Redis进阶

## 1. 事务

redis事务主要是为了方便用户一次执行多个命令，可分为三个阶段：

* 开始事务
* 命令入队
* 执行事务

事务特性：

* 隔离操作，事务中所有命令都会被序列化，按照顺序执行，执行中不会被打断
* 不保证原子性，不支持事务回滚  
  **注：如果想保证原子性，可以使用lua脚本**

事务命令

| 命令 | 说明 |
| --- | --- |
| multi | 开启一个事务 |
| exec | 顺序执行事务中的所有命令，exec后会取消watch的监控 |
| watch key [key...] | 在开启事务之前用来监视一个或多个key。如果事务执行前这些key被改动过，那么整个事务将不被执行 |
| discard | 在multi后exec前，取消事务队列组队 |
| unwatch | 取消watch命令对key的监控 |

例子

开启事务  
127.0.0.1:6379> MULTI  
OK  
127.0.0.1:6379> INCR 1  
QUEUED #命令入队成功  
127.0.0.1:6379> SET num 10  
QUEUED  
#批量执行命令  
127.0.0.1:6379> EXEC  
1) (integer) 1  
2) OK  
  
开启事务  
127.0.0.1:6379> MULTI  
OK  
127.0.0.1:6379> INCR 1  
QUEUED #命令入队成功  
127.0.0.1:6379> SET num 10  
QUEUED  
#取消事务建立  
127.0.0.1:6379> DISCARD  
OK  
  
开启事务之前设置key/value，并监听  
127.0.0.1:6379> set www.biancheng.net hello  
OK  
127.0.0.1:6379> WATCH www.biancheng.net  
OK  
127.0.0.1:6379> get www.biancheng.net  
"hello"  
#开启事务  
127.0.0.1:6379> MULTI  
OK  
#更改key的value值  
127.0.0.1:6379> set www.biancheng.net HELLO  
QUEUED  
127.0.0.1:6379> GET www.biancheng.net  
QUEUED  
#命令执行失败  
127.0.0.1:6379> EXEC  
(error) EXECABORT Transaction discarded because of previous errors.  
#取消监听key  
127.0.0.1:6379> UNWATCH   
OK  
复制

multi可以理解为让redis命令进入事务执行队列中，如果组队过程中出现错误则整个事务队列在执行时会被取消（比如组队命令中出现语法问题）  
而exec可以理解为让在事务队列中的redis命令按照顺序进行，如果执行过程中若干命令出现错误，则只有错误命令无法被执行（命令中出现运行时错误）

对于watch与事务回滚，可以使用CAS来理解原理过程：

* 首先，在线程开始时读取这些多线程共享的数据，并将其保存到当前进程的副本中，我们称为旧值old value，watch命令就是这样的一个功能
* 然后，开启线程业务逻辑，由multi命令提供这一功能。在事务执行更新前，比较当前线程副本保存的旧值和当前线程共享的值是否一致，如果不一致，那么该数据已经被其他线程操作过，此次更新失败
* 为了保持一致，线程就不去更新任何值，而将事务回滚；否则就认为它没有被其他线程操作过，执行对应的业务逻辑，exec命令就是执行“类似”这样的一个功能

但是CAS会出现ABA问题，可以采用添加version字段或者timestamp时间戳字段，排除ABA问题

**因此redis中事务操作的watch涉及到乐观锁的原理**

redis事务三特性：

* 单独隔离操作
* 没有隔离级别的概念
* 不保证原子性

## 2. RDB持久化

RDB即快照模式，它是Redis默认的数据持久化方式，它会将数据库的快照保存在dump.rdb这个二进制文件中

redis本身为单线程，如果需要在满足服务线上请求、负责多个客户端并发读写以及内存数据逻辑读写的同时，完成快照备份的文件IO操作，redis会使用操作系统的多进程COW(Copy On Write)机制来实现快照持久化操作

