2020-逻辑教育iOS面试题集合

总结iOS常见面试题，以及BAT大厂面试分享！笔者一道一道总结，如果你觉得还不错，小心心 **Star** 走一波....
Thanks♪(･ω･)ﾉ

⚠️特别说明：部分来源网络摘抄，如有疑问，立即删除！⚠️

> *本人博客地址：[Cooci-掘金]*

1：谈谈你对KVC的理解

`KVC` 可以通过 `key` 直接访问对象的属性，或者给对象的属性赋值，这样可以在运行时动态的访问或修改对象的属性

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

- 1、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
- 2、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。
- 3、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。
- 4、和上面一样，如果该类即没有set：方法，也没有_和_is成员变量，KVC机制再会继续搜索和is的成员变量。再给它们赋值。
- 5、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。

如果想禁用KVC，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：

- 1、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
- 2、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。

- 3、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。

- 4、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：iOS项目中引用多个第三方库引发冲突的解决方法

> *可能有很多小伙伴还不太清楚，动静态库的开发，这里推荐一篇博客：iOS-制作.a静态库SDK和使用.a静态库*

如果我们存在三方库冲突就会保存：`duplicate symbol _OBJC_IVAR_$_xxxx in:`

目前见效最快的就是把 `.framework` 选中， `taggert Membership` 的对勾取消掉，就编译没有问题了，但是后续的其他问题可能还会出现

> *我想说的是像这种开源的使用率很高的源代码本不应该包含在lib库中，就算是你要包含那也要改个名字是吧。不过没办法现在人家既然包含，我们就只有想办法分离了*

- `mkdir armv7`：创建临时文件夹
- `lipo libALMovie.a -thin armv7 -output armv7/armv7.a`：取出armv7平台的包
- `ar -t armv7/armv7.a`：查看库中所包含的文件列表
- `cd armv7 && ar xv armv7.a`：解压出object file（即.o后缀文件）
- `rm ALButton.o`：找到冲突的包，删除掉（此步可以多次操作）
- `cd … && ar rcs armv7.a armv7/*.o`：重新打包object file
- 多平台的SDK的话，需要多次操作第4步。操作完成后，合并多个平台的文件为一个.a文件：
  `lipo -create armv7.a arm64.a -output new.a`
- 将修改好的文件， 拖拽到原文件夹下，替换原文件即可。

3：GCD实现多读单写

比如在内存中维护一份数据，有多处地方可能会同时操作这块数据，怎么能保证数据安全？这道题目总结得到要满足以下三点：

- 1.读写互斥
- 2.写写互斥
- 3.读读并发

```
@implementation KCPerson
- (instancetype)init
{
    if (self = [super init]) {
       _concurrentQueue = dispatch_queue_create("com.kc_person.syncQueue", DISPATCH_QUEUE_CONCURRENT);
       _dic = [NSMutableDictionary dictionary];
    }
    return self;
}
- (void)kc_setSafeObject:(id)object forKey:(NSString *)key{
    key = [key copy];
    dispatch_barrier_async(_concurrentQueue, ^{
       [_dic setObject:object key:key];
    });
}

- (id)kc_safeObjectForKey: :(NSString *)key{
    __block NSString *temp;
    dispatch_sync(_concurrentQueue, ^{
        temp =[_dic objectForKey: key];
    });
    return temp;
}
@end
```

- 首先我们要维系一个GCD 队列，最好不用全局队列，毕竟大家都知道全局队列遇到栅栏函数是有坑点的，这里就不分析了！
- 因为考虑性能 死锁 堵塞的因素不考虑串行队列，用的是自定义的并发队列！
  `_concurrentQueue = dispatch_queue_create("com.kc_person.syncQueue", DISPATCH_QUEUE_CONCURRENT);`
- 首先我们来看看读操作: `kc_safeObjectForKey` 我们考虑到多线程影响是不能用异步函数的！说明：
  - 线程2 获取： `name` 线程3 获取 `age`
  - 如果因为异步并发，导致混乱 本来读的是 `name` 结果读到了 `age`
  - 我们允许多个任务同时进去! 但是读操作需要同步返回，所以我们选择: `同步函数` **（读读并发）**
- 我们再来看看写操作，在写操作的时候对key进行了copy, 关于此处的解释，插入一段来自参考文献的引用:

> *函数调用者可以自由传递一个 `NSMutableString` 的 `key` ，并且能够在函数返回后修改它。因此我们必须对传入的字符串使用 `copy` 操作以确保函数能够正确地工作。如果传入的字符串不是可变的（也就是正常的 `NSString` 类型），调用 `copy` 基本上是个空操作。*

- 这里我们选择 `dispatch_barrier_async` , 为什么是栅栏函数而不是异步函数或者同步函数，下面分析：
  - 栅栏函数任务：之前所有的任务执行完毕，并且在它后面的任务开始之前，期间不会有其他的任务执行，这样比较好的促使 写操作一个接一个写 **（写写互斥）**，不会乱！
  - 为什么不是异步函数？应该很容易分析，毕竟会产生混乱！

- 为什么不用同步函数？如果读写都操作了，那么用同步函数，就有可能存在：我写需要等待读操作回来才能执行，显然这里是不合理！

4:讲一下atomic的实现机制；为什么不能保证绝对的线程安全（最好可以结合场景来说）？

A: atomic的实现机制

- `atomic` 是 `property` 的修饰词之一，表示是原子性的，使用方式为 `@property(atomic)int age` ;此时编译器会自动生成 `getter/setter` 方法，最终会调用 `objc_getProperty` 和 `objc_setProperty` 方法来进行存取属性。
- 若此时属性用 `atomic` 修饰的话，在这两个方法内部使用 `os_unfair_lock` 来进行加锁，来保证读写的原子性。锁都在 `PropertyLocks` 中保存着（在iOS平台会初始化8个，mac平台64个），在用之前，会把锁都初始化好，在需要用到时，用对象的地址加上成员变量的偏移量为 `key` ，去 `PropertyLocks` 中去取。因此存取时用的是同一个锁，所以atomic能保证属性的存取时是线程安全的。
- 注：由于锁是有限的，不用对象，不同属性的读取用的也可能是同一个锁

B: atomic为什么不能保证绝对的线程安全?

