一口气说出 6种 延时队列的实现方法，面试官也得服

Original 程序员内点事 程序员内点事 2020-05-07

五一期间原计划是写两篇文章，看一本技术类书籍，结果这五天由于自律性过于差，禁不住各种诱惑，我连电脑都没打开过，计划完美宣告失败。所以在这能看出和大佬之间的差距，人家没白没夜的更文，比你优秀的人比你更努力，难以望其项背，真是让我自愧不如。

知耻而后勇，这不逼着自己又学起来了，个人比较喜欢一些实践类的东西，既学习到知识又能让技术落地，能搞出个 `demo` 最好，本来不知道该分享什么主题，好在最近项目紧急招人中，而我有幸做了回面试官，就给大家整理分享一道面试题：“**如何实现延时队列？**”。

下边会介绍多种实现延时队列的思路，文末提供有几种实现方式的 `github` 地址。其实哪种方式都没有绝对的好与坏，只是看把它用在什么业务场景中，技术这东西没有最好的只有最合适的。

一、延时队列的应用

什么是延时队列？顾名思义：首先它要具有队列的特性，再给它附加一个延迟消费队列消息的功能，也就是说可以指定队列中的消息在哪个时间点被消费。

延时队列在项目中的应用还是比较多的，尤其像电商类平台：

1、订单成功后，在30分钟内没有支付，自动取消订单

2、外卖平台发送订餐通知，下单成功后60s给用户推送短信。

3、如果订单一直处于某一个未完结状态时，及时处理关单，并退还库存

4、淘宝新建商户一个月内还没上传商品信息，将冻结商铺等

。。。。

上边的这些场景都可以应用延时队列解决。

二、延时队列的实现

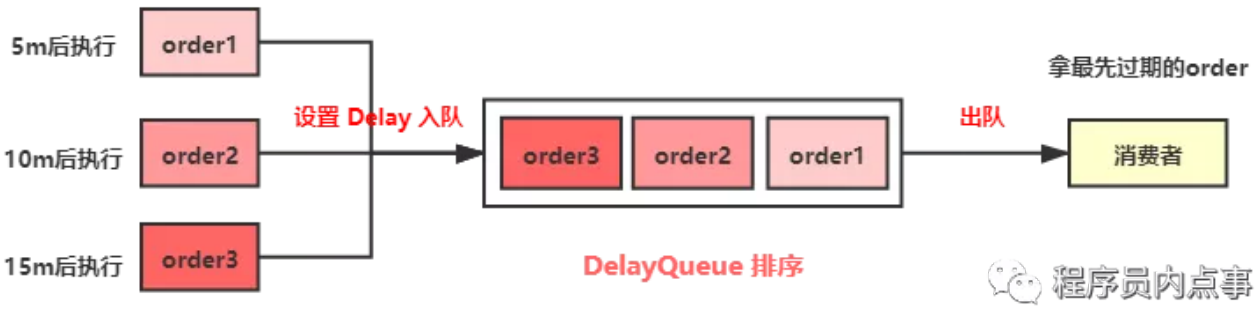
我个人一直秉承的观点：工作上能用 JDK 自带 API 实现的功能，就不要轻易自己重复造轮子，或者引入三方中间件。一方面自己封装很容易出问题（大佬除外），再加上调试验证产生许多不必要的工作量；另一方面一旦接入三方的中间件就会让系统复杂度成倍的增加，维护成本也大大的增加。

1、DelayQueue 延时队列

JDK 中提供了一组实现延迟队列的 API ，位于 Java.util.concurrent 包下 DelayQueue 。

DelayQueue 是一个 BlockingQueue （无界阻塞）队列，它本质就是封装了一个 PriorityQueue （优先队列）， PriorityQueue 内部使用 完全二叉堆 （不知道的自行了解哈）来实现队列元素排序，我们在向 DelayQueue 队列中添加元素时，会给元素一个 Delay （延迟时间）作为排序条件，队列中最小的元素会优先放在队首。队列中的元素只有到了 Delay 时间才允许从队列中取出。队列中可以放基本数据类型或自定义实体类，在存放基本数据类型时，优先队列中元素默认升序排列，自定义实体类就需要我们根据类属性值比较计算了。

