-----------GCD—Dispatch多核CPU处理-------------------------------

总结（7+1个解决方案） RunloopGroup-1号文件大量实验

0. 自定义并发队列

sharedPhotoManager->\_concurrentPhotoQueue =

dispatch\_queue\_create("com.selander.GooglyPuff.photoQueue",

DISPATCH\_QUEUE\_CONCURRENT);

0. 串行队列 和 并发队列

dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT,0)

dispatch\_get\_global\_queue(0,0)

0. 队列执行的挂起和启动 + 释放

/\*如果暂停一个队列不要忘记恢复。暂停和恢复的操作和内存管理中的retain和release类似。调用 dispatch\_suspend会增加暂停计数，而dispatch\_resume则会减少。队列只有在暂停计数变成零的 情况下才开始运行。\*/

dispatch\_suspend(myQueue);

dispatch\_resume(myQueue);

dispatch\_release(myQueue);

1. 同步添加操作 & 异步添加操作

dispatch\_sync( dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT,0) , ^{ …… });

dispatch\_async(dispatch\_get\_main\_queue(), ^{printf(" 主线程队列调用测试\n"); });

2. 重复异步添加操作（循环）

dispatch\_apply( 20 , dispatch\_get\_global\_queue(0,0) ,^(size\_t i){

printf("第 %zu 个代码块\n",i);

});

注意：该循环虽是异步添加任务(使用并发队列的话还是并发执行)，但在所有任务完成前会阻塞所在线程

3. 延后异步添加操作

dispatch\_after( dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)(1.0 \* NSEC\_PER\_SEC)), dispatch\_get\_main\_queue(), ^(void){

;

});

dispatch\_after 工作起来就像一个延迟版的 dispatch\_async 。

4. 单例模式：仅执行一次，执行过程中阻塞其他调用到的线程（跨队列）

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

sharedPhotoManager = [[PhotoManager alloc] init];

sharedPhotoManager->\_photosArray = [NSMutableArray array];

});

5. 障碍机制（用于单例中数据的安全getter&setter）：并发队列中，该任务互斥于前后并发任务

//持有一个为自己服务的自定义并发队列

@property (nonatomic, strong) dispatch\_queue\_t concurrentPhotoQueue;

//新属性的实例化

sharedPhotoManager->\_concurrentPhotoQueue =

dispatch\_queue\_create("com.selander.GooglyPuff.photoQueue",

DISPATCH\_QUEUE\_CONCURRENT);

//制造障碍任务（写操作）

dispatch\_barrier\_async(self.concurrentPhotoQueue, ^{

[\_photosArray addObject:photo];

});

//并发读操作（同步等待copy操作完成）

\_\_block NSArray \*array; //\_\_block 关键字允许对象在 Block 内可变（C本体变量貌似不用）

dispatch\_sync(self.concurrentPhotoQueue, ^{

array = [NSArray arrayWithArray:\_photosArray];

});

return array;

6. 调度组

在一个异步出线程中，通过监控计数，wait判断多个并发任务是否都完成来阻塞该线程）

//声明一个调度组：它的作用就像一个用于未完成任务的计数器。

dispatch\_group\_t downloadGroup = dispatch\_group\_create();

//让下面代码进入调度组监控中：Explicitly indicates that a block has entered the group.

//每次进入都会增加计数器

for( 让下面的异步下载执行n次 ){

dispatch\_group\_enter(downloadGroup);

Photo \*photo = [[Photo alloc] initwithURL:url

withCompletionBlock:^(UIImage \*image, NSError \*\_error) {

if (\_error) { error = \_error; }

//在并发任务结束的某处释放一次计数

dispatch\_group\_leave(downloadGroup);

}];

}

//阻塞当前线程，等待计数器归零才继续线程

dispatch\_group\_wait(downloadGroup, DISPATCH\_TIME\_FOREVER);

因为你在使用的是同步的 dispatch\_group\_wait ，它会阻塞当前线程，所以你要用 dispatch\_async 将 整个方法放入后台队列以避免阻塞主线程。

dispatch\_group\_wait 会一直等待，直到任务全部完成或者超时。如果在所有任务完成前超时了，该函数 会返回一个非零值。你可以对此返回值做条件判断以确定是否超出等待周期；然而，你在这里用 DISPATCH\_TIME\_FOREVER 让它永远等待。它的意思，勿庸置疑就是，永－远－等－待！

也可以不用异步出线程，使用另一个函数将wait单独异步出来也行

//此处将调度组的wait变为通知

dispatch\_group\_notify(downloadGroup, dispatch\_get\_main\_queue(), ^{

if (completionBlock) {

completionBlock(error);

}

});

7. 阻塞（使用信号量）：异步线程如何阻塞主线程的正确姿势 —— GCD-Demo-1调试工程代码中

原先使用空的while循环判断指定参数是否为真，否则持续自旋锁（缺点：消耗cpu巨大）

1. dispatch\_semaphore\_t semaphore = dispatch\_semaphore\_create(0);

2. dispatch\_semaphore\_signal(semaphore);

3. dispatch\_time\_t timeoutTime = dispatch\_time(DISPATCH\_TIME\_NOW, k

DefaultTimeoutLengthInNanoSeconds);

3.if (dispatch\_semaphore\_wait(semaphore, timeoutTime)) {

XCTFail(@"%@ timed out", URLString);

}

1. 创建一个信号量。参数指定信号量的起始值。这个数字是你可以访问的信号量，不需要有人先去增加它的数 量。（注意到增加信号量也被叫做发射信号量）。译者注：这里初始化为0，也就是说，有人想使用信号量 必然会被阻塞，直到有人增加信号量。

2. 在 Completion Block 里你告诉信号量你不再需要资源了。这就会增加信号量的计数并告知其他想使用此 资源的线程。

3. 这会在超时之前等待信号量。这个调用阻塞了当前线程直到信号量被发射。这个函数的一个非零返回值表示 到达超时了。在这个例子里，测试将会失败因为它以为网络请求不会超过 10 秒钟就会返回

8. 获取当前类所在线程断面（调试感知）：暗黑工具 Dispatch Source 调试代码，见GCD-Demo-1大纲第6条

#if DEBUG

dispatch\_queue\_t queue = dispatch\_get\_main\_queue();

static dispatch\_source\_t source = nil;

\_\_typeof(self) \_\_weak weakSelf = self;

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

source = dispatch\_source\_create(DISPATCH\_SOURCE\_TYPE\_SIGNAL, SIGSTOP, 0, queue);

if (source){

dispatch\_source\_set\_event\_handler(source, ^{

NSLog(@"我已经捕获当前类weakSelf对象: %@", weakSelf);

});

dispatch\_resume(source);

}

});

#endif

粘贴使用即可：handler闭包加入断点调试即可进入当前类存在的线程

应用原则

同步添加阻塞添加操作所在线程，等待任务返回。

异步添加不阻塞线程，添加完后立刻继续其他操作。

串行队列串行执行任务，并发队列并行执行所有任务。

每个任务的实际执行线程未知，随时调配，一般是当前的空闲线程

简介

是Apple开发的一个多核编程的较新的解决方法。它主要用于优化应用程序以支持多核处理器以及其他对称多 处理系统。它是一个在线程池模式的基础上执行的并行任务。

GCD是一个替代诸如NSThread等技术的很高效和强大的技术。GCD完全可以处理诸如数据锁定和资源泄漏 等复杂的异步编程问题。GCD的工作原理是让一个程序，根据可用的处理资源，安排他们在任何可用的处 理器核心上平行排队执行特定的任务。这个任务可以是一个功能或者一个程序段。

GCD中的 一个任务 可被用于创造 一个 被放置于队列的 工作项目或者事件源。如果一个任务 被分配 到一 个事件源，那么一个由功能或者程序块组成的 工作单元 会被放置于一个适当的队列中。苹果公司认为 GCD相比于普通的一个接一个的执行任务的方式更为有效率。

（一个任务 、工作项目/事件源 、工/事的队列 、工作单元 、工作单元队列）

具有以下优点：

1.GCD 能通过推迟昂贵计算任务并在后台运行它们来改善你的应用的响应性能。

2.GCD 提供一个易于使用的并发模型而不仅仅只是锁和线程，以帮助我们避开并发陷阱。

3.GCD 具有在常见模式（例如单例）上用更高性能的原语优化你的代码的潜在能力。

功能：

一、调度队列

所有的调度队列都是先进先出队列，因此，队列中的任务的开始的顺序和添加到队列中的顺序相同。GCD 自动的为我们提供了一些调度队列，我们也可以创建新的用于具体的目的。

下面列出几种可用的调度队列类型以及如何使用。

二、调度资源

是个监视某些类型事件的对象。当这些事件发生时，它自动将一个block放入一个调度队列的执行例程中。

三、调度组

允许将多任务分组来方便后来加入执行。任务能作为一个组中的一个成员被加到队列中，客户端能用这个 组对象来等待直到这个组中的所有任务完成。

四、调度信号量

允许客户端并行发布一定数量的任务。

OC学习文档——MyProject——24号文件

X第一个实例（下面有更好的例子）

一个处理文档的应用程序有一个名为analyzeDocument函数，会做一些统计文档的单词和段落的事情。通常 情况下，这将是一个快速的过程，并在用户根本没注意到按下一个键和结果显示出来之间的延迟就已经在 主线 程中执行好了。

-(IBAction)analyzeDocument:(NSButton\*)sender{

NSDictionary\*stats=[myDoc analyze];

[myModel setDict:stats];

[myStatsView setNeedsDisplay:YES];

[stats release];

}

如果一个文档太大并且需要很长的时间去执行，那么主线程将会暂停等待这个函数完成。如果花了很长的时间， 那么用户就会注意到这个延迟，应用程序甚至会没有响应。

-(IBAction)analyzeDocument:(NSButton\*)sender{

dispatch\_async( dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT,0) ,

^{

NSDictionary\*stats=[myDoc analyze];

dispatch\_async( dispatch\_get\_main\_queue(),^{

[myModel setDict:stats];

[myStatsView setNeedsDisplay:YES];

[stats release];

}

);

});

};

在这里，[myDoc analyze]的调用先被放置在了一个块中，然后进入一个全局并发队列里。在完成[myDoc analyze]的运行之后，一个新的块放置在主队列里（应用程序主线程在上面运行），更新了GUI（这 是必要的，因为GUI只能由主线程更新）

术语解释

Serial vs. Concurrent 串行 vs. 并发

任务串行执行就是每次只有一个任务被执行，任务并发执行就是在同一时间可以有多个任务被执行。

Synchronous vs. Asynchronous 同步 vs. 异步

描述当一个任务相对于一个函数完成，此任务是该函数要求 GCD 执行的。一个同步函数只在完成了它预定 的任务后才返回。

一个异步函数，刚好相反，会立即返回，预定的任务会完成但不会等它完成。因此，一个异步函数不会阻塞当 前线程去执行下一个函数。

Critical Section 临界区

就是一段代码不能被并发执行，也就是，两个线程不能同时执行这段代码。这很常见，因为代码去操作一个共 享资源，例如一个变量若能被并发进程访问，那么它很可能会变质

Race Condition 竞态条件

这种状况是指基于特定序列或时机的事件的软件系统以不受控制的方式运行的行为，例如程序的并发任务执行 的确切顺序。竞态条件可导致无法预测的行为，而不能通过代码检查立即发现。

