

Redis-学习笔记

1.NoSql

1 单机MySQL的美好年代

2 Memcached(缓存)+MySQL+垂直拆分

3 Mysql主从读写分离

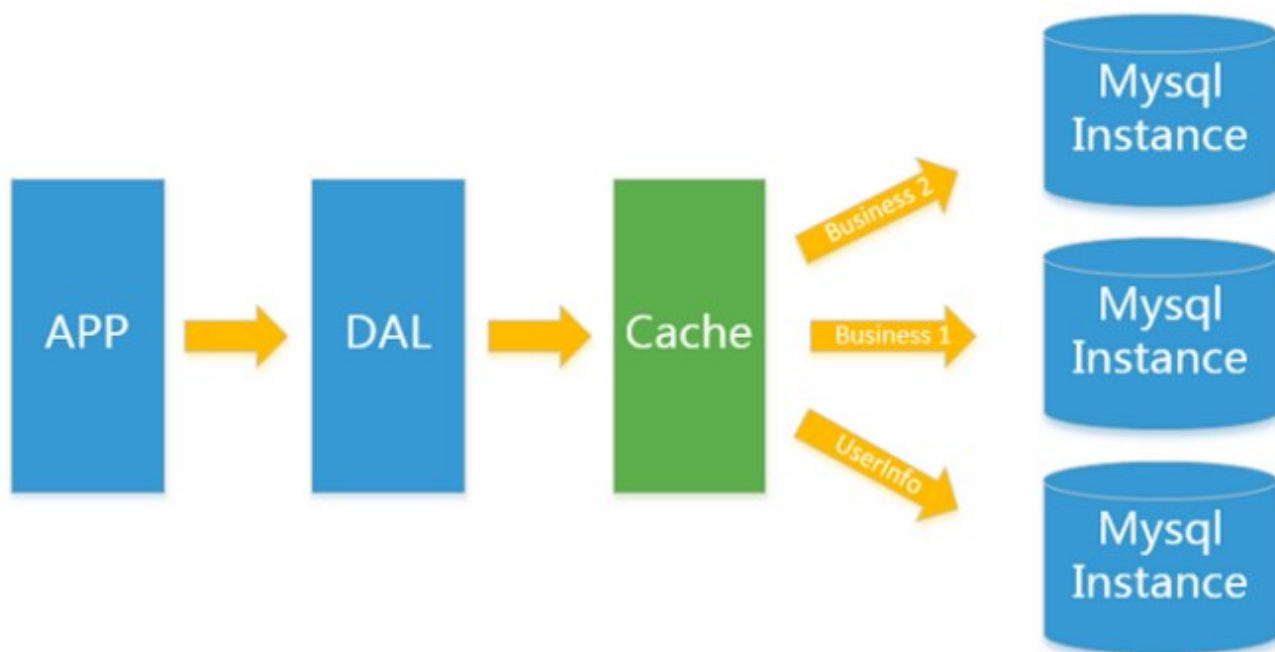
4 分表分库+水平拆分+mysql集群

5 MySQL的扩展性瓶颈

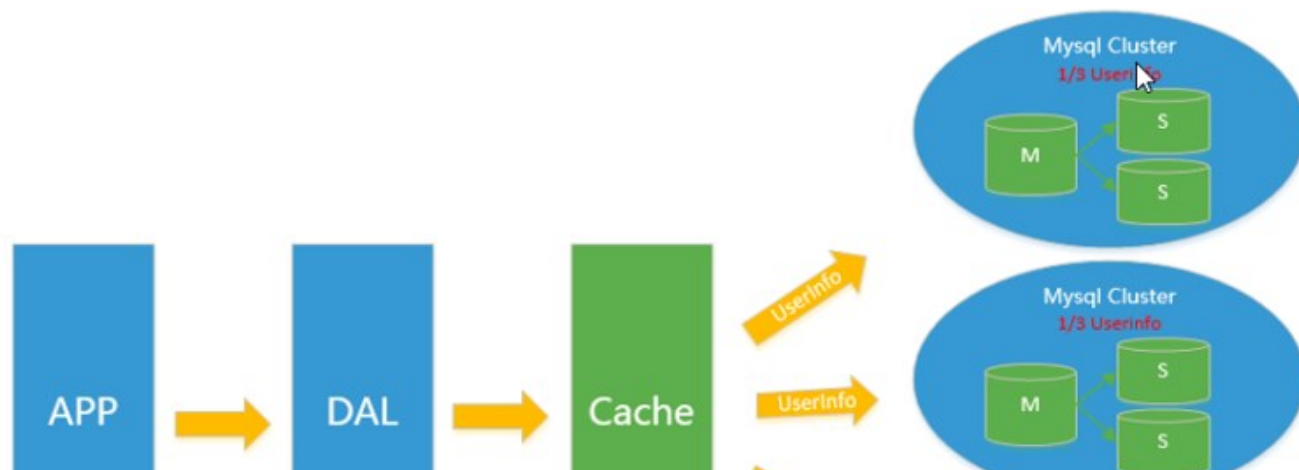
6 今天是什么样子??

