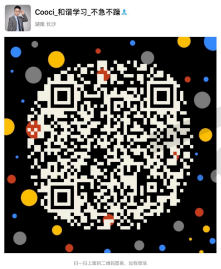
2020-逻辑教育iOS面试题集合

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*
>
> *Github 持续更新面试题*
>
> *有一起加入同步更新这个仓库的小伙伴，请微信我*



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

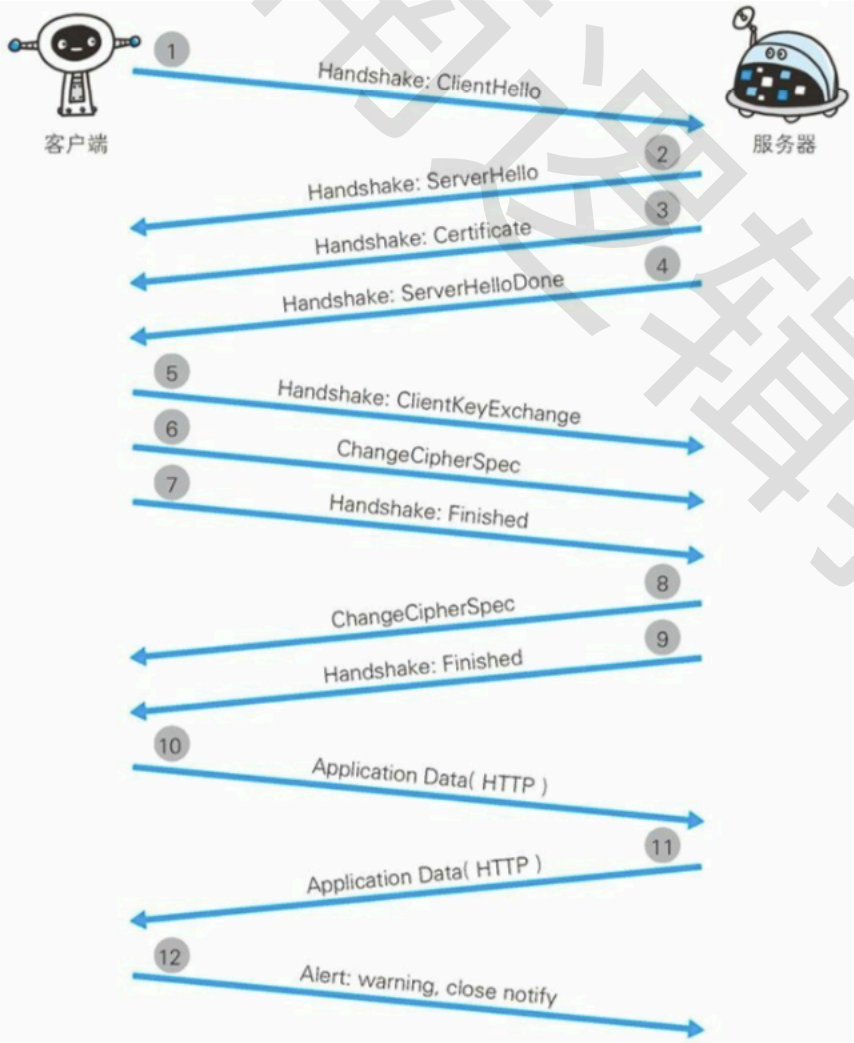
11. 对称加密和非对称加密的区别？分别有哪些算法的实现？

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：

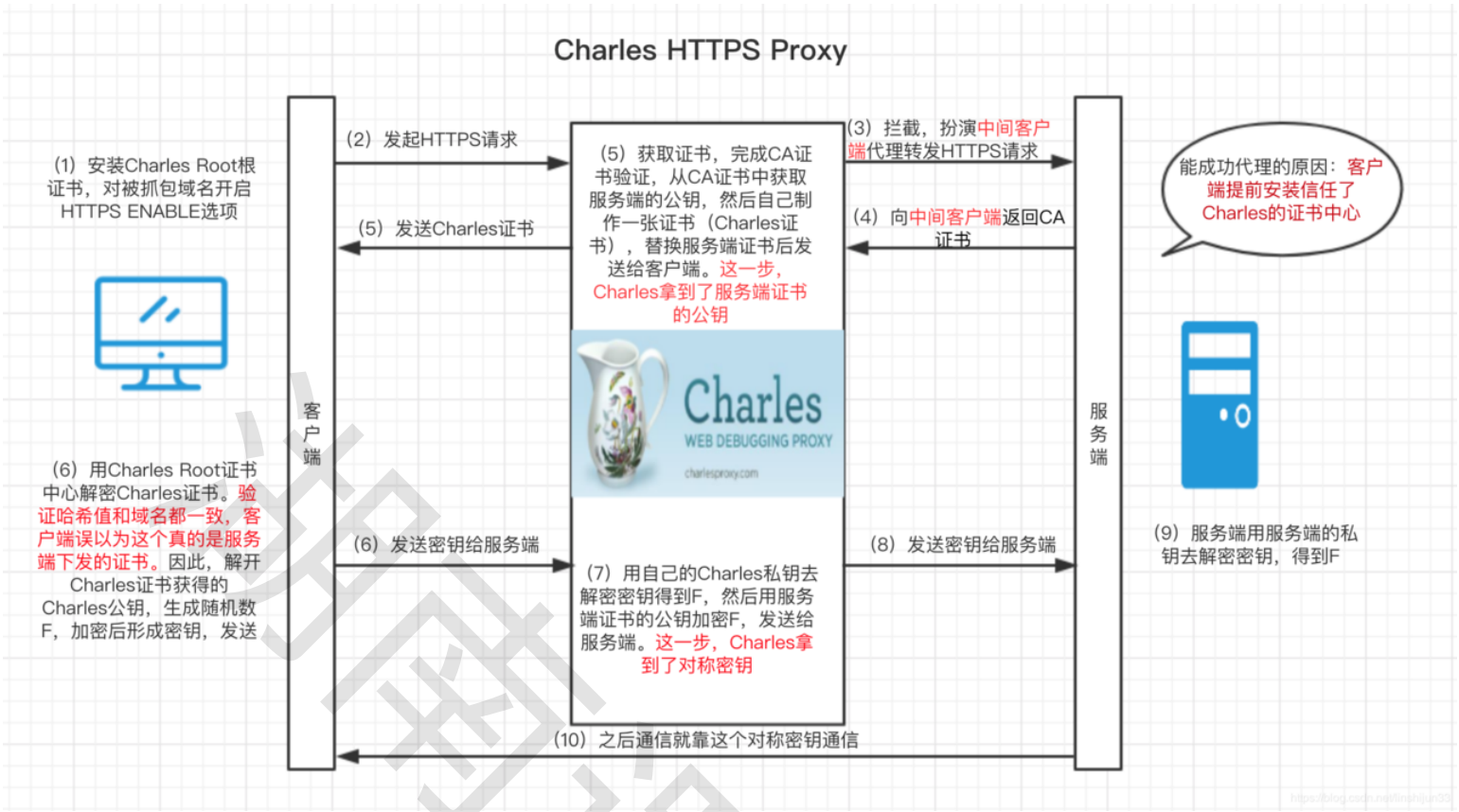
- 首先在手机上安装Charles证书
- 在代理设置中开启Enable SSL Proxying
- 之后添加需要抓取服务端的地址

