并发队列ConcurrentLinkedQueue和阻塞队列LinkedBlockingQueue用法

在Java多线程应用中，队列的使用率很高，多数生产消费模型的首选数据结构就是队列(先进先出)。Java提供的线程安全的Queue可以分为阻塞队列和非阻塞队列，其中阻塞队列的典型例子是BlockingQueue，非阻塞队列的典型例子是ConcurrentLinkedQueue，在实际应用中要根据实际需要选用阻塞队列或者非阻塞队列。

注：什么叫线程安全？这个首先要明确。线程安全就是说多线程访问同一代码，不会产生不确定的结果。

并行和并发区别

1、并行是指两者同时执行一件事，比如赛跑，两个人都在不停的往前跑；
2、并发是指资源有限的情况下，两者交替轮流使用资源，比如一段路(单核CPU资源)同时只能过一个人，A走一段后，让给B，B用完继续给A ，交替使用，目的是提高效率

LinkedBlockingQueue

由于LinkedBlockingQueue实现是线程安全的，实现了先进先出等特性，是作为生产者消费者的首选，
LinkedBlockingQueue 可以指定容量，也可以不指定，不指定的话，默认最大是Integer.MAX_VALUE，其中主要用到put和take方法，put方法在队列满的时候会阻塞直到有队列成员被消费，take方法在队列空的时候会阻塞，直到有队列成员被放进来。

```
package cn.thread;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingQueue;

/**
 * 多线程模拟实现生产者／消费者模型
 *
 * @author 林计钦
 * @version 1.0 2013-7-25 下午05:23:11
 */
public class BlockingQueueTest2 {
    /**
     *
     * 定义装苹果的篮子
     *
     */
    public class Basket {
        // 篮子，能够容纳3个苹果
        BlockingQueue<String> basket = new LinkedBlockingQueue<String>(3);

        // 生产苹果，放入篮子
        public void produce() throws InterruptedException {
            // put方法放入一个苹果，若basket满了，等到basket有位置
            basket.put("An apple");
        }

        // 消费苹果，从篮子中取走
        public String consume() throws InterruptedException {
            // take方法取出一个苹果，若basket为空，等到basket有苹果为止(获取并移除此队列的头部)
            return basket.take();
```

```
        }
    }

    // 定义苹果生产者
    class Producer implements Runnable {
        private String instance;
        private Basket basket;

        public Producer(String instance, Basket basket) {
            this.instance = instance;
            this.basket = basket;
        }

        public void run() {
            try {
                while (true) {
                    // 生产苹果
                    System.out.println("生产者准备生产苹果: " + instance);
                    basket.produce();
                    System.out.println("!生产者生产苹果完毕: " + instance);
                    // 休眠300ms
                    Thread.sleep(300);
                }
            } catch (InterruptedException ex) {
                System.out.println("Producer Interrupted");
            }
        }
    }

    // 定义苹果消费者
    class Consumer implements Runnable {
        private String instance;
        private Basket basket;

        public Consumer(String instance, Basket basket) {
            this.instance = instance;
            this.basket = basket;
        }

        public void run() {
            try {
                while (true) {
                    // 消费苹果
                    System.out.println("消费者准备消费苹果: " + instance);
                    System.out.println(basket.consume());
                    System.out.println("!消费者消费苹果完毕: " + instance);
                    // 休眠1000ms
                    Thread.sleep(1000);
                }
            } catch (InterruptedException ex) {
                System.out.println("Consumer Interrupted");
            }
        }
    }
```

```
    public static void main(String[] args) {
        BlockingQueueTest2 test = new BlockingQueueTest2();

        // 建立一个装苹果的篮子
        Basket basket = test.new Basket();

        ExecutorService service = Executors.newCachedThreadPool();
        Producer producer = test.new Producer("生产者001", basket);
        Producer producer2 = test.new Producer("生产者002", basket);
        Consumer consumer = test.new Consumer("消费者001", basket);
        service.submit(producer);
        service.submit(producer2);
        service.submit(consumer);
        // 程序运行5s后，所有任务停止
//        try {
//            Thread.sleep(1000 * 5);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//        service.shutdownNow();
    }

}
```

ConcurrentLinkedQueue

ConcurrentLinkedQueue是Queue的一个安全实现．Queue中元素按FIFO原则进行排序．采用CAS操作，来保证元素的一致性。

LinkedBlockingQueue是一个线程安全的阻塞队列，它实现了BlockingQueue接口，BlockingQueue接口继承自java.util.Queue接口，并在这个接口的基础上增加了take和put方法，这两个方法正是队列操作的阻塞版本。

```
package cn.thread;

import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConcurrentLinkedQueueTest {
    private static ConcurrentLinkedQueue<Integer> queue = new
ConcurrentLinkedQueue<Integer>();
    private static int count = 2; // 线程个数
    //CountDownLatch，一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
    private static CountDownLatch latch = new CountDownLatch(count);

    public static void main(String[] args) throws InterruptedException {
        long timeStart = System.currentTimeMillis();
        ExecutorService es = Executors.newFixedThreadPool(4);
        ConcurrentLinkedQueueTest.offer();
```

```
        for (int i = 0; i < count; i++) {
            es.submit(new Poll());
        }
        latch.await(); //使得主线程(main)阻塞直到latch.countDown()为零才继续执行
        System.out.println("cost time " + (System.currentTimeMillis() - timeStart) +
"ms");
        es.shutdown();
    }

    /**
     * 生产
     */
    public static void offer() {
        for (int i = 0; i < 100000; i++) {
            queue.offer(i);
        }
    }


    /**
     * 消费
     *
     * @author 林计钦
     * @version 1.0 2013-7-25 下午05:32:56
     */
    static class Poll implements Runnable {
        public void run() {
            // while (queue.size()>0) {
            while (!queue.isEmpty()) {
                System.out.println(queue.poll());
            }
            latch.countDown();
        }
    }
}
```

运行结果:

costtime 2360ms

改用**while (queue.size()>0)**后

运行结果:

cost time 46422ms

结果居然相差那么大，看了下ConcurrentLinkedQueue的API原来.size()是要遍历一遍集合的，难怪那么慢，所以尽量要避免用size而改用isEmpty().

总结了下， 在单位缺乏性能测试下，对自己的编程要求更加要严格，特别是在生产环境下更是要小心谨慎。