Java 并发编程（四）并发容器

作者：wlgzs-sjl
原文链接：https://ld246.com/article/1574582534816
来源网站：链滴

并发容器的引出: 售票问题

  有N张火车票，每张票都有一个编号同时有10个窗口对外售票，请写一个模程序。

实现1:使用List-非原子性操作

```
public class TicketSeller1 {

    static List<String> tickets = new ArrayList<>();

    static {
        for (int i = 0; i < 1000; i++) {
            tickets.add("票-" + i);
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                while (tickets.size() > 0) {
            // List的remove操作不是原子性的
                    System.out.println("销售了： " + tickets.remove(0));
                }
            }).start();
        }
    }
}
```

输出如下,我们发现发生了重售:

```
...
销售了:票-998
销售了:票-999
销售了:票-999
```

实现2:使用Vector-判断与操作分离,复合操作不保证原子性

  Vector的所有操作均为原子性的,但仍会出现问题,因为判断与操作是分离的,成的复合操作不能保证原子性。

```
public class TicketSeller2 {

    static Vector<String> tickets = new Vector<>();

    static {
        for (int i = 0; i < 1000; i++) {
            tickets.add("票-" + i);
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                while (tickets.size() > 0) {
                    // 将问题方法，睡1s
                    try {
                        TimeUnit.SECONDS.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("销售了：" + tickets.remove(0));
                }
            }).start();
        }
    }
}
```

实现3: 使用同步代码块锁住复合操作-保证正确性但效率低

  我们使用synchronized代码块将判断和取票操作锁在一起执行,保证其原子。

  这样可以保证售票过程的正确性,但每次取票都要锁定整个队列,效率低。

```
public class TicketSeller3 {

    static List<String> tickets = new ArrayList<>();

    static {
        for (int i = 0; i < 1000; i++) {
            tickets.add("票-" + i);
        }
    }
```

```
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                while (tickets.size() > 0) {
                    // sychronized 保证了原子性
                    synchronized (tickets) {
                        System.out.println("销售了： " + tickets.remove(0));
                    }
                }
            }).start();
        }
    }
}
```

实现4: 使用并发队列,先取票再判断

  使用JDK1.5之后提供的并发队列ConcurrentLinkedQueue存储元素,其底层用CAS实现而非加锁实现的,其效率较高。

  并发队列ConcurrentLinkedQueue的poll()方法会尝试从队列头中取出一个素,若获取不到,则返回null,对其返回值做判断可以实现先取票后判断,可以避免加锁。

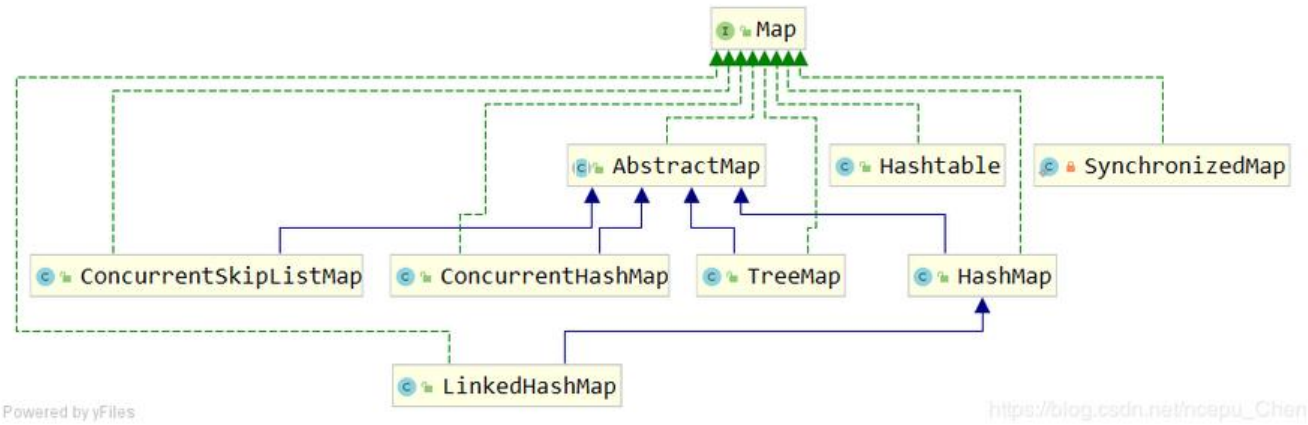
```
public class TicketSeller4 {

    static ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<>();

    static {
        for (int i = 0; i < 1000; i++) {
            queue.add("票-" + i);
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                while (true) {
                    String t = queue.poll(); // 取出头，拿不到就是空值
                    if (t == null) {
                        break;
                    }
                    System.out.println("销售了： " + t);
                }
            }).start();
        }
    }
}
```

