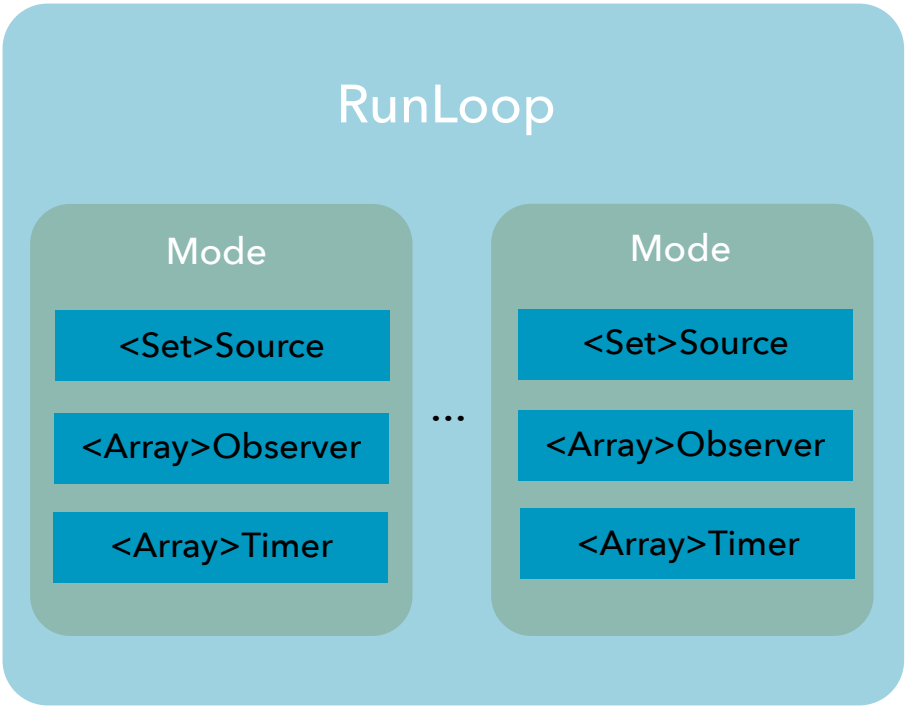
iOS高级经典面试

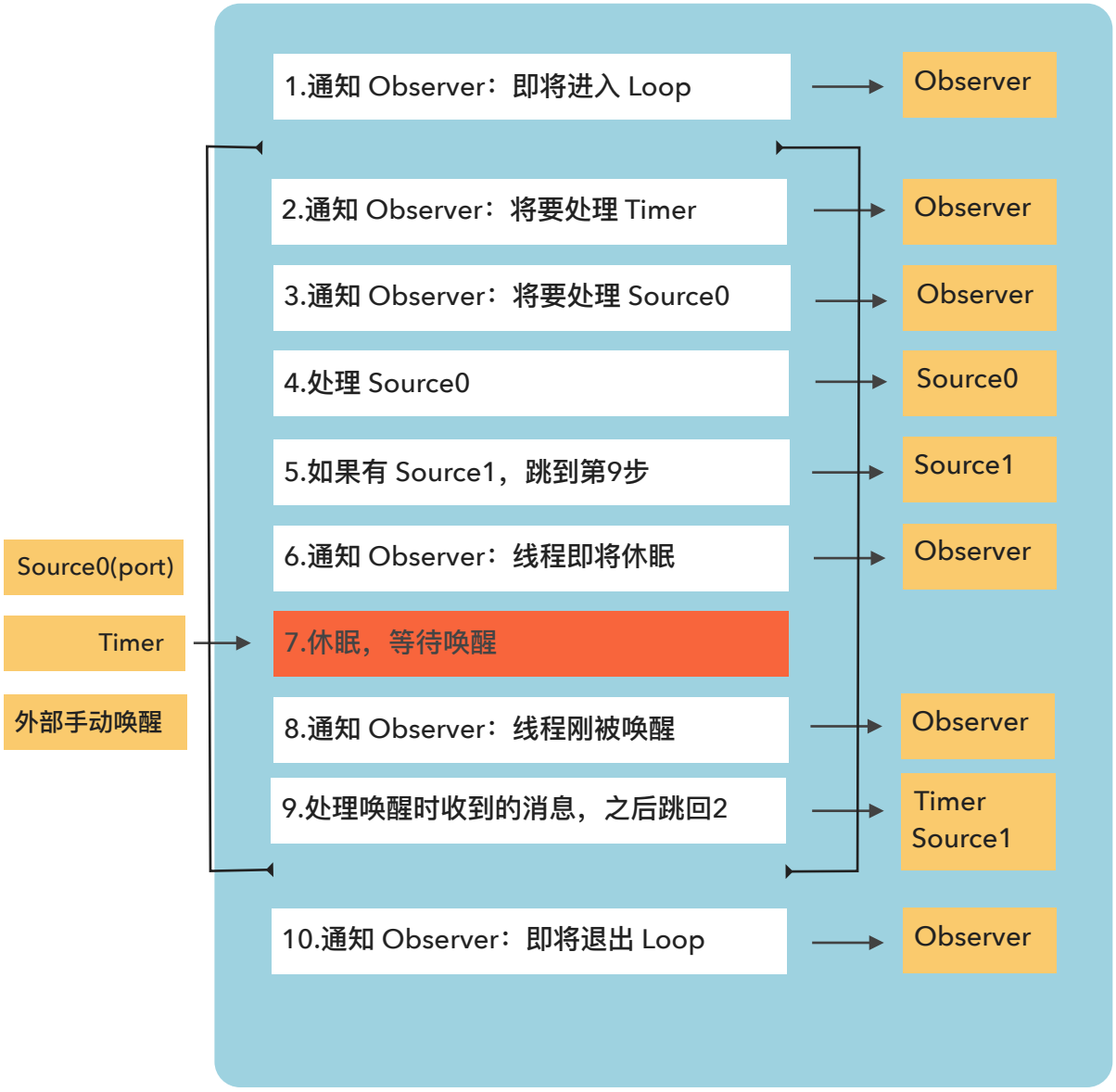
黄小梦 整理
2017年4月21日

1.Runloop和线程的关系

1.一一对应，主线程的runloop已经创建，子线程的必须手动创建
2.runloop在第一次获取时创建，在线程结束时销毁
//在 runloop中有多个运行模式，但是只能选择一种模式运行，mode 中至少要有一个 timer 或者是 source
Mode:
系统默认注册5个 Mode:
kCFRunLoopDefaultMode:App默认 mode，通常主线程在这个 mode 下运行
UITrackingRunLoopMode:界面跟踪 mode，用于 ScrollView 追踪触摸滑动，保证滑动时不受其他 mode 影响
kCFRunLoopCommonModes:占位用的 mode，不是一个真正的 mode
NSRunLoopCommonModes 相当于 NSDefaultRunLoopMode + UITrackingRunLoopMode
UIInitializationRunLoopMode:刚启动 App 时进入的第一个 mode，启动完成之后不再使用
GSEventReceiveRunLoopMode:接受系统事件的内部 mode，通常用不到

runloop 学习链接:
http://blog.csdn.net/hherima/article/details/51746125




1.通知 Observer：即将进入 Loop
Observer
2.通知 Observer：将要处理 Timer
Observer
3.通知 Observer：将要处理 Source0
Observer
4.处理 Source0
Source0
5.如果有 Source1，跳到第9步
Source1
6.通知 Observer：线程即将休眠
Observer
Source0(port)
Timer
外部手动唤醒
7.休眠，等待唤醒
8.通知 Observer：线程刚被唤醒
Observer
9.处理唤醒时收到的消息，之后跳回2
Timer
Source1
10.通知 Observer：即将退出 Loop
Observer

2.自动释放池什么时候释放?

//第一次创建：启动 runloop时候

//最后一次销毁：runloop 退出的时候

//其他时候的创建和销毁：当 runloop 即将睡眠时销毁之前的释放池，重新创建一个新的

3.什么情况下使用 weak 关键字，和 assign 的区别?

1、ARC 中，有可能出现循环引用的地方使用，比如：delegate 属性

2、自定义 IBOutlet 控件属性一般也是使用 weak

区别：weak 表明一种非持有关系，必须用于 OC 对象；assign用于基本数据类型

4.怎么用 copy 关键字?

1、NSString、NSArray、NSDictionary 等等经常使用copy关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary；他们之间可能进行赋值操作，为确保对象中的字符串值不会无意间变动，应该在设置新属性值时拷贝一份。

2、block也使用 copy

5.@property (copy) NSMutableArray *array;这写法会出什么问题?

1、添加,删除,修改数组内的元素的时候,程序会因为找不到对应的方法而崩溃，因为 copy 就是复制一个不可变 NSArray 的对象；

2、使用了 atomic 属性会严重影响性能 ；

6.如何让自己的类用 copy 修饰符? 即让自己写的对象具备拷贝功能

具体步骤：

1、需声明该类遵从 NSCopying 或 NSMutableCopying 协议

2、实现 NSCopying 协议。该协议只有一个方法:

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

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

本质：@property = ivar + getter + setter;（实例变量+getter方法+setter方法）

在编译期自动生成getter、setter，还自动向类中添加适当类型的实例变量，也可以用 `@synthesize` 语法来指定实例变量的名字

8.多线程

1、NSThread线程的生命周期：线程任务执行完毕之后被释放

```
- (void)createChildThread1 {
    //1.创建线程
    NSThread *thread = [[NSThread alloc] initWithTarget:self selector:@selector
        (run:) object:@"abc"];
    //2.启动线程
    [thread start];
}

- (void)createChildThread2 {
    [NSThread detachNewThreadSelector:@selector(run:) toTarget:self
        withObject:@"分离子线程"];
}

- (void)createChildThread3 {
    [self performSelectorInBackground:@selector(run:) withObject:@"开启后台线程"];
}
```

图3-1：NSThread开启子线程的3种方式

2、线程安全（加互斥锁）：

格式：@synchronized (self) { //需要锁定的代码 }

注意：1.锁必须全局唯一；2.加锁的位置；3.加锁的前提条件；4.加锁结果：线程同步

优点：防止多线程抢夺资源造成的数据安全问题

缺点：耗费 CPU 性能

使用前提：多线程使用同一块资源

线程同步：多条线程在同一条线上按顺序的执行任务

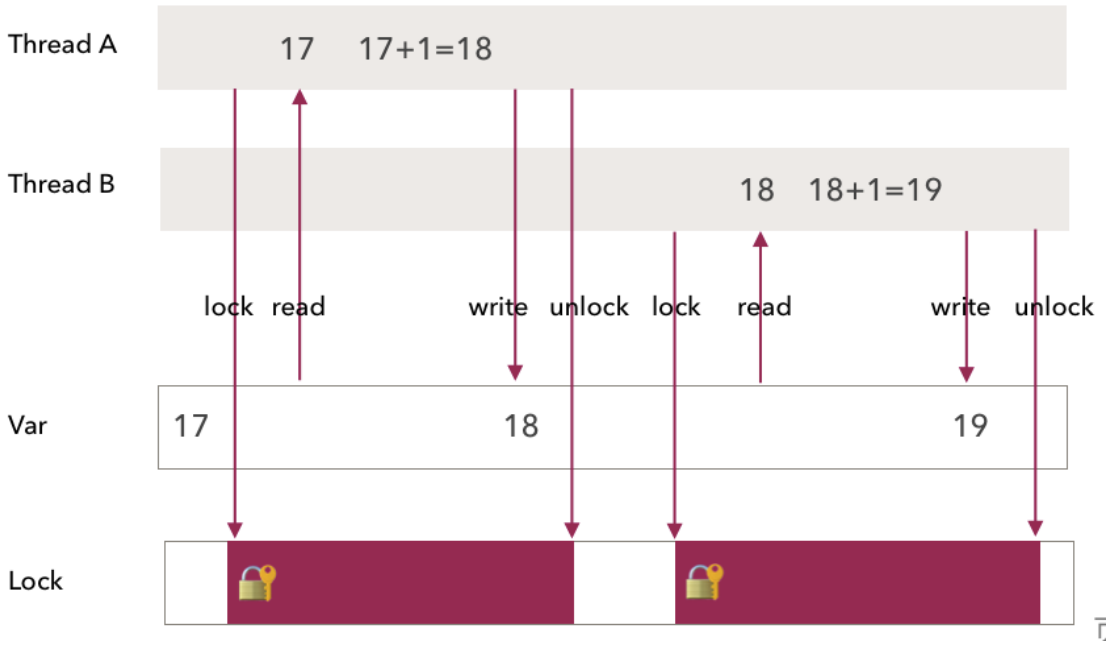


图3-2：互斥锁(官方文档解释)

上图3-2的理解： 线程A(Thread A) 和 线程B(Thread B) 同时访问资源变量 Var，为了防止抢夺资源，Thread A 在读取资源变量 Var 之前先加一把锁，然后读取 Var 的数据并在 Thread A 中完成对数据的操作(17+1=18)，然后把数据写入 Var 中，最后开锁 unlock。在 Thread A 对 Var 操作的过程中，Thread B 是无权访问 Var 的，只有 Thread A unlock 之后，Thread B 才能访问资源变量 Var。

atomic：原子属性，为 setter 方法加锁（默认就是 atomic），线程安全，耗资源

nonatomic：非原子属性，不会为 setter 方法加锁，非线程安全

3、线程间通信

<1>.NSThread 实现

- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(nullable id)arg waitUntilDone:(BOOL)wait;

- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(nullable id)arg waitUntilDone:(BOOL)wait;

<2>.GCD 实现

子线程和主线程相互切换

<3>.NSOperationQueue 实现

4、GCD（任务、队列）

<1>同步和异步的区别

同步：只能在当前线程中执行任务，不能开启新线程

异步：可在新的线程中执行任务，能开启新线程

<2>队列

并发队列：可多个任务并发（同时）执行（会开启多个线程同时执行任务）；只在异步函数（dispatch_async）下有效

串行队列：一个任务执行完毕后，再执行下一个任务

主队列：主队列中的任务都在主线程中执行

```objc
//异步函数+并发队列：会开启多条新线程，队列中的任务是并发执行
- (void)asyncConcurrent {
    //1.创建队列
    dispatch_queue_t queue = dispatch_queue_create("com.iloveu.download",
        DISPATCH_QUEUE_CONCURRENT);
    //2.封装操作---添加任务到队列中
    dispatch_async(queue, ^{
        NSLog(@"download1---%@", [NSThread currentThread]);
    });
    dispatch_async(queue, ^{
        NSLog(@"download2---%@", [NSThread currentThread]);
    });
}

//异步函数+串行队列：会开线程，开一条线程，队列中的任务是串行执行
- (void)asyncSerial {…}

//同步函数+并发队列：不会开线程，任务是串行执行
- (void)syncConcurrent {…}

//同步函数+串行队列：不会开线程，任务是串行执行
- (void)syncSerial {
    dispatch_queue_t queue = dispatch_queue_create("com.iloveu.download",
        DISPATCH_QUEUE_SERIAL);
    dispatch_sync(queue, ^{
        NSLog(@"download1---%@", [NSThread currentThread]);
    });
    dispatch_sync(queue, ^{
        NSLog(@"download2---%@", [NSThread currentThread]);
    });
}
```

```objc
//异步函数+主队列：不会开线程，所有任务都在主线程中执行
- (void)asyncMain {
    dispatch_queue_t queue = dispatch_get_main_queue();
    dispatch_async(queue, ^{
        NSLog(@"download1---%@", [NSThread currentThread]);
    });
    dispatch_async(queue, ^{
        NSLog(@"download2---%@", [NSThread currentThread]);
    });
}

//同步函数+主队列：死锁
- (void)syncMain {
    dispatch_queue_t queue = dispatch_get_main_queue();

    NSLog(@"start---");

    //同步函数：立刻执行，当前任务未执行完，后续任务也不会执行
    dispatch_sync(queue, ^{
        NSLog(@"download1---%@", [NSThread currentThread]);
    });
    dispatch_sync(queue, ^{
        NSLog(@"download2---%@", [NSThread currentThread]);
    });

    NSLog(@"end---");
}
```

主队列特点：如果主队列发现当前主线程有任务在执行，那么主队列会暂停调用队列中的任务，直到主线程空闲为止。

上图：方法-syncMain如果在主线程中调用，则会发生死锁，即102行之后的不会打印；如果在子线程中调用，则不会发生死锁，NSLog 都能打印。

Runloop 的应用：

- NSTimer 在子线程开启一个定时器；控制定时器在特定模式下执行
- imageView 的显示
- performSelector
- 常驻线程（让一个子线程不进入消亡状态，等待其他线程发来消息，处理其他事件）
- 自动释放池

9.@protocol 和 category 中如何使用 @property?

1、在 protocol 中使用 property 只会生成 setter 和 getter 方法声明，使用属性的目的，是希望遵守该协议的对象能实现该属性
2、category 使用 @property 也是只会生成 setter 和 getter 方法声明，如果真的需要给 category 增加属性的实现，需要借助于运行时的两个函数：
`objc_setAssociatedObject`

`objc_getAssociatedObject`

10.@property中有哪些属性关键字?

1、原子性 —— nonatomic 特质

2、读/写权限 —— readwrite(读写)、readonly (只读)

3、内存管理语义 —— assign、strong、weak、unsafe_unretained、copy

4、方法名 — getter=<name> 、setter=<name>

11.weak属性需要在dealloc中置nil么?

不需要，在ARC环境无论是强指针还是弱指针都无需在 dealloc 设置为 nil，ARC 会自动帮我们处理，即便是编译器不帮我们做这些，weak也不需要在 dealloc 中置nil，runtime 内部已经帮我们实现了

12.@synthesize和@dynamic分别有什么作用?

1、@property有两个对应的词，一个是 @synthesize，一个是 @dynamic。如果 @synthesize和 @dynamic都没写，那么默认的就是`@syntheszie var = _var;`
2、@synthesize 的语义是如果你没有手动实现 setter 方法和 getter 方法，那么编译器会自动为你加上这两个方法
3、@dynamic 告诉编译器：属性的 setter 与 getter 方法由用户自己实现，不自动生成。（当然对于 readonly 的属性只需提供 getter 即可）。假如一个属性被声明为 @dynamic var，然后你没有提供 @setter方法和 @getter 方法，编译的时候没问题，但是当程序运行到 `instance.var = someVar`，由于缺 setter 方法会导致程序崩溃；或者当运行到 `someVar = var` 时，由于缺 getter 方法同样会导致崩溃。编译时没问题，运行时才执行相应的方法，这就是所谓的动态绑定。

13.ARC下，不显式指定任何属性关键字时，默认的关键字都有哪些?

