# iOS面试题集锦

## 1.Objective-C堆和栈的区别？

* 管理方式

栈由编译器自动管理；堆则由程序员手动管理，容易发生内存泄露。

* 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。

## 2. 内存管理语义@property

|  |  |
| --- | --- |
| assign | setter方法直接赋值，不进行任何retain操作，为了解决原类型与循环引用问题 |
| retain | setter方法对参数进行release旧值,再retain新值 |
| 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。 |
| strong | 定义“拥有”关系，setter方法对参数进行release旧值再retain新值。 |
| weak | 定义“非拥有”关系，setter方法既不会release旧值，也不retain新值。对象销毁时，属性值会清空（nil out）。 |
| unsafe\_unretained | 与assign相同，但它适用于“对象类型”，表达一种“非拥有”关系，对象销毁时，属性值不会自动清空。 |

- (void)setName:(NSString \*)name

{

if (\_name != name) {

[\_name release];

\_name = [name retain];

}

}

- (void)setName:(NSString \*)name

{

if (\_name != name) {

[name retain];

[\_name release];

\_name = name;

}

}

内存管理原则：

1. 只要调用了alloc、copy、new方法产生了一个新对象，都必须在最后调用一次release或者autorelease
2. 只要调用了retain，都必须在最后调用一次release或者autorelease
3. @property如果用了copy或者retian，就需要对不再使用的属性做一次release操作
4. 如果用了ARC，另外讨论

## 3. 写一标准宏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))

## 4.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。

## 5. HTTP协议中，POST和GET的区别是什么？

|  |  |  |
| --- | --- | --- |
|  | GET | POST |
| 语义 | 用于信息获取的请求 | 可能修改服务器资源的请求 |
| 安全 | 方法提交数据不安全，数据置于请求行，客户端地址栏可见; | 方法提交数据安全，数据置于消息主体内，客户端不可见 |
| 数据大小 | 方法提交的数据大小有限 | 方法提交的数据大小没有限制 |
| 缓存 | 请求可以被缓存，在保留的历史记录中 | 不会被缓存 |

更多参考：<https://www.zhihu.com/question/31640769?rf=37401322>

## 6. UIView与CLayer有什么区别？

两者最大的区别是，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，所有的界面元素都继承于它。它本身是CoreAnimation实现的，然而真正的绘图部分是由CALayer来完成的。

每个UIView都存在一个CALayer主层，而UIView本身更像是一个CALayer管理器。当视图与图层一起时，视图为图层提供事件处理，而图层为视图提供显示内容，即View作为CALayer的CALayerDelegate，UIView可有多个CALayer。

CALayer核心动画与UIView动画的区别：UIView封装的动画执行完毕之后不会反弹。即如果是通过CALayer核心动画改变layer的位置状态，表面上看虽然已经改变了，但是实际上它的位置是没有改变的。

* 每个 UIView 内部都有一个 CALayer 在背后提供内容的绘制和显示，并且 UIView 的尺寸样式都由内部的 Layer 所提供。两者都有树状层级结构，layer 内部有 SubLayers，View 内部有 SubViews.但是 Layer 比 View 多了个AnchorPoint
* 在 View显示的时候，UIView 做为 Layer 的 CALayerDelegate,View 的显示内容由内部的 CALayer 的 display
* CALayer 是默认修改属性支持隐式动画的，在给 UIView 的 Layer 做动画的时候，View 作为 Layer 的代理，Layer 通过 actionForLayer:forKey:向 View请求相应的 action(动画行为)
* layer 内部维护着三分 layer tree,分别是 presentLayer Tree(动画树),modeLayer Tree(模型树), Render Tree (渲染树),在做 iOS动画的时候，我们修改动画的属性，在动画的其实是 Layer 的 presentLayer的属性值,而最终展示在界面上的其实是提供 View的modelLayer
* 两者最明显的区别是 View可以接受并处理事件，而 Layer 不可以

