JUC

Future(CompletableFuture异步)

```
/**
 * 异步调用：CompletableFuture  对将来某个事件的结果进行建模
 * 异步执行
 * 成功回调
 * 失败回调
 */
public class Demo1 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
//        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
//Runable接口
//            try {
//                TimeUnit.SECONDS.sleep(2);
//            } catch (InterruptedException e) {
//                e.printStackTrace();
//            }
//            System.out.println(Thread.currentThread().getName() + "
runAsync");
//        });
//
//        System.out.println("1111");
//        future.get();//会阻塞
//        System.out.println("2222");

        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() ->
{ //供给型接口
            System.out.println(Thread.currentThread().getName() + " 提供返回结
果");

            int i = 10 / 0;
            return 1024;
        });

        System.out.println(   future.whenComplete((u, t) -> {// u是返回的结果，t是
异常
            System.out.println("u=" + u);
            System.out.println("t=" + t);
        }).exceptionally((e) -> { // 有异常才会执行
            System.out.println(e.getMessage());
            return 233;
        }).get()    ); // 得到最后的返回值

    }
}
```

Callable

```
public class TestCallable {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
//        new Thread(new Runnable()).start();
//        new Thread(new FutureTask<>()).start();
//        new Thread(new FutureTask<>(Callable)).start();

        MyCallable myCallable = new MyCallable();
        FutureTask<Integer> futureTask = new FutureTask<>(myCallable);
        new Thread(futureTask,"A").start();
        new Thread(futureTask,"B").start(); //结果会缓存

        Integer result = futureTask.get();//可能会产生阻塞，因为需要等待结果返回；放到
最后或者异步通信
        System.out.println(result);
    }
}

class MyCallable implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println("call");
        return 1024;
    }
}
```

CAS

```
/**
 * CAS:compare and swap、compare and set
 * 比较当前内存的值和主存中的值，如果这个值是期望的，那么执行操作；如果不是，就一直循环，所以不
会切换线程的状态（自旋锁）
 * 缺点：1.循环会耗时 2.一次只能保证一个变量的原子性 3.ABA问题 （原子引用）
 *
 * 如果泛型是包装类，注意引用问题（-128-127之间没问题，其它范围可能出现问题）
 */
public class CASDemo {
    public static void main(String[] args) {

//        AtomicInteger atomicInteger = new AtomicInteger(2020);
//
//        System.out.println(atomicInteger.compareAndSet(2020, 2021));
//        System.out.println(atomicInteger.get());
//
////        System.out.println(atomicInteger.compareAndSet(2020, 2022));//未达到
期望，不能修改
////        System.out.println(atomicInteger.get());
//
//        System.out.println(atomicInteger.compareAndSet(2021, 2020));
//        System.out.println(atomicInteger.get());
//
//        //ABA问题
//        System.out.println(atomicInteger.compareAndSet(2020, 6666));
```

```
//        System.out.println(atomicInteger.get());

        //原子引用 （相当于乐观锁，有个版本号
        AtomicStampedReference<Integer> integerAtomicStampedReference = new
AtomicStampedReference<Integer>(1,1);

        new Thread(()->{
            int stamp = integerAtomicStampedReference.getStamp();
            System.out.println("a1->" + stamp);

            System.out.println(integerAtomicStampedReference.compareAndSet(1, 2,
                    integerAtomicStampedReference.getStamp(),
integerAtomicStampedReference.getStamp() + 1));
            System.out.println("a2->" +
integerAtomicStampedReference.getStamp());

            System.out.println(integerAtomicStampedReference.compareAndSet(2, 1,
                    integerAtomicStampedReference.getStamp(),
integerAtomicStampedReference.getStamp() + 1));
            System.out.println("a3->" +
integerAtomicStampedReference.getStamp());

        }).start();

        new Thread(()->{
            int stamp = integerAtomicStampedReference.getStamp();
            System.out.println("b1->" + stamp);

            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(integerAtomicStampedReference.compareAndSet(1,
10,
                    stamp, stamp + 1));
            System.out.println("b2->" +
integerAtomicStampedReference.getStamp());

        }).start();
    }
}
```

Collection

ConcurrentHashMap

```
public class MapMore {
    public static void main(String[] args) {
        //默认等价于 new Map<>(16,0.75) 加载因子 初始化容量
//        HashMap<String, String> map = new HashMap<>();//不安全
```

```
//        Map<Object, Object> map = Collections.synchronizedMap(new HashMap<>
());
        ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();

        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                map.put(Thread.currentThread().getName(),
UUID.randomUUID().toString().substring(0,5));
                System.out.println(map);
            },String.valueOf(i)).start();
        }
    }
}
```

