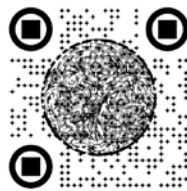


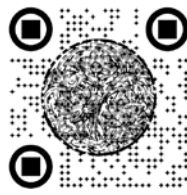
# Redis 教程合集

<http://www.javaboy.org>

江南一点雨  
2019-9-25



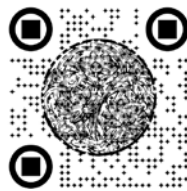
1.Linux 上安装 Redis.....	6
什么是 Redis .....	6
Redis 具有如下特点: .....	6
Redis 具有如下功能: .....	6
Redis 安装 .....	7
2.Redis 中的五种数据类型简介.....	10
五大数据类型介绍.....	10
STRING .....	10
LIST .....	10
HASH .....	10
SET .....	10
ZSET.....	10
key 相关的命令 .....	11
DEL 命令 .....	11
DUMP 命令.....	11
EXISTS 命令 .....	11
TTL 命令 .....	12
EXPIRE 命令 .....	12
PERSIST 命令 .....	12
PEXPIRE 命令 .....	12
PTTL 命令 .....	13
KEYS 命令 .....	13
3.Redis 字符串 STRING 介绍.....	14
STRING .....	14
APPEND.....	14
DECR.....	14
DECRBY .....	14
GET.....	15
GETRANGE.....	15
GETSET .....	15
INCR.....	15



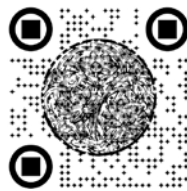
INCRBY .....	16
INCRBYFLOAT .....	16
MGET 与 MSET .....	16
SETEX .....	16
PSETEX .....	16
SETNX .....	17
MSETNX .....	17
SETRANGE .....	17
STRLEN .....	18
4.Redis 字符串 STRING 中 BIT 相关命令 .....	19
准备知识 .....	19
GETBIT .....	19
SETBIT .....	20
BITCOUNT .....	20
BITOP .....	20
BITPOS .....	21
5.Redis 列表与集合 .....	22
列表 .....	22
LPUSH .....	22
LRANGE .....	22
RPUSH .....	22
RPOP .....	23
LPOP .....	23
LINDEX .....	23
LTRIM .....	23
BLPOP .....	24
集合 .....	24
SADD .....	24
SREM .....	24
SISMEMBER .....	25
SCARD .....	25



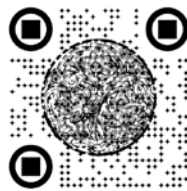
SMEMBERS .....	25
SRANDMEMBER .....	25
SPOP .....	26
SMOVE .....	26
SDIFF .....	26
SDIFFSTORE .....	26
SINTER .....	27
SINTERSTORE .....	27
SUNION .....	27
SUNIONSTORE .....	27
6.Redis 散列与有序集合 .....	29
散列 .....	29
HSET .....	29
HGET .....	29
HMSET .....	29
HMGET .....	29
HDEL .....	30
HSETNX .....	30
HVALS .....	30
HKEYS .....	30
HGETALL .....	31
HEXISTS .....	31
HINCRBY .....	31
HINCRBYFLOAT .....	31
HLEN .....	32
HSTRLEN .....	32
有序集合 .....	32
ZADD .....	32
ZSCORE .....	32
ZRANGE .....	32
ZREVRANGE .....	33
ZCARD .....	33



ZCOUNT.....	33
ZRANGEBYSCORE.....	34
ZRANK.....	34
ZREVRANK.....	34
ZINCRBY.....	35
ZINTERSTORE.....	35
ZREM.....	36
ZLEXCOUNT.....	36
ZRANGEBYLEX.....	36
7.Redis 中的发布订阅和事务.....	38
发布订阅.....	38
tips.....	39
事务.....	39
事务中的异常情况.....	40
WATCH 命令.....	41
8.Redis 快照持久化.....	42
redis 持久化.....	42
快照持久化.....	42
如何配置快照持久化.....	42
快照持久化操作流程.....	44
快照持久化的缺点.....	44
9.Redis 之 AOF 持久化.....	46
AOF 持久化.....	46
AOF 备份的几个关键点.....	47
AOF 文件的重写与压缩.....	47
最佳实践.....	48
10.Redis 主从复制(一).....	50
主从复制.....	50
配置方式.....	50
主从复制注意点.....	53



复制原理.....	53
11.Redis 主从复制(二) .....	55
一场接力赛 .....	55
哨兵模式.....	57
注意问题.....	58
12.Redis 集群搭建.....	59
集群原理.....	59
怎么样投票 .....	60
怎么样判定节点不可用 .....	60
ruby 环境.....	60
集群搭建.....	61
查询集群信息 .....	63
添加主节点 .....	63
删除节点.....	65
13.Jedis 使用 .....	66
有哪些解决方案 .....	66
配置.....	67
Java 端配置 .....	67
14.Spring Data Redis 使用 .....	70
Spring Data Redis 介绍.....	70
环境搭建.....	70



## 1.Linux 上安装 Redis

2019-06-15 20:50:16

hello，各位小伙伴们好久不见！那么从今天开始，我想和各位小伙伴分享下 Redis 的用法，本文我们就先来看看什么是 Redis 以及如何安装 Redis。

### 什么是 Redis

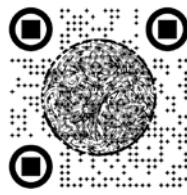
Redis 是一个使用 ANSI C 编写的开源、支持网络、基于内存、可选持久性的键值对存储数据库。从 2015 年 6 月开始，Redis 的开发由 Redis Labs 赞助，而 2013 年 5 月至 2015 年 6 月期间，其开发由 Pivotal 赞助。在 2013 年 5 月之前，其开发由 VMware 赞助。根据月度排行网站 DB-Engines.com 的数据显示，Redis 是最流行的键值对存储数据库。

### Redis 具有如下特点：

1. Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，不会造成数据丢失
2. Redis 支持五种不同的数据结构类型之间的映射，包括简单的 key/value 类型的数据，同时还提供 list, set, zset, hash 等数据结构的存储
3. Redis 支持 master-slave 模式的数据备份

### Redis 具有如下功能：

1. 内存存储和持久化：redis 支持异步将内存中的数据写到硬盘上，在持久化的同时不影响继续服务
2. 取最新 N 个数据的操作，如：可以将最新的 10 条评论的 ID 放在 Redis 的 List 集合里面
3. 数据可以设置过期时间
4. 自带发布、订阅消息系统



## 5. 定时器、计数器

### Redis 安装

Windows 版 Redis 的安装，整体来说还是非常简单的，网上也有很多教程，考虑到 Redis 的大部分使用场景都是在 Linux 上，因此这里我对 Windows 上的安装不做介绍，小伙伴们有兴趣可以自行搜索，下面我们主要来看下 Linux 上怎么安装 Redis。

环境：

- CentOS7 - redis4.0.8

1. 首先下载 Redis，下载地址 <https://redis.io/>，下载获得 redis-4.0.8.tar.gz 后将它放入我们的 Linux 目录 /opt

```
[root@bogon opt]# ls -la
总用量 25576
drwxr-xr-x.  5 root root    132 3月 18 23:34 .
dr-xr-xr-x. 18 root root    282 11月 2 16:36 ..
drwxr-xr-x.  8 root root    158 11月 6 10:13 mongodb
drwxrwxr-x.  6 root root   4096 1月 12 10:06 redis-4.0.6
-rw-r--r--.  1 root root 1723533 1月 9 16:08 redis-4.0.6.tar.gz
-rw-r--r--.  1 root root 1729973 3月 18 23:34 redis-4.0.8.tar.gz
drwxr-xr-x. 12 1001 1001   4096 1月 16 09:32 zk
-rw-r--r--.  1 root root 22724574 1月 16 09:29 zookeeper-3.4.9.tar.gz
[root@bogon opt]#
```

2. /opt 目录下，对文件进行解压，解压命令: tar -zxvf redis-4.0.8.tar.gz，如下：

```
[root@bogon opt]# tar -zxvf redis-4.0.8.tar.gz
redis-4.0.8/
redis-4.0.8/.gitignore
redis-4.0.8/00-RELEASENOTES
redis-4.0.8/BUGS
redis-4.0.8/CONTRIBUTING
```

3. 解压完成后出现文件夹：redis-4.0.8，进入到该目录中: cd redis-4.0.8

```
[root@bogon opt]# cd redis-4.0.8/
[root@bogon redis-4.0.8]#
```

4. 在 redis-4.0.8 目录下执行 make 命令进行编译





```
[root@bogon redis-4.0.8]# make
cd src && make all
make[1]: 进入目录"/opt/redis-4.0.8/src"
    CC Makefile.dep
make[1]: 离开目录"/opt/redis-4.0.8/src"
make[1]: 进入目录"/opt/redis-4.0.8/src"
```

5.如果 make 完成后继续执行 make install 进行安装

```
[root@bogon redis-4.0.8]# make install
cd src && make install
make[1]: 进入目录"/opt/redis-4.0.8/src"
    CC Makefile.dep
make[1]: 离开目录"/opt/redis-4.0.8/src"
make[1]: 进入目录"/opt/redis-4.0.8/src"

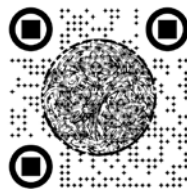
Hint: It's a good idea to run 'make test' ;)

INSTALL install
INSTALL install
INSTALL install
INSTALL install
INSTALL install
make[1]: 离开目录"/opt/redis-4.0.8/src"
[root@bogon redis-4.0.8]#
```

OK，至此，我们的 redis 就算安装成功了。

6.在我们启动之前，需要先做一个简单的配置：修改 redis.conf 文件，将里面的 daemonize no 改成 yes，让服务在后台启动，如下：

```
[root@bogon redis-4.0.8]# vi redis.conf
[root@bogon redis-4.0.8]#
```



```
# On other kernels the period depends on the kernel configuration.
#
# A reasonable value for this option is 300 seconds, which is the new
# Redis default starting with Redis 3.2.1.
tcp-keepalive 300

##### GENERAL #####

# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
daemonize yes
-- INSERT --
```

7.启动，通过 `redis-server redis.conf` 命令启动 redis，如下：

```
[root@bogon redis-4.0.8]# redis-server redis.conf
7144:C 18 Mar 23:51:15.316 # oO0OoO0OoO0Oo Redis is starting oO0OoO0O
7144:C 18 Mar 23:51:15.316 # Redis version=4.0.8, bits=64, commit=00
7144:C 18 Mar 23:51:15.316 # Configuration loaded
[root@bogon redis-4.0.8]#
[root@bogon redis-4.0.8]#
```

8.测试

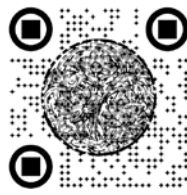
首先我们可以通过 `redis-cli` 命令进入到控制台，然后通过 `ping` 命令进行连通性测试，如果看到 `pong`，表示连接成功了，如下：

```
[root@bogon redis-4.0.8]# redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

9.关闭，通过 `shutdown` 命令我们可以关闭实例，如下：

```
127.0.0.1:6379> SHUTDOWN
not connected> exit
[root@bogon redis-4.0.8]#
```

OK，至此，我们的 Redis 就安装成功了，整体来说还是非常简单的，有问题欢迎留言讨论。



## 2.Redis 中的五种数据类型简介

2019-06-15 20:50:23

上篇文章我们介绍了如何在 Linux 中安装 Redis，本文我们来了解下 Redis 中的五种数据类型。

本文是 Redis 系列的第二篇文章，了解前面的文章有助于更好的理解本文：

### 五大数据类型介绍

redis 中的数据都是以 key/value 的形式存储的，五大数据类型主要是指 value 的数据类型，包含如下五种：

#### STRING

STRING 是 redis 中最基本的数据类型，redis 中的 STRING 类型是二进制安全的，即它可以包含任何数据，比如一个序列化的对象甚至一个 jpg 图片，要注意的是 redis 中的字符串大小上限是 512M。

#### LIST

LIST 是一个简单的字符串列表，按照插入顺序进行排序，我们可以从 LIST 的头部 (LEFT) 或者尾部 (RIGHT) 插入一个元素，也可以从 LIST 的头部 (LEFT) 或者尾部 (RIGHT) 弹出一个元素。

#### HASH

HASH 类似于 Java 中的 Map，是一个键值对集合，在 redis 中可以用来存储对象。

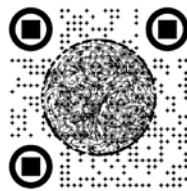
#### SET

SET 是 STRING 类型的无序集合，不同于 LIST，SET 中的元素不可以重复。

#### ZSET

ZSET 和 SET 一样，也是 STRING 类型的元素的集合，不同的是 ZSET 中的每个元素都会关联一个 double 类型的分数，ZSET 中的成员都是唯一的，但是所关联的分数可以重复。

OK，通过上面的介绍，相信小伙伴们对五大数据类型都有一个大致的认识了，接下来我们就来看看这五种数据类型要怎么操作。



## key 相关的命令

由于五大数据类型的数据结构本身有差异，因此对应的命令也会不同，但是有一些命令不管对于哪种数据类型都是存在的，我们今天就先来看看这样一些特殊的命令。

首先通过 `redis-server redis.conf` 命令启动 redis，再通过 `redis-cli` 命令进入到控制台中，如下：

```
[root@localhost redis-4.0.8]# redis-server redis.conf
3717:C 20 Mar 20:17:27.297 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO
3717:C 20 Mar 20:17:27.298 # Redis version=4.0.8, bits=64, commit=00000
3717:C 20 Mar 20:17:27.298 # Configuration loaded
[root@localhost redis-4.0.8]# redis-cli
127.0.0.1:6379>
```

首先我们可以通过 `set` 命令插入一条记录：

```
127.0.0.1:6379> set k1 v1
OK
```

## DEL 命令

看到 OK 表示插入成功。通过 `DEL` 命令我们可以删除一个已经存在的 key，如下：

```
127.0.0.1:6379> DEL k1
(integer) 1
```

看到 (integer) 1 表示数据已经删除成功。

## DUMP 命令

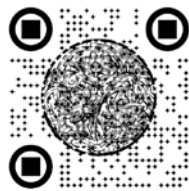
`DUMP` 命令可以序列化给定的 key，并返回序列化之后的值：

