那些著名或非著名的iOS面试题 – 前编

1.如何追踪app崩溃率，如何解决线上闪退

当iOS设备上的App应用闪退时，操作系统会生成一个crash日志，保存在设备上。crash日志上有很多有用的信息，比如每个正在执行线程的完整堆栈跟踪信息和内存映像，这样就能够通过解析这些信息进而定位crash发生时的代码逻辑，从而找到App闪退的原因。通常来说，crash产生来源于两种问题：违反iOS系统规则导致的crash和App代码逻辑BUG导致的crash，下面分别对他们进行分析。

违反iOS系统规则产生crash的三种类型

(1) 内存报警闪退当iOS检测到内存过低时，它的VM系统会发出低内存警告通知，尝试回收一些内存；如果情况没有得到足够的改善，iOS会终止后台应用以回收更多内存；最后，如果内存还是不足，那么正在运行的应用可能会被终止掉。在Debug模式下，可以主动将客户端执行的动作逻辑写入一个log文件中，这样程序童鞋可以将内存预警的逻辑写入该log文件，当发生如下截图中的内存报警时，就是提醒当前客户端性能内存吃紧，可以通过Instruments工具中的Allocations 和 Leaks模块库来发现内存分配问题和内存泄漏问题。

(2) 响应超时当应用程序对一些特定的事件（比如启动、挂起、恢复、结束）响应不及时，苹果的Watchdog机制会把应用程序干掉，并生成一份相应的crash日志。这些事件与下列UIApplicationDelegate方法相对应，当遇到Watchdog日志时，可以检查上图中的几个方法是否有比较重的阻塞UI的动作。

```
application:didFinishLaunchingWithOptions:
applicationWillResignActive:
applicationDidEnterBackground:
applicationWillEnterForeground:
applicationDidBecomeActive:
applicationWillTerminate:
```

(3) 用户强制退出一看到“用户强制退出”，首先可能想到的双击Home键，然后关闭应用程序。不过这种场景一般是不会产生crash日志的，因为双击Home键后，所有的应用程序都处于后台状态，而iOS随时都有可能关闭后台进程，当应用阻塞界面并停止响应时这种场景才会产生crash日志。这里指的“用户强制退出”场景，是稍微比较复杂点的操作：先按住电源键，直到出现“滑动关机”的界面时，再按住Home键，这时候当前应用程序会被终止掉，并且产生一份相应事件的crash日志。

应用逻辑的Bug

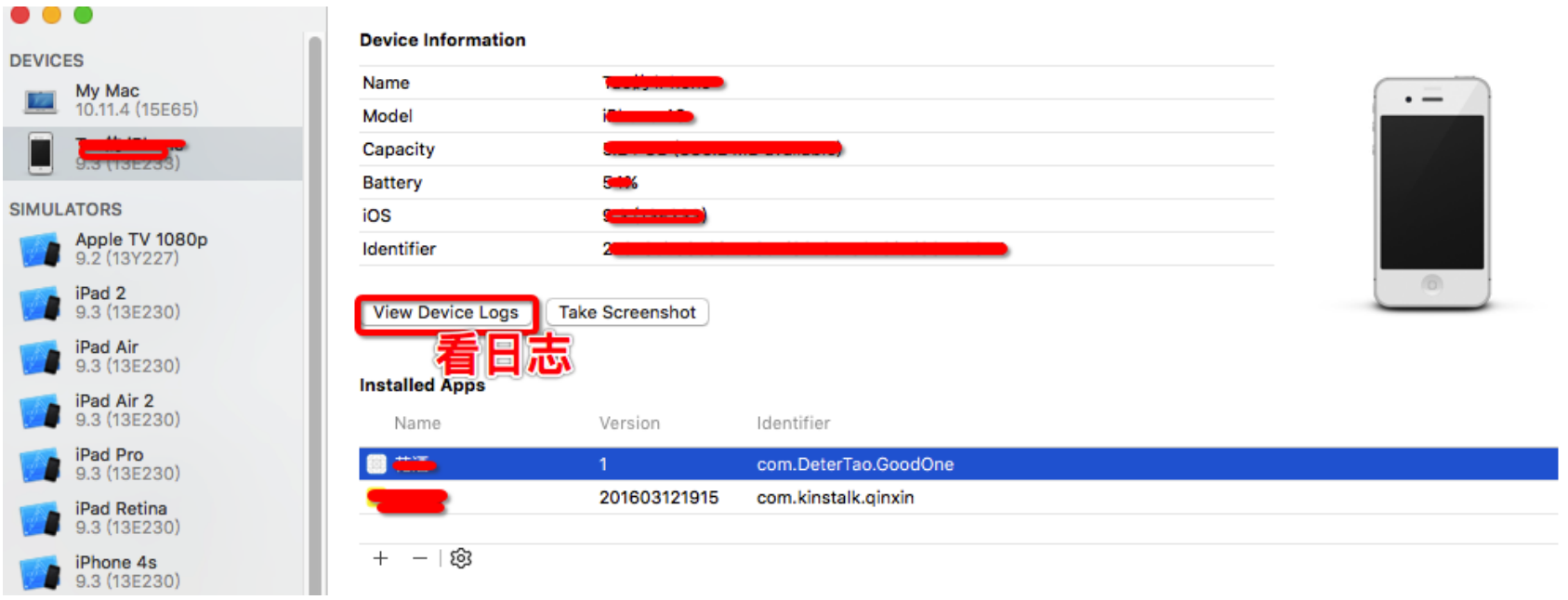
大多数闪退崩溃日志的产生都是因为应用中的Bug，这种Bug的错误种类有很多，比如