RDB原理：RDB实际上是redis内部的一个定时器事件，它每隔一段固定时间就去检查当前数据发生改变的次数和改变的时间频率，看它们是否满足配置文件中规定的持久化触发条件。当满足条件时，Redis就会通过操作系统调用fork()来创建一个子进程，该子进程与父进程享有相同的地址空间  
Redis通过子进程遍历整个内存空间来获取存储的数据，从而完成数据持久化操作。注意，此时的主进程则仍然可以对外提供服务，父子进程之间通过操作系统的COW机制实现了数据段分离，从而保证了父子进程之间互不影响

RDB触发方式

1. 手动触发策略

手动触发是通过save命令或者bgsave命令将内存数据保存到磁盘文件中  
然而bgsave为后台执行数据保存操作，为非阻塞操作，底层会fork()一个子进程完成持久化；而save命令会阻塞redis服务器进程直到dump.rdb创建完毕为止  
  
127.0.0.1:6379> save  
OK  
127.0.0.1:6379> bgsave  
Background saving started  
查看bgsave是否成功  
127.0.0.1:6379> lastsave  
(integer) 1611298430  
复制

1. 自动触发策略：在指定时间内，数据发生变化的次数达到规定值，会自动执行bgsave命令  
   自动触发的条件包含在redis的配置文件中

save 900 1 # 在900秒内至少更新1条数据，redis自动触发bgsave  
save 300 10 # 在300秒内至少更新10条数据，redis自动触发bgsave  
save 60 10000 # 在60秒内至少更新10000条数据，redis自动触发bgsave  
复制

## 3. AOF持久化

AOF被称为追加模式，或日志模式，是Redis提供的另一种持久化策略，它能够存储Redis服务器已经执行过的的命令，并且只记录对内存有过修改的命令，这种数据记录方法，被叫做“增量复制”，其默认存储文件为appendonly.aof

AOF默认未开启，需通过配置文件开启

1. windows系统

修改配置文件，把no改为 yes  
appendonly yes  
确定存储文件名是否正确  
appendfilename "appendonly.aof"  
重启服务器  
redis-server --service-stop  
redis-server --service-start  
复制

1. linux系统

修改配置文件：  
vim /etc/redis/redis.conf  
appendonly yes # 把 no 改为 yes  
确定存储文件名是否正确  
appendfilename "appendonly.aof"  
重启服务：  
sudo /etc/init.d/redis-server restart  
复制

每当有一个修改数据库的命令被执行时，服务器就将命令写入到appendonly.aof文件中，该文件存储了服务器执行过的所有修改命令，因此，只要服务器重新执行一次.aof文件，就可以实现还原数据的目的，这个过程被形象地称之为“命令重演”

1. **写入机制**  
   Redis在收到客户端修改命令后，先进行相应的校验。如果没问题，就立即将该命令存追加到.aof文件中，也就是先存到磁盘中，然后服务器再执行命令  
   Redis为了提升写入效率，它不会将内容直接写入到磁盘中，而是将其放到一个内存缓存区buffer中，等到缓存区被填满时才真正将缓存区中的内容写入到磁盘里
2. **重写机制**  
   当aof文件过大会造成重演非常耗时，可以手动执行bgrewriteaof命令重写瘦身aof文件
   * 新的aof文件记录的数据库数据和原aof文件记录的数据库数据完全一致
   * 新的aof文件会使用尽可能少的命令来记录数据库数据
   * AOF重写期间，服务器不会被阻塞，它可以正常处理客户端发送的命令
3. **自动触发AOF重写**

配置文件默认配置项  
auto-aof-rewrite-percentage 100 表示当aof文件增量大于100%才进行重写，第一次重写时文件大小为64M，第二次触发重写体积为128M  
auto-aof-rewrite-min-size 64mb 表示触发AOF重写的最小文件体积,大于或等于64MB自动触发  
复制

