swift 语言特性

swift与OC的区别

函数式编程

swift、OC都是编译型动态语言，只是他们的编译方式不一样；swift更注重安全性，是强类型语言；OC更注重灵活性；swift有函数式编程、面向对象、面向协议编程，OC只有面向对象编程；swift更注重值类型，OC更遵循指针和索引。 1、数据结构：

- swift将String、Array、Dictionary设计成值类型，OC是引用类型。相较而言，1)、值类型更高效使用内存，它是在栈上操作，引用类型在堆上操作；2)、通过let和var来确认String、Array、Dictionary是可变还是不可变，让线程更加安全；3)、也让String可以遵循Collection这种协议，增加了灵活性
- 初始化的差别：swift的初始化更加严格准确，swift必须保证所有非optional的成员变量都完成初始化，同时新增convenience(便利初始化方法，必须通过调用同一个类中的designed初始化方法来完成)和required（强制子类重写父类所修饰的初始化方法）初始化方法
- swift的protocol协议更灵活，它可以对接口进行抽象，例如Sequence，配合extention、泛型、关联类型实现面向协议编程，同时它还可以用于值类型、如结构体和枚举 2、语言特性： swift中，协议是动态派发，扩展是静态派发，也就是说如果协议中有方法声明，那么方法会根据对象的实际类型进行调用

```
protocol Chef{
    func makeFood()
}
extension Chef{
    func makeFood(){
        print("Make food")
    }
}

struct SeafoodChef:Chef{
    func makeFood(){
        print("cook seafood")
    }
}

let oneC:Chef = SeafoodChef()
let twoC:SeafoodChef = SeafoodChef()
oneC.makeFood()
twoC.makeFood()
//这里oneC和twoC实际上都是SeafoodChef类型，按照上述原则，这里会打印两个"cook seafood"。假如protocol中没有声明makeFood()，那么第一行打印的就会是"Make food"，因为没有声明的话，只会按照声明类型进行静态派发，也就是说oneC被声明成了Chef类型，所以oneC会调用扩展中的实现。
```

Q1、类和结构体的区别： 类是引用类型，结构体是值类型。类可以继承、运行时类型转换、用deinit释放资源、可以被多次引用；Struct结构小，适用于复制操作，相较引用更安全，无须担心内存泄漏和多线程冲突问题。 Q2、weak和unowned的区别weak和unowned的区别QQ2： 当访问对象可能已经被释放时，使用weak，例如delegate；当访问对象确认不可能被释放时，则用unowned，比如self的引用；实际上，为了安全，基本上都是使用weak。 Q3、如何理解copy-on-write: 当值类型进行复制时，实际上复制的对象和原对象还是处于同一个内存中，当且仅当修改复制后的对象时，才会在内存中重新创建一个新对象。这样是内存使用更高效。 Q4、初始化方法对属性的设定以及willSet和didSet里对属性设定都不会触发属性观察。

APP常用架构

MVC

Model —— View —— Controller，Controller负责协调model和View；

MVVM

简单介绍一下MVVM框架及ViewModel作用 说到MVVM之前，首先要先介绍一下MVC框架，MVC框架就是Model-View-Controller组成，其中Model负责呈现数据，View负责UI展示，Controller则负责调解Model和View直接的交互。这样就导致了大部分的处理逻辑都在Controller当中，所以它又被称为“重量级视图控制器”。而MVVM框架则表示Model--ViewModel--（View Controller），它其实就是对MVC的一个优化而已，它将业务逻辑、网络请求和数据解析放在了ViewModel层，大大简化了Controller层的逻辑代码，也让model 、view的功能更加独立单一。

MVP

MVP是在MVC的基础上，将Controller职责分离开，只用它处理View的交互事件、数据绑定等。Presenter被用来沟通View和Model之间的联系，Model不能直接作用于View的更新，只能通过Presenter来通知View进行视图刷新，所以View就只专注于视图相关内容，被动接收Presenter的命令。这样的话，View就只显示，不处理逻辑，Presenter持有Model，Model只用于处理数据相关内容；

模块化、组件化：路由化

模块化

项目的整体架构不止是MVC、MVP这种代码层面的东西，而应该是更高维度的规划。比如说对项目进行分层，分层的意义在意是项目模块化。从底层到上层一次是：独立于APP的通用层、通用业务层、中间层、业务层：

- 独立于APP的通用层：这一层主要是放一些跟APP耦合不是很大的模块，比如我们BT学院的BTCore模块，包括网络请求封装模块、各种category、自定义的一些UI组件等；
- 通用业务层：这个则是针对APP的一些基础模块，比如皮肤相关、接口请求通用处理及日志相关、APP中多个业务模块用到的一些通用组件等；它主要是给各个业务层提供一些通用的业务类代码，比如BT学院的Loading、日志、测试工具、缓存等；
- 中间层：中间层的作用则是协议各个业务层的通信，同时让业务与业务之间解耦；比如BT学院的各个Router；
- 业务层：则是各个单独的业务模块。比如BT学院的题库、问答、学习等；

模块化的优势：各模块直接代码和资源相互独立，模块可以独立维护、测试等。实现简单的插拔式。 其次模块化：主要是有两个方式： 1、通过cocoapod的方案将各个主代码模块打包成pod包的形式。然后通过配置podsepc来进行模块以及库直接的依赖。但是会存在很大问题，一个主要 的是文件夹只有一层，没法做分级。2是库循环依赖问题。主流的方式还是使用pod的方式。 2、使用cocoa touch framework。主要注意的点是混编时，对外的头文件，尤其是swift中使用到的OC头文件放到public中，因为framework不支持bridge；framework中的内核架构

我们模块化的具体实施则是由develop pod的方式进行的。

组件化

组件可以分为基础组件、业务组件：

- 基础组件：比如各种三方库，自己封装的库，自己二次封装的库，业务开发时单独功能的UI框架，比如相册取照片、视频播放器等。
- 业务组件：业务也可以搞成单独的组件，使用pod来进行管理。 这些可以独立出来的都可以算是组件，部分大小；

组件通信

组件通信是中间层的主要内容，是为了解耦各个组件的。 组件通信方案一般有三种：

- URL Router： 在前端，一个url表示一个web页面，在后端，一个url也表示一个接口请求；在iOS中，也会使用官方提供url去打开一个系统设置。所以同样的，我们可以使用url来表示一个控制器、一个组件、甚至一个控件； 蘑菇街的MGJRoutre就是使用这种方案，它使用一个全局的router来管理对应的key和value，key是url，value是对应的对象。获取到对象后进行处理

```
注册路由
[[Router sharedInstance] registerURL:@"myapp://good/detail"
with:^UIViewController *{
     return [GoodDetailViewController new];
}];
通过url获取
UIViewController *vc = [[Router sharedInstance]
openURL:@"myapp://good/detail"]
```

我们则是在MGJRouter的基础之上封装了一个BTRouter放在中间层，然后handler里使用根Root来执行跳转，只将结果block返回。然后每个业务模块对应一个Router分类， 尽可能解耦业务直接的联系。然后会根据本地url和远程url来进行参数的解析。 router的命名管理就显得尤其重要。

