UI相关

图像显示原理

iOS的所有视图控件都是继承自UIView。根据“单一职责”的设计原则，真正负责显示和动画操作的部分是CALayer。CALayer有一个id类型属性叫contents，它实际对应着一个CGImageRef（位图）。也就是说屏幕上显示每一帧都是由n张位图**合成**的。

- 每一帧画面的生成是由CPU和GPU合作完成的。 CPU主要负责包括对象的创建、布局计算、文本计算与渲染、如果是图片的话，还会在提交给GPU之前进行图片的**解码和绘制**。最后将CPU生成的位图，通过Core Animation提交给GPU。GPU的部分就是OpenGL渲染管线相关的工作。主要就包括**顶点着色**、**图元装配**、**几何着色**、**光栅化**、**片段着色**、**片段处理**等，最后将生成的内容放到帧缓存区中。等到VSync信号到来之前，视图控制器就会去帧缓存区进行提取将要显示的内容。总的来说，Core Graphics渲染部分由CPU完成，OpenGL部分由GPU完成。

UI卡顿掉帧的原因

在VSync信号到来后，系统会通过CADisplayLink等机制通知App。然后视图控制器就会去帧缓存区提取显示内容。那如果一个VSync信号周期内，CPU和GPU没能协作完成提交，那么那一帧就会被丢弃，此时屏幕就不会刷新，仍然显示之前的内容。这样就造成了卡顿。

- 按照苹果60FPS的刷帧率，每隔16.7ms就会有一次VSync信号。

卡顿优化

根据上面的图像显示原理，要保证不卡顿，则需要让CPU和GPU在16.tms之内生成将要显示内容。所以卡顿优化就主要从CPU和GPU两个方向入手：

CPU优化

- 对象创建：对象创建会分配内存、调整属性、甚至读取文件，比较消耗CPU。策略就是**尽量用轻量级的对象创建**，比如CALayer代替无需操作的UIView。又比如单张图片时，直接给CALayer添加了一个setImageWithURL的方法，将图片直接赋值给layer.contents。其次就是**尽量推迟创建对象的时间**，比如使用懒加载。而对于可以复用的对象，**尽可能放到缓存池复用**，比如Cell。
- 对象调整：主要是CALayer的一些属性的调整，比如frame、bounds等，它实际上是通过运行时resolveInstanceMethod为对象临时添加一个方法，然后将对应属性保存到Dictionary里，同时通知Delegate、创建动画。 其次改变CALayer的一些**可动画属性值**时，会对模型层数据做动画，最后显示在展示层上，也多了一份数据的拷贝。所以对象(尤其是视图里的对象)的调整应尽量避免，尤其是视图层次，以及频繁地添加和移除视图。
- 布局计算和渲染：尽可能使用**预排版**的方式处理布局和排版。其次尽量少调整，对于复杂的视图来说，尽可能不使用Autolayout和storyboard。
- 控制线程的最大并发数；
- 列表多使用**预排版**、**预渲染**、**异步绘制**、**按需加载**、**延时加载**等技术；
- 图片的编解码(比如：**SDWebImage编解码优化**)；

GPU优化

- 尽量避免出现**离屏渲染**；

- 视图混合：应该尽量减少视图数量与层次。不透明的视图表明opaque属性，避免无用的Alpha通道合成。
- 纹理的渲染：实际就是指图片的渲染。iOS中几乎所有的UI视图最终都是绘制成Bitmap，包括文本、图片、栅格化的内容。所以可以尽量减少短时间内大量图片的显示，尽可能将多张图片合成到一张图片中。其次对于过大的图片，比如超过GPU的最大纹理尺寸（4096*4096）,则需要CPU先预处理。所以尽量不要让图片大小超过这个值。 iOS性能优化

关键名词解析

- **合成**：是指一帧画面是由多张位图组成的，所以将多张位图组合成一帧画面的过程就是合成。比如位图的重叠部分的像素怎么处理等等！
- **解码**：解码主要是因为每一种图片都有不同的格式，要使图片能够显示在屏幕上，就得将不同格式的图片转码成图片的原始像素数据。这样才能进行绘制。
- **顶点着色**：是指把3D坐标转化为2D坐标；
- **图元装配**：根据顶点着色器的输入，将所有坐标装配成点、线、三角形这些基础的图元；
- **几何着色器**：根据图元数据生成几何形状；
- **光栅化**：就是把图元最终映射成屏幕上的像素，生成片段。片段是指一个像素渲染所需要的所有数据。
- **片段着色**：就将光栅化的结果先进行裁剪，去掉超出视图以外的像素，然后进行着色；
- **片段处理**：检测片段的对应深度值和透明度，丢弃被挡住的部分。然后对图层进行混合。
- **SDWebImage编解码优化**：

  iOS默认会在UI主线程对图像进行解码。SDWebImageDecoder的优化思路就是将解码的耗时工作放到了子线程中进行，解码完成后就缓存到内存避免重复解码。

  解码的主要函数：[UIImage decodedImageWithImage:img];
- **离屏渲染**： 在OpenGL中有两种渲染方式：当前屏幕内渲染和离屏渲染(在当前屏幕缓存区以外开辟一个新的缓存区进行渲染)。离屏渲染消耗性能的原因包括： 1.需要创建新的缓存区； 2.在离屏渲染的过程中，需要多次在当前屏幕和新的缓存区之间进行环境切换。 哪些操作会触发离屏渲染？(因为有的组合属性在没有合成之前是不能在当前屏幕缓存区中使用的) 1.光栅化：layer.shouldRasterize = true; 2.遮罩：layer.mask 3.圆角和masksToBounds一起使用的时候 4.阴影。 所以当遇到这些容易触发离屏渲染的情况的时候，可以考虑使用CPU渲染，也就是直接调用Core Graphics的API绘制来替代。
- **处理时机与Core Animation的Observe回调**：Core Animation在RunLoop中注册了一个优先级比较低的Observer，这个Observe监听了RunLoop的BeforeWaiting或Exit状态。当处理了UI时，比如frame、layer的层次，或者手动调用了setNeedsDisplay等方法。对应的CALayer就会标记为待处理，并被提交到一个全局容器里。 当Observe监听到通知后，然后回调方法里就会将待处理的方法进行实际的绘制，最后将合成的位图交由Core Animation。因为GPU处理的单元是Texture(纹理)，所以Core Animation实际会先创建一个OpenGL的Texture，并将contents的CGImageRef和这个Texture进行绑定，通过TextureID来标识。之后GPU将Texture渲染到屏幕上。
- **UIView绘制流程**： 当调用UIView的setNeedsDisplay方法时，会调用CALayer的同名方法setNeedsDisplay，但是这时并没有立即发送绘制，而只是给CALayer打上了脏标记，然后将其放到了一个全局容器里。当Core Animation监听到RunLoop的BeforeWaiting或Exit状态时，则会回调将容器里的CALayer执行display方法，然后这个方法的内部会判断是否实现了layer.delegate的

displayLayer：方法。如果没有实现，则是正常走系统绘制方法，如果实现了，则执行displayLayer。

- 系统绘制：系统绘制首先会创建一个后备缓存(backing store),然后调用CALayer的drawInContext方法，drawInContext方法内会判断是否实现了layer.delegate的drawLayer:inContext:方法，如果实现了，就执行drawLayer:inContext:，如果没有就执行默认的drawRect:方法。
- 异步绘制：一般来说异步绘制就是在layer.delegate的两个方法displayLayer:或者drawLayer:inContext:方法里，在子线程进行绘制工作，然后回到主线程给layer的contens进行赋值。不过最好的选择是使用displayLayer:方法，因为它节省了创建backing store的开销。

```
- (void)displayLayer{
dispatch_async(backgroundQueue, ^{
      CGContextRef ctx = CGBitmapContextCreate(...);
      // draw in context...
      CGImageRef img = CGBitmapContextCreateImage(ctx);
      CFRelease(ctx);
      dispatch_async(mainQueue, ^{
          layer.contents = img;
      });
    });
}
```

- drawRect绘制过程: iOS的绘图操作默认是在drawRect:方法中进行的，我们也可以重写drawRect方法进行执行绘制操作。drawRect方法和drawLayer:inContext:方法都直接有绘制上下文，可以直接通过UIGraphicsGetCurrentContext获取。但是displayLayer方法无法直接获取到上下文，需要自己创建：创建：CGBitmapContextCreate(...);最后使用CGBitmapContextCreateImage获取bitmap。然后给contents赋值。
- CALayer的后备存储Backing Store : 当我们调用CALayer的drawInContext方法时，layer会自动创建一块以layer大小成正比的内存区域，这个区域就是后备存储（backing store）。所以也最好不要随意重写drawRect:等相关方法。

OC基础特性

KVO

KVO是OC观察者模式的实践之一。KVO的实现原理是通过Runtime的**isa混写**技术给当前对象A生成一个NSNotifying_A的子类，然后重写所要监听的属性的Setter方法，Setter方法里会调用willChangeValueForKey:和didChangeValueForKey:方法。然后就会在ObserveValueForKey里监听到变化。

```
- (void)setB:(id)obj
{
    [self willChangeValueForKey:@"B"];
    [super setB:obj]; //调用原类的实现
    [self didChangeValueForKey:@"B"];
}
```

注意：

- kvo触发的关键点在于重写了属性的Setter方法，然后写了willChangeValueForKey和didChangeValueForKey方法，所以**直接给成员变量 _ B赋值是不会触发KVO的**，但是直接手动调用这两个方法也是可以触发KVO的；
- KVC的setValue:ForKey:也能触发KVO，这也是KVC和KVO直接的联系；

KVC

KVC——KeyValueCoding键值编码。是苹果给NSObject添加的一个分类，它可以无需直接访问存Setter、取getter方法就能对属性进行操作(**所以这违反了面向对象的"封装性"的特性**)。主要的方法是setValueForKey:和valueForKey: 及KeyPath的变种。

注意：

- valueForKey:的时候会先按照getKey:、key、isKey、 _ key的顺序查找方法，然后判断accessInstanceVariablesDirectly，走后续流程；
- setValue:forKey：的时候先按照setKey:、 _ setKey:的顺序查找方法，然后判断accessInstanceVariablesDirectly，走后续流程；
- accessInstanceVariablesDirectly 设置为YES，则如果没有找到setKey，会按照 _ key， _ iskey，key，iskey的顺序搜索成员变量，设置成NO就不这样搜索。
- 如果传入的Value值为nil，则会调用setNilValueForKey方法；
- 如果key不存在且没有搜索到和key有关的字段和属性，则会调用valueForUndefinedKey函数。

  KVC详解

Category

category的最直接的作用就是给已有类添加方法。其次还可以拆分单个文件、便于多人同时开发一个类、声明私有方法等。

category是在runtime时决议的（所以category不能为类添加成员变量，因为运行时的类的内存布局已经确定了），category的结构体中包含名称、所属的类、方法列表、类方法列表、协议列表、属性列表。

category是在运行时方法objc_init添加进类当中的，根据运行时方法的查找顺序，所以如果一个类对应多个category，每个category对应着相同方法，则最后编译的category的方法应该是实际生效的。但是并不是所有的方法都会直接这样“覆盖”掉。如果每个类本身和category都实现了**+load**方法，则每个load方法都会执行,执行的顺序是先类后category。

- 扩展： 扩展可以说是一个匿名的分类，但是它又不是分类，因为它是编译时决议的，生命周期随之宿主类，只有声明，没有实现，依附在宿主类中。主要的作用就是用来声明私有属性、私有方法、私有成员变量。不能为系统类添加扩展。

Association关联对象

在category中提到，不能为已有类添加成员变量，因为运行时类的内存布局已经确定了。但是可以添加属性。但其实属性 = setter+ getter + ivar。依然绕不过成员变量这个坎。所以关联对象的出现，正好解决了这个问题。

在runtime源码中我们可以看到一个AssociationsManager单例，它负责维护一个静态变量AssociationHasMap。也就是说我们创建的每一个类的关联对象都放在这同一个容器中。这个map以key为键值，以其他三个参数组成的结构体为value进行赋值。这样正好解决了内存布局的问题。

- 注意：accosiation的移除问题，association的objc_removeAssociationObjects接口是移除某个对象的所有关联对象，如果要移除单个关联对象，则还是应该使用objc_setAssociationObjec传入nil来达到移除目的。

通知

通知是使用观察者模式实现的，主要用于跨层传递消息。可以一对多进行传递。 通知的底层实现应该和关联对象的实现方式类似。主要由通知中心NSNotificationCenter管理一个Map表。Map表的key值是notifIctionName，value是一个列表。列表中的每个元素就是观察者、参数、方法等参数。

注意：

- 在多线程应用中，Notification在哪个线程中post，就在哪个线程中被转发。所以尽可能使用block API进行处理。其次要多注意它产生的循环引用的问题
- 代理：使用代理模式实现的，使用一对一的传递方式！

