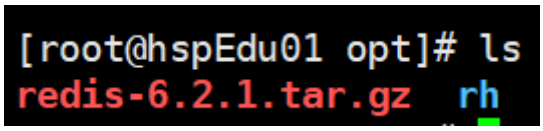
1、redis概述和安装

1.1、安装redis

1. 下载redis

```
https://download.redis.io/releases/
```

2. 将`redis`安装包拷贝到`/opt/`目录



3. 解压

```
tar -zvxf redis-6.2.1.tar.gz
```

4. 安装gcc

```
yum install gcc
```

5. 进入目录

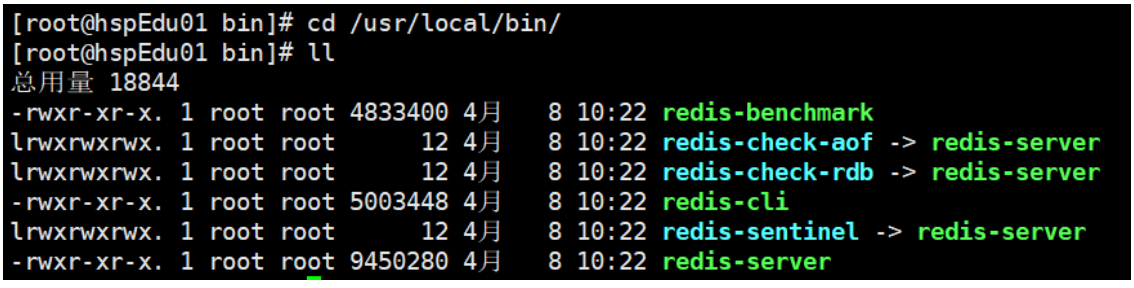
```
cd redis-6.2.1
```

6. 编译

```
make
```

7. 执行`make install`进行安装

8. 查看安装目录：/usr/local/bin



目录介绍

- redis-benchmark：性能测试工具，可以在自己本子允许，看看自己本子性能如何
- redis-check-aof：修复有问题的AOF文件，rdb和aof后面讲
- redis-check-dump：修复有问题的dump.rdb文件
- redis-sentinel：redis集群使用
- redis-server：redis服务器启动命令
- redis-clit：客户端，操作入口

1.2、启动redis

方式1：前台启动（不推荐）

执行`redis-server`命令，这种如果关闭启动窗口，则redis会停止。

```
[root@hspEdu01 bin]# redis-server
58571:C 08 Apr 2022 10:31:46.312 # oooOoooOoooo Redis is starting oooOoooOoooo
58571:C 08 Apr 2022 10:31:46.312 # Redis version=6.2.1, bits=64, commit=00000000, modified=0, pid=58571, just started
58571:C 08 Apr 2022 10:31:46.312 # Warning: no config file specified, using the default config. In order to specify a co
ver /path/to/redis.conf
58571:M 08 Apr 2022 10:31:46.314 * Increased maximum number of open files to 10032 (it was originally set to 1024).
58571:M 08 Apr 2022 10:31:46.314 * monotonic clock: POSIX clock_gettime
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 6.2.1 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 58571
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'
```

方式2：后端启动（推荐）

后台方式启动后，关闭窗口后，redis不会被停止.

步骤如下

1. 复制redis.conf文件到/etc目录

```
cp /opt/redis-6.2.1/redis.conf /etc
```

2. 使用vi命令修改/etc/redis.config中的配置，将后台启动设置daemonize改为yes，如下

```
daemonize yes
```

3. 启动redis

```
redis-server /etc/redis.conf
```

4. 查看redis进程

```
[root@hspEdu01 bin]# ps -ef |grep redis
root      59388      1  0 10:46 ?        00:00:00 redis-server 127.0.0.1:6379
root      59429  53838  0 10:47 pts/1    00:00:00 grep --color=auto redis
```

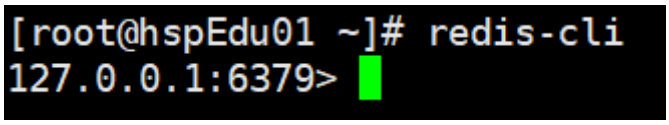
1.3、关闭redis

方式1：kill -9 pid

方式2：redis-cli shutdown

1.4、进入redis命令窗口

执行`redis-cli`即可进入redis命令窗口，然后就可以执行redis命令了。



1.5、redis命令大全

```
http://doc.redisfans.com/
```

Redis 命令参考

本文档是 Redis Command Reference 和 Redis Documentation 的中文翻译版： 所有 Redis 命令文档均已翻译完毕， Redis 最重要的一部分主题（topic）文档， 比如事务、持久化、复制、Sentinel、集群等文章也已翻译完毕。

文档目前描述的内容以 Redis 2.8 版本为准， 查看更新日志(change log)可以了解本文档对 Redis 2.8 所做的更新。

你可以通过网址 doc.redisfans.com 在线阅览本文档， 也可以下载 PDF 格式 或者 HTML 格式 的离线版本。

命令目录(使用 CTRL + F 快速查找):

- Key（键）
  - DEL
  - DUMP
  - EXISTS
  - EXPIRE
  - EXPIREAT
  - KEYS
  - MIGRATE
  - MOVE
  - OBJECT
- String（字符串）
  - APPEND
  - BITCOUNT
  - BITOP
  - DECR
  - DECRBY
  - GET
  - GETBIT
  - GETRANGE
  - GETSET
- Hash（哈希表）
  - HDEL
  - HEXISTS
  - HGET
  - HGETALL
  - HINCRBY
  - HINCRBYFLOAT
  - HKEYS
  - HLEN
  - HMGET
- List（列表）
  - BLPOP
  - BRPOP
  - BRPOPLPUSH
  - LINDEX
  - LINSERT
  - LLEN
  - LPOP
  - LPUSH
  - LPUSHX

1.6、redis介绍相关知识

- 默认端口6379
- 默认16个数据库，类似数组的下标从0开始，初始默认使用0号库
- 使用select <dbid>来切换数据库，如：select 1，切换到第2个库
- 统一密码管理，所有的库密码相同
- dbsize：查看当前数据库的key的数量
- flushdb：情况当前库
- flushall：清空全部库

redis是单线程+多路IO复用技术。

多路复用是指使用一个线程来检测多个文件描述符（socket）的就绪状态，比如调用select和poll函数，传入多个文件描述符，如果有一个文件描述符就绪，则返回，否则阻塞到超时，得到就绪状态后进行真正的操作可以在同一个线程里执行，也可以启动线程执行（比如使用线程池）

串行 vs 多线程+锁（memcached） vs 单线程+多路复用（redis）

（与memcache三不同：支持多数据类型，支持持久化，单线程+多路复用）

redis6.0中提供了多线程，命令解析和io数据读写这部分采用了多线程，而命令的执行还是采用的是单线程，多个客户端发送来的命令会在同一个线程去执行，相当于排队执行，效率极高。

2、redis 5大数据类型

这里说的数据类型是value的数据类型，key的类型都是字符串。

5种数据类型：

- redis字符串（String）
- redis列表（List）
- redis集合（Set）
- redis哈希表（Hash）
- redis有序集合（Zset）

哪里去获取redis常用数据类型操作命令：http://redis.cn/commands.html

2.1、redis键（key）

- keys *：查看当前库所有的key
- exists key：判断某个key是否存在
- type key：查看你的key是什么类型
- del key：删除指定的key数据
- unlink key：根据value删除非阻塞删除，仅仅将keys从keyspace元数据中删除，真正的删除会在后续异步中操作。
- expire key 10：为指定的key设置有效期10秒
- ttl key：查看指定的key还有多少秒过期，-1：表示永不过期，-2：表示已过期
- select dbindex：切换数据库【0-15】，默认为0
- dbsize：查看当前数据库key的数量
- flushdb：清空当前库
- flushall：通杀全部库
-

2.2、redis字符串（String）

2.2.1、简介

String是Redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。

String类型是二进制安全的。意味着Redis的string可以包含任何数据。比如jpg图片或者序列化的对象。

String类型是Redis最基本的数据类型，一个Redis中字符串value最多可以是512M

2.2.2、常用命令

set：添加键值对

```
127.0.0.1:6379> set key value [EX seconds|PX milliseconds|EXAT timestamp|PXAT
milliseconds-timestamp|KEEPTTL] [NX|XX] [GET]
```

- NX：当数据库中key不存在时，可以将key-value添加到数据库
- XX：当数据库中key存在时，可以将key-value添加数据库，与NX参数互斥
- EX：key的超时秒数

- PX：key的超时毫秒数，与EX互斥
- value中若包含空格、特殊字符，需用双引号包裹

get：获取值

```
get <key>
```

示例

```
127.0.0.1:6379> set name ready
OK
127.0.0.1:6379> get name
"ready"
```

apend：追价值

```
append <key> <value>
```

> 将给定的value追加到原值的末尾。

示例

```
127.0.0.1:6379> set k1 hello
OK
127.0.0.1:6379> append k1 " world"
(integer) 11
127.0.0.1:6379> get k1
"hello world"
```

strlen：获取值的长度

```
strlen <key>
```

示例

```
127.0.0.1:6379> set name ready
OK
127.0.0.1:6379> strlen name
(integer) 5
```

setnx：key不存在时，设置key的值

```
setnx <key> <value>
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> setnx site "itsoku.com" #site不存在，返回1，表示设置成功
(integer) 1
127.0.0.1:6379> setnx site "itsoku.com" #再次通过setnx设置site，由于已经存在了，所以设置失败，返回0
(integer) 0
```

incr：原子递增1

```
incr <key>
```

> 将key中存储的值增1，只能对数字值操作，如果key不存在，则会新建一个，值为1

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> set age 30 #age值为30
OK
127.0.0.1:6379> incr age #age增加1，返回31
(integer) 31
127.0.0.1:6379> get age #获取age的值
"31"
127.0.0.1:6379> incr salary #salary不存在，自动创建一个，值为1
(integer) 1
127.0.0.1:6379> get salary #获取salary的值
"1"
```

decr：原子递减1

```
decr <key>
```

> 将key中存储的值减1，只能对数字值操作，如果为空，新增值为-1

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> set age 30 #age值为30
OK
127.0.0.1:6379> decr age #age递减1，返回29
(integer) 29
127.0.0.1:6379> get age #获取age的值
"29"
127.0.0.1:6379> decr salary #salary不存在，自动创建一个，值为-1
(integer) -1
127.0.0.1:6379> get salary #获取salary
"-1"
```

incrby/decrby：递增或者递减指定的数字

```
incrby/decrby <key> <步长>
```

> 将key中存储的数字值递增指定的步长，若key不存在，则相当于在原值为0的值上递增指定的步长。

示例

```
127.0.0.1:6379> set salary 10000 #设置salary为10000
OK
127.0.0.1:6379> incrby salary 5000 #salary添加5000，返回15000
(integer) 15000
127.0.0.1:6379> get salary #获取salary
"15000"
127.0.0.1:6379> decrby salary 800 #salary减去800，返回14200
(integer) 14200
127.0.0.1:6379> get salary #获取salary
"14200"
```

mset：同时设置多个key-value

```
mset <key1> <value1> <key2> <value2> ...
```

示例

```
127.0.0.1:6379> mset name ready age 30
OK
127.0.0.1:6379> get name
"ready"
127.0.0.1:6379> get age
"30"
```

mget：获取多个key对应的值

```
mget <key1> <key2> ...
```

示例

```
127.0.0.1:6379> mset name ready age 30 #同时设置name和age
OK
127.0.0.1:6379> mget name age #同时读取name和age的值
1) "ready"
2) "30"
```

msetnx：当多个key都不存在时，则设置成功

```
msetnx <key1> <value1> <key2> <value2> ...
```

> 原子性的，要么都成功，或者都失败。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> set k1 v1 #设置k1
OK
127.0.0.1:6379> msetnx k1 v1 k2 v2 #当k1和k2都不存在的时候，同时设置k1和k2，由于k1已存在，所以这个操作失败
(integer) 0
127.0.0.1:6379> mget k1 k2 #获取k1、k2，k2不存在
1) "v1"
2) (nil)
127.0.0.1:6379> msetnx k2 v2 k3 v3 #当k2、h3都不存在的时候，同时设置k2和k3，设置成功
(integer) 1
127.0.0.1:6379> mget k1 k2 k3 #后去k1、k2、k3的值
1) "v1"
2) "v2"
3) "v3"
```

getrange：获取值的范围，类似java中的substring

```
getrange key start end
```

> 获取[start,end]返回为的字符串

示例

```
127.0.0.1:6379> set k1 helloworld
OK
127.0.0.1:6379> getrange k1 0 4
"hello"
```

setrange：覆盖指定位置的值

```
setrange <key> <起始位置> <value>
```

示例

```
127.0.0.1:6379> set k1 helloworld
OK
127.0.0.1:6379> get k1
"helloworld"
127.0.0.1:6379> setrange k1 1 java
(integer) 10
127.0.0.1:6379> get k1
"hjavaworld"
```

setex：设置键值&过期时间（秒）

```
setex <key> <过期时间(秒)> <value>
```

示例

```
127.0.0.1:6379> setex k1 100 v1 #设置k1的值为v1，有效期100秒
OK
127.0.0.1:6379> get k1 #获取k1的值
"v1"
127.0.0.1:6379> ttl k1 #获取k1还有多少秒失效
(integer) 96
```

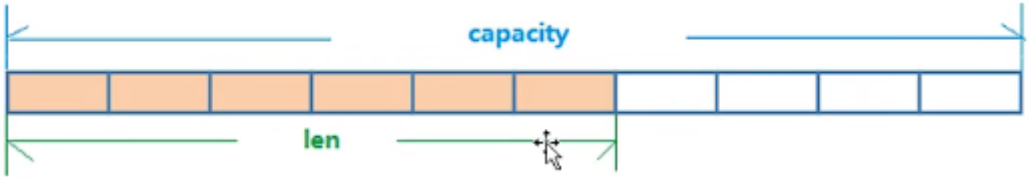
getset：以新换旧，设置新值同时返回旧值

```
getset <key> <value>
```

```
127.0.0.1:6379> set name ready #设置name为ready
OK
127.0.0.1:6379> getset name tom #设置name为tom，返回name的旧值
"ready"
127.0.0.1:6379> getset age 30 #设置age为30，age未设置过，返回age的旧值为null
(nil)
```

2.2.3、数据结构

String的数据结构为简单动态字符串（Simple Dynamic String，缩写SDS）。是可以修改的字符串，内部结构上类似于Java的ArrayList，采用分配冗余空间的方式来减少内存的频繁分配。



如图所示，内部为当前字符串实际分配的空间capacity一般要高于实际字符串长度len。当字符串长度小于1M时，扩容都是加倍现有的空间，如果超过1M，扩容时一次会多扩容1M的空间。

要注意的是字符串最大长度为512M。

2.3、redis列表（List）

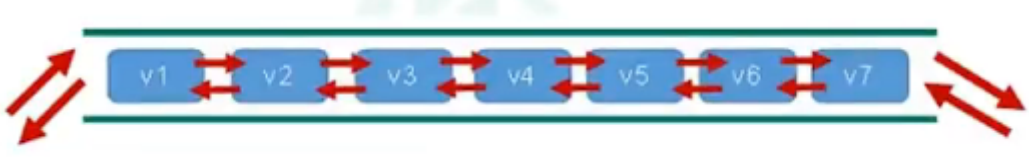
2.3.1、简介

单键多值

redis列表是简单的字符串列表，按照插入顺序排序。

你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

它的底层实际上是使用双向链表实现的，对两端的操作性能很高，通过索引下标操作中间节点性能会较差。



2.3.2、常用命令

lpush/rpush：从左边或者右边插入一个或多个值

```
lpush/rpush <key1> <value1> <key2> <value2> ...
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> rpush name java spring "springboot" "spring cloud" #列表name的左边插入4个元素
(integer) 4
127.0.0.1:6379> lrange name 1 2 #从左边取出索引位于[1,2]范围内的元素
1) "spring"
2) "springboot"
```

lrange：从列表左边获取指定范围内的值

```
lrange <key> <star> <stop>
```

> 返回列表 `key` 中指定区间内的元素，区间以偏移量 `start` 和 `stop` 指定。
>
> 下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。
>
> 你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。
>
> **返回值:**
>
> 一个列表，包含指定区间内的元素。

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> rpush course java c c++ php js nodejs #course集合的右边插入6个元素
(integer) 6
127.0.0.1:6379> lrange course 0 -1 #取出course集合中所有元素
1) "java"
2) "c"
3) "c++"
4) "php"
5) "js"
6) "nodejs"
127.0.0.1:6379> lrange course 1 3 #获取course集合索引[1,3]范围内的元素
1) "c"
2) "c++"
3) "php"
```

lpop/rpop：从左边或者右边弹出多个元素

```
lpop/rpop <key> <count>
```

> count：可以省略，默认值为1
>
> lpop/rpop 操作之后，弹出来的值会从列表中删除
>
> 值在键在，值光键亡。

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> rpush course java c++ php js node js #集合course右边加入6个元素
(integer) 6
127.0.0.1:6379> lpop course #从左边弹出1个元素
"java"
127.0.0.1:6379> rpop course 2 #从右边弹出2个元素
1) "js"
2) "node"
```

rpoplpush：从一个列表右边弹出一个元素放到另外一个列表中

```
rpoplpush source destination
```

> 从source的右边弹出一个元素放到destination列表的左边

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> rpush k1 1 2 3 #列表k1的右边添加3个元素[1,2,3]
(integer) 3
127.0.0.1:6379> lrange k1 0 -1 #从左到右输出k1列表中的元素
1) "1"
2) "2"
3) "3"
127.0.0.1:6379> rpush k2 4 5 6 #列表k2的右边添加3个元素[4,5,6]
(integer) 3
127.0.0.1:6379> lrange k2 0 -1 #从左到右输出k2列表中的元素
1) "4"
2) "5"
3) "6"
127.0.0.1:6379> rpoplpush k1 k2 #从k1的右边弹出一个元素放到k2的左边
"3"
127.0.0.1:6379> lrange k1 0 -1 #k1中剩下2个元素了
1) "1"
2) "2"
127.0.0.1:6379> lrange k2 0 -1 #k2中变成4个元素了
1) "3"
2) "4"
3) "5"
4) "6"
```

lindex：获取指定索引位置的元素（从左到右）

```
lindex key index
```

> 返回列表 `key` 中，下标为 `index` 的元素。
>
> 下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。
>
> 你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。
>
> 如果 `key` 不是列表类型，返回一个错误。
>
> **返回值:**
>
> 列表中下标为 `index` 的元素。
>
> 如果 `index` 参数的值不在列表的区间范围内(out of range)，返回 `nil`

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> rpush course java c c++ php #列表course中放入4个元素
(integer) 4
127.0.0.1:6379> lindex course 2 #返回索引位置2的元素
"c++"
127.0.0.1:6379> lindex course 200 #返回索引位置200的元素，没有
(nil)
127.0.0.1:6379> lindex course -1 #返回最后一个元素
"php"
```

llen：获得列表长度

```
llen key
```

> 返回列表 `key` 的长度。
>
> 如果 `key` 不存在，则 `key` 被解释为一个空列表，返回 `0` .
>
> 如果 `key` 不是列表类型，返回一个错误。

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> rpush name ready tom jack
(integer) 3
127.0.0.1:6379> llen name
(integer) 3
```

linsert：在某个值的前或者后面插入一个值

```
linsert <key> before|after <value> <newvalue>
```

> 将值 `newvalue` 插入到列表 `key` 当中，位于值 `value` 之前或之后。
>
> 当 `value` 不存在于列表 `key` 时，不执行任何操作。
>
> 当 `key` 不存在时， `key` 被视为空列表，不执行任何操作。
>
> 如果 `key` 不是列表类型，返回一个错误。
>
> **返回值:**
>
> 如果命令执行成功，返回插入操作完成之后，列表的长度。
>
> 如果没有找到 `value` ，返回 `-1` 。
>
> 如果 `key` 不存在或为空列表，返回 `0` 。

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> rpush name ready tom jack #列表name中添加3个元素
(integer) 3
127.0.0.1:6379> lrange name 0 -1 #name列表所有元素
1) "ready"
2) "tom"
3) "jack"
127.0.0.1:6379> linsert name before tom lily #tom前面添加lily
(integer) 4
127.0.0.1:6379> lrange name 0 -1 #name列表所有元素
1) "ready"
2) "lily"
3) "tom"
4) "jack"
127.0.0.1:6379> linsert name before xxx lucy # 在元素xxx前面插入lucy，由于xxx元素不存在，插入失败，返回-1
(integer) -1
127.0.0.1:6379> lrange name 0 -1
1) "ready"
2) "lily"
3) "tom"
4) "jack"
```

lrem：删除指定数量的某个元素

```
LREM key count value
```

> 根据参数 `count` 的值，移除列表中与参数 `value` 相等的元素。
>
> `count` 的值可以是以下几种：
>
> - `count > 0` : 从表头开始向表尾搜索，移除与 `value` 相等的元素，数量为 `count` 。
> - `count < 0` : 从表尾开始向表头搜索，移除与 `value` 相等的元素，数量为 `count` 的绝对值。

> - `count = 0` : 移除表中所有与 `value` 相等的值。
>
> **返回值:**
>
> 被移除元素的数量。
>
> 因为不存在的 `key` 被视作空表(empty list)，所以当 `key` 不存在时，总是返回 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> rpush k1 v1 v2 v3 v2 v2 v1 #k1列表中插入6个元素
(integer) 6
127.0.0.1:6379> lrange k1 0 -1 #输出k1集合中所有元素
1) "v1"
2) "v2"
3) "v3"
4) "v2"
5) "v2"
6) "v1"
127.0.0.1:6379> lrem k1 2 v2 #k1集合中从左边删除2个v2
(integer) 2
127.0.0.1:6379> lrange k1 0 -1 #输出列表，列表中还有1个v2，前面2个v2干掉了
1) "v1"
2) "v3"
3) "v2"
4) "v1"
```

lset：替换指定位置的值

```
lset <key> <index> <value>
```

> 将列表 `key` 下标为 `index` 的元素的值设置为 `value` 。
>
> 当 `index` 参数超出范围，或对一个空列表(`key` 不存在)进行lset时，返回一个错误。
>
> **返回值:**
>
> 操作成功返回 `ok` ，否则返回错误信息。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> rpush name tom jack ready #name集合中放入3个元素
(integer) 3
127.0.0.1:6379> lrange name 0 -1 #输出name集合元素
1) "tom"
2) "jack"
3) "ready"
127.0.0.1:6379> lset name 1 lily #将name集合中第2个元素替换为liy
OK
127.0.0.1:6379> lrange name 0 -1 #输出name集合元素
1) "tom"
2) "lily"
3) "ready"
127.0.0.1:6379> lset name 10 lily #索引超出范围，报错
```

```
(error) ERR index out of range
127.0.0.1:6379> lset course 1 java #course集合不存在，报错
(error) ERR no such key
```

2.3.4、数据结构

List的数据结构为快速链表quickList

首先在列表元素较少的情况下会使用一块连续的内存存储，这个结构是ziplist，也就是压缩列表。

它将所有的元素紧挨着一起存储，分配的是一块连续的内存。

当就比较多的时候才会改成quickList。

因为普通的链表需要的附加指针空间太大，会比较浪费空间，比如这个列表里存储的只是int类型的书，结构上还需要2个额外的指针prev和next。

redis将链表和ziplist结合起来组成了quicklist。也就是将多个ziplist使用双向指针串起来使用，这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

2.4、redis集合（Set）

2.4.1、简介

redis set对外提供的功与list类似，是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择。

redis的set是string类型的**无序集合**，他的底层实际是一个value为null的hash表，收益添加，删除，查找复杂度都是O(1)。

一个算法，如果时间复杂度是O(1)，那么随着数据的增加，查找数据的时间不变，也就是不管数据多少，查找时间都是一样的。

2.4.2、常用命令

sadd：添加一个或多个元素

```
sadd <key> <value1> <value2> ...
```

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd k1 v1 v2 v1 v3 v2 #k1中放入5个元素，会自动去重，成功插入3个
(integer) 3
```

smembers：取出所有元素

```
smembers <key>
```

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd k1 v1 v2 v1 v3 v2
(integer) 3
127.0.0.1:6379> smembers k1
1) "v2"
2) "v1"
3) "v3"
```

sismember：判断集合中是否有某个值

```
sismember <key> <value>
```

> 判断集合key中是否包含元素value，1：有，0：没有

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd k1 v1 v2 v1 v3 v2 #k1集合中成功放入3个元素[v1,v2,v3]
(integer) 3
127.0.0.1:6379> sismember k1 v1 #判断k1中是否包含v1，1：有
(integer) 1
127.0.0.1:6379> sismember k1 v5 #判断k1中是否包含v5，0：无
(integer) 0
```

scard：返回集合中元素的个数

```
scard <key>
```

> 返回集合 `key` 的基数(集合中元素的数量)
>
> **返回值：**
>
> 集合的基数。
>
> 当 `key` 不存在时，返回 `0` 。

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd k1 v1 v2 v1 v3 v2
(integer) 3
127.0.0.1:6379> scard k1
(integer) 3
```

srem：删除多个元素

```
srem key member [member ...]
```

> 移除集合 `key` 中的一个或多个 `member` 元素，不存在的 `member` 元素会被忽略。
>
> 当 `key` 不是集合类型，返回一个错误。
>
> **返回值:**
>
> 被成功移除的元素的数量，不包括被忽略的元素。

示例

```
127.0.0.1:6379> flushdb #清空db，方测试
OK
127.0.0.1:6379> sadd course java c c++ python #集合course中添加4个元素
(integer) 4
127.0.0.1:6379> smembers course #获取course集合所有元素
1) "python"
2) "java"
3) "c++"
4) "c"
127.0.0.1:6379> srem course java c #删除course集合中的java和c
(integer) 2
127.0.0.1:6379> smembers course #获取course集合所有元素，剩下2个了
1) "python"
2) "c++"
```

spop：随机弹出多个值

```
spop <key> <count>
```

> 随机从key集合中弹出count个元素，count默认值为1
>
> **返回值:**
>
> 被移除的随机元素。
>
> 当 `key` 不存在或 `key` 是空集时，返回 `nil`

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd course java c c++ python #course集合中添加4个元素
(integer) 4
127.0.0.1:6379> smembers course #获取course集合中所有元素
1) "python"
2) "java"
3) "c++"
4) "c"
127.0.0.1:6379> spop course #随机弹出1个元素，被弹出的元素会被删除
"c++"
127.0.0.1:6379> spop course 2 #随机弹出2个元素
1) "java"
2) "python"
```

```
127.0.0.1:6379> smembers course #输出剩下的元素
1) "c"
```

srandmember：随机获取多个元素，不会从集合中删除

```
srandmember <key> <count>
```

> 从key指定的集合中随机返回count个元素，count可以不指定，默认值是1。
>
> **srandmember 和 spop的区别：**
>
> 都可以随机获取多个元素，srandmember 不会删除元素，而spop会删除元素。
>
> **返回值:**
>
> 只提供 `key` 参数时，返回一个元素；如果集合为空，返回 `nil` 。
>
> 如果提供了 `count` 参数，那么返回一个数组；如果集合为空，返回空数组。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> sadd course java c c++ python #course中放入5个元素
(integer) 4
127.0.0.1:6379> smembers course #输出course集合中所有元素
1) "python"
2) "java"
3) "c++"
4) "c"
127.0.0.1:6379> srandmember course 3 #随机获取3个元素，元素并不会被删除
1) "python"
2) "c++"
3) "c"
127.0.0.1:6379> smembers course #输出course集合中所有元素，元素个数未变
1) "python"
2) "java"
3) "c++"
4) "c"
```

smove：将某个原创从一个集合移动到另一个集合