UIView->UIKit

CALayer->QuartCore

## 7. CFSocket使用有哪几个步骤

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；

## 8. Core Foundation中提供了哪几种操作Socket的方法

CFNetwork 、 CFSocket 和 BSD Socket 。

## 9. 类别的作用？继承和类别在实现中有何区别？

类别可以在不获悉，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。

类别主要有 3 个作用： (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。

继承可以增加，修改方法，并且可以增加属性。

## 10. SDWebImage内部实现过程

1. 入口 setImageWithURL:placeholderImage:options: 会先把 placeholderImage 显示，然后 SDWebImageManager 根据 URL 开始处理图片。
2. 进入 SDWebImageManager-downloadWithURL:delegate:options:userInfo:，交给 SDImageCache 从缓存查找图片是否已经下载 queryDiskCacheForKey:delegate:userInfo:.
3. 先从内存图片缓存查找是否有图片，如果内存中已经有图片缓存，SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo: 到 SDWebImageManager。
4. SDWebImageManagerDelegate 回调 webImageManager:didFinishWithImage: 到 UIImageView+WebCache 等前端展示图片。
5. 如果内存缓存中没有，生成 NSInvocationOperation 添加到队列开始从硬盘查找图片是否已经缓存。
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 notifyDelegate:。
7. 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。
8. 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 imageCache:didNotFindImageForKey:userInfo:。
9. 共享或重新生成一个下载器 SDWebImageDownloader 开始下载图片。
10. 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。
11. connection:didReceiveData: 中利用 ImageIO 做了按图片下载进度加载效果。
12. connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。
13. 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。
14. 在主线程 notifyDelegateOnMainThreadWithInfo: 宣告解码完成，imageDecoder:didFinishDecodingImage:userInfo: 回调给 SDWebImageDownloader。
15. imageDownloader:didFinishWithImage: 回调给 SDWebImageManager 告知图片下载完成。
16. 通知所有的 downloadDelegates 下载完成，回调给需要的地方展示图片。
17. 将图片保存到 SDImageCache 中，内存缓存和硬盘缓存同时保存。写文件到硬盘也在以单独 NSInvocationOperation 完成，避免拖慢主线程。
18. SDImageCache 在初始化的时候会注册一些消息通知，在内存警告或退到后台的时候清理内存图片缓存，应用结束的时候清理过期图片。
19. SDWI 也提供了 UIButton+WebCache 和 MKAnnotationView+WebCache，方便使用。
20. SDWebImagePrefetcher 可以预先下载图片，方便后续使用。

## 11. iOS数据持久化存储方案有哪些？

* plist属性列表存储（如NSUserDefaults）
* 文件存储（如二进制数据写入文件存储，通过NSFileManager来操作将下载起来的二进制数据写一篇文件中存储）
* NSKeydeArchiver归档存储，常见的是自动化归档/解档处理，想要学习如何通过runtime实现自动化归档/解档，可以阅读文章： [***学习通过runtime实现自动化归档/解档***](http://www.henishuo.com/runtime-archive-unarchive-automaticly/)
* 数据库SQLite3存储（如FMDB、Core Data）

## 12. 沙盒的目录结构是怎样的？各自一般用于什么场合？

* Application：存放程序源文件，上架前经过数字签名，上架后不可修改
* Documents: 保存应⽤运行时生成的需要持久化的数据,iTunes同步设备时会备份该目 录。例如,游戏应用可将游戏存档保存在该目录
* tmp: 保存应⽤运行时所需的临时数据,使⽤完毕后再将相应的文件从该目录删除。应用 没有运行时,系统也可能会清除该目录下的文件。iTunes同步设备时 不会备份该目录
* Library/Caches: 保存应用运行时⽣成的需要持久化的数据,iTunes同步设备时不会备份 该目录。⼀一般存储体积大、不需要备份的非重要数据，比如网络数据缓存存储到Caches下
* Library/Preference: 保存应用的所有偏好设置，如iOS的Settings(设置) 应⽤会在该目录中查找应⽤的设置信息。iTunes同步设备时会备份该目录

## 13. 常见的出现内存循环引用的场景有哪些？

* 定时器（NSTimer）：NSTimer经常会被作为某个类的成员变量，而NSTimer初始化时要指定self为target，容易造成循环引用（self->timer->self）。 另外，若timer一直处于validate的状态，则其引用计数将始终大于0，因此在不再使用定时器以后，应该先调用invalidate方法
* block的使用：block在copy时都会对block内部用到的对象进行强引用(ARC)或者retainCount增1(非ARC)。在ARC与非ARC环境下对block使用不当都会引起循环引用问题， 一般表现为，某个类将block作为自己的属性变量，然后该类在block的方法体里面又使用了该类本身，简单说就是self.someBlock = Type var {[self dosomething];或者self.otherVar = XXX;或者\_otherVar = …};出现循环的原因是：self->block->self或者self->block->\_ivar（成员变量）
* 代理（delegate）：在委托问题上出现循环引用问题已经是老生常谈了，规避该问题的杀手锏也是简单到哭，一字诀：声明delegate时请用assign(MRC)或者weak(ARC)，千万别手贱玩一下retain或者strong，毕竟这基本逃不掉循环引用了！

## 14. TCP和UDP的区别是什么？

TCP（传输控制协议/Transmisition Control Protocol）：面向连接、传输可靠（保证数据正确性，保证数据顺序传输）、用于传输大量数据(流模式)、速度慢，建立连接需要开销较多(时间，系统资源)。

