

# Redis

## Redis基本使用

在linux系统下使用;

centos7

redis版本为6.x版本

官方文档: <http://redis.cn/>

// 英文版本下载地址

<http://download.redis.io/releases/redis-6.2.5.tar.gz>

xshell

xftp

- <https://www.netsarang.com/en/free-for-home-school>
- 免费使用

- 连接阿里云服务器

- ssh命令连接, 必须保证开启了22端口;
  - ssh root@ip地址 回车
- 免密码登录, 「mac/linux下配置」, window下直接记住用户名或密码;

- 如果在服务器安装任何软件,都要干一件事. 开放远程访问端口;

- 腾讯服务器 -> 防火墙.
- 阿里/华为 -> 安全组 -> 添加规则

## 第一章: 基本操作

Redis 是一个开源 (BSD许可) 的, 内存中的数据结构存储系统, 它可以用作数据库、缓存和消息中间件.

### 1. 安装

#### a. 源码方式安装

- 安装gcc编译器

## Bash

```
1 yum install cpp
2 yum install binutils
3 yum install glibc
4 yum install glibc-kernheaders
5 yum install glibc-common
6 yum install glibc-devel
7 yum install gcc
8 yum install make
9
10 # 升级gcc编译器
11 yum -y install centos-release-scl
12 yum -y install devtoolset-9-gcc devtoolset-9-gcc-c++ devtoolset-9-binutils
13
14 scl enable devtoolset-9 bash
```

- 下载redis源码「推荐华为云镜像, 速度快一些」

## Bash

```
1 # 6.x 最新稳定版本
2 https://repo.huaweicloud.com/redis/redis-6.0.9.tar.gz
3
4 # 5.x 最新稳定版本
5 https://repo.huaweicloud.com/redis/redis-5.0.9.tar.gz
6
7 # 4.x 最新稳定版本
8 https://repo.huaweicloud.com/redis/redis-4.0.9.tar.gz
9
10 # 下载, 这里以6.x版本为例
11 # cd /usr/local
12 # 先安装wget命令. yum install -y wget
13 wget https://repo.huaweicloud.com/redis/redis-6.0.9.tar.gz
14
15 # 使用此版本即可;
16 wget https://repo.huaweicloud.com/redis/redis-6.2.5.tar.gz
17 https://repo.huaweicloud.com/redis/redis-6.2.5.tar.gz
```

- 解压压缩包

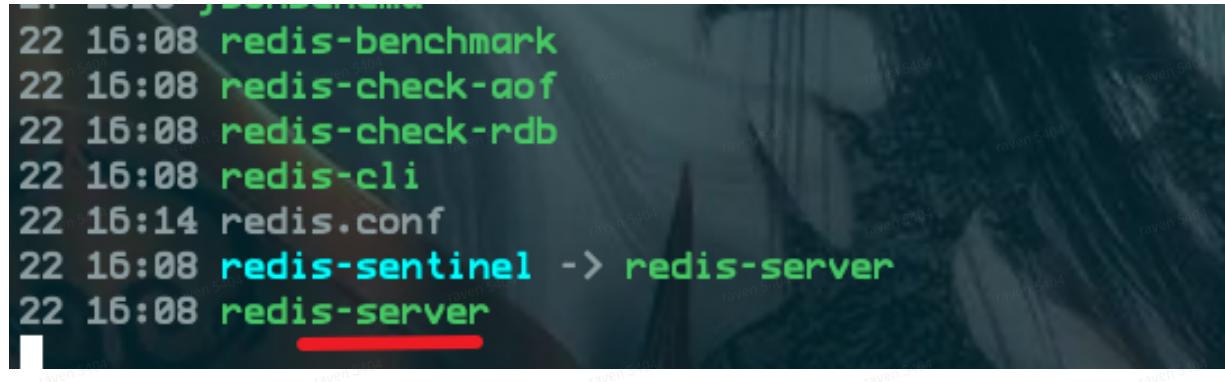
## Bash

```
1 tar -xzvf redis-6.0.9.tar.gz
```

- 安装

## Bash

```
1 # 进入目录
2 cd redis-6.0.9
3
4 # 执行安装命令
5 make
6
7 # 如果没有错误，则直接进行安装即可；
8 make install
9
10 # 或者上述两个命令合并成一条命令执行
11 make && make install
12
13 # centos7默认安装路径：
14 /usr/local/bin
```



## 2. 启动服务端

### 1. 修改配置文件

## Bash

```
1 # 1. 从解压的包中拷贝一份**redis.conf**文件到/usr/local/bin 「redis安装位置」
2 cd /usr/local/redis-6.0.9
3 cp redis.conf myredis.conf
4
5 # 2. 修改配置文件
6 cd /usr/local/bin
7 vim redis.conf
8 set nu # 显示行数
9
10 修改第68行左右:
11 bind 127.0.0.1 --> 修改为: # bind 127.0.0.1
12 0.0.0.0:-1
13 修改224行左右: 守护进程的方式启动
14 no --> 修改为: # daemonize yes
15
16 搜索logfile 303行 # 日志路径,是一个文件, 带后缀 6379.log
17 搜索pidfile 290行 # pid路径
18 搜索requirepass 写上密码 902行 # 访问密码, 必须得加上,要不然容易被攻击
19 /usr/local/redis-6.0.9/src , 安装完之后,这里会生成多个可执行文件;显示是绿色的;
```

```
-rwxr-xr-x 1 root root 5658264 2月 22 16:07 redis-check-aof
-rw-rw-r-- 1 root root 7175 10月 27 2020 redis-check-aof.c
-rw-r--r-- 1 root root 359 2月 22 16:07 redis-check-aof.d
-rw-r--r-- 1 root root 76984 2月 22 16:07 redis-check-aof.o
-rwxr-xr-x 1 root root 5658264 2月 22 16:07 redis-check-rdb
-rw-rw-r-- 1 root root 14455 10月 27 2020 redis-check-rdb.c
-rw-r--r-- 1 root root 359 2月 22 16:07 redis-check-rdb.d
-rw-r--r-- 1 root root 87744 2月 22 16:07 redis-check-rdb.o
-rwxr-xr-x 1 root root 1049352 2月 22 16:07 redis-cli
-rw-rw-r-- 1 root root 309466 10月 27 2020 redis-cli.c
-rw-r--r-- 1 root root 222 2月 22 16:07 redis-cli.d
-rw-r--r-- 1 root root 1242920 2月 22 16:07 redis-cli.o
-rw-rw-r-- 1 root root 60155 10月 27 2020 redismodule.h
-rwxr-xr-x 1 root root 5658264 2月 22 16:07 redis-sentinel
-rwxr-xr-x 1 root root 5658264 2月 22 16:07 redis-server
-rwxrwxr-x 1 root root 3600 10月 27 2020 redis-trib.rb
-rw-rw-r-- 1 root root 2591 2月 22 15:54 release.c
-rw-r--r-- 1 root root 49 2月 22 16:07 release.d
```

## 2. 启动

- 前台启动

## Bash

```
1 redis-server redis.conf # redis.conf 是配置文件
```

- 后台启动

## Bash

```
1 redis-server redis.conf &
```

- 查看是否已经启动了

## Bash

```
1 # 查看进程
2 ps -ef | grep redis
3
4 root    9652      1  0 16:15 ?    00:00:01 redis-server *:6379
5 root    9750  9695  0 16:37 pts/0 00:00:00 grep --color=auto redis
```

## 3. 启动客户端

### ◦ 本机启动

## Bash

```
1 redis-cli, 回车
```

### ◦ 远程连接

- 如果把redis装在了centos服务器上, 想通过终端来连接它. 需要做如下设置:

#### ◦ 启动方式修改

- 不要去启动默认的安装路径上的文件 「/usr/local/bin/redis-service」
- 要启动redis解压完的文件夹中的, src目录下的redis-service.

#### ◦ 配置文件修改

redis.conf, 建议复制一份出来, 再进行修改. 详细修改如下所示:

## Bash

```
1 # 1. 打开启动redis时候指定的配置文件. 搜索bind, 找到bind 127.0.0.1.  
2 # 把这行注释掉了. 加上一行: bind 0.0.0.0  
3 # 2. 搜索protected-mode, 原来是这样的: protected-mode yes, 即开启保护模式, 这种模式  
下远程无法连接上; 手动改为 no, 即protected-mode no  
4 # 3. 设置连接密码「可选操作, 生产环境必须要选」, 搜索: requirepass, 或者直接在配置文件  
中添加此值也可以: requirepass 你的密码「学习的时候密码简单一些无所谓, 再次强调一下, 生  
产环境, 必须尽量的复杂一些」, 连接时候, 必须密码正确才能连接  
5 # 4. 设置后台启动「可选参数」, 搜索daemonize, 原来是这样的: daemonize no, 即默认值是  
no, 修改为: daemonize yes.  
6 # 5. 如果是阿里云服务器, 则需要开启远程端口访问. 安全组那块的内容;  
7 // mysql : 3306, redis: 6379  
8 # 如果, 配置了密码保护, 那么在登录的客户端的时候, 必须指定密码, 否则操作无效;  
9 redis-cli -a 123456 -p 6379, 表示使用6379服务器, 并且密码是: 123456
```

- 如果说不在本机上连 redis, 则需要按如下方式连接;

## Bash

```
1 redis-cli -h host -p port -a password  
2  
3 -h: 主机地址, 写ip地址或者是域名都可以;  
4 -p: 端口号, 默认端口或者你自己修改的端口号, 必须保证在阿里中安全组开放了该端口  
5 -a: 密码, 在redis.conf中配置的: requirepass的值  
6 # 密码在启动时候的配置文件里指定: requirepass 密码  
7  
8 # 连接本地  
9 redis-cli  
10
```

## 4. 测试环境

### Bash

```
1 # 连接好以后, 可以直接输入ping, 如果返回是pong, 则表示连接成功;  
2 127.0.0.1:6379> ping  
3 PONG
```

## 第二章: 常用操作

<https://db-engines.com/en/ranking>

- 操作数据库

命令名称	描述
<b>flushdb</b>	清空数据,慎用
<b>flushall</b>	清空全部数据.不要用. 当这个命令不存在
select index, 了解一下即可,一般都是默认0;	选择哪个数据库,从0到15,一共16了.一般默认0就好了

- 操作key的常用集合「重点内容」

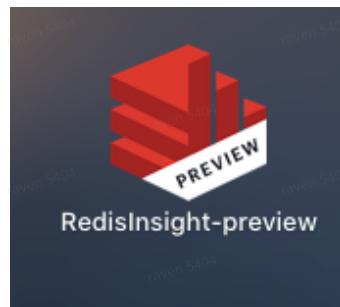
命令名称	描述
exists key-name	根据key的名称,判断key是否存在
<b>expire key</b>	设置某个key的过期时间,「单位以秒为单位」
<b>ttl key</b>	查看key的剩余时间
<b>type key</b>	查看key的类型
<b>keys *</b>	列出所有的key

- 相关示例

## Bash

```
1 127.0.0.1:6379> flushdb
2 OK
3 127.0.0.1:6379> flushall
4 OK
5 127.0.0.1:6379> select 0
6 OK
7 127.0.0.1:6379> keys *
8 (empty array)
9
10 127.0.0.1:6379> set name tom
11 OK
12 127.0.0.1:6379> get name
13 "tom"
14 127.0.0.1:6379> expire name 60
15 (integer) 1
16 127.0.0.1:6379> ttl name
17 (integer) 56
18 127.0.0.1:6379> exists name
19 (integer) 1
20 127.0.0.1:6379> type name
21 string
22 127.0.0.1:6379> ttl name
23 (integer) 30
24 127.0.0.1:6379>
```

## 第三章: 数据类型



redis内存型的数据库, key是string, value有8种数据类型; 「stream新增」

### 1. string

`String` 是一组字节. 在 `redis` 数据库中, 字符串是**二进制安全的**. 这意味着它们具有已知的长度, 并且不受任何特殊终止字符的影响.

## 可以在字符串存储的最多**512兆字节**的内容

### • 基本命令

命令名称	描述
<code>set key value</code> <code>set key value [EX seconds PX milliseconds KEEPTTL] [NX XX]</code> <ul style="list-style-type: none"><li>• <code>EX seconds</code> – 设置键key的过期时间, 单位时秒</li><li>• <code>PX milliseconds</code> – 设置键key的过期时间, 单位时毫秒</li><li>• <code>NX</code> – 只有键key不存在的时候才会设置key的值</li><li>• <code>XX</code> – 只有键key存在的时候才会设置key的值</li></ul>	<code>set</code> 关键字, <code>key</code> 表示键值, <code>value</code> 表示 <code>key</code> 对应的值. 设置值; <code>set name tom EX 30 NX</code>
<code>get key</code>	根据 <code>key</code> 获取对应的 <code>value</code>
<code>append key appendValue</code>	向已经存在的 <code>key</code> 中追加字符串, <code>appendValue</code> 追加值. 如果当前的 <code>key</code> 不存在, 则直接添加. 如果已经存在 <code>key</code> , 则追加值; <code>key</code> 不存在, 添加的时候, 不会报错;
<code>strlen key</code>	根据 <code>key</code> 获取 <code>value</code> 的字符串长度

## Bash

```
1 127.0.0.1:6379> flushall # 清除所有内容
2 OK
3 127.0.0.1:6379> set name tom # name是key, tom是值
4 OK
5 127.0.0.1:6379> get name # 获取name对应的key值
6 "tom"
7 127.0.0.1:6379> append name jerry # 追加操作
8 (integer) 8
9 127.0.0.1:6379> get name # 获取name对应的value值
10 "tomjerry"
11 127.0.0.1:6379> strlen name # 获取name对应的value值的长度
12 (integer) 8
13
14 ##### # append操作
15 # append操作
16 127.0.0.1:6379> append address 北京 # 如果key不存在,执行追加操作.则直接设置值
17 (integer) 6
18 127.0.0.1:6379> get address # 获取key对应的值
19 "\xe5\x8c\x97\xe4\xba\xac" # 自动编码了.
20 127.0.0.1:6379> append address sanhe # 追加操作
21 (integer) 11
22 127.0.0.1:6379> get address # 获取操作
23 "\xe5\x8c\x97\xe4\xba\xacsanhe"
24 127.0.0.1:6379>
```

### • 自增/自减命令

- 如果操作的当前key不存在,则新建一个.直接加/减;

命令名称	描述
incr key	给已经存在key自增1, <b>如果key不存在,则默认从0开始自增.</b> key不存在,也不会报错的;
decr key	给已经存在的key减1, 如果key不存在,则默认从0开始自减.
incrby key step	给已经存在的key自增 step 值, 如果key不存在,则默认从0开始自增.
decrby key step	给已经存在的key 减去 step 值, 如果key不存在,则默认从0开始自减

## Bash

```
1 127.0.0.1:6379> keys * # 查找所有的key
```

```
raven 5404 2 (empty array)
raven 5404 3 127.0.0.1:6379> set count 0 # 设置初始值为0
raven 5404 4 OK
raven 5404 5 127.0.0.1:6379> get count
raven 5404 6 "0"
raven 5404 7 127.0.0.1:6379> incr count # 自增1
raven 5404 8 (integer) 1
raven 5404 9 127.0.0.1:6379> get count
raven 5404 10 "1"
raven 5404 11 127.0.0.1:6379> incrby count 10 # 根据步长自增操作
raven 5404 12 (integer) 11
raven 5404 13 127.0.0.1:6379> get count
raven 5404 14 "11"
raven 5404 15 #####
raven 5404 16 #####
raven 5404 17 # 自减操作
raven 5404 18 127.0.0.1:6379> decr count # 自减
raven 5404 19 (integer) 10
raven 5404 20 127.0.0.1:6379> get count
raven 5404 21 "10"
raven 5404 22 127.0.0.1:6379> decr count # 自减
raven 5404 23 (integer) 9
raven 5404 24 127.0.0.1:6379> get count
raven 5404 25 "9"
raven 5404 26 127.0.0.1:6379> decrby count 5 # 根据步长自减
raven 5404 27 (integer) 4
raven 5404 28 127.0.0.1:6379> get count
raven 5404 29 "4"
raven 5404 30 127.0.0.1:6379>
raven 5404 31 #####
raven 5404 32 #####
raven 5404 33 # 如果key不存在, 此时incr/decr/incrby/decrby则默认key对应的值是0. 从开始自增/自减
raven 5404 34 127.0.0.1:6379> flushall # 清除所有的内容
raven 5404 35 OK
raven 5404 36 127.0.0.1:6379> keys * # 获取所有的key
raven 5404 37 (empty array)
raven 5404 38 127.0.0.1:6379> incr count # key不存在, 则默认值是0, 即从0开始自增1.
raven 5404 39 (integer) 1
raven 5404 40 127.0.0.1:6379> get count
raven 5404 41 "1"
raven 5404 42 #####
raven 5404 43 #####
raven 5404 44 # 如果key对应的字符串无法转换为整型, 则抛出错误
raven 5404 45 127.0.0.1:6379> flushall
raven 5404 46 OK
raven 5404 47 127.0.0.1:6379> set name tom # 设置值
raven 5404 48 OK
raven 5404 49 127.0.0.1:6379> incr name # 使用自增/自减
```

```
raven5404 127.0.0.1:6379> SET name "日本料理"
raven5404 50 (error) ERR value is not an integer or out of range # 报错了
raven5404 51 127.0.0.1:6379>
```

## • 字符串操作

命令名称	描述
<p><b>getrange</b> key startIndex endIndex <b>当成subString()</b></p>	<p>截取key对象值的字符串,返回值是截取之后的字符串,不会修改原来的值;</p> <p>参数:</p> <ul style="list-style-type: none"><li>◦ key, key的名称</li><li>◦ startIndex, 开始截取的位置</li><li>◦ endIndex, 截取的结束位置, 如果是-1,则表示截取到字符串末尾</li></ul> <p>▪ 强调: [startIndex,endIndex], 闭区间. 索引值从0开始</p>
<p><b>setrange</b> key offset value</p>	<p>替换key对应值的字符串, 返回值是替换之后的字符串, <b>会修改原来的key对应的值</b>;</p> <p>参数:</p> <ul style="list-style-type: none"><li>◦ key, key的名称</li><li>◦ offset, key对应值的索引值, 从0开始.</li><li>◦ value, 要替换的值, 该值是字符串.</li></ul>

Bash

```
1 #####  
2 # getrange key start end  
3 127.0.0.1:6379> flushall  
4 OK  
5 127.0.0.1:6379> keys *  
6 (empty array)  
7 127.0.0.1:6379> set w 'hello world'  
8 OK  
9 127.0.0.1:6379> get w  
10 "hello world"  
11 127.0.0.1:6379> getrange w 0 4 # 注意是闭区间  
12 "hello"  
13 127.0.0.1:6379> get w  
14 "hello world"  
15 127.0.0.1:6379> getrange w 0 -1 # end值是-1的时候,表示一直取到末尾  
16 "hello world"  
17 127.0.0.1:6379> get w  
18 "hello world"
```

```

raven 5404 19 ######
raven 5404 20 # 如果操作的key不存在,则返回的是空字符串 "", 不会报错;
raven 5404 21 127.0.0.1:6379> getrange p 0 3
raven 5404 22 ""
raven 5404 23 #####
raven 5404 24 # setrange key offset value
raven 5404 25 127.0.0.1:6379> flushall
raven 5404 26 OK
raven 5404 27 127.0.0.1:6379> keys *
raven 5404 28 (empty array)
raven 5404 29 127.0.0.1:6379> set w abcdefg
raven 5404 30 OK
raven 5404 31 127.0.0.1:6379> get w
raven 5404 32 "abcdefg"
raven 5404 33 127.0.0.1:6379> setrange w 0 0 # 把第0个位置的a替换为0
raven 5404 34 (integer) 7
raven 5404 35 127.0.0.1:6379> get w
raven 5404 36 "0bcdefg"
raven 5404 37 127.0.0.1:6379> setrange w 1 11 # 把第1个位置的b替换为11, 会覆盖掉原来的值;
raven 5404 38 (integer) 7
raven 5404 39 127.0.0.1:6379> get w
raven 5404 40 "011defg"
raven 5404 41 127.0.0.1:6379> setrange w 3 23456789 # 替换的字符串个数超出原来的值了, 则会保留覆盖
raven 5404 42 (integer) 11
raven 5404 43 127.0.0.1:6379> get w
raven 5404 44 "01123456789"
raven 5404 45 #####
raven 5404 46 # 如果key不存在, 则直接进行添加操作; 不会报错;
raven 5404 47 127.0.0.1:6379> setrange k 0 abcdefg
raven 5404 48 (integer) 7
raven 5404 49 127.0.0.1:6379> get k
raven 5404 50 "abcdefg"
raven 5404 51 127.0.0.1:6379>

```

## • 极其重要的命令!!

命令名称	描述
<b>setex</b> 「set with expire」 <ul style="list-style-type: none"> <li>◦ 语法               <ul style="list-style-type: none"> <li>▪ setex key second value</li> </ul> </li> </ul>	在设置key的时候, 同时设置该key的超时时间 参数: <ul style="list-style-type: none"> <li>◦ key, 要设置的key的名称</li> <li>◦ second, expire time, 过期时间, 以秒为单位</li> <li>◦ value, 该key的值</li> </ul>
<b>setnx</b> 「set if with exists」	key值不存在再进行设置. 如果key存在, 则设置失败; 「返回0」

<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ setnx key value</li> </ul> </li> </ul>	<p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, 要设置的key的名称</li> <li>◦ value, key的值           <ul style="list-style-type: none"> <li>▪ <b>如果key不存在,则正确设置;如果key已经存在,则设置无效,也不会报错;</b></li> <li>▪ 成功返回1, 失败返回0;</li> </ul> </li> </ul>
<b>mset 「multiple set」</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ mset k1 v1 k2 v2 ....</li> </ul> </li> </ul>	<p>一次性设置多个值.</p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ k1, key的值</li> <li>◦ v1, value的值</li> <li>◦ ....</li> </ul>
<b>mget 「multiple get」</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ mget k1 k2 k3</li> </ul> </li> </ul>	<p>一次性获取多个key对应的value值</p> <p><b>参数</b></p> <ul style="list-style-type: none"> <li>◦ k1,k2,k3... key的名称</li> </ul>
<b>msetnx 「multiple setnx」</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ msetnx k1 v1 k2 v2 k3 v3</li> </ul> </li> </ul>	<p><b>一次性设置多个值,当key不在的时候设置成功. 其中有一个key已经存在了,则整体设置失败了. 这个操作保证了原子性.</b></p>
<b>getset</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ getset key value</li> </ul> </li> </ul>	<p>获取的时候,重新设置新值,返回值是原来的值;</p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, key的名称</li> <li>◦ value, 要设置的新值</li> </ul>

## • 使用示例

Bash
<pre> 1 ###### 2 # setex, set expire, 设置值并设置过期时间 3 127.0.0.1:6379&gt; flushall 4 OK 5 127.0.0.1:6379&gt; keys * 6 (empty array) 7 127.0.0.1:6379&gt; setex name 120 tom # 设置key并设置过期时间 8 OK 9 127.0.0.1:6379&gt; ttl name # 查看过期时间 10 (integer) 117 11 127.0.0.1:6379&gt; ttl name 12 (integer) 107 13 127.0.0.1:6379&gt; 14 </pre>

```
15 #####  
16 # setnx, 如果key值不存在,则进行值的设置,如果key已经存在,则设置值无效  
17 127.0.0.1:6379> keys *  
18 (empty array)  
19 127.0.0.1:6379> setnx name tom # key不存在,则可以进行正确的设置值;  
20 (integer) 1  
21 127.0.0.1:6379> get name  
22 "tom"  
23 127.0.0.1:6379> setnx name jerry # key已经存在,则设置无效,也不会报错;  
24 (integer) 0  
25 127.0.0.1:6379> get name  
26 "tom"  
27 127.0.0.1:6379>  
28 #####  
29 #####  
30 # mset, multiple set, 一次性设置多个值  
31 127.0.0.1:6379> mset k1 tom k2 jerry k3 kate k4 jack  
32 OK  
33 127.0.0.1:6379> keys *  
34 1) "k4"  
35 2) "k3"  
36 3) "k2"  
37 4) "k1"  
38 127.0.0.1:6379> get k4  
39 "jack"  
40 127.0.0.1:6379>  
41 #####  
42 #####  
43 # mget, multiple get, 一次性获取多个值;  
44 127.0.0.1:6379> mget k1 k2 k3  
45 1) "tom"  
46 2) "jerry"  
47 3) "kate"  
48 127.0.0.1:6379> mget k2 k10 # 如果获取的key中有不存在的key,则返回的值为nil  
49 1) "jerry"  
50 2) (nil)  
51 127.0.0.1:6379>  
52 #####  
53 #####  
54 # msetnx, multiple setnx.  
55 # 一次性设置多个值, 如果有一个key存在则设置整个失败. 保证了该操作的原子性;  
56 127.0.0.1:6379> flushall  
57 OK  
58 127.0.0.1:6379> keys *  
59 (empty array)  
60 127.0.0.1:6379> msetnx k1 tom k2 jack k3 jerry k4 kate k5 rose  
61 (integer) 1
```

```
62 127.0.0.1:6379> mget k1 k2
63 1) "tom"
64 2) "jack"
65 127.0.0.1:6379> msetnx k5 john k6 hehe # 保证原子性.
66 (integer) 0
67 127.0.0.1:6379> mget k5
68 1) "rose"
69 127.0.0.1:6379>
70
71 # 设置复杂对象
72 # user:1:name这个是key值
73 127.0.0.1:6379> msetnx user:1:name tom user:1:age 20
74 (integer) 1
75 127.0.0.1:6379> mget user:1:name user:1:age # 获取值操作
76 1) "tom"
77 2) "20"
78 127.0.0.1:6379> keys *
79 1) "k3"
80 2) "k5"
81 3) "user:1:name"
82 4) "k4"
83 5) "k1"
84 6) "k2"
85 7) "user:1:age"
86 127.0.0.1:6379>
87
88 ######
89 # getset, 获取的时候同时设置新值
90 127.0.0.1:6379> keys *
91 1) "k3"
92 2) "k5"
93 3) "user:1:name"
94 4) "k4"
95 5) "k1"
96 6) "k2"
97 7) "user:1:age"
98 127.0.0.1:6379> get k3
99 "jerry"
100 127.0.0.1:6379> getset k3 polly # 取出原来的jerry , 把polly设置为新值
101 "jerry" # 返回值是原来的值.
102 127.0.0.1:6379> get k3 # 再次查看, 值已经更改了;
103 "polly"
104 127.0.0.1:6379>
```

## 2. hash 「哈希」

哈希就是键值对集合;

在 redis 里, 哈希是字符串字段和字符串值之间的映射. 因此,它们适合表示对象;

## 1. hash相当于java当中的map集合. List<Map<Object, Object>>;

- a. List<Map<String, Map<String, Object>>

## 2. hash更适合存储对象

## 3. 通常情况下,hash的命令都是以h开头的;

## 4. 用法与string类型的类似

### Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HRANDFIELD
- HSCAN
- HSET
- HSETNX
- HSTRLEN
- HVALS

### • 常用命令

命令名称	描述
hset  ◦ 语法 <ul style="list-style-type: none"><li>▪ hset key field value[field value...]</li></ul> 如果 <b>key</b> 不存在, 则创建一个包含哈希的新密钥. 如果 <b>field</b> 散列中已经存在, 则将其覆盖.	向hash中添加值,这里特别注意添加的是键值对  参数: <ul style="list-style-type: none"><li>◦ key, hash的key值</li><li>◦ field, 字段值, 相当于键值对的key值</li><li>◦ value, 设置的值</li><li>◦ [field value], 可选参数,可以设置多个值</li></ul>
hget	根据filed获取hash里的值

<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hget key field</li> </ul> </li> </ul>	<p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, hash中设置的key值</li> <li>◦ field, 键值对的key值</li> </ul>
<h3>hmset</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hmset key field value[field value...]</li> </ul> </li> </ul>	<p><b>设置多个值</b></p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, hash的key值</li> <li>◦ field, 字段值, 相当于键值对的key值</li> <li>◦ value, 设置的值</li> <li>◦ [field value], 可选参数, 可以设置多个值</li> </ul>
<h3>hmget</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hmget key field [field ...]</li> </ul> </li> </ul>	<p><b>获取多个field的值</b></p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, hash中的key值</li> <li>◦ field, 键值对的key值</li> <li>◦ [field ...], 可选参数, 可以同时获取多个field的值</li> </ul>
<h3>hgetall</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hgetall key</li> </ul> </li> </ul>	<p><b>获取指定key的所有的值, 包含filed和value;</b></p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, hash的key</li> </ul>
<h3>hdel</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hdel key field [field ....]</li> </ul> </li> </ul>	<p><b>删除hash对应的key的对应的field值</b></p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, hash的key</li> <li>◦ field, 键值对的字段值</li> <li>◦ [field ....], 可靠参数 , field的值</li> </ul>
<h3>hexists</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hexists key field</li> </ul> </li> </ul>	<p><b>判断hash中指定的字段「field」是否存在</b></p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ key, hash中的key</li> <li>◦ field, 字段名称</li> </ul>
<h3>hkeys</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hkeys key</li> </ul> </li> </ul>	<p><b>获取hash中所有的field值</b></p> <p><b>参数:</b></p> <ul style="list-style-type: none"> <li>◦ hash中的key</li> </ul>
<h3>hvals</h3> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ hvals key</li> </ul> </li> </ul>	<p><b>获取hash指定的key所有的value值</b></p> <p><b>参数</b></p> <ul style="list-style-type: none"> <li>◦ hash中的key</li> </ul>

## ● 示例参考

Bash

```
1 #####  
2 # hset, 添加值, 支持添加多个键值对  
3 # hset key field value [field....]  
4 127.0.0.1:6379> flushall  
5 OK  
6 127.0.0.1:6379> keys *  
7 (empty array)  
8 127.0.0.1:6379> hset user name tom age 20 gender 1  
9 (integer) 3  
10 #####  
11 # hget, 获取一个值  
12 # hset key field  
13 127.0.0.1:6379> hget user name  
14 "tom"  
15 127.0.0.1:6379> hget user age  
16 "20"  
17 #####  
18 # hmset, 添加值, 添加多个键值对  
19 # hmset key field value [field....]  
20 127.0.0.1:6379> hmset user id 1024 address china  
21 OK  
22 #####  
23 # hmget, 获取多个键值对的值  
24 # hmget key field [field....]  
25 127.0.0.1:6379> hmget user id address age name  
26 1) "1024"  
27 2) "china"  
28 3) "20"  
29 4) "tom"  
30 #####  
31 # hgetall , 获取hash中的key所有键值对  
32 # hgetall key  
33 127.0.0.1:6379> hgetall user  
34 1) "name"  
35 2) "tom"  
36 3) "age"  
37 4) "20"  
38 5) "gender"  
39 6) "1"  
40 7) "id"  
41 8) "1024"  
42 9) "address"  
43 10) "china"  
44 #####
```

```

45  # hdel , 删除hash中指定的field的值
46  # hdel key field
47  127.0.0.1:6379> hdel user id
48  (integer) 1
49  127.0.0.1:6379> hgetall user
50  1) "name"
51  2) "tom"
52  3) "age"
53  4) "20"
54  5) "gender"
55  6) "1"
56  7) "address"
57  8) "china"
58  #####
59  # hexists, 判断某一个key中是否存在field, 存在返回1, 不存在返回0
60  # hexists key field
61  127.0.0.1:6379> hexists user name
62  (integer) 1
63  127.0.0.1:6379> hexists user score
64  (integer) 0
65  #####
66  # hkeys , 获取指定的key中所有的键的值
67  # hkeys key
68  127.0.0.1:6379> hkeys user
69  1) "name"
70  2) "age"
71  3) "gender"
72  4) "address"
73  #####
74  # hvals , 获取指定的key中所有的值的值
75  # hvals key
76  127.0.0.1:6379> hvals user
77  1) "tom"
78  2) "20"
79  3) "1"
80  4) "china"
81  127.0.0.1:6379>

```

**key filed value, 更适合存储对象;切记. 命令以h开头,表示hash操作;**

### 3. 列表「list」

基本的数据类型，列表。可以把它当作一个栈，队列，阻塞队列等使用；非常的实用；十分的强大；

- 可以当作栈使用, 先进后出;
- 队列使用, 先进先出;

## • 阻塞队列使用

## • 功能强大

list的基本命令都是以l开头的. 表示该命令是list

命令名称	描述
lpush <ul style="list-style-type: none"><li>◦ 语法<ul style="list-style-type: none"><li>▪ lpush key element [element ...]</li></ul></li></ul>	从列表左边插入新值, <b>这个过程相当于元素入栈操作</b> 把一个值插入到列表的头部.  参数说明: <ul style="list-style-type: none"><li>◦ element, 元素</li><li>◦ [element ...], 可选参数.可以一次性添加多个值</li></ul>
lrange <ul style="list-style-type: none"><li>◦ 语法<ul style="list-style-type: none"><li>▪ lrange key start stop</li></ul></li></ul>	根据key和获取值操作  参数说明: <ul style="list-style-type: none"><li>◦ key, key的值</li><li>◦ start, 开始索引值</li><li>◦ stop, 中止索引值</li><li>◦ 特别说明<ul style="list-style-type: none"><li>▪ [start, stop]闭区间</li><li>▪ 索引值从0开始</li><li>▪ stop = -1, 表示从start位置一直取到末尾</li><li>▪ 正向/负向索引</li><li>▪ 从左往右取值;</li></ul></li></ul>
rpush <ul style="list-style-type: none"><li>◦ 语法<ul style="list-style-type: none"><li>▪ rpush key element [element ..]</li></ul></li></ul>	从列表右边插入新值.  参数说明: <ul style="list-style-type: none"><li>◦ element, 元素</li><li>◦ [element ...], 可选参数.可以一次性添加多个值</li></ul>
lpop <ul style="list-style-type: none"><li>◦ 语法<ul style="list-style-type: none"><li>▪ lpop key</li></ul></li></ul>	从列表左边取出一个值;返回值是取出的元素; <b>真正的从列表当中移除了取出的值;</b>  参数说明: <ul style="list-style-type: none"><li>◦ key, 列表的key</li></ul>
rpop <ul style="list-style-type: none"><li>◦ 语法<ul style="list-style-type: none"><li>▪ rpop key</li></ul></li></ul>	从右边弹出一个值; 返回值是取出的元素; <b>真正的从列表当中移除了取出的值;</b>
lindex <ul style="list-style-type: none"><li>◦ 语法</li></ul>	根据索引值取出一个key中包含的元素  参数说明:

	<ul style="list-style-type: none"> <li>▪ lindex key index</li> </ul>	<ul style="list-style-type: none"> <li>◦ key, 列表的key</li> <li>◦ index, 索引值, 从0开始数</li> <li>◦ 支持负向索引值,从-1</li> <li>◦ 如果超出了索引范围,则返回nil</li> </ul>
llen	<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ llen key</li> </ul> </li> </ul>	<p>获取列表的长度</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ key, 列表的key</li> </ul>
lrem	<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ lrem key count element</li> </ul> </li> </ul>	<p>删除列表当中的一个元素</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ count, 删除的个数</li> <li>◦ element, 元素</li> <li>◦ 对于count参数, 如果list有多个, 则可以根据count的数量进行相应的删除操作. 如果大于真实的个数, 则删除最大的个数.不会报错;</li> </ul>
ltrim	<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ ltrim key start stop</li> </ul> </li> </ul>	<p>截取列表当中的元素,这个操作会更改原来的列表. 返回值是操作结果, 成功或者失败; 截取的结果保存在新的列表中; 「<b>更改原来的列表,截取剩下的元素, 就没有了,只保留截取的元素</b>」 -&gt; 要的就是截取的元素;</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ key, key的值</li> <li>◦ start, 开始索引值</li> <li>◦ stop, 中止索引值</li> <li>▪ <b>start , stop为闭区间</b></li> </ul>
rpoplpush	<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ rpoplpush source dest</li> </ul> </li> </ul>	<p>移除列表的最后一个元素,并把这个移除的这个元素添加新的列表当中;</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ source, 这里是一个列表名称, 源列表名称</li> <li>◦ dest, 也是一个列表名称, 目标列表名称</li> <li>◦ rpoplpush sourceList destList           <ul style="list-style-type: none"> <li>▪ rpoplpush 操作的集合名称</li> <li>▪ sourceList, 从这个列表移除最后一个元素</li> <li>▪ destList, 把刚移除掉的元素添加这个列表当中;</li> <li>▪ 注意一下: sourceList和destList可以是同一个列表. 「自己从右边弹出一个元素,放到左边去;」</li> </ul> </li> </ul>
lset		根据指定索引值的值替换为另外一值

<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ lset key index elements</li> </ul> </li> </ul>	<p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ key, 列表的key</li> <li>◦ index, 第几个元素, 0, 表示第0个元素, 并不等同于索引值;</li> <li>◦ elements, 要设置新的元素的值</li> <li>◦ <b>重点强调</b> <ul style="list-style-type: none"> <li>▪ 如果列表不存在,则更新失败</li> <li>▪ 如果列表存在,但是超出了长度范围,则更新失败</li> <li>▪ 唯一的成功条件: <b>列表存在,且没有超出范围</b></li> </ul> </li> </ul>
<p><b>linsert</b></p> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ linsert key before after srcValue newValue</li> </ul> </li> </ul>	<p><b>在列表对应中某一个值的之前或者之后插入一个新值</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ key, 列表的key</li> <li>◦ before, 某值之前插入</li> <li>◦ after, 某值之后插入</li> <li>◦ srcValue, 列表当中的值</li> <li>◦ newValue, 插入的新值</li> </ul>

## • 示例代码参考

Bash
<pre> 1 ###### 2 # 在列表的头部/从列表的左边插入一值值; 3 # lpush key lelement [element ...] 4 127.0.0.1:6379&gt; keys * 5 (empty array) 6 127.0.0.1:6379&gt; lpush user tom 7 (integer) 1 8 127.0.0.1:6379&gt; lpush user jerry 9 (integer) 2 10 127.0.0.1:6379&gt; lpush user kate rose jack 11 (integer) 5 12 13 ##### 14 # 根据key从列表当中获取值; 15 # lrange key start stop 16 127.0.0.1:6379&gt; lrange user 0 -1 17 1) "jack" 18 2) "rose" 19 3) "kate" 20 4) "jerry" 21 5) "tom" 22 127.0.0.1:6379&gt; </pre>

```
23
24 #####从列表的右边添加值;
25 # 从列表的右边添加值;
26 # rpush key element [element ...]
27 127.0.0.1:6379> keys *
28 1) "user"
29 127.0.0.1:6379> lrange user 0 -1
30 1) "jack"
31 2) "rose"
32 3) "kate"
33 4) "jerry"
34 5) "tom"
35 127.0.0.1:6379> rpush user zhangsan lisi
36 (integer) 7
37 127.0.0.1:6379> lrange user 0 -1
38 1) "jack"
39 2) "rose"
40 3) "kate"
41 4) "jerry"
42 5) "tom"
43 6) "zhangsan" # 新添加的值
44 7) "lisi" # 新添加的值
45 127.0.0.1:6379>
46
47 #####
48 # 从左边取出一个值
49 # lpop key
50 127.0.0.1:6379> lrange user 0 -1 # 显示所有的值
51 1) "jack"
52 2) "rose"
53 3) "kate"
54 4) "jerry"
55 5) "tom"
56 6) "zhangsan"
57 7) "lisi"
58 127.0.0.1:6379> lpop user # 从左边取出一个值
59 "jack"
60 127.0.0.1:6379> lrange user 0 -1 # 取出是真的取出了,永久的取出了.
61 1) "rose"
62 2) "kate"
63 3) "jerry"
64 4) "tom"
65 5) "zhangsan"
66 6) "lisi"
67 127.0.0.1:6379> lpop user
68 "rose"
69 127.0.0.1:6379> lrange user 0 -1
70 1) "kate"
```

```
raven:5404 -> rpop user
71 2) "jerry"
72 3) "tom"
73 4) "zhangsan"
74 5) "lisi"
75 127.0.0.1:6379>
76
77 ######
78 # 从右边取出一个值
79 # rpop key
80 127.0.0.1:6379> lrange user 0 -1
81 1) "kate"
82 2) "jerry"
83 3) "tom"
84 4) "zhangsan"
85 5) "lisi"
86 127.0.0.1:6379> rpop user
87 "lisi"
88 127.0.0.1:6379> lrange user 0 -1
89 1) "kate"
90 2) "jerry"
91 3) "tom"
92 4) "zhangsan"
93 127.0.0.1:6379> rpop user
94 "zhangsan"
95 127.0.0.1:6379> lrange user 0 -1
96 1) "kate"
97 2) "jerry"
98 3) "tom"
99 127.0.0.1:6379>
100
101 #####
102 # 根据索引值取出列表当中的元素，索引值从0开始
103 # lindex key index
104 127.0.0.1:6379> lrange user 0 -1
105 1) "kate"
106 2) "jerry"
107 3) "tom"
108 127.0.0.1:6379> lindex user 0
109 "kate"
110 127.0.0.1:6379> lrange user 0 -1
111 1) "kate"
112 2) "jerry"
113 3) "tom"
114 127.0.0.1:6379> lindex user 1
115 "jerry"
116 127.0.0.1:6379> lindex user 2
117 "tom"
```

```
raven 118 127.0.0.1:6379> lindex user 3
raven 119 (nil)
raven 120 127.0.0.1:6379> lrange user 0 -1
raven 121 1) "kate"
raven 122 2) "jerry"
raven 123 3) "tom"
raven 124 127.0.0.1:6379>
raven 125
raven 126 #####
raven 127 # 获取列表的长度
raven 128 # llen key
raven 129 127.0.0.1:6379> lrange user 0 -1
raven 130 1) "kate"
raven 131 2) "jerry"
raven 132 3) "tom"
raven 133 127.0.0.1:6379> llen user # 获取列表的长度
raven 134 (integer) 3
raven 135 127.0.0.1:6379>
raven 136
raven 137 #####
raven 138 # 删除列表当中的元素
raven 139 # lrem key count element
raven 140 127.0.0.1:6379> lrange user 0 -1
raven 141 1) "polly"
raven 142 2) "kate"
raven 143 3) "jerry"
raven 144 4) "tom"
raven 145 5) "kate"
raven 146 6) "jerry"
raven 147 7) "tom"
raven 148 127.0.0.1:6379> lrem user 1 polly
raven 149 (integer) 1
raven 150 127.0.0.1:6379> lrange user 0 -1 # 有多个，也会只删除一个；
raven 151 1) "kate"
raven 152 2) "jerry"
raven 153 3) "tom"
raven 154 4) "kate"
raven 155 5) "jerry"
raven 156 6) "tom"
raven 157 127.0.0.1:6379> lrem user 2 jerry # 如果存在多个，则根据指定的删除数量进行删除操作
raven 158 (integer) 2
raven 159 127.0.0.1:6379> lrange user 0 -1
raven 160 1) "kate"
raven 161 2) "tom"
raven 162 3) "kate"
raven 163 4) "tom"
raven 164 127.0.0.1:6379>
raven 165
```

```
raven 166 #####  
raven 167 # 截取操作  
raven 168 # ltrim key index stop  
raven 169 127.0.0.1:6379> keys *  
raven 170 1) "user"  
raven 171 127.0.0.1:6379> lrange user 0 -1  
raven 172 1) "tianxiaoqi"  
raven 173 2) "maxiaoliu"  
raven 174 3) "wangexiaowu"  
raven 175 4) "zhangxiaosi"  
raven 176 5) "zhangxiaosan"  
raven 177 6) "zhangxiaosan"  
raven 178 7) "kate"  
raven 179 8) "tom"  
raven 180 9) "kate"  
raven 181 10) "tom"  
raven 182 127.0.0.1:6379> ltrim user 0 3 # 从索引0开始, 到3结束, 包括0和3, 一共四个元素;  
raven 183 OK # 返回操作结果是否成功。  
raven 184 127.0.0.1:6379> lrange user 0 -1 # key = user的列表中的元素就是保存的截取结果  
raven 185 1) "tianxiaoqi"  
raven 186 2) "maxiaoliu"  
raven 187 3) "wangexiaowu"  
raven 188 4) "zhangxiaosi"  
raven 189 127.0.0.1:6379>  
raven 190  
raven 191 #####  
raven 192 # 把取出的元素添加到另外一个列表当中, 即取出列表的最后一个元素并把它添加到新的列表当中  
raven 193 # rpoplpush,  
raven 194 127.0.0.1:6379> lrange user 0 -1 # 原来的列表  
raven 195 1) "tianxiaoqi"  
raven 196 2) "maxiaoliu"  
raven 197 3) "wangexiaowu"  
raven 198 4) "zhangxiaosi"  
raven 199 5) "tom"  
raven 200 6) "kate"  
raven 201 7) "jerry"  
raven 202 8) "rose"  
raven 203 9) "rose"  
raven 204 10) "jack"  
raven 205 11) "jerry"  
raven 206 12) "tom"  
raven 207 13) "kate"  
raven 208 127.0.0.1:6379> rpoplpush user newuser # 取出最后一个, kate, 并且把它添加到key = newuser 的列表中  
raven 209 # 相当于: rpop user, 取出kate, lpush newuser kate两句话的作用;  
raven 210 "kate" # 返回操作的元素  
raven 211 127.0.0.1:6379> keys * # 多了一个key, newuser, 保存的就是kate  
raven 212 1) "newuser"
```

```
raven 5404 212 +) newuser
213 2) "user"
214 127.0.0.1:6379> lrange user 0 -1 # 瞅一眼, 少了kate, 表示被取出来了
215 1) "tianxiaoqi"
216 2) "maxiaoliu"
217 3) "wangexiaowu"
218 4) "zhangxiaosi"
219 5) "tom"
220 6) "kate"
221 7) "jerry"
222 8) "rose"
223 9) "rose"
224 10) "jack"
225 11) "jerry"
226 12) "tom"
227 127.0.0.1:6379> lrange newuser 0 -1 # 瞄一眼, 多了一个kate, 表示存上了;
228 1) "kate"
229 127.0.0.1:6379>
230
231 ######
232 # 根据索引值修改列表中的元素
233 # lset key index value
234 127.0.0.1:6379> lrange user 0 -1
235 1) "tianxiaoqi"
236 2) "maxiaoliu"
237 3) "wangexiaowu"
238 4) "zhangxiaosi"
239 5) "tom"
240 6) "kate"
241 7) "jerry"
242 8) "rose"
243 9) "rose"
244 10) "jack"
245 11) "jerry"
246 12) "tom"
247 127.0.0.1:6379> lset user 0 ergouzi # 修改第0个元素为二狗子
248 OK
249 127.0.0.1:6379> lrange user 0 -1
250 1) "ergouzi"
251 2) "maxiaoliu"
252 3) "wangexiaowu"
253 4) "zhangxiaosi"
254 5) "tom"
255 6) "kate"
256 7) "jerry"
257 8) "rose"
258 9) "rose"
259 10) "jack"
```

