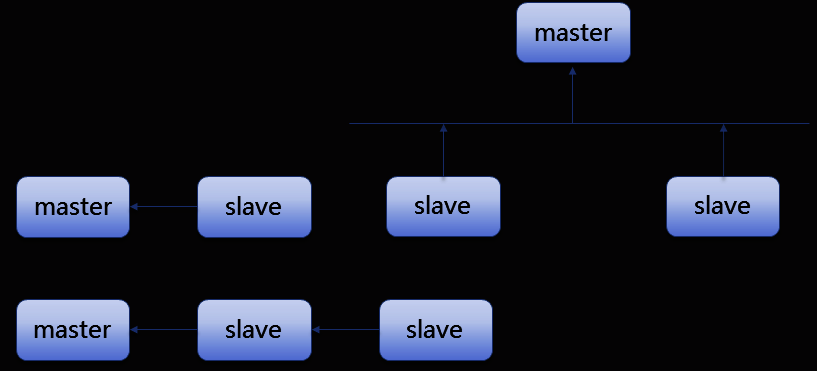
NOSQL-DAY03

# Redis主从复制

## 主从复制概述

### 结构模式：一主一从、一主多从、主从从



### 工作原理：

slave向master发送sync命令

master启动后台存盘进程，并收集所有修改数据命令

master完成后台存盘后，传送整个数据文件到slave

salve接受数据文件，加载到内存中完成首次完全同步

后续有新数据产成时，master继续将新的数据收集到的修改命令以次传给slave，完成同步

### 主从复制缺点：

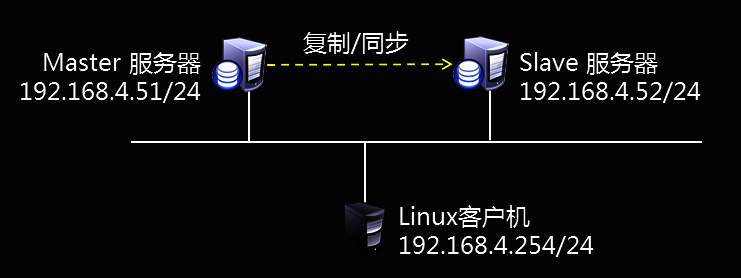
网络繁忙，会产生数据同步延时问题

系统繁忙，会产生数据同步延时长

## 配置主从复制

### 一主一从拓扑结构：

以主服务器为基准，将数据自动同步到从服务器



主从同步配置命令

info replication

slaveof master\_ip master\_port

slaveof no one

查看当前主机的主从库角色：

192.168.4.51:6351> info replication //查看主从配置信息

role:master //每一台主机启动redis服务都是默认主库，因此主库无须配置

connected\_slaves:0 //当前连接从库为0

配置从库：

手动设为从库：

SLAVEOF 主库IP地址 端口号

[root@mysql52 ~]# redis-cli -h 192.168.4.52 -p 6352

192.168.4.52:6352> slaveof 192.168.4.51 6351

192.168.4.52:6352> info replication

role:slave

### 反客为主

主机宕机后，手动将从库设置为主库

192.168.4.52:6352> slaveof no one //手动设为主库

以上配置都是命令行临时配置，重起服务失效;永久配置执行以下操作：

[root@mysql52 ~]# vim /etc/redis/6352.conf

283 slaveof 192.168.4.51 6351 //主库IP与端口

[root@mysql52 ~]# redis-cli -h 192.168.4.52 -p 6352 shutdown

[root@mysql52 ~]# /etc/init.d/redis\_6352 start

[root@mysql52 ~]# redis-cli -h 192.168.4.52 -p 6352

192.168.4.52:6352> info replication //从库信息还在

**配置带验证的主从同步：**

主库51：

[root@mysql51 ~]# vim /etc/init.d/redis\_6351

501 requirepass 123456 //指定验证密码

[root@mysql51 ~]# vim /etc/init.d/redis\_6351

43 $CLIEXEC -h 192.168.4.51 -p $REDISPORT -a 123456 shutdown

[root@mysql51 ~]# /etc/init.d/redis\_6351 start

从库52：

[root@mysql52 ~]# vim /etc/redis/6352.conf

291 masterauth 123456 //主库密码

[root@mysql52 ~]# /etc/init.d/redis\_6352 start

**主库存入数据，从库自动同步**

**一主多从配置类似于一主一从配置，在将要添加为从库的主机配置文件指出主库IP和端口**

**主从从配置**：51（主）——>52（从—主）——>53（从）

从库1：

[root@mysql52 ~]# vim /etc/redis/6352.conf

slaveof 192.168.4.51 6351 //指定51为自己的主库

masterauth 123456 //主库的连接密码

requirepass 123456 //指定验证密码

[root@mysql52 ~]# /etc/init.d/redis\_6352 start

从库2：作为52的从库

