

1. redis和map/guava的区别

- map/guava都是使用本地缓存,主要特点,轻量级,便捷,但是生命周期随jvm销毁而结束,多个实例中,每个实例保存一份实例数据,缓存的数据不一致
- redis memcached称为分布式缓存,多个实例的情况下,每个实例保存一份缓存数据,缓存具有一致性,但是程序架构比较复杂

2.redis和memcached的区别

- **redis**支持丰富的数据结构,list hash set k/v,但是memcached只能保存简单的string
- **redis**支持持久化,把内存数据持久化到磁盘,重启redis 重新加载内存,但是memcached只能保存在内存中
- 集群模式,redis支持cluster模式,memcached没有原生的集群模式,需要客户端进行分片写入数据
- **memcached**使用多线程,非阻塞IO复用网络模型,**redis**使用单线程,多路IO复用模型

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

3.redis设置过期时间

定期删除

redis 默认是每间隔 **100s**,就会随机删除一些设置过期的数据

惰性删除

因为定期删除是随机的,所以一部分过期数据没有被删除掉,此时如果客户端访问过期数据,redis 就会把过期数据删除调

4.redis持久化

4.1 RDB

优点

- RDB会形成多个****快照文件(每个快照文件代表某一时刻redis中的数据,可以把完整的数据文件发送到远程安全存储上去,比如说Azon的S3 阿里云的ODPS分布式存储)适合做冷备
- RDB对外提供的读写服务,影响非常小,redis主进程fork一个子进程,让子进程执行IO,进行RDB持久化备份
- 相对于AOP,来说,基于RDB数据文件来重启恢复redis进程更快

缺点

- RDB 数据快照都是每隔5min,或者更长时间,如果宕机,会损失最近5min的数据
- RDB每次fork子进程来备份的时候,如果文件很大,对客户端提供的服务暂停数秒

4.3 AOF

优点

- AOF可以保证,尽可能少的丢失数据,一般AOF每隔1s,通过一个后台线程执行fsync操作
- AOF通过append-only模式写入,没有磁盘询问地址开销,写入性能非常高,文件破损可以使用redis-check-aof命令修复
- AOF日志文件过大的时候,会出现后台重写操作,不会影响客户端的读写, (因为在rewrite log的时候,对其进行压缩,创建一份需要恢复数据的更小日志出来,在创建新文件的时候,还是会写入老的日志文件,当新的merge后的日志文件ready的时候,删除老的文件)
- AOF日志文件的命令通过非常可读的方式进行记录, 这个特性非常适合做灾难性的误删除的紧急恢复(比如某人不小心用flushall命令清空了所有数据, 只要这个时候后台rewrite还没有发生, 那么就可以立即拷贝AOF文件, 将最后一条flushall命令给删了, 然后再将该AOF文件放回去, 就可以通过恢复机制, 自动恢复所有数据)

缺点

- AOF比RDB快照文件大
- AOF开启后,支持写的QPS比RDB支持写的QPS低,(每秒fsync一次日志文件)
- AOF发生过bug,数据恢复的时候没有一模一样的恢复出来

4.3 RDB和AOF到底该如何选择

- 不要仅仅使用RDB, 因为那样会导致你丢失很多数据
- 也不要仅仅使用AOF, 因为那样有两个问题, 第一, 你通过AOF做冷备, 没有RDB做冷备, 来的恢复速度更快; 第二, RDB每次简单粗暴生成数据快照, 更加健壮, 可以避免AOF这种复杂的备份和恢复机制的bug

- 综合使用AOF和RDB两种持久化机制,用AOF来保证数据不丢失,作为数据恢复的第一选择;用RDB来做不同程度的冷备,在AOF文件都丢失或损坏不可用的时候,还可以使用RDB来进行快速的数据恢复

4.4配置RDB持久化

redis.conf文件,也就是/etc/redis/6379.conf,去配置持久化

save 60 1000

- 每隔60s,如果有超过1000个key发生了变更,那么就生成一个新的dump.rdb文件,这个操作被称为 **snapshotting**
- 也可以手动调用save或者bgsave命令,同步或异步执行rdb快照生成
- save可以设置多个,就是多个snapshotting检查点,