```
127.0.0.1:6379> DUMP k1
"\x00\x02v1\b\x00\xe6\xc8\\\xe1bI\xf3c"
```

## EXISTS 命令

`EXISTS` 命令用来检测一个给定的 key 是否存在，如下：

```
127.0.0.1:6379> EXISTS k1
(integer) 1
127.0.0.1:6379> EXISTS k2
(integer) 0
127.0.0.1:6379>
```



上面的运行结果表示 k1 存在而 k2 不存在。

## TTL 命令

TTL 命令可以查看一个给定 key 的有效时间：

```
127.0.0.1:6379> TTL k1
(integer) -1
127.0.0.1:6379> TTL k2
(integer) -2
```

-2 表示 key 不存在或者已过期；-1 表示 key 存在并且没有设置过期时间（永久有效）。当然，我们可以通过下面的命令给 key 设置一个过期时间：

## EXPIRE 命令

EXPIRE 命令可以给 key 设置有效期，在有效期过后，key 会被销毁。

```
127.0.0.1:6379> EXPIRE k1 30
(integer) 1
127.0.0.1:6379> TTL k1
(integer) 25
127.0.0.1:6379>
```

30 表示 30 秒，TTL k1 返回 25 表示这个 key 的有效期还剩 25 秒。

## PERSIST 命令

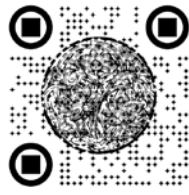
PERSIST 命令表示移除一个 key 的过期时间，这样该 key 就永远不会过期：

```
127.0.0.1:6379> EXPIRE k1 60
(integer) 1
127.0.0.1:6379> ttl k1
(integer) 57
127.0.0.1:6379> PERSIST k1
(integer) 1
127.0.0.1:6379> ttl k1
(integer) -1
```

## PEXPIRE 命令

PEXPIRE 命令的功能和 EXPIRE 命令的功能基本一致，只不过这里设置的参数是毫秒：

```
127.0.0.1:6379> PEXPIRE k1 60000
(integer) 1
```



---

## PTTL 命令

PTTL 命令和 TTL 命令基本一致，只不过 PTTL 返回的是毫秒数：

```
127.0.0.1:6379> PTTL k1  
(integer) 25421
```

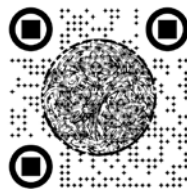
## KEYS 命令

KEYS 命令可以获取满足给定模式的所有 key，比如：

```
127.0.0.1:6379> KEYS *  
1) "k3"  
2) "k2"  
3) "k1"
```

KEYS \* 表示获取所有的 KEY，\* 也可以是一个正则表达式。

OK,key 相关的命令我们就介绍这么多，当然还有很多其他的，小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。



## 3.Redis 字符串 STRING 介绍

2019-06-15 20:51:01

上篇文章我们介绍了五种数据类型中一些通用的命令，本文我们来看看 STRING 数据类型独有的操作命令。

本文是 Redis 系列的第三篇文章，了解前面的文章有助于更好的理解本文：

### STRING

#### APPEND

使用 APPEND 命令时，如果 key 已经存在，则会直接在 value 后追加值，如果 key 不存在，则会先创建一个 value 为空字符串的 key，然后再追加：

```
127.0.0.1:6379> APPEND k1 hello
(integer) 5
127.0.0.1:6379> GET k1
"hello"
127.0.0.1:6379> APPEND k1 world
(integer) 10
127.0.0.1:6379> GET k1
"helloworld"
```

#### DECR

DECR 命令可以实现对 value 的减 1 操作，如果 key 不存在，则 key 对应的初始值会被置为 0，如果 key 的 value 不为数字，则会报错，如下：

```
127.0.0.1:6379> SET k3 19
OK
127.0.0.1:6379> DECR k3
(integer) 18
127.0.0.1:6379> GET k3
"18"
127.0.0.1:6379> SET k4 aa
OK
127.0.0.1:6379> DECR k4
(error) ERR value is not an integer or out of range
```

#### DECRBY

DECRBY 和 DECR 类似，不同的是 DECRBY 可以指定步长，如下：



```
127.0.0.1:6379> GET k3
"8"
127.0.0.1:6379> DECRBY k3 4
(integer) 4
127.0.0.1:6379> GET k3
"4"
```

## GET

GET 命令用来获取对应 key 的 value，如果 key 不存在则返回 nil，如下：

```
127.0.0.1:6379> GET k5
(nil)
```

## GETRANGE

GETRANGE 用来返回 key 所对应的 value 的子串，子串由 start 和 end 决定，从左往右计算，如果下标是负数，则从右往左计算，其中 -1 表示最后一个字符，-2 是倒数第二个...，如下：

```
127.0.0.1:6379> SET k1 helloworld
OK
127.0.0.1:6379> GETRANGE k1 0 2
"hel"
127.0.0.1:6379> GETRANGE k1 -3 -1
"rld"
```

## GETSET

GETSET 命令可以用来获取 key 所对应的 value，并对 key 进行重置，如下：

```
127.0.0.1:6379> SET k1 v1
OK
127.0.0.1:6379> GET k1
"v1"
127.0.0.1:6379> GETSET k1 vv
"v1"
127.0.0.1:6379> GET k1
"vv"
```

## INCR

INCR 操作可以对指定 key 的 value 执行加 1 操作，如果指定的 key 不存在，那么在加 1 操作之前，会先将 key 的 value 设置为 0，如果 key 的 value 不是数字，则会报错。如下：





```
127.0.0.1:6379> INCR k2  
(integer) 1
```

## INCRBY

INCRBY 和 INCR 功能类似，不同的是可以指定增长的步长，如下：

```
127.0.0.1:6379> INCRBY k2 99  
(integer) 100
```

## INCRBYFLOAT

INCRBYFLOAT 命令可以用来增长浮点数，如下：

```
127.0.0.1:6379> SET k1 0.5  
OK  
127.0.0.1:6379> INCRBYFLOAT k1 0.33  
"0.83"
```

## MGET 与 MSET

MGET 与 MSET 分别用来批量设置值和批量获取值，如下：

```
127.0.0.1:6379> MSET k1 v1 k2 v2 k3 v3  
OK  
127.0.0.1:6379> MGET k1 k2 k3  
1) "v1"  
2) "v2"  
3) "v3"
```

## SETEX

SETEX 用来给 key 设置 value，同时设置过期时间，等效于先给 key 设置 value，再给 key 设置过期时间，如下：

```
127.0.0.1:6379> SETEX k1 30 v1  
OK  
127.0.0.1:6379> TTL k1  
(integer) 26  
127.0.0.1:6379> GET k1  
"v1"
```

## PSETEX

PSETEX 的作用和 SETEX 类似，不同的是，这里设置过期时间的单位是毫秒，如下：



```
127.0.0.1:6379> PSETEX k1 60000 v1
OK
127.0.0.1:6379> PTTL k1
(integer) 55412
```

## SETNX

SETNX 是 **SET if Not eXists** 的简写，SET 命令在执行时，如果 key 已经存在，则新值会覆盖掉旧值，而对于 SETNX 命令，如果 key 已经存在，则不做任何操作，如果 key 不存在，则效果等同于 SET 命令。如下：

```
127.0.0.1:6379> SETNX k1 v1
(integer) 1
127.0.0.1:6379> SETNX k1 vv
(integer) 0
127.0.0.1:6379> GET k1
"v1"
```

## MSETNX

MSETNX 兼具了 SETNX 和 MSET 的特性，但是 MSETNX 在执行时，如果有一个 key 存在，则所有的都不会执行，如下：

```
127.0.0.1:6379> MSETNX k1 v1 k2 v2
(integer) 0
```

因为 k1 已经存在，所以 k2 也没执行成功。

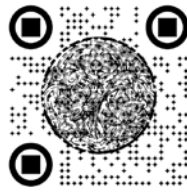
## SETRANGE

SETRANGE 用来覆盖一个已经存在的 key 的 value，如下：

```
127.0.0.1:6379> set k1 helloworld
OK
127.0.0.1:6379> get k1
"helloworld"
127.0.0.1:6379> SETRANGE k1 5 redis
(integer) 10
127.0.0.1:6379> get k1
"helloredis"
```

但是如果已经存在的 key 的 value 长度小于 offset，则不足的地方用 0 补齐，如下：

```
127.0.0.1:6379> set k1 helloredis
OK
```



---

```
127.0.0.1:6379> SETRANGE k1 20 --java
(integer) 26
127.0.0.1:6379> GET k1
"helloredis\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00--java"
```

## STRLEN

STRLEN 用来计算 key 的 value 的长度，如下：

```
127.0.0.1:6379> STRLEN k1
(integer) 26
```

OK,STRING 相关的命令我们就介绍这么多，当然还有很多其他的，小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。



## 4.Redis 字符串 STRING 中 BIT 相关命令

2019-06-15 20:51:15

上篇文章我们对 **STRING** 数据类型中一些基本的命令进行了介绍，但是没有涉及到 **BIT** 相关的命令，本文我们就来看看几个和 **BIT** 相关的命令。

本文是 **Redis** 系列的第四篇文章，了解前面的文章有助于更好的理解本文：

**BIT** 相关的命令是指 **BITCOUNT**/**BITFIELD**/**BITOP**/**BITPOS**/**SETBIT**/**GETBIT** 几个命令，灵活使用这几个命令，可以给我们的项目带来很多惊喜。

### 准备知识

在学习这几个命令之前，我们得先了解下 **redis** 中字符串的存储方式，**redis** 中的字符串都是以二进制的方式进行存储的，比如说我执行如下命令：

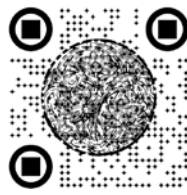
```
127.0.0.1:6379> SET k1 a
OK
```

**a** 对应的 **ASCII** 码是 **97**，转换为二进制数据是 **01100001**，我们 **BIT** 相关命令都是对这个二进制数据进行操作。请继续往下看。

### GETBIT

**GETBIT** 命令可以返回 **key** 对应的 **value** 在 **offset** 处的 **bit** 值，上文提到的 **k1** 为例，**a** 对应的二进制数据是 **01100001**，所以当 **offset** 为 **0** 时，对应的 **bit** 值为 **0**；**offset** 为 **1** 时，对应的 **bit** 值为 **1**；**offset** 为 **2** 时，对应的 **bit** 值为 **1**；**offset** 为 **3** 时，对应的 **bit** 值为 **0**，依此类推....，如下：

```
127.0.0.1:6379> GETBIT k1 0
(integer) 0
127.0.0.1:6379> GETBIT k1 1
(integer) 1
127.0.0.1:6379> GETBIT k1 2
(integer) 1
127.0.0.1:6379> GETBIT k1 3
(integer) 0
127.0.0.1:6379> GETBIT k1 4
(integer) 0
127.0.0.1:6379> GETBIT k1 5
(integer) 0
127.0.0.1:6379> GETBIT k1 6
(integer) 0
```



```
127.0.0.1:6379> GETBIT k1 7  
(integer) 1
```

## SETBIT

SETBIT 可以用来修改二进制数据，比如 a 对应的 ASCII 码为 97，c 对应的 ASCII 码为 99，97 转为二进制是 01100001，99 转为二进制是 01100011，两个的差异在于第六位一个是 0 一个是 1，通过 SETBIT 命令，我们可以将 k1 的第六位的 0 改为 1（第六位是从 0 开始算），如下：

```
127.0.0.1:6379> SETBIT k1 6 1  
(integer) 0  
127.0.0.1:6379> GET k1  
"c"
```

此时，k1 中存储的字符也就变为了 c。SETBIT 在执行时所返回的数字，表示该位上原本的 bit 值。

## BITCOUNT

BITCOUNT 可以用来统计这个二进制数据中 1 的个数，如下：

```
127.0.0.1:6379> BITCOUNT k1  
(integer) 4
```

关于 BITCOUNT，redis 官网上有一个非常有意思的案例：用户上线次数统计。节选部分原文如下：

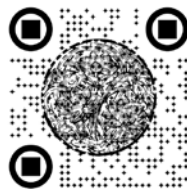
举个例子，如果今天是网站上线的第 100 天，而用户 peter 在今天浏览过网站，那么执行命令 SETBIT peter 100 1；如果明天 peter 也继续浏览网站，那么执行命令 SETBIT peter 101 1，以此类推。当要计算 peter 总共以来的上线次数时，就使用 BITCOUNT 命令：执行 BITCOUNT peter，得出的结果就是 peter 上线的总天数。

这种统计方式最大的好处就是节省空间并且运算速度快。每天占用一个 bit，一年也就 365 个 bit，10 年也就 10\*365 个 bit，也就是 456 个字节，对于这么大的数据，bit 的操作速度非常快。

## BITOP

BITOP 可以对一个或者多个二进制位串执行并 (AND)、或 (OR)、异或 (XOR) 以及非 (NOT) 运算，如下：a 对应的 ASCII 码转为二进制是 01100001，c 对应的二进制位串是 01100011。对这两个二进制位串分别执行 AND 的结果如下：

```
127.0.0.1:6379> set k1 a  
OK
```



```
127.0.0.1:6379> set k2 c
OK
127.0.0.1:6379> BITOP and k3 k1 k2
(integer) 1
127.0.0.1:6379> get k3
"a"
127.0.0.1:6379> BITOP or k3 k1 k2
(integer) 1
127.0.0.1:6379> get k3
"c"
127.0.0.1:6379> BITOP xor k3 k1 k2
(integer) 1
127.0.0.1:6379> get k3
"\x02"
```

另外，BITOP 也可以执行 NOT 运算，但是注意参数个数，如下：

```
127.0.0.1:6379> BITOP not k3 k4
(integer) 1
```

这里会对 k4 的二进制位串取反，将取反结果交给 k3。

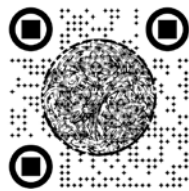
## BITPOS

BITPOS 用来获取二进制位串中第一个 1 或者 0 的位置，如下：

