# 从零搭建4个CentOS集群

## 1.新建一个虚拟机

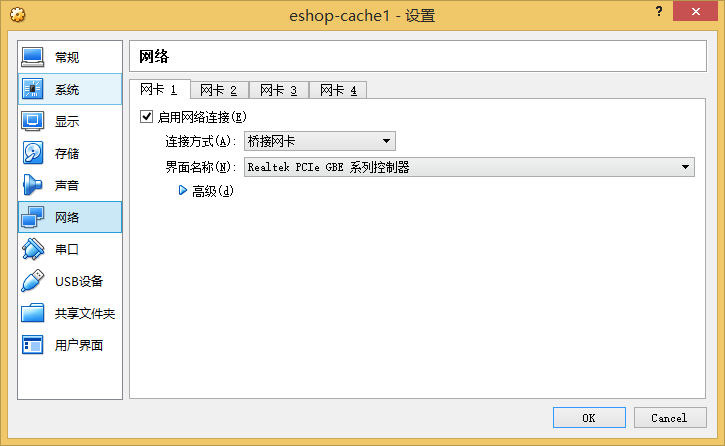
### 1.创建虚拟机，并设置磁盘大小



创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为eshop-cache01，选择操作系统为Linux，选择版本为Red Hat，分配1024MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

### 2.设置虚拟网络

点击虚拟机，点设置，选择网络将，网络设置成桥接网络。

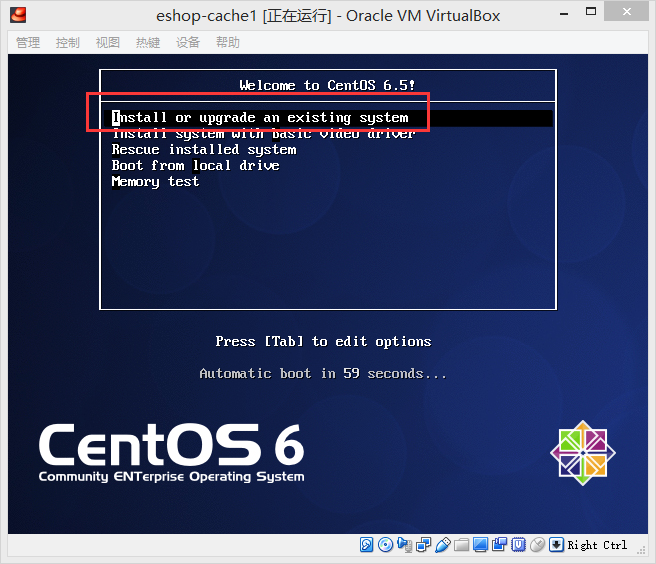


### 3.启动虚拟机，安装CentOS.

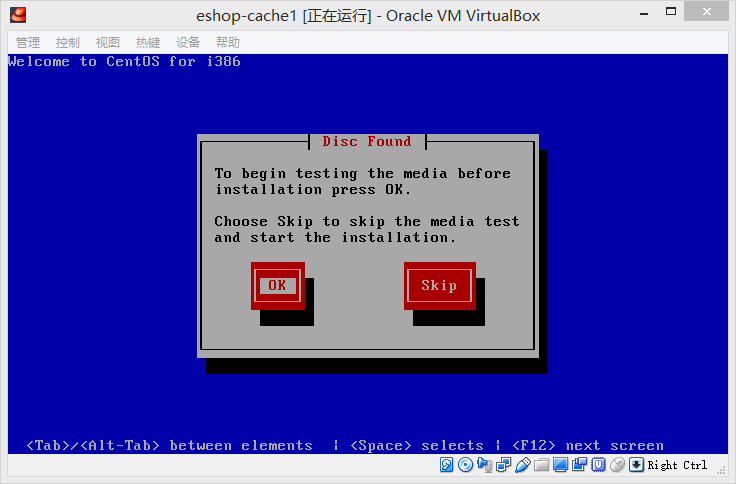
1.选择安装镜像文件，点启动。

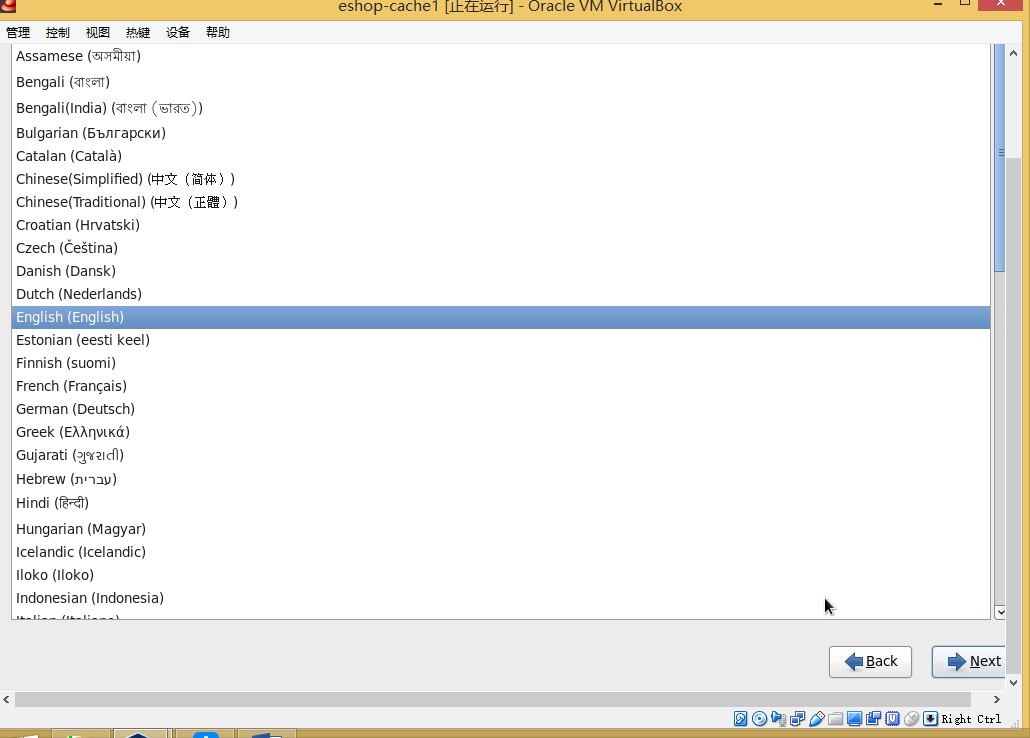


1. 选择安装系统

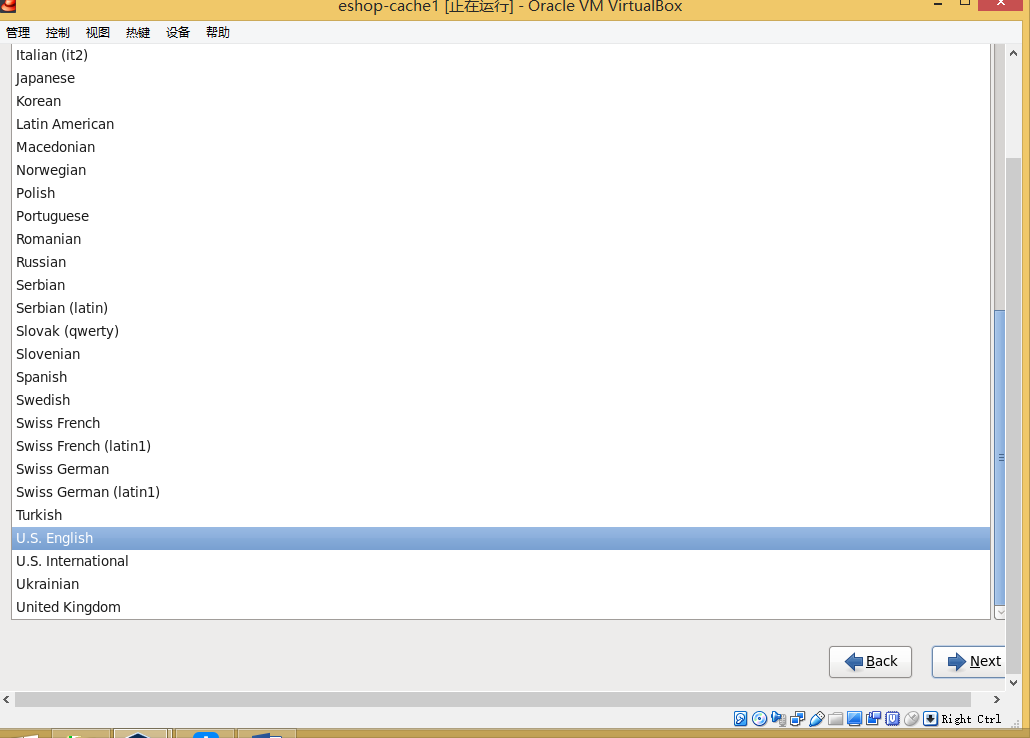


3.选择跳过

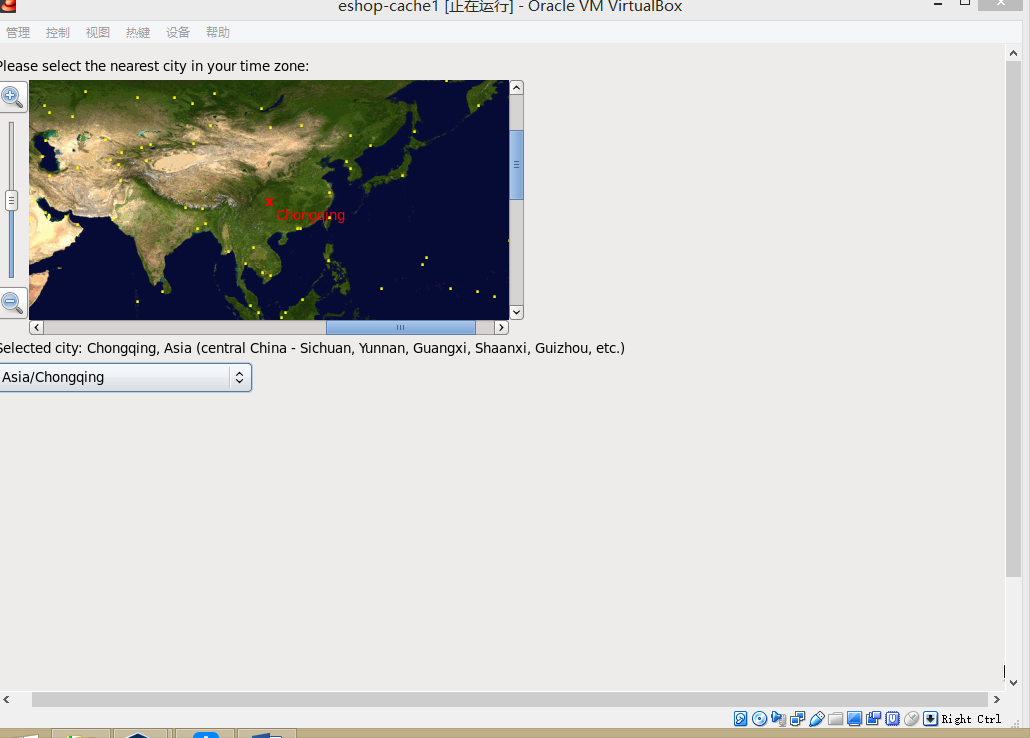


4.选择系统语言

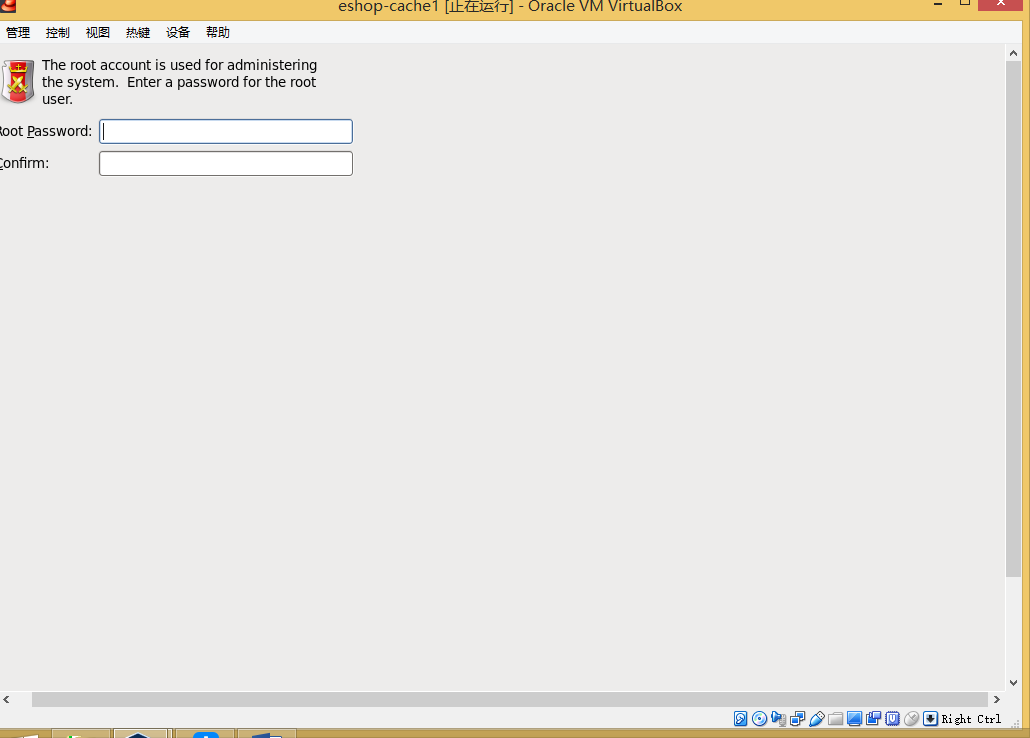
5.选择键盘



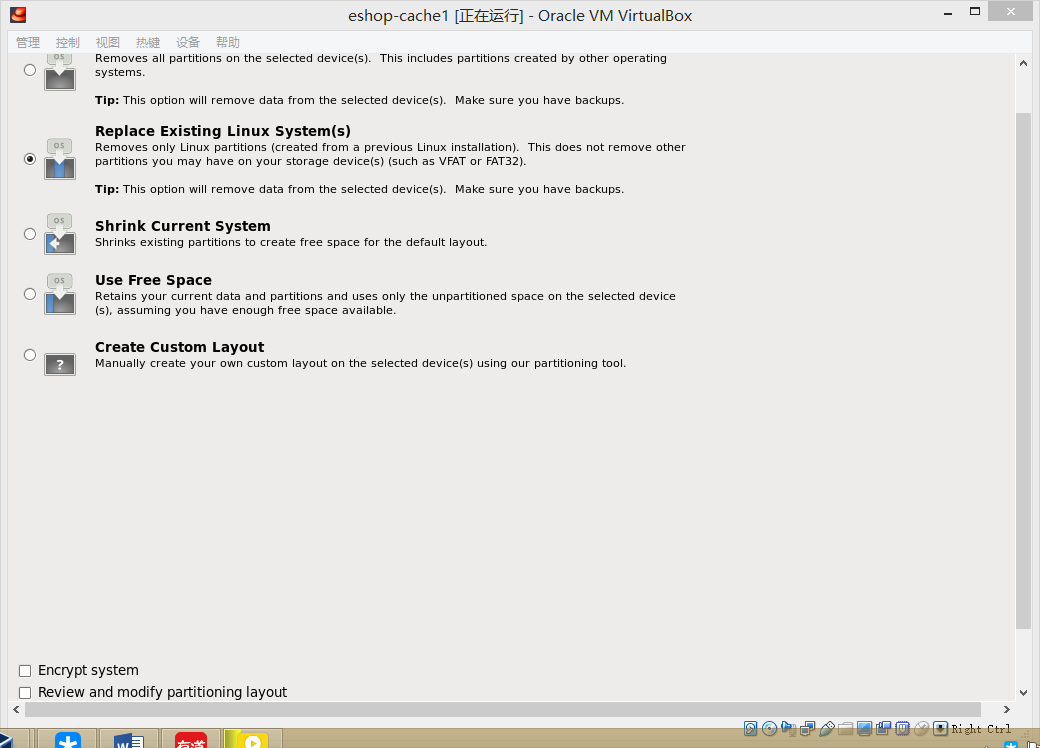
6.选择时区



7.设置root 密码:liu123