4.5RDB持久化工作流程

- redis根据配置自己尝试去生成rdb快照文件
- fork一个子进程出来
- 子进程尝试将数据dump到临时的rdb快照文件中
- 完成rdb快照文件的生成之后,就替换之前的旧的快照文件

4.6配置AOF持久化

- **appendonly yes** 设置为yes 就ok
- 打开appendonly之后,redis每写入一条数据,就会写入os cache中,每个一定时间fsync一下
- aof 和rdb同时开启使用aof
- 可以配置fsync策略
 - always ,每次redis写入一条数据,就会fsync数据到磁盘中,这种性能差
 - everysec 每秒将os cache写入磁盘,性能高QPS可达上万
 - no redis仅仅负责把os cache写入磁盘就不管了,不可控的

4.7AOF rewrite

- redis存储数据是有限的,很多数据会自动过期,或者用户删除,redis 使用自动清除算法删除
- redis 数据被清理调后,但是还保存在aof文件中,aof文件会不断膨胀
- aof每隔一段时间会自动的rewrite操作,保证aof数据最新,不会过大
- redis 2.4之前需要自己写脚本,2.4之后可以通过redis.conf配置
 - auto-aof-rewrite-percentage 100
 - auto-aof-rewrite-min-size 64mb

过程

(1) redis fork一个子进程 (2) 子进程基于当前内存中的数据, 构建日志, 开始往一个新的临时的AOF文件中写入日志 (3) redis主进程, 接收到client新的写操作之后, 在内存中写入日志, 同时新的日志也继续写入旧的AOF文件 (4) 子进程写完新的日志文件之后, redis主进程将内存中的新日志再次追加到新的AOF文件中 (5) 用新的日志文件替换掉旧的日志文件

4.8 AOF修复破损文件

redis-check-aof --fix

(1) 如果RDB在执行snapshotting操作, 那么redis不会执行AOF rewrite; 如果redis再执行AOF rewrite, 那么就不会执行RDB snapshotting (2) 如果RDB在执行snapshotting, 此时用户执行BGREWRITEAOF命令, 那么等RDB快照生成之后, 才会去执行AOF rewrite (3) 同时有RDB snapshot文件和AOF日志文件, 那么redis重启的时候, 会优先使用AOF进行数据恢复, 因为其中的日志更完整

5.企业级备份方案

(1) 写crontab定时调度脚本去做数据备份 (2) 每小时都copy一份rdb的备份, 到一个目录中去, 仅仅保留最近48小时的备份 (3) 每天都保留一份当日的rdb的备份, 到一个目录中去, 仅仅保留最近1个月的备份 (4) 每次copy备份的时候, 都把太旧的备份给删了 (5) 每天晚上将当前服务器上所有的数据备份, 发送一份到远程的云服务上去

```
crontab -e

0 * * * * sh /usr/local/redis/copy/redis_rdb_copy_hourly.sh

redis_rdb_copy_hourly.sh

#!/bin/sh

cur_date=`date +%Y%m%d%k`
rm -rf /usr/local/redis/snapshotting/$cur_date
mkdir /usr/local/redis/snapshotting/$cur_date
cp /var/redis/6379/dump.rdb /usr/local/redis/snapshotting/$cur_date

del_date=`date -d -48hour +%Y%m%d%k`
rm -rf /usr/local/redis/snapshotting/$del_date

每天copy一次备份

crontab -e

0 0 * * * sh /usr/local/redis/copy/redis_rdb_copy_daily.sh

redis_rdb_copy_daily.sh

#!/bin/sh

cur_date=`date +%Y%m%d`
```

```
rm -rf /usr/local/redis/snapshotting/$cur_date
mkdir /usr/local/redis/snapshotting/$cur_date
cp /var/redis/6379/dump.rdb /usr/local/redis/snapshotting/$cur_date

del_date=`date -d -1month +%Y%m%d`
rm -rf /usr/local/redis/snapshotting/$del_date
```

6.AOF RDB数据恢复方案

- redis启动的时候,如果appendonly 是在打开的,redis 会查看是否存在appendonly.aof文件,如果存在直接加载,如果不存在创建,然后加载,因此,数据恢复的时候如果使用rdb恢复,首先要关闭**appendonly**
- 热修改redis 的配置,
- config get appendonly
- config set appendonly set
- 然后重启redis 就ok

