语言

swift & OC

swift是解析型静态语言、OC是编译型动态语言；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；Struts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。

OC基础

KVO & KVC

Associated关联对象

Category

通知

响应链：

1、当触摸事件发生后，系统会将该事件添加进UIApplication管理的队列事件中； 2、UIApplicatioA会从事件队列中取出该事件，并将事件分发给keywindow； 3、然后keywindow会判断hitTestwithEvent来返回最终响应的视图，那在这个过程中会调用pointInsert判断点击事件是否在当前视图的范围内，如果在，则会倒序遍历子视图，来判断是否是该视图响应事件。注意在pointInsertEvent内，需要使用convertpoint：toView：来进行坐标系转换

内存

ARC & MRC

MRC是手动内存管理，ARC是自动内存管理，它通过LLVM编译器和runtime协作在合适的时机给代码添加retain和release等代码，实现自动引用计数的管理。ARC无法显示调用retain和release等函数。

内存管理方案

TaggedPointer：对于一些小对象，比如NSNumber和NSDate。它实际上采用的是TaggedPointer的管理方式进行内存管理的，它是对象指针实际上不执行任何内存地址，而是直接将值存储在了指针本身里，该指针就被拆分成两部分，一部分是直接保存数据，另一部分是作为特殊标记。
NONPOINTER_ISA：在64位架构下，一个isa指针是由64位比特位组成。而实际上存储改对象所属类对象地址只需要用到大概33位，所以剩下的位数就可以用来存储其他的信息。比如第一位存储的是当前指针是否是一个纯isa指针，如果是0则表示是只存储了所属类对象的内存地址，如果是1则表示不是一个纯isa指针，也就是这里的非指针类型的ISA，那NONPOINTER_ISA第位2则存储了当前对象是否有关联对象，第3位表示是否是否使用ARC，接下来的33位表示所属对象的内存地址，接着是表示是否完成初始化的magic，之后1位表示该对象是否有弱引用指针，接着1位表示是否正在执行deallocing，再下1位则表示引用计数是否过大无法存储在isa指针，剩下的几位则用来存储实际的引用计数。也就是说当引用计数小于某个值(大概是10)的时候，并不需要使用外挂散列表存储引用计数相关信息，若超过了这个值，则需要使用。 **散列表：SideTables**：它是由多个散列表组成的SideTables。每个散列表下包含了一个自旋锁、一个引用计数表、一个弱引用表。 q1：为什么使用多个sidetable 因为操作引用计数需要加锁，这样就存在效率问题，而分割成多个sidetable，就可以并行操作，提高访问效率。 q2：怎么实现快速分流，也就是说如何快速定位到是属于哪一张sidetable？ sidetable的本质是一个散列表，那么它

就是通过对象指针作为key值，经过hash计算定位到相对的sidetable的，比如用过key值对sidetables的个数取余，这样就增加了访问效率，无需遍历。

内存数据结构

自旋锁：它是一个“忙等”的锁，也就是说在当前资源被某个线程占用的时候，其他的线程会一直试探该锁有没有解锁，而像信号量，则是会在获取不到资源的时候进行休眠，等资源被释放后，则再被唤醒。 **引用计数表**：引用计数表也是一个hash表，它是通过hash函数插入和获取引用计数，提高访问效率。 hash表里的每个元素是一个unsigned long类型的size_t。它也是由64位比特位组成，其中第一位是表示是否有弱引用，第二位表示是否正在执行dealloc函数。剩下的则是表示引用计数的值，所以计数的时候需要位移2位，也就是加减4。 **弱引用表**：实际上也是一个hash表，它存储的是一个weak_entry_t的结构体数组，它里面的每个对象存储的都是一个弱引用指针。

引用计数

alloc： q1:通过alloc生成的对象，其实并没有设置引用计数为1。但是获取它的引用计数的时候确实是1，为什么。 retain： q1：我们在进行retain操作的时候，系统是怎么查找其对于的引用计数的呢？ 是经过两次hash查找，然后进行+1操作（实际上是位移操作）。 release： 同样是经过两次hash算法，获取到引用计数值，然后进行-1操作。 retainCount： **查找引用计数时，会通过一个初始化为1的局部变量 ➕ 上面各方法中查找到的引用计数值。所以说上面说的isa或散列表等存储的引用计数值实际上就是引用计数的值减一**。而这也说明了为什么执行alloc操作时，没有设置引用计数，但是查找到的retainCount仍然为1的原因。 dealloc: 执行dealloc时，判断是否可以释放的条件包括：是否使用nonpointer_isa、是否有弱引用指针、是否有关联对象、是否使用ARC、是否使用sidetable。如果这些条件都为NO的时候，就可以直接使用c函数free直接释放，否则则要调用objc_dispose()进行进一步清理。 而objc_dispose则会一步步 移除对关联对象、将指向该对象的弱引用指针置为nil，将当前对象在引用计数表的数据清除掉等操作。(这里解决了两个面试题:1、对象在释放的时候，是否有必要移除掉关联对象；2、weak修饰的对象是怎么讲指针置为nil的。答案就是在dealloc内部实现的时候有做这些操作)

