

目录

目录	1
第三章 UITableView 相关.....	8
1、重用机制	8
2、数据源同步.....	9
1、 并发访问，数据拷贝	9
2、 串行访问.....	10
UI 事件传递&响应.....	10
1、CALayer 和 UIView 的区别和联系	10
2、介绍响应者链.....	11
3、响应者链实际应用	12
4、这里有好的示例.....	12
3-4 UI 图像显示原理	13
3-5 UI 卡顿&掉帧原因.....	13
滑动优化方案（基于 tableview 或者 scrollview 你有哪些优化方案需要 做）	13
3-6 UI 绘制原理&异步绘制.....	14
UIView 的绘制原理.....	14
系统绘制流程	15
异步绘制	15
3-7 UI 离屏渲染 相关面试问题&面试总结	15
什么是离屏渲染.....	16
何时会触发？	16
为何要避免	16
既然离屏渲染这么耗性能,为什么有这套机制呢?.....	16
总结:	16
第四章 Objective-C 相关	17
分类相关.....	17
什么是分类	17
我们用分类做了哪些事情？	17
特点:	17
分类中可以添加哪些内容	17
分类有多个，每个分类中有一个同名的方法，最终哪个会生效？	18
Category 的实现原理，以及 Category 为什么只能加方法不能加属性？	18
Category 为什么不能添加成员变量？	18
关联对象.....	18
是否可以给分类添加“成员变量”？<关联对象>.....	18
扩展（Extension）	20
一般用扩展做什么？	20

特点	20
分类和扩展的区别?	20
代理 (Delegate)	20
什么是代理?	20
我们在协议中可以定义什么?	20
代理的工作流程.....	21
注意规避循环引用	21
通知 (NSNotification)	21
特点	21
通知和代理的区别.....	21
什么是通知机制或者说如何实现通知机制?	22
KVO	22
什么是 KVO?	22
KVO 底层实现.....	22
通过 KVC 设置 value 时 KVO 能否生效? (能够生效)	23
为什么通过 KVC 设置 value 能够生效呢?	23
通过成员变量直接赋值 value 时 KVO 能否生效? (不能生效)	23
如何实现手动 KVO?	23
高频题目:	24
KVC.....	24
我们通过键值编码技术是否会破坏面向对象的这种思想呢?	24
valueForKey 系统实现流程	24
setValue: forKey: 实现流程.....	26
属性关键字	26
关于一个被 atomic 修饰的数组添加对象是没有办法保证线程安全的.....	26
weak 和 assign 的区别是什么?	26
深拷贝和浅拷贝.....	26
@property (copy) NSMutableArray *array;(这个有没有问题).....	27
MRC 下如何重写 retain 修饰变量的 setter 方法?	28
属性 readwrite, readonly, assign, retain, copy, nonatomic 各是什么作用, 在那种情况下用?	28
第五章、RunTime 相关.....	29
Runtime 数据结构.....	29
objc_object	29
objc_class	29
Class 类是否是对象呢?	30
isa 指针	30
cache_t.....	31
对象、类对象、元类对象.....	32
类对象和元类对象分别是什么? 类对象和元类对象之间有什么区别?	32
元类对象和根元类对象他们的 isa 指针都是指向根元类的。根元类对象的 superClass 指向的是根类对象。	32
消息传递	33

消息传递大致过程.....	34
方法缓存查找相关（缓存查找具体是怎样的流程和步骤）	34
当前类中查找	35
父类逐级查找	35
消息传递的机制.....	35
消息转发流程	36
Method-Swizzling.....	37
动态添加方法	38
动态方法解析	40
总结.....	40
[obj foo]和 objc_msgSend()函数之间有什么关系?	40
runtime 如何通过 Selector 找到对应的 IMP 地址的?	40
能否向编译后的类中增加实例变量?	40
能否向动态添加的类中增加实例变量?	40
第六章、内存管理	41
6-1 内存布局相关.....	41
6-2 内存管理方案相关.....	42
iOS 操作系统是怎样对内存进行管理的?	42
MRC.....	42
ARC.....	42
什么是 ARC?	42
ARC 和 MRC 有什么区别呢?	43
Retain 操作的时候系统格式怎样查找他的引用计数的?	43
alloc 实现.....	43
AutoreleasePool 的实现原理是怎样的?	43
AutoreasePool 为何可以嵌套使用?	43
什么是自动释放池，自动释放池的结构是什么?	43
双向链表.....	44
栈.....	44
循环引用	44
自循环引用	45
相互循环引用	45
多循环引用	46
如何破除循环引用?	46
具体的解决方案有哪些?	46
__weak 破解（避免产生循环引用）	47
__block 破解.....	47
__unsafe_unretained 破解.....	47
循环引用示例	47
NSTimer 的循环引用问题.....	48
总结.....	48
第七章．Block.....	49
7-1 Block 本质相关.....	49
什么是 block?.....	49

什么是 Block 调用?	49
7-2 Block 截获变量相关	49
关于 block 的截获特性你是否有了解, block 的截获变量特性又是怎么样的呢?	51
7-3 _block 修饰符相关	51
什么场景下使用_block 修饰符?	51
7-4 Block 内存管理相关	53
当我们将栈上的 block 进行 copy 操作之后, 假如在 MRC 情况下, 是否会引起内存泄露呢?	54
Block 的引用循环	54
总结	56
什么是 Block?	56
为什么 Block 会产生循环引用?	56
怎样理解 Block 截获变量的特性?	56
你都遇到过哪些循环引用? 你又是怎样解决的?	56
第八章、多线程	57
在 iOS 系统当中, 为我们提供了哪几种多线程的技术方案呢?	57
同步异步并行串行主要影响区别	57
GCD	58
同步/异步 和 串行/并发	58
Dispatch_sync(dispatch_get_main-queue())	58
Dispatch_async(dispatch_get_main-queue())	58
● 同步调用串行	59
● 同步并发	60
● 异步并发	61
线程栅栏 dispatch_barrier	61
怎样实现多读单写?	63
队列组 dispatch_group	64
使用 GCD 实现需求: A、B、C 三个任务并发, 完成后执行任务 D	65
NSOperation 相关	66
主要优点 (几种任务状态能说上来)	66
状态控制 (会问到怎样控制状态, 一下是回答)	66
系统怎样移除一个 isFinished=YES 的 NSOperation 的?	66
信号量	66
NSThread	68
NSThread 执行原理	68
锁	69
iOS 当中有哪些锁? 你使用过哪些锁, 是怎样使用的	69
@synchronized	69
atomic	69
OSSpinLock	69
NSLock	70

NSRecursiveLock 递归锁	70
信号量	70
总结	71
怎样用 GCD 来实现多读单写呢?	71
iOS 系统为我们提供的几种多线程技术各自的特点是怎样的?	71
第九章、RunLoop	72
RunLoop 本质相关	72
什么是 runloop?	72
Main 函数为什么能保持 runloop 一直的循环而不退出?	72
RunLoop 和 Model 和他所对应的 Source、Timer、Observer 是怎样的一个关系呢?	72
CommonMode	73
RunLoop 与 NSTimer	73
滑动 TableView 的时候我们的定时器还会生效吗?	73
RunLoop 与多线程	74
RunLoop 和线程的关系	74
总结:	74
什么是 RunLoop,他是怎样做到有事做事, 没事儿休息的?	74
RunLoop 与线程是怎样的关系?	74
如何实现一个常驻线程?	75
怎样保证子线程数据回来更新 UI 的时候不中断用户的滑动操作?	75
Main 函数为什么能保持 runloop 一直的循环而不退出?	75
第十章、HTTP	76
Charles 的抓包原理是怎样的?	76
GET 和 POST 方式的区别	76
从语法角度来看, 最直观的区别就是	76
而从语义的角度来看:	76
HTTPS 和 HTTP 的区别	77
对称加密 非对称加密	77
为什么要进行三次握手进行连接? (而不是两次呢?)	78
了解 DNS 吗?	78
DNS 存在哪些问题?	78
DNS 劫持问题:	78
DNS 劫持与 HTTP 的关系是怎样的?	79
怎样解决 DNS 劫持?	79
总结:	79
HTTP 中的 GET 和 POST 方式有什么区别?	79
从语法角度来看, 最直观的区别就是	79
而从语义的角度来看:	80
请简述 TCP 的慢开始过程?	80
TCP 和 UDP 有什么区别?	80
客户端怎样避免 DNS 劫持?	80
HTTPS 连接建立流程是怎样的?	80
tcp 流量控制, 依靠滑动窗口协议。说一下滑动窗口协议	81

请简述 TCP 的慢开始过程?	82
第十一章、设计模式	83
你都了解知道哪些设计原则, 以及你对这些职责有哪些了解?	83
责任链模式	83
桥接模式	83
适配器	84
单例设计模式	86
命令模式	86
总结:	87
请手写单例实现。	87
你都知道哪些设计原则, 请谈谈你的理解?	87
能否用一幅图简单的标识桥接模式的主体结构?	87
UI 事件传递机制是怎样实现的? 你对其中运用到的设计模式是怎样理解的? 《责任链模式》	87
第十三章、算法	87
二分法查找	87
冒泡	88
选择排序	89
插入排序	90
快速排序	91
字符串反转	91
链表反转	92
有序数组合并	92
第十四章、第三方框架	93
AFNetworking 框架及概要	93
AFNetworking 回答	94
SDWebImage 如何为 UIImageView 添加图片	94
Masonry 如何为视图添加约束	95
ReactiveCocoa 作用	95
其他	96
1、iOS 导航模式	96
2、iOS 中持久化方式有哪些?	96
3、iOS 单元测试框架有哪些?	96
4、iOS 你在项目中用过 runtime 吗? 举个例子	96
5、iOS 中实现数据交换的几种方法介绍以及其效率对比	96
6、app 启动过程 程序启动原理	97
7、杨辉三角	99
8、支付宝支付接入详情讲解	99
9、做过视频吗?	101
10、Object-c 的类可以多重继承么? 可以实现多个接口么?	102
11、#import 跟#include 又什么区别, @class 呢, #import<> 跟 #import""又什么区别?	102
12、常见的 object-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	102

13、id 声明的对象有什么特性?	102
14、描述一下 iOS SDK 中如何实现 MVC 的开发模式.....	102
15、我们说的 oc 是动态运行时语言是什么意思?	103
16、什么是推送消息?	103
17、Difference between frame and bounds?	104
18、沙盒机制介绍以及沙盒目录结构	104
19、什么是谓词?	105
20、在主线程更新 UI, 有什么好处?	105
21、ViewController 生命周期.....	105
22、堆和栈的区别	105
23、关键字 const 什么含义?	105
24、static 的作用?	105
25、对数组中的元素去重复.....	106
26、简单描述一下 XIB 与 Storyboards, 说一下他们的优缺点。	107
27、在 App 中混合 HTML5 开发 App 如何实现的。在 App 中使用 HTML5 的优缺点是什么?	108
28、Socket	108
29、XML 数据解析方式各有什么不同, JSON 解析有哪些框架?	108
30、ReactiveCocoa 作用	109
资和信公司	110
新增	114
1、 iOS 判断对象相等.....	114
2、关于 UIView 切圆角的两种方式.....	114
3、UILabel 的抗压缩、抗拉伸、以及控件的约束.....	115
4、__block 修饰的变量为什么能在 block 里面能改变其值	115
5、delegate 和 block 的区别	116
6、load 方法何时加载.....	116
7、线程间的通信? (下面这个理解下, 然后看情况加东西吧)	116
8、工厂设计模式	116
9、操作数据库线程同步问题.....	117
10、如何对定位和分析项目中影响性能的地方? 以及如何性能优化?	117
11、drawrect.....	117
12、block 原理	117
13、图文混排.....	118

第三章 UITableView 相关

1、重用机制

```
UserInfoTableViewCell *ucell = [tableView  
dequeueReusableCellWithIdentifier:@"UserInfoTableViewCell"];
```

这个就使用到了tableView的重用机制。

iOS设备的内存有限，如果用UITableView显示成千上万条数据，就需要成千上万个UITableViewCell对象的话，那将会耗尽iOS设备的内存。要解决该问题，需要重用UITableViewCell对象

重用原理：当滚动列表时，部分UITableViewCell会移出窗口，UITableView会将窗口外的UITableViewCell放入一个对象池中，等待重用。当UITableView要求dataSource返回UITableViewCell时，dataSource会先查看这个对象池，如果池中有未使用的UITableViewCell，dataSource会用新的数据配置这个UITableViewCell，然后返回给UITableView，重新显示到窗口中，从而避免创建新对象。

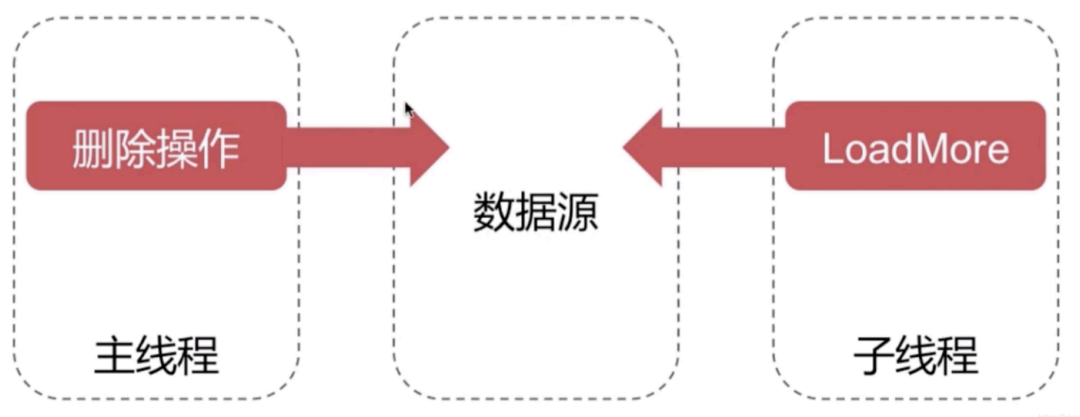
还有一个非常重要的问题：有时候需要自定义UITableViewCell(用一个子类继承UITableViewCell)，而且每一行用的不一定是同一种UITableViewCell，所以一个UITableView可能拥有不同类型的UITableViewCell，对象池中也会有很多不同类型的UITableViewCell，那么UITableView在重用UITableViewCell时可能会得到错误类型的UITableViewCell

解决方案：UITableViewCell有个NSString *reuseIdentifier属性，可以在初始化UITableViewCell的时候传入一个特定的字符串标识来设置reuseIdentifier(一般用UITableViewCell的类名)。当UITableView要求dataSource返回UITableViewCell时，先通过一个字符串标识到对象池中查找对应类型的UITableViewCell对象，如果有，就重用，如果没有，就传入这个字符串标识来初始化一个UITableViewCell对象

2、数据源同步

<https://blog.csdn.net/mchuajian/article/details/86708998>

如何解决tableview在多线程环境下的去修改或者访问他的数据源的同步问题。

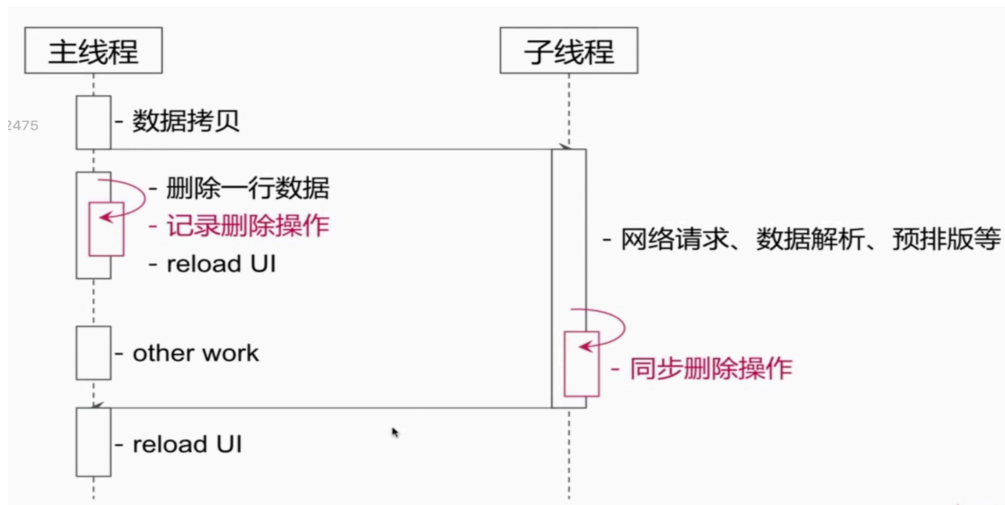


大家都知道，我们一般在主线程中刷新 UI，然后在子线程中去加载网络数据和数据解析，这时候，假如我们用户要在点击删除广告这一操作，这时候子线程又在加载数据（显然我们是在不同线程对同一资源做操作了），我们怎么通知子线程让他在显示数据的时候知道我们删除了这条广告呢。

1、 并发访问，数据拷贝

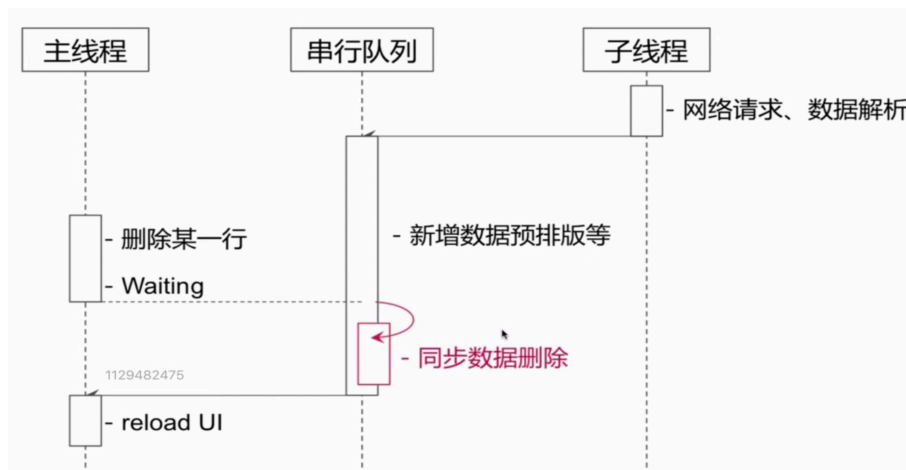
- 并发及多个线程都可以执行在同一段时间，不需要互相等待，主线程与用户互动，子线程加载 cell 所需要的网络数据以及预排版
- 解决方法：如图，主线程首先拷贝一份数据给子线程完成预排版，网络请求与数据解析(json xml 转化)，这时候如果主线程需要删除某些数据源操作，他就记录这条删除操作，在子线程完成各种加载操作后将这条操作与子线程

- 进行同步一下，然后再回到主线程刷新界面
- 缺点：可能需要拷贝大量数据，比较消耗内存



2、串行访问

- 创建一个 GCD 串行队列，主线程增删改操作需要等待子线程操作完成
- 解决方法：如图，我们首先使用 GCD 创建一个串行队列，子线程先加入队列完成网络加载操作，如果这时候主线程需要修改数据源，这个操作就要等待子线程完成才去进行(串行执行)
- 缺点：可能子线程的网络请求速度慢，主线程 UI 操作等待时间长



UI 事件
传递 &
响应

1、CALayer 和 UIView 的区别和联系

<https://www.jianshu.com/p/079e5cf0f014> (详解)

- UIView 主要是对显示内容的管理而 CALayer 主要侧重显示内容的绘制。

- 两者最明显的区别是 View 可以接受并处理事件，而 Layer 不可以。
用到了六大设计原则的单一职责原则(为什么区分 UIView 和 CALayer 的分工)。

UIView和CALayer

- UIView 为其提供内容，以及负责处理触摸等事件，参与响应链
- CALayer 负责显示内容contents

2、介绍响应者链

步骤 1、简单介绍

当用户点击屏幕，能都对用户交互动作事件进行响应的对象组成的链条，就是响应者链，它继承自 `NSResponder`，响应者链能够中断。

有些时候，touch 后系统通过 hit test 机制找到了触碰的 initial View，但是 initial view 并没有或者无法正常处理此次 Touch。这个时候，系统就会通过响应者链寻找下一个响应者，对此次 Touch 进行响应。

响应者链条的顺序是：initial View —>view Controller(如果存在) —>superview —>...—>rootView—>UIWindow —>UIApplication

步骤 2、进行升华

然后可以介绍一下 hit-test 机制：当用户点击屏幕，系统首先找到 `UIResponder`，然后将 touch 以 `UIEvent` 的方式加入到 `UIApplication` 事件队列中。`UIApplication` 从队列中取出最新的触摸事件分发传递到 `UIWindow` 进行处理，`UIWindow` 会通过 `hitTest:withEvent:` 方法寻找触碰点所在的视图，这个过程称之为 hit-test view。

hitTest 的顺序如下

1UIApplication -> UIWindow -> Root View -> ... -> subview

之后顶级视图会向当前视图的所有子视图发送 `hitTest:withEvent:` 消息，然后就是响应者链的过程。如果有 subview 的 `hitTest:withEvent:` 返回非空对象，处理结束

步骤 3、最后补充：

介绍自己如何认识响应者链或者响应者链中遇到的坑：

我入坑是因为在我的应用首页，TableVeiw 最下面的两个一直都是无法点击，根据我最初了解的响应者链，应该是从上到下响应，应该是可以的，后来具体的了解了 hit-test 机制，目标视图点击区域不在父视图的 Frame 上，所以点击没有效果。

以下下是视图没有响应的几个情况：

1.userInteractionEnabled=NO

2.hidden=YES;

3.alpha=0~0.01;

4.没有实现 touchesBegan:withEvent:方法，直接执行 touchesMove:withEvent:等方法；

5.目标视图点击区域不在父视图的 Frame 上 (superView 背景色为 clear Color 的时候经常会忽略这个问题)。

3、响应者链实际应用



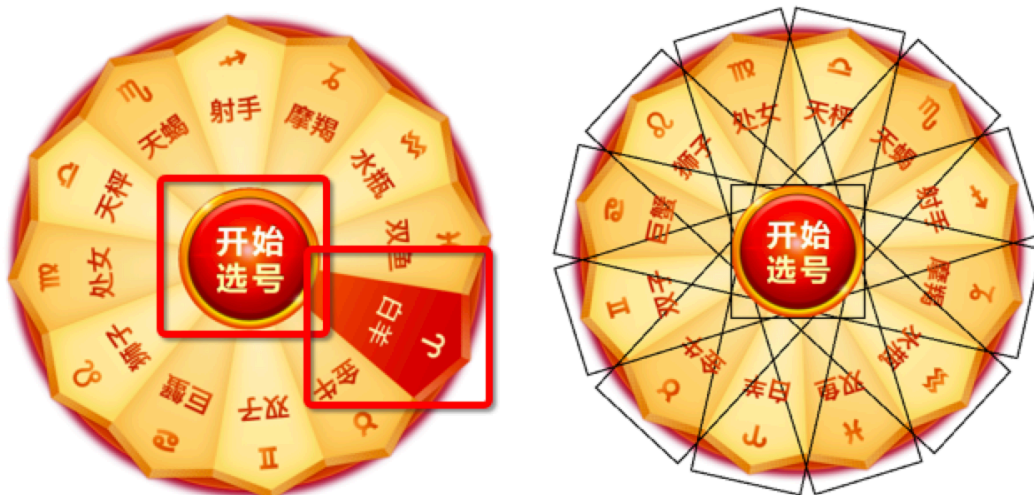
如果遇到上方的面试题：可以回答：

可以自定义 button, 然后重写 hitTest:withEvent 和 - (BOOL)pointInside:(CGPoint)point withEvent:(UIEvent*)event

主要是在 pointInside 方法中指定绘制可点击的区域即可

4、这里有好的示例

<https://www.jianshu.com/p/71379bf3405d>（这里有好的示例）



如何自定义按钮点击的有效区域呢，其实很简单，只要重写一个函数即可：

- (BOOL)pointInside:(CGPoint)point withEvent:(UIEvent*)event

按钮还是矩形的按钮，只是改变了按钮中响应区域而已。用到贝塞尔曲线

3-4 UI 图像显示原理

<https://blog.csdn.net/olsQ93038o99S/article/details/83511583>

1. CPU：计算视图 frame，图片解码，绘制纹理交给 GPU。
2. GPU：纹理混合，顶点变换，渲染到帧缓冲区。

3-5 UI 卡顿&掉帧原因

屏幕显示的时候，是由垂直同步信号(VSync)和水平同步信号(HSync)组成，按照每秒 60FPS 的刷帧率，每隔 16.7ms 就会有一次 VSync 信号，如果每一次 CPU 和 GPU 工作的总耗时超过 16.7ms，就会造成卡顿掉帧

滑动优化方案（基于 tableview 或者 scrollview 你有什么优化方案需要做）

<https://www.jianshu.com/p/a96b7dd7d3ad>

<减轻 CPU 和 GPU 的压力>

● CPU

1. 对象创建、调整、销毁减少 CPU 的消耗时间（不要创建无用对象或者临时增加对象）
2. 预排版（布局计算，文本计算）

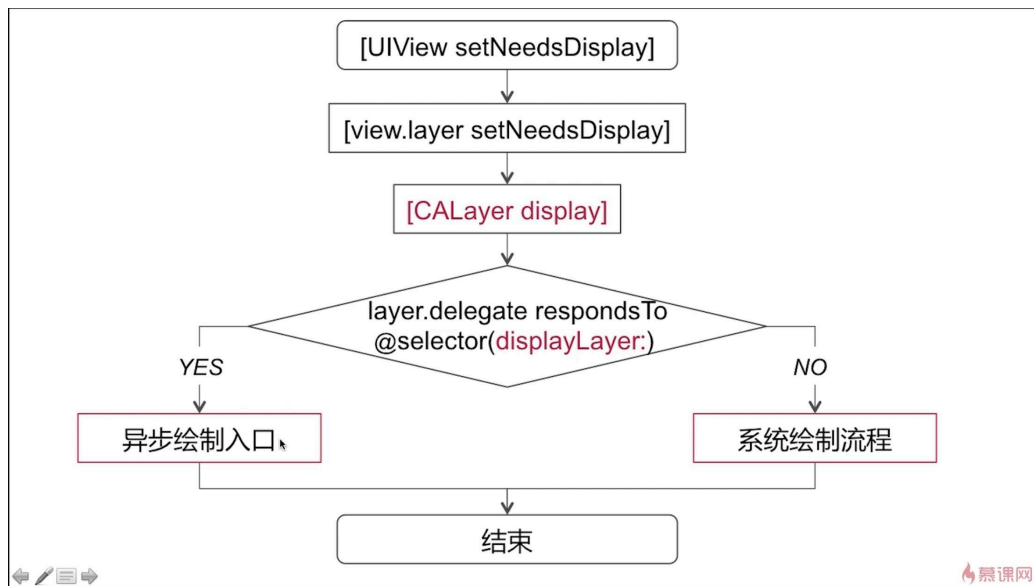
3. 预渲染（文本等异步绘制，图片编解码等）
 4. 尽量把耗时的操作放到子线程
 5. 对于 `tableview`，如果可以的话，使用局部更新
- GPU
 - 1. 文理渲染
 - 2. 视图混合
 - 3. 减少透明的视图（`alpha<1`），不透明的就设置 `opaque` 为 YES
 - 4. 尽量避免出现离屏渲染