AOF写入策略

* always：服务器每写入一个命令，就调用一次 fsync 函数，将缓冲区里面的命令写入到硬盘
* everysec（默认策略）：服务器每一秒调用一次 fsync 函数，将缓冲区里面的命令写入到硬盘。这种模式下，服务器出现故障，最多只丢失一秒钟内的执行的命令数据
* no：服务器不主动调用 fsync 函数，由操作系统决定何时将缓冲区里面的命令写入到硬盘

| RDB持久化 | AOF持久化 |
| --- | --- |
| 全量备份，一次保存整个数据库 | 增量备份，一次只保存一个修改数据库的命令 |
| 每次执行持久化操作的间隔时间较长 | 保存的间隔默认为一秒钟（Everysec） |
| 数据保存为二进制格式，其还原速度快 | 使用文本格式还原数据，所以数据还原速度一般 |
| 执行 SAVE 命令时会阻塞服务器，但手动或者自动触发的 BGSAVE 不会阻塞服务器 | AOF持久化无论何时都不会阻塞服务器 |

## 4. 主从复制

主从复制能实现：

* 读写分离
* 容灾快速恢复

*redis中只能使用一主一从或者一主多从，因为无法确定多主时的命令优先级*  
可以使用info replication查看主从信息

redis在不配置主从角色时均为master，搭建主从模式时，可以执行以下命令指定所属的主redis

redis启动命令  
redis-server --port <slave-port> --slaveof <master-ip> <master-port>  
redis-server --port 6300 --slaveof 127.0.0.1 6379  
  
重点是规划好主从redis各自的ip与port之后，使用slaveof命令指定自己的所属主机  
复制

配置主从有多种方式：

* 启动redis服务时，在启动命令指定自己的所属主机ip与port
* 启动redis服务后，通过redis-cli连接上服务，并执行slaveof指定自己的所属主机
* 创建redis多个配置文件，在redis启动时指定使用的配置文件，在配置文件中使用slaveof ip port指定所属主机

主从复制原理：

1. 从服务器连接主服务器，发送SYNC命令
2. 主服务器接收到SYNC命名后，**开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令**
3. 主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令
4. 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照（全量复制）
5. 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令（增量复制）
6. 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令

redis主从模式的不足：

* 主节点宕机，需要人为干预，把从节点升为主节点
* 主节点宕机的时刻正好在主从同步的过程中，会导致主从数据不一致
* 只有一个主节点，写入能力有限制
* 主从全量同步时，可能会因为数据量大导致卡顿
* 从服务器宕机后，再次启动需手动设置为从服务器，数据会自动同步；主服务器宕机重启后，不需要手动设置角色

薪火相传：主从模式中，从服务器往下还能存在自己的从服务器  
反客为主：主机宕机，从机可以升级为主机，手动执行slaveof no one命令升级

### Sentinel哨兵模式：

反客为主的自动版，后台监控主机是否正常，宕机故障则进行投票自动将从机升级为主机  
  
*哨兵模式存在一个独立的哨兵进程来监控redis集群*

* 哨兵节点会以每秒一次的频率对每个 Redis 节点发送PING命令，并通过 Redis 节点的回复来判断其运行状态。
* 当哨兵监测到主服务器发生故障时，会自动在从节点中选择一台将机器，并其提升为主服务器，然后使用PubSub发布订阅模式，通知其他的从节点，修改配置文件，跟随新的主服务器。

\*多个节点通过Sentinel进程互相监控，Sentinel有点类似zookeeper的客户端去获取服务列表注册\*

* **主观下线**：适用于主服务器与从服务器，规定时间内Sentinel发出的ping命令没有接收到相应的pong命令，则认为对方下线
* **客观下线**：只适用于主服务器，从服务器的Sentinel对主服务器进行故障判断，如果半数以上的从服务器都认为主服务器下线，则为客观下线
* **投票选举**：所有Sentinel节点投票，选举第一个发现主服务器下线的从服务器为领头节点进行Failover故障转移操作。该从节点以一定规则从所有从节点中选举最优作为主服务器，之后通过pubsub通知其他节点更改配置文件

