MARK DRAKE

HOW TO MANAGE A REDIS DATABASE

ISBN 978-0-9997730-7-9

How To Manage a Redis Database

Mark Drake

DigitalOcean, New York City, New York, USA

2020-07

How To Manage a Redis Database

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Introduction

About this Book

Redis is an open-source, in-memory key-value data store known for its flexibility, performance, and broad language support. A NoSQL database, Redis doesn't use structured query language (otherwise known as SQL) to store, manipulate, and retrieve data. Redis instead comes with its own set of commands for managing and accessing data.

The chapters in this book cover a broad range of Redis commands, but they generally focus on connecting to a Redis database, managing a variety of data types, and troubleshooting and debugging problems, along with a few other more specific functions. You're encouraged to jump to whichever chapter is relevant to the task you're trying to complete.

Motivation for this Book

Every available Redis command is covered and thoroughly explained in the official Redis command documentation, as well as in numerous other resources in print and online. However, many resources tend to silo each command off from one another with little tying them together.

This book aims to provide an approachable introduction to Redis concepts by outlining many of the key-value store's commands so readers can learn their patterns and syntax, thus building up readers' understanding gradually. This book covers topics ranging from connecting to a Redis data store, to managing a variety of Redis data types, troubleshooting errors, and more.

Learning Goals and Outcomes

The goal for this book is to serve as an introduction to Redis for those interested in getting started with it, or key-value stores in general. For more experienced users, this book can function as a collection of helpful cheat sheets and in-depth reference. Each chapter is self-contained and can be followed independently of the others. By reading through and following along with each chapter, you'll become acquainted with many of Redis's most widely used commands, which will help you as you begin to build applications that take advantage of its power and speed.

How To Use this Book

The chapters of this book are written as cheat sheets with self-contained examples. We encourage you to jump to whichever chapter is relevant to the task you're trying to complete.

The commands shown in this book were tested on an Ubuntu 18.04 server running Redis version **4.0.9**. To set up a similar environment, you can follow Step 1 of our guide on How To Install and Secure Redis on Ubuntu 18.04. We will demonstrate how these commands behave by running them with `redis-cli`, the Redis command line interface. Note that if you're using a different Redis interface — Redli, for example — the exact output of certain commands may differ.

Alternatively, you could provision a managed Redis database instance to explore Redis's functionality. However, depending on the level of control allowed by your database provider, some commands in this guide may not work as described. To use this book with a DigitalOcean Managed Database, follow our Managed Databases product documentation. Then,

you must either install Redli or set up a TLS tunnel in order to connect to the Managed Database over TLS.

How To Connect to a Redis Database

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. Whether you've installed Redis locally or you're working with a remote instance, you need to connect to it in order to perform most operations. In this tutorial we will go over how to connect to Redis from the command line, how to authenticate and test your connection, as well as how to close a Redis connection.

Connecting to Redis

If you have `redis-server` installed locally, you can connect to the Redis instance with the `redis-cli` command:

```
redis-cli
```

This will take you into `redis-cli`'s interactive mode which presents you with a read-eval-print loop (REPL) where you can run Redis's built-in commands and receive replies.

In interactive mode, your command line prompt will change to reflect your connection. In this example and others throughout this guide, the prompt indicates a connection to a Redis instance hosted locally (`127.0.0.1`) and accessed over Redis's default port (`6379`):

The alternative to running Redis commands in interactive mode is to run them as arguments to the `redis-cli` command, like so:

```
redis-cli redis_command
```

If you want to connect to a remote Redis datastore, you can specify its host and port numbers with the `-h` and `-p` flags, respectively. Also, if you've configured your Redis database to require a password, you can include the `-a` flag followed by your password in order to authenticate:

```
redis-cli -h host -p port_number -a password
```

If you've set a Redis password, clients will be able to connect to Redis even if they don't include the `-a` flag in their `redis-cli` command. However, they won't be able to add, change, or query data until they authenticate. To authenticate after connecting, use the `auth` command followed by the password:

```
auth password
```

If the password passed to `auth` is valid, the command will return `OK`. Otherwise, it will return an error.

If you're working with a managed Redis database, your cloud provider may give you a URI that begins with `redis://` or `rediss://` which you can use to access your datastore. If the connection string begins with `redis://`, you can include it as an argument to `redis-cli` to connect.

However, if you have a connection string that begins with `rediss://`, that means your managed database requires connections over TLS/SSL. `redis-cli` does not support TLS connections, so you'll need to use a different tool that supports the `rediss` protocol in order to connect with the URI. For DigitalOcean Managed Databases, which require connections to be made over TLS, we recommend using Redli to access the Redis instance.

Use the following syntax to connect to a database with Redli. Note that this example includes the `--tls` option, which specifies that the

connection should be made over TLS, and the `-u` flag, which declares that the following argument will be a connection URI:

```
redli --tls -u rediss://connection_URI
```

If you've attempted to connect to an unavailable instance, `redis-cli` will go into disconnected mode. The prompt will reflect this:

Redis will attempt to reestablish the connection every time you run a command when it's in a disconnected state.

Testing Connections

The `ping` command is useful for testing whether the connection to a database is alive. Note that this is a Redis-specific command and is different from the `ping` networking utility. However, the two share a similar function in that they're both used to check a connection between two machines.

If the connection is up and no arguments are included, the `ping` command will return `PONG`:

```
ping
```

Output

```
PONG
```

If you provide an argument to the `ping` command, it will return that argument instead of `PONG` if the connection is successful:

```
ping "hello Redis!"
```

Output

```
"hello Redis!"
```

If you run `ping` or any other command in disconnected mode, you will see an output like this:

```
ping
```

Output

```
Could not connect to Redis at host:port:
Connection refused
```

Note that `ping` is also used by Redis internally to measure latency.

Disconnecting from Redis

To disconnect from a Redis instance, use the `quit` command:

```
quit
```

Running `exit` will also exit the connection:

```
exit
```

Both `quit` and `exit` will close the connection, but only as soon as all pending replies have been written to clients.

Conclusion

This guide details a number of commands used to establish, test, and close connections to a Redis server. If there are other related commands, arguments, or procedures you'd like to see in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Redis Databases and Keys

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. A key-value data store is a type of NoSQL database in which keys serve as unique identifiers for their associated values. Any given Redis instance includes a number of databases, each of which can hold many different keys of a variety of data types. In this tutorial, we will go over how to select a database, move keys between databases, and manage and delete keys.

Managing Databases

Out of the box, a Redis instance supports 16 logical databases. These databases are effectively siloed off from one another, and when you run a command in one database it doesn't affect any of the data stored in other databases in your Redis instance.

Redis databases are numbered from `0` to `15` and, by default, you connect to database `0` when you connect to your Redis instance. However, you can change the database you're using with the `select` command after you connect:

```
select 15
```

If you've selected a database other than `0`, it will be reflected in the `redis-cli` prompt:

To swap all the data held in one database with the data held in another, use the `swapdb` command. The following example will swap the data held

in database `6` with that in database `8`, and any clients connected to either database will be able to see changes immediately:

```
swapdb 6 8
```

`swapdb` will return `OK` if the swap is successful.

If you want to move a key to a different Redis instance, you can run `migrate`. This command ensures the key exists on the target instance before deleting it from the source instance. When you run `migrate`, the command must include the following elements in this order:

- The hostname or IP address of the destination database
- The target database's port number
- The name of the key you want to migrate
- The database number where you want to store the key on the destination instance
- A timeout, in milliseconds, which defines the maximum amount of idle communication time between the two machines. Note that this isn't a time limit for the operation, just that the operation should always make some level of progress within the defined length of time

To illustrate:

```
migrate 203.0.113.0 6379 key_1 7 8000
```

Additionally, `migrate` allows the following options which you can add after the timeout argument:

- `COPY`: Specifies that the key should not be deleted from the source instance
- `REPLACE`: Specifies that if the key already exists on the destination, the `migrate` operation should delete and replace it

- `KEYS`: Instead of providing a specific key to migrate, you can enter an empty string (`""`) and then use the syntax from the `keys` command to migrate any key that matches a pattern. For more information on how `keys` works, see our tutorial on How To Troubleshoot Issues in Redis.

Managing Keys

There are a number of Redis commands that are useful for managing keys regardless of what type of data they hold. We'll go over a few of these in this section.

`rename` will rename the specified key. If it's successful, it will return `OK`:

```
rename old_key new_key
```

You can use `randomkey` to return a random key from the currently selected database:

```
randomkey
```

Output

```
"any_key"
```

Use `type` to determine what type of data the given key holds. This command's output can be either `string`, `list`, `hash`, `set`, `zset`, or `stream`:

```
type key_1
```

Output

```
"string"
```

If the specified key doesn't exist, `type` will return `none` instead.