关键名词解析

- **isa混写**：isa指针指向当前对象所对应的类，也可以说有isa指针的结构体就是一个对象，比如block和runloop。isa混写技术是指在运行时改变isa的指向，实现动态修改对象的类。具体是实现方案是通过给NSObject添加一个分类，分类里调用C函数objc_setClass来修过isa指向

```
//实现主体：
#import "NSObject+SetClass.h"
#import <objc/runtime.h>
@implementation NSObject(SetClass)
- (void)setClass:(Class)aclass {
   objct_setClass(self,aclass); //注意这里要确定两个类的size一样，通常需添加一个class_getInstanceSize的判断
}
@end

//使用：
A *a = [[A alloc]init];
[a setClass:[B class]];
```

- **+load VS +initialize**：
    - +load：在APP启动时，会在main函数之前加载所有的类，load函数就是在这个时候被调用的,每个类和分类都有load方法，并且都会被调用一遍。我们一般在这个函数里执行一些Runtime的方法；
    - +initialize：在消息查找的过程中，如果该消息是发送给该类的第一条消息，那么它将会调用该类的+initialize方法。所以基本就是在我们第一次使用该类的init之前被调用，类似于懒加载的效果。如果有多个初始化实例，那么这个函数也只会被调用1次。如果类和分类都实现了initialize方法，则只有最后编译的category的initialize方法会被调用。系统用它来初始化静态变量。

Runtime

我们知道OC是在C语言基础之上开发的。编译的时候，OC无法直接编译为汇编语言，而是需要先转成C语言再进行编译。而从OC到C的过度就是由runtime实现的。所以可以说runtime是OC的运行环境，它是OC在C的基础上实现面向对象和动态机制的基石。

Runtime 基础数据结构

- 先来分析一下runtime的基础数据结构，从数据结构中了解OC的部分运行时特性： OC中所有的对象都是id类型，id类型在runtime时被编译为objc_object结构体，objc_object包含了一个isa指针。OC中类对象Class在runtime中被编译成objc_class结构体，objc_class继承自objc_object。objc_class包含一个superClass指针，一个cache_t结构体，一个class_data_bits_t结构体。cache_t是一个可增量扩展的哈希表结构，它里面的每一个元素是bucket_t的结构体，bucket_t结构体里包含一个key和一个IMP。class_data_bits_t是对class_rw_t结构体的封装，class_rw_t里包含一个class_ro_t结构体、一个protocols二维数组、一个properties二维数组、一个methods二维数组。class_ro_t里包含一个name、一个ivars数组、一个protocols数组、一个properties数组、一个methods数组。methods数组里是一个method_t的结构体，method_t里包含一个SEL、IMP、const char*类型的字符串。
- 注意几个关键词：
    - isa指针是指向当前对象所属的类。它分为指针型isa和非指针型isa。指针型isa的所有数据都用于存储所属类的地址，非指针型的isa只有部分内容用来存储所属类的地址。我们可以说含有isa指针的结构体都是对象，比如说runloop、block等。其次，类也是一种对象，因为它继承自objc_object，称之为类对象。所以isa的指向就包括：对象的isa指向类对象，类对象的isa指针指向元类，元类的isa指针指向根元类，根元类的isa指针指向它本身
    - 类对象的superClass指针指向父类对象，元类对象的superClass指针指向元类的父类，根元类的superClass指针指向NSObject，NSObject的superClass指针为nil；
    - cache_t 是一个用于快速查找方法执行函数。它是一个可增量扩展的哈希表结构。是计算机局部性原理的应用；runtime通过key进行hash算法定位到对应的bucket_t，然后获取IMP。
- 几个重要的点：
    - 1、class_rw_t里的二维数组实际是对应category里的protocols、properties、methods。每一个category对应一个元素。所以方法查找时会先查找category里的方法；而class_ro_t是指只读的相关数据，也就是说已编译好的对象是无法在通过分类添加成员变量的。在class_rw_t的方法列表中查找时，对于已排到的使用二分法查找，对于没有排序的直接遍历查找。
    - 2、superClass指针的指向代表了方法查找的过程。也就是说对于实例方法，会沿着类对象的方法列表（此处先省略缓存查找等细节）进行查找，找不到就查找父类的方法列表，最后找的NSObject。而类方法的查找是在元类的方法列表中进行查找，以此往上查找，找的根元类的时候，由于根元类的superClass指针指向NSObject，所以又会去查找对应对象的实例方法。

Runtime涉及的事件

- objc_msgSend： objc_msgSend是使用汇编语言写的。因为纯C函数是无法携带未知参数并跳转至任意函数指针的，另外汇编也更加高效。所以**方法查找实际可分为高速查找和缓慢查找**，大致为：
    - 获取传入对象所属的类；
    - 获取该类的方法缓存列表；
    - 使用选择子在缓存中查找；

    - 如果缓存不存在，调用C语言的缓慢查找方法列表。
- 消息查找： runtime会把iOS所有的方法调用转成objc_msgSend函数调用，给receive发送一条selector。之后会进行iOS的方法查找过程：在当前类的cache中进行查找，如果没有就在当前方法列表中进行查找，如果没有，就会去父类中查找，同样先查找缓存列表，再查找方法列表。如此往复，一直到根类。
- 消息转发： 如果方法查找没有找到，则iOS提供了动态方法解析和消息转发流程： 动态方法解析：resolveInstanceMethod: (类方法是resolveClassMethod)。可以在这个方法里动态添加方法实现（使用class_addMethod），添加完成后，会重新走消息发送流程； 如果动态方法解析没有添加方法，则开始走消息转发流程：先查看forwardingTargetForSelector：如果返回不为nil则说明转发给了其他id对象，则会开始走那个id对象的消息转发流程；如果返回nil，则会调用methodSignatureForSelector:方法，如果返回值不为nil，则会调用forwardingInvocation方法执行。否则调用doseNotRecognaziedSelector方法报出异常。
- Method-Swizzing： 方法交换是指使用runtime API：method_exchangeImplementations(Method m1, Method m2)来交换两个方法的实现。 q：两个category同时发生了methodSwizzed会出现什么问题？ 应该会换回去？首先如果分类方法与原始方法同名，则会被分类覆盖掉，而且分类方法如果和原始方法同名，则也无法实现methodswizzling，因为拿到的Method是一样的，所以无法交换。所以这道题的意思只能是两个分类的方法名是一样的，原始方法名不一样，然后在不同的分类分别与原始方法进行methodSwizzling。所以题解：首先分类被运行时加载进来时肯定不可能同时加载，也就是交换方法不可能同时执行，肯定有个先后顺序。然后肯定都得放到load方法里才可能都执行到。所以也就相当于换了之后又被换回去了！所以应该是调用的方法不会发生改变！
- 其他：Category、KVO、KVC、ARC等

Runloop

RunLoop是通过系统内部维护的事件循环来对事件和消息进行管理的对象。事件循环可以有效地对消息和事件进行管理：当有消息需要处理时，会将进程从内核态到用户态进行切换，同时唤醒当前线程处理消息；当没有消息需要处理时，会休眠当前线程，将进程从用户态到内核态进行切换，以避免资源占用。所以runloop可以保证进程不退出，可以处理和监听事件，可以定时渲染UI，可以调节CPU的工作。

runloop基础数据结构：

CFRunLoopRef 里包含一个pthread、一个currentMode、一个modes、一个commonModes、一个commonModeItems。currentMode指当前runloop对应的mode，每次调用runloop的主函数时，只能指定一个mode，它是为了分割开同组的Observer、timer、source,让其互不影响。 currentMode和modes里的数据结构都是对应CFRunLoopMode,CFRunLoopMode对应5种mode：DefaultMode、UITrackingMode、UIInitializationMode、EventReceiveMode、CommonModes。commonModes对应的是字符串，也就是前面5种mode的名称，它是一种技术解决方案，modeItems对应4种item：Observer、timer、source0、source1。

- 注意：
    - RunLoop与线程是一一对应的关系。所有的runloop被存在一个全局的字典容器中。我们无法手动代码创建runloop，只能通过currentRunLoop获取当前的runloop，获取的过程中，以线程为key值进行查找，没有找到就会默认创建，所以子线程中，不手动调用[[NSRunloop current] run]，则该线程中的runloop就不会自动开启。

- 五种mode的区别：DefaultMode是APP默认的mode，通常主线程是在这个mode下运行；TrackingMode是UIScrollView的滑动mode；UIInitializationMode是app启动时的mode，启动完成后就不再使用；EventReceiveMode系统内部接收事件的mode，通常用不到；CommonMode则实际上不是一种模式，而是一种mode的组合方式。一个RunLoop包含多个Mode、每个Mode又包含多个Timer、Observer、Source；
- commonMode是一种技术解决方案。RunLoop启动时只能指定一个Mode作为当前的currentMode，如果要切换Mode，只能先退出当前Mode，重新选择一个Mode进入。所以为了解决一个RunLoop只能有一个Mode的情况，产生CommonMode的技术解决方案。一个mode一旦被标记为common属性的话，则该mode的名字就会被放到commonMode集合中，当runloop的mode发送变化时，runloop会将commonModeItems中的Observer、timer、source同步到具有Common属性的mode中。这样就可以让一些事件可以同时在多个mode中工作，比如NSTimer。
- 4种Mode Item：Observer、timer、source0、source1：Oberver主要是用来观察RunLoop自身的6大状态，Timer实际就是我们用的NSTimer，也就是说NSTimer是根据RunLoop的Timer事件实现的，所以它是不准的。source0用于接收用户事件，比如UIEvent。source1事件是系统内核使用的，它可以主动唤起线程。
- runloop循环状态：

  第一步：通知observers，RunLoop要进入loop了；（也就是发出**KCFRunLoopEntry**通知，将要进入事件循环。）

  第二步：进入事件循环后，会先通知Observer，RunLoop将要触发Timer回调和Sources回调，接着执行block；（其实是先判断是否有timer事件和source0事件，有的话会先发送**KCFRunLoopBeforeTimers**和**KCFRunLoopBeforeSources**，处理对应Timer回调和Sources回调，然后执行block。处理sources回调时如果有Source1是ready状态的话，则会先跳转到handle_msg去处理消息）

  第三步：回调触发后，则通知Observers，RunLoop将要进入休眠状态。（发出**CFRunLoopBeforeWaiting**通知）

  第四步：进入休眠后，会等待mach_port消息，以再次唤醒。（有四个事件能将线程唤醒：基于port的Source1事件、Timer时间到、RunLoop超时、被调用者唤醒）

  第五步：被唤醒后，发出**KCFRunLoopAfterWaiting**消息

  第六步：处理被唤醒的消息：

  第七步：根据当前RunLoop的状态来判断是否要走下一个Loop。如果是被外部强制停止或者loop超时，则不进入下一个loop，否则继续走下一个loop。（退出会发出**KCFRunLoopExit**通知）

  根据这七个步骤也就知道，主要处理事件的阶段是在KCFRunLoopBeforeSources到KCFRunLoopBeforeWaiting之间 和 KCFRunLoopAfterWaiting之后。所以如果KCFRunLoopBeforeSources状态和KCFRunLoopAfterWaiting状态所在的时间过长（超过了卡顿阈值），那么就可以认为有卡顿出现了。

  RunLoop启动时会发出CFRunLoopEntry通知；从休眠中唤醒时会发出CFRunLoopAfterWaiting通知；开启监听模式发出CFRunLoopBeforeObserve通知；然后发出CFRunLoopBeforeTimers、CFRunLoopBeforeSources通知进行优先处理timer和source0事件；当没有消息需要处理时，会从用户态切到内核态，进行休眠，此时发出CFRunLoopBeforeWaiting通知；最后runloop退出时会发出CFRunLoopExit通知。

- source0 和 source1的关系：source1事件是具备唤醒线程的能力，但是主要用于系统内核事件，source0事件则主要用于app应用层事件，不具备唤醒线程的能力。所以一个touch事件，最初是由系统点击屏幕捕获到的系统事件，然后通过mach_msg，在APP内核生成一个source1事件，之后source1唤醒了当前线程，然后将其包装成source0事件去处理。

RunLoop涉及的事件：

- CALayer或者说视图树中的视图如果层次、frame或者调用了setNeedsDisplay\setNeedsLayout，则CALayer就会被标记为待处理。等到VSync信号发出来的时候才会CPU才开始处理，VSync的信号周期也就是 runloop的刷新周期，也就是在每秒60帧的画面，16.7ms的周期；
- runloop使得main函数不会退出，因为UIApplicationMain内部开启了一个主线程的runloop；
- runloop可以提升CPU效率，使其有事的时候做事，没事的时候休息；
- NSTimer实际是根据runloop的CFRunLoopTimerRef来实现的，为了节省资源，默认情况下，NSTimer只能在DefaultMode下被回调，同时Timer有一个Tolerance (宽容度)标示了当前时间点可以有多少误差。，当屏幕滑动时，则就会暂停运行，所以需要通过commonMode的方案来解决。同理，这也导致系统默认的NSTimer不准。CADisplayLink则是一个和屏幕刷新率一致的定时器，但是它同样存在和NSTimer一样的问题。 所以要想要更精准的定时器，最好使用GCD或者Facebook 开源的 AsyncDisplayLink；

