# 15-消息队列的考验：Redis有哪些解决方案？

你好，我是蒋德钧。

现在的互联网应用基本上都是采用分布式系统架构进行设计的，而很多分布式系统必备的一个基础软件就是消息队列。

消息队列要能支持组件通信消息的快速读写，而Redis本身支持数据的高速访问，正好可以满足消息队列的读写性能需求。不过，除了性能，消息队列还有其他的要求，所以，很多人都很关心一个问题：“Redis适合做消息队列吗？”

其实，这个问题的背后，隐含着两方面的核心问题：

- 消息队列的消息存取需求是什么？
- Redis如何实现消息队列的需求？

这节课，我们就来聊一聊消息队列的特征和Redis提供的消息队列方案。只有把这两方面的知识和实践经验串连起来，才能彻底理解基于Redis实现消息队列的技术实践。以后当你需要为分布式系统组件做消息队列选型时，就可以根据组件通信量和消息通信速度的要求，选择出适合的Redis消息队列方案了。

我们先来看下第一个问题：消息队列的消息读取有什么样的需求？

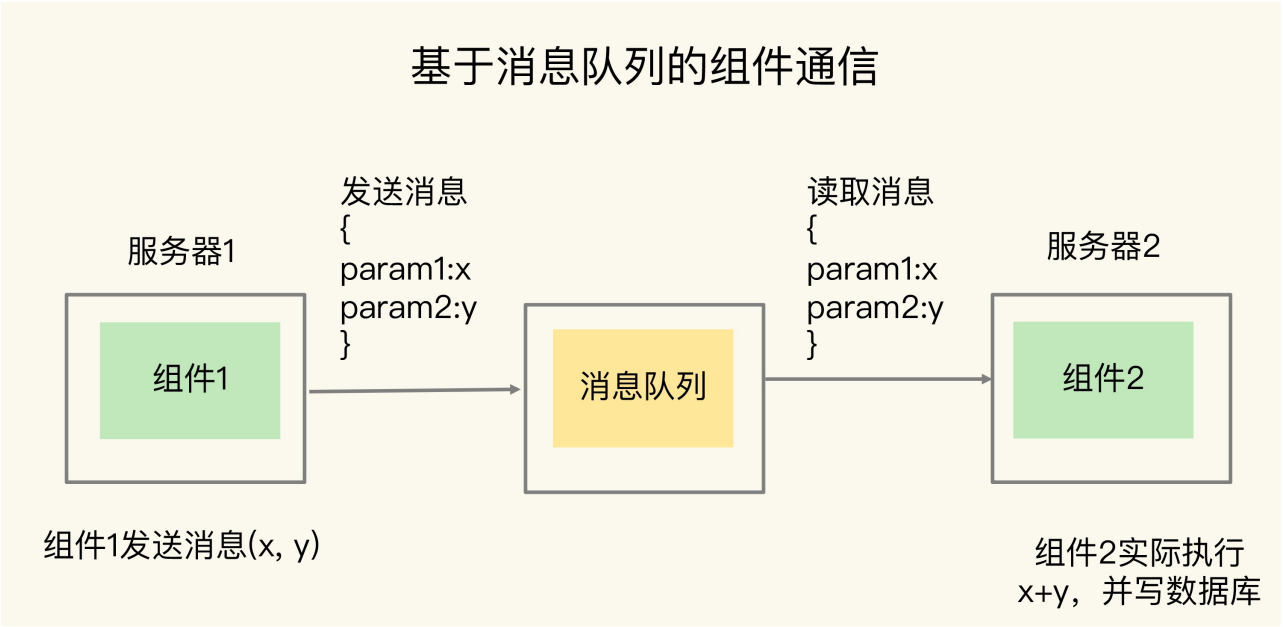
## 消息队列的消息存取需求

我先介绍一下消息队列存取消息的过程。在分布式系统中，当两个组件要基于消息队列进行通信时，一个组件会把要处理的数据以消息的形式传递给消息队列，然后，这个组件就可以继续执行其他操作了；远端的另一个组件从消息队列中把消息读取出来，再在本地进行处理。

