引言

在开发中，往往会遇到一些关于延时任务的需求。例如
生成订单 30 分钟未支付，则自动取消
生成订单 60 秒后,给用户发短信
对上述的任务，我们给一个专业的名字来形容，那就是延时任务。那么这里就会产生一个问题，这个延时任务和定时任务的区别究竟在哪里呢？一共有如下几点区别
定时任务有明确的触发时间，延时任务没有
定时任务有执行周期，而延时任务在某事件触发后一段时间内执行，没有执行周期
定时任务一般执行的是批处理操作是多个任务，而延时任务一般是单个任务
下面，我们以判断订单是否超时为例，进行方案分析

方案分析

(1)数据库轮询

思路
该方案通常是在小型项目中使用，即通过一个线程定时的去扫描数据库，通过订单时间来判断是否有超时的订单，然后进行 update 或 delete 等操作
实现
博主当年早期是用 quartz 来实现的(实习那会的事)，简单介绍一下
maven 项目引入一个依赖如下所示

```
<dependency>
    <groupId>org.quartz-scheduler</groupId>
        <artifactId>quartz</artifactId>
    <version>2.2.2</version>
</dependency>
```

调用 Demo 类 MyJob 如下所示

```
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
```

```
public class MyJob implements Job {
    public void execute(JobExecutionContext context)
            throws JobExecutionException {
        System.out.println("要去数据库扫描啦。。。");
    }

    public static void main(String[] args) throws Exception {
        // 创建任务
        JobDetail jobDetail = JobBuilder.newJob(MyJob.class)
                .withIdentity("job1", "group1").build();
        // 创建触发器 每 3 秒钟执行一次
        Trigger trigger = TriggerBuilder
                .newTrigger()
                .withIdentity("trigger1", "group3")
                .withSchedule(
                        SimpleScheduleBuilder.simpleSchedule()
                                .withIntervalInSeconds(3).repeatForever())
                .build();
        Scheduler scheduler = new StdSchedulerFactory().getScheduler();
        // 将任务及其触发器放入调度器
        scheduler.scheduleJob(jobDetail, trigger);
        // 调度器开始调度任务
        scheduler.start();
    }
}
```

优缺点

优点:简单易行，支持集群操作

缺点:

(1)对服务器内存消耗大

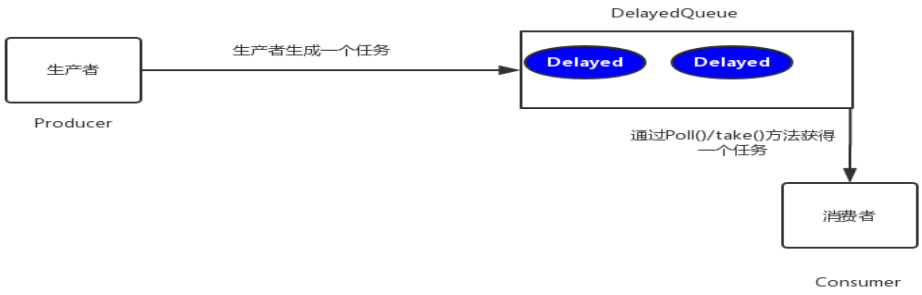
(2)存在延迟，比如你每隔 3 分钟扫描一次，那最坏的延迟时间就是 3 分钟

(3)假设你的订单有几千万条，每隔几分钟这样扫描一次，数据库损耗极大

(2) JDK 的延迟队列

思路

该方案是利用 JDK 自带的 DelayQueue 来实现，这是一个无界阻塞队列，该队列只有在延迟期满的时候才能从中获取元素，放入 DelayQueue 中的对象，是必须实现 Delayed 接口的。DelayedQueue 实现工作流程如下图所示



其中 Poll():获取并移除队列的超时元素，没有则返回空
take():获取并移除队列的超时元素，如果没有则 wait 当前线程，直到有元素满足超时条件，返回结果。
实现
定义一个类 OrderDelay 实现 Delayed，代码如下

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class OrderDelay implements Delayed {

    private String orderId;
    private long timeout;

    OrderDelay(String orderId, long timeout) {
        this.orderId = orderId;
        this.timeout = timeout + System.nanoTime();
    }

    public int compareTo(Delayed other) {
        if (other == this)
            return 0;
        OrderDelay t = (OrderDelay) other;
        long d = (getDelay(TimeUnit.NANOSECONDS) - t
                .getDelay(TimeUnit.NANOSECONDS));
        return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
    }