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

//GCD NSTimer
dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_source_t timer =
dispatch_source_creat(DISPATCH_SOURCE_TYPE_TIMER,0,0,queue);
dispatch_resumr(timer);
/* timer精度为0.1秒 */
dispatch_source_set_timer(timer,dispatch_timer(DISPATCH_TIME_NOW,1.0*NSE
C_PER_SEC),1.0 * NSEC_PER_SEC, 0.1 * NSEC_PER_SEC);
dispatch_source_set_event_handler(timer, ^{
//定时回调了
if (!repeats) {
    //取消心跳后，移除定时器
    dispatch_source_cancel(timer);
}
});
```

- AutoreleasePool的实现原理：AutoreleasePool及其包含的对象的内存管理是根据runloop的运行周期来确定的，具体查看内存管理章节。
- 事件响应：参考上面source0和source1的分析

- performSelector：当使用NSObject的performSelecter:afterDelay:或者performSelector:onThhread：（注意这里一定得是和时间或线程有关的API），实际会在其内部创建一个NSTimer并添加到当前线程的Runloop中。所以如果当前线程没有Runloop，则这个方法会失效，比如在新创建的子线程中，如果不调用[[NSRunLoop currentRunLoop] run]则就不会调用Select中的函数。
- GCD：GCD中的dispatch_async在使用到主线程的时候也使用到了RunLoop。首先GCD的所有线程仍然由libDispatch处理，但是当用到主线程的时候，libDispatch会向主线程的RunLoop发送消息，主线程的RunLoop会被唤醒，并从消息中获取block，然后回调执行。其他线程则仍由libDispatch处理。
- 网络请求框架中的常驻线程；

runloop详解

Block

Block是对函数及其执行上下文封装起来的对象。

block基础数据结构：

Block最终被编译之后得到的是一个结构体，结构体内部成员变量有一个 _ _ block _ impl的结构体，它内部包含一个isa指针，一个FunPtr函数指针。所以说block是一个对象，block调用就是函数调用。

block的类型：

block分为NSGlobalBlock、NSMallocBlock、NSStackBlock，它们分别存在在全局数据区、堆区、栈区。并且在**ARC和MRC中有所区别**：

- 首先无论是ARC和MRC下，只要是没有引用外部变量的block都是NSGlobalBlock，全局block对copy、retain、release没有反应；
- 其次引用了外部变量的block就变成了栈block。因为一旦使用到了外部变量，就可能因为截获变量而得管理相关内存，此时对栈block进行copy操作，那么就会拷贝到堆区；
- 栈block变为堆block，存在一些条件：在ARC下：（调用Block的copy方法、将block赋值给 _ _ strong修饰的类型或成员变量、block作为返回值时、使用GCD的block等）；在MRC下，（对栈block进行copy操作等）；
- 注意： 这里用代码解释一下block在ARC和MRC的区别：

```
void(^block) = ^{};    //没有对外部变量进行任何操作，所以是全局block
int val = 0;
NSLog("%@",^{val = 1;}); //引用了外部变量，所以这个匿名block是一个栈block，如果这是对其使用[^{val = 1;} copy],则就会变成一个堆block；
void (^block1) = ^{val=1;}; //注意，在MRC下，block1是一个栈block，因为它引用了外部变量。但是在ARC下，它是一个堆block，因为 = 左侧block1默认是 __ strong类型，相当于上文说的“将block赋值给__strong修饰的类型或成员变量”，所以在ARC下，大部分block其实都是堆block。
```

所以 在MRC中，block属性只能使用copy关键字，在ARC中，block属性可以使用Strong关键字和copy关键字。在实际开发过程中，为了使代码在ARC和MRC中都能使用，所以我们一般推荐使用copy关键字。

截获变量：

block最常见的使用是异步回调的，也就是说不是顺序执行的，所以就涉及到使用了外部变量的情况下，如果外部变量修改了，会不会影响到block里的使用的问题。block对变量的截获主要分为以下几种情况：

- block对于全局变量和静态变量是不进行截获的，因为它的作用域足够广，所以可以直接使用；
- block对于局部的基础数据类型，直接截获其值。也就是说后续的修改跟它没什么关系；
- block对于局部的对象类型，则是连同其所有权修饰符一起截获；所以这就导致了循环引用的可能性。

block循环引用：

由于block对局部对象类型的截获变量特性，所以就导致了可能存在的循环引用问题。如果对象类型是 _ _ strong类型的，那么block对其进行截获是会连同所有权修饰符一起截获，也就是说在block内部也是使用了一个 _ _ strong类型的变量强引用了它。所以这就导致了循环引用的问题。所以最好的解决方案就是对截获的变量使用 _ _ weak 的变量去替代它，其次常用的解决循环引用的方法就是断环。

- 处理NSTimer导致的循环引用问题 例如：当前类强引用了timer，timer又强引用了target。所以一般我们都考虑在当前类释放的时候， [_ timer invalidate]; _ timer = nil; ，使用断环的方式处理循环引用。但是如果不太容易知晓在何处使用invalidate更好的情况下，怎么解除循环引用是一个难点。

```
_ timer = [NSTimer scheduledTimerWithTimeInterval:1.0f target:self
selector:@selector(fire) userInfo:nil repeats:YES];
-(void)fire{
    NSLog(@"fire");
}
```

解决方案：借助一个虚基类NSProxy,弱引用target

```
@interface FCFProxy : NSProxy
@property (nonatomic,weak) id target;
@end
@implementation FCFProxy

//获取当前的方法签名
- (NSMethodSignature * ) methodSignatureForSelector:(SEL)sel
{
    return [self.target methodSignatureForSelector:sel];
}
//指定当前消息的处理者
-(void)forwardInvocation:(NSInvocation * )invocation
{
    [invocation invokeWithTarget:self.target];
}
//执行
_ hzProxy = [FCFProxy alloc];
```

```
//当前Proxy的target设为当前的self，因为真正要处理消息的其实是当前的viewcontroller（其实这个target就相当于delegate）
_ hzProxy.target  = self;
_ timer = [NSTimer scheduledTimerWithTimeInterval:1.0f target: _ hzProxy selector:@selector(fire) userInfo:nil repeats:YES];
当前 _ timer的对象的处理就变成了 _ hzProxy。
```

--block：

什么是 _ _ block? 添加了 _ _ block修饰符的变量最终会变成一个multiple的结构体，结构体中含有isa指针。也就是说 使用了 _ _ block修饰的变量最终变成了对象。但是在**ARC和MRC下是有区别**的。在MRC下使用 _ _ block修饰的变量，不会增加对象的引用计数；在ARC下，使用 _ _ block修饰的变量，会增加其引用计数。所以就有一道痕经典的面试题:

```
{
    __block id * blockSelf = self;
    _block = ^int(int num){
        return num * blockSelf.var;
    };
    _block(3);
}
//分析：由于在MRC下，__block修饰的变量不会增加引用计数，所以也就不存在强引用的问题，所以这段代码在MRC下是不会有问题的。但是在ARC下，它是存在循环引用的。所以可以使用断环的方式进行处理：
{
    __block id * blockSelf = self;
    _block = ^int(int num){
        int result = num * blockSelf.var;
        blockSelf = nil;
        return result;
    };
    _block(3);
}
```

一般我们在block对成员进行复制的时候要是有__block。

block的copy操作：

通过上面的分析，我们知道对全局block进行copy不会有任务反应，对栈block进行copy则会被拷贝到堆上，对堆block进行copy操作则会增加其引用计数。那实际上，对栈block进行copy操作时，在堆上也产生了一个一样的block，同时也使栈block的 _ _ forwarding指向了堆区的block，所以此时对栈block进行操作实际就是通过_ _forwarding指针找到对应的堆block，然后再进行操作。同时，在MRC下，对栈block进行copy操作就如同执行了一个malloc方法，所以同样存在内存泄漏的问题。

内存管理

内存管理方案

- **TaggedPointer**：对于一些小对象，比如NSNumber和NSDate。它实际上采用的是TaggedPointer的管理方式进行内存管理的，它是对象指针实际上不执行任何内存地址，而是直接将值存储在了指针本身里，该指针就被拆分成两部分，一部分是直接保存数据，另一部分是作为特殊标记。
- **NONPOINTER_ISA**：在runtime章节中有提到isa是指向当前对象所属类的一个指针。一个结构体含有isa指针，那可以说明它是一个对象类型。实际上isa是对应一个isa_t的联合体，在64位架构下，它是由64个0和1的比特位组成。而实际上存储该对象所属类对象地址只需要用到大概33位，所以剩下的位数就可以用来存储其他的信息。 我们来看看NONPOINTER_ISA的基础数据结构：
    - 第1位：是一个标志位，表示了它是否是一个非指针类型的isa，如果是0则说明它只是一个指针，内部的数据都是对应class的地址，如果是1则表示它是一个非指针类型的isa，内部数据只有部分用于存储class的地址；
    - 第2位：表示当前对象是否有管理对象——Association；
    - 第3位：表示是否使用ARC进行内存管理；
    - 第4~33位：表示当前对象所指向的类对象的地址；
    - 第34~40位：表示是否完成初始化；
    - 第41位：表示是否有弱引用指针；
    - 第42位：是否正在执行deallocating操作；
    - 第43位：表示当前对象是否外挂散列表；
    - 剩余的位数：都是表示extra_rc，但是注意，存储的是**真实的引用计数值-1**。
- **散列表：SideTables**：也就是在非指针型的isa的引用计数值太大时，则会使用到散列表来对内存进行管理。散列表中包含了引用计数表和弱引用表。

散列表详解

散列表基础数据结构：

在64位架构下，通常会有64个散列表组成一个全局的SideTables，它类似AssociationMap。里面每个SideTable包含一个自旋锁、一个引用计数表、一个弱引用表。当需要对引用计数进行操作的时候，则通过对象指针作为key值，经过哈希查找定位到对应的散列表，然后进行操作。

- **自旋锁Spinlock_t**： 它是一个“忙等”的锁，也就是说在当前资源被某个线程占用的时候，其他的线程会一直试探该锁有没有解锁，而像信号量，则是会在获取不到资源的时候进行休眠，等资源被释放后，则再被唤醒。它使用与轻量访问。这里之所以要有多个sidetable组成一个sidetables，也是为了提高访问效率。因为每个引用计数的操作都是需要加锁处理，分割成多个表，就可以并行操作。实际上这就是**分离锁**的技术方案；；
- **引用计数表**：引用计数表也是一个hash表，它是通过hash函数插入和获取引用计数，提高访问效率。hash表里的每个元素是一个unsigned long类型的size_t。它也是由64位比特位组成，其中第一位是表示是否有弱引用，第二位表示是否正在执行dealloc函数。剩下的则也是表示**真实的引用计数值-1**。所以取引用计数的时候需要进行向右偏移2位的操作；
- **弱引用表**：实际上也是一个hash表，它存储的是一个weak_entry_t的结构体数组，它里面的每个对象存储的都是一个弱引用指针。作用就是在对象执行dealloc操作的时候将所有指向该对象的weak指针的值设为nil，避免垂悬指针。

ARC & MRC

MRC是手动内存管理，ARC是自动内存管理，它通过LLVM编译器和runtime协作在合适的时机给代码添加retain和release等代码，实现自动引用计数的管理。ARC无法显式调用retain和release等函数。

引用计数值相关：

在RAC中可以使用runtime的objc_retainCount(id objc)来获取引用计数。它内部是通过获取到NONPointISA或者散列表里的引用计数值，**并在此基础上进行+1**后将结果返回，所以这也是为什么真正存储引用计数的地方的值都是**真实的引用计数值-1**的原因。 但是实际上不能完全信任objc_retainCount返回的引用计数值，比如说clang会尽可能把NSString实现成单例对象，所以其引用计数会非常大，TaggedPointer对象里的引用计数可能不太准确。

- alloc、new、copy、mutableCopy： 这四个方法都会使引用计数+1。其内部流程都会调用retain方法。
- retain：新经过hash查找找到对应的sidetable，然后在经过hash查找找到对应的引用计数值，之后获取到引用计数值，进行+1操作； q1：我们在进行retain操作的时候，系统是怎么查找其对于的引用计数的呢？ 是经过两次hash查找，然后进行+1操作。
- release： 同样是经过两次hash算法，获取到存储的引用计数值，然后先判断引用计数是否等于0，如果等于0就将对象标记为正在析构，并发送dealloc消息，返回YES；否则进行-1操作。这样就避免出现负数。以下问题：为什么release只后打印的引用计数还是1呢?

