redis-cli, the Redis command line interface

`redis-cli` is the Redis command line interface, a simple program that allows to send commands to Redis, and read the replies sent by the server, directly from the terminal.

It has two main modes: an interactive mode where there is a REPL (Read Eval Print Loop) where the user types commands and get replies; and another mode where the command is sent as arguments of `redis-cli`, executed, and printed on the standard output.

In interactive mode, `redis-cli` has basic line editing capabilities to provide a good typing experience.

However `redis-cli` is not just that. There are options you can use to launch the program in order to put it into special modes, so that `redis-cli` can definitely do more complex tasks, like simulate a slave and print the replication stream it receives from the master, check the latency of a Redis server and show statistics or even an ASCII-art spectrogram of latency samples and frequencies, and many other things.

This guide will cover the different aspects of `redis-cli`, starting from the simplest and ending with the more advanced ones.

If you are going to use Redis extensively, or if you already do, chances are you happen to use `redis-cli` a lot. Spending some time to familiarize with it is likely a very good idea, you'll see that you'll work more effectively with Redis once you know all the tricks of its command line interface.

Command line usage

To just run a command and have its reply printed on the standard output is as simple as typing the command to execute as separated arguments of `redis-cli`:

```
$ redis-cli incr mycounter
(integer) 7
```

The reply of the command is "7". Since Redis replies are typed (they can be strings, arrays, integers, NULL, errors and so forth), you see the type of the reply between brackets. However that would be not exactly a great idea when the output of `redis-cli` must be used as input of another command, or when we want to redirect it into a file.

Actually `redis-cli` only shows additional information which improves readability for humans when it detects the standard output is a tty (a terminal basically). Otherwise it will auto-enable the *raw output mode*, like in the following example:

```
$ redis-cli incr mycounter > /tmp/output.txt
$ cat /tmp/output.txt
8
```

This time `(integer)` was omitted from the output since the CLI detected the output was no longer written to the terminal. You can force raw output even on the terminal with the `--raw` option:

```
$ redis-cli --raw incr mycounter
9
```

Similarly, you can force human readable output when writing to a file or in pipe to other commands by using `--no-raw`.

Host, port, password and database

By default `redis-cli` connects to the server at 127.0.0.1 port 6379. As you can guess, you can easily change this using command line options. To specify a different host name or an IP address, use `-h`. In order to set a different port, use `-p`.

```
$ redis-cli -h redis15.localnet.org -p 6390 ping
PONG
```

If your instance is password protected, the `-a <password>` option will preform authentication saving the need of explicitly using the AUTH command:

```
$ redis-cli -a myUnguessablePazzzzzword123 ping
PONG
```

Alternatively, it is possible to provide the password to `redis-cli` via the `REDISCLI_AUTH` environment variable.

Finally, it's possible to send a command that operates on a database number other than the default number zero by using the `-n <dbnum>` option:

```
$ redis-cli flushall
OK
$ redis-cli -n 1 incr a
(integer) 1
$ redis-cli -n 1 incr a
(integer) 2
$ redis-cli -n 2 incr a
(integer) 1
```

Some or all of this information can also be provided by using the `-u <uri>` option and a valid URI:

```
$ redis-cli -u redis://p%40ssw0rd@redis-16379.hosted.com:16379/0 ping
PONG
```

SSL/TLS

By default, `redis-cli` uses a plain TCP connection to connect to Redis. You may enable SSL/TLS using the `--tls` option, along with `--cacert` or `--cacertdir` to configure a trusted root certificate bundle or directory.

If the target server requires authentication using a client side certificate, you can specify a certificate and a corresponding private key using `--cert` and `--key`.

Getting input from other programs

There are two ways you can use `redis-cli` in order to get the input from other commands (from the standard input, basically). One is to use as last argument the payload we read from *stdin*. For example, in order to set a Redis key to the content of the file `/etc/services` if my computer, I can use the `-x` option:

```
$ redis-cli -x set foo < /etc/services
OK
$ redis-cli getrange foo 0 50
"#\n# Network services, Internet style\n#\n# Note that "
```

As you can see in the first line of the above session, the last argument of the SET command was not specified. The arguments are just `SET foo` without the actual value I want my key to be set to.

Instead, the -x option was specified and a file was redirected to the CLI's standard input. So the input was read, and was used as the final argument for the command. This is useful for scripting.