Sentinel哨兵配置：

1. 安装sentinel：sudo apt install redis-sentinel
2. 搭建主从模式

启动6379的redis服务器作为master主机:  
sudo /etc/init.d/redis-server start  
  
启动6380的redis服务器，设置为6379的slave:  
redis-server --port 6380  
$ redis-cli -p 6380  
127.0.0.1:6380> slaveof 127.0.0.1 6379  
OK  
  
启动6381的redis服务器，设置为6379的salve  
redis-server --port 6381  
$ redis-cli -p 6381  
127.0.0.1:6381> slaveof 127.0.0.1 6379  
复制

1. 配置sentinel哨兵配置文件

port 26379 #sentinel监听端口，默认是26379，可以更改  
Sentinel monitor biancheng 127.0.0.1 6379 1  
sentinel monitor <master-name> <ip> <redis-port> <quorum> quorum表示当有quorum个sentinel认为主服务器宕机，则认为是真正宕机  
复制

1. 启动sentinel

方式1 redis-sentinel sentinel.conf  
方式2 redis-server sentinel.conf --sentinel  
复制

public static Jedis getJedisFromSentinel() {  
 if(jedisSentinelPool == null) {  
 Set<String> sentinelSet = new HashSet<>();  
 //添加sentinel服务配置  
 sentinelSet.add("192.168.11.103:26379");  
   
 JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();  
 jedisPoolConfig.setMaxTotal(10);//最大可用连接数  
 jedisPoolConfig.setMaxIdle(5);//最大闲置连接数  
 jedisPoolConfig.setMinIdle(5);//最小闲置连接数  
 jedisPoolConfig.setBlockWhenExhausted(true);//连接耗尽是否等待  
 jedisPoolConfig.setMaxWaitMillis(2000);//等待时间  
 jedisPoolConfig.setTestOnBorrow(true);//取连接时候进行一下ping pong测试  
   
 jedisSentinelPool = new JedisSentinelPool("mymaster", sentinelSet, jedisPoolConfig);  
 return jedisSentinelPool.getResource();  
 }  
}  
复制

## 5. 分区与集群

### 分区

redis分区：实现了水平扩容，启动N个节点，数据分步在N个节点当中，每个节点存储总数据的N/1，可以理解为将整个redis实例中的数据分区

redis分区分为两种：

* 范围分区：将特定范围的key映射到指定的Redis实例上。使用方法，key命名格式object\_name:<id>，id为redis实例ID
* 哈希分区：键值对key与redis实例ID的格式关系，id=hash(key)%N，N表示有多少个实例

分区的不足

* 涉及操作多个 key 时，通常不被支持，无法实现在一个实例中操作分散开的 key
* 不支持包含多个 key 的 Redis 事务
* 当使用分区的时候，数据的处理变的非常复杂，比如需要处理多个 .rdb 或者 .aof 存储文件，并且还需要从多个 Redis 实例中备份数据
* Redis 集群支持在运行时增加或减少实例，分区技术不支持这种功能

### 集群

redis容量不够需要扩容或者并发写操作需要分摊，可以使用**无中心化集群**，实现动态扩容（主要针对哨兵模式下的主从模式中，主节点写压力过大）

在redis-cluster中，每个节点都是中心节点，拥有集群的状态元数据，本身又使用**分区**存储部分数据；所有redis节点使用（ping机制）互联，如果集群中某个节点的失败，是需要整个集群中超过半数的节点监测都失效才算真正的失效

redis cluster把所有的redis node映射到 0-16383 个槽位(slot)上，读写需要到指定的redis node上进行操作。cluster预先分配 16384 个(slot)槽位，当需要在redis集群中写入一个key -value的时候，会使用 CRC16(key) mod 16384之后的值，决定将key写入值哪一个槽位从而决定写入哪一个Redis节点上

