## Redis入门

### Introduction to Redis

From: <http://redis.io/topics/introduction>

Redis is an open source, BSD licensed, advanced **key-value store**. It is often referred to as a **data structure server** since keys can contain strings, hashes, lists, sets and sorted sets.

You can run **atomic operations** on these types, like appending to a string; incrementing the value in a hash; pushing to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

In order to achieve its outstanding performance, Redis works with an **in-memory dataset.** Depending on your use case, you can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log.

Redis also supports trivial-to-setup **master-slave replication**, with very fast non-blocking first synchronization, auto-reconnection on net split and so forth.

Other features include Transactions, Pub/Sub, Lua scripting, Keys with a limited time-to-live, and configuration settings to make Redis behave like a cache.

You can use Redis from most programming languages out there.

Redis is written in ANSI C and works in most POSIX systems like Linux, \*BSD, OS X without external dependencies. Linux and OSX are the two operating systems where Redis is developed and more tested, and we recommend using Linux for deploying. Redis may work in Solaris-derived systems like SmartOS, but the support is best effort. There is no official support for Windows builds, but Microsoft develops and maintains a Win32-64 experimental version of Redis.

### Allocator

By default Redis compiles and links against jemalloc under Linux, since

glibc malloc() has memory fragmentation problems.

To force a libc malloc() build use:

% make FORCE\_LIBC\_MALLOC=yes

In all the other non Linux systems the libc malloc() is used by default.

Redis推荐不使用GNU Malloc，而是推荐Jemalloc，该内存分配器为BSD系统默认。或者使用google研发的tcmalloc.

### Redis Command

配置文件redis.conf略…

To run Redis with the default configuration just type:

% cd src

% ./redis-server

If you want to provide your redis.conf, you have to run it using an additional

parameter (the path of the configuration file):

% cd src

% ./redis-server /path/to/redis.conf

You can use redis-cli to play with Redis. Start a redis-server instance,

then in another terminal try the following:

% cd src

% ./redis-cli

redis> ping

PONG

redis> set foo bar

OK

redis> get foo

"bar"

redis> incr mycounter

(integer) 1

redis> incr mycounter

(integer) 2

redis>

官方文档<http://redis.io/commands>

在解析返回结果时，如果不清楚返回的数据可以通过telnet到redis进行查看

often CMD：

set key value #添加key value

setnx key value #如果key已经存在，返回0，nx是not exist的意思。

mset key value [key value] #添加多个key-value

msetnx key value[…]

get key #通过key获取value

exists key #判断key是否存在

del key #删除key value

keys \* #列出所有结点, 例如：keys foo\*

hgetall key #获取所有hash的字段名+值

hkeys key #查看hash的所有字段名

hvals key #查看hash的所有值

select db-index 通过索引选择数据库，默认连接的数据库所有是0,默认数据库数是16个。返回1表示成功，0失败

move key db-index 将key从当前数据库移动到指定数据库。返回1成功。0 如果key不存在，或者已经在指定数据库中

flushdb 删除当前数据库中所有key,此方法不会失败。慎用

flushall 删除所有数据库中的所有key，此方法不会失败。更加慎用

incr key #value++ 主键递增

dbsize #当前结点数(记录数)

info #系统基本信息

flushall #清空所有记录

flushdb #清空当前DB所有记录

auth #密码登录

shundown #将数据同步保存到磁盘，然后关闭服务

save #立即保存数据到硬盘

bgsave #异步保存数据到硬盘

client list #获取客户连接列表

client kill 127.0.01:3422 #终止某个客户端连接

monitor #实时监控服务器收到的请求

设置登录密码：

1. 更改redis.conf配置

# requirepass foobared 去掉注释，foobared改为 自己的password , 如默认的密码为foobared

2.启动redis

/usr/local/redis/src/redis-server -p 6371

### Redis Protocol Specification

From <http://redis.io/topics/protocol>

The Redis protocol is a compromise between the following things:

* Simple to implement
* Fast to parse by a computer
* Easy enough to parse by a human

#### Networking layer

A client connects to a Redis server creating a TCP connection to the port 6379. Every Redis command or data transmitted by the client and the server is terminated by \r\n (CRLF).

#### Requests

Redis accepts commands composed of different arguments. Once a command is received, it is processed and a reply is sent back to the client.

#### The new unified request protocol

The new unified protocol was introduced in Redis 1.2, but it became the standard way for talking with the Redis server in Redis 2.0. This is the protocol you should implement in your Redis client.

In the unified protocol all the arguments sent to the Redis server are binary safe. This is the general form:

\*<number of arguments> CR LF $<number of bytes of argument 1> CR LF

<argument data> CR LF ... $<number of bytes of argument N> CR LF

<argument data> CR LF

See the following example:

\*3 $3 SET $5 mykey $7 myvalue

This is how the above command looks as a quoted string, so that it is possible to see the exact value of every byte in the query, including newlines.

"\*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"

As you will see in a moment this format is also used in Redis replies. The format used for the single argument $6\r\nmydata\r\n is called a**Bulk Reply**.

