iOS线程同步

影子猫
iOS技术分享

iOS开发基础.png
线程同步：即当有一个线程在对内存进行操作时，其他线程都不可以对这个内存地址进行操作，直到该线程完成操作， 其他线程才能对该内存地址进行操作。
所以这里同步应该不是一起、共同完成的意思，可理解为协调就是按预定的先后次序进行工作，好比：不要和我抢了，你先等会我做完了你在做 。
线程同步目的为了多个线程都能很好的工作，合理的访问系统资源不争不抢、和谐共处。iOS开发中常用的保持线程同步有以下几种：

- 通过线程加锁
- 串行队列
- GCD

线程加锁
常用的几种形式的锁

- 1、 @synchronized

- (void)myMethod:(id)anObj { @synchronized(anObj) { //执行的代码操作 } }
通过 `synchronized指令` 自动的添加一个互斥锁，底层通过pthread_mutex实现。通过对一段代码的使用进行加锁。其他试图执行该段代码的线程都会被阻塞，直到加锁线程退出执行该段被保护的代码段。
当在@synchronized()代码块中抛出异常的时候，Objective-C运行时会捕获到该异常，并释放信号量，并把该异常重新抛出给下一个异常处理者。
一个线程是可以以递归的方式多次调用 `myMethod` 。
关于参数 `anObj` ；
作为一个唯一标识符来标记当前线程加锁操作必须是个对象类型，所以对于同一个操作不同的线程应该用同一个对象，否则无法起到标记加锁的作用。 不能为空nil。
常见的基本都是 `self`
@synchronized(self) { //执行的代码操作 }
self作为标记符十分常见，但是很明显会有一个问题：
//方法1 - (void)myMethod1:(id)anObj {
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT , 0), ^{
@synchronized(anObj) { //执行的代码操作 } }); } //方法2 - (void)myMethod2:(id)anObj {
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT , 0), ^{
@synchronized(anObj) { //执行的代码操作 } }); } myMethod1(self); myMethod2(self)
如果myMethod1、myMethod2没用任何关系，如果此时执行myMethod1，那么myMethod2就只能等待其执行完成。所以这种情况更细的粒度来加锁，使用各自的对象互不影响更为合理。

- 2、NSLock

NSLock * lock = [[NSLock alloc]init]; [lock lock]; //执行的代码操作 [lock unlock];
底层通过pthread_mutex实现；方法lock、unlock必须成对出现，必须在同一个线程中操作否则无效。不支持递归，如果多次调用会造成死锁。
如果多个线程共用一个lock ，一个线程加锁后其他请求加锁的线程会形成一个等待队列、按照先进先出的规则等待锁释放后再加锁*(待验证)*。

- 3、NSRecursiveLock 递归锁类似NSLock，但它可以在同一个线程中反复加锁且不会造成死锁。
- 4、 NSCondition 基于信号量方式实现的锁对象，提供单独的信号量管理接口。底层通过 `pthread_cond_t`实现。

NSCondition对象包含锁和条件检测功能，类似于生产者和消费者：消费者消费资源如果没有就继续等待，生产者提供资源然后发出信号激活消费者。锁的作用就是用来保护这一操作防止被其他线程干扰。
`DEMO:`
isWait = true; condition = [[NSCondition alloc]init];
__weak typeof(self) weakSelf = self;
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[weakSelf _user]; });
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[weakSelf _produce]; });
-(void)_user{ [condition lock]; while (isWait) { //等待其他线程发出信号,[condition signal]; //阻塞当前线程 NSLog(@"等待条件满足"); [condition wait]; } { //执行操作 NSLog(@"执行操作"); } //完成 [condition unlock]; NSLog(@"完成"); } -(void)_produce{ [condition lock]; isWait = false; [condition signal]; [condition unlock]; }
输出结果:
[13781:212898] 等待条件满足 [13781:212898] 执行操作 [13781:212898] 完成

- 5、 NSConditionLock 可以使用特定值来加锁和解锁，和 `NSCondition` 表现差不多。

- (instancetype)initWithCondition:(NSInteger)condition 参数 `condition` 作为标识符更容易理解， `lockWhenCondition` 获取指定标记的锁没有的话就阻塞当前线程， `unlockWithCondition:` 释放指定标记的锁，等他的线程获取锁然后继续执行操作。