- `atomic` 在 `getter/setter` 方法中加锁，仅保证了存取时的线程安全，假设我们的属性是 `@property(atomic)NSMutableArray *array` ;可变的容器时,无法保证对容器的修改是线程安全的.
- 在编译器自动生产的 `getter/setter` 方法，最终会调用 `objc_getProperty` 和 `objc_setProperty` 方法存取属性，在此方法内部保证了读写时的线程安全的，当我们重写 `getter/setter` 方法时，就只能依靠自己在 `getter/setter` 中保证线程安全

5. Autoreleasepool所使用的数据结构是什么？AutoreleasePoolPage结构体了解么?

- `Autoreleasepool` 是由多个 `AutoreleasePoolPage` 以双向链表的形式连接起来的.
- `Autoreleasepool` 的基本原理：在每个自动释放池创建的时候，会在当前的 `AutoreleasePoolPage` 中设置一个标记位，在此期间，当有对象调用 `autorelsease` 时，会把对象添加 `AutoreleasePoolPage` 中
- 若当前页添加满了，会初始化一个新页，然后用双向量表链接起来，并把新初始化的这一页设置为 `hotPage` ,当自动释放池pop时，从最下面依次往上pop，调用每个对象的 `release` 方法，直到遇到标志位。

`AutoreleasePoolPage` 结构如下

```
class AutoreleasePoolPage {
    magic_t const magic;
    id *next;//下一个存放autorelease对象的地址
    pthread_t const thread; //AutoreleasePoolPage 所在的线程
    AutoreleasePoolPage * const parent;//父节点
    AutoreleasePoolPage *child;//子节点
    uint32_t const depth;//深度,也可以理解为当前page在链表中的位置
    uint32_t hiwat;
}
```

6: iOS中内省的几个方法？class方法和objc_getClass方法有什么区别?

- 1: 什么是内省?

> *在计算机科学中，内省是指计算机程序在运行时（Run time）检查对象（Object）类型的一种能力，通常也可以称作运行时类型检查。*
> *不应该将内省和反射混淆。相对于内省，反射更进一步，是指计算机程序在运行时（Run time）可以访问、检测和修改它本身状态或行为的一种能力。*

- 2：iOS中内省的几个方法?

`isMemberOfClass` //对象是否是某个类型的对象
`isKindOfClass` //对象是否是某个类型或某个类型子类的对象
`isSubclassOfClass` //某个类对象是否是另一个类型的子类
`isAncestorOfObject` //某个类对象是否是另一个类型的父类
`respondsToSelector` //是否能响应某个方法
`conformsToProtocol` //是否遵循某个协议

- 3: `class` 方法和 `object_getClass` 方法有什么区别?
  - 实例 `class` 方法就直接返回 `object_getClass(self)`
  - 类 `class` 方法直接返回 `self`，而 `object_getClass`(类对象)，则返回的是元类

7: 分类和扩展有什么区别？可以分别用来做什么？分类有哪些局限性？分类的结构体里面有哪些成员？

- 1:分类主要用来为某个类添加方法，属性，协议（我一般用来为系统的类扩展方法或者把某个复杂的类的按照功能拆到不同的文件里）
- 2:扩展主要用来为某个类原来没有的成员变量、属性、方法。注：方法只是声明（我一般用扩展来声明私有属性，或者把.h的只读属性重写成可读写的）

分类和扩展的区别：

- 分类是在运行时把分类信息合并到类信息中，而扩展是在编译时，就把信息合并到类中的
- 分类声明的属性，只会生成 `getter/setter` 方法的声明，不会自动生成成员变量和 `getter/setter` 方法的实现，而扩展会
- 分类不可用为类添加实例变量，而扩展可以
- 分类可以为类添加方法的实现，而扩展只能声明方法，而不能实现

分类的局限性：

- 无法为类添加实例变量，但可通过关联对象进行实现，注：关联对象中内存管理没有weak，用时需要注意野指针的问题，可通过其他办法来实现，具体可参考iOS weak 关键字漫谈
- 分类的方法若和类中原本的实现重名，会覆盖原本方法的实现，注：并不是真正的覆盖
- 多个分类的方法重名，会调用最后编译的那个分类的实现

分类的结构体里有哪些成员

```
struct category_t {
    const char *name; //名字
    classref_t cls; //类的引用
    struct method_list_t *instanceMethods;//实例方法列表
    struct method_list_t *classMethods;//类方法列表
    struct protocol_list_t *protocols;//协议列表
    struct property_list_t *instanceProperties;//实例属性列表
    // 此属性不一定真正的存在
    struct property_list_t *_classProperties;//类属性列表
};
```

8：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。

2.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。

9：HTTPS和HTTP的区别

HTTPS协议 = HTTP协议 + SSL/TLS协议

- `SSL` 的全称是 `Secure Sockets Layer` ，即安全套接层协议，是为网络通信提供安全及数据完整性的一种安全协议。
- TLS的全称是 `Transport Layer Security` ，即安全传输层协议。

即HTTPS是安全的HTTP。

`https` , 全称 `Hyper Text Transfer Protocol Secure` ，相比 `http` ，多了一个 `secure` ，这一个 `secure` 是怎么来的呢？这是由 `TLS (SSL)` 提供的！大概就是一个叫 `openSSL` 的 `library` 提供的。 `https` 和 `http` 都属于 `application layer` ，基于TCP（以及UDP）协议，但是又完全不一样。TCP用的port是80， https用的是443（值得一提的是，google发明了一个新的协议，叫QUIC，并不基于TCP，用的port也是443， 同样是用来给https的。谷歌好牛逼啊。）总体来说，https和http类似，但是比http安全。

10：TCP为什么要三次握手，四次挥手?