The unified request protocol is what Redis already uses in replies in order to send list of items to clients, and is called a **Multi Bulk Reply**. It is just the sum of N different Bulk Replies prefixed by a \*<argc>\r\n string where <argc> is the number of arguments (Bulk Replies) that will follow.

#### Replies

Redis will reply to commands with different kinds of replies. It is always possible to detect the kind of reply from the first byte sent by the server:

* In a Status Reply the first byte of the reply is "+"
* In an Error Reply the first byte of the reply is "-"
* In an Integer Reply the first byte of the reply is ":"
* In a Bulk Reply the first byte of the reply is "$"
* In a Multi Bulk Reply the first byte of the reply s "\*"

#### Status reply

A Status Reply (or: single line reply) is in the form of a single line string starting with "+" terminated by "\r\n". For example:

+OK

The client library should return everything after the "+", that is, the string "OK" in this example.

Status replies are not binary safe and can't include newlines, and are usually returned by commands that don't need to return data, but just some kind of status. Status replies have very little overhead of three bytes (the initial "+" and the final CRLF).

#### Error reply

Error Replies are very similar to Status Replies. The only difference is that the first byte is "-" instead of "+".

C: PING S: +PONG

C: EXISTS somekey S: :0

### redis API

from: <https://github.com/redis/hiredis>

#### Synchronous API

To consume the synchronous API, there are only a few function calls that need to be introduced:

redisContext \*redisConnect(const char \*ip, int port);

void \*redisCommand(redisContext \*c, const char \*format, ...);

void freeReplyObject(void \*reply);

structure:

/\* Context for a connection to Redis \*/

typedef struct redisContext {

int err; /\* Error flags, 0 when there is no error \*/

char errstr[128]; /\* String representation of error when applicable \*/

int fd;

int flags;

char \*obuf; /\* Write buffer \*/

redisReader \*reader; /\* Protocol reader \*/

} redisContext;

/\* This is the reply object returned by redisCommand() \*/

typedef struct redisReply {

int type; /\* REDIS\_REPLY\_\* \*/

long long integer; /\* The integer when type is REDIS\_REPLY\_INTEGER \*/

int len; /\* Length of string \*/

char \*str; /\* Used for both REDIS\_REPLY\_ERROR and REDIS\_REPLY\_STRING \*/

size\_t elements; /\* number of elements, for REDIS\_REPLY\_ARRAY \*/

struct redisReply \*\*element; /\* elements vector for REDIS\_REPLY\_ARRAY \*/

} redisReply;

#### Connecting

The function redisConnect is used to create a so-called redisContext. The context is where Hiredis holds state for a connection. TheredisContext struct has an integer err field that is non-zero when an the connection is in an error state. The field errstr will contain a string with a description of the error. More information on errors can be found in the **Errors** section. After trying to connect to Redis usingredisConnect you should check the err field to see if establishing the connection was successful:

redisContext \*c = redisConnect("127.0.0.1", 6379);

if (c != NULL && c->err) {

printf("Error: %s\n", c->errstr); // handle error

}

#### Sending commands

There are several ways to issue commands to Redis. The first that will be introduced is redisCommand. This function takes a format similar to printf. In the simplest form, it is used like this:

reply = redisCommand(context, "SET foo bar");

The specifier %s interpolates a string in the command, and uses strlen to determine the length of the string:

reply = redisCommand(context, "SET foo %s", value);

When you need to pass binary safe strings in a command, the %b specifier can be used. Together with a pointer to the string, it requires a size\_t length argument of the string:

**reply = redisCommand(context, "SET foo %b", value, valuelen);**

Internally, Hiredis splits the command in different arguments and will convert it to the protocol used to communicate with Redis. One or more spaces separates arguments, so you can use the specifiers anywhere in an argument:

reply = redisCommand(context, "SET key:%s %s", myid, value);

#### Using replies

The return value of redisCommand holds a reply when the command was successfully executed. When an error occurs, the return value is NULL and the err field in the context will be set (see section on **Errors**). Once an error is returned the context cannot be reused and you should set up a new connection.

The standard replies that redisCommand are of the type redisReply. The type field in the redisReply should be used to test what kind of reply was received:

* **REDIS\_REPLY\_STATUS**:
  + The command replied with a status reply. The status string can be accessed using reply->str. The length of this string can be accessed using reply->len.
* **REDIS\_REPLY\_ERROR**:
  + The command replied with an error. The error string can be accessed identical to REDIS\_REPLY\_STATUS.
* **REDIS\_REPLY\_INTEGER**:
  + The command replied with an integer. The integer value can be accessed using the reply->integer field of type long long.
* **REDIS\_REPLY\_NIL**:
  + The command replied with a **nil** object. There is no data to access.
* REDIS\_REPLY\_STRING:
  + A bulk (string) reply. The value of the reply can be accessed using reply->str. The length of this string can be accessed usingreply->len.
* **REDIS\_REPLY\_ARRAY**:
  + A multi bulk reply. The number of elements in the multi bulk reply is stored in reply->elements. Every element in the multi bulk reply is a redisReply object as well and can be accessed via reply->element[..index..]. Redis may reply with nested arrays but this is fully supported.