3-6 UI 绘制原理&异步绘制

<https://blog.csdn.net/BlackWolfSky/article/details/84640703>
<https://blog.csdn.net/flover5724059/article/details/88727820>
<https://www.jianshu.com/p/ad7f7ea4ba67>

UIView 的绘制原理

调用 `UIView` 的 `setNeedsDisplay` 方法时，实际上并没有进行当前视图的绘制过程，而系统会立刻调用它所对应的 `layer` 的 `setNeedsDisplay` 方法，这相当于在 `layer` 上面打上了一个“脏标记”。而后在当前 `runLoop` 快要结束的时候，才会调用 `CALayer` 的 `display` 方法，然后就进入到当前视图真正的绘制流程。对于 `CALayer` 的 `display` 方法，在它的内部实现中会先判断它的 `delegate` 是否会响应 `displayLayer:`这个方法：如果不响应这个方法，那就进入系统的绘制流程。如果响应这个方法，那就进入视图的异步绘制流程。



系统绘制流程

只知道会调用我们所熟悉的 `drawRect:` 方法

异步绘制

1、假如我们在某一时机调用了【`view setNeedsDisplay`】这个方法，系统会在当前 `runloop` 将要结束的时候调用【`CALayer display`】方法，然后如果我们这个 `layer` 代理实现了 `displaylayer` 这个方法

2、然后切换到子线程去做位图的绘制，主线程可以去做其他的操作

3、首先通过 `CGBitmapContextCreate()` 创建一个位图上下文，然后通过 `CoregraphIC` API 可以做当前 UI 控件的一些绘制工作，最后再通过 `CGBitmapContextCreateImage()` 函数来生成一张 `CGImage` 图片

4、最后回到主线程来提交这个位图，设置 `layer` 的 `contents` 属性，这样就完成了一个 UI 控件的异步绘制

3-7 UI 离屏渲染 相关面试问题&面试总结

<https://www.jianshu.com/p/cff0d1b3c915>

<https://www.jianshu.com/p/6225d6796a66>

什么是离屏渲染

离屏渲染

- On-Screen Rendering

意为当前屏幕渲染，指的是GPU的渲染操作是在当前用于显示的屏幕缓冲区中进行

黑马199300

- Off-Screen Rendering

意为离屏渲染，指的是GPU在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作

何时会触发？

- 圆角设置的时候（当和 maskToBounds 一起使用的时候）。
- 设置阴影操作。
- 图层蒙版，为图层设置遮罩（layer.mask）
- 光栅化

为何要避免

在触发离屏渲染的时候，会创建新的渲染缓冲区，会有上下文切换，那么就会增加 GPU 的工作量，而增加了 GPU 的工作量，很有可能 CPU 和 GPU 工作加起来的总耗时超出了 16.7ms,那么就会导致 UI 的卡顿和掉帧。所以我们需要避免离屏渲染。

既然离屏渲染这么耗性能,为什么有这套机制呢？

有些效果被认为不能直接呈现于屏幕，而需要在别的地方做额外的处理预合成。图层属性的混合体没有预合成之前不能直接在屏幕中绘制，所以需要屏幕外渲染。屏幕外渲染并不意味着软件绘制，但是它意味着图层必须在被显示之前在一个屏幕外上下文中被渲染（不论 CPU 还是 GPU）。

总结：

- 系统的 UI 事件传递机制是怎样的？
- 使 UITableView 滚动更流畅的方案或思路都有哪些？
涉及到性能的优化，可以从 CPU 和 GPU 两方面回答。

- 什么事离屏渲染，为什么要离屏渲染。
- UIView 和 CALayer 之间的关系是怎样的？

第四章 Objective-C 相关

分类相关

什么是分类

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，它可以添加实例方法、类方法、协议、属性（不是实例变量），它是运行时决议的，它存在的意义是能够声明私有方法，然后就是将类的实现分散到多个不同的文件或多个不同的框架当中。然后还能把Framework的私有方法公开。

我们用分类做了哪些事情？

- 声明私有方法。
- 将类的实现分散到多个不同文件或多个不同框架中。
- 把 Framework 的私有方法公开。

特点：

- 运行时决议
- 可以为系统类添加分类（而扩展不可以）。

分类中可以添加哪些内容

- 实例方法
- 类方法（用到的不多）
- 协议
- 添加属性；但无法添加实例变量。

注意，在 category 中可以有属性（property），但是该属性只是生成了 getter 和 setter 方法的声明，并没有产生对应的实现，更不会添加对应的实例变量。如果想为实例对象添加实例变量，可以尝试使用关联引用技术。

分类有多个，每个分类中有一个同名的方法，最终哪个会生效？

最后编译的分类中的方法会生效。（因为编译完以后系统会倒叙的去执行方法，当找到方法后就返回该方法运行结果，所以最后编译的分类中的方法会生效）

- 分类添加的方法可以“覆盖”原类方法。
- 同名分类方法谁能生效取决于编译顺序。
- 名字相同的分类会引起编译报错。

Category 的实现原理，以及 Category 为什么只能加方法不能加属性？

分类的实现原理分类是 `category` 中的方法，属性，协议数据放在 `category_t` 结构体中，然后将结构体内的方法列表拷贝到类对象的方法列表中。

分类是由运行时决议的，不同分类中的同名方法谁最终生效取决于谁是最最后参与编译的，最后编译的分类中同名方法会生效。如果有和宿主同名的方法，那么分类里面的优先调用。

Category 为什么不能添加成员变量？

`Category` 可以添加属性，但是并不会自动生成成员变量及 `set/get` 方法。因为 `category_t` 结构体中并不存在成员变量。通过之前对对象的分析我们知道成员变量是存放在实例对象中的，并且编译的那一刻就已经决定好了。而分类是在运行时才去加载的。那么我们就无法再程序运行时将分类的成员变量中添加到实例对象的结构体中。因此分类中不可以添加成员变量。

关联对象

是否可以给分类添加“成员变量”？<关联对象>

当面试的时候首先要回答：能够为其添加“成员变量”，但是需要用到关联对象才能为其添加。

【关联对象的本质，或者怎么给分类添加成员变量---会这样问】

众所周知，`category` 中虽然可以添加属性，但是并不会生成带下划线的成员变量，也不会生成 `getter` 和 `setter` 的实现，我们可以通过关联对象这个技术为 `category`

中的属性添加 getter 和 setter,

```
// 关联对象
```

```
// 使用 objc_setAssociatedObject 函数可以给某个对象关联其他的对象。
```

```
void objc_setAssociatedObject(id object, const void *key, id value,
objc_AssociationPolicy policy)
```

```
// 获取关联的对象
```

```
// 使用 objc_getAssociatedObject 函数可以通过键来取出某个对象的关联对象。
```

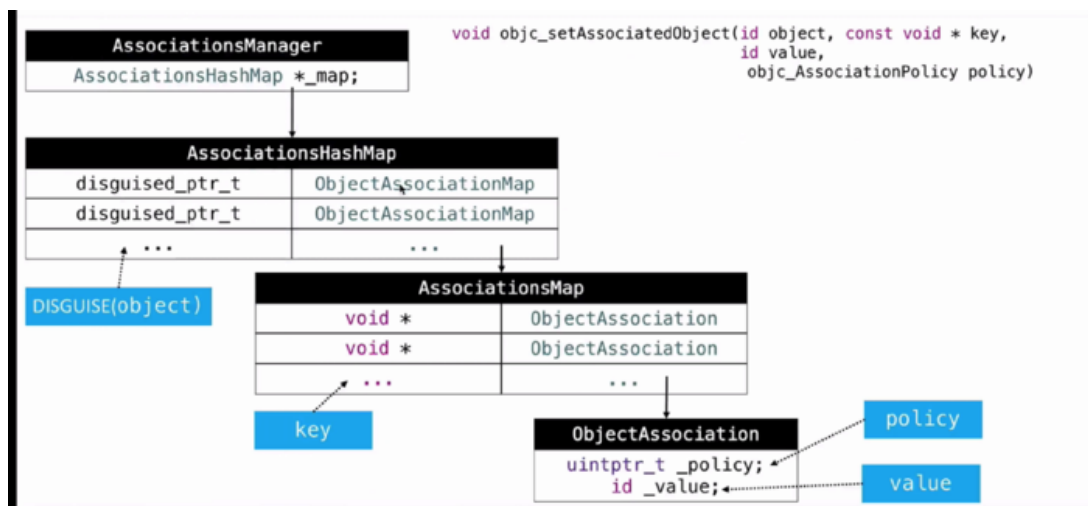
```
id objc_getAssociatedObject(id object, const void *key)
```

```
// 移除关联的对象
```

```
// 使用 objc_removeAssociatedObjects 函数可以移除某个对象身上的所有关联的对象。
```

```
void objc_removeAssociatedObjects(id object)
```

通过 runtime 的源码可以看出关联属性并没有添加到 `category_t`(分类)里边,运行时也不会合并到元类对象里边, 所以呢, 关联对象由 `AssociationsManager` 管理并在 `AssociationsHashMap` 存储。所有对象的关联内容都在同一个全局容器中。



扩展（Extension）

一般用扩展做什么？

- 声明私有属性
- 声明私有方法
- 声明私有成员变量

特点

- 编译时决议
- 只以声明的形式存在，多数情况下寄生于宿主类的.m 中。
- 不能为系统类添加扩展。

分类和扩展的区别？

- 分类是运行时决议的，扩展是编译时决议的。
- 分类可以有声明，有实现，而扩展只能有声明，实现直接是写在宿主类当中的。
- 分类可以为系统类添加分类，但是扩展不能为系统类添加扩展。
- 分类不能直接添加属性，扩展可以添加属性。另外扩展添加的方法是必须要实现的。

代理（Delegate）

什么是代理？

- 准确的说是一种软件设计模式。
- 在 iOS 当中以 **@protocol** 形式体现。
- 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，它的传递方式是一对一的。

我们在协议中可以定义什么？

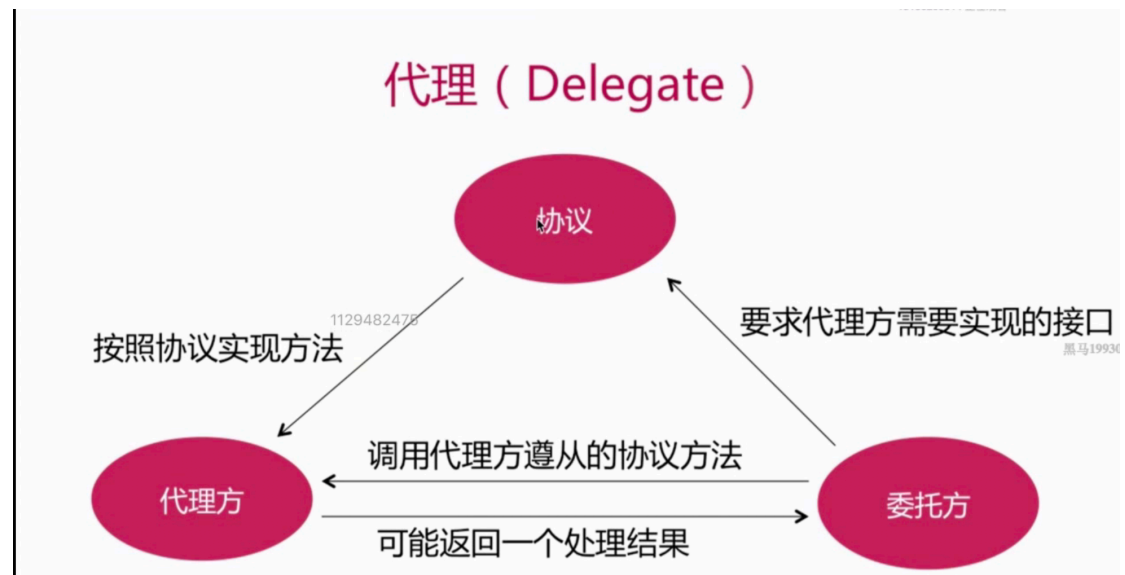
可以定义方法，也可以定义属性

我们在协议中声明的方法和属性，代理方必须要都实现吗？

不一定，如果被标识为 **required** 的，那么就必须要实现。

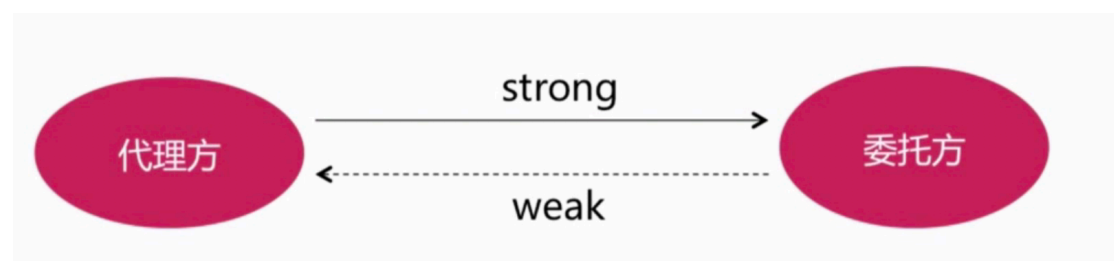
不实现，编译就会报错。如果是 **optional** 的，可以实现，也可以不实现。

代理的工作流程



注意规避循环引用

一般声明为 weak 以规避循环引用。



通知 (NSNotification)

特点

- 是使用 **观察者模式** 来实现的用于跨层（数据层、网络层、业务逻辑层等）传递消息的机制。
- 传递方式是一**对多**。

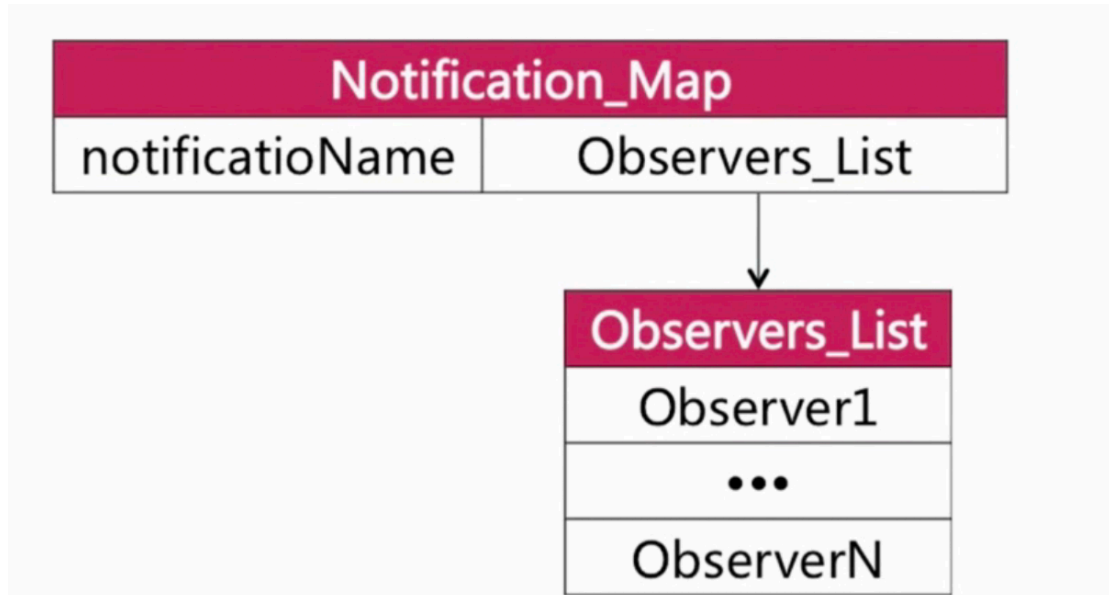
通知和代理的区别

- 代理是用 **代理模式** 实现的，而通知是用 **观察者模式** 实现的

- 通知的传递方式是一对多，代理是一对一。

什么是通知机制或者说如何实现通知机制？

（假如让你实现，你会怎么做）



KVO

什么是 KVO？

- KVO 是键值观察者模式，他提供了观察某一属性变化的方法，他是对 **观察者模式** 的又一实现。
- 使用了 isa 混写（**isa-swizzling**）来实现 KVO。

KVO 底层实现

- 我们调用 `[self addObserver:self forKeyPath:@"highlighted" options:0 context:nil];`
- 系统会自定义 **NSKVONotifying_A** 子类
- 重写 `setName`, 在内部恢复父类做法, 通知观察者
- 如何让外界调用自定义 A 类的子类方法, 修改当前对象的 isa 指针, 指向 **NSKVONotifying_A**

重写 set 方法内部实现：

```

//NSKVO notifying_A 的setter实现
- (void)setValue:(id)obj
{
    [self willChangeValueForKey:@"keyPath"];
    // 调用父类实现，也即原类的实现
    [super setValue:obj];
    [self didChangeValueForKey:@"keyPath"];
}

```

通过 KVC 设置 value 时 KVO 能否生效？（能够生效）

调用这个方法进行测试：[obj setValue:@2 forKey:@"value"];

为什么通过 KVC 设置 value 能够生效呢？

通过成员变量直接赋值 value 时 KVO 能否生效？（不能生效）

调用这个方法测试：[obj increase]; （括号里的话仅用于理解：成员变量赋值就是在这个方法内部为这个 value 赋值，没有调用 setter 方法，如果想实现就需要手动添加 KVO 的实现）

如何实现手动 KVO？

手动 KVO 就是调用了[self willChangeValueForKey:@""];
 之后又调用了[self didChangeValueForKey:@""];
 然后 didChangeValueForKey 内部实现就会触发 KVO 的回调。

- 使用setter方法改变值KVO才会生效。
- 使用setValue:forKey:改变值KVO才会生效。
- 成员变量直接修改需手动添加KVO才会生效。

高频题目：

什么是 KVO？

KVO 是如何通过 isa 机制实现的？

通过成员变量直接赋值 value 时 KVO 能否生效？（不能，如果要是能的话，就需要手动添加 KVO 的实现）。

KVC

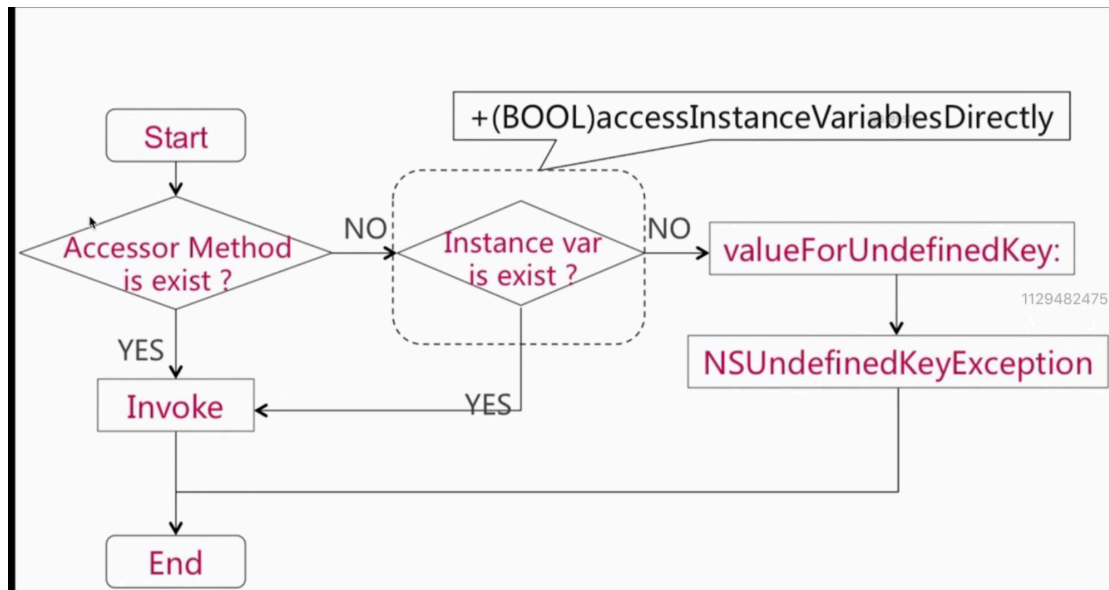
KVC 是苹果系统提供的键值编码技术。它是一种通过字符串标识间接的访问对象属性的一种机制。

- - (id) valueForKey: (NSString *) key
- - (id) setValue: (id) value forKey: (NSString *) key

我们通过键值编码技术是否会破坏面向对象的这种思想呢？

会的。因为他是通过字符串标识间接的访问对象属性，而不是通过调用存取方法。

valueForKey 系统实现流程



accessor method 是 get 方法， instance var 是实例变量

访问器（accessor method）方法判断规则：

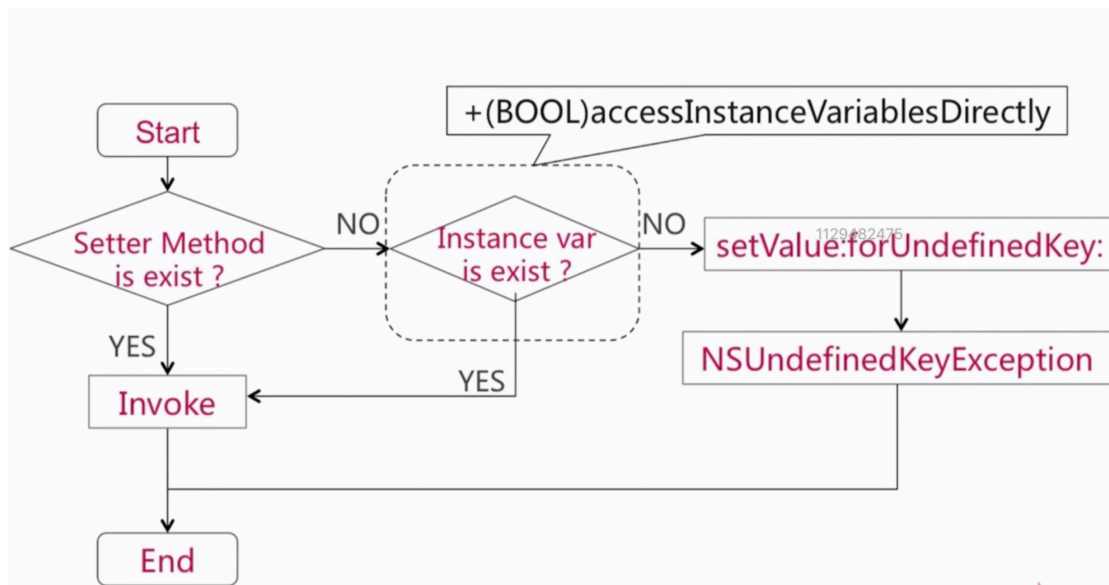
- <getKey>
- <key>
- <isKey>

实例变量判断规则：

- _key
- _isKey
- key
- isKey

文字描述：调用了 **valueForKey** 方法后，首先系统会去寻找访问器方法是否存在，如果存在，那么直接进行调用，然后结束调用流程。如果没有实现，那么就回去寻找实例变量是否存在(寻找实例变量，他会去遍历的寻找 **_key,key,_isKey,isKey** 是否存在)，如果存在，获取实例变量的值，然后结束调用流程。如果还不存在，就会调用当前实例的 **valueForUndefinedKey** 方法，然后抛出未定义 **key** 的异常。然后结束调用流程。

setValue: forKey: 实现流程



属性关键字

关于一个被 **atomic** 修饰的数组添加对象是没有办法保证线程安全的

对于一个数组的赋值和获取能够保证线程安全，但是对于删除和添加是不能保证线程安全的。

weak 和 assin 的区别是什么？

- weak 和 assin, weak 可以修饰对象, assin 既可以修饰对象, 也可以修饰基本数据类型。
- assin 修饰的对象在被释放之后, assin 指针仍然指向原对象的内存地址, 而 weak 修饰的对象被释放之后会 weak 指针是指为 nil 的。
- weak 一般用在代理对象上面, 或者用来解决循环强引用的问题。

深拷贝和浅拷贝

- 浅拷贝：只复制指向对象的指针，而不复制引用对象本身。

- 深拷贝：复制引用对象本身。
- 深拷贝会开辟新的内存空间，浅拷贝不会。
- 深拷贝不会影响引用计数，浅拷贝引用计数会增加

copy关键字

源对象类型	拷贝方式	目标对象类型	拷贝类型(深/浅)
mutable对象	copy	不可变	深拷贝
mutable对象	mutableCopy	可变	深拷贝
immutable对象	copy	不可变	浅拷贝
immutable对象	mutableCopy	可变	深拷贝

- 可变对象的 copy 和 mutableCopy 都是深拷贝。
- 不可变对象的 copy 是浅拷贝，mutableCopy 是深拷贝。
- copy 方法返回的都是不可变对象。

@property (copy) NSMutableArray *array;(这个有没有问题)

首先这样做可能会造成程序异常。因为

- 如果赋值过来的是 NSMutableArray，这个是深拷贝，copy 之后是 NSArray。
- 如果赋值过来的是 NSArray，这个是浅拷贝，copy 之后也是 NSArray。

由于这里写的是 NSMutableArray，所以可能导致用户在文件里面调用 NSMutableArray 的增加、删除方法，但是这时这个数组是不可变的，那么就会导致程序异常。

MRC 下如何重写 **retain** 修饰变量的 **setter** 方法？

```
@property (nonatomic, retain) id obj;

- (void)setObj:(id)obj
{
    1129482475
    if (_obj != obj) {
        [_obj release];
        _obj = [obj retain];
    }
}
```