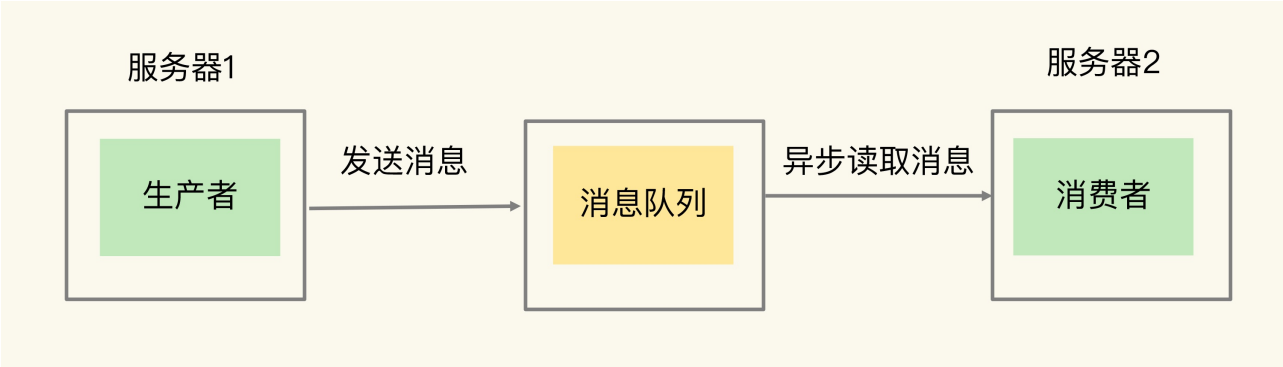
为了方便你理解，我还是借助一个例子来解释一下。

假设组件1需要对采集到的数据进行求和计算，并写入数据库，但是，消息到达的速度很快，组件1没有办法及时地既做采集，又做计算，并且写入数据库。所以，我们可以使用基于消息队列的通信，让组件1把数据x和y保存为JSON格式的消息，再发到消息队列，这样它就可以继续接收新的数据了。组件2则异步地从消息队列中把数据读取出来，在服务器2上进行求和计算后，再写入数据库。这个过程如下图所示：

www.ixuexi.cf 分享站 用户标记 group_share




我们一般把消息队列中发送消息的组件称为生产者（例子中的组件1），把接收消息的组件称为消费者（例子中的组件2），下图展示了一个通用的消息队列的架构模型：




在使用消息队列时，消费者可以异步读取生产者消息，然后再进行处理。这样一来，即使生产者发送消息的速度远远超过了消费者处理消息的速度，生产者已经发送的消息也可以缓存在消息队列中，避免阻塞生产者，这是消息队列作为分布式组件通信的一大优势。

**不过，消息队列在存取消息时，必须要满足三个需求，分别是消息保序、处理重复的消息和保证消息可靠性。**

### 需求一：消息保序

虽然消费者是异步处理消息，但是，消费者仍然需要按照生产者发送消息的顺序来处理消息，避免后发送的消息被先处理了。对于要求消息保序的场景来说，一旦出现这种消息被乱序处理的情况，就可能会导致业务逻辑被错误执行，从而给业务方造成损失。

我们来看一个更新商品库存的场景。

假设生产者负责接收库存更新请求，消费者负责实际更新库存，现有库存量是10。生产者先后发送了消息1和消息2，消息1要把商品X的库存记录更新为5，消息2是把商品X库存更新为3。如果消息1和2在消息队列中无法保序，出现消息2早于消息1被处理的情况，那么，很显然，库存更新就出错了。这是业务应用无法接受的。

面对这种情况，你可能会想到一种解决方案：不要把更新后的库存量作为生产者发送的消息，而是**把库存扣**

**除值作为消息的内容**。这样一来，消息1是扣减库存量5，消息2是扣减库存量2。如果消息1和消息2之间没有库存查询请求的话，即使消费者先处理消息2，再处理消息1，这个方案也能够保证最终的库存量是正确的，也就是库存量为3。

但是，我们还需要考虑这样一种情况：假如消费者收到了这样三条消息：消息1是扣减库存量5，消息2是读取库存量，消息3是扣减库存量2，此时，如果消费者先处理了消息3（把库存量扣减2），那么库存量就变成了8。然后，消费者处理了消息2，读取当前的库存量是8，这就会出现库存量查询不正确的情况。从业务应用层面看，消息1、2、3应该是顺序执行的，所以，消息2查询到的应该是扣减了5以后的库存量，而不是扣减了2以后的库存量。所以，用库存扣除值作为消息的方案，在消息中同时包含读写操作的场景下，会带来数据读取错误的问题。而且，这个方案还会面临一个问题，那就是重复消息处理。

### 需求二：重复消息处理

消费者从消息队列读取消息时，有时会因为网络堵塞而出现消息重传的情况。此时，消费者可能会收到多条重复的消息。对于重复的消息，消费者如果多次处理的话，就可能造成一个业务逻辑被多次执行，如果业务逻辑正好是要修改数据，那就会出现数据被多次修改的问题了。

还是以库存更新为例，假设消费者收到了一次消息1，要扣减库存量5，然后又收到了一次消息1，那么，如果消费者无法识别这两条消息实际是一条相同消息的话，就会执行两次扣减库存量5的操作，此时，库存量就不对了。这当然也是无法接受的。

