Redis配置文件详解(全网最全的原创版本)

juejin.cn/post/6844904032268451853

> 配置文件版本使用的是redis 4.0.14， 某些参数需要处理了解linux kernel，笔者不太了解linux内核参数，后面还要继续努力呀。

1. 常规命令

1.1 `./redis-server /path/to/redis.conf`

启动redis并使配置文件生效

1.2 `include /path/to/local.conf`

include 可以使用多个配置文件，如果配置文件有相同值，后面的会覆盖前面的:

```
include /path/to/local.conf
include /path/to/other.conf
复制代码
```

1.3 `loadmodule /path/to/my_module.so`

加载modules 没什么用好像，还需要继续研究

```
loadmodule /path/to/my_module.so
loadmodule /path/to/other_module.so
复制代码
```

1.4 `bind 127.0.0.1`

绑定ip地址，为了安全最好都绑定

1.5 `protected-mode yes`

保护模式，如果保护模式开了，而且redis既没有bind ip，也没设置密码，那redis只接收127.0.0.1的连接。 **默认都开**

1.6 `port 6379`

端口，设置为0就不会监听

1.7 `tcp-backlog 511`

linux 内核tcp_max_syn_backlog和somaxconn 参数调优

1.8 `unixsocket /tmp/redis.sock unixsocketperm 700`

unix socket ，默认不监听，没用

1.9 `timeout 0`

连接闲置N秒时关闭连接

1.10 `tcp-keepalive 300`

开启TCP长连接，如果设置非0，会使用系统的SO_KEEPALIVE间隔发送TCP ACK给客户端，以防连接被弃用。这个很有用：

- 检测死掉的连接。
- 如果网络之间还有其他的网络设备，可以连接保活

注意，如果想依靠这个机制关闭连接，可能需要两倍的时间，主要取决于kernel的配置。

默认值是300。

2. 标准配置

2.1 `daemonize yes`

默认情况redis不会按照守护进程的模式去运行。如果你需要，可以设置来开启 注意，如果开启守护进程模式，会生成`/var/run/redis.pid`保存pid

2.2 `supervised no`

这个不太明白，暂时不翻译了，搞懂后更新

```
If you run Redis from upstart or systemd, Redis can interact with your
supervision tree. Options:
  supervised no      - no supervision interaction
  supervised upstart - signal upstart by putting Redis into SIGSTOP mode
  supervised systemd - signal systemd by writing READY=1 to $NOTIFY_SOCKET
  supervised auto    - detect upstart or systemd method based on
                       UPSTART_JOB or NOTIFY_SOCKET environment variables
Note: these supervision methods only signal "process is ready."
      They do not enable continuous liveness pings back to your supervisor.
```

复制代码

2.3 `pidfile /var/run/redis_6379.pid`

pid文件路径 ，默认值`/var/run/redis.pid`如果在非守护进程模式下，而且也没配置pidfile路径，那么不会生成pid文件。如果是守护进程模式， pidfile总会生成，没配置pidfile就会用默认路径。

2.4 `loglevel notice`

指定服务的日志级别：

- debug
- verbose
- notice

- warning

默认 notice

2.5 `logfile ""`

指定redis日志文件名称和路径。你也可以设置 `logfile ""` 强制redis将日志输出的标准输出。 注意，如果你使用标准输出，而且redis使用守护进程模式运行，那log日志会被发送给/dev/null，就没了

2.6 `syslog-enabled no`

```
To enable logging to the system logger, just set 'syslog-enabled' to yes,
and optionally update the other syslog parameters to suit your needs.
复制代码
```

2.7 `syslog-ident redis`

Specify the syslog identity.

2.8 `syslog-facility local0`

Specify the syslog facility. Must be USER or between LOCAL0-LOCAL7.

2.9 `databases 16`

- 使用集群模式时，database就是0
- 设置数据库的数量。redis默认的数据库就是0，你可以选择不同的数据库，在一个redis连接中执行selcet ，dbid可选的范围是0~(databases-1)，默认就是0~15

2.10 `always-show-logo yes`

搞笑配置，永远显示redis的logo

3. 快照相关

3.1 开启RDB持久化

开启RDB持久化， `save <seconds> <changes>`
`save 900 1` :就是900秒有一次更改就做一次rdb快照到磁盘。
禁用rdb就是注释掉save行，如果你配置了 `save ""` ，也可以禁用rdb。
下面是默认值

```
save 900 1
save 300 10
save 60 10000
复制代码
```

