1.UITableview创建方式

(1)自定义高度

1>新建一个继承自UITableViewCell的类
2>重写initWithStyle:reuseIdentifier:方法
3>添加所有需要显示的子控件(不需要设置子控件的数据和frame, 子控件要添加到contentView中)
4>进行子控件一次性的属性设置(有些属性只需要设置一次, 比如字体\固定的图片)
5>提供2个模型
数据模型: 存放文字数据\图片数据
frame模型: 存放数据模型\所有子控件的frame\cell的高度
6>cell拥有一个frame模型(不要直接拥有数据模型)
7>重写frame模型属性的setter方法: 在这个方法中设置子控件的显示数据和frame

(2) 自定义高度原理

A 手动计算
1> 由于heightForRow比cellForRow方法先调用，创建frame模型包含微博模型，重写微博模型赋值set方法，提前计算cell子控件的frame并保存，heightForRow方法中取出frame模型中保存的高度，实现自定义高度cell
2> 设置最大尺寸、文本属性，根据文本内容计算正文内容展示尺寸
3> cellForRow中创建自定义cell包含frame属性，重写frame属性set方法创建cell子控件并赋值frame模型保存的子控件尺寸
B. 自动计算
1> 首先设置行高使用autolayout自动计算并预估高度
2> 在stroboard中对cell内容进行自动布局，注意设置图片距离底部约束，cellForRow中创建storyboard中对应标记的自定义cell
3> 由于正文内容的不确定性，设置label多行，拖线图片高度约束，根据图片有无，设置代码设置高度约束

2.Swift和OC的区别

苹果宣称 Swift 的特点是：

(1) 快速、现代、安全、互动，而且明显优于 Objective-C 语言
(2) 可以使用现有的 Cocoa 和 Cocoa Touch 框架
(3) Swift 取消了 Objective C 的指针/地址等不安全访问的使用
(4) 提供了类似 Java 的名字空间(namespace)、泛型(generic)var、运算对象重载(operator overloading
(5) Swift 被简单的形容为 “没有 C 的 Objective-C”（Objective-C without the C）
(6) 为苹果开发工具带来了Xcode Playgrounds功能，该功能提供强大的互动效果，能让Swift源代码在撰写过程中实时显示出其运行结果；
(7) 基于C和Objective-C，而却没有C的一些兼容约束；
(8) 采用了安全的编程模式；
(9) 界面基于Cocoa和Cocoa Touch框架；

(10) 舍弃 Objective C 早期应用 Smalltalk 的语法，保留了Smalltalk的动态特性，全面改为句点表示法

(11) 类型严谨 对比oc的动态绑定

3.synthesize&denamic

(1)通过@synthesize 指令告诉编译器在编译期间产生 getter/setter 方法。

(2)通过@dynamic 指令，自己实现方法。

有些存取是在运行时动态创建的，如在 CoreData 的 NSManagedObject 类使用的某些。如果你想这些情况下，声明和使用属性，但要避免缺少方法在编译时的警告，你可以使用@dynamic 动态指令，而不是@synthesize 合成指令。

4.在项目开发中常用 的开发工具有哪些?

Instrument beyondCompare git corn stone application loadder idea(编写 h5 和 RN)

5.UITableView&UICollection

UICollectionView 是 iOS6 新引进的 API，用于展示集合视图，布局更加灵活，其用法类似于 UITableView。而 UICollectionView、UICollectionViewCell 与 UITableView、UITableViewCell 在用法上有相似 的也有不同的，下面是一些基本的使用方法：

对于 UITableView，仅需要 UITableViewDataSource,UITableViewDelegate 这两个协议，使用 UICollectionView 需要实现 UICollectionViewDataSource,UICollectionViewDelegate，UICollectionViewDelegateFlowLayout 这三个协议，这是因为 UICollectionViewDelegateFlowLayout 实际上是 UICollectionViewDelegate 的一个子协议，它继承 了 UICollectionViewDelegate，它的作用是提供一些定义 UICollectionView 布局模式的函数

5.NSProxy&NSObject

NSObjetct:

NSObject 协议组对所有的 Object－C 下的 objects 都生效。 如果 objects 遵从该协议，就会被看作是 first－class objects(一级类)。另外,遵从该协议的 objects 的 retain,release,autorelease 等方法也服从 objects 的管理和在 Foundation 中定义的释放方法。一些容器中的对象也可以管理这些 objects，比如 说 NSArray 和 NSDictionary 定义的对象。 Cocoa 的根类也遵循该协议，所以所有继承 NSObjects 的 objects 都有遵循该协议的特性。

NSProXY：

NSProxy 是一个虚基类，它为一些表现的像是其它对象替身或者并不存在的对象定义一套 API。一般的，发送给代理的消息被转发给一个真实的对象或者代理本身 load(或者将本身转换成)一个真实的对象。NSProxy 的基类可以被用来透明的转发消息或者耗费巨大的对象的 lazy 初始化。

7.传值通知&推送通知（本地&远程）

传值通知：类似通知，代理，Block 实现值得传递

推送通知：推送到用户手机对应的 App 上（主要是不再前台的情况）,用户获得资源的一种手段。普通情况下，都是客户端主动的 pull。推送则是服务器端主动 push。

本地通知：local notification，用于基于时间行为的通知，比如有关日历或者 todo 列表的小应用。另外，应用　如果在后台执行，iOS 允许它在受限的时间内运行，它也会发现本地通知有用。比如，一个应用，在后台运行，向应用的服务器端获取消息，当消息到达时，比如下载更新版本的提示消息，通过本地通知机制通知用户。

本地通知是 UILocalNotification 的实例，主要有三类属性：

scheduled time，时间周期，用来指定 iOS 系统发送通知的日期和时间；

notification type，通知类型，包括警告信息、动作按钮的标题、应用图标上的 badge（数字标记）和播放的声音；

自定义数据，本地通知可以包含一个 dictionary 类型的本地数据。

对本地通知的数量限制，iOS 最多允许最近本地通知数量是 64 个，超过限制的本地通知将被 iOS 忽略。

远程通知（需要服务器）。流程大概是这样的

1> 生成 CertificateSigningRequest.certSigningRequest 文件

2> 将 CertificateSigningRequest.certSigningRequest 上传进 developer，导出.cer 文件

3> 利用 CSR 导出 P12 文件

4> 需要准备下设备 token 值（无空格）

5> 使用 OpenSSL 合成服务器所使用的推送证书

一般使用极光/友盟推送，步骤是一样的，只是我们使用的服务器是极光的，不需要自己大服务器！

8.第三方库&第三方平台

第三方库:一般是指大牛封装好的一个框架（库），或者第三方给我们提供的一个库，这里比较笼统 *

第三方平台：指第三方提供的一些服务，其实很多方面跟第三方库是一样的，但是还是存在一些区别。

库：AFN，ASI，Alomofire，MJRefresh，MJExtension，MBProgressHUD

平台：极光，百度，友盟，Mob，环信

9. imageName 和 ImageWithContextOfFile 的区别？哪个性能高

用 imageNamed 的方式加载时，图片使用完毕后缓存到内存中，内存消耗多，加载速度快。即使生成的对象被 autoReleasePool 释放了，这份缓存也不释放，如果图像比较大，或者图像比较多，用这种方式会消耗很大的内存。

imageNamed 采用了缓存机制，如果缓存中已加载了图片，直接从缓存读就行了，每次就不用再去读文件了，效率会更高 。

ImageWithContextOfFile 加载，图片是不会缓存的，加载速度慢。

大量使用 imageNamed 方式会在不需要缓存的地方额外增加开销 CPU 的时间.当应用程序需要加载一张比较大的图片并且使用一次性，那么其实是没有必要去缓存这个图片的，用 imageWithContentsOfFile 是最为经济的方式,这样不会因为 UIImage 元素较多情况下，CPU 会被逐个分散在不必要缓存上浪费过多时间.

10. NSCache&NSDcitionary

NSCache 与可变集合有几点不同：

NSCache 类结合了各种自动删除策略，以确保不会占用过多的系统内存。如果其它应用需要内存时，系统自动执行这些策略。当调用这些策略时，会从缓存中删除一些对象，以最大限度减少内存的占用。

NSCache 是线程安全的，我们可以在不同的线程中添加、删除和查询缓存中的对象，而不需要锁定缓存区域。

不像 NSMutableDictionary 对象，一个缓存对象不会拷贝 key 对象。NSCache 和 NSDictionary 类似，不同的是系统回收内存的时候它会自动删掉它的内容。

(1)可以存储(当然是使用内存)

(2)保持强应用，无视垃圾回收. =>这一点同 NSMutableDictionary

(3)有固定客户.

位运算

NSCache 特点：a> 线程安全的 b> 当内存不足的时候,自动释放 c> 缓存数量和缓存成本区别NSMutableDictionary
1> 不能也不应该遍历 2> NSCache对key强引用,NSMutableDictionary对key进行copy

11. UIView 的 setNeedsDisplay 和 setNeedsLayout 方法

(1)在 Mac OS 中 NSWindow 的父类是 NSResponder，而在 iOS 中 UIWindow 的父类是 UIVIew。程序一般只有一个窗口但是会又很多视图。

(2)UIView 的作用：描画和动画，视图负责对其所属的矩形区域描画、布局和子视图管理、事件处理、可以接收触摸事件、事件信息的载体、等等。

(3)UIViewController 负责创建其管理的视图及在低内存的时候将他们从内存中移除。还为标准的系统行为进行响应。

(4)layOutSubViews 可以在自己定制的视图中重载这个方法，用来调整子视图的尺寸和位置。

(5)UIView 的 setNeedsDisplay(需要重新显示，绘制)和 setNeedsLayout(需要重新布局)方法。首先两个方法都是异步执行的。而 setNeedsDisplay 会调用自动调用 drawRect 方法，这样可以拿到 UIGraphicsGetCurrentContext，就可以画画 了。而 setNeedsLayout 会默认调用 layoutSubViews，就可以处理子视图中的一些数据。

综上所述：setNeedsDisplay 方便绘图，而 layoutSubViews 方便出来数据。setNeedDisplay 告知视图它发生了改变，需要重新绘制自身，就相当于刷新界面.

12、 UILayer&UIView

UIView 是 iOS 系统中界面元素的基础，所有的界面元素都继承自它。它本身完全是由 CoreAnimation 来实现的(Mac 下似乎不是这 样)。它真正的绘图部分，是由一个叫 CALaye(Core Animation Layer）的类来管理。UIView 本身，更像是一个 CALayer 的管理器，访问它的跟绘图和跟坐标有关的属性，例如 frame，bounds 等等， 实际上内部都是在访问它所包含的 CALayer 的相关属性。

UIView 有个重要属性 layer，可以返回它的主 CALayer 实例。

UIView 的 CALayer 类似 UIView 的子 View 树形结构，也可以向它的 layer 上添加子 layer，来完成某些特殊的表示。即 CALayer 层是可以嵌套的。

UIView 的 layer 树形在系统内部，被维护着三份 copy。分别是逻辑树，这里是代码可以操纵的；动画树，是一个中间层，系统就在这一层上更改属性，进行各种渲染操作；显示树，其内容就是当前正被显示在屏幕上得内容。

动画的运作：对 UIView 的 subLayer（非主 Layer）属性进行更改，系统将自动进行动画生成，动画持续时间的缺省值似乎是 0.5 秒。

坐标系统：CALayer 的坐标系统比 UIView 多了一个 anchorPoint 属性，使用 CGPoint 结构表示，值域是 0~1，是个比例值。

渲染：当更新层，改变不能立即显示在屏幕上。当所有的层都准备好时，可以调用 setNeedsDisplay 方法来重绘显示。

变换：要在一个层中添加一个 3D 或仿射变换，可以分别设置层的 transform 或 affineTransform 属性。

变形：Quartz Core 的渲染能力，使二维图像可以被自由操纵，就好像是三维的。图像可以在一个三维坐标系中以任意角度被旋转，缩放和倾斜。CATransform3D 的一套方法提供了一些魔术般的变换效果。

13、layoutSubViews&drawRects

layoutSubviews 在以下情况下会被调用(视图位置变化是触发)：

1、init 初始化不会触发 layoutSubviews。

2、addSubview 会触发 layoutSubviews。

3、设置 view 的 Frame 会触发 layoutSubviews，当然前提是 frame 的值设置前后发生了变化。

4、滚动一个 UIScrollView 会触发 layoutSubviews。

5、旋转 Screen 会触发父 UIView 上的 layoutSubviews 事件。

6、改变一个 UIView 大小的时候也会触发父 UIView 上的 layoutSubviews 事件。

7、直接调用 setLayoutSubviews。

drawRect 在以下情况下会被调用：

1、如果在 UIView 初始化时没有设置 rect 大小，将直接导致 drawRect 不被自动调用。drawRect 掉用是在 Controller->loadView, Controller->viewDidLoad 两方法之后掉用的.所以不用担心在 控制器中,这些 View 的 drawRect 就开始画了.这样可以在控制器中设置一些值给 View(如果这些 View draw 的时候需要用到某些变量 值).

2、该方法在调用 sizeToFit 后被调用，所以可以先调用 sizeToFit 计算出 size。然后系统自动调用 drawRect:方法。

3、通过设置 contentMode 属性值为 UIViewContentModeRedraw。那么将在每次设置或更改 frame 的时候自动调用 drawRect:。

4、直接调用 setNeedsDisplay，或者 setNeedsDisplayInRect:触发 drawRect:，但是有个前提条件是 rect 不能为 0。

drawRect 方法使用注意点：

1、 若使用 UIView 绘图，只能在 drawRect：方法中获取相应的 contextRef 并绘图。如果在其他方法中获取将获取到一个 invalidate 的 ref 并且不能用于画图。drawRect：方法不能手动显示调用，必须通过调用 setNeedsDisplay 或 者 setNeedsDisplayInRect，让系统自动调该方法。

2、若使用 calayer 绘图，只能在 drawInContext：中（类似鱼 drawRect）绘制，或者在 delegate 中的相应方法绘制。同样也是调用 setNeedDisplay 等间接调用以上方法 3、若要实时画图，不能使用 gestureRecognizer，只能使用 touchbegan 等方法来掉用 setNeedsDisplay 实时刷新屏幕

14、UDID&UUID

UDID 是 Unique Device Identifier 的缩写,中文意思是设备唯一标识.

在很多需要限制一台设备一个账号的应用中经常会用到,在 Symbian 时代,我们是使用 IMEI 作为设备的唯一标识的,可惜的是 Apple 官方不允许开发者获得设备的 IMEI.

[UIDevice currentDevice] uniqueIdentifier]

但是我们需要注意的一点是,对于已越狱了的设备,UDID 并不是唯一的.使用 Cydia 插件 UDIDFaker,可以为每一个应用分配不同的 UDID. 所以 UDID 作为标识唯一设备的用途已经不大了.

UUID 是 Universally Unique Identifier 的缩写,中文意思是通用唯一识别码.

由网上资料显示,UUID 是一个软件建构的标准,也是被开源软件基金会(Open Software Foundation,OSF)的组织在分布式计算环境(Distributed Computing Environment,DCE)领域的一部份.UUID 的目的,是让分布式系统中的所有元素,都能有唯一的辨识资讯,而不需要透过中央控制端来做辨识资 讯的指定.

备注：UDID 并不是一定不会变，如：重新开关机、手机越狱都会变。决绝办法是将 UDID 保存在钥匙串中，用时去钥匙串中取。UUID 倒是永远不变的，使用 xcode 可以获取，但是没办反通过代码获取。

15、CPU&GPU

CPU:中央处理器（英文 Central Processing Unit）是一台计算机的运算核心和控制核心。CPU、内部存储器和输入/输出设备是电子计算机三大核心部件。其功能主要是解释计算机指令以及处理计算机软件中的数据。

GPU:英文全称 Graphic Processing Unit，中文翻译为“图形处理器”。一个专门的图形核心处理器。GPU 是显示卡的“大脑”，决定了该显卡的档次和大部分性能，同时也是 2D 显示卡和 3D 显示卡的区别依据。2D 显示芯片在处理 3D 图像和特效时主要依赖 CPU 的处理能力，称为“软加速”。3D 显示芯片是将三维图像和特效处理功能集中在显示芯 片内，也即所谓的“硬件加速”功能。

16、点（pt）&像素（px）

像素（pixels）是数码显示上最小的计算单位。在同一个屏幕尺寸，更高的 PPI（每英寸的像素数目），就能显示更多的像素，同时渲染的内容也会更清晰。

点（points）是一个与分辨率无关的计算单位。根据屏幕的像素密度，一个点可以包含多个像素（例如，在标准 Retina 显示屏上 1 pt 里有 2 x 2 个像素）。

当你为多种显示设备设计时，你应该以“点”为单位作参考，但设计还是以像素为单位设计的。这意味着仍然需要以 3 种不同的分辨率导出你的素材，不管你以哪种分辨率设计你的应用。

17、属性与成员变量：

成员变量是不与外界接触的变量，应用于类的内部，如果你说那用@Public 外部不就是可以访问了么。简单的说 public 只能适当使用，不要泛滥，否则就像你把钥匙插在你自己家门上了。谁来都可以开门。毫无安全性。

由于成员变量的私有性，为了解决外部访问的问题就有了属性变量。属性变量个人认为最大的好处就是让其他对象访问这个变量。而且你可以设置只读、可写等等属性，同时设置的方法我们也可以自己定义。记住一点，属性变量主要是用于与其他对象相互交互的变量

如果对于上面所说还是含糊不清那就记住这几点吧！

1. 只有类内使用，属性为 private，那么就定义成员变量。
2. 如果你发现你需要的这个属性需要是 public 的，那么毫不犹豫就用属性在.h 中定义。
3. 当你自己内部需要 setter 实现一些功能的时候，用属性在.m 中定义。
4. 当你自己内部需要 getter 实现一些功能的时候，用属性在.m 中定义。

20.全局变量和静态变量的区别

1> 修饰符

全局变量在声明源文件之外使用,需要extern引用一下; 静态变量使用static来修饰

2> 存储地址

两者都是存储在静态存储区,非堆栈上,它们与局部变量的存储分开

3> 生命周期

两者都是在程序编译或加载时由系统自动分配的,程序结束时消亡

4> 外部可访问性

全局变量在整个程序的任何地方均可访问,而静态变量相当于面向对象中的私有变量,他的可访问性只限定于声明它的那个源文件,即作用于仅局限于本文件中

21、分类 拓展 协议中哪些可以声明属性?

都可以,但分类和协议创建的属性只相当于方法,但是内部没有对成员变量的操作(无法创建成员变量),拓展可以(私有成员变量)

代理中声明属性,没有实际创建成员变量,相当于声明了属性名对应的访问方法,遵守协议的类需要实现对应的访问器方法,否则运行报错

分类中声明属性,警告提示需要手动实现访问器方法(Swift中叫计算型属性),而分类中不能创建成员变量,可以在手写访问器方法中使用runtime的 objc_setAssociatedObject方法关联对象间接创建属性(静态库添加属性)

拓展里可以声明属性,直接可以使用

22 继承和类别的区别

答:

(1) 使用继承:

继承可以增加，修改或者删除方法，并且可以增加属性。

添加新方法和父类方法一致,但父类方法仍需要使用

(2) 类别:

1> 针对系统提供的一些类,系统本身不提倡继承,因为这些类的内部实现对继承有所限制

2> 类别可以将自己构建的类中的方法进行分组,对于大型的类,提高可维护性

(3) 分类的作用

1> 将类的实现分散到多个不同文件或多个不同框架中。

2> 创建对私有方法的前向引用。

3> 向对象添加非正式协议。

(非正式协议:即NSObject的分类,声明方法可以不实现,OC2.0以前protocal没有@optional,主要使用分类添加可选协议方法

oc中声明方法不实现,不调用则只警告不报错

正式协议的优点:可继承,泛型约束

如kvo的observeValueForKeyPath属于nsobject的分类,且不需要调父类,说明可选实现该方法,没警告可能是编译器规则过滤)

4> category 可以在不获悉，不改变原来代码的情况下往里面添加新的方法，只能添加，不能删除修改。并且如果类别和原来类中的方法产生名称冲突，则类别将覆盖原来的方法，因为类别具有更高的优先级。

（5）分类的局限性

无法向类中添加新的实例变量，类别没有位置容纳实例变量。

无法添加实例变量的局限可以使用字典对象解决。

23. category&extension

类别主要有三个作用

(1)可以将类的实现分散到多个不同文件或多个不同框架中，方便代码管理。也可以对框架提供类的扩展（没有源码，不能修改）。

(2)创建对私有方法的前向引用：如果其他类中的方法未实现，在你访问其他类的私有方法时编译器报错这时使用类别，在类别中声明这些方法（不必提供方法实现），编译器就不会再产生警告

(3)向对象添加非正式协议：创建一个 NSObject 的类别称为“创建一个非正式协议”，因为可以作为任何类的委托对象使用。

他们的主要区别是:

1、形式上来看，extension 是匿名的 category。

2、extension 里声明的方法需要在 main implementation 中实现，category 不强制要求。

3、extension 可以添加属性（变量），category 不可以。

Category 和 Extension 都是用来给已定义的类增加新的内容的。

4、Category 和原有类的耦合更低一些，声明和实现都可以写在单独的文件里。但是只能为已定义类增加 Method，而不能加入实例变量。

5、extensions可以认为是一个私有的Category。

Extension 耦合比较高，声明可以单独写，但是实现必须写在原有类的@implementation 中。可以增加 Method 和实例变量。

Extension 给人感觉更像是在编写类时为了封装之类的特性而设计，和类是同时编写的。而 category 则是在用到某一个 framework 中的类时临时增加的特性。

Extension 的一个特性就是可以重新声明一个实例变量，将之从 readonly 改为对内 readwrite.

使用 Extension 可以更好的封装类，在 h 文件中能看到的都是对外的接口，其余的实例变量和对内的@property 等都可以写在 Extension，这样类的结构更加清晰。

24.字符串常用处理

1. 字符串比较

```
NSString *a = @“hello”;
NSString *b = [NSString stringWithFormat:@hello”];
if (a == b){
     nslog(@“a==b”);  }
if ([a isEqualToString: b]){
     nslog(@“a isEqualToString b”);  }
```

== 比较变量中保存的数值(地址)　速度快　　内容同,可能地址不同(常量区,堆区)
isEqualTo 比较字符串　非常耗时

2> 字符串截取
截取字符串”20 | http://www.baidu.com”中,”|”字符前面和后面的数据,分别输出它们。

```
NSString * str = @"20 | http://www.baidu.com";
NSArray *array = [str componentsSeparatedByString:@"|"]; //这是分别输出的截取后的字符串
for (int i = 0; i<[array count]; ++i) {
NSLog(@"%d=%@",i,[array objectAtIndex:i]);
}
```

3> 格式

```
NSString *str1 = [NSString stringWithFormat:@"a"b”];    //报错，a”后加b非法
NSString *str2 = [NSString stringWithFormat:@“a""b”];   //显示  ab
NSString *str3 = [NSString stringWithFormat:@“a\"b”];   //显示  a”b  反斜杠转义
```

25.NSArray和NSDictionary

1> iOS遍历数组/字典的方法

数组：for 循环　　forin　enumerateObjectsUsingBlock（正序）enumerateObjectsWithOptions:usingBlock:(多一个遍历选项,不保证顺序)
字典:

```
1. for(NSString *object in [testDic allValues])
2. for(id akey in [testDic allKeys]){
[sum appendString:[testDic objectForKey:akey]];  }
3. [testDic enumerateKeysAndObjectsUsingBlock:^(idkey,idobj,BOOL*stop) {
   [sum appendString:obj];  } ];
```

速度: 对于数组, 增强for最快,普通for和block速度差不多,增强最快是因为增强for语法会

对容器里的元素的内存地址建立缓冲,遍历的时候直接从缓冲中取元素地址而不是通过调用方法来获取,所以效率高.这也是使用增强for时不能在循环体中修改容器元素的原因之一(可以在循环体中添加标记,在循环体外修改元素)

对于字典,allValues最快,allKey和block差不多,原因是allKey需要做objcetForKey的方法

26.如何避免循环引用

两个对象相互强引用，都无法release，解决办法为一个使用strong，一个使用assign（weak）

27.CFSocket使用有哪几个步骤。

答：创建 Socket 的上下文；创建 Socket ；配置要访问的服务器信息；封装服务器信息；连接服务器；

28.oc几种操作Socket的方法？

答： CFNetwork 、CFSocket 和 BSD Socket 。AsyncSocket

29.解析XML文件有哪几种方式？

答：以 DOM 方式解析 XML 文件；以 SAX 方式解析 XML 文件；

30. 什么是沙盒模型？哪些操作是属于私有api范畴?

答：某个iphone工程进行文件操作有此工程对应的指定的位置，不能逾越。

iphone沙箱模型的有四个文件夹documents，tmp，app，Library，永久数据存储一般放documents文件夹，得到模拟器的路径的可使用NSHomeDirectory()方法。Nsuserdefaults保存的文件在tmp文件夹里。

31. 在一个对象的方法里面：self.name= “object”；和 name =”object” 有什么不同吗?

答：self.name =”object”：会调用对象的setName()方法,

name = “object”：会直接把object赋值给当前对象的name属性。

33.创建控制器、视图的方式

创建控制器的方式

(1)通过代码的方式加载UIViewController *controller = [[UIViewController alloc] init];

(2)通过stroyboard来加载viewController

加载storyboard中箭头指向的viewController

```
UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"Main" bundle:nil];
//加载箭头指向的viewController
CZViewController *controller = [storyboard instantiateInitialViewController];
```

加载storyboard中特定标示的viewController(storyboard可以有多个controller)

```
UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"Main" bundle:nil];
CZViewController*controller= [storyboard instantiateViewControllerWithIdentifier:@"two"];
```

（3）传统方法

1.创建Xib,并指定xib的files owner为自定义控制器类(为了能连线关联管理IB的内容)

2. xib中要有内容，且xib中描述的控制器类的view属性要与xib的view控件完成关联（关联方法两种,一种是control+files owner拖线到xib中搭建的指定view控件,另一种是指定xib中的view拖线到@interface）

3.从xib加载viewController

```
CZViewController *controller = [[CZViewController alloc] initWithNibName:@"CZOneView" bundle:nil];
```

4.bundle中取出xib内容

```
CZViewController *vc = [[NSBundle mainBundle] loadNibNamed:@"Two" owner:nil options:nil].lastObject;
```

创建视图的方式

1.用系统的loadView方法创建控制器的视图

2.如果指定加载某个storyboard文件做控制器的视图，就会加载storyboard里面的描述去创建view

3.如果指定读取某个xib文件做控制器的视图，就根据指定的xib文件去加载创建

4.如果有xib文件名和控制器的类名前缀(也就是去掉controller)的名字一样的 xib文件 就会用这个xib文件来创建控件器的视图 例：控件器的名为 MJViewController xib文件名为MJView.xib 如果xib文件名后有一个字不一样就不会去根据它去创建如：MJView8.xib

5.找和控制器同名的xib文件去创建

6.如果以上都没有就创建一个空的控制器的视图;

34.UIWindow

是一种特殊的UIView,通常在一个程序中只会有一个UIWindow,但可以手 动创建多个UIWindow,同时加到程序里面。UIWindow在程序中主要起到三个作用:

1、作为容器,包含app所要显示的所有视图

2、传递触摸消息到程序中view和其他对象

3、与UIViewController协同工作,方便完成设备方向旋转的支持

35.简述内存分区情况

1).代码区：存放函数二进制代码

2).数据区：系统运行时申请内存并初始化，系统退出时由系统释放。存放全局变量、静态变量、常量

3).堆区：通过malloc等函数或new等操作符动态申请得到，需程序员手动申请和释放

4).栈区：函数模块内申请，函数结束时由系统自动释放。存放局部变量、函数参数

36. 队列和栈有什么区别

答：队列和栈是两种不同的数据容器。从”数据结构”的角度看，它们都是线性结构，即数据元素之间的关系相同。

队列是一种先进先出的数据结构，它在两端进行操作，一端进行入队列操作，一端进行出列队操作。

栈是一种先进后出的数据结构，它只能在栈顶进行操作，入栈和出栈都在栈顶操作。

37.iOS的系统架构

答： iOS的系统架构分为（ 核心操作系统层 theCore OS layer ）、（ 核心服务层theCore Services layer ）、（ 媒体层 theMedia layer ）和（ Cocoa 界面服务层 the Cocoa Touch layer ）四个层次。

38.控件主要响应3种事件

答：1). 基于触摸的事件 ； 2). 基于值的事件 ； 3).基于编辑的事件。

39.xib文件的构成分为哪3个图标？都具有什么功能。

答： File’s Owner 是所有 nib 文件中的每个图标，它表示从磁盘加载 nib 文件的对象；

First Responder 就是用户当前正在与之交互的对象；View 显示用户界面；完成用户交互；是 UIView 类或其子类。

40.简述视图控件器的生命周期。

答： loadView在controller的view为nil时调用。尽管不直接调用该方法，如手动创建自己的视图，那么应该覆盖这个方法并将它们赋值给试图控制器的 view 属性。在C的View为nil时调用，在编程实现View时调用。

viewDidLoad 只有在视图控制器将其视图载入到内存之后才调用该方法，这是执行任何其他初

始化操作的入口。在View从nib初始化的时候调用。

viewVillAppear 当试图将要添加到窗口中并且还不可见的时候或者上层视图移出图层后本视图变成顶级视图时调用该方法，用于执行诸如改变视图方向等的操作。实现该方法时确保调用[super viewWillAppear:

viewDidAppear 当视图添加到窗口中以后或者上层视图移出图层后本视图变成顶级视图时调用，用于放置那些需要在视图显示后执行的代码。确保调用［super viewDidAppear：］。

viewWillDisappear－UIViewController对象的视图即将消失、被覆盖或是隐藏时调用

viewWillDisappear－UIViewController对象的视图即将消失、被覆盖或是隐藏时调用；

viewDidDisappear－UIViewController对象的视图已经消失、被覆盖或是隐藏时调用；

viewVillUnload－当内存过低时，需要释放一些不需要使用的视图时，即将释放时调用；

viewDidUnload 当试图控制器从内存释放自己的方法的时候调用，当内存过低，释放一些不需要的视图。在这里实现将retain的view release，如果是retain的IBOutlet view 属性则不要在这里release，IBOutlet会负责release 。

41.应用的生命周期 各个程序运行状态时代理的回调：

-(BOOL)application:(UIApplication*)application willFinishLaunchingWithOptions:(NSDictionary *)launchOptions 告诉代理进程启动但还没进入状态保存
-(BOOL)application:(UIApplication*)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions 告诉代理启动基本完成程序准备开始运行
- (void)applicationWillResignActive:(UIApplication *)application 当应用程序将要入非活动状态执行，在此期间，应用程序不接收消息或事件，比如来电话了
- (void)applicationDidBecomeActive:(UIApplication *)application 当应用程序入活动状态执行，这个刚好跟上面那个方法相反
- (void)applicationDidEnterBackground:(UIApplication *)application 当程序被推送到后台的时候调用。所以要设置后台继续运行，则在这个函数里面设置即可
- (void)applicationWillEnterForeground:(UIApplication *)application 当程序从后台将要重新回到前台时候调用，这个刚好跟上面的那个方法相反。
- (void)applicationWillTerminate:(UIApplication *)application 当程序将要退出是被调用，通常是用来保存数据和一些退出前的清理工作。

42.简要说明一下 APP 的启动过程，main 文件说起，main 函数中有什么函数？作用是什么？

http://www.jianshu.com/p/3f262ae413b4

打开程序——->执行 main 函数————>UIAPPlicationMain 函数——->初始化 UIAPPlicationMain 函数(设置代理，开启事件循环)————>监听系统事件—->程序结束

先执行 main 函数，main 内部会调用 UIApplicationMain 函数

UIApplicationMain 函数作用：

(1)根据传入的第三个参数创建 UIApplication 对象或它的子类对象。如果该参数为 nil,直接使用该 UIApplication 来创建。(该参数只能传人 UIApplication 或者是它的子类)

(2)根据传入的第四个参数创建 AppDelegate 对象,并将该对象赋值给第 1 步创建的 UIApplication 对象的 delegate 属性。

(3)开启一个事件循环,循环监控应用程序发生的事件。每监听到对应的系统事件时，就会通知 AppDelegate。

main 函数作用：

（1）创建 UIApplication 对象

（2）创建应用程序代理

（3）开启时间循环，包括应用程序的循环运行，并开始处理用户事件。

43.动画有基本类型有哪几种；表视图有哪几种基本样式。

答：动画有两种基本类型：隐式动画和显式动画。

44.Cocoa Touch提供了哪几种Core Animation过渡类型？

答： Cocoa Touch 提供了 4 种 Core Animation 过渡类型，分别为：交叉淡化、推挤、显示和覆盖。

45.Quatrz 2D的绘图功能的三个核心概念是什么并简述其作用。

答：上下文：主要用于描述图形写入哪里；

路径：是在图层上绘制的内容；

状态：用于保存配置变换的值、填充和轮廓， alpha 值等。

46.iPhone OS主要提供了几种播放音频的方法？

答： SystemSound Services、AVAudioPlayer、Audio Queue Services、OpenAL

47.使用AVAudioPlayer类调用哪个框架、使用步骤？

答： AVFoundation.framework

步骤：配置 AVAudioPlayer 对象；

实现 AVAudioPlayer 类的委托方法；

控制 AVAudioPlayer 类的对象；

监控音量水平；

回放进度和拖拽播放。

48.有哪几种手势通知方法、写清楚方法名？

-(void)touchesBegan:(NSSet*)touchedwithEvent:(UIEvent*)event;

-(void)touchesMoved:(NSSet*)touched withEvent:(UIEvent*)event;

-(void)touchesEnded:(NSSet*)touchedwithEvent:(UIEvent*)event;

-(void)touchesCanceled:(NSSet*)touchedwithEvent:(UIEvent*)event;

49.ViewController的didReceiveMemoryWarning怎么被调用

答:[super didReceiveMemoryWarning];

51.用预处理指令#define声明一个常数，用以表明1年中有多少秒（忽略闰年问题）

#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL

我在这想看到几件事情：

#define 语法的基本知识（例如：不能以分号结束，括号的使用，等等）

懂得预处理器将为你计算常数表达式的值，因此，直接写出你是如何计算一年中有多少秒而不是计算出实际的值，是更清晰而没有代价的。

意识到这个表达式将使一个16位机的整型数溢出-因此要用到长整型符号L,告诉编译器这个常数是的长整型数。

如果你在你的表达式中用到UL（表示无符号长整型），那么你有了一个好的起点。记住，第一印象很重要。

52.写一个”标准"宏MIN ，这个宏输入两个参数并返回较小的一个。

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

#define MIN(A,B) ((A) <= (B) ? (A) : (B))

这个测试是为下面的目的而设的：

标识#define在宏中应用的基本知识。这是很重要的，因为直到嵌入(inline)操作符变为标准C的一部分，宏是方便产生嵌入代码的唯一方法，

对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。

三重条件操作符的知识。这个操作符存在C语言中的原因是它使得编译器能产生比 if-then-else 更优化的代码，了解这个用法是很重要的。

懂得在宏中小心地把参数用括号括起来

我也用这个问题开始讨论宏的副作用，例如：当你写下面的代码时会发生什么事？

```
least = MIN(*p++, b);
```

least = MIN(*p++, b);

结果是：

```
((*p++) <= (b) ? (*p++) : (*p++))
```

((*p++) <= (b) ? (*p++) : (*p++))

这个表达式会产生副作用，指针p会作三次++自增操作。

53.关键字const有什么含意？修饰类呢?static的作用,用于类呢?还有externc的作用，const 意味着"只读"，下面的声明都是什么意思？

```
const int a;
```

```
int const a;
```

```
const int *a;
```

int * const a;

int const * a const

const int a;

int const a;

const int *a;

int * const a;

int const * a const;

前两个的作用是一样，a是一个常整型数。

第三个意味着a是一个指向常整型数的指针（也就是，整型数是不可修改的，但指针可以）。

第四个意思a是一个指向整型数的常指针（也就是说，指针指向的整型数是可以修改的，但指针是不可修改的）。

最后一个意味着a是一个指向常整型数的常指针（也就是说，指针指向的整型数是不可修改的，同时指针也是不可修改的）。

结论：

关键字const的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。

如果你曾花很多时间清理其它人留下的垃圾，你就会很快学会感谢这点多余的信息。（当然，懂得用const的程序员很少会留下的垃圾让别人来清理的） 通过给优化器一些附加的信息，使用关键字const也许能产生更紧凑的代码。合理地使用关键字const可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少bug的出现。

1).欲阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初

始化，因为以后就没有机会再去改变它了；

2).对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指

定为 const；

3).在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；

4).对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员

变量；

5).对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。

54.关键字volatile有什么含意?并给出三个不同的例子。

答：一个定义为 volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。

下面是volatile变量的几个例子：

并行设备的硬件寄存器（如：状态寄存器）

一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

多线程应用中被几个任务共享的变量

55.一个参数既可以是const还可以是volatile吗？一个指针可以是volatile吗？

答：1).是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。

2).是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个buffer的指针时。

56.static 关键字的作用

1).函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，

因此其值在下次调用时仍维持上次的值；

2).在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；

3).在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明

它的模块内；

4).在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

5).在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的static 成员变量。

57.列举几种进程的同步机制，并比较其优缺点。

答： 原子操作 信号量机制 自旋锁 管程，会合，分布式系统

58.进程之间通信的途径

答：共享存储系统消息传递系统管道：以文件系统为基础

59.进程死锁的原因

答：资源竞争及进程推进顺序非法

60.死锁的4个必要条件

答：互斥、请求保持、不可剥夺、环路

61.死锁的处理

答：鸵鸟策略、预防策略、避免策略、检测与解除死锁

62.cocoa touch框架

答：iPhone OS 应用程序的基础 Cocoa Touch 框架重用了许多 Mac 系统的成熟模式，但是它更多地专注于触摸的接口和优化。

UIKit 为您提供了在 iPhone OS 上实现图形，事件驱动程序的基本工具，其建立在和 Mac OS X 中一样的 Foundation 框架上，包括文件处理，网络，字符串操作等。