```
Person * p = [Person new];
NSLog(@"%d",[p retainCount]);  //1
[p release];
NSLog(@"%d",[p retainCount]); //1
```

原因就是在执行最后一次release操作的时候，真实存储的引用计数值已经是0了，这个时候对象会被标记为正在析构，并执行deallocing，**且并不会对引用计数值进行减一操作**，而retainCount的值是取真实的引用计数值+1，所以就可能存在返回还是1的情况。

- **dealloc函数及 _ _ weak指针**: 执行dealloc时，判断是否可以释放的条件包括：是否使用nonpointer_isa、是否有弱引用指针、是否有关联对象、是否使用ARC、是否使用sidetable。如果这些条件都为NO的时候，就可以直接使用c函数free直接释放，否则则要调用objc_dispose()进行进一步清理。 而objc_dispose则会一步步 移除对关联对象、将指向该对象的弱引用指针置为nil，将当前对象在引用计数表的数据清除掉等操作。(这里解决了两个面试题:1、对象在释放的时候，是否有必要移除掉关联对象；2、weak修饰的对象是怎么讲指针置为nil的。答案就是在dealloc内部实现的时候有做这些操作)

被 _ weak修饰过的对象

q1：系统是怎样把一个weak变量添加到它对应的弱引用表中的。 一个被声明为 _ weak 的对象指针，经过编译器编译，会调用一个objc_initWeak函数，之后会调用weak_register_no_lock()函数进行弱引用变量的添加，具体添加的位置是通过hash算法进行位置查找的，如果查找的对应位置中已经有了当前对象对应的弱引用数组，则把当前变量添加进弱引用数组，如果没有，则重新创建一个弱引用数组。
q2：当一个对象被释放之后，weak变量是怎么被清理的。 会被置为nil。当对象执行dealloc的时候会调用弱引用清除相关函数，在函数内部会通过弱引用指针找到弱引用数组，然后遍历所有的弱引用指针，分别置为nil。

AutoreleasePool

AutoreleasePool是用于对延迟释放的对象的内存管理。

- 首先来看一下AutoreleasePool的内部结构: AutoreleasePool是一个以AutoreleasePoolPage为节点的双向链表，同时它含有一个pthread的成员。AutoreleasePoolPage是一个栈结构，它里面有一个next指针。在ARC中的实现流程是：当@autoreleasepool{}中，{开始的位置会被编译成AutoreleasePoolPagePush(),它会在当前page栈中插入一个标志位，并且返回。在执行{}之间的代码时，会把当前生成的对象依从插入到next指针的位置，next指针在栈中依从往栈顶走，当栈满了的时候，就会生成下一个page，在}结束前，会把标志位做为入参执行AutoreleasePoolPagePop(xx),这个时候就会给标志位与next指针之间的对象依次发送release消息，然后复位next指针位置。
- 注意： 因为AutoreleasePool的数据结构实际是以标志位插入的方式来实现的，所以无论有多少层嵌套，都只是对标志位进行操作而已。
- AutoreleasePool与runloop的关系及内存管理： AutoreleasePool与线程是一一对应的关系，那么它与runloop自然也是一一对应的关系。在**主线程的runloop**发出CFRunLoopEntry通知的时候，会执行一个AutoreleasePagePush()函数，创建一个AutoreleasePool，然后runloop的事件循环中（因为runloop是在KCFRunLoopBeforObserver后才开启事件循环的），如果监听到了kCFRunLoopBeforeWaiting通知，会先执行AutoreleasePagePop()然后再执行AutoreleasePagePush()，相当于释放旧的AutoreleasePool，创建新的AutoreleasePool。然后在runloop退出前，即接收到KCFRunLoopExit通知时，再执行AutoreleasePagePop()释放掉AutoreleasePool。所以从这里可以看出，在Runloop的每一次事件循环期间，也是会对标志位直接的每一个对象发送一次release的，以及时释放掉无用的对象。所以主线程的几乎所有的对象都被AutoreleasePool环绕。
- 什么样的对象会自动加入到AutoreleasePool中？ 虽然说main函数里有@AutoreleasePool，但是并不是说所有对象都会交给Autoreleasepool来处理。
    - 使用类方法生成的对象（或者说作为方法的返回值的对象），会被自动注册到Autoreleasepool中。比如：id obj = [NSMutableArray array];
    - id指针或者对象指针没有显示指定时会被附加上 **autoreleastring修饰符。比如传参过程中的（NSError **）err 就等同于 <#NSError * _Nullable** autoreleasing * Nullable#>。
- **子线程默认不开启RunLoop，那出现Autorelease对象如何处理？会不会出现内存泄漏？** 在子线程中，如果你创建了Pool的话，产生的Autorelease对象就会交给Pool去管理，如果你没有创建Pool，产生了Autorelease对象，就会调用autoreleaseNoPage方法。在这个方法中，会帮你自动创建一个hotPage(hotPage可以理解为当前正在使用的AutoreleasePoolPage)。然后调用page->add(obj)将对象添加到当前正在使用的AutoreleasePoolPage的栈中，所以这也不会导致内存泄漏。
- main函数的AutoreleasePool有什么作用，可否直接去去掉？ 实际上UIApplicationMain是永远不会返回的，除了系统kill应用，系统会把整个应用占用的内存释放掉。因为UIApplicationMain永远不会返回，所以这里的AutoreleasePool就没有执行pop的机会，所以实际上如果删掉应该也不会有内存泄漏的问题。
- 自动释放池在MRC和ARC中的区别： 如果函数返回值是对象的话，MRC则需要区分调用者是否拥有这个返回值，如果调用者拥有中国返回值，则调用者就要对其进行释放，如果没有就不需要。比如说：

分析：调用者直接拥有返回值。对象初始化时引用计数为1，property是retain的话，赋值的时候，引用计数会加1。 所以即使dealloc的时候，执行[_ property release]，也会存在内存泄漏。正确的做法是：self.property = [[[NSObject alloc] init] autorelease];

```
self.property = [[NSObject alloc] init];

NSObject * a = [[NSObject alloc] init];
self.property = a;
[a release];
```

这里因为执行了一次临时变量的release，所以就不会出现内存泄漏问题。

在ARC下，则无需考虑这两个返回值的区别。Runtime有对Autorelease返回值进行优化。在ARC下，编译器会在返回值使用objc_autoreleaseReturnValue()代替调用autorelease，使用的时候会使用objc_retainAutoreleaseAReturnValue()代替调用retain，最后释放的时候会使用objc_storeStrong()代替release。比如

```
+ (instancetype)createSark {
    return [self new];
}
// caller
Sark * sark = [Sark createSark];
//runtime中变成：
+ (instancetype)createSark {
    id tmp = [self new];
    return objc_autoreleaseReturnValue(tmp); // 代替我们调用autorelease
}
// caller
id tmp = objc_retainAutoreleasedReturnValue([Sark createSark]) // 代替我们调用retain
Sark * sark = tmp;
objc _ storeStrong(&sark, nil); // 相当于代替我们调用了release
```

其次使用了objc_autoreleaseReturnValue方法时，runtime会将返回值存储到TLS(线程局部存储器)中，objc_retainAutoreleasedReturnValue则直接去TLS中取出对象。将TLS作为一个中转站来使用了。

- autorelease 对象赋值给weak对象，是否会立即释放？ 不会，这种情况会通过objc_loadweak把对象注册到AutoreleasePool中，以延长生命周期。比如：

```
NSNumber  _  weak * number = [NSNumber numberWithInt:100];
NSLog(@"number = %@", number); //这里是可以打印出来的
```

锁与多线程

多线程是为了实现并发执行的技术。在单核CPU中，操作系统通过分配CPU计算时间来实现软件层面的多线程，在多核CPU中，则直接可以在硬件层面进行多线程运行。 iOS有多种多线程的方案，pthread、NSThread、performSelect、GCD、NSOperation。

NSThread和performSelector

- NSThread先创建线程再启动线程，NSThread常用于开启常驻线程

```
//1 创建线程
NSThread * thread = [[NSThread alloc] initWithTarget:self
selector:@selector(xxx) objc:nil];
//手动启动
[thread start];
// 2 创建线程 自动启动
[NSThread detachNewThreadSelector:@selector(xxx) toTarget:self
withObject:nil];
```

- performSelector 含有thread的API可以隐式开启线程。但是其含有delay的API则是使用了线程中对应runloop的Timer，需要开启NSRunLoop才能运行

```
[self performSelectorInBackground@selector(xx) withObjct:nil];
```

GCD

GCD是一个多核并行运算的解决方案，它可以自动管理线程的生命周期，只需要告诉GCD干什么就行。

队列和任务:

- 队列分为**串行**和**并发**： 串行派发是指同一时间一次只能执行一个任务，只有在当前任务执行完成后，才会派发新的任务；并发是指多个任务可以在同一时间同时进行，无需等待前面的任务执行完也可以派发新任务；iOS中主队列mian是一种串行队列，全局队列global是一种并发队列；
- 任务分为**同步**执行和**异步**执行： 同步执行是在当前线程中执行，异步执行会在线程池中获取一个线程进行执行，如果线程池中没有则会创建一个新的线程执行；

4种线程组合：

- 同步串行：同步不开启新线程，串行队列让任务一个一个执行。所以它实际不会产生多线程运算；
- 同步并发：并发队列虽然可以并发执行，但是同步执行不开启线程，也就相当于任务只在一个线程中执行，而一个线程一次只能处理一个事件，所以实际也不构成多线程运算；
- 异步串行：异步执行虽然会开线程，但是串行队列中任务一个个执行，所以也不构成多线程运算；
- 异步并发：并发队列可以让任务并发执行，异步执行可以开启多个线程，所以这才会真正实现多线程运算。

```
//eg 01
serialQueue.async{
    serialQueue.sync{
    }
}
// 分析：首先这是在一个串行队列里执行，所以其执行方式必然是一个一个执行。最外层使用异步执行，所以会开辟一个线程；内部的任务是同步执行，所以是在当前线程中执行。因为是串行执行，所以内部任务是在外部任务执行完成后才开始执行，而外部任务的执行完成依赖于内部任务也执行完成。所以就导致了死锁!
SerialQueue.sync{
    SerialQueue.async{
    }
}
//分析：同上面的分析类似。使用串行队列，最外层任务是在当前线程中执行，内部任务会开辟新的线程执行。外部任务的执行完成意味着内部任务也执行完成，而内部任务可以在另外的线程中执行完成，所以这个是可以正常运行的!
```

全局队列优先级

创建全局队列时，有一个参数是优先级的标识，所以如果要在并行队列中，让任务先执行，这可以通过设置这个优先级来达到目的（但是这里要注意，先执行并不一定是第一个执行完，它只能保证开始的执行顺序而已）。它的优先级包括（低-高：background（同步备份数据）、utility（需要时间的下载）、default、user-Initiated(用户出发的，如打开文件)、user-Interractive(用户交互，如主线程事件)）

dispatch_barrier

dispatch_barrier是指栅栏调用，它的特点是无论是同步还是异步，都会阻塞当前线程，也就是说它会等待所有之前入队的任务都执行完成才开始执行，而在之后入队的所有任务会等待dispatch_barrier本身任务执行完成再执行。

- dispatch_barrier_async与dispatch_barrier_sync的执行效果是一样的，区别就在于：sync会阻塞后续线程入队，async不会阻塞入队。但是执行时候的效果是一样的。例如，如果主线程中有打印，则sync会阻塞主线程中的打印，而async不会。
- dispatch_barrier_async和dispatch_barrier_sync只有在自定义的并发队列中才能有栅栏效果，否则它与dispatch_async或dispatch_sync的作用是一样的，比如串行队列或者系统的全局队列都是无效的。所以它在串行队列中同样容易导致死锁。
- dispatch_barrier实现多读单写

```
A{
    set{
        ConcurentQueue.async(flogs:.barrier){
            _a = newvalue
        }
    }
    get{
    //注意这里使用同步读取
        ConcurentQueue.sync{
            return _a
```

```
        }
    }

}
```

dispatch_group调度组

dispatch_group可以实现先并发处理一些任务，然后监听所有任务的执行完成。
dispatch_group_async可以异步添加任务到group中，如果无需涉及线程或者队列的话，则可以使用dispatch_group_enter和dispatch_group_leave**配对**执行，比如接口任务。最后使用dispatch_group_notify通知group所有任务已完成。也就是说可以单独使用dispatcjh_group_async往group添加任务，等执行完成同样会调用notify，比如自己写的请求。那对于已经有队列、异步线程的，则可以使用enter和leave，比如使用第三方请求的时候。

```
{
//注意点：dispatch_group_async与dispatch_group_enter效果是一样的，都是异步添加任务，dispatch_group_enter与dispatch_group_leave必须成对出现，否则group中的任务永远不会完成。
    let group = DispatchGroup()
    DispatchQueue.global().async(group: group, qos: DispatchQoS.default, flags: []) {
      sleep(1000)
      print("任务1")
   }
   DispatchQueue.global().async(group: group, qos: DispatchQoS.default, flags: []) {
      sleep(1000)
      print("任务11")
   }

    dispatch_group_enter(group)
    self.request1({
      sleep(1000)
      print("任务2")
      dispatch_group_leave(group)
    })

    dispatch_group_enter(group)
    self.request2({
      sleep(1000)
      print("任务3")
      dispatch_group_leave(group)
    })

    dispatch_group_notify(group,dispatch_get_main_queue(),^{
        print(finish)
    })
}
```

dispatch_semaphore 信号量

dispatch_semaphore俗称信号量，也叫信号锁，用于控制多线程下资源访问的数量。 三个方法：

- dispatch_semaphore_create: 创建一个带有初始值的信号量dispatch_semaphore_t；
- dispatch_semaphore_wait: 这个方法先查看信号量是否是小于等于0，如果小于等于0，则阻塞，一直等待直到时间结束，当信号量大于0 的时候，则会先将信号量进行减1，然后放行执行后续操作。
- dispatch_semaphore_single: 这个方法用于让信号值加1，然后直接返回，如果先前信号量的值小于0，那么这个方法会唤醒先前等待的线程；