三次握手：

- 客户端向服务端发起请求链接，首先发送 `SYN` 报文， `SYN=1，seq=x` ,并且客户端进入 `SYN_SENT` 状态
- 服务端收到请求链接，服务端向客户端进行回复，并发送响应报文， `SYN=1，seq=y,ACK=1,ack=x+1` ,并且服务端进入到 `SYN_RCVD` 状态
- 客户端收到确认报文后，向服务端发送确认报文， `ACK=1，ack=y+1` ，此时客户端进入到 `ESTABLISHED` ，服务端收到用户端发送过来的确认报文后，也进入到 `ESTABLISHED` 状态，此时链接创建成功

四次挥手：

- 客户端向服务端发起关闭链接，并停止发送数据
- 服务端收到关闭链接的请求时，向客户端发送回应，我知道了，然后停止接收数据
- 当服务端发送数据结束之后，向客户端发起关闭链接，并停止发送数据
- 客户端收到关闭链接的请求时，向服务端发送回应，我知道了，然后停止接收数据

为什么需要三次握手：

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误，假设这是一个早已失效的报文段。但 `server` 收到此失效的连接请求报文段后，就误认为是 `client` 再次发出的一个新的连接请求。于是就向 `client` 发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在 `client` 并没有发出建立连接的请求，因此不会理睬 `server` 的确认，也不会向 `server` 发送数据。但 `server` 却以为新的运输连接已经

建立，并一直等待 `client` 发来数据。这样， `server` 的很多资源就白白浪费掉了。

为什么需要四次挥手：

因为TCP是全双工通信的，在接收到客户端的关闭请求时，还可能在向客户端发送着数据，因此不能再回应关闭链接的请求时，同时发送关闭链接的请求

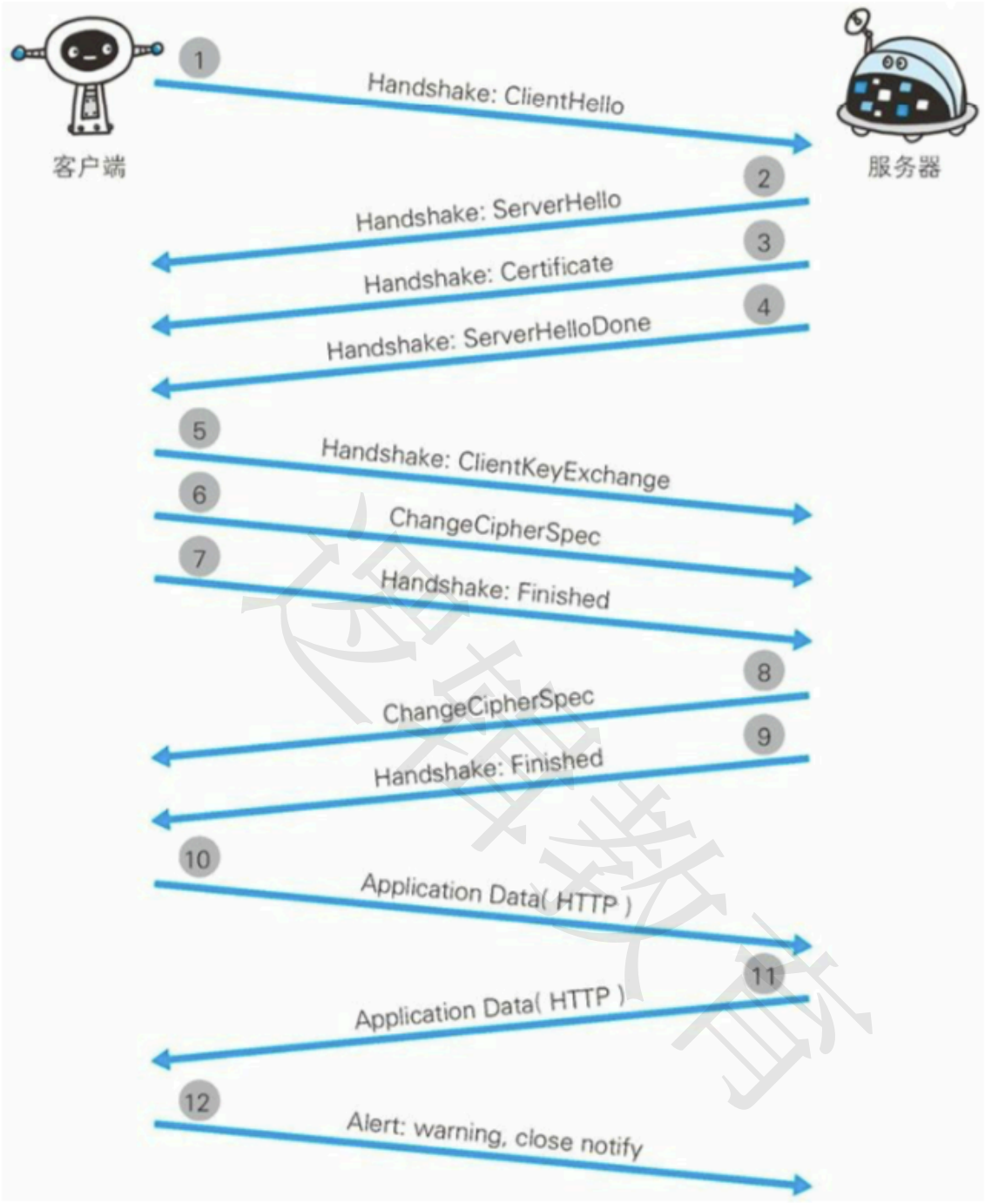
11. 对称加密和非对称加密的区别？分别有哪些算法的实现？

对称加密，加密的加密和解密使用同一密钥。

- 非对称加密，使用一对密钥用于加密和解密，分别为公开密钥和私有密钥。公开密钥所有人都可以获得，通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密，接收方收到通信内容后使用私有密钥解密。
- 对称加密常用的算法实现有AES,ChaCha20,DES,不过DES被认为是不安全的;非对称加密用的算法实现有RSA,ECC