7.redis replication(redis 主从架构读写分离) 10w+QPS+水平扩容

7.1核心机制

- redis采用异步的方式将数据复制到slave上, 2.8 开始slave node会周期确定自己复制的数量
- slave node 连接其他slave node
- slave node 在复制的时候不会block master node 是否正常
- slave node做复制的时候,也不会block 对自己的查询操作,会用旧的数据工作,复制完成后,删除旧的数据集,加载新的数据集,这个时候会暂停对外服务
- slave node 可以横向扩容,增加slave node 可以增加读的吞吐量

7.2 主从架构,master 必须持久化

- 如果master 不开启持久化,master 宕机重启后觉得自己数据是空的,slave node 也会复制为空的
- 即使后面使用哨兵模式,高可用,slave node 会自动接管master ,但是如果master 宕机后,sentinal 没有检测到 master failure ,master 重启后也会导致slave 为空

7.3主从架构核心原理

- 当启动一个slave node 时候会给master 发送PSYNC命令
- 如果slave node 重新连接master node,master node会仅仅复制给slave 部分缺失的数据,如果slave node 是第一次连接master ,会发生full resynchronization
- 开始full resynchronization的时候, master会启动一个后台线程,开始生成一份RDB快照文件,同时还会将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后, master会将这个RDB发送给

slave, slave会先写入本地磁盘, 然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave, slave也会同步这些数据。

- slave node如果跟master node有网络故障, 断开了连接, 会自动重连。master如果发现有多个slave node都来重新连接, 仅仅会启动一个rdb save操作, 用一份数据服务所有slave node。

7.4断点续传

从redis 2.8开始, 就支持主从复制的断点续传

master node会在内存中常见一个backlog, master和slave都会保存一个replica offset还有一个master id, offset就是保存在backlog中的。如果master和slave网络连接断掉了, slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset, 那么就会执行一次resynchronization

7.5无磁盘化复制

master在内存中直接创建rdb, 然后发送给slave, 不会在自己本地落地磁盘了

repl-diskless-sync repl-diskless-sync-delay, 等待一定时长再开始复制, 因为要等更多slave重新连接过来

7.6过期key处理

slave不会过期key, 只会等待master过期key。如果master过期了一个key, 或者通过LRU淘汰了一个key, 那么会模拟一条del命令发送给slave。

7.7全量复制

(1) master执行bgsave, 在本地生成一份rdb快照文件 (2) master node将rdb快照文件发送给salve node, 如果rdb复制时间超过60秒(repl-timeout), 那么slave node就会认为复制失败, 可以适当调节大这个参数 (3) 对于千兆网卡的机器, 一般每秒传输100MB, 6G文件, 很可能超过60s (4) master node在生成rdb时, 会将所有新的写命令缓存在内存中, 在salve node保存了rdb之后, 再将新的写命令复制给salve node (5) client-output-buffer-limit slave 256MB 64MB 60, 如果在复制期间, 内存缓冲区持续消耗超过64MB, 或者一次性超过256MB, 那么停止复制, 复制失败 (6) slave node接收到rdb之后, 清空自己的旧数据, 然后重新加载rdb到自己的内存中, 同时基于旧的数据版本对外提供服务 (7) 如果slave node开启了AOF, 那么会立即执行BGREWRITEAOF, 重写AOF

如果复制的数据量在4G~6G之间, 那么很可能全量复制时间消耗到1分半到2分钟

7.8增量复制

(1) 如果全量复制过程中, master-slave网络连接断掉, 那么salve重新连接master时, 会触发增量复制 (2) master直接从自己的backlog中获取部分丢失的数据, 发送给slave node, 默认backlog就是1MB (3) msater就是根据slave发送的psync中的offset来从backlog中获取数据的

offset

master会在自身不断累加offset, slave也会在自身不断累加offset slave每秒都会上报自己的offset给master, 同时master也会保存每个slave的offset

这个倒不是说特定就用在全量复制的，主要是master和slave都要知道各自的数据的offset，才能知道互相之间的数据不一致的情况

backlog

master node有一个backlog，默认是1MB大小 master node给slave node复制数据时，也会将数据在backlog中同步写一份 backlog主要是用来做全量复制中断候的增量复制的