Deadlock 死锁 —— OC 24号文件的大量实验

两个（有时更多）东西——在大多数情况下是线程——所谓的死锁是指它们都卡住了，并等待对方完成或执 行其它操作。第一个不能完成是因为它在等待第二个的完成。但第二个也不能完成，因为它在等待第一个 的完成。

//添加任务这个操作 属于其发生的线程，及所属线程

//异步添加的任务，任务添加完会立刻 返回所属线程

//同步添加的任务，任务的执行完成前会阻塞 所属线程

//一，二两个任务按顺序被执行，第三个任务在主线程被添加完毕，但第3个任务是"向同队列添加了第4个同 步任务并执行完毕”，它在次线程执行添加了第4个打印任务

//等到第4任务完毕才返回次线程，可惜第4个任务需要第3任务执行完后才能被分配线程资源，而第3任务 还在次线程等待第4任务完毕，所以……懵逼了

//更换第3个任务任务内的队列：第3任务在次线程完成添加(添加操作若是主线程的，那么也会阻塞)，并在 其他队列被其他线程执行并阻塞次线程，完成后返回次线程

//更换第3个任务任务内的同步为异步，保持同队列：在次线程完成同队列的异步添加后，立刻返回次线程并 让次线程执行结束，然后第4个任务被执行

1. 自定义串行队列：在这个状况下要非常小心！如果你正运行在一个队列并调用 dispatch\_sync 放在同一个 队列，那你就百分百地创建了一个死锁。（当前任务正在等待下一个任务的完成）

2. 主队列（串行）：同上面的理由一样，必须非常小心！这个状况同样有潜在的导致死锁的情况。

3. 并发队列：这才是做同步工作的好选择，不论是通过调度障碍，或者需要等待一个任务完成才能执行进一步 处理的情况。

Thread Safe 线程安全（与临界区对应，不是不让并发调用，而是你随便调用也不会产生变质）

线程安全的代码能在多线程或并发任务中被安全的调用，而不会导致任何问题（数据损坏，崩溃，等）。线程不 安全的代码在某个时刻只能在一个上下文中运行。一个线程安全代码的例子是 NSDictionary 。你可以在 同一时间在多个线程中使用它而不会有问题。另一方面，NSMutableDictionary 就不是线程安全的，应 该保证一次只能有一个线程访问它。

向数据加锁，而不是代码块加锁，使得并发进程互相之间的等待时间更短

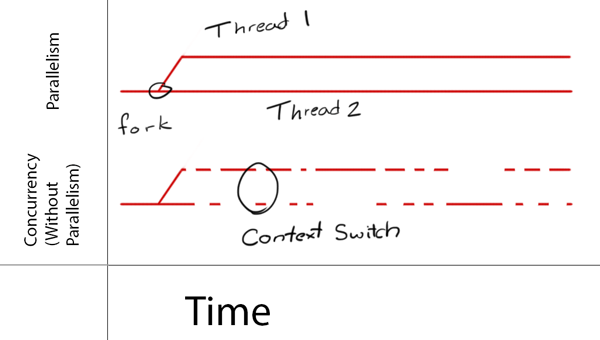
Context Switch 上下文切换

一个上下文切换指当你在单个进程里切换执行不同的线程时存储与恢复执行状态的过程。这个过程在编写多任 务应用时很普遍，但会带来一些额外的开销。

Concurrency vs Parallelism 并发与并行

多核设备通过并行来同时执行多个线程；

然而，为了使单核设备也能实现这一点，它们必须先运行一个线程，执行一个上下文切换，然后运行另一个线 程或进程。这通常发生地足够快以致给我们并发执行地错觉



虽然你可以编写代码在 GCD 下并发执行，但 GCD 会决定有多少并行的需求。

并行要求并发，但并发并不能保证并行

更深入的观点是并发实际上是关于构造。当你在脑海中用 GCD 编写代码，你组织你的代码来暴露能同时运行 的多个工作片段，以及不能同时运行的那些

Queues 队列

GCD 提供有 dispatch queues 来处理代码块，这些队列管理你提供给 GCD 的任务并用 FIFO 顺序执行这 些任务。这就保证了第一个被添加到队列里的任务会是队列中第一个开始的任务，而第二个被添加的任务 将第二个开始，如此直到队列的终点。

所有的调度队列（dispatch queues）自身都是线程安全的，你能从多个线程并行的访问它们。 GCD 的优点 是显而易见的，即当你了解了调度队列如何为你自己代码的不同部分提供线程安全。关于这一点的关键是 选择正确类型的调度队列和正确的调度函数来提交你的工作。

(目前调度队列仅看到add，即使没有安全措施，也是安全的啊)

Serial Queues 串行队列

这些任务的执行时机受到 GCD 的控制；唯一能确保的事情是 GCD 一次只执行一个任务，并且按照我们 添加到队列的顺序来执行。

由于在串行队列中不会有两个任务并发运行，因此不会出现同时访问临界区的风险（比如通过指针提交同 一个代码块时）；相对于这些任务来说，这就从竞态条件下保护了临界区。所以如果访问临界区的唯 一方式是通过提交到串行调度队列的任务，那么你就不需要担心临界区的安全问题了。

Concurrent Queues 并发队列

在并发队列中的任务能得到的保证是它们会按照被添加的顺序开始执行，但这就是全部的保证了。任务可 能以任意顺序完成，你不会知道何时开始运行下一个任务，或者任意时刻有多少 Block 在运行。再 说一遍，这完全取决于 GCD 。

让我想起了调度算法：4个机器同样速度，n个任务排成队列（动态规划和Johnson法则变为排序问题）

Queue Types 队列类型

主队列（main queue） 的特殊队列。和其它串行队列一样，这个队列中的任务一次只能执行一个。然而，它 能保证所有的任务都在主线程执行，而主线程是唯一可用于更新 UI 的线程。这个队列就是用于发生消息 给 UIView 或发送通知的。

全局调度队列（Global Dispatch Queues） 。目前的四个全局队列有着不同的优先级：background、low、 default 以及 high（函数有对应选项）。要知道，Apple 的 API 也会使用这些队列，所以你添加的任何 任务都不会是这些队列中唯一的任务

上面提到异步添加和同步添加，这仅仅是添加后是否立刻回到所属线程，而添加进去的任务并发还是串行执行 取决于你使用的队列

第二个实例 - 循环

这段代码调用的do\_work函数统计次数，将第i次的结果赋值给数组的第i个元素，然后在循环结束之后调用 summarize函数

**for**(i=0;i<count;i++){

     results[i]=do\_work(data,i);

}

total=summarize(results,count);

这些操作时顺序执行的，实际上并不需要这样。假设do\_work函数不需要其他函数调用的结果，那么这些调用就可 以同时运行了

dispatch\_apply( count , dispatch\_get\_global\_queue(0,0) ,

^(size\_t i){

     results[i]=do\_work(data,i);

}

);

total=summarize(results,count);

这里，dispatch\_apply运行传递来的块，在全局队列中放入每一次调用，并且给每次块调用一个从0到n-1 的不同的数字。这样就允许操作系统通过当前的硬件和系统负载选择最佳的线程数来分配合适的工作。 dispatch\_apply知道所有的在给定队列的块运行完成才返回，这样就可以保证在调用summarize之前完成原 来的循环里的所有事情。

注意：该循环虽是异步添加任务(使用并发队列的话还是并发执行)，但在所有任务完成前会阻塞所在线程

项目GCD-Demo-1解析

1. 普通异步添加任务：PhotoDetailViewController 的 viewDidLoad 方法中

有延迟的图片处理函数放在了

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_HIGH, 0), ^{ 中 }）

处理完图片后的图片fade更新放在了主线程中：dispatch\_get\_main\_queue（）

dispatch\_async(dispatch\_get\_main\_queue(), ^{

[self postContentAddedNotification];

});

原本点击展示图片细节的时候，由于需要等待viewDidLoad，所以有一个延迟，现在没有了

2. 延后异步添加任务： PhotoCollectionViewController.m 中 showOrHideNavPrompt（文件末尾）

**double** delayInSeconds = 1.0;

dispatch\_time\_t popTime =

dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)(delayInSeconds \* NSEC\_PER\_SEC));

     dispatch\_after(popTime, dispatch\_get\_main\_queue(), ^(**void**){  …… } )

dispatch\_after 工作起来就像一个延迟版的 dispatch\_async 。你依然不能控制实际的执行时间，且一旦 dispatch\_after 返回也就不能再取消它

3. 单例模式：PhotoManager.m 的 sharedManager

解决方案：

+ (instancetype)sharedManager

{

static PhotoManager \*sharedPhotoManager = nil;

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

sharedPhotoManager = [[PhotoManager alloc] init];

sharedPhotoManager->\_photosArray = [NSMutableArray array];

});

return sharedPhotoManager;

}

试图访问临界区（即传递给 dispatch\_once 的代码）的不同的线程会在临界区已有一个线程的情况下被阻塞， 直到临界区完成为止

原先你认为进入后立刻设置onceToken，使得其他线程关于闭包代码不执行，但若原线程没执行完，其他线程 return的 还是一个nil，所以原线程设置的onceToken会使其他并发调用者跳过闭包，且阻塞线程

问题陈述：

+ (instancetype)sharedManager

{

static PhotoManager \*sharedPhotoManager = nil;

if (!sharedPhotoManager) {

[NSThread sleepForTimeInterval:2];

sharedPhotoManager = [[PhotoManager alloc] init];

NSLog(@"Singleton has memory address at: %@", sharedPhotoManager);

[NSThread sleepForTimeInterval:2];

sharedPhotoManager->\_photosArray = [NSMutableArray array];

}

return sharedPhotoManager;

}

上述方法中，sharedPhotoManager是方法内静态的（方法多次调用指向同一段内存）

若是在进入if代码块后，发生了上下文切换（上述用NSTread闲置来达到强制切换）

另一个代码块在读到sharedPhotoManager后也进入了if代码块

最终会执行两次实例化（若第二次实例化有很多设置，那么第一次实例会覆盖第二次的实例）

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_HIGH, 0), ^{

[PhotoManager sharedManager];

});

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_HIGH, 0), ^{

[PhotoManager sharedManager];

});

上述代码用异步执行来验证两次实例发生

4. 单例类成员的线程安全 —— PhotoManager

解决方案：（首先明白：读和写是互斥的，写和写是互斥的，但读和读是可以并发的）

//持有一个为自己服务的自定义并发队列

@property (nonatomic, strong) dispatch\_queue\_t concurrentPhotoQueue;

//新属性的实例化

sharedPhotoManager->\_concurrentPhotoQueue =

dispatch\_queue\_create("com.selander.GooglyPuff.photoQueue",

DISPATCH\_QUEUE\_CONCURRENT);