[root@mysql53 ~]# vim /etc/redis/6353.conf

slaveof 192.168.4.52 6352 //设置52为自己的主库

masterauth 123456 //设置连接主库的密码

[root@mysql53 ~]# /etc/init.d/redis\_6353 restart

192.168.4.53:6353> config set requirepass 123456 //设置本机连机密码

192.168.4.53:6353> auth 123456

192.168.4.53:6353> config set masterauth 123456 //设置连接主库的密码

### 哨兵模式

主库宕机后，从库自动升级为主库

在slave主机编辑sentinel.conf文件

在slave主机运行哨兵程序

sentinel monitor 主机名 ip地址 端口 票数

主机名：自定义

IP地址：master主机的IP地址

端口：master主机redia服务使用的端口

票数：有几台哨兵主及连接不上主库时，切换主库

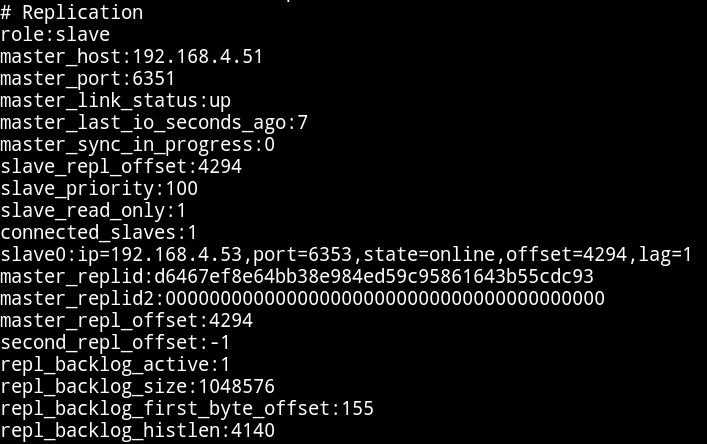
[root@mysql52 ~]# vim /etc/sentinel.conf

sentinel monitor redis51 192.168.4.51 6351 1

sentinel auth-pass redis51 123456 //连接主库密码

[root@mysql52 ~]# redis-cli -h 192.168.4.52 -p 6352 -a 123456

192.168.4.52:6352> info replication

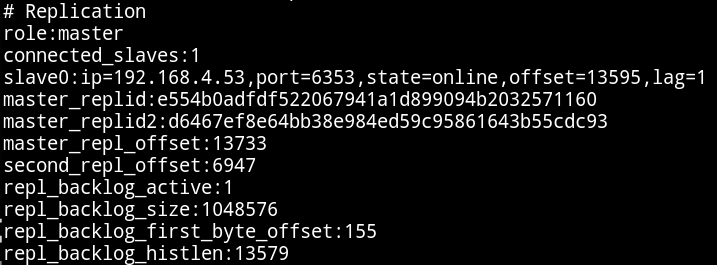


[root@mysql52 ~]# redis-sentinel /etc/sentinel.conf //启动哨兵

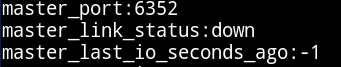
当前主库宕掉，会自动监控并切换主库

[root@mysql52 ~]# redis-cli -h 192.168.4.52 -p 6352 -a 123456

192.168.4.52:6352> info replication



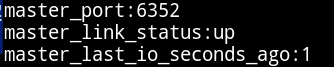
重起51主机redis服务，会自动成为当前主库52的从库



若出现上图信息，则为51主机添加密码：

[root@mysql51 ~]# vim /etc/redis/6351.conf

masterauth 123456

查看状态是否成功

恢复主机51为主

关闭52主机上的哨兵服务

将51主机配置文件中的slaveof 192.168.4.52 6352行去掉

## Redis持久化RDB/AOF

### RDB持久化：

**RDB介绍**：

**redis数据库文件**，全称Redis DataBase

— 数据持久化方式之一

— 按照指定时间间隔，将内存中的数据集快照写入硬盘

— 术语叫Snapshot快照

— 恢复时，将快照文件直接读入内容

**相关配置参数**

文件名

dbfilename “dump.rdb” //文件名

save “” //禁用RDB

数据从内存保存到硬盘的频率

save 900 1 //900秒内且有一次修改

save 300 10 //300秒内且有10次修改

save 60 10000 //60秒内且有10000修改

手动立刻存盘

save //阻塞写存盘

bgsave //不阻塞写存盘

压缩

rdbcompression yes | no

在存储快照后，使用crc16算法做数据校验

rdbchecksum yes | no

bgsave出错时停止写操作

stop-write-on-bgsave-error yes | no

**使用RDB文件恢复数据**

**备份数据**

备份dump.rdb文件到其他位置

**恢复数据**

拷贝备份文件到数据库目录，重起redis服务

**RDB优缺点**

优点：

— 高性能的持久化实现——创建一个子进程来执行持久化，先将数据写入临时文件，持久化过程结束后，再用这个临时文件替换上次持久化好的文件；过程中主进程不做任何IO操作