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)
    }
}
```

21. iOS开发中的加密方式

iOS加密相关算法框架：`CommonCrypto`。

1:对称加密： `DES、3DES、AES`

- 加密和解密使用同一个密钥。
- 加密解密过程：
  - `明文->密钥加密->密文`，
  - `密文->密钥解密->明文`。
- 优点：算法公开、计算量少、加密速度快、加密效率高、适合大批量数据加密；
- 缺点：双方使用相同的密钥，密钥传输的过程不安全，易被破解，因此为了保密其密钥需要经常更换。

> *AES：AES又称高级加密标准，是下一代的加密算法标准，支持128、192、256位密钥的加密，加密和解密的密钥都是同一个。iOS一般使用ECB模式，16字节128位密钥。*

AES算法主要包括三个方面：**轮变化**、**圈数**和**密钥扩展**。

- 优点：高性能、高效率、灵活易用、安全级别高。
- 缺点：加密与解密的密钥相同，所以前后端利用AES进行加密的话，如何安全保存密钥就成了一个问题。

> *DES：数据加密标准，DES算法的入口参数有三个：`Key`、`Data`、`Mode`。*

- 其中Key为7个字节共56位，是DES算法的工作密钥；Data为8个字节64位，是要被加密或被解密的数据；Mode为DES的工作方式，有两种：**加密、解密**。
- 缺点：与AES相比，安全性较低。

> *3DES：3DES是DES加密算法的一种模式，它使用3条64位的密钥对数据进行三次加密。是DES向AES过渡的加密算法，是DES的一个更安全的变形。它以DES为基本模块，通过组合分组方法设计出分组加密算法。*

2.非对称加密:RSA加密

- 非对称加密算法需要成对出现的两个密钥，公开密钥(`publickey`)和私有密钥(`privatekey`)。
- **加密解密过程：**对于一个私钥，有且只有一个与之对应的公钥。生成者负责生成私钥和公钥，并保存私钥，公开公钥。

> *公钥加密，私钥解密；或者私钥数字签名，公钥验证。公钥和私钥是成对的，它们互相解密。*

- **特点：**
  - 1). **对信息保密，防止中间人攻击：**将明文通过接收人的公钥加密，传输给接收人，因为只有接收人拥有对应的私钥，别人不可能拥有或者不可能通过公钥推算出私钥，所以传输过程中无法被中间人截获。只有拥有私钥的接收人才能阅读。**此方法通常用于交换对称密钥。**
  - 2). **身份验证和防止篡改：**权限狗用自己的私钥加密一段授权明文，并将授权明文和加密后的密文，以及公钥一并发送出来，接收方只需要通过公钥将密文解密后与授权明文对比是否一致，就可以判断明文在中途是否被篡改过。**此方法用于数字签名。**
- **优点：**加密强度小，加密时间长，常用于数字签名和加密密钥、安全性非常高、解决了对称加密保存密钥的安全问题。
- **缺点：**加密解密速度远慢于对称加密，不适合大批量数据加密。

3. 哈希算法加密：`MD5加密`、`.SHA加密`、`HMAC加密`

- 哈希算法加密是通过哈希算法对数据加密，加密后的结果不可逆，即加密后不能再解密。
- **特点：**不可逆、算法公开、相同数据加密结果一致。
- **作用：**信息摘要，信息“指纹”，用来做数据识别的。如：用户密码加密、文件校验、数字签名、鉴权协议。

> *MD5加密：**对不同的数据加密的结果都是定长的32位字符。*
>
> ***.SHA加密：**安全哈希算法，主要适用于数字签名标准(`DSS`)里面定义的数字签名算法(DSA)。对于长度小于`2^64`位的消息，`SHA1`会产生一个160位的消息摘要。当接收到消息的时候，这个消息摘要可以用来验证数据的完整性。在传输的过程中，数据很可能会发生变化，那么这时候就会产生不同的消息摘要。当然除了`SHA1`还有`SHA256`以及`SHA512`等。*
>
> ***HMAC加密：**给定一个密钥，对明文加密，做两次“散列”，得到的结果还是32位字符串。*

4. Base64加密

- 一种编码方式，严格意义上来说不算加密算法。其作用就是将二进制数据编码成文本，方便网络传输。
- 用`base64`编码之后，数据长度会变大，增加了大约 1/3，但是好处是编码后的数据可以直接在邮件和网页中显示；
- 虽然`base64`可以作为加密，但是`base64`能够逆运算，非常不安全！

- base64 编码有个非常显著的特点，末尾有个‘=’号。

- **原理：**

  - 1). 将所有字符转化为ASCII码；
  - 2). 将ASCII码转化为8位二进制；
  - 3). 将二进制三位一组不足补0，共24位，再拆分成6位一组共四组；
  - 4). 统一在6位二进制前补两个0到八位；
  - 5). 将补0后的二进制转为十进制；
  - 6). 最后从Base64编码表获取十进制对应的Base64编码。

22. App安全，数字签名，App签名，重签名

因为应用实际上是一个加壳的ipa文件，但是有可能被砸壳甚至越狱手机下载的ipa包直接就是脱壳的，可以直接反编译，所以不要在`plist文件`、项目中的`静态文件`中存储关键的信息。所以敏感信息对称加密存储或者就存储到`keychain`里。而且加密密钥也要定期更换。

数字签名是通过`HASH算法`和`RSA加密`来实现的。
我们将明文数据加上**通过`RSA加密`的数据`HASH值`**一起传输给对方，对方可以解密拿出`HASH值`来进行验证。这个通过RSA加密`HASH值`数据，我们称之为数字签名。

App签名

- 1.在Mac开发机器上生成一对公钥和私钥，这里称为公钥L，私钥L(L：Local)。
- 2.苹果自己有固定的一对公钥和私钥，**私钥**在苹果后台，**公钥**在每个iOS设备上。这里称为公钥A，私钥A(A：Apple)。
- 3.把**开发机器上的公钥L**传到苹果后台，用**苹果后台的私钥A**去签名**公钥L**。得到一个包含**公钥L**以及其**签名**数据证书。
- 4.在苹果后台申请AppID，配置好**设备ID列表**和**APP可使用的权限**，再加上第③步的**证书**，组成的数据用**私钥A**签名，把数据和签名一起组成一个 `Provisioning Profile` 描述文件，下载到**本地Mac开发机器**。
- 5.在开发时，编译完一个APP后，用**本地的私钥L**对这个APP进行签名，同时把第④步得到的 `Provisioning Profile` 描述文件打包进APP里，文件名为 `embedded.mobileprovision` ，把 APP安装到手机上。
- 6.在安装时，iOS系统取得证书，通过**系统内置的公钥A**，去验证 `embedded.mobileprovision` 的**数字签名**是否正确，里面的证书签名也会再验一遍。
- 7.确保了 `embedded.mobileprovision` 里的数据都是苹果授权的以后，就可以取出里面的数据，做各种验证，包括用公钥 L 验证APP签名，验证设备 ID 是否在 ID 列表上，AppID 是否对应得上，权限开关是否跟 APP 里的 Entitlements 对应等。

23. OC数据类型

① 基本数据类型

- C语言基本数据类型（如 `short、int、float` 等）在OC中都不是对象，只是一定字节的内存空间用于存储数值，他们都不具备对象的特性，没有属性方法可以被调用。
- **OC中的基本数据类型：**
  - `NSInteger` (相当于long型整数)、
  - `NSUInteger` (相当于unsigned long型整数)、
  - `CGFloat` (在64位系统相当于double，32位系统相当于float)等。
  - 他们并不是类，只是用 `typedef` 对基本数据类型进行了重定义， `他们依然只是基本数据类型` 。
  - **枚举类型：**其本质是无符号整数。
  - **BOOL类型：**是宏定义，OC底层是使用signed char来代表BOOL。

② 指针数据类型

指针数据类型包括： `类class` 、 `id` 。

- **类class：** `NSString` 、 `NSSet` 、 `NSArray` 、 `NSMutableArray` 、 `NSDictionary` 、 `NSMutableDictionary` 、 `NSValue` 、 `NSNumber(继承NSValue)` 等，都是class，创建后便是对象，继承**NSObject**。

OC中提供了 `NSValue` 、 `NSNumber` 来封装C语言的基本类型，这样我们就可以让他们具有面向对象的特征了。

- **id：** `id` 是指向Objective-C对象的指针，等价于C语言中的 `void*` ，可以映射任何对象指针指向他，或者映射它指向其他的对象。常见的id类型就是类的delegate属性。

集合NSSet和数组NSArray区别：

- 都是存储不同的对象的地址；
- 但是NSArray是有序的集合，NSSet是无序的集合，它们俩可以互相转换。
- NSSet会自动删除重复元素。
- 集合是一种哈希表，运用散列算法，查找集合中的元素比数组速度更快，但是它没有顺序。

③ 构造类型

构造类型包括：结构体、联合体

- **结构体：** `struct`，将多个基本数据类型的变量组合成一个整体。结构体中访问内部成员用点运算符访问。
- **联合体(共用体)：** `union`，有些类似结构体 `struct` 的一种数据结构，联合体(`union`)和结构体(`struct`)同样可以包含很多种数据类型和变量。

结构体和联合体的区别：

- 结构体(`struct`)中所有变量是“共存”的，同一时刻每个成员都有值，其 `sizeof` 为所以成员的和。

  优点：是“有容乃大”，全面；

  缺点：是struct内存空间的分配是粗放的，不管用不用，全

分配，会造成内存浪费。

- 联合体(`union`)中各变量是“互斥”的，同一时刻只有一个成员有值，其 `sizeof` 为最长成员的 `sizeof` 。

  优点：是内存使用更为精细灵活，也节省了内存空间。

  缺点：就是不够“包容”，修改其中一个成员时会覆盖原来的成员值；

24. property和属性修饰符

@property的本质是 `ivar(实例变量)` + `setter` + `getter` .

我们每次增加一个属性时内部都做了什么：

- 1.系统都会在 `ivar_list` 中添加一个成员变量的描述；
- 2.在 `method_list` 中增加 `setter` 与 `getter` 方法的描述；
- 3.在属性列表中增加一个属性的描述；