SEGV：（Segmentation Violation，段违例），无效内存地址，比如空指针，未初始化指针，栈溢出等；
SIGABRT：收到Abort信号，可能自身调用abort()或者收到外部发送过来的信号；
SIGBUS：总线错误。与SIGSEGV不同的是，SIGSEGV访问的是无效地址（比如虚存映射不到物理内存），而SIGBUS
SIGILL：尝试执行非法的指令，可能不被识别或者没有权限；
SIGFPE：Floating Point Error，数学计算相关问题（可能不限于浮点计算），比如除零操作；
SIGPIPE：管道另一端没有进程接手数据；

常见的崩溃原因基本都是代码逻辑问题或资源问题，比如数组越界，访问野指针或者资源不存在，或资源大小写错误等。

crash的收集

如果是在windows上你可以通过itools或pp助手等辅助工具查看系统产生的历史crash日志，然后再根据app来查看。如果是在Mac 系统上，只需要打开xcode->windows->devices，选择device logs进行查看，如下图，这些crash文件都可以导出来，然后再单独对这个crash文件做处理分析。



看日志

市场上已有的商业软件提供crash收集服务，这些软件基本都提供了日志存储，日志符号化解析和服务端可视化管理等服务：

Crashlytics (www.crashlytics.com)
Crittercism (www.crittercism.com)
Bugsense (www.bugsense.com)
HockeyApp (www.hockeyapp.net)
Flurry(www.flurry.com)

开源的软件也可以拿来收集crash日志，比如Razor,QuincyKit（git链接）等，这些软件收集crash的原理其实大同小异，都是根据系统产生的crash日志进行了一次提取或封装，然后将封装后的crash文件上传到对应的服务端进行解析处理。很多商业软件都采用了Plcrashreporter这个开源工具来上传和解析crash，比如HockeyApp,Flurry和crittercism等。

```
Incident Identifier: E838FEFB-ECF6-498C-8B35-D40F0F9FEAE4
CrashReporter Key:   TODO
Hardware Model:      iPhone3,1
Process:         TestFlight [502]
Path:            /var/mobile/Applications/BBDF3F15-3BFD-418A-A120-68C3BB24DD22/TestFlight.app/TestFlight
Identifier:      com.netease.CITestFlight
Version:         1.0
Code Type:       ARM
Parent Process:  launchd [1]

Date/Time:       2014-04-28 06:12:43 +0000
OS Version:      iPhone OS 7.0.4 (11B554a)
Report Version:  104

Exception Type:  SIGABRT
Exception Codes: #0 at 0x3b54f1fc
Crashed Thread:  0

Thread 0 Crashed:
0   libsystem_kernel.dylib          0x3b54f1fc __pthread_kill + 8
1   libsystem_c.dylib               0x3b4ffffd abort + 77
2   libc++abi.dylib                 0x3a82ecd7 abort_message + 75
3   libc++abi.dylib                 0x3a8476e5 default_terminate_handler() + 253
4   libobjc.A.dylib                 0x3af90921 _objc_terminate() + 193
5   libc++abi.dylib                 0x3a8451c7 std::__terminate(void (*)()) + 79
6   libc++abi.dylib                 0x3a844d2d __cxa_increment_exception_refcount + 1
7   libobjc.A.dylib                 0x3af907f7 objc_exception_rethrow + 43
8   CoreFoundation                  0x30b34c9d CFRunLoopRunSpecific + 641
9   CoreFoundation                  0x30b34a0b CFRunLoopRunInMode + 107
10  GraphicsServices                0x35813283 GSEventRunModal + 139
11  UIKit                           0x333d8049 UIApplicationMain + 1137
12  TestFlight                      0x00068b19 0x36000 + 207641
13  libdyld.dylib                   0x3b498ab7 start + 3
...

Binary Images:
    0x36000 -    0x85fff +TestFlight armv7  <4a42d422a46338aa56e88840
    68C3BB24DD22/TestFlight.app/TestFlight
0x2fbcf000 - 0x2fbcffff  Accelerate armv7  <8e17835efc9234da89e3080c4
0x2fbd9000 - 0x2fda5fff  vImage armv7  <a0e38e04253d345684ae9b6020775c4
framework/Frameworks/vImage.framework/vImage
```