Replies should be freed using the freeReplyObject() function. Note that this function will take care of freeing sub-replies objects contained in arrays and nested arrays, so there is no need for the user to free the sub replies (it is actually harmful and will corrupt the memory).

**Important:** the current version of hiredis (0.10.0) free's replies when the asynchronous API is used. This means you should not callfreeReplyObject when you use this API. The reply is cleaned up by hiredis after the callback returns. This behavior will probably change in future releases, so make sure to keep an eye on the changelog when upgrading (see issue #39).

#### Cleaning up

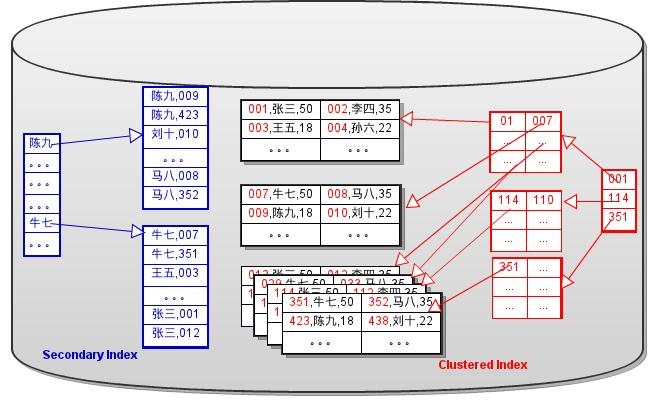
To disconnect and free the context the following function can be used:

void redisFree(redisContext \*c);

This function immediately closes the socket and then free's the allocations done in creating the context.

### Redis数据结构

聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。



Key:【项目名】\_【表名】\_pk/fk\_【字段名】\_【KEY】

Value:【C/C++ structure】

如： 用户账号信息，主键PK为【账号】：使用聚簇索引方式存储

dx\_account\_pk\_user\_ttxie 🡪 {结构体}

同时用户登录的时候又可以通过手机号码，邮箱进行登录，这样的话，就需要通过手机号码，邮箱来找到【账号】，再通过【账号】找到具体的数据。也就是需要二次定位来实现数据的查找。类似与非聚簇索引方式：

dx\_account\_fk\_mobile\_13925001700 🡪 [0xffff]dx\_account\_pk\_user\_ttxie 🡪 {结构体}

为了区分结构体数据项还是索引，我们在非聚簇索引上面加入两个字节：FFFF，用于区分是结构体数据项，还是索引？如果读到是FFFF，则继续往下读数据，直到拿到想要的数据。

redis-benchmark -h localhost -p 6379 -c 100 -n 100000   
100个并发连接，100000个请求，检测host为localhost 端口为6379的redis服务器性能

### Redis数据文件：

* 基本配置

默认：/usr/local/bin/dump.rdb

/etc/redis.conf中调整到指定目录，打开日志，以便于管理。

开启aof，保证数据安全。

开启databases的数量为8192，方便区服的管理，方便区服的清档。

* 恢复过程

只打开aof，使用aof加载

同时打开aof和rdb，使用aof加载

只打开rdb，使用rdb加载

rdb save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。【预留多一倍的内存用于数据的备份】

不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。

恢复测试数据：



云风的做法：在数据服务器的物理机上启动一个监护服务。当游戏服务器向数据服务推送数据并确认成功后，再把这组数据的 ID 同时发送给这个监护服务。它再从 Redis 中把数据读回来，并保存在本地。对于数据落地，我选择的是 [unqlite](http://unqlite.org/) ，几行代码就可以做好它的 Lua 封装。它的数据库文件只有一个，更方便做冷备。

* 建立独立的快照文件

src/rdb.c

~~/\*会影响复制，只能在slave上面操作，不能在master上面操作\*/~~

~~int rdbSave(char filename)~~

~~{~~

~~修改临时文件名称，每次生成的文件都不同。以备数据库的回档。~~

~~}~~

* rdb的合并:

----------------------------# RDB is a binary format. There are no new lines or spaces in the file.

52 45 44 49 53 # Magic String "REDIS"

30 30 30 33 # RDB Version Number in big endian. In this case, version = 0003 = 3

----------------------------

FE 00 # FE = code that indicates database selector. db number = 00

----------------------------# Key-Value pair starts

FD $unsigned int # FD indicates "expiry time in seconds". After that, expiry time is read as a 4 byte unsigned int

$value-type # 1 byte flag indicating the type of value - set, map, sorted set etc.

$string-encoded-key # The key, encoded as a redis string

$encoded-value # The value. Encoding depends on $value-type

----------------------------

FC $unsigned long # FC indicates "expiry time in ms". After that, expiry time is read as a 8 byte unsigned long

$value-type # 1 byte flag indicating the type of value - set, map, sorted set etc.