You can move an individual key to another database in your Redis instance with the `move` command. `move` takes the name of a key and the database where you want to move the key as arguments. For example, to move the key `key_1` to database `8`, you would run the following:

```
move key_1 8
```

`move` will return `OK` if moving the key was successful.

Deleting Keys

To delete one or more keys of any data type, use the `del` command followed by one or more keys that you want to delete:

```
del key_1 key_2
```

If this command deletes the key(s) successfully it will return `(integer) 1`. Otherwise, it will return `(integer) 0`.

The `unlink` command performs a similar function as `del`, with the difference being that `del` blocks the client as the server reclaims the memory taken up by the key. If the key being deleted is associated with a small object, the amount of time it takes for `del` to reclaim the memory is very small and the blocking time may not even be noticeable.

However, it can become inconvenient if, for example, the key you're deleting is associated with many objects, such as a hash with thousands or millions of fields. Deleting such a key can take a noticeably long time, and

you'll be blocked from performing any other operations until it's fully removed from the server's memory.

`unlink`, however, first determines the cost of deallocating the memory taken up by the key. If it's small then `unlink` functions the same way as `del` by the key immediately while also blocking the client. However, if there's a high cost to deallocate memory for a key, `unlink` will delete the key asynchronously by creating another thread and incrementally reclaim memory in the background without blocking the client:

```
unlink key_1
```

Since it runs in the background, it's generally recommended that you use `unlink` to remove keys from your server to reduce errors on your clients, though `del` will also suffice in many cases.

Warning: The following two commands are considered dangerous. The `flushdb` and `flushall` commands will irreversibly delete all the keys in a single database and all the keys in every database on the Redis server, respectively. We recommend that you only run these commands if you are absolutely certain that you want to delete all the keys in your database or server.

It may be in your interest to rename these commands to something with a lower likelihood of being run accidentally.

To delete all the keys in the selected database, use the `flushdb` command:

```
flushdb
```

To delete all the keys in every database on a Redis server (including the currently selected database), run `flushall`:

```
flushall
```

Both `flushdb` and `flushall` accept the `async` option, which allows you to delete all the keys on a single database or every database in the cluster asynchronously. This allows them to function similarly to the `unlink` command, and they will create a new thread to incrementally free up memory in the background.

Backing Up Your Database

To create a backup of the currently selected database, you can use the `save` command:

```
save
```

This will export a snapshot of the current dataset as an `.rdb` file, which is a database dump file that holds the data in an internal, compressed serialization format.

`save` runs synchronously and will block any other clients connected to the database. Hence, the `save` command documentation recommends that this command should almost never be run in a production environment. Instead, it suggests using the `bgsave` command. This tells Redis to fork the database: the parent will continue to serve clients while the child process saves the database before exiting:

```
bgsave
```

Note that if clients add or modify data while the `bgsave` operation is occurring, these changes won't be captured in the snapshot.

You can also edit the Redis configuration file to have Redis save a snapshot automatically (known as snapshotting or RDB mode) after a certain amount of time if a minimum number of changes were made to the

database. This is known as a save point. The following save point settings are enabled by default in the `redis.conf` file:

/etc/redis/redis.conf

```
. . .
save 900 1
save 300 10
save 60 10000
. . .
dbfilename "nextfile.rdb"
. . .
```

With these settings, Redis will export a snapshot of the database to the file defined by the `dbfilename` parameter every 900 seconds if at least 1 key is changed, every 300 seconds if at least 10 keys are changed, and every 60 seconds if at least 10000 keys are changed.

You can use the `shutdown` command to back up your Redis data and then close your connection. This command will block every client connected to the database and then perform a `save` operation if at least one save point is configured, meaning that it will export the database in its current state to an `.rdb` file while preventing clients from making any changes.

Additionally, the `shutdown` command will flush changes to Redis's append-only file before quitting if append-only mode is enabled. The append-only file mode (AOF) involves creating a log of every write operation on the server in a file ending in `.aof` after every snapshot. AOF

and RDB modes can be enabled on the same server, and using both persistence methods is an effective way to back up your data.

In short, the `shutdown` command is essentially a blocking `save` command that also flushes all recent changes to the append-only file and closes the connection to the Redis instance:

Warning: The `shutdown` command is considered dangerous. By blocking your Redis server's clients, you can make your data unavailable to users and applications that depend on it. We recommend that you only run this command if you are testing out Redis's behavior or you are absolutely certain that you want to block all your Redis server's clients.

In fact, it may be in your interest to rename this command to something with a lower likelihood of being run accidentally.

```
shutdown
```

If you've not configured any save points but still want Redis to perform a `save` operation, append the `save` option to the `shutdown command:

```
shutdown save
```

If you have configured at least one save point but you want to shut down the Redis server without performing a save, you can add the `nosave` argument to the command:

```
shutdown nosave
```

Note that the append-only file can grow to be very long over time, but you can configure Redis to rewrite the file based on certain variables by editing the `redis.conf` file. You can also instruct Redis to rewrite the append-only file by running the `bgrewriteaof` command:

```
bgrewriteaof
```

`bgrewriteaof` will create the shortest set of commands needed to bring the database back to its current state. As this command's name implies, it will run in the background. However, if another persistence command is running in a background process already, that command must finish before Redis will execute `bgrewriteaof`.

Conclusion

This guide details a number of commands used to manage databases and keys. If there are other related commands, arguments, or procedures you'd like to see in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Replicas and Clients in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. One of its most sought-after features is its support for replication: any Redis server can replicate its data to any number of replicas, allowing for high read scalability and strong data redundancy. Additionally, Redis was designed to allow many clients (up to 10000, by default) to connect and interact with data, making it a good choice for cases where many users need access to the same dataset.

This tutorial goes over the commands used to manage Redis clients and replicas.

Note: The Redis project uses the terms “master” and “slave” in its documentation and in various commands to identify different roles in replication, though the project’s contributors are taking steps to change this language in cases where it doesn’t cause compatibility issues. DigitalOcean generally prefers to use the alternative terms “primary” and “replica”.

This guide will default to “primary” and “replica” whenever possible, but note that there are a few instances where the terms “master” and “slave” unavoidably come up.

Managing Replicas

One of Redis's most distinguishing features is its built-in replication. When using replication, Redis creates exact copies of the primary instance. These secondary instances reconnect to the primary any time their connections break and will always aim to remain an exact copy of the primary.

If you're not sure whether the Redis instance you're currently connected to is a primary instance or a replica, you can check by running the `role` command:

```
role
```

This command will return either `master` or `slave`, or potentially `sentinel` if you're using Redis Sentinel.

To designate a Redis instance as a replica of another instance on the fly, run the `replicaof` command. This command takes the intended primary server's hostname or IP address and port as arguments:

```
replicaof hostname_or_IP port
```

If the server was already a replica of another primary, it will stop replicating the old server and immediately start synchronizing with the new one. It will also discard the old dataset.

To promote a replica back to being a primary, run the following `replicaof` command:

```
replicaof no one
```

This will stop the instance from replicating the primary server, but will not discard the dataset it has already replicated. This syntax is useful in cases where the original primary fails. After running `replicaof no one` on a replica of the failed primary, the former replica can be used as the new primary and have its own replicas as a failsafe.

Note: Prior to version 5.0.0, Redis instead included a version of this command named `slaveof`.

Managing Clients

A client is any machine or software that connects to a server in order to access a service. Redis comes with several commands that help with tracking and managing client connections.