UDP（用户数据报协议/User Datagram Protocol）：面向非连接、传输不可靠、用于传输少量数据(数据包模式)、速度快，传输的是报文。

## 15. MD5和Base64的区别是什么，各自使用场景是什么？

|  |  |  |
| --- | --- | --- |
|  | MD5 | Base64 |
| 区别 | 是一种不可逆的摘要算法，用于生成摘要，无法逆着破解得到原文。常用的是生成32位摘要，用于验证数据的有效性。比如，在网络请求接口中，通过将所有的参数生成摘要，客户端和服务端采用同样的规则生成摘要，这样可以防篡改 | 属于加密算法，是可逆的，经过encode后，可以decode得到原文。在开发中，有的公司上传图片采用的是将图片转换成base64字符串，再上传。在做加密相关的功能时，通常会将数据进行base64加密/解密。 |
| 使用场景 | 下载文件时，通过生成文件的摘要，用于验证文件是否损坏。 | 常用于邮件、http加密、登录的用户名和密码加密等； |

## 16.Runtime

runtime is everything between you each function call.

就是系统在运行的时候的一些机制，其中最主要的是消息机制。

Oc在三个不同的层面与运行时系统进行交互：

OC源程序；

* 通过NSObject类中定义的方法；
* 直接调用运行时函数；
* 大多数情况下，运行时系统仅仅在幕后自动工作，主要用于编写和编译OC源程序；

作用：

* 在程序运行过程中, 动态创建一个类(比如KVO的底层实现)
* 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
  + objc\_setAssociatedObject/ objc\_getAssociatedObject
* 遍历一个类的所有成员变量(属性)\所有方法
  + class\_copyPropertyList

## 17.进程与线程

进程：具有一定独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的一个独立单位；

特征：独立性：独立存在的实体，拥有自己独立资源和私有地址空间；

动态性：进程是一个正在系统中活动指令集合，具有生命周期和各种状态；

并发性：多个进程可在单处理器并发执行；

线程：线程是进程的一个实体，是进程执行单元，是程序中独立、并发的执行流，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源，只拥有在运行中必不可少的资源（寄存器、栈等），一个线程可以创建和撤销另一个线程；

关系：

1.一个线程属于一个进程，一个进程可以拥有多个线程；

2.资源分配给进程，统一进程内的所有线程共享该进程的所有资源；

3.线程在执行过程中需要协调同步。不同进程间的线程要利用消息通信的办法实现同步；

4.真正在处理机上运行的是线程；

5.线程是指进程内的一个可执行的单元，也是进程内可调度的实体；

区别：

1.调度：线程作为分派和调度的基本单元，进程作为拥有系统资源的基本单元；

2.并发性：不仅线程间可以并发，线程间也可以并发；

3.系统资源：进程拥有资源的一个独立单元，线程不拥有系统资源，可以访问进程的资源；

4.开销：进程的创建和撤销都比较大；

进程间同步机制：原子操作、信号量机制、自旋锁、分布式系统

死锁原因：资源竞争、进程推进顺序非法

死锁的四个必要条件：互斥、请求保持、不可剥夺、环路

死锁处理：鸵鸟策略、预防策略、检测与解除死锁

## 18.事件和响应者链

事件：UIEvent事件对象，记录事件产生的时刻和类型。

三类型：触摸、加速计、远程控制

响应者对象：继承自UIResponder的对象，才能接受并处理事件，称之为“响应者对象”

触摸事件：touchesBegin\touchesMoved\touchesEnded\touchesCanceled

UITouch:一根手指对应一UITouch对象，保存跟手指相关的信息，触摸位置、时间、阶段。

UITouch方法：

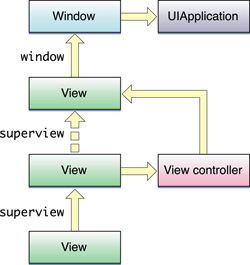
* + (CGPoint)locationInView
  + (CGPoint)previousLocationInView

事件的产生和传递：

* 发生触摸时间后，系统会将该事件加入到UIApplication管理的事件队列中；
* UIApplication会从事件队列中取出最前面的事件，并将事件分发下去，先发送的事件给应用程序的主窗口（keyWindow）；
* 主窗口会在视图层次结构中找到一个合适的视图来处理触摸事件；
* 找到合适的视图控件后，调用该控件的touches方法来处理；
* 处理touches方法，是将该事件顺着响应者链条向上传递，将事件交给上一个响应者处理。

响应者链表示一系列响应者对象，事件被交由第一个响应者处理，如果第一响应者不处理，事件便沿着响应者链向上传递，交给下一个响应者。

View->ViewController->Window->Application->丢弃事件



## 19.打包静态库

## 20. 写一个NSString类的实现

意思是:建立对象需要两个步骤 1分配内存 2初始化

1分配内存,要sent  alloc  or allocWithZone:  message to the object’s class. 也就是常见的[Class alloc].或是不常见的[Class allocWithZone]

2初始化.要调用init的方法进行初始化,各种各样的要或不要参数的init方法都算.