crash信息

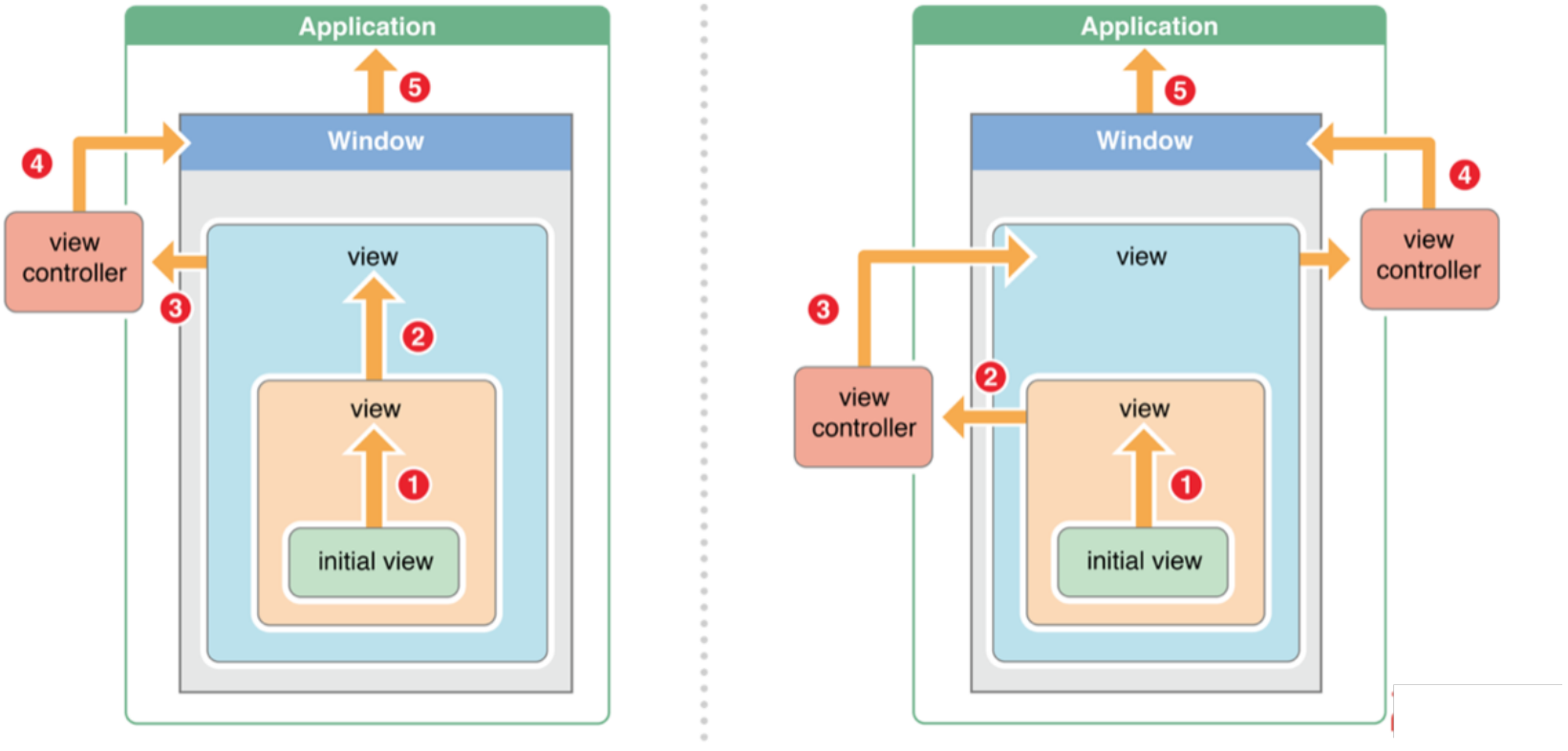
由于自己的crash信息太长，找了一张示例：

1)crash标识是应用进程产生crash时的一些标识信息，它描述了该crash的唯一标识（E838FEFB-ECF6-498C-8B35-D40F0F9FEAE4），所发生的硬件设备类型（iphone3,1代表iphone4），以及App进程相关的信息等；
2）基本信息描述的是crash发生的时间和系统版本；
3）异常类型描述的是crash发生时抛出的异常类型和错误码；
4）线程回溯描述了crash发生时所有线程的回溯信息，每个线程在每一帧对应的函数调用信息（这里由于空间限制没有全部列出）；
5）二进制映像是指crash发生时已加载的二进制文件。以上就是一份crash日志包含的所有信息，接下来就需要根据这些信息去解析定位导致crash发生的代码逻辑， 这就需要用到符号化解析的过程（洋名叫：symbolication)。

解决线上闪退
首先保证，发布前充分测试。发布后依然有闪退现象，查看崩溃日志，及时修复并发布。

2.什么是事件响应链，点击屏幕时是如何互动的，事件的传递。

- 响应者链条：是由多个响应者对象连接起来的链条
- 作用：能很清楚的看见每个响应者之间的联系，并且可以让一个事件多个对象处理。
- 响应者对象：能处理事件的对象



事件响应链

对于IOS设备用户来说，他们操作设备的方式主要有三种：触摸屏幕、晃动设备、通过遥控设施控制设备。对应的事件类型有以下三种：

1、触屏事件（Touch Event）

2、运动事件（Motion Event）

3、远端控制事件（Remote-Control Event）

响应者链（Responder Chain）
响应者对象（Responder Object），指的是有响应和处理事件能力的对象。响应者链就是由一系列的响应者对象构成的一个层次结构。

UIResponder是所有响应对象的基类，在UIResponder类中定义了处理上述各种事件的接口。我们熟悉的UIApplication、 UIViewController、UIWindow和所有继承自UIView的UIKit类都直接或间接的继承自UIResponder，所以它们的实例都是可以构成响应者链的响应者对象。

响应者链有以下特点：
1、响应者链通常是由视图（UIView）构成的；
2、一个视图的下一个响应者是它视图控制器（UIViewController）（如果有的话），然后再转给它的父视图（Super View）；
3、视图控制器（如果有的话）的下一个响应者为其管理的视图的父视图；
4、单例的窗口（UIWindow）的内容视图将指向窗口本身作为它的下一个响应者

需要指出的是，Cocoa Touch应用不像Cocoa应用，它只有一个UIWindow对象，因此整个响应者链要简单一点；
5、单例的应用（UIApplication）是一个响应者链的终点，它的下一个响应者指向nil，以结束整个循环。

点击屏幕时是如何互动的
iOS系统检测到手指触摸(Touch)操作时会将其打包成一个UIEvent对象，并放入当前活动Application的事件队列，单例的UIApplication会从事件队列中取出触摸事件并传递给单例的UIWindow来处理，UIWindow对象首先会使用hitTest:withEvent:方法寻找此次Touch操作初始点所在的视图(View)，即需要将触摸事件传递给其处理的视图，这个过程称之为hit-test view。