Cocoa Touch 具有和 iPhone 用户接口一致的特殊设计。有了 UIKit，您可以使用 iPhone OS 上的独特的图形接口控件，按钮，以及全屏视图的功能，您还可以使用加速仪和多点触摸手势来控制您的应用。

各色俱全的框架 除了UIKit 外，Cocoa Touch 包含了创建世界一流 iPhone 应用程序需要的所有框架，从三维图形，到专业音效，甚至提供设备访问 API 以控制摄像头，或通过 GPS 获知当前位置。

Cocoa Touch 既包含只需要几行代码就可以完成全部任务的强大的 Objective-C 框架，也在需要时提供基础的 C 语言 API 来直接访问系统。这些框架包括：

Core Animation：通过 Core Animation，您就可以通过一个基于组合独立图层的简单的编程模型来创建丰富的用户体验。

Core Audio：Core Audio 是播放，处理和录制音频的专业技术，能够轻松为您的应用程序添加

强大的音频功能。

Core Data：提供了一个面向对象的数据管理解决方案，它易于使用和理解，甚至可处理任何应用或大或小的数据模型。

功能列表：框架分类

下面是 Cocoa Touch 中一小部分可用的框架：

音频和视频：Core Audio ，OpenAL ，Media Library ，AV Foundation

数据管理 ：Core Data ，SQLite

图形和动画 ：Core Animation ，OpenGL ES ，Quartz 2D

网络：Bonjour ，WebKit ，BSD Sockets

用户应用：Address Book ，Core Location ，Map Kit ，Store Kit

63.自动释放池是什么,如何工作

答：当您向一个对象发送一个autorelease消息时，Cocoa就会将该对象的一个引用放入到最新的自动释放.它仍然是个正当的对象，因此自动释放池定义的作用域内的其它对象可以向它发送消息。当程序执行到作用域结束的位置时，自动释放池就会被释放，池中的所有对象也就被释放。

64.sprintf,strcpy,memcpy使用上有什么要注意的地方。

1). sprintf是格式化函数。将一段数据通过特定的格式，格式化到一个字符串缓冲区中去。sprintf格式化的函数的长度不可控,有可能格式化后的字符串会超出缓冲区的大小,造成溢出。

2).strcpy是一个字符串拷贝的函数，它的函数原型为strcpy(char *dst, const char *src

将src开始的一段字符串拷贝到dst开始的内存中去，结束的标志符号为 ‘\0’，由于拷贝的长度不是由我们自己控制的，所以这个字符串拷贝很容易出错。

3). memcpy是具备字符串拷贝功能的函数，这是一个内存拷贝函数，它的函数原型为memcpy(char *dst, const char* src, unsigned int len);将长度为len的一段内存，从src拷贝到dst中去，这个函数的长度可控。但是会有内存叠加的问题。

65.你了解svn,cvs等版本控制工具么？

答： 版本控制 svn,cvs 是两种版控制的器,需要配套相关的svn，cvs服务器。scm是xcode里配置版本控制的地方。版本控制的原理就是a和b同时开发一个项目，a写完当天的代码之后把代码提交给服务器，b要做的时候先从服务器得到最新版本，就可以接着做。 如果a和b都要提交给服务器，并且同时修改了同一个方法，就会产生代码冲突，如果a先提交，那么b提交时，服务

器可以提示冲突的代码，b可以清晰的看到，并做出相应的修改或融合后再提交到服务器。

66. 什么是push

答： 客户端程序留下后门端口，客户端总是监听针对这个后门的请求，于是 服务器可以主动像这个端口推送消息。

67. 静态链接库

答：此为.a文件，相当于java里的jar包，把一些类编译到一个包中，在不同的工程中如果导入此文件就可以使用里面的类，具体使用依然是#import “ xx.h”。

68. 三大特性

1. 封装_点语法

(1)本质

```
//以下代码有什么问题
- (void)setName:(NSString *)name {
  self.name = name;
}
- (NSString *)name {
  return self.name;
}
```

(2)点语法的本质是调用类的getter方法和setter方法，如果类中没有getter方法和setter方法就不能使用点语法。

2.继承

(1)如何实现多重继承**消息转发**

forwardingTargetForSelector methodSignatureForSelector forwardInvocation

delegate和protocol 类别

http://www.cocoachina.com/ios/20130528/6295.html

3.多态

1> 什么是多态

多态：不同对象以自己的方式响应相同的消息的能力叫做多态。子类指针可以赋值给父类。

由于每个类都属于该类的名字空间，这使得多态称为可能。类定义中的名字和类定义外的名字并不会冲突。类的实例变量和类方法有如下特点：

- 和C语言中结构体中的数据成员一样，类的实例变量也位于该类独有的名字空间。

- 类方法也同样位于该类独有的名字空间。与C语言中的方法名不同，类的方法名并不是一个全局符号。一个类中的方法名不会和其他类中同样的方法名冲突。两个完全不同的类可以实现同一个方法。

方法名是对象接口的一部分。对象收到的消息的名字就是调用的方法的名字。因为不同的对象可以有同名的方法，所以对象必须能理解消息的含义。同样的消息发给不同的对象，导致的操作并不相同。

多态的主要好处就是简化了编程接口。它容许在类和类之间重用一些习惯性的命名，而不用为每一个新加的函数命名一个新名字。这样，编程接口就是一些抽象的行为的集合，从而和实现接口的类区分开来。

Objective-C支持方法名的多态，但不支持参数和操作符的多态。

2> OC中如何实现多态

在Objective-C中是通过一个叫做selector的选取器实现的。在Objective-C中，selector有两个意思， 当用在给对象的源码消息时，用来指方法的名字。它也指那个在源码编译后代替方法名的唯一的标识符。编译后的选择器的类型是SEL有同样名字的方法、也有同样的选择器。你可以使用选择器来调用一个对象的方法。

选取器有以下特点：

* 所有同名的方法拥有同样的选取器

* 所有的选取器都是不一样的

(1) SEL和@selector

选择器的类型是 SEL。@selector指示符用来引用选择器，返回类型是SEL。

例如：

SEL responseSEL; responseSEL = @selector(loadDataForTableView:); 可以通过字符串来得到选取器，例如：

responseSEL = NSSelectorFromString(@"loadDataForTableView:");也可以通过反向转换，得到方法名，例如：NSString *methodName = NSStringFromSelector(responseSEL);

(2) 方法和选取器

选取器确定的是方法名，而不是方法实现。这是多态性和动态绑定的基础，它使得向不同类对象发送相同的消息成为现实；否则，发送消息和标准C中调用方法就没有区别，也就不可能支持多态性和动态绑定。

另外，同一个类的同名类方法和实例方法拥有相同的选取器。

(3) 方法返回值和参数类型

消息机制通过选取器找到方法的返回值类型和参数类型，因此，动态绑定（例：向id定义的对象发送消息）需要同名方法的实现拥有相同返回值类型和相同的参数类型；否则，运行时可能出现找不到对应方法的错误。

有一个例外，虽然同名类方法和实例方法拥有相同的选取器，但是它们可以有不同的参数类型和返回值类型。

3> 动态绑定

69. OC的优缺点。

答：优点：1).Cateogies 2).Posing 3).动态识别 4).指标计算 5).弹性讯息传递 6).不是一

个过度复杂的C衍生语言 7).Objective-C 与 C++ 可混合编程

缺点：1).不支持命名空间 2).不支持运算符重载 3).不支持多重继承 4).使用动态运行时类型，所有的方法都是函数调用，所以很多编译时优化方法都用不到。(如内联函数等)，性能低劣。

对于命名冲突可以使用长命名法或特殊前缀解决，如果是引入的第三方库之间的命名冲突，可以使用link命令及flag解决冲突

70. oc中可修改和不可以修改类型。

答：可修改不可修改的集合类，这个我个人简单理解就是可动态添加修改和不可动态添加修改一样。比如NSArray和NSMutableArray，前者在初始化后的内存控件就是固定不可变的，后者可以添加等，可以动态申请新的内存空间。

71. 我们说的oc是动态运行时语言是什么意思?

答：多态。主要是将数据类型的确定由编译时，推迟到了运行时。这个问题其实浅涉及到两个概念，运行时和多态。

简单来说，运行时机制使我们直到运行时才去决定一个对象的类别，以及调用该类别对象指定方法。

多态：不同对象以自己的方式响应相同的消息的能力叫做多态。

意思就是假设生物类(life)都用有一个相同的方法-eat。那人类属于生物，猪也属于生物，都继承了life后，实现各自的eat，但是调用是我们只需调用各自的eat方法。也就是不同的对象以自己的方式响应了相同的消息(响应了eat这个选择器)。因此也可以说，运行时机制是多态的基础?~~~

74.什么是谓词?

答：谓词是通过NSPredicate，是通过给定的逻辑条件作为约束条件，完成对数据的筛选。

```
predicate = [NSPredicate predicateWithFormat:@"customerID == %d",n];

a = [customers filteredArrayUsingPredicate:predicate];

predicate = [NSPredicate predicateWithFormat:@"customerID == %d",n];

a = [customers filteredArrayUsingPredicate:predicate];
```

54. 简单介绍下NSURLConnection类及

+sendSynchronousRequest:returningResponse:error: 与 - initWithRequest:delegate:两个方法的区别?

答 :NSURLConnection 主 要 用 于 网 络 访 问 , 其 中 + sendSynchronousRequest:returningResponse:error:是同步访问数据，即当前线程会阻塞，并等待request的返回的response，而 - initWithRequest:delegate:使用的是异步加载，当其完成网络访问后，会通过delegate回到主线程，并其委托的对象。

62.谈谈OC的内存管理方式及过程？

答: 1).当你使用new,alloc和copy方法创建一个对象时,该对象的保留计数器值为1.当你不再使用该对象时,你要负责向该对象发送一条release或autorelease消息.这样,该对象将在使用寿命结束时被销毁.

2).当你通过任何其他方法获得一个对象时,则假设该对象的保留计数器值为1,而且已经被设置为自动释放,你不需要执行任何操作来确保该对象被清理.如果你打算在一段时间内拥有该对象,则需要保留它并确保在操作完成时释放它.

3).如果你保留了某个对象,你需要(最终)释放或自动释放该对象.必须保持retain方法和release方法的使用次数相等.

63.OC有私有方法吗？私有变量呢？

答: objective-c - 类里面的方法只有两种, 静态方法和实例方法. 这似乎就不是完整的面向对象了,按照OO的原则就是一个对象只暴露有用的东西. 如果没有了私有方法的话, 对于一些小范围的代码重用就不那么顺手了. 在类里面声名一个私有方法

```
@interface Controller : NSObject {
    NSString *something;
}
+ (void)thisIsAStaticMethod;
- (void)thisIsAnInstanceMethod;
@end
@interface Controller (private)
- (void)thisIsAPrivateMethod;
@end
@interface Controller : NSObject {
```

```
    NSString *something;
}
+ (void)thisIsAStaticMethod;
- (void)thisIsAnInstanceMethod;
@end
@interface Controller (private)
- (void)thisIsAPrivateMethod;
@end
```

@private可以用来修饰私有变量

在Objective-C中，所有实例变量默认都是私有的，所有实例方法默认都是公有的

65.事件传递&响应者链

事件响应链。包括点击事件，画面刷新事件等。在视图栈内从上至下，或者从下之上传播。可以说点事件的分发，传递以及处理。

事件的产生和传递过程：

1.当触摸事件发生时,压力转为电信号,iOS系统将产生UIEvent对象,记录事件产生的时间和类型,然后系统将事件加入到一个由UIApplication管理的事件队列中。

2.UIApplication 会从事件队列中取出最前面的事件，并将事件分发下去以便处理，通常会先发送事件给应用程序的主窗口(keyWindow)

3.主窗口会在视图层次结构中找到一个最合适的视图来处理触摸事件

4.找到合适的视图控件后，就会调用视图控件的 touches 方法来作事件的具体处理：touchesBegin… touchesMoved…touchesEnded 等

5.这些 touches 方法默认的做法是将事件顺着响应者链条向上传递，将事件叫个上一个相应者进行处理

一般事件的传递是从父控件传递到子控件的

如果父控件接受不到触摸事件，那么子控件就不可能接收到触摸事件 UIView 不能接收触摸事件的三种情况：

1.不接受用户交互：userInteractionEnabled = NO;

2.隐藏：hidden = YES;

3.透明：alpha = 0.0~0.01

用户的触摸事件首先会由系统截获，进行包装处理等。

然后递归遍历所有的 view，进行碰触测试(hitTest)，直到找到可以处理事件的 view。

- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event; // recursively calls -pointInside:withEvent:. point is in the receiver's coordinate system

- (BOOL)pointInside:(CGPoint)point withEvent:(UIEvent *)event; // default returns YES if point is in bounds

大致的过程 application -> window -> root view ->……->lowest view

响应者链

响应者链条其实就是很多响应者对象(继承自 UIResponder 的对象)一起组合起来的链条称之为响应者链条

一般默认做法是控件将事件顺着响应者链条向上传递，将事件交给上一个响应者进行处理。那么如何判断当前响应者的上一个响应者是谁呢？有以下两个规则：

1.判断当前是否是控制器的 View，如果是控制器的 View，上一个响应者就是控制器

2.如果不是控制器的 View，上一个响应者就是父控件

当有 view 能够处理触摸事件后，开始响应事件。 系统会调用 view 的以下方法：

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;

可以多对象共同响应事件。只需要在以上方法重载中调用 super 的方法。

大致的过程 initial view -> super view -> …. -> view controller -> window -> Application

需要特别注意的一点是，传递链中时没有 controller 的，因为 controller 本身不具有大小的概念。但是响应链中是有 controller 的，因为 controller 继承自 UIResponder。

UIApplication->UIWindow->递归找到最合适处理的控件->控件调用 touches 方法->判断是 否实现 touches 方法->没有实现默认会将事件传递给上一个响应者->找到上一个响应者->找不到方法作废

PS：利用响应者链条我们可以通过调用 touches 的 super 方法，让多个响应者同时响应该事件。

66. frame和bounds有什么不同?

答:frame指的是：该view在父view坐标系统中的位置和大小。(参照点是父亲的坐标系统)

bounds指的是：该view在本身坐标系统中 的位置和大小。(参照点是本身坐标系统)

67. 方法和选择器有何不同?

答：selector是一个方法的名字，method是一个组合体，包含了名字和实现，详情可以看apple文档。

68. 什么是延迟加载?

答：懒汉模式，只在用到的时候才去初始化，也可以理解成延时加载。

我觉得最好也最简单的一个列子就是tableView中图片的加载显示了。一个延时载，避免内存过高，一个异步加载，避免线程堵塞。

69. 是否在一个视图控制器中嵌入两个tableview控制器?

答：一个视图控制只提供了一个View视图，理论上一个tableViewController也不能放吧，只能说可以嵌入一个tableview视图。当然，题目本身也有歧义，如果不是我们定性思维认为的UIViewController，而是宏观的表示视图控制者，那我们倒是可以把其看成一个视图控制者，它可以控制多个视图控制器，比如TabbarController那样的感觉。

70. 一个tableView是否可以关联两个不同的数据源?你会怎么处理?

答：首先我们从代码来看，数据源如何关联上的，其实是在数据源关联的代理方法里实现的。因此我们并不关心如何去关联他，他怎么关联上，方法只是让我返回根据自己的需要去设置如相关的数据源。

因此，我觉得可以设置多个数据源啊，但是有个问题是，你这是想干嘛呢?想让列表如何显示，不同的数据源分区块显示?

71. 什么时候使用NSMutableArray，什么时候使用NSArray?

答：当数组在程序运行时，需要不断变化的，使用NSMutableArray，当数组在初始化后，便不再改变的，使用NSArray。需要指出的是，使用NSArray只表明的是该数组在运行时不发生改变，即不能往NSAarry的数组里新增和删除元素，但不表明其数组内的元素的内容不能发生改变。NSArray是线程安全的，NSMutableArray不是线程安全的，多线程使用到NSMutableArray需要注意。

73. 在应用中可以创建多少autorelease对象，是否有限制?

答案：无

74.如果我们不创建内存池，是否有内存池提供给我们?

答:界面线程维护着自己的内存池，用户自己创建的数据线程，则需要创建该线程的内存池

75.什么时候需要在程序中创建内存池?

答：用户自己创建的数据线程，则需要创建该线程的内存池

76.类NSObject的那些方法经常被使用?

答:NSObject是Objetive-C的基类,其由NSObject类及一系列协议构成。其中类方法alloc、class、description 对象方法init、dealloc、 - performSelector:withObject:afterDelay:等经常被使用

77.什么是简便构造方法?

答：简便构造方法一般由CocoaTouch框架提供，如NSNumber的

+ numberWithBool: + numberWithChar: + numberWithDouble: + numberWithFloat:

+ numberWithInt: + numberWithBool: + numberWithChar: + numberWithDouble:

+ numberWithFloat: + numberWithInt:

Foundation下大部分类均有简便构造方法，我们可以通过简便构造方法，获得系统给我们创建好的对象，并且不需要手动释放。

78.如何使用Xcode设计通用应用?

答：使用MVC模式设计应用，其中Model层完成脱离界面，即在Model层，其是可运行在任何设备上，在controller层，根据iPhone与iPad(独有UISplitViewController)的不同特点选择不同的viewController对象。在View层，可根据现实要求，来设计，其中以xib文件设计时，其设置其为universal。

79.UIView的动画效果有那些?

UIViewAnimationOptionCurveEaseInOut

UIViewAnimationOptionCurveEaseIn

UIViewAnimationOptionCurveEaseOut

UIViewAnimationOptionTransitionFlipFromLeft

UIViewAnimationOptionTransitionFlipFromRight

UIViewAnimationOptionTransitionCurlUp

UIViewAnimationOptionTransitionCurlDown

UIViewAnimationOptionCurveEaseInOut

UIViewAnimationOptionCurveEaseIn

UIViewAnimationOptionCurveEaseOut

UIViewAnimationOptionTransitionFlipFromLeft

UIViewAnimationOptionTransitionFlipFromRight

UIViewAnimationOptionTransitionCurlUp

UIViewAnimationOptionTransitionCurlDown

81.内存管理 Autorelease、retain、copy、assign的set方法和含义？

1).你初始化(alloc/init)的对象，你需要释放(release)它。例如：

NSMutableArray aArray = [[NSArray alloc] init]; 后，需要 [aArray release];

2).你retain或copy的，你需要释放它。例如：

[aArray retain] 后，需要 [aArray release];

3).被传递(assign)的对象，你需要斟酌的retain和release。例如：

obj2 = [[obj1 someMethod] autorelease];

对象2接收对象1的一个自动释放的值，或传递一个基本数据类型(NSInteger，NSString)时：你或希望将对象2进行retain，以防止它在被使用之前就被自动释放掉。但是在retain后，一定要在适当的时候进行释放。

关于索引计数(Reference Counting)的问题

retain值 = 索引计数(Reference Counting)

NSArray对象会retain(retain值加一)任何数组中的对象。当NSArray被卸载(dealloc)的时候，所有数组中的对象会 被 执行一次释放(retain值减一)。不仅仅是NSArray，任何收集类(Collection Classes)都执行类似操作。例如 NSDictionary，甚至UINavigationController。

Alloc/init建立的对象，索引计数为1。无需将其再次retain。

[NSArray array]和[NSDate date]等“方法”建立一个索引计数为1的对象，但是也是一个自动

释放对象。所以是本地临时对象，那么无所谓了。如果是打算在全Class中使用的变量(iVar)，则必须retain它。

缺省的类方法返回值都被执行了“自动释放”方法。(*如上中的NSArray)

在类中的卸载方法“dealloc”中，release所有未被平衡的NS对象。(*所有未被autorelease，而retain值为1的)

82. C和OC如何混用

1).obj-c的编译器处理后缀为m的文件时，可以识别obj-c和c的代码，处理mm文件可以识别obj-c,c,c++代码，但cpp文件必须只能用c/c++代码，而且cpp文件include的头文件中，也不能出现obj-c的代码，因为cpp只是cpp

2).在mm文件中混用cpp直接使用即可，所以obj-c混cpp不是问题

3).在cpp中混用obj-c其实就是使用obj-c编写的模块是我们想要的。

如果模块以类实现，那么要按照cpp class的标准写类的定义，头文件中不能出现obj-c的东西，包括#import cocoa的。实现文件中，即类的实现代码中可以使用obj-c的东西，可以import,只是后缀是mm。

如果模块以函数实现，那么头文件要按c的格式声明函数，实现文件中，c++函数内部可以用obj-c，但后缀还是mm或m。

总结：只要cpp文件和cpp include的文件中不包含obj-c的东西就可以用了，cpp混用obj-c的关键是使用接口，而不能直接使用 实现代 码，实际上cpp混用的是obj-c编译后的o文件，这个东西其实是无差别的，所以可以用。obj-c的编译器支持cpp。

85.深拷贝与前拷贝区别

深拷贝同浅拷贝的区别：浅拷贝是指针拷贝，对一个对象进行浅拷贝，相当于对指向对象的指针进行复制，产生一个新的指向这个对象的指针，那么就是有两个指针指向同一个对象，这个对象销毁后两个指针都应该置空。深拷贝是对一个对象进行拷贝，相当于对对象进行复制，产生一个新的对象，那么就有两个指针分别指向两个对象。当一个对象改变或者被销毁后拷贝出来的新的对象不受影响。

实现深拷贝需要实现 NSCoying 协议，实现- (id)copyWithZone:(NSZone *)zone 方法。当对一个 property 属性含有 copy 修饰符的时候，在进行赋值操作的时候实际上就是调用这个方法。

父类实现深拷贝之后，子类只要重写 copyWithZone 方法，在方法内部调用父类的 copyWithZone 方法，之后实现自己的属性的处理

父类没有实现深拷贝，子类除了需要对自己的属性进行处理，还要对父类的属性进行处理

浅拷贝：本质上没有产生新对象

深拷贝：产生了新对象

2> 什么是深拷贝浅拷贝

对于非容器类对象,不可变对象进行copy操作为浅拷贝,引用计数器加1,其他三种为深拷贝

对于容器类对象,基本和非容器类对象一致,但注意其深拷贝是对象本身是对象复制,其中元素仍为指针复制,系统将initWithArray方法归为了元素深拷贝,但其实如果元素为不可变元素,仍为指针复制,使用归解档可以实现真正的深拷贝,元素也是对象拷贝NSArray* trueDeepCopyArray = [NSKeyedUnarchiver unarchiveObjectWithData: [NSKeyedArchiver archivedDataWithRootObject: array]];

3> 字符串什么时候使用copy,strong

属性引用的对象由两种情况,可变和不可变字符串

引用对象不可变情况下,copy和strong一样,copy为浅拷贝

引用对象可变情况下,如果希望属性跟随引用对象变化,使用strong,希望不跟随变化使用copy

4> 字符串所在内存区域

@“abc” 常量区 stringwithformat 堆区

5> mutablecopy和copy @property(copy) NSMutableArray *arr;这样写有什么问题

mutablecopy返回可变对象,copy返回不可变对象

6> 如何让自定义类可以使用copy修饰符

实现<NSCopying>协议,重写copyWithZone方法

86.id、NSObject＊、instancetype 的区别

Id 声明的对象具有运行时的特性，即可以指向任意类型的 Objcetive-C 的对象;

id 是一个 objc_object 结构体指针，定义是

typedef struct objc_object *id

id 可以理解为指向对象的指针。所有 oc 的对象 id 都可以指向，编译器不会做类型检查，id 调用任何存在的方法都不会在编译阶段报错，当然如果这个 id 指向的对象没有这个方法，该崩溃还是会崩溃的。

NSObject *指向的必须是 NSObject 的子类，调用的也只能是 NSObjec 里面的方法否则就要做强制类型转换。

不是所有的 OC 对象都是 NSObject 的子类，还有一些继承自 NSProxy。NSObject *可指向的类型是 id 的子集

instancetype 只能返回值，编译时判断真实类型，不符合发警告

86.对于语句 NSString*obj = [[NSData alloc] init]; obj 在编译时和运行时分别时什么类型的对象？

编译时是 NSString 的类型；运行时是 NSData 类型的对象

87.#import 跟#include 又什么区别，@class 呢， ＃import<> 跟#import”″又什么区别？

#import 是 Objective-C 导入头文件的关键字，#include 是 C/C++导入头文件的关键字,使用

#import 头文件会自动只导入一次，不会重复导入，相当于#include 和#pragma once；@class 告诉编译器某个类的声明，当执行时，才去查看类的实现文件，循环引用头文件；#import<> 用来包含系统的头文件，#import"" 用来包含用户头文

88. OC的类可以多重继承么?可以实现多个接口么?Category是什么?重写一个类的方法用继承好还是分类好?为什么?

答： OC的类不可以多重继承。可以实现多个接口，通过实现多个接口可以完成类似C++的多重继承。使用代理实现类是c++的多继承。Category是类别。一般情况重写一个类的方法用继承比较好，这样不会影响其他地方正常使用这个方法。

89. 写一个setter方法用于完成@property(nonatomic,retain)NSString *name,写一个setter方法用于完成@property(nonatomic，copy)NSString *name

```
- (void)setName:(NSString*)str {
    if (_name != str) {
        [_name release];
        _name = [str retain];
    }
}
- (void)setName:(NSString *)str {
    if (_name != str) {
        [_name release];
        _name = [str copy];
    }
}
- (void)setName:(NSString*)str {
    if (_name != str) {
        [_name release];
        _name = [str retain];
    }
```

```
}
- (void)setName:(NSString *)str {
    if (_name != str) {
        [_name release];
        _name = [str copy];
    }
}
```

90.常见的Objective-C的数据类型有那些， 和C的基本数据类型有什么区别?如：NSInteger和int

答：Objective-C的数据类型有NSString，NSNumber，NSArray，NSMutableArray，NSData等等，这些都是class，创建后便是对象，而C语言的基本数据类型int，只是一定字节的内存空间，用于存放数值;NSInteger是基本数据类型Int或者Long的别名(NSInteger的定义typedef long NSInteger)，NSInteger表示当前cpu下整型所占最大字节,不同CPU的long型所占字节不同,32位int4 long4,64位int4,long8

91.OC如何对内存管理的,说说你的看法和解决方法?

答：Objective-C的内存管理主要有三种方式ARC(自动内存计数)、手动内存计数、内存池。

1). (Garbage Collection)自动内存计数：这种方式和java类似，在你的程序的执行过程中。始终有一个高人在背后准确地帮你收拾垃圾，你不用考虑它什么时候开始工作，怎样工作。你只需要明白，我申请了一段内存空间，当我不再使用从而这段内存成为垃圾的时候，我就彻底的把它忘记掉，反正那个高人会帮我收拾垃圾。遗憾的是，那个高人需要消耗一定的资源，在携带设备里面，资源是紧俏商品所以iPhone不支持这个功能。所以“Garbage Collection”不是本入门指南的范围，对“Garbage Collection”内部机制感兴趣的同学可以参考一些其他的资料，不过说老实话“Garbage Collection”不大适合适初学者研究。

解决: 通过alloc - initial方式创建的, 创建后引用计数+1, 此后每retain一次引用计数+1, 那么在程序中做相应次数的release就好了.

2). (Reference Counted)手动内存计数：就是说，从一段内存被申请之后，就存在一个变量用于保存这段内存被使用的次数，我们暂时把它称为计数器，当计数器变为0的时候，那么就是释放这段内存的时候。比如说，当在程序A里面一段内存被成功申请完成之后，那么这个计数器就从0变成1(我们把这个过程叫做alloc)，然后程序B也需要使用这个内存，那么计数器就从1变成了2(我们把这个过程叫做retain)。紧接着程序A不再需要这段内存了，那么程序A就把这个计数器减1(我们把这个过程叫做release);程序B也不再需要这段内存的时候，那么也把计数器减

1(这个过程还是release)。当系统(也就是Foundation)发现这个计数器变 成员了0，那么就会调用内存回收程序把这段内存回收(我们把这个过程叫做dealloc)。顺便提一句，如果没有Foundation，那么维护计数器，释放内存等等工作需要你手工来完成。

解决:一般是由类的静态方法创建的，函数名中不会出现alloc或init字样，如[NSString string]和[NSArray arrayWithObject:]，创建后引用计数+0，在函数出栈后释放，即相当于一个栈上的局部变量. 当然也可以通过retain延长对象的生存期.

3). (NSAutoRealeasePool)内存池：可以通过创建和释放内存池控制内存申请和回收的时机.

解决:是由autorelease加入系统内存池，内存池是可以嵌套的，每个内存池都需要有一个创建释放对，就像main函数中写的一样. 使用也很简单，比如[[[NSString alloc]initialWithFormat:@”Hey you!”] autorelease]，即将一个NSString对象加入到最内层的系统内存池，当我们释放这个内存池时，其中的对象都会被释放.

92.原子(atomic)跟非原子(non-atomic)属性有什么区别?

1). atomic提供多线程安全。是防止在写未完成的时候被另外一个线程读取，造成数据错误

2). non-atomic:在自己管理内存的环境中，解析的访问器保留并自动释放返回的值，如果指定了 nonatomic ，那么访问器只是简单地返回这个值。

原子属性采用的是"多读单写"机制的多线程策略
"多读单写"缩小了锁范围,比互斥锁的性能好
规定只在主线程更新UI,就是因为如果在多线程中更新,就需要给UI对象加锁,防止资源抢占写入错误,但是这样会降低UI交互的性能,所以ios设计让所有UI对象都是非线程安全的(不加锁),并规定只在主线程中更新UI,规避多线程抢占资源问题

93.看下面的程序,第一个NSLog会输出什么?这时str的retainCount是多少?第二个和第三个呢? 为什么?

```
NSMutableArray* ary = [[NSMutableArray array] retain];

NSString *str = [NSString stringWithFormat:@"test"];

[str retain];

[aryaddObject:str];

NSLog(@”%@%d”,str,[str retainCount]);

[str retain];

[str release];
```

```
[str release];
NSLog(@”%@%d”,str,[str retainCount]);
[aryremoveAllObjects];
NSLog(@”%@%d”,str,[str retainCount]);
NSMutableArray* ary = [[NSMutableArray array] retain];
NSString *str = [NSString stringWithFormat:@"test"];
[str retain];
[aryaddObject:str];
NSLog(@”%@%d”,str,[str retainCount]);
[str retain];
[str release];
[str release];
NSLog(@”%@%d”,str,[str retainCount]);
[aryremoveAllObjects];
NSLog(@”%@%d”,str,[str retainCount]);
```

str的retainCount创建+1，retain+1，加入数组自动+1 3

retain+1，release-1，release-1 2

数组删除所有对象，所有数组内的对象自动-1 1

94. 内存管理的几条原则时什么?按照默认法则.那些关键字生成的对象需要手动释放?在和property结合的时候怎样有效的避免内存泄露?

谁申请，谁释放

遵循Cocoa Touch的使用原则;

内存管理主要要避免“过早释放”和“内存泄漏”，对于“过早释放”需要注意@property设置特性时，一定要用对特性关键字，对于“内存泄漏”，一定要申请了要负责释放，要细心。

关键字alloc 或new 生成的对象需要手动释放;

设置正确的property属性，对于retain需要在合适的地方释放，

95.如何对iOS设备进行性能测试?

