UI

UIViewController

1. 每当显示一个新界面时，首先会创建一个新的UIViewController对象，然后创建一个对应的全局UIView，UIViewController负责管理这个UIView。
2. UIViewController就是UIView的大管家，负责创建、显示、销毁UIView，负责监听UIView内部的事件，负责处理UIView与用户的交互。
3. UIViewController内部有个UIView属性，它负责管理UIView对象:
   @property(nonatomic, retain) UIView *view。

UIScrollView

1. 不能滚动常见原因：
   1.1. 没有设置contentSize(滚动范围)
   1.2. 禁用滚动scrollEnabled = NO或禁用了用户交互userInteractionEnabled = NO
   1.3. 没有取消autolayout功能(要想scrollView滚动，必须取消autolayout)

UIButton

默认image在左，title在右，若想重新排列image和title布局样式，只需自定义一个CusUIButton继承UIButton，然后实现titleRectForContentRect和imageRectForContentRect在其中重新布局即可。

UIImage两种加载方式(内存优化)

1. + (UIImage *) imageNamed:(NSString *)name; // name是图片文件名
   1.1. 加载到内存以后，会一直停留在内存中，不会随着对象销毁而销毁，有缓存。
   1.2. 加载图片到内存以后，占用的内存归系统管理，程序员无法管理。
   1.3. 相同的图片不会重复加载，重复加载同一张图片占据内存不会增大。
   1.4. 加载到内存当中后，占据内存空间较大。
2. + (UIImage *)imageWithContentsOfFile:(NSString *)path // path是图片全路径
   - (id)initWIthContentsOfFile:(NSString *)path
   2.1. 加载到内存当中后，占据内存空间较小，无缓存。
   2.2. 相同的图片会被重复加载到内存当中，重复加载同一张图片占据内存会不断增大。
   2.3. 对象销毁的时候，加载到内存中的图片会随着一起销毁，图片所占用的内存会（在一些特定操作后）被清除。
3. 放在xcassets里面的图片只能通过imageNamed方法加载，图片放在非cassettes中则可以通过[[NSBundle mainBundle] pathForResource: imageName ofType: imageType];获取对应图片路径后再使用上述2加载图片。
4. 可将多张图片放入一个数组赋值给UIImageView的animationImages属性然后调用startAnimating开始动画播放，不过需要注意的是在动画播放完成后记得将

animationImages置为nil避免长期占用内存资源
[self.imageView performSelector:@selector(setAnimationImages:) withObject:nil
after:self.imageView.animationDuration+0.1]。

IBAction

1. 从返回值角度上看，作用相当于void。
2. 只有返回值声明为IBAction的方法，才能跟storyboard中的控件进行连线。

IBOutlet

1. 只有声明为IBOutlet的属性，才能跟storyboard中的控件进行连线。

退出键盘方式

1. resignFirstResponder: 当叫出键盘的那个控件(第一响应者)调用这个方法时，就能退出键盘。
2. endEditing: 只要调用这个方法的控件内部存在第一响应者，就能退出键盘。

通常开发流程

1. 搭建软件界面(UI)。
2. 获取网络数据(多线程、网络)。
3. 解析网络数据(json、xml)。
4. 展示数据到界面(UI)。

instancetype

1. 在类型表示上，跟id一样，可以表示任何对象类型。
2. 只能用在返回值类型上，不能像id一样用在参数类型上。
3. 比id多一个好处：编译器会检测instancetype的真实类型，若不匹配会报警告⚠️。

NSTimer

1. 启动定时器的两种方法
    1.1. timerWithTimeInterval需要手动将timer加入到消息循环中：

```
NSTimer *timer = [NSTimer timerWithTimeInterval:2.0 target:self selector:
    @selector(nextImage) userInfo:nil repeats:YES];
NSRunLoop *loop = [NSRunLoop currentRunLoop];
[loop addTimer:timer forMode:NSDefaultRunLoopMode];
[timer fire]; //这个方法仅仅是提前执行timer要执行的方法
```

1.2. scheduledTimerWithTimeInterval自动把timer加入到消息循环中，默认NSDefaultRunLoopMode：

```
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self
    selector:@selector(nextImage) userInfo:nil repeats:YES];
```

注意：如果当前线程run loop处于UIEventTrackingRunLoopMode模式会不处理定时器事件。

NSRunLoop

在Cocoa中，每个线程对象中内部都有一个run loop(NSRunLoop)对象用来循环处理输入事件(来自Input sources的异步事件和来自Timer sources的同步事件)。每个run loop可运行在不同的模式下，一个run loop mode是一个集合，其中包含其监听的若干输入事件源，定时器，以及在事件发生时需要通知的run loop observers。

1. Cocoa中预定义模式

1.1. Default模式

NSDefaultRunLoopMode(Cocoa),
KCFRunLoopDefaultMode(Core Foundation)
几乎包含了所有输入源(NSConnection除外)，一般情况下应使用此模式。

1.2. Connection模式

NSConnectionReplyMode(Cocoa)
处理NSConnection对象相关事件，系统内部使用，用户基本不会使用。

1.3. Modal模式

NSModalPanelRunLoopMode(Cocoa)
处理modal panels事件

1.4. Event tracking模式

NSEventTrackingRunLoopMode(Cocoa)
UITrackingRunLoopMode(iOS)
在拖动loop或其他user interface tracking loops时处于此种模式下，在此模式下会限制输入事件的处理。手指按住UITableView拖动时就会处于此模式。

1.5. Common模式

NSRunLoopCommonModes(Cocoa)
kCFRunLoopCommonModes(Core Foundation)
这是一个伪模式，其为一组run loop mode的集合，将输入源加入此模式意味着在Common Modes中包含的所有模式都可以处理。在Cocoa应用程序中，默认情况下Common Modes包含default，modal，event tracking。还可以使用CFRunLoopAddCommonMode方法向Common Modes中添加自定义modes。