UIWindow实例对象会首先在它的内容视图上调用hitTest:withEvent:，此方法会在其视图层级结构中的每个视图上调用pointInside:withEvent:（该方法用来判断点击事件发生的位置是否处于当前视图范围内，以确定用户是不是点击了当前视图），如果pointInside:withEvent:返回YES，则继续逐级调用，直到找到touch操作发生的位置，这个视图也就是要找的hit-test view。

hitTest:withEvent:方法的处理流程如下:首先调用当前视图的pointInside:withEvent:方法判断触摸点是否在当前视图内；若返回NO,则hitTest:withEvent:返回nil;若返回YES,则向当前视图的所有子视图(subviews)发送hitTest:withEvent:消息，所有子视图的遍历顺序是从最顶层视图一直到到最底层视图，即从subviews数组的末尾向前遍历，直到有子视图返回非空对象或者全部子视图遍历完毕；若第一次有子视图返回非空对象，则hitTest:withEvent:方法返回此对象，处理结束；如所有子视图都返回非，则hitTest:withEvent:方法返回自身(self)。

事件的传递和响应分两个链：

传递链：由系统向离用户最近的view传递。UIKit –> active app’s event queue –> window –> root view –>……–>lowest view响应链：由离用户最近的view向系统传递。initial view –> super view –> …..–> view controller –> window –> Application

3.Run Loop是什么，使用的目的，何时使用和关注点

Run Loop是一让线程能随时处理事件但不退出的机制。RunLoop 实际上是一个对象，这个对象管理了其需要处理的事件和消息，并提供了一个入口函数来执行Event Loop 的逻辑。线程执行了这个函数后，就会一直处于这个函数内部 "接受消息->等待->处理" 的循环中，直到这个循环结束（比如传入 quit 的消息），函数返回。让线程在没有处理消息时休眠以避免资源占用、在有消息到来时立刻被唤醒。

OSX/iOS 系统中，提供了两个这样的对象：NSRunLoop 和 CFRunLoopRef。CFRunLoopRef 是在 CoreFoundation 框架内的，它提供了纯 C 函数的 API，所有这些 API 都是线程安全的。NSRunLoop 是基于 CFRunLoopRef 的封装，提供了面向对象的

API，但是这些 API 不是线程安全的。

线程和 RunLoop 之间是一一对应的，其关系是保存在一个全局的 Dictionary 里。线程刚创建时并没有 RunLoop，如果你不主动获取，那它一直都不会有。RunLoop 的创建是发生在第一次获取时，RunLoop 的销毁是发生在线程结束时。你只能在一个线程的内部获取其 RunLoop（主线程除外）。

系统默认注册了5个Mode:

1. kCFRunLoopDefaultMode: App的默认 Mode，通常主线程是在这个 Mode 下运行的。
2. UITrackingRunLoopMode: 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响。
3. UIInitializationRunLoopMode: 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用。
4. GSEventReceiveRunLoopMode: 接受系统事件的内部 Mode，通常用不到。
5. kCFRunLoopCommonModes: 这是一个占位的 Mode，没有实际作用。

Run Loop的四个作用:

使程序一直运行接受用户输入
决定程序在何时应该处理哪些Event
调用解耦
节省CPU时间

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，这就解释了本文开始说的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。

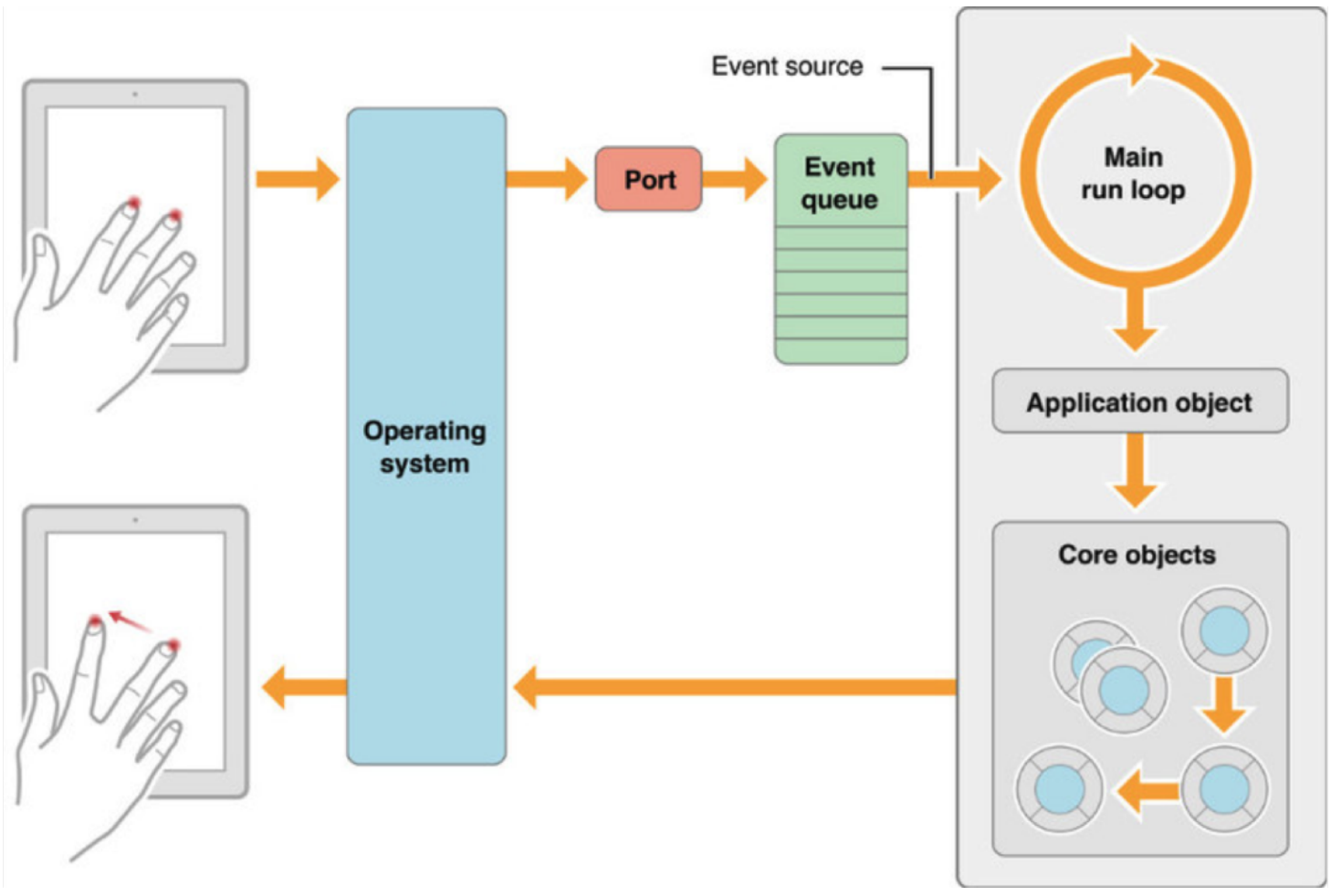
对其它线程来说，run loop默认是没有启动的，如果你需要更多的线程交互则可以手动配置和启动，如果线程只是去执行一个长时间的已确定的任务则不需要。在任何一个Cocoa程序的线程中，都可以通过：