List

```
//java.util.ConcurrentModificationException 并发修改异常
public class UnsafeList {
    public static void main(String[] args) {
//        List<String> list = new ArrayList<String>();//不安全 hashset也同理
//        Vector<Object> list = new Vector<>();//安全，效率低，不推荐使用
//        List<String> list = Collections.synchronizedList(new ArrayList<>());
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();//写入时
复制，CopyOnWrite并发容器用于读多写少的并发场景

        for (int i = 0; i <10 ; i++) {
            new Thread(()->{
                list.add(UUID.randomUUID().toString().substring(0,5));
                System.out.println(list);
            },String.valueOf(i)).start();
        }
    }
}
```

阻塞队列(四组api)

```
/**
 * 阻塞队列的四组API,根据需求自己选择
 * 每组api都有各自的处理策略
 */

/**
 * BlockingQueue是个阻塞队列接口 需设置容量
 */
public class BlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        test4();
    }

    /**
     * 抛出异常
     */
    public static void test1(){
        //指定队列大小
```

```
        ArrayBlockingQueue<Object> arrayBlockingQueue = new ArrayBlockingQueue<>
(3);
        System.out.println(arrayBlockingQueue.add("a"));
        System.out.println(arrayBlockingQueue.add("b"));
        System.out.println(arrayBlockingQueue.add("c"));

        System.out.println(arrayBlockingQueue.element());//查看队首元素

        System.out.println("==================");
        //IllegalStateException: Queue full 抛出异常
//        System.out.println(arrayBlockingQueue.add("d"));

        System.out.println(arrayBlockingQueue.remove());
        System.out.println(arrayBlockingQueue.remove());
        System.out.println(arrayBlockingQueue.remove());

        System.out.println(arrayBlockingQueue.element());//抛出异常，
NoSuchElementException

        //NoSuchElementException 抛出异常
//        System.out.println(arrayBlockingQueue.remove());
    }

    /**
     *有返回值，不抛出异常
     */
    public static void test2(){
        //指定队列大小
        ArrayBlockingQueue<Object> arrayBlockingQueue = new ArrayBlockingQueue<>
(3);

        System.out.println(arrayBlockingQueue.offer("a"));
        System.out.println(arrayBlockingQueue.offer("b"));
        System.out.println(arrayBlockingQueue.offer("c"));
//        System.out.println(arrayBlockingQueue.offer("d"));//返回false，不抛出异常

        System.out.println(arrayBlockingQueue.peek());//返回队首元素

        System.out.println("================");

        System.out.println(arrayBlockingQueue.poll());
        System.out.println(arrayBlockingQueue.poll());
        System.out.println(arrayBlockingQueue.poll());

        System.out.println(arrayBlockingQueue.peek());//返回null，不抛出异常

        System.out.println(arrayBlockingQueue.poll());//返回null，不抛出异常
    }

    /**
     * 等待，阻塞（一直阻塞）
     */
    public static void test3() throws InterruptedException {
        //指定队列大小
        ArrayBlockingQueue<Object> arrayBlockingQueue = new ArrayBlockingQueue<>
(3);

        arrayBlockingQueue.put("a");
```

```
        arrayBlockingQueue.put("b");
        arrayBlockingQueue.put("c");
//        arrayBlockingQueue.put("d");//队列没位置，线程一直阻塞

        System.out.println(arrayBlockingQueue.take());
        System.out.println(arrayBlockingQueue.take());
        System.out.println(arrayBlockingQueue.take());
        System.out.println(arrayBlockingQueue.take());//队列空了，线程一直阻塞
    }

    /**
     * 等待，阻塞（等待超时）
     */
    public static void test4() throws InterruptedException {
        //指定队列大小
        ArrayBlockingQueue<Object> arrayBlockingQueue = new ArrayBlockingQueue<>
(3);

        System.out.println(arrayBlockingQueue.offer("a"));
        System.out.println(arrayBlockingQueue.offer("b",2,TimeUnit.SECONDS));//
能添加就直接添加
        System.out.println(arrayBlockingQueue.offer("c"));

        System.out.println(arrayBlockingQueue.offer("d", 2,
TimeUnit.SECONDS));//满，等待，超时两秒就返回false

        System.out.println("===============");

        System.out.println(arrayBlockingQueue.poll());
        System.out.println(arrayBlockingQueue.poll(3,TimeUnit.SECONDS));//不空就
直接取出
        System.out.println(arrayBlockingQueue.poll());

        System.out.println(arrayBlockingQueue.poll(3,TimeUnit.SECONDS));//空，等
待，超时就返回null
    }
}
```