7.9测压

```
./redis-benchmark -h 192.168.31.187
```

8.哨兵模式

（1）集群监控，负责监控redis master和slave进程是否正常工作 （2）消息通知，如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员 （3）故障转移，如果master node挂掉了，会自动转移到slave node上 （4）配置中心，如果故障转移发生了，通知client客户端新的master地址

8.1核心知识

（1）哨兵至少需要3个实例，来保证自己的健壮性 （2）哨兵 + redis主从的部署架构，是不会保证数据零丢失的，只能保证redis集群的高可用性 （3）对于哨兵 + redis主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练

8.2、为什么redis哨兵集群只有2个节点无法正常工作？

哨兵集群必须部署2个以上节点

如果哨兵集群仅仅部署了个2个哨兵实例，quorum=1

```
+-----+ +-----+ | M1 |-----| R1 || S1 || S2 | +-----+ +-----+
```

Configuration: quorum = 1

master宕机，s1和s2中只要有1个哨兵认为master宕机就可以还行切换，同时s1和s2中会选举出一个哨兵来执行故障转移

同时这个时候，需要majority，也就是大多数哨兵都是运行的，2个哨兵的majority就是2（2的majority=2，3的majority=2，5的majority=3，4的majority=2），2个哨兵都运行着，就可以允许执行故障转移

但是如果整个M1和S1运行的机器宕机了，那么哨兵只有1个了，此时就没有majority来允许执行故障转移，虽然另外一台机器还有一个R1，但是故障转移不会执行

4、经典的3节点哨兵集群

```
+-----+
| M1 |
| S1 |
+-----+
|
```

```
+-----+ | +-----+ | R2 |-----+-----| R3 || S2 || S3 | +-----+ +-----+
```

Configuration: quorum = 2, majority

如果M1所在机器宕机了，那么三个哨兵还剩下2个，S2和S3可以一致认为master宕机，然后选举出一个来执行故障转移

同时3个哨兵的majority是2，所以还剩下的2个哨兵运行着，就可以允许执行故障转移

8.3异步同步数据丢失,脑裂问题

min-slaves-to-write 1 min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了

上面两个配置可以减少异步复制和脑裂导致的数据丢失

（1）减少异步复制的数据丢失

有了min-slaves-max-lag这个配置，就可以确保说，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内

（2）减少脑裂的数据丢失

如果一个master出现了脑裂，跟其他slave丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的slave发送数据，而且slave超过10秒没有给自己ack消息，那么就直接拒绝客户端的写请求

这样脑裂后的旧master就不会接受client的新数据，也就避免了数据丢失

上面的配置就确保了，如果跟任何一个slave丢了连接，在10秒后发现没有slave给自己ack，那么就拒绝新的写请求

因此在脑裂场景下，最多就丢失10秒的数据

什么是脑裂?

8.4slave->master选举算法

如果一个master被认为odown了，而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave来

会考虑slave的一些信息

（1）跟master断开连接的时长 （2）slave优先级 （3）复制offset （4）run id

如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕机的时长，那么slave就被认为不适合选举为master

$(\text{down-after-milliseconds} * 10) + \text{milliseconds_since_master_is_in_SDOWN_state}$

接下来会对slave进行排序

(1) 按照slave优先级进行排序, slave priority越低, 优先级就越高 (2) 如果slave priority相同, 那么看 replica offset, 哪个slave复制了越多的数据, offset越靠后, 优先级就越高 (3) 如果上面两个条件都相同, 那么选择一个run id比较小的那个slave

5、quorum和majority

每次一个哨兵要做主备切换, 首先需要quorum数量的哨兵认为odown, 然后选举出一个哨兵来做切换, 这个哨兵还得得到majority哨兵的授权, 才能正式执行切换

如果quorum < majority, 比如5个哨兵, majority就是3, quorum设置为2, 那么就3个哨兵授权就可以执行切换

但是如果quorum >= majority, 那么必须quorum数量的哨兵都授权, 比如5个哨兵, quorum是5, 那么必须5个哨兵都同意授权, 才能执行切换

9.redis cluster

9.1数据分布算法(hash slot)

redis cluster的hash slot算法

redis cluster有固定的16384个hash slot, 对每个key计算CRC16值, 然后对16384取模, 可以获取key对应的hash slot

redis cluster中每个master都会持有部分slot, 比如有3个master, 那么可能每个master持有5000多个hash slot

hash slot让node的增加和移除很简单, 增加一个master, 就将其他master的hash slot移动部分过去, 减少一个master, 就将它的hash slot移动到其他master上去