并发容器

Map/Set

  　　Map和Set容器类型是类似的,Set无非就是屏蔽了Map的value项,只保留key

.



非并发容器

  主要的非并发容器有 HashMap,TreeMap,LinkedHashMap

并发容器

  主要的并发容器有 HashTable,SynchronizedMap,ConcurrentMap。

- HashTable和SynchronizedMap的效率较低,其同步的实现原理类似,都是给容器的所有方法都加锁.

其中SynchronizedMap使用装饰器模式,调用其构造方法并传入一个Map实现类,返回一个同步的Map器.

- ConcurrentMap的效率较高,有两个实现类:
  - ConcurrentHashMap: 使用哈希表实现,key是无序的
  - ConcurrentSkipListMap: 使用跳表实现,key是有序的

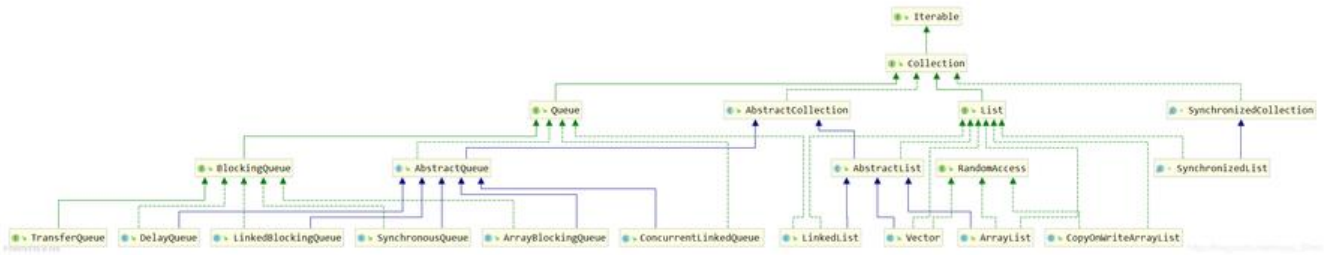
其同步的实现原理在JDK1.8前后不同

- 在JDK1.8以前,其实现同步使用的是分段锁,将整个容器分为16段(Segment),每次操作只锁住操的那一段,是一种细粒度更高的锁.
- 在JDK1.8及以后,其实现同步用的是Node+CAS.关于CAS的实现,可以看这篇文章 CAS乐观锁

```
public class ConcurrentMap {

    public static void main(String[] args) {

        //Map<String, String> map = new HashMap<>();
        //Map<String, String> map = new Hashtable<>(); // 423  每次加锁，都锁一个对象
        //Map<String, String> map = new ConcurrentHashMap<>(); // 309，加的是分段所，将器分为16段，每段都有一个锁 segment; 1.8以后 使用 Node + synchronized+CAS
        Map<String, String> map = new ConcurrentSkipListMap<>(); // 317  并发且排序，插入率较低，但是读取很快

        Random r = new Random();
        Thread[] ths = new Thread[100];
```

```
        CountDownLatch latch = new CountDownLatch(ths.length); // 启动了一个门闩，每有一个程退出，门闩就减1，直到所有线程结束，门闩打开，主线程结束

        long start = System.currentTimeMillis();
        // 创建100个线程，每个线程添加10000个元素到map，并启动这些线程
        for (int i = 0; i < ths.length; i++) {
            ths[i] = new Thread(() -> {
                for (int j = 0; j < 10000; j++) {
                    map.put("a" + r.nextInt(10000), "a" + r.nextInt(100000));
                }
                latch.countDown();
            }, "t" + i);
        }
        Arrays.asList(ths).forEach(Thread::start);

        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        long end = System.currentTimeMillis();
        System.out.println(end - start);
        System.out.println(map.size());
    }

}
```