弱引用 _ weak

q1：系统是怎样把一个weak变量添加到它对应的弱引用表中的。 一个被声明为 _ weak 的对象指针，经过编译器编译，会调用一个objc_initWeak函数，之后会调用weak_register_no_lock()函数进行弱引用变量的添加，具体添加的位置是通过hash算法进行位置查找的，如果查找的对应位置中已经有了当前对象对应的弱引用数组，则把当前变量添加进弱引用数组，如果没有，则重新创建一个弱引用数组。 q2：当一个对象被释放之后，weak变量是怎么被清理的。 会被置为nil。当对象执行dealloc的时候会调用弱引用清除相关函数，在函数内部会通过弱引用指针找到弱引用数组，然后遍历所有的弱引用指针，分别置为nil。

自动释放池

AutoreleasePool的实现原理： AutoreleasePool为何可以嵌套使用：

首先编译器会将@autoreleasepool{}括起来的代码改写成:

```
void * ctx = objc_autoreleasePoolPush();
{}中的代码
objc_autoreleasePoolPop(ctx);
```

objc_autoreleasePoolPush函数内部会调用AutoreleasePoolPage的push函数，objc_autoreleasePoolPop(ctx)也是调用AutoreleasePoolPage的pop(ctx)函数。一次pop操作是一次批量的pop操作，也就是在push的时候，会将{}函数体里的所有对象添加进自动释放池中，当执行pop操作的时候，则会给每一个对象发送一次release操作。所以说是一次批量操作。

q1：什么是自动释放池？ 是以栈为节点通过双向链表的形式组合而成的，是和线程一一对应的。因为AutoreleasePoolPage的数据结构包含一个AutoreleasePoolPage类型的parent指针和一个AutoreleasePoolPage类型的child指针，同时还包含一个pthread和一个id指针。

架构

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。

下载层设计思路，需要考虑的点

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配置了类，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

模块化

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。 文件夹隔离：将各模块的业务代码整理到相应的模块文件夹中，将各个模块用到的资源、宏也沉淀到基础库里。s 路由化：首先我们通过MGJRouter实现简单的路由化，尽可能地解耦各个模块。它是通过注册组件，通过URL调用页面，通过路由表的映射关系进行关联。 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。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。

网络相关

HTTP：

http其实就是超文本传输协议，它主要包括请求报文和响应报文两部分组成。请求报文包括：方法(get、post)、url、http版本、首部字段(媒体类型、Encode编码格式、认证信息等)、实体主题；响应报文包括：http版本、状态码、首部字段(accept_range字节范围、时间、重定向URI)、实体主题。

http的特点：无连接（也就是每次请求连接都需要经历连接和断开的过程）、无状态（同一个用户在多次发送http请求时，server端并不知道是同一个用户发送的）

状态码：2xx、3xx、4xx、5xx，200请求成功，301、302一般是重定向问题、404一般是网络问题、504一般是指服务器问题。

Get和Post的区别：Get请求的参数通过？拼接在URL后面，post则放在Body里面。然后Get是安全的、幂等的、可缓存的，Post是非安全的、非幂等的、非可缓存的。安全性是指是否会引起server端的变化，幂等是指一种请求方法执行多次的结果是否完全相同，可缓存的是指代理服务器是否会进行缓存。

TCP三次握手：客服端向server端发送一个请求报文，server端接收到请求报文后会发送一个确认响应报文同时也附带一个建立连接的请求报文，客户端收到server端发来的请求报文后再会给server端一个确认报文。

TCP四次挥手：客户端向server端发送一个请求断开连接的报文，server端收到后会返回一个确认报文（这样客户端对server的连接就断开了），然后server端向客户端发送一个断开连接的请求报文，客户端收到后回复一个确认报文（这样server端对客户端的连接也断开了）