### 需求三：消息可靠性保证

另外，消费者在处理消息的时候，还可能出现因为故障或宕机导致消息没有处理完成的情况。此时，消息队列需要能提供消息可靠性的保证，也就是说，当消费者重启后，可以重新读取消息再次进行处理，否则，就会出现消息漏处理的问题了。

Redis的List和Streams两种数据类型，就可以满足消息队列的这三个需求。我们先来了解下基于List的消息队列实现方法。

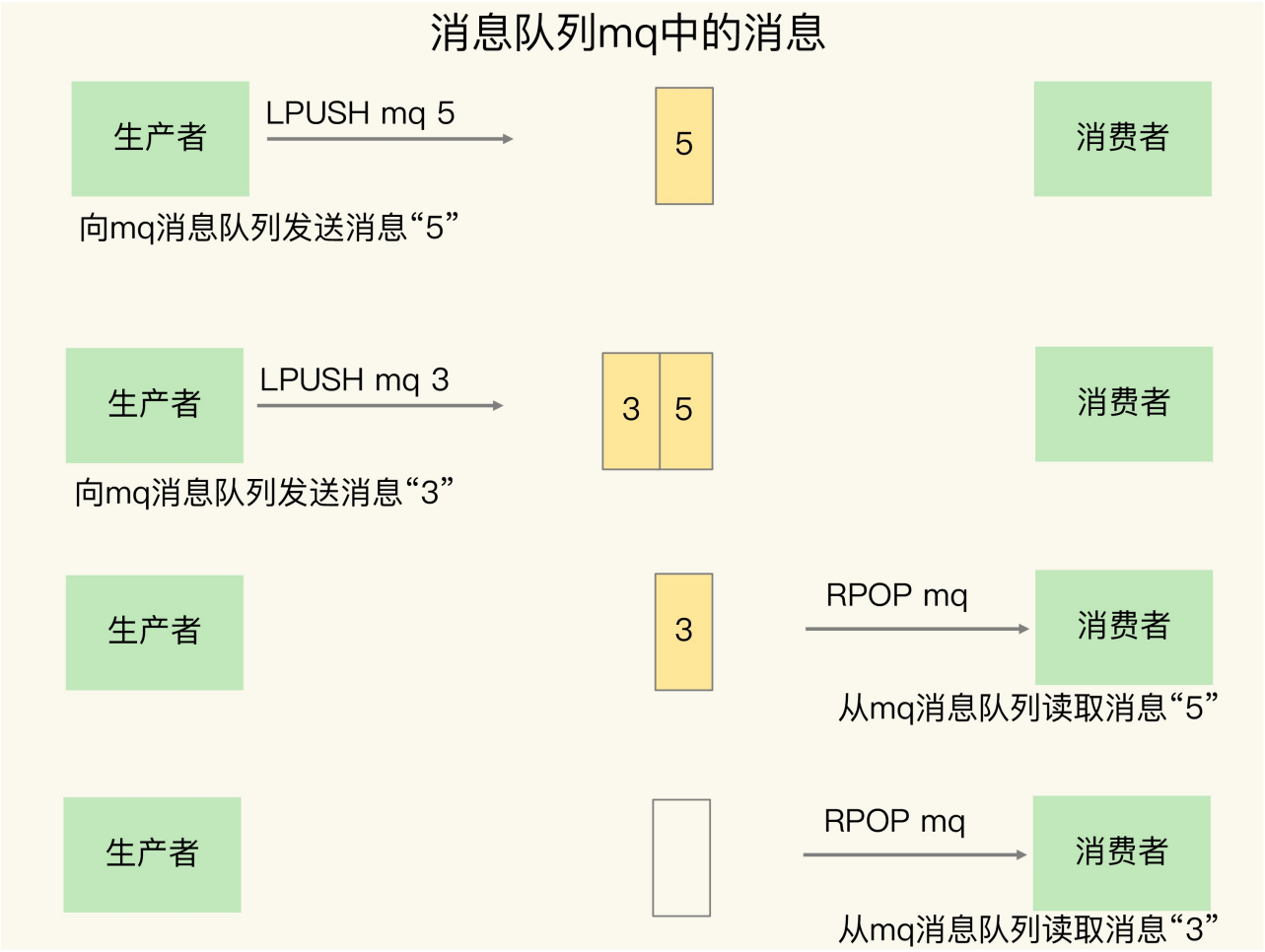
## 基于List的消息队列解决方案

List本身就是按先进先出的顺序对数据进行存取的，所以，如果使用List作为消息队列保存消息的话，就已经能满足消息保序的需求了。

具体来说，生产者可以使用LPUSH命令把要发送的消息依次写入List，而消费者则可以使用RPOP命令，从List的另一端按照消息的写入顺序，依次读取消息并进行处理。

