课程主题

Redis数据类型使用场景与Redis高级用法

课程目标

1. 掌握Redis数据类型使用场景
2. 理解掌握Redis的补充数据类型
3. 理解Redis的消息模式
4. 掌握 Redis Stream 数据类型
5. 掌握Redis事务原理及用法（弱事务）
6. 理解lua概念，能够使用Redis和lua整合使用
7. 理解掌握Redis分布式锁，并清楚其优缺点

知识要点

AOF重写机制

重写并不是读原有文件，而是读取内存数据，转化为指令

set s1 33

set s1 66

set s1 77

优化前文件是65M

优化后 set s1 77

优化后文件 33M

第二次优化 文件达到66M

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

混合持久化

```
of-use-rdb-preamble yes
```

前面文件以RDB文件格式存储，后面以指令存储

第一次开启AOF时候，默认会走AOF重写。 先打开混合持久化模式，再打开AOF

Sentinel 启动过程

1. 初始化服务器
2. 将普通的redis服务器代码替换成sentinel专用代码
3. 初始化sentinel 状态
4. 根据配置文件，初始化sentinel监控的主节点列表
5. 创建连向主服务器的网络连接。

一 Redis基本数据类型使用场景

String JSONstring 缓存功能 计数器 共享用户Session 分布式锁 setnx

List 存储列表结构 -粉丝列表，文章评论列表 lrange 基于Redis实现简单的高性能分页

简单的消息队列

Hash 爆品 商品 jiange

秒杀仓库 xx秒杀商品 商品1 库存量

Set 基于Redis进行全局的Set去重 共同好友 你可能认识

Zset 排行榜 有权重的消息队列 热搜 前面是名称 后面试热度值

二 Redis 的补充数据类型

BitMap

BitMap 就是通过一个 bit 位来表示某个元素对应的值或者状态, 其中的 key 就是对应元素本身，实际上底层也是通过对字符串的操作来实现。Redis 从 2.2 版本之后新增了setbit, getbit, bitcount 等几个 bitmap 相关命令。虽然是新命令，但是本身都是对字符串的操作，我们先来看看语法：

```
SETBIT key offset value
```

其中 offset 必须是数字，value 只能是 0 或者 1

bitop add

bitop not

存储一亿用户 12.5M

```
127.0.0.1:6379> setbit k1 5 1
```

```
(integer) 0
127.0.0.1:6379> getbit k1 5 1
(integer) 1
127.0.0.1:6379> getbit k1 4 0
(integer) 0
127.0.0.1:6379> bitcount k1
(integer) 1
127.0.0.1:6379> setbit k1 3 1
(integer) 0
127.0.0.1:6379> bitcount k1
(integer) 2
127.0.0.1:6379> setbit "200522:active"  67 1
(integer) 0
127.0.0.1:6379> setbit "200522:active" 78 1
(integer) 0
```

其中 offset 必须是数字，value 只能是 0 或者 1

通过 bitcount可以很快速的统计，比传统的关系型数据库效率高很多

1、比如统计年活跃用户数量

用户的ID作为offset，当用户在一年内访问过网站，就将对应offset的bit值设置为“1”；

通过bitcount 来统计一年内访问过网站的用户数量

2、比如统计三天内活跃用户数量

时间字符串作为key，比如 “200522:active“ ；

用户的ID就可以作为offset，当用户访问过网站，就将对应offset的bit值设置为“1”；

统计三天的活跃用户，通过bitop or 获取一周内访问过的用户数量

3、连续三天访问的用户数量 bitop and

4、三天内没有访问的用户数量 bitop not

5、统计在线人数 设置在线key：“online：active”，当用户登录时，通过setbit设置

bitmap的优势，以统计活跃用户为例

每个用户id占用空间为1bit，消耗内存非常少，存储1亿用户量只需要12.5M

bitmap 可以做布隆过滤器 确认访问值是否存在 只要在布隆过滤器里值是0 后面的服务就别访问了

HyperLogLog （2.8）

1.基于bitmap 计数

2. 基于概率基数计数 0.87

这个数据结构的命令有三个：PFADD、PFCOUNT、PFMERGE

内部编码主要分稀疏型和密集型

用途：记录网站IP注册数，每日访问的IP数，页面实时UV、在线用户人数

局限性：只能统计数量，没有办法看具体信息

```
127.0.0.1:6379> pfadd h1 b
(integer) 1
127.0.0.1:6379> pfadd h1 a
(integer) 0
127.0.0.1:6379> pfcount h1
(integer) 2
127.0.0.1:6379> pfadd h1 c
(integer) 1
127.0.0.1:6379> pfadd h2 a
(integer) 1
127.0.0.1:6379> pfadd h3 d
(integer) 1
127.0.0.1:6379> pfmerge h3 h1 h2
OK
127.0.0.1:6379> pfcount h3
(integer) 4
```

Geospatial （3.2）

底层数据结构 Zset GEOADD GEODIST GEOHASH GEOPOP GEOPADUIS GEORADIUSBYMEMBER

可以用来保存地理位置，并作位置距离计算或者根据半径计算位置等。有没有想过用Redis来实现附近的人？或者计算最优地图路径？Geo本身不是一种数据结构，它**本质上还是借助于Sorted Set（ZSET）**

GEOADD key 经度 维度 名称

把某个具体的位置信息（经度，纬度，名称）添加到指定的key中，数据将会用一个sorted set存储，以便稍后能使用GEORADIUS和GEORADIUSBYMEMBER命令来根据半径来查询位置信息。

