1.线程：一行一行代码执行就是一个线程；

多条同时并行执行。

runnable是线程执行的代码的宿主(this在此处不能代表对象)

2.定时器timer

new Timer（）.schedule(TImerTask(run 方法),time(s));

3.线程互斥--汇款付款

4.线程同步(使用wait和notify)

5.线程范围内共享数据 将数据和线程结合放在Map中在各个取值时连同线程一起取值

6.ThreadLocal线程范围内的共享变量相当于一个map

7.多线程共享数据（买票）（++ ——）

8.java并发库 utils.concurrent atomic

atomicInteger : + 后别人加先完成，自获得非期望结果（成员变量，不是局部变量）

addAndGet ， （执行完后别人才能进行操作）

9.ExecutorService Executor.getsin......

生成的线程池

线程池的submit方法返回值,参数是一个线程. callable有返回值 调用call方法

future获得返回值.get(),future.get方法中可以传参数如果规定时间获取不到抛出异常.

CompletionService不一定按写的顺序获取值,

10.lock更加面向对象

读写锁，多个读锁不互斥，读锁和写锁互斥，写锁和写锁互斥

ReentrantLock是一个排它锁，同一时间只允许一个线程访问，

而ReentrantReadWriteLock允许多个读线程同时访问，但不允许写线程和读线程，写线程和写线程同时访问，相对于排它锁，提高了并发性，。在实际应用中，对共享数据如缓存，的访问都是读操作多于写操作，能够提供更好的并发性和吞吐量。

读写锁维护了两个锁，读写两个锁。所有ReadWriteLock实现必须保证写锁的内存同步效果，也要保持相关读锁的联系。也就是说，成功获取读锁的线程会看到写锁之前所做的更新。

支持功能：支持公平和非公平的获取所得方式。

支持可重入。读线程在获取了读锁还可以获取读锁，写线程在获取了写锁后既可以再次获取读锁，又可以获取写锁。

还允许从写锁降级成为读锁，实现方式：先获得写锁，然后获得读锁，最后释放写锁，但是从读锁升级到写锁是不允许的。

读锁和写锁都支持锁获取期间的中断。

condition支持。只有写锁提供了condition实现，读锁不支持condition

10.AQS

首先reentrantlock(可重入锁),抽象的队列式的同步器,AQS定义了一套多线程访问共享资源的同步器框架,许多同步类实现都依赖于他,常用的reentrantlock/semaphore/countDownLatch

维护了一个volatile int state(代表共享资源)和一个FIFO线程等待队列(多线程争用资源被阻塞是会进入).

state的三种方式,get,set,compareandsetstate

两种资源共享方式:Exclusive(独占,只用一个线程能执行,如ReentrantLock)

share(共享多个线程可同时执行,如semaphore/countdownlatch)

不同的自定义同步器争用共享资源的方式也不同.自定义同步器在实现时,只需要共享资源state的获取和释放方式即可,至于具体线程等待队列的维护(如获取资源失败/唤醒出队).

方法:

isHeldExclusively该线程是否独占资源,用到condition是才需要实现它.

tryAcquire()独占方式,尝试获取资源

tryrelease()独占方式,尝试释放资源

tryAcquireShared()尝试获得共享资源,复数表示失败,0表示成功,但没有可剩资源,正数表示成功,且有剩余资源.

tryReleaseShared(),尝试释放资源

一ReenTrantLock为例,state初始化为0,表示未锁定状态,A线程lock时,会调用tryAcquire独占该锁并将state+1,此后,其他线程获取资源时就会失败,直到A线程unlock到state为0;

释放锁之前,A线程是可以重复获取锁的(state会累加),虎丘多少次就要释放多少次,保证state回0状态.

二countdownLatch为例,任务分n个子线程执行,state初始化为n(n表示线程的个数),

这n个子线程是并行执行的,每个子线程执行完,countdown减1,直至state为0,会unpark调用主线程,然后主调用线程会从await返回,继续后续动作.

一般是独占或者共享,但是读写锁的自定义抽象队列同步器,同时实现独占和共享.

------------------------------------------------------------------------------------------------------------------------------