使用上比 `NSCondition` 更方便些，代码更简洁。
用 `NSConditionLock` 改写以上代码:
-(void)_testConditionLock{ __weak typeof(self) weakSelf = self;
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[weakSelf _user1]; });
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[weakSelf _produce1]; }); } -(void)_user1{ NSLog(@"等待条件满足"); [conditionLock lockWhenCondition:11]; NSLog(@"条件满足了"); { //执行操作 NSLog(@"执行操作"); } //完成 [conditionLock unlockWithCondition:0]; NSLog(@"完成"); } [self _testConditionLock];
输出结果:
[7812:120141] 等待条件满足 [7812:120137] 生成条件中... [7812:120141] 条件满足了 [7812:120141] 执行操作 [7812:120141] 完成

- 6、 其他不常用的锁 pthread_mutex pthread_mutex(recursive) POSIX标准的unix多线程,C 语言下多线程实现。

OSSpinLock:自旋锁，一直轮询等待时会消耗大量 CPU 资源。
串行队列
通过创建一个串行队列，把我们的操作添加到队列。
dispatch_queue_t queue =
dispatch_queue_create("com.queue.test",DISPATCH_QUEUE_SERIAL); dispatch_async(queue, ^{ NSLog(@"task 1"); }); dispatch_async(queue, ^{ NSLog(@"task 2"); });
dispatch_async(queue, ^{ NSLog(@"task 3"); });
`感觉创建队列、添加操作到队列太麻烦，不够简洁而且队列的调度肯定占用不少资源.`
GCD
通过 `dispatch_semaphore` 信号量实现线程同步
dispatch_semaphore_create(long value); dispatch_semaphore_wait(dispatch_semaphore_t dsema, dispatch_time_t timeout);//-1 dispatch_semaphore_signal(dispatch_semaphore_t dsema);//+1
`dispatch_semaphore_wait` 在信号量为0时会阻塞当前线程，等待 `dispatch_semaphore_signal` 释放信号然后继续执行。
用信号量改写以上代码:
-(void)_testSemaphore{ __weak typeof(self) weakSelf = self;
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[weakSelf _user2]; });
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[weakSelf _produce2]; }); } -(void)_user2{ NSLog(@"等待条件满足");
dispatch_semaphore_wait(semaphore,DISPATCH_TIME_FOREVER); NSLog(@"条件满足了"); { //执行操作 NSLog(@"执行操作"); } //完成 NSLog(@"完成"); } -(void)_produce2{ NSLog(@"生成条件中..."); dispatch_semaphore_signal(semaphore); }
semaphore = dispatch_semaphore_create(0); NSLog(@"初始化信号0"); [self _testSemaphore];
输出:
[9581:159120] 初始化信号0 [9581:159194] 生成条件中... [9581:159195] 等待条件满足
[9581:159195] 条件满足了 [9581:159195] 执行操作 [9581:159195] 完成
总结
常用的线程间同步方式就这些了，我实际中用的信号量和NSLock比较多。至于其他的不是因为不好而是因为习惯了，不到很必须的时候我感觉都差不多。真正因为其本身所占用的开销一般可忽略不计。
整理此文前前后后持续了一周的时间，总算进一步加深了认知。写完了才感觉这些知识才是自己的，然后在慢慢吸收、消化，伴随我们一步步的走向强大。
未来的路很长，不知道会走多远，只想走好脚下的每一步^_^！

发布于 2020-11-03

iOS 多线程

推荐阅读

iOS多线程到底不安全在哪里?
Larry 发表于码力全开工...

Android 性能优化：多线程系列开篇
ITGeGe

Android开发之多线程
轩羽

你知道Thread线程是如何运作的吗?
背景介绍我们在Android开发过程中，几乎都离不开线程。但是你对线程的了解有多少呢？它完美运行的背后，究竟隐藏了多少不为人知的秘密呢？线程间互通暗语，传递信息究竟是如何做到的呢？Loo...
小开

还没有评论

写下你的评论...