12. HTTPS的握手流程？为什么密钥的传递需要使用非对称加密？双向认证了解么？

HTTPS的握手流程，如下图，摘自图解HTTP



图：**HTTPS** 通信

- 1：客户端发送Client Hello 报文开始SSL通信。报文中包含客户端支持的SSL的版本，加密组件列表。
- 2：服务器收到之后，会以Server Hello 报文作为应答。和客户端一样，报文中包含客户端支持的SSL的版本，加密组件列表。服务器的加密组件内容是从接收到的客户端加密组件内筛选出来的
- 3：服务器发送Certificate报文。报文中包含公开密钥证书。
- 4：然后服务器发送Server Hello Done报文通知客户端，最初阶段的SSL握手协商部分结束

- 5：SSL第一次握手结束之后，客户端以Client Key Exchange报文作为会议。报文中包含通信加密中使用的一种被称为Pre-master secret的随机密码串
- 6：接着客户端发送Change Cipher Space报文。该报文会提示服务器，在次报文之后的通信会采用Pre-master secret密钥加密
- 7：客户端发送Finished 报文。该报文包含链接至今全部报文的整体校验值。这次握手协商是否能够成功，要以服务器是否能够正确揭秘该报文作为判定标准
- 8：服务器同样发送Change Cipher Space报文。
- 9：服务器同样发送Finished报文。
- 10：服务器和客户端的Finished报文交换完毕之后，SSL连接建立完成，从此开始HTTP通信，通信的内容都使用Pre-master secret加密。然后开始发送HTTP请求
- 11：应用层收到HTTP请求之后，发送HTTP响应
- 12：最后有客户端断开连接

为什么密钥的传递需要使用非对称加密?

使用非对称加密是为了后面客户端生成的 `Pre-master secret` 密钥的安全，通过上面的步骤能得知，服务器向客户端发送公钥证书这一步是有可能被别人拦截的，如果使用对称加密的话，在客户端向服务端发送 `Pre-master secret` 密钥的时候，被黑客拦截的话，就能够使用公钥进行解码，就无法保证 `Pre-master secret` 密钥的安全了

双向认证了解么?

上面的HTTPS的通信流程只验证了服务端的身份，而服务端没有验证客户端的身份，双向认证是服务端也要确保客户端的身份，大概流程是客户端在校验完服务器的证书之后，会向服务器发送自己的公钥，然后服务端用公钥加密产生一个新的密钥，传给客户端，客户端再用私钥解密，以后就用此密钥进行对称加密的通信

13. 如何用Charles抓HTTPS的包？其中原理和流程是什么?

流程:

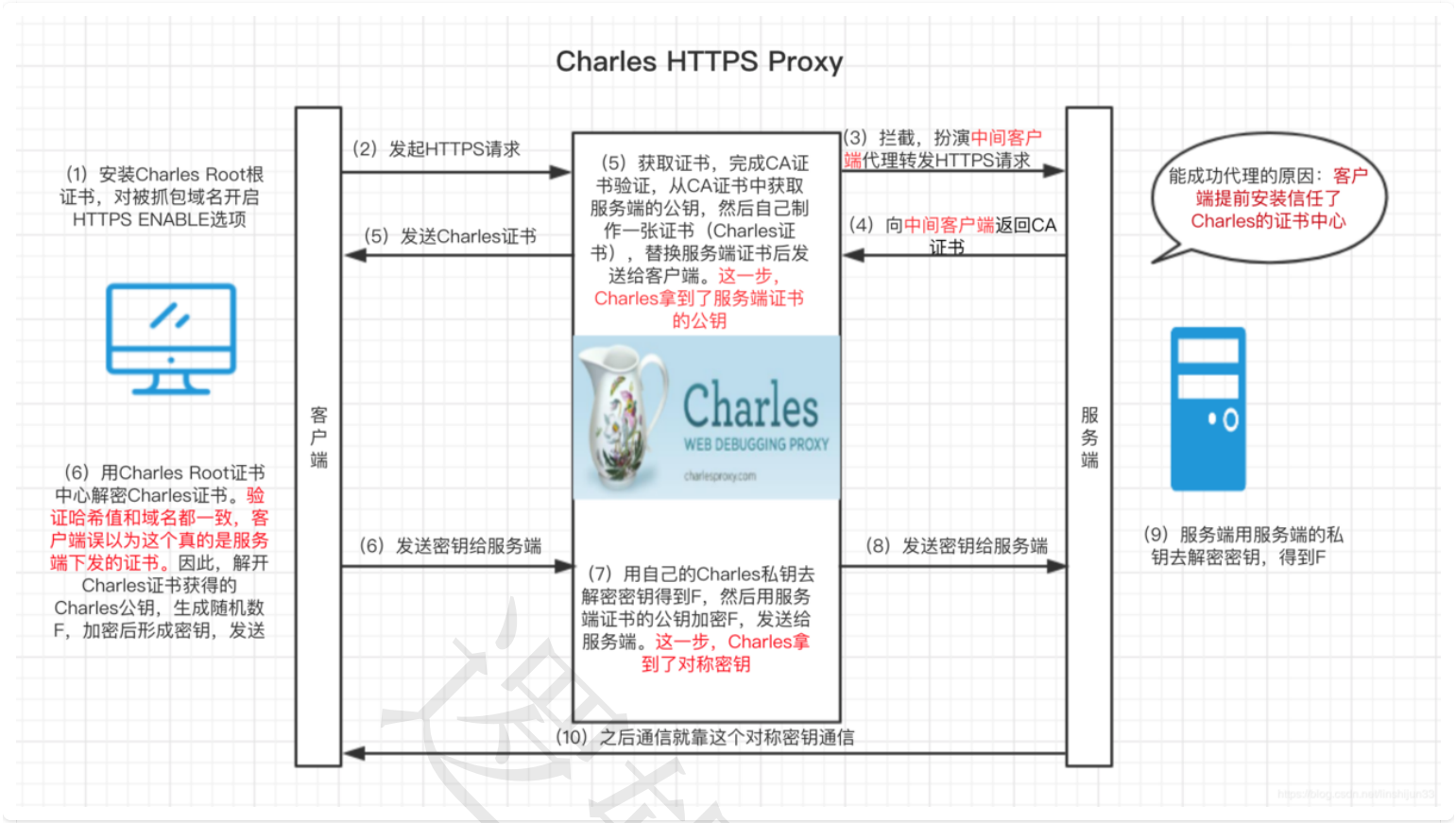
- 首先在手机上安装Charles证书
- 在代理设置中开启Enable SSL Proxying
- 之后添加需要抓取服务端的地址