3.2 `stop-writes-on-bgsave-error yes`

默认情况下，如果RDB快照功能开启而且最后一次rdb快照save失败时，redis会停止接收写请求， 这其实就是一种强硬的方式来告知用户数据持久化功能不正常，否则没有人会知道当前系统出大问题了。 如果后台save进程正常工作了(正常保存了rdb文件)，那么redis会自动允许写请求。 不过如果你已经设置了一些监控到redis服务器，你可能想要禁用这个功能，这样redis在磁盘出问题时依旧可以继续处理写请求。只要set `stop-writes-on-bgsave-error yes`

3.3 `rdbcompression yes`

使用LZF算法对rdb文件进行压缩，如果要节省一些CPU，可以设置为no。

3.4 `rdbchecksum yes`

自从redis 5.0，rdb文件的末尾会设置一个CRC64校验码(循环冗余码)。这可以起到一定的的纠错作用，但是也要 付出10%的性能损失，你可以关闭这个功能来获取最大的性能。 如果rdb文件校验功能关闭，那么系统读取不到检验码时会自动跳过校验。 `rdbchecksum yes`

3.5 `dbfilename dump.rdb`

rdb文件名

3.6 `dir ./`

redis的工作目录 ，aof文件，rdb文件还有redis cluster模式下的node.conf文件均会创建在这个目录下。

3.7 `slaveof <masterip> <masterport>`

主从复制。使用slaveof配置将redis实例变为其他redis服务器的一个拷贝。

- redis的主从复制时异步的，但是当主节点无法连接到给定数量的从节点时，你可以设置主节点停止处理写请求
- 如果主从复制断了一小段时间，redis从节点可以执行一次局部的重新同步，你可能需要设置复制的backlog size
- 主从复制时自动的无需用户干预。如果网络中间断了，从节点会自动重连主节点，并发起一次重新同步。

3.8 `masterauth <master-password>`

如果主节点有密码，从节点必须配置这个密码，否则主节点拒绝复制请求。

3.9 `slave-serve-stale-data yes`

当主从同步失败时，从节点有两种行为：

- 配置为yes，从节点可以继续响应客户端的请求。
- 配置为no，从节点直接报错"SYNC with master in progress"，不过INFO和SALVEOF命令是可以执行的。

3.10 `slave-read-only yes`

你可以设置从节点能够处理写请求。向从节点写入一些临时数据有时候是有用的(因为数据在resync后很快就会删除)，如果配错了也可能会造成一些问题。 **2.6版本以后默认都是read-only**。 read-only不是设计成对抗那些不可信的客户端的。只是怕客户端用错命令。read-only模式下一些管理类命令还是会输出的。如果要限制这种命令，你可以使用rename-command来重命名那些管理类命令

3.11 `repl-diskless-sync no`

主从同步策略：disk或socket。

警告：diskless复制目前只是试验阶段 当出现新的从节点或重连的从节点无法进行增量同步时，就需要做一次全量同步(full synchronization)。 一个RDB文件会从主节点传输到从节点，传输方式有两种：

- disk-backed：主节点创建一个新的进程将RDB文件写到磁盘。然后这个文件会被主进程逐步传送给多个从节点
- diskless:主节点创建一个新的进程，直接将RDB文件写给从节点的socket连接，从头到尾不会碰磁盘。

使用disk-backed复制，在rdb文件生成完毕后，主节点会为每个从节点创建队列来传说RDB文件，直到传输结束。 使用diskless复制，一旦开始传输rdb，当时有多少从节点建立连接，就只能并行传输多少从节点，如果此时有新的从节点发起全量同步，就只能等之前的都传完。 如果使用diskless复制，主节点会在传输之前等待一小段时间(这个时间可以配置)，这样可以让多个从节点到达 ，并做并行传输。 **如果磁盘贼慢，网络带宽特别好，diskless复制策略效果会更好一些。**

3.12 `repl-diskless-sync-delay 5`

如果开启了diskless复制，需要配置一个延迟时间，让主节点等待所有从节点都到达。 这是非常重要的，因为一旦开始传输，主节点就无法响应新的从节点的全量复制请求，只能先到队列中等待下一次RDB传输，所以主节点需要等待一段时间，让所有从节点全量复制都到达。 这个延迟时间的单位是秒，默认是5秒。关闭这个特性可以将其设置成0，这样传输总是马上开始。