//安全的添加方式：等待自定义队列中其他之前任务的执行，自己执行时排斥后面其他任务

- (void)addPhoto:(Photo \*)photo

{

if (photo) {

dispatch\_barrier\_async(self.concurrentPhotoQueue, ^{

[\_photosArray addObject:photo];

dispatch\_async(dispatch\_get\_main\_queue(), ^{

[self postContentAddedNotification];

});

});

}

}

//安全的访问方式：为了防止异步情况下被barrier阻塞而返回空指针，这里使用同步，代码块的执行返回

//\_\_block关键字允许对象在 Block 内可变。没有它，array 在 Block 内部就只是只读的，你的代码甚至不能通过编译。

- (NSArray \*)photos

{

\_\_block NSArray \*array;

dispatch\_sync(self.concurrentPhotoQueue, ^{

array = [NSArray arrayWithArray:\_photosArray];

});

return array;

}

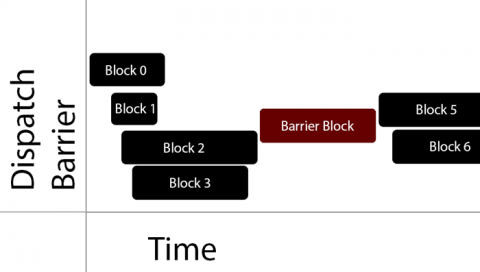
dispatch\_sync() 同步地提交工作并在返回前等待它完成。使用 dispatch\_sync 跟踪你的调度障碍工作， 或者当你需要等待操作完成后才能使用 Block 处理过的数据。如果你使用第二种情况做事，你将不时看 到一个 \_\_block 变量写在 dispatch\_sync 范围之外，以便返回时在 dispatch\_sync 使用处理过的对象

问题陈述：

Apple中array的多线程访问是安全的，但是mutable类型的是不安全的，你可以通过[NSArray arrayWithArray:\_photosArray];来copy当前数组防止getter的恶意修改，但你不能阻止一个线 程调用getter的同时另一个线程调用addPhoto

GCD通过dispatch\_barriers创建读写者锁，提供一个优雅的解决方案，Dispatch barriers 是一组函数，在 并发队列上工作时扮演一个串行式的瓶颈。使用 GCD 的障碍（barrier）API 确保提交的 Block 在那个 特定时间上是指定队列上唯一被执行的条目。这就意味着所有的先于调度障碍提交到队列的条目必能在这 个 Block 执行前完成。

当这个 Block 的时机到达，调度障碍执行这个 Block 并确保在那个时间里队列不会执行任何其它 Block 。 一旦完成，队列就返回到它默认的实现状态。 GCD 提供了同步和异步两种障碍函数。



使用该障碍机制的注意：

1. 自定义串行队列：一个很坏的选择；障碍不会有任何帮助，因为不管怎样，一个串行队列一次都只执行 一个操作。

2. 全局并发队列：要小心；这可能不是最好的主意，因为其它系统可能在使用队列而且你不能垄断它们只 为你自己的目的。

3. 自定义并发队列：这对于原子或临界区代码来说是极佳的选择。任何你在设置或实例化的需要线程安全 的事物都是使用障碍的最佳候选。

4. 为你的单例划定一个自定义并发，针对于该单例的操作，自定义并发还是障碍，要使用讲道理的障碍， 不要障碍了他人

5. Dispatch Groups - 调度组：跨队列监控一组任务的完成

代码位置：photo下载完成闭包回调 + photoManager下载闭包回调 + collection下载传入弹出框闭包

问题陈述：异步的下载任务为3个并发，并发队列未知（在实现内部），你该如何监控并发的多个异步事件，等 他们都完成来触发你的事件？你不知道它们何时完成，而且它们完成的顺序完全是不确定的

问题解决：Dispatch Group 会在整个组的任务都完成时通知你。这些任务可以是同步的，也可以是异步的， 即便在不同的队列也行。而且在整个组的任务都完成时，Dispatch Group 可以用同步的或者异步的方式 通知你。因为要监控的任务在不同队列，那就用一个 dispatch\_group\_t 的实例来记下这些不同的任务。

代码：

PhotoManager.m用下列实现替换 downloadPhotosWithCompletionBlock:

//声明一个调度组：它的作用就像一个用于未完成任务的计数器。

dispatch\_group\_t downloadGroup = dispatch\_group\_create();

//让下面代码进入调度组监控中：Explicitly indicates that a block has entered the group.

//每次进入都会增加计数器

for( 让下面的异步下载执行n次 ){

dispatch\_group\_enter(downloadGroup);

Photo \*photo = [[Photo alloc] initwithURL:url

withCompletionBlock:^(UIImage \*image, NSError \*\_error) {

if (\_error) { error = \_error; }

//在并发任务结束的某处释放一次计数

dispatch\_group\_leave(downloadGroup);

}];

}

//阻塞当前线程，等待计数器归零才继续线程

dispatch\_group\_wait(downloadGroup, DISPATCH\_TIME\_FOREVER);

因为你在使用的是同步的 dispatch\_group\_wait ，它会阻塞当前线程，所以你要用 dispatch\_async 将 整个方法放入后台队列以避免阻塞主线程。

dispatch\_group\_wait 会一直等待，直到任务全部完成或者超时。如果在所有任务完成前超时了，该函数 会返回一个非零值。你可以对此返回值做条件判断以确定是否超出等待周期；然而，你在这里用 DISPATCH\_TIME\_FOREVER 让它永远等待。它的意思，勿庸置疑就是，永－远－等－待！

也可以不用异步出线程，使用另一个函数将wait单独异步出来也行

//此处将调度组的wait变为通知

dispatch\_group\_notify(downloadGroup, dispatch\_get\_main\_queue(), ^{

if (completionBlock) {

completionBlock(error);

}

});

第二种解决方案（快了一丢丢）：

不用异步出线程

使用dispatch\_apply替换了for的操作，使得for中的代码一开始就并发到了其他队列

在apply到并发线程后，其中的下载方法又异步到另一个并发线程

在3个线程中，都完成 异步出下载线程 这个操作后后，执行apply下面的notify异步等待线程

这样使得for中的非异步代码也异步执行了，更快的去执行下一个循环，更快的进入notify等待阶段

6. 使用 Dispatch Source 调试感知

基本上就是一个低级函数的 grab-bag ，能帮助你去响应或监测 Unix 信号、文件描述符、Mach 端口、VFS 节点，以及其它晦涩的东西

函数原型：

dispatch\_source\_t dispatch\_source\_create(

dispatch\_source\_type\_t type,

uintptr\_t handle,

unsigned long mask,

dispatch\_queue\_t queue);

dispatch\_source\_type\_t ，这是最重要的参数，因为它决定了 handle 和 mask 参数将会是什么

下面一个监控当前进程信号（DISPATCH\_SOURCE\_TYPE\_SIGNAL）的 Dispatch Source。 handle 是信号 编号，mask 未使用（传 0 即可）。

这些 Unix 信号组成的列表可在头文件 signal.h 中找到。在其顶部有一堆 #define 语句。你将监控此信 号列 表中的 SIGSTOP 信号。这个信号将会在进程接收到一个无法回避的暂停指令时被发出。在你用 LLDB 调 试器调试应用时你使用的也是这个信号。

//最好是在 DEBUG 模式下编译这些代码，因为这会给“有关方面（Interested Parties）”很多关于你应用的洞察。

#if DEBUG

//获取主线程队列

dispatch\_queue\_t queue = dispatch\_get\_main\_queue();

// 你需要 source 在方法范围之外也可被访问(不像OC对象一样得到ARC管理)，所以你使用了一个 static 变量。

static dispatch\_source\_t source = nil;

//使用 weak Self 以确保不会出现保留环（Retain Cycle，无法释放）

\_\_typeof(self) \_\_weak weakSelf = self;

//初始化 与 活跃源对象 仅仅能执行一次的

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

//初始化 source 变量，提供了 SIGSTOP 信号作为第二个参数，传入主队列，用其处理接收到的事件

source = dispatch\_source\_create(DISPATCH\_SOURCE\_TYPE\_SIGNAL, SIGSTOP, 0, queue);

//如果你提供的参数不合格，那么 Dispatch Source 对象不会被创建

if (source){

//当你收到你所监控的信号时，dispatch\_source\_set\_event\_handler 就会执行

dispatch\_source\_set\_event\_handler(source, ^{

NSLog(@"Hi, I am: %@", weakSelf); });

//默认的，所有源都初始为暂停状态。如果你要开始监控事件，你必须告诉源对象恢复活跃状态

dispatch\_resume(source);

}

});

//结束debuge模式

#endif

在调试器里暂停并立即恢复应用，查看控制台，你会看到这个来自黑暗艺术的函数确实可以工作。

进阶操作：暂停以调试内存

在上述代码的 NSLog打个断点调试

暂停后恢复，捕捉到暂停信号的 源对象 会向 mainQueue 放入要执行的代码块，这时程序在断点停止

在调试器中键入：**po [[weakSelf navigationItem] setTitle:@"hahhaha"]**

然后继续运行，会发现标题改为了hahaha

在断点停止时，调试器打入的代码能够相当于插入到断点之后的代码一样执行!!!!

该操作的意义：

在其他方法中直接通过断点停止，然后插入代码继续执行发现执行失败，你并不知道你断在哪个线程里了

而上述方法通过并发线程执行runloop监听进程信号，发现调试发生就向回调代码块传递需要操作的指针

若将该代码块放置断点调试，那么一定能确定你在该线程能够操作哪些指针

上述代码块打印**weakSelf**时就自动copy了该变量（通过传递弱引用达到闭包弱引用的效果）

-----------数据通信协议--------------------------------------------------------------------------

XML简介 - 语法，模式，框架

声明（开头单行）：<?xml version="1.0" encoding="UTF-8"?>

根元素：根元素只有一个，用于存放主要内容的 <haha> </haha>

子元素：不废话了

属性：<Note id="1"> 这个id就是元素的一个属性，一个元素的属性不能重复

命名空间属性：

语法如如下xmlns:[prefix]=”[url of name]”

其中“xmlns:”是属性名。“prefix”是命名空间的别名，它的值不能为xml。

<sample xmlns:ins=”http://www.lsmx.net.ac”>

<ins:batch-list> <ins:batch>Evening Batch</ins:batch> </ins:batch-list>

</sample>

注意的是batch-list,batch等标记必须在"http://www.lsmx.net.ac"中定义

限定名：<ins:batch-list>

XML解析模式 & 框架

SAX模式：自上而下读取文档，只读不写

基于事件驱动的解析模式—>自上而下读取文档，遇到开始，结束，属性标签时，就会触发相应的事件

DOM模式：树状存储，可读可写

在加载完XML后，该模式将文档元素视为树状结构的节点，一次性读入到内存中

iOS SDK—NSXML：基于OC的XML解析框架，不支持DOM模式

iOS SDK—libxml2：基于C的XML解析器，同时支持SAX和DOM模式

还有一些其他的第三方框架： P347底部

NSXMLParser —20-XMLMyNotes—NotesXMLParser.m文件（委托类，本身启动型）

说在前面：

一般对应XML模板开发一个解析类，需要数据时实例该类，进行解析，解析完成获得传出数据，释放解析类

该类初始化时需要持有XML文件字符流

设置委托类对象来完成对XML解析过程中不同事件的响应

持有主要属性：

解析出的结果为字典，需要强持有该结果：

@property (strong,nonatomic) NSMutableArray<NSDictionary \*> \*notes;