持久连接：应对http无连接特点。指在一定时间范围内不需要反复进行握手和挥手动作。http提供的持久连接的方案就是修改请求头部字段，比如connection: keep-alive(客户端期许采用持久连接)、time:20(持久连接时间)、max：10(这条连接最多可以发生多少次请求和响应对)。 那怎么判断持久连接中的一次连接已结束呢？1)通过响应头部字段：content-length:1024来判断。2)还有就是最后一个报文的chunked字段是否为空来判断。

Charles的抓包原理：利用了http中间人攻击漏洞进行抓包，中间人就是模仿客户端和server端的所有操作。

HTTPS

HTTPS：HTTPS = HTTP + SSL/TLS。 HTTPS连接的建立流程：客户端先向server发送一个SSL版本及支持的加密算法和随机数C，server会返回一个选定的加密算法、随机数S、server端证书。客户端接收到后会先验证server证书（也就是server端公钥），然后通过C、S、预组秘钥组装成会话秘钥，之后通过server端公钥对预组秘钥进行加密发送给server端，server端则通过私钥解密预组秘钥，然后通过C、S、预组秘钥组装会话秘钥。然后客户端和server端相互发送一个加密消息，验证安全通道是否建立完成。 HTTPS都使用了哪些加密手段？ 对称加密和非对称加密。非对称加密在公私钥中使用到，传输过程中则是使用对称加密。 什么是非对称加密和对称加密？ 非对称加密包含两个概念：公钥、私钥。加解密使用的钥匙不一样的。用公钥加密，就得用私钥解密；私钥加密就得用公钥解密。 对称加密：加解密用的是同一个秘钥。

TCP/UDP

UDP：用户数据报协议。 UDP特点：无连接、尽最大努力交付（不保证可靠传输）、面向报文(既不合并报文也不拆分报文，会原封不动传输报文，只是在运输层会拼装一个UDP首部)； UDP提供的功能：复用(就是不同的端口都可以复用传输层UDP数据报)、分用(接收到数据报后，会根据目的端口进行分发)、差错检测(就是发送方通过某种方法计算出某个数据，将其插入到UDP首部传输给接收方，然后接收方接收到数据后，运用相同的方式进行计算，然后对比接收到的数据，进行差错检测。)

TCP：传输控制协议。则需要建立连接。 TCP特点：面向连接（数据传输开始和结束需要建立和释放连接）、可靠传输（无差错、无重复、按序到达）、面向字节流、流量控制、拥塞控制。

Q1:为什么是3次握手，而不是两次？ 假如客户端发送建立连接的请求报文发生了超时，客户端会启用超时重传策略，重新发送连接请求，Server端收到了会回复确认报文，那之后又收到了之前超时的请求连接，那就又会建立一次连接，这样就可能建立了两次连接。而多了那次客户端确认报文则可以解决这种问题。

Q2、为什么是4次挥手，要分别断开两个方向的链接？ 因为TCP建立的是一个全双通的链接，就是无论从客户端到server端，还是server端到客户端，都可以建立单独的发送与确认接收的通道，比如说在4次握手中，如果仅仅是进行了前两步（断开了客户端到server端的链接），那么此时客户端是不能向server端发送数据的，但是server端依旧可以向客户端发送数据。

Q3 、怎么保证可靠传输的？ 可靠传输是通过停止等待协议来实现的。它是包括4方面的：

无差错情况：就是每次报文的传输中，server端收到后都会返回一个确认报文。

超时重传：那如果超时了，也就意味着在这个时间范围内，server端没有收到报文，也自然没有返回确认报文，那客户端就会进行重新发送。

确认丢失：是指server端返回的确认报文丢失了，那么同样的，客户端没有在时间限定内收到确认，所以会进行重新发送。server端会将第一次接到的报文丢失掉。

确认迟到：指确认报文迟到了，客户端同样会进行确认丢失一样的操作。

Q4、面向字节流? 是指TCP并不是原封不动地将发送方发送的字节一次性地完全地传输给接收方，而是会根据实际情况对字节流进行拆分或合并，然后再进行发送。和UDP的面向报文的方式正好相反。

