**1.KVO实现原理？**

**2.说说你理解的埋点？**

**3.消息转发机制原理？**

**4.说说你理解weak属性？**

**5.假如Controller太臃肿，如何优化？**

**6.项目中网络层如何做安全处理？**

**7.main()之前的过程有哪些？**

**1.KVO实现原理？**

**KVO在Apple中的API文档如下：**

Automatic key-value observing is implemented using a technique called isa-swizzling… When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class …

**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。

**KVO深入原理：**

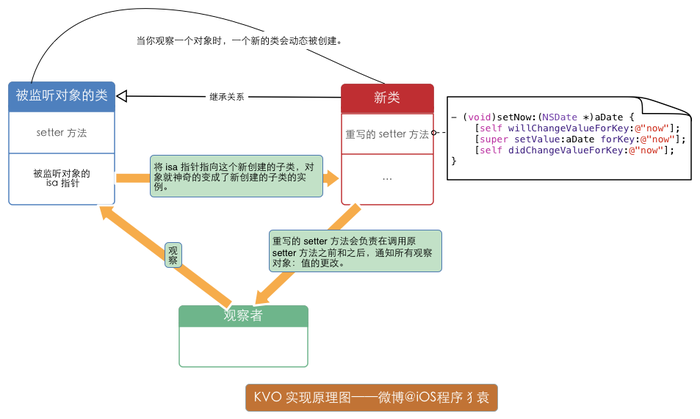
1.Apple 使用了 isa 混写（isa-swizzling）来实现 KVO 。当观察对象A时，KVO机制动态创建一个新的名为：?NSKVONotifying\_A的新类，该类继承自对象A的本类，且KVO为NSKVONotifying\_A重写观察属性的setter?方法，setter?方法会负责在调用原?setter?方法之前和之后，通知所有观察对象属性值的更改情况。

2.NSKVONotifying\_A类剖析：在这个过程，被观察对象的 isa 指针从指向原来的A类，被KVO机制修改为指向系统新创建的子类 NSKVONotifying\_A类，来实现当前类属性值改变的监听；

3.所以当我们从应用层面上看来，完全没有意识到有新的类出现，这是系统“隐瞒”了对KVO的底层实现过程，让我们误以为还是原来的类。但是此时如果我们创建一个新的名为“NSKVONotifying\_A”的类()，就会发现系统运行到注册KVO的那段代码时程序就崩溃，因为系统在注册监听的时候动态创建了名为NSKVONotifying\_A的中间类，并指向这个中间类了。

4.（isa 指针的作用：每个对象都有isa 指针，指向该对象的类，它告诉 Runtime 系统这个对象的类是什么。所以对象注册为观察者时，isa指针指向新子类，那么这个被观察的对象就神奇地变成新子类的对象（或实例）了。）?因而在该对象上对 setter 的调用就会调用已重写的 setter，从而激活键值通知机制。

5.子类setter方法剖析：KVO的键值观察通知依赖于 NSObject 的两个方法:willChangeValueForKey:和 didChangevlueForKey:，在存取数值的前后分别调用2个方法： 被观察属性发生改变之前，willChangeValueForKey:被调用，通知系统该 keyPath?的属性值即将变更；当改变发生后， didChangeValueForKey: 被调用，通知系统该 keyPath?的属性值已经变更；之后，?observeValueForKey:ofObject:change:context: 也会被调用。且重写观察属性的setter?方法这种继承方式的注入是在运行时而不是编译时实现的。



KVO原理图

**2.说说你理解的埋点？**

**以下几篇文章写的相当不错，可以适当借鉴下！**

[**iOS无埋点数据SDK实践之路**](https://www.jianshu.com/p/5f16e1de6d5a)

[**iOS无埋点数据SDK的整体设计与技术实现**](https://www.jianshu.com/p/5f16e1de6d5a)

[**iOS无埋点SDK 之 RN页面的数据收集**](https://www.jianshu.com/p/0e1f2d357e39)

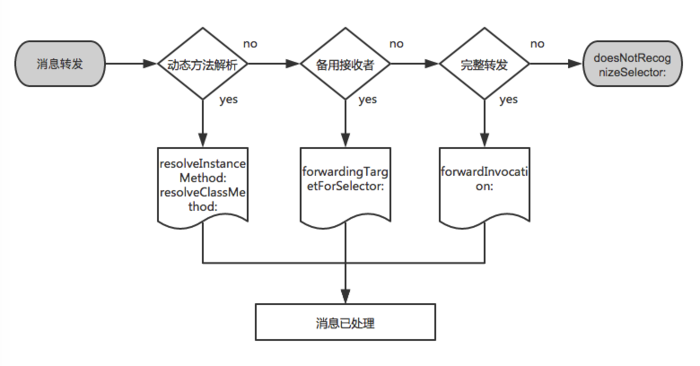
**3.消息转发机制原理？**

**消息转发机制基本分为三个步骤：**

1、动态方法解析

2、备用接受者

3、完整转发



转发机制原理

**新建一个HelloClass的类，定义两个方法：**

|  |  |
| --- | --- |
| 1  2  3 | @interfaceHelloClass:NSObject  - (**void**)hello;  + (HelloClass \*)hi;@end |

