当于客户端生成钥匙并锁上请求。

- 客户端发送加密信息。服务器端会收到由自己发送出去的数字证书加锁的信息。这个时候生成的钥匙也一并被发送到服务器端。这个过程是相当于客户端发送请求。

- 服务器端解锁加密信息。服务器端收到加密信息后，会根据得到的钥匙进行解密，并把要返回的数据进行对称加密。这个过程相当于服务器端`解锁请求、生成、加锁回应信息。

- 服务器端向客户端返回信息。客户端会收到相应的加密信息。这个过程相当于服务器端向客户端发送回应。

- 客户端解锁返回信息。客户端会用刚刚生成的钥匙进行解密，将内容显示在浏览器上。

九. 在一个 app 中间有一个 button，在你手触摸屏幕点击后，到这个 button 收到点击事件，中间发生了什么

响应链大概有以下几个步骤：

设备将 touch 到的 UITouch 和 UIEvent 对象打包, 放到当前活动的 Application 的事件队列中

单例的 UIApplication 会从事件队列中取出触摸事件并传递给单例 UIWindow

UIWindow 使用 hitTest:withEvent:方法查找 touch 操作的所在的视图 view

(备注:UIResponder 是 UIView 的父类，UIView 是 UIControl 的父类。)
RunLoop 这边我大概讲一下

- 主线程的 RunLoop 被唤醒
- 通知 Observer，处理 Timer 和 Source 0
- Springboard 接受 touch event 之后转给 App 进程中
- RunLoop 处理 Source 1，Source1 就会触发回调，并调用 _UIApplicationHandleEventQueue()进行应用内部的分发。
- RunLoop 处理完毕进入睡眠，此前会释放旧的 autorelease pool 并新建一个 autorelease pool

十. main()之前的过程有哪些？

1）dyld 开始将程序二进制文件初始化

2)交由 ImageLoader 读取 image，其中包含了我们的类，方法等各种符号(Class、Protocol 、Selector、 IMP)

3）由于 runtime 向 dyld 绑定了回调，当 image 加载到内存后，dyld 会通知 runtime 进行处理

4）runtime 接手后调用 map_images 做解析和处理

5）接下来 load_images 中调用 call_load_methods 方法，遍历所有加载进来的 Class，按继承层次依次调用 Class 的+load 和其他 Category 的+load 方法

6）至此 所有的信息都被加载到内存中

7）最后 dyld 调用真正的 main 函数

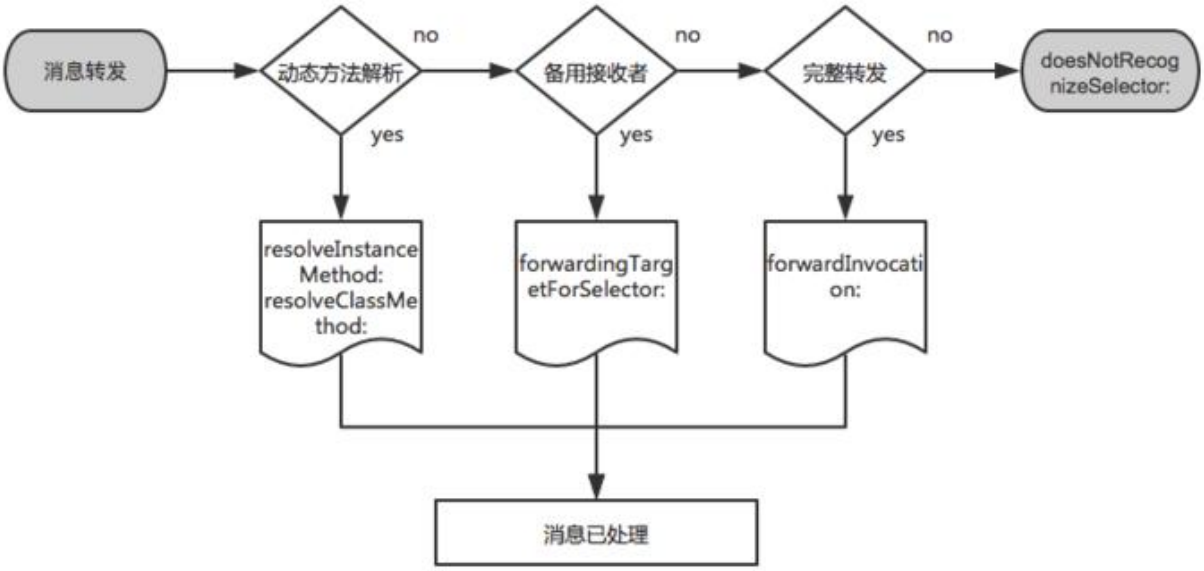
注意：dyld 会缓存上一次把信息加载内存的缓存，所以第二次比第一次启动快一点

十一．消息转发机制原理？

动态方法解析

备用接受者

完整转发



举个 ：
新建一个 `HelloClass` 的类，定义两个方法：

```
@interfaceHelloClass:NSObject

- (void)hello;

+ (HelloClass *)hi;

@end
```

1. 动态方法解析

对象在接收到未知的消息时，首先会调用所属类的类方法 `-resolveInstanceMethod:(实例方法)` 或者 `+resolveClassMethod:(类方法)`。在这个方法中，我们有机会为该未知消息新增一个"处理方法"。不过使用该方法的前提是我们已经实现了该"处理方法"，只需要在运行时通过 `class_addMethod` 函数动态添加到类里面就可以了。

```
void functionForMethod(id self, SEL _cmd)

{

NSLog(@"Hello!");

}

Class functionForClassMethod(id self, SEL _cmd)

{

NSLog(@"Hi!");

return [HelloClass class];

}

#pragma mark - 1、动态方法解析

+ (BOOL)resolveClassMethod:(SEL)sel

{
```

```
NSLog(@"resolveClassMethod");

NSString *selString = NSStringFromSelector(sel);

    if ([selString isEqualToString:@"hi"])

    {

Class metaClass = objc_getMetaClass("HelloClass");

class_addMethod(metaClass, @selector(hi), (IMP)functionForClassMethod, "v@:");

return YES;

 }

 return [super resolveClassMethod:sel];



}

+ (BOOL)resolveInstanceMethod:(SEL)sel

{

 NSLog(@"resolveInstanceMethod");

NSString *selString = NSStringFromSelector(sel);

if ([selString isEqualToString:@"hello"])

   {

class_addMethod(self, @selector(hello), (IMP)functionForMethod, "v@:");

return YES;

   }

return [super resolveInstanceMethod:sel];

}
```