```
smove <source> <destination> member
```

> 将 `member` 元素从 `source` 集合移动到 `destination` 集合。
>
> smove 是原子性操作。
>
> 如果 `source` 集合不存在或不包含指定的 `member` 元素，则 smove 命令不执行任何操作，仅返回 `0` 。否则， `member` 元素从 `source` 集合中被移除，并添加到 `destination` 集合中去。
>
> 当 `destination` 集合已经包含 `member` 元素时，smove 命令只是简单地将 `source` 集合中的 `member` 元素删除。
>
> 当 `source` 或 `destination` 不是集合类型时，返回一个错误。

> **返回值:**
>
> 如果 `member` 元素被成功移除，返回 `1` 。
>
> 如果 `member` 元素不是 `source` 集合的成员，并且没有任何操作对 `destination` 集合执行，那么返回 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> sadd course1 java php js #集合course1中放入3个元素[java,php,js]
(integer) 3
127.0.0.1:6379> sadd course2 c c++ #集合course2中放入2个元素[c,c++]
(integer) 2
127.0.0.1:6379> smove course1 course2 js #将course1中的js移动到course2
(integer) 1
127.0.0.1:6379> smembers course1 #输出course1中的元素
1) "java"
2) "php"
127.0.0.1:6379> smembers course2 #输出course2中的元素
1) "js"
2) "c++"
3) "c"
```

sinter：取多个集合的交集

```
sinter key [key ...]
```

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd course1 java php js #集合course1：[java,php,js]
(integer) 3
127.0.0.1:6379> sadd course2 c c++ js #集合course2：[c,c++,js]
(integer) 3
127.0.0.1:6379> sadd course3 js html #集合course3：[js,html]
(integer) 2
127.0.0.1:6379> sinter course1 course2 course3 #返回三个集合的交集，只有：[js]
1) "js"
```

sinterstore：将多个集合的交集放到一个新的集合中

```
sinterstore destination key [key ...]
```

> 这个命令类似于`sinter`命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。
>
> **返回值:**
>
> 结果集中的成员数量。

sunion：取多个集合的并集，自动去重

```
sunion key [key ...]
```

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd course1 java php js #集合course1：[java,php,js]
(integer) 3
127.0.0.1:6379> sadd course2 c c++ js #集合course2：[c,c++,js]
(integer) 3
127.0.0.1:6379> sadd course3 js html #集合course3：[js,html]
(integer) 2
127.0.0.1:6379> sunion course1 course2 course3 #返回3个集合的并集，会自动去重
1) "php"
2) "js"
3) "java"
4) "html"
5) "c++"
6) "c"
```

sunionstore：将多个集合的并集放到一个新的集合中

```
sinterstore destination key [key ...]
```

> 这个命令类似于 sunion 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。
>
> **返回值:**
>
> 结果集中的成员数量。

sdiff：取多个集合的差集

```
SDIFF key [key ...]
```

> 返回一个集合的全部成员，该集合是所有给定集合之间的差集。
>
> 不存在的 `key` 被视为空集。

示例

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> sadd course1 java php js #集合course1：[java,php,js]
(integer) 3
127.0.0.1:6379> sadd course2 c c++ js #集合course2：[c,c++,js]
(integer) 3
127.0.0.1:6379> sadd course3 js html #集合course3：[js,html]
(integer) 2
127.0.0.1:6379> sdiff course1 course2 course3 #返回course1中有的而course2和course3中都没有的元素
1) "java"
2) "php"
```

sdiffstore：将多个集合的差集放到一个新的集合中

```
sdiffstore destination key [key ...]
```

> 这个命令类似于 sdiff 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。
>
> **返回值:**
>
> 结果集中的成员数量。

3.4.3、数据结构

set数据结构是字典，字典是用hash表实现的。

Java中的HashSet的内部实现使用HashMap，只不过所有的value都指向同一个对象。

Redis的set结构也是一样的，它的内部也使用hash结构，所有的value都指向同一个内部值。

2.5、redis哈希（Hash）

2.5.1、简介

Redis hash是一个键值对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

类似于java里面的Map<String,Object>

2.5.2、常用命令

hset：设置多个field的值

```
hset key field value [field value ...]
```

> 将哈希表 `key` 中的域 `field` 的值设为 `value` 。
>
> 如果 `key` 不存在，一个新的哈希表被创建并进行 hset 操作。
>
> 如果域 `field` 已经存在于哈希表中，旧值将被覆盖。
>
> **返回值:**
>
> 如果 `field` 是哈希表中的一个新建域，并且值设置成功，返回 `1` 。
>
> 如果哈希表中域 `field` 已经存在且旧值已被新值覆盖，返回 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
```

hget：获取指定filed的值

```
hget key field
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
127.0.0.1:6379> hget user name #获取user中的name
"ready"
```

hgetall：返回hash表所有的域和值

```
hgetall key
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
127.0.0.1:6379> hgetall user #获取user所有信息
1) "name"
2) "ready"
3) "age"
4) "30"
```

hmset：和hset类似（已弃用）

```
hmset key field value [field value ...]
```

hexists：判断给定的field是否存在，1：存在，0：不存在

```
hexists key field
```

> 查看哈希表 `key` 中，给定域 `field` 是否存在。
>
> **返回值：**
>
> 如果哈希表含有给定域，返回 `1` 。
>
> 如果哈希表不含有给定域，或 `key` 不存在，返回 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
127.0.0.1:6379> hexists user name #user中存在name域
(integer) 1
127.0.0.1:6379> hexists user address #user中不存在address域，返回0
(integer) 0
127.0.0.1:6379> hexists user1 address #user1这个key不存在，返回0
(integer) 0
```

hkeys：列出所有的filed

```
hkeys key
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
127.0.0.1:6379> hkeys user #获取user中的所有filed
1) "name"
2) "age"
```

hvals：列出所有的value

```
hvals key
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
127.0.0.1:6379> hvals user #获取user中的所有filed的值列表
1) "ready"
2) "30"
```

hlen：返回filed的数量

```
HLEN key
```

> 返回哈希表 `key` 中域的数量。
>
> **返回值：**

> 哈希表中域的数量。
>
> 当 `key` 不存在时，返回 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #哈希表user中设置2个域：name和age，name的值为ready，age的值为30
(integer) 2
127.0.0.1:6379> hlen user
(integer) 2
```

hincrby：filed的值加上指定的增量

```
hincrby key field increment
```

> 为哈希表 key 中的域 field 的值加上增量 increment 。
>
> 增量也可以为负数，相当于对给定域进行减法操作。
>
> 如果 key 不存在，一个新的哈希表被创建并执行 HINCRBY 命令。
>
> 如果域 field 不存在，那么在执行命令前，域的值被初始化为 0 。
>
> 对一个储存字符串值的域 field 执行 HINCRBY 命令将造成一个错误。
>
> **返回值：**
>
> 执行 hincrby 命令之后，哈希表 `key` 中域 `field` 的值。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset siteInfo site itsoku.com pv 1000 #hash表siteInfo中有2个域：{site:"itsoku.com",pv:1000}
(integer) 2
127.0.0.1:6379> hget siteInfo pv #获取siteInfo中pv的值
"1000"
127.0.0.1:6379> hincrby siteInfo pv 10 #siteInfo中的pv值增加10
(integer) 1010
127.0.0.1:6379> hget siteInfo pv #获取siteInfo中的pv
"1010"
127.0.0.1:6379> hincrby siteInfo uv 500 #siteInfo中的uv值增加500，uv这个域不存在，则会先添加，然后再执行hincrby
(integer) 500
```

hsetnx：当filed不存在的时候，设置filed的值

```
hsetnx key field value
```

> 将哈希表 `key` 中的域 `field` 的值设置为 `value` ，当且仅当域 `field` 不存在。
>
> 若域 `field` 已经存在，该操作无效。
>
> 如果 `key` 不存在，一个新哈希表被创建并执行 hsetnx 命令。
>
> **返回值**:
>
> 设置成功，返回 `1` 。
>
> 如果给定域已经存在且没有操作被执行，返回 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> hset user name ready age 30 #创建user，包含2个域：name、age
(integer) 2
127.0.0.1:6379> hsetnx user name tom #name已存在，设置失败，返回0
(integer) 0
127.0.0.1:6379> hget user name #name依旧是ready
"ready"
127.0.0.1:6379> hsetnx user address shanghai #address不存在，设置成功
(integer) 1
127.0.0.1:6379> hget user address #输出address的值
"shanghai"
```

2.5.3、数据结构

Hash类型对应的数据结构是2中：ziplist（压缩列表），hashtable（哈希表）。

当field-value长度较短个数较少时，使用ziplist，否则使用hashtable。

2.6、redis有序集合zset（sorted set）

2.6.1、简介

redis有序集合zset与普通集合set非常相似，是一个没有重复元素的字符串集合。

不同之处是有序集合的每个成员都关联了一个评分（score），这个评分（score）被用来按照从最低分到最高分的方式排序集合中的成员。

集合的成员是唯一的，但是评分是可以重复的。

因为元素是有序的，所以你可以很快的根据评分（score）或者次序（position）来获取一个范围的元素。

访问有序集合中的中间元素也是非常快的，因为你能够使用有序集合作为一个没有重复成员你的智能列表。

2.6.2、常用命令

zadd：添加元素

```
zadd <key> <score1> <member1> <score2> <member2> ...
```

> 将一个或多个 `member` 元素及其 `score` 值加入到有序集 `key` 当中。
>
> 如果某个 `member` 已经是有序集的成员，那么更新这个 `member` 的 `score` 值，并通过重新插入这个 `member` 元素，来保证该 `member` 在正确的位置上。
>
> `score` 值可以是整数值或双精度浮点数。
>
> 如果 `key` 不存在，则创建一个空的有序集并执行 zadd 操作。
>
> 当 `key` 存在但不是有序集类型时，返回一个错误。
>
> **返回值:**
>
> 被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topn 100 java 80 c 90 c++ 50 php 70 js #创建名称为topn的zset，添加了5个元素
(integer) 5
```

zrange：score升序，获取指定索引范围的元素

```
zrange key start top [withscores]
```

> - 返回存储在有序集合 `key` 中的指定范围的元素。 返回的元素可以认为是按score从最低到最高排列，如果得分相同，将按字典排序。
> - 下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。
>
>   你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。
> - zrange key 0 -1：可以获取所有元素
> - withscores：让成员和它的 `score` 值一并返回，返回列表以 `value1,score1, ..., valueN,scoreN` 的格式表示
>
> **可用版本:**
>
> >= 1.2.0
>
> **时间复杂度:**
>
> O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。
>
> **返回值:**
>
> 指定区间内，带有 `score` 值(可选)的有序集成员的列表。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topn 100 java 80 c 90 c++ 50 php 70 js #创建名称为topn的zset，添加了5个元素
(integer) 5
127.0.0.1:6379> zrange topn 0 -1 #按score升序，返回topn中所有元素的值
1) "php"
2) "js"
3) "c"
4) "c++"
5) "java"
127.0.0.1:6379> zrange topn 0 -1 withscores #按score升序，返回所有元素的值以及score
 1) "php"
 2) "50"
 3) "js"
 4) "70"
 5) "c"
 6) "80"
 7) "c++"
 8) "90"
 9) "java"
10) "100"
127.0.0.1:6379> zrange topn 2 4 #返回索引范围[2,4]内的3个元素
1) "c"
2) "c++"
3) "java"
```

zrevrange：score降序，获取指定索引范围的元素

```
zrevrange key start stop [WITHSCORES]
```

> - 返回存储在有序集合`key`中的指定范围的元素。 返回的元素可以认为是按score最高到最低排列， 如果得分相同，将按字典排序。
> - 下标参数`start`和`stop`都以`0`为底，也就是说，以`0`表示有序集第一个成员，以`1`表示有序集第二个成员，以此类推。
>
>   你也可以使用负数下标，以`-1`表示最后一个成员， `-2`表示倒数第二个成员，以此类推。
> - withscores：让成员和它的`score`值一并返回，返回列表以`value1,score1, ..., valueN,scoreN`的格式表示

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topn 100 java 80 c 90 c++ 50 php 70 js #创建名称为topn的zset，添加了5个元素
(integer) 5
127.0.0.1:6379> zrevrange topn 0 -1 #按照score降序获取所有元素
1) "java"
2) "c++"
3) "c"
4) "js"
5) "php"
127.0.0.1:6379> zrevrange topn 0 2 #按照score降序获取前3名
1) "java"
```

```
2) "c++"
3) "c"
```

zrangebyscore：按照score升序，返回指定score范围内的数据

```
zrangebyscore key min max [WITHSCORES] [LIMIT offset count]
```

> 返回有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max`)的成员。有序集成员按 `score` 值递增(从小到大)次序排列。
>
> 具有相同 `score` 值的成员按字典序来排列(该属性是有序集提供的，不需要额外的计算)。
>
> 可选的 `LIMIT` 参数指定返回结果的数量及区间(就像SQL中的 `SELECT LIMIT offset, count`)，注意当 `offset` 很大时，定位 `offset` 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间。
>
> 可选的 `WITHSCORES` 参数决定结果集是单单返回有序集的成员，还是将有序集成员及其 `score` 值一起返回。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topn 100 java 80 c 90 c++ 50 php 70 js #创建名称为topn的zset，添加了5个元素
(integer) 5
127.0.0.1:6379> zrangebyscore topn 70 90 #score升序，获取score位于[70,90]区间中的元素值
1) "js"
2) "c"
3) "c++"
127.0.0.1:6379> zrangebyscore topn 70 90 withscores #score升序，获取score位于[70,90]区间中的元素值及score
1) "js"
2) "70"
3) "c"
4) "80"
5) "c++"
6) "90"
127.0.0.1:6379> zrangebyscore topn 70 90 withscores limit 1 2 #相当于：select value,score from topn集合 where score>=70 and score<=90 order by score asc limit 1,2
1) "c"
2) "80"
3) "c++"
4) "90"
```

zrevrangebyscore：按照score降序，返回指定score范围内的数据

```
zrevrangebyscore key max min [WITHSCORES] [LIMIT offset count]
```

> 返回有序集 `key` 中， `score` 值介于 `max` 和 `min` 之间(默认包括等于 `max` 或 `min`)的所有的成员。有序集成员按 `score` 值递减(从大到小)的次序排列。

> 具有相同 `score` 值的成员按字典序的逆序排列。
>
> 除了成员按 `score` 值递减的次序排列这一点外， zrevrangebyscore 命令的其他方面和 zrangebyscore 命令一样。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topn 100 java 80 c 90 c++ 50 php 70 js #创建名称为topn的zset，添加了5个元素
(integer) 5
127.0.0.1:6379> zrevrangebyscore topn 100 90 #score降序，获取score位于[70,90]区间中的元素值
1) "java"
2) "c++"
127.0.0.1:6379> zrevrangebyscore topn 100 90 withscores #score降序，获取score位于[70,90]区间中的元素值及score
1) "java"
2) "100"
3) "c++"
4) "90"
```

zincrby：为指定元素的score加上指定的增量

```
zincrby key increment member
```

> 为有序集 `key` 的成员 `member` 的 `score` 值加上增量 `increment` 。
>
> 可以通过传递一个负数值 `increment` ，让 `score` 减去相应的值，比如 `ZINCRBY key -5 member` ，就是让 `member` 的 `score` 值减去 `5` 。
>
> 当 `key` 不存在，或 `member` 不是 `key` 的成员时， `ZINCRBY key increment member` 等同于 `ZADD key increment member` 。
>
> 当 `key` 不是有序集类型时，返回一个错误。
>
> `score` 值可以是整数值或双精度浮点数。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ #集合topx中添加3个元素：java、c、c++，对应的score分别是：90、70、80
(integer) 3
127.0.0.1:6379> zrevrange topx 0 -1 withscores #输出集合topx中的元素，包含score
1) "java"
2) "90"
3) "c++"
4) "80"
5) "c"
6) "70"
127.0.0.1:6379> zincrby topx 5 java #对topx中的元素java的score加5，变成95了
"95"
127.0.0.1:6379> zrevrange topx 0 -1 withscores # 输出集合元素，注意java的score是95了
1) "java"
```

```
2) "95"
3) "c++"
4) "80"
5) "c"
6) "70"
```

zrem：删除集合中多个元素

```
zrem key member [member ...]
```

> 移除有序集 `key` 中的一个或多个成员，不存在的成员将被忽略。
>
> 当 `key` 存在但不是有序集类型时，返回一个错误。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ #集合topx中添加3个元素：java、c、c++，对应的score分别是：90、70、80
(integer) 3
127.0.0.1:6379> zrange topx 0 -1 #输出集合topx中所有元素
1) "c"
2) "c++"
3) "java"
127.0.0.1:6379> zrem topx c c++ #删除集合topx中的2个元素：c、c++
(integer) 2
127.0.0.1:6379> zrange topx 0 -1 #输出集合topx中所有元素
1) "java"
```

zremrangebyrank：根据索引范围删除元素

```
zremrangebyrank key start stop
```

> 移除有序集 `key` 中，指定排名(rank)区间内的所有成员。
>
> 区间分别以下标参数 `start` 和 `stop` 指出，包含 `start` 和 `stop` 在内。
>
> 下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。
>
> 你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ #集合topx中添加3个元素：java、c、c++，对应的score分别是：90、70、80
(integer) 3
127.0.0.1:6379> zrange topx 0 -1 #输出集合topx中所有元素
1) "c"
2) "c++"
3) "java"
127.0.0.1:6379> zremrangebyrank topx 0 1 #删除索引范围[0,1]的数据
(integer) 2
127.0.0.1:6379> zrange topx 0 -1 #输出鞣topx中所有元素
1) "java"
```

zremrangebyscore：根据score的范围删除元素

```
zremrangebyscore key min max
```

> 移除有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max`)的成员

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ 50 php #topx集合中添加4个元素
(integer) 4
127.0.0.1:6379> zrange topx 0 -1 withscores #输出topx中所有元素值、score
1) "php"
2) "50"
3) "c"
4) "70"
5) "c++"
6) "80"
7) "java"
8) "90"
127.0.0.1:6379> zremrangebyscore topx 70 80 #删除score位于[70,80]区间的元素
(integer) 2
127.0.0.1:6379> zrange topx 0 -1 withscores #输出剩下的元素
1) "php"
2) "50"
3) "java"
4) "90"
```

zcount：统计指定score范围内元素的个数

```
zcount key min max
```

> 返回有序集 `key` 中， `score` 值在 `min` 和 `max` 之间(默认包括 `score` 值等于 `min` 或 `max`)的成员的数量

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ 50 php #topx集合中添加4个元素
(integer) 4
127.0.0.1:6379> zcount topx 80 100 #统计score位于[80,100]区间中的元素个数
(integer) 2
```

zrank：按照score升序，返回某个元素在集合中的排名

```
zrank key member
```

> 返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递增(从小到大)顺序排列。
>
> 排名以 `0` 为底，也就是说， `score` 值最小的成员排名为 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ 50 php #topx集合中添加4个元素
(integer) 4
127.0.0.1:6379> zrank topx c #获取元素c的排名，返回1表示排名第2
(integer) 1
127.0.0.1:6379> zrange topx 0 -1 #输出集合中所有元素，看一下c的位置确实是2
1) "php"
2) "c"
3) "c++"
4) "java"
```

zrevrank：按照score降序，返回某个元素在集合中的排名

> 返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递减(从大到小)排序。
>
> 排名以 `0` 为底，也就是说， `score` 值最大的成员排名为 `0` 。

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ 50 php #topx集合中添加4个元素
(integer) 4
127.0.0.1:6379> zrange topx 0 -1
1) "php"
2) "c"
3) "c++"
4) "java"
127.0.0.1:6379> zrevrank topx java #score降序，得到java的排名，排在第1位
(integer) 0
```

zscore：返回集合中指定元素的score

```
zscore key member
```

> 返回有序集 `key` 中，成员 `member` 的 `score` 值。
>
> 如果 `member` 元素不是有序集 `key` 的成员，或 `key` 不存在，返回 `nil` 。

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> zadd topx 90 java 70 c 80 c++ 50 php #topx集合中添加4个元素
(integer) 4
127.0.0.1:6379> zrange topx 0 -1 #输出topx集合所有元素
1) "php"
2) "c"
3) "c++"
4) "java"
127.0.0.1:6379> zscore topx java #获取集合topx中java的score
"90"
```

2.6.3、数据结构

SortedSet（zset）是redis提供的一个非常特别的数据结构，内部使用到了2种数据结构。

1、hash表

类似于java中的Map<String,score>，key为集合中的元素，value为元素对应的score，可以用来快速定位元素定义的score，时间复杂度为O(1)

2、跳表

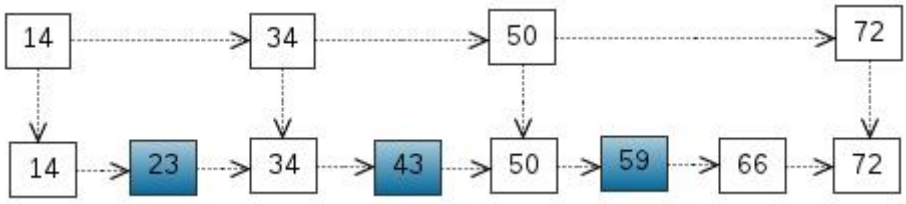
跳表（skiplist）是一个非常优秀的数据结构，实现简单，插入、删除、查找的复杂度均为O(logN)。

类似java中的ConcurrentSkipListSet，根据score的值排序后生成的一个跳表，可以快速按照位置的顺序或者score的顺序查询元素。

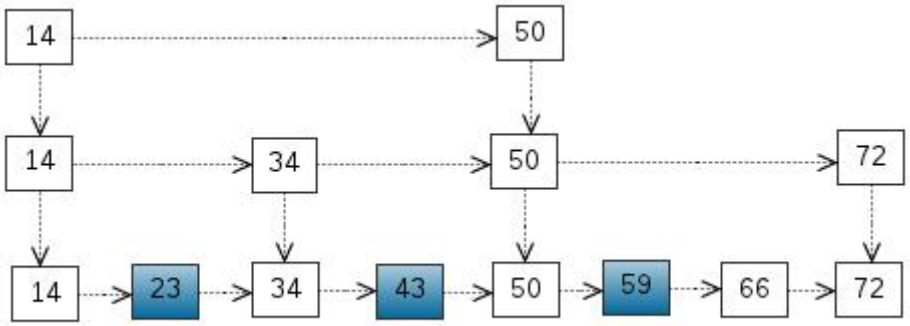
这里我们来看一下跳表的原理：

首先从考虑一个有序表开始：

从该有序表中搜索元素 < 23, 43, 59 > ，需要比较的次数分别为 < 2, 4, 6 >，总共比较的次数为 2 + 4 + 6 = 12 次。有没有优化的算法吗? 链表是有序的，但不能使用二分查找。类似二叉搜索树，我们把一些节点提取出来，作为索引。得到如下结构：



这里我们把 < 14, 34, 50, 72 > 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了。我们还可以再从一级索引提取一些元素出来，作为二级索引，变成如下结构：



这里元素不多，体现不出优势，如果元素足够多，这种索引结构就能体现出优势来了。

3、redis的发布和订阅

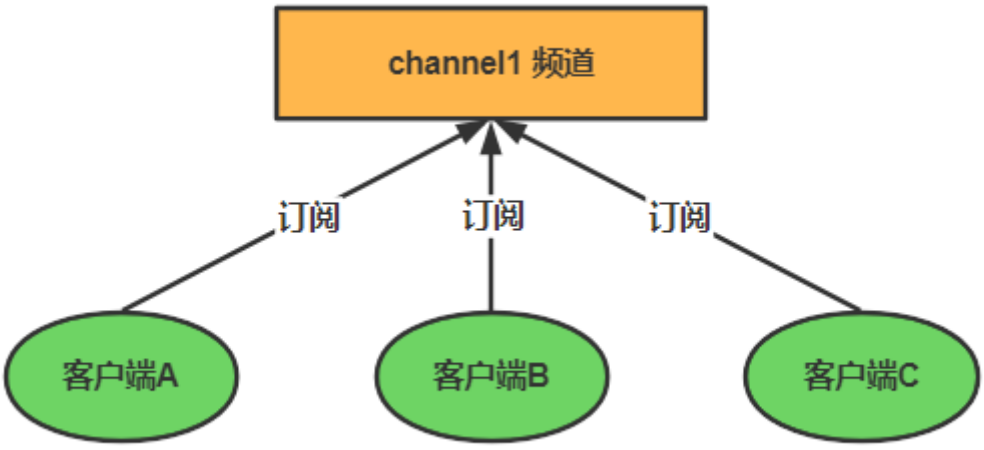
3.1、什么是发布和订阅

redis发布订阅（pub/sub）是一种消息通信模式：发布者（pub）发布消息，订阅者（sub）接收消息。

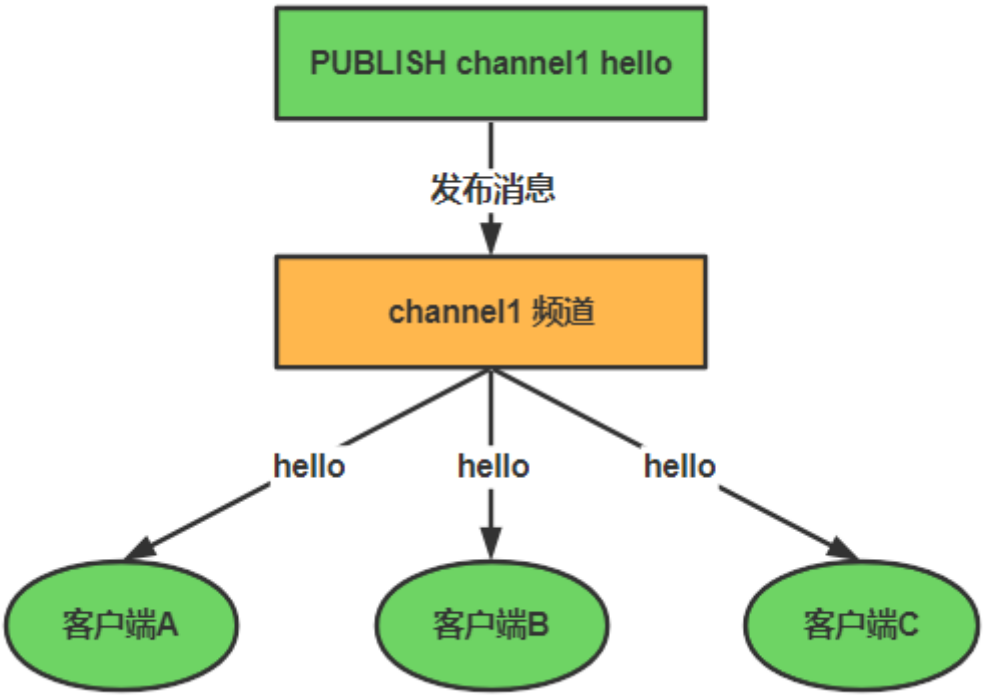
redis客户端可以订阅任意数量的频道。

3.2、redis的发布和订阅

1、客户端可以订阅频道如下图



2、当给这个频道发布消息后，消息就会发送给订阅的客户端



3.3、发布和订阅的命令行实现

1、打开一个客户端订阅channel1

> 订阅命令：`subscribe channel1 channel2 ...`，可以订阅多个频道。

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
```

