二

30 完整案例：实现延迟队列的两种方法

延迟队列是指把当前要做的事情，往后推迟一段时间再做。

延迟队列在实际工作中和面试中都比较常见，它的实现方式有很多种，然而每种实现方式也都有它的优缺点，接下来我们来看。

延迟队列的使用场景

延迟队列的常见使用场景有以下几种：

1. 超过 30 分钟未支付的订单，将会被取消
2. 外卖商家超过 5 分钟未接单的订单，将会被取消
3. 在平台注册但 30 天内未登录的用户，发短信提醒

等类似的应用场景，都可以使用延迟队列来实现。

常见实现方式

Redis 延迟队列实现的思路、优点：目前市面上延迟队列的实现方式基本分为三类，第一类是通过程序的方式实现，例如 JDK 自带的延迟队列 DelayQueue，第二类是通过 MQ 框架来实现，例如 RabbitMQ 可以通过 rabbitmq-delayed-message-exchange 插件来实现延迟队列，第三类就是通过 Redis 的方式来实现延迟队列。

程序实现方式

JDK 自带的 DelayQueue 实现延迟队列，代码如下：

```
public class DelayTest {
    public static void main(String[] args) throws InterruptedException {
        DelayQueue delayQueue = new DelayQueue();
        delayQueue.put(new DelayElement(1000));
        delayQueue.put(new DelayElement(3000));
        delayQueue.put(new DelayElement(5000));
```

```
        System.out.println("开始时间：" +  DateFormat.getDateTimeInstance().format(n
        while (!delayQueue.isEmpty()){
            System.out.println(delayQueue.take());
        }
        System.out.println("结束时间：" +  DateFormat.getDateTimeInstance().format(n
    }

    static class DelayElement implements Delayed {
        // 延迟截止时间（单面：毫秒）
        long delayTime = System.currentTimeMillis();
        public DelayElement(long delayTime) {
            this.delayTime = (this.delayTime + delayTime);
        }
        @Override
        // 获取剩余时间
        public long getDelay(TimeUnit unit) {
            return unit.convert(delayTime - System.currentTimeMillis(), TimeUnit.MI
        }
        @Override
        // 队列里元素的排序依据
        public int compareTo(Delayed o) {
            if (this.getDelay(TimeUnit.MILLISECONDS) > o.getDelay(TimeUnit.MILLISEC
                return 1;
            } else if (this.getDelay(TimeUnit.MILLISECONDS) < o.getDelay(TimeUnit.M
                return -1;
            } else {
                return 0;
            }
        }
        @Override
        public String toString() {
            return DateFormat.getDateTimeInstance().format(new Date(delayTime));
        }
    }
}
```

程序执行结果：

```
开始时间：2019-6-13 20:40:38
2019-6-13 20:40:39
2019-6-13 20:40:41
2019-6-13 20:40:43
结束时间：2019-6-13 20:40:43
```

优点

1. 开发比较方便，可以直接在代码中使用
2. 代码实现比较简单

缺点

1. 不支持持久化保存
2. 不支持分布式系统

MQ 实现方式

RabbitMQ 本身并不支持延迟队列，但可以通过添加插件 rabbitmq-delayed-message-exchange 来实现延迟队列。

优点

1. 支持分布式
2. 支持持久化

缺点

框架比较重，需要搭建和配置 MQ。

Redis 实现方式

Redis 是通过有序集合（ZSet）的方式来实现延迟消息队列的，ZSet 有一个 Score 属性可以用来存储延迟执行的时间。

优点

1. 灵活方便，Redis 是互联网公司的标配，无序额外搭建相关环境；
2. 可进行消息持久化，大大提高了延迟队列的可靠性；
3. 分布式支持，不像 JDK 自身的 DelayQueue；
4. 高可用性，利用 Redis 本身高可用方案，增加了系统健壮性。

缺点

需要使用无限循环的方式来执行任务检查，会消耗少量的系统资源。

结合以上优缺点，我们决定使用 Redis 来实现延迟队列，具体实现代码如下。

代码实战

本文我们使用 Java 语言来实现延迟队列，延迟队列的实现有两种方式：第一种是利用 zrangebyscore 查询符合条件的所有待处理任务，循环执行队列任务。第二种实现方式是每次查询最早的一条消息，判断这条信息的执行时间是否小于等于此刻的时间，如果是则执行此任务，否则继续循环检测。