A different approach is to feed `redis-cli` a sequence of commands written in a text file:

```
$ cat /tmp/commands.txt
set foo 100
incr foo
append foo xxx
get foo
$ cat /tmp/commands.txt | redis-cli
OK
(integer) 101
(integer) 6
"101xxx"
```

All the commands in `commands.txt` are executed one after the other by `redis-cli` as if they were typed by the user interactive. Strings can be quoted inside the file if needed, so that it's possible to have single arguments with spaces or newlines or other special chars inside:

```
$ cat /tmp/commands.txt
set foo "This is a single argument"
strlen foo
$ cat /tmp/commands.txt | redis-cli
OK
(integer) 25
```

Continuously run the same command

It is possible to execute the same command a specified number of times with a user selected pause between the executions. This is useful in different contexts, for example when we want to continuously monitor some key content or INFO field output, or when we want to simulate some recurring write event (like pushing a new item into a list every 5 seconds).

This feature is controlled by two options: `-r <count>` and `-i <delay>`. The first states how many times to run a command, the second configures the delay between the different command calls, in seconds (with the ability to specify decimal numbers like 0.1 in order to mean 100 milliseconds).

By default the interval (or delay) is set to 0, so commands are just executed ASAP:

```
$ redis-cli -r 5 incr foo
(integer) 1
(integer) 2
(integer) 3
(integer) 4
(integer) 5
```

To run the same command forever, use -1 as count. So, in order to monitor over time the RSS memory size it's possible to use a command like the following:

```
$ redis-cli -r -1 -i 1 INFO | grep rss_human
used_memory_rss_human:1.38M
used_memory_rss_human:1.38M
used_memory_rss_human:1.38M
... a new line will be printed each second ...
```

Mass insertion of data using `redis-cli`

Mass insert using `redis-cli` is covered in a separated page since it's a worthwhile topic itself. Please refer to our mass insertion guide.

CSV output

Sometimes you may want to use `redis-cli` in order to quickly export data from Redis to an external program. This can be accomplished using the CSV (Comma Separated Values) output feature:

```
$ redis-cli lpush mylist a b c d
(integer) 4
$ redis-cli --csv lrange mylist 0 -1
"d","c","b","a"
```

Currently it's not possible to export the whole DB like that, but only to run single commands with CSV output.

Running Lua scripts

The `redis-cli` has extensive support for using the new Lua debugging facility of Lua scripting, available starting with Redis 3.2. For this feature, please refer to the Redis Lua debugger documentation.

However, even without using the debugger, you can use `redis-cli` to run scripts from a file in a way more comfortable way compared to typing the script interactively into the shell or as an argument:

```
$ cat /tmp/script.lua
return redis.call('set',KEYS[1],ARGV[1])
$ redis-cli --eval /tmp/script.lua foo , bar
OK
```

The Redis EVAL command takes the list of keys the script uses, and the other non key arguments, as different arrays. When calling EVAL you provide the number of keys as a number. However with `redis-cli` and using the `--eval` option above, there is no need to specify the number of keys explicitly. Instead it uses the convention of separating keys and arguments with a comma. This is why in the above call you see `foo , bar` as arguments.

So `foo` will populate the KEYS array, and `bar` the ARGV array.

The `--eval` option is useful when writing simple scripts. For more complex work, using the Lua debugger is definitely more comfortable. It's possible to mix the two approaches, since the debugger also uses executing scripts from an external file.

Interactive mode

So far we explored how to use the Redis CLI as a command line program. This is very useful for scripts and certain types of testing, however most people will spend the majority of time in `redis-cli` using its interactive mode.

In interactive mode the user types Redis commands at the prompt. The command is sent to the server, processed, and the reply is parsed back and rendered into a simpler form to read.

Nothing special is needed for running the CLI in interactive mode - just launch it without any arguments and you are in:

```
$ redis-cli
127.0.0.1:6379> ping
PONG
```

The string `127.0.0.1:6379>` is the prompt. It reminds you that you are connected to a given Redis instance.

The prompt changes as the server you are connected to changes, or when you are operating on a database different than the database number zero:

```
127.0.0.1:6379> select 2
OK
127.0.0.1:6379[2]> dbsize
(integer) 1
127.0.0.1:6379[2]> select 0
OK
127.0.0.1:6379> dbsize
(integer) 503
```

Handling connections and reconnections

Using the `connect` command in interactive mode makes it possible to connect to a different instance, by specifying the *hostname* and *port* we want to connect to:

```
127.0.0.1:6379> connect metal 6379
metal:6379> ping
PONG
```