1、基本数据类型：atomic、readwrite、assign

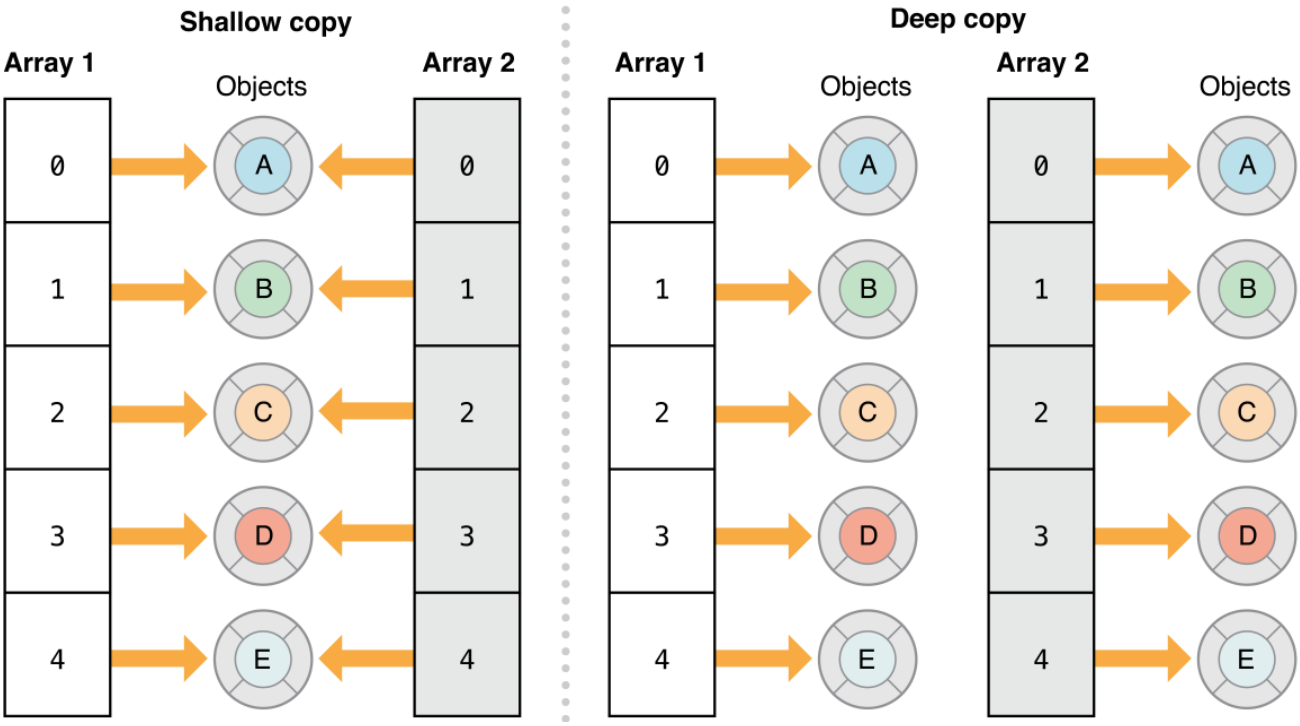
2、普通 OC 对象：atomic、readwrite、strong

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

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

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

浅复制、深复制->浅复制就是指针拷贝、深复制就是内容拷贝，如图：左浅复制、右深复制



不管是集合类对象，还是非集合类对象，接收到copy和mutableCopy消息时，都遵循以下准则：

·copy返回imutable对象；所以，如果对copy返回值使用mutable对象接口就会crash；

·mutableCopy返回mutable对象；

非集合类对象：

在非集合类对象中：对immutable(不可变)对象进行copy操作，是指针复制，mutableCopy操作时内容复制；对mutable(可变)对象进行copy和mutableCopy都是内容复制。简单表示如下：

```
[immutableObject copy] // 浅复制
[immutableObject mutableCopy] //深复制
[mutableObject copy] //深复制
[mutableObject mutableCopy] //深复制
```

在集合类对象中，对immutable对象进行copy，是指针复制，mutableCopy是内容复制；对mutable对象进行copy和mutableCopy都是内容复制。但是：集合对象的内容复制仅限于对象本身，对象元素仍然是指针复制。用代码简单表示如下：

```
[immutableObject copy] // 浅复制
[immutableObject mutableCopy] //单层深复制
[mutableObject copy] //单层深复制
[mutableObject mutableCopy] //单层深复制
```

15.@synthesize合成实例变量的规则是什么？假如property名为foo，存在一个名为_foo的实例变量，那么还会自动合成新变量么？

@synthesize 合成实例变量的规则，有以下几点：

1 如果指定了成员变量的名称,会生成一个指定的名称的成员变量，

2 如果这个成员已经存在了就不再生成了.

3 如果是 @synthesize foo; 还会生成一个名称为foo的成员变量，也就是说：如果没有指定成员变量的名称会自动生成一个属性同名的成员变量，

4 如果是 @synthesize foo = _foo; 就不会生成成员变量了.

假如 property 名为 foo，存在一个名为 _foo 的实例变量，那么还会自动合成新变量么？ 不会。

16.在有了自动合成属性实例变量之后，@synthesize还有哪些使用场景?

1、同时重写了setter和getter时，系统就不会生成ivar，使用@synthesize foo = _foo; 关联 @property 与 ivar

2、重写了只读属性的 getter 时

3、使用了 @dynamic 时

4、在 @protocol 中定义的所有属性

5、在 category 中定义的所有属性

6、重载的属性，当在子类中重载了父类中的属性，必须使用@synthesize来手动合成ivar

17.objc中向一个nil对象发送消息将会发生什么?

在 Objective-C 中向 nil 发送消息是完全有效的—只是在运行时不会有任何作用
如果一个方法返回值是一个对象，那么发送给nil的消息将返回0(nil)，如果向一个nil对象发送消息，首先在寻找对象的isa指针时就是0地址返回了，所以不会出现任何错误。

18.objc中向一个对象发送消息[obj foo]和objc_msgSend()函数之间有什么关系?

[obj foo]; 在objc动态编译时，每个方法在运行时会被动态转为消息发送，即为:
objc_msgSend(obj, @selector(foo));

19.什么时候会报unrecognized selector的异常?

当调用该对象上某个方法，而该对象上没有实现这个方法的时候，可以通过“消息转发”进行解决。
objc在向一个对象发送消息时，runtime库会根据对象的isa指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，如果，在最顶层的父类中依然找不到相应的方法时，程序在运行时会挂掉并抛出异常unrecognized selector sent to XXX。但是在这之前，objc的运行时会给出三次拯救程序崩溃的机会:

1、Method resolution

objc运行时会调用+resolveInstanceMethod:或者 +resolveClassMethod:，让你有机会提供一个函数实现。如果你添加了函数，那运行时系统就会重新启动一次消息发送的过程，否则，运行时就会移到下一步，消息转发（Message Forwarding）。

2、Fast forwarding

如果目标对象实现了-forwardingTargetForSelector:，Runtime 这时就会调用这个方法，给你把这个消息转发给其他对象的机会。只要这个方法返回的不是nil和self，整个消息发送的过程就会被重启，当然发送的对象会变成你返回的那个对象。否则，就会继续Normal Fowarding。这里叫Fast，只是为了区别下一步的转发机制。因为这一步不会创建任何新的对象，但下一步转发会创建一个NSInvocation对象，所以相对更快点。

3、Normal forwarding

这一步是Runtime最后一次给你挽救的机会。首先它会发送-methodSignatureForSelector:消息获得函数的参数和返回值类型。如果-methodSignatureForSelector:返回nil，Runtime则会发出-doesNotRecognizeSelector:消息，程序这时也就挂掉了。如果返回了一个函数签名，Runtime就会创建一个NSInvocation对象并发送-forwardInvocation:消息给目标对象。

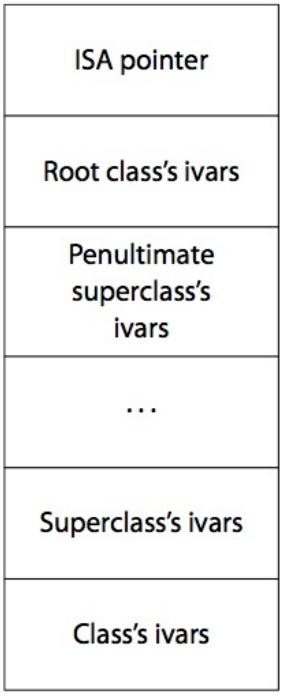
20.一个objc对象如何进行内存布局？（考虑有父类的情况）

- 所有父类的成员变量和自己的成员变量都会存放在该对象所对应的存储空间中.
- 每一个对象内部都有一个isa指针,指向他的类对象,类对象中存放着本对象的:

1 对象方法列表（对象能够接收的消息列表，保存在它所对应的类对象中）
2 成员变量的列表，
3 属性列表，

它内部也有一个isa指针指向元对象(meta class),元对象内部存放的是类方法列表,类对象内部还有一个superclass的指针,指向他的父类对象。

每个 Objective-C 对象都有相同的结构，如下图所示：

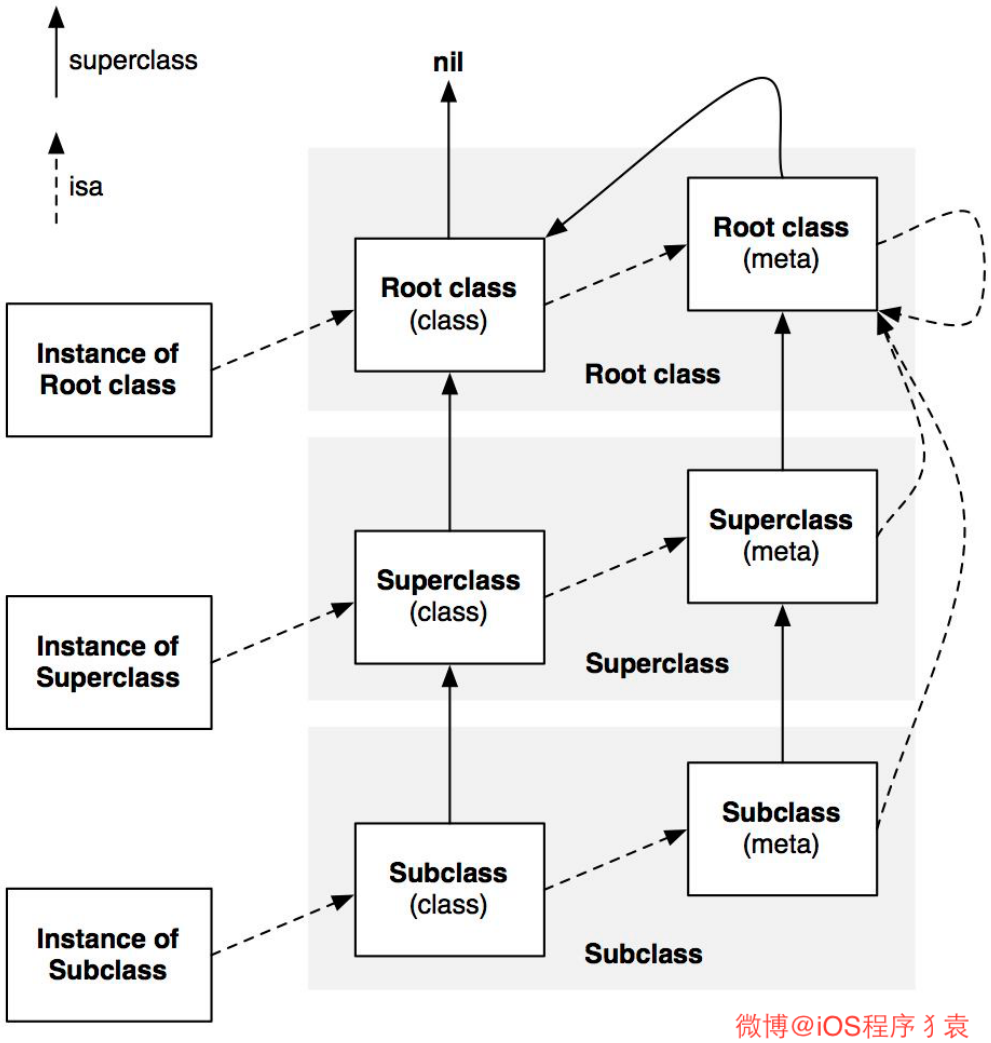


翻译过来就是：

Objective-C 对象的结构图
ISA指针
根类的实例变量
倒数第二层父类的实例变量
...
父类的实例变量
类的实例变量

- 根对象就是NSObject，它的superclass指针指向nil
- 类对象既然称为对象，那它也是一个实例。类对象中也有一个isa指针指向它的元类(meta class)，即类对象是元类的实例。元类内部存放的是类方法列表，根元类的isa指针指向自己，superclass指针指向NSObject类。

如图：



21.一个objc对象的isa指针指向什么？有什么作用？

指向他的类对象，从而可以找到对象上的方法

22.runtime如何通过selector找到对应的IMP地址？（分别考虑类方法和实例方法）

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

23.

21. 下面的代码输出什么?

```
@implementation Son : Father
- (id)init
{
    self = [super init];
    if (self) {
        NSLog(@"%@", NSStringFromClass([self class]));
        NSLog(@"%@", NSStringFromClass([super class]));
    }
    return self;
}
@end
```

答案:

都输出 Son

```
NSStringFromClass([self class]) = Son
NSStringFromClass([super class]) = Son
```

这个题目主要是考察关于 Objective-C 中对 self 和 super 的理解。

我们都知道：self 是类的隐藏参数，指向当前调用方法的这个类的实例。其实 super 是一个 Magic Keyword，它本质是一个编译器标示符，和self是指向的同一个消息接受者！他们两个的不同点在于：super 会告诉编译器，调用class这个方法时，要去父类的方法，而不是本类里的。当使用self调用方法时，会从当前类的方法列表中开始找，如果没有，就从父类中再找；而当使用super时，则从父类的方法列表中开始找，然后调用父类的这个方法。(看不懂，移步原文有详细介绍)

24.使用runtime Associate方法关联的对象，需要在主对象dealloc的时候释放么?

无论在MRC下还是ARC下均不需要。

既然会被销毁，那么具体在什么时间点?

根据 ***WWDC 2011, Session 322 (第36分22秒)*** 中发布的内存销毁时间表，被关联的对象在生命周期内要比对象本身释放的晚很多。它们会在被 NSObject -dealloc 调用的 object_dispose() 方法中释放。

```
// 对象的内存销毁时间表
1. 调用 -release ：引用计数变为零
    * 对象正在被销毁，生命周期即将结束.
    * 不能再有新的 __weak 弱引用， 否则将指向 nil.
    * 调用 [self dealloc]
2. 子类 调用 -dealloc
    * 继承关系中最底层的子类 在调用 -dealloc
    * 如果是 MRC 代码 则会手动释放实例变量们（iVars）
    * 继承关系中每一层的父类 都在调用 -dealloc
3. NSObject 调用 -dealloc
    * 只做一件事：调用 Objective-C runtime 中的 object_dispose() 方法
4. 调用 object_dispose()
    * 为 C++ 的实例变量们（iVars）调用 destructors
    * 为 ARC 状态下的 实例变量们（iVars） 调用 -release
    * 解除所有使用 runtime Associate方法关联的对象
    * 解除所有 __weak 引用
    * 调用 free()
```

25.objc中的类方法和实例方法有什么本质区别和联系?

类方法:

1. 类方法是属于类对象的
2. 类方法只能通过类对象调用
3. 类方法中的self是类对象
4. 类方法可以调用其他的类方法
5. 类方法中不能访问成员变量
6. 类方法中不能直接调用对象方法

实例方法:

1. 实例方法是属于实例对象的
2. 实例方法只能通过实例对象调用
3. 实例方法中的self是实例对象
4. 实例方法中可以访问成员变量
5. 实例方法中直接调用实例方法
6. 实例方法中也可以调用类方法(通过类名)方法

26._objc_msgForward函数是做什么的，直接调用它将会发生什么?

_objc_msgForward是一个函数指针（和 IMP 的类型一样），用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，_objc_msgForward会尝试做消息转发。

objc_msgSend在“消息传递”中的作用。在“消息传递”过程中，objc_msgSend的动作比较清晰：首先在 Class 中的缓存查找 IMP （没缓存则初始化缓存），如果没找到，则向父类的 Class 查找。如果一直查找到根类仍旧没有实现，则用_objc_msgForward函数指针代替 IMP 。最后，执行这个 IMP 。

_objc_msgForward消息转发做的几件事：

1. 调用resolveInstanceMethod:方法 (或 resolveClassMethod:)。允许用户在此时为该 Class 动态添加实现。如果有实现了，则调用并返回YES，那么重新开始objc_msgSend流程。这一次对象会响应这个选择器，一般是因为它已经调用过class_addMethod。如果仍没实现，继续下面的动作。
2. 调用forwardingTargetForSelector:方法，尝试找到一个能响应该消息的对象。如果获取到，则直接把消息转发给它，返回非 nil 对象。否则返回 nil ，继续下面的动作。注意，这里不要返回 self ，否则会形成死循环。
3. 调用methodSignatureForSelector:方法，尝试获得一个方法签名。如果获取不到，则直接调用doesNotRecognizeSelector抛出异常。如果能获取，则返回非nil：创建一个 NSInvocation 并传给forwardInvocation:。
4. 调用forwardInvocation:方法，将第3步获取到的方法签名包装成 Invocation 传入，如何处理就在这里面了，并返回非ni。
5. 调用doesNotRecognizeSelector: ，默认的实现是抛出异常。如果第3步没能获得一个方法签名，执行该步骤。

上面前4个方法均是模板方法，开发者可以override，由 runtime 来调用。最常见的实现消息转发：就是重写方法3和4，吞掉一个消息或者代理给其他对象都是没问题的
也就是说_objc_msgForward在进行消息转发的过程中会涉及以下这几个方法：

1. resolveInstanceMethod:方法 (或 resolveClassMethod:)。
2. forwardingTargetForSelector:方法
3. methodSignatureForSelector:方法
4. forwardInvocation:方法
5. doesNotRecognizeSelector: 方法

一旦调用_objc_msgForward，将跳过查找 IMP 的过程，直接触发“消息转发”，
如果调用了_objc_msgForward，即使这个对象确实已经实现了这个方法，你也会告诉objc_msgSend： “我没有在这个对象里找到这个方法的实现”，

如果用不好会直接导致程序Crash，但是如果用得好，做很多事情，比如JSPatch、RAC(ReactiveCocoa)

27.runtime如何实现weak变量的自动置nil?

runtime 对注册的类， 会进行布局，对于 weak 对象会放入一个 hash 表中。 用 weak 指向的对象内存地址作为 key，当此对象的引用计数为0的时候会 dealloc，假如 weak 指向的对象内存地址是a，那么就会以a为键， 在这个 weak 表中搜索，找到所有以a为键的 weak 对象，从而设置为 nil。

我们可以设计一个函数（伪代码）来表示上述机制：
objc_storeWeak(&a, b)函数：
objc_storeWeak函数把第二个参数--赋值对象（b）的内存地址作为键值key，将第一个参数--weak修饰的属性变量（a）的内存地址（&a）作为value，注册到 weak 表中。如果第二个参数（b）为0（nil），那么把变量（a）的内存地址（&a）从weak表中删除，
你可以把objc_storeWeak(&a, b)理解为：objc_storeWeak(value, key)，并且当key变nil，将value置nil。

weak 修饰的指针默认值是 nil （在Objective-C中向nil发送消息是安全的）

在b非nil时，a和b指向同一个内存地址，在b变nil时，a变nil。此时向a发送消息不会崩溃：在Objective-C中向nil发送消息是安全的。
而如果a是由assign修饰的，则： 在b非nil时，a和b指向同一个内存地址，在b变nil时，a还是指向该内存地址，变野指针。此时向a发送消息极易崩溃。

28.能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么?

