基于Redis的Stream类型的完美消息队列解决方案

慕课牛

1 概述

Redis5.0带来了Stream类型。从字面上看是流类型，但其实从功能上看，应该是Redis对消息队列（MQ，Message Queue）的完善实现。用过Redis做消息队列的都了解，基于Reids的消息队列实现有很多种，例如：

- PUB/SUB，订阅/发布模式
- 基于List的 LPUSH+BRPOP 的实现
- 基于Sorted-Set的实现

每一种实现，都有典型的特点和问题，这个在 Redis 实现消息队列一文中有介绍。基于Redis实现消息队列http://www.hellokang.net/redis/message-queue-by-redis.html

Redis5.0中发布的Stream类型，也用来实现典型的消息队列。该Stream类型的出现，几乎满足了消息队列具备的全部内容，包括但不限于：

- 消息ID的序列化生成
- 消息遍历
- 消息的阻塞和非阻塞读取
- 消息的分组消费
- 未完成消息的处理
- 消息队列监控

消息队列有生产消息者和消费消息者，下面就体验一下Stream类型的精彩：

2 追加新消息，XADD，生产消息

XADD，命令用于在某个stream（流数据）中追加消息，演示如下：

```
127.0.0.1:6379> XADD memberMessage * user kang msg Hello
"1553439850328-0"
127.0.0.1:6379> XADD memberMessage * user zhong  msg nihao
"1553439858868-0"
```

其中语法格式为：

> XADD key ID field string [field string ...]

需要提供key，消息ID方案，消息内容，其中消息内容为key-value型数据。 ID，最常使用*，表示由Redis生成消息ID，这也是强烈建议的方案。 field string [field string], 就是当前消息内容，由1个或多个key-value构成。

上面的例子中，在memberMemsages这个key中追加了 `user kang msg Hello` 这个消息。Redis使用毫秒时间戳和序号生成了消息ID。此时，消息队列中就有一个消息可用了。

3 从消息队列中获取消息，XREAD，消费消息

XREAD，从Stream中读取消息，演示如下：

```
127.0.0.1:6379> XREAD streams memberMessage 0
1) 1) "memberMessage"
   2) 1) 1) "1553439850328-0"
         2) 1) "user"
            2) "kang"
            3) "msg"
            4) "Hello"
      2) 1) "1553439858868-0"
         2) 1) "user"
            2) "zhong"
            3) "msg"
            4) "nihao"
```

上面的命令是从消息队列memberMessage中读取所有消息。XREAD支持很多参数，语法格式为：

> XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]

其中：

- [COUNT count]，用于限定获取的消息数量
- [BLOCK milliseconds]，用于设置XREAD为阻塞模式，默认为非阻塞模式
- ID，用于设置由哪个消息ID开始读取。使用0表示从第一条消息开始。（本例中就是使用0）此处需要注意，消息队列ID是单调递增的，所以通过设置起点，可以向后读取。在阻塞模式中，可以使用$，表示最新的消息ID。（在非阻塞模式下$无意义）。

XRED读消息时分为阻塞和非阻塞模式，使用BLOCK选项可以表示阻塞模式，需要设置阻塞时长。非阻塞模式下，读取完毕（即使没有任何消息）立即返回，而在阻塞模式下，若读取不到内容，则阻塞等待。

一个典型的阻塞模式用法为：

```
127.0.0.1:6379> XREAD block 1000 streams memberMessage $
(nil)
(1.07s)
```

我们使用Block模式，配合$作为ID，表示读取最新的消息，若没有消息，命令阻塞！等待过程中，其他客户端向队列追加消息，则会立即读取到。

因此，典型的队列就是 XADD 配合 XREAD Block 完成。XADD负责生成消息，XREAD负责消费消息。

4 消息ID说明

XADD生成的 `1553439850328-0` ，就是Redis生成的消息ID，由两部分组成:时间戳-序号。时间戳是毫秒级单位，是生成消息的Redis服务器时间，它是个64位整型（int64）。序号是在这个毫秒时间点内的消息序号，它也是个64位整型。较真来说，序号可能会溢出，but真可能吗?

