Key-value Project Task Report

s188026 Yifei Liu

Catalog

1. Introduction

1.1 Scenario – Lighting Deal

11.11 (11th of November) of each year is E-commerce shopping carnival in China. In each electronic commerce platform, there will be many Lighting Deal activities. For the huge amount of population in China, we need solve the high concurrency when many people snap up items at the same time. Therefore, we can do like this:

1. Before Lighting Deal, synchronize product inventory from database to Redis.
2. During Lighting Deal, we decrease inventory and publish the data of orders by Pub/Sub.
3. The subscriber of order's data handle the order.

1.2 Features Analysis in simple case:

1. **Data type**: For the remaining quantity of goods, we only need **String**. But for users, we can use **Set** to make sure that one user can only attempt once in the Lighting Deal of each type of goods.
2. **Transaction**: Before this point, the high concurrency of order will happen at the same time, so it may cause the problem of over sell (make the remaining inventory be negative). In order to solve this problem, we can use WATCH to monitor inventory. WATCH will provide CAS (compare and swap). If at least one of the monitored keys is modified before the EXEC command is executed, the entire transaction will be rolled back without performing any actions, thereby ensuring atomicity.

However, it may cause **another problem – *remaining inventory***. Due to the **optimistic locks** (CAS), a large number of simultaneous concurrent requests would fail, and the inventory is still left in the end.

3. **Lua script**: Since Redis 2.6, it embeds with support of Lua environment, so we can use Lua script to **solve the problem above and keep the atomicity at the same time**. **Because script**

will be executed in one line command while using watch - transaction will execute in multi commands. Also, it can reduce the frequency of linking to Redis.

4. **Message Queue:**

- **Sub/Pub**: Publish the data of order. The subscriber handle with the order.
- **Stream**: Due to message loss, limit for message accumulation and data persistence not supported in Sub/Pub, stream can be used in such activity to solve these problems.

5. **Redis Cluster** & **Hash slot** & **Hash tag**: Except for the features above, when we want to withstand peak pressure of the data generated, we can also use Redis Cluster. If we divide hash slots manually, we must divide all of them otherwise Redis will not work. We can also use hash tag to ensure some data are in the same hash slot.

2. Operations (simple version)

2.1 Preparation

Add inventory amount into Redis.

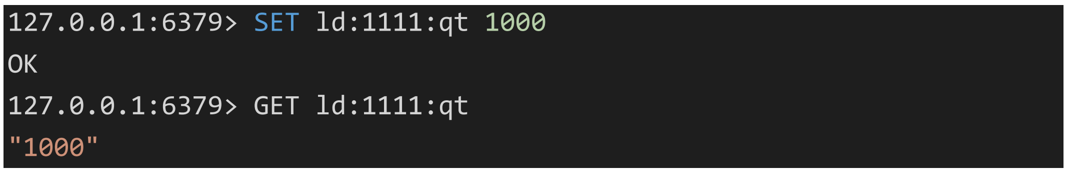
- **ld –** Lighting Deal, the name of the activity this time
- **product_id** – the id of product .
- **qt** – quantity
- **amount** – amount of inventory

```
SET ld:product_id:qt amount
SET ld:1111:qt 1000
```

We also need check the remaining amount:

1. nil -> hasn’t started
2. 0 -> finished

```
GET ld:product_id:qt
GET ld:1111:qt
```



```
127.0.0.1:6379> SET ld:1111:qt 1000
OK
127.0.0.1:6379> GET ld:1111:qt
"1000"
```

2.2 Data

When a user snapped up an item, insert user’s information and decrease the amount of inventory.

```
DECR ld:product_id:qt
DECR ld:1111:qt
SADD ld:product_id:user user_id
SADD ld:1111:user 9527
```

If someone cancel the order:

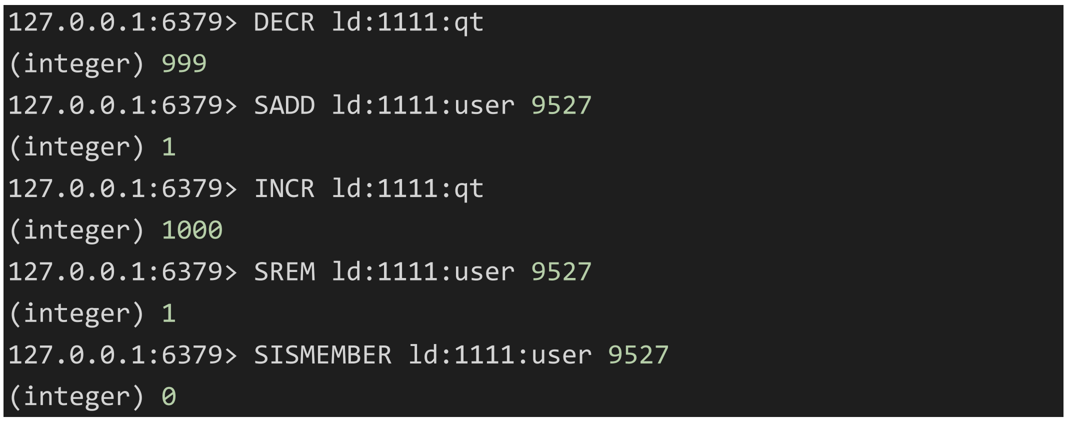
```
INCR ld:product_id:qt
INCR ld:1111:qt
SREM ld:product_id:user user_id
SREM ld:1111:user 9527
```

Check whether we have had this user in set.

```
SISMEMBER ld:product_id:user user_id
SISMEMBER ld:1111:user 9527
```

Other operations in set maybe used here:

```
SUNIONSTORE sumset product1_set product2_set (…)
SUNIONSTORE sumset ld:1111:user ld:1112:user
SCARD ld:product_id:user
SCARD ld:1111:user
SMEMBERS ld:product_id:user
SMEMBERS ld:1111:user
```

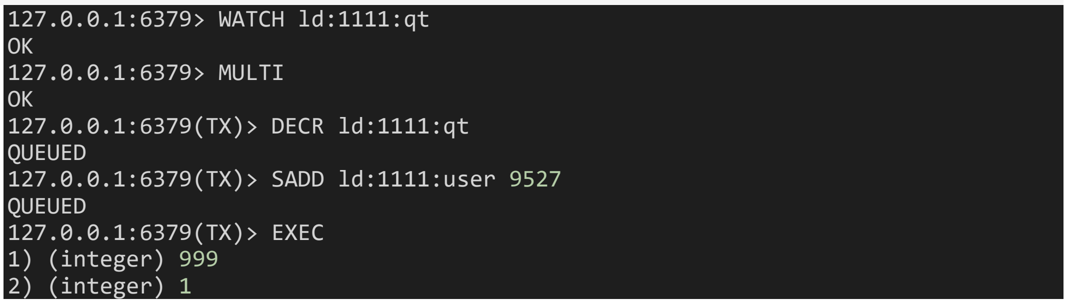


```
127.0.0.1:6379> DECR ld:1111:qt
(integer) 999
127.0.0.1:6379> SADD ld:1111:user 9527
(integer) 1
127.0.0.1:6379> INCR ld:1111:qt
(integer) 1000
127.0.0.1:6379> SREM ld:1111:user 9527
(integer) 1
127.0.0.1:6379> SISMEMBER ld:1111:user 9527
(integer) 0
```

2.3 WHATCH

Before using WATCH, although we can check whether the value of remain quantity (ld:1111:qt) is smaller than 1 in the code of back-end, but when high concurrency the remaining quantity may suddenly be negative at once.
Use WATCH (optimistic locks) of Redis transaction can solve the problem of over sell.

```
WATCH ld:product_id:qt
MULTI
DECR ld:product_id:q
SADD ld:product_id:user user_id
EXEC
```



```
127.0.0.1:6379> WATCH ld:1111:qt
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> DECR ld:1111:qt
QUEUED
127.0.0.1:6379(TX)> SADD ld:1111:user 9527
QUEUED
127.0.0.1:6379(TX)> EXEC
1) (integer) 999
2) (integer) 1
```

From now on, it will have new problem. It may lead many people fail to operate, so it will be replaced in next point (2.5).

