多线程

> *实现多线程编程的方式有非常多，本文列举三种常用方式：NSThread，NSOperation，GCD。三种方式各有利弊，本文最后会给出对比。值得注意的是虽然是三种不同的方式，但是同样操作的是线程，管理的是线程，所以根本来讲，三者有不同，但三者可以混合使用（笔者不推荐，有可能会造成不必要的冲突）。*

章节三 多线程工具之NSThread

NSThread简介

NSThread是对pthread的上层封装，把线程处理为面向对象的逻辑。一个NSThread即代表一个线程。

线程开销

线程是需要**内存和性能开销的**，内存开销包括**系统内核内存**和**应用程序内存**。用来管理和协调线程的内核结构存储在内核。线程的栈空间和每个线程的数据存储在程序的内存空间。占用内存的这些结构大部分是在线程创建的时候生成和初始化的。**因为要和内核交互，所以这个过程是非常耗时的**。线程创建大概的开销如下（其中第二线程的栈空间是可以配置的）：

- 内核数据结构：大约1KB
- 占空间：主线程大约1MB，第二线程大约512KB
- 线程创建时间：大约90毫秒
- 另外一个开销就是程序内线程同步的开销。

创建线程

使用NSThread创建线程一共有4中方法：

- 使用NSThread类方法+detachNewThreadSelector:toTarget:withObject:创建线程并执行：

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

- 使用NSObject的线程扩展（NSThreadPerformAdditions）方法：

```
  [object performSelectorInBackground:@selector(myThreadMainMethod:)
withObject:nil];
```

- 创建一个NSThread对象，然后调用start方法执行：

```
  NSThread* aThread = [[NSThread alloc] initWithTarget:self
selector:@selector(myThreadMainMethod:) object:nil];
  [aThread start];//actually create a thread
```

- 创建一个NSThread子类，然后实例化调用start方法：

```
  //子类化后，需要重写main函数 作为线程的入口
  - (void)main
  {
      @autoreleasepool
      {
          //do thread task
      }
  }
```

配置线程参数

配置线程栈空间

栈空间是用来存储为线程创建的本地变量的，栈空间的大小必须在线程的创建之前设定。不能使用创建线程的第一和第二种方法。在调用NSThread的start方法之前通过setStackSize: 设定新的栈空间大小。

配置线程的本地存储

每个线程都维护一个在线程任何地方都能获取的字典。 我们可以使用NSThread的threadDictionary方法获取一个NSMutableDictionary对象，然后添加我们需要的字段和数据。

设置线程的Detached、Joinable状态

- 脱离线程（Detach Thread）－－－线程完成后，系统自动释放它所占用的内存空间
- 可连接线程（Joinable Thread）－－－－线程完成后，不回收可连接线程的资源

  在应用程序退出时,脱离线程可以立即被中断,而可连接线程则不可以。每个可连接 线程必须在进程被允许可以退出的时候被连接。所以当线程处于周期性工作而不允许被中断的时 候,比如保存数据到硬盘,可连接线程是最佳选择。

  通过NSThread创建的线程都是Detached的。如果你想要创建可连接线程,唯一的办法是使用 POSIX 线程。POSIX 默认创建的 线程是可连接的。通过pthread_attr_setdetachstate函数设置是否脱离属性

设置线程的优先级

每一个新的线程都有一个默认的优先级。系统的内核调度算法根据线程的优先级来决定线程的

执行顺序。通常情况下我们不要改变线程的优先级，提高一些线程的优先级可能会导致低优先级的线程一直得不到执行，如果在我们的应用内存在高优先级线程和低优先级线程的交互的话，因为低优先级的线程得不到执行可能阻塞其他线程的执行。这样会对应用造成性能瓶颈。可以通过NSThread的setThreadPriority:方法设置线程优先级，优先级为0.0到1.0的double类型，1.0为最高优先级。

完善线程入口

Autorelease Pool

在线程的入口处我们需要创建一个Autorelease Pool，当线程退出的时候释放这个Autorelease Pool。这样在线程中创建的autorelease对象就可以在线程结束的时候释放，避免过多的延迟释放造成程序占用过多的内存。如果是一个长寿命的线程的话，应该创建更多的Autorelease Pool来达到这个目的。例如线程中用到了run loop的时候，每一次的迭代都需要创建Autorelease Pool。

```
- (void)myThreadMainRoutine
{
    @autoreleasepool
    {
        //do thread task
    }
}
```