注意：NSTimer和NSURLConnection默认运行在default mode下，当用户在拖动UITableView处于UITrackingRunLoopMode模式时，NSTimer不能fire，NSURLConnection的数据也无法处理；这种情况下可以在另外的线程中处理定时器事件，可把Timer加入到NSOperation中在另一个线程中调度。或者更Timer

运行的run loop mode，将其加入到UITrackingRunLoopMode或NSRunLoopCommonModes中。

NSURLConnection（已被NSURLSession取代）

1. 特点：处理简单网络操作非常简单，但是处理复杂的网络操作就非常繁琐。
2. 下载过程中，没有'进度的跟进'，需要通过代理的方法来处理。
3. 存在内存的峰值，解决方法是接收一点就写入文件一点。
4. NSURLConnection的代理默认是在主线程运行的，需要将其代理移到子线程执行。
   [connection setDelegateQueue:[[NSOperationQueue alloc] init]];
5. 默认下载任务在主线程工作，为避免阻塞UI更新也需要将其移到子线程执行，子线程默认不开启运行循环还需为其开启运行循环
   dispatch_async(dispatch_get_global_queue(0, 0), ^{下载操作})

NSURLSession

1. 是iOS7中出现的网络接口，可通过URL将数据下载到内存、文件系统，还可以将数据上传到指定URL，也可以在后台完成下载上传。是线程安全的。
2. NSURLSessionConfiguration
   2.1. 用于定义和配置NSURLSession对象。每一个NSURLSession对象都可以设置不同的NSURLSessionConfiguration从而满足应用内不同类型的网络请求。
   2.2. 三种类型
      2.2.1. defaultSessionConfiguration
         默认配置，使用硬盘来存储缓存数据，类似NSURLConnection的标准配置。
      2.2.2. ephemeralSessionConfiguration
         临时session配置，不会将缓存、cookie等保存在本地，只会存在内存里，所以当程序退出时，所有的数据都会消失。
      2.2.3. backgroundSessionConfiguration
         后台session配置，与默认配置类似，不同的是会在后台开启另一个线程来处理网络数据。
3. NSURLSessionTask
   3.1. NSURLSession使用NSURLSessionTask来具体执行网络请求的任务。
   3.2. 支持网络请求的取消、暂停和恢复，也能获取数据的读取进度。
   3.3. 三种类型
      3.3.1. NSURLSessionDataTask
         处理一般的NSData数据对象，如通过GET或POST从服务器获取JSON或XML返回等等，但不支持后台获取。
      3.3.2. NSURLSessionUploadTask
         用于PUT上传文件，支持后台上传。
      3.3.3. NSURLSessionDownloadTask
         用于下载文件，支持后台下载。

pch文件

1. pch文件也是一个头文件，里面的内容能被项目中的其他所有源文件共享和访问
2. 通常在pch文件中定义一些全局的宏及放置使用频繁的文件。
3. 在pch文件中添加下面的预处理指令，然后在项目中使用Log输出日志信息，在发布的时候会将NSLog语句移除(调试模式下才有定义DEBUG)。

```
#ifdef DEBUG
#define Log(…) NSLog(__VA_ARGS__)
#else
#define Log(…)
#endif
```

UIWindow

1. 是一种特殊的UIView，也是UIView的子类，通常在一个app中只会有一个UIWindow。
2. iOS程序启动完毕后，创建的第一个视图控件就是UIWindow，接着创建控制器的view，最后将控制器的view添加到UIWindow上，于是控制器的view就显示在屏幕上了。
3. 添加UIView到UIWindow的两种常见方式：
   3.1. -(void)addSubview:(UIView*)view;直接将view添加到UIWindow中，但不理会view对应的UIViewController。
   3.2. @property (nonatomic, retain) UIViewController *rootViewController;自动将rootViewController的view添加到UIWindow中，负责管rootViewController的生命周期。
4. 常用方法：
   4.1. -(void)makeKeyWindow;让当前UIWindow变成keyWindow(主窗口)。
   4.2. -(void)makeKeyAndVisible;让当前UIWindow变成keyWindow并显示出来。
5. 获取某个UIView所在的UIWindow: view.window。

注意: 3.1第一种方式是不能让控制器的view进行旋转，3.2第二种方式控制器的view可以旋转，因为旋转事件传递是由UIApplication->UIWindow(窗口不做旋转处理，只有控制器才会做旋转处理)->UIViewController控制器的。

状态栏设置样式

1. 由控制器的方法 - (UIStatusBarStyle)preferredStatusBarStyle决定。
2. 使用application设置
   application.statusBarStyle = UIStatusBarStyleLightContent;

注意：1.2第二种方式默认不起作用，因为状态栏样式默认由控制器来管理，如果想用application设置状态栏则需在Info.plist中设置View controller-based status bar appearance = NO） 。

导航栏

1. 局部配置： 获取导航控制器的navigationBar属性self.navigationBar
2. 全局配置：通过[UINavigationBar appearance]获取的导航条对象可以设置应用的”所有导航条”的样式

更改NavItem样式

1.获取NavItem: UIBarButtonItem *navItem = [UIBarButtonItem appearance];

2. 更改样式

2.1 改变整个按钮背景

[navItem setBackButtonBackgroundImage: [UIImage imageNamed:@“NavBackButton"] forState:UIControlStateNormal barMetrics:UIBarMetricsDefault];

2.2 设置Item的字体大小

[navItem setTitleTextAttributes:@{NSFontAttributeName:[UIFont systemFontOfSize:18]} forState:UIControlStateNormal];

视图控制器UIViewController的创建方式

1. UIStoryboard
   1.1. 获取storyboard：UIStoryboard *storyboard = [UIStoryboard storyboardWithName:name bundle:nil];。
   1.2. 依据storyboard创建视图控制器：
      1.2.1. [storyboard instantiateInitialViewController]获取箭头所指的视图控制器。
      1.2.2. [storyboard instantiateViewControllerWithIdentifier:ID]获取标识ID所指视图控制器。
2. xib： [[ViewControllerName alloc] initWithNibName:name bundle:nil]。
3. 代码：[[ViewControllerName alloc] init]。

