多线程-概念篇

- 前言
  - 学习多线程之前先理解相关名词和涉及的概念
  - 通过实例了解多线程的实际运用,从而进一步深入理解
- 线程
  - 简单的比较
    - 一个应用(App)至少有两个线程(主线程),每个线程内部包含关系(至少一条)的task(线程),有时会入线程(辅线程),商家还要给常或动及实装(长程事件),到期时唤醒加(线程唤醒),一些特殊的店铺可能要建好子商城(子线程)直到商城倒闭或者歇业(长程结束)
  - 进程
    - 如何产生
      - 打开app和直接重产生,系统就会开启两个进程
      - 一个进程对应在系统中正在运行的一个应用程序
    - 独立的受保护的空间
      - 进程之间是独立的,每个进程均运行在其他其他受保护的内存空间内,互不干扰
    - 进程执行软件功能就需要创建一条线程去处理
  - 线程
    - 1个进程要想执行任务,必须得有线程(每个进程至少要有1条线程)
    - 线程是进程的基本执行单元,一个进程(程序)的所有任务都在线程中执行
    - 线程的执行
      - 一个人三个任务,
      - 单条线程同时只能执行一个任务,完成任务才能执行下一个,是顺序的
      - 三个工人三个任务,
    - 线程的并行
      - 一个进程中可以开启多条线程,每条线程可以并行(同时)执行不同的任务
    - 什么是多线程?就名思义,多线程就是多条单线程
      - 调度(dispatch)
        - 同一时间,一个CPU核心只能处理1条线程,不同条线程在工作(执行)
        - 一台设备CPU核心数是有限,线程数超过核心数,想要流畅运行,就会需要去调度线程,来回切换
      - 多线程如何工作?
        - 多线程并发(同时)执行,其实是CPU快速地在多条线程之间调度(切换)
        - 如果CPU调度线程的时间足够快,就造成了多线程并发执行的假象
      - 线程多好处?
        - CPU会在大量线程之间来回调度,CPU会累死,消耗大量的CPU资源
        - 每条线程被调度执行及完成的频次会降低(线程的执行效率降低)
      - 优点
        - 提高程序的执行效率
        - 提高资源利用率(CPU、内存利用率)
      - 缺点
        - 线程数不宜过多:开启线程需要占用一定的内存空间(默认情况下,主线程占用1M,子线程占用512KB),如果开启太量的线程,会占用大量的内存空间,降低程序的性能
        - 线程越多,线程潜在难度加大
      - iOS中主线程
        - 一个iOS程序运行后,默认会开启1条主线程
        - iOS所有的处理UI事件(比如显示UI刷新UI界面、点击事件、滚动事件、拖拽事件等)都只能在主线程上操作,所以耗时操作会导致页面卡顿
        - 子线程会阻塞不安全
      - 同时创建线程
        - 当一个任务单条线程执行很慢,影响体验时,你需要去开辟一条新的线程
        - 分担主线程的压力,其时的操作放到子线程中去处理
    - 线程的生命周期
      - 新建状态:在内存中,但不在可调度池
      - 可调度状态:在可调度池中,可以执行任务
        - sleep/同步锁
      - 阻塞状态:被移出可调度池,在内存中,不能执行任务
        - sleep到期/同步锁-解锁
      - 死亡状态:被释放
- 线程的管理
  - 队列(queue)?
    - 做任务的地方,是线程的容器,用来管理任务,线程的创建和回收不需要程序员操作,由队列(queue)负责,可调度的线程时间就在线程池里
    - 种类
      - 串行队列
        - 创建队列中的任务只会顺序执行
        - dispatch_queue_t q = dispatch_queue_create("....", dispatch_queue_serial);
      - 并行队列
        - 创建队列中的任务通常会并发执行
        - dispatch_queue_t q = dispatch_queue_create("....", dispatch_queue_concurrent);
      - 全局队列
        - 系统默认以已经创建的,只能拿过来(get)使用,不能创建
        - dispatch_queue_t q = dispatch_get_global_queue(dispatch_queue_priority_default, 0);
      - 主队列
        - 每一个应用程序对应唯一一个主队列,用来操作主线程的,直接get即可
        - dispatch_queue_t q = dispatch_get_main_queue();
    - 单词翻译
      - dispatch 调度
      - queue 队列
      - serial 串行(顺序的)
      - concurrent 并行(同时的)
      - global 全局的
  - 操作?
