iOS 面试题总结

一、响应者链

概念

当我们触控手机屏幕时，系统会把这一操作封装成一个UIEvent或者UITouch事件，然后找到当前正在运行的程序，在这个程序里面逐级寻找能够响应这个事件的所有对象，然后把这些对象放入一个链表，这个链表就是一个响应链。

事件传递顺序

由上往下：UIAPPlication - UIWidow - ViewController - UIView - superView - View

事件的传递是使用hitTest:withEvent 和pointInside两个方法来寻找最合适的响应者

首先会通过调用自身的pointInside方法判断用户的触摸点是否在当前对象的响应范围内，若返回NO，则hitTest返回nil；若返回YES，hitTest方法会接着判断自身是否有子视图，如果有则调用顶层子视图的hitTest方法，返回VIew；如果所有子视图都返回nil，则方法返回自身

响应者顺序

由下往上：View(第一响应者)- superView - ViewController - UIWindow - UIApplication - AppDelegate

问题一：子视图超出了父视图区域，不能响应点击事件怎么处理?

原因：因为父视图的pointInside:withEvent方法返回了NO，就不会遍历子视图了。
解决：重写pointInside:withEvent方法。其中子视图返回nil，让父视图成为Hit-Test view. 父视图成为first responder，子视图把事件沿着响应链转发。

问题二：为什么CALayer 不能响应事件?

因为CALayer 不是继承自UIResponder
iOS 中的对象要想响应事件，就必须直接或间接的继承UIResponder

问题三：如何扩大按钮的点击热区?

重写按钮的pointInside方法，若原热区小于44x44，则放大热区，否则保持原大小不变

参考:https://juejin.im/entry/58f5b7d0570c35005648a1ba

二、Block

Block 的本质其实就是OC 对象，它的内部也有isa指针。所以说block 是封装了函数调用环境的OC 对象

几种Block

1、_NSConcreteGlobalBlock 全局block 存放在程序区
2、_NSConcreteStackBlock 栈block 存放在栈上
3、_NSConcreteMallocBlock 堆block 存放在堆上

Block的copy原理

Block 被创建时，默认的内存是分配到栈上的，栈是由系统自动管理，所以说不定什么时候block就会被释放掉，造成程序崩溃。所以需要使用copy，把block 从栈区复制到堆区。使用retain也可以，只不过retain进行的也是copy操作。
在MRC中，block 是在栈区，使用copy可以把它放多堆区。但在arc中，写不写都行，编译器会自动对block 进行copy操作

多层嵌套block

可以采用一个__weak,一个__strong 的方式来方式循环引用

```
- (void)setUpModel{
XXModel *model = [XXModel new];

__weak typeof(self) weakSelf = self;

model.dodoBlock = ^(NSString *title) {

    __strong typeof(self) strongSelf = weakSelf;//第一层
    strongSelf.titleLabel.text = title;

    __weak typeof(self) weakSelf2 = strongSelf;
    strongSelf.model.dodoBlock = ^(NSString *title2) {

        __strong typeof(self) strongSelf2 = weakSelf2;//第二层
        strongSelf2.titleLabel.text = title2;
    };
};

self.model = model;

}
```

参考链接：https://blog.csdn.net/nathan1987_/article/details/82749057

block 如何捕获局部变量的

全局变量和全局静态变量能被block捕获进去，是因为他们是全局的，作用域广。并且可以在block里面进行值操作，block 结束之后，他们的值也能保存下来。
局部自动变量和局部静态变量，被block 捕获后，就成为了block结果体的成员变量了。而局部静态变量之所以能在block 内修改值，是因为它传进block 的是指针传递。而局部自动变量不能修改值，block 捕获的只是自动变量的值，并没有捕获到它的内存地址，所以不能改变自动变量的值。要想改变自动变量的值，需要用__block来修饰

__block修饰的变量为什么能在block 里面改变其值?

__block 所起到的作用就是观察到该变量被block所持有，就将“外部变量”在栈中的内存地址copy到堆上，进而在修改它的值

block中__block的实现原理

__block变量在没有复制到堆上时，__blcok变量结构体中的__forwarding指针指向自己。当__block变量被复制到堆上后，栈上的__block变量结构体中的__forwarding指针指向堆上的__block变量结构体，堆上的__block变量结构体中的__forwarding指针指向它自己。这样不管__block怎么复制到堆上，还是在栈上都可以通过forwarding只访问到变量的值

三、Runtime

运行时机制原理

OC 的函数调用称为消息发送，属于动态调用过程。在编译的时候并不能决定真正调用哪个函数，只有在真正运行的时候才会根据函数的名称找到对应的函数来调用
我们平时写的OC代码，底层都是基于runtime来实现的

OC方法转化为runtime方法过程

```
 //id objc =[NSObject alloc];

 id objc = objc_msgSend([NSObject class],@selector(alloc));

  // objc = [objc init];

objc = objc_msgSend(objc, @selector(eat));

//[objc message];

objc_msgSend(objc,selector);
```

消息机制方法调用流程

```
1、OC在向一个对象发送消息时，runtime库会根据对象的isa指针找到该对象对应的类或其父类中查找方法；
2、注册方法编号sel_registerName（方便快速查找）；
3、根据方法编号去查找对应方法；
4、找到最终函数实现的地址，根据地址去方法区调用对应的函数
```

