Java并发编程

- 锁
  - Synchronized与Lock
    - Synchronized
      - 是Java中的关键字　　在JVM层面实现了对临界资源的同步互斥访问，通过对对象的头文件来操作，从而达到加锁和释放锁的目的
      - synchronized 代码块是由一对儿 monitorenter/monitorexit 指令实现的,Monitor 对象是同步的基本实现单元。可以理解为在指令中插入了监视器，用于监视对象
      - wait()和notify()必须在synchronized代码块中调用。
        - wait()
          - 释放占有的对象锁，线程进入等待池，释放cpu,而其他正在等待的线程即可抢占此锁，获得锁的线程即可运行程序。
          - 与Sleep的区别　　wait()和sleep()最大的不同在于wait()会释放对象锁，而sleep()不会
        - notify()
          - 对对象锁的唤醒，从而使得wait()的线程可以有机会获取对象锁
          - 调用notify()后，并不会立即释放锁，而是继续执行当前代码，直到synchronized中的代码全部执行完毕，才会释放对象锁
        - notifyAll()　　唤醒所有等待的线程
      - 优缺点
        - 优点
          - 可以只锁需要锁的地方
          - 可以方便的使用wait()和netify()方法
        - 缺点
          - 不能响应中断
          - 同一时刻不管是读还是写都只能有一个线程对共享资源操作，其他线程只能等待
          - 锁的释放由虚拟机来完成，不用人工干预，不过此即使缺点也是优点，优点是不用担心会造成死锁，缺点是由可能获取到锁的线程阻塞之后其他线程会一直等待，性能不高。
      - 释放锁的条件　　执行完成或抛出异常
      - 锁的升级、降级　　JVM 优化 synchronized 运行的机制,当 JVM 检测到不同的竞争状况时,会自动切换到适合的锁实现,这种切换就是锁的升级、降级。
      - 偏斜锁、自旋锁、轻量级锁、重量级锁
        - 偏斜锁
          - 当没有竞争出现时,默认会使用偏斜锁
          - 这样做的假设是基于在很多应用场景中,大部分对象生命周期中最多会被一个线程锁定,使用偏斜锁可以降低无竞争开销。
        - 轻量级锁　　如果有另外的线程试图锁定某个已经被偏斜过的对象,JVM 就需要撤销(revoke)偏斜锁,并切换到轻量级锁实现
        - 重量级锁　　轻量级锁依赖 CAS 操作 Mark Word 来试图获取锁,如果重试成功,就使用普通的轻量级锁;否则,进一步升级为重量级锁。
    - Lock
      - 基于jdk层面实现的接口
      - ReentrantLock
      - ReentrentReadWriteLock（读写锁）
        - 出现的原因：虽然 ReentrantLock 和 synchronized 简单实用,但是行为上有一定局限性,通俗点说就是“太霸道”,要么不占,要么独占。实际应用场景中,有的时候不需要大量竞争的写操作,而是以并发读取为主
        - Java 并发包提供的读写锁等扩展了锁的能力,它所基于的原理是多个读操作是不需要互斥的,因为读操作并不会更改数据,所以不存在互相干扰。而写操作则会导致并发一致性的问题,所以写线程之间、读写线程之间,需要精心设计的互斥逻辑。
        - 缺点：读写锁看起来比 synchronized 的粒度似乎细一些,但在实际应用中,其表现也并不尽如人意,主要还是因为相对比较大的开销
      - StampedLock　　JDK 在后期引入了 StampedLock,在提供类似读写锁的同时,还支持优化读模式,大多数读操作并不需要等待写完成,它能够提供更加出色的性能
      - Lock的优缺点
        - 优点
        - 缺点
    - Synchronized与Lock的区别
      - synchronized在执行完成或者抛出异常时，虚拟机会自动释放锁；而Lock在抛出异常时，必须主动调用unLock()方法，否则不会释放锁
      - Lock要相对更灵活，有很多方法来尝试获取锁，包括
        - 立即返回是否成功的tryLock()
        - 一直尝试获取的lock()方法题
        - 尝试等待指定时间长度获取的方法
      - 通过在读多，写少的高并发情况下，我们用ReentrantReadWriteLock分别获取读锁和写锁来提高系统的性能，因为读锁是共享锁，即可以同时有多个线程读取共享资源，而写锁则保证了对共享资源的修改只能是单线程的。
    - 参考　　线程的同步控制synchronized和lock的对比和区别
- 线程