如下图所示，生产者先用LPUSH写入了两条库存消息，分别是5和3，表示要把库存更新为5和3；消费者则用RPOP把两条消息依次读出，然后进行相应的处理。

www.ixuexi.cf 分享站 用户标记 group_share




不过，在消费者读取数据时，有一个潜在的性能风险点。

在生产者往List中写入数据时，List并不会主动地通知消费者有新消息写入，如果消费者想要及时处理消息，就需要在程序中不停地调用RPOP命令（比如使用一个while(1)循环）。如果有新消息写入，RPOP命令就会返回结果，否则，RPOP命令返回空值，再继续循环。

所以，即使没有新消息写入List，消费者也要不停地调用RPOP命令，这就会导致消费者程序的CPU一直消耗在执行RPOP命令上，带来不必要的性能损失。

为了解决这个问题，Redis提供了BRPOP命令。**BRPOP命令也称为阻塞式读取，客户端在没有读到队列数据时，自动阻塞，直到有新的数据写入队列，再开始读取新数据**。和消费者程序自己不停地调用RPOP命令相比，这种方式能节省CPU开销。

消息保序的问题解决了，接下来，我们还需要考虑解决重复消息处理的问题，这里其实有一个要求：**消费者程序本身能对重复消息进行判断。**

一方面，消息队列要能给每一个消息提供全局唯一的ID号；另一方面，消费者程序要把已经处理过的消息的ID号记录下来。

当收到一条消息后，消费者程序就可以对比收到的消息ID和记录的已处理过的消息ID，来判断当前收到的消息有没有经过处理。如果已经处理过，那么，消费者程序就不再进行处理了。这种处理特性也称为幂等性，幂等性就是指，对于同一条消息，消费者收到一次的处理结果和收到多次的处理结果是一致的。

不过，List本身是不会为每个消息生成ID号的，所以，消息的全局唯一ID号就需要生产者程序在发送消息前自行生成。生成之后，我们在用LPUSH命令把消息插入List时，需要在消息中包含这个全局唯一ID。

例如，我们执行以下命令，就把一条全局ID为101030001、库存量为5的消息插入了消息队列：

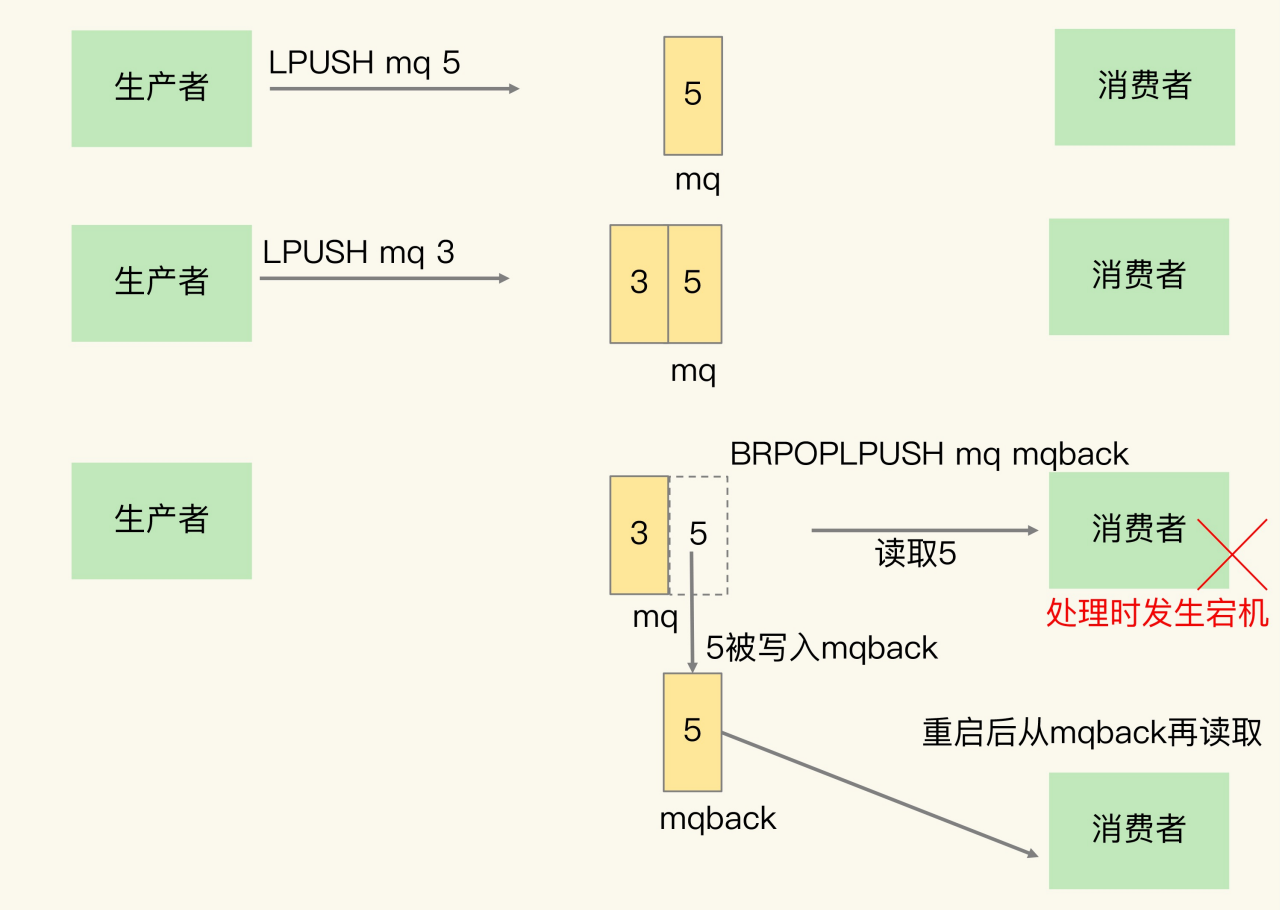
```
LPUSH mq "101030001:stock:5"
(integer) 1
```