- 4.然后计算该属性在对象中的偏移量；
- 5.给出 `setter` 与 `getter` 方法对应的实现，在 `setter` 方法中从偏移量的位置开始赋值，在 `getter` 方法中从偏移量开始取值，为了能够读取正确字节数，系统对象偏移量的指针类型进行了类型强转。

修饰符：

- **MRC下：** `assign、retain、copy、readwrite、readonly、nonatomic、atomic` 等。
- **ARC下：** `assign、strong、weak、copy、readwrite、readonly、nonatomic、atomic、nonnull、nullable、null_resettable、_Null_unspecified` 等。

下面分别解释

- `assign`：用于基本数据类型，不更改引用计数。如果修饰对象(对象在堆需手动释放内存，基本数据类型在栈系统自动释放内存)，会导致对象释放后指针不置为nil 出现野指针。
- `retain`：和strong一样，释放旧对象，传入的新对象引用计数+1；在MRC中和release成对出现。
- `strong`：在ARC中使用，告诉系统把这个对象保留在堆上，直到没有指针指向，并且ARC下不需要担心引用计数问题，系统会自动释放。
- `weak`：在被强引用之前，尽可能的保留，不改变引用计数；weak引用是弱引用，你并没有持有它；它本质上是分配一个不被持有的属性，当引用者被销毁(dealloc)时，weak引用的指针会自动被置为nil。**可以避免循环引用。**
- `copy`：一般用来修饰不可变类型属性字段，如： `NSString` 、 `NSArray` 、 `NSDictionary` 等。用copy修饰可以防止本对象属性受外界影响，在 `NSMutableString` 赋值给 `NSString` 时，修改前者 会导致 后者的值跟着变化。还有 `block` 也经常使用 copy 修饰符，但是其实在ARC中编译器会自动对block进行copy操作，和strong的效果是一样的。但是在MRC中方法内部的block是在栈区，使用copy可以把它放到堆区。
- `readwrite`：可以读、写；编译器会自动生成setter/getter方法。
- `readonly`：只读；会告诉编译器不用自动生成setter方法。属性不能被赋值。

- `nonatomic`：非原子性访问。用nonatomic意味着可以多线程访问变量，会导致读写线程不安全。但是会提高执行性能。
- `atomic`：原子性访问。编译器会自动生成互斥锁，对 setter 和 getter 方法进行加锁来保证属性的 赋值和取值 原子性操作是线程安全的，但不包括可变属性的操作和访问。比如我们对数组进行操作，给数组添加对象或者移除对象，是不在atomic的负责范围之内的，所以给被atomic修饰的数组添加对象或者移除对象是没办法保证线程安全的。**原子性访问的缺点是会消耗性能导致执行效率慢。**
- `nonnull`：设置属性或方法参数不能为空，专门用来修饰指针的，不能用于基本数据类型。
- `nullable`：设置属性或方法参数可以为空。
- `null_resettable`：设置属性，get方法不能返回为空，set方法可以赋值为空。
- `_Null_unspecified`：设置属性或方法参数不确定是否为空。

> *后四个属性应该主要就是为了提高开发规范，提示使用的人应该传什么样的值，如果违反了对规范值的要求，就会有警告。*

weak修饰的对象释放则自动被置为nil的实现原理：

`Runtime` 维护了一个 `weak表`，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` 指针的地址数组（这个地址的值是所指对象的地址）。

weak 的实现原理可以概括一下三步：

- 1、初始化时：`runtime` 会调用 `objc_initWeak` 函数，初始化一个新的weak指针指向对象的地址。
- 2、添加引用时：`objc_initWeak` 函数会调用 `objc_storeWeak()` 函数，`objc_storeWeak()` 的作用是更新指针指向，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 `nil`，最后把这个entry从weak表中删除，最后清理对象的记录。

25. 成员变量 `ivar` 和属性 `property` 的区别，以及不同关键字的作用

成员变量：成员变量的默认修饰符是 `@protected`、不会自动生成set和get方法，需要手动实现、不能使用点语法调用，因为没有set和get方法，只能使用 `->`。

属性：属性会默认生成带下划线的成员变量和 `setter/getter` 方法、可以用点语法调用，实际调用的是set和get方法。

注意：分类中添加的属性是不会自动生成 `setter/getter` 方法的，必须要手动添加。

实例变量：class类进行实例化出来的对象为实例对象

关键字作用：

- **访问范围关键字**
- `@public`：声明公共实例变量，在任何地方都能直接访问对象的成员变量。
- `@private`：声明私有实例变量，只能在当前类的对象方法中直接访问，子类要访问需要调用父类的get/set方法。
- `@protected`：可以在当前类及其子类对象方法中直接访问(系统默认)。
- `@package`：在同一个包下就可以直接访问，比如说在同一个框架。
- **关键字**
- `@property`：声明属性，自动生成一个以下划线开头的成员变量_propertyName(默认用@private修饰)、属性setter、getter方法的声明、属性setter、getter方法的实现。**注意：**在 `协议@protocol` 中只会生成getter和setter方法的声明，所以不仅需要手动实现getter和setter方法还需要手动定义变量。
- `@sythesize`：修改@property自动生成的_propertyName成员变量名，`@synthesize propertyName = newName;`。
- `@dynamic`：告诉编译器：属性的 setter 与 getter 方法由用户自己实现，不自动生成。**谨慎使用：**如果对属性赋值取值可以编译成功，但运行会造成程序崩溃，这就是常说的动态绑定。
- `@interface`：声明类
- `@implementation`：类的实现
- `@selecter`：创建一个SEL，类成员指针
- `@protocol`：声明协议

- `@autoreleasepool`： ARC中的自动释放池
- `@end`： 类结束

26. 类簇

> ***类簇**是Foundation框架中广泛使用的设计模式。**类簇在公共抽象超类下对多个私有的具体子类进行分组**。以这种方式对类进行分组简化了面向对象框架的公共可见体系结构，而不会降低其功能丰富度。**类簇是基于抽象工厂设计模式的**。*

常见的类簇有 `NSString` 、 `NSArray` 、 `NSDictionary` 等。
以数组为例：不管创建的是可变还是不可变的数组，在 `alloc` 之后得到的类都是 `__NSPlaceholderArray` 。而当我们 `init` 一个不可变的空数组之后，得到的是 `__NSArray0` ；如果有且只有一个元素，那就是 `__NSSingleObjectArrayI` ；有多个元素的，叫做 `__NSArrayI` ； `init` 出来一个可变数组的话，都是 `__NSArrayM` 。

优点：

- 可以将抽象基类背后的复杂细节隐藏起来。
- 程序员不会需要记住各种创建对象的具体类实现，简化了开发成本，提高了开发效率。
- 便于进行封装和组件化。
- 减少了 if-else 这样缺乏扩展性的代码。
- 增加新功能支持不影响其他代码。

缺点：

- 已有的类簇非常不好扩展。

我们运用类簇的场景：

- 1. 出现 bug 时，可以通过崩溃报告中的类簇关键字，快速定位 bug 位置。
- 1. 在实现一些固定且并不需要经常修改的事物时，可以高效的选择类簇去实现。例：
  - ▪ 针对不同版本，不同机型往往需要不同的设置，这时可以选择使用类簇。
  - ▪ app 的设置页面这种并不需要经常修改的页面，可以使用类簇去创建大量重复的布局代码。

27. 设计模式

创建型模式：

- **单例模式：**在整个应用程序中，共享一份资源。保证在程序运行过程中，一个类只有一个实例，而且该实例只提供一个全局访问点供外界访问，从而方便控制实例个数，节约系统资源。

  优点是：提供了对唯一实例的受控访问、可扩展、避免频繁创建销毁对象影响性能。

  缺点是：延长了声明周期，一直存在占用内存。如果两个单例循环依赖会造成死锁，所以尽量不去产生单例间的依赖关系。
- **工厂方法模式：**通过类继承创建抽象产品，创建一种产品，子类化创建者并重载工厂方法以创建新产品。
- **抽象工厂模式：**通过对象组合创建抽象产品，可以创建多系列产品，必须修改父类的接口才能支持新的产品。

结构型模式：

- **代理模式：**代理用来处理事件的监听和参数传递。 `@required` 修饰必须实现这个协议方法方法， `@optional` 修饰是可选实现。使用方法时最好先判断方法是否实现 `respondsToSelector:` ，避免找不到方法而崩溃。

> ***delegate和block、Notification对比优缺点：**delegate和block是一对一通信、block比delegate更加简洁清晰，但是如果通信事件较多时delegate运行成本较低且不易造成循环引用；通知适合一对多通信，代码清晰简单，但问题查找溯源会比较困难，并且注册通知要注意在合适的时间移除，避免对野指针发送消息引起崩溃（**注意：**iOS9之后已经做了弱引用处理不需要移除了，之前版本使用不安全引用__unsafe_unretained是为了兼容旧版本）。*

- **类簇：**见上边 5. 类簇
- **装饰模式：**在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。如：**分类**。
- **享元模式：使用共享物件，减少同一类对象的大量创建。如：**UITableviewCell复用。

行为型模式：

- **观察者模式：**其本质上是一种 `发布-订阅` 模型，用来消除具有不同行为的对象之间的耦合，通过这一模式，不同对象可以协同工作。如：**KVO**。
- **命令模式：**是一种将方法调用封装为对象的设计模式，在iOS中具体实现为**NSInvocation**。下边为 `NSInvocation` 的实现代码。