3.13 `repl-ping-slave-period 10`

从节点在一定间隔时间发送ping到主节点。默认是是10秒。

3.14 `repl-timeout 60`

这个值对三个场景都有效：

1. 大量的I/O操作，从节点收到主节点的响应时间。
2. 从节点认为主节点的超时时间
3. 主节点认为从节点的超时时间

注意，这个值一定要设置的比 `repl-ping-slave-period` 大，否则每次心跳检测都超时

3.15 `repl-disable-tcp-nodelay no`

在从节点socket 发起SYNC同步后是否需要关闭TCP_NODELAY？ 如果选择YES，redis会使用较小的tcppacket和较小的带宽去发送数据到从节点。但是这会让主从复制增加部分延迟，差不多40毫秒，取决于linux kernel配置。 如果选择no，主从复制延迟会稍微减少，但是会消耗更大的网络带宽。 默认我们倾向于低延迟，但是如果网络状况不好的情况时将这个选项置为yes或许是个好方案。

3.16 `repl-backlog-size 1mb`

设置主从复制backlog大小。backlog是一个缓冲区。 当主从不同步时，主节点缓存主从复制数据到backlog缓冲区中，当从节点重新连接到主节点时，从节点可以从缓冲区中拿到增量同步数据，并进行增量同步(partitial synchronization)。 backlog越大，允许从节点断线的时间就越长。backlog缓冲区只有在最少有一个从节点连接时才会创建。

3.17 `repl-backlog-ttl 3600`

如果主节点再也没有连接到从节点，那个从节点的backlog会被释放。 当从节点断线开始，这个配置的时间就开始计时了。**单位是秒**。 设置为0说明永远都不会释放backlog。

3.18 `slave-priority 100`

这个配置是给哨兵模式用的，当主节点挂掉时，哨兵会选取一个priority最小的从节点去升主，如果某个redis节点的这个值配成0，那么这个节点永远都不会被升为主节点。默认值就是100。

3.19 `min-slaves-to-write 3`和`min-slaves-max-lag 10`

如果lag秒内主节点在线的从节点少于N个，主节点停止接收写请求。 例如10秒内最少3个从节点在线时，主节点才接受写请求，可以用如下配置：

```
min-slaves-to-write 3
min-slaves-max-lag 10
复制代码
```

将这两个配置任意一个设置为0，就禁用此功能。默认是禁用的。

3.20 `slave-announce-ip 5.5.5.5` 和 `slave-announce-port 1234`

有多种方式可以显示主节点当前在线的从节点的ip和端口。 例如，info replication 部分，或者在主节点执行ROLE命令。

```
#
# The listed IP and address normally reported by a slave is obtained
# in the following way:
#
#   IP: The address is auto detected by checking the peer address
#   of the socket used by the slave to connect with the master.
#
#   Port: The port is communicated by the slave during the replication
#   handshake, and is normally the port that the slave is using to
#   list for connections.
#
# However when port forwarding or Network Address Translation (NAT) is
# used, the slave may be actually reachable via different IP and port
# pairs. The following two options can be used by a slave in order to
# report to its master a specific set of IP and port, so that both INFO
# and ROLE will report those values.
#
# There is no need to use both the options if you need to override just
# the port or the IP address.
#
# slave-announce-ip 5.5.5.5
# slave-announce-port 1234
```
复制代码

4. 安全

4.1 `requirepass foobared`

给redis设置密码，因为redis快的一逼，一秒钟攻击者能尝试150000次密码，所以你的密码必须非常强壮否则很容易被**暴力破解**。

4.2 `rename-command CONFIG ""`和`rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52`

完全杀掉一个命令就用`rename-command CONFIG ""`
`rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52`可以将命令改掉，这样彩笔程序员就不会使用危险命令了。

注意，如果你把命令给改名了，那么从节点什么的都要统一改名字，否则会有问题。

5. 客户端

5.1 `maxclients 10000`

设置同一时刻的最大客户端数。 默认值是10000，只要达到最大值，redis会关闭所有新的链接，并且发送一个错误“max number of clients readched”给客户端。

6. 内存管理

6.1 maxmemory

设置一个内存的最大值。当内存达到最大值之后，redis会按照选择的内存淘汰策略去删除key。 如果redis根据淘汰策略无法删除key，或者淘汰策略是noeviction，客户端发送写请求时redis会开始返回报错，并且不会使用更多的内存。但是读请求还是会继续支持的。 注意，如果你有很多从节点，那么内存设置不能太大，否则从节点发起全量同步时，output buffer占用的内存也在这个maxmemory的范围内，例如，最大值配的是4GB，如果内存已经3G了，此时一个从节点发起全量同步，outputbuffer你设置的是2G这样内存直接就满了，然后就要开始淘汰key，这肯定不是我们想要的。