$string-encoded-key # The key, encoded as a redis string

$encoded-value # The value. Encoding depends on $value-type

----------------------------

$value-type # This key value pair doesn't have an expiry. $value\_type guaranteed != to FD, FC, FE and FF

$string-encoded-key

$encoded-value

----------------------------

FE $length-encoding # Previous db ends, next db starts. Database number read using length encoding.

----------------------------

... # Key value pairs for this database, additonal database

FF ## End of RDB file indicator

8 byte checksum ## CRC 64 checksum of the entire file.

* Database Selector

A Redis instance can have multiple databases.

A single byte 0xFE flags the start of the database selector. After this byte, a variable length field indicates the database number. See the section “Length Encoding” to understand how to read this database number.

* CRC64 Check Sum

Starting with RDB Version 5, an 8 byte CRC 64 checksum is added to the end of the file. It is possible to disable this checksum via a parameter in redis.conf.

**When checksum is disabled, this field will have zeroes。不用去调整redis.conf的检查配置项，直接跳过检查。**

依据该特性，则直接可以将被合并数据的主体部分【去掉文件头9个字节】合并，再修改属于自己的数据库编号，最后为了忽视redis的crc64的rdb检查，我们将8字节的crc全部置0.

执行**BGREWRITEAOF**时都发生了什么,下面来说一下Redis是如何实现的.分下面几步:

1. fork! Redis通过fork产生子进程.
2. 子进程对当前数据执行遍历操作,将当前所有数据都生成一条写入日志,将这些日志写入一个临时文件.(其实是子进程写了一个临时文件,又再rename成了另一个临时文件)
3. 父子进程是并行执行的,在子进程遍历并写临时文件的时候,父进程在照常接收请求,处理请求,写AOF,不过这时他是把新来的AOF写在一个缓冲区中.
4. 当子进程完成遍历操作,写完临时文件后,就会退出.这时父进程的wait3函数会接收到子进程退出的消息,他会把自己现在收集在缓冲区中的所有AOF追加在临时文件中.
5. 最后一步,把临时文件rename一下,改名为appendonly.aof,这时原来的aof文件被覆盖.整个过程完成.

## Redis性能

最后我做了一点简单的性能测试，用 lua 基于新的 lua 版 socket 库编写了一个符合 Redis 的 ping 协议的服务器。就是接收到一个以 PING\r\n 结束的数据包时，回应一条 +PONG\r\n 。

使用 redis-benchmark -t ping -n 100000 -c 10 做了个简单的测试。

可想而知，使用 C 语言编写的 Redis 服务器一定可以在这个测试中得到最高分，因为这个协议处理非常简单，完全是考虑 IO 处理的能力。在我一台旧机器上，redis 可以跑到 40k qps 。

redis-benchmark可以去深究下。

## Redis数据组织

String：String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

Hash：Redis Hash对应Value内部实际就是一个HashMap，实际这里会有2种不同实现，这个Hash的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，对应的value redisObject的encoding为zipmap,当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。

Zipmap: <zmlen><len>"foo"<len><free>"bar"<len>"hello"<len><free>"world"

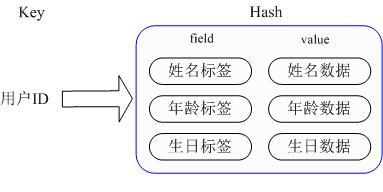
"\x02\x03foo\x03\x00bar\x05hello\x05\x00world\xff"

List(链表)：Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

Set(去重)：set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的

sorted Set(排序)：Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

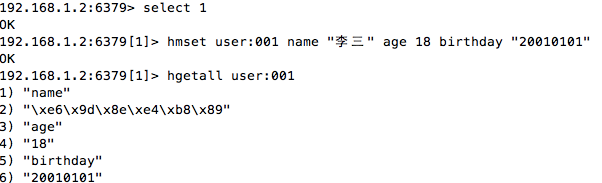
两层结构，可以在 Value 里保存一组 Hashes 。

下面是hash的常见应用场景，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，下图:  


 $redis->hset('h', 'field1', 'value1');

Redis-Dump开源小工具，将redis导出到json.

{"db":0,"key":"hashkey","ttl":-1,"type":"hash","value":{"field\_a":"value\_a","field\_b":"value\_b","field\_c":"value\_c"},"size":42} {"db":0,"key":"listkey","ttl":-1,"type":"list","value":["value\_0","value\_1","value\_2","value\_0","value\_1","value\_2"],"size":42} {"db":0,"key":"setkey","ttl":-1,"type":"set","value":["value\_2","value\_0","value\_1","value\_3"],"size":28} {"db":0,"key":"zsetkey","ttl":-1,"type":"zset","value":[["value\_0","100"],["value\_1","100"],["value\_2","200"],["value\_3","300"],["value\_4","400"]],"size":50} {"db":0,"key":"stringkey","ttl":79,"type":"string","value":"stringvalue","size":11}



redis> HSET people jack "Jack Sparrow"