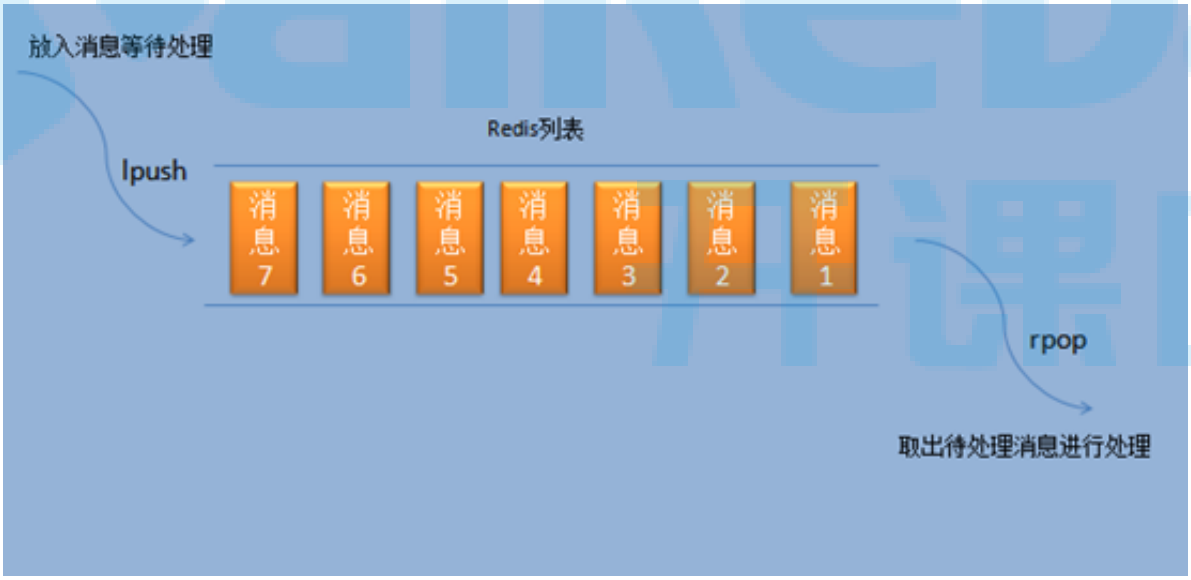
```
127.0.0.1:6379> GEOADD cities 116.404269 39.91582 "beijing" 121.478799
31.235456 "shanghai"
(integer) 2
127.0.0.1:6379> ZRANGE cities 0 -1
1) "shanghai"
2) "beijing"
127.0.0.1:6379> ZRANGE cities 0 -1 WITHSCORES
1) "shanghai"
2) "4054803475356102"
3) "beijing"
4) "4069885555377153"
127.0.0.1:6379> GEODIST cities beijing shanghai km
"1068.5677"
127.0.0.1:6379> GEOPOS cities beijing shanghai
1) 1) "116.40426903963088989"
```

```
   2) "39.91581928642635546"
2) 1) "121.47879928350448608"
   2) "31.23545629441388627"
127.0.0.1:6379> GEOADD cities 120.165036 30.278973 hangzhou
(integer) 1
127.0.0.1:6379> GEORADIUS cities 120 30 500 km
1) "hangzhou"
2) "shanghai"
127.0.0.1:6379> GEORADIUSBYMEMBER cities shanghai 200 km
1) "hangzhou"
2) "shanghai"
127.0.0.1:6379> ZRANGE cities 0 -1
1) "hangzhou"
2) "shanghai"
3) "beijing"
127.0.0.1:6379>
```

三 Redis 消息模式

队列模式

使用**list**类型的**lpush**和**rpop**实现消息队列



注意事项：

- 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。
- 可以使用**brpop**命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null

缺点：

做消费者确认ACK麻烦，不能保证消费者消费消息后是否成功处理的问题（宕机或处理异常等），通常需要维护一个Pending列表，保证消息处理确认。

不能做广播模式，如pub/sub，消息发布/订阅模型

不能重复消费，一旦消费就会被删除

不支持分组消费

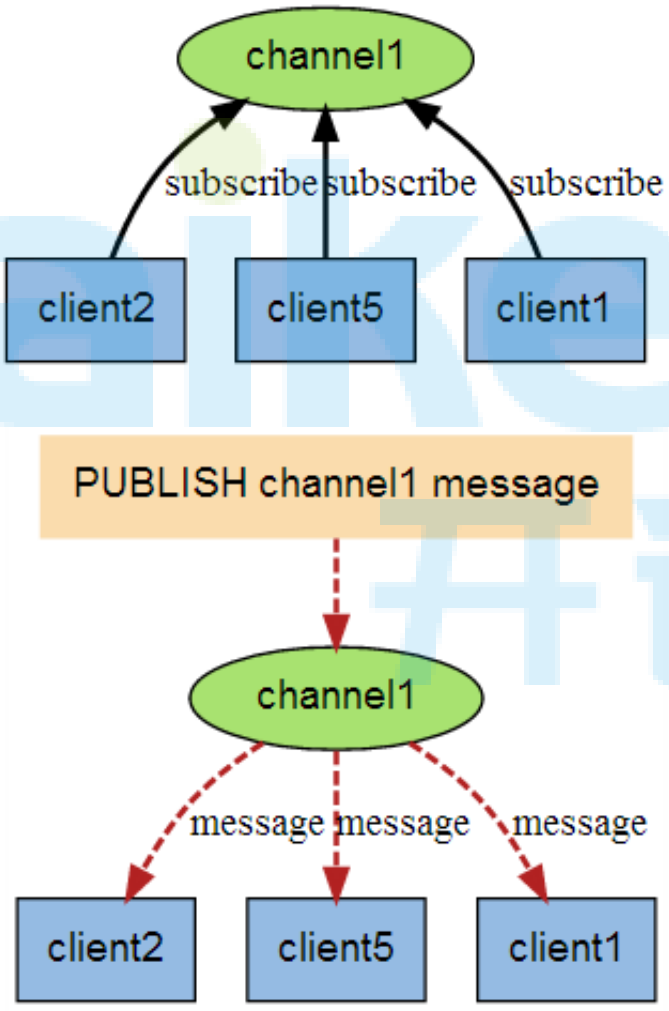
发布订阅模式

SUBSCRIBE，用于订阅信道

PUBLISH，向信道发送消息

UNSUBSCRIBE，取消订阅

此模式允许生产者只生产一次消息，由中间件负责将消息复制到多个消息队列，每个消息队列由对应的消费组消费。



优点

典型的广播模式，一个消息可以发布到多个消费者

多信道订阅，消费者可以同时订阅多个信道，从而接收多类消息

消息即时发送，消息不用等待消费者读取，消费者会自动接收到信道发布的消息

缺点

消息一旦发布，不能接收。换句话就是发布时若客户端不在线，则消息丢失，不能寻回

不能保证每个消费者接收的时间是一致的

若消费者客户端出现消息积压，到一定程度，会被强制断开，导致消息意外丢失。通常发生在消息的生产远大于消费速度时

可见，Pub/Sub 模式不适合做消息存储，消息积压类的业务，而是擅长处理广播，即时通讯，即时反馈的业务。

基于Sorted-Set的实现

Sortes Set(有序列表)，类似于java的SortedSet和HashMap的结合体，一方面它是一个set，保证内部value的唯一性，另一方面它可以给每个value赋予一个score，代表这个value的排序权重。内部实现是“跳跃表”。

