The scheduled task has a clear trigger time

Delayed tasks are executed within a period after an event is triggered.

* Delayed tasks are generally the single task

Scenario - Use Messages

RabbitMQ

* Set **X-Message-tt** for Queue and message to control the lifetime of the message.
* If time out the message becomes a dead letter.
* **X-dead-letter-exchange** and **x-dead-letter-routing-key** are used to control the occurrence of the dead letter in the queue and re-routing according to the parameter.

Advantages of Rabbit MQ

* Distributed nature of Rabbit MQ – expand easily and message supports persistence to increase reliability.

Disadvantage

* Cost becomes higher for maintenance

Scenario 2 : DelayQueue from JDK

* Unbounded blocking queue
* This queue get elements from when delay expires.
* Object place in delayed queue must implement delay interface.
* Among them, Poll(): Get and remove the timeout element of the queue, and return empty if there is no
* take(): Obtain and remove the timeout element of the queue, if not, wait the current thread until an element meets the timeout condition, and return the result.

package com.omgzui.delay;  
​  
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);  
 }  
​  
 public long getDelay(TimeUnit unit) {  
 return unit.convert(timeout - System.nanoTime(), TimeUnit.NANOSECONDS);  
 }  
​  
 void print() {  
 System.out.println(orderId + "Order delete");  
 }  
​  
}

package com.omgzui.delay;  
​  
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) {  
 List<String> list = new ArrayList<String>();  
 list.add("00000001");  
 list.add("00000002");  
 list.add("00000003");  
 list.add("00000004");  
 list.add("00000005");  
​  
 DelayQueue<OrderDelay> queue = newDelayQueue < OrderDelay > ();  
 long start = System.currentTimeMillis();  
 for (int i = 0; i < 5; i++) {  
 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) {  
 e.printStackTrace();  
 }  
 }  
 }  
​  
}

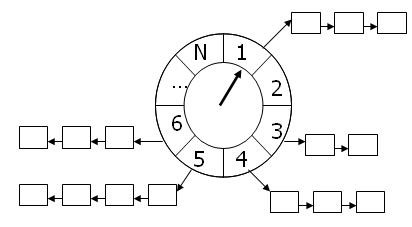
**Advantage**

* High efficiency and low task trigger time delay.

**Shortcoming**

* After the server restarts, all data disappears, fearing downtime
* Cluster expansion is quite troublesome
* Due to the limitation of memory conditions, for example, if too many orders have not been paid, OOM exceptions will easily occur.
* High code complexity

**Scenario 3: Time Wheel Algorithm**

[[](https://www.java67.com/2018/06/data-structure-and-algorithm-interview-questions-programmers.html)](https://www.java67.com/2018/06/data-structure-and-algorithm-interview-questions-programmers.html)

The time wheel algorithm can be compared to a clock, as shown in the above figure, the arrow (pointer) rotates at a fixed frequency in a certain direction, and each beating is called a tick. It can be seen that the timing wheel has three important attribute parameters, ticksPerWheel (number of ticks in a round), tickDuration (duration of a tick) and timeUnit (time unit), for example, when ticksPerWheel=60, tickDuration=1, timeUnit = second, this is completely similar to the constant movement of the second hand in reality.

If the current pointer is on 1 and I have a task that needs to be executed in 4 seconds, then the thread callback or message for this execution will be placed on 5. So what if it needs to be executed after 20 seconds, since the number of slots in this ring structure is only 8, if it takes 20 seconds, the pointer needs to rotate 2 more times. Position is above 5 after 2 laps (20 % 8 + 1)

Implemented with Netty’s HashedWheelTimer

package com.omgzui.delay;  
​  
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 {  
 System.out.println("delete");  
 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) {  
 e.printStackTrace();  
 }  
 System.out.println(i + "s");  
 i++;  
 }  
 }  
​  
}

**Advantage**

* High efficiency, task trigger time delay is lower than delayQueue, code complexity is lower than delayQueue.

**Shortcoming**

* After the server restarts, all data disappears, fearing downtime.
* Cluster expansion is quite troublesome
* Due to the limitation of memory conditions, for example, if there are too many orders that have not been paid, OOM exceptions will easily occur.