队列



低并发队列

  低并发队列有: Vector和SynchronizedList,其中vector类似ashTable,是JDK1.2就提供的类;SynchronizedList类似SynchronizedMap使用装饰器模式,其构造函接受一个List实现类并返回同步List,在java.util.Collections包下。

  它们实现同步的原理都是将所有方法用同步代码块包裹起来。

```
public class SynchronizedList {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        // 返回的实例，每个方法都加了一个互斥锁
        List<String> syncList = Collections.synchronizedList(list);
    }
```

```
}
```

写时复制CopyOnWriteList

   CopyOnWriteArrayList位于java.util.concurrent包下,它实现同步的方式是:当发生写操作(添加,删除,修改)时,就会复制原有容器然后对新复制出的容器进行写操作,操作完成后将用指向新的容器.其写效率非常低,读效率非常高

- 优点: 读写分离,使得读操作不需要加锁,效率极高。
- 缺点: 写操作效率极低
- 应用场合: 应用在读少写多的情况,如事件监听器

```
public class CopyOnWriteList {

    public static void main(String[] args) {

        List<String> list =
                //new ArrayList<>(); //会出现并发问题
                //new Vector<>();
        new CopyOnWriteArrayList<>();  // 写速极慢，读取快

        Random r = new Random();
        Thread[] ths = new Thread[100];

        for (int i = 0; i < ths.length; i++) {
            Runnable task = () -> {
                for (int j = 0; j < 1000; j++) {
                    list.add("a" + r.nextInt(100));
                }
            };
            ths[i] = new Thread(task);

        }
        runAndComputeTime(ths);
        System.out.println(list.size());

    }

    static void runAndComputeTime(Thread[] ths) {
        long start = System.currentTimeMillis();
        Arrays.asList(ths).forEach(Thread::start);
        Arrays.asList(ths).forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        long end = System.currentTimeMillis();
        System.out.println(end - start);
    }
```

}

高并发队列

| 方法 | 抛出异常 | 返回特殊值| 一直阻塞(非阻塞队列不可用) | 阻塞一段时间(非阻塞队列不可用) |

| --- | --- | --- |

|插入元素|add(element)|offer(element)|put(element)|offer(element,time,unit)|

|移除首个元素|remove()|poll()|take()|poll(time,unit)|

|返回首个元素|element()|peek()|不可用|不可用|

对于高并发队列,若使用不同的方法对空队列执行查询和删除,以及对满队列执行插入,会产生不同行为。

● 抛出异常: 使用add(),remove(),element()方法,若执行错误操作会直接抛出异常

● 返回特殊值: 若使用offer(),poll(),peek()方法执行错误操作会返回false或null,并放弃当前错误操作,抛出异常.

● 一直阻塞: 若使用put(),take()方法执行错误操作,当前线程会一直阻塞直到条件允许才唤醒线程执行作.

● 阻塞一段时间: 若使用offer(),poll()方法并传入时间单位,会将当前方法阻塞一段时间,若阻塞时间结后仍不满足条件则返回false或null,并放弃当前错误操作,不抛出异常.