有序集合的方案是在自己确定消息顺ID时比较常用，使用集合成员的Score来作为消息ID，保证顺序，还可以保证消息ID的单调递增。通常可以使用时间戳+序号的方案。确保了消息ID的单调递增，利用SortedSet的依据Score排序的特征，就可以制作一个有序的消息队列了。

优点

就是可以自定义消息ID，在消息ID有意义时，比较重要。

缺点

缺点也明显，不允许重复消息（因为是集合），同时消息ID确定有错误会导致消息的顺序出错。

四 Redis Stream

Redis 5.0 全新的数据类型：streams，官方把它定义为：以更抽象的方式建模日志的数据结构。Redis的streams主要是一个**append only**（AOF）的数据结构，至少在概念上它是一种在内存中表示的抽象数据类型，只不过它们实现了更强大的操作，以克服日志文件本身的限制。

如果你了解MQ，那么可以把streams当做基于内存的MQ。如果你还了解kafka，那么甚至可以把streams当做基于内存的kafka。listpack存储信息，Rax组织listpack 消息链表

listpack是对ziplist的改进，它比ziplist少了一个定位最后一个元素的属性

另外，这个功能有点类似于redis以前的Pub/Sub，但是也有基本的不同：

- streams支持多个客户端（消费者）等待数据（Linux环境开多个窗口执行XREAD即可模拟），并且每个客户端得到的是完全相同的数据。
- Pub/Sub是发送忘记的方式，并且不存储任何数据；**而streams模式下，所有消息被无限期追加在streams中，除非用于显式执行删除（XDEL）**。 XDEL 只做一个标记位 其实信息和长度还在
- streams的Consumer Groups也是Pub/Sub无法实现的控制方式。

streams数据结构

它主要有消息、生产者、消费者、消费组4组成

streams数据结构本身非常简单，但是streams依然是Redis到目前为止最复杂的类型，其原因是实现的一些额外的功能：一系列的阻塞操作允许消费者等待生产者加入到streams的新数据。另外还有一个称为Consumer Groups的概念，Consumer Group概念最先由kafka提出，Redis有一个类似实现，和kafka的Consumer Groups的目的是一样的：允许一组客户端协调消费相同的信息流！

发布消息

```
127.0.0.1:6379> xadd mystream * message apple
"1589994652300-0"
127.0.0.1:6379> xadd mystream * message orange
"1589994679942-0"
```

读取消息

```
127.0.0.1:6379> xrange mystream - +
1) 1) "1589994652300-0"
   2) 1) "message"
      2) "apple"
2) 1) "1589994679942-0"
   2) 1) "message"
      2) "orange"
```

阻塞读取

```
xread block 0 streams mystream $
```

发布新消息

```
127.0.0.1:6379> xadd mystream * message strawberry
```

创建消费组

```
127.0.0.1:6379> xgroup create mystream mygroup1 0
OK
127.0.0.1:6379> xgroup create mystream mygroup2 0
OK
```

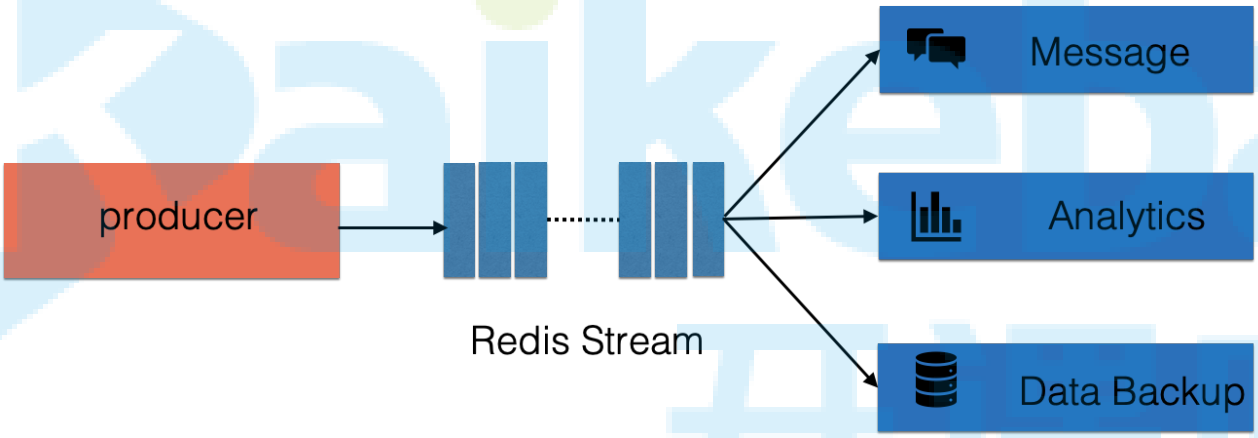
通过消费组读取消息

```
127.0.0.1:6379> xreadgroup group mygroup1 zange count 2 streams mystream >
1) 1) "mystream"
   2) 1) 1) "1589994652300-0"
         2) 1) "message"
            2) "apple"
```

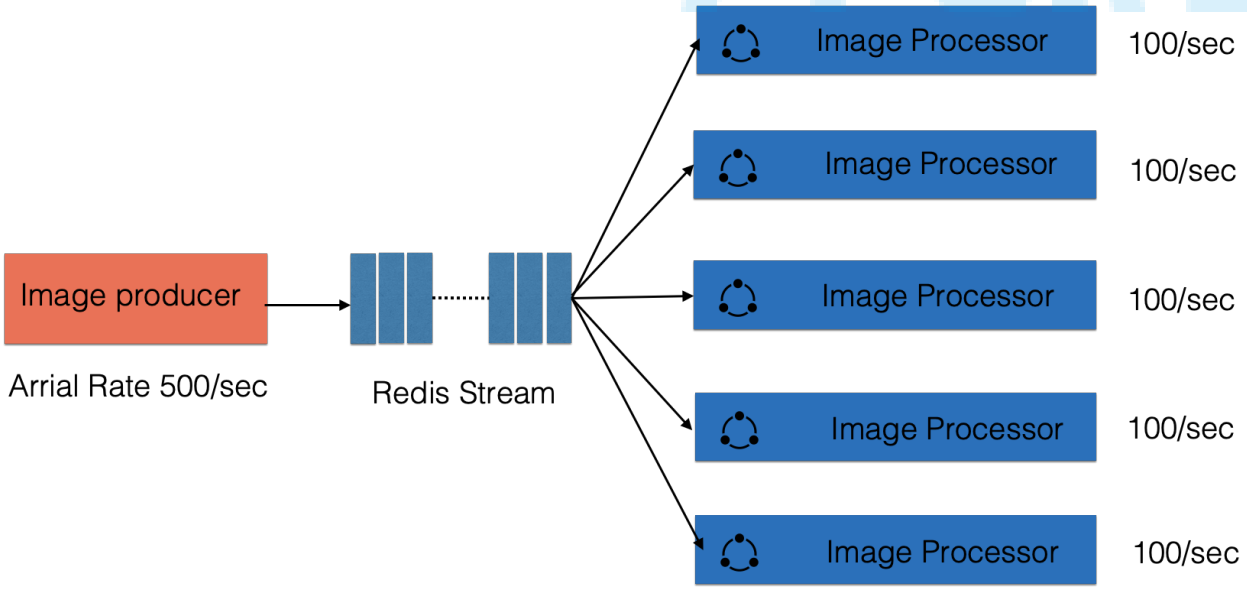
```
      2) 1) "1589994679942-0"
         2) 1) "message"
            2) "orange"
127.0.0.1:6379> xreadgroup group mugroup1 tuge count 2 streams mystream >
1) 1) "mystream"
   2) 1) 1) "1589995171242-0"
         2) 1) "message"
            2) "strawberry"

127.0.0.1:6379> xreadgroup group mugroup2 tuge count 1 streams mystream >
1) 1) "mystream"
   2) 1) 1) "1589995171242-0"
         2) 1) "message"
            2) "apple"
```