2. 备用接受者

动态方法解析无法处理消息，则会走备用接受者。这个备用接受者只能是一个新的对象，不能是 self 本身，否则就会出现无限循环。如果我们没有指定相应的对象来处理 aSelector，则应该调用父类的实现来返回结果。

```
#pragma mark - 2、备用接收者

- (id)forwardingTargetForSelector:(SEL)aSelector

{

NSLog(@"forwardingTargetForSelector");

 NSString *selectorString = NSStringFromSelector(aSelector);

// 将消息交给_helper 来处理    if ([selectorString isEqualToString:@"hello"]) {



        return _helper;

}

  return [super forwardingTargetForSelector:aSelector];

}
```

在本类中需要实现这个新的接受对象

```
@interfaceHelloClass()

{

 RuntimeMethodHelper *_helper;

}

@end

@implementation HelloClass

- (instancetype)init{
```

```
self = [super init];

   if (self){

_helper = [RuntimeMethodHelper new];

  }

    return self;

    }
```

`RuntimeMethodHelper` 类需要实现这个需要转发的方法：

```
#import"RuntimeMethodHelper.h"

@implementationRuntimeMethodHelper

- (void)hello

{

 NSLog(@"%@, %p", self, _cmd);

}

@end
```

3. 完整消息转发

如果动态方法解析和备用接受者都没有处理这个消息，那么就会走完整消息转发：

```
#pragma mark - 3、完整消息转发

- (void)forwardInvocation:(NSInvocation *)anInvocation

{

 NSLog(@"forwardInvocation");
```

```
 if ([RuntimeMethodHelper instancesRespondToSelector:anInvocation.selector]) {
[anInvocation invokeWithTarget:_helper];
   }
}
/*必须重新这个方法，消息转发机制使用从这个方法中获取的信息来创建NSInvocation 对象*/
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
NSMethodSignature *signature = [super methodSignatureForSelector:aSelector];
    if (!signature)
{
       if ([RuntimeMethodHelper instancesRespondToSelector:aSelector])
   {
signature = [RuntimeMethodHelper instanceMethodSignatureForSelector:aSelector];
}
  }
 return signature;
}
```

十二. 说说你理解 weak 属性？

weak 实现原理：

Runtime 维护了一个 weak 表，用于存储指向某个对象的所有 weak 指针。weak 表其实是一个 hash（哈希）表，Key 是所指对象的地址，Value 是 weak 指针的地址（这个地址的值是所指对象的地址）数组。

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。

添加引用时：objc_initWeak 函数会调用 objc_storeWeak()函数，objc_storeWeak()的作用是更新指针指向，创建对应的弱引用表。

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。

追问的问题一：

1.实现 weak 后，为什么对象释放后会自动为 nil？

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

追问的问题二：

2.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？

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

十三. 遇到 tableView 卡顿嘛？会造成卡顿的原因大致有哪些？

可能造成 tableView 卡顿的原因有：

最常用的就是 cell 的重用， 注册重用标识符
如果不重用 cell 时，每当一个 cell 显示到屏幕上时，就会重新创建一个新的 cell
如果有很多数据的时候，就会堆积很多 cell。
如果重用 cell，为 cell 创建一个 ID，每当需要显示 cell 的时候，都会先去缓冲池中寻找可循环利用的 cell，如果没有再重新创建 cell

避免 cell 的重新布局
cell 的布局填充等操作 比较耗时，一般创建时就布局好
如可以将 cell 单独放到一个自定义类，初始化时就布局好

提前计算并缓存 cell 的属性及内容
当我们创建 cell 的数据源方法时，编译器并不是先创建 cell 再定 cell 的高度
而是先根据内容一次确定每一个 cell 的高度，高度确定后，再创建要显示的 cell，
滚动时，每当 cell 进入屏幕都会计算高度，提前估算高度告诉编译器，编译器知道高度后，紧接着就会创建 cell，这时再调用高度的具体计算方法，这样的方式不用浪费时间去计算显示以外的 cell

减少 cell 中控件的数量
尽量使 cell 得布局大致相同，不同风格的 cell 可以使用不用的重用标识符，初始化时添加控件，
不适用的可以先隐藏

不要使用 ClearColor，无背景色，透明度也不要设置为 0,因为渲染耗时比较长

使用局部更新
如果只是更新某组的话，使用 reloadSection 进行局部更新

加载网络数据，下载图片，使用异步加载，并缓存

少使用 addView 给 cell 动态添加 view

按需加载 cell，cell 滚动很快时，只加载范围内的 cell

不要实现无用的代理方法，tableView 只遵守两个协议

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

不要做多余的绘制工作。在实现 drawRect:的时候，它的 rect 参数就是需要绘制的区域，这个区域之外的不需要进行绘制。例如上例中，就可以用

CGRectIntersectsRect、CGRectIntersection 或 CGRectContainsRect 判断是否需要绘制 image 和 text，然后再调用绘制方法。

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

使用正确的数据结构来存储数据。

-

十四. UIView 和 CALayer 的区别和联系

联系:

每个 UIView 内部都有一个 CALayer 在背后提供内容的绘制和显示，并且 UIView 的尺寸样式都由内部的 CALayer 所提供。两者都有树状层级结构，layer 内部有 SubLayers，UIView 内部有 SubViews,但是 CALayer 比 UIView 多了个 AnchorPoint;

在 View 显示的时候，UIView 做为 CALayer 的 CALayerDelegate，View 的显示内容由内部的 CALayer 来 display。

CALayer 是默认修改属性支持隐式动画的，在给 UIView 的 Layer 做动画的时候，View 作为 Layer 的代理，Layer 通过 actionForLayer:forKey:向 View 请求相应的 action(动画行为)

layer 内部维护者三份 layer tree,分别是 presentLayer Tree(动画树)、modeLayer Tree(模型树)、Render Tree(渲染树)，在做 iOS 动画的时候，我们修改动画的属性，其实是 Layer 的 presentLayer 的属性值，而最终展示在界面上的其实是提供 view 的 modelLayer。

区别:

UIView 和 CALayer 最大的区别是 UIView 可以接受并处理触摸事件，而 CALayer 不可以。

十五. 什么是离屏渲染，为什么会触发离屏渲染，离屏渲染的危害

1. 什么是离屏渲染:

GPU 渲染机制:

CPU 计算好显示内容提交到 GPU，GPU 渲染完成后将渲染结果放入帧缓冲区，随后视频控制器会按照 VSync 信号逐行读取帧缓冲区的数据，经过可能的数模转换传递给显示器显示。

GPU 屏幕渲染**有以下两种方式：**

On-Screen Rendering
意为当前屏幕渲染，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
意为离屏渲染，指的是 GPU 在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作。

特殊的"离屏渲染"
如果将不在 GPU 的当前屏幕缓冲区中进行的渲染都称为离屏渲染，那么就还有另一种特殊的"离屏渲染"方式：CPU 渲染。

如果我们重写了 drawRect 方法，并且使用任何 Core Graphics 的技术进行了绘制操作，就涉及到了 CPU 渲染。整个渲染过程由 CPU 在 App 内同步地完成，渲染得到的 bitmap 最后再交由 GPU 用于显示。

2. 为什么会触发离屏渲染

设置了以下属性时，都会触发离屏绘制：

```
- shouldRasterize（光栅化）

- masks（遮罩）

- shadows（阴影）

- edge antialiasing（抗锯齿）

- group opacity（不透明）

- cornerRadius， 如果能够只用 cornerRadius 解决问题，不设置 masksToBounds，则不会引起离屏渲染，如果既设置了 cornerRadius，又设置了 masksToBounds，就会触发离屏渲染
```

因为当设置锯齿，阴影，遮罩的时候，图层属性的混合体被指定为在未预合成之前不能直接在屏幕中绘制，所以就需要屏幕外渲染被唤起。

这句话的意思是当你给一个控件设置锯齿，阴影，遮罩的时候，这时候控件需要去混合各个图层的像素来找出各个图层的正确显示效果，所以触发离屏渲染。

3. 离屏渲染的危害

相比于当前屏幕渲染，离屏渲染的代价是很高的，主要体现在两个方面：

- **创建新缓冲区**

要想进行离屏渲染，首先要创建一个新的缓冲区。

- **上下文切换**

离屏渲染的整个过程，需要多次切换上下文环境：先是从当前屏幕（On-Screen）切换到离屏（Off-Screen）；等到离屏渲染结束以后，将离屏缓冲区的渲染结果显示到屏幕上有需要将上下文环境从离屏切换到当前屏幕。而上下文环境的切换是要付出很大代价的。

16、什么时候用 delegate，什么时候用 Notification？

```
Delegate(委托模式)：1 对 1 的反向消息通知功能。

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

17、什么是 KVO 和 KVC？

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

举例说明：

    stu.name = @"张三" // 点语法给属性赋值

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

    stu1.nameLabel.text = @"张三";

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

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

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

```
// 通过下方方法为属性添加KVO观察
- (void)addObserver:(NSObject *)observer
                 forKeyPath:(NSString *)keyPath
                 options:(NSKeyValueObservingOptions)options
                 context:(nullable void *)context;
// 当被观察的属性发送变化时，会自动触发下方方法
- (void)observeValueForKeyPath:(NSString *)keyPath
                          ofObject:(id)object
                             change:(NSDictionary *)change
                          context:(void *)context{}
```

KVC 和 KVO 的 keyPath 可以是属性、实例变量、成员变量。

18、KVC 的底层实现？

当一个对象调用 setValue 方法时，方法内部会做以下操作：

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

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

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

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

这些方法的默认实现都是抛出异常，我们可以根据需要重写它们。

19、KVO 的底层实现？

KVO 基于 runtime 机制实现。

20、ViewController 生命周期

按照执行顺序排列：

1. initWithCoder：通过 nib 文件初始化时触发。

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

3. loadView：开始加载视图控制器自带的 view。

4. viewDidLoad：视图控制器的 view 被加载完成。

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

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

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

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

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

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

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

21、方法和选择器有何不同？

selector 是一个方法的名字，方法是一个组合体，包含了名字和实现。

22、你是否接触过 OC 中的反射机制？简单聊一下概念和使用

1). class 反射

通过类名的字符串形式实例化对象。

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

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

```
将类名变为字符串。

    Class class =[Student class];

    NSString *className = NSStringFromClass(class);2). SEL 的反射

通过方法的字符串形式实例化方法。

    SEL selector = NSSelectorFromString(@"setName");

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

将方法变成字符串。

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

23、调用方法有两种方式：

```
1) . 直接通过方法名来调用。[person show];

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

24、如何对 iOS 设备进行性能测试？

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

25、开发项目时你是怎么检查内存泄露？

```
1). 静态分析 analyze。

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

26、什么是懒加载？

```
答：懒加载就是只在用到的时候才去初始化。也可以理解成延时加载。
```

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

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

@public 任何地方都能访问；

@protected 该类和子类中访问，是默认的；

@private 只能在本类中访问；

@package 本包内使用，跨包不可以。

28、什么是谓词？

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

```
//定义谓词对象，谓词对象中包含了过滤条件(过滤条件比较多)
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"age<%d",30];
//使用谓词条件过滤数组中的元素，过滤之后返回查询的结果
NSArray *array = [persons filteredArrayUsingPredicate:predicate];
```

29、isa 指针问题

isa：是一个 Class 类型的指针.