原理:

`Charles` 作为中间人，对客户端伪装成服务端，对服务端伪装成客户端。简单来说:

- 截获客户端的HTTPS请求，伪装成中间人客户端去向服务端发送HTTPS请求
- 接受服务端返回，用自己的证书伪装成中间人服务端向客户端发送数据内容。

具体流程如下图:[扯一扯HTTPS单向认证、双向认证、抓包原理、反抓包策略]



14. 什么是中间人攻击？如何避免？

中间人攻击就是截获到客户端的请求以及服务器的响应，比如 `Charles` 抓取HTTPS的包就属于中间人攻击。

避免的方式：客户端可以预埋证书在本地，然后进行证书的比较是否是匹配的

15. 了解编译的过程么？分为哪几个步骤？

1:预编译：主要处理以“#”开始的预编译指令。

2:编译：

- 词法分析：将字符序列分割成一系列的记号。
- 语法分析：根据产生的记号进行语法分析生成语法树。
- 语义分析：分析语法树的语义，进行类型的匹配、转换、标识等。
- 中间代码生成：源码级优化器将语法树转换成中间代码，然后进行源码级优化，比如把 1+2 优化为 3。中间代码使得编译器被分为前端和后端，不同的平台可以利用不同的编译器后端将中间代码转换为机器代码，实现跨平台。
- 目标代码生成：此后的过程属于编译器后端，代码生成器将中间代码转换成目标代码（汇编代码），其后目标代码优化器对目标代码进行优化，比如调整寻址方式、使用位移代替乘法、删除多余指令、调整指令顺序等。

3:汇编：汇编器将汇编代码转变成机器指令。

- 静态链接：链接器将各个已经编译成机器指令的目标文件链接起来，经过重定位过后输出一个可执行文件。

- 装载：装载可执行文件、装载其依赖的共享对象。
- 动态链接：动态链接器将可执行文件和共享对象中需要重定位的位置进行修正。
- 最后，进程的控制权转交给程序入口，程序终于运行起来了。

16. 静态链接了解么？静态库和动态库的区别?

- 静态链接是指将多个目标文件合并为一个可执行文件，直观感觉就是将所有目标文件的段合并。需要注意的是可执行文件与目标文件的结构基本一致，不同的是是否“可执行”。
- 静态库：链接时完整地拷贝至可执行文件中，被多次使用就有多份冗余拷贝。
- 动态库：链接时不复制，程序运行时由系统动态加载到内存，供程序调用，系统只加载一次，多个程序共用，节省内存。

17. App网络层有哪些优化策略?

- 优化DNS解析和缓存
- 对传输的数据进行压缩，减少传输的数据
- 使用缓存手段减少请求的发起次数
- 使用策略来减少请求的发起次数，比如在上一个请求未着地之前，不进行新的请求
- 避免网络抖动，提供重发机制

18：[self class] 与 [super class]

```
@implementation Son : Father
- (id)init
{
    self = [super init];
    if (self)
    {
        NSLog(@"%@", NSStringFromClass([self class]));
        NSLog(@"%@", NSStringFromClass([super class]));
    }
return self;
}
@end
```

self和super的区别：

- `self` 是类的一个隐藏参数，每个方法的实现的第一个参数即为 `self` 。
- super并不是隐藏参数，它实际上只是一个**”编译器标示符”**，它负责告诉编译器，当调用方法时，去调用父类的方法，而不是本类中的方法。

在调用 `[super class]` 的时候， `runtime` 会去调用 `objc_msgSendSuper` 方法，而不是 `objc_msgSend`

```
OBJC_EXPORT void objc_msgSendSuper(void /* struct objc_super *super, SEL op, ... */ )
```

```
/// Specifies the superclass of an instance.
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained id receiver;

    /// Specifies the particular superclass of the instance to message.
#if !defined(__cplusplus)  &&  !__OBJC2__
    /* For compatibility with old objc-runtime.h header */
    __unsafe_unretained Class class;
#else
    __unsafe_unretained Class super_class;
#endif
    /* super_class is the first class to search */
}
```

在 `objc_msgSendSuper` 方法中，第一个参数是一个 `objc_super` 的结构体，这个结构体里面有两个变量，一个是接收消息的 `receiver` ，一个是当前类的父类 `super_class` 。

入院考试第一题错误的原因就在这里，误认为 `[super class]` 是调用的 `[super_class class]` 。

`objc_msgSendSuper` 的工作原理应该是这样的:

- 从 `objc_super` 结构体指向的 `superClass` 父类的方法列表开始查找selector，
- 找到后以 `objc->receiver` 去调用父类的这个 `selector` 。注意，最后的调用者是 `objc->receiver` ，而不是 `super_class` ！

那么 `objc_msgSendSuper` 最后就转变成

```
objc_msgSend(objc_super->receiver, @selector(class))

+ (Class)class {
    return self;
}
```

18.isKindOfClass 与 isMemberOfClass

下面代码输出什么?

```
@interface Sark : NSObject
 @end

 @implementation Sark
 @end

 int main(int argc, const char * argv[]) {
@autoreleasepool {
    BOOL res1 = [(id)[NSObject class] isKindOfClass:[NSObject class]];
```

```
    BOOL res2 = [(id)[NSObject class] isMemberOfClass:[NSObject class]];
    BOOL res3 = [(id)[Sark class] isKindOfClass:[Sark class]];
    BOOL res4 = [(id)[Sark class] isMemberOfClass:[Sark class]];

   NSLog(@"%d %d %d %d", res1, res2, res3, res4);
}
return 0;
}
```