Q5、怎么做到流量控制、按序到达？ 通过滑动窗口协议实现。TCP的发送缓存当中的数据都有字节编号，然后我们进行排序。将每次发送出去后收到的确认报文位置进行标记，将将要发送的字节流中的最后字节进行标记。然后这大概就是字节流中的一个小的字节窗口。但是为了避免接收方数据溢出，所以接收方需要动态调节发送方的窗口大小来控制发送速率（比如接收方只能接收2个字节了，那么发送方就最多只能发送2个字节。这个应该是放在报文的首部字段）。同样的，接收缓存中也会对已接收的字节进行排序，它会对按序到底的下一个期望到达的字节进行标记（比如已经接收了1、2、3字节，那么期望标记就是第4个字节，而返回给长层应用程序的字节就是这些已按序到底的字节部分，那比如说还接收到了第6个字节，则暂时不会进行处理。）。所以呢，这个滑动窗口协议就可以进行流量控制和按序到达了。

Q6、拥塞控制： 慢开始、拥塞避免：一开始先发送一个报文，如果没有发送拥塞，则翻倍发送2个，然后再4个、16个(指数增长的方式)。一直达到窗口的门限初始值为止；然后再通过拥塞避免的策略，以线性增长的方式发送报文，可能达到某个值得时候，就产生了网络拥塞（比如连续3个报文没有收到确认报文），此时就越高采用拥塞避免的乘法减小的策略，只发送一个报文，同时将门限值降低，然后重新开始“慢开始”。

快恢复、快重传：是指在达到拥塞时，回到新的门限值，以线性增长的方式发送报文，而不经过前面指数增长的慢开始阶段。

DNS解析：

DNS解析过程？ DNS服务器是提供域名到IP之间的解析服务，一般计算机就是一个IP地址，但是纯数字不符合人类的记忆习惯，所以一般会有一个域名，比如www.baidu.com ，当我们代码对某个域名发起访问的时候，则要通过DNS解析，找到对应的IP，然后再进行访问 。一般DNS解析都是有运营商进行管理，比如移动的卡发出来的访问，先经过移动运营商，然后移动运营商找到对应的ip，然后进行访问。DNS采用UDP数据报文，53端口号，且明文。

DNS解析查询方式： 1、递归查询：按照 本地DNS——根域DNS——顶级DNS——权限DNS的层级一层层递归查找； 2、迭代查找：先查询本地DNS，然后本地DNS依次询问根域DNS、顶级DNS、权限DNS； DNS劫持问题？ 因为DNS是UDP明文传输，就有可能被钓鱼DNS劫持，返回错误的IP地址。DNS劫持与解析都与http无关，它是发生在http之前的操作。 解决DNS劫持： 1、httpDNS：实际上DNS解析是指DNS协议向DNS服务器的53端口进行请求，采用HTTPDNS这种方式则是直接通过http协

议向DNS服务器的80端口进行请求，这样实际上就不存在DNS解析了，所以也就不存在DNS解析问题了。比如：http://119.29.29.29/d?dn=www.baidu.com&ip=163.177.153.109 (其中http://119.29.29.29/d 是国内最大的DNS域名服务器，dn=www.baidu.com 是需要解析的域名，后面是本地IP地址) 2、长连接： 客户端采用 长连server 从API Server通过内网专线获取IP的方式

DNS解析转发问题？ 是指DNS解析服务器为了节省资源，将解析请求发送给其他DNS域名服务器，依次转发，最后返回的IP地址可能不是同一运营商的网络，存在跨网访问的可能，造成一些请求缓慢等效率问题。

Session/Cookie

应对http无状态特点，指多次发送同一个请求，server端无法知道是否是同一个用户。 Cookie主要是用来记录用户状态，区分用户；状态主要保存在客户端； Session也是主要用来记录用户状态，区分用户；状态主要存放在server端。

socket

- socket与http：
- socket 打洞：

Block

- 全局类型block：指没有用到任何外部变量，只用到全局变量、静态变量的block称作全局block，生命周期与应用程序等同。存放在已初始化数据区中； 进行copy操作，等于什么都没做。
- 栈类型上的block：只用到外部局部变量、成员属性变量，无强指针引用的block。 放到栈里；进copy操作，copy的结果是在堆上产生了一个block
- 堆上的block：有强指针引用或使用了copy关键字修饰的block。 放到堆里；进copy操作，copy的结果是会增加其引用计数

2、block截获变量是根据被截获变量的类型进行区分的：

- 局部变量：基本数据类型、对象类型: 对于基本数据类型的局部变量只截获其值，对于对象类型的局部变量，连同所有权修饰符一起截获,也就是直接操作对象。
- 局部静态变量：以指针形式进行截获。也就是说直接操作的指针，所以会被改变。
- 全局变量：不截获！所以操作就是直接使用全局变量本身。
- 全局静态变量：不截获！所以操作就是直接使用全局静态变量本身。