视图UIView的创建流程

控制器view加载顺序从高到低依次为 loadView() > storyboard > nibName.xib > View.xib > ViewController.xib > 空白view。

查找最合适的控件来处理事件

1. 判断自己是否能接收触摸事件(下述三种情况不接收触摸事件)
   1.1. 不接收用户交互userInteractionEnabled = NO
   1.2. 隐藏 hidden = YES

1.3. 透明度alpha在0到0.0.1之间
2. 判断触摸点是否在自己身上
2.1. pointInSide:withEvent返回NO代表不在自己身上，不再遍历子控件，也不接收触摸事件；返回YES代表在自己身上，继续遍历子控件。
3. 依次遍历子控件，重复前面两个步骤
4. 若没有符合条件的子控件，则自己就是最适合处理的控件
5. 找到最合适的控件后就调用touchesBegin/touchesMoved/touchesEnd方法
注意：UIImageView的userInteractionEnabled为NO，所以UIImageView及其子类默认是不能响应触摸事件的。

手势识别

1. 敲击手势UITapGestureRecognizer
2. 长按手势UILongPressGestureRecognizer
3. 轻扫手势UISwipeGestureRecognizer
4. 捏合手势UIPinchGestureRecognizer
4.1. scale缩放比例是累加的需在使用完后重置为1
5. 旋转手势UIRotationGestureRecognizer
5.1. rotation旋转角度是累加的需在使用完后重置为0
5.2. 若想和捏合手势同时使用，需设置其中一个手势的代理并实现
- (BOOL)gestureRecognizer:
shouldRecognizeSimultaneouslyWithGestureRecognizer: 返回YES即可，允许跟其他手势一起发生
6. 拖拽手势UIPanGestureRecognizer

bounds

设置当前视图左上角相对子视图新的坐标值，默认为(0,0)，若设为(50,0)则子视图在视觉上向左偏移50，即以前的原点 (0,0)变为了(50,0)。

核心动画

开发步骤：
1. 初始化一个动画对象(CAAnimation)并设置一些动画相关属性(如opacity，position，transform，bounds，contents等)。
2. 添加动画对象到层(CALayer)中，开始执行动画：调用CALayer addAnimation:forKey。
3. 停止层中的动画：removeAnimationForKey。

1. CAAnimation

1.1. 所有动画对象的父类，负责控制动画的持续时间和速度. 是个抽象类. 不能直接使用. 应使用它具体的子类。

1.2. 常用属性

duration：动画持续时间；

repeatCount：动画的重复次数；

repeatDuration：动画的重复时间；

removedOnCompletion：默认为YES，代表动画执行完毕后就行图层上移除，图层恢复到动画执行前的状态。若想让图层保持显示动画执行后的状态，就设置为NO，并且还要设置fillMode为kCAFillModeForwards；

fillMode：决定当前对象在非active时间段的行为，如动画开始前后动画结束后。

beginTime：可以用来设置动画延迟执行时间。

若想延迟2s，就设置为CACurrentMediaTime()+2。

timingFunction：速度控制函数. 控制动画运行的节奏。

delegate：动画代理。

2. CAPropertyAnimation

2.1. 是CAAnimation的子类，也是个抽象类，要想创建动画对象，应该使用它的两个子类：CABasicAnimation和CAKeyframeAnimation。

2.2. 属性解析

keyPath：通过指定CALayer的一个属性名称为keyPath(NSString类型)，并且对CALayer的这个属性的值进行修改，达到相应的动画效果。比如指@“position”为keyPath，就修改CALayer的position属性的值，以达到平移的动画效果。

3. CABasicAnimation

3.1. CAPropertyAnimation子类。

3.2. 属性解析

fromValue：keyPath相应属性的初始值。

toValue：keyPath相应属性的结束值。

byValue：动画增加值。

随着动画的进行，在长度为duration的持续时间内，keyPath相应属性的值从fromValue渐渐地变为toValue。如果fillMode=kCAFillModeForwards和removedOnCompletion=NO,那么在动画执行完毕后，图层会保持显示动画执行后的状态。但在实质上，图层的属性值还是动画执行前的初始值，并没有真正被改变比如，CALayer 的position初始值为(0,0)，CABasicAnimation的fromValue为(10,10)，toValue为(100,100)，虽然动画执行完毕后图层保持在(100,100)这个位置，实质上图层的position还是为(0,0)。或者设置animation的delegate在代理方法animationDidStop中处理让动画保持执行后的状态。

4. CAKeyframeAnimation

4.1. CAPropertyAnimation子类，跟CABasicAnimation的区别是:CABasicAnimation只能从一个数值(fromValue)变到另一个数值(toValue),而CAKeyframeAnimation会使用一个NSArray保存这些数值。

4.2. 属性解析

values：就是上述的NSArray对象。里面的元素称为”关键帧”(keyframe)。动画对象会在指定的时间(duration)内，依次显示values数组中的每一个关键帧。

path：可以设置一个CGPathRef\CGMutablePathRef，让层跟着路径移动。path只对CALayer的anchorPoint和position起作用。如果你设置了path，那values将被忽略。

keyTimes：可以为对应的关键帧指定对应的时间点，其取值范围为0到1.0，keyTimes中的每一个时间值都对应values中的每一帧。当keyTimes没有设置的时候，各个关键帧的时间是平分的。

CABasicAnimation可看做是最多只有2个关键帧的CAKeyframeAnimation。可用于监听设备的抖动即实现摇一摇功能。

5. CAAnimationGroup

5.1. CAAnimation的子类，可以保存一组动画对象，将CAAnimationGroup对象加入层后，组中所有动画对象可以同时并发运行。

5.2. 属性解析

animations：用来保存一组动画对象的NSArray。默认情况下，一组动画对象是同时运行的，也可以通过设置动画对象的beginTime属性来更改动画的开始时间。

