REDIS

tutorialspoint

SIMPLYEASYLEARNING

About the Tutorial

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

This tutorial provides good understanding on Redis concepts, needed to create and deploy a highly scalable and performance-oriented system.

Audience

This tutorial is designed for Software Professionals who are willing to learn Redis in simple and easy steps. After completing this tutorial, you will be at an intermediate level of expertise from where you can take yourself to a higher level of expertise.

Prerequisites

Before proceeding with this tutorial, you should have basic knowledge of Data Structures.

Disclaimer & Copyright

Table of Contents

Redis - Basics

1. Redis – Overview

Redis is an open source, advanced key-value store and an apt solution for building high-performance, scalable web applications.

Redis has three main peculiarities that sets it apart.

- Redis holds its database entirely in the memory, using the disk only for persistence.
- Redis has a relatively rich set of data types when compared to many key-value data stores.
- Redis can replicate data to any number of slaves.

Redis Advantages

Following are certain advantages of Redis.

- **Exceptionally fast**: Redis is very fast and can perform about 110000 SETs per second, about 81000 GETs per second.
- **Supports rich data types**: Redis natively supports most of the datatypes that developers already know such as list, set, sorted set, and hashes. This makes it easy to solve a variety of problems as we know which problem can be handled better by which data type.
- **Operations are atomic**: All Redis operations are atomic, which ensures that if two clients concurrently access, Redis server will receive the updated value.
- **Multi-utility tool**: Redis is a multi-utility tool and can be used in a number of use cases such as caching, messaging-queues (Redis natively supports Publish/Subscribe), any short-lived data in your application, such as web application sessions, web page hit counts, etc.

Redis Versus Other Key-value Stores

- Redis is a different evolution path in the key-value DBs, where values can contain more complex data types, with atomic operations defined on those data types.
- Redis is an in-memory database but persistent on disk database, hence it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than the memory.

  Another advantage of in-memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk. Thus, Redis can do a lot with little internal complexity.

2. Redis – Environment

In this chapter, you will learn about the environmental setup for Redis.

Install Redis on Ubuntu

To install Redis on Ubuntu, go to the terminal and type the following commands:

```
$sudo apt-get update
$sudo apt-get install redis-server
```

This will install Redis on your machine.

Start Redis

```
$redis-server
```

Check If Redis is Working

```
$redis-cli
```

This will open a redis prompt.

```
redis 127.0.0.1:6379>
```

In the above prompt, **127.0.0.1** is your machine's IP address and **6379** is the port on which Redis server is running. Now type the following **PING** command.

```
redis 127.0.0.1:6379> ping
PONG
```

This shows that Redis is successfully installed on your machine.

Install Redis Desktop Manager on Ubuntu

To install Redis desktop manager on Ubuntu, just download the package from http://redisdesktop.com/download

Open the downloaded package and install it.

Redis desktop manager will give you UI to manage your Redis keys and data.

3. Redis – Configuration

In Redis, there is a configuration file (redis.conf) available at the root directory of Redis. Although you can get and set all Redis configurations by Redis **CONFIG** command.

Syntax

Following is the basic syntax of Redis **CONFIG** command.

```
redis 127.0.0.1:6379> CONFIG GET CONFIG_SETTING_NAME
```

Example

```
redis 127.0.0.1:6379> CONFIG GET loglevel


1) "loglevel"
2) "notice"
```

To get all configuration settings, use ***** in place of CONFIG_SETTING_NAME

Example

```
redis 127.0.0.1:6379> CONFIG GET *


  1) "dbfilename"
  2) "dump.rdb"
  3) "requirepass"
  4) ""
  5) "masterauth"
  6) ""
  7) "unixsocket"
  8) ""
  9) "logfile"
 10) ""
 11) "pidfile"
 12) "/var/run/redis.pid"
 13) "maxmemory"
 14) "0"
```

```
15) "maxmemory-samples"
16) "3"
17) "timeout"
18) "0"
19) "tcp-keepalive"
20) "0"
21) "auto-aof-rewrite-percentage"
22) "100"
23) "auto-aof-rewrite-min-size"
24) "67108864"
25) "hash-max-ziplist-entries"
26) "512"
27) "hash-max-ziplist-value"
28) "64"
29) "list-max-ziplist-entries"
30) "512"
31) "list-max-ziplist-value"
32) "64"
33) "set-max-intset-entries"
34) "512"
35) "zset-max-ziplist-entries"
36) "128"
37) "zset-max-ziplist-value"
38) "64"
39) "hll-sparse-max-bytes"
40) "3000"
41) "lua-time-limit"
42) "5000"
43) "slowlog-log-slower-than"
44) "10000"
45) "latency-monitor-threshold"
46) "0"
47) "slowlog-max-len"
48) "128"
49) "port"
50) "6379"
```

```
51) "tcp-backlog"
52) "511"
53) "databases"
54) "16"
55) "repl-ping-slave-period"
56) "10"
57) "repl-timeout"
58) "60"
59) "repl-backlog-size"
60) "1048576"
61) "repl-backlog-ttl"
62) "3600"
63) "maxclients"
64) "4064"
65) "watchdog-period"
66) "0"
67) "slave-priority"
68) "100"
69) "min-slaves-to-write"
70) "0"
71) "min-slaves-max-lag"
72) "10"
73) "hz"
74) "10"
75) "no-appendfsync-on-rewrite"
76) "no"
77) "slave-serve-stale-data"
78) "yes"
79) "slave-read-only"
80) "yes"
81) "stop-writes-on-bgsave-error"
82) "yes"
83) "daemonize"
84) "no"
85) "rdbcompression"
86) "yes"
```

```
 87) "rdbchecksum"
 88) "yes"
 89) "activerehashing"
 90) "yes"
 91) "repl-disable-tcp-nodelay"
 92) "no"
 93) "aof-rewrite-incremental-fsync"
 94) "yes"
 95) "appendonly"
 96) "no"
 97) "dir"
 98) "/home/deepak/Downloads/redis-2.8.13/src"
 99) "maxmemory-policy"
100) "volatile-lru"
101) "appendfsync"
102) "everysec"
103) "save"
104) "3600 1 300 100 60 10000"
105) "loglevel"
106) "notice"
107) "client-output-buffer-limit"
108) "normal 0 0 0 slave 268435456 67108864 60 pubsub 33554432 8388608 60"
109) "unixsocketperm"
110) "0"
111) "slaveof"
112) ""
113) "notify-keyspace-events"
114) ""
115) "bind"
116) ""
```

Edit Configuration

To update configuration, you can edit **redis.conf** file directly or you can update configurations via **CONFIG set** command.

Syntax

Following is the basic syntax of **CONFIG SET** command.

```
redis 127.0.0.1:6379> CONFIG SET CONFIG_SETTING_NAME NEW_CONFIG_VALUE
```

Example

```
redis 127.0.0.1:6379> CONFIG SET loglevel "notice"
OK
redis 127.0.0.1:6379> CONFIG GET loglevel

1) "loglevel"
2) "notice"
```

4. Redis – Data Types

Redis supports 5 types of data types.

Strings

Redis string is a sequence of bytes. Strings in Redis are binary safe, meaning they have a known length not determined by any special terminating characters. Thus, you can store anything up to 512 megabytes in one string.

Example

```
redis 127.0.0.1:6379> SET name "tutorialspoint"
OK
redis 127.0.0.1:6379> GET name
"tutorialspoint"
```

In the above example, **SET** and **GET** are Redis commands, **name** is the key used in Redis and **tutorialspoint** is the string value that is stored in Redis.

Note: A string value can be at max 512 megabytes in length.

Hashes

A Redis hash is a collection of key value pairs. Redis Hashes are maps between string fields and string values. Hence, they are used to represent objects.

Example

```
redis 127.0.0.1:6379> HMSET user:1 username tutorialspoint password
tutorialspoint points 200
OK
redis 127.0.0.1:6379> HGETALL user:1

1) "username"
2) "tutorialspoint"
3) "password"
4) "tutorialspoint"
5) "points"
6) "200"
```

In the above example, hash data type is used to store the user's object which contains basic information of the user. Here **HMSET, HGETALL** are commands for Redis, while **user:1** is the key.

Every hash can store up to 232 - 1 field-value pairs (more than 4 billion).

Lists

Redis Lists are simply lists of strings, sorted by insertion order. You can add elements to a Redis List on the head or on the tail.

Example

```
redis 127.0.0.1:6379> lpush tutoriallist redis
(integer) 1
redis 127.0.0.1:6379> lpush tutoriallist mongodb
(integer) 2
redis 127.0.0.1:6379> lpush tutoriallist rabitmq
(integer) 3
redis 127.0.0.1:6379> lrange tutoriallist 0 10

1) "rabitmq"
2) "mongodb"
3) "redis"
```

The max length of a list is 232 - 1 elements (4294967295, more than 4 billion of elements per list).

Sets

Redis Sets are an unordered collection of strings. In Redis, you can add, remove, and test for the existence of members in O(1) time complexity.

Example

```
redis 127.0.0.1:6379> sadd tutoriallist redis
(integer) 1
redis 127.0.0.1:6379> sadd tutoriallist mongodb
(integer) 1
redis 127.0.0.1:6379> sadd tutoriallist rabitmq
(integer) 1
redis 127.0.0.1:6379> sadd tutoriallist rabitmq
(integer) 0
```

```
redis 127.0.0.1:6379> smembers tutoriallist


1) "rabitmq"
2) "mongodb"
3) "redis"
```

Note: In the above example, **rabitmq** is added twice, however due to unique property of the set, it is added only once.

The max number of members in a set is 232 - 1 (4294967295, more than 4 billion of members per set).

Sorted Sets

Redis Sorted Sets are similar to Redis Sets, non-repeating collections of Strings. The difference is, every member of a Sorted Set is associated with a score, that is used in order to take the sorted set ordered, from the smallest to the greatest score. While members are unique, the scores may be repeated.

Example

```
redis 127.0.0.1:6379> zadd tutoriallist 0 redis
(integer) 1
redis 127.0.0.1:6379> zadd tutoriallist 0 mongodb
(integer) 1
redis 127.0.0.1:6379> zadd tutoriallist 0 rabitmq
(integer) 1
redis 127.0.0.1:6379> zadd tutoriallist 0 rabitmq
(integer) 0
redis 127.0.0.1:6379> ZRANGEBYSCORE tutoriallist 0 1000


1) "redis"
2) "mongodb"
3) "rabitmq"
```

Redis – Commands

5. Redis – Commands

Redis commands are used to perform some operations on Redis server.

To run commands on Redis server, you need a Redis client. Redis client is available in Redis package, which we have installed earlier.

Syntax

Following is the basic syntax of Redis client.

```
$redis-cli
```

Example

Following example explains how we can start Redis client.

To start Redis client, open the terminal and type the command **redis-cli**. This will connect to your local server and now you can run any command.

```
$redis-cli
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

In the above example, we connect to Redis server running on the local machine and execute a command **PING**, that checks whether the server is running or not.

Run Commands on the Remote Server

To run commands on Redis remote server, you need to connect to the server by the same client **redis-cli**

Syntax

```
$ redis-cli -h host -p port -a password
```

Example

Following example shows how to connect to Redis remote server, running on host 127.0.0.1, port 6379 and has password mypass.

```
$redis-cli -h 127.0.0.1 -p 6379 -a "mypass"

redis 127.0.0.1:6379>

redis 127.0.0.1:6379> PING


PONG
```

6. Redis – Keys

Redis keys commands are used for managing keys in Redis. Following is the syntax for using redis keys commands.

Syntax

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
redis 127.0.0.1:6379> DEL tutorialspoint
(integer) 1
```

In the above example, **DEL** is the command, while **tutorialspoint** is the key. If the key is deleted, then the output of the command will be (integer) 1, otherwise it will be (integer) 0.

Redis Keys Commands

Following table lists some basic commands related to keys.

Sr. No.	Command & Description
1	**DEL key** This command deletes the key, if it exists
2	**DUMP key** This command returns a serialized version of the value stored at the specified key
3	**EXISTS key** This command checks whether the key exists or not
4	**EXPIRE key** seconds Sets the expiry of the key after the specified time
5	**EXPIREAT key timestamp** Sets the expiry of the key after the specified time. Here time is in Unix timestamp format
6	**PEXPIRE key milliseconds**

	Set the expiry of key in milliseconds
7	**PEXPIREAT key milliseconds-timestamp** Sets the expiry of the key in Unix timestamp specified as milliseconds
8	**KEYS pattern** Finds all keys matching the specified pattern
9	**MOVE key db** Moves a key to another database
10	**PERSIST key** Removes the expiration from the key
11	**PTTL key** Gets the remaining time in keys expiry in milliseconds
12	**TTL key** Gets the remaining time in keys expiry
13	**RANDOMKEY** Returns a random key from Redis
14	**RENAME key newkey** Changes the key name
15	**RENAMENX key newkey** Renames the key, if a new key doesn't exist
16	**TYPE key** Returns the data type of the value stored in the key

Keys Del Command

Redis **DEL** command is used to delete the existing key in Redis.

Return Value

Number of keys that were removed.

Syntax

Following is the basic syntax of Redis **DEL** command.

```
redis 127.0.0.1:6379> DEL KEY_NAME
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
```

Now, delete the previously created key.

```
redis 127.0.0.1:6379> DEL tutorialspoint
(integer) 1
```

Keys Dump Command

Redis **DUMP** command is used to get a serialized version of data stored at specified key in Redis.

Return Value

Serialized value (String)

Syntax

Following is the basic syntax of Redis **DUMP** command.

```
redis 127.0.0.1:6379> DUMP KEY_NAME
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
```

Now, create dump of the previously created key.

```
redis 127.0.0.1:6379> DUMP tutorialspoint
"\x00\x05redis\x06\x00S\xbd\xc1q\x17z\x81\xb2"
```

Keys Exists Command

Redis **EXISTS** command is used to check whether the key exists in Redis or not.

Return Value

Integer value

- 1, if the key exists.
- 0, if the key does not exist.

Syntax

Following is the basic syntax of Redis **EXISTS** command.

```
redis 127.0.0.1:6379> EXISTS KEY_NAME
```

Example

```
redis 127.0.0.1:6379> EXISTS tutorialspoint-new-key
(integer) 0
```

Now, create a key with the name tutorialspoint-new-key and check for its existence.

```
redis 127.0.0.1:6379> EXISTS tutorialspoint-new-key
(integer) 1
```

Keys Expire Command

Redis **Expire** command is used to set the expiry of a key. After the expiry time, the key will not be available in Redis.

Return Value

Integer value 1 or 0

- 1, if timeout is set for the key.
- 0, if the key does not exist or timeout could not be set.

Syntax

Following is the basic syntax of Redis **Expire** command.

```
redis 127.0.0.1:6379> Expire KEY_NAME TIME_IN_SECONDS
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
```

Now, set timeout of the previously created key.

```
redis 127.0.0.1:6379> EXPIRE tutorialspoint 60
(integer) 1
```

In the above example, 1 minute (or 60 seconds) time is set for the key tutorialspoint. After 1 minute, the key will expire automatically.

Keys Expireat Command

Redis **Expireat** command is used to set the expiry of key in Unix timestamp format. After the expiry time, the key will not be available in Redis.

Return Value

Integer value 1 or 0

- 1, if timeout is set for key.
- 0, if key does not exists or timeout could not set.

Syntax

Following is the basic syntax of Redis **Expireat** command.

```
redis 127.0.0.1:6379> Expireat KEY_NAME TIME_IN_UNIX_TIMESTAMP
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
```

Now, set timeout of the previously created key.

```
redis 127.0.0.1:6379> EXPIREAT tutorialspoint 1293840000
(integer) 1
EXISTS tutorialspoint
(integer) 0
```

Keys Pexpire Command

Redis **Pexpire** command is used to set the expiry of the key in milliseconds. After the expiry time, the key will not be available in Redis.

Return Value

Integer value 1 or 0

- 1, if the timeout is set for the key.
- 0, if the key does not exist or timeout could not be set.

Syntax

Following is the basic syntax of Redis **Expire** command.

```
redis 127.0.0.1:6379> PEXPIRE KEY_NAME TIME_IN_MILLISECONDS
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
```

Now, set timeout of the previously created key.

```
redis 127.0.0.1:6379> PEXPIRE tutorialspoint 5000
(integer) 1
```

In the above example, 5 seconds time is set for the key tutorialspoint. After 5 seconds, the key will expire automatically.

Keys Pexpireat Command

Redis **Pexpireat** command is used to set the expiry of the key in Unix timestamp specified in milliseconds. After the expiry time, the key will not be available in Redis.

Return Value

Integer value 1 or 0

- 1, if timeout is set for the key.
- 0, if the key does not exist or timeout could not be set.

Syntax

Following is the basic syntax of Redis **Pexpireat** command.

```
redis 127.0.0.1:6379> PEXPIREAT KEY_NAME TIME_IN_MILLISECONDS_IN_UNIX_TIMESTAMP
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
```

Now, set timeout of the previously created key.

```
redis 127.0.0.1:6379> PEXPIREAT tutorialspoint 1555555555005
(integer) 1
```

Keys Command

Redis **KEYS** command is used to search keys with a matching pattern.

Return Value

List of keys with a matching pattern (Array).

Syntax

Following is the basic syntax of Redis **KEYS** command.

```
redis 127.0.0.1:6379> KEYS PATTERN
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
redis 127.0.0.1:6379> SET tutorial2 mysql
OK
redis 127.0.0.1:6379> SET tutorial3 mongodb
OK
```

Now, search Redis with keys starting from the keyword tutorial.

```
redis 127.0.0.1:6379> KEYS tutorial*
1) "tutorial3"
2) "tutorial1"
3) "tutorial2"
```

To get a list of all the keys available in Redis, use only *

```
redis 127.0.0.1:6379> KEYS *
1) "tutorial3"
2) "tutorial1"
3) "tutorial2"
```

Keys Move Command

Redis **MOVE** command is used to move a key from the currently selected database to the specified destination database.

Return Value

Integer value 1 or 0

- 1, if the key is moved.
- 0, if the key is not moved.

Syntax

Following is the basic syntax of Redis **MOVE** command.

```
redis 127.0.0.1:6379> MOVE KEY_NAME DESTINATION_DATABASE
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
```

In Redis, by default 0th database is selected, so now we are moving the generated key in the second database.

```
redis 127.0.0.1:6379> MOVE tutorial1 1
1) (integer) 1
```

Keys Persist Command

Redis **PERSIST** command is used to remove the expiration from the key.

Return Value

Integer value 1 or 0

- 1, if timeout is removed from the key.
- 0, if the key does not exist or does not have an associated timeout.

Syntax

Following is the basic syntax of Redis **PERSIST** command.

```
redis 127.0.0.1:6379> PERSIST KEY_NAME
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
```

Now, set the expiry of the key, and later remove the expiry.

```
redis 127.0.0.1:6379> EXPIRE tutorial1 60
1) (integer) 1
redis 127.0.0.1:6379> TTL tutorial1
```

```
1) (integer) 60
redis 127.0.0.1:6379> PERSIST tutorial1
1) (integer) 1
redis 127.0.0.1:6379> TTL tutorial1
1) (integer) -1
```

Keys Pttl Command

Redis **PTTL** command is used to get the remaining time of the key expiry in milliseconds.

Return Value

Integer value TTL in milliseconds, or a negative value.

- TTL in milliseconds.
- -1, if the key does not have an expiry timeout.
- -2, if the key does not exist.

Syntax

Following is the basic syntax of Redis **PTTL** command.

```
redis 127.0.0.1:6379> PTTL KEY_NAME
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialname redis
OK
```

Now set the expiry of the key, and later check the remaining expiry time.

```
redis 127.0.0.1:6379> EXPIRE tutorialname 1
1) (integer) 1
redis 127.0.0.1:6379> PTTL tutorialname
1) (integer) 999
```

Keys Ttl Command

Redis **TTL** command is used to get the remaining time of the key expiry in seconds.

Return Value

Integer value TTL in milliseconds, or a negative value.

- TTL in milliseconds.

- -1, if key does not have expiry timeout.
- -2, if key does not exist.

Syntax

Following is the basic syntax of Redis **TTL** command.

```
redis 127.0.0.1:6379> TTL KEY_NAME
```

Example

First, create a key in Redis and set some value in it.

```
redis 127.0.0.1:6379> SET tutorialname redis
OK
```

Now, set the expiry of the key, and later check the remaining expiry time.

```
redis 127.0.0.1:6379> EXPIRE tutorialname 60
1) (integer) 1
redis 127.0.0.1:6379> TTL tutorialname
1) (integer) 59
```

Keys Random Key Command

Redis **RANDOMKEY** command is used to get a random key from Redis database.

Return Value

String, a random key or nil, if the database is empty.