每个实例对象有个 isa 的指针，他指向对象的类，而 Class 里也有个 isa 的指针，指向 meteClass(元类)。

元类保存了类方法的列表。

当类方法被调用时，先会从本身查找类方法的实现，如果没有，元类会向他父类查找该方法。

同时注意的是:元类(meteClass)也是类，它也是对象。

元类也有 isa 指针，它的 isa 指针最终指向的是一个根元类(root meteClass)。

根元类的 isa 指针指向本身，这样形成了一个封闭的内循环。

30、如何访问并修改一个类的私有属性？

```
1). 一种是通过 KVC 获取。

2). 通过 runtime 访问并修改私有属性。
```

31、一个 objc 对象的 isa 的指针指向什么？有什么作用？

```
答：指向他的类对象,从而可以找到对象上的方法。
```

32、下面的代码输出什么？

```
@implementation Son : Father

- (id)init {

   if (self = [super init]) {

       NSLog(@"%@", NSStringFromClass([self class])); // Son

       NSLog(@"%@", NSStringFromClass([super class])); // Son

   }

return self;

}@end// 解析:self 是类的隐藏参数,指向当前调用方法的这个类的实例。super 是一个 Magic Keyword，它本质是一个编译器标示符，和 self 是指向的同一个消息接收者。

不同的是：super 会告诉编译器，调用 class 这个方法时，要去父类的方法，而不是本类里的。

上面的例子不管调用[self class]还是[super class]，接受消息的对象都是当前 Son *obj 这个对象。
```

33、写一个完整的代理，包括声明、实现

```
// 创建@protocol MyDelagate@required

-(void)eat:(NSString *)foodName; @optional
```

```
-(void)run;@end//  声明 .h@interface person: NSObject<MyDelagate>@end//  实现 .m@implementation person

- (void)eat:(NSString *)foodName {

   NSLog(@"吃:%@!", foodName);

}

- (void)run {

   NSLog(@"run!");

}@end
```

34、isKindOfClass、isMemberOfClass、selector 作用分别是什么

```
isKindOfClass：作用是某个对象属于某个类型或者继承自某类型。

isMemberOfClass：某个对象确切属于某个类型。

selector：通过方法名，获取在内存中的函数的入口地址。
```

35、delegate 和 notification 的区别

```
1) . 二者都用于传递消息，不同之处主要在于一个是一对一的，另一个是一对多的。

2) . notification 通过维护一个 array，实现一对多消息的转发。

3) . delegate 需要两者之间必须建立联系，不然没法调用代理的方法；notification 不需要两者之间有联系。

45、什么是 block？

闭包（block）：闭包就是获取其它函数局部变量的匿名函数。
```

36、block 反向传值

在控制器间传值可以使用代理或者 block，使用 block 相对来说简洁。

在前一个控制器的 touchesBegan:方法内实现如下代码。

```
// OneViewController.m

  TwoViewController *twoVC = [[TwoViewController alloc] init];

  twoVC.valueBlcok = ^(NSString *str) {

    NSLog(@"OneViewController 拿到值：%@", str);

  };

  [self presentViewController:twoVC animated:YES completion:nil];

 // TwoViewController.h    （在.h 文件中声明一个 block 属性）

  @property (nonatomic ,strong) void(^valueBlcok)(NSString *str);

// TwoViewController.m    （在.m 文件中实现方法）

- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event {

    // 传值:调用 block

    if (_valueBlcok) {

        _valueBlcok(@"123456");

    }

}
```

37、block 的注意点

1). 在 block 内部使用外部指针且会造成循环引用情况下，需要用__week 修饰外部指针：

```
    __weak typeof(self) weakSelf = self;
```

2). 在 block 内部如果调用了延时函数还使用弱指针会取不到该指针，因为已经被销毁了，需要在 block 内部再将弱指针重新强引用一下。

```
    __strong typeof(self) strongSelf = weakSelf;

3). 如果需要在 block 内部改变外部栈区变量的话，需要在用__block 修饰外部变量。
```

38、BAD_ACCESS 在什么情况下出现？

```
答：这种问题在开发时经常遇到。原因是访问了野指针，比如访问已经释放对象的成员变量或者发消息、死循环等。
```

39、lldb（gdb）常用的控制台调试命令？

```
1). p 输出基本类型。是打印命令，需要指定类型。是 print 的简写

    p (int)[[[self view] subviews] count]

2). po 打印对象，会调用对象 description 方法。是 print-object 的简写

    po [self view]

3). expr 可以在调试时动态执行指定表达式，并将结果打印出来。常用于在调试过程中修改变量的值。
4). bt：打印调用堆栈，是 thread backtrace 的简写，加 all 可打印所有 thread 的堆栈

5). br l：是 breakpoint list 的简写
```

40、你一般是怎么用 Instruments 的？

```
Instruments 里面工具很多，常用：

1). Time Profiler：性能分析

2). Zombies：检查是否访问了僵尸对象，但是这个工具只能从上往下检查，不智能。

3). Allocations：用来检查内存，写算法的那批人也用这个来检查。

4). Leaks：检查内存，看是否有内存泄露。
```

41、iOS 中常用的数据存储方式有哪些？

```
数据存储有四种方案：NSUserDefault、KeyChain、file、DB。

    其中 File 有三种方式：plist、Archive（归档）

    DB 包括：SQLite、FMDB、CoreData
```

