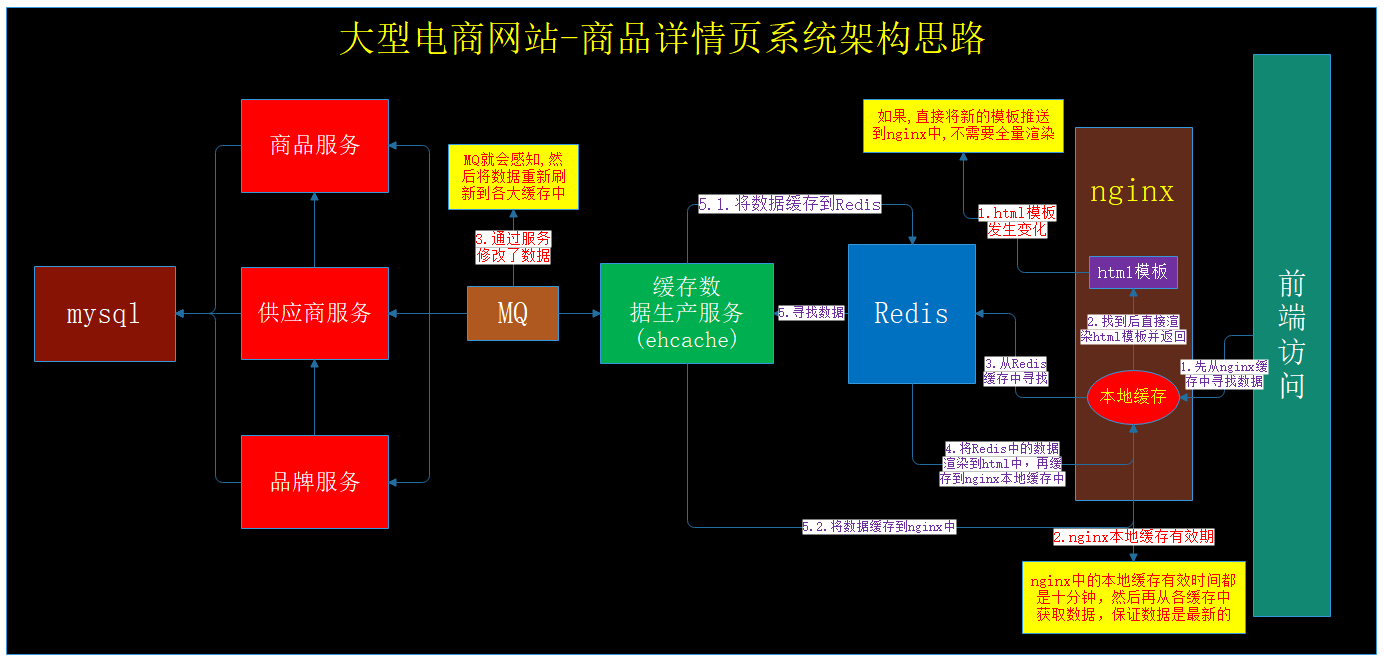
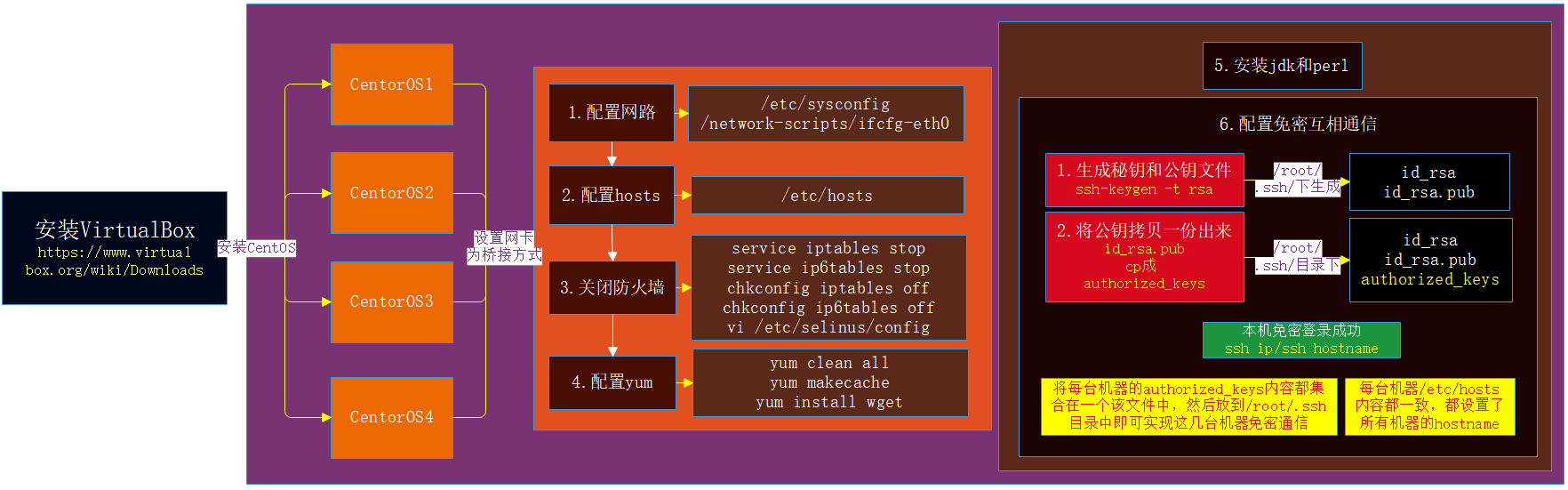
# 大型电商网站-异步多级缓存构建-nginx数据本地化动态渲染的架构



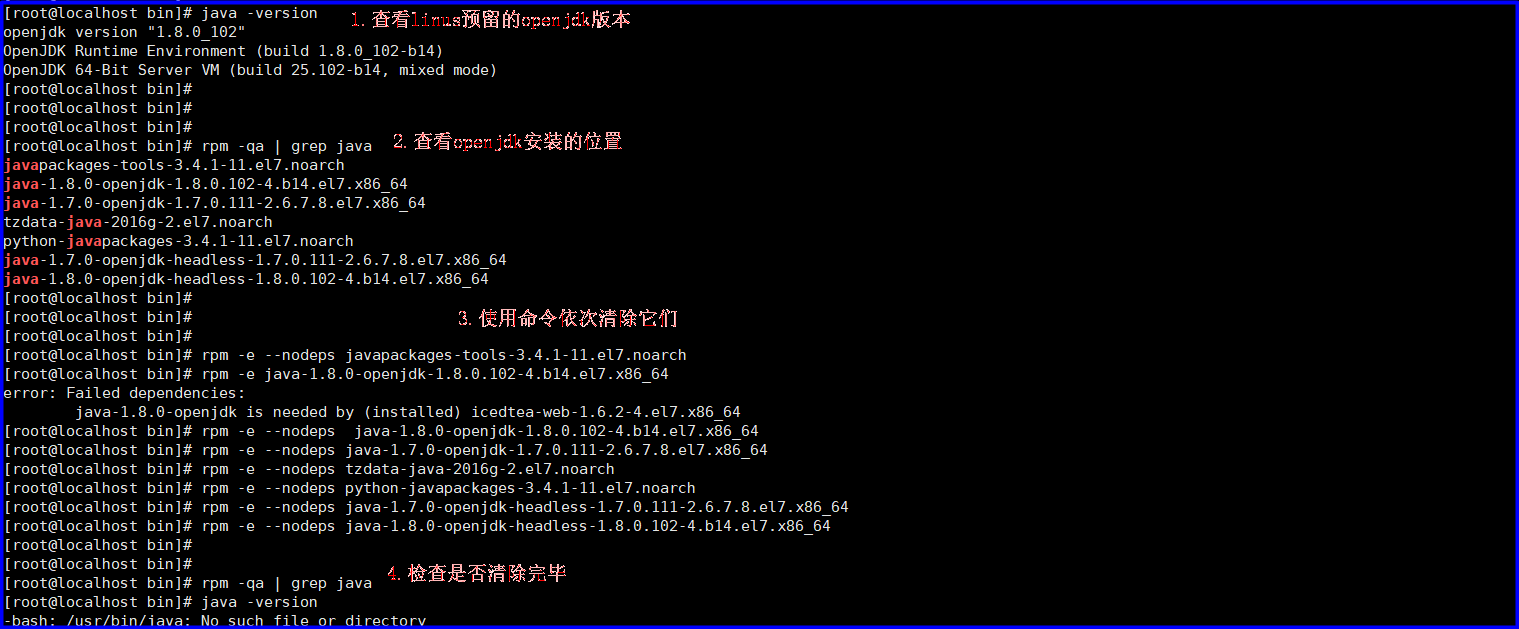
# 从零在虚拟机搭建一个4个节点的CenterOS集群



## 在每个CentorOS中安装jdk和perl

### 安装jdk

（1）卸载linus中自带的openjdk



（2）下载rmp的jdk包并安装

//其中i-install表示安装

//v-verbose安装过程详细信息

//h-hash表示显示进度

rpm -ivh jdk-7u65-linux-i586.rpm

（3）配置jdk环境变量

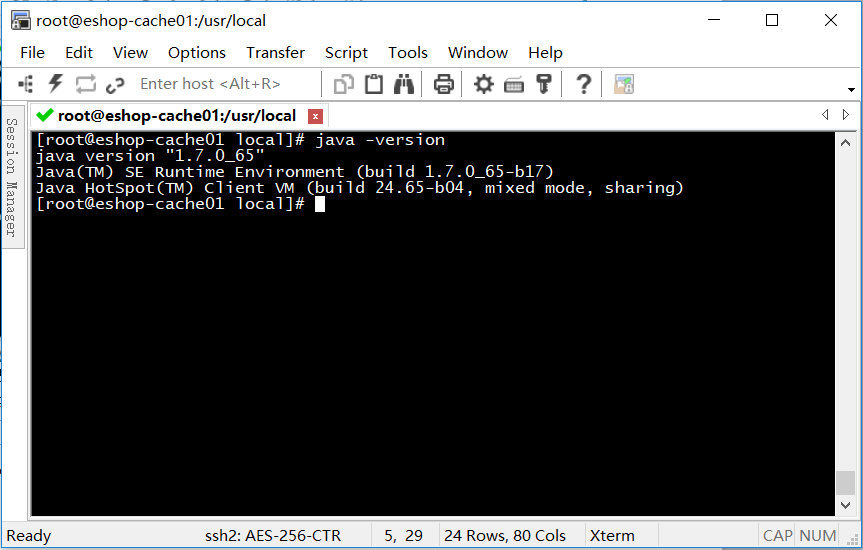
vi ~/.bashrc //find / -name .bashrc,寻找.bashrc文件位置

export JAVA\_HOME=/usr/java/latest

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

source ~/.bashrc //退出文件执行source

bash 是一个能解释你输入进终端程序的东西，并且基于你的输入来运行命令。它在一定程度上支持使用脚本来定制功能，这时候就要用到 .bashrc 了。为了加载你的配置，bash 在每次启动时都会加载 .bashrc 文件的内容。每个用户的 home 目录都有这个 shell 脚本。它用来存储并加载你的终端配置和环境变量。终端配置可以包含很多不同的东西。最常见的，.bashrc 文件包含用户想要用的别名。别名允许用户通过更短的名字或替代的名字来指向命令，对于经常在终端下工作的人来说这可是一个省时利器。



### 安装perl

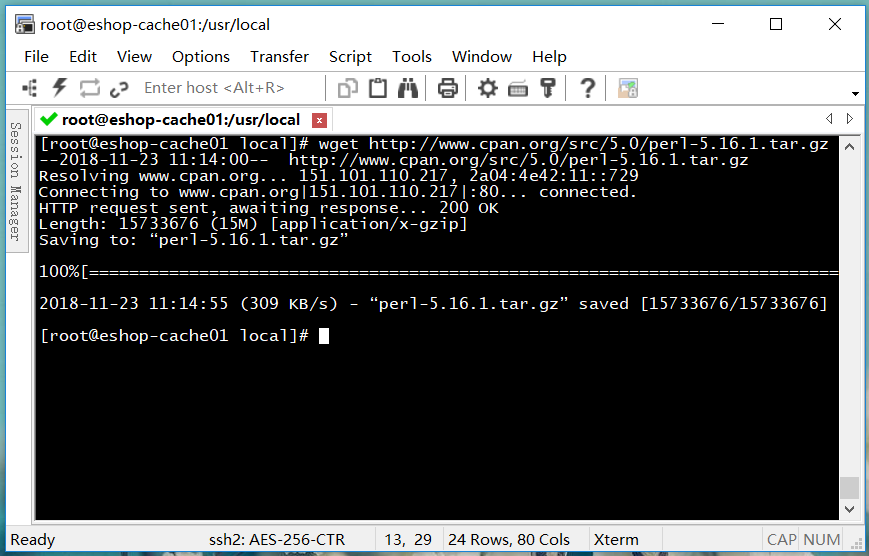
（1）为什么要装perl

整个大型电商网站的详情页系统比较复杂;java+nginx+lua需要perl，它是一个基础的编程语言的安装，tomcat跑java web应用需要用到。

（2）安装perl前的准备工作

//1.利用wget命令下载perl安装包并解压

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



//解压perl压缩包并删除tar.gz压缩包

//tar -zxvf 文件.tar.gz -C /path

//z-gunzip表示该文件是被gunzip解压的,具有gzip属性的

//x-extract表示将文件提取出来

//v-verbose显示解压过程中的一些信息

//f-file指定解压的文件,后面必须跟文件名

//-C 指定解压到文件的位置

tar -zxvf perl-5.16.1.tar.gz

rm -rf perl-5.16.1.tar.gz

//2.安装语言编译库gcc

yum -y install gcc //耗时较长

（3）安装perl

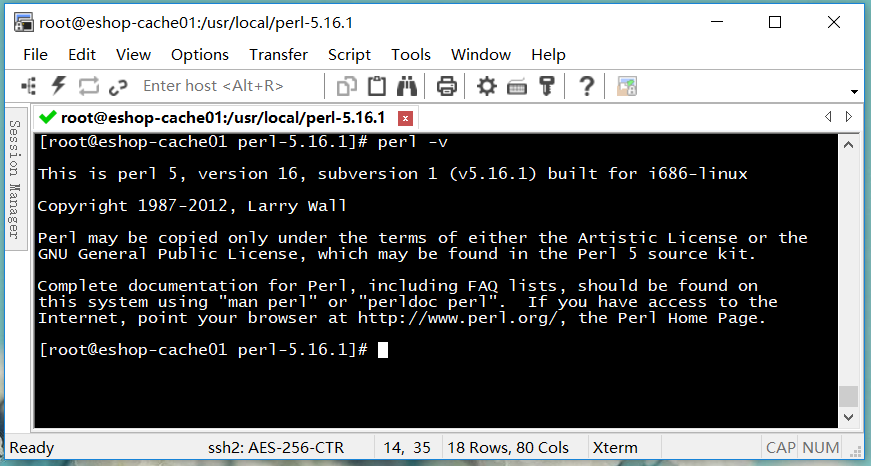
cd /usr/local/perl-5.16.1//进入per解压后的目录

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

make && make test && make install //耗时很长

（4）测试perl是否装好

perl -v

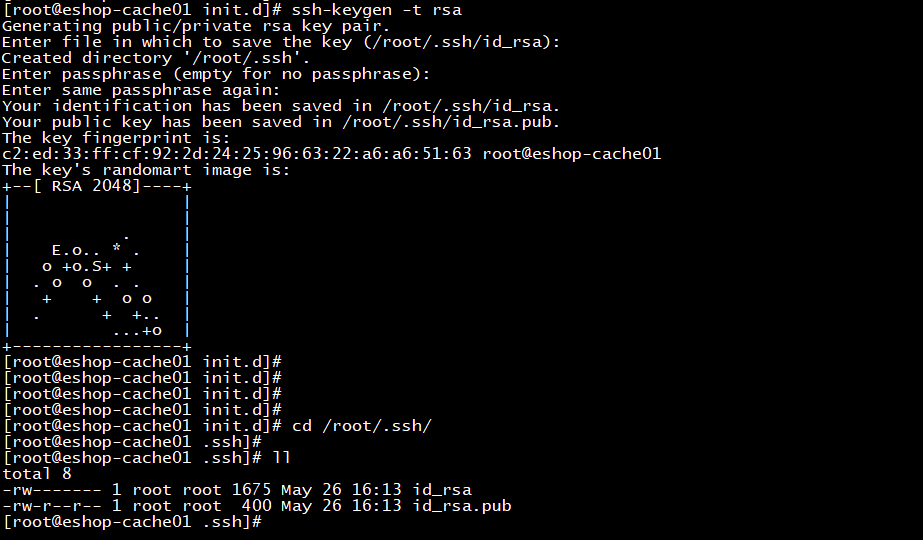


## 为4台CentOS配置ssh免密互相通信

### 生成本机公钥

//过程中不断敲回车,公钥放在/root/.ssh目录下

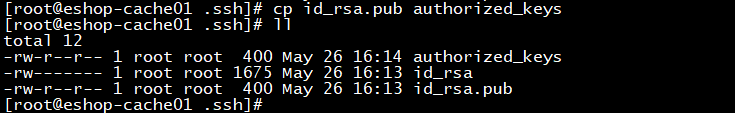
ssh-keygen -t rsa



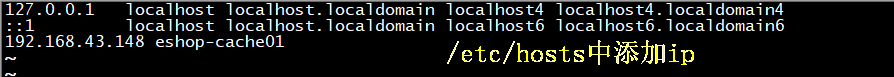
### 将公钥cp一份为authorized\_keys

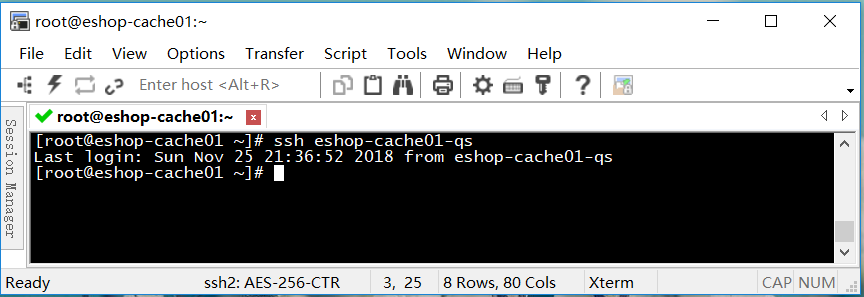
cd /root/.ssh

cp id\_rsa.pub authorized\_keys



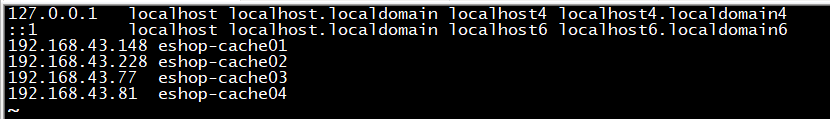
此时ssh以ip免密码可以直接登录，还需要在每台机器中的/etc/hosts中配置所有电脑的ip name，以ssh hostname登录更方便，如下图所示：

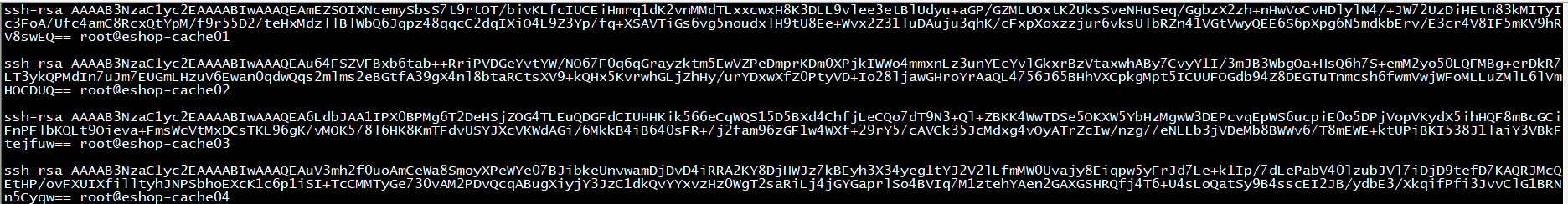




### 配置三台机器之间ssh免密码登录

保证每台机器/etc/hosts和/root/authorized\_keys(四台机器综合)内容一致，其中authorized\_keys是四台机器各自生成并copy公钥内容的总和





### /root/.ssh/中文件作用

（1）authorized\_keys：

里面记录了每台机器的访问权限信息-公钥-id\_rsa.pub,有其他机器的公钥时，其他机器访问本机器时无需密码;

（2）id\_rsa:

本机器的私钥,在执行命令 ssh-keygen -t rsa时和公钥id\_rsa.pub一起生成;