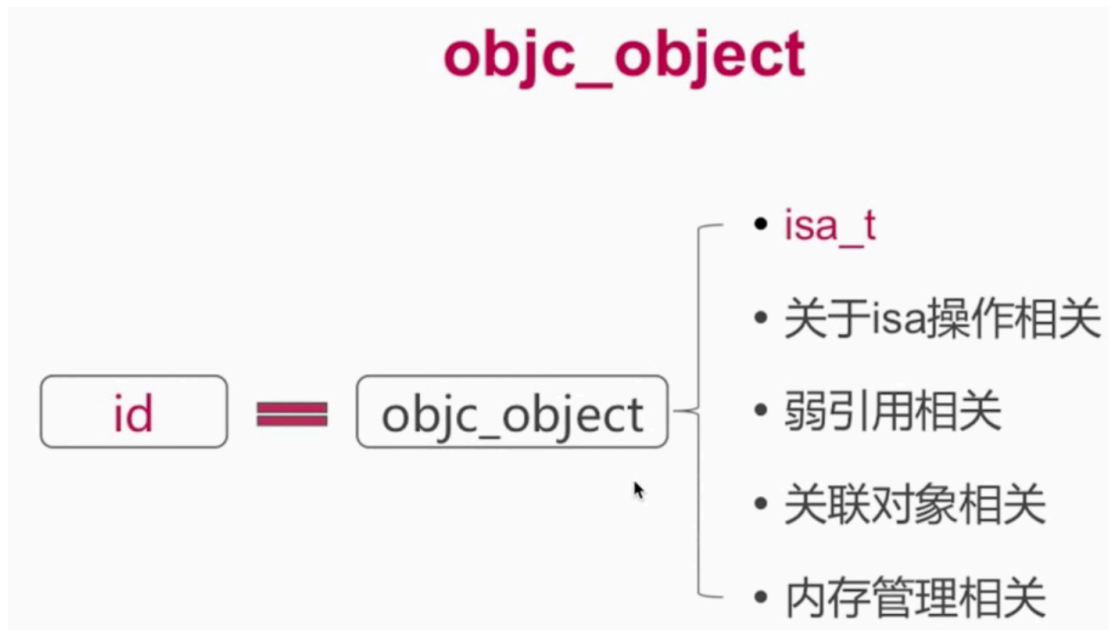
属性 **readwrite**, **readonly**, **assign**, **retain**, **copy**, **nonatomic** 各是什么作用，在那种情况下用？

- **readwrite** 是可读可写特性；需要生成 **getter** 方法和 **setter** 方法时
- **readonly** 是只读特性 只会生成 **getter** 方法 不会生成 **setter** 方法 ;不希望属性在类外改变
- **assign** 是赋值特性，**setter** 方法将传入参数赋值给实例变量；仅设置变量时；
- **retain** 表示持有特性，**setter** 方法将传入参数先保留，再赋值，传入参数的 **retaincount** 会+1；
- **copy** 表示拷贝特性，**setter** 方法将传入对象复制一份；需要完全一份新的变量时。
- **nonatomic** 非原子操作，决定编译器生成的 **setter** **getter** 是否是原子操作，**atomic** 表示多线程安全，一般使用 **nonatomic**

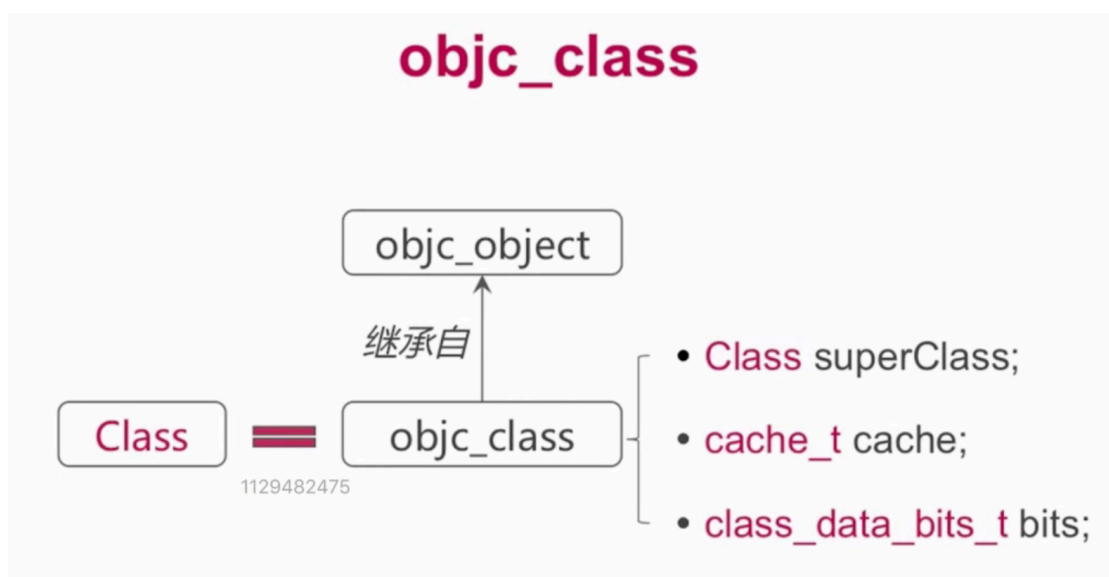
第五章、RunTime 相关

Runtime 数据结构

objc_object



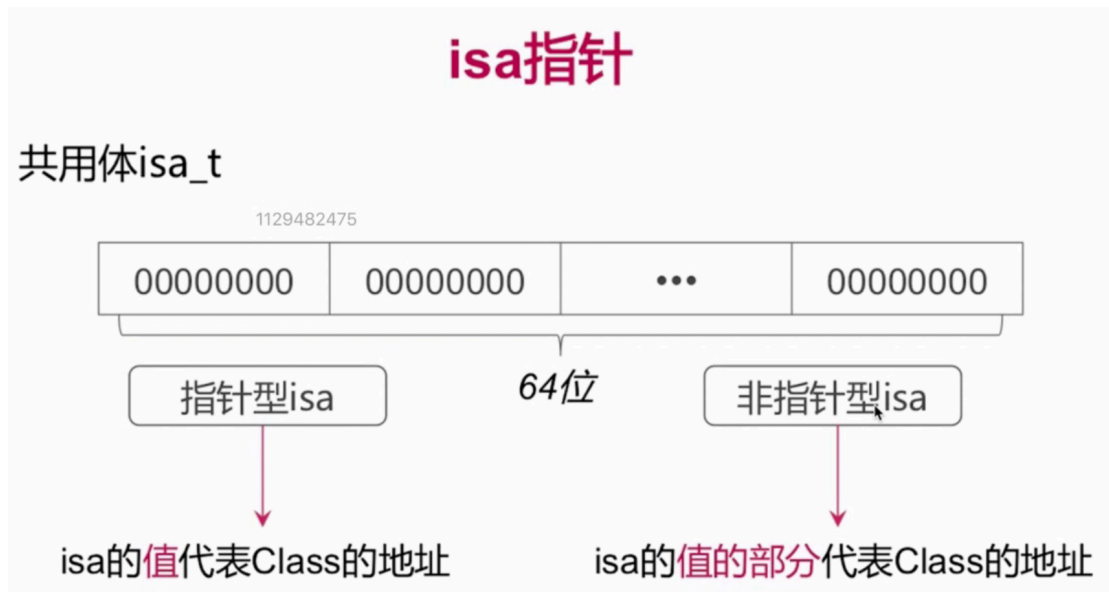
objc_class



Class 类是否是对象呢？

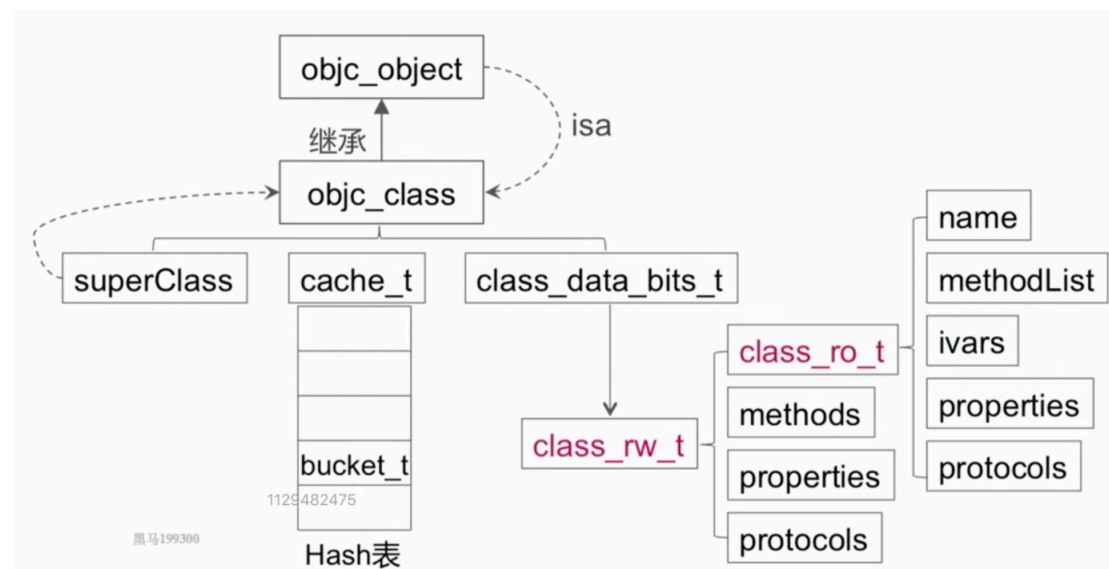
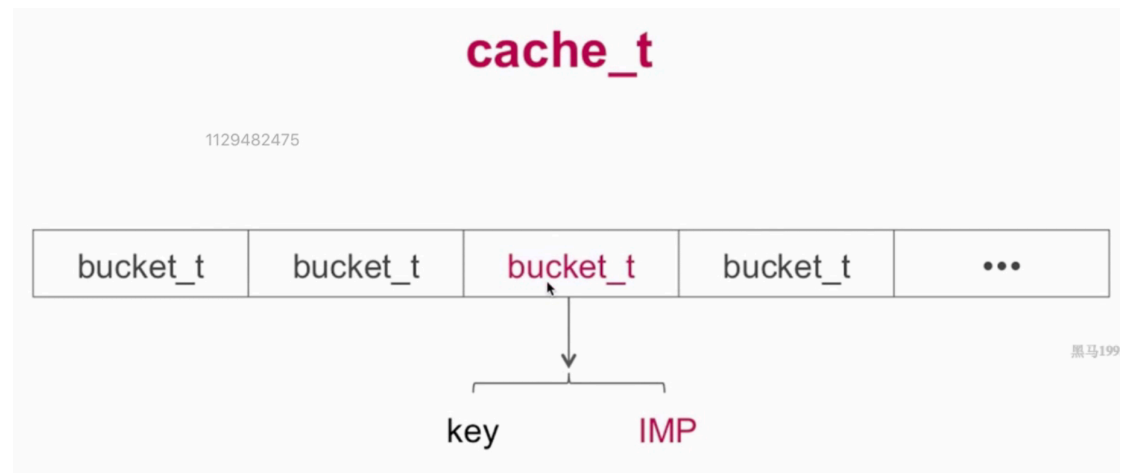
Class 也是一个对象，我们称他为类对象，因为它继承自 objc_object.

isa 指针

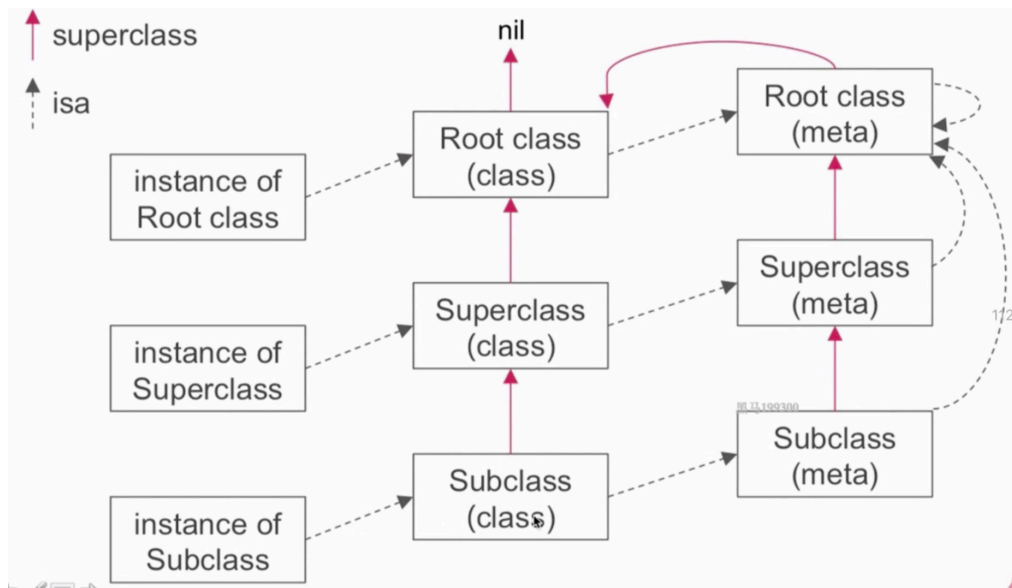


cache_t

- 用于快速查找方法执行函数。
- 是可增量扩展的哈希表结构。
- 是局部性原理的最佳应用。



对象、类对象、元类对象



类对象和元类对象分别是什么？类对象和元类对象之间有什么区别？

- 实例对象的 isa 指针指向的是类对象，类对象存储实例方法列表等信息；
- 类对象可以通过 isa 指针找到它的元类对象，元类对象存储类方法列表等信息。
- 类对象和元类对象都是 objc_class 数据结构的，这个数据结构由于继承了 objc_object，所以他们才有 isa 指针，进而才能实现我们上方所描述的。

元类对象和根元类对象他们的 isa 指针都是指向根元类的。根元类对象的 superClass 指向的是根类对象。

笔试题

```
#import "Mobile.h"
@interface Phone : Mobile

@end

@implementation Phone

- (id)init
{
    self = [super init];
    if (self) {
        NSLog(@"%@", NSStringFromClass([self class]));
        NSLog(@"%@", NSStringFromClass([super class]));
    }
    return self;
}

@end
```

黑马19930

?

打印出来的值两个都是 **phone**，没有 **mobile**。

消息传递

```
void objc_msgSend(id self, SEL cmd, ...)
```

```
void objc_msgSend(void /* id self, SEL op, ... */) {
```

[self class] ↔ objc_msgSend(self, @selector(class))

```
void objc_msgSendSuper(void /* struct objc_super *super, SEL op, ... */) {
```

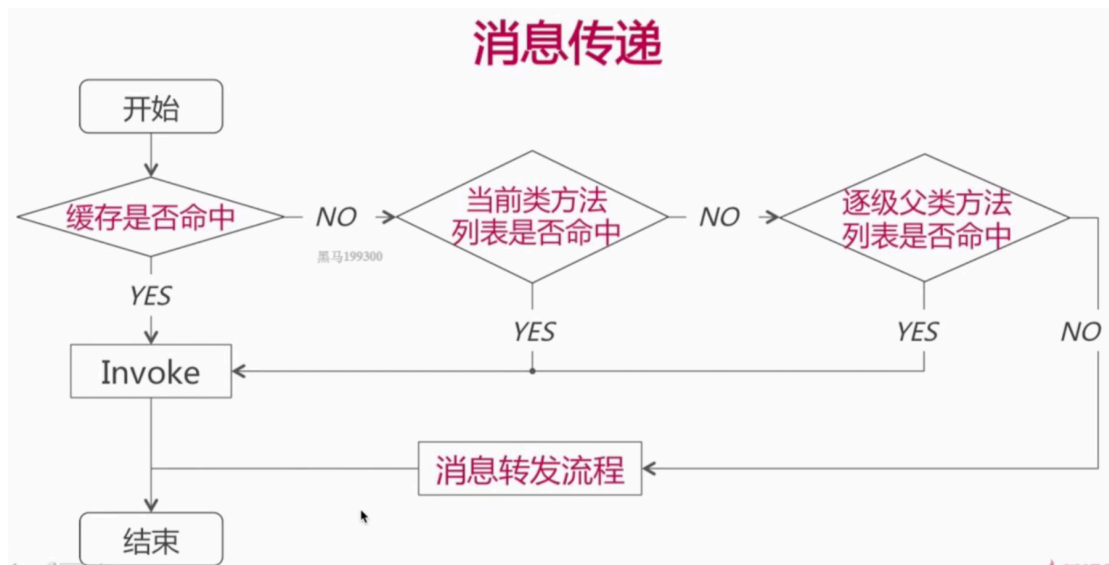
```
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained id receiver;
};
```

self

[super class] ↔ objc_msgSendSuper(super, @selector(class))

这个 **receiver** 最终传入的还是当前类的对象，就是即使用 **super** 调用方法，通过消息传递，最终传递进去的还是当前类的对象。

消息传递大致过程



1. 首先我们调用一个方法之后会查找缓存，看缓存里面是否有选择器对应的方法实现，如果有的话，我们通过函数指针调用函数，然后就完成了一次消息传递。
2. 如果缓存没有命中的话，那么我们会根据当前实例的 **isa 指针** 去查找当前类对象的方法列表，看看是否有同样名称的方法，如果找到的话，我们通过函数指针进行调用，结束消息传递。如果当前类方法列表中没有对应名称的方法，那么我们需要逐级父类方法中查找，(这里就相当于当前类的 **super class** 指针去查找他的父类的方法列表)。如果在父类方法列表没有找到，就一直往上指，指到 **nil** 为止。
3. 如果在父类方法当中一直查到根类，仍然没有查到同名的方法实现，那么这个时候就要进入到消息转发流程。然后结束消息传递的过程。

方法缓存查找相关（缓存查找具体是怎样的流程和步骤）

首先根据给定的方法选择器，通过一个函数映射出 **bucket_t** 在数组当中的位置，这一步实际就是**哈希查找**。然后就可以提取它对应的 **IMP** 函数指针返回给调用方就可以了。

例： 给定值是SEL，目标值是对应bucket_t中的IMP。



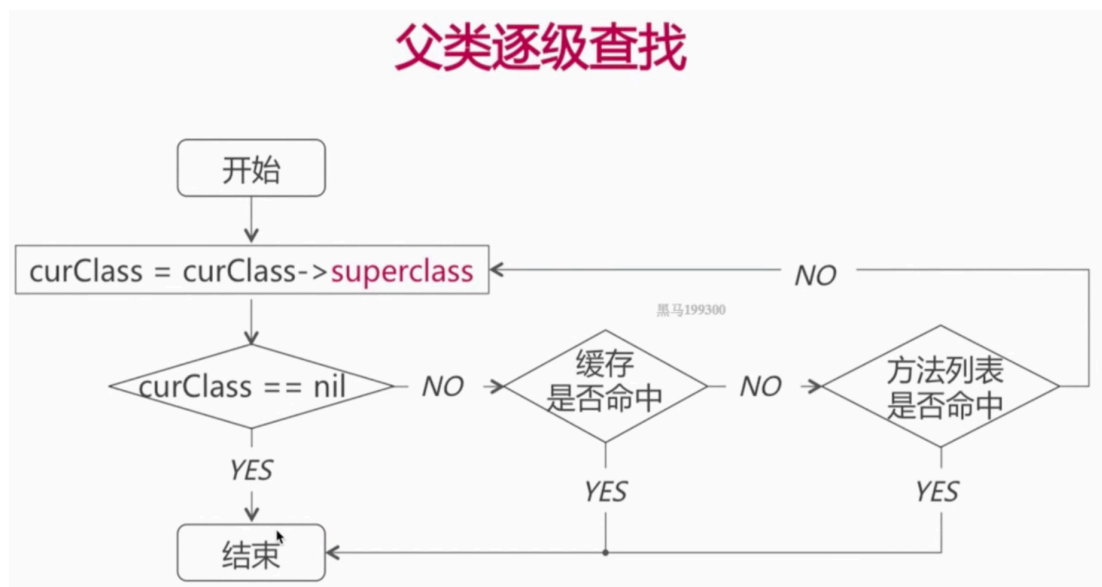
黑马199300

$$f(key) = key \& mask$$

当前类中查找

- 对于已排序好的方法列表，采用二分查找算法查找方法对应执行函数。
- 对于没有排序的列表，采用一般遍历查找方法对应执行函数。

父类逐级查找



消息传递的机制

1. 首先我们调用一个方法之后会查找缓存，看缓存里面是否有选择器对应的方

法实现，如果有的话，我们通过函数指针调用函数，然后就完成了一次消息传递。

2. 如果缓存没有命中的话，那么我们会根据当前实例的 isa 指针去查找当前类对象的方法列表，看看是否有同样名称的方法，如果找到的话，我们通过函数指针进行调用，结束消息传递。如果当前类方法列表中没有对应名称的方法，那么我们需要逐级父类方法中查找，(这里就相当于当前类的 super class 指针去查找他的父类的方法列表)。如果在父类方法列表没有找到，就一直往上指，直到 nil 为止。
3. 如果在父类方法当中一直查到根类，仍然没有查到同名的方法实现，那么这个时候就要进入到消息转发流程。然后结束消息传递的过程。
4. 对于缓存查找，实际上是哈希查找。
5. 当前类方法列表查找

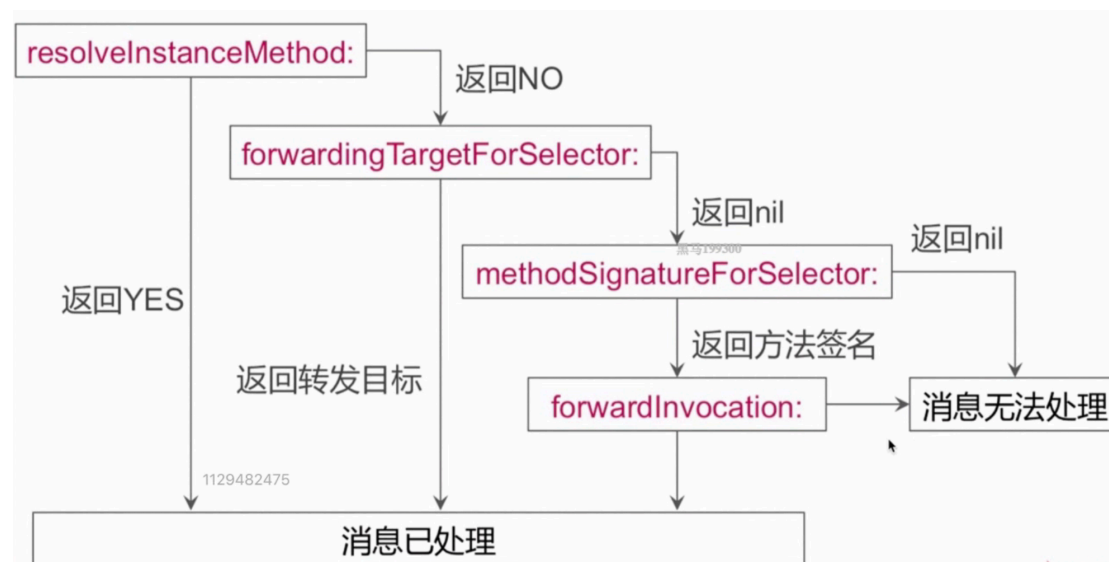
对于已排序好的方法列表，采用二分查找算法查找方法对应执行函数。

对于没有排序的列表，采用一般遍历查找方法对应执行函数

6. 对于逐级父类方法列表查找，是根据 superclass 指针，一点一点往上找，查找父类，一点一点遍历父类，每个父类中的方法查找也是同样两个步骤，首先查找父类的缓存，缓存没有命中，就再查找父类对应的方法列表。

消息转发流程

<https://www.jianshu.com/p/a0dc0364d717>



对于消息转发流程，首先是动态方法解析
如果是实例对象，此时会调用

```
+ (BOOL)resolveInstanceMethod:(SEL)selector
```

如果是类对象的转发流程，此时会调用：

```
+ (BOOL)resolveClassMethod:(SEL)selector
```

如果返回的是 YES，那么相当于当前消息已处理，那么结束了这次转发流程。
如果返回的是 No,那么会进入完整的信息转发流程，这时候会回调：

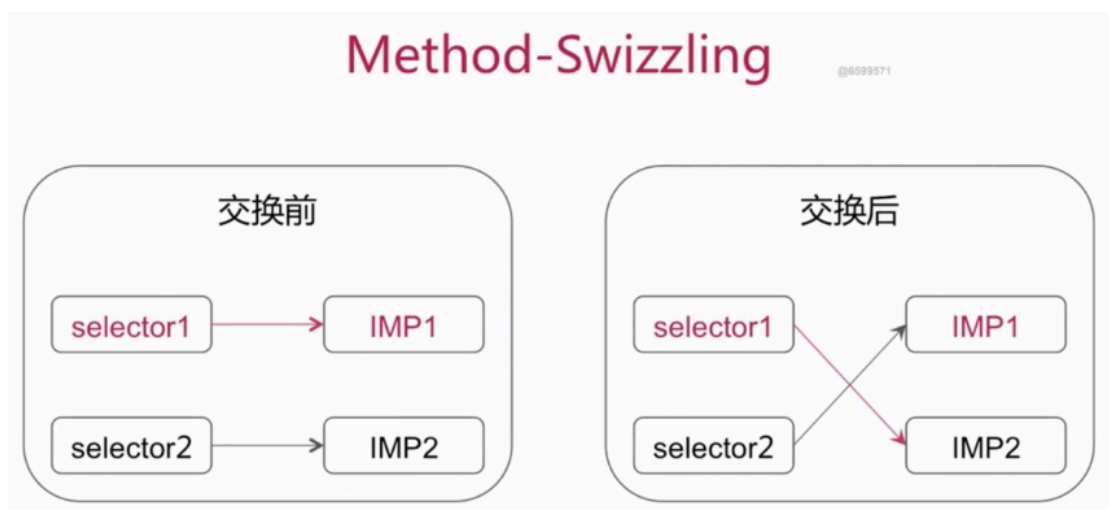
```
- (id)forwardingTargetForSelector:(SEL)selector
```

如果有返回转发目标，那么就结束了这次转发流程，如果返回的是 nil，那么系统还会调用 methodSignatureForSelector:方法，如果返回了方法签名，那么系统会调用

```
- (void)forwardInvocation:(NSInvocation *)invocation
```

如果该方法能够处理该转发流程的话，那么该转发流程就结束了，如果 methodSignatureForSelector 返回的是空，或者 forwardInvocation 没有办法处理这条消息的话，会被标记为消息无法处理，然后就出现了我们常见的 crash，“未识别选择器”。

Method-Swizzling



- Method originalMethod = class_getInstanceMethod(aClass, originalSel);
- Method swizzleMethod = class_getInstanceMethod(aClass, swizzleSel);
- method_exchangeImplementations(originalMethod, swizzleMethod);

方法交换，对一些页面的进出加一些统计信息，我们不可能页面每个方法都去加这种信息，使用方法交换给我们提供了很大的便利。直接父类里面对某个方法进行方法交换，然后新增一些东西，然后在调用原来的方法里面的内容。

```
+ (void)load{
    // 获取test方法
    Method test = class_getInstanceMethod(self, @selector(test));
    // 获取otherTest方法
    Method otherTest = class_getInstanceMethod(self,
        @selector(otherTest));
    // 交换两个方法的实现
    method_exchangeImplementations(test, otherTest);
}

- (void)test{
    NSLog(@"test");
}

- (void)otherTest
{
    // 实际上是调用test的具体实现
    [self otherTest];
    NSLog(@"otherTest");
}
```