参考：https://juejin.im/post/593f77085c497d006ba389f0#comment

给分类添加伪属性

通过关联对象为分类增加伪属性

```
objc_getAssociatedObject(self, _cmd)
objc_setAssociatedObject(self, @selector(), title, OBJC_ASSOCIATION_RETAIN);
```

实现页面埋点、防崩溃保护

使用method swizzing 交换两个方法的实现，以便达到hook的效果

```
class_getInstanceMethod
class_addMethod
class_replaceMethod
method_exchangeImplementations
```

实现自动归解档

```
利用class_copyIvarList 获取成员变量列表
使用ivar_getName 来获取成员变量名称
然后利用KVC 来code 和decode
```

字典转模型

```
调用class_copyPropertyList 获取属性列表
调用property_getName 获取属性名称
```

实现多播代理

```
原理：实现一个管理类，将需要回调的对象注册进来，然后将事件消息发送给这个管理类，由于这个管理类是没有实现委托方法的，就不能正常处理这个消息，这个时候就会走消息转发流程；然后我们通过消息转发流程，将消息转发到注册进来的对象中去，这样子就可以实现我们的多播委托了。
```

访问并修改类的私有属性

利用class*copyIvarList 获取成员变量列表 使用ivar*getName 来获取成员变量名称

也可以通过kvc的方式访问

四、RunLoop

RunLoop的概念

```
一个runloop就是一个事件处理循环，它的内部实现其实就是do-while 循环。每个线程都有一个对应的runloop，主线程的runloop；默认是开启的，子线程的runloop默认是关闭的，如果需要开启，则要调用【runlop run】方法。RunLoop在第一次获取时创建，在线程结束时销毁
```

子线程的runloop

```
苹果不允许直接创建 RunLoop，它只提供了两个自动获取的函数：CFRunLoopGetMain() 和 CFRunLoopGetCurrent()。

[NSRunLoop currentRunLoop];方法调用时，会先看一下字典里有没有存子线程相对用的RunLoop，如果有则直接返回RunLoop，如果没有则会创建一个，并将与之对应的子线程存入字典中。
```

RunLoop的创建

RunLoop是CFRunLoopRef的OC 封装，CFRunLoop实际也是一个结构体；一个runloop对象，主要包含一个线程、若干个mode、若干个commonMode和一个currentMode

CFRunLoopRef通过CFRunLoopGetMain()和CFRunLoopGetCurrent()来获取主线程runloop和当前线程runloop。这两个函数内部都调用了_CFRunLoopGet0(), CFRunLoopGetMain传入的线程是pthread_main_thread_np(), CFRunLoopGetCurrent传入的是pthread_self().

_CFRunLoopGet0(pthread t)的内部实现大概是这样的：定义一个静态的可变字典CFRunloops，用来存放线程和runloop 的映射。如果这个字典CFRunloops不存在（表明是第一次进来），则创建创建主线程的runloop CFRunLoopRef mainLoop = __CFRunLoopCreate(pthread_main_thread_np()); 然后把主线程的runloop 保存到dict中，key是线程，value是runloop。如果不是第一次进来，从当前字典CFRunloops中获取传入的线程t的runloop。如果没有获取到，则根据线程t创建一个runloop CFRunLoopRef newLoop = __CFRunLoopCreate(t);然后把newloop存入字典_CFRunloops,key是线程t

RunLoop的实现

RunLoop的实现主要在CFRunLoopRunSpecific这个函数里面。这个函数的执行过程大概是这样的：

1、Entry:通知OB(创建Pool)；

2、执行阶段：按顺序通知OB并执行timer,source0;若有source1执行source1；

3、休眠阶段：利用mach_msg判断进入休眠，通知OB(pool的销毁重建)；被消息唤醒通知OB；

4、执行阶段：按消息类型处理事件；

5、判断退出条件：如果符合退出条件（一次性执行，超时，强制停止，modeItem为空）则退出，否则回到第2阶段；

6、Exit:通知OB（销毁pool）；

RunLoop监听的状态

```
typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
kCFRunLoopEntry         = (1UL << 0), // 即将进入Loop
kCFRunLoopBeforeTimers  = (1UL << 1), // 即将处理 Timer
kCFRunLoopBeforeSources = (1UL << 2), // 即将处理 Source
kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠
kCFRunLoopAfterWaiting  = (1UL << 6), // 刚从休眠中唤醒
kCFRunLoopExit          = (1UL << 7), // 即将退出Loop
```

};

RunLoop 和 CFRunLoopRef 区别

CFRunLoopRef基于C,线程安全，NSRunLoop基于CFRunloopRef面向对象的API,是不安全的

Runloop本质

mach port和mach_msg()

Runloop有两个关键判断点：一个是通过msg决定Runloop是否等待（等待状态mach_msg_trap()），一个是通过判断退出条件来决定是否runloop循环

RunLoop的作用

1、保持程序运行

2、处理App中的各种事件

3、节省CPU资源，提高程序性能

RunLoop的六大函数

```
CFRunLoop_Observer

CFRunLoop_Block

CFRunLoop_Dispatch_queue

CFRunLoop_Timer

CFRunLoop_Source0

CFRunLoop_Source1
```

RunLoop的本质

```
RunLoop通过mach_msg()函数发送消息，如果没有port消息，内核会处于等待状态mach_msg_trap().如果有消息，就处理消息
```

