iOS之多线程

NSThread

简介

苹果公司的Cocoa框架共支持三种多线程机制，分别为NSThread、GCD（Grand Central Dispatch）、Cocoa NSOperation。

NSThread是官方推荐的线程处理方式，它在处理机制上，需要开发者负责手动管理Thread的生命周期，包括子线程与主线程之间的同步等.

线程共享同一应用程序的部分内存空间，它们拥有对数据相同的访问权限。你得协调多个线程 对同一数据的访问，一般做法是在访问之前加锁，这会导致一定的性能开销。

在 iOS 中我们可以使用多种形式的 thread。 比其他两个轻量级 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。

demo

- NSThread有2种创建形式：

    1. 实例方法--直接创建线程并开始线程：

```
 - (instancetype)initWithTarget:(id)target selector:(SEL)selector object:(null
able id)argument
```

    1. 类方法--先创建线程对象，然后运行线程操作，在运行线程操作前可以设置线程的优先级等信息：

```
+ (void)detachNewThreadSelector:(SEL)selector toTarget:(id)target withObject:(
nullable id)argument
```

- 线程间的通讯:(NSObject实例方法)

    在主线程中操作：

```
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(nullable id)arg
 waitUntilDone:(BOOL)wait modes:(nullable NSArray<NSString *> *)array
```

    在指定线程中操作：

```
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(nullable id)arg waitUntilDone:(BOOL)wait modes:(nullable NSArray<NSString *> *)array
```

在当前线程中：

```
- (void)performSelector:(SEL)aSelector withObject:(nullable id)anArgument afterDelay:(NSTimeInterval)delay inModes:(NSArray<NSString *> *)modes
```

取消发送给当前线程的某个消息：

```
+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget selector:(SEL)aSelector object:(nullable id)anArgument
```

NSOperation

简介

如果需要让线程同时并行运行多个，可以将线程加入队列（Queue）中。

NSOperationQueue类就是一个线程队列管理类，他提供了线程并行、队列的管理。可以认为NSOperationQueue就是一个线程管理器。

通过addOperations方法，我们可以一次性把多个（数组形式）线程添加到队列中。同时，NSOperationQueue允许通过setMaxConcurrentOperationCount方法设置队列的并行（同一时间）运行数量。

demo

建立NSoperationQueue和NSOperations

NSOperationQueue会建立一个线程管理器，每个加入到线程operation会有序的执行。

1. 建立一个NSOperationQueue的对象
2. 建立一个NSOperation的对象
3. 将operation加入到NSOperationQueue中
4. release掉operation

```
NSOperationQueue *queue = [NSOperationQueue new];
NSInvocationOperation *operation = [[NSInvocationOperation alloc] initWithTarget:self selector:@selector(doWork:) object:someObject];
[queue addObject:operation];
```

多线程互斥问题

在iOS中有以下方法解决 `多线程访问同一个内存地址的互斥同步问题` ：

1. @synchronized(id anObject) :

   (最简单的方法) 会自动对参数对象加锁，保证临界区内的代码线程安全。

   ```
   @synchronized(self) {
       // 这段代码对其他 @synchronized(self) 都是互斥的
       // self 指向同一个对象
   }
   ```

2. NSLock

   NSLock对象实现了NSLocking protocol，包含几个方法： lock:加锁、 unlock：解锁、tryLock：尝试加锁。

   如果失败了，并不会阻塞线程，只是立即返回NO。

   lockBeforeDate:在指定的date之前暂时阻塞线程（如果没有获取锁的话），如果到期还没有获取锁，则线程被唤醒，函数立即返回NO

   比如：

   ```
   NSLock *theLock = [[NSLock alloc] init];
   if ([theLock lock]) {
   //do something here
   [theLock unlock];
   }
   ```

3. NSrecursiveLock递归锁

   多次调用不会阻塞已获取该锁的线程。

   ```
   NSRecursiveLock *theLock = [[NSRecursiveLock alloc] init];
   void MyRecursiveFunction(int value) {
   [theLock lock];
   if (value != 0) {
       —value;
       MyRecursiveFunction(value);
   }
   [theLock unlock];
   }
   MyRecursiveFunction(5);
   ```

4. NSConditionLock条件锁

条件锁，可以设置条件

```
//公共部分
id condLock = [[NSConditionLock alloc] initWithCondition:NO_DATA];

//线程一，生产者
while(true) {

[condLock lockWhenCondition:NO_DATA];

//生产数据
[condLock unlockWithCondition:HAS_DATA];

}

//线程二，消费者
while (true) {

[condLock lockWhenCondition:HAS_DATA];

//消费
[condLock unlockWithCondition:NO_DATA];
}
```

5. NSDistributedLock，分布锁

   NSDistributedLock，分布锁，文件方式实现，可以跨进程 用tryLock方法获取锁。 用unlock方法释放锁。 如果一个获取锁的进程在释放锁之前挂了，那么锁就一直得不到释放了，此时可以通过breakLock强行获取锁。

GCD

Grand Central Dispatch (GCD)是Apple开发的一个多核编程的解决方法。该方法在Mac OS X 10.6雪豹中首次推出，并随后被引入到了iOS4.0中。GCD是一个替代诸如NSThread, NSOperationQueue, NSInvocationOperation等技术的很高效和强大的技术，它看起来象就其它语言的闭包(Closure)一样，但苹果把它叫做blocks。

- GCD下的dispatch_queue队列都是FIFO队列,都会按照提交到队列的顺序执行. 只是根据队列的性质,分为 `串行队列` 、 `并行队列` 。
- 同步(dispatch*sync)*、*异步方式(dispatch*async). 配合串行队列和并行队列使用. 同步队列直接提交两个任务就可以.