黑马15

动态添加方法

你是否使用过 performSelector:这个方法？

// type: 方法类型: **void** 用 **v** 来表示, **id** 参数用**@**来表示, **SEL** 用:**:**来表示

class_addMethod(self, sel, (IMP) effect, "**v@:@**");

```

37 // [self addMethods];
38 // [self addAttributes];
39
40
41 }
42
43 #pragma mark -动态添加方法
44 #pragma mark -----
45 - (void)addMethods
46 {
47     SHPerson *person = [[SHPerson alloc] init];
48     //默认person, 没有实现eat方法, 可以通过performSelector调用, 但是会报错。
49     //动态添加方法就不会报错
50     [person performSelector:@selector(eat)];
51 }
52
53 - (void)addMethods2
54 {
55     SHPerson2 *person = [[SHPerson2 alloc] init];
56     [person performSelector:@selector(writeCode:) withObject:@100];
57 }
58 }
59

```

```

(lldb) po self
SHPerson2
(lldb) po sel
"writeCode:"
2018-01-01 12:03:51.595260+0800 dddd[2714:525189] effect ==> 100

```

```

// 定义 eat 方法 默认会前面两个隐式参数 id self, SEL _cmd 第三个才是我们自己传入的参数
void eat(id self, SEL _cmd, id object )
{
    NSLog(@"吃了%@", object);
} //使用到了未定义的对象方法
+(BOOL)resolveInstanceMethod:(SEL)sel
{
    NSLog(@"%@", NSStringFromSelector(sel));
    if (sel == @selector(eat:)) { //动态添加方法
        /**
        <#Description#>
        @param cls#> 那个类 description#>
        @param name#> 方法名 description#>
        @param imp#> 方法指针 description#>
        @param types#> 方法类型 description#>
        @return <#return value description#>
        */
        // class_addMethod(<#Class _Nullable __unsafe_unretained cls#>, <#SEL
        _Nonnull name#>, <#IMP _Nonnull imp#>, <#const char * _Nullable types#>)
        class_addMethod([self class], @selector(eat:), (IMP)eat, "v");
    }
    return [super resolveInstanceMethod:sel];
}
@end

```

动态方法解析

面试可能会问你是否使用过@dynamic 这个编译器关键字

@dynamic

- 动态运行时语言将函数决议推迟到运行时。（当我们把一个属性表示为@dynamic，代表着编译器在编译的时候不需要为我们生成该属性的 set 方法和 get 方法的具体实现，而是在运行时具体的我们调用了 set 方法或者 get 方法，再去添加具体的实现）。
- 编译时语言在编译期进行函数决议。

总结

[obj foo]和 objc_msgSend()函数之间有什么关系？

（这个可以回答方法的调用其实就是消息的转发，[obj foo]会转化成objc_msgSend(self,"foo"),然后开始了 runtime 消息的传递）。

runtime 如何通过 Selector 找到对应的 IMP 地址的？

这个其实是考你 runtime 消息传递的实质的。

能否向编译后的类中增加实例变量？

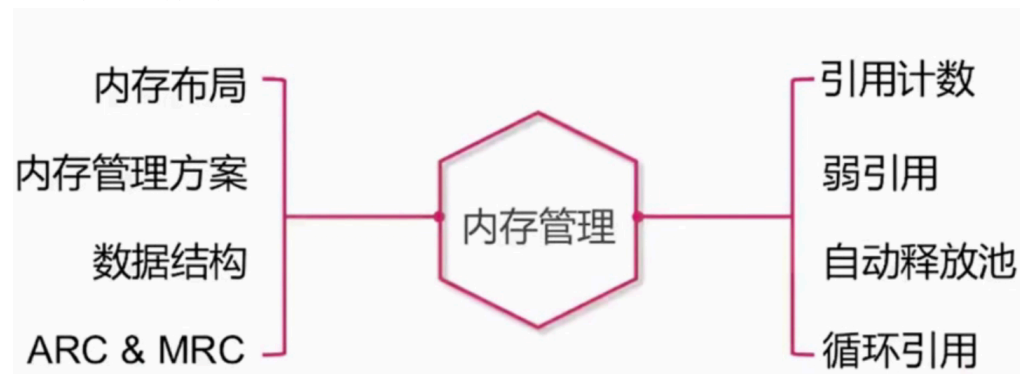
一般类在编译前就完成了实例变量的布局，在编译后是没有办法进行修改的，所以说对于编译后的类是不能增加实例变量的。

能否向动态添加的类中增加实例变量？

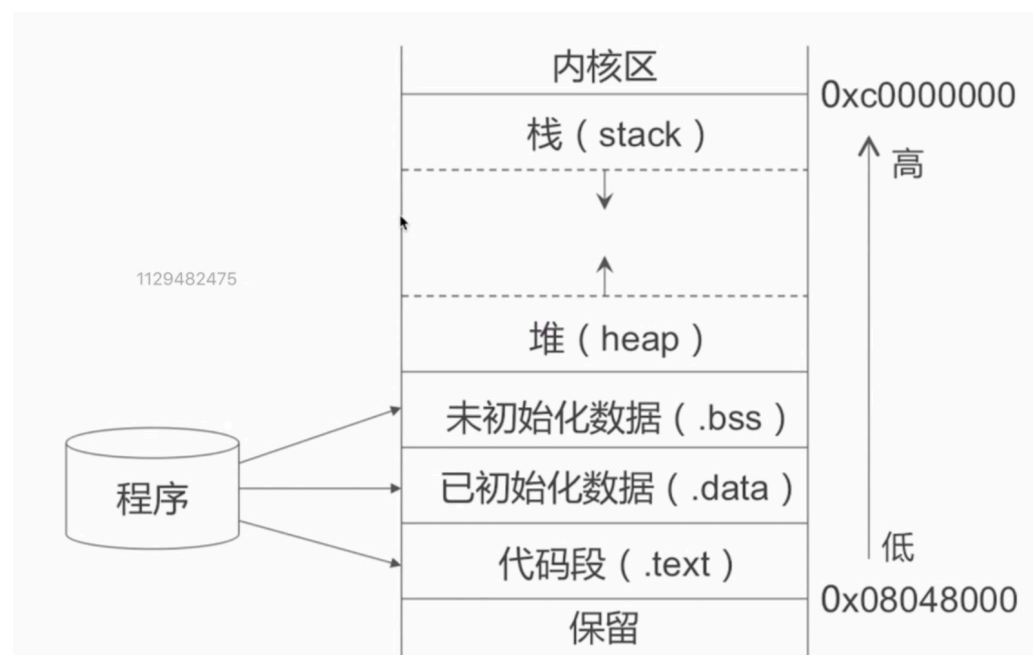
可以的，因为我们再动态添加类的过程中，只要在它的注册类对方法之前去完成实例变量的添加就是可以实现的。

第六章、内存管理

是的发送到发顺丰的



6-1 内存布局相关



- **stack**: 方法调用。
- **heap**: 通过 **alloc** 等分配的对象。
- **bss**: 未初始化的全局变量。
- **data**: 已初始化的全局变量
- **text**: 程序代码

6-2 内存管理方案相关

iOS 操作系统是怎样对内存进行管理的？

iOS 操作系统中，不同场景会采用不同的内存管理方案。不同场景的内存管理方案说明如下：

- 1)、TaggedPointer: 小对象采用这种内存管理方案，比如 NSNumber 等
- 2)、NONPOINTER_ISA: 64 位架构采用这种内存管理方案。在 64 位机中数据一般用 40 位就够了，苹果为了提高内存利用率在剩余的比特位上存储了内存管理相关的内容。
- 3)、散列表：其中包括弱引用表和引用计数表

iOS 通过引用计数器来管理内存，对象产生时引用计数为 1；当一个新的引用指向对象，那么这个对象的引用计数值增加 1；当对象被少一次引用时对象的引用记数值减 1；当对象的引用记数值为 0 的时候，代表这个对象没有被使用，此时系统会自动回收掉此对象，回收这个对象的同时自动调用这个对象的 dealloc 方法。

MRC

手动引用计数

MRC 是手动引用计数，包含的关键字有 alloc、retain、release、retainCount、autorelease、dealloc。在 MRC 中重写 dealloc 方法需要手动去调用父类的 dealloc 方法。

在 ARC 中，retain、release、retainCount、autorelease 这四个关键字不能出现，否则会引起编译报错。

ARC

自动引用计数

什么是 ARC？

- ARC 是自动引用计数，ARC 是 LLVM（编译器）和 Runtime 协作的结果，如 weak 关键字实现的内存管理就是由 Runtime 实现的。

ARC 其他特点：

- ARC 中禁止手动调用 retain、release、retainCount、dealloc，且重写 dealloc 方法时不能手动调用父类的 dealloc 方法。
- ARC 中新增 weak、strong 属性关键字。

ARC 和 MRC 有什么区别呢？

- MRC 是手动管理内存，ARC 是通过 **LLVM（编译器）** 和 **Runtime 协作** 来实现自动引用计数管理。
- MRC 可以调用的一些引用计数的方法，ARC 不能调用，如 ARC 中禁止手动调用 `retain`、`release`、`retainCount`、`dealloc`，且重写 `dealloc` 方法时不能手动调用父类的 `dealloc` 方法。

Retain 操作的时候系统格式怎样查找他的引用计数的？

Retain 是经过两次哈希查找来查到他的引用计数值，然后进行相应的+1 操作。

alloc 实现

经过一系列调用，最终调用了 c 函数 `calloc`。此时并没有设置引用计数为 1，但是 `retainCount` 确实为 1，为什么？对象初始化时引用计数表中的值为 0，`retainCount` 值为 1 是因为取值时从一取值。

<https://blog.csdn.net/BlackWolfSky/article/details/84641357>

AutoreleasePool 的实现原理是怎样的？

- 自动释放池是以**栈为**结点通过**双向链表**的形式组合而成的数据结构。

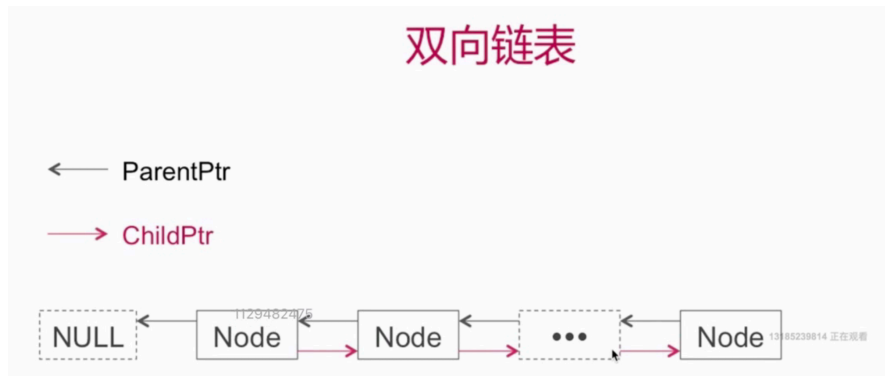
AutoreasePool 为何可以嵌套使用？

多层嵌套就是多次插入哨兵对象。

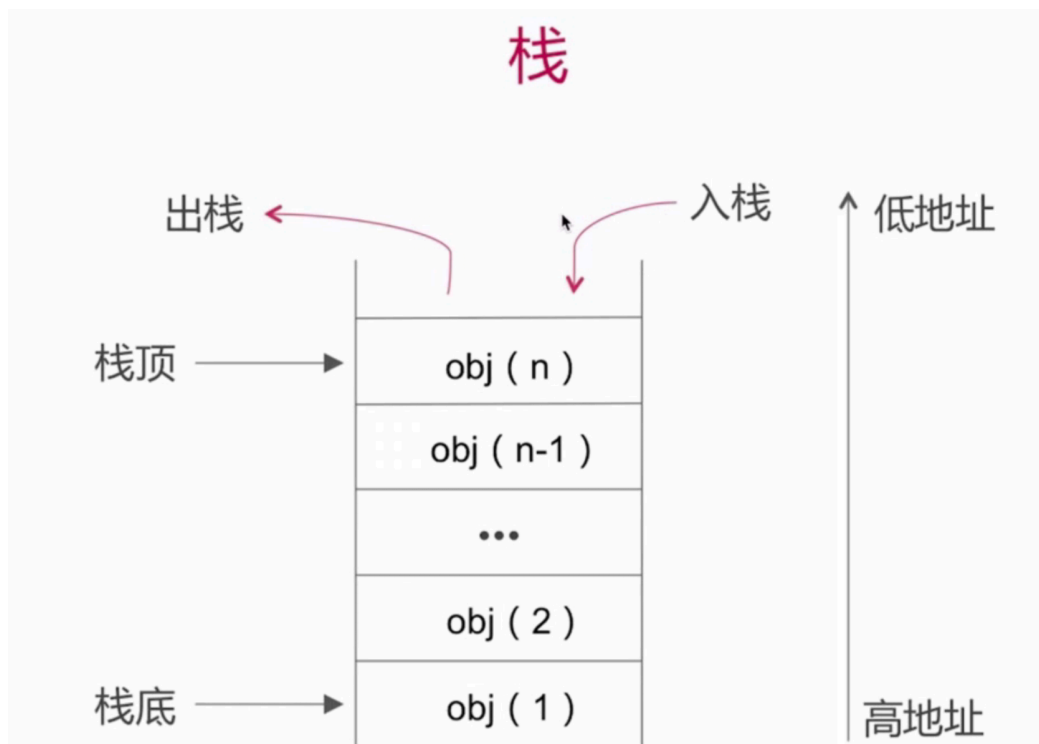
什么是自动释放池，自动释放池的结构是什么？

- 自动释放池是以**栈为**结点通过**双向链表**的形式组合而成；
- 是和**线程**——对应的。

双向链表



栈



循环引用

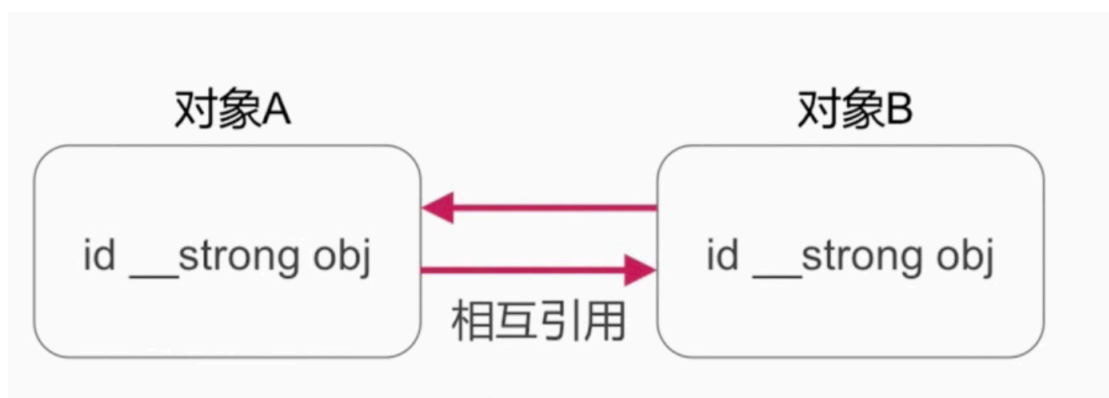
三种循环引用

- 自循环引用
- 相互循环引用
- 多循环引用

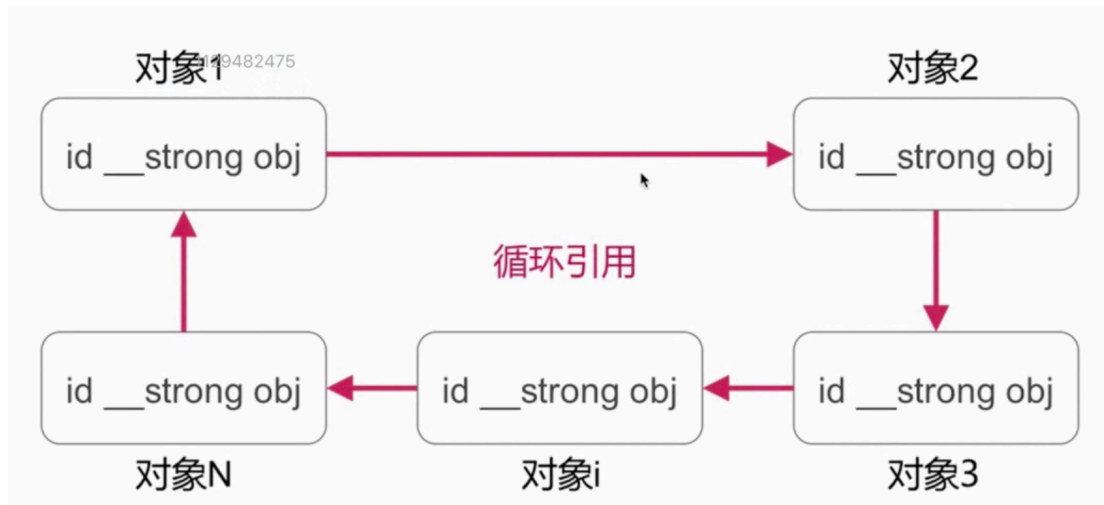
自循环引用



相互循环引用



多循环引用



如何破除循环引用？

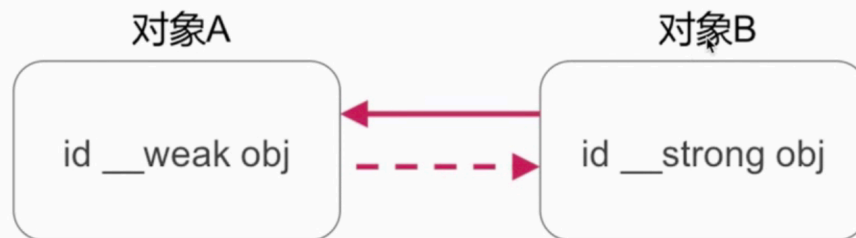
- 避免产生循环引用。
- 在合适的时机手动断环。

具体的解决方案有哪些？

- `__weak`
- `__block` （一般用在 `block` 产生的循环引用）
- `__unsafe_unretained`

__weak 破解（避免产生循环引用）

避免产生循环引用的常用方案



__block 破解

*注意

黑马199300

- **MRC**下，__block修饰对象不会增加其引用计数，**避免**了循环引用。
- **ARC**下，__block修饰对象会被强引用，**无法避免**循环引用，**需手动解环**。

__unsafe_unretained 破解

- 修饰对象不会增加其引用计数，**避免**了循环引用。
- 如果被修饰对象在某一时机被释放，会产生**悬垂指针**！由于悬垂指针的原因会导致内存泄露，所以一般不建议使用__unsafe_unretained。

循环引用示例

你在开发工程中是否遇到过循环引用，你又是怎样解决循环引用问题的？

- Block 的使用示例（请参看 block 章节的讲解）
- NSTimer 的循环引用问题。

NSTimer 的循环引用问题

<https://www.jianshu.com/p/d4589134358a>

总结

- 什么是 ARC?
ARC 是编译器和 runtime 共同协作
- 为什么 weak 指针指向的对象在废弃之后会被自动置为 nil?
当对象被废弃之后, dealloc 内部方法实现的时候, 会调用清除弱引用的一个方法, 然后清除弱引用的方法当中会通过哈希算法来查找被废弃对象在弱引用表当中的位置, 来提取弱引用指针的列表数组, 然后进行 for 循环遍历, 把每一个 weak 指针都置为 nil.
- 苹果是如何实现 autoreleasePool 的?
autoreleasePool 是以栈为节点由双向链表形式来合成的这么样的数据结构。
- 什么是循环引用? 你遇到过哪些循环引用, 是怎么样解决的?

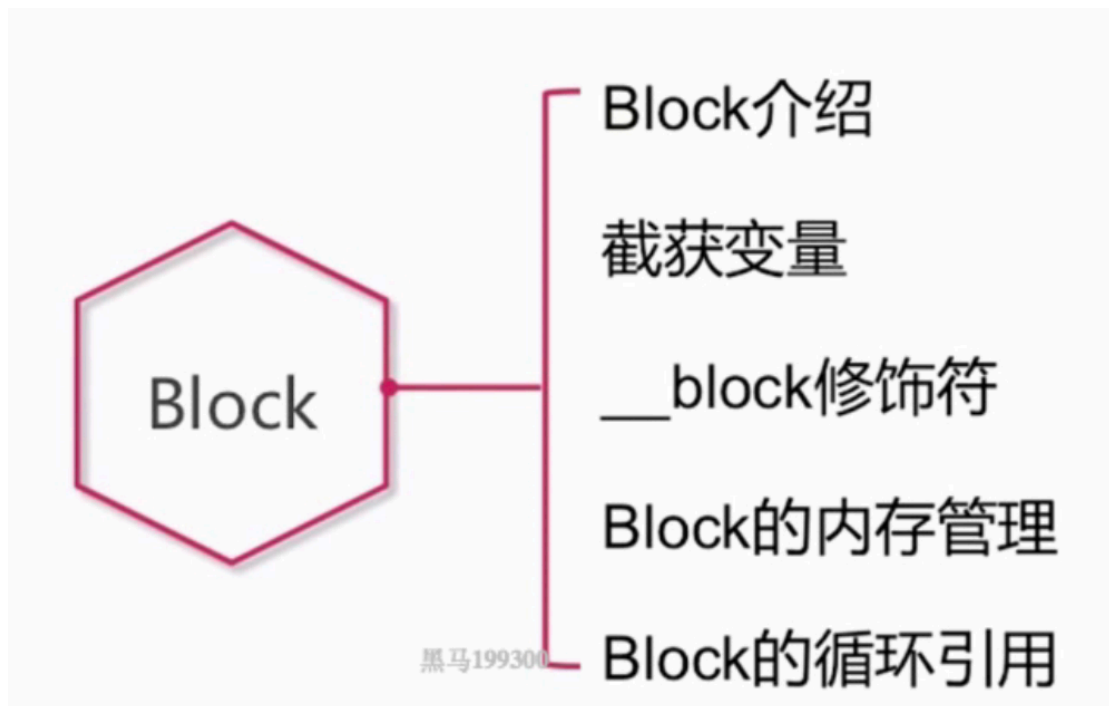
循环引用就是相互强引用，无法释放。

使用 Block 的时候遇到过，使用__weakSelf 代替 self

```
(__weak __typeof(self)weakSelf = self)
```

[illegible]

第七章. Block



7-1 Block 本质相关

什么是 block?

Block 是将函数及其执行上下文封装起来的对象。

什么是 Block 调用?

看过编译之的源码后，我们知道 Block 的调用就是函数的调用。

7-2 Block 截获变量相关

(如下笔试题)

答案是 12;

```
14 int global_var = 4;
15 // 静态全局变量
16 static int static_global_var = 5;
17
18 - (void)method
19 {
20     int multiplier = 6;
21     int(^Block)(int) = ^int(int num)
22     {
23         return num * multiplier;
24     };
25     multiplier = 4;
26     NSLog(@"result is %d", Block(2));
27 }
28
```

2018-03-11 20:49:27.959873+0800 Block[6309:734231] result is 12

```
5 // 静态全局变量
6 static int static_global_var = 5;
7
8 - (void)method
9 {
10     static int multiplier = 6;
11     int(^Block)(int) = ^int(int num)
12     {
13         return num * multiplier;
14     };
15     multiplier = 4;
16     NSLog(@"result is %d", Block(2));
17 }
18
```

03-11 20:50:56.771435+0800 Block[6325:735601] result is 8

局部变量(基本数据类型、对象类型)、静态局部变量、全局变量、静态全局变量。

关于 **block** 的截获特性你是否有了了解，**block** 的截获变量特性又是怎样的呢？

- 对于**基本数据**类型的**局部变量**截获其值。
- 对于**对象**类型的局部变量**连同所有权修饰符**一起截获。
- 以**指针形式**截获局部静态变量。
- **不截获**全局变量、静态全局变量。

7-3 __block 修饰符相关

什么场景下使用 **__block** 修饰符？

- **一般情况下**，需要对被截获变量进行**赋值**操作时，要使用 **__block** 修饰符。

Block 的内存管理。需要注意的是，赋值并不等于使用，如下第一个例子，只是使用，不需要 **__block** 修饰符：

```

{
    NSMutableArray *array = [NSMutableArray array];
    void(^Block)(void) = ^{
        [array addObject:@123];
    };

    Block();
}

```

<https://blog.csdn.net/BlackWolfSky>

下面代码是对截获变量的赋值，需要使用__block修饰符：

```

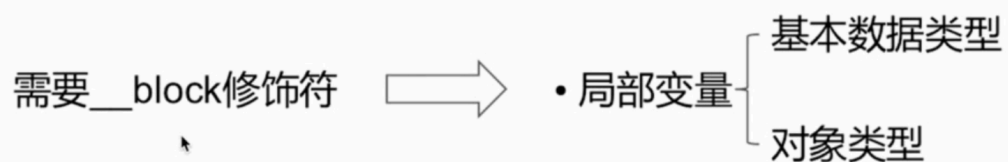
{
    NSMutableArray *array = nil;
    void(^Block)(void) = ^{
        array = [NSMutableArray array];
    };

    Block();
}

```

<https://blog.csdn.net/BlackWolfSky>

对变量进行赋值时



对变量进行赋值时



__block相关的笔试题

```
{
    __block int multiplier = 6;
    int(^Block)(int) = ^int(int num){
        return num * multiplier;
    };
    multiplier = 4;
    NSLog(@"result is %d", Block(2)); ➡ "result is 8"
}
```

在 Block 外部__block 修饰的变量编译后变成了对象，这等于是在原变量外部包了一层，Block 截获时截获的是新对象的指针，通过新对象的指针就可以为原对象赋值了。

7-4 Block 内存管理相关

Block 的 Copy 操作

Block的Copy操作

Block类别	源	Copy结果
__NSConcreteStackBlock	栈	堆
__NSConcreteGlobalBlock	数据区	什么也不做
__NSConcreteMallocBlock	堆	增加引用计数

当我们在对栈上的 **block** 进行 **copy** 操作之后，假如在 **MRC** 情况下，是否会引起内存泄露呢？

是的

Block 的引用循环

Block的引用循环

```
82475 {
    _array = [NSMutableArray arrayWithObject:@"block"];
    _strBlk = ^NSString*(NSString* num){
        return [NSString stringWithFormat:@"helloc_%@", _array[0]];
    };
    _strBlk(@"hello");
}
```

对象

持有self

黑马199300

解决方案：

答案

```
{
    _array = [NSMutableArray arrayWithObject:@"block"];
    __weak NSArray* weakArray = _array;
    _strBlk = ^NSString*(NSString* num){
        return [NSString stringWithFormat:@"helloc_%@", weakArray[0]];
    };
    _strBlk(@"hello");
}
```

对象

弱引用

黑马199300

```
__weak typeof(self) weakSelf = self
```

Block的引用循环

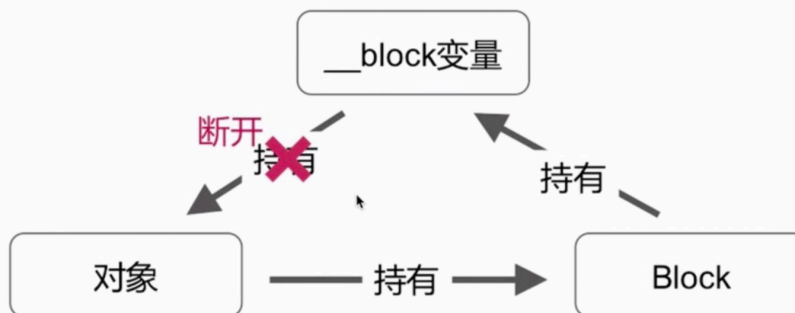
1129482475

```
{  
    __block MCBLOCK* blockSelf = self;  
    _blk = ^int(int num){  
        // var = 2  
        return num * blockSelf.var ;  
    };  
  
    _blk(3);  
}
```

黑马1

请思考，这段代码有问题吗？

ARC下的引用循环



ARC下的解决方案

```
{  
    __block MCBLOCK* blockSelf = self;  
    _blk = ^int(int num){  
        // var = 2  
        int result = num * blockSelf.var;  
        blockSelf = nil;  
        return result;  
    };  
  
    _blk(3);  
}
```

1129482475

MRC下同样没有问题

总结

什么是 Block?