RunLoop的source

```
source0：非基于Port 的用于用户主动除法（点击button或者点击屏幕）

source1：基于Port 的，通过内核和其他线程相互发送消息（与内核相关），source1由runloop 和内核管理

Soure1在处理的时候会分发一些操作给Source0去处理
```

RunLoop如何处理界面刷新

```
 在操作UI时，比如修改了Frame或者手动调用了UIView/CALayer的setNeedsLayout/setNeedsDisplay方法后，这个UIView/CALayer会被标记为待处理，并提交到一个全局的容器中去。

 苹果注册了一个Observer来监听runloop的BeforeWaiting(即将进入休眠) 和 Exit (即将退出Loop) 事件，然后回调函数会遍历容器中所有待处理的事件，以执行实际的调整，并更新界面
```

Runloop如何处理UI事件响应

```
苹果注册了一个Source1用来接收系统事件，它的回调函数为__IOHIDEventSystemClientQueueCallback（）。当一个硬件事件（触摸/锁屏/摇晃）发生后。首先由IOKit.framework生成一个IOHidEvent事件，并由SpringBoard接收。随后用mach port转发给需要的APP 进程。随后苹果注册的source1就会触发回调，并调用_UIApplicationHandleEventQueue()进行应用内部的分发。_UIApplicationHandleEventQueue会把IOHidEvent处理并包装成UIEvent事件进行分发。
```

Runloop如何处理手势

```
当上面的_UIApplicationHandleEventQueue()识别了一个手势后，首先会调用cancel将当前的touchesBegin/Move/End系列回调打断。随后将对应的UIGestureRecognizer标记为待处理。苹果注册了一个Observer检测BeforeWaiting(Runloop即将进入休眠)事件，然后Observer会在其回调函数内部获取所有刚被标记为待处理的UIGestureRecognizer，并执行UIGestureRecognizer的回调。
```

Runloop如何处理GCD 任务

```
当调用disPathch_async(dispatch_get_main_queue(), block) 时,libDispatch会向主线程的Runloop发送消息，Runloop会被唤醒，并从消息中取得这个block，并在回调里执行这个block。Runloop只处理主线程的block，dispatch到其他线程仍然是由libDispatch处理的
```

Runloop如何处理网络请求

```
 关于网络请求的最底层接口是CFSocket层，然后是CFNetwork将其封装，然后是NSURLConnectiond对CFNetwork进行面向对象的封装。

 网络开始传输的时候，CFSocket线程处理底层的socket链接。NSURLConnectionLoader会使用Runloop来接收socket的事件，并通过之前添加的Source0通知到上层的Delegate。
```

AutoReleasePool的创建和销毁是在什么时候发生的

APP在启动时，苹果在主线程的runloop里面注册了两个Observer,其回调都是_wrapRunloopWithAutoreleasePoolHandler().

第一个observer监听的事件是Entry(即将进入runloop)，其内部会调用_objc_autoreleasePoolPush()创建自动释放池。其Order是-2147483647,优先级最高，保证创建释放池发生在其他所有回调之前。

第二个Observer监听两个事件BeforeWaiting(即将进入休眠)时调用_objc_autoReleasePoolPop()和_objc_autoreleasePoolPush()释放旧的池子并创建新的池子；Exit(即将退出loop)时调用_objc_autoreleasePoolPop（）来释放自动释放池。这个observer的优先级最低，order是2147483647，保证其释放池子发生在其他所有回调之后。

参考:https://juejin.im/entry/587c2c4ab123db005df459a1 https://blog.csdn.net/hherima/article/details/51746125 https://juejin.im/post/5aca2b0a6fb9a028d700e1f8

五、多线程

六、设计模式

MVC

Model 、View 、Controller
Model层负责处理数据逻辑和保存应用模型
View层负责视图展示
Controller负责model和view的通讯、业务逻辑的处理、网络请求等

MVVM

MVVM就是在MVC的基础上分离出业务处理的逻辑到viewModel层

model层：负责API请求的原始数据、数据持久化

View层：视图展示，由ViewController来控制

ViewModel层：负责网络请求、业务逻辑处理和数据转化，来配合Controller工作，从而使Controller更加简单

简单来说就是API请求完数据，解析成model，之后在viewModel中转化成能够直接被视图层使用的数据返回给view层。view层是由viewController控制的，view层只做展示，不做业务处理。view层的数据由viewModel提供

优点：可测试性高，减轻了Controller的工作

缺点：代码量增加

MVP

MVP是由MVC演变而来，将Controller替换成Presenter。

优点：完全分类Model和View

单例模式

用一个静态方法返回这个类的对象。这个对象在全局是唯一的。整个项目里面只开辟一块内存。一个单例在项目中只有一个类别，提供类方法供全局调用

优点：使用简单，方便传值和修改单例的属性，便于资源共享控制

缺点：这块内存直到项目退出时才能释放

委托模式

通过代理和协议的方式，实现一对一传值

优点：解耦合

缺点：

观察者模式

通过kvo 机制观察某个类的属性，属性变化时，通知观察者

优点：解耦合，只负责负责发送消息，不关心谁去接收

工厂模式

通过类方法，批量的生产对象，快速创建对象的方式 （比如UIView的initWithFrame，创建常用按钮、textField等）

优势：易于替换，面向抽象变成

缺点：增加了代码的复杂度，调用层次和内存负担

七、KVO/KVC

KVO

原理

当一个类的属性第一次被观察时，系统会动态的给这个类创建一个派生类，并把isa指针指向这个类，在这个派生类里面重写基类中被观察的属性的set方法。在重写的setter方法中实现真正的通知机制