先简单实现一下看看效果，添加三个 order 入队 DelayQueue ，分别设置订单在当前时间的 5秒 、 10秒 、 15秒 后取消。



要实现 DelayQueue 延时队列，队中元素要 implements Delayed 接口，这哥接口里只有一个 getDelay 方法，用于设置延期时间。 Order 类中 compareTo 方法负责对队列中的元素进行排序。

```
public class Order implements Delayed {
```

```
    /**
     * 延迟时间
     */
    @JsonFormat(locale = "zh", timezone = "GMT+8", pattern = "yyyy-MM-dd HH:mm:ss")
    private long time;
    String name;

    public Order(String name, long time, TimeUnit unit) {
        this.name = name;
        this.time = System.currentTimeMillis() + (time > 0 ? unit.toMillis(time) : 0);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return time - System.currentTimeMillis();
    }
    @Override
    public int compareTo(Delayed o) {
        Order Order = (Order) o;
        long diff = this.time - Order.time;
        if (diff <= 0) {
            return -1;
        } else {
            return 1;
        }
    }
}
```

`DelayQueue` 的 `put` 方法是线程安全的，因为 `put` 方法内部使用了 `ReentrantLock` 锁进行线程同步。 `DelayQueue` 还提供了两种出队的方法 `poll()` 和 `take()` ， `poll()` 为非阻塞获取，没有到期的元素直接返回null； `take()` 阻塞方式获取，没有到期的元素线程将会等待。

```
public class DelayQueueDemo {

    public static void main(String[] args) throws InterruptedException {
        Order Order1 = new Order("Order1", 5, TimeUnit.SECONDS);
```

```
        Order Order2 = new Order("Order2", 10, TimeUnit.SECONDS);
        Order Order3 = new Order("Order3", 15, TimeUnit.SECONDS);
        DelayQueue<Order> delayQueue = new DelayQueue<>();
        delayQueue.put(Order1);
        delayQueue.put(Order2);
        delayQueue.put(Order3);

        System.out.println("订单延迟队列开始时间:" + LocalDateTime.now().format(DateTimeFormat
        while (delayQueue.size() != 0) {
            /**
             * 取队列头部元素是否过期
             */
            Order task = delayQueue.poll();
            if (task != null) {
                System.out.format("订单:{%s}被取消, 取消时间:{%s}\n", task.name, LocalDateTime
            }
            Thread.sleep(1000);
        }
    }
}
```

上边只是简单的实现入队与出队的操作，实际开发中会有专门的线程，负责消息的入队与消费。

执行后看到结果如下， `Order1` 、 `Order2` 、 `Order3` 分别在 `5秒` 、 `10秒` 、 `15秒` 后被执行，至此就用 `DelayQueue` 实现了延时队列。

```
订单延迟队列开始时间:2020-05-06 14:59:09
订单:{Order1}被取消, 取消时间:{2020-05-06 14:59:14}
订单:{Order2}被取消, 取消时间:{2020-05-06 14:59:19}
订单:{Order3}被取消, 取消时间:{2020-05-06 14:59:24}
```

2、Quartz 定时任务

`Quartz` 一款非常经典任务调度框架，在 `Redis` 、 `RabbitMQ` 还未广泛应用时，超时未支付取消订单功能都是由定时任务实现的。定时任务它有一定的周期性，可能很多单子已经超

时，但还没到达触发执行的时间点，那么就会造成订单处理的不够及时。

引入 `quartz` 框架依赖包

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-quartz</artifactId>
</dependency>
```

在启动类中使用 `@EnableScheduling` 注解开启定时任务功能。

```
@EnableScheduling
@SpringBootApplication
public class DelayqueueApplication {
 public static void main(String[] args) {
  SpringApplication.run(DelayqueueApplication.class, args);
 }
}
```

编写一个定时任务，每个5秒执行一次。