2、打开另一个客户端，给channel1发布消息hello

> 发标消息命令：`publish channel 消息`，返回值表示有几个订阅者

```
127.0.0.1:6379> publish channel1 helloworld
(integer) 1
```

3、切换到订阅者窗口，可以看到收到信息了

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "message"
2) "channel1"
3) "helloworld"
```

消息
频道名称
内容

3.4、发布和订阅常用命令

3.4.1、subscribe：订阅一个或者多个频道

```
SUBSCRIBE channel [channel ...]
```

订阅给定的一个或多个频道的信息。

返回值：接收到的信息(请参见下面的代码说明)。

```
# 订阅 msg 和 chat_room 两个频道

# 1 - 6 行是执行 subscribe 之后的反馈信息
# 第 7 - 9 行才是接收到的第一条信息
# 第 10 - 12 行是第二条

redis> subscribe msg chat_room
Reading messages... (press Ctrl-C to quit)
1) "subscribe"        # 返回值的类型：显示订阅成功
2) "msg"              # 订阅的频道名字
3) (integer) 1        # 目前已订阅的频道数量

1) "subscribe"
2) "chat_room"
3) (integer) 2

1) "message"          # 返回值的类型：信息
2) "msg"              # 来源(从那个频道发送过来)
3) "hello moto"       # 信息内容

1) "message"
2) "chat_room"
3) "testing...haha"
```

3.4.2、publish：发布消息到指定的频道

```
PUBLISH channel message
```

将信息 `message` 发送到指定的频道 `channel` 。

返回值：接收到信息 `message` 的订阅者数量。

```
# 对没有订阅者的频道发送信息
redis> publish bad_channel "can any body hear me?"
(integer) 0

# 向有一个订阅者的频道发送信息
redis> publish msg "good morning"
(integer) 1

# 向有多个订阅者的频道发送信息
redis> publish chat_room "hello~ everyone"
(integer) 3
```

3.4.2、psubscribe：订阅一个或多个符合给定模式的频道

```
PSUBSCRIBE pattern [pattern ...]
```

订阅一个或多个符合给定模式的频道。

每个模式以 `*` 作为匹配符，比如 `it*` 匹配所有以 `it` 开头的频道(`it.news` 、 `it.blog` 、 `it.tweets` 等等)， `news.*` 匹配所有以 `news.` 开头的频道(`news.it` 、 `news.global.today` 等等)，诸如此类。

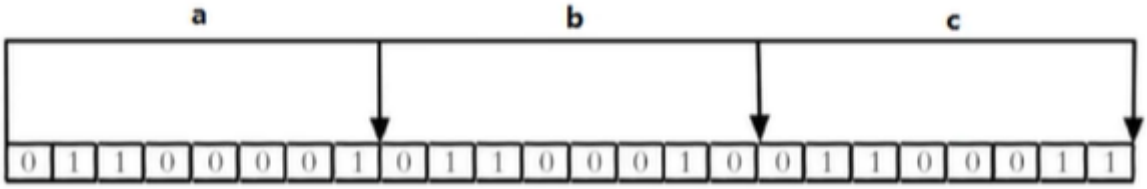
```
# 订阅 news.* 和 tweet.* 两个模式

# 第 1 - 6 行是执行 psubscribe 之后的反馈信息
# 第 7 - 10 才是接收到的第一条信息
# 第 11 - 14 是第二条
# 以此类推。。。

redis> psubscribe news.* tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"                    # 返回值的类型：显示订阅成功
2) "news.*"                        # 订阅的模式
3) (integer) 1                     # 目前已订阅的模式的数量

1) "psubscribe"
2) "tweet.*"
3) (integer) 2

1) "pmessage"                      # 返回值的类型：信息
2) "news.*"                        # 信息匹配的模式
3) "news.it"                       # 信息本身的目标频道
4) "Google buy Motorola"           # 信息的内容

1) "pmessage"
2) "tweet.*"
3) "tweet.huangz"
4) "hello"

1) "pmessage"
2) "tweet.*"
3) "tweet.joe"
4) "@huangz morning"

1) "pmessage"
2) "news.*"
3) "news.life"
4) "An apple a day, keep doctors away"
```

4、redis新的3种数据类型

4.1、Bitmaps：位操作字符串

4.1.1、简介

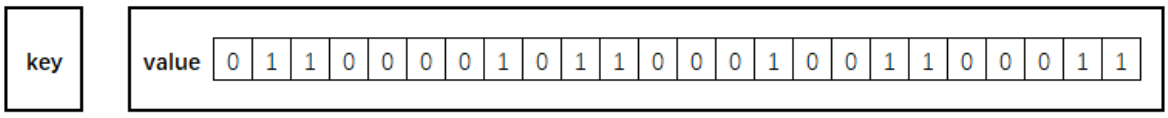
现代计算机使用二进制（位）作为信息的基本单位，1个字节等于8位，例如“abc”字符串是有3个字节组成，但实际在计算机内存储时将其使用二进制表示，“abc”分别对应的ASCII码是：97、98、99，对应的二进制分别是01100001、01100010、01100011，如下图



合理地使用位操作能够有效地提高内存使用率和开发效率。

Redis提供了Bitmaps这个“数据类型”可以实现对位的操作：

- Bitmaps本身不是一种数据类型， 实际上它就是字符串（key-value） ， 但是它可以对字符串的位进行操作，字符串中每个字符对应1个字节，也就是8位，一个字符可以存储8个bit位信息。
- Bitmaps单独提供了一套命令， 所以在Redis中使用Bitmaps和使用字符串的方法不太相同。 可以把Bitmaps想象成一个以位为单位的数组， 数组的每个单元只能存储0和1， 数组的下标在Bitmaps中叫做偏移量。



4.1.2、常用命令

setbit：设置某个偏移量的值（0或1）

```
SETBIT key offset value
```

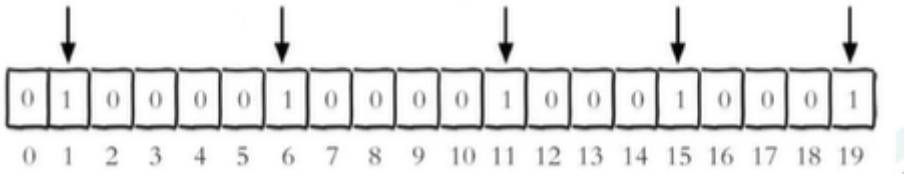
> 设置offset偏移位的值为value，offset的值是从0开始的，**n代表第n+1个bit位置的。**
>
> `offset` 参数必须大于或等于 `0` ，小于 2^32 (bit 映射被限制在 512 MB 之内)。
>
> `value`的值只能为0或1
>
> **返回值**：指定偏移量原来储存的位。

```
redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1

redis> GETBIT bit 100   # bit 默认被初始化为 0
(integer) 0
```

示例

每个独立用户是否访问过网站存放在bitmaps中，将访问的用户记做1，没有访问的用户记做0，用户id作为offset。

假设现在有20个用户，userid=1,6,11,15,19的用户对网站进行了访问，那么当前bitmaps初始化结果如图



`users:20220409`这个bitmaps中表示2022-04-09这天独立访问的用户，如下

```
127.0.0.1:6379> setbit users:20220409 1 1
(integer) 0
127.0.0.1:6379> setbit users:20220409 6 1
(integer) 0
127.0.0.1:6379> setbit users:20220409 11 1
(integer) 0
127.0.0.1:6379> setbit users:20220409 15 1
(integer) 0
127.0.0.1:6379> setbit users:20220409 19 1
(integer) 0
```

getbit：获取某个偏移位的值

```
GETBIT key offset
```

> 获取key所对应的bitmaps中offset偏移位的值。
>
> 返回值：0或者1

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> setbit users 1001 1 #设置偏移量1001的bit位的值为1
(integer) 0
127.0.0.1:6379> getbit users 1001 #获取偏移位1001的bit位的值
(integer) 1
127.0.0.1:6379> getbit users 1000 #获取偏移位1000的bit位的值，未设置，返回0
(integer) 0
```

bitcount：统计bit位都为1的数量

```
BITCOUNT key [start] [end]
```

> 统计字符串被设置为1的bit数，一般情况下，给定的整个字符串都会被进行统计，通过指定额外的start或者end参数，可以让计数只在特定的位上进行，`start` 和 `end` 参数，都可以使用负数值：比如 `-1` 表示最后一个位，而 `-2` 表示倒数第二个位，以此类推。
>
> **注意了：start、end是指bit组的字节的下标数，一个直接对应8个bit，所以[a,b]对应的offset范围是[8a,8b+7]**

```
127.0.0.1:6379> flushdb # 清空db，方便测试
OK
```

```
127.0.0.1:6379> setbit user 7 1 # 设置user这个bitmaps中偏移量为7的bit为值为1，也就是第8个bit位的值
(integer) 0
127.0.0.1:6379> setbit user 15 1 # 设置user这个bitmaps中偏移量为15的bit为值为1
(integer) 0
127.0.0.1:6379> setbit user 23 1 # 设置user这个bitmaps中偏移量为23的bit为值为1
(integer) 0
127.0.0.1:6379> bitcount user # 获取user这个bitmaps中1的数量
(integer) 3
127.0.0.1:6379> bitcount user 0 1 # 获取[0,1]这个字节内bit位上1的数量，也就是offset是[0,15]的位置上1的数量，所以是2个
(integer) 2
127.0.0.1:6379> bitcount user 0 0 # 获取[0,0]这个字节内bit位上1的数量，也就是offset是[0,7]的位置上1的数量，只有7这个位置，所以是1个
(integer) 1
```

bittop：对一个多个bitmaps执行位操作

```
BITOP operation destkey key [key ...]
```

> 对一个或多个保存二进制位的字符串 `key` 进行位元操作，并将结果保存到 `destkey` 上。
>
> `operation` 可以是 `AND` 、 `OR` 、 `NOT` 、 `XOR` 这四种操作中的任意一种：
>
> - `BITOP AND destkey key [key ...]` ，对一个或多个 `key` 求逻辑并，并将结果保存到 `destkey` 。
> - `BITOP OR destkey key [key ...]` ，对一个或多个 `key` 求逻辑或，并将结果保存到 `destkey` 。
> - `BITOP XOR destkey key [key ...]` ，对一个或多个 `key` 求逻辑异或，并将结果保存到 `destkey` 。
> - `BITOP NOT destkey key` ，对给定 `key` 求逻辑非，并将结果保存到 `destkey` 。
>
> 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。
>
> **返回值**：保存到 `destkey` 的字符串的长度，和输入 `key` 中最长的字符串长度相等。

```
redis> SETBIT bits-1 0 1        # bits-1 = 1001
(integer) 0

redis> SETBIT bits-1 3 1
(integer) 0

redis> SETBIT bits-2 0 1        # bits-2 = 1011
(integer) 0

redis> SETBIT bits-2 1 1
(integer) 0

redis> SETBIT bits-2 3 1
(integer) 0

redis> BITOP AND and-result bits-1 bits-2
(integer) 1

redis> GETBIT and-result 0      # and-result = 1001
(integer) 1
```

```
redis> GETBIT and-result 1
(integer) 0

redis> GETBIT and-result 2
(integer) 0

redis> GETBIT and-result 3
(integer) 1
```

4.1.3、bitmaps与set比较

假设网站有 1 亿用户， 每天独立访问的用户有 5 千万， 如果每天用集合类型和 Bitmaps 分别存储活跃用户可以得到表：

set 和 Bitmaps 存储一天活跃用户对比

数据类型	每个用户 id 占用空间	需要存储的用户量	全部内存量
set集合	64 位	50000000	64 位 * 50000000 = 400MB
Bitmaps	1位	100000000	1 位 * 100000000 = 12.5MB

很明显， 这种情况下使用 Bitmaps 能节省很多的内存空间， 尤其是随着时间推移节省的内存还是非常可观的。

set 和 Bitmaps 存储独立用户空间对比

数据类型	一天	一月	一年
set集合	400MB	12GB	144GB
Bitmaps	12.5MB	375MB	4.5GB

但 Bitmaps 并不是万金油， 假如该网站每天的独立访问用户很少， 例如只有 10 万（大量的僵尸用户）， 那么两者的对比如下表所示， 很显然， 这时候使用 Bitmaps 就不太合适了， 因为基本上大部分位都是 0。

数据类型	每个 userid 占用空间	需要存储的用户量	全部内存量
集合	64 位	100000	64 位 * 100000 = 800KB
Bitmaps	1 位	100000000	1 位 * 100000000 = 12.5MB

4.2、HyperLoglog

4.2.1、简介

在工作当中，我们经常会遇到与统计相关的功能需求，比如统计网站 PV（PageView 页面访问量），可以使用 Redis 的 incr、incrby 轻松实现。但像 UV（UniqueVisitor 独立访客）、独立 IP 数、搜索记录数等需要去重和计数的问题如何解决？这种求集合中不重复元素个数的问题称为基数问题。

解决基数问题有很多种方案：

数据存储在 MySQL 表中，使用 distinct count 计算不重复个数。

使用 Redis 提供的 hash、set、bitmaps 等数据结构来处理。

以上的方案结果精确，但随着数据不断增加，导致占用空间越来越大，对于非常大的数据集是不切实际的。能否能够降低一定的精度来平衡存储空间？Redis 推出了 HyperLogLog。

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是：在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。

在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。

但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

什么是基数?

比如数据集 {1, 3, 5, 7, 5, 7, 8}，那么这个数据集的基数集为 {1, 3, 5 ,7, 8}，基数 (不重复元素) 为 5。 基数估计就是在误差可接受的范围内，快速计算基数。

4.2.2、命令

pfadd：添加多个元素

```
pfadd key element [element ...]
```

> 向HyperLoglog类型的key中添加一个或者多个元素。
>
> 添加一个或者多个元素到key对应的集合中。
>
> **返回值：**
>
> 1：添加成功
>
> 0：添加失败

```
127.0.0.1:6379> flushdb # 清空db方便测试
OK
127.0.0.1:6379> pfadd program java php c c++ # program中添加4个元素
[java,php,c,c++]，添加成功发，返回1
(integer) 1
127.0.0.1:6379> pfadd program java # 再次添加java，由于已经存在，所以添加失败，返回0
(integer) 0
127.0.0.1:6379> pfadd program java js # 再次添加2个元素，java已经存在了，但是js不存在，
添加成功，返回1
(integer) 1
```

pfcount：获取多个HLL合并后元素的个数

```
pfcount key1 key2 ...
```

> 统计一个或者多个key去重后元素的数量。

示例

```
127.0.0.1:6379> flushdb # 清空db方便测试
OK
127.0.0.1:6379> pfadd uv1 a b c d e #uv1中5个元素：[a,b,c,d,e]
(integer) 1
127.0.0.1:6379> pfcount uv1 #uv1中数量为5
(integer) 5
127.0.0.1:6379> pfadd uv2 b c d e f #uv2中5个元素：[b,c,d,e,f]
(integer) 1
127.0.0.1:6379> pfcount uv2 #uv2中数量为5
(integer) 5
127.0.0.1:6379> pfcount uv1 uv2 # 获取uv1和uv2去重之后数量合集：[a,b,c,d,e,f]，数量为5
(integer) 5
```

pfmerge：将多个HLL合并后元素放入另外一个HLL

```
pfmerge destkey sourcekey [sourcekey ...]
```

> 将多个`sourcekey`合并后放到`destkey`中。

示例

```
127.0.0.1:6379> flushdb # 清空db方便测试
OK
127.0.0.1:6379> pfadd uv1 a b c d e #uv1中5个元素：[a,b,c,d,e]
(integer) 1
127.0.0.1:6379> pfcount uv1 #uv1中数量为5
(integer) 5
127.0.0.1:6379> pfadd uv2 b c d e f #uv2中5个元素：[b,c,d,e,f]
(integer) 1
127.0.0.1:6379> pfcount uv2 #uv2中数量为5
(integer) 5
127.0.0.1:6379> pfmerge uv_dest uv1 uv2 #将uv1和uv2合并后放入uv_dest
OK
127.0.0.1:6379> pfcount uv_dest #uv_dest元素个数为6
(integer) 6
```

4.3、Geographic

4.3.1、简介

Reids3.2 中增加了对GEO类型的支持，GEO（Geographic），地理信息的缩写。

该类型，就是元素的 2 维坐标，在地图上就是经纬度，redis基于该类型，提供了经纬度设置、查询、范围查询、距离查询，经纬度Hash等常见操作。

4.3.2、命令

geoadd：添加多个位置的经纬度

```
geoadd key longitude latitude member [longitude latitude member ...]
```

> longitude latitude member：经度 纬度 名称

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai #添加上海的经纬度
(integer) 1
127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqing 114.05 22.52 shenzhen
116.38 39.90 beijing #添加重庆、深圳、北京 3 个城市的经纬度
(integer) 3
127.0.0.1:6379> type china:city #发现geo实际上使用zset类型存储的
zset
127.0.0.1:6379> zrange china:city 0 -1
1) "chongqing"
2) "shenzhen"
3) "shanghai"
4) "beijing"
127.0.0.1:6379> zrange china:city 0 -1 withscores
1) "chongqing"
2) "4026042091628984"
3) "shenzhen"
4) "4046432193584628"
5) "shanghai"
6) "4054803462927619"
7) "beijing"
8) "4069885332386336"
```

两级无法直接添加，一般会下载城市数据，直接通过java程序一次性导入。

有效的经纬度从-180度到180度，有效的维度从-85.05112878度到85.05112878度。

当坐标位置超出指定范围时，该命令将会返回一个错误。

已经添加的数据，是无法再次往里面添加的。

geopos：获取多个位置的坐标值

```
geopos key member [member ...]
```

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai #添加上海的经纬度
(integer) 1
127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqing 114.05 22.52 shenzhen
116.38 39.90 beijing #添加重庆、深圳、北京 3 个城市的经纬度
(integer) 3
127.0.0.1:6379> geopos china:city wuhan beijing chongqing #获取武汉、北京、重庆 3个城
市的坐标，由于没有添加武汉的数据，所以没有获取到，其他2个获取到了
1) (nil)
2) 1) "116.38000041246414185"
   2) "39.90000009167092543"
3) 1) "106.49999767541885376"
   2) "29.52999957900659211"
```

geodist：获取两个位置的直线距离

```
geodist key member1 member2 [m|km|ft|mi]
```

> 单位：[m|km|ft|mi] -》[米|千米|英里|英尺]，默认为米

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai #添加上海的经纬度
(integer) 1
127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqing 114.05 22.52 shenzhen
116.38 39.90 beijing #添加重庆、深圳、北京 3 个城市的经纬度
(integer) 3
127.0.0.1:6379> geodist china:city beijing chongqing km #获取北京到重庆的直线距离
"1462.9505"
```

georadius：以给定的经纬度为中心，找出某一半径内的元素

```
georadius key longitude latitude radius m|km|ft|mi
```

> 单位：[m|km|ft|mi] -》[米|千米|英里|英尺]，默认为米

示例

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai #添加上海的经纬度
(integer) 1
127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqing 114.05 22.52 shenzhen
116.38 39.90 beijing #添加重庆、深圳、北京 3 个城市的经纬度
(integer) 3
127.0.0.1:6379> georadius china:city 110 30 1000 km #在china:city中检索：以经纬度
(110,30)为中心，半径为1000km内的位置列表
1) "chongqing"
2) "shenzhen"
```

5、Jedis操作Redis6

5.1、介绍

Jedis是java开发的操作redis的工具包。

5.2、Jedis的用法

5.2.1、引入maven依赖

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>4.2.1</version>
</dependency>
```

5.2.2、使用redis的api操作redis

> 案例代码如下，重点在于`Jedis`工具类，这个类中包含了操作redis的所有方法。

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPubSub;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.TimeUnit;

/**
 * 公众号：Java充电社
 * 官网：http://www.itsoku.com
 */
public class JedisDemo {
    Jedis jedis;

    @Before
    public void before() {
        this.jedis = new Jedis("192.168.200.129", 6379);
    }

    @After
    public void after() {
        //关闭jedis
        this.jedis.close();
    }

    /**
     * 测试redis是否连通
     */
    @Test
```

```
    public void test1() {
        String ping = jedis.ping();
        System.out.println(ping);
    }

    /**
     * string类型测试
     */
    @Test
    public void stringTest() {
        jedis.set("site", "http://www.itsoku.com");
        System.out.println(jedis.get("site"));
        System.out.println(jedis.ttl("site"));
    }

    /**
     * list类型测试
     */
    @Test
    public void listTest() {
        jedis.rpush("courses", "java", "spring", "springmvc", "springboot");
        List<String> courses = jedis.lrange("courses", 0, -1);
        for (String course : courses) {
            System.out.println(course);
        }
    }

    /**
     * set类型测试
     */
    @Test
    public void setTest() {
        jedis.sadd("users", "tom", "jack", "ready");
        Set<String> users = jedis.smembers("users");
        for (String user : users) {
            System.out.println(user);
        }
    }

    /**
     * hash类型测试
     */
    @Test
    public void hashTest() {
        jedis.hset("user:1001", "id", "1001");
        jedis.hset("user:1001", "name", "张三");
        jedis.hset("user:1001", "age", "30");
        Map<String, String> userMap = jedis.hgetAll("user:1001");
        System.out.println(userMap);
    }

    /**
     * zset类型测试
     */
    @Test
    public void zsetTest() {
        jedis.zadd("languages", 100d, "java");
        jedis.zadd("languages", 95d, "c");
```

```
        jedis.zadd("languages", 70d, "php");

        List<String> languages = jedis.zrange("languages", 0, -1);
        System.out.println(languages);
    }

    /**
     * 订阅消息
     *
     * @throws InterruptedException
     */
    @Test
    public void subscribeTest() throws InterruptedException {
        //subscribe(消息监听器,频道列表)
        jedis.subscribe(new JedisPubSub() {
            @Override
            public void onMessage(String channel, String message) {
                System.out.println(channel + ":" + message);
            }
        }, "sitemsg");
        TimeUnit.HOURS.sleep(1);
    }

    /**
     * 发布消息
     *
     * @throws InterruptedException
     */
    @Test
    public void publishTest() {
        jedis.publish("sitemsg", "hello redis");
    }

}
```

6、SpringBoot整合Redis

6.1、引入redis的maven配置

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

6.2、application.properties中配置redis信息

```
# redis 服务器 ip
spring.redis.host=192.168.200.129
# redis 服务器端口
spring.redis.port=6379
# redis 密码
#spring.redis.password=root
# 连接超时时间（毫秒）
spring.redis.timeout=60000
# Redis默认情况下有16个分片，这里配置具体使用的分片，默认是0
spring.redis.database=0
```

6.3、使用RedisTemplate工具类操作redis

springboot中使用RedisTemplate来操作redis，需要在我们的bean中注入这个对象，代码如下：

```
@Autowired
private RedisTemplate<String, String> redisTemplate;

// 用下面5个对象来操作对应的类型
this.redisTemplate.opsForValue(); //提供了操作string类型的所有方法
this.redisTemplate.opsForList(); // 提供了操作list类型的所有方法
this.redisTemplate.opsForSet(); //提供了操作set的所有方法
this.redisTemplate.opsForHash(); //提供了操作hash表的所有方法
this.redisTemplate.opsForZSet(); //提供了操作zset的所有方法
```

6.4、RedisTemplate示例代码

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

@RestController
@RequestMapping("/redis")
public class RedisController {
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    @RequestMapping("/stringTest")
    public String stringTest() {
        this.redisTemplate.delete("name");
        this.redisTemplate.opsForValue().set("name", "路人");
        String name = this.redisTemplate.opsForValue().get("name");
        return name;
    }

    @RequestMapping("/listTest")
    public List<String> listTest() {
        this.redisTemplate.delete("names");
```

```
        this.redisTemplate.opsForList().rightPushAll("names", "刘德华", "张学友",
"郭富城", "黎明");
        List<String> courses = this.redisTemplate.opsForList().range("names", 0,
-1);
        return courses;
    }

    @RequestMapping("setTest")
    public Set<String> setTest() {
        this.redisTemplate.delete("courses");
        this.redisTemplate.opsForSet().add("courses", "java", "spring",
"springboot");
        Set<String> courses = this.redisTemplate.opsForSet().members("courses");
        return courses;
    }

    @RequestMapping("hashTest")
    public Map<Object, Object> hashTest() {
        this.redisTemplate.delete("userMap");
        Map<String, String> map = new HashMap<>();
        map.put("name", "路人");
        map.put("age", "30");
        this.redisTemplate.opsForHash().putAll("userMap", map);

        Map<Object, Object> userMap =
this.redisTemplate.opsForHash().entries("userMap");
        return userMap;
    }

    @RequestMapping("zsetTest")
    public Set<String> zsetTest() {
        this.redisTemplate.delete("languages");

        this.redisTemplate.opsForZSet().add("languages", "java", 100d);
        this.redisTemplate.opsForZSet().add("languages", "c", 95d);
        this.redisTemplate.opsForZSet().add("languages", "php", 70);

        Set<String> languages =
this.redisTemplate.opsForZSet().range("languages", 0, -1);
        return languages;
    }
}
```

7、redis事务操作

7.1、redis事务定义

redis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。

redis事务的主要作用就是串联多个命令防止 别的命令插队。

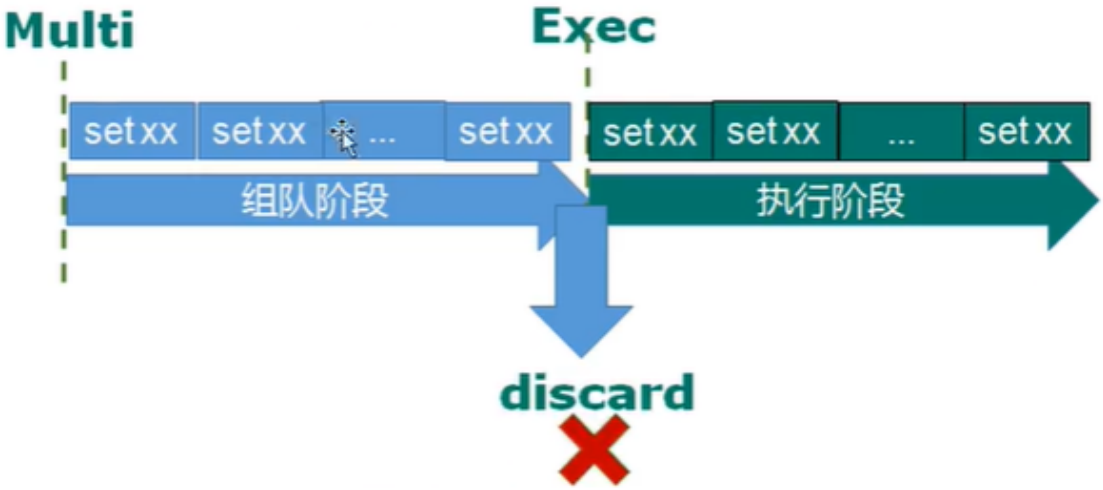
7.2、Multi、Exec、discard

从输入Multi命令开始，输入的命令都会依次进入命令队列中，但不会执行，直到输入Exec后，redis会将之前的命令依次执行。

组队的过程中可以通过discard来放弃组队。

redis事务分2个阶段：组队阶段、执行阶段

- **组队阶段**：只是将所有命令加入命令队列
- **执行阶段**：依次执行队列中的命令，在执行这些命令的过程中，不会被其他客户端发送的请求命令插队或者打断。



7.2.1、相关的几个命令

multi：标记一个事务块的开始

标记一个事务块的开始。

事务块内的多条命令会按照先后顺序被放进一个队列当中，最后由 exec 命令原子性(atomic)地执行。

示例

