

# Redis

---

官方网站：<http://www.redis.io>

中文网站：<http://www.redis.cn>

开源的（BSD协议），使用ANSI C 编写，基于内存的且支持持久化，高性能的Key-Value的NoSQL数据库。支持数据结构类型丰富，有如 字符串（strings），散列（hashes），列表（lists），集合（sets），有序集合（sorted sets）与范围查询，bitmaps，hyperloglogs 和 地理空间（geospatial）索引半径查询。丰富的支持主流语言的客户端，C、C++、Python、Erlang、R、C#、Java、PHP、Objective-C、Perl、Ruby、Scala、Go、JavaScript。

用途：缓存（StackOverFlow）、数据库（微博）、消息中间件（微博）

Redis版本

目前主要版本为3.2

redis在2017年也发布了4.x版本，目前是4.0.x

部署环境是Linux，本次部署在CentOS 6.x上。

Windows版本由微软提供 <https://github.com/MicrosoftArchive/redis>

可视化工具RedisDesktopManager。windows目前使用redis-desktop-manager-0.8.x，0.9有点问题。

---

## 安装

### Linux单节点安装

```
# yum -y install gcc tcl

# tar xf redis-3.2.12.tar.gz
# mv redis-3.2.12/ redis
# cd redis
# make
```

如果出错，需要redis目录下

```
# cd deps
# make jemalloc
# make lua
# make linenoise
# make hiredis
# cd ..
```

```
# mkdir -p /magedu/redis
```

安装

```
# make install
```

默认安装到/usr/local/bin

或者

```
# make PREFIX=/magedu/redis install
```

可执行文件

```
redis-benchmark
redis-check-aof
redis-check-dump
redis-cli
redis-server
```

环境变量

可以将下面的变量追加加入到~/.bash\_profile文件末尾

```
export REDIS_HOME=/magedu/redis
export PATH=$PATH:$REDIS_HOME/bin
```

redis服务

可以查看redis-server启动的各种命令参考帮助

```
# redis-server --help
```

redis-3.2.12.tar.gz中的utils目录

```
# ./install_server.sh
```

完成后，配置文件在/etc/redis/6379.conf

可以把redis做成服务

```
# mv /etc/init.d/redis_6379 /etc/init.d/redisd
# chkconfig redisd on
```

查看后修改配置文件后，可以启动redis服务

```
# service redisd start
# service redisd start|stop|restart|status
# ss -tanl
```

使用install\_server.sh脚本生成的redis配置文件为/etc/redis/6379.conf

```
# 3.x增加了bind，不设置在本地所有网络接口监听
#bind 127.0.0.1
bind 192.168.142.135 127.0.0.1

# 3.x增加了保护模式，默认开启
# 开启保护模式时，如果不使用bind绑定IP地址，或不使用密码，那么只能本地地址访问或使用Unix socket文件
protected-mode yes

# 后台服务
daemonize yes
port 6379

# 缺省DB是0，设置DB的数目
```

databases 16

```
# 2种持久化方式
# RDB: Redis DB 默认开启
# 下面是执行快照的条件
#   In the example below the behaviour will be to save:
#   after 900 sec (15 min) if at least 1 key changed
#   after 300 sec (5 min) if at least 10 keys changed
#   after 60 sec if at least 10000 keys changed
save 900 1
save 300 10
save 60 10000
dbfilename dump.rdb
```

```
# AOF 默认关闭
appendonly no
appendfilename "appendonly.aof"
appendfsync everysec
```

说明：

用于测试，或者内存小可以设置最大内存，但是生产环境一定尽量使用内存  
maxmemory <bytes>

## Redis Windows安装

项目地址 <https://github.com/MicrosoftArchive/redis>

下载地址 <https://github.com/MicrosoftArchive/redis/releases/download/win-3.2.100/Redis-x64-3.2.100.zip>

解压缩，就可以用了。

## Redis数据模型

redis支持数据模型非常丰富

### 键Key

Redis key 值是二进制安全的，这意味着可以用任何二进制序列作为key值，从形如“foo”的简单字符串到一个JPEG文件的内容都可以。空字符串也是有效key值

Key取值原则

键值不需要太长，消耗内存，而且查找这类键值的计算成本较高

键值不宜过短，可读性较差

习惯上key采用'user:123:password'形式，表示用户id为123的用户的密码

### 字符串

字符串是一种最基本简单的Redis值类型。Redis字符串是二进制安全的，这意味着一个Redis字符串能包含任意类型的数据，例如：一张JPEG格式的图片或者一个序列化的Ruby对象。

一个字符串类型的值最多能存储512M字节的内容。

## python中redis编程

安装redis库

```
$ pip install redis
```

```
import redis

db = redis.Redis() # 默认本地6379的0号库

db.set('testbin', 0b01100010) # 0x62
print(db.get('testbin'))

db.set(0b11, 0x63)
print(db.get(0b11))
print(db.get(3))
print(db.get('3'))

print(db.keys('*'))
```

注意：上例中0x62实际上发生了类型变化，因为返回的bytes类型98，实际上对应ASCII的98，已经是2字节了。

数值会先转换成10进制64位有符号数后，再转成字符串，存入redis中。

### 查看帮助

```
> Help 查看帮助
> Help <tab> 使用tab键切换帮助
> Help set 查看set命令帮助
> Help @string 查看命令组帮助
```

### 字符串设置

```
SET key value [EX seconds][PX milliseconds] [NX|XX]
```

设置字符串值

EX 设置过期时间，秒，等同于 `SETEX key seconds value`

PX 设置过期时间，毫秒，等同于 `PSETEX key milliseconds value`

NX 键不存在，才能设置，等同于 `SETNX key value`

XX 键存在时，才能设置

```
MSET key value [key value ...]
```

设置多个键的字符串值，key存在则覆盖，key不存在则增加

原子操作

```
MSETNX key value [key value ...]
```

key不存在则设置，key存在则失败。nx指不存在。

原子操作

```
set s1 abc
set s2 xyz
set s3 abcd ex 15

mset s3 3 s4 4 s5 5
msetnx s5 a5 s6 6
```

## 过期操作和生存时间

Redis中可以给每个Key设置一个生存时间（秒或毫秒），当达到这个时长后，这些键值将会被自动删除

```
EXPIRE key seconds
```

```
PEXPIRE key milliseconds
```

设置多少秒或者毫秒后过期

```
EXPIREAT key timestamp
```

```
PEXPIREAT key milliseconds-timestamp
```

设置在指定Unix时间戳过期

```
PERSIST key
```

持久key，即取消过期

适用场景

一、多少秒过期，例如一个缓存数据失效

二、PEXPIREAT key milliseconds-timestamp，比如现在开始缓存数据，到0点失效

Time To Live，Key的剩余生存时间

```
TTL key
```

```
PTTL key
```

key存在但没有设置TTL，返回-1

key存在，但还在生存期内，返回剩余的秒或者毫秒

key曾经存在，但已经消亡，返回-2（2.8版本之前返回-1）

```
set s5 abc ex 20
ttl s5

setnx s6 6
expire s6 60
pttl s6
persist s6
ttl s6

EXPIREAT cache 1355292000
PEXPIREAT mykey 1555555555005
```

## key操作

```
keys pattern
```

pattern可以取如下值：

1. \* 任意长度字符
2. ? 任意一个字符
3. [] 字符集合，表示一个字符

```
keys *
keys s?
keys s[13]
keys s*
keys ??
```

TYPE key key类型

EXISTS key key是否存在

RENAME key newkey、RENAMENX key newkey 键重命名

DEL key [key ...] 键删除

## 字符串获取

GET key 获取值

MGET key [key ...] 获取多个给定的键的值

GETSET key value 返回旧值并设置新值，如果键不存在，就创建并赋值

STRLEN key 字符串长度

```
get s4
mget s1 s3 s5 s7
strlen s3
mgetset s5 100
```

## 字符串操作

APPEND key value 追加字符串。如果键存在就追加；如果不存在就等同于SET key value

获取子字符串

GETRANGE key start end 索引值从0开始，支持负索引，-1表示最后一个字符。范围是[start, end]，start必须在end的左边，否则返回空串

SETRANGE key offset value 从指定索引处开始覆盖字符串，返回覆盖后字符串长度。key不存在会创建新的

```
append s2 abc
get s2
getrange s2 1 3
getrange s2 0 -1
setrange s2 3 12
setrange s2 3 12345
setrange s7 3 abc
get s7 # \x00\x00\x00abc
```

## 自增、自减

INCR key 和 DECR key 步长1的增减

INCRBY key decrement 和 DECRBY key decrement 步长增减

字符串值会被解释成64位有符号的十进制整数来操作，结果依然转成字符串

```
get s3 3
incrby s3 4
decr s3
decrby s3 2
```

## 库操作

```
redis-cli --help
```

```
redis-cli -n 2
```

登录不同的库

```
FLUSHDB
```

清除**当前库**数据

```
FLUSHALL
```

清除**所有库**中的数据

## 位图bitmap

位图不是真正的数据类型，它是定义在字符串类型上，只不过把字符串按位操作一个字符串类型的值最多能存储512M字节的内容，可以表示 $2^{32}$ 位  
位上限：

