目录

26 惊叹面试官：由浅入深手写队列

更新时间：2019-11-04 10:16:59

> 人生的价值，并不是用时间，而是用深度去衡量的。
>
> ——列夫·托尔斯泰

引导语

现在不少大厂面试的时候会要求手写代码，我曾经看过一个大厂面试时，要求在线写代码，题目就是：在不使用 Java 现有队列 API 的情况下，手写出一个队列的实现出来，队列的数据结构，入队和出队方式都自己定义。

这题其实考察的有几个点：

1. 考察你对队列的内部结构熟不熟悉；
2. 考察你定义 API 的功底；
3. 考察写代码的基本功，代码风格。

本章就和大家一起，结合以上几点，手写一个队列出来，一起来熟悉一下思路和过程，完整队列代码见：demo.four.DIYQueue 和 demo.four.DIYQueueDemo

1 接口定义

在实现队列之前，我们首先需要定义出队列的接口，就是我们常说的 API，API 是我们队列的门面，定义时主要原则就是简单和好用。

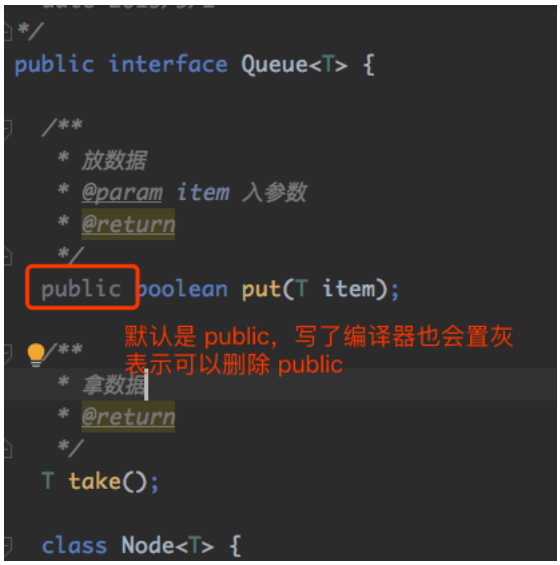
我们这次实现的队列只定义放数据和拿数据两个功能，接口定义如下：

```
/**
* 定义队列的接口，定义泛型，可以让使用者放任意类型到队列中去
* author  wenhe
* date 2019/9/1
```

```
  /**
   * 放数据
   * @param item 入参
   * @return true 成功、false 失败
   */
  boolean put(T item);

  /**
   * 拿数据，返回一个泛型值
   * @return
   */
  T take();

  // 队列中元素的基本结构
  class Node<T> {
    // 数据本身
    T item;
    // 下一个元素
    Node<T> next;

    // 构造器
    public Node(T item) {
      this.item = item;
    }
  }
}
```

有几点我们说明下：

1. 定义接口时，一定要写注释，接口的注释，方法的注释等等，这样别人看我们的接口时，会轻松很多‘；
2. 定义接口时，要求命名简洁明了，最好让别人一看见命名就知道这个接口是干啥的，比如我们命名为 Queue，别人一看就清楚这个接口是和队列相关的；
3. 用好泛型，因为我们不清楚放进队列中的到底都是那些值，所以我们使用了泛型 T，表示可以在队列中放任何值；
4. 接口里面无需给方法写上 public 方法，因为接口中的方法默认都是 public 的，你写上编译器也会置灰，如下图：



5. 我们在接口中定义了基础的元素 Node，这样队列子类如果想用的话，可以直接使用，增加了复用的可能性。

2 队列子类实现

接着我们就要开始写子类实现了，我们准备写个最常用的链表数据结构的队列。

2.1 数据结构

底层数据结构我们采用链表，一说到链表，大家应该马上就会想到三个关键要素：链表头、链表尾和链表元素，我们也实现了，代码如下：

```
/**
 * 队列头
 */
private volatile Node<T> head;

/**
 * 队列尾
 */
private volatile Node<T> tail;

/**
 * 自定义队列元素
 */
class DIYNode extends Node<T>{
  public DIYNode(T item) {
    super(item);
  }
}
```

除了这些元素之外，我们还有队列容器的容量大小、队列目前的使用大小、放数据锁、拿数据锁等等，代码如下：

```
/**
 * 队列的大小，使用 AtomicInteger 来保证其线程安全
 */
private AtomicInteger size = new AtomicInteger(0);
```

```
private final Integer capacity;

/**
 * 放数据锁
 */
private ReentrantLock putLock = new ReentrantLock();

/**
 * 拿数据锁
 */
private ReentrantLock takeLock = new ReentrantLock();
```

2.2 初始化

我们提供了使用默认容量（Integer 的最大值）和指定容量两种方式，代码如下：

```
/**
 * 无参数构造器，默认最大容量是 Integer.MAX_VALUE
 */
public DIYQueue() {
  capacity = Integer.MAX_VALUE;
  head = tail = new DIYNode(null);
}

/**
 * 有参数构造器，可以设定容量的大小
 * @param capacity
 */
public DIYQueue(Integer capacity) {
  // 进行边界的校验
  if(null == capacity || capacity < 0){
    throw new IllegalArgumentException();
  }
  this.capacity = capacity;
  head = tail = new DIYNode(null);
}
```