```
- (void)viewDidLoad {
    NSMethodSignature *signature = [ViewController instanceMethodSignatureForSelector:@selector(sendMessageWithPhone:WithName:)]; // 方
    NSInvocation *invocation = [NSInvocation invocationWithMethodSignature:signature];
    invocation.target = self; // 目标：接收消息的对象
    invocation.selector = @selector(sendMessageWithPhone:WithName:); // 选择器：被发送的消息，方法必须和签名中的方法一致。

    NSString *phone = @"13512345678";
    // 注意：设置参数的索引时不能从0开始，因为0已经被self占用，1已经被_cmd占用
    [invocation setArgument:&phone atIndex:2]; // 参数：可以添加任意数量的参数。
    NSString *name = @"Dezi";
    [invocation setArgument:&name atIndex:3];
    /*
     注：调用invocation的invoke方法，就代表需要执行NSInvocation对象中指定对象的指定方法，并且传递指定的参数
     */
    [invocation invoke];
}
- (void)sendMessageWithPhone:(NSString*)phone WithName:(NSString*)name {
    NSLog(@"电话号=%@，姓名=%@",phone, name);
}
// 电话号=13512345678，姓名Dezi
```

- MVC和MVVM算是架构。

28. 架构设计

**MVC：

- **M** 是数据模型 `Model`，负责处理数据，以及数据改变时发出通知(`Notification、KVO`)， `Model` 和 `View` 不能直接进行通信，这样会违背MVC设计模式；
- **V** 是视图 `View`，用来展示界面，和用户进行交互，为了解耦合一般不会直接持有 或者 操作数据层中的数据模型(可以通过 `action-target` 、 `delegate` 、 `block` 等方式解耦)；
- **C** 是控制器 `Controller` 用来调节 `Model` 和 `View` 之间的交互，可以直接与Model还有View进行通信，操作Model进行数据更新，刷新View。

  优点： `View` 、 `Model` 低耦合、高复用、容易维护。

  缺点： `Controller` 的代码过于臃肿，如果 `View` 与 `Model` 直接交互会导致 `View` 和 `Model` 之间的耦合性比较大、网络逻辑会加重 `Controller` 的臃肿。

MVVM： `Model – View – ViewModel`

- **MVVM**衍生于**MVC**，是MVC的一种演进，促进了UI代码和业务逻辑的分离，抽取 `Controller` 中的展示逻辑放到 `ViewModel` 里边。
- **M：** 数据模型 `Model` 。
- **V：** 就是 `View` 和 `Controller` 联系到一起，视为是一个组件 `View` 。View和Controller都不能直接引用模型Model，可以引用视图模型ViewModel。ViewController 尽量不涉及业务逻辑，让 ViewModel 去做这些事情。ViewController 只是一个中间人，负责接收 View 的事件、调用 ViewModel 的方法、响应 ViewModel 的变化。
- **VM：** `ViewModel` 负责封装业务逻辑、网络处理和数据缓存。使用ViewModel会轻微的增加代码量，但是总体上减少了代码的复杂性。 `ViewModel` 之间可以有依赖。

注意事项：

- `View` 引用 `ViewModel` ，但反过来不行，因为如果VM跟V产生了耦合，不方便复用。即不要在 `viewModel` 中引入 `#import UIKit.h` ，任何视图本身的引用都不应该放在 `viewModel` 中 (**注意：基本要求，必须满足**)。
- `ViewModel` 可以引用 `Model` ，但反过来不行。

优点：

低耦合、可复用、数据流向清晰、而且兼容MVC，便于代码的移植、并且ViewModel可以拆出来独立开发、方便测试。

缺点：
类会增多、 `ViewModel` 会越来越庞大、调用复杂度增加、双向绑定数据会导致问题调试变得困难。

总结：

- **MVVM**其实是MVC的变种。 `MVVM` 只是帮 `MVC` 中的 `Controller` 瘦身，把一些逻辑代码和网络请求分离出去。不让Controller处理更多的东西，不会变得臃肿， `MVVM` 和 `MVC` 可以根据实际需求进行灵活选择。
- **MVVM** 在使用当中，通常还会利用双向绑定技术，使得 `Model` 变化时， `ViewModel` 会自动更新，而 `ViewModel` 变化时， `View` 也会自动变化。OC中可以用**RAC(ReactiveCocoa)**函数响应式框架来实现响应式编程。

29. ReactiveCocoa的使用及优缺点

> *ReactiveCocoa简称RAC，是函数响应式编程框架，因为它具有函数式编程和响应式编程的特性。*

- 由于该框架的编程思想，使得它具有相当魅惑人心的功能，它能实现传统设计模式和事件监听所能实现的功能，比如**KVO、通知、block回调、action、协议**等等，它的全面性并不是它最为优越的特色，RAC最值得炫耀的是它提供了统一的消息传递机制，这种机制使得它的代码更加的简洁，同一功能代码块更少，这正是符合了我们编程的思想：高聚合、低耦合，它非常适合MVVM设计模式的开发。
- 不过它也并不能完全取代传统的编码方式，在多人开发和代码维护方面，RAC还是有着一些让人头痛的问题。

优点：使用灵活方便、代码简洁、逻辑清晰

缺点：维护成本较高、问题溯源困难

使用：

RAC的统一消息传递机制，其所以动作都离不开 `信号(sigal)` 。

- 1). **信号的创建、发送、接收**

```
// 创建  此时为冷信号，并不会被触发
RACSignal *signal = [RACSignal createSignal:^RACDisposable *(id<RACSubscribersubscriber) {
    // 发送信号
    [subscriber sendNext:@"oh my god"];
    // 回收资源。注意：手动创建一个signal一定要记得回收资源，不然程序会崩溃
    return [RACDisposable disposableWithBlock:^{
        NSLog(@"信号发送完成");
    }];
}];
// 订阅信号后才会变为热信号，可以被触发
[signal subscribeNext:^(id x) {
    NSLog(@"singalContent:%@", x);
}];
```

- 2). **RAC的ControlEvents**
  这个方法可以简单的实现监听操作，并且逻辑在其后的 `block` 中处理，而且也能添加手势并进行监听。

```
[[self.textField rac_signalForControlEvents:UIControlEventEditingDidBegin] subscribeNext:^(id x) {
    NSLog(@"%@", x);
}];

UITapGestureRecognizer *tap = [UITapGestureRecognizer new];
[[tap rac_gestureSignal] subscribeNext:^(id x) {
    NSLog(@"three:%@", x);
}];
[self.view addGestureRecognizer:tap];
```

- 3). **RAC的KVO**

```
[[self.textField rac_valuesAndChangesForKeyPath:@"text" options:NSKeyValueObservingOptionNew observer:self] subscribeNext:^(id x) {
    NSLog(@"%@", x);
}];
```

- 4). **RAC的通知**

```
[[[NSNotificationCenter defaultCenter] rac_addObserverForName:UIKeyboardDidShowNotification object:nil] subscribeNext:^(id x) {
    NSLog(@"键盘弹起");
}];
```

- 5). **RAC的协议**

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 代理
    self.textField.delegate = self;
    [[self rac_signalForSelector:@selector(textFieldDidBeginEditing:) fromProtocol:@protocol(UITextFieldDelegate)] subscribeNext:^(id x)
        NSLog(@"打印点击信息:%@", x);
    }];
}
- (void)textFieldDidBeginEditing:(UITextField *)textField {
    NSLog(@"开始编辑了");
}
```

- 6). **RAC遍历数组和字典**

相当于枚举遍历，但是效率相比更高

```
NSArray *arr = @[@"1", @"2", @"3", @"4", @"5"];
[arr.rac_sequence.signal subscribeNext:^(id x) {
    NSLog(@"arr : %@", x);
}];
NSDictionary *dic = @{@"name":@"yangBo", @"age":@"19"};
[dic.rac_sequence.signal subscribeNext:^(id x) {
    NSLog(@"dic : %@", x);
}];
```

- 7). **RAC信号处理（map、filter、combine）**

① 对信号不做处理

```
[[self.textField rac_textSignal] subscribeNext:^(id x) {
    NSLog(@"doNothing:%@", x);
}];
```

② 对信号进行过滤（filter）
可以对信号进行条件判断是否处理。

```
[[[self.textField rac_textSignal] filter:^BOOL(NSString* value) {
    if (value.length 3) {
        return YES;
    }
    return NO;
}] subscribeNext:^(id x) {
    NSLog(@"filter:%@", x);
}];
```

③ 对信号进行映射（map）
映射也可以理解为**转换**，第一个block返回的是id类型，如果返回 `"map now"` ，就相当于信号转换，第二个block打印的值就是你return的值 `"map now"` 。

```
[[[self.textField rac_textSignal] map:^id(NSString* value) {
    if (value.length 3) {
        return @"map now";
    }
    return value;
}] subscribeNext:^(id x) {
    NSLog(@"map:%@", x);
```

```
}];
```

④ 信号的联合（combine）

```
// 创建需要联合的信号
RACSignal *firstCombineSignal = [self.textField rac_textSignal];
RACSignal *secondeCombineSignal = [tap rac_gestureSignal];
// 信号联合处理返回self.label的背景色
RAC(self.label, backgroundColor) = [RACSignal combineLatest:@[firstCombineSignal, secondeCombineSignal] reduce:^id(NSString *text, UI
TapGestureRecognizer * tap){
    // 这里进行信号逻辑判断和处理
    if (text.length == 3 && tap.state == UIGestureRecognizerStateEnded) {
        return [UIColor redColor];
    }
    return [UIColor cyanColor];
}];
```

⑤ 信号关联

```
RAC(self.label, text) = [self.textField rac_textSignal];
```

30. 类的继承，类能否多继承，协议能不能做继承

- OC的类不支持多继承只支持单继承。
- 协议可以实现多继承，遵循多个协议即可。
- 消息的转发也可以实现多继承，但并不建议，维护成本高。

继承和类别在实现中有何区别?