自己实现KVO

1、先检查对象有没有setter方法，如果没有就抛出异常；

2、检查对象的isa指针指向的类是不是一个KVO类，如果不是，新建一个继承原来类的子类，并把isa指针指向这个新建的子类；

3、检查这个KVO类有没有重写属性的setter方法，如果没有，就重写setter方法；

4、添加观察者

KVO为什么要创建子类来实现

如何关掉系统KVO，自己实现

KVC

内部实现

一个对象在调用setValue方法时，方法内部会做以下操作：

1、检查是否存在对应的key的set方法，如果存在，直接调用set方法赋值

2、如果set方法不存在，就会查找与key相同名称并且带有下划线的成员变量_key,如果有，则直接赋值

3、如果没有找到_key,就会查找相同名称的属性key,如果有就直接赋值
4、如果还是没找到，就调用valueForUndefinedKey:和setValue：forUndefineKey:来抛出异常

八、三方原理

1、AFNetworking

问题一：AFN为什么添加一条常驻线程

目的就是开辟线程请求网络数据。如果没有一条常驻线程的话，每次网络请求就去开辟新线程，完成之后再销毁线程，这样就造成资源的浪费。开启常驻线程可以避免这种浪费，每次有网络请求的话都添加到这条线程里。

2、SDWebImage

原理：

1）从缓存中（字典）找图片（当这个图片在本次程序加载过），找到直接使用；
2）从磁盘中找，找到直接使用，缓存到内存。
3）从网络上获取，使用，缓存到内存，缓存到沙盒。

问题一：SDWebImage为什么要进行解码操作?

因为服务端存储的图片都是经过编码的图像文件，图片信息就包含在图像文件中，这样做的好处是体积小。而我们下载下来之后就是使用算法把图像文件转换为位图图像，位图图像占用体积较大，所以磁盘缓存不会直接缓存位图数据，而是解码压缩后的png或jpg数据。

还有一个原因就是在我们使用 UIImage 的时候，创建的图片通常不会直接加载到内存，而是在渲染的时候再进行解压并加载到内存。这就会导致 UIImage 在渲染的时候效率上不是那么高效。为了提高效率通过 decodedImageWithImage 方法把图片提前解压加载到内存，这样这张新图片就不再需要重复解压了，提高了渲染效率。这是一种空间换时间的做法。

参考： https://www.jianshu.com/p/5e742ad3876c https://www.jianshu.com/p/d527ff0c4950

问题二：并发请求同一url，如何保证实际上只进行一次网络请求?

在SDWebimageDownloder这个类中的下载方法里，有设置一个isFirst 的bool变量，调用下载方法的时候首先会在本地的url 字典里面查找这个url，如果有的话，就不再进行网络请求。

参考答案解析： https://blog.csdn.net/jacky_jin/article/details/73312810

问题三：图片的内容改变，url却没变，如何更新图片?

后台可以在给的url中增加时间戳字段，图片更新了就更新下这个字段。 客户端根据这个字段来设置是否从网络上请求

https://www.jianshu.com/p/d559cb3ca1b3

问题四：使用SDWebImage 下载高分辨率图片的时候会导致内存暴涨的解决办法?

```
  1、在控制器的+load方法中关闭decode操作，在delloc 中恢复
   load 方法：
   SDImageCache *canche = [SDImageCache sharedImageCache];
 SDImageCacheOldShouldDecompressImages = canche.shouldDecompressImages;
 canche.shouldDecompressImages = NO;

 SDWebImageDownloader *downloder = [SDWebImageDownloader sharedDownloader];
SDImagedownloderOldShouldDecompressImages = downloder.shouldDecompressImages;
downloder.shouldDecompressImages = NO;


delloc 方法：
-(void)dealloc {
SDImageCache *canche = [SDImageCache sharedImageCache];
canche.shouldDecompressImages = SDImageCacheOldShouldDecompressImages;

SDWebImageDownloader *downloder = [SDWebImageDownloader sharedDownloader];
downloder.shouldDecompressImages = SDImagedownloderOldShouldDecompressImages;

}
```

https://blog.csdn.net/guojiezhi/article/details/52033796

2、修改SDWebImage 框架中的UIImage+MultiFormat.m文件，添加一个等比压缩图片的方法，然后再+ (UIImage *)sd_imageWithData:(NSData *)data方法中，对一些大图进行压缩,再配合请缓存 [[SDImageCache sharedImageCache] setValue:nil forKey:@"memCache"];就可以了

https://www.jianshu.com/p/9356937ecad6

九、内存管理

iOS 是使用引用计数的机制来管理内存的。

自己生成的对象，自己持有；

非自己生产的对象，自己也可以持有

自己持有的对象不需要时，需要对其进行释放

非自己持有的对象无法释放

不论是ARC还是MRC 都遵循该方式，只是在ARC模式下这些工作被编译器做了

什么是ARC

ARC是automatic reference counting 自动引用计数，在程序编译时自动加入retain/release。在对象被创建时，retainCount +1，在对象被release时，retainCount -1，当count=0时，对象销毁