```
redis> MULTI             # 标记事务开始
OK
redis> INCR user_id      # 多条命令按顺序入队，返回值为QUEUED，表示这个命令加入队列了，还没有被执行。
QUEUED
redis> INCR user_id
QUEUED
redis> INCR user_id
QUEUED
redis> PING
QUEUED
redis> EXEC              # 执行
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG
```

exec：执行所有事务块内的命令

执行所有事务块内的命令。

假如某个(或某些) key 正处于`watch`命令的监视之下，且事务块中有和这个(或这些) key 相关的命令，那么`exec`命令只在这个(或这些) key 没有被其他命令所改动的情况下执行并生效，否则该事务被打断(abort)。

返回值：

事务块内所有命令的返回值，按命令执行的先后顺序排列。

当操作被打断时，返回空值`nil`。

3个示例

```
# 示例1：事务被成功执行
redis> MULTI
OK
redis> INCR user_id
QUEUED
redis> INCR user_id
QUEUED
redis> INCR user_id
QUEUED
redis> PING
QUEUED
redis> EXEC
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG

# 示例2：监视 key，且事务成功执行
redis> WATCH lock lock_times
OK
redis> MULTI
OK
redis> SET lock "huangz"
QUEUED
redis> INCR lock_times
QUEUED
redis> EXEC
1) OK
2) (integer) 1

# 示例3：监视 key，且事务被打断
redis> WATCH lock lock_times
OK
redis> MULTI
OK
redis> SET lock "joe"          # 就在这时，另一个客户端修改了 lock_times 的值
QUEUED
redis> INCR lock_times
QUEUED
redis> EXEC                    # 因为 lock_times 被修改， joe 的事务执行失败
(nil)
```

discard：取消事务

取消事务，放弃执行事务块内的所有命令。

返回值：

总是返回 `OK` 。

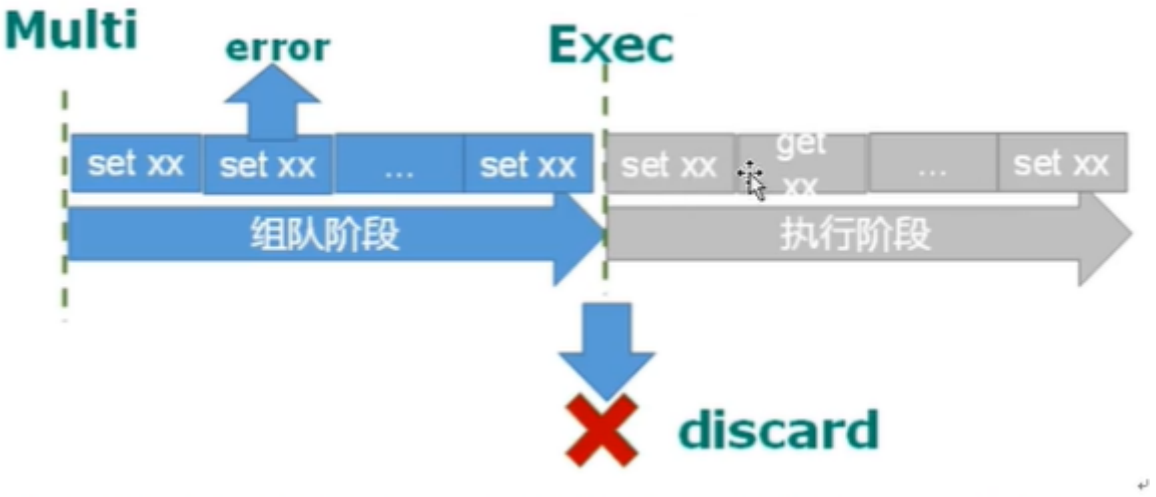
示例

```
redis> MULTI
OK
redis> PING
QUEUED
redis> SET greeting "hello"
QUEUED
redis> DISCARD
OK
```

7.3、事务的错误处理

7.3.1、情况1：组队中命令有误，导致所有命令取消执行

组队中某个命令出现了错误报告，执行时整个队列中所有的命令都会被取消。

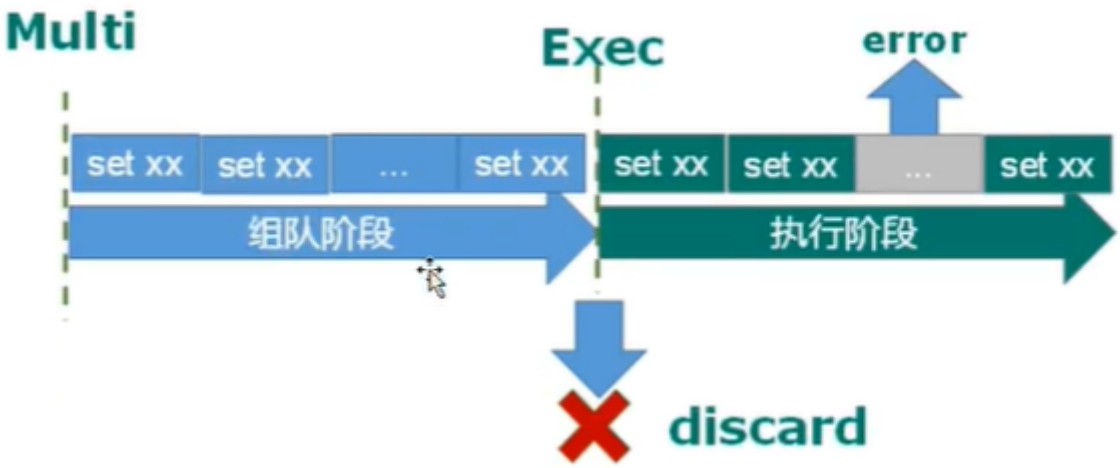


示例代码如下，事务中执行了3个set命令，而第3个命令`set address`命令本身有问题，加入队列失败，最后执行exec的时候，所有的命令都被取消执行。

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> multi #开启一个事务块
OK
127.0.0.1:6379(TX)> set name ready
QUEUED
127.0.0.1:6379(TX)> set age 30
QUEUED
127.0.0.1:6379(TX)> set address #命令有问题，导致加入队列失败
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379(TX)> exec #执行exec的时候，事务中所有命令都被取消
(error) EXECABORT Transaction discarded because of previous errors.
```

7.3.2、情况2：组队中没有问题，执行中部分成功部分失败

命令组队的过程中没有问题，执行中出现了错误会导致部分成功部分失败。



示例代码如下，事务中有3个命令，3个命令都入队列成功了，执行exec命令的时候，1和3命令成功了，第2个失败了

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v1 #命令1：设置k1的值为v1
QUEUED
127.0.0.1:6379(TX)> incr k1 #命令2：k1的值递增1，由于k1的值不是数字，执行的时候会失败的
QUEUED
127.0.0.1:6379(TX)> set k2 v2 #命令3：设置k2的值为v2
QUEUED
127.0.0.1:6379(TX)> exec #执行命令，1和3命令成功，第2个失败了
1) OK
2) (error) ERR value is not an integer or out of range
3) OK
127.0.0.1:6379> mget k1 k2 #查看k1和k2的值
1) "v1"
2) "v2"
```

7.4、事务冲突的问题

7.4.1、例子

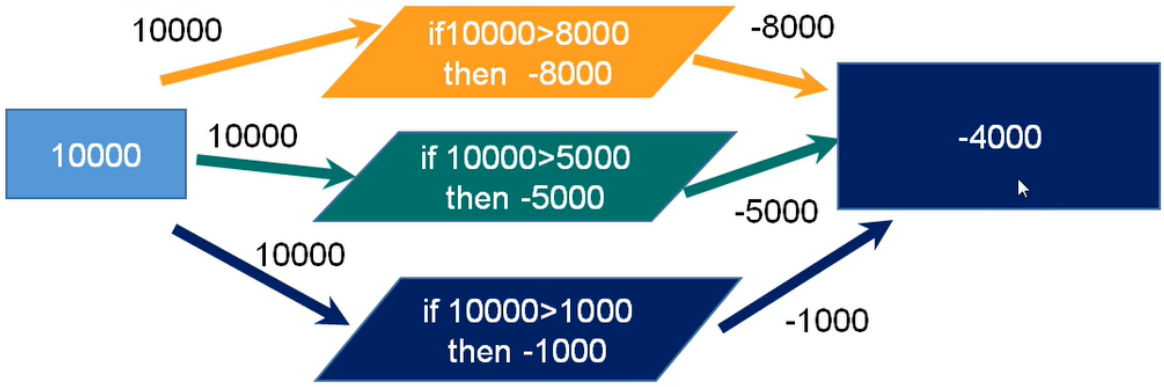
想象一个场景：

你的账户中只有10000，有多个人使用你的账户，同时去参加双十一抢购

一个请求想给金额减8000

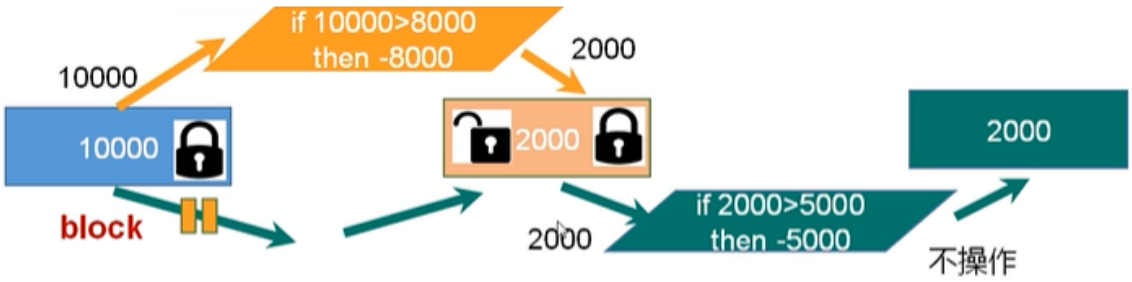
一个请求想给金额减5000

一个请求想给金额减1000



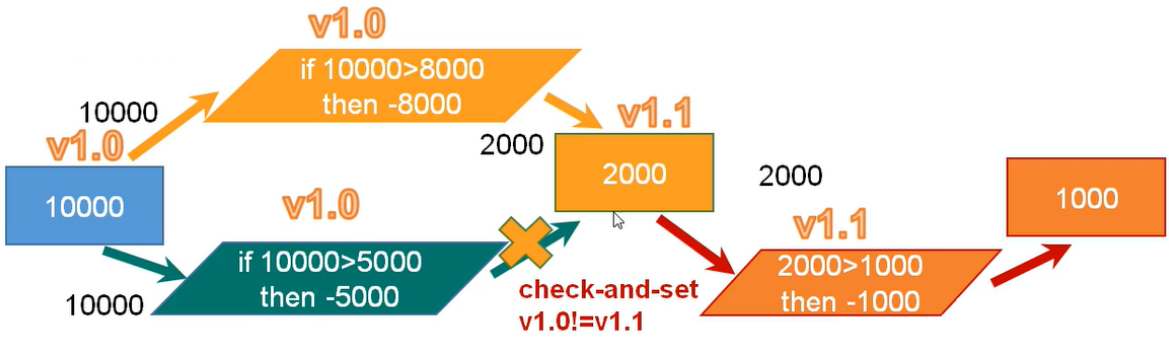
3个请求同时来带①，看到的余额都是10000，大于操作金额，都去执行修改余额的操作，最后导致金额变成了-4000，这显然是有问题的。

7.4.2、悲观锁



悲观锁（Pessimistic Lock），顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人拿到这个数据就会block直到它拿到锁。传统的关系型数据库里面就用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等，都是在做操作之前先上锁。

7.4.3、乐观锁



乐观锁（Optimistic Lock），顾名思义，就是很乐观，每次去那数据的时候都认为别人不会修改，所以不会上锁，但是在修改的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量。redis就是使用这种check-and-set机制实现事务的。

7.4.4、watch key [key ...]

在执行multi之前，先执行watch key1 [key2 ...]，可以监视一个或者多个key，若在事务的exec命令之前这些key对应的值被其他命令所改动了，那么事务中所有命令都将被打断，即事务所有操作将被取消执行。

示例

开启2个窗口，按照下表的时间点在不同的窗口执行对应的命令，注意看结果。

时刻	窗口1	窗口2
T1	flushdb	
T2	set balance 100	
T3	watch balance	
T4	multi	
T5	set name ready	incrby balance 50
T6	incrby balance 10	get balance
T7	exec	
T8	get balance	
T9	get name	

窗口1中，对balance进行了监视，也就是说在执行`watch balance`命令之后，在`exec`命令之前，如果有其他请求对balance进行了修改，那么窗口1事务中所有的命令都会将会被取消执行。

窗口1`watch balance`后，由于T5时刻窗口2对balance进行了修改，导致窗口1中事务所有命令被取消执行。

窗口1执行结果如下

```
127.0.0.1:6379> flushdb #清空db，方便测试
OK
127.0.0.1:6379> set balance 100 #设置balance的值为100
OK
127.0.0.1:6379> watch balance #监视balance，若balance在事务阶段被其他命令修改，事务执行将被取消
OK
127.0.0.1:6379> multi #开启事务
OK
127.0.0.1:6379(TX)> set name ready #设置name的值为ready
QUEUED
127.0.0.1:6379(TX)> incrby balance 10 #将balance的值+10
QUEUED
127.0.0.1:6379(TX)> exec #执行事务，由于balance被窗口2修改了，所以本事务执行失败，返回nil
(nil)
127.0.0.1:6379> get balance #获取balance，原始值为100，被窗口2加了50，结果为150
"150"
127.0.0.1:6379> get name #获取name的值，事务中set name 未成功，所以name没有
(nil)
```

窗口2执行结果

```
127.0.0.1:6379> incrby balance 50 #balance原子+50
(integer) 150
127.0.0.1:6379> get balance #获取balance的值，为150
"150"
```

7.4.5、unwatch：取消监视

取消 WATCH 命令对所有 key 的监视。

如果在执行 WATCH 命令之后， EXEC 命令或 DISCARD 命令先被执行了的话，那么就不需要再执行 UNWATCH 了。

因为 EXEC 命令会执行事务，因此 WATCH 命令的效果已经产生了；而 DISCARD 命令在取消事务的同时也会取消所有对 key 的监视，因此这两个命令执行之后，就没有必要执行 UNWATCH 了。

```
redis> WATCH lock lock_times
OK

redis> UNWATCH
OK
```

7.5、redis事务三特性

（1）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，事务在执行过程中，不会被其他客户端发送来的命令请求所打断。

（2）没有隔离级别的概念

队列中的命令没有提交（exec）之前，都不会实际被执行，因为事务提交前任何指令都不会被实际执行。

（3）不能保证原子性

事务中如果有一条命令执行失败，后续的命令仍然会被执行，没有回滚。

如果在组队阶段，有1个失败了，后面都不会成功；如果在组队阶段成功了，在执行阶段有那个命令失败就这条失败，其他的命令则正常执行，不保证都成功或都失败。

8、redis持久化之RDB（Redis DataBase）

8.1、总体介绍

Redis是一个基于内存的数据库，它的数据是存放在内存中，内存有个问题就是关闭服务或者断电会丢失。

Redis的数据也支持写到硬盘中，这个过程就叫做持久化。

Redis提供了2种不同形式的持久化方式。

- RDB（Redis DataBase）
- AOP（Append Of File）

8.2、RDB（Redis DataBase）

8.2.1、RDB是什么?

在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是行话讲的Snapshot快照，它恢复时是键快照文件直接读到内存里。

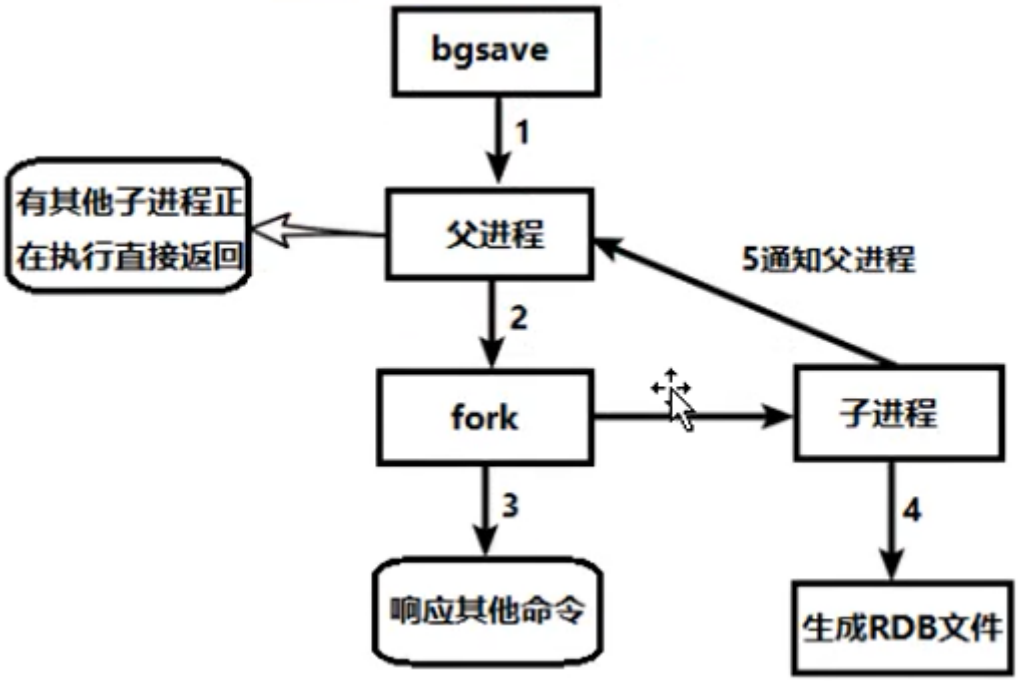
8.2.2、备份是如何执行的

Redis会单独创建（fork）一个子进程进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束后，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就是确保了极高的性能，如果需要进行大规模的恢复，且对数据恢复的完整性不是非常敏感，那RDB方式要不AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。

8.2.3、Fork

- Fork的作用是复制一个与当前进程一样的进程，新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，它是一个全新的进程，并作为原进程的子进程。
- 在Linux程序中，fork()会产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用，处于效率考虑，linux中引入了“写时复制技术”
- 一般情况父进程和子进程会共用一段物理内存，只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。

8.2.4、RDB持久化流程



8.2.5、指定备份文件的名称

在redis.conf中，可以修改rdb备份文件的名称，默认为dump.rdb，如下：

```
418  # sanitize-dump-payload no
419
420  # The filename where to dump the DB
421  dbfilename dump.rdb
422
```

8.2.6、指定备份文件存放的目录

在redis.conf中，rdb文件的保存的目录是可以修改的，默认为Redis启动命令所在的目录，如下

```
441  # The Append Only File will also be created inside this dir
442  #
443  # Note that you must specify a directory here, not a file n
444  dir ./
445
```

默认值./，表示执行redis-server命令启动redis时所在的目录

8.2.7、触发RDB备份

方式1：自动备份，需配置备份规则

可在redis.conf中配置自动备份的规则，默认规则如下：

```
364  # Unless specified otherwise, by default Redis will save the DB:
365  #   * After 3600 seconds (an hour) if at least 1 key changed
366  #   * After 300 seconds (5 minutes) if at least 100 keys changed
367  #   * After 60 seconds if at least 10000 keys changed
368  #
369  # You can set these explicitly by uncommenting the three following lines.
370  #
371  # save 3600 1
372  # save 300 100
373  # save 60 10000
374
```

默认备份规则是：
1分钟内修改了1万次，或5分钟内需修改了10次，或30分钟内修改了1次

save用来配置备份的规则

save的格式：`save 秒钟 写操作次数`

默认是1分钟内修改了1万次，或5分钟内需修改了10次，或30分钟内修改了1次。

示例：设置20秒内有最少有3次key发生变化，则进行备份

```
save 20 3
```

方式2：手动执行命令备份（save | bgsave）

有2个命令可以触发备份。

save：save时只管保存，其他不管，全部阻塞，手动保存，不建议使用。

bgsave：redis会在后台异步进行快照操作，快照同时还可以响应客户端情况。

可以通过`lastsave`命令获取最后一次成功生成快照的时间。

方式3：flushall命令

执行flushall命令，也会产生dump.rdb文件，但里面是空的，无意义。

8.2.8、redis.conf 其他一些配置

stop-writes-on-bgsave-error：当磁盘满时，是否关闭redis的写操作

> stop-writes-on-bgsave-error用来指定当redis无法写入磁盘的话，是否直接关掉redis的写操作，推荐yes。

```
# However if you have setup your proper monitoring of the Redis server
# and persistence, you may want to disable this feature so that Redis will
# continue to work as usual even if there are problems with disk,
# permissions, and so forth.
stop-writes-on-bgsave-error yes

```

rdbcompression：rdb备份是否开启压缩

> 对于存储到磁盘中的rdb快照文件，可以设置是否进行压缩，如果是的话，redis会采用LZF算法进行压缩。
>
> 如果你不想小号CPU来进行压缩的话，可以设置为关闭此功能，推荐yes。

```
# Compress string objects using LZF when dump .rdb databases?
# By default compression is enabled as it's almost always a win.
# If you want to save some CPU in the saving child set it to 'no' but
# the dataset will likely be bigger if you have compressible values or keys.
rdbcompression yes

```

rdbchecksum：是否检查rdb备份文件的完整性

> 存储快照后，还可以让redis使用CRC64算法来进行数据校验，但是这样做会增加大约10%的性能消耗，如果希望获取最大的性能提升，可以关闭此功能。
>
> 推荐yes。

```
# RDB files created with checksum disabled have a checksum of zero that will
# tell the loading code to skip the check.
rdbchecksum yes

```

8.2.9、rdb的备份和恢复

1. 先通过config get dir 查询rdb文件的目录

```
127.0.0.1:6379> config get dir
1) "dir"
2) "/root"
127.0.0.1:6379>
```

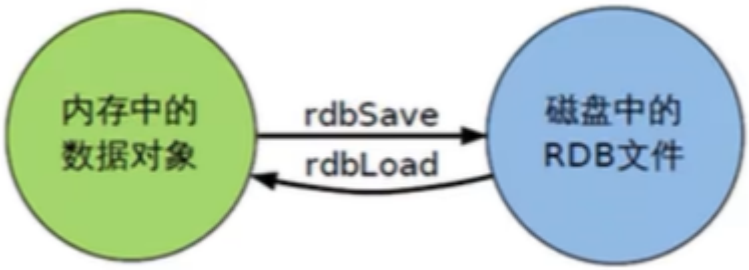
2. 然后将rdb的备份文件 `*.rdb` 文件拷贝到别的地方

```
cp dump.rdb dump2.rdb
```

3. rdb的恢复
   - 关闭redis
   - 先把备份的文件拷贝到工作目录 `cp dump2.rdb dump.rdb`
   - 启动redis，备份数据直接加载，数据被恢复

8.2.10、优势

- 适合大规模数据恢复
- 对数据完整性和一致性要求不高更适合使用
- 节省磁盘空间
- 恢复速度快



8.2.10、劣势

- Fork的时候，内存中的数据会被克隆一份，大致2倍的膨胀，需要考虑
- 虽然Redis在fork的时候使用了写时拷贝技术，但是如果数据庞大时还是比较消耗性能
- 在备份周期在一定间隔时间做一次备份，所以如果Redis意外down的话，就会丢失最后一次快照后所有修改

8.2.11、如何停止RDB?

动态停止RDB：`redis-cli config set save ""` #save后给空值，表示禁用保存策略。

9、redis持久化之AOF（Append Only File)

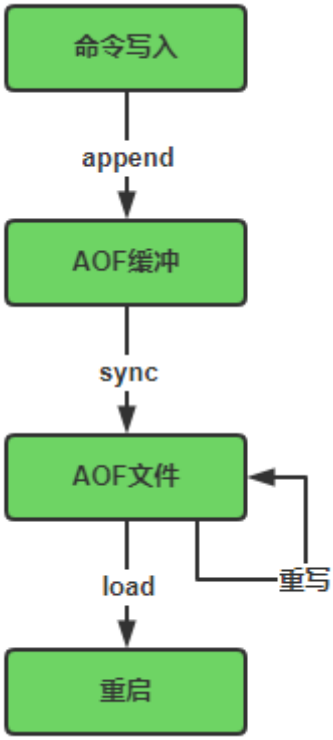
9.1、AOF（Append Only File)

9.1.1、是什么

以日志的形式来记录每个写操作（增量保存），将redis执行过的所有写指令记录下来（读操作不记录），只允追加文件但不可改写文件，redis启动之初会读取该文件重新构造数据，换言之，redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。

9.1.2、AOF持久化流程

- 客户端的请求写命令会被append追加到AOF缓冲区内
- AOF缓冲区会根据AOF持久化策略[always,everysec,no]将操作sync同步到磁盘的AOF文件中
- AOF文件大小超过重写策略或手动重写时，会对AOF文件进行重写（rewrite），压缩AOF文件容量
- redis服务器重启时，会重新load加载AOF文件中的写操作达到数据恢复的目的



9.1.3、AOF默认不开启

可以在`redis.conf`文件中对AOF进行配置