(integer) 1

redis> HSET people gump "Forrest Gump"

(integer) 1

redis> HGETALL people

1) "jack" # 域

2) "Jack Sparrow" # 值

3) "gump"

4) "Forrest Gump"

redis五种数据类型的使用

<http://jandyu.diandian.com/post/2012-03-15/16145594>

* 从MySQL到Redis，提升数据迁移的效率

MySQL数据表结构：

CREATE TABLE events\_all\_time ( id int(11) unsigned NOT NULL AUTO\_INCREMENT, action varchar(255) NOT NULL, count int(11) NOT NULL DEFAULT 0, PRIMARY KEY (id), UNIQUE KEY uniq\_action (action) );

Redis存储结构：

HSET events\_all\_time [action] [count]

下面是重点，能过下面SQL语句将MySQL输出直接变更成redis-cli可接收的格式：

-- events\_to\_redis.sql

SELECT CONCAT( "\*4\r\n", '$', LENGTH(redis\_cmd), '\r\n', redis\_cmd, '\r\n', '$', LENGTH(redis\_key), '\r\n', redis\_key, '\r\n', '$', LENGTH(hkey), '\r\n', hkey,

'\r\n', '$', LENGTH(hval), '\r\n', hval, '\r' ) FROM ( SELECT 'HSET' as redis\_cmd, 'events\_all\_time' AS redis\_key, action AS hkey, count AS hval FROM events\_all\_time ) AS t

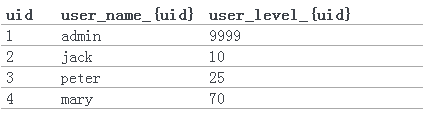
然后用管道符重定向输出即可：

mysql stats\_db

--skip-column-names --raw < events\_to\_redis.sql | redis-cli --pipe

## 排序

* 将哈希表作为get或by的参数

除了可以将字符串键之外， 哈希表也可以作为 get 或 by 选项的参数来使用。比如说，对于前面给出的用户信息表：  


我们可以不将用户的名字和级别保存在 user\_name\_{uid} 和 user\_level\_{uid} 两个字符串键中， 而是用一个带有 name 域和 level 域的哈希表 user\_info\_{uid} 来保存用户的名字和级别信息：

redis 127.0.0.1:6379> hmset user\_info\_1 name admin level 9999  
ok

redis 127.0.0.1:6379> hmset user\_info\_2 name jack level 10  
ok

redis 127.0.0.1:6379> hmset user\_info\_3 name peter level 25  
ok

redis 127.0.0.1:6379> hmset user\_info\_4 name mary level 70  
ok

之后， by 和 get 选项都可以用 key->field 的格式来获取哈希表中的域的值， 其中 key 表示哈希表键， 而 field 则表示哈希表的域：

redis 127.0.0.1:6379> sort uid by user\_info\_\*->level  
1) "2"  
2) "3"  
3) "4"  
4) "1"

redis 127.0.0.1:6379> sort uid by user\_info\_\*->level get user\_info\_\*->name  
1) "jack"  
2) "peter"  
3) "mary"  
4) "admin"

存储排序结果

redis 127.0.0.1:6379> sort numbers store sorted-numbers  
(integer) 10

## 复制

* 配置Slave：

只需要将redis-slave.conf中REPLICATION段中的slaveof <masterip> <masterport>行的注释去掉，并修改为：  
slaveof 127.0.0.1 6379  
即完成该Slave的配置，并指向本地端口为6379的Master端。

masterauth  
如果Master端通过requirepass设置了密码，Slave需要对应的通过masterauth <password>设置密码；

slave-serve-stale-data  
当Slave和Master断开连接时，Slave是直接返回错误提示还是利用历史数据响应客户端（或是直接返回空数据，当全量复制进行时）。yes是缺省值，即利用历史数据响应。

slave-read-only  
缺省模式下，Slave服务器是只读的。

repl-ping-slave-peroid  
即心跳检测间隔时间，缺省值为10秒。

repl-timeout 60  
复制超时

* 手动同步

如果主服务器是 Redis 2.8 或以上版本，那么从服务器使用 PSYNC 命令来进行同步。  
如果主服务器是 Redis 2.8 之前的版本，那么从服务器使用 SYNC 命令来进行同步。

* 同步改进

2.8开始，当Master和Slave之间的连接断开之后，他们之间可以采用持续复制处理方式代替采用全量同步。  
Master端为复制流维护一个内存缓冲区（in-memory backlog），记录最近发送的复制流命令；同时，Master和Slave之间都维护一个复制偏移量(replication offset)和当前Master服务器ID（Master run id）。当网络断开，Slave尝试重连时：  
a. 如果MasterID相同（即仍是断网前的Master服务器），并且从断开时到当前时刻的历史命令依然在Master的内存缓冲区中存在，则Master会将缺失的这段时间的所有命令发送给Slave执行，然后复制工作就可以继续执行了；  
b. 否则，依然需要全量复制操作:master生成dump文件同步到slave；