Redis Stream使用场景

可用作时通信等，大数据分析，异地数据备份等



客户端可以平滑扩展，提高处理能力



五 Redis 事务

严格意义上说 redis事务只是个批处理 有隔离性 但是没有原子性

事务演示

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s1 111
QUEUED
127.0.0.1:6379> hset set1 name zhangsan
QUEUED
127.0.0.1:6379> exec
1) OK
2) (integer) 1
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s2 222
QUEUED
127.0.0.1:6379> hset set2 age 20
QUEUED
127.0.0.1:6379> discard
OK
127.0.0.1:6379> exec
(error) ERR EXEC without MULTI

127.0.0.1:6379> watch s1
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s1 555
QUEUED
127.0.0.1:6379> exec      # 此时在没有exec之前，通过另一个命令窗口对监控的s1字段进行修改
(nil)
127.0.0.1:6379> get s1
111
```

暗号： 学以致用，知行合一

事务失败处理

- `Redis`语法错误

  整个事务的命令在队列里都清除

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> sets s1 111
(error) ERR unknown command 'sets'
127.0.0.1:6379> set s1
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get s4
(nil)
```

- `Redis`运行错误

  在队列里正确的命令可以执行 （弱事务性）

  弱事务性：

  1、在队列里正确的命令可以执行 （非原子操作）

  2、不支持回滚

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> lpush s4 111 222
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get s4
"444"
127.0.0.1:6379>
```

- `Redis`不支持事务回滚（为什么呢）

  1、大多数事务失败是因为**语法错误或者类型错误**，这两种错误，在开发阶段都是可以预见的

  2、`Redis`为了**性能方面**就忽略了事务回滚。 （回滚记录历史版本）

Redis乐观锁

乐观锁基于CAS（Compare And Swap）思想（比较并替换），是不具有互斥性，不会产生锁等待而消耗资源，但是需要反复的重试，但也是因为重试的机制，能比较快的响应。因此我们可以利用redis来实现乐观锁。具体思路如下：

1、利用redis的watch功能，监控这个redisKey的状态值
2、获取redisKey的值
3、创建redis事务
4、给这个key的值+1
5、然后去执行这个事务，如果key的值被修改过则回滚，key不加1

```
public void watch() {
    try {
        String watchKeys = "watchKeys";
        //初始值 value=1
        jedis.set(watchKeys, 1);
        //监听key为watchKeys的值
        jedis.watch(watchkeys);
```

```
        //开启事务
        Transaction tx = jedis.multi();

        //watchKeys自增加一
        tx.incr(watchKeys);

        //执行事务，如果其他线程对watchKeys中的value进行修改，则该事务将不会执行
        //通过redis事务以及watch命令实现乐观锁
        List<Object> exec = tx.exec();
        if (exec == null) {
            System.out.println("事务未执行");
        } else {
            System.out.println("事务成功执行，watchKeys的value成功修改");
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        jedis.close();
    }
}
```

Redis乐观锁实现秒杀

```
public class SecKill {
  public static void main(String[] arg) {
    String redisKey = "second";

    ExecutorService executorService = Executors.newFixedThreadPool(20);
    try {
      Jedis jedis = new Jedis("127.0.0.1", 6378);
      // 初始值
      jedis.set(redisKey, "0");
      jedis.close();
    } catch (Exception e) {
      e.printStackTrace();
    }

    for (int i = 0; i < 1000; i++) {

      executorService.execute(() -> {

        Jedis jedis1 = new Jedis("127.0.0.1", 6378);
        try {
          jedis1.watch(redisKey);
          String redisValue = jedis1.get(redisKey);
          int valInteger = Integer.valueOf(redisValue);
```

```
            String userInfo = UUID.randomUUID().toString();

            // 没有秒完
            if (valInteger < 20) {
              Transaction tx = jedis1.multi();
              tx.incr(redisKey);
              List list = tx.exec();
              // 秒成功    失败返回空list而不是空
              if (list != null && list.size() > 0) {

                System.out.println("用户: " + userInfo + ", 秒杀成功! 当前成功人数: "
+ (valInteger + 1));

              }
              // 版本变化，被别人抢了。
              else {
                System.out.println("用户: " + userInfo + ", 秒杀失败");
              }
            }
            // 秒完了
            else {
              System.out.println("已经有20人秒杀成功，秒杀结束");

            }
          } catch (Exception e) {
            e.printStackTrace();
          } finally {
            jedis1.close();
          }

        });
      }
      executorService.shutdown();

    }

}
```

256 个命令

六 Redis 和lua 整合

Redis整合lua是对Redis事务的补充。

什么是lua

lua是一种轻量小巧的**脚本语言**，用标准**C语言**编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Redis中使用lua的好处

1. **减少网络开销**，在Lua脚本中可以把多个命令放在同一个脚本中运行
2. **原子操作**，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件。 **隔离性**
3. **复用性**，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑

lua的安装和语法

lua 教程 https://www.runoob.com/lua/lua-tutorial.html

Redis整合lua脚本

从Redis2.6.0版本开始，通过**内置的lua编译/解释器**，可以使用EVAL命令对lua脚本进行求值。

EVAL命令

- 在redis客户端中，执行以下命令：