可以通过multi批处理，来验证序号的递增：

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> XADD memberMessage * msg one
QUEUED
127.0.0.1:6379> XADD memberMessage * msg two
QUEUED
127.0.0.1:6379> XADD memberMessage * msg three
QUEUED
127.0.0.1:6379> XADD memberMessage * msg four
```

```
QUEUED
127.0.0.1:6379> XADD memberMessage * msg five
QUEUED
127.0.0.1:6379> EXEC
1) "1553441006884-0"
2) "1553441006884-1"
3) "1553441006884-2"
4) "1553441006884-3"
5) "1553441006884-4"
```

由于一个redis命令的执行很快，所以可以看到在同一时间戳内，是通过序号递增来表示消息的。

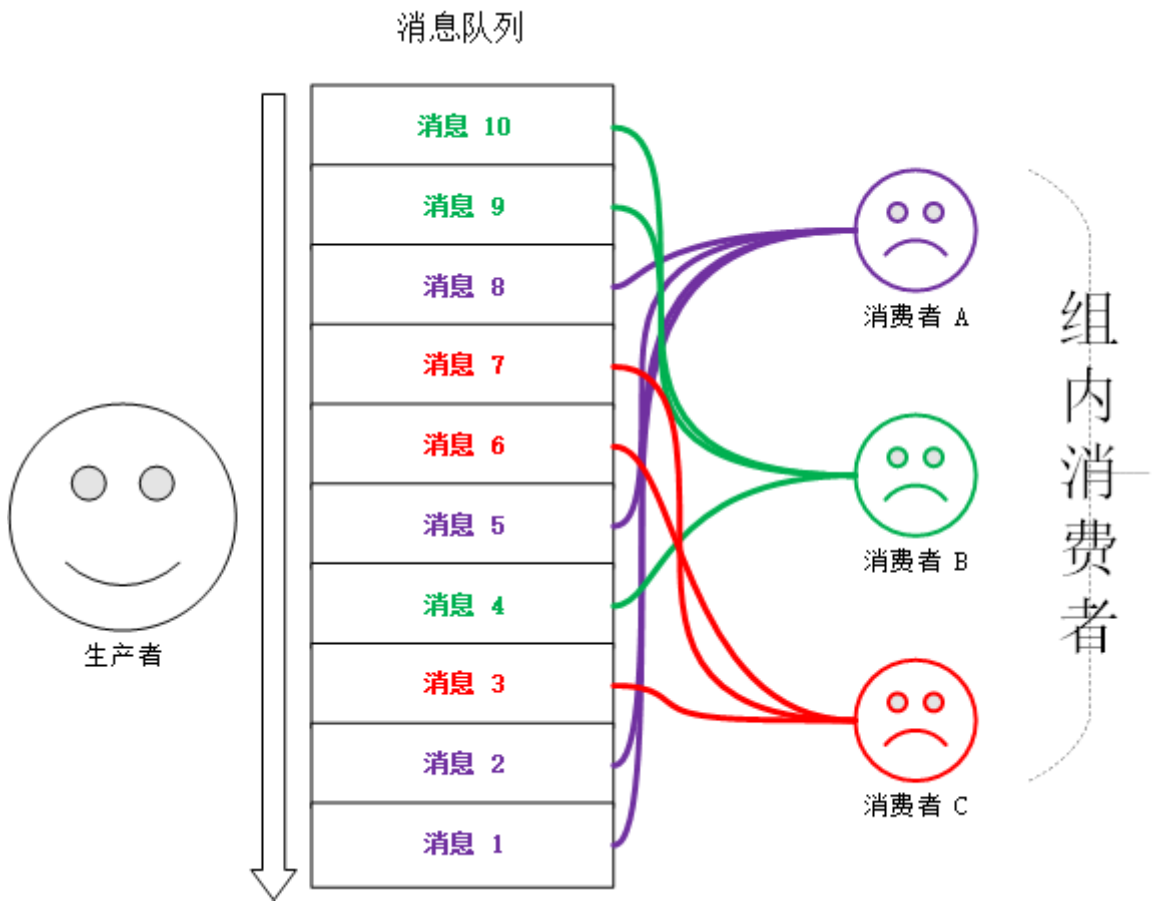
为了保证消息是有序的，因此Redis生成的ID是*单调递增*有序的。由于ID中包含时间戳部分，为了避免服务器时间错误而带来的问题（例如服务器时间延后了），Redis的每个Stream类型数据都维护一个latest_generated_id属性，用于记录最后一个消息的ID。若发现当前时间戳退后（小于latest_generated_id所记录的），则采用时间戳不变而序号递增的方案来作为新消息ID（这也是序号为什么使用int64的原因，保证有足够多的的序号），从而保证ID的单调递增性质。

强烈建议使用Redis的方案生成消息ID，因为这种时间戳+序号的单调递增的ID方案，几乎可以满足你全部的需求。但同时，记住ID是支持自定义的，别忘了！

5 消费者组模式，consumer group

当多个消费者（consumer）同时消费一个消息队列时，可以重复的消费相同的消息，就是消息队列中有10条消息，三个消费者都可以消费到这10条消息。

但有时，我们需要多个消费者配合协作来消费同一个消息队列，就是消息队列中有10条消息，三个消费者分别消费其中的某些消息，比如消费者A消费消息1、2、5、8，消费者B消费消息4、9、10，而消费者C消费消息3、6、7。也就是三个消费者配合完成消息的消费，可以在消费能力不足，也就是消息处理程序效率不高时，使用该模式。该模式就是消费者组模式。如下图所示：



消费者组模式的支持主要由两个命令实现：

- XGROUP，用于管理消费者组，提供创建组，销毁组，更新组起始消息ID等操作
- XREADGROUP，分组消费消息操作

进行演示，演示时使用5个消息，思路是：创建一个Stream消息队列，生产者生成5条消息。在消息队列上创建一个消费组，组内三个消费者进行消息消费：