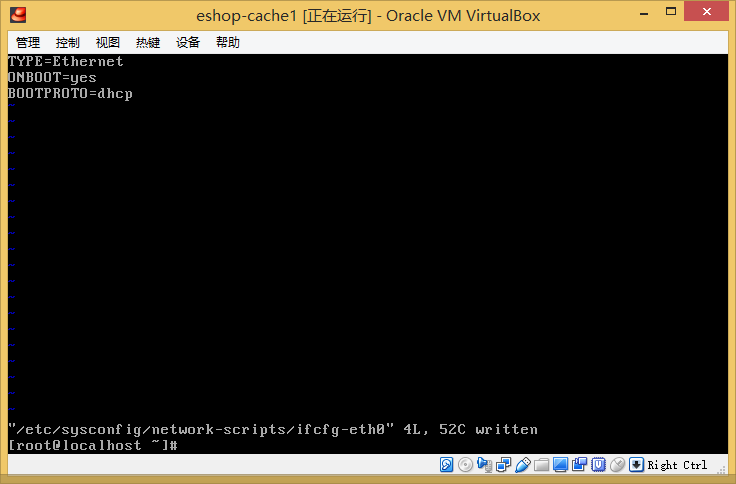
8.直接下一步，进行安装，安装完成后选择reboot进行重启。



### 4.配置CentOS网络环境

1.删掉多余配置，将ONBOOT修改yes

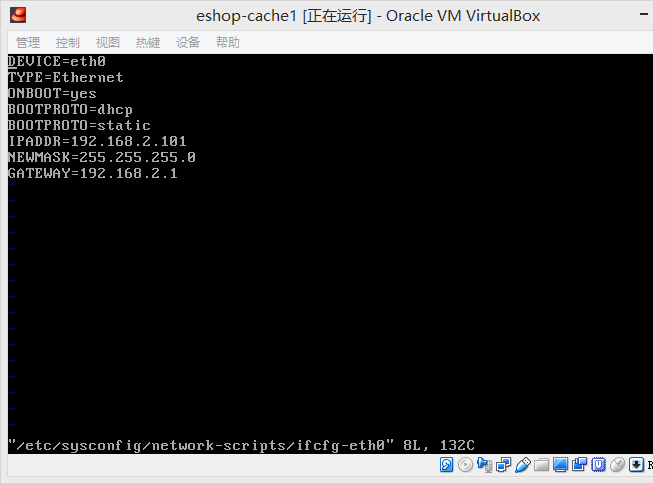
vi /etc/sysconfig/network-scripts/ifcfg-eth0



2.重启网卡：

service network restart

3.设置静态IP,在重启网卡



### 5.配置host

vi /etc/hosts

配置本机的hostname到ip地址的映射

### 6.关闭防火墙

service iptables stop

service ip6tables stop

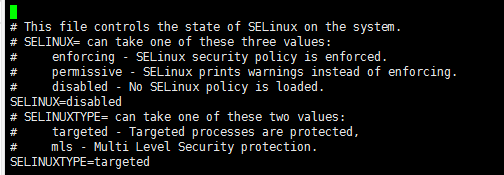
chkconfig iptables off

chkconfig ip6tables off

vi /etc/selinux/config

SELINUX=disabled

修改成disabled



### 7配置yum

yum clean all

yum makecache

yum install - y wget

### 8.安装JDK 和perl

（1）安装JDK

1、将jdk-7u60-linux-i586.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-7u65-linux-i586.rpm