答： Profile-> Instruments ->Time Profiler

96.设计模式

设计模式：MVC 模式、单例模式、观察者模式、MVVM 模式、工厂模式、代理模式、策略模式、适配器模式、模板模式、外观模式、创建模式

参考：http://blog.jobbole.com/20496/ 有 23 中设计模式

1.mvc 模式：model 保存应用模型和处理数据逻辑、view 负责 model 数据和交互控件的显示、controller 负责 model 和 View 之间的通讯

2.单例模式：用一个静态方法返回这个类的对象。这个对象是全局唯一的。整个项目里面只开辟一块内层，比如登录之后获取的用户数据存储、NSNotificationcenter、NSUserdefaults, sharedApplication。
缺点：这块内层直到项目推出时才能释放。
优势：使用简单，延时求值，易于跨模块，便于资源共享控制，方便传值和修改单例的属性
敏捷原则：单一职责原则
注意事项：确保使用者只能通过 getInstance方法才能获得，单例类的唯一实例。oc中，重写allocWithZone方法，保证即使用户用 alloc方法直接创建单例类的实例，返回的也只是此单例类的唯一静态变量。

3.观察者模式：通过添加观察者来观察某个对象的实例变量的变化、当该被观察的对象发生时，发出通知，通知观察者。如常用的导航栏渐变。应用场景：一般为 model 层对 controller 和 view 进行的通知方式，不关心谁去接收，只负责发布信息。优势：解耦合 敏捷原则：接口隔离原则，开放-封闭原则 实例：通知中心，注册通知中心，任何位置可以发送消息，注册观察者的对象可以接收。

4.工厂模式：快速创建对象的方式。将对象的创建和属性赋值封装成类方法，如：创建常用按钮、textFeild 等，forState 这些枚举值不用反复写，可以使调用工厂方法的地方代码更加简洁。
应用场景：工厂方式创建类的实例，多与proxy模式配合，创建可替换代理类。
优势：易于替换，面向抽象编程，application只与抽象工厂和易变类的共性抽象类发生调用关系。
敏捷原则：DIP依赖倒置原则
实例：项目部署环境中依赖多个不同类型的数据库时，需要使用工厂配合proxy完成易用性替换
注意事项：项目初期，软件结构和需求都没有稳定下来时，不建议使用此模式，因为其劣势也很明显，增加了代码的复杂度，增加了调用层次，增加了内存负担。所以要注意防止模式的滥用。

6. 代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。如招人干活，干完告诉我。常见的如QQ的自动回复就属于代理拦截，代理模式在iphone中得到广泛应用。有点像c++中多继承。增加对象的方法和属性。代理模式使项目的逻辑结构比较直观，比如tableView的delegate和DataSource。优势：解耦合 敏捷原则：开放-封闭原则。代理的目的是改变或传递控制链，允许一个类在某些特定时刻通知到其他类，而不需要获取到那些类的指针，可以减少框架复杂度和耦合度。另外一点，代理可以理解为java中的回调监听机制的一种类似。

7. 策略模式：把一些独立的算法单独封装起来，如我以前有个车管的 app 里面根据北斗定位步标

设备最后一次上传数据库的时间和车辆状态，来解析车辆当前的状态，数据库中的 16 进制的状态（应用的是交通部的 808 协议），移动端获得将此状态字段转换成 2 进制，判断出车辆的 24 中状态。cell 多种响应效果

应用场景：定义算法族，封装起来，使他们之间可以相互替换。优势：使算法的变化独立于使用算法的用户

敏捷原则：接口隔离原则；多用组合，少用继承；针对接口编程，而非实现。

实例：排序算法，NSArray的sortedArrayUsingSelector

8. 适配器模式：根据不同的场景选择不同的对象，不如接手了一个旧代码，一进公司就得修改需求，这时候的代码逻辑没法去反复理解，如我的一个老项目里面有一个认证功能将货主认证的 model 和车主认证 model 放同一个 model 里面，现在需要增加货主认证 model 属性修改，此时就可以使用适配器了，原来其他地方还是走货住认证，因为企业也是货主的一种，可以建一个新的货主 model 新需求走新的货主认证。

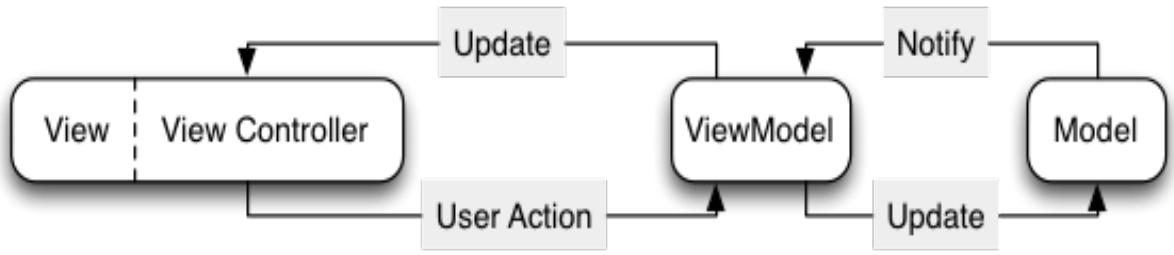
9. 模板模式：比如现在的项目建的基类 baseViewController，baseTableViewController，

10. 外观模式：专门为外部提供子类模块功能的 api 类，如果保险下单，只需要支付用你选的支付方式和保险种类及填写的保险的必要信息一起传给下单的外观对象即可，在外观类里面封装了有下单和支付两个子步簇，只需要将下保险的是否成功的结果返给下单界面就行。

11. 创建模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示，假如在一个工具对象内对轨迹点去重复、纠偏、漂移过滤等，例加一个对象对外提供该时间段轨迹停车时长和平均速度、平局耗油量等参数的接口。拿到数组点在地图上展示一下就可以了，此数组对象的生成和使用可以分开。当然了这些复杂的操作都在服务端做了。

12. MVP 模式从经典的 MVC 模式演变而来，将 Controller 替换成 Presenter，依据 MVP 百度百科中的解释，MVP 的优点相比较于 MVC 是完全分离 Model 与 View，Model 与 View 的信息传递只能通过 Controller/Presenter，我查阅资料发现在其他平台上的 MVC 模式 View 与 Model 能否直接通讯有着不同的说法，但在 iOS 开发中，Apple 是这么说的。在 MVC 下，所有的对象被归类为一个 model，一个 view，或一个 controller。Model 持有数据，View 显示与用户交互的界面，而 View Controller 调解 Model 和 View 之间的交互，在 iOS 开发中我按照 Model 与 View 无法相互通讯来理解。

13. MVVM



MVVM 模式原理分析

视图(View)、视图模型(ViewModel)、模型(Model)三部分组成，MVVM 中，我们将视图处理逻辑从 C 中剥离出来给 V，剩下的业务逻辑部分被称做 View-Model。

使用 MVVM 模式的 iOS 应用的可测试性要好于 MVC，因为 ViewModel 中并不包含对 View 的更新，相比于 MVC，减轻了 Controller 的负担，使功能划分更加合理。

使用 MVVM 模式有几大好处：

1. 低耦合。View 可以独立于 Model 变化和修改，一个 ViewModel 可以绑定到不同的 View 上，当 View 变化的时候 Model 可以不变，当 Model 变化的时候 View 也可以不变。
2. 可重用性。可以把一些视图的逻辑放在 ViewModel 里面，让很多 View 重用这段视图逻辑。
3. 独立开发。开发人员可以专注与业务逻辑和数据的开发(ViewModel)。设计人员可以专注于界面(View)的设计。
4. 可测试性。可以针对 ViewModel 来对界面(View)进行测试
5. 在 iOS 上使用 MVVM 的动机,就是让它能减少 View Controller 的复杂性并使得表示逻辑更易于测试
6. 将网络请求抽象到单独的类中
7. 将界面的拼装抽象到专门的类中
8. 构造 ViewModel　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。抽象之后，View 只接受 ViewModel，而 Controller 只需要传递 ViewModel 这么一行代码。而另外构造 ViewModel 的过程，我们就可以移动到另外的类中了。
9. MVC 设计模式中的 ViewController 进一步拆分，构造出 网络请求层、ViewModel 层、Service 层、Storage 层等其它类，来配合 Controller 工作，从而使 Controller 更加简单，我们的 App 更容易维护。 Controller 的代码抽取出来，是有助于我们做测试工作的。
10. ViewModel：存放各种业务逻辑和网络请求

在 MVC 里，View 是可以直接访问 Model 的！从而，View 里会包含 Model 信息，不可避免的还要包括一些业务逻辑。 MVC 模型关注的是 Model 的不变，所以，在 MVC 模型里，Model 不依赖于 View，但是 View 是依赖于 Model 的。不仅如此，因为有一些业务逻辑在 View 里实现了，导致要更改 View 也是比较困难的，至少那些业务逻辑是无法重用的。

MVVM 在概念上是真正将页面与数据逻辑分离的模式，它把数据绑定工作放到一个 JS 里去实现，而这个 JS 文件的主要功能是完成数据的绑定，即把 model 绑定到 UI 的元素上。

有人做过测试：使用 Angular（MVVM）代替 Backbone（MVC）来开发，代码可以减少一半。

此外，MVVM 另一个重要特性，双向绑定。它更方便你同时维护页面上都依赖于某个字段的 N 个区域，而不用手动更新它们。

总结：

优点：MVVM 就是在 MVC 的基础上加入了一个视图模型 viewModel，用于数据有效性的验证，视图的展示逻辑，网络数据请求及处理，其他的 数据处理逻辑集合，并定下相关接口和协议。相比起 MVC，MVVM 中 vc 的职责和复杂度更小，对数据处理逻辑的测试更加方便，对 bug 的原因排查更加方便，代码可阅读性，重用性和可维护性更高。MVVM 耦合性更低。MVVM 不同层级的职责更加明确，更有利于代码的编写和团队的协作。 缺点：MVVM 相比 MVC 代码量有所增加。MVVM 相比 MVC 在代码编写之前需要有更清晰的模式思路。

实例

我们应该为 app delegate 的根视图创建一个 ViewModel，当我们要生成或展示另一个次级 ViewController 时，采用当前的 ViewModel 为其创建一个子 ViewModel。

viewModel　　tableView 的布局实现，主要是计算行高。

ListViewModel 加载网络数据,缓存图片，用调度组实现。监听下载完成，异步回调。

98. 说说常用的几种传值方式

1.方法传值

2.属性传值：常用在从上一个页面向下一个页面传值，需要在下一个页面添加属性，

3.delegate 传值：需要定义协议方法，服从代理、建立代理关系实现传值，一对一的使用场景，代理是类似于 c++实现多继承的（oc 没有多继承），代理方式可以直观的看出对象的逻辑关系。例如 UITableView 的 delegate 和 DataSource，delegate 可以设置必选和可选的方法实现。用于 sender 接受到 reciever 的某个功能反馈值。

4.通知传值：可以适用于一对多的场景，界面直接不需直接的联系，缺点是不直观，并且当通知不需要时要从通知中心移除。子线程发通知可能会导致内层泄露。例如：封装在请求头里面的 Token,当 Token 过期需要弹出登录界面。任何一个界面都可能 token 过期。此时用通知很好在 App 的控制器基类 BaseViewController 里面接收通知就好。

5.单例传值：同上题 2 点

6.block传值：一般应用于需要回调的场景。使代码更紧凑，可以访存局部变量. 不需要像以前的回调一样，把在操作后所有需要用到的数据封装成特定的数据结构, 你完全可以直接访问局部变量.

7.数据存储传值，如使用 userDefault，sql 等

101. 对于单例的理解

答：在objective-c中要实现一个单例类，至少需要做以下四个步骤：

1).为单例对象实现一个静态实例，并初始化，然后设置成nil，

2).实现一个实例构造方法检查上面声明的静态实例是否为nil，如果是则新建并返回一个本类的实例，

3).重写allocWithZone方法，用来保证其他人直接使用alloc和init试图获得一个新实力的时候不产生一个新实例，

4).适当实现allocWitheZone、copyWithZone、release和autorelease。

102.从设计模式角度分析代理，通知和 KVO 区别？ios SDK 提供 的 framework 使用了哪些设计模式，为什么使用？有哪些好处和坏处?

NSNotification 是通知模式在 iOS 的实现，

KVC（key-value coding）是一个通过属性名访问属性变量的机制。

KVO 的全称是键值观察(Key-value observing),其是基于 KVC 的，

例如 将 Module 层的变化，通知到多个 Controller 对象时，可以使用 NSNotification；如果是只需要观察某个对象的某个属性，可以使用 KVO。

对于委托模式，在设计模式中是对象适配器模式，其是 delegate 是指向某个对象的，这是一对一的关系，

而在通知模式中，往往是一对多的关系。

委托模式，从技术上可以实现改变 delegate 指向的对象，但不建议这样做，会让人迷惑，如果一个 delegate 对象不断改变，指向不同的对象。

三种模式都是一个对象传递事件给另外一个对象，并且不要他们有耦合。三种模式都是对象来通知某个事件发生了的方法，或者更准确的说，是允许其他的对象收到这种事件的方法。

delegate 的优势：

1.非常严格的语法。所有将听到的事件必须是在 delegate 协议中有清晰的定义。

2.如果 delegate 中的一个方法没有实现那么就会出现编译警告/错误

3.协议必须在 controller 的作用域范围内定义

4.在一个应用中的控制流程是可跟踪的并且是可识别的；

5.在一个控制器中可以定义定义多个不同的协议，每个协议有不同的 delegates

6.没有第三方对象要求保持/监视通信过程。

7.能够接收调用的协议方法的返回值。这意味着 delegate 能够提供反馈信息给 controller

缺点：

1.需要定义很多代码：1.协议定义；2.controller 的 delegate 属性；3.在 delegate 本身中实现 delegate 方法定义

2.在释放代理对象时，需要小心的将 delegate 改为 nil。一旦设定失败，那么调用释放对象的方法将会出现内存 crash

3.在一个 controller 中有多个 delegate 对象，并且 delegate 是遵守同一个协议，但还是很难告诉多个对象同一个事件，不过有可能。

它是一个单例对象，允许当事件发生时通知一些对象。它允许我们在低程度耦合的情况下，满足控制器与一个任意的对象进行通信的目的。这种模式的基本特征是为了让其他的对象能够接收到在该 controller 中发生某种事件而产生的消息，controller 用一个 key（通知名称）。这样对于 controller 来说是匿名的，其他的使用同样的 key 来注册了该通知的对象（即观察者）能够对通知的事件作出反应。

优势：

1.不需要编写多少代码，实现比较简单；

2.对于一个发出的通知，多个对象能够做出反应，即 1 对多的方式实现简单

3.controller 能够传递 context 对象（dictionary），context 对象携带了关于发送通知的自定义的信息

缺点：

1.在编译期不会检查通知是否能够被观察者正确的处理；

2.在释放注册的对象时，需要在通知中心取消注册；

3.在调试的时候应用的工作以及控制过程难跟踪；

4.需要第三方对喜爱那个来管理 controller 与观察者对象之间的联系；

5.controller 和观察者需要提前知道通知名称、UserInfo dictionary keys。如果这些没有在工作区间定义，那么会出现不同步的情况；

6.通知发出后，controller 不能从观察者获得任何的反馈信息。

KVO 是一个对象能够观察另外一个对象的属性的值，并且能够发现值的变化。前面两种模式更加适合一个 controller 与任何其他的对象进行通信，而 KVO 更加适合任何类型的对象侦听另外一个任意对象的改变（这里也可以是 controller，但一般不是 controller）。这是一个对象与另外一个对象保持同步的一种方法，即当另外一种对象的状态发生改变时，观察对象马上作出反应。它只能用来对属性作出反应，而不会用来对方法或者动作作出反应。

优点：

1.能够提供一种简单的方法实现两个对象间的同步。例如：model 和 view 之间同步；

2.能够对非我们创建的对象，即内部对象的状态改变作出响应，而且不需要改变内部对象（SKD 对象）的实现；

3.能够提供观察的属性的最新值以及先前值；

4.用 key paths 来观察属性，因此也可以观察嵌套对象；

5.完成了对观察对象的抽象，因为不需要额外的代码来允许观察值能够被观察

缺点：

1.我们观察的属性必须使用 strings 来定义。因此在编译器不会出现警告以及检查；

2.对属性重构将导致我们的观察代码不再可用；

3.复杂的“IF”语句要求对象正在观察多个值。这是因为所有的观察代码通过一个方法来指向；

103.KVO，NSNotification，delegate 及 block 区别

KVO 就是 cocoa 框架实现的观察者模式，一般同 KVC 搭配使用，通过 KVO 可以监测一个值的变化，比如 View 的高度变化。是一对多的关系，一个值的变化会通知所有的观察者。

NSNotification 是通知，也是一对多的使用场景。在某些情况下，KVO 和 NSNotification 是一样的，都是状态变化之后告知对方。NSNotification 的特点，就是需要被观察者先主动发出通知，然后观察者注册监听后再来进行响应，比 KVO 多了发送通知的一步，但是其优点是监听不局限于属性的变化，还可以对多种多样的状态变化进行监听，监听范围广，使用也更灵活。

delegate 是代理，就是我不想做的事情交给别人做。比如狗需要吃饭，就通过 delegate 通知主人，主人就会给他做饭、盛饭、倒水，这些操作，这些狗都不需要关心，只需要调用 delegate（代理人）就可以了，由其他类完成所需要的操作。所以 delegate 是一对一关系。

block 是 delegate 的另一种形式，是函数式编程的一种形式。使用场景跟 delegate 一样，相比 delegate 更灵活，而且代理的实现更直观。

KVO 一般的使用场景是数据，需求是数据变化，比如股票价格变化，我们一般使用 KVO（观察者模式）。delegate 一般的使用场景是行为，需求是需要别人帮我做一件事情，比如买卖股票，我们一般使用 delegate。

Notification 一般是进行全局通知，比如利好消息一出，通知大家去买入。delegate 是强关联，就是委托和代理双方互相知道，你委托别人买股票你就需要知道经纪人，经纪人也不要知道自己的顾客。Notification 是弱关联，利好消息发出，你不需要知道是谁发的也可以做出相应的反应，同理发消息的人也不需要知道接收的人也可以正常发出消息

104.runtime/消息转发机制

1.runtime　http://www.cocoachina.com/ios/20150715/12540.html

1> 什么是runtime

Runtime运行时机制，最主要的是消息机制，是一套比较底层的纯C语言API，属于1个C语言库，包含了很多底层的C语言API。(引入<objc/runtime.h>或者<objc/message.h>)

在我们平时编写的OC代码中，程序运行过程时，其实最终都是转成了runtime的C语言代码，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能根据函数的名称找到对应的函数来调用。runtime算是OC的幕后工作者,objc_msgSend

2> runtime干什么用,使用场景

1.runtime是属于OC的底层，可以进行一些非常底层的操作(用OC是无法现实的，不好实现)

2.动态创建一个类(比如KVO的底层实现) objc_allocateClassPair， class_addIvar，objc_registerClassPair 例如：热创建

在程序运行过程中，动态地为某个类添加属性/方法，修改属性值/方法(修改封装的框架) objc_setAssociatedObject　object_setIvar　例如：热更新

3.遍历一个类的所有成员变量(属性)\所有方法(字典转模型,归解档)　class_copyIvarList　class_copyPropertyList　class_copyMethodList　如YYmodel、MJextension、JsonModel

4、查找对象 实现万能跳转跳转，例如收到推送的通知跳转到对应的页面

2.消息机制

1> 消息转发的原理

当向一个对象发送消息时，objc_msgSend方法根据对象的isa指针找到对象的类，然后在类的调度表（dispatch table）中查找selector。如果无法找到selector，objc_msgSend通过指向父类的指针找到父类，并在父类的调度表（dispatch table）中查找selector，以此类推直到NSObject类。一旦查找到selector，objc_msgSend方法根据调度表的内存地址调用该实现。 通过这种方式，message与方法的真正实现在执行阶段才绑定。

为了保证消息发送与执行的效率，系统会将全部selector和使用过的方法的内存地址缓存起来。每个类都有一个独立的缓存，缓存包含有当前类自己的 selector以及继承自父类的selector。查找调度表（dispatch table）前，消息发送系统首先检查receiver对象的缓存。 缓存命中的情况下，消息发送（messaging）比直接调用方法（function call）只慢一点点点点。

2> SEL isa super cmd 是什么

sel: 一种类型,表示方法名称,类似字符串(可互转)

isa:在方法底层对应的objc_msgSend调用时,会根据isa找到对象所在的类对象,类对象中包含了调度表(dispatch table),该表将类的sel和方法的实际内存地址关联起来

super_class:每一个类中还包含了一个super_class指针,用来指向父类对象

_cmd在Objective-C的方法中表示当前方法的selector，正如同self表示当前方法调用的对象实例

IMP定义为 id (*IMP) (id, SEL, …)。这样说来， IMP是一个指向函数的指针，这个被指向的函数包括id(“self”指针)，调用的SEL（方法名），再加上一些其他参数.说白了IMP就是实现

方法

3> 动态绑定

—在运行时确定要调用的方法

动态绑定将调用方法的确定也推迟到运行时。在编译时，方法的 调用并不和代码绑定在一起，只有在消实发送出来之后，才确定被调用的代码。通过动态类型和动态绑定技术，您的代码每次执行都可以得到不同的结果。运行时因 子负责确定消息的接收者和被调用的方法。运行时的消息分发机制为动态绑定提供支持。当您向一个动态类型确定了的对象发送消息时，运行环境系统会通过接收者 的 isa 指针定位对象的类，并以此为起点确定被调用的方法，方法和消息是动态绑定的。而且，您不必在 Objective-C 代码中做任何工作，就可以自动获取动态绑定的好处。您在每次发送消息时，特别是当消息的接收者是动态类型已经确定的对象时，动态绑定就会例行而透明地发生

106、使用 bugly 进行崩溃分析

1.测试的时候我们通常通过打印日志、断点、崩溃信息等定位 bug。（null、数组取了 null,未实现的方法）

2.打包后的崩溃，特别是不是一直出现的 bug，这些可以通过友盟，将友盟的崩溃信息列表下载到本地，使用终端解析，太麻烦了

3.集成 bugly 可以快速查看 崩溃信息，需要将 DYSM 映射表传到 bugly 网站，

buly 可以定定义日志 可以用来记录重要事件的崩溃 环境，bugly 可以自定义异常如某个接口不能出现空值。

107. jenkens 持续打包，

普通的打包方式，如使用 xcode 打包 传蒲公英 再将下载二维码 发给相关测试人员比较耗时，使用 jenkens 是一个 shell 语言开发的脚本工具，经这几个步族连到一起 使打包更节约时间。

107.KVO&KVC

底层实现：

KVC 运用了一个 isa-swizzling 技术。isa-swizzling 就是类型混合指针机制。KVC 主要通过 isa- swizzling，来实现其内部查找定位的。isa 指针，如其名称所指，（就是 is a kind of 的意思），指向维护分发表的对象的类。该分发表实际上包含了指向实现类中的方法的指针，和其它数据。

当观察者为一个对象的属性进行了注册，被观察对象的 isa 指针被修改的时候，isa 指针就会指向一个中间类，而不是真实的类。所以 isa 指 针其实不需要指向实例对象真实的类。所以我们的程序最好不要依赖于 isa 指针。在调用类的方法的时候，最好要明确对象实例的类名。

KVO 概述

KVO,即：Key-Value Observing，它提供一种机制，当指定的对象的属性被修改后，则对象就会接受到通知。简单的说就是每次指定的被观察的对象的属性被修改后，KVO 就会自动通知相应的观察者了。

使用方法

系统框架已经支持 KVO，所以程序员在使用的时候非常简单。

1：注册，指定被观察者的属性，

2：实现回调方法

3：移除观察

KVC 概述

KVC 是 KeyValueCoding 的简称，它是一种可以直接通过字符串的名字(key)来访问类属性(实例变量)的机制。而不是通过调用 Setter、Getter 方法访问。

当使用 KVO、Core Data、CocoaBindings、AppleScript(Mac 支持)时，KVC 是关键技术。

使用方法

关键方法定义在：NSKeyValueCodingprotocol

KVC 支持类对象和内建基本数据类型。

获取值

valueForKey:，传入 NSString 属性的名字。

valueForKeyPath:，传入 NSString 属性的路径，xx.xx 形式。

valueForUndefinedKey 它的默认实现是抛出异常，可以重写这个函数做错误处理。

修改值

setValue:forKey:

setValue:forKeyPath:

setValue:forUndefinedKey:

setNilValueForKey:当对非类对象属性设置 nil 时，调用，默认抛出异 常。

一对多关系成员的情况

mutableArrayValueForKey：有序一对多关系成员 NSArray

mutableSetValueForKey：无序一对多关系成员 NSSet

补充：KVO 与 Notification 之间的区别：

notification 是需要一个发送 notification 的对象，一般是 notificationCenter，来通知观察者。

KVO 是直接通知到观察对象，并且逻辑非常清晰，实现步骤简单。

19. 什么是KVO和KVC?

答：KVC：键值编码是一种间接访问对象的属性使用字符串来标识属性，而不是通过调用存取方法，直接或通过实例变量访问的机制。

KVO：键值观察机制，他提供了观察某一属性变化的方法，极大的简化了代码。

比如我自定义的一个button

```
[self addObserver:self forKeyPath:@"highlighted" options:0 context:nil];

#pragma mark KVO

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context {

    if ([keyPath isEqualToString:@"highlighted"] ) {

        [self setNeedsDisplay];

    }

}
```

```
[self addObserver:self forKeyPath:@"highlighted" options:0 context:nil];

#pragma mark KVO

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context {

    if ([keyPath isEqualToString:@"highlighted"] ) {

        [self setNeedsDisplay];

    }

}
```

对于系统是根据keyPath去取的到相应的值发生改变，理论上来说是和kvc机制的道理是一样的。

对于kvc机制如何通过key寻找到value：

“当通过KVC调用对象时，比如：[self valueForKey:@”someKey”]时，程序会自动试图通过几种不同的方式解析这个调用。首先查找对象是否带有 someKey 这个方法，如果没找到，会继续查找对象是否带有someKey这个实例变量(iVar)，如果还没有找到，程序会继续试图调用 -(id) valueForUndefinedKey:这个方法。如果这个方法还是没有被实现的话，程序会抛出一个NSUndefinedKeyException异常错误。

(注意：Key-Value Coding查找方法的时候，不仅仅会查找someKey这个方法，还会查找getsomeKey这个方法，前面加一个get，或者_someKey以及_getsomeKey这几种形式。同时，查找实例变量的时候也会不仅仅查找someKey这个变量，也会查找_someKey这个变量是否存在。)

设计valueForUndefinedKey:方法的主要目的是当你使用-(id)valueForKey方法从对象中请求值时，对象能够在错误发生前，有最后的机会响应这个请求。

3.KVC和KVO

1> 如何调用私有变量　　如何修改系统的只读属性　　KVC的查找顺序

KVC在某种程度上提供了访问器的替代方案。不过访问器方法是一个很好的东西，以至于只要是有可能，KVC也尽量再访问器方法的帮助下工作。为了设置或者返回对象属性，KVC按顺序使用如下技术：

①检查是否存在-<key>、-is<key>（只针对布尔值有效）或者-get<key>的访问器方法，如果有可能，就是用这些方法返回值；

检查是否存在名为-set<key>:的方法，并使用它做设置值。对于 -get<key>和 -set<key>:方法，将大写Key字符串的第一个字母，并与Cocoa的方法命名保持一致；

②如果上述方法不可用，则检查名为-_<key>、-_is<key>（只针对布尔值有效）、-_get<key>和-_set<key>:方法；

③如果没有找到访问器方法，可以尝试直接访问实例变量。实例变量可以是名为：<key>或_<key>;

④如果仍为找到，则调用valueForUndefinedKey:和setValue:forUndefinedKey:方法。这些方法的默认实现都是抛出异常，我们可以根据需要重写它们。

2> 什么是键-值,键路径是什么

模型的性质是通过一个简单的键（通常是个字符串）来指定的。视图和控制器通过键来查找相应的属性值。在一个给定的实体中，同一个属性的所有值具有相同的数据类型。键-值编码技术用于进行这样的查找—它是一种间接访问对象属性的机制。

键路径是一个由用点作分隔符的键组成的字符串，用于指定一个连接在一起的对象性质序列。第一个键的性质是由先前的性质决定的，接下来每个键的值也是相对于其前面的性质。键路径使您可以以独立于模型实现的方式指定相关对象的性质。通过键路径，您可以指定对象图中的一

个任意深度的路径，使其指向相关对象的特定属性。

4> kvo的实现机制

当某个类的对象第一次被观察时，系统就会在运行时动态地创建该类的一个派生类，在这个派

生类中重写原类中被观察属性的setter方法,派生类在被重写的setter方法实现真正的通知机制(Person->NSKVONotifying_Person).
派生类重写了 class 方法以“欺骗”外部调用者它就是起初的那个类。然后系统将这个对象的isa指针指向这个新诞生的派生类，因此这个对象就成为该派生类的对象了，因而在该对象上对setter的调用就会调用重写的setter，从而激活键值通知机制。此外，派生类还重写了dealloc方法来释放资源。

5> KVO计算属性　设置依赖键

监听的某个属性可能会依赖于其它多个属性的变化(类似于swift，可以称之为计算属性)，不管所依赖的哪个属性发生了变化，都会导致计算属性的变化,此时该属性如果不能通过set方法来监听(如get中进行计算

```
- (NSString *)accountForBank {

    return [NSString stringWithFormat:@“%@ for %@", self.accountName, self.bankCodeEn];
}
```

),则可以设置依赖键,两种方法:

1>

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key {

    NSSet *keyPaths = [super keyPathsForValuesAffectingValueForKey:key];

    if ([key isEqualToString:@"accountForBank"]) {

        keyPaths = [keyPaths setByAddingObjectsFromArray:@[@"accountName", @"bankCodeEn"]];
    }

    return keyPaths;
}
```

2>

```
+ (NSSet *)keyPathsForValuesAffectingAccountForBank {

    return [NSSet setWithObjects:@"accountBalance", @"bankCodeEn", nil];
}
```

6> KVO集合属性

可对可变集合的元素改变进行监听(如添加、删除和替换元素),使用集合监听对象

5> kvo使用场景

①实现上下拉刷新控件 contentoffset
②webview混合排版 contentsize
③监听模型属性实时更新UI

108、SDWebImage(SDWebImage 的实现机制)

主要功能:

提供 UIImageView 的一个分类，以支持网络图片的加载与缓存管理
一个异步的图片加载器
一个异步的内存+磁盘图片缓存
支持 GIF 图片
支持 WebP 图片
后台图片解压缩处理
确保同一个 URL 的图片不被下载多次
确保虚假的 URL 不会被反复加载
确保下载及缓存时，主线程不被阻塞
SDWebImage 下载的核心其实就是利用 NSURLConnection 对象来加载数据。每个图片的下载都由一个 Operation 操作来完成，并将这些操作放到一个操作队列中。这样可以实现图片的并发下载。

缓存

为了减少网络流量的消耗，我们都希望下载下来的图片缓存到本地，下次再去获取同一张图片时，可以直接从本地获取，而不再从远程服务器获取。这样做的另一个好处是提升了用户体验，用户第二次查看同一幅图片时，能快速从本地获取图片直接呈现给用户。SDWebImage 提供了对图片缓存的支持，而该功能是由 SDImageCache 类来完成的。该类负责处理内存缓存及一个可选的磁盘缓存。其中磁盘缓存的写操作是异步的，这样就不会对 UI 操作造成影响。

内存缓存与磁盘缓存

内存缓存的处理是使用 NSCache 对象来实现的。NSCache 是一个类似于集合的容器。它存储 key-value 对，这一点类似于 NSDictionary 类。我们通常用使用缓存来临时存储短时间使用但创建昂贵的对象。重用这些对象可以优化性能，因为它们的值不需要重新计算。另外一方面，这些对象对于程序来说不是紧要的，在内存紧张时会被丢弃。

磁盘缓存的处理则是使用 NSFileManager 对象来实现的。图片存储的位置是位于 Cache 文件夹。另外，SDImageCache 还定义了一个串行队列，来异步存储图片。

SDImageCache 提供了大量方法来缓存、获取、移除及清空图片。而对于每个图片，为了方便地在内存或磁盘中对它进行这些操作，我们需要一个 key 值来索引它。在内存中，我们将其作为 NSCache 的 key 值，而在磁盘中，我们用这个 key 作为图片的文件名。对于一个远程服务器下载的图片，其 url 是作为这个 key 的最佳选择了。我们在后面会看到这个 key 值的重要性。

SDWebImage 的主要任务就是图片的下载和缓存。为了支持这些操作，

它主要使用了以下知识点：

dispatch_barrier_sync 函数：该方法用于对操作设置屏幕，确保在执行完任务后才会执行后续操作。该方法常用于确保类的线程安全性操作。
NSMutableURLRequest：用于创建一个网络请求对象，我们可以根据需要来配置请求报头等信息。
NSOperation 及 NSOperationQueue：操作队列是 Objective-C 中一种高级的并发处理方法，现在它是基于 GCD 来实现的。相对于 GCD 来说，操作队列的优点是可以取消在任务处理队列中的任务，另外在管理操作间的依赖关系方面也容易一些。对 SDWebImage 中我们就看到了如何使用依赖将下载顺序设置成后进先出的顺序。

NSURLConnection：用于网络请求及响应处理。在 iOS7.0 后，苹果推出了一套新的网络请求接口，即 NSURLSession 类。
开启一个后台任务。

NSCache 类：一个类似于集合的容器。它存储 key-value 对，这一点类似于 NSDictionary 类。我们通常用使用缓存来临时存储短时间使用但创建昂贵的对象。重用这些对象可以优化性能，因为它们的值不需要重新计算。另外一方面，这些对象对于程序来说不是紧要的，在内存紧张时会被丢弃。
清理缓存图片的策略：特别是最大缓存空间大小的设置。如果所有缓存文件的总大小超过这一大小，则会按照文件最后修改时间的逆序，以每次一半的递归来移除那些过早的文件，直到缓存的实际大小小于我们设置的最大使用空间。
对图片的解压缩操作：这一操作可以查看 SDWebImageDecoder.m 中 +decodedImageWithImage 方法的实现。
对 GIF 图片的处理
对 WebP 图片的处理

图片下载-->显示:

异步下载: NSOperation + 操作队列

业务逻辑: 图片缓存(防止图片错位,提升效率) + 操作缓存(防止重复创建操作) + 沙盒缓存(磁盘缓存)

业务拆分: 不同的功能写在不同的类中.高层次的封装.

要求:一句话自动实现异步图片的下载并且显示图片. 沙盒缓存(磁盘缓存)

一句话代码的接口: UIImageView 的一个分类.这个分类专门负责提供一句话代码的接口.
异步图片下载: 自定义的操作.专门负责下载图片

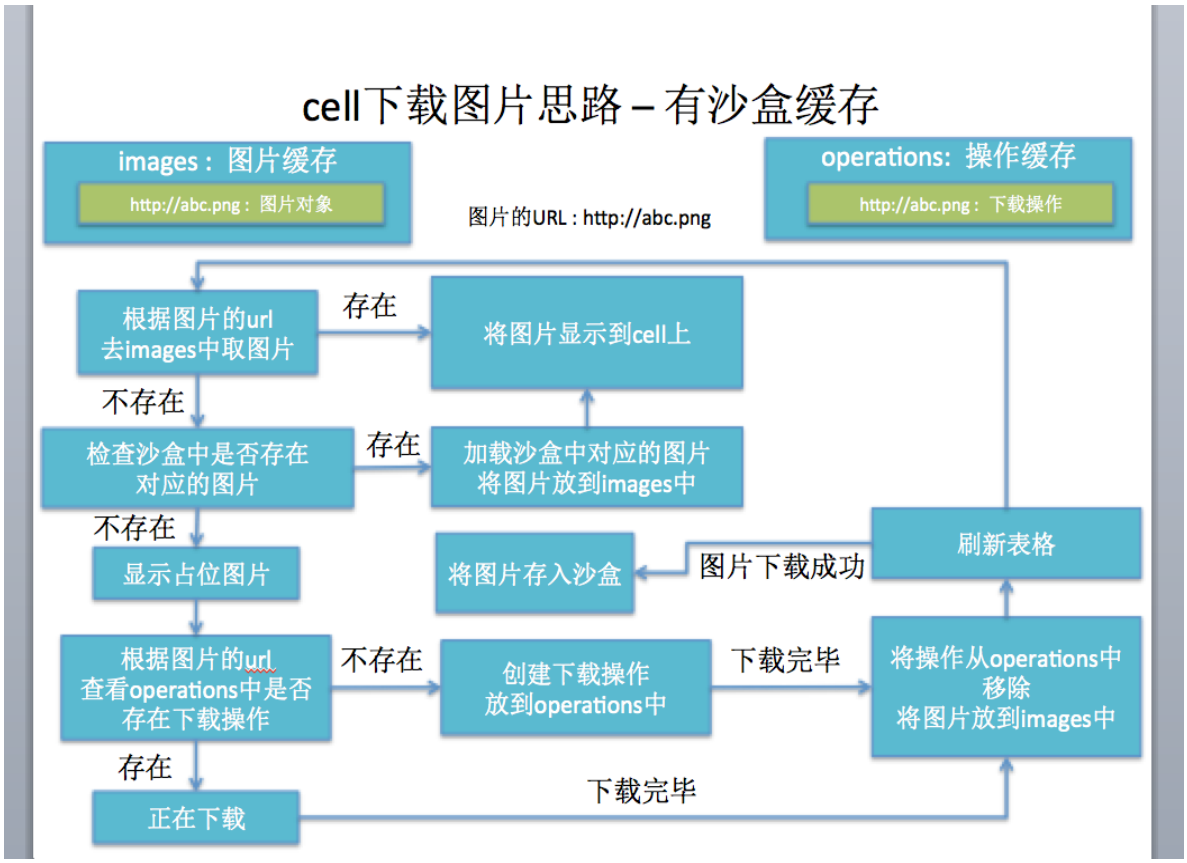
显示图片: 管理工具类.负责将图片下载和图片显示联系起来(业务功能逻辑由它实现(协调)). 图片缓存+操作缓存
沙盒缓存: 自定义的沙盒缓存工具类.专门负责沙盒中图片的读和写.

// 解决图片错位问题,需要判定 cell 对应的图片地址已经改变!

给每一个 imageView 都绑定一个下载地址. 如果外界传入的下载地址改变, 让 iamgeView 绑定的地址变成新的地址,原来的下载操作取消.开始新的下载操作.
// 如何给 imageView 绑定下载地址:利用运行时,在分类中动态的为 imageView 添加一个属性(urlString).

框架 SDWebimage 的缓存机制

1 UIImageView+WebCache: setImageWithURL:placeholderImage:options: 先显示 placeholderImage ，同时由SDWebImageManager 根据 URL 来在本地查找图片。
2 SDWebImageManager: downloadWithURL:delegate:options:userInfo: SDWebImageManager 是将UIImageView+WebCache同SDImageCache链接起来的类， SDImageCache：queryDiskCacheForKey:delegate:userInfo:用来从缓存根据CacheKey查找图片是否已经在缓存中
3 如果内存中已经有图片缓存， SDWebImageManager会回调SDImageCacheDelegate : imageCache:didFindImage:forKey:userInfo:
4 而 UIImageView+WebCache 则回调SDWebImageManagerDelegate: webImageManager:didFinishWithImage:来显示图片。
5 如果内存中没有图片缓存，那么生成 NSInvocationOperation 添加到队列，从硬盘查找图片是否已被下载缓存。
6 根据 URLKey 在硬盘缓存目录下尝试读取图片文件。这一步是在 NSOperation 进行的操作，所以回主线程进行结果回调
7 notifyDelegate:
8 如果上一操作从硬盘读取到了图片，将图片添加到内存缓存中（如果空闲内存过小，会先清空内存缓存）。SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo:进而回调展示图片。
9 如果从硬盘缓存目录读取不到图片，说明所有缓存都不存在该图片，需要下载图片，回调
10imageCache:didNotFindImageForKey:userInfo:
11共享或重新生成一个下载器 SDWebImageDownloader 开始下载图片。
12图片下载由 NSURLConnection 来做，实现相关 delegate 来判断图片下载中、下载完成和下载失败。
13connection:didReceiveData: 中利用 ImageIO 做了按图片下载进度加载效果。
14connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。
15图片解码处理在一个 NSOperationQueue 完成，不会拖慢主线程 UI。如果有需要对下载的图片进行二次处理，最好也在这里完成，效率会好很多。
16在主线程 notifyDelegateOnMainThreadWithInfo: 宣告解码完成，imageDecoder:didFinishDecodingImage:userInfo: 回调给 SDWebImageDownloader。
17imageDownloader:didFinishWithImage: 回调给 SDWebImageManager 告知图片下载完成。
18通知所有的 downloadDelegates 下载完成，回调给需要的地方展示图片。
19 将图片保存到 SDImageCache 中，内存缓存和硬盘缓存同时保存。
20 写文件到硬盘在单独 NSInvocationOperation 中完成，避免拖慢主线程。
21 如果是在iOS上运行，SDImageCache 在初始化的时候会注册notification 到 UIApplicationDidReceiveMemoryWarningNotification 以及



UIApplicationWillTerminateNotification,在内存警告的时候清理内存图片缓存，应用结束的时候清理过期图片。

22SDWebImagePrefetcher 可以预先下载图片，方便后续使用。

109．网络安全

问题：用户密码不能以明文的形式保存,需要对用户密码加密之后再保存!

密码的安全原则:

1> 本地和服务器都不允许保存用户的密码明文.

2> 在网络上,不允许传输用户的密码明文.

3> <4> 数据加密算法:

1>对称加密：

1.加密、解密使用相同的密钥和算法。

2.最快速简单的加密方式。

3.算法公开。

4.通常使用小于 256bit 的密钥。密钥越大安全性越强，但加密和解密的过程越慢。适合大数据加密。常用的有 AES、DES、IDEA。

2> 非对称加密算法:

1.加密、解密使用相同的密钥和算法。

2.最快速简单的加密方式。

3.算法公开。

4.通常使用小于 256bit 的密钥。密钥越大安全性越强，但加密和解密

加密综合方案：解决的办法是将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。

openssl :是一个强大的安全套接字层密码库,囊括主要的密码算法,常用的密钥和证书封装管理功能以及 SSL (Secure socket Laye)协议.提供丰富的应用程序测试功能

终端命令:

```
echo hello |openssl md5

echo hello |openssl sha1

echo hello |openssl sha -sha256

echo hello |openssl sha -sha512
```

}

/*---------------------------- 04 信息安全加 -----------------------------*/

了解:常用加密方法: 1> base64 2> MD5 3> MD5 加盐 4> HMAC 5> 时间戳密码(用户密码动态变化)

{

1> base64

{

base64 编码是现代密码学的基础

原本是 8 个 bit 一组表示数据,改为 6 个 bit 一组表示数据,不足的部分补零,每 两个 0 用 一个 = 表示.

用 base64 编码之后,数据长度会变大,增加了大约 1/3 左右.

base64 基本能够达到安全要求,但是,base64 能够逆运算,非常不安全!

base64 编码有个非常显著的特点,末尾有个 '=' 号.

利用终端命令进行 base64 运算:

```
// 将文件 meinv.jpg 进行 base64 运算之后存储为 meinv.txt

base64 meinv.jpg -o meinv.txt
```

```
// 讲 meinv.txt 解码生成 meinv.png
base64 -D meinv.txt -o meinv.png
// 将字符串 "hello" 进行 base 64 编码 结果:aGVsbG8=
echo "hello" | base64
// 将 base64 编码之后的结果 aGVsbG8= 反编码为字符串
echo aGVsbG8= | base64 -D
```

}

2> MD5 -- (信息-摘要算法) 哈希算法之一.

{

把一个任意长度的字节串变换成一定长度的十六进制的大整数. 注意,字符串的转换过程是不可逆的.

用于确保'信息传输'完整一致.

MD5 特点:

*1.压缩性: 任意长度的数据,算出的 MD5 值长度都是固定的.

*2.容易计算: 从原数据计算出 MD5 值很容易.

*3.抗修改性: 对原数据进行任何改动,哪怕只修改一个字节,所得到的 MD5 值都有很大区别.

*4.弱抗碰撞: 已知原数据和其 MD5 值,想找到一个具有相同 MD5 值的数据(即伪造数据)是非常困难的.

*5.强抗碰撞: 想找到两个不同数据,使他们具有相同的 MD5 值,是非常困难的.

MD5 应用:

*1. 一致性验证: MD5 将整个文件当做一个大文本信息,通过不可逆的字符串变换算法,产生一个唯一的 MD5 信息摘要.就像每个人都有自己独一无二的指纹,MD5 对任何文件产生一个独一无二的"数字指纹".

利用 MD5 来进行文件校验, 被大量应用在软件下载站,论坛数据库,系统文件安全等方面.

*2. 数字签名;

*3. 安全访问认证

}

3> MD5 加盐

{

MD5 本身是不可逆运算,但是,目前网络上有很多数据库支持反查询.

MD5 加盐 就是在密码哈希过程中添加的额外的随机值.

注意:加盐要足够长,足够复杂.

}

4> HMAC(Message Authentication Code, 消息认证码算法)

{

HMAC 利用哈希算法,以一个密钥和一个消息为输入,生成一个消息摘要作为输出.

HMAC 主要使用在身份认证中;

认证流程:

*1. 客户端向服务器发送一个请求.

*2. 服务器接收到请求后,生成一个'随机数'并通过网络传输给客户端.

*3. 客户端将接收到的'随机数'和'密钥'进行 HMAC-MD5 运算,将得到的结构作为认证数据传递给服务器.

(实际是将随机数提供给 ePass,密钥也是存储在 ePass 中的)

*4. 与此同时,服务器也使用该'随机数'与存储在服务器数据库中的该客户'密钥'进行 HMAC-MD5 运算,如果

服务器的运算结果与客户端传回的认证数据相同,则认为客户端是一个合法用法.

}

5> 时间戳密码(用户密码动态变化)

{

相同的密码明文 + 相同的加密算法 ===》 每次计算都得出不同的结果.可以充分保证密码的安全性.

原理:将当前时间加入到密码中;

因为每次登陆时间都不同,所以每次计算出的结果也都不相同.

服务器也需要采用相同的算法.这就需要服务器和客户端时间一致.

注意:服务器端时间和客户端时间,可以有一分钟的误差(比如:第 59S 发送的网络请求,一秒钟后服务器收到并作出响应,这时服务器当前时间比客户端发送时间晚一分钟)

这就意味着,服务器需要计算两次(当前时间和一分钟之前两个时间点各计算一次).只要有一个结果是正确的,就可以验证成功

```
    }
    // IP 辅助/手机绑定..
}
/*---------------------------------------- 05 钥匙串访问
----------------------------------------*/
```

重点: 1.钥匙串访问

{

苹果在 iOS 7.0.3 版本以后公布钥匙串访问的 SDK. 钥匙串访问接口是纯 C 语言的.

钥匙串使用 AES 256 加密算法,能够保证用户密码的安全.

钥匙串访问的第三方框架(SSKeychain),是对 C 语言框架 的封装.注意:不需要看源码.

钥匙串访问的密码保存在哪里?只有苹果才知道.这样进一步保障了用户的密码安全.

使用步骤:

```
    {
        // 获取应用程序唯一标识.
        NSString *bundleId = [NSBundle mainBundle].bundleIdentifier
        // 1.利用第三方框架,将用户密码保存在钥匙串
        [SSKeychain setPassword:self.pwdText.text forService:bundleId
account:self.usernameText.text];
```

"注意"三个参数:

1.密码:可以直接使用明文.钥匙串访问本身是使用 AES 256 加密,就是安全的.所以使用的时候,直接传递密码明文就可以了.

2.服务名:可以随便乱写,建议唯一! 建议使用 bundleId

bundleId 是应用程序的唯一标识,每一个上架的应用程序都有一个唯一的 bundleId

3.账户名:直接用用户名称就可以.