6.2 `maxmemory-policy noeviction`

内存淘汰策略，决定了当redis内存满时如何删除key。 默认值是noeviction。

- volatile-lru -> 在过期key中使用近似LRU驱逐
- allkeys-lru -> 在所有key中使用近似LRU
- volatile-lfu -> 在过期key中使用近似LFU驱逐
- allkeys-lfu -> 在所有key中使用近似LFU
- volatile-random -> 在过期key中随机删除一个
- allkeys-random -> 在所有的key中随机删除一个
- volatile-ttl -> 谁快过期就删谁
- noeviction -> 不删除任何key，内存满了直接返回报错

LRU means Least Recently Used LFU means Least Frequently Used

LRU, LFU and volatile-ttl 基于近似随机算法实现。 注意，使用上述策略时，如果没有合适的key去删除时，redis在处理写请求时都会返回报错。

写明令: set setnx setex append incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby getset mset msetnx exec sort。

6.3 `maxmemory-samples 5`

LRU, LFU and minimal TTL algorithms 不是精确的算法，是一个近似的算法(主要为了节省内存)， 所以你可以自己权衡速度和精确度。默认redis会检查5个key，选择一个最近最少使用的key，你可以改变这个数量。默认的5可以提供不错的结果。你用10会非常接近真实的LRU但是会耗费更多的CPU，用3会更快，但是就不那么精确了。

7. LAZY FREEZING 懒释放

redis有两个删除key的基本命令。一个是DEL，这是一个阻塞的删除。DEL会让redis停止处理新请求，然后redis会用一种同步的方式去回收DEL要删除的对象的内存。如果这个key对应的是一个非常小的对象，那么DEL的执行时间会非常短，接近O(1)或者O(log n)。不过，如果key对应的对象很大,redis就会阻塞很长时间来完成这个命令。鉴于上述的问题，redis也提供了非阻塞删除命令，例如UNLINK(非阻塞的DEL)和异步的删除策略：FLUSHALL和FLUSHDB，这样可以在后台进行内存回收。这些命令的执行时间都是常量时间。一个新的线程会在后台渐进的删除并释放内存。

上面说的那些命令都是用户执行的，具体用哪种命令，取决于用户的场景。 但是redis本身也会因为一些原因去删除key或flush掉整个内存数据库。 除了用户主动删除，redis自己去删除key的场景有以下几个：

- 内存淘汰(eviction)，设置了内存淘汰策略后，为了给新数据清理空间，需要删除被淘汰的数据，否则内存就爆了。
- 过期(expire)，当一个key过期时
- key已经存在时的一些边际影响。例如，set一个已经存在的key，旧的value需要被删除，然后设置新的key。
- 主从复制时，从节点执行一个全量同步，从节点之前的内存数据需要被flush掉。

如果你希望上面那四种场景使用异步删除，可以使用如下配置：

```
lazyfree-lazy-eviction no
lazyfree-lazy-expire no
lazyfree-lazy-server-del no
slave-lazy-flush no
复制代码
```

8. AOF

8.1 `appendonly no`

默认情况下，redis异步的dump内存镜像到磁盘(RDB)。这个模式虽然已经很不错了，但是如果在发起dump之前机器宕机，就会丢失一些数据。 AOF(Append only file)是一种可选的持久化策略提供更好数据安全性。使用默认配置的情况下， redis最多丢失一秒钟的写入数据，你甚至可以提高级别，让redis最多丢失一次write操作。 AOF和RDB持久化可以同时开启。如果开了AOF，redis总会先加载AOF的文件，因为AOF提供更高的可用性。

8.2 `appendfilename "appendonly.aof"`

aof 文件的名称。

8.3 `appendfsync everysec`

对操作系统的fsync()调用告诉操作系统将output buffer中的缓冲数据写入到磁盘。有些操作系统会 真正的写磁盘，有一些会尽量去写，也可能会等一下。 redis 支持三种方式：

- no: 不去主动调用fsync()，让操作系统自己决定何时写磁盘
- always：每次write操作之后都调用fsync()，非常慢，但是数据安全性最高。
- everysec:每秒调用一次fsync()，一个折中的策略。

默认就是everysec，一般也是推荐的策略，平衡了速度和数据安全性。

```
appendfsync always
appendfsync everysec
appendfsync no
复制代码
```