3、配置jdk相关的环境变量

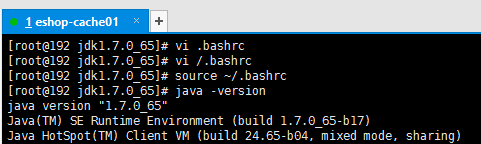
vi .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java –version



（2）安装Perl

yum install -y gcc

wget http://www.cpan.org/src/5.0/perl-5.16.1.tar.gz

tar -xzf perl-5.16.1.tar.gz

cd perl-5.16.1

./Configure -des -Dprefix=/usr/local/perl

make && make test && make install

perl -v

为什么要装perl？整个大型电商网站的详情页系统，复杂。java+nginx+lua，需要perl。

perl，是一个基础的编程语言的安装，tomcat，跑java web应用

# 2.安装单机版Redis +生产环境启动方式

### 1.先安装tcl

1.下载tcl包至/usr/local：wget <http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz>

2.解压缩 tar -xzvf tcl8.6.1-src.tar.gz

3.进入unix目录运行: cd /usr/local/tcl8.6.1/unix/

4.运行 ./configure

5.安装：make && make install

### 2.安装redis

1.将redis 压缩包上传至：/usr/local

2.解压redis安装包 tar -zxvf redis-3.2.8.tar.gz

3.进入redis解压缩目录 cd redis-3.2.8

4.运行安装命令：make && make test && make install

安装make test 常见问题解决方案

<https://blog.csdn.net/for_tech/article/details/51880647>

<https://blog.csdn.net/duyuanhai/article/details/53115955>

### 3.配置生产环境启动方式

1. redis utils目录下，有个redis\_init\_script脚本拷贝到linux的/etc/init.d目录中。

2.将redis\_init\_script重命名为redis\_6379，6379是我们希望这个redis实例监听的端口号

3.修改redis\_6379脚本的第6行的REDISPORT，设置为相同的端口号（默认就是6379）

4. 创建两个目录：/etc/redis（存放redis的配置文件），/var/redis/6379（存放redis的持久化文件）

5. 修改redis配置文件（默认在根目录下，redis.conf），拷贝到/etc/redis目录中，修改名称为6379.conf

6.修改redis.conf中的部分配置为生产环境

daemonize yes 让redis以daemon进程运行

pidfile /var/run/redis\_6379.pid 设置redis的pid文件位置

port 6379 设置redis的监听端口号

dir /var/redis/6379 设置持久化文件的存储位置

7. 启动redis，执行cd /etc/init.d, chmod 777 redis\_6379，./redis\_6379 start

8. 确认redis进程是否启动，ps -ef | grep redis

9.让redis跟随系统启动自动启动在redis\_6379脚本中，最上面，加入两行注释

# chkconfig: 2345 90 10

# description: Redis is a persistent key-value database

运行命令：chkconfig redis\_6379 on

10. redis-cli PING，ping redis的端口，看是否正常

11. redis-cli，进入交互式命令行使用命令测试redis安装是否正常。

SET k1 v1

GET k1

# 3. redis持久化的意义，在于故障恢复

比如你部署了一个redis，作为cache缓存，当然也可以保存一些较为重要的数据如果没有持久化的话，redis遇到灾难性故障的时候，就会丢失所有的数据。

如果通过持久化将数据搞一份儿在磁盘上去，然后定期比如说同步和备份到一些云存储服务上去，那么就可以保证数据不丢失全部，还是可以恢复一部分数据回来的。

# 4.RDB 和AOF两种持久化机制的工作原理。

## 1、RDB和AOF两种持久化机制的介绍

RDB持久化机制，对redis中的数据执行周期性的持久化

AOF机制对每条写入命令作为日志，以append-only的模式写入一个日志文件中，在redis重启的时候，可以通过回放AOF日志中的写入指令来重新构建整个数据集

如果我们想要redis仅仅作为纯内存的缓存来用，那么可以禁止RDB和AOF所有的持久化机制

通过RDB或AOF，都可以将redis内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云，云服务

如果redis挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动redis，redis就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务

如果同时使用RDB和AOF两种持久化机制，那么在redis重启的时候，会使用AOF来重新构建数据，因为AOF中的数据更加完整

## 2、RDB持久化机制的优点

（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说Amazon的S3云服务上去，在国内可以是阿里云的ODPS分布式存储上，以预定好的备份策略来定期备份redis中的数据

RDB可以做冷备份，生成多个文件，每个文件都代表了某一时刻的完整的数据快照。

AOF也可以做冷备，只有一个文件，但是可以自己控制写程序没隔一定时间，去copy一份这个文件出来。

RDB冷备优势：有redis去控制固定时长生成快照文件的事情，比较方便；

AOF,还需要自己写一些脚步。

（2）RDB对redis对外提供的读写服务，影响非常小，可以让redis保持高性能，因为redis主进程只需要fork一个子进程，让子进程执行磁盘IO操作来进行RDB持久化即可

RDB:每次写，都是直接写redis内存，只是在一定的时候，才会将数据写入磁盘。

AOF:每次都要写文件，虽然可以快速的写入os cache中，但是还是有一定的时间开销，速度肯定比RDB慢。

（3）相对于AOF持久化机制来说，直接基于RDB数据文件来重启和恢复redis进程，更加快速

RDB:就是一份数据文件，恢复的时候，直接加载到内存中即可。

AOF:存放的指令日志，做恢复的时候，其实时要回放和执行所有指令日志来恢复出来内存中的所有数据的。

综合上述有点RDB文件更适合做为冷备份。

## 3、RDB持久化机制的缺点

（1）如果想要在redis故障时，尽可能少的丢失数据，那么RDB没有AOF好。一般来说，RDB数据快照文件，都是每隔5分钟，或者更长时间生成一次，这个时候就得接受一旦redis进程宕机，那么会丢失最近5分钟的数据。

这个问题：也是RDB最大的缺点，就是不适合做第一优先的恢复方案；如果你依赖RDB做第一优先恢复方案，会导致数据丢失的比较多。

（2）RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒。

一般不要让RDB的间隔太长，否则每次生成的RDB文件太大了，对redis本身的可能是会有影响的。

## 4、AOF持久化机制的优点

（1）AOF可以更好的保护数据不丢失，一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据

（2）AOF日志文件以append-only模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复

（3）AOF日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在rewrite log的时候，会对其中的指导进行压缩，创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后的日志文件ready的时候，再交换新老日志文件即可。

（4）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

## 5、AOF持久化机制的缺点

（1）对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

（2）AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的

（3）以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

（4）数据恢复的时候没有RDB快，做冷备的时候不太方便，需要自己手动写复杂备份程序。

## 6、RDB和AOF到底该如何选择

（1）不要仅仅使用RDB，因为那样会导致你丢失很多数据

（2）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

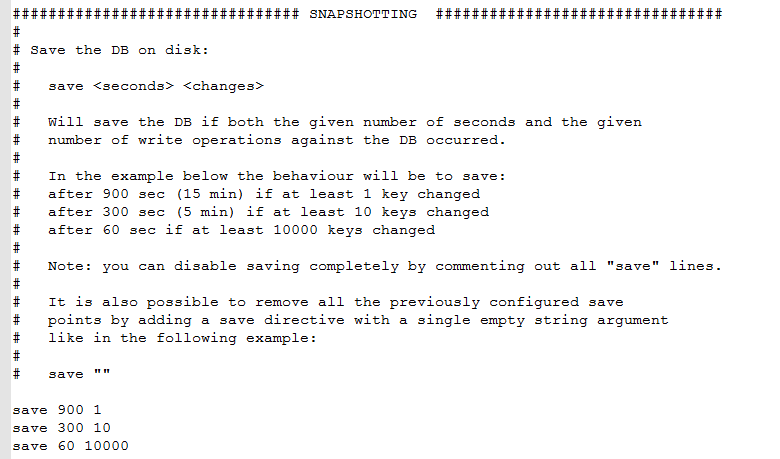
（3）综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择; 用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复。

# 5.RDB持久化配置，以及数据恢复

## 1.如何配置RDB持久化文件

1.找到redis配置文件redis.cof 文件 ：cd /rtc/redis

2.在save 查询配置生成RDB文件的方案。



* 每隔60s，如果有超过10000个key发生了变更，那么就生成一个新的dump.rdb文件，就是当前redis内存中完整的数据快照，这个操作也被称之为snapshotting，快照。
* 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，每到一个检查点，就会去check一下，是否有指定的key数量发生了变更，如果有，就生成一个新的dump.rdb文件。

## 2.RDB持久化机制的工作流程

（1）redis根据配置自己尝试去生成rdb快照文件

（2）fork一个子进程出来

（3）子进程尝试将数据dump到临时的rdb快照文件中

（4）完成rdb快照文件的生成之后，就替换之前的旧的快照文件

dump.rdb，每次生成一个新的快照，都会覆盖之前的老快照。

## 3. 基于RDB持久化机制的数据恢复实验

（1）在redis中保存几条数据，立即停掉redis进程，然后重启redis，看看刚才插入的数据还在不在

数据还在，为什么？通过redis-cli SHUTDOWN这种方式去停掉redis，其实是一种安全退出的模式，redis在退出的时候会将内存中的数据立即生成一份完整的rdb快照。

