链滴

基于 Redis 实现 DelayQueue 延迟队列设计方案

作者：shirenchuang

原文链接：https://ld246.com/article/1565796946371

来源网站：链滴

版权声明：本文为博主原创文章，遵循CC 4.0 by-sa 版权协议，转载请附上原文出处链接和本声明。
本文链接: http://blog.shiyi.online/articles/2019/08/14/1565796937508.html

应用场景

- 创建订单10分钟之后自动支付
- 订单超时取消
- .....等等...

实现方式

- 最简单的方式,定时扫表;例如每分钟扫表一次十分钟之后未支付的订单进行主动支付；

优点: 简单

缺点: 每分钟全局扫表,浪费资源,有一分钟延迟

- 使用RabbitMq 实现　RabbitMq实现延迟队列

优点: 开源,现成的稳定的实现方案;

缺点: RabbitMq是一个消息中间件;延迟队列只是其中一个小功能,如果团队技术栈中本来就是使用RabitMq那还好,如果不是,那为了使用延迟队列而去部署一套RabbitMq成本有点大;

- 使用Java中的延迟队列,DelayQueue

优点: java.util.concurrent包下一个延迟队列,简单易用;拿来即用

缺点: 单机、不能持久化、宕机任务丢失等等;

基于Redis自研延迟队列

既然上面没有很好的解决方案,因为Redis的zset、list的特性,我们可以利用Redis来实现一个延迟队列
edisDelayQueue

设计目标

- 实时性: 允许存在一定时间内的秒级误差
- 高可用性:支持单机,支持集群
- 支持消息删除:业务费随时删除指定消息
- 消息可靠性: 保证至少被消费一次
- 消息持久化: 基于Redis自身的持久化特性,上面的消息可靠性基于Redis的持久化,所以如果redis数丢失,意味着延迟消息的丢失,不过可以做主备和集群保证;

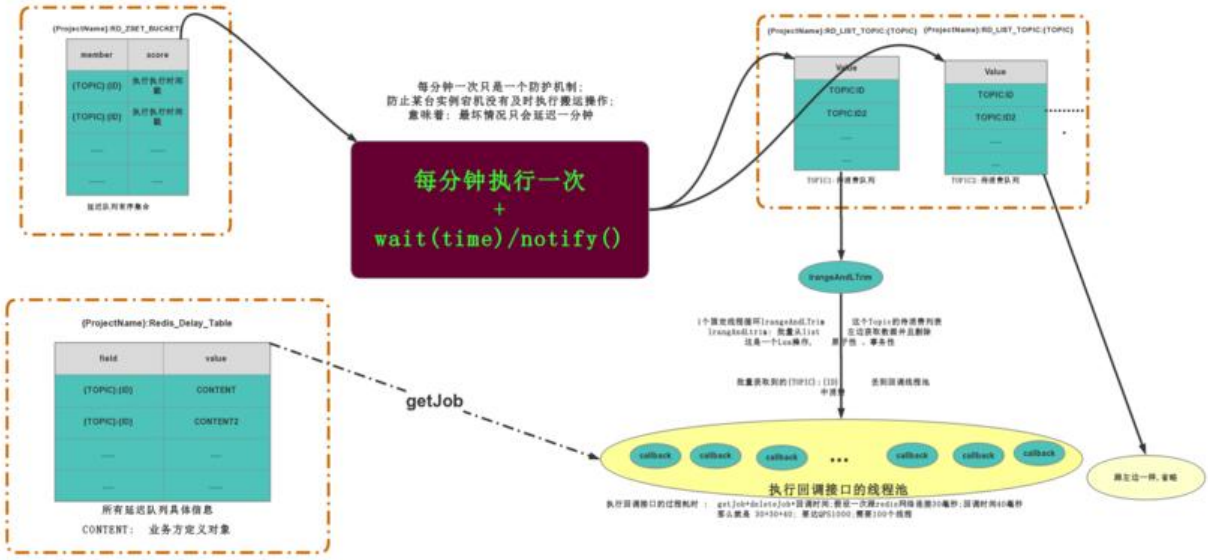
数据结构

● **Redis_Delay_Table**: 是一个Hash_Table结构；里面存储了所有的延迟队列的信息;KV结构；K=TOPC:ID　V=CONENT；V由客户端传入的数据,消费的时候回传；