$512=2^9$

$1M=1024*1024=2^{10+10}$

$1Byte=8bit=2^3bit$

$2^{(9+10+10+3)} = 2^{32} b = 4294967296 b$ ，接近43亿个位

`SETBIT key offset value` 设置某一位上的值

offset偏移量，从0开始

value不写，默认是0

`GETBIT key offset` 获取某一位上的值

`BITPOS key bit [start][end]` 返回指定值0或者1在指定区间上第一次出现的位置

`BITCOUNT key [start][end]` 统计指定位区间上值为1的个数

从左向右从0开始，从右向左从-1开始，注意官方start、end是位，测试后是字节

`BITCOUNT testkey 0 0` 表示从索引为0个字节到索引为0个字节，就是第一个字节的统计

`BITCOUNT testkey 0 -1` 等同于 `BITCOUNT testkey`

最常用的就是 `BITCOUNT testkey`

```
set s4 7 # 请问在redis中如何理解存入了什么？一下的各位返回多少
getbit s4 0
getbit s4 1
getbit s4 2
getbit s4 3
getbit s4 4
getbit s4 5
bitpos s4 1 # 1第一次出现的索引
bitcount s4 0 0 # 5
```

```
set str1 abc
setbit str1 6 1
setbit str1 7 0
get str1 # 请问等于什么？
```

## 位操作

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

当 BITOP 处理不同长度的字符串时，较短的那个字符串所缺少的部分会被看作 0

空的 key 也被看作是包含 0 的字符串序列

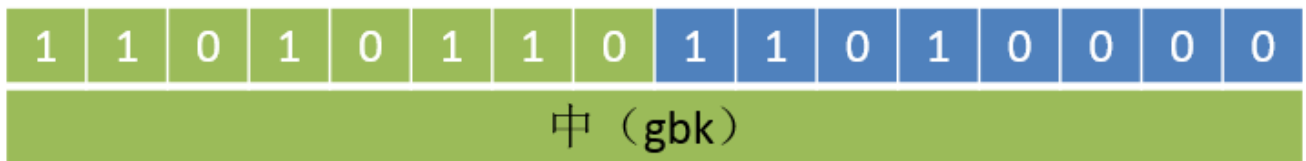
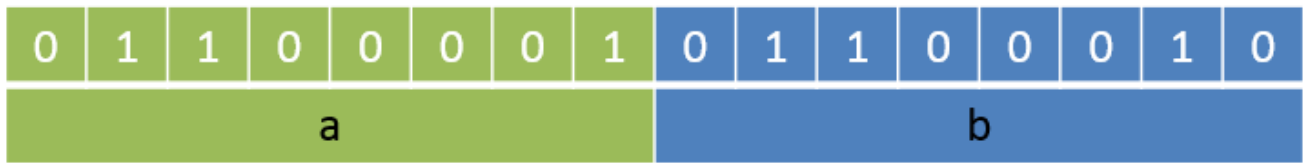
思考：'a'位或'b'是什么？

```
set s1 ab
bitcount s1
bitcount s1 0 0
bitcount s1 1 1

set s2 a
set s3 b
bitop or s8 s2 s3 # 等于什么？

set cn 中
get cn
bitcount cn
```





## 习题

- 1、网站用户的上线次数统计（活跃用户）
- 2、按天统计网站活跃用户

## 参考

- 1、网站用户的上线次数统计（活跃用户）

用户ID为key，天作为offset，上线置为1  
 ID为500的用户，今年的第1天上线、第30天上线  
 SETBIT u500 1 1  
 SETBIT u500 30 1  
 BITCOUNT u500  
 KYES u\*

```
import redis

r = redis.Redis(host='192.168.56.201', port=6379, db=2)

# u1
r.setbit('u1', 1, 1)
r.setbit('u1', 30, 1)
# u2
r.setbit('u2', 110, 1)
r.setbit('u2', 300, 1)

# u101
for i in range(3, 365, 3):
    r.setbit('u101', i, 1)
```

```
# u105
for i in range(4,365,2):
    r.setbit('u105', i, 1)

userList = r.keys('u*')
print(userList)

Au = []
Nau = []
for u in userList:
    loginCount = r.bitcount(u)
    if loginCount > 100:
        Au.append((u,loginCount))
    else:
        Nau.append((u,loginCount))

for l in Au:
    print l[0] + ' is a Active User.' + str(l[1])

print "~~~~~"

for l in Nau:
    print l[0] + ' is not a Active User.' + str(l[1])
```

## 2、按天统计网站活跃用户

天作为key，用户ID为offset，上线置为1

求一段时间内活跃用户数

SETBIT 20160602 15 1

SETBIT 20160601 123 1

SETBIT 20160606 123 1

求6月1日到6月10日的活跃用户

BITOP OR 20160601-10 20160601 20160602 20160603 20160610

BITCOUNT 20160601-10

结果为2

## List列表

- 其列表是基于双向链表实现，列表头尾增删快，中间增删慢
- 元素是字符串类型
- 元素可以重复出现
- 索引支持正索引和负索引，从左至右从0开始，从右至左从-1开始

命令说明

字母	说明
B	Block阻塞
L	Left 左起
R	Right 右起
X	exist 存在

`LLEN key` 返回列表元素个数

`LPUSH key value [value ...]` 从左边向队列中压入元素

`LPUSHX key value` 从左边向队列加入元素，要求key必须存在

`RPUSH key value [value ...]` 从右边向队列中压入数据

`RPUSHX key value` 要求key存在

`LPOP key` 从左边弹出列表中一个元素

`RPOP key` 从右边弹出列表中一个元素

`RPOPLPUSH source destination` 从源列表中右边pop一个元素，从左边加入到目标列表

`LRANGE key start stop` 返回列表中指定访问的元素，例如LRANGE user 0 -1

`LINDEX key index` 返回列表中指定索引的元素

`LSET key index value` 设置列表中指定索引位置的元素值，index不能超界

`LREM key count value` 从左边删除列表中与value相等的元素

count > 0 从左至右搜索，移除与 value 相等的元素，数量至多为 count 次

count < 0 从右至左搜索，移除与 value 相等的元素，数量至多为 -count次

count = 0 移除列表中所有value值

`LTRIM key start stop` 去除指定范围外的元素

`RPUSH listkey c abc c ab 123 ab bj ab redis list`

`LTRIM listkey 0 -1 # 什么都没有去除`

`LTRIM listkey 1 -1 # 去掉左边头`

`LTRIM listkey 1 10000`

`LINSERT key BEFORE|AFTER pivot value` 在列表中某个存在的值 ( pivot ) 前或后插入元素一次，key或pivot不存在，不进行任何操作

`RPUSH list 1 2 3 4 2 8`

`LINSERT list AFTER 2 Python`

`LINSERT list BEFORE 2 Ruby`

## 阻塞

如果弹出的列表不存在或者为空，就会阻塞

超时时间设置为0，就是永久阻塞，直到有数据可以弹出

如果多个客户端阻塞在同一个列表上，使用First In First Service原则，先到先服务

`BLPOP key [key ...] timeout` 列表左边阻塞弹出元素。timeout是超时秒数，为0为永久阻塞。

`BRPOP key [key ...] timeout` 列表右边阻塞弹出元素

`BRPOPLPUSH source destination timeout` 从一个列表尾部阻塞弹出元素压入到另一个列表的头部

```
# 阻塞式消息队列
BLPOP MyQueue 0
RPUSH MyQueue hello
```