```
# 生产者生成10条消息
127.0.0.1:6379> MULTI
127.0.0.1:6379> XADD mq * msg 1 # 生成一个消息：msg 1
127.0.0.1:6379> XADD mq * msg 2
127.0.0.1:6379> XADD mq * msg 3
127.0.0.1:6379> XADD mq * msg 4
127.0.0.1:6379> XADD mq * msg 5
127.0.0.1:6379> EXEC
 1) "1553585533795-0"
 2) "1553585533795-1"
 3) "1553585533795-2"
 4) "1553585533795-3"
 5) "1553585533795-4"

# 创建消费组 mqGroup
```

```
127.0.0.1:6379> XGROUP CREATE mq mqGroup 0 # 为消息队列 mq 创建消费组 mgGroup
OK

# 消费者A，消费第1条
127.0.0.1:6379> XREADGROUP group mqGroup consumerA count 1 streams mq > #消费组内消费者A，从
1) 1) "mq"
   2) 1) 1) "1553585533795-0"
         2) 1) "msg"
            2) "1"
# 消费者A，消费第2条
127.0.0.1:6379> XREADGROUP GROUP mqGroup consumerA COUNT 1 STREAMS mq >
1) 1) "mq"
   2) 1) 1) "1553585533795-1"
         2) 1) "msg"
            2) "2"
# 消费者B，消费第3条
127.0.0.1:6379> XREADGROUP GROUP mqGroup consumerB COUNT 1 STREAMS mq >
1) 1) "mq"
   2) 1) 1) "1553585533795-2"
         2) 1) "msg"
            2) "3"
# 消费者A，消费第4条
127.0.0.1:6379> XREADGROUP GROUP mqGroup consumerA count 1 STREAMS mq >
1) 1) "mq"
   2) 1) 1) "1553585533795-3"
         2) 1) "msg"
            2) "4"
# 消费者C，消费第5条
127.0.0.1:6379> XREADGROUP GROUP mqGroup consumerC COUNT 1 STREAMS mq >
1) 1) "mq"
   2) 1) 1) "1553585533795-4"
         2) 1) "msg"
            2) "5"
```

上面的例子中，三个在同一组 mpGroup 消费者A、B、C在消费消息时（消费者在消费时指定即可，不用预先创建），有着互斥原则，消费方案为，A->1, A->2, B->3, A->4, C->5。语法说明为：

`XGROUP CREATE mq mqGroup 0` ，用于在消息队列mq上创建消费组 mpGroup，最后一个参数0，表示该组从第一条消息开始消费。（意义与XREAD的0一致）。除了支持 `CREATE` 外，还支持 `SETID` 设置起始ID， `DESTROY` 销毁组， `DELCONSUMER` 删除组内消费者等操作。

`XREADGROUP GROUP mqGroup consumerA COUNT 1 STREAMS mq >` ，用于组 `mqGroup` 内消费者 `consumerA` 在队列 `mq` 中消费，参数 `>` 表示未被组内消费的起始消息，参数 `count 1` 表示获取一条。

语法与 XREAD 基本一致，不过是增加了组的概念。

可以进行组内消费的基本原理是，STREAM类型会为每个组记录一个最后处理（交付）的消息ID（last_delivered_id），这样在组内消费时，就可以从这个值后面开始读取，保证不重复消费。

以上就是消费组的基础操作。除此之外，消费组消费时，还有一个必须要考虑的问题，就是若某个消费者，消费了某条消息，但是并没有处理成功时（例如消费者进程宕机），这条消息可能会丢失，因为组内其他消费者不能再次消费到该消息了。下面继续讨论解决方案。

6 Pending 等待列表

为了解决组内消息读取但处理期间消费者崩溃带来的消息丢失问题， STREAM 设计了 Pending 列表，用于记录读取但并未处理完毕的消息。命令 XPENDIING 用来获消费组或消费内消费者的未处理完毕的消息。演示如下：