7 为什么用NoSQL

Memcached (缓存) +Mysql+垂直拆分

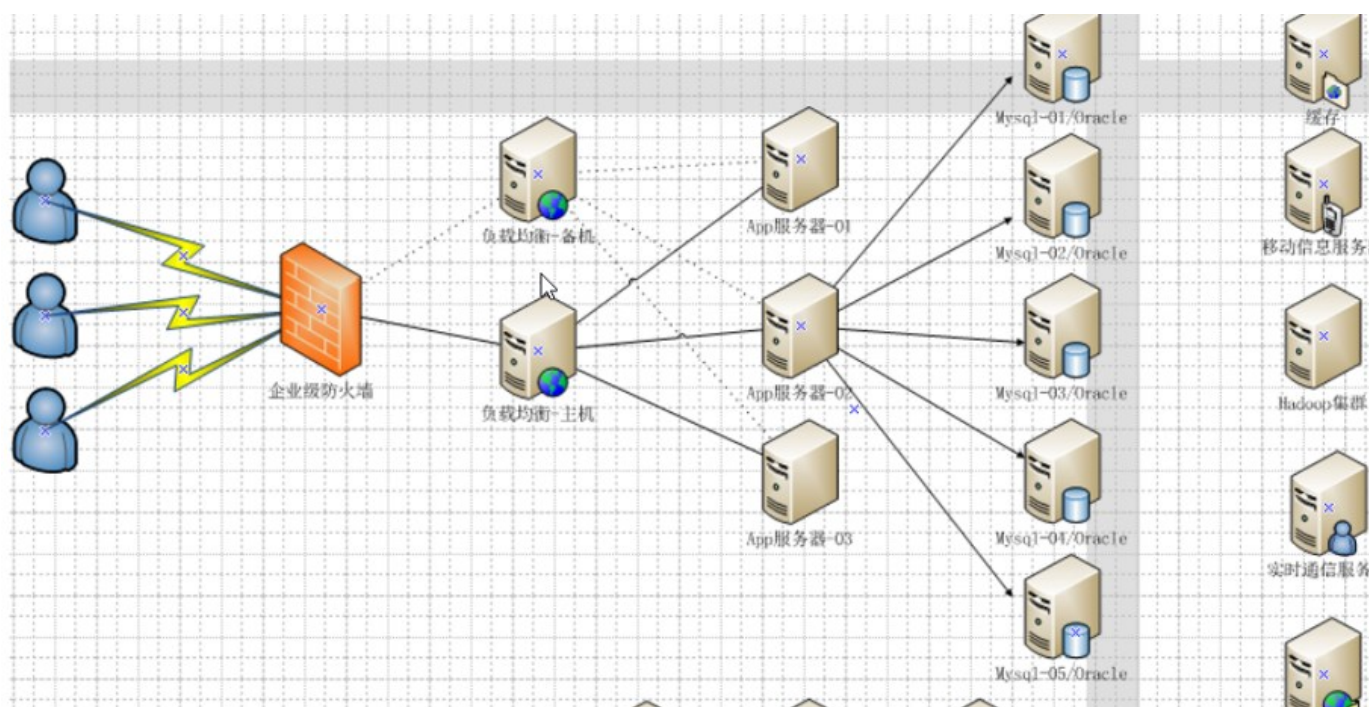


分表分库+水平拆分+mysql集群





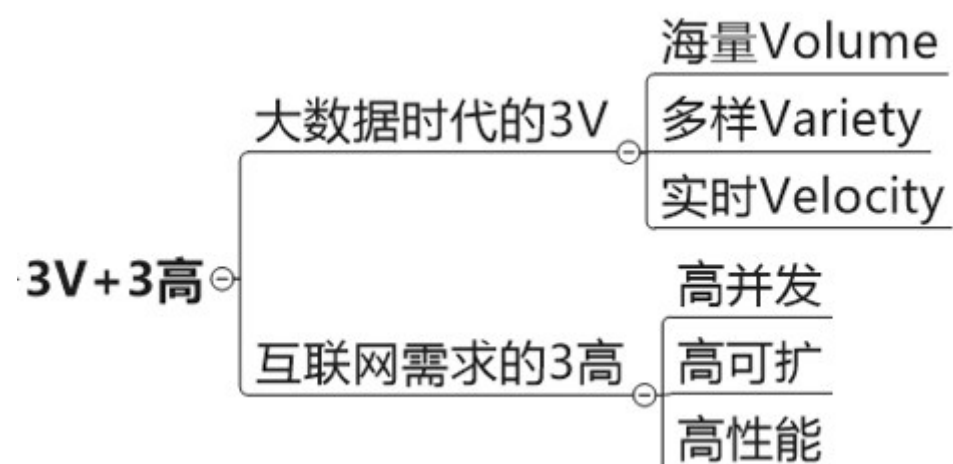
现如今



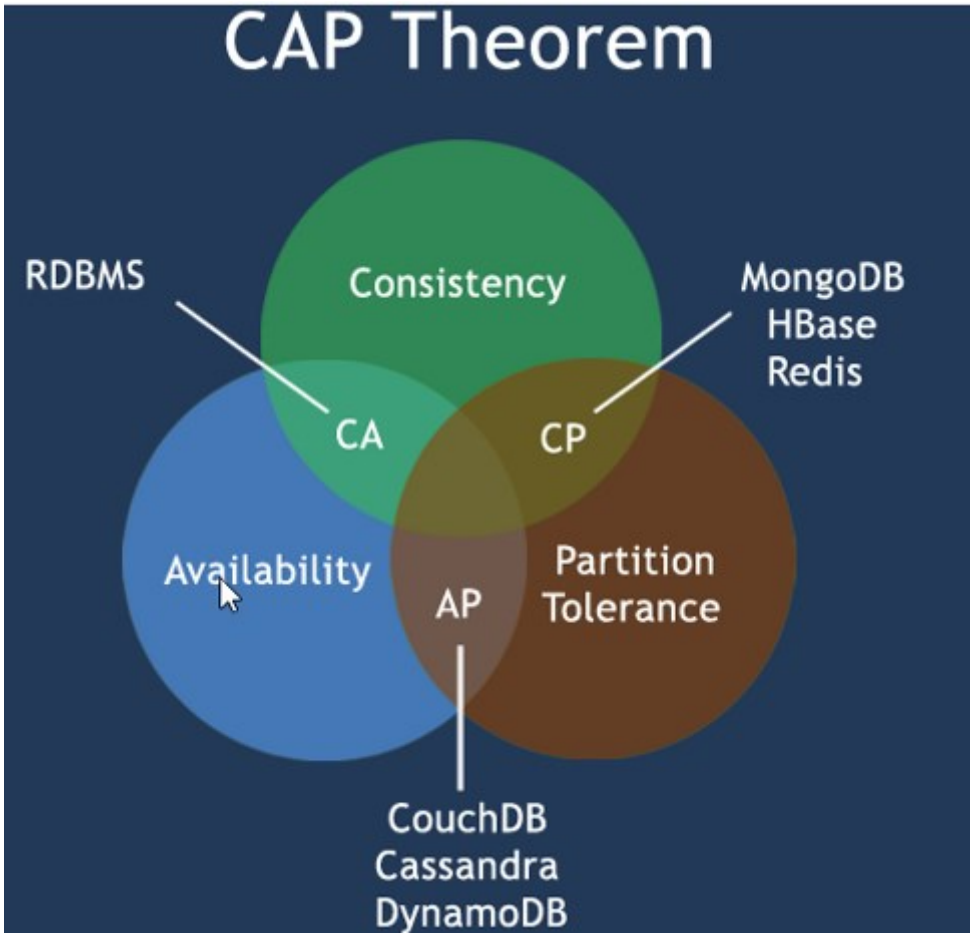
Redis

KV, cache, persistence

互联网时代



分布式数据库 原理CAP+BASE



A: 高可用性 C: 强一致性 P: 分布式容忍性

CA 传统Oracle数据库

AP 大多数网站架构的选择

CP Redis、Mongodb

分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值（key-value） ^[3]	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB	内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。 ^[3]	Key 指向 Value 的键值对，通常用hash table来实现 ^[3]	查找速度快	数据无结构化，通常只被当作字符串或者二进制数据 ^[3]
列存储数据库 ^[3]	Cassandra, HBase, Riak	分布式的文件系统	以列簇式存储，将同一列数据存在一起	查找速度快，可扩展性强，更容易进行分布式扩展	功能相对局限
文档型数据库 ^[3]	CouchDB, MongoDB	Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了	Key-Value 对应的键值对，Value为结构化数	数据结构要求不严格，表结构可变，不需要像关系型数	查询性能不高，而且缺乏统一的查询语法。

		解Value的内容)	据	据库一样需要 预先定义表结 构	
图形 (Graph)数	Neo4J, InfoGrid, Infinite Graph	社交网络, 推荐系 统等。专注于构建	图结构	利用图结构相 关算法。比如	很多时候需要对 整个图做计算才

下载Redis

redis下载地址

<https://github.com/dmajkic/redis/downloads>

<http://www.redis.cn/>

tar -zxvf redis.tar.gz

安装gcc

能上网: `yum install gcc-c++`

不能上网

7. 在终端中输入 `cd /media/CentOS_5.2_Final/CentOS` 回车

```

root@atguigu:/media/CentOS_5.5_Final/CentOS
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
[root@atguigu ~]# cd /media/CentOS_5.5_Final/CentOS/
[root@atguigu CentOS]#

```

8. 分别执行如下命令

```

rpm -ivh cpp-4.1.2-48.el5.i386.rpm 回车
rpm -ivh kernel-headers-2.6.18-194.el5.i386.rpm 回车
rpm -ivh glibc-headers-2.5-49.i386.rpm 回车
rpm -ivh glibc-devel-2.5-24.i386.rpm 回车
rpm -ivh libgomp-4.4.0-6.el5.i386.rpm 回车
rpm -ivh gcc-4.1.2-48.el5.i386.rpm 回车

```

二次make

Jemalloc/jemalloc.h: 没有那个文件或目录。运行make distclean之后再make

```
Hint: It's a good idea to run 'make test' :)
make[1]: Leaving directory `/usr/zzyy_soft/redis-3.0.4/src'
[root@cloud redis-3.0.4]# make test
cd src && make test
make[1]: Entering directory `/usr/zzyy_soft/redis-3.0.4/src'
You need tcl 8.5 or newer in order to run the Redis test
make[1]: *** [test] 错误 1
make[1]: Leaving directory `/usr/zzyy_soft/redis-3.0.4/src'
make: *** [test] 错误 2
[root@cloud redis-3.0.4]# pwd
/usr/zzyy_soft/redis-3.0.4
[root@cloud redis-3.0.4]# cd src
[root@cloud src]# pwd
/usr/zzyy_soft/redis-3.0.4/src
[root@cloud src]# ls -ll
总计 22556
```

下载TCL的网址:

<http://www.linuxfromscratch.org/blfs/view/cvs/general/tcl.html>

<http://www.linuxfromscratch.org/blfs/view/cvs/general/tcl.html>

```
wget http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz
```

<http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz>

安装TCL

1.安装tcl

解压tcl包: tar zxvf tcl8.4.11-src.tar.gz

进入: tcl8.4.11/unix

执行1: ./configure

执行2: make

执行3: make install

拷贝: cp tclUnixPort.h ../generic/

2.安装expect

解压expect包: tar zxvf tar zxvf expect-5.43.0.tar.gz

进入：cd expect-5.43/

执行1：./configure --with-tcl=/usr/local/lib/ --with-tclinclude=/home/nagios/tcl8.4.11/generic/ --with-x=no

执行2：make

执行3：make install

```
58915 05-30 18:25
]# vim redis.conf
```

修改配置

是否以后台进程运行

```
# 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
```

```
-rwxr-xr-x 1 root root 5276892 04-30 12:07 redis-server
```

```
[root@atguigu bin]# redis-server /myredis/redis.conf
```

```
[root@atguigu bin]# redis-cli -p 6379
```

```
127.0.0.1:6379> ping
```

```
PONG
```

```
127.0.0.1:6379> set k1 hello
```

```
OK
```

```
127.0.0.1:6379> get k1
```

```
"hello"
```

```
127.0.0.1:6379> SHUTDOWN
```

```
not connected> exit
```

启动

端口

退出

redis有16个数据库

```
# Set the number of databases. The default database is DB 0, you can select
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16
```

```
##### SNAPSHOTTING #####
```

基本命令

select N 查看第N个数据库

keys *查看该数据库下的所有key值

flushdb删除该数据库下所有key

flushall删除所有数据库下的所有key

dbsize查看当前数据库的key的数量

redis的五大数据类型:

命令查询网站<http://redisdoc.com/>

Redis 命令参考 »

[next](#) | [index](#)

Redis 命令参考

本文档是 [Redis Command Reference](#) 和 [Redis Documentation](#) 的中文翻译版，阅读这个文档可以帮助你了解 Redis 命令的具体使用方法，并学会如何使用 Redis 的事务、持久化、复制、Sentinel、集群等功能。