The `client list` command returns a set of human-readable information about current client connections:

```
client list
```

Output

```
"id=18165 addr=[2001:db8:0:0::12]:47460 fd=7
name=jerry age=72756 idle=0 flags=N db=0 sub=0
psub=0 multi=-1 qbuf=0 qbuf-free=0 obl=0 oll=0
omem=0 events=r cmd=ping
id=18166 addr=[2001:db8:0:1::12]:47466 fd=8 name=
age=72755 idle=5 flags=N db=0 sub=0 psub=0
multi=-1 qbuf=0 qbuf-free=0 obl=0 oll=0 omem=0
events=r cmd=info
id=19381 addr=[2001:db8:0:2::12]:54910 fd=9 name=
age=9 idle=0 flags=N db=0 sub=0 psub=0 multi=-1
qbuf=26 qbuf-free=32742 obl=0 oll=0 omem=0
events=r cmd=client
"
```

Here is what each of these fields mean:

- `id`: a unique 64-bit client ID
- `name`: the name of the client connection, as defined by a prior `client setname` command
- `addr`: the address and port from which the client is connecting
- `fd`: the file descriptor that corresponds to the socket over which the client is connecting
- `age`: the total duration of the client connection, in seconds
- `flags`: a set of one or more single-character flags that provide more granular detail about the clients; see the `client list` command documentation for more details
- `db`: the current database ID number that the client is connected to (can be from `0` to `15`)
- `sub`: the number of channels the client is subscribed to
- `psub`: the number of the client's pattern-matching subscriptions
- `mutli`: the number of commands the client has queued in a transaction (will show `-1` if the client hasn't begun a transaction or `0` if it has only started a transaction and not queued any commands)
- `qbuf`: the client's query buffer length, with `0` meaning it has no pending queries
- `qbuf-free`: the amount of free space in the client's query buffer, with `0` meaning that the query buffer is full
- `obl`: the client's output buffer length
- `oll`: the length of the client's output list, where replies are queued when its buffer is full
- `omem`: the memory used by the client's output buffer

- `events`: the client's file descriptor events, these can be `r` for "readable", `w` for "writable," or both
- `cmd`: the last command run by the client

Setting client names is useful for debugging connection leaks in whatever application is using Redis. Every new connection starts without an assigned name, but `client setname` can be used to create one for the current client connection. There's no limit to how long client names can be, although Redis usually limits string lengths to 512 MB. Note, though, that client names cannot include spaces:

```
client setname elaine
```

To retrieve the name of a client connection, use the `client getname` command:

```
client getname
```

Output

```
"elaine"
```

To retrieve a client's connection ID, use the `client id` command:

```
client id
```

Output

```
(integer) "19492"
```

Redis client IDs are never repeated and are monotonically incremental. This means that if one client has an ID greater than another, then it was established at a later time.

Blocking Clients and Closing Client Connections

Replication systems are typically described as being either synchronous or asynchronous. In synchronous replication, whenever a client adds or changes data it must receive some kind of acknowledgement from a certain number of replicas for the change to register as having been committed. This helps to prevent nodes from having data conflicts but it comes at a cost of latency, since the client must wait to perform another operation until it has heard back from a certain number of replicas.

In asynchronous replication, on the other hand, the client sees a confirmation that the operation is finished as soon as the data is written to local storage. There can, however, be a lag between this and when the replicas actually write the data. If one of the replicas fails before it can write the change, that write will be lost forever. So while asynchronous replication allows clients to continue performing operations without the latency caused by waiting for replicas, it can lead to data conflicts between nodes and may require extra work on the part of the database administrator to resolve those conflicts.

Because of its focus on performance and low latency, Redis implements asynchronous replication by default. However, you can simulate synchronous replication with the `wait` command. `wait` blocks the current client connection for a specified amount of time (in milliseconds) until all the previous write commands are successfully transferred and accepted by a specified number of replicas. This command uses the following syntax:

```
wait number_of_replicas number_of_milliseconds
```

For example, if you want to block your client connection until all the previous writes are registered by at least 3 replicas within a 30 millisecond timeout, your `wait` syntax would look like this:

```
wait 3 30
```

The `wait` command returns an integer representing the number of replicas that acknowledged the write commands, even if not every replica does so:

Output

```
2
```

To unblock a client connection that has been previously blocked, whether from a `wait`, `brpop`, or `xread` command, you can run a `client unblock` command with the following syntax:

```
client unblock client_id
```

To temporarily suspend every client currently connected to the Redis server, you can use the `client pause` command. This is useful in cases where you need to make changes to your Redis setup in a controlled way. For example, if you're promoting one of your replicas to be the primary instance, you might pause every client beforehand so you can promote the replica and have the clients connect to it as the new primary without losing any write operations in the process.

The `client pause` command requires you to specify the amount of time (in milliseconds) you'd like to suspend the clients. The following example will suspend all clients for one second:

```
client pause 1000
```

The `client kill` syntax allows you to close a single connection or a set of specific connections based on a number of different filters. The syntax looks like this:

```
client kill filter_1 value_1 ... filter_n value_n
```

In Redis versions 2.8.12 and later, the following filters are available:

- `addr`: allows you to close a client connection from a specified IP address and port
- `client-id`: allows you to close a client connection based on its unique ID field
- `type`: closes every client of a given type, which can be either `normal`, `master`, `slave`, or `pubsub`
- `skipme`: the value options for this filter are `yes` and `no`:
    - if `no` is specified, the client calling the `client kill` command will not get skipped, and will be killed if the other filters apply to it
    - if `yes` is specified, the client running the command will be skipped and the kill command will have no effect on the client. `skipme` is always `yes` by default

Conclusion

This guide details a number of commands used to manage Redis clients and replicas. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Strings in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. In Redis, strings are the most basic type of value you can create and manage. This tutorial provides an overview of how to create and retrieve strings, as well as how to manipulate the values held by string keys.

Creating and Managing Strings

Keys that hold strings can only hold one value; you cannot store more than one string in a single key. However, strings in Redis are binary-safe, meaning a Redis string can hold any kind of data, from alphanumeric characters to JPEG images. The only limit is that strings must be 512 MB long or less.

To create a string, use the `set` command. For example, the following `set` command creates a key named `key_Welcome1` that holds the string `"Howdy"`:

```
set key_Welcome1 "Howdy"
```

Output

```
OK
```

To set multiple strings in one command, use `mset`:

```
mset key_Welcome2 "there" key_Welcome3 "partners,"
```

You can also use the `append` command to create strings:

```
append key_Welcome4 "welcome to Texas"
```

If the string was created successfully, `append` will output an integer equal to how many characters the string includes:

Output

```
(integer) 16
```

Note that `append` can also be used to change the contents of strings. See the section on manipulating strings for details on this.

Retrieving Strings

To retrieve a string, use the `get` command:

```
get key_Welcome1
```

Output

```
"Howdy"
```

To retrieve multiple strings with one command, use `mget`:

```
mget key_Welcome1 key_Welcome2 key_Welcome3
key_Welcome4
```

Output

```
1) "Howdy"
2) "there"
3) "partners,"
4) "welcome to Texas"
```

For every key passed to `mget` that doesn't hold a string value or doesn't exist at all, the command will return `nil`.

Manipulating Strings

If a string is made up of an integer, you can run the `incr` command to increase it by one:

```
set key_1 3
incr key_1
```

Output

```
(integer) 4
```

Similarly, you can use the `incrby` command to increase a numeric string's value by a specific increment:

```
incrby key_1 16
```

Output

```
(integer) 20
```

The `decr` and `decrby` commands work the same way, but they decrease the integer stored in a numeric string:

```
decr key_1
```

Output

```
(integer) 19
```

```
decrby key_1 16
```

Output

```
(integer) 3
```

If an alphabetic string already exists, `append` will append the value onto the end of the existing value and return the new length of the string. To illustrate, the following command appends `", y'all"` to the string held by the key `key_Welcome4`, so now the string will read `"welcome to Texas, y'all"`:

```
append key_Welcome4 ", y'all"
```

Output

```
(integer) 15
```

You can also append integers to a string holding a numeric value. The following example appends `45` to `3`, the integer held in `key_1`, so it will then hold `345`. In this case, `append` will also return the new length of the string, rather than its new value:

```
append key_1 45
```

Output

```
(integer) 3
```

Because this key still only holds a numeric value, you can perform the `incr` and `decr` operations on it. You can also append alphabetic characters to an integer string, but if you do this then running `incr` and

`decr` on the string will produce an error as the string value is no longer an integer.

Conclusion

This guide details a number of commands used to create and manage strings in Redis. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Lists in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. In Redis, a list is a collection of strings sorted by insertion order, similar to linked lists. This tutorial covers how to create and work with elements in Redis lists.

Creating Lists

A key can only hold one list, although any list can hold over four billion elements. Redis reads lists from left to right, and you can add new list elements to the head of a list (the "left" end) with the `lpush` command or the tail (the "right" end) with `rpush`. You can also use `lpush` or `rpush` to create a new list:

```
lpush key value
```

Both commands will output an integer showing how many elements are in the list. To illustrate, run the following commands to create a list containing the dictum "I think therefore I am":

```
lpush key_philosophy1 "therefore"
lpush key_philosophy1 "think"
rpush key_philosophy1 "I"
lpush key_philosophy1 "I"
rpush key_philosophy1 "am"
```

The output from the last command will read:

Output

```
(integer) 5
```

Note that you can add multiple list elements with a single `lpush` or `rpush` statement:

```
rpush key_philosophy1 "-" "Rene" "Decartes"
```

The `lpushx` and `rpushx` commands are also used to add elements to lists, but will only work if the given list already exists. If either command fails, it will return `(integer) 0`:

```
rpushx key_philosophy2 "Happiness" "is" "the"
"highest" "good" "-" "Aristotle"
```

Output

```
(integer) 0
```

To change an existing element in a list, run the `lset` command followed by the key name, the index of the element you want to change, and the new value:

```
lset key_philosophy1 5 "sayeth"
```

If you try adding a list element to an existing key that does not contain a list, it will lead to a clash in data types and return an error. For example, the following `set` command creates a key holding a string, so the following attempt to add a list element to it with `lpush` will fail:

```
set key_philosophy3 "What is love?"
lpush key_philosophy3 "Baby don't hurt me"
```

Output

```
(error) WRONGTYPE Operation against a key holding
the wrong kind of value
```

It isn't possible to convert Redis keys from one data type to another, so to turn `key_philosophy3` into a list you would need to delete the key and start over with an `lpush` or `rpush` command.