2.4 Final version to solve problems above (Lua + Java)

script.lua:

- Check user exists in the set by sismember
- Check remaining inventory to solve over sell.
- Each time execute the whole script in one command which will not let others fail when this

user fail. Solve the problem of optimistic lock.

```
local user_id=KEYS[1];
local product_id=KEYS[2];
local qtKey="ld:"..product_id..":qt";
local usersKey="ld:"..product_id..":user';
local userExists=redis.call("sismember", usersKey, user_id);
if tonumber(userExists)==1 then
        return 2;
end
local num= redis.call("get", qtKey);
if tonumber(num)<=0 then
        return 0;
else
        redis.call("decr", qtKey);
        redis.call("sadd", usersKey,user_id);
        return 1;
end
```

```
127.0.0.1:6379> eval "local user_id=KEYS[1]; local
product_id=KEYS[2];local qtKey='ld:'..product_id..':qt';local
usersKey='ld:'..product_id..':user'; local
userExists=redis.call('sismember', usersKey, user_id);if
tonumber(userExists)==1 then return 2;end local num= redis.cal('get',
qtKey);if tonumber(num)<=0 then return 0; else redis.call('decr', qtKey);
redis.call('sadd', usersKey,user_id);end" 2 9527 1111

(integer) 2

127.0.0.1:6379> eval "local user_id=KEYS[1]; local
product_id=KEYS[2];local qtKey='ld:'..product_id..':qt';local
usersKey='ld:'..product_id..':user'; local
userExists=redis.call('sismember', usersKey, user_id);if
tonumber(userExists)==1 then return 2;end local num= redis.call('get',
qtKey);if tonumber(num)<=0 then return 0; else redis.call('decr', qtKey);
redis.call('sadd', usersKey,user_id); return 1; end" 2 9525 1111

(integer) 1
```

Depending on the different return value in lua script, we can handle the result in back-end.

```
// Load script
String script = jedis.scriptLoad(scriptAbove);
// Execute script
Object result= jedis.evalsha(script, 2, uid, prodid);
// Handle return value
String returnValue=String.valueOf(result);
if ("0".equals( returnValue ) ) {
        System.err.println("Sold out");
}else if("1".equals( returnValue )) {
        System.out.println("Snap up sucessfully");
}else if("2".equals( returnValue )) {
        System.err.println("This user has bought one");
}else{
        System.err.println("Exception");
}
```

2.5 Sub/Pub

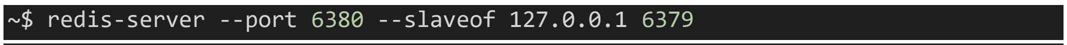
Subscribe a channel which will publish the order data to subscriber server which will be used to handle data of order.

a) In subscriber client

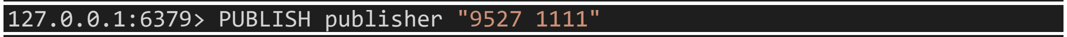
```
SUBSCRIBE publisher
```

b) In publish client (publish which user buy which product)

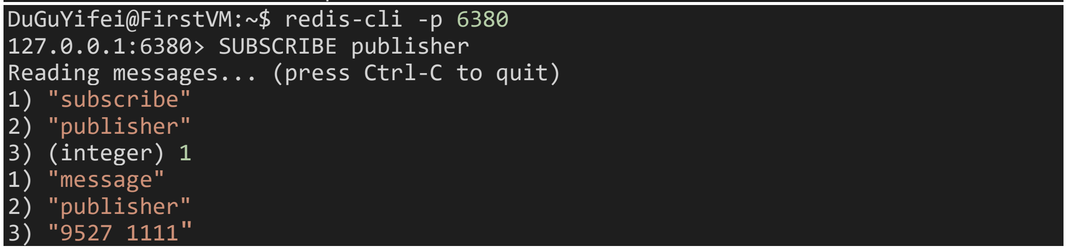
```
PUBLISH publisher “user_id product_id”
PUBLISH publisher “9527 1111”
```



```
~$ redis-server --port 6380 --slaveof 127.0.0.1 6379
```



```
127.0.0.1:6379> PUBLISH publisher "9527 1111"
```