- `category` 可以在不获悉、不改变原来代码的情况下往里面添加并且只能添加方法不能删除修改。 如果类别和原来类中的方法产生名称冲突，则类别将覆盖原来的方法，因为类别具有更高的优先级。类别不会影响到其他类与原有类的关系。
- 类别主要有3个作用：
  - (1)将类的实现分散到多个不同文件或多个不同框架中。
  - (2)创建对私有方法的前向引用。
  - (3)向对象添加非正式协议。
- 继承可以增加，修改或者删除方法，并且可以增加属性。

31. 分类(category)和类扩展(extension)的区别

- 1). 分类实现原理

  `Category` 编译之后的底层结构是 `struct category_t` ，里面存储着分类的对象方法、类方法、属性、协议信息。在程序运行的时候， `runtime` 会将 `Category` 的数据，合并到类信息中（类对象、元类对象中）。

- 2). Category和Extension的区别是什么?
  - 类扩展可以为类添加私有变量和私有方法，在类的源文件中书写，不能被子类继承，类扩展在编译的时候，它的数据就已经包含在类信息中。
  - 分类可以为类添加方法并且可以被子类继承，因为分类是运行时才会将数据合并到类信息中。但是分类不能直接添加属性，需要借助运行时关联对象。
- 3). 分类为啥不能添加成员变量?

```
struct _category_t {
    const char *name;
    struct _class_t *cls;
    const struct _method_list_t *instance_methods;
    const struct _method_list_t *class_methods;
    const struct _protocol_list_t *protocols;
```

```
    const struct _prop_list_t *properties;
};
```

- 从结构体可以知道，有 `属性列表` ，所以分类可以声明属性，但是分类只会生成该属性对应的 `get和set的声明` ，没有去实现该方法。
- 结构体没有成员变量列表，所以不能声明成员变量。

32. 如何实现week

weak：该属性定义了一种非拥有关系。为属性设置新值时，设置方法既不持有新值，也不释放旧值。

weak实现原理：

- 1. 当一个对象被weak指针指向时，这个weak指针会以对象为key，存储到 `sideTable类` 的 `weak_table` 散列表上对应的一个weak指针数组里面。
- 1. 当一个对象的 `dealloc` 方法被调用时， `Runtime` 会以 `obj` 为 `key` ，从 `sideTable` 的 `weak_table` 散列表中，找出对应的weak指针列表，然后将里面的weak指针逐个置为 `nil` 。

`key` 是**weak指向的对象内存地址**， `value` 是**所有指向该对象的weak指针表**。

33. 字典注意事项：setvalue和setobject的区别

- `setObject:ForKey` ： 是 `NSMutableDictionary` 特有的。
- `setValue:ForKey` ： 是 `KVC` 的主要方法。
- `setobject` 中的 `key` 和 `value` 可以为nil以外的任何对象。
- `setValue` 中的key只能为字符串，value可以为 `nil` 也可以为空对象 `[NSNull null]` 以及全部对象

34. 多线程和锁

进程：是资源分配的基本单位，它是程序执行时的一个实例，在程序运行时创建。

线程：是程序执行的最小单位，是进程的一个执行流，一个进程由多个线程组成。

多线程：

一个进程中并发执行多个线程，叫做**多线程**。在一个时间片内，CPU只能**处理一个线程中的一个任务**，对于一个单核CPU来说，在不同的时间片来执行不同线程中的任务，就形成了多个任务在同时执行的“假象”。

多线程的几种方式：

- 1. `pthread` ： 即 `POSIX Thread` ，缩写称为 `pthread` ，是线程的POSIX标准，**是一套通用的多线程API，可以**在 `Unix/Linux/Windows` 等平台**跨平台使用**。iOS中基本不使用。
- 1. `NSThread` ： **苹果封装的面向对象的线程类，可以直接操作线程**，比起 `GCD` ， `NSThread` 效率更高，**由程序员自行创建**，当线程中的任务执行完毕后，线程会自动退出，**程序员也可手动管理线程的生命周期**。使用频率较低。
- 1. `GCD` ： 全称 `Grand Central Dispatch` ，**由C语言实现，是苹果为多核并行运算提出的解决方案**，CGD会自动利用更多的CPU内核，**自动管理线程的生命周期**，程序员只需要告诉GCD需要执行的任务，**无需编写任何管理线程的代码**。GCD也是iOS使用频率最高的多线程技术。
- 1. `NSOperation` ： **基于GCD封装的面向对象的多线程技术，常配合NSOperationQueue使用**，使用频率较高。

GCD和NSOperation区别

- 1. `GCD` 仅仅支持 `先进先出FIFO队列` ，不支持异步操作之间的依赖关系设置。而 `NSOperation` 中的队列可以被重新设置优先级，从而实现不同操作的执行顺序调整。
- 1. `NSOperation` 支持 `KVO` ，可以观察任务的执行状态。
- 1. `GCD` 更接近底层， `GCD` 在追求性能的底层操作来说，是速度最快的。
- 1. 从异步操作之间的事务性，顺序行，依赖关系。 `GCD` 需要自己写更多的代码来实现，而 `NSOperation` 已经内建了这些支持。
- 1. 如果异步操作的过程需要更多的被交互和UI呈现出来， `NSOperation` 更好。底层代码中，任务之间不太互相依赖，而需要更高的并发能力， `GCD` 则更有优势。

线程池原理

- 使用线程执行任务的时候，需要到线程池中去取线程进行任务分配。
- 首先判断线程池大小是否小于核心线程池大小，如果小于的话，创建新的线程执行任务；
- 如果当前小城池大小**大于**了核心线程池大小，然后开始判断工作队列是否已满，如果没满，将任务提交到工作队列。

- 如果工作队列已满，判断线程池的线程是否都在工作，如果有空闲线程没有在工作，就交给它去执行任务。
- 如果线程池中的线程都在工作，那么就交给饱和策略去执行。

饱和策略分为下面四种：

- `AbortPolicy` 直接抛出 `RejectedExecutionExeception` 异常来阻止系统正常运行；
- `CallerRunsPolicy` 将任务回退到调用者；
- `DisOldestPolicy` 丢掉等待最久的任务‘；
- `DisCardPolicy` 直接丢弃任务。

线程间通讯

直接同步或者异步向任务队列添加任务。

通过NSPort端口的形式进行发送消息，实现不同的线程间的通信。使用的时候注意需要将NSPort加入的线程的RunLoop中去。

- **直接消息传递：**通过 `performSelector` 的一系列方法，可以实现由某一线程指定在另外的线程上执行任务。因为任务的执行上下文是目标线程，这种方式发送的消息将会自动的被序列化。
- **全局变量、共享内存块和对象：**在两个线程之间传递信息的另一种简单方法是使用全局变量，共享对象或共享内存块。尽管共享变量既快速又简单，但是它们比直接消息传递更脆弱。必须使用锁或其他同步机制仔细保护共享变量，以确保代码的正确性。 否则可能会导致竞争状况，数据损坏或崩溃。
- **条件执行：**条件是一种同步工具，可用于控制线程何时执行代码的特定部分。您可以将条件视为关守，让线程仅在满足指定条件时运行。
- **Runloop sources：**一个自定义的 `Runloop source配置` 可以让一个线程上收到特定的应用程序消息。由于 `Runloop source` 是**事件驱动**的，因此在无事可做时，线程会自动进入睡眠状态，从而提高了线程的效率。
- **Ports and sockets：**基于端口的通信是在两个线程之间进行通信的一种更为复杂的方法，但它也是一种非常可靠的技术。更重要的是，端口和套接字可用于与外部实体（例如其他进程和服务）进行通信。为了提高效率，使用 `Runloop source` 来实现端口，因此当端口上没有数据等待时，线程将进入睡眠状态。
- **消息队列：**传统的多处理服务定义了 `先进先出（FIFO）队列` 抽象，用于管理传入和传出数据。尽管消息队列既简单又方便，但是它们不如其他一些通信技术高效。
- **Cocoa 分布式对象：**分布式对象是一种 Cocoa 技术，可提供基于端口的通信的高级实现。尽管可以将这种技术用于线程间通信，但是强烈建议不要这样做，因为它会产生大量开销。分布式对象更适合与其他进程进行通信，尽管在这些进程之间进行事务的开销也很高。

线程安全就是在同一时刻，对同一个数据操作的线程只有一个。这时就用到了锁。

- **锁：**是保证线程安全的同步工具。每一个线程在访问数据资源之前，要先获取(acquire)锁，然后在访问结束之后释放(release)锁。如果锁已经被占用，其它要获取锁的线程会等待，直到锁重新可用。
- iOS中的锁分为**互斥锁**、**自旋锁**、**信号量**这三种。
  互斥锁：就是在多线程编程中，防止两条线程同时对同一公共资源(比如全局变量)进行读写。

`@synchronized` ： 是递归锁

- 调用 `synchronzied` 的每个对象， `runtime` 都会为其分配一个递归锁并存储在哈希表中。
- 如果在 `synchronzied` 内部对象被释放或为nil，会执行类似 `objc_sync_nil` 的空方法。
- 注意不要像 `synchronzied` 传入 `nil` ，这将会从代码中移走线程安全。