缺点就是要在使用前进行注册和内存占用；

- Target Action： 这种方式则主要是利用iOS的反射机制，通过NSClassFromString来生成target类，然后再通过Runtime或者performSelector执行target的action，在action中进行目标类的实例化操作。利用这种机制，可以将任意类的实例化过程封装到任意一个target中，相比于URL Router，它无需注册和内存占用，缺点就是编译阶段是无法发现潜在的问题，对命名规则就更严格。 这种方案的开源框架就是CTMediador。

https://juejin.im/post/5ccfd378e51d453b6c1d9cf5 组件间通信 组件化及其通讯方案

图片与SDWebImage

图片 内存、解码相关：

图片加载

iOS 提供了UIImage用来加载图片，提供了UIImageView用来显示图片；

- imageNamed：可以缓存已经加载的图片。使用时会根据文件名在系统缓存中寻找图片，如果找到了就返回，如果没有找到就在Bundle内查找文件名，找到后将其放到UIImage里返回，**并没有进行实际的文件读取和解码**。当UIImage第一次显示到屏幕上时，其内部解码方法才会被调用，同时解码结果会保存到一个全局的缓存中。这个全局缓存会在APP第一次退到后台和收到内存警告时才会被清空。
- imageWithContentsOfFile：方法则是直接返回图片，不会进行缓存。但是其解码依然要等到第一次显示该图片的时候；

解码

在UI的显示原理中，CALayer负责显示和动画操作相关内容，其中CALayer的属性contents是对应一张CGImageRef的位图。**位图**实际上就是一个像素数组，数组中的每个像素就代码图片中的一个点。**Image Buffer**就是内存中用来存储位图像素数据的区域； 而项目中无论是网络下载还是本地的图片，基本都是JPEG、PNG等类型格式的压缩图片。 其中png是图片无损压缩格式，支持alpha通道。JPEG是图片有损压缩格式，可以指定0~100%的压缩比。所以如果要设置图片alpha，就只能用png格式。而jpeg则更小，但是也就损失了图片质量；**Data Buffer**就是用来存储JPEG、PNG格式图片的元数据，对应着源图片在磁盘中的大小； **解码就是将将不同格式的图片转码成图片的原始像素数据（Image），然后绘制到屏幕上**。

UIImage就负责解压Data Buffer内容并申请Image Buffer存储解压后的图片信息； UIImageView就负责将Image Buffer拷贝至frame Buffer(帧缓存区)，用于屏幕上显示；

> ImageBuffer按照每个像素RGBA四个字节大小，一张1080p的图片解码后的位图大小是1920 * 1080 * 4 / 1024 / 1024，约7.9mb，而原图假设是jpg，压缩比1比20，大约350kb，可见解码后的内存占用是相当大的。

图片相关优化

降低采样率(DownSampling)

在视图比较小，但是图片缺较大的场景下，直接显示原图会造成不必要的内存和CPU消耗。这里就可以使用ImageIO的接口，DownSampling，也就是生成缩略图

```
    // 获取缩略图
    func downSample(imageAt url:URL, to size:CGSize, scale:CGFloat) ->
UIImage {
        //避免缓存解码后的数据，因为这个是缩略图，之后的使用场景可能就不一样，所以不要做缓
存。
        let imageSourceOptions = [kCGImageSourceShouldCache : false] as
CFDictionary
        let imgSource = CGImageSourceCreateWithURL(url as
CFURL,imageSourceOptions)!

        let maxDimendionInPixels = max(size.width,size.height) * scale
```

```
        //kCGImageSourceShouldCacheImmediately设为YES，则就立马解码，而不是等到渲染的时候才解码
        let downsampleOptions = [kCGImageSourceCreateThumbnailFromImageAlways : true,
                                 kCGImageSourceShouldCacheImmediately : true,
                                 kCGImageSourceCreateThumbnailWithTransform : true,
                                 kCGImageSourceThumbnailMaxPixelSize : maxDimendionInPixels] as CFDictionary
        let downsampledImage = CGImageSourceCreateThumbnailAtIndex(imgSource,0,downsampleOptions)!
        return UIImage(cgImage:downsampledImage)
    }
```

将解码过程放到异步线程

解码放在主线程一定会造成阻塞，所以应该放到异步线程。 iOS 10之后，UITableView和CollectionView都提供了一个预加载的接口:tableView(_ : prefetchRowsAt:) 提前为cell加载数据。

```
    let serailQueue = DispatchQueue(label: "decode queue")
    func tableView(_ tableView: UITableView, prefetchRowsAt indexPaths: [IndexPath]) {
        for index in indexPaths {
            serailQueue.async {
                let downSampledImg = "" //解码操作
                DispatchQueue.main.async {
                    self.update(at:index,with:downSampledImg)
                }
            }
        }
    }
    //这里使用串行队列，避免开启多个线程，因为线程消耗也是很大的
```

平时UI代码注意的细节点

- 重写drawRect:UIView是通过CALayer创建FrameBuffer最后显示的。重写了drawRect，CALayer会创建一个backing store，然后在backing store中执行draw函数。而backing store默认大小与UIView大小成正比的。存在的问题：backing store的创建造成不必要的内存开销；UIImage的话先绘制到Backing store，再渲染到frameBuffer，中间多了一层内存拷贝；
- 更多使用Image Assets：更快地查找图片、运行时对内存管理也有优化；
- 使用离屏渲染的场景推荐使用UIGraphicsImageRender替代UIGraphicsBeginIMageContext，性能更高，并且支持广色域。
- 对于图片的实时处理，比如灰色值，这种最好推荐使用CoreImage框架，而不是使用CoreGraphics修改灰度值。因为CoreGraphics是由CPU进行处理，所以使用CoreImage交由GPU去做；

正确的图片加载方式

类似SDWebImage流程

下载图片主要流程：

- 1、从网络下载图片源数据，默认放入内存和磁盘缓存中；
- 2、异步解码，解码后的数据放入内存缓存中；
- 3、回调主线程渲染图片；
- 4、内部维护磁盘和内存的cache，支持设置定时过期清理，设置内存cache的上限等

加载图片流程简化：

- 1、从内存中查找数据，如果有，并且已经解码，直接返回数据，如果没有解码，异步解码缓存内存后返回；
- 2、内存中未查找到图片数据，从磁盘查找，磁盘查找到后，加载图片源数据到内存，异步解码缓存内存后返回，如果没有去网络下载图片，走上面的流程；

总结：这个流程就主要避免了在主线程中解码图片的问题；然后通过缓存内存的方式，避免了频繁的磁盘IO；缓存解码后的图片数据，避开了频繁解码的CPU消耗；

超大图片处理

如果是非常大的图，比如1902 * 1080，那解码之后的大小就达到了近7.9mb。像上述的图片加载方案或者SDWebImage的加载方式，默认就会自动解码缓存，那么如果有连续多张的情况，那内存将瞬间暴涨，甚至闪退。 那解决方案就分为两个场景：

