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

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

- CTMeditor

组件化、模块化、路由化

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。

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

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

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，找到后将文件名放到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，起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改图片的时候；

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，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。

https://juejin.im/post/5c84bd676fb9a049e702ecd8 https://hnxczk.github.io/blog/articles/image_decode.html#imagewithcontentsoffile

SDWebImage：

https://www.jianshu.com/p/06f0265c22eb

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

播放器 与 音视频相关

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。

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

直播框架与实践

IM框架：

单元测试与可持续集成

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

Swift Package Manager

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

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。

项目优化相关：

检测

优化

最佳实践

https://zhuanlan.zhihu.com/p/96963676

https://zhuanlan.zhihu.com/p/38284322

https://juejin.im/post/5d3ee3e7e51d4561e53538a1#heading-22

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
```