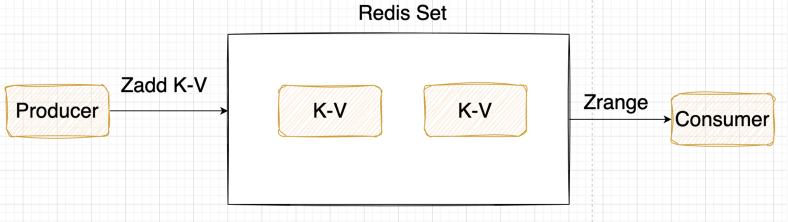
**Scenario 4: redis cache**

Using redis’s zset, zset is an ordered collection, each element (member) is associated with a score, and the value in the collection is obtained by sorting the score

* Add element: ZADD key score member [score member …]
* Query elements in order: ZRANGE key start stop [WITHSCORES]
* Query element score: ZSCORE key member
* Remove element: ZREM key member [member …]

redis> ZADD page\_rank 10 google.com  
(integer) 1  
​  
redis> ZADD page\_rank 9 medium.com 8 bing.com  
(integer) 2  
​  
redis> ZRANGE page\_rank 0 -1 WITHSCORES  
1) "bing.com"  
2) "8"  
3) "medium.com"  
4) "9"  
5) "google.com"  
6) "10"  
​  
redis> ZSCORE page\_rank bing.com  
"8"  
​  
redis> ZREM page\_rank google.com  
(integer) 1  
​  
redis> ZRANGE page\_rank 0 -1 WITHSCORES  
1) "bing.com"  
2) "8"  
3) "medium.com"  
4) "9"

We set the order timeout timestamp and order number as score and member respectively, and the system scans the first element to determine whether it is timed out



package com.omgzui.delay;  
​  
import redis.clients.jedis.Jedis;  
import redis.clients.jedis.JedisPool;  
import redis.clients.jedis.Tuple;  
​  
import java.util.Calendar;  
import java.util.Set;  
​  
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();  
 }  
​  
 public void productionDelayMessage() {  
 for (int i = 0; i < 5; i++) {  
 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 order" + "OID0000001" + i);  
 }  
 }  
  
​  
 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("wait");  
 try {  
 Thread.sleep(500);  
 } catch (InterruptedException e) {  
 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 task" + orderId);  
 }  
 }  
 }  
​  
 public static void main(String[] args) {  
 AppTest appTest = new AppTest();  
 appTest.productionDelayMessage();  
 appTest.consumerDelayMessage();  
 }  
​  
}

**Advantage**

* Since Redis is used as the message channel, the messages are all stored in Redis. If the sending program or the task handler hangs up, after restarting, there is still the possibility of reprocessing the data.
* It is quite convenient to do cluster expansion
* High time accuracy

**Shortcoming**

* Additional redis maintenance is required

**Scenario 5: Database Polling**

This solution is usually used in small projects, that is, to scan the database regularly through a thread, judge whether there is an overtime order by the order time, and then perform operations such as update or delete.

It can be implemented with quartz, calling the Demo class MyJob.

package com.omgzui.delay;  
​  
import org.quartz.\*;  
import org.quartz.impl.StdSchedulerFactory;  
​  
public class MyJob implements Job {  
​  
 public void execute(JobExecutionContext context) throws JobExecutionException {  
 System.out.println("do");  
 }  
​  
 public static void main(String[] args) throws Exception {  
 JobDetail jobDetail = JobBuilder.newJob(MyJob.class)  
 .withIdentity("job1", "group1").build();  
 Trigger trigger = TriggerBuilder  
 .newTrigger()  
 .withIdentity("trigger1", "group3")  
 .withSchedule(  
 SimpleScheduleBuilder  
 .simpleSchedule()  
 .withIntervalInSeconds(3).  
 repeatForever())  
 .build();  
 Scheduler scheduler = new StdSchedulerFactory().getScheduler();  
 scheduler.scheduleJob(jobDetail, trigger);  
 scheduler.start();  
 }  
​  
}