标签的开始与结束中，字符串方法会被调用多次，但该方法并不一定知道自己所处的标签位置，所以需要

在标签开始的时候存储标签的开始位置：

@property (strong,nonatomic) NSString \*currentTagName;

启动解析的方法：

NSXMLParser \*parser = [[NSXMLParser alloc] initWithContentsOfURL:url];

parser.delegate = self;

[parser parse];

NSXMLParser本身还可以使用Data和IO流对象初始化：initWithData 、initWithStream

五个响应事件看下面

解析开始事件：

- (void)parserDidStartDocument:(NSXMLParser \*)parser

由于在一个文档开始时仅执行一次，可以用作dict字典对象的初始化

解析出错的时候触发

- (void)parser:(NSXMLParser \*)parser parseErrorOccurred:(NSError \*)parseError

遇到开头标签时触发

- (void)parser:(NSXMLParser \*)parser didStartElement:(NSString \*)elementName

namespaceURI:(NSString \*)namespaceURI

qualifiedName:(NSString \*)qualifiedName

attributes:(NSDictionary \*)attributeDict

在该方法中，会将当前“开始标签”存入属性中，方便后面”字符串触发“时知道目前在处理什么元素的内容

范例中，拦截了Note开始标签来创建字典对象存入notes中，后面子元素都添加到该字典中

属性可以从attributes参数中获得：NSString \*\_id = [attributeDict objectForKey:@"id"];

遇到字符串的时候触发—字符串是单独的方法，并不知道当前字符串位于那个标签，所以……

- (void)parser:(NSXMLParser \*)parser foundCharacters:(NSString \*)string

该方法中，根据之前存好的标签属性进行判断，就可以在字典中存入正确的键值对了

转义回车符和空格：因为这两个字符也会触发该方法，所以需要滤掉

[str stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceAndNewlineCharacterSet]];

NSCharacterSet类的静态方法指定了字符集为换行符和回车符

遇到结束标签后触发

- (void)parser:(NSXMLParser \*)parser didEndElement:(NSString \*)elementName

namespaceURI:(NSString \*)namespaceURI

qualifiedName:(NSString \*)qName;

方法中，将当前标签设为nil（这样就防止非标签内内容的无端插入）

文档结束后触发—使用通知中心传参，之后原对象被置为nil

- (void)parserDidEndDocument:(NSXMLParser \*)parser

{ [[NSNotificationCenter defaultCenter] postNotificationName:@"reloadViewNotification" object:self.notes userInfo:nil];

self.notes = nil; }

TBXML插件 —20-XMLMyNotes—NotesTBXMLParser.m

说在前面：

该框架采用DOM模式，需要配置使用环境

对于DOM节点的万能操作，无非就是这几个方法了：

获取根节点 、按照name获取子节点 、获取同级的下一个元素 、获取节点内容 、获取节点属性

配置使用环境：

下载TBXML的压缩包并解压后，需要将TMXML-Headers 和 TBXML-Code文件夹添加到工程中

然后需要添加TBXML所依赖的库：Foundation.框架，UIKit.框架，CoreGraphics.框架，libz.dylib

在工程预编译头文件MyNotes-Prefix.pch中添加代码

File-New-File创建PCH File

在该文件中添加代码：

#import <UIKit/UIKit.h>

#import <Foundation/Foundation.h>

#define ARC\_ENABLED 该定义打开TBXML的ARC开关，其本身支持MRC管理的（手动引用计数）

预编译文件可以预先编译，其内容可被添加到工程中所有的代码模块中（包括宏定义）

Build Settings —> Apple LLVM 7.0 Language —> Prefix Header 中配置头文件到工程中

直接使用文件将XML加载到内存中

BXML\* tbxml = [[TBXML alloc] initWithXMLFile:@"Notes.xml" error:nil];

先获取根元素节点，从该节点出发，遍历其子节点

TBXMLElement \* root = tbxml.rootXMLElement;

子节点的获取 与 遍历

获取一个节点（通过父节点，获得第一个满足要求的子节点）：

TBXMLElement \* noteElement = [TBXML childElementNamed:@"Note" parentElement:root];

以该节点为父节点，继续获取子节点

if ( noteElement != nil) {

TBXMLElement \*CDate = [TBXML childElementNamed:@"CDate" parentElement:noteElement];

}

获得下一个节点（同级的下一个满足情况的节点，类似于jQuery的next）

noteElement = [TBXML nextSiblingNamed:@"Note" searchFromElement:noteElement];

获取节点内容

NSString \*CDate = [TBXML textForElement:CDateElement];

获取节点属性

NSString \*\_id = [TBXML valueOfAttributeNamed:@"id" forElement:noteElement error:nil];

NSJSONSerialization — 21-JSONMyNotes—MasterViewController.m

JSON语法：字典使用{}符号，键：值；数组使用[]号，元素都使用逗号分隔，就酱

解析工具：P356有许多解码器，最好的还是iOS5后的NSJSONSerialization

关键代码：

获取Data

NSString\* path = [[NSBundle mainBundle] pathForResource:@"Notes" ofType:@"json"];

NSData \*jsonData = [[NSData alloc] initWithContentsOfFile:path];

解析并获得字典

NSError \*error;

id jsonObj = [NSJSONSerialization JSONObjectWithData:jsonData

options:NSJSONReadingMutableContainers error:&error];

if (!jsonObj || error) { …… }

option参数指定解析JSON的模式，该参数是枚举类型NSJSONReadingOptions定义的：

MutableContainers—指定解析返回的是可变字典

MutableLeaves—指定叶节点是可变的

AllowFragments—指定顶级节点可以不是数组或字典

提取需要的子字典（传入的json一般由：网络状态码 、内容 两部分组成，需要提取内容作为实际操作对象）

NSMutableArray\* self.objects = [jsonObj objectForKey:@"Record"];

JSON编码：

dataWithJSONObject：options：error 或 writeJSONObject：toStream：options：error

NSData \*reverseData = [NSJSONSerialization dataWithJSONObject:jsonObj

options:NSJSONWritingPrettyPrinted

error:&error];

NSLog(@"%@",[[NSString alloc] initWithData:reverseData encoding:NSUTF8StringEncoding]);

-----------SQLite数据库-----------------------------------------

官方API： http://www.sqlite.org/cintro.html

http://www.sqlite.org/lang.html（SQL语句）

建议先看大量的SQLite架构介绍再去看上述进阶

项目范例: MyNotes-SQLite

转化路径string为C的string（由于数据库操作函数都是C实现的，所以不认对象）

#define DBFILE\_NAME @"NotesLis.sqlite3"

NSString \*writableDBPath = [self applicationDocumentsDirectoryFile];

const char\* cpath = [writableDBPath UTF8String];

从char\*转换回string：NSString \*strDate = [[NSString alloc] initWithUTF8String: bufDate];

ASCII是 0-48 A-65 a-97 这种编码格式的

UTF-8是一种变长字节编码方式。对于某一个字符的UTF-8编码，如果只有一个字节则其最高二进制位为0； 如果是多字节，其第一个字节从最高位开始，连续的二进制位值为1的个数决定了其编码的位数，其余各 字节均以10开头。UTF-8最多可用到6个字节

SQL语句汇总：

CREATE TABLE IF NOT EXISTS Note (cdate TEXT PRIMARY KEY, content TEXT)

INSERT OR REPLACE INTO note (cdate, content) VALUES (?,?)

DELETE from note where cdate =?

UPDATE note set content=? where cdate =?

SELECT cdate,content FROM Note where cdate =?

用该路径打开数据库

sqlite3 \*db;

sqlite3\_open(cpath, &db) != SQLITE\_OK

sqlite3\_close(db);

SQLITE\_OK等于0，open成功返回0

db是类成员变量，由Dao持有该句柄

sqlite3\_open函数作用：若有指定的数据库文件，则将该文件信息封装到结构体db中，若没有就创建再封装

struct sqlite3：http://blog.csdn.net/iukey/article/details/7336369

这是一个庞大的结构体，多达108行（上面网址对每一行都有英文注释）

这个 sqlite3 结构体就是被用来描述我们磁盘里的数据库文件的，有了这个描述符我们就可以对这个数据 库进行各种操作了，操作的具体内情我们不必要了解，我们只需要知道怎么去调用API就好

向数据库插入表

const char\* cSql = "CREATE TABLE IF NOT EXISTS Note (cdate TEXT PRIMARY KEY, content TEXT)";

char \*err;

sqlite3\_exec(db, cSql,NULL,NULL,&err) != SQLITE\_OK

sqlite3\_close(db);

第一行：源代码使用NSString，然后UTF8String：转化（无非想说明建表可以使动态的）

第二行：用于容纳建表过程中的报错信息

第三行：通过db句柄来执行sql语句，并返回成功与否 ——> 关闭数据库

数据库操作一般是 open（path，&db）—> 若成功 —> 使用db操作 —> 使用db关闭数据库

orcale中的建表代码为：(可以看出有类似的结构)

create table emp（emp\_id number(10) primary key,dept\_id number(10) references dept(dept\_id)）;

create sequence seq\_emp;

查询数据

说在前面：

该机制以结构体statement为核心，各个阶段函数以从该结构体提取所需信息（sql语句，游标位置，结

果信息，数据库句柄db）来执行并返回。在close后，释放该结构体的内存。这种方法与具体信息无

关的模式值得注意

声明信息状态结构体：sqlite3\_stmt \*statement;

声明sql：const char\* cSql = "SELECT cdate,content FROM Note where cdate =?";

预处理sql：sqlite3\_prepare\_v2(db, cSql, -1, &statement, NULL) == SQLITE\_OK

预处理的目的是将SQL编译成二进制代码，提高SQL执行速度

第一个参数将数据库文件句柄db交给了statement

第三个参数代表全部SQL字符串的长度，在其他方法中也总是-1

绑定参数—之前SQL的？号就是占位符

NSDateFormatter \*dateFormatter = [[NSDateFormatter alloc] init];

[dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];

NSString \*strDate = [dateFormatter stringFromDate:model.date];

const char\* cDate = [strDate UTF8String];

sqlite3\_bind\_text(statement, 1, cDate, -1, NULL);

第二个参数为占位符序号（从1开始）

和提取字段值类似，也有sqlite3\_bind\_int存在，用于占位存放不同类型的值

执行SQL（与上面的sqlite3\_exec不同，这个有返回结果）

while（ (sqlite3\_step(statement) == SQLITE\_ROW ） { …… }

该函数使用到statement中的信息：db句柄（数据库接口）—>游标位置（从何处检索？）—>

SQL（执行）—>结果信息（将检索结果放在此处）—>函数返回SQLITE\_ROW（值为100，表示

has another row ready，检索有结果，有时可以用在循环中迭代读取结果）

当继续使用sqlite3\_step函数时，它会从新游标开始继续检索，若是还有符合的行就放入结果信息中

提取字段数据

char \*bufDate = (char \*) sqlite3\_column\_text(statement, 0);