Syntax

Following is the basic syntax of Redis **RANDOMKEY** command.

```
redis 127.0.0.1:6379> RANDOMKEY
```

Example

First, create some keys in Redis and set some values in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
redis 127.0.0.1:6379> SET tutorial2 mysql
OK
redis 127.0.0.1:6379> SET tutorial3 mongodb
```

```
OK
```

Now, get a random key from Redis.

```
redis 127.0.0.1:6379> RANDOMKEY
1) tutorial3
```

Keys Rename Command

Redis **RENAME** command is used to change the name of a key.

Return Value

String reply OK or error.

It returns an error if the old key and the new key names are equal, or when the key does not exist. If the new key already exists, then it overwrites the existing key.

Syntax

Following is the basic syntax of Redis **RENAME** command.

```
redis 127.0.0.1:6379> RENAME OLD_KEY_NAME NEW_KEY_NAME
```

Example

First, create some keys in Redis and set some values in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
```

Now, rename the key 'tutorial1' to 'new-tutorial'.

```
redis 127.0.0.1:6379> RENAME tutorial1 new-tutorial
OK
```

Keys Renamenx Command

Redis **RENAMENX** command is used to change the name of a key, if the new key does not exist.

Return Value

Integer reply 1 or 0.

- 1, if key is renamed to new key.
- 0, if a new key already exists.

Syntax

Following is the basic syntax of Redis **RENAMENX** command.

```
redis 127.0.0.1:6379> RENAMENX OLD_KEY_NAME NEW_KEY_NAME
```

Example

First, create some keys in Redis and set some values in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
redis 127.0.0.1:6379> SET tutorial2 mongodb
OK
```

Now, rename the key 'tutorial1' to 'new-tutorial'.

```
redis 127.0.0.1:6379> RENAMENX tutorial1 new-tutorial
(integer) 1
redis 127.0.0.1:6379> RENAMENX tutorial2 new-tutorial
(integer) 0
```

Keys Type Command

Redis **TYPE** command is used to get the data type of the value stored in the key.

Return Value

String reply, data type of the value stored in the key or none.

Syntax

Following is the basic syntax of Redis **TYPE** command.

```
redis 127.0.0.1:6379> TYPE KEY_NAME
```

Example

First, create some keys in Redis and set some values in it.

```
redis 127.0.0.1:6379> SET tutorial1 redis
OK
```

Now, check the type of the key.

```
redis 127.0.0.1:6379> TYPE tutorial1
string
```

7. Redis – Strings

Redis strings commands are used for managing string values in Redis. Following is the syntax for using Redis string commands.

Syntax

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
redis 127.0.0.1:6379> GET tutorialspoint
"redis"
```

In the above example, **SET and GET** are the commands, while **tutorialspoint** is the key.

Redis Strings Commands

Following table lists some basic commands to manage strings in Redis.

Sr. No.	Command & Description
1	**SET key value** This command sets the value at the specified key
2	**GET key** Gets the value of a key
3	**GETRANGE key start end** Gets a substring of the string stored at a key
4	**GETSET key value** Sets the string value of a key and return its old value
5	**GETBIT key offset** Returns the bit value at the offset in the string value stored at the key
6	**MGET key1 [key2..]** Gets the values of all the given keys

7	**SETBIT key offset value** Sets or clears the bit at the offset in the string value stored at the key
8	**SETEX key seconds value** Sets the value with the expiry of a key
9	**SETNX key value** Sets the value of a key, only if the key does not exist
10	**SETRANGE key offset value** Overwrites the part of a string at the key starting at the specified offset
11	**STRLEN key** Gets the length of the value stored in a key
12	**MSET key value [key value ...]** Sets multiple keys to multiple values
13	**MSETNX key value [key value ...]** Sets multiple keys to multiple values, only if none of the keys exist
14	**PSETEX key milliseconds value** Sets the value and expiration in milliseconds of a key
15	**INCR key** Increments the integer value of a key by one
16	**INCRBY key increment** Increments the integer value of a key by the given amount
17	**INCRBYFLOAT key increment** Increments the float value of a key by the given amount
18	**DECR key** Decrements the integer value of a key by one
19	**DECRBY key decrement** Decrements the integer value of a key by the given number
20	**APPEND key value** Appends a value to a key

String Set Command

Redis **SET** command is used to set some string value in Redis key.

Return Value

Simple string reply. OK, if the value is set in the key. Null, if the value does not set.

Syntax

Following is the basic syntax of Redis **SET** command.

```
redis 127.0.0.1:6379> SET KEY_NAME VALUE
```

Example

```
redis 127.0.0.1:6379> SET tutorialspoint redis

OK
```

Options

In **SET** command, there are many options available, that modify the behavior of command. Following is the basic syntax of SET command with available options.

```
redis 127.0.0.1:6379> SET KEY VALUE [EX seconds] [PX milliseconds] [NX|XX]
```

- **EX seconds** - Sets the specified expire time, in seconds.
- **PX milliseconds** - Sets the specified expire time, in milliseconds.
- **NX** - Only sets the key if it does not already exist.
- **XX** - Only sets the key if it already exists.

Example

```
redis 127.0.0.1:6379> SET tutorialspoint redis EX 60 NX

OK
```

The above example will set the key 'tutorialspoint', with expiry of 60 seconds, if the key does not exist.

String Get Command

Redis **GET** command is used to get the value stored in the specified key. If the key does not exist, then nil is returned. If the returned value is not a string, then error is returned.

Return Value

Simple string reply. Value or key or nil.

Syntax

Following is the basic syntax of Redis **GET** command.

```
redis 127.0.0.1:6379> GET KEY_NAME
```

Example

First, set a key in Redis and then get it.

```
redis 127.0.0.1:6379> SET tutorialspoint redis
OK
redis 127.0.0.1:6379> GET tutorialspoint
"redis"
```

String Getrange Command

Redis **GETRANGE** command is used to get the substring of the string value stored at the key, determined by the offsets start and end (both are inclusive). Negative offsets can be used in order to provide an offset starting from the end of the string.

The function handles out of range requests by limiting the resulting range to the actual length of the string.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **GETRANGE** command.

```
redis 127.0.0.1:6379> GETRANGE KEY_NAME start end
```

Example

First, set a key in Redis and then get some part of it.

```
redis 127.0.0.1:6379> SET mykey "This is my test key"
OK
redis 127.0.0.1:6379> GETRANGE mykey 0 3
"This"
redis 127.0.0.1:6379> GETRANGE mykey 0 -1
"This is my test key"
```

String Getset Command

Redis **GETSET** command sets the specified string value in Redis key and returns its old value.

Return Value

Simple string reply, old value of the key. If the key does not exist, then nil is returned.

Syntax

Following is the basic syntax of Redis **GETSET** command.

```
redis 127.0.0.1:6379> GETSET KEY_NAME VALUE
```

Example

```
redis 127.0.0.1:6379> GETSET mynewkey "This is my test key"
(nil)
redis 127.0.0.1:6379> GETSET mynewkey "This is my new value to test getset"
"This is my test key"
```

String Getbit Command

Redis **GETBIT** command is used to get the bit value at the offset in the string value stored at the key.

Return Value

Integer, the bit value stored at the offset.

Syntax

Following is the basic syntax of Redis **GETBIT** command.

```
redis 127.0.0.1:6379> GETBIT KEY_NAME OFFSET
```

Example

```
redis 127.0.0.1:6379> SETBIT mykey 7 1
(integer) 0
redis 127.0.0.1:6379> GETBIT mykey 0
(integer) 0
redis 127.0.0.1:6379> GETBIT mykey 7
(integer) 1
redis 127.0.0.1:6379> GETBIT mykey 100
```

```
(integer) 0
```

String Mget Command

Redis **MGET** command is used to get the values of all specified keys. For every key that does not hold a string value or does not exist, the special value nil is returned.

Return Value

Array, list of values at the specified keys.

Syntax

Following is the basic syntax of Redis **MGET** command.

```
redis 127.0.0.1:6379> MGET KEY1 KEY2 .. KEYN
```

Example

```
redis 127.0.0.1:6379> SET key1 "hello"
OK
redis 127.0.0.1:6379> SET key2 "world"
OK
redis 127.0.0.1:6379> MGET key1 key2 someOtherKey
1) "Hello"
2) "World"
3) (nil)
```

String Setbit Command

Redis **GETBIT** command is used to get the bit value at the offset in the string value stored at the key.

Return Value

Integer, the bit value stored at the offset.

Syntax

Following is the basic syntax of Redis **GETBIT** command.

```
redis 127.0.0.1:6379> GETBIT KEY_NAME OFFSET
```

Example

```
redis 127.0.0.1:6379> SETBIT mykey 7 1
(integer) 0
redis 127.0.0.1:6379> GETBIT mykey 0
(integer) 0
redis 127.0.0.1:6379> GETBIT mykey 7
(integer) 1
redis 127.0.0.1:6379> GETBIT mykey 100
(integer) 0
```

String Setex Command

Redis **SETEX** command is used to set some string value with a specified timeout in Redis key.

Return Value

Simple string reply. OK, if the value is set in key. Null, if the value is not set.

Syntax

Following is the basic syntax of Redis **SETEX** command.

```
redis 127.0.0.1:6379> SETEX KEY_NAME TIMEOUT VALUE
```

Example

```
redis 127.0.0.1:6379> SETEX mykey 60 redis
OK
redis 127.0.0.1:6379> TTL mykey
60
redis 127.0.0.1:6379> GET mykey
"redis
```

String Setnx Command

Redis **SETNX** command is used to set some string value in Redis key, if the key does not exist in Redis. Fullform of SETNX is **SET** if **N**ot e**X**ists.

Return Value

Integer reply 1 or 0

- 1, if the key is set.
- 0, if the key is not set.

Syntax

Following is the basic syntax of Redis **SETNX** command.

```
redis 127.0.0.1:6379> SETNX KEY_NAME VALUE
```

Example

```
redis 127.0.0.1:6379> SETNX mykey redis
(integer) 1
redis 127.0.0.1:6379> SETNX mykey mongodb
(integer) 0
redis 127.0.0.1:6379> GET mykey
"redis"
```

String Setrange Command

Redis **SETRANGE** command is used to overwrite a part of a string at the key starting at the specified offset.

Return Value

Integer reply, the length of the string after it was modified by the command.

Syntax

Following is the basic syntax of Redis **SETRANGE** command.

```
redis 127.0.0.1:6379> SETRANGE KEY_NAME OFFSET VALUE
```

Example

```
redis 127.0.0.1:6379> SET key1 "Hello World"
OK
redis 127.0.0.1:6379> SETRANGE key1 6 "Redis"
(integer) 11
redis 127.0.0.1:6379> GET key1
"Hello Redis"
```

String Strlen Command

Redis **STRLEN** command is used to get the length of the string value stored at the key. An error is returned when the key holds a non-string value.

Return Value

Integer reply, the length of the string at the key, or 0 when the key does not exist.

Syntax

Following is the basic syntax of Redis **SETRANGE** command.

```
redis 127.0.0.1:6379> STRLEN KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SET key1 "Hello World"
OK
redis 127.0.0.1:6379> STRLEN key1
(integer) 11
redis 127.0.0.1:6379> STRLEN key2
(integer) 0
```

String Mset Command

Redis **MSET** command is used to set multiple values to multiple keys.

Return Value

Simple string reply OK.

Syntax

Following is the basic syntax of Redis **MSET** command.

```
redis 127.0.0.1:6379> MSET key1 value1 key2 value2 .. keyN valueN
```

Example

```
redis 127.0.0.1:6379> MSET key1 "Hello" key2 "World"
OK
redis 127.0.0.1:6379> GET key1
"Hello"
redis 127.0.0.1:6379> GET key2
1) "World"
```

String Msetnx Command

Redis **MSETNX** command is used to set multiple values to multiple keys, only if none of them already exist. If any one from the current operation exists in Redis, then MSETNX does not perform any operation.

Return Value

Integer reply 1 or 0

- 1, if all the keys are set in Redis.
- 0, if no keys are set in Redis.

Syntax

Following is the basic syntax of Redis **MSETNX** command.

```
redis 127.0.0.1:6379> MSETNX key1 value1 key2 value2 .. keyN valueN
```

Example

```
redis 127.0.0.1:6379> MSETNX key1 "Hello" key2 "world"
(integer) 1
redis 127.0.0.1:6379> MSETNX key2 "worlds" key3 "third key"
(integer) 0
redis 127.0.0.1:6379> MGET key1 key2 key3
1) "Hello"
2) "world"
3) (nil)
```

Redis – String Psetex Command

Redis **PSETEX** command is used to set the value of a key, with expiration of time in milliseconds.

Return Value

Simple string reply OK.

Syntax

Following is the basic syntax of Redis **PSETEX** command.

```
redis 127.0.0.1:6379> PSETEX key1 EXPIRY_IN_MILLISECONDS value1
```

Example

```
redis 127.0.0.1:6379> PSETEX mykey 1000 "Hello"
OK
redis 127.0.0.1:6379> PTTL mykey
999
redis 127.0.0.1:6379> GET mykey
1) "Hello"
```

String Incr Command

Redis **INCR** command is used to increment the integer value of a key by one. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that cannot be represented as an integer. This operation is limited to 64 bit signed integers.

Return Value

Integer reply, the value of the key after the increment.

Syntax

Following is the basic syntax of Redis **INCR** command.

```
redis 127.0.0.1:6379> INCR KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SET visitors 1000
OK
redis 127.0.0.1:6379> INCR visitors
(integer) 1001
redis 127.0.0.1:6379> GET visitors
(integer) 1001
```

String Incrby Command

Redis **INCRBY** command is used to increment the number stored at the key by the specified value. If the key does not exist, it is set to 0 before performing the operation. An error is returned, if the key contains a value of the wrong type or contains a string that cannot be represented as an integer.

Return Value

Integer reply, the value of key after the increment.

Syntax

Following is the basic syntax of Redis **INCRBY** command.

```
redis 127.0.0.1:6379> INCRBY KEY_NAME INCR_AMOUNT
```

Example

```
redis 127.0.0.1:6379> SET visitors 1000
OK
redis 127.0.0.1:6379> INCRBY visitors 5
(integer) 1005
redis 127.0.0.1:6379> GET visitors
(integer) 1005
```

String Incrbyfloat Command

Redis **INCRBYFLOAT** command is used to increment the string representing a floating point number, stored at the key by the specified increment. If the key does not exist, it is set to 0 before performing the operation. If the key contains a value of the wrong type or the current key content or the specified increment are not parsable as floating point number, then an error is returned.

Return Value

String reply, the value of the key after the increment.

Syntax

Following is the basic syntax of Redis **INCRBYFLOAT** command.

```
redis 127.0.0.1:6379> INCRBYFLOAT KEY_NAME INCR_AMOUNT
```

Example

```
redis 127.0.0.1:6379> SET visitors 1000.20
OK
redis 127.0.0.1:6379> INCRBYFLOAT visitors .50
1000.70
redis 127.0.0.1:6379> GET visitors
1000.70
```

String Decr Command

Redis **DECR** command is used to decrement the integer value of a key by one. If the key does not exist, it is set to 0 before performing the operation. An error is returned, if the key contains a value of the wrong type or contains a string that cannot be represented as an integer. This operation is limited to 64 bit signed integers.

Return Value

Integer reply, the value of the key after the increment.

Syntax

Following is the basic syntax of Redis **DECR** command.

```
redis 127.0.0.1:6379> DECR KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SET visitors 1000
OK
redis 127.0.0.1:6379> DECR visitors
(integer) 999
redis 127.0.0.1:6379> SET visitors "131313123123123123123l2rgergerg"
Ok
redis 127.0.0.1:6379> DECR visitors
ERR value is not an integer or out of range
```

String Decrby Command

Redis **DECRBY** command is used to decrement the number stored at the key by the specified value. If the key does not exist, it is set to 0 before performing the operation. An error is returned, if the key contains a value of the wrong type or contains a string that cannot be represented as an integer.

Return Value

Integer reply, the value of the key after the increment.

Syntax

Following is the basic syntax of Redis **DECRBY** command.

```
redis 127.0.0.1:6379> DECRBY KEY_NAME DECREMENT_AMOUNT
```

Example

```
redis 127.0.0.1:6379> SET visitors 1000
OK
```

```
redis 127.0.0.1:6379> DECRBY visitors 5
(integer) 995
```

String Append Command

Redis **APPEND** command is used to add some value in a key.

Return Value

Integer reply, the length of the string after the append operation.

Syntax

Following is the basic syntax of Redis **APPEND** command.

```
redis 127.0.0.1:6379> APPEND KEY_NAME NEW_VALUE
```

Example

```
redis 127.0.0.1:6379> SET mykey "hello"
OK
redis 127.0.0.1:6379> APPEND mykey " tutorialspoint"
(integer) 20
redis 127.0.0.1:6379> GET mykey
"hello tutorialspoint"
```

8. Redis – Hashes

Redis Hashes are maps between the string fields and the string values. Hence, they are the perfect data type to represent objects.

In Redis, every hash can store up to more than 4 billion field-value pairs.

Example

```
redis 127.0.0.1:6379> HMSET tutorialspoint name "redis tutorial" description
"redis basic commands for caching" likes 20 visitors 23000
OK
redis 127.0.0.1:6379> HGETALL tutorialspoint


1) "name"
2) "redis tutorial"
3) "description"
4) "redis basic commands for caching"
5) "likes"
6) "20"
7) "visitors"
8) "23000"
```

In the above example, we have set Redis tutorials detail (name, description, likes, visitors) in hash named ‘tutorialspoint’.

Redis Hash Commands

Following table lists some basic commands related to hash.

Sr. No.	Command & Description
1	**HDEL key field2 [field2]** Deletes one or more hash fields.
2	**HEXISTS key field** Determines whether a hash field exists or not.

3	**HGET key field** Gets the value of a hash field stored at the specified key
4	**HGETALL key** Gets all the fields and values stored in a hash at the specified key
5	**HINCRBY key field increment** Increments the integer value of a hash field by the given number
6	**HINCRBYFLOAT key field increment** Increments the float value of a hash field by the given amount
7	**HKEYS key** Gets all the fields in a hash
8	**HLEN key** Gets the number of fields in a hash
9	**HMGET key field1 [field2]** Gets the values of all the given hash fields
10	**HMSET key field1 value1 [field2 value2]** Sets multiple hash fields to multiple values
11	**HSET key field value** Sets the string value of a hash field
12	**HSETNX key field value** Sets the value of a hash field, only if the field does not exist
13	**HVALS key** Gets all the values in a hash
14	**HSCAN key cursor [MATCH pattern] [COUNT count]** Incrementally iterates hash fields and associated values

Hash Hdel Command

Redis **HDEL** command is used to remove specified fields from the hash stored at the key. Specified fields that do not exist within this hash are ignored. If the key does not exist, it is treated as an empty hash and this command returns 0.

Return Value

Integer reply, the number of fields that were removed from the hash, not including specified but non-existing fields.

Syntax

Following is the basic syntax of Redis **HDEL** command.

```
redis 127.0.0.1:6379> HDEL KEY_NAME FIELD1.. FIELDN
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HDEL myhash field1
(integer) 1
redis 127.0.0.1:6379> HDEL myhash field2
(integer) 1
```

Hash Hexists Command

Redis **HEXISTS** command is used to check whether a hash field exists or not.

Return Value

Integer reply, 1 or 0.

- 1, if the hash contains a field.
- 0 if the hash does not contain a field, or the key does not exist.

Syntax

Following is the basic syntax of Redis **HEXISTS** command.

```
redis 127.0.0.1:6379> HEXISTS KEY_NAME FIELD_NAME
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HEXISTS myhash field1
(integer) 1
redis 127.0.0.1:6379> HEXISTS myhash field2
(integer) 0
```

Hash Hget Command

Redis **HGET** command is used to get the value associated with the field in the hash stored at the key.

Return Value

String reply, the value associated with the field. Nil, when the field is not present in the hash or the key does not exist.

Syntax

Following is the basic syntax of Redis **HGET** command.

```
redis 127.0.0.1:6379> HGET KEY_NAME FIELD_NAME
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HGET myhash field1
"foo"
redis 127.0.0.1:6379> HEXISTS myhash field2
(nil)
```

Hash Hgetall Command

Redis **HGETALL** command is used to get all the fields and values of the hash stored at the key. In the returned value, every field name is followed by its value, so the length of the reply is twice the size of the hash.

Return Value

Array reply, list of fields and their values stored in the hash, or an empty list when the key does not exist.