● **RD_ZSET_BUCKET**: 延迟队列的有序集合; 存放member=TOPIC:ID 和score=执行时间戳; 根据时戳排序;

● **RD_LIST_TOPIC**: list结构; 每个Topic一个list；list存放的都是当前需要被消费的延迟Job;

设计图



任务的生命周期

1. 新增一个　**Job**,会在**Redis_Delay_Table**中插入一条数据,记录了业务消费方的 数据结构; **RD_ZSET_BUCKET** 也会插入一条数据,记录了执行时间戳;

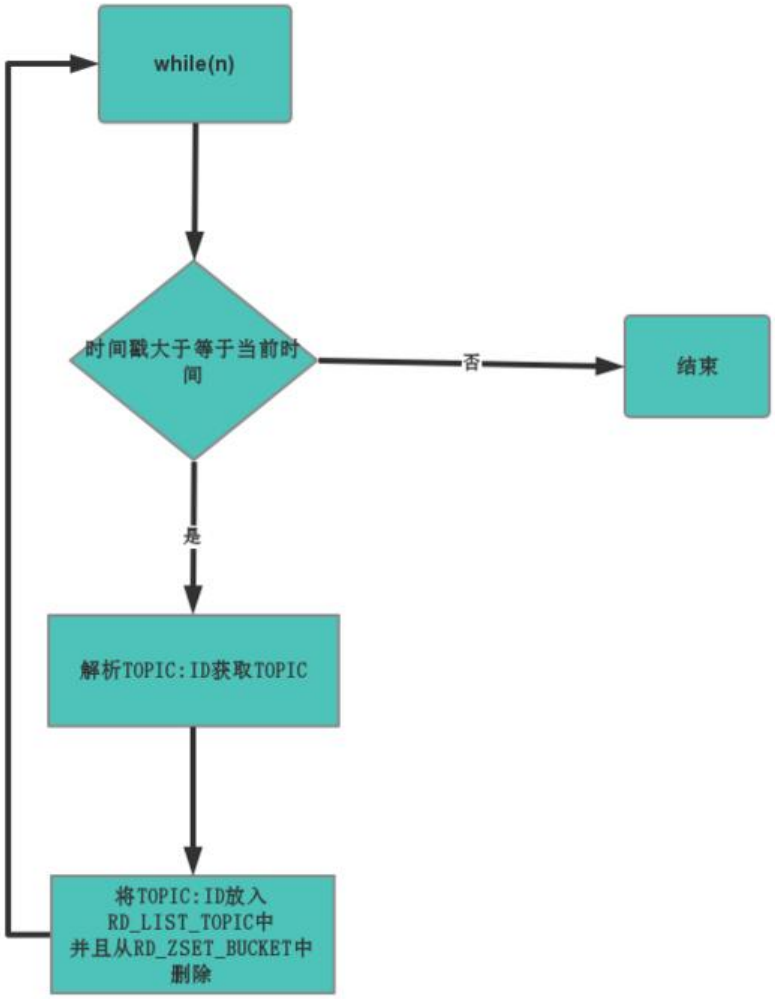
2. 搬运线程会去　**RD_ZSET_BUCKET**中查找哪些执行时间戳**runTimeMillis**比现在的时间小;将这些录全部删除;同时会解析出来每个任务的**Topic**是什么,然后将这些任务**push**到**Topic**对应的列表**RD_LIT_TOPIC**中;

3. 每个Topic的List都会有一个监听线程去批量获取List中的待消费数据;获取到的数据全部扔给这个**opic**的消费线程池

4. 消息线程池执行会去Redis_Delay_Table查找数据结构,返回给回调接口,执行回调方法;

以上所有操作,都是基于Lua脚本做的操作,Lua脚本执行的优点在于,批量命令执行具有原子性,事务性,且降低了网络开销,毕竟只有一次网络开销;