（2）在redis中再保存几条新的数据，用kill -9粗暴杀死redis进程，模拟redis故障异常退出，导致内存数据丢失的场景

这次就发现，redis进程异常被杀掉，数据没有进dump文件，几条最新的数据就丢失了。

（3）手动设置一个save检查点，save 5 1



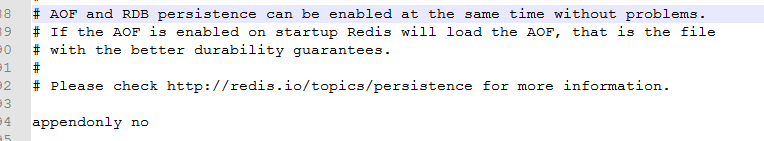
（4）写入几条数据，等待5秒钟，会发现自动进行了一次dump rdb快照，在dump.rdb中发现了数据

（5）异常停掉redis进程，再重新启动redis，看刚才插入的数据还在

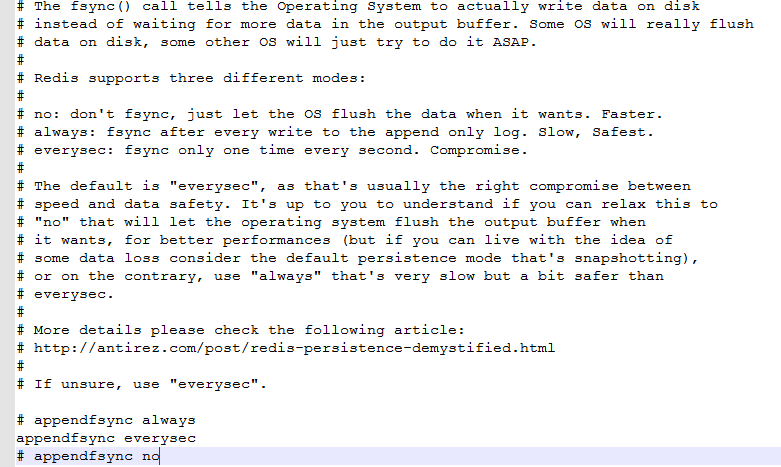
# 6.AOF持久化配置，以及数据恢复

## 1、AOF持久化的配置

* AOF持久化，默认是关闭的，默认是打开RDB持久化。



* appendonly yes，可以打开AOF持久化机制，在生产环境里面，一般来说AOF都是要打开的，除非你说随便丢个几分钟的数据也无所谓。
* 打开AOF持久化机制之后，redis每次接收到一条写命令，就会写入日志文件中，当然是先写入os cache的，然后每隔一定时间再fsync一下。
* 而且即使AOF和RDB都开启了，redis重启的时候，也是优先通过AOF进行数据恢复的，因为aof数据比较完整。
* 可以配置AOF的fsync策略，有三种策略可以选择，一种是每次写入一条数据就执行一次fsync; 一种是每隔一秒执行一次fsync; 一种是不主动执行fsync。



**always:** 每次写入一条数据，立即将这个数据对应的写日志fsync到磁盘上去，性能非常非常差，吞吐量很低; 确保说redis里的数据一条都不丢，那就只能这样了。

mysql -> 内存策略，大量磁盘，QPS到多少，一两k。QPS，每秒钟的请求数量。

redis -> 内存，磁盘持久化，QPS到多少，单机，一般来说，上万QPS没问题。

**everysec:** 每秒将os cache中的数据fsync到磁盘，这个最常用的，生产环境一般都这么配置，性能很高，QPS还是可以上万的。

**no:** 仅仅redis负责将数据写入os cache就撒手不管了，然后后面os自己会时不时有自己的策略将数据刷入磁盘，不可控了。

## 2、AOF持久化的数据恢复实验

（1）先仅仅打开RDB，写入一些数据，然后kill -9杀掉redis进程，接着重启redis，发现数据没了，因为RDB快照还没生成

（2）打开AOF的开关，启用AOF持久化

（3）写入一些数据，观察AOF文件中的日志内容

* 其实你在appendonly.aof文件中，可以看到刚写的日志，它们其实就是先写入os cache的，然后1秒后才fsync到磁盘中，只有fsync到磁盘中了，才是安全的，要不然光是在os cache中，机器只要重启，就什么都没了

（4）kill -9杀掉redis进程，重新启动redis进程，发现数据被恢复回来了，就是从AOF文件中恢复回来的

* redis进程启动的时候，直接就会从appendonly.aof中加载所有的日志，把内存中的数据恢复回来

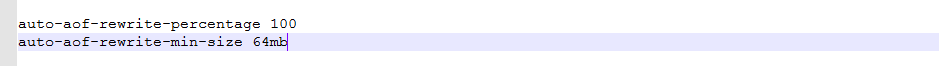
## 3、AOF rewrite

（1）redis中的数据其实有限的，很多数据可能会自动过期，可能会被用户删除，可能会被redis用缓存清除的算法清理掉。redis中的数据会不断淘汰掉旧的，就一部分常用的数据会被自动保留在redis内存中所以可能很多之前的已经被清理掉的数据，对应的写日志还停留在AOF中，AOF日志文件就一个，会不断的膨胀，到很大很大。所以AOF会自动在后台每隔一定时间做rewrite操作。

比如日志里已经存放了针对100w数据的写日志了; redis内存只剩下10万; 基于内存中当前的10万数据构建一套最新的日志，到AOF中; 覆盖之前的老日志; 确保AOF日志文件不会过大，保持跟redis内存数据量一致。

（2）redis 2.4之前，还需要手动，开发一些脚本，crontab，通过BGREWRITEAOF命令去执行AOF rewrite，但是redis 2.4之后，会自动进行rewrite操作。

（3）在redis.conf中，可以配置rewrite策略



auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb

比如说上一次AOF rewrite之后，是128mb然后就会接着128mb继续写AOF的日志，如果发现增长的比例，超过了之前的100%，256mb，就可能会去触发一次rewrite

但是此时还要去跟min-size，64mb去比较，256mb > 64mb，才会去触发rewrite

**rewrite原理**

（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、AOF破损文件的修复

如果redis在append数据到AOF文件时，机器宕机了，可能会导致AOF文件破损

用redis-check-aof --fix命令来修复破损的AOF文件。

## 5、AOF和RDB同时工作

（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。

# 7. 企业级的数据备份方案以及数据恢复

## 1.数据备份方案

（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

小时备份定时脚本

|  |
| --- |
| crontab –e  0 \* \* \* \* sh /usr/local/redis/copy/redis\_rdb\_copy\_hourly.sh |

小时Copy脚本

|  |
| --- |
| 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 |

日备份定时脚本

|  |
| --- |
| crontab -e  0 0 \* \* \* sh /usr/local/redis/copy/redis\_rdb\_copy\_daily.sh |

日copy脚本

|  |
| --- |
| 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 |

## 2. 数据恢复方案

（1）如果是redis进程挂掉，那么重启redis进程即可，直接基于AOF日志文件恢复数据

（2）如果是redis进程所在机器挂掉，那么重启机器后，尝试重启redis进程，尝试直接基于AOF日志文件进行数据恢复。

* AOF没有破损，也是可以直接基于AOF恢复的
* AOF append-only，顺序写入，如果AOF文件破损，那么用redis-check-aof fix

（3）如果redis当前最新的AOF和RDB文件出现了丢失/损坏，那么可以尝试基于该机器上当前的某个最新的RDB数据副本进行数据恢复。

操作方式：

* 停止redis，关闭aof，拷贝rdb备份，重启redis，确认数据恢复，直接在命令行热修改redis配置，打开aof，这个redis就会将内存中的数据对应的日志，写入aof文件中此时aof和rdb两份数据文件的数据就同步了
* redis config set热修改配置参数，可能配置文件中的实际的参数没有被持久化的修改，再次停止redis，手动修改配置文件，打开aof的命令，再次重启redis

（4）如果当前机器上的所有RDB文件全部损坏，那么从远程的云服务上拉取最新的RDB快照回来恢复数据。

（5）如果是发现有重大的数据错误，比如某个小时上线的程序一下子将数据全部污染了，数据全错了，那么可以选择某个更早的时间点，对数据进行恢复。

# 8. redis replication

## 1. 核心机制

（1）redis采用异步方式复制数据到slave节点，不过redis 2.8开始，slave node会周期性地确认自己每次复制的数据量

（2）一个master node是可以配置多个slave node的

（3）slave node也可以连接其他的slave node

（4）slave node做复制的时候，是不会block master node的正常工作的

（5）slave node在做复制的时候，也不会block对自己的查询操作，它会用旧的数据集来提供服务; 但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了

（6）slave node主要用来进行横向扩容，做读写分离，扩容的slave node可以提高读的吞吐量

slave，高可用性，有很大的关系。

## 2、master持久化对于主从架构的安全保障的意义

如果采用了主从架构，那么建议必须开启master node的持久化！

不建议用slave node作为master node的数据热备，因为那样的话，如果你关掉master的持久化，可能在master宕机重启的时候数据是空的，然后可能一经过复制，salve node数据也丢了

master -> RDB和AOF都关闭了 -> 全部在内存中

master宕机，重启，是没有本地数据可以恢复的，然后就会直接认为自己IDE数据是空的

master就会将空的数据集同步到slave上去，所有slave的数据全部清空

100%的数据丢失

master节点，必须要使用持久化机制。

## 3.、主从架构的核心原理

* 当启动一个slave node的时候，它会发送一个PSYNC命令给master node
* 如果这是slave node重新连接master node，那么master node仅仅会复制给slave部分缺少的数据; 否则如果是slave node第一次连接master node，那么会触发一次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。

## 4、主从复制的断点续传

* 从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份
* 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

3、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

4、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。

# 9. 项目中部署redis的读写分离架构（包含节点间认证口令）

## 1.安装从节点

安装生产环境单机版安装方式

在slave node上配置：slaveof 主节点的ip 和端口，即可。



由于默认配置bind 127.0.0.1 -> 本地的开发调试的模式，就只能127.0.0.1本地才能访问到6379的端口。所以将每个redis.conf中的bind 127.0.0.1 -> bind自己的ip地址



