生产者与消费者模式

1.使用wait和notify

```
public class Test {
    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>(queueSize);

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
                synchronized (queue) {
                    while(queue.size() == 0){
                        try {
                            System.out.println("队列空，等待数据");
                            queue.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
                    }
                    queue.poll();           //每次移走队首元素
                    queue.notify();
                    System.out.println("从队列取走一个元素，队列剩余"+queue.size()+"个元素");
                }
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                synchronized (queue) {
                    while(queue.size() == queueSize){
                        try {
                            System.out.println("队列满，等待有空余空间");
```

```
                            queue.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
                    }
                    queue.offer(1);         //每次插入一个元素
                    queue.notify();
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+(queueSize-queue.size()));
                }
            }
        }
    }
}
```

2.使用Condition

```
public class Test {
    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>(queueSize);
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
                lock.lock();
                try {
                    while(queue.size() == 0){
                        try {
                            System.out.println("队列空，等待数据");
                            notEmpty.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    queue.poll();                //每次移走队首元素
                    notFull.signal();
                    System.out.println("从队列取走一个元素，队列剩余"+queue.size()+"个元素");
                } finally{
                    lock.unlock();
                }
```

```
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                lock.lock();
                try {
                    while(queue.size() == queueSize){
                        try {
                            System.out.println("队列满，等待有空余空间");
                            notFull.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    queue.offer(1);         //每次插入一个元素
                    notEmpty.signal();
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+(queueSize-queue.size()));
                } finally{
                    lock.unlock();
                }
            }
        }
    }
}
```

3.使用阻塞队列

```
public class Test {
    private int queueSize = 10;
    private ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(queueSize);

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
```

```
                try {
                    queue.take();
                    System.out.println("从队列取走一个元素，队列剩余"+queue.size()+"个元素");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                try {
                    queue.put(1);
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+(queueSize-queue.size()));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

一。两个线程交替打印奇偶数：

1.基于等待通知模式：

```
public class TwoThreadWaitNotify {
        private int start = 1;
        private boolean flag = false;
        public static void main(String[] args) {
            TwoThreadWaitNotify twoThread = new TwoThreadWaitNotify();
            Thread t1 = new Thread(new OuNum(twoThread));
            t1.setName("A");
            Thread t2 = new Thread(new JiNum(twoThread));
            t2.setName("B");
            t1.start();
            t2.start();
        }
        /** * 偶数线程 */
        public static class OuNum implements Runnable {
            private TwoThreadWaitNotify number;
            public OuNum(TwoThreadWaitNotify number) {
```

```
                this.number = number;
            }
            @Override
            public void run() {
                while (number.start <= 100) {
                    synchronized (TwoThreadWaitNotify.class) {
                        System.out.println("偶数线程抢到锁了");
                        if (number.flag) {
                            System.out.println(Thread.currentThread().getName()
+ "+-+偶数" + number.start);
                            number.start++;
                            number.flag = false;
                            TwoThreadWaitNotify.class.notify();
                        } else {
                            try {
                                TwoThreadWaitNotify.class.wait();
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                }
            }
        }
        /** * 奇数线程 */
        public static class JiNum implements Runnable {
            private TwoThreadWaitNotify number;
            public JiNum(TwoThreadWaitNotify number) {
                this.number = number;
            }
            @Override
            public void run() {
                while (number.start <= 100) {
                    synchronized (TwoThreadWaitNotify.class) {
                        System.out.println("奇数线程抢到锁了");
                        if (!number.flag) {
                            System.out.println(Thread.currentThread().getName()
+ "+-+奇数" + number.start);
```

```
                    number.start++;
                    number.flag = true;
                    TwoThreadWaitNotify.class.notify();
                } else {
                    try {
                        TwoThreadWaitNotify.class.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

2.基于锁模式

```
packagecn.edu.neu.practice;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class TwoThread {
    private int start = 1;
    /**
     * 保证内存可见性 其实用锁了之后也可以保证可见性 这里用不用 volatile 都一样
     */
    private boolean flag = false;
    /**
     * 重入锁
     */
    private final static Lock LOCK = new ReentrantLock();
    public static void main(String[] args) {
        TwoThread twoThread = new TwoThread();
        Thread t1 = new Thread(new OuNum(twoThread));
        t1.setName("t1");
        Thread t2 = new Thread(new JiNum(twoThread));
        t2.setName("t2");
        t1.start();
        t2.start();
    }
```

```
    /**
     * 偶数线程
     */
    public static class OuNum implements Runnable {
        private TwoThread number;
        public OuNum(TwoThread number) {
            this.number = number;
        }
        @Override
        public void run() {
            while (number.start <= 100) {
                if (number.flag) {
                    try {
                        LOCK.lock();
                        System.out.println(Thread.currentThread().getName() +
"+-+" + number.start);
                        number.start++;
                        number.flag = false;
                    } finally {
                        LOCK.unlock();
                    }
                } else {
                    try {
                        // 防止线程空转
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
    /**
     * 奇数线程
     */
    public static class JiNum implements Runnable {
        private TwoThread number;
        public JiNum(TwoThread number) {
```

```
            this.number = number;
        }
        @Override
        public void run() {
            while (number.start <= 100) {
                if (!number.flag) {
                    try {
                        LOCK.lock();
                        System.out.println(Thread.currentThread().getName() +
"+-+" + number.start);
                        number.start++;
                        number.flag = true;
                    } finally {
                        LOCK.unlock();
                    }
                } else {
                    try {
                        // 防止线程空转
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

二.如何顺序执行多个线程

1.利用join函数

在main方法中，先是调用了t1.start方法，启动t1线程，随后调用t1的join方法，main所在的主线程就需要等待t1子线程中的run方法运行完成后才能继续运行，所以主线程卡在t2.start方法之前等待t1程序。等t1运行完后，主线程重新获得主动权，继续运行t2.start和t2.join方法，与t1子线程类似，main主线程等待t2完成后继续执行，如此执行下去，join方法就有效的解决了执行顺序问题。