ARC管理原则

只要一个对象没有被强指针修饰就会被销毁，默认局部变量对象都是强指针，存放在堆里面，只是局部变量的强指针会在代码块结束后释放，对应所指向的内存空间也就销毁了，而set方法会多做影响引用计数方面的事情。

MRC管理原则

MRC 没有strong、weak，局部变量对象就是相当于基本数据类型。MRC给成员变量赋值一定要使用set方法，不能直接访问下划线成员赋值，因为使用下划线是直接赋值（如_name = name），

weak原理

系统对于每一个有弱引用的对象，都维护一个表来记录它所有的弱引用的指针地址。当一个对象的引用计数为0时，系统通过这张表找到所有的弱引用指针，把他们都置为nil。

weak和assign的区别

assign用来修饰基本数据类型，weak用来修饰OC对象，并且是一个弱引用。

为什么不用assign来修饰OC 对象，而用weak 呢?

因为被assign修饰的OC对象在被释放之后，指针指向的地址还存在，也就是会造成野指针；而weak修饰的对象在释放之后，指针地址会被置为nil。

__strong

__strong表示强引用，指向并持有该对象。

```
id __strong obj = [[NSObject alloc] init];
```

复制代码编译器会转换成下面代码：

```
id obj = objc_msgSend(NSObject, @selector(alloc));

objc_msgSend(obj, @selector(init));

// ...
objc_release(obj);
```

ARC下的Core Foundation对象的内存管理

使用Create、Copy、retain 的对象，都需要调用CFRelease(对象)或者free(对象)

Core Foundtaion 对象转为OC 时使用的关键字

__bridge：不修改引用计数，需要调用CFRelease
__bridge_retained:引用计数+1，需要调用CFRelease
__bridge_transfer：把引用计数交给arc管理，不需要调用CFRelease

AutoReleasePool

AutoReleasePool并没有单独的数据结构，而是由若干个以AutoreleasePoolPage为节点的双向链表组合而成的，每个Page 的大小是4096kBytes，也就是虚拟内存一页的大小(为什么是4096KBytes?其实就是虚拟内存每个扇区4096个字节,4K对齐的说法)

AutoreleasePoolPage

AutoreleasePoolPage 的内存结构包含 magic (用来校验AutoreleasePoolPage是否完整)、next(指向最新添加的autoreleased对象的下一个位置，初始化时begin())、thread（指向当前线程）、parent(指向父节点，第一个结点的parent值为nil)、child(指向子结点，最后一个结点的child值为nil)、depth （链表的深度，节点个数）、POOL_BOUNDARY （是一个边界对象nil，之前的源代码变量名是POOL_SENTINEL哨兵对象，用来区分每个page的边界）;当next = begin()时，表示page为空；当next==end()时，表示page已满

AutoreleasePool与runloop关系

```
AutoReleasePool与线程与runloop都是一一对应的关系
```

POOL_BOUNDARY

```
POOL_Boundary(边界对象)：它的作用是为了分割每次的push进Page的对象，返回的是一个内存地址，然后再pop的时候把一次push进来的对象全部释放掉
```

@AutoreleasePool{}

```
 @autoreleasePool使用Clang编译成C++文件后的实现如下：
 struct __AtAutoreleasePool {
__AtAutoreleasePool() {
atautoreleasepoolobj = objc_autoreleasePoolPush();
}
~__AtAutoreleasePool() {
objc_autoreleasePoolPop(atautoreleasepoolobj);
}
 void * atautoreleasepoolobj;
};


单个autoreleasePool的运行过程可以简单理解为objc_autoreleasePoolPush()、objc_autoreleasePoolPop()
```

objc_autoreleasePoolPush()

```
objc_autoreleasePoolPush()实际上是调用了autoreleasePoolPage的push()函数，push()函数里面又调用了autoreleaseFast(...)函数,这里面的判断逻辑是
1、当前page存在且没有满时，直接将对象添加到当前的page中；
2、当前page存在且已满时，创建一个新的page，并将对象添加到新创建的page中
3、当前page不存在时，创建第一个page，并将对象添加到新创建的page中
每次push后都会插入一个POOL_BOUNDARY，在执行Pop操作的时候，作为参数入参
```

objc_autoreleasePoolPop()

```
objc_autoreleasePoolPop （）实际上是调用了autoreleasePoolPage的pop(...)函数，
pop的入参就是push的返回值，也就是POOL_BOUNDARY的内存地址，找到token所在的page，对token之后的所有对象，发送release消息，然后销毁多余的page
releaseUntil函数的内部实现：

1、遍历所有的autorelease对象，直到遍历到POOL_BOUNDARY。
2、如果当前hotPage没有POOL_BOUNDARY，就将hotPage设置为父节点。
3、然后给当前的released对象发送release消息
4、然后再次配置hotPage
```

参考：https://blog.csdn.net/ZCMUCZX/article/details/80040910 http://blog.leichunfeng.com/blog/2015/05/31/objective-c-autorelease-pool-implementation-principle/

关于autoreleasePoolPage的一些详细博客： https://blog.csdn.net/ZCMUCZX/article/details/80040910 https://www.jianshu.com/p/03fb16ba1d31