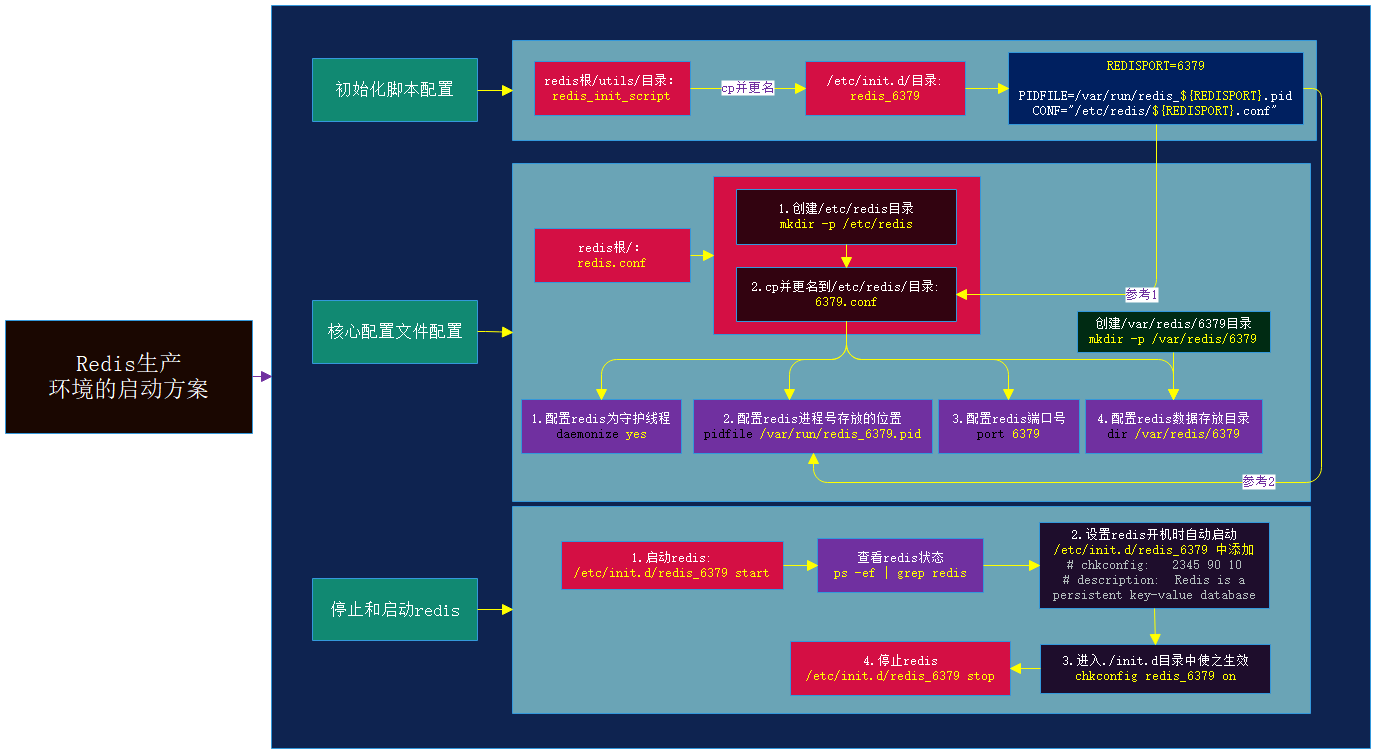
（3）id\_rsa.pub:

本机器的公钥，需要将它cp为authorized\_keys，方便本机器访问本机器，并且便于其他机器作为识别是否有本机器权限的一个唯一标识;

（4）know\_hosts:

会记录本机器访问过其他哪些机器的公钥，如果没有记录，第一次会询问yes/no，若选择yes，则会在文件中记录访问机器的公钥;

# Redis单机版的安装及生产环境启动方案



## Redis单机版的安装

### 下载并安装tcl

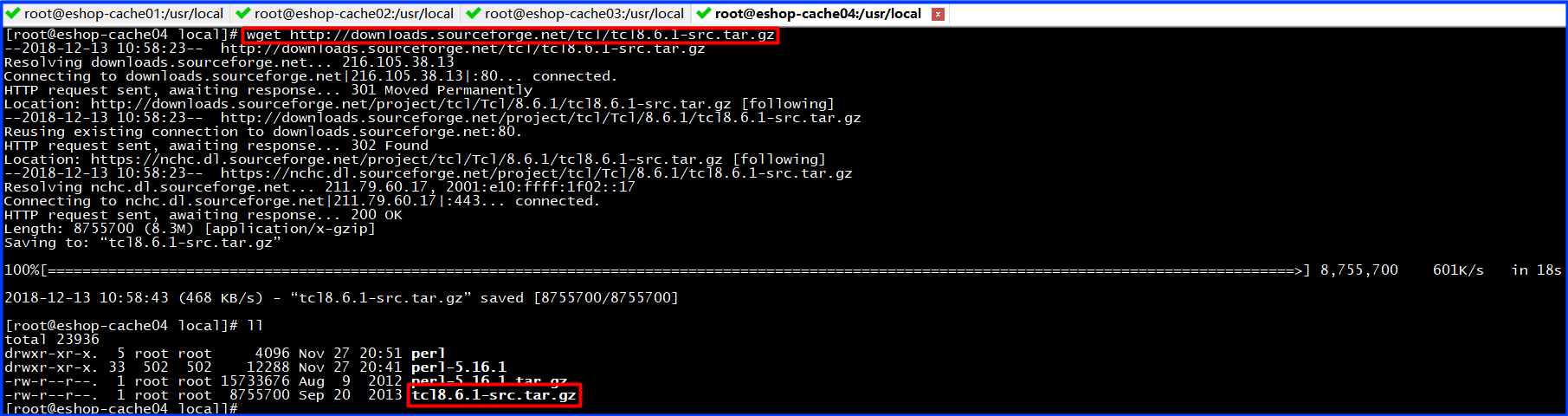
1、下载wget

wget 是一个从网络上自动下载文件的自由工具，支持通过 HTTP、HTTPS、FTP 三个最常见的 [TCP/IP协议](https://baike.baidu.com/item/TCP/IP%E5%8D%8F%E8%AE%AE) 下载，并可以使用 HTTP 代理。"wget" 这个名称来源于 “World Wide Web” 与 “get” 的结合。



2、下载tcl

wget http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz



3、安装tcl

1):解压tcl

tar -zxvf tcl8.6.1-src.tar.gz

2):执行.configure文件

cd /usr/local/tcl8.6.1/unix/

./configure

3):安装tcl-漫长

cd /usr/local/tcl8.6.1/unix

make && make test && make install

### 安装redis

1、准备redis的安装包

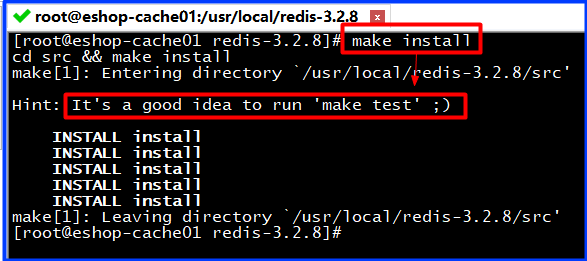


2、解压并安装-漫长

tar -zxvf redis-3.2.8.tar.gz

cd /usr/local/redis-3.2.8

make && make test && make install



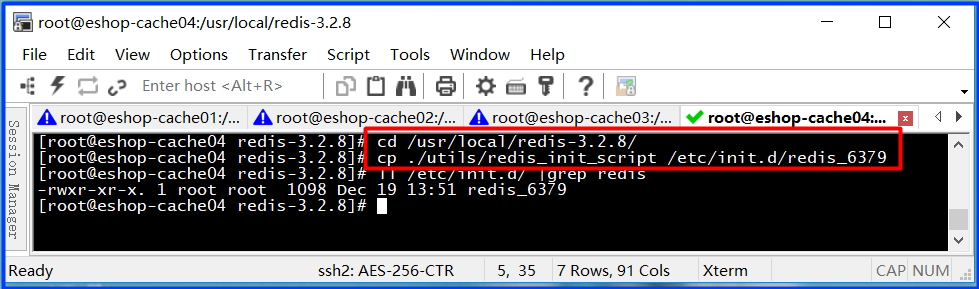
1. 再次在../redis-3.2.8目录下安装make install,否则报如下错误



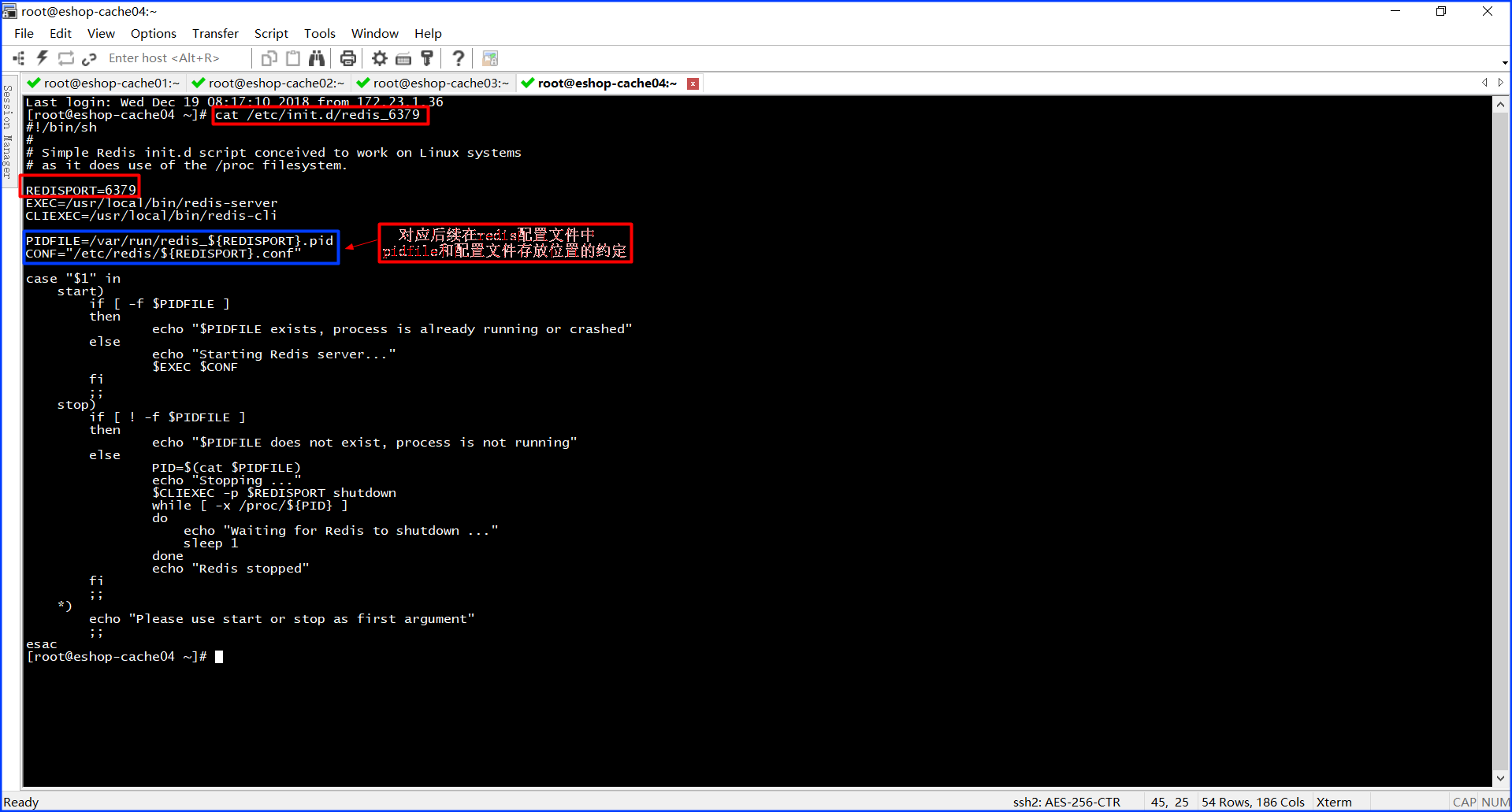
## Redis生产环境的启动方案

### 初始化脚本文件配置

将redis/utils/redis\_init\_script拷贝到/etc/init.d/下,并重命名为redis\_6379

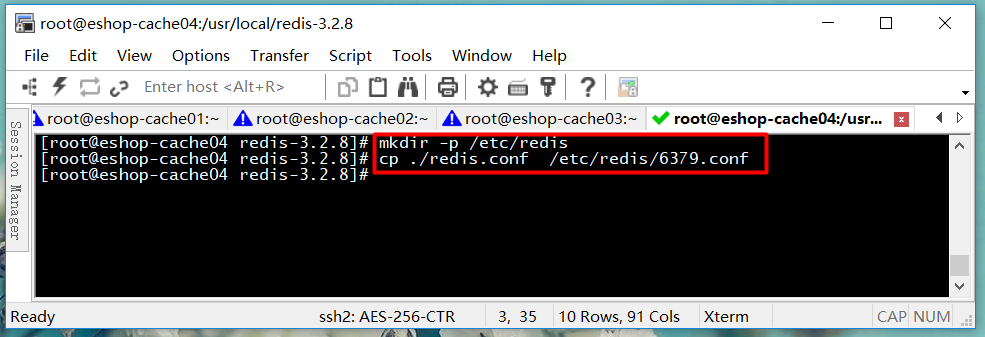


默认redis\_6379文件中的REDISPORT为6379



### 核心配置文件配置

1.将redis根/redis.conf拷贝到/etc/redis/目录下，并重命名为6379.conf



2.配置/etc/redis/6379.conf

1):daemonize yes

使redis以守护进程的方式运行;

2):pidfile /var/run/redis\_6379.pid

启动redis成功后,会在/var/run/目录下生成redis\_6379.pid记录当前进程号;

3):port 6379

redis的默认端口;

4):dir /var/redis/6379

/var/redis/6379该目录需要创建,是redis的数据存储位置;

### 启动和停止redis服务

1.常规启动

1):启动redis:

chmod 777 /etc/init.d/redis\_6379

/etc/init.d/redis\_6379 start

2):查看redis是否启动成功:

//ps-process是linus下最常用的也是非常强大的进程查看命令

//e-environment

//f-format

//grep-global regular expression print

ps -ef | grep redis

2.停止redis服务

/etc/init.d/redis\_6379 stop

### 让redis服务开机自动自动

1.在/etc/init.d/redis\_6379中添加一下两行

# chkconfig: 2345 90 10

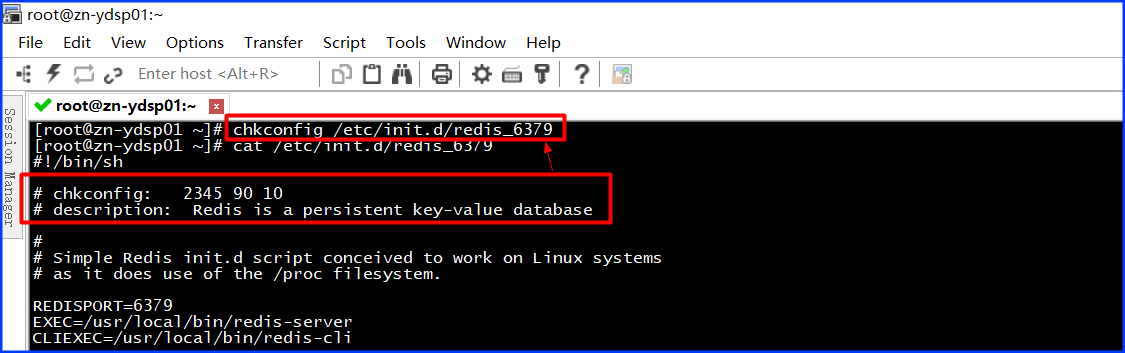
# description: Redis is a persistent key-value database