```
//1线程同步
let semaphonre = DispatchSemaphore(value: 0)
self.request2({
        print("执行耗时任务，比如接口请求")
        sleep(1000)
        semaphore.signal()
})
print("开始等待")
semaphore.wait(timeout: DispatchTime.distantFuture)
print("任务结束")

//2资源加锁
//创建一个线程
let semaphonre = DispatchSemaphore(value: 1)
static int tickets = 100; //100张票
var i = 0
while(i<3){
        //开启了3个线程
    DispatchQueue.global.async({
        //开启并发，每个线程类似一个售票窗口
        semaphore.wait(timeout: DispatchTime.distantFuture) //-1
        sleep(1000)
        tickets -= 1 //卖票
        semaphore.signal()
    })
    i += 1
}

//3改变全局队列里设置好的优先级。
var highQueue = Dispatch.global(qos:.userInitiated)
var lowQueue = Dispatch.global(qos:.utility)

let semaphore = DispatchSemaphore(value:1)
lowQueue.asycn{
    semaphore.wait()
    sleep(1000)
    semaphore,signal()
}
```

```
highQueue.asycn{
    semaphore.wait()
    sleep(1000)
    semaphore,signal()
}
//这里lowQueue的优先级更高。
```

注意： 在信号量使用过程中对信号量进行重新赋值或者置空操作会crash。在libdispatch源码中，使用一个变量dsema_value保存当前信号量的值，使用dsema_orig保存初始值。执行过程中，当使用wait时会对dsema_value减1，使用signal的时候对dsema_value加1。当我们对信号量进行置空或重新赋值的时候，会调用disponse释放信号量，在disponse函数中如果dsema_value小于dsema_orig，则会直接调用CRASH使程序崩溃。 dispatch_semaphore

dispatch_semaphore使用

Operation：

NSOperation是基于GCD更高一层的封装。它是基于两个概念**操作**和**队列**来实现多线程的。

- **操作**：
    - 操作其实就是对应着GCD的block。但是NSOperation的操作更复杂一些，它可以管理自己的状态和优先级。NSOperation使用其子类来定义操作:NSInvocationOperation、NSBlockOperation、自定义子类。
    - 每个操作都对应着4个状态：isReady(是否就绪)、isExecuting(是否进行中)、isCancelled(是否取消)、isFinished(是否完成)。
    - 单独的操作只在当前线程中执行。
- **队列**：
    - NSOperation的队列有区别于GCD的先进先出的调度队列。NSOperationQueue是根据NSOperation的依赖关系来决定将队列中的哪个Operation置成就绪状态，然后进入就绪状态的Operation的**开始执行顺序**取决于操作之间的优先级；
    - OperationQueue实现了暂停、继续、终止、优先顺序、依赖等操作。
    - NSOperationQueue只提供了两种不同的队列：主队列和自定义队列；主队列运行在主线程之上，自定义队列运行在后台。
    - NSOperationQueue通过设置最大并发量maxConcurentOperationCount来确定其实串行还是并发。
- 如何实现多线程
- 单独的NSOperation使用同步运行，把它放到NSOperationQueue中，然后设置最大并发量大于1就可以实现多线程的运行效果；

NSInvocationOperation

单独使用NSInvocationOperation是在当前线程中执行，并不开启线程。将其放到其他线程下面，才会开启新线程

```
//test函数省略， 单独这样使用的话，test就在当前线程中运行，和多线程没有任何关系。也可以将其放到任一一个子线程中运行。
- (void)useInvocationOperation{
        //创建对象
    NSInvocationOperation *iop = [[NSInvocationOperation alloc] initWithTarget:self selector:@selector(test) object:nil];
    //手动开启
    [iop start];
}
```

NSBlockOperation

同样的，**单独的**的NSBlockOperation也是在当前线程中运行，并不会开启子线程。

```
- (void)useBlockOperation{
        //创建对象
    NSBlockOperation *bop = [NSBlockOperation blockOperationWithBlock:^{
                //模拟耗时操作
                [NSThread sleepForTimeInterval:20];
    }];
    //手动开启
    [bop start];
}
```

但是，NSBlockOperation还提供了addExcutionBlock。通过addExcutionBlock可以为NSBlockOperation添加额外的操作。而这些操作(包括NSBlockOperation本身)可以在不同的线程中同步执行，具体哪些线程在其他线程中执行以及开多少条子线程则由系统决定。

```
- (void)useBlockOperationWithAddExcution {
        //创建Block对象
    NSBlockOperation *bop = [NSBlockOperation blockOperationWithBlock:^{
                //模拟耗时操作
                [NSThread sleepForTimeInterval:20];
                NSLog(@"2---%@", [NSThread currentThread]);
    }];
    //添加额外操作1
    [bop addExcutionBlock:^{
        //模拟耗时操作
                [NSThread sleepForTimeInterval:20];
                NSLog(@"2---%@", [NSThread currentThread]);
    }];
    //添加额外操作2,可以添加n个额外操作
    [bop addExcutionBlock:^{
        //模拟耗时操作
                [NSThread sleepForTimeInterval:20];
                NSLog(@"2---%@", [NSThread currentThread]);
    }];
```

```
    //手动开启
    [bop start];
}
```

自定义Operation

如果想自己控制操作的状态，则可以自定义Operation，自定义的Operation又分为非并行的Operation和并行Operation：

- 非并行Operation： 非并行的Operation只需要实现main函数即可，将操作代码放到main函数里：

```
@interface CustomSyncOpretion:NSOperation
@end
@implementation CustomSyncOpretion
- (void)main{
        if ([self isCancelled] == NO) {
        [Thread sleepForTimeInterval:20];
        }
}
@end

//使用
CustomSyncOpretion *cop = [[CustomSyncOpretion alloc] init];
//调用 start 方法开始执行操作
[cop start];
```

- 并行Operation子类 要实现并发的子类则要复杂很多，因为要自己管理操作的状态，所以必须重写以下方法：
    - start：所有并行的Operation必须重写这个方法，然后在你想要执行的线程中手动调用这个方法。注意：不要调用其父类的start方法；
    - isExecuting：是否执行中，需要使用KVO实现；
    - isFinished：是否完成，需要使用KVO实现；
    - isAsynchronous：该方法返回NO，表示非并发执行。并发执行需要自己定义并返回YES，其后的操作根据这个值来决定是否手动开启子线程；

```
@interface CustomAsyncOperation : NSOperation
@end

@interface CustomAsyncOperation(){
    BOOL excuting; //执行中
    BOOL finished; //已完成
}
@end

@implementation CustomAsyncOperation
- (instanceType)init{
```

```
    self = [super init];
    if (self){
        excuting = NO;
        finished = NO;
    }
    return self;
}

- (BOOL)isAsynchronous {
    return YES;
}
- (BOOL)isExecuting {
    return excuting;
}
- (BOOL)isFinished {
    return finished;
}

- (void)start {
        @autoreleasepool{
        if (self.cancelled) {
            [self willChangeValueForKey:@"isFinished"];
                finished = YES;
                [self didChangeValueForKey:@"isFinished"];
                return;
        }

        // 任务。。。模拟耗时操作
                [NSThread sleepForTimeInterval:20];
                NSLog(@"2---%@", [NSThread currentThread]);
                [self completeOperation];
    }
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}

@end

//使用
CustomAsyncOperation *cop2 = [[CustomAsyncOperation alloc] init];
```

```
//调用 start 方法开始执行操作
[cop start];
```

队列

NSOperationQueue中一共有两种队列：主队列、自定义队列。自定义队列同时包含了串行和并发功能。

- 主队列： 凡是放到主队列中的操作，都会放到主线程中执行。(注意：这里不包括addExecutionBlock添加的额外操作，额外操作可能在其他线程中执行)

```
//获取主队列
NSOperationQueue * queue = [NSOperationQueue mainQueue];
```

- 自定义队列： 自定义这种队列的操作，会自动放到子线程中执行。

```
//注意，这里虽然用了上述的操作，但是在Queue里使用就无需手动start了。
NSOperationQueue * cusQueue = [NSOperationQueue new];
[cusQueue addOperation:iop];
[cusQueue addOperation:bop];
[cusQueue addOperation:cop1];
[cusQueue addOperation:cop2];

//直接使用block添加操作
[cusQueue addOperationWithBlock:^{
        //模拟耗时操作
        [NSThread sleepForTimeInterval:20];
}];
```

队列控制串行并发 队列使用属性**macConcurrentOperationCount**来控制串行和并发。注意：macConcurrentOperationCount并不表示并发线程的数量，而是一个队列中同时能并发执行的最大操作数。且一个操作并非只在一个线程下完成。macConcurrentOperationCount默认值是-1，表示不进行限制，可进行并发执行，如上代码；值为1时，队列为串行执行；值大于1时，队列并发操作，当然了，最大并发数肯定是无法超过系统限制的。

NSOperation操作依赖

NSOperation的操作依赖用来控制添加到队列中的操作进入准备就绪状态。

- addDependency:(NSOperation *) op; 添加依赖，使当前操作依赖于操作 op 的完成。
- removeDependency:(NSOperation *) op; 移除依赖，取消当前操作对op的依赖。
- @property (readonly, copy) NSArray<NSOperation *> *dependencies；在当前对象开始执行之前，完成数组里的所有操作对象。 **注意**：操作直接如果相互依赖会导致死锁。

NSOperation 优先级

NSOperation的优先级queuePriority属性，进入就绪状态的操作的**开始执行顺序**则是通过优先级属性决定的。 queuePriority包含五种取值：NSOperationQueuePriorityVeryLow、NSOperationQueuePriorityLow、NSOperationQueuePriorityNormal、NSOperationQueuePriorityHigh、NSOperationQueuePriorityVeryHigh。 优先级的属性只适用于同

一队列中的操作。

总结：**依赖关系**决定了准备就绪状态，**优先级**决定了开始执行顺序：当一个操作的所有依赖已经完成时，操作会进入准备就绪状态。比如op1、op2、op3、op4四个操作。op3 dependency op2、op2 dependency op1。现在4个操作都添加到队列里，op1和op4都没有依赖，所以都是准备就绪状态，op2要等到op1执行完成后才进入准备就绪状态，op3要等到op2执行完成后才进入准备就绪状态。而同时进入准备就绪状态的操作的开始执行顺序则是由priority决定的。

Operation的暂停和取消操作

操作的暂停和取消操作并不代表里面就能将当前操作进行暂停和取消，而是当当前操作执行完毕之后不再执行新的操作。暂停和取消的区别则在于：暂停操作之后还可以恢复操作，继续向下执行；而取消操作就会清空索引的操作，后续无法继续。

NSOperation基础 自定义Operation 自定义Operation2

GCD VS NSOperation：

GCD是基于C语言实现的，NSOperation是基于GCD实现的！ NSOperation可以添加依赖、优先级、最大并发量，操作可以控制状态，执行暂停和取消等操作。 GCD则有栅栏、group、信号量、单例、延时执行等。在暂停、取消、最大并发量这些操作更不容易实现。

锁

锁主要可以分为三大类：**信号量**、**互斥锁**、**自旋锁**：

- 信号量查看GCD相关知识点；
- phread_mutex是互斥锁，互斥锁在访问被锁的资源时，调用者线程会休眠，此时CPU可以调度其他线程工作，直播被锁资源释放，此时线程才会被唤醒。 pthread_mutex可以传入不同参数，实现递归锁pthread_mutex(recursive)。 其次NSLock、NSCondition、NSRecursiveLock、NSConditionLock都是内部封装的pthread_mutex,都属于互斥锁。@synchronize是NSLock的封装，牺牲了效率，简洁了语法。
- OSSpinLock是自旋锁，自旋锁是忙等的锁，被访问资源被锁时，线程不会休眠，而是不停地循环等待，直到被锁资源被释放。自旋锁效率较高，但是存在优先级反转的问题：就是等待线程的优先级更高，会一直占用CPU，优先级低的线程就无法释放锁；
- os_unfair_lock 用来替换OSSpinLock，但是它并非忙等的锁；
- pthread_mutex 互斥锁，等待线程时会进行休眠。它同时含有多种锁，比如pthread_mutex—递归锁、pthread_mutex—条件锁；
- NSLock是对pthread_mutex 普通锁的封装；
- NSRecursiveLock是对pthread_mutex—递归锁的封装；
- NSCondition是对phread_mutex—条件锁的封装；
- NSConditionLock是对NSCondition的又一层封装；
- @synchronized是对pthread_mutex—递归锁的封装；

锁的具体实现

另外信号量和同步队列也能实现类似锁的操作！

死锁的几种情况