命令目录(使用 CTRL + F 快速查找)：⌕

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

key

exists key 判断某个key是否存在

```
(integer) 0
127.0.0.1:6379[2]> keys *
(empty list or set)
127.0.0.1:6379[2]> set k1 v1
OK
127.0.0.1:6379[2]> exists k1
(integer) 1
127.0.0.1:6379[2]> exists k2
```

```
(integer) 0
```

测试连通性

```
public class Demo01 {  
    public static void main(String[] args) {  
        //连接本地的 Redis 服务  
        Jedis jedis = new Jedis("127.0.0.1",6379);  
        //查看服务是否运行，打出pong表示OK  
        System.out.println("connection is OK=====>: "+jedis.ping());  
    }  
}
```

move key db 从当前库移动到db库

expire key time 为给定 key 设置生存时间，当 key 过期时(生存时间为 0)，它会被自动删除。---只能用于现存在的key中；

ttl key 查看还有多少秒过期，-1表示永不过期，-2表示已过期；

```
127.0.0.1:6379[1]> keys *  
1) "k1"  
127.0.0.1:6379[1]> expire k1 10  
(integer) 1  
127.0.0.1:6379[1]> ttl k1  
(integer) 4  
127.0.0.1:6379[1]> ttl k1  
(integer) 1  
127.0.0.1:6379[1]> ttl k1  
(integer) -2  
127.0.0.1:6379[1]> get k1  
(nil)
```

type key 查看你的key是什么类型

```
127.0.0.1:6379> type k1  
string
```

1、) String (字符串)

String是redis最气本的类型，一个key对应一个value；

String类型是二进制安全的，意思是可以包含任何数据，比如jpg图片或者序列化的对象；

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

incr/decr/incrby/decrby 一定要是数字才能进行加减

```
OK  
127.0.0.1:6379> get k2
```



```

"3"
127.0.0.1:6379> incr k2
(integer) 4
127.0.0.1:6379> get k2
"4"
127.0.0.1:6379> decr k2
(integer) 3
127.0.0.1:6379> get k2
"3"
127.0.0.1:6379> incrby k2
(error) ERR wrong number of arguments for 'incrby' command
127.0.0.1:6379> incrby k2 2
(integer) 5
127.0.0.1:6379> get k2
"5"

```

getrange/setrange 用 value 参数覆写(overwrite)给定 key 所储存的字符串值，从偏移量 offset 开始。

```

127.0.0.1:6379> get k1
"v1"
127.0.0.1:6379> setrange k1 0 xwqer
(integer) 5
127.0.0.1:6379> get k1
"xwqer"
127.0.0.1:6379> setrange k1 0 v1
(integer) 5
127.0.0.1:6379> get k1
"v1qer"
127.0.0.1:6379> getrange k1 0 2
"v1q"

```

setex/SETEX 将 key 的值设为 value，当且仅当 key 不存在。若给定的 key 已经存在，则 **SETEX** 不做任何动作。

```

v1q
127.0.0.1:6379> setex k3 10 v3
OK
127.0.0.1:6379> ttl k3
(error) ERR unknown command 'ttl'
127.0.0.1:6379> ttl k3
(integer) 2
127.0.0.1:6379> ttl k3
(integer) -2
127.0.0.1:6379> setnx k1 f1
(integer) 0
127.0.0.1:6379> setnx k3 se
(integer) 1
127.0.0.1:6379> get k3
"se"
127.0.0.1:6379> ttl
(error) ERR wrong number of arguments for 'ttl' command
127.0.0.1:6379> ttl \k3
(integer) -2
127.0.0.1:6379> ttl k3
(integer) -1

```

mset/mget/msetnx 同时设置一个或多个 key-value 对。/返回所有(一个或多个)给定 key 的值。

同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在

同时设置一个或多个 key-value 对，且只当所有给定 key 都存在时。

即使只有一个给定 key 已存在，**MSETNX** 也会拒绝执行所有给定 key 的设置操作。

```
127.0.0.1:6379> mset k4 v4 k5 v5 k6 v6
OK
127.0.0.1:6379> get k4
"v4"
127.0.0.1:6379> mget k4 k5 k6
1) "v4"
2) "v5"
3) "v6"
127.0.0.1:6379> msetnx k6 v7 k7 v7
(integer) 0
127.0.0.1:6379> msetnx k7 v7 k8 v8
(integer) 1
127.0.0.1:6379> mget k7 k8
1) "v7"
2) "v8"
```

2、) Hash (哈希, 类似java里面的Map)

KV模式不变, 但V是一个键值对

hset 添加/**hget** 获取/**hmset** 添加多个/**hmget** 获取多个/**hgetall**获取全部/**hdel**删除

hlen长度

```
127.0.0.1:6379> hset user id 11
(integer) 1
127.0.0.1:6379> hget user id
"11"
127.0.0.1:6379> hmset customer id 11 name lisi age 26
OK
127.0.0.1:6379> hget customer id name age
(error) ERR wrong number of arguments for 'hget' command
127.0.0.1:6379> hmget customer id name age
1) "11"
2) "lisi"
3) "26"
127.0.0.1:6379> hgetall customer
1) "id"
2) "11"
3) "name"
4) "lisi"
5) "age"
6) "26"
127.0.0.1:6379> hdel user name
(integer) 0
127.0.0.1:6379> hdel customer name
(integer) 1
127.0.0.1:6379> hlen customer
(integer) 2
127.0.0.1:6379> hgetall customer
1) "id"
2) "11"
3) "age"
4) "26"
```

hexists key 在key里面的某个值的key

```
127.0.0.1:6379> hexists customer age
(integer) 1
127.0.0.1:6379> hexists customer name
(integer) 0
```

hkeys 获取所有的key / **hvals** 获取所有的value

```
(integer) 0
127.0.0.1:6379> hkeys customer
1) "id"
2) "age"
127.0.0.1:6379> hvals customer
1) "11"
2) "26"
```

hincrby 整数类型相加 / **hincrbyfloat** 浮点型相加

```
127.0.0.1:6379> hkeys customer
1) "id"
2) "age"
127.0.0.1:6379> hvals customer
1) "11"
2) "26"
127.0.0.1:6379> hincrby customer age 2
(integer) 28
127.0.0.1:6379> hincrbyfloat customer age 3
"31"
127.0.0.1:6379> hincrbyfloat customer age 3.1
"34.1"
```

hsetnx增加不存在的数据

```
127.0.0.1:6379> hsetnx customer age 1
(integer) 0
127.0.0.1:6379> hsetnx customer email 324@qq.com
(integer) 1
127.0.0.1:6379> hgetall customer
1) "id"
2) "11"
3) "age"
4) "34.1"
5) "email"
6) "324@qq.com"
```

3、) List (列表)

单值多value

```
127.0.0.1:6379> lpush list 1 2 3 4 5
(integer) 5
127.0.0.1:6379> lrange list
(error) ERR wrong number of arguments for 'lrange' command
127.0.0.1:6379> lrange list 0 -1
```

```
1) "5"  
2) "4"  
3) "3"  
4) "2"  
5) "1"
```

lpush/rpush/lrange

```
127.0.0.1:6379> lpush list1 1 2 3 4 5  
(integer) 5  
127.0.0.1:6379> lrange list1 0 -1  
1) "5"  
2) "4"  
3) "3"  
4) "2"  
5) "1"  
127.0.0.1:6379> rpush list2 1 2 3 4 5  
(integer) 5  
127.0.0.1:6379> lrange list2 0 -1  
1) "1"  
2) "2"  
3) "3"  
4) "4"  
5) "5"
```

lpop/rpop 移除并返回列表 key 的头元素。/移除并返回列表 key 的尾元素。

```
(integer) 5  
127.0.0.1:6379> lrange list2 0 -1  
1) "1"  
2) "2"  
3) "3"  
4) "4"  
5) "5"  
127.0.0.1:6379> lpop list1  
"5"  
127.0.0.1:6379> lrange list1 0 -1  
1) "4"  
2) "3"  
3) "2"  
4) "1"  
127.0.0.1:6379> rpop list1  
"1"  
127.0.0.1:6379> lrange list1 0 -1  
1) "4"  
2) "3"  
3) "2"  
127.0.0.1:6379> █
```

llen key 返回列表 key 的长度。

lrem key 删N个value

ltrim key 开始index结束index，截取指定范围的值后再赋值给key

```
10) "4"  
11) "4"  
12) "4"  
13) "4"  
14) "4"  
15) "4"
```

```

16) "5"
17) "5"
18) "5"
127.0.0.1:6379> ltrim list 3 7
OK
127.0.0.1:6379> lrange list
(error) ERR wrong number of arguments for 'lrange'
127.0.0.1:6379> lrange list 0 -1
1) "1"
2) "1"
3) "2"
4) "3"
5) "4"
127.0.0.1:6379>