最后，我们再来看下，List类型是如何保证消息可靠性的。

当消费者程序从List中读取一条消息后，List就不会再留存这条消息了。所以，如果消费者程序在处理消息的过程出现了故障或宕机，就会导致消息没有处理完成，那么，消费者程序再次启动后，就没法再次从List中读取消息了。

为了留存消息，List类型提供了BRPOPLPUSH命令，这个命令的作用是让消费者程序从一个List中读取消息，同时，Redis会把这个消息再插入到另一个List（可以叫作备份List）留存。这样一来，如果消费者程序读了消息但没能正常处理，等它重启后，就可以从备份List中重新读取消息并进行处理了。

我画了一张示意图，展示了使用BRPOPLPUSH命令留存消息，以及消费者再次读取消息的过程，你可以看下。




生产者先用LPUSH把消息“5”“3”插入到消息队列mq中。消费者程序使用BRPOPLPUSH命令读取消息“5”，同时，消息“5”还会被Redis插入到mqback队列中。如果消费者程序处理消息“5”时宕机了，等它重启后，可以从mqback中再次读取消息“5”，继续处理。

好了，到这里，你可以看到，基于List类型，我们可以满足分布式组件对消息队列的三大需求。但是，在用List做消息队列时，我们还可能遇到过一个问题：**生产者消息发送很快，而消费者处理消息的速度比较慢，**

www.ixuexi.cf 分享站 用户标记 group_share

**这就导致List中的消息越积越多，给Redis的内存带来很大压力**。

这个时候，我们希望启动多个消费者程序组成一个消费组，一起分担处理List中的消息。但是，List类型并不支持消费组的实现。那么，还有没有更合适的解决方案呢？这就要说到Redis从5.0版本开始提供的Streams数据类型了。

和List相比，Streams同样能够满足消息队列的三大需求。而且，它还支持消费组形式的消息读取。接下来，我们就来了解下Streams的使用方法。

## 基于Streams的消息队列解决方案

Streams是Redis专门为消息队列设计的数据类型，它提供了丰富的消息队列操作命令。

- XADD：插入消息，保证有序，可以自动生成全局唯一ID；
- XREAD：用于读取消息，可以按ID读取数据；
- XREADGROUP：按消费组形式读取消息；
- XPENDING和XACK：XPENDING命令可以用来查询每个消费组内所有消费者已读取但尚未确认的消息，而XACK命令用于向消息队列确认消息处理已完成。

首先，我们来学习下Streams类型存取消息的操作XADD。

XADD命令可以往消息队列中插入新消息，消息的格式是键-值对形式。对于插入的每一条消息，Streams可以自动为其生成一个全局唯一的ID。

比如说，我们执行下面的命令，就可以往名称为mqstream的消息队列中插入一条消息，消息的键是repo，值是5。其中，消息队列名称后面的*，表示让Redis为插入的数据自动生成一个全局唯一的ID，例如“1599203861727-0”。当然，我们也可以不用*，直接在消息队列名称后自行设定一个ID号，只要保证这个ID号是全局唯一的就行。不过，相比自行设定ID号，使用*会更加方便高效。

```
XADD mqstream * repo 5
"1599203861727-0"
```

可以看到，消息的全局唯一ID由两部分组成，第一部分“1599203861727”是数据插入时，以毫秒为单位计算的当前服务器时间，第二部分表示插入消息在当前毫秒内的消息序号，这是从0开始编号的。例如，“1599203861727-0”就表示在“1599203861727”毫秒内的第1条消息。