redis服务必须在运行级2，3，4，5下被启动或关闭，启动的优先级是90，关闭的优先级是10

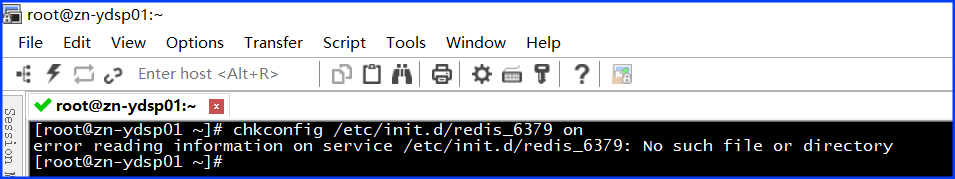
1. 执行如下命令

cd /etc/init.d/

chkconfig redis\_6379 on//此处一定要进入到目录中执行

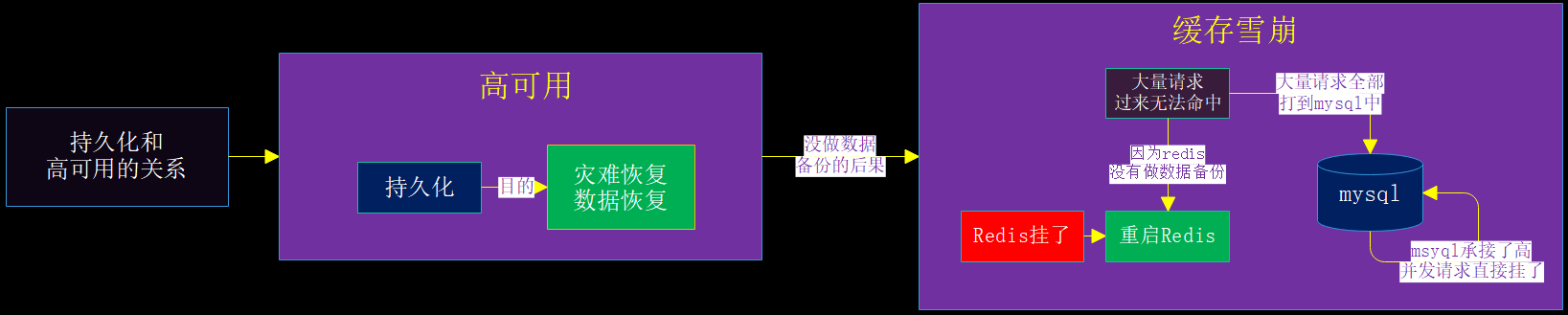


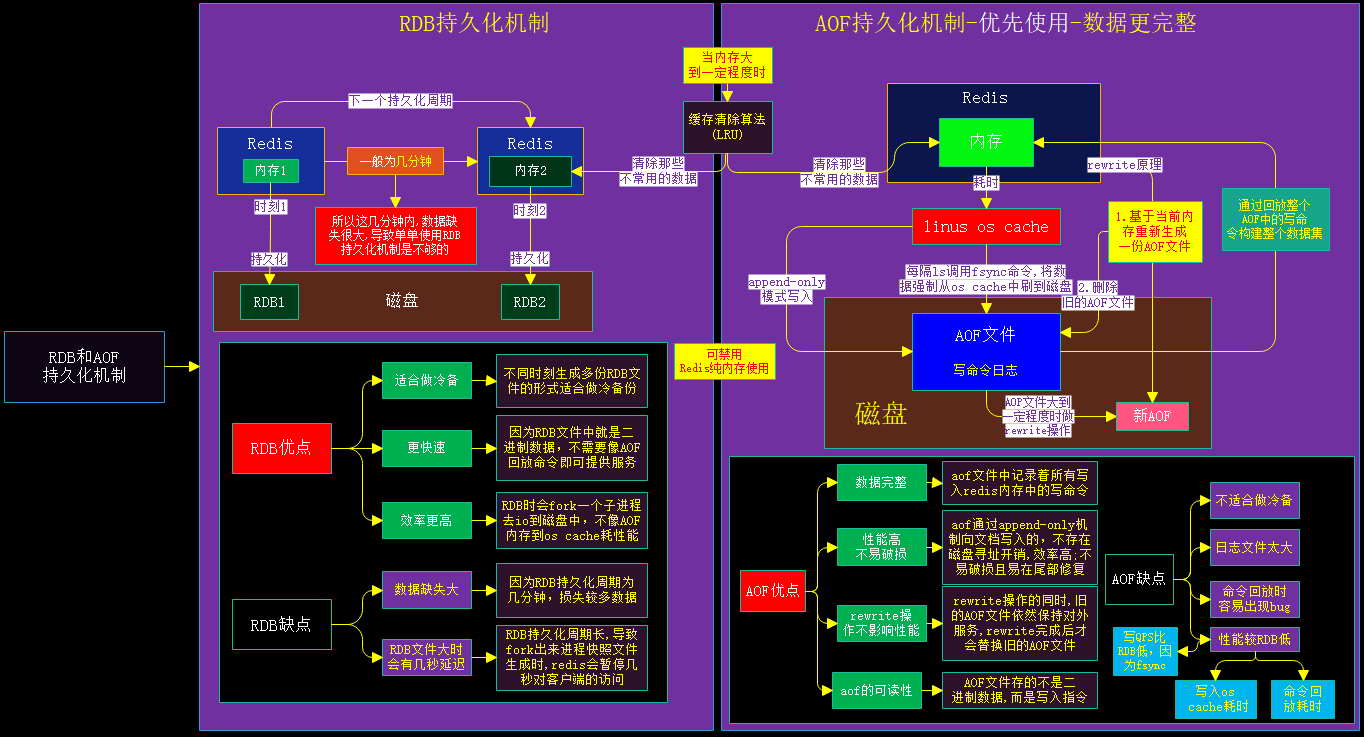
否则报如下错误

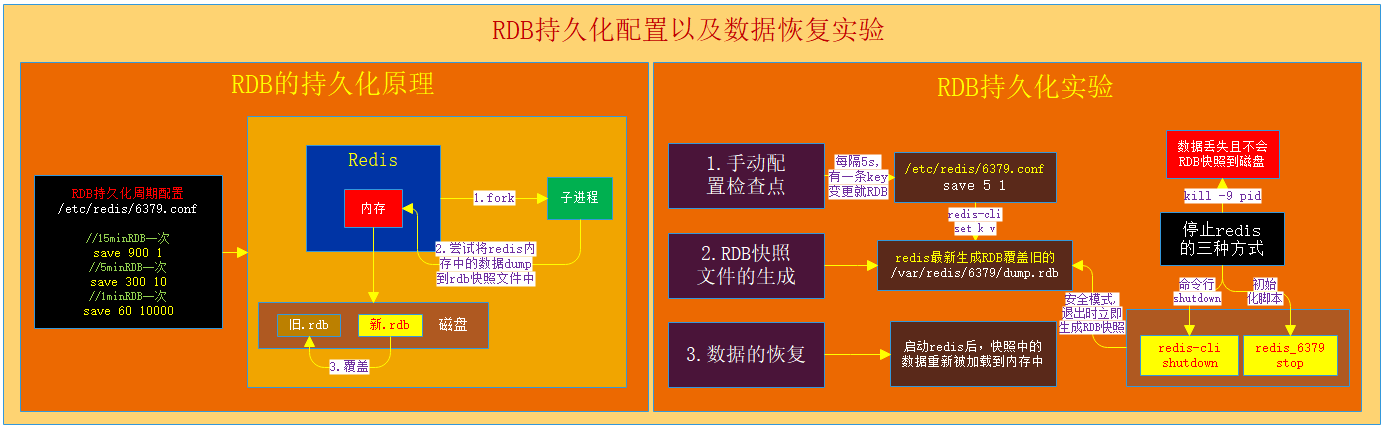


# Redis持久化机制工作原理和企业级数据备份恢复演练

## 持久化机制工作原理和持久化配置演练

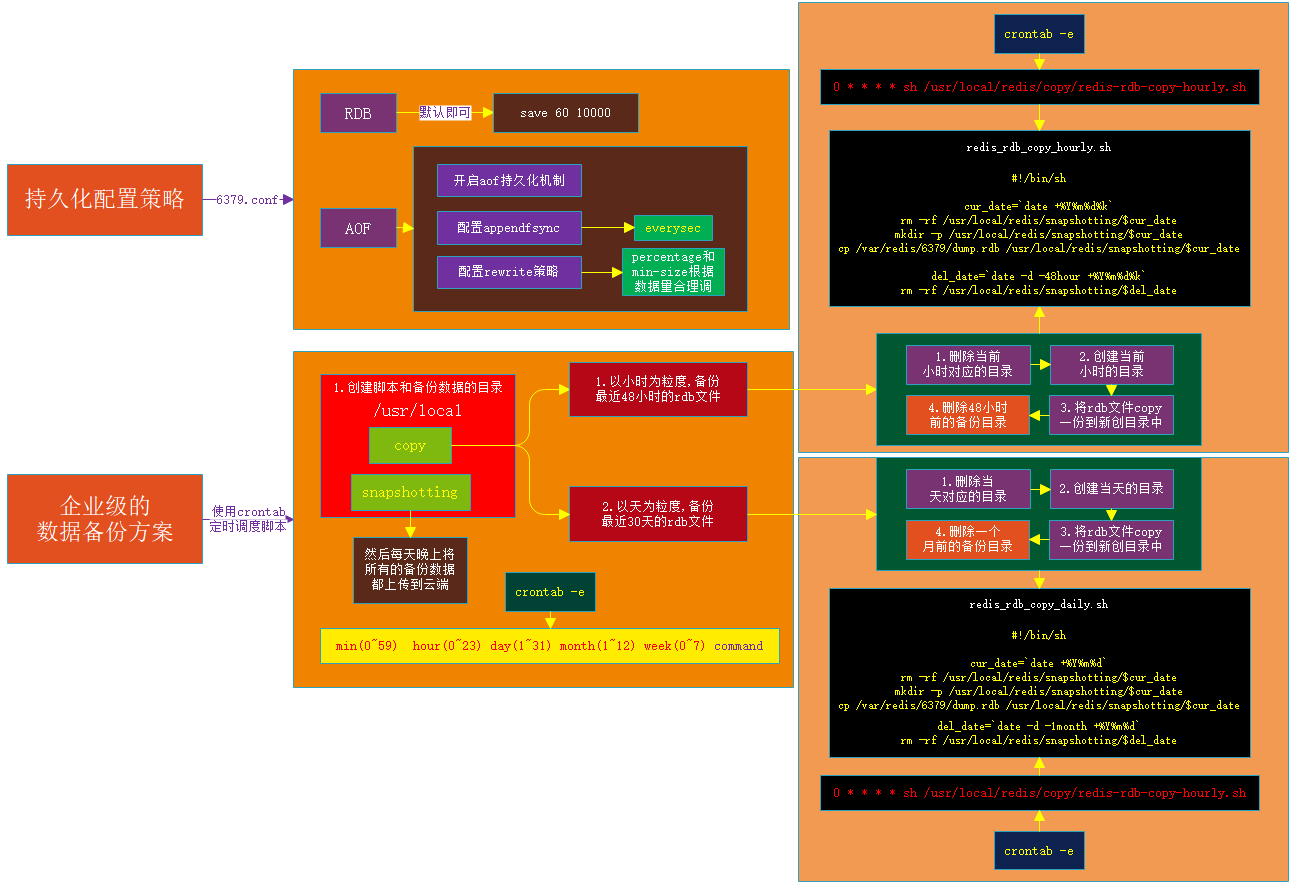


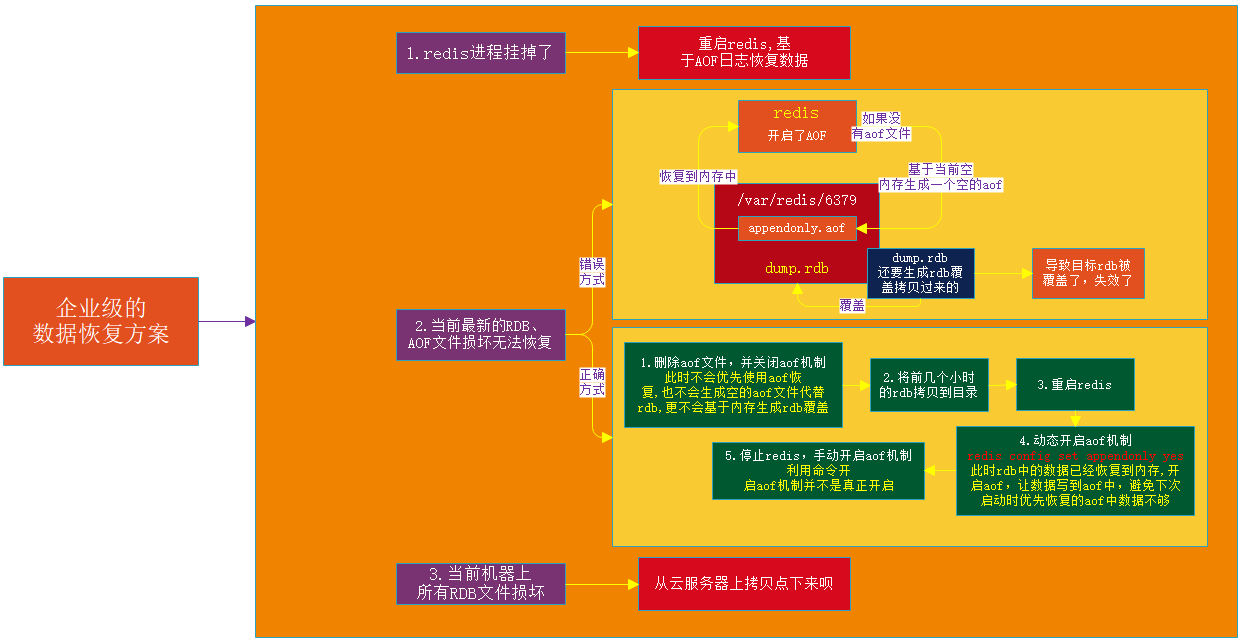




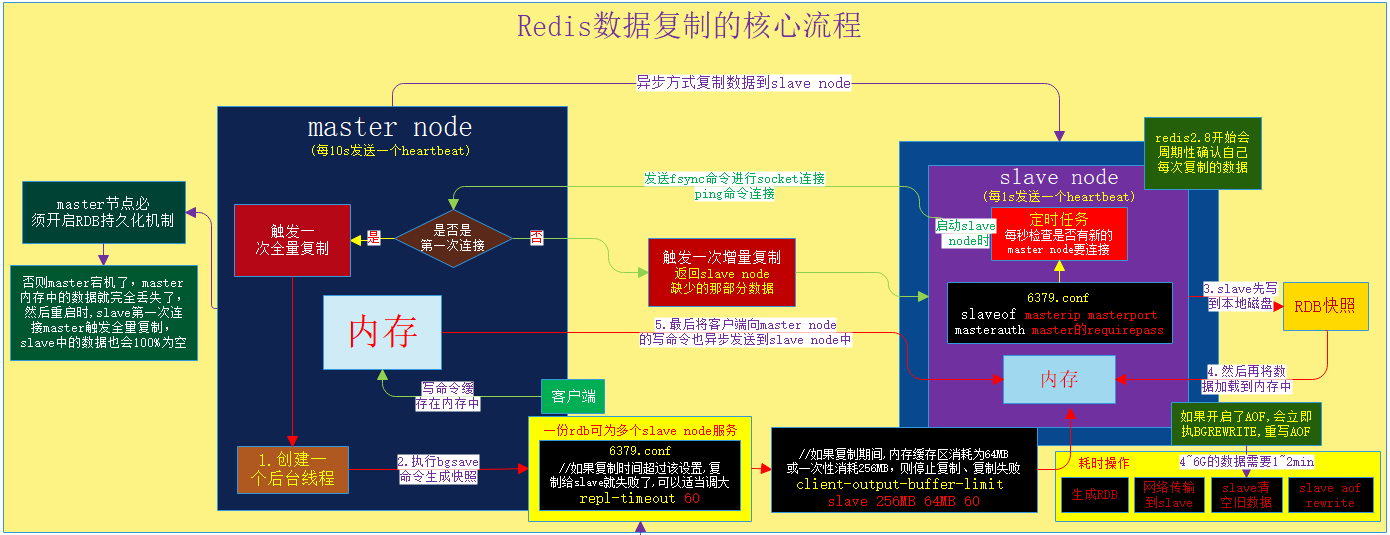


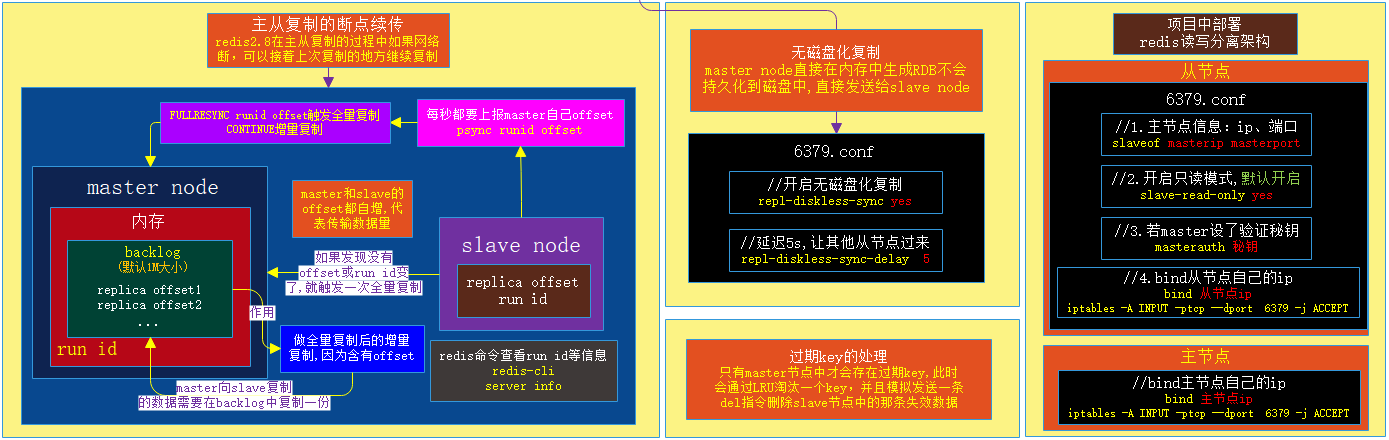
## 企业级数据备份方案及数据恢复容灾演练





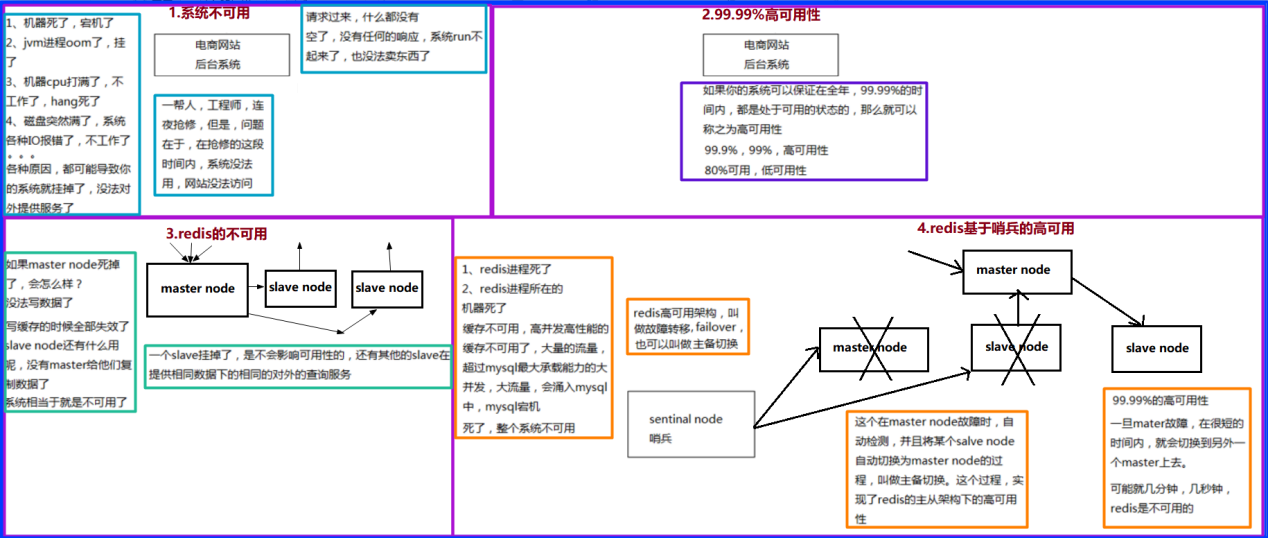
# 构建Redis主从架构-读写分离-支持水平扩容的读高并发架构







# redis主从架构下如何才能做到99.99%的高可用性?



# 哨兵架构的基础

## sentinal/哨兵的主要功能

### 集群监控

负责监控redis master和slave进程是否正常工作;

### 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;

### 故障转移

如果master node挂掉了,会自动转移到slave node上;

### 配置中心

如果故障转移发生了,通知client客户端连接新的master地址;

分布式选举,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,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;

哨兵集群的高可用性,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,就不能保证高可用的特点了;

哨兵新版本的完善,目前采用的是sentinal 2版本,sentinal 2相对于sentinal 1来说,重写了很多代码,主要是让故障转移的机制和算法变得更加健壮和简单;

## 哨兵的核心点

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;

## 哨兵集群只有2个实例是无法工作的

+----+ +----+

| M1 |---------| R1 |

| S1 | | S2 |

+----+ +----+

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,配置中就Configuration: quorum = 1,master宕机,s1和s2中只要有1个哨兵认为master宕机就可以进行切换,同时s1和s2中会选举出一个哨兵来执行故障转移,同时这个时候就需要majority,也就是大多数哨兵都是运行的;

2的majority=2,3的majority=2,4的majority=2,5的majority=3,2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;

## 经典的3节点哨兵集群

+----+

| M1 |

| S1 |

+----+

+----+ | +----+

| R2 |----+----| R3 |

| S2 | | S3 |

+----+ +----+

Configuration: quorum = 2,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;

## 思考和注意

### 是哨兵集群至少要三个实例而不是redis

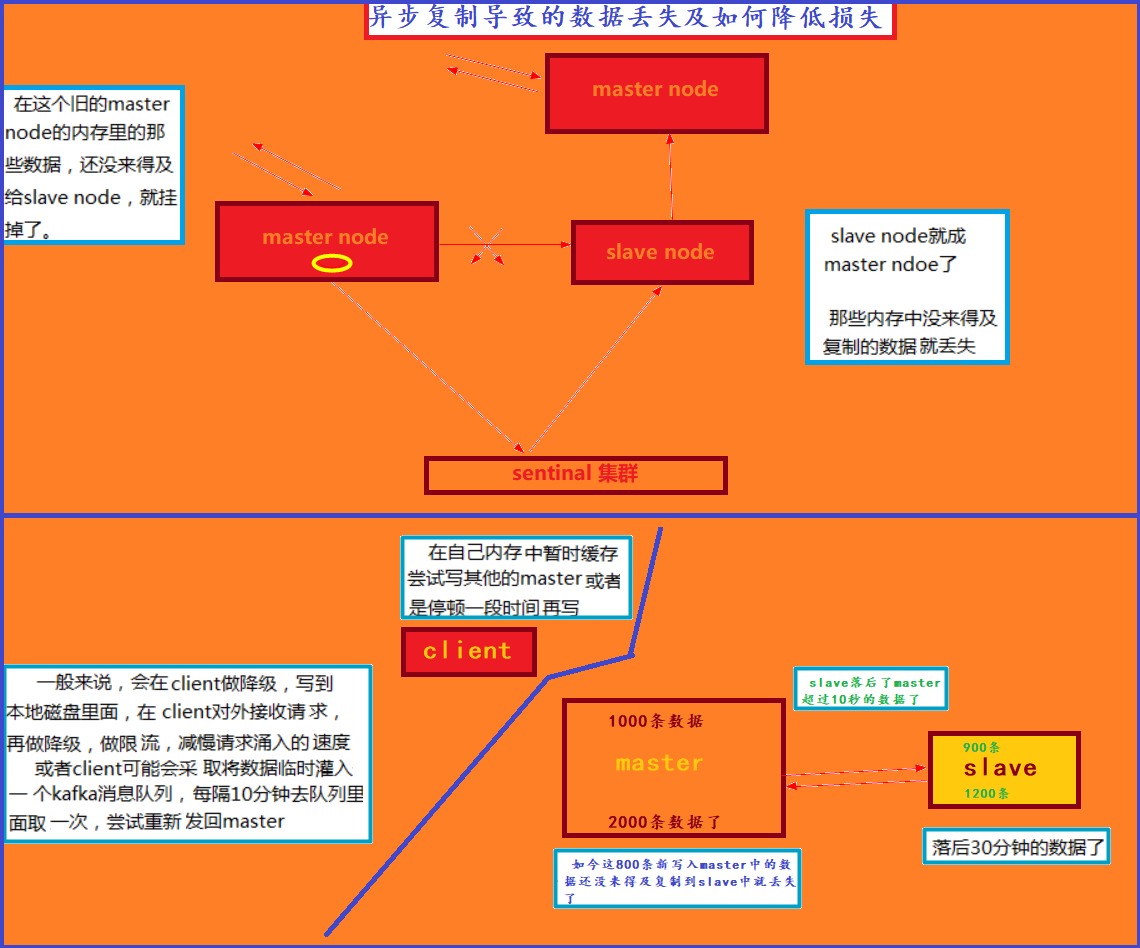
### 执行故障转移功能的是哨兵不是redis,且需要选举

# 异步复制和集群脑裂导致的数据丢失

## 两种数据丢失的情况

### 异步复制导致数据丢失

主备切换的过程,可能会导致数据丢失,因为master到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;



### 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,此时哨兵可能就认为master宕机,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,还继续写向旧的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;

## 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1个slave,且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;

### 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master宕机后损失的数据太多了,那么就拒绝client向master再写请求,这样可以把master宕机时由于部分数据未同步到slave,而导致的数据丢失,降低到可控范围内;

### 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;

## 思考和注意

### 主从节点异步复制的深度理解

正是由于主从节点数据的复制是异步而不是同步的，所以在主备切换的时，会存在少部分的数据的丢失。

### 脑裂后源主节点的处理和数据变化

原主节点重新启动时，常会被设置成从节点并清空内存中的数据，再连接到新选举出的主节点上;

## The remote system refused the connection

### 检查是否联网



# Redis哨兵的多个核心底层原理深入解析

## 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down-after-milliseconds参数指定的毫秒数后,就主观认为master宕机;sdown到odown的转换条件很简单,如果一个哨兵,在指定的时间内收到了来自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;

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,该哨兵还得得到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,那就3个哨兵授权就可以执行切换;但是如果quorum >= majority,那必须quorum数量的哨兵都授权,比如5个哨兵,quorum是5,那必须5个哨兵都同意授权,才能执行切换;

## 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;

## 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复制现有master的数据;

如果slave连接到了一个错误master上,如故障转移之后,那么哨兵会确保它们连接到正确的master上;

## 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.与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;

## 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,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;

## 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;

# 在项目中以经典三节点方式部署哨兵集群

## 哨兵的配置文件

../redis根/sentinel.conf,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,如下配置中的mymaster:

sentinel monitor master-group-name hostname port quorum

sentinel monitor mymaster 172.23.22.171 6379 2

sentinel down-after-milliseconds mymaster 60000

sentinel failover-timeout mymaster 180000

sentinel parallel-syncs mymaster 1

sentinel monitor resque 172.23.41.135 6380 4

sentinel down-after-milliseconds resque 10000

sentinel failover-timeout resque 180000

sentinel parallel-syncs resque 5

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,因为后面分布式集群架构里会讲解,可以配置多个master做数据拆分;

上面两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

**quorum：**

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;

**failover-timeout:**

执行故障转移的timeout超时时长;

**parallel-syncs:**

新的master主备切换之后,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;