循环引用

野指针

十、组件化与模块化

模块化

```
 第一层：底层网络请求及数据持久化

 第二层：三方库、推送和支付等

 第三层：业务相关

 第四层：UI
```

组件化

```
1、Target-action

2、Router

3、Protocol
```

十一、APP 的优化

代码健壮性

```
1、防崩溃保护
 1）使用AvoidCrash
 2）使用runtime 进行Method互换
2、使用try....catch
```

页面卡顿优化

```
1、tableView cell 高度缓存
2、异步计算
3、避免离屏渲染
4、减少View层级
```

耗电优化

资源优化

```
1、减少资源体积，删除无用资源

2、某些模块framework化，减少编译时间

3、资源压缩：png无损压缩、js/html压缩、音视频压缩

4、不常用的资源换为下载
```

编译优化

```
1、去除debug符号

2、开启编译优化

3、避免编译多个架构
```

十二、项目中遇到的问题及解决方案

十三、instructments的使用

Leaks

```
检测内存泄露和循环引用
```

Time Profiler

```
性能分析：分析耗时操作 、CPU 占有率
```

Allocations

```
用以检测内存分配
```

Zombies

```
检查是否访问了僵尸对象，但是这个工具只能从上往下检查，不智能
```

Energy Log

```
检测耗电量
```

十四、消息转发

OC 中的方法调用流程

```
  1、通过对象的isa指针找到对应的Class;
  2、在Class的方法列表中找到对应的seletor;
  3、如果在当前Class中未能找到seletor则往父类的方法列表中继续查找
  4、如果找到对应的selector，则去执行对象的方法实现（IMP）
  5、如果找到根类NSObject 中也没有找到对应的seletor，就会进入消息转发objc_msgSend(id,SEL)。
```

消息转发流程

```
   1、调用resolveInstanceMethod 或者 resolveClassMethod,判段类中是否有这个方法，若有，使用runtime的class_addMethod 动态的为添加方法实现；若没有进入下一步
   2、调用forwardingTargetForSelector 来将方法转发为其他对象类处理（实现代理模式），若返回为nl，表示不能处理，接下来就进入了完整的消息转发
   3、runtime把方法调用的所有细节封装到NSInvocation 对象中，然后转发给多个对象来处理该消息。先调用methodSignatureForSelector方法签名，然后forwardInvocation实现消息的派发（这个阶段其实实现了多重继承模式）
   4、若以上阶段都没有处理消息的话，则调用NSObject 的doesNotRecognizeSeletor方法来处理，从而抛出异常导致crash
```

消息转发应用

```
  1、实现@dynamic属性(为方法添加实现)
  2、实现代理模式（转发给别的对象实现）
  3、实现多重继承（转发给多个对象实现）
  4、也可用作防崩溃处理
```

小知识点

```
  1、- methodSignatureForSelector:消息获得函数的参数和返回值类型
  2、NSProxy 类是基于消息转发机制来实现的动态代理模式
  3、OC 中的方法调用是如何提高效率的?
    OC中引入了Class Cache（objc_cache），查找方法时，会先从缓存找，找不到再去Class 的方法列表中找。每个类都有一份方法缓存
  4、class_addMethod([self class], aSEL, (IMP)fooMethod, "v@:"); 这里第一字符v代表函数返回类型void，第二个字符@代表self的类型id，第三个字符:代表_cmd的类型SEL。这些符号可以在xcode的开发者文档中Type Encodings 中有解释
```

开发者文档关于类型符号：https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html#//apple_ref/doc/uid/TP40008048-CH100

参考：http://www.enkichen.com/2017/04/21/ios-message-forwarding/

十五、OC 与 JS 的双向交互

十六、统计原理

十七、谈谈APP 的安全与加密

加密

```
  1、MD5加密

  2、Hash加密

  3、Base64加密
```

安全

1、HTTP与HTTPS

2、TCP与UDP

3、代码混淆

4、APP 砸壳

十八、crash 的监测与统计

1、Bugly ：腾讯旗下的专业的carsh 监测和统计的工具,使用符号表对APP发生Crash的程序堆栈进行解析和还原。
2、友盟 和 bugly比较 ：
友盟比较慢，通常1-2天才能显示出来，bugly比较及时；
友盟分析过程比较复杂，需要下载crash 文件，然后下载友盟crash 分析工具，再使用命令行才能解析出来； buglg 只需要上传符号表（dysm文件）就可以了
3、本地收集使用NSSetUncaughtExceptionHandle来捕获到崩溃，把崩溃信息存储在本地，然后再下次启动的时候上传服务器
具体操作：1）使用系统NSException
2）使用xcode 自带工具aots 来符号化堆栈信息 ，参考：https://www.jianshu.com/p/7cdbaa4ba97c
4、通过itunes connect后台获取到用户上报的Crash日志。

十九、基础算法

冒泡排序

描述

比较相邻的两个元素，如果第一个比第二个大，就交换它们两个；针对所有元素重复以上比较，一次遍历只比较两个元素

复杂度 O（n^2）

选择排序

描述

首先找出最小（大）元素，存放到第一位，然后再从剩余元素中继续找出最小（大）元素，放到第二位，以此类推，知道排序结束

复杂度 O(n^2)，需要n-1趟才能排序完成，n表示序列的个数

插入排序

首先第一个元素默认是已排序的，取出下一个元素，作为待插入元素，在已经排序的元素序列中从后向前扫描，如果该元素大于待插入元素，就将带插入元素插入到该元素前面；然后依次类推取下一个元素继续跟已排序元素比较

