应用 2：缓兵之计 —— 延时队列

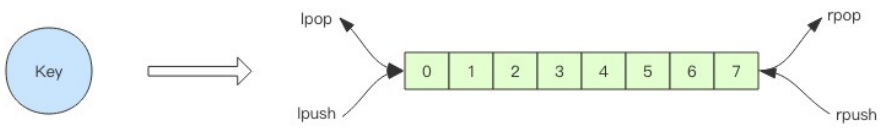
我们平时习惯于使用 Rabbitmq 和 Kafka 作为消息队列中间件，来给应用程序之间增加异步消息传递功能。这两个中间件都是专业的消息队列中间件，特性之多超出了大多数人的理解能力。

使用过 Rabbitmq 的同学知道它使用起来有多复杂，发消息之前要创建 Exchange，再创建 Queue，还要将 Queue 和 Exchange 通过某种规则绑定起来，发消息的时候要指定 routing-key，还要控制头部信息。消费者在消费消息之前也要进行上面一系列的繁琐过程。但是绝大多数情况下，虽然我们的消息队列只有一组消费者，但还是需要经历上面这些繁琐的过程。

有了 Redis，它就可以让我们解脱出来，对于那些只有一组消费者的消息队列，使用 Redis 就可以非常轻松的搞定。Redis 的消息队列不是专业的消息队列，它没有非常多的高级特性，没有 ack 保证，如果对消息的可靠性有着极致的追求，那么它就不适合使用。

异步消息队列

Redis 的 list(列表) 数据结构常用来作为异步消息队列使用，使用 `rpush/lpush` 操作入队列，使用 `lpop` 和 `rpop` 来出队列。




```
> rpush notify-queue apple banana pear
(integer) 3
> llen notify-queue
```

```
(integer) 3
> lpop notify-queue
"apple"
> llen notify-queue
(integer) 2
> lpop notify-queue
"banana"
> llen notify-queue
(integer) 1
> lpop notify-queue
"pear"
> llen notify-queue
(integer) 0
> lpop notify-queue
(nil)
```

上面是 rpush 和 lpop 结合使用的例子。还可以使用 lpush 和 rpop 结合使用，效果是一样的。这里不再赘述。

队列空了怎么办?

客户端是通过队列的 pop 操作来获取消息，然后进行处理。处理完了再接着获取消息，再进行处理。如此循环往复，这便是作为队列消费者的客户端的生命周期。

可是如果队列空了，客户端就会陷入 pop 的死循环，不停地 pop，没有数据，接着再 pop，又没有数据。这就是浪费生命的空轮询。空轮询不但拉高了客户端的 CPU，redis 的 QPS 也会被拉高，如果这样空轮询的客户端有几十来个，Redis 的慢查询可能会显著增多。

通常我们使用 sleep 来解决这个问题，让线程睡一会，睡个 1s 钟就可以了。不但客户端的 CPU 能降下来，Redis 的 QPS 也降下来了。

```
time.sleep(1)  # python 睡 1s
Thread.sleep(1000)  # java 睡 1s
```

队列延迟

用上面睡眠的办法可以解决问题。但是有个小问题，那就是睡眠会导致消息的延迟增大。如果只有 1 个消费者，那么这个延迟就是 1s。如果有多个消费者，这个延迟会有所下降，因为每个消费者的睡觉时间是岔开来的。

有没有什么办法能显著降低延迟呢？你当然可以很快想到：那就把睡觉的时间缩短点。这种方式当然可以，不过有没有更好的解决方案呢？当然也有，那就是blpop/brpop。

这两个指令的前缀字符 `b` 代表的是 `blocking` ，也就是阻塞读。

阻塞读在队列没有数据的时候，会立即进入休眠状态，一旦数据到来，则立刻醒过来。消息的延迟几乎为零。用 `blpop/brpop` 替代前面的 `lpop/rpop` ，就完美解决了上面的问题。

空闲连接自动断开

你以为上面的方案真的很完美么？先别急着开心，其实他还有个问题需要解决。

什么问题？—— **空闲连接**的问题。

如果线程一直阻塞在哪里，Redis 的客户端连接就成了闲置连接，闲置过久，服务器一般会主动断开连接，减少闲置资源占用。这个时候 `blpop/brpop` 会抛出异常来。

所以编写客户端消费者的时候要小心，注意捕获异常，还要重试。

锁冲突处理

上节课我们讲了分布式锁的问题，但是没有提到客户端在处理请求时加锁没加成功怎么办。一般有 3 种策略来处理加锁失败：

1. 直接抛出异常，通知用户稍后重试；
2. sleep 一会再重试；
3. 将请求转移至延时队列，过一会再试；

直接抛出特定类型的异常

这种方式比较适合由用户直接发起的请求，用户看到错误对话框后，会先阅读对话框的内容，再点击重试，这样就可以起到人工延时的效果。如果考虑到用户体验，可以由前端的代码替代用户自己来进行延时重试控制。它本质上是对当前请求的放弃，由用户决定是否重新发起新的请求。

sleep

sleep 会阻塞当前的消息处理线程，会导致队列的后续消息处理出现延迟。如果碰撞的比较频繁或者队列里消息比较多，sleep 可能并不合适。如果因为个别死锁的 key 导致加锁不成功，线程会彻底堵死，导致后续消息永远得不到及时处理。