```
127.0.0.1:6379> set k1 a
OK
127.0.0.1:6379> BITPOS k1 1
(integer) 1
127.0.0.1:6379> BITPOS k1 0
(integer) 0
```

也可以在后面设置一个范围，不过后面的范围是字节的范围，而不是二进制位串的范围。

OK,STRING 中 BIT 相关的命令我们就介绍这么多，更多命令小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。



## 5.Redis 列表与集合

2019-06-15 20:51:20

前面文章我们介绍了 **STRING** 的基本命令，本文我们来看看 Redis 中的列表与集合。

本文是 Redis 系列的第五篇文章，了解前面的文章有助于更好的理解本文：

### 列表

列表是 Redis 中另外一种数据类型。下面我们来看看列表中一些基本的操作命令。

#### LPUSH

将一个或多个值 **value** 插入到列表 **key** 的表头，如果有多个 **value** 值，那么各个 **value** 值按从左到右的顺序依次插入到表头，如下：

```
127.0.0.1:6379> LPUSH k1 v1 v2 v3
(integer) 3
```

#### LRANGE

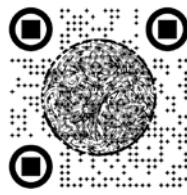
返回列表 **key** 中指定区间内的元素，区间以偏移量 **start** 和 **stop** 指定，下标 (**index**) 参数 **start** 和 **stop** 都以 0 为底，即 0 表示列表的第一个元素，1 表示列表的第二个元素，以此类推。我们也可以使用负数下标，以 -1 表示列表的最后一个元素，-2 表示列表的倒数第二个元素，以此类推。如下：

```
127.0.0.1:6379> LRANGE k1 0 -1
1) "v3"
2) "v2"
3) "v1"
```

#### RPUSH

RPUSH 与 LPUSH 的功能基本一致，不同的是 RPUSH 的中的 **value** 值是按照从右到左的顺序依次插入，如下：

```
127.0.0.1:6379> RPUSH k2 1 2 3 4 5
(integer) 5
127.0.0.1:6379> LRANGE k2 0 -1
1) "1"
2) "2"
3) "3"
```



- 4) "4"
- 5) "5"

## RPOP

RPOP 命令可以移除并返回列表 `key` 的尾元素。如下：

```
127.0.0.1:6379> RPOP k2
"5"
127.0.0.1:6379> LRANGE k2 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
```

## LPOP

LPOP 和 RPOP 类似，不同的是 LPOP 移除并返回列表 `key` 的头元素，如下：

```
127.0.0.1:6379> LPOP k2
"1"
127.0.0.1:6379> LRANGE k2 0 -1
1) "2"
2) "3"
3) "4"
```

## LINDEX

LINDEX 命令可以返回列表 `key` 中，下标为 `index` 的元素，正数下标 0 表示第一个元素，也可以使用负数下标，-1 表示倒数第一个元素，如下：

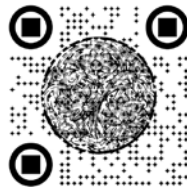
```
127.0.0.1:6379> LINDEX k2 0
"2"
127.0.0.1:6379> LINDEX k2 -1
"4"
```

## LTRIM

LTRIM 命令可以对一个列表进行修剪，即让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。下标与之前介绍的写法都一致，这里不赘述。如下：

```
127.0.0.1:6379> LRANGE k1 0 -1
1) "v3"
2) "v2"
3) "v1"
```





```
127.0.0.1:6379> LTRIM k1 0 1
OK
127.0.0.1:6379> LRANGE k1 0 -1
1) "v3"
2) "v2"
```

## BLPOP

BLPOP 是阻塞式列表的弹出原语。它是命令 LPOP 的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BLPOP 命令阻塞。当给定多个 key 参数时，按参数 key 的先后顺序依次检查各个列表，弹出第一个非空列表的头元素。同时，在使用该命令时也需要指定阻塞的时长，时长单位为秒，在该时长内如果没有元素可供弹出，则阻塞结束。返回的结果是 key 和 value 的组合，如下：

```
127.0.0.1:6379> BLPOP k1 10
1) "k1"
2) "v2"
127.0.0.1:6379> BLPOP k1 10
(nil)
(10.03s)
```

最后，BRPOP、BPOPLPUSH、BRPOPLPUSH 都是相应命令的阻塞版本，这里就不赘述了。

## 集合

接下来我们来看看集合中一些常见的操作命令：

### SADD

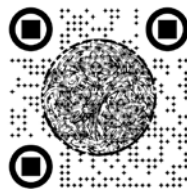
SADD 命令可以添加一个或多个指定的 member 元素到集合的 key 中，指定的一个或者多个元素 member 如果已经在集合 key 中存在则忽略，如果集合 key 不存在，则新建集合 key，并添加 member 元素到集合 key 中。如下：

```
127.0.0.1:6379> SADD k1 v1 v2 v3 v4
(integer) 4
```

### SREM

SREM 命令可以在 key 集合中移除指定的元素，如果指定的元素不是 key 集合中的元素则忽略。如果 key 集合不存在则被视为一个空的集合，该命令返回 0。如下：

```
127.0.0.1:6379> SREM k1 v2
(integer) 1
```



---

```
127.0.0.1:6379> SREM k1 v10
(integer) 0
```

## SISMEMBER

SISMEMBER 命令可以返回成员 **member** 是否是存储的集合 **key** 的成员。如下:

```
127.0.0.1:6379> SISMEMBER k1 v3
(integer) 1
```

## SCARD

SCARD 命令可以返回集合存储的 **key** 的基数(集合元素的数量), 如下:

```
127.0.0.1:6379> SCARD k1
(integer) 3
```

## SMEMBERS

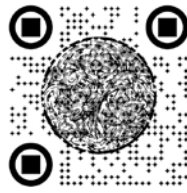
SMEMBERS 命令可以返回 **key** 集合所有的元素, 如下:

```
127.0.0.1:6379> SMEMBERS k1
1) "v4"
2) "v1"
3) "v3"
```

## SRANDMEMBER

SRANDMEMBER 仅需我们提供 **key** 参数,它就会随机返回 **key** 集合中的一个元素,从 Redis2.6 开始,该命令也可以接受一个可选的 **count** 参数,如果 **count** 是整数且小于元素的个数,则返回 **count** 个随机元素,如果 **count** 是整数且大于集合中元素的个数时,则返回集合中的所有元素,当 **count** 是负数,则会返回一个包含 **count** 的绝对值的个数元素的数组,如果 **count** 的绝对值大于元素的个数,则返回的结果集里会出现一个元素出现多次的情况。如下:

```
127.0.0.1:6379> SRANDMEMBER k1
"v4"
127.0.0.1:6379> SRANDMEMBER k1 2
1) "v4"
2) "v1"
127.0.0.1:6379> SRANDMEMBER k1 5
1) "v4"
2) "v1"
3) "v3"
127.0.0.1:6379> SRANDMEMBER k1 -1
1) "v4"
```



---

```
127.0.0.1:6379> SRANDMEMBER k1 -5
```

- 1) "v3"
- 2) "v1"
- 3) "v1"
- 4) "v3"
- 5) "v3"

## SPOP

SPOP 命令的用法和 SRANDMEMBER 类似，不同的是，SPOP 每次选择一个随机的元素之后，该元素会出栈，而 SRANDMEMBER 则不会出栈，只是将该元素展示出来。

## SMOVE

SMOVE 命令可以将 member 从 source 集合移动到 destination 集合中，如下：

```
127.0.0.1:6379> SMOVE k1 k2 v1
(integer) 1
127.0.0.1:6379> SMEMBERS k1
1) "v4"
2) "v3"
127.0.0.1:6379> SMEMBERS k2
1) "v1"
```

## SDIFF

SDIFF 可以用来返回一个集合与给定集合的差集的元素，如下：

```
127.0.0.1:6379> SDIFF k1 k2
1) "v4"
2) "v3"
```

k1 中的元素是 v3、v4，k2 中的元素是 v1，差集就是 v3、v4。

## SDIFFSTORE

SDIFFSTORE 命令与 SDIFF 命令基本一致，不同的是 SDIFFSTORE 命令会将结果保存在一个集合中，如下：

```
127.0.0.1:6379> SDIFFSTORE key k1 k2
(integer) 2
127.0.0.1:6379> SMEMBERS key
1) "v4"
2) "v3"
```



## SINTER

SINTER 命令可以用来计算指定 key 之间元素的交集，如下：

```
127.0.0.1:6379> SMEMBERS k1
1) "v4"
2) "v3"
127.0.0.1:6379> SMEMBERS k2
1) "v1"
2) "v3"
127.0.0.1:6379> SINTER k1 k2
1) "v3"
```

## SINTERSTORE

SINTERSTORE 命令和 SINTER 命令类似，不同的是它会将结果保存到一个新的集合中，如下：

```
127.0.0.1:6379> SINTERSTORE k3 k1 k2
(integer) 1
127.0.0.1:6379> SMEMBERS k3
1) "v3"
```

## SUNION

SUNION 可以用来计算两个集合的并集，如下：

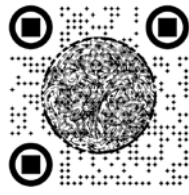
```
127.0.0.1:6379> SUNION k1 k2
1) "v4"
2) "v1"
3) "v3"
```

## SUNIONSTORE

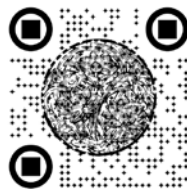
SUNIONSTORE 和 SUNION 命令类似，不同的是它会将结果保存到一个新的集合中，如下：

```
127.0.0.1:6379> SUNIONSTORE k4 k1 k2
(integer) 3
127.0.0.1:6379> SMEMBERS k4
1) "v4"
2) "v1"
3) "v3"
```

OK,列表和集合的命令我们就介绍这么多，更多命令小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。



公众号 · 江南一点雨



## 6.Redis 散列与有序集合

2019-06-15 20:51:25

前面文章我们介绍了列表与集合中的基本命令，本文我们来看看 Redis 中的散列与有序集合。

本文是 Redis 系列的第六篇文章，了解前面的文章有助于更好的理解本文：

### 散列

很多时候，散列就像一个微缩版的 redis，在本文中，小伙伴们对看到的许多散列命令都会有似曾相识的感觉。

#### HSET

HSET 命令可以用来设置 key 指定的哈希集中指定字段的值，如下：

```
127.0.0.1:6379> HSET k1 h1 v1
(integer) 1
```

#### HGET

HGET 命令可以用来返回 key 指定的哈希集中该字段所关联的值，如下：

```
127.0.0.1:6379> HGET k1 h1
"v1"
```

#### HMSET

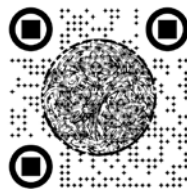
HMSET 命令可以批量设置 key 指定的哈希集中指定字段的值，如下：

```
127.0.0.1:6379> HMSET k2 h1 v1 h2 v2 h3 v3
OK
```

#### HMGET

HMGET 可以批量返回 key 指定的哈希集中指定字段的值，如下：

```
127.0.0.1:6379> HMGET k2 h1 h2 h3
1) "v1"
2) "v2"
3) "v3"
```



## HDEL

HDEL 命令可以从 **key** 指定的哈希集中移除指定的域，在哈希集中不存在的域将被忽略，如下：

```
127.0.0.1:6379> HMGET k2 h1 h2 h3
1) "v1"
2) "v2"
3) "v3"
127.0.0.1:6379> HDEL k2 h1
(integer) 1
127.0.0.1:6379> HMGET k2 h1 h2 h3
1) (nil)
2) "v2"
3) "v3"
```

## HSETNX

HSETNX 命令只在 **key** 指定的哈希集中不存在指定的字段时，设置字段的值，如果字段已存在，该操作无效果。如下：

```
127.0.0.1:6379> HSETNX k2 h3 1
(integer) 0
127.0.0.1:6379> HSETNX k2 h4 1
(integer) 1
```

## HVALS

HVALS 命令可以返回 **key** 指定的哈希集中所有字段的值，如下：

```
127.0.0.1:6379> HVALS k2
1) "v2"
2) "v3"
3) "1"
```

## HKEYS

HKEYS 命令可以返回 **key** 指定的哈希集中所有字段的名称，如下：

```
127.0.0.1:6379> HKEYS k2
1) "h2"
2) "h3"
3) "h4"
```



## HGETALL

HGETALL 命令可以返回 key 指定的哈希集中所有的字段和值。返回值中，每个字段名的下一个是它的值，所以返回值的长度是哈希集大小的两倍，如下：

```
127.0.0.1:6379> HGETALL k2
1) "h2"
2) "v2"
3) "h3"
4) "v3"
5) "h4"
6) "1"
```

## HEXISTS

HEXISTS 命令可以返回 hash 里面 field 是否存在，如下：

```
127.0.0.1:6379> HEXISTS k2 h3
(integer) 1
```

## HINCRBY

HINCRBY 可以增加 key 指定的哈希集中指定字段的数值。如果 key 不存在，会创建一个新的哈希集并与 key 关联。如果字段不存在，则字段的值在该操作执行前被设置为 0，HINCRBY 支持的值的范围限定在 64 位有符号整数，如下：

```
127.0.0.1:6379> HEXISTS k2 h3
(integer) 1
127.0.0.1:6379>
127.0.0.1:6379> HGET k2 h4
"1"
127.0.0.1:6379> HINCRBY k2 h4 5
(integer) 6
127.0.0.1:6379> HGET k2 h4
"6"
127.0.0.1:6379> HGET k2 h5
(nil)
127.0.0.1:6379> HINCRBY k2 h5 99
(integer) 99
127.0.0.1:6379> HGET k2 h5
"99"
```

## HINCRBYFLOAT

HINCRBYFLOAT 与 HINCRBY 用法基本一致，只不过这里允许 float 类型的数据，不赘述。





## HLEN

HLEN 返回 key 指定的哈希集包含的字段的数量，如下：

```
127.0.0.1:6379> HLEN k2  
(integer) 4
```

## HSTRLEN

HSTRLEN 可以返回 hash 指定 field 的 value 的字符串长度，如果 hash 或者 field 不存在，返回 0，如下：

```
127.0.0.1:6379> HSTRLEN k2 h2  
(integer) 2
```

## 有序集合

有序集合类似 Sets ,但是每个字符串元素都关联到一个叫 score 浮动数值。里面的元素总是通过 score 进行着排序，因此它是可以检索的一系列元素。

## ZADD

ZADD 命令可以将所有指定成员添加到键为 key 的有序集合里面。添加时可以指定多个分数/成员（score/member）对。如果指定添加的成员已经是有序集合里面的成员，则会更新该成员的分数（score）并更新到正确的排序位置。如下：