```
127.0.0.1:6379> XPENDING mq mqGroup # mpGroup的Pending情况
1) (integer) 5 # 5个已读取但未处理的消息
2) "1553585533795-0" # 起始ID
3) "1553585533795-4" # 结束ID
4) 1) 1) "consumerA" # 消费者A有3个
      2) "3"
   2) 1) "consumerB" # 消费者B有1个
      2) "1"
   3) 1) "consumerC" # 消费者C有1个
      2) "1"

127.0.0.1:6379> XPENDING mq mqGroup - + 10 # 使用 start end count 选项可以获取详细信息
1) 1) "1553585533795-0" # 消息ID
   2) "consumerA" # 消费者
   3) (integer) 1654355 # 从读取到现在经历了1654355ms，IDLE
   4) (integer) 5 # 消息被读取了5次，delivery counter
2) 1) "1553585533795-1"
   2) "consumerA"
   3) (integer) 1654355
   4) (integer) 4
# 共5个，余下3个省略 ...

127.0.0.1:6379> XPENDING mq mqGroup - + 10 consumerA # 在加上消费者参数，获取具体某个消费者的P
1) 1) "1553585533795-0"
   2) "consumerA"
   3) (integer) 1641083
```

```
    4) (integer) 5
# 共3个，余下2个省略 ...
```

每个Pending的消息有4个属性：

1. 消息ID
2. 所属消费者
3. IDLE，已读取时长
4. delivery counter，消息被读取次数

上面的结果我们可以看到，我们之前读取的消息，都被记录在Pending列表中，说明全部读到的消息都没有处理，仅仅是读取了。那如何表示消费者处理完毕了消息呢？使用命令 `XACK` 完成告知消息处理完成，演示如下：

```
127.0.0.1:6379> XACK mq mqGroup 1553585533795-0 # 通知消息处理结束，用消息ID标识
(integer) 1

127.0.0.1:6379> XPENDING mq mqGroup # 再次查看Pending列表
1) (integer) 4 # 已读取但未处理的消息已经变为4个
2) "1553585533795-1"
3) "1553585533795-4"
4) 1) 1) "consumerA" # 消费者A，还有2个消息处理
      2) "2"
   2) 1) "consumerB"
      2) "1"
   3) 1) "consumerC"
      2) "1"
127.0.0.1:6379>
```

有了这样一个Pending机制，就意味着在某个消费者读取消息但未处理后，消息是不会丢失的。等待消费者再次上线后，可以读取该Pending列表，就可以继续处理该消息了，保证消息的有序和不丢失。

此时还有一个问题，就是若某个消费者宕机之后，没有办法再上线了，那么就需要将该消费者Pending的消息，转义给其他的消费者处理，就是消息转移。请继续。

7 消息转移

消息转移的操作时将某个消息转移到自己的Pending列表中。使用语法 `XCLAIM` 来实现，需要设置组、转移的目标消费者和消息ID，同时需要提供IDLE（已被读取时长），只有超过这个时长，才能被转

移。演示如下：

```
# 当前属于消费者A的消息1553585533795-1，已经15907,787ms未处理了
127.0.0.1:6379> XPENDING mq mqGroup - + 10
1) 1) "1553585533795-1"
   2) "consumerA"
   3) (integer) 15907787
   4) (integer) 4

# 转移超过3600s的消息1553585533795-1到消费者B的Pending列表
127.0.0.1:6379> XCLAIM mq mqGroup consumerB 3600000 1553585533795-1
1) 1) "1553585533795-1"
   2) 1) "msg"
      2) "2"

# 消息1553585533795-1已经转移到消费者B的Pending中。
127.0.0.1:6379> XPENDING mq mqGroup - + 10
1) 1) "1553585533795-1"
   2) "consumerB"
   3) (integer) 84404 # 注意IDLE，被重置了
   4) (integer) 5 # 注意，读取次数也累加了1次
```

以上代码，完成了一次消息转移。转移除了要指定ID外，还需要指定IDLE，保证是长时间未处理的才被转移。被转移的消息的IDLE会被重置，用以保证不会被重复转移，以为可能会出现将过期的消息同时转移给多个消费者的并发操作，设置了IDLE，则可以避免后面的转移不会成功，因为IDLE不满足条件。例如下面的连续两条转移，第二条不会成功。