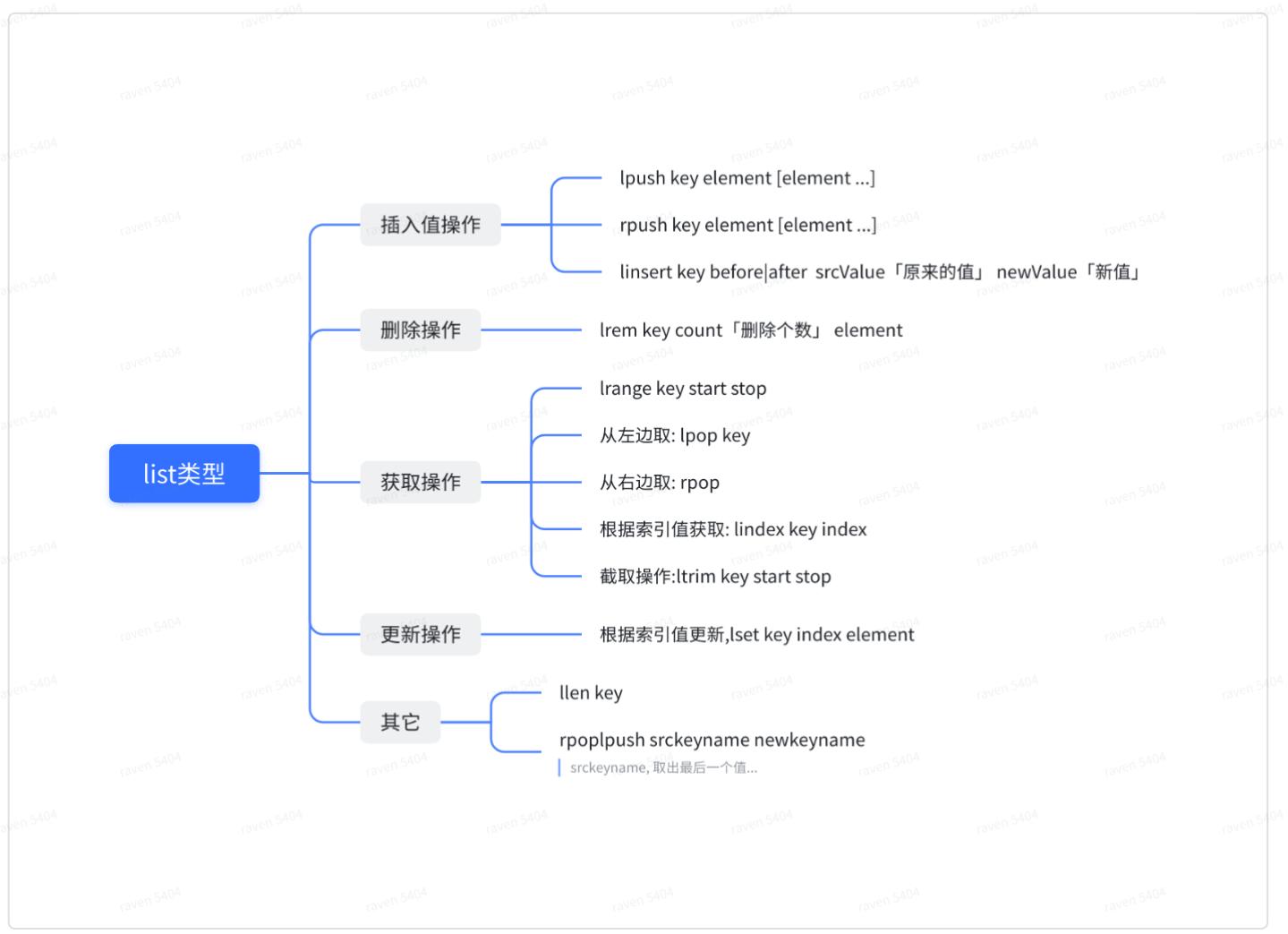
```
raven 260 11) "jerry"
raven 261 12) "tom"
raven 262 127.0.0.1:6379> lset nouser 0 meiyoukey # key不存在,报错
raven 263 (error) ERR no such key
raven 264 127.0.0.1:6379> lset user 100 chaochule # 索引值超出了,则报错
raven 265 (error) ERR index out of range
raven 266 127.0.0.1:6379>
raven 267
raven 268 #####
raven 269 # 在某一个元素之前或者之后插入一个值
raven 270 # linsert key before/after value insertValue
raven 271 127.0.0.1:6379> lrange user 0 -1
raven 272 1) "ergouzi"
raven 273 2) "maxiaoliu"
raven 274 3) "wangexiaowu"
raven 275 4) "zhangxiaosi"
raven 276 5) "tom"
raven 277 6) "kate"
raven 278 7) "jerry"
raven 279 8) "rose"
raven 280 9) "rose"
raven 281 10) "jack"
raven 282 11) "jerry"
raven 283 12) "tom"
raven 284 127.0.0.1:6379> linsert user before jack insertjack # 在jack前边插入一个新值;
raven 285 (integer) 13
raven 286 127.0.0.1:6379> lrange user 0 -1
raven 287 1) "ergouzi"
raven 288 2) "maxiaoliu"
raven 289 3) "wangexiaowu"
raven 290 4) "zhangxiaosi"
raven 291 5) "tom"
raven 292 6) "kate"
raven 293 7) "jerry"
raven 294 8) "rose"
raven 295 9) "rose"
raven 296 10) "insertjack"
raven 297 11) "jack"
raven 298 12) "jerry"
raven 299 13) "tom"
raven 300 127.0.0.1:6379> linsert user after tom firsttomafter # 在tom后边插入一个新值, 如果有多个tom,则第一次出现的tom后边插入新值;
raven 301 (integer) 14
raven 302 127.0.0.1:6379> lrange user 0 -1
raven 303 1) "ergouzi"
raven 304 2) "maxiaoliu"
raven 305 3) "wangexiaowu"
raven 306 4) "zhangxiaosi"
```

```

raven 307 5) "tom"
raven 308 6) "firsttomafter"
raven 309 7) "kate"
raven 310 8) "jerry"
raven 311 9) "rose"
raven 312 10) "rose"
raven 313 11) "insertjack"
raven 314 12) "jack"
raven 315 13) "jerry"
raven 316 14) "tom"
raven 317 127.0.0.1:6379>

```

## ● 总结



## 4. 集合「set」

set里的值是不能重复的且没有顺序的存放的;跟java当中的set集合差不多;

set命令一般都使用s开头,表示是一个set命令;

命令名称	描述
<code>sadd</code>	向集合中添加一个或者多个元素.

<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ sadd key member [member ...]</li> </ul> </li> </ul>	<p><b>如果添加的member已经存在了,即添加重复的元素,则不会添加,操作结果返回为0; 正确添加,操作结果返回1;</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ key, 集合当中的key</li> <li>◦ member, 元素</li> </ul>
<b>smembers</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ smembers key</li> </ul> </li> </ul>	<p><b>获取集合当中的元素</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ key, 集合当中的key</li> </ul>
<b>sismember key member</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ sismembers key member</li> </ul> </li> </ul>	<p><b>判断某一个成员是否存在于集合当中</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ key, 集合当中的key</li> <li>◦ member, 要判断成员的值</li> </ul>
<b>scard</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ scard key</li> </ul> </li> </ul>	<p><b>获取集合当中元素的个数</b></p>
<b>srem</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ srem key member [member ...]</li> </ul> </li> </ul>	<p><b>删除集合当中一个或者多个元素;</b></p> <p><b>如果不存在的元素,则不删除,但是也不会报错;只会删除集合当中存在的元素;</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ member, 元素</li> </ul>
<b>srandmember</b> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ srandmember key [count]</li> </ul> </li> </ul>	<p><b>从集合当中随机取出一个或者多个元素. 「random member」</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ key, 集合的key</li> <li>◦ [count], 可选参数, 表示要取出元素的个数</li> <li>◦ <b>count 是正数的情况下</b> <ul style="list-style-type: none"> <li>▪ <b>count &lt; 总长度, 此时会随机取出足够数据的value</b></li> <li>▪ <b>count &gt; 总长度, 只会取出最大的长度数量的数据</b></li> </ul> </li> <li>◦ <b>count是负数和情况下</b> <ul style="list-style-type: none"> <li>▪ <b>首先会保证取出的数量</b></li> <li>▪ <b>count &lt; 总长度, 则取出的数据不重复</b></li> <li>▪ <b>count &gt; 总长度, 取出的数据会重复</b></li> </ul> </li> <li>◦ <b>count = 0</b> <ul style="list-style-type: none"> <li>▪ <b>不返回内容</b></li> </ul> </li> </ul>
<b>spop</b> <ul style="list-style-type: none"> <li>◦ 语法</li> </ul>	<p><b>随机从集合当中取出一个或者多个元素.</b></p> <p><b>弹出了, 真正从原来集合当中删除了.修改了原来的集合;</b></p>

<ul style="list-style-type: none"> <li>▪ spop key [count]</li> </ul>	<p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ [count], 可选参数, 表示要取出元素的个数</li> </ul>
<p><b>smove</b></p> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ smove source destination member</li> </ul> </li> </ul>	<p>把一个集合中的值移到另外一个集合当中去;</p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ source, 要从哪个集合当中移出</li> <li>◦ destination, 要把这个值移动到哪个集合当中去.</li> <li>◦ member, 要从source集合当中移哪个值;</li> </ul>

<b>Bash</b>	<pre> 1 127.0.0.1:6379&gt; keys * 2 (empty array) 3 # 添加操作, 添加多个值 4 127.0.0.1:6379&gt; sadd user tom jack lucy polly lilei rose 5 (integer) 6 6 # 添加一个值 7 127.0.0.1:6379&gt; sadd user zhansan 8 (integer) 1 9 # 查看所有的元素 10 11 127.0.0.1:6379&gt; smembers user 12 1) "polly" 13 2) "lucy" 14 3) "zhansan" 15 4) "tom" 16 5) "rose" 17 6) "lilei" 18 7) "jack" 19 # 随机取出两个 20 127.0.0.1:6379&gt; srandmember user 2 21 1) "lucy" 22 2) "polly" 23 127.0.0.1:6379&gt; smembers user 24 1) "zhansan" 25 2) "tom" 26 3) "rose" 27 4) "lilei" 28 5) "jack" 29 6) "lucy" 30 7) "polly" 31 # 查看集合长度 32 127.0.0.1:6379&gt; scard user 33 (integer) 7 34 # 随机取出3个, spop修改了原来的集合. 35 127.0.0.1:6379&gt; spop user 3 </pre>
-------------	---

```

35 127.0.0.1:6379> spop user 3
36 1) "zhansan"
37 2) "lucy"
38 3) "jack"
39 127.0.0.1:6379>
40 # 从key=user集合中取出rose, 再把它存入到key=newuser对应的集合当中;
41 127.0.0.1:6379> smove user newuser rose
42 (integer) 1
43 127.0.0.1:6379> smembers user # 查看一下
44 1) "tom"
45 2) "lilei"
46 3) "polly"
47 127.0.0.1:6379> smembers newuser # 查看新的集合中的元素
48 1) "rose"
49 127.0.0.1:6379> sismember user tom # 判断tom是否存在于key=user的集合当中; 存在则返回1, 不存在则返回0;
50 (integer) 1
51 # 判断tom2是否存在于key=user的集合当中; 存在则返回1, 不存在则返回0;
52 127.0.0.1:6379> sismember user tom2
53 (integer) 0
54 # 从key = user集合当中删除掉lilei, 成功返回1, 失败返回0;
55 127.0.0.1:6379> srem user lilei
56 (integer) 1
57 127.0.0.1:6379> srem user lilei
58 (integer) 0
59 127.0.0.1:6379>

```

## • 数字集合常用命令

命令名称	描述
sdiff	<p><b>差集合</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ sdiff s1 s2</li> </ul> </li> </ul>
sinter	<p><b>交集</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ sinter s1 s2</li> </ul> </li> </ul>
sunion	<p><b>并集</b></p> <p><b>参数说明:</b></p> <ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ sunion</li> </ul> </li> </ul>

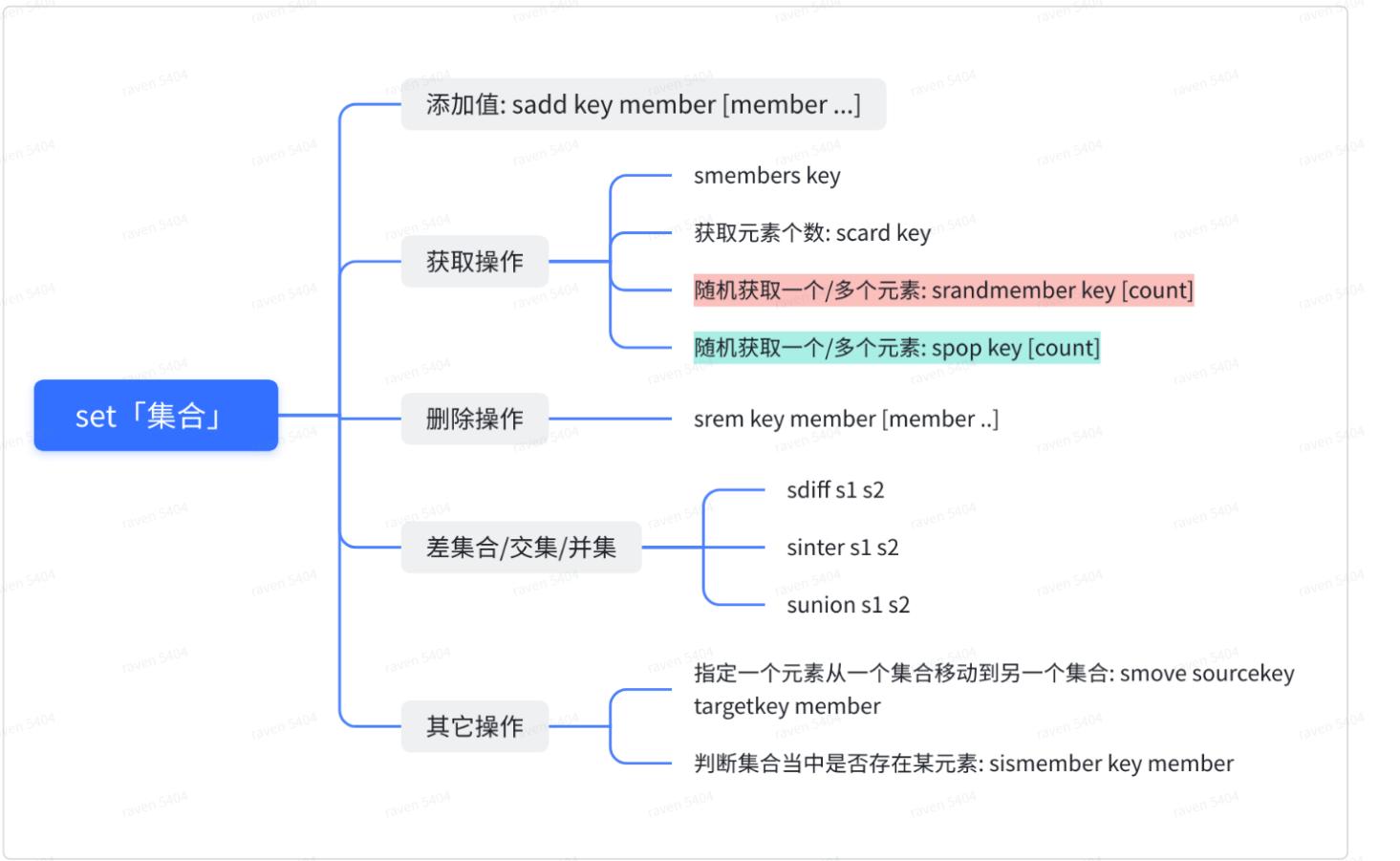
◦ s2, 集合2

Bash

```
1 127.0.0.1:6379> flushdb
2 OK
3 127.0.0.1:6379>
4 # 创建一个集合并指定它的元素
5 127.0.0.1:6379> sadd s1 tom jerry jack 1      2 3 4 5
6 (integer) 8
7 # 创建一个集合并指定它的元素
8 127.0.0.1:6379> sadd s2 jerry jack rose 3 4 5 6 7
9 (integer) 8
10 # 查看一下key = s1的集合
11 127.0.0.1:6379> smembers s1
12 1) "jerry"
13 2) "1"
14 3) "5"
15 4) "tom"
16 5) "jack"
17 6) "3"
18 7) "2"
19 8) "4"
20 # 查看一下key = s2的集合
21 127.0.0.1:6379> smembers s2
22 1) "jerry"
23 2) "5"
24 3) "rose"
25 4) "6"
26 5) "jack"
27 6) "3"
28 7) "7"
29 8) "4"
30 # 求集合的差值, s1 - s2
31 127.0.0.1:6379> sdiff s1 s2
32 1) "1"
33 2) "tom"
34 3) "2"
35 # 求集合的差值, s2 - s1
36 127.0.0.1:6379> sdiff s2 s1
37 1) "6"
38 2) "rose"
39 3) "7"
40 # 求集合的交集 s1∩s2
41 127.0.0.1:6379> sinter s1 s2
42 1) "jerry"
43 2) "5"
```

```
raven:5404 ~ - - - - -  
44) "jack"  
45) "3"  
46) "4"  
47) "# 求集合的交集 s2∩s1  
48) 127.0.0.1:6379> sinter s2 s1  
49) 1) "jerry"  
50) 2) "5"  
51) 3) "jack"  
52) 4) "3"  
53) 5) "4"  
54) "# 求集合的并集 s1∪s2  
55) 127.0.0.1:6379> sunion s1 s2  
56) 1) "1"  
57) 2) "5"  
58) 3) "6"  
59) 4) "jack"  
60) 5) "jerry"  
61) 6) "tom"  
62) 7) "rose"  
63) 8) "3"  
64) 9) "2"  
65) 10) "7"  
66) 11) "4"  
67) 127.0.0.1:6379>
```

## ● 总结



## 5. 有序集合「zset」

**有序, 不可重复;跳表实现**

ZSet其实是在Set的基础上绑定了一个score来实现集合数据按照score排序的集合;

**有序集合和集合一样也是String类型元素的集合, 且不可以重复;**

每一个元素都会关联一个double类型的Score. redis通过score为集合中的成员进行从小到大排序的;

**zset命令一般都是以z开头的.**

- 相关命令

命令名称	描述
<b>zadd</b> <ul style="list-style-type: none"> <li>○ 语法           <ul style="list-style-type: none"> <li>▪ zadd key [nx xx] [ch] [incr]</li> <li>score member [score member]</li> <li>▪ <b>sadd key member1 member2</b></li> </ul> </li> </ul>	向有序集合中添加一个或多个score/member对 返回的是添加元素的个数; 参数说明: <ul style="list-style-type: none"> <li>○ key, zset中的key</li> <li>○ nx: 不更新存在的成员。只添加新成员。「新增操作」</li> <li>○ xx: 仅仅更新存在的成员, 不添加新成员。「更新操作」</li> </ul>

- **ch:** 修改返回值为发生变化的成员总数，原始是返回新添加成员的总数 (CH 是 *changed* 的意思)。更改的元素是新添加的成员，已经存在的成员更新分数。所以在命令中指定的成员有相同的分数将不被计算在内。注：在通常情况下，**ZADD** 返回值只计算新添加成员的数量。简单来说：原来返回值是根据添加的个数，现在通过「**ch**」修改为**score**更改的个数。
- **incr:** 当 **ZADD** 指定这个选项时，成员的操作就等同**ZINCRBY**命令，对成员的分数进行递增操作
- **score:** 设置**score**, 排序的时候会使用此参数
- **member:** member的值;
- **[score member]:** 可选参数,可以同时设置多个值;

## zrange

- 语法
  - **zrange key start stop [withscores]**
  - **zrange student 0 30 byscore**
    - 0, 30表示**score**的值, 按照**Score**进行取值, 此时没有写(, 表示闭区间. [0, 30]
    - **zrange student (0 (30 byscore**, 0和30表示的还是**score**的值,表示开区间
  - **ZRANGE student 0 1024 byscore limit 5 5** , 此处必须搭建**byscore**一块用. **limit** 索引值 个数,
    - 0 1024 表示**score**的值;

获取指定范围内zset当中的对应key的内容.排好序的, 根据Score升序排序;

参数说明:

- **start**, 开始索引值从0开始.
- **stop**, 结束索引值, 如果是-1,则表示取到末尾
- 闭区间[**start**, **stop**]
- **[withscores]**, 可选参数, 是否显示**score**值;

## zrevrange

- 语法
  - **zrevrange key start stop**

获取指定范围内zset当中的对应key的内容.和**zrange**正好相反, 它是倒序排序,即降序排列;

参数说明:

- **start**, 开始索引值从0开始.
- **stop**, 结束索引值, 如果是-1,则表示取到末尾
- 闭区间[**start**, **stop**]

## zrangebyscore

- 语法
  - **zrangebyscore key min max [withscores] [limit offset**

返回**score**「权重」在**min**和**max**「闭区间」之间的成员

参数说明:

- **min**, **score**最小值
- **max**, **score**最大值

	<ul style="list-style-type: none"> <li>◦ count]</li> </ul>	<ul style="list-style-type: none"> <li>◦ 闭区间[min,max]</li> <li>◦ 可以选择是开区间还是闭区间</li> <li>◦ zrange score key (min max) <ul style="list-style-type: none"> <li>▪ 哪也不加, 就是闭区间, 也是默认值;</li> <li>▪ min是开区间, max是闭区间, (min max)</li> <li>▪ (min (max, 这两个都是开区间</li> <li>▪ limit 偏移量 取的个数</li> </ul> </li> </ul>
zcount	<ul style="list-style-type: none"> <li>◦ 某一Score内元素个数;</li> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zcount key min max 「score的值」</li> </ul> </li> </ul>	<p>统计集合中分数在min和max之间的元素个数</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ min, score最小值</li> <li>◦ max, score最大值</li> <li>◦ 闭区间[min,max]</li> <li>◦ 可以指定是开区间还是闭区间. 使用(小括号的方式指定;</li> </ul>
zscore	<ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zscore key member</li> </ul> </li> </ul>	<p>返回成员的score值</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ key</li> <li>◦ member,要返回的成员名称</li> </ul>
zcard	<ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zcard key</li> </ul> </li> </ul>	<p>返回集合当中的元素个数</p>
zrank	<ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zrank key member</li> </ul> </li> </ul>	<p>获取元素在集合中的排名, 从小到大, 最小的是0</p> <p>从小到大排序</p>
zrevrank	<ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zrevrank key member</li> </ul> </li> </ul>	<p>获取元素在集合中的排序, 从大到小</p> <p>从大到小排序</p>
zincrby	<ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ ZINCRBY key increment member</li> </ul> </li> </ul>	<p>给元素增加score, 如果不存在就新创建元素, 并赋予对应的score值</p> <p>如果存在此元素,则直接添加新的值;</p>
<b>zinterstore</b>	<ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zinterscore destinaltion numkeys key [key ... ]</li> </ul> </li> </ul>	<p>计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ destinaltion, 要存储交集结果的新集合</li> <li>◦ numkeys, 有几个有序集合取交集</li> </ul>

	<ul style="list-style-type: none"> <li>◦ key, 有序集合的key</li> <li>◦ [key ...] 有序集合的key, 可选参数</li> </ul>
zrem	<p>从集合中弹出一个元素</p> <ul style="list-style-type: none"> <li>◦ 语法 <ul style="list-style-type: none"> <li>▪ zrem key member [member ...]</li> </ul> </li> </ul>
zlexcount	<p>计算有序集合中指定字典区间内成员数量</p> <ul style="list-style-type: none"> <li>◦ 说明: <ul style="list-style-type: none"> <li>◦ 成员名称前需要加 [ 符号作为开头, [ 符号与成员之间不能有空格</li> <li>◦ <b>可以使用 - 和 + 表示得分最小值和最大值</b></li> <li>◦ min 和 max 不能反, max 放前面 min 放后面会导致返回结果为0</li> <li>◦ 计算成员之间的成员数量时,参数 min 和 max 的位置也计算在内。 <b>闭区间;</b></li> <li>◦ min 和 max 参数的含义与 zrangebylex 命令中所描述的相同</li> </ul> </li> </ul>
zrangebylex	<p>返回指定成员区间内的成员, 按成员字典倒序排序, 分数必须相同。 <b>获取指定区间的元素, 分数必须相同</b></p> <p><b>返回值: 指定成员范围的元素列表。</b></p> <p><b>说明:</b></p> <ul style="list-style-type: none"> <li>◦ “-” 和 “+” 表示得分最小值和最大值</li> <li>◦ 分数必须相同! 如果有序集合中的成员分数有不一致的,返回的结果就不准。</li> <li>◦ 成员字符串作为二进制数组的字节数进行比较。</li> <li>◦ 默认是以ASCII字符集的顺序进行排列。如果成员字符串包含utf8这类字符集的内容,就会影响返回结果,所以建议不要使用。</li> <li>◦ 源码中采用C语言中 memcmp() 函数, 从字符的第0位到最后一位进行排序,如果前面部分相同,那么较长的字符串比较短的字符串排序靠前。</li> <li>◦ 默认情况下, “max” 和 “min” 参数前必须加 “[” 符号作为开头。”[” 符号与成员之间不能有空格, 返回成员结果集会包含参数 “max” 和 “min”</li> <li>◦ “max” 和 “min” 参数前可以加 “(“ 符号作为开头表示小于, “(“ 符号与成员之间不能有空格。返回成员结果集不会包含 “max” 和 “min” 成员。</li> <li>◦ 可以使用 “-” 和 “+” 表示得分最小值和最大值</li> </ul>

- “max” 和 “min” 不能反，“max” 放后面 “min” 放前面会导致返回结果为空
- 与 ZREVRANGEBYLEX 获取顺序相反的指令是 [ZREVRANGEBYLE](#)

## ● 参考示例代码

Bash

```
1 127.0.0.1:6379> flushall
2 OK
3 127.0.0.1:6379>
4 # 添加元素
5 127.0.0.1:6379> zadd stu 20 tom 1 jack 43 rose 9 jerry 3 lucy
6 (integer) 5
7 # 根据范围进行查看集合当中的数据
8 127.0.0.1:6379> zrange stu 0 -1
9 1) "jack"
10 2) "lucy"
11 3) "jerry"
12 4) "tom"
13 5) "rose"
14 # 查看某一个元素的score「权重」
15 127.0.0.1:6379> zscore stu lucy
16 "3"
17 # 反向查看集合当中的数据
18 127.0.0.1:6379> zrevrange stu 0 -1
19 1) "rose"
20 2) "tom"
21 3) "jerry"
22 4) "lucy"
23 5) "jack"
24 # 查看集合当中的元素个数
25 127.0.0.1:6379> zcard stu
26 (integer) 5
27 # 查看限定的范围的元素个数
28 127.0.0.1:6379> zcount stu 0 10
29 (integer) 3
30 # 获取score范围内的成员
31 127.0.0.1:6379> zrangebyscore stu 0 10
32 1) "jack"
33 2) "lucy"
34 3) "jerry"
35 # 获取score范围内的成员，同时查看所对应的score, 此时是闭区间，score包括0和10
36 127.0.0.1:6379> zrangebyscore stu 0 10 withscores
37 1) "jack"
38 2) "1"
```

```
50 4) "lucy"
39 3) "lucy"
40 4) "3"
41 5) "jerry"
42 6) "9"
43 # 获取score范围内的成员，同时查看所对应的score,此时是闭区间，score不包括1和9
44 # zrangebyscore key min max withscores
45 127.0.0.1:6379> zrangebyscore stu (1 (10 withscores
46 1) "lucy"
47 2) "3"
48 127.0.0.1:6379>
49 # zrank, 获取成员在集合当中的排名，默认是从小到大排，最小值是0
50 127.0.0.1:6379> zrange stu 0 -1 withscores
51 1) "jack"
52 2) "1"
53 3) "lucy"
54 4) "3"
55 5) "jerry"
56 6) "9"
57 7) "tom"
58 8) "20"
59 9) "rose"
60 10) "43"
61 127.0.0.1:6379> zrank stu lucy # 获取成员排名
62 (integer) 1
63 127.0.0.1:6379> zrank stu tom # 获取成员排名
64 (integer) 3
65 127.0.0.1:6379>
66 # zrevrank, 获取成员在集合当中的排名，倒序排列。从大到小排列;
67 127.0.0.1:6379> zrevrank stu tom
68 (integer) 1
69 127.0.0.1:6379> zrevrank stu rose
70 (integer) 0
71 127.0.0.1:6379>
72 127.0.0.1:6379> zrange stu 0 -1
73 1) "jack"
74 2) "lucy"
75 3) "jerry"
76 4) "tom"
77 5) "rose"
78
79 127.0.0.1:6379> zrange stu 1 -1 withscores
80 1) "lucy"
81 2) "3"
82 3) "jerry"
83 4) "9"
84 5) "tom"
85 6) "20" # 没有自增之前是20
```

```
raven 86 7) "rose"
raven 87 8) "43"
raven 88 # 给tom的score增加了10
raven 89 127.0.0.1:6379> zincrby stu 10 tom
raven 90 "30"
raven 91 127.0.0.1:6379> zrange stu 1 -1 withscores
raven 92 1) "lucy"
raven 93 2) "3"
raven 94 3) "jerry"
raven 95 4) "9"
raven 96 5) "tom" # zincrby 之后score更改为了30 = 20 + 10
raven 97 6) "30"
raven 98 7) "rose"
raven 99 8) "43"
raven 100 127.0.0.1:6379>
raven 101 # zincrby key increment withscores
raven 102 # 如果increment是负值,则表示相关. 也就是自减操作;
raven 103 127.0.0.1:6379> zincrby stu -5 tom
raven 104 "25"
raven 105 127.0.0.1:6379> zrange stu 1 -1 withscores
raven 106 1) "lucy"
raven 107 2) "3"
raven 108 3) "jerry"
raven 109 4) "9"
raven 110 5) "tom"
raven 111 6) "25" # 更改成了: 30 + (-5) = 25
raven 112 7) "rose"
raven 113 8) "43"
raven 114 127.0.0.1:6379>
raven 115 # zinterstore dest count key1 key2 [key ....]
raven 116 # 计算给定的一个或者多个有序的集合的交集,并将结果存储在新的集合当中;
raven 117 127.0.0.1:6379> zrange z1 0 -1
raven 118 1) "lucy"
raven 119 2) "tom"
raven 120 3) "jack"
raven 121 127.0.0.1:6379> zrange z2 0 -1
raven 122 1) "lucy"
raven 123 2) "tom"
raven 124 3) "jerry"
raven 125 4) "raven"
raven 126 127.0.0.1:6379> zinterstore z5 2 z1 z2
raven 127 (integer) 2
raven 128 127.0.0.1:6379> zrange z5 0 -1
raven 129 1) "lucy"
raven 130 2) "tom"
raven 131 127.0.0.1:6379>
raven 132 # 从集合中删除成员
raven 133 # zrem key member [member ...]
```

```
raven 5404 134 127.0.0.1:6379> zrange stu 0 -1
raven 5404 135 1) "jack"
raven 5404 136 2) "lucy"
raven 5404 137 3) "jerry"
raven 5404 138 4) "tom"
raven 5404 139 5) "rose"
raven 5404 140 127.0.0.1:6379> zrem stu jack tom
raven 5404 141 (integer) 2
raven 5404 142 127.0.0.1:6379> zrange stu 0 -1
raven 5404 143 1) "lucy"
raven 5404 144 2) "jerry"
raven 5404 145 3) "rose"
raven 5404 146 127.0.0.1:6379>
raven 5404 147 # ZLEXCOUNT 命令用于计算有序集合中指定成员之间的成员数量。
raven 5404 148 # zlexcount key
raven 5404 149 127.0.0.1:6379> ZADD myzset 1 a 2 b 3 c 4 d 5 e 6 f 7 g
raven 5404 150 (integer) 7
raven 5404 151 127.0.0.1:6379> ZRANGE myzset 0 -1
raven 5404 152 1) "a"
raven 5404 153 2) "b"
raven 5404 154 3) "c"
raven 5404 155 4) "d"
raven 5404 156 5) "e"
raven 5404 157 6) "f"
raven 5404 158 7) "g"
raven 5404 159 127.0.0.1:6379> ZLEXCOUNT myzset - +
raven 5404 160 (integer) 7
raven 5404 161 127.0.0.1:6379> ZLEXCOUNT myzset [c +
raven 5404 162 (integer) 5
raven 5404 163 127.0.0.1:6379> ZLEXCOUNT myzset - [c
raven 5404 164 (integer) 3
raven 5404 165 127.0.0.1:6379>
raven 5404 166 # RANGEBYLEX 返回指定成员区间内的成员，按成员字典倒序排序 score必须相同。
raven 5404 167 # score必须相同。
raven 5404 168 # score必须相同。
raven 5404 169 # score必须相同。
raven 5404 170 # score必须相同。
raven 5404 171 127.0.0.1:6379> zadd zset 0 a 0 aa 0 abc 0 apple 0 b 0 c 0 d 0 d1 0 dd 0
raven 5404 dobble 0 z 0 z1
raven 5404 172 (integer) 12
raven 5404 173 127.0.0.1:6379> ZRANGEBYLEX zset - +
raven 5404 174 1) "a"
raven 5404 175 2) "aa"
raven 5404 176 3) "abc"
raven 5404 177 4) "apple"
raven 5404 178 5) "b"
raven 5404 179 6) "c"
raven 5404 180 7) "d"
```

```
raven 180 1) "a" raven 5404
raven 181 8) "d1" raven 5404
raven 182 9) "dd" raven 5404
raven 183 10) "dobble" raven 5404
raven 184 11) "z" raven 5404
raven 185 12) "z1" raven 5404
raven 186 # 获取分页数据 raven 5404
raven 187 127.0.0.1:6379> ZRANGEBYLEX zset - + limit 0 3 raven 5404
raven 188 1) "a" raven 5404
raven 189 2) "aa" raven 5404
raven 190 3) "abc" raven 5404
raven 191 127.0.0.1:6379> ZRANGEBYLEX zset - + limit 3 3 raven 5404
raven 192 1) "apple" raven 5404
raven 193 2) "b" raven 5404
raven 194 3) "c" raven 5404
raven 195 127.0.0.1:6379> ZRANGEBYLEX zset - + limit 6 3 raven 5404
raven 196 1) "d" raven 5404
raven 197 2) "d1" raven 5404
raven 198 3) "dd" raven 5404
raven 199 127.0.0.1:6379> raven 5404
```

## 6. geospatial 「地理空间」 「了解一下, 知道有这个数据类型即可」

参考文档: <http://redis.cn/commands/geoadd.html>

## 7. HyperLoglogs

- 以下文档, 通读一次;

文档: <https://thoughtbot.com/blog/hyperloglogs-in-redis#how-to-use-hyperloglogs-in-redis>

基数: 数学上集合的元素个数, 是不能重复的。

UV (Unique visitor) : 是指通过互联网访问、浏览这个网页的自然人。访问的一个电脑客户端为一个访客, 一天内同一个访客仅被计算一次。

Redis 2.8.9 版本更新了 hyperloglog 数据结构, 是基于基数统计的算法。

hyperloglog 的优点是占用内存小, 并且是固定的。存储  $2^{64}$  个不同元素的基数, 只需要 12 KB 的空间。但是也可能有 0.81% 的错误率。

这个数据结构常用于统计网站的 UV。传统的方式是使用 set 保存用户的 ID, 然后统计 set 中元素的数量作为判断标准。

但是这种方式保存了大量的用户 ID, ID 一般比较长, 占空间, 还很麻烦。我们的目的是计数, 不是保存数据, 所以这样做有弊端。但是如果使用 hyperloglog 就比较合适了。

- 命令

命令名称	描述
pfadd ◦ 语法 ▪ pfadd key element [element ...]	添加元素 参数说明: ◦ element, 添加的元素 ◦ [element ...], 添加元素, 可选参数
pfcount ◦ 语法 ▪ pfcount key	统计元素个数
pfmerge ◦ 语法 ▪ pfmerge destkey sourcekey [sourcekey ...]	合并操作,也就是取多个key的并集操作 参数说明: ◦ destkey, 将取并集之后的结果存储到新的key中的key的名字 ◦ sourcekey, 要取并集的key ◦ [sourcekey ...], 可选参数, 要取并集的key

```

127.0.0.1:6379> pfadd mykey a b c d e f g h 1 1 2 3 4 5 6
(integer) 1
127.0.0.1:6379> pfcount mykey   统计元素个数
(integer) 14
127.0.0.1:6379> pfadd mykey2 231 2 43 1 4 5 2 1 243 4  添加元素
(integer) 1
127.0.0.1:6379> pfmerge combo mykey mykey2    合并操作， 也就是并集操作
OK
127.0.0.1:6379> pfcount combo
(integer) 17
127.0.0.1:6379> 
```

**添加元素**

**添加元素**

**合并操作， 也就是并集操作**

**pfmerge 全并之后的名称 要合并的名称1，要合并的名称2。。。。**

**做为统计数据时使用，只占12K这空间。**

## Bash

```
1 127.0.0.1:6379> pfadd mykey a b c d e f g h i j # 创建第一组元素(integer) 1
2 127.0.0.1:6379> PFCOUNT mykey      # 统计 mykey 基数(integer) 10
3 127.0.0.1:6379> PFADD mykey2 i j z x c v b n m # 创建第二组元素(integer) 1
4 127.0.0.1:6379> PFCOUNT mykey2      # 统计 mykey2 基数(integer) 9
5 127.0.0.1:6379> PFMERGE mykey3 mykey mykey2 # 合并两组 mykey mykey2 =>
mykey3OK127.0.0.1:6379> PFCOUNT mykey3
6 (integer) 15
7 127.0.0.1:6379>
```

**错误率: 0.81%**

- 统计注册 IP 数
- 统计每日访问 IP 数
- 统计页面实时 UV 数
- 统计在线用户数
- 统计用户每天搜索不同词条的个数

## 8. bitmaps

重要内容:

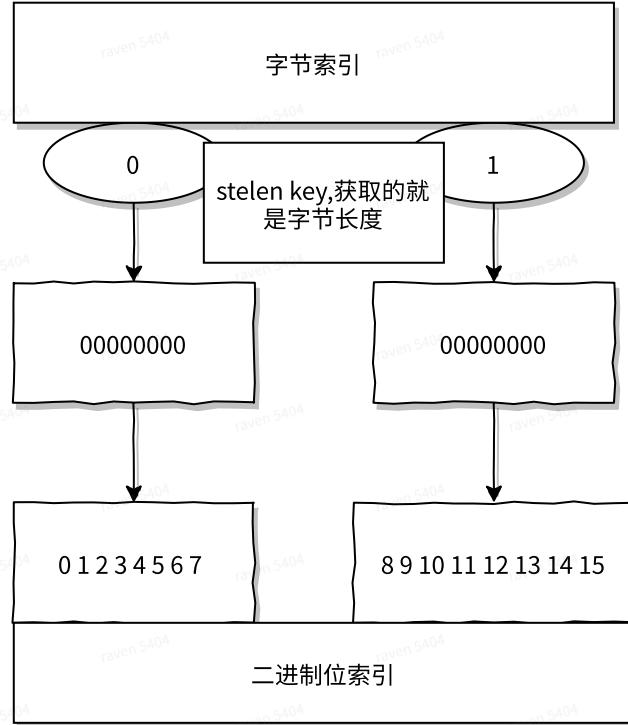
参考文档: <http://redis.cn/topics/data-types-intro.html#bitmaps>

bitmap不是实际的数据类型，而是在String类型上定义的一组面向位的操作。由于字符串是二进制安全Blob，并且最大长度为512 MB，因此它们适合设置多达 $2^{32}$ 个不同的位。

位操作分为两类：固定时间的单个位操作（如将一个位设置为1或0或获取其值），以及对位组的操作，例如计算给定位范围内设置的位的数量（例如，人口计数）。

**位图的最大优点之一是，它们在存储信息时通常可以节省大量空间**

**bitmap表示位图，数据结构，操作的的是二进制数。就只有0和1两个状态。使用场景可以是签到，统计用户信息，登录状态，有两个状态值的时候，可以使用bitmap来完成。**



**setbit key offset 「二进制位的偏移」 value**

## Bash

```

1 # 这里的k1是key, 第一个1是二进制位索引, 第三个1是它的值;
2 127.0.0.1:6379> setbit k1 1 1
3 (integer) 0
4 127.0.0.1:6379> get k1 # 默认是ascii编码方式, get方法是操作string类型的方法. 返回值
      是: 0100 0000 即64, 就是@符号
5 "@"
6 127.0.0.1:6379>

```

命令名称	描述
setbit	<p>设置值</p> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ <b>setbit key offset value</b></li> </ul> </li> <li>◦ 参数说明           <ul style="list-style-type: none"> <li>◦ <b>offset, 二进制位的索引值,从0开始</b></li> <li>◦ <b>value, 设置的值,0或者1;</b></li> </ul> </li> </ul>
getbit	<p>获取值</p> <ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ <b>getbit key offset</b></li> </ul> </li> </ul>