8.4 `no-appendfsync-on-rewrite no`

当AOF fsync 策略设置成always或者everysec，而且一个后台的save进程(可能RDB的bgsave进程，也可能是 AOF rewrite进程)正在执行大量磁盘I/O操作，在一些linux配置中，redis可能会对fsync()执行太长的调用。 这个问题目前没什么办法修复，也就是说就算起一个后台进程去做fsync，如果之前已经有进程再做fsync了，后来的调用 会被阻塞。 为了缓和这个问题，可以使用下面的配置，当已经有BGSAVE和BGREWRITEAOF在做fsync()时，就不要再起新进程 了。 如果已经有子进程在做bgsave或者其他的磁盘操作时，redis无法继续写aof文件，等同于appendsync none。 在实际情况中，这意味着可能会丢失多达30秒的日志。也就是说，这是会丢数据的，如果对数据及其敏感，要注意这个问题。如果你有延迟类问题，可以设置成yes,否则设置为no，这样能保证数据的安全性最高，极少丢数据。

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

自动重写aof文件。当aof文件增大到某个百分比时，redis会重写aof文件。 redis会记住上次rewrite后aof文件的大小（如果启动后还没发生过rewrite，那么会使用aof原始大小）。 这个size大小会和当前aof文件的size大小做比较。如果当前size大于指定的百分比，就做rewrite。 并且，还要指定最小的size，如果当前aof文件小于最小size，不会触发rewrite，这是为了防止文件其实很小，但是 已经符合增长百分比时的多余的rewrite操作。 如果指定percentage为0代表禁用aof rewrite功能

8.6 `aof-load-truncated yes`

当Redis启动时会加载AOF文件将数据还原到内存中，但是有时候这个AOF的文件可能被损坏掉了 ，例如文件末尾是坏的。这种情况一般都是由于redis宕机导致的，尤其是使用ext4文件系统挂载 时没配置 data=ordered选项。 在这种情况下，redis可以直接报错，或者尽可能的读取剩余可读的AOF文件。

如果 aof-load-truncated=yes，redis依然会读取这个损坏的aof文件，但是会打出一个报错日志， 通知用户。 如果 aof-load-truncated=no，redis就会报错并拒绝启动服务，用户需要使用redis-check-aof工具 修复aof文件，再启动redis。 如果redis运行时aof文件崩溃，redis依然会报错并退出。这个选项救不了这种情况。

8.7 `aof-use-rdb-preamble no`

当redis重写aof文件时，redis可以先读一个rdb来加快重写的速度，当这个选项打开时，重写的aof文件由 两部分组成：rdb文件+aof文件。 当redis启动时加载的aof文件以"REDIS"开头，就会加载rdb文件，然后再读取剩余的AOF文件。 默认这个选项是关闭的，

9. LUA脚本

9.1 `lua-time-limit 5000`

表示一个lua脚本的最大执行毫秒数。 如果执行时间达到了最大时间，redis会log这个脚本已经超时了，并且会报个error。 当一个脚本执行超时，只有 `SCRIPT KILL` 和 `SHUTDOWN NOSAVE` 命令是可用的。第一个命令可以去 停止一个不包含写命令的脚本。第二个命令是

唯一一个可以停掉超时写命令的脚本。 将lua-time-limit设置成0或负数表示你不限制执行时间，并且不会有任何警告。

10. Redis Cluster

10.1 `cluster-enabled yes`

普通的redis实例无法成为cluster的一员的；只有node可以。想要将节点加入redis cluster，需要将cluster-enabled设置为yes。

10.2 `cluster-config-file nodes-6379.conf`

所有cluster node都有一个cluster配置文件。这个文件不是为了人工编辑的，是redis自己创建的。 每个redis-node都要使用不同的cluster配置文件。一定要确保运行在同一个系统中的多个redis cluster节点使用的是不同的redis配置文件，不要互相覆盖。

10.3 `cluster-node-timeout 15000`

cluster node timeout是一个节点无响应的最长毫秒数。大多数超时时间限制都是这个值的倍数。

10.4 `cluster-slave-validity-factor 10`

一个主节点宕机后，如果它的从节点A数据太旧(长期处于未同步状态)，那么A不会触发failover， 它不会升为主。 **没有一个简单方式去策略一个从节点的“数据年龄”**。下面提供了两种方式来评估从节点的数据是否过老：

- 如果有多个从节点都可以failover，他们会交换信息选出一个拥有最大复制offset的从节点(这说明这个节点从主节点那里复制了更多的数据)。各个从节点会计算各自的offset级别，在开始failover之前会延迟一段时间，具体多久取决于他们的offset级别。
- 每个从节点计算上次和主节点交互的时间。这个交互可以是最后一次ping操作，或者是主节点推送过来的写命令，再或者是上次和主节点断开的时间。如果上次交互时间已经过去太久了，这个从节点就根本不会发起failover。