```
EVAL script numkeys key [key ...] arg [arg ...]
```

- **命令说明：**
  - **script参数：**是一段Lua脚本程序，它会被运行在Redis服务器上下文中，这段脚本不必(也不应该)定义为一个Lua函数。
  - **numkeys参数：**用于指定键名参数的个数。
  - **key [key ...]参数：** 从EVAL的第三个参数开始算起，使用了numkeys个键（key），表示在脚本中所用到的那些Redis键(key)，这些键名参数可以在Lua中通过全局变量**KEYS**数组，用1为基址的形式访问(KEYS[1] ， KEYS[2] ，以此类推)。
  - **arg [arg ...]参数：**可以在Lua中通过全局变量**ARGV**数组访问，访问的形式和KEYS变量类似(ARGV[1] 、 ARGV[2] ，诸如此类)。
- **例如：**

```
./redis-cli
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second

1) "key1"

2) "key2"

3) "first"

4) "second"
```

lua 脚本调用Redis 命令

redis.call();

返回值就是redis命令执行的返回值

如果出错，返回错误信息，不继续执行

redis.pcall();

返回值就是redis命令执行的返回值

如果出错了 记录错误信息，继续执行

注意事项

- 在脚本中，使用return语句将返回值返回给客户端，如果没有return，则返回nil
- 示例：

```
127.0.0.1:6379> eval "return redis.call('set',KEYS[1],'bar')" 1 foo

OK
```

redis-cli --eval

> 可以使用**redis-cli --eval**命令指定一个lua脚本文件去执行。

脚本文件(redis.lua)，内容如下：

```
local num = redis.call('GET', KEYS[1]);

if not num then
 return 0;
else
 local res = num * ARGV[1];
 redis.call('SET',KEYS[1], res);
 return res;
end
```

在redis客户机，执行脚本命令：

```
[root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbyml , 8
(integer) 0
[root@localhost bin]# ./redis-cli incr lua:incrbyml
(integer) 1
[root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbyml , 8
(integer) 8
[root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbyml , 8
(integer) 64
[root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbyml , 2
(integer) 128
[root@localhost bin]# ./redis-cli
```

命令格式说明：

```
--eval：告诉redis客户端去执行后面的lua脚本
redis.lua：具体的lua脚本文件名称
lua:incrbymul ： lua脚本中需要的key
 8：lua脚本中需要的value
```

- 注意事项：

```
上面命令中keys和values中间需要使用逗号隔开，并且逗号两边都要有空格
```

执行.lua脚本 不需要写key的个数

Redis+lua 秒杀

秒杀场景经常使用这个东西，主要利用他的原子性

1.首先定义redis数据结构

```
goodId:
 {
 "total":100,
 "released":0;
 }
```

- 其中goodId为商品id号，可根据此来查询相关的数据结构信息，total为总数，released为发放出去的数量，可使用数为total-released

  2.编写lua脚本

```
local n = tonumber(ARGV[1])
if not n or n == 0 then
return 0
end
local vals = redis.call("HMGET", KEYS[1], "total", "released");
local total = tonumber(vals[1])
```

```
local blocked = tonumber(vals[2])
if not total or not blocked then
return 0
end
if blocked + n <= total then
redis.call("HINCRBY", KEYS[1], "released", n)
return n;
end
return 0
```

> - 执行脚本命令`EVAL script_string 1 goodId apply_count`
> - 若库存足够则返回申请的数量，否则返回0，不返回可满足的剩余数
>
> 3.spring boot 调用
>
> - pom dependency

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-redis</artifactId>
<version>2.0.1.RELEASE</version>
```

```
        long count = redisHelper.getStrCache().execute(new RedisCallback<Long>
() {
            @Nullable
            @Override
            public Long doInRedis(RedisConnection redisConnection) throws
DataAccessException {
                long ret =
redisConnection.eval(script.getScriptAsString().getBytes(), ReturnType.INTEGER,
1, key.getBytes(), String.valueOf(count).getBytes());
                return ret;
            }
        });
```

4.redis->database

> 针对redis到databases的更新，思考了很久，没有找到较好的解决办法，先采用定时任务异步更新。至于数据是否丢失的问题，如果redis挂了，重启后redis会恢复数据，等下次定时任务就可以将数据库中的数据保持一致，缺点是redis挂了秒杀活动会失败。