串行队列

```
dispatch_queue_t serilQueue = dispatch_queue_create("com.quains.myQueue", 0);
//开始时间
NSDate *startTime = [NSDate date];


__block UIImage *image = nil;

//1.先去网上下载图片
dispatch_async(serilQueue, ^{
    //下载图片
});

//2.在主线程展示到界面里
dispatch_async(serilQueue, ^{

NSLog(@"%@",[NSThread currentThread]);

// 在主线程展示
dispatch_async(dispatch_get_main_queue(), ^{
  //显示图片
});

//3.清理
dispatch_release(serilQueue);
[image release];
```

注意:

1. __block变量分配在栈,retain下,防止被回收.
2. dispatch要手动create和release.
3. 提交到主线程队列的时候,**慎用同步dispatch_sync方法,有可能造成死锁.** 因为主线程队列是串行队列,要等队列里的任务一个一个执行.所以提交一个任务到队列,如果用同步方法就会阻塞住主线程,而主线程又要等主线程队列里的任务都执行完才能执行那个刚提交的,所以主线程队列里还有其他的任务的话,但他已经被阻塞住了,没法先完成队列里的其他任务,即,最后一个任务也没机会执行到,于是造成死锁.
4. 提交到串行队列可以用同步方式,也可以用异步方式.

并行队列

采用并行队列的时候,可以采用同步的方式把任务提交到队列里去,即可以实现同步的方式

```
 dispatch_queue_t concurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR
ITY_DEFAULT, 0);
 //计时
 NSDate *startTime = [NSDate date];
//加入队列
dispatch_async(concurrentQueue, ^{
    __block UIImage *image = nil;

    //1.先去网上下载图片
    dispatch_sync(concurrentQueue, ^{
        //下载图片
    });

    //2.在主线程展示到界面里
    dispatch_sync(dispatch_get_main_queue(), ^{
        //显示图片
    });
});
```

两个同步的任务用一个异步的包起来,提交到并行队列里去,即可实现同步的方式.

使用分组方式

group本身是将几个有关联的任务组合起来,然后提供给开发者一个知道这个group结束的点. 虽然这个只有一个任务,但是可以利用group的结束点,去阻塞线程,从而来实现同步方式.

例如让后台2个线程并行执行，然后等2个线程都结束后，再汇总执行结果。 这个可以用dispatch*group, dispatch*group*async 和 dispatch*group_notify来实现：

```
dispatch_group_t group = dispatch_group_create();
 dispatch_group_async(group, dispatch_get_global_queue(0,0), ^{
      // 并行执行的线程一
 });
 dispatch_group_async(group, dispatch_get_global_queue(0,0), ^{
      // 并行执行的线程二
 });
 dispatch_group_notify(group, dispatch_get_global_queue(0,0), ^{
      // 汇总结果
 });
 // 释放掉
dispatch_release(group);
```

dispatch*notify()提供了一个知道group什么时候结束的点. 当然也可以使用dispatch*wait()去阻塞

信号量

`信号量` 和 `琐` 的作用差不多,可以用来实现同步的方式. 但是信号量通常用在`允许几个线程同时访问一个资源,通过信号量来控制访问的线程个数.`

```
// 信号量初始化为1
 dispatch_semaphore_t semaphore = dispatch_semaphore_create(1);

 dispatch_queue_t queue = dispatch_get_global_queue(0, 0);

NSDate *startTime = [NSDate date];

__block UIImage *image = nil;

//1.先去网上下载图片
dispatch_async(queue, ^{

    // wait操作-1
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    // 开始下载
    // signal操作+1
    dispatch_semaphore_signal(semaphore);
});

// 2.等下载好了再在刷新主线程
dispatch_async(dispatch_get_main_queue(), ^{

    // wait操作-1
    dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
    //显示图片
    // signal操作+1
    dispatch_semaphore_signal(semaphore);
});
```

dispatch*wait会阻塞线程并且检测信号量的值,直到信号量值大于0才会开始往下执行,同时对信号量执行-1操作. dispatch*signal则是+1操作.

后台运行

GCD的另一个用处是可以让程序在后台较长久的运行。在没有使用GCD时，当app被按home键退出后，app仅有最多5秒钟的时候做一些保存或清理资源的工作。但是
在`使用GCD后，app最多有10分钟的时间在后台长久运行`。这个时间可以用来做清理本地缓存，发送统计数据等工作。

```
// AppDelegate.h文件
@property (assign, nonatomic) UIBackgroundTaskIdentifier backgroundUpdateTask;

// AppDelegate.m文件
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [self beingBackgroundUpdateTask];
    // 在这里加上你需要长久运行的代码
    [self endBackgroundUpdateTask];
}

- (void)beingBackgroundUpdateTask
{
    self.backgroundUpdateTask = [[UIApplication sharedApplication] beginBackground
TaskWithExpirationHandler:^{
        [self endBackgroundUpdateTask];
    }];
}

- (void)endBackgroundUpdateTask
{
    [[UIApplication sharedApplication] endBackgroundTask: self.backgroundUpdateTas
k];
    self.backgroundUpdateTask = UIBackgroundTaskInvalid;
}
```