第二点用户可以自行调整。如果一个从节点和主节点上次交互时间大于`(node-timeout * slave-validity-factor) + repl-ping-slave-period` ，从节点就不会发生failover。例如，如果node-timeout=30秒，slave-validity-factor=10，repl-ping-slave-period=10秒， 如果从节点与主节点上次交互时间已经过去了310秒，那么从节点就不会做failover。调大slave-validity-factor会允许从节点持有过旧的数据时提升为主节点，调小这个值可能会 导致从节点永远都无法升为主节点。 考虑最高的可用性，可以将 `slave-validity-factor` 设置为0，这样从节点会忽略和主节点的上次 交互时间，永远都会尝试去做failover。(但是依然会做延迟选举的操作)

10.5 `cluster-migration-barrier 1`

从节点可以迁移至孤儿主节点(这种主节点没有从节点)。 从节点只有在原来的主节点最少有N个从节点时才会迁移到其他的孤儿主节点，这个给定的数字N就是migration-barrier，也叫迁移临界点。migration barrier=1代表，主节点如果有2个从节点， 当集群中出现孤儿主节点时，其中一个从节点可以被迁移过去。 想要禁止从节点迁移可以将这个值设置成很大的值，例如999。 只有在debug模式才可以将这个值设置为0，生产环境别乱设置。

10.6 `cluster-require-full-coverage yes`

默认情况下，redis cluster在发现还有最少1个hash slot没有被分配时会禁止查询操作。 **这样的话，如果cluster出现部分宕机时，整个集群就不可用了。**只有在其他的hash slot都被分配才可以。 你可能会需要cluster的子集可以继续提供服务，要想这样，只要设置`cluster-require-full-coverage no`即可

10.7 `cluster-slave-no-failover no`

这个选项如果设置为yes，在主节点宕机是，从节点永远都不会升为主。但是主节点依然可以执行常规的failover。 在多数据中心的场景下，这个配置会比较有用，我们希望某一个数据中心永远都不要升级为主节点，否则主节点就漂移到别的数据中心了，这可能挺麻烦的。

11. CLUSTER DOCKER/NAT 支持

11.1 集群主动告知ip

在一些特定的部署场景下，redis cluster 节点地址自动发现会失败，因为地址被NAT了，或者端口 被转发了(Docker容器中)。 为了让redis cluster在这种环境下正常工作，需要静态配置地址和端口，具体配置如下：

- cluster-announce-ip 10.10.10.10
- cluster-announce-port 6379
- cluster-announce-bus-port 6380

如果配置文件中没有上述配置项，那么rediscluster会使用标准的自动发现机制。

12. SLOW LOG

12.1 `slowlog-log-slower-than 10000` 和 `slowlog-max-len 128`

redis slow log是系统记录慢操作的，只要超过了给定的时间，都会记录。 执行时间不包括I/O操作的时间。 你可以通过两个参数来配置slow log：

- slowlog-log-slower-than 10000:单位是微秒，1000000等于1秒
- slowlog-max-len 128：slow长度，如果命令大于128，老的那个就没了，这个值没有限制，如果设置太大会占内存

可以通过SLOWLOG RESET命令来重置这个队列。

13. LATENCY MONITOR

13.1 `latency-monitor-threshold 0`

redis 延迟监控系统会在运行时抽样一部分命令来帮助用户分析redis卡顿的原因。 通过 `LATENCY` 命令可以打印一些视图和报告。 redis只会记录那些大于设定毫秒数的命令。 如果要关闭这个功能，就将 `latency-monitor-threshold` 设置为0。 默认情况下monitor是关闭的，没有延迟问题不要一直开着monitor，因为开这个功能可能会对性能有很大影响。 在运行时也可以开这个功能，执行这个命令即可： `CONFIG SET latency-monitor-threshold <milliseconds>`

14. 事件通知

14.1 `notify-keyspace-events ""`

当特定的key space有事件发生时，redis 可以通知 pub/sub 客户端。 如果开启事件通知功能，一个client对key"foo"执行了del操作，通过pub/sub，两条消息会被推送：

- PUBLISH **keyspace@0**:foo del
- PUBLISH **keyevent@0**:del foo

可以选择redis通知的事件级别。所有的级别都被标记为一个单独的字符：

- K Keyspace events, published with **keyspace@** prefix.
- E Keyevent events, published with **keyevent@** prefix.
- g Generic commands (non-type specific) like DEL, EXPIRE, RENAME, ...
- $ String commands
- l List commands
- s Set commands
- h Hash commands
- z Sorted set commands
- x Expired events (events generated every time a key expires)
- e Evicted events (events generated when a key is evicted for maxmemory)
- A Alias for g$lshzxe, so that the "AKE" string means all the events.