6. CATransition

6.1. CAAnimation的子类，用于做转场动画，能够为层提供移出屏幕和移入屏幕的动画效果。

6.2. UINavigationController就是通过CATransition实现了将控制器的视图推入屏幕的动画效果。

6.3. 属性解析

type：动画过渡类型。

subtype：动画过渡方向。

startProgress：动画起点(在整体动画的百分比)。

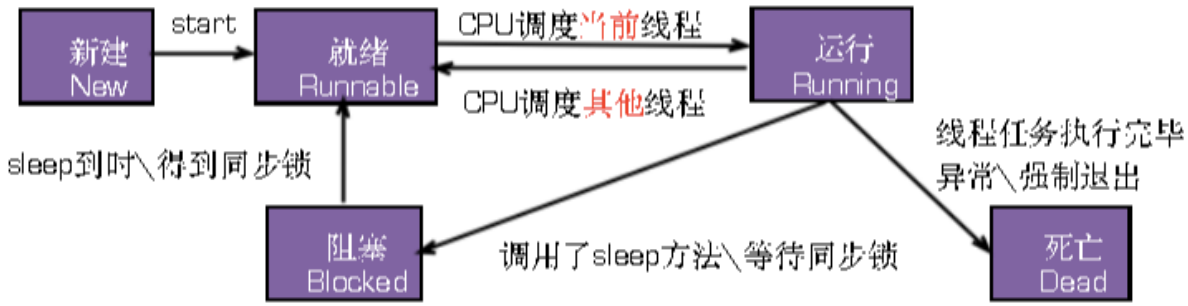
endProgress：动画终点(在整体动画的百分比)。

进程

1. 进程是指在系统中正在运行的一个应用程序。
2. 每个进程之间是独立的，每个进程运行在其自己专用且受保护的内存空间里。

线程

1. 线程是进程的基本执行单元，一个进程的所有任务都在线程中执行。
2. 一个线程中任务的执行是串行(顺序执行)的，也就是说在同一时间内，一个线程只能执行一个任务，例如在一个线程中要下载三个文件只能一个一个的下载。
3. 若要在一个线程中执行多个任务，则只能一个一个地按顺序执行这些任务。



多线程

能适当提高程序的执行效率和资源(CPU，内存)利用率，但是开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），若开启大量的线程会占用大量的内存空间，降低程序的性能。线程越多，CPU在调度线程上的开销也就越大。

1. pthread
   1.1. 简介
      POSIX线程，是线程的POSIX标准。是一套通用的多线程API，可以在Unix/Linux/Mac OS X等系统跨平台使用，Windows中也有其移植版pthread-win32。使用C语言编写，需要程序员自己管理线程的生命周期，在开发中几乎不使用。
   1.2. 使用
      1.2.1. 包含头文件
         #import <pthread.h>
      1.2.2. 创建线程，并开启线程执行任务
         // 定义一个pthread_t类型变量
         pthread_t thread;
         // 创建线程，开启线程，执行任务
         pthread_create(&thread, NULL, run, NULL);
         // 设置子线程的状态为detached，该线程运行结束后会自动释放所有资源
         pthread_detach(thread);
         // run方法
         void *run(void *param) {
            NSLog(@“%@”, [NSThread currentThread]);
            return NULL;
         }
2. NSThread
   2.1. 先创建后启动（可对线程进行更详细的设置，例如线程名字）
      NSThread *thread = [[NSThread alloc] initWithTarget:self selector:
         @selector(download:) object:nil];
      [thread start];
   2.2. 创建完自动启动

[NSThread detachNewThreadSelector:@selector(download:) toTarget:self withObject:nil];

2.3. 隐式创建（自动启动）

[self performSelectorInBackground:@selector(download:) withObject:nil];

3. GCD（核心为任务和队列，即执行什么操作和用来存放任务）

3.1. 同步：在当前线程中执行任务，不具备开启新线程的能力

dispatch_sync(dispatch_queue_t queue, dispatch_block_t block)，queue为队列，block为任务。

3.2. 异步：马上执行，在新的线程中执行任务，具备开启新线程的能力。

dispatch_async(dispatch_queue_t queue, dispatch_block_t block)。

3.3. 并发：多个任务并发(同时)执行(自动开启多个线程同时执行任务，开启线程时间由GCD底层决定)，只在异步下有效。使用dispatch_get_global_queue函数可获得全局并发队列。

3.4. 串行：多个任务一个一个执行，一个任务执行完成后再执行下一个任务，最多开一个线程。使用dispatch_queue_create函数可创建串行队列。

3.5. 主队列：专门负责在主线程上调度任务，即与主线程相关联的队列，不会在子线程调度任务。在主队列不允许开启新线程。主队列是GCD自带的一种特殊的串行队列，放在主队列中的任务都会放到主线程中执行。可使用dispatch_get_main_queue()获得主队列

串行队列同步执行：不开线程，在原线程中一个一个顺序执行。

串行队列异步执行：开一个新线程，在新线程中一个一个顺序执行。

并发队列同步执行：不开线程，在原线程中一个一个顺序执行。

并发队列异步执行：开多个新线程，在新线程中并发(执行先后顺序不确定）执行。

主队列同步执行：死锁。

主队列异步执行：不会开启新线程，先执行主线程中的任务再执行异步队列中任务。

3.6. 只执行一次

```
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
    // 执行代码，这里面默认是线程安全的
});
```

3.7. 队列组

```
// 创建一个队列组
dispatch_group_t group = dispatch_group_create();
// 获取一个全局队列
dispatch_queue_t queue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_async(group, queue, ^{
    // 执行耗时操作A
});
dispatch_group_async(group, queue, ^{
```

```
    // 执行耗时操作B
});
// 监听操作A和操作B执行完后通知执行操作C
dispatch_group_notify(group, queue, ^{
    // 执行操作C
});
```

4. NSOperation

先将需要执行的操作封装到一个NSOperation对象中，然后将NSOperation对象添加到NSOperationQueue中，系统会自动将NSOperationQueue中的NSOperation取出来放到一条新线程中异步执行。