该函数从statement中的结果信息处，提取该行索引为 0 位置的字段（根据创建表的顺序来定）

字段数据的提取还有其他方法：

sqlite3\_column\_blob 、sqlite3\_column\_double 、int 、int64 、text 、 text16

释放statement，关闭db接口

sqlite3\_finalize(statement);

sqlite3\_close(db);

增、删、改数据

使用path和db，通过sqlite3\_open打开数据库并检查连接是否成功

声明SQL 、statement，并使用sqlite3\_prepare\_v2预处理SQL语句

处理参数为char\*，使用sqlite3\_bind\_text绑定参数

传入statement，使用sqlite3\_step执行SQL

使用sqlite3\_finalize 、sqlite3\_close释放资源

删：sqlite3\_step(statement) != SQLITE\_DONE

由于没有检索结果，所以若成功执行，则返回值为101的SQLITE\_DONE

增：与上没什么不同:

NSString \*sql = @"INSERT OR REPLACE INTO note (cdate, content) VALUES (?,?)";

sqlite3\_bind\_text(statement, 1, cDate, -1, NULL);

sqlite3\_bind\_text(statement, 2, cContent, -1, NULL);

改：与上没什么不同，绑定参数变为两个

SQLite基础知识（待学）

FMDB第三方（待学）

CoreData总结

1. CoreData机制设计了同一的模式，为不同的持久化方式提供统一的存取接口

2. CoreData在内存映射的维持一个NSArray<NSDictionary>，增删改查都是操作这个，saveCtx来保存到文件

3. 接口接收与返回NSManagerObject的子类实例，代表表中的一行

4. 用到的类和方法：

NSManagedObjectModel 、NSPersistentStoreCoordinator 、NSManagedObjectContext

NoteManagedObject

NSFetchRequest 、NSEntityDescription 、NSSortDescriptor 、NSPredicate

NSManagedObjectContext的大量+方法

5. 你可以修改“持久化存储协调器”，来自定义不同持久文件向NSArray<NSDictionary>的读与写。

6. 你可以修改“持久化存储协调器”与”被管理对象的模型“，来自定义不同建模方式，以及对 NSArray<NSDictionary>结构的影响。

CoreData准备工作—AppDelegate.m — CoreDataHelloWorld项目文件夹

说在前面：

以下准备工作涉及：CoreData对象声明，持久化文件（数据库）连接，接口方法封装

关键属性（顺序存在依赖）

“持久化对象存储“：未暴露出来

有三种持久化实现方式：SQLite、二进制文件、内存

NSManagedObjectModel \*managedObjectModel;

“被管理对象的模型“：系统中的实体，与数据库中的表对应，静态模板文件（后续可视化设置）

NSPersistentStoreCoordinator \*persistentStoreCoordinator;

“持久化存储协调器“：在”持久化对象存储“上提供一个控制接口，可以考虑为数据库的连接

@property (readonly, strong, nonatomic) NSManagedObjectContext \*managedObjectContext;

“被管理对象的上下文”：在上下文中可以增删查对象，然后通过“栈“同步到”持久化对象存储“中

关系： n个”持久化对象存储“ —连接— 1个”协调器“ —连接— n个”上下文“

1个”协调器“ —读取— n个”对象模型“

getter方法：基本思路都是若存在则返回，若不存在则创建。创建时的依赖关系：

模型 <- 协调器 <- 上下文

获取”被管理对象的模型“ （读取模板文件供”协调器“使用）

NSURL \*modeURL = [[NSBundle mainBundle] URLForResource:@"HelloWorld" withExtension:@"momd"];

该方法类似于：

[[NSBundle mainBundle] pathForResource:@"hehe" ofType:@"html"]

NSURL \*bundleUrl = [NSURL fileURLWithPath:[[NSBundle mainBundle]bundlePath]];

没有经历检查NSFileManager的存在判定，该方法使得即使文件不存在，也能创建

然后使用该路径指向的文件初始化“被管理对象的模型”

\_managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelURL];

实例化“持久化存储协调器”

使用”对象模型“初始化：

\_persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]

initWithManagedObjectModel:[self managedObjectModel]];

连接持久化对象存储（SQLite3数据库）

NSURL \*storeURL = [[self applicationDocumentsDirectory]

URLByAppendingPathComponent:@"HelloWorld.sqlite"];

NSError \*error = nil;

NSString \*failureReason = @"There was an error creating …… saved data.";

if (![\_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType configuration:nil URL:storeURL options:nil error:&error])

该方法中还可指定其他存储类型

NSSQLiteStoreType：指数据持久化的类型为SQLite数据

NSBinaryStoreType：持久化类型是 二进制文件

NSInMemoryStoreType：持久化类型是 内存形式

该方法中使用NSDictionary封装了一系列错误信息，然后传给Error，供Error打印

返回”被管理对象上下文“ （以供使用）

\_managedObjectContext = [[NSManagedObjectContext alloc] init];

[\_managedObjectContext setPersistentStoreCoordinator: [self persistentStoreCoordinator]];

由于本地“协调器”可能为空（数据库加载错误），所以判断若获取的为nil，则返回的“上下文”也为nil

保存”被管理对象上下文“：当插入、删除、修改数据后，需要通过该方法保存“被管理对象”的上下文

if ([managedObjectContext hasChanges] ){ [managedObjectContext save:&error] }

- (void)applicationWillTerminate:(UIApplication \*)application { [self saveContext]; }

若上下文内容有变，则执行保存操作（作者把保存操作放在了应用退出时来做）

保存操作也有error，该方法有返回值，返回为NO表示保存出错

题外话-如何正确的报错：

NSError \*error = nil;

NSMutableDictionary \*dict = [NSMutableDictionary dictionary];

dict[NSLocalizedDescriptionKey] = @"Failed to initialize the application's saved data";

dict[NSLocalizedFailureReasonErrorKey] = failureReason;

dict[NSUnderlyingErrorKey] = error;

error = [NSError errorWithDomain:@"YOUR\_ERROR\_DOMAIN" code:9999 userInfo:dict];

NSLog(@"Unresolved error %@, %@", error, [error userInfo]);//最后打印该error信息

abort(); //登出

Core Data建表——CoreDataHelloWorld项目文件夹

建模：

默认情况下，采用模板时会生成与项目名同名的”被管理对象的模型“文件

若不采用模板，使用File—New—File打开新文件模板 —> iOS中选择Core Data —>右侧选择Data Model

xcdatamodeld将在编译发布时变成了momd

注意：数据模型文件属于资源文件，默认在项目发布主目录中查找，所以多层架构时将新建的模型文件放在视

图层项目下

创建实体：

选择模型文件 —> 下方Add Entity —> 右侧Attributes添加需要的属性

右下角的“三方块堆叠”图标表示可以在 表视图 与 属性视图之间切换

生成实体代码文件：

选择实体note —> 展开右侧隐藏面板 —> class属性中编辑生成的类名

File—New—File —> iOS—Core—NSManageObject —> 选择数据模型文件（若是你创建了多个）—> 选

择该模型文件中的一个实体

该代码继承NSManageObject：因为需要被Core Data管理

该代码在implement中使用dynamic：说明属性的getter或setter方法会在运行时动态创建

Core Data实际项目——MyNotesCoreData项目文件夹

NoteManagedObject 与 Note.h

两者是需要共同存在的

因为，数据模型接纳与返回的的，只能是继承了NSManagedObject的实体类；而该实体类且不说结构可能

与Note会有区别，它必须被限制在“持久层”这一限定，就需要Note的存在了，Note可以出现

在返回在其他层中，进行数据的传递

虽然，在Dao中将面临Note与ManagedObject之间的转换，但在大型架构中，会显现其巨大威力

说在前面：

基于CoreData准备工作提供的接口，一般是： 获取上下文对象 —> 依赖上下文获取表对象（获取新插入行 对象） —> 声明查询对象 —> 向查询赋予entity/order/predicate等查询信息 —> 执行查询获取行 对象 —> 操作对象（改查） —> 保存上下文

无条件查询—NoteDao.m—findAll方法

数据提取请求对象：用于执行查询（类似于SELECT，后面将在其中设置各个参数）

NSFetchRequest \*fetchRequest = [[NSFetchRequest alloc] init];

设置要查询的表：（相当于select \* from Note）

//从上下文中获取实体类描述

NSManagedObjectContext \*cxt = [self managedObjectContext];

NSEntityDescription \*entity = [NSEntityDescription entityForName:@"Note"

inManagedObjectContext:cxt];

fetchRequest.entity = entity;

设置排序描述：（相当于order by date ascending）

NSSortDescriptor \*sortDescriptor =[[NSSortDescriptor alloc] initWithKey:@"date" ascending:YES];

NSArray \*sortDescriptors = @[sortDescriptor];

fetchRequest.sortDescriptors = sortDescriptors;

执行：需要传入Error指针；返回结果为ManagerObject对象的Array

NSError \*error = nil;

NSArray \*listData = [cxt executeFetchRequest:fetchRequest error:&error];

转换ManageredObject为Note对象

Note \*note = [[Note alloc] initWithDate:mo.date content:mo.content];

有条件查询—NoteDao.m—findById方法

与findAll不同的是：去除了sortPredicate设定，加入了predicate设定（相当于where date = ？？）

fetchRequest.predicate = [NSPredicate predicateWithFormat: @"date = %@",model.date];

修改数据—NoteDao.m—modify方法

之前与findById一样，需要设定predicate来规定修改哪些

NSArray \*listData = [cxt executeFetchRequest:request error:&error];

if ([listData count] > 0) {

NoteManagedObject \*note = [listData lastObject];

note.content = model.content;

error = nil;

//A Boolean value that indicates whether the receiver has uncommitted changes.