    // 返回距离你自定义的超时时间还有多少
    public long getDelay(TimeUnit unit) {
```

```
            return unit.convert(timeout - System.nanoTime(), TimeUnit.NANOSECONDS);
        }

        void print() {
            System.out.println(orderId+"编号的订单要删除啦。。。。");
        }
}
```

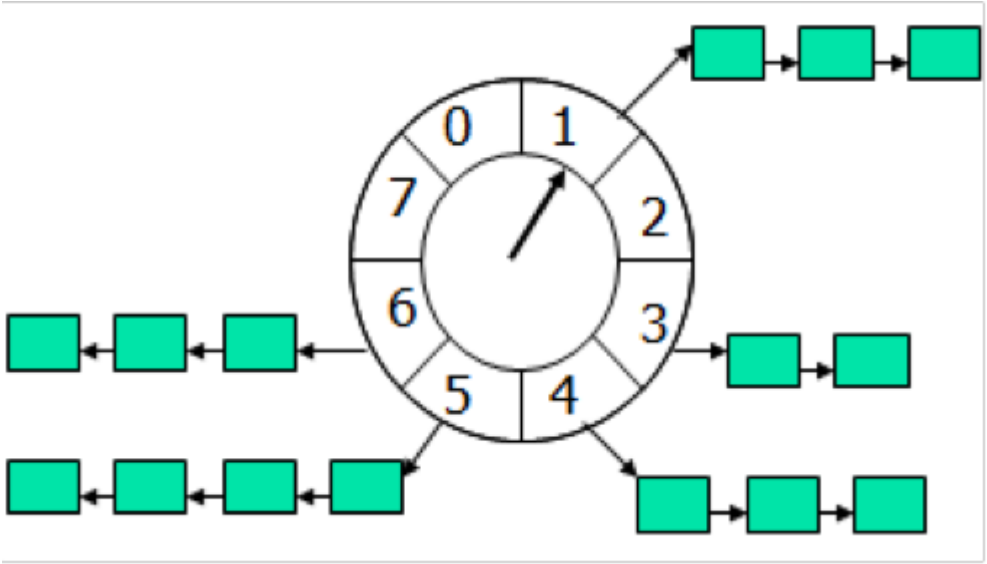
运行的测试 Demo 为，我们设定延迟时间为 3 秒

```
package com.rjzheng.delay2;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.TimeUnit;

public class DelayQueueDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<String> list = new ArrayList<String>();
        list.add("00000001");
        list.add("00000002");
        list.add("00000003");
        list.add("00000004");
        list.add("00000005");
        DelayQueue<OrderDelay> queue = new DelayQueue<OrderDelay>();
        long start = System.currentTimeMillis();
        for(int i = 0;i<5;i++){
            //延迟三秒取出
            queue.put(new OrderDelay(list.get(i),
                    TimeUnit.NANOSECONDS.convert(3, TimeUnit.SECONDS)));
            try {
                queue.take().print();
                System.out.println("After " +
                        (System.currentTimeMillis()-start) + " MilliSeconds");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

}
```

可以看到都是延迟 3 秒，订单被删除
优缺点
优点:效率高,任务触发时间延迟低。
缺点:(1)服务器重启后，数据全部消失，怕宕机
(2)集群扩展相当麻烦
(3)因为内存条件限制的原因，比如下单未付款的订单数太多，那么很容易就出现 OOM 异常
(4)代码复杂度较高

(3)时间轮算法

思路
先上一张时间轮的图(这图到处都是啦)



时间轮算法可以类比于时钟，如上图箭头（指针）按某一个方向按固定频率轮动，每一次跳动称为一个 tick。这样可以看出定时轮由个 3 个重要的属性参数，ticksPerWheel（一轮的 tick 数），tickDuration（一个 tick 的持续时间）以及 timeUnit（时间单位），例如当 ticksPerWheel=60，tickDuration=1，timeUnit=秒，这就和现实中的始终的秒针走动完全类似了。
如果当前指针指在 1 上面，我有一个任务需要 4 秒以后执行，那么这个执行的线程回调或者消息将会被放在 5 上。那如果需要在 20 秒之后执行怎么办，由于这个环形结构槽数只到 8，如果要 20 秒，指针需要多转 2 圈。位置是在 2 圈之后的 5 上面（20 % 8 + 1）
实现
我们用 Netty 的 HashedWheelTimer 来实现
给 Pom 加上下面的依赖

```
<dependency>
    <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.24.Final</version>
</dependency>
```

测试代码 HashedWheelTimerTest 如下所示