4.1. NSInvocationOperation

```
// 创建NSInvocationOperation对象
- (id)initWithTarget:(id)target selector:(SEL)sel object:(id)arg;
// 调用start方法开始执行操作，一旦执行操作就会调用target的sel方法
- (void)start;
```

注意：默认情况下，调用了start方法后并不会开一条新线程去执行操作，而是在当前线程中同步执行操作，只有将NSOperation放到一个NSOperationQueue中，才会异步执行操作。

4.2. NSBlockOperation

```
// 创建NSBlockOperation对象
+ (id)blockOperationWithBlock:(void (^)(void))block;
// 通过addExecutionBlock：方法添加更多的操作
- (void)addExecutionBlock:(void (^)(void))block;
```

注意：只要NSBlockOperation封装的操作数大于1就会异步执行操作。

4.3. 自定义NSOperation子类重写main方法

经常通过-(BOOL)isCancelled方法检测操作是否被取消，对取消做出响应。

4.4. 添加任务到NSOperationQueue中

```
- (void)addOperation:(NSOperation *)op;
- (void)addOperationWithBlock:(void (^)(void))block;
```

4.5. 设置最大并发数，最大并发数不是线程总数，而是同时执行线程最多个数

```
- (NSInteger)maxConcurrentOperationCount;
- (void)setMaxConcurrentOperationCount:(NSInteger)cnt;
```

4.6. 取消所有操作，正在执行的不会被取消会继续执行完

```
- (void)cancelAllOperations;
// 也可以调用NSOperation的 - (void)cancel方法取消单个操作
```

4.7. 暂停和恢复所有操作

```
- (void)setSuspended:(BOOL)b; // YES代表暂停队列，NO代表恢复队列
```

4.8. 操作之间的依赖，可以跨队列，NSOperation之间可以设置依赖来保证执行顺序，不能相互依赖，如A依赖B，B依赖A。

```
[operationB addDependency:operationA]; // 操作B依赖于操作A，等操作A执行完毕后，才会执行操作B。
```

4.9. 监听操作执行完毕

```
- (void (^)(void))completionBlock;
```

- (void)setCompletionBlock:(void (^)(void))block;

5. 线程间通信
   5.1. thread
   - (void)performSelector:(SEL)aSelector onThread: (NSThread *)thread withObject: (id)arg waitUntilDone: (BOOL)wait;
   5.2. gcd
   dispatch_async(dispatch_get_global_queue(
   DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
   // 执行耗时的异步操作...
   dispatch_async(dispatch_get_main_queue(), ^{
   // 回到主线程，执行UI刷新操作
   });
   });
   5.3. NSOperation
   NSOperationQueue *queue = [[NSOperationQueue alloc] init];
   [queue addOperationWithBlock:^{
   // 1.执行一些比较耗时的操作
   // 2.回到主线程
   [[NSOperationQueue mainQueue] addOperationWithBlock:^{
   // 执行更新操作
   }];
   }];
6. 从其他线程回到主线程
   6.1. perform
   [self performSelectorOnMainThread:<#(SEL)#> withObject:<#(id)#> waitUntilDone:<#(BOOL)#>];
   6.2. gcd
   dispatch_async(dispatch_get_main_queue(), ^{});
   6.3. NSOperation
   [[NSOperationQueue mainQueue] addOperationWithBlock:^{}];
7. 延时执行
   7.1. NSObject
   [self performSelector:@selector(method) withObject:nil afterDelay: 1.0];
   7.2. gcd
   dispatch_after(dispatch_time(DISPATCH_TIME_NOW,
   (int64_t)(1.0 * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
   // 延迟执行代码
   });

Block块

1. self
   在block中使用self有很大风险会造成循环引用，建议使用
   __weak typeof(self) weakSelf = self。
2. 内存

```
// block的内存默认在栈里面(系统自动管理)
void (^test)() = ^{}
// 若对block进行了Copy操作, block的内存会迁移到堆里面(需通过代码管理内存)。
Block_copy(test);
// 在不需要使用block的时候, 应该做1次release操作
Block_release(test);
[test release];
```

Reachability

1. 监测联网状态

```
Reachability *reach = [Reachability
    reachabilityWithHostName:@"www.baidu.com"];
```

2. 判断reach.currentReachabilityStatus值
3. 利用通知中心实时监听联网状态

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(networkStatusChanged)
    name:kReachabilityChangedNotification object:nil];
self.reachability = [Reachability reachabilityForInternetConnection];
[self.reachability startNotifier];
```

SDWebImage

1. 默认缓存时长一周，**SDImageCacheConfig**中

```
static const NSInteger kDefaultCacheMaxCacheAge = 60 * 60 * 24 * 7; // 1 week
```

2. 防止**url**对应图片重复下载：先看**icon->**再看内存缓存**->**沙盒缓存，如果已经存在于下载队列中就不再添加到里面。

ip查询

1. 获得本机真实ip地址：https://httpbin.org/ip, 通常是接入家庭的光猫ip地址得到的
2. 查询ip详细信息：http://ip.taobao.com/service/getIpInfo.php?ip=171.113.94.94，查询ip对应的物理位置

规档和解档

1. 实现NSCoding协议
2. 实现-encodeWithCoder:(NSCoder *)encoder；和

```
- (instancetype)initWithCoder:(NSCoder *)aDecoder;
```

3. 使用NSKeyedArchiver和NSKeyedUnarchiver分别归/解档

JSON

1. 定义

   是一种轻量级的数据格式，一般用于数据交互。本质上是一个特殊格式的非NSString字符串。

2. JSON解析

   2.1. 第三方框架：JSONKit > SBJson > TouchJSON (性能从左到右，越差)

   2.2. 苹果原生NSJSONSerialization(性能最好)

      2.2.1. JSON数据 —> OC对象

```
+ (id)JSONObjectWithData:(NSData *)data options:
    (NSJSONReadingOptions)opt error:(NSError **)error;
```

      2.2.2. OC对象 —> JSON数据

