**Morning：**

1、Redis主从复制

**-工作原理**

1. Slave向Master发送sync请求；
2. Master启动后台存盘进程，收集所有新修改的数据命令到一个数据文件；
3. Master传送整个数据文件到Slave；

4）Slave接收数据文件，加载到内存中完成首次完全同步

5）若Slave再次发起sync请求，重复步骤2）

Tips：网络繁忙、系统繁忙，都会产生同步延时。

**-结构模式**

一主一从、一主多从、主从从

1. 配置主从复制

环境准备：所有主机为独立的Redis服务器(取消集群配置)

**-配置从库**

1）命令行(临时)

192.168.4.52:6352> config set masterauth ”123456”

#临时修改配置项(马上生效)

192.168.4.52:6352> slaveof 192.168.4.51 6351 #配置为从库

192.168.4.51:6351> info replication #查看主从信息

role:master

connected\_slaves:1

slave0:ip=192.168.4.52,port=6352,state=online,offset=42,lag=0

192.168.4.52:6352> slaveof no one #反客为主

Tips：服务运行后，默认是master；从库默认不可写(set)。

重启服务即还原为独立服务器。

从库首次完全同步，会把本机所有的数据覆盖，务必做好备份。

2）配置文件(永久)

[root@redis52 ~]# vim /etc/redis/6379.conf

slaveof 192.168.4.51 6351 #设置主库信息

masterauth 123456 #指定主库连接密码(若有)

[root@redis52 ~]# /etc/init.d/redis\_6379 start #重启服务

**-一主多从、主从从**

Tips：配置方法与一主一从相同，直接在配置文件设置主库信息。

**-哨兵模式**

[root@redis54 ~]# cp ./redis/redis-4.0.8/sentinel.conf /etc/

#sentinel.conf模板，安装Redis后提供哨兵程序

[root@redis54 ~]# vim /etc/sentinel.conf

bind 0.0.0.0 #哨兵服务使用的IP，0.0.0.0为所有本机地址

port 26379 #哨兵服务端口

sentinel monitor redis55 192.168.4.56 6356 1

sentinel auth-pass redis55 654321

#sentinel monitor <被监视的主机名> <IP地址> <端口> <票数>

#票数：有几台哨兵主机连接不上主库时，切换主库

[root@redis54 ~]# redis-sentinel /etc/sentinel.conf #启动

Tips：哨兵模式适合主从从结构，当主库宕机，从库自动提升为新主库；

当宕机恢复，该服务器自动变为新主库的slave

**Afternoon：**

1、Redis持久化

**方式一：RDB**

1）RDB介绍

-RDB指的是Redis数据库文件/var/lib/redis/6379/dump.rdb；

-服务启动时，定期将内存数据写入硬盘；

-服务停止时，自动将内存数据写入硬盘；

-恢复时，将快照文件重新加载到内存。

1. RDB优缺点

优点：高性能。创建一个子进程来执行持久化，不占用Redis主进程任何资源；

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，最后一次持久化数据会丢失

3）RDB配置参数及指令

[root@redis502 ~]# vim /etc/redis/6379.conf

dbfilename "dump.rdb" #RDB文件名

save 900 1 #900s且1次修改(存盘频率)

save 300 10 #300s且10次修改

save 60 10000 #60s且10000次修改

#save "" #禁用RDB

#rdbcompression yes|no #RDB压缩，节省空间但占用CPU

#rdbchecksum yes|no #CRC16算法校验，保证数据完整性

#stop-writes-on-bgsave-error yes|no #bgsave出错时停止存盘

192.168.4.50:6350> save #阻塞写存盘(存盘时不允许写)

192.168.4.50:6350> bgsave #不阻塞写存盘

4）备份与恢复

[root@redis50 ~]# cp 数据库目录/dump.rdb 备份目录

#RDB文件备份

[root@redis50 ~]# /etc/init.d/redis\_6379 stop

[root@redis50 ~]# cp 备份目录/dump.rdb 数据库目录

[root@redis50 ~]# /etc/init.d/redis\_6379 start

#RDB文件恢复(操作顺序不能颠倒，会导致dump.rdb被覆盖)

**方式二：AOF**

1）AOF介绍

-记录Redis服务所有写操作(类似于binlog日志)；

-不断将新的写操作，追加到文件的末尾；

-使用cat命令可以查看文件内容。

-AOF与RBD同时开启，**AOF优先**。

2）AOF优缺点

优点：可以灵活设置持久化方式：always、everysec、no；

意外宕机时，仅可能丢失1s的数据。

缺点：持久化文件体积大于RDB方式；

执行fsync策略时速度比RDB更慢。

3）AOF配置参数与操作(面试会考)

192.168.4.50:6350> config set appendonly yes

#运行状态下启用AOF

[root@redis50 ~]# vim /etc/redis/6379.conf

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(占用CPU)**

**appendfsync everysec #每秒记录一次(默认)**

**# appendfsync no #从不记录**

auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb

#重写(对文件作整理而非新建)