Block 是将函数及其执行上下文封装起来的对象。

为什么 Block 会产生循环引用?

如果当前 Block 对当前对象的某一成员变量进行截获的话，那么这个 Block 对应变量有一个强引用，而当前 block 由于当前对象对其有一个强引用，就产生了一个自循环引用的循环引用。我们可以申明其为 `_weak` 变量来进行循环引用的消除，如果我们定义了一个 `_block` 说明符的话，那么也会产生循环引用，但是是区分场景的，例如在 ARC 下是会产生循环引用的，而在 MRC 下不会。同时，在 ARC 下我们可以通过断环的形式去减除对应的循环引用，但是有一个弊端，如果这个 Block 一直得不到调用的话，是不会减除的。

怎样理解 Block 截获变量的特性?

- 对于基本数据类型的局部变量截获其值。
- 对于对象类型的局部变量连同所有权修饰符一起截获。
- 以指针形式截获局部静态变量。
- 不截获全局变量、静态全局变量。

你都遇到过哪些循环引用？你又是怎样解决的？

（上面的第二题可以解答）

第八章、多线程



<https://blog.csdn.net/BlackWolfSky/article/details/84658303>

<https://www.jianshu.com/p/96b93aa05bcd>

在 iOS 系统当中，为我们提供了哪几种多线程的技术方案呢？

GCD NSOperation NSThread

同步异步并行串行主要影响区别

- 同步和异步主要影响：能不能开启新的线程

同步：在当前线程中执行任务，不具备开启新线程的能力

异步：在新的线程中执行任务，具备开启新线程的能力

- 并行和串行主要影响：任务的执行方式

并行：多个任务并发（同时）执行

串行：一个任务执行完毕后，再执行下一个任务

GCD

同步/异步 和 串行/并发

- `dispatch_sync(serial_queue, ^{...});` 同步调用串行队列
- `dispatch_async(serial_queue, ^{...});` 异步调用串行队列
- `dispatch_sync(concurrent_queue, ^{...});` 同步调用并行行队列
- `dispatch_async(concurrent_queue, ^{...});` 异步调用并行行队列

`Dispatch_sync(dispatch_get_main-queue())`

`Dispatch_async(dispatch_get _main-queue())`

```
- (void)viewDidLoad {  
    dispatch_sync(dispatch_get_main_queue(), ^{  
        [self doSomething];  
    });  
}
```



<https://www.cnblogs.com/BlackWolfSky> **死锁!**

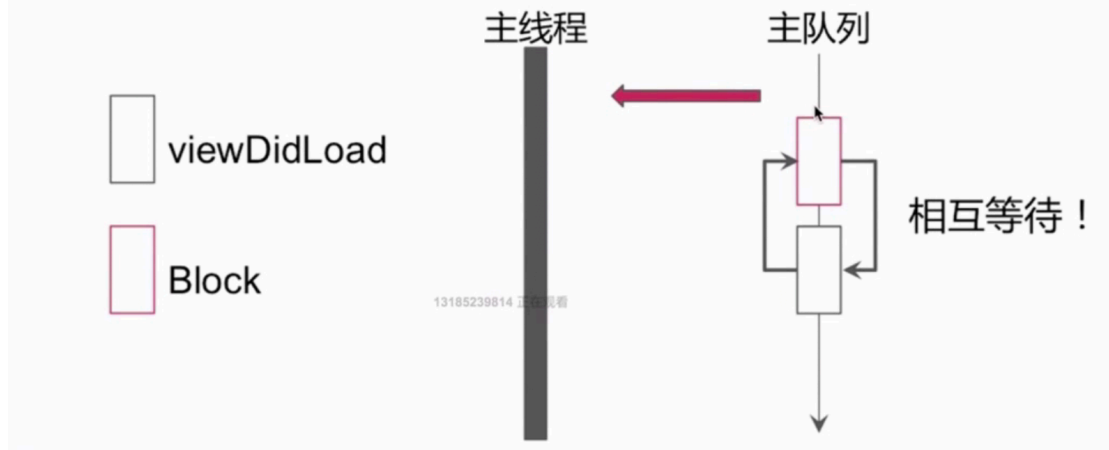
- 这个是同步分配到主线程的任务，如果异步主线程，效果是一样的，也是只在主线程执行，会造成死锁。

回答出死锁的 6 分，还要回答出为什么死锁才能满分。死锁原因如下：

队列引起的循环等待

死锁原因

队列引起的循环等待



- 同步调用串行

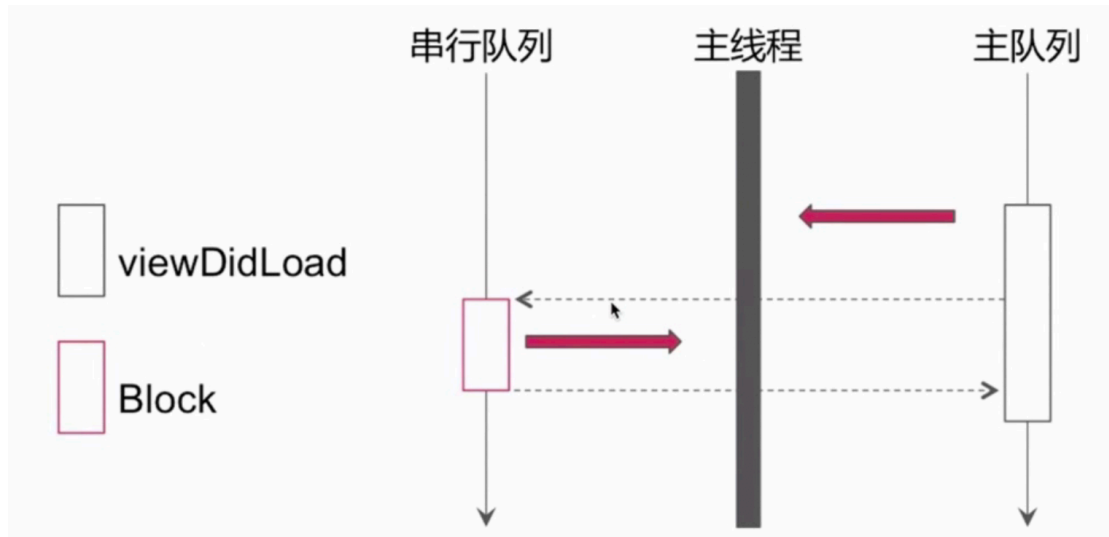
同步串行

```
- (void)viewDidLoad {  
    dispatch_sync(serialQueue, ^{  
        [self doSomething];  
    });  
}
```



没问题！

分析如下：这是自定义的一个串行的队列，不是主队列



- 同步并发

```

- (void)viewDidLoad {
    NSLog(@" 1" );
    dispatch_sync(global_queue, ^{
        NSLog(@" 2" );
        dispatch_sync(global_queue, ^{
            NSLog(@" 3" );
        });
        NSLog(@" 4" );
    });
    NSLog(@" 5" );
}

```

黑马199300

答案：12345

解析：由于是同步队列，那么肯定都是在当前线程执行，已同步形式提交到并发队列，那么都是在主线程执行。（用于理解：由于是并发队列，所以在第一个 block 没有完全完成的时候，可以执行包含在它里面的那个 block。）

- 如果 `global_queue` 换成串行队列，那么就会发生死锁。因为第一个 `block` 没有完全执行完，里面包含的那个也加入到了串行队列，那么就会造成相互等待。

- 异步并发

```
- (void)viewDidLoad {  
    dispatch_async(global_queue, ^{  
        NSLog(@" 1" );  
        [self performSelector: @selector(printLog)  
            withObject: nil  
            afterDelay: 0 ];  
        NSLog(@" 3" );  
    }];  
}  
  
- (void)printLog { NSLog(@" 2" ); }
```

 答案 : 13

解析：日志 2 不会被打印出来的，异步方式分配到全局并发队列，会在底层某个线程进行处理，这个现在默认情况下是不会开启 `runloop` 的，即使 `afterDelay` 是 0 秒，但是还是会在 0 秒后提交到 `runloop` 的，没有 `runloop`，那么就不会被执行。

线程栅栏 `dispatch_barrier`

线程栅栏可以阻塞某个 `queue` 中任务的执行直到 `queue` 中栅栏之前的任务被执行完毕。

线程栅栏 `dispatch_barrier`

线程栅栏可以阻塞某个 `queue` 中任务的执行直到 `queue` 中 栅栏 之前的任务被执行完毕。

```
- (void)dispatch_barrier_test {
    dispatch_queue_t queue = dispatch_queue_create("test_queue", DISPATCH_QUEUE_CONCURRENT);

    for (int i = 1; i <= 3; i++) {
        dispatch_async(queue, ^{
            sleep(3);
            NSLog(@"任务%d结束", i);
        });
    }
    NSLog(@"代码经过栅栏");
    dispatch_barrier_sync(queue, ^{
        sleep(5);
        NSLog(@"栅栏结束");
    });
    NSLog(@"代码通过栅栏");

    for (int i = 4; i <= 6; i++) {
        dispatch_async(queue, ^{
            sleep(3);
            NSLog(@"任务%d结束", i);
        });
    }
    NSLog(@"代码结束");
}
```

打印结果:

```
2018-03-20 16:44:07.347195+0800 test[4987:607391] 代码经过栅栏
2018-03-20 16:44:10.351954+0800 test[4987:607438] 任务1结束
2018-03-20 16:44:10.351954+0800 test[4987:607439] 任务3结束
2018-03-20 16:44:10.351984+0800 test[4987:607440] 任务2结束
2018-03-20 16:44:15.353557+0800 test[4987:607391] 栅栏结束
2018-03-20 16:44:15.353836+0800 test[4987:607391] 代码通过栅栏
2018-03-20 16:44:15.354086+0800 test[4987:607391] 代码结束
2018-03-20 16:44:18.357376+0800 test[4987:607441] 任务5结束
2018-03-20 16:44:18.357369+0800 test[4987:607440] 任务4结束
2018-03-20 16:44:18.357457+0800 test[4987:607438] 任务6结束
```

- 同步栅栏会阻塞之后普通代码的执行、异步栅栏则不会。
应用线程栅栏的特性、我们可以更好的做一些线程同步。某些情况下不需要写好几层任务组来同步任务。

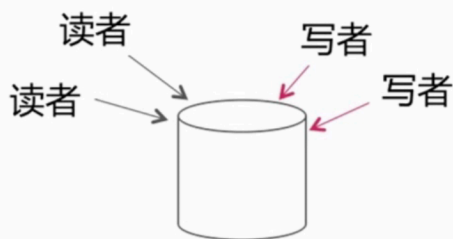
- 同步栅栏 `dispatch_barrier_sync` 异步栅栏 `dispatch_barrier_async`

- 例如 4/5/6 任务想要等待 1/2/3 任务。用任务组的话、需要一个任务组包含 (1/2/3)。然后在任务组完成的回调中再并发出三个任务(4/5/6)。而且还无法控制线程的阻塞、除非在想要阻塞的地方加入最后一个同步任务。想想就很麻烦....

怎样实现多读单写？

dispatch_barrier_async()

怎样利用GCD实现多读单写？



- 读者、读者并发
- 读者、写者互斥
- 写者、写者互斥

`dispatch_barrier_async`

1)、怎样通过GCD实现多读单写？

`dispatch_barrier_async(concurrent_queue, ^{//写操作})`

```

38 - (id)objectForKey:(NSString *)key
39 {
40     __block id obj;
41     // 同步读取指定数据
42     dispatch_sync(concurrent_queue, ^{
43         obj = [userCenterDic objectForKey:key];
44     });
45
46     return obj;
47 }
48
49 - (void)setObject:(id)obj forKey:(NSString *)key
50 {
51     // 异步栅栏调用设置数据
52     dispatch_barrier_async(concurrent_queue, ^{
53         [userCenterDic setObject:obj forKey:key];
54     });
55 }

```

<https://blog.csdn.net/BlackWolfSky>

读操作为什么不用异步呢？因为读操作需要立即返回结果。而并发队列保证了多个线程可以并发执行。

队列组 `dispatch_group`

队列组 `dispatch_group`

```
//队列组
- (void)dispatch_group_test {

    dispatch_queue_t queue = dispatch_queue_create("queue_test", DISPATCH_QUEUE_CONCURRENT);
    dispatch_group_t group = dispatch_group_create();

    dispatch_group_async(group, queue, ^{
        NSLog(@"任务1—准备休眠3秒");
        sleep(3);
        NSLog(@"任务1—完成");
    });

    NSLog(@"主线程—准备休眠5秒");
    sleep(5);
    NSLog(@"主线程休眠结束");

    dispatch_group_async(group, queue, ^{
        NSLog(@"任务2—准备休眠10秒");
        sleep(10);
        NSLog(@"任务2—完成");
    });

    dispatch_group_notify(group, queue, ^{
        NSLog(@"任务组完成");
    });

    NSLog(@"主线程结束");
}
```

打印结果:

```
2018-03-20 10:43:32.135642+0800 test[1763:94602] 主线程—准备休眠5秒
2018-03-20 10:43:32.135649+0800 test[1763:94645] 任务1—准备休眠3秒
2018-03-20 10:43:35.141269+0800 test[1763:94645] 任务1—完成
2018-03-20 10:43:37.137225+0800 test[1763:94602] 主线程休眠结束
2018-03-20 10:43:37.137513+0800 test[1763:94602] 主线程结束
2018-03-20 10:43:37.137515+0800 test[1763:94645] 任务2—准备休眠10秒
2018-03-20 10:43:47.139737+0800 test[1763:94645] 任务2—完成
2018-03-20 10:43:47.140116+0800 test[1763:94645] 任务组完成
```

这里可以扩展一下任务组的机制

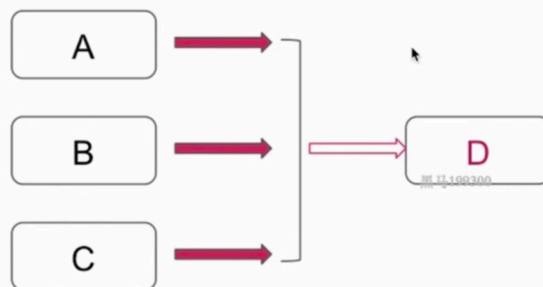
- `dispatch_group_notify` 任务组的结束通知可以添加多次、并且会多次调用。

- 类似于 **NSOperation** 中的依赖、GCD 的 **group** 监听是可以追加的。只要任务组中的任务没有全部完成、**group** 完成的监听就不会被调用、哪怕是后追加的任务。

使用 **GCD** 实现需求：A、B、C 三个任务并发，完成后执行任务 D

dispatch_group_async()

使用 GCD 实现这个需求：A、B、C 三个任务并发，完成后执行任务 D？



```
一 使用队列组

- (void)dispatch_group_test {
    dispatch_queue_t queue = dispatch_queue_create("queue_test", DISPATCH_QUEUE_CONCURRENT);
    dispatch_group_t group = dispatch_group_create();

    dispatch_group_async(group, queue, ^{
        NSLog(@"任务");
        执行A、B、C
        NSLog(@"任务完成");
    });

    NSLog(@"添加完成监听");
    dispatch_group_notify(group, queue, ^{
        //当添加到组中的所有任务执行完成之后会调用该Block
        NSLog(@"任务组完成");
        执行D
    });
}
```

二 还可以使用异步栅栏的方式, (线程栅栏可以阻塞某个queue中任务的执行直到queue中栅栏之前的任务被执行完毕)

NSOperation 相关

主要优点（几种任务状态能说上来）

- a、可以添加任务依赖
- b、可以控制任务执行状态：**isReady**、**isExecuting**、**isFinished**、**isCancelled**。
- c、可以控制最大并发量

状态控制（会问到怎样控制状态，一下是回答）

- 如果只重写了 main 方法，则系统会控制任务执行状态的变化以及任务的退出。
- 如果重写了 start 方法，则需要自行控制任务状态。

系统怎样移除一个 isFinished=YES 的 NSOperation 的？

任务完成后通过 KVO 的检测 finished 状态为 YES，系统通过 NSOperation Queue 合适的时机移除该 NSOperation 对象。

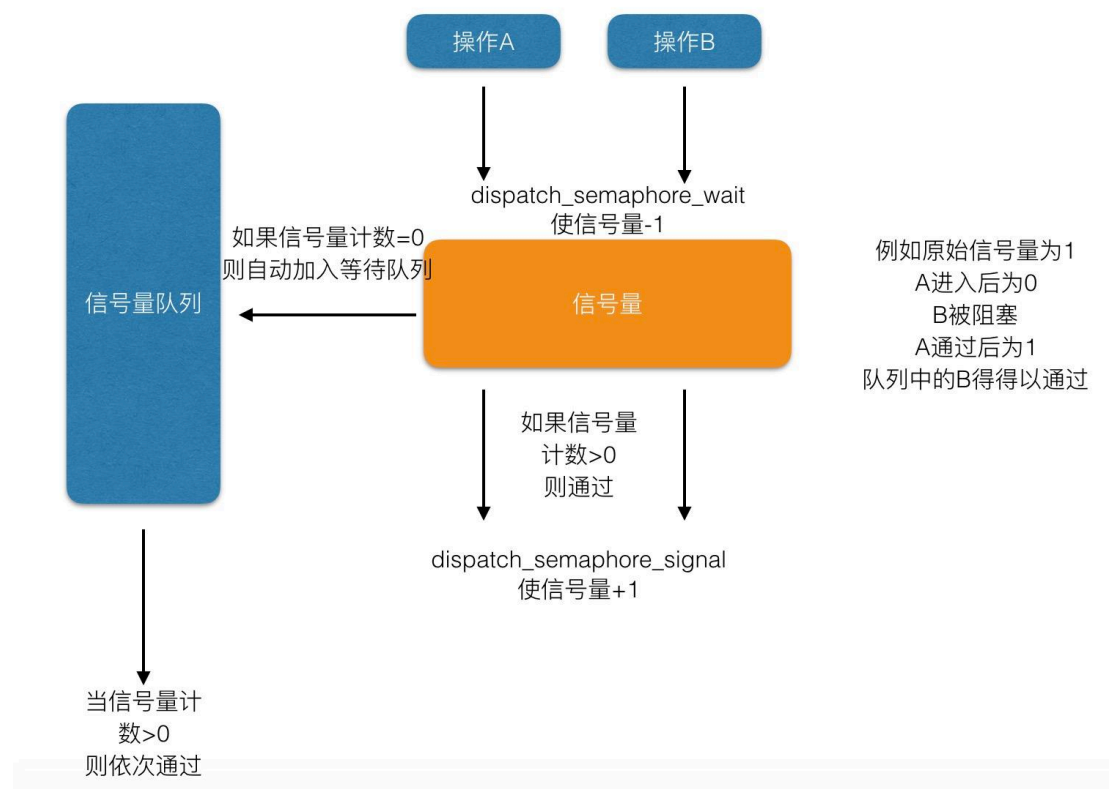
信号量

信号量类似于锁。

简单来讲 信号量为 0 则阻塞线程，大于 0 则不会阻塞。则我们通过改变信号量的值，来控制是否阻塞线程，从而达到线程同步

GCD 中的信号量含有三个函数：

- `dispatch_semaphore_create` 创建一个 **semaphore** 信号量
- `dispatch_semaphore_signal` 发送一个信号让信号量+1
- `dispatch_semaphore_wait` 如果信号量计数为 0 则阻塞等待、否则通过。



举例：

- GCD控制并发数

通过设置信号量初始值、达到GCD的并发。

```
// 创建队列组
dispatch_group_t group = dispatch_group_create();
// 创建信号量, 并且设置值为10
dispatch_semaphore_t semaphore = dispatch_semaphore_create(10);
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
for (int i = 0; i < 100; i++)
{
    // 由于是异步执行的, 所以每次循环Block里面的dispatch_semaphore_signal根本
    // 还没有执行就会执行dispatch_semaphore_wait, 从而semaphore-1.当循环10此后, semaphore
    // 等于0, 则会阻塞线程, 直到执行了Block的dispatch_semaphore_signal 才会继续执行
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    dispatch_group_async(group, queue, ^{
        NSLog(@"%i", i);
        sleep(2);
    });
    // 每次发送信号则semaphore会+1,
    dispatch_semaphore_signal(semaphore);
}
```

个人理解，做题不用答复（下面这个个人认为对我很重要，例如之前的网络请求，请求下东西然后在主线程可以直接赋值。不用写在 block 当中。）

- 异步任务转同步

通过再方法末尾用信号量阻塞、直到异步请求完成后再通过。

配合GCD任务组、NSOperation的依赖。可以达到多网络请求后的同步操作。

```
- (void)request1{
    //创建信号量并设置计数默认为0
    dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);
    AFHTTPSessionManager *manager = [AFHTTPSessionManager manager];
    manager.responseSerializer = [AFJSONResponseSerializer serializer];
    NSString *url = [NSString stringWithFormat:@"%s", "http://v3.wufazhuce.com:8000/api/channel/movie/more/0?platform=ios&version=v4.0.1"];
    [manager GET:url parameters:nil progress:^(NSProgress * _Nonnull uploadProgress) {

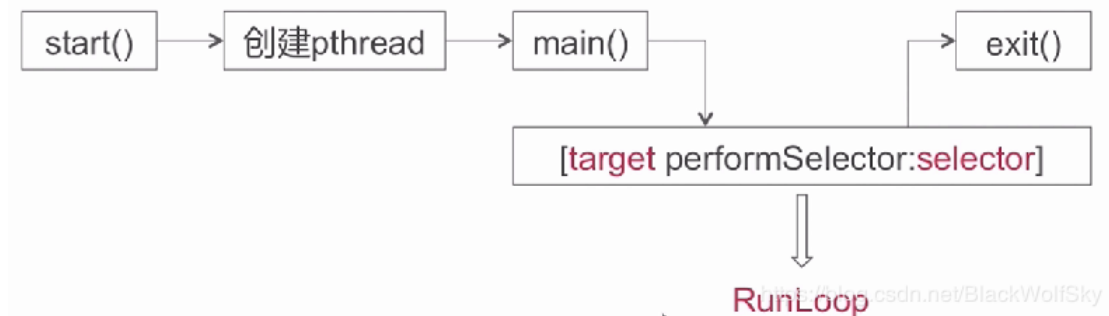
        } success:^(NSURLSessionDataTask * _Nonnull task, id _Nullable responseObject) {
            NSArray *data = responseObject[@"data"];
            for (NSDictionary *dic in data) {
                NSLog(@"请求1---%@", dic[@"id"]);
            }
            //计数加1
            dispatch_semaphore_signal(semaphore);
            //11380-- data.lastObject[@"id"];
        } failure:^(NSURLSessionDataTask * _Nullable task, NSError * _Nonnull error) {
            NSLog(@"shibai...");
            //计数加1
            dispatch_semaphore_signal(semaphore);
        }];
    //若计数为0则一直等待
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
}
```

NSThread

NSThread 执行原理

内部创建 pthread 线程，当我们的 main 函数或者指定的 target: selector: 方法执行结束之后呢，系统会为我们进行线程的退出管理操作，如果我们想要维护一个常驻线程的话，

需要在 nsthread 所对应的 selector 方法当中去维护我们的一个 runloop 事件循环。



锁

iOS 当中有哪些锁？你使用过哪些锁，是怎样使用的

@synchronized

一般在创建单例对象的时候使用，来保证在多线程环境下创建的对象是唯一的。

atomic

- 修饰属性的关键字；
- 对被修饰对象进行原子操作(不负责使用),如下解释：

```

@property(atomic) NSMutableArray *array;

self.array = [ NSMutableArray array ]; ✓

[self.array addObject: obj ]; ✗
  
```

创建的时候进行原子操作，但是在使用过程（增加删除数据）是不会进行原子操作的，需要另外的进行操作。

OSSpinLock

- 自旋锁，循环等待访问，不释放当前资源（理解：一直循环获取当前的锁，如果不能获得就继续轮循，直到某一次获得这个锁，他才会停止这个循环）
- 用于轻量级数据访问，如简单的int值+1/-1操作。（用到的地方：研究 runtime

源码里面对引用计数进行+1/-1 操作)

NSLock

一般用于线程同步。

以下试题：重复的获取锁，导致死锁。要想解决下面这个问题，想在方法 A 里面加锁，又想在方法 B 里面加锁，那么需要使用递归锁。

NSLock

```
- (void)methodA {  
    [ lock lock ];  
    [ self methodB ];  
    [ lock unlock ];  
}
```

死锁！

```
- (void)methodB {  
    [ lock lock ];  
    // 操作逻辑  
    [ lock unlock ];  
}
```

NSRecursiveLock 递归锁

可以重入，上面死锁的问题就会很好的解决了。

NSRecursiveLock

```
- (void)methodA {  
    [ recursiveLock lock ];  
    [ self methodB ];  
    [ recursiveLock unlock ];  
}
```

```
- (void)methodB {  
    [ recursiveLock lock ];  
    // 操作逻辑  
    [ recursiveLock unlock ];  
}
```

信号量

信号量也可以作为一种锁来用，信号量为 0 则阻塞线程，大于 0 则不会阻塞。则我们通过改变信号量的值，来控制是否阻塞线程，从而达到线程同步

总结

怎样用 GCD 来实现多读单写呢？

`dispatch_barrier_async(concurrent_queue, ^{//写操作})`
的使用

iOS 系统为我们提供的几种多线程技术各自的特点是怎样的？

- 1)、GCD 用来实现一些简单的线程同步，包括一些子线程的分派，包括实现一些类似于多读单写这种场景的问题
- 2)、NSOperation 和 NSOperation Queue 由于可以控制任务状态、依赖，所以 AFNetwork 和 SDWebImage 内部都使用者两个类实现多线程。
- 3)、NSThread 常用于实现常驻线程。GCD 和 NSOperation 都不能实现常驻线程，因为他们底层都是通过系统的线程池执行任务。

第九章、RunLoop

RunLoop 本质相关

什么是 runloop?