- 如果显示的UIView较小，则应该通过上述降低采样率的方式，加载缩略图；
- 如果是那种像微信、微博详情那样的大图，则应该全屏加载大图，通过拖动来查看不同位置图片的细节。技术细节就是使用苹果的CATiledLayer去加载，它可以分片渲染，滑动时通过映射原图指定位置的部分图片数据解码渲染。

iOS图像最佳实践 周小可—图片的编码与解码

image/io

图片渲染相关

ios绘制

SDWebImage：

源码架构与基础流程

- 架构简述： SDWebImage是通过给UIImageView写的一个分类：UIImageView + WebCache；而支撑整个框架的核心类是SDWebImageManager。它通过管理SDWebImageDownloader和SDImageCache来协调异步下载和图片缓存。
- 流程 基于4.3版本，5.0之后的版本改成了面向协议的方式，但是主要流程和类还是没什么大的改变的

```
1、入口函数会先把placeholderImage显示，然后根据URL开始处理下载；
2、进入下载流程后会先使用url作为key值去缓存中查找图片，如果内存中已经有图片，则回调返回展示（这里就不会再管磁盘有没有的情况了）；
3、如果没有找到，则会生成一个Operation进行磁盘异步查找，如果找到了会进行异步解码，解码完成后将结果回调，同时会同步到缓存中去。
4、如果没有在本地找到，则会生成一个自定义的Operation开启异步下载。
5、下载完成后，将下载的结果进行解码处理，然后返回。同时将图片保存到内存和磁盘。
```

SDWebImage原理 周小可—SDWebImage源码解析

注意的细节与常考点，与上述流程对应

- 第1步中：会先取消当前正在进行加载的Operation，设置placeholder。然后根据URL开启下载。整个库中的key值默认使用图片URL，比如缓存、下载操作等。URL中可能会含有一部分动态变化的部分（比如获取权限的部分），所以我们可以取url不变的值scheme、host、path作为key值；
- 第2、3步中：首先会判断是否只使用了内存查找，如果是的话，则不进行磁盘查找，也不将查找的图片存到磁盘；否的话会先生成一个NSOperation赋值给SDWebImageCombinedOperation的cacheOperation，用于cancel。然后会封装一个block来执行磁盘查找，block根据设置来确实是同步还是异步查找，如果是异步查找的话，会放到一个串行的IO队列中。在查找期间会先判断Operation是否取消，如果已经取消则不进行查找。查找的过程中会创建一个@autoreleasePool用来及时释放内存；如果在磁盘中找到了data，那么会将data解码成Image,并同时存一份到内存中，如果内存空间过小，则会先清一波内存缓存；
- 第4步中：1、每张图片的下载是由自定义的NSOperation的子类进行的，它实现了start方法。start方法中创建了一个NSURLSession开启下载，使用了RunLoop来确保从start到结果响应期间不会被干掉，如果运行后台下载的话，也是在这里进行处理的。2、Operation被放到一个NSOperationQueue中并发执行，队列中的最大并发量是6；DownloadQueue使用了信号量来确保线程安全；3、每个Operation、结果回调block、进度block都是包装存储到一个URLCallBack中的，它以url为key值缓存在一个NSMutableDictionary的字典中，以便cancel及其他操作。但是因为可能存在多个操作同时进行的情况，所以这里就使用了dispatch_barrier来确保NSMutableDictionary的线程安全；4、下载过程中，如果返回了304 not Modified，则表示客户端是有缓存的，则可以直接cancel掉Operation，返回回调返回缓存的image。
- 第5步：下载完成后，会在URLSessionTaskDelegate的回调方法里使用一个串行队列异步进对下载图片进行解码。解压完成后，如果是JPEG这种可压缩格式的图片则会按照设置进行压缩后再返回。如果有缩略设置，也会对图片进行缩放等；
- 其他：
  - SDWebImageCombinedOperation：它实际上不是一个NSOperation，它只是持有了downloadOperation和cacheOperation（真实的NSOperation类型），downloadOpetation对应着SDWebImageDownloadToken类型，它包含着一个SDWebImageDownloaderOperation和url，也就是URLSession的实际下载操作；
  - 内存缓存使用的是NSCache的子类。NSCache是类似NDDictionary的容器，它是线程安全的，所以在任意线程进行添加、删除都不需要加锁，而且在内存紧张时会自动释放一些对象，存储对象时也不会对key值进行copy操作。SDImageCache在收到内存经过或退到后台的时候会清理内存缓存，应用结束时会清理过期图片；
  - 磁盘缓存使用的是NSFileManager来实现的。图片存储的位置位于Cache文件夹，文件名是对key值进行MD5后的值，SDImageCache定义了一个串行队列来对图片进行异步写操作，不会对主线程造成影响；存到磁盘的同时会检查是jpeg还是png（这里主要是通过alpha通道来

判断的），然后将其转成对应的压缩格式进行存储；读取磁盘缓存也会先从沙盒中读取，然后再从bundle中读取，读取成功后才进行转换，转换过程中先转成image，然后根据设备进行@2x、@3x缩放，如果需要解压缩再解压缩，之后才是后续解码操作。
    - 清理磁盘缓存可以选择全部清空和部分清空。全部清空则是把缓存文件夹删除，部分清空会根据参数设置来判断，主要看文件缓存有效期和最大缓存空间，文件默认有效期是1周；文件默认缓存空间大小是0，也就是表示可以随意存储，如果设置了的话，则会先判断总大小是否已经超出最大值，如果超出了，则优先保留最近最先使用的图片，递归删掉其他过早的图片，直到总大小小于最大值。
    - 使用主队列来代替是否在主线程的判断；
    - 后台下载：使用`-[UIApplication beginBackgroundTaskWithExpirationHandler:]`方法使 app 退到后台时还能继续执行任务, 不再执行后台任务时，需要调用`-[UIApplication endBackgroundTask:]`方法标记后台任务结束

- 框架中使用最大的锁是dispatch_semaphore_t，其次是@synchronized互斥锁；

SDWebImage相关面试题

SDWebImage源码阅读笔记

Crash 及 APP 性能监控相关

面试常考题：介绍你碰到过的印象较深刻的外网crash，并介绍发现、定位、解决的过程。

Crash相关

常见的Crash及处理

- 找不到方法：unrecognized selector sent to instance： eg：比如说未实现的代理方法；对可变集合使用了copy后，对其进行修改操作。 解决方案：给NSObject写个分类，截获这种未实现的方法：

```
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector    {
    if ([self respondsToSelector:aSelector]) {
    //已经实现不做处理
        return [self methodSignatureForSelector:aSelector];
    }
    return [NSMethodSignature signatureWithObjCTypes:"v@:"];
}
- (void)forwardInvocation:(NSInvocation *)anInvocation {
    NSLog(@"在 %@ 类中, 调用了没有实现的实例方法: %@
",NSStringFromClass([self
class]),NSStringFromSelector(anInvocation.selector));
}
```