```

RPOPLPUSH source destination

命令 **RPOPLPUSH** 在一个原子时间内，执行以下两个动作：

- 将列表 source 中的最后一个元素(尾元素)弹出，并返回给客户端。
- 将 source 弹出的元素插入到列表 destination，作为 destination 列表的头元素。

```

127.0.0.1:6379> lrange list1 0 -1
1) "4"
2) "3"
3) "2"
127.0.0.1:6379> lrange list2 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
127.0.0.1:6379> rpoplpush list1 list2
"2"
127.0.0.1:6379> lrange list1 0 -1
1) "4"
2) "3"
127.0.0.1:6379> lrange list2 0 -1
1) "2"
2) "1"
3) "2"
4) "3"
5) "4"
6) "5"

```

LSET key index value

将列表 key 下标为 index 的元素的值设置为 value。

当 index 参数超出范围，或对一个空列表(key 不存在)进行 **LSET** 时，返回一个错误。

```

127.0.0.1:6379> lrange list2 0 -1
1) "2"
2) "1"
3) "2"
4) "3"
5) "4"
6) "5"
127.0.0.1:6379> lset list1 2 z

```



```
(error) ERR index out of range
127.0.0.1:6379> lset list2 2 z
OK
127.0.0.1:6379> lrange list2 0 -1
1) "2"
2) "1"
3) "z"
4) "3"
5) "4"
6) "5"
```

LINSERT key BEFORE|AFTER pivot value

将值 value 插入到列表 key 当中，位于值 pivot 之前或之后。

当 pivot 不存在于列表 key 时，不执行任何操作。

当 key 不存在时，key 被视为空列表，不执行任何操作。

如果 key 不是列表类型，返回一个错误。

```
127.0.0.1:6379> lrange list2 0 -1
1) "2"
2) "1"
3) "z"
4) "3"
5) "4"
6) "5"
127.0.0.1:6379> linsert list2 before 2 s
(integer) 7
127.0.0.1:6379> lrange list2
(error) ERR wrong number of arguments for 'lrange'
127.0.0.1:6379> lrange list2 0 -1
1) "s"
2) "2"
3) "1"
4) "z"
5) "3"
6) "4"
7) "5"
127.0.0.1:6379> linsert list2 after 5 e
(integer) 8
127.0.0.1:6379> lrange list2 0 -1
1) "s"
2) "2"
3) "1"
4) "z"
5) "3"
6) "4"
7) "5"
8) "e"
```

它是一个字符串链表，left、right都可以插入添加；

如果键不存在，创建新的链表；

如果键已存在，新增内容；

如果值全移除，对应的键也就小时了。

链表的操作无论是头和尾效率都极高，但假如是对中间元素进行操作，效率就很惨淡了。

4、) Set (集合)

```
127.0.0.1:6379> sadd set1 v1
(integer) 1
127.0.0.1:6379> sadd set1 v1
(integer) 0
127.0.0.1:6379> sadd set1 v1
(integer) 0
127.0.0.1:6379> sadd set v2 v3 v4
(integer) 3
127.0.0.1:6379> smembers set1
1) "v1"
127.0.0.1:6379> sadd set1 v2 v3 v4
(integer) 3
127.0.0.1:6379> smembers set1
1) "v4"
2) "v3"
3) "v2"
4) "v1"
```

sadd key value1 value2 ...

增加key为key的set集合

smembers key查看set集合内容

```
127.0.0.1:6379> smembers set1
1) "v4"
2) "v3"
3) "v2"
4) "v1"
127.0.0.1:6379> sismember set v2
(integer) 1
127.0.0.1:6379> scard set1
(integer) 4
127.0.0.1:6379> srem set1 v2
(integer) 1
127.0.0.1:6379> smembers set1
1) "v4"
2) "v3"
3) "v1"
```

sismember 集合key里面是否存在该值

scard 获取集合里面的元素个数

srem key value 删除集合中元素

```
127.0.0.1:6379> smembers set1
1) "v4"
2) "v3"
3) "v1"
127.0.0.1:6379> srandmember set1
```

```

127.0.0.1:6379> srandmember set1
"v4"
127.0.0.1:6379> srandmember set1
"v3"
127.0.0.1:6379> srandmember set1
"v1"
127.0.0.1:6379> srandmember set1
"v4"
127.0.0.1:6379> srandmember set1
"v3"
127.0.0.1:6379> spop set1
"v4"
127.0.0.1:6379> spop set1
"v3"

```

srandmember key某个整数（随机出几个数）

spop key随机出栈（删除）

smove key1 key2 作用是将key1里面的某个值赋给key2

```

127.0.0.1:6379> smembers set1
1) "v2"
2) "v1"
3) "v6"
4) "v7"
5) "v4"
6) "v3"
7) "v5"
127.0.0.1:6379> smembers set2
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
127.0.0.1:6379> smove set1 set2
(error) ERR wrong number of arguments for 'smove' command
127.0.0.1:6379> smove set1 set2 v
(integer) 1
127.0.0.1:6379> smembers set1
1) "v6"
2) "v7"
3) "v4"
4) "v3"
5) "v1"
6) "v5"
127.0.0.1:6379> smembers set2
1) "v2"
2) "2"
3) "1"
4) "5"
5) "4"
6) "3"
7) "6"

```

sdiff/sinter/sunion差集/交集/并集

```

127.0.0.1:6379> smembers set1
1) "2"
2) "3"
3) "v5"

```

```

4) "v6"
5) "v7"
6) "1"
7) "v4"
8) "4"
9) "v3"
10) "v1"
127.0.0.1:6379> smembers set2
1) "v2"
2) "2"
3) "1"
4) "5"
5) "4"
6) "3"
7) "6"

```

```

127.0.0.1:6379> sdiff set1 set2
1) "v3"
2) "v1"
3) "v7"
4) "v6"
5) "v4"
6) "v5"
127.0.0.1:6379> sinter set1 set2
1) "2"
2) "1"
3) "4"
4) "3"
127.0.0.1:6379> sunion set1 set2
1) "2"
2) "3"
3) "v5"
4) "6"
5) "v2"
6) "v6"
7) "v7"
8) "1"
9) "5"
10) "v4"
11) "4"
12) "v3"
13) "v1"

```

5、) Zset (sorted set: 有序集合)

zadd 添加集合数据 / **zrange** 获取集合

```

(error) ERR value is not a valid float
127.0.0.1:6379> zadd zset01 60.1 v1 60.2 v2 60.3 v3 60.4 v4 60.5 v5
(integer) 5
127.0.0.1:6379> zrange zset01
(error) ERR wrong number of arguments for 'zrange' command
127.0.0.1:6379> zrange zset01 0 -1
1) "v1"
2) "v2"
3) "v3"
4) "v4"
5) "v5"
127.0.0.1:6379> zrange zset01 0 -1 withscores
(error) ERR syntax error

```

```
127.0.0.1:6379> zrange zset01 0 -1 withscores
1) "v1"
2) "60.1000000000000001"
3) "v2"
4) "60.2000000000000003"
5) "v3"
6) "60.299999999999997"
7) "v4"
8) "60.399999999999999"
9) "v5"
10) "60.5"
```

zrangebyscore key 开始score 结束score

```
127.0.0.1:6379> zrange zset01 0 -1 withscores
1) "v1"
2) "60.1000000000000001"
3) "v2"
4) "60.2000000000000003"
5) "v3"
6) "60.299999999999997"
7) "v4"
8) "60.399999999999999"
9) "v5"
10) "60.5"
127.0.0.1:6379> zrangebyscore zset01 60 60.25
1) "v1"
2) "v2"
```

withscores

(不包含

Limit 作用是返回限制 limit 开始下标步 多少步

```
127.0.0.1:6379> ZRANGEBYSCORE zset01 60 (90
1) "v1"
2) "v2"
3) "v3"
127.0.0.1:6379> ZRANGEBYSCORE zset01 (60 (90
1) "v2"
2) "v3"
127.0.0.1:6379> ZRANGEBYSCORE zset01 60 90
1) "v1"
2) "v2"
3) "v3"
4) "v4"
127.0.0.1:6379> ZRANGEBYSCORE zset01 60 90 limit 2 2
1) "v3"
2) "v4"
```

zrem key 某score下对应的value值，作用是删除元素

```
127.0.0.1:6379> zrem zset01 v3
(integer) 1
127.0.0.1:6379> zrange zset 0 -1
(empty list or set)
127.0.0.1:6379> zrange zset01 0 -1
```



```
1) "v1"
2) "v2"
3) "v4"
4) "v5"
```

zcard/zcount key score区间个数

zrank key values值，作用是获得下标值

zscore key对应值，获得分数

```
127.0.0.1:6379> zrange zset01 0 -1
1) "v1"
2) "v2"
3) "v4"
4) "v5"
127.0.0.1:6379> zcard zset01
(integer) 4
127.0.0.1:6379> zcount zset01 60.2 60.3
(integer) 1
127.0.0.1:6379> zcount zset01 60 60.3
(integer) 2
127.0.0.1:6379> zrank zset01 v5
(integer) 3
127.0.0.1:6379> zscore zset01 v5
"60.5"
```

zrevrank key values值，作用是逆序获得下标值

zrevrange返回有序集 key 中，指定区间内的成员。

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

```
127.0.0.1:6379> zrevrangebyscore zset01 60.2 60
1) "v2"
2) "v1"
```