当消费者需要读取消息时，可以直接使用XREAD命令从消息队列中读取。

XREAD在读取消息时，可以指定一个消息ID，并从这个消息ID的下一条消息开始进行读取。

例如，我们可以执行下面的命令，从ID号为1599203861727-0的消息开始，读取后续的所有消息（示例中一共3条）。

```
XREAD BLOCK 100 STREAMS  mqstream 1599203861727-0
1) 1) "mqstream"
   2) 1) 1) "1599274912765-0"
         2) 1) "repo"
            2) "3"
      2) 1) "1599274925823-0"
         2) 1) "repo"
            2) "2"
      3) 1) "1599274927910-0"
         2) 1) "repo"
            2) "1"
```

另外，消费者也可以在调用XRAED时设定block配置项，实现类似于BRPOP的阻塞读取操作。当消息队列中没有消息时，一旦设置了block配置项，XREAD就会阻塞，阻塞的时长可以在block配置项进行设置。

举个例子，我们来看一下下面的命令，其中，命令最后的“$”符号表示读取最新的消息，同时，我们设置了block 10000的配置项，10000的单位是毫秒，表明XREAD在读取最新消息时，如果没有消息到来，XREAD将阻塞10000毫秒（即10秒），然后再返回。下面命令中的XREAD执行后，消息队列mqstream中一直没有消息，所以，XREAD在10秒后返回空值（nil）。

```
XREAD block 10000 streams mqstream $
(nil)
(10.00s)
```

刚刚讲到的这些操作是List也支持的，接下来，我们再来学习下Streams特有的功能。

Streams本身可以使用XGROUP创建消费组，创建消费组之后，Streams可以使用XREADGROUP命令让消费组内的消费者读取消息，

例如，我们执行下面的命令，创建一个名为group1的消费组，这个消费组消费的消息队列是mqstream。

```
XGROUP create mqstream group1 0
OK
```

然后，我们再执行一段命令，让group1消费组里的消费者consumer1从mqstream中读取所有消息，其中，命令最后的参数“>”，表示从第一条尚未被消费的消息开始读取。因为在consumer1读取消息前，group1中没有其他消费者读取过消息，所以，consumer1就得到mqstream消息队列中的所有消息了（一共4条）。

```
XREADGROUP group group1 consumer1 streams mqstream >
1) 1) "mqstream"
   2) 1) 1) "1599203861727-0"
         2) 1) "repo"
            2) "5"
```

www.ixuexi.cf 分享站 用户标记 group_share

```
      2) 1) "1599274912765-0"
         2) 1) "repo"
            2) "3"
      3) 1) "1599274925823-0"
         2) 1) "repo"
            2) "2"
      4) 1) "1599274927910-0"
         2) 1) "repo"
            2) "1"
```

需要注意的是，消息队列中的消息一旦被消费组里的一个消费者读取了，就不能再被该消费组内的其他消费者读取了。比如说，我们执行完刚才的XREADGROUP命令后，再执行下面的命令，让group1内的consumer2读取消息时，consumer2读到的就是空值，因为消息已经被consumer1读取完了，如下所示：

```
XREADGROUP group group1 consumer2  streams mqstream 0
1) 1) "mqstream"
   2) (empty list or set)
```

使用消费组的目的是让组内的多个消费者共同分担读取消息，所以，我们通常会让每个消费者读取部分消息，从而实现消息读取负载在多个消费者间是均衡分布的。例如，我们执行下列命令，让group2中的consumer1、2、3各自读取一条消息。

```
XREADGROUP group group2 consumer1 count 1 streams mqstream >
1) 1) "mqstream"
   2) 1) 1) "1599203861727-0"
         2) 1) "repo"
            2) "5"

XREADGROUP group group2 consumer2 count 1 streams mqstream >
1) 1) "mqstream"
   2) 1) 1) "1599274912765-0"
         2) 1) "repo"
            2) "3"

XREADGROUP group group2 consumer3 count 1 streams mqstream >
1) 1) "mqstream"
   2) 1) 1) "1599274925823-0"
         2) 1) "repo"
            2) "2"
```

为了保证消费者在发生故障或宕机再次重启后，仍然可以读取未处理完的消息，Streams会自动使用内部队列（也称为PENDING List）留存消费组里每个消费者读取的消息，直到消费者使用XACK命令通知Streams“消息已经处理完成”。如果消费者没有成功处理消息，它就不会给Streams发送XACK命令，消息仍然会留存。此时，消费者可以在重启后，用XPENDING命令查看已读取、但尚未确认处理完成的消息。

例如，我们来查看一下group2中各个消费者已读取、但尚未确认的消息个数。其中，XPENDING返回结果的第二、三行分别表示group2中所有消费者读取的消息最小ID和最大ID。

```
XPENDING mqstream group2
1) (integer) 3
2) "1599203861727-0"
3) "1599274925823-0"
4) 1) 1) "consumer1"
      2) "1"
   2) 1) "consumer2"
      2) "1"
   3) 1) "consumer3"
      2) "1"
```