```
//1、串行队列：异步里同步嵌套
```

```
SerialQueue.async{
    print(1)
    SerialQueue.sync{
        print(2)
    }
}
//2、串行队列：同步里同步嵌套
SerialQueue.sync{
    print(1)
    SerialQueue.sync{
        print(2)
    }
}
//3、主线程中执行同步操作
viewDidLoad(){
    DispatchQueue.main.sync{
        print(3)
    }
}
//4、NSOperation 线程间依赖
let operaA = Operation()
let operaB = Operation()
operaA.addDependency(operaB)
operaB.addDependency(operaA)
```

产生死锁的四个必要条件：

- 互斥条件：进程所分配的资源不允许其他进程访问。若其他进程访问，只能等待；
- 请求和保持条件：进程获得一定资源后，又对其他资源发出请求，但是其他资源可能被其他进程占有，此时请求阻塞，但又对自己获得的资源保持不放；
- 不可剥夺条件：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完自己释放；
- 环路等待条件：指进程若发生死锁，必然存在一个“进程——资源”之间的环形链

高效使用多线程

- 减少队列切换
- 控制线程数量。一个进程最多开启多少线程？目前我不是很清楚。但是过多的线程及线程调度肯定是非常耗性能的，所以最好的做法是开启和CPU核心数量一样的串行队列，提高CPU使用效率。
- 权衡线程优先级。比如大量图片异步解压过程。应该让查询磁盘的线程优先级高一点，让解压的优先级低一点。
- 主线程优化。比如Cell的复用机制、懒加载机制都能减少CPU的使用

如何高效使用多线程

事件及手势

触摸——事件——响应者

- 触摸UITouch：就是一次触摸屏幕。它对应生成一个UITouch对象。多个手指触摸生成多个UITouch对象。在同一个位置双击，则会更新第一次单击的tap count值。每个UITouch对象记录了

触摸的时间、位置、阶段、所处视图、窗口等。
- 事件UIEvent：一个触摸事件对应一个UIEvent。当然了UIEvent也可能不对应UITouch，也可能是其他的事件，比如抖动事件。
- 响应值UIResponder：每个响应者都是一个UIResponder对象。比如UIView、AppDelegat等。

事件响应

- 1、手指触碰到屏幕后，将由系统判断决定是否传递给前台APP进程
- 2、APP进程接收到触摸事件后，触发source1回调，主线程的runloop就被唤醒；
- 3、主线程被唤醒之后，source1事件被包装成source0事件，添加到UIApplication对象的事件队列中；
- 4、等到事件出队后，就开始寻找最佳响应者的过程；
- 5、寻找到最佳响应者之后，接下来的事情便是事件在响应链中的传递及响应了；

寻找最佳响应者

- 1、事件出队后，UIApplication首先将其传递给UIWindow，如果存在多个Window，则优先传给后显示的window；
- 2、若该window不能响应事件，则将事件传递给其他window。若该window能响应事件，则从后往前询问窗口子视图；
- 3、子视图中，若能响应，则自下而上传递，若其子视图不响应，则自身是最合适的响应者。若不能响应，则传递给其同级的前一个兄弟视图；
- 4、找到最佳响应者后，然后将其回溯返回给UIWindow。
- 5、找到最佳响应者后，则开始将事件在响应链中进行传递。**注意**前面步骤的传递是自下而上（window到最佳响应者）地传递，为了找到最佳响应者。而这里的事件传递则是将直接在响应链中传递，以便将事件进行响应，这是自上而下（最佳响应者向UIWindow传递）进行的；
- 6、每个响应者都有4个默认实现的响应事件触摸的方法：touchBegan:withEvent:、touchMoved:withEvent:、touchEnded:withEvent:、touchCancelled:withEvent:。所以可以在这些方法中对事件进行拦截做一些额外的处理。**touchBegan:withEvent也正是事件在响应链中的传递方式**；所以重写touchBegan:withEvent也能够让事件不再往下传递。

如何判断是否能响应?

每个UIView都有一个hitTest:withEvent方法，该方法返回一个能响应的UIView对象。比如：事件传递到UIWindow，UIWindow执行hitTest:withEvent:判断自身能否响应事件，若可以，则调用子视图的hitTest:withTest:将事件传递给子视图，并在子视图中查找最佳响应者。找到了之后就回溯给UIApplication一个最佳响应者视图。 hitTest:withEvent:方法的判断标准包括：是否允许交互、是否隐藏、透明度是否小于0.01，若这三个条件都通过，则会判断触摸点是否在当前视图的坐标系范围内。

如何判断触摸点是否在当前视图坐标系范围内?

pointINside:withEvent:方法用于判断触摸点是否在自身坐标范围内，返回一个bool值。注意的是次判断的时候都得以自身坐标系为准，所以应当使用covertPoint:toView:先进行坐标系转换。所以一般项目中碰到扩大按钮点击区域、tabBar凸出的按钮点击不响应等问题，则可以重写pointInside:withEvent:方法。

知道最佳响应者后，事件如何继续沿着响应链传递?

首先每个响应者都有一个nextResponder方法，用于将事件传递下一个响应者：若UIView是视图控制器的根视图，则其nextResponder是视图控制器，否则其nextResponder是其父视图；若视图控制器是window的根控制器，则其nextResponder是UIWindoi，否则是其前一个控制器；UIWindow的nextResponder是UIApplication；UIApplication的nextResponder是AppDelegate。 所以一旦找到最佳响应者后，整个响应链也就确定了。然后UIApplication会将事件通过sendEvent根据UITouch绑定的window传递给对应的UIWindow，UIWindow同样使用sendEvent通过UITouch绑定的touch对象将事件传递给hitTestView。

UIResponder、UIGestureRecognizer、UIControl同时存在时，会怎么响应?

- **手势识别器比UIResponder的响应优先级更高**。Window在将事件传递给最佳响应者之前，会将事件先传递给最佳响应者视图对应的视图控制器的手势识别器中。若手势识别器成功识别了事件，则就会将touchBegin/Move/End等回调打断，然后将UIGestureRecorognizer标记为待处理。Observe检测到BeforeWaiting事件后，会回调获取刚被标记成待处理的UIGestureRecorognizer，并执行UIGestureRecorognizer的回调；如果手势有变化过程，则回调也会进行对应处理。
- **UIControl比手势识别器的响应优先级更高**。比如UIButton。

触摸事件全家桶

网络相关

HTTP：

http其实就是超文本传输协议，它主要包括请求报文和响应报文两部分组成。请求报文包括：方法(get、post)、url、http版本、首部字段(媒体类型、Encode编码格式、认证信息等)、实体主题；响应报文包括：http版本、状态码、首部字段(accept_range字节范围、时间、重定向URI)、实体主题。

http的特点：无连接（也就是每次请求连接都需要经历连接和断开的过程）、无状态（同一个用户在多次发送http请求时，server端并不知道是同一个用户发送的）

状态码：2xx、3xx、4xx、5xx，200请求成功，301、302一般是重定向问题、404一般是网络问题、504一般是指服务器问题。

Get和Post的区别：Get请求的参数通过？拼接在URL后面，post则放在Body里面。然后Get是安全的、幂等的、可缓存的，Post是非安全的、非幂等的、非可缓存的。安全性是指是否会引起server端的变化，幂等是指一种请求方法执行多次的结果是否完全相同，可缓存的是指代理服务器是否会进行缓存。

持久连接：应对http无连接特点。指在一定时间范围内不需要反复进行握手和挥手动作。http提供的持久连接的方案就是修改请求头部字段，比如connection: keep-alive(客户端期许采用持久连接)、time:20(持久连接时间)、max：10(这条连接最多可以发生多少次请求和响应对)。 那怎么判断持久连接中的一次连接已结束呢？1)通过响应头部字段：content-length:1024来判断。2)还有就是最后一个报文的chunked字段是否为空来判断。

Charles的抓包原理：利用了http中间人攻击漏洞进行抓包，中间人就是模仿客户端和server端的所有操作。

HTTPS

HTTPS：HTTPS = HTTP + SSL/TLS。 HTTPS连接的建立流程：客户端先向server发送一个SSL版本及支持的加密算法和随机数C，server会返回一个选定的加密算法、随机数S、server端证书。客户端接收到后会先验证server证书（也就是server端公钥），然后通过C、S、预组秘钥组装成会话秘钥，之后通过server端公钥对预组秘钥进行加密发送给server端，server端则通过私钥解密预组秘钥，然后通过C、S、预组秘钥组装会话秘钥。然后客户端和server端相互发送一个加密消息，验证安全通道是否建立完成。 HTTPS都使用了哪些加密手段？ 对称加密和非对称加密。非对称加密在公私钥中使用到，传输过程中则是使用对称加密。 什么是非对称加密和对称加密？ 非对称加密包含两个概念：公钥、私钥。加解密使用的钥匙不一样的。用公钥加密，就得用私钥解密；私钥加密就得用公钥解密。 对称加密：加解密用的是同一个秘钥。

Q：如何使用Charles抓取Https协议，原理是什么？

首先Charles是根据中间人攻击的方式来抓取网络请求的。具体过程就是：Charles拦截客服端发往服务端的网络请求，然后再往服务端发送请求，服务端返回的响应数据，Charles也会再截取之后往客户端发送。一句话**Charles就相当于同时干了server端和客户端的所有事情**。而HTTPS又是怎么拦截呢？首先原理是一样的，其次HTTPS较HTTP多了一个TLS协议，要支持HTTPS协议的服务器也必须去CA申请一个证书，然后客户端校验通过后才会建立起连接通道，所以Charles想要拦截HTTPS协议，也必须先去CA申请一个证书（过程就是用手机浏览器访问charlesproxy.com/getssl,安装好证书。然后Charles需要在SSL Proxying添加上APP的访问域名和端口号，注意**端口号是443**）。然后使用的过程中手机需要先去设置里信任证书。意思就是客户端安装的是Charles服务器的CA证书，Charles安装的是服务器的CA证书。

Q：HTTPS的验证证书怎么做的

TCP/UDP

UDP：用户数据报协议。 UDP特点：无连接、尽最大努力交付（不保证可靠传输）、面向报文(既不合并报文也不拆分报文，会原封不动传输报文，只是在运输层会拼装一个UDP首部)； UDP提供的功能：复用(就是不同的端口都可以复用传输层UDP数据报)、分用(接收到数据报后，会根据目的端口进行分发)、差错检测(就是发送方通过某种方式对数据报中的数据进行计算，然后将得出的结果插入到UDP首部传输给接收方，然后接收方接收到数据后，运用相同的方式进行计算，然后对比接收到的数据，进行差错检测。)

TCP：传输控制协议。则需要建立连接。

TCP特点：面向连接（数据传输开始和结束需要建立和释放连接）、可靠传输（无差错、无重复、按序到达）、面向字节流、流量控制、拥塞控制。

三次握手：客服端向server端发送一个请求报文，server端接收到请求报文后会发送一个确认响应报文同时也附带一个建立连接的请求报文，客户端收到server端发来的请求报文后再会给server端一个确认报文。

四次挥手：客户端向server端发送一个请求断开连接的报文，server端收到后会返回一个确认报文（这样客户端对server的连接就断开了），然后server端向客户端发送一个断开连接的请求报文，客户端收到后回复一个确认报文（这样server端对客户端的连接也断开了）

实现细节：TCP的连接是全双工的，即连接双方的读写可通过一个连接来进行，所以这也是必须4次挥手的原因。连接过程中，**发送端和接收端都有一个数据缓冲区**。发送方连续发送的多次写操作的数据会先被放入TCP的发送缓冲区，当数据真正需要发送时，缓冲区的数据则可能会被封装成一个或多个数据包发出，具体实现则要看滑动窗口的设置。接收端接收到一个或多个TCP报文后，则会按照TCP报文的序

号将其有序放入到接收端缓冲区中。接收端应用也通过滑动窗口的设置来读取一段完整有序的数据。

TCP头部：TCP头部最长是60字节，头部包含的信息有：字节编号、端口号、确认号、标志位（ACK标志、SYN标志、FIN标志、PSH标志（提醒应用层从缓存区读取数据））、16位窗口大小（控制流量）、16位校验和（差错检测）。

Q1:为什么是3次握手，而不是两次？ 假如客户端发送建立连接的请求报文发生了超时，客户端会启用超时重传策略，重新发送连接请求，Server端收到了会回复确认报文，那之后又收到了之前超时的请求连接，那就又会建立一次连接，这样就可能建立了两次连接。而多了那次客户端确认报文则可以解决这种问题。

Q2、为什么是4次挥手，要分别断开两个方向的链接？ 因为TCP建立的是一个全双通的链接，就是无论从客户端到server端，还是server端到客户端，都可以建立单独的发送与确认接收的通道，比如说在4次握手中，如果仅仅是进行了前两步（断开了客户端到server端的链接），那么此时客户端是不能向server端发送数据的，但是server端依旧可以向客户端发送数据。

Q3 、怎么保证可靠传输的？ 可靠传输是通过停止等待协议来实现的。它是包括4方面的：

无差错情况：就是每次报文的传输中，server端收到后都会返回一个确认报文。

超时重传：那如果超时了，也就意味着在这个时间范围内，server端没有收到报文，也自然没有返回确认报文，那客户端就会进行重新发送。

确认丢失：是指server端返回的确认报文丢失了，那么同样的，客户端没有在时间限定内收到确认，所以会进行重新发送。server端会将第一次接到的报文丢失掉。

确认迟到：指确认报文迟到了，客户端同样会进行确认丢失一样的操作。

Q4、面向字节流? 是指TCP并不是原封不动地将发送方发送的字节一次性地完全地传输给接收方，而是会根据实际情况对字节流进行**拆分或合并**，然后再进行发送。和UDP的面向报文的方式正好相反。

Q5、怎么做到流量控制、按序到达？ 通过滑动窗口协议实现。TCP的发送方和接收方都有一个缓存区域，缓存区域的数据都是有序的，可以把它当作是有顺序编号的。在发送方，有一个类似滑动窗口用于从缓存数据里取一个有序序列往接收方发送；为了避免接收方数据溢出或因网络问题导致大数据无法发送，接收方在每一次接收完成后会在确认报文里返回一个数据，告诉发送方当前可以接收的数据。同时发送方在每次发送完成收到确认报文后会将已发送的最后字节标记，然后按照确认报文里的数据来确定滑动窗口的大小，比如说缓存区有1~9个数据，1~5是已经发送完成的，这个时候接收方接收到前三个数据后，表示由于网络环境较差，只能接收2个字节，那么就会在确认报文里返回2，这样滑动窗口的大小就变成了2。一句话说就是通过接收方来动态调节发送方的发送速率。这样就可以做到流量控制！

同样的，接收方的缓存区，如果接受到了1、2、3、6编号的数据，那么它只会对按序到底的下一个期望到底的字节进行标记，这里也就是4，而往应用层发送已排序好的1、2、3序列部分，那如果后面又接收到了非期望值的数据，则不会处理。这样就做到了按序到底！

Q6、拥塞控制： 慢开始、拥塞避免：一开始先发送一个报文，如果没有发送拥塞，则翻倍发送2个，然后再4个、16个(指数增长的方式)。一直达到窗口的门限初始值为止；然后再通过拥塞避免的策略，以线性增长的方式发送报文，可能达到某个值得时候，就产生了网络拥塞（比如连续3个报文没有收到确认报文），此时就越高采用拥塞避免的乘法减小的策略，只发送一个报文，同时将门限值降低，然后重新开始“慢开始”。

快恢复、快重传：是指在达到拥塞时，回到新的门限值，以线性增长的方式发送报文，而不经过前面指数增长的慢开始阶段。

Q7：为什么不直接用TCP，而要用HTTP?

TCP是建立在传输层上的协议，HTTP是应用层协议。TCP主要是用于建立传输通道的，HTTP则是建立在TCP的连接通道之上进行数据收发的。

DNS解析：

DNS解析过程？ DNS服务器是提供域名到IP之间的解析服务，一般计算机就是一个IP地址，但是纯数字不符合人类的记忆习惯，所以一般会有一个域名，比如www.baidu.com ，当我们代码对某个域名发起访问的时候，则要通过DNS解析，找到对应的IP，然后再进行访问 。一般DNS解析都是有运营商进行管理，比如移动的卡发出来的访问，先经过移动运营商，然后移动运营商找到对应的ip，然后进行访问。DNS采用UDP数据报文，53端口号，且明文。

DNS解析查询方式： 1、递归查询：按照 本地DNS——根域DNS——顶级DNS——权限DNS的层级一层层递归查找； 2、迭代查找：先查询本地DNS，然后本地DNS依次询问根域DNS、顶级DNS、权限DNS； DNS劫持问题？ 因为DNS是UDP明文传输，就有可能被钓鱼DNS劫持，返回错误的IP地址。DNS劫持与解析都与http无关，它是发生在http之前的操作。 解决DNS劫持： 1、httpDNS：实际上DNS解析是指DNS协议向DNS服务器的53端口进行请求，采用HTTPDNS这种方式则是直接通过http协议向DNS服务器的80端口进行请求，这样实际上就不存在DNS解析了，所以也就不存在DNS解析问题了。比如：http://119.29.29.29/d?dn=www.baidu.com&ip=163.177.153.109 (其中http://119.29.29.29/d 是国内最大的DNS域名服务器，dn=www.baidu.com 是需要解析的域名，后面是本地IP地址) 2、长连接： 客户端采用 长连server 从API Server通过内网专线获取IP的方式

DNS解析转发问题？ 是指DNS解析服务器为了节省资源，将解析请求发送给其他DNS域名服务器，依次转发，最后返回的IP地址可能不是同一运营商的网络，存在跨网访问的可能，造成一些请求缓慢等效率问题。

Session/Cookie

应对http无状态特点，指多次发送同一个请求，server端无法知道是否是同一个用户。 Cookie主要是用来记录用户状态，区分用户；状态主要保存在客户端； Session也是主要用来记录用户状态，区分用户；状态主要存放在server端。

socket

socket是对TCP/IP协议的封装，它本身并不是协议，而是一套API，通过socket才能使用TCP/IP协议，它包含了：连接使用的协议、本地主机IP、本地进程协议端口、远程主机IP、远程主机协议端口等信息。 IP地址：用于区分哪一台机器需要建立连接； 端口号：用于区分哪一个应用需要建立连接；socket通信报文一般分为报头和正文。报头一般包含：操作指令用于解开正文、正文长度、报文认证信息等。

socket与http：

- http HTTP是超文本传输协议，建议在TCP协议之上的应用层协议。HTTP请求时在数据需要更新时由客户端发送网络请求到服务端，请求结束后会主动断开连接。由于这种每次建立连接到关闭都是一次性连接，所以HTTP也被称为短连接。
- socket socket可以支持不同的传输层协议(TCP、UDP)，当使用TCP时，socket连接就是TCP连接。Socket是长连接、双向通信的，建立连接后客户端和服务端会一直连着，当有数据更新时，服务器会直接发送给客户端，不需要客户端主动请求。当然了，在连接过程中,长期不活跃的连接有可能被系统断掉，为了保证连接不断开，客户端也会定时不间断地发送心跳数据，如果连接断了，

则需要自己手动重连。

socket粘包和半包处理：

如果socket是TCP连接，则就可能涉及到粘包和半包问题。粘包就是多组数据被一并接收了，粘在一起，无法划分。半包就是数据不完整，无法处理。解决的方式：在socket通信报文的头部的信息中，会约定好一个字段专门用于描述数据包的长度，这样就使数据有了边界，依靠这个边界，粘包的数据就能划分出来，半包的数据也能知晓数据的缺失。

socket Net打洞：

- 什么是NAT穿透技术？ NAT叫做网络地址转换，主要是将内部的私有IP转换成可以在公网使用的公网IP；
- 打洞流程：

  1、客户端A、B分别发送消息给服务器S;

  2、S转发A的IP+port给B，同时也转发B的IP+port给A，使得A、B都知道对方的IP+port；

  3、A直接发消息给B，此时B会屏蔽这条消息，但是也让A的NAT映射中加上了一条映射，允许接收来自B的消息。所以B——>A打洞成功

  4、B发消息给A，由于A能接收到这条消息，同时B的NAT映射上也添加了一条可以接收来自A的消息的映射。此时A——>B打洞成功。

swift socket实战

数据库

SQLite基础语句学习

SQLite数据库中有一个sqlite_master的表，它定义数据库的模式，可以在其上查询以获取所有表的索引。

- 创建表：先判断表是否存在如果表不存在就新建：