    - 同步操作
      - dispatch_sync
        - 会依次顺序执行,能够决定任务的执行顺序
        - 先阻塞当前线程,直到队列中的任务被分派到某个线程被执行完毕,再解除当前线程主线程
        - 不能被阻塞,否则会造成死锁
    - 异步操作
      - dispatch_async
        - 会并发执行,无法确定任务的执行顺序
        - 不阻塞当前线程,任务未完成之前也会直接执行当前之后的任务
  - 队列 + 操作 的组合
    - 概念
      - 串行队列 具有管理一条线程的能力
      - 并行队列 具有管理多条线程的能力
      - 全局队列 具有管理多条线程的能力
      - 主队列 具有管理一条主线程的能力
      - 同步操作 具有创建一条线程的能力
      - 异步操作 具有创建多条线程的能力
    - 组合
      - 串行或 并行 或 全局 队列 + 同步 不会新建线程,顺序执行
      - 串行队列异步 新建线程,顺序执行
      - 并行队列 (或 全局队列) 异步 新建多个线程,操作无序执行
      - 主队列异步(X) 都应该在主线程上顺序执行的,不存在异步的
      - 主队列同步(X) 发生会死锁,永远不会被执行
        - - (void)viewDidLoad {
          - [super viewDidLoad];
          - //存放这个主线程,而这个block就永远没有机会执行了,所以就会造成死锁
          - dispatch_async(dispatch_get_main_queue(0, 0), ^{
          - });
        - }
- 并发与并行
  - 并发
    - 同时操作只是假象
    - 单核设备同一时刻只能执行一项任务,想要实现一样的效果,首先必须运行一个线程,执行上下文切换,然后运行另外一个线程或进程,称之为并发
  - 并行
    - 同时操作多条
    - 多核设备会开辟多个线程同时执行代码的不同部分,称之为并行
- 线程常见
  - NSThread
    - 使用nsthread对象建立一个线程非常方便
    - 管理多个线程非常困难,不推荐使用
  - GCD
    - grand central dispatch,庞大的线程中枢调度技术
    - 基于c语言
    - 用block定义任务,使用起来非常灵活便捷
  - NSOperation/NSOperationQueue
    - 封装GCD实现的一套Objective-C的api
    - 是化操作之间的依赖关系使用
- dispatch_once
  - 多线程中被设置静态变量onceToken标记的代码只能执行一次
  - dispatch_once(&onceToken, ^{});
- 线程优先级
- 多线程安全问题
  - 当多线程中同时存在某个对象的读写操作时,那么它是线程不安全的
  - 对于不可变对象,或者修改时进行同步锁的操作,是线程安全的
- GCD-栅栏函数
  - dispatch_barrier_async 同步栅栏函数和dispatch_barrier_sync 异步栅栏函数
  - 使用前需要理解几个概念
    - 开始执行某行代码A 处理和理解代码本身,立马完成
    - 开始执行某个任务B 去处理代码交代要干的活,不能立即完成,需要一定的时间
  - 代码示例
    - 伪代码(从上往下)
      - NSLog(@"代码已执行到此处1");
      - 同步任务1
      - 异步任务2
      - NSLog(@"代码已执行到此处2");
      - dispatch_barrier_sync(concurrentQueue, ^{
        - 异步任务3
        - 异步任务4
      - });
      - NSLog(@"代码已执行到此处3");
      - 异步任务5
      - NSLog(@"代码已执行到此处4");
      - dispatch_barrier_async(concurrentQueue, ^{
        - 异步任务6
        - 异步任务7
      - });
      - NSLog(@"代码已执行到此处5");
      - 异步任务8
      - NSLog(@"代码已执行到此处6");
      - 结果
        - 代码已执行到此处1
        - 同步任务1 – 完成
        - 代码已执行到此处2
        - 异步任务2 – 完成
        - ----完全隔离----
        - 异步任务3 – 完成
        - 异步任务4 – 完成
        - ----完全隔离----
        - 代码已执行到此处3
        - 代码已执行到此处4
        - 代码已执行到此处5