# 命令速查

# 一、Key

　　Key命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| DEL | 删除给定的一个或多个 key，不存在的 key 会被忽略，返回值：被删除 key 的数量 |
| DUMP | 序列化给定 key，返回被序列化的值，使用 RESTORE 命令可以将这个值反序列化为 Redis 键 |
| EXISTS | 检查给定 key 是否存在 |
| EXPIRE | 为给定key设置有效时间，接受时间点 |
| EXPIREAT | 为给定key设置有效时间，接受时间戳timestamp |
| KEYS | 查找所有符合给定模式 pattern 的 key；KEYS \* 匹配数据库中所有 key；KEYS h?llo 匹配 hello，hallo等。KEYS h[ae]llo匹配hello和hallo |
| MIGRATE | 将 key 原子性地从当前实例传送到目标实例的指定数据库上，一旦传送成功， key 保证会出现在目标实例上，而当前实例上的 key 会被删除。执行的时候会阻塞进行迁移的两个实例 |
| MOVE | 将当前数据库的 key 移动到给定的数据库 db 当中 |
| OBJECT | 从内部察看给定 key 的 Redis 对象 |
| PERSIST | 移除给定 key 的有效时间 |
| PEXPIRE | 以毫秒为单位设置 key 的有效时间 |
| PEXPIREAT | 以毫秒为单位设置 key 的有效时间(timespan) |
| PTTL | 以毫秒为单位返回key的剩余有效时间 |
| RANDOMKEY | 从当前数据库中随机返回(已使用的)一个key |
| RENAME | 将Key改名 |
| RENAMENX | 当且仅当 newkey 不存在时，将 key 改名为 newkey |
| RESTORE | 反序列化给定的序列化值，并将它和给定的 key 关联 |
| SORT | 返回或保存给定列表、集合、有序集合 key 中经过排序的元素 |
| TTL | 以秒为单位，返回给定 key 的剩余有效时间 |
| TYPE | 返回 key 所储存的值的类型 |
| SCAN | 增量迭代 |

# 二、String

　　String命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| APPEND | 将值追加到指定key的值末尾，如果key不存在，则相当于增加操作。 |
| BITCOUNT | 计算给定字符串中，被设置为 1 的Bit位的数量。 |
| BITOP | 对一个或多个保存二进制位的字符串 key 进行位元操作 |
| DECR | 将 key 中储存的数字值减一。Key不存在，则将值置0，key类型不正确返回一个错误。 |
| DECRBY | 将key所储存的值减去指定数量 |
| GET | 返回key所关联的字符串值，如果Key储存的值不是字符串类型，返回一个错误。 |
| GETBIT | 对key所储存的字符串值，获取指定偏移量上的位 |
| GETRANGE | 返回key中字符串值的子字符串，字符串的截取范围由start和end两个偏移量决定 |
| GETSET | 将给定key的值设为value，并返回key的旧值。非字符串报错。 |
| INCR | 将 key 中储存的数字值增一。不能转换为数字则报错。 |
| INCRBY | 将key所储存的值加上指定增量 |
| INCRBYFLOAT | 为key中所储存的值加上指定的浮点数增量 |
| MGET | 返回所有(一个或多个)给定key的值 |
| MSET | 同时设置一个或多个key-value对 |
| MSETNX | 同时设置一个或多个key-value对，若一个key已被占用，则全部的执行取消。 |
| PSETEX | 以毫秒为单位设置 key 的有效时间 |
| SET | 将字符串值value关联到key |
| SETBIT | 对key所储存的字符串值，设置或清除指定偏移量上的位(bit) |
| SETEX | 将值value关联到 key，并将key的有效时间(秒) |
| SETNX | 当key未被使用时，设置为指定值 |
| 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 |
| STRLEN | 返回key所储存的字符串值的长度 |

# 三、Hash

　　Hash命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| HDEL | 删除哈希表 key 中的一个或多个指定域，不存在的域将被忽略。 |
| HEXISTS | 查看哈希表 key 中，给定域 field 是否存在 |
| HGET | 返回哈希表 key 中给定域 field 的值 |
| HGETALL | 返回哈希表 key 中，所有的域和值 |
| HINCRBY | 为哈希表 key 中的域 field 的值加上指定增量 |
| HINCRBYFLOAT | 为哈希表 key 中的域 field 加上指定的浮点数增量 |
| HKEYS | 返回哈希表 key 中的所有域 |
| HLEN | 返回哈希表 key 中域的数量 |
| HMGET | 返回哈希表 key 中，一个或多个给定域的值 |
| HMSET | 同时将多个 field-value (域-值)对设置到哈希表 key 中 |
| HSET | 将哈希表 key 中的域 field 的值设为 value |
| HSETNX | 当且仅当域 field 不存在时，将哈希表 key 中的域 field 的值设置为 value |
| HVALS | 返回哈希表 key 中所有域的值 |
| HSCAN | 增量迭代 |