As you can see the prompt changes accordingly. If the user attempts to connect to an instance that is unreachable, the `redis-cli` goes into disconnected mode and attempts to reconnect with each new command:

```
127.0.0.1:6379> connect 127.0.0.1 9999
Could not connect to Redis at 127.0.0.1:9999: Connection refused
not connected> ping
Could not connect to Redis at 127.0.0.1:9999: Connection refused
not connected> ping
Could not connect to Redis at 127.0.0.1:9999: Connection refused
```

Generally after a disconnection is detected, the CLI always attempts to reconnect transparently: if the attempt fails, it shows the error and enters the disconnected state. The following is an example of disconnection and reconnection:

```
127.0.0.1:6379> debug restart
Could not connect to Redis at 127.0.0.1:6379: Connection refused
not connected> ping
PONG
127.0.0.1:6379> (now we are connected again)
```

When a reconnection is performed, `redis-cli` automatically re-select the last database number selected. However, all the other state about the connection is lost, such as the state of a transaction if we were in the middle of it:

```
$ redis-cli
127.0.0.1:6379> multi
OK
127.0.0.1:6379> ping
QUEUED

( here the server is manually restarted )

127.0.0.1:6379> exec
(error) ERR EXEC without MULTI
```

This is usually not an issue when using the CLI in interactive mode for testing, but you should be aware of this limitation.

Editing, history, completion and hints

Because `redis-cli` uses the linenoise line editing library, it always has line editing capabilities, without depending on `libreadline` or other optional libraries.

You can access a history of commands executed, in order to avoid retyping them again and again, by pressing the arrow keys (up and down). The history is preserved between restarts of the CLI, in a file called `.rediscli_history` inside the user home directory, as specified by the `HOME` environment variable. It is possible to use a different history filename by setting the `REDISCLI_HISTFILE` environment variable, and disable it by setting it to `/dev/null`.

The CLI is also able to perform command names completion by pressing the TAB key, like in the following example:

```
127.0.0.1:6379> Z<TAB>
127.0.0.1:6379> ZADD<TAB>
127.0.0.1:6379> ZCARD<TAB>
```

Once you've typed a Redis command name at the prompt, the CLI will display syntax hints. This behavior can be turned on and off via the CLI preferences.

Preferences

There are two ways to customize the CLI's behavior. The file `.redisclirc` in your home directory is loaded by the CLI on startup. You can override the file's default location by setting the `REDISCLI_RCFILE` environment variable to an alternative path. Preferences can also be set during a CLI session, in which case they will last only the the duration of the session.

To set preferences, use the special `:set` command. The following preferences can be set, either by typing the command in the CLI or adding it to the `.redisclirc` file:

- `:set hints` - enables syntax hints
- `:set nohints` - disables syntax hints

Running the same command N times

It's possible to run the same command multiple times by prefixing the command name by a number:

```
127.0.0.1:6379> 5 incr mycounter
(integer) 1
(integer) 2
(integer) 3
(integer) 4
(integer) 5
```

Showing help about Redis commands

Redis has a number of commands and sometimes, as you test things, you may not remember the exact order of arguments. `redis-cli` provides online help for most Redis commands, using the `help` command. The command can be used in two forms:

- `help @<category>` shows all the commands about a given category. The categories are: `@generic`, `@list`, `@set`, `@sorted_set`, `@hash`, `@pubsub`, `@transactions`, `@connection`, `@server`, `@scripting`, `@hyperloglog`.

- `help <commandname>` shows specific help for the command given as argument.

For example in order to show help for the PFADD command, use:

127.0.0.1:6379> help PFADD

PFADD key element [element ...] summary: Adds the specified elements to the specified HyperLogLog. since: 2.8.9

Note that `help` supports TAB completion as well.

Clearing the terminal screen

Using the `clear` command in interactive mode clears the terminal's screen.

Special modes of operation

So far we saw two main modes of `redis-cli`.

- Command line execution of Redis commands.
- Interactive "REPL-like" usage.

However the CLI performs other auxiliary tasks related to Redis that are explained in the next sections:

- Monitoring tool to show continuous stats about a Redis server.
- Scanning a Redis database for very large keys.
- Key space scanner with pattern matching.
- Acting as a Pub/Sub client to subscribe to channels.
- Monitoring the commands executed into a Redis instance.
- Checking the latency of a Redis server in different ways.
- Checking the scheduler latency of the local computer.
- Transferring RDB backups from a remote Redis server locally.
- Acting as a Redis slave for showing what a slave receives.
- Simulating LRU workloads for showing stats about keys hits.
- A client for the Lua debugger.