Syntax

Following is the basic syntax of Redis **HGETALL** command.

```
redis 127.0.0.1:6379> HGETALL KEY_NAME
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HSET myhash field2 "bar"
```

```
(integer) 1
redis 127.0.0.1:6379> HGETALL myhash
1) "field1"
2) "Hello"
3) "field2"
4) "World"
```

Hash Hincrby Command

Redis **HINCRBY** command is used to increment the number stored at the field in the hash, stored at the key by increment. If the key does not exist, a new key holding a hash is created. If the field does not exist, the value is set to 0 before the operation is performed.

Return Value

Integer reply, the value at the field after the increment operation.

Syntax

Following is the basic syntax of Redis **HINCRBY** command.

```
redis 127.0.0.1:6379> HINCRBY KEY_NAME FIELD_NAME INCR_BY_NUMBER
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 20
(integer) 1
redis 127.0.0.1:6379> HINCRBY myhash field 1
(integer) 21
redis 127.0.0.1:6379> HINCRBY myhash field -1
(integer) 20
```

Hash Hincrbyfloat Command

Redis **HINCRBYFLOAT** command is used to increment the specified field of a hash stored at the key, and representing a floating point number, by the specified increment. If the field does not exist, it is set to 0 before performing the operation. If the field contains a value of wrong type or specified increment is not parsable as floating point number, then an error occurs.

Return Value

String reply, the value of the field after the increment.

Syntax

Following is the basic syntax of Redis **HINCRBYFLOAT** command.

```
redis 127.0.0.1:6379> HINCRBYFLOAT KEY_NAME FIELD_NAME INCR_BY_NUMBER
```

Example

```
redis 127.0.0.1:6379> HSET myhash field 20.50
(integer) 1
redis 127.0.0.1:6379> HINCRBYFLOAT mykey field 0.1
"20.60"
```

Hash Hkeys Command

Redis **HKEYS** command is used to get all field names in the hash stored at the key.

Return Value

Array reply, list of fields in the hash, or an empty list when the key does not exist.

Syntax

Following is the basic syntax of Redis **HKEYS** command.

```
redis 127.0.0.1:6379> HKEYS KEY_NAME FIELD_NAME INCR_BY_NUMBER
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HSET myhash field2 "bar"
(integer) 1
redis 127.0.0.1:6379> HKEYS myhash
1) "field1"
2) "field2"
```

Hash Hlen Command

Redis **HLEN** command is used to get the number of fields contained in the hash stored at the key.

Return Value

Integer reply, number of fields in the hash, or 0 when the key does not exist.

Syntax

Following is the basic syntax of Redis **HLEN** command.

```
redis 127.0.0.1:6379> HLEN KEY_NAME
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HSET myhash field2 "bar"
(integer) 1
redis 127.0.0.1:6379> HLEN myhash
(integer) 2
```

Hash Hmget Command

Redis **HMGET** command is used to get the values associated with the specified fields in the hash stored at the key. If the field does not exist in Redis hash, then a nil value is returned.

Return Value

Array reply, list of values associated with the given fields, in the same order as they are requested.

Syntax

Following is the basic syntax of Redis **HMGET** command.

```
redis 127.0.0.1:6379> HMGET KEY_NAME FIELD1...FIELDN
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HSET myhash field2 "bar"
(integer) 1
redis 127.0.0.1:6379> HMGET myhash field1 field2 nofield
1) "foo"
2) "bar"
3) (nil)
```

Hash Hmset Command

Redis **HMSET** command is used to set the specified fields to their respective values in the hash stored at the key. This command overwrites any existing fields in the hash. If the key does not exist, a new key holding a hash is created.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **HMSET** command.

```
redis 127.0.0.1:6379> HMSET KEY_NAME FIELD1 VALUE1 ...FIELDN VALUEN
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo" field2 "bar"
OK
redis 127.0.0.1:6379> HGET myhash field1
"foo"
redis 127.0.0.1:6379> HMGET myhash field2
"bar"
```

Hash Hset Command

Redis **HSET** command is used to set field in the hash stored at the key to value. If the key does not exist, a new key holding a hash is created. If the field already exists in the hash, it is overwritten.

Return Value

Integer reply

- 1 if fthe ield is a new field in the hash and value was set.
- 0 if the field already exists in the hash and the value was updated.

Syntax

Following is the basic syntax of Redis **HSET** command.

```
redis 127.0.0.1:6379> HSET KEY_NAME FIELD VALUE
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
OK
```

```
redis 127.0.0.1:6379> HGET myhash field1
"foo"
```

Hash Hsetnx Command

Redis **HSETNX** command is used to set field in the hash stored at the key to value, only if the field does not yet exist. If the key does not exist, a new key holding a hash is created. If the field already exists, this operation has no effect.

Return Value

Integer reply

- 1 if the field is a new field in the hash and value was set.
- 0 if the field already exists in the hash and no operation was performed.

Syntax

Following is the basic syntax of Redis **HSETNX** command.

```
redis 127.0.0.1:6379> HSETNX KEY_NAME FIELD VALUE
```

Example

```
redis 127.0.0.1:6379> HSETNX myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HSETNX myhash field1 "bar"
(integer) 0
redis 127.0.0.1:6379> HGET myhash field1
"foo"
```

Hash Hvals Command

Redis **HVALS** command is used to get all the values in the hash stored at the key.

Return Value

Array reply, list of values in the hash, or an empty list when the key does not exist.

Syntax

Following is the basic syntax of Redis **HVALS** command.

```
redis 127.0.0.1:6379> HVALS KEY_NAME FIELD VALUE
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
(integer) 1
redis 127.0.0.1:6379> HSET myhash field2 "bar"
(integer) 1
redis 127.0.0.1:6379> HVALS myhash
1) "foo"
2) "bar"
```

Hash Hset Command

Redis **HSET** command is used to set the field in the hash stored at the key to value. If the key does not exist, a new key holding a hash is created. If the field already exists in the hash, it is overwritten.

Return Value

Integer reply

- 1, if the field is a new field in the hash and the value was set.
- 0, if the field already exists in the hash and the value was updated.

Syntax

Following is the basic syntax of Redis **HSET** command.

```
redis 127.0.0.1:6379> HSET KEY_NAME FIELD VALUE
```

Example

```
redis 127.0.0.1:6379> HSET myhash field1 "foo"
OK
redis 127.0.0.1:6379> HGET myhash field1
"foo"
```

9. Redis – Lists

Redis Lists are simply lists of strings, sorted by insertion order. You can add elements in Redis lists in the head or the tail of the list.

Maximum length of a list is 2^{32} - 1 elements (4294967295, more than 4 billion of elements per list).

Example

```
redis 127.0.0.1:6379> LPUSH tutorials redis
(integer) 1
redis 127.0.0.1:6379> LPUSH tutorials mongodb
(integer) 2
redis 127.0.0.1:6379> LPUSH tutorials mysql
(integer) 3
redis 127.0.0.1:6379> LRANGE tutorials 0 10

1) "mysql"
2) "mongodb"
3) "redis"
```

In the above example, three values are inserted in Redis list named 'tutorials' by the command **LPUSH**.

Redis Lists Commands

Following table lists some basic commands related to lists.

Sr. No.	Command & Description
1	**BLPOP key1 [key2] timeout** Removes and gets the first element in a list, or blocks until one is available
2	**BRPOP key1 [key2] timeout** Removes and gets the last element in a list, or blocks until one is available
3	**BRPOPLPUSH source destination timeout** Pops a value from a list, pushes it to another list and returns it; or blocks until one is available

4	**LINDEX key index** Gets an element from a list by its index
5	**LINSERT key BEFORE|AFTER pivot value** Inserts an element before or after another element in a list
6	**LLEN key** Gets the length of a list
7	**LPOP key** Removes and gets the first element in a list
8	**LPUSH key value1 [value2]** Prepends one or multiple values to a list
9	**LPUSHX key value** Prepends a value to a list, only if the list exists
10	**LRANGE key start stop** Gets a range of elements from a list
11	**LREM key count value** Removes elements from a list
12	**LSET key index value** Sets the value of an element in a list by its index
13	**LTRIM key start stop** Trims a list to the specified range
14	**RPOP key** Removes and gets the last element in a list
15	**RPOPLPUSH source destination** Removes the last element in a list, appends it to another list and returns it
16	**RPUSH key value1 [value2]** Appends one or multiple values to a list
17	**RPUSHX key value** Appends a value to a list, only if the list exists

List Blpop Command

Redis **BLPOP** command is used to remove and get the first element in a list, or block until one is available. **BLPOP** command just returns the first element, if available, or blocks the client for specific time to execute any command.

Return Value

String reply, the value of element stored at the key or nil.

Syntax

Following is the basic syntax of Redis **BLPOP** command.

```
redis 127.0.0.1:6379> BLPOP LIST1 LIST2 .. LISTN TIMEOUT
```

Example

```
redis 127.0.0.1:6379> BLPOP list1 100
```

The above example will block the client for 100 seconds to execute any command. If any data comes in the specified key list1, then it returns; otherwise after 100 seconds nil value is returned.

```
(nil)
(100.06s)
```

List Brpop Command

Redis **BRPOP** command is used to remove and get the last element in a list, or block until one is available. **BRPOP** command just returns the last element, if available, or blocks the client for specific time to execute any command.

Return Value

String reply, the value of element stored at the key or nil.

Syntax

Following is the basic syntax of Redis **BRPOP** command.

```
redis 127.0.0.1:6379> BRPOP LIST1 LIST2 .. LISTN TIMEOUT
```

Example

```
redis 127.0.0.1:6379> BRPOP list1 100
```

The above example will block the client for 100 seconds to execute any command. If any data comes in the specified key list1, then it returns; otherwise after 100 seconds nil value is returned.

```
(nil)
(100.06s)
```

List Brpoplpush Command

Redis **BRPOPLPUSH** command is used to pop a value from a list, push it to another list and return it, or block until one is available. **BRPOPLPUSH** command just returns the last element and inserts it into another list, if available, or blocks the client for specific time to execute any command.

Return Value

String reply, the value of element stored at the key or nil.

Syntax

Following is the basic syntax of Redis **BRPOPLPUSH** command.

```
redis 127.0.0.1:6379> BRPOPLPUSH LIST1 ANOTHER_LIST TIMEOUT
```

Example

```
redis 127.0.0.1:6379> BRPOPLPUSH list1 list2 100
```

The above example will block the client for 100 seconds to execute any command. If any data comes in the specified key list1, then it will pop data and push it into another list; otherwise after 100 seconds nil value is returned.

```
(nil)
(100.06s)
```

List Lindex Command

Redis **LINDEX** command is used to get the element at the index in the list stored at the key. The index is zero-based, so 0 means the first element, 1 the second element, and so on. Negative indices can be used to designate elements starting at the tail of the list. Here, -1 means the last element, -2 means the penultimate, and so forth.

Return Value

String reply, the requested element, or nil when the index is out of range.

Syntax

Following is the basic syntax of Redis **LINDEX** command.

```
redis 127.0.0.1:6379> LINDEX KEY_NAME INDEX_POSITION
```

Example

```
redis 127.0.0.1:6379> LPUSH list1 "foo"
(integer) 1
redis 127.0.0.1:6379> LPUSH list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LINDEX list1 0
"foo"
redis 127.0.0.1:6379> LINDEX list1 -1
"bar"
redis 127.0.0.1:6379> LINDEX list1 5
nil
```

List Linsert Command

Redis **LINSERT** command inserts the value in the list stored at the key either before or after the reference value pivot. When the key does not exist, it is considered an empty list and no operation is performed. An error is returned when the key exists but does not hold a list value.

Return Value

Integer reply, the length of the list after the insert operation, or -1 when the value pivot was not found.

Syntax

Following is the basic syntax of Redis **LINSERT** command.

```
redis 127.0.0.1:6379> LINSERT KEY_NAME BEFORE EXISTING_VALUE NEW_VALUE
```

Example

```
redis 127.0.0.1:6379> RPUSH list1 "foo"
(integer) 1
redis 127.0.0.1:6379> RPUSH list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LINSERT list1 BEFORE "bar" "Yes"
(integer) 3
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1) "foo"
2) "Yes"
3) "bar"
```

List Llen Command

Redis **LLEN** command returns the length of the list stored at the key. If the key does not exist, it is interpreted as an empty list and 0 is returned. An error is returned when the value stored at the key is not a list.

Return Value

Integer reply, the length of the list at the key.

Syntax

Following is the basic syntax of Redis **LLEN** command.

```
redis 127.0.0.1:6379> LLEN KEY_NAME
```

Example

```
redis 127.0.0.1:6379> RPUSH list1 "foo"
(integer) 1
redis 127.0.0.1:6379> RPUSH list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LLEN list1
(integer) 2
```

List Lpop Command

Redis **LPOP** command removes and returns the first element of the list stored at the key.

Return Value

String reply, the value of the first element, or nil when the key does not exist.

Syntax

Following is the basic syntax of Redis **LPOP** command.

```
redis 127.0.0.1:6379> LPOP KEY_NAME
```

Example

```
redis 127.0.0.1:6379> RPUSH list1 "foo"
(integer) 1
redis 127.0.0.1:6379> RPUSH list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LPOP list1
```

```
"foo"
```

List Lpush Command

Redis **LPUSH** command inserts all the specified values at the head of the list stored at the key. If the key does not exist, it is created as an empty list before performing the push operations. When the key holds a value that is not a list, an error is returned.

Return Value

Integer reply, the length of the list after the push operations.

Syntax

Following is the basic syntax of Redis **LPUSH** command.

```
redis 127.0.0.1:6379> LPUSH KEY_NAME VALUE1.. VALUEN
```

Example

```
redis 127.0.0.1:6379> LPUSH list1 "foo"
(integer) 1
redis 127.0.0.1:6379> LPUSH list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LRANGE list1 0 -1
1) "foo"
2) "bar"
```

List Lpushx Command

Redis **LPUSHX** command inserts the value at the head of the list stored at the key, only if the key already exists and holds a list.

Return Value

Integer reply, the length of the list after the push operations.

Syntax

Following is the basic syntax of Redis **LPUSHX** command.

```
redis 127.0.0.1:6379> LPUSHX KEY_NAME VALUE1.. VALUEN
```

Example

```
redis 127.0.0.1:6379> LPUSH list1 "foo"
```

```
(integer) 1
redis 127.0.0.1:6379> LPUSHX list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LPUSHX list2 "bar"
(integer) 0
redis 127.0.0.1:6379> LRANGE list1 0 -1
1) "foo"
2) "bar"
```

List Lrange Command

Redis **LRANGE** command returns the specified elements of the list stored at the key. The offsets start and stop are zero-based indexes, with 0 being the first element of the list (the head of the list), 1 being the next element, and so on. These offsets can also be negative numbers indicating offsets starting at the end of the list. For example, -1 is the last element of the list, -2 the penultimate, and so on.

Return Value

Array reply, list of elements in the specified range.

Syntax

Following is the basic syntax of Redis **LRANGE** command.

```
redis 127.0.0.1:6379> LRANGE KEY_NAME START END
```

Example

```
redis 127.0.0.1:6379> LPUSH list1 "foo"
(integer) 1
redis 127.0.0.1:6379> LPUSH list1 "bar"
(integer) 2
redis 127.0.0.1:6379> LPUSHX list1 "bar"
(integer) 0
redis 127.0.0.1:6379> LRANGE list1 0 -1
1) "foo"
2) "bar"
3) "bar"
```

List Lrem Command

Redis **LREM** command removes the first count occurrences of elements equal to the value from the list stored at the key. The count argument influences the operation in the following ways:

- **count > 0**: Removes the elements equal to the value moving from the head to tail.
- **count < 0**: Removes the elements equal to the value moving from the tail to head.
- **count =** 0: Removes all elements equal to value.

Return Value

Integer reply, the number of removed elements.

Syntax

Following is the basic syntax of Redis **LREM** command.

```
redis 127.0.0.1:6379> LREM KEY_NAME COUNT VALUE
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 2
redis 127.0.0.1:6379> RPUSH mylist "foo"
(integer) 3
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 4
redis 127.0.0.1:6379> LREM mylist -2 "hello"
(integer) 2
```

List Lset Command

Redis **LSET** command sets the list element at an index to the value. For more information on the index argument, see LINDEX. An error is returned for out of range indexes.

Return Value

String reply, OK.

Syntax

Following is the basic syntax of Redis **LSET** command.

```
redis 127.0.0.1:6379> LSET KEY_NAME INDEX VALUE
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 2
redis 127.0.0.1:6379> RPUSH mylist "foo"
(integer) 3
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 4
redis 127.0.0.1:6379> LSET mylist 0 "bar"
OK
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1: "bar"
2) "hello"
3) "foo"
4) "hello"
```

List Ltrim Command

Redis **LTRIM** command trims an existing list so that it contains only the specified range of elements. Both start and stop are zero-based indexes, where 0 is the first element of the list (the head), 1 the next element, and so on.

Return Value

String reply, OK.

Syntax

Following is the basic syntax of redis **LTRIM** command.

```
redis 127.0.0.1:6379> LTRIM KEY_NAME START STOP
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 2
redis 127.0.0.1:6379> RPUSH mylist "foo"
(integer) 3
redis 127.0.0.1:6379> RPUSH mylist "bar"
(integer) 4
redis 127.0.0.1:6379> LTRIM mylist 1 -1
OK
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "foo"
3) "bar"
```

List Rpop Command

Redis **RPOP** command removes and returns the last element of the list stored at the key.

Return Value

String reply, the value of the last element, or nil when the key does not exist.

Syntax

Following is the basic syntax of Redis **RPOP** command.

```
redis 127.0.0.1:6379> RPOP KEY_NAME
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 2
redis 127.0.0.1:6379> RPUSH mylist "foo"
```

```
(integer) 3
redis 127.0.0.1:6379> RPUSH mylist "bar"
(integer) 4
redis 127.0.0.1:6379> RPOP mylist
OK
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "hello"
3) "foo"
```

List Rpoplpush Command

Redis **RPOPLPUSH** command returns and removes the last element (tail) of the list stored at the source, and pushes the element at the first element (head) of the list stored at the destination.

Return Value

String reply, the element being popped and pushed.

Syntax

Following is the basic syntax of Redis **RPOPLPUSH** command.

```
redis 127.0.0.1:6379> RPOPLPUSH SOURCE_KEY_NAME DESTINATION_KEY_NAME
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "foo"
(integer) 2
redis 127.0.0.1:6379> RPUSH mylist "bar"
(integer) 3
redis 127.0.0.1:6379> RPOPLPUSH mylist myotherlist
"bar"
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "foo"
```

List Rpush Command

Redis **RPUSH** command inserts all the specified values at the tail of the list stored at the key. If the key does not exist, it is created as an empty list before performing the push operation. When the key holds a value that is not a list, an error is returned.

Return Value

Integer reply, the length of the list after the push operation.

Syntax

Following is the basic syntax of Redis **RPUSH** command.

```
redis 127.0.0.1:6379> RPUSH KEY_NAME VALUE1..VALUEN
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "foo"
(integer) 2
redis 127.0.0.1:6379> RPUSH mylist "bar"
(integer) 3
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "foo"
3) "bar"
```

List Rpushx Command

Redis **RPUSHX** command inserts the value at the tail of the list stored at the key, only if the key already exists and holds a list. In contrary to RPUSH, no operation will be performed when the key does not yet exist.

Return Value

Integer reply, the length of the list after the push operation.

Syntax

Following is the basic syntax of Redis **RPUSHX** command.

```
redis 127.0.0.1:6379> RPUSHX KEY_NAME VALUE1..VALUEN
```

Example

```
redis 127.0.0.1:6379> RPUSH mylist "hello"
(integer) 1
redis 127.0.0.1:6379> RPUSH mylist "foo"
(integer) 2
redis 127.0.0.1:6379> RPUSHX mylist2 "bar"
(integer) 0
redis 127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "foo"
```

10. Redis – Sets

Redis Sets are an unordered collection of unique strings. Unique means sets does not allow repition of data in a key.

In Redis set add, remove, and test for the existence of members in O(1) (constant time regardless of the number of elements contained inside the Set). The maximum length of a list is 2^{32} - 1 elements (4294967295, more than 4 billion of elements per set).

Example

```
redis 127.0.0.1:6379> SADD tutorials redis
(integer) 1
redis 127.0.0.1:6379> SADD tutorials mongodb
(integer) 1
redis 127.0.0.1:6379> SADD tutorials mysql
(integer) 1
redis 127.0.0.1:6379> SADD tutorials mysql
(integer) 0
redis 127.0.0.1:6379> SMEMBERS tutorials

1) "mysql"
2) "mongodb"
3) "redis"
```

In the above example, three values are inserted in Redis set named 'tutorials' by the command **SADD**.

Redis Sets Commands

Following table lists some basic commands related to sets.