```
        // 2.从钥匙串加载密码
        self.pwdText.text = [SSKeychain passwordForService:bundleId
account:self.usernameText.text];
    }
}
```

HMAC_SHA1 是一种安全的基于加密 hash 函数和共享密钥的消息认证协议。它可以有效地防止数据在传输过程中被截获和篡改，维护数据的完整性、可靠性和安全性。

HMAC_SHA1 消息认证机制的成功在于一个加密的 hash 函数、一个加密的随机密钥和一个安全的密钥交换机制。

1. 在原始 URL 里加入一个名称为 e 的时间戳参数，避免缓存而得到旧的数据；
2. 分解请求的 URL，从中取出 path 部分和 query_string 部分；
3. 将 query_string 中的参数按名称排序，然后按顺序用 = 和 & 连接成新的 query_string ；
4 用字符串拼接的方式组装 path 和 query_string，两者之间以 ？ 连接成新的 URL；
5. 用 HMAC_SHA1 算法生成摘要 digest ，其中第一个参数 SECRET_KEY 为私钥，第二个参数是已拼接的字符串 URL；
6. 对 digest 进行 base64 编码；
7 对 base64 编码后的 digest 进行 URL 安全处理，即将其中的 / 替换为 _，将 + 替换为 -；
8 用 ACCESS_KEY 明文与编码后的 digest 进行拼接，中间使用冒号 ： 连接，得到最终的的 access_token；

(1)在原始 URL 里面加入时间戳参数，请求 url,取出 path 和 query_string
(2)将 query_string 中的参数按名称排序，然后连接成新的 query_string
(3)拼接 path 和 query_string，生成新的 url
（4）用 HMAC_SHA1 算法生成摘要
（5）对摘要进行 base64
（6）对 base64 后的 url 进行安全处理
（7）用 ACCESS_KEY 明文与编码后的摘要进行拼接，得到 access_token 值

- **ACCESS_KEY 和 SECRET_KEY 的值**
  ACCESS_KEY = "D39690AAB8AF914630E99150C2891F55B3BFBDA3"
  SECRET_KEY = "99197B09707944D9E8C458CF707E19A1BB05FDB8"
- **原始 URL**
  http://111.4.115.170:9011/api?method=study-list
- **加入时间戳参数后**
  http://111.4.115.170:9011/api?method=study-list&e=1421317725
- **参数排序后**
  http://localhost:9000/api?cardNo=2013110000000001&e=1411117759&method=query-card
- **HMAC_SHA1 签名（摘要 → base64 编码 → URL 安全处理）**
  D39690AAB8AF914630E99150C2891F55B3BFBDA3:kNHMOygciCLd_6ZaQxjOHf-dhj4=
- **最终拼接完成的 URL**
  http://111.4.115.170:9011/api?method=study-list&e=1421317725&token=D39690AAB8AF914630E99150C2891F55B3BFBDA3:kNHMOygciCLd_6ZaQxjOHf-dhj4=

110. 多线程

1> 多线程的概念：

是同步完成多项任务，提高了资源的使用效率，多核的CPU运算多线程更为出色;在iOS应用中，对多线程最初的理解，就是并发。通过Cocoa的封装，可以让我们更为方便的使用线程，做过C++的同学可能会对线程有更多的理解，比如线程的创立，信号量、共享变量有认识，Cocoa框架下会方便很多，它对线程做了封装，有些封装，可以让我们创建的对象，本身便拥有线程，也就是线程的对象化抽象，从而减少我们的工程，提供程序的健壮性。

1>多线程的作用：1.实现负载均衡问题，提高 cpu 利用率。

2>使用场景： 数据请求框架中AFN、多张图片上传前要多需要对图片压缩、文件下载、文件读写、视屏图像的采集、处理、显示、保存等耗时操作的地方。通知、Timer和异步函数等都有使用多线程。

3> NSOperationQueue 和 GCD 的区别是什么

GCD(Grand Central Dispatch)是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中,在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；NSOperation是一个抽象类，它封装了线程的细节实现，我们可以通过子类化该对象，加上NSQueue来同面向对象的思维，管理多线程程序。而Operation为我们提供了更多的选择；

2.在 NSOperationQueue 中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而 GCD 没法停止已经加入 queue 的 block(其实是有的，但需要许多复杂的代码)；
3.NSOperation 能够方便地设置依赖关系，我们可以让一个 Operation 依赖于另一个 Operation，这样的话尽管两个 Operation 处于同一个并行队列中，但前者会直到后者执行完毕后再执行；
4.我们能将 KVO 应用在 NSOperation 中，可以监听一个 Operation 是否完成或取消，这样子能比 GCD 更加有效地掌控我们执行的后台任务；
5.在 NSOperation 中，我们能够设置 NSOperation 的 priority 优先级，能够使同一个并行队列中的任务区分先后地执行，而在 GCD 中，我们只能区分不同任务队列的优先级，如果要区分 block 任务的优先级，也需要大量的复杂代码；
6.我们能够对 NSOperation 进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将 block 任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。
7.GCD 是严格的队列，先进先出 FIFO；NSOperation 可以改动 优先级（或者说服务质量）改变执行顺序
8.NSOperation 的高级：最大并发数，控制线程个数，优化了线程的暂停、继续、取消功能（GCD 实现起来太难，可以用 KVO ），依赖关系，可以让异步任务同步执行.

2、GCD 与 NSThread 的区别:

1). NSThread 通过 @selector 指定要执行的方法，代码分散，依靠的是NSObject的分类实现的线程之间的通讯，如果要开线程必须创建多个线程对象。经常只用的是[NSTread current]查看当前的线程。

NSThread是一个控制线程执行的对象，它不如NSOperation抽象，通过它我们可以方便的得到一个线程，并控制它。但NSThread的线程之间的并发控制，是需要我们自己来控制的，可以通过NSCondition实现。

2).GCD 通过 block 指定要执行的代码，代码集中，所有的代码写在一起的，让代码更加简单，易于阅读和维护,不需要管理线程的创建/销毁/复用的过程！程序员不用关心线程的生命周期

3、进程和线程的区别与联系是什么?

进程和线程都是由操作系统所体会的程序运行的基本单元，系统利用该基本单元实现系统对应用的并发性

一个程序至少有一个进程,一个进程至少有一个线程：

进程：拥有独立的内存单元，而多个线程共享一块内存

线程：线程是指进程内的一个执行单元。

联系：线程是进程的基本组成单位

区别：(1)调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位

(2)并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行

(3)拥有资源：进程是拥有系统资源（单独的地址空间），线程不拥有系统资源，但可以访问隶属于进程的资源.

(4)系统开销：在创建或撤消进程时，系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。一个线程死掉就等于整个进程死掉。所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

（5）但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程

4.分别异步执行两个耗时操作，等两次耗时操作都执行完毕后,再回到主线程执行操作.使用队列组(dispatch_group_t)快速,高效的实现上述需求.

```
dispatch_group_t group = dispatch_group_create();                  // 队列组
dispatch_queue_t queue = dispatch_get_global_queue(0, 0); // 全局并发队列

dispatch_group_async(group, queue, ^{          // 异步执行操作 1
    // longTime1
});

dispatch_group_async(group, queue, ^{          // 异步执行操作 2
    // longTime2
});
```

```
    dispatch_group_notify(group, dispatch_get_main_queue(), ^{     // 在主线
程刷新数据
        // reload Data
    });
}
```

实现方式

线程创建有三种方法：使用NSThread创建、使用GCD的dispatch、使用子类化的NSOperation,然后将其加入NSOperationQueue;

在主线程执行代码，方法是performSelectorOnMainThread，

如果想延时执行代码可以用performSelector:onThread:withObject:waitUntilDone:

NSOperation queue是存放NSOperation的集合类。操作和操作队列，基本可以看成java中的线程和线程池的概念。

111. 在项目什么时候选择使用GCD，什么时候选择NSOperation?

答: 项目中使用NSOperation的优点是NSOperation是对线程的高度抽象，在项目中使用它，会使项目的程序结构更好,子类化NSOperation的设计思路,是具有面向对象的优点(复用、封装)，使得实现是多线程支持，而接口简单，建议在复杂项目中使用。

项目中使用GCD的优点是GCD本身非常简单、易用，对于不复杂的多线程操作，会节省代码量，而Block参数的使用，会是代码更为易读，建议在简单项目中使用。

3> 对比iOS中的多线程技术

3.1> pthread
pthread跨平台,使用难度大,需要手动管理线程生命周期
pthread_create创建线程,传参线程标记,线程属性,初始函数,函数参数
3.2> NSThread
NSThread需要手动管理线程生命周期和

3.2> GCD仅仅支持FIFO队列，只可以设置队列的优先级,而NSOperationQueue中的每一个任务都可以被重新设置优先级(setQueuePriority:)，从而实现不同操作的执行顺序调整
3.3> GCD不支持异步操作之间的依赖关系设置。如果某个操作的依赖另一个操作的数据，使用NSOperationQueue能够设置依赖按照正确的顺序执行操作(addDependency:)。GCD则没有内建的依赖关系支持(只能通过Barrior和同步任务手动实现)。
3.4> NSOperationQueue方便停止队列中的任务(cancelAllOperations, suspended),GCD不方便停止队列中的任务.
3.5> NSOperationQueue支持KVO，可以监测operation是否正在执行（isExecuted)、是否结束（isFinished)，是否取消（isCanceld）
3.6> GCD的执行速度比NSOperationQueue快
3.7> NSOperationQueue可设置最大并发数量(节电),GCD具有dispatch_once(只执行一次,单例)

和dispatch_after(延迟执行)功能
3.8> NSObject分类(perform)和NSThread遇到对象分配需要手动内存管理,手动管理线程生命周期
3.10> NSObject分类线程通信

1> 多线程优缺点
优点:
使应用程序的响应速度更快,用户界面在进行其他工作的同时仍始终保持活动状态;
优化任务执行,适当提高资源利用率(cpu, 内存);
缺点:
线程占用内存空间,管理线程需要额外的CPU开销,开启大量线程,降低程序性能;
增加程序复杂度,如线程间通信,多线程的资源共享等;
2> 在多线程中使用通知需要注意什么问题?

3> iOS中的延迟操作
1>[self performSelector:@selector(clearCache) withObject:nil afterDelay:duration];
1>2> dispatch_after(dispatch_time(DISPATCH_TIME_NOW,(int64_t)(2.0 * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{..});

2.利用 NSOperation 与 NSOperationqueue 处理多线程时，有 3 个 NSOperation 分别为 A,B,C，要求 A,B 执行完毕后，在执行 C，如何做？

有三种实现方案

方案一：串行队列同步执行（GCD 实现）

A,B 放在串行队列中执行，执行完毕，主队列执行任务 C

方案二：队列依赖实现

方案三：并发队列和主队列实现

http://www.jianshu.com/p/0b0d9b1f1f19

（1）串行队列同步执行和异步主队列

dispatch_sync+串行队列 ,同步执行，能够保证任务 A,B 顺序执行

dispatch_async　　A,B 任务执行完毕后，主线程再执行任务 C

（2）NSOperation 有一个非常实用的功能，那就是添加依赖。比如有 3 个任务：A：从服务器上下载一张图片，B：给这张图片加个水印，C：把图片返回给服务器。这时就可以用到依赖了：

利用 NSBlockOperation 创建三个任务

设置依赖关系

创建队列并加入任务

（3）同步执行 ：我们可以使用多线程的知识，把多个线程都要执行此段代码添加到同一个串行队列，这样就实现了线程同步的概念。当然这里可以使用 GCD 和 NSOperation 两种方案。

NSOperationQueue 有一个参数 maxConcurrentOperationCount 最大并发数，用来设置最多可以让多少个任务同时执行。当你把它设置为 1 的时候，他不就是串行了嘛！

```
[[NSOperationQueue mainQueue] addOperationWithBlock:^{

这里面执行任务 C

}];
```

112. dispatch_barrier_async 的作用是什么？

在并行队列中 ,为了保持某些任务的顺序 ,需要等待一些任务完成后才能继续进行 ,使用 barrier 来等待之前任务完成，避免数据竞争等问题。 dispatch_barrier_async 函数会等待追加到 Concurrent Dispatch Queue 并行队列中的操作全部执行完之后，然后再执行 dispatch_barrier_async 函数追加的处理，等 dispatch_barrier_async 追加的处理执行结束之后，Concurrent Dispatch Queue 才恢复之前的动作继续执行。

打个比方：比如你们公司周末跟团旅游，高速休息站上，司机说：大家都去上厕所，速战速决，上完厕所就上高速。超大的公共厕所，大家同时去，程序猿很快就结束了，但程序媛就可能会慢一些，即使你第一个回来，司机也不会出发，司机要等待所有人都回来后，才能出发。dispatch_barrier_async 函数追加的内容就如同 “上完厕所就上高速”这个动作。

（注意：使用 dispatch_barrier_async ，该函数只能搭配自定义并行队列 dispatch_queue_t 使用。不能使用： dispatch_get_global_queue ，否则 dispatch_barrier_async 的作用会和 dispatch_async 的作用一模一样。 ）

一般情况下，选择使用 GCD 的 dispatch_after。

1. performSelector 和 scheduledTimerWithTimeInterval 方法都是基于 runloop 的。

当一个应用启动时，系统会开启一个主线程，并且把主线程的 runloop 激活，也就是 run 起来，并且主线程的 runloop 是不会停止的。所以，当这两个方法在主线程可以被正常调用。

们更多的逻辑是放在子线程中执行的。而子线程的 runloop 是默认关闭的。这时如果不手动激活 runloop，performSelector 和 scheduledTimerWithTimeInterval 的调用将是无效的。

2.NSTimer 的创建与撤销必须在同一个线程操作、performSelector 的创建与撤销必须在同一个线程操作。

scheduledTimerWithTimeInterval 方法将 target 设为 A 对象时,A 对象会被这个 timer 所持有，也就是会被 retain 一次，timer 会被当前的 runloop 所持有。
performSelector:withObject:afterDelay:方法实际上是在当前线程的 runloop 里帮你创建的一个 timer 去执行任务，所以和 scheduledTimerWithTimeInterval 方法一样会 retain 其调用对象。但是，我们往往不希望因为这些延迟操作而影响对象的生命周期，更甚至是，导致对象无法释放.

3.内存管理有潜在泄露的风险

3.dispatch_after 有个致命的弱点：dispatch_after 一旦执行后，就不能撤销了。而 performSelector 可以使用 cancelPreviousPerformRequestsWithTarget 方法撤销，NSTimer 也可以调用 invalidate 进行撤销。

解决办法：

```
// 拿到一个队列
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
// 创建一个timer放到队列里面
dispatch_source_t timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, queue);
// 设置timer的首次执行的时间、执行时间间隔、精确度
dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, 2.0 * NSEC_PER_SEC, 0.1 * NSEC_PER_SEC);
// 设置timer执行的事件
__weak typeof(self) weakSelf = self;
dispatch_source_set_event_handler(timer, ^{
    [weakSelf doSomething];
});
// 激活timer
dispatch_resume(timer);

// 取消timer
dispatch_source_cancel(timer);
```

113.在多线程 Core Data 中，NSC,MOC,NSObjectModel 哪些需要在线程中创建或者传递？你是用什么策越来实现的？

Core Data 是 iOS5 之后才出现的一个框架，它提供了对象-关系映射(ORM)的功能，即能够将 OC 对象转化成数据，保存在 SQLite 数据库文件中，也能够将保存在数据库中的数据还原成 OC 对象。在此数据操作期间，我们不需要编写任何 SQL 语句，这个有点类似于著名的 Hibernate 持久化框架，不过功能肯定是没有 Hibernate 强大的。简单地用下图描述下它的作用：

利用 Core Data 框架，我们就可以轻松地将数据库里面的 2 条记录转换成 2 个 OC 对象，也可以轻松地将 2 个 OC 对象保存到数据库中，变成 2 条表记录，而且不用写一条 SQL 语句，另外支持自动撤销机制，一对多关联等

Core data 是 Cocoa 中处理数据，绑定数据的关键特性，其重要性不言而喻，但也比较复杂。Core Data 相关的类比较多，初学者往往不太容易弄懂。1> CoreData 是对 SQLite 数据库的封装

2> CoreData 中的 NSManagedObjectContext 在多线程中不安全

3> 如果想要多线程访问 CoreData 的话，最好的方法是一个线程一个 NSManagedObjectContext

4> 每个 NSManagedObjectContext 对象实例都可以使用同一个

NSPersistentStoreCoordinator 实例，这是因为 NSManagedObjectContext 会在便用 NSPersistentStoreCoordinator 前上锁

NSC 持久化存储调度器

PSC 使用 NSPersistentStore 对象与数据库进行交互，NSPersistentStore 对象会将 MOC 提交的改变同步到数据库中

MOC 管理对象上下文

NSObjectMode 对象模型

于 MOC 不是线程安全的，如果多线程共用 MOC 的话会出现数据混乱，甚至更严重的会导致程序崩溃。此外 MO 也不是线程安全的，但是 PSC 是线程安全的，因为 Context 将改变提交给 PSC 时，系统会为其上锁，确保一次只执行一个 MOC 提交的改变。

一个线程使用一个 MOC 对象，并且在该线程中只能操作它所监听的 MO。

使用两个 MOC，一个负责在后台处理各种耗时的操作，一个负责与 UI 进行协作。

我们知道 MOC 和 MO 不是线程安全的，为了解决这个问题我们在一个线程中仅使用一个 MOC，因此不能跨线程访问同一个 MOC 和 MO。但是这会存在问题。

比如：我在后台线程中执行 update 操作，主线程中的上下文是不知道的，所以当我在主线程中去进行查询时，查询出来的结果并不是已更新后的结果。

为了解决这个问题，我们需要监听通知：NSManagedObjectContextDidSaveNotification，在耗时操作处理完之后去告诉主上下文发生了哪些改变。我们可以通过主线程执行合并操作来实现。

我们将探讨从旧模型中提取数据并使用这些数据来填充具有新的实体和关系的目标模型。

当你要升级你的数据模型到新版，你将先选择一个基准模型。对于轻量级迁移，持久化存储会为你自动推断一个映射模型。然而，如果你对新模型所做的修改并不被轻量级迁移所支持，那么你就需要创建一个映射模型。一个映射模型需要一个源数据模型和一个目标数据模型。

114.+(void)load 与 +(void)initialize 区别

load 方法：

1. 当类加载到 OC 运行时环境(内存)中的时候,就会调用一次(一个类只会加载一次).
2. 程序一启动就会调用.
3. 程序运行过程中,只会调用 1 次.

initialize 方法：

1. 当第一次使用这个类的时候(比如调用了类的某个方法)才会调用.
2. 并非程序一启动就会调用.

load 和 initialize 的共同特点

在不考虑开发者主动使用的情况下，系统最多会调用一次

如果父类和子类都被调用，父类的调用一定在子类之前

都是为了应用运行提前创建合适的运行环境

在使用时都不要过重地依赖于这两个方法，除非真正必要

它们的相同点在于：方法只会被调用一次。（其实这是相对 runtime 来说的，后边会做进一步解释）。

load 是只要类所在文件被引用就会被调用，而 initialize 是在类或者其子类的第一个方法被调用前调用。所以如果类没有被引用进项目，就不会有 load 调用；但即使类文件被引用进来，但是没有使用，那么 initialize 也不会被调用。

文档也明确阐述了方法调用的顺序：父类(Superclass)的方法优先于子类(Subclass)的方法，类中的方法优先于类别(Category)中的方法。

2> load 和 initialize方法的区别

	+(void)load	+(void)initialize
执行时机	在程序运行后立即执行	在类的方法第一次被调时执行
若自身未定义，是否沿用父类的方法?	否	是
类别中的定义	全都执行，但后于类中的方法	覆盖类中的方法，只执行一个

115.http 的 post 与区别与联系，实践中如何选择它们？

（1）get 是从服务器上获取数据，post 是向服务器传送数据。

（2）在客户端，Get 方式在通过 URL 提交数据，数据在 URL 中可以看到；POST 方式，数据放置 body 内提交。

（3）对于 get 方式，服务器端用 Request.QueryString 获取变量的值，对于 post 方式，服务器端用 Request.Form 获取提交的数据。

（4）GET 方式提交的数据最多只能有 1024 字节，而 POST 则没有此限制。

（5）安全性问题。正如在（1）中提到，使用 Get 的时候，参数会显示在地址栏上，而 Post 不会。所以，如果这些数据是中文数据而且是非敏感数据，那么使用 get；如果用户输入的数据不是中文字符而且包含敏感数据，那么还是使用 post 为好。

(6)post 可以设置书签

（7）在做数据查询时，建议用 Get 方式

116.说说关于 UDP/TCP 的区别？

UDP（User Data Protocol，用户数据报协议）是与 TCP 相对应的协议。它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去！ UDP 适用于一次只传送少量数据、对可靠性要求不高的应用环境。

1.只管发送，不确认对方是否接收到

2.将数据及源和目的封装成数据包中，不需要建立连接

3.每个数据报的大小限制在 64K 之内

4.因为无需连接，因此是不可靠协议

5.不需要建立连接，速度快

应用场景：多媒体教室 / 网络流媒体

TCP（Transmission Control Protocol，传输控制协议）是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次“对话”才能 建立起来，其中的过程非常复杂，我们这里只做简单、形象的介绍，你只要做到能够理解这个过程即可。

1.建立连接，形成传输数据的通道

2.在连接中进行大数据传输（数据大小不收限制）

3.通过三次握手完成连接，是可靠协议，安全送达

4.必须建立连接，效率会稍低

TCP传输控制协议主要包含下列任务和功能：

* 确保IP数据报的成功传递。

* 对程序发送的大块数据进行分段和重组。

* 确保正确排序及按顺序传递分段的数据。

* 通过计算校验和，进行传输数据的完整性检查。

简单的说，TCP注重数据安全，而UDP数据传输快点，但安全性一般，流模式与数据报模式，TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证

十八：长链接&短链接

TCP 短连接

我们模拟一下 TCP 短连接的情况，client 向 server 发起连接请求，server 接到请求，然后双方建立连接。client 向 server 发送消息，server 回应 client，然后一次读写就完成了，这时候双方任何一个都可以发起 close 操作，不过一般都是 client 先发起 close 操作。为什么呢，一般的 server 不会回复完 client 后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作

TCP 长连接

接下来我们再模拟一下长连接的情况，client 向 server 发起连接，server 接受 client 连接，双方建立连接。Client 与 server 完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

长连接短连接操作过程

短连接的操作步骤是：

建立连接——数据传输——关闭连接…建立连接——数据传输——关闭连接

长连接的操作步骤是：

建立连接——数据传输…（保持连接）…数据传输——关闭连接

什么时候用长连接，短连接？

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个 TCP 连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话 那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就 OK 了，不用建立 TCP 连接。例如：数

据库的连接用长连接， 如果用短连接频繁的通信会造成 socket 错误，而且频繁的 socket 创建也是对资源的浪费。

而像 WEB 网站的 http 服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像 WEB 网站这么频繁的成千上万甚至上亿客户端的连接 用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操 作情况下需用短连好。

长连接和短连接的优点和缺点

由上可以看出，长连接可以省去较多的 TCP 建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。不过这里存在一个问 题，存活功能的探测周期太长，还有就是它只是探测 TCP 连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。在长连接的应用场景 下，client 端一般不会主动关闭它们之间的连接，Client 与 server 之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越 多，server 早晚有扛不住的时候，这时候 server 端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可 以避免一些恶意连接导致 server 端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在 TCP 的建立和关闭操作上浪费时间和带宽。

长连接和短连接的产生在于 client 和 server 采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

TCP 传输原理

1>TCP如何防止乱序和丢包

TCP数据包的头格式中有两个概念,Sequence Number是数据包的序号，用来解决网络包乱序（reordering）问题。Acknowledgement Number就是ACK——用于确认收到，用来解决不丢包的问题。

位码即tcp标志位，有6种标示：SYN(synchronous建立联机) ACK(acknowledgement 确认) PSH(push传送) FIN(finish结束) RST(reset重置) URG(urgent紧急)Sequence number(顺序号码) Acknowledge number(确认号码).

SeqNum的增加是和传输的字节数相关的,TCP传输数据时,A主机第一次传输1440个字节,seq=1,那么第二次时seq = 1441,B拼接数据就是根据seq进行拼接的,seq数字不断累加避免了乱序.B主机收到第一次数据包以后会返回ack = 1441.

A主机收到B的ack = 1441时,就知道第一个数据包B已收到. 如果B没有收到第一次的数据包,那么B再收到A的数据包时,他就会发ack = 1回去,A收到B的回复,发现B没有收到第一次数据包,就会重发第一次数据包,这样就可以防止丢包.

2>描述一下三次握手

第一次握手：建立连接时，客户端发送syn包(syn=j)到服务器，并进入SYN_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。完成三次握手，客户端与服务器开始传送数据.

3> 三次握手实现的过程：

第一次握手：建立连接时，客户端发送同步序列编号到服务器，并进入发送状态，等待服务器确认

第二次：服务器收到同步序列编号，并确认同时自己也发送一个同步序列编号+确认标志，此时服务器进入接收状态

第三次：客户端收到服务器发送的包，并向服务器发送确认标志，随后链接成功。

注意：是在链接成功后在进行数据传输。

117.http 和 scoket 通信的区别?socket 连接相关库,TCP,UDP 的连接方法,HTTP 的几种常用方式?

http 和 scoket 通信的区别:

http 是客户端用 http 协议进行请求,发送请求时候需要封装 http 请求头,并绑定请求的数据,服务器一般有 web 服务器配合(当然也非绝对)。 http 请求方式为客户端主动发起请求,服务器才能给响应,一次请求完毕后则断开连接,以节省资源。服务器不能主动给客户端响应(除非采取 http 长连接技术)。iphone 主要使用类是 NSUrlConnection。

scoket 是客户端跟服务器直接使用 socket“套接字”进行连接,并没有规定连接后断开,所以客户端和服务器可以保持连接通道,双方都可以主动发送数据。一般在游戏开发或股票开发这种要求即时性很强并且保持发送数据量比较大的场合使用。主要使用类是 CFSocketRef。

118. HTTP请求常用的几种方式

GET ： 获取指定资源
POST ： 2M 向指定资源提交数据进行处理请求，在RESTful 风格用于新增资源
HEAD ： 获取指定资源头部信息
PUT ：替换指定资源（不支持浏览器操作）
DELETE: 删除指定资源

119.BLOCK

定义：1.对于闭包(block),闭包就是能够读取其它函数内部变量的函数。

2.Block是可以获取其他函数局部变量的匿名函数

实现原理：block的实现是基于指针和函数指针，block属性是指向结构体的指针，
使用场景：动画、数组字典排序遍历、回调状态、错误控制、多线程GCD
注意循环引用：__block 修饰局部变量,这个变量在 block 内外属于同一个地址 上的变量,可以被 block 内部的代码修改
优点：其不但方便开发，并且可以大幅提高应用的执行效率(多核心CPU可直接处理Block指令)

block:循环引用问题

1> 使用 block 时什么情况会发生引用循环，如何解决？

一个对象中强引用了 block，在 block 中又使用了该对象，就会发射循环引用。 解决方法是将该对象使用__weak 或者__block 修饰符修饰之后再在 block 中使用。

id weak weakSelf = self; 或者 weak __typeof(&*self)weakSelf = self 该方法可以设置宏

id __block weakSelf = self;

2> 在 block 内如何修改 block 外部变量？

在 block 中访问的外部变量是复制过去的，即：写操作不对原变量生效。但是你可以加上__block 来让其写操作生效，示例代码如下：

```
__block int a = 0;

void  (^foo)(void) = ^{

    a = 1;

}

f00();

//这里，a 的值被修改为 1
```

3> Block&MRC-Block

block 虽然好用，但是里面也有不少坑，最大的坑莫过于循环引用问题。稍不注意，可能就会造成内存泄漏。这节，我将从源码的角度来分析造成循环引用问题的根本原因。并解释变量前加 block，和 weak 的区别。

明确两点

1,Block 可以访问 Block 函数以及语法作用域以内的外部变量。也就是说:一个函数里定义了个 block，这个 block 可以访问该函数的内部变量(当然还包括静态，全局变量)-即 block 可以使用和本身定义范围相同的变量。 2,Block 其实是特殊的 Objective-C 对象，可以使用 copy,release 等来管理内存,但和一般的 NSObject 的管理方式有些不 同，稍后会说明。

MRC:防止 block 对 self 的引用 解决办法

__block typeof(self) weakSelf = self;

ARC:防止 block 对 self 的引用 解决办法

__weak typeof(self) weakSelf = self;

对于非 ARC 下，为了防止循环引用，我们使用__block 来修饰在 Block 中使用的对象:

对于 ARC 下，为了防止循环引用，我们使用 weak 来修饰在 Block 中使用的对象。原理就是:ARC 中，Block 中如果引用了 strong 修饰符的自动变量，则相当于 Block 对该变量的引用计数+1。

使用实例:

cocoaTouch框架下动画效果的Block的调用

使用typed声明block

// 这就声明了一个didFinishBlock类型的block，

typedef void(^didFinishBlock) (NSObject *obj);

// 这就声明了一个didFinishBlock类型的block，

typedef void(^didFinishBlock) (NSObject *obj);

然后便可用

@property (nonatomic,copy) didFinishBlock finishBlock;

@property (nonatomic,copy) didFinishBlock finishBlock;

声明一个blokc对象，注意对象属性设置为copy，接到block 参数时，便会自动复制一份。(栈->堆)
__block是一种特殊类型，

使用该关键字声明的局部变量，可以被block所改变，并且其在原函数中的值会被改变。

60.block 面试技巧

答: 面试时，面试官会先问一些，是否了解block，是否使用过block，这些问题相当于开场白，往往是下面一系列问题的开始，所以一定要如实根据自己的情况回答。

1). 使用block和使用delegate完成委托模式有什么优点?

首先要了解什么是委托模式，委托模式在iOS中大量应用，其在设计模式中是适配器模式中的对象适配器，Objective-C中使用id类型指向一切对象，使委托模式更为简洁。了解委托模式的细节：

iOS设计模式—-委托模式

使用block实现委托模式，其优点是回调的block代码块定义在委托对象函数内部，使代码更为紧凑;

适配对象不再需要实现具体某个protocol，代码更为简洁。

2). 多线程与block

GCD与Block

使用 dispatch_async 系列方法，可以以指定的方式执行block

GCD编程实例

dispatch_async的完整定义

```
void dispatch_async(dispatch_queue_t queue,dispatch_block_t block);
```

1

```
void dispatch_async(dispatch_queue_t queue,dispatch_block_t block);
```

功能：在指定的队列里提交一个异步执行的block，不阻塞当前线程

通过queue来控制block执行的线程。主线程执行前文定义的 finishBlock对象

```
dispatch_async(dispatch_get_main_queue(),^(void){finishBlock();});
```

```
dispatch_async(dispatch_get_main_queue(),^(void){finishBlock();});
```

50.写出一个使用Block执行UIVew动画?