notify-keyspace-events参数接受多个字符，或者0个字符串。如果将notify-keyspace-events设置为 空字符串，就等于禁用通知。

15. 高级设置

15.1 ziplist相关配置

当数据量很少时，哈希值可以使用一种更高效的数据结构。这个阈值可以使用以下的配置来设置：

```
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
复制代码
```

15.2 `list-max-ziplist-size -2`

list也可以使用一种特殊编码方式来节省内存。list底层的数据结构是quicklist，quicklist的每一个 节点都是一个ziplist，这个参数主要来控制每个ziplist的大小，如果配置正数， 那么quicklist每个ziplist中的节点数最大不会超过配置的值。 如果配置负数，就是指定ziplist的长度：

- -5: max size: 64 Kb <-- not recommended for normal workloads
- -4: max size: 32 Kb <-- not recommended
- -3: max size: 16 Kb <-- probably not recommended
- -2: max size: 8 Kb <-- good
- -1: max size: 4 Kb <-- good

配置-2和-1是性能最高的

15.3 `list-compress-depth 0`

list也可以被压缩。 list底层是一个双向链表，压缩深度代表除了head和tail节点有多少node不会被压缩。 head和tail节点是永远都不会被压缩的。

- 0: 关闭压缩
- 1: 代表除了head和tail之外所有的内部节点都会被压缩 [head]->node->node->...->node->[tail] [head], [tail] 不会被压缩; 内部 nodes 会被压缩.
- 2: [head]->[next]->node->node->...->node->[prev]->[tail] head 、 head->next 、 tail->prev 和 tail四个节点不会被压缩, 他们之间的其他节点会被压缩.
- 3: [head]->[next]->[next]->node->node->...->node->[prev]->[prev]->[tail]

以此类推

15.4 `set-max-intset-entries 512`

set也支持内部优化，当set内部元素都是64位以下的十进制整数时，这个set的底层实现会使用intset， 当添加的元素大于set-max-intset-entries时，底层实现会由intset转换为dict。

15.5 `zset-max-ziplist-entries 128` 和 `zset-max-ziplist-value 64`

当zset内部元素大于128，或者value超过64字节时，zset底层将不再使用ziplist

15.6 `hll-sparse-max-bytes 3000`

HyperLogLog稀疏表示字节限制。这个限制包括16字节的header。当HyperLogLog使用稀疏表示时，如果 达到了这个限制，它将会转换成紧凑表示。 这个值设置成大于16000是没意义的，因为16000时用紧凑表示对内存会更友好。 推荐的值是0~3000，这样可以布降低PFADD命令的执行时间时还能节省空间。如果内存空间相对cpu资源更紧张， ，可以将这个值提升到10000。

15.7 `activerehashing yes`

动态rehash使用每100毫秒中的1毫秒来主动为Redis哈希表做rehash操作。Redis的哈希表实现默认使用 一种lazy rehash模式：你对hash表的操作越多，越多rehash步骤会被执行，所以如果服务在空闲状态下， rehash操作永远都不会结束，而且hash表会占用更多的内存。 默认会每秒做10次动态rehash来释放内存。

15.8 客户端传输缓冲区相关配置

client output buffer限制是用来强制断开client连接的，当client没有及时将缓冲区的数据读取完 时，redis会认为这个client可能出现宕机，就会断掉连接。

这个限制可以根据三种不同的情况去设置：

- normal -> 一般的clients包括MONITOR client
- slave -> 从节点 clients
- pubsub -> pub/sub clients

设置语法如下： `client-output-buffer-limit <class> <hard limit> <soft limit> <soft seconds>`

- hard limit:如果缓冲区的数据达到了hard limit的值，redis直接断开和这个客户端的连接
- soft limit & soft seconds:如果缓冲区的数据达到了soft limit的值，redis和这个client的连接还会保留soft seconds

默认情况下normal的clients不会有这个限制，因为normal的clients获取数据都是先执行个命令，不存在 redis主动给normal推送数据的情况。 如果设置三个0代表无限制，永远不断连接。 但是这样可能会撑爆内存。

```
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
复制代码
```