`NSLock:` 遵循 NSLocking 协议， `lock` 方法是加锁， `unlock` 是解锁， `tryLock` 是尝试加锁，如果失败的话返回 NO， `lockBeforeDate` : 是在指定Date之前尝试加锁，如果在指定时间之前都不能加锁，则返回NO。注意不能多次调用lock方法，会造成死锁。

自旋锁：线程会反复检查锁变量是否可用，线程在这一过程中保持执行，是一种忙等待，一旦获取了自旋锁，线程会一直保持该锁，直至显式释放自旋锁，自旋锁避免了进程上下文的调度开销，因此对于线程只会阻塞很短时间的场合是有效的。 `atomic` 就是通过个set和get方法添加一个自旋锁。

信号量： `dispatch_semaphore`

使用：

- 1). 通过 `dispatch_semaphore_create(value)` 创建一个信号量，初始为1。
- 2). 等待信号量 `dispatch_semaphore_wait` ，可以理解为lock加锁，会使信号量-1。

- 3). 发送信号量 `dispatch_semaphore_signal`，可以理解为 unlock解锁，会使 `signal` 信号量+1。

互斥锁是 `dispatch_semaphore` 在取值0/1时的特例。信号量可以有更多的取值空间，用来实现更加复杂的同步，而不仅仅是线程间互斥。

35. 通知，能不能跨线程

不能跨线程，在哪个线程发送的通知，就会在哪个线程接收。所以需要手动切换到主线程更新UI。

- **测试过程：**发出通知的线程决定接收通知处理代码线程

主线程发通知 — **子线程**监听通知：接收通知代码在主线程处理

子线程发通知 — **主线程**监听通知：接收通知代码在子线程处理

36. 网络TCP协议，三次握手

网络相关面试题

`TCP` 是传输控制协议，具有面向连接、可靠传输、点到点通信、全双工服务等特点，TCP侧重可靠传输。
`UDP` 是用户数据报协议，具有非连接的、不可靠的、点到多点的通信等特点，UDP侧重快速传输。

TCP/IP协议： `TCP/IP` 不是一个协议，而是一个协议簇的统称。通常使用的网络是在TCP/IP协议簇的基础上运行的。**应用层**的 `HTTP` 、 `DNS` 、 `FTP` ，**传输层**的 `TCP` (传输控制协议)、 `UDP` (用户数据报协议)，**网络层**的 `IP` 等等都属于它内部的一个子集。

TCP报文重点字段：
`序号` ： Seq序号，占32位，用来标识从TCP源端向目的端发送的字节流，发起方发送数据时对此进行标记。
`确认序号` ： Ack序号，占32位，只有ACK标志位为1时，确认序号字段才有效，Ack=Seq+1。
`标志位` ： 共6个，即URG、ACK、PSH、RST、SYN、FIN等，具体含义如下：

- (A) `URG` ： 紧急指针（urgent pointer）有效。
- (B) `ACK` ： 确认序号有效。
- (C) `PSH` ： 接收方应该尽快将这个报文交给应用层。
- (D) `RST` ： 重置连接。
- (E) `SYN` ： 发起一个新连接。
- (F) `FIN` ： 释放一个连接。

TCP是面向连接的协议，建立连接需要经过 `三次握手` ：

- 1.首先客户端将 `标志位SYN置为1` ，然后随机产生一个 `序号seq=J` ，将数据包发送给服务端，客户端进入 `SYN_SENT` 状态，等待服务端确认。
- 2.服务器收到数据包后由 `标志位SYN=1` 知道Client请求建立连接，Server将 `标志位SYN和ACK都置为1` ， `确认序号ack=J+1` ，随机产生一个 `序号seq=K` ，并将该数据包发送给服务端确认连接请求，服务端进入 `SYN_RCVD` 状态。
- 3.客户端收到确认后，检查 `确认序号ack是否为J+1` ， `标志位ACK是否为1` ，确认正确后将 `标志位ACK=1` 和 `确认序号ack=K+1` 数据包发送给服务端，服务端检查 `标志位ACK为1` 、 `确认序号ack=K+1` 成功后，客户端和服务端进入 `ESTABLISHED` (建立连接)状态，完成三次握手，两端可以进行数据传输。

TCP断开连接需要经过 `四次挥手` ：

- 1.首先客户端向服务端发送 `释放连接的标志位 FIN` ，客户端进入 `FIN_WAIT_1` 状态，等待服务端确认。
- 2.服务端收到 `标志位FIN` 后发送 `确认标志位ACK` 给客户端，确认序号为 `收到序号+1` (与SYN相同，一个FIN占用一个序号)，服务端进入 `CLOSE_WAIT` 状态。
- 3.服务端发送自己的 `释放连接的标志位 FIN` 。服务端进入 `LAST_ACK` 状态。
- 4.客户端收到 `标志位FIN` 后进入 `TIME_WAIT` 状态，然后发送一个 `确认标志位ACK` 给服务端，确认序号为 `收到序号+1` ，服务端进入 `CLOSED` 状态，完成四次挥手。

客户端进入 `TIME_WAIT` 状态而不是直接进入 `CLOSED` 状态是因为：我们必须要假想网络是不可靠的，你无法保证你最后发送的 `ACK报文` 一定会被对方收到，因此对方处于 `LAST_ACK` 状态下的 `SOCKET` 可能会因为超时未收到 `ACK` 报文，而重发FIN报文，所以这个 `TIME_WAIT` 状态的作用就是用来重发可能丢失的 `ACK` 报文。

为什么不能用两次握手进行连接?

> *因为三次握手是为了双方都知道彼此已做好准备，如果改成两次握手，会发生死锁。假设客户端给服务端发送连接请求，服务端收到后发送确认连接给客户端，如果两次握手的话此时服务端会认为已经建立了连接，可以发送数据了，但是如果确认连接数据丢失，客户端还不知道已经建立了连接，会一直等待确认连接序号，导致数据传输超时，服务端重复发送同样的数据，造成死锁。*

为什么是四次挥手而不是三次?

因为TCP是全双工连接，关闭连接需要双向确认关闭才算是真正的关闭，否则未关闭一方仍可以继续发送数据。

37. HTTPS的加密原理

`HTTPS` 是身披 `SSL` 外壳的 `HTTP` ，是一种通过计算机网络进行安全通信的传输协议，经由 `HTTP` 进行通信，利用 `SSL/TLS` 建立安全信道，加密数据包。

- 1. **客户端向服务端发送请求https://baidu.com，然后连接到server的443端口**
- 1. **服务端本身是要有一套数字证书的，这套证书其实就是一对公钥和私钥**

> *服务端必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面，这套证书其实就是一对公钥和私钥。*

- 1. **服务端把证书(公钥)传送给客户端**
  这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间、服务端的公钥，第三方证书认证机构(CA)的签名，服务端的域名信息等内容。
- 1. **客户端解析证书，验证无误后用证书加密一个随机生成的值(秘钥)**
  这部分工作是由客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随机值（秘钥）。然后用证书对该随机值进行加密。
- 1. **客户端传送加密后的秘钥给服务端**
  这部分传送的是用证书加密后的秘钥，目的就是让服务端得到这个秘钥，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。
- 1. **服务端用私钥解密随机值，然后通过该值对称加密信息**
  服务端用私钥解密秘钥，得到了客户端传过来的私钥，然后把内容通过该值进行对称加密。
- 1. **再传输加密后的信息给客户端**
  这部分信息是服务端用私钥加密后的信息，可以在客户端被还原。
- 1. **因为是对称加密，客户端可以解密信息**
  客户端用之前生成的私钥解密服务端传过来的信息，于是获取了解密后的内容。

HTTP和HTTPS的区别?

- `https` 协议需要到ca申请证书，一般免费证书很少，需要交费。
- `http` 是超文本传输协议，信息是明文传输， `https` 则是具有安全性的ssl加密传输协议。
- `http` 和 `https` 使用的是完全不同的连接方式用的端口也不一样，前者是80，后者是443。
- `http` 的连接很简单，是无状态的。
- `HTTPS` 协议是由 `SSL+HTTP` 协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全

38. WebSocket与TCP Socket的区别

- `Socket` 是抽象层，并不是一个协议，是为了方便使用TCP或UDP而抽象出来的一层，是位于应用层和传输层(TCP/UDP)之间的一组接口。
- `WebSocket` 和HTTP一样，都是应用层协议，是基于TCP 连接实现的双向通信协议，替代 `HTTP轮询` 的一种技术方案。是全双工通信协议， `HTTP` 是单向的，注意WebSocket和Socket是完全不同的两个概念。
  - `WebSocket` 建立连接时通过HTTP传输，但是建立之后，在真正传输时候是不需要HTTP协议的。
  - 相对于传统 `HTTP` 每次请求–应答都需要客户端与服务端建立连接的模式， `WebSocket` 是类似 `Socket` 的 `TCP 长连接` 的通讯模式，一旦 `WebSocket` 连接建立后，后续数据都以帧序列的形式传输。在客户端断开 `WebSocket` 连接或 `Server` 端断掉连接前，不需要客户端和服务端重新发起连接请求。在海量并发及客户端与服务器交互负载流量大的情况下，极大的节省了网络带宽资源的消耗，有明显的性能优势，且客户端发送和接受消息是在同一个持久连接上发起，实时性优势明显。
  - `WebSocket` 连接过程 —— 握手过程
    - 1.浏览器、服务器建立TCP连接，三次握手。这是通信的基础，传输控制层，若失败后续都不执行。
    - 2.TCP连接成功后，浏览器通过HTTP协议向服务器传送 `WebSocket` 支持的版本号等信息。（开始前的HTTP握手）
    - 3.服务器收到客户端的握手请求后，同样采用 `HTTP` 协议回馈数据。
    - 4.当收到了连接成功的消息后，通过 `TCP` 通道进行传输通信。
- `Socket.IO` 是一个为浏览器（客户端）和服务器之间提供实时，双向和基于事件的通信软件库。