**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。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **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**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | + (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];  } |

**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**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | - (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**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @interfaceHelloClass()  {  RuntimeMethodHelper \*\_helper;  }  @end  @implementationHelloClass- (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：**

|  |  |
| --- | --- |
| 1  2  3  4  5 | #**import**"RuntimeMethodHelper.h"  @implementationRuntimeMethodHelper- (**void**)hello  {   NSLog(@"%@, %p", self, \_cmd);  }@end |

**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**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | - (**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\*/**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | - (NSMethodSignature \*)methodSignatureForSelector:(SEL)aSelector  {  NSMethodSignature \*signature = [**super** methodSignatureForSelector:aSelector];  **if** (!signature)  {  **if** ([RuntimeMethodHelper instancesRespondToSelector:aSelector])  {   signature = [RuntimeMethodHelper instanceMethodSignatureForSelector:aSelector];  }  }  **return** signature;  } |

**4.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。

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。

**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

**5.假如Controller太臃肿，如何优化？**

**1.将网络请求抽象到单独的类中**

方便在基类中处理公共逻辑；

方便在基类中处理缓存逻辑，以及其它一些公共逻辑；

方便做对象的持久化。

**2.将界面的封装抽象到专门的类中**

构造专门的 UIView 的子类，来负责这些控件的拼装。这是最彻底和优雅的方式，不过稍微麻烦一些的是，你需要把这些控件的事件回调先接管，再都一一暴露回 Controller。

**3.构造 ViewModel**

借鉴MVVM。具体做法就是将 ViewController 给 View 传递数据这个过程，抽象成构造 ViewModel 的过程。

**4.专门构造存储类**

专门来处理本地数据的存取。

**5.整合常量**

**6.项目中网络层如何做安全处理？**

**1、尽量使用https**

https可以过滤掉大部分的安全问题。https在证书申请，服务器配置，性能优化，客户端配置上都需要投入精力，所以缺乏安全意识的开发人员容易跳过https，或者拖到以后遇到问题再优化。https除了性能优化麻烦一些以外其他都比想象中的简单，如果没精力优化性能，至少在注册登录模块需要启用https，这部分业务对性能要求比较低。

**2、不要传输明文密码**

不知道现在还有多少app后台是明文存储密码的。无论客户端，server还是网络传输都要避免明文密码，要使用hash值。客户端不要做任何密码相关的存储，hash值也不行。存储token进行下一次的认证，而且token需要设置有效期，使用refresh

token去申请新的token。

**3、Post并不比Get安全**

事实上，Post和Get一样不安全，都是明文。参数放在QueryString或者Body没任何安全上的差别。在Http的环境下，使用Post或者Get都需要做加密和签名处理。

**4、不要使用301跳转**

301跳转很容易被Http劫持攻击。移动端http使用301比桌面端更危险，用户看不到浏览器地址，无法察觉到被重定向到了其他地址。如果一定要使用，确保跳转发生在https的环境下，而且https做了证书绑定校验。

**5、http请求都带上MAC**

所有客户端发出的请求，无论是查询还是写操作，都带上MAC（Message Authentication

Code）。MAC不但能保证请求没有被篡改（Integrity），还能保证请求确实来自你的合法客户端（Signing）。当然前提是你客户端的key没有被泄漏，如何保证客户端key的安全是另一个话题。MAC值的计算可以简单的处理为hash（request

params＋key）。带上MAC之后，服务器就可以过滤掉绝大部分的非法请求。MAC虽然带有签名的功能，和RSA证书的电子签名方式却不一样，原因是MAC签名和签名验证使用的是同一个key，而RSA是使用私钥签名，公钥验证，MAC的签名并不具备法律效应。

**6、http请求使用临时密钥**

高延迟的网络环境下，不经优化https的体验确实会明显不如http。在不具备https条件或对网络性能要求较高且缺乏https优化经验的场景下，http的流量也应该使用AES进行加密。AES的密钥可以由客户端来临时生成，不过这个临时的AES

key需要使用服务器的公钥进行加密，确保只有自己的服务器才能解开这个请求的信息，当然服务器的response也需要使用同样的AES

key进行加密。由于http的应用场景都是由客户端发起，服务器响应，所以这种由客户端单方生成密钥的方式可以一定程度上便捷的保证通信安全。

**7、AES使用CBC模式**

不要使用ECB模式，记得设置初始化向量，每个block加密之前要和上个block的秘文进行运算。

**7.main()之前的过程有哪些？**

**1、main之前的加载过程**

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**