在每个节点上都运行一下命令确保端口都打开:

iptables -A INPUT -ptcp --dport 6379 -j ACCEPT

## 2、强制读写分离

* 基于主从复制架构，实现读写分离
* redis slave node只读，默认开启，slave-read-only
* 开启了只读的redis slave node，会拒绝所有的写操作，这样可以强制搭建成读写分离的架构

## 3、集群安全认证

* master上启用安全认证，requirepass
* slave node master连接口令，masterauth

## 4、读写分离架构的测试

先启动主节点，eshop-cache01上的redis实例

再启动从节点，eshop-cache02上的redis实例

使用客户端连接方式：

redis-cli -h ipaddr

查看主从机信息：

info replication

在主上写，在从上读。

## 5.对集群进行压测

redis自己提供的redis-benchmark压测工具，是最快捷最方便的，当然啦，这个工具比较简单，用一些简单的操作和场景去压测。

1、对redis读写分离架构进行压测，单实例写QPS+单实例读QPS

|  |
| --- |
| cd redis-3.2.8/src  ./redis-benchmark -h 192.168.31.187  可选参数  -c <clients> Number of parallel connections (default 50)  -n <requests> Total number of requests (default 100000)  -d <size> Data size of SET/GET value in bytes (default 2) |

# 10. redis哨兵架构

## 1. redis哨兵架构的相关基础知识

### 1、哨兵的介绍

sentinal，中文名是哨兵，哨兵是redis集群架构中非常重要的一个组件，主要功能如下：

（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

哨兵本身也是分布式的，作为一个哨兵集群去运行，互相协同工作

（1）故障转移时，判断一个master node是宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题

（2）即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了

### 2、哨兵的核心知识

（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

### 3、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

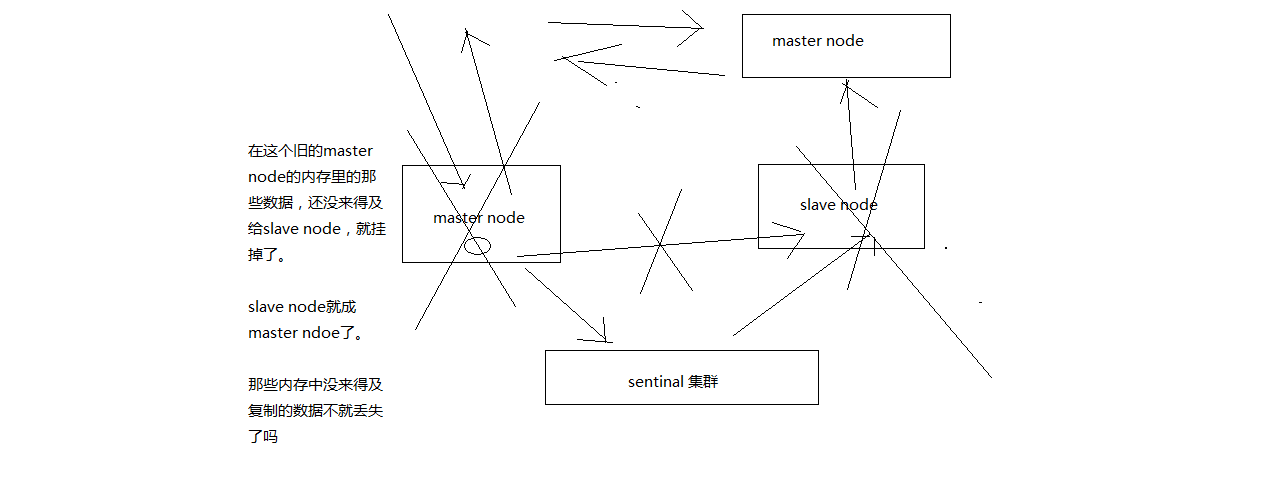
同时3个哨兵的majority是2，所以还剩下的2个哨兵运行着，就可以允许执行故障转移

## 2. 主备切换的数据丢失问题

### 1、两种数据丢失的情况

（1）异步复制导致的数据丢失

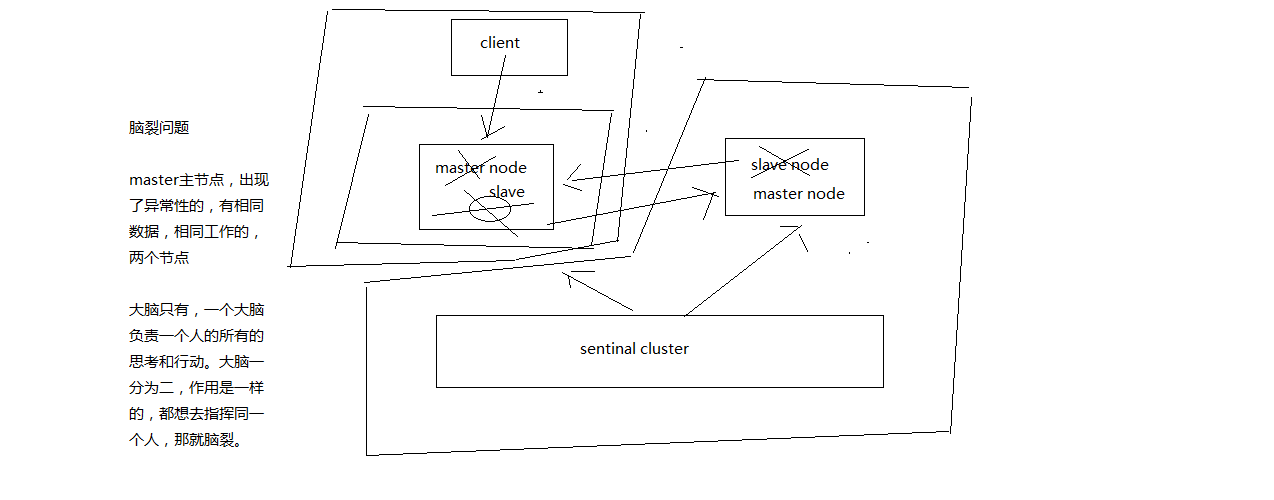
因为master -> slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，此时这些部分数据就丢失了



（2）脑裂导致的数据丢失

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着，此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master这个时候，集群里就会有两个master，也就是所谓的脑裂。

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master的数据可能也丢失了。因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据。



### 2、解决异步复制和脑裂导致的数据丢失

|  |
| --- |
| 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。

## C:\Users\L\AppData\Local\Temp\Rar$DRa0.693\22_redis哨兵主备切换的数据丢失问题：异步复制、集群脑裂\资料\异步复制导致数据丢失如何降低损失.png3. redis哨兵的多个核心底层原理的深入解析

### 1、sdown和odown转换机制

sdown和odown两种失败状态

* sdown是主观宕机，就一个哨兵如果自己觉得一个master宕机了，那么就是主观宕机。
* odown是客观宕机，如果quorum数量的哨兵都觉得一个master宕机了，那么就是客观宕机。
* sdown达成的条件很简单，如果一个哨兵ping一个master，超过了is-master-down-after-milliseconds指定的毫秒数之后，就主观认为master宕机。
* sdown到odown转换的条件很简单，如果一个哨兵在指定时间内，收到了quorum指定数量的其他哨兵也认为那个master是sdown了，那么就认为是odown了，客观认为master宕机。

### 2、哨兵集群的自动发现机制

* 哨兵互相之间的发现，是通过redis的pub/sub系统实现的，每个哨兵都会往\_\_sentinel\_\_:hello这个channel里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在。
* 每隔两秒钟，每个哨兵都会往自己监控的某个master+slaves对应的\_\_sentinel\_\_:hello channel里发送一个消息，内容是自己的host、ip和runid还有对这个master的监控配置。
* 每个哨兵也会去监听自己监控的每个master+slaves对应的\_\_sentinel\_\_:hello channel，然后去感知到同样在监听这个master+slaves的其他哨兵的存在。
* 每个哨兵还会跟其他哨兵交换对master的监控配置，互相进行监控配置的同步。

### 3、slave配置的自动纠正

哨兵会负责自动纠正slave的一些配置，比如slave如果要成为潜在的master候选人，哨兵会确保slave在复制现有master的数据; 如果slave连接到了一个错误的master上，比如故障转移之后，那么哨兵会确保它们连接到正确的master上。

### 4、slave->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

(down-after-milliseconds \* 10) + 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。

### 6、configuration epoch

* 哨兵会对一套redis master+slave进行监控，有相应的监控的配置
* 执行切换的那个哨兵，会从要切换到的新master（salve->master）那里得到一个configuration epoch，这就是一个version号，每次切换的version号都必须是唯一的
* 如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待failover-timeout时间，然后接替继续执行切换，此时会重新获取一个新的configuration epoch，作为新的version号

### 7、configuraiton传播

* 哨兵完成切换之后，会在自己本地更新生成最新的master配置，然后同步给其他的哨兵，就是通过之前说的pub/sub消息机制
* 这里之前的version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的
* 其他的哨兵都是根据版本号的大小来更新自己的master配置的