```
@Component
public class QuartzDemo {

    //每隔五秒
    @Scheduled(cron = "0/5 * * * * ? ")
    public void process(){
        System.out.println("我是定时任务! ");
    }
}
```

3、Redis sorted set

Redis 的数据结构 Zset ，同样可以实现延迟队列的效果，主要利用它的 score 属性， redis 通过 score 来为集合中的成员进行从小到大的排序。

通过 zadd 命令向队列 delayqueue 中添加元素，并设置 score 值表示元素过期的时间；向 delayqueue 添加三个 order1 、 order2 、 order3 ，分别是 10秒 、 20秒 、 30秒 后过期。

```
zadd delayqueue 3 order3
```

消费端轮询队列 delayqueue ， 将元素排序后取最小时间与当前时间比对，如小于当前时间代表已经过期移除 key 。

```
/**
 * 消费消息
 */
public void pollOrderQueue() {

    while (true) {
        Set<Tuple> set = jedis.zrangeWithScores(DELAY_QUEUE, 0, 0);

        String value = ((Tuple) set.toArray()[0]).getElement();
        int score = (int) ((Tuple) set.toArray()[0]).getScore();

        Calendar cal = Calendar.getInstance();
        int nowSecond = (int) (cal.getTimeInMillis() / 1000);
        if (nowSecond >= score) {
            jedis.zrem(DELAY_QUEUE, value);
            System.out.println(sdf.format(new Date()) + " removed key:" + value);
        }
```

```
            if (jedis.zcard(DELAY_QUEUE) <= 0) {
                System.out.println(sdf.format(new Date()) + " zset empty ");
                return;
            }
            Thread.sleep(1000);
        }
    }
```

我们看到执行结果符合预期

```
2020-05-07 13:24:09 add finished.
2020-05-07 13:24:19 removed key:order1
2020-05-07 13:24:29 removed key:order2
2020-05-07 13:24:39 removed key:order3
2020-05-07 13:24:39 zset empty
```

4、Redis 过期回调

`Redis` 的 `key` 过期回调事件，也能达到延迟队列的效果，简单来说我们开启监听key是否过期的事件，一旦key过期会触发一个callback事件。

修改 `redis.conf` 文件开启 `notify-keyspace-events Ex`

```
notify-keyspace-events Ex
```

`Redis` 监听配置，注入Bean `RedisMessageListenerContainer`

```
@Configuration
public class RedisListenerConfig {
    @Bean
    RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory) {
```

```
        RedisMessageListenerContainer container = new RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        return container;
    }
}
```

编写Redis过期回调监听方法，必须继承 `KeyExpirationEventMessageListener` ，有点类似于MQ的消息监听。

```
@Component
public class RedisKeyExpirationListener extends KeyExpirationEventMessageListener {

    public RedisKeyExpirationListener(RedisMessageListenerContainer listenerContainer) {
        super(listenerContainer);
    }
    @Override
    public void onMessage(Message message, byte[] pattern) {
        String expiredKey = message.toString();
        System.out.println("监听到key: " + expiredKey + "已过期");
    }
}
```

到这代码就编写完成，非常的简单，接下来测试一下效果，在 `redis-cli` 客户端添加一个 `key` 并给定 `3s` 的过期时间。