先来分析一下源码这两个函数的对象实现

```
+ (Class)class {
    return self;
}

- (Class)class {
    return object_getClass(self);
}

Class object_getClass(id obj)
{
    if (obj) return obj->getIsa();
    else return Nil;
}

inline Class
objc_object::getIsa()
{
    if (isTaggedPointer()) {
        uintptr_t slot = ((uintptr_t)this >> TAG_SLOT_SHIFT) & TAG_SLOT_MASK;
        return objc_tag_classes[slot];
    }
    return ISA();
}

inline Class
objc_object::ISA()
{
    assert(!isTaggedPointer());
    return (Class)(isa.bits & ISA_MASK);
}

+ (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = object_getClass((id)self); tcls; tcls = tcls->superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}

- (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = [self class]; tcls; tcls = tcls->superclass) {
```

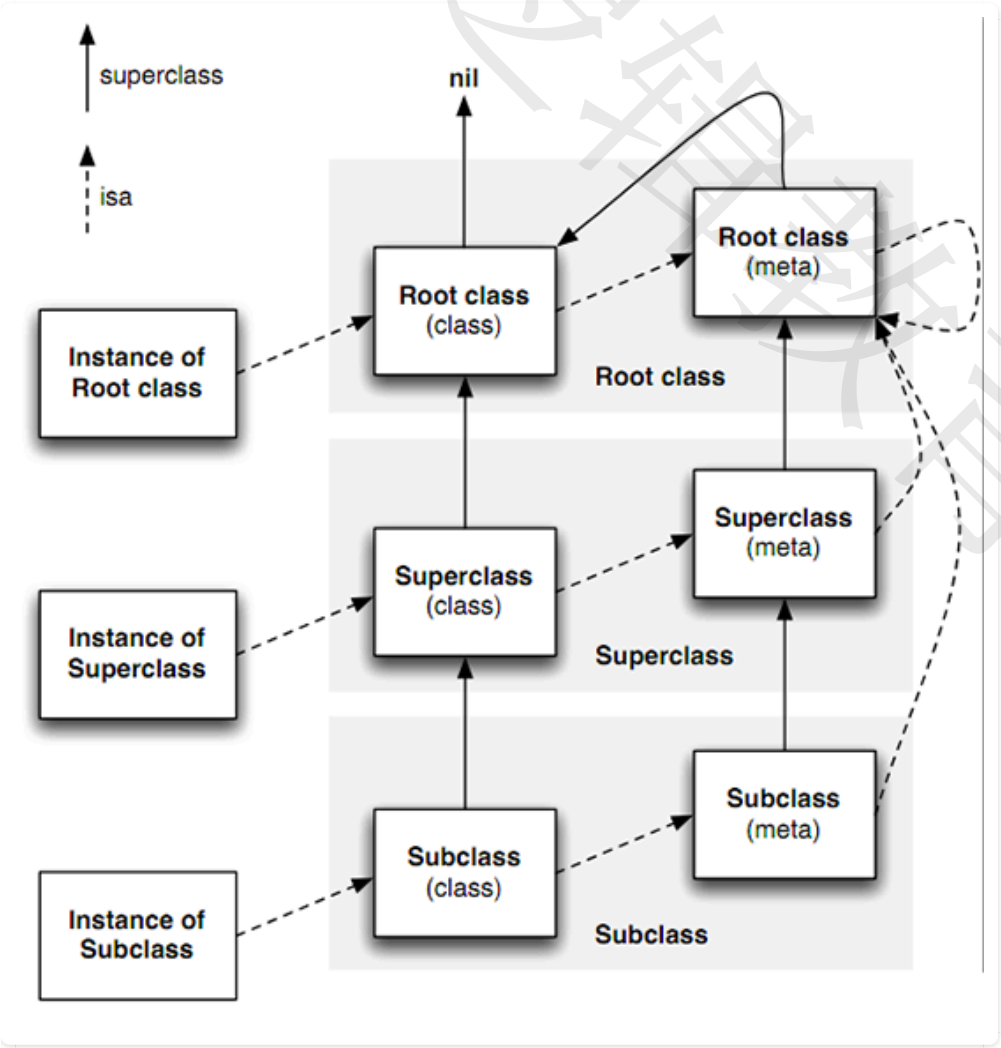
```
        if (tcls == cls) return YES;
    }
    return NO;
}

+ (BOOL)isMemberOfClass:(Class)cls {
    return object_getClass((id)self) == cls;
}

- (BOOL)isMemberOfClass:(Class)cls {
    return [self class] == cls;
}
```

首先题目中NSObject 和 Sark分别调用了class方法。

- `+ (BOOL)isKindOfClass:(Class)cls` 方法内部，会先去获得 `object_getClass` 的类，而 `object_getClass` 的源码实现是去调用当前类的 `obj->getIsa()` ，最后在 `ISA()` 方法中获得 `meta class` 的指针。
- 接着在 `isKindOfClass` 中有一个循环，先判断 `class` 是否等于 `meta class` ，不等就继续循环判断是否等于 `super class` ，不等再继续取 `super class` ，如此循环下去。



- `[NSObject class]` 执行完之后调用 `isKindOfClass` ，第一次判断先判断 `NSObject` 和 `NSObject` 的

`meta class` 是否相等，之前讲到 `meta class` 的时候放了一张很详细的图，从图上我们也可以看出， `NSObject` 的 `meta class` 与本身不等。