以上建立对象的方式,返回值要送进 对象自动管理池

+ (id) stringWithCString: (const char\*)nullTerminatedCString

encoding: (NSStringEncoding)encoding

{

NSString \*obj;

obj = [self allocWithZone: NSDefaultMallocZone()];

obj = [obj initWithCString: nullTerminatedCString encoding: encoding];

return AUTORELEASE(obj);

}

## 21.XML解析

* DOM：Document Object Model文档对象模型：读入整个文档，将文档转换为模型常驻内存的树状——DOM树，然后获取节点及节点的父子关系

缺点：DOM常驻内存，开销大；

有点：可修改树节点，随机访问任一节点；

* SAX：Simple API for XML，通过事件处理方法实现对XML的访问。在文档开始、元素开始、文本和文档结束时，就会向外发送一次事件，通过监听获取XML的信息；

## 22.iOS网络请求

（1）NSURLConnection可根据URL来加载异步请求，其实现NSURLConnectionDelegate方法来处理异步加载过程中的事件；

（2）创建NSURLRequest对象，该对象代表对远程服务器的请求，该请求包括请求的URL、缓存策略、超时时长等；

（3）NSMutableURLRequest

（4）代码

NSString\* str = @"http://www.crazyit.org/ethos.php";

totalData = [[NSMutableData alloc] init];

// 以指定NSString创建NSURL对象

NSURL\* url = [NSURL URLWithString:str];

// 创建NSURLRequest对象

// NSURLRequest\* request = [NSURLRequest requestWithURL:url];

// 通过这种方式创建的NSURLRequest可以指定缓存策略、超时时长

NSURLRequest\* request = [NSURLRequest requestWithURL:url

cachePolicy:NSURLRequestReloadIgnoringLocalCacheData timeoutInterval:5];

// 以指定URL、delegate创建连接、发送请求

NSURLConnection\* conn = [NSURLConnection connectionWithRequest:request

delegate:self];

其代理方法：

// 当服务器系响应生成时激发该方法。

-(void)connection:(NSURLConnection \*)connection

didReceiveResponse:(NSURLResponse \*)response

{}

// 每次读取服务器响应的数据时，都会激发该方法。

// 对于一个请求而言，服务器数据可能要分几次才能读取，因此该方法将会被触发多次。

// 如果程序需要将这些数据转换成字符串，建议使用NSMutableData来收集这些数据、然后整体转换

-(void)connection:(NSURLConnection \*)connection didReceiveData:(NSData\* )data

{}

// 当连接服务器出现错误时激发该方法。可通过error获取错误信息

-(void)connection:(NSURLConnection \*)connection didFailWithError:(NSError \*)error

{}

// 当数据load完成时激发该方法，对于每次请求，该方法只会被激发一次。

-(void)connectionDidFinishLoading:(NSURLConnection \*)connection

{}

## 23.JSON解析

## 24.layoutSubView与drawRect

（1）layoutSubView调用时机

设置view 的frame;

UIScrollView的滚动；

旋转Screen；

改变view的大小；

直接调用setLayoutSubviews/setNeedsLayout

（2）drawRect方法调用时机

在loadView->viewDidLoad后调用；

sizeToFit;

setNeedsDisplay/setNeedsDisplay:(CGRect)invalidRect

## 25.UIView继承体系

UIButton->UIControl->UIView->

UILabel->UIView->UIResponder->NSObject

UITableView->UIScrollView->UIView->

UISwitch->UIControl->

## 26.同步监视器

线程同步：任何时刻只能有一个线程可以获得对同步监视器的锁定，当同步代码块执行完成后，该线程会释放对同步监视器的锁定。

虽然Objective-C允许使用任何对象作为同步监视器，但想一下同步监视器的目的——阻止两个线程对同一个共享资源的进行并发访问，因此通常推荐使用可能被并发访问的共享资源充当同步监视器。

可变类的线程安全是以降低程序的运行效率作为代价的，为了减少线程安全所带来的负面影响，程序可以采用如下策略：

》不要对线程安全类的所有方法都进行同步，只对那些会改变竞争资源（竞争资源也属于共享资源）的方法进行同步。例如上面的FKAccount类中的accountNo属性就无须同步，所以程序只对draw方法进行同步控制；

》如果可变类有两种运行环境：单线程环境和多线程环境，则应该为该可变类提供两种版本——线程不安全版本和线程安全版本。在单线程环境中使用线程不安全版本以保证性能，在多线程环境中使用线程安全版本；

释放对同步监视器的锁定：

》当前线程的同步代码块结束；

》当线程在同步代码块中遇到goto、return终止了该代码块、该方法继续执行时；

》当线程在同步代码块出现错误；

@synchronized(obj)

{

//….  
}

## 27.同步锁

NSLock是控制多个线程对共享资源进行访问的工具，同步锁提供了对共享资源的独立访问，每次只能有一个线程对NSLock对象加锁，线程开始访问共享资源之前应先获得NSLock对象。

NSLock \*lock = [[NSLock alloc]init];

[lock lock];

//…..