3、如果我们声明一个对象的成员变量是一个block，然后在栈上创建block，同时赋值给成员变量。如果成员变量block没有使用copy关键字，而是使用assign，那么当栈函数内存被释放的时候，继续访问这个block就会导致崩溃。那如果使用了copy关键字，那么在堆上就会产生一个一模一样的block。那么在MRC下，此时堆上的block就没有被释放掉，导致内存泄露。而被copy关键字进行修饰后的block，无论在哪对block进行访问，其实都是通过 _ forwarding 指针访问的堆上的block。

q:何时需要对block进行copy操作。 所以说如果一个block成员变量在栈上进行创建的话，那么就应该进行copy操作，避免block跟随栈函数被释放。

多线程

GCD:

多核并行运算的解决方案，可以合理利用更多CPU内核。更主要的是自动管理线程的生命周期，只需要告诉GCD干什么就行。

1、任务方面：同步Sync（执行完当前任务才会执行下一个任务）、异步Async（不需要等待当前任务执行完毕）； 队列相关：串行Serial（任务一个一个发出）、并行Concurrent（多个任务同时执行）；

2、global_queue是一个并发队列，创建全局队列时，第一个参数是一个优先级的标识，所以如果要在并行队列中，让任务先执行，这可以通过设置这个优先级来达到目的（但是这里要注意，先执行并不一定是第一个执行完，它只能保证开始的执行顺序而已）。它的优先级包括（低-高：background（同步备份数据）、utility（需要时间的下载）、default、user-Initiated(用户出发的，如打开文件)、user-Interractive(用户交互，如主线程事件)）

3、Dispatch_Barrier栅栏，实现多读单写

```
A{
    set{
        ConcurentQueue.async(flogs:.barrier){
            _a = newvalue
        }
    }
    get{
        ConcurentQueue.sync{
            return _a
        }
    }

}
```

4、信号量：信号量可以改变全局队列里设置好的优先级。

```
var highQueue = Dispatch.global(qos:.userInitiated)
var lowQueue = Dispatch.global(qos:.utility)

let semaphore = DispatchSemaphore(value:1)
lowQueue.asycn{
    semaphore.wait()
    sleep(1000)
    semaphore,signal()
}
highQueue.asycn{
    semaphore.wait()
    sleep(1000)
    semaphore,signal()
}
//这里lowQueue的优先级更高。
```

5、死锁的几种情况

```
//1、串行队列：异步里同步嵌套
SerialQueue.async{
    print(1)
    SerialQueue.sync{
```

```
        print(2)
    }
}
//2、串行队列：同步里同步嵌套
SerialQueue.sync{
    print(1)
    SerialQueue.sync{
        print(2)
    }
}
//3、主线程中执行同步操作
viewDidLoad(){
    DispatchQueue.main.sync{
        print(3)
    }
}
//4、NSOperation 线程间依赖
let operaA = Operation()
let operaB = Operation()
operaA.addDependency(operaB)
operaB.addDependency(operaA)
```

6、dispatch_group dispatch_group_async里应该执行同步请求，如果执行异步请求，线程会立即返回，达不到想要的效果，所以要使用dispatch_group_enter(group)和dispatch_group_leave(group)来实现

```
{
    dispatch_group_t group = dispatch_group_create()

    dispatch_group_enter(group)
    self.request1({
      sleep(1000)
      dispatch_group_leave(group)
    })

    dispatch_group_enter(group)
    self.request2({
      sleep(1000)
      dispatch_group_leave(group)
    })

    dispatch_group_notify(group,dispatch_get_main_queue(),^{
        print(finish)
    })
}
```