配置文件介绍

Units单位

```
# Redis configuration file example

# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1GB are all the same.
```

1 配置大小单位,开头定义了一些基本的度量单位,只支持bytes,不支持bit

2 对大小写不敏感

1. `daemonize yes` #是否以后台进程运行,默认为no
2. `pidfile /var/run/redis.pid` #如以后台进程运行,则需指定一个pid,默认为/var/run/redis.pid
3. `bind 127.0.0.1` #绑定主机IP,默认值为127.0.0.1(注释)
4. `port 6379` #监听端口,默认为6379
5. `timeout 300` #超时时间,默认为300(秒)
6. `loglevel notice` #日志记slave-serve-stale-data yes: 在master服务器挂掉或者同步失败时,从服务器是否继续提供服务。录等级,有4个可选值,debug,verbose(默认值),notice,warning
7. `logfile /var/log/redis.log` #日志记录方式,默认值为stdout
8. `databases 16` #可用数据库数,默认值为16,默认数据库为0
9. `save 900 1` #900秒(15分钟)内至少有1个key被改变
10. `save 300 10` #300秒(5分钟)内至少有300个key被改变
11. `save 60 10000` #60秒内至少有10000个key被改变
12. `rdbcompression yes` #存储至本地数据库时是否压缩数据,默认为yes
13. `dbfilename dump.rdb` #本地数据库文件名,默认值为dump.rdb
14. `dir ./` #本地数据库存放路径,默认值为 ./
- 15.
16. `slaveof 10.0.0.12 6379` #当本机为从服务时,设置主服务的IP及端口(注释)
17. `masterauth elain` #当本机为从服务时,设置主服务的连接密码(注释)
18. `slave-serve-stale-data yes` #在master服务器挂掉或者同步失败时,从服务器是否继续提供服务。
19. `requirepass elain` #连接密码(注释)
- 20.
21. `maxclients 128` #最大客户端连接数,默认不限制(注释)
22. `maxmemory` #设置最大内存,达到最大内存设置后,Redis会先尝试清除已到期或即将到期的Key,当此方法处理后,任到达最大内存设置,将无法再进行写入操作。(注释)
23. `appendonly no` #是否在每次更新操作后进行日志记录,如果不开启,可能会在断电时导致一段时间内的数据丢失。因为redis本身同步数据文件是按上面save条件来同步的,所以有的数据会在一段时间内只存在于内存中。默认值为no
24. `appendfilename appendonly.aof` #更新日志文件名,默认值为appendonly.aof(注释)
25. `appendfsync everysec` #更新日志条件,共有3个可选值。no表示等操作系统进行数据缓存同步到磁盘,always表示每次更新操作后手动调用fsync()将数据写到磁盘,everysec表示每秒同步一次(默认值)。
- 26.
27. `really-use-vm yes`
28. `vm-enabled yes` #是否使用虚拟内存,默认值为no
29. `vm-swap-file /tmp/redis.swap` #虚拟内存文件路径,默认值为/tmp/redis.swap,不可多个Redis实例共享
30. `vm-max-memory 0` #vm大小限制。0: 不限制,建议60-80% 可用内存大小。
31. `vm-page-size 32` #根据缓存内容大小调整,默认32字节。

```
32. | vm-pages 134217728 #page数。每 8 page, 会占用1字节内存。
33. | vm-page-size #vm-pages 等于 swap 文件大小
34. | vm-max-threads 4 #vm 最大io线程数。注意: 0 标志禁止使用vm
35. | hash-max-ipmap-entries 512
36. | hash-max-ipmap-value 64
37. |
38. | list-max-ziplist-entries 512
39. | list-max-ziplist-value 64
40. | set-max-intset-entries 512
41. | activerehashing yes
```

GENERAL通用

tcp-backlog连接队列--连接池

设置tcp的backlog, backlog七十是一个连接队列, backlog队列总和=未完成三次握手队列+已经完成三次握手队列。

在高并发环境下你需要一个高backlog值来避免慢客户端连接问题, 注意linux内核会将这个值减小到/proc/sys/net/core/somaxconn的值, 来达到想要的效果

*初步保持出厂设置511即可

timeout 超时时间, 0表示一直连着不关闭。

tcp-keepalive 集群检测通信 (单位为秒, 如果设置为0, 则不会进行Keepalive检测, 建议设置成60)

loglevel notice 日志级别 (debug开发阶段, verbose, notice, warning一般生产时)

logfile 日志存放不用为空("")

syslog-enabled no 系统日志默认关闭

syslog-ident 日志默认开头

syslog-facility local0 输出日志设备（用的话一般local0-local7）

databases 默认redis安装16个数据库

sercurity安全

config get requirepass查询密码

config get dir 启动redis目录(在什么目录下启动，配置文件就在该目录下dir ./ 设置路径)

config set requirepass "password" 设置redis密码

auth " password "输入redis密码，在操作之前

LIMITS限制

maxclients最大连接数，默认10000

maxmemory最大内存

缓存的移除策略、删除策略（volatile-lru、allkeys-lru、volatile-random、allkeys-random、volatile-ttl、noeviction）

volatile-lru 使用LRU算法移除Key，只对设置了过期时间的键

allkeys-lru 使用LRU算法移除Key

volatile-random 在过期集合中移除随机的key，只对设置了过期时间的键（最近最少使用）

allkeys-random 移除随机的key

volatile-ttl 移除那些TTL值最小的key，即那些最近要过期的key

noeviction 不进行移除。针对set操作，只是返回错误信息

maxmemory-policy过期策略选择，默认noeviction

Maxmemory-samples设置样本数量，LRU算法和最小TTL算法都并非是精确的算法，而是估算值，所以你可以设置样本的大小

redis默认回检查这么多个key并选择其中LRU的那个，默认选择5个



常见配置reids.conf介绍_3.txt
2018-06-03 00:28:46, 4.32 KB

redis持久化

rdb (redis database)

在指定的时间间隔内将内存中的数据快照写入磁盘

也就是行话讲的Snapshot快照，它恢复时是将快照文件直接读到内存里

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

fork,fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致。但是是一个全新的进程，并作为原进程的子进程

rdb保存的是dump.rdb文件

配置文件的位置SNAPSHOTTING快照

save <seconds> <changes>

默认

900秒内只要有一个key变动

300秒内10个key变动

60秒内10000key变动

修改后重启

*当shutdown时，迅速保存

*dbfilename dump.rdb重启后默认把dump.rdb读回内存

*save或bgsave命令会会当前进程进行备份

*save或者bgsave命令三则进行备份，

*save时只管保存，其他不管，全部阻塞；

*bgsave，redis会在后台异步进行快照操作，快照同时还可以响应客户端请求。可以通过lastsave命令获取最后一次成功执行快照的时间；

stop-writes-on-bgsave-error yes 默认出错后就停止

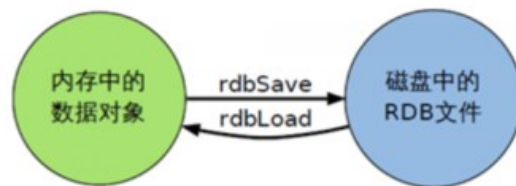
rdbcompression yes 对于存储到磁盘中的快照，可以设置是否进行压缩存储。如果是的话，redis会采用LZF算法进行压缩。如果你不想消耗CPU来进行压缩的话，可以设置为关闭此功能。

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

优势：适合大规模的数据恢复；对数据完整性和一致性要求不高；

劣势：在一定间隔时间做一次备份，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改；fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑如何停止；

如何停止：动态所有停止rdb保存规则的方法redis-cli config set save ""



- RDB是一个非常紧凑的文件
- RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他IO操作，所以RDB持久化方式可以最大化redis的性能.
- 与AOF相比,在恢复大的数据集的时候，RDB方式会更快一些.