  - 什么是线程　　线程是系统调度的最小单元,一个进程可以包含多个线程,作为任务的真正运作者,有自己的栈(Stack)、寄存器(Register)、本地存储(Thread Local)等,但是会和进程内其他线程共享文件描述符、虚拟地址空间等。
  - 线程的生命周期和状态转移
    - 新建(NEW)　　线程被创建出来还没真正启动的状态,可以认为它是个 Java 内部状态
    - 就绪(RUNNABLE)　　表示该线程已经在 JVM 中执行,当然由于执行需要计算资源,它可能是正在运行,也可能还在等待系统分配给它 CPU片段,在就绪队列里面排队
    - 阻塞(BLOCKED)　　阻塞表示线程在等待 Monitor lock。比如,线程试图通过 synchronized去获取某个锁,但是其他线程已经独占了,那么当前线程就会处于阻塞状态。
    - 等待(WAITING)　　表示正在等待其他线程采取某些操作。一个常见的场景是类似生产者消费者模式,发现任务条件尚未满足,就让当前消费者线程等待(wait),另外的生产者线程去准备任务数据,然后通过类似 notify 等动作,通知消费线程可以继续工作了。Thread.join() 也会令线程进入等待状态
    - 计时等待(TIMED_WAIT)　　其进入条件和等待状态类似,但是调用的是存在超时条件的方法,比如 wait 或 join 等方法的指定超时版本
    - 终止(TERMINATED)　　不管是意外退出还是正常执行结束,线程已经完成使命,终止运行,也有人把这个状态叫作死亡
  - 一个线程两次调用start()方法会出现什么情况　　在第二次调用 start() 方法的时候,线程可能处于终止或者其他(非 NEW)状态,但是不论如何,都是不可以再次启动的。
  - 线程与进程的关系和区别
  - 主线程与子线程　　主线程中调用子线程，如何等待子线程运行完，再接着执行主线程
    - Sleep
    - Join
    - CountDownLatch
    - CycliclBarrier
  - ThreadLocal　　Java 提供的一种保存线程私有信息的机制,因为其在整个线程生命周期内有效,所以可以方便地在一个线程关联的不同业务模块之间传递信息,比如事务 ID、Cookie 等上下文相关信息
  - 死锁
    - 什么是死锁　　死锁是一种特定的程序状态,在实体之间,由于循环依赖导致彼此一直处于等待之中,没有任何个体可以继续前进。
    - 什么情况下会产生死锁　　通常来说,我们大多是聚焦在多线程场景中的死锁,指两个或多个线程之间,由于互相持有对方需要的锁,而永久处于阻塞的状态。
    - 如何定位、修复
      - 定位死锁最常见的方式就是利用 jstack 等工具获取线程栈,然后定位互相之间的依赖关系,进而找到死锁。
      - 如果是比较明显的死锁,往往 jstack 等就能直接定位,类似 JConsole 甚至可以在图形界面进行有限的死锁检测。
    - 如何避免死锁
      - 尽量避免使用多个锁,并且只有需要时才持有锁。
      - 如果必须使用多个锁,尽量设计好锁的获取顺序
        - 可以参看著名的银行家算法
        - 将对象(方法)和锁之间的关系,用图形化的方式表示分别抽取出来
        - 然后根据对象之间组合、调用的关系对比和组合,考虑可能调用时序。
        - 按照可能时序合并,发现可能死锁的场景。
        - 使用带超时的方法,为程序带来更多可控性。
        - 通过静态代码分析(如 FindBugs)去查找固定的模式,进而定位可能的死锁或者竞争情况。
- Java并发包
  - 并发工具类
    - 提供了比 synchronized 更加高级的各种同步结构
      - CountDownLatch　　允许一个或多个线程等待某些操作完成
      - CyclicBarrier　　一种辅助性的同步结构,允许多个线程等待到达某个屏障。
      - Sempahore　　Java 版本的信号量实现
    - 线程安全的容器
      - List　　类似快照机制,实现线程安全的动态数组 CopyOnWriteArrayList
      - Set　　CopyOnWriteArraySet
        - CopyOnWrite的意思就是在对容器进行修改时，会复制一个新的数组来替代原来的
        - 这种数据结构,相对比较适合读多写少的操作,不然修改的开销还是非常明显的。
      - Map
        - ConcurrentHashMap　　如果我们的应用侧重于 Map放入或者获取的速度,而不在乎顺序,大多推荐使用 ConcurrentHashMap
        - 有序的ConcunrrentSkipListMap　　如果我们需要对大量数据进行非常频繁地修改,ConcurrentSkipListMap 也可能表现出优势
      - Concurrent与CopyOnWrite的区别
        - Concurrent 类型没有类似 CopyOnWrite 之类容器相对较重的修改开销