#当aof文件是上次rewrite后大小的100%，且文件大于64M时再触发

[root@redis50 6379]# redis-check-aof --fix appendonly.aof

#修复AOF文件

4）备份与恢复

直接对/var/lib/redis/applendonly.aof进行拷贝；

**-RDB与AOF的选择**

关于备份：

1）如果Redis只作缓存用，不用开启任何持久化方式

2）如果数据量大，且对数据一致性要求不高，可只使用RDB

3）一般建议AOF+RDB，因为RDB更适合备份数据库(AOF不断更新，不好备份)，AOF可能存在潜在的BUG，RDB数据文件以防万一。

关于优化：

1. 因为RDB文件只用作后备用途，可以只在Slave上持久化RDB文件；
2. 如果启动了AOF，好处是数据丢失不超过2秒。代价一是带来了持续的IP(占用CPU资源)，二是rewrite的最后将rewrite过程中产生的新数据写入时会造成阻塞。因此，应尽量降低rewrite频率，可以把auto-aof-rewrite-min-size设到5GB以上。

3）可以仅靠Master-Slave复制(持久化RDB)实现高可用性。可以省下非常多的CPU资源，也避免了rewrite时的阻塞。代价是如果Master/Slave同时宕机，会丢失十几分钟的数据。

1. Redis数据类型

**-String字符串**

|  |  |
| --- | --- |
| 语法 | 案例 |
| **Set *key value* [ex *数字*] [px *数字*] [nx|xx]**  #过期时间ex(秒)、px(毫秒)  #nx仅变量不存在时赋值，xx仅变量存在时赋值 | > Set tel 123456 ex 5 nx  “123456” |
| **Setrange *key offset value***  #修改变量指定范围的值 | > Setrange tel 2 \*\*  “12\*\*56” |
| **Strlen *key***  #统计字符长度 | > Strlen tel  (integer) 6 |
| **Append *key value***  #追加赋值，不存在则创建 | > Append tel abc  "123456abc" |
| **Setbit *key offset value***  #设置位存储，offset为0~2^32，value为0/1  #应用场景：记录网站用户上线天数 | > Setbit tmp\_bits 0 1  "\x80" |
| **Bitcount *key***  #统计字串中为1的位数量 | > Bitcount tmp\_bits  (integer) 1 |
| **Decr *key***  #key值自减1；若不存在则先初始化为0，再减1 | > decr x  (integer) -1 |
| **Decrby *key decrement***  #key值自减整数 | > decrby x 10  (integer) -11 |
| **Get *key***  #返回key的字符串值，不存在则返回nil  #若不是String类型，则报错 | > get abc  (nil) |
| **Getrange *key start end***  #截取字串。从0开始，负数表示从末尾开始计数 | > getrange tel 0 -1  "123456abc" |
| **incr *key***  #key值自加1；若不存在则先初始化为0，再加1 | > incr y  (integer) 1 |
| **increby *key increment***  #key值自加整数 | > incrby y 10  (integer) 11 |
| **incrbyfloat *key increment***  #key值自加浮点数 | > incrbyfloat y 0.5  "11.5" |
| **mget *key [key...]***  #获取多个key的值 | > mget x y  1) "-11"  2) "11.5" |
| **mset *key value [key value ...]***  #设置多个key值 | > mset a 11 b 22 |

**-List列表**

字符队列(相当于数组)，先进后出(堆栈)，一个key可以有多个值。

|  |  |
| --- | --- |
| 语法 | 案例 |
| **lpush *key value [value...]***  #将值插入到列表key的表头(入栈) | > lpush ls\_tmp aa bb cc |
| **lrange *key start stop***  #范围读取列表key的值(0开始,负数表示末尾) | > lrange ls\_tmp 0 -1  1) "cc"  2) "bb"  3) "aa" |
| **lpop *key***  #移除并显示列表头元素数据，不存在则返回nil | > lpop ls\_tmp  "cc" |
| **llen *key***  #返回列表key的长度 | > llen ls\_tmp  (integer) 2 |
| **lindex *key index***  #返回列表key第index个值(0开始) | > lindex ls\_tmp 0  "bb" |
| **lset *key index value***  #修改列表index位置的值(0开始) | > lset ls\_tmp 0 yy  1) "yy"  2) "aa" |
| **rpush *key value [value...]***  #将value插入到key的末尾 | > rpush ls\_tmp en1 en2  1) "yy"  2) "aa"  3) "en1"  4) "en2" |
| **rpop *key***  #删除key末尾的值 | > rpop ls\_tmp  "en2" |

**-Hash表**

是一个string类型的field(列)和value(值)的映射表

|  |  |
| --- | --- |
| 语法 | 案例 |
| **hset *key field value***  #给hash表中单个field赋值 | > hset hs\_tmp f1 val1 |
| **hget *key field***  #获取hash表中单个field的值 | > hget hs\_tmp f1  "val1" |
| **hmset** ***key field value [field value..]***  #给hash表中多个field赋值 | > hmset hs\_tmp f2 val2 f3 val3 |
| **hmget *key field [field...]***  #获取hash表中多个field的值 | > hmget hs\_tmp f2 f3  1) "val2"  2) "val3" |
| **hkeys *key***  #返回hash表中所有field名称 | > hkeys hs\_tmp  1) "f1"  2) "f2"  3) "f3" |
| **hgetall *key***  #返回hash表中所有field名和对应的值 | > hgetall hs\_tmp  1) "f1"  2) "val1"  3) "f2"  4) "val2"  5) "f3"  6) "val3" |
| **hvals *key***  #返回hash表中所有值 | > hvals hs\_tmp  1) "val1"  2) "val2"  3) "val3" |
| **hdel *key field [field...]***  #删除hash表中多个field的值 | > hdel hs\_tmp f1 f3 |

Tips：其他常用全局操作指令，参考阶段3 -> 3.NoSQL -> DAY01 -> Page2