- 接着第二次循环判断 `NSObject` 与 `meta class` 的 `superclass` 是否相等。还是从那张图上面我们可以看到：`Root class(meta)` 的 `superclass` 就是 `Root class(class)` ，也就是 `NSObject` 本身。所以第二次循环相等，于是第一行 `res1` 输出应该为 `YES` 。
- 同理， `[Sark class]` 执行完之后调用 `isKindOfClass` ，第一次 `for` 循环， `Sark` 的 `Meta Class` 与 `[Sark class]` 不等，第二次 `for循环` ， `Sark` `Meta Class的super class` 指向的是 `NSObject Meta Class` ， 和 `Sark Class` 不相等。
- 第三次for循环， `NSObject Meta Class` 的 `super class` 指向的是 `NSObject Class` ，和 `Sark Class` 不相等。第四次循环， `NSObject Class` 的 `super class` 指向 `nil` ， 和 `Sark Class` 不相等。第四次循环之后，退出循环，所以第三行的 `res3输出为NO` 。
- 如果把这里的Sark改成它的实例对象， `[sark isKindOfClass:[Sark class]` ，那么此时就应该输出 `YES` 了。因为在 `isKindOfClass` 函数中，判断sark的 `meta class` 是自己的元类 `Sark` ，第一次for循环就能输出 `YES` 了。
- `isMemberOfClass` 的源码实现是拿到自己的 `isa指针` 和自己比较，是否相等。
- 第二行 `isa` 指向 `NSObject` 的 `Meta Class` ，所以和 `NSObject Class` 不相等。第四行， `isa` 指向 `Sark` 的 `Meta Class` ，和 `Sark Class` 也不等，所以第二行 `res2` 和第四行 `res4` 都输出NO。

19.Class与内存地址

下面的代码会？ **`Compile Error / Runtime Crash / NSLog…?`**

```
@interface Sark : NSObject
@property (nonatomic, copy) NSString *name;
- (void)speak;
@end
@implementation Sark
- (void)speak {
    NSLog(@"my name's %@", self.name);
}
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    id cls = [Sark class];
    void *obj = &cls;
    [(__bridge id)obj speak];
}
@end
```

这道题有两个难点。

- 难点一: `obj` 调用 `speak` 方法，到底会不会崩溃。
- 难点二:如果 `speak` 方法不崩溃，应该输出什么？

首先需要谈谈隐藏参数self和_cmd的问题。
当 `[receiver message]` 调用方法时，系统会在运行时偷偷地动态传入两个隐藏参数 `self` 和 `_cmd` ，之所以称它们为隐藏参数，是因为在源代码中没有声明和定义这两个参数。 `self` 在已经明白了，接下来就来说说 `_cmd` 。 `_cmd` 表示当前调用方法，其实它就是一个方法选择器 `SEL` 。

- 难点一，能不能调用 `speak` 方法？

```
id cls = [Sark class];
void *obj = &cls;
```

答案是可以的。 `obj` 被转换成了一个指向 `Sark Class` 的指针，然后使用 `id` 转换成了 `objc_object` 类型。 `obj` 现在已经是一个 `Sark` 类型的实例对象了。当然接下来可以调用speak的方法。

- 难点二，如果能调用 `speak` ，会输出什么呢?

很多人可能会认为会输出sark相关的信息。这样答案就错误了。

正确的答案会输出

```
my name is <ViewController: 0x7ff6d9f31c50>
```

> *内存地址每次运行都不同，但是前面一定是 `ViewController。why?`*

我们把代码改变一下，打印更多的信息出来。

```
- (void)viewDidLoad {
    [super viewDidLoad];

    NSLog(@"ViewController = %@ , 地址 = %p", self, &self);

    id cls = [Sark class];
    NSLog(@"Sark class = %@ 地址 = %p", cls, &cls);

    void *obj = &cls;
    NSLog(@"Void *obj = %@ 地址 = %p", obj,&obj);

    [(__bridge id)obj speak];

    Sark *sark = [[Sark alloc]init];
    NSLog(@"Sark instance = %@ 地址 = %p",sark,&sark);

    [sark speak];

}
```

我们把对象的指针地址都打印出来。输出结果：

```
ViewController = <ViewController: 0x7fb570e2ad00> , 地址 = 0x7fff543f5aa8
Sark class = Sark 地址 = 0x7fff543f5a88
Void *obj = <Sark: 0x7fff543f5a88> 地址 = 0x7fff543f5a80

my name is <ViewController: 0x7fb570e2ad00>
```

```
Sark instance = <Sark: 0x7fb570d20b10> 地址 = 0x7fff543f5a78
my name is (null)
```

objc_msgSendSuper2 解读

```
// objc_msgSendSuper2() takes the current search class, not its superclass.
OBJC_EXPORT id objc_msgSendSuper2(struct objc_super *super, SEL op, ...)
    __OSX_AVAILABLE_STARTING(__MAC_10_6, __IPHONE_2_0);
```

objc_msgSendSuper2方法入参是一个objc_super *super。

```
/// Specifies the superclass of an instance.
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained id receiver;

    /// Specifies the particular superclass of the instance to message.
#if !defined(__cplusplus)  &&  !__OBJC2__
    /* For compatibility with old objc-runtime.h header */
    __unsafe_unretained Class class;
#else
    __unsafe_unretained Class super_class;
#endif
    /* super_class is the first class to search */
};
#endif
```

所以按viewDidLoad执行时各个变量入栈顺序从高到底为 `self`，`_cmd`，`self.class`，`self`，`obj`。

- 第一个 `self` 和第二个 `_cmd` 是隐藏参数。
- 第三个 `self.class` 和第四个 `self` 是 `[super viewDidLoad]` 方法执行时候的参数。
- 在调用 `self.name` 的时候，本质上是 `self` 指针在内存向高位地址偏移一个指针。在32位下面，一个指针是4字节=4*8bit=32bit。**（64位不一样但是思路是一样的）**
- 从打印结果我们可以看到，`obj` 就是 `cls` 的地址。在 `obj` 向上偏移 `32bit` 就到了 `0x7fff543f5aa8`，这正好是 `ViewController` 的地址。

所以输出为my name is `<ViewController: 0x7fb570e2ad00>`。

至此，`Objc` 中的对象到底是什么呢?