[lock unlock];

## 28.NSCondition控制线程通信

当线程在系统内运行时，线程的调度具有一定的透明性，程序通常无法准确控制线程的轮换执行，但我们可以通过一些机制来保证线程协调运行，也就是处理线程之间的通信。

NSCondition实现NSLocking协议，因此可调用lock、unlock实现线程同步。同时NSCondition可以让那些已经锁定NSCondition对象却无法继续执行的线程释放NSCondition对象，NSCondition对象也可以唤醒其他处于等待状态的线程；

》wait:该方法导致当前线程一直等待，直到其他线程调用该NSCondition的signal方法或broadcast方法来唤醒该线程。

》signal:唤醒在此NSCondition对象等待的单个线程。如果所有线程都在改NSCondition对象上等待，则会选择唤醒其中一个线程。选择是忍一下的。只有当前线程放弃对该NSCondition对象的锁定后（使用wait方法），才可以执行被唤醒的线程。

》broadcast:唤醒在此NSCondition对象上等待的所有线程。只有当前线程放弃对该NSCondition对象的锁定后，才可以执行被唤醒的线程。

## 29.Core Data

Core Data是一个 ORM框架，可用于管理实体以及实体之间的关联关系的持久化。

Core Data底层的持久化存储方式可以是SQLite数据库，也可以是XML文档，甚至可以直接以内存作为持久化存储设备。

Core Data的核心概念是实体。实体是由Core Data管理（简称为被托管）的模型对象，它必须是NSManagedObject类或其子类的实例。整个应用的所有实体以及实体之间的关联关系被称为托管对象模型(NSManagedObjectModel)。

Core Data的核心对象是托管对象上下文（NSManagedObjectContext）,所有实体都处于托管对象上下文管理中，Core Data应用对实体所做的任何增、删、改、查操作都必须通过托管对象上下文来完成。

Core Data的托管对象上下文底层与持久化存储协调器（NSPersistentStoreCoordinator）衔接，持久化存储协调器负责管理底层的存储形式。

## 30.单例模式

+ (id)sharedManager {

static CustomClass \*shared = nil;

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

shared = [[self alloc]init];

});

return shared;

}

## 31.openUrl

openUrl可以发短信、发邮件、打开网页、打开其他程序。

[[UIApplication sharedApplication]openUrl:[NSURL URLWithString:@”sms://10086”]]

@”tel://10086”

## 32.程序启动过程与APP生命周期

程序启动过程：

打开程序-》main函数-》UIApplicationMain函数-》初始化UIApplication（创建和设置代理对象，开启事件循环）-》监听系统事件-》。。。

APP生命周期：

## 33.view的创建

loadView —NO—> storyboard —YES—> 根据storyboard创建

|

NO

|

nibname —YES—> 根据nibName去创建

|

NO

|

MJView.xib —YES—> 根据nibName去创建

|

NO

|

MJViewController.xib —YES—>根据ViewController.xib创建

|

NO

|

创建一个空的view

## 34.手势识别

敲击、捏合、拖拽、轻扫、旋转、长按

Tap、Pinch、Pan、Swipe、Rotation、LongPress

## 35.动画

动画：基本动画、关键帧动画、动画集合、转场动画

* CAAnimation

Duration、repeatCount、repeatDuration、fillMode（动画执行完之后是否保存其状态）、beginTime（动画延迟时间）

* CABasicAnimation

fromValue、toValue，其实改过之后，它们View的相应属性并未改变。

* CAKeyFrameAnimation

values:”关键帧”集合，即values的值集合；

path:CGPathRef/CGMutablePathRef，动画的路径；

keyTime:关键帧时间。

* CAAnimationGroup

多个动画集合；

* CATransition
* UIView动画

[UIView beginAnimations:@"" context:nil];

[UIView commitAnimations];

* Block动画

[UIView animateWithDuration:1.0 animations:^{

} completion:^(BOOL finished) {

}];

* UIImageView帧动画

UIImageView \*imageV = [[UIImageView alloc]init];

imageV.animationImages = @[];

## 36.Runloop

一般来讲，一个线程一次只能执行一个任务，执行完成后线程就会退出。如果我们需要一个机制，让线程能随时处理事件但并不退出，通常的代码逻辑是这样的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function loop() {      initialize();      do {          var message = get\_next\_message();          process\_message(message);      } while (message != quit);  } |