## 习题

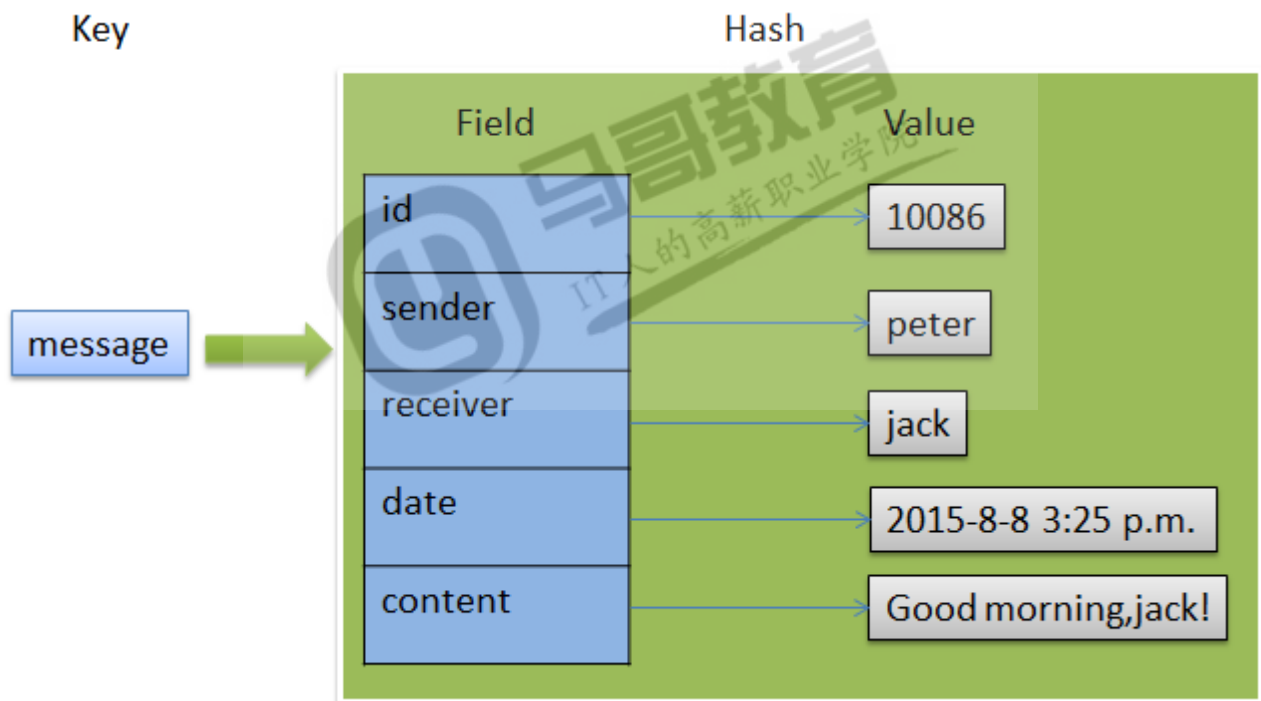
微博某贴最后评论的50条

```
LPUSH u1234:forumid:comments "这是第1条评论"
LPUSH u1234:forumid:comments "这是第2条评论"
LPUSH u1234:forumid:comments "这是第3条评论"
LTRIM u1234:forumid:comments 0 499
```

## hash散列

值是由field和value组成的map键值对

field和value都是字符串类型



`HSET key field value` 设置单个字段。field不存在创建，存在覆盖value

`HSETNX key field value` 设置单个字段，要求field不存在。如果key不存在，相当于field也不存在

`HMSET key field value [field value ...]` 设置多个字段

`HLEN key` 返回字段个数

`HEXISTS key field` 判断字段是否存在。key或者field不存在，返回0

`HGET key field` 返回字段值

`HMGET key field [field ...]` 返回多个字段值

`HGETALL key` 返回所有的键值对

`HKEYS key` 返回所有字段名

HVALS key 返回所有值

HINCRBY key field increment 在字段对应的值上进行整数的增量计算

HINCRBYFLOAT key field increment 在字段对应的值上进行浮点数的增量计算

HDEL key field [field ...] 删除指定的字段

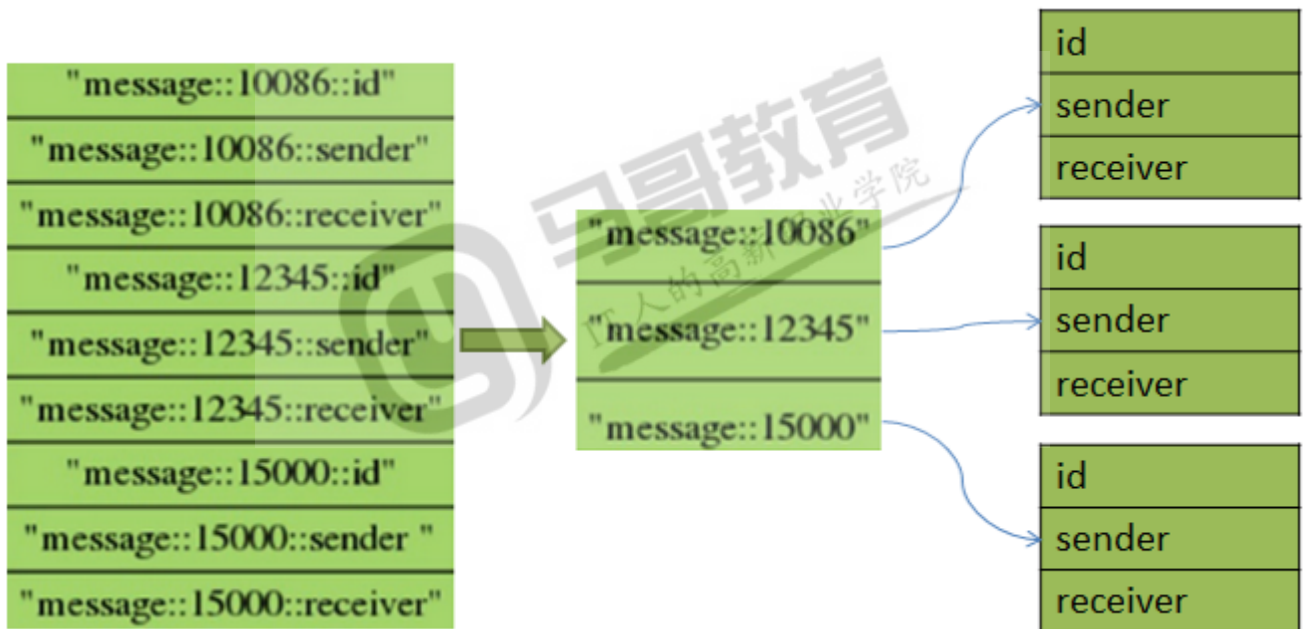
```
HINCRBY numbers x 100
HINCRBY numbers x -50
HINCRBYFLOAT numbers x 3.14
HDEL numbers x
```

## hash用途

节约内存空间

每创建一个键，它都会为这个键储存一些附加的管理信息（比如这个键的类型，这个键最后一次被访问的时间等等）

所以数据库里面的键越多，redis数据库服务器在储存附加管理信息方面耗费的内存就越多，花在管理数据库键上的CPU时间也会越多



## 不适合hash的情况

使用二进制位操作命令：因为Redis目前支持对字符串键进行SETBIT、GETBIT、BITOP等操作，如果你想使用这些操作，那么只能使用字符串键，虽然散列也能保存二进制数据

使用过期键功能：Redis的键过期功能目前只能对键进行过期操作，而不能对散列的字段进行过期操作，因此如果你要对键值对数据使用过期功能的话，那么只能把键值对储存在字符串里面

## 习题

用户维度统计

统计数包括：关注数、粉丝数、喜欢商品数、发帖数

用户为Key，不同维度为Field，Value为统计数

比如关注了5人

HSET user:100000 follow 5  
HINCRBY user:100000 follow 1

商品维度统计

统计值包括喜欢数，评论数，购买数，浏览数等

HSET item:58000 fav 500  
HINCRBY item:58000 fav 1

缓存用户信息

登录后，反复需要读取用户的常用信息，最好的方式就是缓存起来

```
set user:001 "bob,18,20010101"
mset user:001:name "bob" user:001:age 18 user:001:birthday "20010101"
hmset user:001 name "bob" age 18 birthday "20010101"
```

## Set集合

集合的元素是无序的、去重的，元素是字符串类型。

`SADD key member [member ...]` 增加一个或多个元素，元素已存在将忽略

`SREM key member [member ...]` 移除一个或多个元素，元素不存在自动忽略

`SCARD key` 返回集合中元素的个数。不需要遍历。

`SMEMBERS key` 返回集合中的所有元素。注意，如果集合中元素过多，应当避免使用该方法

`SISMEMBER key member` 元素是否是在集合中

```
SADD f1 "peter" "jack" "tom" "john" "may" "ben"
SADD f2 "peter" "jack"
SADD f2 "tom" "john"
SADD f2 "may" "ben"
SMEMBERS f1
SMEMBERS f2
```

元素相同的两个集合，未必有相同的顺序。去重且有序可使用有序集合

`SRANDMEMBER key [count]` 随机返回集合中指定个数的元素如果 count 为正数，且小于集合基数，那么命令返回一个包含 count 个元素的数组，数组中的元素各不相同。如果 count 大于等于集合基数，那么返回整个集合  
如果 count 为负数，那么命令返回一个数组，数组中的元素可能会重复出现多次，而数组的长度为 count 的绝对值

如果 count 为 0，返回空

如果 count 不指定，随机返回一个元素

`SPOP key` 从集合中随机移除一个元素并返回该元素

`SMOVE source destination member` 把元素从源集合移动到目标集合

集合运算

差集

`SDIFF key [key ...]` 从第一个key的集合中去除其他集合和自己的交集部分

`SDIFFSTORE destination key [key ...]` 将差集结果存储在目标key中

```
SADD number1 123 456 789
SADD number2 123 456 999
SDIFF number1 number2
```

### 交集

`SINTER key [key ...]` 取所有集合交集部分

`SINTERSTORE destination key [key ...]` 将交集结果存储在目标key中

```
SADD number1 123 456 789
SADD number2 123 456 999
SINTER number1 number2
```

### 并集

`SUNION key [key ...]` 取所有集合并集

`SUNIONSTORE destination key [key ...]` 将并集结果存储在目标key中

```
SADD number1 123 456 789
SADD number2 123 456 999
SUNION number1 number2
```