```
127.0.0.1:6379> ZADD k1 60 v1  
(integer) 1
```

## ZSCORE

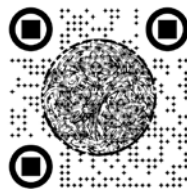
ZSCORE 命令可以返回有序集 key 中，成员 member 的 score 值。如下：

```
127.0.0.1:6379> ZSCORE k1 v1  
"60"
```

## ZRANGE

ZRANGE 命令可以根据 index 返回 member，该命令在执行时加上 withscores 参数可以连同 score 一起返回：

```
127.0.0.1:6379> ZRANGE k1 0 3  
1) "v1"  
2) "v2"  
3) "v3"
```



```
4) "v4"
127.0.0.1:6379> ZRANGE k1 0 3 withscores
1) "v1"
2) "60"
3) "v2"
4) "70"
5) "v3"
6) "80"
7) "v4"
8) "90"
```

## ZREVRANGE

ZREVRANGE 和 ZRANGE 功能基本一致，不同的是 ZREVRANGE 是反着来的，如下：

```
127.0.0.1:6379> ZREVRANGE k1 0 3
1) "v5"
2) "v4"
3) "v3"
4) "v2"
127.0.0.1:6379> ZREVRANGE k1 0 3 withscores
1) "v5"
2) "100"
3) "v4"
4) "90"
5) "v3"
6) "80"
7) "v2"
8) "70"
```

## ZCARD

ZCARD 命令可以返回 key 的有序集元素个数。如下：

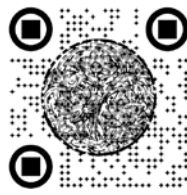
```
127.0.0.1:6379> ZCARD k1
(integer) 5
```

## ZCOUNT

ZCOUNT 命令可以返回有序集 key 中，score 值在 min 和 max 之间(默认包括 score 值等于 min 或 max)的成员。如下：

```
127.0.0.1:6379> ZCOUNT k1 60 90
(integer) 4
```

如果在统计时，不需要包含 60 或者 90，则添加一个 ( 即可，如下：



---

```
127.0.0.1:6379> ZCOUNT k1 60 (90
(integer) 3
```

## ZRANGEBYSCORE

ZRANGEBYSCORE 命令可以按照 score 范围范围元素，加上 withscores 可以连 score 一起返回。如下：

```
127.0.0.1:6379> ZRANGEBYSCORE k1 60 80
1) "v1"
2) "v2"
3) "v3"
127.0.0.1:6379> ZRANGEBYSCORE k1 60 80 withscores
1) "v1"
2) "60"
3) "v2"
4) "70"
5) "v3"
6) "80"
127.0.0.1:6379> ZRANGEBYSCORE k1 (60 80 withscores
1) "v2"
2) "70"
3) "v3"
4) "80"
```

## ZRANK

ZRANK 命令可以返回有序集 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，即 score 值最小的成员排名为 0。如下：

```
127.0.0.1:6379> ZRANK k1 v1
(integer) 0
127.0.0.1:6379> ZRANK k1 v2
(integer) 1
```

## ZREVRANK

ZREVRANK 和 ZRANK 命令功能基本一致，不同的是，ZREVRANK 中的排序是从大到小：

```
127.0.0.1:6379> ZREVRANK k1 v1
(integer) 4
127.0.0.1:6379> ZREVRANK k1 v2
(integer) 3
```



## ZINCRBY

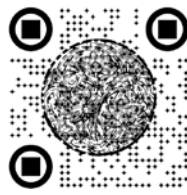
ZINCRBY 命令可以为有序集 key 的成员 member 的 score 值加上增量 increment。如果 key 中不存在 member，就在 key 中添加一个 member，score 是 increment（就好像它之前的 score 是 0.0）。如果 key 不存在，就创建一个只含有指定 member 成员的有序集合：

```
127.0.0.1:6379> ZINCRBY k1 3 v1
"63"
127.0.0.1:6379> ZRANGE k1 0 0 withscores
1) "v1"
2) "63"
```

## ZINTERSTORE

ZINTERSTORE 命令可以计算给定的 numkeys 个有序集合的交集，并且把结果放到 destination 中。在给定要计算的 key 和其它参数之前，必须先给定 key 个数 (numberkeys)。该命令也可以在执行的过程中给原 score 乘以 weights 后再求和，如下：

```
127.0.0.1:6379> ZADD k2 2 v1
(integer) 1
127.0.0.1:6379> ZADD k2 3 v2
(integer) 1
127.0.0.1:6379> ZADD k2 4 v3
(integer) 1
127.0.0.1:6379> ZADD k3 9 v2
(integer) 1
127.0.0.1:6379> ZADD k3 10 v3
(integer) 1
127.0.0.1:6379> ZADD k3 11 v4
(integer) 1
127.0.0.1:6379> ZINTERSTORE k4 2 k2 k3
(integer) 2
127.0.0.1:6379> ZRANGE k4 0 -1 withscores
1) "v2"
2) "12"
3) "v3"
4) "14"
127.0.0.1:6379> ZINTERSTORE k5 2 k2 k3 weights 3 1
(integer) 2
127.0.0.1:6379> ZRANGE k5 0 -1 withscores
1) "v2"
2) "18"
```



- 3) "v3"
- 4) "22"

## ZREM

ZREM 命令可以从集合中弹出一个元素，如下：

```
127.0.0.1:6379> ZRANGE k2 0 -1 withscores
1) "v1"
2) "2"
3) "v2"
4) "3"
5) "v3"
6) "4"
127.0.0.1:6379> ZREM k2 v1
(integer) 1
127.0.0.1:6379> ZRANGE k2 0 -1 withscores
1) "v2"
2) "3"
3) "v3"
4) "4"
```

## ZLEXCOUNT

ZLEXCOUNT 命令用于计算有序集合中指定成员之间的成员数量。如下：

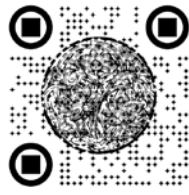
```
127.0.0.1:6379> ZLEXCOUNT k2 - +
(integer) 2
127.0.0.1:6379> ZLEXCOUNT k2 [v2 [v4
(integer) 2
```

**注意：**可以用 `-` 和 `+` 表示得分最小值和最大值，如果使用成员名的话，一定要在成员名之前加上 `[`。

## ZRANGEBYLEX

ZRANGEBYLEX 返回指定成员区间内的成员，按成员字典正序排序，分数必须相同。如下：

```
127.0.0.1:6379> ZRANGEBYLEX k2 [v2 [v4
1) "v2"
2) "v3"
127.0.0.1:6379> ZRANGEBYLEX k2 - +
1) "v2"
2) "v3"
127.0.0.1:6379>
```

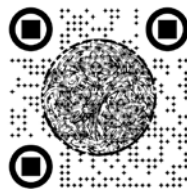


---

注意 `min` 和 `max` 参数的写法和 `ZLEXCOUNT` 一致。

OK,散列和有序集合的命令我们就介绍这么多，更多命令小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。

公众号 · 江南一点雨



## 7.Redis 中的发布订阅和事务

2019-06-15 20:51:31

hello，小伙伴们好久不见！前面我们说了 redis 中的基本数据类型，本文我们来看看 redis 中的发布订阅和事务，因为这两个都比较简单，因此我放在一篇文章中来讲。

本文是 Redis 系列的第七篇文章，了解前面的文章有助于更好的理解本文：

### 发布订阅

redis 的发布订阅系统有点类似于我们生活中的电台，电台可以在某一个频率上发送广播，而我们可以接收任何一个频率的广播，Android 中的 broadcast 也和这类似。

订阅消息的方式如下：

```
127.0.0.1:6379> SUBSCRIBE c1 c2 c3
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "c1"
3) (integer) 1
1) "subscribe"
2) "c2"
3) (integer) 2
1) "subscribe"
2) "c3"
3) (integer) 3
```

这个表示接收 c1，c2，c3 三个频道传来的消息，发送消息的方式如下：

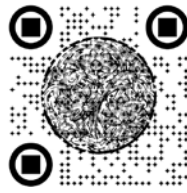
```
127.0.0.1:6379> PUBLISH c1 "hello redis!"
(integer) 1
```

当 c1 这个频道上有消息发出时，此时在消息订阅控制台可以看到如下输出：

```
1) "message"
2) "c1"
3) "hello redis!"
```

在 redis 中，我们也可以使用模式匹配订阅，如下：

```
127.0.0.1:6379> PSUBSCRIBE c*
Reading messages... (press Ctrl-C to quit)
```



- 1) "psubscribe"
- 2) "c\*"
- 3) (integer) 1

此时可以接收到所有以 c 开头的频道发来的消息。

## tips

redis 中的发布订阅系统在某些场景下还是非常好用的，但是也有一些问题需要注意：由于网络在传输过程中可能会遭遇断线等意外情况，断线后需要进行重连，而这会导致断线期间的数据丢失。

## 事务

既然 redis 是一种 NoSQL 数据库，那它当然也有事务的功能，不过这里的事务和我们关系型数据库中的事务有一点点差异。

redis 中事务的用法非常简单，我们通过 MULTI 命令开启一个事务，如下：

```
127.0.0.1:6379> MULTI
OK
```

在 MULTI 命令执行之后，我们可以继续发送命令去执行，此时的命令不会被立马执行，而是放在一个队列中，如下：

```
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
```

当所有的命令都输入完成后，我们可以通过 EXEC 命令发起执行，也可以通过 DISCARD 命令清空队列，如下：

```
127.0.0.1:6379> EXEC
1) OK
2) OK
3) OK
```





## 事务中的异常情况

redis 中事务的异常情况总的来说分为两类：

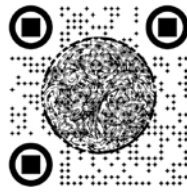
1. 进入队列之前就能发现的错误，比如命令输错；
2. 执行 EXEC 之后才能发现的错误，比如给一个非数字字符加 1；

那么对于这两种不同的异常，redis 中有不同的处理策略。对于第一种错误，服务器会对命令入队失败的情况进行记录，并在客户端调用 EXEC 命令时，拒绝执行并自动放弃这个事务（这个是 2.6.5 之后的版本做法，之前的版本做法小伙伴可以参考官方文档）。如下：

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set kv1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3 3 3
QUEUED
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) OK
3) (error) ERR syntax error
4) OK
127.0.0.1:6379> keys *
1) "k4"
2) "k2"
3) "kv1"
```

而对于第二种情况，redis 并没有对它们进行特别处理，即使事务中有某个/某些命令在执行时产生了错误，事务中的其他命令仍然会继续执行。如下：

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set k1 vv
QUEUED
127.0.0.1:6379> INCR k1
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) (error) ERR value is not an integer or out of range
127.0.0.1:6379> GET k1
"vv"
```



不同于关系型数据库，redis 中的事务出错时没有回滚，对此，官方的解释如下：

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

## WATCH 命令

事务中的 WATCH 命令可以用来监控一个 key，通过这种监控，我们可以为 redis 事务提供(CAS)行为。如果有至少一个被 WATCH 监视的键在 EXEC 执行之前被修改了，那么整个事务都会被取消，EXEC 返回 nil-reply 来表示事务已经失败。如下：

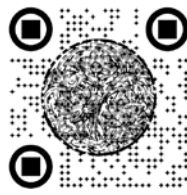
```
127.0.0.1:6379> set k4 2
OK
127.0.0.1:6379> WATCH k4
OK
127.0.0.1:6379> set k4 5
OK
127.0.0.1:6379> get k4
"5"
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set k4 8
OK
127.0.0.1:6379> EXEC
(nil)
127.0.0.1:6379> get k4
"5"
127.0.0.1:6379>
```

WATCH命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务将不会执行，监控一直持续到EXEC命令（事务中的命令是在EXEC之后才执行的，EXEC命令执行完之后被监控的键会自动被UNWATCH）

通过 unwatch 命令，可以取消对一个 key 的监控，如下：

```
127.0.0.1:6379> set k4 20
OK
127.0.0.1:6379> get k4
"20"
127.0.0.1:6379> WATCH k4
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set k4 30
OK
127.0.0.1:6379> EXEC
OK
127.0.0.1:6379> get k4
"30"
127.0.0.1:6379>
```

OK,发布订阅和事务我们就介绍这么多，更多命令小伙伴们可以参考[官方文档](#)。小伙伴们在看官方文档时，有什么问题欢迎留言讨论。



## 8.Redis 快照持久化

2019-06-15 20:51:35

redis 的基础知识我们已经准备的差不多了，接下来两篇文章，我想和大家聊聊 redis 持久化这个话题。

本文是 Redis 系列的第八篇文章，了解前面的文章有助于更好的理解本文：

### redis 持久化

整体上来说，redis 持久化有两种方式，快照持久化和 AOF，在项目中我们可以根据实际情况选择合适的持久化方式，也可以不用持久化，这关键看我们的 redis 在项目中扮演了什么样的角色。那么我将分别用两篇文章来介绍这两种不同的持久化方式，本文我们先来看看第一种方式。

### 快照持久化

快照持久化，顾名思义，就是通过拍摄快照的方式实现数据的持久化，redis 可以在某个时间点上对内存中的数据创建一个副本文件，副本文件中的数据在 redis 重启时会被自动加载，我们也可以将副本文件拷贝到其他地方一样可以使用。

### 如何配置快照持久化

redis 中的快照持久化默认是开启的，redis.conf 中相关配置主要有如下几项：

```
save 900 1
save 300 10
save 60 10000
stop-writes-on-bgsave-error yes
rdbcompression yes
dbfilename dump.rdb
dir ./
```

前面三个 save 相关的选项表示备份的频率，分别表示 900 秒内至少一个键被更改则进行快照，300 秒内至少 10 个键被更改则进行快照，60 秒内至少 10000 个键被更改则进行快照，stop-writes-on-bgsave-error 表示在快照创建出错后，是否继续执行写命令，rdbcompression 则表示是否对快照文件进行压缩，dbfilename 表示生成的快照文件的名字，dir 则表示生成的快照文件的位置，在 redis 中，快照持久化默认就是开启的。我们可以通过如下步骤验证快照持久化的效果：



1.进入 redis 安装目录，如果有 dump.rdb 文件，先将之删除。如下：