搬运线程操作流程图

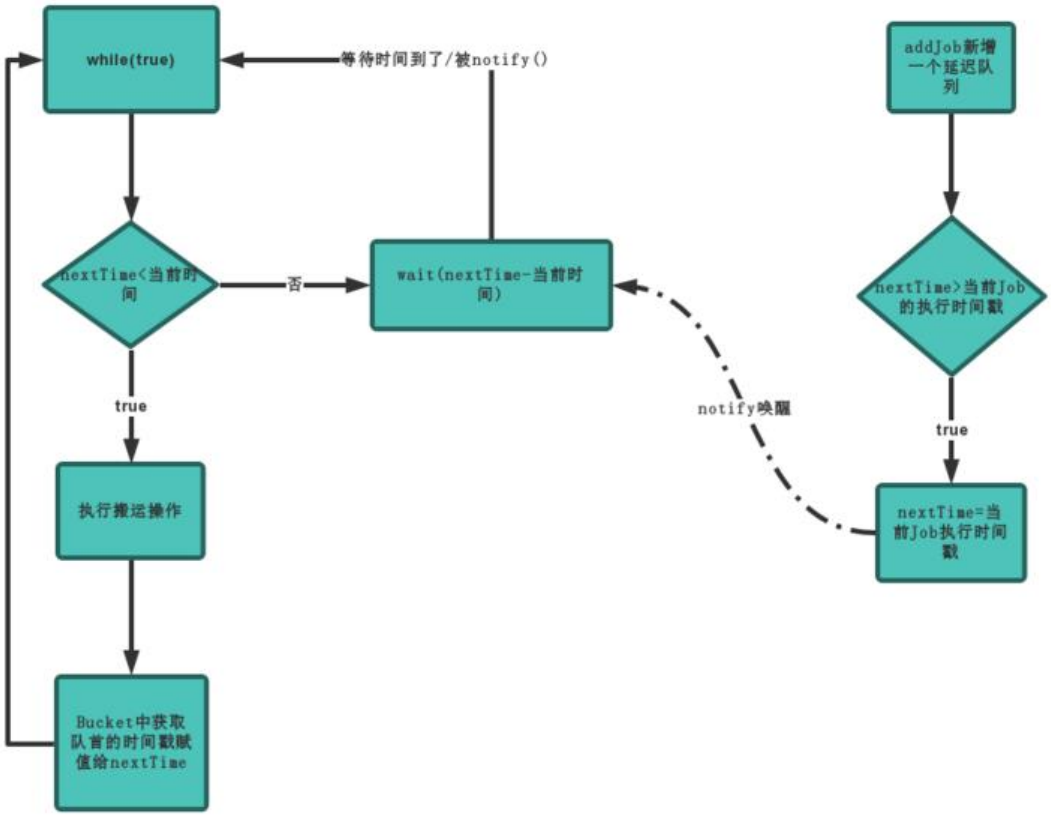


设计细节

搬运操作

1.搬运操作的时机

为了避免频繁的执行搬运操作, 我们基于 wait(time)/notify 的方式来通知执行搬运操作;



我们用一个**AtomicLong nextTime** 来保存下一次将要搬运的时间;服务启动的时候**nextTime**=0;所肯定比当前时间小,那么就会先去执行一次搬运操作,然后返回搬运操作之后的**ZSET**的表头时间戳,这个间戳就是下一次将要执行的时间戳, 把这个时间戳赋值给 **nextTime**; 如果表中没有元素了则将**nextTime=Long.MaxValue** ;因为while循环,下一次又会跟当前时间对比;如果**nextTime**比当前时间大则说明需要等待; 那么我们**wait(nextTime-System.currentTimeMills())**; 等到时间到了之后,再次判断一下,就会比当前时间小,就会执行一次搬运操作;

那么当有新增延迟任务Job的时间怎么办,这个时候又会将当前新增Job的执行时间戳跟**nextTime**做个比;如果小的话就重新赋值;

重新赋值之后,还是调用一下 **notifyAll()** 通知一下搬运线程; 让他重新去判断一下 新的时间是否比当时间小;如果还是大的话,那么就继续**wait(nextTime-System.currentTimeMills())**; 但是这个时候**wat**的时间又会变小;更精准;