## 正式的配置

哨兵默认用26379端口,默认不能跟其他机器在指定端口连通,只能在本地访问

mkdir /etc/sentinal

mkdir -p /var/sentinal/5000

/etc/sentinel/5000.conf

port 5000

bind 192.168.31.187

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

port 5000

bind 192.168.31.19

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

port 5000

bind 192.168.31.227

dir /var/sentinal/5000

**sentinel monitor mymaster 192.168.31.187 6379 2**

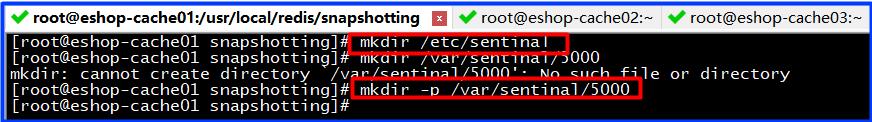
sentinel down-after-milliseconds mymaster 30000

sentinel failover-timeout mymaster 60000

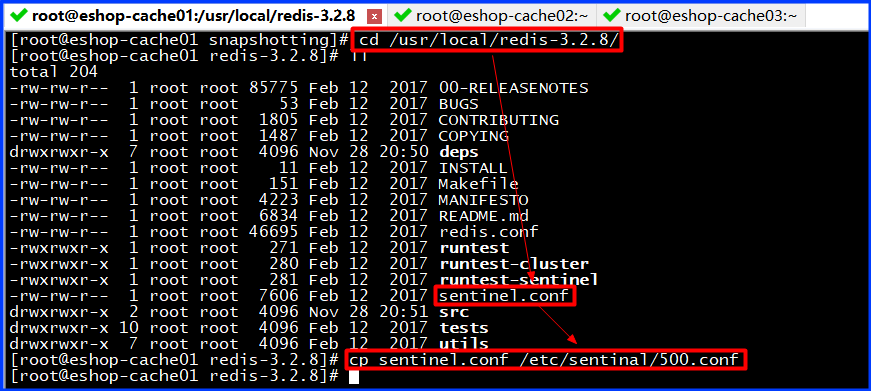
sentinel parallel-syncs mymaster 1

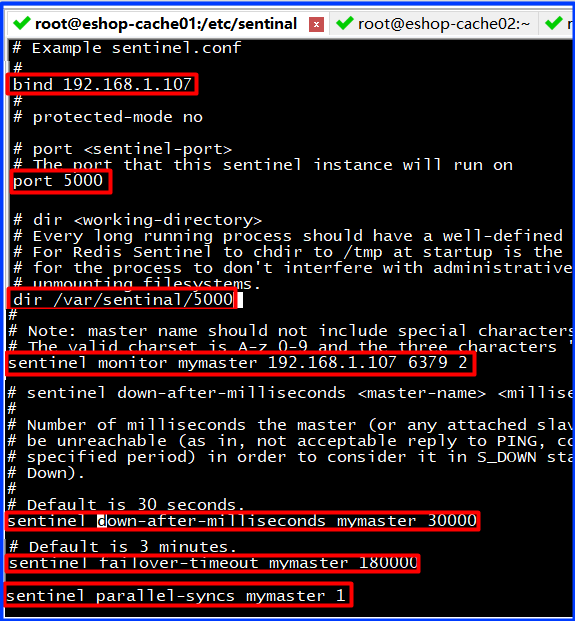
## 实战操作

### 创建哨兵和存放数据文件的目录



### 配置三个哨兵集群的配置





### 启动哨兵进程并检查状态

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,如下检查哨兵的状态,启动哨兵的前提下重新打开一个窗口:

redis-cli -h 192.168.31.187 -p 5000

sentinel master mymaster

sentinel slaves mymaster

sentinel sentinels mymaster

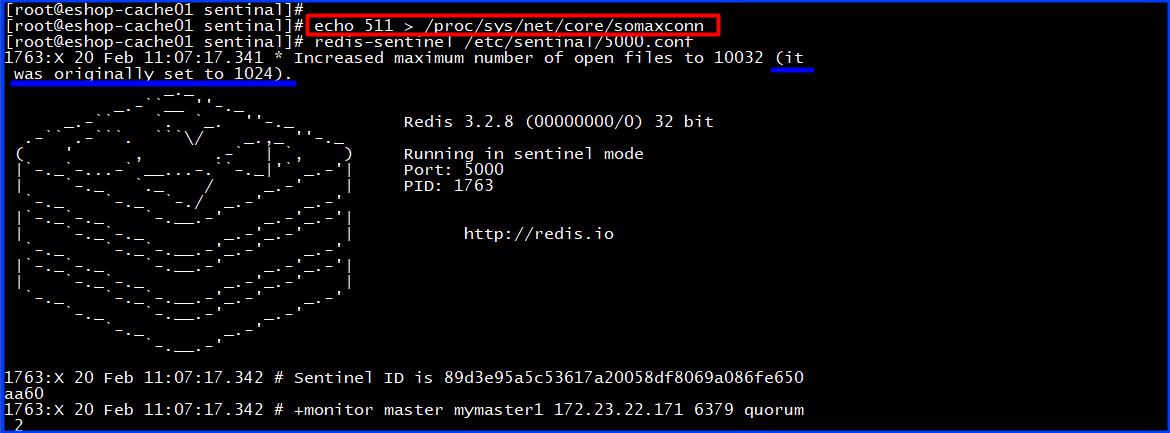
Sentinel get-master-addr-by-name mymaster

## 常见错误

### WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.

1. 动态命令修改

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



2.配置永久解决

在文件/etc/sysctl.conf最后添加net.core.somaxconn= 1024然后执行sysctl -p

### 哨兵启动后直接宕机

1. 查看redis实例是否是因为配置了requirepass和masterauth,而导致哨兵ping主机的时间超过了down-after-millseconds时间而误认为宕机了;
2. 检查Centos的防火墙是否关闭

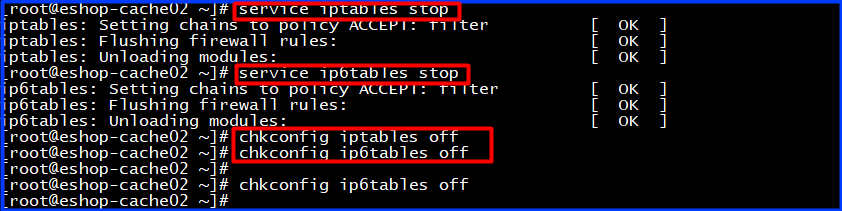
service iptables stop

service ip6tables stop

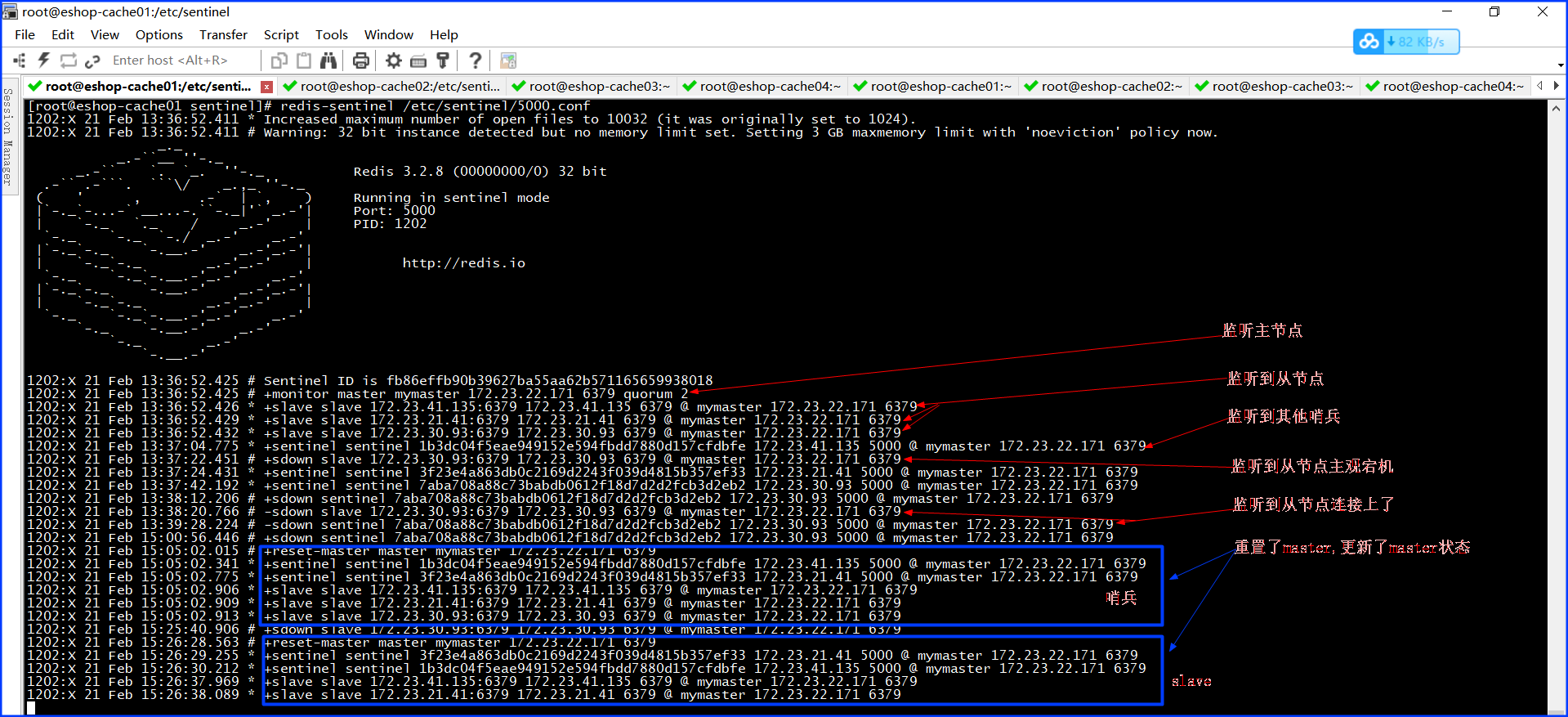
chkconfig iptables off

chkconfig ip6tables off

vi /etc/selinus/config SELINUS=disabled



# 对项目中的哨兵节点进行管理



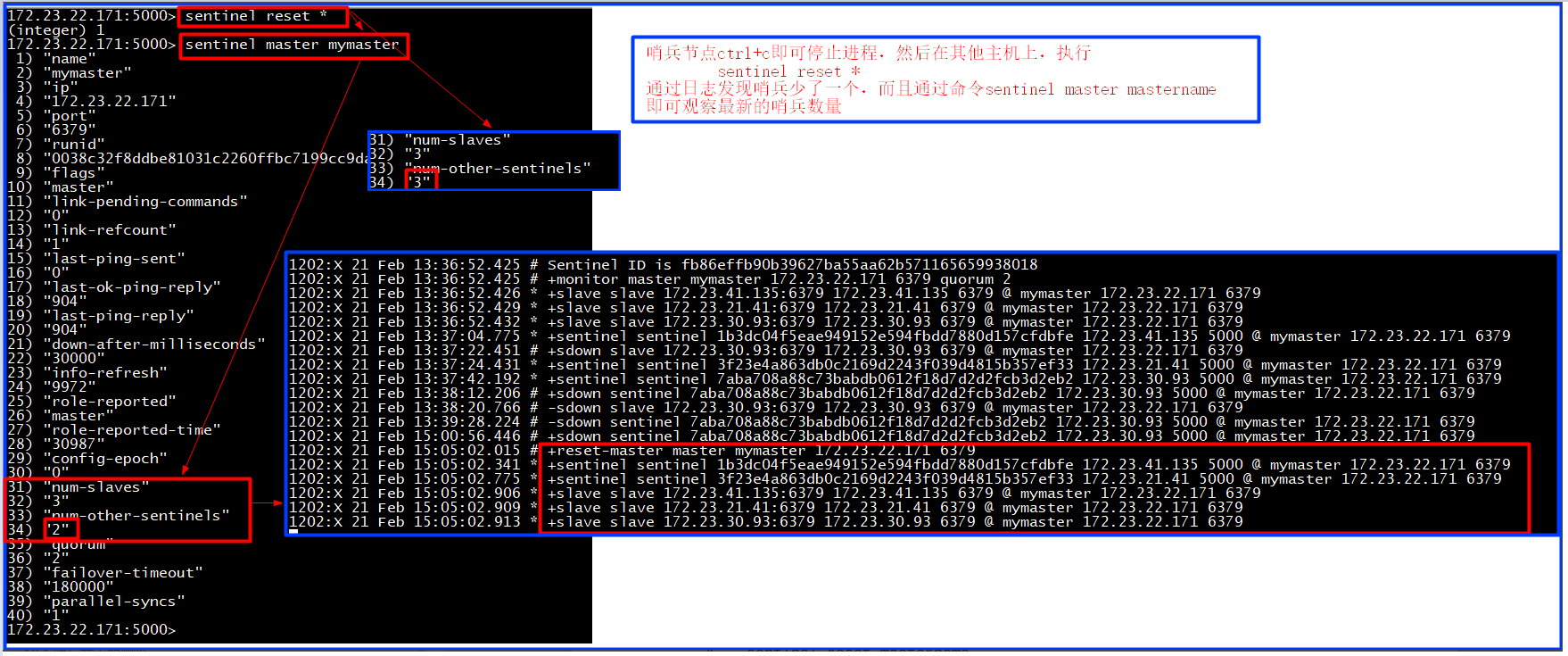
## 哨兵节点的删除

1.停止sentinal进程，crtl+c;

2.在所有sentinal上清理master状态,执行:sentinel reset \*;

3.在所有sentinal上查看sentinal的数量是否达成了一致,执行:

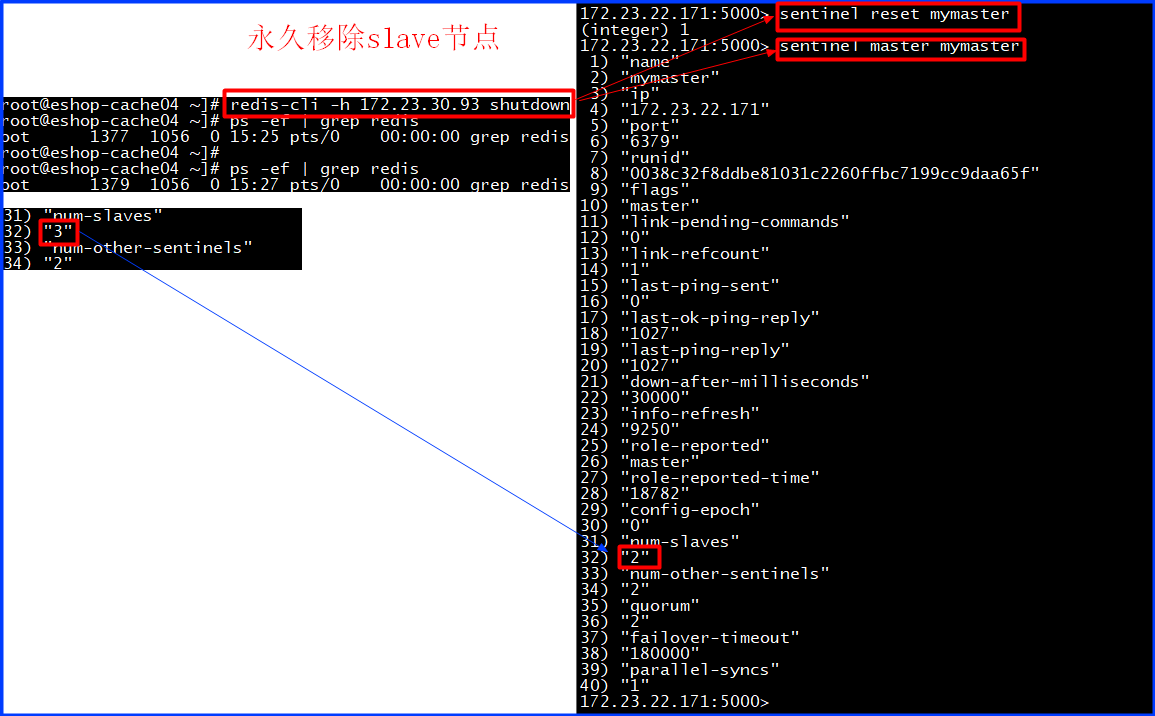
sentinel master mastername;



## slave的永久下线

让master摘除某个已经下线的slave,在所有的哨兵上面执行:

**sentinel reset mastername**



## slave切换为master优先级

6379.conf中的配置slave-priority,值越小优先级越高;

## 基于哨兵集群架构下的安全认证

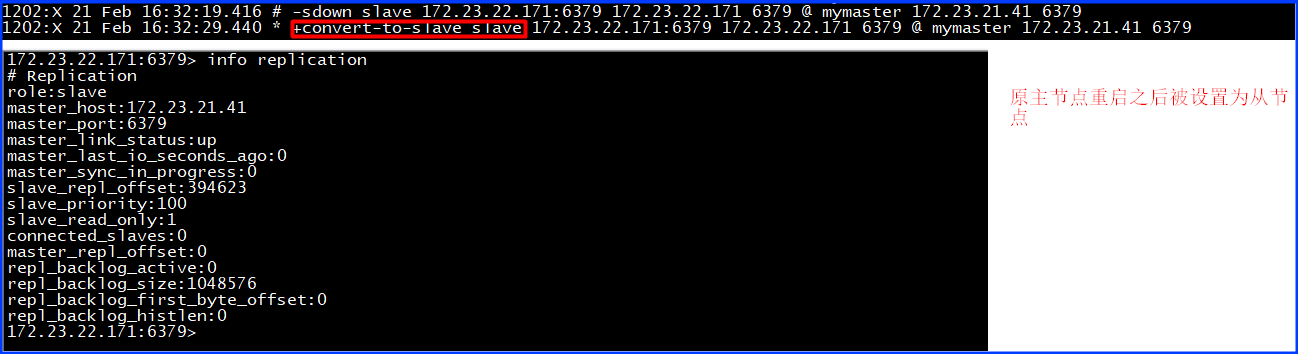
每个slave都有可能切换成master,所以每个实例都要配置两个指令

master上启用安全认证: requirepass

master连接口令: masterauth

# 高可用redis集群的容灾演练





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

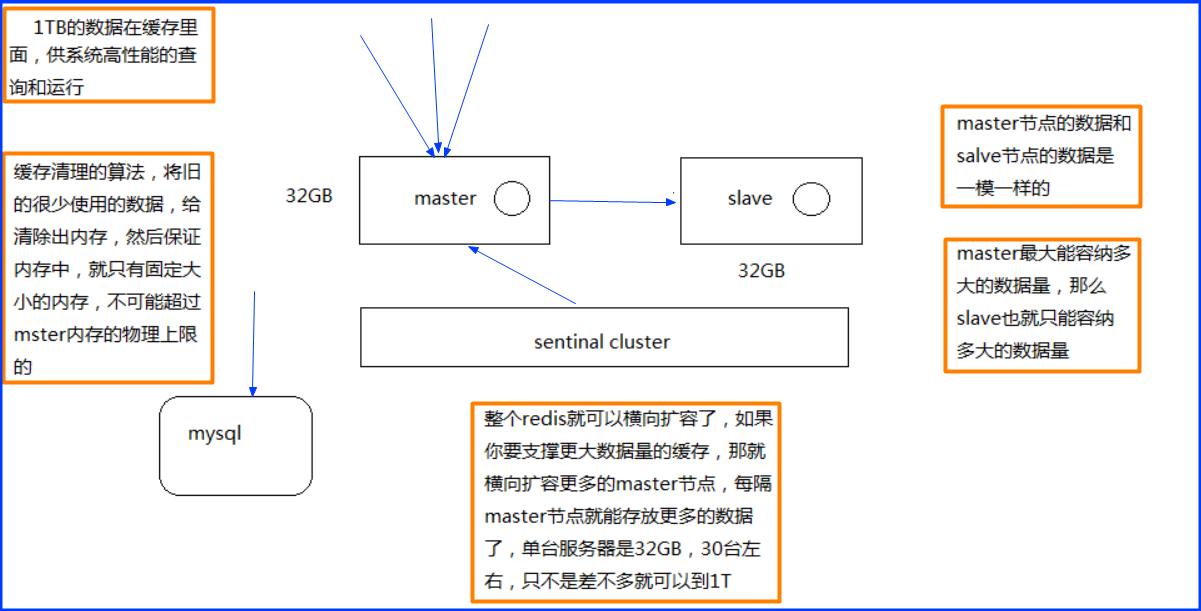
# 哨兵的生产环境部署

mkdir -p /var/log/sentinal/5000

daemonize yes//5000.conf

logfile /var/log/sentinal/5000/sentinel.log//5000.conf

# redis如何在保持读写分离+高可用的架构下，还能横向扩容支撑1T+海量数据



## Redis cluster介绍

自动将数据进行分片,每个master上放一部分数据,提供内置的高可用支持,部分master不可用时,还是可以继续工作的;

在redis cluster架构下,每个redis要开放两个端口号,比如一个是6379,另外一个就是加10000+的端口号,比如16379,16379端口号是用来进行节点间通信的,也就是cluster bus的东西,集群总线;

cluster bus的通信是用来进行故障检测、配置更新、故障转移授权,cluster bus用了另外一种二进制的协议gossip,主要用于节点间进行高效的数据交换,占用更少的网络带宽和处理时间;

## Redis cluster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;

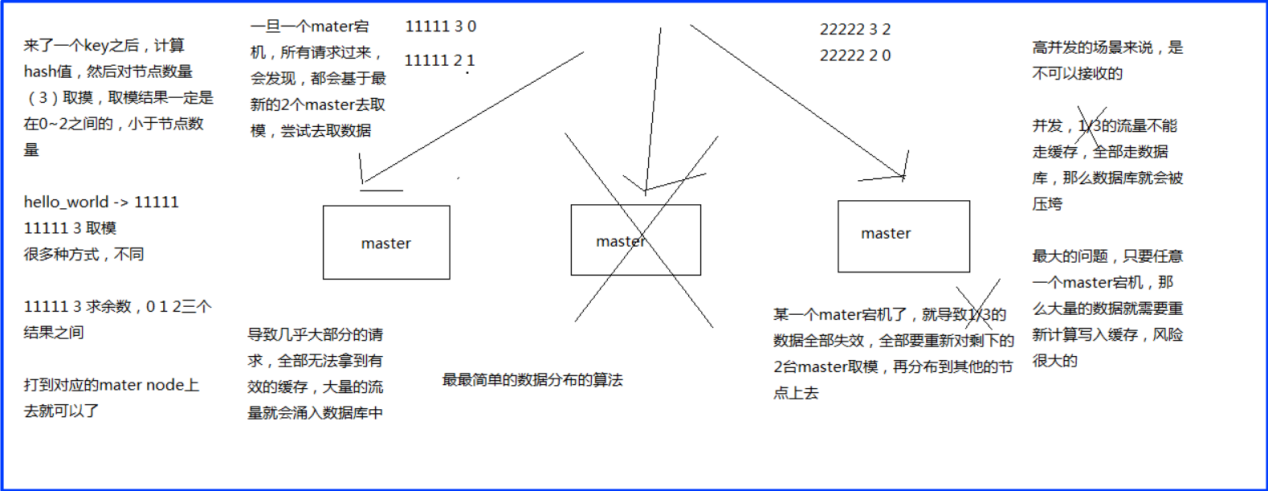
redis cluster多master + 读写分离 + 高可用,我们只要基于redis cluster去搭建redis集群即可,不需要手工去搭建replication复制+主从架构+读写分离+哨兵集群+高可用;