```
[UIView transitionWithView:self.view
                  duration:0.2
                   options:UIViewAnimationOptionTransitionFlipFromLeft
                animations:^{  [[blueViewController  view]  removeFromSuperview];
```

```
[[self view] insertSubview:yellowViewController.view atIndex:0]; }

                completion:NULL];

[UIView transitionWithView:self.view

                  duration:0.2

                   options:UIViewAnimationOptionTransitionFlipFromLeft

                animations:^{  [[blueViewController  view]  removeFromSuperview];
[[self view] insertSubview:yellowViewController.view atIndex:0]; }

                completion:NULL];
```

51. 写出上面代码的Block的定义。

```
typedef void(^animations) (void);

typedef void(^completion) (BOOL finished);

typedef void(^animations) (void);

typedef void(^completion) (BOOL finished);
```

120.Weak、strong、copy、assign 使用

1. 什么情况使用 weak 关键字，相比 assign 有什么不同？

什么情况使用 weak 关键字？

在 ARC 中,在有可能出现循环引用的时候,往往要通过让其中一端使用 weak 来解决,比如:delegate 代理属性

自身已经对它进行一次强引用,没有必要再强引用一次,此时也会使用 weak,自定义 IBOutlet 控件属性一般也使用 weak；当然，也可以使用 strong。在下文也有论述：《IBOutlet 连出来的视图属性为什么可以被设置成 weak?》

不同点：

weak 此特质表明该属性定义了一种“非拥有关系” (nonowning relationship)。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。此特质同 assign 类似， 然而在属性所指的对象遭到摧毁时，属性值也会清空(nil out)。 而 assign 的“设置方法”只会执行针对“纯量类型” (scalar type，例如 CGFloat 或 NSInteger 等)的简单赋值操作。

assigin 可以用非 OC 对象,而 weak 必须用于 OC 对象

2. 怎么用 copy 关键字？

用途：

NSString、NSArray、NSDictionary 等等经常使用 copy 关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary；

block 也经常使用 copy 关键字，具体原因见官方文档：Objects Use Properties to Keep Track of Blocks：

block 使用 copy 是从 MRC 遗留下来的“传统”,在 MRC 中,方法内部的 block 是在栈区的,使用 copy 可以把它放到堆区.在 ARC 中写不写都行：对于 block 使用 copy 还是 strong 效果是一样的，但写上 copy 也无伤大雅，还能时刻提醒我们：编译器自动对 block 进行了 copy 操作。如果不写 copy ，该类的调用者有可能会忘记或者根本不知道“编译器会自动对 block 进行了 copy 操作”，他们有可能会在调用之前自行拷贝属性值。这种操作多余而低效。

copy 此特质所表达的所属关系与 strong 类似。然而设置方法并不保留新值，而是将其“拷贝” (copy)。 当属性类型为 NSString 时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个 NSMutableString 类的实例。这个类是 NSString 的子类，表示一种可修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变” (immutable)的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的” (mutable)，就应该在设置新属性值时拷贝一份。

一: weak&strong

strong 表示保留它指向的堆上的内存区域不再指向这块区域了。 也就是说我强力指向了一个区域，我们不再指向它的条件只有我们指向 nil 或者我自己也不在内存上，没有人 strong 指向我了。

weak 表示如果还没有人指向它了，它就会被清除内存，同时被指向 nil，因为我不能读取不存在的东西。

weak 只在 iOS5.0 使用

这并不是垃圾回收，我们用 reference count 表示堆上还有多少 strong 指针，当它变为 0 就马上释放。

本地变量都是 strong，编辑器帮你计算.

补充

管理机制：使用了一种叫做引用计数的机制来管理内存中的对象。OC 中每个对象都对应着他们自己的引用计数，引用计数可以理解为一个整数计数器，当使 用 alloc 方法创建对象的时

候，持有计数会自动设置为 1。当你向一个对象发送 retain 消息 时，持有计数数值会增加 1。相反，当你像一个对象发送 release 消息时，持有计数数值会减小 1。当对象的持有计数变为 0 的时候，对象会释放自己所占用 的内存。 retain(引用计数加 1)->release（引用计数减 1）alloc（申请内存空间）->dealloc(释放内存空间) readwrite: 表示既有 getter，也有 setter (默认) readonly: 表示只有 getter，没有 setter nonatomic:不考虑线程安全 atomic:线程操作安全 （默认） 线程安全情况下的 setter 和 getter:

(NSString*) value { @synchronized(self) { return [[_value retain] autorelease];

} }

(void) setValue:(NSString)aValue { @synchronized(self) { [aValue retain]; [value release]; value = aValue;

} } retain: release 旧的对象，将旧对象的值赋予输入对象，再提高输入对象的索引计数为 1 assign: 简单赋值，不更改索引计数 （默认） copy: 其实是建立了一个相同的对象,地址不同（retain：指针拷贝 copy：内容拷贝） strong:（ARC 下的）和（MRC）retain 一样 （默认） weak:（ARC 下的）和（MRC）assign 一样， weak 当指向的内存释放掉后自动 nil 化，防止野指针 unsafe_unretained 声明一个弱应用，但是不会自动 nil 化，也就是说，如果所指向的内存区域被释放了，这个指针就是一个野指针了,声明的属性 dealloc 里面设置为 nil。 autoreleasing 用来修饰一个函数的参数，这个参数会在函数返回的时候被自动释放。1、类变量的@protected ,@private,@public,@package，声明各有什么含义？ @private：作用范围只能在自身类 @protected：作用范围在自身类和继承自己的子类 （默认） @public：作用范围最大，可以在任何地方被访问。 @package：这个类型最常用于框架类的实例变量,同一包内能用，跨包就不能访问

3. 这个写法会出什么问题： @property (copy) NSMutableArray *array

两个问题：1、添加,删除,修改数组内的元素的时候,程序会因为找不到对应的方法而崩溃.因为 copy 就是复制一个不可变 NSArray 的对象；2、使用了 atomic 属性会严重影响性能

第 1 条的相关原因在下文中有论述《用@property 声明的 NSString（或 NSArray,NSDictionary）经常使用 copy 关键字，为什么？如果改用 strong 关键字，可能造成什么问题？》 以及上文《怎么用 copy 关键字？》也有论述。

比如下面的代码就会发生崩溃

// .h 文件

// http://weibo.com/luohanchenyilong/

```
// https://github.com/ChenYilong

// 下面的代码就会发生崩溃

@property (nonatomic, copy) NSMutableArray *mutableArray;

// .m 文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 下面的代码就会发生崩溃

NSMutableArray *array = [NSMutableArray arrayWithObjects:@1,@2,nil];

self.mutableArray = array;

[self.mutableArray removeObjectAtIndex:0];
```

接下来就会奔溃：

```
-[__NSArrayI removeObjectAtIndex:]: unrecognized selector sent to instance 0x7fcd1bc30460
```

第 2 条原因，如下：

该属性使用了同步锁，会在创建时生成一些额外的代码用于帮助编写多线程程序，这会带来性能问题，通过声明 nonatomic 可以节省这些虽然很小但是不必要额外开销。

在默认情况下，由编译器所合成的方法会通过锁定机制确保其原子性(atomicity)。如果属性具备 nonatomic 特质，则不使用同步锁。请注意，尽管没有名为“atomic”的特质(如果某属性不具备 nonatomic 特质，那它就是“原子的”(atomic))。

在 iOS 开发中，你会发现，几乎所有属性都声明为 nonatomic。

一般情况下并不要求属性必须是“原子的”，因为这并不能保证“线程安全”（ thread safety)，若要实现“线程安全”的操作，还需采用更为深层的锁定机制才行。例如，一个线程在连续多次读取某属性值的过程中有别的线程在同时改写该值，那么即便将属性声明为 atomic，也还是会读到不同的属性值。

因此，开发 iOS 程序时一般都会使用 nonatomic 属性。但是在开发 Mac OS X 程序时，使用 atomic 属性通常都不会有性能瓶颈

4. 如何让自己的类用 copy 修饰符？如何重写带 copy 关键字的 setter？

若想令自己所写的对象具有拷贝功能，则需实现 NSCopying 协议。如果自定义的对象分为可变版本与不可变版本，那么就要同时实现 NSCopying 与 NSMutableCopying 协议

具体步骤：

需声明该类遵从 NSCopying 协议

实现 NSCopying 协议。该协议只有一个方法：

- (id)copyWithZone:(NSZone *)zone;

注意：一提到让自己的类用 copy 修饰符，我们总是想覆写 copy 方法，其实真正需要实现的却是 “copyWithZone” 方法

但在实际的项目中，不可能这么简单，遇到更复杂一点，比如类对象中的数据结构可能并未在初始化方法中设置好，需要另行设置。举个例子，假如 CYLUser 中含有一个数组，与其他 CYLUser 对象建立或解除朋友关系的那些方法都需要操作这个数组。那么在这种情况下，你得把这个包含朋友对象的数组也一并拷贝过来。下面列出了实现此功能所需的全部代码：

【注：深浅拷贝的概念，在下文中有介绍，详见下文的：用@property 声明的 NSString（或 NSArray，NSDictionary）经常使用 copy 关键字，为什么？如果改用 strong 关键字，可能造成什么问题？】

在例子中，存放朋友对象的 set 是用 “copyWithZone:” 方法来拷贝的，这种浅拷贝方式不会逐个复制 set 中的元素。若需要深拷贝的话，则可像下面这样，编写一个专供深拷贝所用的方法：

6. @property 的本质是什么？ivar、getter、setter 是如何生成并添加到这个类中的

@property = ivar + getter + setter;

属性” (property)作为 Objective-C 的一项特性，主要的作用就在于封装对象中的数据。Objective-C 对象通常会把其所需要的数据保存为各种实例变量。实例变量一般通过“存取方法”(access method)来访问。其中，“获取方法” (getter)用于读取变量值，而“设置方法”(setter)用于写入变量值。这个概念已经定型，并且经由“属性”这一特性而成为 Objective-C 2.0 的一部分。 而在正规的 Objective-C 编码风格中，存取方法有着严格的命名规范。 正因为有了这种严格的命名规范，所以 Objective-C 这门语言才能根据名称自动创建出存取方法。其实也可以把属性当做一种关键字，其表示：

编译器会自动写出一套存取方法，用以访问给定类型中具有给定名称的变量。 所以你也可以这么说：

@property = getter + setter;

ivar、getter、setter 是如何生成并添加到这个类中的?

“自动合成”(autosynthesis)

完成属性定义后，编译器会自动编写访问这些属性所需的方法，此过程叫做“自动合成”(autosynthesis)。需要强调的是，这个过程由编译 器在编译期执行，所以编辑器里看不到这些“合成方法”(synthesized method)的源代码。除了生成方法代码 getter、setter 之外，编译器还要自动向类中添加适当类型的实例变量，并且在属性名前面加下划线，以此作为实例变量的名字。在前例中，会生成两个实例变量，其名称分别为 _firstName 与 _lastName。也可以在类的实现代码里通过 @synthesize 语法来指定实例变量的名字.

@implementation Person

@synthesize firstName = _myFirstName;

@synthesize lastName = _myLastName;

@end

6.用@property 声明的 NSString（或 NSArray，NSDictionary）经常使用 copy 关键字，为什么？如果改用 strong 关键字，可能造成什么问题？

因为父类指针可以指向子类对象,使用 copy 的目的是为了让本对象的属性不受外界影响,使用 copy 无论给我传入是一个可变对象还是不可对象,我本身持有的就是一个不可变的副本.

如果我们使用是 strong,那么这个属性就有可能指向一个可变对象,如果这个可变对象在外部被修改了,那么会影响该属性.

7.@protocol 和 category 中如何使用 @property

在 protocol 中使用 property 只会生成 setter 和 getter 方法声明,我们使用属性的目的,是希望遵守我协议的对象能实现该属性

category 使用 @property 也是只会生成 setter 和 getter 方法的声明,如果我们真的需要给 category 增加属性的实现,需要借助于运行时的两个函数：

objc_setAssociatedObject

objc_getAssociatedObject

8.runtime 如何通过 selector 找到对应的 IMP 地址？

每一个类对象中都一个方法列表（isa）,方法列表中记录着方法的名称,方法实现,以及参数类型,其实 selector 本质就是方法名称,通过这个方法名称就可以在方法列表中找到对应的方法实现.

9.retain 和 copy 区别

copy 其实是建立了一个相同的对象，而 retain 不是：

比如一个 NSString 对象，地址为 0×1111，内容为@”STR”

Copy 到另外一个 NSString 之 后，地址为 0×2222，内容相同，新的对象 retain 为 1， 旧有对象没有变化

retain 到另外一个 NSString 之 后，地址相同（建立一个指针，指针拷贝），内容当然相同，这个对象的 retain 值+1

也就是说，retain 是指针拷贝，copy 是内容拷贝

copy 是创建一个新对象，retain 是创建一个指针。 copy：建立一个索引计数为 1 的新对象，然后释放旧对象。新的对象 retain 为 1，与旧有对象引用技术无关，减少了对象对上下文的依赖。retain：释放旧的对象，将旧对象的值赋予输入新对象，再提高输入对象的索引计数为 1，新对象和旧对象指针相同。

10.copy 和 strong 的使用？

我们在声明一个 NSString 属性时，对于其内存相关特性，通常有两种选择(基于 ARC 环境)：strong 与 copy。那这两者有什么区别呢？什么时候该用 strong，什么时候该用 copy 呢？

由于 NSMutableString 是 NSString 的子类,所以一个 NSString 指针可以指向 NSMutableString 对象，让我们的 strongString 指针指向一个可变字符串是 OK 的。

而上面的例子可以看出，当源字符串是 NSString 时，由于字符串是不可变的，所以，不管是 strong 还是 copy 属性的对象，都是指向源对象，copy 操作只是做了次浅拷贝。

当源字符串是 NSMutableString 时，strong 属性只是增加了源字符串的引用计数，而 copy 属性则是对源字符串做了次深拷贝，产生一个新的对象，且 copy 属性对象指向这个新的对象。另外需要注意的是，这个 copy 属性对象的类型始终是 NSString，而不是 NSMutableString，因此其是不可变的。

这里还有一个性能问题，即在源字符串是 NSMutableString，strong 是单纯的增加对象的引用计数，而 copy 操作是执行了一次深拷贝，所以性能上会有所差异。而如果源字符串是 NSString 时，则没有这个问题。

所以，在声明 NSString 属性时，到底是选择 strong 还是 copy，可以根据实际情况来定。不过，一般我们将对象声明为 NSString 时，都不希望它改变，所以大多数情况下，我们建议用 copy，以免因可变字符串的修改导致的一些非预期问题。

2.NSString 和 NSMutableString，前者线程安全，后者线程不安全。

NSString 和 NSMutableString，前者接收到 copy 时，是浅拷贝，后者是深拷贝，但返回的都是不可变对象，即 NSString 对象

NSString 和 NSMutableString，接收到 mutableCopy 时，都是深拷贝，并且返回的都是可变对象，即 NSMutableString 对象

NSString 和 NSMutableString，strong 修饰时属性时一样，都是强引用的概念，赋值时不会接收到 copy 或 mutableCopy 消息

很简单,假如有一个 NSMutableString,现在用他给一个 retain 修饰 NSString 赋值,那么只是将 NSString 指向了 NSMutableString 所指向的位置,并对 NSMUtbaleString 计数器加一,此时,如果对 NSMutableString 进行修改,也会导致 NSString 的值修改,原则上这是不允许的. 如果是 copy 修饰的 NSString 对象,在用 NSMutableString 给他赋值时,会进行深拷贝,及把内容也给拷贝了一份,两者指向不同的位置,即使改变了 NSMutableString 的值,NSString 的值也不会改变.

所以用 copy 是为了安全,防止 NSMutableString 赋值给 NSString 时,前者修改引起后者值变化而用的.

86. readwrite，readonly，assign，retain，copy，weak ,strong,nonatomic 属性的作用

答：@property是一个属性访问声明，扩号内支持以下几个属性：

1).getter=getterName，setter=setterName，设置setter与 getter的方法名

2).readwrite,readonly，设置可供访问级别

2).assign，setter方法直接赋值，不进行任何retain操作，为了解决原类型与环循引用问题。用于非指针变量。用于基础数据类型 （例如NSInteger)和C数据类型(int, float, double, char, 等）,另外还有id,其setter方法直接赋值，不进行任何retain操作

3).retain，setter方法对参数进行release旧值再retain新值，所有实现都是这个顺序(CC上有相关资料)

4).copy，setter方法进行Copy操作，与retain处理流程一样，先旧值release，再 Copy出新的对象，retainCount为1。这是为了减少对上下文的依赖而引入的机制。

5).nonatomic，决定编译器生成的setter getter是非原子操作,非原子性访问，不加同步，多线程并发访问会提高性能。注意，如果不加此属性，则默认是两个访问方法都为原子型事务访问。锁被加到所属对象实例级。

6).weak 用于指针变量,比assign多了一个功能,当对象消失后自动把指针变成nil,由于消息发送给空对象表示无操作,这样有效的避免了崩溃(野指针),为了解决原类型与循环引用问题
7).strong 用于指针变量,setter方法对参数进行release旧值再retain新值

120.block 循环应用的问题

并不是所有的 block 都会产生循环引用，例如：
1. 系统的 block 不会导致循坏引用 如 UIview 动画 block 不会产生循环引用是因为这是一个

类方法，控制器没有办法持有一个类，并且动画执行完成了 block 也会被释放。

2. AFN(封装的 AFHTTPRequestOperation) 里面做了__weak 弱引用处理
3. GCD 不会导致循环引用，self 没办法持有 GCD。 但是 gcd 中的对象会延迟释放。

持有 block 的页面先释放也不会导致循环引用 如：block 比持有页面先释放，也不会导致循环引用。如 a 持有 b b 持有 block 在 a 中直接把 b 移除了

121.OC与JS的交互（iOS与H5混编）

在开发过程中，经常会出现需要iOS移动端与H5混编的使用场景。 iOS中加载html网页， 可以使用UIWebView或WKWebView. 本篇博客将介绍两种控件使用过程中如何实现OC与JS的交互。

UIWebView delegate 协议方法

```
//网页即将开始加载
-(BOOL)webView:(UIWebView*)webView
shouldStartLoadWithRequest:(NSURLRequest*)request
navigationType:(UIWebViewNavigationType)navigationType;

//网页开始加载
- (void)webViewDidStartLoad:(UIWebView *)webView;

//网页加载完成
- (void)webViewDidFinishLoad:(UIWebView *)webView;

//网页加载失败
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error;



//UIWebView自带了一个方法，可以直接调用JS代码(转化为string类型的js代码)
- (nullable NSString *)stringByEvaluatingJavaScriptFromString:(NSString *)script;

//例如修改id为‘html’标签内部的text属性
[web
stringByEvaluatingJavaScriptFromString:@"document.getElementById('html').innerText
='修改内容'"];

//也可以执行多行js代码
[web                                stringByEvaluatingJavaScriptFromString:@"vardiv=
document.getElementById('html'); div.innerText = '修改内容'"];
```

利用JavaScriptCore实现交互

JavaScriptCore中类及协议：

```
JSContext：给JavaScript提供运行的上下文环境
JSValue：JavaScript和Objective-C数据和方法的桥梁
JSManagedValue：管理数据和方法的类
```

JSVirtualMachine：处理线程相关，使用较少
JSExport：这是一个协议，如果采用协议的方法交互，自己定义的协议必须遵守此协议
OC中提供了JavaScriptCore 这个库，使得OC与js的交互变得更加方便。
使用方法：
1 加入JavaScriptCore 这个framework
2 引入头文件<JavaScriptCore/JavaScriptCore.h>
3 在VC里面加入一个JSContext属性
@property (strong, nonatomic) JSContext *context;
JSContext是什么那？ 我们看一下api里面的解释

```
@interface
@discussion A JSContext is a JavaScript execution environment. All
JavaScript execution takes place within a context, and all JavaScript values
are tied to a context.
```

大概意思是说：JSContext是一个JS的执行环境，所有的JS执行都发生在一个context里面， 所有的JS value都绑定到context里面

具体使用

```
//初始化context
self.context = [webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"];

//OC调用JS

//（1）例如html的script中一个方法

  function dolike(a,b,c){

  }

//通过OC调用此方法

NSString * method = @"dolike";

JSValue * function = [self.context objectForKeyedSubscript:method];

//这里面的a,b,c就是OC调用JS的时候给JS传的参数
[function callWithArguments:@[a,b,c]];
```

```
//JS调用OC
//例如网页中有个标签,点击button的时候调用Jump方法， 此处3为传入的参数
<button onclick="jump('3')">点我</button>

//当点击网页中的button的时候,触发jump方法， 在OC中用如下代码可以捕捉到jump方法， 并拿到JS给我传的参数 '3'
self.context[@"jump"] = ^(NSString * str){

//此处 str 值为'3'(js调用OC时传给OC的参数)

};
```

说到WKWebView， 首先要说下WKWebView的优势

1 更多的支持HTML5的特性
2 官方宣称的高达60fps的滚动刷新率以及内置手势
3 将UIWebViewDelegate与UIWebView拆分成了14类与3个协议,以前很多不方便实现 的功能得以实现
4 Safari相同的JavaScript引擎
5 占用更少的内存

类：

WKBackForwardList: 之前访问过的 web 页面的列表，可以通过后退和前进动作来访问到。
WKBackForwardListItem: webview 中后退列表里的某一个网页。
WKFrameInfo: 包含一个网页的布局信息。
WKNavigation: 包含一个网页的加载进度信息。
WKNavigationAction: 包含可能让网页导航变化的信息，用于判断是否做出导航变化。
WKNavigationResponse: 包含可能让网页导航变化的返回内容信息，用于判断是否做出导航变化。
WKPreferences: 概括一个 webview 的偏好设置。
WKProcessPool: 表示一个 web 内容加载池。
WKUserContentController: 提供使用 JavaScript post 信息和注射 script 的方法。
WKScriptMessage: 包含网页发出的信息。
WKUserScript: 表示可以被网页接受的用户脚本。
WKWebViewConfiguration: 初始化 webview 的设置。
WKWindowFeatures: 指定加载新网页时的窗口属性。

协议：

WKNavigationDelegate: 提供了追踪主窗口网页加载过程和判断主窗口和子窗口是否进行页面加载新页面的相关方法。
WKScriptMessageHandler: 提供从网页中收消息的回调方法。
WKUIDelegate: 提供用原生控件显示网页的方法回调。

加载方式：

```
//方式一
WKWebView *webView = [[WKWebView alloc] initWithFrame:self.view.bounds];
[webView loadRequest:[NSURLRequest requestWithURL:[NSURL URLWithString:@"http://www.baidu.com"]]];
```

```
 [self.view addSubview:webView];

 //方式二
 WKWebViewConfiguration * configuration = [[WKWebViewConfiguration alloc] init];
 webView = [[WKWebView alloc] initWithFrame:self.view.bounds
configuration:configuration];
 [webView loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.baidu.com"]]];
 [self.view addSubview:webView];
```

协议方法介绍：

```
 #pragma mark - WKNavigationDelegate
 // 页面开始加载时调用
 - (void)webView:(WKWebView *)webView didStartProvisionalNavigation:(WKNavigation
*)navigation{

 }
 // 当内容开始返回时调用
 - (void)webView:(WKWebView *)webView didCommitNavigation:(WKNavigation
*)navigation{

 }
 // 页面加载完成之后调用
 - (void)webView:(WKWebView *)webView didFinishNavigation:(WKNavigation
*)navigation{

 }
 // 页面加载失败时调用
 - (void)webView:(WKWebView *)webView didFailProvisionalNavigation:(WKNavigation
*)navigation{

 }
 // 接收到服务器跳转请求之后调用
 - (void)webView:(WKWebView *)webView
didReceiveServerRedirectForProvisionalNavigation:(WKNavigation *)navigation{

 }
 // 在收到响应后，决定是否跳转
 - (void)webView:(WKWebView *)webView
decidePolicyForNavigationResponse:(WKNavigationResponse *)navigationResponse
decisionHandler:(void (^)(WKNavigationResponsePolicy))decisionHandler{

 NSLog(@"%@",navigationResponse.response.URL.absoluteString);
```

```
//允许跳转
decisionHandler(WKNavigationResponsePolicyAllow);
//不允许跳转
//decisionHandler(WKNavigationResponsePolicyCancel);
}
// 在发送请求之前，决定是否跳转
-                    (void)webView:(WKWebView                    *)webView
decidePolicyForNavigationAction:(WKNavigationAction              *)navigationAction
decisionHandler:(void (^)(WKNavigationActionPolicy))decisionHandler{

NSLog(@"%@",navigationAction.request.URL.absoluteString);
//允许跳转
decisionHandler(WKNavigationActionPolicyAllow);
//不允许跳转
decisionHandler(WKNavigationActionPolicyCancel);
}
#pragma mark - WKUIDelegate
// 创建一个新的WebView
-(WKWebView*)webView:(WKWebView*)webView
createWebViewWithConfiguration:(WKWebViewConfiguration*)configuration
forNavigationAction:(WKNavigationAction*)navigationAction
windowFeatures:(WKWindowFeatures *)windowFeatures{
return [[WKWebView alloc]init];
}
// 输入框
-(void)webView:(WKWebView*)webView
runJavaScriptTextInputPanelWithPrompt:(NSString*)prompt      defaultText:(nullable
NSString *)defaultText initiatedByFrame:(WKFrameInfo *)frame completionHandler:(void
(^)(NSString * __nullable result))completionHandler{
completionHandler(@"http");
}
// 确认框
-(void)webView:(WKWebView *)webView runJavaScriptConfirmPanelWithMessage:(NSString
*)message  initiatedByFrame:(WKFrameInfo  *)frame  completionHandler:(void  (^)(BOOL
result))completionHandler{
completionHandler(YES);
}
// 警告框
- (void)webView:(WKWebView *)webView runJavaScriptAlertPanelWithMessage:(NSString
*)message     initiatedByFrame:(WKFrameInfo     *)frame     completionHandler:(void
(^)(void))completionHandler{
NSLog(@"%@",message);
completionHandler();
}
```

OC与JS的交互
WKWebView
WKWebView的 UIDelegate 提供了三个协议方法， 可以让前端很方便的拦截JS的alert, confirm, prompt方法。除此之外，OC，JS互调可以按照如下方法。
1 OC 调用JS
可以使用webkit这个库

```
- (void)evaluateJavaScript:(NSString *)javaScriptString completionHandler:(void (^_Nullable)(_Nullable id, NSError * _Nullable error))completionHandler;
```

```
//例如OC调用JS的方法 setName

   [webView evaluateJavaScript:@"setname('张三')" completionHandler:nil];
```

//此处 setname为JS定义的方法名， 内部 ‘张三’为传给JS的参数。 如果setname方法需要传入一个json或者array等非字符参数， 需要用format方法将其转为string类型,在调用evaluate方法。例如

```
   NSString * para = [NSString stringWithFormat:@"setname('%@')",json];
```

JS调用OC
此时就要用到WKScriptMessageHandler了

```
 //首先.m中加入属性
 @property (nonatomic ,strong)WKUserContentController * userCC;


//1 遵循WKScriptMessageHandler协议
//2 初始化
   WKWebViewConfiguration * config = [[WKWebViewConfiguration alloc]init];

 self.wkWebViw=[[WKWebView alloc]initWithFrame:self.view.bounds configuration:config];

 [self.wkWebView loadRequest:[NSURLRequest requestWithURL:[NSURL URLWithString:self.webPageUrl]]];

 [self.view addSubview:self.wkWebViw];

 self.userCC = config.userContentController;

 [self.userCC addScriptMessageHandler:self name:@"callOSX"];

 //此处相当于监听了JS中callFunction这个方法
```

```
 [self.userCC addScriptMessageHandler:self name:@"callFunction"];

 //当JS发出callFunction这个方法指令的时候， WKScriptMessageHandler的协议方法中我们就会收到这个消息

 #pragma mark  WKScriptMessageHandler delegate
 -   (void)userContentController:(WKUserContentController   *)userContentController didReceiveScriptMessage:(WKScriptMessage *)message
 {
//这个回调里面， message.name代表方法名（‘本例为 callFunction’）， message.body代表JS给我们传过来的参数

 }

//最后， VC销毁的时候一定要把handler移除
-(void)dealloc
{
    [_userContentController removeScriptMessageHandlerForName:@"callFunction"];
}


//对应的JS代码

<button onclick="buttonClick('温馨提示')">点我</button>

<script>
       function buttonClick(string){
       //JS调用OC, 格式如下
       // (window.webkit.messageHandlers.Method_Name.postMessage(parameterToOC))
              window.webkit.messageHandlers.callFunction.postMessage(string)
        }
</script>
```

123.tableViewCell 使用 SDWebImage 加载图片

对于 cell 里的图片采用异步的方式，加载好后缓存。当图片还没有请求加载时，你可以使用默认图片。

一旦你缓存好图片，使用 cell 的重用机制时就可以从关联好的视图源里以相应的 url 来找到对应的缓存图片，缓存大大节省重复请求图片的耗损。只是你要考虑内存级别的缓存还是磁盘级别的缓存，记得使用完毕清缓存哦！(记得减少内存级别的拷贝)

为了防止图片多次下载，我们需要对图片做缓存，缓存分为内存缓存于沙盒缓存，我们当然两种都要实现。

般情况下在我们会在 cellForRow 方法里面设置 cell 的图片数据源，也就是说如果一个 cell 的 imageview 对象开启了一个下载任务，这个时候该 cell 对象发生了重用，新的 image 数据源会开启另外的一个下载任务，由于他们关联的 imageview 对象实际上是同一个 cell 实例的 imageview 对象，就会发生 2 个下载任务回调给同一个 imageview 对象。这个时候就有必要做一些处理，避免回调发生时，错误的 image 数据源刷新了 UI。

在我们向下滑动 tableview 的时候我们需要手动去取消掉下载操作，当用户停止滑动，再去执行下载操作

如果快速滑下去，然后又滑回来的话，图片是过了一会才显示出来，这是因为快速滑动的时候，旧数据源的下载任务被取消掉了。

异步下载图片我们用的是 NSOperation，并且创建一个全局的 queue 来管理下载图片的操作。

在把图片显示到 Cell 上之前

先判断内存中(images 字典中)有没有图片，

如果有，则取出 url 对应的图片来显示，

如果没有，再去沙盒缓存中查看，当然存到沙盒中都是 NSData。

如果沙盒缓存中有，我们取出对应的数据给 Cell 去显示

如果沙盒中也没有图片，我们先显示占位图片。再创建 operation 去执行下载操作了。

当然在创建 operation 之前，我们要判断这个 operation 操作是否存在

如果没有下载操作，我们才需要真正的去创建 operation 执行下载。

创建好下载操作之后应该把该操作存放到全局队列中去异步执行，同时吧操作放入 operations 字典中记录下来。

下载完成之后：

把下载好的图片放到内存中、同时存到沙盒缓存中

执行完上面的操作之后回到主线程刷新表格，

从 operations 字典中移除下载操作(防止 operations 越来越大，同时保证下载失败后，能重新下载)

124.TableView 为什么会卡？tableView 性能优化

主要由以下原因：

cellForRowAtIndexPath:方法中处理了过多业务

iewCell 的 subview 层级太复杂，做了大量透明处理

cell 的 height 动态变化时计算方式不对

优化核心思想：UITableViewCell 重用机制

简单的理解就是：UITableView 只会创建一屏幕（或一屏幕多一点）的 UITableViewCell，其他都是从中取出来重用的。每当 Cell 滑出屏幕时，就会放入到一个集合（或数组）中（这里就相当于一个重用池），当要显示某一位置的 Cell 时，会先去集合（或数组）中取，如果有，就直接拿来显示；如果没有，才会创建。这样做的好处可想而知，极大的减少了内存的开销。

tableView:cellForRowAtIndexPath:和 tableView:heightForRowAtIndexPath:

UITableView 是继承自 UIScrollView 的，需要先确定它的 contentSize 及每个 Cell 的位置，然后才会把重用的 Cell 放置到对应的位置。所以事实上，UITableView 的回调顺序是先多次调用 tableView:heightForRowAtIndexPath:以确定 contentSize 及 Cell 的位置，然后才会调用 tableView:cellForRowAtIndexPath:，从而来显示在当前屏幕的 Cell。

思路是把赋值和计算布局分离。这样让 tableView:cellForRowAtIndexPath:方法只负责赋值，tableView:heightForRowAtIndexPath:方法只负责计算高度。

可以在获得数据后，直接先根据数据源计算出对应的布局，并缓存到数据源中，这样在 tableView:heightForRowAtIndexPath:方法中就直接返回高度，而不需要每次都计算了。

Tips：

提前计算并缓存好高度（布局），因为 heightForRowAtIndexPath:是调用最频繁的方法；

异步绘制,遇到复杂界面,参考 Facebook 的 AsyncDisplayKit 和 YYAsyncLayer 异步绘制框架；

缓存图片（SDWebImage），提前处理好 UIImageView 图片的尺寸按需加载而不是加载原图；

计算等耗时操作异步处理，处理完再回主线程更新 UI；

图文混排不定高度采用 CoreText 排版，缓存 Cell 高度参考 YYKit；

实现 Cell 的 drawRect:方法直接绘制，减少 UIView，UIImageView，UILabel 等容器的使用。

Bonus：

正确使用 reuseIdentifier 来重用 Cell；

尽量少用或不用透明图层或 View；

如果 Cell 内现实的内容来自 web，使用异步加载，缓存请求结果；

减少 subviews 的数量在 heightForRowAtIndexPath:中尽量不使用 cellForRowAtIndexPath:，如果你需要用到它，只用一次然后缓存结果；

尽量少用 addView 给 Cell 动态添加 View，可以初始化时就添加，然后通过 hide 来控制是否显示；

固定高度不要实现 heightForRowAtIndexPath:方法。

1. cell 的行高不是固定值，需要计算，则要尽可能缓存行高值，避免重复计算行高。因为 heightForRowAtIndexPath:是调用最频繁的方法。

2. 滑动时按需加载，这个在大量图片展示，网络加载的时候很管用！（SDWebImage 已经实现异步加载，配合这条性能杠杠的）。

3. 正确使用 reuseIdentifier 来重用 Cells

4. 尽量少用或不用透明图层

5. 如果 Cell 内现实的内容来自 web，使用异步加载，缓存请求结果

6. 减少 subviews 的数量

7. 在 heightForRowAtIndexPath:中尽量不使用 cellForRowAtIndexPath:，如果你需要用到它，只用一次然后缓存结果

8. 所有的子视图都预先创建，如果不需要显示可以设置 hidden，尽量少动态给 Cell 添加 View

9. 颜色不要使用 alph

10. 栅格化，

11. cell 的 subViews 的各级 opaque 值要设成 YES，尽量不要包含透明的子 View

opaque 用于辅助绘图系统，表示 UIView 是否透明。在不透明的情况下，渲染视图时需要快速地渲染，以提高性能。渲染最慢的操作之一是混合(blending)。提高性能的方法是减少混合操作的次数，其实就是 GPU 的不合理使用，这是硬件来完成的（混合操作由 GPU 来执行，因为这个硬件就是用来做混合操作的，当然不只是混合）。 优化混合操作的关键点是在平衡 CPU 和 GPU 的负载。还有就是 cell 的 layer 的 shouldRasterize 要设成 YES。

12. cell 异步加载图片以及缓存

13. 异步绘制

（1）在绘制字符串时，尽可能使用 drawAtPoint: withFont:，而不要使用更复杂的 drawAtPoint:(CGPoint)point forWidth:(CGFloat)width withFont:(UIFont *)font lineBreakMode:(UILineBreakMode)lineBreakMode; 如果要绘制过长的字符串，建议自己先截断，然后使用 drawAtPoint: withFont:方法绘制。

（2）在绘制图片时，尽量使用 drawAtPoint，而不要使用 drawInRect。drawInRect 如果在绘制过程中对图片进行放缩，会特别消耗 CPU。

（3）其实，最快的绘制就是你不要做任何绘制。有时通过 UIGraphicsBeginImageContextWithOptions() 或者 CGBitmapContextCeate() 创建位图会显得更有意义，从位图上面抓取图像，并设置为 CALayer 的内容。

如果你必须实现 -drawRect:，并且你必须绘制大量的东西，这将占用时间。

（4）如果绘制 cell 过程中，需要下载 cell 中的图片，建议在绘制 cell 一段时间后再开启图片下载任务。譬如先画一个默认图片，然后在 0.5S 后开始下载本 cell 的图片。

（5）即使下载 cell 图片是在子线程中进行，在绘制 cell 过程中，也不能开启过多的子线程。最好只有一个下载图片的子线程在活动。否则也会影响 UITableViewCell 的绘制，因而影响了 UITableViewCell 的滑动速度。(建议结合使用 NSOpeartion 和 NSOperationQueue 来下载图片，如果想尽可能找的下载图片，可以把[self.queuesetMaxConcurrentOperationCount:4];)

（6）最好自己写一个 cache，用来缓存 UITableView 中的 UITableViewCell，这样在整个 UITableView 的生命周期里，一个 cell 只需绘制一次，并且如果发生内存不足，也可以有效的释放掉缓存的 cell。

14.不要将 tableview 的背景颜色设置成一个图片。这回严重影响 UITableView 的滑动速度。在限时免费搜索里，我曾经翻过一个错误：self.tableView_.backgroundColor = [UIColorcolorWithPatternImage:[UIImageimageNamed:@"background.png"]]; 通过这种方式设置 UITableView 的背景颜色会严重影响 UTIableView 的滑动流畅性。修改成 self.tableView_.backgroundColor = [UIColor clearColor];之后，fps 从 43 上升到 60 左右。滑动比较流畅。

如果做到以上 14 点，则 UITableView 滑动的 fps 可以达到 60 fps。滑动非常顺畅

125.tableView性能优化2

我们经常在注意cellForRowAtIndexPath：中为每一个cell绑定数据，实际上在调用cellForRowAtIndexPath：的时候cell还没有被显示出来，为了提高效率我们应该把数据绑定的操作放在cell显示出来后再执行，可以在tableView：willDisplayCell：forRowAtIndexPath：（以后简称willDisplayCell）方法中绑定数据。
注意willDisplayCell在cell 在tableview展示之前就会调用，此时cell实例已经生成，所以不能更改cell的结构，只能是改动cell上的UI的一些属性（例如label的内容等）。

2、cell高度的计算

这边我们分为两种cell，一种是定高的cell，另外一种是动态高度的cell。
（1）定高的cell，应该采用如下方式：
self.tableView.rowHeight = 88;
这个方法指定了所有cell高度都是88的tableview，rowHeight默认的值是44，所以一个空的TableView会显示成这个样子。对于定高cell，直接采用上面方式给定高度，不需要实现tableView:heightForRowAtIndexPath:以节省不必要的计算和开销。
（2）动态高度的cell
我们需要实现它的代理，来给出高度：