  其次，尽量少使用performselector这种API；

- KCV造成的Crash： eg:给NSObject添加KVC；key为nil；key不存在。一句话：**给不存在的key（包括nil）设置value**； 解决方案：重写类的setValue:forUndinedKey:和valueForUndefinedKey:

```
-(void)setValue:(id)value forUndefinedKey:(NSString *)key{
    NSLog(@"在 %@ 类中给不存在的key设置value",NSStringFromClass([self
class]));
}
-(id)valueForUndefinedKey:(NSString *)key{
    return nil;
}
```

- EXC_BAD_ACCESS: eg:使用没有实现的block；对象未初始化；访问野指针(比如assign、unsafe_unretained修饰对象类型，关联属性修饰使用不对)；地址越界； 解决方案：这类对象大多数是由于操作不当导致的，所以可以利用xcode的一些工具
    - 1、使用XCode，在Debug模式下开启僵尸模式，Release时关闭；
    - 2、使用XCode的Address Sanitizer检查地址访问越界；
    - 3、对象属性修改方式要使用正确；
    - 4、使用block要先做判断；
- KVO造成的Crash： eg:观察者是局部变量；被观察者是局部变量；没有实现observeValueForKeyPath:；重复移除观察者；另外需要注意一个不会崩溃的现象：如果重复添加观察者，不会导致崩溃，但是一次改变会被观察多次； 解决方案：
    - 1、尽可能合理使用；
    - 2、add和remove一定要成对出现；
- 集合类Crash： eg:数组越界；向数组添加nil对象；在数组遍历时进行移除操作；字典使用setObject:forKey:时，key为nil；字典使用setObject:forKey:时，objct为nil;字典使用setValue:forKey:时，key为nil。这里要注意setValue和setObjct的区别：

```
NSMutableDictionary * dic = [NSMutableDictionary dictionary];
 [dic setObject:nil forKey:@"1"]; //crash
 [dic setValue:nil forKey:@"1"]; // not crash
 [dic setObject:@"1" forKey:nil]; //crash
 [dic setValue:@"1" forKey:nil]; //crash
```

  解决方案：
    - 1、可以给集合类添加category，method swizzling原来方法，判断后再处理；
    - 2、使用可变字典添加元素时，尽可能使用setValue:forKey:
    - 3、NSMutableArray、NSMutableDictionary不是线程安全的，所以在多线程下要保证写操作的原子性。可以使用加锁、信号量、串行队列、dispatch_barrier_async等；或者使用NSCache代替；
- 多线程Crash： eg:子线程更新UI；Dispatch_Group中level比enter次数更多；Dispatch_semaphore使用时重新复制或置空；多线程下非线程安全的可变集合的使用； 解决方案：
    - 1、使用集合时要确保操作的原子性；可以使用加锁、信号量、串行队列、dispatch_barrier_async等；或者使用NSCache代替；
    - 2、熟练使用Dispatch_Group、Dispatch_semaphore等；
    - 3、多线程发送Crash时，会收到SIGSEGV信号，表面视图访问未分配给自己的内存、或视图往没有写权限的地址写数据；

- Socket造成的Crash： 原因：当服务器close一个连接时，若client端接着发数据。根据TCP协议的规定，会收到一个RST响应，client若再往这个服务器发送数据时，系统会发出一个SIGPIPE信号给进程，告诉进程这个连接已断开，不要再写数据了。而根据信号的默认处理规则，SIGPIPE信号的默认执行动作是terminate(终止、退出),所以client会退出。 eg：长连接socket或重定向管道进入后台，没有关闭导致崩溃的解决方法； 解决方案：
    - 1、切换到后台是，关闭长连接和管道，回到前台重新创建；
    - 2、使用signal(SIGPIPE,SIG_IGN)将SIGPIPE设置为SIG_IGN,相当于将SIGPIP交给系统处理，客户端不执行默认操作，即不退出。
- Watch Dog超时造成的Crash： 原因：主线程执行耗时操作，导致主线程被卡超过一定时间。一般异常编码是0x8babf00d，表示应用发送超时而被iOS系统终止。 eg:长期卡顿导致崩溃； 解决方案： 尽可能将耗时操作异步放到后台线程。主线程只负责更新UI和事件响应。
- 后台返回NSNull导致的崩溃： eg:后台返回NSNull，解析完成后，使用时会crash；

```
NSNull *nullStr = [[NSNull alloc] init];
NSMutableDictionary* dic = [NSMutableDictionary dictionary];
[dic setValue:nullStr forKey:@"key"]; //not crash
NSNumber* number = [dic valueForKey:@"key"];// crash 相当于调用了get方法，会报“unrecognized selector”
```

  解决方案：NSNull用于OC对象的占位，一般会作为集合类型的占位元素，给NSNull对象发送消息会crash。
    - 可以使用NullSafe第三方库。

[iOS中常见Crash总结]

Crash捕获

iOS的主要崩溃分为三大类，一类是OC抛出的Exception异常，可以通过注册NSUncaugthExceptionHandler来捕获；一类是Mach异常，比如说野指针访问、线程问题，这一类异常会被转换成Signal信号，可以通过注册signalHandler来捕获。还有一类则是无法使用信号和Exception捕获的，比如像后台任务超时、内存被爆、主线程卡顿超阈值等。

前两类异常的捕获：

这里需要注意的是，避免与Bugly这种工具冲突覆盖掉handler的问题，所以使用之前要先进行判断。再处理完自己的handler之后再抛出去。

```
//一、OC异常处理函数
// OC层中未被捕获的异常，通过注册NSUncaughtExceptionHandler捕获异常信息
void InstallUncaughtExceptionHandler(void) {
//注册
    if(NSGetUncaughtExceptionHandler() != custom_exceptionHandler)
    oldhandler = NSGetUncaughtExceptionHandler();
    uncaught_exception_handler(&custom_exceptionHandler);
}
static void uncaught_exception_handler (NSException *exception) {
    //获取exception的异常堆栈，NSThread也对应一个类方法的callStackSymbols
    NSArray *stackArray = [exception callStackSymbols];
```

```
    //出现异常的原因
    NSString *reason = [exception reason];
    //异常名称
    NSString *exceptionInfo = [NSString stringWithFormat:@"Exception reason:
%@\nException name: %@\nException stack: %@",
                               name,
                               reason,
                               stackArray];
    NSMutableArray *tmpArr = [NSMutableArray arrayWithArray:stackArray];
    [tmpArr insertObject:reason atIndex:0];
    //保存到本地，以便下次启动查看
    [exceptionInfo writeToFile:[NSString
stringWithFormat:@"%@/Documents/error.log",NSHomeDirectory()]
                    atomically:YES
                      encoding:NSUTF8StringEncoding
                         error:nil];
}

//二、Unix标准的signal机制处理函数
// 内存访问错误，重复释放等错误就无能为力了，因为这种错误它抛出的是Signal，所以必须要专门
做Signal处理。 OC中层不能转换的Mach异常，利用Unix标准的signal机制，注册SIGABRT,
SIGBUS, SIGSEGV等信号发生时的处理函数。
void registerSignalHandler(void) {
    signal(SIGSEGV, handleSignalException);
    signal(SIGFPE, handleSignalException);
    signal(SIGBUS, handleSignalException);
    signal(SIGPIPE, handleSignalException); //这个就是上文中socket长连接,服务器关
闭之后系统发出的终止进程的信号，如果想不退出，则可以signal(SIGPIPE, SIG_IGN)，然后重定
向服务器。
    signal(SIGHUP, handleSignalException);
    signal(SIGINT, handleSignalException);
    signal(SIGQUIT, handleSignalException);
    signal(SIGABRT, handleSignalException);
    signal(SIGILL, handleSignalException);
    }

void handleSignalException(int signal) {
    NSMutableString *crashString = [[NSMutableString alloc]init];
    void* callstack[128];
    int i, frames = backtrace(callstack, 128);
    char** traceChar = backtrace_symbols(callstack, frames);
    for (i = 0; i <frames; ++i) {
        [crashString appendFormat:@"%s\n", traceChar[i]];
    }
    NSLog(crashString);
}
```