## redis cluster vs. replication + sentinal

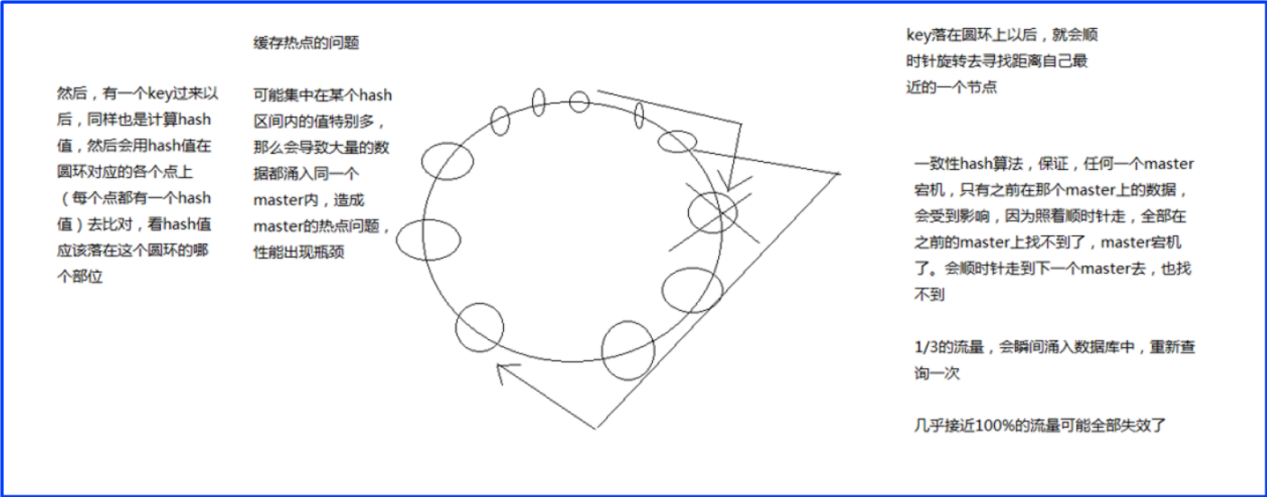
如果你的数据量很少,主要是承载高并发高性能的场景,比如你的缓存一般就几个G,单机足够了,replication,一个mater,多个slave,要几个slave跟你的要求的读吞吐量有关系,然后自己搭建一个sentinal集群,去保证redis主从架构的高可用性,就可以了,redis cluster主要是针对海量数据+高并发+高可用的场景,如果你的数据量很大,那么建议就用redis cluster;

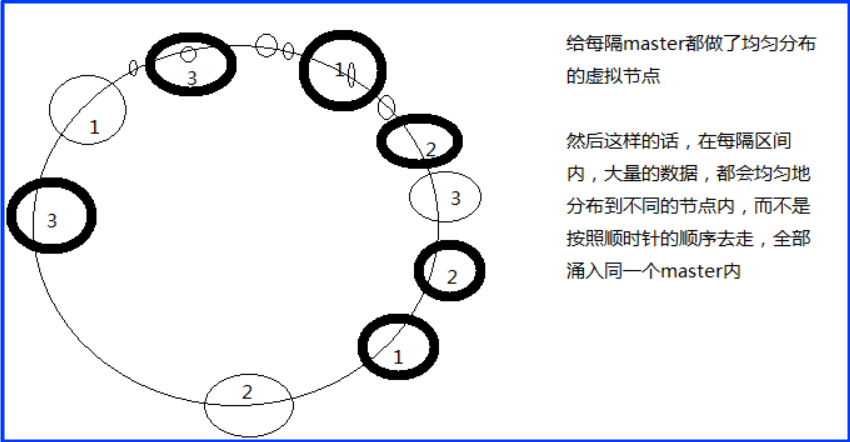
# 数据分布算法

## hash算法和弊端/大量缓存重建



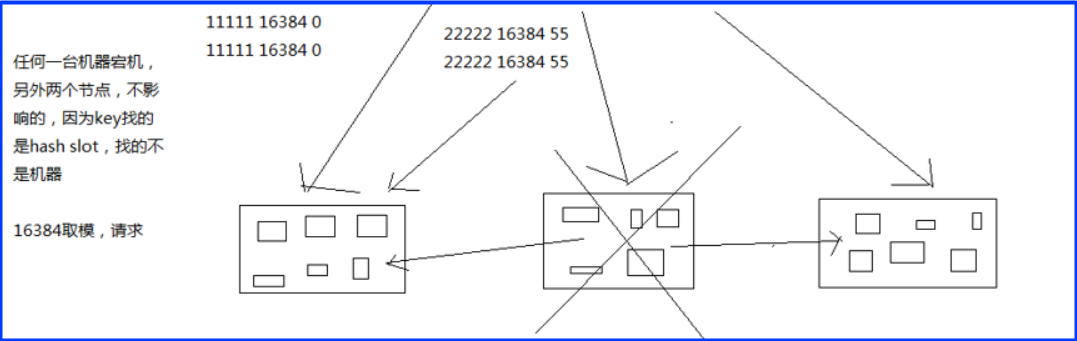
## 一致性hash算法/自动缓存迁移/虚拟节点自动负载均衡





## 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;



# 在项目中重新搭建一套读写分离、高可用、多master的redis cluster集群

## Redis cluster集群的特点

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是最少的要求,正式环境下建议都是说在6台机器上去搭建,但是至少3台机器,保证每个master都跟自己的slave不在同一台机器上;

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;

## 准备生产环境的启动脚本

1.redis\_700\_1、2一台机器，3、4一台机器，5、6一台机器下的/etc/init.d/

**redis\_7001,redis\_7002,redis\_7003,redis\_7004,redis\_7005,redis\_7006**

2.每个启动脚本内，都修改对应的端口号为700x;

## 在3台机器上搭建6个redis实例的redis cluster

### 集群相关配置介绍

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线,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;

### 搭建集群的准备工作

mkdir -p /etc/redis-cluster //是存放node\_700x.conf文件的

mkdir -p /var/log/redis

mkdir -p /var/redis/7001

**daemonize yes**

**pidfile /var/run/redis\_7001.pid**

**dir /var/redis/7001**

**port 7001**

**bind 192.168.31.187**

**logfile /var/log/redis/7001.log**

**appendonly yes**

**cluster-enabled yes**

**cluster-config-file /etc/redis-cluster/node-7001.conf**

**cluster-node-timeout 15000**

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;

## 创建6个redis实例的集群

### 下载和安装ruby

wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.1.tar.gz

tar -zxvf ruby-2.3.1.tar.gz

cd ./ruby-2.3.1

./configure -prefix=/usr/local/ruby

make && make install

cd /usr/local/ruby

cp bin/ruby /usr/local/bin

cp bin/gem /usr/local/bin

/usr/local/bin/ruby --version

//因为系统找命令的文件是从/usr/bin中寻找的，所以需要再/usr/bin中创建//一个链接，指向/usr/local/bin;

ln -s /usr/local/bin/ruby /usr/bin/ruby

ln -s /usr/local/bin/gem /usr/bin/gem

### 下载和安装gem

wget http://rubygems.org/downloads/redis-3.3.0.gem

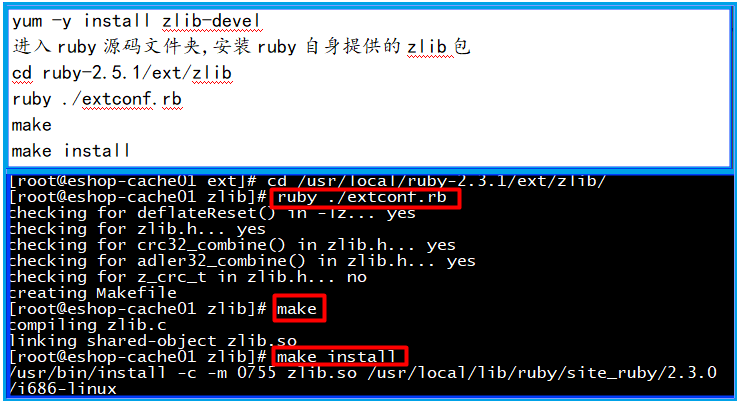
gem install -l ./redis-3.3.0.gem

gem --version

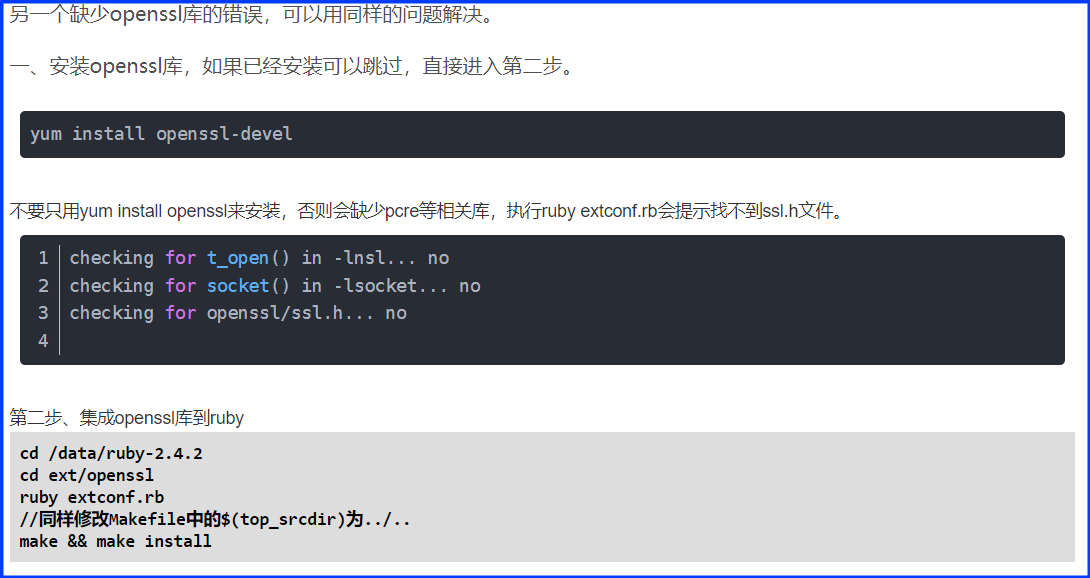
缺少zlib和没有安装openssl库



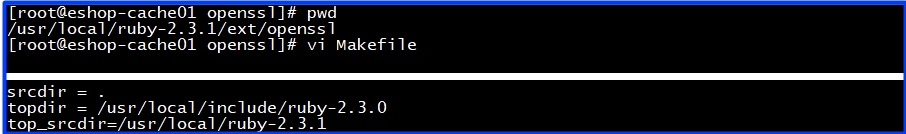
1.安装zlib



2.安装openssl







gem install -l ./redis-3.3.0.gem

gem --version

gem install redis

gem list --check redis gem

cp /usr/local/redis-3.2.8/src/redis-trib.rb /usr/local/bin

### 创建redis-cluster集群

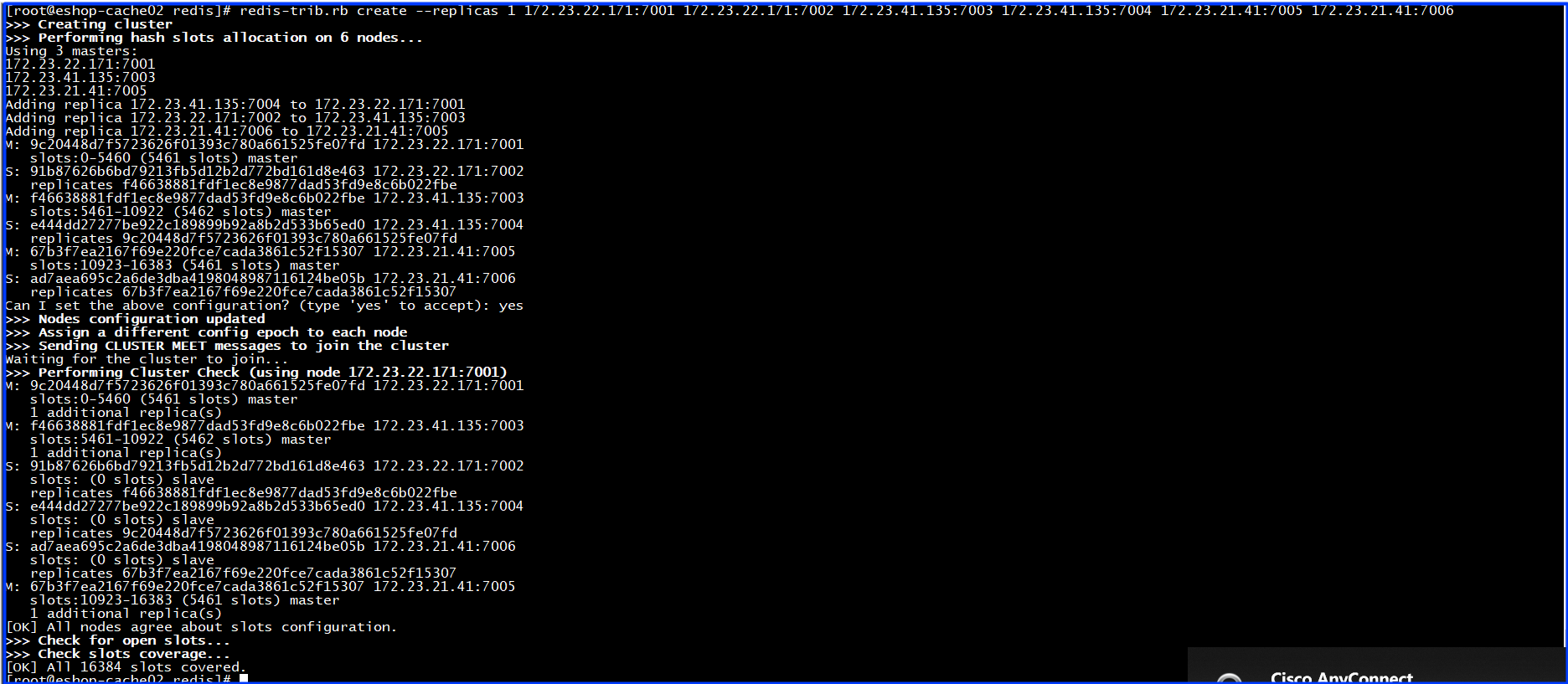
redis-trib.rb create --replicas 1 192.168.31.187:7001 192.168.31.187:7002 192.168.31.19:7003 192.168.31.19:7004 192.168.31.227:7005 192.168.31.227:7006

遇到提示一路yes加回车

--replicas: 每个master有几个slave

### 检查集群状态

redis-trib.rb check 192.168.31.187:7001



### 停止指定ip和端口上的redis服务

redis-cli -h 192.168.43.148 -p 7002 shutdown

# 对项目的redis cluster实验多master写入、读写分离、高可用性

## 实验多master写入、海量数据的分布式存储

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;

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,即分布式的数据存储;如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、大型的java系统架构,还专注在大数据系统架构、分布式、分布式存储、hadoop hdfs、分布式资源调度、hadoop yarn、分布式计算、hadoop mapreduce/hive,分布式的nosql数据库hbase、分布式的协调zookeeper、分布式通用计算引擎spark、分布式的实时计算引擎storm,如果你要处理海量数据,就涉及到了一个名词,叫做大数据,只要涉及到大数据,那么其实就会涉及到分布式,redis cluster分布式,因为我来讲java系统的架构,有时候跟其他人不一样,纯搞java,但是我因为工作时间很长,早期专注做java架构好多年,大数据兴起,就一直专注大数据系统架构,大数据相关的系统,也涉及很多的java系统架构,高并发、高可用、高性能、可扩展、分布式系统,会给大家稍微拓展一下知识面,从不同的角度去讲解一块知识;

redis，高并发、高性能、每日上亿流量的大型电商网站的商品详情页系统的缓存架构,来讲解的,redis是作为大规模缓存架构中的底层的核心存储的支持高并发、高性能、每日上亿流量,redis持久化就是为了应对灾难的时候做数据恢复,复制replication是为了读写分离、扩容slave、支撑更高的读吞吐,redis

怎么支撑读QPS超过10万、几十万,哨兵集群在redis主从、一主多从,保证了99.99%可用性;

java架构课,架构思路和设计是很重要的,但是另外一点,我希望能够带着大家用真正java架构师的角度去看待一些技术,而不是仅仅停留在技术的一些细节的点给大家从一些大数据的角度,去分析一下我们java架构领域中的一些技术,天下武功,都出自一脉,研究过各种大数据的系统,redis cluster讲解了很多原理,跟elasticsearch,很多底层的分布式原理都是类似的,redis AOF、fsync和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分布式的一些,hadoop、spark、storm里面很多核心的思想都是类似的;

后面,马上把redis架构给讲完之后就开始讲解业务系统的开发,包括高并发的商品详情页系统的大型的缓存架构,jedis cluster相关api去封装和测试对redis cluster的访问jedis cluster api,就可以自动针对多个master进行写入和读取;

## 实验不同master各自的slave读取、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;

实验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,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中执行了指令readonly后再去get才能在slave进行读取,redis cluster主从架构是相比读写分离复杂了点,也可以做,jedis客户端对redis cluster的读写分离的支持不太好,默认的话就是读和写都到master上去执行的,如果你要让最流行的jedis做redis cluster的读写分离的访问,那可能还得自己修改一点jedis的源码,成本比较高,要不然你就是自己基于jedis,封装一下,自己做一个redis cluster的读写分离的访问api;

核心的思路就是说,redis cluster的架构,就没有所谓的读写分离的概念了,读写分离,是为了什么呢,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,不会去跟大家直接讲解的,很多东西都要带着一些疑问、未知,实际经过一些实验和操作之后,让你体会的更加深刻一些redis cluster,不太好,server层面、jedis client层面对master做扩容,所以说扩容master,跟之前扩容slave效果是一样的;

## 实验自动故障切换、高可用性

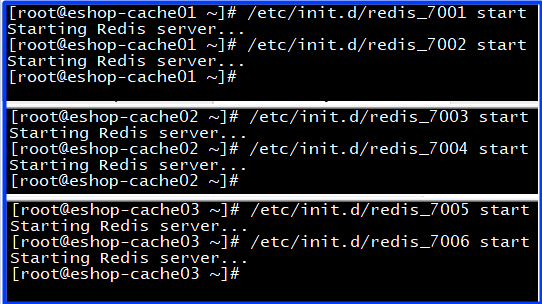
redis-trib.rb check 192.168.31.187:7001

比如把master1,187:7001杀掉,看看它对应的19:7004能不能自动切换成master,可以自动切换切换成master后的19:7004，可以直接读取数据;

再试着把187:7001给重新启动,恢复过来,自动作为slave挂载到了19:7004上面去;

## 操作步骤

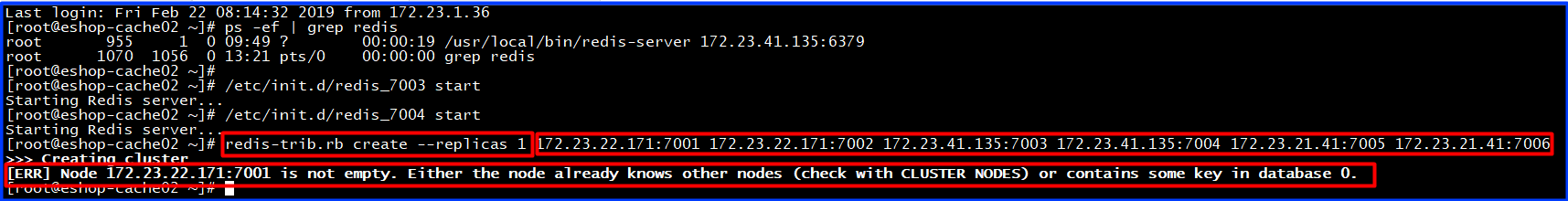
### 启动三台机器上的六个redis实例



### 创建6个redis实例的集群

redis-trib.rb create --replicas 1

172.23.22.171:7001 172.23.22.171:7002 172.23.41.135:7003 172.23.41.135:7004 172.23.21.41:7005 172.23.21.41:7006



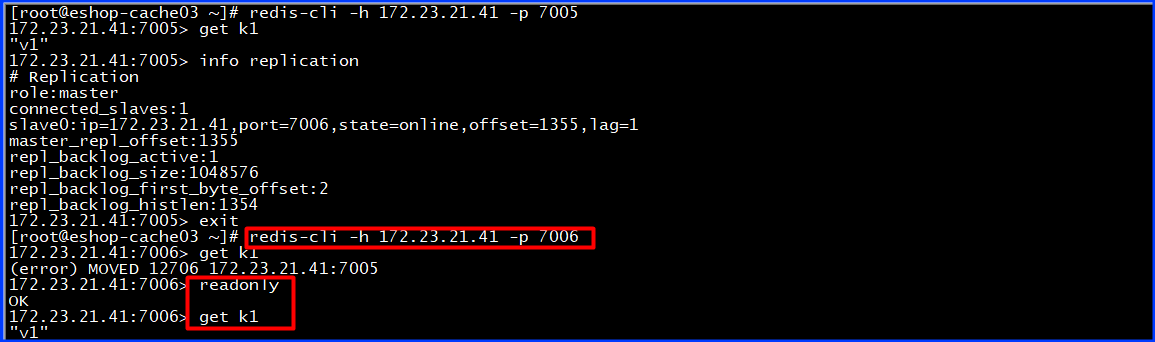


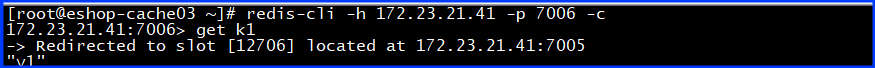
### 从节点中读、写数据

1.主节点上读、写数据



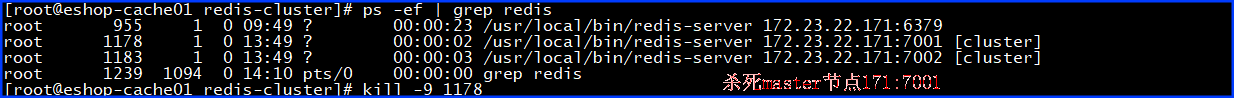
2.从节点上读数据



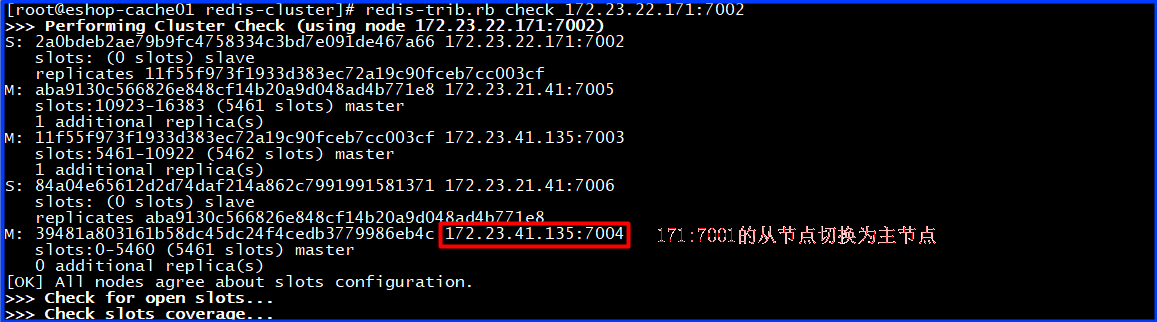


### 主备切换的演练

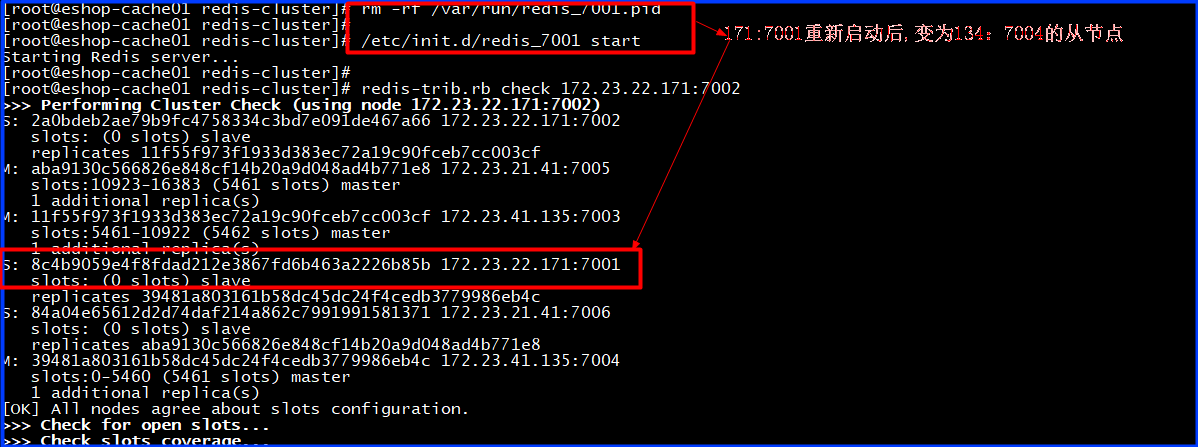
1.杀死171:7001



2.查看到171:7001的从节点升级为主节点



3.重新启动171:7001变为之前从节点的slave





# Redis Cluster通过master水平扩容来支撑更高的读写吞吐、海量数据

redis cluster模式下,不建议做物理的读写分离了,我们建议通过master的水平扩容,来横向扩展读写吞吐量、还有支撑更多的海量数据;redis单机,读吞吐5w/s,写吞吐2w/s,扩展redis更多master,那么如果有5台master,读吞吐可以达到总量25w/s QPS、写可以达到10w/s QPS,redis单机内存一般为6G~8G,fork类操作的时候很耗时,会导致请求延时的问题,扩容到5台master,能支撑的总的缓存数据量就是30G,40G,100台就到600G~800G,甚至1T+的海量数据,以下为redis是怎么扩容的方案;