- 不能向编译后得到的类中增加实例变量；
- 能向运行时创建的类中添加实例变量；

解释下：

- 因为编译后的类已经注册在 runtime 中，类结构体中的 objc_ivar_list 实例变量的链表和 instance_size 实例变量的内存大小已经确定，同时runtime 会调用 class_setIvarLayout 或 class_setWeakIvarLayout 来处理 strong weak 引用。所以不能向存在的类中添加实例变量；
- 运行时创建的类是可以添加实例变量，调用 class_addIvar 函数。但是得在调用 objc_allocateClassPair 之后，objc_registerClassPair 之前，原因同上。

29.runloop和线程有什么关系?

实际上，run loop和线程是紧密相连的，可以这样说run loop是为了线程而生，没有线程，它就没有存在的必要。Run loops是线程的基础架构部分， Cocoa 和 CoreFundation 都提供了 run loop 对象方便配置和管理线程的 run loop （以下都以 Cocoa 为例）。每个线程，包括程序的主线程（ main thread ）都有与之相应的 run loop 对象。

runloop 和线程的关系：

1 主线程的run loop默认是启动的。

iOS的应用程序里面，程序启动后会有一个如下的main()函数

```
int main(int argc, char * argv[]) {
   @autoreleasepool {
       return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
class]));
   }
}
```

重点是UIApplicationMain()函数，这个方法会为main thread设置一个NSRunLoop对象，这就解释了：为什么我们的应用可以在无人操作的时候休息，需要让它干活的时候又能立马响应。

2 对其它线程来说，run loop默认是没有启动的，如果你需要更多的线程交互则可以手动配置和启动，如果线程只是去执行一个长时间的已确定的任务则不需要。

3 在任何一个 Cocoa 程序的线程中，都可以通过以下代码来获取到当前线程的 run loop 。

```
NSRunLoop *runloop = [NSRunLoop currentRunLoop];
```

30.runloop的mode作用是什么?

model 主要是用来指定事件在运行循环中的优先级的，分为：

- NSDefaultRunLoopMode（kCFRunLoopDefaultMode）：默认，空闲状态
- UITrackingRunLoopMode：ScrollView滑动时
- UIInitializationRunLoopMode：启动时
- NSRunLoopCommonModes（kCFRunLoopCommonModes）：Mode集合

苹果公开提供的 Mode 有两个：

1 NSDefaultRunLoopMode（kCFRunLoopDefaultMode）

2 NSRunLoopCommonModes（kCFRunLoopCommonModes）

31.以+ scheduledTimerWithTimeInterval...的方式触发的timer，在滑动页面上的列表时，timer会暂定回调，为什么？如何解决?

RunLoop只能运行在一种mode下，如果要换mode，当前的loop也需要停下重启成新的。利用这个机制，ScrollView滚动过程中NSDefaultRunLoopMode（kCFRunLoopDefaultMode）的mode会切换到UITrackingRunLoopMode来保证ScrollView的流畅滑动：只有在NSDefaultRunLoopMode模式下处理的事件会影响ScrollView的滑动。

如果我们把一个NSTimer对象以NSDefaultRunLoopMode（kCFRunLoopDefaultMode）添加到主运行循环中的时候，ScrollView滚动过程中会因为mode的切换，而导致NSTimer将不再被调度。

同时因为mode还是可定制的，所以：
Timer计时会被scrollView的滑动影响的问题可以通过将timer添加到NSRunLoopCommonModes（kCFRunLoopCommonModes）来解决。

32.猜想runloop内部是如何实现的?

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  function loop() {
3      initialize();
4      do {
5          var msg = get_next_msg();
6          process_msg(msg);
7      } while (msg != nil);
8  }
```

或使用伪代码来展示下:

```
int main(int argc, char * argv[]) {
    //程序一直运行状态
    while (AppIsRunning) {
        //睡眠状态，等待唤醒事件
        id whoWakesMe = SleepForWakingUp();
        //得到唤醒事件
        id event = GetEvent(whoWakesMe);
        //开始处理事件
        HandleEvent(event);
    }
    return 0;
}
```

33.objc使用什么机制管理对象内存?

通过 retainCount 的机制来决定对象是否需要释放。 每次 runloop 的时候，都会检查对象的retainCount，如果retainCount 为 0，说明该对象没有地方需要继续使用了，可以释放掉了。

34.ARC通过什么方式帮助开发者管理内存?

~~编译时根据代码上下文，插入 retain/release~~
ARC相对于MRC，不是在编译时添加retain/release/autorelease这么简单。应该是编译期和运行期两部分共同帮助开发者管理内存。
在编译期，ARC用的是更底层的C接口实现的retain/release/autorelease，这样做性能更好，也是为什么不能在ARC环境下手动retain/release/autorelease，同时对同一上下文的同一对象的成对retain/release操作进行优化（即忽略掉不必要的操作）；ARC也包含运行期组件，这个地方做的优化比较复杂，但也不能被忽略。【TODO:后续更新会详细描述下】

35.BAD_ACCESS在什么情况下出现?

访问了野指针，比如对一个已经释放的对象执行了release、访问已经释放对象的成员变量或者发消息。 死循环

36.使用block时什么情况会发生引用循环，如何解决?

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

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

或者将其中一方强制置空 `xxx = nil`。

37.在block内如何修改block外部变量?

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。block 内部的变量会被 copy 到堆区，进而在block内部也可以修改外部变量的值。
block也属于“函数”的范畴，变量进入block，实际就是已经改变了作用域。

38.苹果是如何实现autoreleasepool的?

autoreleasepool 以一个队列数组的形式实现,主要通过下列三个函数完成.

```
1   objc_autoreleasepoolPush
2   objc_autoreleasepoolPop
3   objc_autorelease
```

看函数名就可以知道，对 autorelease 分别执行 push，和 pop 操作。销毁对象时执行release操作。

举例说明：我们都知道用类方法创建的对象都是 Autorelease 的，那么一旦 Person 出了作用域，当在 Person 的 dealloc 方法中打上断点，我们就可以看到这样的调用堆栈信息：

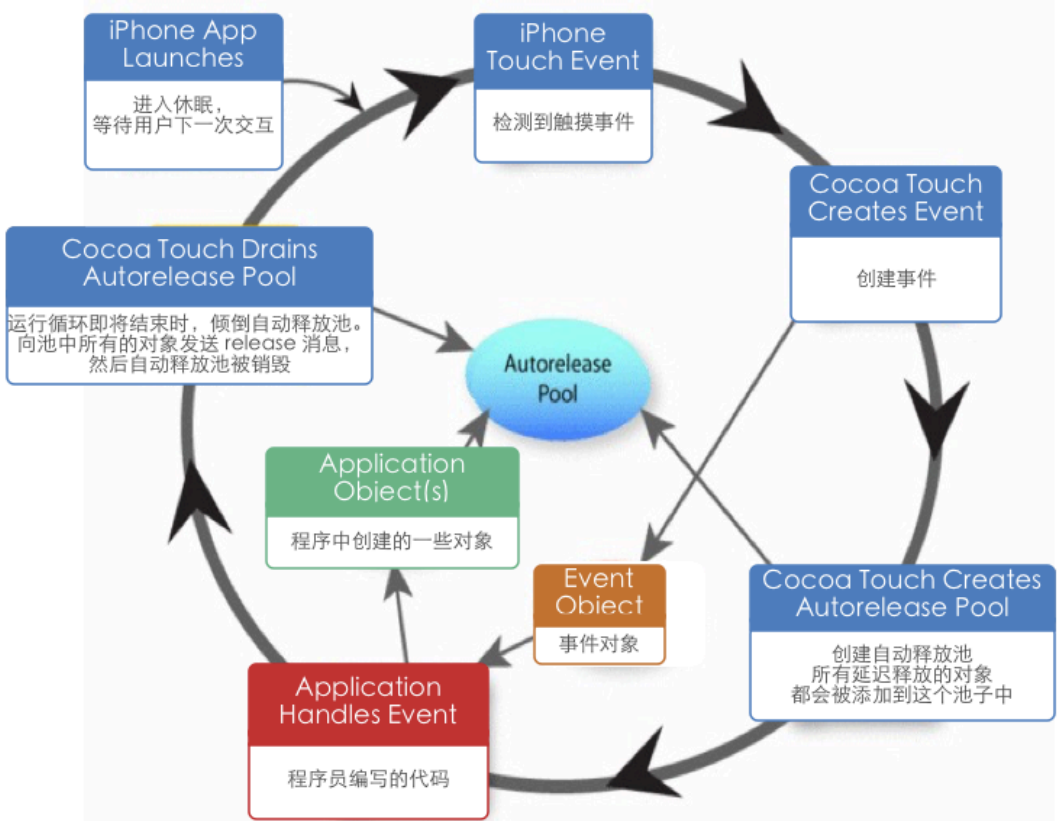
39.不手动指定autoreleasepool的前提下，一个autorealese对象在什么时刻释放？（比如在一个vc的viewDidLoad中创建）

分两种情况：手动干预释放时机、系统自动去释放。

1. 手动干预释放时机--指定autoreleasepool 就是所谓的：当前作用域大括号结束时释放。
2. 系统自动去释放--不手动指定autoreleasepool

Autorelease对象出了作用域之后，会被添加到最近一次创建的自动释放池中，并会在当前的runloop 迭代结束时释放。

释放的时机总结起来，可以用下图来表示：



下面对这张图进行详细的解释：
从程序启动到加载完成是一个完整的运行循环，然后会停下来，等待用户交互，用户的每一次交互都会启动一次运行循环，来处理用户所有的点击事件、触摸事件。

所有autorelease的对象，在出了作用域之后，会被自动添加到最近创建的自动释放池中。

但是如果每次都放进应用程序的 `main.m` 中的 autoreleasepool 中，迟早有被撑满的一刻。这个过程中必定有一个释放的动作。何时？在一次完整的运行循环结束之前，会被销毁。
那什么时间会创建自动释放池？运行循环检测到事件并启动后，就会创建自动释放池。

子线程的 runloop 默认是不工作，无法主动创建，必须手动创建。

自定义的 NSOperation 和 NSThread 需要手动创建自动释放池。比如： 自定义的 NSOperation 类中的 main 方法里就必须添加自动释放池。否则出了作用域后，自动释放对象会因为没有自动释放池去处理它，而造成内存泄露。

但对于 blockOperation 和 invocationOperation 这种默认的Operation ，系统已经帮我们封装好了，不需要手动创建自动释放池。

@autoreleasepool 当自动释放池被销毁或者耗尽时，会向自动释放池中的所有对象发送 release 消息，释放自动释放池中的所有对象。

如果在一个vc的viewDidLoad中创建一个 Autorelease对象，那么该对象会在 viewDidAppear 方法执行前就被销毁了。

40.使用系统的某些block api（如UIView的block版本写动画时），是否也考虑引用循环问题?

UIView的block版本写动画时不需要考虑，所谓“引用循环”是指双向的强引用，所以那些“单向的强引用”（block 强引用 self ）没有问题，比如这些不用考虑：

```
[UIView animateWithDuration:duration animations:^{ [self.superview
layoutIfNeeded]; }];
[[NSOperationQueue mainQueue] addOperationWithBlock:^{ self.someProperty =
xyz; }];
[[NSNotificationCenter defaultCenter] addObserverForName:@"someNotification"
                                                  object:nil
                                                   queue:[NSOperationQueue mainQueue]
                                              usingBlock:^(NSNotification *
notification) {
                                                  self.someProperty = xyz; }];
```

如果你使用一些参数中可能含有 ivar 的系统 api ，如 GCD 、NSNotificationCenter就要小心一点：比如GCD 内部如果引用了 self，而且 GCD 的其他参数是 ivar，则要考虑到循环引用：

```
__weak __typeof__(self) weakSelf = self;
dispatch_group_async(_operationsGroup, _operationsQueue, ^
{
__typeof__(self) strongSelf = weakSelf;
[strongSelf doSomething];
[strongSelf doSomethingElse];
} );
```

类似的：

```
 __weak __typeof__(self) weakSelf = self;
 _observer = [[NSNotificationCenter defaultCenter] addObserverForName:@"testKey"
                                                                object:nil
                                                                 queue:nil
usingBlock:^(NSNotification *note) {
     __typeof__(self) strongSelf = weakSelf;
     [strongSelf dismissModalViewControllerAnimated:YES];
 }];
```

self --> _observer --> block --> self 显然这也是一个循环引用。

41.GCD的队列（dispatch_queue_t）分哪两种类型?

1 串行队列Serial Dispatch Queue
2 并行队列Concurrent Dispatch Queue

42.如何用GCD同步若干个异步调用？（如根据若干个url异步加载多张图片，然后在都下载完成后合成一张整图）

使用Dispatch Group追加block到Global Group Queue,这些block如果全部执行完毕，就会执行Main Dispatch Queue中的结束处理的block。

```
dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t group = dispatch_group_create();
dispatch_group_async(group, queue, ^{ /*加载图片1 */ });
dispatch_group_async(group, queue, ^{ /*加载图片2 */ });
dispatch_group_async(group, queue, ^{ /*加载图片3 */ });
dispatch_group_notify(group, dispatch_get_main_queue(), ^{
        // 合并图片
});
```

实际代码如下图：

```
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    dispatch_group_t group = dispatch_group_create();
    //下图片1,放线程组中
    dispatch_group_async(group, queue, ^{
        NSURL *url = [NSURL URLWithString:@""];
        NSData *imgData1 = [NSData dataWithContentsOfURL:url];
        self.img1 = [UIImage imageWithData:imgData1];
    });

    //下载图片2
    dispatch_group_async(group, queue, ^{
        NSURL *url = [NSURL URLWithString:@""];
        NSData *imgData2 = [NSData dataWithContentsOfURL:url];
        self.img2 = [UIImage imageWithData:imgData2];
    });

    //合并图片
    dispatch_group_notify(group, queue, ^{
        NSLog(@"combie---%@", [NSThread currentThread]);
        UIGraphicsBeginImageContext(CGSizeMake(200, 200));
        [self.img1 drawInRect:CGRectMake(0, 0, 200, 100)];
        self.img1 = nil;
        [self.img2 drawInRect:CGRectMake(0, 100, 200, 100)];
        self.img2 = nil;
        UIImage *img = UIGraphicsGetImageFromCurrentImageContext();
        UIGraphicsEndImageContext();

        dispatch_async(dispatch_get_main_queue(), ^{
            NSLog(@"UI---%@", [NSThread currentThread]);
            //self.imgView.image = img;
        });
    });
```

43.dispatch_barrier_async的作用是什么?

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

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

44.苹果为什么要废弃dispatch_get_current_queue?

dispatch_get_current_queue容易造成死锁

45.以下代码运行结果如何?

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    NSLog(@"1");
    dispatch_sync(dispatch_get_main_queue(), ^{
        NSLog(@"2");
    });
    NSLog(@"3");
}
```

只输出：1 。发生主线程锁死。

46.addObserver:forKeyPath:options:context:各个参数的作用分别是什么，observer中需要实现哪个方法才能获得KVO回调?

```
// 添加键值观察
/*
1 观察者，负责处理监听事件的对象
2 观察的属性
3 观察的选项
4 上下文
*/
[self.person addObserver:self forKeyPath:@"name"
options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
context:@"Person Name"];
```

observer中需要实现一下方法:

```
// 所有的 kvo 监听到事件，都会调用此方法
/*
 1. 观察的属性
 2. 观察的对象
 3. change 属性变化字典（新 / 旧）
 4. 上下文，与监听的时候传递的一致
 */
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
change:(NSDictionary *)change context:(void *)context;
```

48.如何手动触发一个value的KVO?

所谓的“手动触发”是区别于“自动触发”：
自动触发是指类似这种场景：在注册 KVO 之前设置一个初始值，注册之后，设置一个不一样的值，就可以触发了。

自动触发 KVO 的原理：
键值观察通知依赖于 NSObject 的两个方法：willChangeValueForKey: 和 didChangevlueForKey: 。在一个被观察属性发生改变之前， willChangeValueForKey: 一定会被调用，这就 会记录旧的值。而当改变发生后，observeValueForKey:ofObject:change:context: 会被调用，继而 didChangeValueForKey: 也会被调用。如果可以手动实现这些调用，就可以实现“手动触发”了。

“手动触发”的使用场景是什么？一般我们只在希望能控制“回调的调用时机”时才会这么做。
而“回调的调用时机”就是在你调用 didChangeValueForKey: 方法时。
具体做法如下：
如果这个 value 是表示时间的 self.now ，那么代码如下：最后两行代码缺一不可。

```
//@property (nonatomic, strong) NSDate *now;
- (void)viewDidLoad {
    [super viewDidLoad];
    _now = [NSDate date];
    [self addObserver:self forKeyPath:@"now"
options:NSKeyValueObservingOptionNew context:nil];
    NSLog(@"1");
    [self willChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。
    NSLog(@"2");
    [self didChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。
    NSLog(@"4");
}
```

但是平时我们一般不会这么干，我们都是等系统去“自动触发”。“自动触发”的实现原理：
比如调用 setNow: 时，系统还会以某种方式在中间插入 wilChangeValueForKey: 、didChangeValueForKey: 和 observeValueForKeyPath:ofObject:change:context: 的调用。

49.若一个类有实例变量 NSString *_foo ，调用setValue:forKey:时，可以以foo还是 _foo 作为key?