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    InstallUncaughtExceptionHandler();
    registerSignalHandler();
    return YES;
}
```

对于无法捕获的崩溃怎么处理

- 后台崩溃： 当程序被退到后台后，只有几秒的时间可以执行代码，接着会被挂起，挂起后会暂停所有线程。但是如果是数据读写的线程则无法暂停只能被中断，中断的话，系统会主动杀掉APP，而且中断时数据容易被损坏，。 APP退到后台后，默认是使用Background Task方式，就是系统提供了beginBackgroundTaskWithExpirationHandler方法来延长后台执行时间，可以给到3分钟左右时间去处理退到后台还需处理的任务。3分钟未执行完成的话，还是会被杀掉。
    - 如何避免呢？对于要在后台处理的数据要严格把控大小，太大的数据可以考虑下次启动的时候处理。
    - 怎么监控呢？在Background Task里进行计时，如果时间接近3分钟，任务还在执行，那么就可以判断它即将后台崩溃，然后记录下内容，进行上报。
- 内存被爆 和 主线程超时： 内存被爆和主线程超时，都是由于Watch Dog检测到超时，而向系统杀掉导致的。那这类监控和后台崩溃一样，需要先找到它的阈值，然后临近阈值时进行收集和上报。

各种检测

子线程UI检测

原理：Hook UIView的三个必须在主线程操作的绘制方法：setNeedsDisplay、setNeedsLayout、setNeedsDisplayRect。然后判断他们是否在子线程中操作，如果是在子线程中进行的话，打印出当前代码调用堆栈。

帧率监测

原理：帧率FPS检测主要是检测APP的界面卡顿，判断流畅性。通常的做法是基于CADisplayLin做FPS计算，CADisplayLink是Core Animation提供的一个类似NSTimer的定时器，它会在屏幕每次刷新回调一次，所以它也是以runloop的帧率为标准。所以只需统计每秒方法执行的次数，次数/时间就可以得出帧率了。但是它无法真正定位到性能。

```
-(void)starRecord{
    if(_link) {
        _link.paused = NO;
    }else{
        _link = [CADisplayLink displayLinkWithTarget:self
selector:@selector(trigger:)];
        [_link addToRunLoop:[NSRunLoop mainRunLoop]
forMode:NSRunLoopCommonModes];
    }
}
- (void)trigger:(CADisplayLink * link) {
    if ( lastTime == 0 ){
        lastTime = link.timestamp;
```

```
        return;
    }

    count ++;
    NSTimeInterval delta = link.timestamp - lastTime;
    if (delta < 1) return;
    lastTime = link.timestamp;
    CGFloat fps = count / delta;
    count = 0;
}
```

CPU使用监测

原理：CPU长时间处于高消耗的状态，会使手机发热，耗电量加剧，导致APP产生卡顿。所以要对APP的CPU使用进行监测； 方案就是：使用task_threads函数，获取当前APP所有的线程列表，然后遍历每一个线程，通过thread_info函数获取每一个非闲置线程的cpu使用。

```
+ (CGFloat)cpuUsageForApp {
    // mach_port_t 类似的指针数组，用于存放从线程
    thread_array_t          thread_list;
    //unsigned int 存放获取的线程数
    mach_msg_type_number_t thread_count;
    thread_info_data_t     thinfo;
    mach_msg_type_number_t thread_info_count;
    /*
    线程基本信息：thread_basic_info_t
    struct thread_basic_info {
       time_value_t    user_time;  //用户运行时长
       time_value_t    system_time;  //系统运行时长
       integer_t       cpu_usage;  //cpu使用率，应该是指占整体的百分百
       policy_t        policy;     //调度策略
       integer_t       run_state;   //运行状态
       integer_t       flags;        //各种标记
       integer_t       suspend_count;  //暂停线程时的计数，不知道有啥用
       integer_t       sleep_time;    //休眠时长
    };
    */
    thread_basic_info_t basic_info_th;

    /* get threads in the task
    获取当前进程中 线程列表
    mach_task_self() 返回 mach_port_t 类型，应该是指当前进程
    c语言中引用一般是用于做返回值的，所以这个函数的作用就是从当前进程中获取所有线程数组，及线程数。返回是否成功的Bool值（C语言中非0就是true）。
    */
    kern_return_t kr = task_threads(mach_task_self(), &thread_list,
&thread_count);
    if (kr != KERN_SUCCESS)
        return -1;
```

```
    float tot_cpu = 0;
    for (int j = 0; j < thread_count; j++) {
        thread_info_count = THREAD_INFO_MAX;
        //thread_info用来获取当前这个线程的具体信息。
        kr = thread_info(thread_list[j], THREAD_BASIC_INFO,
                         (thread_info_t)thinfo, &thread_info_count);
        if (kr != KERN_SUCCESS)
            return -1;
        basic_info_th = (thread_basic_info_t)thinfo;
        //这个与操作就是判断线程是否是一个空闲线程
        if (!(basic_info_th->flags & TH_FLAGS_IDLE)) {
            //宏定义TH_USAGE_SCALE返回CPU处理总频率
            tot_cpu += basic_info_th->cpu_usage / (float)TH_USAGE_SCALE;
            //如果要获取线程堆栈应该放在这里。
            ... ...
        }
    }

    // 注意方法最后要调用 vm_deallocate，防止出现内存泄漏
    kr = vm_deallocate(mach_task_self(), (vm_offset_t)thread_list,
thread_count * sizeof(thread_t));
    assert(kr == KERN_SUCCESS);
    return tot_cpu;
}
```

内存消耗监测：

内存消耗监测与CPU使用监测一样的，通过使用task_info来获取进程的虚拟信息，然后获取到phys_footprint。