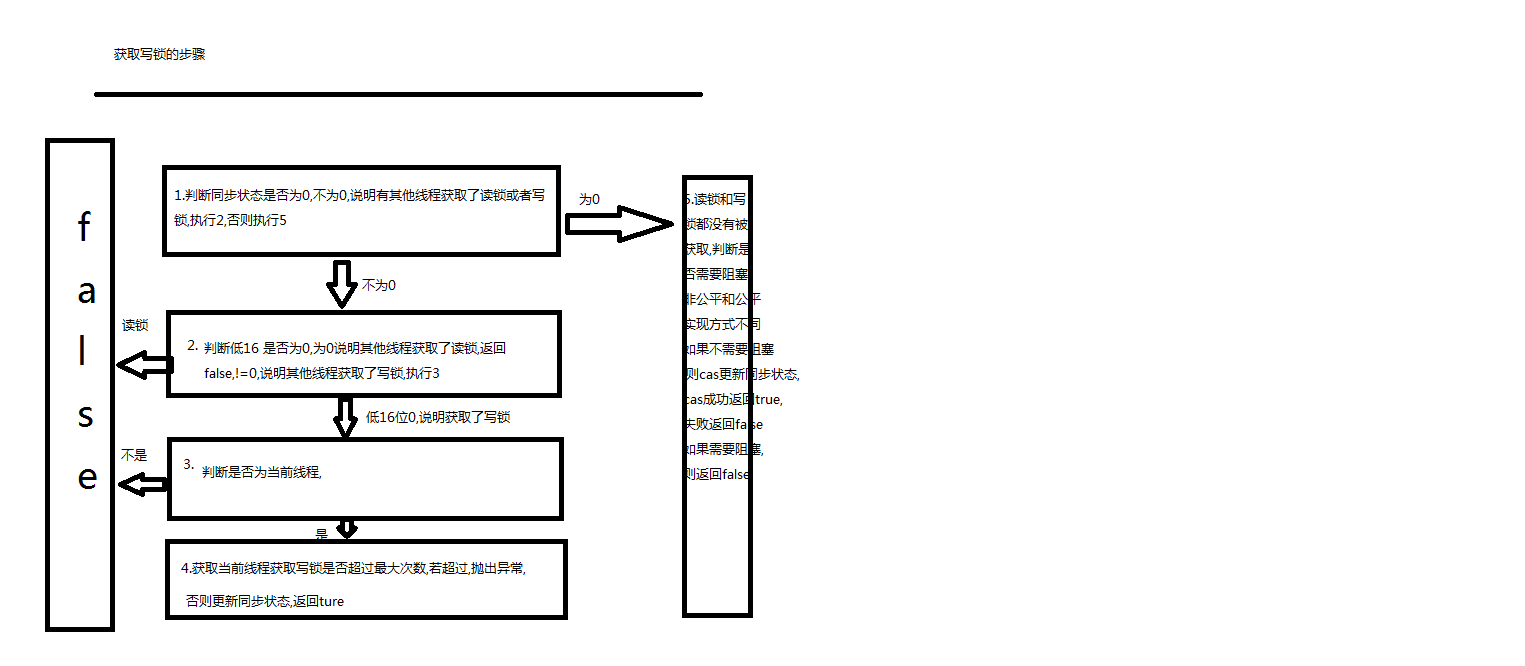
读写锁,基于AQS实现的,同步器需要在同步状态(一个整型变量state)维护多个读线程和一个写线程的状态,使得该状态的设计成为读写锁实现的关键.如果在一个整型变量上维护多种状态,就一定需要"按位切割使用"这个变量,读写锁将变量切分成两个部分,高16位表示读,低16位表示写

写锁是一个可重入的独占锁,使用AQS提供的独占式获取同步状态的策略,

======-===========-========

获取写锁的步骤:

判断同步状态state是否为0,



13.condition执行条件,类似于wait和notify方法,

只用lock只能保证互斥,不能保证协调运行

lock.newCondition()

condition.await()

condition.signal();

14.semaphore信号灯

固定多少线程访问自身,访问离开后别的线程继续访问,构造函数中可以设置访问线程的优先级

semaphore.acquire()线程获取锁

semaphore.release()线程释放锁

semaphore.availablePermits();当前仍允许多少线程访问,

15.CyclicBarrier

`集合后出发,活动后集合开始执行,

cb.await() 等待完成后,在执行下一步(没有成功实现)原因在于每个线程的时间设置有问题.

cb.getNumbersWait();获取已经到达的线程

16.countDownlatch

等待执行条件执行结束后,线程才会执行

线程等待执行条件await() 线程执行条件执行结束后的响应countdown

17.exchanger

两个线程之间的数据交换,同时碰面才会交换数据

所有的都会涉及到线程到达的时间,一定要确保时间不一致,使用Exechanger.exchanger方法交换数据.参数和接收值是要交换的数据,只涉及到两个线程,参数名称无关.

18.阻塞队列

ArrayBlockingQueue 实现了BlockingQueue

添加add(抛异常), offer(放入失败,返回fasle),put(阻塞,直到加入)

remove(抛异常)poll()take()

只有put和take才能实现阻塞的功能

19.Concurrent

线程实现池 runnable callable

阻塞队列(队列,链表)

Synchronizers semaphore信号灯(有限的线程访问) cyclicbarrier(统一) countdownlatch(为0) exechanger()

concurrent connections

常用的并发集合:

hashset

hashmap

arrayList

推荐使用concurrenthashmap

没有concurrent时,使用Connection.sychronizemap(null) 在调用对象时,加上同步synchronize

hashmap和hashset(内部使用hashmap只用了key)

ConcurrentHashMap map差不多

ConcurrentSkipListMap 排序的map(有比较规则) 相当于treemap

ConcurrentSkipListSet 排序的set(有比较规则) 相当于treeset

在迭代的时候,去掉除去倒数2的都会报错

----------------------------------------

CopyOnWriteArrayList 正常

CopyOnWriteArraySet

20.调用多线程打印日志

将生成日志时,将日志添加到阻塞队列中, 在多线程中死循环读出(面试题失败)

21.调用多线程,有序,依次打印(面试题失败)

解析:synchronizeQueue同步队列,

阻塞队列:每一次插入操作必须等待一个取的操作, 保证了有序

等待操作,信号灯,过了才能继续过

22.相同的key不同的输出时间,设置锁,锁表示互斥,