`Socket.IO` 是把数据传输抽离成 Engine.IO，内部对轮询（Polling）和 WebSocket 等进行了封装，抹平一些细节和平台兼容的问题，提供统一的 API。

> *注意 `Socket.IO` 不是 `WebSocket` 的实现，只是在必要时使用 `WebSocket` 传输数据，并在此基础上会加一些 `MetaData` 。这就是为什么 `WebSocket` 的客户端/服务器 无法和 `Socket.IO` 的服务器/客户端进行通信。*

39. 事件传递和响应机制

在iOS中只有继承了 `UIResponder` 的对象才能接受并处理事件。

事件传递：事件的传递是从上到下（父控件到子控件）

- 产生触摸事件A后，触摸事件会被添加到由 `UIApplication` 管理的事件队列中(首先接收到事件的是UIApplication)。
- `UIApplication` 会从事件队列中取出最前面的事件(此处假设为触摸事件A)，将**事件对象由上往下传递**(`UIApplication` – `keyWindow` –父控件–子控件)，查找最合适的控件处理事件
- 只要事件传递给控件，就会调用自身的 `hitTest:withEvent:` 方法，寻找能够响应事件最合适的view(其内部会调用 `pointInside:withEvent:` 判断触摸点是否在自己身上)。

响应机制：从下到上（顺着响应者链条向上传递：子控件到父控件）
当事件传递到某个控件，但是最终 `hitTest:withEvent:` 没有找到第一响应者，那么该事件会沿着响应者链向上回溯，如果 `UIWindow实例` 和 `UIApplication实例` 都不能处理该事件，则该事件会被丢弃。

响应者链条：在iOS程序中视图的摆放是有前后关系的，一个控件可以放到另一个控件上面或下面，那么用户点击某个控件时是触发上面的控件还是下面的控件呢，这种先后关系构成一个链条就叫响应者链。也可以说，**响应者链是由多个响应者对象连接起来的链条**。

- 如果父控件不能接受触摸事件，那么子控件就不可能接收到触摸事件
- **UIView有三种情况不能接收事件：**
  - 不接收用户交互： `userInteractionEnabled = NO`
  - 隐藏： `hidden = YES`
  - 透明： `alpha<0.01`

子视图在父视图之外区域点击是否有效?

- 无效**(父视图的clipsToBounds=NO，这样超过父视图bound区域的子视图内容也会显示)，**因为父视图的 `pointInside:withEvent:` 方法会返回NO，就不会向下遍历子视图了。**
- 但是可以通过重写 `pointInside:withEvent:` 来处理。

40. runloop

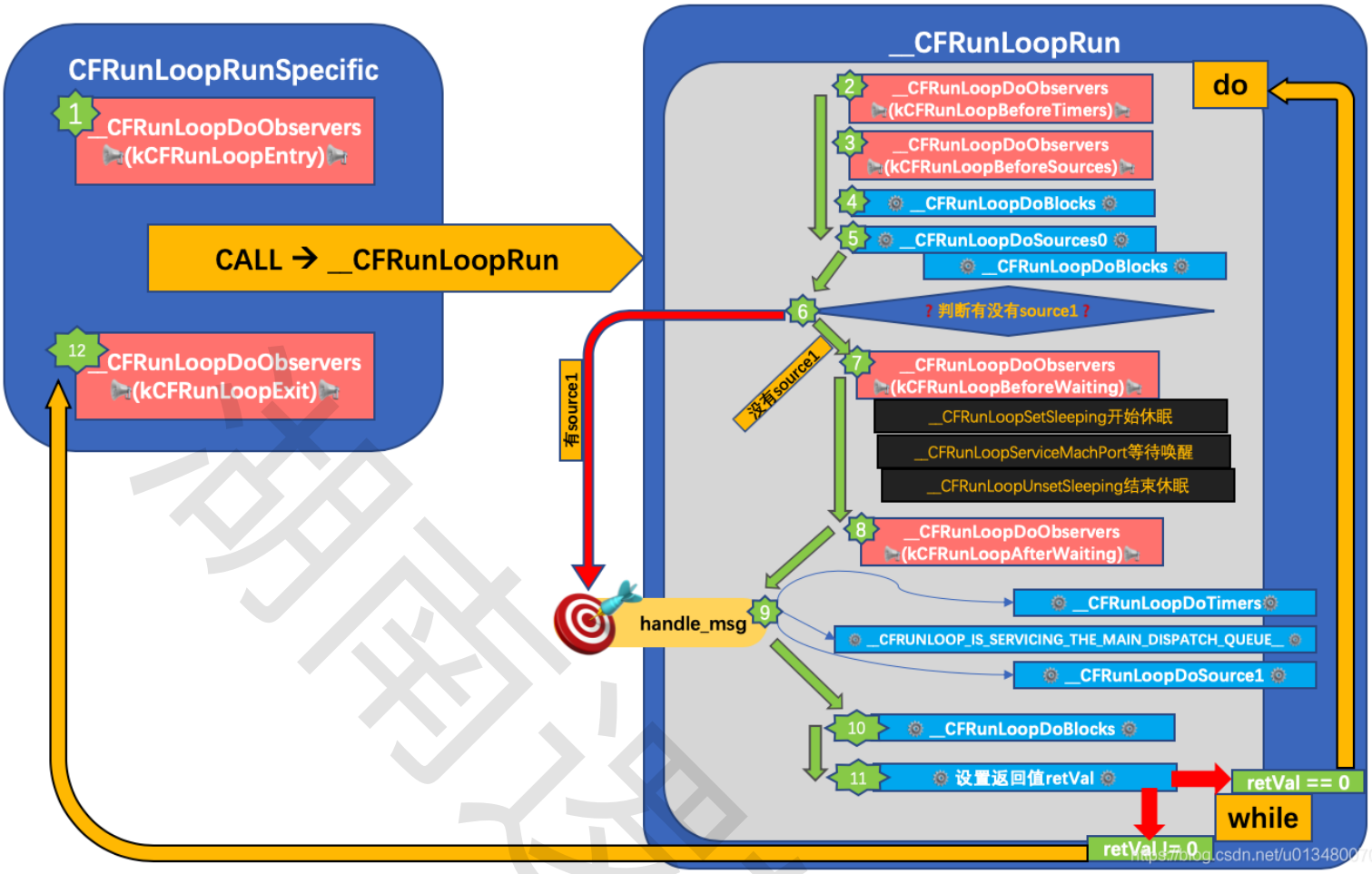
> *`runloop` ：通过系统内部维护的循环进行事件/消息管理的一个对象。runloop实际上就是一个 `do...while` 循环，有任务时开始，无任务时休眠。*

其本质是通过 `mach_msg()` 函数接收、发送消息。

RunLoop 与线程的关系：

- 1. `RunLoop` 的作用就是来管理线程的，当线程的 `RunLoop` 开启后，线程就会在执行完任务后，处于休眠状态，随时等待接受新的任务，不会退出。
- 1. 只有主线程的 `RunLoop` 是默认开启的，其他线程的 `RunLoop` 需要手动开启。所以当程序开启后，主线程会一直运行，不会退出。

runloop 事件循环机制内部流程



RunLoop主要涉及五个类：

`CFRunLoop`：RunLoop对象、

`CFRunLoopMode`：五种RunLoop运行模式、

`CFRunLoopSource`：输入源/事件源，包括`Source0`和`Source1`

`CFRunLoopTimer`：定时源，就是NSTimer、

`CFRunLoopObserver`：观察者，用来监听RunLoop。

- 1. `CFRunLoop`：**RunLoop对象**
- 1. `CFRunLoopMode`：**RunLoop运行模式，有五种：**
  - `kCFRunLoopDefaultMode`：默认的运行模式，通常主线程是在这个 Mode 下运行的。
  - `UITrackingRunLoopMode`：界面跟踪 Mode，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`：接受系统事件的内部 Mode，通常用不到。
  - `kCFRunLoopCommonModes`：是一个伪模式，可以在标记为CommonModes的模式下运行，RunLoop会自动将`_commonModeItems`里的`Source`、`Observer`、`Timer`同步到具有此标记的Mode里。

1. `CFRunLoopSource`：**输入源/事件源，包括Source0 和 Source1两种：**
   - `Source1`：基于mach_Port，处理来自系统内核或其它进程的事件，比如点击手机屏幕。
   - `Source0`：非基于Port的处理事件，也就是应用层事件，需要手动标记为待处理和手动唤醒RunLoop。
   - 简单举例：一个APP在前台静止，用户点击APP界面，屏幕表面的事件会先包装成`Event`告诉`source1(mach_port)`，`source1`唤醒`RunLoop`将事件`Event`分发给`source0`，由`source0`来处理。