```
-(NSInterger)useMemoryForApp {
    task_vm_info_data_t vmInfo;
    mach_msg_type_number_t count = TASK_VM_INFO_COUNT;
    kern_return_t kernelReturn = task_info(mach_task_self(), TASK_VM_INFO,
(task_info_t) &vmInfo, &count);
    if(kernelReturn == KERN_SUCCESS)
    {
        int64_t memoryUsageInByte = (int64_t) vmInfo.phys_footprint;
        return (NSInteger)(memoryUsageInByte/1024/1024);
    }
    else
    {
        return -1;
    }
}
```

监测卡顿：

我使用的卡顿检测方法就是：重写一个NSThread的子类，设置一个时间间隔和最大Watch Dog时间阈值。然后重写她的main函数，在函数里开启一个while循环，然后在时间间隔内查看主线程是否能够及时处理事件，如果主线程能及时处理消息，则说明不卡顿。如果不能处理，则获取主线程的线程堆栈，然后在主线程操作信号量发出之后或者超过最大Watch Dog时间阈值，则将获取到的线程堆栈及卡顿时长进行记录，视为一次卡顿，如果超过一个watch dog阈值则可以视为一次卡顿崩溃，先暂时存起来，如果后续有主线程的信号过来，则把假定的Watch Dog崩溃日志去除。

```
- (void)main {
  //在main函数里，只要没有取消当前子线程，就while循环，查看主线程是否可以响应事件。每次循环都sleep一个阈值的时间。
      while (!self.cancelled) {
          printf("\n");
          //查看线程是否活跃
          if (_isApplicationInActive) {
              //标志位，用于查看主线程是否有处理的标志
              self.mainThreadBlock = YES;
              //主线程堆栈信息
              self.reportInfo = @"";
              self.startTimeValue = [[NSDate date] timeIntervalSince1970];
              printf("1开始查看\n");
              dispatch_async(dispatch_get_main_queue(), ^{
                  self.mainThreadBlock = NO;
                  printf("2执行了主线程，发出信号\n");
                  verifyReport();
                  //如果主线程能够响应，则对信号量进行加一
                  dispatch_semaphore_signal(self.semaphore);
              });
              //阻塞当前线程threshold秒
              [NSThread sleepForTimeInterval:self.threshold];
              printf("3sleep醒来\n");
              if (self.isMainThreadBlock) {
                  printf("4主线程未来得急执行\n");
                  //如果发生了卡顿，则在threshold秒后，上面主线程肯定是没有执行的。
                  self.reportInfo = [DoraemonBacktraceLogger doraemon_backtraceOfMainThread]; //这里包含堆栈的查看信息
              }
              //如果信号量小于等于0，则阻塞，如果大于0，则先对信号量减一，再执行block操作。或者到最大卡顿崩溃时间之后自动执行，下面这个300可以视为watch Dog的查杀时间
              dispatch_semaphore_wait(self.semaphore, dispatch_time(DISPATCH_TIME_NOW, 300.0 * NSEC_PER_SEC));
              {
                  //卡顿超时情况;
                  if self.isMainThreadBlock {
                      //说明是超过了300.0秒
                      printf("5超过最大Watch Dog时间阈值：%f\n",[[NSDate date] timeIntervalSince1970]);
                  }else{
                      printf("5刚收到信号：%f\n",[[NSDate date] timeIntervalSince1970]);
```

```
                    }
                    verifyReport();
                }
                //如果不卡顿，执行顺序是1、2、3、5；如果卡顿了，则执行顺序是1、3、4、2、5。主线程卡顿多长时间，wait就等待多长时间，或者主线程卡顿的时间超过了300秒。等主线程反应过来之后，对信号量进行加1，则就可以走之后的流程了。但是这也存在一个问题，如果是卡顿太久，导致了崩溃呢！这里就收不到信号了。
            } else {
                //阻塞当前线程threshold秒
                [NSThread sleepForTimeInterval:self.threshold];
            }
        }
    }
```

流量监控

iOS的类NSURLProtocol可以拦截NSURLConnect、NSURLSession、UIWebView中的所有网络请求，获取每一次请求的request和response对象。但是这个类没法拦截TCP请求。可以使用URLProtocol做如下事情：

- 重定向网络请求
- 忽略网络请求，使用本地缓存
- 自定义网络请求的返回结果
- 一些全局的网络请求设置

NSURLProtocol的使用

堆栈收集与分析。

上述过程中的CPU使用检测、卡顿检测如果要追踪到具体函数，则需要获取卡顿线程的具体函数和堆栈，使用的方法流程大概是：

1、先将NSThread转成内核线程：先通过task_threads获取线程列表，

iOS Crash的捕获知识

crash崩溃堆栈分析(略) 移动监控体系之技术原理 了解和分析iOS Crash Report iOS crash日志堆栈解析 [iOS崩溃异常捕获](https://juejin.im/post/5a93c9385188257a7b5aba42

Core Text

上述异步绘制中设计到CoreText，所以这里简单介绍一下： 三个类：CTFrameRef: 画布；CTLineRef: 每一行；CTRunRef:每一小段。 每个画布(CTFrameRef)可以包含多行(CTLineRef)，每一行可以包含多个小段(CTRunRef)。 绘制步骤： 首先一般的绘制都是异步绘制，所以基本是在display函数或者drawRect函数中。因为这样才能拿到context

```
    //
```

```
    CGContextRef context = UIGraphicsGetCurrentContext();
    //变换坐标
    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);
    //设置绘制的路径
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddRect(path, NULL, self.bounds);
    /创建属性字符串
    NSMutableAttributedString * attStr = [[NSMutableAttributedString alloc]
initWithString:str4];

    //颜色
    [attStr addAttribute:(__bridge NSString
*)kCTForegroundColorAttributeName value:(__bridge id)[UIColor
redColor].CGColor range:NSMakeRange(5, 10)];

    //字体
    UIFont * font = [UIFont systemFontOfSize:25];
    CTFontRef fontRef = CTFontCreateWithName((__bridge
CFStringRef)font.fontName, 25, NULL);
    [attStr addAttribute:(__bridge NSString *)kCTFontAttributeName value:
(__bridge id)fontRef range:NSMakeRange(20, 10)];

    //空心字
    [attStr addAttribute:(__bridge NSString *)kCTStrokeWidthAttributeName
value:@(3) range:NSMakeRange(36, 5)];
    [attStr addAttribute:(__bridge NSString *)kCTStrokeColorAttributeName
value:(__bridge id)[UIColor blueColor].CGColor range:NSMakeRange(37, 10)];

    //下划线
    [attStr addAttribute:(__bridge NSString *)kCTUnderlineStyleAttributeName
value:@(kCTUnderlineStyleSingle | kCTUnderlinePatternDot)
range:NSMakeRange(45, 15)];

    CTFramesetterRef framesetter =
CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attStr);
    CTFrameRef frame = CTFramesetterCreateFrame(framesetter, CFRangeMake(0,
attStr.length), path, NULL);