```
appendonly no # 是否开启AOF，yes：开启，no：不开启，默认为no
appendfilename "appendonly.aof" # aof文件名称，默认为appendonly.aof
dir ./  # aof文件所在目录，默认./，表示执行启动命令时所在的目录，比如我们在/opt目录中，去执行
redis-server /etc/redis.conf 来启动redis，那么dir此时就是/opt目录
```

9.1.4、AOF和RDB同时开启，redis听谁的?

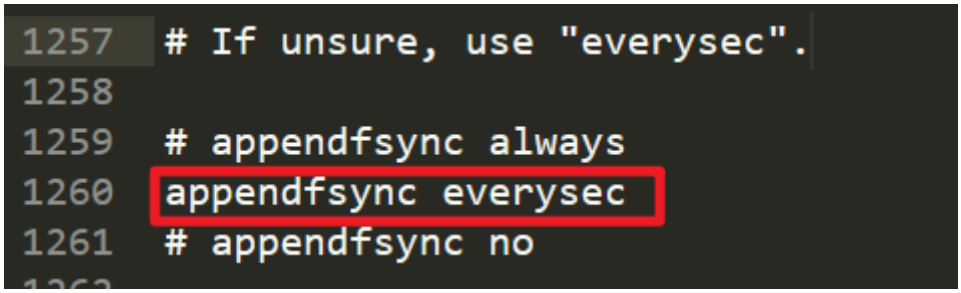
AOF和RDB同时开启，系统默认取AOF的数据（数据不会存在丢失）

9.1.5、AOF启动/修复/恢复

- AOF的备份机制和性能虽然和RDB不同，但是备份和恢复的操作同RDB一样，都是拷贝备份文件，需要恢复时再拷贝到Redis工作目录下，启动系统即加载。
- 正常恢复
  - 修改默认的appendonly no，改为yes
  - 将有数据的aof文件复制一份保存到对应的目录（查看目录：config get dir）
  - 恢复：重启redis然后重新加载
- 异常恢复
  - 修改默认的appendonly no，改为yes
  - 如遇到aof文件损坏，通过`/usr/local/bin/redis-check-aof --fix appendonly.aof`进行恢复

9.1.6、AOF同步频率设置

可以在redis.config中配置AOF同步的频率



appendfsync always：每次写入立即同步

始终同步，每次redis的写入都会立刻记入日志；性能较差但数据完整性比较好。

appendfsync everysec：每秒同步

每秒同步，每秒记录日志一次，如果宕机，本秒数据可能丢失；更新的命令会放在内存中AOF缓冲区，每秒将缓冲区的命令追加到AOF文件

appendfsync no：不主动同步

redis不主动进行同步，把同步交给操作系统。

9.1.7、rewrite压缩（AOF文件压缩）

rewrite压缩是什么?

AOF采用文件追加方式，文件会越来越大，为了避免出现此情况，新增了重写机制，当AOF文件的大小超过锁审定的阈值时，Redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集，可以使用命令bgrewriteaof触发重写。

重写原理，如何实现重写?

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写（也是先写临时文件，最后在rename替换旧文件），redis4.0版本后的重写，是指就把rdb的快照，以二进制的形式附在新的aof头部，作为已有的历史数据，替换掉原来的流水账操作。

触发机制，何时重写?

bgrewriteaof：手动触发重写

从 Redis 2.4 开始， AOF 重写由 Redis 自行触发， `bgrewriteaof` 仅仅用于手动触发重写操作。

redis会记录上次重写的aof大小，默认配置是当aof文件大小是上次rewrite后大小的2倍且文件大于64M时触发。

重写虽然可以节约大量磁盘空间，减少恢复时间，但是每次重写还是有一定负担的，因此设置redis满足一定条件才会进行重新。

auto-aof-rewrite-percentage：设置重写基准值

设置重写的基准值，默认100，当文件达到100%时开始重写（文件是原来重写后文件的2倍时重写）。

auto-aof-rewrite-min-size：设置重写基准值

设置重写的基准值，默认64MB，AOF文件大小超过这个值开始重写。

举个例子

文件达到70MB开始重写，降到50MB，下次什么时候开始重写？100MB

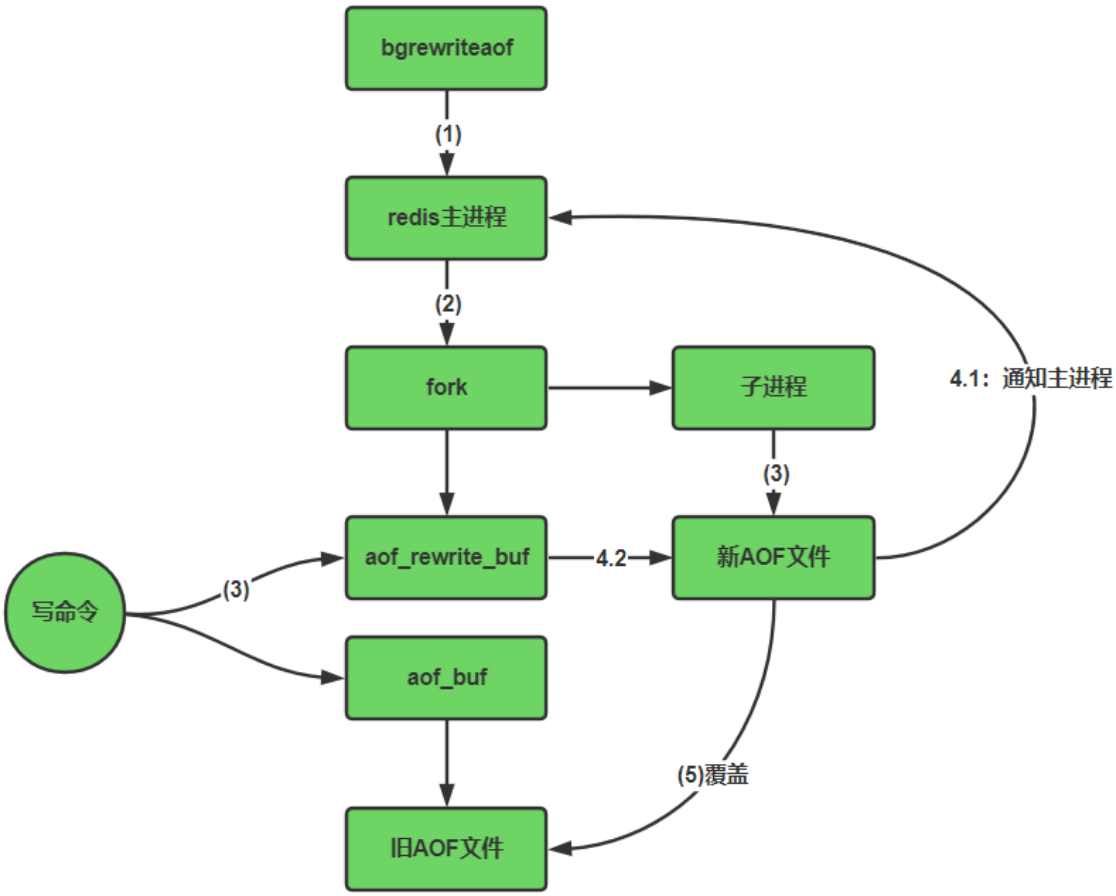
系统载入时或者上次重写完毕时，redis会记录此时AOF大小，设置base_size。

如果Redis的AOF当前大小>=base_size+base_size*100%(`auto-aof-rewrite-percentage`默认值) 且 当前大小>=64mb(`auto-aof-rewrite-min-size`默认值)的情况下，redis会对AOF进行重写。

重写流程

```
127.0.0.1:6379> bgrewriteaof
Background append only file rewriting started
```

- 手动执行`bgrewriteaof`命令触发重写，判断是否当前有bgfsave或bgrewriteaof在运行，如果有，则等待该命令结束后再继续执行
- 主进程fork出子进程执行重写操作，保证主进程不会阻塞
- 子进程遍历redis内存中的数据到临时文件，客户端的写请求同时写入aof_buf缓冲区和aof_rewrite_buf重写缓冲区保证原AOF文件完整性以及新AOF文件生成期间的新的数据修改动作不会丢失
- 子进程写完新的AOF文件后，向主进程发送信号，父进程更新统计信息
- 主进程把aof_rewrite_buf中的数据写入到新的AOF文件
- 使用新的AOF文件覆盖旧的AOF文件，完成AOF重写



no-appendfsync-on-rewrite：重写时，不会执行appendfsync操作

该参数表示在正在进行AOF重写时不会将AOF缓冲区中的数据同步到旧的AOF文件磁盘，也就是说在进行AOF重写的时候，如果此时有写操作进俩，此时写操作的命令会放在aof_buf缓存中（内存中），而不会将其追加到旧的AOF文件中，这么做是为了避免同时写旧的AOF文件和新的AOF文件对磁盘产生的压力。

默认是ON，表示关闭，即在AOF重写时，会对AOF缓冲区中的数据做同步磁盘操作，这在很大程度上保证了数据的安全性。

但在数据量很大的场景，因为两者都会消耗磁盘IO，对磁盘的影响较大，可以将其设置为“yes”减轻磁盘压力，但在极端情况下可能丢失整个AOF重写期间的数据。

如果no-appendfsync-on-rewrite为yes，不写入aof文件，只写入缓存，用户请求不会阻塞，但是在这段时间如果宕机会丢失这段时间的缓存数据。（降低数据安全性，提高性能）

如果no-appendfsync-on-rewrite为no，还是会把数据库往磁盘里刷，但是遇到重写操作，可能会发生阻塞。（数据安全，但是性能降低）

9.1.8、AOF优势

- 备份机制更稳健，丢失数据概率更低
- 可读的日志文本，通过操作AOF文件，可以处理误操作

9.1.9、劣势

- 比RDB占用更多的磁盘空间
- 恢复备份速度要慢
- 每次读写都同步的话，有一定的性能压力
- 存在个别bug，造成不能恢复

9.1.10、小总结

- AOF文件是一个只进行追加的日志文件
- Redis可以在AOF文件体积变得过大时，自动地在后台对AOF文件进行重写
- AOF文件有序地保存了对数据库执行的所有写入操作，这些写入操作以redis协议的格式保存，因此AOF文件的内容非常容易被人读懂，对文件进行分析也很轻松。
- 对于相同的数据集来说，AOF文件的体积通常要大于RDB文件的体积
- 根据所使用的fsync策略，AOF的速度可能会慢于RDB

9.2、总结

9.2.1、用哪个好?

官方推荐2个都启用。

如果对数据不敏感，可以单独用RDB。

不建议单独使用AOF，因为可能会出现BUG。

如果只是做纯内存缓存，可以都不用。

9.2.2、官网建议

- RDB持久化方式能够在指定的时间间隔对你的数据进行快照存储
- AOP持久化方式记录每次对服务器写的操作，当服务器重启的时候会重新执行这些命令来恢复原始数据，AOF命令以redis协议追加保存每次写的操作到AOF文件末尾
- Redis还能对AOF文件进行后台重写，使得AOF文件的体积不至于过大
- 只做缓存：如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式
- 同时开启两种持久化方式
- 在这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据，因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整
- RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件，那要要只是用AOF呢?
- 建议不要，因为RDB更适合用于备份数据库（AOF在不断变化不好北非），快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段
- 性能建议
    - 因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留`save 900 1`这一条
    - 如果使用AOF，好处是在最恶劣的情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了
    - AOF的代价，一是带来持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据（aof_rewrite_buf）写到文件造成的阻塞几乎是不可避免的
    - 只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基数大小默认值64M（auto-aof-rewrite-min-size）太小了，可以设置到5G以上
    - 默认超过原大小100%（auto-aof-rewrite-percentage）大小时重写可以改到适当的数值。

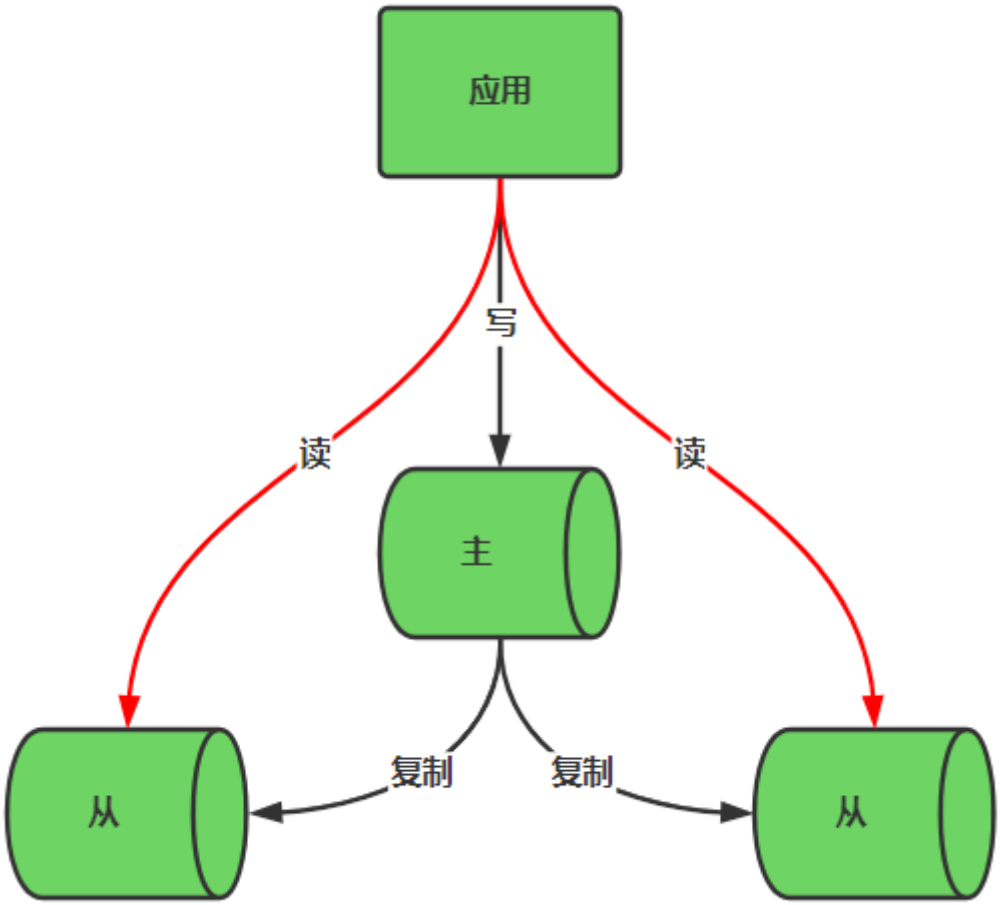
10、redis主从复制

10.1、是什么?

主机更新后根据配置和策略，自动同步到备机的master/slave机制，Master以写为主，Slave以读为主。

10.2、能干嘛?

- 读写分离，性能扩展，降低主服务器的压力
- 容灾，快速恢复，主机挂掉时，从机变为主机

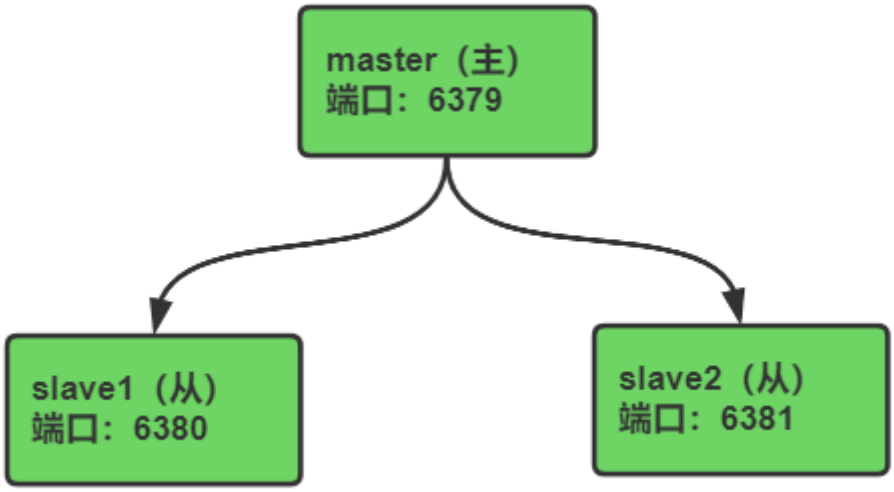


10.3、主从复制：怎么玩?

10.3.1、配置1主2从

下面我们来配置1主2从的效果，现实中是需要3台机器的，为了方便，我们就在一台机器上来演示，通过不同的端口来区分机器，3台机器的配置

角色	端口
master（主）	6379
slave1（从）	6380
slave2（从）	6381



10.3.2、配置主从

1）创建案例工作目录：master-slave

执行下面命令创建`/opt/master-slave`目录，本次所有操作，均在`master-slave`目录进行。

```
ps -ef | grep redis | awk -F" " '{print $2;}' | xargs kill -9 # 方便演示，停止所有的redis
mkdir /opt/master-slave
cd /opt/master-slave/
```

2）将redis.conf复制到master-slave目录

```
cp /opt/redis-6.2.1/redis.conf /opt/master-slave/
```

3）创建master的配置文件：redis-6379.conf

在/opt/master-slave目录创建`redis-6379.conf`文件，内容如下，注意`192.168.200.129`是这个测试机器的ip，大家需要替换为自己的

```
#redis.conf是redis原配置文件，内部包含了很多默认的配置，这里使用include将其引用，相当于把redis.conf内容直接贴进来了
include /opt/master-slave/redis.conf
daemonize yes
bind 192.168.200.129
#配置密码
requirepass 123456
dir /opt/master-slave/
logfile /opt/master-slave/6379.log

#端口
port 6379
#rdb文件
dbfilename dump_6379.rdb
#pid文件
pidfile /var/run/redis_6379.pid
```

4）创建slave1的配置文件：redis-6380.conf

在/opt/master-slave目录创建`redis-6380.conf`文件，内容如下，和上面master的类似，多了后面2行

```
include /opt/master-slave/redis.conf
daemonize yes
bind 192.168.200.129
requirepass 123456
dir /opt/master-slave/

port 6380
dbfilename dump_6380.rdb
pidfile /var/run/redis_6380.pid
logfile /opt/master-slave/6380.log

#用来指定主机：slaveof 主机ip 端口
slaveof 192.168.200.129 6379
#主机的密码
masterauth 123456
```

5）创建slave2的配置文件：redis-6381.conf

```
include /opt/master-slave/redis.conf
daemonize yes
bind 192.168.200.129
requirepass 123456
dir /opt/master-slave/

port 6381
dbfilename dump_6381.rdb
pidfile /var/run/redis_6381.pid
logfile /opt/master-slave/6381.log

#用来指定主机：slaveof 主机ip 端口
slaveof 192.168.200.129 6379
#主机的密码
masterauth 123456
```

6）启动master

```
redis-server /opt/master-slave/redis-6379.conf
```

7）启动slave1

```
redis-server /opt/master-slave/redis-6380.conf
```

8）启动slave2

```
redis-server /opt/master-slave/redis-6381.conf
```

若启动有误，大家好好检查下配置，也可以看日志，3台机器启动会在`/opt/master-slave`目录产生日志，如下

```
[root@hspEdu01 master-slave]# pwd
/opt/master-slave
[root@hspEdu01 master-slave]# ll
总用量 128
-rw-r--r--. 1 root root  2527 4月  14 19:09 6379.log
-rw-r--r--. 1 root root  2133 4月  14 19:08 6380.log
-rw-r--r--. 1 root root  2134 4月  14 19:09 6381.log
-rw-r--r--. 1 root root   175 4月  14 19:09 dump_6379.rdb
-rw-r--r--. 1 root root   175 4月  14 19:08 dump_6380.rdb
-rw-r--r--. 1 root root   175 4月  14 19:09 dump_6381.rdb
-rw-r--r--. 1 root root   261 4月  14 19:05 redis-6379.conf
-rw-r--r--. 1 root root   328 4月  14 19:06 redis-6380.conf
-rw-r--r--. 1 root root   328 4月  14 19:07 redis-6381.conf
-rw-r--r--. 1 root root 92222 4月  14 18:33 redis.conf
```

9）查看主机的信息

通过redis-cli命令连接主机，如下

```
redis-cli -h 192.168.200.129 -p 6379 -a 123456
```

通过下面命令，查看主机信息

```
info Replication
```

```
[root@hspEdu01 master-slave]# redis-cli -h 192.168.200.129 -p 6379 -a 123456   连接主机
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
192.168.200.129:6379> info Replication   查看主从信息的命令
# Replication
role:master
connected_slaves:2
slave0:ip=192.168.200.129,port=6380,state=online,offset=294,lag=1
slave1:ip=192.168.200.129,port=6381,state=online,offset=294,lag=0
master_failover_state:no-failover
master_replid:06afe9a22eaba2c7d99c23d4773a7208fc6b5068
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:294
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:294
192.168.200.129:6379>
```

role：当期的角色，master表示主机
connected_slaves：2，表示有2个从机
下面2行是从机的信息（ip、端口等信息）

10）查看slave1的信息

通过下面2个命令查询从机slave1的信息

```
redis-cli -h 192.168.200.129 -p 6380 -a 123456
info Replication
```

```
[root@hspEdu01 ~]# redis-cli -h 192.168.200.129 -p 6380 -a 123456
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
192.168.200.129:6380> info Replication
# Replication
role:slave
master_host:192.168.200.129
master_port:6379
master_link_status:up
master_last_io_seconds_ago:4
master_sync_in_progress:0
slave_repl_offset:910
slave_priority:100
slave_read_only:1
connected_slaves:0
master_failover_state:no-failover
master_replid:06afe9a22eaba2c7d99c23d4773a7208fc6b5068
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:910
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:910
```

role：slave，表示当前是从机的角色
master_host：主机的ip
master_port：主机的端口

11）同样查看slave2的信息

```
redis-cli -h 192.168.200.129 -p 6381 -a 123456
info Replication
```

```
[root@hspEdu01 master-slave]# redis-cli -h 192.168.200.129 -p 6381 -a 123456
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
192.168.200.129:6381> info Replication
# Replication
role:slave
master_host:192.168.200.129
master_port:6379
master_link_status:up
master_last_io_seconds_ago:9
master_sync_in_progress:0
slave_repl_offset:1134
slave_priority:100
slave_read_only:1
connected_slaves:0
master_failover_state:no-failover
master_replid:06afe9a22eaba2c7d99c23d4773a7208fc6b5068
master_replid2:0000000000000000000000000000000000000000
```

12）验证主从同步效果

在master上面执行下面2个命令

```
192.168.200.129:6379> flushdb
OK
192.168.200.129:6379> set name ready
OK
192.168.200.129:6379> set age 30
OK
```

到slave1上执行下面命令，可以看出来数据已经同步过来了

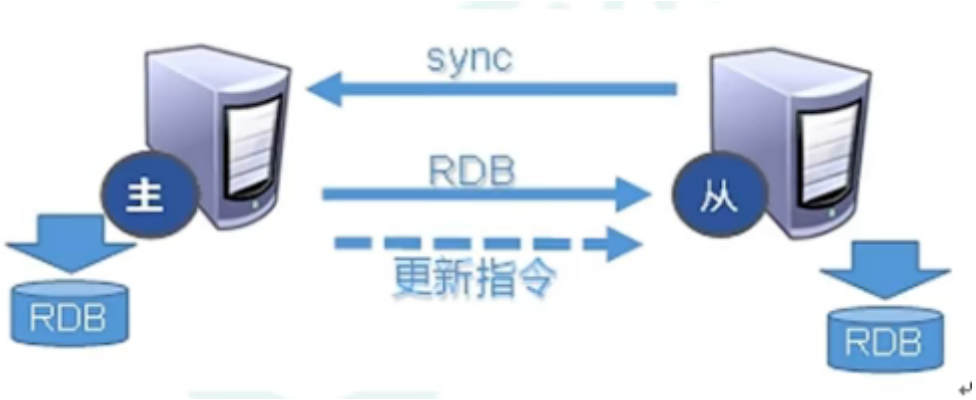
```
192.168.200.129:6380> mget name age
1) "ready"
2) "30"
192.168.200.129:6380>
```

同样到slave2上也执行一下，效果如下

```
192.168.200.129:6381> mget name age
1) "ready"
2) "30"
192.168.200.129:6381>
```

10.3.3、主从复制原理

- slave启动成功连接到master后，会给master发送数据同步消息（发送sync命令）
- master接收到slave发来的数据同步消息后，把主服务器的数据进行持久化到rdb文件，同时会收集接收到的用于修改数据的命令，master将传rdb文件发送给你slave，完成一次完全同步
- 全量复制：而slave服务在接收到master发来的rdb文件后，将其存盘并加载到内存
- 增量复制：master继续将收集到的修改命令依次传给slave，完成同步
- 但是只要重新连接master，一次完全同步（全量复制）将会被自动执行



10.3.4、小结

主redis挂掉以后情况会如何？从机是上位还是原地待命？

主机挂掉后，从机会待命，小弟还是小弟，会等着大哥恢复，不会篡位。

从挂掉后又恢复了，会继续从主同步数据么？

会的，当从重启之后，会继续将中间缺失的数据同步过来。

info Replication：查看主从复制信息

上面已经演示过了，主、从上都可以执行，用来查看主从信息。

10.2、常用的主从结构

10.2.1、一主二从

刚刚上面演示的就是一主二从，不过采用的都是配置文件的方式，实际上从机可以采用命令的方式配置，下面我们来演示一遍，大家看好了。

1）创建案例工作目录：master-slave

执行下面命令创建`/opt/master-slave`目录，本次所有操作，均在`master-slave`目录进行。

```
ps -ef | grep redis | awk -F" " '{print $2;}' | xargs kill -9 # 方便演示，停止所有的redis
mkdir /opt/master-slave
cd /opt/master-slave/
```

2）将redis.conf复制到master-slave目录

```
cp /opt/redis-6.2.1/redis.conf /opt/master-slave/
```

3）创建master的配置文件：redis-6379.conf

在/opt/master-slave目录创建`redis-6379.conf`文件，内容如下，注意`192.168.200.129`是这个测试机器的ip，大家需要替换为自己的

```
include /opt/master-slave/redis.conf
daemonize yes
bind 192.168.200.129
requirepass 123456
dir /opt/master-slave/

port 6379
dbfilename dump_6379.rdb
pidfile /var/run/redis_6379.pid
logfile /opt/master-slave/6379.log
```

4）创建slave1的配置文件：redis-6380.conf

在/opt/master-slave目录创建`redis-6380.conf`文件，内容如下，和上面master的类似，只是将6379换成6380了

```
include /opt/master-slave/redis.conf
daemonize yes
bind 192.168.200.129
requirepass 123456
dir /opt/master-slave/

port 6380
dbfilename dump_6380.rdb
pidfile /var/run/redis_6380.pid
logfile /opt/master-slave/6380.log
```

5）创建slave2的配置文件：redis-6381.conf

```
include /opt/master-slave/redis.conf
daemonize yes
bind 192.168.200.129
requirepass 123456
dir /opt/master-slave/

port 6381
dbfilename dump_6381.rdb
pidfile /var/run/redis_6381.pid
logfile /opt/master-slave/6381.log
```

6）启动master