```
[root@localhost redis-4.0.8]# ls -l
总用量 300
-rw-rw-r--. 1 root root 150927 2月 3 00:39 00-RELEASENOTES
-rw-rw-r--. 1 root root 53 2月 3 00:39 BUGS
-rw-rw-r--. 1 root root 1815 2月 3 00:39 CONTRIBUTING
-rw-rw-r--. 1 root root 1487 2月 3 00:39 COPYING
drwxrwxr-x. 6 root root 192 5月 12 14:17 deps
-rw-r--r--. 1 root root 92 5月 12 14:18 dump.rdb
-rw-rw-r--. 1 root root 11 2月 3 00:39 INSTALL
-rw-rw-r--. 1 root root 151 2月 3 00:39 Makefile
-rw-rw-r--. 1 root root 4223 2月 3 00:39 MANIFESTO
-rw-rw-r--. 1 root root 20543 2月 3 00:39 README.md
-rw-rw-r--. 1 root root 58354 5月 12 14:19 redis.conf
-rwxrwxr-x. 1 root root 271 2月 3 00:39 runtest
-rwxrwxr-x. 1 root root 280 2月 3 00:39 runtest-cluster
-rwxrwxr-x. 1 root root 281 2月 3 00:39 runtest-sentinel
-rw-rw-r--. 1 root root 7606 2月 3 00:39 sentinel.conf
drwxrwxr-x. 3 root root 8192 5月 12 14:18 src
drwxrwxr-x. 10 root root 167 2月 3 00:39 tests
drwxrwxr-x. 8 root root 4096 2月 3 00:39 utils
```

2.启动 redis，随便向 redis 中存储几个数据，然后关闭 redis 并退出，如下：

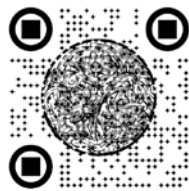
```
[root@localhost redis-4.0.8]# redis-server redis.conf
[root@localhost redis-4.0.8]# redis-cli
127.0.0.1:6379> set k1 v1
OK
127.0.0.1:6379> set k2 v2
OK
127.0.0.1:6379> SHUTDOWN
not connected> exit
```

3.退出来后，我们发现刚刚删掉的 dump.rdb 文件又回来了，这就是生成的备份文件。

4.此时再次启动 redis 并进入，发现刚刚存储的数据都还在，这是因为 redis 在启动时加载了 dump.rdb 中的数据。好了，关闭 redis 并退出。

5.将 redis 目录下的 dump.rdb 文件删除。

6.再次启动 redis 并进入到控制台，所有的数据都不存在了。



## 快照持久化操作流程

通过上面的介绍，小伙伴们对快照持久化都有一个大致的认识了，那么这个东西到底是怎么运行的？持久化的时机是什么？我们来仔细扒一扒。

1.在 redis 运行过程中，我们可以向 redis 发送一条 `save` 命令来创建一个快照，`save` 是一个阻塞命令，redis 在接收到 `save` 命令之后，开始执行备份操作之后，在备份操作执行完毕之前，将不再处理其他请求，其他请求将被挂起，因此这个命令我们用的不多。`save` 命令执行如下：

```
127.0.0.1:6379> SAVE
OK
```

2.在 redis 运行过程中，我们也可以发送一条 `bgsave` 命令来创建一个快照，不同于 `save` 命令，`bgsave` 命令会 `fork` 一个子进程，然后这个子进程负责执行将快照写入硬盘，而父进程则继续处理客户端发来的请求，这样就不会导致客户端命令阻塞了。如下：

```
127.0.0.1:6379> BGSAVE
Background saving started
```

3.如果我们在 `redis.conf` 中配置了如下选项：

```
save 900 1
save 300 10
save 60 10000
```

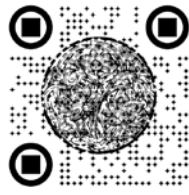
那么当条件满足时，比如 900 秒内有一个 `key` 被操作了，那么 redis 就会自动触发 `bgsave` 命令进行备份。我们可以根据实际需求在 `redis.conf` 中配置多个这种触发规则。

4.还有一种情况也会触发 `save` 命令，那就是我们执行 `shutdown` 命令时，当我们用 `shutdown` 命令关闭 redis 时，此时也会执行一个 `save` 命令进行备份操作，并在备份操作完成后将服务器关闭。

5.还有一种特殊情况也会触发 `bgsave` 命令，就是在主从备份的时候。当从机连接上主机后，会发送一条 `sync` 命令来开始一次复制操作，此时主机会开始一次 `bgsave` 操作，并在 `bgsave` 操作结束后向从机发送快照数据实现数据同步。

## 快照持久化的缺点

快照持久化有一些缺点，比如 `save` 命令会发生阻塞，`bgsave` 虽然不会发生阻塞，但是 `fork` 一个子进程又要耗费资源，在一些极端情况下，`fork` 子进程的时间甚至超过数据备份的时间。定期的持久化也会让我们存在数据丢失的风险，最坏的情况

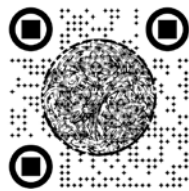


---

我们可能丢失掉最近一次备份到当下的数据，具体丢失多久的数据，要看我们项目的承受能力，我们可以根据项目的承受能力配饰 `save` 参数。

OK,快照持久化我们就介绍这么多，更多资料，小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。

公众号 · 江南一点雨



## 9.Redis 之 AOF 持久化

2019-06-15 20:51:39

上篇文章和小伙伴聊了使用快照的方式实现 redis 数据的持久化，这只是持久化的一种方式，本文我们就来看看另一种持久化方式，AOF(append-only file)。

本文是 Redis 系列的第九篇文章，了解前面的文章有助于更好的理解本文：

### AOF 持久化

与快照持久化不同，AOF 持久化是将被执行的命令写到 aof 文件末尾，在恢复时只需要从头到尾执行一遍写命令即可恢复数据，AOF 在 redis 中默认也是没有开启的，需要我们手动开启，开启方式如下：

打开 redis.conf 配置文件，修改 appendonly 属性值为 yes，如下：

```
appendonly yes
```

另外几个和 AOF 相关的属性如下：

```
appendfilename "appendonly.aof"  
# appendfsync always  
appendfsync everysec  
# appendfsync no  
no-appendfsync-on-rewrite no  
auto-aof-rewrite-percentage 100  
auto-aof-rewrite-min-size 64mb
```

这几个属性的含义分别如下：

- 1.appendfilename 表示生成的 AOF 备份文件的文件名。
- 2.appendfsync 表示备份的时机，always 表示每执行一个命令就备份一次，everysec 表示每秒备份一次，no 表示将备份时机交给操作系统。
- 3.no-appendfsync-on-rewrite 表示在对 aof 文件进行压缩时，是否执行同步操作。
- 4.最后两行配置表示 AOF 文件的压缩时机，这个我们一会再细说。

同时为了避免快照备份的影响，我们将快照备份关闭，关闭方式如下：

```
save ""  
# save 900 1  
# save 300 10  
# save 60 10000
```





此时，当我们在 redis 中进行数据操作时，就会自动生成 AOF 的配置文件 `appendonly.aof`，如下：

```
[root@localhost redis-4.0.8]# ls -l
总用量 300
-rw-rw-r--. 1 root root 150927 2月 3 00:39 00-RELEASENOTES
-rw-r--r--. 1 root root 81 5月 12 22:04 appendonly.aof
-rw-rw-r--. 1 root root 53 2月 3 00:39 BUGS
-rw-rw-r--. 1 root root 1815 2月 3 00:39 CONTRIBUTING
-rw-rw-r--. 1 root root 1487 2月 3 00:39 COPYING
drwxrwxr-x. 6 root root 192 5月 12 14:17 deps
-rw-rw-r--. 1 root root 11 2月 3 00:39 INSTALL
-rw-rw-r--. 1 root root 151 2月 3 00:39 Makefile
-rw-rw-r--. 1 root root 4223 2月 3 00:39 MANIFESTO
-rw-rw-r--. 1 root root 20543 2月 3 00:39 README.md
-rw-rw-r--. 1 root root 58354 5月 12 22:01 redis.conf
-rwxrwxr-x. 1 root root 271 2月 3 00:39 runtest
-rwxrwxr-x. 1 root root 280 2月 3 00:39 runtest-cluster
-rwxrwxr-x. 1 root root 281 2月 3 00:39 runtest-sentinel
-rw-rw-r--. 1 root root 7606 2月 3 00:39 sentinel.conf
drwxrwxr-x. 3 root root 8192 5月 12 14:18 src
drwxrwxr-x. 10 root root 167 2月 3 00:39 tests
drwxrwxr-x. 8 root root 4096 2月 3 00:39 utils
[root@localhost redis-4.0.8]#
```

注意此时没有 `dump.rdb` 文件，这时我们将 redis 关闭并重启，会发现之前的数据都还在，这就是 AOF 备份的结果。

## AOF 备份的几个关键点

- 1.通过上面的介绍，小伙伴们了解到 `appendfsync` 的取值一共有三种，我们在项目中首选 `everysec`，`always` 选项会严重降低 redis 性能。
- 2.使用 `everysec`，最坏的情况下我们可能丢失 1 秒的数据。

## AOF 文件的重写与压缩

AOF 备份有很多明显的优势，当然也有劣势，那就是文件大小。随着系统的运行，AOF 的文件会越来越大，甚至把整个电脑的硬盘填满，AOF 文件的重写与压缩机制可以在一定程度上缓解这个问题。

当 AOF 的备份文件过大时，我们可以向 redis 发送一条 `bgrewriteaof` 命令进行文件重写，如下：

```
127.0.0.1:6379> BGREWRITEAOF
Background append only file rewriting started
(0.71s)
```





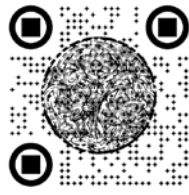
`bgrewriteaof` 的执行原理和我们上文说的 `bgsave` 的原理一致，这里我就不再赘述，因此 `bgsave` 执行过程中存在的问题在这里也一样存在。

`bgrewriteaof` 也可以自动执行，自动执行时间则依赖于 `auto-aof-rewrite-percentage` 和 `auto-aof-rewrite-min-size` 配置，`auto-aof-rewrite-percentage 100` 表示当目前 `aof` 文件大小超过上一次重写时的 `aof` 文件大小的百分之多少时会再次进行重写，如果之前没有重写，则以启动时的 `aof` 文件大小为依据，同时还要求 `AOF` 文件的大小至少要大于 `64M`(`auto-aof-rewrite-min-size 64mb`)。

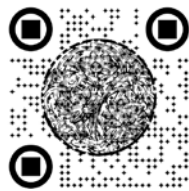
## 最佳实践

1. 如果 `redis` 只做缓存服务器，那么可以不使用任何持久化方式。
2. 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` 的数据不完整时，同时使用两者时服务器重启也只会找 `AOF` 文件。那要不要只使用 `AOF` 呢？作者建议不要，因为 `RDB` 更适用于备份数据库（`AOF` 在不断变化不好备份），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。
3. 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` 这条规则。
4. 如果 `Enable AOF`，好处是在最恶劣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。代价一是带来了持续的 `IO`，二是 `AOF rewrite` 的最后将 `rewrite` 过程中产生的新数据写到新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` 重写的基础大小默认值 `64M` 太小了，可以设到 `5G` 以上。默认超过原大小 `100%` 大小时重写可以改到适当的数值。
5. 如果不 `Enable AOF`，仅靠 `Master-Slave Replication` 实现高可用性也可以。能省掉一大笔 `IO` 也减少了 `rewrite` 时带来的系统波动。代价是如果 `Master/Slave` 同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个 `Master/Slave` 中的 `RDB` 文件，载入较新的那个。

OK, `redis` 数据持久化我们就介绍这么多，更多资料，小伙伴们可以参考[官方文档](#)。小伙伴在看官方文档时，有什么问题欢迎留言讨论。



公众号 · 江南一点雨



## 10.Redis 主从复制(一)

2019-06-15 20:51:43

前面两篇文章和小伙伴们聊了 redis 中的数据备份问题，也对快照备份和 AOF 备份做了对比，本文我们来聊聊 redis 中的主从复制问题，算是数据备份的第三种解决方案。

本文是 Redis 系列的第十篇文章，了解前面的文章有助于更好的理解本文：

### 主从复制

主从复制可以在一定程度上扩展 redis 性能，redis 的主从复制和关系型数据库的主从复制类似，从机能够精确的复制主机上的内容。实现了主从复制之后，一方面能够实现数据的读写分离，降低 master 的压力，另一方面也能实现数据的备份。

### 配置方式

假设我有三个 redis 实例，地址分别如下：

192.168.248.128:6379

192.168.248.128:6380

192.168.248.128:6381

即同一台服务器上三个实例，配置方式如下：

1.将 redis.conf 文件更名为 redis6379.conf，方便我们区分，然后把 redis6379.conf 再复制两份，分别为 redis6380.conf 和 redis6381.conf。如下：



```
-rw-rw-r--. 1 root root 150927 2月 3 00:39 00-RELEASENOTES
-rw-r--r--. 1 root root 81 5月 12 22:44 appendonly.aof
-rw-rw-r--. 1 root root 53 2月 3 00:39 BUGS
-rw-rw-r--. 1 root root 1815 2月 3 00:39 CONTRIBUTING
-rw-rw-r--. 1 root root 1487 2月 3 00:39 COPYING
drwxrwxr-x. 6 root root 192 5月 12 14:17 deps
-rw-rw-r--. 1 root root 11 2月 3 00:39 INSTALL
-rw-rw-r--. 1 root root 151 2月 3 00:39 Makefile
-rw-rw-r--. 1 root root 4223 2月 3 00:39 MANIFESTO
-rw-rw-r--. 1 root root 20543 2月 3 00:39 README.md
-rw-rw-r--. 1 root root 58354 5月 12 22:01 redis6379.conf
-rw-r--r--. 1 root root 58354 5月 13 14:16 redis6380.conf
-rw-r--r--. 1 root root 58354 5月 13 14:16 redis6381.conf
-rwxrwxr-x. 1 root root 271 2月 3 00:39 runtest
-rwxrwxr-x. 1 root root 280 2月 3 00:39 runtest-cluster
-rwxrwxr-x. 1 root root 281 2月 3 00:39 runtest-sentinel
-rw-rw-r--. 1 root root 7606 2月 3 00:39 sentinel.conf
drwxrwxr-x. 3 root root 8192 5月 12 14:18 src
drwxrwxr-x. 10 root root 167 2月 3 00:39 tests
drwxrwxr-x. 8 root root 4096 2月 3 00:39 utils
[root@localhost redis-4.0.8]#
```

2.打开 redis6379.conf，将如下配置均加上 6379,(默认是 6379 的不用修改)，如下：