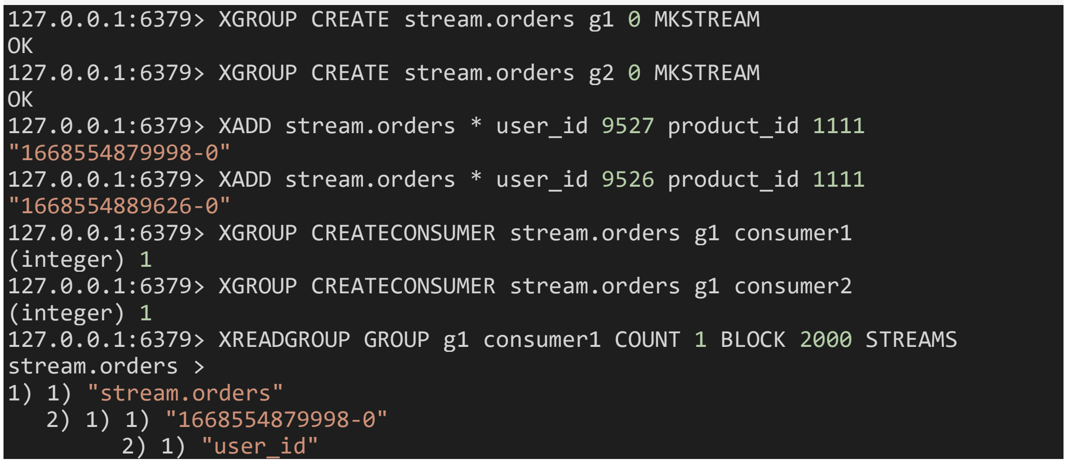


```
DuGuYifei@FirstVM:~$ redis-cli -p 6380
127.0.0.1:6380> SUBSCRIBE publisher
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "publisher"
3) (integer) 1
1) "message"
2) "publisher"
3) "9527 1111"
```

2.6 Stream

Due to the reason mentioned in 1.2.5, we can also use Stream messages.
The codes below respectively are creating the consumer group, add new order information and send message to the stream when user snap up successfully.

```
XGROUP CREATE streamKey groupname 0 MKSTREAM
XGROUP CREATE stream.orders g1 0 MKSTREAM
-----------------------------------------------------------
XADD key ID field value
XADD stream.orders * user_id 9527 product_id 1111
XADD stream.orders * user_id 9526 product_id 1111
----------------------------------------------------------
XREADGROUP GROUP g1 consumer1 COUNT 1 BLOCK 2000 STREAMS stream.orders >
XACK stream.orders g1 ID
```



```
127.0.0.1:6379> XGROUP CREATE stream.orders g1 0 MKSTREAM
OK
127.0.0.1:6379> XGROUP CREATE stream.orders g2 0 MKSTREAM
OK
127.0.0.1:6379> XADD stream.orders * user_id 9527 product_id 1111
"1668554879998-0"
127.0.0.1:6379> XADD stream.orders * user_id 9526 product_id 1111
"1668554889626-0"
127.0.0.1:6379> XGROUP CREATECONSUMER stream.orders g1 consumer1
(integer) 1
127.0.0.1:6379> XGROUP CREATECONSUMER stream.orders g1 consumer2
(integer) 1
127.0.0.1:6379> XREADGROUP GROUP g1 consumer1 COUNT 1 BLOCK 2000 STREAMS
stream.orders >
1) 1) "stream.orders"
   2) 1) 1) "1668554879998-0"
         2) 1) "user_id"
```

```
            2) "9527"
            3) "product_id"
            4) "1111"
127.0.0.1:6379> XREADGROUP GROUP g1 consumer2 COUNT 1 BLOCK 2000 STREAMS
stream.orders >
1) 1) "stream.orders"
   2) 1) 1) "1668554889626-0"
         2) 1) "user_id"
            2) "9526"
            3) "product_id"
            4) "1111"
127.0.0.1:6379> XREADGROUP GROUP g2 consumer1 COUNT 1 BLOCK 2000 STREAMS
stream.orders >
1) 1) "stream.orders"
   2) 1) 1) "1668554879998-0"
         2) 1) "user_id"
            2) "9527"
            3) "product_id"
            4) "1111"
127.0.0.1:6379> XREADGROUP GROUP g2 consumer1 COUNT 1 BLOCK 2000 STREAMS
stream.orders >
1) 1) "stream.orders"
   2) 1) 1) "1668554889626-0"
         2) 1) "user_id"
            2) "9526"
            3) "product_id"
            4) "1111"
127.0.0.1:6379> XREADGROUP GROUP g2 consumer2 COUNT 1 BLOCK 2000 STREAMS
stream.orders >
(nil)
(2.05s)
127.0.0.1:6379> XREADGROUP GROUP g2 consumer1 COUNT 1 BLOCK 2000 STREAMS
stream.orders 0
1) 1) "stream.orders"
   2) 1) 1) "1668554879998-0"
         2) 1) "user_id"
            2) "9527"
            3) "product_id"
            4) "1111"
127.0.0.1:6379> XACK stream.orders g2 1668554879998-0
(integer) 1
127.0.0.1:6379> XREADGROUP GROUP g2 consumer1 COUNT 1 BLOCK 2000 STREAMS
stream.orders 0
1) 1) "stream.orders"
   2) 1) 1) "1668554889626-0"
         2) 1) "user_id"
            2) "9526"
            3) "product_id"
            4) "1111"
```