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

来获取到当前线程的run loop。

一个run loop就是一个事件处理循环，用来不停的监听和处理输入事件并将其分配到对应的目标上进行处理。

NSRunLoop是一种更加高明的消息处理模式，他就高明在对消息处理过程进行了更好的抽象和封装，这样才能是的你不用处理一些很琐碎很低层次的具体消息的处理，在NSRunLoop中每一个消息就被打包在input source或者是timer source中了。使用run loop可以使你的线程在有工作的时候工作，没有工作的时候休眠，这可以大大节省系统资源。



RunLoop

什么时候使用run loop

仅当在为你的程序创建辅助线程的时候，你才需要显式运行一个run loop。Run loop是程序主线程基础设施的关键部分。所以，Cocoa和Carbon程序提供了代码运行主程序的循环并自动启动run loop。IOS程序中UIApplication的run方法（或Mac OS X中的NSApplication）作为程序启动步骤的一部分，它在程序正常启动的时候就会启动程序的主循环。类似的，RunApplicationEventLoop函数为Carbon程序启动主循环。如果你使用xcode提供的模板创建你的程序，那你永远不需要自己去显式的调用这些例程。

对于辅助线程，你需要判断一个run loop是否是必须的。如果是必须的，那么你要自己配置并启动它。你不需要在任何情况下都去启动一个线程的run loop。比如，你使用线程来处理一个预先定义的长时间运行的任务时，你应该避免启动run loop。Run loop在你要和线程有更多的交互时才需要，比如以下情况：

使用端口或自定义输入源来和其他线程通信
使用线程的定时器
Cocoa中使用任何performSelector…的方法
使线程周期性工作

关注点

1. Cocoa中的NSRunLoop类并不是线程安全的我们不能再一个线程中去操作另外一个线程的run loop对象，那很可能会造成意想不到的后果。不过幸运的是CoreFundation中的不透明类CFRunLoopRef是线程安全的，而且两种类型的run loop完全可以混合使用。Cocoa中的NSRunLoop类可以通过实例方法：

   ```
   - (CFRunLoopRef)getCFRunLoop;
   ```

   获取对应的CFRunLoopRef类，来达到线程安全的目的。

2. Run loop的管理并不完全是自动的。我们仍必须设计线程代码以在适当的时候启动run loop并正确响应输入事件，当然前提是线程中需要用到run loop。而且，我们还需要使用while/for语句来驱动run loop能够循环运行，下面的代码就成功驱动了一个run loop：

   ```
   BOOL isRunning = NO;
   do {
        isRunning = [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDa
   } while (isRunning);
   ```

3. Run loop同时也负责autorelease pool的创建和释放在使用手动的内存管理方式的项目中，会经常用到很多自动释放的对象，如果这些对象不能够被即时释放掉，会造成内存占用量急剧增大。Run loop就为我们做了这样的工作，每当一个运行循环结束的时候，它都会释放一次autorelease pool，同时pool中的所有自动释放类型变量都会被释放掉。

4. ARC和MRC

Objective-c中提供了两种内存管理机制MRC（MannulReference Counting）和ARC(Automatic Reference Counting)，分别提供对内存的手动和自动管理，来满足不同的需求。Xcode 4.1及其以前版本没有ARC。

在MRC的内存管理模式下，与对变量的管理相关的方法有：retain,release和autorelease。retain和release方法操作的是引用记数，当引用记数为零时，便自动释放内存。并且可以用NSAutoreleasePool对象，对加入自动释放池（autorelease调用）的变量进行管理，当drain时回收内存。
（1） retain，该方法的作用是将内存数据的所有权附给另一指针变量，引用数加1，即

retainCount+= 1;
（2） release，该方法是释放指针变量对内存数据的所有权，引用数减1，即retainCount-= 1;
（3） autorelease，该方法是将该对象内存的管理放到autoreleasepool中。

在ARC中与内存管理有关的标识符，可以分为变量标识符和属性标识符，对于变量默认为__strong，而对于属性默认为unsafe_unretained。也存在autoreleasepool。

其中assign/retain/copy与MRC下property的标识符意义相同，strong类似与retain,assign类似于unsafe_unretained，strong/weak/unsafe_unretained与ARC下变量标识符意义相同，只是一个用于属性的标识，一个用于变量的标识(带两个下划短线__)。所列出的其他的标识符与MRC下意义相同。