方式一

一次性查询所有满足条件的任务，循环执行，代码如下：

```
import redis.clients.jedis.Jedis;
import utils.JedisUtils;

import java.time.Instant;
import java.util.Set;

/**
 * 延迟队列
 */
public class DelayQueueExample {
    // zset key
    private static final String _KEY = "myDelayQueue";

    public static void main(String[] args) throws InterruptedException {
        Jedis jedis = JedisUtils.getJedis();
        // 延迟 30s 执行（30s 后的时间）
        long delayTime = Instant.now().plusSeconds(30).getEpochSecond();
        jedis.zadd(_KEY, delayTime, "order_1");
        // 继续添加测试数据
        jedis.zadd(_KEY, Instant.now().plusSeconds(2).getEpochSecond(), "order_2");
        jedis.zadd(_KEY, Instant.now().plusSeconds(2).getEpochSecond(), "order_3");
        jedis.zadd(_KEY, Instant.now().plusSeconds(7).getEpochSecond(), "order_4");
        jedis.zadd(_KEY, Instant.now().plusSeconds(10).getEpochSecond(), "order_5")
        // 开启延迟队列
        doDelayQueue(jedis);
    }

    /**
     * 延迟队列消费
     * @param jedis Redis 客户端
     */
    public static void doDelayQueue(Jedis jedis) throws InterruptedException {
        while (true) {
            // 当前时间
            Instant nowInstant = Instant.now();
            long lastSecond = nowInstant.plusSeconds(-1).getEpochSecond(); // 上一秒
            long nowSecond = nowInstant.getEpochSecond();
            // 查询当前时间的所有任务
            Set<String> data = jedis.zrangeByScore(_KEY, lastSecond, nowSecond);
            for (String item : data) {
                // 消费任务
                System.out.println("消费: " + item);
            }
```

```
            // 删除已经执行的任务
            jedis.zremrangeByScore(_KEY, lastSecond, nowSecond);
            Thread.sleep(1000); // 每秒轮询一次
        }
    }
}
```

以上程序执行结果如下：

```
消费：order_2
消费：order_3
消费：order_4
消费：order_5
消费：order_1
```

方式二

每次查询最早的一条任务，与当前时间判断，决定是否需要执行，实现代码如下：

```
import redis.clients.jedis.Jedis;
import utils.JedisUtils;

import java.time.Instant;
import java.util.Set;

/**
 * 延迟队列
 */
public class DelayQueueExample {
    // zset key
    private static final String _KEY = "myDelayQueue";

    public static void main(String[] args) throws InterruptedException {
        Jedis jedis = JedisUtils.getJedis();
        // 延迟 30s 执行（30s 后的时间）
        long delayTime = Instant.now().plusSeconds(30).getEpochSecond();
        jedis.zadd(_KEY, delayTime, "order_1");
        // 继续添加测试数据
        jedis.zadd(_KEY, Instant.now().plusSeconds(2).getEpochSecond(), "order_2");
        jedis.zadd(_KEY, Instant.now().plusSeconds(2).getEpochSecond(), "order_3");
        jedis.zadd(_KEY, Instant.now().plusSeconds(7).getEpochSecond(), "order_4");
        jedis.zadd(_KEY, Instant.now().plusSeconds(10).getEpochSecond(), "order_5")
        // 开启延迟队列
        doDelayQueue2(jedis);
    }

    /**
     * 延迟队列消费（方式2）
     * @param jedis Redis 客户端
```

```
     */
    public static void doDelayQueue2(Jedis jedis) throws InterruptedException {
        while (true) {
            // 当前时间
            long nowSecond = Instant.now().getEpochSecond();
            // 每次查询一条消息，判断此消息的执行时间
            Set<String> data = jedis.zrange(_KEY, 0, 0);
            if (data.size() == 1) {
                String firstValue = data.iterator().next();
                // 消息执行时间
                Double score = jedis.zscore(_KEY, firstValue);
                if (nowSecond >= score) {
                    // 消费消息（业务功能处理）
                    System.out.println("消费消息：" + firstValue);
                    // 删除已经执行的任务
                    jedis.zrem(_KEY, firstValue);
                }
            }
            Thread.sleep(100); // 执行间隔
        }
    }
}
```

以上程序执行结果和实现方式一相同，结果如下：

```
消费：order_2
消费：order_3
消费：order_4
消费：order_5
消费：order_1
```

其中，执行间隔代码 `Thread.sleep(100)` 可根据实际的业务情况删减或配置。

小结

本文我们介绍了延迟队列的使用场景以及各种实现方案，其中 Redis 的方式是最符合我们需求的，它主要是利用有序集合的 score 属性来存储延迟执行时间，再开启一个无限循环来判断是否有符合要求的任务，如果有的话执行相关逻辑，没有的话继续循环检测。

上一页 下一页