42、iOS 的沙盒目录结构是怎样的？

```
沙盒结构：

1). Application：存放程序源文件，上架前经过数字签名，上架后不可修改。

2). Documents：常用目录，iCloud 备份目录，存放数据。（这里不能存缓存文件，否则上架不被通过）

3). Library：

        Caches：存放体积大又不需要备份的数据。(常用的缓存路径)

        Preference：设置目录，iCloud 会备份设置信息。

4). tmp：存放临时文件，不会被备份，而且这个文件下的数据有可能随时被清除的可能。
```

43、iOS 多线程技术有哪几种方式？

```
答：pthread、NSThread、GCD、NSOperation
```

44、GCD 与 NSOperation 的区别：

```
GCD 和 NSOperation 都是用于实现多线程：

    GCD 基于 C 语言的底层 API，GCD 主要与 block 结合使用，代码简洁高效。

    NSOperation 属于 Objective-C 类，是基于 GCD 更高一层的封装。复杂任务一般用 NSOperation 实现。
```

45、写出使用 GCD 方式从子线程回到主线程的方法代码

```
答：dispatch_async(dispatch_get_main_queue(), ^{ });
```

46、调用代码使 APP 进入后台，达到点击 Home 键的效果。（私有 API）

```
[[UIApplication sharedApplication] performSelector:@selector(suspend)];suspend 的英文意思：悬、挂、暂停
```

47、获取 UIWebView 的高度

```
(void)webViewDidFinishLoad:(UIWebView *)webView  {      CGFloat height = [[webView stringByEvaluatingJavaScriptFromString:@``"document.body.offsetHeight"``] floatValue];      CGRect frame = webView.frame;

webView.frame = CGRectMake(frame.origin.x, frame.origin.y, frame.size.width, height);

 }
```

48、设置 UILable 的行间距 和 计算带行间距的高度

```
/*

UILabel* 展示的控件

(NSString*)str  内容

withFont:(float)font 字体大小

WithSpace:(float)space 行间距

*///给 UILabel 设置行间距和字间距

-(void)setLabelSpace:(UILabel*)label withValue:(NSString*)str withFont:(float)font WithSpace:(float)space{

NSMutableParagraphStyle *paraStyle = [[NSMutableParagraphStyle alloc] init];

paraStyle.lineBreakMode = NSLineBreakByCharWrapping;
```

```
paraStyle.alignment = NSTextAlignmentLeft;

paraStyle.lineSpacing = space; //设置行间距

paraStyle.hyphenationFactor = 1.0;

paraStyle.firstLineHeadIndent = 0.0;

paraStyle.paragraphSpacingBefore = 0.0;

paraStyle.headIndent = 0;

paraStyle.tailIndent = 0;//设置字间距 NSKernAttributeName:@1.5fUIFont *tfont = [UIFont
systemFontOfSize:font];NSDictionary *dic = @{NSFontAttributeName:tfont,
NSParagraphStyleAttributeName:paraStyle, NSKernAttributeName:@1.5f

};NSAttributedString *attributeStr = [[NSAttributedString alloc] initWithString:str
attributes:dic];

label.attributedText = attributeStr;

}/*

计算UILabel 的高度(带有行间距的情况)

(NSString*)str  内容

withFont:(float)font 字体大小

WithSpace:(float)space 行间距

(CGFloat)width  UILable 的宽度

*///计算UILabel 的高度(带有行间距的情况)

-(CGFloat)getSpaceLabelHeight:(NSString*)str withFont:(float)font
withWidth:(CGFloat)width WithSpace:(float)space{NSMutableParagraphStyle *paraStyle =
[[NSMutableParagraphStyle alloc] init];

paraStyle.lineBreakMode = NSLineBreakByCharWrapping;

paraStyle.alignment = NSTextAlignmentLeft;

paraStyle.lineSpacing = space;
```

```
paraStyle.hyphenationFactor = 1.0;

paraStyle.firstLineHeadIndent = 0.0;

paraStyle.paragraphSpacingBefore = 0.0;

paraStyle.headIndent = 0;

paraStyle.tailIndent = 0;UIFont *tfont = [UIFont systemFontOfSize:font];NSDictionary
*dic = @{NSFontAttributeName:tfont, NSParagraphStyleAttributeName:paraStyle,
NSKernAttributeName:@1.5f

};CGSize size = [str boundingRectWithSize:CGSizeMake(width, CGFLOAT_MAX)
options:NSStringDrawingUsesLineFragmentOrigin attributes:dic context:nil].size;return
size.height;

}
```

49、禁止程序运行时自动锁屏

```
[[UIApplication sharedApplication] setIdleTimerDisabled:YES];
```

50、CocoaPods pod install/pod update 更新慢的问题

```
pod install -verbose -no-repo-update pod update -verbose -no-repo-update 如果不加后面
的参数，

默认会升级 CocoaPods 的 spec 仓库，加一个参数可以省略这一步，然后速度就会提升不少。
```

51、修改 textFieldplaceholder 字体颜色和大小

```
textField.placeholder = @``"请输入用户名"``;

 [textField setValue:[UIColor redColor]
forKeyPath:@``"_placeholderLabel.textColor"``];

 [textField setValue:[UIFont boldSystemFontOfSize:16]
forKeyPath:@``"_placeholderLabel.font"``];
```

52、禁止 textField 和 textView 的复制粘贴菜单

```
-(BOOL)canPerformAction:(SEL)action withSender:(id)sender{

 if ([UIMenuController sharedMenuController]) {

  [UIMenuController sharedMenuController].menuVisible = NO;

 }    return NO;

}
```

53、级三级页面隐藏系统 tabbar 1、单个处理

```
YourViewController *yourVC = [YourViewController new];

yourVC.hidesBottomBarWhenPushed = YES;

[self.navigationController pushViewController:yourVC animated:YES];
```

54、取消系统的返回手势

```
self.navigationController.interactivePopGestureRecognizer.enabled = NO;
```