redis-cluster支持节点主从替换，类似于哨兵模式sentinel功能，但是不需要部署sentinel，cluster已经支持主从切换

## 6. 应用问题

### 1. 秒杀

秒杀过程：

1. userid与prodid的非空判断
2. 连接redis
3. 拼接库存key与秒杀成功用户key
4. 获取库存，判断库存是否为null或者0
5. 判断用户是否重复秒杀，秒杀成功用户集合中是否存在相应用户记录
6. 秒杀过程：库存减一，秒杀成功用户添加入set集合

秒杀可能会出现的问题：

1. 连接超时
2. 超卖，商品被清空还是能被秒杀，属于并发问题（线程安全问题）

public static boolean doSecKill(String uid, String prodid) throws IOException {  
   
 if(uid == null || prodid == null) {  
 return false;  
 }  
  
 JedisPool jedisPoolInstance = JedisPoolUtil.getJedisPoolInstance();  
 Jedis jedis = jedisPoolInstance.getResource();  
  
 String kcKey = "sk:" + prodid + ":qt";  
 String userKey = "sk:" + proid + ":user";  
  
 //实则就是使用了watch底层的乐观锁  
 jedis.watch(kcKey);  
  
 String kc = jedis.get(kcKey);  
 if(kc == null) {  
 System.out.println("秒杀还没有开始，请等待");  
 jedis.close();  
 return false;  
 }  
  
 if(jedis.sismember(userKey, uid)) {  
 System.out.println("已经秒杀成功了，不能重复秒杀");  
 jedis.close();  
 return false;  
 }  
  
 if(Integer.parseInt(kc) <= 0) {  
 System.out.println("秒杀已经结束了");  
 jedis.close();  
 return false;  
 }  
  
 //这一部分代码可能会出现kcKey键值对版本号变化但值还大于0的情况，这种情况下事务也有可能无法执行，导致库存剩余  
 Transaction multi = jedis.multi();  
 multi.decr(kcKey);  
 multi.sadd(userKey, uid);  
 List<Object> results = multi.exec();  
  
 if(results == null || results.size() == 0) {  
 System.out.println("秒杀失败了...");  
 jedis.close();  
 return false;  
 }  
   
 System.out.println("秒杀成功了...");  
 jedis.close();  
 return true;  
}  
复制

1. 乐观锁造成的库存遗留，解决超卖问题时事务因为键值对版本号变化而无法执行  
   使用lua脚本语言，lua脚本存在一定的原子性，不会被其他命令插队，可以解决争抢问题，**实际上是redis利用单线程特性，用任务队列的串行方式解决多任务并发问题**

public static boolean doSecKill(String uid, String prodid) throws IOException {  
  
 JedisPool jedisPoolInstance = JedisPoolUtil.getJedisPoolInstance();  
 Jedis jedis = jedisPoolInstance.getResource();  
  
 String shal = jedis.scriptLoad(secKillScript);  
 Object result = jedis.evalsha(shal, 2, userId, prodid);  
  
 String result = String.valueOf(result);  
 if("0".equals(result)) {  
 System.out.println("以抢空");  
 } else if("1".equals(result)) {  
 System.out.println("抢购成功");  
 } else if("2".equals(result)) {  
 System.out.println("已抢购过");  
 } else {  
 System.out.println("抢购异常");  
 }  
 jedis.close();  
 return true;  
}  
复制

local userid = KEYS[1];  
local prodid = KEYS[2];  
local qtkey = "sk:"..prodid..":qt";  
local userskey = "sk:"..prodid..":usr";  
local userExists = redis.call("sismember", userskey, userid);  
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, userid);  
end  
return 1;  
复制