```
CREATE TABLE IF NOT EXISTS User(uid interger primary key, uname
varchar(20),mobile varchar(20))
```

- 查询表：

```
SELECT count(*) FROM sqlite_master WHERE type="table" AND name="查询的表名"
```

- 获取某张表的建表语句

```
SELECTF sql FROM sqlite_master WHERE type="table" AND name="查询的表名"
```

- 获取表中的所有列名

```
PRAGMA table_info([查询的表名])
```

- 插入数据

```
INSERT INTO User(uid, uname, mobile)
```

- 更新数据

```
UPDATE
```

- 删除数据
- Demo

```
//获取数据库实例
let db = SQLiteDB.sharedInstance

//打开数据库
        _ = db.openDB()
//如果表还不存在则创建表（其中uid为自增主键）
let result = db.execute(sql: "create table if not exists t_user(uid integer
primary key,uname varchar(20),mobile varchar(20))")

//查询表
let result = db.query(sql: "SELECT count(*) FROM sqlite_master WHERE
type=\"table\" AND name = \"t_user\"")
print("result: \(result)")
//结果判断
if result.count > 0 && result[0].count > 0 {
    print("存在t_user表!")
}else{
    print("不存在t_user表!")
}

//获取某张表的建表语句
let result = db.query(sql: "SELECT sql FROM sqlite_master WHERE type=\"table\"
AND name = \"t_user\"")
//输出结果
print("---查询结果---\n\(result)")
print("---建表SQL---\n\(result[0]["sql"]!)")

//获取表中的所有列名
let result = db.query(sql: "PRAGMA table_info([t_user])")
//结果判断
print("---查询结果---\n\(result)")
//提取列名
print("---共\(result.count)列---")
for column in result {
    print(column["name"]!)
}
```

```
//搜索
let data = db.query(sql: "select * from t_user")
if data.count > 0 {
        //获取最后一行数据显示
        let user = data[data.count - 1]
        txtUname.text = user["uname"] as? String
        txtMobile.text = user["mobile"] as? String
}

//插入数据库，这里用到了esc字符编码函数，其实是调用bridge.m实现的
let sql = "insert into t_user(uname,mobile) values('\(uname)','\(mobile)')"
print("sql: \(sql)")
//通过封装的方法执行sql
let result = db.execute(sql: sql)
print(result)
```

事务

事务是指一个操作序列，这些操作要么执行，要么不执行。比如说转账：从一个账号里扣款，使另一账号增款。这两个操作要么执行，要么不执行。 事务是数据库维护数据一致性的单位，每个事务结束时，都能保持数据一致性。 事务的基本特征：

- 原子性：指事务中的所有操作，要么全部成功，要么全部失败；
- 一致性：只有合法的数据可以被写入数据库，否则事务应该将其回滚到最初状态；
- 隔离性：事务允许多个用户对同一个数据进行并发访问，而不破坏数据的正确性和完整性。同时，并行事务的修改必须与其他并行事务的修改相互独立；
- 持久性：事务结束后，事务处理的结果必须能够得到固化；

Realm学习：

FMDB学习:

https://www.hangge.com/blog/cache/detail_2318.html

https://juejin.im/entry/5a1d44a6f265da432f30dd09

加解密

加密解密主要是为了解决数据安全的问题。比如使用POST请求，因为POST请求数据放在body里面相对get直接将参数暴露在外部更安全一些，其次使用HTTPS，对接口进行加密。 数据安全基本原则：

- 在网络上不允许传输用户隐私数据的明文；
- 在本地不允许保存用户隐私数据明文；

常用加密方法

- 编码方案：Base64: 可以将任意二进制数据进行Base64编码，所有的数据都能被编码为只用65个字符A~Z a~z 0~9 + / =）就能表示的文本文件；
- 哈希函数：

    - MD5摘要算法： 对相同的数据进行加密，得到的结果是一样的；对不同的数据进行加密，得到的都是32位的定长字符串；其结果是不可逆的，也就是只能加密，不能解密； 但是现在的MD5已不再安全，可以被暴力破解。所以可以使用"加盐"的方案来增加解密的难度。比如，在明文里插入随机串，再进行MD5；也可以对明文进行乱序或多次MD5等。或者使用最新的SHA-2算法。 **哈希函数其实就是对数据内容进行计算，算出一个散列值，这个散列值就用来校验信息的完整性**。
- 对称加密算法：
    - DES：数据加密标准
    - AES：高级加密标准 **对称加密的特点就是加密/解密使用相同的密钥**；加密和解密的过程是可逆的；加密的过程是先加密再base64编码，解密的过程是先base64解码再解密。
- 非对称加密算法：
    - RSA： **非对称加密的特点：使用公钥加密，使用私钥解密，或者使用私钥加密，使用公钥解密**；公钥是公开的，私钥是保密的。加密安全，但是性能消耗较大。
- 数字签名： 信息的发送者用私钥加密，接收者用公钥解密。也就是说私钥加密的密文只能用公钥来解密，所以叫做数字签名，通常用于身份认证。证书就是经由CA机构数字签名之后的公钥证书，用于身份认证。
- HTTPS：HTTP+TLS：略 总结： 对称加密：甲方使用某一个密钥对信息进行加密，乙方也使用同一套密钥对信息进行加密。但是密钥的保存和传递就成了最头疼的问题； 非对称加密：乙方生成两把密钥(公钥和私钥)。公钥是公开的，任何人都可以获取，私钥是保密的。甲方获取乙方公钥，然后用它对信息进行加密。乙方得到加密后的信息，使用私钥进行解密。甲方获取到乙方的信息后也可以直接用公钥去解密私钥加密的数据。

TLS协议

TLS协议就是HTTP协议中的'S'。TLS协议可以对传输内容进行加密、进行身份验证、避免传输内容串改。TLS协议包括TLS握手协议和TLS记录协议。TLS握手协议使用到的加密手法包括：非对称加密、散列函数、数字签名技术；TLS记录协议使用了对称加密、散列函数等技术。

通常的做法：

- 1、使用Base64防止数据明文传输；
- 2、对普通请求可以使用MD5进行数据完整性校验；
- 3、对于重要的数据，使用RSA进行数字签名；
- 4、对于敏感的数据，客户端使用RSA加密，服务器返回DES(AES)加密；
- 5、想要安全的数据传输，使用HTTPS。但是最好加上双向验证防止中间人攻击；

ios常用加密手法

动画

动画就是由一帧帧画面组成的。 基于iOS系统的屏幕刷新率是60FPS，也就是每秒60帧画面，每一帧相当于16.7ms。 动画的本质就是在一段时间间隔内，对象的位置、形状、透明度等属性随着时间变化的过程。 而负责动画操作的则是CALayer，CALayer的作用主要是为了内容展示和动画操作。

逐帧 和 关键帧

iOS的动画主要分为两大类：逐帧动画 和 关键帧动画：

逐帧动画：就是对过程中的每一帧画面进行绘制

实现方式就是周期性地调用绘制方法，绘制每帧的动画对象。这里的周期性使用的是CADisplayerLink，即屏幕每次刷新都进行调用。

关键帧动画：理论上就是提供关键节点，比如起始点、中间点、终点等关键位置，其他的过度位置就交由计算机自动生成。

一般我们提供起始帧、结束帧、动画时间、匀速运动；起始帧和结束帧是我们设定的，其他的过渡帧是系统生成的。CALayer的同一个属性值，会分别保存在模型层和展示层。当我修改一个属性值时，修改的是模型层的数值，动画时系统根据模型层的变化生成过渡值，将其保存在展现层中。执行的过程：

- 1、动画前，显示模型层的当前值；
- 2、动画开始，切换显示展示层的值；
- 3、动画过程中，展示层的值会随着时间变化，我们看到的实际是展示层的值在变化；
- 4、动画结束时，切换回显示模型层的值。此时的模型层的值也被修改为动画结束时展示层的值。
  代码实现：