55、百度坐标跟火星坐标相互转换

```
//百度转火星坐标

+ (CLLocationCoordinate2D )bdToGGEncrypt:(CLLocationCoordinate2D)coord

{double x = coord.longitude - 0.0065, y = coord.latitude - 0.006;double z = sqrt(x *
x + y * y) - 0.00002 * sin(y * M_PI);double theta = atan2(y, x) - 0.000003 * cos(x *
M_PI);

CLLocationCoordinate2D transformLocation ;

transformLocation.longitude = z * cos(theta);

transformLocation.latitude = z * sin(theta);return transformLocation;

}//火星坐标转百度坐标

+ (CLLocationCoordinate2D )ggToBDEncrypt:(CLLocationCoordinate2D)coord
```

```
{double x = coord.longitude, y = coord.latitude;double z = sqrt(x * x + y * y) + 0.00002 * sin(y * M_PI);double theta = atan2(y, x) + 0.000003 * cos(x * M_PI);

CLLocationCoordinate2D transformLocation ;

transformLocation.longitude = z * cos(theta) + 0.0065;

transformLocation.latitude = z * sin(theta) + 0.006;return transformLocation;

}
```

56、添加 pch 文件的步骤

```
1：创建新文件 ios->other->PCH file，创建一个 pch 文件："工程名-Prefix.pch"：

2：将 building setting 中的 precompile header 选项的路径添加"$(SRCROOT)/项目名称/pch 文件名"（例如：$(SRCROOT)/LotteryFive/LotteryFive-Prefix.pch）

3：将 Precompile Prefix Header 为 YES，预编译后的 pch 文件会被缓存起来，可以提高编译速度
```

57、关于 Masonry

```
a:



make.equalTo 或 make.greaterThanOrEqualTo （至多） 或 make.lessThanOrEqualTo（至少）

make.left.greaterThanOrEqualTo(label);

make.left.greaterThanOrEqualTo(label.mas_left);//width >= 200 && width <= 400

make.width.greaterThanOrEqualTo(@200);

make.width.lessThanOrEqualTo(@400)

b：masequalTo 和 equalTo 区别：masequalTo 比 equalTo 多了类型转换操作，一般来说，大多数时候两个方法都是 通用的，但是对于数值元素使用 mas_equalTo。对于对象或是多个属性的处理，使用 equalTo。特别是多个属性时，必须使用 equalTo



c：一些简便赋值
```

```
// make top = superview.top + 5, left = superview.left + 10,// bottom = superview.bottom - 15, right = superview.right - 20

make.edges.equalTo(superview).insets(UIEdgeInsetsMake(5, 10, 15, 20))// make width and height greater than or equal to titleLabel

make.size.greaterThanOrEqualTo(titleLabel)// make width = superview.width + 100, height = superview.height - 50

make.size.equalTo(superview).sizeOffset(CGSizeMake(100, -50))// make centerX = superview.centerX - 5, centerY = superview.centerY + 10

make.center.equalTo(superview).centerOffset(CGPointMake(-5, 10))
```

d:and 关键字运用

```
make.left.right.and.bottom.equalTo(superview);

make.top.equalTo(otherView);
```

e:优先：优先权(.priority,.priorityHigh,.priorityMedium,.priorityLow)

.priority 允许您指定一个确切的优先级

.priorityHigh 等价于 UILayoutPriorityDefaultHigh

.priorityMedium 介于高跟低之间

.priorityLow 等价于 UILayoutPriorityDefaultLow

实例：

```
make.left.greaterThanOrEqualTo(label.mas_left).with.priorityLow();

make.top.equalTo(label.mas_top).with.priority(600);
```

g:使用 mas_makeConstraints 创建 constraint 后，你可以使用局部变量或属性来保存以便下次引用它；如果创建多个 constraints，你可以采用数组来保存它们// 局部或者全局@property (nonatomic, strong) MASConstraint *topConstraint;// 创建约束并赋值

```
[view1 mas_makeConstraints:^(MASConstraintMaker *make) {self.topConstraint = make.top.equalTo(superview.mas_top).with.offset(padding.top);
```

```
make.left.equalTo(superview.mas_left).with.offset(padding.left);

}];// 过后可以直接访问 self.topConstraint

[self.topConstraint uninstall];
```

h:mas_updateConstraints 更新约束，有时你需要更新 constraint(例如，动画和调试)而不是创建固定 constraint，可以使用 mas_updateConstraints 方法

```
- (void)updateConstraints {

[self.growingButton mas_updateConstraints:^(MASConstraintMaker *make) {

make.center.equalTo(self);

make.width.equalTo(@(self.buttonSize.width)).priorityLow();

make.height.equalTo(@(self.buttonSize.height)).priorityLow();

make.width.lessThanOrEqualTo(self);

make.height.lessThanOrEqualTo(self);

}];//调用父 updateConstraints

[super updateConstraints];

}
```

i:mas_remakeConstraints 更新约束，mas_remakeConstraints 与 mas_updateConstraints 比较相似，都是更新 constraint。不过，mas_remakeConstraints 是删除之前 constraint，然后再添加新的 constraint(适用于移动动画)；而 mas_updateConstraints 只是更新 constraint 的值。

```
- (void)changeButtonPosition {

[self.button mas_remakeConstraints:^(MASConstraintMaker *make) {

make.size.equalTo(self.buttonSize);if (topLeft) {

make.top.and.left.offset(10);

} else {

make.bottom.and.right.offset(-10);
```

```
}

}];

}
```

58、UIWebView 在 IOS9 下底部出现黑边解决方式

```
UIWebView 底部的黑条很难看(在 IOS8 下不会，在 IOS9 会出现)，特别是在底部还有透明控件的时候，

隐藏的做法其实很简单，只需要将 opaque 设为 NO，背景色设为 clearColor 即可
```