5. 线程和进程

进程，是并发执行的程序在执行过程中分配和管理资源的基本单位，是一个动态概念，竞争计算机系统资源的基本单位。每一个进程都有一个自己的地址空间，即进程空间或（虚空间）。进程空间的大小 只与处理机的位数有关，一个 16 位长处理机的进程空间大小为 216 ，而 32 位处理机的进程空间大小为 232 。进程至少有 5 种基本状态，它们是：初始态，执行态，等待状态，就绪状态，终止状态。

线程，在网络或多用户环境下，一个服务器通常需要接收大量且不确定数量用户的并发请求，为每一个请求都创建一个进程显然是行不通的，——无论是从系统资源开销方面或是响应用户请求的效率方面来看。因此，操作系统中线程的概念便被引进了。线程，是进程的一部分，一个没有线程的进程可以被看作是单线程的。线程有时又被称为轻权进程或轻量级进程，也是 CPU 调度的一个基本单位。

进程的执行过程是线状的，尽管中间会发生中断或暂停，但该进程所拥有的资源只为该线状执行过程服务。一旦发生进程上下文切换，这些资源都是要被保护起来的。这是进程宏观上的执行过程。而进程又可有单线程进程与多线程进程两种。我们知道，进程有 一个进程控制块 PCB ，相关程序段 和 该程序段对其进行操作的数据结构集 这三部分，单线程进程的执行过程在宏观上是线性的，微观上也只有单一的执行过程；而多线程进程在宏观上的执行过程同样为线性的，但微观上却可以有多个执行操作（线程），如不同代码片段以及相关的数据结构集。**线程的改变只代表了 CPU 执行过程的改变，而没有发生进程所拥有的资源变化**。除了 CPU 之外，**计算机内的软硬件资源的分配与线程无关**，线程只能共享它所属进程的资源。与进程控制表和 PCB 相似，每个线程也有自己的线程控制表 TCB ，而这个 TCB 中所保存的线程状态信息则要比 PCB 表少得多，这些信息主要是相关指针用堆栈（系统栈和用户栈），寄存器中的状态数据。**进程拥有一个完整的虚拟地址空间，不依赖于线程而独立存在；反之，线程是进程的一部分，没有自己的地址空间，与进程内的其他线程一起共享分配给该进程的所有资源。**

线程可以有效地提高系统的执行效率，但并不是在所有计算机系统中都是适用的，如某些很少做进程调度和切换的实时系统。使用线程的好处是有多个任务需要处理机处理时，减

少处理机的切换时间；而且，线程的创建和结束所需要的系统开销也比进程的创建和结束要小得多。最适用使用线程的系统是多处理机系统和网络系统或分布式系统。

6. 平常常用的多线程处理方式及优缺点

iOS有四种多线程编程的技术，分别是：NSThread，Cocoa NSOperation，GCD（全称：Grand Central Dispatch）,pthread。

四种方式的优缺点介绍:

1）NSThread**优点**：**NSThread 比其他两个轻量级。缺点**：需要自己管理线程的生命周期，线程同步。线程同步对数据的加锁会有一定的系统开销。

2）Cocoa NSOperation**优点:**不需要关心线程管理， 数据同步的事情，可以把精力放在自己需要执行的操作上。Cocoa operation相关的类是NSOperation, NSOperationQueue.NSOperation是个抽象类,使用它必须用它的子类，可以实现它或者使用它定义好的两个子类: NSInvocationOperation和NSBlockOperation.创建NSOperation子类的对象，把对象添加到NSOperationQueue队列里执行。

3)**GCD**(全优点)Grand Central dispatch(GCD)是Apple开发的一个多核编程的解决方案。在iOS4.0开始之后才能使用。GCD是一个替代NSThread, NSOperationQueue,NSInvocationOperation等技术的很高效强大的技术。

4) pthread是一套通用的多线程API，适用于Linux\Windows\Unix,跨平台，可移植，使用C语言，生命周期需要程序员管理，IOS开发中使用很少。

GCD线程死锁

GCD 确实好用 ，很强大，相比NSOpretion 无法提供 取消任务的功能。如此强大的工具用不好可能会出现线程死锁。 如下代码:

```
- (void)viewDidLoad{
[super viewDidLoad];
NSLog(@"=================4");
dispatch_sync(dispatch_get_main_queue(),
              ^{ NSLog(@"=================5"); });
NSLog(@"=================6");
}
```

GCD Queue 分为三种:
1，The main queue： 主队列，主线程就是在个队列中。
2，Global queues： 全局并发队列。
3，用户队列:是用函数 dispatch_queue_create创建的自定义队列

dispatch_sync 和 dispatch_async 区别:

dispatch_async(queue,block) async 异步队列，dispatch_async 函数会立即返回, block 会在后台异步执行。

dispatch_sync(queue,block) sync 同步队列，dispatch_sync 函数不会立即返回，及阻塞当前线程,等待 block同步执行完成。

分析上面代码：
viewDidLoad 在主线程中， 及在dispatch_get_main_queue() 中，执行到sync 时 向dispatch_get_main_queue()插入 同步 threed。sync 会等到 后面block 执行完成才返回， sync 又再 dispatch_get_main_queue() 队列中，它是串行队列，sync 是后加入的，前一个是主线程，所以 sync 想执行 block 必须等待主线程执行完成，主线程等待 sync 返回，去执行后续内容。照成死锁，sync 等待mainThread 执行完成，mianThread 等待sync 函数返回。下面例子：

```
- (void)viewDidLoad{
[super viewDidLoad];
dispatch_async(dispatch_get_global_queue(0, 0), ^{
                NSLog(@"=================1");
               dispatch_sync(dispatch_get_main_queue(), ^{
               NSLog(@"=================2"); });
NSLog(@"=================3"); });
}
```