2. `CFRunLoopTimer`： **定时源**，就是NSTimer。在预设的时间点唤醒RunLoop执行回调。因为它是基于RunLoop的，因此它不是实时的（就是NSTimer 是不准确的。 因为RunLoop只负责分发源的消息。如果线程当前正在处理繁重的任务，就有可能导致Timer本次延时，或者少执行一次）。
3. `CFRunLoopObserver`： **观察者，用来监听以下时间点：CFRunLoopActivity**
    - `kCFRunLoopEntry`： RunLoop准备启动
    - `kCFRunLoopBeforeTimers`： RunLoop将要处理一些Timer相关事件
    - `kCFRunLoopBeforeSources`： RunLoop将要处理一些Source事件
    - `kCFRunLoopBeforeWaiting`： RunLoop将要进行休眠状态,即将由用户态切换到内核态
    - `kCFRunLoopAfterWaiting`： RunLoop被唤醒，即从内核态切换到用户态后
    - `kCFRunLoopExit`： RunLoop退出
    - `kCFRunLoopAllActivities`： 监听所有状态

各数据结构之间的联系：

- 1： `Runloop` 和 `线程` 是**一对一**的关系
- 2： `Runloop` 和 `RunloopMode` 是**一对多**的关系
- 3： `RunloopMode` 和 `RunloopSource` 是**一对多**的关系
- 4： `RunloopMode` 和 `RunloopTimer` 是**一对多**的关系
- 5： `RunloopMode` 和 `RunloopObserver` 是**一对多**的关系

为什么 main 函数能够保持一直存在且不退出?

> *在 main 函数内部会调用 `UIApplicationMain` 这样一个函数，而在 `UIApplicationMain` 内部会启动主线程的 `runloop`，可以做到有消息处理时，能够迅速从**内核态到用户态**的切换，立刻唤醒处理，而没有消息处理时通过**用户态到内核态**的切换进入等待状态，**避免资源占用**。**因此 main 函数能够一直存在且不退出**。*

41. runtime

什么是runtime?

runtime 一套c、c++、汇编编写的API，为OC提供运行时功能。能够将数据类型的确定由编译期推迟到运行时。

问1：方法的本质，问2：runtime的消息机制

方法的本质其实就是发送消息。

发送消息主要流程：

- 1. 快速查找： `objc_msgSend` 查找 `cache_t` 缓存消息
- 1. 慢速查找：递归自己和父类查找方法 `lookUpImpOrForward`
- 1. 查找不到消息，进行动态方法解析： `resolveInstanceMethod`
- 1. 消息快速转发： `forwardingTargetForSelector`
- 1. 消息慢速转发：消息签名 `methodSignatureForSelector` 和分发 `forwardInvocation`
- 1. 最终仍未找到消息：程序crash，报经典错误信息 `unrecognized selector sent to instance xxx`

SEL是什么？IMP是什么？两者有什么联系?

- `SEL` 是方法编号，即方法名称，在dyld加载镜像时，通过read_image方法加载到内存的表中了。
- `IMP` 是函数实现指针，找IMP就是找函数的过程
- **两者的关系：** `sel` 相当于书本的目录标题， `imp` 就是书本的页码。查找具体的函数就是想看这本书里面具体篇章的内容：
    - 1). 我们首先知道想看什么，也就是title –sel
    - 2). 然后根据目录对应的页码 –imp
    - 3). 打开具体的内容 –方法的具体实现

runtime应用：

- 1.方法的交换：具体应用拦截系统自带的方法调用（Method Swizzling黑魔法）
- 2.实现给分类增加属性
- 3.实现字典的模型和自动转换
- 4.JSPatch替换已有的OC方法实行等
- 5.aspect 切面编程

能否向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？

- 1.不能向编译后得到的类中增加实例变量。
- 2.可以向运行时创建的类中添加实例变量。
- 3.因为编译后的类已经注册在 runtime 中，类结构体中的 `objc_ivar_list` **实例变量的链表**和 `instance_size` **实例变量的内存大小**已经确定，同时 `runtime` 会调用 `class_setIvarLayout` 或 `class_setWeakIvarLayout` 来处理 `strong` 、 `weak` 引用，所以不能向存在的类中添加实例变量。

运行时创建的类是**可以添加实例变量**，调用 `class_addIvar` 函数。但是得在调用 `objc_allocateClassPair` 之后， `objc_registerClassPair` 之前，原因同上。

Category中添加属性和成员变量的区别

- 1. `Category` 它的主要作用是在不改变原有类的前提下，动态地给这个类添加一些方法。
- 1. 分类的结构体指针中，没有属性列表，只有方法列表。原则上它只能添加方法，不能添加属性(成员变量)，但是可以借助运行时关联对象 `objc_setAssociatedObject(self, @selector(name), name, OBJC_ASSOCIATION_COPY_NONATOMIC);` 、 `objc_getAssociatedObject(self,@selector(name));` 。
- 1. 分类中的可以写 `@property` ，但不会生成setter/getter方法声明和实现，也不会生成私有的成员变量，会编译通过，但是引用变量会报错。
- 1. 如果分类中有和原有类同名的方法，会优先调用分类中的方法，就是说会忽略原有类的方法，同名方法调用的优先级为 分类 本类 父类，因为方法是放在方法栈中，遵循先进后出原则；

42. isa指针

- `isa` 是一个 `Class类型` 的指针，其源码结构为 `isa_t` 联合体，在类中以Class对象存在，指向类的地址，大小为8字节(64位)。
- 每个实例对象都有 `isa` 的指针指向对象的类。 `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。元类（meteClass）也是类，它也是对象，也有isa指针。

isa的指向：对象的 `isa` 指向类，类的 `isa` 指向元类(meta class)，元类 `isa` 指向根元类，根元类的 `isa` 指向本身，形成了一个封闭的内循环。isa可以帮助一个对象找到它的方法。

isa指向图中类的继承关系： `LGTeacher -> LGPerson -> NSObject -> nil` 。这里需要注意的是根元类的父类是 `NSObject` ， `NSObject` 的父类是 `nil` 。

43. block

什么是Block
Block是将函数及其执行上下文封装起来的对象。

什么是Block调用
Block调用即是函数的调用。

Block的几种形式(类型)

Block有三种形式，包括：

- **全局Block**(`_NSConcreteGlobalBlock`)：当我们声明一个block时，如果这个block没有捕获外部的变量，那么这个block就位于全局区(已初始化数据(.data)区)。
- **栈Block**(`_NSConcreteStackBlock`)：
  - 1). ARC环境下，当我们声明并且定义了一个block，系统默认使用 `__strong修饰符` ，如果该 `Block` 捕获了外部变量，实质上是从 `__NSStackBlock__` 转变到 `__NSMallocBlock__` 的过程，只不过是系统帮我们完成了copy操作，将栈区的block迁移到堆区，延长了Block的生命周期。对于 `栈block` 而言，变量作用域结束，空间被回收。
  - 2). ARC的环境下，如果我们在声明一个block的时候，使用了 `__weak` 或者 `__unsafe__unretained` 的修饰符，那么系统就**不会做copy的操作**，也

就**不会将其迁移到堆区**。

- **堆Block**(`_NSConcreteMallocBlock`):
    - 1). 在MRC环境下，我们需要手动调用copy方法才可以将block迁移到堆区
    - 2). 而在ARC环境下， `__strong` 修饰的（默认）block捕获了外部变量就会位于堆区， `NSMallocBlock` 支持 `retain` 、 `release` ，会对其引用计数＋1或 –1。
    - **只有局部变量 – 和定义的属性 才会拷贝到堆区**

1). 存储在程序的数据区域，在 block 内部没有引用任何外部变量。

2). 使用外部变量并且未进行copy操作的block是栈block。

3). 对 `栈block` 进行copy操作，就是 `堆block` 。对 `堆Block` 进行copy，将会 `增加引用计数` 。对 `全局block` 进行copy，仍是 `全局block` 。

在什么场景下使用__block修饰符呢?

- 1). 对截获变量进行赋值操作需要添加 `__block修饰符` （赋值 != 使用）。
- 2). 对局部变量（基本数据类型和对象类型）进行赋值需要 `__block修饰符` 。
  其内部其实是对该 `__block对象` 进行拷贝，所以通过 `__block` 可以修改被截获变量的值且不会和外部变量互相影响。
- 3). 对静态局部变量、全局变量、静态全局变量不需要 `__block修饰符` 。

block 的截获变量特性?

基本数据类型的局部变量 Block 可以截获其值。

对于对象类型的局部变量连同所有权修饰符一起截获。

局部静态变量以指针的形式进行截获。

全局变量和静态全局变量，block 是不截获的。

weak打破Block循环引用原理

> *`block` 内部操作的是 `weakSelf` 的指针地址，它和 `self` 是两个不同的指针地址，即 没有直接持有 `self` ，所以可以 `weakSelf` 可以打破 `self` 的循环引用关系 `self –block – weakSelf` 。*