至于redis到database更新方案：

- redis存一份相关hash键名单表，通过读取名单表来读取更新
- 通过流式读取databases中的表来读取更新。

七 Redis分布式锁

业务场景

1、库存超卖　比如　5个笔记本　A 看 准备买3个　B 买2个　C 4个　一下单 3+2+4 =9

2、防止用户重复下单

3、MQ消息去重

4、订单操作变更

分析：

业务场景共性：

共享资源竞争

　用户id、订单id、商品id。。。

解决方案

　共享资源互斥

　共享资源串行化

问题转化

　锁的问题　（将需求抽象后得到问题的本质）

锁的处理

- 单应用中使用锁：（单进程多线程）

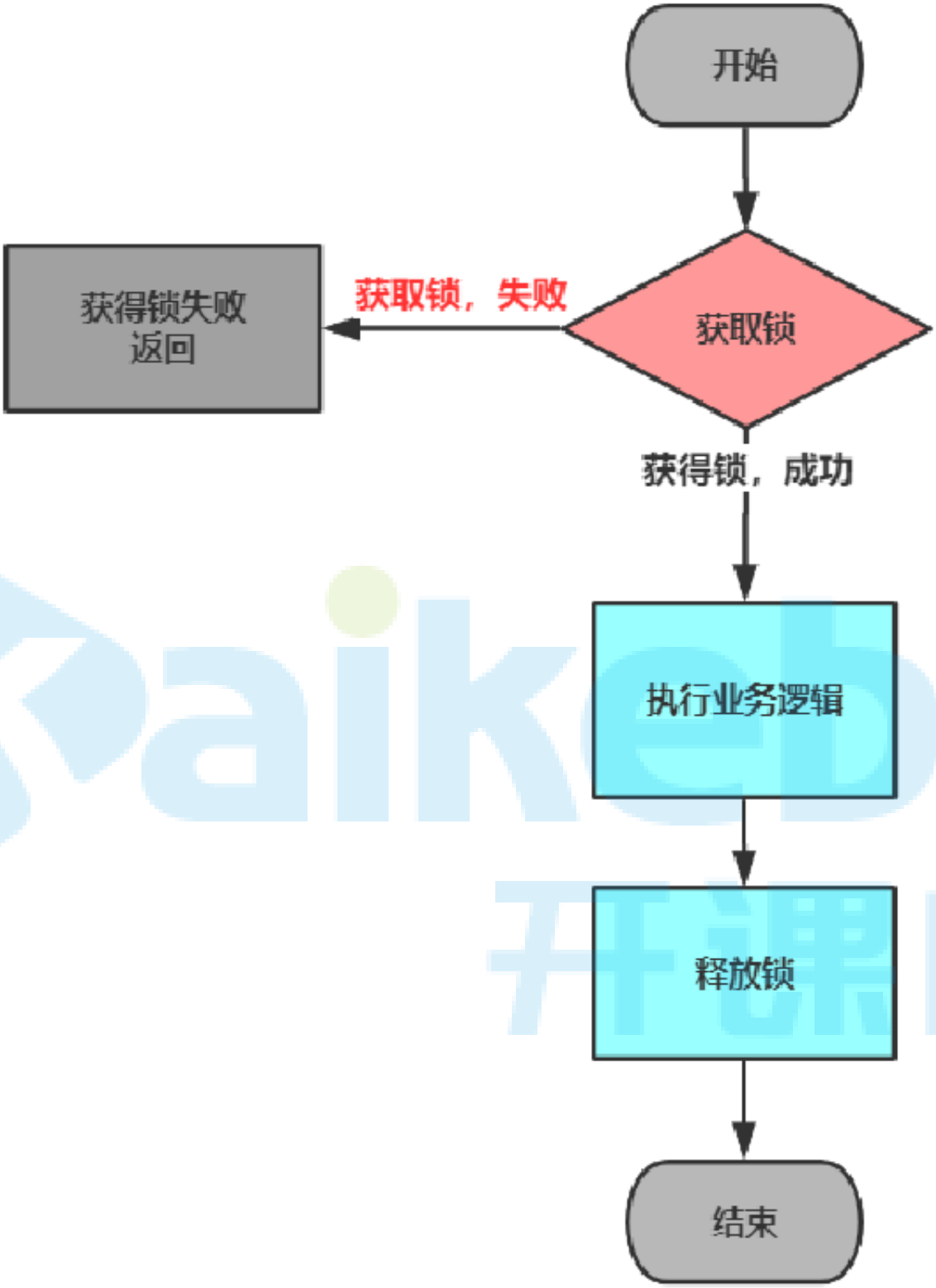
  ```
  synchronize、ReentrantLock
  ```

- 分布式应用中使用锁：（多进程多线程）

  ```
  分布式锁是控制分布式系统之间同步访问共享资源的一种方式。
  ```

分布式锁

流程图



分布式锁的状态

1. 客户端通过竞争获取锁才能对共享资源进行操作(①获取锁)；
2. 当持有锁的客户端对共享资源进行操作时（②占有锁）
3. 其他客户端都不可以对这个资源进行操作（③阻塞）
4. 直到持有锁的客户端完成操作(④释放锁)；

分布式锁特点

互斥性

在任意时刻，只有一个客户端可以持有锁（排他性）

高可用，具有容错性

只要锁服务集群中的大部分节点正常运行，客户端就可以进行加锁解锁操作

避免死锁

具备锁失效机制，锁在一段时间之后一定会释放。（正常释放或超时释放）

加锁和解锁为同一个客户端

一个客户端不能释放其他客户端加的锁了

分布式锁的实现方式

基于数据库实现分布式锁

基于zookeeper时节点的分布式锁

基于Redis的分布式锁

基于Etcd的分布式锁

Redis方式实现分布式锁

只留下正确的方式

获取锁

Redis2.6.12版本之前，使用Lua脚本保证原子性，获取锁代码

```
    // 加锁脚本
    private static final String SCRIPT_TRYLOCK = "if redis.call('setnx', KEYS[1],
ARGV[1]) == 1 then redis.call('pexpire', KEYS[1], ARGV[2]) return 1 else return 0
end";
     /**
     * 使用Lua脚本，尝试获取redis分布式锁
     * @param jedis
     * @param lockKey 锁
     * @param lockValue 锁值
     * @param expireTime 超期时间，单位秒
     * @return 是否获取成功
     */
    public static boolean tryLockLua(Jedis jedis, String lockKey, String lockValue,
int expireTime) {
        int result =
(int)jedis.eval(SCRIPT_TRYLOCK,1,lockKey,lockValue,String.valueOf(expireTime));//设置锁
        if (result == 1) {//获取锁成功
            return true;
        }
        return false;
    }
```

从Redis2.6.12版本开始，使用Set一个命令实现加锁，获取锁代

```
/**
     * 获取redis分布式锁
     * @param jedis
     * @param lockKey 锁
     * @param lockValue 锁值
     * @param expireTime 超期时间，单位毫秒
     * @return 是否获取成功
     */
    public static boolean tryLock(Jedis jedis, String lockKey, String lockValue, int
expireTime) {
        String result = jedis.set(lockKey, lockValue, "NX", "PX", expireTime);

        if ("OK".equals(result)) {
            return true;
        }
        return false;

    }
```

释放锁

redis+lua 脚本

```
public static boolean releaseLock(String lockKey, String requestId) {
    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end";
    Object result = jedis.eval(script, Collections.singletonList(lockKey),
Collections.singletonList(requestId));
    if (result.equals(1L)) {
      return true;
    }
    return false;
}
```

Redis分布式锁--优缺点

优点

Redis是基于内存存储，并发性能好。

缺点

1. 需要考虑原子性、超时、误删等情形。
2. 获锁失败时，客户端只能自旋等待，在高并发情况下，性能消耗比较大。

rediscluster--redis主从复制的坑

`redis`高可用最常见的方案就是`主从复制`（master-slave），这种模式也给`redis分布式锁`挖了一坑。

`redis cluster`集群环境下，假如现在`A客户端`想要加锁，它会根据路由规则选择一台`master`节点写入`key mylock`，在加锁成功后，`master`节点会把`key`异步复制给对应的`slave`节点。

如果此时`redis master`节点宕机，为保证集群可用性，会进行`主备切换`，`slave`变为了`redis master`。`B客户端`在新的`master`节点上加锁成功，而`A客户端`也以为自己还是成功加了锁的。

此时就会导致同一时间内多个客户端对一个分布式锁完成了加锁，导致各种脏数据的产生。

至于解决办法嘛，目前看**还没有什么根治**的方法，**只能尽量保证机器的稳定性**，减少发生此事件的概率。

本质分析

本质分析

CAP模型分析

P：容错

A:高可用

C：一致性

在分布式环境下不可能满足三者共存，只能满足其中的两者共存，在分布式下P不能舍弃(舍弃P就是单机了)。

所以只能是CP（强一致性模型）和AP(高可用模型)。

分布式锁是CP模型，Redis集群是AP模型。 (base)

为什么还可以用Redis实现分布式锁?