```
redis-server /opt/master-slave/redis-6379.conf
```

7）启动slave1

```
redis-server /opt/master-slave/redis-6380.conf
```

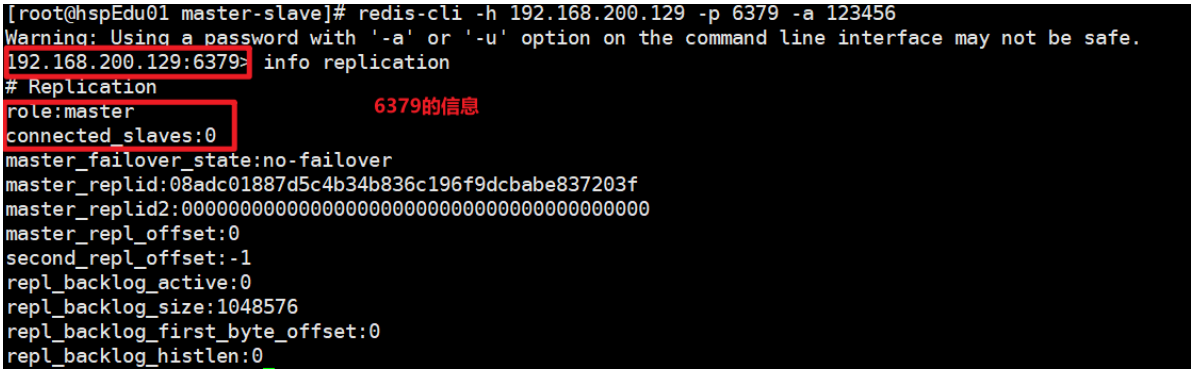
8）启动slave2

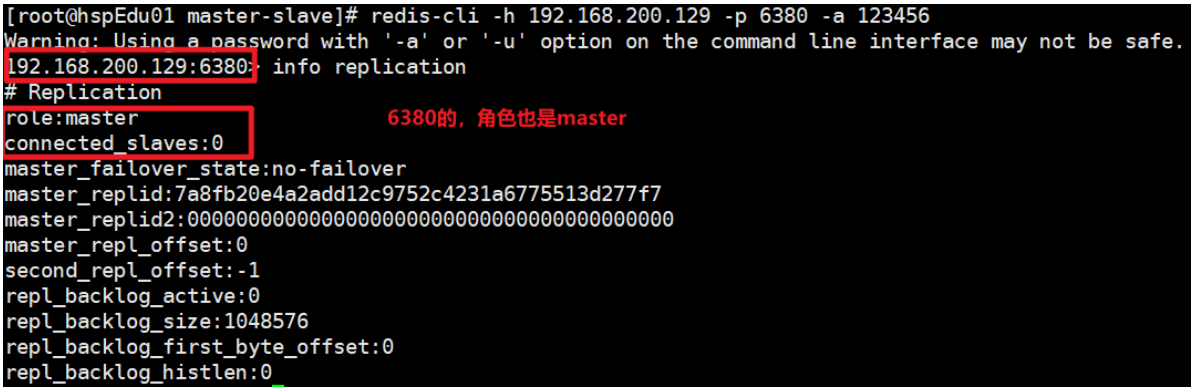
```
redis-server /opt/master-slave/redis-6381.conf
```

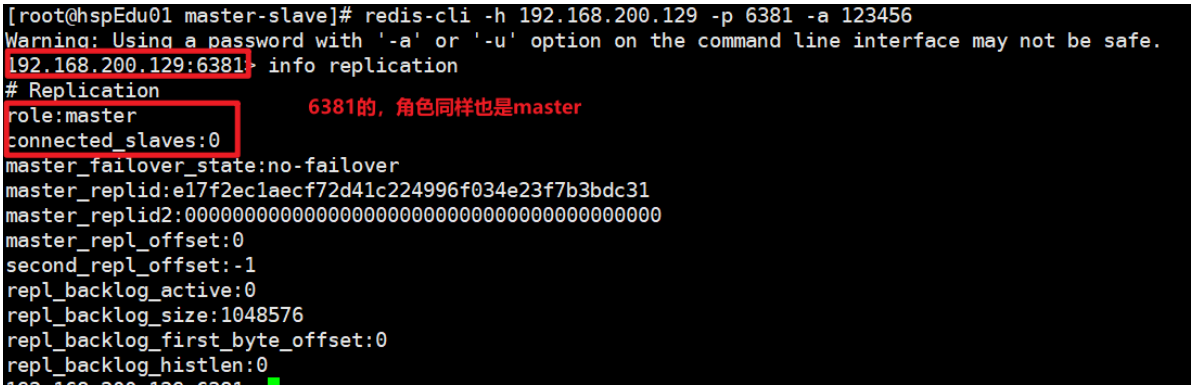
9）分别登陆3台机器，查看各自主从信息

本次我们并没有在slave1和slave2的配置文件通过`slaveof`命令配置主从信息，所以目前3台机器的角色都是master。

分别登陆对3个redis，然后用`info replication`命令看下3个的主从信息，如下：







下面我们将通过控制台命令来指定slave1和slave2的为master的从库。

10）配置slave1为master的从库

（1）执行下面命令，连接slave1

```
redis-cli -h 192.168.200.129 -p 6380 -a 123456
```

（2）执行下面命令，设置master的密码

由于master需要密码，所以在slave1中需要指定master的密码，否则无法同步数据。

```
config set masterauth 123456
```

（3）执行下面命令，指定slave1的作为master的从机

```
slaveof 192.168.200.129 6379
```

（4）如下，使用`info replication`查看下slave1的主从信息

```
192.168.200.129:6381> info replication
# Replication
role:slave
master_host:192.168.200.129
master_port:6379
master_link_status:up
master_last_io_seconds_ago:7
master_sync_in_progress:0
slave_repl_offset:280
slave_priority:100
slave_read_only:1
connected_slaves:0
master_failover_state:no-failover
master_replid:c7d54b37ad90159f8f02735b525e11af4c228a76
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:280
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:267
repl_backlog_histlen:14
```

11）配置slave2为master的从库

（1）执行下面命令，连接slave2

```
redis-cli -h 192.168.200.129 -p 6381 -a 123456
```

（2）执行下面命令，设置master的密码

由于master需要密码，所以在slave2中需要指定master的密码，否则无法同步数据。

```
config set masterauth 123456
```

（3）执行下面命令，指定slave2的作为master的从机

```
slaveof 192.168.200.129 6379
```

（4）如下，使用`info replication`查看下slave2的主从信息

```
192.168.200.129:6381> info replication
# Replication
role:slave
master_host:192.168.200.129
master_port:6379
master_link_status:up
master_last_io_seconds_ago:7
master_sync_in_progress:0
slave_repl_offset:280
slave_priority:100
slave_read_only:1
connected_slaves:0
master_failover_state:no-failover
master_replid:c7d54b37ad90159f8f02735b525e11af4c228a76
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:280
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:267
repl_backlog_histlen:14
```

12）再来看看master的主从信息

```
[root@hspEdu01 ~]# redis-cli -h 192.168.200.129 -p 6379 -a 123456
192.168.200.129:6379> info replication
```

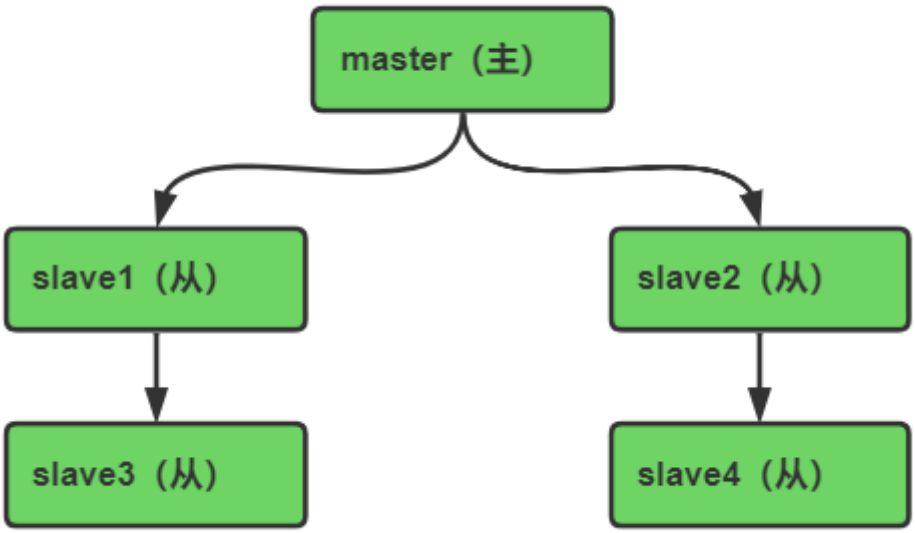
```
192.168.200.129:6379>
[root@hspEdu01 ~]# redis-cli -h 192.168.200.129 -p 6379 -a 123456   连接master
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
192.168.200.129:6379> info replication   查看主从信息
# Replication
role:master
connected_slaves:2                                                  主从信息
slave0:ip=192.168.200.129,port=6380,state=online,offset=532,lag=0
slave1:ip=192.168.200.129,port=6381,state=online,offset=532,lag=0
master_failover_state:no-failover
master_replid:c7d54b37ad90159f8f02735b525e11af4c228a76
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:532
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:532
```

注意：通过`slaveof`命令指定主从的方式，slave重启之后主从配置会失效，所以，重启后需要在slave上重新通过`slaveof`命令进行设置，这个不要忘记了。

中途通过`slaveof`变更转向，本地的数据会被清除，会从新的master重新同步数据。

10.2.2、薪火相传

若master下面挂很多slave，master会有压力，实际上slave下面也可以挂slave，如下图，配置这里就不演示了，和上面的类似。



10.2.3、反客为主

当master挂掉之后，我们可以从slave中选择一个作为主机。

比如我们想让slave1作为主机，那么可以在slave1上执行下面的命令就可以了。

```
slaveof no one
```

此时slave1就变成主机了，然后再去其他slave上面执行`slaveof`命令将其挂在slave1上。

这种主备切换有个缺点：需要手动去执行命令去操作，不是太方便。

下面来介绍另外一种方式：哨兵模式，主挂掉之后，自动从slave中选举一个作为主机，自动实现故障转移。

10.3、哨兵(Sentinel)模式

10.3.1、什么是哨兵模式?

反客为主的自动版，能够自动监控master是否发生故障，如果故障了会根据投票数从slave中挑选一个作为master，其他的slave会自动转向同步新的master，实现故障自动转义。

10.3.2、原理

sentinel会按照指定的频率给master发送ping请求，看看master是否还活着，若master在指定时间内未正常响应sentinel发送的ping请求，sentinel则认为master挂掉了，但是这种情况存在误判的可能，比如：可能master并没有挂，只是sentinel和master之间的网络不通导致，导致ping失败。

为了避免误判，通常会启动多个sentinel，一般是奇数个，比如3个，那么可以指定当有多个sentinel都觉得master挂掉了，此时才断定master真的挂掉了，通常这个值设置为sentinel的一半，比如sentinel的数量是3个，那么这个量就可以设置为2个。

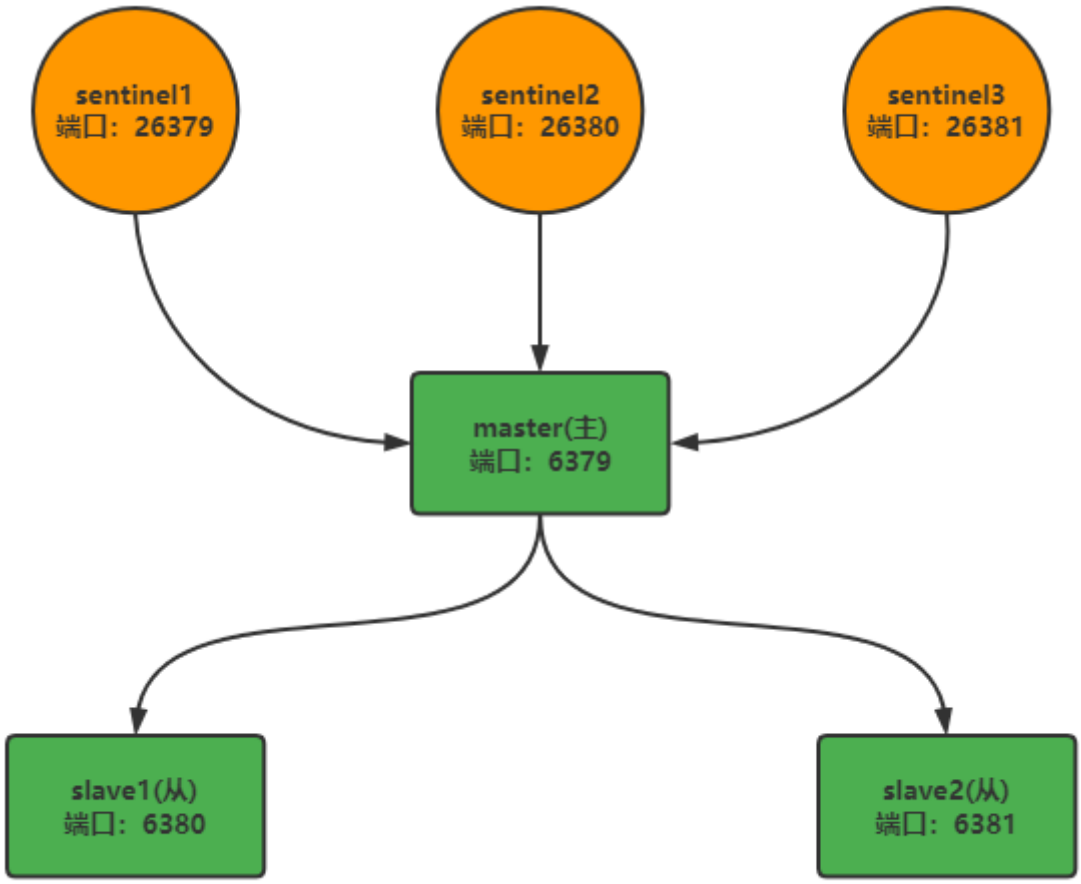
当多个sentinel经过判定，断定master确实挂掉了，接下来sentinel会进行故障转移：会从slave中投票选出一个服务器，将其升级为新的主服务器， 并让失效主服务器的其他从服务器slaveof指向新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。

10.3.3、怎么玩?

1）需求：配置1主2从3个哨兵

下面我们来实现1主2从3个sentinel的配置，当从的挂掉之后，要求最少有2个sentinel认为主的挂掉了，才进行故障转移。

为了方便，我们在一台机器上进行模拟，我的机器ip是：192.168.200.129，通过端口来区分6个不同的节点（1个master、2个slave、3个sentinel），节点配置信息如下



2）创建案例工作目录：sentinel

执行下面命令创建`/opt/sentinel`目录，本次所有操作，均在`sentinel`目录进行。

```
 # 方便演示，停止所有的redis
ps -ef | grep redis | awk -F" " '{print $2;}' | xargs kill -9
mkdir /opt/sentinel
cd /opt/sentinel/
```

3）将redis.conf复制到sentinel目录

> redis.conf 是redis默认配置文件

```
cp /opt/redis-6.2.1/redis.conf /opt/sentinel/
```

4）创建master的配置文件：redis-6379.conf

在/opt/sentinel目录创建`redis-6379.conf`文件，内容如下，注意`192.168.200.129`是这个测试机器的ip，大家需要替换为自己的

```
include /opt/sentinel/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/sentinel/

port 6379
dbfilename dump_6379.rdb
pidfile /var/run/redis_6379.pid
logfile "./6379.log"
```

5）创建slave1的配置文件：redis-6380.conf

在/opt/sentinel目录创建`redis-6380.conf`文件，内容如下，和上面master的类似，只是将6379换成6380了

```
include /opt/sentinel/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/sentinel/

port 6380
dbfilename dump_6380.rdb
pidfile /var/run/redis_6380.pid
logfile "./6380.log"
```

6）创建slave2的配置文件：redis-6381.conf

在/opt/sentinel目录创建`redis-6381.conf`文件，内容如下

```
include /opt/sentinel/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/sentinel/

port 6381
dbfilename dump_6381.rdb
pidfile /var/run/redis_6381.pid
logfile "./6381.log"
```

7）启动master、slave1、slave2

```
redis-server /opt/sentinel/redis-6379.conf
redis-server /opt/sentinel/redis-6380.conf
redis-server /opt/sentinel/redis-6381.conf
```

8）配置slave1为master的从库

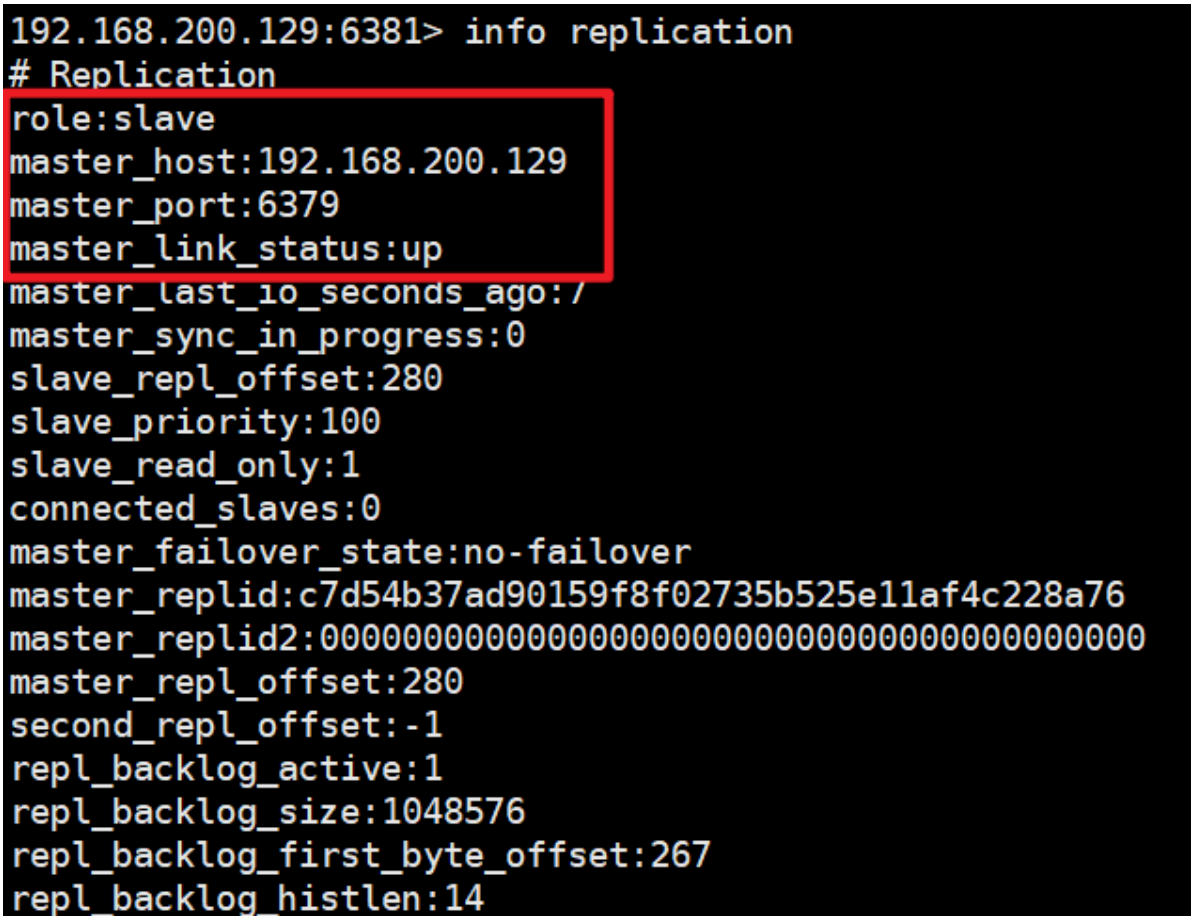
（1）执行下面命令，连接slave1

```
redis-cli -h 192.168.200.129 -p 6380
```

（2）执行下面命令，指定slave1的作为master的从机

```
slaveof 192.168.200.129 6379
```

（3）如下，使用`info replication`查看下slave1的主从信息



11）配置slave2为master的从库

（1）执行下面命令，连接slave2

```
redis-cli -h 192.168.200.129 -p 6381
```

（2）执行下面命令，指定slave2的作为master的从机

```
slaveof 192.168.200.129 6379
```

（3）如下，使用`info replication`查看下slave2的主从信息

```
192.168.200.129:6381> info replication
# Replication
role:slave
master_host:192.168.200.129
master_port:6379
master_link_status:up
master_last_io_seconds_ago:7
master_sync_in_progress:0
slave_repl_offset:280
slave_priority:100
slave_read_only:1
connected_slaves:0
master_failover_state:no-failover
master_replid:c7d54b37ad90159f8f02735b525e11af4c228a76
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:280
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:267
repl_backlog_histlen:14
```

12）验证主从复制是否正常

运行下面命令，连接master

```
redis-cli -h 192.168.200.129 -p 6379
```

运行下面命令，查看master主从信息

```
info replication
```

```
192.168.200.129:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=192.168.200.129,port=6380,state=online,offset=140,lag=1
slave1:ip=192.168.200.129,port=6381,state=online,offset=140,lag=1
master_failover_state:no-failover
master_replid:f53c3bf4aeb5f4968b35c12f087d2b89fd7540c5
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:140
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
```

可以看到2个slave

在master中执行下面命令，写入数据

```
flushdb
set name jack
```

如下，查看slave1中name的值

```
192.168.200.129:6380> get name
"jack"
```

如下，查看slave2中name的值

```
192.168.200.129:6381> get name
"jack"
```

数据一致，说明同步正常。

13）创建sentinel1的配置文件：sentinel-26379.conf

在/opt/sentinel目录创建`sentinel-26379.conf`文件，内容如下

```
# 配置文件目录
dir /opt/sentinel/
# 日志文件位置
logfile "./sentinel-26379.log"
# pid文件
pidfile /var/run/sentinel_26379.pid
# 是否后台运行
daemonize yes
# 端口
port 26379
# 监控主服务器master的名字：mymaster，IP：192.168.200.129，port：6379，最后的数字2表示当Sentinel集群中有2个Sentinel认为master存在故障不可用，则进行自动故障转移
sentinel monitor mymaster 192.168.200.129 6379 2
# master响应超时时间（毫秒），Sentinel会向master发送ping来确认master，如果在20秒内，ping不通master，则主观认为master不可用
sentinel down-after-milliseconds mymaster 60000
# 故障转移超时时间（毫秒），如果3分钟内没有完成故障转移操作，则视为转移失败
sentinel failover-timeout mymaster 180000
# 故障转移之后，进行新的主从复制，配置项指定了最多有多少个slave对新的master进行同步，那可以理解为1是串行复制，大于1是并行复制
sentinel parallel-syncs mymaster 1
# 指定mymaster主的密码（没有就不指定）
# sentinel auth-pass mymaster 123456
```

14）创建sentinel2的配置文件：sentinel-26380.conf

在/opt/sentinel目录创建`sentinel-26380.conf`文件，内容如下

```
# 配置文件目录
dir /opt/sentinel/
# 日志文件位置
logfile "./sentinel-26380.log"
# pid文件
pidfile /var/run/sentinel_26380.pid
# 是否后台运行
daemonize yes
# 端口
port 26380
# 监控主服务器master的名字：mymaster，IP：192.168.200.129，port：6379，最后的数字2表示当Sentinel集群中有2个Sentinel认为master存在故障不可用，则进行自动故障转移
sentinel monitor mymaster 192.168.200.129 6379 2
```

```
# master响应超时时间（毫秒），Sentinel会向master发送ping来确认master，如果在20秒内，ping不通master，则主观认为master不可用
sentinel down-after-milliseconds mymaster 60000
# 故障转移超时时间（毫秒），如果3分钟内没有完成故障转移操作，则视为转移失败
sentinel failover-timeout mymaster 180000
# 故障转移之后，进行新的主从复制，配置项指定了最多有多少个slave对新的master进行同步，那可以理解为1是串行复制，大于1是并行复制
sentinel parallel-syncs mymaster 1
# 指定mymaster主的密码（没有就不指定）
# sentinel auth-pass mymaster 123456
```

15）创建sentinel3的配置文件：sentinel-26381.conf

在/opt/sentinel目录创建`sentinel-26381.conf`文件，内容如下

```
# 配置文件目录
dir /opt/sentinel/
# 日志文件位置
logfile "./sentinel-26381.log"
# pid文件
pidfile /var/run/sentinel_26381.pid
# 是否后台运行
daemonize yes
# 端口
port 26381
# 监控主服务器master的名字：mymaster，IP：192.168.200.129，port：6379，最后的数字2表示当Sentinel集群中有2个Sentinel认为master存在故障不可用，则进行自动故障转移
sentinel monitor mymaster 192.168.200.129 6379 2
# master响应超时时间（毫秒），Sentinel会向master发送ping来确认master，如果在20秒内，ping不通master，则主观认为master不可用
sentinel down-after-milliseconds mymaster 60000
# 故障转移超时时间（毫秒），如果3分钟内没有完成故障转移操作，则视为转移失败
sentinel failover-timeout mymaster 180000
# 故障转移之后，进行新的主从复制，配置项指定了最多有多少个slave对新的master进行同步，那可以理解为1是串行复制，大于1是并行复制
sentinel parallel-syncs mymaster 1
# 指定mymaster主的密码（没有就不指定）
# sentinel auth-pass mymaster 123456
```

16）启动3个sentinel

启动sentinel有2种方式

```
方式1：redis-server sentinel.conf --sentinel
方式2：redis-sentinel sentinel.conf
```

下面我们使用方式2来启动3个sentinel

```
/opt/redis-6.2.1/src/redis-sentinel /opt/sentinel/sentinel-26379.conf
/opt/redis-6.2.1/src/redis-sentinel /opt/sentinel/sentinel-26380.conf
/opt/redis-6.2.1/src/redis-sentinel /opt/sentinel/sentinel-26381.conf
```

17）分别查看3个sentinel的信息

分别对3个sentinel执行下面命令，查看每个sentinel的信息

```
redis-cli -p sentinel的端口
info sentinel
```

`sentinel1`的信息如下，其他2个sentinel的信息这里就不列了，大家自己去看一下

```
[root@hspEdu01 sentinel]# redis-cli -p 26379
127.0.0.1:26379> info sentinel
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=mymaster,status=ok,address=192.168.200.129:6379,slaves=2,sentinels=3
127.0.0.1:26379>
```

被监控的第1个主机，名称：mymaster，address：主ip:端口，
slaves:master上面挂的slave的数量，sentinels：表示有几个哨兵（3个）

18）验证故障自动转移是否成功

step1：在master中执行下面命令，停止master

```
192.168.200.129:6379> shutdown
```

step2：等待2分钟，等待完成故障转移

sentinel中我们配置`down-after-milliseconds`的值是60秒，表示判断主机下线时间是60秒，所以我们等2分钟，让系统先自动完成故障转移。

step3：查看slave1的主从信息，如下

> 使用`info replication`命令查看主从信息

```
192.168.200.129:6380> info replication
# Replication
role:slave
master_host:192.168.200.129
master_port:6381
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_repl_offset:355220
slave_priority:100
slave_read_only:1
connected_slaves:0
master_failover_state:no-failover
master_replid:8bbee785fd5c348c9c230d38421b5a2370f234b6
master_replid2:faa729b8da623ce8d8817358f9eed3477ce8ef26
master_repl_offset:355220
second_repl_offset:337686
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:355220
192.168.200.129:6380>
```

角色依然是slave
master变了，变成6381了，这个端口是slave2的端口

step4：查看slave2的主从信息，如下

> slave2变成master了，且slave2变成slave1的从库了，完成了故障转移。

step5：下面验证下slave1和slave2是否同步

在slave2中执行下面命令

```
192.168.200.129:6381> set address china
OK
```

在slave1中执行下面命令，查询一下address的值，效果如下，说明slave2和slave1同步正常

```
192.168.200.129:6380> get address
"china"
```

19）恢复旧的master自动俯首称臣

当旧的master恢复之后，会自动挂在新的master下面，咱们来验证下是不是这样的。

step1：执行下面命令，启动旧的master