```
port 6379
pidfile /var/run/redis_6379.pid
logfile "6379.log"
dbfilename dump6379.rdb
appendfilename "appendonly6379.aof"
```

3.同理，分别打开 redis6380.conf 和 redis6381.conf 两个配置文件，将第二步涉及到 6379 的分别改为 6380 和 6381。

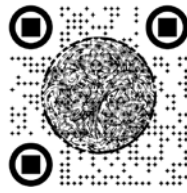
4.输入如下命令，启动三个 redis 实例：

```
[root@localhost redis-4.0.8]# redis-server redis6379.conf
[root@localhost redis-4.0.8]# redis-server redis6380.conf
[root@localhost redis-4.0.8]# redis-server redis6381.conf
```

5.输入如下命令，分别进入三个实例的控制台：

```
[root@localhost redis-4.0.8]# redis-cli -p 6379
[root@localhost redis-4.0.8]# redis-cli -p 6380
[root@localhost redis-4.0.8]# redis-cli -p 6381
```

此时我就成功配置了三个 redis 实例了。



6.假设在这三个实例中，6379 是主机，即 master，6380 和 6381 是从机，即 slave，那么如何配置这种实例关系呢，很简单，分别在 6380 和 6381 上执行如下命令：

```
127.0.0.1:6381> SLAVEOF 127.0.0.1 6379
OK
```

这一步也可以通过在两个从机的 redis.conf 中添加如下配置来解决：

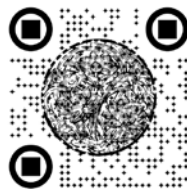
```
slaveof 127.0.0.1 6379
```

OK，主从关系搭建好后，我们可以通过如下命令可以查看每个实例当前的状态，如下：

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=56,lag=1
slave1:ip=127.0.0.1,port=6381,state=online,offset=56,lag=0
master_replid:26ca818360d6510b717e471f3f0a6f5985b6225d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:56
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:56
```

我们可以看到 6379 是一个主机，上面挂了两个从机，两个从机的地址、端口等信息都展现出来了。如果我们在 6380 上执行 INFO replication，显示信息如下：

```
127.0.0.1:6380> INFO replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:6
master_sync_in_progress:0
slave_repl_offset:630
slave_priority:100
slave_read_only:1
connected_slaves:0
master_replid:26ca818360d6510b717e471f3f0a6f5985b6225d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:630
```



```
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:630
```

我们可以看到 6380 是一个从机，从机的信息以及它的主机的信息都展示出来了。

7.此时，我们在主机中存储一条数据，在从机中就可以 `get` 到这条数据了。

## 主从复制注意点

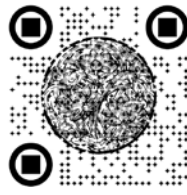
1. 如果主机已经运行了一段时间了，并且已经存储了一些数据了，此时从机连上来，那么从机将会将主机上所有的数据进行备份，而不是从连接的那个时间点开始备份。
2. 配置了主从复制之后，主机上可读可写，但是从机只能读取不能写入（可以通过修改 `redis.conf` 中 `slave-read-only` 的值让从机也可以执行写操作）。
3. 在整个主从结构运行过程中，如果主机不幸挂掉，重启之后，他依然是主机，主从复制操作也能够继续进行。

## 复制原理

每一个 `master` 都有一个 `replication ID`，这是一个较大的伪随机字符串，标记了一个给定的数据集。每个 `master` 也持有一个偏移量，`master` 将自己产生的复制流发送给 `slave` 时，发送多少字节的数据，自身的偏移量就会增加多少，目的是当有新的操作修改自己的数据集时，它可以以此更新 `slave` 的状态。复制偏移量即使在没有一个 `slave` 连接到 `master` 时，也会自增，所以基本上每一对给定的 `Replication ID, offset` 都会标识一个 `master` 数据集的确切版本。当 `slave` 连接到 `master` 时，它们使用 `PSYNC` 命令来发送它们记录的旧的 `master replication ID` 和它们至今为止处理的偏移量。通过这种方式，`master` 能够仅发送 `slave` 所需的增量部分。但是如果 `master` 的缓冲区中没有足够的命令积压缓冲记录，或者如果 `slave` 引用了不再知道的历史记录（`replication ID`），则会转而进行一个全量重同步：在这种情况下，`slave` 会得到一个完整的数据集副本，从头开始(参考 [redis 官网](#))。

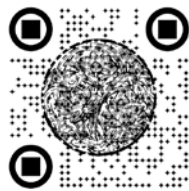
简单来说，就是以下几个步骤：

4. `slave` 启动成功连接到 `master` 后会发送一个 `sync` 命令。



5. **Master** 接到命令启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令。
6. 在后台进程执行完毕之后，**master** 将传送整个数据文件到 **slave** ,以完成一次完全同步。
7. 全量复制：而 **slave** 服务在接收到数据库文件数据后，将其存盘并加载到内存中。
8. 增量复制：**Master** 继续将新的所有收集到的修改命令依次传给 **slave** ,完成同步。
9. 但是只要是重新连接 **master** ,一次完全同步（全量复制)将被自动执行。

OK,redis 主从复制我们先介绍这么多，更多资料小伙伴们可以参考[官方文档](#)。小伙伴们在看官方文档时，有什么问题欢迎留言讨论。



## 11.Redis 主从复制(二)

2019-06-15 20:51:53

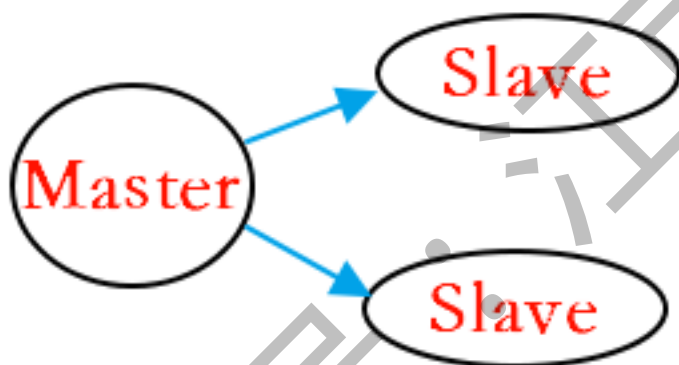
上篇文章和小伙伴们一起搭建了 redis 主从复制环境，但是还不完善，本文我想再和小伙伴们聊聊主从复制环境搭建的一些细节。

本文是 Redis 系列的第十一篇文章，了解前面的文章有助于更好的理解本文：

本文接上文，所用三个 redis 实例和上文一致，这里就不再赘述三个实例搭建方式。

### 一场接力赛

在上篇文章中，我们搭建的主从复制模式是下面这样的：



实际上，一主二仆的主从复制，我们可以搭建成下面这种结构：

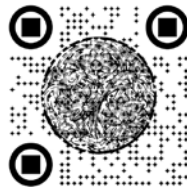


搭建方式很简单，在前文基础上，我们只需要修改 6381 的 master 即可，在 6381 实例上执行如下命令，让 6381 从 6380 实例上复制数据，如下：

```
127.0.0.1:6381> SLAVEOF 127.0.0.1 6380
OK
```

此时，我们再看 6379 的 slave，如下：



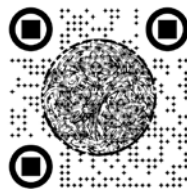


```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=0,lag=1
master_replid:4a38bbfa37586c29139b4ca1e04e8a9c88793651
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:0
```

只有一个 slave，就 6380，我们再看 6380 的信息，如下：

```
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:70
slave_priority:100
slave_read_only:1
connected_slaves:1
slave0:ip=127.0.0.1,port=6381,state=online,offset=70,lag=0
master_replid:4a38bbfa37586c29139b4ca1e04e8a9c88793651
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:70
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:70
```

6380 此时的角色是一个从机，它的主机是 6379，但是 6380 自己也有一个从机，那就是 6381。此时我们的主从结构如下图：



## 哨兵模式

结合上篇文章，我们一共介绍了两种主从模式了，但是这两种，不管是哪一种，都会存在这样一个问题，那就是当主机宕机时，就会发生群龙无首的情况，如果在主机宕机时，能够从从机中选出一个来充当主机，那么就不用我们每次去手动重启主机了，这就涉及到一个新的话题，那就是哨兵模式。

所谓的哨兵模式，其实并不复杂，我们还是在我们的基础上来搭建哨兵模式。假设现在我的 master 是 6379，两个从机分别是 6380 和 6381，两个从机都是从 6379 上复制数据。先按照上文的步骤，我们配置好一主二仆，然后在 redis 目录下打开 sentinel.conf 文件，做如下配置：

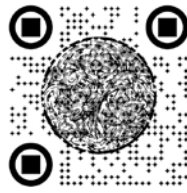
```
sentinel monitor mymaster 127.0.0.1 6379 1
```

其中 mymaster 是给要监控的主机取的名字，随意取，后面是主机地址，最后面的 2 表示有多少个 sentinel 认为主机挂掉了，就进行切换（我这里只有一个，因此设置为 1）。好了，配置完成后，输入如下命令启动哨兵：

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

然后启动我们的一主二仆架构，启动成功后，关闭 master，观察哨兵窗口输出的日志，如下：

```
5160:X 13 May 20:30:05.345 # +sdown master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.345 # +odown master mymaster 127.0.0.1 6379 #quorum 1/1
5160:X 13 May 20:30:05.345 # +new-epoch 1
5160:X 13 May 20:30:05.345 # +try-failover master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.350 # +vote-for-leader alab5ae4dc8252f368edc9e9c537d7ca700ba85e 1
5160:X 13 May 20:30:05.350 # +elected-leader master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.350 # +failover-state-select-slave master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.454 # +selected-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.454 * +failover-state-send-slaveof-noone slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.521 * +failover-state-wait-promotion slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.895 # +promoted-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.895 # +failover-state-reconf-slaves master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.906 * +slave-reconf-sent slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.924 * +slave-reconf-inprog slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.924 * +slave-reconf-done slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.978 # +failover-end master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.978 # +switch-master mymaster 127.0.0.1 6379 127.0.0.1 6380
5160:X 13 May 20:30:06.979 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6380
5160:X 13 May 20:30:06.979 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380
5160:X 13 May 20:30:37.021 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380
```



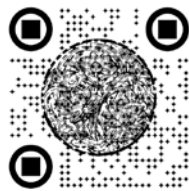
---

小伙伴们可以看到，6379 挂掉之后，redis 内部重新举行了选举，6380 重新上位。此时，如果 6379 重启，也不再是扛把子了，只能屈身做一个 slave 了。

## 注意问题

由于所有的写操作都是先在 Master 上操作，然后同步更新到 Slave 上，所以从 Master 同步到 Slave 机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave 机器数量的增加也会使这个问题更加严重。因此我们还需要集群来进一步提升 redis 性能，这个问题我们将在后面说到。

OK,redis 主从复制问题我们就介绍这么多，更多资料小伙伴们可以参考官方文档 <http://www.redis.net.cn/tutorial/3501.html>。小伙伴在看官方文档时，有什么问题欢迎留言讨论。



## 12.Redis 集群搭建

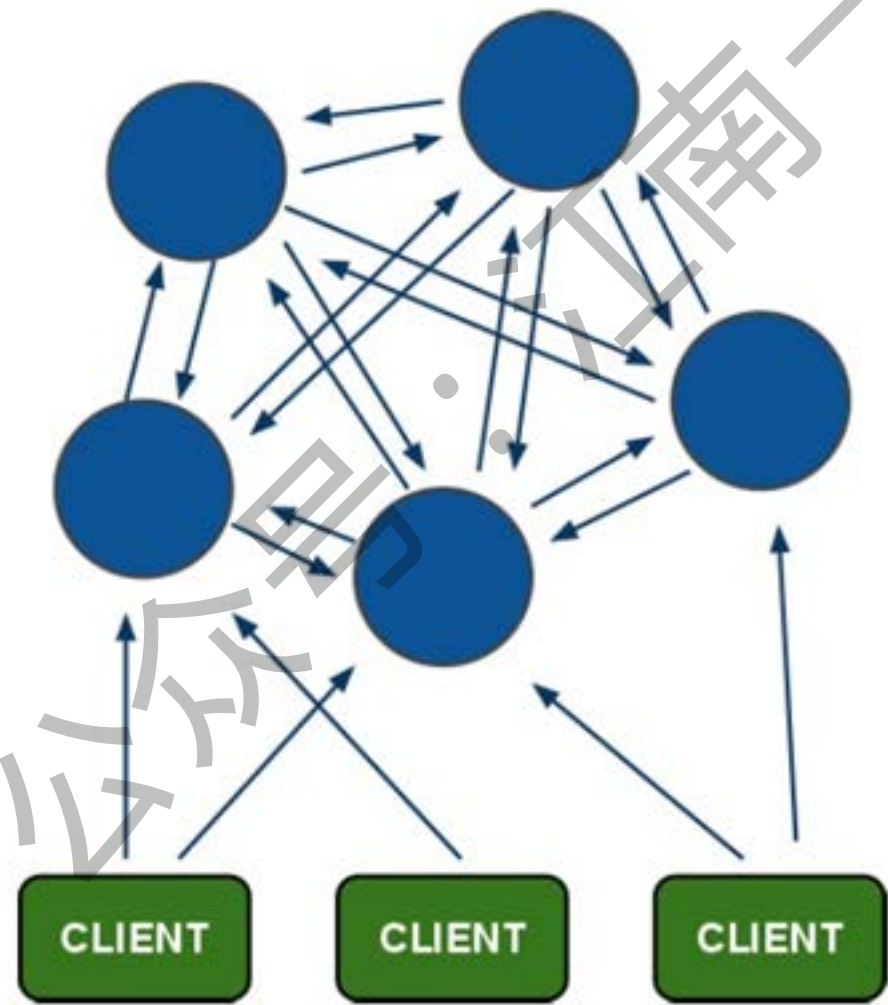
2019-06-15 20:52:02

主从的搭建差不多说完了，本文我们来看看集群如何搭建。

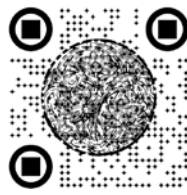
本文是 Redis 系列的第十二篇文章，了解前面的文章有助于更好的理解本文：

### 集群原理

Redis 集群架构如下图：



Redis 集群运行原理如下：



1. 所有的 Redis 节点彼此互联( PING-PONG 机制),内部使用二进制协议优化传输速度和带宽
2. 节点的 fail 是通过集群中超过半数的节点检测失效时才生效
3. 客户端与 Redis 节点直连,不需要中间 proxy 层, 客户端不需要连接集群所有节点, 连接集群中任何一个可用节点即可
4. Redis-cluster 把所有的物理节点映射到 [0-16383] slot 上, cluster (簇)负责维护 node<->slot<->value 。Redis 集群中内置了 16384 个哈希槽, 当需要在 Redis 集群中放置一个 key-value 时, Redis 先对 key 使用 crc16 算法算出一个结果, 然后把结果对 16384 求余数, 这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽, Redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