<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ bitcount key [start end]</li> </ul> </li> </ul>	<p><b>统计结果, 统计是1的个数;</b></p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ key</li> <li>◦ <b>start, 开始字节索引值</b></li> <li>◦ <b>end, 结束字节索引值</b></li> </ul>
<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ bitop operation destkey key [key ...]</li> </ul> </li> </ul>	<p>对二进制数据做位元操作, 与,或,非,异或操作</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ operation, 操作符</li> <li>◦ destkey, 存储结果的key的名称</li> <li>◦ key, 操作的key</li> <li>◦ [key ..] 可选参数, 操作的key</li> </ul>
<ul style="list-style-type: none"> <li>◦ 语法           <ul style="list-style-type: none"> <li>▪ bitpos key start end</li> </ul> </li> </ul>	<p>返回字符串里面第一个被设置为1或者0的bit位</p> <p>返回值是出现的二进制的位置,它是从0开始排列的,全量的值.</p> <p>参数说明:</p> <ul style="list-style-type: none"> <li>◦ key, 就是key</li> <li>◦ start, 从哪个字节开始找, 是字节的索引值</li> <li>◦ end, 找到哪个字节结束</li> </ul>

## Bash

- 1 BITOP 命令支持 AND 、 OR 、 NOT 、 XOR 这四种操作中的任意一种参数:
- 2 BITOP AND destkey srckey1 srckey2 srckey3 ... srckeyN , 对一个或多个 key 求逻辑并, 并将结果保存到 destkey 。
- 3 BITOP OR destkey srckey1 srckey2 srckey3 ... srckeyN, 对一个或多个 key 求逻辑或, 并将结果保存到 destkey 。
- 4 BITOP XOR destkey srckey1 srckey2 srckey3 ... srckeyN, 对一个或多个 key 求逻辑异或, 并将结果保存到 destkey 。
- 5 BITOP NOT destkey srckey, 对给定 key 求逻辑非, 并将结果保存到 destkey 。 「只能接受一个key进行操作.两个或者多个会报错;」
- 6 除了 NOT 操作之外, 其他操作都可以接受一个或多个 key 作为输入。
- 7 执行结果将始终保持到destkey里面

## Bash

```
1 127.0.0.1:6379> setbit sign 0 1 # 周一打卡了(integer) 0
2 127.0.0.1:6379> setbit sign 1 0 # 周二未打卡(integer) 0
3 127.0.0.1:6379> setbit sign 2 0 # 周三未打卡(integer) 0
4 127.0.0.1:6379> setbit sign 3 1
5 (integer) 0
6 127.0.0.1:6379> setbit sign 4 1
7 (integer) 0
8 127.0.0.1:6379> setbit sign 5 1
9 (integer) 0
10 127.0.0.1:6379> setbit sign 6 0
11 (integer) 0
12 127.0.0.1:6379>
13 # 查看某一天是否打卡了
14 127.0.0.1:6379> GETBIT sign 3
15 (integer) 1
16 127.0.0.1:6379> GETBIT sign 6
17 (integer) 0
18 127.0.0.1:6379>
19 # 统计结果
20 127.0.0.1:6379> BITCOUNT sign
21 (integer) 4
22 127.0.0.1:6379>
```

- 统计某人某一时间段之内是否登录过账号
  - 时间段是可选的.
  - 利用 **bitmap**
- 掌握住这些个数据类型及常用的命令
- 配置文件常用配置项
  - port
  - deamin
  - bind
  - logfile
  - pidfile
  - 指定配置文件启动.如果不指定,则会找redis.conf配置文件;
  - 推荐是在源码目录下启动redis

## 第四章: 事务

### 1. 几点重要的说明

- 单条命令执行保证是原子性。但是事务不保证其原子性；「不回滚」
- redis并没有事务隔离级别的概念
- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

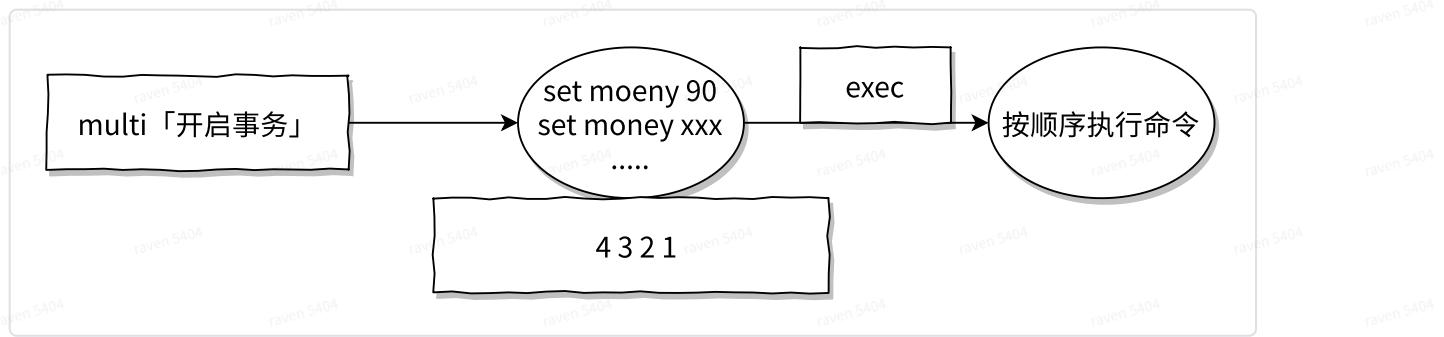
- 指的命令

## 2. 事务的本质

- 一组命令的集合，一个事务当中所有命令都会被序列化。在事务执行中，会按照顺序执行
- 所有命令在事务当中，并没有直接被执行，只有发起执行命令时才会执行：exec

## 3. 相关命令

命令	描述
multi	<p>开启事务,返回值是ok, 表示开启了事务;</p> <p><b>MULTI 命令用于开启一个事务，它总是返回 OK</b>。MULTI 执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当 EXEC 命令被调用时，所有队列中的命令才会被执行。</p> <p>另一方面，通过调用 DISCARD，客户端可以清空事务队列，并放弃执行事务。</p>
exec	<p><b>负责触发并执行事务中的所有命令</b></p> <ul style="list-style-type: none"> <li>如果客户端在使用 MULTI 开启了一个事务之后，却因为断线而没有成功执行 EXEC，那么事务中的所有命令都不会被执行。</li> <li>另一方面，如果客户端成功在开启事务之后执行 EXEC，那么事务中的所有命令都会被执行。<b>「不管结果正确与否，它只是执行队列当中的命令」</b></li> <li>EXEC 命令的回复是一个数组，数组中的每个元素都是执行事务中的命令所产生的回复。其中，回复元素的先后顺序和命令发送的先后顺序一致。</li> <li>当客户端处于事务状态时，所有传入的命令都会返回一个内容为 QUEUED 的状态回复 (status reply)，这些被入队的命令将在 EXEC 命令被调用时执行。</li> </ul>
discard	<p><b>取消事务, 清空事务队列中的命令；</b></p> <p>当执行 DISCARD 命令时，<b>事务会被放弃，事务队列会被清空</b>，并且客户端会从事务状态中退出：</p>



## 4. 事务的相关操作

### 操作步骤

Bash

```

1 # 1. 开启事务: multi
2 # 2. 命令入队「就是执行各种命令」
3 # 3. 执行命令: exec

```

Bash

```

1 # 开启事务: multi
2 codertalk.cn:6379> multi
3 OK
4 # 各种命令操作
5 codertalk.cn:6379> set s1 tom
6 QUEUED
7 codertalk.cn:6379> set s2 jerry
8 QUEUED
9 codertalk.cn:6379> set k3 kate
10 QUEUED
11 codertalk.cn:6379> get k2 # 没有此键值,应该报错,但是由于开启了事务, 所以直到真正执行
   的时候, 会默认一个值给它;
12 QUEUED
13 # 执行各种命令, 相当于提交事务了
14 codertalk.cn:6379> exec
15 1) OK
16 2) OK
17 3) OK
18 4) (nil)
19 codertalk.cn:6379>
20 # 开启事务, 测试: 事务在进入命令队列之前报错;
21 codertalk.cn:6379> multi
22 OK
23 codertalk.cn:6379> set user tom
24 QUEUED
25 codertalk.cn:6379> set age 18
26 QUEUED
27 codertalk.cn:6379> set gender 男

```

```
28 QUEUED
29 # 进入命令队列之前就报错了。
30 codertalk.cn:6379> getset age
31 (error) ERR wrong number of arguments for 'getset' command
32 codertalk.cn:6379> exec
33 # 进入命令队列之前就报错是了,那么该事务会被取消掉;
34 (error) EXECABORT Transaction discarded because of previous errors.
35 codertalk.cn:6379> exec
36 (error) ERR EXEC without MULTI
37 # 值没有被正确的取出来
38 codertalk.cn:6379> get user
39 (nil)
40 codertalk.cn:6379>
41
42 # redis事务不保证其原子性
43 codertalk.cn:6379> set k1 tom
44 OK
45 codertalk.cn:6379> get k1
46 "tom"
47 codertalk.cn:6379> multi # 开启事务
48 OK
49 codertalk.cn:6379> set k2 jerry
50 QUEUED
51 codertalk.cn:6379> set k3 10
52 QUEUED
53 codertalk.cn:6379> incr k1 # 给k1自增操作, 语法上是正确的. 但是结果是错误的;
54 QUEUED
55 codertalk.cn:6379> incr k3
56 QUEUED
57 # 执行命令
58 codertalk.cn:6379> exec
59 1) OK
60 2) OK
61 # incr k1, 任务执行失败, 但是不会影响到其它的可能;
62 3) (error) ERR value is not an integer or out of range
63 4) (integer) 11
64 codertalk.cn:6379>
65
```

- 为什么redis事务不回滚?「官网答案」
    - Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中。
    - 因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速。
  - 总结

- 如果在命令进入队列的时候就出现了异常，那么在事务被执行的时候「exec」的时候，那么任务在被执行的时候会被取消掉；
- 如果在命令进入队列之前，没有出现异常，即「语法是正确的，但是结果可能是错误的」，此种情况，命令队列执行的时候，执行出现问题的，直接抛出执行过程上的异常，不会影响其它的任务队列操作；-> 事务不回滚。
- 至于那些在 EXEC 命令执行之后所产生的错误，并没有对它们进行特别处理：即使事务中有某个/某些命令在执行时产生了错误，事务中的其他命令仍然会继续执行。

## 第五章: 监控「watch」

**WATCH** 命令可以为 Redis 事务提供 check-and-set (CAS) 行为。

被 **WATCH** 的键会被监视，并会发觉这些键是否被改动过了。如果有至少一个被监视的键在 **EXEC** 执行之前被修改了，那么整个事务都会被取消，**EXEC** 返回 **nil-reply** 来表示事务已经失败。

### 1. 两个重要的锁概念

- 乐观锁「重要」
  - 认为什么时候都不会出问题，所以不会加锁。
  - 更新数据的时候判断一下，在些期间是否有人修改过这些数据；
    - 获取version，解决ABA问题；
    - 更新的时候进行比较操作；
- 悲观锁
  - 无论什么都会加锁，这样会影响到性能。

### 2. 问题的引出

假设想为某一个key原子性的加上1。「假设不使用incr」

#### Bash

```

1 val = GET mykey
2 val = val + 1
3 SET mykey $val

```

上面的这个实现在只有一个客户端的时候可以执行得很好。但是，当多个客户端同时对同一个键进行这样的操作时，就会产生竞争条件。举个例子，如果客户端 A 和 B 都读取了键原来的值，比如 10，那么两个客户端都会将键的值设为 11，但正确的结果应该是 12 才对。

- watch上场了
  - 修改如下

## Bash

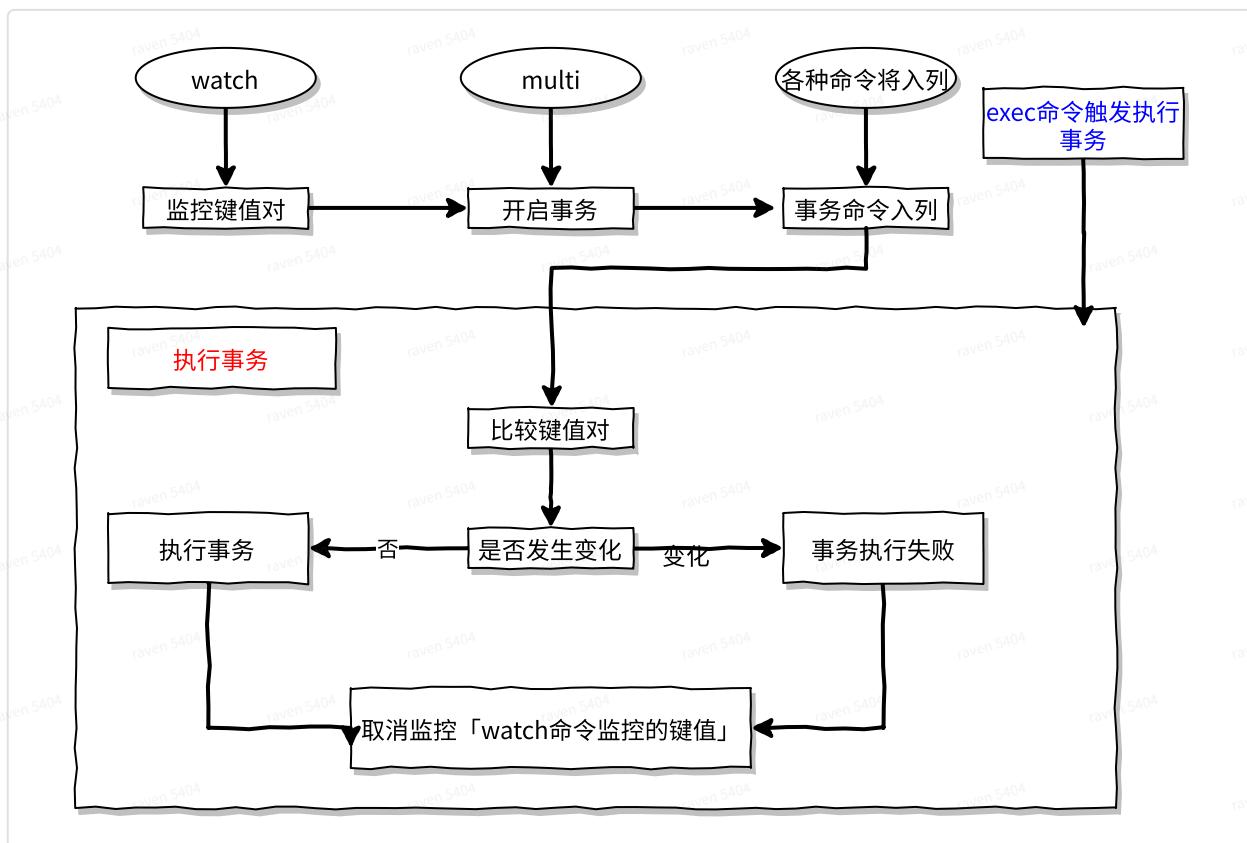
```
1 # 伪代码
2 WATCH mykey
3 val = GET mykey
4 val = val + 1
5 MULTI
6 SET mykey $val
7 EXEC
```

使用上面的代码，**如果在 WATCH 执行之后，EXEC 执行之前，有其他客户端修改了 mykey 的值，那么当前客户端的事务就会失败。** 程序需要做的，就是不断重试这个操作，直到没有发生碰撞为止。

这种形式的锁被称作**乐观锁**，它是一种非常强大的锁机制。并且因为大多数情况下，不同的客户端会访问不同的键，碰撞的情况一般都很少，所以通常并不需要进行重试。

### WATCH 使得 EXEC 命令需要有条件地执行：

事务只能在所有被监视键都没有被修改的前提下执行，如果这个前提不能满足的话，事务就不会被执行。



### 要求：

- 乐观锁实现原理
- ABA如何解决
- 执行事务的必须步骤

- watch key

- multi

- 把所有要执行命令入队

- exec

- redis不支持回滚,不保证原子性.

- 主动清空事务的队列, 入队的时候命令出现了语法错误也会清空命令的队列

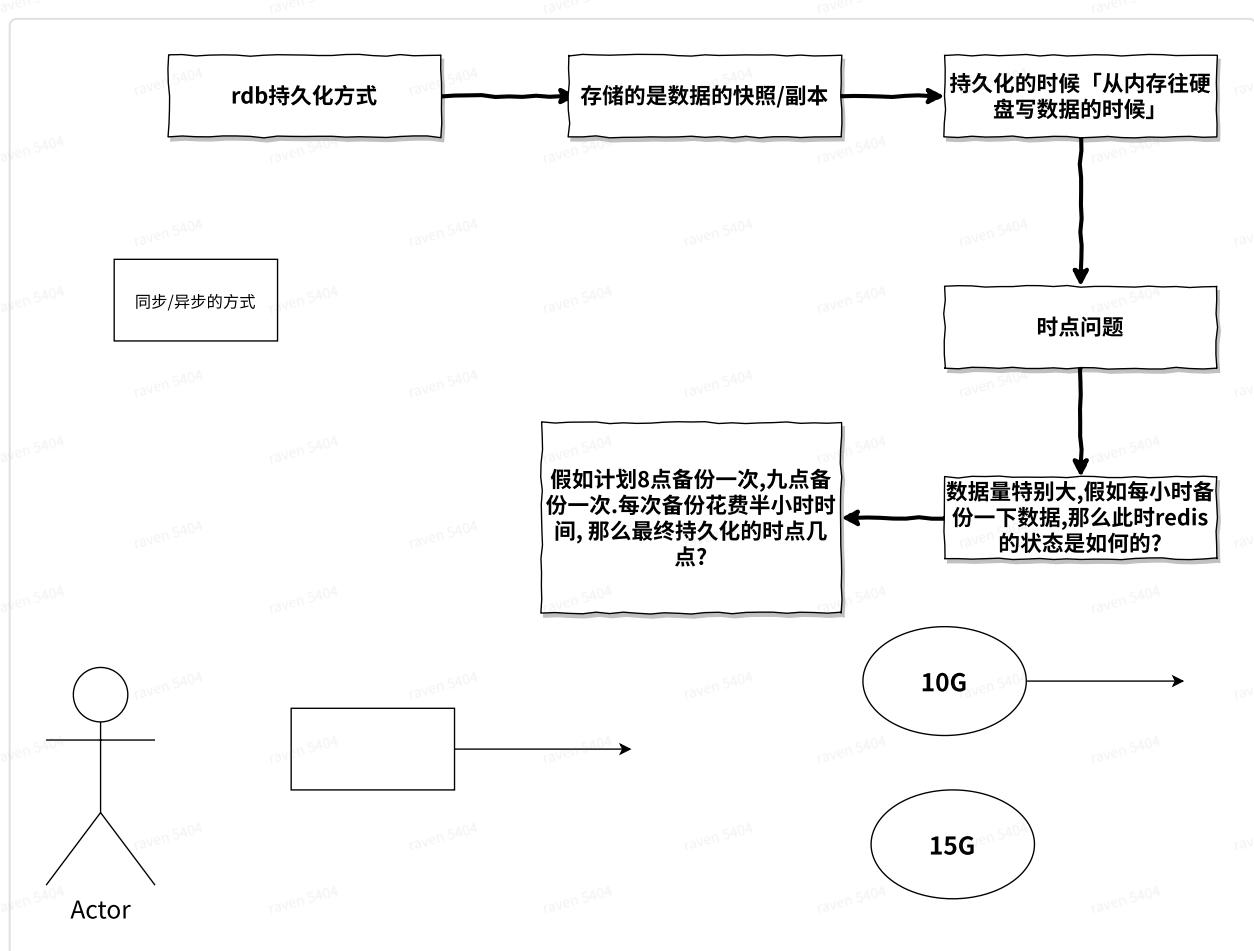
- 命令语法正确, 执行结果是错误的, 不影响其它命令的执行;

- 乐观锁和悲观锁

## 第六章: 持久化操作 「重点内容, 必须掌握」

redis是内存型数据库, 如果不将内存中的数据库状态保存到硬盘上, 那么一旦服务器进程退出. 服务器中的数据库状态也会消失. 所以redis提供了持久化功能;

### • 根据昨天学的内容思考如下的问题:



## 0. 管道符的作用

- 管道符的作用, 把前边的输出, 作为后边的输入;

- | 会触发创建子进程操作 ;

## Bash

```
1 ls lh | more
2
3 num=0
4 echo $num # 输出num的值;
5 ((num++)) # num自增1操作;
6 echo $num # 输出num的值;
7
8 # | 符,可以创建子进程
9 ((num++)) | echo tom
10 # 输出结果是tom
11 # 此时开启了三个进程, 主界面一个进程, 交互式进程.
12 # 管道符左右各开启一个子进程进行操作
13
```

- echo \$\$ ,输出当前进程的pid

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $$ 
2 29953 # 当前的进程id
3 [root@iZ2ze8biiph4vi0x5z06heZ local]#
```

- echo \$BASHPID, 取出当前进程的id号,作用同上

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID
2 29953 # 效果是一样的, 都是获取当前进程的id号;
3 [root@iZ2ze8biiph4vi0x5z06heZ local]#
4
5 # 已经知道了, | 符号可以创建子进程. 那么如下输出的进程是父进程的,还是子进程的?
6 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $$ | more
```

- \$\$ 的优先级高于管道符号

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $$ | more
2 # echo $$, 可以获取到当前进程的pid
3 30076 # 输出的是 | 符号创建的子进程的pid
4 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID | more
5 30076 # 输出的是 | 符号创建的子进程的pid
6 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID | more
7 30082 # 输出的是 | 符号创建的子进程的pid
8 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID | more
9 30084 # 输出的是 | 符号创建的子进程的pid
10 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID | more
11 30086 # 输出的是 | 符号创建的子进程的pid
12 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID | more
13 30088 # 输出的是 | 符号创建的子进程的pid
14 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $BASHPID | more
15 30090 # 输出的是 | 符号创建的子进程的pid
16 [root@iZ2ze8biiph4vi0x5z06heZ local]#
```

- pstree, 查看进程树
- /bin/bash 开启一个新的交互进程, pid就是一个新的pid喽;
- |, 创建一个子进程.
- echo \$\$, 优先级比 | 高一些;/echo \$BASHPID

## 1. 知识储备

- 进程的概念

程序在计算机中运行的实体。程序被加载到计算机内存中运行，即进行中的程序。不同进程间有用独立的内存；

### wiki.kfd.me

<https://wiki.kfd.me/zh-hans/%E8%A1%8C%E7%A8%8B>

- 父进程和子进程

### wiki.kfd.me

<https://wiki.kfd.me/wiki/%E5%AD%90%E8%BF%9B%E7%A8%8B>



- 在Linux下，父进程创建子进程需要用到fork()函数
- 一个进程通过fork()函数，创建一个和自己一样的进程。创建出来的进程子进程，而创建子进程的进程就是父进程
- 子进程是父进程的一个副本，是它的复制，两者不共享地址空间。
- fork()函数最主要的特点是，调用一次，返回两次。
  - 当父进程fork()创建子进程失败时，fork()返回-1。
  - 当父进程fork()创建子进程成功时，此时，父进程会返回子进程的pid，而子进程返回的是0。所以可以通过返回的pid来控制父子进程后续所要执行的操作。

- 父进程中的数据对于子进程来说，是不是可见的？

- 一般思想，父进程和子进程的数据是相互隔离的

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $$  
2 29953 # 当前父进程的pid  
3 [root@iZ2ze8biiph4vi0x5z06heZ local]# num=1 # 给个变量赋值  
4 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $num # 输出一下值  
5 1  
6 [root@iZ2ze8biiph4vi0x5z06heZ local]# /bin/bash # 开启一个子进程  
7 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $$ # 查看子进程的pid  
8 30149  
9 [root@iZ2ze8biiph4vi0x5z06heZ local]# pstree # 查看进程树  
10 systemd—2*[agetty]  
11     |—assist_daemon—7*[{assist_daemon}]  
12     |—atd  
13     |—auditd—{auditd}  
14     |—chronyrd  
15     |—containerd—8*[{containerd}]  
16     |—dbus-daemon  
17     |—dhclient  
18     |—dockerd—15*[{dockerd}]  
19     |—master—pickup  
20     |     |—qmgr  
21     |—polkitd—6*[{polkitd}]  
22     |—redis-server—3*[{redis-server}]  
23     |—rsyslogd—2*[{rsyslogd}]  
24     |—sshd—sshd—bash—bash—pstree  
25     |—systemd-journal  
26     |—systemd-logind  
27     |—systemd-udevd  
28     |—tuned—4*[{tuned}]  
29 [root@iZ2ze8biiph4vi0x5z06heZ local]# echo $num # 读取在父进程中定义的变量  
30  
31 # 发现，在子进程中，是无法获取到父进程中定义的变量的；这是一种常规思想。也就是说，父子进程  
之间数据是相互隔离的；两者并不共享内存空间；
```

- 进阶思想，父进程可以让子进程看见它的数据

- **export 数值变量**

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $$  
2 30220  
3 [root@iZ2ze8biiph4vi0x5z06heZ ~]# num=1  
4 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $num  
5 1  
6 [root@iZ2ze8biiph4vi0x5z06heZ ~]# export num  
7 [root@iZ2ze8biiph4vi0x5z06heZ ~]# pstree
```

```
raven 5404
8  systemd—2*[agetty] # 查看进程树
9      |—assist_daemon—7*[{assist_daemon}]
10     |—atd
11     |—auditd—{auditd}
12     |—chrony
13     |—containerd—8*[{containerd}]
14     |—dbus-daemon
15     |—dhclient
16     |—dockerd—15*[{dockerd}]
17     |—master—pickup
18         |—qmgr
19     |—polkitd—6*[{polkitd}]
20     |—redis-server—3*[{redis-server}]
21     |—rsyslogd—2*[{rsyslogd}]
22     |—sshd—sshd—bash—pstree # 进程树程序在当前进程中, pid = 30220
23     |—systemd-journal
24     |—systemd-logind
25     |—systemd-udevd
26         |—tuned—4*[{tuned}]
27 [root@iZ2ze8biiph4vi0x5z06heZ ~]# /bin/bash # 开启一个子进程
28 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $$ # 输出一下子进程pid
29 30243
30 [root@iZ2ze8biiph4vi0x5z06heZ ~]# pstree # 查看进程树
31 systemd—2*[agetty]
32     |—assist_daemon—7*[{assist_daemon}]
33     |—atd
34     |—auditd—{auditd}
35     |—chrony
36     |—containerd—8*[{containerd}]
37     |—dbus-daemon
38     |—dhclient
39     |—dockerd—15*[{dockerd}]
40     |—master—pickup
41         |—qmgr
42     |—polkitd—6*[{polkitd}]
43     |—redis-server—3*[{redis-server}]
44     |—rsyslogd—2*[{rsyslogd}]
45     |—sshd—sshd—bash—bash—pstree # 当前两个bash, 说明是在子进程中工作;
46     |—systemd-journal
47     |—systemd-logind
48     |—systemd-udevd
49         |—tuned—4*[{tuned}]
50 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $num # 输出在父进程中定义的变量值.
```

- 特别强调:

## ▪ 在子进程中修改父进程中定义的数据,不会成功!

### • 验证过程

- a. 新建一个Bash脚本. 内容如下: 「此处不必解脚本代码啥意思, 主要体会意图和思想即可;」

- i. t.sh, 复制如下内容到文件当中;

```
Bash

1 #!/bin/bash
2 echo "在子进程中尝试读取父进程的数据~~~~~"
3 echo "父进程中的原始数据是:"
4 echo $num
5
6 echo "在子进程中,尝试修改父进程中定义的数据~~~~~"
7 $num=678
8 echo "在子进程中,修改完的数据是: "
9 echo $num
10
11 sleep 30
12 echo "最终值是: "
13 echo $num
```

- 给新建的文件, 可执行权限;
- chmod +x t1.sh,
  - 以守护进程的方式执行该脚本: ./t1.sh &
  - 守护进程就是一个子进程. 不同于父进程;

```
Bash

1 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $$
2 30435
3 [root@iZ2ze8biiph4vi0x5z06heZ games]# num=1
4 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num
5 1
6 [root@iZ2ze8biiph4vi0x5z06heZ games]# export num # 导出父进程中的变量,给子进程使用;并在子进程中修改该变量的值;
7 [root@iZ2ze8biiph4vi0x5z06heZ games]# pstree
8 systemd---2*[agetty]
9     |---assist_daemon---7*[{assist_daemon}]
10    |---atd
11    |---auditd---{auditd}
12    |---chrony
13    |---containerd---8*[{containerd}]
14    |---dbus-daemon
15    |---dhclient
16    |---dockerd---15*[{dockerd}]
```

```

16      └── ulockerd └── 15x1[ulockerd]
17          ├── master └── pickup
18              └── qmgr
19          ├── polkitd └── 6*[{polkitd}]
20          ├── redis-server └── 3*[{redis-server}]
21          ├── rsyslogd └── 2*[{rsyslogd}]
22          ├── sshd └── sshd └── bash └── pstree
23          ├── systemd-journal
24          ├── systemd-logind
25          ├── systemd-udevd
26          └── tuned └── 4*[{tuned}]
27 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num
28 1
29 [root@iZ2ze8biiph4vi0x5z06heZ games]# ./t1.sh & # 后台启动shell脚本;执行脚本里的业务逻辑;
30 [1] 30456
31 [root@iZ2ze8biiph4vi0x5z06heZ games]# 在子进程中尝试读取父进程的数据~~~~~
32 父进程中的原始数据是:
33 1 # 父进程中的变量的值;
34 在子进程中,尝试修改父进程中定义的数据~~~~
35 在子进程中,修改完的数据是:
36 678 # 子进程中,修改了;
37
38 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num
39 1 # 在子进程中,修改了父进程的值,回到父进程中,读取值,还是原来的值;
40 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num
41 1
42 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num
43 1
44 [root@iZ2ze8biiph4vi0x5z06heZ games]# 最终值是:
45 678
46
47 [1]+ Done                  ./t1.sh # 脚本执行完成;
48 [root@iZ2ze8biiph4vi0x5z06heZ games]#

```

### ▪ 父进程的修改也不会破坏子进程

```

[root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $$
30517 父进程pid
[root@iZ2ze8biiph4vi0x5z06heZ ~]# num=1
[root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $num
1
[root@iZ2ze8biiph4vi0x5z06heZ ~]# export num → 导出变量给子进程使用
[root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $num
1
[root@iZ2ze8biiph4vi0x5z06heZ ~]# cd /usr/local/games/
[root@iZ2ze8biiph4vi0x5z06heZ games]# ./t1.sh &
[1] 30554 子进程pid
当前子进程的pid =
30554 子进程pid
在子进程中尝试读取父进程的数据~~~~~
父进程中的原始数据是:
1
在子进程中,尝试修改父进程中定义的数据~~~~
在子进程中,修改完的数据是:
678
[root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num → 子进程修改了数据,父进程不影响,还是原来的值:
1

```

```
[root@iZ2ze8biiph4vi0x5z06heZ games]# num=876
[root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num
876
[root@iZ2ze8biiph4vi0x5z06heZ games]# 最终值是:
678
子进程中的最终值,还是在子进程中修改的值;
[1]+  Done                  ./t1.sh
[root@iZ2ze8biiph4vi0x5z06heZ games]#
```

→ 父进程中修改数据，并且在父进程中已经修改了数据，而且输出了；

→子进程中的最终值,还是在子进程中修改的值;

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $$  
2 30517  
3 [root@iZ2ze8biiph4vi0x5z06heZ ~]# num=1  
4 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $num  
5 1  
6 [root@iZ2ze8biiph4vi0x5z06heZ ~]# export num  
7 [root@iZ2ze8biiph4vi0x5z06heZ ~]# echo $num  
8 1  
9 [root@iZ2ze8biiph4vi0x5z06heZ ~]# cd /usr/local/games/  
10 [root@iZ2ze8biiph4vi0x5z06heZ games]# ./t1.sh &  
11 [1] 30554  
12 [root@iZ2ze8biiph4vi0x5z06heZ games]# 当前子进程的pid =  
13 30554  
14 在子进程中尝试读取父进程的数据~~~~~  
15 父进程中的原始数据是：  
16 1  
17 在子进程中，尝试修改父进程中定义的数据~~~~~  
18 在子进程中，修改完的数据是：  
19 678  
20  
21 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num  
22 1  
23 [root@iZ2ze8biiph4vi0x5z06heZ games]# num=876  
24 [root@iZ2ze8biiph4vi0x5z06heZ games]# echo $num  
25 876  
26 [root@iZ2ze8biiph4vi0x5z06heZ games]# 最终值是：  
27 678  
28  
29 [1]+ Done ./t1.sh  
30 [root@iZ2ze8biiph4vi0x5z06heZ games]#
```

- 结论「作为一个常识,必须死死的记住;」

- 这个结论必须记住:

## Bash

```
1 # 1. 进程和进程之间数据是隔离的。  
2 #      1. 数据可以可见，但是  
3 #          在子进程中修改数据，父进程不受影响  
4 #          在父进程修改数据，子进程不受影响
```

• 思考问题?

## Bash

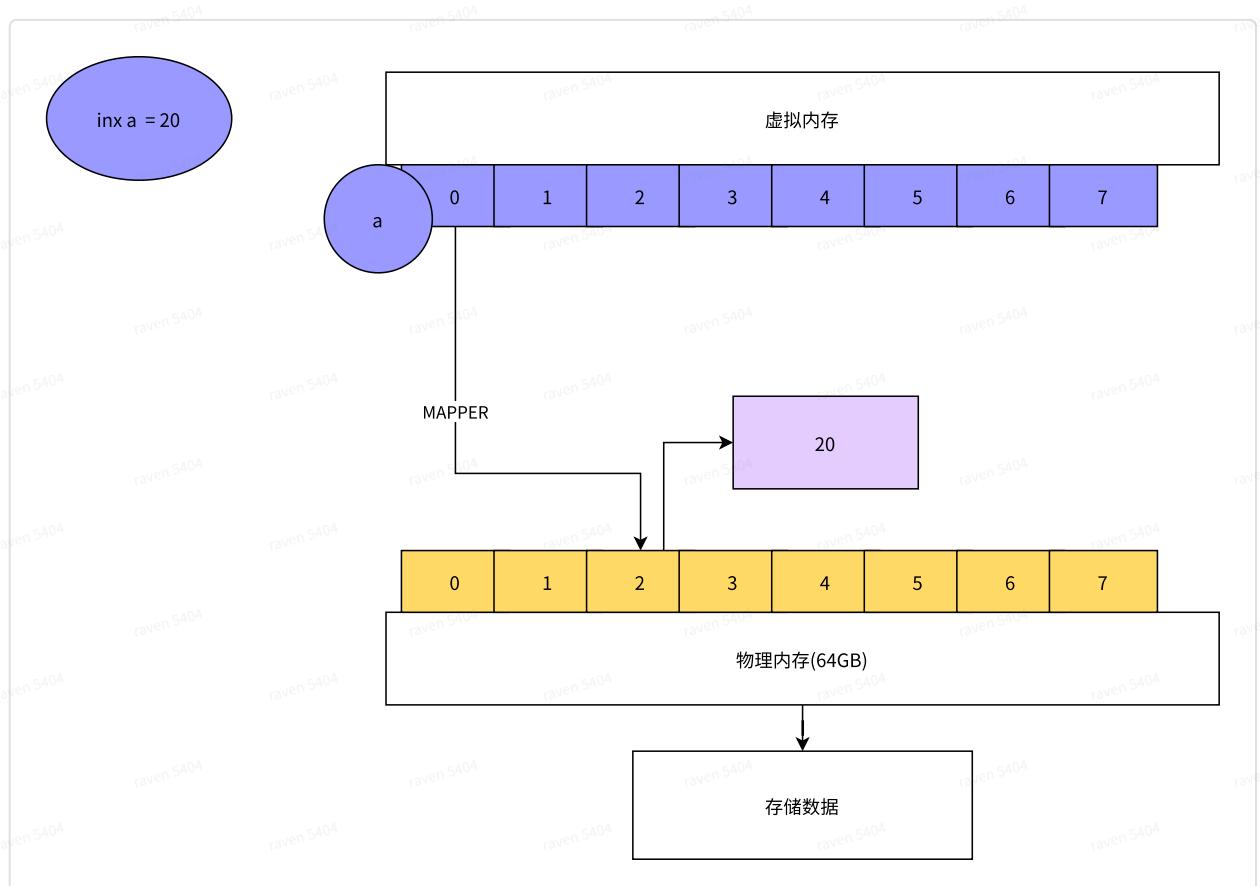
```
1 # 1. 如果redis进行快照/副本操作的时候，咱们儿猜测一下，是如何做的？
2     # 1. 如果采用同步的方式，那以redis会停止对外的服务。这样明显是不可取的。造成了服务不可用；
3     # 2. 异步的方式，采用创建子进程的方式来实现；
4         # 父进程，提供对外的服务。
5         # 子进程，从内存往硬盘写数据；--> 采用的方案；
6
7 # 2. 采用子进程的方式实现，那么还是有问题？
8     # 1. 创建子进程的速度如何？
9     # 2. 内存空间，够不够用？
10
11    # 继续想，如果redis内存占用了10G，那么在子进程从内存往硬盘写数据的时候，拷贝的文件大小是否是10G，那么这个时间是占用多长，时点问题怎么解决？
```

## ● 补充知识

- 物理内存/地址

### 真实物理机的内存和地址

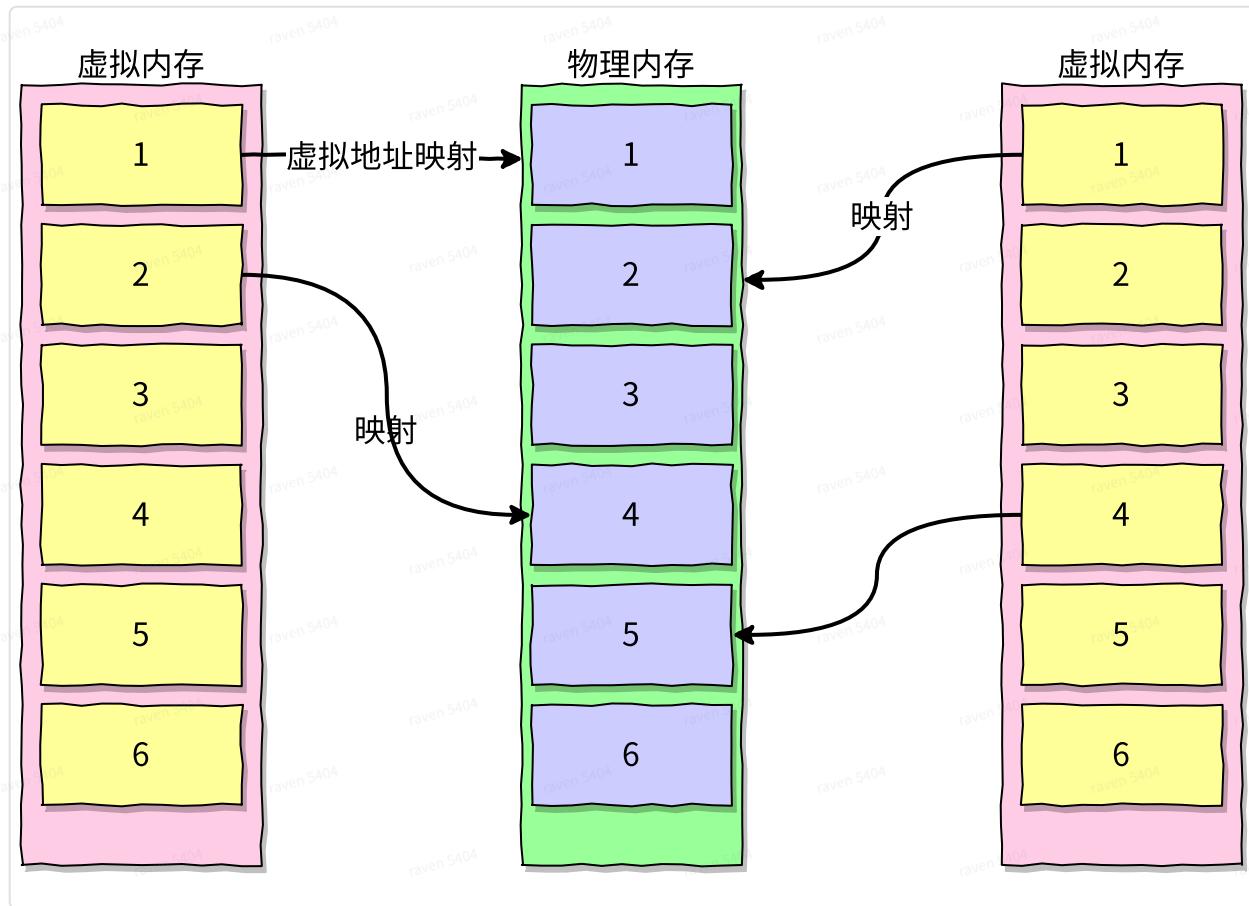
- 虚拟内存/地址



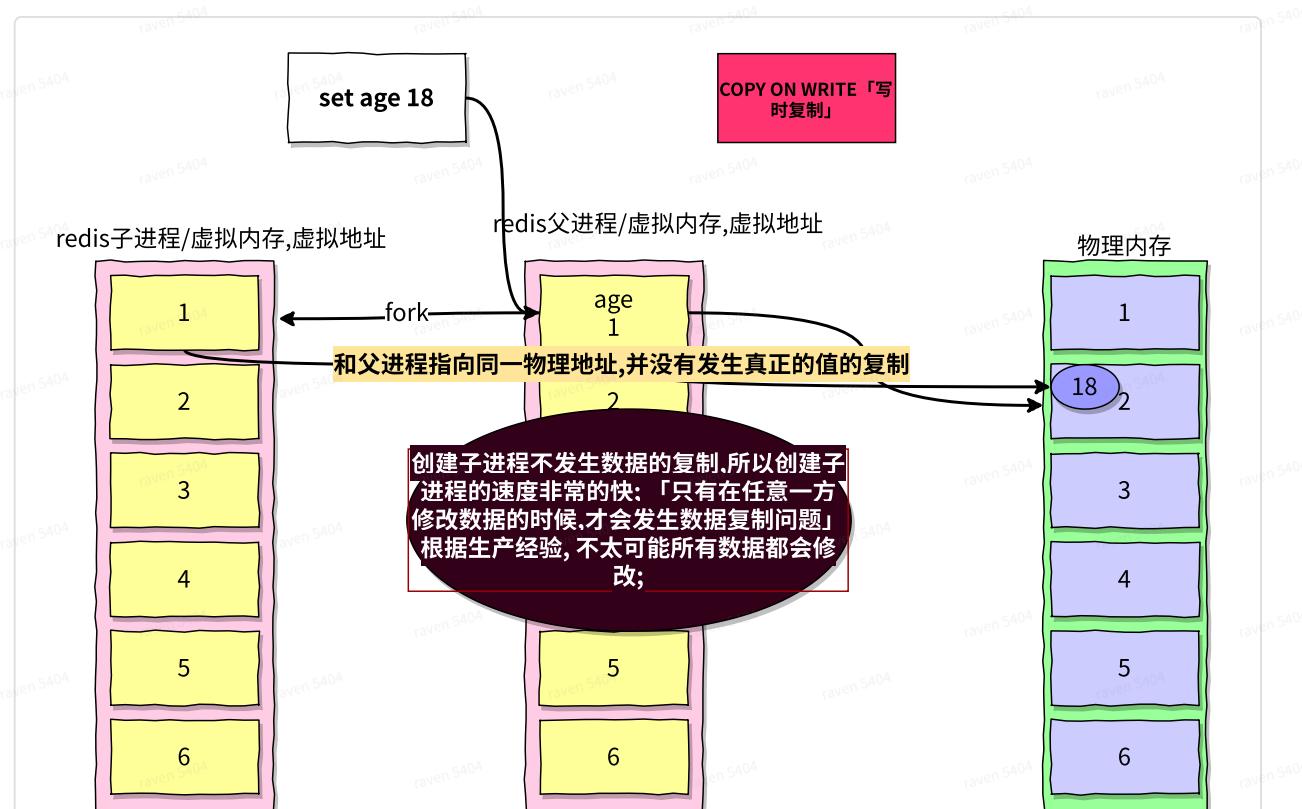


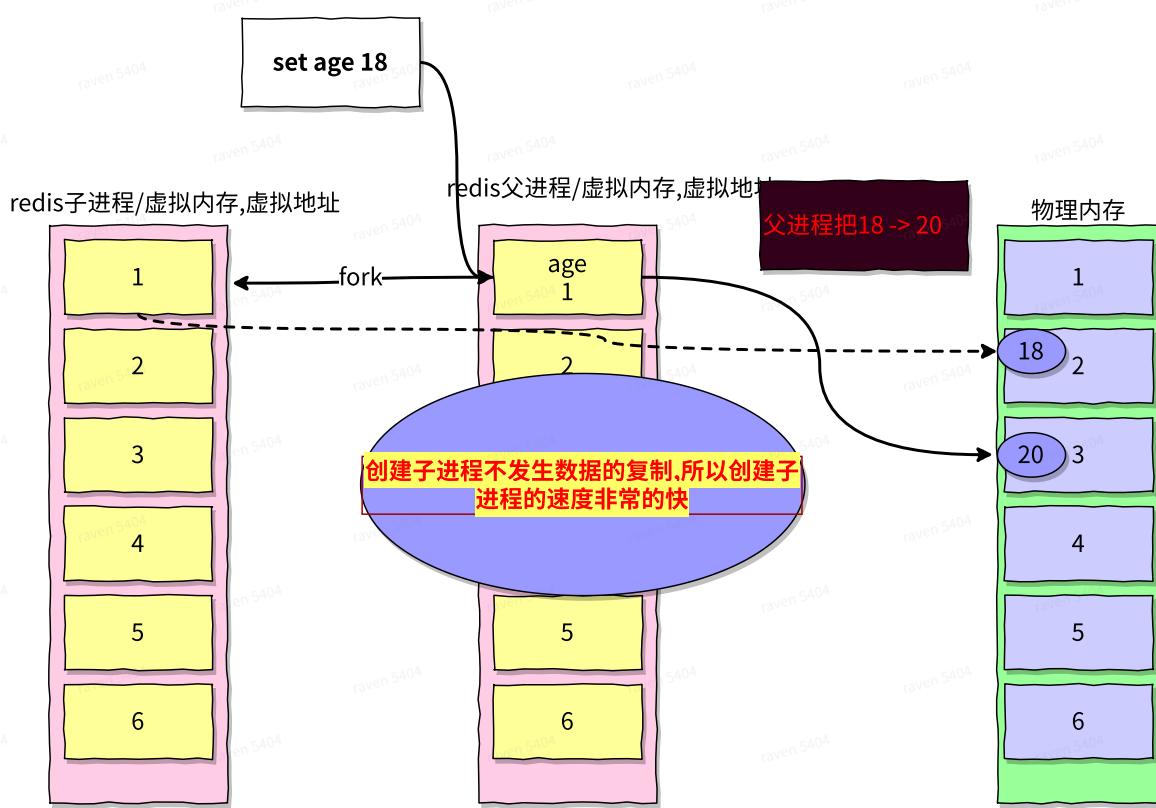
- 避免用户直接访问物理内存地址，防止一些破坏性操作，保护操作系统。
- 每个进程都被分配了4GB的虚拟内存，用户程序可使用比实际物理内存更大的地址空间

### ○ 虚拟地址与物理地址的映射



### ● redis是如何做的？





总结: Fork和copy on write机制 内核提供的机制;由linux内核提供; 「和Redis没有啥关系」  
什么时候发生数据复制?