```
-(CGFloat)tableView:(UITableView *)tableViewheightForRowAtIndexPath:(NSIndexPath *)indexPath{

        // return xxx
}
```

这个代理方法实现后，上面的rowHeight的设置将会变成无效。在这个方法中，我们需要提高cell高度的计算效率，来节省时间。
自从iOS8之后有了self-sizing cell的概念，cell可以自己算出高度，使用self-sizing cell需要满足以下三个条件：
（1）使用Autolayout进行UI布局约束（要求cell.contentView的四条边都与内部元素有约束关系）。
（2）指定TableView的estimatedRowHeight属性的默认值。
（3）指定TableView的rowHeight属性为UITableViewAutomaticDimension。

```
1 - (void)viewDidload {
2     self.myTableView.estimatedRowHeight = 44.0;
3     self.myTableView.rowHeight = UITableViewAutomaticDime
4 }
```

除了提高cell高度的计算效率之外，对于已经计算出的高度，我们需要进行缓存，对于已经计算过的高度，没有必要进行计算第二次。

3、渲染

为了保证TableView的流畅，当快速滑动的时候，cell必须被快速的渲染出来。所以cell渲染的速度必须快。如何提高cell的渲染速度呢？
（1）当有图像时，预渲染图像，在bitmap context先将其画一遍，导出成UIImage对象，然后再绘制到屏幕，这会大大提高渲染速度。具体内容可以自行查找“利用预渲染加速显示iOS图像”相关资料。
（2）渲染最好时的操作之一就是混合(blending)了，所以我们不要使用透明背景，将cell的

opaque值设为Yes，背景色不要使用clearColor，尽量不要使用阴影渐变等
（3）由于混合操作是使用GPU来执行，我们可以用CPU来渲染，这样混合操作就不再执行。可以在UIView的drawRect方法中自定义绘制。

7、异步化UI，不要阻塞主线程
我们时常会看到这样一个现象，就是加载时整个页面卡住不动，怎么点都没用，仿佛死机了一般。原因是主线程被阻塞了。所以对于网路数据的请求或者图片的加载，我们可以开启多线程，将耗时操作放到子线程中进行，异步化操作。这个或许每个iOS开发者都知道的知识，不必多讲。

8、滑动时按需加载对应的内容
如果目标行与当前行相差超过指定行数，只在目标滚动范围的前后指定3行加载。

```
-(void)scrollViewWillEndDragging:(UIScrollView *)scrollViewwithVelocity:(CGPoint)velocitytargetContentOffset:(inoutCGPoint *)targetContentOffset{

        NSIndexPath *ip=[selfindexPathForRowAtPoint:CGPointMake(0,targetContentOffset->y)];

        NSIndexPath *cip=[[selfindexPathsForVisibleRows]firstObject];

        NSIntegerskipCount=8;

        if(labs(cip.row-ip.row)>skipCount){

                NSArray *temp=[selfindexPathsForRowsInRect:CGRectMake(0,targetContentOffset->y,self.width,self.height)];

                NSMutableArray *arr=[NSMutableArrayarrayWithArray:temp];

                if(velocity.y<0){

                        NSIndexPath *indexPath=[templastObject];

                        if(indexPath.row+33){

                                [arraddObject:[NSIndexPathindexPathForRow:indexPath.row-3inSection:0]];

                                [arraddObject:[NSIndexPathindexPathForRow:indexPath.row-2inSection:0]];

          [arraddObject:[NSIndexPathindexPathForRow:indexPath.row-1inSection:0]];

                        }
```

```
                }
                [needLoadArraddObjectsFromArray:arr];
        }
}
```

记得在tableView:cellForRowAtIndexPath:方法中加入判断：

```
if(needLoadArr.count>0&&[needLoadArrindexOfObject:indexPath]==NSNotFound){
        [cellclear];
        return;
}
```

滑动很快时，只加载目标范围内的cell，这样按需加载（配合SDWebImage），极大提高流畅度。

9、最后想谈下离屏渲染的问题：

9.1、下面的情况或操作会引发离屏渲染：

- 为图层设置遮罩（layer.mask）
- 将图层的layer.masksToBounds / view.clipsToBounds属性设置为true
- 将图层layer.allowsGroupOpacity属性设置为YES和layer.opacity小于1.0
- 为图层设置阴影（layer.shadow *）。
- 为图层设置layer.shouldRasterize=true
- 具有layer.cornerRadius，layer.edgeAntialiasingMask，layer.allowsEdgeAntialiasing的图层
- 文本（任何种类，包括UILabel，CATextLayer，Core Text等）。
- 使用CGContext在drawRect :方法中绘制大部分情况下会导致离屏渲染，甚至仅仅是一个空的实现

9.2、优化方案

官方对离屏渲染产生性能问题也进行了优化：

iOS 9.0 之前UIimageView跟UIButton设置圆角都会触发离屏渲染。

iOS 9.0 之后UIButton设置圆角会触发离屏渲染，而UIImageView里png图片设置圆角不会触发离屏渲染了，如果设置其他阴影效果之类的还是会触发离屏渲染的。

（1）圆角优化

在APP开发中，圆角图片还是经常出现的。如果一个界面中只有少量圆角图片或许对性能没有非常大的影响，但是当圆角图片比较多的时候就会APP性能产生明显的影响。

我们设置圆角一般通过如下方式：

```
imageView.layer.cornerRadius=CGFloat(10);

imageView.layer.masksToBounds=YES;
```

这样处理的渲染机制是GPU在当前屏幕缓冲区外新开辟一个渲染缓冲区进行工作，也就是离屏渲染，这会给我们带来额外的性能损耗，如果这样的圆角操作达到一定数量，会触发缓冲区的频繁合并和上下文的的频繁切换，性能的代价会宏观地表现在用户体验上——掉帧。

优化方案1：使用贝塞尔曲线UIBezierPath和Core Graphics框架画出一个圆角

```objc
UIImageView *imageView = [[UIImageView alloc]initWithFrame:CGRectMake(100, 100, 100, 100)
imageView.image = [UIImage imageNamed:@"myImg"];
//开始对imageView进行画图
UIGraphicsBeginImageContextWithOptions(imageView.bounds.size, NO, 1.0);
//使用贝塞尔曲线画出一个圆形图
[[UIBezierPath bezierPathWithRoundedRect:imageView.bounds cornerRadius:imageView.frame.si
[imageView drawRect:imageView.bounds];
imageView.image = UIGraphicsGetImageFromCurrentImageContext();
//结束画图
UIGraphicsEndImageContext();
[self.view addSubview:imageView];
```

优化方案2：使用CAShapeLayer和UIBezierPath设置圆角

```objc
UIImageView
*imageView=[[UIImageViewalloc]initWithFrame:CGRectMake(100,100,100,100)];

imageView.image=[UIImageimageNamed:@"myImg"];

UIBezierPath
*maskPath=[UIBezierPathbezierPathWithRoundedRect:imageView.boundsbyRoundingCorne
rs:UIRectCornerAllCornerscornerRadii:imageView.bounds.size];

CAShapeLayer *maskLayer=[[CAShapeLayeralloc]init];

//设置大小

maskLayer.frame=imageView.bounds;

//设置图形样子

maskLayer.path=maskPath.CGPath;

imageView.layer.mask=maskLayer;

[self.viewaddSubview:imageView];
```

对于方案2需要解释的是：

- CAShapeLayer继承于CALayer,可以使用CALayer的所有属性值；
- CAShapeLayer需要贝塞尔曲线配合使用才有意义（也就是说才有效果）
- 使用CAShapeLayer(属于CoreAnimation)与贝塞尔曲线可以实现不在view的drawRect（继承于CoreGraphics走的是CPU,消耗的性能较大）方法中画出一些想要的图形

- CAShapeLayer动画渲染直接提交到手机的GPU当中，相较于view的drawRect方法使用CPU渲染而言，其效率极高，能大大优化内存使用情况。

总的来说就是用CAShapeLayer的内存消耗少，渲染速度快，建议使用优化方案2。

（2）shadow优化

对于shadow，如果图层是个简单的几何图形或者圆角图形，我们可以通过设置shadowPath来优化性能，能大幅提高性能。示例如下：

```
imageView.layer.shadowColor=[UIColorgrayColor].CGColor;

imageView.layer.shadowOpacity=1.0;

imageView.layer.shadowRadius=2.0;

UIBezierPath *path=[UIBezierPathbezierPathWithRect:imageView.frame];

imageView.layer.shadowPath=path.CGPath;
```

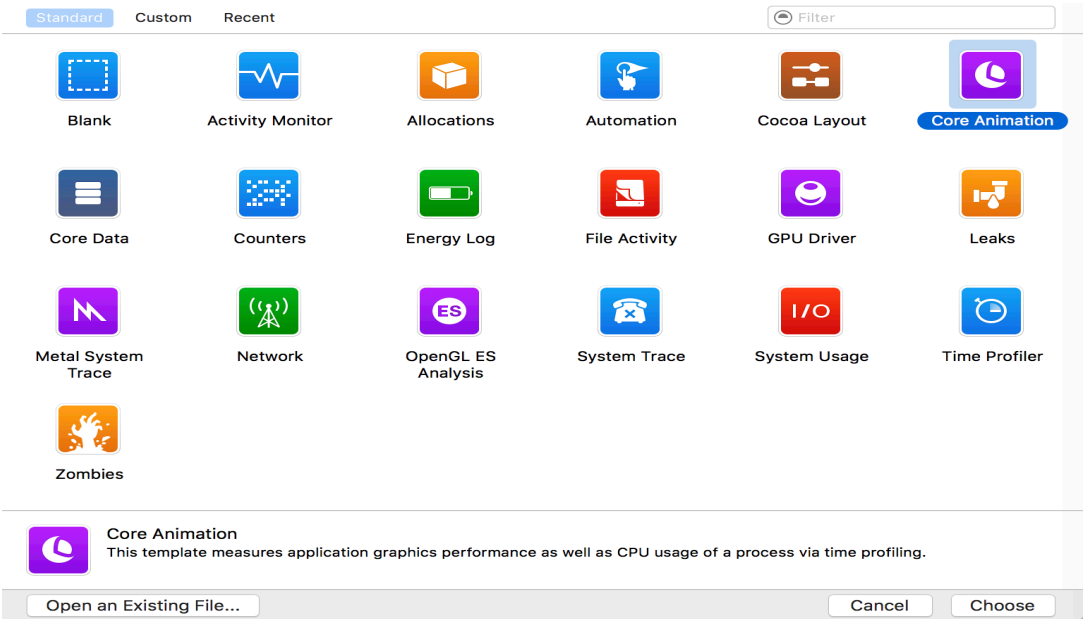
我们还可以通过设置shouldRasterize属性值为YES来强制开启离屏渲染。其实就是光栅化（Rasterization）。既然离屏渲染这么不好，为什么我们还要强制开启呢？当一个图像混合了多个图层，每次移动时，每一帧都要重新合成这些图层，十分消耗性能。当我们开启光栅化后，会在首次产生一个位图缓存，当再次使用时候就会复用这个缓存。但是如果图层发生改变的时候就会重新产生位图缓存。所以这个功能一般不能用于UITableViewCell中，cell的复用反而降低了性能。最好用于图层较多的静态内容的图形。而且产生的位图缓存的大小是有限制的，一般是2.5个屏幕尺寸。在100ms之内不使用这个缓存，缓存也会被删除。所以我们要根据使用场景而定。

（3）其他的一些优化建议

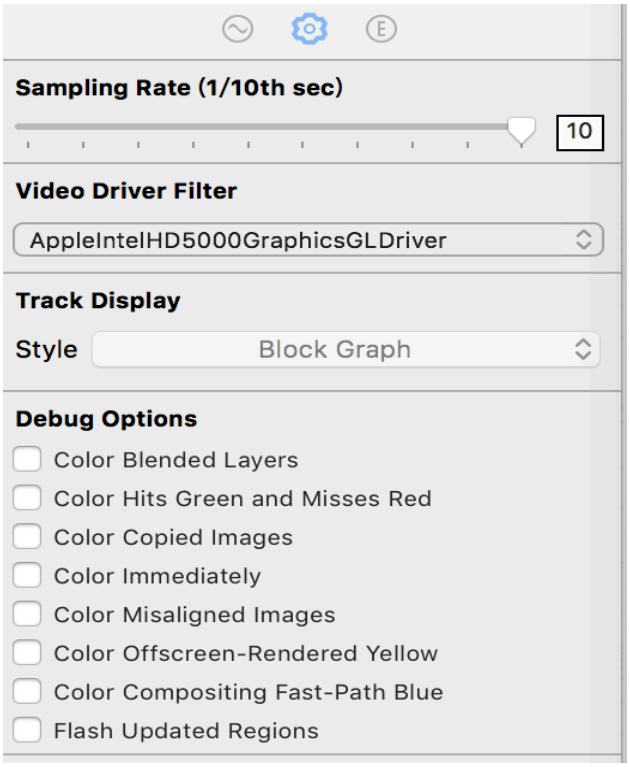
- 当我们需要圆角效果时，可以使用一张中间透明图片蒙上去
- 使用ShadowPath指定layer阴影效果路径
- 使用异步进行layer渲染（Facebook开源的异步绘制框架AsyncDisplayKit）
- 设置layer的opaque值为YES，减少复杂图层合成
- 尽量使用不包含透明（alpha）通道的图片资源
- 尽量设置layer的大小值为整形值
- 直接让美工把图片切成圆角进行显示，这是效率最高的一种方案
- 很多情况下用户上传图片进行显示，可以让服务端处理圆角
- 使用代码手动生成圆角Image设置到要显示的View上，利用UIBezierPath（CoreGraphics框架）画出来圆角图片

（4）Core Animation工具检测离屏渲染

对于离屏渲染的检测，苹果为我们提供了一个测试工具Core Animation。可以在Xcode->Open Develeper Tools->Instruments中找到，如下图：



Core Animation工具用来监测Core Animation性能，提供可见的FPS值，并且提供几个选项来测量渲染性能。如下图：



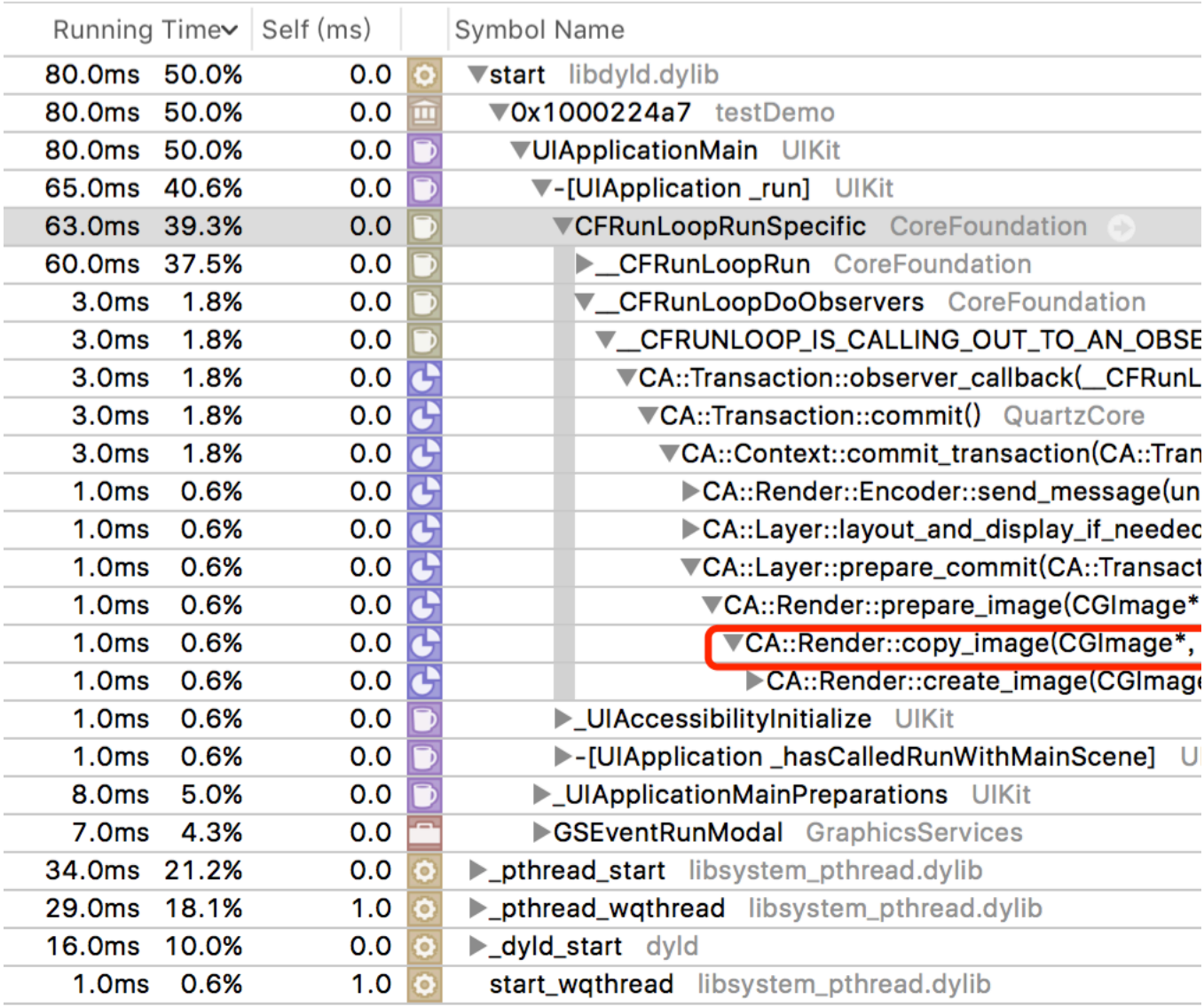
下面我们来说明每个选项的功能：

Color Blended Layers：这个选项如果勾选，你能看到哪个layer是透明的，GPU正在做混合计算。显示红色的就是透明的，绿色就是不透明的。

Color Hits Green and Misses Red：如果勾选这个选项，且当我们代码中有设置shouldRasterize为YES，那么红色代表没有复用离屏渲染的缓存，绿色则表示复用了缓存。我们当然希望能够复用。

Color Copied Images：按照官方的说法，当图片的颜色格式GPU不支持的时候，Core Animation会

拷贝一份数据让CPU进行转化。例如从网络上下载了TIFF格式的图片，则需要CPU进行转化，这个区域会显示成蓝色。还有一种情况会触发Core Animation的copy方法，就是字节不对齐的时候。如下图：



Color Immediately：默认情况下Core Animation工具以每毫秒10次的频率更新图层调试颜色，如果勾选这个选项则移除10ms的延迟。对某些情况需要这样，但是有可能影响正常帧数的测试。
Color Misaligned Images：勾选此项，如果图片需要缩放则标记为黄色，如果没有像素对齐则标记为紫色。像素对齐我们已经在上面有所介绍。
Color Offscreen-Rendered Yellow：用来检测离屏渲染的，如果显示黄色，表示有离屏渲染。当然还要结合Color Hits Green and Misses Red来看，是否复用了缓存。
Color OpenGL Fast Path Blue：这个选项对那些使用OpenGL的图层才有用，像是GLKView或者CAEAGLLayer，如果不显示蓝色则表示使用了CPU渲染，绘制在了屏幕外，显示蓝色表示正常。
Flash Updated Regions：当对图层重绘的时候回显示黄色，如果频繁发生则会影响性能。可以用增加缓存来增强性能。
以上就是本人的一些总结，当然对于UITableView的性能优化，网上有很多相关的资料。如果有什么不同的观点，欢迎大家补充。

126.注册成为环信开发者

2.在开发者后台创建 APP，获取 key

3.下载 SDK,获取 DEMO

4.集成 SDK

如果项目中使用-ObjC 有冲突,可以添加-force_load 来解决

SDK 不支持 bitcode,向 Build Settings → Linking → Enable Bitcode 中设置 NO。

5.SDK 同步/异步方法区分:

SDK 中，大部分与网络有关的操作，提供的是同步方法(注:同步方法会阻塞主线程,需要用户自己创建异步线程执行;带有 async 的方法为异步方法)

6.初始化 SDK

AppKey: 区别 app 的标识，开发者注册及管理后台

apnsCertName: iOS 中推送证书名称。制作与上传推送证书

环信为 im 部分提供了 apns 推送功能，如果您要使用，请跳转到 apns 离线推送

7.注册

开发注册和授权注册

只有开放注册时，才可以客户端注册，开放注册主要是测试使用。

授权注册的流程应该是您服务器通过环信提供的 rest api 注册，之后保存到您的服务器或返回给客户端。

8.登录

9.自动登录

自动登录在以下几种情况下会被取消

1）用户调用了 SDK 的登出动作;

2）用户在别的设备上更改了密码，导致此设备上自动登陆失败;

3）用户的账号被从服务器端删除;

4）用户从另一个设备登录，把当前设备上登陆的用户踢出.

在您调用登录方法前，应该先判断是否设置了自动登录，如果设置了，则不需要您再调用

10.重连

当掉线时，IOS SDK 会自动重连，只需要监听重连相关的回调，无需进行任何操作。

11.退出登录

（1）主动退出登录：调用 SDK 的退出接口；

（2）被动退出登录：正在登陆的账号在另一台设备上登陆； 2、 正在登陆的账号被从服务器端删除。

12.好友管理

环信不是好友也可以聊天，不推荐使用环信的好友机制。如果你有自己的服务器或好友关系，请自己维护好友关系。

（1） 从服务器获取所有的好友

（2）从数据库获取所有的好友

13.添加好友

如果您已经发过，并且对方没有处理，您将不能再次发送

14.实时通话管理

发起实时通话

被叫方同意实时通话

结束实时通话

127.蓝牙

在 iOS 中，蓝牙是基于 4.0 标准的，设备间低功耗通信。

其中 Peripheral 外设相当于 Socket 编程中的 Server 服务端，Central 中心相当于 Client 客户端(ps 吐槽下，Central 中心，作为服务端，不更适合吗！)

本地中心 -> 远程外设

本地外设 -> 远程中心

建立中心角色 —> 扫描外设（discover）—> 发现外设后连接外设(connect) —> 扫描外设中的服务和特征(discover) —> 与外设做数据交互(explore and interact) —> 断开连接(disconnect)。

1.建立中心角色

上面的 delegate 为 CBCentralManagerDelegate，后续蓝牙相关的回调都会在此。queue 代表蓝牙在哪个队列里面操作，如果传入 nil 默认为主队列，值得注意的是后续的回调也是在传入的队列中调用的，所以如果传入的是非主线程的队列，在 delegate 中需要操作 UI 时需要手动切换到主线程

CBCentralManager 对象创建后会回调到 centralManagerDidUpdateState 方法来检测蓝牙可用状态，这时我们可以提醒用户设备是否支持蓝牙，是否打开了蓝牙。

2.扫描外设

如果 serviceUUIDS 为 nil 则会扫描周围所有的设外设，反之只会扫描 UUID 匹配的外设。CBCentralManagerScanOptionAllowDuplicatesKey 默认为 false，此次扫描中发现过设备则跳过不回调，我们这里传入 true，因为下面做外设掉线的处理时需要用到

传入的 serviceUUIDS 数组元素为 CBUUID 类型，千万不要传入 String，后面的操作也是如此，不然会碰到很多奇葩问题

发现外设后会回调到
centralManager(central:,didDiscoverPeripheral:,advertisementData:, RSSI:)

其中，perpheral 则代表着外设，我们需要保存起来，后续的对外设的操作都是基于 perpheral 对象的

3.连接外设

传入上面保存的外设对象，如果连接失败后会回调到 centralManager(central:, didFailToConnectPeripheral:, error:)，

连接成功后会回调到 centralManager(central:didConnectPeripheral:)，这个时候我们只是连接上外设而已，还需要发现外设中的服务与特征

4.发现服务与特征

5.发送数据

6.读取数据

7.断开连接

在蓝牙交互的二种角色中，通常 APP 端扮演中央 Central 的角色，设备扮演外设 Peripheral 的角色

创建 CBCentralManager 对象时传入的 queue 决定了后续 CBCentralManagerDelegate、CBPeripheralDelegate 等回调的所在线程

一个外设设备可包含一个或多个服务，一个服务可包含一个或多个特征，读写操作最终是针对特征。

蓝牙的缓冲大小只有 20bytes，在发送数据时最多只能发送 20bytes，所以得分多次发送，数据的一体性可以用 EOM 标识符表标识

127. 什么是NSManagedObject模型?

答：NSManagedObject是NSObject的子类 ，也是coredata的重要组成部分，它是一个通用的类，实现了core data 模型层所需的基本功能，用户可通过子类化NSManagedObject，建立自己的数据模型。

128. 什么是NSManagedobjectContext?

答：NSManagedobjectContext对象负责应用和数据库之间的交互。

115. ios 平台怎么做数据的存储的?coredata 和sqlite有无必然联系？coredata是一个关系型数据库吗？

答：iOS 中可以有四种持久化数据的方式：

1）存入到NSUserDefaults属性列表(系统plist文件中)

2).通过web服务，保存在服务器上

3).通过NSCoder固化机制，将对象保存在文件中

4).通过SQlite或CoreData保存在文件数据库中

coredata是苹果提供一套数据保存框架，其基于SQlite

core data 可以使你以图形界面的方式快速的定义 app 的数据模型，同时在你的代码中容易获取到它。 coredata 提供了基础结构去处理常用的功能，例如保存，恢复，撤销和重做，允许你在 app 中继续创建新的任务。在使用 core data 的时候，你不用安装额外的数据库系统，因为 core data 使用内置的 sqlite 数据库。 core data 将你 app 的模型层放入到一组定义在内存中的数据对象。 coredata 会追踪这些对象的改变，同时可以根据需要做相反的改变，例如用户执行撤销命令。当 core data 在对你 app 数据的改变进行保存的时候，core data 会把这些数据归档，并永久性保存。 mac os x 中sqlite 库，它是一个轻量级功能强大的关系数据引擎，也很容易嵌入到应用程序。可以在多个平台使用， sqlite 是一个轻量级的嵌入式 sql 数据库编程。与 core data 框架不同的是， sqlite 是使用程序式的， sql 的主要的 API 来直接操作数据表。 Core Data 不是一个关系型数据库，也不是关系型数据库管理系统 (RDBMS) 。虽然 Core Dta 支持SQLite 作为一种存储类型，但它不能使用任意的 SQLite 数据库。 Core Data 在使用的过程种自己创建这个数据库。 Core Data 支持对一、对多的关系。

二十：CoreData&SQLite3

首先，coredata 和 sqlite 的概念不同，core 为对象周期管理，而 sqlite 为 dbms。

使用方便性。实际上，一个成熟的工程中一定是对数据持久化进行了封装的，因此底层使用的到底是 core data 还是 sqlite，不应该被业务逻辑开发者关心。因此，即使习惯写 SQL 查询的人，也应该避免在业务逻辑中直接编写 SQL 语句。

存储性能，在写入性能上，因为都是使用的 sqlite 格式作为磁盘存储格式，因此其性能是一样的，如果你觉得用 core data 写的慢，很可能是你用 sqlite 的时候写的每条数据的内容没有 core data 时多，或者是你批量写入的时候每写入一条就调用了一次 save。

查询性能，core data 因为要兼容多种后端格式，因此查询时，其可用的语句比直接使用 sqlite 少，因此有些 fetch 实际上不是在 sqlite 中执行的。但这样未必 会降低查询效率。因

为 iPhone 的 flash memory 速度还是很快的。我的经验是大部分时候，在内存不是很紧张时，直接 fetch 一个 entity 的所有数据然后在内存中做 filter 往往比使 用 predicate 在 fetch 时过滤更快。如果你觉的查询慢，很可能是查询方式有问题，可以把 core data 的 debug 模式打开，看一下到底执行了多少 SQL 语句，相信其中大部分是可以通过改写 core data 的调用方式避免的。

core data 的一个比较大的痛点是多人合作开发的时候，管理 coredata 的模型需要很小心，尤其是合并的时候，他的 data model 是 XML 格式的，手动 resolve 比较烦心。

core data 还有其他 sql 所不具备的优点，比如对 undo 的支持，多个 context 实现 sketchbook 类似的功能。为 ManagedObject 优化的 row cash 等。

另外core data是支持多线程的，但需要thread confinement的方式实现,使用了多线程之后可以最大化的防止阻塞主线程

129.数据存储

1.数据存储技术
1> 数据存储的几种方式
XML属性列表（plist）归档
Preference(偏好设置)
NSKeyedArchiver归档(NSCoding)
SQLite3
Core Data
2> 各自特点
Plist：
属性列表是一种XML格式的文件，拓展名为plist
如果对象是NSString、NSDictionary、NSArray、NSData、NSNumber等类型，就可以使用writeToFile:atomically:方法直接将对象写到属性列表文件中
将一个NSDictionary对象归档到一个plist属性列表中

```
// 将数据封装成字典
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
[dict setObject:@"母鸡" forKey:@"name"];
// 将字典持久化到Documents/stu.plist文件中
[dict writeToFile:path atomically:YES];
```

面试考点：
1. plist的根节点 只能是NSDictionary和NSArray，所以存储内容必须转为对象类型
2. 使用场景 功能动态更新 应用级别数据更新 XML的替代品

偏好设置：
每个应用都有个NSUserDefaults实例，通过它来存取偏好设置
比如，保存用户名、字体大小、是否自动登录

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
[defaults setObject:@"itcast" forKey:@"username"];
```

```
[defaults setFloat:18.0f forKey:@"text_size"];
[defaults setBool:YES forKey:@"auto_login"];
```

面试考点：
1.使用场景 保存应用信息
2.特点　不会自动删除，itune同步，不适合存大数据
3.使用单例模式、
4.直接存取结构体，基本数据类型，无需转换
5.即时操作注意同步

归档：
如果对象是NSString、NSDictionary、NSArray、NSData、NSNumber等类型，可以直接用NSKeyedArchiver进行归档和恢复
不是所有的对象都可以直接用这种方法进行归档，只有遵守了NSCoding协议的对象才可以
NSCoding协议有2个方法：
encodeWithCoder:
每次归档对象时，都会调用这个方法。一般在这个方法里面指定如何归档对象中的每个实例变量，可以使用encodeObject:forKey:方法归档实例变量
initWithCoder:
每次从文件中恢复(解码)对象时，都会调用这个方法。一般在这个方法里面指定如何解码文件中的数据为对象的实例变量，可以使用decodeObject:forKey方法解码实例变量
归档一个NSArray对象到Documents/array.archive

```
NSArray *array = [NSArray arrayWithObjects:@”a”,@”b”,nil];
[NSKeyedArchiver archiveRootObject:array toFile:path];
```

使用archiveRootObject:toFile:方法可以将一个对象直接写入到一个文件中，但有时候可能想将多个对象写入到同一个文件中，那么就要使用NSData来进行归档对象
归档（编码）

```
// 新建一块可变数据区
NSMutableData *data = [NSMutableData data];
// 将数据区连接到一个NSKeyedArchiver对象
NSKeyedArchiver *archiver = [[[NSKeyedArchiver alloc] initForWritingWithMutableData:data] autorelease];
// 开始存档对象，存档的数据都会存储到NSMutableData中
[archiver encodeObject:person1 forKey:@"person1"];
[archiver encodeObject:person2 forKey:@"person2"];
// 存档完毕(一定要调用这个方法)
[archiver finishEncoding];
// 将存档的数据写入文件
[data writeToFile:path atomically:YES];
```

1　恢复（解码）

```
// 从文件中读取数据
NSData *data = [NSData dataWithContentsOfFile:path];
```

```
// 根据数据，解析成一个NSKeyedUnarchiver对象
NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] initForReadingWithData:data];
Person *person1 = [unarchiver decodeObjectForKey:@"person1"];
Person *person2 = [unarchiver decodeObjectForKey:@"person2"];
// 恢复完毕
[unarchiver finishDecoding];
```

利用归档实现深复制

```
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:person1];
// 解析data，生成一个新的Person对象
Student *person2 = [NSKeyedUnarchiver unarchiveObjectWithData:data];
```

面试考点：

1. 特点： 存入Document，itune同步，不会自动删除，可存放大型用户数据
2. 使用场景： 用户产生的数据，如游戏，操作记录等等
3. 可保存自定义对象，需要遵守NSCoding协议，实现对应的encodeWithCoder initWithCoder 方法
4. 和NSData的配合

4.1> 多对象单目录存储

4.2> 字典/数组内容的深拷贝

5. 不能直接存基本类型和结构体，需要转成对象 NSValue NSNumber

2> 沙盒目录结构

2.1> Library Caches Preferences

2.2> Documents

2.3> tmp

3> 如何读取沙盒中plist的内容

1> 3.1> 读取沙盒并拼接plist的文件路径

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"app.plist" ofType:nil];
```

3.2> 根据plist根节点类型读取plist文件

```
NSArray *apps = [NSArray arrayWithContentsOfFile:path];
```

2.数据库技术（SQLite&CoreData）

1> SQLite和CoreData的区别

1.1> CoreData可以在一个对象更新时,其关联的对象也会随着更新,相当于你更新一张表时,其关联的其他表的也回随着更新

1.2> CoreData提供更简单的性能管理机制,可以限制查询记录的总数,这个类会自动更新其缓存

1.3> 多表查询方面,CoreData没有SQL直观,没有类似外连接,左连接等操作.

iOS App 升级安装 - CoreData 数据库升级

1.选中你的 mydata.xcdatamodeld 文件，选择菜单 editor->Add Model Version 比如取名：mydata2.xcdatamodel

2.设置当前版本

3..修改新数据模型 mydata2，在新的文件上添加字段及表

4.删除原来的类文件，重新生成下类。在 appdelegate 中添加 *optionsDictionary，原来 options:nil 改成 options:optionsDictionary

5.重新编译下程序。

增加模型版本——->选择最新的版本——->选择表，添加字段——>删除之前的类，重新生成。

130. OC堆和栈的区别？

答：管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。

申请大小：

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS下，栈的大小是2M（也有的说是1M,总之是一个编译时就确定的常数),如果申请的空间超过栈的剩余空间时,将提示overflow。因 此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出

分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的。

堆栈空间分配区别：

1、栈（操作系统）：由操作系统自动分配释放,存放函数的参数值,局部变量的值等。其操作方式类似于数据结构中的栈；

2、堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收，分配方式倒是类似于链表。

堆栈缓存方式区别：

1、栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放

2、堆是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

堆栈数据结构区别：

堆（数据结构）：堆可以被看成是一棵树，如：堆排序；

栈（数据结构）：一种先进后出的数据结构。

内存其他补充：

全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放

文字常量区—常量字符串就是放在这里的。 程序结束后由系统释放

程序代码区—存放函数体的二进制代码。

131：内存泄露&内存溢出

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现 out of memory；比如申请了一个 integer,但给它存了 long 才能存下的数，那就是内存溢出。

内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

memory leak 会最终会导致 out of memory!

内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出

132.iOS内存区域

(1)栈区:由编译器自动分配释放,存放函数的参数值,局部变量的值等.其操作方式类似于数据结构中的栈.

(2)堆区

一般由程序员分配释放,若程序员不释放,程序结束时由系统回收

(3)全局区(静态区)

全局变量和静态变量的存储是放在一块的,初始化的全局变量和静态变量在一块区域,未初始化的全局变量和未初始化的静态变量相邻的另一块区域.

全局区分为未初始化全局区: .bss段 和初始化全局区: data段.

(4)常量区

常量字符串就是放在常量区

(5)代码区

存放函数体的二进制代码

注释：创建字符串的内存空间　堆　常量区

134.你是如何优化内存管理

1> 使用ARC

2> 延迟加载 懒加载

3> 重用 在正确的地方使用reuseIndentifier

4> 缓存 NSCache 保存计算数据

5> 处理内存警告　移除对缓存，图片 object 和其他一些可以重创建的 objects 的强引用

5.1>app delegate 中使用 `applicationDidReceiveMemoryWarning:` 的方法

5.2>自定义UIViewController 的子类(subclass)中覆盖`didReceiveMemoryWarning`

5.3>在自定义类中注册并接收 UIApplicationDidReceiveMemoryWarningNotification 的通知

6>重用大开销对象NSDateFormatter和NSCalendar 懒加载/单例　_formatter.dateFormat = @"EEE MMM dd HH:mm:ss Z yyyy”; 设置和创建速度一样慢

7>自动释放池　手动添加自动释放池

8>是否缓存图片 imageNamed　imageWithContentOfFile

9>混编

10>循环引用　delegate block nstimer

11>移除 kvo　nsnotificationcenter　并未强引用,只记录内存地址,野指针报错 UIViewController自动移除　一般在dealloc中

13> performselector 延迟操作[NSObject cancelPreviousPerformRequestsWithTarget:self]

135.autorelease的使用

1> 工厂方法为什么不释放对象

很多类方法为了在代码块结束时引用的对象不会因无强引用而被释放内存采用自动释放的方式,当其最近的自动释放池释放时该对象才会释放.

2> ARC下autorelease的使用场景

ARC中手动添加autoreleasepool可用于提前释放使用自动释放策略的对象,防止大量自动释放的对象堆积造成内存峰值过高.

3> 自动释放池如何工作

自动释放池时栈结构,每个线程的runloop运行时都会自动创建自动释放池,程序员可以代码手动创建自动释放池,自动释放的对象会被添加到最近的(栈顶)自动释放池中,系统自动创建的自动释放池在每个运行循环结束时销毁释放池并给池中所有对象发release消息,手动创建释放池在所在代码块结束时销毁释放池并发消息统一release

136.ARC和MRC的混用

1.MRC—>ARC

把MRC的代码转换成ARC的代码，删除内存管理操作（手动）
xcode提供了自动将MRC转换成ARC的功能，操作菜单栏edit -> Refacotor（重构） -> Convert to Objective-C ARC

2.ARC-->MRC

在ARC项目中继续使用MRC编译的类，在编译选项中标识MRC文件即可"-fno-objc-arc"
在MRC项目中继续使用ARC编译的类在编译选项中标识MRC文件即可"-fobjc-arc"

137.NSTimer的内存管理

以下代码有什么问题?