Sr. No.	Command & Description
1	**SADD key member1 [member2]** Adds one or more members to a set
2	**SCARD key** Gets the number of members in a set
3	**SDIFF key1 [key2]**

	Subtracts multiple sets
4	**SDIFFSTORE destination key1 [key2]** Subtracts multiple sets and stores the resulting set in a key
5	**SINTER key1 [key2]** Intersects multiple sets
6	**SINTERSTORE destination key1 [key2]** Intersects multiple sets and stores the resulting set in a key
7	**SISMEMBER key member** Determines if a given value is a member of a set
8	**SMEMBERS key** Gets all the members in a set
9	**SMOVE source destination member** Moves a member from one set to another
10	**SPOP key** Removes and returns a random member from a set
11	**SRANDMEMBER key [count]** Gets one or multiple random members from a set
12	**SREM key member1 [member2]** Removes one or more members from a set
13	**SUNION key1 [key2]** Adds multiple sets
14	**SUNIONSTORE destination key1 [key2]** Adds multiple sets and stores the resulting set in a key
15	**SSCAN key cursor [MATCH pattern] [COUNT count]** Incrementally iterates set elements

Set Sadd Command

Redis **SADD** command is used to add members to a set stored at the key. If the member already exists, then it is ignored. If the key does not exist, then a new set is created and the members are added into it. If the value stored at the key is not set, then an error is returned.

Return Value

Integer reply, the number of elements that were added to the set, not including all the elements already present in the set.

Syntax

Following is the basic syntax of Redis **SADD** command.

```
redis 127.0.0.1:6379> SADD KEY_NAME VALUE1..VALUEN
```

Example

```
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset "foo"
(integer) 1
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 0
redis 127.0.0.1:6379> SMEMBERS myset
1) "hello"
2) "foo"
```

Set Scard Comment

Redis **SCARD** command is used to return the number of elements stored in a set.

Return Value

Integer reply, the cardinality (number of elements) of the set, or 0 if the key does not exist.

Syntax

Following is the basic syntax of Redis **SCARD** command.

```
redis 127.0.0.1:6379> SCARD KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset "foo"
(integer) 1
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 0
redis 127.0.0.1:6379> SCARD myset
(integer) 2
```

Set Sdiff Command

Redis **SDIFF** command returns the members of the set resulting from the difference between the first set and all the successive sets. If the keys do not exist in Redis, then it it considered as empty sets.

Return Value

Array reply, list with members of the resulting set.

Syntax

Following is the basic syntax of Redis **SDIFF** command.

```
redis 127.0.0.1:6379> SDIFF FIRST_KEY OTHER_KEY1..OTHER_KEYN
```

Example

```
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset "foo"
(integer) 1
redis 127.0.0.1:6379> SADD myset "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "world"
(integer) 1
redis 127.0.0.1:6379> SDIFF myset myset2
1) "foo"
2) "bar"
```

Set Sdiffstore Command

Redis **SDIFFSTORE** command stores the members of the set, resulting from the difference between the first set and all the successive sets, into a set specified in the command. If the destination already exists, it is overwritten.

Return Value

Integer reply, the number of elements in the resulting set.

Syntax

Following is the basic syntax of Redis **SDIFFSTORE** command.

```
redis 127.0.0.1:6379> SDIFFSTORE DESTINATION_KEY KEY1..KEYN
```

Example

```
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset "foo"
(integer) 1
redis 127.0.0.1:6379> SADD myset "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "world"
(integer) 1
redis 127.0.0.1:6379> SDIFFSTORE destset myset myset2
(integer) 2
redis 127.0.0.1:6379> SMEMBERS destset
1) "foo"
2) "bar"
```

Set Sinter Command

Redis **SINTER** command gets the elements of a set after intersection of all specified sets. Keys that do not exist are considered to be empty sets. With one of the keys being an empty set, the resulting set is also empty (since set intersection with an empty set always results in an empty set).

Return Value

Array reply, list with members of the resulting set.

Syntax

Following is the basic syntax of Redis **SINTER** command.

```
redis 127.0.0.1:6379> SINTER KEY KEY1..KEYN
```

Example

```
redis 127.0.0.1:6379> SADD myset "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset "foo"
```

```
(integer) 1
redis 127.0.0.1:6379> SADD myset "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "world"
(integer) 1
redis 127.0.0.1:6379> SINTER myset myset2
1) "hello"
```

Set Sinterstore Command

Redis **SINTERSTORE** command stores the elements in a set after intersection of all specified sets. Keys that do not exist are considered to be empty sets. With one of the keys being an empty set, the resulting set is also empty (since set intersection with an empty set always results in an empty set).

Return Value

Integer reply, the number of elements in the resulting set.

Syntax

Following is the basic syntax of Redis **SINTERSTORE** command.

```
redis 127.0.0.1:6379> SINTERSTORE DESTINATION_KEY KEY KEY1..KEYN
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "foo"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "world"
(integer) 1
redis 127.0.0.1:6379> SINTERSTORE myset myset1 myset2
(integer) 1
redis 127.0.0.1:6379> SMEMBERS myset
```

```
1) "hello"
```

Set Sismember Command

Redis **SISMEMBER** returns an element that already exists in the set stored at the key or not.

Return Value

Integer reply

- 1, if the element is a member of the set.
- 0, if the element is not a member of the set, or if the key does not exist.

Syntax

Following is the basic syntax of Redis **SISMEMBER** command.

```
redis 127.0.0.1:6379> SISMEMBER KEY VALUE
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SISMEMBER myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SISMEMBER myset1 "world"
(integer) 0
```

Redis – Set Sismember Command

Redis **SISMEMBER** returns an element that already exists in the set stored at the key or not.

Return Value

Integer reply

- 1, if the element is a member of the set.
- 0, if the element is not a member of the set, or if the key does not exist.

Syntax

Following is the basic syntax of Redis **SISMEMBER** command.

```
redis 127.0.0.1:6379> SISMEMBER KEY VALUE
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SISMEMBER myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SISMEMBER myset1 "world"
(integer) 0
```

Set Smove Command

Redis **SMOVE** command is used to move an element of a set from one key to another key. If the source set does not exist or does not contain the specified element, no operation is performed and 0 is returned. Otherwise, the element is removed from the source set and added to the destination set. When the specified element already exists in the destination set, it is only removed from the source set. An error is returned, if the source or destination does not hold a set value.

Return Value

Integer reply.

- 1, if the element is moved.
- 0, if the element is not a member of the source and no operation was performed.

Syntax

Following is the basic syntax of Redis **SMOVE** command.

```
redis 127.0.0.1:6379> SMOVE SOURCE DESTINATION MEMBER
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "world"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "foo"
(integer) 1
```

```
redis 127.0.0.1:6379> SMOVE myset1 myset2 "bar"
(integer) 1
redis 127.0.0.1:6379> SMEMBERS myset1
1) "World"
2) "Hello"
redis 127.0.0.1:6379> SMEMBERS myset2
1) "foo"
2) "bar"
```

Set Spop Command

Redis **SPOP** command is used to remove and return a random member from the set stored at a specified key.

Return Value

String reply, the removed element, or nil when the key does not exist.

Syntax

Following is the basic syntax of Redis **SPOP** command.

```
redis 127.0.0.1:6379> SPOP KEY
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "world"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SPOP myset1
"bar"
redis 127.0.0.1:6379> SMEMBERS myset1
1) "Hello"
2) "world"
```

Set Srandmember Command

Redis **SRANDMEMBER** command is used to get a random member from the set stored at a specified key. If called with the additional count argument, it returns an array of count distinct elements, when the count is positive. If called with a negative count, the behavior changes and the command is allowed to return the same element multiple times. In this case, the numer of returned elements is the absolute value of the specified count.

Return Value

String reply, without the additional count argument. The command returns a Bulk Reply with the randomly selected element, or nil when the key does not exist. Array reply, when the additional count argument is passed the command returns an array of elements, or an empty array when the key does not exist.

Syntax

Following is the basic syntax of Redis **SRANDMEMBER** command.

```
redis 127.0.0.1:6379> SRANDMEMBER KEY [count]
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "world"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SRANDMEMBER myset1
"bar"
redis 127.0.0.1:6379> SRANDMEMBER myset1 2
1) "Hello"
2) "world"
```

Set Srem Command

Redis **SREM** command is used to remove the specified member from the set stored at the key. If the member does not exist, then the command returns 0. If the stored value at the key is not set, then an error is returned.

Return Value

Integer reply, the number of members that were removed from the set, not including non-existing members.

Syntax

Following is the basic syntax of Redis **SREM** command.

```
redis 127.0.0.1:6379> SREM KEY MEMBER1..MEMBERN
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "world"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SREM myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SREM myset1 "foo"
(integer) 0
redis 127.0.0.1:6379> SMEMBERS myset1
1) "bar"
2) "world"
```

Set Sunion Command

Redis **SUNION** command is used to get the members of the set resulting from the union of all the given sets. Keys that do not exist are considered to be empty sets.

Return Value

Array reply, list with members of the resulting set.

Syntax

Following is the basic syntax of Redis **SUNION** command.

```
redis 127.0.0.1:6379> SUNION KEY KEY1..KEYN
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "world"
(integer) 1
```

```
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "bar"
(integer) 1
redis 127.0.0.1:6379> SUNION myset1 myset2
1) "bar"
2) "world"
3) "hello"
4) "foo"
```

Set Sunionstore Command

Redis **SUNIONSTORE** command is used to store the members of the set resulting from the union of all the given sets. Keys that do not exist are considered to be empty sets.

Return Value

Integer reply, the number of elements in the resulting set.

Syntax

Following is the basic syntax of Redis **SUNIONSTORE** command.

```
redis 127.0.0.1:6379> SUNIONSTORE DESTINATION KEY KEY1..KEYN
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "world"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset2 "bar"
(integer) 1
redis 127.0.0.1:6379> SUNIONSTORE myset myset1 myset2
(integer) 1
```

```
redis 127.0.0.1:6379> SMEMBERS myset
1) "bar"
2) "world"
3) "hello"
4) "foo"
```

Set Sscan Command

Redis **SSCAN** command iterates the elements of a set stored at a specified key.

Return Value

Array reply.

Syntax

Following is the basic syntax of Redis **SSCAN** command.

```
redis 127.0.0.1:6379> SSCAN KEY [MATCH pattern] [COUNT count]
```

Example

```
redis 127.0.0.1:6379> SADD myset1 "hello"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "hi"
(integer) 1
redis 127.0.0.1:6379> SADD myset1 "bar"
(integer) 1
redis 127.0.0.1:6379> sscan myset1 0 match h*
1) "0"
2) 1) "hello"
   2) "h1"
```

11. Redis – Sorted Sets

Redis Sorted Sets are similar to Redis Sets with the unique feature of values stored in a set. The difference is, every member of a Sorted Set is associated with a score, that is used in order to take the sorted set ordered, from the smallest to the greatest score.

In Redis sorted set, add, remove, and test for the existence of members in O(1) (constant time regardless of the number of elements contained inside the set). Maximum length of a list is 2^{32} - 1 elements (4294967295, more than 4 billion of elements per set).

Example

```
redis 127.0.0.1:6379> ZADD tutorials 1 redis
(integer) 1
redis 127.0.0.1:6379> ZADD tutorials 2 mongodb
(integer) 1
redis 127.0.0.1:6379> ZADD tutorials 3 mysql
(integer) 1
redis 127.0.0.1:6379> ZADD tutorials 3 mysql
(integer) 0
redis 127.0.0.1:6379> ZADD tutorials 4 mysql
(integer) 0
redis 127.0.0.1:6379> ZRANGE tutorials 0 10 WITHSCORES

1) "redis"
2) "1"
3) "mongodb"
4) "2"
5) "mysql"
6) "4"
```

In the above example, three values are inserted with its score in Redis sorted set named ‘tutorials’ by the command **ZADD**.

Redis Sorted Sets Commands

Following table lists some basic commands related to sorted sets.

Sr. No.	Command & Description
1	**ZADD key score1 member1 [score2 member2]** Adds one or more members to a sorted set, or updates its score, if it already exists
2	**ZCARD key** Gets the number of members in a sorted set
3	**ZCOUNT key min max** Counts the members in a sorted set with scores within the given values
4	**ZINCRBY key increment member** Increments the score of a member in a sorted set
5	**ZINTERSTORE destination numkeys key [key ...]** Intersects multiple sorted sets and stores the resulting sorted set in a new key
6	**ZLEXCOUNT key min max** Counts the number of members in a sorted set between a given lexicographical range
7	**ZRANGE key start stop [WITHSCORES]** Returns a range of members in a sorted set, by index
8	**ZRANGEBYLEX key min max [LIMIT offset count]** Returns a range of members in a sorted set, by lexicographical range
9	**ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]** Returns a range of members in a sorted set, by score
10	**ZRANK key member** Determines the index of a member in a sorted set
11	**ZREM key member [member ...]** Removes one or more members from a sorted set
12	**ZREMRANGEBYLEX key min max** Removes all members in a sorted set between the given lexicographical range
13	**ZREMRANGEBYRANK key start stop** Removes all members in a sorted set within the given indexes

14	**ZREMRANGEBYSCORE key min max** Removes all members in a sorted set within the given scores
15	**ZREVRANGE key start stop [WITHSCORES]** Returns a range of members in a sorted set, by index, with scores ordered from high to low
16	**ZREVRANGEBYSCORE key max min [WITHSCORES]** Returns a range of members in a sorted set, by score, with scores ordered from high to low
17	**ZREVRANK key member** Determines the index of a member in a sorted set, with scores ordered from high to low
18	**ZSCORE key member** Gets the score associated with the given member in a sorted set
19	**ZUNIONSTORE destination numkeys key [key ...]** Adds multiple sorted sets and stores the resulting sorted set in a new key
20	**ZSCAN key cursor [MATCH pattern] [COUNT count]** Incrementally iterates sorted sets elements and associated scores

Sorted Set Zadd Command

Redis **ZADD** command adds all the specified members with the specified scores to the sorted set stored at the key. It is possible to specify multiple score/member pairs. If a specified member is already a member of the sorted set, the score is updated and the element is reinserted at the right position to ensure correct ordering. If the key does not exist, a new sorted set with the specified members as sole members is created, like if the sorted set was empty. If the key exists but does not hold a sorted set, an error is returned.

Return Value

Integer reply. The number of elements added to the sorted sets, not including elements already existing for which the score was updated.

Syntax

Following is the basic syntax of Redis **ZADD** command.

```
redis 127.0.0.1:6379> ZADD KEY_NAME SCORE1 VALUE1.. SCOREN VALUEN
```

Example

```
redis 127.0.0.1:6379> ZADD myset 1 "hello"
(integer) 1
```

```
redis 127.0.0.1:6379> ZADD myset 1 "foo"
(integer) 1
redis 127.0.0.1:6379> ZADD myset 2 "world" 3 "bar"
(integer) 2
redis 127.0.0.1:6379> ZRANGE myzset 0 -1 WITHSCORES
1) "hello"
2) "1"
3) "foo"
4) "1"
5) "world"
6) "2"
7) "bar"
8) "3"
```

Sorted Set Zcard Comment

Redis **ZCARD** command returns the number of elements stored in a set at a specified key.

Return Value

Integer reply, the cardinality (number of elements) of the sorted set, or 0 if the key does not exist.

Syntax

Following is the basic syntax of Redis **ZCARD** command.

```
redis 127.0.0.1:6379> ZCARD KEY_NAME
```

Example

```
redis 127.0.0.1:6379> ZADD myset 1 "hello"
(integer) 1
redis 127.0.0.1:6379> ZADD myset 1 "foo"
(integer) 1
redis 127.0.0.1:6379> ZADD myset 2 "world" 3 "bar"
(integer) 2
redis 127.0.0.1:6379> ZCARD myzset
(integer) 4
```

Sorted Set Zcount Command

Redis **ZCOUNT** command returns the number of elements in the sorted set at the key with a score between min and max.

Return Value

Integer reply, the number of elements in the specified score range.

Syntax

Following is the basic syntax of Redis **ZCOUNT** command.

```
redis 127.0.0.1:6379> ZCOUNT KEY_NAME
```

Example

```
redis 127.0.0.1:6379> ZADD myset 1 "hello"
(integer) 1
redis 127.0.0.1:6379> ZADD myset 1 "foo"
(integer) 1
redis 127.0.0.1:6379> ZADD myset 2 "world" 3 "bar"
(integer) 2
redis 127.0.0.1:6379> ZCOUNT myzset (1 3
(integer) 2
```

Sorted Set Zincrby Command

Redis **ZINCRBY** command increments the score of member in the sorted set stored at the key by increment. If the member does not exist in the sorted set, it is added with increment as its score (as if its previous score was 0.0). If the key does not exist, a new sorted set with the specified member as its sole member is created. An error is returned when the key exists but does not hold a sorted set.

Return Value

String reply, the new score of the member (a double precision floating point number), represented as string.

Syntax

Following is the basic syntax of Redis **ZINCRBY** command.

```
redis 127.0.0.1:6379> ZINCRBY KEY INCREMENT MEMBER
```

Example

```
redis 127.0.0.1:6379> ZADD myset 1 "hello"
```

```
(integer) 1
redis 127.0.0.1:6379> ZADD myset 1 "foo"
(integer) 1
redis 127.0.0.1:6379> ZINCRBY myzset 2 "hello"
(integer) 3
redis 127.0.0.1:6379> ZRANGE myzset 0 -1 WITHSCORES
1) "foo"
2) "2"
3) "hello"
4) "3"
```

Sorted Set Zinterstore Command

Redis **ZINTERSTORE** command computes the intersection of numkeys sorted sets given by the specified keys, and stores the result in the destination. It is mandatory to provide the number of input keys (numkeys) before passing the input keys and the other (optional) arguments.

Return Value

Integer reply, the number of elements in the resulting sorted set at the destination.

Syntax

Following is the basic syntax of Redis **ZINTERSTORE** command.

```
redis 127.0.0.1:6379> ZINTERSTORE KEY INCREMENT MEMBER
```

Example

```
redis 127.0.0.1:6379> ZADD myset 1 "hello"
(integer) 1
redis 127.0.0.1:6379> ZADD myset 2 "world"
(integer) 1
redis 127.0.0.1:6379> ZADD myset2 1 "hello"
(integer) 1
redis 127.0.0.1:6379> ZADD myset2 2 "world"
(integer) 1
redis 127.0.0.1:6379> ZADD myset2 3 "foo"
(integer) 1
redis 127.0.0.1:6379> ZINTERSTORE out 2 myset1 myset2 WEIGHTS 2 3"
(integer) 3
```

```
redis 127.0.0.1:6379> ZRANGE out 0 -1 WITHSCORES
1) "hello"
2) "5"
3) "world"
4) "10"
```

Sorted Set Zlexcount Command

When all the elements in a sorted set are inserted with the same score, in order to force lexicographical ordering, this command returns the number of elements in the sorted set at the key with a value between min and max.

Return Value

Integer reply, the number of elements in the specified score range.

Syntax

Following is the basic syntax of Redis **ZLEXCOUNT** command.

```
redis 127.0.0.1:6379> ZLEXCOUNT KEY MIN MAX
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 0 b 0 c 0 d 0 e
(integer) 5
redis 127.0.0.1:6379> ZADD myzset 0 f 0 g
(integer) 2
redis 127.0.0.1:6379> ZLEXCOUNT myzset - +
(integer) 7
redis 127.0.0.1:6379> ZLEXCOUNT myzset [b [f
(integer) 5
```

Sorted Set Zrange Command

Redis **ZRANGE** command returns the specified range of elements in the sorted set stored at the key. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with an equal score. Both start and stop are zero-based indexes, where 0 is the first element, 1 is the next element and so on. They can also be negative numbers indicating offsets from the end of the sorted set, with -1 being the last element of the sorted set, -2 the penultimate element and so on.

Return Value

Array reply, list of elements in the specified range (optionally with their scores).

Syntax

Following is the basic syntax of Redis **ZRANGE** command.

```
redis 127.0.0.1:6379> ZRANGE KEY START STOP [WITHSCORES]
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 0 b 0 c 0 d 0 e
(integer) 5
redis 127.0.0.1:6379> ZADD myzset 0 f 0 g
(integer) 2
redis 127.0.0.1:6379> ZRANGE myzset 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
7) "g"
redis 127.0.0.1:6379> ZLEXCOUNT myzset [b [f
(integer) 5
```

Sorted Set Zrangebylex Command

Redis **ZRANGEBYLEX** command returns the specified range of elements in the sorted set stored at the key. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with an equal score. Both start and stop are zero-based indexes, where 0 is the first element, 1 is the next element, and so on. They can also be negative numbers indicating offsets from the end of the sorted set, with -1 being the last element of the sorted set, -2 the penultimate element, and so on.

Return Value

Array reply, list of elements in the specified score range.