都可以。

50.KVC的keyPath中的集合运算符如何使用?

1 必须用在集合对象上或普通对象的集合属性上
2 简单集合运算符有@avg， @count ， @max ， @min ，@sum，
3 格式 @"@sum.age"或 @"集合属性.@max.age"

51.KVC和KVO的keyPath一定是属性么?

KVC 支持实例变量；
如果将一个对象设定成属性，这个属性是自动支持KVO的；如果这个对象是一个实例变量，那么，这个KVO是需要我们自己来实现的。手动支持

手动设定实例变量的KVO实现监听，如下图：

```
// 手动设定KVO
- (void)setAge:(NSString *)age
{
    [self willChangeValueForKey:@"age"];
    _age = age;
    [self didChangeValueForKey:@"age"];
}
- (NSString *)age
{
    return _age;
}
+ (BOOL)automaticallyNotifiesObserversForKey:
    (NSString *)key
{
    // 如果监测到键值为age,则指定为非自动监听对象
    if ([key isEqualToString:@"age"])
    {
        return NO;
    }

    return [super
        automaticallyNotifiesObserversForKey:key]
        ;
}
@end
```

```
#import <Foundation/Foundation.h>

@interface Student : NSObject

{
    NSString  *_age;
}
- (void)setAge:(NSString *)age;
- (NSString *)age;

@property (nonatomic, strong) NSStrin

@end
```

52.如何关闭默认的KVO的默认实现，并进入自定义的KVO实现?

利用 Runtime 动态创建类、实现 KVO

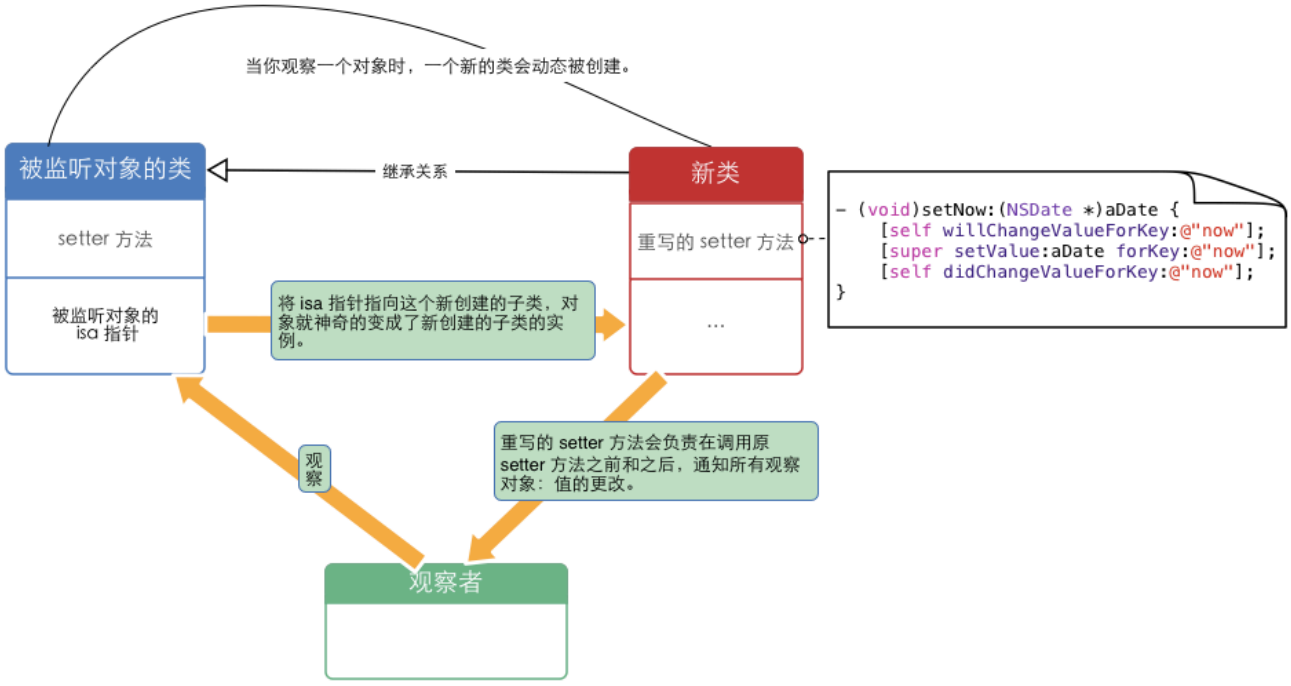
链接：http://tech.glowing.com/cn/implement-kvo/

53.apple用什么方式实现对一个对象的KVO?

KVO 实现：

在使用KVC命名约定时，当你观察一个对象时，一个新的类会被动态创建。这个类继承自该对象的原本的类，并重写了被观察属性的 setter 方法。重写的 setter 方法会负责在调用原 setter 方法之前和之后，通知所有观察对象：值的更改。最后通过 isa 混写（isa-swizzling） 把这个对象的 isa 指针 (isa 指针告诉 Runtime 系统这个对象的类是什么) 指向这个新创建的子类，对象就神奇的变成了新创建的子类的实例。

实现原理如下图：



Apple 使用了 isa 混写（isa-swizzling）来实现 KVO 。

KVO在调用存取方法之前总是调用 willChangeValueForKey: ，之后总是调用 didChangeValueForkey: 。怎么做到的呢?答案是通过 isa 混写（isa-swizzling）。第一次对一个对象调用 addObserver:forKeyPath:options:context: 时，框架会创建这个类的新的 KVO 子类，并将被观察对象转换为新子类的对象。在这个 KVO 特殊子类中，Cocoa 创建观察属性的 setter ，大致工作原理如下：

```
- (void)setNow:(NSDate *)aDate {
```

```
    [self willChangeValueForKey:@"now"];
    [super setValue:aDate forKey:@"now"];
    [self didChangeValueForKey:@"now"];
}
```

54.IBOutlet连出来的视图属性为什么可以被设置成weak?

因为有外链那么视图在xib或者storyboard中肯定存在，视图已经对它有一个强引用了。不过这个回答漏了个重要知识，使用storyboard（xib不行）创建的vc，会有一个叫_topLevelObjectsToKeepAliveFromStoryboard的私有数组强引用所有top level的对象，所以这时即便outlet声明成weak也没关系。

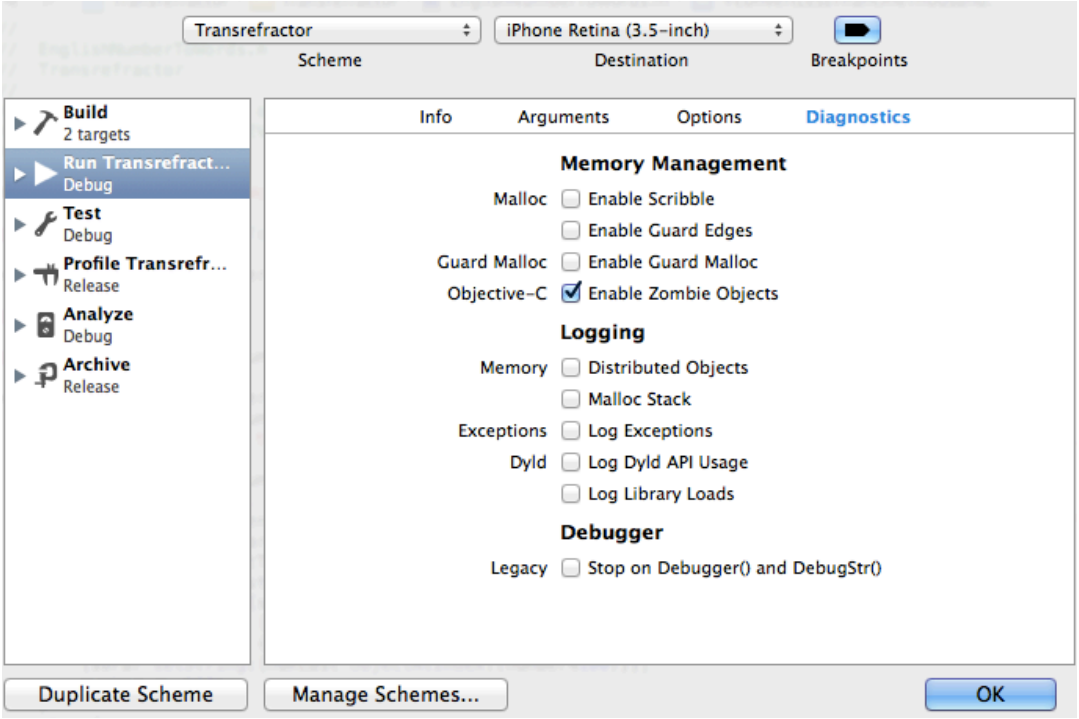
55.IB中User Defined Runtime Attributes如何使用？（应该知道在哪儿吧）

它能够通过KVC的方式配置一些你在interface builder 中不能配置的属性。当你希望在IB中作尽可能多得事情，这个特性能够帮助你编写更加轻量级的viewcontroller。

56.如何调试BAD_ACCESS错误

1、重写object的respondsToSelector方法，现实出现EXEC_BAD_ACCESS前访问的最后一个object

2、通过 Zombie



3、设置全局断点快速定位问题代码所在行

4、Xcode 7 已经集成了BAD_ACCESS捕获功能：**Address Sanitizer**。 用法如下：在Build Settings 中勾选✅Enable Address Sanitizer

57.lldb（gdb）常用的调试命令?

- breakpoint 设置断点定位到某一个函数
- n 断点指针下一步
- po打印对象

----------华丽而优美的分割线----------

58.iOS 容易引起“循环引用”的几种场景

最简单的理解：对象 A 持有对象 B，对象 B 又持有对象 A，就会造成循环引用

1、parent-child相互持有、委托模式

2、block：

隐式（间接）循环引用。ObjectA 持有ObjectB，ObjectB持有block。那么在block中调用ObjectA的self会造成间接循环引用；

即使在你的block代码中没有显式地出现”self”，也会出现循环引用！只要你在block里用到了self所拥有的东西！

显式循环引用，编译器会报警告，在block引用self的时候最好使用weak-strong dance技术。

3、NSTimer：

NSTimer会持有对象，所以：在删除对象之前，需要[timer invalidate]。

```
1   @interface FtKeepAlive : NSObject
2   {
3       NSTimer*              _keepAliveTimer; // 发送心跳timer
4   }
5   //实现文件
6
    _keepAliveTimer = [NSTimer scheduledTimerWithTimeInterval:_expired target:
    self selector:@selector(keepLiveStart) userInfo:nil repeats:YES];
```

如上代码，类持有_keepAliveTimer，_keepAliveTimer又持有self，造成循环引用。

4、比如把self加入array中。也会造成循环引用

5、使用类别添加属性

有一个类A，给A动态添加属性p。如果p中再引用类A，容易造成循环引用

59.类方法load和initialize的区别

iOS会在运行期提前并且自动调用这两个方法。

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

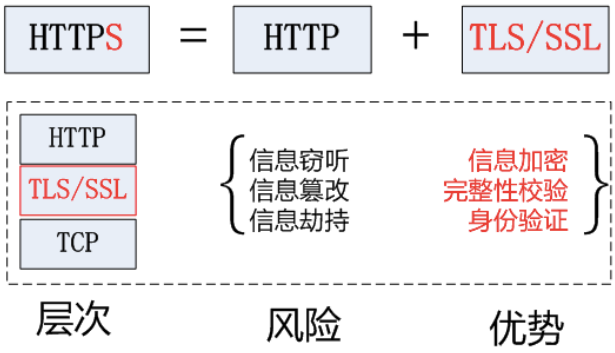
相同点在于：方法只会被调用一次。（其实这是相对runtime来说的）。

方法调用的顺序：父类(Superclass)的方法优先于子类(Subclass)的方法，类中的方法优先于类别(Category)中的方法。

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

当类对象被引入项目时，runtime 会向每一个类对象发送 load 消息. load 方法还是非常的神奇的，因为它会在每一个类甚至分类被引入时仅调用一次，调用的顺序是父类优先于子类，子类优先于分类. 而且 load 方法不会被类自动继承，每一个类中的 load 方法都不需要像 viewDidLoad 方法一样调用父类的方法。

60.HTTP 和 HTTPS



HTTP是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准(TCP)，用于从WWW服务器传输超文本到本地浏览器的传输协议。HTTP是采用明文形式进行数据传输，极易被不法份子窃取和篡改。

HTTPS是在HTTP上建立SSL加密层，并对传输数据进行加密，是HTTP协议的安全版。HTTPS主要作用是：

（1）对数据进行加密，并建立一个信息安全通道，来保证传输过程中的数据安全；

（2）对网站服务器进行真实身份认证。

区别：

1、HTTPS是加密传输协议，HTTP是名文传输协议；

2、HTTPS需要用到SSL证书，而HTTP不用；

3、HTTPS比HTTP更加安全，对**搜索引擎**更友好

4、HTTPS标准端口443，HTTP标准端口80；

5、HTTPS基于传输层，HTTP基于应用层；

6、HTTPS在浏览器显示绿色安全锁，HTTP没有显示；

如下图 HTTP 和 HTTPS 的网络分层：

61.常见的Exception Type

1> EXC_BAD_ACCESS
此类型的Excpetion是我们最长碰到的Crash，通常用于访问了不改访问的内存导致。一般EXC_BAD_ACCESS后面的"()"还会带有补充信息。
SIGSEGV: 通常由于重复释放对象导致，这种类型在切换了ARC以后应该已经很少见到了
SIGABRT: 收到Abort信号退出，通常Foundation库中的容器为了保护状态正常会做一些检测，例如插入nil到数组中等会遇到此类错误
SEGV: (Segmentation Violation)，代表无效内存地址，比如空指针，未初始化指针，栈溢出等
SIGBUS: 总线错误，与 SIGSEGV 不同的是，SIGSEGV 访问的是无效地址，而 SIGBUS 访问的是有效地址，但总线访问异常(如地址对齐问题)
SIGILL: 尝试执行非法的指令，可能不被识别或者没有权限

2> EXC_BAD_INSTRUCTION
此类异常通常由于线程执行非法指令导致

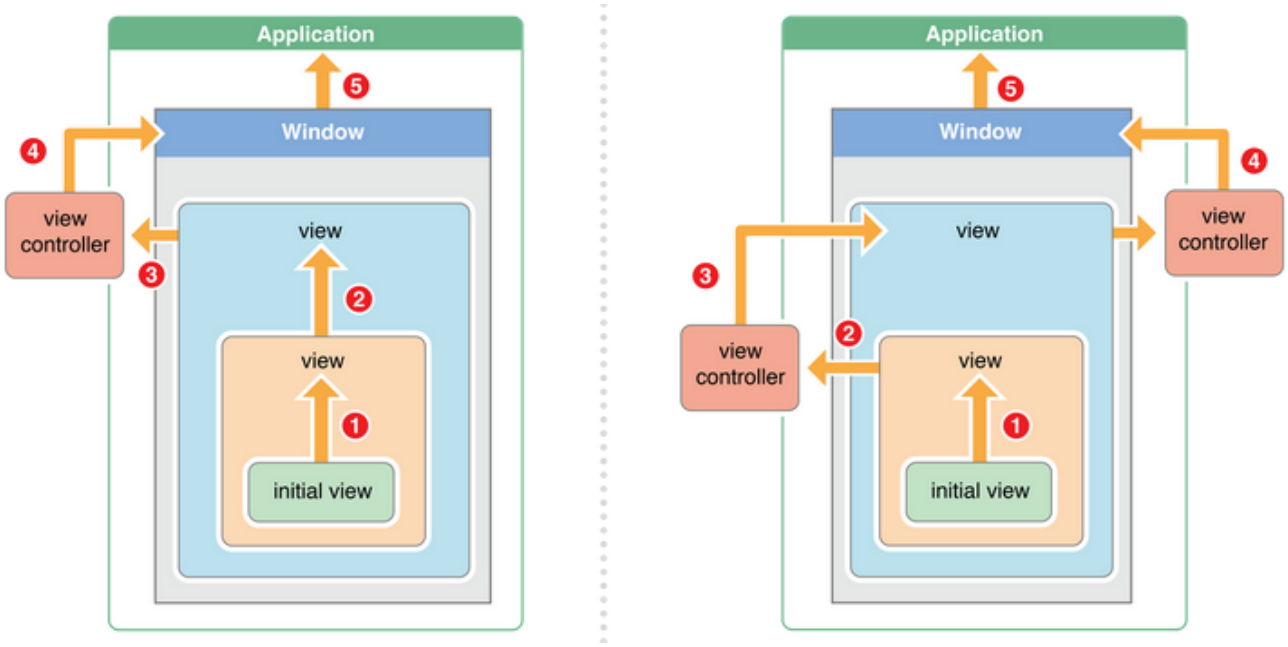
3> EXC_ARITHMETIC
除零错误会抛出此类异常

处理此类异常方式：符号化Crash日志。比较常用的有Crashlytics、Flurry、友盟等。

62.Category 和 Extension

Category的方法不一定非要在@implementation中实现，也可以在其他位置实现，但是当调用Category的方法时，依据继承树没有找到该方法的实现，程序则会崩溃。
Category理论上不能添加变量，但是可以使用 @dynamic 来弥补这种不足。
Category为原始类添加方法，必须要小心不要去重写已经存在的方法
Extension中的方法必须在@implementation中实现，否则编译会报错。
Extension管理类的私有方法