Retrieving Elements from a List

To retrieve a range of items in a list, use the `lrange` command followed by a start offset and a stop offset. Each offset is a zero-based index, meaning that `0` represents the first element in the list, `1` represents the next, and so on.

The following command will return all the elements from the example list created in the previous section:

```
lrange key_philosophy1 0 7
```

Output

```
1) "I"
2) "think"
3) "therefore"
4) "I"
5) "am"
6) "sayeth"
7) "Rene"
8) "Decartes"
```

The offsets passed to `lrange` can also be negative numbers. When used in this case, `-1` represents the final element in the list, `-2` represents the second-to-last element in the list, and so on. The following example returns the last three elements of the list held in `key_philosophy1`:

```
lrange key_philosophy1 -3 -1
```

Output

```
1) "I"
2) "am"
3) "sayeth"
```

To retrieve a single element from a list, you can use the `lindex` command. However, this command requires you to supply the element's index as an argument. As with `lrange`, the index is zero-based, meaning that the first element is at index `0`, the second is at index `1`, and so on:

```
lindex key_philosophy1 4
```

Output

```
"am"
```

To find out how many elements are in a given list, use the `llen` command, which is short for "list length":

```
llen key_philosophy1
```

Output

```
(integer) 8
```

If the value stored at the given key does not exist, `llen` will return an error.

Removing Elements from a List

The `lrem` command removes the first of a defined number of occurrences that match a given value. To experiment with this, create the following list:

```
rpush key_Bond "Never" "Say" "Never" "Again" "You"
"Only" "Live" "Twice" "Live" "and" "Let" "Die"
"Tomorrow" "Never" "Dies"
```

The following `lrem` example will remove the first occurence of the value `"Live"`:

```
lrem key_Bond 1 "Live"
```

This command will output the number of elements removed from the list:

Output

```
(integer) 1
```

The number passed to an `lrem` command can also be negative. The following example will remove the last two occurences of the value `"Never"`:

```
lrem key_Bond -2 "Never"
```

Output

```
(integer) 2
```

The `lpop` command removes and returns the first, or “leftmost” element from a list:

```
lpop key_Bond
```

Output

```
"Never"
```

Likewise, to remove and return the last or “rightmost” element from a list, use `rpop`:

```
rpop key_Bond
```

Output

```
"Dies"
```

Redis also includes the `rpoplpush` command, which removes the last element from a list and pushes it to the beginning of another list:

```
rpoplpush key_Bond key_AfterToday
```

Output

```
"Tomorrow"
```

If the source and destination keys passed to `rpoplpush` command are the same, it will essentially rotate the elements in the list.

Conclusion

This guide details a number of commands that you can use to create and manage lists in Redis. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Hashes in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. A Redis hash is a data type that represents a mapping between a string field and a string value. Hashes can hold many field-value pairs and are designed to not take up much space, making them ideal for representing data objects. For example, a hash might represent a customer, and include fields like `name`, `address`, `email`, or `customer_id`.

This tutorial will go over how to manage hashes in Redis, from creating them to retrieving and deleting the data held within a hash.

Creating Hashes

To create a hash, run the `hset` command. This command accepts the name of the hash key, the field string, and corresponding value string as arguments:

```
hset poet:Verlaine nationality French
```

Note: In this example and the following ones, `poet:Verlaine` is the hash key. Dots, dashes, and colons are commonly used to make multi-word keys and fields more readable. It's helpful to make sure that your keys follow a consistent and easily readable format.

`hset` returns `(integer) 1` if the field specified is a new field and the value was set correctly:

Output

```
(integer) 1
```

If, however, you fail to include a value, field, or name for the hash key, `hset` will return an error.

Also, note that `hset` will overwrite the contents of the hash if it already exists:

```
hset poet:Verlaine nationality Francais
```

If the field already exists and its value was updated successfully, `hset` will return `(integer) 0`:

Output

```
(integer) 0
```

You can also use `hsetnx` to add fields to hashes, but it will only work if the field does not yet exist. If the specified field does already exist, the `hsetnx` won't have any effect and will return `(integer) 0`:

```
hsetnx poet:Verlaine nationality French
```

Output

```
(integer) 0
```

To set multiple field/value pairs to a given set, use the `hmset` command followed by the corresponding field/value strings:

```
hmset poet:Verlaine born 1844 died 1896 genre
Decadent
```

`hmset` will just return `OK` if it was successful.

Retrieving Information from Hashes

You can determine if a field exists for a given hash with the `hexists` command:

```
hexists poet:Verlaine nationality
```

`hexists` will return `(integer) 1` if the field does exist, and `(integer) 0` if it doesn't.

To return a field's value, run the `hget` command followed by the hash key and the field whose value you want to retrieve:

```
hget poet:Verlaine nationality
```

Output

```
"Francais"
```

`hmget` uses the same syntax, but can return the values of multiple fields

```
hmget poet:Verlaine born died
```

Output

```
1) "1844"
2) "1896"
```

If the hash you pass to `hget` or `hmget` does not exist, both commands will return `(nil)`:

```
hmget poet:Dickinson born died
```

Output

```
1) (nil)
2) (nil)
```

To obtain a list of all the fields held within a certain hash, run the `hkeys` command:

```
hkeys poet:Verlaine
```

Output

```
1) "nationality"
2) "born"
3) "died"
4) "genre"
```

Conversely, run `hvals` to retrieve a list of values held within a hash:

```
hvals poet:Verlaine
```

Output

```
1) "French"
2) "1844"
3) "1896"
4) "Decadent"
```

To return a list of every field held by a hash and their associated values, run `hgetall`:

```
hgetall poet:Verlaine
```

Output

```
1) "nationality"
2) "French"
3) "born"
4) "1844"
5) "died"
6) "1896"
7) "genre"
8) "Decadent"
```

You can find the number of fields in a hash by running `hlen`, which stands for “hash length”:

```
hlen poet:Verlaine
```

Output

```
(integer) 4
```

You can find the length of the value string associated with a field with `hstrlen`, which stands for “hash string length”:

```
hstrlen poet:Verlaine nationality
```

Output

```
(integer) 8
```

`hlen` will return `(integer) 0` if the hash does not exist.

Removing Fields from Hashes

To delete a field from a hash, run the `hdel` command. `hdel` can accept multiple fields as arguments, and will return an integer indicating how many fields were removed from the hash:

```
hdel poet:Verlaine born died
```

Output

```
(integer) 2
```

If you pass a field that does not exist to `hdel`, it will ignore that field but delete any other existing fields you specify.

Conclusion

This guide details a number of commands used to create and manage hashes in Redis. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Sets in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. Sets in Redis are collections of strings stored at a given key. When held in a set, an individual record value is referred to as a member. Unlike lists, sets are unordered and do not allow repeated values.

This tutorial explains how to create sets, retrieve and remove members, and compare the members of different sets.

Creating Sets

The `sadd` command allows you to create a set and add one or more members to it. The following example will create a set at a key named `key_horror` with the members `"Frankenstein"` and `"Godzilla"`:

```
sadd key_horror "Frankenstein" "Godzilla"
```

If successful, `sadd` will return an integer showing how many members it added to the set:

Output

```
(integer) 2
```

If you try adding members of a set to a key that's already holding a non-set value, it will return an error. The first command in this block creates a list named `key_action` with one element, `"Shaft"`. The next

command tries to add a set member, `"Shane"`, to the list, but this produces an error because of the clashing data types:

```
rpush key_action "Shaft"
sadd key_action "Shane"
```

Output

```
(error) WRONGTYPE Operation against a key holding
the wrong kind of value
```

Note that sets don't allow more than one occurrence of the same member:

```
sadd key_comedy "It's" "A" "Mad" "Mad" "Mad" "Mad"
"Mad" "World"
```

Output

```
(integer) 4
```

Even though this `sadd`command specifies eight members, it discards four of the duplicate `"Mad"` members resulting in a set size of 4.