同步队列

```
/**
 * 同步队列  是BlockingQueue的子类
 * 容量为1，必须等待取出来后，才能往里面放一个元素
 * 用put() take()
 */
public class SynchronousQueueDemo {
    public static void main(String[] args) {
        SynchronousQueue queue = new SynchronousQueue();
    }
}
```

三个常用类

CountDownLatch

```
/**
 * 可理解为 减法计数器 指定线程个数执行后再执行一些操作
 * countDownLatch.countDown()使计数器数量-1
 * countDownLatch.await()等待计数器为0，再执行下去
 *
 */
public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(10);//设置初始值

        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName() + "go out");
                countDownLatch.countDown();//减1
            },String.valueOf(i+1)).start();
        }

        countDownLatch.await();//等待计数器值为0，然后被唤醒

        System.out.println("关门");
    }
}
```

CyclicBarrier

```
/**
 * 可理理解为 加法计数器
 * 和countDownLatch差不多，但是可以实现更高级的功能
 */
public class CyclicBarrierDemo {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("召唤神龙成功");
        });

        for (int i = 0; i < 7; i++) {
            final int temp = i; //lambda表达式相当于是个匿名内部类，所以i的作用域达不到
的
            new Thread(()->{
                System.out.println(Thread.currentThread().getName() + "获得第" +
(temp+1) + "颗龙珠");
                try {
                    cyclicBarrier.await();//等待，也相当于计数了，到达7的时候会被唤醒，
唤醒前会先去执行“召唤神龙”那条线程
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
```

```
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + "恭喜");
            }).start();
        }

    }
}
```

Semaphore

```
/**
 * 信号量，给定指定的许可证，同一时间只有指定数量个线程可以运行
 * 可以用来限流
 * semaphore.acquire()获得许可
 * semaphore.release()释放许可
 */
public class SemaphoreDemo {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(3);//指定同时运行的线程数量，这里指停车位

        for (int i = 0; i < 6; i++) {
            new Thread(()->{
                try {
                    semaphore.acquire(); //获得许可
                    System.out.println(Thread.currentThread().getName() + "获得了
车位");
                    TimeUnit.SECONDS.sleep(2);
                    System.out.println(Thread.currentThread().getName() + "离开了
车位");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    semaphore.release();//释放车位
                }
            },String.valueOf(i+1)).start();
        }

    }
}
```

ForkJoin

> 任务类

```
/**
 * 拆分合并，工作窃取
 * 把大任务拆成小任务，并行执行，提高效率。
 * 适合大数据量，且拆分后对计算任务无影响
 *
 * 用法:
 * 1.先写个任务类，继承RecursiveTask
```

```
 * 2.用new ForkJoinPool()的submit或execute方法调用
 */

import java.util.concurrent.RecursiveTask;

/**
 * 求和计算任务
 */
public class ForkJoinDmo extends RecursiveTask<Long> {//任务类
    private long start;
    private long end;

    //临界值
    private long temp = 10000L;

    public ForkJoinDmo(long start,long end){
        this.start = start;
        this.end = end;
    }


    @Override
    protected Long compute() {
        if((end - start)<temp){
            long sum = 0L;
            for(long i = start; i<=end; i++){
                sum += i;
            }
            return sum;

        }else{
            //分支合并计算
            long middle = (start + end)/2;
            ForkJoinDmo task1 = new ForkJoinDmo(start, middle);
            ForkJoinDmo task2 = new ForkJoinDmo(middle+1, end);
            task1.fork();//拆分任务，把任务压入线程队列
            task2.fork();

            return task1.join() + task2.join();

        }
    }
}
```