## 加入新master

### 创建一个新的redis实例

1):搞一个/etc/redis/7007.conf;

2):再搞一个/etc/init.d/redis\_7007启动脚本;

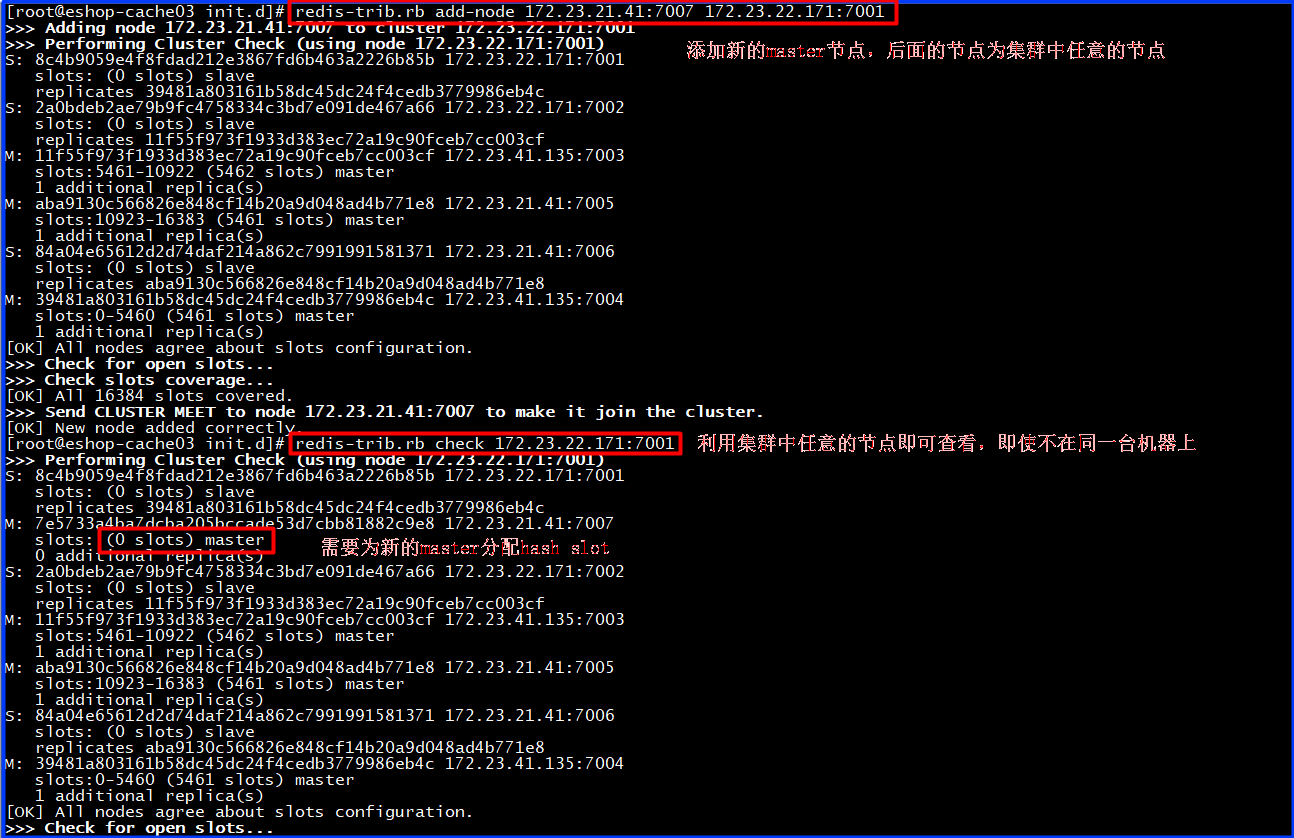
3):创建/var/redis/7007目录;

4):手动启动一个新的redis实例,在7007端口上;

### 将新的redis实例添加到cluster中

**redis-trib.rb add-node 172.23.21.41:7007 172.23.22.171:7001**

**redis-trib.rb check 172.23.22.171:7001**



### 给新增master分配hash slot

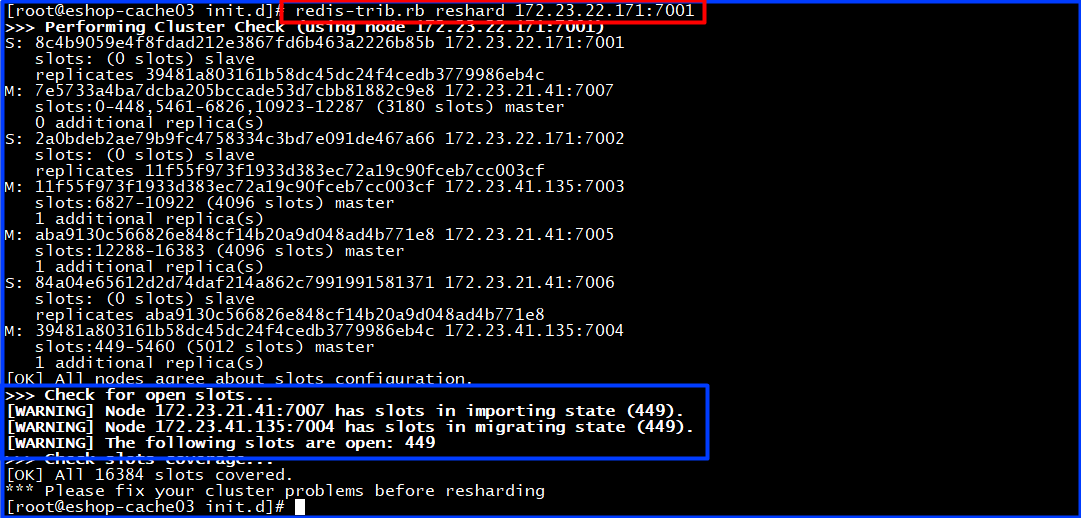
连接到新的redis实例上,确认自己是否加入了集群作为了一个新的master;

reshard一些数据过去,resharding的意思就是把一部分hash slot从一些node上迁移到另外一些node上

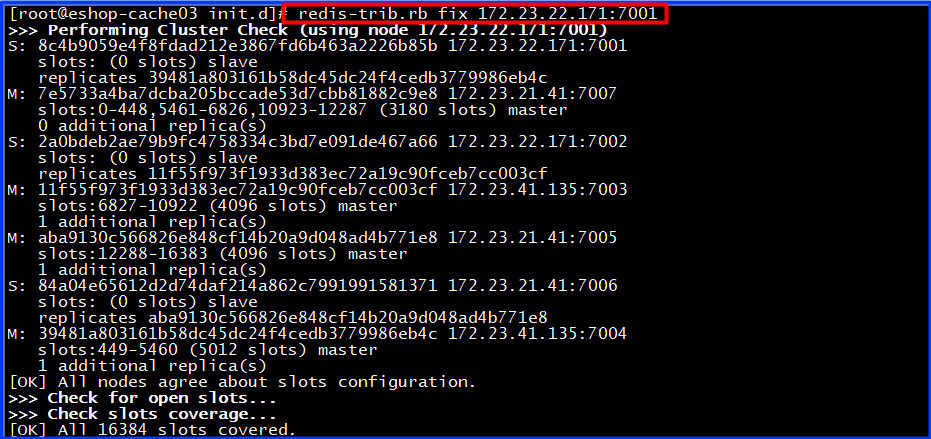
**redis-trib.rb reshard 172.23.22.171:7001**

要把之前3个master上,总共4096个hashslot迁移到新的第四个master上去

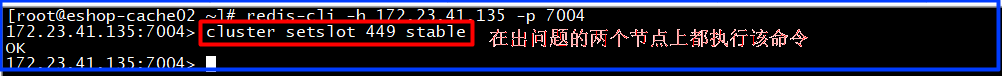


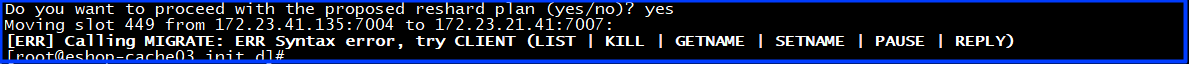


1.方法一：fix

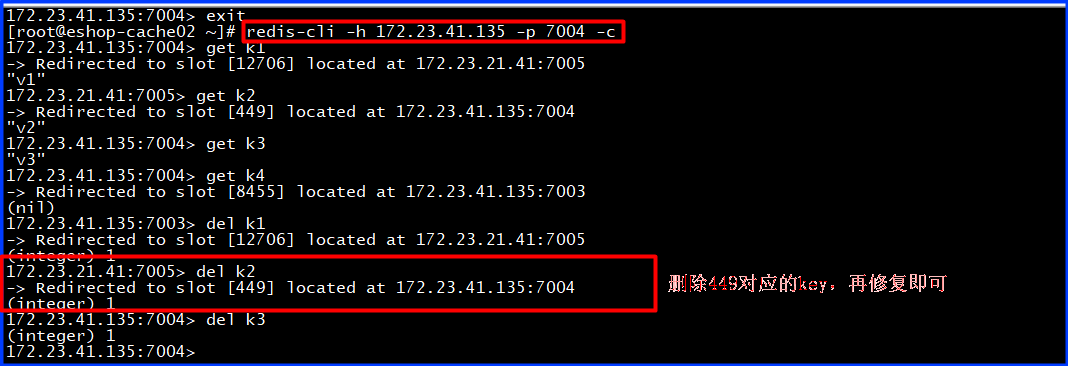


2.方法二：cluster setslot 449 stable





3.方法三：删除449 hash slot对应的key



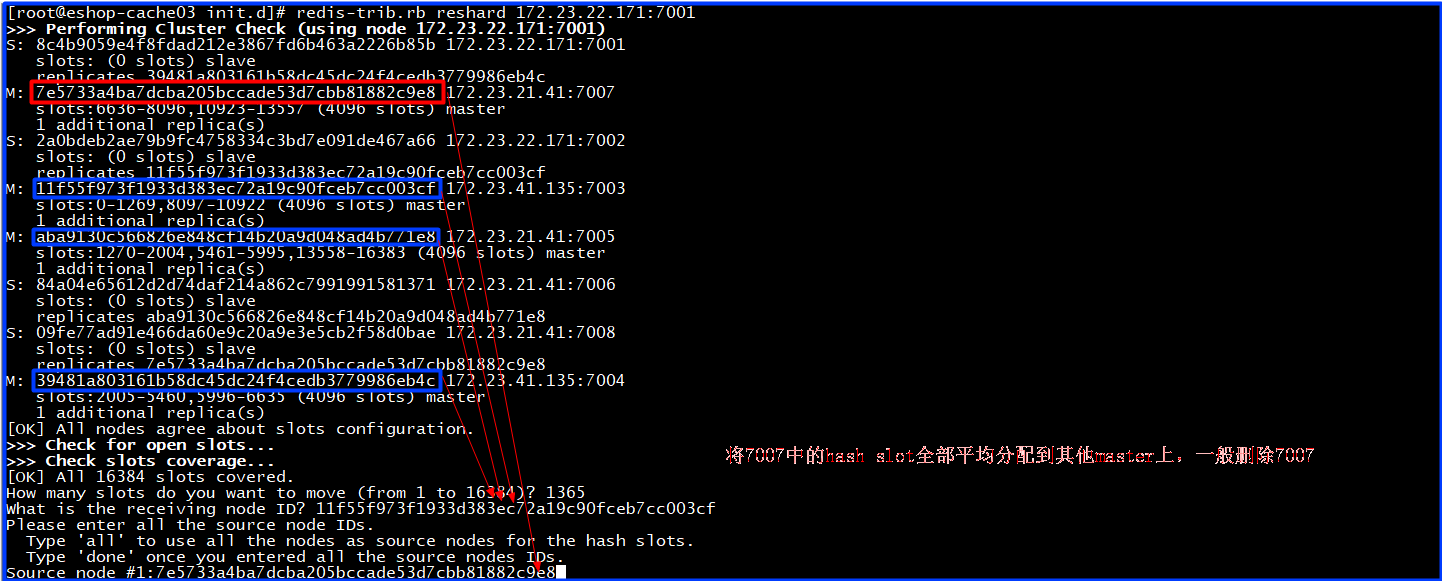
## 添加node作为slave

创建一个新的redis实例,再执行如下新增slave指令

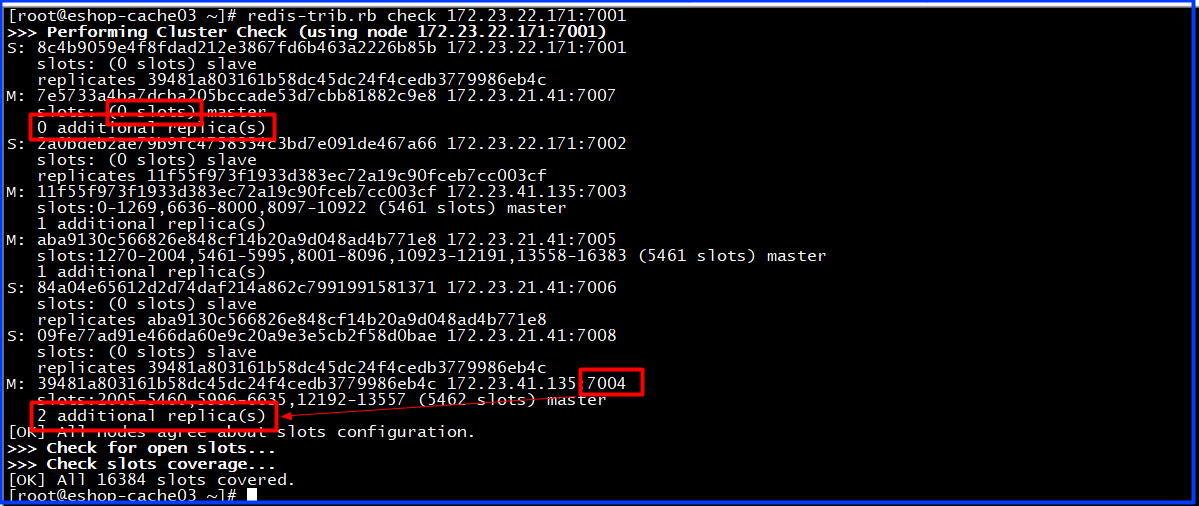
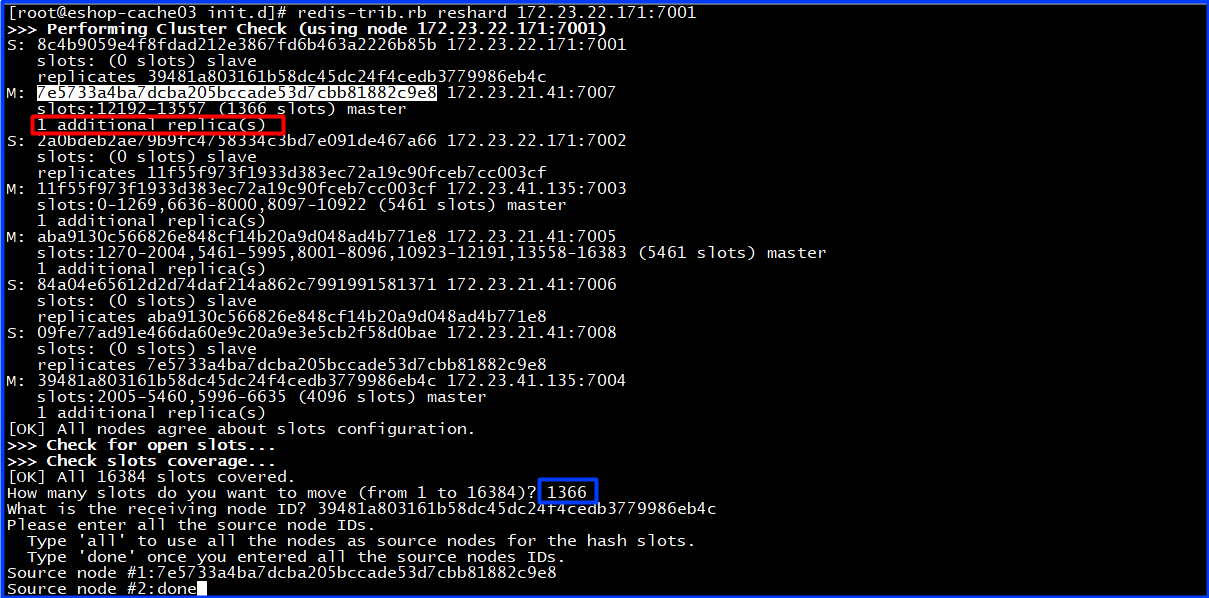


## 删除redis-cluster中的master

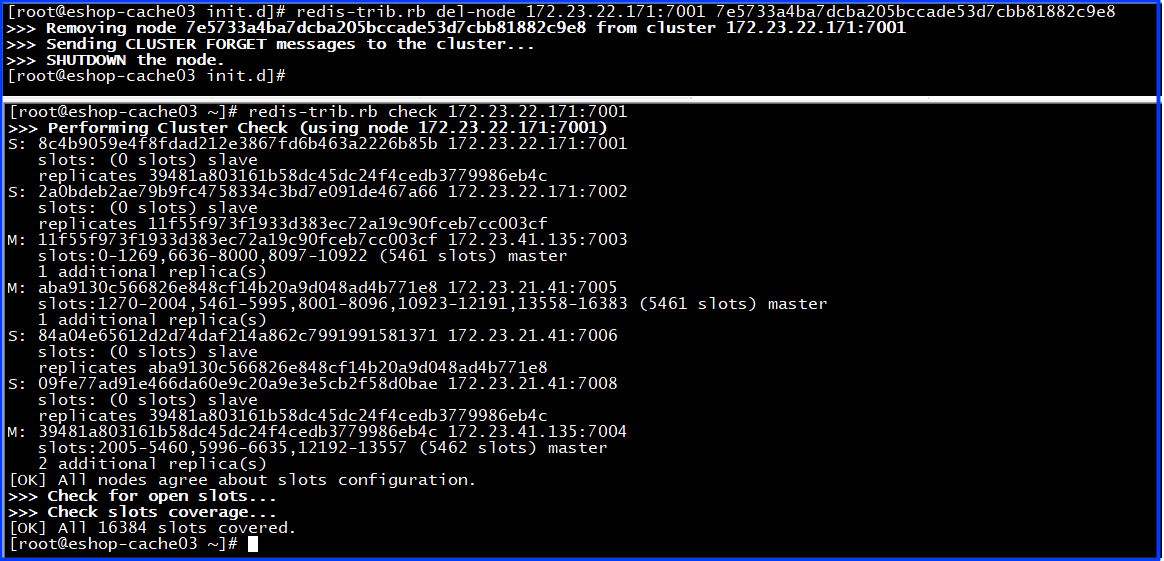
### 移除master中的hash slot



### hash slot清空了之后slave会转移到其他master中



### 删除master节点



## Redis cluster自动化冗余slave的迁移、实现更强的高可用架构部署方案

现在有10个master,每个master有1个slave,此时新增了3个slave作为冗余,有的master就有2个slave了,有的master出现了salve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;