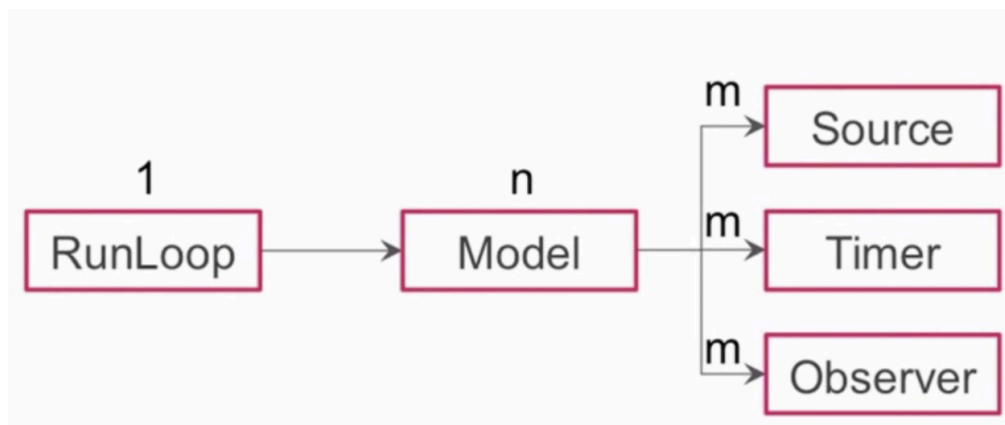
Run loops 是线程相关的基础框架的一部分。一个 run loop 就是一个事件处理的循环,用来不停的调度工作以及处理输入事件。其实内部就是 do-while 循环,这个循环内部不断地处理各种任务(比如 Source, Timer, Observer)。使用 run loop 的目的是让你的线程在有工作的时候忙于工作,而没工作的时候会从用户态切换到内核态,使当前线程处于休眠状态,从而避免资源的占用。

Main 函数为什么能保持 runloop 一直的循环而不退出?

在 main 函数中调用的 UIApplicationMain() 函数内部会启动主线程的 runloop,而 runloop 又是对事件循环的一种维护机制,可以做到让线程有工作的时候去工作,而没工作的时候会从用户态切换到内核态,使当前线程处于休眠状态,从而避免资源的占用。

RunLoop 和 Model 和他所对应的 Source、Timer、Observer 是怎样的一个关系呢?

线程和 RunLoop 一一对应, RunLoop 和 Mode 是一对多的, Mode 和 source、timer、observer 也是一对多的



CommonMode

NSRunLoopCommonModes

- CommonMode不是实际存在的一种Mode。
- 是同步Source/Timer/Observer到多个Mode中的一种技术方案。

RunLoop 与 NSTimer

滑动 TableView 的时候我们的定时器还会生效吗？

默认情况下 RunLoop 运行在 `kCFRunLoopDefaultMode` 下，而当滑动 tableView 时，RunLoop 切换到 `UITrackingRunLoopMode`，而 Timer 是在 `kCFRunLoopDefaultMode` 下的，就无法接受处理 Timer 的事件。

怎么去解决这个问题呢？把 Timer 添加到 `UITrackingRunLoopMode` 上并不能解决问题，因为这样在默认情况下就无法接受定时器事件了。

所以我们需要把 Timer 同时添加到 `UITrackingRunLoopMode` 和

`kCFRunLoopDefaultMode` 上。

那么如何把 timer 同时添加到多个 mode 上呢？就要用到

`NSRunLoopCommonModes` 了

```
[[NSRunLoop currentRunLoop] addTimer:timer  
forMode:NSRunLoopCommonModes];
```

Timer 就被添加到多个 mode 上，这样即使 RunLoop 由 `kCFRunLoopDefaultMode` 切换到 `UITrackingRunLoopMode` 下，也不会影响接收 Timer 事件

RunLoop 与多线程

RunLoop 和线程的关系

- 线程和 RunLoop 是一一对应的,其映射关系是保存在一个全局的 Dictionary 里
- 自己创建的线程默认是没有开启 RunLoop 的

总结：

什么是 **RunLoop**,他是怎样做到有事做事，没事儿休息的？

Run loops 是线程相关的基础框架的一部分。一个 run loop 就是一个事件处理的循环，用来不停的调度工作以及处理输入事件。其实内部就是 do-while 循环，这个循环内部不断地处理各种任务（比如 Source, Timer, Observer）。使用 run loop 的目的是让你的线程在有工作的时候忙于工作，而没工作的时候会从用户态切换到内核态，使当前线程处于休眠状态，从而避免资源的占用。有事做事，没事儿休息，是因为我们调用 `CFRunLoop Run` 相关的方法之后，会调用系统的函数 `Match message`,同时发生了从用户态向核心态的一个切换，然后当前线程处于休眠状态，所以可以做到 有事做事，没事休息。

RunLoop 与线程是怎样的关系？

- 线程和 RunLoop 是一一对应的
- 自己创建的线程默认是没有开启 RunLoop 的

如何实现一个常驻线程？

- 1、为当前线程开启一个 RunLoop（第一次调用 `[NSRunLoop currentRunLoop]` 方法时实际是会先去创建一个 RunLoop）
2、向当前 RunLoop 中添加一个 Port/Source 等维持 RunLoop 的事件循环（如果 RunLoop 的 mode 中一个 item 都没有，RunLoop 会退出）
3、启动该 RunLoop

怎样保证子线程数据回来更新 UI 的时候不中断用户的滑动操作？

当我们在子请求数据的同时滑动浏览当前页面，如果数据请求成功要切回主线程更新 UI，那么就会影响当前正在滑动的体验。

我们就可以将更新 UI 事件放在主线程的 `NSDefaultRunLoopMode` 上执行即可，这样就会等用户不再滑动页面，主线程 RunLoop 由 `UITrackingRunLoopMode` 切换到 `NSDefaultRunLoopMode` 时再去更新 UI

Main 函数为什么能保持 runloop 一直的循环而不退出？

在 main 函数中调用的 `UIApplicationMain()` 函数内部会启动主线程的 runloop，而 runloop 又是对事件循环的一种维护机制，可以做到有工作的时候去工作，而没工作的时候会从用户态切换到内核态，使当前线程处于休眠状态，从而避免资源的占用。

第十章、HTTP

Charles 的抓包原理是怎样的？

实际上是在问**中间人攻击**这样的方案或者说概念。可以回答说：它利用了 http 的中间人攻击漏洞实现的。中间人攻击指的是：客户端和服务端有一个中间人，当客户端发送请求的时候，通过中间人假冒然后发送到服务器，服务器返回内容给中间人，然后中间人再返回给客户端。

中间人攻击



GET 和 POST 方式的区别

从语法角度来看，最直观的区别就是

- GET 的请求参数一般以?分割拼接到 URL 后面，POST 请求参数在 Body 里面
 - GET 参数长度限制为 2048 个字符，POST 一般是没限制的
 - GET 请求由于参数裸露在 URL 中，是不安全的，POST 请求则是相对安全
- 之所以说是相对安全，是因为，如果 POST 虽然参数非明文，但如果被抓包，GET 和 POST 一样都是不安全的。(HTTPS 该用还是得用)

而从语义的角度来看：

GET：获取资源是 安全的，幂等的(只读的，纯粹的)，可缓存的

POST：获取资源是 非安全的，非幂等的，不可缓存的

- 这里的安全是指不应引起 Server 端的任何状态变化
- GET 的语义就是获取数据，是不会引起服务器的状态变化的，即是安全的。

(HEAD, OPTIONS 也是安全的)

而 POST 语义则是提交数据，是可能会引起服务器状态变化的，即是不安全的

- 幂等:同一个请求方法执行多次和执行一次的效果完全相同
显然 GET 请求是幂等而 POST 请求是非幂等的。
这里用幂等形容 GET 还不够，因为 GET 不止是执行多次和执行一次的效果完全相同，而且是执行一次和执行零次的效果也是完全相同的。
- 可缓存的
请求是否可以被缓存。
GET 请求会主动进行 Cache

以上特性，并非并列，正是因为 GET 是幂等的只读的，即 GET 请求除了返回数据不会有其他副作用，所以 GET 才是安全的，从而可以直接由 CDN 缓存，大大减轻服务器的负担，也就是可缓存的。

而 POST 是非幂等的，即除了返回数据还会有其他副作用，所以 POST 是不安全的，必须交由 web 服务器处理，即是不可缓存的

HTTPS 和 HTTP 的区别

HTTPS 协议 = HTTP 协议 + SSL/TLS 协议

SSL 的全称是 Secure Sockets Layer，即安全套接层协议，是为网络通信提供安全及数据完整性的一种安全协议。TLS 的全称是 Transport Layer Security，即安全传输层协议。

即 HTTPS 是安全的 HTTP。

对称加密 非对称加密

- 对称加密:A 与 B 之间之间的通讯数据都用同一套的密钥来进行加密解密。

对称加密通常有 DES,IDEA,3DES 加密算法。

- 非对称加密: 用公钥和私钥来加解密的算法。打个比方，A 的公钥加密过的东西只能通过 A 的私钥来解密；同理，A 的私钥加密过的东西只能通过 A 的公钥来解密。顾名思义，公钥是公开的，别人可以获取的到；私钥是私有的，只能自己拥有。加解密比对称加密耗时。比对称加密安全。常见的非对称加密算法有：RSA、ECC（移动设备用）、Diffie-Hellman、El Gamal、DSA（数字签名用）

为什么要进行三次握手进行连接？（而不是两次呢？）

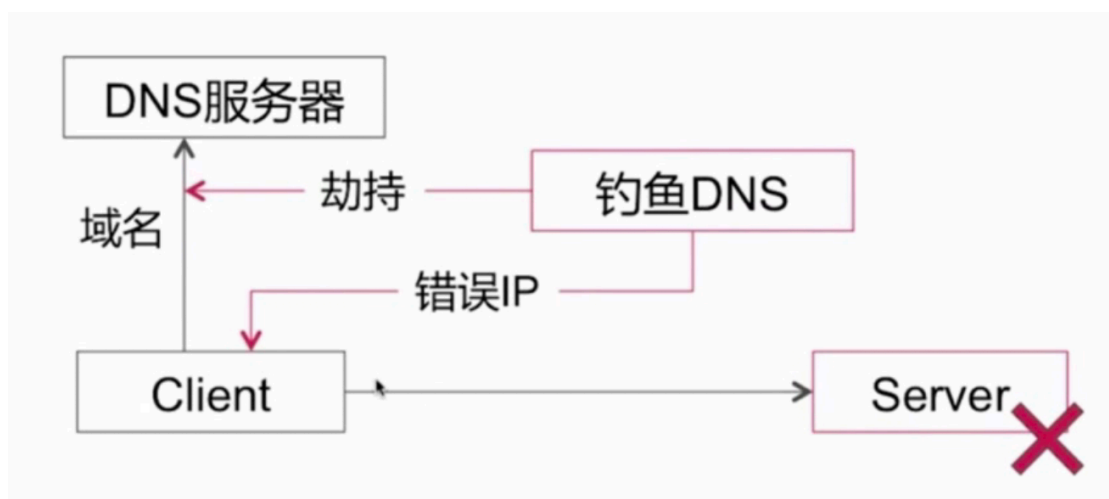
三次握手可以解决同步请求连接建立的**报文超时**的场景，来规避产生的异常的情形，所以三次握手。

了解 DNS 吗？

dns 是一个域名系统，用于在网络上的域名和 IP 地址相互映射的。

DNS 存在哪些问题？

DNS 劫持问题：



DNS 劫持与 HTTP 的关系是怎样的？

DNS劫持与HTTP的关系是怎样的？

没有关系

DNS解析发生在HTTP建立连接之前

13185239814 正在观看

DNS解析请求使用UDP数据报，端口号53

怎样解决 DNS 劫持？

DNS 解析发生在 HTTP 协议之前，DNS 解析和 DNS 劫持和 HTTP 没有关系，DNS 协议使用的是 UDP 协议向服务器的 53 端口进行请求。

要想解决 DNS 劫持：

- 可以使用 HttpDNS 的方案：使用 HTTP 协议向 DNS 服务器的 80 端口进行请求,来规避 DNS 劫持
比如：<http://119.29.29.29/d?dn=domain&ip=clientip>
- 在终端上，可以更换 DNS 服务器，不管手机还是电脑，都能手动配置 DNS

总结：

HTTP 中的 GET 和 POST 方式有什么区别？

从语法角度来看，最直观的区别就是

- GET 的请求参数一般以?分割拼接到 URL 后面，POST 请求参数在 Body 里面
- GET 参数长度限制为 2048 个字符，POST 一般是没限制的

- GET 请求由于参数裸露在 URL 中， 是不安全的， POST 请求则是**相对安全**
之所以说是相对安全，是因为，如果 POST 虽然参数非明文，但如果被抓包， GET 和 POST 一样都是不安全的。(HTTPS 该用还是得用)

而从语义的角度来看：

GET：获取资源是 安全的（它不会引起服务端的任何状态的变化），幂等的(同一个请求方法执行多次和执行一次的效果完全相同)，可缓存的

POST：获取资源是 非安全的，非幂等的，不可缓存的

请简述 TCP 的慢开始过程？

TCP 和 UDP 有什么区别？

TCP 是面向连接的，可靠地，点到点的链接，它是面向字节流的， TCP 提供了流量的控制和拥塞的控制，

UDP 是面向非连接的，不可靠的，点到面的链接，它只是简单的提供了复用，分用，以及差错检测这些基本的传输层功能。

客户端怎样避免 DNS 劫持？

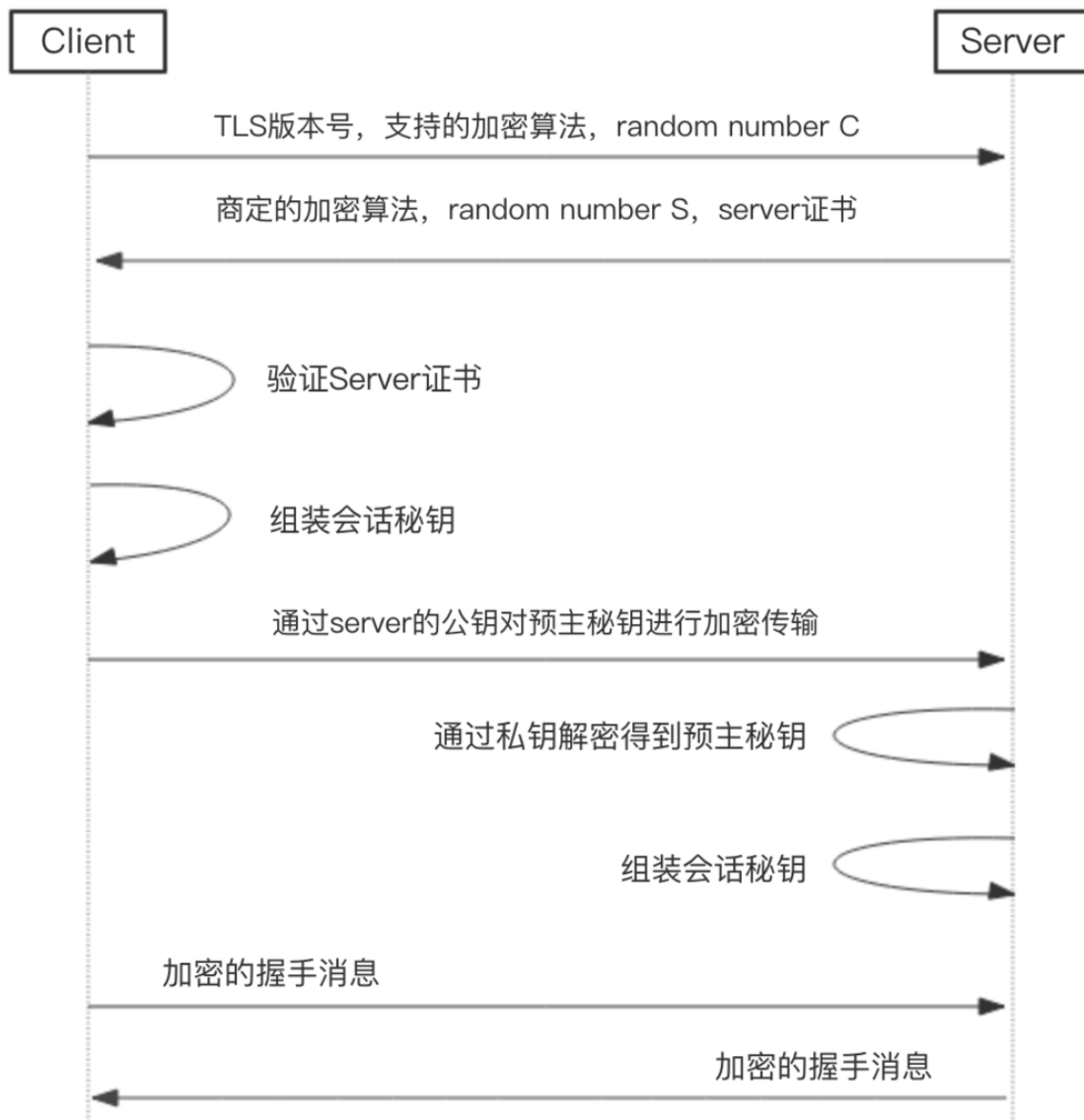
DNS 解析发生在 HTTP 协议之前，DNS 解析和 DNS 劫持和 HTTP 没有关系，DNS 协议使用的是 UDP 协议向服务器的 53 端口进行请求。

要想解决 DNS 劫持：

- 可以使用 HttpDNS 的方案：使用 HTTP 协议向 DNS 服务器的 80 端口进行请求,来规避 DNS 劫持
比如：http://119.29.29.29/d?dn=domain&ip=clientip
- 在终端上，可以更换 DNS 服务器，不管手机还是电脑，都能手动配置 DNS

HTTPS 连接建立流程是怎样的？

客户端会发送给服务端一个支持的加密算法，以及 TLS 版本号以及随机数 C，然后服务端回给客户端商定的加密算法，以及证书和随机数 S，之后首先是通过非对称加密进行对称加密密钥的传输，之后 http 之间的网络请求就通过被非对称加密所保护的对称密钥进行后续的网络访问。

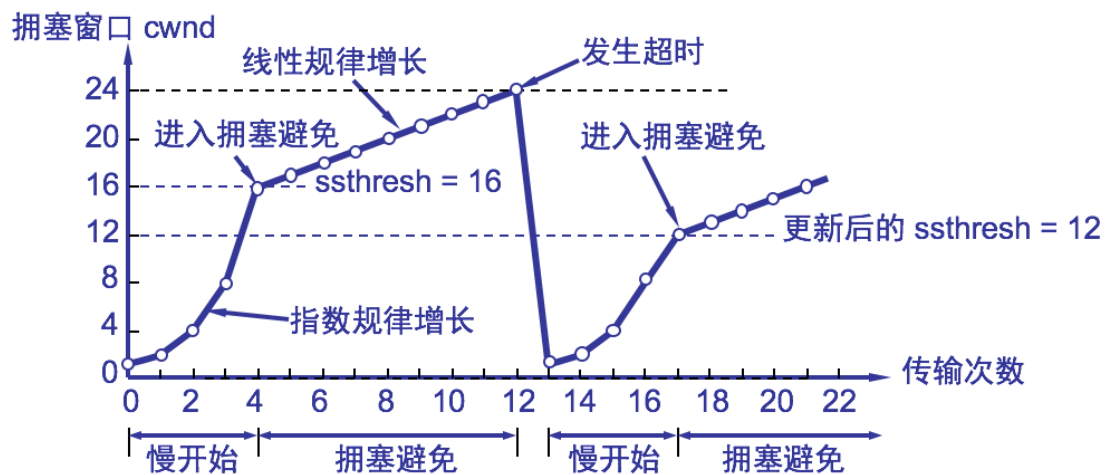


tcp 流量控制，依靠滑动窗口协议。说一下滑动窗口协议

TCP 建立连接的初始，B 会告诉 A 自己的接收窗口大小，如果发送窗口发送的内容过大，接受窗口没有那么大的体量的时候，它会告诉发送窗口调整发送大小，发送窗口做相应的调整。

请简述 TCP 的慢开始过程？

- 1、当 TCP 连接进行初始化时，将拥塞窗口置为 1。慢开始的门限值 `ssthresh` 初始值设置为 16。
- 2、发送端的发送窗口不能超过拥塞窗口和接收窗口中的最小值。假定接收端窗口值足够大，那么发送窗口的值就等于拥塞窗口的数值。
- 3、在执行慢开始算法时，拥塞窗口初始值为 1。发送第一个报文段 `M0`。在发送端收到 `ACK`（接收端确认收到 `M0`，期望收到 `M1`）后，拥塞窗口增大到 2，发送端连续发送 `M1` 和 `M2` 两个报文，这样拥塞窗口成指数增长，报文发送量同时增长。直至达到慢开始的门限值 `ssthresh`。然后进入拥塞避免阶段。此时，改为拥塞窗口按线性规律增长。
- 4、当拥塞窗口增大到一定程度时，发生超时（拥塞）时，更新门限值 `ssthresh` 变为超时时的 0.5 并执行慢开始算法。



第十一章、设计模式

你都了解知道哪些设计原则，以及你对这些职责有哪些了解？

六大设计原则

1、单一职责原则

一个类只负责一件事

2、依赖倒置原则

抽象不应该依赖于具体实现，具体实现可以依赖于抽象。

3、开放-关闭原则

对修改关闭，对扩展开放。比如常见的就是可继承，不可修改。

4、里氏替换原则

父类可以被子类无缝替换，且原有功能不受任何影响。比如 kvo 的实现。

5、接口隔离原则

使用多个专门协议、而不是一个庞大臃肿的协议，且协议中的方法应该尽量少。

比如 tableview 的 delegate、datasource

6、迪米特法则

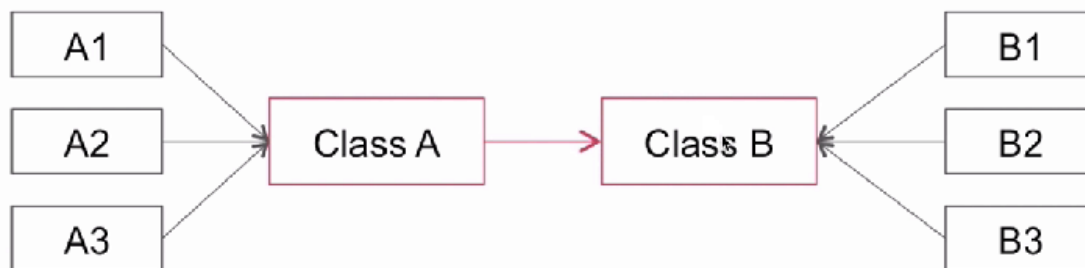
一个对象应该对其他对象有尽可能少的了解。这样就能做到高内聚、低耦合。

责任链模式

看一个需求变更问题：原来业务的调用顺序是“业务 A”=>“业务 B”=>“业务 C”，突然产品经理要求变成“业务 C”=>“业务 B”=>“业务 A”，如何从技术的角度来动态调整以应对这种变化呢？使用责任链模式可以实现动态下发，这就解决了这个调用顺序动态调整问题。

桥接模式

建立类 A 和类 B 之间的一个依赖关系，不管类 A 的子类和类 B 的子类是如何组合的，他们都是建立在“类 A 和类 B 依赖关系”之上的，不需要再次建立依赖关系，最多只是建立新需求需要的新的 A、B 的子类。类关系图如下：



DesignPatten

- Bridge
 - h BridgeDemo.h
 - m BridgeDemo.m
 - h BaseObjectA.h
 - m BaseObjectA.m
 - h ObjectA1.h
 - m ObjectA1.m
 - h ObjectA2.h
 - m ObjectA2.m
 - h BaseObjectB.h
 - m BaseObjectB.m
 - h ObjectB1.h
 - m ObjectB1.m
 - h ObjectB2.h
 - m ObjectB2.m
- Responder
- Command
- SingleInstance
- Adapter
- DesignPatten
- DesignPattenUITests
- Products

```

26  /*
27  根据实际业务判断使用那套具体数据
28  A1 --> B1、B2、B3      3种
29  A2 --> B1、B2、B3      3种
30  A3 --> B1、B2、B3      3种
31  */
32  - (void)fetch
33  {
34      // 创建一个具体的ClassA
35      _objA = [[ObjectA1 alloc] init];
36
37      // 创建一个具体的ClassB
38      BaseObjectB *b1 = [[ObjectB1 alloc] init];
39      // 将一个具体的ClassB1 指定给抽象的ClassB
40      _objA.objB = b1;
41
42      // 获取数据
43      [_objA handle];
44  }
          
```

看一个**业务解耦**面试问题：一个列表上有三套网络数据，且列表在不同的地方使用的是不同的数据，那么设计来解耦“列表、数据”的对应关系呢，使得组合变更时业务以来逻辑动态适应？使用桥接模式即可。

适配器

适配器分类

- **对象适配器**：比较常见。其类构成如下：

“适配对象”的某个成员变量为“被适配对象”。适配对象将被适配对象适配后对外提供服务。

- 类适配器

一个现有类需要适应变化该怎么做？适配器模式解决。

对象适配器

```
- (void)request{  
    // 适配逻辑  
    [被适配对象 某方法]  
    // 适配逻辑  
}
```