//Attempts to commit unsaved changes to registered objects to the receiver’s parent store， // YES if the save succeeds, otherwise NO.

if ([cxt hasChanges] && ![cxt save:&error]){

NSLog(@"修改数据失败:%@, %@", error, [error userInfo]);

return -1;

}

}

删除数据—NoteDao.m—remove方法

其他部分与修改数据的方法一致，但是在查到需要删除的对象后，是使用该对象，通过context来删除

NoteManagedObject \*note = [listData lastObject];

[self.managedObjectContext deleteObject:note];

虽然说传给的是指针，修改会映射，但指针仅仅是单行实例的指针，并不是Array表的指针，删除还需通过上

下文来完成，修改则可以直接操作该对象

插入数据—NoteDao.m—create方法

NoteManagedObject \*note = [NSEntityDescription insertNewObjectForEntityForName:@"Note"

inManagedObjectContext:cxt];

//这里用到键值编码

[note setValue: model.content forKey:@"content"];

[note setValue: model.date forKey:@"date"];

这里使用上下文先插入实例，然后获得这个已经插入的引用，然后修改（或者说是赋值）

不同于上面上下文获得表数据：[NSEntityDescription entityForName:@"Note" inManagedObjectContext:cxt];

和上面完全不同，首先是少了request对象，毕竟不需要查

-----------文件操作-------------------------------

文件结构与获取路径

基本结构：

Douments目录

用于存储大文件，及更新频率大的数据，能够进行iTunes，iCloud备份。获取目录路径的代码：

NSArray \*doc = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);

NSString \*myDoc = [document lastObject];

Library目录

该目录下有Preferences和Caches目录，前者存放应用程序设置数据，后者用于存放缓存数据

Tmp目录

临时目录文件，用户可以访问它，但不能被备份，获取位置的代码：

NSLog(@"%@",NSTemporaryDirectory());

资源目录

貌似是项目文件根目录，在编译完成后，成为resource目录安装并保存在客户端

NSString\*path=[[NSBundle mainBundle]resourcePath]stringByAppendingPathComponent:@""];

iTunes在与iPhone同步时，备份所有的Documents和Library文件。

iPhone在重启时，会丢弃所有的tmp文件。

获取各种目录 — 项目Space里的RunloopLibrary项目

0. 总结：

手机端中除了 资源目录、temp目录 其他的前缀都是：

/var/mobile /Containers/Data/Application/APP号/……

/Devices/A4……A71/data /Containers/Data/Application/APP号/……

temp目录（真机上多了个前缀，虚拟机上的和其他一样前缀）：

/private/var/mobile/Containers/Data/Application/ APP号/tmp/

/Devices/A4……A71/data/Containers/Data/Application/ APP号/tmp/

source目录（data变成了Bundle）：

/var/mobile/Containers/Bundle/Application/另一个APP号/All.app

/Devices/A4……A71/data/Containers/Bundle/Application/另一个APP号/All.app

source目录 与 mainBundle 一样

唯一的区别就是一个是URL对象，一个返回NSString

1. 获取Home目录：

NSString \*homeDirectory = NSHomeDirectory();

打印结果：/var/mobile/Containers/Data/Application/E5D4……0082

打印结果：/Devices/A4……A71/data/Containers/Data/Application/3B2E……9DD1

2. 获取document目录

NSArray \*paths=NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);

NSString \*path = [paths objectAtIndex:0];

打印结果：/var/mobile/Containers/Data/Application/E5D4……0082/Documents

打印结果：/Devices/A4……A71/data/Containers/Data/Application/3B2E……9DD1/Documents

3. 获取Library目录

NSArray \*paths = NSSearchPathForDirectoriesInDomains(NSLibraryDirectory, NSUserDomainMask, YES);

NSString \*path = [paths objectAtIndex:0];

打印结果：/var/mobile/Containers/Data/Application/E5D4……0082/Library

打印结果：/Devices/A4……A71/data/Containers/Data/Application/3B2E……9DD1/ Library

4. 获取Cache目录

NSArray \*paths = NSSearchPathForDirectoriesInDomains(NSCachesDirectory, NSUserDomainMask, YES);

NSString \*path = [paths objectAtIndex:0];

打印结果：/var/mobile/Containers/Data/Application/E5D4……0082/Library/Caches

打印结果：/Devices/A4……A71/data/Containers/Data/Application/3B2E……9DD1/Library/Caches

5. 获取Tmp目录（目录结构和其他不一样，不在mobile/Application里）

NSString \*tmpDir = NSTemporaryDirectory();

打印结果：/private/var/mobile/Containers/Data/Application/E5D4……0082/tmp/

打印结果：/Devices/A4……A71/data/Containers/Data/Application/3B2E……9DD1/tmp/

6. 获取资源路径（代码，图片，.plist文件根路径）

NSString\*path=[[[NSBundle mainBundle]resourcePath]stringByAppendingPathComponent:@""]

打印结果：/var/mobile/Containers/Bundle/Application/5A01……896E/All.app

打印结果：/Devices/A4……A71/data/Containers/Bundle/Application/C597……4AD1/All.app

7. 获取MainBundle（NSURL对象）

NSURL \*URL = [NSBundle mainBundle]

打印结果：/var/mobile/Containers/Bundle/Application/5A01……896E/All.app

打印结果：/Devices/A4……A71/data/Containers/Bundle/Application/C597……4AD1/All.app

文件系统操作方法汇总

返回指定文件NSString

NSString \*htmlPath = [[NSBundle mainBundle] pathForResource:@"hehe" ofType:@"html"];

获取应用根目录全路径：[NSBundle mainBundle]bundlePath]

返回指定文件NSURL

NSURL \*URL = [[NSBundle mainBundle] URLForResource:@"HelloWorld" withExtension:@"momd"];

获得文件协议型的地址（该方法将传入路径打包为文件传输协议格式 —— file://……）

NSURL \*bundleUrl = [NSURL fileURLWithPath:[[NSBundle mainBundle]bundlePath]];

检测文件是否存在

NSFileManager \*fileManager = [NSFileManager defaultManager];

NSString \*writableDBPath = [self applicationDocumentsDirectoryFile];

BOOL dbexits = [fileManager fileExistsAtPath:writableDBPath];

文件拷贝

NSError \*error;

BOOL b = [fileManager copyItemAtPath:defaultDBPath toPath:writableDBPath error:&error];

----------库 , Bundle , Nib , CocoaPods-------------------------------

库的创建于使用 - 项目NotesLibrary

制作库（或添加库）

1. 添加其他库：项目空白处右键—>Add Files—>选择.xcodeproj后缀文件（我添加了iOSStructure）

2. 静态链接库提供给使用者只有两个： .a（编译之后的链接库文件） 、.h（提供的头文件）

3. 使.h文件能够被复制到产品发布目录下：点击项目—>右侧Build Phases—>左上+ New Header Phases —>打开Headers 点下方+号添加你的头文件—>将头文件拖到public（我仅拖入了view.h）

Public中.h文件可被其他工程访问，在应用编译时也被复制到产品发布目录(未同使用者建立依赖还不行)

4. 编译该库

上述操作会使得其他项目导入库的时候，在该项目编译完后，能够自动将库.a文件放在Debug-iphoneos也 就是APP根目录下，而头文件会放在Debug-iphoneos/usr/local/include目录下

使用库文件（同使用者建立依赖）

添加库：点击项目—>右侧Build Phases—>Link Binary With Libraries—>+号添加库文件

设置头文件的搜索位置：点击项目—>右侧Build Setting—>Search Paths—>User Header Search—>展开 双击弹出对话框添加$(BUILT\_PRODUCTS\_DIR)（产品发布目录）并选择recursive（递归添加目录路径）

也可以直接将路径改为：$(BUILT\_PRODUCTS\_DIR)/usr/local/include

注意：发现曾经行得通的项目失败：只需要将各个库文件分别再编译一下就行了

制作APP库 - without资源

说在前面：

1. 如何去除自动加载main故事版，但引出获取错误的screen.bounds问题
2. 如何使用代码加载window和rootController
3. 如何加载资源中的storyBoard
4. 如何不带加载资源的代码，制作APP库，并使用和运行
5. 如何带加载资源的代码，使用APP库（手动引入storyBoard）

先去除故事版initial Controller这个设定，否则可能会自动加载一下故事版，然后被抹去从而影响效率

然而即使这样，项目在运行时还是会自动加载storyBoard，如何纯代码而不依赖storyBorad？？

解决：在APP-Target项目的info.plist中删除两行和launchScreen，mainStoryBoard有关的即可。

但是：即使在代码中动态获取screen.bounds，加载出的窗口还是4s大小，而放回launchScreen就恢复了

问题：storyBoard和LaunchScreen在初始化窗口尺寸上有什么作用？？对UIScreen有什么影响

然后插入如下代码就可以单独加载代码文件运行

- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions {

FirstViewController \*firstController = [[FirstViewController alloc]init];

\_window = [[UIWindow alloc]initWithFrame:UIScreen.mainScreen.bounds];

\_window.rootViewController = firstController;

\_window.backgroundColor = [UIColor whiteColor];

[\_window makeKeyAndVisible];

return YES;

}

如何将故事板的Tab Controller也加载进来

故事版中将指定controller命名一个ID就可以用如下代码获得controller对象了

然后插入window的root里面去执行

UIStoryboard\* mainStoryboard = [UIStoryboard storyboardWithName:@"Main" bundle:nil];

UITabBarController \*tabController = [mainStoryboard

instantiateViewControllerWithIdentifier:@"tabController"];

不带storyBoard，仅代码包装为静态库然后提供接口，供生成APP-Target的项目中的main函数调用

新建iOS静态库项目

导入之前APP的所有代码，资源（Assets、Base/storyBoard）

Build Phases中Compile Source添加所有 .h/.m 文件。加Headers然后Public里拖入Delegate.h

编译该库

向AllHere项目的Build Phases中的Link Library中添加上面的库，在Build Setting中search Path添加……

向AllHere项目的main文件中的main函数中引入Delegate.h，并跑这个类

最后成功运行，但是仅仅显示viewController，而不是TabController

带读取storyBoard的代码（失败）

我怀疑是因为代码在读取storyBoard的时候默认是在当前资源目录下搜索的，虽然搜到mian.storyBoard， 但发现里面并没有ID为tabController的，所以crash了（报错信息：Could not find a storyboard named 'Pull' in bundle NSBundle </Users/zhengyiwei/Library/Developer/CoreSimulator/Devices/

A4B35F0A-FAFB-43C0-B1A8-BD677E5A8A71/data/Containers/Bundle/Application

/23282644-2AA6-4458-8E12-5EC2B1234822/AllHere.app）

所以在下文中，我可能想办法将main. storyBoard改名pull. storyBoard，并使得All项目引入库的时候 自动引入该故事版在资源目录中，使得代码能够在资源目录中找到正确的storyBoard

手动添加Pull.StoryBoard到All中，然后运行成功，并未尝试自动引入storyBoard

库的基本知识 - 引出Bundle

网址：http://www.cnblogs.com/wendingding/p/3893095.html

什么是库：库是程序代码的集合，是共享程序代码的一种方式

库的分类：根据源代码的公开情况，库可以分为2种类型

1. 开源库：公开源代码，能看到具体实现，比如SDWebImage、AFNetworking

2. 闭源库：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，主要为：静态库、动态库

静态库和动态库：

静态库和动态库的存在形式： 静态库：.a 和 .framework 动态库：.dylib 和 .framework

静态库和动态库在使用上的区别

静态库：链接时，静态库会被完整地复制到可执行文件中，被多次使用就有多份冗余拷贝（图1所示）

动态库：链接时不复制，程序运行时由系统动态加载到内存，供多个程序程序调用，而系统只加载一次，

注意：大部分框架都是动态链接库的形式。因为只有苹果才能在iOS设备上安装动态库，所以我们无法创建这 种类型的框架。（代码使用动态库的话，代码无法通过AppStore）

头文件太多该怎么办？

　　 如果静态库需要暴露出来的 .h 比较多，可以考虑创建一个主头文件（一般 主头文件 和 静态库 同名）

　　 在主头文件中包含所有其他需要暴露出来的 .h 文件

　　 使用静态库时，只需要#import 主头文件，实际上苹果官方就是这么做的，例如：#import <UIKit/UIKit.h> 静态库中包含了Category（分类）会报错