测试类

```
public class Test {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        test1();
        test2();
        test3();
    }

    public static void test1() {
        long sum = 0L;
```

```
        long start = System.currentTimeMillis();
        for (long i =1; i <= 10_0000_0000; i++) {
            sum += i;
        }
        long end = System.currentTimeMillis();

        System.out.println("sum=" + sum + " 时间：" + (end - start));

    }

    public static void test2() throws ExecutionException, InterruptedException {
        long start = System.currentTimeMillis();

        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinTask<Long> task = new ForkJoinDmo(1L, 10_0000_0000L);
        ForkJoinTask<Long> submit = forkJoinPool.submit(task);
        Long sum = submit.get();

        long end = System.currentTimeMillis();

        System.out.println("sum=" + sum + "时间：" + (end - start));
    }

    public static void test3() {
        long start = System.currentTimeMillis();
        long sum = LongStream.rangeClosed(0L,
10_0000_0000L).parallel().reduce(0,Long::sum);
        long end = System.currentTimeMillis();

        System.out.println("sum=" + sum + "时间：" + (end - start));

    }
}
```

Lock

死锁

```
/**
 * 死锁问题
 *
 * 解决方法：
 * 1.使用jps -l定位进程号
 * 2.使用jstack 进程号 查看进程信息
 */
public class DeadLock {
    public static void main(String[] args) {
        String s1 = "bob";
        String s2 = "john";

        new Thread(new Dead(s1,s2),"A").start();
        new Thread(new Dead(s2,s1),"B").start();
    }
}
```

```
class Dead implements Runnable{
    private String s1;
    private String s2;

    public Dead(String s1, String s2) {
        this.s1 = s1;
        this.s2 = s2;
    }

    @Override
    public void run() {
        synchronized (s1){
            System.out.println(Thread.currentThread().getName() + "  " + s1);

            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            synchronized (s2){
                System.out.println(Thread.currentThread().getName() + "  " +
s2);
            }
        }
    }
}
```

Lock接口

```
/**
 * Lock接口
 * ReentrantLock 可重入锁 是lock的实现类
 * synchronized锁与lock锁的区别：
 * 1.synchronized是关键字，Lock是接口类
 * 2.synchronized无法获取锁的状态，lock可以判断是否获得到了锁
 * 3.synchronized会自动释放锁，lock只能手动释放，否则会出现死锁
 * 4.synchronized一个线程获得锁，另一个就只能等待；lock就不一定会等待下去（trylock()方法）
 * 5.synchronized是可重入锁，不可中断，非公平；lock锁，可重入锁，可以判断锁，默认非公平锁
（可以设置 如new reentrantlock(true)创建公平锁）
 * 6.synchronized适合锁少量代码，lock适合锁大量代码
 */

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class InterLock {
    public static void main(String[] args) {
        Ticket1 t = new Ticket1();
        new Thread(()->{for(int i =0 ;i<30; i++) t.sale();},"A").start();
        new Thread(()->{for(int i =0 ;i<30; i++) t.sale();},"B").start();
        new Thread(()->{for(int i =0 ;i<30; i++) t.sale();},"C").start();
    }
}
```

```
class Ticket1{
    private int tick = 50;
    Lock lock = new ReentrantLock();

    public void sale() {
        lock.lock();

        try{
            if(tick > 0 ){
                System.out.println(Thread.currentThread().getName() + "买了第" +
tick-- + "张票，剩余" + tick +"张票");
            }
        }finally {
            lock.unlock();
        }
    }

}
```

可重入锁

```
/**
 * 可重入锁，表示拿到了外面的锁，就自动拿到了里面的锁
 */
public class RepeatLock {
    public static void main(String[] args) {
        Phone phone = new Phone();

        new Thread(()->{
            phone.sms();
        },"A").start();

        new Thread(()->{
            phone.sms();
        },"B").start();
    }

}

//用lock锁同理
class Phone{
    public synchronized void sms(){
        System.out.println(Thread.currentThread().getName() + "sms");
        call();
    }

    public synchronized void call() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + "call");
    }
```

```
}
```

自旋锁

```
/**
 * 自己实现自旋锁
 */
public class SpinLock {
    AtomicReference<Thread> atomicReference = new AtomicReference<>();

    //加锁
    public void myLock(){
        Thread thread = Thread.currentThread();

        while (!atomicReference.compareAndSet(null, thread)) {

        }
        System.out.println(thread.getName() + "-->lock");
    }

    //解锁
    public void myUnLock(){
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + "-->unlock");

        atomicReference.compareAndSet(thread,null);
    }

}
```