```
@interface SvCheatYourself () {
    NSTimer *_timer;
}

@end

@implementation SvCheatYourself

- (id)init {
    self = [super init];
    if (self) {
        _timer=[NSTimer        scheduledTimerWithTimeInterval:1        target:self
```

```
selector:@selector(testTimer:) userInfo:nil repeats:YES];
    }
    return self;
}

- (void)dealloc {
    [_timer invalidate];
}

- (void)testTimer:(NSTimer*)timer{
    NSLog(@"haha!");
}
@end
```

1）timer都会对它的target进行retain，对于重复性的timer，除非手动关闭，否则对象不会释放，场景：导航控制器关联的控制器无法销毁
2）NSTimer要加到异步线程中，防止线程繁忙导致定时器失准
3）timer必须加入到runloop中才会有效，主线程runloop默认开启，异步线程手动启动
4）注意runloop模式

133.ARC的实现原理

在程序预编译阶段,将 ARC 的代码转换为非 ARC 的 代码,自动加入 release、autorelease、retain

132.Runloop

1> 每个线程上都有一个runloop,主线程默认开启,辅助线程需要手动开启,主要用于
- 使用端口或自定义输入源来和其他线程通信
- 使用线程的定时器
- Cocoa中使用任何performSelector…的方法
- 使线程周期性工作

2> runloop的工作流程

131 OC和C框架对象引用

oc和c桥接

__bridge 不更改归属权
__bridge_transfer 所有权给OC
__bridge_retain 解除OC的所有权

132.fmmpeg框架

答： 音视频编解码框架，内部使用UDP协议针对流媒体开发，内部开辟了六个端口来接受流媒

体数据，完成快速接受之目的。

133.fmdb框架

答：数据库框架，对sqllite的数据操作进行了封装，使用着可把精力都放在sql语句上面。

134. 320框架

答： ui框架，导入320工程作为框架包如同添加一个普通框架一样。cover(open) flower框架(2d 仿射技术)，内部核心类是CATransform3D.

135.UIKit 和 CoreAnimation 和 CoreGraphics 的关系是什么？在开发中是否使用过 CoreAnimation 和 CoreGraphics?

绝大多数图形界面都由 UIKit 完成，UIKit 依赖于 Core Graphics 框架，也是基于 Core Graphics 框架实现的。某些更底层的功能，使用 Core Graphics 完成，是一组自由度更大的图形绘制和动画 API。

UIKit 和 CoreGraphics 主要区别：

（1）Core Graphics 其实是一套基于 C 的 API 框架，使用了 Quartz 作为绘图引擎。这也就意味着 Core Graphics 不是面向对象的。

（2）Core Graphics 需要一个图形上下文（Context）

使用 Core Graphics 来绘图，最简单的方法就是自定义一个类继承自 UIView，并重写子类的 drawRect 方法。在这个方法中绘制图形。

Core Graphics 绘图的步骤：

获取上下文（画布）

创建路径（自定义或者调用系统的 API）并添加到上下文中。

进行绘图内容的设置（画笔颜色、粗细、填充区域颜色、阴影、连接点形状等）

开始绘图（CGContextDrawPath）

释放路径（CGPathRelease）

(1)核心图形 Core Graphics 是用来实现用户界面视觉设计方案的重要技术框架

(2)核心动画 Core Animation 提供了一套用于创建和渲染动态交互效果的简单易行的解决方案，通过与 UIKit 的紧密配合，核心动画可以将界面交互对象与动画过渡效果进行完美地整合。

(3)UIkit 是用来打造 iOS 应用的最重要的图形技术框架，它提供了用于构造触屏设备用户界面的全部工具和资源，并在整个交互体验的塑造过程中扮演着至关重要的角色

2> trasform

修改位移\形变\旋转,transform不同于board\center\frame,前者中记录的是形变的数据,不发生形变其值是空的,所以我们需要新建结构体,用CGAffineTransform(仿射变换)函数给对象结构体属性赋值,而后者是控件的固有属性,内存数据是始终存在的,当我们用他们做移动等操作时,是改变其值,所以是结构体赋值三步曲,不用CG的函数

使用情景区别: transform一般用于有来有回的变化,而frame是有去无回

136.点讲动画和layer ,view的区别

图层不会直接渲染到屏幕上，UIView是iOS系统中界面元素的基础，所有的界面元素都是继承自它。它本身完全是由CoreAnimation来实现的。它真正的绘图部分，是由一个CALayer类来管理。UIView本身更像是一个CALayer的管理器。一个UIView上可以有n个CALayer，每个layer显示一种东西，增强UIView的展现能力。

1>都可以显示屏幕效果

2> 如果需要用户交互就要用UIVIew,其可接收触摸事件(继承UIResponder),而CALayer不能接收触摸事件

3> 如果没有用户交互可选用CALayer,因为其所在库较小,占用的资源较少

详解

1.图层与视图

一个视图就是在屏幕上显示的一个矩形块（比如图片，文字或者视频），它能够 1)拦截类似于鼠标点击或者触摸手势等用户输入。视图在层级关系中可以互相嵌套，一个视图可以管理它的所有子视图的位置。

所有的视图都从一个叫做 UIVIew 的基类派生而来，UIView 可以处理触摸　2)(layer 不行,但是 layer 有锚点和 position 的概念 选装变换的时候用)，可以支持基于 Core Graphics 绘图，可以做仿射变换（例如旋转或者缩放），或者简单的类似于滑动或者渐变的动画。

CALayer 类在概念上和 UIView 类似，同样也是一些被层级关系树管理的矩形块，同样也可以包含一些内容（像图片，文本或者背景色），管理子图层的位置。它们有一些方法和属性用来做动画和变换。和 UIView 最大的不同是 CALayer 不处理用户的交互。

CALayer 并不清楚具体的响应链（iOS 通过视图层级关系用来传送触摸事件的机制），于是它并不能够响应事件，即使 3)它提供了一些方法来判断是否一个触点在图层的范围之内。

2.平行的层级关系

4)每一个 UIview 都有一个 CALayer 实例的图层属性，也就是所谓的 backing layer，视图的职责就是创建并管理这个图层，以确保当子视图在层级关系中添加或者被移除的时候，他们关联的图层也同样对应在层级关系树当中有相同的操作。

实际上这些背后关联的图层才是真正用来在屏幕上显示和做动画，UIView 仅仅是对它的一个封装，提供了一些 iOS 类似于处理触摸的具体功能，以及 Core Animation 底层方法的高级接口。

5)但是为什么 iOS 要基于 UIView 和 CALayer 提供两个平行的层级关系呢？为什么不用一个简单的层级来处理所有事情呢？原因在于要做职责分离，这样也能避免很多重复代码。在 iOS 和 Mac OS 两个平台上，事件和用户交互有很多地方的不同，基于多点触控的用户界面和基于鼠标键盘有着本质的区别，这就是为什么 iOS 有 UIKit 和 UIView，但是 Mac OS 有 AppKit 和 NSView 的原因。他们功能上很相似，但是在实现上有着显著的区别。

绘图，布局和动画，相比之下就是类似 Mac 笔记本和桌面系列一样应用于 iPhone 和 iPad 触屏的概念。把这种功能的逻辑分开并应用到独立的 Core Animation 框架，苹果就能够在 iOS 和 Mac OS 之间共享代码，使得对苹果自己的 OS 开发团队和第三方开发者去开发两个平台的应用更加便捷。

实际上，这里并不是两个层级关系，而是四个，每一个都扮演不同的角色，除了视图层级和图层树之外，还存在呈现树和渲染树，将在第七章“隐式动画”和第十二章“性能调优”分别讨论。

3.图层的能力

如果说 CALayer 是 UIView 内部实现细节，那我们为什么要全面地了解它呢？苹果当然为我们提供了优美简洁的 UIView 接口,那么我们是否就没必要直接去处理 Core Animation 的细节了呢？

某种意义上说的确是这样，对一些简单的需求来说，我们确实没必要处理 CALayer，因为苹果已经通过 UIView 的高级 API 间接地使得动画变得很简单。

但是这种简单会不可避免地带来一些灵活上的缺陷。如果 6)你略微想在底层做一些改变，或者使用一些苹果没有在 UIView 上实现的接口功能，这时除了介入 Core Animation 底层之外别无选择。

我们已经证实了图层不能像视图那样处理触摸事件，那么他能做哪些视图不能做的呢？这里有一些 UIView 没有暴露出来的 CALayer 的功能：(CALayer 高级用法)

阴影，圆角，带颜色的边框

3D 变换

非矩形范围

透明遮罩

多级非线性动画

我们将会在后续章节中探索这些功能，首先我们要关注一下在应用程序当中 CALayer 是怎样被利用起来的。

4. 使用图层

首先我们来创建一个简单的项目，来操纵一些 layer 的属性。打开 Xcode，使用 Single View Application 模板创建一个工程。

在屏幕中央创建一个小视图（大约 200 X 200 的尺寸），当然你可以手工编码，或者使用 Interface Builder（随你方便）。确保你的视图控制器要添加一个视图的属性以便可以直接访问它。我们把它称作 layerView。

运行项目，应该能在浅灰色屏幕背景中看见一个白色方块（图1.3），如果没看见，可能需要调整一下背景window或者view的颜色

136. new和alloc init的区别

new = [alloc]init，采用alloc的方式可以用其他定制的初始化方法。

137. 动画

1> ios界面切换

2> iOS中各种动画的类型&特点&使用场景
CAPropertyAnimation
是CAAnimation的子类，也是个抽象类，要想创建动画对象，应该使用它的两个子类：CABasicAnimation和CAKeyframeAnimation
属性解析：
keyPath：通过指定CALayer的一个属性名称为keyPath(NSString类型)，并且对CALayer的这个属性的值进行修改，达到相应的动画效果。比如，指定@”position”为keyPath，就修改CALayer的position属性的值，以达到平移的动画效果

CABasicAnimation
CAPropertyAnimation的子类
属性解析:
fromValue：keyPath相应属性的初始值
toValue：keyPath相应属性的结束值
随着动画的进行，在长度为duration的持续时间内，keyPath相应属性的值从fromValue渐渐地变为toValue
如果fillMode=kCAFillModeForwards和removedOnComletion=NO，那么在动画执行完毕后，图层会保持显示动画执行后的状态。但在实质上，图层的属性值还是动画执行前的初始值，并没有真正被改变。比如，CALayer的position初始值为(0,0)，CABasicAnimation的fromValue为(10,10)，toValue为(100,100)，虽然动画执行完毕后图层保持在(100,100)这个位置，实质上

图层的position还是为(0,0)

CAKeyframeAnimation
CApropertyAnimation的子类，跟CABasicAnimation的区别是：CABasicAnimation只能从一个数值(fromValue)变到另一个数值(toValue)，而CAKeyframeAnimation会使用一个NSArray保存这些数值
属性解析：
values：就是上述的NSArray对象。里面的元素称为”关键帧”(keyframe)。动画对象会在指定的时间(duration)内，依次显示values数组中的每一个关键帧
path：可以设置一个CGPathRef\CGMutablePathRef,让层跟着路径移动。path只对CALayer的anchorPoint和position起作用。如果你设置了path，那么values将被忽略
keyTimes：可以为对应的关键帧指定对应的时间点,其取值范围为0到1.0,keyTimes中的每一个时间值都对应values中的每一帧.当keyTimes没有设置的时候,各个关键帧的时间是平分的
CABasicAnimation可看做是最多只有2个关键帧的CAKeyframeAnimation

CAAnimationGroup
CAAnimation的子类，可以保存一组动画对象，将CAAnimationGroup对象加入层后，组中所有动画对象可以同时并发运行
属性解析：
animations：用来保存一组动画对象的NSArray
默认情况下，一组动画对象是同时运行的，也可以通过设置动画对象的beginTime属性来更改动画的开始时间

CATransition
CAAnimation的子类，用于做转场动画，能够为层提供移出屏幕和移入屏幕的动画效果。iOS比Mac OS X的转场动画效果少一点
UINavigationController就是通过CATransition实现了将控制器的视图推入屏幕的动画效果
属性解析:
type：动画过渡类型
subtype：动画过渡方向
startProgress：动画起点(在整体动画的百分比)
endProgress：动画终点(在整体动画的百分比)

UIView动画
UIKit直接将动画集成到UIView类中，当内部的一些属性发生改变时，UIView将为这些改变提供动画支持
执行动画所需要的工作由UIView类自动完成，但仍要在希望执行动画时通知视图，为此需要将改变属性的代码放在[UIView beginAnimations:nil context:nil]和[UIView commitAnimations]之间

Block动画

帧动画

138.UICollectionView

1> 如何实现瀑布流,流水布局
1.1> 使用UICollectionView
1.2> 使用自定义的FlowLayout
1.3> 需要在layoutAttributesForElementsInRect中设置自定义的布局(item的frame)
1.4> 在 prepareLayout中计算布局
1.5> 遍历数据内容，根据索引取出对应的attributes(使用layoutAttributesForCellWithIndexPath),根据九宫格算法设置布局
1.6> 细节1: 实时布局,重写shouldInvalidateLayoutForBoundsChange(bounds改变重新布局,scrollview的contentoffset>bounds)
1.7> 细节2: 计算设置itemsize(保证内容显示完整,uicollectionview的content size是根据itemize计算的),根据列最大高度/对应列数量求出,最大高度累加得到
1.8> 细节3: 追加item到最短列,避免底部参差不齐.

2> 和UITableView的使用区别
1）必须使用下面的方法进行Cell类的注册：
1 - (void)registerClass:forCellWithReuseIdentifier:
2 - (void)registerClass:forSupplementaryViewOfKind:withReuseIdentifier:
3 - (void)registerNib:forCellWithReuseIdentifier:
2）collectionView与tableView最大的不同点，collectionView必须要使用自己的layout（UICollectionViewLayout）
如：

```
• UICollectionViewFlowLayout *flowLayout = [[UICollectionViewFlowLayout alloc]
      init];
• flowLayout.itemSize = CGSizeMake(52, 52);                // cell大小
• flowLayout.minimumInteritemSpacing = 1;                  // cell间距
• flowLayout.minimumLineSpacing = 1;                       // cell行距
• flowLayout.sectionInset = (UIEdgeInsets){81,1,1,1};      // cell边距
```

创建collectionView需要带Layout的初始化方法：

```
• - (id)initWithFrame:(CGRect)frame
      collectionViewLayout:(UICollectionViewLayout *)layout;
```

139.UIImage

1> 有哪几种加载方式
1.1> 二进制　imageWithData
1.2> Bundle　imageWithName
1.3> 本地路径　imageWithContentOfFile
1.4>

140.webview

1>解决webview的内存占用和泄露

141.描述九宫格算法

1> 根据格子宽appW高appH和每行格数totalCol计算格子间隙marginX

```
CGFloat marginX = (self.view.frame.size.width - totalCol * appW)/(totalCol + 1);
```

2> 根据序号i和每行格数totalCol计算行号列号

```
    int row = i / totalCol;
       int col = i % totalCol;
```

3> 根据格子间隙、格子宽高和行号列号计算x,y

```
CGFloat appX = marginX + col * (appW + marginX);
CGFloat appY = row * (appH + marginY);
```

142.实现图片轮播图

1> UIScrollView设置contentSize，添加图片并设置frame，设置分页
2> 添加分页控制器，在UIScrollView滚动代理方法中根据contentOffset计算当前页数并设置
3> 设置定时器，主动改变contentOffset,设置定时器的模式进行并发操作(终极方案定时器放在异步线程)

143.iOS网络框架

1> NSURLConnection和NSURLSession的区别
1.1> 异步请求不需要NSOperation包装
1.2> 支持后台运行的网络任务(后台上传下载)
1.3> 根据每个Session做配置(http header，Cache,Cookie,protocal,Credential)，不再在整个App层面共享配置
1.4> 支持网络操作的取消和断点续传(继承系统类，重新main方法)
1.5> 改进了授权机制的处理

144.网络

数据解析

1> XML解析方式
SAX 方式解析
－只读
－速度快
－从上向下
－通过5个代理方法解析，每个代理方中都需要写一些代码！
－如果要实现SAX解析，思路最重要！
－适合比价大的XML的解析
DOM解析的特点
－一次性将XML全部加载到内存，以树形结构
－好处，可以动态的修改，添加，删除节点

－内存消耗非常大！尤其横向节点越深！
－iOS默认不支持 DOM 解析！
－在 MAC 端，或者服务器端开发，都基本上使用 DOM 解析
－在 iOS 端如果需要使用 DOM 方式解析，可以使用第三方框GData/KissXML(XMPP)
－适合比较小的 XML 文件
－在 MAC 中，苹果提供了一个 NSXML 的类，能够做 DOM 解析，在 iOS 不能使用！

2> json&xml的区别

1)解码难度: json的解码难度基本为零,xml需要考虑子节点和父节点
2)数据体积&传输速度: json相对于xml来讲,数据体积小,json的速度远远快于xml
3)数据交互: json与JavaScript的交互更加方面,更容易解析处理,更好的数据交互
4)数据描述: xml对数据描述性比较好

144.网络传输

1>DNS是如何工作的

DNS是domain name server的简称,每个网络的计算机都有ip,但是不好记,所以用域名替代(如www.baidu.com),在 Internet 上真实在辨识机器的还是 IP，所以当使用者输入Domain Name后，浏览器必须要先去一台有 Domain Name 和IP 对应资料的主机去查询这台电脑的 IP，而这台被查询的主机,我们称它为 Domain Name Server,简称DNS,例如:当你输入www.pchome.com.tw时，浏览器会将www.pchome.com.tw这个名字传送到离他最近的 DNS Server 去做辨识，如果查到，则会传回这台主机的 IP,进而跟它索取资料，但如果没查到，就会发生类似 DNS NOT FOUND的情形，所以一旦DNS Server当机，就像是路标完全被毁坏，没有人知道该把资料送到那里

2> POST请求常见的数据格式

145.AFN

1>实现原理
AFN的直接操作对象AFHTTPClient不同于ASI，是一个实现了NSCoding和NSCopying协议的NSObject子类。 AFHTTPClient是一个封装了一系列操作方法的“工具类”，处理请求的操作类是一系列单独的，基于NSOperation封装 的，AFURLConnectionOperation的子类。AFN的示例代码中通过一个静态方法，使用dispatch_once()的方式创建 AFHTTPClient的共享实例，这也是官方建议的使用方法。在创建AFHTTPClient的初始化方法中，创建了OperationQueue并 设置一系列参数默认值。在getPath:parameters:success:failure方法中创建NSURLRequest，以NSURLRequest对象实例作为参数，创建一个NSOperation，并加入在初始化发方中创建的NSOperationQueue。以上操作都 是在主线程中完成的。在NSOperation的start方法中，以此前创建的NSURLRequest对象为参数创建NSURLConnection 并开启连结。

2> 传递指针 如何使一个方法返回多个返回值
传参指针变量的地址,方法内部通过*运算符使用该地址可以修改该地址保存的内容(引用对象的地址),当外部再次使用该指针变量取出引用对象时,引用对象已经在方法内部发生了改变,指针变量指向其他数据,相当于方法的返回值(经方法处理后生成的外部可使用的结果数据).

146. AFNetworking&ASIHttpRequest&MKNetWorking

一、底层实现

1、AFN 的底层实现基于 OC 的 NSURLConnection 和 NSURLSession

2、ASI 的底层实现基于纯 C 语言的 CFNetwork 框架

3、因为 NSURLConnection 和 NSURLSession 是在 CFNetwork 之上的一层封装，因此 ASI 的运行性能高于 AFN

二、对服务器返回的数据处理

1、ASI 没有直接提供对服务器数据处理的方式，直接返回的是 NSData/NSString

2、AFN 提供了多种对服务器数据处理的方式

(1)JSON 处理-直接返回 NSDictionary 或者 NSArray

(2)XML 处理-返回的是 xml 类型数据，需对其进行解析

(3)其他类型数据处理

三、监听请求过程

1、AFN 提供了 success 和 failure 两个 block 来监听请求的过程（只能监听成功和失败）

* success ： 请求成功后调用

* failure ： 请求失败后调用

2、ASI 提供了 3 套方案，每一套方案都能监听请求的完整过程

（监听请求开始、接收到响应头信息、接受到具体数据、接受完毕、请求失败）

* 成为代理，遵守协议，实现协议中的代理方法

* 成为代理，不遵守协议，自定义代理方法

* 设置 block

四、在文件下载和文件上传的使用难易度

1、AFN

*不容易实现监听下载进度和上传进度

*不容易实现断点续传

*一般只用来下载不大的文件

2、ASI(ipv6)

*非常容易实现下载和上传

*非常容易监听下载进度和上传进度

*非常容易实现断点续传

*下载大文件或小文件均可

3、实现下载上传推荐使用 ASI

五、网络监控

1、AFN 自己封装了网络监控类，易使用

2、ASI 使用的是 Reachability，因为使用 CocoaPods 下载 ASI 时，会同步下载 Reachability，但 Reachability 作为网络监控使用较为复杂（相对于 AFN 的网络监控类来说）

3、推荐使用 AFN 做网络监控-AFNetworkReachabilityManager

六、ASI 提供的其他实用功能

1、控制信号旁边的圈圈要不要在请求过程中转

2、可以轻松地设置请求之间的依赖：每一个请求都是一个 NSOperation 对象

3、可以统一管理所有请求（还专门提供了一个叫做 ASINetworkQueue 来管理所有的请求对象）

* 暂停/恢复/取消所有的请求

* 监听整个队列中所有请求的下载进度和上传进度

MKNetworkKit 是一个使用十分方便，功能又十分强大、完整的 iOS 网络编程代码库。它只有两个类，它的目标是使用像 AFNetworking 这么简单，而功能像 ASIHTTPRequest(已经停止维护)那么强大。它除了拥有 AFNetworking 和 ASIHTTPRequest 所有功能以外，还有一些新特色，包括：

1、高度的轻量级，仅仅只有 2 个主类

2、自主操作多个网络请求

3、更加准确的显示网络活动指标

4、自动设置网络速度，实现自动的 2G、3G、wifi 切换

5、自动缓冲技术的完美应用，实现网络操作记忆功能，当你掉线了又上线后，会继续执行未完成的网络请求

6、可以实现网络请求的暂停功能

7、准确无误的成功执行一次网络请求，摒弃后台的多次请求浪费

8、支持图片缓冲

9、支持 ARC 机制

10、在整个 app 中可以只用一个队列（queue），队列的大小可以自动调整

147 如何进行性能优化

1> 内存优化的点　重用　懒加载

2> 渲染优化　尽量使用不透明的图　把 views 设置为透明

3> 在ImageView设置前,尽量先调整好图片大小　尤其放在uiscrolliview中自动缩放耗能

4> 避免使用过大的xib　和分镜的区别　一次性加载

5> 不要阻塞主线程　除渲染,触摸响应等　尽量异步处理　如存储,网络 异步线程通知

6> 缓存　网络响应,图片,计算结果(行高)　网络响应NSUrlconnection默认缓存request,设置策略　非网络请求 使用nscache nsdictionary

7> 避免反复处理数据　在服务器端和客户端使用相同的数据结构

8> 选择正确的数据格式　json　速度快 解析方便　xml　sax方式逐行解析 解析大文件不占用内存和损失性能

9> 优化tableview　重用cell 缓存行高 cell子视图尽量少且不透明

10> 选择正确的数据存储选项　plist nscoding NSUserDefaults sqlite coredata

148.算法和数据结构

（1）数据结构

二叉树

链表
递归

(2)常用的算法

交换数值的几种方法　中间变量　加减法　异或
oc/c实现常用排序

149.手写单例

```
@interface Singleton: NSObject

+(instance)shareInstance;

@end

#import “Singleton”

@implemention Singleton

static Singleton  *_instance = nil;

+(instance)shareInstance{

      static dispatch_once_t onceToken;

     dispatch_once({

      _instance = [[seif alloc]init];
});

  return _instance;
}
```

149.开发技巧

149.description

重写类的description，打印该对象创建的类，可以打印我问想要的输出

150.静态库

如何给静态库添加属性　分类+runtime
如何调用私有方法　performselector　category(前向引用)

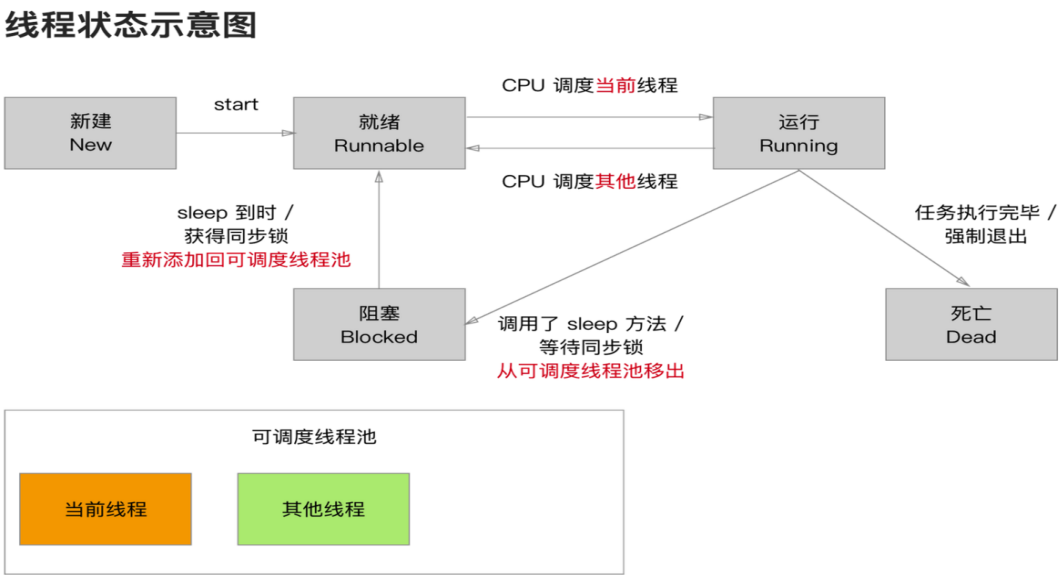
151.多线程专篇

一、多线程的基本概念

- 进程：可以理解成一个运行中的应用程序，是系统进行资源分配和调度的基本单位，是操作系统结构的基础，主要管理资源。
- 线程：是进程的基本执行单元，一个进程对应多个线程。
- 主线程：处理UI，所有更新UI的操作都必须在主线程上执行。不要把耗时操作放在主线程，会卡界面。
- 多线程：在同一时刻，一个CPU只能处理1条线程，但CPU可以在多条线程之间快速的切换，只要切换的足够快，就造成了多线程一同执行的假象。
- 线程就像火车的一节车厢，进程则是火车。车厢（线程）离开火车（进程）是无法跑动的，而火车（进程）至少有一节车厢（主线程）。多线程可以看做多个车厢，它的出现是为了提高效率。
- 多线程是通过提高资源使用率来提高系统总体的效率。
- 我们运用多线程的目的是：将耗时的操作放在后台执行！

二、线程的状态与生命周期

下图是线程状态示意图，从图中可以看出线程的生命周期是：新建 - 就绪 - 运行 - 阻塞 - 死亡



下面分别阐述线程生命周期中的每一步

- 新建：实例化线程对象
- 就绪：向线程对象发送start消息，线程对象被加入可调度线程池等待CPU调度。
- 运行：CPU 负责调度可调度线程池中线程的执行。线程执行完成之前，状态可能会在就绪和运行之间来回切换。就绪和运行之间的状态变化由CPU负责，程序员不能干预。
- 阻塞：当满足某个预定条件时，可以使用休眠或锁，阻塞线程执行。sleepForTimeInterval（休眠指定时长），sleepUntilDate（休眠到指定日期），@synchronized(self)：（互斥锁）。
- 死亡：正常死亡，线程执行完毕。非正常死亡，当满足某个条件后，在线程内部中止执行/在主线程中止线程对象

- 还有线程的exit和cancel
- [NSThread exit]：一旦强行终止线程，后续的所有代码都不会被执行。
- [thread cancel]取消：并不会直接取消线程，只是给线程对象添加 isCancelled 标记。

三、多线程的四种解决方案

多线程的四种解决方案分别是：pthread，NSThread，GCD， NSOperation。
下图是对这四种方案进行了解读和对比。

方案	简介	语言	线程生命周期	使用频率
pthread	• 一套通用的多线程API • 适用于 Unix / Linux / Windows 等系统 • 跨平台\可移植 • 使用难度大	C	程序员管理	几乎不用
NSThread	• 使用更加面向对象 • 简单易用，可直接操作线程对象	OC	程序员管理	偶尔使用
GCD	• 旨在替代 NSThread 等线程技术 • 充分利用设备的多核	C	自动管理	经常使用
NSOperation	• 基于GCD（底层是GCD） • 比 GCD 多了一些更简单实用的功能 • 使用更加面向对象	OC	自动管理	经常使用

四、线程安全问题

当多个线程访问同一块资源时，很容易引发数据错乱和数据安全问题。就好比几个人在同一时修改同一个表格，造成数据的错乱。
解决多线程安全问题的方法
4 方法一：互斥锁（同步锁）

```
@synchronized(锁对象) {
    // 需要锁定的代码
}
```

判断的时候锁对象要存在，如果代码中只有一个地方需要加锁，大多都使用self作为锁对象，这样可以避免单独再创建一个锁对象。
加了互斥做的代码，当新线程访问时，如果发现其他线程正在执行锁定的代码，新线程就会进入休眠。
- 方法二：自旋锁

加了自旋锁，当新线程访问代码时，如果发现有其他线程正在锁定代码，新线程会用死循环的方式，一直等待锁定的代码执行完成。相当于不停尝试执行代码，比较消耗性能。
属性修饰atomic本身就有一把自旋锁。
下面说一下属性修饰nonatomic 和 atomic

nonatomic 非原子属性,同一时间可以有很多线程读和写

atomic 原子属性(线程安全)，保证同一时间只有一个线程能够写入(但是同一个时间多个线程都可以取值)，atomic 本身就有一把锁(自旋锁)

atomic：线程安全，需要消耗大量的资源

nonatomic：非线程安全，不过效率更高，一般使用nonatomic

五、NSThread的使用

No.1：NSThread创建线程

NSThread有三种创建方式：

- init方式
- detachNewThreadSelector创建好之后自动启动
- performSelectorInBackground创建好之后也是直接启动

```
/**  方法一，需要start  */
NSThread *thread1 = [[NSThread alloc] initWithTarget:self selector:@selector(doSomething1:) object:@"NSThread1"];
//  线程加入线程池等待CPU调度，时间很快，几乎是立刻执行
[thread1 start];
/**  方法二，创建好之后自动启动  */
[NSThread detachNewThreadSelector:@selector(doSomething2:) toTarget:self withObject:@"NSThread2"];
/**  方法三，隐式创建，直接启动  */
[self performSelectorInBackground:@selector(doSomething3:) withObject:@"NSThread3"];

- (void)doSomething1:(NSObject *)object {
    //  传递过来的参数
    NSLog(@"%@",object);
    NSLog(@"doSomething1：%@",[NSThread currentThread]);
}

- (void)doSomething2:(NSObject *)object {
    NSLog(@"%@",object);
    NSLog(@"doSomething2：%@",[NSThread currentThread]);
}

- (void)doSomething3:(NSObject *)object {
    NSLog(@"%@",object);
    NSLog(@"doSomething3：%@",[NSThread currentThread]);
}
```

No.2：NSThread的类方法

- 返回当前线程

```
// 当前线程
[NSThread currentThread];
NSLog(@"%@",[NSThread currentThread]);

// 如果number=1，则表示在主线程，否则是子线程
```

- 打印结果：{number = 1, name = main}

阻塞休眠

```
//休眠多久
```

```
[NSThread sleepForTimeInterval:2];
//休眠到指定时间
[NSThread sleepUntilDate:[NSDate date]];
```

类方法补充

```
//退出线程
[NSThread  exit];
//判断当前线程是否为主线程
[NSThread  isMainThread];
//判断当前线程是否是多线程
[NSThread  isMultiThreaded];
//主线程的对象
NSThread  *mainThread  =  [NSThread  mainThread];
```

No.3：NSThread的一些属性

```
//线程是否在执行
thread.isExecuting;
//线程是否被取消
thread.isCancelled;
//线程是否完成
thread.isFinished;
//是否是主线程
thread.isMainThread;
//线程的优先级，取值范围0.0到1.0，默认优先级0.5，1.0表示最高优先级，优先级高，CPU调度的频率高
  thread.threadPriority;
```

Demo：WHMultiThreadDemo

六、GCD的理解与使用

No.1：GCD的特点

- GCD会自动利用更多的CPU内核
- GCD自动管理线程的生命周期（创建线程，调度任务，销毁线程等）
- 程序员只需要告诉 GCD 想要如何执行什么任务，不需要编写任何线程管理代码

No.2：GCD的基本概念

任务（block）：任务就是将要在线程中执行的代码，将这段代码用block封装好，然后将这个任务添加到指定的执行方式（同步执行和异步执行），等待CPU从队列中取出任务放到对应的线程中执行。

同步（sync）：一个接着一个，前一个没有执行完，后面不能执行，不开线程。

异步（async）：开启多个新线程，任务同一时间可以一起执行。异步是多线程的代名词

队列：装载线程任务的队形结构。(系统以先进先出的方式调度队列中的任务执行)。在GCD中有两种队列：串行队列和并发队列。

并发队列：线程可以同时一起进行执行。实际上是CPU在多条线程之间快速的切换。（并发功能只有在异步（dispatch_async）函数下才有效）

串行队列：线程只能依次有序的执行。

GCD总结：将任务(要在线程中执行的操作block)添加到队列(自己创建或使用全局并发队列)，并且指定执行任务的方式(异步dispatch_async，同步dispatch_sync)

No.3：队列的创建方法

- 使用dispatch_queue_create来创建队列对象，传入两个参数，第一个参数表示队列的唯一标识符，可为空。第二个参数用来表示串行队列（DISPATCH_QUEUE_SERIAL）或并发队列（DISPATCH_QUEUE_CONCURRENT）。

```
//  串行队列
dispatch_queue_t  queue  =  dispatch_queue_create("test",  DISPATCH_QUEUE_SERIAL);
//  并发队列
dispatch_queue_t  queue1  =  dispatch_queue_create("test",  DISPATCH_QUEUE_CONCURRENT);
```

GCD的队列还有另外两种：

主队列：主队列负责在主线程上调度任务，如果在主线程上已经有任务正在执行，主队列会等到主线程空闲后再调度任务。通常是返回主线程更新UI的时候使用。

```
dispatch_get_main_queue()
    dispatch_async(dispatch_get_global_queue(0,  0),  ^{
            //  耗时操作放在这里

            dispatch_async(dispatch_get_main_queue(),  ^{
                    //  回到主线程进行UI操作

            });
    });
```

- 全局并发队列：全局并发队列是就是一个并发队列，是为了让我们更方便的使用多线程。

    dispatch_get_global_queue(0, 0)

```
//全局并发队列
dispatch_queue_t  queue  =  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,  0);
//全局并发队列的优先级
#define  DISPATCH_QUEUE_PRIORITY_HIGH  2  //  高优先级
#define  DISPATCH_QUEUE_PRIORITY_DEFAULT  0  //  默认（中）优先级
#define  DISPATCH_QUEUE_PRIORITY_LOW  (-2)  //  低优先级
#define  DISPATCH_QUEUE_PRIORITY_BACKGROUND  INT16_MIN  //  后台优先级
//iOS8开始使用服务质量，现在获取全局并发队列时，可以直接传0
dispatch_get_global_queue(0,  0);
```

No.4：同步/异步/任务、创建方式

同步（sync）使用dispatch_sync来表示。

异步（async）使用dispatch_async。

任务就是将要在线程中执行的代码，将这段代码用block封装好。

代码如下：

```
        //  同步执行任务
        dispatch_sync(dispatch_get_global_queue(0,  0),  ^{
                //  任务放在这个block里
                NSLog(@"我是同步执行的任务");

        });
        //  异步执行任务
        dispatch_async(dispatch_get_global_queue(0,  0),  ^{
                //  任务放在这个block里
                NSLog(@"我是异步执行的任务");

        });
```

Demo：WHMultiThreadDemo

No.5：GCD的使用

由于有多种队列（串行/并发/主队列）和两种执行方式（同步/异步），所以他们之间可以有多种组合方式。

1 串行同步
2 串行异步
3 并发同步
4 并发异步
5 主队列同步
6 主队列异步

• 串行同步

执行完一个任务，再执行下一个任务。不开启新线程。

```
/**  串行同步  */
-  (void)syncSerial  {

        NSLog(@"\n\n**************串行同步**************\n\n");