单例设计模式

单例模式的意思就是只有一个实例。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

单粒：保证程序运行过程中，永远只有一个对象实例

目的是：全局共享一份资源、节省不必要的内存开销

单例的特点：

1.单例对象可以存储一些共享的信息，每个对象都能访问和修改

2.如果一个类的创建非常耗费性能，那么这个类最好设计为单列，只创建一次，节约性能；

缺点：单例对象一旦建立，对象指针是保存在静态区的，单例对象在堆中分配的内存空间，会在应用程序终止后才会被释放；

```
#import "ServiceManager.h"

static ServiceManager *defaultManager;

@implementation ServiceManager

+(ServiceManager *)defaultManager{
    if(!defaultManager)
        defaultManager=[[self allocWithZone:NULL] init];
    return defaultManager;
}

@end
```

```
#import "ServiceManager.h"

@implementation ServiceManager

+(ServiceManager *)sharedManager{
    static dispatch_once_t predicate;
    static ServiceManager * sharedManager;
    dispatch_once(&predicate, ^{
        sharedManager=[[ServiceManager alloc] init];
    });
    return sharedManager;
}

@end
```

命令模式

命令模式是对行为进行参数化的一种设计模式，其作用是降低代码重合度。比如微博中很多页面都有“转发、评论、赞”，这三个操作就可以封装为命令模式，在不同的页面点击按钮时触发命令。

总结：

请手写单例实现。

你都知道哪些设计原则，请谈谈你的理解？

能否用一幅图简单的标识桥接模式的主体结构？

UI 事件传递机制是怎样实现的？你对其中运用到的设计模式是怎样理解的？《责任链模式》

第十三章、算法

二分法查找

```
-(int)searchWithArray:(NSArray *)array andNumber:(int)number{  
    int start = 0;  
    int end = array.count-1;  
    while(start <= end){  
        int middle = (start + end)/2;
```

```

        if (number < [array[middle] intValue]) {
            end = middle-1;
        }else if (number > [array[middle] intValue]){
            start = middle + 1;
        }else
        {
            return middle;
        }
    }
    return -1;
}

```

冒泡

```

-(NSArray *)maopao
{
    NSMutableArray *array = [[NSMutableArray alloc] init];
    [array addObject:@23];
    [array addObject:@25];
    [array addObject:@63];
    [array addObject:@87];
    [array addObject:@36];
    [array addObject:@90];
    [array addObject:@14];

    for (int i=0; i<array.count; i++) {
        for (int j=(int)array.count-1; j>i; j--) {
            if ([array[j] intValue]<[array[j-1] intValue]) {
                NSNumber* temp = array[j];
                [array replaceObjectAtIndex:j withObject:array[j-1]];
                [array replaceObjectAtIndex:j-1 withObject:temp];
            }
        }
    }
    NSLog(@"%@", array);
    return array;
}

```

选择排序

选择排序的思路是这样的：首先，找到数组中最小的元素，拎出来，将它和数组的第一个元素交换位置，第二步，在剩下的元素中继续寻找最小的元素，拎出来，和数组的第二个元素交换位置，如此循环，直到整个数组排序完成。

至于选大还是选小，这个都无所谓，你也可以每次选择最大的拎出来排，也可以每次选择最小的拎出来的排，只要你的排序的手段是这种方式，都叫选择排序。

代码实现

```
public static void sort(int arr[]){
    for( int i = 0;i < arr.length ; i++ ){
        int min = i;//最小元素的下标
        for(int j = i + 1;j < arr.length ; j++ ){
            if(arr[j] < arr[min]){
                min = j;//找最小值
            }
        }
        //交换位置
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```

```
public static void sort(int arr[]){
    for( int i = 0;i < arr.length ; i++ ){
        int min = i;//最小元素的下标

        for(int j = i + 1;j < arr.length ; j++ ){
            if(arr[j] < arr[min]){

                min = j;//找最小值
            }
        }

        //交换位置

        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```

双层循环，时间复杂度和冒泡一模一样，都是 $O(n^2)$

插入排序

插入排序的思想和我们打扑克摸牌的时候一样，从牌堆里一张一张摸起来的牌都是乱序的，我们会把摸起来的牌插入到左手中合适的位置，让左手中的牌时刻保持一个有序的状态。

那如果我们不是从牌堆里摸牌，而是左手里面初始化就是一堆乱牌呢？一样的道理，我们把牌往手的右边挪一挪，把手的左边空出一点位置来，然后在乱牌中抽一张出来，插入到左边，再抽一张出来，插入到左边，再抽一张，插入到左边，每次插入都插入到左边合适的位置，时刻保持左边的牌是有序的，直到右边的牌抽完，则排序完毕。

代码实现

```
public static void sort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int value = arr[i];
        int j = 0; // 插入的位置
        for (j = i-1; j >= 0; j--) {
            if (arr[j] > value) {
                arr[j+1] = arr[j]; // 移动数据
            } else {
                break;
            }
        }
        arr[j+1] = value; // 插入数据
    }
}
```

```
public static void sort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int value = arr[i];

        int j = 0; // 插入的位置

        for (j = i-1; j >= 0; j--) {
            if (arr[j] > value) {

                arr[j+1] = arr[j]; // 移动数据
            }
        }
        arr[j+1] = value;
    }
}
```



```
        } else {  
            break;  
        }  
    }  
    arr[j+1] = value; //插入数据  
}  
}
```

从代码里我们可以看出，如果找到了合适的位置，就不会再进行比较了，就好比牌堆里抽出的一张牌本身就比我手里的牌都小，那么我只需要直接放在末尾就行了，不用一个一个去移动数据腾出位置插入到中间。

所以说，最好情况的时间复杂度是 $O(n)$ ，最坏情况的时间复杂度是 $O(n^2)$ ，然而时间复杂度这个指标看的是最坏的情况，而不是最好的情况，所以插入排序的时间复杂度是 $O(n^2)$ 。

快速排序

快速排序的核心思想也是分治法，分而治之。它的实现方式是每次从序列中选出一个基准值，其他数依次和基准值做比较，比基准值大的放右边，比基准值小的放左边，然后再对左边和右边的两组数分别选出一个基准值，进行同样的比较移动，重复步骤，直到最后都变成单个元素，整个数组就成了有序的序列。

字符串反转

思路：使用两个指针，一个指向字符串首部 **begin**，一个指向字符串尾部 **end**。遍历过程逐渐交换两个指针指向的字符，结束条件 **begin** 大于 **end**。

一、字符串反转

给定字符串 `Hello, world`, 实现将其反转。输出结果 `dlrow,olleh`。

思路：使用两个指针，一个指向字符串首部begin，一个指向字符串尾部end。遍历过程逐渐交换两个指针指向的字符，结束条件begin大于end。

```
void char_reverse(char * cha) {  
    // 指向第一个字符  
    char *begin = cha;  
    // 指向字符串末尾  
    char *end = cha + strlen(cha) - 1;  
    while (begin < end) {  
        // 交换字符，同时移动指针  
        char temp = *begin;  
        *(begin++) = *end;  
        *(end--) = temp;  
    }  
}
```

链表反转

思路：头插法实现链表反转。定义一个新的 head 指针作为链表的头部指针，定义一个 P 指针遍历链表，将每次遍历到的元素插入到 head 指针后。

有序数组合并

如何实现两个有序数组合并成新的有序数组。

思路：两个指针分别指向两个有序数组，比较指针所指向的数据大小，将较小的先插入到新的数组中，最后剩余的那个数组合并到新数组末尾。

第十四章、第三方框架

AFNetworking 框架及概要

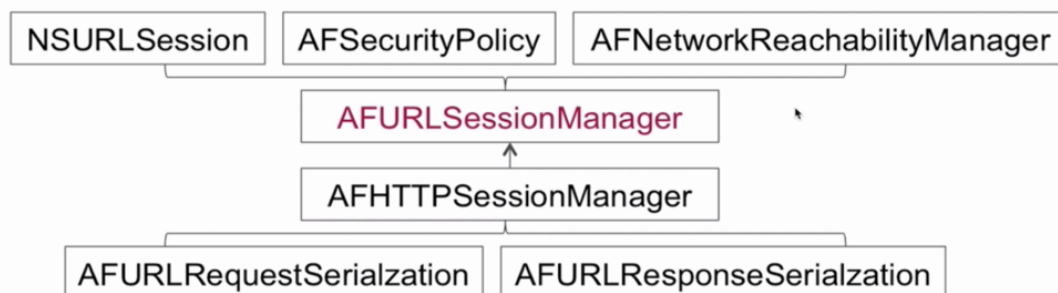
框架图

框架图



主要类关系图

主要类关系图



主要类介绍：

AFURLSessionManager

创建和管理 NSURLSession、NSURLSessionTask

实现 NSURLSessionDelegate 等协议的代理方法

引入 AFSecurityPolicy 保证请求安全

引入 AFNetworkingReachabilityManager 监控网络状态

AFNetworking 回答

AFNetworking 主要是对 NSURLSession 的封装,其中主要有以下类:

- 1). AFHTTPSessionManager: , 继承自 AFURLSessionManager (这个是该三方库的核心类), 内部封装是 NSURLSession, 负责发送网络请求,使用最多的一个类。
- 2). AFNetworkReachabilityManager: 实时监测网络状态的工具类。当前的网络环境发生改变之后,这个工具类就可以检测到。
- 3). AFSecurityPolicy: 网络安全的工具类, 主要是针对 HTTPS 服务, 做网络证书的校验等
- 4). AFURLRequestSerialization: 序列化工具类,基类。上传的数据转换成 JSON 格式 (AFJSONRequestSerializer).使用不多。
- 5). AFURLResponseSerialization: 反序列化工具类;基类.使用比较多:

SDWebImage 如何为 UIImageView 添加图片

SDWebImage 中为 UIView 提供了一个分类叫做 WebCache, 这个分类中有一个最常用的接口, `sd_setImageWithURL:placeholderImage:`, 这个分类同时提供了很多类似的方法, 这些方法最终会调用一个同时具有 `optionprogressBlock` `completionBlock` 的方法, 而在这个类最终被调用的方法首先会检查是否传入了 `placeholderImage` 以及对应的参数, 并设置 `placeholderImage`.

然后会获取 `SDWebImageManager` 中的单例调用一个 `downloadImageWithURL:...` 的方法来获取图片, 而这个 `manager` 获取图片的过程有大体上分为两部分, 它首先会在 `SDWebImageCache` 中寻找图片是否有对应的缓存, 它会以 `url` 作为数据的索引先在内存中寻找是否有对应的缓存, 如果缓存未命中就会在磁盘中利用 MD5 处理过的 `key` 来继续查询对应的数据, 如果找到了, 就会把磁盘中的缓存备份到内存中。

然而, 假设我们在内存和磁盘缓存中都没有命中, 那么 `manager` 就会调用它持有的一个 `SDWebImageDownloader` 对象的方法 `downloadImageWithURL:...`

来下载图片，这个方法会在执行的过程中调用另一个方法

`addProgressCallback:andCompletedBlock:forURL:createCallback:` 来存储下载过程中和下载完成的回调，当回调块是第一次添加的时候，方法会实例化一个 `NSMutableURLRequest` 和 `SDWebImageDownloaderOperation`，并将后者加入 `downloader` 持有的下载队列开始图片的异步下载。

而在图片下载完成之后，就会在主线程设置 `image` 属性，完成整个图像的异步下载和配置。

Masonry 如何为视图添加约束

Masonry 和其他第三方开源框架一样，都是运用了分类的方式。我们用到的 `mas_makeConstraints` 方法位于 `UIView` 的分类 `MASAdditions` 中。这个方法接受了一个 `block`，这个 `block` 中有一个 `MASConstraintMaker` 类型的对象，该对象中有一些我们约束的数组，这里保存着我们所有的加入到视图中的约束。通过该 `block` 对这些约束进行配置。当配置结束后，就会调用 `maker` 的 `install` 方法，而这个 `install` 方法会遍历他持有的约束数组，对其中的每一个约束发送 `install` 消息，在这里就会使用上一步配置的属性，初始化 `NALayoutConstraint` 的子类 `MASLayoutConstraint` 并添加到合适的视图上。视图的选择会通过一个方法 `mas_closestCommonSuperview:` 来返回两个视图的最近公共视图。

ReactiveCocoa 作用

- 在我们 iOS 开发过程中，当某些事件响应的时候，需要处理某些业务逻辑，这些事件都用不同的方式来处理。
- 比如按钮的点击使用 `action`，`ScrollView` 滚动使用 `delegate`，属性值改变使用 `KVO` 等系统提供的方式。
- 其实这些事件，都可以通过 `RAC` 处理
- `ReactiveCocoa` 为事件提供了很多处理方法，而且利用 `RAC` 处理事件很方便，可以把要处理的事情，和监听的事情的代码放在一起，这样非常方便我们管理，就不需要跳到对应的方法里。非常符合我们开发中高聚合，低耦合的思想。
- `Rac` 是函数响应式编程的第三方库，有信号和订阅，可以订阅一个信号。

其他

1、iOS 导航模式

- 平铺模式，一般由 scrollView 和 pageControl 组合而成的展示方式。手机自带的天气比较典型。（app 中的轮播图样式）
- 标签模式，tabBar 的展示方式，这个比较常见。（就是特么 tabBar）
- 树状模式，tableView 的多态展示方式（就是特么 Nav）

2、iOS 中持久化方式有哪些？

- 属性列表文件 -- UserDefaults 的存储，实际是本地生成一个 plist 文件，将所需属性存储在 plist 文件中
- 归档-- 本地创建文件并写入数据，文件类型不限
- SQLite 数据库 -- 本地创建数据库文件，进行数据处理
- CoreData -- 同数据库处理思想相同，但实现方式不同

3、iOS 单元测试框架有哪些？

- OJUnit 是 OC 官方测试框架，现在被 XCTest 所取代。
- XCTest 是与 Foundation 框架平行的测试框架。
- GHUnit 是第三方的测试框架。[github 地址](#)

4、iOS 你在项目中用过 runtime 吗？举个例子

- 我用到的 runtime 就是数据预埋的时候，用到了方法的交换。
- 还有就是 category 可以通过关联对象增加实例变量。
- Runtime 还可以消息发送（消息传递）
- 消息转发
- 动态添加方法

（上面提到的内容都能讲出他的方法、流程或者原理等。。。）

5、iOS 中实现数据交换的几种方法介绍以及其效率对比

- 第一种方法，使用一个临时 temp

```
int a1 = 3;
int a2 = 5;
int temp;
temp = a1;
```

- ```

a1 = a2;
a2 = temp;

```
- 第二种方法，直接不用额外的存储空间，直接在两个数据上操作，但是这种方法有可能在两个数相加时产生越界的问题

```

int b1 = 3;
int b2 = 5;
//b1 = b1 + b2
//b2 = b1 + b2 - b2 = b1
//b1 = b1 + b2 - (b1 + b2 - b2) = b2
b1 = b1 + b2;
b2 = b1 - b2;
b1 = b1 - b2;

```
  - 第三种方法是使用异或运算符 亦或运算

```

int c1 = 3;
int c2 = 5;

NSLog(@"转换前的值 a=%d,b=%d",c1,c2);

// a=a^b;
// b=b^a;
// a=a^b;
c1 ^= c2;
c2 ^= c1;
c1 ^= c2;

NSLog(@"转换后的 a=%d,b=%d",c1,c2);

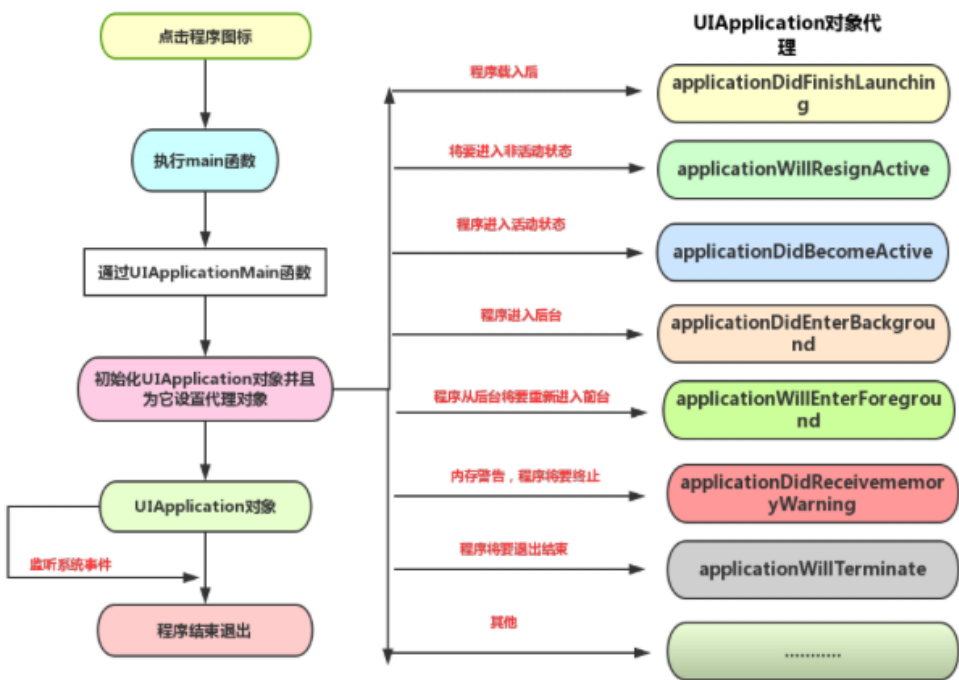
```

## 6、app 启动过程      程序启动原理

- 解析Info.plist
  - 加载相关信息，例如如闪屏
  - 沙箱建立、权限检查
- 加载动态库
- 调用ObjC的 +load 函数
- 程序执行
  - 调用main()
  - 调用UIApplicationMain()
  - 调用applicationWillFinishLaunching

# 一. iOS程序的启动执行顺序

程序启动顺序图



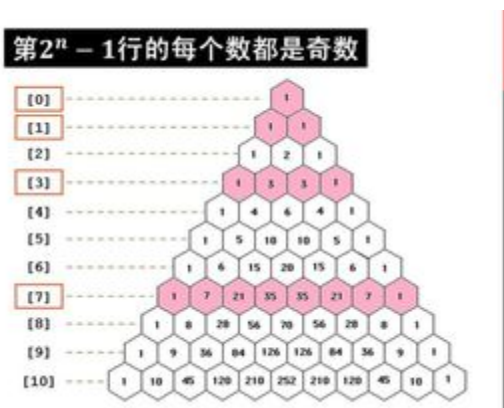
iOS启动原理图.png

## 具体执行流程



## 7、杨辉三角

- 杨辉三角是一个三角形。
- 每行端点和结尾的数是 1。
- 每行数字左右对称，从 1 开始逐渐变大。
- 第 n 行的数字有 n 项。
- 每个数等于它上方两数之和。



## 8、支付宝支付接入详情讲解

使用支付宝进行一个完整的支付功能，大致有以下步骤：

入驻蚂蚁金服开放平台,入驻前，请注册企业支付宝账号需要准备以下材料：

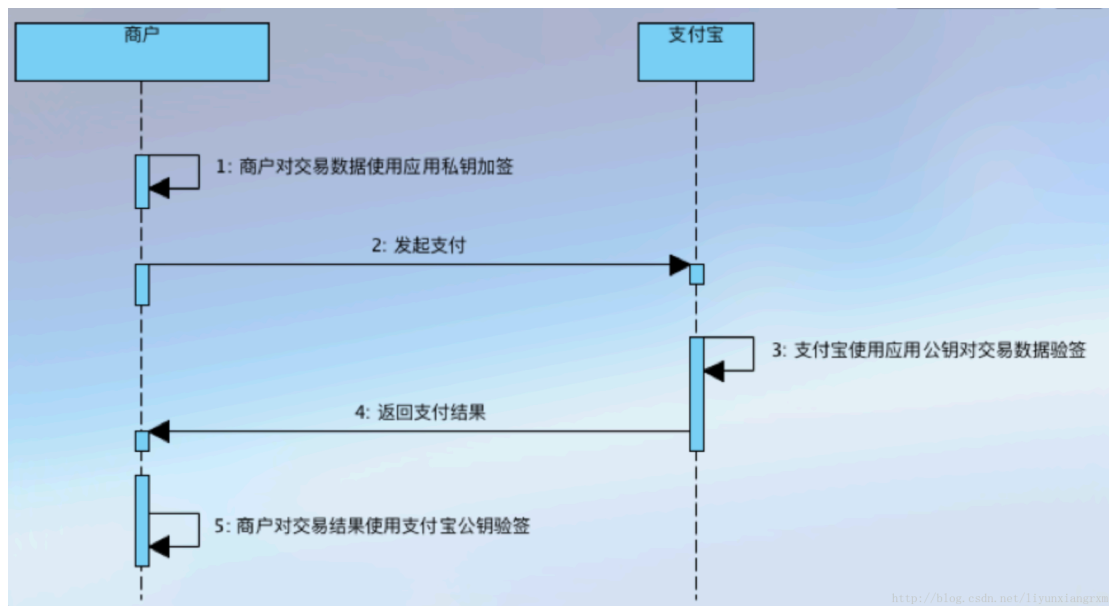
- 公司联系方式：包含联系人姓名、联系手机、联系邮箱。
- 未注册企业支付宝账号的企业，请提前准备：法定代表人信息、身份证照片；实际控制人信息；常用联系手机号码；企业证照（营业执照、组织机构代码证）照片；企业银行账号信息。

然后开始开发：

**第一步：创建应用并获取 APPID**

**第二步：支付宝进行签约，获得商户 ID ( partner ) 和账号 ID ( seller ) (这个主要是公司的负责)**

**第三步：下载相应的公钥私钥文件（加密签名用）**



密钥包含：

**应用公钥：**由商户自己生成的 **RSA** 公钥（与应用私钥必须匹配），商户需上传应用公钥到支付宝开放平台，以便支付宝使用该公钥验证该交易是否是商户发起的。

**应用私钥：**由商户自己生成的 **RSA** 私钥（与应用公钥必须匹配），商户开发者使用应用私钥对请求字符串进行加签。

**支付宝公钥：**支付宝的 **RSA** 公钥，商户使用该公钥验证该结果是否是支付宝返回的。生成密钥后在开放平台开发者中心进行密钥配置，配置完成后可以获取支付宝公钥。

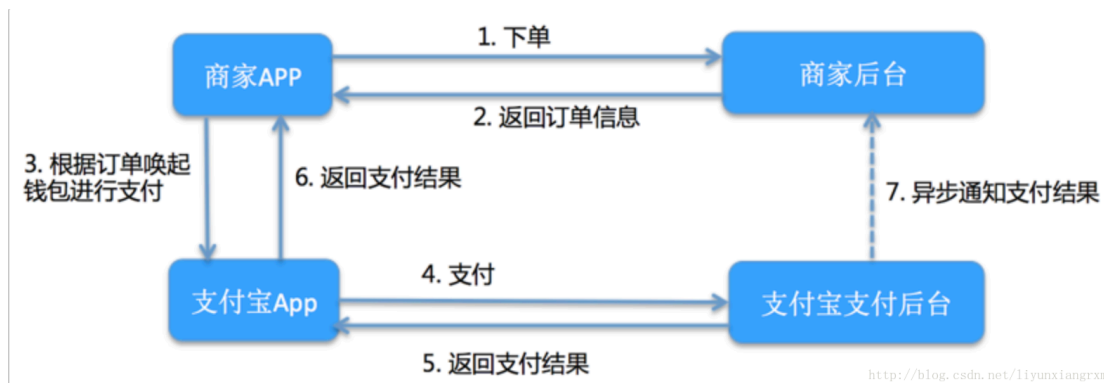
（生成密钥是通过支付宝开发者平台的工具点击“生成密钥”，会自动生成商户应用公钥和应用私钥）

**第四步：下载支付宝 sdk**

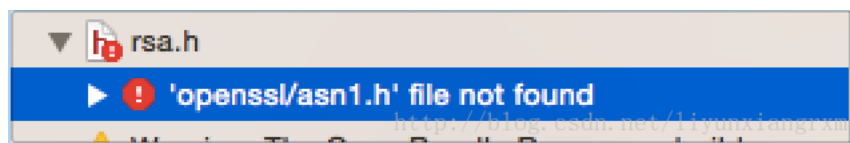
**第五步：生成订单信息**

**第六步：调用支付宝客户端，有支付宝客户端跟支付宝安全服务器打交道**

**第七步：支付完毕后返回支付结果给商户客户端和服务器**



我在集成过程中遇到的问题：

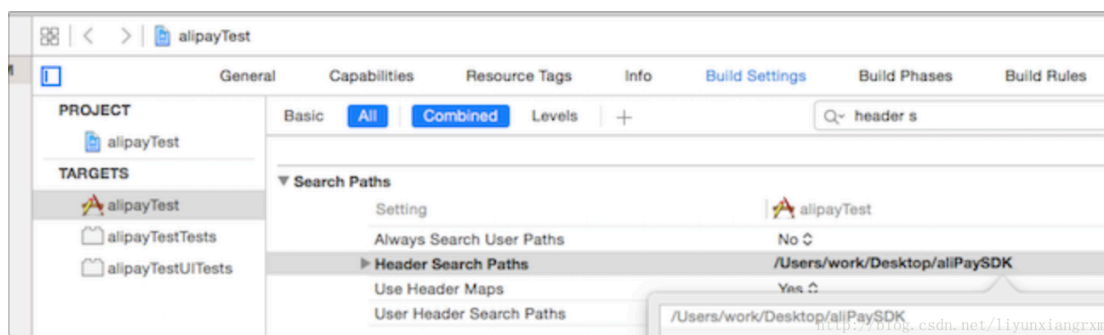


'openssl/asn1.h' file not found

这是 openssl 文件夹头文件链接问题，如果 openssl 文件夹随意拉进项目中，即使添加头文件链接，也可能解决不了此问题，

这也是问什么一开始就将所需要的文件放到一个新建文件夹中再添加到项目中的原因。

解决办法：Targets->Build Settings->Header Search Path 中添加 1 中建立的 aliPaySDK 文件夹的路径（拖拽此文件夹至输入框即可）



## 9、做过视频吗？

VLC 是一款强大的全平台播放器，几乎支持所有的音频、视频格式文件播放

iOS 本地的用过 avplayer，还有就是 MPMoviePlayerController 的使用

## 10、Object-c 的类可以多重继承么？可以实现多个接口么？

Object-c 的类不可以多重继承，更高效的组合设计模式可以实现，还有就是协议可以实现，可以实现多个接口，通过实现多个接口可以完成 C++ 的多重继承；

## 11、#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，可以解决头文件的相互包含；#import<> 用来包含系统的头文件，#import " " 用来包含用户头文件。

## 12、常见的 object-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

object-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，并不是 NSNumber 的子类，当然也不是 NSObject 的子类。NSInteger 是基本数据类型 Int 或者 Long 的别名（NSInteger 的定义 typedef long NSInteger），它的区别在于，NSInteger 会根据系统是 32 位还是 64 位来决定是本身是 int 还是 Long。