> 测试

```
public class SpinLockTest {
    public static void main(String[] args) {
        SpinLock spinLock = new SpinLock();

        new Thread(()->{
            spinLock.myLock();

            try{
                TimeUnit.SECONDS.sleep(3);

            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                spinLock.myUnLock();
            }

        },"T1").start();

        new Thread(()->{
            spinLock.myLock();
            try {
                TimeUnit.SECONDS.sleep(1);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                spinLock.myUnLock();
            }
        },"T2").start();
    }
}
```

读写锁

```
/**
 * 读写锁
 *
 * 读-读 可以共存
 * 读-写 不能共存
 * 写-写 不能共存
 *
 * 写锁是独占锁
 * 读锁是共享锁
 */

public class ReadWriteLockDemo {
    public static void main(String[] args) {
        MyCache cache = new MyCache();

        for (int i = 1; i <= 5; i++) {
            final int temp = i;
            new Thread(()->{
                cache.put(temp + "",temp + "");
            },String.valueOf(i)).start();
        }

        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(()->{
                cache.get(temp + "");
            },String.valueOf(i)).start();
        }
    }
}

class MyCache{
    private volatile Map<String,Object> map= new HashMap<>();
    private ReadWriteLock lock  = new ReentrantReadWriteLock();

    //写
    public void put(String key,Object value){
        lock.writeLock().lock();

        try {
            System.out.println(Thread.currentThread().getName() + "写入" + key);
            map.put(key,value);
            System.out.println(Thread.currentThread().getName() + "写入OK");
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.writeLock().unlock();
        }

    }

    //读
    public void get(String key) {
        lock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + "读取" + key);
            map.get(key);
            System.out.println(Thread.currentThread().getName() + "读取OK");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

生产者消费者进阶

Condition

```
/**
 * juc生产者消费者问题
 *
 * synchronized对应Lock接口的lock和unlock
 * wait和notify notifyAll对应condition接口的await和signal signalAll
 */
public class Better {
    public static void main(String[] args) {
        Data2 data = new Data2();
        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                try {
                    data.decrease();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                try {
                    data.increase();
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
        },"B").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                try {
                    data.decrease();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"C").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                try {
                    data.increase();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"D").start();
    }
}


//判断等待，业务，通知
class Data2{  //资源类
    private int num = 0;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();


    public void increase() throws InterruptedException {
        lock.lock();
        try{
            while(num != 0){//一定要用while，不用if
//              this.wait();
                condition.await();
            }

            num++;
            System.out.println(Thread.currentThread().getName() + "->" + num);
//          this.notifyAll();
            condition.signalAll();
        }finally {
            lock.unlock();
        }
    }

    public void decrease() throws InterruptedException {
        lock.lock();
        try{
            while (num == 0) {//用while
//              this.wait();
                condition.await();
            }
```

```
            num--;
            System.out.println(Thread.currentThread().getName() + "->" + num);
//          this.notifyAll();
            condition.signalAll();
        }finally {
            lock.unlock();
        }
    }

}
```

实现精确唤醒

```
/**
 * lock还能实现精确唤醒指定线程
 */
public class MoreBetter {
    public static void main(String[] args) {
        Data3 data = new Data3();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                data.ptintA();
            }
        },"1").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                data.ptintB();
            }
        },"2").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                data.ptintC();
            }
        },"3").start();
    }
}

class Data3{
    private int num = 1;
    private Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();

    public void ptintA(){
        lock.lock();
        try {
            while (num != 1){
                condition1.await();
            }
            num = 2;
            System.out.println(Thread.currentThread().getName() + "AAAAA");
```

```
            condition2.signal();

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }

    }
    public void ptintB(){
        lock.lock();
        try {
            while (num != 2){
                condition2.await();
            }
            num = 3;
            System.out.println(Thread.currentThread().getName() + "BBBBB");
            condition3.signal();

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }

    }
    public void ptintC(){
        lock.lock();
        try {
            while (num != 3){
                condition3.await();
            }
            num = 1;
            System.out.println(Thread.currentThread().getName() + "CCCCC");
            condition1.signal();

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }

    }
}
```

线程池

Excutors

```
/**
 * Executors可以看成是个工具类 有三大方法
 * 不推荐使用Executors,推荐使用ThreadPoolExecutor创建线程池
 * ThreadPoolExecutor是Executors的底层实现，所以要用底层去创建线程池
 *
```