```java
import io.netty.util.HashedWheelTimer;
import io.netty.util.Timeout;
import io.netty.util.Timer;
import io.netty.util.TimerTask;

import java.util.concurrent.TimeUnit;

public class HashedWheelTimerTest {
    static class MyTimerTask implements TimerTask{
        boolean flag;
        public MyTimerTask(boolean flag){
            this.flag = flag;
        }
        public void run(Timeout timeout) throws Exception {
            // TODO Auto-generated method stub
             System.out.println("要去数据库删除订单了。。。。");
             this.flag =false;
        }
    }
    public static void main(String[] argv) {
        MyTimerTask timerTask = new MyTimerTask(true);
        Timer timer = new HashedWheelTimer();
        timer.newTimeout(timerTask, 5, TimeUnit.SECONDS);
        int i = 1;
        while(timerTask.flag){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println(i+"秒过去了");
            i++;
        }
    }
}
```

优缺点

优点:效率高,任务触发时间延迟时间比 delayQueue 低，代码复杂度比 delayQueue 低。

缺点:(1)服务器重启后，数据全部消失，怕宕机

(2)集群扩展相当麻烦

(3)因为内存条件限制的原因，比如下单未付款的订单数太多，那么很容易就出现 OOM 异常

(4)redis 缓存

思路一

利用 redis 的 zset,zset 是一个有序集合，每一个元素(member)都关联了一个 score,通过 score 排序来取集合中的值

zset 常用命令

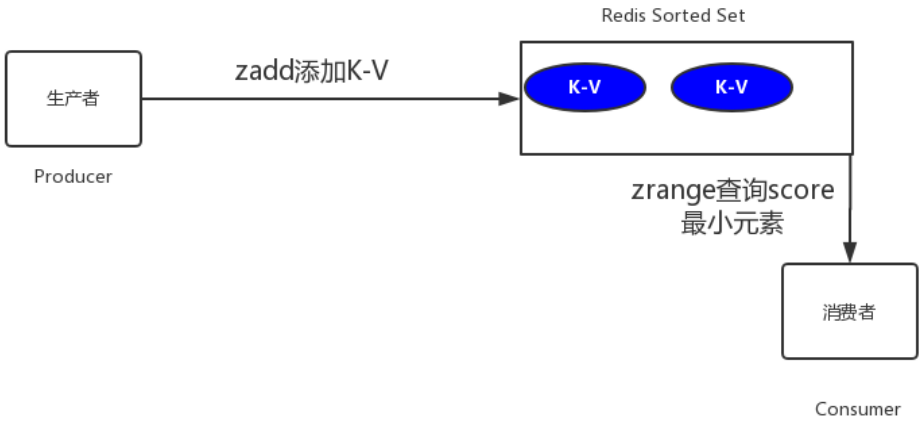
添加元素:ZADD key score member [[score member] [score member] ...]

按顺序查询元素:ZRANGE key start stop [WITHSCORES]

查询元素 score:ZSCORE key member

移除元素:ZREM key member [member ...]

那么如何实现呢？我们将订单超时时间戳与订单号分别设置为 score 和 member,系统扫描第一个元素判断是否超时，具体如下图所示




```
import java.util.Calendar;
import java.util.Set;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.Tuple;

public class AppTest {
    private static final String ADDR = "127.0.0.1";
    private static final int PORT = 6379;
    private static JedisPool jedisPool = new JedisPool(ADDR, PORT);

    public static Jedis getJedis() {
        return jedisPool.getResource();
    }
```

```
    //生产者,生成 5 个订单放进去
    public void productionDelayMessage(){
        for(int i=0;i<5;i++){
            //延迟 3 秒
            Calendar cal1 = Calendar.getInstance();
            cal1.add(Calendar.SECOND, 3);
            int second3later = (int) (cal1.getTimeInMillis() / 1000);
            AppTest.getJedis().zadd("OrderId", second3later,"OID0000001"+i);
            System.out.println(System.currentTimeMillis()+"ms:redis 生成了一个订单任务：订单 ID 为
"+"OID0000001"+i);
        }
    }

    //消费者，取订单
    public void consumerDelayMessage(){
        Jedis jedis = AppTest.getJedis();
        while(true){
            Set<Tuple> items = jedis.zrangeWithScores("OrderId", 0, 1);
            if(items == null || items.isEmpty()){
                System.out.println("当前没有等待的任务");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                continue;
            }
            int    score = (int) ((Tuple)items.toArray()[0]).getScore();
            Calendar cal = Calendar.getInstance();
            int nowSecond = (int) (cal.getTimeInMillis() / 1000);
            if(nowSecond >= score){
                String orderId = ((Tuple)items.toArray()[0]).getElement();
                jedis.zrem("OrderId", orderId);
                System.out.println(System.currentTimeMillis() +"ms:redis 消费了一个任务：消费的订单 OrderId
为"+orderId);
            }
        }
    }

    public static void main(String[] args) {
        AppTest appTest =new AppTest();
        appTest.productionDelayMessage();
```

```
            appTest.consumerDelayMessage();
        }


}
```

可以看到，几乎都是 3 秒之后，消费订单。

然而，这一版存在一个致命的硬伤，在高并发条件下，多消费者会取到同一个订单号，我们上测试代码 ThreadTest