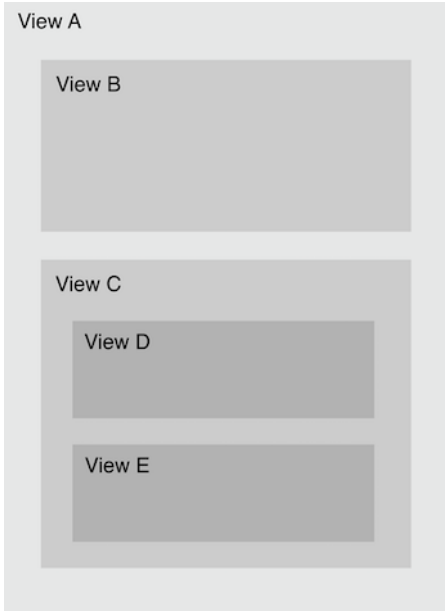
63.响应者链（responder chain）



如上图，响应者链有以下特点：

- 响应者链通常是由 initial view 开始；
- UIView 的 nextResponder 它的 superview；如果 UIView 已经是其所在的 UIViewController 的 top view，那么 UIView 的 nextResponder 就是 UIViewController；
- UIViewController 如果有 Super ViewController，那么它的 nextResponder 为其 Super ViewController 最表层的 View；如果没有，那么它的 nextResponder 就是 UIWindow；
- UIWindow 的 contentView 指向 UIApplication，将其作为 nextResponder；
- UIApplication 是一个响应者链的终点，它的 nextResponder 指向nil，整个 responder chain 结束。

Hit-Test View 与 Hit-Testing



假设用户触摸了上图的 View E 区域，那么 iOS 将会按下面的顺序反复检测 subview 来寻找 Hit-Test View

1. 触摸区域在视图 A 内，所以检测视图 A 的 subview B 和 C；
2. 触摸区域不在视图 B 内，但是在视图 C 内，所以检查视图 C 的 subview D 和 E；
3. 触摸区域不在视图 D 内，在视图 E 中；

视图 E 在整个视图体系中是 lowest view，所以视图 E 就是 Hit-Test View 。

事件的链有两条：事件的响应链；Hit-Testing 时事件的传递链。

- 响应链：由离用户最近的view向系统传递。 initial view -> super view -> …..-> view controller -> window -> Application -> AppDelegate
- Hit-Testing 链：由系统向离用户最近的view传递。 UIKit -> active app's event queue -> window -> root view ->……->lowest view

64.UITableView 的优化

1 重用 cell
2 缓存行高
   - 若 cell 定高，删除代理中的方法 tableView:heightForRowAtIndexPath: 方法，设置 self.tableView.rowHeight = 88;
   - 若 cell 不定高，不要设置estimatedHeightForRow(因为 estimatedHeightForRow 不能和 heightForRow 里面的 layoutIfNeed 同时存在，这两者同时存在会出现“窜动”的bug)，解决方法是在请求到数据的时候提前计算好行高，用个字典缓存好高度；
3 加载网络数据，下载图片，使用异步加载，并缓存，从网络搂回来图片后先根据需要显示的图片大小切成合适大小的图，每次只显示处理过大小的图片，当查看大图时在显示大图，如果服务器能直接返回预处理好的小图和图片的大小更好。图片数量多时，必要的时候要准备好预览图和高清图，需要时再加载高清图，图片的’懒加载’方法，即延迟加载，当滚动速度很快时避免频繁请求服务器数据。
4 使用局部刷新
   - 尽量不要使用 reloadData，刷新某一分组、某一行使用对应的方法做局部刷新
5 渲染，尽量少用或不用透明图层
   - 将cell的opaque值设为Yes，背景色和子 view 不要使用clearColor，尽量不要使用阴影渐变、透明度也不要设置为0
6 少用addSubview给Cell动态添加子View，初始化时直接设置好，通过hidden控制显示隐藏，布局也在初始化时直接布局好，避免 cell 的重新布局
7 如果 cell 内显示的内容来自 web，使用异步加载，缓存结果请求。

8 按需加载cell，cell滚动很快时，只加载范围内的cell，如果目标行与当前行相差超过指定行数，只在目标滚动范围的前后制定n行加载。滚动很快时，只加载目标范围内的cell，这样按需加载，极大地提高了流畅性。方法如下：

```
//按需加载 - 如果目标行与当前行相差超过指定行数，只在目标滚动范围的前后指定3行加载。
- (void)scrollViewWillEndDragging:(UIScrollView *)scrollView withVelocity:(CGPoint)velocity
    targetContentOffset:(inout CGPoint *)targetContentOffset{
    NSIndexPath *ip = [self indexPathForRowAtPoint:CGPointMake(0, targetContentOffset->y)];
    NSIndexPath *cip = [[self indexPathsForVisibleRows] firstObject];
    NSInteger skipCount = 8;
    if (labs(cip.row-ip.row) > skipCount) {
        NSArray *temp = [self indexPathsForRowsInRect:CGRectMake(0, targetContentOffset->y, self.
            width, self.height)];
        NSMutableArray *arr = [NSMutableArray arrayWithArray:temp];
        if (velocity.y<0) {
            NSIndexPath *indexPath = [temp lastObject];
            if (indexPath.row+33) {
                [arr addObject:[NSIndexPath indexPathForRow:indexPath.row-3 inSection:0]];
                [arr addObject:[NSIndexPath indexPathForRow:indexPath.row-2 inSection:0]];
                [arr addObject:[NSIndexPath indexPathForRow:indexPath.row-1 inSection:0]];
            }
        }
        [needLoadArr addObjectsFromArray:arr];
    }
}
```

记得在tableView:cellForRowAtIndexPath:方法中加入判断：

```
if (needLoadArr.count>0&&[needLoadArr indexOfObject:indexPath]==NSNotFound) {
    [cell clear];
    return;
}
```

滚动很快时，只加载目标范围内的Cell，这样按需加载，极大的提高流畅度。

9 遇到复杂界面，像朋友圈涉及图文混排的，需要异步绘制，继承UITableViewCell，给自定义的Cell添加draw方法，在方法中利用 GCD 异步绘制；或者直接重写drawRect方但如果在重写drawRect方法就不需要用GCD异步线程了，因为drawRect本来就是异步绘制

绘制 cell 不建议使用 UIView，建议使用 CALayer。 UIView 的绘制是建立在 CoreGraphic 上的，使用的是 CPU。CALayer 使用的是 Core Animation，CPU、GPU 通吃，由系统决定使用哪个。View的绘制使用的是自下向上的一层一层的绘制，然后渲染Layer处理的是Texure，利用GPU的Texture Cache和独立的浮点数计算单元加速纹理的处理。GPU 不喜欢 透明，所以所有的绘图一定要弄成不透明，对于圆角和阴影这些的可以截个伪透明的小图然后绘制上去。在layer的回调里一定也只做绘图，不做计算！

cell被重用时，它内部绘制的内容并不会被自动清除，因此需要调用setNeedsDisplay或setNeedsDisplayInRect:方法。

65.离屏渲染（Offscreen-Renderd）

下面的情况或操作会引发离屏渲染：

- 为图层设置遮罩（layer.mask）
- 将图层的layer.masksToBounds / view.clipsToBounds属性设置为true
- 将图层layer.allowsGroupOpacity属性设置为YES和layer.opacity小于1.0
- 为图层设置阴影（layer.shadow *）。
- 为图层设置layer.shouldRasterize=true（光栅化）
- 具有layer.cornerRadius，layer.edgeAntialiasingMask，layer.allowsEdgeAntialiasing的图层（圆角、抗锯齿）
- 文本（任何种类，包括UILabel，CATextLayer，Core Text等）。
- 使用CGContext在drawRect :方法中绘制大部分情况下会导致离屏渲染，甚至仅仅是一个空的实现

优化方案
圆角优化 使用CAShapeLayer和UIBezierPath设置圆角；直接覆盖一张中间为圆形透明的图片（推荐使用）
shadow优化 使用ShadowPath指定layer阴影效果路径，优化性能
使用异步进行layer渲染（Facebook开源的异步绘制框架AsyncDisplayKit）
设置layer的opaque值为YES，减少复杂图层合成
尽量使用不包含透明（alpha）通道的图片资源
尽量设置layer的大小值为整形值

Core Animation工具检测离屏渲染

对于离屏渲染的检测，苹果为我们提供了一个测试工具Core Animation。可以在Xcode->Open Develeper Tools->Instruments中找到，如下图：

洋红色是因为像素没对齐，比如上面的 label，一般情况下因为像素没对齐，需要抗锯齿，图像会出现模糊的现象。
解决办法：在设置 view 的 frame 时，在高分屏避免出现 21.3，6.7这样的小数，尤其是 x，y坐标，用 ceil 或 floor 或 round 取整。每 0.5 个点对应一个 pixel，0.3,0.7这样的就难为 iPhone 了，低分屏不要出现小数。

黄色是因为显示的图片实际大小与显示大小不同，对图片进行了拉伸，测试显示使用 image view 显示实际大小的图也会变黄。
减少洋红色和黄色可以提升滚动的流畅性

65.UIView 和 CALayer

UIView是iOS中所有的界面元素都继承自它，它本身完全是由CoreAnimation来实现的。每一个UIView内部都默认关联着一个layer，真正的绘图部分，是由一个叫CALayer（Core Animation Layer）的类来管理；

UIView有个layer属性，可以返回它的主CALayer实例，UIView有一个layerClass方法，返回主layer所使用的类，UIView的子类，可以通过重载这个方法，来让UIView使用不同的CALayer来显示；

UIView的layer树形在系统内部，被维护着三份copy（presentLayer Tree、modeLayer Tree、render Tree），修改动画的属性，其实是 Layer 的 presentLayer 的属性值；

动画的运作：对UIView的subLayer（非主Layer）属性进行更改，系统将自动进行动画生成。

区别
1 UIView继承自UIResponder，能接收并响应事件， 负责显示内容的管理； 而CALayer继承自NSObject，不能响应事件，负责显示内容的绘制；

2 UIView侧重于展示内容，而CALayer则侧重于图形和界面的绘制；

3 当View展示的时候，View是layer的CALayerDelegate，View展示的内容是由CALayer进行display的；

4 view内容展示依赖CALayer对内容的绘制，UIView的frame也是由内部的CALayer进行绘制；

5 对UIView的属性修改，不会引起动画效果，但是对于CALayer的属性修改，是支持默认动画效果的，在view执行动画的时候，view是layer的代理，layer通过actionForLayer：forkey向对应的代理view请求动画action；

6 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，layer 比 view 多了个 anchorPoint；

7 一个CALayer的frame是由其anchorPoint，position，bounds，transform共同决定的，而一个UIView的的frame只是简单地返回CALayer的frame，同样UIView的center和bounds也只是简单返回CALayer的Position和Bounds对应属性。

66.TCP 和 UDP

TCP：传输控制协议,提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个TCP连接，一个TCP连接必须要经过“三次握手”才能建立起来，之后才能传输数据。TCP提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

UDP：用户数据报协议，是面向数据报的运输层协议。它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去！UDP适用于一次只传送少量数据、对可靠性要求不高的应用环境。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快

tcp协议和udp协议的差别
tcp-面向连接 udp-面向非连接
tcp-传输可靠 udp-不可靠
tcp-传输大量数据 udp-少量数据
tcp-速度慢 udp-快

TCP连接的三次握手

第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN_SEND状态，等待服务器确认；
第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；
第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次握手”（过程就不细写了，就是服务器和客户端交互，最终确定断开）

67.socket 和 http

socket连接和http连接的区别

http是基于socket之上的。socket是一套完成tcp，udp协议的接口。

HTTP协议：简单对象访问协议，对应于应用层 ，HTTP协议是基于TCP连接的

tcp协议：　　对应于传输层

ip协议：　　对应于网络层

TCP/IP是传输层协议，主要解决数据如何在网络中传输；而HTTP是应用层协议，主要解决如何包装数据。

Socket是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口，通过Socket，我们才能使用TCP/IP协议。

http连接：短连接，即客户端向服务器端发送一次请求，服务端响应后连接，请求结束后，会主动释放连接即会断掉

socket连接：长连接，理论上客户端和服务端一旦建立连接将不会主动断掉；但由于各种因素可能会断开，比如：服务端或客户端主机down了，网络故障，或两者之间长时间没有数据传输，网络防火墙可能会断开该连接以释放网络资源。所以当一个socket连接中没有数据的传输，为了维持连接需要发送心跳消息；

socket建立网络连接的步骤

建立socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket ，另一个运行于服务器端，称为ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

1 服务器监听：服务端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

2 客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务端套接字的地址和端口号，然后就向服务端套接字提出连接请求。

3 连接确认：当服务端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

socket编程中（荐研究GCDAsyncSocket）的断线重连机制

断线重连：使得IM软件能够长在线，或者短时间内掉线，最好可以做到用户无感知。目的是让IM软件维持在线的状态。

实现方法

IM客户端始终尽可能的保持连接跟服务器的连接，客户端维护已登录状态，以便断线重连。从逻辑层次上来说，断线重连的逻辑是基于登录逻辑的，首次登录成功后，都有可能断线重连

断线重连，实质分两步：一、使客户端断线；二、让客户端重连服务器。一般来说这两步是一个有前后顺序，完整的过程。

一、使客户端断线，即让客户端处于“未连接”状态。以下情况将触发这个事件：

1.网络切换，如从WiFi切换到4G，网络事件。

2.网络连接失败、网络不可用。

3.心跳失败、心跳超时，统称心跳失败。

4.IM软件后台运行即将结束。

二、让客户端重连服务器，客户端根据以下几种情况实现重连服务器。

1.iOS系统“网络可用”的通知

2.IM软件切换到前台，用户触发事件。

3.网络切换，如从WiFi切换到4G，网络事件。

4.心跳失败的事件。

5.客户端重新启动事件。

断线重连的场景总结为：

1．重新启动（自动登录）

需要提前加载用户缓存，保证用户到达主界面后能看到历史信息。

2．网络错误，网络切换

网络连接失败有很多种，不同的场景，客户端要使用不同的逻辑处理。

3．心跳失败

心跳超时，失败统称心跳失败。这个案例说明当前客户端—服务器连接已经损坏，或者当前用户身份有变化。心跳失败后首先将客户端离线，然后进行断线重连操作，避免心跳失败和网络错误事件一并发生，造成两次登录。

4．网络可达或者切换到前台

为了避免重复登录，当IM软件处于“登录成功”、“连接中”或者“已注销”的几个状态的时候，客户端忽略“网络可达或者切换到前台”的事件。

客户端心跳

IM基本的底层逻辑中有“心跳”概念，即客户端定时向Server发一个信令包，表示客户端还“活”着。注意，是客户端发起的。那么心跳终止了会发生什么事情呢？分两种情况：Server主动断开socket，客户端主动断开socket。

1．Server主动断开socket

Server只是接收客户端发起的心跳。假如，Server长时间没有收到客户端的心跳，Server认为客户端已经“死了”，主动断开这个连接。此时客户端可能就是假在线了。

2．客户端断开socket

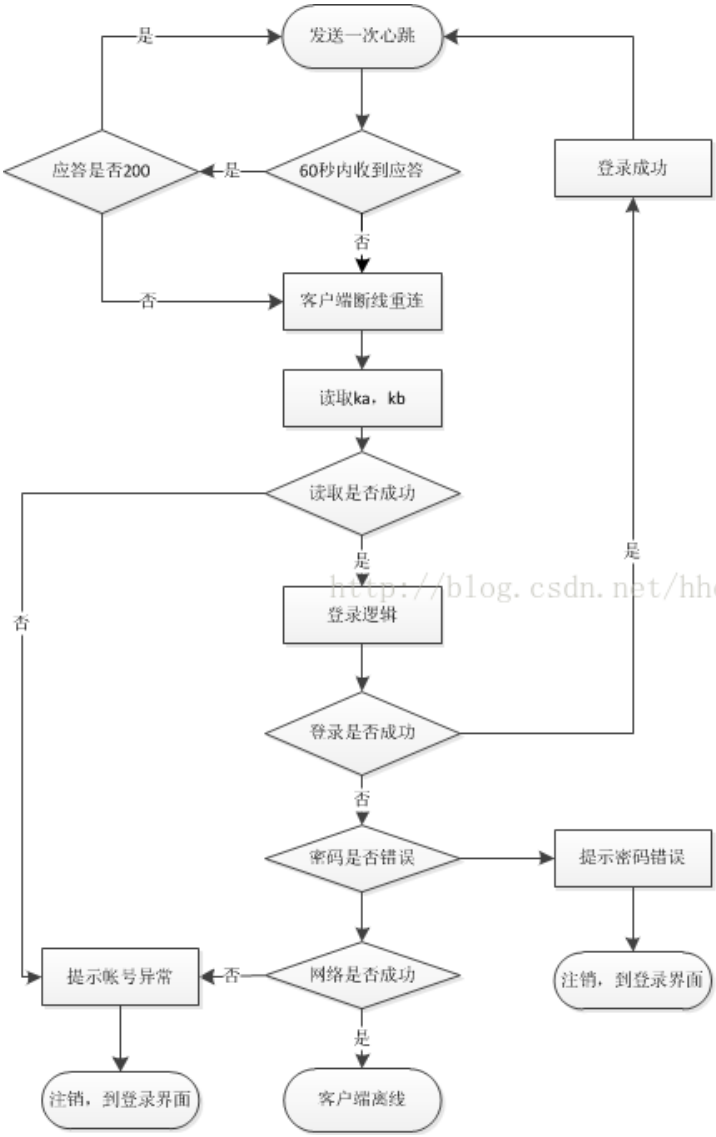
客户端对待心跳，要比Server麻烦一些。客户端要关注两个值：

- 心跳间隔值，即客户端多长时间发一次心跳？
- 心跳的超时时间。客户端发送一次心跳，如果长时间得不到Server应答，代表网络糟糕。客户端需要断开socket，主动离线。