Syntax

Following is the basic syntax of Redis **ZRANGEBYLEX** command.

```
redis 127.0.0.1:6379> ZRANGEBYLEX key min max [LIMIT offset count]
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 0 b 0 c 0 d 0 e
(integer) 5
redis 127.0.0.1:6379> ZADD myzset 0 f 0 g
(integer) 2
redis 127.0.0.1:6379> ZRANGEBYLEX myzset - [c
1) "a"
2) "b"
3) "c"
redis 127.0.0.1:6379> ZRANGEBYLEX myzset - (c
1) "a"
2) "b"
```

Sorted Set Zrangebyscore Command

Redis **ZRANGEBYSCORE** command returns all the elements in the sorted set at the key with a score between min and max (including elements with score equal to min or max). The elements are considered to be ordered from low to high scores. The elements having the same score are returned in lexicographical order (this follows from a property of the sorted set implementation in Redis and does not involve further computation).

Return Value

Array reply, list of elements in the specified score range (optionally with their scores).

Syntax

Following is the basic syntax of Redis **ZRANGEBYSCORE** command.

```
redis 127.0.0.1:6379> ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset
count]
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 1 b 2 c 3 d 4 e
(integer) 5
redis 127.0.0.1:6379> ZADD myzset 5 f 6 g
(integer) 2
redis 127.0.0.1:6379> ZRANGEBYSCORE myzset 1 2
1) "b"
2) "c"
redis 127.0.0.1:6379> ZRANGEBYSCORE myzset (1 2
1) "b"
```

Sorted Set Zrank Command

Redis **ZRANK** command returns the rank of member in the sorted set stored at the key, with the scores ordered from low to high. The rank (or index) is 0-based, which means that the member with the lowest score has rank 0.

Return Value

- If the member exists in the sorted set, Integer reply: the rank of member.
- If the member does not exist in the sorted set or the key does not exist, Bulk string reply: nil.

Syntax

Following is the basic syntax of Redis **ZRANK** command.

```
redis 127.0.0.1:6379> ZRANK key member
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 1 b 2 c 3 d 4 e
(integer) 5
redis 127.0.0.1:6379> ZADD myzset 5 f 6 g
(integer) 2
redis 127.0.0.1:6379> ZRANK myzset b
(integer) 1
redis 127.0.0.1:6379> ZRANK myzset t
nil
```

Sorted Set Zrem Command

Redis **ZREM** command removes the specified members from the sorted set stored at the key. Non-existing members are ignored. An error is returned when the key exists and does not hold a sorted set.

Return Value

Integer, number of members removed from the sorted set, not including non-existing members.

Syntax

Following is the basic syntax of Redis **ZREM** command.

```
redis 127.0.0.1:6379> ZREM key member [member ...]
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 1 b 2 c 3 d 4 e
(integer) 5
redis 127.0.0.1:6379> ZREM myzset b
(integer) 1
redis 127.0.0.1:6379> ZRANGE myzset 0 -1 WITHSCORES
1) "a"
2) "0"
3) "c"
4) "2"
5) "d"
6) "3"
7) "e"
8) "4"
```

Sorted Set Zremrangebylex Command

Redis **ZREMRANGEBYLEX** command removes all the elements in the sorted set stored at the key between the lexicographical range specified by min and max.

Return Value

Integer reply, the number of elements removed.

Syntax

Following is the basic syntax of Redis **ZREMRANGEBYLEX** command.

```
redis 127.0.0.1:6379> ZREMRANGEBYLEX key min max
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 0 a 1 b 2 c 3 d 4 e
(integer) 5
redis 127.0.0.1:6379> ZREMRANGEBYLEX myzset 0 -1
(integer) 1
redis 127.0.0.1:6379> ZRANGE myzset 0 -1 WITHSCORES
```

Sorted Set Zremrangebyrank Command

Redis **ZREMRANGEBYRANK** command removes all elements in the sorted set stored at the key with the rank between start and stop. Both start and stop are 0-based indexes with 0 being the element with the lowest score. These indexes can be negative numbers, where they indicate offsets starting at the element with the highest score. For example: -1 is the element with the highest score, -2 the element with the second highest score and so forth.

Return Value

Integer reply, the number of elements removed.

Syntax

Following is the basic syntax of Redis **ZREMRANGEBYRANK** command.

```
redis 127.0.0.1:6379> ZREMRANGEBYRANK key start stop
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 1 b 2 c 3 d 4 e
(integer) 4
redis 127.0.0.1:6379> ZREMRANGEBYRANK myzset 0 3
(integer) 3
redis 127.0.0.1:6379> ZRANGE myzset 0 -1 WITHSCORES
1) "e"
2) "4"
```

Sorted Set Zremrangebyscore Command

Redis **ZREMRANGEBYSCORE** command removes all the elements in the sorted set stored at the key with a score between min and max (inclusive).

Return Value

Integer reply, the number of elements removed.

Syntax

Following is the basic syntax of Redis **ZREMRANGEBYSCORE** command.

```
redis 127.0.0.1:6379> ZREMRANGEBYSCORE key min max
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 1 b 2 c 3 d 4 e
(integer) 4
redis 127.0.0.1:6379> ZREMRANGEBYSCORE myzset -inf (2
(integer) 1
redis 127.0.0.1:6379> ZRANGE myzset 0 -1 WITHSCORES
1) "b"
2) "2"
3) "c"
4) "3"
5) "d"
6) "4"
7) "e"
8) "5"
```

Sorted Set Zrevrange Command

Redis **ZREVRANGE** command returns the specified range of elements in the sorted set stored at the key. The elements are considered to be ordered from the highest to the lowest score. Descending lexicographical order is used for elements with an equal score.

Return Value

Array reply, list of elements in the specified range (optionally with their scores).

Syntax

Following is the basic syntax of Redis **ZREVRANGE** command.

```
redis 127.0.0.1:6379> ZREVRANGE key min max
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 1 b 2 c 3 d 4 e
(integer) 4
redis 127.0.0.1:6379> ZREVRANGE myzset 0 -1
1) "e"
2) "d"
3) "c"
4) "b"
```

Sorted Set Zrevrangebyscore Command

Redis **ZREVRANGEBYSCORE** command returns all the elements in the sorted set at the key with a score between max and min (including elements with score equal to max or min). In contrary to the default ordering of sorted sets, for this command the elements are considered to be ordered from high to low scores. The elements having the same score are returned in a reverse lexicographical order.

Return Value

Array reply, list of elements in the specified score range (optionally with their scores).

Syntax

Following is the basic syntax of Redis **ZREVRANGEBYSCORE** command.

```
redis 127.0.0.1:6379> ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset
count]
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 1 b 2 c 3 d 4 e
(integer) 4
redis 127.0.0.1:6379> ZREVRANGEBYSCORE myzset +inf -inf
1) "e"
2) "d"
3) "c"
4) "b"
redis 127.0.0.1:6379> ZREVRANGEBYSCORE myzset 2 1
1) "c"
2) "b"
```

Sorted Set Zrevrank Command

Redis **ZREVRANK** command returns the rank of the member in the sorted set stored at the key, with the scores ordered from high to low. The rank (or index) is 0-based, which means that the member with the highest score has rank 0.

Return Value

- If the member exists in the sorted set, Integer reply: the rank of member.
- If the member does not exist in the sorted set or the key does not exist, Bulk string reply: nil.

Syntax

Following is the basic syntax of Redis **ZREVRANK** command.

```
redis 127.0.0.1:6379> ZREVRANK key member
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 1 b 2 c 3 d 4 e
(integer) 4
redis 127.0.0.1:6379> ZREVRANK myzset "c"
(integer) 3
redis 127.0.0.1:6379> ZREVRANK myzset "y"
(nil)
```

Sorted Set Zscore Command

Redis **ZSCORE** command returns the score of the member in the sorted set at the key. If the member does not exist in the sorted set, or the key does not exist, nil is returned.

Return Value

Bulk string reply, the score of member (a double precision floating point number), represented as string.

Syntax

Following is the basic syntax of Redis **ZSCORE** command.

```
redis 127.0.0.1:6379> ZSCORE key member
```

Example

```
redis 127.0.0.1:6379> ZADD myzset 1 b 2 c 3 d 4 e
(integer) 4
redis 127.0.0.1:6379> ZSCORE myzset "c"
(integer) 3
redis 127.0.0.1:6379> ZSCORE myzset "y"
(nil)
```

Sorted Set Zunionstore Command

Redis **ZUNIONSTORE** command calculates the union of numkeys sorted sets given by the specified keys, and stores the result in the destination. It is mandatory to provide the number of input keys (numkeys) before passing the input keys and the other (optional) arguments.

Return Value

Integer reply, the number of elements in the resulting sorted set at the destination.

Syntax

Following is the basic syntax of Redis **ZUNIONSTORE** command.

```
redis 127.0.0.1:6379> ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS
weight [weight ...]] [AGGREGATE SUM|MIN|MAX]
```

Example

```
redis 127.0.0.1:6379> ZADD myzset1 1 b 2 c
(integer) 2
redis 127.0.0.1:6379> ZADD myzset2 1 b 2 c 3 d
(integer) 3
redis 127.0.0.1:6379> ZUNIONSTORE out 2 myzset1 myzset2 WEIGHTS 2 3
(integer) 3
redis 127.0.0.1:6379> ZRANGE out 0 -1 WITHSCORES

1) "b"
2) "5"
3) "c"
4) "9"
5) "d"
6) "10"
```

Sorted Set Zscan Command

Redis **ZSCAN** command iterates the elements of Sorted Set types and their associated scores.

Return Value

Array reply.

Syntax

Following is the basic syntax of Redis **ZSCAN** command.

```
redis 127.0.0.1:6379> ZSCAN key cursor [MATCH pattern] [COUNT count]
```

12. Redis – HyperLogLog

Redis HyperLogLog is an algorithm that uses randomization in order to provide an approximation of the number of unique elements in a set using just a constant, and small amount of memory.

HyperLogLog provides a very good approximation of the cardinality of a set even using a very small amount of memory around 12 kbytes per key with a standard error of 0.81%. There is no limit to the number of items you can count, unless you approach 2^{64} items.

Example

Following example explains how Redis HyperLogLog works.

```
redis 127.0.0.1:6379> PFADD tutorials "redis"

1) (integer) 1

redis 127.0.0.1:6379> PFADD tutorials "mongodb"

1) (integer) 1

redis 127.0.0.1:6379> PFADD tutorials "mysql"

1) (integer) 1

redis 127.0.0.1:6379> PFCOUNT tutorials

(integer) 3
```

Redis HyperLogLog Commands

Following table lists some basic commands related to Redis HyperLogLog.

Sr. No.	Command & Description
1	**PFADD key element [element ...]** Adds the specified elements to the specified HyperLogLog.
2	**PFCOUNT key [key ...]** Returns the approximated cardinality of the set(s) observed by the HyperLogLog at key(s).

3	**PFMERGE destkey sourcekey [sourcekey ...]** Merges N different HyperLogLogs into a single one.

HyperLogLog Pfadd Command

Redis **PFADD** command adds all the element arguments to the HyperLogLog data structure stored at the key name specified as first argument.

Return Value

Integer reply, 1 or 0.

Syntax

Following is the basic syntax of Redis **PFADD** command.

```
redis 127.0.0.1:6379> PFADD KEY_NAME ELEMENTS_TO_BE_ADDED
```

Example

```
redis 127.0.0.1:6379> PFADD mykey a b c d e f g h i j
(integer) 1
redis 127.0.0.1:6379> PFCOUNT mykey
(integer) 10
```

HyperLogLog Pfcount Command

Redis **PFCOUNT** command is used to get the approximated cardinality computed by the HyperLogLog data structure stored at the specified variable. If the key does not exist, then it returns 0.

Return Value

Integer reply, approximated number of unique elements.

When **PFCOUNT** command is used with multiple keys, then it returns approximated cardinality of the union of the HyperLogLogs.

Syntax

Following is the basic syntax of Redis **PFCOUNT** command.

```
redis 127.0.0.1:6379> PFCOUNT KEY1 KEY@... KEYN
```

Example

```
redis 127.0.0.1:6379> PFADD mykey a b c d e f g h i j
```

```
(integer) 1
redis 127.0.0.1:6379> PFCOUNT mykey
(integer) 10
redis 127.0.0.1:6379> PFCOUNT mykey
(integer) 10
redis 127.0.0.1:6379> PFCOUNT mykey mynewkey
(integer) 10
```

HyperLogLog Pfmerge Command

Redis **PFMERGE** command is used to merge multiple HyperLogLog values into a unique value that will approximate the cardinality of the union of the observed sets of the source HyperLogLog structures.

Return Value

Simple string reply OK.

Syntax

Following is the basic syntax of Redis **PFMERGE** command.

```
redis 127.0.0.1:6379> PFMERGE KEY1 KEY@... KEYN
```

Example

```
redis 127.0.0.1:6379> PFADD hll1 foo bar zap a
(integer) 1
redis 127.0.0.1:6379> PFADD hll2 a b c foo
(integer) 1
redis 127.0.0.1:6379> PFMERGE hll3 hll1 hll2
OK
redis 127.0.0.1:6379> PFCOUNT hll3
(integer) 6
```

13. Redis – Publish Subscribe

Redis Pub/Sub implements the messaging system where the senders (in redis terminology called publishers) sends the messages while the receivers (subscribers) receive them. The link by which the messages are transferred is called **channel**.

In Redis, a client can subscribe any number of channels.

Example

Following example explains how publish subscriber concept works. In the following example, one client subscribes a channel named 'redisChat'.

```
redis 127.0.0.1:6379> SUBSCRIBE redisChat


Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
```

Now, two clients are publishing the messages on the same channel named 'redisChat' and the above subscribed client is receiving messages.

```
redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"


(integer) 1


redis 127.0.0.1:6379> PUBLISH redisChat "Learn redis by tutorials point"


(integer) 1



1) "message"
2) "redisChat"
3) "Redis is a great caching technique"
1) "message"
2) "redisChat"
3) "Learn redis by tutorials point"
```

Redis PubSub Commands

Following table lists some basic commands related to Redis Pub/Sub.

Sr. No.	Command & Description
1	**PSUBSCRIBE pattern [pattern ...]** Subscribes to channels matching the given patterns.
2	**PUBSUB subcommand [argument [argument ...]]** Tells the state of Pub/Sub system. For example, which clients are active on the server.
3	**PUBLISH channel message** Posts a message to a channel.
4	**PUNSUBSCRIBE [pattern [pattern ...]]** Stops listening for messages posted to channels matching the given patterns.
5	**SUBSCRIBE channel [channel ...]** Listens for messages published to the given channels.
6	**UNSUBSCRIBE [channel [channel ...]]** Stops listening for messages posted to the given channels.

PubSub Psubscribe Command

Redis **PSUBSCRIBE** command is used to subscribe to channels matching the given patterns.

Syntax

Following is the basic syntax of Redis **PSUBSCRIBE** command.

```
redis 127.0.0.1:6379> PSUBSCRIBE CHANNEL_NAME_OR_PATTERN [PATTERN...]
```

Following listing shows some supported patterns in Redis.

- h?llo subscribes to hello, hallo and hxllo
- h*llo subscribes to hllo and heeeello
- h[ae]llo subscribes to hello and hallo, but not hillo

Example

```
redis 127.0.0.1:6379> PSUBSCRIBE mychannel
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
```

```
2) "mychannel"
3) (integer) 1
```

Redis - PubSub SubCommand

Redis **PUBSUB** command is an introspection command that allows to inspect the state of the Pub/Sub subsystem. It is composed of subcommands that are documented separately.

Syntax

Following is the basic syntax of Redis **PUBSUB** command.

```
redis 127.0.0.1:6379> PUBSUB subcommand [argument [argument ...]]
```

Return Value

Array reply, a list of active channels.

Example

```
redis 127.0.0.1:6379> PUBSUB CHANNELS
(empty list or set)
```

PubSub Publish Command

Redis **PUBLISH** command posts a message to a channel.

Syntax

Following is the basic syntax of Redis **PUBLISH** command.

```
redis 127.0.0.1:6379> PUBLISH channel message
```

Return Value

Integer reply: the number of clients that received the message.

Example

```
redis 127.0.0.1:6379> PUBLISH mychannel "hello, i m here"
(integer) 1
```

PubSub Punsubscribe Command

Redis **PUNSUBSCRIBE** command unsubscribes the client from the given patterns, or from all of them if none is given. When no patterns are specified, the client is unsubscribed from all the previously subscribed patterns. In this case, a message for every unsubscribed pattern will be sent to the client.

Syntax

Following is the basic syntax of Redis **PUNSUBSCRIBE** command.

```
redis 127.0.0.1:6379> PUNSUBSCRIBE [pattern [pattern ...]]
```

Return Value

Array reply.

Example

```
redis 127.0.0.1:6379> PUNSUBSCRIBE mychannel
1) "punsubscribe"
2) "a"
3) (integer) 1
```

PubSub Subscribe Command

Redis **SUBSCRIBE** command subscribes the client to the specified channels.

Syntax

Following is the basic syntax of Redis **SUBSCRIBE** command.

```
redis 127.0.0.1:6379> SUBSCRIBE channel [channel ...]
```

Return Value

Array reply.

Example

```
redis 127.0.0.1:6379> SUBSCRIBE mychannel
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "mychannel"
3) (integer) 1
1) "message"
2) "mychannel"
3) "a"
```

PubSub Unsubscribe Command

Redis **UNSUBSCRIBE** command unsubscribes the client from the given channels, or from all of them if none is given.

Syntax

Following is the basic syntax of Redis **UNSUBSCRIBE** command.

```
redis 127.0.0.1:6379> UNSUBSCRIBE channel [channel ...]
```

Return Value

Array reply.

Example

```
redis 127.0.0.1:6379> UNSUBSCRIBE mychannel
1) "unsubscribe"
2) "a"
3) (integer) 0
```

14. Redis – Transactions

Redis transactions allow the execution of a group of commands in a single step. Following are the two properties of Transactions.

- All commands in a transaction are sequentially executed as a single isolated operation. It is not possible that a request issued by another client is served in the middle of the execution of a Redis transaction.

- Redis transaction is also atomic. Atomic means either all of the commands or none are processed.

Sample

Redis transaction is initiated by command **MULTI** and then you need to pass a list of commands that should be executed in the transaction, after which the entire transaction is executed by **EXEC** command.

```
redis 127.0.0.1:6379> MULTI
OK
List of commands here
redis 127.0.0.1:6379> EXEC
```

Example

Following example explains how Redis transaction can be initiated and executed.

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET tutorial redis
QUEUED
redis 127.0.0.1:6379> GET tutorial
QUEUED
redis 127.0.0.1:6379> INCR visitors
QUEUED
redis 127.0.0.1:6379> EXEC

1) OK
2) "redis"
3) (integer) 1
```

Redis Transaction Commands

Following table shows some basic commands related to Redis transactions.

Sr. No.	Command & Description
1	**DISCARD** Discards all commands issued after MULTI
2	**EXEC** Executes all commands issued after MULTI
3	**MULTI** Marks the start of a transaction block
4	**UNWATCH** Forgets about all watched keys
5	**WATCH key [key ...]** Watches the given keys to determine the execution of the MULTI/EXEC block

Transactions Discard Command

Transactions Exec Command

Redis **EXEC** command executes all previously queued commands in a transaction and restores the connection state to normal.

Return Value

Array reply, each element being the reply to each of the commands in the atomic transaction.

Syntax

Following is the basic syntax of Redis **EXEC** command.

```
redis 127.0.0.1:6379> EXEC
```

Transactions Multi Command

Redis **MULTI** command marks the start of a transaction block. Subsequent commands will be queued for atomic execution using EXEC.

Return Value

Simple string reply: always OK.

Syntax

Following is the basic syntax of Redis **MULTI** command.

```
redis 127.0.0.1:6379> MULTI
```

Transactions Unwatch Command

Redis **UNWATCH** command flushes all the previously watched keys for a transaction.

Return Value

Simple string reply: always OK.

Syntax

Following is the basic syntax of Redis **UNWATCH** command.

```
redis 127.0.0.1:6379> UNWATCH
```

Transactions Watch Command

Redis **WATCH** command marks the given keys to be watched for conditional execution of a transaction.

Return Value

Simple string reply: always OK.

Syntax

Following is the basic syntax of Redis **WATCH** command.

```
redis 127.0.0.1:6379> WATCH key [key ...]
```

15. Redis – Scripting

Redis scripting is used to evaluate scripts using the Lua interpreter. It is built into Redis starting from version 2.6.0. The command used for scripting is **EVAL**command.

Syntax

Following is the basic syntax of **EVAL** command.

```
redis 127.0.0.1:6379> EVAL script numkeys key [key ...] arg [arg ...]
```

Example

Following example explains how Redis scripting works.

