多线程和并发

1. Future/FutureTask/CompletableFuture

2. 几个关键属性
   - synchronized
   - volatile
     - 可见性，不能保证原子性，每次读取都强制从主内存刷数据，替代方案： Atomic 原子操作类
   - final
     - 构造函数结束返回时，final 域最新的值被保证对其他线程可见

3. 线程池
   - 核心参数
     - 核心线程数(corePoolSize)
       - 业务线程数安装CPU密集型IN，IO密集型2N，多了会抢占资源 （Nlinux服务CPU核大小）
     - 最大线程数(maximumPoolSize)
     - 空闲线程的保活时间(keepAliveTime)
       - 多出的核心线程的空闲线程存活时间
     - 任务队列（workQueue）
       - BlockingQueue 接口的某个实现（常使用 ArrayBlockingQueue 和 LinkedBlockingQueue）
     - 绝句策略defaultHandler
       - 默认：丢弃任务并抛出RejectedExecutionException异常，RejectedExecutionHandler defaultHandler = new AbortPolicy()，一共四种策略
     - threadFactory 用于生成线程
   - 
     参数值设置多大核心源码
   - 使用注意（尽量自己自己设置具体参数创建线程池）
     - 四类程序：newFixedThreadPool　newSingleThreadExecutor　newCachedThreadPool　newScheduledThreadPool
     - 1：newFixedThreadPool （坑1：LinkedBlockingQueue的队列默认值数很大，会导致OOM） 2：newCachedThreadPool （坑1：线程数默认值，会导致OOM）

4. 集合和并发安全类型
   - ArrayList
     1.
     2.
     3.
   - LinkedList
   - CopyOnWriteArrayList
     1.
     2.
   - HashMap
   - LinkedHashMap
   - ConcurrentHashMap

AQS（抽象出来统一的同步协调处理器AbstractQueuedSynchronizer）

1. 模板设计模式，子类同步器实现（独占，共享四个方法tryAcquire()，tryAcquireShared()，tryRelease()，tryReleaseShared()）
2. 同步状态 state（由CAS控制原原子性）， CLH 同步 FIFO 队列双端双向链表，线程加入到同步队列后会被挂起，等待释放锁唤醒后继节点，使得继续获得同步状态
3. AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中
4. 相关实现类
   - ReentrantLock（独占）
     1. 有三个核心类:1：Sync extendsAbstractQueuedSynchronizer 2：NonfairSync extends Sync 3：FairSync extends Sync，默认的构造函数初始化非公平锁（非公平锁会有更好的性能，因为它的吞吐量比较大），实现tryAcquire()，，tryRelease() 加锁和解锁
     2. 可重入锁（ Synchronized也是）：当前线程再次获取锁时，state加1 ，解锁减1，所以lock和unlock必须一一配对
     3. Lock的实现类
   - ReentrantReadWriteLock
     - 一个读锁（共享），一个写锁（独占）。读时可以多个线程，写时操作的内存影响只能一个线程，适合读多写少的场景
   - CountDownLatch（倒计时器 共享）
     1. 它允许一个或多个线程一直等待，直到其他线程的操作执行完后再执行。例如，应用程序的主线程希望在负责启动框架服务的线程已经启动所有的框架服务之后再执行
     2. 实现：countDown()方法时，计数器count减1，当计数器count等于0时，会唤醒AQS等待队列中的线程。调用await()方法，线程会被挂起，它会等待直到count值为0才继续执行
   - Semaphore（信号量 共享）
     - 使用场景：同一时间控制并发线程数，Semaphore经常用于限制获取某种资源的线程数量
   - CyclicBarrier(循环栅栏)
     - CountDownLatch是计数器，线程完成一个记录一个，只不过计数不是递增而是递减，而CyclicBarrier更像是一个阀门，需要所有线程都到达，阀门才能打开，然后继续执行。