- 数据丢失风险大
- RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级不能相应客户端请求

aof (append only file)

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

查找APPEND ONLY MODE

如遇断电等造成备份出错，可使用命令进行快速纠正备份中语法错误

redis-check-aof --fix xxx.aof-----y-----出现successfully truncated AOF

appendfsync:

Always: 同步持久化每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好

Everysec: 出厂默认推荐，异步操作，每秒记录，如果一秒内宕机，有数据丢失

No: 不进行记录

No-appendfsync-on-rewrite: 重写时是否可以运用Appendfsync，用默认no即可，保证数据安全性。

Auto-aof-rewrite-min-size: 设置重写的基准值，重写后的大小，制定一个重写下线，用来避免增长百分比够了，但是日志文件还很小的情况。

Auto-aof-rewrite-percentage: 设置重写的基准值，个值设置为100（当前写入日志文件的大小超过上一次rewrite之后的文件大小的百分之100时就是2倍时触发Rewrite）

```
# This base size is compared to the current size. If the current size is
# bigger than the specified percentage, the rewrite is triggered. Also
# you need to specify a minimal size for the AOF file to be rewritten, this
# is useful to avoid rewriting the AOF file even if the percentage increase
# is reached but it is still pretty small.
#
# Specify a percentage of zero in order to disable the automatic AOF
# rewrite feature.

auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

AOF启动/修改/恢复

正常恢复:

启动: 设置yes 修改默认的appendonly no, 改为yes

将有数据的aof文件复制一份保存到对应目录 (config get dir)

恢复: 重启redis然后重新加载

异常恢复:

启动: 设置yes

备份被写坏的AOF文件

修复被写坏的AOF文件: 使用redis-check-aof --fix

修复: Redis-check-aof --fix进行修复

恢复: 重启redis然后重新加载

rewrite

是什么: AOF采用文件追加方式, 文件会越来越大为避免出现此种情况, 新增了重写机制, 当AOF文件的大小超过所设定的阈值时, Redis就会启动AOF文件的内容压缩, 只保留可以恢复数据的最小指令集, 可以使用命令bgrewriteaof

重写原理: AOF文件持续增长而过大时, 会fork出一条新进程来将文件重写(也是先写临时文件最后再rename), 遍历新进程的内存中数据, 每条记录有一条的Set语句。重写aof文件的操作, 并没有读写旧的aof文件, 而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件, 这点和快照有点类似

触发机制: redis会记录上次重写时的AOF大小, 默认配置时当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发

优势:

每秒同步: appendfsync always 同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较高

每修改同步: appendfsync everysec 异步操作, 每秒记录 如果一秒内宕机, 有数据丢失

不同步: appendfsync no 从不同步

劣势:

相同数据集的数据而言aof文件要远大于rdb文件, 恢复速度慢于rdb

Aof运行效率要慢于rdb, 每秒同步策略效率较好, 不同步效率和rdb相同

官网建议 

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾。

Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大

只做缓存: 如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式。

同时开启两种持久化方式

在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

RDB的数据不实时,同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢? 作者建议不要,因为RDB更适合用于备份数据库(AOF在不断变化不好备份),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。

性能建议 



- AOF文件是一个只进行追加的日志文件
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写
- AOF 文件有序地保存了对数据库执行的所有写入操作，这些写入操作以 Redis 协议的格式保存，因此 AOF 文件的内容非常容易被别人读懂，对文件进行分析也很轻松

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB



性能建议_4.txt

2018-06-05 11:42:07, 732 字节

Redis事务

可以一次执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，**按顺序地串行化执行执行而不会被其他命令插入，不许加塞**

一个队列中，一次性、顺序性、排他性的执行一系列命令

multi、exec、discard

Redis 事务命令

下表列出了 redis 事务的相关命令：

序号	命令及描述
1	DISCARD 取消事务，放弃执行事务块内的所有命令。
2	EXEC 执行所有事务块内的命令。
3	MULTI 标记一个事务块的开始。
4	UNWATCH 取消 WATCH 命令对所有 key 的监视。
5	WATCH key [key...] 监视一个(或多个) key，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
  
```

```

127.0.0.1:6379> set k0 v1
QUEUED
127.0.0.1:6379> get k0
QUEUED
127.0.0.1:6379> exec
1) OK
2) OK
3) OK
4) "v1"
127.0.0.1:6379> get k0
"v1"
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set v3 k3
QUEUED
127.0.0.1:6379> discard
OK
127.0.0.1:6379> get v3
(nil)

```

正常执行

放弃事务

全体连坐

```

127.0.0.1:6379> MULTI
OK
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
127.0.0.1:6379> getset k3
(error) ERR wrong number of arguments for 'getset' command
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> set k5 v5
QUEUED
127.0.0.1:6379> EXEC
(error) EXECABORT Transaction discarded because of previous errors.

```

冤头债主

```

1) "k2"
2) "k3"
3) "k1"
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> incr k1
QUEUED
127.0.0.1:6379> set k2 22
QUEUED
127.0.0.1:6379> set k3 33
QUEUED
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> get k4
QUEUED

```



```
127.0.0.1:6379> EXEC
1) (error) ERR value is not an integer or out of range
2) OK
3) OK
4) OK
5) "v4"
127.0.0.1:6379> get k4
"v4"
```

出错命令打出QUEUED

watch监控

乐观锁/悲观锁/CAS (Check And Set)

悲观锁:

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

乐观锁:

乐观锁(Optimistic Lock), 顾名思义, 就是很乐观, 每次去拿数据的时候都认为别人不会修改, 所以不会上锁, 但是在更新的时候会判断一下在此期间别人有没有去更新这个数据, 可以使用版本号等机制。乐观锁适用于多读的应用类型, 这样可以提高吞吐量,

乐观锁策略:提交版本必须大于记录当前版本才能执行更新

加塞

```
root@atguigu:/usr/local/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
127.0.0.1:6379> WATCH balance
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> decrby balance 20
QUEUED
127.0.0.1:6379> incrby debt 20
QUEUED
127.0.0.1:6379> EXEC
1) (integer) 80
2) (integer) 20
127.0.0.1:6379> WATCH balance
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> decrby balance 20
QUEUED
127.0.0.1:6379> incrby debt 20
QUEUED
127.0.0.1:6379> EXEC
(nil)
127.0.0.1:6379> get balance
"800"
127.0.0.1:6379>
```

```
root@atguigu:/usr/local/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
127.0.0.1:6379> get balance
(nil)
127.0.0.1:6379> get balance
"80"
127.0.0.1:6379> set balance 800
OK
127.0.0.1:6379>
```

UNWATCH

```
127.0.0.1:6379> get debt
"20"
127.0.0.1:6379> WATCH balance
```

```

OK
127.0.0.1:6379> set balance 500
OK
127.0.0.1:6379> UNWATCH
OK
127.0.0.1:6379> WATCH balance
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set balance 80
QUEUED
127.0.0.1:6379> set debt 20
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) OK

```

*—但执行了unwatch或者exec，之前加的监控锁都会被取消掉

Watch指令，类似乐观锁，事务提交时，如果Key的值已被别的客户端改变，比如某个list已被别的客户端push/pop过了，整个事务队列都不会被执行

小结：通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Nullmulti-bulk应答以通知调用者事务执行失败

开启：以MULTI开始一个事务

入队：将多个命令入队到事务中，接到这些命令并不会立即执行，而是放到等待执行的事务队列里面

执行：由EXEC命令触发事务

特性：

单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

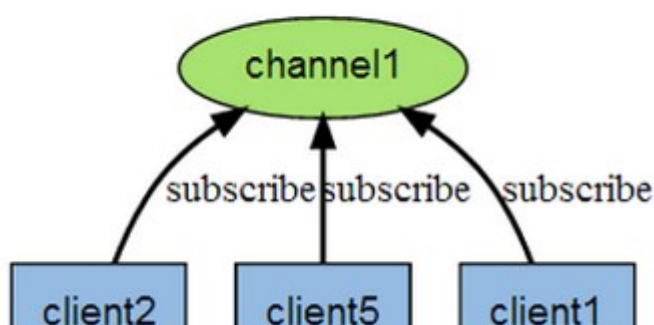
没有隔离级别的概念：队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行，也就不存在“事务内的查询要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题

不保证原子性：redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

Redis的发布订阅

进程间的一种消息通信模式：发送者（pub）发布消息，订阅者（sub）接收消息

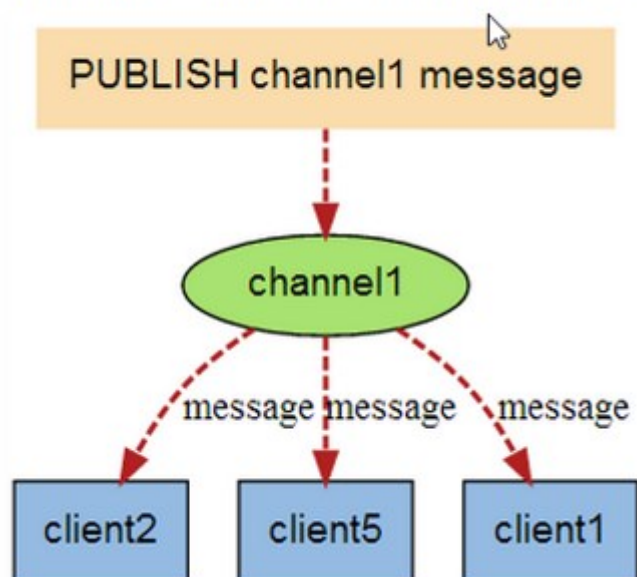
下图展示了频道 channel1，以及订阅这个频道的三个客户端—— client2、client5 和 client1 之间的关系：





1

当有新消息通过 PUBLISH 命令发送给频道 channel1 时，这个消息就会被发送给订阅它的三个客户端：



命令

序号	命令及描述
1	<u>PSUBSCRIBE pattern [pattern ...]</u> 订阅一个或多个符合给定模式的频道。
2	<u>PUBSUB subcommand [argument [argument ...]]</u> 查看订阅与发布系统状态。
3	<u>PUBLISH channel message</u> 将信息发送到指定的频道。
4	<u>PUNSUBSCRIBE [pattern [pattern ...]]</u> 退订所有给定模式的频道。
5	<u>SUBSCRIBE channel [channel ...]</u> 订阅给定的一个或多个频道的信息。
6	<u>UNSUBSCRIBE [channel [channel ...]]</u> 指退订给定的频道。