```
 set xiaofu 123 ex 3
```

在控制台成功监听到了这个过期的 `key` 。

```
监听到过期的key为: xiaofu
```

5、RabbitMQ 延时队列

利用 RabbitMQ 做延时队列是比较常见的一种方式，而实际上 RabbitMQ 自身并没有直接支持提供延迟队列功能，而是通过 RabbitMQ 消息队列的 TTL 和 DXL 这两个属性间接实现的。

先来认识一下 TTL 和 DXL 两个概念：

Time To Live（TTL）：

TTL 顾名思义：指的是消息的存活时间，RabbitMQ 可以通过 x-message-tt 参数来设置指定 Queue （队列）和 Message （消息）上消息的存活时间，它的值是一个非负整数，单位为微秒。

RabbitMQ 可以从两种维度设置消息过期时间，分别是 队列 和 消息本身

- 设置队列过期时间，那么队列中所有消息都具有相同的过期时间。
- 设置消息过期时间，对队列中的某一条消息设置过期时间，每条消息 TTL 都可以不同。

如果同时设置队列和队列中消息的 TTL ，则 TTL 值以两者中较小的值为准。而队列中的消息存在队列中的时间，一旦超过 TTL 过期时间则成为 Dead Letter （死信）。

Dead Letter Exchanges （DLX）

DLX 即死信交换机，绑定在死信交换机上的即死信队列。RabbitMQ 的 Queue （队列）可以配置两个参数 x-dead-letter-exchange 和 x-dead-letter-routing-key （可选），一旦队列内出现了 Dead Letter （死信），则按照这两个参数可以将消息重新路由到另一个 Exchange （交换机），让消息重新被消费。

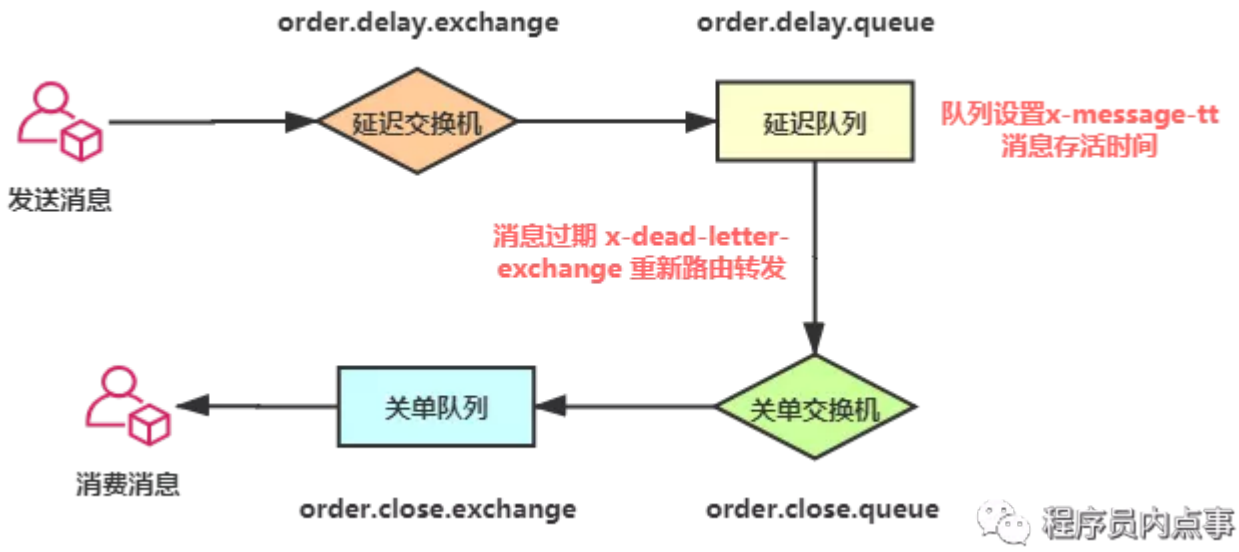
x-dead-letter-exchange：队列中出现 Dead Letter 后将 Dead Letter 重新路由转发到指定 exchange （交换机）。

x-dead-letter-routing-key：指定 routing-key 发送，一般为要指定转发的队列。

队列出现 Dead Letter 的情况有：

- 消息或者队列的 `TTL` 过期
- 队列达到最大长度
- 消息被消费端拒绝（basic.reject or basic.nack）

下边结合一张图看看如何实现超30分钟未支付关单功能，我们将订单消息A0001发送到延迟队列 `order.delay.queue` ，并设置 `x-message-tt` 消息存活时间为30分钟，当到达30分钟后订单消息A0001成为了 `Dead Letter` （死信），延迟队列检测到有死信，通过配置 `x-dead-letter-exchange` ，将死信重新转发到能正常消费的关单队列，直接监听关单队列处理关单逻辑即可。



发送消息时指定消息延迟的时间