因为在子进程中或者父进程中对数据进行修改,对方是不可见的.当任意一方发生了: copy on write  
「写时复制」机制. 就会触发在内存的复制行为;根据经验,不太可能把所有数据都修改一次;

当父进程, 修改数据的时候, 会先把值存储在物理内存当中, 然后父进程把虚拟地址映射到物理地址上;子进程再读的时候,就读取到新的值了;

当父进程修改数据的时候,那么父进程映射的物理地址发生改以后,父进程指向了新的物理地址;

rdb速度快

### ◦ fork系统调用

系统调用: man 2 fork

## Bash

```
1 Under Linux, fork() is implemented using copy-on-write pages, so the only
2 penalty that it incurs is the time and memory required to duplicate the par-
3 ent's page tables, and to create a unique task structure for the child.
4
5 Since version 2.3.3, rather than invoking the kernel's fork() system
6 call, the glibc fork() wrapper that is provided as part of the NPTL
7 threading
8 implementation invokes clone(2) with flags that provide the same
9 effect as the traditional system call. (A call to fork() is equivalent to a
10 call to
11 clone(2) specifying flags as just SIGCHLD.) The glibc wrapper invokes
12 any fork handlers that have been established using pthread_atfork(3).
```

### ◦ copy on write

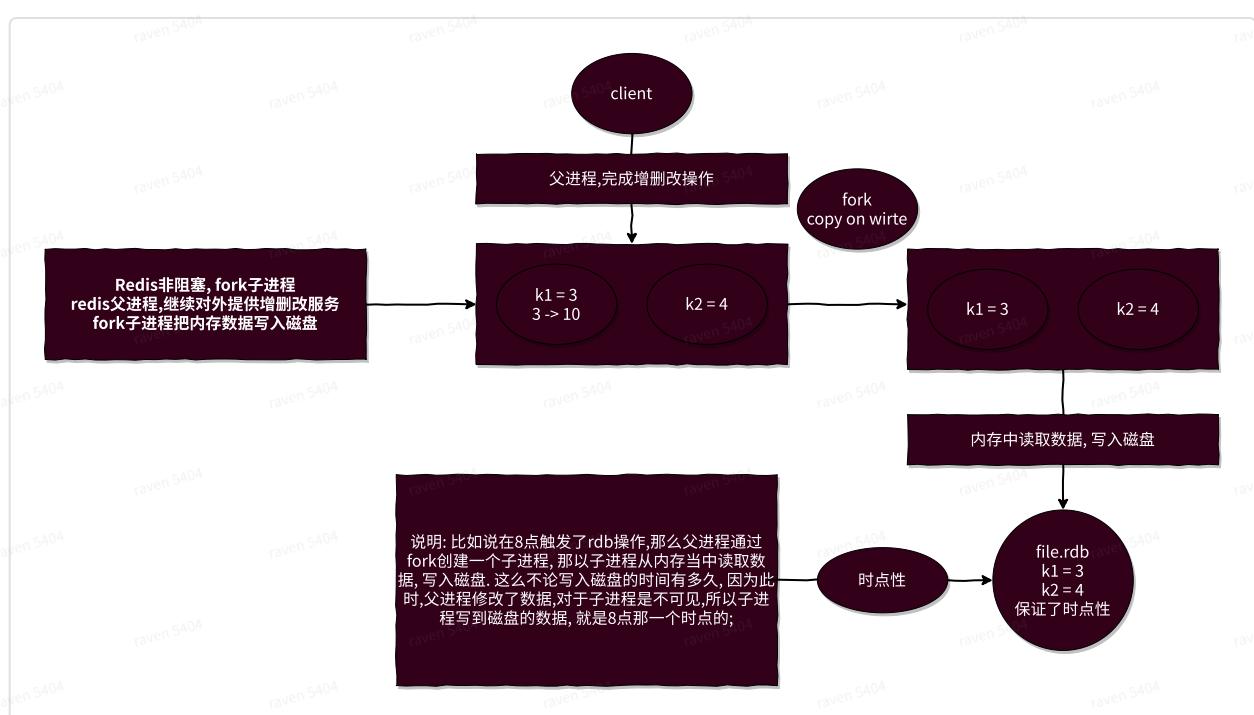
- 底层实现的和保证的;

[en.wikipedia.org](https://en.wikipedia.org)

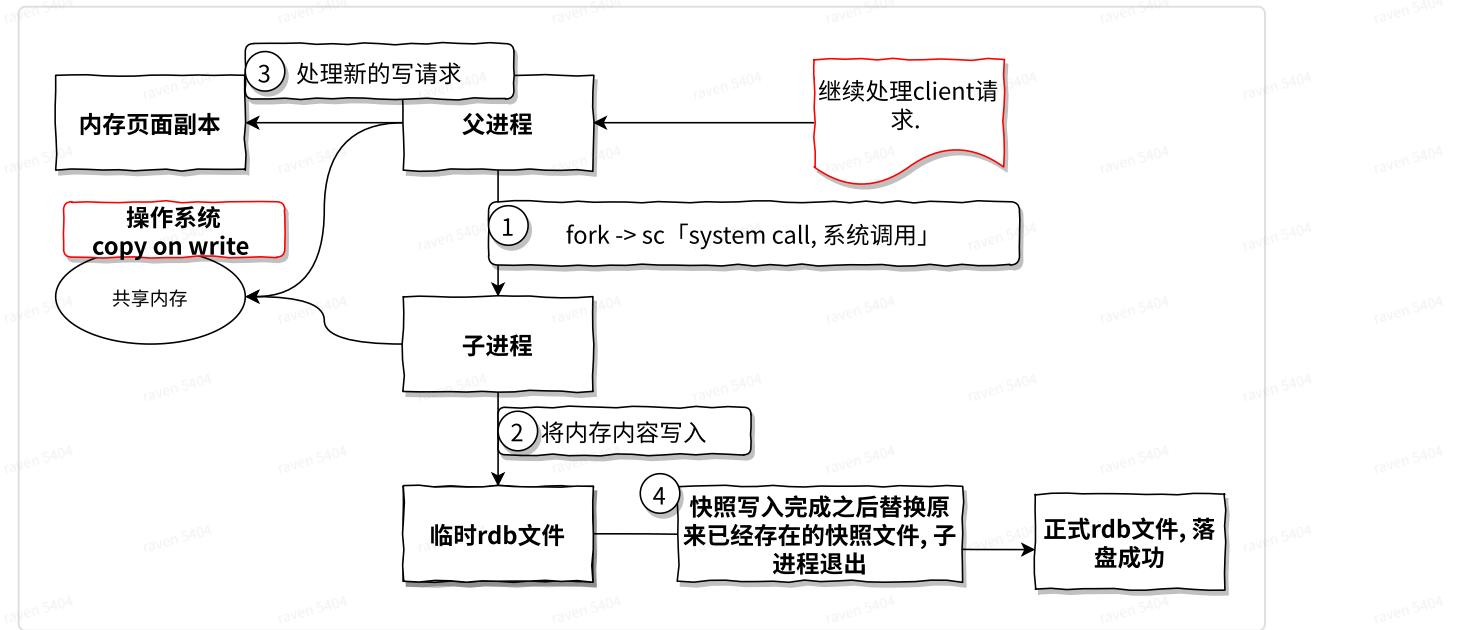
<https://en.wikipedia.org/wiki/Copy-on-write>

## Bash

```
1 copy on write # 写时复制
2 # 创建子进程并发生复制
```

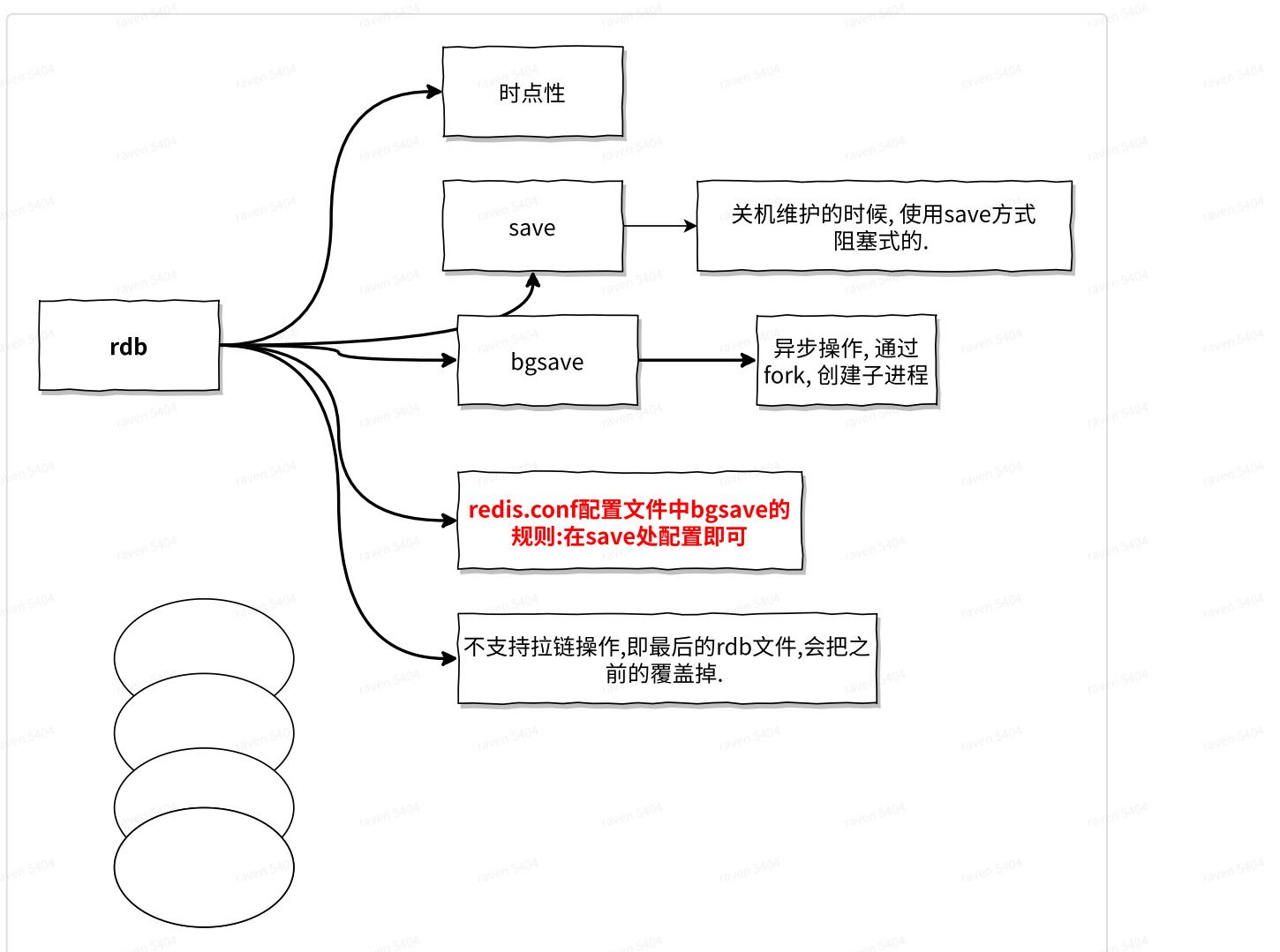


- **rdb实现原理图**



## 1. rdb方式「redis database」 「快照/副本,具有时点性特征」

redis默认开启;



### • 缺点

- 丢失数据相对多一些, 时点与时点之间数据容易丢失

- 优点

- 恢复速度的比较快一些, 类似于java中的序列化;

- 相关配置文件

### XML

```
1  285 ##### SNAPSHOTTING
#####
2  286 #
3  287 # Save the DB on disk:
4  288 #
5  289 # save <seconds> <changes>
6  290 #
7  291 # Will save the DB if both the given number of seconds and the given
8  292 # number of write operations against the DB occurred.
9  293 #
10 294 # In the example below the behavior will be to save:
11 295 # after 900 sec (15 min) if at least 1 key changed
12 296 # after 300 sec (5 min) if at least 10 keys changed
13 297 # after 60 sec if at least 10000 keys changed
14 298 #
15 299 # Note: you can disable saving completely by commenting out all "save"
16 300 #
17 301 # It is also possible to remove all the previously configured save
18 302 # points by adding a save directive with a single empty string argument
19 303 #
20 304 #
21 305 # save "" # 关闭了Rdb操作
```

### 相关配置项:

## Bash

```
1 # 配置保存策略, 虽然名称是save, 但是执行的是Bgsave策略, 即fork子进程, 去读数据, 写入磁
  盘;
2 save 3600 1 //900秒 至少一个key发生变化
3 save 300 10 //300秒 至少10个key发生变化
4 save 60 10000 //60秒 至少10000个key发生变化
5 # 如果想要关闭rdb功能, 那可以把上述配置删除掉, 或者配置一个save ""即可;
6 # The filename where to dump the DB
7 # 配置rdb持久化的文件名称
8 dbfilename dump.rdb, 一般的话修改为当前redis的端口号即可;
9 # 配置持久化文件的目录
10 dir /usr/local/games/ #以后会和aof共用此目录
```

- 触发机制

- save规则满足的情况下,会自动触发rdb规则;
- 执行flushall命令的时候, 也会触发rdb规则, 「数据都被清空了;没有啥意思;」
- 退出redis, 也就是使用shutdown命令的时候,也会触发rdb规则;
- 执行bgsave命令时候,也会触发rdb操作; 「手动触发」

- 恢复机制

只要将Rdb文件放在我们redis启动目录就可以了, redis启动的时候,会自动检查dump.rdb恢复其中的数据. **自动恢复.咱们不需要干预;**

- 使用场景及特性

1. 适合大规模的数据恢复.
2. 对数据完整性要求不高的时候使用;
3. 需要一定的时间间隔操作,如果redis意外挂了,那么最后一次修改的数据就没有;
4. fork进程的时候, 会占用一些内存空间;

- 重点内容

- rdb运行机制
  - fork子进程
    - 创建子进程速度非常的快.
  - copy on write
    - 子进程可以读取父进程当中的数据
    - 子进程修改数据无法影响父进程中的数据
    - 父进程修改数据也不会影响到子进程当中数据;

## 2. aof 「append only file」 「日志存储」

redis的写操作记录到文件中。「只有写入操作会存储」

以日志的形式记录每个操作, 将Redis执行过程中的所有指令都记录下来(读取操作并不记录), 只许追加文件, 但是不可以改写文件. redis启动之初会读取该文件重新构建数据. 换言之, redis重启之后的话就根据日志文件的内容将写指令从前到后执行一次完成数据的恢复工作;

set name tom

set age 20

- 默认情况下并不开启, 需要手动在配置文件里进行开启操作;

- 优点

- 丢失数据少, 根据不同的写入规则, 丢失的数据是不一样的. 默认的每一秒写入一次. 这样最多一秒.

- 缺点

- 数据恢复慢一些. 需要一条一条的执行记录里的命令.

- **rdb和aof可以同时开启, 但是如果开启了aof, 会使用aof的方式进行的数据恢复;**

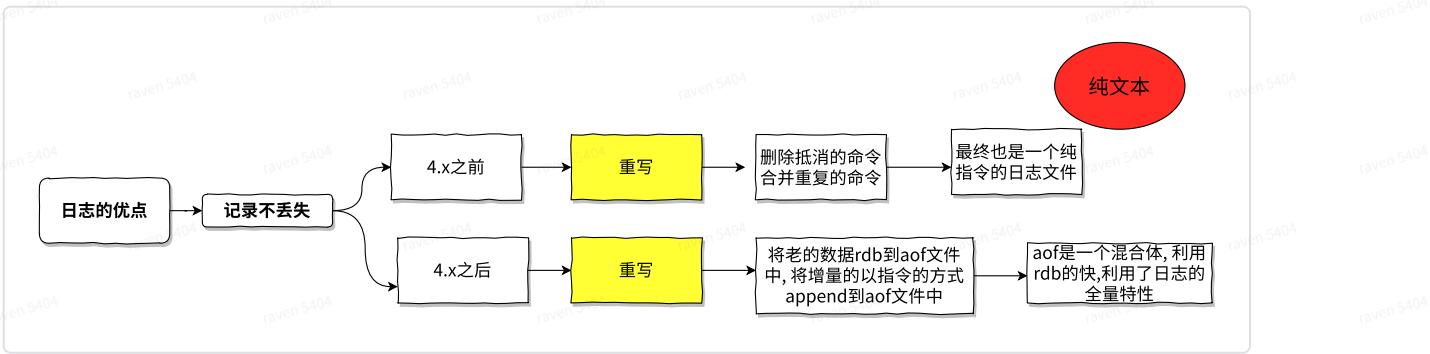
- **4.x版本之后, aof中包含rdb的全量操作.** 增加记录新的写操作, 这样在恢复的时候, 速度稍微快一些.

### XML

```
1 # 配置文件
2 appendonly yes # 开启aof配置, 默认值是no, 表示不开启. 使用的时候, 手动开启一下;
3 appendfilename "appendonly.aof" // aof文件名称. 存放的目录就是在rdb中配置的dir目录
4
5 set age 10
6 $3^M
7 set^M
8 $3^M
9 age^M
10 $2^M
11 10^M
12
13 *2$6SELECT$10*$3$3set$4name$4jack
14
15 把那个混合模式给关闭掉;否则默认就开启了混合模型: aof + rdb
16 63 aof-use-rdb-preamble no, 默认开启的.混合模式.把它关掉测试aof;
```

## 3. rdb与aof混合使用

只能使用aof进行恢复. 其中包含了rdb的全量数据;



**XML**

```

1 1118 # appendfsync always # 每一条调用一次flush, 刷新数据到硬盘上去
2 1119 appendfsync everysec # 每一秒调用一次
3 1120 # appendfsync no # 内核缓冲区写满了, 自动刷新
4
5 auto-aof-rewrite-percentage 100
6 auto-aof-rewrite-min-size 64mb

```

## 注意事项:

- 配置日志的输出路径

**Bash**

```

1 logfile 自己的路径

```

- 配置rdb文件的目录, 同时此目录也是aof文件的目录、

**Bash**

```

1 dir ...

```

- aof重写

**Bash**

```

1 bgsrewriteaof # 触发aof写入操作;

```

**Bash**

```

1 $>SELECT$1 0*3$3^Mset^M$3^Mage^M$2^M18^M*2^M 6
2 SELECT 0
3 set age 18
4

```

- 整个走一波

- 操作步骤

Bash

```
1 # 1. 修改配置文件.  
2 #     1. 关闭日志输出  
3 #     2. 关闭守护进程的方式, 使用前台启动的方式, 启动redis  
4 #     3. 开启rdb和aof, 关闭混合模式  
5 # 2. 测试aof和rdb  
6 #     1. 触发aof操作. 打开瞅一眼.生成的aof文件;  
7 #     2. 设置一些值, 再次打开aof文件. 看看生成的日志内容;  
8 #         * 号后的数字表示指令的条数  
9 #         $ 号后边表示字符个数  
10 #     3. 触发rdb  
11 #         查看rdb文件内容, 注意开头有大写的REDIS, 表示它是一个rdb文件;  
12 #         观察前台的输出;注意一下, pid号,看看是否开启了子进程. fork等信息;  
13 # 3. 开启混合模式  
14 #     1. 触发aof操作, 查看生成的日志文件;  
15 #     2. 设置一些重复的指令, 比如对一个key, 重复的设置值; 查看生成的日志文件;  
16 #     3. 触发重写操作, 此时, 会合并和抵消一些重复的指令. 对比一下生成的文件大小;  
17 #     4. 触发rdb操作. 查看aof文件和rdb文件  
18 #     5. 再次执行指令, 查看aof文件, 看看是否包含rdb文件. 即开头是REDIS字符串;  
19 #     6. 整理一下最后的逻辑;  
20 # 4. 执行flushall操作/flushdb等相关的操作产生的恶果;
```

**任何时候, 不要使用flush相关的命令. 再bgrewriteaof命令.**

**任何时候, 不要使用flush相关的命令. 再bgrewriteaof命令.**

**任何时候, 不要使用flush相关的命令. 再bgrewriteaof命令.**



redis是内存型数据库，如果不把redis的数据库状态保存到硬盘中的话，那么万一服务器故障了就会导致服务器数据丢失，因此需要进行持久化操作。

redis的持久化操作主要有两种，第一种是RDB方式，采用快照副本的方式对数据进行保存，保存的是全量数据，具有时点性，但是会产生数据丢失的问题。那它有一系列的触发机制，主要有这么四个，使用save命令会自动触发，然后使用flushall命令，退出redis的时候，以及使用bgsave命令的时候都会进行触发，当你需要恢复数据的时候，只需要把rdb文件放到redis的启动目录中，那么下次启动的时候，就会进行自动恢复

第二种方式是AOF方式，采用日志存储的方式，把一条一条的指令保存日志中，特点的话就是丢失数据少，但是恢复的速度相对较慢，需要一条一条的执行指令进行数据恢复，那它也有这么几个触发机制，每一条调用一次flush，每秒调用一次，还有就是内核缓冲区写满了会进行自动刷新，以及使用bgrewriteaof命令也会触发aof操作

第三种方式是RDB和AOF混合使用，这种方式只能使用AOF进行数据恢复，包含了RDB的全量数据，在4.x之前会删除抵消的命令，合并重复的命令，4.x之后是做了修改，AOF变成了一个混合体，即包含了RDB全量数据，也包含了指令形式的日志，恢复的时候，全量数据就按照RDB的方式恢复，指令按照AOF的方式进行恢复。相较于前面两种，速度变快了，丢失的数据也相对少了一些。那在混合使用的时候，**千万不要使用flush相关的命令，然后在bgrewriteaof,因为会导致数据无法恢复。**

- 配置参数说明

### Bash

```
1 appendonly yes # 开启aof
2
3 # 设置刷新策略
4 # appendfsync always
5 appendfsync everysec
6 # appendfsync no
7
8 # 默认的触发重写机制，执行bgrewriteaof
9 auto-aof-rewrite-percentage 100
10 auto-aof-rewrite-min-size 64mb
11
12 # 开启rdb与aof混合模式
13 aof-use-rdb-preamble yes
14
15 # 手动触发重写，命令
16 bgrewriteaof
```

- 相关参数配置「来源于网络」

## ◦ redis.conf, 配置文件

Bash

```
1 #redis.conf
2 # Redis configuration file example.
3 # ./redis-server /path/to/redis.conf
4
5 ##### INCLUDES #####
6 # 这在你有标准配置模板但是每个 redis 服务器又需要个性设置的时候很有用。
7 include /path/to/local.conf
8 include /path/to/other.conf
9
10 ##### GENERAL #####
11
12 # 是否在后台执行, yes: 后台运行; no: 不是后台运行 (老版本默认)
13 daemonize yes
14
15 #3.2 里的参数, 是否开启保护模式, 默认开启。要是配置里没有指定 bind 和密码。开启该参数
后, redis 只会本地进行访问, 拒绝外部访问。要是开启了密码 和 bind, 可以开启。否 则最
好关闭, 设置为 no。
16 protected-mode yes
17
18 #redis 的进程文件
19 pidfile /var/run/redis/redis-server.pid
20
21 #redis 监听的端口号。
22 port 6379
23
24 # 此参数确定了 TCP 连接中已完成队列(完成三次握手之后) 的长度, 当然此值必须不大于 Linux
系统定义的 /proc/sys/net/core/somaxconn 值, 默认是 511, 而 Linux 的默认参数值是
128。当系统并发量大并且客户端速度缓慢的时候, 可以将这二个参数一起参考设定。该内核参数默认
值一般是 128, 对于负载很大的服务程序来说大大的不够。一般会将它修改为 2048 或者更大。在 /
etc/sysctl.conf 中添加: net.core.somaxconn = 2048, 然后在终端中执行 sysctl -p。
25 tcp-backlog 511
26
27 # 指定 redis 只接收来自于该 IP 地址的请求, 如果不进行设置, 那么将处理所有请求
28 bind 127.0.0.1
29
30 # 配置 unix socket 来让 redis 支持监听本地连接。
31 # unixsocket /var/run/redis/redis.sock
32 # 配置 unix socket 使用文件的权限
33 # unixsocketperm 700
34
35 # 此参数为设置客户端空闲超过 timeout, 服务端会断开连接, 为 0 则服务端不会主动断开连接,
不能小于 0。
36 timeout 0
```

```
37
38 #tcp keepalive 参数。如果设置不为 0，就使用配置 tcp 的 SO_KEEPALIVE 值，使用
39 #keepalive 有两个好处：检测挂掉的对端。降低中间设备出问题而导致网络看似连接却已经与对端端
40 #口的问题。在 Linux 内核中，设置了 keepalive，redis 会定时给对端发送 ack。检测到对端关
41 #闭需要两倍的设置值。
42 tcp-keepalive 0
43
44 # 指定了服务端日志的级别。级别包括：debug（很多信息，方便开发、测试），verbose（许多有用
45 #的信息，但是没有 debug 级别信息多），notice（适当的日志级别，适合生产环境），warn（只有
46 #非常重要的信息）
47 loglevel notice
48
49 # 指定了记录日志的文件。空字符串的话，日志会打印到标准输出设备。后台运行的 redis 标准输
50 #出是 /dev/null。
51 logfile /var/log/redis/redis-server.log
52
53 # 是否打开记录 syslog 功能
54 # syslog-enabled no
55
56 # syslog 的标识符。
57 # syslog-ident redis
58
59 # 日志的来源、设备
60 # syslog-facility local0
61
62 # 数据库的数量，默认使用的数据库是 DB 0。可以通过”SELECT “命令选择一个 db
63 databases 16
64
65 ##### SNAPSHOTTING #####
66 # 快照配置
67
68 # 注释掉“save”这一行配置项就可以让保存数据库功能失效
69
70 # 设置 sedis 进行数据库镜像的频率。
71
72 # 900 秒 (15 分钟) 内至少 1 个 key 值改变 (则进行数据库保存 -- 持久化)
73 # 300 秒 (5 分钟) 内至少 10 个 key 值改变 (则进行数据库保存 -- 持久化)
74 # 60 秒 (1 分钟) 内至少 10000 个 key 值改变 (则进行数据库保存 -- 持久化)
75
76 # 当 RDB 持久化出现错误后，是否依然进行继续进行工作，yes：不能进行工作，no：可以继续进行
77 # 工作，可以通过 info 中的 rdb_last_bgsave_status 了解 RDB 持久化是否有错误
78 stop-writes-on-bgsave-error yes
79
80
81 # 使用压缩 rdb 文件，rdb 文件压缩使用 LZF 压缩算法，yes：压缩，但是需要一些 cpu 的消
82 #耗。no：不压缩，需要更多的磁盘空间
83
84 rdbcompression yes
85
86 # 且不垃圾收集，从而大大减少垃圾回收的频率。垃圾回收的频率越高，文件的生命周期上 CPU 的垃圾
```

```
raven 5404 76 # 延古仅当 rdb 写入。从 rdb 读取的时延会变大，但 rdb 写入的时延会变小。CRDB4 的影响  
raven 5404 和。这跟有利于文件的容错性，但是在保存 rdb 文件的时候，会有大概 10% 的性能损耗，所以如果你追求高性能，可以关闭该配置。  
raven 5404 77 rdbchecksum yes  
raven 5404 78  
raven 5404 79 #rdb 文件的名称  
raven 5404 80 dbfilename dump.rdb  
raven 5404 81  
raven 5404 82 # 数据目录，数据库的写入会在这个目录。rdb、aof 文件也会写在这个目录  
raven 5404 83 dir /var/lib/redis  
raven 5404 84  
raven 5404 85 ##### REPLICATION  
raven 5404 #####  
raven 5404 86 # 复制选项，slave 复制对应的 master。  
raven 5404 87 # slaveof <masterip> <masterport>  
raven 5404 88  
raven 5404 89 # 如果 master 设置了 requirepass，那么 slave 要连上 master，需要有 master 的密码才行。masterauth 就是用来配置 master 的密码，这样可以在连上 master 后进行认证。  
raven 5404 90 # masterauth <master-password>  
raven 5404 91  
raven 5404 92 # 当从库同主机失去连接或者复制正在进行，从机库有两种运行方式：1) 如果 slave-serve-  
raven 5404 stale-data 设置为 yes(默认设置)，从库会继续响应客户端的请求。2) 如果 slave-serve-  
raven 5404 stale-data 设置为 no，除去 INFO 和 SLAVOF 命令之外的任何请求都会返回一个错误”SYNC  
raven 5404 with master in progress”。  
raven 5404 93 slave-serve-stale-data yes  
raven 5404 94  
raven 5404 95 # 作为从服务器，默认情况下是只读的 (yes)，可以修改成 NO，用于写 (不建议)。  
raven 5404 96 slave-read-only yes  
raven 5404 97  
raven 5404 98 # 是否使用 socket 方式复制数据。目前 redis 复制提供两种方式，disk 和 socket。如果新的  
raven 5404 slave 连上来或者重连的 slave 无法部分同步，就会执行全量同步，master 会生成 rdb 文件。  
raven 5404 有 2 种方式：disk 方式是 master 创建一个新的进程把 rdb 文件保存到磁盘，再把磁盘上的  
raven 5404 rdb 文件传递给 slave。socket 是 master 创建一个新的进程，直接把 rdb 文件以 socket 的  
raven 5404 方式发给 slave。disk 方式的时候，当一个 rdb 保存的过程中，多个 slave 都能共享这个 rdb  
raven 5404 文件。socket 的方式就一个个 slave 顺序复制。在磁盘速度缓慢，网速快的情况下推荐用  
raven 5404 socket 方式。  
raven 5404 99 repl-diskless-sync no  
raven 5404 100  
raven 5404 101 #diskless 复制的延迟时间，防止设置为 0。一旦复制开始，节点不会再接收新 slave 的复制请求  
raven 5404 直到下一个 rdb 传输。所以最好等待一段时间，等更多的 slave 连上来。  
raven 5404 102 repl-diskless-sync-delay 5  
raven 5404 103  
raven 5404 104 #slave 根据指定的时间间隔向服务器发送 ping 请求。时间间隔可以通过  
raven 5404 repl_ping_slave_period 来设置，默认 10 秒。  
raven 5404 105 # repl-ping-slave-period 10  
raven 5404 106  
raven 5404 107 # 复制连接超时时间。master 和 slave 都有超时时间的设置。master 检测到 slave 上次发送  
raven 5404 的时间超过 repl-timeout，即认为 slave 离线，清除该 slave 信息。slave 检测到上次和
```

*master* 交互的时间超过 *repl-timeout*, 则认为 *master* 离线。需要注意的是 *repl-timeout* 需要设置一个比 *repl-ping-slave-period* 更大的值, 不然会经常检测到超时。

```
108 # repl-timeout 60
109
110 # 是否禁止复制 tcp 链接的 tcp nodelay 参数, 可传递 yes 或者 no。默认是 no, 即使用 tcp
nodelay。如果 master 设置了 yes 来禁止 tcp nodelay 设置, 在把数据复制给 slave 的时候,
会减少包的数量和更小的网络带宽。但是这也可能带来数据的延迟。默认我们推荐更小的延迟, 但
是在数据量传输很大的场景下, 建议选择 yes。
111 repl-disable-tcp-nodelay no
112
113 # 复制缓冲区大小, 这是一个环形复制缓冲区, 用来保存最新复制的命令。这样在 slave 离线的时
候, 不需要完全复制 master 的数据, 如果可以执行部分同步, 只需要把缓冲区的部分数据复制给
slave, 就能恢复正常复制状态。缓冲区的大小越大, slave 离线的时间可以更长, 复制缓冲区只有
在有 slave 连接的时候才分配内存。没有 slave 的一段时间, 内存会被释放出来, 默认 1m。
114 # repl-backlog-size 5mb
115
116 #master 没有 slave 一段时间会释放复制缓冲区的内存, repl-backlog-ttl 用来设置该时间长
度。单位为秒。
117 # repl-backlog-ttl 3600
118
119 # 当 master 不可用, Sentinel 会根据 slave 的优先级选举一个 master。最低的优先级的
slave, 当选 master。而配置成 0, 永远不会被选举。
120 slave-priority 100
121
122 #redis 提供了可以让 master 停止写入的方式, 如果配置了 min-slaves-to-write, 健康的
slave 的个数小于 N, master 就禁止写入。master 最少得有多少个健康的 slave 存活才能执行写
命令。这个配置虽然不能保证 N 个 slave 都一定能接收到 master 的写操作, 但是能避免没有足
够健康的 slave 的时候, master 不能写入来避免数据丢失。设置为 0 是关闭该功能。
123 # min-slaves-to-write 3
124
125 # 延迟小于 min-slaves-max-lag 秒的 slave 才认为是健康的 slave。
126 # min-slaves-max-lag 10
127
128 # 设置 1 或另一个设置为 0 禁用这个特性。
129 # Setting one or the other to 0 disables the feature.
130 # By default min-slaves-to-write is set to 0 (feature disabled) and
131 # min-slaves-max-lag is set to 10.
132
133 ##### SECURITY
134 #requirepass 配置可以让用户使用 AUTH 命令来认证密码, 才能使用其他命令。这让 redis 可以
使用在不受信任的网络中。为了保持向后的兼容性, 可以注释该命令, 因为大部分用户也不需要认证。
使用 requirepass 的时候需要注意, 因为 redis 太快了, 每秒可以认证 15w 次密码, 简单的密
码很容易被攻破, 所以最好使用一个更复杂的密码。
135 # requirepass foobared
136
137 # 把危险的命令给修改成其他名称。比如 CONFIG 命令可以重命名为一个很难被猜到的命令, 这样用
户不能使用, 而内部工具还能接着使用。
```

```
138 # rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52
139
140 # 设置成一个空的值，可以禁止一个命令
141 # rename-command CONFIG ""
142 ##### LIMITS
143
144 # 设置能连上 redis 的最大客户端连接数量。默认是 10000 个客户端连接。由于 redis 不区分
    连接是客户端连接还是内部打开文件或者和 slave 连接等，所以 maxclients 最小建议设置到
    32。如果超过了 maxclients，redis 会给新的连接发送'max number of clients reached'，
    并关闭连接。
145 # maxclients 10000
146
147 #redis 配置的最大内存容量。当内存满了，需要配合 maxmemory-policy 策略进行处理。注意
    slave 的输出缓冲区是不计算在 maxmemory 内的。所以为了防止主机内存使用完，建议设置的
    maxmemory 需要更小一些。
148 # maxmemory <bytes>
149
150 # 内存容量超过 maxmemory 后的处理策略。
151 #volatile-lru: 利用 LRU 算法移除设置过过期时间的 key。
152 #volatile-random: 随机移除设置过过期时间的 key。
153 #volatile-ttl: 移除即将过期的 key，根据最近过期时间来删除（辅以 TTL）
154 #allkeys-lru: 利用 LRU 算法移除任何 key。
155 #allkeys-random: 随机移除任何 key。
156 #noeviction: 不移除任何 key，只是返回一个写错误。
157 # 上面的这些驱逐策略，如果 redis 没有合适的 key 驱逐，对于写命令，还是会返回错误。redis
    将不再接收写请求，只接收 get 请求。写命令包括：set setnx setex append incr decr
    rpush lpush rpushx lpushx linsert lset rpoplpush sadd sinter sinterstore
    sunion sunionstore sdiff sdifftree zadd zincrby zunionstore zinterstore hset
    hsetnx hmset hincrby incrby decrby getset mset msetnx exec sort。
158 # maxmemory-policy noeviction
159
160 #lru 检测的样本数。使用 lru 或者 ttl 淘汰算法，从需要淘汰的列表中随机选择 sample 个
    key，选出闲置时间最长的 key 移除。
161 # maxmemory-samples 5
162
163 ##### APPEND ONLY MODE
164
165 # 默认 redis 使用的是 rdb 方式持久化，这种方式在许多应用中已经足够用了。但是 redis 如果
    中途宕机，会导致可能有几分钟的数据丢失，根据 save 来策略进行持久化，Append Only File
    是另一种持久化方式，可以提供更好的持久化特性。Redis 会把每次写入的数据在接收后都写入
    appendonly.aof 文件，每次启动时 Redis 都会先把这个文件的数据读入内存里，先忽略 RDB 文
    件。
166 appendonly no
167 #aof 文件名
168 appendfilename "appendonly.aof"
```

```
169
170 #aof 持久化策略的配置
171 #no 表示不执行 fsync, 由操作系统保证数据同步到磁盘, 速度最快。
172 #always 表示每次写入都执行 fsync, 以保证数据同步到磁盘。
173 #everysec 表示每秒执行一次 fsync, 可能会导致丢失这 1s 数据。
174 appendfsync everysec
175
176 # 在 aof 重写或者写入 rdb 文件的时候, 会执行大量 IO, 此时对于 everysec 和 always 的
  aof 模式来说, 执行 fsync 会造成阻塞过长时间, no-appendfsync-on-rewrite 字段设置为默
  认设置为 no。如果对延迟要求很高的应用, 这个字段可以设置为 yes, 否则还是设置为 no, 这样对
  持久化特性来说这是更安全的选择。设置为 yes 表示 rewrite 期间对新写操作不 fsync, 暂时存
  在内存中, 等 rewrite 完成后再写入, 默认为 no, 建议 yes。Linux 的默认 fsync 策略是 30
  秒。可能丢失 30 秒数据。
177 no-appendfsync-on-rewrite no
178
179 #aof 自动重写配置。当目前 aof 文件大小超过上一次重写的 aof 文件大小的百分之多少进行重
  写, 即当 aof 文件增长到一定大小的时候 Redis 能够调用 bgrewriteaof 对日志文件进行重写。
  当前 AOF 文件大小是上次日志重写得到 AOF 文件大小的二倍 (设置为 100) 时, 自动启动新的日志
  重写过程。
180 auto-aof-rewrite-percentage 100
181 # 设置允许重写的最小 aof 文件大小, 避免了达到约定百分比但尺寸仍然很小的情况还要重写
182 auto-aof-rewrite-min-size 64mb
183
184 #aof 文件可能在尾部是不完整的, 当 redis 启动的时候, aof 文件的数据被载入内存。重启可能
  发生在 redis 所在的主机操作系统宕机后, 尤其在 ext4 文件系统没有加上 data=ordered 选项
  (redis 宕机或者异常终止不会造成尾部不完整现象。) 出现这种现象, 可以选择让 redis 退出,
  或者导入尽可能多的数据。如果选择的是 yes, 当截断的 aof 文件被导入的时候, 会自动发布一个
  log 给客户端然后 load。如果是 no, 用户必须手动 redis-check-aof 修复 AOF 文件才可以。
185 aof-load-truncated yes
186
187 ##### LUA SCRIPTING #####
188 # 如果达到最大时间限制 (毫秒), redis 会记个 log, 然后返回 error。当一个脚本超过了最大
  时限。只有 SCRIPT KILL 和 SHUTDOWN NOSAVE 可以用。第一个可以杀没有调 write 命令的东
  西。要是已经调用了 write, 只能用第二个命令杀。
189 lua-time-limit 5000
190
191 ##### REDIS CLUSTER #####
192 # 集群开关, 默认是不开启集群模式。
193 # cluster-enabled yes
194
195 # 集群配置文件的名称, 每个节点都有一个集群相关的配置文件, 持久化保存集群的信息。这个文件
  并不需要手动配置, 这个配置文件有 Redis 生成并更新, 每个 Redis 集群节点需要一个单独的配置
  文件, 请确保与实例运行的系统中配置文件名称不冲突
196 # cluster-config-file nodes-6379.conf
197
198 # 节点互连超时的阀值。集群节点超时毫秒数
199 # cluster-node-timeout 15000
200
```

```
raven 201 # 在进行故障转移的时候，全部 slave 都会请求申请为 master，但是有些 slave 可能与  
raven 201 master 断开连接一段时间了，导致数据过于陈旧，这样的 slave 不应该被提升为 master。该参数  
raven 201 就是用来判断 slave 节点与 master 断线的时间是否过长。判断方法是：  
raven 202 # 比较 slave 断开连接的时间和(node-timeout * slave-validity-factor) + repl-ping-  
raven 202 slave-period  
raven 203 # 如果节点超时时间为三十秒，并且 slave-validity-factor 为 10，假设默认的 repl-ping-  
raven 203 slave-period 是 10 秒，即如果超过 310 秒 slave 将不会尝试进行故障转移  
raven 204 # cluster-slave-validity-factor 10  
raven 205  
raven 206 #master 的 slave 数量大于该值，slave 才能迁移到其他孤立 master 上，如这个参数若被设为  
raven 206 2，那么只有当一个主节点拥有 2 个可工作的从节点时，它的一个从节点会尝试迁移。  
raven 207 # cluster-migration-barrier 1  
raven 208  
raven 209 # 默认情况下，集群全部的 slot 有节点负责，集群状态才为 ok，才能提供服务。设置为 no，可以  
raven 209 在 slot 没有全部分配的时候提供服务。不建议打开该配置，这样会造成分区的时候，小分区的  
raven 209 master 一直在接受写请求，而造成很长时间数据不一致。  
raven 210 # cluster-require-full-coverage yes  
raven 211  
raven 212 ##### SLOW LOG  
raven 212 #####  
raven 213 ####slog log 是用来记录 redis 运行中执行比较慢的命令耗时。当命令的执行超过了指定时间，就  
raven 213 记录在 slow log 中，slog log 保存在内存中，所以没有 IO 操作。  
raven 214 # 执行时间比 slowlog-log-slower-than 大的请求记录到 slowlog 里面，单位是微秒，所以  
raven 214 1000000 就是 1 秒。注意，负数时间会禁用慢查询日志，而 0 则会强制记录所有命令。  
raven 215 slowlog-log-slower-than 10000  
raven 216  
raven 217 # 慢查询日志长度。当一个新的命令被写进日志的时候，最老的那个记录会被删掉。这个长度没有限  
raven 217 制。只要有足够的内存就行。你可以通过 SLOWLOG RESET 来释放内存。  
raven 218 slowlog-max-len 128  
raven 219  
raven 220 ##### LATENCY MONITOR  
raven 220 #####  
raven 221 # 延迟监控功能是用来监控 redis 中执行比较缓慢的一些操作，用 LATENCY 打印 redis 实例在  
raven 221 跑命令时的耗时图表。只记录大于等于下边设置的值的操作。0 的话，就是关闭监视。默认延迟监控  
raven 221 功能是关闭的，如果你需要打开，也可以通过 CONFIG SET 命令动态设置。  
raven 222 latency-monitor-threshold 0  
raven 223  
raven 224 ##### EVENT NOTIFICATION  
raven 224 #####  
raven 225 # 键空间通知使得客户端可以通过订阅频道或模式，来接收那些以某种方式改动了 Redis 数据集的  
raven 225 事件。因为开启键空间通知功能需要消耗一些 CPU，所以在默认配置下，该功能处于关闭状态。  
raven 226 #notify-keyspace-events 的参数可以是以下字符的任意组合，它指定了服务器该发送哪些类型的  
raven 226 通知：  
raven 227 ##K 键空间通知，所有通知以 __keyspace@__ 为前缀  
raven 228 ##E 键事件通知，所有通知以 __keyevent@__ 为前缀  
raven 229 ##g DEL 、 EXPIRE 、 RENAME 等类型无关的通用命令的通知  
raven 230 ##$ 字符串命令的通知  
raven 231 ### 列表命令的通知
```

```
201      "#$s 集合命令的通知  
232      ##$h 哈希命令的通知  
233      ##$z 有序集合命令的通知  
234      ##$x 过期事件：每当有过期键被删除时发送  
235      ##$e 驱逐(evict) 事件：每当有键因为 maxmemory 政策而被删除时发送  
236      ##$A 参数 g$lshzxe 的别名  
237      # 输入的参数中至少要有一个 K 或者 E，否则的话，不管其余的参数是什么，都不会有任何 通知被  
分发。详细使用可以参考 http://redis.io/topics/notifications  
238  
239      notify-keyspace-events ""  
240  
241      ##### ADVANCED CONFIG  
242      #####  
243      # 数据量小于等于 hash-max-ziplist-entries 的用 ziplist, 大于 hash-max-ziplist-  
entries 用 hash  
244      hash-max-ziplist-entries 512  
245      #value 大小小于等于 hash-max-ziplist-value 的用 ziplist, 大于 hash-max-ziplist-  
value 用 hash。  
246      hash-max-ziplist-value 64  
247  
248      # 数据量小于等于 list-max-ziplist-entries 用 ziplist, 大于 list-max-ziplist-  
entries 用 list。  
249      list-max-ziplist-entries 512  
250      #value 大小小于等于 list-max-ziplist-value 的用 ziplist, 大于 list-max-ziplist-  
value 用 list。  
251      list-max-ziplist-value 64  
252  
253      # 数据量小于等于 set-max-intset-entries 用 iniset, 大于 set-max-intset-entries 用  
set。  
254      set-max-intset-entries 512  
255  
256      # 数据量小于等于 zset-max-ziplist-entries 用 ziplist, 大于 zset-max-ziplist-  
entries 用 zset。  
257      zset-max-ziplist-entries 128  
258      #value 大小小于等于 zset-max-ziplist-value 用 ziplist, 大于 zset-max-ziplist-  
value 用 zset。  
259      zset-max-ziplist-value 64  
260  
261      #value 大小小于等于 hll-sparse-max-bytes 使用稀疏数据结构 (sparse) , 大于 hll-  
sparse-max-bytes 使用稠密的数据结构 (dense) 。一个比 16000 大的 value 是几乎没用的，  
建议的 value 大概为 3000。如果对 CPU 要求不高，对空间要求较高的，建议设置到 10000 左  
右。  
262      hll-sparse-max-bytes 3000  
263  
264      #Redis 将在每 100 毫秒时使用 1 毫秒的 CPU 时间来对 redis 的 hash 表进行重新 hash, 可  
以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受 Redis 时不时的对请  
求有 2 毫秒的延迟的话，把这项配置为 no。如果没有这么严格实时性要求，可以设置为 yes, 以
```