很明显，第二点就是客户端主动断开的情况，一般情况下，超时时间为60秒。

网上也有争论：到底是否需要心跳，微信是没有心跳的，qq和飞信有心跳。
也有专家说心跳包已经影响到移动网络，因为心跳是定时频繁发送。

下面是“心跳失败”引起的断线重连的流程图



68.远程推送 APNS

远程推送的基本过程

1 客户端的app需要将用户的UDID和app的bundleID发送给apns服务器，进行注册，apns将加密后的device Token返回给app；

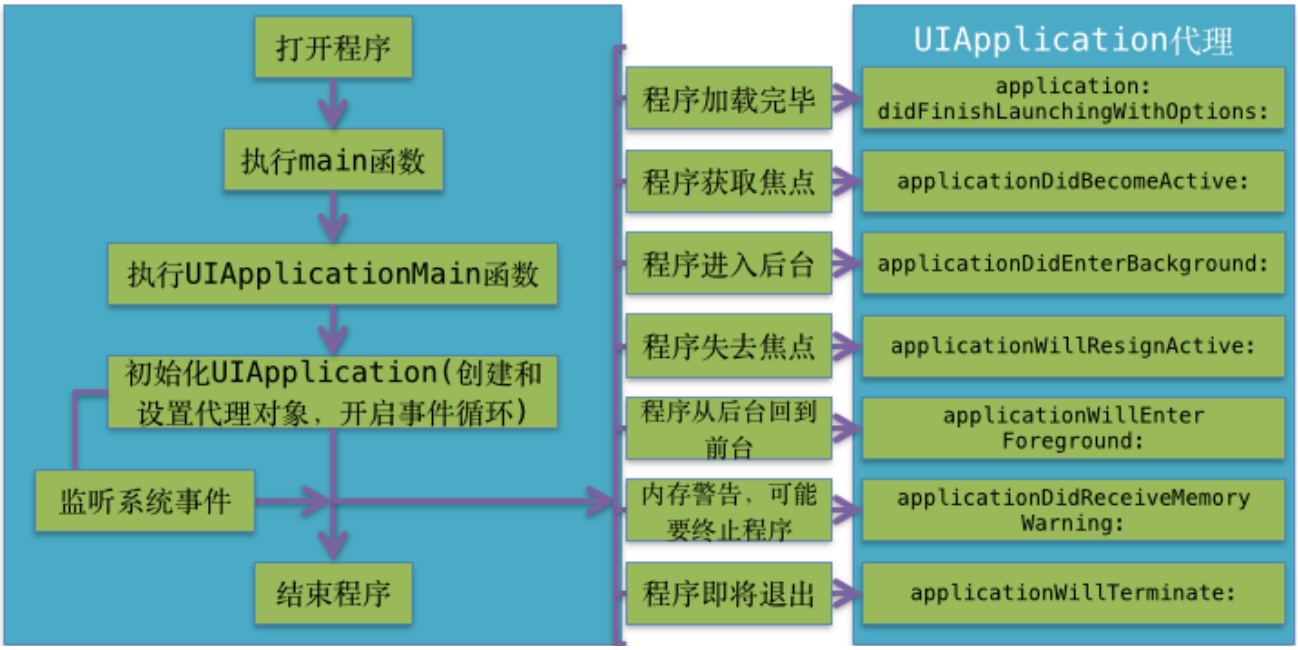
2 app获得device Token后，上传到公司服务器；

3 当需要推送通知时，公司服务器会将推送内容和device Token一起发给apns服务器；

4 apns再将推送内容推送到相应的客户端app上。

69.iOS应用程序生命周期

ios程序启动原理（过程）：如下



appdelegate 执行顺序：如下

```objc
#pragma mark - app 第一次启动
- (BOOL)application:(UIApplication *)application willFinishLaunchingWithOptions:(NSDictionary *)
    launchOptions {
    //进程启动但还没进入状态保存
    return YES;
}
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)
    launchOptions {
    //启动基本完成程序准备开始运行
    return YES;
}
- (void)applicationDidBecomeActive:(UIApplication *)application {
    //当应用程序进入活动状态执行
}

#pragma mark - 按 home 键
- (void)applicationWillResignActive:(UIApplication *)application {
    //当应用程序将要入非活动状态执行，在此期间，应用程序不接收消息或事件，比如来电话
}
- (void)applicationDidEnterBackground:(UIApplication *)application {
    //当程序被推送到后台的时候调用。所以要设置后台继续运行，则在这个函数里面设置即可
    [application beginBackgroundTaskWithExpirationHandler:^{
        NSLog(@"begin Background Task With Expiration Handler");
    }];
}

#pragma mark - 双击home键，再打开程序
- (void)applicationWillEnterForeground:(UIApplication *)application {
    //当程序从后台将要重新回到前台时候调用
}
- (void)applicationDidBecomeActive:(UIApplication *)application {
    //当应用程序进入活动状态执行
}

- (void)applicationWillTerminate:(UIApplication *)application {
    //当程序将要退出是被调用，通常用来保存数据和一些退出前的清理工作。需要要设置UIApplicationExitsOnSuspend的键值
}
```

1、应用程序的状态

状态如下：

Not running　未运行　程序没启动

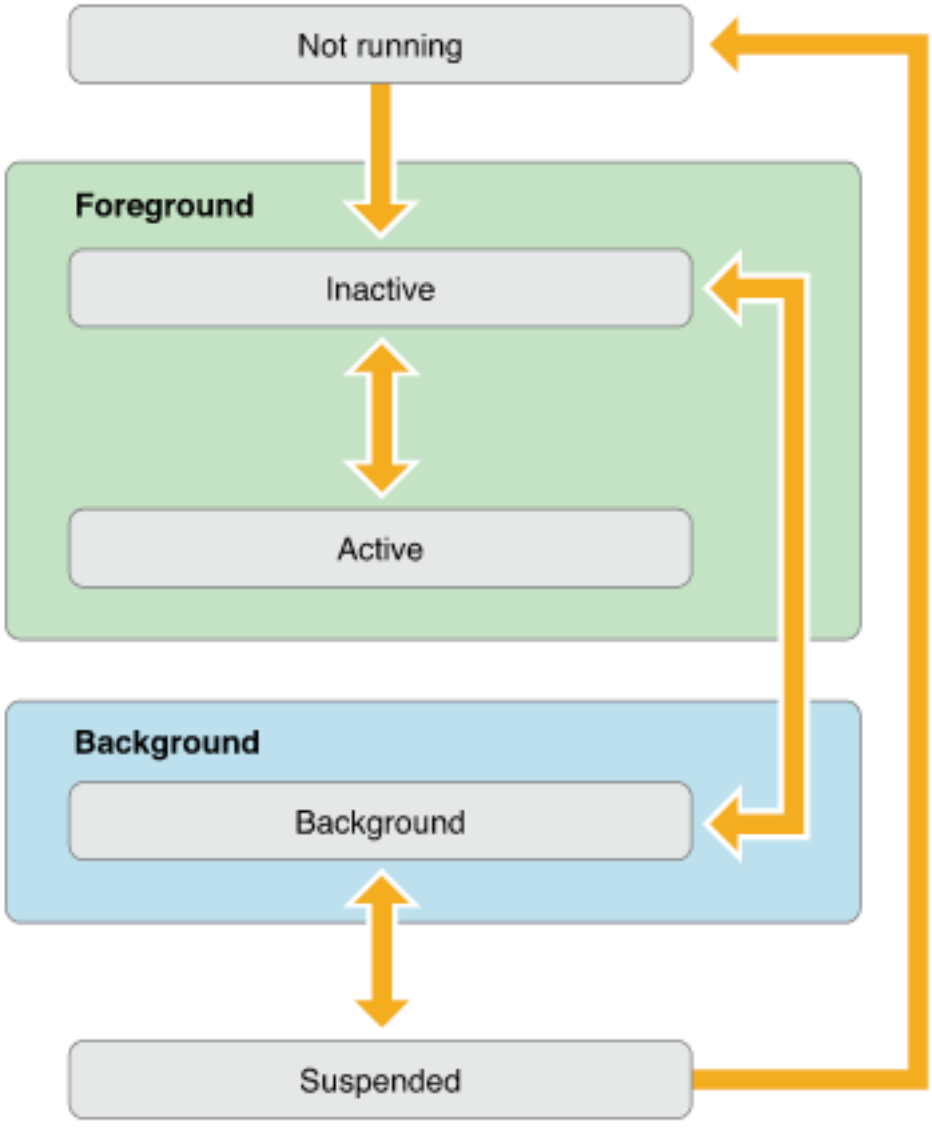
Inactive　未激活　程序在前台运行，不过没有接收到事件。在没有事件处理情况下程序通常停留在这个状态

Active　激活　程序在前台运行而且接收到了事件。这也是前台的一个正常的模式

Backgroud　后台　程序在后台而且能执行代码，大多数程序进入这个状态后会在在这个状态上停留一会。时间到之后会进入挂起状态(Suspended)。有的程序经过特殊的请求后可以长期处于Backgroud状态

Suspended　挂起　程序在后台不能执行代码。系统会自动把程序变成这个状态而且不会发出通知。当挂起时，程序还是停留在内存中的，当系统内存低时，系统就把挂起的程序清除掉，为前台程序提供更多的内存。

下图是程序状态变化图：



UIApplicationMain

```
UIApplicationMain(int argc, char *argv[], NSString *principalClassName, NSString *delegateClassName);
   /*
    argc：系统或者用户传入的参数个数
    argv：系统或者用户传入的实际参数
    argc、argv：直接传递给UIApplicationMain进行相关处理即可
    1.principalClassName：指定应用程序类名（app的象征），该类必须是UIApplication(或子类)。如果为nil，则系统默认UIApplication类
    2.delegateClassName：创建并设置UIApplication对象的代理，指定应用程序的代理类，该类必须遵守UIApplicationDelegate协议
      UIApplicationMain函数会根据principalClassName创建UIApplication对象，根据delegateClassName创建一个delegate对象，并将该delegate对象赋值给UIApplication对象中的delegate属性
    3.接着会建立应用的Main Runloop（事件循环），处理与用户交互产生的事件(首先会在程序完毕后调用delegate对象的application:didFinishLaunchingWithOptions:方法)
      程序正常退出时UIApplicationMain函数才返回
    */
```

70.view视图生命周期

视图创建：调用viewDidLoad方法
视图即将可见：调用viewWillAppear方法
视图已经可见：调用viewDidAppear方法
视图即将不可见：调用viewWillDisappear方法
视图已经不可见：调用viewDidDisappear方法
系统低内存：调用didReceiveMemoryWarning方法和viewDidUnload方法
注意：
1 viewDidLoad方法在应用运行的时候只会调用一次，其他方法会被调用多次。
2 低内存情况下，iOS会调用didReceiveMemoryWarning和viewDidUnLoad方法，但是iOS6以后就不在使用viewDidUnLoad方法，仅支持didReceiveMemoryWarning方法，该方法主要用于释放内存(视图控制器中的一些成员变量和视图的释放)

71.autorelease pool

autorelease的原理是什么?

autorelease实际上只是把对release的调用延迟了，对于每一个autorelease，系统只是把该Object放入了当前的Autorelease pool中，当该pool被释放时，该pool中的所有Object会被调用Release。

autorelease的对象何时释放?

对于autorelease pool本身，会在如下两个条件发生时候被释放

1、手动释放Autorelease pool

2、Runloop结束后自动释放

对于autorelease pool内部的对象在引用计数 == 0的时候释放。release和autorelease pool的drain都会触发引用计数减一。

NSAutoreleasePool是什么?

NSAutoreleasePool实际上是个对象引用计数自动处理器，在官方文档中被称为是一个类。NSAutoreleasePool可以同时有多个，它的组织是个栈，总是存在一个栈顶pool，也就是当前pool，每创建一个pool，就往栈里压一个，改变当前pool为新建的pool，然后，每次给pool发送drain消息，就弹出栈顶的pool，改当前pool为栈里的下一个 pool。

NSAutoreleasePool 中还提到，每一个线程都会维护自己的 autoreleasepool 堆栈，每一个 autoreleasepool 只对应一个线程。

NSAutoreleasePool和autoreleasepool的区别

当你使用ARC，就必须将NSAutoreleasePool的地方换成 @autoreleasepool

两者的作用时间不一样。AutoReleasePool对象的写法作用于运行时，@autoreleasepool作用于编译阶段。如果要启用ARC的话，在编译阶段就需要告诉编译器启用自动引用计数管理，而不能在运行时动态添加。

autoreleased 对象是被添加到了当前最近的 autoreleasepool 中的，只有当这个 autoreleasepool 自身 drain 的时候，autoreleasepool 中的 autoreleased 对象才会被 release 。这个对象的引用计数 -1 ，变成了 0 该 autoreleased 对象最终被释放

向一个对象发送-autorelease消息，就是将这个对象加入到当前AutoreleasePoolPage的栈顶next指针指向的位置。

自动释放池是NSAutoreleasePool的实例，其中包含了收到autorelease消息的对象。当一个自动释放池自身被销毁（dealloc）时，它会给池中每一个对象发送一个release消息（如果你给一个对象多次发送autorelease消息，那么当自动释放池销毁时，这个对象也会收到同样数目的release消息）

autorelease作用：

1. 对象执行autorelease方法时会将对象添加到自动释放池中
2. 当自动释放池销毁时自动释放池中所有对象作release操作

3 对象执行autorelease方法后自身引用计数器不会改变，而且会返回对象本身

4 autorelease实际上只是把对象release的调用延迟了，对于对象的autorelease系统只是把当前对象放入了当前对应的autorelease pool中，当该pool被释放时（[pool drain]），该pool中的所有对象会被调用Release,从而释放使用的内存。这个可以说是autorelease的优点，因为无需我们再关注他的引用计数，直接交给系统来做!

@autoreleasepool
自己 开启 一个线程，你需要创建自己的自动释放池块。如果你的应用或者线程长时间存活，并可能产生大量的自动释放的对象；你应该手动创建自动释放池块；否则，自动释放的对象会积累并占用你的内存。当使用ARC来管理内存时，在线程中使用 for 大量分配对象而不用autoreleasepool则可能会造成内存泄露
如果你创建的线程没有调用 Cacoa，你不需要使用自动释放池块。

在ARC环境下，使用@autoreleasepool可以释放池上下文，但是如下代码，@autoreleasepool有必要吗?

```
for (int i = 0; i < 1000000; i++) {
            @autoreleasepool {
             NSString *str = @"ABc";
             NSString *string = [str lowercaseString];
             string = [string stringByAppendingString:@"xyz"];
             NSLog(@"%@",string);
        }
    }
```

有必要，在遇到需要大量创建对象的地方使用autoreleasepool可以加快对象释放的速度。

72.view layout

相关方法

```
- (CGSize)sizeThatFits:(CGSize)size
- (void)sizeToFit
————————-

- (void)layoutSubviews
- (void)layoutIfNeeded
- (void)setNeedsLayout
————————–

- (void)setNeedsDisplay
- (void)drawRect
```

layoutSubviews在以下情况下会被调用:

1、init初始化不会触发layoutSubviews，但是用initWithFrame初始化时，当rect的值不为CGRectZero时，也会触发

2、addSubview会触发layoutSubviews

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

4、滚动一个UIScrollView会触发layoutSubviews
5、旋转Screen会触发父UIView上的layoutSubviews事件
6、改变一个UIView大小的时候也会触发父UIView上的layoutSubviews事件

layoutSubviews，当我们在某个类的内部调整子视图位置时，需要调用。
反过来的意思就是说：如果你想要在外部设置subviews的位置，就不要重写。

刷新子对象布局

-layoutSubviews：默认没有做任何事情，需要子类进行重写
-setNeedsLayout：标记为需要重新布局，异步调用layoutIfNeeded刷新布局，不立即刷新，但layoutSubviews一定会被调用
-layoutIfNeeded：如果有需要刷新的标记，立即调用layoutSubviews进行布局（如果没有标记，不会调用layoutSubviews）

注：setNeedsLayout方法并不会立即刷新，立即刷新需要调用layoutIfNeeded方法。
如果要立即刷新，要先调用[view setNeedsLayout]，把标记设为需要布局，然后马上调用[view layoutIfNeeded]，实现布局
setNeedsLayout 在receiver标上一个需要被重新布局的标记，在系统runloop的下一个周期自动调用layoutSubviews

layoutIfNeeded UIKit会判断该receiver是否需要layout，遍历的不是superview链，应该是subviews链

在视图第一次显示之前，标记总是“需要刷新”的，可以直接调用[view layoutIfNeeded]

重绘

-drawRect:(CGRect)rect：重写此方法，执行重绘任务
-setNeedsDisplay：标记为需要重绘，异步调用drawRect
-setNeedsDisplayInRect:(CGRect)invalidRect：标记为需要局部重绘

sizeToFit会自动调用sizeThatFits方法；
sizeToFit不应该在子类中被重写，应该重写sizeThatFits
sizeThatFits传入的参数是receiver当前的size，返回一个适合的size
sizeToFit可以被手动直接调用
sizeToFit和sizeThatFits方法都没有递归，对subviews也不负责，只负责自己

————————-

layoutSubviews对subviews重新布局
layoutSubviews方法调用先于drawRect

drawRect是对receiver的重绘，能获得context
setNeedDisplay在receiver标上一个需要被重新绘图的标记，在下一个draw周期自动重绘，
iphone device的刷新频率是60hz，也就是1/60秒后重绘

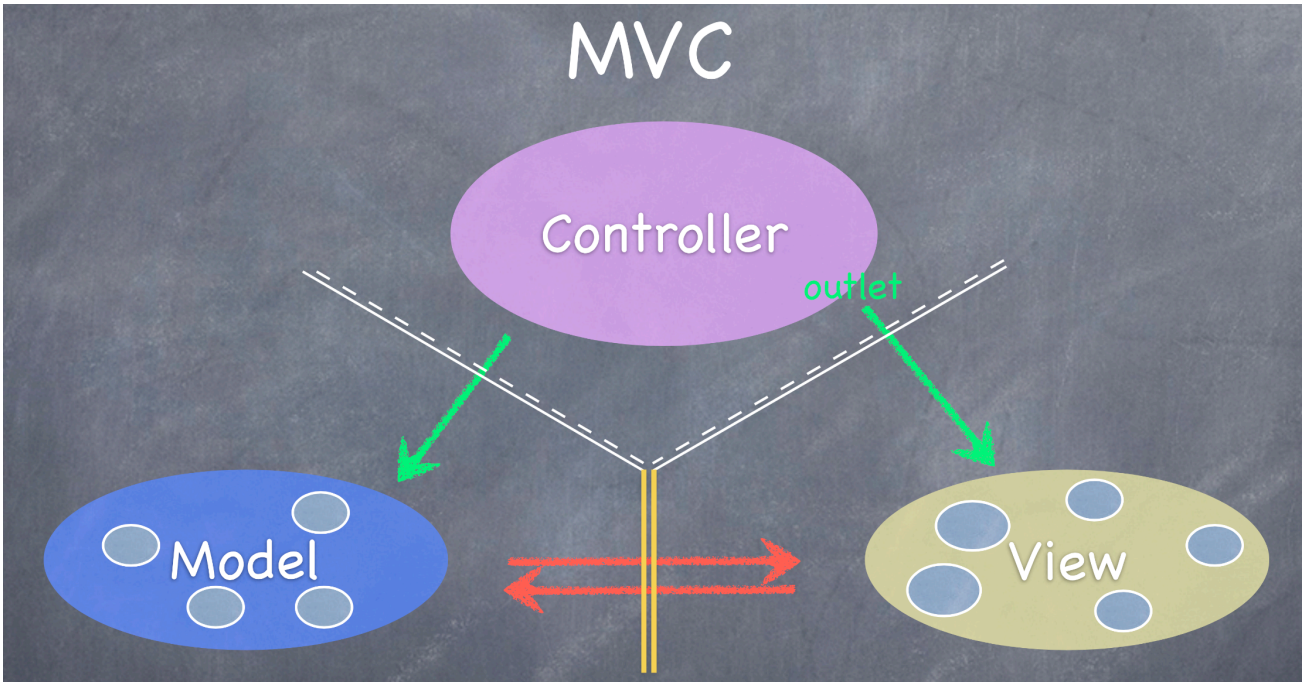
setNeedsDisplay 该方法在调用时，会自动调用drawRect方法。drawRect方法主要用来画图