快速排序

在序列中选择一个作为基准关键字（通常是第一个），然后遍历序列，所有比关键字小的都放到左边，比关键字大的都放大右边。然后再对这两部分分别进行快速排序

哈希表

哈希表是根据关键码值（key-Value）而直接进行访问的数据结构。也就是通过key-Value映射到表中的一个位置来访问记录，以加快查找速度

哈希冲突

两个不同的输入值，对应同一个输出值时，就会产生“碰撞”，即哈希冲突。

哈希冲突解决方法：

链地址法：就找hash表剩下空余的空间，找到空余的空间然后插入。

如果空间不足就不能用链地址法了。

开发定址法：在原地址新建一个空间，然后以链表结点的形式插入到该空间。

哈希函数比较全的介绍：https://www.cnblogs.com/s-b-b/p/6208565.html

二十、平时都遇到过哪些Crash

NSInvalidArgumentException(in'vao'li'd'a'g'ment') 异常：

```
    1、野指针访问：
     (1) 错用属性修饰词
     (2) 通知或者kvo 用完没有移除
     (3) block 回调之前没有判空直接调用
     (4) CoreFoundation层对象Toll-Free Bridging到Foundation层中，已经用了__bridge_transfer关键字转移了对象的所有权之后，又对CoreFoundation层对象调用了一次CFRelease、在objc_setAssociatedObject方法中该用OBJC_ASSOCIATION_RETAIN_NONATOMIC修饰的对象误用成OBJC_ASSOCIATION_ASSIGN
    2、找不到指定的方法：unrecognized selector sent to instance（runtime可以解决）
    3、tableView 返回空cell
    4、Array/Dictionary 初始化的时候插入了nil对象 （解决办法：可以使用runtime的swizzle method把nil对象转化为NSNull对象）
    5、NSJSONSerialization序列化的时候传入的data为nil（解决办法：判断非空）
```

SIGSEGV（c'g'sai'v 内存） 异常：

```
    1、可以使用Leaks 来检测内存泄露
    2、低内存时也会产生这种异常
    NSRangeException 异常
    1、 数组越界；
```

SIGPIPE 异常：

```
  对一个端已经关闭的socket调用两次write，第二次write将会产生SIGPIPE信号，该信号默认结束进程
  解决：写一段代码忽略这个信号（详见印象笔记）
```

基础题目：

一、Category 和 extention

Category

```
可以添加实例方法，不能添加实例变量，在运行期决议，方法的级别要高于原来的类
```

extention

```
 可以添加私有方法和实例变量，在编译期决议,必须得有一个类的源码才能为其添加extention，所以无法为系统的类（比如NSString）添加extention
```

Category的原理：

```
动态的将分类中的方法、属性以及协议数据放在Category_t结构体中，然后将结构体内的方法列表拷贝到类对象的方法列表中。
```

为什么分类不能添加实例?

```
 因为在运行期，对象的内存布局已经确定，如果添加实例变量就会破坏类的内部布局，这对编译型语言来说是灾难性的。
 还有一个原因是因为Category_t 结构体重并不存在成员变量
```

二、swift和OC的区别

1、swift 文件和结构更加简易化

2、OC 中的好多语法在swift中都是可以继续使用的，比如arc、block、属性、引用计数、接口、协议等，但是swift中的泛型和元组确实OC 中没有的

3、swift运行速度快，运算性能更高

4、swift 引用了可选类型(Options)，类似于OC 中的nil指针，适用于所有的数据类型，不仅仅是类

5、swift中有！和?

6、Swift 取消了Objective C 的指针/地址等不安全访问的使用

7、swift有属性监视器：willSet 和DidSet

========== 细节使用区别 ==========

1、swift不分.h和.m文件 ， 一个类只有.swift一个文件，所以整体的文件数量比起OC有一定减少。

2、swift句尾不需要分号 ， 除非你想在一行中写三行代码就加分号隔开。

3、swift数据类型都会自动判断 ， 只区分变量var 和常量let

4、强制类型转换格式不同 OC强转：(int)a Swift强转：Int(a)

5、关于BOOL类型更加严格 ， Swift不再是OC的非0就是真，而是true才是真false才是假

6、swift的 循环语句中必须加{} 就算只有一行代码也必须要加

7、swift的switch语句后面可以跟各种数据类型了 ， 如Int、字符串都行，并且里面不用写break（OC好像不能字符串）

8、swift if后的括号可以省略: if a>b {}，而OC里 if后面必须写括号。

9、swift打印 用print("") 打印变量时可以 print("(value)")，不用像OC那样记很多%@，d%等。

10、Swift3的【Any】可以代表任何类型的值，无论是类、枚举、结构体还是任何其他Swift类型，这个对应OC中的【id】类型。

三、ViewController 的生命周期

```
init
loadView
viewDidLoad
viewWillAppear
viewDidAppear
viewWillDisappear
viewDidDisappear
dealloc
```

四、delegate、block和NSNotification的区别

delegate

```
一对一；效率高，更直接，面向过程，只能单向传值，用weak修饰
```

block

```
面向结果，可以双向传值，操作简单，用copy修饰，可以捕获自动变量
```

KVO

```
一对多
```

NSNotification

```
多对多；不关心结果，模块直接联系不紧密
```

五、APPDelegate的生命周期

点击程序图标启动：

```
applicationDidFinishLaunching

applicationDidBecomeActive(进入活动状态)

applicationDidReceivementoryWarning （内存警告，程序将要终止）

applicationWillTerminate (将哟啊退出结束)
```