Operation：

Operation是指一个单独的任务，然后把它放到OperationQueue中实现多线程运行效果，OperationQueue实现了暂停、继续、终止、优先顺序、依赖等操作。同时通过设置最大并发量maxConcurentOperationCount来确定其实串行还是并发。 Operation包含有常见4个状态：isReady(就绪)、Executing(运行中)、Cancelled(取消)、finished(完成)。 实现NSOperation可以通过它的子类NSIvocationOperation和NSBlockOperation，或者继承自NSOperation自定义。 1、Operation的取消操作，如果任务已经开始，那此时调用cancel是没有用的，所以得等任务结束之后判断是否isCancel来确定是否继续接下来的操作 2、依赖关系，如果存在相互依赖的环就会造成死锁 3、单纯的NSIvocationOperation和NSBlockOperation的start方法是同步执行的，要放到OperationQueue里才能实现异步执行 4、自定义Operation的时候，如果不重新它的状态方法，只重写main函数(执行体在main函数里)，则它的状态可以直接使用不需要代码控制，所以说也可以通过重写isReady、isCancelled、isFinish等方法来控制状态。 5、自定义Operation的时候，main函数里的执行体同样是同步执行的，那如果要异步请求一些操作，则应该和runloop结合起来使用

```
class CusOperation:Operation{
    var over:Bool = false
    override main(){
        ConCurentQueue.async{
            sleep(1000)
            self.over = true
        }
        while(!self.over && !self.isCancelled){

 NSRunloop.current.runmode(.default,beforeDate:NSDate.distanceFuture)
        }
    }
}
```

所以总结一下：NSOperation较与GCD，就是可以添加依赖、最大并发量、控制状态。GCD是基于C语言实现的，NSOperation是基于GCD实现的！

锁

- 自旋锁 OSSpinLock 存在优先级反转的问题。就是等待线程的优先级更高，会一直占用CPU，优先级低的线程就无法释放锁；
- os_unfair_lock 用来替换OSSpinLock，但是它并非忙等的锁；
- pthread_mutex 互斥锁，等待线程时会进行休眠。它同时含有多种锁，比如pthread_mutex—递归锁、pthread_mutex—条件锁；
- NSLock是对pthread_mutex 普通锁的封装；
- NSRecursiveLock是对pthread_mutex—递归锁的封装；
- NSCondition是对phread_mutex—条件锁的封装；
- NSConditionLock是对NSCondition的又一层封装；
- @synchronized是对pthread_mutex—递归锁的封装；

另外信号量和同步队列也能实现类似锁的操作！

实践与优化

runtime 及其埋点和越界崩溃

runtime是指在程序运行期间才能确认对应的数据类型和方法调用等，OC有三大动态特性动态类型、动态加载、动态绑定，这都是通过runtime的机制实现的。

- 埋点 埋点主要是为了抓取一些主要的日志数据，然后上报。以便于在一些特殊的节点，可以通过日志查找到相关的数据和用户操作信息，更利于解决问题。 首先我们需要对日志进行分类，1是基本的接口日志(主要运用于测试)，2是用户的主要操作日志，3是主要节点日志(比如购买的信息身份的转变等)，4是异常日志(这个主要是依赖于bugly) 1、页面的统计：使用Method Swizzing代码混淆，重写UIViewController的ViewDidLoad方法。 2、点击统计：通过Method Swizzing代码混淆，hook sendAction:to:forEvent:方法。 3、tableview的cell点击：通过hook setDelegate方法，在设置代理时，再Swizzing代理实现了的didSelect方法。 4、然后剩余的关键节点的日志就得使用代码进行log了。

然后就是上传服务器，可以在日志达到一定容量时进行上传。记得添加断点续传

- 越界崩溃 首先越界崩溃也是通过runtime的方法混淆来做的，但是需要注意的是：抽象工厂模式的一些类的具体实现是隐藏的，也就是说，它们对应的真实名字分别是：NSArray----- NSArrayI；NSMutablArray ---- NSArrayM；NSDictionary---NSDictionaryI；NSMutableDictionary ---- NSDictionaryM。 然后使用swizing混淆相应方法。比如objectAtIndex

runloop与tableview卡顿

1、RunLoop是系统内部维护事件循环的一个对象。而事件循环可以不断地对消息或事件进行管理：当没有消息时，会将进程从用户态切向内核态，由此对当前线程进行休眠，以避免资源占用；当有消息需要处理时，会将进程从内核态切到用户态，以便及时唤醒当前线程（用户态指的是上层应用程序的活动空间；内核态指内核资源，为上层应用程序提供资源。）。所以，RunLoop可以保证进程不退出，可以监听事件，可以定时渲染UI，可以调节CPU的工作等

- runloop mode：包括default、tracking(界面跟踪mode，比如scrollview滑动)、UIInitialization(刚启动app时mode)、eventreceive(接受系统事件的mode)、common(其实是同步source、timer、observer到多个mode的技术方案)
- runloop消息循环机制：在runloop启动的时候会发出KCFRunLoopEntry通知，告知即将进去runloop，那一启动后就进入到了用户态模式。所以优先处理timer和source0（需要手动唤醒线程）事件，接着如果有source1(有唤醒线程的能力)事件，就接着处理source1事件，没有就发出beforWaiting通知进入休眠状态。在休眠期间，如果收到Timer、source1、手动事件，则又会被唤醒。