        //  串行队列
        dispatch_queue_t  queue  =  dispatch_queue_create("test",  DISPATCH_QUEUE_SERIAL);

        //  同步执行
        dispatch_sync(queue,  ^{
                for  (int  i  =  0;  i  <  3;  i++)  {
                        NSLog(@"串行同步1      %@",[NSThread  currentThread]);
                }
        });
        dispatch_sync(queue,  ^{
                for  (int  i  =  0;  i  <  3;  i++)  {
                        NSLog(@"串行同步2      %@",[NSThread  currentThread]);
                }
        });
```

```
        dispatch_sync(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"串行同步3      %@",[NSThread currentThread]);
                }
        });
}
```

输入结果为顺序执行，都在主线程：

```
串行同步1      {number = 1, name = main}
串行同步1      {number = 1, name = main}
串行同步1      {number = 1, name = main}
串行同步2      {number = 1, name = main}
串行同步2      {number = 1, name = main}
串行同步2      {number = 1, name = main}
串行同步3      {number = 1, name = main}
串行同步3      {number = 1, name = main}
串行同步3      {number = 1, name = main}
```

- 串行异步

开启新线程，但因为任务是串行的，所以还是按顺序执行任务。

```
/**  串行异步  */
- (void)asyncSerial {

        NSLog(@"\n\n**************串行异步***************\n\n");

        // 串行队列
        dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_SERIAL);

        // 同步执行
        dispatch_async(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"串行异步1      %@",[NSThread currentThread]);
                }
        });
        dispatch_async(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"串行异步2      %@",[NSThread currentThread]);
                }
        });
        dispatch_async(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"串行异步3      %@",[NSThread currentThread]);
                }
        });
}
```

输入结果为顺序执行，有不同线程：

```
串行异步1      {number = 3, name = (null)}
串行异步1      {number = 3, name = (null)}
串行异步1      {number = 3, name = (null)}
串行异步2      {number = 3, name = (null)}
串行异步2      {number = 3, name = (null)}
串行异步2      {number = 3, name = (null)}
串行异步3      {number = 3, name = (null)}
串行异步3      {number = 3, name = (null)}
串行异步3      {number = 3, name = (null)}
```

- 并发同步

因为是同步的，所以执行完一个任务，再执行下一个任务。不会开启新线程。

```
/**  并发同步  */
- (void)syncConcurrent {

    NSLog(@"\n\n**************并发同步***************\n\n");

    // 并发队列
    dispatch_queue_t queue = dispatch_queue_create("test",
    DISPATCH_QUEUE_CONCURRENT);

    // 同步执行
    dispatch_sync(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"并发同步1      %@",[NSThread currentThread]);
        }
    });
    dispatch_sync(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"并发同步2      %@",[NSThread currentThread]);
        }
    });
    dispatch_sync(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"并发同步3      %@",[NSThread currentThread]);
        }
    });
}
```

输入结果为顺序执行，都在主线程：

```
并发同步1      {number = 1, name = main}
并发同步1      {number = 1, name = main}
并发同步1      {number = 1, name = main}
并发同步2      {number = 1, name = main}
```

```
并发同步2       {number = 1, name = main}
并发同步2       {number = 1, name = main}
并发同步3       {number = 1, name = main}
并发同步3       {number = 1, name = main}
并发同步3       {number = 1, name = main}
```

- 并发异步

任务交替执行，开启多线程。

```
/**  并发异步  */
- (void)asyncConcurrent {

        NSLog(@"\n\n**************并发异步**************\n\n");

        //  并发队列
        dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_CONCURRENT);

        //  同步执行
        dispatch_async(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"并发异步1      %@",[NSThread currentThread]);
                }
        });
        dispatch_async(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"并发异步2      %@",[NSThread currentThread]);
                }
        });
        dispatch_async(queue, ^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"并发异步3      %@",[NSThread currentThread]);
                }
        });
}
```

输入结果为无序执行，有多条线程：

```
并发异步1       {number = 3, name = (null)}
并发异步2       {number = 4, name = (null)}
并发异步3       {number = 5, name = (null)}
并发异步1       {number = 3, name = (null)}
并发异步2       {number = 4, name = (null)}
并发异步3       {number = 5, name = (null)}
并发异步1       {number = 3, name = (null)}
并发异步2       {number = 4, name = (null)}
并发异步3       {number = 5, name = (null)}
```

- 主队列同步

如果在主线程中运用这种方式，则会发生死锁，程序崩溃。

```
/**  主队列同步  */
-  (void)syncMain  {

        NSLog(@"\n\n**************主队列同步，放到主线程会死锁**************\n\n");

        //  主队列
        dispatch_queue_t  queue  =  dispatch_get_main_queue();

        dispatch_sync(queue,  ^{
                for  (int  i  =  0;  i  <  3;  i++)  {
                        NSLog(@"主队列同步1      %@",[NSThread  currentThread]);
                }
        });
        dispatch_sync(queue,  ^{
                for  (int  i  =  0;  i  <  3;  i++)  {
                        NSLog(@"主队列同步2      %@",[NSThread  currentThread]);
                }
        });
        dispatch_sync(queue,  ^{
                for  (int  i  =  0;  i  <  3;  i++)  {
                        NSLog(@"主队列同步3      %@",[NSThread  currentThread]);
                }
        });
}
```

主队列同步造成死锁的原因：

1 如果在主线程中运用主队列同步，也就是把任务放到了主线程的队列中。

2 而同步对于任务是立刻执行的，那么当把第一个任务放进主队列时，它就会立马执行。

3 可是主线程现在正在处理syncMain方法，任务需要等syncMain执行完才能执行。

4 syncMain执行到第一个任务的时候，又要等第一个任务执行完才能往下执行第二个和第三个任务。

5 这样syncMain方法和第一个任务就开始了互相等待，形成了死锁。

• 主队列异步

在主线程中任务按顺序执行。

```
/**  主队列异步  */
-  (void)asyncMain  {

        NSLog(@"\n\n**************主队列异步**************\n\n");

        //  主队列
        dispatch_queue_t  queue  =  dispatch_get_main_queue();

        dispatch_sync(queue,  ^{
                for  (int  i  =  0;  i  <  3;  i++)  {
                        NSLog(@"主队列异步1      %@",[NSThread  currentThread]);
```

```
        }
    });
    dispatch_sync(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"主队列异步2 %@",[NSThread currentThread]);
        }
    });
    dispatch_sync(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"主队列异步3 %@",[NSThread currentThread]);
        }
    });
}
```

输入结果为在主线程中按顺序执行：

```
主队列异步1 {number = 1, name = main}
主队列异步1 {number = 1, name = main}
主队列异步1 {number = 1, name = main}
主队列异步2 {number = 1, name = main}
主队列异步2 {number = 1, name = main}
主队列异步2 {number = 1, name = main}
主队列异步3 {number = 1, name = main}
主队列异步3 {number = 1, name = main}
主队列异步3 {number = 1, name = main}
```

• GCD线程之间的通讯

开发中需要在主线程上进行UI的相关操作，通常会把一些耗时的操作放在其他线程，比如说图片文件下载等耗时操作。

当完成了耗时操作之后，需要回到主线程进行UI的处理，这里就用到了线程之间的通讯。

```
- (IBAction)communicationBetweenThread:(id)sender {

    // 异步
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        // 耗时操作放在这里，例如下载图片。（运用线程休眠两秒来模拟耗时操作）
        [NSThread sleepForTimeInterval:2];
NSString*picURLStr=@"http://www.bangmangxuan.net/uploads/allimg/160320/74-160320130500.jpg";
        NSURL *picURL = [NSURL URLWithString:picURLStr];
        NSData *picData = [NSData dataWithContentsOfURL:picURL];
        UIImage *image = [UIImage imageWithData:picData];

        // 回到主线程处理UI
        dispatch_async(dispatch_get_main_queue(), ^{
            // 在主线程上添加图片
            self.imageView.image = image;
        });
    });
```

```
}
```
上面的代码是在新开的线程中进行图片的下载，下载完成之后回到主线程显示图片。

• GCD栅栏

当任务需要异步进行，但是这些任务需要分成两组来执行，第一组完成之后才能进行第二组的操作。这时候就用了到GCD的栅栏方法dispatch_barrier_async。

```
- (IBAction)barrierGCD:(id)sender {

    // 并发队列
    dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_CONCURRENT

    // 异步执行
    dispatch_async(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"栅栏：并发异步1 %@",[NSThread currentThread]);
        }
    });
    dispatch_async(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"栅栏：并发异步2 %@",[NSThread currentThread]);
        }
    });

    dispatch_barrier_async(queue, ^{
        NSLog(@"------------barrier------------%@", [NSThread currentThread]);
        NSLog(@"******* 并发异步执行，但是34一定在12后面 *********");
    });

    dispatch_async(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"栅栏：并发异步3 %@",[NSThread currentThread]);
        }
    });
    dispatch_async(queue, ^{
        for (int i = 0; i < 3; i++) {
            NSLog(@"栅栏：并发异步4 %@",[NSThread currentThread]);
        }
    });
}
```
上面代码的打印结果如下，开启了多条线程，所有任务都是并发异步进行。但是第一组完成之后，才会进行第二组的操作。

```
栅栏：并发异步1 {number = 3, name = (null)}
栅栏：并发异步2 {number = 6, name = (null)}
栅栏：并发异步1 {number = 3, name = (null)}
栅栏：并发异步2 {number = 6, name = (null)}
```

```
栅栏：并发异步1      {number = 3, name = (null)}
栅栏：并发异步2      {number = 6, name = (null)}
  ------------barrier------------{number = 6, name = (null)}
*******  并发异步执行，但是34一定在12后面  *********
栅栏：并发异步4      {number = 3, name = (null)}
栅栏：并发异步3      {number = 6, name = (null)}
栅栏：并发异步4      {number = 3, name = (null)}
栅栏：并发异步3      {number = 6, name = (null)}
栅栏：并发异步4      {number = 3, name = (null)}
栅栏：并发异步3      {number = 6, name = (null)}
```

• GCD延时执行

当需要等待一会再执行一段代码时，就可以用到这个方法了：dispatch_after。

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(5.0 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
        //  5秒后异步执行
        NSLog(@"我已经等待了5秒！");
});
```

GCD实现代码只执行一次

使用dispatch_once能保证某段代码在程序运行过程中只被执行1次。可以用来设计单例。

```
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
        NSLog(@"程序运行过程中我只执行了一次！");
});
```

• GCD快速迭代

GCD有一个快速迭代的方法dispatch_apply，dispatch_apply可以同时遍历多个数字。

```
- (IBAction)applyGCD:(id)sender {

        NSLog(@"\n\n**************  GCD快速迭代  **************\n\n");

        //  并发队列
        dispatch_queue_t queue = dispatch_get_global_queue(0, 0);

        //  dispatch_apply几乎同时遍历多个数字
        dispatch_apply(7, queue, ^(size_t index) {
                NSLog(@"dispatch_apply：%zd======%@",index, [NSThread currentThread]);
        });
}
```

打印结果如下：

```
dispatch_apply：0======{number = 1, name = main}
dispatch_apply：1======{number = 1, name = main}
dispatch_apply：2======{number = 1, name = main}
dispatch_apply：3======{number = 1, name = main}
dispatch_apply：4======{number = 1, name = main}
dispatch_apply：5======{number = 1, name = main}
```

```
dispatch_apply：6======{number = 1, name = main}
```

• GCD队列组

异步执行几个耗时操作，当这几个操作都完成之后再回到主线程进行操作，就可以用到队列组了。

队列组有下面几个特点：

1 所有的任务会并发的执行(不按序)。

2 所有的异步函数都添加到队列中，然后再纳入队列组的监听范围。

3 使用dispatch_group_notify函数，来监听上面的任务是否完成，如果完成，就会调用这个方法。

队列组示例代码：

```
- (void)testGroup {
        dispatch_group_t group = dispatch_group_create();

        dispatch_group_async(group, dispatch_get_global_queue(0, 0), ^{
                NSLog(@"队列组：有一个耗时操作完成！");
        });

        dispatch_group_async(group, dispatch_get_global_queue(0, 0), ^{
                NSLog(@"队列组：有一个耗时操作完成！");
        });

        dispatch_group_notify(group, dispatch_get_main_queue(), ^{
                NSLog(@"队列组：前面的耗时操作都完成了，回到主线程进行相关操作");
        });
}
```

打印结果如下：

```
队列组：有一个耗时操作完成！
队列组：有一个耗时操作完成！
队列组：前面的耗时操作都完成了，回到主线程进行相关操作
```

至此，GCD的相关内容叙述完毕。下面让我们继续学习NSOperation。

Demo：WHMultiThreadDemo

七、NSOperation的理解与使用

No.1：NSOperation简介

NSOperation是基于GCD之上的更高一层封装，NSOperation需要配合NSOperationQueue来实现多线程。

NSOperation实现多线程的步骤如下：

1. 创建任务：先将需要执行的操作封装到NSOperation对象中。
2. 创建队列：创建NSOperationQueue。
3. 将任务加入到队列中：将NSOperation对象添加到NSOperationQueue中。

需要注意的是，NSOperation是个抽象类，实际运用时中需要使用它的子类，有三种方式：

1 使用子类NSInvocationOperation

2 使用子类NSBlockOperation

3 定义继承自NSOperation的子类，通过实现内部相应的方法来封装任务。

No.2：NSOperation的三种创建方式

• NSInvocationOperation的使用

创建NSInvocationOperation对象并关联方法，之后start。

```
-  (void)testNSInvocationOperation  {
        //  创建NSInvocationOperation
        NSInvocationOperation  *invocationOperation  =  [[NSInvocationOperation  alloc]
             initWithTarget:self  selector:@selector(invocationOperation)  object:nil];
        //  开始执行操作
        [invocationOperation  start];
}

-  (void)invocationOperation  {
NSLog(@"NSInvocationOperation包含的任务，没有加入队列========%@",  [NSThread  currentThread]
}
```

打印结果如下，得到结论：程序在主线程执行，没有开启新线程。

这是因为NSOperation多线程的使用需要配合队列NSOperationQueue，后面会讲到NSOperationQueue的使用。

```
1      NSInvocationOperation包含的任务，没有加入队列========{number  =  1,  name  =  main}
```

• NSBlockOperation的使用

把任务放到NSBlockOperation的block中，然后start。

```
-  (void)testNSBlockOperation  {
        //  把任务放到block中
        NSBlockOperation  *blockOperation  =  [NSBlockOperation  blockOperationWithBlock:^{
  NSLog(@"NSBlockOperation包含的任务，没有加入队列========%@",  [NSThread  currentThread]);
        }];

        [blockOperation  start];
}
```

执行结果如下，可以看出：主线程执行，没有开启新线程。

同样的，NSBlockOperation可以配合队列NSOperationQueue来实现多线程。

```
1      NSBlockOperation包含的任务，没有加入队列========{number  =  1,  name  =  main}
```

但是NSBlockOperation有一个方法addExecutionBlock:，通过这个方法可以让NSBlockOperation实现多线程。

```
-  (void)testNSBlockOperationExecution  {
 NSBlockOperation  *blockOperation  =  [NSBlockOperation  blockOperationWithBlock:^{
    NSLog(@"NSBlockOperation运用addExecutionBlock主任务========%@",  [NSThread  currentThrea
 }];

[blockOperation  addExecutionBlock:^{
NSLog(@"NSBlockOperation运用addExecutionBlock方法添加任务1========%@",[NSThread  currentThre
}];

[blockOperation  addExecutionBlock:^{
NSLog(@"NSBlockOperation运用addExecutionBlock方法添加任务2========%@",[NSThread  currentThre
```

```
}];
[blockOperation  addExecutionBlock:^{
NSLog(@"NSBlockOperation运用addExecutionBlock方法添加任务3========%@",[NSThread  currentThre
        }];
[blockOperation  start];
}
```

执行结果如下，可以看出，NSBlockOperation创建时block中的任务是在主线程执行，而运用addExecutionBlock加入的任务是在子线程执行的。

```
NSBlockOperation运用addExecutionBlock========{number  =  1,  name  =  main}
addExecutionBlock方法添加任务1========{number  =  3,  name  =  (null)}
addExecutionBlock方法添加任务3========{number  =  5,  name  =  (null)}
addExecutionBlock方法添加任务2========{number  =  4,  name  =  (null)}
```

- 运用继承自NSOperation的子类

首先我们定义一个继承自NSOperation的类，然后重写它的main方法，之后就可以使用这个子类来进行相关的操作了。

```
/*******************"WHOperation.h"************************/

#import  @interface  WHOperation  :  NSOperation

@end


/*******************"WHOperation.m"************************/

#import  "WHOperation.h"

@implementation  WHOperation

-  (void)main  {
        for  (int  i  =  0;  i  <  3;  i++)  {
                NSLog(@"NSOperation的子类WHOperation======%@",[NSThread  currentThread]);
        }
}

@end


/*****************回到主控制器使用WHOperation*********************/

-  (void)testWHOperation  {
        WHOperation  *operation  =  [[WHOperation  alloc]  init];
        [operation  start];
}
```

运行结果如下，依然是在主线程执行。

```
1       SOperation的子类WHOperation======{number = 1, name = main}
2       NSOperation的子类WHOperation======{number = 1, name = main}
3       NSOperation的子类WHOperation======{number = 1, name = main}
```

所以，NSOperation是需要配合队列NSOperationQueue来实现多线程的。下面就来说一下队列NSOperationQueue。

No.3：队列NSOperationQueue

NSOperationQueue只有两种队列：主队列、其他队列。其他队列包含了串行和并发。

主队列的创建如下，主队列上的任务是在主线程执行的。

```
1       NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
```

其他队列（非主队列）的创建如下，加入到‘非队列’中的任务默认就是并发，开启多线程。

```
1       NSOperationQueue *queue = [[NSOperationQueue alloc] init];
```

注意：

1 非主队列（其他队列）可以实现串行或并行。

2 队列NSOperationQueue有一个参数叫做最大并发数：maxConcurrentOperationCount。

3 maxConcurrentOperationCount默认为-1，直接并发执行，所以加入到‘非队列’中的任务默认就是并发，开启多线程。

4 当maxConcurrentOperationCount为1时，则表示不开线程，也就是串行。

5 当maxConcurrentOperationCount大于1时，进行并发执行。

6 系统对最大并发数有一个限制，所以即使程序员把maxConcurrentOperationCount设置的很大，系统也会自动调整。所以把最大并发数设置的很大是没有意义的。

No.4：NSOperation + NSOperationQueue

把任务加入队列，这才是NSOperation的常规使用方式。

• addOperation添加任务到队列

先创建好任务，然后运用- (void)addOperation:(NSOperation *)op 方法来吧任务添加到队列中，示例代码如下：

```
- (void)testOperationQueue {
    // 创建队列，默认并发
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];

  // 创建操作，NSInvocationOperation
NSInvocationOperation *invocationOperation = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(invocationOperationAddOperation) object:nil];
// 创建操作，NSBlockOperation
NSBlockOperation *blockOperation = [NSBlockOperation blockOperationWithBlock:^{
      for (int i = 0; i < 3; i++) {
          NSLog(@"addOperation把任务添加到队列======%@", [NSThread currentThread]);
      }
}];

        [queue addOperation:invocationOperation];
        [queue addOperation:blockOperation];
}

- (void)invocationOperationAddOperation {
```

```
NSLog(@"invocationOperation===aaddOperation把任务添加到队列====%@", [NSThread currentThrea
}
```

运行结果如下，可以看出，任务都是在子线程执行的，开启了新线程！

```
invocationOperation===addOperation把任务添加到队列===={number = 4, name = (null)}
addOperation把任务添加到队列======{number = 3, name = (null)}
addOperation把任务添加到队列======{number = 3, name = (null)}
addOperation把任务添加到队列======{number = 3, name = (null)}
```

• addOperationWithBlock添加任务到队列

这是一个更方便的把任务添加到队列的方法，直接把任务写在block中，添加到任务中。

```
- (void)testAddOperationWithBlock {
        // 创建队列，默认并发
        NSOperationQueue *queue = [[NSOperationQueue alloc] init];

        // 添加操作到队列
        [queue addOperationWithBlock:^{
                for (int i = 0; i < 3; i++) {
          NSLog(@"addOperationWithBlock把任务添加到队列======%@", [NSThread currentThread]
                }
        }];
}
```

运行结果如下，任务确实是在子线程中执行。

```
addOperationWithBlock把任务添加到队列======{number = 3, name = (null)}
addOperationWithBlock把任务添加到队列======{number = 3, name = (null)}
addOperationWithBlock把任务添加到队列======{number = 3, name = (null)}
```

• 运用最大并发数实现串行

上面已经说过，可以运用队列的属性maxConcurrentOperationCount(最大并发数)来实现串行，值需要把它设置为1就可以了，下面我们通过代码验证一下。

```
- (void)testMaxConcurrentOperationCount {
        // 创建队列，默认并发
        NSOperationQueue *queue = [[NSOperationQueue alloc] init];

        // 最大并发数为1，串行
        queue.maxConcurrentOperationCount = 1;

        // 最大并发数为2，并发
//        queue.maxConcurrentOperationCount = 2;

        // 添加操作到队列
        [queue addOperationWithBlock:^{
                for (int i = 0; i < 3; i++) {
        NSLog(@"addOperationWithBlock把任务添加到队列1======%@", [NSThread currentThread])
                }
        }];
        // 添加操作到队列
```

```
        [queue  addOperationWithBlock:^{
                for  (int  i  =  0;  i  <  3;  i++)  {
        NSLog(@"addOperationWithBlock把任务添加到队列2======%@",  [NSThread  currentThread])
                }
        }];

        //  添加操作到队列
        [queue  addOperationWithBlock:^{
                for  (int  i  =  0;  i  <  3;  i++)  {
        NSLog(@"addOperationWithBlock把任务添加到队列3======%@",  [NSThread  currentThread])
                }
        }];
}
```

运行结果如下，当最大并发数为1的时候，虽然开启了线程，但是任务是顺序执行的，所以实现了串行。

你可以尝试把上面的最大并发数变为2，会发现任务就变成了并发执行。

```
addOperationWithBlock把任务添加到队列1======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列1======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列1======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列2======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列2======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列2======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列3======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列3======{number  =  3,  name  =  (null)}
addOperationWithBlock把任务添加到队列3======{number  =  3,  name  =  (null)}
```

No.5：NSOperation的其他操作

- 取消队列NSOperationQueue的所有操作，NSOperationQueue对象方法

```
-  (void)cancelAllOperations
```

- 取消NSOperation的某个操作，NSOperation对象方法

```
-  (void)cancel
```

- 使队列暂停或继续

```
//  暂停队列
[queue  setSuspended:YES];
```

- 判断队列是否暂停

```
-  (BOOL)isSuspended
```

暂停和取消不是立刻取消当前操作，而是等当前的操作执行完之后不再进行新的操作。

No.6：NSOperation的操作依赖

NSOperation有一个非常好用的方法，就是操作依赖。可以从字面意思理解：某一个操作（operation2）依赖于另一个操作（operation1），只有当operation1执行完毕，才能执行operation2，这时，就是操作依赖大显身手的时候了。

```
-  (void)testAddDependency  {
        //  并发队列
        NSOperationQueue  *queue  =  [[NSOperationQueue  alloc]  init];
        //  操作1
```

```
        NSBlockOperation *operation1 = [NSBlockOperation blockOperationWithBlock:^{
                for (int i = 0; i < 3; i++) {
                        NSLog(@"operation1======%@", [NSThread currentThread]);
                }
        }];
        // 操作2
        NSBlockOperation *operation2 = [NSBlockOperation blockOperationWithBlock:^{
    NSLog(@"****operation2依赖于operation1，只有当operation1执行完毕，operation2才会执行****
                for (int i = 0; i < 3; i++) {
                  NSLog(@"operation2======%@", [NSThread currentThread]);
                }
        }];
        // 使操作2依赖于操作1
        [operation2 addDependency:operation1];
        // 把操作加入队列
        [queue addOperation:operation1];
        [queue addOperation:operation2];
}
```

运行结果如下，操作2总是在操作1之后执行，成功验证了上面的说法。

```
operation1======{number = 3, name = (null)}
operation1======{number = 3, name = (null)}
operation1======{number = 3, name = (null)}
****operation2依赖于operation1，只有当operation1执行完毕，operation2才会执行****
operation2======{number = 4, name = (null)}
operation2======{number = 4, name = (null)}
operation2======{number = 4, name = (null)}
```

152.多线程通信

说明：在 1 个进程中，线程往往不是孤立存在的，多个线程之间需要经常进行通信

线程间通信的体现

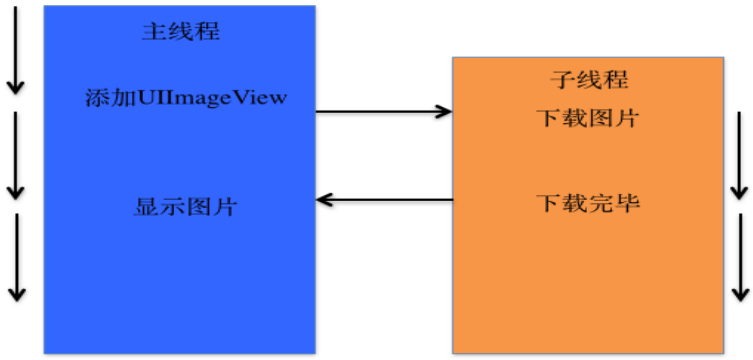
1 个线程传递数据给另 1 个线程

在 1 个线程中执行完特定任务后，转到另 1 个线程继续执行任务

线程间通信常用方法

```
-(void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg waitUntilDone:(BOOL)wait;
-(void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(id)arg waitUntilDone:(BOOL)wait;
```

线程间通信示例 - 图片下载



代码讲解

```
#import "YYViewController.h"
@interface YYViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *iconView;
@end

@implementation YYViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
 {

// 在子线程中调用download方法下载图片
[self performSelectorInBackground:@selector(download)
withObject:nil];
 }



 -(void)download
 {
    //1.根据URL下载图片

    //从网络中下载图片
    NSURL *urlstr=[NSURL URLWithString:@"fdsf"];
```

```
    //把图片转换为二进制的数据

    NSData *data=[NSData dataWithContentsOfURL:urlstr];//这一行操作会比较耗时

    //把数据转换成图片

    UIImage *image=[UIImage imageWithData:data];

    //2.回到主线程中设置图片

    [self performSelectorOnMainThread:@selector(settingImage:) withObject:image waitUntilDone:NO];
 }



 //设置显示图片
 -(void)settingImage:(UIImage *)image
 {
    self.iconView.image=image;
 }

@end
```

代码2：

```
#import "YYViewController.h"
#import <NSData.h>

@interface YYViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *iconView;
@end

@implementation YYViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
 }
```

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
// 在子线程中调用download方法下载图片
// 将当前的逻辑转到后台线程去执行
[self performSelectorInBackground:@selector(download)
withObject:nil];
}

-(void)download
{
//1.根据URL下载图片

//从网络中下载图片
NSURL *urlstr=[NSURL URLWithString:@"fdsf"];

//把图片转换为二进制的数据

NSData *data=[NSData dataWithContentsOfURL:urlstr];//这一行操作会比
较耗时

//把数据转换成图片
UIImage *image=[UIImage imageWithData:data];

//2.回到主线程中设置图片

//第一种方式
//设置 YES ，代表等待当前线程执行完毕 NO 表示不等
//[self performSelectorOnMainThread:@selector(settingImage:)
withObject:image waitUntilDone:NO];

    //第二种方式
    //    [self.imageView performSelector:@selector(setImage:)
onThread:[NSThread mainThread] withObject:image waitUntilDone:NO];

    //第三种方式
    [self.iconView
performSelectorOnMainThread:@selector(setImage:) withObject:image
```

```
waitUntilDone:NO];
 }


 //设置显示图片
 //-(void)settingImage:(UIImage *)image
 //{
 //    self.iconView.image=image;
 //}

@end
```

2.可以指定线程通讯

```
  //支持自定义线程通信执行相应的操作
    NSThread * thread = [[NSThread alloc]init];
  [thread start];
  //当我们需要在特定的线程内去执行某一些数据的时候，我们需要指定某一个线程操作
  [self performSelector:@selector(dothing:) onThread:thread withObject:nil
  waitUntilDone:YES];
  //支持自定义线程通信执行相应的操作
  NSThread * thread = [[NSThread alloc]initWithTarget:self
  selector:@selector(testThread) object:nil];
  [thread start];
  //当我们需要在特定的线程内去执行某一些数据的时候，我们需要指定某一个线程操作
  [self performSelector:@selector(dothing:) onThread:thread withObject:nil
  waitUntilDone:YES];
```

GCD 线程间的通讯

```
//开启一个全局队列的子线程通讯
 dispatch_async(dispatch_get_global_queue(0, 0), ^{
//1. 开始请求数据
 //...
 // 2. 数据请求完毕
//我们知道 UI 的更新必须在主线程操作，所以我们要从子线程回调到主线程
dispatch_async(dispatch_get_main_queue(), ^{
            //我已经回到主线程更新
      });
 });
```

//开启一个自定义的子线程通讯

```
//自定义队列，开启新的子线程     dispatch_queue_t    custom_queue =
```

```
dispatch_queue_create("concurrentqueue", DISPATCH_QUEUE_CONCURRENT);
for (int i=0; i<10; i++) {
dispatch_async(custom_queue, ^{
 NSLog(@"## 并行队列 %d ##",i);
 //数据更新完毕回调主线程 线程之间的通信
dispatch_async(dispatch_get_main_queue(), ^{
            NSLog(@"### 我在主线程 通信 ##");
        });
    });
 }

//线程延迟调用 通信
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(5.0 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
   NSLog(@"## 在主线程延迟 5 秒调用 ##");
});
```

153.tableViewDelegate 和 UItableDataSource 的区别

1 顾名思义，dataSource 意思为数据源，delegate 意为代理，其内部包含很多方法。

UITableView 需要显示数据，则数据源（datasource）可以给他提供数据从而显示出来，其会向数据源查询一共有多少行数据以及显示什么数据等，如果没有设置数据源的，那么 UITableView 就不会产生任何数据，没有作用。同时，遵守 UITableViewDataSource 协议的都可以是数据源，所以添加协议从而提供数据源。

同时 UITableView 也要设置对象，以防在 UITableView 触发事件时可以做出处理，比如选中某一行和某一列。遵守 UITableViewDataSource 协议的都可以是代理对象。

2.上边提到的 UITableViewDataSource 协议中常用方法：

（1）.设置行高

```
-(CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return 55;
}
```

（2）.选中 cell 时触发

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

（3.）设置 tableViewCell 的编辑样式(插入/删除)

```
-(UITableViewCellEditingStyle)tableView:(UITableView*)tableView
editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath
```

（4.）设置当点击编辑按钮时上面显示的文字,如显示删除

```
-(NSString*)tableView:(UITableView*)tableView
titleForDeleteConfirmationButtonForRowAtIndexPath:(NSIndexPath*)indexPath
```

NS_AVAILABLE_IOS(3_0) { return @"删除"; }
3. （5）.设置 cell 移动的位置
4. -(NSIndexPath*)tableView:(UITableView*)tableView
targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath*)sourceIndexPath
toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath

总结：tableViewDataSource 是提供数据源的 tableViewDelete 是提供交互的方法

153. 使用**mas_remakeConstraints** 并不能立即生效 需要添加**setNeedsUpdateConstraints**（约束需要更新）、然后**updateConstraintsIfNeeded**、**layoutIfNeeded** 才能生效。
参考：**https://blog.csdn.net/u011415099/article/details/53160595**
https://blog.csdn.net/woaifen3344/article/details/50114415

app 性能优化

卡顿优化、内存优化、网络优化、代码优化等方面
1. 卡顿优化
可能原因：布局嵌入的层次太深 2. feed流量加载的数据较大 3. 耗时操作添加在了主线程 4. 如果是tableView 请看tableView的性能优化
2. 内层优化：
1. 内层泄露 2. 图片相关(图片过大或者使用了动图)
3. 网络优化：
使用网关进行街口组合
4. 代码优化

ios高性能编程

1. 内层 最小的内层平均值和缝制
2. 耗电量 高效的算法和数据结构
3. 初始化时间 app再启动时花费的时间
例如：app启动时可能包含操作：
1>. 检查版本更新
2>. 初始化三方 地图 环信(可能还有登录) 分享 统计
3>. 游客身份可登录下 需要获取用户的信息
4>. 其他业务接口
方案：后果就是首页界面感觉很卡，明明都是在子线程发送的数据请求。为什么会这样样呢。
1. 即使是cpu的多核在能够并发，但是cpu在同一个时间段内只能执行一个任务。线程太多cpu需要在线程间切换，也是耗性能的 2. 这是应为那么多接口请求的数据可能都需要需要刷新首页的界面相关视图。好几次连续的刷新，也会使界面很卡，对于自己的接口尽量进行组合，可使用网关的模式，为什么显示首页界面会非常卡呢。
4. 执行速度
耗时的操作放在子线程
5. 响应速度
耗时的算法和业务逻辑尽量转到后台

6. 本地存储 使用正确的存储方式
7. 互操作性
8. 网络环境 在网络状态不好时，给出合理的提示
9. 数据刷新：数据刷新时 优美的动画
10. 多用户支持，是单点登录还是多用户登录
11. 安全 使用高效的加密方式
12. 崩溃：尽量使用 自动崩溃解析品台
13. 应用性能分析 采样 埋点

https://blog.csdn.net/kangguang/article/details/78325792 ios 高性能编程

https://www.cnblogs.com/wanghuaijun/p/7302303.html 数据结构

https://blog.csdn.net/wishfly/article/details/7370277 runloop 讲解

https://blog.csdn.net/sinat_25921367/article/details/48318587 block 循环引用专将

154. 深拷贝和浅拷贝专题

1. NSString

先看 NSString 的深浅拷贝

```
NSString * string = @"hello world";
/** 浅拷贝 */
NSString * copyString = [string copy];
/** 深拷贝 */
NSMutableString * mutableCopyString = [string mutableCopy];
NSLog(@"\nstring = %p\ncopyString
= %p\nmutableCopyString= %p",string,copyString,mutableCopyString);
```

输出结果为:

```
string = 0x104cae7d0
copyString = 0x104cae7d0
mutableCopyString = 0x7fa400f04fd0
```

由此看出 NSString 中
- 浅拷贝：未产生新对象
- 深拷贝：产生新对象

2. NSSMutableString

再看 NSSMutableString 的深浅拷贝

```
NSMutableString * mString = [NSMutableString stringWithString:@"hello world"];
/**  浅拷贝 */
NSString * copyMString = [mString copy];
/**  深拷贝 */
NSMutableString * mutableCopyMString = [mString mutableCopy];
NSLog(@"\nmString = %p\ncopyMString
= %p\nmutableCopyMString= %p",mString,copyMString,mutableCopyMString);
```

输出结果为：

```
mString = 0x7ffe42467130
copyMString = 0x7ffe42473b90
mutableCopyMString = 0x7ffe42402a50
```

由此看出 NSSMutableString 中
- 浅拷贝：产生新对象
- 深拷贝：产生新对象

备注

```
NSString 中，stringWithString 已被弃用，直接更改为赋值
NSMutableString 中，只可用 stringWithString 或 stringWithFormat，而无法直接赋值
```

在 OC 中，stringWithFormat 会新申请一片空间并初始化字符串，所以每一个用 stringWithFormat 方法得到的字符串其指针都是不相同的。

而 stringWithString 是通过浅拷贝的方式得到字符串，浅拷贝只拷贝指针不拷贝对象，因此指针与内容相同。

另外

快速初始化(initWithString)是首先根据一定的方法(此方法和 NSSet(集合)中的存放对象的方法一样都是 hash 算法)在内存中查找是否已经存在了这样的一个对象，若是存在则放回此对象的指针，若不存在，则根据一定的方法找到一片内存空间存放对象，并返回指针。

3. 以下原理相同

```
NSArray;
NSMutableArray;
NS*;
NSMutable*;
```

即：

源对象类型	拷贝方法	副本对象类型	是否产生了新对象	拷贝类型
NSString	copy	NSString	NO	浅拷贝(指针拷贝)
	mutableCopy	NSMutableString	YES	深拷贝(内容拷贝)
NSMutableString	copy	NSString	YES	深拷贝(内容拷贝)
	mutableCopy	NSMutableString	YES	深拷贝(内容拷贝)
NSArray	copy	NSArray	NO	浅拷贝(指针拷贝)
	mutableCopy	NSMutableArray	YES	深拷贝(内容拷贝)
NSMutableArray	copy	NSArray	YES	深拷贝(内容拷贝)
	mutableCopy	NSMutableArray	YES	深拷贝(内容拷贝)
NS*	copy	NS*	NO	浅拷贝(指针拷贝)
	mutableCopy	NSMutable*	YES	深拷贝(内容拷贝)

源对象类型	拷贝方法	副本对象类型	是否产生了新对象	拷贝类型
NSMutable*	copy	NS*	YES	深拷贝(内容拷贝)
	mutableCopy	NSMutable*	YES	深拷贝(内容拷贝)

通过以上这个实例我们可以对面试官提出的深拷贝和浅拷贝的问题 进行更加完善的回答

对NS 开头的对象使用copy 不会产生新的对象 是浅拷贝

使用MutableCopy 会产生新的对象 是深拷贝

对NSMutable 开头的对象使用copy 会产生新的对象 是深拷贝

使用MutableCopy 会产生新的对象 是深拷贝