Retrieving Members from Sets

In this section, we'll go over a number of Redis commands that return information about the members held in a set. To practice the commands outlined here, run the following command, which will create a set with six members at a key called `key_stooges`:

```
sadd key_stooges "Moe" "Larry" "Curly" "Shemp"
"Joe" "Curly Joe"
```

To return every member from a set, run the `smembers` command followed by the key you want to inspect:

```
smembers key_stooges
```

Output

```
1) "Curly"
2) "Moe"
3) "Larry"
4) "Shemp"
5) "Curly Joe"
6) "Joe"
```

To check if a specific value is a member of a set, use the `sismember` command:

```
sismember key_stooges "Harpo"
```

If the element `"Harpo"` is a member of the `key_stooges` set, `sismember` will return `1`. Otherwise, it will return `0`:

Output

```
(integer) 0
```

To see how many members are in a given set (in other words, to find the cardinality of a given set), run `scard`:

```
scard key_stooges
```

Output

```
(integer) 6
```

To return a random element from a set, run `srandmember`:

```
srandmember key_stooges
```

Output

```
"Larry"
```

To return multiple random, distinct elements from a set, you can follow the `srandmember` command with the number of elements you want to retrieve:

```
srandmember key_stooges 3
```

Output

```
1) "Larry"
2) "Moe"
3) "Curly Joe"
```

If you pass a negative number to `srandmember`, the command is allowed to return the same element multiple times:

```
srandmember key_stooges -3
```

Output

```
1) "Shemp"
2) "Curly Joe"
3) "Curly Joe"
```

The random element function used in `srandmember` is not perfectly random, although its performance improves in larger data sets. See the command's official documentation for more details.

Removing Members from Sets

Redis comes with three commands used to remove members from a set: `spop`, `srem`, and `smove`.

`spop` randomly selects a specified number of members from a set and returns them, similar to `srandmember`, but then deletes them from the set. It accepts the name of the key containing a set and the number of members to remove from the set as arguments. If you don't specify a number, `spop` will default to returning and removing a single value.

The following example command will remove and return two randomly-selected elements from the `key_stooges` set created in the previous section:

```
spop key_stooges 2
```

Output

```
1) "Shemp"
2) "Larry"
```

`srem` allows you to remove one or more specific members from a set, rather than random ones:

```
srem key_stooges "Joe" "Curly Joe"
```

Instead of returning the members removed from the set, `srem` returns an integer showing how many members were removed:

Output

```
(integer) 2
```

Use `smove` to move a member from one set to another. This command accepts as arguments the source set, the destination set, and the member to move, in that order. Note that `smove` only allows you to move one member at a time:

```
smove key_stooges key_jambands "Moe"
```

If the command moves the member successfully, it will return `(integer) 1`:

Output

```
(integer) 1
```

If `smove` fails, it will instead return `(integer) 0`. Note that if the destination key does not already exist, `smove` will create it before moving the member into it.

Comparing Sets

Redis also provides a number of commands that find the differences and similarities between sets. To demonstrate how these work, this section will reference three sets named `presidents`, `kings`, and `beatles`. If you'd like to try out the commands in this section yourself, create these sets and populate them using the following `sadd` commands:

```
sadd presidents "George" "John" "Thomas" "James"
sadd kings "Edward" "Henry" "John" "James"
```

```
"George"
sadd beatles "John" "George" "Paul" "Ringo"
```

`sinter` compares different sets and returns the set intersection, or values that appear in every set:

```
sinter presidents kings beatles
```

Output

```
1) "John"
2) "George"
```

`sinterstore` performs a similar function, but instead of returning the intersecting members it creates a new set at the specified destination containing these intersecting members. Note that if the destination already exists, `sinterstore` will overwrite its contents:

```
sinterstore new_set presidents kings beatles
smembers new_set
```

Output

```
1) "John"
2) "George"
```

`sdiff` returns the set difference — members resulting from the difference of the first specified set from each of the following sets:

```
sdiff presidents kings beatles
```

Output

```
1) "Thomas"
```

In other words, `sdiff` looks at each member in the first given set and then compares those to members in each successive set. Any member in the first set that also appears in the following sets is removed, and `sdiff` returns the remaining members. Think of it as removing members of subsequent sets from the first set.

`sdiffstore` performs a function similar to `sdiff`, but instead of returning the set difference it creates a new set at a given destination, containing the set difference:

```
sdiffstore new_set beatles kings presidents
smembers new_set
```

Output

```
1) "Paul"
2) "Ringo"
```

Like `sinterstore`, `sdiffstore` will overwrite the destination key if it already exists.

`sunion` returns the set union, or a set containing every member of every set you specify:

```
sunion presidents kings beatles
```

Output

```
1) "Thomas"
2) "George"
3) "Paul"
4) "Henry"
5) "James"
6) "Edward"
7) "John"
8) "Ringo"
```

`sunion` treats the results like a new set in that it only allows one occurrence of any given member.

`sunionstore` performs a similar function, but creates a new set containing the set union at a given destination instead of just returning the results:

```
sunionstore new_set presidents kings beatles
```

Output

```
(integer) 8
```

As with `sinterstore` and `sdiffstore`, `sunionstore` will overwrite the destination key if it already exists.

Conclusion

This guide details a number of commands used to create and manage sets in Redis. If there are other related commands, arguments, or procedures

you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Manage Sorted Sets in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. In Redis, sorted sets are a data type similar to sets in that both are non repeating groups of strings. The difference is that each member of a sorted set is associated with a score, allowing them to be sorted from the smallest score to the largest. As with sets, every member of a sorted set must be unique, though multiple members can share the same score.

This tutorial explains how to create sorted sets, retrieve and remove their members, and create new sorted sets from existing ones.

Creating Sorted Sets and Adding Members

To create a sorted set, use the `zadd` command. `zadd` accepts as arguments the name of the key that will hold the sorted set, followed by the score of the member you're adding and the value of the member itself. The following command will create a sorted set key named `faveGuitarists` with one member, `"Joe Pass"`, that has a score of `1`:

```
zadd faveGuitarists 1 "Joe Pass"
```

`zadd` will return an integer that indicates how many members were added to the sorted set if it was created successfully.

Output

```
(integer) 1
```

You can add more than one member to a sorted set with `zadd`. Note that their scores don't need to be sequential, there can be gaps between scores, and multiple members held in the same sorted set can share the same score:

```
zadd faveGuitarists 4 "Stephen Malkmus" 2 "Rosetta
Tharpe" 3 "Bola Sete" 3 "Doug Martsch" 8
"Elizabeth Cotten" 12 "Nancy Wilson" 4 "Memphis
Minnie" 12 "Michael Houser"
```

Output

```
(integer) 8
```

`zadd` can accept the following options, which you must enter after the key name and before the first member score:

- `NX` or `XX`: These options have opposite effects, so you can only include one of them in any `zadd` operation:
    - `NX`: Tells `zadd` not to update existing members. With this option, `zadd` will only add new elements.
    - `XX`: Tells `zadd` to only update existing elements. With this option, `zadd` will never add new members.
- `CH`: Normally, `zadd` only returns the number of new elements added to the sorted set. With this option included, though, `zadd` will return the number changed elements. This includes newly added members and members whose scores were changed.

- `INCR`: This causes the command to increment the member's score value. If the member doesn't yet exist, the command will add it to the sorted set with the increment as its score, as if its original score was `0`. With `INCR` included, the `zadd` will return the member's new score if it's successful. Note that you can only include one score/member pair at a time when using this option.

Instead of passing the `INCR` option to `zadd`, you can instead use the `zincrby` command which behaves the exact same way. Instead of giving the sorted set member the value indicated by the score value like `zadd`, it increments that member's score up by that value. For example, the following command increments the score of the member `"Stephen Malkmus"`, which was originally `4`, up by `5` to `9`.

```
zincrby faveGuitarists 5 "Stephen Malkmus"
```

Output

```
"9"
```

As is the case with the `zadd` command's `INCR` option, if the specified member doesn't exist then `zincrby` will create it with the increment value as its score.

Retrieving Members from Sorted Sets

The most fundamental way to retrieve the members held within a sorted set is to use the `zrange` command. This command accepts as arguments the name of the key whose members you want to retrieve and a range of members held within it. The range is defined by two numbers that

represent zero-based indexes, meaning that `0` represents the first member in the sorted set (or, the member with the lowest score), `1` represents the next, and so on.