```
+ (NSData *)dataWithJSONObject:(id)obj options:
    (NSJSONWritingOptions)opt error:(NSError **)error;
```

XML

1. 定义

   和JSON一样，也是常用的一种用于交互的数据格式。

2. 文档声明

   在XML文档的最前面，必须编写一个文档声明，用来声明XML文档的类型，如<?xml version="1.0" encoding="UTF-8" ?> // encoding属性说明文档的字符编码

3. XML解析

   3.1. 第三方框架

      3.1.1. libxml2

         纯C语言，默认包含在iOS SDK中，同时支持DOM和SAX方式解析。

      3.1.2 GDataXML

         DOM方式解析，由Google开发，基于libxml2。

   3.2. 苹果原生NSXMLParser(SAX方式解析，使用简单)

      3.2.1. 使用步骤

```
// 传入XML数据，创建解析器
NSXMLParser *parser = [[NSXMLParser alloc] initWithData:data];
// 设置代理，监听解析过程
parser.delegate = self;
// 开始解析
[parser parse];
```

      3.2.2. NSXMLParserDelegate

```
// 当扫描到文档的开始时调用(开始解析)
- (void)parserDidStartDocument:(NSXMLParser *)parser;
// 当扫描到文档的结束时调用(解析完毕)
- (void)parserDidEndDocument:(NSXMLParser *)parser;
// 当扫描到元素的开始时调用(attributeDict存放着元素的属性)
- (void)parser:(NSXMLParser *)parser didStartElement:
    (NSString *)elementName namespaceURI:(NSString *)namespaceURI
```

```
    qualifiedName:(NSString *)qName attributes:(NSDictionary *)attributeDict
// 当扫描到元素的结束时调用
- (void)parser:(NSXMLParser *)parser didEndElement:
    (NSString *)elementName namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName
```

Cookie

1. 定义

   由服务器端生成，发送给客户端，客户端将Cookie的Key/value保存到某个目录下的文本文件内。如果客户端支持Cookie，下次请求同一网站时就可以将Cookie直接发送给服务器。Cookie名称和值由服务器端开发自己定义。iOS程序中默认支持Cookie，不需任何处理，若服务器返回Cookie，会自动保存在沙盒的Library/Cookies目录中。

2. 读取Cookie内容

```
// 读取所有cookie
NSArray *cookies = [[NSHTTPCookieStorage sharedHTTPCookieStorage]
    cookies];
for (NSHTTPCookie *cookie in cookies) {
    if ([cookie.name isEqualToString:@“userName”]) {
        self.username.text = cookie.value;
    }
    if ([cookie.name isEqualToString:@“userPassword”]) {
        self.password.text = cookie.value;
    }
}
```

3. 删除Cookie

```
NSArray *cookies = [[NSHTTPCookieStorage sharedHTTPCookieStorage]
    cookies];
for (NSHTTPCookie *cookie in cookies) {
    [[NSHTTPCookieStorage sharedHTTPCookieStorage] deleteCookie: cookie];
}
```

4. 缺陷

   4.1. 会被附加在每个HTTP请求中，会增加额外的流量。

   4.2. 在HTTP请求中的Cookie是明文传递的，会有安全隐患，故建议使用HTTPS。

   4.3. Cookie的大小限制在4KB左右，不适合存储复杂的数据信息。

HTTP协议

1. 定义

   Hypertext Transfer Protocol超文本传输协议，访问远程网络资源，格式是http://，通信速度很快，允许传输任意类型的数据。

2. 请求

   2.1. 请求行

包含了请求方法、请求资源路径、HTTP协议版本 (GET /aa/a.png HTTP/1.1)

2.2. 请求头

包含了对客户端的环境描述、客户端请求的主机地址等信息

Host: 192.168.1.101:8080 // 客户端想访问的服务器主机地址

User-Agent: Mac+OS+X/10.13 (17A405) CalendarAgent/399 // 客户端类型和软件环境

Accept: text/html, */* // 客户端所能接收的数据类型

Accept-Language: zh-cn // 客户端的语言环境

Accept-Encoding: br, gzip, deflate // 客户端支持的数据压缩格式

2.3. 请求体

客户端发给服务器的具体数据。

3. 响应

3.1. 状态行

包含了HTTP协议版本、状态码、状态英文名称(HTTP/1.1 200 OK)

3.2. 响应头

包含了对服务器的描述、对返回数据的描述

Server: Apache-Coyote/1.1 // 服务器的类型

Content-Type: image/jpeg // 放回数据的类型

Content-Length: 56811 // 返回数据的长度

Date: Mon,23 Jun 2017 12:22:43 GMT // 响应的时间

3.3. 实体内容

服务器返回给客户端的具体数据

资源打包

1. 图片被放到Image.xcassets里面

1.1. ios8及以上，打包的资源包中的图片会被放到Assets.car中，图片有被压缩。

1.2. ios8以下，打包的资源包中的图片会被放在MainBundle中，图片没有被压缩。

2. 非Image.xcassets中

图片会直接暴露在沙盒的资源包(MainBundle)中，不会压缩到Assets.car中。

3. 压缩到Assets.car中的图片只能通过图片名imageNamed:来加载，直接暴露在沙盒的资源包(MainBundle)中的图片可通过全路径imageWithContentsOfFile：来加载。

内存泄漏分析

1. 内存泄漏

堆里不再使用的对象，没有被销毁，依然占据着内存。简单理解就是不再使用的对象没有被释放。

2. 内存溢出
    2.1. 内存不够用了。
    2.2. 数据长度比较小的数据类型存储了数据长度比较大的数据。
3. 静态内存分析
    3.1. 不运行程序，直接根据程序的语法结果进行分析Product -- > Analyze。
    3.2. 分析内存非常快，可以对整个项目的内存进行分析。但相对不太准确，若有提示可能出现内存泄漏则需要依据实际情况查看分析是否有内存泄漏。
4. 动态内存分析
    4.1. 需要运行程序，可对某一个操作来具体分析。可以查看做出了某个操作后(比如点击了某个按钮/显示了某个控制器)，内存是否有暴增的情况(突然变化)，比较准确。Product --> Profile -->Leaks --> 选择想要运行的项目 --> Record。
    4.2. 分析速度比较慢，需一步一步分析，分析过程中可能会有遗漏代码。
5. 减少内存泄漏
    5.1. 非ARC
        5.1.1. Foundation对象(OC对象)
            只要方法中包含了alloc\new\copy\mutableCopy\retain等关键字, 那么这些方法产生的对象, 就必须在不再使用的时候调用1次release或者1次autorelease
        5.1.2. CoreFoundation对象(C对象)
            只要函数中包含了create\new\copy\retain等关键字, 那么这些方法产生的对象, 就必须在不再使用的时候调用1次CFRelease或者其他release函数
    5.2. ARC
        5.2.1. Foundation对象(OC对象)
            会自动管理OC对象，但不会自动管理C对象
        5.2.2. CoreFoundation对象(C对象)
            只要函数中包含了create\new\copy\retain等关键字, 那么这些方法产生的对象就必须在不再使用的时候调用1次CFRelease或者其他release函数

闭源库

1. 定义
    不公开源代码，是经过编译后的二进制文件，看不到具体实现
2. 静态库
    2.1. 存在形式
        .a和.framework
    2.2. 使用
        链接时，静态库会被完整的复制到可执行文件中，被多次使用就有多份冗余拷贝
3. 动态库
    3.1. 存在形式
        .dylib和.framework
    3.2. 使用

链接时不复制，程序运行时由系统动态加载到内存，供程序调用，系统只加载一次，多个程序共用，节省内存。

4. 常用命令

4.1. 检测库的类型及支持的CPU框架

lipo -info 库名

4.2. 合并（也可以将Build Active Architecture Only的YES改为NO）