        - Concurrent 往往提供了较低的遍历一致性。你可以这样理解所谓的弱一致性
          - 当利用迭代器遍历时,如果容器发生修改,迭代器仍然可以继续进行遍历
          - 与弱一致性对应的,就是我介绍过的同步容器常见的行为“fast-fail”,也就是检测到容器在遍历过程中发生了修改,则抛出 ConcurrentModificationException,不再继续遍历。
          - 弱一致性的另外一个体现是,size 等操作准确性是有限的,未必是 100% 准确
          - 与此同时,读取的性能具有一定的不确定性。
    - 并发队列
      - Deque　　Deque 的侧重点是支持对队列头尾都进行插入和删除
        - 尾部插入时需要的 addLast(e)、offerLast(e)
        - 尾部删除所需要的 removeLast()、pollLast()
      - Queue
        - Blocking
          - Blocking 意味着其提供了特定的等待性操作,获取时(take)等待元素进队,或者插入时(put)等待队列出现空位
          - BlockingQueue 基本都是基于锁实现
          - 各种 BlockedQueue 实现
            - ArrayBlockingQueue　　最典型的的有界队列
            - SynchorousQueue
              - 每个删除操作都要等待插入操作
              - 每个插入操作也都要等待删除动作
              - 其内部容量是 0　　？？？
            - PriorityBlockingQueue　　无边界的优先队列　　虽然严格意义上来讲，其大小总归是要受系统资源影响
            - LinkedBlockingQueue　　有界的，但是如果在创建队列时没有指定队列大小，就自动被设置为 Integer.MAX_VALUE,成为了无界队列
        - DelayedQueue　　无边界的队列
        - LinkedTransferQueue　　无边界的队列
      - 哪些队列是有界的,哪些是无界的?
        - 有界
        - 无界
      - ConcurrentLinkedQueue与LinkedBlockingQueue的区别　　LinkedBlockingQueue
    - 强大的 Executor 框架
      - 线程池
        - 可以创建各种不同类型的线程池,调度任务运行等,绝大部分情况下,不再需要自己从头实现线程池和任务调度器。
        - 线程池的组成
          - 1、线程池管理器（ThreadPool）
          - 2、工作线程（PoolWorker）
          - 3、任务接口（Task）
          - 4、任务队列（taskQueue）
        - 初始化参数
          - corePoolSize
          - maximumPoolSize
          - keepAliveTime和TimeUnit
          - workQueue　　工作队列,必须是 BlockingQueue
        - 常见线程池
          - newCachedThreadPool()（推荐使用）　　可缓存线程池，当线程池大小超过了处理任务所需的线程，那么就会回收部分空闲（一般是60秒无执行）的线程，当有任务来时，又智能的添加新线程来执行。
          - newFixedThreadPool(int nThreads)　　固定数量的线程池，没提交一个任务就是一个线程，直到达到线程池的最大数量，然后后面进入等待队列直到前面的任务完成才继续执行
          - newSingleThreadExecutor()　　单个线程的线程池，即线程池中每次只有一个线程工作，单线程串行执行任务
          - newScheduledThreadPool(int corePoolSize)　　大小无限制的线程池，支持定时和周期性的执行线程
          - newWorkStealingPool(int parallelism)
        - 线程池的工作步骤
        - 为什么要用线程池
          - 1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
          - 2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。
          - 3.将任务的提交和执行进行解耦合，开发的时候不需要考虑太多线程的代码，只关心业务逻辑。
        - 使用：　　ExecutorService pool=Executors.newFixedThreadPool(2);