```
UIView *view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 100, 100)];
    view.backgroundColor = [UIColor redColor];
    [self.view addSubview:view];

    CABasicAnimation *animation = [CABasicAnimation
animationWithKeyPath:@"position"];
    animation.fromValue = [NSValue valueWithCGPoint:CGPointMake(50, 0)];
    animation.toValue = [NSValue valueWithCGPoint:CGPointMake(150, 0)];
    [view.layer addAnimation:animation forKey:nil];

//    view.frame = CGRectOffset(view.frame, 100, 0);
```

解析： 这里注释了最后一步，也就是动画结束的设置，所以动画结束后又回到了原来的位置。

隐式动画

我们显式地给CALayer添加动画的方式叫做显示动画，比如上面的逐帧和关键帧动画。而如果我们改变CALayer的一个可动画属性值时，就会触发系统的隐式动画 比如，我们修改CALayer的frame。只是UIView对应的CALayer，系统关闭了隐式动画，所以修改UIView的CALayer的相关属性时，变化是直接生效的，没有动画效果。 所以如果我们做逐帧动画时给一个单独自建的CALayer变换frame，那么也会触发系统的隐式动画，这个时候就需要我们手动关闭隐式动画。

UIView的一系列animationWithDuration等动画方法。在这些方法中UIView的CALayer都会恢复隐式动画，所以在block中修改属性时，会触发隐式动画。

相关类

- CAAnimation : 核心动画基础类，不能直接使用；
- CAPropertyAnimation : 属性动画的基类（通过属性进行动画设置，注意是可动画属性），不能直接使用;
- CAAnimationGroup : 动画组，动画组是一种组合模式设计，可以通过动画组来进行所有动画行为的统一控制，组中所有动画效果可以并发执行;
- CATransition : 转场动画，主要通过滤镜进行动画效果设置;
- CABasicAnimation : 基础动画，通过属性修改进行动画参数控制，只有初始状态和结束状态;

- CAKeyframeAnimation : 关键帧动画，同样是通过属性进行动画参数控制，但是同基础动画不同的是它可以有多个状态控制;

基础动画、关键帧动画都属于属性动画，就是通过修改属性值产生动画效果，开发人员只需要设置初始值和结束值，中间的过程动画（又叫“补间动画”）由系统自动计算产生。和基础动画不同的是关键帧动画可以设置多个属性值，每两个属性中间的补间动画由系统自动完成，因此从这个角度而言基础动画又可以看成是有两个关键帧的关键帧动画; 隐式属性动画的本质是这些属性的变动默认隐含了CABasicAnimation动画实现;

总结：

- 关键帧动画的实现，只需要修改某个属性值就可以；
- 逐帧动画则绘制过程复杂、开销较大；
- 我们平时开发中使用UIView的隐式动画更多一些；

动画1 动画2

编译连接

编译型和解释型语言的区别：

- 编译型语言：先将代码经过编译器先编译成机器代码，然后再执行。所以每次修改完代码都得先编译，才能执行结果。优点就是**代码执行效率高**，缺点就是**编写调试周期长**。我们也称这种执行方式为“**AOT预先编译**”
- 解析型语言：不需要经过编译。执行的时候，通过一个解释器将代码解释成cpu可以执行的代码（实际上就相当于一边编译一边执行）。优点是**编写调试方便**，缺点是**执行效率不够高**。我们也称这种方式为“**JIT即时编译**”

OC的编译连接过程

OC和swift都是编译型语言。通过LLVM将代码转成机器码，主要的流程包括：

1、预编译：主要是处理源代码中以“#”开头的预编译指令：

```
clang -E xx.m -o xx.i
```

- “#define”删除并展开对应宏定义；
- 处理所有预编译指令：#if、#ifdef、#else、#endif；
- "#include、#import"包含的文件递归插入此处；
- 删除所有注释；
- 添加行号和文件名标识；

2、编译：就是把预编译得到的.i文件进行：词法分析、语法分析、静态分析、优化生成相应的汇编代码

```
clang -S xx.i -o xx.s
```

- 词法分析：顾名思义就是对每个词进行分析，把每一行代码分割成一个个token，主要是包括关键字、标识符变量名、字面量、特殊符号等。比如把变量名放到符号表等；
- 语法分析：生成抽象语法树AST。比如运算符优先级、括号匹配等；
- 静态分析：主要类型声明与匹配的问题。比如整形与字符串相加肯定会报错；

- 中间语言生成：这里是指根据AST自顶向下生成LLVM的IR语言（它是区别与源码和机器码的一种中间代码）
- 目标代码生成：根据IR生成依赖具体机器的汇编语言。

3、汇编：把上面得到的.s文件里的汇编指令翻译成机器指令

```
clang -C xx.s -o xx.o
```

4、链接：把目标文件（一个或多个）和需要的库（静态库、动态库）链接成可执行文件Mach-O

```
clang xx.o -o xx
```

Clang 和 LLVM

- LLVM是一个模块化可重用的编译器和工具链技术的集合。 LLVM IR 是一种“中间描述”。它是整个编译过程中有别于源码和机器码的中间代码。因此它是LLVM进行优化和代码生成的关键。LLVM的核心功能就是围绕IR建立的。 所以整个LLVM的编译架构就是：前端——>IR——>后端;所以这也让LLVM的扩展性做的非常好，如果需要支持一种新的编程语言，则只需要实现一种新的前端即可，如果要支持一种新的设备，则只需实现一个新的后端就好。
- Clang是LLVM的子项目，是C、C++、OC的前端编译器。 主要负责的工作就是上诉的编译流程：预处理——词法分析——语法分析——静态分析——生成LLVM IR；
- OC采用Clang作为编译器前端，Swift采用Swift作为编译器前端， swift编译器则是先生成一种SIL的中间代码，然后由SIL生成IR。

LLVM and Clang

MachO文件

MachO文件是iOS和OS X操作系统的可执行文件格式。 因为不同的CPU平台支持的指令集不一样，所以通常不同的CPU对应不同格式的MachO文件，比如arm64和x86。**通用二进制文件**则是指多种架构下的Mach-O文件"打包"在一起。 通用二进制文件常用命令：

```
//查看通用二进制文件中的MachO文件信息
$ file bq
$ otool -f -V bq

//lipo命令增、删、提取指定的MachO文件
//提取
$ lipo bq -extract armv7 -o bq_v7
//删除
$ lipo bq -remove armv7 -o bq_V7
//瘦身
$ lipo bq -thin armv7 -o bq_V7
```

MachO文件里主要就是代码和数据（全局变量）。而代码段和数据段是存在不一样的内存地址的，所以这个就需要**链接器**将其关联起来。

静态库与动态库

- 静态库：以.a和.framework为后缀；链接是会被完整复制到可执行文件中，被多次使用就有多份拷贝；
- 动态库：以.tbd和.framework为后缀；链接时不复制，而是由运行时动态加载到内存，系统只加载一次，多个程序共用，比如UIKit.framework;

WebView 及 前端 学习

JSBridge

https://juejin.im/post/5abca877f265da238155b6bc

JavaScriptCore

https://zhuanlan.zhihu.com/p/81634837

WebCore

https://blog.csdn.net/HorkyChen/article/details/8888428

设计模式

设计模式主要是 为了代码的可重用性，将某些功能的模块代码规范化，然后尽可能让他人容易理解。

任何一种设计模式都是一种设计原则。设计模式就是实现了设计原则，达到了代码复用和可维护性。设计模式又包含创建型模式、结构型模式、行为型模式。

设计原则

- **单一职责原则**：一个类只承担一个职责。比如UIView和CALayer的职责分离；
- **开闭原则**：对修改关闭、对扩展开发，如果要修改也尽量用继承或组合的方式来修改。比如category给元类添加方法；
- **里氏替换原则**：是指父类可以被子类无缝替换，且原有功能不受影响； 比如说KVO原理，动态实现了一个NSNotifing的子类，而我们感受不到任何变化；
- **接口隔离原则**：使用多个专门的协议而不是一个庞大臃肿的协议，协议中方法也应该尽量少。 比如UITableView将代理分成了delegate和datasource；
- **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。比如我们写数据库操作的时候，先用协议实现增删改查的接口，至于具体的实现：使用什么数据库、存储在什么位置等细节，上层业务则无需感知了；
- **迪米特法则**：一个对象应当对其他对象尽可能少的了解，尽可能实现高内聚、低耦合；比如项目中路由的使用，MVVM中viewmodel的使用；

常用设计模式

创建型模式

- 工厂方法模式：工厂模式就是给定一个工厂方法，然后通过传入的参数来确定生成具体的哪个实例。代码：

```
//一个抽象产品类：Phone,声明了一个call方法
protocol Phone {
    func call;
```

```
}
//两个具体的产品类：iPhone和HUAWEI
class IPhone : Phone {
    func call() {print("iphone call")}
}
class HUAWEI : Phone {
    func call() {print("huawei call")}
}

//工厂方法
class PhoneFactory {
    func createPhonre(_ name:String)-> Phone? {
        if name.contains("iPhonre") {
            return IPhone()
        }else if name.contains("HUAWEI") {
            return HUWEI()
        }
        return nil
    }
}

//使用
let p = PhoneFactory.createPhonre("iPhonre")
p.call()
```

优点：工厂类方法更好区分一个类簇的各色特点；缺点是一旦有新的产品添加的话，就得修改工厂方法，违反了开闭原则。

- 单例模式：保证当前类全局只有一个实例；
- 创建者模式：将复杂对象的构建与它的表示(属性、方法)分离；略
- 原型模式：简单来说就是通过深拷贝的方式创建对象；比如说A创建了一份简历，B则把A的简历拷贝了一份，然后修改了对应的姓名等部分信息。这就是原型模式的实现。 优点：对一些创建过程繁琐的对象来说，原型模式可以提高生成效率； 缺点：如果对象层级太多的话，手写clone方法也会比较复杂。 iOS 应用：OC中使用NSCopying协议，配合-(id)copyWithZone:(NSZone *)zone方法；或者使用NSMutableCopying协议配合mutableCopyWithZone方法实现。

结构型模式

(这里只讲几种自己熟悉的模式)

- 适配器模式：如果无法修改一个类已有的实现，则可以生成一个新类，持有该类主要是对于一些接口，如果当前类用不上，则可以通过二次封装，让其只包含当前类可用的接口，这就是适配器模式。
- 桥接模式：
- 组合模式：
- 装饰模式：
- 外观模式：
- 享元模式：
- 代理模式：

行为型模式

(这里只讲几种自己熟悉的模式)

- **责任链模式**:
- 命令模式:
- 解析器模式:
- 迭代器模式:
- 中介者模式:
- 备忘录模式:
- 观察者模式:
- 状态模式:
- 策略模式:
- 模板方法模式:
- 访问者模式:

https://www.jianshu.com/p/e5c69c7b8c00