```
redis 127.0.0.1:6379> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1
key2 first second


1) "key1"
2) "key2"
3) "first"
4) "second"
```

Redis Scripting Commands

Following table lists some basic commands related to Redis Scripting.

Sr. No.	Command & Description
1	**EVAL script numkeys key [key ...] arg [arg ...]** Executes a Lua script.
2	**EVALSHA sha1 numkeys key [key ...] arg [arg ...]** Executes a Lua script.
3	**SCRIPT EXISTS script [script ...]** Checks the existence of scripts in the script cache.
4	**SCRIPT FLUSH** Removes all the scripts from the script cache.

5	**SCRIPT KILL** Kills the script currently in execution.
6	**SCRIPT LOAD script** Loads the specified Lua script into the script cache.

Scripting Eval Command

Redis **EVAL** command is used to evaluate scripts using the Lua interpreter. The first argument of EVAL is a Lua 5.1 script. The script does not need to define a Lua function (and should not). It is just a Lua program that will run in the context of the Redis server. The second argument of EVAL is the number of arguments that follows the script (starting from the third argument) that represent Redis key names. These arguments can be accessed by Lua using the KEYS global variable in the form of a one-based array (so KEYS[1], KEYS[2], ...). All the additional arguments should not represent key names and can be accessed by Lua using the ARGV global variable, similar to what happens with the keys (so ARGV[1], ARGV[2], ...).

Syntax

Following is the basic syntax of Redis **EVAL** command.

```
redis 127.0.0.1:6379> EVAL script numkeys key [key ...] arg [arg ...]
```

Example

```
redis 127.0.0.1:6379> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1
key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

Scripting Evalsha Command

Redis **EVALSHA** command evaluates a script cached on the server side by its SHA1 digest. Scripts are cached on the server side using the SCRIPT LOAD command. The command is otherwise identical to EVAL.

Syntax

Following is the basic syntax of Redis **EVALSHA** command.

```
redis 127.0.0.1:6379> EVALSHA sha1 numkeys key [key ...] arg [arg ...]
```

Example

```
redis 127.0.0.1:6379> EVALSHA "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1
key2 first second

1) "key1"

2) "key2"

3) "first"

4) "second"
```

Scripting Script Exists Command

Redis **SCRIPT EXISTS** command returns information about the existence of the scripts in the script cache. This command accepts one or more SHA1 digests and returns a list of ones or zeros to signal if the scripts are already defined or not inside the script cache. This can be useful before a pipelining operation to ensure that scripts are loaded (and if not, to load them using SCRIPT LOAD) so that the pipelining operation can be performed solely using EVALSHA instead of EVAL to save bandwidth.

Syntax

Following is the basic syntax of Redis **SCRIPT EXISTS** command.

```
redis 127.0.0.1:6379> SCRIPT EXISTS script [script ...]
```

Return Value

Array reply - The command returns an array of integers that corresponds to the specified SHA1 digest arguments. For every corresponding SHA1 digest of a script that actually exists in the script cache, 1 is returned, otherwise 0 is returned.

Example

```
redis 127.0.0.1:6379> SCRIPT LOAD "return 1"

ERR Unknown or disabled command 'SCRIPT'

redis 127.0.0.1:6379> SCRIPT EXISTS
ff9d4800c877a703b823dsdsfsffewfwefwefweac0578ff8db

ERR Unknown or disabled command 'SCRIPT'
```

Scripting Script Flush Command

Redis **SCRIPT FLUSH** command flushes the Lua scripts cache.

Syntax

Following is the basic syntax of Redis **SCRIPT FLUSH** command.

```
redis 127.0.0.1:6379> SCRIPT FLUSH
```

Return Value

Simple string reply.

Example

```
redis 127.0.0.1:6379> SCRIPT FLUSH
OK
```

Scripting Script Kill Command

Redis **SCRIPT KILL** command kills the currently executing Lua script, assuming no write operation was yet performed by the script. This command is mainly useful to kill a script that is running for too much time (for instance because it entered an infinite loop because of a bug). The script will be killed and the client currently blocked into EVAL will see the command returning an error. If the script already performed write operations, it cannot be killed in this way because it would violate Lua script atomicity contract. In such a case only SHUTDOWN NOSAVE is able to kill the script, killing the Redis process in a hard way preventing it to persist with half-written information.

Syntax

Following is the basic syntax of Redis **SCRIPT KILL** command.

```
redis 127.0.0.1:6379> SCRIPT KILL
```

Return Value

Simple string reply.

Example

```
redis 127.0.0.1:6379> SCRIPT KILL
OK
```

Scripting Script Load Command

Redis **SCRIPT LOAD** command loads a script into the scripts cache, without executing it. After the specified command is loaded into the script cache, it will be callable using EVALSHA with the correct SHA1 digest of the script, exactly like after the first successful invocation of EVAL. The script is guaranteed to stay in the script cache forever (unless SCRIPT FLUSH is called). The command works in the same way even if the script was already present in the script cache.

Syntax

Following is the basic syntax of Redis **SCRIPT LOAD** command.

```
redis 127.0.0.1:6379> SCRIPT LOAD script
```

Return Value

Bulk string reply - This command returns the SHA1 digest of the script added into the script cache.

Example

```
redis 127.0.0.1:6379> SCRIPT LOAD "return 1"

"e0e1f9fabfc9d4800c877a703b823ac0578ff8db"
```

16. Redis – Connections

Redis connection commands are basically used to manage client connections with Redis server.

Example

Following example explains how a client authenticates itself to Redis server and checks whether the server is running or not.

```
redis 127.0.0.1:6379> AUTH "password"
OK
redis 127.0.0.1:6379> PING
PONG
```

Redis Connection Commands

Following table lists some basic commands related to Redis connections.

Sr. No.	Command & Description
1	**AUTH password** Authenticates to the server with the given password
2	**ECHO message** Prints the given string
3	**PING** Checks whether the server is running or not
4	**QUIT** Closes the current connection
5	**SELECT index** Changes the selected database for the current connection

Connection Auth Command

Redis **AUTH** command is used to authenticate to the server with the given password. If the password matches the password in the configuration file, the server replies with the OK status code and starts accepting commands. Otherwise, an error is returned and the clients needs to try a new password.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **AUTH** command.

```
redis 127.0.0.1:6379> AUTH PASSWORD
```

Example

```
redis 127.0.0.1:6379> AUTH PASSWORD
(error) ERR Client sent AUTH, but no password is set
redis 127.0.0.1:6379> CONFIG SET requirepass "mypass"
OK
redis 127.0.0.1:6379> AUTH mypass
Ok
```

Connection Echo Command

Redis **ECHO** command is used to print the given string.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **ECHO** command.

```
redis 127.0.0.1:6379> ECHO SAMPLE_STRING
```

Example

```
redis 127.0.0.1:6379> ECHO "Hello World"
"Hello World"
```

Connection Ping Command

Redis **PING** command is used to check whether the server is running or not.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **PING** command.

```
redis 127.0.0.1:6379> PING
```

Example

```
redis 127.0.0.1:6379> PING

PONG
```

Connection Quit Command

Redis **QUIT** command asks the server to close the connection. The connection is closed as soon as all pending replies have been written to the client.

Return Value

String reply OK.

Syntax

Following is the basic syntax of Redis **QUIT** command.

```
redis 127.0.0.1:6379> QUIT
```

Example

```
redis 127.0.0.1:6379> QUIT

OK
```

Connection Select Command

Redis **SELECT** command is used to select the DB having the specified zero-based numeric index. New connections always use DB 0.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **SELECT** command.

```
redis 127.0.0.1:6379> SELECT DB_INDEX
```

Example

```
redis 127.0.0.1:6379> SELECT 1
OK
redis 127.0.0.1:6379[1]>
```

17. Redis – Server

Redis server commands are basically used to manage Redis server.

Example

Following example explains how we can get all statistics and information about the server.

```
redis 127.0.0.1:6379> INFO


# Server
redis_version:2.8.13
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:c2238b38b1edb0e2
redis_mode:standalone
os:Linux 3.5.0-48-generic x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.7.2
process_id:3856
run_id:0e61abd297771de3fe812a3c21027732ac9f41fe
tcp_port:6379
uptime_in_seconds:11554
uptime_in_days:0
hz:10
lru_clock:16651447
config_file:


# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0


# Memory
```

```
used_memory:589016
used_memory_human:575.21K
used_memory_rss:2461696
used_memory_peak:667312
used_memory_peak_human:651.67K
used_memory_lua:33792
mem_fragmentation_ratio:4.18
mem_allocator:jemalloc-3.6.0


# Persistence
loading:0
rdb_changes_since_last_save:3
rdb_bgsave_in_progress:0
rdb_last_save_time:1409158561
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:0
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1
aof_last_bgrewrite_status:ok
aof_last_write_status:ok


# Stats
total_connections_received:24
total_commands_processed:294
instantaneous_ops_per_sec:0
rejected_connections:0
sync_full:0
sync_partial_ok:0
sync_partial_err:0
expired_keys:0
evicted_keys:0
keyspace_hits:41
```

```
keyspace_misses:82
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:264

# 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

# CPU
used_cpu_sys:10.49
used_cpu_user:4.96
used_cpu_sys_children:0.00
used_cpu_user_children:0.01

# Keyspace
db0:keys=94,expires=1,avg_ttl=41638810
db1:keys=1,expires=0,avg_ttl=0
db3:keys=1,expires=0,avg_ttl=0
```

Redis Server Commands

Following table lists some basic commands related to Redis server.

Sr. No.	Command & Description
1	**BGREWRITEAOF** Asynchronously rewrites the append-only file
2	**BGSAVE** Asynchronously saves the dataset to the disk
3	**CLIENT KILL [ip:port] [ID client-id]** Kills the connection of a client
4	**CLIENT LIST** Gets the list of client connections to the server
5	**CLIENT GETNAME** Gets the name of the current connection
6	**CLIENT PAUSE timeout** Stops processing commands from the clients for a specified time
7	**CLIENT SETNAME connection-name** Sets the current connection name
8	**CLUSTER SLOTS** Gets an array of Cluster slot to node mappings
9	**COMMAND** Gets an array of Redis command details
10	**COMMAND COUNT** Gets total number of Redis commands
11	**COMMAND GETKEYS** Extracts the keys given a full Redis command
12	**BGSAVE** Asynchronously saves the dataset to the disk
13	**COMMAND INFO command-name [command-name ...]** Gets an array of specific Redis command details

14	**CONFIG GET parameter** Gets the value of a configuration parameter
15	**CONFIG REWRITE** Rewrites the configuration file with the in-memory configuration
16	**CONFIG SET parameter value** Sets a configuration parameter to the given value
17	**CONFIG RESETSTAT** Resets the stats returned by INFO
18	**DBSIZE** Returns the number of keys in the selected database
19	**DEBUG OBJECT key** Gets debugging information about a key
20	**DEBUG SEGFAULT** Makes the server crash
21	**FLUSHALL** Removes all the keys from all databases
22	**FLUSHDB** Removes all the keys from the current database
23	**INFO [section]** Gets information and statistics about the server
24	**LASTSAVE** Gets the UNIX time stamp of the last successful save to the disk
25	**MONITOR** Listens for all the requests received by the server in real time
26	**ROLE** Returns the role of the instance in the context of replication
27	**SAVE** Synchronously saves the dataset to the disk
28	**SHUTDOWN [NOSAVE] [SAVE]** Synchronously saves the dataset to the disk and then shuts down the server

29	**SLAVEOF host port** Makes the server a slave of another instance, or promotes it as a master
30	**SLOWLOG subcommand [argument]** Manages the Redis slow queries log
31	**SYNC** Command used for replication
32	**TIME** Returns the current server time

Server Bgrewriteaof Command

Redis **BGREWRITEAOF** command instructs Redis to start an Append Only File rewrite process. The rewrite will create a small optimized version of the current Append Only File. If BGREWRITEAOF fails, no data gets lost as the old AOF will be untouched. The rewrite will be only triggered by Redis, if there is not already a background process doing persistence.

Return Value

Simple string reply: always OK.

Syntax

Following is the basic syntax of Redis **BGREWRITEAOF** command.

```
redis 127.0.0.1:6379> BGREWRITEAOF
```

Server Bgsave Command

Redis **BGSAVE** command saves the DB in the background. The OK code is immediately returned. Redis forks, the parent continues to serve the clients, the child saves the DB on the disk, then exits. A client may be able to check if the operation succeeded using the LASTSAVE command.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **BGSAVE** command.

```
redis 127.0.0.1:6379> BGSAVE
```

Server Client Kill Command

Redis **CLIENT KILL** command closes a given client connection.

Return Value

Simple string reply: OK if the connection exists and has been closed.

Syntax

Following is the basic syntax of Redis **CLIENT KILL** command.

```
redis 127.0.0.1:6379> CLIENT KILL [ip:port] [ID client-id] [TYPE
normal|slave|pubsub] [ADDR ip:port] [SKIPME yes/no]
```

With Redis 2.8.12 or greater, the command can be run with multiple options as shown below:

- CLIENT KILL ADDR ip:port. This is exactly the same as the old three-arguments behavior.

- CLIENT KILL ID client-id. Allows to kill a client by its unique ID field, which was introduced in the CLIENT LIST command starting from Redis 2.8.12.

- CLIENT KILL TYPE type, where type is one of normal, slave, pubsub. This closes the connections of all the clients in the specified class. Note: Clients blocked into the MONITOR command are considered to belong to the normal class.

- CLIENT KILL SKIPME yes/no. By default, this option is set to yes, that is, the client calling the command will not get killed. However, setting this option to no will have the effect of also killing the client calling the command.

Server Client List Command

Redis **CLIENT LIST** command returns the information and statistics about the client connections server in a human readable format.

Return Value

Bulk string reply, a unique string.

Syntax

Following is the basic syntax of Redis **CLIENT LIST** command.

```
redis 127.0.0.1:6379> CLIENT LIST
```

Description of Fields

- **id**: Unique 64-bit client ID (introduced in Redis 2.8.12)
- **addr**: Address/port of the client
- **fd**: File descriptor corresponding to the socket

- **age**: Total duration of the connection in seconds
- **idle**: Idle time of the connection in seconds
- **flags**: Client flags (see below)
- **db**: Current database ID
- **sub**: Number of channel subscriptions
- **psub**: Number of pattern matching subscriptions
- **multi**: Number of commands in a MULTI/EXEC context
- **qbuf**: Query buffer length (0 means no query pending)
- **qbuf-free**: Free space of the query buffer (0 means the buffer is full)
- **obl**: Output buffer length
- **oll**: Output list length (replies are queued in this list when the buffer is full)
- **omem**: Output buffer memory usage
- **events**: File descriptor events (see below)
- **cmd**: Last command played

Server Client Getname Command

Redis **CLIENT GETNAME** command returns the name of the current connection as set by CLIENT SETNAME. Since every new connection starts without an associated name, if no name was assigned, a null bulk reply is returned.

Return Value

Bulk string reply: The connection name, or a null bulk reply, if no name is set.

Syntax

Following is the basic syntax of Redis **CLIENT GETNAME** command.

```
redis 127.0.0.1:6379> CLIENT GETNAME
```

Server Client Pause Command

Redis **CLIENT PAUSE** command is a connections control command, able to suspend all the Redis clients for the specified amount of time (in milliseconds). The command performs the following actions:

- It stops processing all the pending commands from normal and pub/sub clients. However, interactions with slaves will continue normally.

- It returns OK to the caller ASAP, so the CLIENT PAUSE command execution is not paused by itself.

- When the specified amount of time has elapsed, all the clients are unblocked: this will trigger the processing of all the commands accumulated in the query buffer of every client during the pause.

Return Value

Simple string reply: The command returns OK or an error if the timeout is invalid.

Syntax

Following is the basic syntax of Redis **CLIENT PAUSE** command.

```
redis 127.0.0.1:6379> CLIENT PAUSE timeout
```

Server Client Setname Command

Redis **CLIENT SETNAME** command assigns a name to the current connection. The assigned name is displayed in the output of CLIENT LIST so that it is possible to identify the client that performed a given connection.

Return Value

Simple string reply, OK if the connection name was successfully set.

Syntax

Following is the basic syntax of Redis **CLIENT SETNAME** command.

```
redis 127.0.0.1:6379> CLIENT SETNAME connection-name
```

Example

```
redis 127.0.0.1:6379> CLIENT SETNAME "my connection"
OK
```

Server Cluster Slots Command

Redis **CLUSTER SLOTS** returns an array reply of the current cluster state.

Return Value

Array reply - Nested list of slot ranges with IP/Port mappings.

Syntax

Following is the basic syntax of Redis **CLUSTER SLOTS** command.

```
redis 127.0.0.1:6379> CLUSTER SLOTS
```

Sample Reply

```
redis 127.0.0.1:6379> CLIENT SETNAME "my connection"
1) 1) (integer) 0
```

```
   2) (integer) 4095
   3) 1) "127.0.0.1"
      2) (integer) 7000
   4) 1) "127.0.0.1"
      2) (integer) 7004
2) 1) (integer) 12288
   2) (integer) 16383
   3) 1) "127.0.0.1"
      2) (integer) 7003
   4) 1) "127.0.0.1"
      2) (integer) 7007
3) 1) (integer) 4096
   2) (integer) 8191
   3) 1) "127.0.0.1"
      2) (integer) 7001
   4) 1) "127.0.0.1"
      2) (integer) 7005
4) 1) (integer) 8192
   2) (integer) 12287
   3) 1) "127.0.0.1"
      2) (integer) 7002
   4) 1) "127.0.0.1"
      2) (integer) 7006
```

Server Command

Redis **COMMAND** array replies the details about all Redis commands.

Return Value

Array reply - Nested list of command details. Commands are returned in a random order.