2.7 Redis Cluster & Hash slot & Hash tag

For all the keys showed above, there is a product_id inside. So we can do like this: `ld:{product_id}:qt` and `ld:{product_id}:user_id` to let these data in the same slot so that in same node.
Also we will set the conf file: `cluster-enabled yes` to open the Redis cluster.
e.g.

```
port 6379
dir "/redisConfig"
logfile "clusterMaster1.log"
cluster-enabled yes
cluster-config-file nodes-6379.conf
```

Use command to build communication between nodes:

```
cluster meet {ip} {port}
cluster meet 172.17.0.3 6380
cluster meet 172.17.0.4 6381
cluster meet 172.17.0.5 6382
cluster meet 172.17.0.6 6383
cluster meet 172.17.0.7 6384
```

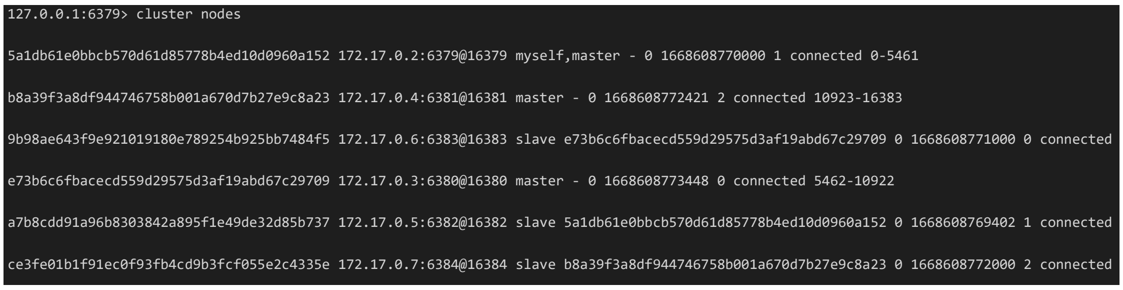
We can divide hash slot when open Redis client:

```
redis-cli -p 6379 cluster addslots $(seq 0 5461)
redis-cli -p 6380 cluster addslots $(seq 5462 10922)
redis-cli -p 6381 cluster addslots $(seq 10923 16383)
```

We only create 3 master nodes here. So other nodes we create can be used as slave:

```
127.0.0.1:6382> cluster replicate ****************** (The Instance ID of mater node)
```

If want to check the clusters situation or instance ID, we can use this command: cluster nodes.



```
127.0.0.1:6379> cluster nodes

5a1db61e0bbcb570d61d85778b4ed10d0960a152 172.17.0.2:6379@16379 myself,master - 0 1668608770000 1 connected 0-5461

b8a39f3a8df944746758b001a670d7b27e9c8a23 172.17.0.4:6381@16381 master - 0 1668608772421 2 connected 10923-16383

9b98ae643f9e921019180e789254b925bb7484f5 172.17.0.6:6383@16383 slave e73b6c6fbacecd559d29575d3af19abd67c29709 0 1668608771000 0 connected

e73b6c6fbacecd559d29575d3af19abd67c29709 172.17.0.3:6380@16380 master - 0 1668608773448 0 connected 5462-10922

a7b8cdd91a96b8303842a895f1e49de32d85b737 172.17.0.5:6382@16382 slave 5a1db61e0bbcb570d61d85778b4ed10d0960a152 0 1668608769402 1 connected

ce3fe01b1f91ec0f93fb4cd9b3fcf055e2c4335e 172.17.0.7:6384@16384 slave b8a39f3a8df944746758b001a670d7b27e9c8a23 0 1668608772000 2 connected
```

Now we have 3 master nodes and 3 slave nodes, and divide hash slots into three master nodes. During the high concurrency of orders, for example, 3000 accesses to Redis cache per second will almost evenly distributed to three master nodes as 1000 accesses. We can also add new nodes if we want during the server working, it will be really helpful for the business of Lighting Deal.