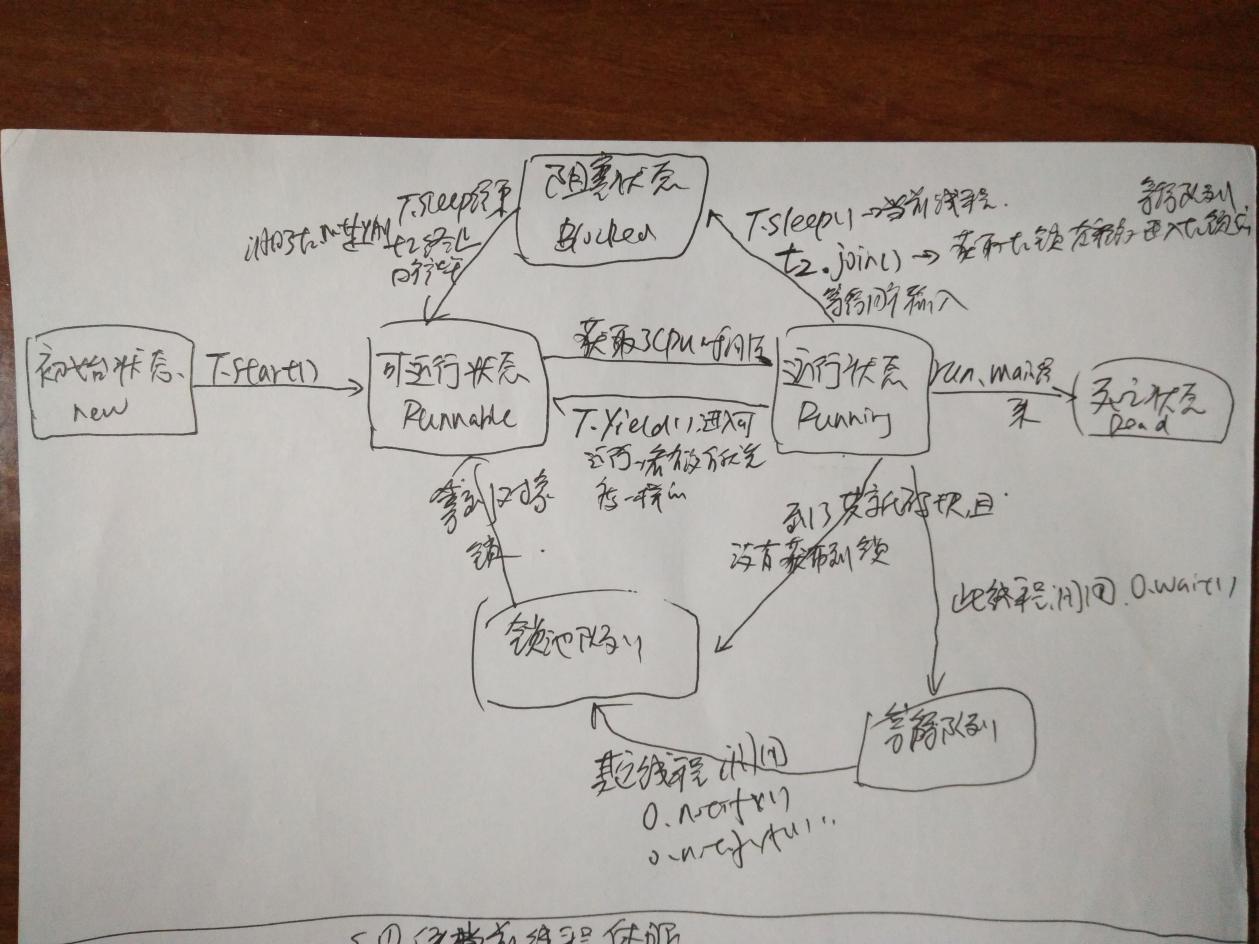
## 多线程专题

1. 线程状态



1. sleep、yield、wait、join
2. Thread.Sleep(n)
3. 使当前线程休眠
4. 不释放锁
5. 让出cpu时间给其他线程
6. 成为其他阻塞状态
7. 当时间到了，直接进入可运行状态
8. 当重新获取cpu时间，进入运行状态
9. Thread.Yield()
10. 不释放锁
11. 当前线程调用，让出cpu时间，直接成为可运行状态
12. 检测是否有相同优先级的线程处于可运行状态，如果有给它cpu时间，没有继续运行
13. Object.Wait()
14. 必须在锁代码块内，synchronized(object)
15. Object.wait()，当前线程释放object锁，进入该锁的等待队列中，为等待阻塞状态
16. 当其他线程调用Object.notifyAll()时，唤醒Object对象的等待队列中的所有线程，使它们进入锁池队列；当调用Object.notify()时，只随机唤醒等待队列中的一个线程
17. 当锁池队列中某个线程拿到锁，进入可运行状态
18. Object.join()
19. 当前线程执行T1.join()方法
20. 当前线程获取了T1对象的锁，并调用了T1.wait()方法，进入T1对象的等待队列
21. 进入等待队列，等待T1线程结束，自动调用T1.notifyAll()方法唤醒线程
22. 内部结构 while(isAlive) wait();只要T1线程不结束，一直等待!
23. Synchronized
24. 缺点
25. 假如占有锁的线程被阻塞了，但又没有释放锁，那么其他线程会一直等待下去
26. 多个读操作之间也会冲突
27. 不能判断是否获取到锁
28. 优点
29. synchronized是JVM层面实现的，会不断优化，且不用手动释放锁
30. LOCK是JAVA的接口，基于JDK层面实现的，需要手动释放锁
31. 作用域
32. 对象实例：synchronized aMethod();不同实例对象之间不干扰
33. 类范围：synchronized static aStaticMethod 所有实例都起作用
34. 在方法中的某个块中，表示只对这个区块的资源互斥访问
35. Synchronized（this），作用域是对象
36. synchronized不能继承，基类的方法在子类中是普通方法
37. 零长度的byte数组比任何对象都经济， private byte[] lock = new byte[0];
38. Runnable接口比继承Thread的优势
39. 适合多个线程处理一个资源
40. 可以避免java中的单继承限制
41. 增加程序的健壮性，代码可以被多个线程共享，代码和数据独立