与业务有关

当业务不需要数据强一致性时，比如：社交场景，就可以使用Redis实现分布式锁

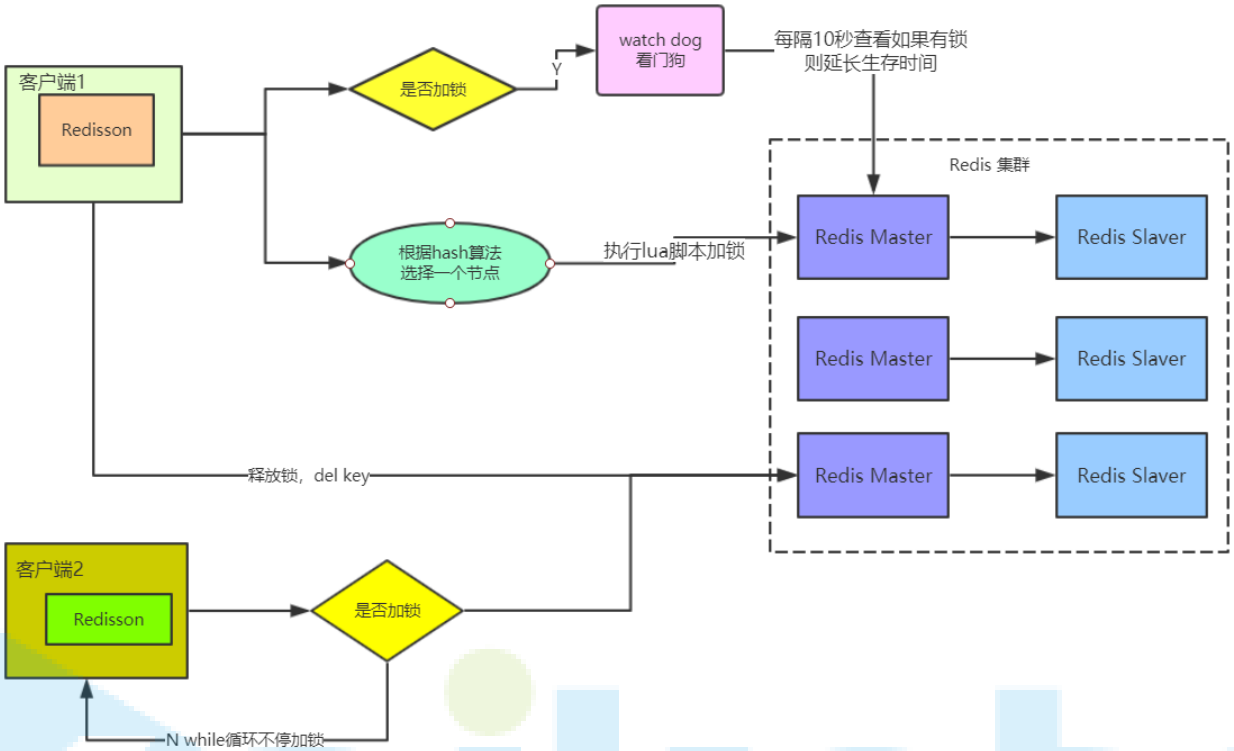
当业务必须要数据的强一致性，即不允许重复获得锁，比如金融场景（重复下单，重复转账）就不要使用

可以使用CP模型实现，比如：zookeeper和**etcd**。

生产环境中的分布式锁

目前落地生产环境用分布式锁，一般采用开源框架，比如Redisson。下面来讲一下Redisson对Redis分布式锁的实现。

Redisson分布式锁的实现原理



加锁机制

如果该客户端面对的是一个redis cluster集群，他首先会根据hash节点选择一台机器。

发送lua脚本到redis服务器上，脚本如下:

```
"if (redis.call('exists',KEYS[1])==0) then "+
  "redis.call('hset',KEYS[1],ARGV[2],1) ; "+
  "redis.call('pexpire',KEYS[1],ARGV[1]) ; "+
  "return nil; end ;" +
"if (redis.call('hexists',KEYS[1],ARGV[2]) ==1 ) then "+
    "redis.call('hincrby',KEYS[1],ARGV[2],1) ; "+
    "redis.call('pexpire',KEYS[1],ARGV[1]) ; "+
  "return nil; end ;" +
"return redis.call('pttl',KEYS[1]) ;"
```

lua的作用：保证这段复杂业务逻辑执行的原子性。

lua的解释：

KEYS[1])： 加锁的key

ARGV[1]： key的生存时间，默认为30秒

ARGV[2]： 加锁的客户端ID (*UUID.randomUUID()*） + *":"* + *threadId*)

第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。如何加锁呢？很简单，用下面的命令：

hset myLock

8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：

myLock :{"8743c9c0-0795-4907-87fd-6c719a6b4586:1":1 }

上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁key完成了加锁。

接着会执行“pexpire myLock 30000”命令，设置myLock这个锁key的生存时间是30秒。

锁互斥机制

那么在这个时候，如果客户端2来尝试加锁，执行了同样的一段lua脚本，会咋样呢？

很简单，第一个if判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl myLock返回的一个数字，这个数字代表了myLock这个锁key的**剩余生存时间。**比如还剩15000毫秒的生存时间。

此时客户端2会进入一个while循环，不停的尝试加锁。

自动延时机制

只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

可重入锁机制

第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。

第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：

incrby myLock

8743c9c0-0795-4907-87fd-6c71a6b4586:1 1

通过这个命令，对客户端1的加锁次数，累加1。数据结构会变成：

myLock :{"8743c9c0-0795-4907-87fd-6c719a6b4586:1":2 }

释放锁机制

执行lua脚本如下：

```
#如果key已经不存在，说明已经被解锁，直接发布（publish）redis消息
"if (redis.call('exists', KEYS[1]) == 0) then " +
                        "redis.call('publish', KEYS[2], ARGV[1]); " +
                        "return 1; " +
                    "end;" +
# key和field不匹配，说明当前客户端线程没有持有锁，不能主动解锁。
                    "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
                        "return nil;" +
                    "end; " +
# 将value减1
                    "local counter = redis.call('hincrby', KEYS[1], ARGV[3],
-1); " +
# 如果counter>0说明锁在重入，不能删除key
                    "if (counter > 0) then " +
                        "redis.call('pexpire', KEYS[1], ARGV[2]); " +
                        "return 0; " +
# 删除key并且publish 解锁消息
                    "else " +
                        "redis.call('del', KEYS[1]); " +
                        "redis.call('publish', KEYS[2], ARGV[1]); " +
                        "return 1; "+
                    "end; " +
                    "return nil;",
```