非阻塞队列ConcurrentLinkedQueue

  非阻塞队列使用CAS保证操作的原子性,不会因为加锁而阻塞线程.类似于oncurrentMap

```
public class T04_ConcurrentQueue {

    public static void main(String[] args) {
        Queue<String> queue = new ConcurrentLinkedQueue<>(); // LinkedQueue，无界队列

        for (int i = 0; i < 10; i++) {
            queue.offer("a" + i); // 有返回值，返回false代表没有加入成功，true 代表成功，并且此方
不会阻塞
        }

        System.out.println(queue);
        System.out.println(queue.size());

        System.out.println(queue.poll()); // 取出队头
        System.out.println(queue.size());

        System.out.println(queue.peek()); // 取出队头，但是不删除队头
        System.out.println(queue.size());

        // 双端队列 Deque 发音：  dai ke
        //Deque<String> deque = new ConcurrentLinkedDeque<>();
        //deque.addFirst();
        //deque.addLast();
        //deque.pollFirst();
        //deque.pollLast();
```

```
        //deque.peekFirst();
        //deque.peekLast();
    }

}
```

阻塞队列BlockingQueue

  阻塞队列的常用实现类有 LinkedBlockingQueue,ArrayBlocingQueue,DelayedQueue,TransferQueue,SynchronousQueue。分别对应于不同的应用场景。

经典阻塞队列LinkedBlockingQueue和ArrayBlockingQueue

   LinkedBlockingQueue和ArrayBlockingQueue是阻塞队列的最常用实现类用来更容易地实现生产者/消费者模式。

```
public class LinkedBlockingQueue {

    public static void main(String[] args) {

        BlockingQueue<String> queue = new LinkedBlockingQueue<>();

        // 启动生产者线程生产
        new Thread(() -> {
            for (int j = 0; j < 100; j++) {
                try {
                    queue.put("aaa" + j); // put 方法，给容器添加元素，如果容器已经满了，则会阻塞等

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "p").start();

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

        // 启用消费者线程消费
        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                while (true) {
                    try {
                        System.out.println(Thread.currentThread().getName() + ":" + queue.take()); //
队列中拿数据，如果空了，则会阻塞等待
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }, "c" + i).start();
        }
```

```
    }

}

public class ArrayBlockingQueue {

    public static void main(String[] args) throws InterruptedException {

        BlockingQueue queue = new ArrayBlockingQueue<>(10);

        for (int i = 0; i < 10; i++) {
            queue.put("a" + i);
        }

        //queue.put("a11"); // 会阻塞
        //queue.add("a11"); // 会抛出异常
        //System.out.println(queue.offer("a11")); // 会返回false
        System.out.println(queue.offer("a11", 1, TimeUnit.SECONDS)); // 会等待1s,返回false, 如果1
内有空闲,则添加成功后返回true

    }

}
```

延迟队列DelayedQueue

  延迟队列DelayedQueue中存储的元素必须实现Delay接口,其中定义了getDe
ay()方法;而Delay接口继承自Comparable接口,其中定义了compareTo()方法.各方法作用如下:

- getDelay(): 规定当前元素的延时,Delay类型的元素必须要等到其延时过期后才能从容器中取出,提取会取不到.
- compareTo(): 规定元素在容器中的排列顺序,按照compareTo()的结果升序排列。

Delayqueue可以用来执行定时任务。

```
public class DelayQueue {

    public static void main(String[] args) throws InterruptedException {
        long timestamp = System.currentTimeMillis();
        MyTask myTask1 = new MyTask(timestamp + 1000); // 1s后执行
        MyTask myTask2 = new MyTask(timestamp + 2000);
        MyTask myTask3 = new MyTask(timestamp + 1500);
        MyTask myTask4 = new MyTask(timestamp + 2500);
        MyTask myTask5 = new MyTask(timestamp + 500);

        DelayQueue<MyTask> tasks = new DelayQueue<>();
        tasks.put(myTask1);
        tasks.put(myTask2);
        tasks.put(myTask3);
        tasks.put(myTask4);
        tasks.put(myTask5);

        System.out.println(tasks);  // 确实按照我们排的顺序执行的
```

```
        while (!tasks.isEmpty()){
            System.out.println(tasks.take());
        }
    }

    static class MyTask implements Delayed {
        private long runningTime;

        public MyTask(long runTime) {
            this.runningTime = runTime;
        }

        // 这是每个元素的等待时间, 越是后加入的元素,时间等待的越长
        @Override
        public long getDelay(TimeUnit unit) {
            return unit.convert(runningTime - System.currentTimeMillis(), TimeUnit.MILLISECONDS
;
        }

        // 这是排序规律, 执行等待时间最短的排在上面
        @Override
        public int compareTo(Delayed o) {
            return (int) (o.getDelay(TimeUnit.MILLISECONDS) - this.getDelay(TimeUnit.MILLISECO
DS));
        }

        @Override
        public String toString() {
            return runningTime + "";
        }
    }

}
```