如果静态库中包含了Category，有时候在使用静态库的工程中会报“方法找不到”的错误（unrecognized selector sent to instance）

解决方案：在使用静态库的工程中Build Setting中配置Other Linker Flags为-ObjC

我们为什么需要框架 - 静态框架的破解使用请见下面

想用一种开发者友好的方式共享库很麻烦的。你不仅需要包含库本身，还要copy入所有的头文件，资源等等。

苹果解决这个问题的方式是框架（framework）。基本上，这是含有固定结构并包含了引用该库时所必需的所 有东西的文件夹。不幸的是，iOS禁止所有的动态库。同时，苹果也从Xcode中移除了创建静态iOS框 架的功能。

Bundle与资源 - Bundle的使用请见下面

自己制作的静态库中要用到的图片资源，不建议直接以png的后缀名方式拖到项目中使用，而是推荐使用放到 bundle文件中。这样可以避免静态库的图片名和使用静态库的项目中存在的图片产生冲突。

Bundle封装资源/访问

新建Bundle – 1 —— 保留SDK为OSX

1. 新建target或项目，在OSX中选择Bundle

2. 不修改Build Setting中的SDK，plist路径等，不删除任何自动生成的文件

3. 向同名group中Copy式的添加文件，这时可以看到build phases中的copy resource同步添加了这些

4. cmd + b编译bundle，在finder中查看，显示包内容就可以看到添加进去的文件了

新建Bundle – 2 —— 修改SDK为iOS

1. 文件不能使故事板，否则会提示：storyBoards does not support target device type "Mac"

2. 针对此问题修改SDK为iOS后，发现虽然编译成功，但是所有后续添加的文件都没有导入到包中

3. 在新建Bundle项目后编译前就设置SDK为iOS，发现添加图片和故事板编译能成功，且在目标bundle中 能看到图片被加入，只是show in finder无法工作

4. 第3条的基础上，改回SDK为OSX，在去除故事版的引用后，编译成功，且show in finder能工作，但是 finder到的路径是第3条同层文件夹下的debug文件夹里

5. 第4条的基础上，再改SDK为iOS，编译成功，发现show in finder查到的bundle没变化，但是 Debug-iphonesimulator中的bundle变化了

6. OSX 与 iOS的bundle结构上不一样，iOS的就是文件的铺展，OSX的有以content为根目录的结构

7. 总结：iOS和OSX的bundle编译结果路径不一样，Xcode的默认finder显示的是OSX的编译结果

使用Bundle访问source

1. 将上面iOS编译出的bundle强行复制到桌面，在通过All项目引用进来

2. 然后修改delegate代码为：

NSBundle \*bundle = [NSBundle bundleWithURL:[[NSBundle mainBundle]

URLForResource:@"houhou" withExtension:@"bundle"]];

UIStoryboard\* mainStoryboard = [UIStoryboard storyboardWithName:@"Pull" bundle:bundle];

3. 上述代码通过资源目录（也就是mainBundle）找到了对应的bundle，然后读取故事板加载

4. 该方法绝对不可取：

访问资源的代码必须全部换为bundle的访问

每次资源更换都需要重新编译资源，在finder中找到bundle，复制到APP项目，APP引入 一系列操作

Nib文件是什么

类的编码和解码 — 回忆《过渡区 — 对象归档》

Nib文件是Interface Builder文档。通常会使用Interface Builder设计应用程序的可视部分

有些时候，您可能也用它来配置一些不可视的对象，例如控制器对象（用于管理窗口和视图）。实际上，所有这些操作都是在编辑一份Interface Builder文档，即在创建一份对象图。保存文件的时候，就是将该对象图进行归档，加载文件的时候，就是将对象图进行解档。

Nib文件可以包含占位对象，这种对象用于映射存在于文档之外的对象。这些对象虽然不在文档中，但却可能含有

文档内对象的引用，或者被文档内的对象引用。文件拥有者就是一种特殊的占位对象。

在程序运行的时候，您可以使用 loadNibNamed:owner:(或者此方法变种)加载Nib文件。在Nib文件中，文件拥

有者是个占位对象，它表示您通过owner参数传给此方法的对象。加载Nib文件的时候，所有在Interface

Builder中建立的关联（文件拥有者和其他对象之间的关联）都能够在运行时重新建立起来。

静态工厂

nib文件是一种数据文件，用于存储可在应用程序需要时使用的一些“冻结”的对象。大多数情况下，应用程序使用nib文件来存储构成用户界面的窗口和视图。当您将nib文件载入应用程序时，nib装载代码会将文件中的内容转化为应用程序可以操作的真正对象。通过这个机制，nib文件省去了用代码创建那些对象的工作。

使用静态框架？（看网址而深入学习）

网址：http://www.oschina.net/question/234345\_55508

概述：“伪”框架是通过破解Xcode的目标Bundle（使用某些脚本）来实现的。它在表面上以及使用时跟静态框架 并无区别。“伪”框架项目的功能几乎和真实的框架项目没有区别（不是全部）。

“嵌入”框架是静态框架的一个包装，以便Xcode能获取框架内的资源（图片、plist、nib等）。

CocoaPods依赖管理详述（没看懂）

其解决的问题

在没有使用CocoaPods以前，我需要:

把这些第三方开源库的源代码文件复制到项目中，或者设置成git的submodule。

对于这些开源库通常需要依赖系统的一些framework，我需要手工地将这些framework一一增加到项目依赖

中，比如通常情况下，一个网络库就需要增加以下framework: CFNetwork, SystemConfiguration, MobileCoreServices, CoreGraphics, zlib。

对于某些开源库，我还需要设置-licucore或者 -fno-objc-arc等编译参数管理这些依赖包的更新。

在使用CocoaPods之后，我只需要将用到的第三方开源库放到一个名为Podfile的文件中，然后执行pod install。

CocoaPods就会自动将这些第三方开源库的源码下载下来，并且为我的工程设置好相应的系统依赖和编译参数。

原理简述

CocoaPods会将所有的第三方库以target的方式组成一个名为Pods的工程，该工程就放在刚才新生成的Pods目

录下。整个第三方库工程会生成一个名称为libPods.a的静态库提供给我们自己的CocoaPodsTest工程使用。

对于资源文件，CocoaPods提供了一个名为Pods-resources.sh的bash脚本，该脚本在每次项目编译的时候都会

执行，将第三方库的各种资源文件复制到目标目录中。

原来的工程设置已经被更改了，这时候我们直接打开原来的工程文件去编译就会报错，只能使用新生成的workspace

来进行项目管理。

CocoaPods通过一个名为Pods.xcconfig的文件来在编译时设置所有的依赖和参数。

核心组件 — http://objccn.io/issue-6-4/

CocoaPods是用 Ruby 写的，并由若干个 Ruby 包 (gems) 构成的。在解析整合过程中，最重要的几个 gems 分

别是： CocoaPods/CocoaPods, CocoaPods/Core, 和 CocoaPods/Xcodeproj

CocoaPods/CocoaPod

这是是一个面向用户的组件，每当执行一个 pod 命令时，这个组件都将被激活。该组件包括了所有使用 CocoaPods 涉及到的功能，并且还能通过调用所有其它的 gems 来执行任务。

CocoaPods/Core

Core 组件提供支持与 CocoaPods 相关文件的处理，文件主要是 Podfile 和 podspecs。

Podspec

.podspec 也是一个文件，该文件描述了一个库是怎样被添加到工程中的。它支持的功能有：列出源文件、framework、编译选项和某个库所需要的依赖等。

CocoaPods/Xcodeproj

这个 gem 组件负责所有工程文件的整合。它能够对创建并修改 .xcodeproj 和 .xcworkspace 文件。它也可以作为单独的一个 gem 包使用。如果你想要写一个脚本来方便的修改工程文件，那么可以使用这个 gem。

运行过程 — http://objccn.io/issue-6-4/

**读取 Podfile 文件**

你是否对 Podfile 的语法格式感到奇怪过，那是因为这是用 Ruby 语言写的。相较而言，这要比现有的其他

格式更加简单好用一些。

在安装期间，第一步是要弄清楚显示或隐式的声明了哪些第三方库。在加载 podspecs 过程中，CocoaPods

就建立了包括版本信息在内的所有的第三方库的列表。Podspecs被存储在本地路径 **~/.cocoapods** 中。

**版本控制和冲突**

CocoaPods 使用[语义版本控制 - Semantic Versioning](http://semver.org/) 命名约定来解决对版本的依赖。由于冲突解决系统

建立在非重大变更的补丁版本之间，这使得解决依赖关系变得容易很多。例如，两个不同的 pods 依赖于 CocoaLumberjack 的两个版本，假设一个依赖于 **2.3.1**，另一个依赖于 **2.3.3**，此时冲突解决系统可以使用最新的版本 **2.3.3**，因为这个可以向后与 **2.3.1** 兼容。

但这并不总是有效。有许多第三方库并不使用这样的约定，这让解决方案变得非常复杂。

当然，总会有一些冲突需要手动解决。如果一个库依赖于 CocoaLumberjack 的 **1.2.5**，另外一个库则依赖

于 **2.3.1**，那么只有最终用户通过明确指定使用某个版本来解决冲突。

**加载源文件**

CocoaPods 执行的下一步是加载源码。每个**.podspec**文件都包含一个源代码的索引，这些索引一般包裹一

个 git 地址和 git tag。它们以 commit SHAs 的方式存储在 **~/Library/Caches/CocoaPods** 中。这个路径中文件的创建是由 Core gem 负责的。

CocoaPods 将依照 **Podfile**、**.podspec** 和缓存文件的信息将源文件下载到 **Pods** 目录中。

**生成 Pods.xcodeproj**

每次 **pod install** 执行，如果检测到改动时，CocoaPods 会利用 Xcodeproj gem 组件对 **Pods.xcodeproj** 进行更新。如果该文件不存在，则用默认配置生成。否则会将已有的配置项加载至内存中。

**安装第三方库**

当 CocoaPods 往工程中添加一个第三方库时，不仅仅是添加代码这么简单，还会添加很多内容。由于每个

第三方库有不同的 target，因此对于每个库，都会有几个文件需要添加，每个 target 都需要：

1. 一个包含编译选项的 **.xcconfig** 文件
2. 一个同时包含编译设置和 CocoaPods 默认配置的私有 **.xcconfig** 文件
3. 一个编译所必须的 **prefix.pch** 文件
4. 另一个编译必须的文件 **dummy.m**

一旦每个 pod 的 target 完成了上面的内容，整个 **Pods** target 就会被创建。这增加了相同文件的同时，

还增加了另外几个文件。如果源码中包含有资源 bundle，将这个 bundle 添加至程序 target 的指令将被添加到 **Pods-Resources.sh** 文件中。还有一个名为 **Pods-environment.h** 的文件，文件中包含了一些宏，这些宏可以用来检查某个组件是否来自 pod。最后，将生成两个认可文件，一个是 **plist**，另一个是 **markdown**，这两个文件用于给最终用户查阅相关许可信息。