# 四、List

　　List命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| BLPOP | 它是 LPOP 命令的阻塞版本，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，直到等待超时或发现可弹出元素为止 |
| BRPOP | 与BLPOP同义，弹出位置不同 |
| BRPOPLPUSH | 当列表 source 为空时， BRPOPLPUSH 命令将阻塞连接，直到等待超时 |
| LINDEX | 返回列表 key 中，下标为 index 的元素 |
| LINSERT | 将值 value 插入到列表 key 当中 |
| LLEN | 返回列表 key 的长度 |
| LPOP | 移除并返回列表 key 的头元素 |
| LPUSH | 将一个或多个值 value 插入到列表 key 的表头 |
| LPUSHX | 将值 value 插入到列表 key 的表头，当且仅当 key 存在并且是一个列表 |
| 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 |
| LREM | 根据参数 count 的值，移除列表中与参数 value 相等的元素 |
| LSET | 将列表 key 下标为 index 的元素的值设置为 value |
| LTRIM | 对一个列表进行修剪(trim)，就是说，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 |
| RPOP | 移除并返回列表 key 的尾元素 |
| RPOPLPUSH | 命令 RPOPLPUSH 在一个原子时间内，执行两个动作：  1、将列表 source 中的最后一个元素(尾元素)弹出，并返回给客户端。  2、将 source 弹出的元素插入到列表 destination ，作为 destination 列表的的头元素。 |
| RPUSH | 将一个或多个值 value 插入到列表 key 的表尾 |
| RPUSHX | 将值 value 插入到列表 key 的表尾，当且仅当 key 存在并且是一个列表 |

# 五、Set

 　　Set命令速查

|  |  |
| --- | --- |
| 命令 | 说明 |
| SADD | 将一个或多个 member 元素加入到集合 key 当中，已经存在于集合的 member 元素将被忽略 |
| SCARD | 返回集合 key 的集合中元素的数量 |
| SDIFF | 返回一个集合的全部成员，该集合是所有给定集合之间的差集 |
| SDIFFSTORE | 这个命令的作用和 SDIFF 类似，但它将结果保存到新集合，而不是简单地返回结果集 |
| SINTER | 返回一个集合的全部成员，该集合是所有给定集合的交集 |
| SINTERSTORE | 与SINTER类似，不过可以指定保存到新集合 |
| SISMEMBER | 判断 member 元素是否集合 key 的成员 |
| SMEMBERS | 返回集合 key 中的所有成员 |
| SMOVE | 将 member 元素从一个集合移动到另一个集合 |
| SPOP | 移除并返回集合中的一个随机元素 |
| SRANDMEMBER | 仅仅返回随机元素，而不对集合进行任何改动，与SPOP的区别在于不移除 |
| SREM | 移除集合 key 中的一个或多个 member 元素，不存在的 member 元素会被忽略 |
| SUNION | 返回一个集合的全部成员，该集合是所有给定集合的并集 |
| SUNIONSTORE | 与SUNION类似，不过可以指定保存到新集合 |
| SSCAN | 增量迭代 |

# 六、SortedSet

 　　SortedSet命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| ZADD | 将一个或多个 member 元素及其 score 值加入到有序集 key 当中 |
| ZCARD | 返回有序集 key 的基数 |
| ZCOUNT | 返回有序集 key 中， score 值在 min 和 max 之间(包括 score 值等于 min 或 max )的成员的数量 |
| ZINCRBY | 为有序集 key 的成员 member 的 score 值加上指定增量 |
| ZRANGE | 返回有序集 key 中，指定区间内的成员(小到大排列) |
| ZRANGEBYSCORE | 返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员 |
| ZRANK | 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 |
| ZREM | 移除有序集 key 中的一个或多个成员，不存在的成员将被忽略 |
| ZREMRANGEBYRANK | 移除有序集 key 中，指定排名(rank)区间内的所有成员 |
| ZREMRANGEBYSCORE | 移除有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员 |
| ZREVRANGE | 返回有序集 key 中，指定区间内的成员，成员位置按score大到小排列 |
| 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。成员按 score 值递减(从大到小)排列 |
| ZREVRANK | 返回有序集 key 中成员 member 的排名。其中有序集成员按 score 值递减(从大到小)排序 |
| ZSCORE | 返回有序集 key 中，成员 member 的 score 值 |
| ZUNIONSTORE | 计算给定的一个或多个有序集的并集，其中给定 key 的数量必须以 numkeys 参数指定，并将该并集(结果集)储存到新集合 |
| ZINTERSTORE | 计算给定的一个或多个有序集的交集，其中给定 key 的数量必须以 numkeys 参数指定，并将该交集(结果集)储存到新集合 |
| ZSCAN | 增量迭代 |

# 七、Pub/Sub

　　Pub/Sub命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| PSUBSCRIBE | 订阅一个或多个符合给定模式的频道 |
| PUBLISH | 将信息 message 发送到指定的频道 |
| PUBSUB | PUBSUB 是一个查看订阅与发布系统状态的内省命令 |
| PUNSUBSCRIBE | 指示客户端退订所有给定模式 |
| SUBSCRIBE | 订阅给定的一个或多个频道的信息 |
| UNSUBSCRIBE | 指示客户端退订给定的频道 |