  程序输出如下,我们发现延迟队列中的元素按照 compareTo() 果升序排列,且5个元素都被阻塞式的取出。

```
[1574580515467, 1574580514967, 1574580514467, 1574580513967, 1574580513467]
1574580515467
1574580514967
1574580514467
1574580513967
1574580513467
```

阻塞消费队列TransferQueue

   TransferQueue继承自BlockingQueue,向其中添加元素的方法除了BlockingQueue的add(),offer(),put()之外,还有一个transfer()方法,该方法会使当前线程阻塞直到消费者将该线消费为止。

   transfer()与put()的区别: put()方法会阻塞直到元素成功添加进队列,transfer()法会阻塞直到元素成功被消费。

  TransferQueue特有的方法如下:

- transfer(E): 阻塞当前线程直到元素E成功被消费者消费。
- tryTransfer(E): 尝试将当前元素送给消费者线程消费,若没有消费者接受则返回false且放弃元素E,不其放入容器中。
- tryTransfer(E,long,TimeUnit): 阻塞一段时间等待消费者线程消费,超时则返回false且放弃元素E,不其放入容器中。
- hasWaitingConsumer(): 指示是否有阻塞在当前容器上的消费者线程。
- getWaitingConsumerCount(): 返回阻塞在当前容器上的消费者线程的个数。

```
public class TransferQueue {

    public static void main(String[] args) {

        TransferQueue mq = new LinkedTransferQueue();

    // 启动消费者线程,睡五秒后再来消费
        new Thread(() -> {
            try {
        TimeUnit.SECONDS.sleep(5);
                System.out.println(mq.take());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        // 再让生产者线程生产
        try {
            mq.transfer("aaa");  // put add 都不会阻塞，会添加到容器中，只有transfer才有此种功能等待消费者直接获取），所以transfer是有容量的
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}
```

  运行程序,我们发现生产者线程会一直阻塞直到五秒后 "product2"被消费者线程消费。

零容量的阻塞消费队列SynchronousQueue

   SynchronousQueue是一种特殊的TransferQueue,特殊之处在于其容量为0。因此对其调用add(),offer()方法都会使程序发生错误(抛出异常或阻塞线程).只能对其调用put()方法,内部调用transfer()方法,将元素直接交给消费者而不存储在容器中。

```
public class T09_SynchronousQueue {

    public static void main(String[] args) throws InterruptedException {

        BlockingQueue synchronousQueue = new SynchronousQueue();
```

```
        // 启动消费者线程,睡五秒后再来消费
        new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(5);
                System.out.println(synchronousQueue.take());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        System.out.println(synchronousQueue.size()); // 输出0

        // 启动生产者线程,使用put()方法添加元素,其内部调用transfer()方法,会阻塞等待元素被消费
        new Thread(() -> {
            try {
                // synchronousQueue.add("product");    // SynchronousQueue容量为0,调用add()方
会报错
                synchronousQueue.put("product");    // put()方法内部调用transfer()方法会阻塞等待元
成功被消费
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }

}
```

  该程序的输出行为与上一程序类似,生产者线程调用 put()方后阻塞五秒直到消费者线程消费该元素。

   SynchronousQueue应用场景: 网游的玩家匹配: 若一个用户登录,相当于给务器的消息队列发送一个take()请求;若一个用户准备成功,相当于给服务器的消息队列发送一个put()求。因此若玩家登陆但未准备好 或 只有一个玩家准备好 时游戏线程都会阻塞,直到两个人都准备好了游戏线程才会被唤醒,游戏继续。