Syntax

Following is the basic syntax of Redis **COMMAND** command.

```
redis 127.0.0.1:6379> COMMAND
```

Example

```
redis 127.0.0.1:6379> COMMAND
1) 1) "pfcount"
     2) (integer) -2
     3) 1) write
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
  2) 1) "command"
     2) (integer) 0
     3) 1) readonly
        2) loading
        3) stale
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
  3) 1) "zscan"
     2) (integer) -3
     3) 1) readonly
        2) random
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
  4) 1) "echo"
     2) (integer) 2
     3) 1) readonly
        2) fast
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
  5) 1) "select"
     2) (integer) 2
     3) 1) readonly
        2) loading
        3) fast
     4) (integer) 0
```

```
      5) (integer) 0
      6) (integer) 0
   6) 1) "zcount"
      2) (integer) 4
      3) 1) readonly
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
   7) 1) "substr"
      2) (integer) 4
      3) 1) readonly
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
   8) 1) "pttl"
      2) (integer) 2
      3) 1) readonly
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
   9) 1) "hincrbyfloat"
      2) (integer) 4
      3) 1) write
         2) denyoom
         3) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
  10) 1) "hlen"
      2) (integer) 2
      3) 1) readonly
         2) fast
      4) (integer) 1
      5) (integer) 1
```

```
    6) (integer) 1
11) 1) "incrby"
    2) (integer) 3
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
12) 1) "setex"
    2) (integer) 4
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
13) 1) "persist"
    2) (integer) 2
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
14) 1) "setbit"
    2) (integer) 4
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
15) 1) "info"
    2) (integer) -1
    3) 1) readonly
       2) loading
       3) stale
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
16) 1) "scard"
    2) (integer) 2
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
17) 1) "srandmember"
    2) (integer) -2
    3) 1) readonly
       2) random
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
18) 1) "lrem"
    2) (integer) 4
    3) 1) write
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
19) 1) "append"
    2) (integer) 3
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
20) 1) "hgetall"
    2) (integer) 2
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
21) 1) "zincrby"
```

```
    2) (integer) 4
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
22) 1) "rpop"
    2) (integer) 2
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
23) 1) "cluster"
    2) (integer) -2
    3) 1) readonly
       2) admin
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
24) 1) "ltrim"
    2) (integer) 4
    3) 1) write
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
25) 1) "flushdb"
    2) (integer) 1
    3) 1) write
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
26) 1) "rpoplpush"
    2) (integer) 3
    3) 1) write
```

```
       2) denyoom
    4) (integer) 1
    5) (integer) 2
    6) (integer) 1
27) 1) "expire"
    2) (integer) 3
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
28) 1) "psync"
    2) (integer) 3
    3) 1) readonly
       2) admin
       3) noscript
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0

29) 1) "zremrangebylex"
    2) (integer) 4
    3) 1) write
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
30) 1) "pubsub"
    2) (integer) -2
    3) 1) readonly
       2) pubsub
       3) random
       4) loading
       5) stale
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
```

```
31) 1) "setnx"
    2) (integer) 3
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
32) 1) "pexpireat"
    2) (integer) 3
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
33) 1) "psubscribe"
    2) (integer) -2
    3) 1) readonly
       2) pubsub
       3) noscript
       4) loading
       5) stale
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
34) 1) "zrevrange"
    2) (integer) -4
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
35) 1) "hmget"
    2) (integer) -3
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
36) 1) "object"
    2) (integer) -2
    3) 1) readonly
    4) (integer) 2
    5) (integer) 2
    6) (integer) 2
37) 1) "watch"
    2) (integer) -2
    3) 1) readonly
       2) noscript
       3) fast
    4) (integer) 1
    5) (integer) -1
    6) (integer) 1
38) 1) "setrange"
    2) (integer) 4
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
39) 1) "sdiffstore"
    2) (integer) -3
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) -1
    6) (integer) 1
40) 1) "flushall"
    2) (integer) 1
    3) 1) write
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
41) 1) "sadd"
```

```
    2) (integer) -3
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
42) 1) "renamenx"
    2) (integer) 3
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 2
    6) (integer) 1
43) 1) "zrangebyscore"
    2) (integer) -4
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
44) 1) "bitop"
    2) (integer) -4
    3) 1) write
       2) denyoom
    4) (integer) 2
    5) (integer) -1
    6) (integer) 1
45) 1) "get"
    2) (integer) 2
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
46) 1) "hmset"
    2) (integer) -4
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
47) 1) "type"
    2) (integer) 2
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
48) 1) "evalsha"
    2) (integer) -3
    3) 1) noscript
       2) movablekeys
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
49) 1) "zrevrangebyscore"
    2) (integer) -4
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
50) 1) "set"
    2) (integer) -3
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
51) 1) "getset"
    2) (integer) 3
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
52) 1) "punsubscribe"
    2) (integer) -1
    3) 1) readonly
       2) pubsub
       3) noscript
       4) loading
       5) stale
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
53) 1) "publish"
    2) (integer) 3
    3) 1) readonly
       2) pubsub
       3) loading
       4) stale
       5) fast
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
54) 1) "lset"
    2) (integer) 4
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
55) 1) "rename"
    2) (integer) 3
    3) 1) write
    4) (integer) 1
    5) (integer) 2
    6) (integer) 1
56) 1) "bgsave"
    2) (integer) 1
    3) 1) readonly
       2) admin
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
57) 1) "decrby"
    2) (integer) 3
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
58) 1) "sunion"
    2) (integer) -2
    3) 1) readonly
       2) sort_for_script
    4) (integer) 1
    5) (integer) -1
    6) (integer) 1
59) 1) "blpop"
    2) (integer) -3
    3) 1) write
       2) noscript
    4) (integer) 1
    5) (integer) -2
    6) (integer) 1
60) 1) "zrem"
    2) (integer) -3
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
61) 1) "readonly"
    2) (integer) 1
    3) 1) readonly
       2) fast
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
62) 1) "exists"
    2) (integer) 2
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
63) 1) "linsert"
    2) (integer) 5
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
64) 1) "lindex"
    2) (integer) 3
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
65) 1) "scan"
    2) (integer) -2
    3) 1) readonly
       2) random
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
66) 1) "migrate"
```

```
    2) (integer) -6
    3) 1) write
       2) admin
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
67) 1) "ping"
    2) (integer) 1
    3) 1) readonly
       2) stale
       3) fast
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
68) 1) "zunionstore"
    2) (integer) -4
    3) 1) write
       2) denyoom
       3) movablekeys
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
69) 1) "latency"
    2) (integer) -2
    3) 1) readonly
       2) admin
       3) noscript
       4) loading
       5) stale
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
70) 1) "role"
    2) (integer) 1
    3) 1) admin
       2) noscript
```

```
      3) loading
      4) stale
   4) (integer) 0
   5) (integer) 0
   6) (integer) 0
71) 1) "ttl"
   2) (integer) 2
   3) 1) readonly
      2) fast
   4) (integer) 1
   5) (integer) 1
   6) (integer) 1
72) 1) "del"
   2) (integer) -2
   3) 1) write
   4) (integer) 1
   5) (integer) -1
   6) (integer) 1
73) 1) "wait"
   2) (integer) 3
   3) 1) readonly
      2) noscript
   4) (integer) 0
   5) (integer) 0
   6) (integer) 0
74) 1) "zscore"
   2) (integer) 3
   3) 1) readonly
      2) fast
   4) (integer) 1
   5) (integer) 1
   6) (integer) 1
75) 1) "zrevrangebylex"
   2) (integer) -4
   3) 1) readonly
   4) (integer) 1
```

```
    5) (integer) 1
    6) (integer) 1
76) 1) "sscan"
    2) (integer) -3
    3) 1) readonly
       2) random
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
```

```
77) 1) "incrbyfloat"
    2) (integer) 3
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
78) 1) "decr"
    2) (integer) 2
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
79) 1) "getbit"
    2) (integer) 3
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
80) 1) "spop"
    2) (integer) 2
    3) 1) write
       2) noscript
       3) random
       4) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
```

```
81) 1) "hkeys"
    2) (integer) 2
    3) 1) readonly
       2) sort_for_script
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
82) 1) "pfmerge"
    2) (integer) -2
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) -1
    6) (integer) 1
83) 1) "zrange"
    2) (integer) -4
    3) 1) readonly
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
84) 1) "monitor"
    2) (integer) 1
    3) 1) readonly
       2) admin
       3) noscript
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
85) 1) "zinterstore"
    2) (integer) -4
    3) 1) write
       2) denyoom
       3) movablekeys
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
86) 1) "rpushx"
    2) (integer) 3
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
87) 1) "llen"
    2) (integer) 2
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
88) 1) "hincrby"
    2) (integer) 4
    3) 1) write
       2) denyoom
       3) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
89) 1) "save"
    2) (integer) 1
    3) 1) readonly
       2) admin
       3) noscript
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
```

```
90) 1) "zremrangebyrank"
    2) (integer) 4
    3) 1) write
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
91) 1) "auth"
    2) (integer) 2
    3) 1) readonly
       2) noscript
       3) loading
       4) stale
       5) fast
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
92) 1) "zcard"
    2) (integer) 2
    3) 1) readonly
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
93) 1) "psetex"
    2) (integer) 4
    3) 1) write
       2) denyoom
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
```

```
94) 1) "shutdown"
    2) (integer) -1
    3) 1) readonly
       2) admin
       3) loading
       4) stale
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
95) 1) "sync"
    2) (integer) 1
    3) 1) readonly
       2) admin
       3) noscript
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
96) 1) "dbsize"
    2) (integer) 1
    3) 1) readonly
       2) fast
    4) (integer) 0
    5) (integer) 0
    6) (integer) 0
97) 1) "expireat"
    2) (integer) 3
    3) 1) write
       2) fast
    4) (integer) 1
    5) (integer) 1
    6) (integer) 1
```

```
 98) 1) "subscribe"
     2) (integer) -2
     3) 1) readonly
        2) pubsub
        3) noscript
        4) loading
        5) stale
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
 99) 1) "brpop"
     2) (integer) -3
     3) 1) write
        2) noscript
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
100) 1) "sort"
     2) (integer) -2
     3) 1) write
        2) denyoom
        3) movablekeys
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
101) 1) "sunionstore"
     2) (integer) -3
     3) 1) write
        2) denyoom
     4) (integer) 1
     5) (integer) -1
     6) (integer) 1
```

```
102) 1) "zrangebylex"
     2) (integer) -4
     3) 1) readonly
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
103) 1) "zlexcount"
     2) (integer) 4
     3) 1) readonly
        2) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
104) 1) "lpush"
     2) (integer) -3
     3) 1) write
        2) denyoom
        3) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
105) 1) "incr"
     2) (integer) 2
     3) 1) write
        2) denyoom
        3) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
106) 1) "mget"
     2) (integer) -2
     3) 1) readonly
     4) (integer) 1
     5) (integer) -1
     6) (integer) 1
107) 1) "getrange"
     2) (integer) 4
     3) 1) readonly
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
108) 1) "slaveof"
     2) (integer) 3
     3) 1) admin
        2) noscript
        3) stale
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
109) 1) "bitpos"
     2) (integer) -3
     3) 1) readonly
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
110) 1) "rpush"
     2) (integer) -3
     3) 1) write
        2) denyoom
        3) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
```

```
111) 1) "config"
     2) (integer) -2
     3) 1) readonly
        2) admin
        3) stale
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
112) 1) "srem"
     2) (integer) -3
     3) 1) write
        2) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
113) 1) "mset"
     2) (integer) -3
     3) 1) write
        2) denyoom
     4) (integer) 1
     5) (integer) -1
     6) (integer) 2
114) 1) "lrange"
     2) (integer) 4
     3) 1) readonly
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
115) 1) "replconf"
     2) (integer) -1
     3) 1) readonly
        2) admin
        3) noscript
        4) loading
        5) stale
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
116) 1) "hsetnx"
     2) (integer) 4
     3) 1) write
        2) denyoom
        3) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
117) 1) "discard"
     2) (integer) 1
     3) 1) readonly
        2) noscript
        3) fast
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
118) 1) "pexpire"
     2) (integer) 3
     3) 1) write
        2) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
119) 1) "pfdebug"
     2) (integer) -3
     3) 1) write
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
120) 1) "asking"
     2) (integer) 1
     3) 1) readonly
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
121) 1) "client"
     2) (integer) -2
     3) 1) readonly
        2) admin
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
122) 1) "pfselftest"
     2) (integer) 1
     3) 1) readonly
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
123) 1) "bgrewriteaof"
     2) (integer) 1
     3) 1) readonly
        2) admin
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
124) 1) "zremrangebyscore"
     2) (integer) 4
     3) 1) write
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
125) 1) "sinterstore"
     2) (integer) -3
     3) 1) write
        2) denyoom
     4) (integer) 1
     5) (integer) -1
     6) (integer) 1
126) 1) "lpushx"
```

```
     2) (integer) 3
     3) 1) write
        2) denyoom
        3) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
127) 1) "restore"
     2) (integer) -4
     3) 1) write
        2) denyoom
        3) admin
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
128) 1) "unsubscribe"
     2) (integer) -1
     3) 1) readonly
        2) pubsub
        3) noscript
        4) loading
        5) stale
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
```

```
129) 1) "zrank"
     2) (integer) 3
     3) 1) readonly
        2) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
130) 1) "readwrite"
     2) (integer) 1
     3) 1) readonly
        2) fast
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
131) 1) "hget"
     2) (integer) 3
     3) 1) readonly
        2) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
132) 1) "bitcount"
     2) (integer) -2
     3) 1) readonly
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
133) 1) "randomkey"
     2) (integer) 1
     3) 1) readonly
        2) random
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
```

```
134) 1) "restore-asking"
     2) (integer) -4
     3) 1) write
        2) denyoom
        3) admin
        4) asking
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
135) 1) "time"
     2) (integer) 1
     3) 1) readonly
        2) random
        3) fast
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
136) 1) "zrevrank"
     2) (integer) 3
     3) 1) readonly
        2) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
137) 1) "hset"
     2) (integer) 4
     3) 1) write
        2) denyoom
        3) fast
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
138) 1) "sinter"
     2) (integer) -2
     3) 1) readonly
        2) sort_for_script
```

```
      4) (integer) 1
      5) (integer) -1
      6) (integer) 1
139) 1) "dump"
      2) (integer) 2
      3) 1) readonly
         2) admin
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
140) 1) "move"
      2) (integer) 3
      3) 1) write
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
141) 1) "strlen"
      2) (integer) 2
      3) 1) readonly
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
142) 1) "unwatch"
      2) (integer) 1
      3) 1) readonly
         2) noscript
         3) fast
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
143) 1) "lpop"
      2) (integer) 2
      3) 1) write
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
144) 1) "smembers"
      2) (integer) 2
      3) 1) readonly
         2) sort_for_script
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
145) 1) "msetnx"
      2) (integer) -3
      3) 1) write
         2) denyoom
      4) (integer) 1
      5) (integer) -1
      6) (integer) 2
146) 1) "pfadd"
      2) (integer) -2
      3) 1) write
         2) denyoom
         3) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
147) 1) "zadd"
      2) (integer) -4
      3) 1) write
         2) denyoom
         3) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
148) 1) "lastsave"
      2) (integer) 1
      3) 1) readonly
```

```
         2) random
         3) fast
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
149) 1) "exec"
      2) (integer) 1
      3) 1) noscript
         2) skip_monitor
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
150) 1) "sismember"
      2) (integer) 3
      3) 1) readonly
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
151) 1) "debug"
      2) (integer) -2
      3) 1) admin
         2) noscript
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
152) 1) "slowlog"
      2) (integer) -2
      3) 1) readonly
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
153) 1) "hexists"
      2) (integer) 3
      3) 1) readonly
         2) fast
```

```
     4) (integer) 1
     5) (integer) 1
     6) (integer) 1
154) 1) "eval"
     2) (integer) -3
     3) 1) noscript
        2) movablekeys
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
155) 1) "smove"
     2) (integer) 4
     3) 1) write
        2) fast
     4) (integer) 1
     5) (integer) 2
     6) (integer) 1
156) 1) "multi"
     2) (integer) 1
     3) 1) readonly
        2) noscript
        3) fast
     4) (integer) 0
     5) (integer) 0
     6) (integer) 0
157) 1) "sdiff"
     2) (integer) -2
     3) 1) readonly
        2) sort_for_script
     4) (integer) 1
     5) (integer) -1
     6) (integer) 1
158) 1) "hscan"
     2) (integer) -3
     3) 1) readonly
        2) random
```

```
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
159) 1) "brpoplpush"
      2) (integer) 4
      3) 1) write
         2) denyoom
         3) noscript
      4) (integer) 1
      5) (integer) 2
      6) (integer) 1
160) 1) "script"
      2) (integer) -2
      3) 1) readonly
         2) admin
         3) noscript
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
161) 1) "keys"
      2) (integer) 2
      3) 1) readonly
         2) sort_for_script
      4) (integer) 0
      5) (integer) 0
      6) (integer) 0
162) 1) "hdel"
      2) (integer) -3
      3) 1) write
         2) fast
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
163) 1) "hvals"
      2) (integer) 2
      3) 1) readonly
```

```
         2) sort_for_script
      4) (integer) 1
      5) (integer) 1
      6) (integer) 1
```

Server Command Count Command

Redis **COMMAND COUNT** returns the number of total commands in this Redis server.

Return Value

Integer reply – The number of commands returned by COMMAND.

Syntax

Following is the basic syntax of Redis **COMMAND COUNT** command.

```
redis 127.0.0.1:6379> COMMAND COUNT
```

Example

```
redis 127.0.0.1:6379> COMMAND COUNT
(integer) 163
```

Server Command Getkeys Command

Redis **COMMAND GETKEYS** is a helper command to let you find the keys from a full Redis command.

Return Value

Array reply – The list of keys from your command.

Syntax

Following is the basic syntax of Redis **COMMAND GETKEYS** command.

```
redis 127.0.0.1:6379> COMMAND GETKEYS
```

Example

```
redis 127.0.0.1:6379> COMMAND GETKEYS MSET a b c d e f
1) "a"
2) "c"
3) "e"
```

Server Bgsave Command

Redis **BGSAVE** command saves the DB in the background. The OK code is immediately returned. Redis forks, the parent continues to serve the clients, the child saves the DB on the disk, then exits. A client may be able to check if the operation succeeded using the LASTSAVE command.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **BGSAVE** command.

```
redis 127.0.0.1:6379> BGSAVE
```

Server Command Info Command

Redis **COMMAND INFO** returns the details about multiple Redis commands.

Return Value

Array reply – The nested list of command details.

Syntax

Following is the basic syntax of Redis **COMMAND INFO** command.

```
redis 127.0.0.1:6379> COMMAND INFO command-name [command-name ...]
```

Example

```
redis 127.0.0.1:6379> COMMAND INFO get set eval
1) 1) "get"
   2) (integer) 2
   3) 1) readonly
      2) fast
   4) (integer) 1
   5) (integer) 1
   6) (integer) 1
2) 1) "set"
   2) (integer) -3
   3) 1) write
      2) denyoom
   4) (integer) 1
```

```
5) (integer) 1
6) (integer) 1
```

```
3) 1) "eval"
   2) (integer) -3
   3) 1) noscript
      2) movablekeys
   4) (integer) 0
   5) (integer) 0
   6) (integer) 0
redis> COMMAND INFO foo evalsha config bar
1) (nil)
2) 1) "evalsha"
   2) (integer) -3
   3) 1) noscript
      2) movablekeys
   4) (integer) 0
   5) (integer) 0
   6) (integer) 0
3) 1) "config"
   2) (integer) -2
   3) 1) readonly
      2) admin
      3) stale
   4) (integer) 0
   5) (integer) 0
   6) (integer) 0
4) (nil)
```

Server Config Get Command

Redis **CONFIG GET** command is used to read the configuration parameters of a running Redis server. Not all the configuration parameters are supported in Redis 2.4, while Redis 2.6 can read the whole configuration of a server using this command.

Return Value

The return type of the command is a Bulk string reply.

Syntax

Following is the basic syntax of Redis **CONFIG GET** command.

```
redis 127.0.0.1:6379> CONFIG GET parameter
```

Example

```
redis 127.0.0.1:6379> config get *max-*-entries*
1) "hash-max-zipmap-entries"
2) "512"
3) "list-max-ziplist-entries"
4) "512"
5) "set-max-intset-entries"
6) "512"
```

Server Config Rewrite Command

Redis **CONFIG REWRITE** command rewrites the redis.conf file the server was started with, applying minimal changes needed to reflect the configuration currently used by the server. It may be different compared to the original one because of the use of the CONFIG SET command.

Return Value

String reply – OK, when the configuration is rewritten properly. Otherwise, an error is returned.

Syntax

Following is the basic syntax of Redis **CONFIG REWRITE** command.

```
redis 127.0.0.1:6379> CONFIG REWRITE parameter
```

Server Config Set Command

Redis **CONFIG Set** command is used in order to reconfigure the server at run time without the need to restart Redis. You can change both trivial parameters or switch from one to another persistence option using this command.

Return Value

String reply – OK, when the configuration is set properly. Otherwise, an error is returned.

Syntax

Following is the basic syntax of Redis **CONFIG Set** command.

```
redis 127.0.0.1:6379> CONFIG Set parameter value
```

Example

```
redis 127.0.0.1:6379> CONFIG Get "requirePass"
""
redis 127.0.0.1:6379> CONFIG Set "requirePass" "pass1"
OK
```

Server Config Resetstat Command

Redis **CONFIG RESETSTAT** command resets the statistics reported by Redis using the INFO command. Following counters can be reset using this command:

- Keyspace hits
- Keyspace misses
- Number of commands processed
- Number of connections received
- Number of expired keys
- Number of rejected connections
- Latest fork(2) time
- The aof_delayed_fsync counter

Return Value

String reply - Always OK.

Syntax

Following is the basic syntax of Redis **CONFIG RESETSTAT** command.

```
redis 127.0.0.1:6379> CONFIG RESETSTAT
```

Server Dbsize Command

Redis **DBSIZE** command is used to get the number of keys in selected database.

Return Value

Integer reply.

Syntax

Following is the basic syntax of Redis **DBSIZE** command.

```
redis 127.0.0.1:6379> DBSIZE
```

Example

```
redis 127.0.0.1:6379> DBSIZE
(integer) 147
```

Server Debug Object Command

Redis **DEBUG OBJECT** is a debugging command that should not be used by the clients. Check the OBJECT command instead.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **DEBUG OBJECT** command.

```
redis 127.0.0.1:6379> DEBUG OBJECT key
```

Example

```
redis 127.0.0.1:6379> SET a a
OK
redis 127.0.0.1:6379> DEBUG OBJECT a
Value at:0x7f68f7886df0 refcount:1 encoding:raw serializedlength:2 lru:1566733
lru_s
```

Server Debug Segfault Command

Redis **DEBUG SEGFAULT** performs an invalid memory access that crashes Redis. It is used to simulate bugs during the development.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **DEBUG SEGFAULT** command.

```
redis 127.0.0.1:6379> DEBUG SEGFAULT
```

Example

```
redis 127.0.0.1:6379> DEBUG SEGFAULT
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

```
not connected>
```

Server Flushall Command

Redis **FLUSHALL** deletes all the keys of all the existing databases, not just the currently selected one. This command never fails.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **FLUSHALL** command.

```
redis 127.0.0.1:6379> FLUSHALL
```

Example

```
redis 127.0.0.1:6379> FLUSHALL
OK
```

Server Flushdb Command

Redis **FLUSHDB** deletes all the keys of the currently selected DB. This command never fails.

Return Value

String reply.

Syntax

Following is the basic syntax of Redis **FLUSHDB** command.