## 怎么样投票

投票过程是集群中所有 master 参与,如果半数以上 master 节点与 master 节点通信超过 `cluster-node-timeout` 设置的时间,认为当前 master 节点挂掉。

## 怎么样判定节点不可用

- 1.如果集群任意 master 挂掉,且当前 master 没有 slave, 集群进入 fail 状态,也可以理解成集群的 slot 映射 [0-16383] 不完整时进入 fail 状态。
- 2.如果集群超过半数以上 master 挂掉, 无论是否有 slave ,集群进入 fail 状态, 当集群不可用时,所有对集群的操作做都不可用, 收到 ((error) CLUSTERDOWN The cluster is down) 错误。

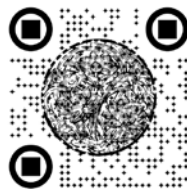
## ruby 环境

Redis 集群管理工具 `redis-trib.rb` 依赖 ruby 环境, 首先需要安装 ruby 环境:

安装 ruby:

```
yum install ruby
yum install rubygems
```

但是这种安装方式装好的 ruby 版本可能不适用, 如果安装失败, 可以参考这篇文章解决 [redis requires Ruby version >= 2.2.2](#)。



## 集群搭建

首先我们对集群做一个简单规划，假设我的集群中一共有三个节点，每个节点一个主机一个从机，这样我一共需要 6 个 Redis 实例。首先创建 redis-cluster 文件夹，在该文件夹下分别创建 7001、7002、7003、7004、7005、7006 文件夹，用来存放我的 Redis 配置文件，如下：

```
[root@localhost redis-cluster]# ls
7001 7002 7003 7004 7005 7006 redis-4.0.9
```

将 Redis 也在 redis-cluster 目录下安装一份，然后将 redis.conf 文件向 7001-7006 这 6 个文件夹中分别拷贝一份，拷贝完成后，分别修改如下参数：

```
port 7001
#bind 127.0.0.1
cluster-enabled yes
cluster-config-XX XXX7001.conf
protected no
daemonize yes
```

这是 7001 目录下的配置，其他的文件夹将 7001 改为对应的数字即可。修改完成后，进入到 redis 安装目录中，分别启动各个 redis，使用刚刚修改过的配置文件，如下：

```
[root@localhost redis-cluster]# cd redis-4.0.9/
[root@localhost redis-4.0.9]# redis-server ../7001/redis.conf
7688:C 01 Jun 11:15:24.737 # o000o000o000o Redis is starting o000o000o000o
7688:C 01 Jun 11:15:24.737 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7688, just started
7688:C 01 Jun 11:15:24.737 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7002/redis.conf
7693:C 01 Jun 11:15:30.039 # o000o000o000o Redis is starting o000o000o000o
7693:C 01 Jun 11:15:30.039 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7693, just started
7693:C 01 Jun 11:15:30.039 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7003/redis.conf
7698:C 01 Jun 11:15:35.890 # o000o000o000o Redis is starting o000o000o000o
7698:C 01 Jun 11:15:35.890 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7698, just started
7698:C 01 Jun 11:15:35.890 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7004/redis.conf
7703:C 01 Jun 11:15:39.731 # o000o000o000o Redis is starting o000o000o000o
7703:C 01 Jun 11:15:39.731 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7703, just started
7703:C 01 Jun 11:15:39.731 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7005/redis.conf
7708:C 01 Jun 11:15:44.856 # o000o000o000o Redis is starting o000o000o000o
7708:C 01 Jun 11:15:44.856 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7708, just started
7708:C 01 Jun 11:15:44.857 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7006/redis.conf
7713:C 01 Jun 11:15:52.596 # o000o000o000o Redis is starting o000o000o000o
7713:C 01 Jun 11:15:52.596 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7713, just started
7713:C 01 Jun 11:15:52.596 # Configuration loaded
```

启动成功后，我们可以查看 redis 进程，如下：





```
[root@localhost redis-4.0.9]# ps -aux|grep redis
root      4001    0.0  0.2 16264  5448 pts/1    S+   10:13   0:00 redis-cli -p 6380
root      4003    0.1  0.5 147300  9792 ?        Ssl  10:13   0:07 redis-server *:6381
root      4007    0.0  0.2 16264  5444 pts/2    S+   10:13   0:00 redis-cli -p 6381
root      4017    0.0  0.2 16264  5424 pts/0    S+   10:19   0:00 redis-cli
root      7689    0.1  0.5 151400  9680 ?        Ssl  11:15   0:00 redis-server *:7001 [cluster]
root      7694    0.1  0.5 147304  9680 ?        Ssl  11:15   0:00 redis-server *:7002 [cluster]
root      7699    0.1  0.5 147304  9680 ?        Ssl  11:15   0:00 redis-server *:7003 [cluster]
root      7704    0.2  0.5 147304  9680 ?        Ssl  11:15   0:00 redis-server *:7004 [cluster]
root      7709    0.2  0.5 147304  9676 ?        Ssl  11:15   0:00 redis-server *:7005 [cluster]
root      7714    0.4  0.5 147304  9680 ?        Ssl  11:15   0:00 redis-server *:7006 [cluster]
root      7719    0.0  0.0 112668   972 pts/3    R+   11:16   0:00 grep --color=auto redis
[root@localhost redis-4.0.9]#
```

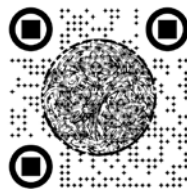
这个表示各个节点都启动成功了。接下来我们就可以进行集群的创建了，首先将 redis/src 目录下的 redis-trib.rb 文件拷贝到 redis-cluster 目录下，然后在 redis-cluster 目录下执行如下命令：

```
./redis-trib.rb create --replicas 1 192.168.248.128:7001 192.168.248.128:7002 192.168.248.128:7003 192.168.248.128:7004 192.168.248.128:7005 192.168.248.128:7006
```

注意，replicas 后面的 1 表示每个主机都带有 1 个从机，执行过程如下：

```
[root@localhost redis-cluster]# ./redis-trib.rb create --replicas 1 192.168.248.128:7001 192.168.248.128:7002 192.168.248.128:7003 192.168.248.128:7004 192.168.248.128:7005 192.168.248.128:7006
>>> Creating cluster
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
192.168.248.128:7001
192.168.248.128:7002
192.168.248.128:7003
Adding replica 192.168.248.128:7005 to 192.168.248.128:7001
Adding replica 192.168.248.128:7006 to 192.168.248.128:7002
Adding replica 192.168.248.128:7004 to 192.168.248.128:7003
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: 96aff797e48a9c832354a0f26f71bcd9c803184 192.168.248.128:7001
slots:0-5460 (5461 slots) master
M: 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 192.168.248.128:7002
slots:5461-10922 (5462 slots) master
M: 5e68d470aldaf742207165b7d5e042f8c81f3081 192.168.248.128:7003
slots:10923-16383 (5461 slots) master
S: 269ef58fd4104d535b364ad781f3c4fa88ff8f49 192.168.248.128:7004
replicas 96aff797e48a9c832354a0f26f71bcd9c803184
S: 40502040f08494c8dcl9dd28b59a5a387edc8fcl 192.168.248.128:7005
replicas 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468
S: 2d08b81328910c5a525253ceebal831cd8d9bf74 192.168.248.128:7006
replicas 5e68d470aldaf742207165b7d5e042f8c81f3081
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 192.168.248.128:7001)
M: 96aff797e48a9c832354a0f26f71bcd9c803184 192.168.248.128:7001
slots:0-5460 (5461 slots) master
1 additional replica(s)
M: 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 192.168.248.128:7002
slots:5461-10922 (5462 slots) master
1 additional replica(s)
S: 269ef58fd4104d535b364ad781f3c4fa88ff8f49 192.168.248.128:7004
slots: (0 slots) slave
replicas 96aff797e48a9c832354a0f26f71bcd9c803184
S: 2d08b81328910c5a525253ceebal831cd8d9bf74 192.168.248.128:7006
slots: (0 slots) slave
replicas 5e68d470aldaf742207165b7d5e042f8c81f3081
```

注意创建过程的日志，每个 redis 都获得了一个编号，同时日志也说明了哪些实例做主机，哪些实例做从机，每个从机的主机是谁，每个主机所分配到的 hash 槽范围等等。



## 查询集群信息

集群创建成功后，我们可以登录到 Redis 控制台查看集群信息，注意登录时要添加 -c 参数，表示以集群方式连接，如下：

```
[root@localhost redis-4.0.9]# redis-cli -p 7001 -c
127.0.0.1:7001> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:1
cluster_stats_messages_ping_sent:265
cluster_stats_messages_pong_sent:256
cluster_stats_messages_sent:521
cluster_stats_messages_ping_received:251
cluster_stats_messages_pong_received:265
cluster_stats_messages_meet_received:5
cluster_stats_messages_received:521
127.0.0.1:7001>
```

```
127.0.0.1:7001> cluster nodes
2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 192.168.248.128:7002@17002 master - 0 1527823610000 2 connected 5461-10922
269ef58fd4104d535b364ad781f3c4fa88ff8f49 192.168.248.128:7004@17004 slave 96aff797e48a9c832354a0f26f71bcd9c803184 0 1527823611000 4 connected
2d08b81328910c5a525253ceeb1831cd8d9bf74 192.168.248.128:7006@17006 slave 5e68d470aldaf742207165b7d5e042f8c81f3081 0 1527823612653 6 connected
40502040f08494c8dc19dd28b59a5a387edc8fcl 192.168.248.128:7005@17005 slave 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 0 1527823611644 5 connected
5e68d470aldaf742207165b7d5e042f8c81f3081 192.168.248.128:7003@17003 master - 0 1527823610000 3 connected 10923-16383
96aff797e48a9c832354a0f26f71bcd9c803184 192.168.248.128:7001@17001 myself,master - 0 1527823611000 1 connected 0-5460
127.0.0.1:7001>
```

## 添加主节点

首先我们准备一个端口为 7007 的主节点并启动，准备方式和前面步骤一样，启动成功后，通过如下命令添加主节点：

```
./redis-trib.rb add-node 127.0.0.1:7007 127.0.0.1:7001
```

主节点添加之后，我们可以通过 cluster nodes 命令查看主节点是否添加成功，此时我们发现新添加的节点没有分配到 slot，如下：

```
127.0.0.1:7007> cluster nodes
66dae8e6a340837acefbd495c7eff5a0b85a7a5 192.168.248.128:7006@17006 slave 91b6e0a668b21f3cad35c2589fd959839b8ce6db 0 15278368
ted
c5e409f65d0bd292088f0500283ebb5db7fa08e2 192.168.248.128:7004@17004 slave 80fd8e9e5f981e63d9852dc29c273e703d653afa 0 15278368
ted
91b6e0a668b21f3cad35c2589fd959839b8ce6db 192.168.248.128:7003@17003 master - 0 1527836867038 3 connected 11256-16383
d9dbe9c4fd4b22f794694c85f90d6f3d066c85ed 192.168.248.128:7005@17005 slave flbea8b2cb0fde9c3b1806f42a410526c34cb2a2 0 15278368
ted
90fd8e9e5f981e63d9852dc29c273e703d653afa 127.0.0.1:7001@17001 master - 0 1527836862000 9 connected 167-5794 10923-11255
flbea8b2cb0fde9c3b1806f42a410526c34cb2a2 192.168.248.128:7002@17002 master - 0 1527836864000 8 connected 0-166 5795-10922
edlcc96d20909a503b40f6d69f8a23ac0d6ba845 127.0.0.1:7007@17007 myself,master - 0 1527836865000 0 connected
127.0.0.1:7007>
```





没有分配到 slot 将不能存储数据，此时我们需要手动分配 slot，分配命令如下：

```
./redis-trib.rb reshard 127.0.0.1:7001
```

后面的地址为任意一个节点地址，在分配的过程中，我们一共要输入如下几个参数：

1.一共要划分多少个 hash 槽出来？就是我们总共要给新添加的节点分多少 hash 槽，这个参数依实际情况而定，如下：

```
How many slots do you want to move (from 1 to 16384)?
```

2.这些划分出来的槽要给谁，这里输入 7007 节点的编号，如下：

```
What is the receiving node ID? ed1cc96d20909a503b40f6d69f8a23ac0d6ba845
```

p326

3.要让谁出血？因为 hash 槽目前已经全部分配完毕，要重新从已经分好的节点中拿出来一部分给 7007，必然要让另外三个节点把吃进去的吐出来，这里我们可以输入多个节点的编号，每次输完一个点击回车，输完所有的输入 done 表示输入完成，这样就让这些几个节点让出部分 slot，如果要是让所有具有 slot 的节点都参与到此 slot 重新分配的活动中，那么这里直接输入 all 即可，如下：

```
What is the receiving node ID? ed1cc96d20909a503b40f6d69f8a23ac0d6ba845
Please enter all the source node IDs.
Type 'all' to use all the nodes as source nodes for the hash slots.
Type 'done' once you entered all the source nodes IDs.
Source node #1:all
```