2.3 put 方法的实现

```
public boolean put(T item) {
  // 禁止空数据
  if(null == item){
    return false;
  }
  try{
    // 尝试加锁，500 毫秒未获得锁直接被打断
    boolean lockSuccess = putLock.tryLock(300, TimeUnit.MILLISECONDS);
        if(!lockSuccess){
          return false;
        }
    // 校验队列大小
    if(size.get() >= capacity){
      log.info("queue is full");
      return false;
    }
    // 追加到队尾
    tail = tail.next = new DIYNode(item);
    // 计数
    size.incrementAndGet();
```

```
    return false;
  } catch(Exception e){
    log.error("put error", e);
    return false;
  } finally {
    putLock.unlock();
  }
}
```

put 方法的实现有几点我们需要注意的是：

1. 注意 try catch finally 的节奏，catch 可以捕捉多种类型的异常，我们这里就捕捉了超时异常和未知异常，在 finally 里面一定记得要释放锁，不然锁不会自动释放的，这个一定不能用错，体现了我们代码的准确性；
2. 必要的逻辑检查还是需要的，比如入参是否为空的空指针检查，队列是否满的临界检查，这些检查代码可以体现出我们逻辑的严密性；
3. 在代码的关键地方加上日志和注释，这点也是非常重要的，我们不希望关键逻辑代码注释和日志都没有，不利于阅读代码和排查问题；
4. 注意线程安全，此处实现我们除了加锁之外，对于容量的大小（size）我们选择线程安全的计数类：AtomicInteger，来保证了线程安全；
5. 加锁的时候，我们最好不要使用永远阻塞的方法，我们一定要用带有超时时间的阻塞方法，此处我们设置的超时时间是 300 毫秒，也就是说如果 300 毫秒内还没有获得锁，put 方法直接返回 false，当然时间大小你可以根据情况进行设置；
6. 根据不同的情况设置不同的返回值，put 方法返回的是 false，在发生异常时，我们可以选择返回 false，或者直接抛出异常；
7. put 数据时追加到队尾的，所以我们只需要把新数据转化成 DIYNode，放到队列的尾部即可。

2.4 take 方法的实现

take 方法和 put 方法的实现非常类似，只不过 take 是从头部拿取数据，代码实现如下：

```
public T take() {
  // 队列是空的，返回 null
  if(size.get() == 0){
    return null;
  }
  try {
    // 拿数据我们设置的超时时间更短
    boolean lockSuccess = takeLock.tryLock(200,TimeUnit.MILLISECONDS);
        if(!lockSuccess){
            throw new RuntimeException("加锁失败");
        }
    // 把头结点的下一个元素拿出来
    Node expectHead = head.next;
    // 把头结点的值拿出来
    T result = head.item;
    // 把头结点的值置为 null，帮助 gc
    head.item = null;
    // 重新设置头结点的值
    head = (DIYNode) expectHead;
    size.decrementAndGet();
    // 返回头结点的值
    return result;
  } catch (InterruptedException e) {
```

```
  }finally {
      takeLock.unlock();
  }
  return null;
}
```

通过以上几步，我们的队列已经写完了，完整代码见：demo.four.DIYQueue。

3 测试

API 写好了，接下来我们要针对 API 写一些场景测试和单元测试，我们先写个场景测试，看看 API 能否跑通，代码如下：

```
@Slf4j
public class DIYQueueDemo {
        // 我们需要测试的队列
  private final static Queue<String> queue = new DIYQueue<>();
        // 生产者
  class Product implements Runnable{
    private final String message;

    public Product(String message) {
      this.message = message;
    }

    @Override
    public void run() {
      try {
        boolean success = queue.put(message);
        if (success) {
          log.info("put {} success", message);
          return;
        }
        log.info("put {} fail", message);
      } catch (Exception e) {
        log.info("put {} fail", message);
      }
    }
  }
        // 消费者
  class Consumer implements Runnable{
    @Override
    public void run() {
      try {
        String message = (String) queue.take();
        log.info("consumer message :{}",message);
      } catch (Exception e) {
        log.info("consumer message fail",e);
      }
    }
  }
        // 场景测试
  @Test
  public void testDIYQueue() throws InterruptedException {
    ThreadPoolExecutor executor =
        new ThreadPoolExecutor(10,10,0,TimeUnit.MILLISECONDS,
                               new LinkedBlockingQueue<>());
    for (int i = 0; i < 1000; i++) {
        // 是偶数的话，就提交一个生产者，奇数的话提交一个消费者
```

```
        }
        executor.submit(new Consumer());
    }
    Thread.sleep(10000);
}
```