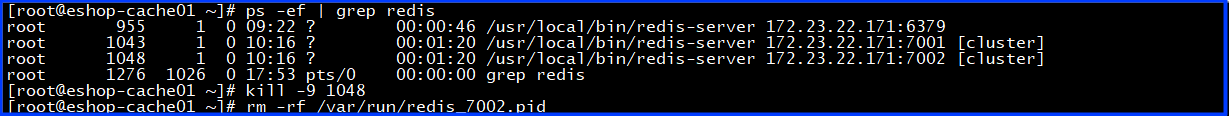
为了避免的场景就是,如果你每个master只有一个slave,万一说一个slave死了,然后很快master也死了,那可用性还是降低了,但是如果你给整个集群挂载了一些冗余slave,那么某个master的slave死了,冗余的slave会被自动迁移过去、作为master的新slave,此时即使那个master也死了,还是有一个slave会切换成master的;

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;

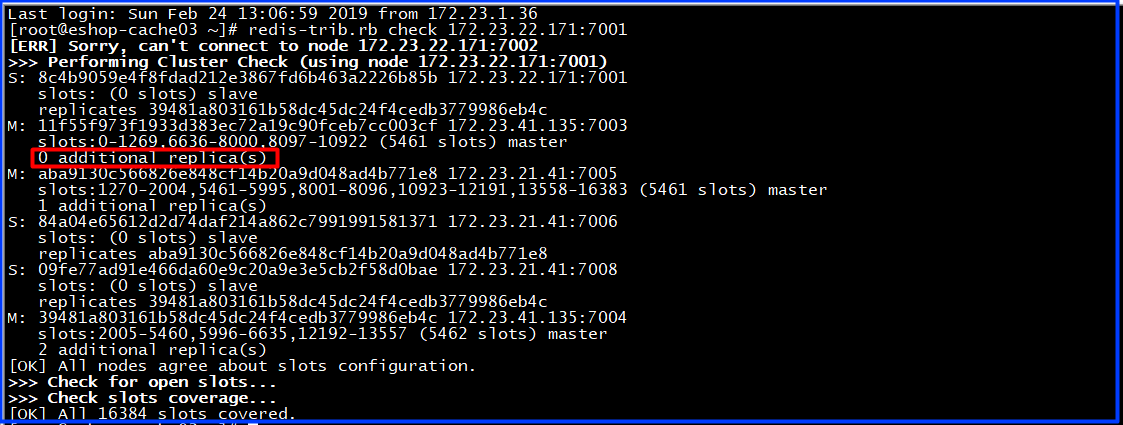
1.停止slave之前状态;



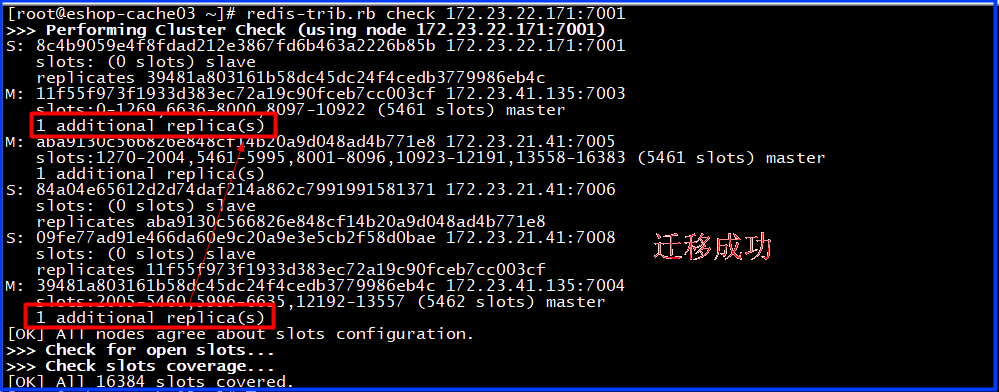
2.杀死slave节点



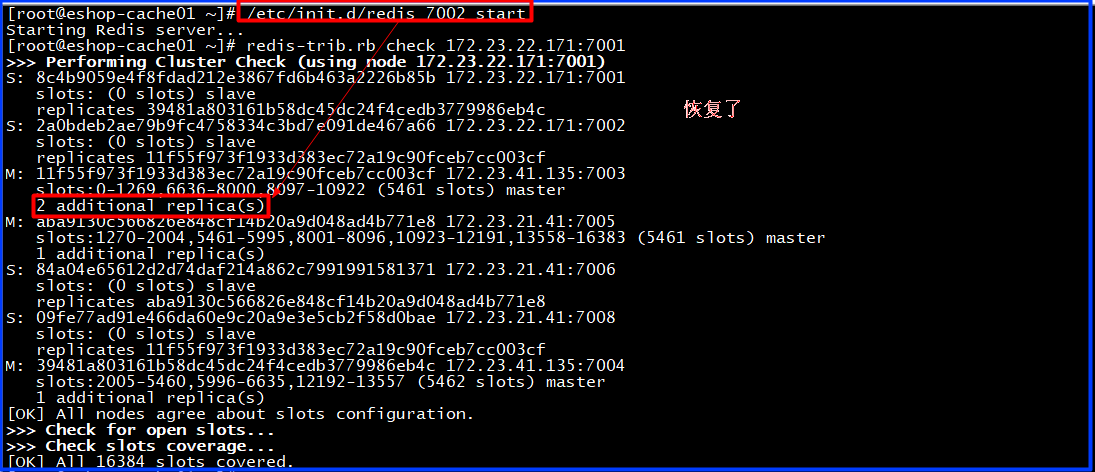
3.杀死后瞬间状态



4.杀死后几秒后

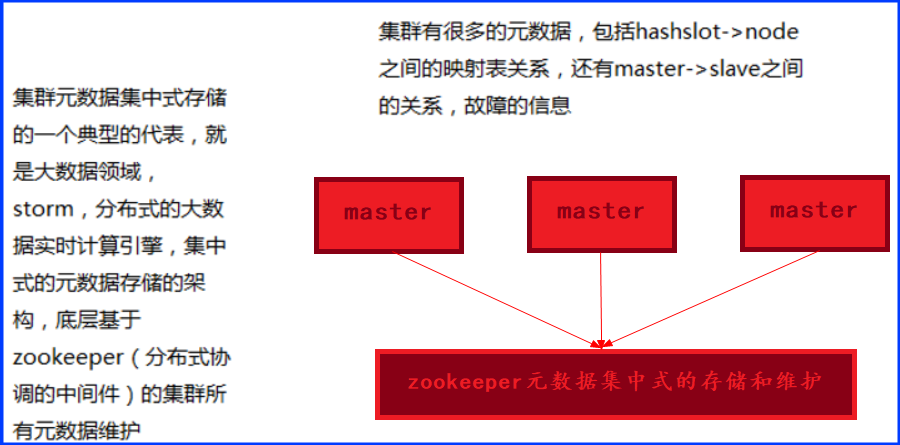


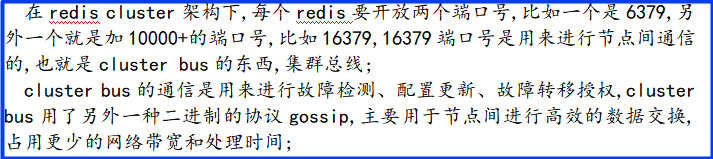
1. 重新启动杀死的slave



# Redis-cluster核心原理分析:gossip通信、smart jedis、redis cluster主备切换

## 节点间的内部通信机制





### 基础通信原理

1.redis cluster节点间采取gossip协议进行通信

gossip协议通信,跟集中式不同,不是将集群元数据(hashslot到master的映射、master到slave的映射、节点信息、故障等等)集中存储在某个节点上,而是集群节点互相之间不断通信,保持整个集群所有节点的数据是完整的;

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;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,同时其他几个节点接收到ping之后返回pong;

3.交换的信息

节点的增加和移除,hash slot、故障信息等等;

### gossip协议

gossip协议包含多种消息,包括meet、ping、pong、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;其实内部就是发送了一个gossip meet 消息,给新加入的节点,通知哪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;

### 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;

## 面向集群的jedis内部实现原理

jedis是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;

### 基于重定向的客户端

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,可以去查看一个key对应的hash slot是什么:

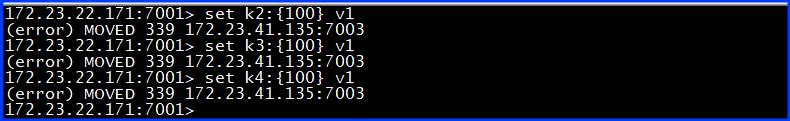


用redis-cli的时候;可以加入-c参数,支持自动的请求重定向,redis-cli接收到moved之后,会自动重定向到对应的节点执行命令;

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;

### 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jedis,都是smart的;本地维护一份hashslot->node的映射表缓存,而大部分情况下,直接走本地缓存就可以找到hashslot -> node的关系,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都会在本地计算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;

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 JedisClusterMaxRedirectionException 异常;

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;

## 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

### 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;

### 从节点选举

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

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;

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,给进行选举的slave进行投票,此时如果大部分的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;

### 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;

# redis在实践中的一些常见问题以及优化思路(包含linux内核参数优化)

## 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1个G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的内存在10GB以内,如slave -> master全量复制等

耗时操作;

## AOF的阻塞问题

redis将数据写入AOF缓冲区,单独开一个现场做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,用SSD将大幅度提升磁盘读写的速度;

## 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;

## 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;

## 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

## swapiness

cat /proc/version，查看linux内核版本

如果linux内核版本<3.5，那么swapiness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swapiness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

## 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

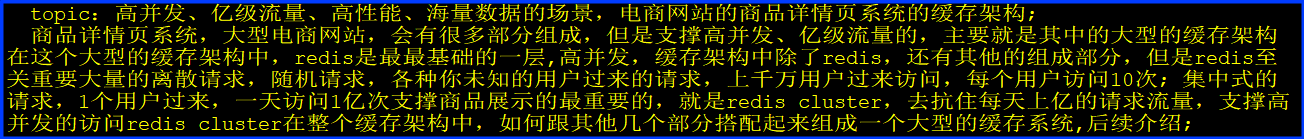
## tcp backlog

cat /proc/sys/net/core/somaxconn

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

# redis阶段性总结：1T以上海量数据+10万以上QPS高并发+99.99%高可用

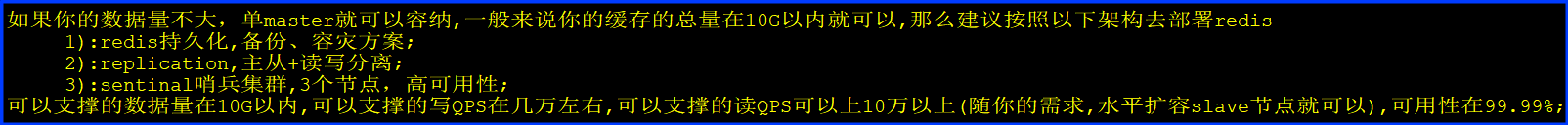
## 讲解redis是为了什么？



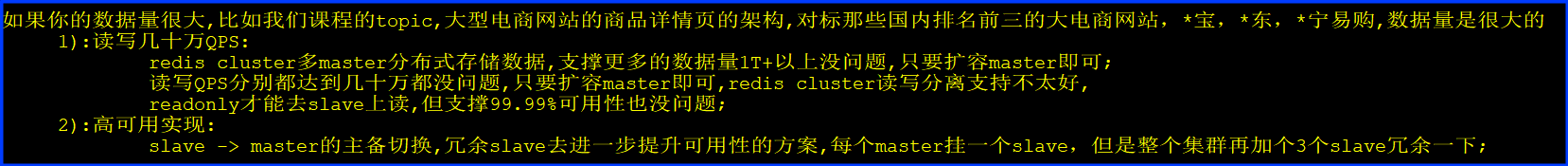
## 讲解的redis可以实现什么效果？



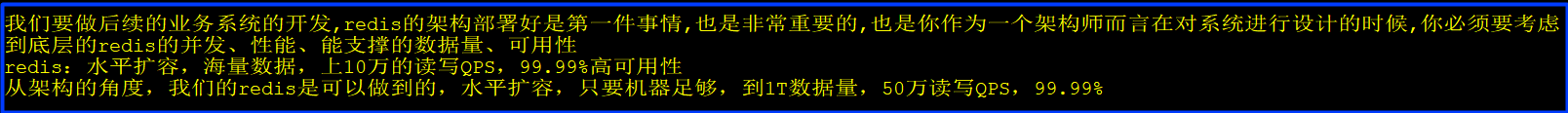
## redis的第一套企业级的架构



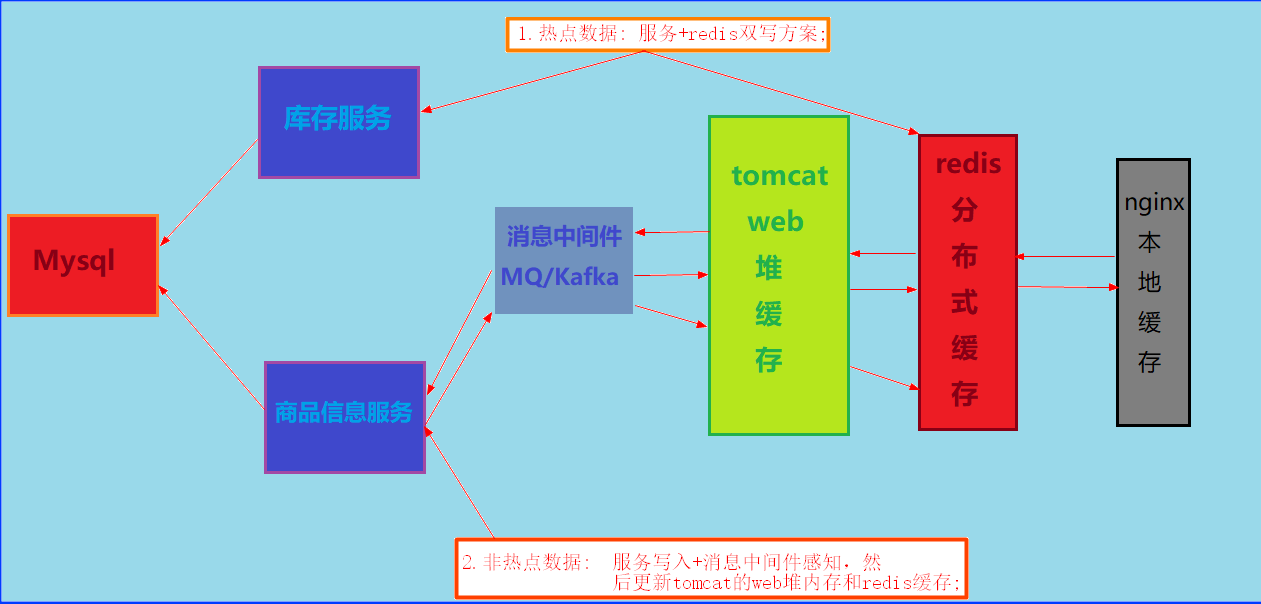
## redis的第二套企业级架构



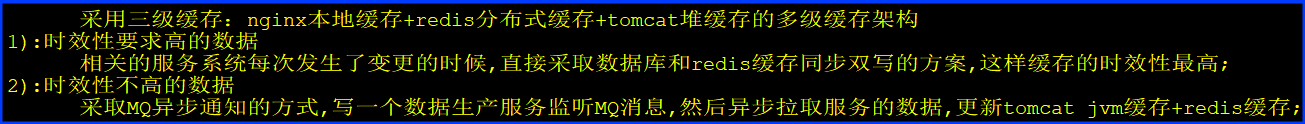
## 我们现在课程讲解的项目进展到哪里了？



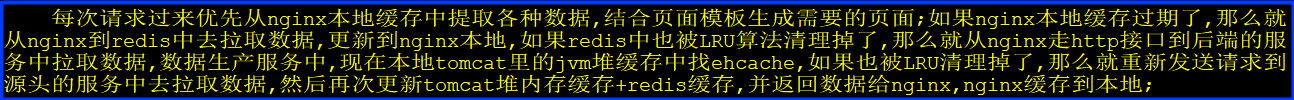
# 亿级流量商品详情页的多级缓存架构以及架构中每一层的意义



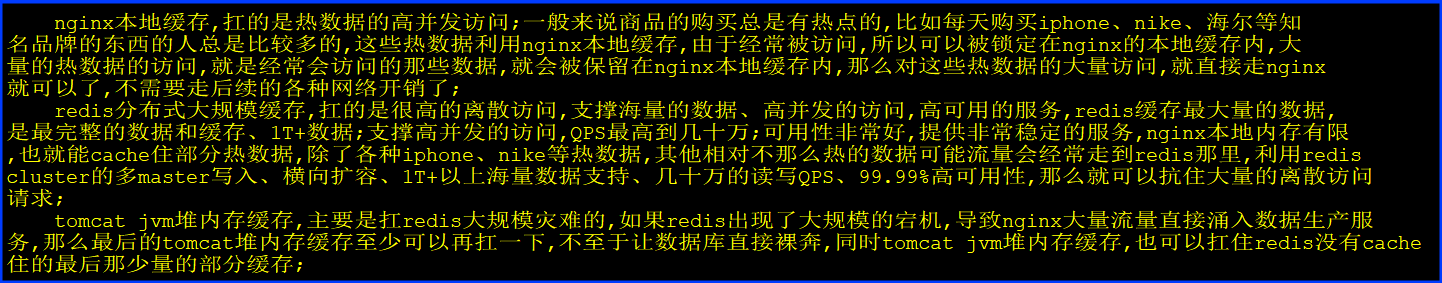
## 上亿流量的商品详情页系统的多级缓存架构



## nginx+lua脚本做页面动态生成的工作



## 多级缓存架构中每一层的意义



# Cache Aside Pattern与数据库读写模式的分析

## 什么是cache aside pattern

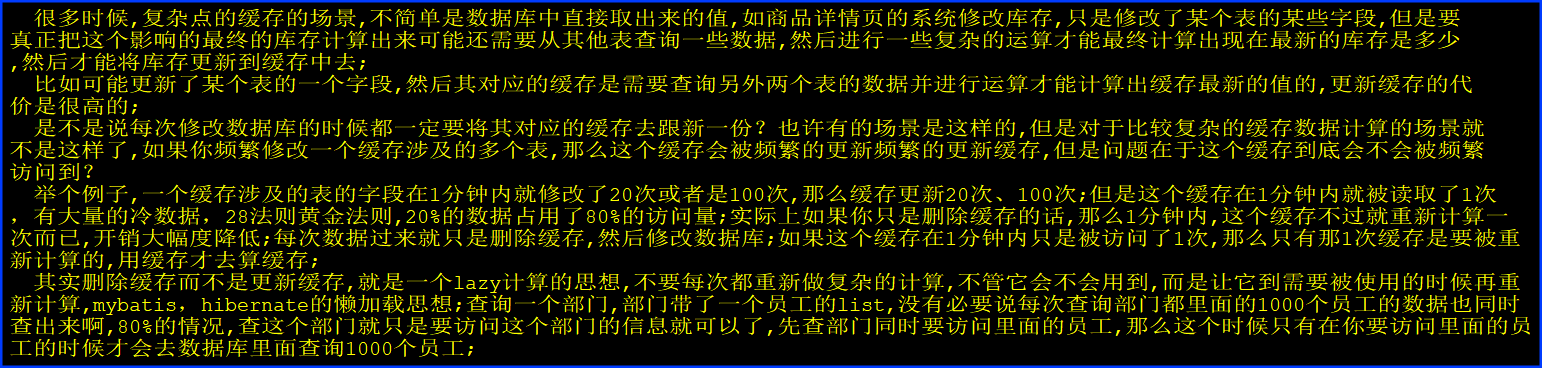
1.读时

先读缓存,缓存没有的话那么就读数据库,然后取出数据后放入缓存,同时返回响应;

2.更新时

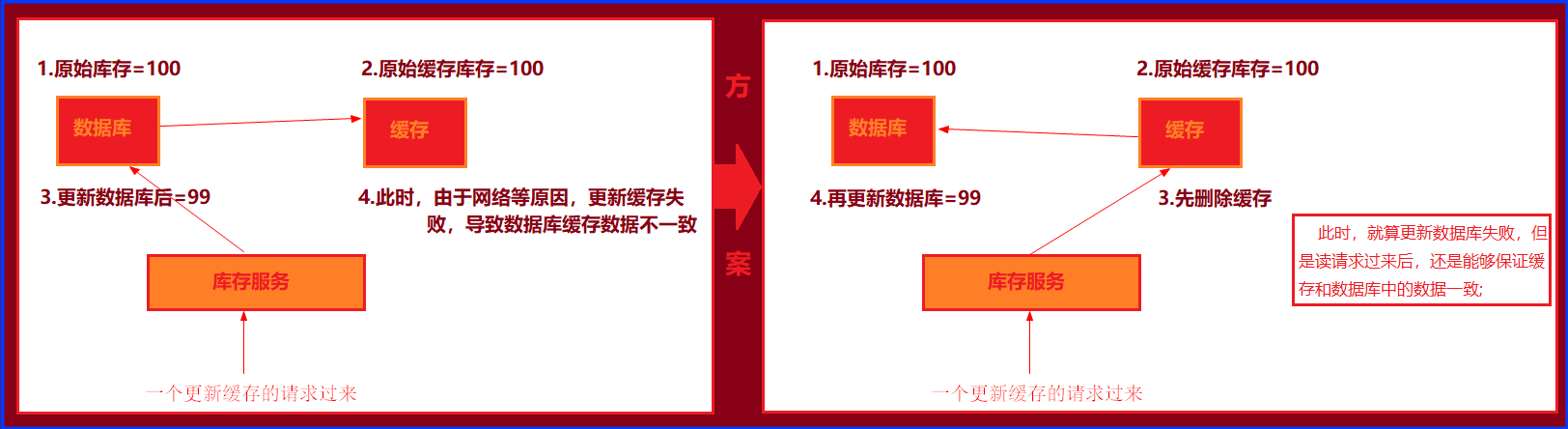
先删除缓存,然后再更新数据库;

