**Redis**

**Redis - Remote Dictionary Server**

<http://try.redis.io/#run>

Port Number: 6379

**Redis** is what is called a key-value store, often referred to as a NoSQL database. The essence of a key-value store is the ability to store some data, called a value, inside a key. This data can later be retrieved only if we know the exact key used to store it. We can use the command **SET** to store the value "fido" at key "server:name":

SET server:name "fido"

Redis will store our data permanently, so we can later ask "What is the value stored at key server:name?" and Redis will reply with "fido":

GET server:name => "fido"

=======================================================

Other common operations provided by key-value stores are **DEL** to delete a given key and associated value, SET-if-not-exists (called **SETNX** on Redis) that sets a key only if it does not already exist, and **INCR** to atomically increment a number stored at a given key , same as **DECR** to atomically decrement a number stored at a given key.

SET connections 10

INCR connections => 11

INCR connections => 12

DEL connections

INCR connections => 1

SETNX key value

TIME COMPLEXITY: O(1)

DESCRIPTION: SETNX works exactly like SET with the only difference that if the key already exists no operation is performed. SETNX actually means "SET if Not eXists".

RETURN VALUE: Integer reply, specifically:

**1 if the key was set**

**0 if the key was not set**

Example:

> setnx redhat centos

(integer) 1==========> the key was not set previously, so it assign new key

> get redhat

"centos"

=======================================================

**INCR**

There is something special about INCR. Why do we provide such an operation if we can do it ourself with a bit of code? After all it is as simple as:

x = GET count

x = x + 1

SET count x

The problem is that doing the increment in this way will only work as long as there is a single client using the key. See what happens if two clients are accessing this key at the same time:

Client A reads count as 10.

Client B reads count as 10.

Client A increments 10 and sets count to 11.

Client B increments 10 and sets count to 11.

We wanted the value to be 12, but instead it is 11! This is because incrementing the value in this way is not an atomic operation. Calling the INCR command in Redis will prevent this from happening, because it is an atomic operation. Redis provides many of these atomic operations on different types of data.

============================================================

Redis can be told that a key should only exist for a certain length of time. This is accomplished with the **EXPIRE** and **TTL** commands.

SET resource:lock "Redis Demo"

EXPIRE resource:lock 120

This causes the key resource:lock to be deleted in 120 seconds. You can test how long a key will exist with the TTL command. It returns the number of seconds until it will be deleted.

TTL resource:lock => 113

// after 113s

TTL resource:lock => -2

**The -2 for the TTL of the key means that the key does not exist (anymore). A -1 for the TTL of the key means that it will never expire. Note that if you SET a key, its TTL will be reset.**

SET resource:lock "Redis Demo 1"

EXPIRE resource:lock 120

TTL resource:lock => 119

SET resource:lock "Redis Demo 2"

TTL resource:lock => -1

**Example:**

> **[set red hat](http://try.redis.io/#run)**

**OK**

> **[get red](http://try.redis.io/#run)**

**"hat"**

> **[ttl red](http://try.redis.io/#run)**

**(integer) -1 =======> Means it will never expire(Because we are not mention the expiry details.)**

> **[expire red 10](http://try.redis.io/#run)**

**(integer) 1**

> **[ttl red](http://try.redis.io/#run)**

**(integer) 3 =========> 3 seconds left for expiry**

> **[ttl red](http://try.redis.io/#run)**

**(integer) -2 =========> Means the key does not exist**

> **[get red](http://try.redis.io/#run)**

**(nil) ===========> The key has expired.**

**========================================================**

Redis also supports several more complex data structures. The first one we'll look at is a list. A list is a series of ordered values. Some of the important commands for interacting with lists are **RPUSH, LPUSH, LLEN, LRANGE, LPOP, and RPOP**. You can immediately begin working with a key as a list, as long as it doesn't already exist as a different type.

**RPUSH puts the new value at the end of the list.**

RPUSH friends "Alice"

RPUSH friends "Bob"

**LPUSH puts the new value at the start of the list.**

LPUSH friends "Sam"

**LRANGE** gives a subset of the list. It takes the index of the first element you want to retrieve as its first parameter and the index of the last element you want to retrieve as its second parameter. A value of -1 for the second parameter means to retrieve elements until the end of the list.

LRANGE friends 0 -1 => 1) "Sam", 2) "Alice", 3) "Bob"

LRANGE friends 0 1 => 1) "Sam", 2) "Alice"

LRANGE friends 1 2 => 1) "Alice", 2) "Bob"

**LLEN returns the current length of the list.**

LLEN friends => 3

**LPOP removes the first element from the list and returns it.**

LPOP friends => "Sam"

**RPOP removes the last element from the list and returns it.**

RPOP friends => "Bob"

Note that the list now only has one element:

LLEN friends => 1

LRANGE friends 0 -1 => 1) "Alice"

============================================================

The next data structure that we'll look at is a **set**. A set is similar to a list, except it does not have a specific order and each element may only appear once. Some of the important commands in working with sets are **SADD, SREM, SISMEMBER, SMEMBERS and SUNION.**

**SADD adds the given value to the set.**

SADD superpowers "flight"

SADD superpowers "x-ray vision"

SADD superpowers "reflexes"

**SREM removes the given value from the set.**

SREM superpowers "reflexes"

**SISMEMBER tests if the given value is in the set. It returns 1 if the value is there and 0 if it is not.**

SISMEMBER superpowers "flight" => 1

SISMEMBER superpowers "reflexes" => 0

**SMEMBERS returns a list of all the members of this set.**

SMEMBERS superpowers => 1) "flight", 2) "x-ray vision"

**SUNION combines two or more sets and returns the list of all elements.**

SADD birdpowers "pecking"

SADD birdpowers "flight"

SUNION superpowers birdpowers => 1) "pecking", 2) "x-ray vision", 3) "flight"

============================================================

Sets are a very handy data type, but as they are unsorted they don't work well for a number of problems. This is why Redis 1.2 introduced Sorted Sets.

**A sorted set is similar to a regular set, but now each value has an associated score. This score is used to sort the elements in the set.**

ZADD hackers 1940 "Alan Kay"

ZADD hackers 1906 "Grace Hopper"

ZADD hackers 1953 "Richard Stallman"

ZADD hackers 1965 "Yukihiro Matsumoto"

ZADD hackers 1916 "Claude Shannon"

ZADD hackers 1969 "Linus Torvalds"

ZADD hackers 1957 "Sophie Wilson"

ZADD hackers 1912 "Alan Turing"

In these examples, the scores are years of birth and the values are the names of famous hackers.

**ZREM is used to remove.**

> **[zrem hackers "Linus Torvalds"](http://try.redis.io/#run)**

**1**

**ZRANGE is used to sort the elements.**

> **[zrange hackers 0 -1](http://try.redis.io/#run)**

**1) "Grace Hopper"**

**2) "Alan Turing"**

**3) "Claude Shannon"**

**4) "Alan Kay"**

**5) "Richard Stallman"**

**6) "Sophie Wilson"**

**7) "Yukihiro Matsumoto"**

> **[zrange hackers 2 4](http://try.redis.io/#run)**

**1) "Claude Shannon"**

**2) "Alan Kay"**

**3) "Richard Stallman"**