如果我们还需要进一步查看某个消费者具体读取了哪些数据，可以执行下面的命令：

```
XPENDING mqstream group2 - + 10 consumer2
1) 1) "1599274912765-0"
   2) "consumer2"
   3) (integer) 513336
   4) (integer) 1
```

可以看到，consumer2已读取的消息的ID是1599274912765-0。

一旦消息1599274912765-0被consumer2处理了，consumer2就可以使用XACK命令通知Streams，然后这条消息就会被删除。当我们再使用XPENDING命令查看时，就可以看到，consumer2已经没有已读取、但尚未确认处理的消息了。

```
 XACK mqstream group2 1599274912765-0
(integer) 1
XPENDING mqstream group2 - + 10 consumer2
(empty list or set)
```

现在，我们就知道了用Streams实现消息队列的方法，我还想再强调下，Streams是Redis 5.0专门针对消息队列场景设计的数据类型，如果你的Redis是5.0及5.0以后的版本，就可以考虑把Streams用作消息队列了。

## 小结

这节课，我们学习了分布式系统组件使用消息队列时的三大需求：消息保序、重复消息处理和消息可靠性保证，这三大需求可以进一步转换为对消息队列的三大要求：消息数据有序存取，消息数据具有全局唯一编号，以及消息数据在消费完成后被删除。

我画了一张表格，汇总了用List和Streams实现消息队列的特点和区别。当然，在实践的过程中，你也可以根据新的积累，进一步补充和完善这张表。

| | 基于List | 基于Streams |
| --- | --- | --- |
| 消息保序 | 使用LPUSH/RPOP | 使用XADD/XREAD |
| 阻塞读取 | 使用BRPOP | 使用XREAD block |
| 重复消息处理 | 生产者自行实现全局唯一ID | Streams自动生成全局唯一ID |
| 消息可靠性 | 使用BRPOPLPUSH | 使用PENDING List自动留存消息，使用XPENDING查看，使用XACK确认消息 |
| 适用场景 | Redis5.0前版本的部署环境<br>消息总量小 | Redis 5.0及以后的版本的部署环境<br>消息总量大，需要消费组形式读取数据 |

其实，关于Redis是否适合做消息队列，业界一直是有争论的。很多人认为，要使用消息队列，就应该采用Kafka、RabbitMQ这些专门面向消息队列场景的软件，而Redis更加适合做缓存。

根据这些年做Redis研发工作的经验，我的看法是：Redis是一个非常轻量级的键值数据库，部署一个Redis实例就是启动一个进程，部署Redis集群，也就是部署多个Redis实例。而Kafka、RabbitMQ部署时，涉及额外的组件，例如Kafka的运行就需要再部署ZooKeeper。相比Redis来说，Kafka和RabbitMQ一般被认为是重量级的消息队列。

所以，关于是否用Redis做消息队列的问题，不能一概而论，我们需要考虑业务层面的数据体量，以及对性能、可靠性、可扩展性的需求。如果分布式系统中的组件消息通信量不大，那么，Redis只需要使用有限的内存空间就能满足消息存储的需求，而且，Redis的高性能特性能支持快速的消息读写，不失为消息队列的一个好的解决方案。

## 每课一问

按照惯例，我给你提个小问题。如果一个生产者发送给消息队列的消息，需要被多个消费者进行读取和处理（例如，一个消息是一条从业务系统采集的数据，既要被消费者1读取进行实时计算，也要被消费者2读取并留存到分布式文件系统HDFS中，以便后续进行历史查询），你会使用Redis的什么数据类型来解决这个问题呢？

欢迎在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎你帮我分享给更多人。我们下节课见。

## 精选留言：

- Kaito 2020-09-11 01:51:20

  如果一个生产者发送给消息队列的消息，需要被多个消费者进行读取和处理，你会使用Redis的什么数据类型来解决这个问题？

  这种情况下，只能使用Streams数据类型来解决。使用Streams数据类型，创建多个消费者组，就可以实现同时消费生产者的数据。每个消费者组内可以再挂多个消费者分担读取消息进行消费，消费完成后，各自向Redis发送XACK，标记自己的消费组已经消费到了哪个位置，而且消费组之间互不影响。

  另外，老师在介绍使用List用作队列时，为了保证消息可靠性，使用BRPOPLPUSH命令把消息取出的同时，还把消息插入到备份队列中，从而防止消费者故障导致消息丢失。

  这种情况下，还需要额外做一些工作，也就是维护这个备份队列：每次执行BRPOPLPUSH命令后，因为都会把消息插入一份到备份队列中，所以当消费者成功消费取出的消息后，最好把备份队列中的消息删除，防止备份队列存储过多无用的数据，导致内存浪费。