The following example will return the first four members from the `faveGuitarists` sorted set created in the previous section:

```
zrange faveGuitarists 0 3
```

Output

```
1) "Joe Pass"
2) "Rosetta Tharpe"
3) "Bola Sete"
4) "Doug Martsch"
```

Note that if the sorted set you pass to `zrange` has two or more elements that share the same score, it will sort those elements in lexicographical, or alphabetical, order.

The start and stop indexes can also be negative numbers, with `-1` representing the last member, `-2` representing the second to last, and so on:

```
zrange faveGuitarists -5 -2
```

Output

```
1) "Memphis Minnie"
2) "Elizabeth Cotten"
3) "Stephen Malkmus"
4) "Michael Houser"
```

`zrange` can accept the `WITHSCORES` argument which, when included, will also return the members' scores:

```
zrange faveGuitarists 5 6 WITHSCORES
```

Output

```
1) "Elizabeth Cotten"
2) "8"
3) "Stephen Malkmus"
4) "9"
```

`zrange` can only return a range of members in ascending numerical order. To reverse this and return a range in descending order, you must use the `zrevrange` command. Think of this command as temporarily reversing the order of the given sorted set before returning the members that fall within the specified range. So with `zrevrange`, `0` will represent the last member held in the key, `1` will represent the second to last, and so on:

```
zrevrange faveGuitarists 0 5
```

Output

```
1) "Nancy Wilson"
2) "Michael Houser"
3) "Stephen Malkmus"
4) "Elizabeth Cotten"
5) "Memphis Minnie"
6) "Doug Martsch"
```

`zrevrange` can also accept the `WITHSCORES` option.

You can return a range of members based on their scores with the `zrangebyscore` command. In the following example, the command will return any member held in the `faveGuitarists` key with a score of 2, 3, or 4:

```
zrangebyscore faveGuitarists 2 4
```

Output

```
1) "Rosetta Tharpe"
2) "Bola Sete"
3) "Doug Martsch"
4) "Memphis Minnie"
```

The range is inclusive in this example, meaning that it will return members with scores of 2 or 4. You can exclude either end of the range by preceding it with an open parenthesis (`(`). The following example will return every member with a score greater than or equal to `2`, but less than `4`:

```
zrangebyscore faveGuitarists 2 (4
```

Output

```
1) "Rosetta Tharpe"
2) "Bola Sete"
3) "Doug Martsch"
```

As with `zrange`, `zrangebyscore` can accept the `WITHSCORES` argument. It also accepts the `LIMIT` option, which you can use to retrieve

only a selection of elements from the `zrangebyscore` output. This option accepts an offset, which marks the first member in the range that the command will return, and a count, which defines how many members the command will return in total. For example, the following command will look at the first six members of the `faveGuitarists` sorted set but will only return 3 members from it, starting from the second member in the range, represented by `1`:

```
zrangebyscore faveGuitarists 0 5 LIMIT 1 3
```

Output

```
1) "Rosetta Tharpe"
2) "Bola Sete"
3) "Doug Martsch"
```

The `zrevrangebyscore` command returns a reversed range of members based on their scores. The following command returns every member of the set with a score between 10 and 6:

```
zrevrangebyscore faveGuitarists 10 6
```

Output

```
1) "Stephen Malkmus"
2) "Elizabeth Cotten"
```

As with `zrangebyscore`, `zrevrangebyscore` can accept both the `WITHSCORES` and `LIMIT` options. Additionally, you can exclude either end of the range by preceding it with an open parenthesis.

There may be times when all the members in a sorted set have the same score. In such a case, you can force redis to return a range of elements sorted lexicographically, or in alphabetical order, with the `zrangebylex` command. To try out this command, run the following `zadd` command to create a sorted set where each member has the same score:

```
zadd SomervilleSquares 0 Davis 0 Inman 0 Union 0
porter 0 magoun 0 ball 0 assembly
```

`zrangebylex` must be followed by the name of a key, a start interval, and a stop interval. The start and stop intervals must begin with an open parenthesis (`(`) or an open bracket (`[`), like this:

```
zrangebylex SomervilleSquares [a [z
```

Output

```
1) "assembly"
2) "ball"
3) "magoun"
4) "porter"
```

Notice that this example returned only four of the eight members in the set, even though the command sought a range from `a` to `z`. This is because Redis values are case-sensitive, so the members that begin with uppercase letters were excluded from its output. To return those, you could run the following:

```
zrangebylex SomervilleSquares [A [z
```

Output

```
1) "Davis"
2) "Inman"
3) "Union"
4) "assembly"
5) "ball"
6) "magoun"
7) "porter"
```

`zrangebylex` also accepts the special characters `-`, which represents negative infinity, and `+`, which represents positive infinity. Thus, the following command syntax will also return every member of the sorted set:

```
zrangebylex SomervilleSquares - +
```

Note that `zrangebylex` cannot return sorted set members in reverse lexicographical (ascending alphabetical) order. To do that, use `zrevrangebylex`:

```
zrevrangebylex SomervilleSquares + -
```

Output

```
1) "porter"
2) "magoun"
3) "ball"
4) "assembly"
5) "Union"
6) "Inman"
7) "Davis"
```

Because it's intended for use with sorted sets where every member has the same score, `zrangebylex` does not accept the `WITHSCORES` option. It does, however, accept the `LIMIT` option.

Retrieving Information about Sorted Sets

To find out how many members are in a given sorted set (or, in other words, to determine its cardinality), use the `zcard` command. The following example shows how many members are held in the `faveGuitarists` key from the first section of this guide:

```
zcard faveGuitarists
```

Output

```
(integer) 9
```

`zcount` can tell you how many elements are held within a given sorted set that fall within a range of scores. The first number following the key is the start of the range and the second one is the end of the range:

```
zcount faveGuitarists 3 8
```

Output

```
(integer) 4
```

`zscore` outputs the score of a specified member of a sorted set:

```
zscore faveGuitarists "Bola Sete"
```

Output

```
"3"
```

If either the specified member or key don't exist, `zscore` will return `(nil)`.

`zrank` is similar to `zscore`, but instead of returning the given member's score, it instead returns its rank. In Redis, a rank is a zero-based index of the members of a sorted set, ordered by their score. For example, `"Joe Pass"` has a score of `1`, but because that is the lowest score of any member in the key, it has a rank of `0`:

```
zrank faveGuitarists "Joe Pass"
```

Output

```
(integer) 0
```

There's another Redis command called `zrevrank` which performs the same function as `zrank`, but instead reverses the ranks of the members in the set. In the following example, the member `"Joe Pass"` has the lowest score, and consequently has the highest reversed rank:

```
zrevrank faveGuitarists "Joe Pass"
```

Output

```
(integer) 8
```

The only relation between a member's score and their rank is where their score stands in relation to those of other members. If there is a score gap between two sequential members, that won't be reflected in their rank.

Note that if two members have the same score, the one that comes first alphabetically will have the lower rank.

Like `zscore`, `zrank` and `zrevrank` will return `(nil)` if the key or member doesn't exist.

`zlexcount` can tell you how many members are held in a sorted set between a lexicographical range. The following example uses the `SomervilleSquares` sorted set from the previous section:

```
zlexcount SomervilleSquares [M [t
```

Output

```
(integer) 5
```

This command follows the same syntax as the `zrangebylex` command, so refer to the previous section for details on how to define a string range.

Removing Members from Sorted Sets

The `zrem` command can remove one or more members from a sorted set:

```
zrem faveGuitarists "Doug Martsch" "Bola Sete"
```

`zrem` will return an integer indicating how many members it removed from the sorted set:

Output

```
(integer) 2
```

There are three Redis commands that allow you to remove members of a sorted set based on a range. For example, if each member in a sorted set has the same score, you can remove members based on a lexicographical range with `zremrangebylex`. This command uses the same syntax as `zrangebylex`. The following example will remove every member that begins with a capital letter from the `SomervilleSquares` key created in the previous section:

```
zremrangebylex SomervilleSquares [A [Z
```

`zremrangebylex` will output an integer indicating how many members it removed:

Output

```
(integer) 3
```

You can also remove members based on a range of scores with the `zremrangebyscore` command, which uses the same syntax as the `zrangebyscore` command. The following example will remove every member held in `faveGuitarists` with a score of 4, 5, or 6:

```
zremrangebyscore faveGuitarists 4 6
```

Output

```
(integer) 1
```

You can remove members from a set based on a range of ranks with the `zremrangebyrank` command, which uses the same syntax as `zrangebyrank`. The following command will remove the three

members of the sorted set with the lowest rankings, which are defined by a range of zero-based indexes:

```
zremrangebyrank faveGuitarists 0 2
```

Output

```
(integer) 3
```

Note that numbers passed to `remrangebyrank` can also be negative, with `-1` representing the highest rank, `-2` the next highest, and so on.