2、UI显示原理：首先UI视图的显示是由CPU和GPU协作完成的。CPU的工作主要是负责UI布局、文本计算、位图的渲染及一些图片编解码等准备工作，最终位图将通过Core Animation框架提交给GPU。GPU的工作主要就是图层的渲染和纹理的合成，具体就包括顶点着色、光栅化等。GPU产生的最终结果将放到帧缓冲区中，当视图控制器接收到VSync信号后，就会去帧缓冲区提取帧视图，最终显示在屏幕上。一个VSync信号的发送周期是一个runloop的迭代周期，刷新屏幕大概是60帧/s，也就是16.7ms左右。也就是说CPU和GPU的合成工作必须在一个VSync周期之内，否则就容易造成掉帧，掉帧最终也就造成了卡顿等性能问题。所以一般我们解决卡顿等性能优化就可以从CPU和GPU两个方面入手！对于CPU，我们可以将一些较耗时的对象的生成释放、文本高度等复杂的计算放到子线程中，亦或通过实现layer的delegate方法displaylayer调用Core Graphics的API进行异步绘制；而对于GPU的话，就应该要避免离屏渲染，因为离屏渲染会在当前显示的屏幕缓冲区之外再开辟缓冲区，导致性能的消耗。而如果我们实现了CALayer的一些必须合成后使用的属性就可能会触发离屏渲染，比如阴影、图层蒙版、光栅、圆角与maskToBounds一起使用等！

3、创建一个常驻线程

```
#pragma mark -- RunLoop 实现常驻线程
static BOOL runAlways = YES;
- (void)usethread {
    [self performSelector:@selector(subThreadRun) onThread:self.thread
withObject:nil waitUntilDone:NO];
}

//线程安全的方式创建thread
- (NSThread *)thread {
    if (_thread == nil) {
        @synchronized (self) {
            _thread = [[NSThread alloc] initWithTarget:self
selector:@selector(runThread) object:nil];
            [_thread setName:@"com.fcf.thread"];
            [_thread start];// 启动
        }
    }
    return _thread;
}

- (void)runThread {
    //方法一：创建一个可控的runloop
    //创建一个source
    CFRunLoopSourceContext context =
{0,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL};
    CFRunLoopSourceRef source = CFRunLoopSourceCreate(kCFAllocatorDefault,
0, &context);

    //创建runloop，同时向runloop的defaultmode添加source CFRunLoopGetCurrent类似
懒加载方法
    CFRunLoopAddSource(CFRunLoopGetCurrent(), source,
kCFRunLoopDefaultMode);

    while (runAlways) {
        @autoreleasepool {
            //将当前runloop运行在kCFRunLoopDefaultMode下
            CFRunLoopRunInMode(kCFRunLoopDefaultMode, 1.0e10, true);
        }
    }

    //当runaway为NO时跳出runloop，线程退出
    CFRunLoopRemoveSource(CFRunLoopGetCurrent(), source,
kCFRunLoopDefaultMode);
    CFRelease(source);

    //方法二：创建一个一直存在的runloop
//    @autoreleasepool {
//        NSRunLoop * runloop = [NSRunLoop currentRunLoop];
//        [runloop addPort:[NSMachPort port] forMode:NSRunLoopCommonModes];
```

```
//        NSLog(@"启动RunLoop前--%@",runloop.currentMode);
//        [runloop run];
//    }

}

- (void) subThreadRun {
    NSLog(@"启动RunLoop后--%@",[NSRunLoop currentRunLoop].currentMode);
    NSLog(@"%@----子线程任务开始",[NSThread currentThread]);
    [NSThread sleepForTimeInterval:3.0];
    NSLog(@"%@----子线程任务结束",[NSThread currentThread]);
}
```

4、UITableView优化：

- 可以利用UI渲染的原理先对cell进行一些处理，比如说把高度计算、对象生成这些先在reload之前做好，然后cell上控件 实现异步绘制等，比如使用AsynDisplayKit，其次尽量避免离屏渲染。
- 也可以利用runloop，利用runloop休眠时机执行耗时操作。方法就是使用一个定时器，然后将定时器放置到runloop里面，然后实现一个监听方法监听runloop的runloopbeforewaiting状态。然后在observer方法中执行耗时任务。