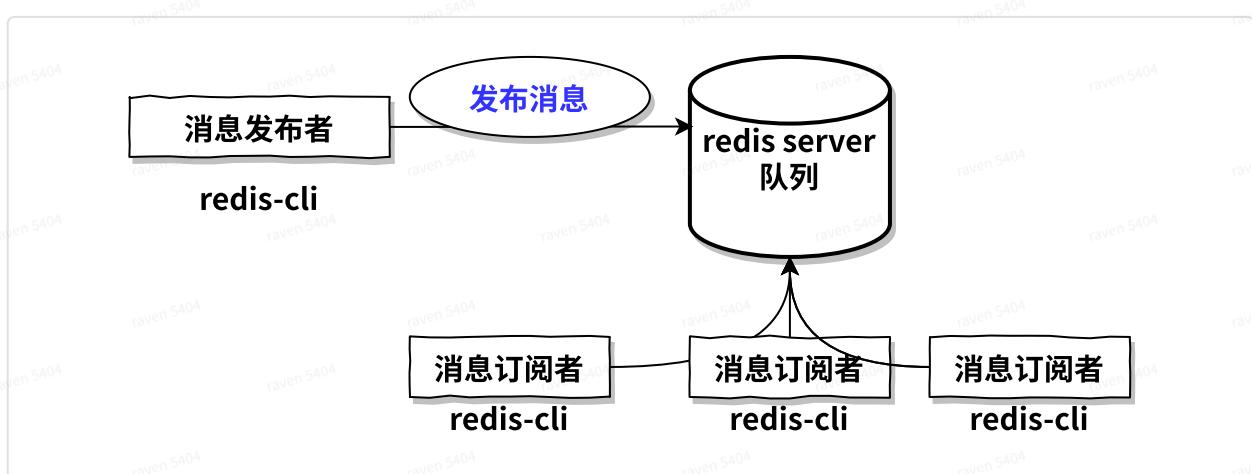
```

    便能够尽可能快的释放内存。
265 activerehashing yes
266
267 ## 对客户端输出缓冲进行限制可以强迫那些不从服务器读取数据的客户端断开连接，用来强制关闭传输缓慢的客户端。
268 # 对于 normal client，第一个 0 表示取消 hard limit，第二个 0 和第三个 0 表示取消 soft limit，normal client 默认取消限制，因为如果没有寻问，他们是不会接收数据的。
269 client-output-buffer-limit normal 0 0 0
270 # 对于 slave client 和 MONITER client，如果 client-output-buffer 一旦超过 256mb，又或者超过 64mb 持续 60 秒，那么服务器就会立即断开客户端连接。
271 client-output-buffer-limit slave 256mb 64mb 60
272 # 对于 pubsub client，如果 client-output-buffer 一旦超过 32mb，又或者超过 8mb 持续 60 秒，那么服务器就会立即断开客户端连接。
273 client-output-buffer-limit pubsub 32mb 8mb 60
274
275 #redis 执行任务的频率为 1s 除以 hz。
276 hz 10
277
278 # 在 aof 重写的时候，如果打开了 aof-rewrite-incremental-fsync 开关，系统会每 32MB 执行一次 fsync。这对于把文件写入磁盘是有帮助的，可以避免过大的延迟峰值。
279 aof-rewrite-incremental-fsync yes

```

## 第七章: 订阅发布

redis发布订阅(pub/sub)是一种消息通信模型. 发送者(pub)发送消息. 订阅者(sub)接收消息;



- 几个重要的概念

- 消息发布者
  - 谁发送的消息
- 频道
  - 双方公认频道, 数据往哪里发
- 接收者
- 重要的api

命令名称	作用
subscribe channel [channel ...]	<ul style="list-style-type: none"> <li>• subscribe: 表示我们成功订阅到响应的第二个元素提供的频道。第三个参数代表我们现在订阅的频道的数量。</li> </ul>
unsubscribe channel [channle]	<ul style="list-style-type: none"> <li>• unsubscribe: 表示我们成功取消订阅到响应的第二个元素提供的频道。第三个参数代表我们目前订阅的频道的数量。当最后一个参数是0的时候，我们不再订阅到任何频道。当我们在Pub/Sub以外状态，客户端可以发出任何redis命令。</li> </ul>
message	message: 这是另外一个客户端发出的发布命令的结果。第二个元素是来源频道的名称，第三个参数是实际消息的内容

## Bash

```

1 PSUBSCRIBE pattern [pattern ...]
2 summary: Listen for messages published to channels matching the given
   patterns
3 since: 2.0.0
4
5 PUBLISH channel message
6 summary: Post a message to a channel
7 since: 2.0.0
8
9 PUBSUB subcommand [argument [argument ...]]
10 summary: Inspect the state of the Pub/Sub subsystem
11 since: 2.8.0
12
13 PUNSUBSCRIBE [pattern [pattern ...]] # 正则表达式。*号表示所有
14 summary: Stop listening for messages posted to channels matching the given
   patterns
15 since: 2.0.0
16
17 SUBSCRIBE channel [channel ...]
18 summary: Listen for messages published to the given channels
19 since: 2.0.0
20
21 UNSUBSCRIBE [channel [channel ...]]
22 summary: Stop listening for messages posted to the given channels
23 since: 2.0.0

```

## 第八章: 主从复制

<http://redis.cn/topics/replication.html>



- 主备
  - 客户端只能访问主机,不会使用备机,当主机挂掉的时候,备机顶上.备机不参与业务;
- 主从
  - 客户端可以访问主机,也可以访问从机;
  - 主机接收写操作
  - 从机接收读操作

## 1. 单机单节点/单实例的问题



- 单点故障
- 容量有限
- 并发压力

### 1. AKF解决方案

AKF 扩展立方体 (AKF Scale Cube) 是一个描述从单体应用到一个分布式可扩展架构的模型概念.

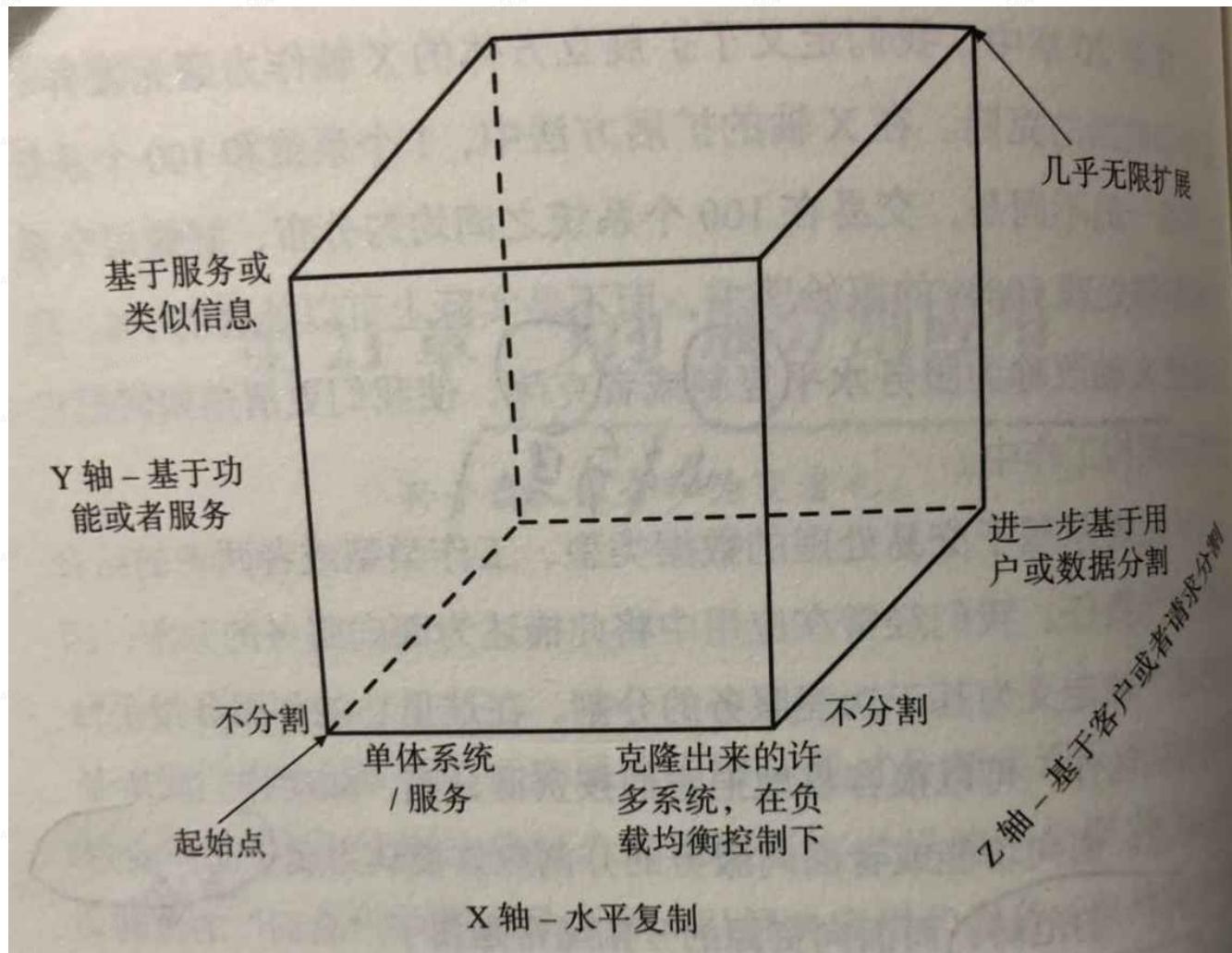
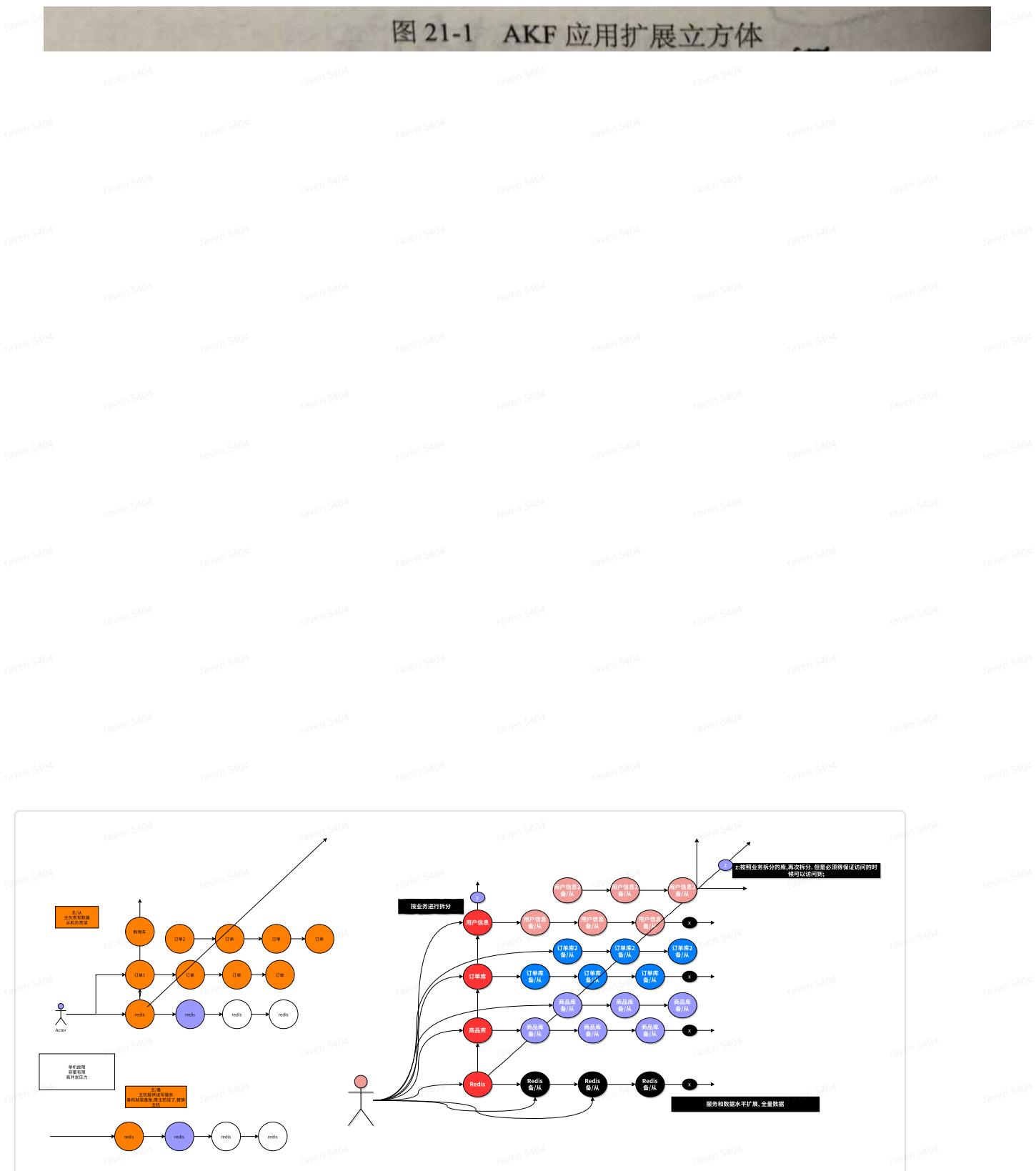


图 21-1 AKF 应用扩展立方体



## 2. AKF详细解决方案



## 1. 「x轴方向解决」 - 解决单点故障问题.

- a. 按照主备设计, 主机负责读写操作, 备机只做故障时候的切换.
- b. 主机和备机存放的都是全量数据;
- c. **无法解决单机容量和并发压力问题;**

改进操作: 主备换成主从, 主机负责写操作, 各个从机负责读操作. 这样备机的资源不会被浪费掉;

## 2. 「y轴方向解决」 - 单机容量和并发压力问题

### a. 将redis按业务进行划分.

- i. 如根据模块的不同, 拆分成不同的redis库, 如订单库, 商品库等;
- b. 可以对y轴进行x轴方向的水平扩展, 即做「备/从」操作;

## 3. 「z轴方向解决」 - y轴方向上的单机容量和并发压力问题

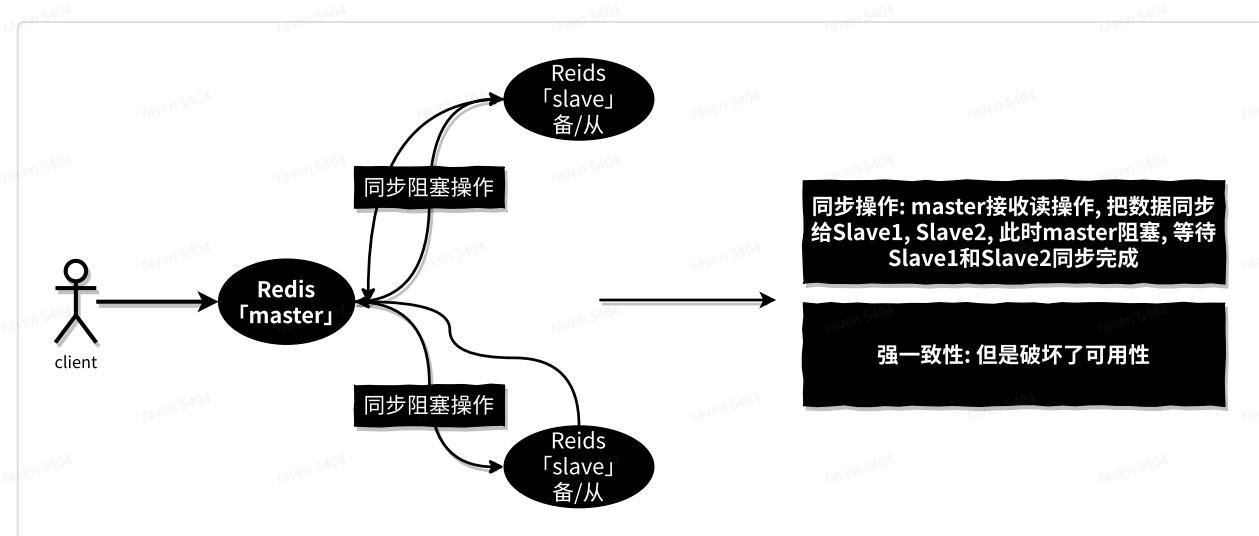
- a. 单y轴被拆分成单个实例, 达到一定规则之后, 容量和单机访问压力就会增大;
- b. 此时将已经按业务拆分的redis库, 再次进行拆分. 「粒度再细一些」
- c. **如把订单库, 再次拆分, 但是要保证访问的时候, 可以访问得到;**

## 3. AKF拆分之后产生的问题

拆分之后, redis从单一节点变成了多节点. 从而带来了数据同步问题;

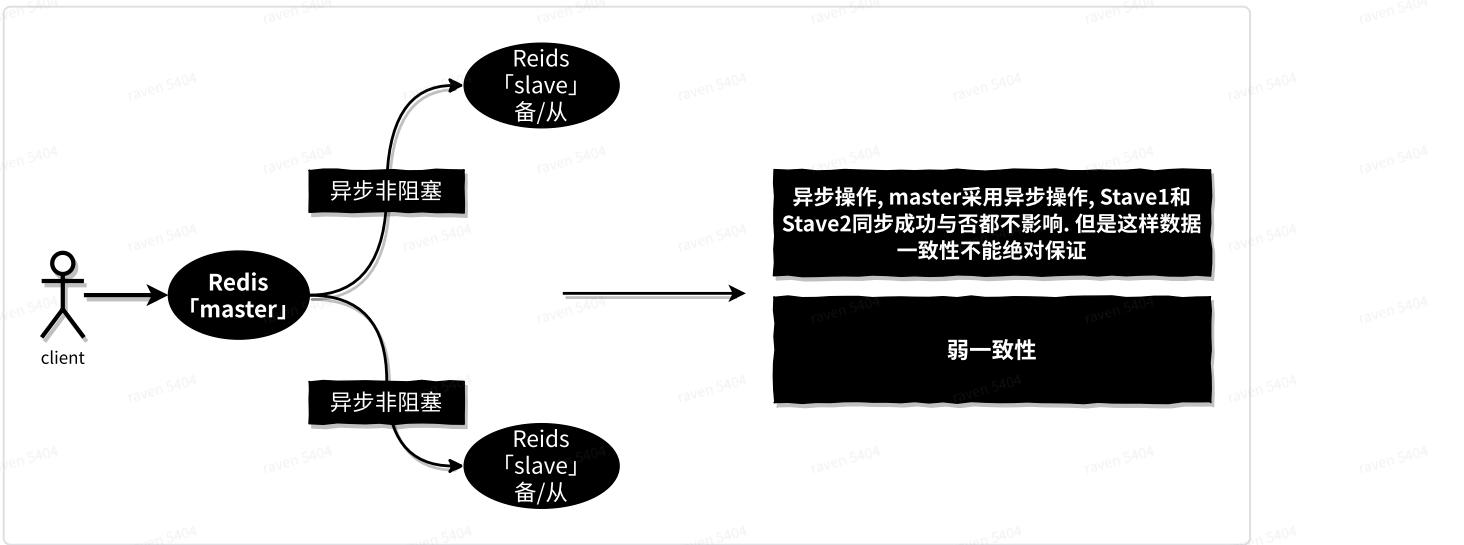
### 数据同步方案:

- 强一致性
  - 所有节点阻塞直到所有节点数据全部一致;
  - 同步阻塞方案, 但是会破坏可用性;



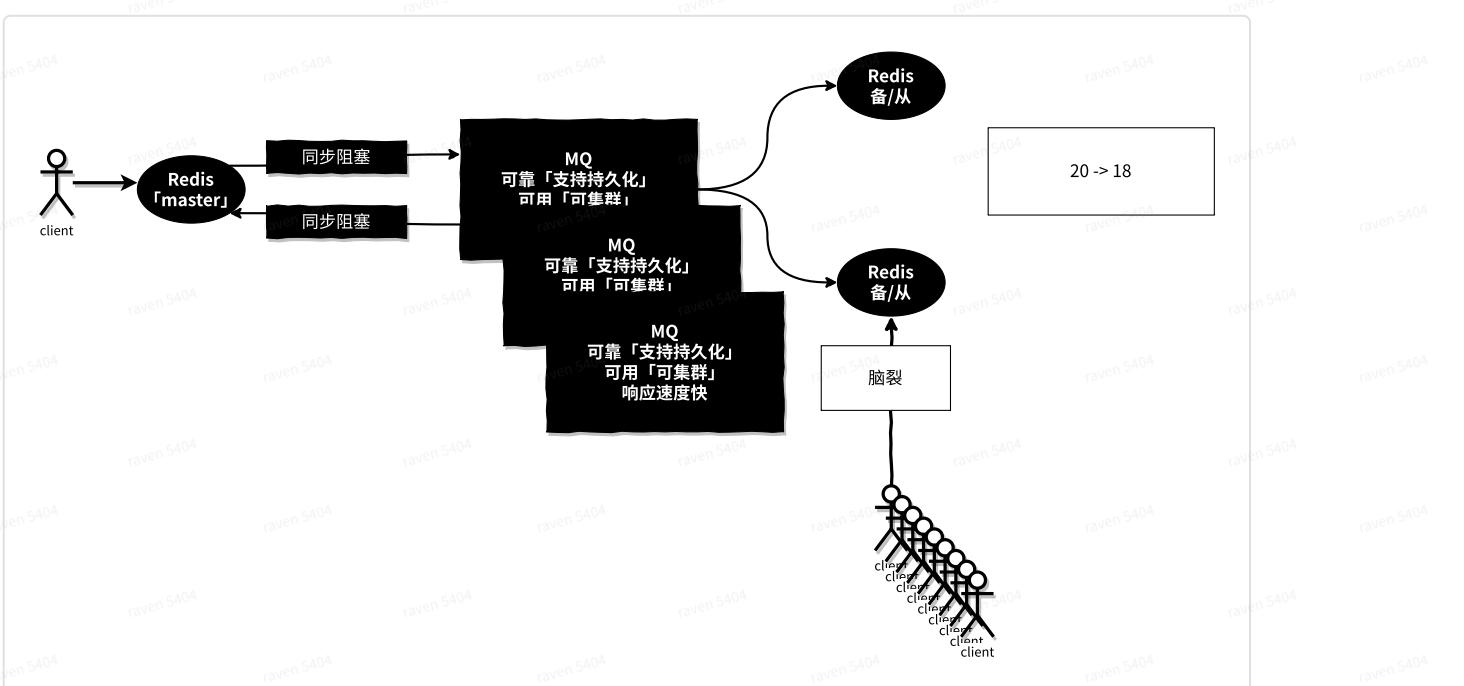
### • 弱一致性

- 异步非阻塞方案, 可能会有部分数据丢失;



- 最终一致性

- 同步非阻塞



- 最终一致性会产生问题. 当客户端访问不同的结果的时候, 数据可能会存在不一致性问题;
- 可以通过方案强调成强一致性;

## 2. Redis主从复制

REDIS sentinel-old -- Redis中国用户组(CRUG)

redis

<http://redis.cn/topics/replication.html>

主从复制, 是指将一台Redis服务器的数据. 复制到其他的redis服务器. 前者称为主节点「master」, 后者称为从节点「slave」; 复制数据是单向的, 只能由主节点到从节点. Master以写为主, Slave以读为主;

默认情况下,每台redis服务器都是主节点;且一个主节点可以有多个从节点「或者没有从节点」,但是一个从节点只能有一个主节点;

1. 数据冗余: 主从复制实现了数据的热备份, 是持久化之外一种数据冗余方式;

- 故障恢复: 当主节点出现问题时, 可以由从节点提供服务, 实现快速的故障恢复,; 实际上是一种服务的冗余;

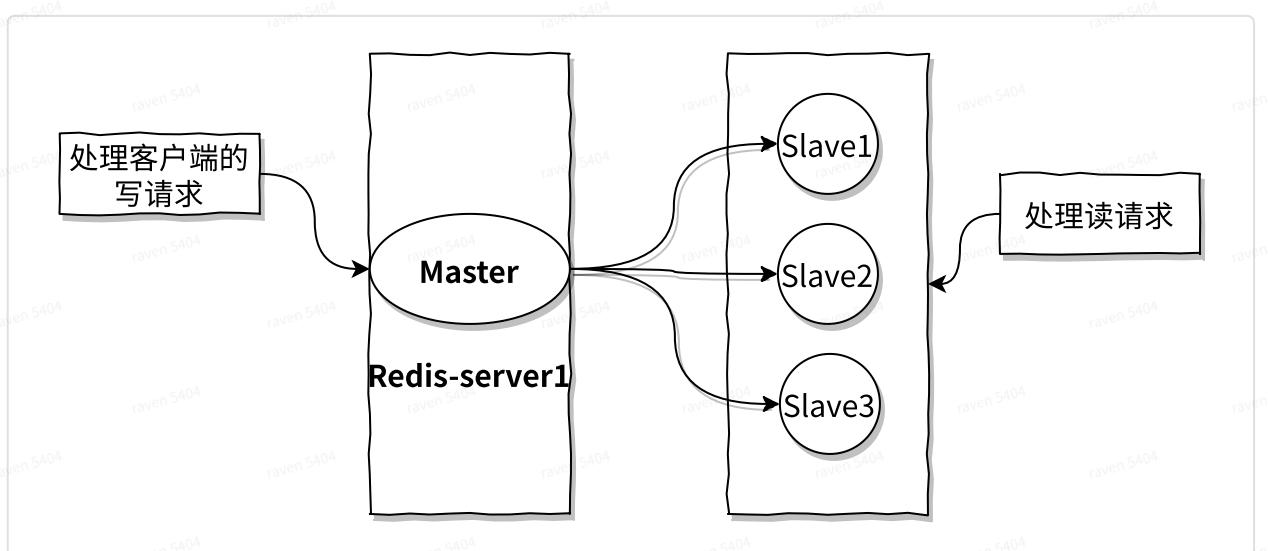
2. 负载均衡: 在主从复制的基础上, 配合读写分离, 可以由主节点提供「写」服务, 由从节点提供「读」服务. 「即写redis数据时应用连接主节点, 读redis数据时应用连接从节点」分担服务器负载; 尤其是在写少读多的场景下, 通过多个从节点分担「读」负载, 可以大大提高redis服务器的并发量;

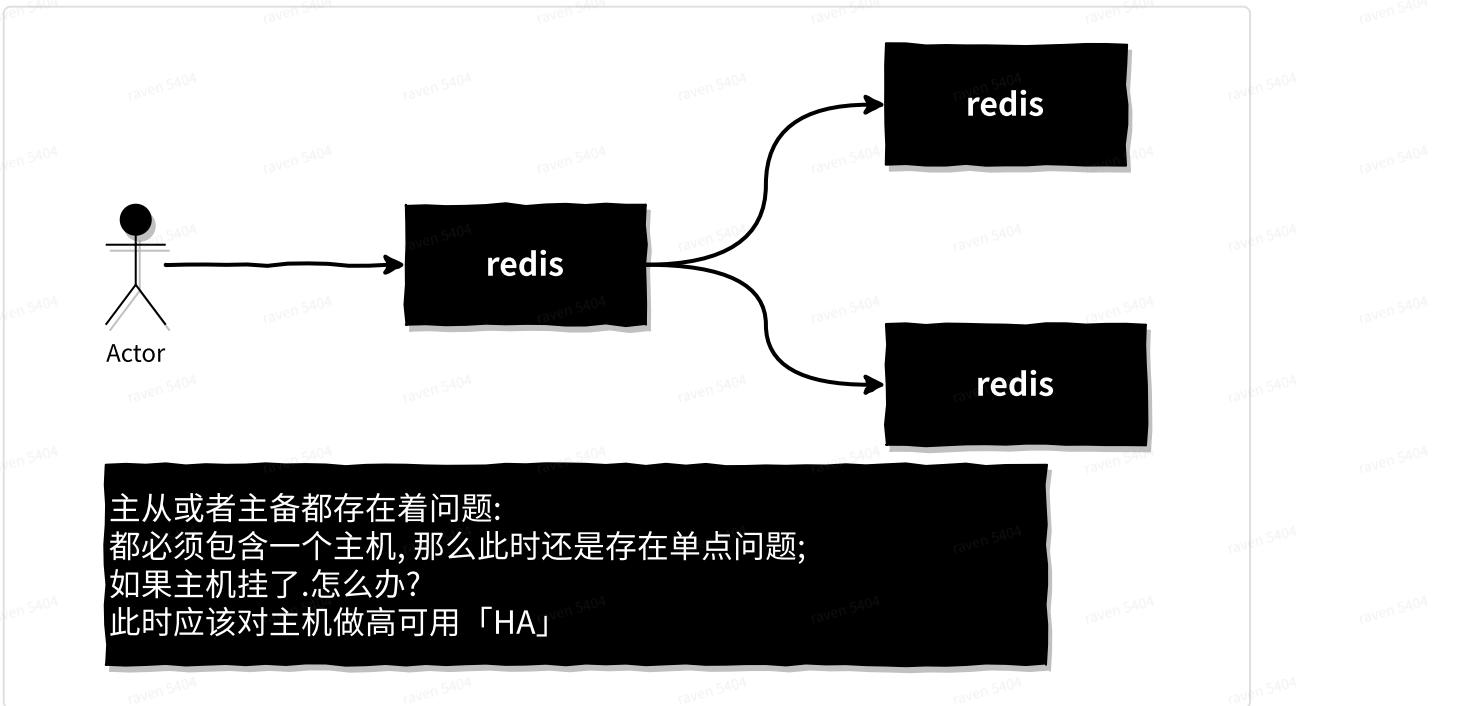
3. 高可用用基石: 除了上述作用之外, 主从复制还是哨兵和集群能够实施的基础, 因此说主从复制是redis的高可用的基础;

- 单节点的缺点

a. 从节点上, 单个redis服务器会发生单点故障, 并且一台服务器需要处理所有的请求负载, 压力较大;

b. 从容量上, 单个redis服务器内存容量有限, 就算一台redis服务器内存容量为256G, 也不能将所有内存用作redis存储内存. 一般来说, 单台redis容量最大使用内存应该不超过20G;





主从或者主备都存在着问题:  
都必须包含一个主机, 那么此时还是存在单点问题;  
如果主机挂了.怎么办?  
此时应该对主机做高可用「HA」

## REDIS sentinel-old -- Redis中国用户组(CRUG)

redis

<http://redis.cn/topics/sentinel.html>

强调对外的表现,看起来不出现问题;而不是强调不挂掉;

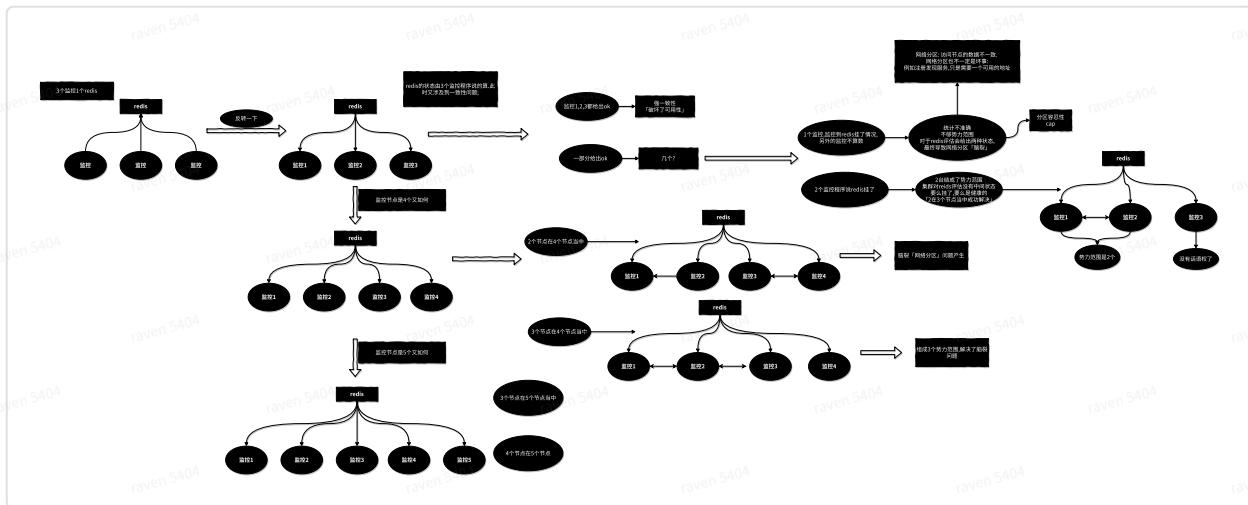
- 人工可以做
- 自动故障转移,对外高可用;

对主做HA「高可用」  
自动的故障转移, 代替人的操作

对程序进行监控

技术实现,程序层面上解决问题  
只要是一个程序就会有单点故障问题,所以监控的程序  
应该也是一个集群;

## ● 监控程序



总结一下：

- 监控程序n /2 + 1台;
    - 奇数和偶数台,所承担的风险一样.偶数承担风险高一些;
  - CAP原则

baike.baidu.com

<https://baike.baidu.com/item/CAP%E5%8E%9F%E5%88%99#:~:text=CAP%E5%8E%9F%E5%88%99%E5%8F%88%E7%...>

## 1. 实操主从复制

- ### • 配置环境

Bash

```
1 # 1. 在redis安装目录中,复制6379.conf两份, 文件名称是: 6379.conf, 6380.conf,  
2 6381.conf  
3 # 2. 修改配置文件  
4 # 把三个配置文件, 修改成对应的端口号即可;  
5 # 1. 关闭后台启动: daemonize no  
6 # 2. 修改配置对应的端口号: port: 6379  
7 # 3. 更改pid名称: pidfile /var/run/redis_6379.pid  
8 # 4. 更改logfile名称: logfile "/usr/local/games/6379.log" , 这一行先注释掉  
9 # 5. 更改rdb文件名称: dbfilename 6379.rdb  
10 # 6. 更改aof文件名称: appendfilename "6379.aof"  
11 # 7. 关闭aof: appendonly no  
12 # 8.对于Redis集群, 如果设置了requirepass密码, 则一定要设置masterauth密码, 否则从节点无法正常工作  
13 ##### 补充部分 #####  
14 实操补充:  
15 # 1. 在redis安装目录中,复制6379.conf两份原来的名称是「redis.conf」, 复制一份更改名字:  
16 6379.conf -> 6379.conf_6380.conf_6381.conf
```

```
0579.COM」，入门口你走。0579.COM, 0508.COM, 0501.COM
16 # 2. 修改配置文件
17 # 把三个配置文件，修改成对应的端口号即可；
18 # 1. 关闭后台启动: daemonize no
19 # 2. 修改配置对应的端口号: port: 6379
20 # 3. 更改pid名称: pidfile /var/run/redis_6379.pid
21 # 4. 更改logfile名称: logfile "/usr/local/games/6379.log"，这一行先注释掉
22 # 5. 更改rdb文件名称: dbfilename 6379.rdb
23 # 6. 更改aof文件名称: appendfilename "6379.aof"
24 # 7. 关闭aof: appendonly no
25 # 8. 对于Redis集群，如果设置了requirepass密码，则一定要设置masterauth密码，否则从节点无法正常工作
26
27 # 操作步骤：
28 # 1. 启动三个redis服务「指定配置文件的方式启动」；
29 # 2. 查看三个`redis`实例是否都开启了: ps -ef | grep redis 或者netstat -nltp，查看对应的端口号
30 # 3. 启动三个客户端，分别连接三个 `redis` 实例: ./redis-cli -p 「redis服务端口号」 -a redis密码
31 # 例如：连接：6379服务。可以使用此种方式: ./redis-cli -p 6379 -a 密码
32 # 4. 查看每个`redis`实例的角色信息；「特别注意：在连接的客户端中操作」
33 # 使用命令: info replication, 默认redis实例都是master，没有从机数量；
34 # 5. 在`6379`的`redis`实例上设置一些数据；
35 # Set name tom
36 # set age 18
37 # 6. 让`6380`去跟随`6379`的`redis`实例；
38 # 就是把`6379`当作主机「master」。6380作为从机「slave」
39 # 在6380的客户端中进行操作，使用如下的命令：
40 # replicaof 主机的ip地址 主机的端口号，这里可以写成: replicaof 127.0.0.1
41 # 6379
42 # 因为三台实例，都在同一台物理机上，所以可以写：127.0.0.1作为主机的地址，如果三
43 # 台机器不在同一台物理机上，则需要写上对应机器的公网ip；
44 # 7. 设置完成之后，分别查看：6379和6380服务的输出日志；
45 # 8. 在6380中，查看是否已经有6379同步的数据；
46 # keys *
47 # get name
48 # 9. 分别查看6379和6380的角色信息。「info replication」
49 # 10. 查看落盘的`rdb`文件
50 # 打开6380.rdb，其中注意查看一下，保存了它所跟随的主机id信息；此时只开启了rdb
51 # 持久化方式，并未开启aof；
52 # 11. 此时，在6379中写入一些数据，查看6380的日志信息，查看6380中是否已经同步了数据；
53 # 12. 把6381也设置为6379的从机；
54 # replicaof 127.0.0.1 6379
55 # 查看，6381是否已经同步了数据；查看对应的日志输出；
56 # 13. 关闭6380服务，然后再重新启动6380服务，看看数据恢复情况；
```

```
raven 57      这里非常的重要,尤其是日志.一定要仔细查看;
raven 58 # 15. 再次断开6380, 在次启动6380, 在6380启动的时候,追加参数: --appendonly yes, 即开
raven 59      启aof
raven 59      # 使用如下命令, 指定跟随的主机,并且开启aof;
raven 60      # ./redis-server ../6380.conf replicaof 127.0.0.1 6379 --appendonly
raven 60      yes
raven 61      # 在生成aof文件当中,没有了主从关系.也就是replica_id没有了.
raven 62 # 16. 在6379中写入一些数据,查看同步情况;
raven 63 # 17. 查看落盘文件, 重点查看: 6380.aof 「启动的时候开启了aof」 .
raven 64 # 注意一下, 在aof文件中,它所跟随的主机id不见了; “rdb文件当中是保存了这个主机
raven 64      的id的.”
raven 65 # 18. 断开6379, 查看两个从机情况. 然后再重新启动6379. 观察两台从机是否会自动连接主机;
raven 66 # 19. 如果想把6380作为主机, 让6381去跟随6380. 「此时假设6379永久的挂掉了;」
raven 67 #
raven 68 #          操作步骤:
raven 68 #          1. 在6380客户端中执行如下的命令:
raven 69 #          replicaof no one 「谁也不跟随,我就是主机」
raven 70 #          2. 在6381中, 让6381去跟随6380;
raven 71 #          replicaof 127.0.0.1 6380
raven 72 # 20. 79也可以跟随80, 数据也恢复了.
raven 73
raven 74 # 命令总结
raven 75
```