```
redis-server /opt/sentinel/redis-6379.conf
```

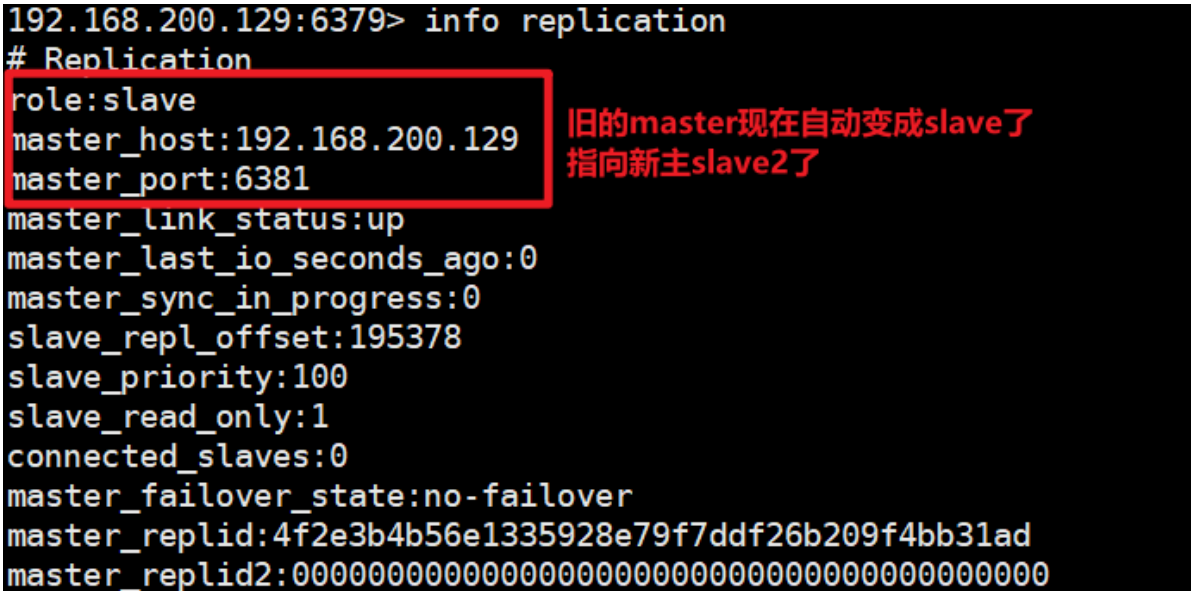
step2：执行下面命令，连接旧的master

```
redis-cli -h 192.168.200.129 -p 6379
```

step3：执行下面命令，查看其主从信息

```
info replication
```

效果如下，确实和期望的一致。



10.3.4、更多Sentinel介绍

关于sentinel更多信息，见：Redis-Sentinel

10.3.5、SpringBoot整合Sentinel模式

1）引入redis的maven配置

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2）application.properties中配置redis sentinel信息

```
# redis sentinel主服务名称，这个可不是随便写的哦，来源于：sentinel配置文件中sentinel
monitor后面跟的那个名称
spring.redis.sentinel.master=mymaster
# sentinel节点列表(host:port)，多个之间用逗号隔开
spring.redis.sentinel.nodes=192.168.200.129:26379,192.168.200.129:26380,192.168.
200.129:26381
# sentinel密码
#spring.redis.sentinel.password=
# 连接超时时间（毫秒）
spring.redis.timeout=60000
# Redis默认情况下有16个分片，这里配置具体使用的分片，默认是0
spring.redis.database=0
```

3）使用RedisTemplate工具类操作redis

springboot中使用RedisTemplate来操作redis，需要在我们的bean中注入这个对象，代码如下：

```
@Autowired
private RedisTemplate<String, String> redisTemplate;

// 用下面5个对象来操作对应的类型
this.redisTemplate.opsForValue(); //提供了操作string类型的所有方法
this.redisTemplate.opsForList(); // 提供了操作list类型的所有方法
this.redisTemplate.opsForSet(); //提供了操作set的所有方法
this.redisTemplate.opsForHash(); //提供了操作hash表的所有方法
this.redisTemplate.opsForZSet(); //提供了操作zset的所有方法
```

2）RedisTemplate示例代码

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataAccessException;
import org.springframework.data.redis.connection.RedisConnection;
import org.springframework.data.redis.core.RedisCallback;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

@RestController
@RequestMapping("/redis")
public class RedisController {
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**
     * string测试
     *
     * @return
     */
    @RequestMapping("/stringTest")
    public String stringTest() {
        this.redisTemplate.delete("name");
        this.redisTemplate.opsForValue().set("name", "路人");
        String name = this.redisTemplate.opsForValue().get("name");
        return name;
    }

    /**
     * list测试
     *
     * @return
     */
    @RequestMapping("/listTest")
    public List<String> listTest() {
        this.redisTemplate.delete("names");
        this.redisTemplate.opsForList().rightPushAll("names", "刘德华", "张学友",
"郭富城", "黎明");
```

```
        List<String> courses = this.redisTemplate.opsForList().range("names", 0,
-1);
        return courses;
    }

    /**
     * set类型测试
     *
     * @return
     */
    @RequestMapping("setTest")
    public Set<String> setTest() {
        this.redisTemplate.delete("courses");
        this.redisTemplate.opsForSet().add("courses", "java", "spring",
"springboot");
        Set<String> courses = this.redisTemplate.opsForSet().members("courses");
        return courses;
    }

    /**
     * hash表测试
     *
     * @return
     */
    @RequestMapping("hashTest")
    public Map<Object, Object> hashTest() {
        this.redisTemplate.delete("userMap");
        Map<String, String> map = new HashMap<>();
        map.put("name", "路人");
        map.put("age", "30");
        this.redisTemplate.opsForHash().putAll("userMap", map);

        Map<Object, Object> userMap =
this.redisTemplate.opsForHash().entries("userMap");
        return userMap;
    }

    /**
     * zset测试
     *
     * @return
     */
    @RequestMapping("zsetTest")
    public Set<String> zsetTest() {
        this.redisTemplate.delete("languages");

        this.redisTemplate.opsForZSet().add("languages", "java", 100d);
        this.redisTemplate.opsForZSet().add("languages", "c", 95d);
        this.redisTemplate.opsForZSet().add("languages", "php", 70);

        Set<String> languages =
this.redisTemplate.opsForZSet().range("languages", 0, -1);
        return languages;
    }

    /**
     * 查看redis机器信息
     *
```

```
     * @return
     */
    @RequestMapping(value = "/info", produces = MediaType.TEXT_PLAIN_VALUE)
    public String info() {
        Object obj = this.redisTemplate.execute(new RedisCallback<Object>() {
            @Override
            public Object doInRedis(RedisConnection connection) throws
DataAccessException {
                return connection.execute("info");
            }
        });
        return obj.toString();
    }
}
```

11、redis集群（Cluster）

11.1、存在的问题

单台redis容量限制，如何进行扩容？继续加内存、加硬件么？

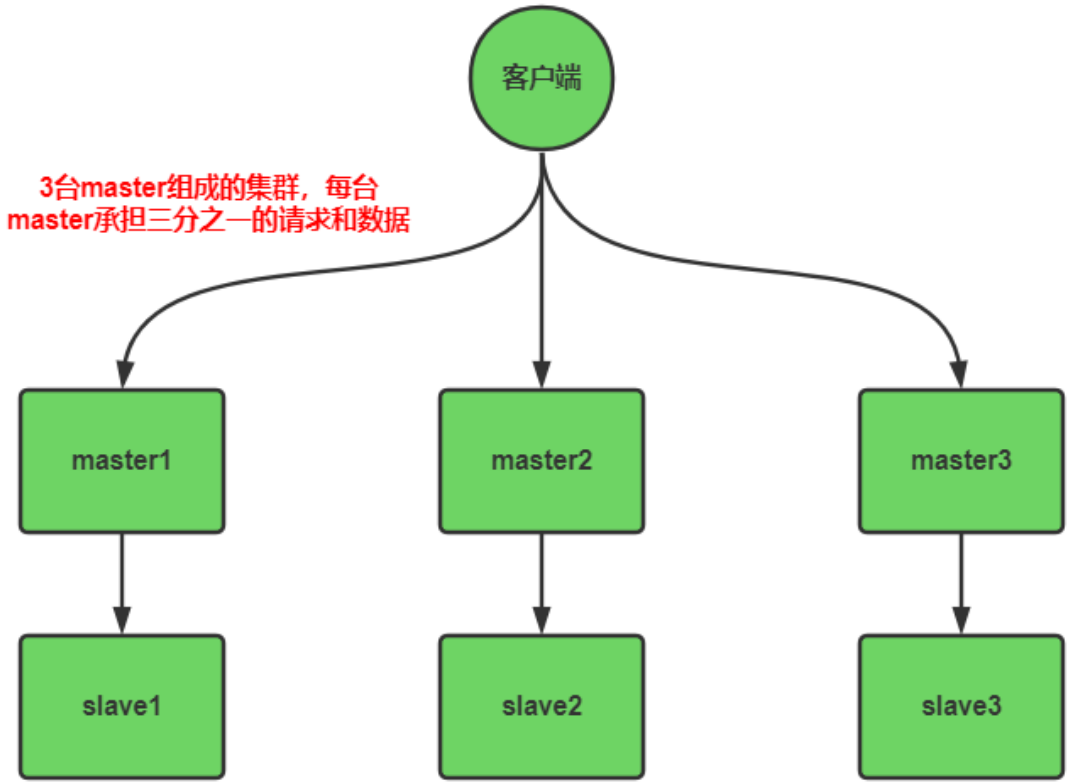
单台redis并发写量太大有性能瓶颈，如何解决？

redis3.0中提供了集群可以解决这些问题。

11.2、什么是集群

redis集群是对redis的水平扩容，即启动N个redis节点，将整个数据分布存储在这个N个节点中，每个节点存储总数据的1/N。

如下图：由3台master和3台slave组成的redis集群，每台master承接客户端三分之一请求和写入的数据，当master挂掉后，slave会自动替代master，做到高可用。

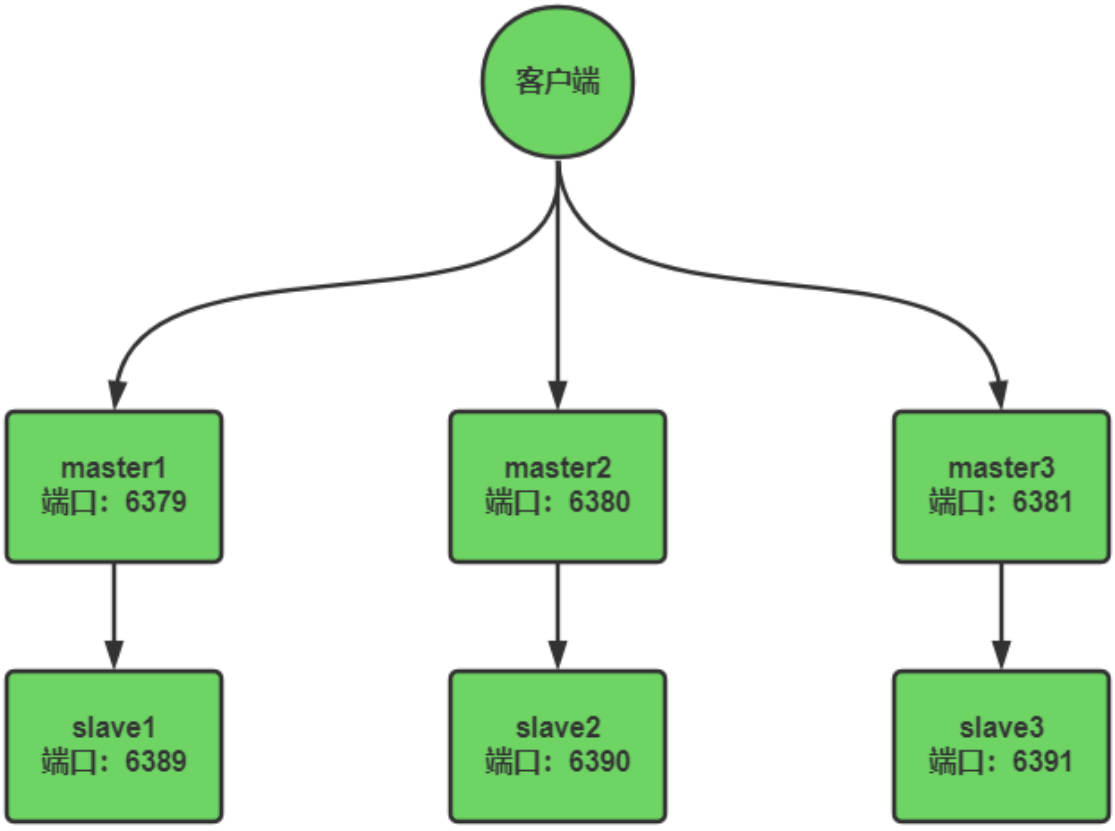


11.3、集群如何配置?

1）需求：配置3主3从集群

下面我们来配置一个3主3从的集群，每个主下面挂一个slave，master挂掉后，slave会被提升为master。

为了方便，我们在一台机器上进行模拟，我的机器ip是：192.168.200.129，通过端口来区分6个不同的节点，配置信息如下



2）创建案例工作目录：cluster

执行下面命令创建`/opt/cluster`目录，本次所有操作，均在`cluster`目录进行。

```
 # 方便演示，停止所有的redis
ps -ef | grep redis | awk -F" " '{print $2;}' | xargs kill -9
mkdir /opt/cluster
cd /opt/cluster/
```

3）将redis.conf复制到cluster目录

> redis.conf 是redis默认配置文件

```
cp /opt/redis-6.2.1/redis.conf /opt/cluster/
```

4）创建master1的配置文件：redis-6379.conf

在/opt/cluster目录创建`redis-6379.conf`文件，内容如下，注意`192.168.200.129`是这个测试机器的ip，大家需要替换为自己的

```
include /opt/cluster/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/cluster/

port 6379
dbfilename dump_6379.rdb
pidfile /var/run/redis_6379.pid
logfile "./6379.log"

# 开启集群设置
cluster-enabled yes
# 设置节点配置文件
cluster-config-file node-6379.conf
# 设置节点失联时间，超过该时间(毫秒)，集群自动进行主从切换
cluster-node-timeout 15000
```

5）创建master2的配置文件：redis-6380.conf

在/opt/cluster目录创建`redis-6380.conf`文件，内容如下，和上面master的类似，只是将6379换成6380了

```
include /opt/cluster/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/cluster/

port 6380
dbfilename dump_6380.rdb
pidfile /var/run/redis_6380.pid
logfile "./6380.log"
```

```
# 开启集群设置
cluster-enabled yes
# 设置节点配置文件
cluster-config-file node-6380.conf
# 设置节点失联时间，超过该时间(毫秒)，集群自动进行主从切换
cluster-node-timeout 15000
```

6）创建master3的配置文件：redis-6381.conf

在/opt/cluster目录创建`redis-6381.conf`文件，内容如下

```
include /opt/cluster/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/cluster/

port 6381
dbfilename dump_6381.rdb
pidfile /var/run/redis_6381.pid
logfile "./6381.log"

# 开启集群设置
cluster-enabled yes
# 设置节点配置文件
cluster-config-file node-6381.conf
# 设置节点失联时间，超过该时间(毫秒)，集群自动进行主从切换
cluster-node-timeout 15000
```

4）创建slave1的配置文件：redis-6389.conf

在/opt/cluster目录创建`redis-6389.conf`文件，内容如下

```
include /opt/cluster/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/cluster/

port 6389
dbfilename dump_6389.rdb
pidfile /var/run/redis_6389.pid
logfile "./6389.log"

# 开启集群设置
cluster-enabled yes
# 设置节点配置文件
cluster-config-file node-6389.conf
# 设置节点失联时间，超过该时间(毫秒)，集群自动进行主从切换
cluster-node-timeout 15000
```

5）创建slave2的配置文件：redis-6390.conf

在/opt/cluster目录创建`redis-6390.conf`文件，内容如下

```
include /opt/cluster/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/cluster/

port 6390
dbfilename dump_6390.rdb
pidfile /var/run/redis_6390.pid
logfile "./6390.log"

# 开启集群设置
cluster-enabled yes
# 设置节点配置文件
cluster-config-file node-6390.conf
# 设置节点失联时间，超过该时间(毫秒)，集群自动进行主从切换
cluster-node-timeout 15000
```

6）创建slave3的配置文件：redis-6391.conf

在/opt/cluster目录创建`redis-6391.conf`文件，内容如下

```
include /opt/cluster/redis.conf
daemonize yes
bind 192.168.200.129
dir /opt/cluster/

port 6391
dbfilename dump_6391.rdb
pidfile /var/run/redis_6391.pid
logfile "./6391.log"

# 开启集群设置
cluster-enabled yes
# 设置节点配置文件
cluster-config-file node-6391.conf
# 设置节点失联时间，超过该时间(毫秒)，集群自动进行主从切换
cluster-node-timeout 15000
```

7）启动master、slave1、slave2

```
 # 方便演示，停止所有的redis
ps -ef | grep redis | awk -F" " '{print $2;}' | xargs kill -9
# 下面启动6个redis
redis-server /opt/cluster/redis-6379.conf
redis-server /opt/cluster/redis-6380.conf
redis-server /opt/cluster/redis-6381.conf
redis-server /opt/cluster/redis-6389.conf
redis-server /opt/cluster/redis-6390.conf
redis-server /opt/cluster/redis-6391.conf
```

8）查看6个redis的启动情况

```
ps -ef | grep redis
```

```
[root@hspEdu01 cluster]# ps -ef | grep redis
root      34663      1  0 18:09 ?        00:00:00 redis-server 192.168.200.129:6379 [cluster]
root      34669      1  0 18:09 ?        00:00:00 redis-server 192.168.200.129:6380 [cluster]
root      34675      1  0 18:09 ?        00:00:00 redis-server 192.168.200.129:6381 [cluster]
root      34681      1  0 18:09 ?        00:00:00 redis-server 192.168.200.129:6389 [cluster]
root      34686      1  0 18:09 ?        00:00:00 redis-server 192.168.200.129:6390 [cluster]
root      34693      1  0 18:09 ?        00:00:00 redis-server 192.168.200.129:6391 [cluster]
root      34777  34293  0 18:13 pts/0    00:00:00 grep --color=auto redis
```

9）确保node-xxxx.conf文件已正常生成

稍后我们会将6个实例合并到一个集群，在组合之前，我们要确保6个redis实例启动后，nodes-xxxx.conf文件都生成正常，如下，`/opt/cluster`目录中确实都生成成功了

```
[root@hspEdu01 cluster]# cd /opt/cluster/
[root@hspEdu01 cluster]# ll
总用量 164
-rw-r--r--. 1 root root  2381 4月  17 18:09 6379.log
-rw-r--r--. 1 root root  1193 4月  17 18:09 6380.log
-rw-r--r--. 1 root root  1193 4月  17 18:09 6381.log
-rw-r--r--. 1 root root  1193 4月  17 18:09 6389.log
-rw-r--r--. 1 root root  1193 4月  17 18:09 6390.log
-rw-r--r--. 1 root root  1193 4月  17 18:09 6391.log
-rw-r--r--. 1 root root   114 4月  17 17:42 node-6379.conf
-rw-r--r--. 1 root root   114 4月  17 18:09 node-6380.conf
-rw-r--r--. 1 root root   114 4月  17 18:09 node-6381.conf
-rw-r--r--. 1 root root   114 4月  17 18:09 node-6389.conf
-rw-r--r--. 1 root root   114 4月  17 18:09 node-6390.conf
-rw-r--r--. 1 root root   114 4月  17 18:09 node-6391.conf
-rw-r--r--. 1 root root   391 4月  17 17:42 redis-6379.conf
-rw-r--r--. 1 root root   391 4月  17 18:00 redis-6380.conf
-rw-r--r--. 1 root root   391 4月  17 18:00 redis-6381.conf
-rw-r--r--. 1 root root   391 4月  17 18:06 redis-6389.conf
-rw-r--r--. 1 root root   391 4月  17 18:07 redis-6390.conf
-rw-r--r--. 1 root root   391 4月  17 18:07 redis-6391.conf
-rw-r--r--. 1 root root 92222 4月  17 17:36 redis.conf
```

正常生成了node文件

10）将6个节点合成一个集群

执行下面命令，将6个redis合体

```
/opt/redis-6.2.1/src/redis-cli --cluster create --cluster-replicas 1
192.168.200.129:6379 192.168.200.129:6380 192.168.200.129:6381
192.168.200.129:6389 192.168.200.129:6390 192.168.200.129:6391
```

> - 合体的命令后面会跟上所有节点的ip:port列表，多个之间用空格隔开，注意ip不要写127.0.0.1，要写真实ip
> - --cluster-replicas 1：表示采用最简单的方式配置集群，即每个master配1个slave，6个节点就形成了3主3从

执行过程如下，期间会让我们确定是否同样这样的分配方式，输入：yes，然后等几秒，集群合体成功

```
[root@hspEdu01 src]# redis-cli --cluster create --cluster-replicas 1
192.168.200.129:6379 192.168.200.129:6380 192.168.200.129:6381
192.168.200.129:6389 192.168.200.129:6390 192.168.200.129:6391
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
```

```
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 192.168.200.129:6390 to 192.168.200.129:6379
Adding replica 192.168.200.129:6391 to 192.168.200.129:6380
Adding replica 192.168.200.129:6389 to 192.168.200.129:6381
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: ccf3abb791e026380ad3ad2a166aa788df738437 192.168.200.129:6379
   slots:[0-5460] (5461 slots) master
M: 3c372392d5a91dad64a6febadfe9524ea2cbd8c0 192.168.200.129:6380
   slots:[5461-10922] (5462 slots) master
M: 2c905be9c975be367bd66c962167beca1ef66af3 192.168.200.129:6381
   slots:[10923-16383] (5461 slots) master
S: 4a0f860081b969162767aac26801994de54d80a5 192.168.200.129:6389
   replicates ccf3abb791e026380ad3ad2a166aa788df738437
S: 62c9f37a362459c212e8af6dd744b6562f5fe6a7 192.168.200.129:6390
   replicates 3c372392d5a91dad64a6febadfe9524ea2cbd8c0
S: a2f89efc09681520f9d9502707b18e1f46a40b90 192.168.200.129:6391
   replicates 2c905be9c975be367bd66c962167beca1ef66af3
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
.
>>> Performing Cluster Check (using node 192.168.200.129:6379)
M: ccf3abb791e026380ad3ad2a166aa788df738437 192.168.200.129:6379
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: 2c905be9c975be367bd66c962167beca1ef66af3 192.168.200.129:6381
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: 62c9f37a362459c212e8af6dd744b6562f5fe6a7 192.168.200.129:6390
   slots: (0 slots) slave
   replicates 3c372392d5a91dad64a6febadfe9524ea2cbd8c0
M: 3c372392d5a91dad64a6febadfe9524ea2cbd8c0 192.168.200.129:6380
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 4a0f860081b969162767aac26801994de54d80a5 192.168.200.129:6389
   slots: (0 slots) slave
   replicates ccf3abb791e026380ad3ad2a166aa788df738437
S: a2f89efc09681520f9d9502707b18e1f46a40b90 192.168.200.129:6391
   slots: (0 slots) slave
   replicates 2c905be9c975be367bd66c962167beca1ef66af3
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

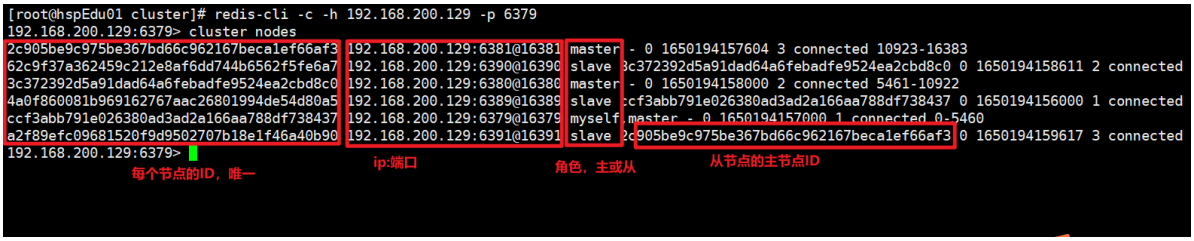
11）连接集群节点，查看集群信息：cluster nodes

需要使用`redis-cli -c`命令连接集群中6个节点中任何一个节点都可以，注意和之前的连接参数有点不同`redis-cli`命令后面多了一个`-c`参数，表示采用集群的方式连接，连上以后，然后使用`cluster nodes`可以查看集群节点信息，如下

```
192.168.200.129:6379> cluster nodes
2c905be9c975be367bd66c962167beca1ef66af3 192.168.200.129:6381@16381 master - 0
1650194157604 3 connected 10923-16383
62c9f37a362459c212e8af6dd744b6562f5fe6a7 192.168.200.129:6390@16390 slave
3c372392d5a91dad64a6febadfe9524ea2cbd8c0 0 1650194158611 2 connected
3c372392d5a91dad64a6febadfe9524ea2cbd8c0 192.168.200.129:6380@16380 master - 0
1650194158000 2 connected 5461-10922
4a0f860081b969162767aac26801994de54d80a5 192.168.200.129:6389@16389 slave
ccf3abb791e026380ad3ad2a166aa788df738437 0 1650194156000 1 connected
ccf3abb791e026380ad3ad2a166aa788df738437 192.168.200.129:6379@16379
myself,master - 0 1650194157000 1 connected 0-5460
a2f89efc09681520f9d9502707b18e1f46a40b90 192.168.200.129:6391@16391 slave
2c905be9c975be367bd66c962167beca1ef66af3 0 1650194159617 3 connected
192.168.200.129:6379>
```

如下图，对`cluster nodes`的结果做下解释，先看下红字的注释，集群中的每个节点都会生成一个ID，这个ID信息会被写到node-xxxx.conf文件中，为什么要生成id呢?

因为节点的ip和端口可能会发生变化，但是节点的ID是不会变的，其他节点可以通过其他节点的ID来认识各个节点。



12）验证集群数据的读写操作

如下，我们连接`6379`这个节点，然后执行一个set操作，效果如下，写入成功

```
[root@hspEdu01 cluster]# redis-cli -c -h 192.168.200.129 -p 6379
192.168.200.129:6379> set name ready
-> Redirected to slot [5798] located at 192.168.200.129:6380
OK
192.168.200.129:6380>
```

大家可能注意到了，我们明明在`6379`上操作的，但是请求被转发到了6380这个节点去处理了，这里就是我们后面要说的slot的知识了，先向后看。

11.4、redis集群如何分配这6个节点?

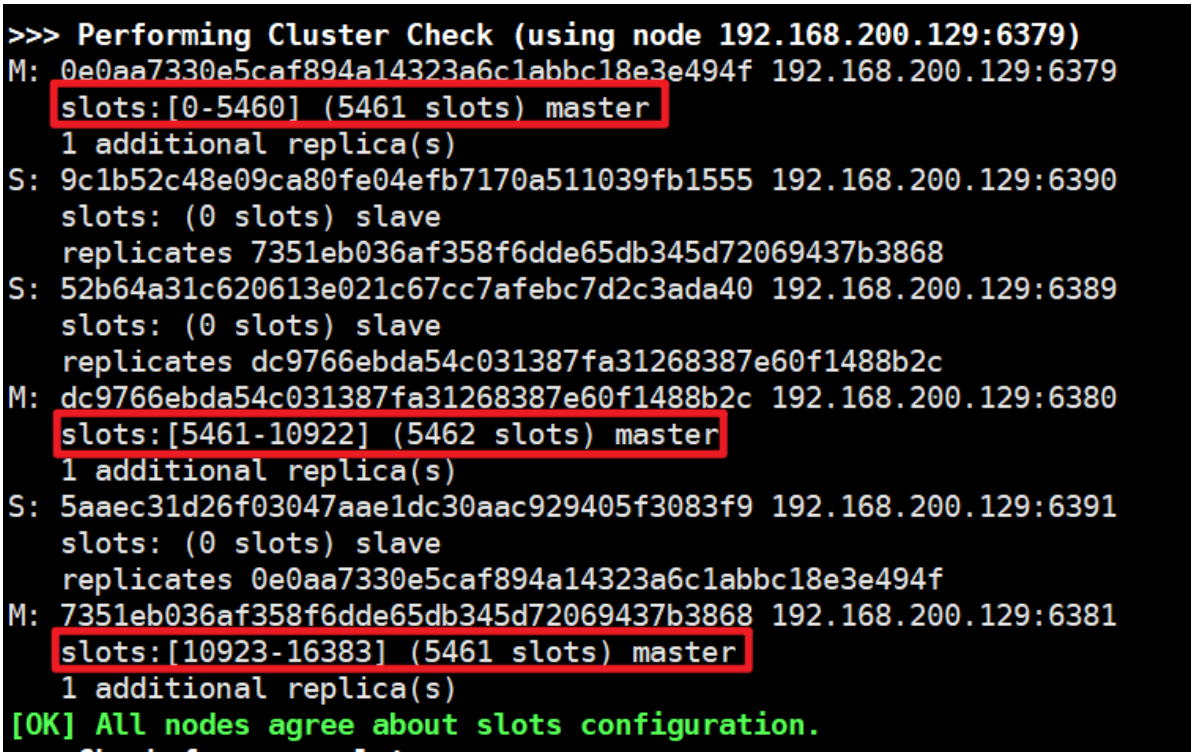
一个集群至少有3个主节点，因为新master的选举需要大于半数的集群master节点同意才能选举成功，如果只有两个master节点，当其中一个挂了，是达不到选举新master的条件的。

选项--cluster-replicas 1表示我们希望为集群中的每个主节点创建一个从节点。

分配原则尽量保证每个主库运行在不同的ip，每个主库和从库不在一个ip上，这样才能做到高可用。

11.5、什么是slots（槽）

如下图，咱们再来看看集群合并的过程中输出的一些信息



Redis集群内部划分了16384个slots（插槽），合并的时候，会将每个slots映射到一个master上面，比如上面3个master和slots的关系如下：

redis主节点	槽位范围
master1(端口：6379)	[0-5460]，插槽的位置从0开始的，0表示第1个插槽
master2(端口：6380)	[5460-10922]
master3(端口：6381)	[10923-16383]
slave1,slave2,slave3	从节点没有槽位，slave是用来对master做替补的

而数据库中的每个key都属于16384个slots中的其中1个，当通过key读写数据的时候，redis需要先根据key计算出key对应的slots，然后根据slots和master的映射关系找到对应的redis节点，key对应的数据就在这个节点上面。

集群中使用公式 `CRC16(key)%16384` 计算key属于哪个槽

11.6、在集群中录入值

在 `redis-cli` 每次录入、查询键值，redis都会计算key对应的插槽，如果不是当前redis节点的插槽，redis会报错，并告知应前往的redis实例地址和端口，效果如下，我们连接了6379这个实例来操作k1，这个节点发现k1的槽位在6381上面，返回了错误信息，怎么办呢?