15.9 `client-query-buffer-limit 1gb`

client query buffer缓冲新的命令。默认使用一个固定数量来避免protocol desynchronization。

15.10 `proto-max-bulk-len 512mb`

redis协议中，大多数的请求，都是string，默认value不会大于512mb，当然，你可以改这个限制。

15.11 `hz 10`

redis调用一些内部函数来执行很多后台任务，像关闭超时连接，清理从未请求的过期的key等等。 不是所有的后台任务都使用相同的频率来执行，redis使用hz参数来决定执行任务的频率。 默认hz是10.提高这个值会在redis空闲时消耗更多的cpu，但是同时也会让redis更主动的清理过期 key，而且清理超时连接的操作也会更精确。 这个值的范围是1~500,不过并不推荐设置大于100的值。多数的用户应该使用默认值，或者最多调高到100。

15.12 `aof-rewrite-incremental-fsync yes`

当子进程重写aof文件是，如果这个功能开启，redis会以每32MB的数据主动提交到文件。这种 递增提交文件到磁盘可以避免大的延迟尖刺。

15.13 LFU调优

redis lfu淘汰策略可以调优。 每个key的LFU counter只有8个key，最大值是255，所以redis使用一个基于概率的对数增长算法， 并不是每次访问key都会counter+1。当一个key被访问后，会按照以下方式去做counter+1:

1. 在0~1之间生成一个随机数R
2. 计算概率P=1/(old_value*lfu_log_factor+1)
3. r<p就counter+1

默认 lfu-log-factor 是 10. 下面是不同的factor下的增长速度，可以看到，factor越小增长越快:

```
+--------+------------+------------+------------+------------+------------+
| factor | 100 hits   | 1000 hits  | 100K hits  | 1M hits    | 10M hits   |
+--------+------------+------------+------------+------------+------------+
| 0      | 104        | 255        | 255        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 1      | 18         | 49         | 255        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 10     | 10         | 18         | 142        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 100    | 8          | 11         | 49         | 143        | 255        |
+--------+------------+------------+------------+------------+------------+
```

注意: 上表结论是执行以下命令得出的:

```
redis-benchmark -n 1000000 incr foo
redis-cli object freq foo
```

注意 2: counter初始值是5,否则key很快就被淘汰了
复制代码

lfu-decay-time我还不太理解，等研究透彻后补充

lfu-log-factor 10 lfu-decay-time 1

16. 动态碎片整理(处于试验阶段，暂不翻译)

```
# WARNING THIS FEATURE IS EXPERIMENTAL. However it was stress tested
# even in production and manually tested by multiple engineers for some
# time.
#
# What is active defragmentation?
# -------------------------------
#
# Active (online) defragmentation allows a Redis server to compact the
# spaces left between small allocations and deallocations of data in memory,
# thus allowing to reclaim back memory.
#
# Fragmentation is a natural process that happens with every allocator (but
# less so with Jemalloc, fortunately) and certain workloads. Normally a server
# restart is needed in order to lower the fragmentation, or at least to flush
# away all the data and create it again. However thanks to this feature
# implemented by Oran Agra for Redis 4.0 this process can happen at runtime
# in an "hot" way, while the server is running.
#
# Basically when the fragmentation is over a certain level (see the
# configuration options below) Redis will start to create new copies of the
# values in contiguous memory regions by exploiting certain specific Jemalloc
# features (in order to understand if an allocation is causing fragmentation
# and to allocate it in a better place), and at the same time, will release the
# old copies of the data. This process, repeated incrementally for all the keys
# will cause the fragmentation to drop back to normal values.
#
# Important things to understand:
#
# 1. This feature is disabled by default, and only works if you compiled Redis
#    to use the copy of Jemalloc we ship with the source code of Redis.
#    This is the default with Linux builds.
#
# 2. You never need to enable this feature if you don't have fragmentation
#    issues.
#
# 3. Once you experience fragmentation, you can enable this feature when
#    needed with the command "CONFIG SET activedefrag yes".
#
# The configuration parameters are able to fine tune the behavior of the
# defragmentation process. If you are not sure about what they mean it is
# a good idea to leave the defaults untouched.

# Enabled active defragmentation
# activedefrag yes

# Minimum amount of fragmentation waste to start active defrag
# active-defrag-ignore-bytes 100mb

# Minimum percentage of fragmentation to start active defrag
# active-defrag-threshold-lower 10

# Maximum percentage of fragmentation at which we use maximum effort
# active-defrag-threshold-upper 100

# Minimal effort for defrag in CPU percentage
# active-defrag-cycle-min 25

# Maximal effort for defrag in CPU percentage
```

```
# active-defrag-cycle-max 75
复制代码
```