## 更新的时候,为什么是删除缓存,而不是更新缓存呢？



# 高并发场景下的缓存、数据库双写不一致问题分析与解决方案设计

## 最初级的数据库缓存数据不一致问题与解决方案



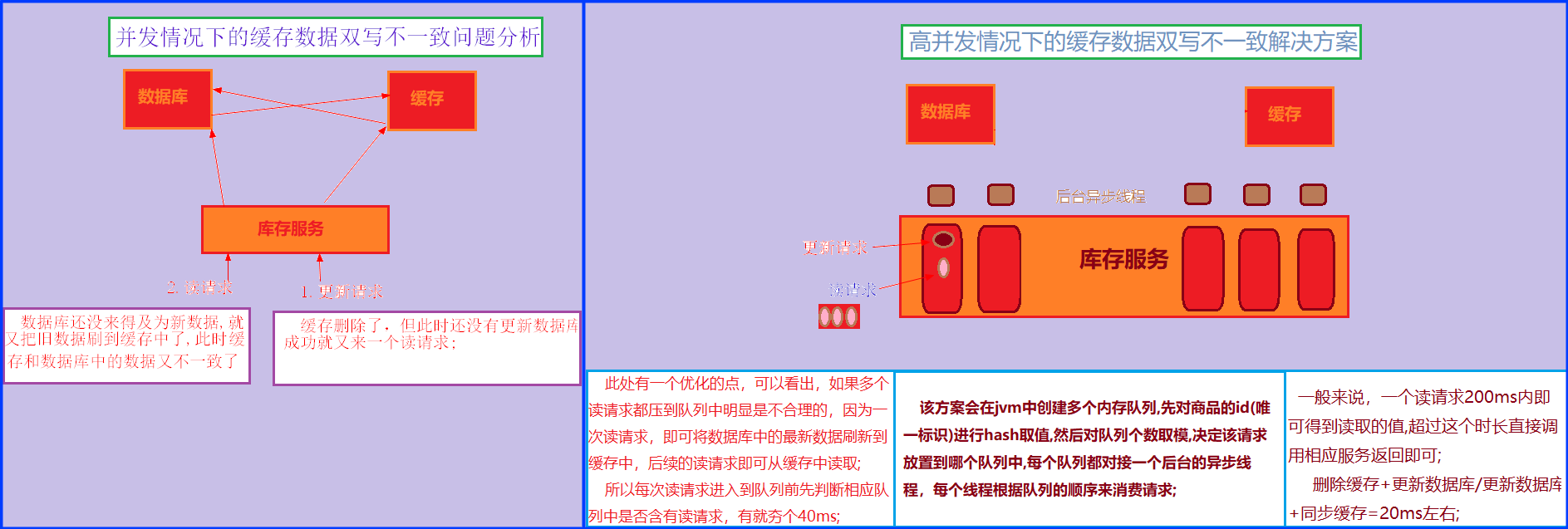
### 问题分析

先修改数据库再删除缓存,此时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,数据缓存出现数据不一致;

### 解决方案

换个顺序,先删除缓存再修改数据库,如果删除缓存成功了而修改数据库失败了,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会更新到缓存中;

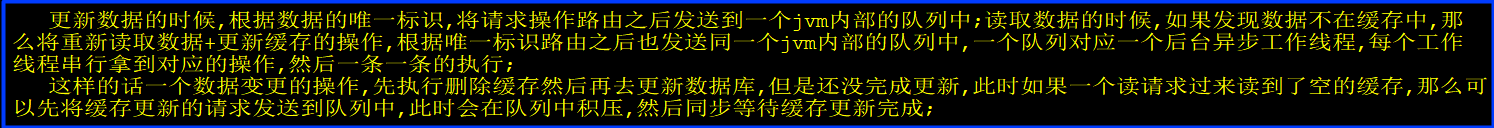
## 比较复杂的数据库缓存数据不一致问题分析



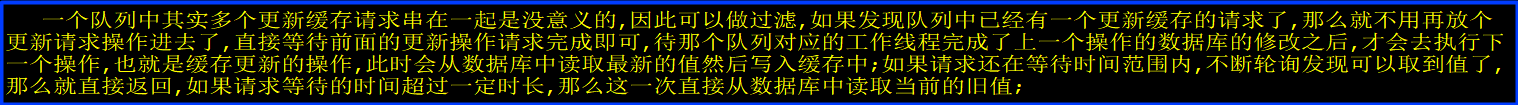
### 问题分析

沿用初级问题的解决方案,此时数据发生了变更,先删除了缓存,然后要去修改数据库,此时还没修改数据库成功;一个读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并放到了缓存中,而第一个数据变更的变更程序完成了数据库的修改,此时数据库和缓存中的数据又不一样了;

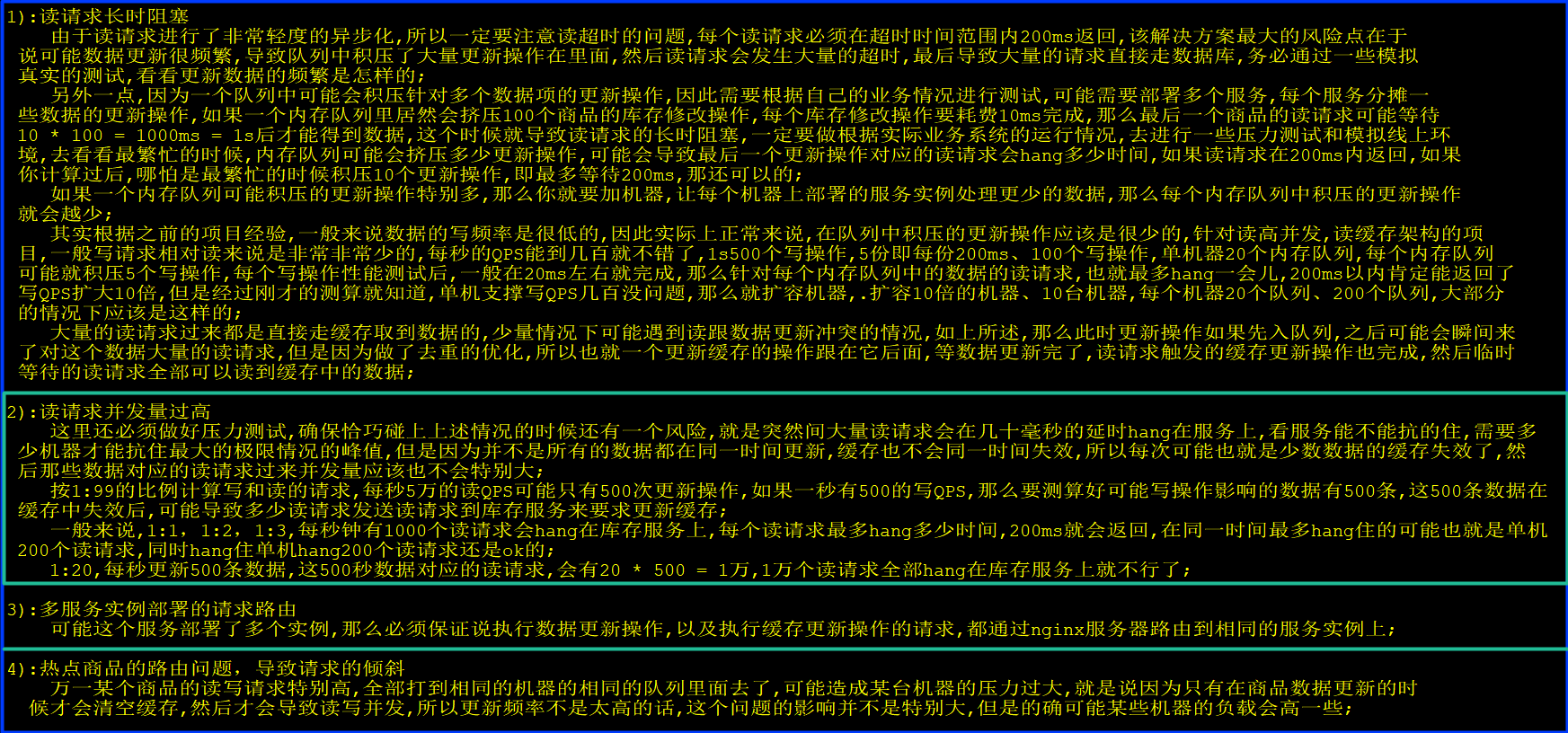
### 解决方案-数据库与缓存更新与读取操作进行异步串行化



### 方案优化点



### 高并发的场景下,该解决方案要注意的问题



# Eshop-库存项目的初始化工作

## 通过yum安装mysql数据库的

yum install -y mysql-server//安装数据库5.1.73

service mysqld start//启动mysql服务

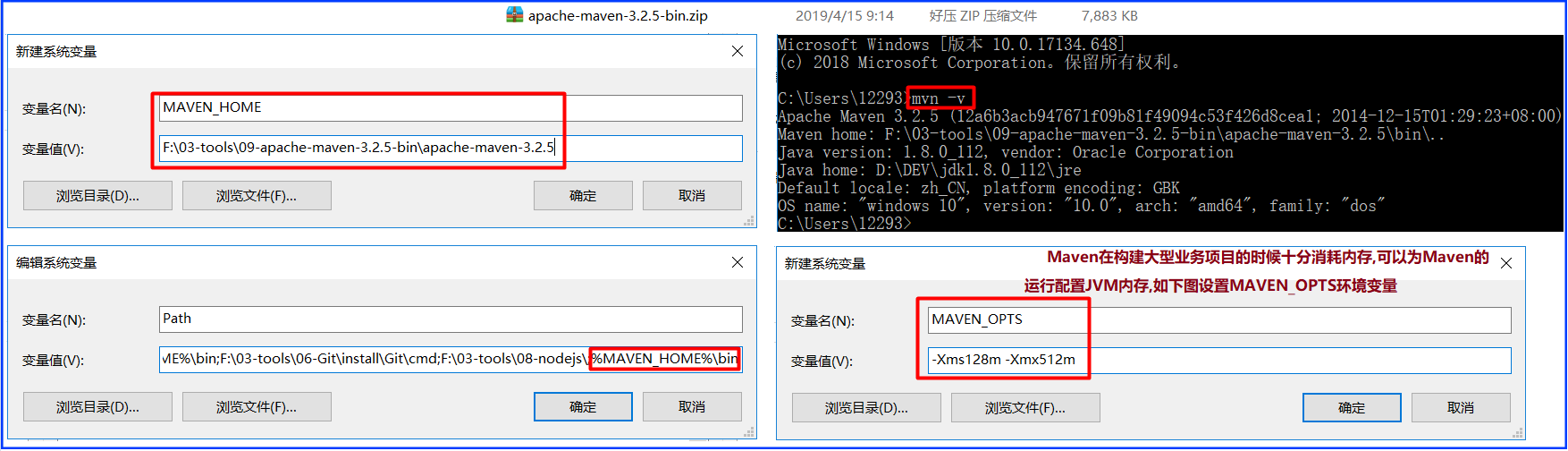
chkconfig mysqld on//设置开机时自动启动

yum install -y mysql-connector-java//安装mysql客户端

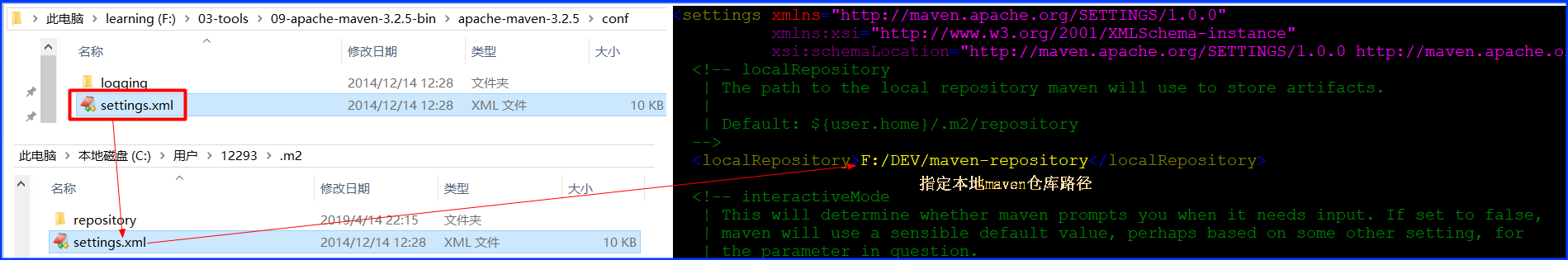


## Maven的安装和配置

### 下载并配置maven的环境变量



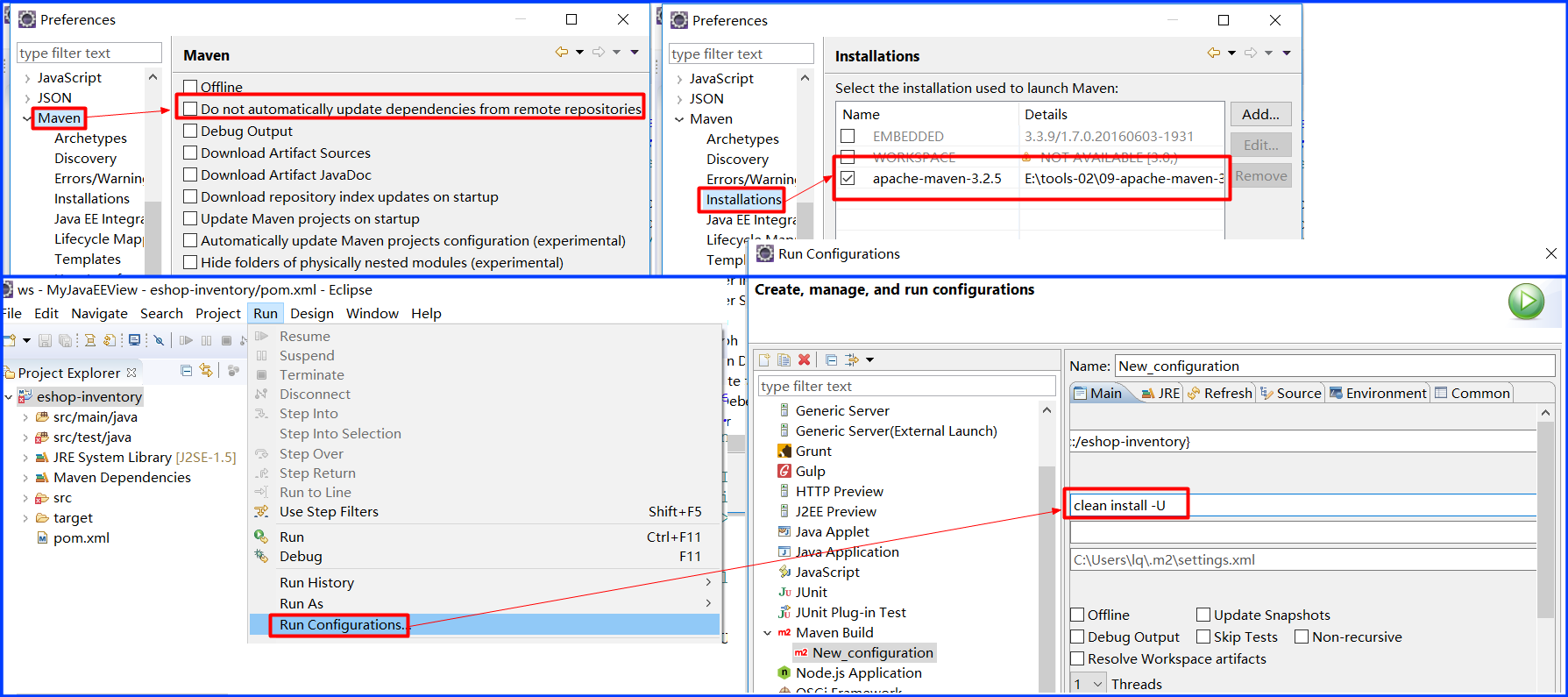
### 设置maven的本地仓库



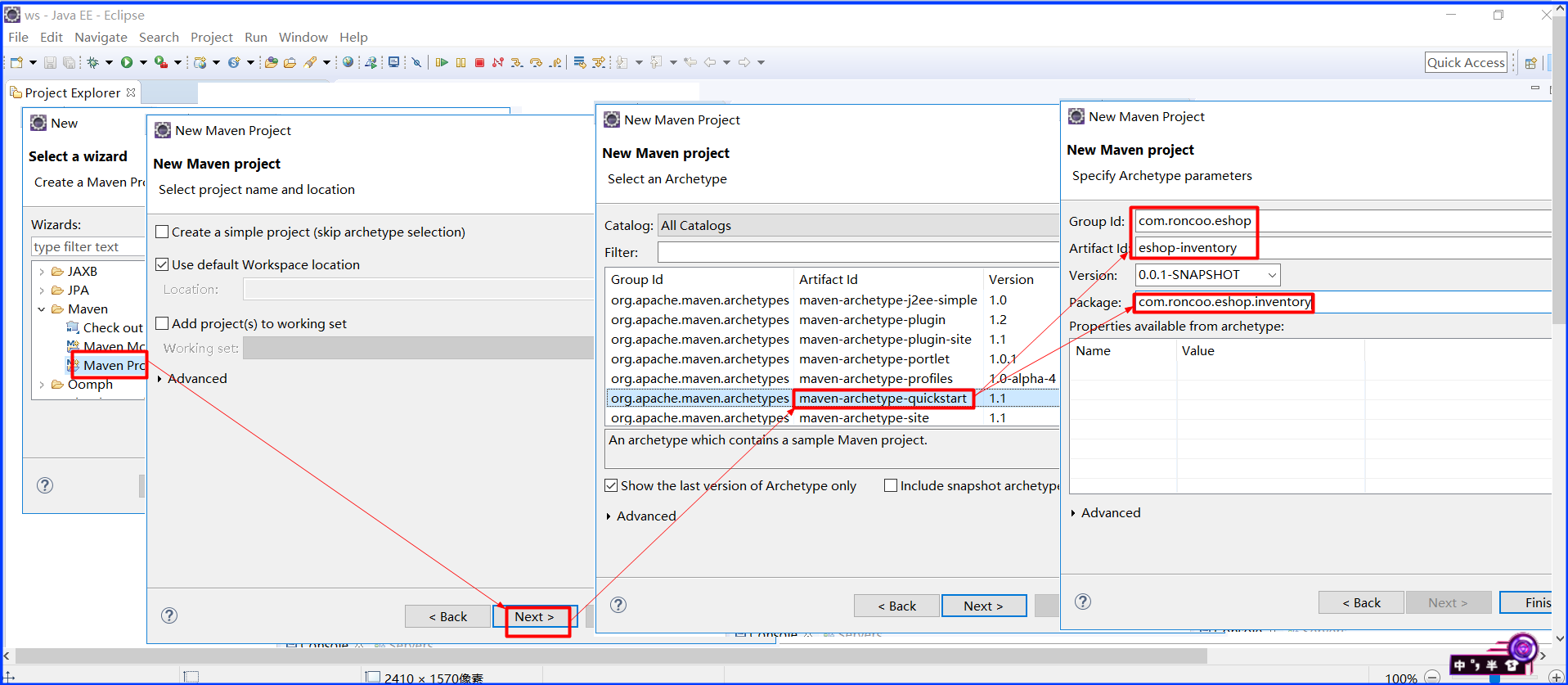
### Eclipse中关联maven



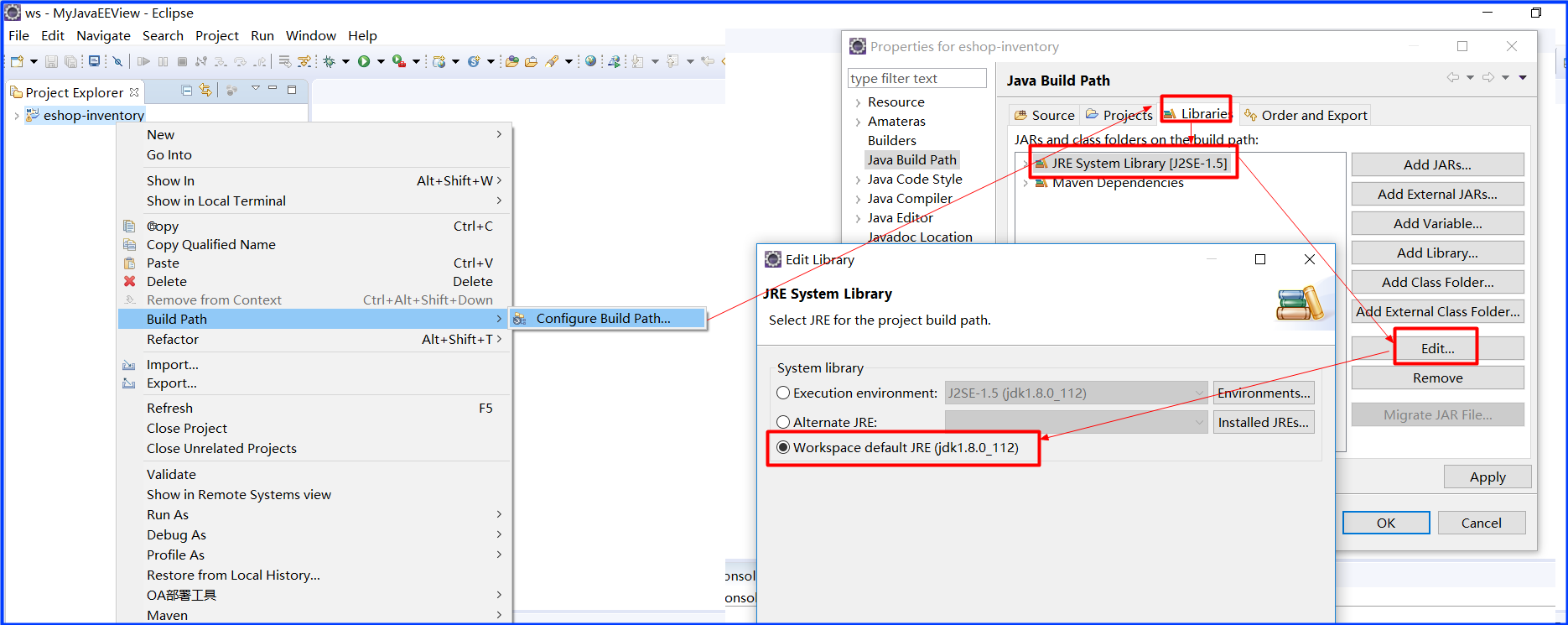
### Maven配置Eclipse自动下载



## Eshop-库存项目的maven项目创建



1. jdk修改为默认的1.8



1. j