点击home键，进入后台：

```
applicationWillResignActive(将要进入非活动状态)

applicationDidEnterBackground(进入后台)
```

再点击图标进入程序：

```
applicationWillEnterForground(将要重新进入前台)

applicationDidBecomeActive(进入活动状态)
```

六、数据缓存（数据持久化）

```
1、NSUserDefault
2、CoreData
3、SQLite
4、writeToFile 写入方式：包括写入plist文件和.text文件,归档
5、FMDB ，扩展FMDB 原理
 FMDB 是基于sqlite的封装，采用的是子线程异步调用的方式
```

七、__block 和 __weak 的区别

```
修饰对象：
        __block可以修饰对象和基本数据类型；
        __weak 只能修饰对象
arc/mrc环境：
        __block 既能用于arc,也能用于mrc
        __weak 只能用于arc
```

八、字典的底层实现原理

字典使用hash表存储的

九、NSArray的子类都有哪些?

参考： https://www.aopod.com/2017/02/24/class-clusters/

```
1、alloc后所得到的类为__NSPlaceholderArray。
2、当init为一个空数组后，变成了__NSArray0
3、如果有且仅有一个元素，那么为__NSSingleObjectArrayI
4、如果数组大于一个元素，那么为__NSArrayI
代码示例如下：
NSArray *placeholder = [NSArray alloc];
NSArray *arr1 = [placeholder init];
NSArray *arr2 = [placeholder initWithObjects:@0, nil];
NSArray *arr3 = [placeholder initWithObjects:@0, @1, nil];
NSArray *arr4 = [placeholder initWithObjects:@0, @1, @2, nil];

NSLog(@"placeholder: %s", object_getClassName(placeholder));    // placeholder: __NSPlaceholderArray
NSLog(@"arr1: %s", object_getClassName(arr1));                  // arr1: __NSArray0
NSLog(@"arr2: %s", object_getClassName(arr2));                  // arr2: __NSSingleObjectArrayI
NSLog(@"arr3: %s", object_getClassName(arr3));                  // arr3: __NSArrayI
NSLog(@"arr4: %s", object_getClassName(arr4));                  // arr4: __NSArrayI
```

十、定义NSString的属性为什么要用copy修饰?

为了防止把一个可变字符串在未使用copy方法时赋值给这个字符串对象时，修改原字符串时，本字符串也会跟着被动修改的情况发生。

十一、为什么说Object-C是一门动态的语言?

1、objcect-c类的类型和数据变量的类型都是在运行时确定的，而不是在编译期确定。比如多态特性，我们可以使用父类的指针来指向子类对象，并且可以用来当调用子类的方法
2、运行时特性，我们可以动态的添加方法，或者替换方法。

十二、说一下OC 的NSObject对象

NSObject内部有一个Class类型的isa指针，Class是一个objc*class结构体类型,objc*class结构体如下：

```
struct objc_class {
Class _Nonnull isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
Class _Nullable super_class                              OBJC2_UNAVAILABLE;
const char * _Nonnull name                               OBJC2_UNAVAILABLE;
long version                                             OBJC2_UNAVAILABLE;
long info                                                OBJC2_UNAVAILABLE;
long instance_size                                       OBJC2_UNAVAILABLE;
struct objc_ivar_list * _Nullable ivars                  OBJC2_UNAVAILABLE;
struct objc_method_list * _Nullable * _Nullable methodLists                    OBJC2_UNAVAILABLE;
struct objc_cache * _Nonnull cache                       OBJC2_UNAVAILABLE;
struct objc_protocol_list * _Nullable protocols          OBJC2_UNAVAILABLE;
#endif

 } OBJC2_UNAVAILABLE;
```

参数解析：

isa指针

isa指针是和Class同类型的objc_class结构指针，类对象的指针指向其所属的类，即元类。元类中存储着对象的类方法。对象通过对象的isa指针指向所属类

super_class指针

指向该类所继承的父类对象，如果该类已经是最顶层的根类（如NSObjec或NSProxy），则super_class为NULL

name

类名

version

类的版本信息，默认是0。这对于对象的序列化非常有用，它可以让我们识别出不同类定义版本中实例变量布局的改变

info

供运行期使用的一些位标识

instance_size

存储该类的实例变量大小

ivars

是objc_ivar_list类型的指针， 存储每个实例变量的内存地址数组

methodLists

是指向objc_method_list指针的指针，存储的方法列表，根绝info 的信息确定是类方法还是实例方法，运行什么函数方法等

cache

是一个objc_cache的指针，用于缓存最近使用的方法。一个接受者对象接收到一个消息时，它会根据isa指针去查找能够响应这个消息的对象。在实际使用中这个对象只有一部分方法是常用的，很多方法其实很少用或者根本用不上。这种情况下，如果每次消息来时，我们都去methodLists中遍历一遍，性能势必很差。这时，cache就派上用场了。在我们每次调用过一个方法后，这个方法就会被缓存到cache列表中，这次调用的时候runtime就会优先去cache中查找，如果cache中没有，才会去methodLists中查找方法。这样，对于那些经常用到的方法的调用，就提高了调用的效率

protocols

```
是一个objc_protocol_list的指针
```

一个NSObject类型的对象，在内存中占16个字节。