Creating New Sorted Sets from Existing Ones

Redis includes two commands that allow you to compare members of multiple sorted sets and create new ones based on those comparisons: `zinterstore` and `zunionstore`. To experiment with these commands, run the following `zadd` commands to create some example sorted sets.

```
zadd NewKids 1 "Jonathan" 2 "Jordan" 3 "Joey" 4
"Donnie" 5 "Danny"
zadd Nsync 1 "Justin" 2 "Chris" 3 "Joey" 4 "Lance"
5 "JC"
```

`zinterstore` finds the members shared by two or more sorted sets — their intersection — and produces a new sorted set containing only those members. This command must include, in order, the name of a destination key where the intersecting members will be stored as a sorted set, the number of keys being passed to `zinterstore`, and the names of the keys you want to analyze:

```
zinterstore BoyBands 2 NewKids Nsync
```

`zinterstore` will return an integer showing the number of elements stored to the destination sorted set. Because `NewKids` and `Nsync` only share one member, `"Joey"`, the command will return `1`:

Output

```
(integer) 1
```

Be aware that if the destination key already exists, `zinterstore` will overwrite its contents.

`zunionstore` will create a new sorted set holding every member of the keys passed to it. This command uses the same syntax as `zinterstore`, and requires the name of a destination key, the number of keys being passed to the command, and the names of the keys:

```
zunionstore SuperGroup 2 NewKids Nsync
```

Like `zinterstore`, `zunionstore` will return an integer showing the number of elements stored in the destination key. Even though both of the original sorted sets held five members, because sorted sets can't have repeating members and each key has one member named `"Joey"`, the resulting integer will be `9`:

Output

```
(integer) 9
```

Like `zinterstore`, `zunionstore` will overwrite the contents of the destination key if it already exists.

To give you more control over member scores when creating new sorted sets with `zinterstore` and `zunionstore`, both of these commands accept the `WEIGHTS` and `AGGREGATE` options.

The `WEIGHTS` option is followed by one number for every sorted set included in the command which weight, or multiply, the scores of each member. The first number after the `WEIGHTS` option weights the scores of the first key passed to the command, the second number weights the second key, and so on.

The following example creates a new sorted set holding the intersecting keys from the `NewKids` and `Nsync` sorted sets. It weights the scores in the `NewKids` key by a factor of three, and weights those in the `Nsync` key by a factor of seven:

```
zinterstore BoyBandsWeighted 2 NewKids Nsync
WEIGHTS 3 7
```

If the `WEIGHTS` option isn't included, the weighting defaults to `1` for both `zinterstore` and `zunionstore`.

`AGGREGATE` accepts three sub-options. The first of these, `SUM`, implements `zinterstore` and `zunionstore`'s default behavior by adding the scores of matching members in the combined sets.

If you run a `zinterstore` or `zunionstore` operation on two sorted sets that share one member, but this member has a different score in each set, you can force the operation to assign the lower of the two scores in the new set with the `MIN` suboption.

```
zinterstore BoyBandsWeightedMin 2 NewKids Nsync
WEIGHTS 3 7 AGGREGATE MIN
```

Because the two sorted sets only have one matching member with the same score (`3`), this command will create a new set with a member that

has the lower of the two weighted scores:

```
zscore BoyBandsWeightedMin "Joey"
```

Output

```
"9"
```

Likewise, `AGGREGATE` can force `zinterstore` or `zunionstore` to assign the higher of the two scores with the `MAX` option:

```
zinterstore BoyBandsWeightedMax 2 NewKids Nsync
WEIGHTS 3 7 AGGREGATE MAX
```

This command creates a new set with on one member, `"Joey"`, that has the higher of the two weighted scores:

```
zscore BoyBandsWeightedMax "Joey"
```

Output

```
"21"
```

It can be helpful to think of `WEIGHTS` as a way to temporarily manipulate members' scores before they're analyzed. Likewise, it's helpful to think of the `AGGREGATE` option as a way to decide how to control members' scores before they're added to their new sets.

Conclusion

This guide details a number of commands used to create and manage sorted sets in Redis. If there are other related commands, arguments, or

procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Run Transactions in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. Redis allows you to plan a sequence of commands and run them one after another, a procedure known as a transaction. Each transaction is treated as an uninterrupted and isolated operation, which ensures data integrity. Clients cannot run commands while a transaction block is being executed

This tutorial goes over how to execute and cancel transactions, and also includes some information on pitfalls commonly associated with transactions.

Running Transactions

The `multi` command tells Redis to begin a transaction block. Any subsequent commands will be queued up until you run an `exec` command, which will execute them.

The following commands form a single transaction block. The first command initiates the transaction, the second sets a key holding a string with the value of `1`, the third increases the value by 1, the fourth increases its value by 40, the fifth returns the current value of the string, and the last one executes the transaction block:

```
multi
set key_MeaningOfLife 1
incr key_MeaningOfLife
incrby key_MeaningOfLife 40
```

```
get key_MeaningOfLife
exec
```

After running `multi`, `redis-cli` will respond to each of the following commands with `QUEUED`. After you run the `exec` command, it will show the output of each of those commands individually:

Output

```
1) OK
2) (integer) 2
3) (integer) 42
4) "42"
```

Commands included in a transaction block are run sequentially in the order they're queued. Redis transactions are atomic, meaning that either every command in a transaction block is processed (meaning that it's accepted as valid and queued to be executed) or none are. However, even if a command is successfully queued, it may still produce an error when executed. In such cases, the other commands in the transaction can still run, but Redis will skip the error-causing command. See the section on understanding transaction errors for more details.

Canceling Transactions

To cancel a transaction, run the `discard` command. This prevents any previously-queued commands from running:

```
multi
set key_A 146
```

```
incrby key_A 10
discard
```

Output

```
OK
```

The `discard` command returns the connection to a normal state, which tells Redis to run single commands as usual. You'll need to run `multi` again to tell the server you're starting another transaction.

Understanding Transaction Errors

Some commands may be impossible to queue, such as commands with syntax errors. If you attempt to queue a syntactically incorrect command Redis will return an error.

The following transaction createst a key named `key_A` and then attempts to increment it by 10. However, a spelling error in the `incrby` command causes and error and closes the transaction:

```
multi
set key_A 146
incrbuy key_A 10
```

Output

```
(error) ERR unknown command 'incrbuy'
```

If you try to run an `exec` command after trying to queue a command with a syntax error like this one, you will receive another error message

telling you that the transaction was discarded:

```
exec
```

Output

```
(error) EXECABORT Transaction discarded because of
previous errors.
```

In cases like this, you'll need to restart the transaction block and make sure you enter each command correctly.

Some impossible commands are possible to queue, such as running `incr` on a key containing only a string. Because such command is syntactically correct, Redis won't return an error if you try to include it in a transaction and won't prevent you from running `exec`. In cases like this, all other commands in the queue will be executed, but the impossible command will return an error:

```
multi
set key_A 146
incrby key_A "ten"
exec
```

Output

```
1) OK
2) (error) ERR value is not an integer or out of
range
```

For more information on how Redis handles errors inside transactions, see the official documentation on the subject.

Conclusion

This guide details a number of commands used to create, run, and cancel transactions in Redis. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Expire Keys in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. Redis keys are persistent by default, meaning that the Redis server will continue to store them unless they are deleted manually. There may, however, be cases where you've set a key but you know you will want to delete it after a certain amount of time has passed; in other words, you want the key to be volatile. This tutorial explains how to set keys to expire, check the time remaining until a key's expiration, and cancel a key's expiration setting.

Setting Keys to Expire

You can set an expiration time for an existing key with the `expire` command, which takes the name of the key and the number of seconds until expiration as arguments. To demonstrate this, run the following two commands. The first creates a string key named `key_melon` with a value of `"cantaloupe"`, and the second sets it to expire after 450 seconds:

```
set key_melon "cantaloupe"
expire key_melon 450
```

If the timeout was set successfully, the `expire` command will return `(integer) 1`. If setting the timeout failed, it will instead return `(integer) 0`.

Alternatively, you can set the key to expire at a specific point in time with the `expireat` command. Instead of the number of seconds before expiration, it takes a Unix timestamp as an argument. A Unix timestamp is

the number of seconds since the Unix epoch, or 00:00:00 UTC on January 1, 1970. There are a number of tools online you can use to find the Unix timestamp of a specific date and time, such as EpochConverter or UnixTimestamp.com.

For example, to set `key_melon` to expire at 8:30pm GMT on May 1, 2025 (represented by the Unix timestamp `1746131400`), you could use the following command:

```
expireat key_melon 1746131400
```

Note that if the timestamp you pass to `expireat` has already occurred, it will delete the key immediately.