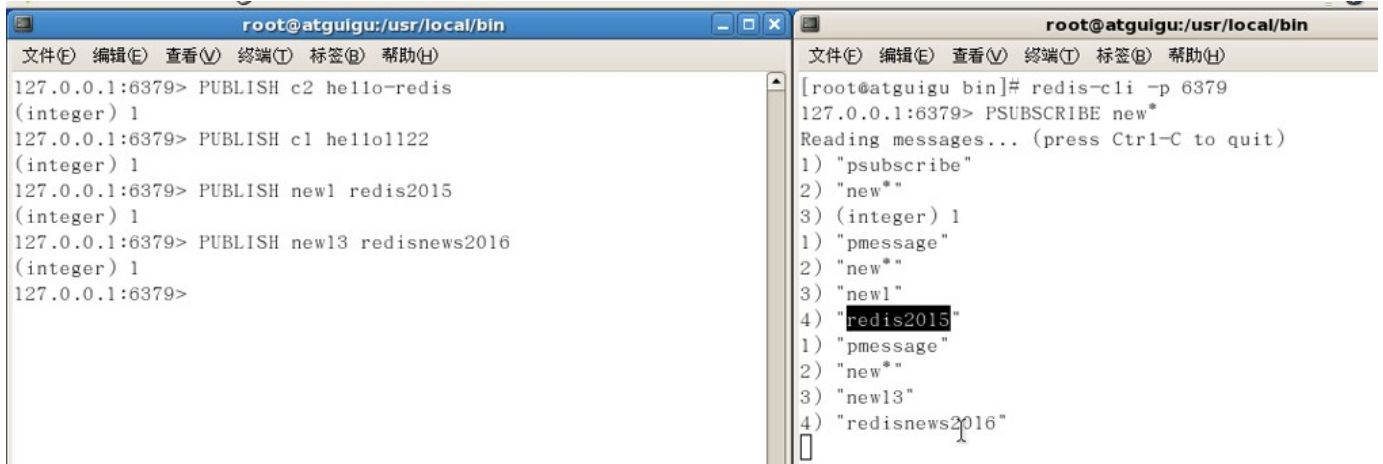
案例1:

```
127.0.0.1:6379> PUBLISH c2 hello-redis
(integer) 1
127.0.0.1:6379> PUBLISH c1 hello1122
(integer) 1
127.0.0.1:6379> █
```

```
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
1) "message"
2) "c2"
3) "hello-redis"
1) "message"
```

```
2) "c1"
3) "hello1122"
[]
```

案例2:



```
root@atguigu: /usr/local/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
127.0.0.1:6379> PUBLISH c2 hello-redis
(integer) 1
127.0.0.1:6379> PUBLISH c1 hello1122
(integer) 1
127.0.0.1:6379> PUBLISH new1 redis2015
(integer) 1
127.0.0.1:6379> PUBLISH new13 redisnews2016
(integer) 1
127.0.0.1:6379>

root@atguigu: /usr/local/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
[root@atguigu bin]# redis-cli -p 6379
127.0.0.1:6379> PSUBSCRIBE new*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "new*"
3) (integer) 1
1) "pmessage"
2) "new*"
3) "new1"
4) "redis2015"
1) "pmessage"
2) "new*"
3) "new13"
4) "redisnews2016"
```

主从复制(Master/Slave)、读写分离

主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写的为主，Slave以读为主

作用读写分离、容灾恢复

使用:

- 1、配从（库）不配主（库）
- 2、从库配置：命令 `slaveof`主库IP主库端口

每次与master断开之后，都需要重新连接，除非你配置进redis.conf文件

Info replication

- 3、修改配置文件细节操作:

拷贝多个redis.conf文件

开启daemonize yes

Pid文件名字

指定端口

log文件名字

Log文件名字

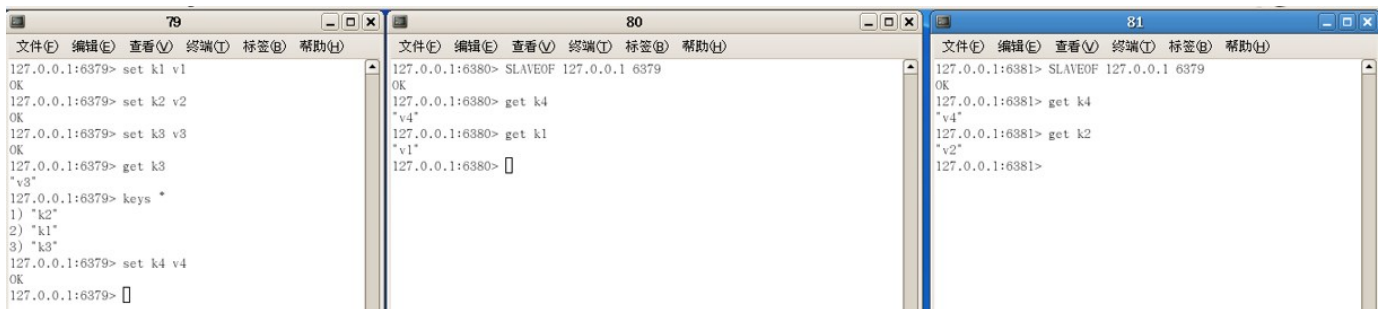
Dump.rdb名字

4、常用

info replication

```
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:867
master_link_down_since_seconds:40
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6380> []
```

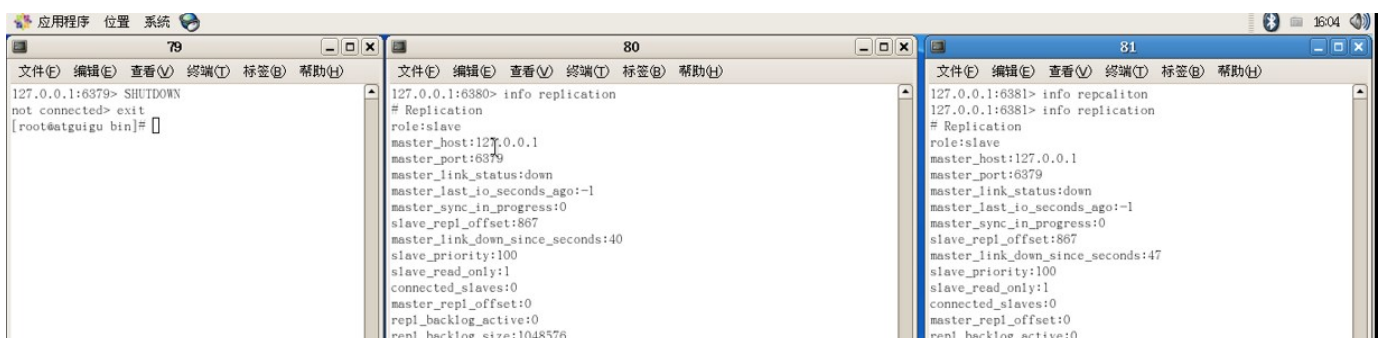
一主二仆



只能主机添加，从机无法写入



主机宕机，从机还是不变，原地待命




```
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6380> []
```

```
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6381> []
```

从机死了，重启后将解除关系



```
79
127.0.0.1:6379> set k8 v8
OK
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6381,state=online,offset=320,lag=1
master_repl_offset:320
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:319
127.0.0.1:6379> get k8
"v8"
127.0.0.1:6379> []

80
127.0.0.1:6380> SHUTDOWN
not connected> exit
Could not connect to Redis at 127.0.0.1:6380: Connection refused
not connected> exit
[root@atguigu bin]# redis-server /myredis/redis6380.conf
[root@atguigu bin]# redis-cli -p 6380
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6380> get k8
(nil)
127.0.0.1:6380>

81
127.0.0.1:6381> get k7
"v7"
127.0.0.1:6381> get k8
"v8"
127.0.0.1:6381> []
```