### 习题

#### 微博的共同关注

需求：当用户访问另一个用户的时候，会显示出两个用户共同关注哪些相同的用户

设计：将每个用户关注的用户放在集合中，求交集即可

```
peter={'john','jack','may'}
ben={'john','jack','tom'}
那么peter和ben的共同关注为：
SINTER peter ben 结果为 {'john','jack'}
```

## SortedSet有序集合

类似Set集合，有序的集合。

每一个元素都关联着一个浮点数值（Score），并按照分值从小到大的顺序排列集合中的元素。分值可以相同

## 一个保存了水果价格的有序集合

分值	2.0	3.2	4.0	6.8	7.0	9.2	12.0
元素	西瓜	香蕉	番石榴	芒果	梨	葡萄	苹果

## 一个保存了员工薪水的有序集合

分值	3500.0	4000.0	4000.0	4500.0	25000.0
元素	jack	john	peter	tom	david

## 一个有序集合，保存了正在阅读某些技术书的人数

分值	251	347	928	1030	3436
元素	编程人生	人月神话	设计模式	深入了解计算机系统	算法导论

`ZADD key score member [score member ...]` 增加一个或多个元素。如果元素已经存在，则使用新的score

`ZCARD key` 返回集合的元素个数

`ZCOUNT key min max` 返回指定score范围元素的个数

`ZSCORE key member` 显示分值

`ZINCRBY key increment member` 增加或减少分值。increment为负数就是减少

`ZRANGE key start stop [WITHSCORES]` 返回指定索引区间元素

如果score相同，则按照字典序lexicographical order 排列

默认按照score从小到大，如果需要score从大到小排列，使用ZREVRANGE

`ZREVRANGE key start stop [WITHSCORES]` 返回指定索引区间元素

如果score相同，则按照字典序lexicographical order 的 **逆序** 排列

默认按照score从大到小，如果需要score从小到大排列，使用ZRANGE

`ZRANK key member` 返回元素的排名（索引）

`ZREVRANK key member` 返回元素的逆序排名（索引）



```
ZADD employees 3500 jack 4000 peter 4000 john 4500 tom 2500 david
ZCOUNT employees 3000 4000
ZADD employees 3.2 david
ZSCORE employees david

ZINCRBY employees 1.5 jack
ZINCRBY employees -500 tom

ZRANGE employees 0 -1 WITHSCORES
ZRANK employees peter
ZREVRANGE employees 0 -1 WITHSCORES # 逆序后的索引0到-1，即返回所有
ZREVRANK employees peter
```

`ZRANGEBYSCORE key min max [WITHSCORES][LIMIT offset count]` 返回指定分数区间的元素  
返回score默认属于[min,max]之间，元素按照score升序排列，score相同字典序  
LIMIT中offset代表跳过多少个元素，count是返回几个。类似于Mysql  
使用小括号，修改区间为开区间，例如 (5 或者 (10、5)  
-inf和+inf表示负无穷和正无穷

`ZREVRANGEBYSCORE key max min [WITHSCORES][LIMIT offset count]` 降序返回指定分数区间的元素  
返回score默认属于[min,max]之间，元素按照score降序排列，score相同字典降序

```
ZRANGEBYSCORE employees 3500 4000
ZRANGEBYSCORE employees (4000 5000
ZRANGEBYSCORE employees 4000 5000 LIMIT 1 5 # 跳过一个，返回至多5个

ZREVRANGEBYSCORE employees +inf -inf
```

`ZREM key member [member ...]` 移除一个或多个元素。元素不存在，自动忽略

`ZREMRANGEBYRANK key start stop` 移除指定排名范围的元素

`ZREMRANGEBYSCORE key min max` 移除指定分值范围的元素

```
ZREMRANGEBYRANK employees 0 1
ZREMRANGEBYSCORE employees 4000 5000
```

## 集合运算

### 并集

`ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight] [AGGREGATE SUM|MIN|MAX]`

numkeys指定key的数量，必须

WEIGHTS选项，与前面设定的key对应，对应key中每一个score都要乘以这个权重

AGGREGATE选项，指定并集结果的聚合方式

SUM：将所有集合中某一个元素的score值之和作为结果集中该成员的score值，默认

MIN：将所有集合中某一个元素的score值中最小值作为结果集中该成员的score值

MAX：将所有集合中某一个元素的score值中最大值作为结果集中该成员的score值

```
ZADD scores1 70 tom 80 peter 60 john
ZADD scores2 90 peter 60 ben
ZUNIONSTORE scores-all 2 scores1 scores2
ZUNIONSTORE scores-all1 2 scores1 scores2 AGGREGATE SUM
ZUNIONSTORE scores-all2 2 scores1 scores2 WEIGHTS 1 0.5 AGGREGATE SUM
```

## 交集

```
ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight] [AGGREGATE SUM|MIN|MAX]
```

numkeys指定key的数量，必须

WEIGHTS选项，与前面设定的key对应，对应key中每一个score都要乘以这个权重

AGGREGATE选项，指定并集结果的聚合方式

SUM：将所有集合中某一个元素的score值之和作为结果集中该成员的score值

MIN：将所有集合中某一个元素的score值中最小值作为结果集中该成员的score值

MAX：将所有集合中某一个元素的score值中最大值作为结果集中该成员的score值

## 习题

音乐排行榜怎样实现

榜单 <span>更多 →</span>		
新 New 歌榜	热 Hot 歌榜	原 Original 歌榜
云音乐新歌榜	云音乐热歌榜	网易原创歌曲榜
1 平凡之路	1 平凡之路	1 一个短篇
2 不再见	2 后会无期	2 白日
3 时间煮雨	3 小苹果	3 半支烟
4 山水之间	4 不再见	4 妈妈一起飞吧，妈妈一起摇滚吧
5 惊鸿一面	5 夜空中最亮的星	5 夏天的情歌
6 爱的勇气	6 女儿情	6 一个人看小丸子
7 无痕	7 泡沫	7 梦中的额吉
8 你是对的人	8 惊鸿一面	8 当你来临的时候 - Reproduction
9 一起老去	9 时间煮雨	9 知了
10 1987我不知会遇见你	10 剑心	10 出发
<a href="#">查看全部&gt;</a>	<a href="#">查看全部&gt;</a>	<a href="#">查看全部&gt;</a>

每首歌的歌名作为元素（先不考虑重复）

每首歌的播放次数作为分值

ZREVRANGE来获取播放次数最多的歌曲（就是最多播放榜了，云音乐热歌榜，没有竞价，没有权重）

```
import redis

r = redis.Redis(host='192.168.142.135', port=6379, db=3)

#
```

```

r.zadd('mboard','yellow',1,'rolling in the deep',1,'happy',1,'just the way you are',1)
r.zadd('mboard','eye of the tiger',1,'billie jean',1,'say you say me',1,'payphone',1)
r.zadd('mboard','my heart will go on',1,'when you believe',1,'hero',1)

r.zincrby('mboard','yellow',50)
r.zincrby('mboard','rolling in the deep',60)
r.zincrby('mboard','my heart will go on',68.8)
r.zincrby('mboard','when you believe',70)

# 所有元素
allmusic = r.zrange('mboard', 0, -1, withscores=True)
print(type(allmusic))
for m in allmusic:
    print(m)

print('-'*30)
# 排行榜
musicboard = r.zrevrange('mboard', 0, 9, True)

print('欧美热曲榜')
for i, m in enumerate(musicboard):
    print(i, *m)

```

## 新浪微博翻页

新闻网站、博客、论坛、搜索引擎，页面列表条目多，都需要分页

blog这个key中使用时间戳作为score

ZADD blog 1407000000 '今天天气不错'

ZADD blog 1450000000 '今天我们学习Redis'

ZADD blog 1560000000 '几个Redis使用示例'

ZREVRANGE blog 10 20

显示所有博客中的最后的指定的条目

## 京东图书畅销榜

统计单日榜，计算出周榜单、月榜单、年榜单

怎么做？

每天统计一次排行榜

ZADD bk:it:01 1000 'java' 1500 'Redis' 2000 'hadoop' 100 'scala' 80 'python'

ZADD bk:it:02 1020 'java' 1500 'Redis' 2100 'hadoop' 120 'python' 110 'scala'

ZADD bk:it:03 1620 'java' 1510 'Redis' 3000 'hadoop' 150 'storm' 120 'python'

求销售前10名

ZUNIONSTORE bk:it:01-03 3 bk:it:01 bk:it:02 bk:it:03

行吗？

因为上面的单日榜单是累计值，所以不能直接使用并集，要指定聚合运算为MAX

```

ZUNIONSTORE bk:it:01-03 3 bk:it:01 bk:it:02 bk:it:03 AGGREGATE MAX
ZREVRANGE bk:it:01-03 0 9 WITHSCORES