这种模型通常被称作 Event Loop。 Event Loop 在很多系统和框架里都有实现，比如 Node.js 的事件处理，比如 Windows 程序的消息循环，再比如 OSX/iOS 里的 RunLoop。实现这种模型的关键点在于：如何管理事件/消息，如何让线程在没有处理消息时休眠以避免资源占用、在有消息到来时立刻被唤醒。

所以，RunLoop 实际上就是一个对象，这个对象管理了其需要处理的事件和消息，并提供了一个入口函数来执行上面 Event Loop 的逻辑。线程执行了这个函数后，就会一直处于这个函数内部 "接受消息->等待->处理" 的循环中，直到这个循环结束（比如传入 quit 的消息），函数返回。

Main Run Loop负责处理用户相关的事件。UIApplication对象在程序启动时启动main run Loop，它处理事件和更新视图的界面。看Main Run Loop就知道，它是运行在程序的主线程上的。这样保证了接收到用户相关操作的事件是按顺序处理的。



* 一种异步执行代码的机制，不能并行执行任务；
* 主队列中，MainRunloop直接配合任务的执行，负责处理UI事件，计时器以及其他内核相关的事件；
* 主线程默认已经配置好Runloop，其他线程没有设置Runloop。

## 37.反射机制

Class class = NSClassFromString(@”classname”);

Student \*stu = [[class alloc]init];

NSString \*str = NSStringFromClass([Student Classs]);

SEL selector = NSSelectorFromString(@”selname”);

[stu performSelector:selector WithObject:nil];

NSString \*str = NSStringFromSelector(@selector(selname:));

## 38. #import 跟#include、@class有什么区别

* #import和#include都能完整地包含某个文件的内容，#import能防止同一个文件被包含多次
* @class仅仅是声明一个类名，并不会包含类的完整声明;@class还能解决循环包含的问题。在实现这个接口的实现类中，如果需要引用这个类的实体变量或者方法之类的，还是需要import在@class中声明的类进来。
* #import <> 用来包含系统自带的文件，#import “”用来包含自定义的文件

## 39. 语句NSString \*obj = [[NSData alloc] init]; ，编译时和运行时obj分别是什么类型？

编译时是NSString类型，运行时是NSData类型

## 40.viewController生命周期

view的加载过程：



view的卸载过程：





当你alloc并init了一个ViewController时，这个ViewController应该是还没有创建view的。ViewController的view是使用了lazyInit方式创建，就是说你调用的view属性的getter：［self view］。在getter里会先判断view是否创建，如果没有创建，那么会调用loadView来创建view。

代码组织：

ViewController生命周期中有那么多函数，一个重要问题就是什么代码该写在什么地方。

1. init里不要出现创建view的代码。良好的设计，在init里应该只有相关数据的初始化，而且这些数据都是比较关键的数据。init里不要掉self.view，否则会导致viewcontroller创建view。（因为view是lazyinit的）。
2. loadView中只初始化view， 一般用于创建比较关键的view如UITableViewController的tabView，UINavigationController的navgationBar，不可调用view的getter（在[super loadView]前），最好也不要初始化一些非关键的view。如果你是从nib文件中创建的viewController在这里一定要首先调用super的loadView方法，但建议不要重载这个方法。
3. viewDidLoad 这时候view已经有了，最适合创建一些附加的view和控件了。有一点需要注意的是，viewDidLoad会调用多次（viewcontroller可能多次载入view，参见图2）。
4. viewWillAppear 这个一般在view被添加到superview之前，切换动画之前调用。在这里可以进行一些显示前的处理。比如键盘弹出，一些特殊的过程动画（比如状态条和navigationbar颜色）。
5. viewDidAppear 一般用于显示后，在切换动画后，如果有需要的操作，可以在这里加入相关代码。
6. viewDidUnload 这时候viewController的view已经是nil了。由于这一般发生在内存警告时，所以在这里你应该将那些不在显示的view释放了。比如你在viewcontroller的view上加了一个label，而且这个label是viewcontroller的属性，那么你要把这个属性设置成nil，以免占用不必要的内存，而这个label在viewDidLoad时会重新创建。

## 41. OC中创建线程的方法是什么？如果指定在主线程中执行代码？如何延时执行代码？

1. 创建线程的方法

* NSThread
* NSOperationQueue和NSOperation
* GCD

1. 主线程中执行代码

* [self performSelectorOnMainThread: withObject: waitUntilDone:];
* [self performSelector: onThread:[NSThread mainThread] withObject: waitUntilDone:];
* dispatch\_async(dispatch\_get\_main\_queue(), ^{

});

1. 延时执行

* double delayInSeconds = 2.0;

dispatch\_time\_t popTime = dispatch\_time(DISPATCH\_TIME\_NOW,

(int64\_t)(delayInSeconds \* NSEC\_PER\_SEC));

dispatch\_after(popTime, dispatch\_get\_main\_queue(), ^(void){

});