```
public void send(String delayTimes) {
        amqpTemplate.convertAndSend("order.pay.exchange", "order.pay.queue","大家好我是延迟数据
            // 设置延迟毫秒值
            message.getMessageProperties().setExpiration(String.valueOf(delayTimes));
            return message;
        });
    }
}
```

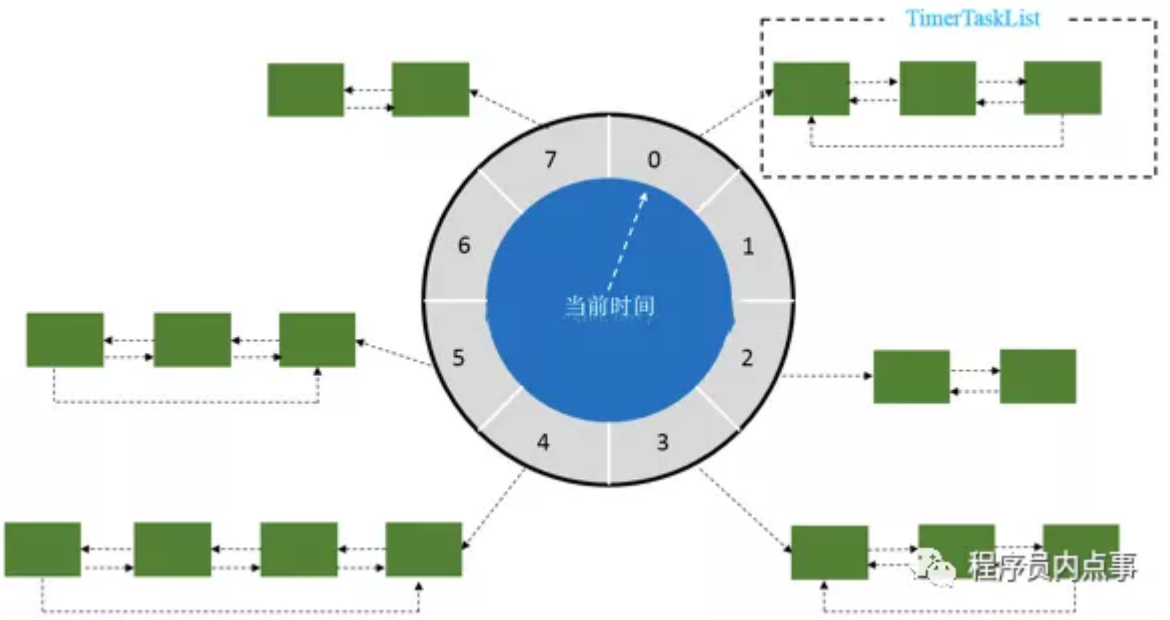
设置延迟队列出现死信后的转发规则

```
/**
     * 延时队列
     */
    @Bean(name = "order.delay.queue")
    public Queue getMessageQueue() {
        return QueueBuilder
                .durable(RabbitConstant.DEAD_LETTER_QUEUE)
                // 配置到期后转发的交换
                .withArgument("x-dead-letter-exchange", "order.close.exchange")
                // 配置到期后转发的路由键
                .withArgument("x-dead-letter-routing-key", "order.close.queue")
                .build();
    }
```

6、时间轮

前边几种延时队列的实现方法相对简单，比较容易理解，时间轮算法就稍微有点抽象了。 `kafka` 、 `netty` 都有基于时间轮算法实现延时队列，下边主要实践 `Netty` 的延时队列讲一下时间轮是什么原理。

先来看一张时间轮的原理图，解读一下时间轮的几个基本概念



`wheel` ：时间轮，图中的圆盘可以看作是钟表的刻度。比如一圈 `round` 长度为 `24秒` ，刻度数为 `8` ，那么每一个刻度表示 `3秒` 。那么时间精度就是 `3秒` 。时间长度 / 刻度数值越

大，精度越大。

当添加一个定时、延时 任务A ，假如会延迟 25秒 后才会执行，可时间轮一圈 round 的长度才 24秒 ，那么此时会根据时间轮长度和刻度得到一个圈数 round 和对应的指针位置 index ，也是就 任务A 会绕一圈指向 0格子 上，此时时间轮会记录该任务的 round 和 index 信息。当round=0，index=0 ，指针指向 0格子 任务A 并不会执行，因为 round=0不满足要求。

所以每一个格子代表的是一些时间，比如 1秒 和 25秒 都会指向0格子上，而任务则放在每个格子对应的链表中，这点和 HashMap 的数据有些类似。

Netty 构建延时队列主要用 HashedWheelTimer ， HashedWheelTimer 底层数据结构依然是使用 DelayedQueue ，只是采用时间轮的算法来实现。

下面我们用 Netty 简单实现延时队列， HashedWheelTimer 构造函数比较多，解释一下各参数的含义。

- ThreadFactory ：表示用于生成工作线程，一般采用线程池；
- tickDuration 和 unit ：每格的时间间隔，默认100ms；
- ticksPerWheel ：一圈下来有几格，默认512，而如果传入数值的不是2的N次方，则会调整为大于等于该参数的一个2的N次方数值，有利于优化 hash 值的计算。