2.一次搬运操作的最大数量

redis的执行速度非常快,在一个Lua里面循环遍历1000个10000个根本没差; 而且是在Lua里面操作,就有一次网络开销;一次操作多少个元素根本就不会是问题;

搬运操作的防护机制

1.每分钟唤醒定时线程

在消费方多实例部署的情况下, 如果某一台机器挂掉了,但是这台机器的nextTime是最小的,就在一分之后(新增job的时候落到这台机器,刚好时间戳很小), 其他机器可能是1个小时之后执行搬运操作; 如这台机器立马重启,那么还会立马执行一次搬运操作;万一他没有重启;那可能就会很久之后才会搬运;

所以我们需要一种防护手段来应对这种极端情况;

比如每分钟将nextTime=0;并且唤醒wait;

那么就会至少每分钟会执行一次搬运操作! 这是可以接受的

LrangeAndLTrim 批量获取且删除待消费任务

1.执行时机以及如何防止频繁请求redis

这是一个守护线程,循环去做这样的操作,把拿到的数据给线程池去消费;

但是也不能一直不停的去执行操作,如果list已经没有数据了去操作也没有任何意义,不然就太浪费资源了幸好List中有一个**BLPOP**阻塞原语,如果list中有数据就会立马返回,如果没有数据就会一直阻塞在那里直到有数据返回,可以设置阻塞的超时时间,超时会返回NULL;

第一次去获取N个待消费的任务扔进到消费线程池中;如果获取到了0个,那么我们就立马用**BLPOP**来阻,等有元素的时候 BLPOP就返回数据了,下次就可以尝试去**LrangeAndLTrim**一次了. 通过**BLPOP**阻塞我们避免了频繁的去请求redis,并且更重要的是提高了实时性;

2.批量获取的数量和消费线程池的阻塞队列

执行上面的一次获取N个元素是不定的,这个要看线程池的maxPoolSize 最大线程数量; 因为避免消费任务过多而放入线程池的阻塞队列, 放入阻塞队列有宕机丢失任务的风险,关机重启的时候还要讲阻塞列中的任务重新放入List中增加了复杂性;

所以我们每次**LrangeAndLTrim**获取的元素不能大于当前线程池可用的线程数; 这样的一个控制可用信号量**Semaphore**来做

Codis集群对BLPOP的影响

如果redis集群用了codis方案或者Twemproxy方案; 他们不支持BLPOP的命令;

codis不支持的命令集合

那么就不能利用BLPOP来防止频繁请求redis;那么退而求其次改成每秒执行一次LrangeAndLTrim操作;

集群对Lua的影响

Lua脚本的执行只能在单机器上, 集群的环境下如果想要执行Lua脚本不出错，那么Lua脚本中的所有ke必须落在同一台机器;

为了支持集群操作Lua,我们利用hashtag;　用{}把三个jey的关键词包起来;

{projectName}:Redis_Delay_Table

{projectName}:Redis_Delay_Table

{projectName}:RD_LIST_TOPIC

那么所有的数据就会在同一台机器上了

重试机制

消费者回调接口如果抛出异常了,或者执行超时了,那么会将这个Job重新放入到RD_LIST_TOPIC中等被下一次消费;默认重试2次;可以设置不重试;

超时机制

超时机制的主要思路都一样,就是监听一个线程的执行时间超过设定值之后抛出异常打断方法的执行;

这是使用的方式是 利用Callable接口实现异步超时处理

```
public class TimeoutUtil {

    /**执行用户回调接口的 线程池;    计算回调接口的超时时间          **/
    private static ExecutorService executorService = Executors.newCachedThreadPool();

    /**
     * 有超时时间的方法
     * @param timeout 时间秒
     * @return
     */
    public static void timeoutMethod(long timeout, Function function) throws InterruptedExce
tion, ExecutionException, TimeoutException {
        FutureTask futureTask = new FutureTask(()->(function.apply("")));
        executorService.execute(futureTask);
        //new Thread(futureTask).start();
        try {
            futureTask.get(timeout, TimeUnit.MILLISECONDS);
        } catch (InterruptedException | ExecutionException | TimeoutException e) {
            //e.printStackTrace();
            futureTask.cancel(true);
            throw e;
        }

    }
}
```