# 11.节点方式部署哨兵集群

## 1、哨兵的配置文件

最小的配置：

每一个哨兵都可以去监控多个maser-slaves的主从架构因为可能你的公司里，为不同的项目，部署了多个master-slaves的redis主从集群相同的一套哨兵集群，就可以去监控不同的多个redis主从集群你自己给每个redis主从集群分配一个逻辑的名称。

|  |
| --- |
| sentinel monitor mymaster 127.0.0.1 6379 2  sentinel down-after-milliseconds mymaster 60000  sentinel failover-timeout mymaster 180000  sentinel parallel-syncs mymaster 1  sentinel monitor resque 192.168.1.3 6380 4  sentinel down-after-milliseconds resque 10000  sentinel failover-timeout resque 180000  sentinel parallel-syncs resque 5 |

sentinel monitor mymaster 127.0.0.1 6379

类似这种配置，来指定对一个master的监控，给监控的master指定的一个名称。

sentinel down-after-milliseconds mymaster 60000

sentinel failover-timeout mymaster 180000

sentinel parallel-syncs mymaster 1

上面的三个配置，都是针对某个监控的master配置的，给其指定上面分配的名称即可

上面这段配置，就监控了两个master node这是最小的哨兵配置，如果发生了master-slave故障转移，或者新的哨兵进程加入哨兵集群，那么哨兵会自动更新自己的配置文件。

|  |
| --- |
| sentinel monitor master-group-name hostname port quorum  quorum的解释如下：  （1）至少多少个哨兵要一致同意，master进程挂掉了，或者slave进程挂掉了，或者要启动一个故障转移操作  （2）quorum是用来识别故障的，真正执行故障转移的时候，还是要在哨兵集群执行选举，选举一个哨兵进程出来执行故障转移操作  （3）假设有5个哨兵，quorum设置了2，那么如果5个哨兵中的2个都认为master挂掉了; 2个哨兵中的一个就会做一个选举，选举一个哨兵出来，执行故障转移; 如果5个哨兵中有3个哨兵都是运行的，那么故障转移就会被允许执行  down-after-milliseconds：  超过多少毫秒跟一个redis实例断了连接，哨兵就可能认为这个redis实例挂了  parallel-syncs：  新的master别切换之后，同时有多少个slave被切换到去连接新master，重新做同步，数字越低，花费的时间越多。假设你的redis是1个master，4个slave然后master宕机了，4个slave中有1个切换成了master，剩下3个slave就要挂到新的master上面去。这个时候，如果parallel-syncs是1，那么3个slave，一个一个地挂接到新的master上面去，1个挂接完，而且从新的master sync完数据之后，再挂接下一个如果parallel-syncs是3，那么一次性就会把所有slave挂接到新的master上去  failover-timeout：  执行故障转移的timeout超时时长。 |

## 2、正式的配置

1.创建哨兵配置文件目录

mkdir /etc/sentinel

2.创建哨兵工作目录

mkdir -p /var/sentinal/5000

3.配置只单个哨兵配置文件

/etc/sentinel/5000.conf

port 5000

bind 192.168.0.113

dir /var/sentinal/5000

sentinel monitor mymaster 192.168.31.187 6379 2

sentinel down-after-milliseconds mymaster 30000

sentinel failover-timeout mymaster 60000

sentinel parallel-syncs mymaster 1

## 3、启动哨兵进程

在eshop-cache01、eshop-cache02、eshop-cache03三台机器上，分别启动三个哨兵进程，组成一个集群，观察一下日志的输出

redis-sentinel /etc/sentinal/5000.conf

redis-server /etc/sentinal/5000.conf --sentinel

日志里会显示出来，每个哨兵都能去监控到对应的redis master，并能够自动发现对应的slave

哨兵之间，互相会自动进行发现，用的就是之前说的pub/sub，消息发布和订阅channel消息系统和机制

## 4、检查哨兵状态

redis-cli -h 192.168.31.187 -p 5000

sentinel master mymaster

SENTINEL slaves mymaster

SENTINEL sentinels mymaster

SENTINEL get-master-addr-by-name mymaster

# 12. redis集群的容灾演练

## 1、哨兵节点的增加和删除

增加sentinal，会自动发现。

删除sentinal的步骤

（1）停止sentinal进程

（2）SENTINEL RESET \*，在所有sentinal上执行，清理所有的master状态

（3）SENTINEL MASTER mastername，在所有sentinal上执行，查看所有sentinal对数量是否达成了一致。

## 2、slave的永久下线

让master摘除某个已经下线的slave：SENTINEL RESET mastername，在所有的哨兵上面执行

## 3、slave切换为Master的优先级

slave->master选举优先级：slave-priority，值越小优先级越高。

## 4、基于哨兵集群架构下的安全认证

每个slave都有可能切换成master，所以每个实例都要配置两个指令

master上启用安全认证，requirepass

master连接口令，masterauth

sentinal，sentinel auth-pass <master-group-name> <pass>

## 5、容灾演练

通过哨兵看一下当前的master：SENTINEL get-master-addr-by-name mymaster

把master节点kill -9掉，pid文件也删除掉

查看sentinal的日志，是否出现+sdown字样，识别出了master的宕机问题; 然后出现+odown字样，就是指定的quorum哨兵数量，都认为master宕机了

（1）三个哨兵进程都认为master是sdown了

（2）超过quorum指定的哨兵进程都认为sdown之后，就变为odown

（3）哨兵1是被选举为要执行后续的主备切换的那个哨兵

（4）哨兵1去新的master（slave）获取了一个新的config version

（5）尝试执行failover

（6）投票选举出一个slave区切换成master，每隔哨兵都会执行一次投票

（7）让salve，slaveof noone，不让它去做任何节点的slave了; 把slave提拔成master; 旧的master认为不再是master了

（8）哨兵就自动认为之前的187:6379变成了slave了，19:6379变成了master了

（9）哨兵去探查了一下187:6379这个salve的状态，认为它sdown了

所有哨兵选举出了一个，来执行主备切换操作

如果哨兵的majority都存活着，那么就会执行主备切换操作

再通过哨兵看一下master：SENTINEL get-master-addr-by-name mymaster

尝试连接一下新的master

故障恢复，再将旧的master重新启动，查看是否被哨兵自动切换成slave节点

（1）手动杀掉master

（2）哨兵能否执行主备切换，将slave切换为master

（3）哨兵完成主备切换后，新的master能否使用

（4）故障恢复，将旧的master重新启动

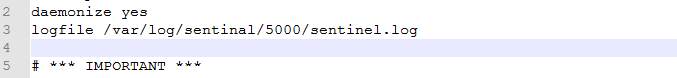
（5）哨兵能否自动将旧的master变为slave，挂接到新的master上面去，而且也是可以使用的