— 比较适合大规模数据恢复，且对数据完整性要求不是非常高的场合

缺点：

意外宕机时，最后一次持久化的数据会丢失

### AOF持久化

**AOF介绍**

只做追加操作的文件，Append Only File

— 记录redis服务所有写操作

— 不断的将新的写操作，追加到文件的末尾

— 使用cat命令可以查看文件内容

**相关配置参数**：

— appendfilename “appendonly.aof” //指定文件名

— appendonly yes //启用aof，默认no

— appendfsync always //有新写操作立即记录

— appendfsync everysec //每秒记录一次

— appendfsync no //从不记录

**开启AOF持久化功能：**

[root@mysql51 ~]# vim /etc/redis/6351.conf

appendonly yes

[root@mysql51 ~]# /etc/init.d/redis\_6351 restart

**同时启动RDB与AOF，在启动redis服务时，优先加载AOF文件；如果在启动RDB并且存入数据之后再启动AOF，重起之后会丢失数据，因为启动服务加载的是AOF文件不是RDB文件**

**日志文件会不断增大，何时触发日志重写：**

redis会记录上次重写时AOF文件的大小

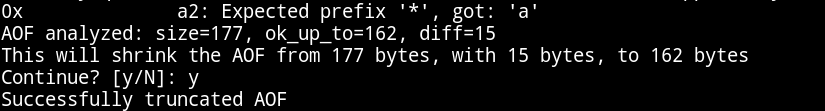
默认配置宕aof文件是上次rewrite后大小的1倍且文件大于64M时触发

**修复AOF文件**

把文件恢复到最后一次的正确操作：

如果误操作AOF文件，导致文件错误，可使用修复工具修复

# redis-check-aof --fix /var/lib/redis/6351/appendonly.aof



[root@mysql51 ~]# rm -rf /var/run/redis\_6351.pid

[root@mysql51 ~]# /etc/init.d/redis\_6351 start

[root@mysql51 ~]# ss -natpul | grep redis

**使用AOF文件恢复数据**

**备份数据**

备份appendonly.aof文件到其他位置

# cp /var/lib/redis/6351/appendonly.aof /root

**恢复数据**

拷贝备份文件到数据库目录，重起redis服务

# cp /root/appendonly.aof /var/lib/redis/6351/appendonly.aof

# /etc/init.d/redis\_6351 start

**AOF优缺点**

**优点：**

— 可以灵活设置持久化方式，同步持久化appendfsync always 或 异步持久化appendfsync everysec

— 出现意外宕机时，尽可能丢失1秒的数据

**缺点：**

— 持久化文件的体积通常会大于RDB方式

— 执行fsync策略时的速度可能会比RDB方式慢

## String子赋串

### 字符串操作

**set key value [EX seconds] [PX milliseconds] [NX|XX]**

— 设置key及值，过期时间可以使用秒或毫秒为单位

> set a 12 nx //变量a存在不赋值，不存在则赋值

> set b 12 xx //变量b存在赋值，不存在不赋值

**setrange key offset valus**

— 从偏移量开始赋写key的特定位的值

> set as 10000

> setrange as 2 xx

> get as

"10xx0"

**strlen key**，统计字串长度

> strlen as

(integer) 5

**append key value**

— 存在则追加，不存在则创建key及value，返回key长度

> set name pan //新建变量name，赋值pan

> get name

"pan"

> append name jing //变量name存在，因此追加

> get name

"panjing"

> append name1 nana //变量name1不存在，因此新建变量并赋值

(integer) 4

**setbit key offset value**

— 对key所存储字串，设置或清除特定偏移量上的位（bit）

— value值可以为1或0,offset为0～2^32之间

— key不存在，则创建新key

> setbit yaya 0 1

> setbit yaya 1 0

> setbit yaya 2 1

> setbit yaya 3 1

> setbit yaya 4 0

(integer) 0

场景说明：记录网站用户上线频率,如用户yaya上线了多少天等类似的数据

**bitcount key**

— 统计字串中被设置为1的比特为数量

> bitcount yaya //统计变量yaya被设置为1的比特为数量

(integer) 3

**decr key**

— 将key中的值减1，key不存在则先初始化为0，再减1

> set x 10

> decr x //变量存在,每次变量值减1；变量不存在，初始化为0再减1

(integer) 9

**decrby key decrement**

— 将key中的值，减去decrement

> decrby x 2 //每次减2

**get key**

— 返回key存储的字符串值，若key不存在则返回null

— 若key的值不是字串，则返回错误，get只能处理字串

**getrange key start end**

— 返回字串值的子字串，截取范围为start 和 end

— 负数偏移量表示从末尾开始计数，-1表示最后一个字符，-2表示倒数第二个字符

> set name qweasd

> getrange name -2 -1 //取出变量的倒数第二位和第一位

"sd"