```

注意：如果参与并集元素的元素太多，会耗费大量内存和计算时间，可能会导致Redis服务阻塞，如果非要计算，选在空闲时间或备用服务器上计算。

另一种统计

```
ZADD bk:it:01 50 'java' 20 'Redis' 40 'hadoop'
ZADD bk:it:02 70 'java' 30 'Redis' 20 'hadoop'
ZADD bk:it:03 20 'java' 30 'Redis' 5 'hadoop'
每天统计当天销售量，统计IT类图书一段时间的最新销售榜单
ZUNIONSTORE bk:it:01-03 3 bk:it:01 bk:it:02 bk:it:03 AGGREGATE SUM
ZREVRANGE bk:it:01-03 0 9 WITHSCORES
```

## Redis持久化

持久化：将数据从掉电易失的内存存储到能够永久存储的设备上

Redis服务是使用内存来存储数据，如果掉电、服务崩溃都会导致Redis中数据丢失，如有必要，可以持久化数据。

Redis持久化方式：RDB ( Redis DB )、AOF ( AppendOnlyFile )

### RDB

在默认情况下，Redis 将某时间点的数据库快照保存在名字为 dump.rdb的二进制文件中

策略

- 自动：按照配置文件中的条件满足就执行BGSAVE
- 手动：客户端发起SAVE、BGSAVE命令

### 配置

```
save 900 1
save 300 10
save 60 10000
dbfilename dump.rdb
dir /var/lib/redis/6379
```

save 60 1000，Redis要满足在60秒内至少有1000个键被改动，会自动保存一次

只要满足上面3个条件之一，就自动执行快照

执行完后，时间计数器和次数计数器都会归零重新计数。这多个条件不是叠加效果

SAVE命令：阻塞式命令，执行期间不响应客户端的请求

BGSAVE：非阻塞命令，执行期间还可以接收并处理客户端请求，会fork一个子进程创建RDB文件

```
192.168.142.135:6379> SAVE
192.168.142.135:6379> BGSAVE
```

优点

- 完全备份，不同时间的数据集备份可以做到多版本恢复

- 紧凑的单一文件，方便网络传输，适合灾难恢复
- 快照文件直接恢复，大数据集速度较AOF快

#### 缺点

- 会丢失最近写入、修改的而未能持久化的数据
- fork过程非常耗时，会造成毫秒级不能响应客户端请求

#### RDB备份策略

创建一个定时任务cron job，每小时或者每天将dump.rdb复制到指定目录  
确保备份文件名称带有日期时间信息，便于管理和还原对应的时间点的快照版本  
定时任务删除过期的备份  
如果有必要，跨物理主机、跨机架、异地备份

## AOF

Append only file，采用追加的方式保存，默认文件appendonly.aof。  
记录所有的**写操作命令**，在服务启动的时候使用这些命令就可以还原数据库

### AOF写入机制

#### AOF方式不能保证绝对不丢失数据

目前常见的操作系统中，执行系统调用write函数，将一些内容写入到某个文件里面时，为了提高效率，系统通常不会直接将内容写入硬盘里面，而是先将内容放入一个内存缓冲区（buffer）里面，等到缓冲区被填满，或者用户执行fsync调用和fdatsync调用时才将储存在缓冲区里的内容真正的写入到硬盘里，未写入磁盘之前，数据可能会丢失

#### 写入磁盘的策略

appendfsync选项，这个选项的值可以是always、everysec或者no

- Always：服务器每写入一个命令，就调用一次fdatsync，将缓冲区里面的命令写入到硬盘。这种模式下，服务器出现故障，也不会丢失任何已经成功执行的命令数据
- Everysec（默认）：服务器每一秒重调用一次fdatsync，将缓冲区里面的命令写入到硬盘。这种模式下，服务器出现故障，最多只丢失一秒钟内的执行的命令数据
- No：服务器不主动调用fdatsync，由操作系统决定何时将缓冲区里面的命令写入到硬盘。这种模式下，服务器遭遇意外停机时，丢失命令的数量是不确定的

运行速度：always的速度慢，everysec和no都很快

### AOF重写机制

写操作越来越多的被记录，AOF文件会很大。Redis会合并写操作，以压缩AOF文件。  
合并重复的写操作，AOF会使用尽可能少的命令来记录。

#### 重写过程

1. fork一个子进程负责重写AOF文件
2. 子进程会创建一个临时文件写入AOF信息
3. 父进程会开辟一个内存缓冲区接收新的写命令
4. 子进程重写完成后，父进程会获得一个信号，将父进程接收到的新的写操作由子进程写入到临时文件中

## 5. 新文件替代旧文件

注：如果写入操作的时候出现故障导致命令写半截，可以使用redis-check-aof工具修复

原有AOF文件	重写后的AOF文件
SELECT 0	SELECT 0
SADD fruits "apple"	SADD fruits "apple" "banana" "cherry"
SADD fruits "banana"	SET msg "hello world again!"
SADD fruits "cherry"	RPUSH lst 3 5 7
SADD fruits "apple"	
INCR counter	
INCR counter	
DEL counter	
SET msg "hello world"	
SET msg "hello world again!"	
RPUSH lst 1 3 5	
RPUSH lst 7	
LPOP lst	

### AOF重写触发

手动：客户端向服务器发送BGREWRITEAOF命令

自动：配置文件中的选项，自动执行BGREWRITEAOF命令

`auto-aof-rewrite-min-size <size>`，触发AOF重写所需的最小体积：只要在AOF文件的体积大于等于size时，才会考虑是否需要AOF重写，这个选项用于避免对体积过小的AOF文件进行重写

`auto-aof-rewrite-percentage <percent>`，指定触发重写所需的AOF文件体积百分比：当AOF文件的体积大于`auto-aof-rewrite-min-size`指定的体积，并且超过上一次重写之后的AOF文件体积的percent %时，就会触发AOF重写。（如果服务器刚刚启动不久，还没有进行过AOF重写，那么使用服务器启动时载入的AOF文件的体积来作为基准值）。将这个值设置为0表示关闭自动AOF重写

### 重写配置举例

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
当AOF文件大于64MB时候，可以考虑重写AOF文件
只有当AOF文件的增量大于起始size的100%时（就是文件大小翻了一倍），启动重写

appendonly yes
默认关闭，请开启
```

## 优点

- 写入机制，默认fysnc每秒执行，性能很好不阻塞服务，最多丢失一秒的数据
- 重写机制，可以优化AOF文件体积
- 如果误操作了（FLUSHALL等），只要AOF未被重写，停止服务移除AOF文件尾部FLUSHALL命令，重启Redis，可以将数据集恢复到 FLUSHALL 执行之前的状态

## 缺点

- 相同数据集，AOF文件体积较RDB大了很多
- 恢复数据库速度叫RDB慢（文本，命令重演）

---

# Redis集群

---

Redis集群分为：

- 主从复制Replication
- 高可用 Sentinel
- 集群Cluster

## 主从复制

典型的主从模型，主Redis服务称为Master，从Redis服务称为Slave。

一主可以多从。

Master会一直将自己的数据更新同步到Slave，以保持主从同步。

只有Master可以执行读写操作，Slave只能执行读操作。客户端可以连接到任一Slave执行读操作，来降低Master的读取压力。

### 创建主从复制

#### 1. 命令创建

```
redis-server --slaveof <master-ip> <master-port>
```

配置当前服务称为某Redis服务的Slave

```
redis-server --port 6380 --slaveof 127.0.0.1 6379
```

#### 2. 指令创建

SLAVEOF host port命令，将当前服务器状态从Master修改为别的服务器的Slave

redis > SLAVEOF 192.168.1.1 6379，将服务器转换为Slave

redis > SLAVEOF NO ONE，将服务器重新恢复到Master，不会丢弃已同步数据

#### 3. 配置方式

启动时，服务器读取配置文件，并自动成为指定服务器的从服务器

```
slaveof <masterip> <masterport>
```

```
slaveof 127.0.0.1 6379
```

### 主从实验

Master 192.168.140.135 6379

Slave 192.168.140.140 6379

```
# redis-cli -h 192.168.142.140 -p 6379
slave> SET testkey abc
slave> KEYS *
slave> SLAVEOF 192.168.142.135 6379
slave> KEYS *    看看是否发生改变
slave> set slavekey 123
(error) READONLY You can't write against a read only slave.