### 2. 缓存穿透

条件：

1. 应用服务器增大
2. 请求数据先查询缓存层，再查询持久层
3. 缓存层命中率因为请求量大而降低，大量请求到达持久层中，导致数据库崩溃

解决方案：

1. 对空值作缓存：如果一个查询返回的数据为空（不管数据是否存在），仍然把这个空值作缓存，设置空值键值对过期时间很短，**这样一些过量的恶意请求就会倾斜向缓存层**
2. 设置可访问的白名单：使用bitmaps类型定义一个可访问名单，名单id作为bitmaps偏移量，每次访问和bitmap中的id作比较，如果id不存在，进行拦截不允许访问，**但是该bitmaps会被大量读取导致效率低下**
3. 采用布隆过滤器，底层也是一个bitmaps
4. 进行实时监控：监控redis中的命中率，设置黑名单白名单

### 3. 缓存击穿

条件：

1. 数据库访问压力瞬时增加
2. redis不存在大量过期key，命中率正常
3. redis正常运行
4. redis中存在某些热点key经常被大量访问，但是现在过期了，导致请求转移到数据库中

解决方案：

1. 预先设置热门数据：在高峰访问前，把热门数据提前存入缓存层，并增加过期时间
2. 实时调整：实时监控热点key，同时调整key的过期时间
3. 使用锁：在并发的多个请求中，只有第一个请求线程能拿到锁并执行数据库查询操作，其他的线程拿不到锁就阻塞等着，等到第一个线程将数据写入缓存后，直接走缓存，**因此只有第一个请求可能会访问到持久层**

### 4. 缓存雪崩（击穿的升级）

条件：大量的热点 key 设置了相同的过期时间，导在缓存在同一时刻全部失效，造成瞬时数据库请求量大、压力骤增，引起雪崩，甚至导致数据库被打挂

解决方案：

1. 过期时间打散
2. 热点数据不过期
3. 加互斥锁或者队列
4. 设置多级缓存：nginx缓存+redis缓存+ehcache缓存

### 5. 分布式锁（多个分布式服务操作同一个redis实例）

举例说明 ，系统A和系统B是两个部署在不同节点的相同应用（集群部署），这时客户端请求传来，两个系统都受到了请求，并且该请求是对数据表进行插入操作，如果这个时候不加锁来控制，可能会导致数据库新增两条记录，这时系统也不能允许的，由于是在不同应用内，在单个应用内加JVM级别的锁，另一个应用是感知不到的，这时需要用到分布式锁

**redis实现分布式锁方式：setnx key value添加锁，del key释放锁**

存在问题以及解决方案：

1. 锁一直没有被释放，可以设置key过期时间，让key自动释放，使用expire命令
2. 上锁之后突然出现异常，无法设置过期时间，可以在上锁的同时设置过期时间，使用set key value nx ex seconds命令（set...nx...相当于setnx，set...ex...相当于expire）

分布式锁存在的问题：

1. 服务A先进行操作，A上锁后由于执行卡顿，锁已经自行过期释放了；服务B抢到锁后，还没完成自己的操作，就轮到A中止卡顿最后释放锁。因此整体而言就是A操作过长导致释放其他服务获取到的锁
   * 解决方案：使用uuid作为key锁的value来标识区分锁，防止不同服务之间误删对方的锁；所有服务，在获取锁后设置key value为uuid，释放锁时必须先判断key value是否为自己服务设置的uuid
2. 在服务A获取锁与释放锁的中间过程中，锁因为自身的过期属性而自动释放，此时A还没手动释放；同时服务B瞬间获取锁并进行业务操作，这时B获取的锁就会被A释放
   * 解决方案：使用lua脚本编写锁的释放过程，*因为锁的释放过程是先获取锁键值判断是否为uuid后再删除锁这两个操作，而问题就发生在这两个操作中锁自动过期被删后其他服务又获取锁*，因此保证锁释放的原子性可以防止其他服务在锁自动过期被删时，无法操作获取锁