> *实质：`Objc` 中的对象是一个指向 `ClassObject` 地址的变量，即 `id obj = &ClassObject`，而对象的实例变量 `void *ivar = &obj + offset(N)`*

加深一下对上面这句话的理解，下面这段代码会输出什么?

```
- (void)viewDidLoad {
    [super viewDidLoad];

    NSLog(@"ViewController = %@ , 地址 = %p", self, &self);

    NSString *myName = @"halfrost";

    id cls = [Sark class];
    NSLog(@"Sark class = %@ 地址 = %p", cls, &cls);

    void *obj = &cls;
    NSLog(@"Void *obj = %@ 地址 = %p", obj,&obj);

    [(__bridge id)obj speak];

    Sark *sark = [[Sark alloc]init];
    NSLog(@"Sark instance = %@ 地址 = %p",sark,&sark);

    [sark speak];

}
ViewController = <ViewController: 0x7fff44404ab0> ,  地址  = 0x7fff56a48a78
Sark class = Sark  地址  = 0x7fff56a48a50
Void *obj = <Sark: 0x7fff56a48a50>  地址 = 0x7fff56a48a48

my name is halfrost

Sark instance = <Sark: 0x6080000233e0>  地址 = 0x7fff56a48a40
my name is (null)
```

由于加了一个字符串，结果输出就完全变了，`[(__bridge id)obj speak]`;这句话会输出`“my name is halfrost”`

> *原因还是和上面的类似。按`viewDidLoad`执行时各个变量入栈顺序从高到底为`self`，`_cmd`，`self.class`，`self`，`myName`，`obj`。`obj`往上偏移32位，就是`myName`字符串，所以输出变成了输出`myName`了。*

20. 排序题：冒泡排序，选择排序，插入排序，快速排序（二路，三路）能写出那些?

这里简单的说下几种快速排序的不同之处，随机快排，是为了解决在近似有序的情况下，时间复杂度会退化为`O(n^2)`，双路快排是为了解决快速排序在大量数据重复的情况下，时间复杂度会退化为`O(n^2)`，三路快排是在大量数据重复的情况下，对双路快排的一种优化。

- 冒泡排序

```
extension Array where Element : Comparable{
public mutating func bubbleSort() {
let count = self.count
for i in 0..<count {
for j in 0..<(count - 1 - i) {
if self[j] > self[j + 1] {
(self[j], self[j + 1]) = (self[j + 1], self[j])
}
}
}
}
}
```

- 选择排序

```
extension Array where Element : Comparable{
public mutating func selectionSort() {
    let count = self.count
    for i in 0..<count {
        var minIndex = i
        for j in (i+1)..<count {
            if self[j] < self[minIndex] {
                minIndex = j
            }
        }
        (self[i], self[minIndex]) = (self[minIndex], self[i])
    }
}
}
```

- 插入排序

```
extension Array where Element : Comparable{
public mutating func insertionSort() {
    let count = self.count
    guard count > 1 else { return }
    for i in 1..<count {
        var preIndex = i - 1
        let currentValue = self[i]
        while preIndex >= 0 && currentValue < self[preIndex] {
            self[preIndex + 1] = self[preIndex]
            preIndex -= 1
        }
        self[preIndex + 1] = currentValue
    }
}
}
```

- 快速排序

```
extension Array where Element : Comparable{
public mutating func quickSort() {
    func quickSort(left:Int, right:Int) {
        guard left < right else { return }
        var i = left + 1,j = left
        let key = self[left]
        while i <= right {
            if self[i] < key {
                j += 1
                (self[i], self[j]) = (self[j], self[i])
            }
            i += 1
        }
        (self[left], self[j]) = (self[j], self[left])
        quickSort(left: j + 1, right: right)
        quickSort(left: left, right: j - 1)
    }
    quickSort(left: 0, right: self.count - 1)
}
}
```

- 随机快排

```
extension Array where Element : Comparable{
public mutating func quickSort1() {
    func quickSort(left:Int, right:Int) {
        guard left < right else { return }
        let randomIndex = Int.random(in: left...right)
        (self[left], self[randomIndex]) = (self[randomIndex], self[left])
        var i = left + 1,j = left
        let key = self[left]
        while i <= right {
            if self[i] < key {
                j += 1
                (self[i], self[j]) = (self[j], self[i])
            }
            i += 1
        }
        (self[left], self[j]) = (self[j], self[left])
        quickSort(left: j + 1, right: right)
        quickSort(left: left, right: j - 1)
    }
    quickSort(left: 0, right: self.count - 1)
}
}
```

- 双路快排

```
extension Array where Element : Comparable{
public mutating func quickSort2() {
    func quickSort(left:Int, right:Int) {
        guard left < right else { return }
        let randomIndex = Int.random(in: left...right)
        (self[left], self[randomIndex]) = (self[randomIndex], self[left])
        var l = left + 1, r = right
        let key = self[left]
        while true {
            while l <= r && self[l] < key {
                l += 1
            }
            while l < r && key < self[r]{
                r -= 1
            }
            if l > r { break }
            (self[l], self[r]) = (self[r], self[l])
            l += 1
            r -= 1
        }
        (self[r], self[left]) = (self[left], self[r])
        quickSort(left: r + 1, right: right)
        quickSort(left: left, right: r - 1)
    }
    quickSort(left: 0, right: self.count - 1)
}
}
```

- 三路快排

```
// 三路快排
extension Array where Element : Comparable{
public mutating func quickSort3() {
    func quickSort(left:Int, right:Int) {
        guard left < right else { return }
        let randomIndex = Int.random(in: left...right)
        (self[left], self[randomIndex]) = (self[randomIndex], self[left])
        var lt = left, gt = right
        var i = left + 1
        let key = self[left]
        while i <= gt {
            if self[i] == key {
                i += 1
            }else if self[i] < key{
                (self[i], self[lt + 1]) = (self[lt + 1], self[i])
                lt += 1
                i += 1
            }else {
                (self[i], self[gt]) = (self[gt], self[i])
                gt -= 1
            }
        }
        (self[left], self[lt]) = (self[lt], self[left])
        quickSort(left: gt + 1, right: right)
        quickSort(left: left, right: lt - 1)
    }
    quickSort(left: 0, right: self.count - 1)
}
}
```