```
lipo -create Debug-iphoneos/libTools.a Debug-iphonesimulator/libTools.a
    -output libTools.a
```

UIDynamic

1. 定义

是从iOS7开始引入的一种新技术，隶属于UIKit框架。可认为是一种物理引擎，能模拟和仿真现实生活中的物理现象。

2. 使用步骤

2.1. 创建一个物理仿真器，顺便指定仿真范围

```
UIDynamicAnimator *animator = [[UIDynamicAnimator alloc]
initWithReferenceView:self.view];
```

2.2. 创建相应的物理仿真行为，顺便添加物理仿真元素

```
UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
initWithItems:@[self.redView]];
```

2.3. 将物理仿真行为添加到物理仿真器中，开始仿真

```
[animator addBehavior:gravity];
```

3. 三大概念

3.1. 物理仿真元素(Dynamic Item)

谁要进行物理仿真。任何遵守了UIDynamicItem协议的对象，UIView和UICollectionViewLayoutAttributes类都已经默认遵守了UIDynamicItem协议。

3.2. 物理仿真行为(Dynamic Behavior)

执行怎样的物理仿真效果，怎样的动画效果。下述所有的物理仿真行为都继承自UIDynamicBehavior，所有的UIDynamicBehavior都可以独立进行。

3.2.1. UIGravityBehavior

重力行为，给定重力方向、加速度、让物体朝着重力方向掉落。

3.2.2. UICollisionBehavior

碰撞行为，可以让物体之间实现碰撞效果，可通过添加边界(boundary)让物理碰撞局限在某个空间中。

3.2.3. UISnapBehavior

捕捉行为，可以让物体迅速冲到某个位置(捕捉位置)，捕捉到位置后会带有一定的震动，若要进行连续的捕捉行为，需要先把前面的捕捉行为从物理仿真器中移除。

3.2.4. UIPushBehavior: 推动行为

3.2.5. UIAttachmentBehavior: 附着行为

3.2.6. UIDynamicItemBehavior: 动力元素行为。

3.3. 物理仿真器(Dynamic Animator)

让物理仿真元素执行具体的物理仿真行为。是UIDynamicAnimator类型的对象。

初始化: - (instancetype)initWithReferenceView:(UIView *)view;其中view参数是一个参照视图，表示物理仿真的范围。

注意：不是任何对象都能做物理仿真元素，不是任何对象都能进行物理仿真。

AutoLayout

1. 定义

是一种“自动布局”技术，专门用来布局UI界面。自iOS6开始引入，自iOS7(Xcode5)开始得到较大推广。

2. 核心

2.1. 参照

2.2. 约束

3. 警告和错误

3.1. 警告

控件的frame不匹配所添加的约束，比如约束控件的宽度为100，而控件现在的宽度是110。

3.2. 错误

3.2.1. 缺乏必要的约束，比如只约束了宽度和高度，没有约束具体的位置。

3.2.2. 两个约束冲突，比如1个约束控件的宽度为100，1个约束控件的宽度为110

4. 添加约束的规则

在创建约束之后，需要将其添加到作用的view上，在添加时要注意目标view需要遵循以下规则：

4.1. 对于两个同层级view(兄弟view)之间的约束关系，添加到它们的父view上。

4.2. 对于两个不同层级view之间的约束关系，添加到它们最近的共同父view上。

4.3. 对于有层次关系的两个view之间的约束关系，添加到层级较高的父view上。

总之对于两个view之间的约束关系添加到它们最近的一个共同祖先view上。

5. 代码实现

5.1. 利用NSLayoutConstraint类创建具体的约束对象

+ (id)constraintWithItem:(id)view1 attribute:(NSLayoutAttribute)attr1 relatedBy:(NSLayoutRelation)relation toItem:(id)view2 attribute:(NSLayoutAttribute)attr2 multiplier:(CGFloat)multiplier constant:(CGFloat)c;

其中view1为要约束的控件，attr1约束的类型(做怎样的约束)，releation与参照控件之间的关系，view2参照的控件，attr2约束的类型，multiplier乘数，c常量。

5.2. 添加约束对象到相应的view上

- (void)addConstraint:(NSLayoutConstraint *)constraint;

- (void)addConstraints:(NSArray *)constraints;

注意点：

a. 要先禁止autoresizing功能，设置view下面属性为NO

view.translatesAutoresizingMaskIntoConstraints = NO;

b. 添加约束前，一定要保证相关控件都已经在各自的父控件上。
c. 不用再给view设置frame。

6. VFL
6.1. 定义
VFL全称是Visual Format Language，可视化格式语言，是苹果公司为了简化Autolayout的编码而推出的抽象语言。
6.2. 示例
6.2.1. H:[cancelButton[72]]-12-[acceptButton[50]]
cancelButton宽72、acceptButton宽50、它们之间间距12。
6.2.2. H:[wideView[>=60@700]]
wideView宽度大于等于60point、该约束条件优先级为700(优先级最大值为1000、优先级越高的约束越先被满足)。
6.2.3. V:[redBox][yellowBox(==redBox)]
竖直方向上先有一个redBox、其下方紧接一个高度等于redBox高度的yellowBox。
6.2.4. H:|-10-[Find]-[FindNext]-[FindField[>=20]]-|
水平方向上、Find距离父view左边缘默认间隔宽度、之后是FindNext距离Find间隔默认宽度、再之后是宽度不小于20的FindField、它和FindNext以及父view右边缘的间距都是默认宽度。(竖线“|”表示superview的边缘)
6.3. 使用VFL来创建约束数组
+ (NSArray *)constraintsWithVisualFormat:(NSString *)format options:(NSLayoutFormatOptions)opts metrics:(NSDictionary *)metrics views:(NSDictionary *)views;
其中format为VFL语句，opts约束类型，metrics为VFL语句中用到的具体数值，views为VFL语句中用到的控件。

7. 基于Autolayout的动画
在修改了约束之后，只要执行下面代码，就能做动画效果
[UIView animateWithDuration:1.0 animations:^{
[添加了约束的view layoutIfNeeded];
}];

传感器

1. 定义
传感器是一种感应/检测装置。
2. 作用
用于感应/检测设备周边的信息。不同类型的传感器，检测的信息也不一样。
3. 应用场景
3.1. 在地图应用中，能判断出手机头面向的方向。
3.2. 开关灯，iPhone会自动调节亮度让屏幕显得不是那么刺眼。
3.3. 打电话时，人脸贴近iPhone屏幕时，屏幕会自动锁屏，达到省电的目的。
4. 类型

4.1. 环境光传感器(Ambient Light Sensor)

随着环境的变化设备会自动调节亮度。当使用iPhone拍照时，闪光灯会在一定条件下自动开启。当周围光线弱到一定条件时，Mac会自动开启键盘背光。

4.2. 距离传感器(Proximity Sensor)

用于检测是否有其他物体靠近设备屏幕。当打接电话将电话屏幕贴近耳边，iPhone会自动关闭屏幕以节省电量和防止耳朵或面部不小心触摸屏幕而引发一些不想要的意外操作。