Checking How Long Until a Key Expires

Any time you set a key to expire, you can check the time remaining until expiry (in seconds) with `ttl`, which stands for "time to live":

```
ttl key_melon
```

Output

```
(integer) 433
```

For more granular information, you can run `pttl` which will instead return the amount of time until a key expires in milliseconds:

```
pttl key_melon
```

Output

```
(integer) 431506
```

Both `ttl` and `pttl` will return `(integer) -1` if the key hasn't been set to expire and `(integer) -2` if the key does not exist.

Persisting Keys

If a key has been set to expire, any command that overwrites the contents of a key — like `set` or `getset` — will clear a key's timeout value. To manually clear a key's timeout, use the `persist` command:

```
persist key_melon
```

The `persist` command will return `(integer) 1` if it completed successfully, indicating that the key will no longer expire.

Conclusion

This guide details a number of commands used to manipulate and check key persistence in Redis. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Troubleshoot Issues in Redis

Written by Mark Drake

Redis is an open-source, in-memory key-value data store. It comes with several commands that can help with troubleshooting and debugging issues. Because of Redis's nature as an in-memory key-value store, many of these commands focus on memory management, but there are others that are valuable for providing an overview of the state of your Redis server. This tutorial will provide details on how to use some of these commands to help diagnose and resolve issues you may run into as you use Redis.

Troubleshooting Memory-related Issues

`memory usage` tells you how much memory is currently being used by a single key. It takes the name of a key as an argument and outputs the number of bytes it uses:

```
memory usage key_meaningOfLife
```

Output

```
(integer) 42
```

For a more general understanding of how your Redis server is using memory, you can run the `memory stats` command:

```
memory stats
```

This command outputs an array of memory-related metrics and their values. The following are the metrics reported by `memory stats`:

- `peak.allocated`: The peak number of bytes consumed by Redis
- `total.allocated`: The total number of bytes allocated by Redis
- `startup.allocated`: The initial number of bytes consumed by Redis at startup
- `replication.backlog`: The size of the replication backlog, in bytes
- `clients.slaves`: The total size of all replica overheads (the output and query buffers and connection contexts)
- `clients.normal`: The total size of all client overheads
- `aof.buffer`: The total size of the current and rewrite append-only file buffers
- `db.0`: The overheads of the main and expiry dictionaries for each database in use on the server, reported in bytes
- `overhead.total`: The sum of all overheads used to manage Redis's keyspace
- `keys.count`: The total number of keys stored in all the databases on the server
- `keys.bytes-per-key`: The ratio of the server's net memory usage and `keys.count`
- `dataset.bytes`: The size of the dataset, in bytes
- `dataset.percentage`: The percentage of Redis's net memory usage taken by `dataset.bytes`
- `peak.percentage`: The percentage of `peak.allocated` taken out of `total.allocated`

- `fragmentation`: The ratio of the amount of memory currently in use divided by the physical memory Redis is actually using

`memory malloc-stats` provides an internal statistics report from jemalloc, the memory allocator used by Redis on Linux systems:

```
memory malloc-stats
```

If it seems like you're running into memory-related issues, but parsing the output of the previous commands proves to be unhelpful, you can try running `memory doctor`:

```
memory doctor
```

This feature will output any memory consumption issues that it can find and suggest potential solutions.

Getting General Information about Your Redis Instance

A debugging command that isn't directly related to memory management is `monitor`. This command allows you to see a constant stream of every command processed by the Redis server:

```
monitor
```

Output

```
OK
1566157213.896437 [0 127.0.0.1:47740] "auth"
"foobared"
1566157215.870306 [0 127.0.0.1:47740] "set"
"key_1" "878"
```

Another command useful for debugging is `info`, which returns several blocks of information and statistics about the server:

```
info
```

Output

```
# Server
redis_version:4.0.9
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:9435c3c2879311f3
redis_mode:standalone
os:Linux 4.15.0-52-generic x86_64
. . .
```

This command returns a lot of information. If you only want to see one info block, you can specify it as an argument to `info`:

```
info CPU
```

Output

```
# CPU
used_cpu_sys:173.16
used_cpu_user:70.89
used_cpu_sys_children:0.01
used_cpu_user_children:0.04
```

Note that the information returned by the `info` command will depend on which version of Redis you're using.

Using the `keys` Command

The `keys` command is helpful in cases where you've forgotten the name of a key, or perhaps you've created one but accidentally misspelled its name. `keys` looks for keys that match a pattern:

`keys` **`pattern`**

The following glob-style variables are supported

- `?` is a wildcard standing for any single character, so `s?mmy` matches `sammy`, `sommy`, and `sqmmy`
- `*` is a wildcard that stands for any number of characters, including no characters at all, so `sa*y` matches `sammy`, `say`, `sammmmmmy`, and `salmony`
- You can specify two or more characters that the pattern can include by wrapping them in brackets, so `s[ai]mmy` will match `sammy` and `simmy`, but not `summy`
- To set a wildcard that disregards one or more letters, wrap them in brackets and precede them with a carrot (`^`), so `s[^oi]mmy` will match `sammy` and `sxmmy`, but not `sommy` or `simmy`
- To set a wildcard that includes a range of letters, separate the beginning and end of the range with a hyphen and wrap it in brackets, so `s[a-o]mmy` will match `sammy`, `skmmy`, and `sommy`, but not `srmmy`

Warning: The Redis documentation warns that `keys` should almost never be used in a production environment, since it can have a major negative impact on performance.

Conclusion

This guide details a number of commands that are helpful for troubleshooting and resolving issues one might encounter as they work with Redis. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.

How To Change Redis's Configuration from the Command Line

Written by Mark Drake

Introduction

Redis is an open-source, in-memory key-value data store. Redis has several commands that allow you to make changes to the Redis server's configuration settings on the fly. This tutorial will go over some of these commands, and also explain how to make these configuration changes permanent.

Changing Redis's Configuration

The commands outlined in this section will only alter the Redis server's behavior for the duration of the current session, or until you run `config rewrite` which will make them permanent. You can alter the Redis configuration file directly by opening and editing it with your preferred text editor. For example, you can use `nano` to do so:

```
sudo nano /etc/redis/redis.conf
```

Warning: The `config set` command is considered dangerous. By changing your Redis configuration file, it's possible that you will cause your Redis server to behave in unexpected or undesirable ways. We recommend that you only run the `config set` command if you are testing out its behavior or you're absolutely certain that you want to make changes to your Redis configuration.

It may be in your interest to rename this command to something with a lower likelihood of being run accidentally.

`config set` allows you to reconfigure Redis at runtime without having to restart the service. It uses the following syntax:

```
config set parameter value
```

For example, if you wanted to change the name of the database dump file Redis will produce after you run a `save` command, you might run a command like the following:

```
config set "dbfilename" "new_file.rdb"
```

If the configuration change is valid, the command will return `OK`. Otherwise it will return an error.

Note: Not every parameter in the `redis.conf` file can be changed with a `config set` operation. For example, you cannot change the authentication password defined by the `requirepass` parameter.

Making Configuration Changes Permanent

`config set` does not permanently alter the Redis instance's configuration file; it only changes Redis's behavior at runtime. To edit `redis.conf` after running a `config-set` command and make the current session's configuration permanent, run `config rewrite`:

```
config rewrite
```

This command does its best to preserve the comments and overall structure of the original `redis.conf` file, with only minimal changes to match the settings currently used by the server.

Like `config set`, if the rewrite is successful `config rewrite` will return `OK`.

Checking Redis's Configuration

To read the current configuration parameters of a Redis server, run the `config get` command. `config get` takes a single argument, which can be either an exact match of a parameter used in `redis.conf` or a glob pattern. For example:

```
config get repl*
```

Depending on your Redis configuration, this command might return:

Output

```
 1) "repl-ping-slave-period"
 2) "10"
 3) "repl-timeout"
 4) "60"
 5) "repl-backlog-size"
 6) "1048576"
 7) "repl-backlog-ttl"
 8) "3600"
 9) "repl-diskless-sync-delay"
10) "5"
11) "repl-disable-tcp-nodelay"
12) "no"
13) "repl-diskless-sync"
14) "no"
```

You can also return all of the configuration parameters supported by `config set` by running `config get *`.

Conclusion

This guide details the `redis-cli` commands used to make changes to a Redis server's configuration file on the fly. If there are other related commands, arguments, or procedures you'd like to see outlined in this guide, please ask or make suggestions in the comments below.

For more information on Redis commands, see our tutorial series on How to Manage a Redis Database.