```
public HashedWheelTimer(ThreadFactory threadFactory, long tickDuration, TimeUnit unit, int t
        this(threadFactory, tickDuration, unit, ticksPerWheel, true);
    }
```

- TimerTask ：一个定时任务的实现接口，其中run方法包装了定时任务的逻辑。
- Timeout ：一个定时任务提交到 Timer 之后返回的句柄，通过这个句柄外部可以取消这个定时任务，并对定时任务的状态进行一些基本的判断。
- Timer ：是 HashedWheelTimer 实现的父接口，仅定义了如何提交定时任务和如何停止整个定时机制。

```
public class NettyDelayQueue {

    public static void main(String[] args) {

        final Timer timer = new HashedWheelTimer(Executors.defaultThreadFactory(), 5, TimeUr

        //定时任务
        TimerTask task1 = new TimerTask() {
            public void run(Timeout timeout) throws Exception {
                System.out.println("order1  5s 后执行 ");
                timer.newTimeout(this, 5, TimeUnit.SECONDS);//结束时候再次注册
            }
        };
        timer.newTimeout(task1, 5, TimeUnit.SECONDS);
        TimerTask task2 = new TimerTask() {
            public void run(Timeout timeout) throws Exception {
                System.out.println("order2  10s 后执行");
                timer.newTimeout(this, 10, TimeUnit.SECONDS);//结束时候再注册
            }
        };

        timer.newTimeout(task2, 10, TimeUnit.SECONDS);

        //延迟任务
        timer.newTimeout(new TimerTask() {
            public void run(Timeout timeout) throws Exception {
                System.out.println("order3  15s 后执行一次");
            }
        }, 15, TimeUnit.SECONDS);

    }
}
```

从执行的结果看， `order3` 、 `order3` 延时任务只执行了一次，而 `order2` 、 `order1` 为定时任务，按照不同的周期重复执行。

```
order1  5s 后执行
order2  10s 后执行
order3  15s 后执行一次
```

```
order1  5s 后执行
order2  10s 后执行
```

总结

为了让大家更容易理解，上边的代码写的都比较简单粗糙，几种实现方式的 `demo` 已经都提交到 `github` 地址： `https://github.com/chengxy-nds/delayqueue` ，感兴趣的小伙伴可以下载跑一跑。

这篇文章肝了挺长时间，写作一点也不比上班干活轻松，查证资料反复验证demo的可行性，搭建各种 `RabbitMQ` 、 `Redis` 环境，只想说我太难了！

可能写的有不够完善的地方，如哪里有错误或者不明了的，欢迎大家踊跃指正！！！

最后

原创不易，码字不易，点个再看吧~

往期精彩回顾

我司用了 6 年的 Redis 分布式限流器，可以说是非常厉害了！

一口气说出 9种 分布式ID生成方式，面试官有点懵了

你的简历写了“熟悉” zookeeper ？那这些你会吗？

redis 分布式锁的 5个坑，真是又大又深

基于 Java 实现的人脸识别功能（附源码）

一口气说出 6种 @Transactional 注解失效场景

一口气说出 4种“附近的人”实现方式，面试官笑了

关注，迈开成长的第一步

点在看！我就承认你比我帅

People who liked this content also liked

Redis缓存那点破事 | 绝杀面试官 25 问!

微观技术

MYSQL 那点破事！索引、SQL调优、事务、B+树、分表 ....

微观技术

不费脑子学习MySQL体系架构，yyds！！

冰河技术