这种方式有一点不好就是太费线程了,相当于线程使用翻了一倍;但是相比其他的方式,这种算是更好一的

优雅停机

在Jvm那里注册一个 Runtime.getRuntime().addShutdownHook(Runnable)停机回调接口;在这里做好善后工作;

- 关闭异步AddJob线程池
- 关闭每分钟唤醒线程
- 关闭搬运线程 while(!stop)的形式
- 关闭所有的topic监听线程 while(!stop)的形式
- 关闭关闭所有topic的消费线程 ;先调用shutdown;再executor.awaitTermination(20, TimeUnit.SECONDS);检查是否还有剩余的线程任务没有执行完; 如果还没有执行完则等待执行完；最多等待20秒之强制调用shutdownNow强制关闭;
- 关闭重试线程 while(!stop)的形式
- 关闭 异常未消费Job重入List线程池

优雅停止线程一般是用下面的方式

①、 while(!stop)的形式 用标识位来停止线程

②.先 调用executor.shutdown(); 阻止接受新的任务;然后等待当前正在执行的任务执行完; 如果有阻则需要调用executor.shutdownNow()强制结束;所以要给一个等待时间;

```
 /**
    * shutdownNow 终止线程的方法是通过调用Thread.interrupt()方法来实现的
    * 如果线程中没有sleep 、wait、Condition、定时锁等应用, interrupt()方法是无法中断当前的线的。
    * 上面的情况中断之后还是可以再执行finally里面的方法的;
    * 但是如果是其他的情况 finally是不会被执行的
    * @param executor
    */
   public static void closeExecutor(ExecutorService executor, String executorName) {
       try {
           //新的任务不进队列
           executor.shutdown();
           //给10秒钟没有停止完强行停止;
           if(!executor.awaitTermination(20, TimeUnit.SECONDS)) {
               logger.warn("线程池: {},{}没有在20秒内关闭,则进行强制关闭",executorName,executor);
               List<Runnable> droppedTasks = executor.shutdownNow();
               logger.warn("线程池: {},{} 被强行关闭,阻塞队列中将有{}个将不会被执行.", executorName executor,droppedTasks.size() );
           }
           logger.info("线程池:{},{} 已经关闭...",executorName,executor);
       } catch (InterruptedException e) {
           logger.info("线程池:{},{} 打断...",executorName,executor);
       }
   }
```

BLPOP阻塞的情况如何优雅停止监听redis的线程

如果不是在**codis**集群的环境下,BLPOP是可以很方便的阻塞线程的;但是停机的时候可能会有点问题;

假如正在关机,当前线程正在**BLPOP**阻塞, 那关机线程等我们20秒执行, 刚好在倒数1秒的时候**BLPOP**取到了数据,丢给消费线程去消费;如果消费线程1秒执行不完,那么20秒倒计时到了,强制关机,那么这个务就会被丢失了; 怎么解决这个问题呢?

①. 不用**BLPOP**, 每次都sleep一秒去调用**LrangeAndLTrim**操作;

②.关机的时候杀掉 redis的blpop客户端; 杀掉之后 BLPOP会立马返回null; 进入下一个循环体;

不足

- 因为Redis的持久化特性,做不到消息完全不丢失,如果要保证完成不丢失,Redis的持久化刷盘策略要紧
- 因为Codis不能使用BLPOP这种阻塞的形式,在获取消费任务的时候用了每秒一次去获取,有点浪费性;
- 支持消费者多实例部署,但是可能存在不能均匀的分配到每台机器上去消费;
- 虽然支持redis集群,但是其实是伪集群,因为Lua脚本的原因,让他们都只能落在一台机器上;

总结

1. **实时性**

正常情况下 消费的时间误差不超过1秒钟; 极端情况下,一台实例宕机,另外的实例nextTime很迟; 那么大误差是1分钟; 真正的误差来自于业务方的接口的消费速度

2. **QPS**

完全视业务方的消费速度而定; 延迟队列不是瓶颈

项目已经开源 ,源码地址: RedisDelayQueue

接入方法:

RedisDelayQueue接入