```
import java.util.concurrent.CountDownLatch;


public class ThreadTest {
    private static final int threadNum = 10;
    private static CountDownLatch cdl = new CountDownLatch(threadNum);
    static class DelayMessage implements Runnable{
        public void run() {
            try {
                cdl.await();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            AppTest appTest =new AppTest();
            appTest.consumerDelayMessage();
        }
    }
    public static void main(String[] args) {
        AppTest appTest =new AppTest();
        appTest.productionDelayMessage();
        for(int i=0;i<threadNum;i++){
            new Thread(new DelayMessage()).start();
            cdl.countDown();
        }
    }
}
```

显然，出现了多个线程消费同一个资源的情况。

解决方案

(1)用分布式锁，但是用分布式锁，性能下降了，该方案不细说。

(2)对 ZREM 的返回值进行判断，只有大于 0 的时候，才消费数据，于是将 consumerDelayMessage()方法里的

```
if(nowSecond >= score){
    String orderId = ((Tuple)items.toArray()[0]).getElement();
    jedis.zrem("OrderId", orderId);
```

```
        System.out.println(System.currentTimeMillis() +"ms:redis 消费了一个任务：消费的订单 OrderId 为"+orderId);
    }
```

修改为

```
if(nowSecond >= score){
        String orderId = ((Tuple)items.toArray()[0]).getElement();
        Long num = jedis.zrem("OrderId", orderId);
        if( num != null && num>0){
            System.out.println(System.currentTimeMillis()+"ms:redis 消费了一个任务：消费的订单 OrderId 为
"+orderId);
        }
    }
```

在这种修改后，重新运行 ThreadTest 类，发现输出正常了

思路二

该方案使用 redis 的 Keyspace Notifications，中文翻译就是[键空间机制](), 就是利用该机制可以在 key 失效之后，提供一个回调，实际上是 redis 会给客户端发送一个消息。是需要 redis 版本 2.8 以上。

实现二

在 redis.conf 中，加入一条配置

notify-keyspace-events Ex

运行代码如下

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPubSub;

public class RedisTest {
    private static final String ADDR = "127.0.0.1";
    private static final int PORT = 6379;
    private static JedisPool jedis = new JedisPool(ADDR, PORT);
    private static RedisSub sub = new RedisSub();

    public static void init() {
        new Thread(new Runnable() {
            public void run() {
                jedis.getResource().subscribe(sub, "__keyevent@0__:expired");
            }
        }).start();
    }

    public static void main(String[] args) throws InterruptedException {
        init();
        for(int i =0;i<10;i++){
```

```
                String orderId = "OID000000"+i;
                jedis.getResource().setex(orderId, 3, orderId);
                System.out.println(System.currentTimeMillis()+"ms:"+orderId+"订单生成");
            }
        }

        static class RedisSub extends JedisPubSub {
            @Override
            public void onMessage(String channel, String message) {
                System.out.println(System.currentTimeMillis()+"ms:"+message+"订单取消");
            }
        }
}
```

可以明显看到 3 秒过后，订单取消了
ps:redis 的 pub/sub 机制存在一个硬伤，官网内容如下
原:Because Redis Pub/Sub is fire and forget currently there is no way to use this feature if your application demands reliable notification of events, that is, if your Pub/Sub client disconnects, and reconnects later, all the events delivered during the time the client was disconnected are lost.
翻: Redis 的发布/订阅目前是即发即弃(fire and forget)模式的，因此无法实现事件的可靠通知。也就是说，如果发布/订阅的客户端断链之后又重连，则在客户端断链期间的所有事件都丢失了。
因此，方案二不是太推荐。当然，如果你对可靠性要求不高，可以使用。
优缺点
优点:(1)由于使用 Redis 作为消息通道，消息都存储在 Redis 中。如果发送程序或者任务处理程序挂了，重启之后，还有重新处理数据的可能性。
(2)做集群扩展相当方便
(3)时间准确度高
缺点:(1)需要额外进行 redis 维护

(5)使用消息队列

我们可以采用 rabbitMQ 的延时队列。RabbitMQ 具有以下两个特性，可以实现延迟队列
RabbitMQ 可以针对 Queue 和 Message 设置 x-message-tt，来控制消息的生存时间，如果超时，则消息变为 dead letter
lRabbitMQ 的 Queue 可以配置 x-dead-letter-exchange 和 x-dead-letter-routing-key（可选）两个参数，用来控制队列内出现了 deadletter，则按照这两个参数重新路由。
结合以上两个特性，就可以模拟出延迟消息的功能,具体的，
优缺点
优点: 高效,可以利用 rabbitmq 的分布式特性轻易的进行横向扩展,消息支持持久化增加了可靠性。
缺点：本身的易用度要依赖于 rabbitMq 的运维.因为要引用 rabbitMq,所以复杂度和成本变高