42. 线程池中只能放入实现Runnable或callable类，不能放继承Thread的类
43. JAVA内存模型与线程
44. 起因

由于计算机的存储设备与处理器的运算速度有几个量级的差距，所以加入一层读写速度尽可能接近处理运算速度的高速缓存来作为内存与处理器之间的缓冲。（将运算所需的数据复制到缓存中，运算结束，再从缓存同步到内存之中）

1. 后果

缓存一致性。在多处理器系统中

1. 每个处理器都有自己的高速缓存
2. 它们又共享同一主内存
3. 当涉及到共享数据时，会发生脏数据情况
4. 内存模型

不同构建的物理机器可以拥有不一样的内存模型，而JAVA虚拟机也有自己的内存模型

1. 所有的变量（共享变量）都存储在主内存中
2. 每个线程有自己的工作内存（可以高速缓存类比）
3. 线程对变量的所有操作都必须在工作内存中进行（包括volatile）不能直接读写主内存中的变量
4. 线程间变量值的传递需要通过主内存
5. 主内存、工作内存与JAVA内存区域没有关系，但可以类比，主内存为JAVA堆，工作内存为JAVA栈
6. 另一个类比，主内存为物理硬件内存；工作内存为寄存器和高速缓存区
7. 多次lock后，只有执行相同次数的UNLOCK操作，变量才会解锁。耗时间，所有有了偏向锁和轻量级锁优化。
8. 如果对一个变量执行lock操作，那么将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量值
9. Volatile
10. 保证可见性

每次对元素进行操作，都要去主存中取异拆（但取完后可以进行别的操作，这样就不能保持一致性了），其他线程可能取到的还是脏数据

1. 禁止指令重排
2. 双重检测，单例模式用了volatile
3. Volatile变量读操作的性能消耗与普通变量几乎没区别
4. 写操作可能慢些，因为它需要在本地代码中插入许多内存屏障指令禁止指令重排
5. JAVA与线程

主流的操作系统都提供了线程的实现，JAVA提供了在不同硬件和操作系统下对线程操作的同一处理。

1. **内核线程实现**
2. 线程由内核来完成线程的切换
3. 各种线程操作（创建、同步）都需要系统调用，需要在内核态与用户态中来回切换，代价高
4. 每个轻量级进程（即线程）都需要一个内核线程支持，需要消耗一定的内核资源（内核线程栈空间），因此一个系统支持轻量级线程数量有限
5. **用户线程实现**
6. 不需要内核支持
7. 但所有的线程操作都要用户自己处理
8. 很复杂，高性能数据库
9. **用户线程+轻量级线程混合实现**

结合了1、2的优点

1. JAVA线程调度
2. 协同式

线程把自己的工作完成后，主动通知系统切换到另一个线程，没有线程同步问题

1. 抢占式

所有线程在不停的抢占cpu时间