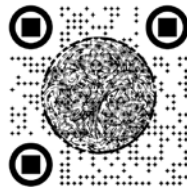
OK，主要就是这几个参数，输完之后进入到 slot 重新分配环节，分配完成后，通过 cluster nodes 命令，我们可以发现 7007 已经具有 slot 了，如下：

```
127.0.0.1:7001> cluster nodes
80fd8e9e5f981e63d9852dc29c273e703d653afa 127.0.0.1:7001@17001 myself,master - 0 1527838117000 9 connected 531-5794 10923-11255
d8db9e9cf4db22f794694c85f90def3d066c85ed 192.168.248.128:7005@17005 slave flbea8b2cb0fde9c3b1806f42a410526c34cb2a2 0 1527838117767 8 conn
66dae8e64a340837acefbdb485c7eff5a0b35a7a5 192.168.248.128:7006@17006 slave 91b6e0ae66b21f3cad35c2588fd959839b8ce6db 0 1527838117000 6 conn
ed1cc96d20909a503b40f6d69f8a23ac0d6ba845 127.0.0.1:7007@17007 master - 0 1527838115735 10 connected 0-530 5795-5950 11256-11567
91b6e0ae66b21f3cad35c2588fd959839b8ce6db 192.168.248.128:7003@17003 master - 0 1527838118783 3 connected 11568-16383
c5e409f65d0bd292088f0500283ebb5db7fa08e2 192.168.248.128:7004@17004 slave 80fd8e9e5f981e63d9852dc29c273e703d653afa 0 1527838115000 9 conn
ted
flbea8b2cb0fde9c3b1806f42a410526c34cb2a2 192.168.248.128:7002@17002 master - 0 1527838117000 8 connected 5951-10922
127.0.0.1:7001>
```

OK，刚刚我们是添加主节点，我们也可以添加从节点，比如我要把 7008 作为 7007 的从节点，添加方式如下：

```
./redis-trib.rb add-node --slave --master-id 79bbb30bba66b4997b9360dd09849c67d2d02bb9 192.168.31.135:7008 192.168.31.135:7007
```

其中 79bbb30bba66b4997b9360dd09849c67d2d02bb9 是 7007 的编号。



---

## 删除节点

删除节点也比较简单，如下：

```
./redis-trib.rb del-node 127.0.0.1:7005 4b45eb75c8b428fbd77ab979b85080146a9bc017
```

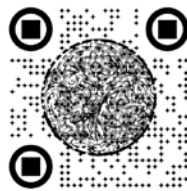
注意 4b45eb75c8b428fbd77ab979b85080146a9bc017 是要删除节点的编号。

再注意：删除已经占有 hash 槽的结点会失败，报错如下：

```
[ERR] Node 127.0.0.1:7005 is not empty! Reshard data away and try again.
```

需要将该结点占用的 hash 槽分配出去（分配方式与上文一致，不赘述）。

好了，redis 集群搭建我们先说这么多，有问题欢迎留言讨论。



## 13.Jedis 使用

2019-06-15 20:52:08






Redis 的知识我们已经介绍的差不多了，本文我们来看看如何使用 Java 操作 redis。

本文是 Redis 系列的第十三篇文章，了解前面的文章有助于更好的理解本文：

### 有哪些解决方案

查看 redis 官网，我们发现用 Java 操作 redis，我们有多种解决方案，如下图：

#### Java

Redisson	😊 ★ 🏠 🐍	distributed and scalable Java data structures on top of Redis server	
aredis	🐍	Asynchronous, pipelined client based on the Java 7 NIO Channel API	
JDBC-Redis	🏠 🐍		
Jedis	😊 ★ 🐍		
JRedis	😊 🏠 🐍		
lettuce	😊 🐍	Thread-safe client supporting async usage and key/value codecs	
mod-redis	🐍	Official asynchronous redis.io bus module for Vert.x	
redis-protocol	😊 🐍	Up to 2.6 compatible high-performance Java, Java w/Netty & Scala (finagle) client	
RedisClient	😊 🐍	redis client GUI tool	
RJC	🐍		

这里的解决方案有多种，我们采用 Jedis，其他的框架也都大同小异，我这里权当抛砖引玉，小伙伴也可以研究研究其他的方案，欢迎投稿。



## 配置

客户端要能够成功连接上 redis 服务器，需要检查如下三个配置：

1. 远程 Linux 防火墙已经关闭，以我这里的 CentOS7 为例，关闭防火墙命令 `systemctl stop firewalld.service`，同时还可以再补一刀 `systemctl disable firewalld.service` 表示禁止防火墙开机启动。

2. 关闭 redis 保护模式，在 `redis.conf` 文件中，修改 `protected` 为 `no`，如下：

```
protected-mode no
```

3. 注释掉 redis 的 ip 地址绑定，还是在 `redis.conf` 中，将 `bind:127.0.0.1` 注释掉，如下：

```
# bind:127.0.0.1
```

确认了这三步之后，就可以远程连接 redis 了。

## Java 端配置

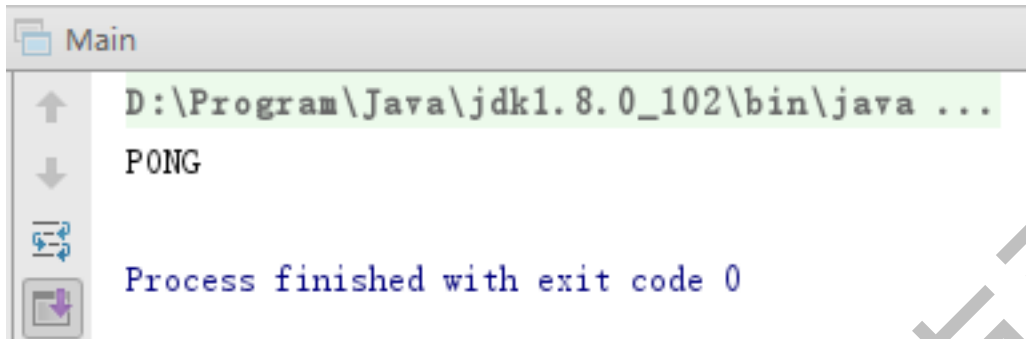
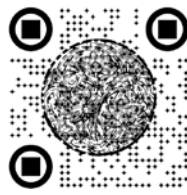
上面的配置完成后，我们可以创建一个普通的 JavaSE 工程来测试下了，Java 工程创建成功后，添加 Jedis 依赖，如下：

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>
```

然后我们可以通过如下一个简单的程序测试一下连接是否成功：

```
public static void main(String[] args) {
    Jedis jedis = new Jedis("192.168.248.128", 6379);
    String ping = jedis.ping();
    System.out.println(ping);
}
```

运行之后，看到如下结果表示连接成功了：



连接成功之后，剩下的事情就比较简单了，Jedis 类中方法名称和 redis 中的命令基本是一致的，看到方法名小伙伴就知道是干什么的，因此这些我这里不再重复叙述。

频繁的创建和销毁连接会影响性能，我们可以采用连接池来部分的解决这个问题：

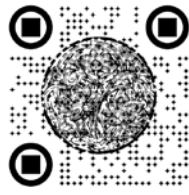
```
public static void main(String[] args) {
    GenericObjectPoolConfig config = new GenericObjectPoolConfig();
    config.setMaxTotal(100);
    config.setMaxIdle(20);
    JedisPool jedisPool = new JedisPool(config, "192.168.248.128", 6379);
    Jedis jedis = jedisPool.getResource();
    System.out.println(jedis.ping());
}
```

这样就不会频繁创建和销毁连接了，在 JavaSE 环境中可以把连接池配置成一个单例模式，如果用了 Spring 容器的话，可以把连接池交给 Spring 容器管理。

上面这种连接都是连接单节点的 Redis，如果是一个 Redis 集群，要怎么连接呢？很简单，如下：

```
Set<HostAndPort> clusterNodes = new HashSet<HostAndPort>();
clusterNodes.add(new HostAndPort("192.168.248.128", 7001));
clusterNodes.add(new HostAndPort("192.168.248.128", 7002));
clusterNodes.add(new HostAndPort("192.168.248.128", 7003));
clusterNodes.add(new HostAndPort("192.168.248.128", 7004));
clusterNodes.add(new HostAndPort("192.168.248.128", 7005));
clusterNodes.add(new HostAndPort("192.168.248.128", 7006));
JedisCluster jc = new JedisCluster(clusterNodes);
jc.set("address", "深圳");
String address = jc.get("address");
System.out.println(address);
```

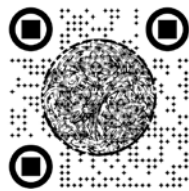
JedisCluster 中的方法与 Redis 命令也是基本一致，我就不再重复介绍了。



---

好了，jedis 就说这么多，有问题欢迎留言讨论。

公众号 · 江南一点雨



## 14.Spring Data Redis 使用

2019-06-15 20:52:21

上文我们介绍了 Redis，在开发环境中，我们还有另外一个解决方案，那就是 Spring Data Redis。本文我们就来看看这个东西。

本文是 Redis 系列的第十四篇文章，了解前面的文章有助于更好的理解本文：

### Spring Data Redis 介绍

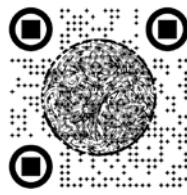
Spring Data Redis 是 Spring 官方推出，可以算是 Spring 框架集成 Redis 操作的一个子框架，封装了 Redis 的很多命令，可以很方便的使用 Spring 操作 Redis 数据库，Spring 对很多工具都提供了类似的集成，如 Spring Data MongoDB、Spring Data JPA 等，Spring Data Redis 只是其中一种。

### 环境搭建

要使用 SDR，首先需要搭建 Spring+SpringMVC 环境，由于这个不是本文的重点，因此这一步我直接略过，Spring+SpringMVC 环境搭建成功后，接下来我们要整合 SDR，首先需要添加如下依赖：

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-redis</artifactId>
  <version>RELEASE</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-pool2</artifactId>
  <version>RELEASE</version>
</dependency>
```

然后创建在 resources 目录下创建 redis.properties 文件作为 redis 的配置文件，如下：



```
redis.host=192.168.248.128
redis.port=6379
redis.maxIdle=300
redis.maxTotal=600
redis.maxWait=1000
redis.testOnBorrow=true
```

在 spring 的配置文件中，添加如下 bean：

```
<!--引入 redis.properties 文件-->
<context:property-placeholder location="classpath:redis.properties"/>
<!--配置连接池信息-->
<bean class="redis.clients.jedis.JedisPoolConfig" id="poolConfig">
    <property name="maxIdle" value="${redis.maxIdle}"/>
    <property name="maxTotal" value="${redis.maxTotal}"/>
    <property name="maxWaitMillis" value="${redis.maxWait}"/>
    <property name="testOnBorrow" value="${redis.testOnBorrow}"/>
</bean>
<!--配置基本连接信息-->
<bean class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" id="connectionFactory">
    <property name="hostName" value="${redis.host}"/>
    <property name="port" value="${redis.port}"/>
    <property name="poolConfig" ref="poolConfig"/>
</bean>
<!--配置 RedisTemplate-->
<bean class="org.springframework.data.redis.core.RedisTemplate" id="redisTemplate">
    <property name="connectionFactory" ref="connectionFactory"/>
    <!--key 和 value 要进行序列化，否则存储对象时会出错-->
    <property name="keySerializer">
        <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
    </property>
    <property name="valueSerializer">
        <bean class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer"/>
    </property>
</bean>
```

好了，在 Spring 中配置了 redisTemplate 之后，接下来我们就可以在 Dao 层注入 redisTemplate 进而使用了。

接下来我们首先创建实体类 User，注意 User 一定要可序列化：

```
public class User implements Serializable{
    private String username;
```





```
    private String password;
    private String id;
    //get/set 省略
}
```

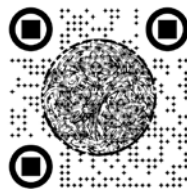
然后在 Dao 层实现数据的添加和获取，如下：

```
@Repository
public class HelloDao {
    @Autowired
    RedisTemplate redisTemplate;
    public void set(String key, String value) {
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set(key, value);
    }
    public String get(String key) {
        ValueOperations ops = redisTemplate.opsForValue();
        return ops.get(key).toString();
    }
    public void setUser(User user) {
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set(user.getId(), user);
    }
    public User getUser(String id) {
        ValueOperations<String, User> ops = redisTemplate.opsForValue
();
        User user = ops.get(id);
        System.out.println(user);
        return user;
    }
}
```

SDR 官方文档中对 RedisTemplate 的介绍，通过 RedisTemplate 可以调用 ValueOperations 和 ListOperations 等等方法，分别是对 Redis 命令的高级封装。但是 ValueOperations 等等这些命令最终是要转化成为 RedisCallback 来执行的。也就是说通过使用 RedisCallback 可以实现更强的功能。

最后，给大家展示下我的 Service 和 Controller，如下：

```
@Service
public class HelloService {
    @Autowired
    HelloDao helloDao;
    public void set(String key, String value) {
        helloDao.set(key,value);
    }
}
```



```
public String get(String key) {
    return helloDao.get(key);
}

public void setUser(User user) {
    helloDao.setUser(user);
}

public String getUser(String id) {
    String s = helloDao.getUser(id).toString();
    return s;
}
}

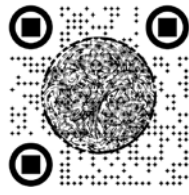
Controller:
@Controller
public class HelloController {
    @Autowired
    HelloService helloService;

    @RequestMapping("/set")
    @ResponseBody
    public void set(String key, String value) {
        helloService.set(key, value);
    }

    @RequestMapping("/get")
    @ResponseBody
    public String get(String key) {
        return helloService.get(key);
    }

    @RequestMapping("/setuser")
    @ResponseBody
    public void setUser() {
        User user = new User();
        user.setId("1");
        user.setUsername("深圳");
        user.setPassword("sang");
        helloService.setUser(user);
    }

    @RequestMapping(value = "/getuser", produces = "text/html;charset=UTF-8")
    @ResponseBody
    public String getUser() {
        return helloService.getUser("1");
    }
}
```



---

```
}  
}
```

测试过程就不再展示了，小伙伴们可以用 POSTMAN 等工具自行测试。

好了，Spring Data Redis 我们就说到这里，有问题欢迎留言讨论。

公众号 · 江南一点雨