移动hash slot的成本是非常低的

客户端的api, 可以对指定的数据, 让他们走同一个hash slot, 通过hash tag来实现

9.2多master 写入,海量数据分布式存储

你在redis cluster写入数据的时候, 其实是你可以将请求发送到任意一个master上去执行

但是, 每个master都会计算这个key对应的CRC16值, 然后对16384个hashslot取模, 找到key对应的hashslot, 找到hashslot对应的master

如果对应的master就在自己本地的话, set mykey1 v1, mykey1这个key对应的hashslot就在自己本地, 那么自己就处理掉了

但是如果计算出来的hashslot在其他master上, 那么就会给客户端返回一个moved error, 告诉你, 你得到哪个master上去执行这条写入的命令

elasticsearch建立索引的时候, 先写内存缓存, 每秒钟把数据刷入os cache, 接下来再每隔一定时间fsync到磁盘上去

redis cluster, 写可以到任意master, 任意master计算key的hashslot以后, 告诉client, 重定向, 路由到其他mater去执行, 分布式存储的一个经典的做法

elasticsearch，建立索引的时候，也会根据doc id/routing value，做路由，路由到某个其他节点，重定向到其他节点去执行

9.3 slave的自动迁移

比如现在有10个master，每个有1个slave，然后新增了3个slave作为冗余，有的master就有2个slave了，有的master出现了slave冗余

如果某个master的slave挂了，那么redis cluster会自动迁移一个冗余的slave给那个master

只要多加一些冗余的slave就可以了

为了避免的场景，就是说，如果你每个master只有一个slave，万一一个slave死了，然后很快，master也死了，那可用性还是降低了

但是如果你给整个集群挂载了一些冗余slave，那么某个master的slave死了，冗余的slave会被自动迁移过去，作为master的新slave，此时即使那个master也死了

还是有一个slave会切换成master的

之前有一个master是有冗余slave的，直接让其他master其中的一个slave死掉，然后看有冗余slave会不会自动挂载到那个master

9.4 节点间内部通讯

1、基础通信原理

（1）redis cluster节点间采取gossip协议进行通信

跟集中式不同，不是将集群元数据（节点信息，故障，等等）集中存储在某个节点上，而是互相之间不断通信，保持整个集群所有节点的数据是完整的

维护集群的元数据用得，集中式，一种叫做gossip

集中式：好处在于，元数据的更新和读取，时效性非常好，一旦元数据出现了变更，立即就更新到集中式的存储中，其他节点读取的时候立即就可以感知到；不好在于，所有的元数据的更新压力全部集中在一个地方，可能会导致元数据的存储有压力

gossip：好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续，打到所有节点上去更新，有一定的延时，降低了压力；缺点，元数据更新有延时，可能导致集群的一些操作会有一些滞后

我们刚才做reshard，去做另外一个操作，会发现说，configuration error，达成一致

（2）10000端口

每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如7001，那么用于节点间通信的就是17001端口

每隔节点每隔一段时间都会往另外几个节点发送ping消息，同时其他节点接收到ping之后返回pong

（3）交换的信息

故障信息，节点的增加和移除，hash slot信息，等等

2、gossip协议

gossip协议包含多种消息，包括ping, pong, meet, fail, 等等

meet: 某个节点发送meet给新加入的节点，让新节点加入集群中，然后新节点就会开始与其他节点进行通信

```
redis-trib.rb add-node
```

其实内部就是发送了一个gossip meet消息，给新加入的节点，通知那个节点去加入我们的集群

ping: 每个节点都会频繁给其他节点发送ping，其中包含自己的状态还有自己维护的集群元数据，互相通过ping交换元数据

每个节点每秒都会频繁发送ping给其他的集群，ping，频繁的互相之间交换数据，互相进行元数据的更新

pong: 返回ping和meet，包含自己的状态和其他信息，也可以用于信息广播和更新

fail: 某个节点判断另一个节点fail之后，就发送fail给其他节点，通知其他节点，指定的节点宕机了

3、ping消息深入

ping很频繁，而且要携带一些元数据，所以可能会加重网络负担

每个节点每秒会执行10次ping，每次会选择5个最久没有通信的其他节点

当然如果发现某个节点通信延时达到了 $\text{cluster_node_timeout} / 2$ ，那么立即发送ping，避免数据交换延时过长，落后的时间太长了