```
    public class Test {
        public static void main(String[] args) throws InterruptedException {
            Thread t1 = new Thread(new MyThread1());
            Thread t2 = new Thread(new MyThread2());
            Thread t3 = new Thread(new MyThread3());
```

```
            t1.start();
            t1.join();
            t2.start();
            t2.join();
            t3.start();
        }
    }
    class MyThread1 implements Runnable {
        @Override
        public void run() {
            System.out.println("I am thread 1");
        }
    }
    class MyThread2 implements Runnable {
        @Override
        public void run() {
            System.out.println("I am thread 2");
        }
    }
    class MyThread3 implements Runnable {
        @Override
        public void run() {
            System.out.println("I am thread 3");
        }
    }
```

2.利用线程池

利用并发包里的Excutors的newSingleThreadExecutor产生一个单线程的线程池，而这个线程池的底层原理就是一个先进先出（FIFO）的队列。代码中executor.submit依次添加了123线程，按照FIFO的特性，执行顺序也就是123的执行结果，从而保证了执行顺序。

```
        public static void main(String[] args) throws InterruptedException {
            Thread t1 = new Thread(new MyThread1());
            Thread t2 = new Thread(new MyThread2());
            Thread t3 = new Thread(new MyThread3());
            ExecutorService executor = Executors.newSingleThreadExecutor();
            executor.submit(t1);
            executor.submit(t2);
            executor.submit(t3);
            executor.shutdown();
```

```
    }
}
class MyThread1 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 1");
    }
}
class MyThread2 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 2");
    }
}
class MyThread3 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 3");
    }
}
```

快速失败

迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。 注意：这里异常的抛出条件是检测到 modCount！=expectedmodCount 这个条件。如果集合发生变化时修改modCount值刚好又设置为了expectedmodCount值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的bug。场景：java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）

它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生fail-fast机制。

ConcurrentModificationException异常：当方法检测到对象的并发修改，但不允许这种修改时就抛出该异常。同时需要注意的是，该异常不会始终指出对象已经由不同线程并发修改，如果单线程违反了规则，同样也有可能会抛出改异常。ConcurrentModificationException 应该仅用于检测 bug。

产生原因： 迭代器在调用next()、remove()方法时都是调用checkForComodification()方法，该方法主要就是检测modCount == expectedModCount ? 若不等则抛出ConcurrentModificationException 异常，从而产生fail-fast机制。 expectedModCount 的值是不会改变的，所以会变的是modCount。集合中无论add、remove、clear方法只要是涉及了改变ArrayList元素的个数的方法都会导致modCount的改变。所以我们这里

可以初步判断由于expectedModCount 的值与modCount的改变不同步，导致两者之间不等从而产生fail-fast机制。

解决方法：

方案一：在遍历过程中所有涉及到改变modCount值得地方全部加上synchronized或者直接使用Collections.synchronizedList，这样就可以解决。但是不推荐，因为增删造成的同步锁可能会阻塞遍历操作。

方案二：**：**使用CopyOnWriteArrayList来替换ArrayList。

CopyOnWriteArrayList为ArrayList 的一个线程安全的变体，其中所有可变操作（add、set 等等）都是通过对底层数组进行一次新的复制来实现的。 该类产生的开销比较大，但是在两种情况下，它非常适合使用。1：在不能或不想进行同步遍历，但又需要从并发线程中排除冲突时。2：当遍历操作的数量大大超过可变操作的数量时。遇到这两种情况使用CopyOnWriteArrayList来替代ArrayList再适合不过了。

第一、CopyOnWriterArrayList的无论是从数据结构、定义都和ArrayList一样。它和ArrayList一样，同样是实现List接口，底层使用数组实现。在方法上也包含add、 remove、 clear、 iterator等方法。 第二、CopyOnWriterArrayList根本就不会产生ConcurrentModificationException异常，也就是它使用迭代器完全不会产生fail-fast机制。 就在于copy原来的array，再在copy数组上进行add操作，这样做就完全不会影响COWIterator中的array了。

安全失败机制：

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。

缺点：基于拷贝内容的优点是避免了Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。