## Bash

```
1 重要的命令:  
2 指定配置文件启动  
3     redis服务  
4     ./redis-service ../6379.conf, 「配置文件目录和名称写自己哦」  
5     redis安装目录,src目录中执行;  
6 查看redis的启动情况  
7     ps -ef | grep redis  
8     netstat -ntlp, 查看相关端口和pid情况  
9 客户端连接指定的  
10    redis服务  
11    命令格式:  
12    ./redis-cli -p 端口号 -a redis密码  
13    这里的密码是指在配置文件当中配置的: requirepass 处的密码.  
14 例如, 连接端口号是6379的  
15    redis服务,可以写成如下的格式  
16    ./redis-cli -a Tom123 -p 6379  
17    redis安装目录,src目录中执行;  
18 查看redis实例的角色信息  
19     info replication  
20 在客户端中去执行;  
21 在客户端窗口中设置一台redis实例跟随另外一台redis实例例如: 设置  
22 6380跟随6379, 即把6379作为主机, 把6380作为从机;  
23     设置方式: 在6380的客户端的窗口中执行如下命令:  
24     replicaof 127.0.0.1 6379  
25 重要说明  
26 这里的ip地址,因为三个实例都在一台物理机上,所以可以写: 127.0.0.1.这里的ip地址是指主机的  
ip地址和端口号;  
27 启动redis时, 指定跟随的主机  
28     ./redis-server ../6380.conf --replicaof 127.0.0.1 6379  
29 启动redis的时候,开启aof  
30     ./redis-server ../6380.conf --appendonly yes  
31 启动redis时, 指定跟随的主机并且开启aof支持;  
32     ./redis-server ../6380.conf replicaof 127.0.0.1 6379 --appendonly yes  
33 把某一个实例设置为主机「master」  
34     replicaof no one 「谁也不跟随,就是主机」
```

### 1. 前台启动的方式分别开启三个redis服务, 指定不同的配置文件;

我本机上的是这个路径,同学们自己测试的时候,改成具体的路径;

- redis的安装目录下的src目录去执行

## Bash

```
1 ./redis-server ../6379.conf  
2 ./redis-server ../6380.conf  
3 ./redis-server ../6381.conf
```

```
[root@iZ2ze8biiph4vi0x5z06heZ src]# ./redis-server ../6379.conf  
26922:C 07 May 2021 23:53:55.566 * o000o000o000o Redis is starting o000o000o000o  
26922:C 07 May 2021 23:53:55.567 * Redis version=6.0.9, bits=64, commit=00000000, modified=0, pid=26922,  
26922:C 07 May 2021 23:53:55.567 * Configuration loaded  
  
Redis 6.0.9 (00000000/0) 64 bit  
Running in standalone mode  
Port: 6379  
PID: 26922  
  
http://redis.io  
  
26922:M 07 May 2021 23:53:55.568 * WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/ipv4/tcp_max_syn_backlog is at its default value of 8.  
26922:M 07 May 2021 23:53:55.568 * Server initialized  
26922:M 07 May 2021 23:53:55.568 * WARNING overcommit_memory is set to 0! Background save may fail under low memory conditions  
26922:M 07 May 2021 23:53:55.568 * WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will negatively impact Redis performance. To fix this issue run the command 'echo madvise > /sys/kernel/mm/transparent_hugepage/enabled' or 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' to retain the setting after a reboot. Redis must be restarted after THP is disabled (set to 'madvise' or 'never')  
26922:M 07 May 2021 23:53:55.568 * Ready to accept connections
```

```
[root@iZ2ze8biiph4vi0x5z06heZ src]# ./redis-server ../6380.conf  
26931:C 07 May 2021 23:54:01.752 * o000o000o000o Redis is starting o000o000o000o  
26931:C 07 May 2021 23:54:01.752 * Redis version=6.0.9, bits=64, commit=00000000, modified=0, pid=26931  
26931:C 07 May 2021 23:54:01.752 * Configuration loaded  
  
Redis 6.0.9 (00000000/0) 64 bit  
Running in standalone mode  
Port: 6380  
PID: 26931  
  
http://redis.io  
  
26931:M 07 May 2021 23:54:01.754 * WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/ipv4/tcp_max_syn_backlog is at its default value of 8.  
26931:M 07 May 2021 23:54:01.754 * Server initialized  
26931:M 07 May 2021 23:54:01.754 * WARNING overcommit_memory is set to 0! Background save may fail under low memory conditions  
26931:M 07 May 2021 23:54:01.754 * WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will negatively impact Redis performance. To fix this issue run the command 'echo madvise > /sys/kernel/mm/transparent_hugepage/enabled' or 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' to retain the setting after a reboot. Redis must be restarted after THP is disabled (set to 'madvise' or 'never')  
26931:M 07 May 2021 23:54:01.754 * Ready to accept connections
```

```
[root@iZ2ze8biiph4vi0x5z06heZ src]# ./redis-server ../6381.conf
```

```
26939:C 07 May 2021 23:54:06.263 * 0000000000 Redis is starting 0000000000
26939:C 07 May 2021 23:54:06.263 * Redis version=6.0.9, bits=64, commit=00000000, modified=0, pi
26939:C 07 May 2021 23:54:06.263 * Configuration loaded

                               Redis 6.0.9 (00000000/0) 64 bit
                               Running in standalone mode
                               Port: 6381
                               PID: 26939

                               http://redis.io

26939:M 07 May 2021 23:54:06.264 * WARNING: The TCP backlog setting of 511 cannot be enforced be
8.
26939:M 07 May 2021 23:54:06.264 * Server initialized
26939:M 07 May 2021 23:54:06.264 * WARNING overcommit_memory is set to 0! Background save may fo
t_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memo
26939:M 07 May 2021 23:54:06.264 * WARNING you have Transparent Huge Pages (THP) support enabled
with Redis. To fix this issue run the command 'echo madvise > /sys/kernel/mm/transparent_hugepag
etain the setting after a reboot. Redis must be restarted after THP is disabled (set to 'madvise
26939:M 07 May 2021 23:54:06.264 * Ready to accept connections
```

## 2. 查看三个redis服务是否都启动了

Bash

```
1 ps -ef | grep redis  
2 netstat -nltp
```

```
[root@iZ2ze8biiph4vi0x5z06heZ src]# ps -ef | grep redis
root 25630 25595 0 23:30 pts/4 00:00:00 ./redis-cli -p 6380
root 26922 24428 0 23:53 pts/0 00:00:00 ./redis-server 127.0.0.1:6379
root 26931 25308 0 23:54 pts/1 00:00:00 ./redis-server 127.0.0.1:6380
root 26939 25332 0 23:54 pts/2 00:00:00 ./redis-server 127.0.0.1:6381
root 27027 25440 0 23:55 pts/3 00:00:00 grep --color=auto redis
[root@iZ2ze8biiph4vi0x5z06heZ src]#
```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	127.0.0.1:6379	0.0.0.0:*	LISTEN	26922./redis-serv
tcp	0	0	127.0.0.1:6380	0.0.0.0:*	LISTEN	26931./redis-serv
tcp	0	0	0.0.0.0:3308	0.0.0.0:*	LISTEN	6863/docker-droxi
tcp	0	0	127.0.0.1:6381	0.0.0.0:*	LISTEN	26939./redis-serv
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	3958/ssna
tcp	0	0	127.0.0.1:25	0.0.0.0:*	LISTEN	967/master
tcp6	0	0	:::3308	:::*	LISTEN	6867/docker-proxy
tcp6	0	0	:::1:25	:::*	LISTEN	967/master

### 3. 连接对应端口的服务

## Bash

```
1 ./redis-cli -p 6379  
2 ./redis-cli -p 6380  
3 ./redis-cli -p 6381
```

```
[root@iZ2ze8biiph4vi0x5z06heZ src]# ./redis-cli -p 6379
```

#### 4. 查看当前的主从关系

三个客户端都查看一下自己,可以发现,当前的角色都是master,也就是说,每一个redis服务,默认都是主机;

Bash

## 1 info replication

5. 在6379客户端上,设置一些数据。「一会把它作为主机,看看从机同步过程」

```
raven 540A 127.0.0.1:6379> set k1 tom  
OK  
raven 540A 127.0.0.1:6379> set k2 jack  
OK  
raven 540A 127.0.0.1:6379> set k3 jerry  
OK  
raven 540A 127.0.0.1:6379> set k4 pet  
OK  
raven 540A 127.0.0.1:6379>
```

6. 在6380上,设置跟随6379,即:把6380设置为从机;

```
raven504 master_replid:00000000000000000000000000000000  
raven504 master_repl_offset:0  
raven504 second_repl_offset:-1  
raven504 repl_backlog_active:0  
raven504 repl_backlog_size:1048576  
raven504 repl_backlog_first_byte_offset:0  
raven504 repl_backlog_histlen:0  
raven504 127.0.0.1:6380> REPLICAOF 127.0.0.1 6379  
raven504 OK  
raven504 127.0.0.1:6380>
```

## 7. 查看6379服务的日志输出

```
stein the setting after a reboot. Redis must be restarted after TIEP is disabled (set to madvise or never).
26922:M 07 May 2021 23:53:55.568 * Ready to accept connections
26922:M 08 May 2021 00:05:29.748 * Replica 127.0.0.1:6380 asks for synchronization 异步方式进行复制操作
26922:M 08 May 2021 00:05:29.748 * Partial resynchronization not accepted: Replication ID mismatch (Replica asked for '32359d5223d71c3ae3b00fc0d0df04cbda6b924c', my replication IDs are '37d0a46d9fce38b8647bb82588e956d3fc286c8' and '0000000000000000000000000000000000000000000000000000000000000000')
26922:M 08 May 2021 00:05:29.748 * Replication backlog created, my new replication IDs are '0289886fc043cd034cd034c8aba1b6db5b6d95676278c' and '0000000000000000000000000000000000000000000000000000000000000000'
26922:M 08 May 2021 00:05:29.748 * Starting BGSAVE for SYNC with target: disk →当6380设置为了6379为主机，则触发了主机的bgsave，即rdb操作。
26922:M 08 May 2021 00:05:29.748 * Background saving started by pid 27551 →fork子进程，从内存向硬盘写数据
27551:C 08 May 2021 00:05:29.750 * DB saved on disk
27551:C 08 May 2021 00:05:29.751 * RDB: 0 MB of memory used by copy-on-write
26922:M 08 May 2021 00:05:29.848 * Background saving terminated with success 写入成功
26922:M 08 May 2021 00:05:29.849 * Synchronization with replica 127.0.0.1:6380 succeeded →把6379的数据，通过异步的方式给6380，操作成功
```

## 8. 查看6380服务的日志输出

26931:IN 07 May 2021 23:54:01.754 \* Ready to accept connections  
26931:S 08 May 2021 00:05:29.475 \* Before turning into a slave, I can already apply my own master parameters to synthesize a cached master: I may be able to synchronize with the new master with just a partial transfer.  
**夏刷王机6379开始**  
26931:S 08 May 2021 00:05:29.475 \* REPLICAOF 127.0.0.1:6379 enabled (user request from 'id=4 addr=127.0.0.1:52362 fd=7 name= age=28 idle=0 flags=N db=0 sub=0 psub =0 multi-l=1 ubuf=44 qbuf-free=32724 arqv-mem=22 ovl=0 omem=0 totl-mem=49822 events=r cmd=replicaof user=default')  
26931:S 08 May 2021 00:05:29.747 \* Connecting to MASTER 127.0.0.1:6379  
26931:S 08 May 2021 00:05:29.748 \* MASTER <-> REPLICA sync started → 连接上了主机6379  
26931:S 08 May 2021 00:05:29.748 \* Non blocking connect for SYNC fired the event.  
26931:S 08 May 2021 00:05:29.748 \* Master replied to PING, replication can continue...  
26931:S 08 May 2021 00:05:29.748 \* Trying a partial resynchronization (request 32359d5223d71a3ae3b00fc0d8df04cbda6b924c:1).  
26931:S 08 May 2021 00:05:29.748 \* Full resync from master: 0289886fc043cd034c8aba1bb6d556d59576278c:0  
26931:S 08 May 2021 00:05:29.748 \* Discarding previously cached master state.  
26931:S 08 May 2021 00:05:29.849 \* MASTER <-> REPLICA sync: receiving 224 bytes from master to disk  
26931:S 08 May 2021 00:05:29.849 \* MASTER <-> REPLICA sync: Flushing old data → 从主机6379同步过来的数据信息，接收了224字节，从磁盘读取的  
26931:S 08 May 2021 00:05:29.849 \* MASTER <-> REPLICA sync: Loading DB in memory → 先把自己的数据清空一下  
26931:S 08 May 2021 00:05:29.850 \* Loading RDB produced by version 6.0.9  
26931:S 08 May 2021 00:05:29.850 \* RDB age 0 seconds  
26931:S 08 May 2021 00:05:29.850 \* RDB memory usage when created 1.91 Mb  
26931:S 08 May 2021 00:05:29.851 \* MASTER <-> REPLICA sync: Finished with success → 操作成功

## 9. 查看6380中,是否已经有数据了

```
raven5404 127.0.0.1:6380> keys *
raven5404 1) "k1" 和主机6379数据一样
raven5404 2) "k3"
raven5404 3) "k4"
raven5404 4) "k2"
raven5404 5) "k6"
raven5404 127.0.0.1:6380> get k1
raven5404 "tom"
raven5404 127.0.0.1:6380>
```

```
127.0.0.1:6379> keys *
```

```
3) "k2"  
4) "k4"  
5) "k3"  
127.0.0.1:6379> get k1  
"tom"  
127.0.0.1:6379>
```

## 10. 查看6379的主从状态

```
127.0.0.1:6379> info replication  
# Replication  
role:master → 角色还是主机  
connected_slaves:1 → 有一个小弟了  
slave0:ip=127.0.0.1,port=6380,state=online,offset=1162,lag=0  
master_replid:0289886fc043cd034c8aba1b6db5b6d95676278c  
master_replid2:0000000000000000000000000000000000000000000000000000000000  
master_repl_offset:1162  
second_repl_offset:-1  
rep1_backlog_active:1  
rep1_backlog_size:1048576  
rep1_backlog_first_byte_offset:1  
rep1_backlog_histlen:1162  
127.0.0.1:6379>
```

## 11. 查看6380的主从状态

```
127.0.0.1:6380> info replication  
# Replication  
role:slave → 角色变为了从机  
master_host:127.0.0.1 → 大哥host地址  
master_port:6379  
master_link_status:up  
master_last_io_seconds_ago:8  
master_sync_in_progress:0  
slave_repl_offset:1246  
slave_priority:100  
slave_read_only:1  
connected_slaves:0  
master_replid:0289886fc043cd034c8aba1b6db5b6d95676278c  
master_replid2:0000000000000000000000000000000000000000000000000000000000  
master_repl_offset:1246  
second_repl_offset:-1  
rep1_backlog_active:1  
rep1_backlog_size:1048576  
rep1_backlog_first_byte_offset:1  
rep1_backlog_histlen:1246  
127.0.0.1:6380>
```

## 12. 查看落盘的rdb文件

```
-rw-r--r-- 1 root root 49 Mar 11 17:29 2021-03-11.txt  
-rw-r--r-- 1 root root 224 May 8 00:05 6379.rdb  
-rw-r--r-- 1 root root 224 May 8 00:05 6380.rdb
```

## 13. 主机6379,写入一个数据,看看6380是否同步了

```
127.0.0.1:6379> set k8 hello  
OK
```

127.0.0.1:6379>

127.0.0.1:6380> get k8

"hello"

127.0.0.1:6380>

- 6379服务日志

```
* RDB: 0 MB of memory used by copy-on-write  
* Background saving terminated with success
```

- 6380服务日志

```
28470:C 08 May 2021 00:22:54.710 * RDB: 0 MB of memory used by copy-on-write  
26931:S 08 May 2021 00:22:54.807 * Background saving terminated with success
```

#### 14. 把6381也设置为从机

- 6379日志

```
# Replication  
role:master  
connected_slaves:2  
slave0:ip=127.0.0.1,port=6380,state=online,offset=1903,lag=1  
slave1:ip=127.0.0.1,port=6381,state=online,offset=1903,lag=0  
master_replid:0289886fc043cd034c8aba1b6db5b6d95676278c  
master_replid2:000000000000000000000000000000000000000000000000000000000000000  
master_repl_offset:1903  
second_repl_offset:-1  
rep1_backlog_active:1  
rep1_backlog_size:1048576  
rep1_backlog_first_byte_offset:1  
rep1_backlog_histlen:1903  
127.0.0.1:6379>
```

#### 15. 关闭6380服务,再用如下命令进行启动.看看数据恢复情况

Bash

```
1 ./redis-server ../6380.conf replicaof 127.0.0.1 6379# 直接指定主机
```

```
26922:M 08 May 2021 00:27:08.159 * Synchronization with replica 127.0.0.1:6381 succeeded  
26922:M 08 May 2021 00:34:30.540 * Connection with replica 127.0.0.1:6380 lost
```

6379日志

```
127.0.0.1:6379> info replication  
# Replication  
role:master  
connected_slaves:1
```

```
slave0:ip=127.0.0.1,port=6381,state=online,offset=2561,lag=1
master_replid:0289886fc043cd034c8aba1b6db5b6d95676278c
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:2561
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:2561
127.0.0.1:6379>
```

## 6379主从信息

### ● 再次连接

```
29188:S 08 May 2021 00:36:26.078 * WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value.
29188:S 08 May 2021 00:36:26.078 * Server initialized
29188:S 08 May 2021 00:36:26.078 * WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
29188:S 08 May 2021 00:36:26.078 * WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo madvise > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled (set to 'madvise' or 'never').
29188:S 08 May 2021 00:36:26.078 * Loading RDB produced by version 6.0.9
29188:S 08 May 2021 00:36:26.078 * RDB age 116 seconds
29188:S 08 May 2021 00:36:26.078 * RDB memory usage when created 1.87 Mb
29188:S 08 May 2021 00:36:26.078 * DB loaded from disk: 0.000 seconds
29188:S 08 May 2021 00:36:26.078 * Before turning into a replica, using my own master parameters to synthesize a cached master: I may be able to synchronize the new master with just a partial transfer.
29188:S 08 May 2021 00:36:26.078 * Ready to accept connections
29188:S 08 May 2021 00:36:26.078 * Connecting to MASTER 127.0.0.1:6379
29188:S 08 May 2021 00:36:26.078 * MASTER <-> REPLICA sync started
29188:S 08 May 2021 00:36:26.078 * Non blocking connect for SYNC fired the event.
29188:S 08 May 2021 00:36:26.078 * Master replied to PING, replication can continue...
29188:S 08 May 2021 00:36:26.079 * Trying a partial resynchronization (request 0289886fc043cd034c8aba1b6db5b6d95676278c:2478).
29188:S 08 May 2021 00:36:26.079 * Successful partial resynchronization with master.
29188:S 08 May 2021 00:36:26.079 * MASTER <-> REPLICA sync: Master accepted a Partial Resynchronization.
```

## 6380日志

```
26922:M 08 May 2021 00:36:26.079 * Replica 127.0.0.1:6380 asks for synchronization
26922:M 08 May 2021 00:36:26.079 * Partial resynchronization request from 127.0.0.1:6380 accepted. Sending 168 bytes of backlog starting from offset 2478.
```

## 6379日志

### ● 查看6380数据

数据恢复了。

```
127.0.0.1:6380> keys *
1) "k6"
2) "k4"
3) "k3"
4) "k8"
5) "k2"
6) "k1"
127.0.0.1:6380> get k8
"hello"
127.0.0.1:6380>
```

16. 再次断开6380服务. 使用6379服务, 添加一些数据.再启动6380, 看看恢复的时候,是否会追加6379添加到的数据到6380中;

```
127.0.0.1:6379> keys *
1) "k6"
2) "k3"
3) "k1"
4) "k2"
5) "k4"
6) "k8"
127.0.0.1:6379> set name aabb
OK
127.0.0.1:6379> keys *
1) "name"
```

6379新添加的数据, 此时6380断开了

```
2) "k6"
3) "k3"
4) "k1"
5) "k2"
6) "k4"
7) "k8"
127.0.0.1:6379>
```

连接: ./redis-server ../6380.conf replicaof 127.0.0.1 6379# 直接指定主机

```
127.0.0.1:6380> keys *
1) "k2"
2) "k4"
3) "k3"
4) "k1"
5) "k8"
6) "k6"
7) "name"
127.0.0.1:6380> get name
"aaabb"
127.0.0.1:6380>
```

**6380已经有数据了, 会追加数据**

**6379日志**

## 17. 再次断开6380, 开启aof, 查看数据同步情况.查看日志内容

Bash

```
1 ./redis-server ../6380.conf --replicaof 127.0.0.1 6379 --appendonly yes
```

**rdb文件中保存了主从关系,也就是说,从机知道自己跟随的是哪台主机,但是如果是aof的方式,**

**尽管使用的是rdb的数据,但是再把rdb文件写入到aof文件时候,这个关系会被忽略掉; ??????**

## 18. 断开6379, 查看两个从机情况. 重新启动6379, 看看从机是否会自动连接;

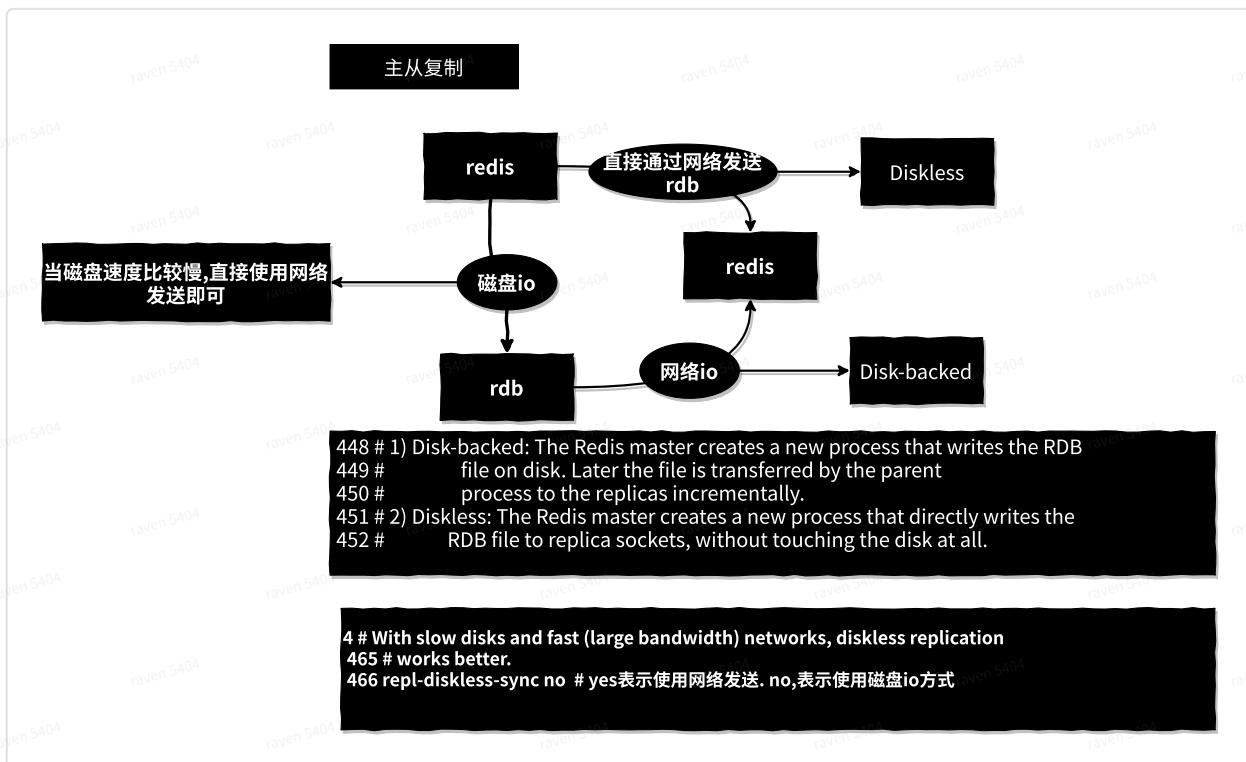
## 19. 如果想把6380由从机变成主机, 让6381去跟随6380.可以执行如下操作; 「假设此时6379已经挂掉了」

## Bash

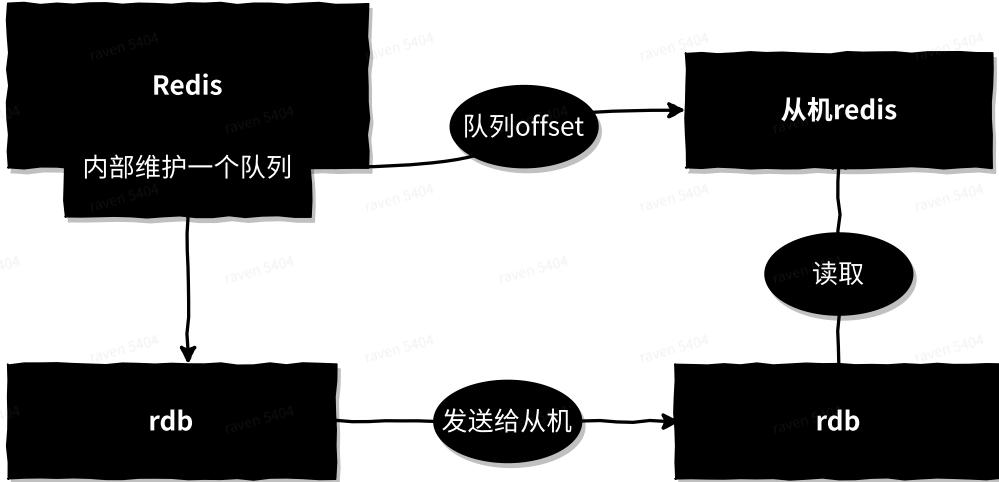
```
1 //把当前的从机改为主机;
2 replicaof no one # 6380执行
3
4 # 跟随主机;
5 replicaof 127.0.0.1 6380 # 6381执行
6
7 # info replication, 查看当前节点的信息, 如角色, 主机/从机, 如果是主机有几个从机, 各个
ip地址;
8
9 # 启动服务的时候, 指定主机;
10 ./redis-service ../xxx.conf --replicaof 主机的ip 主机的端口
11 # 启动服务的时候, 指定跟随主机并且开启aof支持;
12 ./redis-service ../xxx.conf --replicaof 主机的ip 主机的端口 --appendfile yes
```

- 数据是如何同步的

- 主从同步的时候.



- 增量更新如何实现



```

# Set the replication backlog size. The backlog is a buffer that accumulates
541 # replica data when replicas are disconnected for some time, so that when a
542 # replica wants to reconnect again, often a full resync is not needed, but a
543 # partial resync is enough, just passing the portion of data the replica
544 # missed while disconnected.
545 #
546 # The bigger the replication backlog, the longer the replica can endure the
547 # disconnect and later be able to perform a partial resynchronization.
548 #
549 # The backlog is only allocated if there is at least one replica connected.
550 #
551 # repl-backlog-size 1mb # 设置redis内部队列大小, 根据业务自己修改;

```

- 重点

### Plain Text

- 1 1. 主从复制  
主机负责写，从机读
- 2 2. 数据同步机制  
主机数据如何同步给从机
- 3 3. 增量同步  
从机断开，再次连接主机的时候，数据如何同步的；
- 4 4. 跟随主机
- 5 5. aof文件没有保留主机id

## 1. 配置文件「自动」

### Bash

- 1 # 复制相关配置参数
- 2 ##### REPLICATION
- 3 #####
- 4 # Master-Replica replication. Use replicaof to make a Redis instance a copy of

5 # another Redis server. A few things to understand ASAP about Redis  
replication.

6 371 #

7 372 # +-----+ +-----+  
8 373 # | Master | ---> | Replica |  
9 374 # | (receive writes) | | (exact copy) |  
10 375 # +-----+ +-----+

11 376 #

12 377 # 1) Redis replication is asynchronous, but you can configure a master to

13 378 # stop accepting writes if it appears to be not connected with at  
least

14 379 # a given number of replicas.

15 380 # 2) Redis replicas are able to perform a partial resynchronization with  
the

16 381 # master if the replication link is lost for a relatively small amount  
of

17 382 # time. You may want to configure the replication backlog size (see  
the next

18 383 # sections of this file) with a sensible value depending on your  
needs.

19 384 # 3) Replication is automatic and does not need user intervention. After  
a

20 385 # network partition replicas automatically try to reconnect to masters

21 386 # and resynchronize with them.

22 387 #

23 388 # replicaof <masterip> <masterport>

24 389

25 390 # If the master is password protected (using the "requirepass"  
configuration

26 391 # directive below) it is possible to tell the replica to authenticate  
before

27 392 # starting the replication synchronization process, otherwise the master  
will

28 393 # refuse the replica request.

29 394 #

30 395 # masterauth <master-password>

31 396 #

32 397 # However this is not enough if you are using Redis ACLs (for Redis  
version

33 398 # 6 or greater), and the default user is not capable of running the PSYNC

34 399 # command and/or other commands needed for replication. In this case it's

35 400 # better to configure a special user to use with replication, and specify

the

401 # masteruser configuration as such:

402 #

403 # masteruser <username>

404 #

405 # When masteruser is specified, the replica will authenticate against its

406 # master using the new AUTH form: AUTH <username> <password>.

407

408 # When a replica loses its connection with the master, or when the

replication

409 # is still in progress, the replica can act in two different ways:

410 #

411 # 1) if replica-serve-stale-data is set to 'yes' (the default) the

replica will

412 # still reply to client requests, possibly with out of date data, or

the

413 # data set may just be empty if this is the first synchronization.

414 #

415 # 2) If replica-serve-stale-data is set to 'no' the replica will reply

with

416 # an error "SYNC with master in progress" to all commands except:

417 # INFO, REPLICAOF, AUTH, PING, SHUTDOWN, REPLCONF, ROLE, CONFIG,

SUBSCRIBE,

418 # UNSUBSCRIBE, PSUBSCRIBE, PUNSUBSCRIBE, PUBLISH, PUBSUB, COMMAND,

POST,

419 # HOST and LATENCY.

420 #

421 replica-serve-stale-data yes

422

423 # You can configure a replica instance to accept writes or not. Writing

against

424 # a replica instance may be useful to store some ephemeral data (because

data

425 # written on a replica will be easily deleted after resync with the

master) but

426 # may also cause problems if clients are writing to it because of a

427 # misconfiguration.

428 #

429 # Since Redis 2.6 by default replicas are read-only.

430 #

431 # Note: read only replicas are not designed to be exposed to untrusted

clients

432 # on the internet. It's just a protection layer against misuse of the

instance.

433 # Still a read only replica exports by default all the administrative

commands

434 # such as CONFIG, DEBUG, and so forth. To a limited extent you can

improve

70 435 # security of read only replicas using 'rename-command' to shadow all the  
raven 5404  
71 436 # administrative / dangerous commands.  
raven 5404  
72 437 replica-read-only yes # 是否是只读模式,如果是yes,表示开启,不提供写服务;  
raven 5404  
73 438  
raven 5404  
74 439 # Replication SYNC strategy: disk or socket. # 同步方式选择  
raven 5404  
75 440 #  
raven 5404  
76 441 # New replicas and reconnecting replicas that are not able to continue  
the  
raven 5404  
77 442 # replication process just receiving differences, need to do what is  
called a  
raven 5404  
78 443 # "full synchronization". An RDB file is transmitted from the master to  
the  
raven 5404  
79 444 # replicas.  
raven 5404  
80 445 #  
raven 5404  
81 446 # The transmission can happen in two different ways:  
raven 5404  
82 447 #  
raven 5404  
83 448 # 1) Disk-backed: The Redis master creates a new process that writes the  
RDB  
raven 5404  
84 449 # file on disk. Later the file is transferred by the  
parent  
raven 5404  
85 450 # process to the replicas incrementally.  
raven 5404  
86 451 # 2) Diskless: The Redis master creates a new process that directly  
writes the  
raven 5404  
87 452 # RDB file to replica sockets, without touching the disk at  
all.  
raven 5404  
88 453 #  
raven 5404  
89 454 # With disk-backed replication, while the RDB file is generated, more  
replicas  
raven 5404  
90 455 # can be queued and served with the RDB file as soon as the current child  
raven 5404  
91 456 # producing the RDB file finishes its work. With diskless replication  
instead  
raven 5404  
92 457 # once the transfer starts, new replicas arriving will be queued and a  
new  
raven 5404  
93 458 # transfer will start when the current one terminates.  
raven 5404  
94 459 #  
raven 5404  
95 460 # When diskless replication is used, the master waits a configurable  
amount of  
raven 5404  
96 461 # time (in seconds) before starting the transfer in the hope that  
multiple  
raven 5404  
97 462 # replicas will arrive and the transfer can be parallelized.  
raven 5404  
98 463 #  
raven 5404  
99 464 # With slow disks and fast (large bandwidth) networks, diskless  
replication  
raven 5404  
100 465 # works better.  
raven 5404  
101 466 rep1-diskless-sync no

```
101    466 # REPL_DISKLESS_SYNC NO
102    467
103    468 # When diskless replication is enabled, it is possible to configure the
104      delay
105    469 # the server waits in order to spawn the child that transfers the RDB via
106      socket
107    470 # to the replicas.
108    471 #
109    472 # This is important since once the transfer starts, it is not possible to
110      serve
111    473 # new replicas arriving, that will be queued for the next RDB transfer,
112      so the
113    474 # server waits a delay in order to let more replicas arrive.
114    475 #
115    476 # The delay is specified in seconds, and by default is 5 seconds. To
116      disable
117    477 # it entirely just set it to 0 seconds and the transfer will start ASAP.
118    478 repl-diskless-sync-delay 5
119    479
120    480 # -----
121    481 # -----#
122    482 # WARNING: RDB diskless load is experimental. Since in this setup the
123      replica
124    483 # does not immediately store an RDB on disk, it may cause data loss
125      during
126    484 # failovers. RDB diskless load + Redis modules not handling I/O reads may
127      also
128    485 # cause Redis to abort in case of I/O errors during the initial
129      synchronization
130    486 # -----
131    487 # -----
132    488 # Replica can load the RDB it reads from the replication link directly
133      from the
134    489 # socket, or store the RDB to a file and read that file after it was
135      completely
136    490 # received from the master.
137    491 #
138    492 # In many cases the disk is slower than the network, and storing and
139      loading
140    493 # the RDB file may increase replication time (and even increase the
141      master's
142    494 # Copy on Write memory and save buffers).
143    495 # However, parsing the RDB file directly from the socket may mean that we
144      have
145    496 # to flush the contents of the current database before the full .rdb was
146      received. For this reason we have the following options:
```

```
raven 133 498 # raven 5404
raven 134 499 # "disabled" - Don't use diskless load (store the rdb file to the disk
raven first)
raven 135 500 # "on-empty-db" - Use diskless load only when it is completely safe.
raven 136 501 # "swapdb" - Keep a copy of the current db contents in RAM while
raven parsing
raven 137 502 # the data directly from the socket. note that this
raven requires
raven 138 503 # sufficient memory, if you don't have it, you risk an
raven OOM kill.
raven 139 504 repl-diskless-load disabled
raven 140 505
raven 141 506 # Replicas send PINGs to server in a predefined interval. It's possible
raven to
raven 142 507 # change this interval with the repl_ping_replica_period option. The
raven default
raven 143 508 # value is 10 seconds.
raven 144 509 #
raven 145 510 # repl-ping-replica-period 10
raven 146 511
raven 147 512 # The following option sets the replication timeout for:
raven 148 513 #
raven 149 514 # 1) Bulk transfer I/O during SYNC, from the point of view of replica.
raven 150 515 # 2) Master timeout from the point of view of replicas (data, pings).
raven 151 516 # 3) Replica timeout from the point of view of masters (REPLCONF ACK
pings).
raven 152 517 #
raven 153 518 # It is important to make sure that this value is greater than the value
raven 154 519 # specified for repl-ping-replica-period otherwise a timeout will be
detected
raven 155 520 # every time there is low traffic between the master and the replica. The
raven default
raven 156 521 # value is 60 seconds.
raven 157 522 #
raven 158 523 # repl-timeout 60
raven 159 524
raven 160 525 # Disable TCP_NODELAY on the replica socket after SYNC?
raven 161 526 #
raven 162 527 # If you select "yes" Redis will use a smaller number of TCP packets and
raven 163 528 # less bandwidth to send data to replicas. But this can add a delay for
raven 164 529 # the data to appear on the replica side, up to 40 milliseconds with
raven 165 530 # Linux kernels using a default configuration.
raven 166 531 #
raven 167 532 # If you select "no" the delay for data to appear on the replica side
will
raven 168 533 # be reduced but more bandwidth will be used for replication.
raven 169 534 #
raven 170 535 # By default we optimize for low latency, but in very high traffic
```

conditions

171 536 # or when the master and replicas are many hops away, turning this to  
"yes" may

172 537 # be a good idea.

173 538 repl-disable-tcp-nodelay no

174 539

175 540 # Set the replication backlog size. The backlog is a buffer that  
accumulates

176 541 # replica data when replicas are disconnected for some time, so that when  
a

177 542 # replica wants to reconnect again, often a full resync is not needed,  
but a

178 543 # partial resync is enough, just passing the portion of data the replica  
179 544 # missed while disconnected.

180 545 #

181 546 # The bigger the replication backlog, the longer the replica can endure  
the

182 547 # disconnect and later be able to perform a partial resynchronization.

183 548 #

184 549 # The backlog is only allocated if there is at least one replica  
connected.

185 550 #

186 551 # repl-backlog-size 1mb

187 552

188 553 # After a master has no connected replicas for some time, the backlog  
will be

189 554 # freed. The following option configures the amount of seconds that need  
to

190 555 # elapse, starting from the time the last replica disconnected, for the  
backlog

191 556 # buffer to be freed.

192 557 #

193 558 # Note that replicas never free the backlog for timeout, since they may  
be

194 559 # promoted to masters later, and should be able to correctly "partially  
195 560 # resynchronize" with other replicas: hence they should always accumulate  
backlog.

196 561 #

197 562 # A value of 0 means to never release the backlog.

198 563 #

199 564 # repl-backlog-ttl 3600

200 565

201 566 # The replica priority is an integer number published by Redis in the  
INFO

202 567 # output. It is used by Redis Sentinel in order to select a replica to  
promote

203 568 # into a master if the master is no longer working correctly.

204 569 #

```
204      509 #
205      570 # A replica with a low priority number is considered better for
206      571 # promotion, so
207      572 # for instance if there are three replicas with priority 10, 100, 25
208      573 #
209      574 # However a special priority of 0 marks the replica as not able to
210      575 # perform the
211      576 # role of master, so a replica with priority of 0 will never be selected
212      577 #
213      578 # By default the priority is 100.
214      579 replica-priority 100
215      580
216      581 # It is possible for a master to stop accepting writes if there are less
217      582 # than
218      583 #
219      584 # The N replicas need to be in "online" state.
220      585 #
221      586 # The lag in seconds, that must be <= the specified value, is calculated
222      587 # from
223      588 #
224      589 # This option does not GUARANTEE that N replicas will accept the write,
225      590 # but
226      591 # will limit the window of exposure for lost writes in case not enough
227      592 #
228      593 # replicas
229      594 #
230      595 # min-replicas-to-write 3
231      596 # min-replicas-max-lag 10
232      597 #
233      598 # Setting one or the other to 0 disables the feature.
234      599 #
235      600 # By default min-replicas-to-write is set to 0 (feature disabled) and
236      601 # min-replicas-max-lag is set to 10.
237      602
238      603 # A Redis master is able to list the address and port of the attached
239      604 # replicas in different ways. For example the "INFO replication" section
240      605 # offers this information, which is used, among other tools, by
241      606 # Redis Sentinel in order to discover replica instances.
```

242 607 # Another place where this info is available is in the output of the  
243 608 # "ROLE" command of a master.  
244 609 #  
245 610 # The listed IP address and port normally reported by a replica is  
246 611 # obtained in the following way:  
247 612 #  
248 574 # However a special priority of 0 marks the replica as not able to  
249 575 # perform the  
250 576 # role of master, so a replica with priority of 0 will never be selected  
251 577 #  
252 578 # By default the priority is 100.  
253 579 replica-priority 100  
254 580  
255 581 # It is possible for a master to stop accepting writes if there are less  
256 582 # N replicas connected, having a lag less or equal than M seconds.  
257 583 #  
258 584 # The N replicas need to be in "online" state.  
259 585 #  
260 586 # The lag in seconds, that must be <= the specified value, is calculated  
261 587 # from  
262 588 #  
263 589 # This option does not GUARANTEE that N replicas will accept the write,  
264 590 # but  
265 591 # will limit the window of exposure for lost writes in case not enough  
266 592 #  
267 593 # replicas  
268 594 #  
269 595 # min-replicas-to-write 3  
270 596 # min-replicas-max-lag 10  
271 597 #  
272 598 # Setting one or the other to 0 disables the feature.  
273 599 #  
274 600 # By default min-replicas-to-write is set to 0 (feature disabled) and  
275 601 # min-replicas-max-lag is set to 10.  
276 602  
277 603 # A Redis master is able to list the address and port of the attached  
278 604 # replicas in different ways. For example the "INFO replication" section  
279 605 # offers this information, which is used, among other tools, by  
280 606 # Redis Sentinel in order to discover replica instances.  
281 607 # Another place where this info is available is in the output of the

```
raven 5404 282 608 # "ROLE" command of a master.  
raven 5404 283 609 #  
raven 5404 284 610 # The listed IP address and port normally reported by a replica is  
raven 5404 285 611 # obtained in the following way:  
raven 5404 286 612 #  
raven 5404 287 613 # IP: The address is auto detected by checking the peer address  
raven 5404 288 614 # of the socket used by the replica to connect with the master.  
raven 5404 289 615 #  
raven 5404 290 616 # Port: The port is communicated by the replica during the replication  
raven 5404 291 617 # handshake, and is normally the port that the replica is using to  
raven 5404 292 618 # listen for connections.  
raven 5404 293 619 #  
raven 5404 294 620 # However when port forwarding or Network Address Translation (NAT) is  
raven 5404 295 621 # used, the replica may actually be reachable via different IP and port  
raven 5404 296 622 # pairs. The following two options can be used by a replica in order to  
raven 5404 297 623 # report to its master a specific set of IP and port, so that both INFO  
raven 5404 298 624 # and ROLE will report those values.  
raven 5404 299 625 #  
raven 5404 300 626 # There is no need to use both the options if you need to override just  
raven 5404 301 627 # the port or the IP address.  
raven 5404 302 628 #  
raven 5404 303 629 # replica-announce-ip 5.5.5.5  
raven 5404 304 630 # replica-announce-port 1234
```

- 如果配置好了主从关系, 使用如下命令进行查看

### Bash

```
1 info replication  
2 # 作业:  
3 # 起三个redis服务  
4 #     配置文件三份:  
5 #         端口号  
6 #         pid 「pidfile」  
7 #         日志文件名称 「logfile」  
8 #         rdb, aop文件名称 「改」
```

## 2. 总结

- 必要的配置



1. **replica-serve-stale-data yes**

2. **replica-read-only yes**

3. **repl-diskless-sync no**

4. **repl-backlog-size 1mb** # 表示队列大小

| 有从机断开的情况, 再连接. 之后的同步操作策略;

a. 增量复制

b. 如果写的速度比较快, 超出了队列的长度, 那么会丢失数据, 此时会触发一个全量操作;

c. 如果写的速度比较慢, 那么会增加更新;

5. **min-replicas-to-write 3**

6. **min-replicas-max-lag 10**

- 主从同步问题

| 需要人工去管理.