程序会完成执行，为什么不会出现死锁。
首先： async 在主线程中 创建了一个异步线程 加入 全局并发队列，async 不会等待 block 执行完成，立即返回，
1，async 立即返回， viewDidLoad 执行完毕，及主线程执行完毕。
2，同时，全局并发队列立即执行异步 block ， 打印 1， 当执行到 sync 它会等待 block 执行完成才返回， 及等待dispatch_get_main_queue() 队列中的 mianThread 执行完成，然后才开始调用block 。因为1 和 2 几乎同时执行，因为2 在全局并发队列上， 2 中执行到sync 时 1 可能已经执行完成或 等了一会，mainThread 很快退出， 2 等已执行后继续内容。如果阻塞了主线程，2 中的sync 就无法执行啦，mainThread 永远不会退出，sync 就永远等待着。

7. 大量数据表的优化方案

1.对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

```
select id from t where num is null
```

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

```
select id from t where num=0
```

3.应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

4.应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：

```
 select id from t where num=10 or Name='admin'
```

可以这样查询：

```
 select id from t where num=10 union all select id from t where Name='admin'
```

5.in 和 not in 也要慎用，否则会导致全表扫描，如：

```
select id from t where num in (1,2,3)
```

对于连续的数值，能用 between 就不要用 in 了：

```
 select id from t where num between 1 and 3
```

很多时候用 exists 代替 in 是一个好的选择：

```
 select num from a where num in (select num from b)
```

用下面的语句替换：

```
 select num from a where exists (select 1 from b where num=a.num)
```

6.下面的查询也将导致全表扫描：

```
select id from t where name like ‘%abc%’
```

若要提高效率，可以考虑全文检索。

7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

```
select id from t where num=@num
```

可以改为强制查询使用索引：

```
select id from t with (index(索引名)) where num=@num
```

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num/2=100
```

应改为:

```
select id from t where num=100*2
```

9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
 select id from t where substring(name,1,3)=’abc’ -–name以abc开头的id
 select id from t where datediff(day,createdate,’2015-11-30’)=0 -–‘2015-11-30’ --生成的ic
```

应改为:

```
select id from t where name like'abc%'
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'
```

10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能

的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构:

```
select col1,col2 into #t from t where1=0
```

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样:

```
create table #t(…)
```

13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。

14.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

15.select count(*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

16.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

17.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

18.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连 接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

19.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

20.任何地方都不要使用

```
  select * from t
```

用具体的字段列表代替“*”，不要返回用不到的任何字段。

21.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只

有主键索引）。

22.避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

23.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

24.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。

29.尽量避免大事务操作，提高系统并发能力。

30.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

实际案例分析：拆分大的 DELETE 或INSERT 语句，批量提交SQL语句

如果你需要在一个在线的网站上去执行一个大的 DELETE 或 INSERT 查询，你需要非常小心，要避免你的操作让你的整个网站停止相应。因为这两个操作是会锁表的，表一锁住了，别的操作都进不来了。

Apache 会有很多的子进程或线程。所以，其工作起来相当有效率，而我们的服务器也不希望有太多的子进程，线程和数据库链接，这是极大的占服务器资源的事情，尤其是内存。

如果你把你的表锁上一段时间，比如30秒钟，那么对于一个有很高访问量的站点来说，这30秒所积累的访问进程/线程，数据库链接，打开的文件数，可能不仅仅会让你的WEB服务崩溃，还可能会让你的整台服务器马上挂了。

所以，如果你有一个大的处理，你一定把其拆分，使用 LIMIT oracle(rownum),sqlserver(top)条件是一个好的方法。下面是一个mysql示例：

```
while(1){//每次只做1000条
mysql_query("delete from logs where log_date <= '2015-11-01' limit 1000");
if(mysql_affected_rows() == 0){//删除完成，退出！break;
}//每次暂停一段时间，释放表让其他进程/线程访问。
usleep(50000)
}
```

8. 常用到的动画库

Facebook 开源动画库 Pop 的 GitHub 主页：facebook/pop · GitHub，介绍：Playing with Pop (i)

Canvas 项目主页：Canvas - Simplify iOS Development，介绍：Animate in Xcode Without Code

拿 Canvas 来和 Pop 比其实不大合适，虽然两者都自称「动画库」，但是「库」这个词的含义有所区别。本质上 Canvas 是一个「动画合集」而 Pop 是一个「动画引擎」。

先说 Canvas。Canvas 的目的是「Animate in Xcode Without Code」。开发者可以通过在 Storyboard 中指定 User Defined Runtime Attributes 来实现一些 Canvas 中预设的动画，也就是他网站上能看到的那些。但是除了更改动画的 delay 和 duration 基本上不能调整其他的参数。

Pop 就不一样了。如果说 Canvas 是对 Core Animation 的封装，Pop 则是对 Core Animation（以及 UIDynamics）的再实现。

Pop 语法上和 Core Animation 相似，效果上则不像 Canvas 那么生硬（时间四等分，振幅硬编码）。这使得对 Core Animation 有了解的程序员可以很轻松地把原来的「静态动画」转换成「动态动画」。

同时 Pop 又往前多走了一步。既然动画的本质是根据时间函数来做插值，那么理论上任何一个对象的任何一个值都可以用来做插值，而不仅仅是 Core Animation 里定死的那一堆大小、位移、旋转、缩放等 animatable properties。

9. Restful架构

REST是一种架构风格，其核心是面向资源，REST专门针对网络应用设计和开发方式，以降低开发的复杂性，提高系统的可伸缩性。REST提出设计概念和准则为：

1.网络上的所有事物都可以被抽象为资源(resource)
2.每一个资源都有唯一的资源标识(resource identifier)，对资源的操作不会改变这些标识
3.所有的操作都是无状态的