总结
当需要刷新布局时，用setNeedsLayOut方法；当需要重新绘画时，调用setNeedsDisplay方法

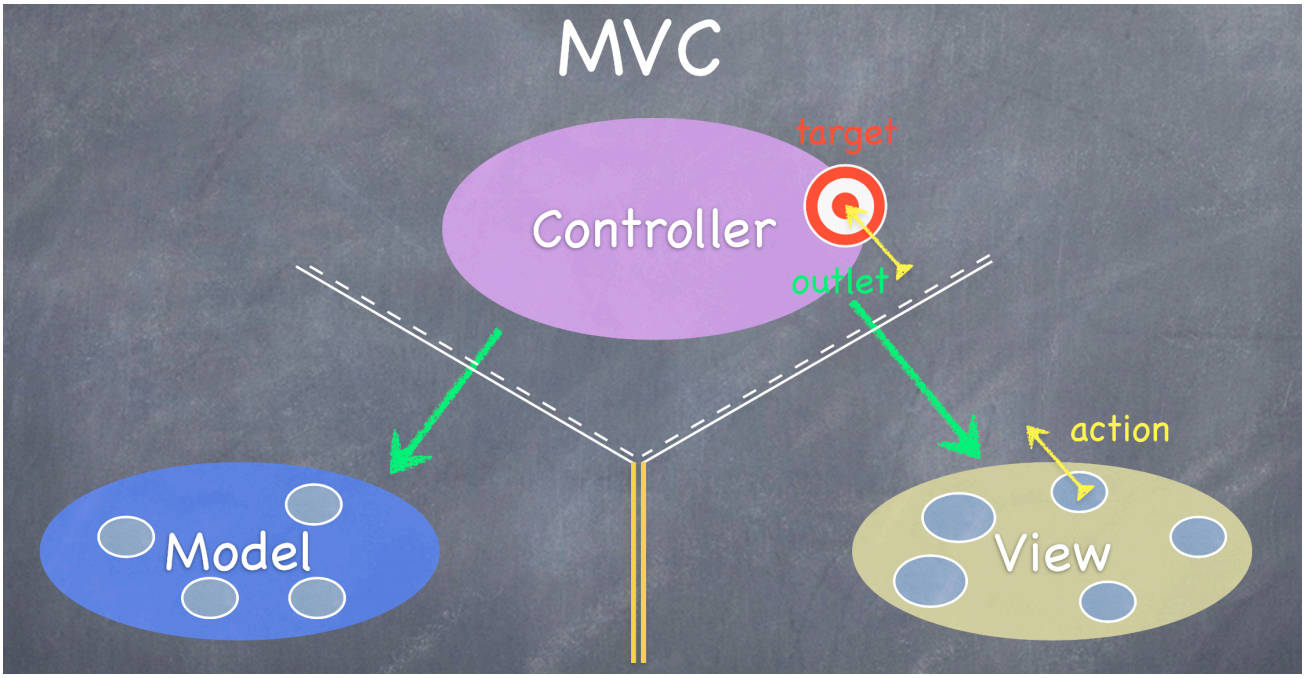
73.应用程序的架构 MVC、MVVM

- 界面
- 数据
- 事件
- 业务

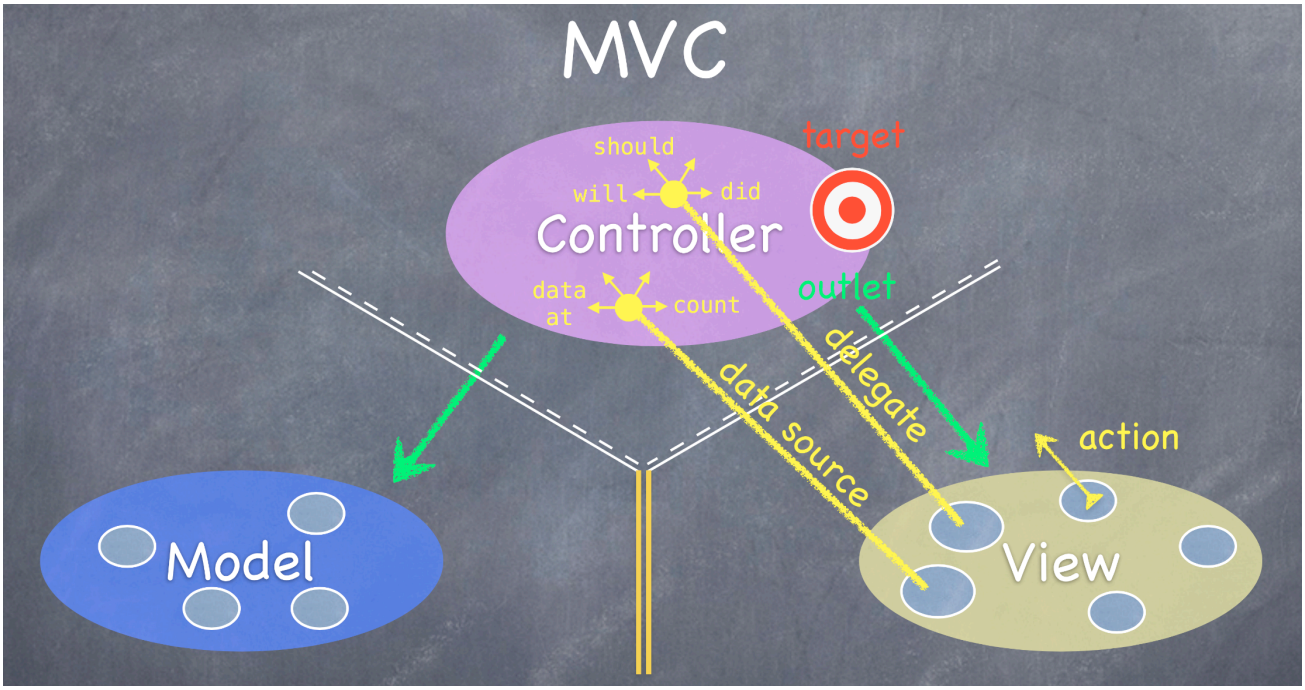
MVC
iOS应用程序都遵循Model-View-Controller的架构，Model负责存储数据和处理业务逻辑，View负责显示数据和与用户交互，Controller是两者的中介，协调Model和View相互协作。它们的通讯规则如下：
1、Controller能够访问Model和View，Model和View不能互相访问



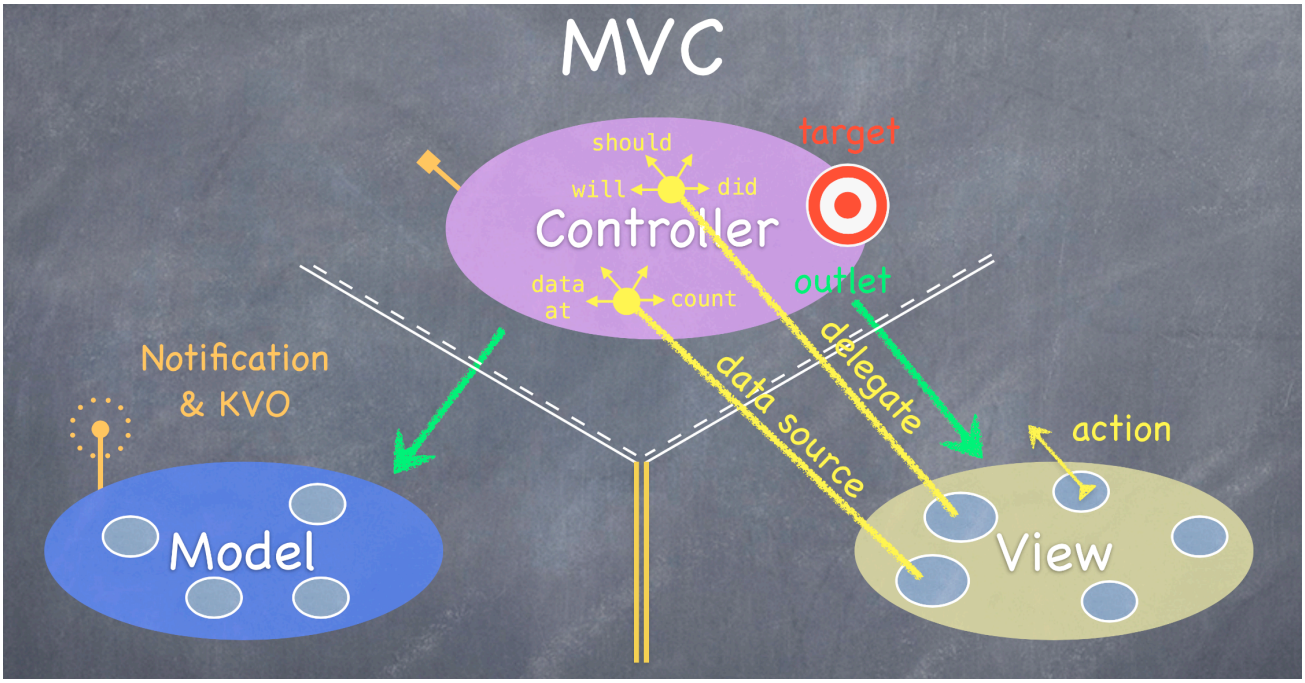
2、当View与用户交互产生事件时，使用target-action方式来处理



3、当View需要处理一些特殊UI逻辑或获取数据源时，通过delegate或data source方式交给Controller来处理



4、Model不能直接与Controller通信，当Model有数据更新时，可以通过Notification或KVO (Key Value Observing)来通知Controller更新View



MVVM

MVVM就是在MVC的基础上分离出业务处理的逻辑到viewModel层，即：

model层，API请求的原始数据、数据持久化

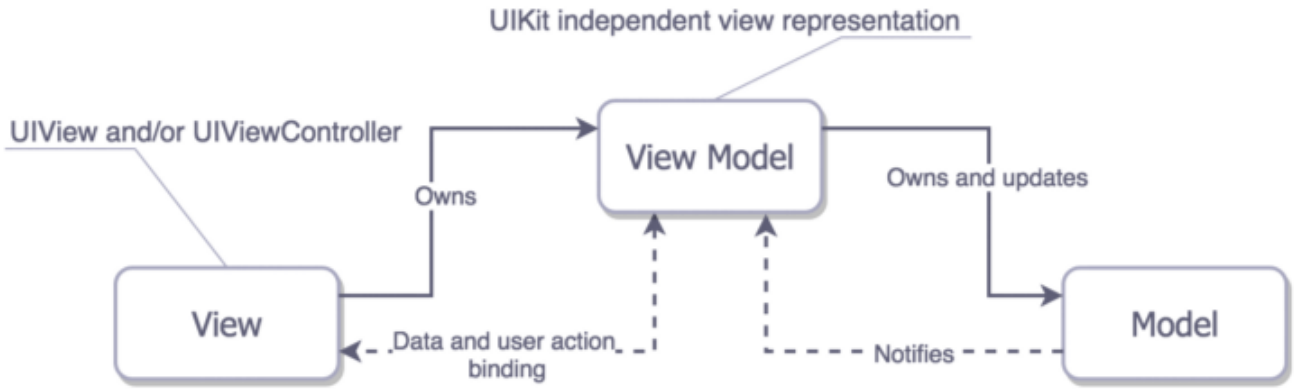
view层，视图展示，由viewController来控制

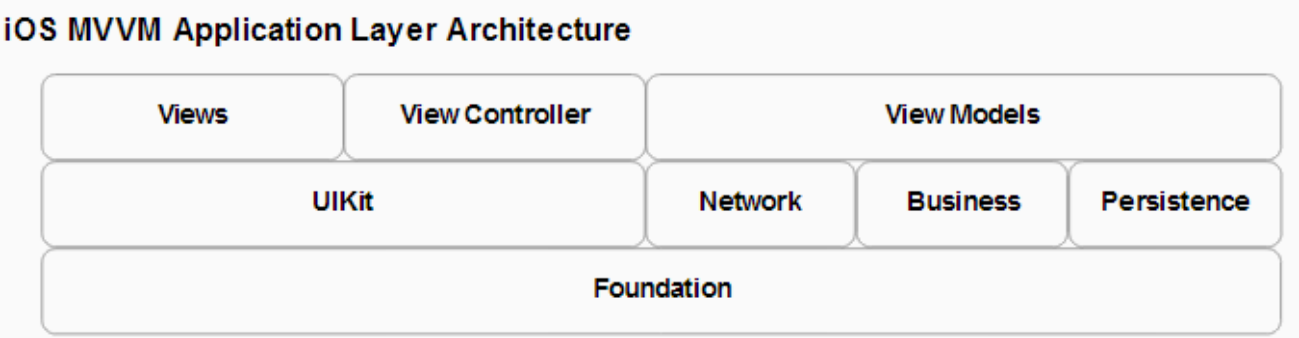
viewModel层，负责网络请求、业务处理和数据转化

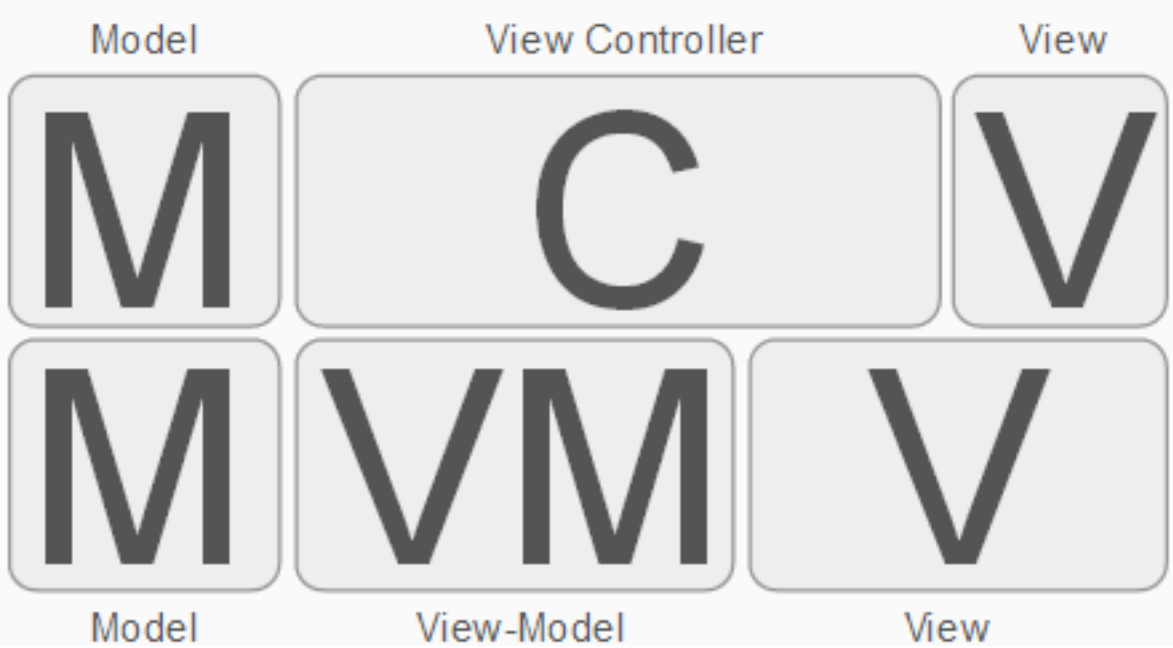
简单来说，就是API请求完数据，解析成model，之后在viewModel中转化成能够直接被视图层使用的数据，交付给前端。

经过viewModel转化之后的数据由viewModel保存，与数据相关的处理都将在viewModel中处理。viewModel返回给view层

view层是由viewController控制的。view层只做展示，不做业务处理。view层的数据由viewModel提供。







链接有 MVVM 使用详细介绍，里面也附有 Demo
http://www.cocoachina.com/ios/20170313/18870.html
http://www.cocoachina.com/ios/20170213/18659.html
http://www.cocoachina.com/ios/20150526/11930.html

74.FMDB、SQLite

FMDatabaseQueue 用于在多线程中执行多个查询或更新，它是线程安全的。

FMDatabase 这个类是线程不安全的，如果在多个线程中同时使用一个 FMDatabase 实例，会造成数据混乱等问题。

1、增(创)表

格式:

(1)create table 表名 (字段名 1 字段类型 1, 字段名 2 字段类型 2, ...) ;

(2)create table if not exists 表名 (字段名 1 字段类型 1, 字段名 2 字段类型 2, ...) ;

例如:

```
create table t_student (id integer, name text, age inetger, score real)
```

2、删表

格式:

(1)drop table 表名 ;

(2)drop table if exists 表名 ;

例如:

```
drop table t_student;
```

3、插入数据(insert)

格式:

insert into 表名 (字段1, 字段2, ...) values (字段1的值, 字段2的值, ...) ;

例如:

```
insert into t_student (name, age) values (‘mj’, 10);
```

【备注】数据库中的字符串内容应该用单引号 ’ 括住

4、更新数据(update)

格式:

update 表名 set 字段1= 字段1的值, 字段2= 字段2的值,...;

例如:

```
update t_student set name = ‘jack’, age = 20;
```

【备注】上面的示例会将 t_student 表中所有记录的 name 都改为 jack, age 都改为 20。 6、

5、删除数据(delete)

格式: delete from 表名 ;

例如:

```
delete from t_student;
```

75.简述内存分区情况

- 代码区：存放函数二进制代码
- 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
- 堆区：通过 malloc 等函数或 new 等动态申请到的，需手动申请和释放
- 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

76.各属性作用

- readwrite：可读可写，需要生成 getter、setter 方法
- readonly：只读，只生成 getter，不会生成 setter，不希望属性在类外部被更改
- assign：赋值，setter 方法将传入的参数赋值给实例变量；仅设置变量时，assign 用于基本数据类型
- retain：表示持有特性，setter 方法将传入的参数先保留，在赋值，传入参数的引用计数会+1
- copy：表示赋值特性，setter 方法将传入的对象复制一份
- 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

77.简述 Notification、KVC、KVO、delegate? 区别?