## 第九章: 哨兵

### REDIS sentinel-old -- Redis中国用户组(CRUG)

redis

<http://redis.cn/topics/sentinel.html>

- 参照文档

◦ 监控 (Monitoring) : Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。

◦ 提醒 (Notification) : 当被监控的某个 Redis 服务器出现问题时, Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

◦ 自动故障迁移 (Automatic failover) : 当一个主服务器不能正常工作时, Sentinel 会开始一次自动故障迁移操作, 它会将失效主服务器的其中一个从服务器升级为新的主服务器, 并让失效主服务器的其他从服务器改为复制新的主服务器; 当客户端试图连接失效的主服务器时, 集群也会向客户端返回新主服务器的地址, 使得集群可以使用新主服务器代替失效服务器。

- 配置一下

## Bash

```
1 # 新建配置文件.36379.conf
2 port 36379 # 端口号
3 sentinel monitor mymaster 127.0.0.1 6379 2
4 # sentinel
5 # mymaster , 逻辑名称,一组哨兵可以监听多套服务;
6 # 127.0.0.1, 主机的ip地址
7 # 6379, 主机的端口号
8 # 2, 投票权重
9 注: 如果主从服务器设置密码, 需主从服务器密码保持一致, 否则哨兵机制会失败!
10 sentinel auth-pass mymaster 12345678 #主密码, 不设置的话不能动态切换
11
12 port 36379
13 sentinel monitor mymaster 127.0.0.1 6379 2
14 sentinel auth-pass mymaster Jack12348765..
```

- 复制,两份,更改端口号;
- 使用3台哨兵监控;
- 操作步骤

- 先启动redis服务, 3个. 端口号:6379/6380/6381
- 启动的时候让6380/6381直接跟随6379或者直接使用配置文件配置一下跟随的主机即可;

## Bash

```
1 ./redis-sevrer ../6380.conf --replicaof 127.0.0.1 6379 # 3380
2 ./redis-sevrer ../6381.conf --replicaof 127.0.0.1 6379 # 3381
```

- 启动监控程序「3个」

## Bash

```
1 # 哨兵也是一个独立的进程;可以单独启动,也有自己的配置文件;
2 # 也可以使用redis-server进行启动. 因为redis-sentinel软连接指向的就是redis-server;
3
4 redis-server ./36379.conf --sentinel
5 redis-server ./36380.conf --sentinel
6 redis-server ./36381.conf --sentinel
7
8 # 特别注意:
9 # 特别注意:
10 # 特别注意:
11 # 特别注意:
12 # 特别注意:
13 1. 当所有服务全部停掉了,再次重新启动Redis服务和redis哨兵的时候,要把如下配置文件当中的多
   作的数据删除掉;
14     1. 36379/36380/36381文件当中, 把`# Generated by CONFIG REWRITE`这行之下的所有
      数据删除掉;包括这一行;
15     2. 6379/6380/6381配置文件最下边的两行,删除掉;
16     # 2055 user default on #9fb47c64dfd19793847381be41c9297f924e75fb6fdb6181a2
   ebe92a42cab337 ~* &* +@all
17 # 2056 replicaof 127.0.0.1 6380 , 以上两行删除掉;
```

## 1. 主从复制/HA(哨兵, 对主做高可用)解决的问题和未解决的问题

- 生产环境当中用的方案
- 目前只解决了单机故障问题;也就是akf当中的水平扩展问题;
- 回忆一下, 单机节点的问题;



- 单点故障
  - 通过主从复制和高可用可以解决掉了;
- 容量有限
  - 未解决,实际上主还全量的数据.终归是有上限的;
- 并发压力
  - 未解决,并发访问的时候还是会遇到瓶颈;

## 2. 解决容量问题

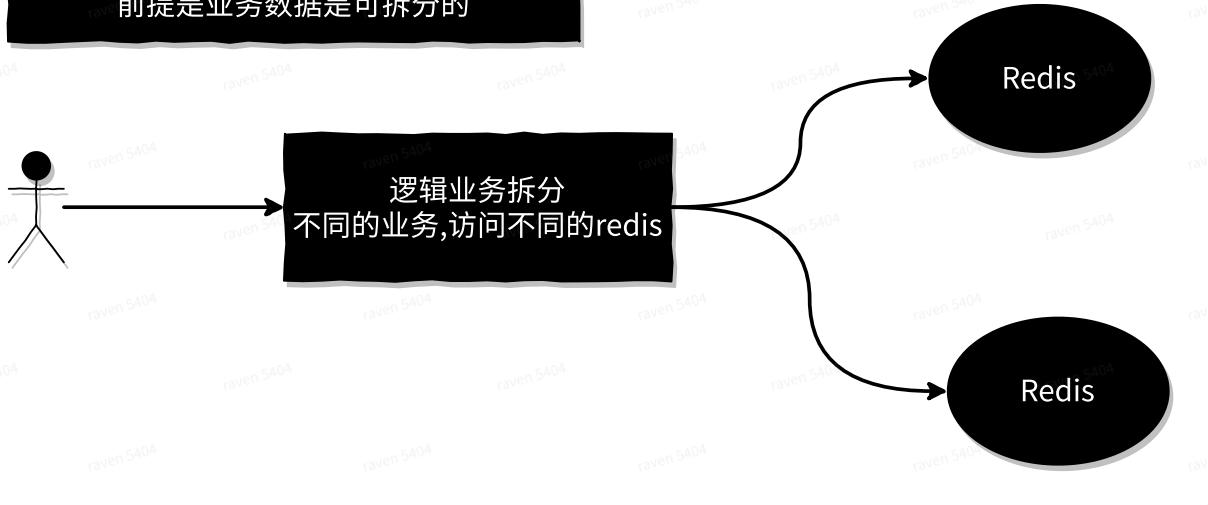
控制redis的内存使用量;「分治思想」

- 客户端解决方案

- 按业务进行拆分服务,不同的业务访问不同的Redis;

- 针对于可拆分的业务

## 1. 客户端逻辑控制「按业务进行拆分」 前提是业务数据是可拆分的



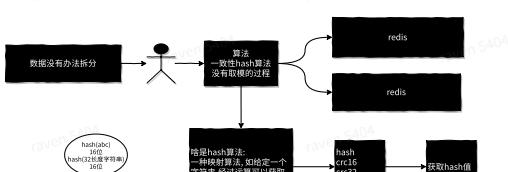
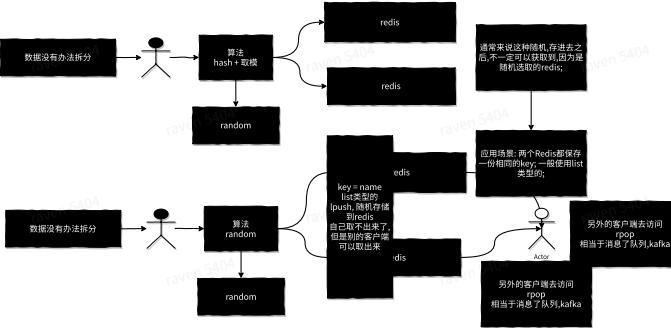
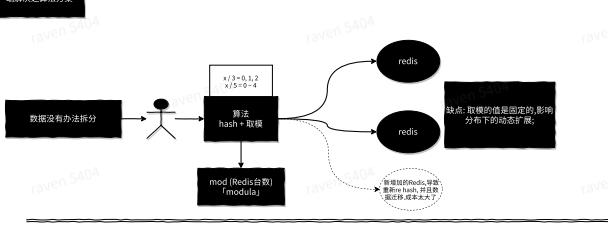
- 按算法拆分「3种方式实现」

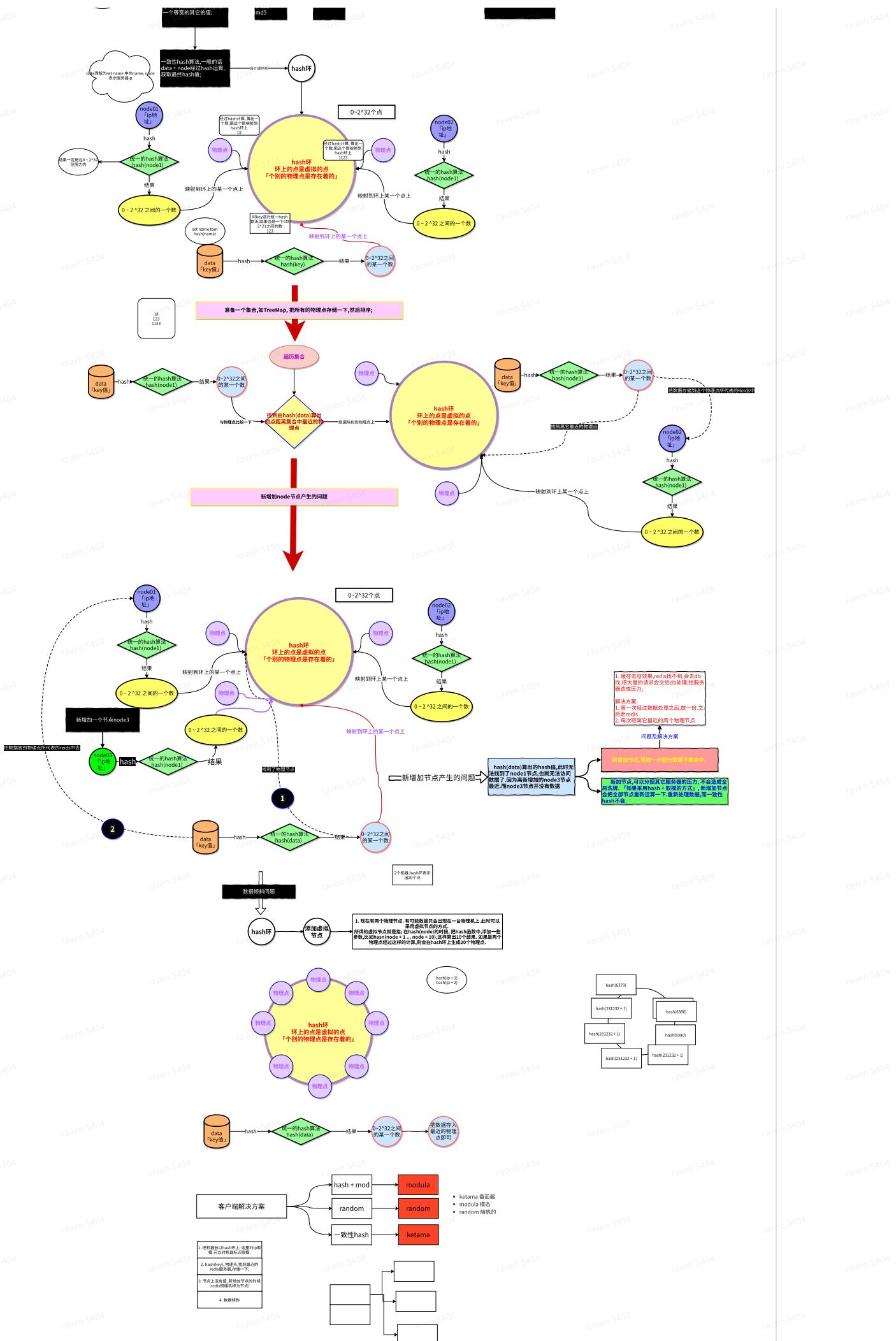
- 此时数据无法拆分的时候,可以使用此种方式;



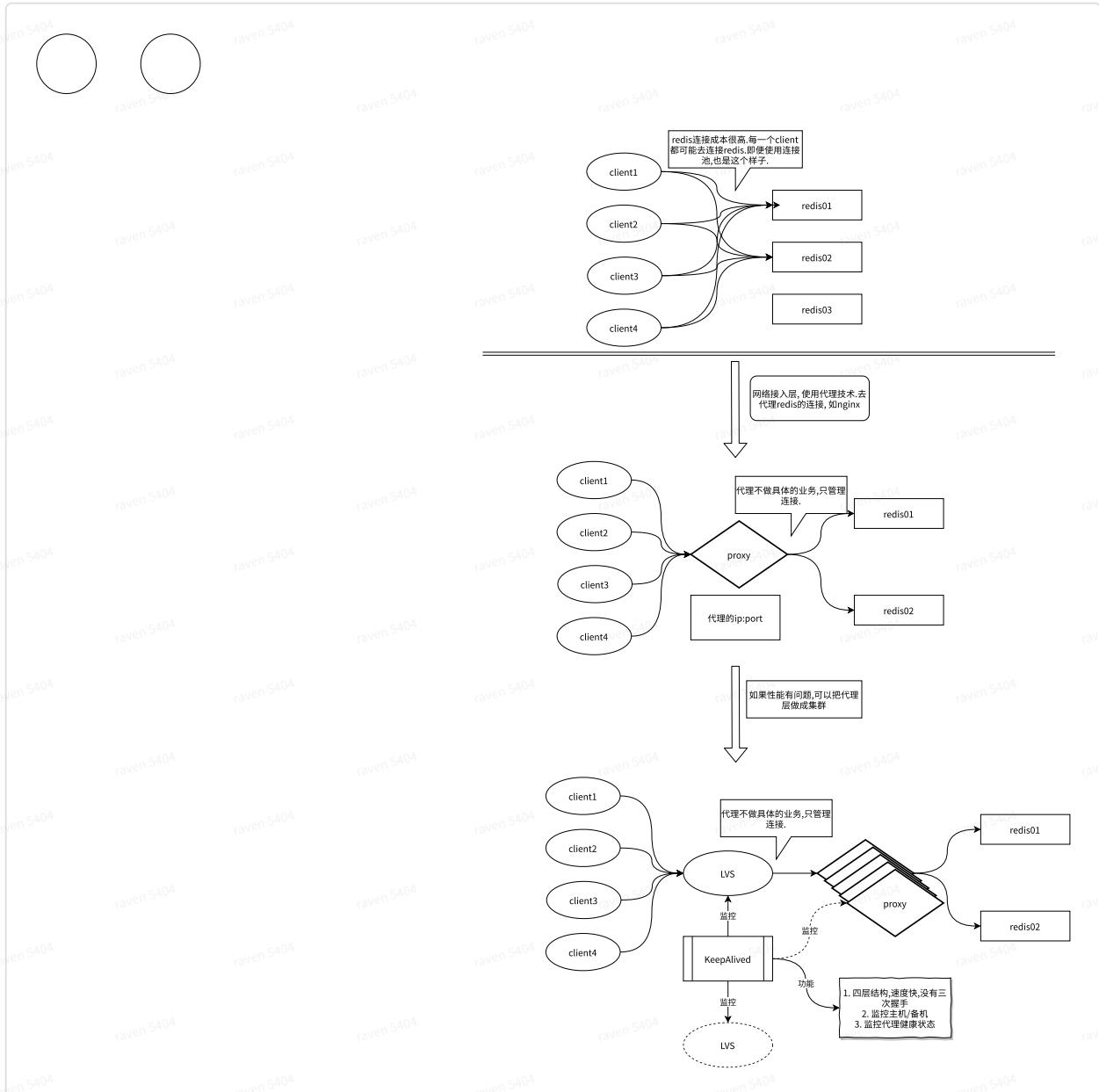
**分片 (Sharding)** 将一个数据分成两个或多个较小的块, 称为逻辑分片 (logical shards)。然后, 逻辑分片 (logical shards) 分布在单独的数据库节点上, 称为物理分片 (physical shards)。物理分片 (physical shards) 可以容纳多个逻辑分片 (logical shards)。尽管如此, 所有分片中保存的数据, 共同代表整个逻辑数据集。

### 2. 客户端解决之算法方案





- 客户端解决方案产生的问题



## ◦ LVS

LVS项目中的有关中文文档

LVS的有关中文文档



<http://www.linuxvirtualserver.org/zh/>

- keepalived

github.com

<https://github.com/acassen/keepalived>

### • 服务器端解决策略「重点内容」

- 通过代理层来解决.

### ◦ 三种逻辑算法实现方式, 可以简单理解为这种逻辑算法是「无状态」

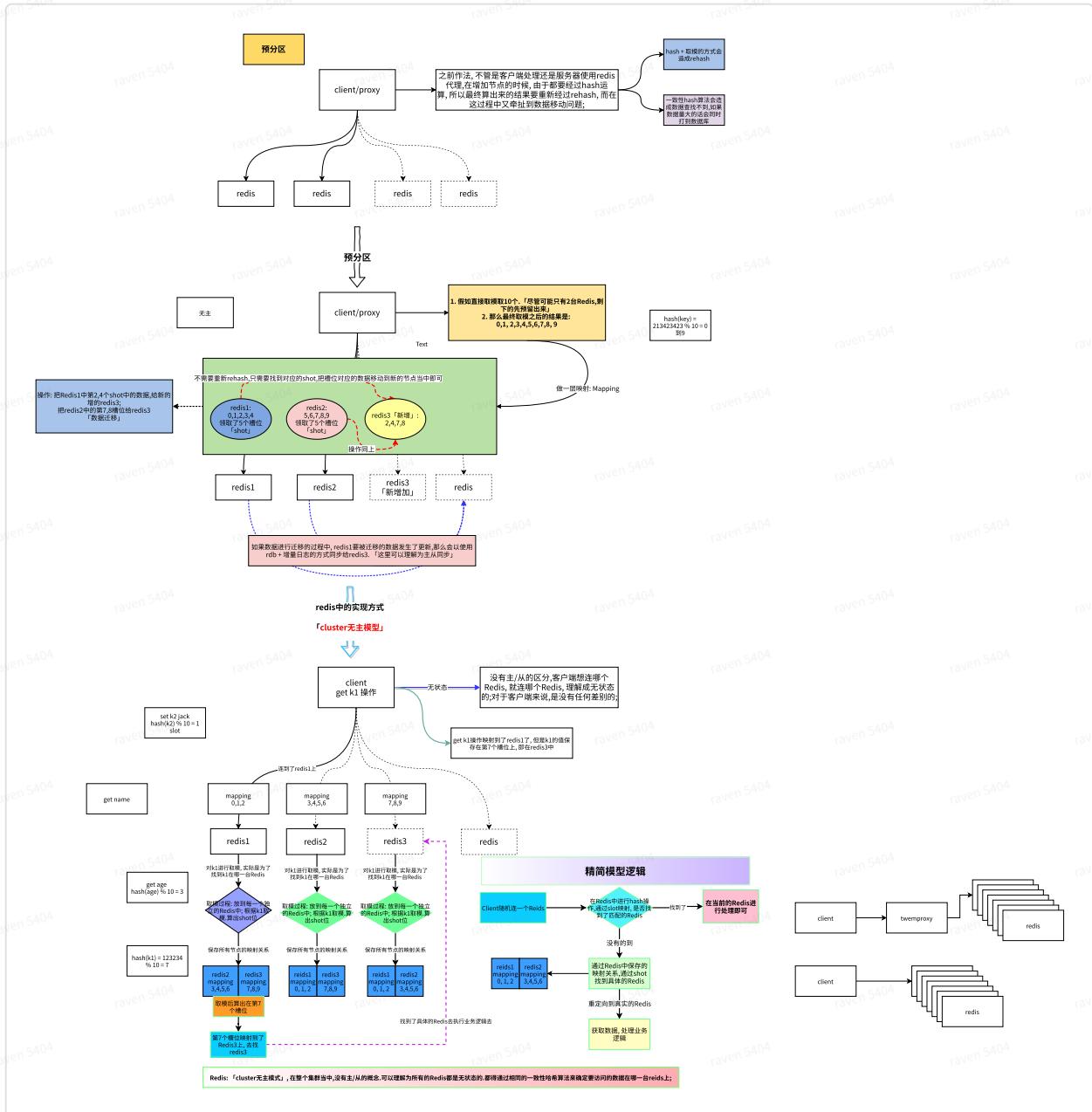
- modula
  - 哈希取模
- random
  - 随机
- kemata
  - 一致性hash

- 把复杂的客户端交给服务器处理, 那么客户端使用的时候, 变得极其简单. 而服务器端的所有行为对于客户端是无感知的;
- 具体可用的实现方案

- twemproxy 「常用」
  - 推特开源的实现方案
  - <https://github.com/twitter/twemproxy>
  - <https://github.com/twitter/twemproxy/releases/download/0.5.0/twemproxy-0.5.0.tar.gz>
- prdixy
  - <https://github.com/joyieldInc/predixy>
- cluster 「redis提供的集群策略」
  - <http://redis.cn/topics/partitioning.html>
  - <https://redis.io/topics/partitioning>
- codis
  - <https://github.com/CodisLabs/codis>

## 3. 预分区

| 以上说的几种方式, 都有着共同的缺点就是不能拿来做数据库使用, 只能当作缓存使用;



### ○ 数据分治的问题

- 聚合操作很难实现
    - 如果非要实现,则需要移动数据进行计算.划不来;
  - 事务也很难实现

## 1. 解决方案

- a. hash tag

b. {name}: tom

{name}: jack

- Redis分区实现

## REDIS cluster-tutorial -- Redis中文资料站 -- Redis中国用户组(CRUG)

redis

<http://redis.cn/topics/cluster-tutorial.html>

## 4. 实操

### 1. twemproxy

#### twitter/twemproxy

A fast, light-weight proxy for memcached and redis - twitter/twemproxy

🔗 <https://github.com/twitter/twemproxy>

#### twitter/twemproxy

A fast, light-weight proxy for memcached and redis - twitter/twemproxy

🔗 <https://github.com/twitter/twemproxy/blob/master/README.md>

### 新版本的安装过程

- 通过编译好的包进行安装
  - 下载
  - 解压操作
  - 进行主到解压目录, twemproxy目录, 执行 `./configure`
  - 执行 `make`
  - 再执行 `make install`, 那么它就立刻安装好了;
  - 在`/usr/local/sbin/nutcracker`可执行程序, 就是 twemproxy`代理的可执行程序;
    - 同时在src下也会生成一个可执行程序;
  - 如果src目录的程序无法启动,那么尝试启动: '`usr/local/sbin/``这里的程序;
  - 启动代理, `conf`和`nutcracker`必须在同一文件夹下, 不然起不来。
- 通过源码安装
  - 如果没有 `git` 直接下载

Bash

```
1 yum install -y git
```

◦ <https://github.com/twitter/twemproxy.git>

◦ 安装工具

raven 5404

Bash

1 yum install -y libtool

- 进入目录

raven 5404

Bash

1 cd twemproxy

C#

```
1 $ cd twemproxy
2 $ autoreconf -fvi
3 $ ./configure --enable-debug=full
4 $ make
5 $ ./src/nutcracker -h # 测试是否安装成功
6 [root@hecs-215393 src]# ./nutcracker -h
7
8 # 输出日志如下所示:
9 This is nutcracker-0.5.0
10 async event backend: epoll
11 debugging assertions are enabled (--enable-debug=yes|full), nutcracker may be
less efficient
12
13 Usage: nutcracker [-?hVdDt] [-v verbosity level] [-o output file]
14 [-c conf file] [-s stats port] [-a stats addr]
15 [-i stats interval] [-p pid file] [-m mbuf size]
16
17 Options:
18 -h, --help : this help
19 -V, --version : show version and exit
20 -t, --test-conf : test configuration for syntax errors and exit
21 -d, --daemonize : run as a daemon
22 -D, --describe-stats : print stats description and exit
23 -v, --verbose=N : set logging level (default: 5, min: 0, max: 11)
24 -o, --output=S : set logging file (default: stderr)
25 -c, --conf-file=S : set configuration file (default:
conf/nutcracker.yml)
26 -s, --stats-port=N : set stats monitoring port (default: 22222)
27 -a, --stats-addr=S : set stats monitoring ip (default: 0.0.0.0)
28 -i, --stats-interval=N : set stats aggregation interval in msec (default:
30000 msec)
29 -p, --pid-file=S : set pid file (default: off)
30 -m, --mbuf-size=N : set size of mbuf chunk in bytes (default: 16384
bytes)
```

- 准备配置文件

配置文件在下载的源码目录当中的 `conf` 目录当中;

## Bash

```
1 [root@hecs-215393 conf]# pwd
2 /usr/local/tom/twemproxy/conf
3 [root@hecs-215393 conf]# ls
4 nutcracker.leaf.yml  nutcracker.root.yml  nutcracker.yml
5 [root@hecs-215393 conf]#
```

- 拷贝配置文件目录到到 `/usr/local/sbin/`

## Bash

```
1 cp -r /usr/local/tom/twemproxy/conf /usr/local/sbin/
2 # 如果自己拷贝, 源目录必须改成自己的目录. 目标目录是一样的;
```

- 到配置文件目录修改配置文件

拷贝一份再进行修改.

```
-rw-r--r--. 1 root root 209 Nov 4 23:23 nutcracker.leaf.yml
-rw-r--r--. 1 root root 151 Nov 4 23:23 nutcracker.root.yml
-rw-r--r--. 1 root root 250 Nov 5 00:16 nutcracker.yml
-rw-r--r--. 1 root root 1301 Nov 4 23:23 nutcracker.yml.bak
[root@hecs-215393 conf]#
```

→配置文件

- 内容如下所示:

## YAML

```
1 alpha:
2   listen: 127.0.0.1:22121 # nutcracker 监听端口号
3   hash: fnv1a_64 # 指定的hash算法
4   distribution: ketama # 描述
5   redis_auth: Tom12348765 # 认证密码, redis.conf当中写的密码, 6379.conf密码
6   auto_eject_hosts: true
7   redis: true # 代理类型是`redis`类型
8   server_retry_timeout: 2000 # 超时时间
9   server_failure_limit: 1
10  servers: # 要代理的`redis`服务器地址
11    - 127.0.0.1:6379:1
12    - 127.0.0.1:6380:1
```

- 验证配置文件是否正确

进入到 `/usr/local/sbin` 目录当中, 可以看到 `nutcracker` 可执行文件.

```
[root@hecs-215393 sbin]# pwd
/usr/local/sbin
[root@hecs-215393 sbin]# ls
```

```
[root@hecs-215393 sbin]# conf keepalived nutcracker
```

## Bash

```
1 [root@hecs-215393 sbin]# ./nutcracker -t
2 nutcracker: configuration file 'conf/nutcracker.yml' syntax is ok
3 [root@hecs-215393 sbin]#
```

- `nutcracker --help` 使用

## Bash

```
1 Usage: nutcracker [-?hVdDt] [-v verbosity level] [-o output file]
2                               [-c conf file] [-s stats port] [-a stats addr]
3                               [-i stats interval] [-p pid file] [-m mbuf size]
4 复制代码
5 Options:
6 -h, -help          : 查看帮助文档, 显示命令选项
7 -V, -version       : 查看nutcracker版本
8 -t, -test-conf     : 测试配置脚本的正确性
9 -d, -daemonize     : 以守护进程运行
10 -D, -describe-stats : 打印状态描述
11 -v, -verbosity=N   : 设置日志级别 (default: 5, min: 0, max: 11)
12 -o, -output=S      : 设置日志输出路径, 默认为标准错误输出 (default:
13           stderr)          : 指定配置文件路径 (default:
14           conf/nutcracker.yml)
15 -s, -stats-port=N   : 设置状态监控端口, 默认22222 (default: 22222)
16 -a, -stats-addr=S    : 设置状态监控IP, 默认0.0.0.0 (default: 0.0.0.0)
17 -i, -stats-interval=N : 设置状态聚合间隔 (default: 30000 msec)
18 -p, -pid-file=S     : 指定进程pid文件路径, 默认关闭 (default: off)
```

- 配置文件

## YAML

```
1 alpha:
2   listen: 127.0.0.1:22121
3   hash: fnv1a_64
4   distribution: ketama
5   auto_eject_hosts: true
6   redis: true
7   redis_auth: Tom12348765 # 有密码必须配上它
8   hash_tag: "{}" # 让key映射到同一台Redis服务器上去;
9   server_retry_timeout: 2000
10  server_failure_limit: 1
11  servers:
12    - 127.0.0.1:6379:1
13    - 127.0.0.1:6380:1
```

- 以上步骤操作完成之后,可以在系统的任意地方使用 `nutcracker` 程序了;

```
[root@iZ2ze8biiph4vi0x5z06heZ bin]# nu
numfmt      nutcracker
[root@iZ2ze8biiph4vi0x5z06heZ bin]# nu
numfmt      nutcracker
[root@iZ2ze8biiph4vi0x5z06heZ bin]# nu
```

有提示了,表示ok

- 启动的时候指定配置文件. 启动脚本说明如下所示:

这里的配置文件可以写在任意的地方,只要启动的时候指定一下就可以了;

## Bash

```
1 # 进入到/usr/local/sbin/目录中;
2 ./nutcracker -d -c ./conf/nutcracker.yml -p
  /usr/local/raven/twemproxy/twemproxy.pid -o
  /usr/local/raven/twemproxy/twemproxy.log
3
4 # 启动的时候,这里的路径必须修改成自己的路径. 不要直接拷贝.
5 ./nutcracker -d -c ./conf/nutcracker.yml -p /usr/local/java-18/twemproxy-
  file/twemproxy.pid -o /usr/local/java-18/twemproxy-file/twemproxy.log
```

## ● 连接测试

### Bash

```
1 redis-cli -p 22121 -a # 认证密码“如果没有则不写”
```

## ● hash tag

### Bash

```
1 set {key}name tom
2 set {key}sex 1
3 set {key}age 18
```

使用客户端直接连接 redis 服务,看看key落到哪个服务器上去了;

## ● 缺点说明

### Bash

```
1 127.0.0.1:22121> keys *
2 Error: Server closed the connection # 数据分治了, 无法统计
3 127.0.0.1:22121>
4
5 127.0.0.1:22121> WATCH name
6 Error: Server closed the connection # 原因同上, 无法查找到具体的key在哪个服务器上
7 127.0.0.1:22121>
8
9 127.0.0.1:22121> MULTI
10 Error: Server closed the connection
11 127.0.0.1:22121>
```

### Java

```
1 # 从本地上传文件到服务器
2 scp -r local_folder remote_username@remote_ip:remote_folder 或者
3 scp -r local_folder remote_ip:remote_folder
4
5 # 从远程服务器下载到本地
6 scp -r remote_username@remote_ip remote_folder :local_folder:
7
8 scp root@114.115.184.250:/usr/local/soft/tabby-1.0.173-macos-x86_64.pkg
   /Users/jeren/Desktop
```

## 2. predixy

课下自己完成;

## joyieldInc/predixy

A high performance and fully featured proxy for redis, support redis sentinel and redis cluster - joyieldInc/predixy

 <https://github.com/joyieldInc/predixy>

## predixy:一款吊打众对手的redis代理，你喜欢吗?\_rebaic的专栏-CSDN博客\_predixy

predixy，一款高性能全特征redis代理，支持redis sentinel，redis cluster。功能全面，性能出众，本文对比了predixy、twemproxy、codis、cerberus几款redis代理的特征和性能。

 <https://blog.csdn.net/rebaic/article/details/76384028>

- 下载可执行文件;

### Bash

```
1 wget https://github.com/joyieldInc/predixy/releases/download/1.0.5/predixy-1.0.5-bin-amd64-linux.tar.gz
2
3 # 解压
4 tar -xzvf predixy-1.0.5-bin-amd64-linux.tar.gz
```

```
[root@iZ2ze8biiph4vi0x5z06heZ predixy-1.0.5]# ll
total 36
drwxrwxr-x 2 501 501 4096 Oct 20 2018 bin
drwxrwxr-x 2 501 501 4096 Oct 20 2018 conf
drwxrwxr-x 3 501 501 4096 Oct 20 2018 doc
-rw-rw-r-- 1 501 501 1537 Oct 20 2018 LICENSE
-rw-rw-r-- 1 501 501 5680 Oct 20 2018 README_CN.md
-rw-rw-r-- 1 501 501 4200 Oct 20 2018 README.md
drwxrwxr-x 2 501 501 4096 Oct 20 2018 test
[root@iZ2ze8biiph4vi0x5z06heZ predixy-1.0.5]#
```

- 修改配置文件

## Bash

```
1  解压目录下的conf目录。
2  # 修改predixy.conf文件。
3  # 修改sentinel.conf文件,注意先各复制一份,别直接修改;
4
5  # 当前行复制到行尾的内容
6  .:, $y -> 回车
7  G, 到行尾
8  p -> 粘贴
9
10 # 快速替换当前行到行尾的字符
11 .:, $s/要替换的字符/替换的内容 / -> 回车即可;
```

## ● 相关配置文件

### Bash

```
1  # predixy.conf
2  # 配置连接ip地址
3  Bind 127.0.0.1:7617
4  # 配置服务模式
5  Include sentinel.conf # 哨兵模式
```

## Bash

```
1 # sentinel.conf配置
2 SentinelServerPool {
3     Databases 16
4     Hash crc16
5     HashTag "{}"
6     Distribution modula
7     MasterReadPriority 60
8     StaticSlaveReadPriority 50
9     DynamicSlaveReadPriority 50
10    RefreshInterval 1
11    ServerTimeout 1
12    ServerFailureLimit 10
13    ServerRetryTimeout 1
14    KeepAlive 120
15    Sentinels { # 配置哨兵,三个. 一个哨兵集群可以监控多套redis服务
16        + 127.0.0.1:36379
17        + 127.0.0.1:36380
18        + 127.0.0.1:36381
19    }
20    # 这个组的名称是在哨兵配置文件中指定的;
21    Group mymaster1 { # 配置哨兵监控的redis服务
22    }
23    }
24    # 这个组的名称是在哨兵配置文件中指定的;
25    Group mymaster2 {
26    }
27 }
28 }
```

## Bash

```
1 # 36379.conf 哨兵的配置文件
2 port 36379
3 sentinel monitor mymaster1 127.0.0.1 46379 2
4 sentinel monitor mymaster2 127.0.0.1 56379 2
```

- 启动 predixy 程序, 测试一波

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ bin]# ./predixy ../conf/predixy.conf
2 2021-05-11 00:56:39.518390 N Proxy.cpp:112 predixy listen in 127.0.0.1:7617
3 2021-05-11 00:56:39.518490 N Proxy.cpp:143 predixy running with
Name:PredixyExample Workers:1
4 2021-05-11 00:56:39.518652 N Handler.cpp:454 h 0 create connection pool for
server 127.0.0.1:36379
5 2021-05-11 00:56:39.518670 N ConnectConnectionPool.cpp:42 h 0 create server
connection 127.0.0.1:36379 5
6 2021-05-11 00:56:39.518783 N Handler.cpp:454 h 0 create connection pool for
server 127.0.0.1:36381
7 2021-05-11 00:56:39.518794 N ConnectConnectionPool.cpp:42 h 0 create server
connection 127.0.0.1:36381 6
8 2021-05-11 00:56:39.519259 N StandaloneServerPool.cpp:472 sentinel server pool
group mymaster1 create slave server 127.0.0.1:46380
9 2021-05-11 00:56:39.519270 N StandaloneServerPool.cpp:422 sentinel server pool
group mymaster2 create master server 127.0.0.1:56379
10 2021-05-11 00:56:39.519278 N StandaloneServerPool.cpp:422 sentinel server pool
group mymaster1 create master server 127.0.0.1:46379
11 2021-05-11 00:56:39.519304 N StandaloneServerPool.cpp:472 sentinel server pool
group mymaster2 create slave server 127.0.0.1:56380
12 2021-05-11 00:56:41.522714 N Handler.cpp:454 h 0 create connection pool for
server 127.0.0.1:36380
13 2021-05-11 00:56:41.522773 N ConnectConnectionPool.cpp:42 h 0 create server
connection 127.0.0.1:36380 7
14 2021-05-11 00:57:34.621484 N Handler.cpp:371 h 0 accept c 127.0.0.1:35696 8
assign to h 0
```

- 使用redis-cli连接predixy

## Bash

```
1 ./redis cli -p 7671
2 ./redis-cli -p 7617# 可以愉快的测试了
3 # 添加key,获取内容
4 # 分别连接两个主机,看看这些key落到哪个主机上了
5 # 测试哨兵,断开其中一个主机;然后再获取内容,看看是否还有.
6 # 测试hash tag
```

## Bash

```
1 # hash tag测试
2 127.0.0.1:7617> set {order}k1 23324
3 OK
4 127.0.0.1:7617> set {order}k2 2ad234asdf
5 OK
6 127.0.0.1:7617> get {order}k1
7 "23324"
8 127.0.0.1:7617>
9
10 # 带有hash tag{}的key会被映射到同一台服务器上了;
11 127.0.0.1:56379> keys *
12 1) "{order}k2"
13 2) "{order}k1"
14 3) "age"
15 4) "n1"
16 127.0.0.1:56379> get {order}k2
17 "2ad234asdf"
18 127.0.0.1:56379>
```

- 这种多组模式的问题

## Bash

```
1 # 1. 不支持keys *
2 127.0.0.1:7617> keys *
3 (error) ERR unknown command 'keys'
4 127.0.0.1:7617>
5 # 2. 不支持watch, 多组模式下, 尽量使用hash tag, 即key在同一台服务器上,但是依然不
6 # 持事务
7
8 # 数据被映射到同一台服务器上了.
9 127.0.0.1:56379> keys *
10 1) "{order}k2"
11 2) "{order}k1"
12 3) "age"
13 4) "n1"
14 127.0.0.1:56379> get {order}k2
15 "2ad234asdf"
16 127.0.0.1:56379>
17
18 # watch hash tag
19 127.0.0.1:7617> watch {order}k1
20 (error) ERR forbid transaction in current server pool
21 127.0.0.1:7617>
22
23 # 3. 不支持事务
24 127.0.0.1:7617> MULTI
25 (error) ERR forbid transaction in current server pool
26 127.0.0.1:7617>
```

### • 单组模式支持事务

## Bash

```
1 # 修改sentinel.conf文件
2
3 # 注释掉,mymaster2, 重新启动predixy
4 #Group mymaster2 {
5 #}
```

## Bash

```
1 # 只监控一组了
2
3 [root@iz2ze8biiph4vi0x5z06heZ bin]# ./predixy ../conf/predixy.conf
4 2021-05-11 01:20:07.683552 N Proxy.cpp:112 predixy listen in 127.0.0.1:7617
5 2021-05-11 01:20:07.683656 N Proxy.cpp:143 predixy running with
    Name:PredixyExample Workers:1
6 2021-05-11 01:20:07.683816 N Handler.cpp:454 h 0 create connection pool for
    server 127.0.0.1:36381
7 2021-05-11 01:20:07.683835 N ConnectConnectionPool.cpp:42 h 0 create server
    connection 127.0.0.1:36381 5
8 2021-05-11 01:20:07.683946 N Handler.cpp:454 h 0 create connection pool for
    server 127.0.0.1:36380
9 2021-05-11 01:20:07.683957 N ConnectConnectionPool.cpp:42 h 0 create server
    connection 127.0.0.1:36380 6
10 2021-05-11 01:20:07.684376 N StandaloneServerPool.cpp:472 sentinel server pool
    group mymaster1 create slave server 127.0.0.1:46379
11 2021-05-11 01:20:07.684393 N StandaloneServerPool.cpp:422 sentinel server pool
    group mymaster1 create master server 127.0.0.1:46380
```

- 此时再设置hash tag,就可以支持事务了

## Bash

```
1 # 相关操作
2 127.0.0.1:7617> set {order}k1 23423423432 # 设置hash tag
3 OK
4 127.0.0.1:7617> set {order}k2 abadfadsfadf
5 OK
6 127.0.0.1:7617> get {order}k1
7 "23423423432"
8 127.0.0.1:7617> get {order}k2
9 "abadfadsfadf"
10 127.0.0.1:7617> WATCH {order}k1 # 开启事务
11 OK
12 127.0.0.1:7617> MULTI
13 OK
14 127.0.0.1:7617> set name jack
15 QUEUED
16 127.0.0.1:7617> set age 18
17 QUEUED
18 127.0.0.1:7617> EXEC
19 1) OK
20 2) OK
21 127.0.0.1:7617>
22
23
24 # 数据被映射到46380这台服务上了;
25 127.0.0.1:46380> KEYS *
26 1) "{order}k2"
27 2) "{order}k1"
28 3) "name"
29 4) "n2"
30 127.0.0.1:46380> GET {order}k1
31 "23423423432"
32 127.0.0.1:46380> GET {order}k2
33 "abadfadsfadf"
34 127.0.0.1:46380>
```

## 3. cluster 「无主模型」

redis自带的.

重要的内容: cluster

## REDIS 分区 -- Redis中国用户组(CRUG)

redis

<http://redis.cn/topics/partitioning.html>

- redis和memcached



- **客户端分区**就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。大多数客户端已经实现了客户端分区。
- **代理分区**意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis和memcached的一种代理实现就是Twemproxy
- **查询路由(Query routing)**的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接*redirected*到正确的redis节点。「无主模型」

- redis的脚本

Bash

```
1 /usr/local/redis-6.0.9/utils/create-cluster # 脚本目录
```

- create-cluster命令说明

Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# ./create-cluster -h
2 Usage: ./create-cluster [start|create|stop|watch|tail|clean|call]
3 start      -- Launch Redis Cluster instances.
4 create [-f] -- Create a cluster using redis-cli --cluster create.
5 stop       -- Stop Redis Cluster instances.
6 watch      -- Show CLUSTER NODES output (first 30 lines) of first node.
7 tail <id>   -- Run tail -f of instance at base port + ID.
8 tailall    -- Run tail -f for all the log files at once.
9 clean      -- Remove all instances data, logs, configs.
10 clean-logs -- Remove just instances logs.
11 call <cmd>  -- Call a command (up to 7 arguments) on all nodes.
```

- 创建Redis实例. 共6个, 一主一从;

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# ./create-cluster start
2 Starting 30001
3 Starting 30002
4 Starting 30003
5 Starting 30004
6 Starting 30005
7 Starting 30006
```

- 分配slot

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# ./create-cluster create
2 >>> Performing hash slots allocation on 6 nodes...
3 Master[0] -> Slots 0 - 5460 # 分配槽位
4 Master[1] -> Slots 5461 - 10922 # 分配槽位
5 Master[2] -> Slots 10923 - 16383 # 分配槽位
6 Adding replica 127.0.0.1:30005 to 127.0.0.1:30001 # 三个主机
7 Adding replica 127.0.0.1:30006 to 127.0.0.1:30002
8 Adding replica 127.0.0.1:30004 to 127.0.0.1:30003
9 >>> Trying to optimize slaves allocation for anti-affinity
10 [WARNING] Some slaves are in the same host as their master
11 M: 28494e4a0fd74cf06f6d728f7fd71b5830970697 127.0.0.1:30001
12     slots:[0-5460] (5461 slots) master
13 M: b70eca7695eb3befa6dcfc6c4e4069d50adb3941 127.0.0.1:30002
14     slots:[5461-10922] (5462 slots) master
15 M: 328d79e95ee8a0603fa9c901f38411449a5b4a6f 127.0.0.1:30003
16     slots:[10923-16383] (5461 slots) master
17 S: 7d402de5f87f0b2af14833f611e3da46458a6f40 127.0.0.1:30004
18     replicates 328d79e95ee8a0603fa9c901f38411449a5b4a6f
19 S: c965fdce21ecda07640fa0a88a1069eb548c44c8 127.0.0.1:30005
20     replicates 28494e4a0fd74cf06f6d728f7fd71b5830970697
21 S: 272e86c46f3789533de09f8b91306454b01e8663 127.0.0.1:30006
22     replicates b70eca7695eb3befa6dcfc6c4e4069d50adb3941
23 Can I set the above configuration? (type 'yes' to accept): yes
24 >>> Nodes configuration updated
25 >>> Assign a different config epoch to each node
26 >>> Sending CLUSTER MEET messages to join the cluster
27 Waiting for the cluster to join
28 .
29 >>> Performing Cluster Check (using node 127.0.0.1:30001)
30 M: 28494e4a0fd74cf06f6d728f7fd71b5830970697 127.0.0.1:30001
31     slots:[0-5460] (5461 slots) master
32     1 additional replica(s)
33 M: b70eca7695eb3befa6dcfc6c4e4069d50adb3941 127.0.0.1:30002
```

```
raven5 34 slots:[5461-10922] (5462 slots) master
raven5 35 1 additional replica(s)
raven5 36 S: c965fdce21ecda07640fa0a88a1069eb548c44c8 127.0.0.1:30005
raven5 37 slots: (0 slots) slave
raven5 38 replicates 28494e4a0fd74cf06f6d728f7fd71b5830970697
raven5 39 M: 328d79e95ee8a0603fa9c901f38411449a5b4a6f 127.0.0.1:30003
raven5 40 slots:[10923-16383] (5461 slots) master
raven5 41 1 additional replica(s)
raven5 42 S: 7d402de5f87f0b2af14833f611e3da46458a6f40 127.0.0.1:30004
raven5 43 slots: (0 slots) slave
raven5 44 replicates 328d79e95ee8a0603fa9c901f38411449a5b4a6f
raven5 45 S: 272e86c46f3789533de09f8b91306454b01e8663 127.0.0.1:30006
raven5 46 slots: (0 slots) slave
raven5 47 replicates b70eca7695eb3befa6dcfc6c4e4069d50adb3941
raven5 48 [OK] All nodes agree about slots configuration.
raven5 49 >>> Check for open slots...
raven5 50 >>> Check slots coverage...
raven5 51 [OK] All 16384 slots covered.
raven5 52 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]#
```

## ● 测试

### Bash

```
raven5 1 ./redis-cli -p 30001
raven5 2
raven5 3 127.0.0.1:30001> set k1 tom
raven5 4 (error) MOVED 12706 127.0.0.1:30003 # 报错了.这个是因为当前服务器没有与之对应的
raven5 slot
raven5 5 127.0.0.1:30001>
```

## ● 使用 **redis-cli -c -p** 进行连接「集群模式下的连接」

## Ruby

```
1 [root@iZ2ze8biiph4vi0x5z06heZ src]# ./redis-cli -c -p 30001
2 127.0.0.1:30001> set k1 tom
3 -> Redirected to slot [12706] located at 127.0.0.1:30003 # 重到30003服务器上了
4 OK
5 127.0.0.1:30003> # 直接连接了30003服务器
6
7 127.0.0.1:30001> get k1
8 -> Redirected to slot [12706] located at 127.0.0.1:30003
9 "tom"
10 127.0.0.1:30003> set k2 jack
11 -> Redirected to slot [449] located at 127.0.0.1:30001
12 OK
13 127.0.0.1:30001> set pet dog
14 OK
15 127.0.0.1:30001> get k2
16 "jack"
17 127.0.0.1:30001> get k1
18 -> Redirected to slot [12706] located at 127.0.0.1:30003
19 "tom"
20 127.0.0.1:30003>
```

## ● 事务支持情况

使用 hash tag.