    //绘制内容
    CTFrameDraw(frame, context);
```

[Core Text编程指南]

启动优化

- 冷启动是指APP不在后台，第一次打开

- 热启动是指APP在后台被唤起。

冷启动优化

APP启动主要分为三个阶段：main函数之前、main函数之后、首屏渲染完成：

- main函数之前： 主要工作：加载可执行文件；加载动态链接库；运行时处理(包括类的注册、category注册、selector唯一性检测等)；初始化(+load方法、创建c++静态全局变量)。 所以可以做的事情就包括：
  - 减少动态库的加载，苹果公司建议使用更大的动态库，可以考虑将多个库进行合并，数量上建议是6个非系统动态库；
  - 删减无用的代码和类；
  - +load方法尽量少用，或者将里面的内容挪到其他地方，比如+initialize()；控制c++全局变量的数量；
- main函数之后：

  这个阶段主要是指从main函数开始，到APPDelegatD的didFinishLaunchingWithOptions方法里首屏渲染相关方法的执行；我的理解就是main函数到设置window的root结束。 可以做的事情包括：
  - 不要将各种无必要的类的初始化、配置文件的读写、首屏列表的数据获取和渲染相关等放到这个区间里面。比如说首页列表，通常会直接在viewdidload里做，这样是不好的。
- 首屏渲染完成： 这个阶段就是从首屏渲染到didFinishLaunchingWiithOptions方法作用域结束为止。这个阶段用户已经能看到首屏数据了，所以要注意的就是主线程卡顿相关的问题。

启动耗时检测：略

播放器 与 音视频相关

基础类

- NSURL 支持 本地文件url 和 网络的url
- AVPlayerItem 通过 URL 初始化的一个播放对象（状态获取）
- AVPlayer 通过 AVPlayerItem 初始化的播放控制器（控制）
- AVPlayerLayer 通过 AVPlayer初始化的一个播放展示视图（展示）

AVPlayerItem的各种需要监听的状态

- 通过监听“status”字段来监听播放状态，主要存在三种状态：.unknown、.readyToPlay、.failed。如果是.readyToPlay状态，则可以获得播放时长、视频大小、视频首帧等信息；
- 通过监听"loadedTimeRanges"字段来监听缓冲大小；
- 通过监听"playbackBufferEmpty"字段表示缓冲区空了，需要加载；
- 通过监听"playbackLikelyToKeepUp"来表示缓冲区满了。

视频首帧获取