```
127.0.0.1:6379> XCLAIM mq mqGroup consumerB 3600000 1553585533795-1
127.0.0.1:6379> XCLAIM mq mqGroup consumerC 3600000 1553585533795-1
```

这就是消息转移。至此我们使用了一个Pending消息的ID，所属消费者和IDLE的属性，还有一个属性就是消息被读取次数，delivery counter，该属性的作用由于统计消息被读取的次数，包括被转移也算。这个属性主要用在判定是否为错误数据上。请继续看：

8 坏消息问题，Dead Letter，死信问题

正如上面所说，如果某个消息，不能被消费者处理，也就是不能被XACK，这是要长时间处于Pending列表中，即使被反复的转移给各个消费者也是如此。此时该消息的delivery counter就会累加（上一节的例子可以看到），当累加到某个我们预设的临界值时，我们就认为是坏消息（也叫死信，DeadLetter，无法投递的消息），由于有了判定条件，我们将坏消息处理掉即可，删除即可。删除一个消息，使用 XDEL 语法，演示如下：

```
# 删除队列中的消息
127.0.0.1:6379> XDEL mq 1553585533795-1
(integer) 1
# 查看队列中再无此消息
127.0.0.1:6379> XRANGE mq - +
1) 1) "1553585533795-0"
   2) 1) "msg"
      2) "1"
2) 1) "1553585533795-2"
   2) 1) "msg"
      2) "3"
```

注意本例中，并没有删除Pending中的消息因此你查看Pending，消息还会在。可以执行 XACK 标识其处理完毕!

9 信息监控，XINFO

Stream提供了XINFO来实现对服务器信息的监控，可以查询：

查看队列信息

```
127.0.0.1:6379> Xinfo stream mq
 1) "length"
 2) (integer) 7
 3) "radix-tree-keys"
 4) (integer) 1
 5) "radix-tree-nodes"
 6) (integer) 2
 7) "groups"
 8) (integer) 1
 9) "last-generated-id"
10) "1553585533795-9"
11) "first-entry"
12) 1) "1553585533795-3"
    2) 1) "msg"
       2) "4"
13) "last-entry"
14) 1) "1553585533795-9"
    2) 1) "msg"
       2) "10"
```

消费组信息

```
127.0.0.1:6379> Xinfo groups mq
1) 1) "name"
   2) "mqGroup"
   3) "consumers"
   4) (integer) 3
   5) "pending"
   6) (integer) 3
   7) "last-delivered-id"
   8) "1553585533795-4"
```

消费者组成员信息

```
127.0.0.1:6379> XINFO CONSUMERS mq mqGroup
1) 1) "name"
   2) "consumerA"
   3) "pending"
   4) (integer) 1
   5) "idle"
   6) (integer) 18949894
2) 1) "name"
   2) "consumerB"
   3) "pending"
   4) (integer) 1
   5) "idle"
   6) (integer) 3092719
3) 1) "name"
   2) "consumerC"
   3) "pending"
   4) (integer) 1
   5) "idle"
   6) (integer) 23683256
```

至此，消息队列的操作说明大体结束！

10 命令一览

|命令|说明| |:---|:---| |XACK|结束Pending| |XADD|生成消息| |XCLAIM|消息转移| |XDEL|删除消息| |XGROUP|消费组管理| |XINFO|得到消费组信息| |XLEN|消息队列长度| |XPENDING|Pending列表| |XRANGE|获取消息队列中消息| |XREAD|消费消息| |XREADGROUP|分组消费消息| |XREVRANGE|逆序获取消息队列中消息| |XTRIM|消息队列容量|

11 Stream数据结构，RadixTree，基数树

`Stream` 是基于 `RadixTree` 数据结构实现的。另立话题讨论。基数树，http://www.hellokang.net/algorithm/radix-tree.html

12 相关产品

很多成熟的MQ产品：

* Disque，disquedurinterne.net/

* Kafka，Apache Kafka

* ActiveMQ，Apache ActiveMQ ™ -- Index

* RockMQ，Apache RocketMQ

* RabbitMQ，Messaging that just works

* ZeroMQ，Distributed Messaging - zeromq

请关注：小韩说课，微信公众号！获取最新内容

编辑于 2019-03-26 23:14