音视频上传下载（断点续传）

断点下载：主要是利用Alamofire的 Downloader resume方法，传入已下载好的缓存的data。

断点续传：断点续传稍微复杂一点，但是和断点下载原理是一样的。大概流程就是我们data数据分割成多个片段；然后建一个小的数据model，它里边包含了每个片段的data数据，以及对应片段标记上传状态(waiting、finish、loading)，之后存储到本地。然后使用信号量和group将片段一个一个上传。如果网络中断了，下次进来的时候，查看一下是否有需要上传的文件，然后从上传断开处接着上传就好了。

```
//模拟代码，没有做本地缓存操作
{
        //分片阶段
        NSString *ps = [P stringByAppendingPathComponent:@"bz.jpg"];
    NSData* imgData =[[NSData alloc] initWithContentsOfFile:ps];
    NSUInteger totalLength = [imgData length];
    NSUInteger minBlock = 10*1024;
    NSUInteger count = totalLength/minBlock +(totalLength%minBlock?1:0);
    NSUInteger lastIndexLength = totalLength - (count -1)*minBlock;
    NSMutableArray *list = [[NSMutableArray alloc] initWithCapacity:count];
    for(NSUInteger i =0; i<count-1; i++){
        NSData *d = [imgData subdataWithRange:NSMakeRange(i*(minBlock),
minBlock)];
        [list addObject:d];
    }
    //最后一片
    NSData *d = [imgData subdataWithRange:NSMakeRange((count -1)*minBlock,
lastIndexLength)];
     [list addObject:d];
```

```
    /*
    将分片data重新整合，实验整合的图片能不能正常使用
    NSMutableData *mutableData = [[NSMutableData alloc] init];
    for (NSData *d in list) {
        [mutableData appendData:d];
    }
    UIImage *img = [[UIImage alloc] initWithData:mutableData];
    */

    //上传
    dispatch_group_t group = dispatch_group_create();
    dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);//使用信号量控
制每次
    dispatch_queue_t queue =
dispatch_queue_create(NULL,DISPARCH_QUEUE_SERIAL)//使用串行，因为要保证片段是按顺序
上传的
    for (int i =0; i<[list count]  ; i++) {
         dispatch_group_async(group, queue, ^{
         NSData *d = list[i];
        AFHTTPSessionManager *manager = [AFHTTPSessionManager manager];
        manager.responseSerializer = [AFHTTPResponseSerializer serializer];
        [manager POST:@"http://10.0.1.4:8778/hb/user/uploaddd.do"
parameters:@{@"tmpId":@(i)}
constructingBodyWithBlock:^(id<AFMultipartFormData>  _Nonnull formData) {
            [formData appendPartWithFileData:d name:@"uploadFile"
fileName:@"uploadFile" mimeType:@"application/octet-stream"];
        } progress:^(NSProgress * _Nonnull uploadProgress) {
            NSLog(@"%lld     ------
%lld",uploadProgress.totalUnitCount,uploadProgress.completedUnitCount);
        } success:^(NSURLSessionDataTask * _Nonnull task, id  _Nullable
responseObject) {
            NSLog(@"%d",i);//顺序打印
            dispatch_semaphore_signal(semaphore);
        } failure:^(NSURLSessionDataTask * _Nullable task, NSError * _Nonnull
error) {
            NSLog(@"%d",i);//顺序打印
            dispatch_semaphore_signal(semaphore);
        }];
            dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
        });
    }


    dispatch_group_notify(group, queue, ^{
        //所有请求返回数据后执行


    });
}
```

播放器

通过kvo的方式监听"Status"和"loadedTimeranges"来分别监听播放状态和缓存时长。

FPS检测原理：CADisplayLink

直播的流程和API:

IM流程：

音视频相关:

常用三方库的实现原理和总结

3、Alamofire、SwiftJSon、SDWebImage、AsyncDisplayKit、realm大概使用与实现原理

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

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！

SDWebImage：

整体框架：SDWebImage更多的是封装的UIKit的一些分类方法，比如说UIImageView+WebCache。主要功能是由SDWebImageManager进行管理，在此之下主要分为两部分：SDImageCache和SDWebImageDownloader，SDImageCache又同时分为磁盘缓存和内存缓存。 加载图片的流程：通过图片URL的hash值作为key值去查找内存缓存，如果内存缓存找不到则查找磁盘缓存，如果仍然没有查找到就去进行网络下载

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。

Realm：

https://juejin.im/entry/5a1d44a6f265da432f30dd09

算法

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
```