# 八、Transaction

　　Transaction命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| DISCARD | 取消事务，放弃执行事务块内的所有命令 |
| EXEC | 执行所有事务块内的命令 |
| MULTI | 标记一个事务块的开始 |
| UNWATCH | 取消 WATCH 命令对所有 key 的监视 |
| WATCH | 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 |

# 九、Script

　　script命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| EVAL | 通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值 |
| EVALSHA | 根据给定的 sha1 校验码，对缓存在服务器中的脚本进行求值 |
| SCRIPT EXISTS | 给定一个或多个脚本的 SHA1 校验和，返回一个包含 0 和 1 的列表，  表示校验和所指定的脚本是否已经被保存在缓存当中 |
| SCRIPT FLUSH | 清除所有 Lua 脚本缓存 |
| SCRIPT KILL | 停止当前正在运行的 Lua 脚本，当且仅当这个脚本没有执行过任何写操作时，这个命令才生效。  这个命令主要用于终止运行时间过长的脚本 |
| SCRIPT LOAD | 将脚本 script 添加到脚本缓存中，但并不立即执行这个脚本 |

# 十、Connection

 　　connection命令速查:

|  |  |
| --- | --- |
| 命令 | 说明 |
| AUTH | 通过设置配置文件中 requirepass 项的值，可以使用密码来保护 Redis 服务器 |
| ECHO | 打印一个特定的信息 message ，测试时使用。 |
| PING | 使用客户端向 Redis 服务器发送一个 PING ，如果服务器运作正常的话，会返回一个 PONG，通常用于测试与服务器的连接是否仍然生效，或者用于测量延迟值 |
| QUIT | 请求服务器关闭与当前客户端的连接 |
| SELECT | 切换到指定的数据库，数据库索引号 index 用数字值指定，以 0 作为起始索引值 |

# 十一、Server

　　server命令速查：

|  |  |
| --- | --- |
| 命令 | 说明 |
| BGREWRITEAOF | 执行一个 AOF文件 重写操作。重写会创建一个当前 AOF 文件的体积优化版本。 |
| BGSAVE | 在后台异步(Asynchronously)保存当前数据库的数据到磁盘 |
| CLIENT GETNAME | 返回 CLIENT SETNAME 命令为连接设置的名字 |
| CLIENT KILL | 关闭地址为 ip:port 的客户端 |
| CLIENT LIST | 以人类可读的格式，返回所有连接到服务器的客户端信息和统计数据 |
| CLIENT SETNAME | 为当前连接分配一个名字 |
| CONFIG GET | CONFIG GET 命令用于取得运行中的 Redis 服务器的配置参数 |
| CONFIG RESETSTAT | 重置 INFO 命令中的某些统计数据 |
| CONFIG REWRITE | CONFIG REWRITE 命令对启动 Redis 服务器时所指定的 redis.conf 文件进行改写 |
| CONFIG SET | CONFIG SET 命令可以动态地调整 Redis 服务器的配置而无须重启 |
| DBSIZE | 返回当前数据库的 key 的数量 |
| DEBUG OBJECT | DEBUG OBJECT 是一个调试命令，它不应被客户端所使用 |
| DEBUG SEGFAULT | 执行一个不合法的内存访问从而让 Redis 崩溃，仅在开发时用于 BUG 模拟 |
| FLUSHALL | 清空整个 Redis 服务器的数据(删除所有数据库的所有 key ) |
| FLUSHDB | 清空当前数据库中的所有 key |
| INFO | 返回关于 Redis 服务器的各种信息和统计数值 |
| LASTSAVE | 返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示 |
| MONITOR | 实时打印出 Redis 服务器接收到的命令，调试用 |
| PSYNC | 用于复制功能的内部命令 |
| SAVE | SAVE 命令执行一个同步保存操作，将当前 Redis 实例的所有数据快照(snapshot)以 RDB 文件的形式保存到硬盘。 一般来说，在生产环境很少执行 SAVE 操作，因为它会阻塞所有客户端，保存数据库的任务通常由 BGSAVE 命令异步地执行。然而，如果负责保存数据的后台子进程不幸出现问题时， SAVE 可以作为保存数据的最后手段来使用。 |
| SHUTDOWN | SHUTDOWN 命令执行以下操作：  停止所有客户端 如果有至少一个保存点在等待，执行 SAVE 命令 如果 AOF 选项被打开，更新 AOF 文件 关闭 redis 服务器(server) |
| SLAVEOF | SLAVEOF 命令用于在 Redis 运行时动态地修改复制(replication)功能的行为 |
| SLOWLOG | Slow log 是 Redis 用来记录查询执行时间的日志系统 |
| SYNC | 用于复制功能的内部命令 |
| TIME | 返回当前服务器时间 |