**多线程总结**

# 锁

## 1.2.基本对象锁的使用，锁一段代码

//任何线程要执行下面代码,必须拿到对象o的锁，synchronized (this)锁定当前类  
synchronized (o){

count--;//一个需要加锁修改的变量  
}

## 3. 16.方法锁的使用，锁整个方法。

优化：对象锁锁一段代码效率更高。

public synchronized void m(){//等同于在方法的代码执行时要synchronized(this)  
 count--;  
}

## 4.静态方法中不存在this

public static void mm(){  
 synchronized (T.class){//考虑这里写synchronized(this)是否可以? 静态中不存在this引用  
 *count*--;  
 }  
}

## 5.

## 6.

## 7. 8.

同步和非同步方法可以同时调用，但访问同一个变量时，需要加锁，否则会产生数据更新不同步，或者脏读问题

## 9. 10.

一个同步方法可以调用另一个同步方法，同一个锁（父子类同锁）。

## 11. 异常会释放锁

## 12. volatile 属性可见性修饰符

## 13.

## 14. volatile不可替代synchronized

## 15. count.incrementAndGet();//等于 原子性count++ 用底层

多个方法连续调用中间有间隙，不能保证原子性

17.18.

//o的属性发生改变不影响锁，引用发生改变会影响，并且不以字符串做对象锁

# 集合

## 19. 门闩使用

*实现一个容器，提供两个方法，add,size  
写两个线程，线程1添加10个元素到容器中，线程2实现监控元素个数，当个数到5个时，线程2给出提示并结束*

容器List加volatile

使用门闩CountDownLatch latch = new CountDownLatch(1); //

监控线程先启动并latch.awiat()暂停 添加线程加到5 latch.countDown()打开门闩

## 20.ReentrantLock 公平锁

ReentrantLock lock = new ReentrantLock(true); //true为公平锁

*lock*.lock();//上锁

*lock*.unlock();//放锁

locked = lock.tryLock(5,TimeUnit.*SECONDS*);//尝试访问锁5秒 没成功则返回boolean

## 21.

*\* 面试题：写一个固定容量同步容器，拥有put和get方法，一级getCount方法，  
\* 能够支持两个生产者线程以及10个消费者线程的阻塞调用  
\* 使用wait和notify/notifyALl来实现*

*生产者方法+锁*

*while(lists.size() == MAX){ //判断满  
 this.wait();  
}*

*lists.add(t);//加元素 count++*

*this.notifyAll();//唤醒全部线程*

*消费者方法+锁*

*while(lists.size() == 0){ //判断空  
 this.wait();  
}*

*lists.removeFirst(t);//取出元素 count--*

*this.notifyAll();//唤醒全部线程*

## 22.ThreadLocal<Object>

空间换时间，各线程做自己的事，等同于Map<Thread,Object>

## 23.Singleton 单例模式

# 高并发容器

24.25. 高并发容器  
1.对于map/set的选择使用  
不加锁，非多线程  
hashmap  
treemap  
linkedhashmap  
加锁  
Hashtable 并发量小  
Collections.synchronizedXXX 并发量小  
concurrenthashmap 并发较高时  
concurrentskiplistmap 跳表 并发较高时,且要排序  
2.队列  
非同步  
ArrayList  
LinkedList  
同步  
并发量小  
Collections.synchronizedXXX  
CopyOnWriteList 写少读多  
Queue  
 并发量高  
 ConcurrentLinkedQueue  
 BlockingQueue 阻塞式队列  
 LinkedBlockingQueue 无界队列  
 ArrayBlockingQueue 有界队列  
 TransferQueue 直接交给消费者线程，不经过队列  
 SynchronousQueue 特殊的TransferQueue容量为0  
 DelayQueue 执行定时任务，每个元素添加进去时要设置时间，根据时间排序最长的先执行

# 线程池

## 26.线程池

线程新建与销毁都要消耗内存，使用线程池可以重用线程节省资源

ExecutorService service = Executors.

*1) newFixedThreadPool*(5);//固定个数的线程池

*2) newCachedThreadPool*();//来一个任务启动一个线程，有空闲线程则再使用，默认超过60秒空闲销毁，MAX看设备

*3) newSingleThreadExecutor*();// 只有一个线程，保证任务执行顺序，只用一个线程执行

*4) newScheduledThreadPool*(4);//定时执行任务的线程池

service.scheduleAtFixedRate(Runnable command, long initialDelay, long period,

TimeUnit unit);//runnable任务，延时开始，间隔时间，时间单位

*5) newWorkStealingPool();精灵线程，一个线程空闲了回去别的线程队列里拿任务做，属于守护线程，执行结束了还在后台跑*

6) ForkJoinPool fjp = new ForkJoinPool(); //用于实现精灵线程，可以将任务拆分计算再合并

自定义任务类MyTask继承RecursiveAction（无返回值）或者RecursiveTask<Long>（有返回值），定义参数构造器，重写compute方法，写拆分算法递归调用MyTask拆分计算最后合并返回值。

ForkJoinPool fjp = new ForkJoinPool();//拆分合并  
AddTask task = new AddTask(0,*nums*.length);//  
fjp.execute(task);

7) ThreadPoolExecutor //大部分线程池的源头 ForkJoin不是，属于ExecutorService+

(int corePoolSize, //线程池大小，多少个线程  
int maximumPoolSize,//线程池容量MAX  
long keepAliveTime,// 多长时间消失  
TimeUnit unit,//时间单位  
BlockingQueue<Runnable> workQueue)//任务队列

8) parallelStream() stream api 运用多线程计算

nums.forEach(v->*isPrime*(v));//单线程

nums.parallelStream().forEach(T14\_ParallelStreamAPI::*isPrime*);//多线程

1.synchronized

不使用String对象

优化 使用对象锁代码块 并且减少代码

脏读、更新不同步

2.volatile 属性可见性

3.CountDownLatch 门闩

4.Lock ReentrantLock

new ReentrantLock(true) 公平锁+

lock()与unlock

5.Condition 分组

6.ThreadLocal Map<Thread,Object>

7.单例模式

8.队列集合Queue ConcurrentLinkedQueue