## 6、哨兵的生产环境部署

mkdir -p /var/log/sentinel/5000

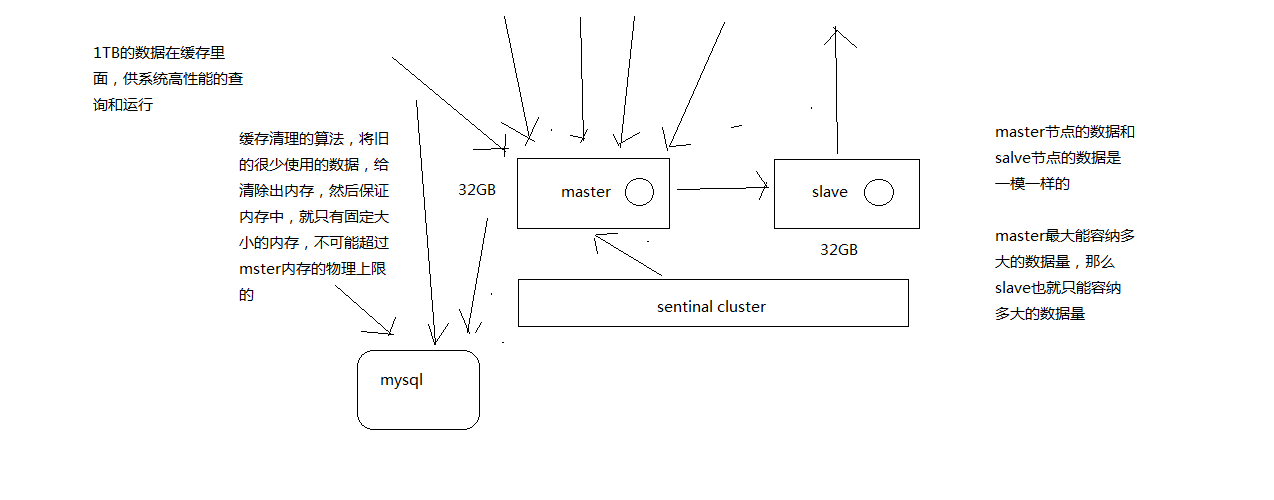
daemonize yes

logfile /var/log/sentinel/5000/sentinel.log

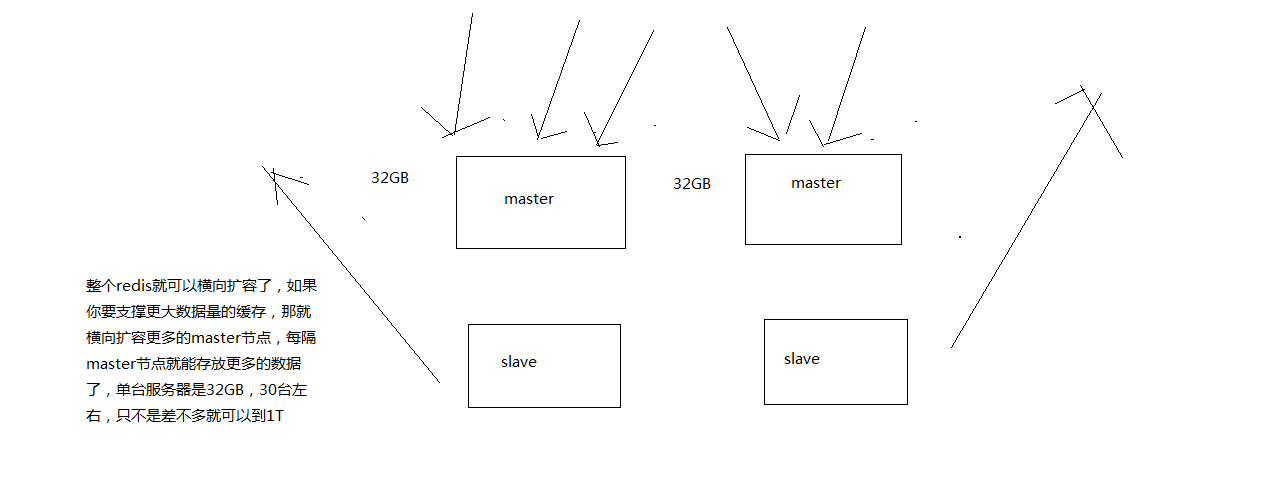


# 13. redis cluster 简单介绍

## 1、单机redis在海量数据面前的瓶颈



## 2、怎么才能够突破单机瓶颈，让redis支撑海量数据？



## 3、redis的集群架构

redis cluster：

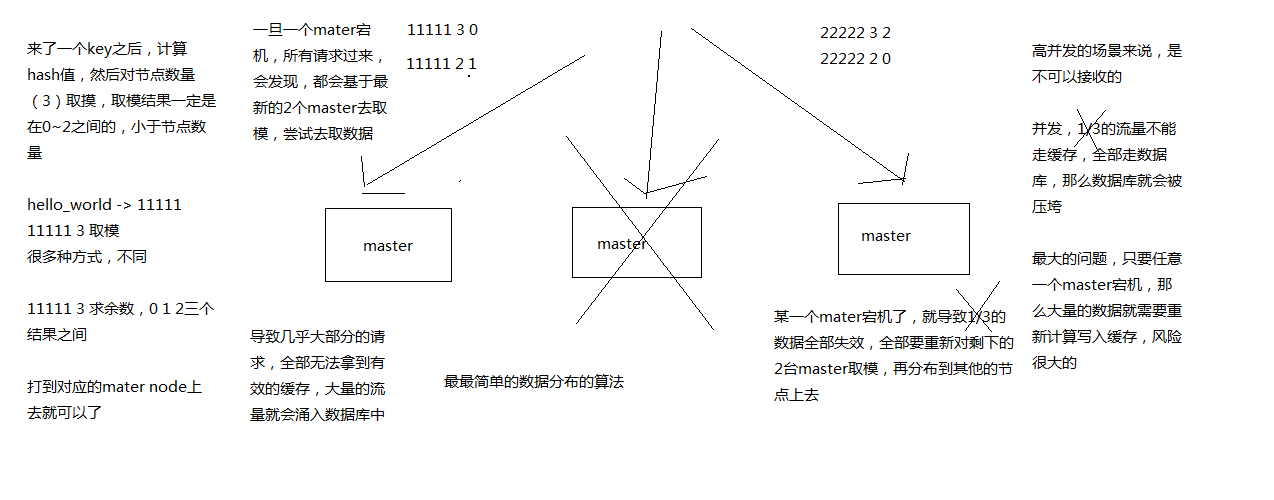
* + 支撑N个redis master node，每个master node都可以挂载多个slave node。
  + 读写分离的架构，对于每个master来说，写就写到master，然后读就从mater对应的slave去读。
  + 高可用，因为每个master都有salve节点，那么如果mater挂掉，redis cluster这套机制，就会自动将某个slave切换成master。
  + redis cluster（多master + 读写分离 + 高可用） 我们只要基于redis cluster去搭建redis集群即可，不需要手工去搭建replication复制+主从架构+读写分离+哨兵集群+高可用。

## 4、redis cluster vs. replication + sentinel

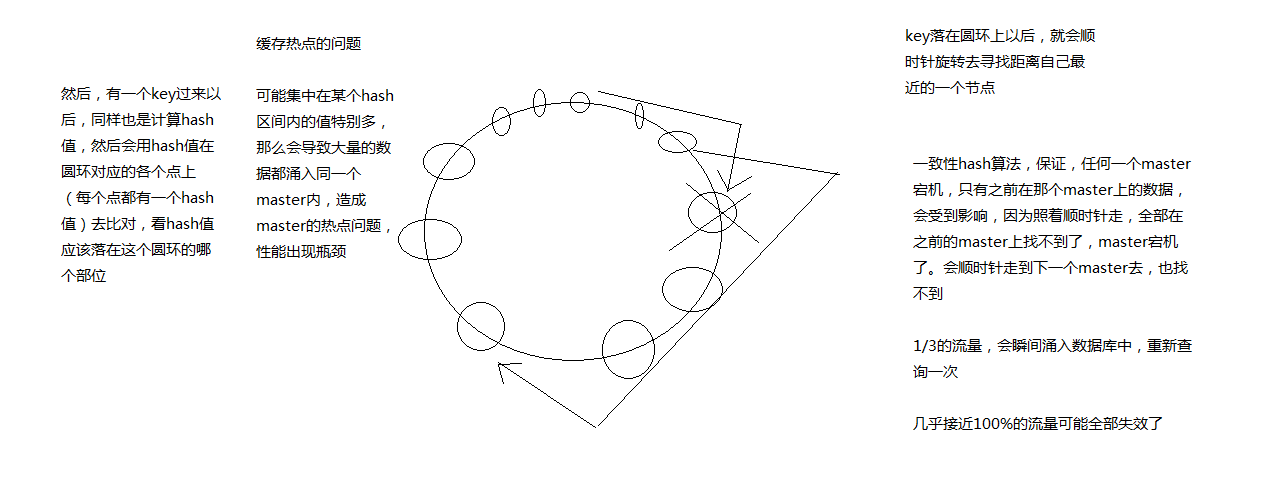
* + 如果你的数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个G，单机足够了。
  + replication，一个mater，多个slave，要几个slave跟你的要求的读吞吐量有关系，然后自己搭建一个sentinal集群，去保证redis主从架构的高可用性，就可以了。
  + redis cluster，主要是针对海量数据+高并发+高可用的场景，海量数据，如果你的数据量很大，那么建议就用redis cluster。

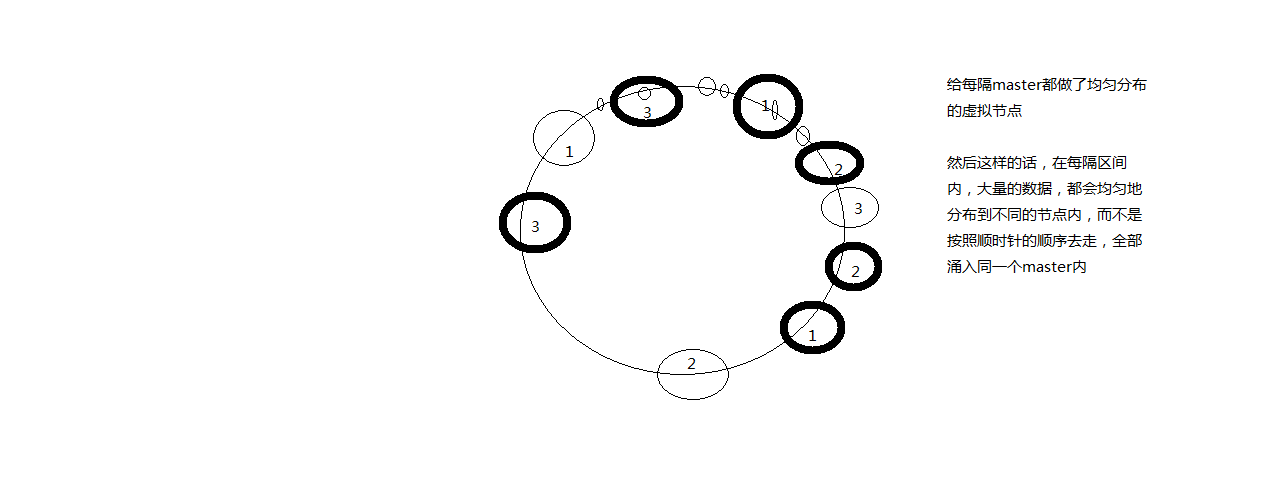
# 14. 分布式数据存储的核心算法

## 1. hash算法



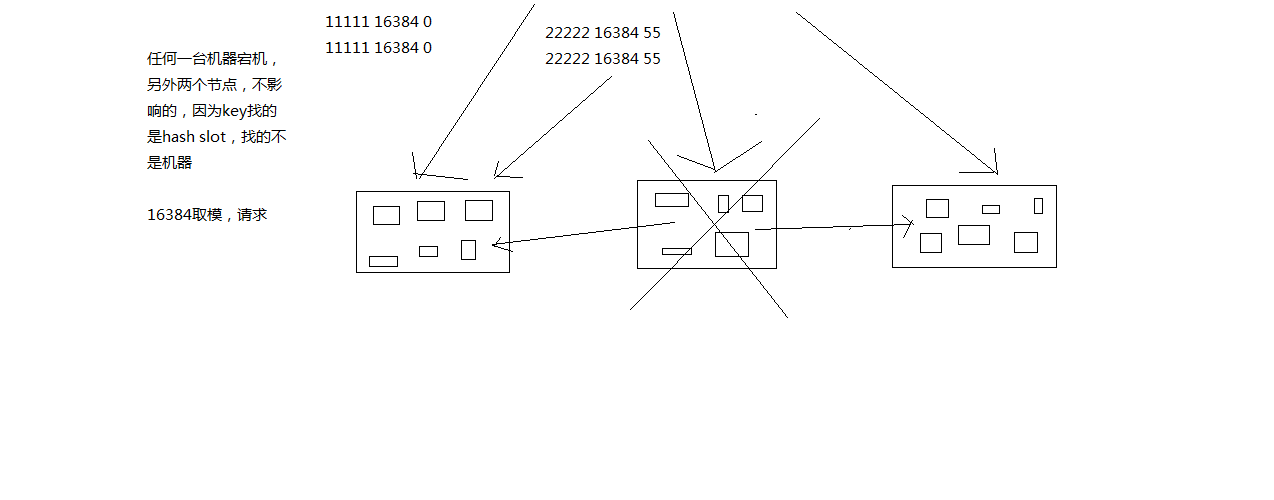
## 2、一致性hash算法（自动缓存迁移）+虚拟节点（自动负载均衡）





## 3、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。



# 15.配置redis cluster集群

## 1、redis cluster的重要配置

|  |
| --- |
| cluster-enabled <yes/no>  cluster-config-file <filename>：这是指定一个文件，供cluster模式下的redis实例将集群状态保存在那里，包括集群中其他机器的信息，比如节点的上线和下限，故障转移，不是我们去维护的，给它指定一个文件，让redis自己去维护的。  cluster-node-timeout <milliseconds>：节点存活超时时长，超过一定时长，认为节点宕机，master宕机的话就会触发主备切换，slave宕机就不会提供服务。 |