比如说，两个节点之间都10分钟没有交换数据了，那么整个集群处于严重的元数据不一致的情况，就会有问题

所以cluster_node_timeout可以调节，如果调节比较大，那么会降低发送的频率

每次ping，一个是带上自己节点的信息，还有就是带上1/10其他节点的信息，发送出去，进行数据交换

至少包含3个其他节点的信息，最多包含总节点-2个其他节点的信息

二、面向集群的jedis内部实现原理

开发，jedis，redis的java client客户端，redis cluster，jedis cluster api

jedis cluster api与redis cluster集群交互的一些基本原理

1、基于重定向的客户端

redis-cli -c，自动重定向

(1) 请求重定向

客户端可能会挑选任意一个redis实例去发送命令，每个redis实例接收到命令，都会计算key对应的hash slot

如果在本地就在本地处理，否则返回moved给客户端，让客户端进行重定向

cluster keyslot mykey，可以查看一个key对应的hash slot是什么

用redis-cli的时候，可以加入-c参数，支持自动的请求重定向，redis-cli接收到moved之后，会自动重定向到对应的节点执行命令

（2）计算hash slot

计算hash slot的算法，就是根据key计算CRC16值，然后对16384取模，拿到对应的hash slot

用hash tag可以手动指定key对应的slot，同一个hash tag下的key，都会在一个hash slot中，比如set mykey1:{100}和set mykey2:{100}

（3）hash slot查找

节点间通过gossip协议进行数据交换，就知道每个hash slot在哪个节点上

2、smart jedis

（1）什么是smart jedis

基于重定向的客户端，很消耗网络IO，因为大部分情况下，可能都会出现一次请求重定向，才能找到正确的节点

所以大部分的客户端，比如java redis客户端，就是jedis，都是smart的

本地维护一份hashslot -> node的映射表，缓存，大部分情况下，直接走本地缓存就可以找到hashslot -> node，不需要通过节点进行moved重定向

（2）JedisCluster的工作原理

在JedisCluster初始化的时候，就会随机选择一个node，初始化hashslot -> node映射表，同时为每个节点创建一个JedisPool连接池

每次基于JedisCluster执行操作，首先JedisCluster都会在本地图算key的hashslot，然后在本地映射表找到对应的节点

如果那个node正好还是持有那个hashslot，那么就ok; 如果说进行了reshard这样的操作，可能hashslot已经不在那个node上了，就会返回moved

如果JedisCluster API发现对应的节点返回moved，那么利用该节点的元数据，更新本地的hashslot -> node映射表缓存

重复上面几个步骤，直到找到对应的节点，如果重试超过5次，那么就报错，JedisClusterMaxRedirectionException

jedis老版本，可能会出现在集群某个节点故障还没完成自动切换恢复时，频繁更新hash slot，频繁ping节点检查活跃，导致大量网络IO开销

jedis最新版本，对于这些过度的hash slot更新和ping，都进行了优化，避免了类似问题

（3）hashslot迁移和ask重定向

如果hash slot正在迁移，那么会返回ask重定向给jedis

jedis接收到ask重定向之后，会重新定位到目标节点去执行，但是因为ask发生在hash slot迁移过程中，所以JedisCluster API收到ask是不会更新hashslot本地缓存

已经可以确定说，hashslot已经迁移完了，moved是会更新本地hashslot->node映射表缓存的

三、高可用性与主备切换原理

redis cluster的高可用的原理，几乎跟哨兵是类似的

1、判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是pfail，主观宕机

如果多个节点都认为另外一个节点宕机了，那么就是fail，客观宕机，跟哨兵的原理几乎一样，sdown，odown

在cluster-node-timeout内，某个节点一直没有返回pong，那么就被认为pfail

如果一个节点认为某个节点pfail了，那么会在gossip ping消息中，ping给其他节点，如果超过半数的节点都认为pfail了，那么就会变成fail

2、从节点过滤

对宕机的master node，从其所有的slave node中，选择一个切换成master node

检查每个slave node与master node断开连接的时间，如果超过了cluster-node-timeout * cluster-slave-validity-factor，那么就没有资格切换成master