REST简化开发，其架构遵循CRUD原则，该原则告诉我们对于资源(包括网络资源)只需要四种行为：创建，获取，更新和删除就可以完成相关的操作和处理。您可以通过统一资源标识符（Universal Resource Identifier，URI）来识别和定位资源，并且针对这些资源而执行的操作是通过 HTTP 规范定义的。其核心操作只有GET,PUT,POST,DELETE。

由于REST强制所有的操作都必须是stateless的，这就没有上下文的约束，如果做分布式，集群都不需要考虑上下文和会话保持的问题。极大的提高系统的可伸缩性。

RESTful架构：

（1）每一个URI代表一种资源；

（2）客户端和服务器之间，传递这种资源的某种表现层；

（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

10. 请分析下SDWebImage的原理

这个类库提供一个UIImageView类别以支持加载来自网络的远程图片。具有缓存管理、异步下载、同一个URL下载次数控制和优化等特征。

SDWebImage 加载图片的流程

1.入口 setImageWithURL:placeholderImage:options: 会先把 placeholderImage 显示，然后 SDWebImageManager 根据 URL 开始处理图片。

2.进入 SDWebImageManager-downloadWithURL:delegate:options:userInfo:，交给 SDImageCache 从缓存查找图片是否已经下载 queryDiskCacheForKey:delegate:userInfo:.

3.先从内存图片缓存查找是否有图片，如果内存中已经有图片缓存，SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo: 到 SDWebImageManager。

4.SDWebImageManagerDelegate 回调 webImageManager:didFinishWithImage: 到 UIImageView+WebCache 等前端展示图片。

5.如果内存缓存中没有，生成 NSInvocationOperation 添加到队列开始从硬盘查找图片是否已经缓存。

6.根据 URLKey 在硬盘缓存目录下尝试读取图片文件。这一步是在 NSOperation 进行的操作，所以回主线程进行结果回调 notifyDelegate:。

7.如果上一操作从硬盘读取到了图片，将图片添加到内存缓存中（如果空闲内存过小，会先清空内存缓存）。SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo:。进而回调展示图片。

8.如果从硬盘缓存目录读取不到图片，说明所有缓存都不存在该图片，需要下载图片，回调 imageCache:didNotFindImageForKey:userInfo:。

9.共享或重新生成一个下载器 SDWebImageDownloader 开始下载图片。

10.图片下载由 NSURLConnection 来做，实现相关 delegate 来判断图片下载中、下载完成和下载失败。

11.connection:didReceiveData: 中利用 ImageIO 做了按图片下载进度加载效果。

12.connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。

13.图片解码处理在一个 NSOperationQueue 完成，不会拖慢主线程 UI。如果有需要对下载的图片进行二次处理，最好也在这里完成，效率会好很多。

14.在主线程 notifyDelegateOnMainThreadWithInfo: 宣告解码完成，imageDecoder:didFinishDecodingImage:userInfo: 回调给 SDWebImageDownloader。

15.imageDownloader:didFinishWithImage: 回调给 SDWebImageManager 告知图片下载完成。

16.通知所有的 downloadDelegates 下载完成，回调给需要的地方展示图片。

17.将图片保存到 SDImageCache 中，内存缓存和硬盘缓存同时保存。写文件到硬盘也在以单独 NSInvocationOperation 完成，避免拖慢主线程。

18.SDImageCache 在初始化的时候会注册一些消息通知，在内存警告或退到后台的时候清理内存图片缓存，应用结束的时候清理过期图片。

19.SDWI 也提供了 UIButton+WebCache 和 MKAnnotationView+WebCache，方便使用。

20.SDWebImagePrefetcher 可以预先下载图片，方便后续使用。

SDWebImage库的作用

通过对UIImageView的类别扩展来实现异步加载替换图片的工作。

主要用到的对象：
1、UIImageView (WebCache)类别，入口封装，实现读取图片完成后的回调
2、SDWebImageManager，对图片进行管理的中转站，记录那些图片正在读取。
向下层读取Cache（调用SDImageCache），或者向网络读取对象（调用SDWebImageDownloader）。
实现SDImageCache和SDWebImageDownloader的回调。

3、SDImageCache，根据URL的MD5摘要对图片进行存储和读取（实现存在内存中或者存在硬盘上两种实现）
实现图片和内存清理工作。
4、SDWebImageDownloader，根据URL向网络读取数据（实现部分读取和全部读取后再通知回调两种方式）

其他类：SDWebImageDecoder，异步对图像进行了一次解压……

1、SDImageCache是怎么做数据管理的?

SDImageCache分两个部分，一个是内存层面的，一个是硬盘层面的。内存层面的相当是个缓存器，以Key-Value的形式存储图片。当内存不够的时候会清除所有缓存图片。用搜索文件系统的方式做管理，文件替换方式是以时间为单位，剔除时间大于一周的图片文件。当SDWebImageManager向SDImageCache要资源时，先搜索内存层面的数据，如果有直接返回，没有的话去访问磁盘，将图片从磁盘读取出来，然后做Decoder，将图片对象放到内存层面做备份，再返回调用层。

2、为啥必须做Decoder?
由于UIImage的imageWithData函数是每次画图的时候才将Data解压成ARGB的图像，所以在每次画图的时候，会有一个解压操作，这样效率很低，但是只有瞬时的内存需求。为了提高效率通过SDWebImageDecoder将包装在Data下的资源解压，然后画在另外一张图片上，这样这张新图片就不再需要重复解压了。
这种做法是典型的空间换时间的做法。