* [self performSelector: withObject: afterDelay:];
* [NSTimer scheduledTimerWithTimeInterval: target: selector: userInfo: repeats:];

## 42.分类与扩展

分类作用；

分类对类进行模块化设计，解耦，可重用；

分类实现非正式协议；（即重写原有类中的方法）

1. 分类是有名称的，类扩展没有名称
2. 分类只能扩充方法和属性、不能扩充成员变量；类扩展可以扩充方法和成员变量
3. 类扩展一般就写在.m文件中，用来扩充私有的方法和成员变量（属性）

## 43.面向对象

三大特性是：封装,继承,多态

1. 所谓封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。封装是面向对象的特征之一，是对象和类概念的主要特性。 简单的说，一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。
2. 所谓继承是指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力；

1. 所谓多态就是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

五大基本原则：

1. 单一职责原则SRP(Single Responsibility Principle)

是指一个类的功能要单一，不能包罗万象。如同一个人一样，分配的工作不能太多，否则一天到晚虽然忙忙碌碌的，但效率却高不起来。

1. 开放封闭原则OCP(Open－Close Principle)

一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。比如：一个网络模块，原来只服务端功能，而现在要加入客户端功能，

那么应当在不用修改服务端功能代码的前提下，就能够增加客户端功能的实现代码，这要求在设计之初，就应当将服务端和客户端分开，公共部分抽象出来。

1. 替换原则(the Liskov Substitution Principle LSP)

子类应当可以替换父类并出现在父类能够出现的任何地方。

1. 依赖原则(the Dependency Inversion Principle DIP) 具体依赖抽象，上层依赖下层。假设B是较A低的模块，但B需要使用到A的功能，

这个时候，B不应当直接使用A中的具体类： 而应当由B定义一抽象接口，并由A来实现这个抽象接口，B只使用这个抽象接口：这样就达到

了依赖倒置的目的，B也解除了对A的依赖，反过来是A依赖于B定义的抽象接口。通过上层模块难以避免依赖下层模块，假如B也直接依赖A的实现，那么就可能造成循环依赖。一个常见的问题就是编译A模块时需要直接包含到B模块的cpp文件，而编译B时同样要直接包含到A的cpp文件。

1. 接口分离原则(the Interface Segregation Principle ISP)

模块间要通过抽象接口隔离开，而不是通过具体的类强耦合起来

## 44.SQLite与FMDB

增：INSERT INTO table\_name( column1, column2....columnN) VALUES ( value1, value2....valueN);

删：DELETE FROM table\_name WHERE {CONDITION};

改：UPDATE table\_name SET column1 = value1, column2 = value2....columnN=valueN

[ WHERE CONDITION ];

查：SELECT COUNT(column\_name) FROM table\_name WHERE CONDITION;

FMDB：

类

* FMDatabase：一个FMDatabase对象就代表一个单独的SQLite数据库

用来执行SQL语句

* FMResultSet：使用FMDatabase执行查询后的结果集
* FMDatabaseQueue：用于在多线程中执行多个查询或更新，它是线程安全的

增、删、改，均称为更新：- (BOOL)executeUpdate:(NSString\*)sql, ...

查：- (FMResultSet \*)executeQuery:(NSString\*)sql, ...

## 45.UITableView重用机制和优化策略

TableView的重用机制，为了做到显示和数据分离，iOS TableView的实现并且不是为每个数据项创建一个tableCell。而是只创建屏幕可显示最大个数的cell，然后重复使用这些cell，对cell做单独的显示配置，来达到既不影响显示效果，又能充分节约内容的目的。下面简要分析一下它的实现原理。

重用实现分析：

　　查看UITableView头文件，会找到NSMutableArray\* visiableCells，和NSMutableDictnery\* reusableTableCells两个结构。visiableCells内保存当前显示的cells，reusableTableCells保存可重用的cells。

　　TableView显示之初，reusableTableCells为空，那么tableView dequeueReusableCellWithIdentifier:CellIdentifier返回nil。开始的cell都是通过[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]来创建，而且cellForRowAtIndexPath只是调用最大显示cell数的次数。

　　比如：有100条数据，iPhone一屏最多显示10个cell。程序最开始显示TableView的情况是：

　　1. 用[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]创建10次cell，并给cell指定同样的重用标识(当然，可以为不同显示类型的cell指定不同的标识)。并且10个cell全部都加入到visiableCells数组，reusableTableCells为空。

　　2. 向下拖动tableView，当cell1完全移出屏幕，并且cell11(它也是alloc出来的，原因同上)完全显示出来的时候。cell11加入到visiableCells，cell1移出visiableCells，cell1加入到reusableTableCells。