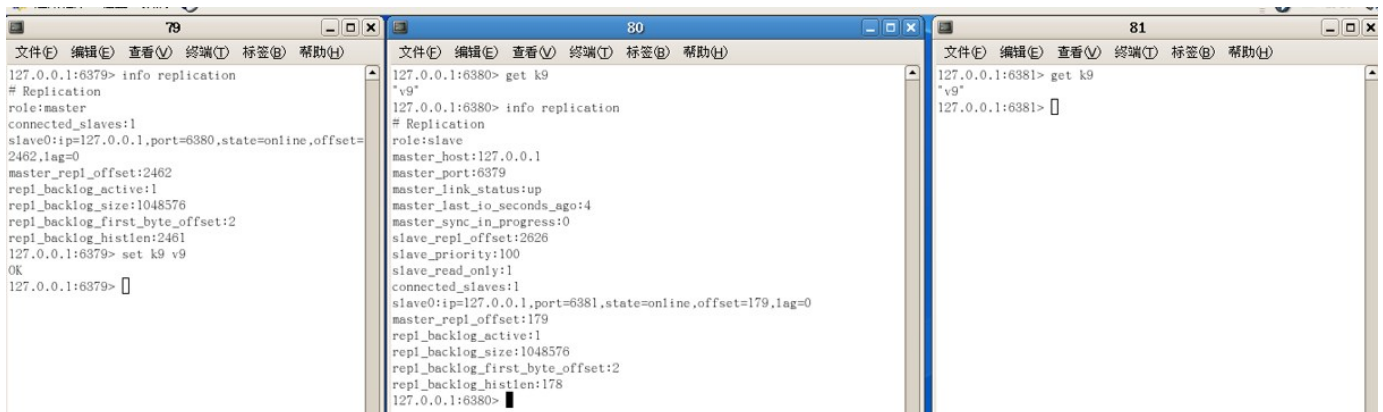
从机挂掉后重连，第一次全量，后续增量

薪火相传

上一个Slave可以是下一个slave的Master，Slave同样可以接收其他slaves的连接和同步请求，那么该slave作为了链条中下一个的master，可以有效减轻master的写压力

中途变更转向：会清除之前的数据，重新建立拷贝最新的

slaveof新主库IP新主库端口



```
79
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=2462,lag=0
master_repl_offset:2462
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:2461
127.0.0.1:6379> set k9 v9
OK
127.0.0.1:6379> []

80
127.0.0.1:6380> get k9
"v9"
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:4
master_sync_in_progress:0
slave_repl_offset:2626
slave_priority:100
slave_read_only:1
connected_slaves:1
slave0:ip=127.0.0.1,port=6381,state=online,offset=179,lag=0
master_repl_offset:179
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:178
127.0.0.1:6380>

81
127.0.0.1:6381> get k9
"v9"
127.0.0.1:6381> []
```

反客为主

slaveof no one 使当前数据库停止与其他数据库的同步，转成主数据库



```
79
127.0.0.1:6379> SHUTDOWN
not connected> exit
[root@atguigu bin]# redis-server /myredis/redis6379.conf
[root@atguigu bin]# redis-cli -p 6379
127.0.0.1:6379> get k10
(nil)
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

80
127.0.0.1:6380> SLAVEOF no one
OK
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:513
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:514
127.0.0.1:6380> keys *
1) "k5"
2) "k6"
3) "k4"
4) "k3"

81
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:2906
master_link_down_since_seconds:64
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
```

```
127.0.0.1:6379>
```

```
5) "k2"
6) "k8"
7) "k7"
8) "k1"
9) "k9"
127.0.0.1:6380> set k10 v10
OK
127.0.0.1:6380> get k10
"v10"
127.0.0.1:6380>
```

```
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6381> SLAVEOF 127.0.0.1 6380
OK
127.0.0.1:6381> get k10
"v10"
127.0.0.1:6381>
```

主从复制的复制原理

Slave启动成功连接到master后会发送一个sync命令

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，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

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。

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

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

哨兵模式

反客为主的自动版，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库

调整结构，6379带着80、81

自定义的/myredis目录下新建sentinel.conf文件，名字绝不能错

配置哨兵，填写内容 `sentinel monitor 被监控数据库名字(自己起名字) 127.0.0.1 6379 1`

上面最后一个数字1，表示主机挂掉后salve投票看让谁接替成为主机，得票数多

启动哨兵

正常主从演示

原有的master挂了

投票新选


重新主从继续开工,info replication查查看

问题：如果之前的master重启回来，会不会双master冲突？

```
[root@atguigu bin]# cd /myredis/
[root@atguigu myredis]# ls -l
总计 220
-rw-r--r-- 1 root root 41420 05-03 15:45 redis6379.conf
-rw-r--r-- 1 root root 41420 05-03 15:46 redis6380.conf
-rw-r--r-- 1 root root 41420 05-03 15:47 redis6381.conf
-rw-r--r-- 1 root root 41405 05-03 11:14 redis_aof.conf
-rw-r--r-- 1 root root 41404 05-03 10:18 redis.conf
[root@atguigu myredis]# touch sentinel.conf
[root@atguigu myredis]# ls -l
总计 220
-rw-r--r-- 1 root root 41420 05-03 15:45 redis6379.conf
-rw-r--r-- 1 root root 41420 05-03 15:46 redis6380.conf
-rw-r--r-- 1 root root 41420 05-03 15:47 redis6381.conf
-rw-r--r-- 1 root root 41405 05-03 11:14 redis_aof.conf
-rw-r--r-- 1 root root 41404 05-03 10:18 redis.conf
-rw-r--r-- 1 root root 0 05-03 16:46 sentinel.conf
```


vim sentinel.conf

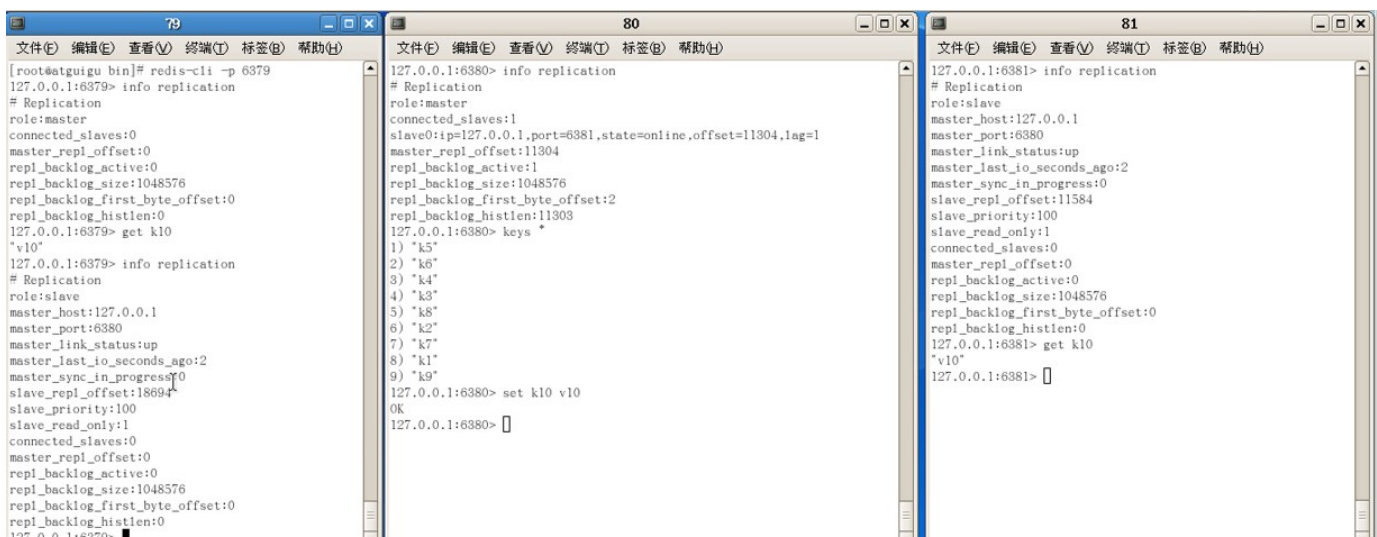


启动哨兵  Redis-sentinel /myredis/sentinel.conf
上述目录依照各自的实际情况配置，可能目录不同

案例:



主机重启后



复制的缺点:

复制延时

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。

测试联通



Jedis所需要的jar包

Commons-pool-1.6.jar

Jedis-2.1.0.jar

<terminated> testPing [Java Application] /opt/jdk1.7.0_67/bin/java (2016-5-4 上午9:22:29)

PONG

```
public class TestPing {
    public static void main(String[] args)
    {
        Jedis jedis = new Jedis("127.0.0.1",6379);
        System.out.println(jedis.ping());
    }
}
```

Jedis事务

```
public class TestTX {
    public static void main(String[] args) {
        Jedis jedis = new Jedis("127.0.0.1",6379);

        Transaction transaction = jedis.multi();

        transaction.set("k4","v4");
        transaction.set("k5","v5");

        transaction.exec();
    }
}
```

/**

- * 通俗点讲，watch命令就是标记一个键，如果标记了一个键，
 - * 在提交事务前如果该键被别人修改过，那事务就会失败，这种情况通常可以在程序中
 - * 重新再尝试一次。
 - * 首先标记了键balance，然后检查余额是否足够，不足就取消标记，并不做扣减；
 - * 足够的话，就启动事务进行更新操作，
 - * 如果在此期间键balance被其它人修改，那在提交事务（执行exec）时就会报错，
 - * 程序中通常可以捕获这类错误再重新执行一次，直到成功。
- */

```
int amtToSubtract = 10; // 实刷额度
```

```
jedis.watch("balance");
```

```
//jedis.set("balance","5");//此句不该出现，讲课方便。模拟其他程序已经修改了该条目
```

```
hread.sleep(1000);  
balance = Integer.parseInt(jedis.get("balance"));  
if (balance < amtToSubtract) {  
    jedis.unwatch();  
    System.out.println("modify");  
    return false;  
} else {  
    System.out.println("*****transaction");
```

JedisPool

示例:



JedisPoolUtil.txt