        - 代码已执行到此处6
        - 异步任务5 – 完成
        - ----异步任务隔离----
        - 异步任务6 – 完成
        - 异步任务7 – 完成
        - ----异步任务隔离----
        - 异步任务8 – 完成
    - 栅栏函数执行解析
      - 共同点
        - 都会等待在它前面所有的任务先执行完才开始
      - 不同点
        - 同步栅栏:
          - 开始执行时会堵塞线程
          - 后面的同步任务也要等待
        - 异步栅栏:
          - 后面的异步任务等待,但是同步任务不受影响
    - 实际案例
      - 四个异步任务,(A、B)-> C -> D ,要求是A、B完成后,执行 C ,然后C完成后执行D
      - 使用异步栅栏
        - 异步任务A
        - 异步任务B
        - dispatch_barrier_async(concurrentQueue, ^{
          - 异步任务C
        - });
        - 异步任务D
- Dispatch Groups 队列组
  - 异步任务ABC、等AB完成后执行C
    - 代码
      - dispatch_group_t downloadGroup = dispatch_group_create();
      - dispatch_group_enter(downloadGroup);
        - 任务A , B
      - dispatch_group_leave(downloadGroup);
      - });
      - //当组中的所有任务完成时 会通知主队列执行这个闭包
      - dispatch_group_notify(downloadGroup, dispatch_get_main_queue(), ^{
        - 任务C
      - });
  - GCD-wait 阻塞等待
    - dispatch_group_wait(downloadGroup, DISPATCH_TIME_FOREVER);
    - 添加该行代码会阻塞线程,后面任务需等待
  - 通知功能:AB完成后发送通知开始处理C
  - 对比栅栏函数区别:不需要等待前面的任务执行完成
- 单例 dispatch_once
  - @implementation Tool
  - static id _instance;
  - + (instancetype)sharedTool {
    - static dispatch_once_t onceToken;
    - dispatch_once(&onceToken, ^{
      - _instance = [[Tool alloc] init];
    - });
    - return _instance;
  - }
  - @end
  - 其它alloc等方法也要重写
- NSOperation NSOperationQueue
  - 相关概念
    - OC语法,基于 GCD 封装
    - 可以设置任务执行的优先级
    - 可以随方便的取消一个任务的执行
    - 监听(KVO)执行任务的状态
    - 可以设置最大并发数
- 回到主线程/主队列上执行
  - GCD
    - 异步(常用)
      - 异步:
      - dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        - // 需要在主线程执行的代码
      - });
    - 同步
      - 同步:(先堵塞当前子线程任务)
      - dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        - // 需要在主线程执行的代码
      - });
  - NSOperationQueue
    - 异步:
    - [[NSOperationQueue mainQueue] addOperationWithBlock:^{
      - //需要在主线程执行的代码
    - }];
  - NSThread
    - 代码
      - waitUntilDone:NO等价于GCD方式的异步回到主线程,YES等价于GCD方式中的同步回到主线程
      - [self performSelectorOnMainThread:@selector(WantToGoBackMainThread:) withObject:@"" waitUntilDone:YES];
      - - (void)WantToGoBackMainThread(id)object{
        - //需要在主线程执行的代码
        - NSLog(@"object:%@",object);
      - }