　　3. 接着向下拖动tableView，因为reusableTableCells中已经有值，所以，当需要显示新的cell，cellForRowAtIndexPath再次被调用的时候，tableView dequeueReusableCellWithIdentifier:CellIdentifier，返回cell1。cell1加入到visiableCells，cell1移出reusableTableCells；cell2移出visiableCells，cell2加入到reusableTableCells。之后再需要显示的Cell就可以正常重用了。

　　所以整个过程并不难理解，但需要注意正是因为这样的原因：配置Cell的时候一定要注意，对取出的重用的cell做重新赋值，不要遗留老数据。

一些情况：

　　使用过程中，我注意到，并不是只有拖动超出屏幕的时候才会更新reusableTableCells表，还有：

　　1. reloadData，这种情况比较特殊。一般是部分数据发生变化，需要重新刷新cell显示的内容时调用。在cellForRowAtIndexPath调用中，所有cell都是重用的。我估计reloadData调用后，把visiableCells中所有cell移入reusableTableCells，visiableCells清空。cellForRowAtIndexPath调用后，再把reuse的cell从reusableTableCells取出来，放入到visiableCells。

　　2. reloadRowsAtIndex，刷新指定的IndexPath。如果调用时reusableTableCells为空，那么cellForRowAtIndexPath调用后，是新创建cell，新的cell加入到visiableCells。老的cell移出visiableCells，加入到reusableTableCells。于是，之后的刷新就有cell做reuse了。

注意：

1-重取出来的cell是有可能已经捆绑过数据或者加过子视图的，所以，如果有必要，要清除数据（比如textlabel的text）和remove掉add过的

子视图（使用tag）。

2-这样设计的目的是为了避免频繁的 alloc和delloc cell对象而已,没有多复杂。

3-设计的关键是实现cell和数据的完全分离

重点：避免重用机制出错

1.重用机制调用的就是dequeueReusableCellWithIdentifier这个方法，方法的意思就是“出列可重用的cell”，因而只要将它换为cellForRowAtIndexPath（只从要更新的cell的那一行取出cell），就可以不使用重用机制，因而问题就可以得到解决，但会浪费一些空间

2.为每个cell指定不同的重用标识符(reuseIdentifier)来解决。重用机制是根据相同的标识符来重用cell的，标识符不同的cell不能彼此重用。

优化策略：

* 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，遇到性能瓶颈时，可能就是突破口；
* 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）。
* 正确使用reuseIdentifier来重用Cells
* 尽量使所有的view opaque，包括Cell自身，尽量少用或不用透明图层
* 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

## 46.MVC



1. 模型对象封装了应用程序的数据，并定义操控和处理该数据的逻辑和运算。例如，模型对象可能是表示游戏中的角色或地址簿中的联系人。用户在视图层中所进行的创建或修改数据的操作，通过控制器对象传达出去，最终会创建或更新模型对象。模型对象更改时（例如通过网络连接接收到新数据），它通知控制器对象，控制器对象更新相应的视图对象。
2. 视图对象是应用程序中用户可以看见的对象。视图对象知道如何将自己绘制出来，并可能对用户的操作作出响应。视图对象的主要目的，就是显示来自应用程序模型对象的数据，并使该数据可被编辑。尽管如此，在 MVC 应用程序中，视图对象通常与模型对象分离。
3. 在应用程序的一个或多个视图对象和一个或多个模型对象之间，控制器对象充当媒介。控制器对象因此是同步管道程序，通过它，视图对象了解模型对象的更改，反之亦然。控制器对象还可以为应用程序执行设置和协调任务，并管理其他对象的生命周期。

控制器对象解释在视图对象中进行的用户操作，并将新的或更改过的数据传达给模型对象。模型对象更改时，一个控制器对象会将新的模型数据传达给视图对象，以便视图对象可以显示它。

## 47.链表与数组

## 48.函数指针与指针函数

## 49.自动释放池

autorelease就是自动释放，看上去很像ARC，但是更类似于C语言中的自动变量（局部变量）的特性。但是不同的是可以设定变量的作用域。

## 50.代理与通知

同：都能实现对象间的通信。

代理：针对一对一的关系，其具体实现是先定义一个协议，声明该协议需要实现的方法，然后指定某个视图或者控制器为代理，并实现协议中指定的方法；

通知：可以实现多对多的关系，有一个通知中心，通过通知中心来进行对象之间的相互通信，需要注意的是不用通知的时候需要从通知中心移除监听。

程序中如果过度使用通知，将会很难跟踪程序的运行。而代理只要通过合理的实现协议，很容易理解代码的运行流程。还有，在开发时，代理如果实现错误，Xcode会有错误提醒和警告。而通知，则只能在运行时才会发现错误。

## 51.

## 52.

## 53.

## 54.

## 55.

## 56.

## 57.

## 58.

## 58.

## 60.

## 61.

## 62.

## 63.

## 64.

## 65.

## 66.

## 67.

## 68.

## 69.

## 70.