```
redis 127.0.0.1:6379> FLUSHDB
```

Example

```
redis 127.0.0.1:6379> FLUSHDB
OK
```

Server Info Command

Redis **INFO** command returns information and statistics about the server in a format that is simple to parse by the computers and easy to read by humans.

Following are some optional parameters.

- **server**: General information about the Redis server
- **clients**: Client connections section
- **memory**: Memory consumption-related information
- **persistence**: RDB and AOF related information
- **stats**: General statistics
- **replication**: Master/slave replication information
- **cpu**: CPU consumption statistics
- **commandstats**: Redis command statistics
- **cluster**: Redis Cluster section
- **keyspace**: Database-related statistics
- **all**: Return all sections
- **default**: Return only the default set of sections

Return Value

String reply - As a collection of text lines.

Syntax

Following is the basic syntax of Redis **INFO** command.

```
redis 127.0.0.1:6379> INFO
```

Example

```
redis 127.0.0.1:6379> INFO
# Server
redis_version:2.8.13
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:c2238b38b1edb0e2
redis_mode:standalone
os:Linux 3.5.0-48-generic x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.7.2
process_id:3856
run_id:0e61abd297771de3fe812a3c21027732ac9f41fe
tcp_port:6379
uptime_in_seconds:11554
uptime_in_days:0
```

```
hz:10
lru_clock:16651447
config_file:

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:589016
used_memory_human:575.21K
used_memory_rss:2461696
used_memory_peak:667312
used_memory_peak_human:651.67K



used_memory_lua:33792
mem_fragmentation_ratio:4.18
mem_allocator:jemalloc-3.6.0

# Persistence
loading:0
rdb_changes_since_last_save:3
rdb_bgsave_in_progress:0
rdb_last_save_time:1409158561
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:0
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1
aof_last_bgrewrite_status:ok
```

```
aof_last_write_status:ok

# Stats
total_connections_received:24
total_commands_processed:294
instantaneous_ops_per_sec:0
rejected_connections:0
sync_full:0
sync_partial_ok:0
sync_partial_err:0
expired_keys:0
evicted_keys:0
keyspace_hits:41
keyspace_misses:82
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:264

# 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

# CPU
used_cpu_sys:10.49
used_cpu_user:4.96
used_cpu_sys_children:0.00
used_cpu_user_children:0.01

# Keyspace
db0:keys=94,expires=1,avg_ttl=41638810
db1:keys=1,expires=0,avg_ttl=0
```

```
db3:keys=1,expires=0,avg_ttl=0
```

Server Lastsave Command

Redis **LASTSAVE** command returns the UNIX TIME of the last DB save executed with success. A client may check if a BGSAVE command succeeded reading the LASTSAVE value, then issuing a BGSAVE command and checking at regular intervals every N seconds if LASTSAVE has changed.

Return Value

Integer reply, a UNIX time stamp.

Syntax

Following is the basic syntax of Redis **LASTSAVE** command.

```
redis 127.0.0.1:6379> LASTSAVE
```

Example

```
redis 127.0.0.1:6379> LASTSAVE
(integer) 1410853592
```

Server Monitor Command

Redis **MONITOR** is a debugging command that streams back every command processed by the Redis server. It can help in understanding what is happening to the database. This command can be used both via Redis-cli and via telnet. The ability to see all the requests processed by the server is useful in order to spot bugs in an application, both while using Redis as a database and as a distributed caching system.

Syntax

Following is the basic syntax of Redis **MONITOR** command.

```
redis 127.0.0.1:6379> MONITOR
```

Example

```
redis 127.0.0.1:6379> MONITOR
OK
1410855382.370791 [0 127.0.0.1:60581] "info"
1410855404.062722 [0 127.0.0.1:60581] "get" "a"
```

Server Role Command

Redis **ROLE** is a debugging command that streams back every command processed by the Redis server. It can help in understanding what is happening to the database. This command can be used both via Redis-cli and via telnet. The ability to see all the requests processed by the server is useful in order to spot bugs in an application, both while using Redis as a database and as a distributed caching system.

Syntax

Following is the basic syntax of Redis **ROLE** command.

```
redis 127.0.0.1:6379> ROLE
```

Return Value

The command returns an array of elements. The first element is the role of the instance, as one of the following three strings:

- master
- slave
- sentinel

Example

```
redis 127.0.0.1:6379> ROLE
1) "master"
2) (integer) 3129659
3) 1) 1) "127.0.0.1"
      2) "9001"
      3) "3129242"
   2) 1) "127.0.0.1"
      2) "9002"
      3) "3129543"
```

Server Save Command

Redis **SAVE** command performs a synchronous save of the dataset producing a point in time snapshot of all the data inside the Redis instance, in the form of a RDB file.

Return Value

String reply - The commands returns OK on success.

Syntax

Following is the basic syntax of Redis **SAVE** command.

```
redis 127.0.0.1:6379> SAVE
```

Example

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

Server Shutdown Command

Redis **SHUTDOWN** command stops all clients, performs a save, flushes all append only files (if AOF is enabled) and quits the server.

Return Value

Simple string reply on error. On success nothing is returned, since the server quits and the connection is closed.

Syntax

Following is the basic syntax of Redis **SHUTDOWN** command.

```
redis 127.0.0.1:6379> SHUTDOWN [NOSAVE] [SAVE]
```

Example

```
redis 127.0.0.1:6379> SHUTDOWN
```

Server Slaveof Command

Redis **SLAVEOF** command can change the replication settings of a slave on the fly. If a Redis server is already acting as a slave, the command SLAVEOF NO ONE will turn off the replication, turning the Redis server into a MASTER. In the proper form SLAVEOF hostname port will make the server a slave of another server listening at the specified hostname and port. If a server is already a slave of some master, SLAVEOF hostname port will stop the replication against the old server and start the synchronization against the new one, discarding the old dataset.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **SLAVEOF** command.

```
redis 127.0.0.1:6379> SLAVEOF host port
```

Server Slowlog Command

The Redis Slow Log is a system to log queries that exceeded a specified execution time. The execution time does not include I/O operations like talking with the client, sending the reply and so forth, but just the time needed to actually execute the command (this is the only stage of command execution where the thread is blocked and cannot serve other requests in the meantime). You can configure the slow log with two parameters: slowlog-log-slower-than tells Redis what is the execution time, in microseconds, to exceed in order for the command to get logged.

Note, that a negative number disables the slow log, while a value of zero forces the logging of every command. slowlog-max-len is the length of the slow log. The minimum value is zero. When a new command is logged and the slow log is already at its maximum length, the oldest one is removed from the queue of logged commands in order to make space. The configuration can be done by editing **redis.conf** or while the server is running using the CONFIG GET and CONFIG SET commands.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **SLOWLOG** command.

```
redis 127.0.0.1:6379> SLOWLOG subcommand [argument]
```

Sample Output

```
redis 127.0.0.1:6379> slowlog get 2
1) 1) (integer) 14
   2) (integer) 1309448221
   3) (integer) 15
   4) 1) "ping"
2) 1) (integer) 13
   2) (integer) 1309448128
   3) (integer) 30
   4) 1) "slowlog"
      2) "get"
      3) "100"
```

Server Sync Command

Redis **SYNC** command is used to sync slave to master.

Return Value

Simple string reply.

Syntax

Following is the basic syntax of Redis **SYNC** command.

```
redis 127.0.0.1:6379> SYNC
```

Example

```
redis 127.0.0.1:6379> SYNC
Entering slave output mode...  (press Ctrl-C to quit)
SYNC with master, discarding 18 bytes of bulk transfer...
SYNC done. Logging commands from master.
"PING"
"PING"
"PING"
"PING"
```

Server Time Command

Redis **TIME** command returns the current server time as a two-items list: a Unix timestamp and the amount of microseconds already elapsed in the current second. Basically, the interface is similar to the one of the gettimeofday system call.

Return Value

A multi bulk reply containing two elements:

- Unix time in seconds.
- Microseconds.

Syntax

Following is the basic syntax of Redis **TIME** command.

```
redis 127.0.0.1:6379> TIME
```

Example

```
redis 127.0.0.1:6379> TIME
```

```
1) "1410856598"
2) "928370"
```

Redis – Advanced

18. Redis – Backup

Redis **SAVE** command is used to create a backup of the current Redis database.

Syntax

Following is the basic syntax of redis **SAVE** command.

```
127.0.0.1:6379> SAVE
```

Example

Following example creates a backup of the current database.

```
127.0.0.1:6379> SAVE

OK
```

This command will create a **dump.rdb** file in your Redis directory.

Restore Redis Data

To restore Redis data, move Redis backup file (dump.rdb) into your Redis directory and start the server. To get your Redis directory, use **CONFIG** command of Redis as shown below.

```
127.0.0.1:6379> CONFIG get dir

1) "dir"
2) "/user/tutorialspoint/redis-2.8.13/src"
```

In the output of the above command **/user/tutorialspoint/redis-2.8.13/src** is the directory, where Redis server is installed.

Bgsave

To create Redis backup, an alternate command **BGSAVE** is also available. This command will start the backup process and run this in the background.

Example

```
127.0.0.1:6379> BGSAVE

Background saving started
```

19. Redis – Security

Redis database can be secured, such that any client making a connection needs to authenticate before executing a command. To secure Redis, you need to set the password in the config file.

Example

Following example shows the steps to secure your Redis instance.

```
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) ""
```

By default, this property is blank, which means no password is set for this instance. You can change this property by executing the following command.

```
127.0.0.1:6379> CONFIG set requirepass "tutorialspoint"
OK
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) "tutorialspoint"
```

After setting the password, if any client runs the command without authentication, then **(error) NOAUTH Authentication required.** error will return. Hence, the client needs to use **AUTH** command to authenticate himself.

Syntax

Following is the basic syntax of **AUTH** command.

```
127.0.0.1:6379> AUTH password
```

Example

```
127.0.0.1:6379> AUTH "tutorialspoint"
OK
127.0.0.1:6379> SET mykey "Test value"
OK
127.0.0.1:6379> GET mykey
"Test value"
```

20. Redis – Benchmarks

Redis benchmark is the utility to check the performance of Redis by running **n** commands simultaneously.

Syntax

Following is the basic syntax of Redis benchmark.

```
redis-benchmark [option] [option value]
```

Example

Following example checks Redis by calling 100000 commands.

```
redis-benchmark -n 100000


PING_INLINE: 141043.72 requests per second
PING_BULK: 142857.14 requests per second
SET: 141442.72 requests per second
GET: 145348.83 requests per second
INCR: 137362.64 requests per second
LPUSH: 145348.83 requests per second
LPOP: 146198.83 requests per second
SADD: 146198.83 requests per second
SPOP: 149253.73 requests per second
LPUSH (needed to benchmark LRANGE): 148588.42 requests per second
LRANGE_100 (first 100 elements): 58411.21 requests per second
LRANGE_300 (first 300 elements): 21195.42 requests per second
LRANGE_500 (first 450 elements): 14539.11 requests per second
LRANGE_600 (first 600 elements): 10504.20 requests per second
MSET (10 keys): 93283.58 requests per second
```

Following is a list of available options in Redis benchmark.

Sr. No.	Options	Description	Default Value
1	**-h**	Specifies server host name	127.0.0.1
2	**-p**	Specifies server port	6379

3	**-s**	Specifies server socket	
4	**-c**	Specifies the number of parallel connections	50
5	**-n**	Specifies the total number of requests	10000
6	**-d**	Specifies data size of SET/GET value in bytes	2
7	**-k**	1=keep alive, 0=reconnect	1
8	**-r**	Use random keys for SET/GET/INCR, random values for SADD	
9	**-p**	Pipeline <numreq> requests	1
10	**-h**	Specifies server host name	
11	**-q**	Forces Quiet to Redis. Just shows query/sec values	
12	**--csv**	Output in CSV format	
13	**-l**	Generates loop, Run the tests forever	
14	**-t**	Only runs the comma-separated list of tests	
15	**-I**	Idle mode. Just opens N idle connections and wait	

Example

Following example shows the multiple usage options in Redis benchmark utility.

```
redis-benchmark -h 127.0.0.1 -p 6379 -t set,lpush -n 100000 -q


SET: 146198.83 requests per second

LPUSH: 145560.41 requests per second
```

21. Redis – Client Connection

Redis accepts clients' connections on the configured listening TCP port and on the Unix socket, if enabled. When a new client connection is accepted, the following operations are performed:

- The client socket is put in non-blocking state since Redis uses multiplexing and non-blocking I/O.
- The TCP_NODELAY option is set in order to ensure that we don't have delays in our connection.
- A readable file event is created so that Redis is able to collect the client queries as soon as new data is available to be read on the socket.

Maximum Number of Clients

In Redis config (redis.conf), there is a property called **maxclients**, which describes the maximum number of clients that can connect to Redis.

Following is the basic syntax of command.

```
config get maxclients


1) "maxclients"

2) "10000"
```

By default, this property is set to 10000 (depending upon the maximum number of file descriptors limit of OS), although you can change this property.

Example

In the following example, we have set the maximum number of clients to 100000, while starting the server.

```
redis-server --maxclients 100000
```

Client Commands

Sr. No.	Command	Description
1	**CLIENT LIST**	Returns the list of clients connected to Redis server
2	**CLIENT SETNAME**	Assigns a name to the current connection
3	**CLIENT GETNAME**	Returns the name of the current connection as set by CLIENT SETNAME

4	**CLIENT PAUSE**	This is a connections control command able to suspend all the Redis clients for the specified amount of time (in milliseconds)
5	**CLIENT KILL**	This command closes a given client connection.

22. Redis – Pipelining

Redis is a TCP server and supports request/response protocol. In Redis, a request is accomplished with the following steps:

- The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response.
- The server processes the command and sends the response back to the client.

Meaning of Pipelining

The basic meaning of pipelining is, the client can send multiple requests to the server without waiting for the replies at all, and finally reads the replies in a single step.

Example

To check the Redis pipelining, just start the Redis instance and type the following command in the terminal.

```
$(echo -en "PING\r\n SET tutorial redis\r\nGET tutorial\r\nINCR visitor\r\nINCR
visitor\r\nINCR visitor\r\n"; sleep 10) | nc localhost 6379


+PONG
+OK
redis
:1
:2
:3
```

In the above example, we will check Redis connection by using **PING** command. We have set a string named **tutorial** with value **redis.** Later, we get that keys value and increment the visitor number three times. In the result, we can see that all commands are submitted to Redis once, and Redis provides the output of all commands in a single step.

Benefits of Pipelining

The benefit of this technique is a drastically improved protocol performance. The speedup gained by pipelining ranges from a factor of five for connections to localhost up to a factor of at least one hundred over slower internet connections.

23. Redis – Partitioning

Partitioning is the process of splitting your data into multiple Redis instances, so that every instance will only contain a subset of your keys.

Benefits of Partitioning

- It allows for much larger databases, using the sum of the memory of many computers. Without partitioning you are limited to the amount of memory that a single computer can support.

- It allows to scale the computational power to multiple cores and multiple computers, and the network bandwidth to multiple computers and network adapters.

Disadvantages of Partitioning

- Operations involving multiple keys are usually not supported. For instance, you can't perform the intersection between two sets if they are stored in the keys that are mapped to different Redis instances.

- Redis transactions involving multiple keys cannot be used.

- The partitioning granuliary is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set.

- When partitioning is used, data handling is more complex. For instance, you have to handle multiple RDB/AOF files, and to get a backup of your data you need to aggregate the persistence files from multiple instances and hosts.

- Adding and removing the capacity can be complex. For instance, Redis Cluster supports mostly transparent rebalancing of data with the ability to add and remove nodes at runtime. However, other systems like client-side partitioning and proxies don't support this feature. A technique called **Presharding** helps in this regard.

Types of Partitioning

There are two types of partitioning available in Redis. Suppose we have four Redis instances, R0, R1, R2, R3 and many keys representing users like user:1, user:2, ... and so forth.

Range Partitioning

Range partitioning is accomplished by mapping ranges of objects into specific Redis instances. Suppose in our example, the users from ID 0 to ID 10000 will go into instance R0, while the users from ID 10001 to ID 20000 will go into instance R1 and so forth.

Hash Partitioning

In this type of partitioning, a hash function (eg. modulus function) is used to convert the key into a number and then the data is stored in different-different Redis instances.

24. Redis – Java

Before you start using Redis in your Java programs, you need to make sure that you have Redis Java driver and Java set up on the machine. You can check our Java tutorial for Java installation on your machine.

Installation

Now, let us see how to set up Redis Java driver.

- You need to download the jar from the path Download jedis.jar. Make sure to download the latest release of it.
- You need to include the **jedis.jar** into your classpath.

Connect to Redis Server

```
import redis.clients.jedis.Jedis;
public class RedisJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //check whether server is running or not
      System.out.println("Server is running: "+jedis.ping());
 }
}
```

Now, let's compile and run the above program to test the connection to Redis server. You can change your path as per your requirement. We are assuming the current version of **jedis.jar** is available in the current path.

```
$javac RedisJava.java
$java RedisJava
Connection to server sucessfully
Server is running: PONG
```

Redis Java String Example

```
import redis.clients.jedis.Jedis;
public class RedisStringJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //set the data in redis string
      jedis.set("tutorial-name", "Redis tutorial");
     // Get the stored data and print it
     System.out.println("Stored string in redis:: "+ jedis.get("tutorial-
name"));
 }
}
```

Now, let's compile and run the above program.

```
$javac RedisStringJava.java
$java RedisStringJava
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis Java List Example

```
import redis.clients.jedis.Jedis;
public class RedisListJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //store data in redis list
      jedis.lpush("tutorial-list", "Redis");
      jedis.lpush("tutorial-list", "Mongodb");
      jedis.lpush("tutorial-list", "Mysql");
     // Get the stored data and print it
     List<String> list = jedis.lrange("tutorial-list", 0 ,5);
     for(int i=0; i<list.size(); i++) {
       System.out.println("Stored string in redis:: "+list.get(i));
      }
   }
}
```

Now, let's compile and run the above program.

```
$javac RedisListJava.java
$java RedisListJava
Connection to server sucessfully
Stored string in redis:: Redis
Stored string in redis:: Mongodb
Stored string in redis:: Mysql
```

Redis Java Keys Example

```
import redis.clients.jedis.Jedis;
public class RedisKeyJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //store data in redis list
     // Get the stored data and print it
     List<String> list = jedis.keys("*");
     for(int i=0; i<list.size(); i++) {
       System.out.println("List of stored keys:: "+list.get(i));
     }
   }
}
```

Now, let's compile and run the above program.

```
$javac RedisKeyJava.java
$java RedisKeyJava
Connection to server sucessfully
List of stored keys:: tutorial-name
List of stored keys:: tutorial-list
```

25. Redis – PHP

Before you start using Redis in your PHP programs, you need to make sure that you have Redis PHP driver and PHP set up on the machine. You can check PHP tutorial for PHP installation on your machine.

Installation

Now, let us check how to set up Redis PHP driver.

You need to download the phpredis from github repository https://github.com/nicolasff/phpredis. Once you've downloaded it, extract the files to phpredis directory. On Ubuntu, install the following extension.

```
cd phpredis
sudo phpize
sudo ./configure
sudo make
sudo make install
```

Now, copy and paste the content of "modules" folder to the PHP extension directory and add the following lines in **php.ini.**

```
extension = redis.so
```

Now, your Redis PHP installation is complete.

Connect to Redis Server

```
<?php
   //Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //check whether server is running or not
   echo "Server is running: ".$redis->ping();
?>
```

When the program is executed, it will produce the following result.

```
Connection to server sucessfully
Server is running: PONG
```

Redis PHP String Example

```
<?php
   //Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //set the data in redis string
   $redis->set("tutorial-name", "Redis tutorial");
   // Get the stored data and print it
   echo "Stored string in redis:: " .$redis→get("tutorial-name");
?>
```

When the above program is executed, it will produce the following result.

```
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis PHP List Example

```
<?php
   //Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //store data in redis list
   $redis->lpush("tutorial-list", "Redis");
   $redis->lpush("tutorial-list", "Mongodb");
   $redis->lpush("tutorial-list", "Mysql");

   // Get the stored data and print it
   $arList = $redis->lrange("tutorial-list", 0 ,5);
   echo "Stored string in redis:: ";
   print_r($arList);
?>
```

When the above program is executed, it will produce the following result.

```
Connection to server sucessfully
Stored string in redis::
```

```
Redis
Mongodb
Mysql
```

Redis PHP Keys Example

```
<?php
   //Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   // Get the stored keys and print it
   $arList = $redis->keys("*");
   echo "Stored keys in redis:: "
   print_r($arList);
?>
```

When the program is executed, it will produce the following result.

```
Connection to server sucessfully
Stored string in redis::
tutorial-name
tutorial-list
```