设置Run Loop

当创建线程的时候我们有两种选择，一种是线程执行一个很长的任务然后再任务结束的时候退出。另外一种是线程可以进入一个循环，然后处理动态到达的任务。每一个线程默认都有一个NSRunloop，主线程是默认开启的，其他线程要手动开启。

终止线程

终止线程最好不要用POSIX接口直接杀死线程，这种粗暴的方法会导致系统无法回收线程使用的资源，造成内存泄露，还有可能对程序的运行造成影响。终止线程最好的方式是能够让线程接收取消和退出消息，这样线程在受到消息的时候就有机会清理已持有的资源，避免内存泄露。这种方案的一种实现方式是使用NSRunloop的input source来接收消息，每一次的NSRunloop循环都检查退出条件是否为YES，如果为YES退出循环回收资源，如果为NO，则进入下一次NSRunloop循环。

- 非子类化代码:

```
  - (void)threadRoutine
  {
      NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    BOOL moreWorkToDo = YES;
      BOOL exitNow = NO;
      NSRunLoop* runLoop = [NSRunLoop currentRunLoop];
      NSMutableDictionary* threadDict = [[NSThread currentThread]
threadDictionary];
      [threadDict setValue:[NSNumber numberWithBool:exitNow]
forKey:@"ThreadShouldExitNow"];
    //添加事件源
      [self myInstallCustomInputSource];
      while (moreWorkToDo && !exitNow)
      {
          //执行线程真正的工作方法，如果完成了可以设置moreWorkToDo为False
          [runLoop runUntilDate:[NSDate date]];
          exitNow = [[threadDict valueForKey:@"ThreadShouldExitNow"]
boolValue];
      }
      [pool release];
  }
```

如果需要在子线程运行的时候让子线程结束操作，子线程每次Run Loop迭代中检查相应的标志位来判断是否还需要继续执行，可以使用threadDictionary以及设置Input Source的方式来通知这个子线程。那么什么是Run Loop呢？这是涉及NSThread及线程相关的编程时无法回避的一个问题。

- 子类化代码:

```
- (void)main
{
    @autoreleasepool {
        NSLog(@"starting thread.......");
        NSTimer *timer = [NSTimer timerWithTimeInterval:2
target:self selector:@selector(doTimerTask) userInfo:nil repeats:YES];
        [[NSRunLoop currentRunLoop] addTimer:timer
forMode:NSDefaultRunLoopMode];
        [timer release];
        while (! self.isCancelled) {
            [self doOtherTask];
            BOOL ret = [[NSRunLoop currentRunLoop]
runMode:NSDefaultRunLoopMode beforeDate:[NSDate distantFuture]];
            NSLog(@"after runloop counting.........: %d", ret);
        }
        NSLog(@"finishing thread.........");
    }
}
- (void)doTimerTask
{
    NSLog(@"do timer task");
}
- (void)doOtherTask
{
    NSLog(@"do other task");
}
```

我们看到入口方法里创建了一个NSTimer，并且以NSDefaultRunLoopMode模式加入到当前子线程的NSRunLoop中。进入循环后肯定会执行-doOtherTask方式法一次，然后再以NSDefaultRunLoopMode模式运行NSRunLoop，如果一次Timer事件触发处理后，这个Run Loop会返回吗？答案是不会，Why？ NSRunLoop的底层是由CFRunLoopRef实现的，你可以想象成一个循环或者类似Linux下select或者epoll，当没有事件触发时，你调用的Run Loop运行方法不会立刻返回，它会持续监听其他事件源，如果需要Run Loop会让子线程进入sleep等待状态而不是空转，只有当Timer Source或者Input Source事件发生时，子线程才会被唤醒，然后处理触发的事件，然而由于Timer source比较特殊，Timer Source事件发生处理后，Run Loop运行方法- (BOOL)runMode:(NSString *)mode beforeDate:(NSDate *)limitDate;也不会返回；而其他非Timer事件的触发处理会让这个Run Loop退出并返回YES。当Run Loop运行在一个特定模式时，如果该模式下没有事件源，运行Run Loop会立刻返回NO。

RunLoop补充

NSRunLoop的运行接口：

```
//运行 NSRunLoop，运行模式为默认的NSDefaultRunLoopMode模式，没有超时限制
- (void)run;
//运行 NSRunLoop: 参数为运行模式、时间期限，返回值为YES表示是处理事件后返回的，NO表示是超时或者停止运行导致返回的
- (BOOL)runMode:(NSString *)mode beforeDate:(NSDate *)limitDate;
//运行 NSRunLoop: 参数为运时间期限，运行模式为默认的NSDefaultRunLoopMode模式
-(void)runUntilDate:(NSDate *)limitDate;
```