这个也是跟哨兵是一样的，从节点超时过滤的步骤

3、从节点选举

哨兵：对所有从节点进行排序，slave priority，offset，run id

每个从节点，都根据自己对master复制数据的offset，来设置一个选举时间，offset越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举

所有的master node开始slave选举投票，给要进行选举的slave进行投票，如果大部分master node ($N/2 + 1$) 都投票给了某个从节点，那么选举通过，那个从节点可以切换成master

从节点执行主备切换，从节点切换为主节点

4、与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，redis cluster功能强大，直接集成了replication和sentinal的功能

10.性能优化

1、fork耗时导致高并发请求延时

RDB和AOF的时候，其实会有生成RDB快照，AOF rewrite，耗费磁盘IO的过程，主进程fork子进程

fork的时候，子进程是需要拷贝父进程的空间内存页表的，也是会耗费一定的时间的

一般来说，如果父进程内存有1G的数据，那么fork可能会耗费在20ms左右，如果是10G~30G，那么就会耗费 $20 * 10$ ，甚至 $20 * 30$ ，也就是几百毫秒的时间

info stats中的latest_fork_usec，可以看到最近一次fork的时长

redis单机QPS一般在几万，fork可能一下子就会拖慢几万条操作的请求时长，从几毫秒变成1秒

优化思路

fork耗时跟redis主进程的内存有关系，一般控制redis的内存存在10GB以内，slave -> master，全量复制

2、AOF的阻塞问题

redis将数据写入AOF缓冲区，单独开一个线程做fsync操作，每秒一次

但是redis主线程会检查两次fsync的时间，如果距离上次fsync时间超过了2秒，那么写请求就会阻塞

everysec，最多丢失2秒的数据

一旦fsync超过2秒的延时，整个redis就被拖慢

优化思路

优化硬盘写入速度，建议采用SSD，不要用普通的机械硬盘，SSD，大幅度提升磁盘读写的速度

3、主从复制延迟问题

主从复制可能会超时严重，这个时候需要良好的监控和报警机制

在info replication中，可以看到master和slave复制的offset，做一个差值就可以看到对应的延迟量

如果延迟过多，那么就进行报警

4、主从复制风暴问题

如果一下子让多个slave从master去执行全量复制，一份大的rdb同时发送到多个slave，会导致网络带宽被严重占用

如果一个master真的要挂载多个slave，那尽量用树状结构，不要用星型结构

5、vm.overcommit_memory

0: 检查有没有足够内存，没有的话申请内存失败 1: 允许使用内存直到用完为止 2: 内存地址空间不能超过swap + 50%

如果是0的话，可能导致类似fork等操作执行失败，申请不到足够的内存空间

```
cat /proc/sys/vm/overcommit_memory echo "vm.overcommit_memory=1" >> /etc/sysctl.conf sysctl vm.overcommit_memory=1
```

6、swappiness

cat /proc/version，查看linux内核版本

如果linux内核版本<3.5，那么swappiness设置为0，这样系统宁愿swap也不会oom killer（杀掉进程） 如果linux内核版本>=3.5，那么swappiness设置为1，这样系统宁愿swap也不会oom killer

保证redis不会被杀掉

```
echo 0 > /proc/sys/vm/swappiness echo vm.swapiness=0 >> /etc/sysctl.conf
```

7、最大打开文件句柄

```
ulimit -n 10032 10032
```

自己去上网搜一下，不同的操作系统，版本，设置的方式都不太一样

8、tcp backlog

```
cat /proc/sys/net/core/somaxconn echo 511 > /proc/sys/net/core/somaxconn
```

11.常见问题

1.缓存雪崩

简介：缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决办法（中华石杉老师在他的视频中提到过，视频地址在最后一个问题中有提到）：

- 事前：尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。
- 事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL崩掉
- 事后：利用 redis 持久化机制保存的数据尽快恢复缓存

—

2.缓存穿透

简介：一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决办法：

- 最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。
- 另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

3.缓存数据库双写数据不一致

问题：先修改数据库，再删除缓存，如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据出现不一致

先删除缓存，再修改数据库，如果删除缓存成功了，如果修改数据库失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致

因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中

4. 如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

参考：

- <https://www.jianshu.com/p/8bddd381de06>