这篇文章主要是讲消息队列的使用，借这个机会，也顺便总结一下使用消息队列时的注意点：

在使用消息队列时，重点需要关注的是如何保证不丢消息？

那么下面就来分析一下，哪些情况下，会丢消息，以及如何解决？

1、生产者在发布消息时异常：

a) 网络故障或其他问题导致发布失败（直接返回错误，消息根本没发出去）
b) 网络抖动导致发布超时（可能发送数据包成功，但读取响应结果超时了，不知道结果如何）

情况a还好，消息根本没发出去，那么重新发一次就好了。但是情况b没办法知道到底有没有发布成功，所以也只能再发一次。所以这两种情况，生产者都需要重新发布消息，直到成功为止（一般设定一个最大重试次数，超过最大次数依旧失败的需要报警处理）。这就会导致消费者可能会收到重复消息的问题，所以消费者需要保证在收到重复消息时，依旧能保证业务的正确性（设计幂等逻辑），一般需要根据具体业务来做，例如使用消息的唯一ID，或者版本号配合业务逻辑来处理。

2、消费者在处理消息时异常：

也就是消费者把消息拿出来了，但是还没处理完，消费者就挂了。这种情况，需要消费者恢复时，依旧能处理之前没有消费成功的消息。使用List当作队列时，也就是利用老师文章所讲的备份队列来保证，代价是增加了维护这个备份队列的成本。而Streams则是采用ack的方式，消费成功后告知中间件，这种方式处理起来更优雅，成熟的队列中间件例如RabbitMQ、Kafka都是采用这种方式来保证消费者不丢消息的。

3、消息队列中间件丢失消息

上面2个层面都比较好处理，只要客户端和服务端配合好，就能保证生产者和消费者都不丢消息。但是，如果消息队列中间件本身就不可靠，也有可能会丢失消息，毕竟生产者和消费这都依赖它，如果它不可靠，那么生产者和消费者无论怎么做，都无法保证数据不丢失。

a) 在用Redis当作队列或存储数据时，是有可能丢失数据的：一个场景是，如果打开AOF并且是每秒写盘，因为这个写盘过程是异步的，Redis宕机时会丢失1秒的数据。而如果AOF改为同步写盘，那么写入性能会下降。另一个场景是，如果采用主从集群，如果写入量比较大，从库同步存在延迟，此时进行主从切换，也存在丢失数据的可能（从库还未同步完成主库发来的数据就被提成主库）。总的来说，Redis不保证严格的数据完整性和主从切换时的一致性。我们在使用Redis时需要注意。

b) 而采用RabbitMQ和Kafka这些专业的队列中间件时，就没有这个问题了。这些组件一般是部署一个集群，生产者在发布消息时，队列中间件一般会采用写多个节点+预写磁盘的方式保证消息的完整性，即便其中一个节点挂了，也能保证集群的数据不丢失。当然，为了做到这些，方案肯定比Redis设计的要复杂（毕竟是专们针对队列场景设计的）。

综上，Redis可以用作队列，而且性能很高，部署维护也很轻量，但缺点是无法严格保数据的完整性（个人认为这就是业界有争议要不要使用Redis当作队列的地方）。而使用专业的队列中间件，可以严格保证数据的完整性，但缺点是，部署维护成本高，用起来比较重。

所以我们需要根据具体情况进行选择，如果对于丢数据不敏感的业务，例如发短信、发通知的场景，可以采用Redis作队列。如果是金融相关的业务场景，例如交易、支付这类，建议还是使用专业的队列中间件。 [8赞]

- 杨逸林 2020-09-11 02:13:26

  1. 问题回答

  每个 Stream 都可以挂多个消费组，每个消费组会有个游标 last_delivered_id 在 Stream 数组之上往前移动，表示当前消费组已经消费到哪条消息了。如果这是对的，那只需要开两个消费者组，消费者 1 在组1，消费者2在组2不就行了么。

  这个 last_delivered_id 有点像 Kafka 的 offset。

  2. 有点不同的观点

  Kafka 其实已经考虑去掉 ZK 了，而且适合高吞吐量的业务，不过不适合流式处理。专业的事，还是教给专业的来做好点。

  我之前用过 Redis 的 publish 和 subscribe 命令来做不同系统的数据同步。不过那个在用 Jedis 来订阅有坑，必须循环调用，不然会在一定时间后被 down 掉。 [1赞]

- pedro 2020-09-11 09:01:53

  每次想回答问题，看到 Kaito 的留言，顿时就有一种“眼前有景道不得，崔颢题诗在上头”的感觉。

- 刀斧手何在 2020-09-11 08:34:40

  Redis做消息队列的优势是高并发写入性能和水平扩展能力。最大的坑 我觉得就是Kaito同学说的第3点 数据可靠性

- MClink 2020-09-11 07:22:37

  发布/订阅"(publish/subscribe)模式，支持重复消费