详细讲解下NSRunLoop的三个运行接口：

- -(void)run; 无条件运行

  不建议使用，因为这个接口会导致Run Loop永久性的运行在NSDefaultRunLoopMode模式，即使使用CFRunLoopStop(runloopRef);也无法停止Run Loop的运行，那么这个子线程就无法停止，只能永久运行下去。

- -(void)runUntilDate:(NSDate *)limitDate;

  有一个超时时间限制 比上面的接口好点，有个超时时间，可以控制每次Run Loop的运行时间，也是运行在NSDefaultRunLoopMode模式。这个方法运行Run Loop一段时间会退出给你检查运行条件的机会，如果需要可以再次运行Run Loop。注意CFRunLoopStop(runloopRef);也无法停止Run Loop的运行，因此最好自己设置一个合理的Run Loop运行时间。示例：

  ```
  while (!Done)
  {
      [[NSRunLoop currentRunLoop] runUntilDate:[NSDate
                  dateWithTimeIntervalSinceNow:10]];
      NSLog(@"exiting runloop.........:");
  }
  ```

- -(BOOL)runMode:(NSString *)mode beforeDate:(NSDate *)limitDate;

  有一个超时时间限制，而且设置运行模式 这个接口在非Timer事件触发、显式的用CFRunLoopStop停止Run Loop、到达limitDate后会退出返回。如果仅是Timer事件触发并不会让Run Loop退出返回；如果是PerfromSelector***事件或者其他Input Source事件触发处理后，Run Loop会退出返回YES。示例：

  ```
  while (!Done)
  {
      BOOL ret = [[NSRunLoop currentRunLoop]
runMode:NSDefaultRunLoopMode
                                          beforeDate:[NSDate
distantFuture]];
      NSLog(@"exiting runloop.........: %d", ret);
  }
  ```

线程同步

在多个线程访问相同的数据时，有可能会造成数据的冲突。比如常见的售票问题。

```
-(void)saleTicket{

    int current = _ticket;

    if (current == 0) {

        NSLog(@"sold %d",_sold);

        return;
    }
    usleep(100000);
    _ticket = current-1;

    NSLog(@"ticket: %d",_ticket);

    _sold++;


    [self saleTicket];

}
```

数据同步锁

通过使用加锁的方式解决上述问题是最常见的方式。Foundation框架中提供了NSLock对象来实现锁。

```
-(void)saleTicket{

//    加锁
    [lock lock];

    int current = _ticket;

    if (current == 0) {

        NSLog(@"sold %d",_sold);

//       解锁
        [lock unlock];
        return;
    }
    usleep(100000);
    _ticket = current-1;

    NSLog(@"ticket: %d",_ticket);

    _sold++;

    [lock unlock];

    [self saleTicket];

}
```

同步等待

多线程中经常遇到一种问题，A线程需要等待B线程执行后的某个结果继续执行，也就是同步问题，这时就会需要A等待B，解决方式如下：

```
-(void)cook{

    [lock2 lock];
    NSLog(@"cook begin");
    [lock2 wait];


    NSLog(@"go on");
}


-(void)buyStuff{
        NSLog(@"buy begin");

    usleep(20000);

    NSLog(@"back");

    [lock2 signal];
}
```

nonatomic和atomic

atomic：默认是有该属性的，这个属性是为了保证程序在多线程情况下，编译器会自动生成一些互斥加锁代码，避免该变量的读写不同步问题。 nonatomic：如果该对象无需考虑多线程的情况，请加入这个属性，这样会让编译器少生成一些互斥加锁代码，可以提高效率。

atomic的意思就是setter/getter这个函数，是一个原语操作。如果有多个线程同时调用setter的话，不会出现某一个线程执行完setter全部语句之前，另一个线程开始执行setter情况，相当于函数头尾加了锁一样，可以保证数据的完整性。nonatomic不保证setter/getter的原语行，所以你可能会取到不完整的东西。因此，在多线程的环境下原子操作是非常必要的，否则有可能会引起错误的结果。

比如setter函数里面改变两个成员变量，如果你用nonatomic的话，getter可能会取到只更改了其中一个变量时候的状态，这样取到的东西会有问题，就是不完整的。当然如果不需要多线程支持的话，用nonatomic就够了，因为不涉及到线程锁的操作，所以它执行率相对快些。

小结：NSThread 作为多线程编程的重要的工具类，我们应该尝试使用并理解其中的方式。但是NSThread的缺点很明显，我们需要手动实现非常复杂的管理线程逻辑。那么更简单的方式有哪些呢？请看下节。