```
 * 用线程池的方式创建线程
 */
public class ExecutorsDemo {
    public static void main(String[] args) {
//        ExecutorService service = Executors.newSingleThreadExecutor();//单个线
程
//        ExecutorService service = Executors.newFixedThreadPool(5);//创建一个固定
大小的线程池
        ExecutorService service = Executors.newCachedThreadPool();//可伸缩的

        try {
            for (int i = 0; i < 10; i++) {
                service.execute(()->{
                    System.out.println(Thread.currentThread().getName() + "
ok");
                    System.out.println(Thread.currentThread().getName() + "
hh");
                });
            }
        } finally {
            service.shutdown();
        }

    }
}
```

ThreadPoolExecutor与七大参数

```
/**
 * 推荐使用的线程池技术
 * 七大参数
 * 1.核心线程大小
 * 2.最大线程大小
 * 3.存活时间，线程池里的线程超过指定的时间没有被人调用，就会释放（核心线程大小应该不会释放）
 * 4.存活时间单位
 * 5.阻塞队列
 * 6.线程工厂，创建线程的，一般不用动
 * 7.拒绝策略
 *
 * 4种拒绝策略：
 *AbortPolicy()  阻塞队列和线程池都满了，还有线程进来，就不处理了，并抛出异常
 * CallerRunsPolicy()  阻塞队列和线程池都满了，还有线程进来，就哪来的回哪里
 * DiscardPolicy() 阻塞队列和线程池都满了，不会抛出异常，丢弃任务
 * DiscardOldestPolicy() 阻塞队列和线程池都满了,有新的线程要执行，就会抛弃阻塞队列里最老的
线程
 */
public class SevenParaThreadPoolExecutor {
    public static void main(String[] args) {
        //最大线程数该如何定义？  用来调优
        //CPU密集型 几核就取几，保持CPU效率最高
        //IO密集型 判断程序中有多少个十分耗费IO的线程，取大于该值的数，比如取2倍

        ExecutorService service = new ThreadPoolExecutor(
                2,
                Runtime.getRuntime().availableProcessors(),//CPU核数
```

```
                3,
                TimeUnit.SECONDS,
                new LinkedBlockingDeque<>(3),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.DiscardOldestPolicy()
        );

        try {
            for (int i = 0; i < 9; i++) { //i值自己改着测试
                service.execute(()->{
                    System.out.println(Thread.currentThread().getName() + "
ok");
                    System.out.println(Thread.currentThread().getName() + "
hh");
                });
            }
        } finally {
            service.shutdown();
        }

    }
}
```

volatile

可见性

```
/**
 * 8种操作，两两必须成对出现 （这不能理解为是原子性） 比如lock必须unlock
 *
 * volatile保证可见性，不保证原子性，禁止指令重排
 */

//可见性
public class JMM {
    private volatile static int num = 0;

    public static void main(String[] args) {//如果不加volatile，主存的值该线程是不可
见的，所以会一直执行下去
        new Thread(()->{
            while (num == 0){

            }
        }).start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        num=1;
        System.out.println(num);
```

```
    }
}
```

原子性

```
//volatile不保证原子性    synchronized能保证
//用volatile + 原子类 实现原子性
public class Atomic {
    private volatile static AtomicInteger num = new AtomicInteger();

    public static void test() {
//        num ++;
        num.getAndIncrement();  //CAS原理，效率高 非常高效
    }


    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            new Thread(()->{
                for (int i1 = 0; i1 < 1000; i1++) {
                    test();
                }
            }).start();
        }

        while (Thread.activeCount() > 2) {
            Thread.yield();
        }

        System.out.println(num);//正常应该为20000
    }

}
```

指令重排

> 见单例模式

```
/**
 * 懒汉式
 * //普通懒汉式有线程安全问题
 *
 * 最安全的单例：枚举
 */
public class LazyMan {

    private static boolean flag = false; // 为了防止反射

    private LazyMan() {

        synchronized (LazyMan.class) {
//            if (lazyMan != null) {
//                throw new RuntimeException("不要用反射破坏单例模式");
//            }
```

```
            if (flag == false) {
                flag = true;
            }else{
                throw new RuntimeException("不要用反射破坏单例模式");
            }
        }
    }

    private volatile static LazyMan lazyMan; //防止指令重排

    public static LazyMan getInstance() {

        //DCL懒汉式，双重检测  直接锁方法效率太低了
        if (lazyMan == null) {
            synchronized (LazyMan.class){
                if (lazyMan == null) {
                    lazyMan = new LazyMan();//1.分配内存空间 2.构造函数，初始化 3.指向该区域
                }
            }
        }

        return lazyMan;
    }
}
```