Continuous stats mode

This is probably one of the lesser known features of `redis-cli`, and one very useful in order to monitor Redis instances in real time. To enable this mode, the `--stat` option is used. The output is very clear about the behavior of the CLI in this mode:

```
$ redis-cli --stat
------- data ------ -------------------- load -------------------- -
keys       mem      clients blocked requests            connections
506        1015.00K 1       0       24 (+0)             7
506        1015.00K 1       0       25 (+1)             7
506        3.40M    51      0       60461 (+60436)      57
```

```
506         3.40M     51       0       60461 (+60436)      57
506         3.40M     51       0       146425 (+85964)     107
507         3.40M     51       0       233844 (+87419)     157
507         3.40M     51       0       321715 (+87871)     207
508         3.40M     51       0       408642 (+86927)     257
508         3.40M     51       0       497038 (+88396)     257
```

In this mode a new line is printed every second with useful information and the difference between the old data point. You can easily understand what's happening with memory usage, clients connected, and so forth.

The `-i <interval>` option in this case works as a modifier in order to change the frequency at which new lines are emitted. The default is one second.

Scanning for big keys

In this special mode, `redis-cli` works as a key space analyzer. It scans the dataset for big keys, but also provides information about the data types that the data set consists of. This mode is enabled with the `--bigkeys` option, and produces quite a verbose output:

```
$ redis-cli --bigkeys

# Scanning the entire keyspace to find biggest keys as well as
# average sizes per key type.  You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[00.00%] Biggest string found so far 'key-419' with 3 bytes
[05.14%] Biggest list   found so far 'mylist' with 100004 items
[35.77%] Biggest string found so far 'counter:__rand_int__' with 6 by
[73.91%] Biggest hash   found so far 'myobject' with 3 fields

-------- summary -------

Sampled 506 keys in the keyspace!
Total key length in bytes is 3452 (avg len 6.82)
```

```
Biggest string found 'counter:__rand_int__' has 6 bytes
Biggest   list found 'mylist' has 100004 items
Biggest   hash found 'myobject' has 3 fields

504 strings with 1403 bytes (99.60% of keys, avg size 2.78)
1 lists with 100004 items (00.20% of keys, avg size 100004.00)
0 sets with 0 members (00.00% of keys, avg size 0.00)
1 hashs with 3 fields (00.20% of keys, avg size 3.00)
0 zsets with 0 members (00.00% of keys, avg size 0.00)
```

In the first part of the output, each new key larger than the previous larger key (of the same type) encountered is reported. The summary section provides general stats about the data inside the Redis instance.

The program uses the SCAN command, so it can be executed against a busy server without impacting the operations, however the `-i` option can be used in order to throttle the scanning process of the specified fraction of second for each 100 keys requested. For example, `-i 0.1` will slow down the program execution a lot, but will also reduce the load on the server to a tiny amount.

Note that the summary also reports in a cleaner form the biggest keys found for each time. The initial output is just to provide some interesting info ASAP if running against a very large data set.

Getting a list of keys

It is also possible to scan the key space, again in a way that does not block the Redis server (which does happen when you use a command like KEYS *), and print all the key names, or filter them for specific patterns. This mode, like the `--bigkeys` option, uses the SCAN command, so keys may be reported multiple times if the dataset is changing, but no key would ever be missing, if that key was present since the start of the iteration. Because of the command that it uses this option is called `--scan`.

```
$ redis-cli --scan | head -10
key-419
key-71
key-236
key-50
key-38
key-458
key-453
key-499
key-446
key-371
```

Note that `head -10` is used in order to print only the first lines of the output.

Scanning is able to use the underlying pattern matching capability of the SCAN command with the `--pattern` option.

```
$ redis-cli --scan --pattern '*-11*'
key-114
key-117
key-118
key-113
key-115
key-112
key-119
key-11
key-111
key-110
key-116
```

Piping the output through the `wc` command can be used to count specific kind of objects, by key name:

```
$ redis-cli --scan --pattern 'user:*' | wc -l
3829433
```

Pub/sub mode

The CLI is able to publish messages in Redis Pub/Sub channels just using the PUBLISH command. This is expected since the PUBLISH command is very similar to any other command. Subscribing to channels in order to receive messages is different - in this case we need to block and wait for messages, so this is implemented as a special mode in `redis-cli`. Unlike other special modes this mode is not enabled by using a special option, but simply by using the SUBSCRIBE or PSUBSCRIBE command, both in interactive or non interactive mode:

```
$ redis-cli psubscribe '*'
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "*"
3) (integer) 1
```

The *reading messages* message shows that we entered Pub/Sub mode. When another client publishes some message in some channel, like you can do using `redis-cli PUBLISH mychannel mymessage`, the CLI in Pub/Sub mode will show something such as:

```
1) "pmessage"
2) "*"
3) "mychannel"
4) "mymessage"
```

This is very useful for debugging Pub/Sub issues. To exit the Pub/Sub mode just process `CTRL-C`.