```
// 开启距离感应功能
[UIDevice currentDevice].proximityMonitoringEnabled = YES;
// 监听距离感应的通知
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(
proximityChange:) name:UIDeviceProximityStateDidChangeNotification
object:nil];
// 距离更改执行方法
- (void)proximityChange:(NSNotificationCenter *)notification {
    if ([UIDevice currentDevice].proximityState == YES) {
        NSLog(@“某个物体靠近了设备屏幕”); // 屏幕会自动锁住
    } else {
        NSLog(@“某个物体远离了设备屏幕”); // 屏幕会自动解锁
    }
}
```

4.3. 磁力计传感器(Magnetometer Sensor)

可感应地球磁场，获得方向信息，使位置服务数据更精准。可用于电子罗盘和导航应用。

4.4. 内部问题传感器(Internal Temperature Sensor)

用于检测内部组件温度，当温度超过系统设定的阀值时会出现警告提示。

4.5. 湿度传感器(Moisture Sensor)

是一个简单的物理传感器，简单来说湿度传感器就是一张遇水变红的试纸。

4.6. 陀螺仪(Gyroscope)

可用于检测设备的持握方式，陀螺仪的原理是检测设备在X、Y、Z轴上所旋转的角速度。

4.7. 运动传感器/加速度传感器(Motion/Accelerometer Sensor)，即加速计

加速计用于检测设备在X、Y、Z轴上的加速度。感应设备的运动，常用于摇一摇和计步器。

4.7.1. UIAccelerometer

iOS4以前，iOS5已过期，用法简单，部分程序还有残留。

```
// 获得单例对象
UIAccelerometer *accelerometer = [UIAccelerometer sharedAccelerometer];
// 设置代理
accelerometer.delegate = self;
// 设置采样间隔
accelerometer.updateInterval = 1.0 / 30.0; // 1秒钟采样30次
```

```
// 实现代理方法，accelerometer中x、y、z分别代表每个轴上的加速度
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
```

4.7.2. CoreMotion

iOS4以后，不仅提供了实时的加速度值和旋转速度值，还集成了很多好的算法。可通过push(实时采集所有数据，采集频率高)和pull(在有需要的时候，再主动去采集数据)两种方式获取数据。

```
// 创建运动管理者对象
CMMotionManager *mgr = [[CMMotionManager alloc] init];
// 判断加速计是否可用
if (mgr.isAccelerometerAvailable) {
    // 加速计可用
}
// pull
// 开始采样
- (void)startAccelerometerUpdates;
// 在需要的时候采集加速度数据
CMAcceleration acc = mgr.accelerometerData.acceleration;

// push
// 设置采样间隔
mgr.accelerometerUpdateInterval = 1.0 / 30.0;
// 开始采样，采样到数据就会调用handler、handler会在queue中执行
- (void)startAccelerometerUpdatesToQueue:(NSOperationQueue *)queue
withHandler:(CMAccelerometerHandler)handler;
```