**写入至磁盘**

直到现在，许多工作都是在内存中进行的。为了让这些成果能被重复利用，我们需要将所有的结果保存到一个

文件中。所以 **Pods.xcodeproj** 文件被写入磁盘，另外两个非常重要的文件：**Podfile.lock** 和 **Manifest.lock** 都将被写入磁盘。

**Podfile.lock**

这是 CocoaPods 创建的最重要的文件之一。它记录了需要被安装的 pod 的每个已安装的版本。如果你想知道已安装的 pod 是哪个版本，可以查看这个文件。推荐将 Podfile.lock 文件加入到版本控制中，这有助于整个团队的一致性。

**Manifest.lock**

这是每次运行 **pod install** 命令时创建的 **Podfile.lock** 文件的副本。如果你遇见过这样的错误 **沙盒文件与 Podfile.lock 文件不同步 (The sandbox is not in sync with the Podfile.lock)**，这是因为 Manifest.lock 文件和 **Podfile.lock** 文件不一致所引起。由于 **Pods** 所在的目录并不总在版本控制之下，这样可以保证开发者运行 app 之前都能更新他们的 pods，否则 app 可能会 crash，或者在一些不太明显的地方编译失败。

**xcproj**

如果你已经依照我们的建议在系统上安装了 [xcproj](https://github.com/0xced/xcproj)，它会对 **Pods.xcodeproj** 文件执行一下 **touch** 以将其转换成为旧的 ASCII plist 格式的文件。为什么要这么做呢？虽然在很久以前就不被其它软件支持了，但是 Xcode 仍然依赖于这种格式。如果没有 xcproj，你的 **Pods.xcodeproj** 文件将会以 XML 格式的 plist 文件存储，当你用 Xcode 打开它时，它会被改写，并造成大量的文件改动。

**结果**

运行 **pod install** 命令的最终结果是许多文件被添加到你的工程和系统中。这个过程通常只需要几秒钟。当然没有 Cocoapods 这些事也都可以完成。只不过所花的时间就不仅仅是几秒而已了。

CocoaPods & Ruby环境安装

http://blog.csdn.net/iunion/article/details/17010267

http://www.th7.cn/Program/IOS/201410/296605.shtml

简介：

当你开发iOS应用时，会经常使用到很多第三方开源类库，比如JSONKit，AFNetWorking等等。可能某个 类库又用到其他类库，所以要使用它

iOS Mac系统下Ruby环境安装

要安装cocoapods需要Ruby环境，安装Ruby环境首先需要安装Xcode（因为xcode开发工具，他帮我们 安装好Unix环境需要的开发包）然后需要安装homebrew。接下来需要安装RVM 最后安装Ruby环境

Xcode->homebrew->RVM->Ruby->CocoaPats;

homebrew：http://brew.sh/index\_zh-cn.html —— （下面某个安装隐式访问了这个东西）

homebrew是一个包管理器，用于在mac上安装一些os x上没有的UNiX工具（比如wget）（wget我 不知道是什么），但是作为一个菜鸟来说，只要知道它如同window的350软件管理器就行了，用来 安装一些软件。

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

大量的访问了：/usr/local/ /usr/bin/

RVM：Ruby Version Manager

是一个非常好用的Ruby版本以及安装工具。就是用来安装和控制Ruby版本的工具

$ curl -L https://get.rvm.io | bash -s stable 剩下按照其英语提示进行

建议使用：rvm remove 、info 、reset 、reinstall 等命令来控制需要安装的ruby和环境变量配置

Ruby: $ rvm install 2.2.0

Ruby是一种面向对象的脚本语言，简单易用，功能强大。跨平台和可移植性好等。其实就是种脚本语言。

ruby的安装包含了：gem的安装 、和环境变量的配置。而gem用来安装cocoaPads

CocoaPads

$ sudo gem install cocoapods （可能会遇到中国防火墙，可查询网址寻找帮助）

gem ($ gem update)是用于ruby程序和程序库的一套打包系统，它让开发人员可以把自己的ruby程序 库打包成 一种易于维护和安装的形式。

DEBUG: warning: Insecure world writable dir /users/zhengyiwei/tomcat in PATH, mode 040777

SOLVE：Your home folder should only be writable by you, not by anyone else. The reason gem is complaining about this is that you have folders in your PATH that are inside your (insecure) home folder, and that means that anyone who wants to could hack you by renaming and moving your .rvm folder and replacing it with an impostor.

To fix your home folder, run chmod go-w /users/zhengyiwei/tomcat.

这拒绝组成员和其他人创建或删除 mydir （go-w ）中的文件的许可权：

等同于命令序列：chmod g-w mydir chmod o-w mydir

使用：利用CocoaPods，在项目中导入AFNetworking类库

AFNetworking类库在GitHub地址是：https://github.com/AFNetworking/AFNetworking

为了确定AFNetworking是否支持CocoaPods，可以用CocoaPods的搜索功能验证一下。在终端中输 入： $ pod search AFNetworking 过几秒钟之后，你会在终端中看到关于AFNetworking类库的一些 信息。这说明，AFNetworking是支持CocoaPods，所以我们可以利用CocoaPods将AFNetworking 导入你的项目中。

CocoaPods为什么能下载AFNetworking呢，有个文件来控制CocoaPods该下载什么。这个文件就叫 做“Podfile”（注意，一定得是这个文件名，而且没有后缀）。你创建一个Podfile文件，然后在里面添加 你需要下载的类库，注意，Podfile文件应该和你的工程文件.xcodeproj在同一个目录下。

创建Podfile：

inhibit\_all\_warnings!

platform :ios, '7.0' //最低版本要求

pod 'AFNetworking', '~> 3.1.0'

在终端中的当前项目目录下，运行以下命令：$ pod install

Analyzing dependencies Downloading dependencies

……

[!] From now on use `CocoaPodsDemo.xcworkspace`.

注意最后一句话，意思是：以后打开项目就用 CocoaPodsDemo.xcworkspace 打开，而不是之前 的.xcodeproj文件。为什么会出现.xcodeproj文件呢。这正是你刚刚运行$ pod install命令产生的新文 件。除了这个文件，你会发现还多了另外一个文件“Podfile.lock”和一个文件夹“Pods”。 点击 CocoaPodsDemo.xcworkspace 打开之后工程之后，项目Xcode目录结构如下图：

其他技巧

不去更新spec仓库

pod install --verbose --no-repo-update

pod update --verbose --no-repo-update

所有的项目的Podspec文件都托管在https://github.com/CocoaPods/Specs。第一次执行pod setup时，CocoaPods会将这些podspec索引文件更新到本地的 ~/.cocoapods/目录下，这个索引文件比较大，有80M左右。

关于Podfile文件编辑时，第三方库版本号的各种写法

pod ‘AFNetworking’ //不显式指定依赖库版本，表示每次都获取最新版本

pod ‘AFNetworking’, ‘2.0’ //只使用2.0版本

pod ‘AFNetworking’, ‘>2.0′ //使用高于2.0的版本

pod ‘AFNetworking’, ‘>=2.0′ //使用大于或等于2.0的版本

pod ‘AFNetworking’, ‘<2.0′ //使用小于2.0的版本

pod ‘AFNetworking’, ‘<=2.0′ //使用小于或等于2.0的版本

pod ‘AFNetworking’, ‘~>0.1.2′ //使用大于等于0.1.2但小于0.2的版本，相当于>=0.1.2并且<0.2.0

pod ‘AFNetworking’, ‘~>0.1′ //使用大于等于0.1但小于1.0的版本

pod ‘AFNetworking’, ‘~>0′ //高于0的版本，写这个限制和什么都不写是一个效果，都表示使用最新版本

其他知识点 — $ pod --help

1. pod 'SAMCategories', '0.5.2'pod 'Nimbus/AttributedLabel', '1.2.0'

注意到Nimbus/AttributedLabel和其他pod不一样，多了一个"/"？大的pods常常将他们的库拆 分成核心模块和一些小的子模块。在这个例子中，“/”表示AttributedLabel是Nimbus的子模块。

2. sstoolk 是一个便捷的工具包，用于解决iOS开发者们经常遇到的问题。

Nimbus ，全称NimbusKit，是另一个很有用的工具包。

3. 如果你加入的引用有很多警告，甚至编译报错，很可能它已经不再被维护了。

遇到这种情况，你可以在 CocoaPods website 上查找一个替换的组件。

还有大部分的警告都来源于iOS7废弃了一些方法，你可以安全地无视掉它们。为了让CocoaPods 的引入不显示警告，在Podfile最上方加上：inhibit\_all\_warnings!

4. 对Podfile修改安装之后编译项目，可能会出现如下警告 ：

The sandbox is not in sync with the Podfile.lock. Run 'pod install' or update your CocoaPods installation.

此时，正确的做法是：

I. Project Cleanup

In the project navigator, select your project

Select your target

Remove all libPods\*.a in Linked Frameworks and Libraries /删除所有和pods相关的库依赖

II. Update CocoaPods

Launch Terminal and go to your project directory.

Update CocoaPods using the command pod install

5. DEBUG: Analyzing dependencies

xcode-select: error: tool 'xcodebuild' requires Xcode, but active developer directory '

SOLVE：原因是Xcode重命名后，xcodebuild找不到原来的Xcode了。解决方法：在终端输入命令

$ sudo xcode-select --switch /Applications/Xcode-Beta.app/Contents/Developer/

6. DEBUG：Updating local specs repositories

SOLVE：pod install 换成pod install --verbose --no-repo-update这个命令，前面的命令被墙了

7. DEBUG：NoMethodError - undefined method `to\_ary'

SOLVE（失败）: http://www.tuicool.com/articles/InIBbaf

1. 前段时间 Ruby 发布了 2.3 ，手贱就直接升级了一下，结果 gem 全部都要重装，但是这 并不是最坑的。最麻烦的是安装完 cocoapods 之后， pod install 和 pod update ，都会报 错 NoMethodError - undefined method 'to\_ary'

2. 其实就是加了一个方法， 使用 vim 或者你熟悉的编辑工具打开 lazy\_specification.rb

/Users/zhengyiwei/.rvm/rubies/ruby-2.3.0/lib/ruby/gems/2.3.0/gems/cocoapods-0.39.0/lib/cocoapods/resolver

在第16行也就是 subspec\_by\_name 的方法上面加入下面代码：

def respond\_to\_missing?(method, include\_all = false)

specification.respond\_to?(method, include\_all)

end

SOLVE（成功）:尝试回退到ruby2.2.0：rvm remove all —> rvm install 2.2.0

研究与实验

研究结果单条

1. 库文件编译完后暴露接口是怎么回事，Finder中查看暴露的接口存在吗？

有的，使用finder查看可以看到在 usr/local/include 目录下有你共享的头文件

1. All在space中添加了某库，当某库的代码发生变化并保存后（并未编译），编译运行All发现更改被应用了

每次编译All，都会重编译All通过Space引用的库，那么这样分别打包为库加速编译就显得没意义了

这是开始