master> SET masterkey 123
slave> GET masterkey

slave> SLAVEOF NO ONE    解除从
slave> KEYS *
slave> SET slavekey1 abc    可写了
slave> KEYS *
```

采用上面的多种方式都可以实现主从模式，一般来说，主从服务器都是固定的，采用配置文件方式。

### 主从复制问题

一个Master可以有多个Slaves。

如果Slave下线，只是读请求的处理能力下降。

但Master下线，写请求无法执行。

当Master下线，其中一台Slave使用SLAVEOF no one命令成为Master，其它Slaves执行SLAVEOF命令指向这个新的Master，从它这里同步数据。

这个主从转移的过程手动的，如果要实现自动故障转移，这就需要Sentinel哨兵，实现故障转移Failover操作

## 高可用Sentinel

Redis官方的高可用方案，可以用它管理多个Redis服务实例。

Redis Sentinel是一个分布式系统，可以在一个架构中运行多个Sentinel进程

使用编译时产生的redis-sentinel文件，在新的版本中，它就是redis-server的软链接。

Sentinel启动启动一个运行在Sentinel模式下的Redis服务实例

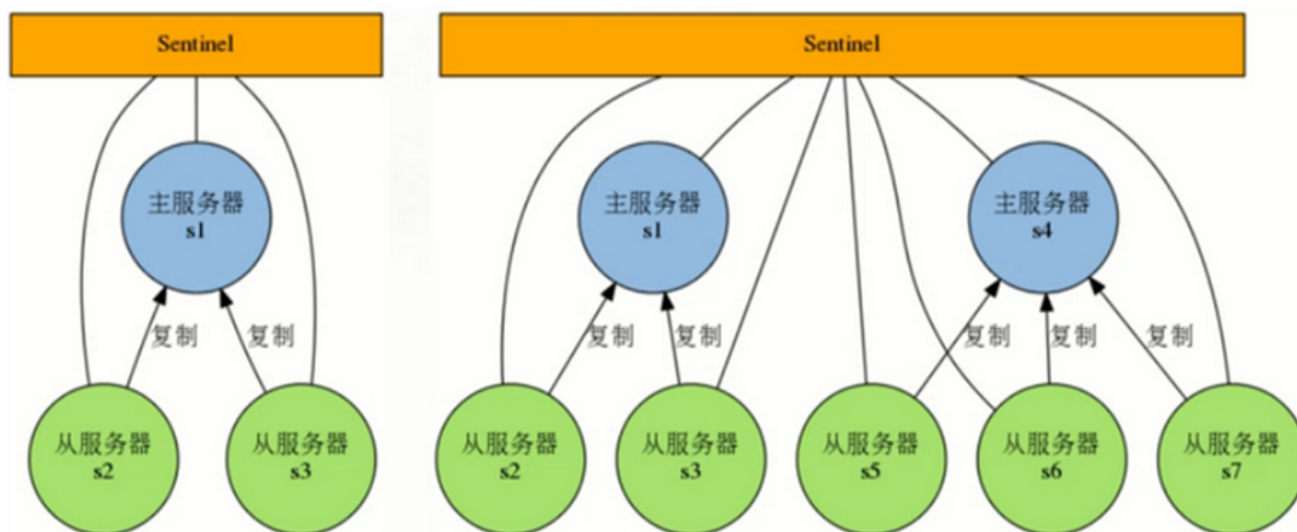
```
redis-sentinel /path/to/sentinel.conf
```