## 2、 编写配置文件

redis cluster集群，要求至少3个master，去组成一个高可用，健壮的分布式的集群，每个master都建议至少给一个slave，3个master，3个slave，最少的要求。正式环境下，建议都是说在6台机器上去搭建，至少3台机器保证，每个master都跟自己的slave不在同一台机器上，如果是6台自然更好，一个master+一个slave就死了。3台机器去搭建6个redis实例的redis cluster。

|  |
| --- |
| 1.分别在每台机子上创建文件夹  mkdir -p /etc/redis-cluster  mkdir -p /var/log/redis  mkdir -p /var/redis/7001  2.配置6台redis.conf 文件  port 7001  cluster-enabled yes  cluster-config-file /etc/redis-cluster/node-7001.conf  cluster-node-timeout 15000  daemonize yes  pidfile /var/run/redis\_7001.pid  dir /var/redis/7001  logfile /var/log/redis/7001.log  bind 192.168.31.187  appendonly yes  至少要用3个master节点启动，每个master加一个slave节点，先选择6个节点，启动6个实例将上面的配置文件，在/etc/redis下放6个，分别为: 7001.conf，7002.conf，7003.conf，7004.conf，7005.conf，7006.conf  3.准备生产环境的启动脚本  在/etc/init.d下，放6个启动脚本，分别为: redis\_7001, redis\_7002, redis\_7003, redis\_7004, redis\_7005, redis\_7006  每个启动脚本内，都修改对应的端口号    4.分别在3台机器上，启动6个redis实例  5.安装redis-cluster 集群环境，只需在一台机上安装即可  yum install -y ruby  yum install -y rubygems  gem install redis  cp /usr/local/redis-3.2.8/src/redis-trib.rb /usr/local/bin  6.运行集群命令  redis-trib.rb create --replicas 1 192.168.28.111:7001 192.168.28.111:7002 192.168.28.112:7003 192.168.28.112:7004 192.168.28.113:7005 192.168.28.113:7006  --replicas: 每个master有几个slave  6台机器，3个master，3个slave，尽量自己让master和slave不在一台机器上  运行命令后出现以下的集群方案：    输入“yes”创建集群。  7.查看集群信息  redis-trib.rb check 192.168.28.111:7001 |

## 3. 读写分离+高可用+多master

* 读写分离：每个master都有一个slave
* 高可用：master宕机，slave自动被切换过去
* 多master：横向扩容支持更大数据量

# 16.redis cluster多master写入、读写分离、高可用性

## 1. 多master写入

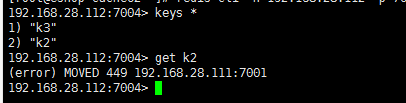
什么叫做多master的写入，就是每条数据只能存在于一个master上，不同的master负责存储不同的数据，分布式的数据存储。100w条数据，5个master，每个master就负责存储20w条数据，分布式数据存储。



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。

## 2. 读写分离



* 在这个redis cluster中，如果你要在slave读取数据，那么需要带上readonly指令，get mykey1
* redis-cli -c启动，就会自动进行各种底层的重定向的操作

redis cluster的读写分离的时候，会发现有一定的限制性，默认情况下，redis cluster的核心的理念，主要是用slave做高可用的，每个master挂一两个slave，主要是做数据的热备，还有master故障时的主备切换，实现高可用的。

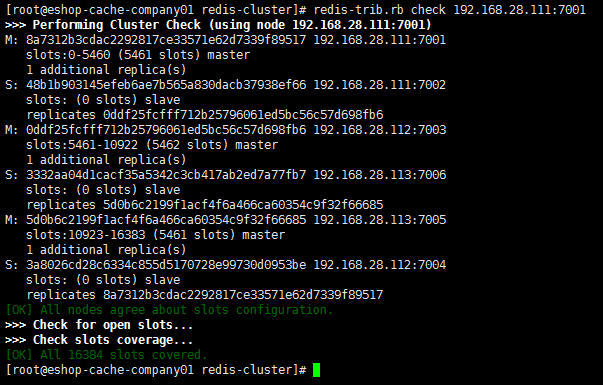
redis cluster默认是不支持slave节点读或者写的，跟我们手动基于replication搭建的主从架构不一样的。

slave node，readonly，get，这个时候才能在slave node进行读取。

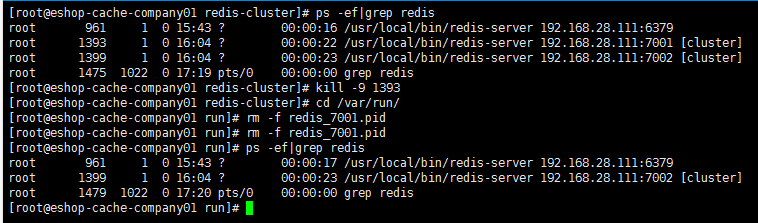
读写分离，是为了什么，主要是因为要建立一主多从的架构，才能横向任意扩展slave node去支撑更大的读吞吐量。redis cluster的架构下，实际上本身master就是可以任意扩展的，你如果要支撑更大的读吞吐量，或者写吞吐量，或者数据量，都可以直接对master进行横向扩展就可以了。也可以实现支撑更高的读吞吐的效果。

## 3. 高可用性

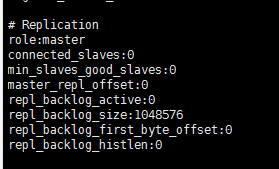
1.先用命令查看集群信息： redis-trib.rb check 192.168.28.111:7001。



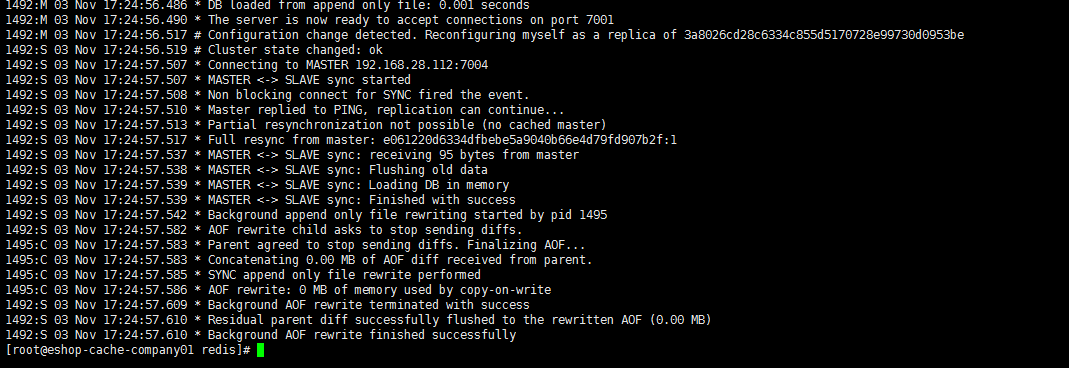
2.强制将7001master 停掉。



3.在使用redis-trib.rb check 192.168.28.111:7004 查看集群信息，7002是否切换成了master.

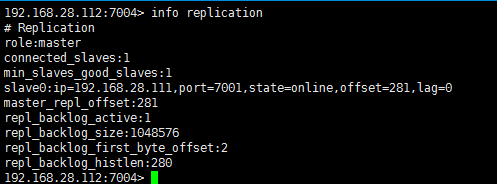


4.重新启动7001，查看是否自动挂到7004节点，变为slave。



通过启动日志发现，7001原本的master切换成了slave 连接到7004上去了。

通过info Replication 命令查看主从关系。



# 17.redis cluster通过master水平扩容来支撑更高的读写吞吐+海量数据

## 1、加入新master

## 2、reshard一些数据过去

## 3、添加node作为slave

## 4、删除node

## 5.、冗余slave故障自动迁移迁移

# 18.redis cluster的核心原理分析

## 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，所有的元数据的跟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。

### （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端口每个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。

## 4、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

1.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。

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

3.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。

4.如果JedisCluter 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。

5.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

重定向如果hash slot正在迁移，那么会返回ask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。

# 19. 数据库双写不一致问题

## 1、最初级的缓存不一致问题以及解决方案

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

解决思路

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

## 2、比较复杂的数据不一致问题分析

问题：数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改

一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中数据变更的程序完成了数据库的修改

解决思路：

数据库与缓存更新与读取操作进行异步串行化

更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个jvm内部的队列中

读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm内部的队列中

一个队列对应一个工作线程每个工作线程串行拿到对应的操作，然后一条一条的执行

这样的话，一个数据变更的操作，先执行删除缓存，然后再去更新数据库，但是还没完成更新此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。

这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中

如果查询请求还在等待时间范围内，不断轮询发现缓存可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。

实现流程：

1.在应用启动的时候初始化请求队列的工作线程。

2、两种请求对象封装

3、请求异步执行Service封装

4、请求处理线程封装

5、两种请求Controller接口封装

6、读请求去重优化

7、空数据读请求过滤优化

1.考虑线程的并发问题，查询数据去重已放在线程中。