```
// 获取视频第一帧
-(UIImage*)getVideoPreView{
    AVAssetImageGenerator *assetGen = [[AVAssetImageGenerator alloc]
initWithAsset:self.playerItem.asset];
    assetGen.appliesPreferredTrackTransform = YES;
    CMTime time = CMTimeMakeWithSeconds(0.0, 600);
    NSError *error = nil;
    CMTime actualTime;
    CGImageRef image = [assetGen copyCGImageAtTime:time
actualTime:&actualTime error:&error];
    UIImage *videoImage = [[UIImage alloc] initWithCGImage:image];
    CGImageRelease(image);
    return videoImage;
}
```

https://juejin.im/post/5da1a30de51d457825210a8c

直播框架与实践

移动端主要框架

移动端直播框架就主要分为主播端和观看端。

- 主播端：
  - 音视频采集：AVFoundation；
  - 视频处理(美颜、水印)：GPUImage；
  - 音视频编码压缩：音频压缩FFmpeg，视频压缩：x264；
  - 封装音视频后进行推流：libremp框架；
- 观看端：
  - 音视频解码：FFmpeg视频解码，VideoToolBox视频硬解码，AudioToolBox音频硬解码；
  - 播放：ijkplayer；
- IM聊天：聊天室、点亮、推送、超过、黑名单、聊天信息、滚动弹幕等；
- 礼物相关：各种礼物、红包、排行榜等；

主要是对业务层的搭建

直播端业务逻辑倒是不是特别复杂，大部分可以直接使用MVC框架即可。主控制器的独立业务太多的话，可以拆分成多个单独的Category；对于Model的数据变化可以采用notification的形式通知，便于做多处绑定；然后对于各种类型消息可以使用面向协议的方式编写； 礼物：要使用队列存储礼物； 消息：聊天消息要注意高度计算及卡顿相关问题，然后各种消息的分发也应该使用队列的形式； 弹幕：也要用队列存储弹幕，同时要限制条数； 聊天室：聊天室的各种消息类型比较多，尤其是频繁进出房间的时候。导致客户端与服务端之间的消息过多，可能就会产生一些性能问题，可以考虑在用户过多的情况下不发送进出房间消息，或者根据用户优先级来确定是否发送进出房的同步消息；

直播协议比较：

- HLS：苹果退出的流媒体技术，是以点播的技术方式来实现直播，使用HTTP短链接。优势：兼容

性、性能和、穿墙和HTTP一样；劣势：高延时、文件碎片；
- RTMP：实时消息传送协议，TCP长连接，端口1935，有可能会被墙掉，低延时。 HLS与RTMP对比: HLS主要是延时比较大，RTMP主要优势在于延时低HLS协议的小切片方式会生成大量的文件，存储或处理这些文件会造成大量资源浪费，相比使用RTSP协议的好处在于，一旦切分完成，之后的分发过程完全不需要额外使用任何专门软件，普通的网络服务器即可，大大降低了CDN边缘服务器的配置要求，可以使用任何现成的CDN,而一般服务器很少支持RTSP。

直播关键性能指标

- 直播为什么会卡顿?
    - 关键词：**帧率**FPS和APP刷新帧率是一个概念，表示每秒显示的图片数，**码率**：图片进行压缩后每秒显示的数据量,码率主要用于推流和拉流，跟网速有关。
    - 推流帧率太低：如果主播端手机性能较差，或者很占CPU的后台程序在运行，可能导致视频帧率太低。正常情况下FPS达到每秒15帧以上才能保证观看端的流畅度，如果FPS低于10帧，可以判定是**帧率太低**，则全部观众体验都会卡顿;
    - 上传阻塞：主播端在推流时会源源不断产生音视频数据，如果手机上传网速太小，那么音视频数据就会堆积在手机里传不出去，导致全部观众体验卡顿；
    - 下行不佳：观众端带宽不足，比如直播流的码率是2Mbps，也就是每秒2M数据要下载，下行带宽不够那么就会影响当前用户卡顿。
- 解决方案：
    - 查看当前推流SDK的CPU的占用情况和当前系统的CPU占用情况；如果当前系统占用率超过80%，那么采集和编码都会受到影响。所以要找到直播之外的CPU消耗情况，进行优化；
    - 不要盲目追求高分辨率：较高的视频分辨率如果没有较高的码率，则就无法带来好的体验，所以要根据网络情况选择分辨率。
    - 有的SDK，如果发现APP的CPU使用率过高，则会切到硬编码来降低CPU的使用率，比如腾讯SDK。
    - 上传阻塞的解决方案：使用更好的网络；使用合理的编码设置，比如低分辨率；
    - 下行不加：卡顿延迟，因为由于网络不行，所以可能就拿不到足够的数据，所以可以考虑让APP缓存到足够的数据后再播放，不过这个会导致高延时，而且播放时间越久就越延时，HLS就是通过引入延时20~30秒来实现流畅的播放体验；腾讯SDK提供了多种延时控制方案：自动模式————会根据网络情况自动调节延迟大小；极速模式————高互动的秀场直播，就是在不卡顿的情况下，将延时调节到最低；流畅模式————适用于游戏直播，会在出现卡顿的时候loading知道缓冲区蓄满；

重要腾讯直播SDK

[袁峥如何快速开发一个完整的iOS直播APP]https://www.jianshu.com/p/bd42bacbe4cc 研发直播APP的收获 开发直播APP中要了解的原理

下载模块与AFNetWorking

下载框架

首先需要一个manager管理整个app的下载事件；它负责管理每一个request。比如说取消、重新加载等操作。其次需要有一个Config配置类，用来配置基础信息，比如配置请求类型、cookie、时间等信息。然后有一个对response进行处理的工具，比如日志的筛选打印、对一些异常错误的处理等等

AFNetworking

整体框架：AFNetWorking整体框架主要是由会话模块(NSURLSession)、网络监听模块、网络安全模块、请求序列化和响应序列化的封装以及UIKit的集成模块(比如原生分类)。 其中最核心类是AFURLSessionManager，其子类AFHTTPSessionManager包含了AFURLRequestionSerialzation(请求序列化)、AFURLResponseSerialzation(响应序列化)两部分；同时AFURLSessionManager还包含了NSURLSession(会话模块)、AFsecurityPolicy(网络安全模块：证书校验)、AFNetWorkingReachabilityManager(负责对网络连接进行监听)； AFURLSessionManager主要工作包括哪些？ 1、负责管理和创建NSURLSession、NSURLSessionTask 2、实现NSURLSessionDelegate等协议的代理方法 3、引入AFSecurityPolicy保证请求安全 4、引入AFNetWorkingReachabilityManager监听网络状态 https://www.jianshu.com/p/b3c209f6a709

Alamofire：同一个作者写的swift版本的AFNetWorking

整体框架：Alamofire核心部分都在其Core文件夹内，它包含了核心的2个类、3个枚举、2个结构体；另一个文件夹Feature则包含了对这些核心数据结构的扩展。 2个类：Manager(提供对外接口，处理NSURLSession的代理方法)；Request(对请求的处理)； 3枚举：Method(请求方法)；ParameterEncoding(编码方式)；Result(请求成功或失败数据结构) 2结构体：Response(响应结构体)；Error(错误对象) 扩展中包括Manager的Upload、Download、Stream扩展、以及Request的扩展Validation和ResponseSerialization。 怎么处理多并发请求？ 使用NSOperetionQueue！

[AFNetWork图片解码相关]https://www.jianshu.com/p/90558187932f

AsyncDisplayKit：

整体框架： 正常情况下，UIView作为CALayer的delegate，而CALayer作为UIView的一个成员变量，负责视图展示工作。ASDK则是在此之上封装了一个ASNod类，它有点view的成员变量，可以生成一个UIView，同时UIView有一个.node成员属性，可以获取到它所对应的Node。而ASNode是线程安全的，它可以放到后台线程创建和修改。所以平时我们对UIView的一些相关修改就可以落地到对ASNode的属性的修改和提交，同时模仿Core Animation提交setneeddisplayer的这种形式把对ASNode的修改进行封装提交到一个全局容器中，然后监听runloop的beforewaiting的通知，当runloop进入休眠时，ASDK则可以从全局容器中把ASNode提取出来，然后把对应的属性设置一次性设置给UIView。

主要解决的问题：布局的耗时运算(文本宽高、视图布局运算)、渲染(文本渲染、图片解码、图形绘制)、UIKit的对象处理(对象创建、对象调整、对象销毁)。因为这些对象基本都是在UIKit和Core Animation框架下，而UIKit和Core Animation相关操作必须在主线程中进行。所以ASDK的任务就是把这些任务从主线挪走，挪不走的就尽量优化。

IM

- IM中信息的可靠性传输：消息不丢失、消息不重复； 首先我们大概看一下消息的发送流程：
    - 步骤1：用户A发送信息到达IM服务器；
    - 步骤2：服务器进行消息暂存；

    - 步骤3：暂存成功后，将成功的消息返回给A；
    - 步骤4：返回确认消息的同时将消息推送给用户B； 这些步骤中1~3步任一失败的话，用户A会被提示发送失败。后面步骤中，可能出现消息未能推送给B导致失败，也可能出现B写入本地数据库失败导致消息丢失。 解决方案： 基本原理就是：ACK确认机制+消息重传+消息完整性校验，来解决消息丢失的问题。 ACK确认机制就是TCP的ACK确认回复，在三次握手、四次挥手中都有使用到。首先TCP报文字节都是有数据序列号的。ACK报文每次回复确认都会带上序列号，告诉发送方接收到了哪些数据，所以这也保证了消息的有序性。然后如果ACK确认报文丢失，那么可能是发送数据丢失位到底IM服务器，也可能是到底了服务器，但是返回的确认报文丢失了。无论是这两种的那种情况，TCP都可以使用超时重传的策略解决，只是如果是ACK确认丢失了，则服务器会先忽略掉新的数据，然后发送ACK应答。其次IM业务层也基本参考了ACK确认机制和超时重传机制。比如推送消息给B时，会携带一个标识，然后将退出去的消息添加到"待ACK确认消息队列"，用户B收到消息后会回复一条ACK包，然后服务器把消息从待确认队列中删除。否则进行重新推送。之所以要加业务层的ACK确认机制，是因为TCP只能保证传输层的消息是否到底，但是业务层可能到底之后还需要进行处理，比如说存入本地数据库。 消息完整性校验：则是可以通过每一条消息带上时间戳的形式，然后每次重连后对比时间戳，则可以知道大概哪些消息没有到达，然后将待确认队列的时间戳之后的数据都按序发送一遍。 如何确保消息不重复？ 同样是给每个消息带上一个ID，接收方接到消息后，先进行业务去重，然后才考虑是否使用这个消息。
- IM数据库如何设计表
    - 会话表：用于存储所有会话。
    - 聊天详情表：主要用于存储消息
    - 群组表：存储每个群组相关数据
    - 群组信息表：用于关联群组表和群成员表
    - 群成员表：存储群成员信息
    - 联系人表：存储所有联系人。

单元测试与可持续集成

https://juejin.im/post/5a3090f2f265da4310485d01

桂林 单元测试

Swift Package Manager

https://www.jianshu.com/p/479986e9ae80

APP 相关

APP 证书

- 苹果如何保证iOS系统只安装苹果的软件?

APP如何与后台进行通信

算法

排序相关：

二叉树相关：

链表相关：

二叉树:

前序、中序、后序遍历指的是根节点的位置。 中序：

```
//中序，使用栈
-(void)sourt1:(Node *)root{
   Node* p = root;
   stack<Node *> s;
   while(!s.empty() || p){
       if(p){ //先将左子树全部入栈
           s.push(p);
           p = p.leftchild;
       }else{
           p = s.top();
           s.pop();
           print(p.value);//打印
           p = p.rightChild; //进入右子数
       }
   }
}
//前序：也是使用栈
-(void)sourt2:(Node *)root{
   Node* p = root;
   stack<Node *> s;
   while(!s.empty() || p){
       if(p){ //先将左子树全部入栈
           print(p.value);//打印
           s.push(p);
           p = p.leftchild;
       }else{
           p = s.top;
           s.pop();
           p = p.rightChild; //进入右子数
       }
   }
}
//后序：也是使用栈。这个好难，先放弃吧
```