```
redis-server /path/to/sentinel.conf --sentinel
```

推荐使用第一种

### Sentinel原理





Sentinel会监控Master、Slave是否正常，可以监控多个Master、Slave。

Sentinel网络：监控同一个Master的Sentinel会自动连接，组成分布式的Sentinel网络，相互通信并交换监控信息。

### 服务器下线

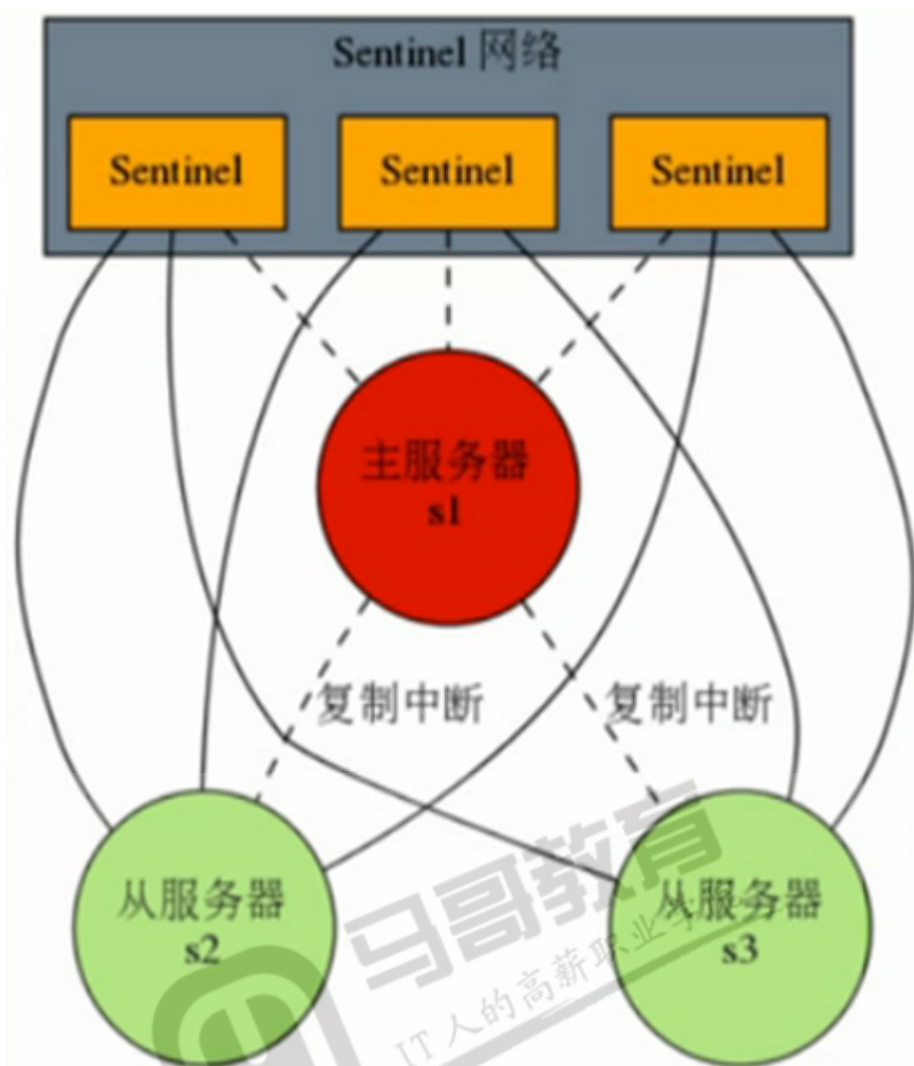
当一个sentinel认为被监视的服务器已经下线时，它会向网络中的其他Sentinel进行确认，判断该服务器是否真的已经下线

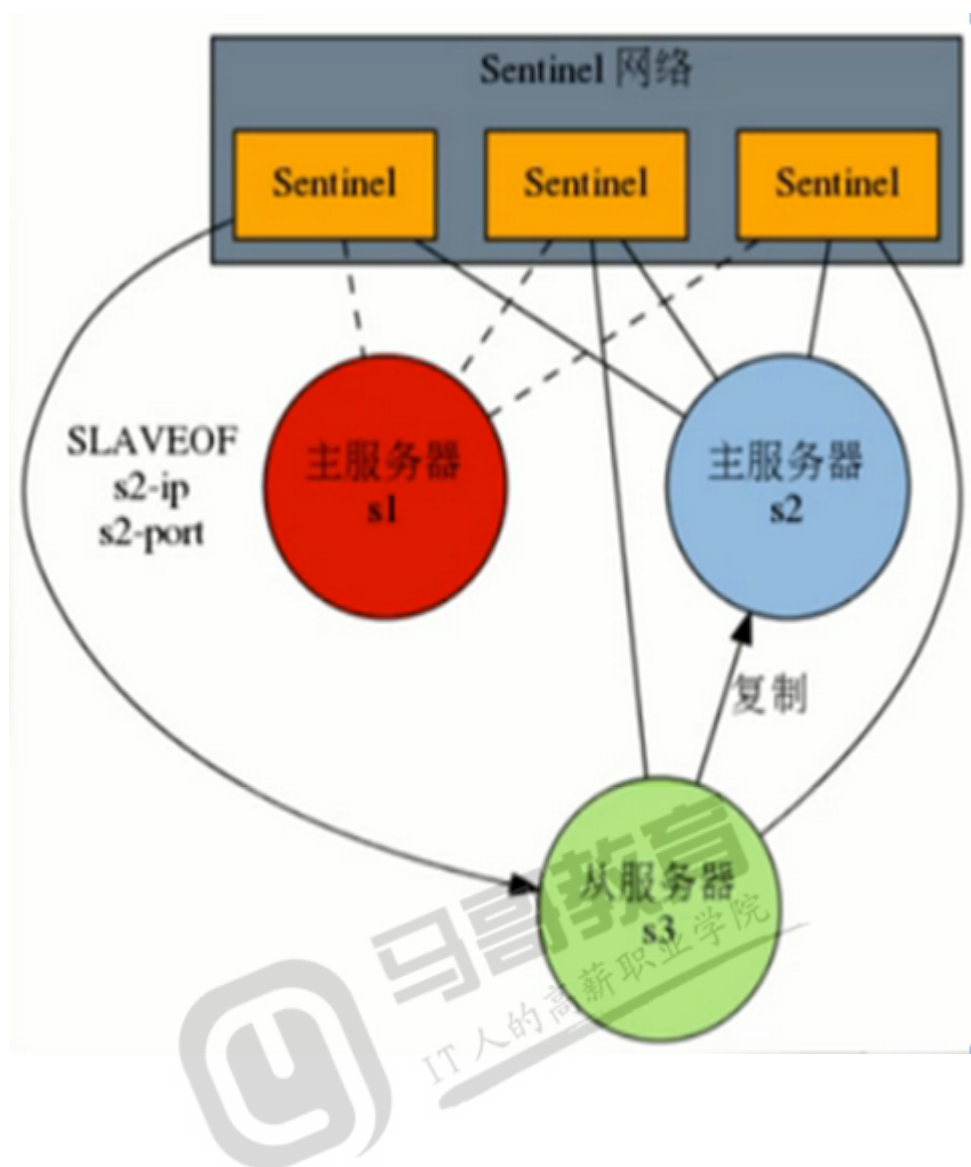
如果下线的服务器为主服务器，那么sentinel网络将对下线主服务器进行自动故障转移，通过将下线主服务器的某个从服务器提升为新的主服务器，并让其从服务器转为复制新的主服务器，以此来让系统重新回到上线的状态

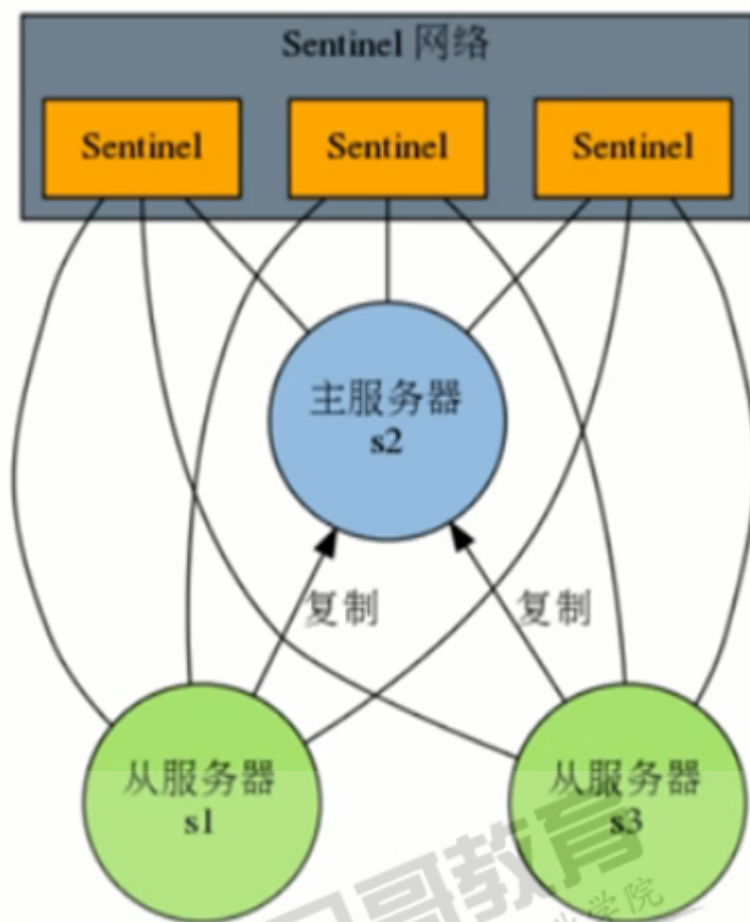
如果原来的主服务器恢复，只能成为一台Slave服务器。

主观下线sdown：单个Sentinel认为服务器下线

客观下线odown：多个Sentinel通信后做出了服务器下线的判断







## 配置

至少包含一个监控配置选项，用于指定被监控Master的相关信息

`sentinel monitor<name><ip><port><quorum>`，例如 `sentinel monitor mymaster 127.0.0.1 6379 2` 监视 mymaster 的主服务器，服务器ip和端口，将这个主服务器判断为下线失效至少需要2个Sentinel同意，如果多数 Sentinel 同意才会执行故障转移

Sentinel会根据Master的配置自动发现Master的Slaves

`sentinel down-after-milliseconds mymaster 60000`

认为服务器下线的毫秒数。Sentinel在指定的毫秒数内没有返回给Sentinel的Ping回复，视为主管下线Sdown。

`sentinel failover-timeout mymaster 180000`

若sentinel在该配置值内未能完成failover操作（即故障时主从自动切换），则认为本次failover失败

`sentinel parallel-syncs mymaster 1`

在执行故障转移时，最多可以有多少个从服务器同时对新的主服务器进行同步。1表示只能有1台从服务器从新主服务器同步数据，以便其他从服务器继续提供客户端服务的响应。

`port 26379`

Sentinel默认端口号为26379

## Sentinel实验

Master 192.168.140.135 6379

Slave 192.168.140.140 6379

先启动主从2个Redis服务

提供Sentinel配置文件

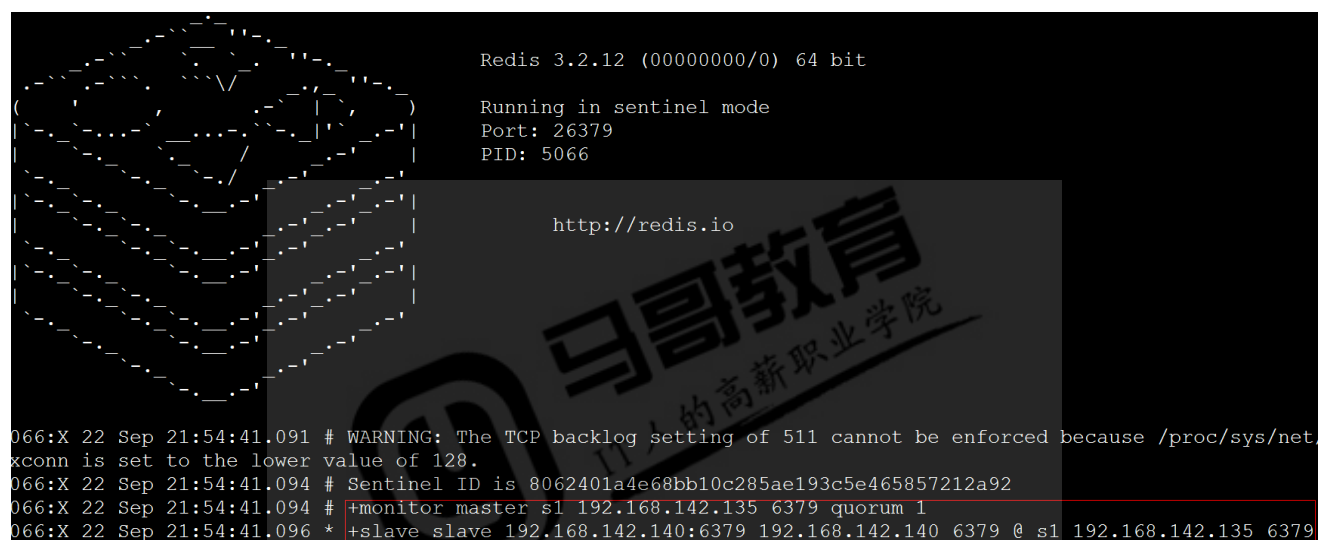
sentinel1.conf文件内容如下

```
port 26379
Sentinel monitor s1 192.168.142.135 6379 1
sentinel down-after-milliseconds s1 6000
sentinel failover-timeout s1 10000
sentinel parallel-syncs s1 1
```

启动Sentinel

```
redis-sentinel sentinel1.conf
```

在从服务器的配置中增加 `slaveof 192.168.142.135 6379` , 并重启该服务



```
Redis 3.2.12 (00000000/0) 64 bit
Running in sentinel mode
Port: 26379
PID: 5066
http://redis.io

066:X 22 Sep 21:54:41.091 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net.
xconn is set to the lower value of 128.
066:X 22 Sep 21:54:41.094 # Sentinel ID is 8062401a4e68bb10c285ae193c5e465857212a92
066:X 22 Sep 21:54:41.094 # +monitor master s1 192.168.142.135 6379 quorum 1
066:X 22 Sep 21:54:41.096 * +slave slave 192.168.142.140:6379 192.168.142.140 6379 @ s1 192.168.142.135 6379
```

模拟Master下线

```
# ps aux | grep redis-server
root      4588  0.1  0.9 38736 9752 ?        Ssl  14:44   0:34 /magedu/redis/bin/redis-server
192.168.142.135:6379
root      4754  0.0  0.0 103244   852 pts/2    S+   20:23   0:00 grep redis-server
# kill -9 4588
```