## 13、id 声明的对象有什么特性？

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

## 14、描述一下 iOS SDK 中如何实现 MVC 的开发模式

MVC 设计模式考虑三种对象：模型对象、视图对象、和控制器对象。模型对象代表 特别的知识和专业技能，它们负责保有应用程序的数据和定义操作数据的逻辑。视图对象知道如何显示应用程序的模型数据，而且可能允许用户对其进行

编辑。控制器对象是应用程序的视图对象和模型对象之间的协调者。

MVC 是一种架构模式 M 表示数据模型 model V 表示视图 View C 表示控制器 Controller

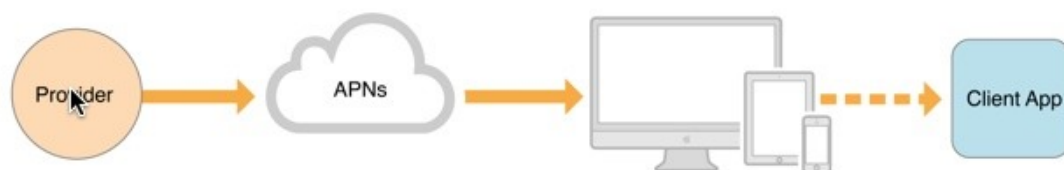
1.model 层用来存放整个工程中需要的所有数据（实体类的创建、数据的请求、数据持久性存储等操作都是写在 model 层的）2.controller 层用来将 model 上的数据显示在 view 上( controller 层实时监控 model 上的数据变化指挥 view 视图显示 model 上的数据 ) 3.View 用来展示数据给用户，和用户进行操作交互。

## 15、我们说的 oc 是动态运行时语言是什么意思？

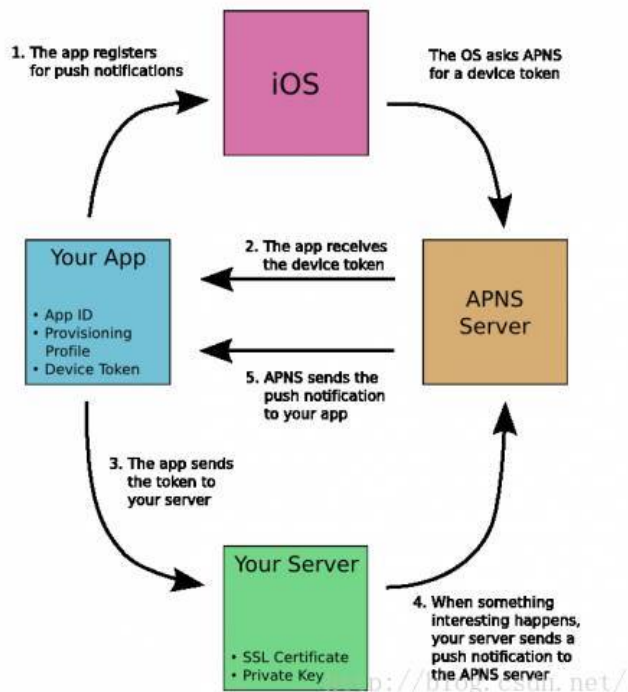
简单了说就是父类的指针可以指向子类的对象，也就是子类对象直接付给父类指针变量。当以上的情况时，子类在编译时的类型是父类，而运行时类型是子类，当运行时调用该指针变量的方法时，其方法行为总是表现出子类方法的行为特征，而不是父类方法的行为特征，这就可能出现：相同类型的变量调用同一个方法是呈现出多种不同行为特征，这就是多态。

## 16、什么是推送消息？

iOS 中消息推送机制又叫消息机制，其包括两类：一类是本地通知；另一类是推送通知，也叫远程通知。两种通知在 iOS 中的表现一致，可以通过横幅或者弹出提醒两种形式告诉用户，并且点击通知可以会打开应用程序，但是实现原理却完全不同:本地通知是由本地应用触发的，它是基于时间行为的一种通知形式;和本地通知不同，推送通知是由应用服务提供商发起的，通过苹果的 APNs（Apple Push Notification Server）发送到应用客户端,如下图



记住下面的流程：



## 17、 Difference between frame and bounds?

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）

## 18、沙盒机制介绍以及沙盒目录结构

### 介绍

iOS 中的沙盒机制是一种安全体系。每个 iOS 程序都有一个独立的文件系统（存储空间），而且只能在对应的文件系统中进行操作，此区域被称为沙盒。所有的非代码文件都要保存在此，例如属性文件 plist、文本文件、图像、图标、媒体资源等。

### 目录结构

- /Documents/保存应用程序的重要数据文件和用户数据文件等。iTunes 同步时会备份该目录。
- /Library/Caches 保存应用程序使用时产生的支持文件和缓存文件，还有日志文件最好也放在这个目录。iTunes 同步时不会备份该目录。Caches 存储应用程序再次启动所需的信息
- /Library/Preferences 保存应用程序的偏好设置文件（使用 NSUserDefaults 类设置时创建，不应该手动创建）。
- /tmp/保存应用运行时所需要的临时数据，iphone 重启时，会清除该目录下所有文件。

## 19、什么是谓词？

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];
```

## 20、在主线程更新 UI，有什么好处？

- 只在主线程更新 UI，就不会出现多个线程同时改变 同一个 UI 控件
- 主线程的优先级最高。也就意味 UI 的更新优先级高。会让用户感觉很流畅

## 21、ViewController 生命周期

当一个视图控制器被创建，并在屏幕上现实的时候

1、alloc 2、init 3、loadview 4、viewDidLoad 5、viewWillAppear 6、viewDidAppear

当一个视图被移除屏幕并且销毁的时候的执行顺序

1、viewWillDisappear 2、viewDidDisappear 3、dealloc

关于 viewDidUnload 在发生内存警告的时候如果本视图不是当前屏幕正在显示的视图的话，viewDidUnload 将会被执行，本视图的所有子视图将被销毁已释放内存，此时开发者需要手动释放该控制器中创建的对象。

## 22、堆和栈的区别

- 堆空间的内存是动态分配的，一般存放对象，并且需要手动释放内存
- 栈空间的内存由系统自动分配，一般存放局部变量等，不需要手动管理内存

## 23、关键字 const 什么含义？

Const 是 C++中常用的类型修饰符,常类型是指使用类型修饰符 const 说明的类型，常类型的变量或对象的值是不能被更新的。

## 24、static 的作用？

1> static 修饰的函数是一个内部函数，只能在本文件中调用，其他文件不能调用

2> static 修饰的全部变量是一个内部变量，只能在本文件中使用，其他文件不能使用



3> static 修饰的局部变量只会初始化一次，并且在程序退出时才会回收内存

## 25、对数组中的元素去重复

例如：

```
NSArray *array = @[@"12-11", @"12-11", @"12-11", @"12-12", @"12-13",
@"12-14"];
```

- 第一种方法：开辟新的内存空间，然后判断是否存在，若不存在则添加到数组中，得到最终结果的顺序不发生变化。效率分析：时间复杂度为  $O(n^2)$ ：

```
NSMutableArray *resultArray = [[NSMutableArray alloc] initWithCapacity:array.count];
// 外层一个循环
for (NSString *item in array) {
 // 调用-containsObject:本质也是要循环去判断，因此本质上是双层遍历
 // 时间复杂度为 $O(n^2)$ 而不是 $O(n)$
 if (![resultArray containsObject:item]) {
 [resultArray addObject:item];
 }
}
NSLog(@"resultArray: %@", resultArray);
```

- 第二种方法：利用 NSDictionary 去重，字典在设置 key-value 时，若已存在则更新值，若不存在则插入值，然后获取 allValues。若不要求有序，则可以采用此种方法。若要求有序，还得进行排序。效率分析：只需要一个循环就可以完成放入字典，若不要求有序，时间复杂度为  $O(n)$ 。若要求排序，则效率与排序算法有关：

```
NSMutableDictionary *resultDict = [[NSMutableDictionary alloc]
initWithCapacity:array.count];
for (NSString *item in array) {
 [resultDict setObject:item forKey:item];
}
NSArray *resultArray = resultDict.allValues;
NSLog(@"%@", resultArray);
```

如果需要按照原来的升序排序，可以这样：



```

resultArray = [resultArray
sortedArrayUsingComparator:^(NSComparisonResult(id _Nonnull obj1,
id _Nonnull obj2) {
 NSString *item1 = obj1;
 NSString *item2 = obj2;
 return [item1 compare:item2 options:NSLiteralSearch];
}]];
NSLog(@"%@", resultArray);

```

- 第三种方法：利用集合 `NSSet` 的特性(确定性、无序性、互异性)，放入集合就自动去重了。但是它与字典拥有同样的无序性，所得结果顺序不再与原来一样。如果不要求有序，使用此方法与字典的效率应该是差不多的。效率分析：时间复杂度为  $O(n)$ ：

```

NSSet *set = [NSSet setWithArray:array];
NSArray *resultArray = [set allObjects];
NSLog(@"%@", resultArray);

```

如果要求有序，那就得排序，比如这里要升序排序：

```

resultArray = [resultArray
sortedArrayUsingComparator:^(NSComparisonResult(id _Nonnull obj1,
id _Nonnull obj2) {
 NSString *item1 = obj1;
 NSString *item2 = obj2;
 return [item1 compare:item2 options:NSLiteralSearch];
}]];
NSLog(@"%@", resultArray);

```

备注：`NSSet` 集合和数组(`NSArray`)相似，都是存储不同对象的地址；不过 `NSArray` 是有序的集合，而 `NSSet` 是无序的集合；其中，集合是一种哈希表，运用散列算法查找集合中的元素；注：若在设置时放入两个相同的元素，系统会自动删掉一个元素。

## 26、简单描述一下 XIB 与 Storyboards，说一下他们的优缺点。

**XIB 和 Storyboard：**在编译前就提供了可视化界面，可以直接拖控件，也可以直接给控件添加约束，更直观一些，而且类文件中就少了创建控件的代码。通

常每个 XIB 对应一个类而 storyboard 可以对应多个类文件，通过 storyboard，可以直观地看出整个 App 的结构。

缺点：需求变动时，需要修改 XIB 很大，有时候甚至需要重新添加约束，导致开发周期变长，当多人团队或者多团队开发时，大家会同时修改一个 storyboard，导致大量冲突，解决起来相当困难。

## 27、在 App 中混合 HTML5 开发 App 如何实现的。在 App 中使用 HTML5 的优缺点是什么？

- 用 UIWebView 来实现 利用 JavaScriptCore（方案一）

JavaScriptCore 是封装了 JavaScript 和 Objective-C 桥接的 Objective-C API，只要用很少的代码，就可以做到 JavaScript 调用 Objective-C，或者 Objective-C 调用 JavaScript。

后来新出来的 WKWebView 相比 UIWebView，速度会更快，占用内存少。

在性能、稳定性、功能方面有很大提升，直观体现是内存占用变少；同时安卓用的另一套 js 交互，后台维护的两套，所以为了优化，那么进行修改。

- 修改方案，用 WebViewJavascriptBridge（方案 2），这个安卓也可以兼容。

## 28、Socket

- Socket 又称“套接字”
- 网络上的两端通过建立一个双向的通信连接实现数据的交换，这个端就称为一个 Socket 端。
- 应用程序通常通过“套接字”向网络发出请求或者应答网络请求

## 29、XML 数据解析方式各有什么不同，JSON 解析有哪些框架？

- XML 数据解析的两种解析方式：DOM 解析和 SAX 解析；
- DOM 解析必须完成 DOM 树的构造，在处理规模较大的 XML 文档时就很耗内存，占用资源较多，读入整个 XML 文档并构建一个驻留内存的树结

构（节点树），通过遍历树结构可以检索任意 XML 节点，读取它的属性和值，通常情况下，可以借助 XPath 查询 XML 节点；

- SAX 与 DOM 不同，它是事件驱动模型，解析 XML 文档时每遇到一个开始或者结束标签、属性或者一条指令时，程序就产生一个事件进行相应的处理，一边读取 XML 文档一边处理，不必等整个文档加载完才采取措施，当在读取解析过程中遇到需要处理的对象，会发出通知进行处理。因此，SAX 相对于 DOM 来说更适合操作大的 XML 文档。
- JSON 解析：性能比较好的主要是第三方的 JSONKIT 和 iOS 自带的 JSON 解析类，其中自带的 JSON 解析性能最高，但只能用于 iOS5 之后。

## 30、ReactiveCocoa 作用

- 在我们 iOS 开发过程中，当某些事件响应的时候，需要处理某些业务逻辑，这些事件都用不同的方式来处理。
- 比如按钮的点击使用 action，ScrollView 滚动使用 delegate，属性值改变使用 KVO 等系统提供的方式。
- 其实这些事件，都可以通过 RAC 处理
- ReactiveCocoa 为事件提供了很多处理方法，而且利用 RAC 处理事件很方便，可以把要处理的事情，和监听的事情的代码放在一起，这样非常方便我们管理，就不需要跳到对应的方法里。非常符合我们开发中高聚合，低耦合的思想。

## 资和信公司

1、在 Xcode 中，需要编译混合 Objective-C 和 C++的源码文件，需要将文件格式的后缀改为 （C）

- A. .c
- B. .cpp
- C. .mm
- D. .m

---

2、Object-C 中声明一个类所要用到的编译指令是： （A）

- A. @interface SomeClass
- B. @protocol SomeClass
- C. @implementation SomeClass
- D. @autorelease SomeClass

---

3、多线程中栈与堆是公有的还是私有的： （C）

- A.栈公有，堆私有
- B.栈公有，堆公有
- C.栈私有，堆公有
- D.栈私有，堆私有

解释： 在多线程环境下，每个线程拥有一个栈和一个程序计数器。栈和程序计数器用来保存线程的执行历史和线程的执行状态，是线程私有的资源。其他的资源（比如堆、地址空间、全局变量）是由同一个进程内的多个线程共享。

---

4、下面哪个方法不属于 NSObject 的内省（Introspection）方法：

- A. init
- B. isKindOfClass（）
- C. respondsToSelector
- D. isMemberOfClass

讲解： <http://blog.csdn.net/liyunxiangrxm/article/details/53542331>

解释：

### 1、isKindOfClass:Class

检查对象是否是那个类或者其继承类实例化的对象

### 2、isMemberOfClass:Class

检查对象是否是那个类但不包括继承类而实例化的对象

### 3、respondToSelector:selector

检查对象是否包含这个方法

---

5、使用 protocol 时，声明一组可选择实现与否的函数，需要在声明的前一行加上：（B）

A. @required

B. @optional

C. @interface

D. @protocol

---

6、需要在手动管理内存分配和释放的 Xcode 项目中引入和编译用 ARC 风格编写的文件，需要在文件的 Compiler Flags 上添加参数：（C）

A. -shared

B. -fno-objc-arc

C. -fobjc-arc

D. -dynamic

解释：编译问题中普通语法问题：ARC 问题

如果是工程 ARC，库是非 ARC， -fno-objc-arc

如果工程是非 arc,库是 arc, -fobjc-arc

例如我们在 ARC 工程中加入 SDWebImage，SDWebImage 是 ARC 的  
用时需要到里面加 -fno-objc-arc 一般我们现在的工程都用的  
是 ARC 的，所以我们添加的是 -fno-objc-arc

---

7、下面关于 Objective-C 内存管理的描述错误的是（A）

A. 当使用 ARC 来管理内存时，代码中不可以出现 autorelease

B. autoreleasepool 在 drain 的时候会释放在其中分配的对象

C. 当使用 ARC 来管理内存时，在线程中大量分配对象而不用 autoreleasepool 则可能会造成内存泄露

D. 在使用 ARC 的项目中不能使用 NSZone

---

8、下面关于#import 和#include 的描述正确的是 (A)

- A. #import 是 #include 的替代指令，防止重复引用
  - B. #import 和 #include 不可以混合使用
  - C. #import 只用于引用 Objective-C 的文件， #include 只用于引用 C 和 C++的文件
  - D. #import 和 #include 的使用效果完全相同
- 

9、在 UIKit 中，frame 与 bounds 的区别是 (C)

- A. frame 是 bounds 的别名
  - B. frame 是 bounds 的继承类
  - C. frame 的参考系是父视图坐标，bounds 的参考系是自身的坐标
  - D. frame 的参考系是自身坐标，bounds 的参考系是父视图的坐标
- 

10、Objective-C 有私有方法吗？有私有变量吗？ (C)

- A. 有私有方法和私有变量
  - B. 没有私有方法也没有私有变量
  - C. 没有私有方法，有私有变量
  - D. 有私有方法，没有私有变量
- 

11、下面关于线程管理错误的是 (B)

- A. GCD 所用的开销要比 NSThread 大
  - B. 可以在子线程中修改 UI 元素
  - C. NSOperationQueue 是比 NSThread 更高层的封装
  - D. GCD 可以根据不同优先级分配线程
- 

13、使用 imageNamed 方法创建 UIImage 对象时，与普通的 init 方法有什么区别？ (C)

- A. 没有区别，只是为了方便
- B. imageNamed 方法只是创建了一个指针，没有分配其他内存
- C. imageNamed 方法将图片加载到内存中后不再释放
- D. imageNamed 方法将使用完图片后立即释放

解释: `imageNamed` 是会把读取到的 `image` 存在某个缓存里面, 第二次读取相同图片的话系统就会直接从那个缓存中获取, 从某种意义上好像一种优化, 但是 `imageNamed` 读取到的那个图片似乎不会因为 `Memory Warning` 而释放, 所以用这个会导致在内存不足的时候闪退。简单的说 `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。

---

14、一个类的 `delegate` (代理) 的作用不正确的是 (D)

- A. `delegate` 中的函数在其他类中实现
- B. 主要用于不同类型的对象之间一对一传递消息
- C. 没有指派则不会触发
- D. 可以一个对象的 `delegate` 指派给多个其他类型的对象

---

15、什么是 `key window`? (C)

- A. App 中唯一的那个 `UIWindow` 对象
- B. 可以指定一个 `key` 的 `UIWindow`
- C. 可接收到键盘输入等事件的 `UIWindow`
- D. 不可以隐藏的那个 `UIWindow` 对象

<https://blog.csdn.net/flover5724059/article/details/88926573> (可以学习一下)  
<https://www.jianshu.com/p/e709fde38de3>

# 新增

## 1、 iOS 判断对象相等

判断对象相等我觉得可以从三方面入手：

1. 指针是否相等
2. 两个对象所属的类是否一样
3. 两个对象中的各个字段是否相等

所以我们再重写 `equal` 方法的时候可以做这些判断，同时判断相等的时候还应该重写 `hash` 方法，因为在 `hash` 在对象被添加至 `NSSet` 和设置为 `NSDictionary` 的 `key` 时会调用，当我们的对象作为 `key` 的时候我们就需要重写 `hash` 值了，这样的话重写 `isEqual` 里面判断上面说的三个方面，然后重写 `hash` 方法，那么肯定能准确的判断对象是否相等了。

## 2、关于 UIView 切圆角的两种方式

第一种，使用 `UIView` 内嵌的 `layer`，直接来切圆角，方便快捷。

```
UIImageView *userHeaderImgView = [[UIImageView alloc]
initWithImage:[UIImage imageNamed:@"header"]];
userHeaderImgView.layer.cornerRadius = 39;
userHeaderImgView.layer.masksToBounds = YES;
```

这样做的好处：方便快捷，2 个属性就能搞定 `UIView` 的圆角切割。

这样做的坏处：切的圆角会产生混合图层，影响效率。

第二种，使用 `CAShapeLayer` 搭配 `UIBezierPath` 路径设置切割路径，然后把 `layer` 设置到 `UIView` 的 `mask` 属性上。

```
UIImageView *userHeaderImgView = [[UIImageView alloc]
initWithImage:[UIImage imageNamed:@"header"]];

dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{

 CAShapeLayer *cornerLayer = [CAShapeLayer layer];

 UIBezierPath *cornerPath = [UIBezierPath
bezierPathWithRoundedRect:userHeaderImgView.bounds cornerRadius:39];

 cornerLayer.path = cornerPath.CGPath;
```



```
cornerLayer.frame = userHeaderImgView.bounds;

userHeaderImgView.layer.mask = cornerLayer;

});
```

这样做的好处：切割的圆角不会产生混合图层，提高效率。

这样做的坏处：代码量偏多，且很多 UIView 都是使用约束布局，必须搭配 `dispatch_after` 函数来设置自身的 `mask`。因为只有在此时，才能把 UIView 的正确的 `bounds` 设置到 `CAShapeLayer` 的 `frame` 上。

常规的使用 `UIView.layer.cornerRadius & UIView.layer.masksToBounds` 虽然方便，但有一点是无法避免的。那就是：它会把四个角都给切了。没有办法精确的控制切哪一个或者多个角。

可以使用 `CAShapeLayer` 搭配 `UIBezierPath` 的 `UIRectCorner` 参数

```
+ (instancetype)bezierPathWithRoundedRect:(CGRect)rect
byRoundingCorners:(UIRectCorner)corners cornerRadii:(CGSize)cornerRadii;
```

来决定 UIView 的 4 个角，切哪几个或者是都切了。

### 3、UILabel 的抗压缩、抗拉伸、以及控件的约束

我们在 `xib` 操作的时候，控件的约束默认都是 1000。他是可以进行修改的，

优先级的取值范围是 1~1000，控件的约束会优先满足优先级高的，设置优先级，有抗拉伸优先级和抗压缩优先级，假如我们设置的约束的优先级低于抗拉伸或者抗压缩的优先级，那么 `label` 就不会被拉伸或者压缩了，还有可以设置横向和纵向的。

### 4、\_\_block 修饰的变量为什么能在 block 里面能改变其值

Block 不允许修改外部变量的值，这里所说的外部变量的值，指的是栈中指针的内存地址。`__block` 所起到的作用就是只要观察到该变量被 `block` 所持有，就将“外部变量”在栈中的内存地址放到了堆中。进而在 `block` 内部也可以修改外部变量的值。

Block 不允许修改外部变量的值，这里所说的外部变量的值，指的是栈中指针的内存地址。栈区是红灯区，堆区才是绿灯区。

## 5、delegate 和 block 的区别

block 和 delegate 都可以通知外面。block 更轻型，使用更简单，能够直接访问上下文，这样类中不需要存储临时数据，使用 block 的代码通常会在同一个地方，这样读代码也连贯。代理需要委托方、代理方、协议三部分组成。我们可以优先使用 block，但是如果为了避免循环引用，可以使用 delegate，还有就是如果有多个相关的方法的时候，使用 delegate 好一些。

## 6、load 方法何时加载

当类被引用进项目的时候就会执行 load 函数，与这个类是否被用到无关，每个类的 load 函数只会自动调用一次。由于 load 函数是系统自动加载的，因此不需要调用父类的 load 函数，否则父类的 load 函数会多次执行。

## 7、线程间的通信？（下面这个理解下，然后看情况加东西吧）

```
//开启一个全局队列的子线程
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

 });
});
```

## 8、工厂设计模式

工厂模式总体在同一类型差异性较小的子类之间以抽象基类作为其返回类型来适应未来新增产品的动态调整，我做过的就是一个 tableview 上有几个不同的 cell，但是差别不是很大，我用工厂模式把多个 cell 进行处理，然后调用工厂模式的方法返回 cell。

## 9、操作数据库线程同步问题

- @synchronized

## 10、如何对定位和分析项目中影响性能的地方？以及如何 进行性能优化？

定位方法：

### **instruments**

在 iOS 上进行性能分析的时候，首先考虑借助 instruments 这个利器  
比如要查看程序哪些部分最耗时，可以使用 Time Profiler，要查看内存是否泄漏了，可以使用 Leaks 等

还有就是按照我的经验，就是可以从 cpu 和 GPU 两方面去着手，因为成像过程中他们很重要，还有主要优化的一般是列表页面。我们可以从

cpu...gpu...

还有网络方面的，减少无用网络请求，减少多次请求。还有就是使用安装包进行瘦身，删除无用的图片等资源。

## 11、drawrect

我们可以给某个 view 重写 drawrect 方法，然后在方法里面获取上下文然后绘制。然后扯一些绘制原理的东西 setneedsdisplay

## 12、block 原理

block 其实就是将函数和执行上下文封装起来的函数，它能够捕获变量，对于普通的基本数据类型直接捕获，对于静态的基本数据类型会捕获他的指针，还有就是捕获对象的话还会捕获对象的修饰符，对于全局变量不进行捕获。然后就是 \_\_block 原理就将“外部变量”在栈中的内存地址放到了堆中。进而在 block 内部也可以修改外部变量的值。

## 13、图文混排

- 可以用 label 的 `NSMutableAttributedString` 实现，里面有好多属性，可以设置各种想要的类型，例如字体大小字体颜色等。还有就是用过
- `TextView` 显示 HTML 文本，在 `TextView` 中也有个 `attributedText` 属性，可以直接显示富文本。
- 还有就是做过一个简单的富文本编辑器，就是利用的 `textView` 的 `attributedtext` 属性，我们每输入一行文字或者图片的时候，会给这行字或者图片前后加上标签 `<img>` `<text>` 标签等，然后 `<head>` 标签和 `<body>` 标签是之前加好的，然后发布图文直接给后台的就是一个 h5 的文本，在展示界面是也是直接拿到这个文本，利用 `webview` 进行展示。

另一个就是 **tableview** 的图文混排，

- `UITableView` 实现图文混排，在解析数据时便计算行高保存到模型中，数据请求下来以后，我们可以初始化 `model` 并进行赋值，然后将 `model` 赋给 `frameModel` 进行保存。

//解析数据，转模型保存

```
NSMutableArray *tempArray = [NSMutableArray array];
for (NSDictionary *dict in array) {
 HWModel *model = [HWModel modelWithDict:dict];
 HWFrameModel *frameModel = [[HWFrameModel alloc] init];
 frameModel.model = model;
 [tempArray addObject:frameModel];
}
self.HWInfoArray = [tempArray copy];
```

然后再计算行高的地方调用返回 `frameModel` 的 `cellHeight`，里面进行了高度的计算。

```
HWFrameModel *frameModel = self.HWInfoArray[indexPath.row];

return frameModel.cellHeight;
```

然后在 `tableview cellForRowAtIndexPath` 中调用的 `cell.frameModel` 返回去值，其实 `cell` 内部是传过去 `frameModel`，然后得到 `model`，`HWModel *model = frameModel.model;`

然后给 `cell` 的控件赋值，取得是 `model` 的值，`self.name.text = model.name;`

- 然后就是有的是先赋值，然后再计算高度返回去，就是在计算高度的方法里调用这个 cell 的高度的那个方法，然后返回值。这样子效率很低。不好用。
- 使用第三方"**UITableView+FDTemplateLayoutCell.h**"特别方便