代码测试的场景比较简单，从 0 开始循环到 1000，如果是偶数，就让生产者去生产数据，并放到队列中，如果是奇数，就让消费者去队列中拿数据出来进行消费，运行之后的结果如下：

```
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 914 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :894
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 916 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :896
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 918 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :898
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 920 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :900
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 922 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :902
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 924 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :904
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 926 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :906
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 928 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :908
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 930 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :910
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 932 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :912
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 934 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :914
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 936 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :916
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 938 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :918
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 940 success
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - consumer message :920
12:22:43.330 [pool-1-thread-2] INFO demo.four.DIYQueueDemo - put 942 success
```

从显示的结果来看，咱们写的 DIYQueue 没有太大的问题，当然如果想大规模的使用，还需要详细的单元测试和性能测试。

4 总结

通过本章的学习，不知道你有没有一种队列很简单的感觉，其实队列本身就很简单，没有想象的那么复杂。

只要我们懂得了队列的基本原理，清楚几种常用的数据结构，手写队列问题其实并不大，你也赶紧来试一试吧。

← 25 整体设计：队列设计思想、工作中使用场景

27 Thread 源码解析 →

精选留言 6

欢迎在这里发表留言，作者筛选后可公开显示

而且take之后如果size==0 应该让 tail =head 不然put之后 take的值一直都是null 我是按照文章代码调试得到的结果

0 回复 2020-01-10

啊穆

提到的完整代码 在哪里呀

0 回复 2019-11-20

慕桂英0056004 回复 **啊穆**

同问, 我也是碰到这个问题

回复 2019-11-22 15:39:11

文贺 回复 **啊穆**

here: https://github.com/luanqiu/java8 https://github.com/luanqiu/java8_demo

回复 2019-11-23 16:43:05

文贺 回复 **慕桂英0056004**

here: https://github.com/luanqiu/java8 https://github.com/luanqiu/java8_demo

回复 2019-11-23 16:43:08

前田慶次

老师您好，在初始化队列构造器的操作是不是少了一段头尾节点关联的代码，head.next = tail;否则在take数据的时候程序会出错。

0 回复 2019-11-01

文贺 回复 **前田慶次**

同学你好，不会的哈，head = tail = new DIYNode(null);指的是头尾初始化时指向同一个引用，put 时 tail = tail.next = new DIYNode(item); 这行代码，也会修改 head 的值，你可以尝试一下 debug，把 take 方法打一个断点，即不让消费，你会发现 head.next.next.next 都会有很多值，不是像我们看代码时好像为空的感觉。

回复 2019-11-04 10:11:51

前田慶次 回复 **文贺**

确实像老师说的那样，之前是因为take时size没有减一，造成加锁时程序报错

回复 2019-11-12 14:33:25

weixin_bailangning

老师take方法中返回数据前是不是应该有个size大小减一的操作呢，put方法有增加操作，但是没有看到什么地方有减少操作。

0 回复 2019-11-01

文贺 回复 **weixin_bailangning**

同学你说的很有道理，是应该加一下。

回复 2019-11-04 10:09:03

有fianlly去释放锁，应该和put是一样的，需要手动释放锁。 2.put的时候，应该是300毫秒之后，还没有获取锁，就会直接返回false，这里应该是作者的一个笔误。 麻烦作者看下，谢谢！

0 回复 2019-10-29

文贺 回复 **都被占用啦**

你说的很对，已经在修改ing了

回复 2019-10-31 13:13:15

慕码人6169125

感觉take方法有点不理解。head指针始终指向的是初始化时候建立的item为null的DIYNode。取出的时候result=head.item会得到null的

0 回复 2019-10-29

文贺 回复 **慕码人6169125**

不会的哈，有一个 if(size.get() == 0){return null;} 的判断

回复 2019-10-31 10:44:00

慕码人6169125 回复 **文贺**

老师，我的意思是您在构造方法中head = tail = new DIYNode(null);初始化head和tail都是指向item为null的节点。在put方法中tail = tail.next = new DIYNode(item);只是将新添加的节点连接到尾部，tail的引用变了，但是head依旧是指向null的节点。 take方法中 Node expectHead = head.next; T result = head.item; result难道不应该是expectedHead的item吗？ head.item = null; 下一步此处应该是expectedHead的item设为null head = (DIYNode) expectHead; 再将expected设为新的head

回复 2019-10-31 20:04:44

文贺 回复 **慕码人6169125**

同学你好，put 方法虽然只对 tail 进行了操作，但 head.item 不会指向 null ，你 debug 下就会发现，只要 put 能够不断成功，head.next 就会一直有值，take 为空只有一种情况：我们这个 demo 在模拟生产者和消费者的时候是多线程的，会出现生产者生产的速度比消费者慢的情况，只有这种情况下，可能 take 时是null。 take方法中 Node expectHead = head.next; T result = head.item; result难道不应该是expectedHead的item吗？：这个不是的哈，expectHead.item 不是头节点的，是头节点的next的item。 下一步此处应该是expectedHead的item设为null，这个也不是哈，expectedHead 是下次 take 时的头节点，不能置为null。

回复 2019-11-04 10:06:11

点击展开后面 5 条

千学不如一看，千看不如一练