```
# Sentinel ID is 8062401a4e68bb10c285ae193c5e465857212a92
# +monitor master s1 192.168.142.135 6379 quorum 1
* +slave slave 192.168.142.140:6379 192.168.142.140 6379 @ s1 192.168.142.135 6379
# +sdown master s1 192.168.142.135 6379
# +odown master s1 192.168.142.135 6379 #quorum 1/1
# +new-epoch 1
# +try-failover master s1 192.168.142.135 6379
# +vote-for-leader 8062401a4e68bb10c285ae193c5e465857212a92 1
# +elected-leader master s1 192.168.142.135 6379
# +failover-state-select-slave master s1 192.168.142.135 6379
# +selected-slave slave 192.168.142.140:6379 192.168.142.140 6379 @ s1 192.168.142.135 6379
* +failover-state-send-slaveof-noone slave 192.168.142.140:6379 192.168.142.140 6379 @ s1 1
* +failover-state-wait-promotion slave 192.168.142.140:6379 192.168.142.140 6379 @ s1 192.1
# +promoted-slave slave 192.168.142.140:6379 192.168.142.140 6379 @ s1 192.168.142.135 6379
# +failover-state-reconf-slaves master s1 192.168.142.135 6379
# +failover-end master s1 192.168.142.135 6379
# +switch-master s1 192.168.142.135 6379 192.168.142.140 6379
* +slave slave 192.168.142.135:6379 192.168.142.135 6379 @ s1 192.168.142.140 6379
# +sdown slave 192.168.142.135:6379 192.168.142.135 6379 @ s1 192.168.142.140 6379
```

Master下线后，开始投票，决议通过后，提升从为新主

Master再次上线

```
# +switch-master s1 192.168.142.135 6379 192.168.142.140 6379
* +slave slave 192.168.142.135:6379 192.168.142.135 6379 @ s1 192.168.142.140 6379
# +sdown slave 192.168.142.135:6379 192.168.142.135 6379 @ s1 192.168.142.140 6379
# -sdown slave 192.168.142.135:6379 192.168.142.135 6379 @ s1 192.168.142.140 6379
* +convert-to-slave slave 192.168.142.135:6379 192.168.142.135 6379 @ s1 192.168.142.140 6379
```

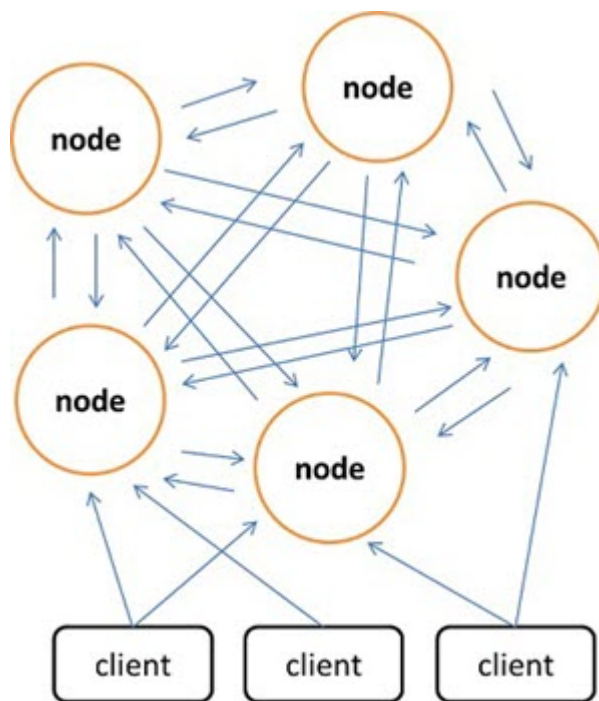
原来的主上线后，被迫转换为从。

## Redis Cluster

从3.0开始，Redis支持分布式集群。

Redis集群采用无中心节点设计，每一个Redis节点间互相通信。

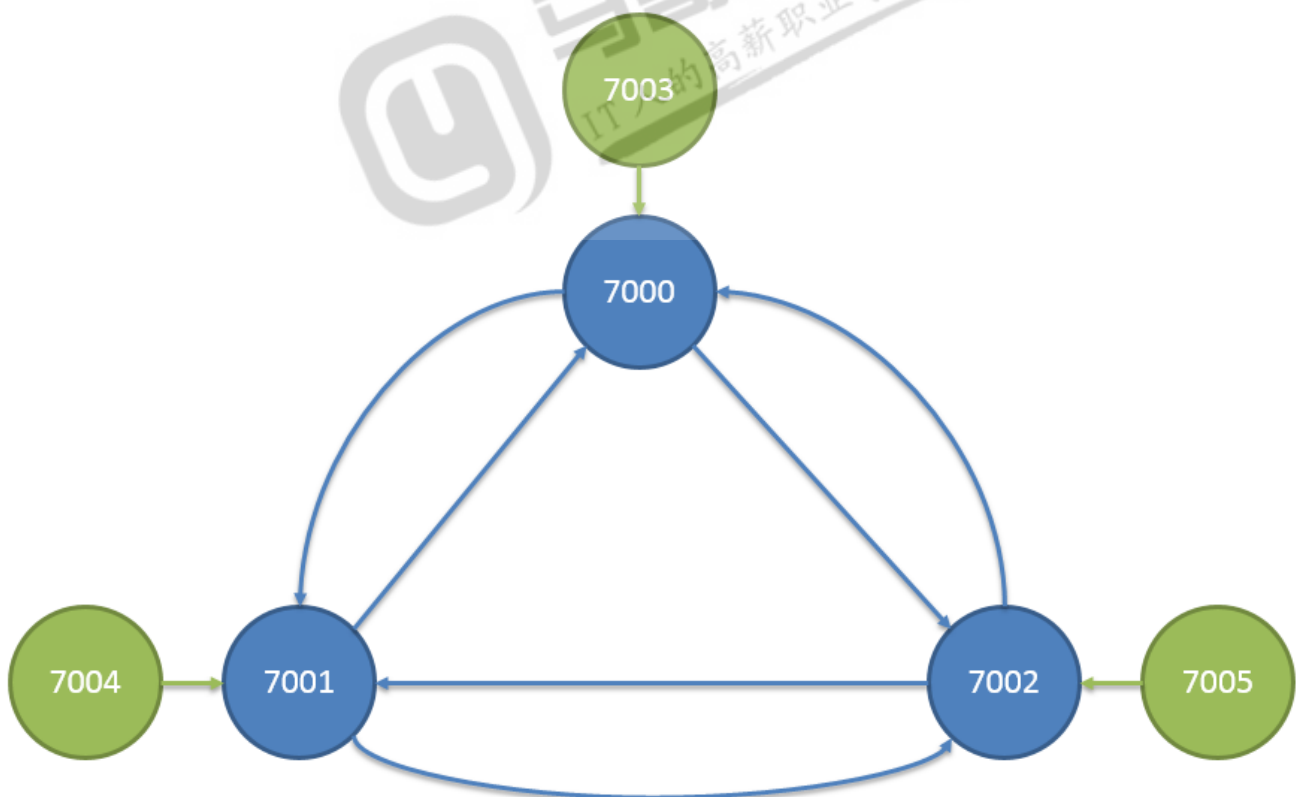
客户端可以连接任意一个集群的节点。



### Redis集群节点复制

Redis集群的每个节点都有两种角色可选：主节点master node、从节点slave node。其中主节点用于存储数据，而从节点则是某个主节点的复制品

当用户需要处理更多读请求的时候，添加从节点可以扩展系统的读性能，因为Redis集群重用了单机Redis复制特性的代码，所以集群的复制行为和我们之前介绍的单机复制特性的行为是完全一样的



### Redis集群故障转移

Redis集群的主节点内置了类似Redis Sentinel的节点故障检测和自动故障转移功能，当集群中的某个主节点下线时，集群中的其他在线主节点会注意到这一点，并对已下线的主节点进行故障转移

集群进行故障转移的方法和Redis Sentinel进行故障转移的方法基本一样，不同的是，在集群里面，故障转移是由

集群中其他在线的主节点负责进行的，所以集群不必另外使用Redis Sentinel

### Redis集群分片

集群将整个数据库分为16384个槽位slot，所有key都数据这些slot中的一个，key的槽位计算公式为 $\text{slot\_number} = \text{crc16}(\text{key}) \% 16384$ ，其中crc16为16位的循环冗余校验和函数

集群中的每个主节点都可以处理0个至16383个槽的访问请求，当16384个槽都有某个节点在负责处理时，集群进入上线状态，并开始处理客户端发送的数据命令请求

举例

三个主节点7000、7001、7002平均分配16384个slot槽位

节点7000指派的槽位为0到5460

节点7001指派的槽位为5461到10922

节点7002指派的槽位为10923到16383

### Redis集群Redirect转向

由于Redis集群无中心节点，请求会发给任意主节点

主节点只会处理自己负责槽位的命令请求，其它槽位的命令请求，该主节点会返回客户端一个转向错误  
客户端根据错误中包含的地址和端口，重新向正确的负责的主节点发起命令请求

### Redis集群总结

Redis集群是一个由多个节点组成的分布式服务集群，它具有复制、高可用和分片特性

Redis的集群没有中心节点，并且带有复制和故障转移特性，这可用避免单个节点成为性能瓶颈，或者因为某个节点下线而导致整个集群下线

集群中的主节点负责处理槽（储存数据），而从节点则是主节点的复制品

Redis集群将整个数据库分为16384个槽，数据库中的每个键都属于16384个槽中的其中一个

集群中的每个主节点都可以管理槽，当16384个槽都有节点在负责时，集群进入上线状态，可以执行客户端发送的数据命令

主节点只会执行和自己负责的槽有关的命令，当节点接收到不属于自己处理的槽的命令时，它将会处理指定槽的节点的地址返回给客户端，而客户端会向正确的节点重新发送