– KEYS[1]： 需要加锁的key，这里需要是字符串类型。

– KEYS[2]： redis消息的ChannelName,一个分布式锁对应唯一的一个channelName: “redisson_lock**channel**{” + getName() + “}”

– ARGV[1]： reids消息体，这里只需要一个字节的标记就可以，主要标记redis的key已经解锁，再结合redis的Subscribe，能唤醒其他订阅解锁消息的客户端线程申请锁。

– ARGV[2]： 锁的超时时间，防止死锁

– ARGV[3]： 锁的唯一标识，也就是刚才介绍的 id（UUID.randomUUID()） + “:” + threadId

如果执行lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del myLock”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。

Redisson分布式锁的使用

加入jar包的依赖

```
<dependency>
        <groupId>org.redisson</groupId>
        <artifactId>redisson</artifactId>
         <version>3.7.2</version>
 </dependency>
```

配置Redisson

```
public class RedissonManager {
    private static Config config = new Config();
    //声明redisso对象
    private static Redisson redisson = null;
   //实例化redisson
 static{
     config.useClusterServers()

// 集群状态扫描间隔时间，单位是毫秒

    .setScanInterval(2000)

    //cluster方式至少6个节点(3主3从，3主做sharding，3从用来保证主宕机后可以高可用)

    .addNodeAddress("redis://127.0.0.1:6379" )

    .addNodeAddress("redis://127.0.0.1:6380")

    .addNodeAddress("redis://127.0.0.1:6381")

    .addNodeAddress("redis://127.0.0.1:6382")

    .addNodeAddress("redis://127.0.0.1:6383")

    .addNodeAddress("redis://127.0.0.1:6384");

          //得到redisson对象
        redisson = (Redisson) Redisson.create(config);

}

 //获取redisson对象的方法
    public static Redisson getRedisson(){
        return redisson;
    }
}
```

锁的获取和释放

```
public class DistributedRedisLock {
   //从配置类中获取redisson对象
    private static Redisson redisson = RedissonManager.getRedisson();
    private static final String LOCK_TITLE = "redisLock_";
   //加锁
    public static boolean acquire(String lockName){
       //声明key对象
        String key = LOCK_TITLE + lockName;
       //获取锁对象
        RLock mylock = redisson.getLock(key);
       //加锁，并且设置锁过期时间3秒，防止死锁的产生    uuid+threadId
        mylock.lock(3,TimeUtil.SECOND);
       //加锁成功
        return  true;
    }
  //锁的释放
    public static void release(String lockName){
       //必须是和加锁时的同一个key
        String key = LOCK_TITLE + lockName;
       //获取锁对象
        RLock mylock = redisson.getLock(key);
      //释放锁（解锁）
        mylock.unlock();

    }
}
```

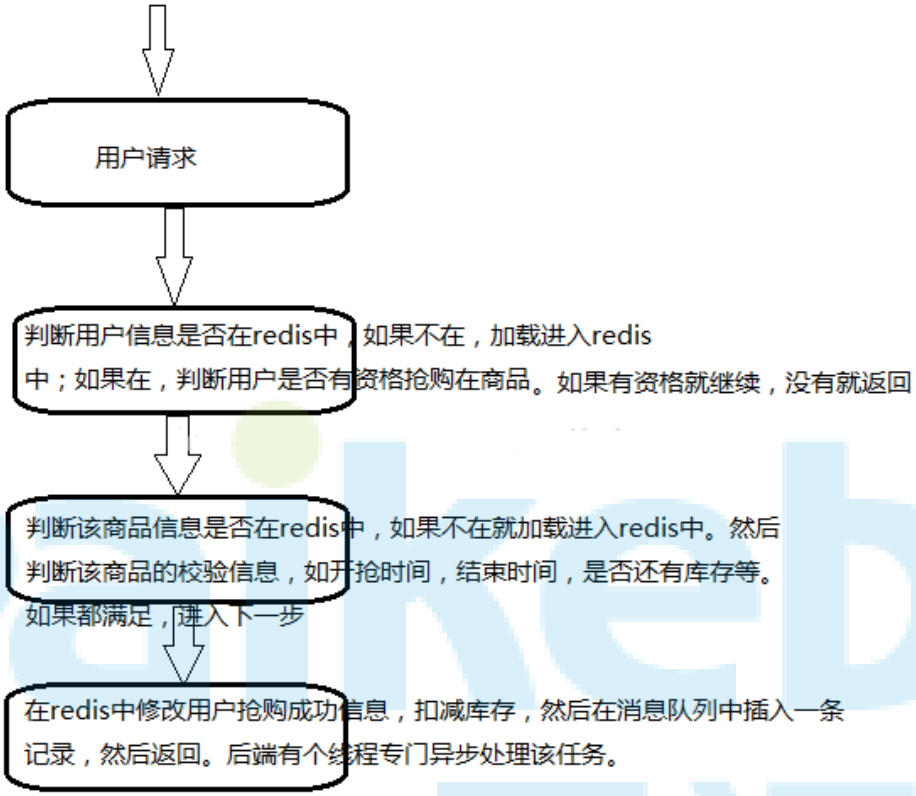
业务逻辑中使用分布式锁

```
public String discount() throws IOException{
         String key = "test123";
        //加锁
        DistributedRedisLock.acquire(key);
        //执行具体业务逻辑
        dosoming
        //释放锁
        DistributedRedisLock.release(key);
       //返回结果
        return soming;
    }
```

补充：Redis 使用

https://github.com/feibabm/seckill

> srping boot + Mybatis + redis 商品秒杀
>
> 主要实现了一个抢购接口: http://localhost:8080/seckill/product/1?userId=1
>
> seckill.sql文件为建表sql pro_insert.sql文件为success_killed表中数据添加10000条用户预约记录 seckill_insert.sql文件为seckill生成一条产品信息
>
> 具体验证逻辑是执行test文件夹下的两个test类： RemoteInvote.java RemoteInvote2.java 这两个test类没有什么差别，主要是为了增加并发量



#