## Ruby

```
1 127.0.0.1:30003> WATCH k1
2 OK
3 127.0.0.1:30003> set age 20
4 -> Redirected to slot [741] located at 127.0.0.1:30001
5 OK
6 127.0.0.1:30001> MULTI
7 OK
8 127.0.0.1:30001> set age 30
9 QUEUED
10 127.0.0.1:30001> set address yan
11 QUEUED
12 127.0.0.1:30001> set name jack # 又重定向其它服务器了.此时事务有问题了.
13 -> Redirected to slot [5798] located at 127.0.0.1:30002
14 OK
15 127.0.0.1:30002>
16 127.0.0.1:30002> exec
17 (error) ERR EXEC without MULTI
18 127.0.0.1:30002>
```

- 正确姿势,使用 hash tag

- 保证所有的key都存到一台服务器上;
- 事务支持,只能支持一台服务器上做操作;

## Bash

```
1 127.0.0.1:30002> set {oo}k1 tom
2 -> Redirected to slot [1629] located at 127.0.0.1:30001
3 OK
4 127.0.0.1:30001> set {oo}k2 jack
5 OK
6 127.0.0.1:30001> set {oo}age 18
7 OK
8 127.0.0.1:30001> watch {oo}age
9 OK
10 127.0.0.1:30001> set {oo}k1 rose
11 OK
12 127.0.0.1:30001> MULTI
13 OK
14 127.0.0.1:30001> set {oo}k1 jerry
15 QUEUED
16 127.0.0.1:30001> set {oo}age 30
17 QUEUED
18 127.0.0.1:30001> exec
19 1) OK
20 2) OK
21 127.0.0.1:30001>
22
23 # hash tag使带有{oo}的key映射到同一台服务器了.此时事务没有啥问题了;
```

- 手工创建cluster

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# ./create-cluster stop # 停止所有
  服务
2 Stopping 30001
3 Stopping 30002
4 Stopping 30003
5 Stopping 30004
6 Stopping 30005
7 Stopping 30006
8 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# ./create-cluster clean # 清除一
  下数据
9 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]#
```

- 相关命令

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# redis-cli --cluster help
```

```
raven 5404 2 Cluster Manager Commands:
raven 5404 3   create          host1:port1 ... hostN:portN # 创建集群
raven 5404 4               --cluster-replicas <arg> # 从机数量
raven 5404 5   check           host:port
raven 5404 6               --cluster-search-multiple-owners
raven 5404 7   info            host:port
raven 5404 8               --cluster-search-multiple-owners
raven 5404 9               --cluster-fix-with-unreachable-masters
raven 5404 10  reshard        host:port
raven 5404 11               --cluster-from <arg>
raven 5404 12               --cluster-to <arg>
raven 5404 13               --cluster-slots <arg>
raven 5404 14               --cluster-yes
raven 5404 15               --cluster-timeout <arg>
raven 5404 16               --cluster-pipeline <arg>
raven 5404 17               --cluster-replace
raven 5404 18   rebalance      host:port
raven 5404 19               --cluster-weight <node1=w1...nodeN=wN>
raven 5404 20               --cluster-use-empty-masters
raven 5404 21               --cluster-timeout <arg>
raven 5404 22               --cluster-simulate
raven 5404 23               --cluster-pipeline <arg>
raven 5404 24               --cluster-threshold <arg>
raven 5404 25               --cluster-replace
raven 5404 26   add-node       new_host:new_port existing_host:existing_port
raven 5404 27               --cluster-slave
raven 5404 28               --cluster-master-id <arg>
raven 5404 29   del-node       host:port node_id
raven 5404 30               host:port command arg arg .. arg
raven 5404 31               --cluster-only-masters
raven 5404 32               --cluster-only-replicas
raven 5404 33   call            host:port milliseconds
raven 5404 34   set-timeout    host:port
raven 5404 35               host:port
raven 5404 36               --cluster-from <arg>
raven 5404 37               --cluster-copy
raven 5404 38               --cluster-replace
raven 5404 39   import          host:port backup_directory
raven 5404 40   help
raven 5404 41
raven 5404 42 For check, fix, reshard, del-node, set-timeout you can specify the host and
raven 5404 43 port of any working node in the cluster.
raven 5404 44 [root@iz2ze8biiph4vi0x5z06heZ create-cluster]#
```

- 自己创建 **cluster** 操作

- `./redis-cli --cluster help`, 查看帮助手册

## Bash

```
1 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# ./create-cluster start
2 Starting 30001
3 Starting 30002
4 Starting 30003
5 Starting 30004
6 Starting 30005
7 Starting 30006 # 此处可以自己决定启动哪些个redis服务,这里为了方便.就这么干了. 不要指定
    主从关系
8
9 # 这里指定的ip:port表示redis实例,分配slot的时候这些服务必须都处于启动状态;
10 # 分配slot
11 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# redis-cli --cluster create
    127.0.0.1:30001 127.0.0.1:30002 127.0.0.1:30003 127.0.0.1:30004
    127.0.0.1:30005 127.0.0.1:30006 --cluster-replicas 1
12 # 分别指定redis的host和port
13 # 指定从机数量
14
15 # 重要操作
16 # reshard 数据迁移操作;
17 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# redis-cli --cluster reshard
    127.0.0.1:30001 # 数据迁移命令reshard + 任意一个存活的节点
18 >>> Performing Cluster Check (using node 127.0.0.1:30001)
19 M: d6f57ea6f5644682fa532768dee4b510ca8aa8cf 127.0.0.1:30001
20     slots:[2000-5460] (3461 slots) master
21     1 additional replica(s)
22 M: c9961dc23542a87a49a8b4ccf0e1c9bdc256fce7 127.0.0.1:30003
23     slots:[10923-16383] (5461 slots) master
24     1 additional replica(s)
25 M: 98ec370d401b02af5a27af301b35a52add6cb579 127.0.0.1:30002
26     slots:[0-1999],[5461-10922] (7462 slots) master
27     1 additional replica(s)
28 S: 1d76145d1e87839329b00aca2cbcb72c04a08447 127.0.0.1:30005
29     slots: (0 slots) slave
30     replicates 98ec370d401b02af5a27af301b35a52add6cb579
31 S: 38377d8f85bde1386aed5eb9ea57de12fedf9b92 127.0.0.1:30004
32     slots: (0 slots) slave
33     replicates d6f57ea6f5644682fa532768dee4b510ca8aa8cf
34 S: e6473d782a3387ec0f37e88561380b0e1755256f 127.0.0.1:30006
35     slots: (0 slots) slave
36     replicates c9961dc23542a87a49a8b4ccf0e1c9bdc256fce7
37
38 [OK] All nodes agree about slots configuration.
39 >>> Check for open slots...
40 >>> Check slots coverage...
41 [OK] All 16384 slots covered.
```

```
raven 5404 42 # 你想要移动多少个slot
raven 5404 43 How many slots do you want to move (from 1 to 16384)? 1000
raven 5404 44 # 你想往哪个节点上移动,写上边的id「那个大长串」
raven 5404 45 What is the receiving node ID? d6f57ea6f5644682fa532768dee4b510ca8aa8cf
raven 5404 46 # 你想从哪些个节点上移动
raven 5404 47 Please enter all the source node IDs.
raven 5404 48 # all 表示从所有节点上随机抽取一些;
raven 5404 49 Type 'all' to use all the nodes as source nodes for the hash slots.
raven 5404 50 # 当填写了id的时候, done表示结束选择;
raven 5404 51 Type 'done' once you entered all the source nodes IDs.
raven 5404 52 Source node #1: c9961dc23542a87a49a8b4ccf0e1c9bdc256fce7 # 从30002移动
raven 5404 53 Source node #2: done # 表示配置完成;开始移动数据
raven 5404 54
raven 5404 55 Do you want to proceed with the proposed reshard plan (yes/no)? yes
raven 5404 56
raven 5404 57 # 查看信息
raven 5404 58 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# redis-cli --cluster info
raven 5404 59 127.0.0.1:30001
raven 5404 60 127.0.0.1:30001 (d6f57ea6...) -> 2 keys | 4461 slots | 1 slaves.
raven 5404 61 127.0.0.1:30003 (c9961dc2...) -> 1 keys | 5461 slots | 1 slaves.
raven 5404 62 127.0.0.1:30002 (98ec370d...) -> 0 keys | 6462 slots | 1 slaves.
raven 5404 63 [OK] 3 keys in 3 masters.
raven 5404 64 0.00 keys per slot on average.
raven 5404 65 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]#
raven 5404 66 # 检查cluster状态
raven 5404 67 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]# redis-cli --cluster check
raven 5404 68 127.0.0.1:30001 (d6f57ea6...) -> 2 keys | 4461 slots | 1 slaves.
raven 5404 69 127.0.0.1:30003 (c9961dc2...) -> 1 keys | 5461 slots | 1 slaves.
raven 5404 70 127.0.0.1:30002 (98ec370d...) -> 0 keys | 6462 slots | 1 slaves.
raven 5404 71 [OK] 3 keys in 3 masters.
raven 5404 72 0.00 keys per slot on average.
raven 5404 73 >>> Performing Cluster Check (using node 127.0.0.1:30001)
raven 5404 74 M: d6f57ea6f5644682fa532768dee4b510ca8aa8cf 127.0.0.1:30001
raven 5404 75 slots:[0-999],[2000-5460] (4461 slots) master
raven 5404 76 1 additional replica(s)
raven 5404 77 M: c9961dc23542a87a49a8b4ccf0e1c9bdc256fce7 127.0.0.1:30003
raven 5404 78 slots:[10923-16383] (5461 slots) master
raven 5404 79 1 additional replica(s)
raven 5404 80 M: 98ec370d401b02af5a27af301b35a52add6cb579 127.0.0.1:30002
raven 5404 81 slots:[1000-1999],[5461-10922] (6462 slots) master
raven 5404 82 1 additional replica(s)
raven 5404 83 S: 1d76145d1e87839329b00aca2cbc72c04a08447 127.0.0.1:30005
raven 5404 84 slots: (0 slots) slave
raven 5404 85 replicates 98ec370d401b02af5a27af301b35a52add6cb579
raven 5404 86 S: 38377d8f85bde1386aed5eb9ea57de12fedf9b92 127.0.0.1:30004
raven 5404 87 slots: (0 slots) slave
```

```
raven 5404 88      replicates d6f57ea6f5644682fa532768dee4b510ca8aa8cf
raven 5404 89 S: e6473d782a3387ec0f37e88561380b0e1755256f 127.0.0.1:30006
raven 5404 90   slots: (0 slots) slave
raven 5404 91   replicates c9961dc23542a87a49a8b4ccf0e1c9bdc256fce7
raven 5404 92 [OK] All nodes agree about slots configuration.
raven 5404 93 >>> Check for open slots...
raven 5404 94 >>> Check slots coverage...
raven 5404 95 [OK] All 16384 slots covered.
raven 5404 96 [root@iZ2ze8biiph4vi0x5z06heZ create-cluster]#
```

## Cluster Manager Commands:

**create**

host1:port1 ... hostN:portN

--cluster-relicas <arg>

**check**

host:port

--cluster-search-multiple-owners

host:port

host:port

--cluster-search-multiple-owners

--cluster-fix-with-unreachable-masters

**reshard**

host:port

--cluster-from <arg>

--cluster-to <arg>

--cluster-slots <arg>

--cluster-yes

--cluster-timeout <arg>

--cluster-pipeline <arg>

--cluster-replace

**rebalance**

host:port

--cluster-weight <node1=w1...nodeN=wN>

--cluster-use-empty-masters

--cluster-timeout <arg>

--cluster-simulate

--cluster-pipeline <arg>

--cluster-threshold <arg>

--cluster-replace

**add-node**

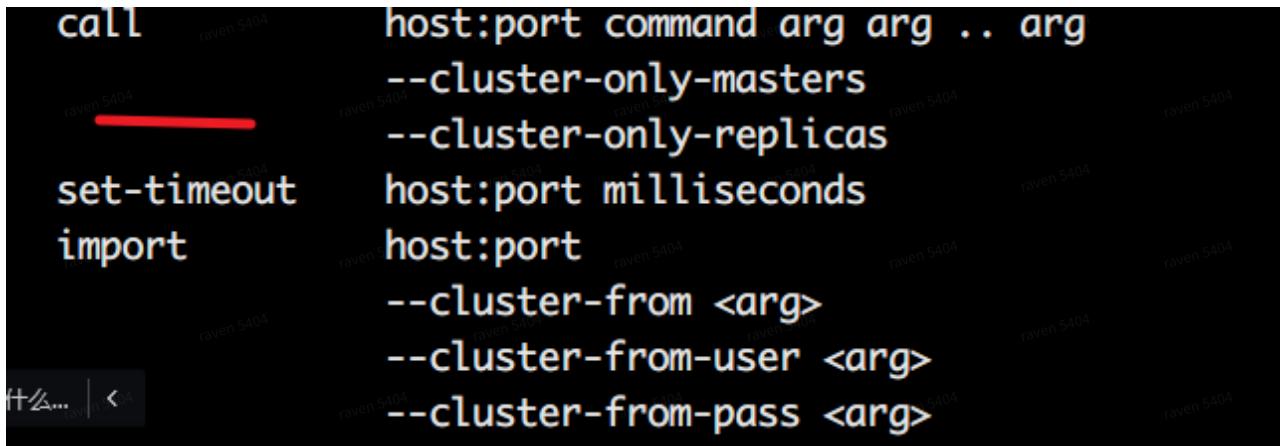
new\_host:new\_port existing\_host:existing\_

--cluster-slave

--cluster-master-id <arg>

**del-node**

host:port node\_id

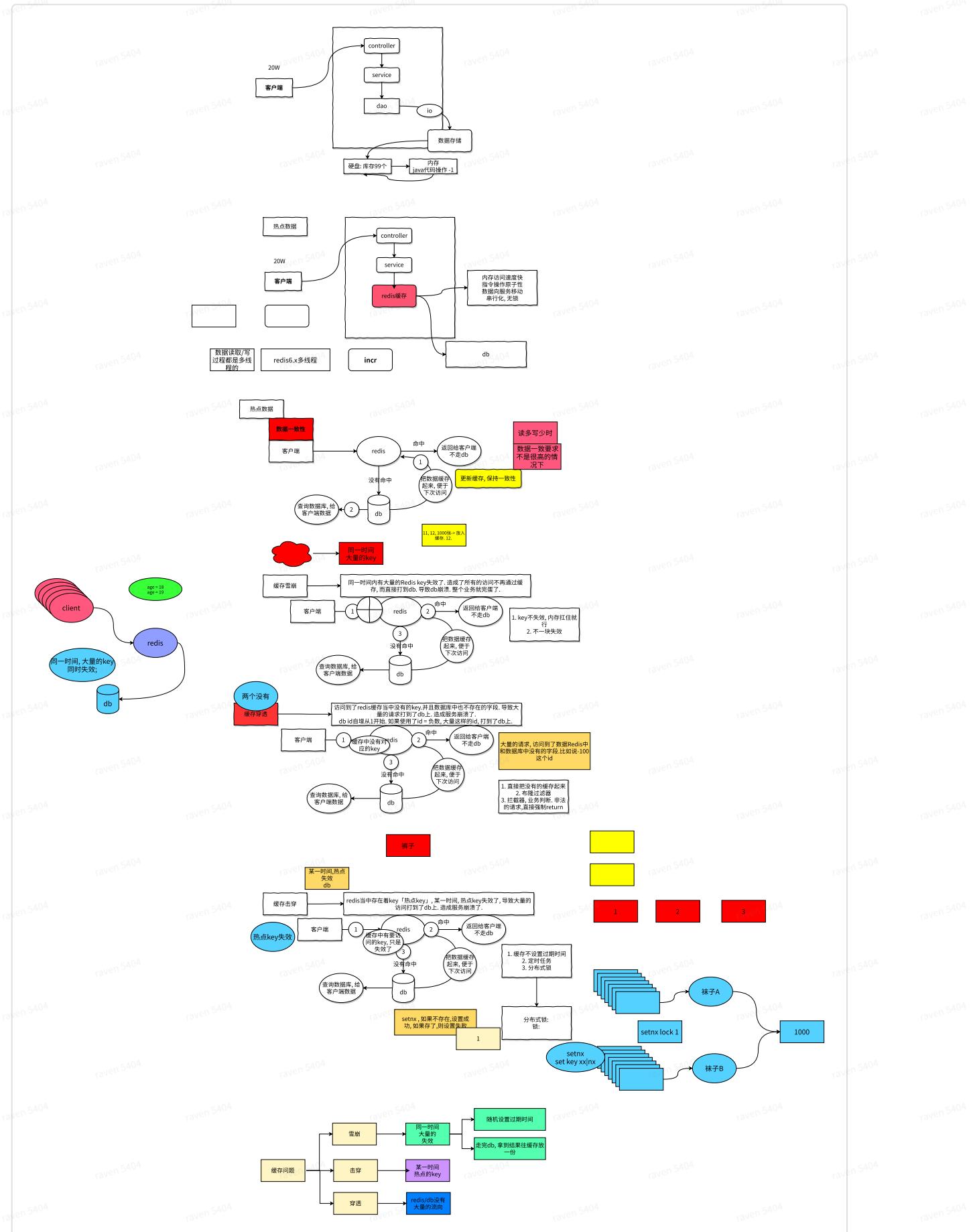


## 代理方式

cluster 「重点」, 无主模型

- 数据预分区思想.
- hash slot思想;
  - 实操,自己做一下;
  - predixy, 「挺厉害的, 自己实践一下」

## 第十章: 缓存



## 1. 缓存雪崩

如果缓存在某一个时刻出现大规模的key失效，那么就会导致大量的请求打在了数据库上面，导致数据库压力巨大，如果在高并发的情况下，可能瞬间就会导致数据库宕机。这时候如果运维马上又

重启数据库，马上又会有新的流量把数据库打死。这就是缓存雪崩。

造成缓存雪崩的关键在于同一时间的大规模的key失效，为什么会出现这个问题，主要有两种可能：**第一种是Redis宕机，第二种可能就是采用了相同的过期时间。**

- 解决方案「设计」

均匀过期：设置不同的过期时间，让缓存失效的时间尽量均匀，避免相同的过期时间导致缓存雪崩，造成大量数据库的访问

分级缓存：第一级缓存失效的基础上，访问二级缓存，每一级缓存的失效时间都不同。

热点数据缓存永远不过期。

- 物理不过期，针对**热点key**不设置过期时间
- 逻辑过期，把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建

保证Redis缓存的高可用，防止Redis宕机导致缓存雪崩的问题。**可以使用主从+哨兵，Redis集群来避免Redis全盘崩溃的情况。**

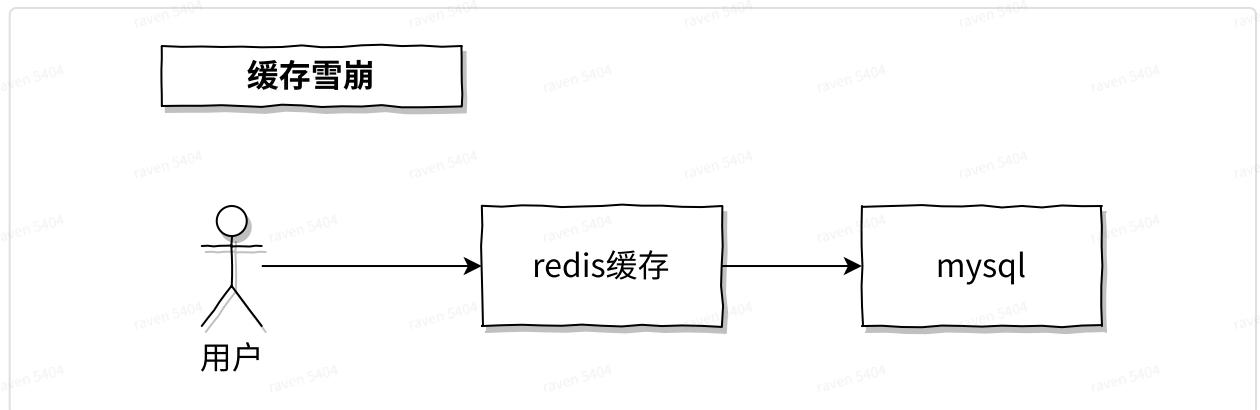
- 解决方案「事中」

- 互斥锁

- 在缓存失效后，通过互斥锁或者队列来控制读数据写缓存的线程数量，比如某个key只允许一个线程查询数据和写缓存，其他线程等待。这种方式会阻塞其他的线程，此时系统的吞吐量会下降
  - 使用熔断机制，限流降级。当流量达到一定的阈值，直接返回“系统拥挤”之类的提示，防止过多的请求打在数据库上将数据库击垮，至少能保证一部分用户是可以正常使用，其他用户多刷新几次也能得到结果。

- 解决方案「事后」

- 开启Redis持久化机制，尽快恢复缓存数据，一旦重启，就能从磁盘上自动加载数据恢复内存中的数据。



## 2. 缓存击穿

**某个热点key失效**，高并发的情况下对其请求，造成大量的请求读取不到缓存数据，从而去访问数据库，这样数据库压力剧增。

## • 解决方案

- 互斥或者队列控制读数据写缓存的线程数量, key只允许一个线程查询数据和写缓存, 其他线程等待. 这种方式会阻塞其他的线程, 此时系统的吞吐量会下降.

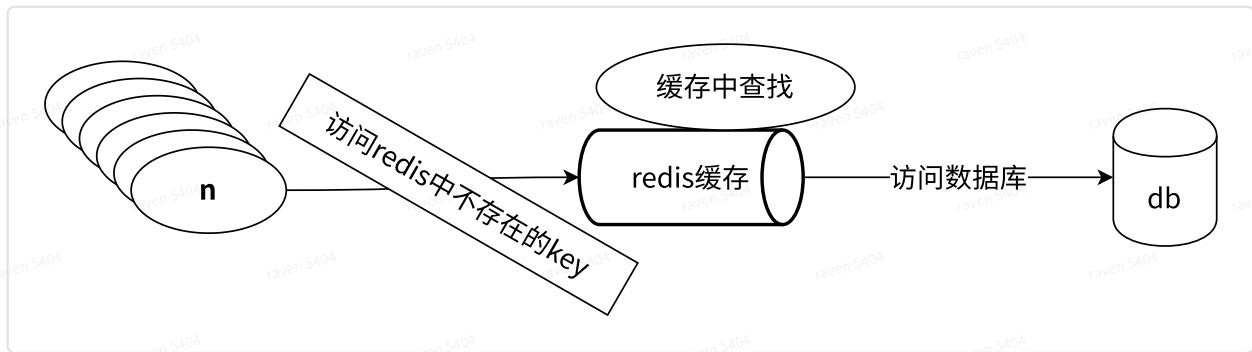
## • 热点数据缓存永远不过期

- 物理不过期, 针对热点key不设置过期时间
- 逻辑过期, 把过期时间存在key对应的value里, 如果发现要过期了, 通过一个后台的异步线程进行缓存的构建

## 3. 缓存穿透

缓存穿透是指用户请求的数据在缓存中不存在即没有命中, 同时在数据库中也不存在, 导致用户每次请求该数据都要去数据库中查询一遍。如果有恶意攻击者不断请求系统中不存在的数据, 会导致短时间大量请求落在数据库上, 造成数据库压力过大, 甚至导致数据库承受不住而宕机崩溃

缓存穿透的关键在于在Redis中查不到key值, 它和缓存击穿的根本区别在于传进来的key在Redis中是不存在的.



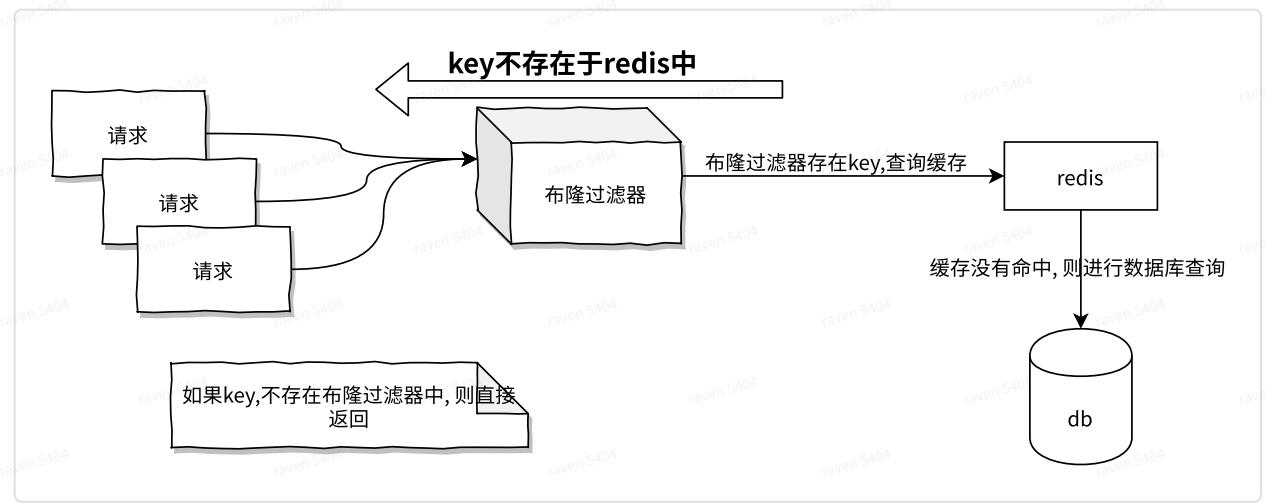
## • 解决方案

- 将无效的key存放进Redis中.

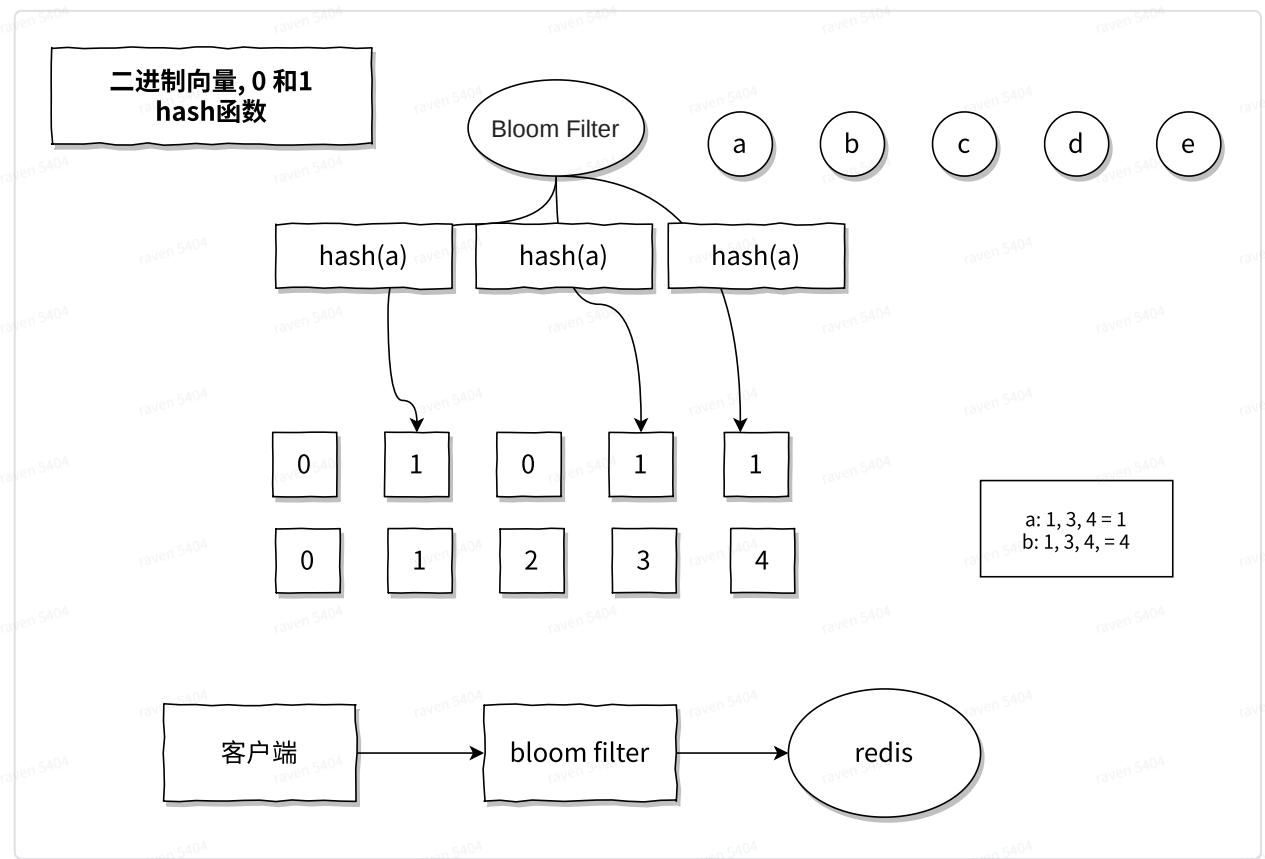
- 当出现Redis查不到数据, 数据库也查不到数据的情况, 我们就把这个key保存到Redis中, 设置value="null", 并设置其过期时间极短, 后面再出现查询这个key的请求的时候, 直接返回null, 就不需要再查询数据库了。但这种处理方式是有问题的, 假如传进来的这个不存在的Key值每次都是随机的, 那存进Redis也没有意义。

## ◦ 使用布隆过滤器

- 如果布隆过滤器判定某个key不存在布隆过滤器中, 那么就一定不存在, 如果判定某个key存在, 那么很大可能是存在(存在一定的误判率)。于是我们可以在缓存之前再加一个布隆过滤器, 将数据库中的所有key都存储在布隆过滤器中, 在查询Redis前先去布隆过滤器查询key是否存在, 如果不存在就直接返回, 不让其访问数据库, 从而避免了对底层存储系统的查询压力。



- 布隆过滤器「Bloom Filter」



Java

```

1 <!-- https://mvnrepository.com/artifact/com.google.guava -->
2 <dependency>
3   <groupId>com.google.guava</groupId>
4   <artifactId>guava</artifactId>
5   <version>31.0.1-jre</version>
6 </dependency>

```

- 推荐使用: guava

## Java

```
1 public class Test {  
2     public static void main(String[] args) {  
3         final int size = 10000;  
4         final float positivesRate = 0.01F;  
5         int count = 0;  
6         // 创建一个bloom过滤器  
7         BloomFilter<Integer> bloomFilter =  
8             BloomFilter.create(Funnels.integerFunnel(), size, positivesRate);  
9  
10        for (int i = 0; i < size; i++) {  
11            bloomFilter.put(i);  
12        }  
13  
14        for (int i = size; i < size + 1000; i++) {  
15            if (bloomFilter.mightContain(i)) {  
16                count++;  
17                System.out.println(i + " --> " + "被误判了!");  
18            }  
19        }  
20        System.out.println("一共误判了: " + count);  
21    }  
22}
```

<https://www.hutool.cn/>

## 第十一章: ssm与redis集成

集成到Spring到中。

以前用的是命令行, 现在想通过java代码来操作Redis.----> Jedis 「方法跟命令行可以对得上」

版本需求:

spring-context: 5.3.4

连接: @bean

## XML

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <project xmlns="http://maven.apache.org/POM/4.0.0"  
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/  
5   xsd/maven-4.0.0.xsd">
```

```
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>com.example</groupId>
8      <artifactId>java-14-redis-spring</artifactId>
9      <version>1.0-SNAPSHOT</version>
10     <name>java-14-redis-spring</name>
11     <packaging>war</packaging>
12
13     <properties>
14         <maven.compiler.target>1.8</maven.compiler.target>
15         <maven.compiler.source>1.8</maven.compiler.source>
16         <junit.version>5.7.0</junit.version>
17         <spring.version>5.3.4</spring.version>
18         <junit4.version>4.13</junit4.version>
19         <jdeis.version>3.5.1</jdeis.version>
20         <spring.redis.version>2.4.5</spring.redis.version>
21     </properties>
22
23     <dependencies>
24         <dependency>
25             <groupId>javax.servlet</groupId>
26             <artifactId>javax.servlet-api</artifactId>
27             <version>4.0.1</version>
28             <scope>provided</scope>
29         </dependency>
30         <dependency>
31             <groupId>org.junit.jupiter</groupId>
32             <artifactId>junit-jupiter-api</artifactId>
33             <version>${junit.version}</version>
34             <scope>test</scope>
35         </dependency>
36         <dependency>
37             <groupId>org.junit.jupiter</groupId>
38             <artifactId>junit-jupiter-engine</artifactId>
39             <version>${junit.version}</version>
40             <scope>test</scope>
41         </dependency>
42
43         <dependency>
44             <groupId>org.springframework</groupId>
45             <artifactId>spring-context</artifactId>
46             <version>${spring.version}</version>
47         </dependency>
48
49         <dependency>
50             <groupId>junit</groupId>
51             <artifactId>junit</artifactId>
52             <version>${junit4.version}</version>
```

```

53             <scope>test</scope>
54         </dependency>
55
56     <dependency>
57         <groupId>org.springframework</groupId>
58         <artifactId>spring-test</artifactId>
59         <version>${spring.version}</version>
60     </dependency>
61
62     <dependency>
63         <groupId>redis.clients</groupId>
64         <artifactId>jedis</artifactId>
65         <version>${jedis.version}</version>
66     </dependency>
67
68     <dependency>
69         <groupId>org.springframework.data</groupId>
70         <artifactId>spring-data-redis</artifactId>
71         <version>${spring.redis.version}</version>
72     </dependency>
73 </dependencies>
74
75 <build>
76     <plugins>
77         <plugin>
78             <groupId>org.apache.maven.plugins</groupId>
79             <artifactId>maven-war-plugin</artifactId>
80             <version>3.3.0</version>
81         </plugin>
82     </plugins>
83 </build>
84 </project>

```

- 新建一个spring工程.
- 引入相关坐标
- 配置相关参数
  - 连接池
  - 连接信息「host, port, password」
  - 配置RedisTemplate
  - 测试

## 1. xml方式配置

### 2. 采用全注解方式配置.

#### a. 配置类: Configuration

- b. 开启扫描: @ComponentScan
- c. 引入其它配置: @Import
- d. 引入properties: @PropertyResource()
- e. 单元测试
  - i. @RunWith
  - ii. ContextConfiguration()

### 3. xml配置

- a. 见工程代码

### 4. 作业

- 哨兵, 主从配置.
- 哨兵和主从配置使用java进行创建与连接
- ssm
- ssm无法识别redis.properties文件的解决方案
  - 在其它properties配置文件当中添加属性: ignore-unresolvable="true"

#### Java

```
1
2 <context:property-placeholder location="classpath:jdbc.properties" ignore-
  unresolvable="true" />
```

## 第十二章: Pipelined 「自己看看」

## 第十三章: 底层原理

### 1. 环境搭建

#### Bash

```
1 yum install -y man man-pages
```

### 2. 必要的命令

#### Bash

```
1 cd /proc/pid/fd # 查看某个进程的文件描述符
```

### 3. 查看指定的key的编码

## Bash

```
1 object encoding key
```

## 4. 缓存淘汰策略

- 放内存了.过期设置一下
  - redis缓存淘汰策略
  - lru
  - Lfu
- 彻底弄懂Redis的内存淘汰策略 - 知乎 (zhihu.com)

## 5. 查看文档

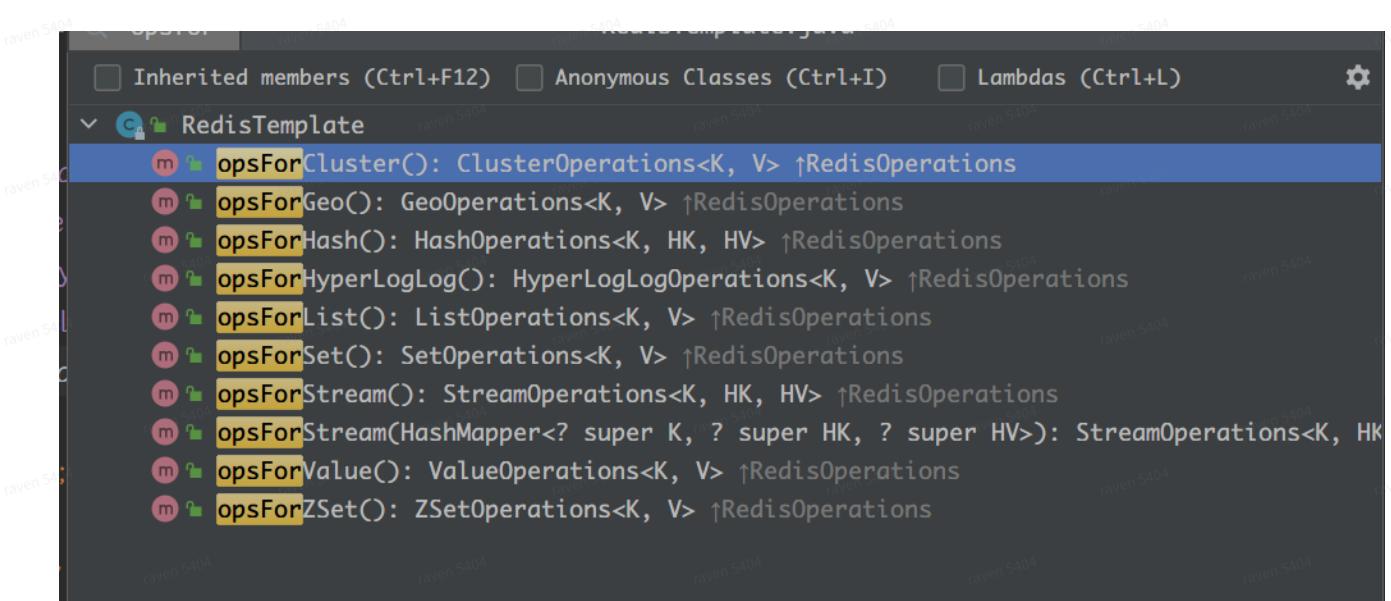
## Bash

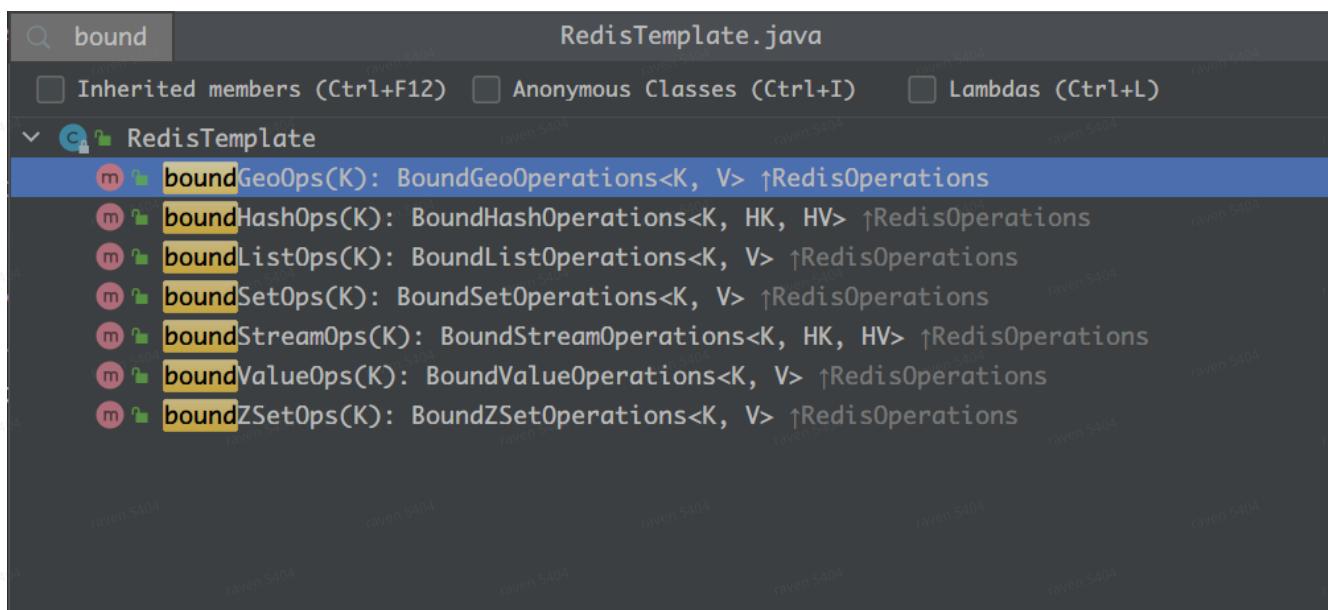
```
1 # redis-cli中查看: 输入? 或者help可以打开帮助文档界面;
2 * help @group
3   查看group的一些列命令说明, group为操作的分组, 比如list为一组操作, hash为一组操作
4 * help 具体命令
5   查看命令详情
6 * help @命令
```

## • 总结

- 数据类型
  - string类型的实现细节. sds.
  - list
  - set
  - hash
  - zset
    - 特征
    - 使用场景
    - 常用的一些命令
  - bitmap
    - 用处非常的大.
  - 地理空间
  - XXX
- 单机模型
  - 缺点及解决方案

- akf
- cap
- base
- 主从/哨兵
  - 主从, 选择, 脑裂, 网络分区
- 客户端
  - akf, y轴
  - 哈希取模
  - Random
  - 一致性哈希
- 服务器
  - 代理
    - **twemproxy**
    - **predixy**
  - 路由「预分区」
    - 集群: cluster.hash slot , 16384
- zset/set, 底层实现跳表
- redis缓存淘汰策略
- 布隆过滤器
- akf, base, cap
- 后期说
  - spring boot的redis集成;
    - 一些应用场景
      - 排行榜
      - 分布式锁





- **StringRedisTemplate**