59、数组逆序遍历

```
1、枚举法

NSArray *array = @[@"1",@"2",@"3",@"5",@"6"];

[array enumerateObjectsWithOptions:NSEnumerationReverse usingBlock:^(id  _Nonnull obj,
NSUInteger idx, BOOL * _Nonnull stop) {NSLog(@"%@",obj);

}];2、for 循环

NSArray*array=@[@"1",@"2",@"3",@"5",@"6"];for (NSInteger index = array.count-1;
index>=0; index--) {NSLog(@"%@",array[index]);}
```

60、把时间字符串 2015-04-10 格式化日期转为 NSDate 类型

```
参考答案:

NSString *timeStr = @"2015-04-10";NSDateFormatter *formatter = [[NSDateFormatter alloc]
init];

formatter.dateFormat = @"yyyy-MM-dd";

formatter.timeZone = [NSTimeZone defaultTimeZone];NSDate *date = [formatter
dateFromString:timeStr];// 2015-04-09 16:00:00 +0000NSLog(@"%@", date);
```

61、远程推送原理是什么？

由 App 向 iOS 设备发送一个注册通知

iOS 向 APNs 远程推送服务器发送 App 的 Bundle Id 和设备的 UDID

APNs 根据设备的 UDID 和 App 的 Bundle Id 生成 deviceToken 再发回给 App

App 再将 deviceToken 发送给远程推送服务器(商家自己的服务器)，由服务器保存在数据库中

当商家想发送推送时，在远程推送服务器中输入要发送的消息并选择发给哪些用户的 deviceToken，由远程推送服务器发送给 APNs

APNs 根据 deviceToken 发送给对应的用户

62、http 和 scoket 通信的区别?

socket 连接相关库,TCP,UDP 的连接方法,HTTP 的几种常用方式?

http 和 scoket 通信的区别: http 是客户端用 http 协议进行请求,发送请求时候需要封装 http 请求头,并绑定请求的数据,服务器一般有 web 服务器配 合(当然也非绝对)。 http 请求方式为客户端主动发起请求,服务器才能给响应,一次请求完毕后则断开连接,以节省资 源。服务器不能主动给客户端响应(除非采取 http 长连接技术)。iphone 主要使用类是 NSUrlConnection。

scoket 是客户端跟服务器直接使用 socket“套接字”进行连接,并没有规定连接后断开,所以客户端和服务器可以保持连 接通道,双方都可以主动发送数据。一般在游戏开发或股票开发这种要求即时性很强并且保持发送数据量比较大的场合使用。主要使用类是 CFSocketRef。

UDP:是用户数据报协议:主要用在实时性要求高以及对质量相对较弱的地方,但面对现在高质量的线路不是容易丢包除非 是一些拥塞条件下,如流媒体

TCP:是传输控制协议:是面连接的,那么运行环境必然要求其可靠性不可丢包有良好的拥塞控制机制如 http ftp telnet 等

http 的常用方式:get,post

63、import、include 和@class 有什么区别

@class 一般用于头文件中需要声明该类的某个实例变量的时候用到,它只是声明了一个类名,关于这个类的内部实现都没 有告诉编译器,在 m 文件中还是需要使用#import。

而#import 比起#include 的好处就是不会引起交叉编译。

64、举出 5 个以上你所熟悉的 ios sdk 库有哪些和第三方库有哪些?

ios-sdk:Foundation.framework,CoreGraphics.framework,UIKit.framework, MediaPlayer.framework, CoreAudio.framework

第三方库: 1.json 编码解码;

2. ASIHTTPRequest 等相关协议封装;
3. EGORefreshTableHeaderView 下拉刷新代 码;
4. AsyncImageView 异步加载图片并缓存;
5. 5.SDWebImage——简化网络图片处理

65、viewDidLoad,viewWillAppear,viewDidUnload,dealloc、init 分别是在什么时 候调用的?在自定义 ViewController 的时候这几个函数里面应该做什么工作?

1、viewDidLoad 此方法只有当 view 从 nib 文件初始化的时候才被调用

2、viewDidUnload 当系统内存吃紧的时候会调用该方法,在该方法中将所有 IBOutlet(无论是 property 还是实例变量) 置 为 nil(系统 release view 时已经将其 release 掉了)在该方法中释放其他与 view 有关的对象、其他在运行时创建(但 非系统必须)的对象、在 viewDidLoad 中 被创建的对象、缓存数据等 release 对象后,将对象置为 nil(IBOutlet 只需要 将其置为 nil,系统 release view 时已经将其 release 掉了) dealloc 方法,viewDidUnload 和 dealloc 方法没有关联,dealloc 还是继续做它该做的事情流程应该是这样:(loadView/nib 文件)来加载 view 到内存 -->viewDidLoad 函数进一步初始化这些 view -->内存不足时,调用 viewDidUnload 函数释放 views -->当需要使用 view 时有回到第一步如此循环

3、viewWillAppear 方法,视图即将过渡到屏幕上时调用,(一般在返回需要刷新页面时,我都选择使用代理,所以很少用到)

4、viewWillDisappear 方法,这个 A->B 之后,A 在 B 之后的操作

66、简述你对 UIView、UIWindow 和 CALayer 的理解

UIView:属于 UIkit.framework 框架,负责渲染矩形区域的内容,为矩形区域添加动画,响应区域的触摸事件,布局和管 理一个或多个子视图 ;

UIWindow:属于 UIKit.framework 框架,是一种特殊的 UIView,通常在一个程序中只会有一个 UIWindow,但可以手 动创建多个 UIWindow,同时加到程序里面。UIWindow 在程序中主要起到三个作用: 1、作为容器,包含 app 所要显示的所有视图 2、传递触摸消息到程序中 view 和其他对象 3、与 UIViewController 协同工作,方便完成设备方向旋转的支持

CAlayer:属于 QuartzCore.framework,是用来绘制内容的,对内容进行动画处理依赖与 UIView 来进行显示,不能处 理用户事件。UIView 和 CALayer 是相互依赖的,UIView 依赖 CALayer 提供内容,CALayer 依赖 UIView 一共容器显示 绘制内容。

延伸: UIViewController:管理视图的子视图,每个视图控制器都有一个自带的视图,并且负责这个视图相关的一切事务。方便 管理视图中的子视图,负责 model 与 view 的通信;检测设备旋转以及内存警告;是所有视图控制类的积累,定义了控制器 的基本功能。

67、为什么很多内置类如 UITableViewController 的 delegate 属性都是 assign 而不是 retain 的?