```
[root@hspEdu01 cluster]# redis-cli -h 192.168.200.129 -p 6379
192.168.200.129:6379> set k1 v1
(error) MOVED 12706 192.168.200.129:6381
```

使用redis-cli客户端提供了-c参数可以解决这个问题，表示以集群方式执行，执行命令的时候当前节点处理不了的时候，会自动将请求重定向到目标节点，效果如下，被重定向到6381了

```
[root@hspEdu01 cluster]# redis-cli -c -h 192.168.200.129 -p 6379
192.168.200.129:6379> set k1 v1
-> Redirected to slot [12706] located at 192.168.200.129:6381
OK
192.168.200.129:6381>
```

同样，执行get会被重定向，效果如下

```
[root@hspEdu01 cluster]# redis-cli -c -h 192.168.200.129 -p 6379
192.168.200.129:6379> get k1
-> Redirected to slot [12706] located at 192.168.200.129:6381
"v1"
192.168.200.129:6381>
```

不在一个slot下面，不能使用mget、mset等多键操作，效果如下

```
192.168.200.129:6381> mset k1 v1 k2 v2
(error) CROSSSLOT Keys in request don't hash to the same slot
192.168.200.129:6381> mget k1 k2
(error) CROSSSLOT Keys in request don't hash to the same slot
```

可以通过{}来定义组的概念，从而使key中{}内相同的键值放到一个slot中去，效果如下

```
192.168.200.129:6381> mset k1{g1} v1 k2{g1} v2 k3{g1} v3
OK
192.168.200.129:6381> mget k1{g1} k2{g1} k3{g1}
1) "v1"
2) "v2"
3) "v3"
```

11.7、slot相关的一些命令

- cluster keyslot <key>：计算key对应的slot
- cluster coutkeysinslot <slot>：获取slot槽位中key的个数
- cluster getkeysinslot <slot> <count> 返回count个slot槽中的键

```
192.168.200.129:6381> cluster keyslot k1{g1}
(integer) 13519
192.168.200.129:6381> cluster countkeysinslot 13519
(integer) 3
192.168.200.129:6381> cluster getkeysinslot 13519 3
1) "k1{g1}"
2) "k2{g1}"
3) "k3{g1}"
```

11.8、故障恢复

如果主节点下线，从节点是否能够提升为主节点？注意：**要等15秒**

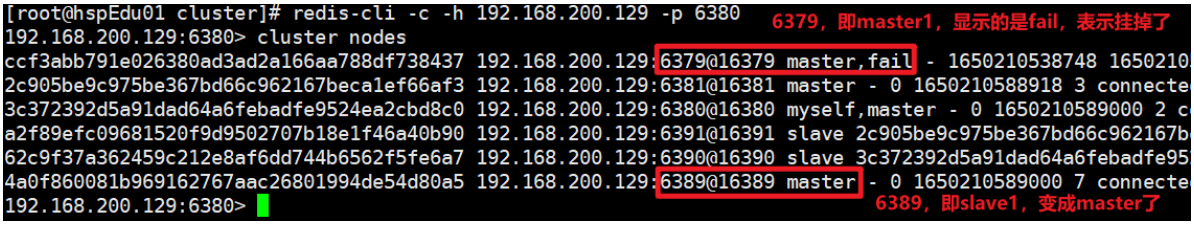
下面我们来试试，如下，连接master1，然后将master1停掉

```
[root@hspEdu01 cluster]# redis-cli -c -h 192.168.200.129 -p 6379
192.168.200.129:6379> shutdown
not connected>
```

执行下面命令，连接master1，看下集群节点的信息

```
redis-cli -c -h 192.168.200.129 -p 6380
cluster nodes
```

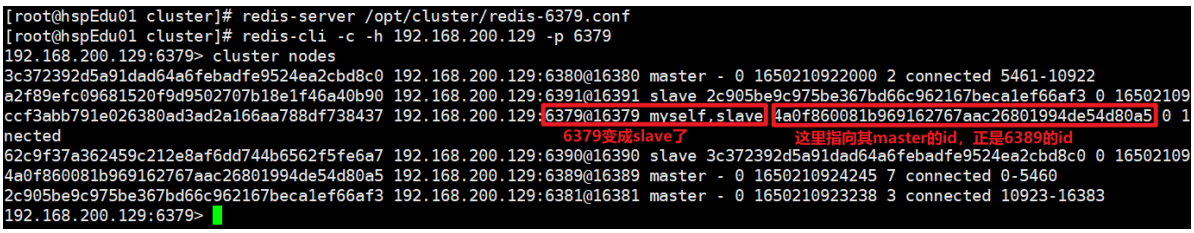
输出如下，可以看到slave1（6389）确实变成master了，而它原来的master：master1（6379）下线了



下面我们再来启动6379，然后再看看集群变成什么样了，命令如下

```
[root@hspEdu01 cluster]# redis-server /opt/cluster/redis-6379.conf
[root@hspEdu01 cluster]# redis-cli -c -h 192.168.200.129 -p 6379
192.168.200.129:6379> cluster nodes
```

执行结果如下，6379变成slave了，挂在了6389下面了



如果某一段插槽的主从都宕机了，redis服务是否还能继续?

这个时候要看`cluster-require-full-coverage`参数的值了

- **yes(默认值)**：整个集群都都无法提供服务了
- **no**：宕机的这部分槽位数据全部不能使用，其他槽位正常

11.0、SpringBoot整合redis集群

1）引入redis的maven配置

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2）application.properties中配置redis cluster信息

```
# 集群节点(host:port)，多个之间用逗号隔开
spring.redis.cluster.nodes=192.168.200.129:6379,192.168.200.129:6380,192.168.200
.129:6381,192.168.200.129:6389,192.168.200.129:6390,192.168.200.129:6391
# 连接超时时间（毫秒）
spring.redis.timeout=60000
```

3）使用RedisTemplate工具类操作redis

springboot中使用RedisTemplate来操作redis，需要在我们的bean中注入这个对象，代码如下：

```
@Autowired
private RedisTemplate<String, String> redisTemplate;

// 用下面5个对象来操作对应的类型
this.redisTemplate.opsForValue(); //提供了操作string类型的所有方法
this.redisTemplate.opsForList(); // 提供了操作list类型的所有方法
this.redisTemplate.opsForSet(); //提供了操作set的所有方法
this.redisTemplate.opsForHash(); //提供了操作hash表的所有方法
this.redisTemplate.opsForZSet(); //提供了操作zset的所有方法
```

2）RedisTemplate示例代码

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisCallback;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

@RestController
@RequestMapping("/redis")
public class RedisController {
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**
     * string测试
     *
     * @return
     */
    @RequestMapping("/stringTest")
    public String stringTest() {
        this.redisTemplate.delete("name");
        this.redisTemplate.opsForValue().set("name", "路人");
        String name = this.redisTemplate.opsForValue().get("name");
        return name;
    }

    /**
     * list测试
     *
```

```
     * @return
     */
    @RequestMapping("/listTest")
    public List<String> listTest() {
        this.redisTemplate.delete("names");
        this.redisTemplate.opsForList().rightPushAll("names", "刘德华", "张学友",
"郭富城", "黎明");
        List<String> courses = this.redisTemplate.opsForList().range("names", 0,
-1);
        return courses;
    }

    /**
     * set类型测试
     *
     * @return
     */
    @RequestMapping("setTest")
    public Set<String> setTest() {
        this.redisTemplate.delete("courses");
        this.redisTemplate.opsForSet().add("courses", "java", "spring",
"springboot");
        Set<String> courses = this.redisTemplate.opsForSet().members("courses");
        return courses;
    }

    /**
     * hash表测试
     *
     * @return
     */
    @RequestMapping("hashTest")
    public Map<Object, Object> hashTest() {
        this.redisTemplate.delete("userMap");
        Map<String, String> map = new HashMap<>();
        map.put("name", "路人");
        map.put("age", "30");
        this.redisTemplate.opsForHash().putAll("userMap", map);

        Map<Object, Object> userMap =
this.redisTemplate.opsForHash().entries("userMap");
        return userMap;
    }

    /**
     * zset测试
     *
     * @return
     */
    @RequestMapping("zsetTest")
    public Set<String> zsetTest() {
        this.redisTemplate.delete("languages");

        this.redisTemplate.opsForZSet().add("languages", "java", 100d);
        this.redisTemplate.opsForZSet().add("languages", "c", 95d);
        this.redisTemplate.opsForZSet().add("languages", "php", 70);
```

```
        Set<String> languages =
this.redisTemplate.opsForZSet().range("languages", 0, -1);
        return languages;
    }

    /**
     * 查看redis机器信息
     *
     * @return
     */
    @RequestMapping(value = "/info", produces = MediaType.TEXT_PLAIN_VALUE)
    public String info() {
        return this.redisTemplate.execute((RedisCallback<String>) connection ->
String.valueOf(connection.execute("info")));
    }
}
```

12、redis应用问题解决

12.1、缓存穿透

12.1.1、问题描述

当系统中引入redis缓存后，一个请求进来后，会先从redis缓存中查询，缓存有就直接返回，缓存中没有就去db中查询，db中如果有就会将其丢到缓存中，但是有些key对应更多数据在db中并不存在，每次针对此次key的请求从缓存中取不到，请求都会压到db，从而可能压垮db。

比如用一个不存在的用户id获取用户信息，不论缓存还是数据库都没有，若黑客利用大量此类攻击可能压垮数据库。

12.1.2、解决方案

（1）对空值缓存

如果一个查询返回的数据为空（不管数据库是否存在），我们仍然把这个结果（null）进行缓存，给其设置一个很短的过期时间，最长不超过五分钟

（2）设置可访问的名单（白名单）

使用redis中的bitmaps类型定义一个可以访问的名单，名单id作为bitmaps的偏移量，每次范文和bitmap里面的id进行比较，如果访问的id不在bitmaps里面，则进行拦截，不允许访问

（3）采用布隆过滤器

布隆过滤器（Bloom Filter）是1970年有布隆提出的，它实际上是一个很长的二进制向量（位图）和一系列随机映射函数（哈希函数）。

布隆过滤器可以用于检测一个元素是否在一个集合中，它的优点是空间效率和查询的世界都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

将所有可能存在的数据哈希到一个足够大的bitmaps中，一个一定不存在的数据会被这个bitmaps拦截掉，从而避免了对底层存储系统的查询压力。

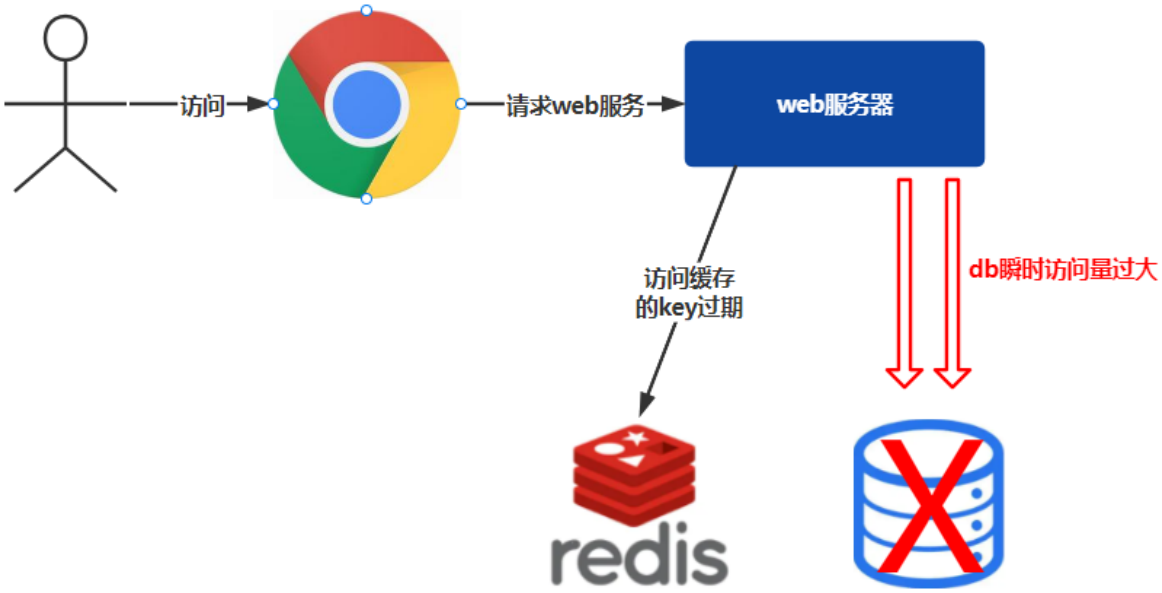
（4）进行实时监控

当发现redis的命中率开始急速降低，需要排查访问对象和访问的数据，和运维人员配合，可以设置黑名单限制对其提供服务（比如：IP黑名单）

12.2、缓存击穿

12.2.1、问题描述

redis中某个热点key（访问量很高的key）过期，此时大量请求同时过来，发现缓存中没有命中，这些请求都打到db上了，导致db压力瞬时大增，可能会打垮db，这种情况成为缓存击穿。



缓存击穿出现的现象

- 数据库访问压力瞬时增大
- redis里面没有出现大量的key过期
- redis正常运行

12.2.2、解决方案

key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据，这个时候，要考虑一个问题：缓存被“击穿”的问题，常见的解决方案如下

（1）预先设置热门数据，适时调整过期时间

在redis高峰之前，把一些热门数据提前存入到redis里面，对缓存中的这些热门数据进行监控，实时调整过期时间。

（2）使用锁

缓存中拿不到数据的时候，此时不是立即去db中查询，而是去获取分布式锁（比如redis中的setnx），拿到锁再去db中load数据；没有拿到锁的线程休眠一段时间再重试整个获取数据的方法。

12.3、缓存雪崩

12.3.1、问题描述

key对应的数据存在，但是极短时间内有大量的key集中过期，此时若有大量的并发请求过来，发现缓存没有数据，大量的请求就会落到db上去加载数据，会将db击垮，导致服务奔溃。

缓存雪崩与缓存击穿的区别在于：前者是大量的key集中过期，而后者是某个热点key过期。

12.3.2、解决方案

缓存失效时的雪崩效益对底层系统的冲击非常可怕，常见的解决方案如下

（1）构建多级缓存

nginx缓存+redis缓存+其他缓存（ehcache等）

（2）使用锁或队列

用加锁或者队列的方式来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上，不适用高并发情况。

（3）监控缓存过期，提前更新

监控缓存，发下缓存快过期了，提前对缓存进行更新。

（4）将缓存失效时间分散开

比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样缓存的过期时间重复率就会降低，就很难引发集体失效的事件。

12.4、分布式锁

12.4.1、问题描述

随着业务发展的需要，原单体单机部署的系统被演化成分布式集群系统后，由于分布式系统多线程、多进程且分布在不同机器上，这将使原单机部署情况下的并发控制锁策略失效，单纯的Java API并不能提供分布式锁的能力，为了解决这个问题就需要一种跨JVM的互斥机制来控制共享资源的访问，这就是分布式锁要解决的问题。

12.4.2、分布式锁主流的实现方案

1. 基于数据库实现分布式锁
2. 基于缓存（redis等）
3. 基于zookeeper

每一种分布式锁解决方案都有各自的优缺点

1. 性能：redis最高
2. 可靠性：zookeeper最高

这里我们就基于redis实现分布式锁。

12.4.3、解决方案：使用redis实现分布式锁

需要使用下面这个命令来实现分布式锁

```
set key value NX PX 有效期(毫秒)
```

这条命令表示：当key不存在的时候，设置其值为value，且同时设置其有效期

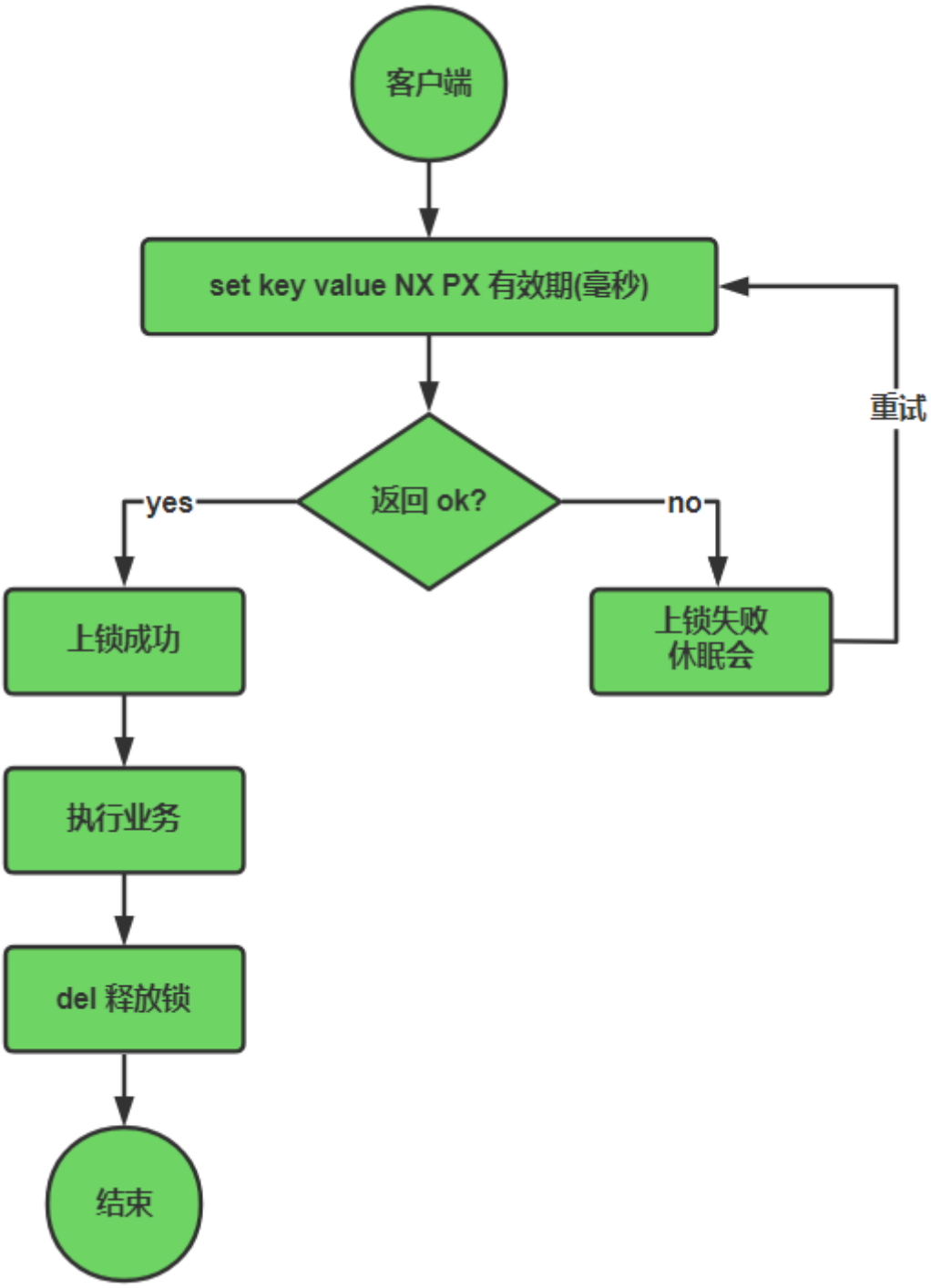
示例

```
set sku:1:info "ok" NX PX 10000
```

表示当`sku:1:info`不存在的时候，设置值为ok，且有效期为1万毫秒.

（1）上锁的过程

过程如下图，执行`set key value NX PX 有效期(毫秒)`命令，返回ok表示执行成功，则获取锁成功，多个客户端并发执行此命令的时候，redis可确保只有一个可以执行成功。



（2）为什么要设置过期时间?

客户端获取锁后，由于系统问题，如系统宕机了，会导致锁无法释放，其他客户端就无法或锁了，所以需要给锁指定一个使用期限。

（3）如果设置的有效期太短怎么办?

比如有效期设置了10秒，但是10秒不够业务方使用，这种情况客户端需要实现续命的功能，可以解决这个问题。

（4）解决锁误删的问题

锁存在误删的情况：所谓误删就是自己把别人持有的锁给删掉了。

比如线程A获取锁的时候，设置的有效期是10秒，但是执行业务的时候，A程序突然卡主了超过了10秒，此时这个锁就可能被其他线程拿到，比如被线程B拿到了，然后A从卡顿中恢复了，继续执行业务，业务执行完毕之后，去执行了释放锁的操作，此时A会执行del命令，此时就出现了锁的误删，导致的结果就是把B持有的锁给释放了，然后其他线程又会获取这个锁，挺严重的。

如何解决呢?

获取锁的之前，生成一个全局唯一id，将这个id也丢到key对应的value中，释放锁之前，从redis中将这个id拿出来和本地的比较一下，看看是不是自己的id，如果是的再执行del释放锁的操作。

（5）还是存在误删的可能（原子操作问题）

刚才上面说了，del之前，会先从redis中读取id，然后和本地id对比一下，如果一致，则执行删除，伪代码如下

```
step1:判断 redis.get("key").id==本地id 是否相当,如果是则执行step2
step2:del key;
```

此时如果执行step2的时候系统卡主了，比如卡主了10秒，然后redis才收到，这个期间锁可能又被其他线程获取了，此时又发生了误删的操作。

这个问题的根本原因是：判断和删除这2个步骤对redis来说不是原子操作导致的，怎么解决呢?

需要使用Lua脚本来解决。

（6）终极方案：Lua脚本来释放锁

将复杂的或者多步的redis操作，写为一个脚本，一次提交给redis执行，减少反复连接redis的次数，提升性能。

Lua脚本类似于redis事务，有一定的原子性，不会被其他命令插队，可以完成一些redis事务的操作。

但是注意redis的LUA脚本功能，只能在redis2.6以上版本才能使用。

代码如下:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Arrays;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

@RestController
public class LockTest {
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    @RequestMapping(value = "/lock", produces = MediaType.TEXT_PLAIN_VALUE)
    public String lock() {
        String lockKey = "k1";
        String uuid = UUID.randomUUID().toString();
        //1.获取锁,有效期10秒
```

```
        if (this.redisTemplate.opsForValue().setIfAbsent(lockKey, uuid, 10,
TimeUnit.SECONDS)) {

            //2.执行业务
            // todo 业务

            //3.使用Lua脚本释放锁(可防止误删)
            String script = "if redis.call('get',KEYS[1])==ARGV[1] then return
redis.call('del',KEYS[1]) else return 0 end";
            DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
            redisScript.setScriptText(script);
            redisScript.setResultType(Long.class);
            Long result = redisTemplate.execute(redisScript,
Arrays.asList(lockKey), uuid);
            System.out.println(result);
            return "获取锁成功!";
        } else {
            return "加锁失败!";
        }
    }
}
```

(7) 分布式锁总结

为了确保分布式锁可用，我们至少需要确保分布式锁的实现同时满足以下四个条件

- 互斥性，在任意时刻只能有一个客户端能够持有锁
- 不糊发生死锁，即使有一个客户端在持有锁期间崩溃而没有释放锁，也能够保证后续其他客户端能够加锁
- 解锁还需寄铃人，加锁和解锁必须是同一个客户端，客户端不能把别人的锁给解了
- 加锁和解锁必须有原子性
-