> getrange name -3 -1

"asd"

> getrange name 0 2 //取出变量的前三位（0为第一位）

"qwe"

**incr key**

— 将key的值加1，如果key不存在，则初始为0后再加1

— 主要应用为计数器

> set y 7

> incr y //变量值默认自加1

**incrby key increment**

— 将key的值增加increment

> incrby y 2 //变量值定义自加2

**incrbyfloat key increment**

— 为key中所存储的值加上浮点数增量increment

> get y

"13"

> incrbyfloat y 0.5 //变量值定义自加0.5

"13.5"

**mget key [key...]**

— 获取一个或多个key的值，空格分隔，具有原子性

> mget as a b x

1) "10xx"

2) "12"

3) "14"

4) "-13"

**mset key value [key value...]**

— 设置多个key及值，空格分隔，具有原子性

> mset box qwe asd zxc p sdf

OK

> mget box asd p

1) "qwe"

2) "zxc"

3) "sdf"

## List列表

### list列表简介

redis的list是一个字符队列

**先进后出**

一个key可以有多个值

**list列表操作**

**lpush key value [value...]**

— 将一个或多个值value插入到列表key的表头

— key不存在，则创建key

> lpush grgname lucy lili tom jum //grgname值顺序从右往左

(integer) 4

> type grgname

list

**lrange key start stop**

— 从开始为值读取key的值到stop结束

> lrange grgname 0 2 //从0位开始，读到2位为止

1) "jum"

2) "tom"

3) "lili"

> lrange grgname 0 -1 //从开始读到结束为止

1) "jum"

2) "tom"

3) "lili"

4) "lucy"

> lrange grgname -3 -1 //从倒数第3位读到倒数第1位

1) "tom"

2) "lili"

3) "lucy"

**lpop key**

— 移除并返回列表头元素数据，key不存在则返回nil

> lpop grgname //移除grgname的头元素数据

"jum"

**llen key**

— 返回列表key的长度

> llen grgname

(integer) 3

**lindex key index**

— 返回列表中第index个值

> lindex grgname 1

"lili"

**lset key index value**

— 将key中index位置的值修改为value

> lset grgname 1 sarah //将grgname中第1个值修改位sarah

> lindex grgname 1

"sarah"

**rpush key value [values...]**

— 将value插入到key的末尾

> rpush list a b c

> lrange list 0 -1

1) "a"

2) "b"

3) "c"

> rpush list d

> lrange list 0 -1

1) "a"

2) "b"

3) "c"

4) "d"

**rpop key**

— 删除并返回key末尾的值

> rpush qname zhangsan lisi wangwu

> rpop qname

"sarah"

### Hash表简介

hash表类型的变量：让一个变量可以有多列 每列有对应的值

redis hash

— 是一个string类型的field，一个field和value的映射表

— 一个key可对应多个field，一个field对应一个value

— 将一个对象存储为hash类型，较于每个字段都存储成string类型更能节省内存

hash表操作

hset key field value

— 将hash表中field值设置为value

> hset book name seby

(integer) 1

> type book

hash

> hset book worker gtq

(integer) 1

> hset book pay 18

(integer) 1

hget key field

— 获取hash表中field的值

> hget book worker

"gtq"

> hget book gtq

(nil)

hkeys key

— 返回hash表中所有field名称

> hkeys book

1) "name"

2) "worker"

3) "pay"

hmset key field value [field value...]

— 同时给hash表中的多个field赋值

> hmset book count 10000 version 2.0

OK

> hkeys book

1) "name"

2) "worker"

3) "pay"

4) "count"

5) "version"

hmget key field [field...]

— 返回hash表中多个field的值

> hmget book count version

1) "10000"

2) "2.0"

hgetall key

— 返回hash表中所有key名和对应的值列表

> hgetall book

1) "name"

2) "seby"

3) "worker"

4) "gtq"

5) "pay"

6) "18"

7) "count"

8) "10000"

9) "version"

10) "2.0"

hvals key

— 返回hash表中所有key的值

> hvals book

1) "seby"

2) "gtq"

3) "18"

4) "10000"

5) "2.0"

hdel key field [field...]

— 删除hash表中多个field的值，不存在则忽略

> hdel book count version

(integer) 2

> hvals book

1) "seby"

2) "gtq"

3) "18"

S