会引起循环引用----若是 retain,在 alloc 一次之后,若 release 一次,会导致内训泄漏,若 release 两次会导致两个对象的 dealloc 嵌套执行,结果就是都没有执行成功,最后崩溃! 所有的引用计数系统,都存在循环应用的问题。例如下面的引用关系:

对象 a 创建并引用到了对象 b. * 对象 b 创建并引用到了对象 c. * 对象 c 创建并引用到了对象 b. 这时候 b 和 c 的引用计数分别是 2 和 1。 当 a 不再使用 b,调用 release 释放对 b 的所有权,因为 c 还引用了 b,所以 b 的引用计数为 1,b 不会被释放。 b 不释放,c 的引用计数就是 1,c 也不会被释放。从此,b 和 c 永远留在内存中。这种情况,必须打断循环引用,通过其他规则来维护引用关系。

我们常见的 delegate 往往是 assign 方式的属性而不是 retain 方式 的属性,赋值不会增加引用计数,就是为了防止 delegation 两端产生不必要的循环引用。 如果一个 UITableViewController 对象 a 通过 retain 获取了 UITableView 对象 b 的所有权,这个 UITableView 对象 b 的 delegate 又是 a,如果这个 delegate 是 retain 方式的,那基本上就没有机会释放这两个对象了。自己在设计使用 delegate 模式时,也要注意这点。

68、简述 NotificationCenter、KVC、KVO、Delegate?并说明它们之间的区别?

Notification 是观察者模式的实现,KVO 是观察者模式的 OB-C 底层实现。
NOtification 通过 Notifydcation addobserver 和 remove observer 工作。

KVO 是键值监听,键值观察机制,提供了观察某一属性变化的方法

KVC 是键值编码,是一种间接访问对象的属性,使用字符串来标示属性(例如:setValue:forKey:)

Delegate:把某个对象要做的事情委托给别的对象去做。那么别的对象就是这个对象的代理,代替它来打理要做的 事。反映到程序中,首先要明确一个对象的委托方是哪个对象,委托所做的内容是什么。

69、线程与进程的区别和联系?

线程是进程的基本单位 进程和线程都是由操作系统所体会的程序运行的基本单元, 系统利用该基本单元实现系统对应用的并发性。 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间, 一个进程崩溃后,

在保护模式下 不会对其它进程产生影响, 而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量, 但线程之间没有单独的 地址空间, 一个线程死掉就等于整个进程死掉, 所以多进程的程序要比多线程的程序健壮, 但在进程切换时, 耗费资源较 大, 效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作, 只能用线程, 不能用进程

70、简述多线程的作用以及什么地方会用到多线程?OC 实现多线程的方法有哪些?谈谈多线程安全问题的几种解决方案?何为线程同步,如何实现的?分线程回调主线程方法是什么,有什么作用?

(1)、多线程的作用:可以解决负载均衡问题,充分利用 cpu 资源 。为了提高 CPU 的使用率,采用多线程的方式去同时完 成几件事情而互不干扰,

(2)、大多情况下,要用到多线程的主要是需要处理大量的 IO 操作时或处理的情况需要花大量的时间等等,比如:读写文 件、视频图像的采集、处理、显示、保存等。

(3)、ios 有三种主要方法:1、NSThread。2、NSOperation。3、GCD。

(4)解决方案:使用锁:锁是线程编程同步工具的基础。锁可以让你很容易保护代码中一大块区域以便你可以确保代码的正 确性。使用 NSLock 类;使用 @synchronized 指令等。

(5)回到主线程的方法: dispatch_async(dispatch_get_main_queue(), ^{ });

作用:主线程是显示 UI 界面,子线程多数是进行数据处理

71、Objective-C 如何对内存管理的,说说你的看法和解决方法?每个对象都有一个引用计数器，每个新对象的计数器是 1，当对象的计数器减为 0 时，就会被销毁通过 retain 可以让对象的计数器+1、release 可以让对象的计数器-1 还可以通过 autorelease pool 管理内存如果用 ARC，编译器会自动生成管理内存的代码

72、内存管理的几条原则时什么？按照默认法则.哪些方法生成的对象需要手动释放？在和 property 结合的时候怎样有效的避免内存泄露？

只要调用了 alloc、copy、new 方法产生了一个新对象，都必须在最后调用一次 release 或者 autorelease

只要调用了 retain，都必须在最后调用一次 release 或者 autorelease

@property 如果用了 copy 或者 retian，就需要对不再使用的属性做一次 release 操作

如果用了 ARC，另外讨论

73、What is Singleton? (单例是什么)

单例：保证程序运行过程中，永远只有一个对象实例

目的是：全局共享一份资源、节省不必要的内存开销

74、对象是什么时候被释放的？

每个对象都有一个引用计数器，每个新对象的计数器是 1，当对象的计数器减为 0 时，就会被销毁

75、什么情况下会发生内存泄漏和内存溢出？

当程序在申请内存后，无法释放已申请的内存空间(例如一个对象或者变量使用完成后没有释放,这个对象一直占用着内存)，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。内存泄露会最终会导致内存溢出！

当程序在申请内存时，没有足够的内存空间供其使用，出现 out of memory；比如申请了一个 int,但给它存了 long 才能存下的数，那就是内存溢出。