延时队列

这种方式比较适合异步消息处理，将当前冲突的请求扔到另一个队列延后处理以避开冲突。

延时队列的实现

延时队列可以通过 Redis 的 zset(有序列表) 来实现。我们将消息序列化成一个字符串作为 zset 的 `value` ，这个消息的到期处理时间作为 `score` ，然后用多个线程轮询 zset 获取到期的任务进行处理，多个线程是为了保障可用性，万一挂了一个线程还有其它线程可以继续处理。因为有多个线程，所以需要考虑并发争抢任务，确保任务不能被多次执行。

py

```
def delay(msg):
    msg.id = str(uuid.uuid4())  # 保证 value 值唯一
    value = json.dumps(msg)
    retry_ts = time.time() + 5  # 5 秒后重试
    redis.zadd("delay-queue", retry_ts, value)


def loop():
    while True:
        # 最多取 1 条
        values = redis.zrangebyscore("delay-queue", 0, time.time(), start=0, num=1)
        if not values:
            time.sleep(1)  # 延时队列空的，休息 1s
            continue
        value = values[0]  # 拿第一条，也只有一条
        success = redis.zrem("delay-queue", value)  # 从消息队列中移除该消息
        if success:  # 因为有多进程并发的可能，最终只会有一个进程可以抢到消息
            msg = json.loads(value)
            handle_msg(msg)
```

Redis 的 zrem 方法是多线程多进程争抢任务的关键，它的返回值决定了当前实例有没有抢到任务，因为 loop 方法可能会被多个线程、多个进程调用，同一个任务可能会被多个进程线程抢到，通过 zrem 来决定唯一的属主。

同时，我们要注意一定要对 handle_msg 进行异常捕获，避免因为个别任务处理问题导致循环异常退出。以下是 Java 版本的延时队列实现，因为要使用到 Json 序列化，所以还需要 fastjson 库的支持。

```
import java.lang.reflect.Type;
import java.util.Set;
import java.util.UUID;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.TypeReference;

import redis.clients.jedis.Jedis;

public class RedisDelayingQueue<T> {

  static class TaskItem<T> {
    public String id;
    public T msg;
  }

  // fastjson 序列化对象中存在 generic 类型时，需要使用 TypeReference
```

```
    private Type TaskType = new TypeReference<TaskItem<T>>() {
    }.getType();

    private Jedis jedis;
    private String queueKey;

    public RedisDelayingQueue(Jedis jedis, String queueKey) {
      this.jedis = jedis;
      this.queueKey = queueKey;
    }

    public void delay(T msg) {
      TaskItem<T> task = new TaskItem<T>();
      task.id = UUID.randomUUID().toString(); // 分配唯一的 uuid
      task.msg = msg;
      String s = JSON.toJSONString(task); // fastjson 序列化
      jedis.zadd(queueKey, System.currentTimeMillis() + 5000, s); // 塞入延时队列 ,5s 后再试
    }

    public void loop() {
      while (!Thread.interrupted()) {
        // 只取一条
        Set<String> values = jedis.zrangeByScore(queueKey, 0, System.currentTimeMillis(),
        if (values.isEmpty()) {
          try {
            Thread.sleep(500); // 歇会继续
          } catch (InterruptedException e) {
            break;
          }
          continue;
        }
        String s = values.iterator().next();
        if (jedis.zrem(queueKey, s) > 0) { // 抢到了
          TaskItem<T> task = JSON.parseObject(s, TaskType); // fastjson 反序列化
          this.handleMsg(task.msg);
        }
      }
    }

    public void handleMsg(T msg) {
      System.out.println(msg);
    }

    public static void main(String[] args) {
      Jedis jedis = new Jedis();
      RedisDelayingQueue<String> queue = new RedisDelayingQueue<>(jedis, "q-demo");
      Thread producer = new Thread() {

        public void run() {
          for (int i = 0; i < 10; i++) {
```

```
            queue.delay("codehole" + i);
          }
        }

    };
    Thread consumer = new Thread() {

      public void run() {
        queue.loop();
      }

    };
    producer.start();
    consumer.start();
    try {
      producer.join();
      Thread.sleep(6000);
      consumer.interrupt();
      consumer.join();
    } catch (InterruptedException e) {
    }
  }
}
```

进一步优化

上面的算法中同一个任务可能会被多个进程取到之后再使用zrem进行争抢，那些没抢到的进程都是白取了一次任务，这是浪费。可以考虑使用lua scripting来优化一下这个逻辑，将zrangebyscore和zrem一同挪到服务器端进行原子化操作，这样多个进程之间争抢任务时就不会出现这种浪费了。

思考

1. Redis 作为消息队列为什么不能保证 100% 的可靠性?
2. 使用lua scripting来优化延时队列的逻辑