- KVO：一对多，观察者模式，键值观察机制，提供了观察某一属性变化的方法，极大简化了代码
- KVC：键值编码，一个对象在调用 setValue
    - 检查是否存在相应 key 的 set 方法，存在就调用 set 方法
    - set 方法不存在，就查找 _key 的成员变量是否存在，存在就直接赋值
    - 如果 _key 没找到，就查找相同名称的 key，存在就赋值
    - 如果都没有找到，就调用 valueForUndefinedKey 和 setValue:forUndefinedKey
- delegate：发送者和接收者的关系是直接、一对一的关系
- Notification：观察者模式，发送者和接收者的关系是间接、多对多的关系

区别：

- delegate 的效率比 Notification 高
- delegate 比 Notification 更直接，需要关注返回值，常带有 should 关键词；Notification 不关心结果，常带有 did 关键词
- 两个模块之间的联系不是很紧密，就用 Notification 传值，比如多线程之间的传值
- KVO 容易实现两个对象的同步，比如 Model 和 View 的同步

78.id 和 nil 代表什么

- id类型的指针可以指向任何 OC 对象
- nil 代表空值（空指针的值，0）

79.nil、Nil、NULL、NSNull

OC 中

- nil、Nil、NULL 都表示 (void *)0
- NSNull 继承于 NSObject，很特殊的类，表示是空，什么也不存储，但它却是对象，只是一个占位对象。使用场景：比如服务端接口让传空，NSDictionary *params = @{@“arg1” : @“value1”, @“arg2” : arg2.isEmpty ? [NSNull null] : arg2};

区别

- NULL 是宏，是对于 C 语言指针而使用的，表示空指针
- nil 是宏，是对于 OC 中的对象而使用的，表示对象为空
- Nil 是宏，是对于 OC 中的类而使用的，表示类指向空
- NSNull 是类类型，是用于表示空的占位对象，于 JS 或服务端的 null 类似的含义

80.向一个 nil 对象发送消息会发生什么?

- 向 nil 发送消息是完全有效的 —— 只是在运行时不会有任何作用
- 如果一个方法返回值是一个对象，那么发送给 nil 的消息将返回0(nil)
- 如果方法返回值为指针类型，其指针大小为小于或等于 sizeof(void *)，float、double、long double、long long 的整型标量，发送给 nil 的消息将返回0
- 如果方法返回值为结构体，发送给 nil 的消息将返回0，结构体中各个字段的值也都是0

81.self. 和 self-> 的区别

- self. 是调用 getter 或 setter 方法
- self 是当前本身，是一个指向当前对象的指针
- self-> 是直接访问成员变量

82.如何访问并修改一个类的私有属性

- 通过 KVC
- 通过 runtime 访问并修改

83.如何为 class 定义一个对外只读、对内可读写的属性

在头文件 .h 中将属性定义为 readonly，在 .m 中将原属性重新定义为 readwrite

84.OC 中，meta-class 指的是什么?

meta-class 是 class 对像的类，为这个 class 类存储类方法，当一个类发送消息时，就去这个类对应的 meta-class 中查找那个消息，每个 class 都有不同的 meta-class，所有的 meta-class 都使用基类的 meta-class(假如类继承自 NSObject，那么他所对应的 meta-class 也是 NSObject)作为他们的类。

85.NSString 用 copy 和 strong 的区别

NSString 为不可变字符串，用 copy 和 strong 都是只分配一次内存，但是如果用 copy，需要先判断字符串是不是不可变字符串，如果是不可变字符串，就不再分配空间，如果是可变字符串才分配空间。如果程序中用到的 NSString 特别多，每一次都要先判断就会耗费性能，用 strong 就不会再判断了，所以不可变字符串可以直接用 strong。

86.创建一个对象的步骤

- 开辟内存空间
- 初始化参数
- 返回内存地址值

87.setter、getter

- setter 方法：为外界提供一个设置成员变量的方法，好处 —— 不让数据暴露在外，保证数据安全性；对设置的数据过滤
- getter 方法：为调用者返回对象内部的成员变量
- 点语法的本质是对 setter 或 getter 方法的调用

88.id、instancetype是什么? 区别 ?

- id：万能指针，能作为参数、方法的返回类型
- instancetype：只能作为方法的返回类型，并且返回的类型是当前定义类的类类型

89.内存管理

- ARC 所做的是在代码编译期自动在合适的位置插入 release 或 autorelease，只要没有强指针指向对象，对象就会被释放。ARC 中不能手动使用 NSZone，也不能调用父类的 dealloc

调用对象的 release 方法会销毁对象吗?

- 不会，调用 release 只是将对象的引用计数器 -1，当对象的引用计数器 =0 时候会调用对象的 dealloc 方法才能释放对象的内存

objc 使用什么机制管理对象内存?

- 通过引用计数器（retainCount）的机制来决定对象是否需要释放。每次 RunLoop 完成一个循环的时候，都会检查对象的 retainCount，如果 retainCount 为0，说明该对象没有地方需要继续使用了，就被释放了。
- ARC 的判断准则：只要没有强指针指向对象，对象就会被释放

内存管理的范围?

- 管理所有继承自 NSObject 的对象，对基本数据类型无效。是因为对象和其他基本数据类型在系统中存储的空间不一样，其他局部变量主要存储在栈区（因为基本数据类型占用的存储空间是固定的，一般存放于栈区），而对象存储于堆中，当代码块结束时，这个代码块中的所有局部变量会自动弹栈清空，指向对象的指针也会被回收，这时对象就没有指针指向了，但依然存在于堆中，造成内存泄漏。

内存管理研究的对象：

- 野指针：指针变量没有进行初始化或指向的空间已经被释放。
  - 使用野指针调用对象方法，会报异常，程序崩溃
  - 在调用完 release 后，把保存对象指针的地址清空，赋值为 nil，找 oc 中没有空指针异常所以 [nil retain] 调用方法不会有异常
- 内存泄漏
  - 如 Person *p = [Person new];（对象提前赋值 nil 或清空）在栈区的 p 已经被释放，而堆区 new 产生的对象还没有释放，就会造成内存泄漏
  - MRC 造成内存泄漏的情况：1、没有配对释放，不符合内存管理原则；2、对象提前赋值 nil 或清空，导致 release 不起作用
- 僵尸对象：堆中已经被释放的对象（retainCount=0）
- 空指针：指针赋值为空，nil

- alloc、allocWithZone、new 时：该对象引用计数 +1；
- retain：手动为该对象引用计数 +1；
- copy：对象引用计数 +1；//注意：copy 的 oc 数据类型是否有 mutable，如有为深拷贝，新对象引用计数为 1；如果没有，为浅拷贝，引用计数 +1
- mutableCopy：生成一个新对象，新对象引用计数 +1；
- release：手动为该对象引用计数 -1；
- autorelease：把该对象放入自动释放池，当自动释放池释放时，向池中的对象发送 release 消息，其内的对象引用计数 -1，只能释放自己拥有的对象；

- NSAutoreleasePool：NSAutoreleasePool 是通过接收对象向它发送的 autorelease 消息，记录该对象的 release 消息，当自动释放池被销毁时，会自动向池中对象发送 release 消息。
- 对象如何加入池中：调用对象的 autorelease 方法
- 多次调用对象的 autorelease 方法会导致：野指针异常

90.KVC 的底层实现?

当一个对象调用 setValue 方法时，方法内部会做以下操作：
1、检查是否存在对应 key 的 set 方法，如果存在，就调用 set 方法
2、如果 set 方法不存在，就会查找与 key 相同名称并且带下划线的成员变量 _key，如果有，则直接给成员变量赋值
3、如果没有找到 _key，就会查找相同名称的属性 key，如果有就直接赋值
4、如果还没找到，则调用 valueForUndefinedKey: 和 setValue:forUndefinedKey:
这些方法的默认实现都是抛出异常，可以根据需要重写他们

91.block 的内存管理

- 无论 ARC 还是 MRC，只要 block 没有访问外部变量，block 始终在全局区
- MRC 下：
  - block 如果访问外部变量，block 在栈区
  - 不能对 block 使用 retain，否则不能保存在堆区
  - 只有使用 copy，才能放到堆区
- ARC 下
  - block 如果访问外部变量，block 在堆区
  - block 是一个对象，可以使用 copy 或 strong 修饰，最好是使用 copy

92.App 的启动过程，从 main 说起

App 启动分两类：1、有 stroyboard；2、无 storyboard
有 stroyboard 情况下：
1 main 函数
2 UIApplicationMain
  - 创建 UIApplication 对象
  - 创建 UIApplication 的 delegate 对象
3 根据 info.plist 获得 Main.storyboard 的文件名，加载 Main.storyboard
  - 创建 UIWindow
  - 创建和设置 UIWindow 的 rootViewController
  - 显示窗口 window

没有 stroyboard 情况下：

1 main 函数
2 UIApplicationMain
  - 创建 UIApplication 对象
  - 创建 UIApplication 的 delegate 对象
3 delegate 对象开始处理（监听）系统事件
  - 程序启动完毕时，调用 didFinishLaunching 方法
  - didFinishLaunching 方法中创建 UIWindow 设置 rootViewController 并显示窗口 window

93.tableview 的 cell 里面如何嵌套 collectionview

用自定义的继承自 UITableViewCell 的类，在 initWithFrame 方法里面初始化自定义的继承自 UICollectionView 的类

94.awakeFromNib 和 viewDidLoad 的区别

- awakeFromNib：当 .nib 文件被加载的时候，会发送一个 awakeFromNib 消息到 .nib 文件中每个对象，每个对象都可以定义自己的 awakeFromNib 来响应这个消息。即通过.nib 文件创建的 view 对象会执行awakeFromNib
- viewDidLoad：当 view 被加载到内存就会执行，不管是通过 nib 还是代码形式

95.常见的 Crash 场景

- 访问僵尸对象
- 访问了不存在的方法
- 数组越界
- 在定时器下一次回调前将定时器释放，会 crash

96.AFN 断点续传

- 检查服务端文件信息
- 检查本地文件
- 如果本地文件比服务端文件小，断点续传，利用 HTTP 请求头的 Range 实现断点续传
- 如果比服务端文件大，重新下载
- 如果和服务端文件一样，下载完成

afn 默认超时时间是60s

97.客户端的缓存机制

缓存分为：内存数据缓存、数据库缓存、文件缓存
- 每次想获取数据的时候，先检测内存中有无缓存
- 在检测本地有无缓存（数据库/文件）
- 最终发送网络请求
- 将服务端返回的数据进行缓存（内存、数据库、文件）

98.数据存储方式

4种数据持久化：属性列表（plist）、对象归档、sqlite、Core Data
NSUserDefaults 用于存储配置信息

如何存储用户的一些敏感信息，如登录的 token?
使用 keychain 来存储，即钥匙串，使用 keychain 需要导入 Security 框架

使用 NSUserdefaults 时，如何处理布尔的默认值？（比如返回 NO，不知道是真的 NO，还是没有设置过的 NO）

```
if ([[NSUserDefaults standardUserDefaults] objectForKey:@"key"] == nil) {
        NSLog(@"没有设置过");
}
```

99.App 需要加载超大量的数据，给服务器发送请求，但服务器卡住了，如何解决?

- 设置请求超时
- 给用户提示请求超时
- 根据用户操作再次请求数据

100.网络图片处理问题中怎么解决一个相同的网络地址重复请求的问题?

- 利用字典（图片地址为 key，下载操作为 value）

101.如何用 GCD 同步若干个异步调用？（如根据若干个 url 异步加载多张图片，然后在都下载完成后合成一张整图）

使用 dispatch_group_async 追加 block 到 global queue 中，这些 block 全部执行完毕，就会执行 main dispatch queue 中的结束处理的 block。

代码如下：

```
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    dispatch_group_t group = dispatch_group_create();
    //下图片1,放线程组中
    dispatch_group_async(group, queue, ^{
        NSURL *url = [NSURL URLWithString:@""];
        NSData *imgData1 = [NSData dataWithContentsOfURL:url];
        self.img1 = [UIImage imageWithData:imgData1];
    });

    //下载图片2
    dispatch_group_async(group, queue, ^{
        NSURL *url = [NSURL URLWithString:@""];
        NSData *imgData2 = [NSData dataWithContentsOfURL:url];
        self.img2 = [UIImage imageWithData:imgData2];
    });

    //合并图片
    dispatch_group_notify(group, queue, ^{
        NSLog(@"combie---%@", [NSThread currentThread]);
        UIGraphicsBeginImageContext(CGSizeMake(200, 200));
        [self.img1 drawInRect:CGRectMake(0, 0, 200, 100)];
        self.img1 = nil;
        [self.img2 drawInRect:CGRectMake(0, 100, 200, 100)];
        self.img2 = nil;
        UIImage *img = UIGraphicsGetImageFromCurrentImageContext();
        UIGraphicsEndImageContext();

        dispatch_async(dispatch_get_main_queue(), ^{
            NSLog(@"UI---%@", [NSThread currentThread]);
            //self.imgView.image = img;
        });
    });
```

102.NSOperation、GCD、NSThread 的区别

NSOperation 与 GCD 的区别：

GCD：

- GCD 是iOS4.0推出的，纯 c
- GCD 是将任务（block）添加到队列（串行、并行、全局、主队列），并且以同步/异步的方式执行任务的函数
- GCD 提供 NSOperation 不具备的功能：
  - 一次性执行
  - 延迟执行
  - 调度组
  - GCD 是严格的队列，先进先出 FIFO

NSOperation：

- NSOperation 是 iOS2.0推出的，iOS4.0 以后又重写的
- NSOperation 是将操作（异步任务）添加到队列（并发队列），就会执行指定的函数
- NSOperation 提供的方便操作：
  - 最大并发数
  - 队列的暂停和继续
  - 取消所有的操作
  - 指定操作之间的依赖关系，可以让异步任务同步执行
  - 可以利用 KVO 监听一个 operation 是否完成
  - 可以设置 operation 的优先级，能使同一个并行队列中的任务区分先后地执行
  - 对 NSOperation 继承，在这之上添加成员变量和成员方法，提高代码的复用度

GCD 与 NSThread 的区别：

- NSThread 使用 @selector 指定要执行的方法，代码分散
- GCD 通过 block 指定要执行的方法，代码集中
- GCD 不用管理线程的生命周期（创建、销毁、复用）
- 如果要开多个线程 NSThread 必须实例化多个线程对象
- NSThread 通过 performSelector 方法实现线程间通信

为什么要取消/恢复队列?

- 一般内存警告后取消队列中的操作
- 为了保证 scrollView 在滚动的时候流畅，通常在滚动开始时，暂停队列中的所有操作，滚动结束后，恢复操作

__bridge : 用于 Foundation 和 Core Fundation 之间数据的桥接

103.是否可以把比较耗时的操作放在 NSNotificationCenter 中

- 如果在异步线程发的通知，可以执行耗时的操作
- 如果在主线程发的通知，则不可以执行耗时的操作

备注:

本文 **3-7、9-57** 题均出自https://github.com/ChenYilong/iOSInterviewQuestions 但是不是全部的拷贝，只是整理自我感觉重要的、关键点上的内容，如果看不懂，请移步原文详细阅读。

本文剩下的题目均是参考了网络各个大v的资源，外加自己学习总结，每个知识点均是查阅不下于 20 个相关内容的网页（可以说一个知识点基本要花费一天的时间吧），进行提炼整理，力尽准确、精要。

每个题目中感觉重要的都是用 其他色 做标注了

利用预渲染加速iOS设备的图像显示

```
1.  static const CGSize imageSize = {100, 100};
2.
3.  - (void)awakeFromNib {
4.      if (!self.image) {
5.          self.image = [UIImage imageNamed:@"random.jpg"];
6.          if (NULL != UIGraphicsBeginImageContextWithOptions)
7.
                UIGraphicsBeginImageContextWithOptions(imageSize, YES, 0
    );
8.          else
9.              UIGraphicsBeginImageContext(imageSize);
10.         [image drawInRect:imageRect];
11.         self.image = UIGraphicsGetImageFromCurrentImageContext();
12.         UIGraphicsEndImageContext();
13.     }
14. }
```

这里需要判断一下UIGraphicsBeginImageContextWithOptions是否为NULL，因为它是iOS 4.0才加入的。

由于JPEG图像是不透明的，所以第二个参数就设为YES。

第三个参数是缩放比例，iPhone 4是2.0，其他是1.0。虽然这里可以用[UIScreen mainScreen].scale来获取，但实际上设为0后，系统就会自动设置正确的比例了。

TableView中实现平滑滚动延迟加载图片

利用CFRunLoopMode的特性，可以将图片的加载放到NSDefaultRunLoopMode的mode里，这样在滚动UITrackingRunLoopMode这个mode时不会被加载而影响到。主线程繁忙的时候performSelector:withObject:afterDelay:会延后执行，所以在发生触摸或是视图还在滚动时这个方法不会运行；

```
UIImage *downloadedImage = ...;
[self.avatarImageView performSelector:@selector(setImage:)
     withObject:downloadedImage
     afterDelay:0
     inModes:@[NSDefaultRunLoopMode]];
```

注意：使用masonry进行label的多行显示设置时，主要是如下两个参数的设置

1、@property(nonatomic)CGFloat preferredMaxLayoutWidth

2、- (void)setContentHuggingPriority:(UILayoutPriority)priority forAxis:(UILayoutConstraintAxis)axis