Monitoring commands executed in Redis

Similarly to the Pub/Sub mode, the monitoring mode is entered automatically once you use the MONITOR mode. It will print all the commands received by a Redis instance:

```
$ redis-cli monitor
OK
1460100081.165665 [0 127.0.0.1:51706] "set" "foo" "bar"
1460100083.053365 [0 127.0.0.1:51707] "get" "foo"
```

Note that it is possible to use to pipe the output, so you can monitor for specific patterns using tools such as `grep`.

Monitoring the latency of Redis instances

Redis is often used in contexts where latency is very critical. Latency involves multiple moving parts within the application, from the client library to the network stack, to the Redis instance itself.

The CLI has multiple facilities for studying the latency of a Redis instance and understanding the latency's maximum, average and distribution.

The basic latency checking tool is the `--latency` option. Using this option the CLI runs a loop where the PING command is sent to the Redis instance, and the time to get a reply is measured. This happens 100 times per second, and stats are updated in a real time in the console:

```
$ redis-cli --latency
min: 0, max: 1, avg: 0.19 (427 samples)
```

The stats are provided in milliseconds. Usually, the average latency of a very fast instance tends to be overestimated a bit because of the latency due to the kernel scheduler of the system running `redis-cli` itself, so the average latency of 0.19 above may easily be 0.01 or less. However this is usually not a big problem, since we are interested in events of a few millisecond or more.

Sometimes it is useful to study how the maximum and average latencies evolve during time. The `--latency-history` option is used for that purpose: it works exactly like `--latency`, but every 15 seconds (by default) a new sampling session is started from scratch:

```
$ redis-cli --latency-history
min: 0, max: 1, avg: 0.14 (1314 samples) -- 15.01 seconds range
min: 0, max: 1, avg: 0.18 (1299 samples) -- 15.00 seconds range
min: 0, max: 1, avg: 0.20 (113 samples)^C
```

You can change the sampling sessions' length with the `-i <interval>` option.

The most advanced latency study tool, but also a bit harder to interpret for non experienced users, is the ability to use color terminals to show a spectrum of latencies. You'll see a colored output that indicates the different percentages of samples, and different ASCII characters that indicate different latency figures. This mode is enabled using the `--latency-dist` option:

```
$ redis-cli --latency-dist
(output not displayed, requires a color terminal, try it!)
```

There is another pretty unusual latency tool implemented inside `redis-cli`. It does not check the latency of a Redis instance, but the latency of the computer you are running `redis-cli` on. What latency you may ask? The latency that's intrinsic to the kernel scheduler, the hypervisor in case of virtualized instances, and so forth.

We call it *intrinsic latency* because it's opaque to the programmer, mostly. If your Redis instance has bad latency regardless of all the obvious things that may be the source cause, it's worth to check what's the best your system can do by running `redis-cli` in this special mode directly in the system you are running Redis servers on.

By measuring the intrinsic latency, you know that this is the baseline, and Redis cannot outdo your system. In order to run the CLI in this mode, use the `--intrinsic-latency <test-time>`. The test's time is in seconds, and specifies how many seconds `redis-cli` should check the latency of the system it's currently running on.

```
$ ./redis-cli --intrinsic-latency 5
Max latency so far: 1 microseconds.
Max latency so far: 7 microseconds.
Max latency so far: 9 microseconds.
Max latency so far: 11 microseconds.
Max latency so far: 13 microseconds.
Max latency so far: 15 microseconds.
Max latency so far: 34 microseconds.
Max latency so far: 82 microseconds.
Max latency so far: 586 microseconds.
Max latency so far: 739 microseconds.

65433042 total runs (avg latency: 0.0764 microseconds / 764.14 nanose
Worst run took 9671x longer than the average latency.
```

IMPORTANT: this command must be executed on the computer you want to run Redis server on, not on a different host. It does not even connect to a Redis instance and performs the test only locally.

In the above case, my system cannot do better than 739 microseconds of worst case latency, so I can expect certain queries to run in a bit less than 1 millisecond from time to time.

Remote backups of RDB files

During Redis replication's first synchronization, the master and the slave exchange the whole data set in form of an RDB file. This feature is exploited by `redis-cli` in order to provide a remote backup facility, that allows to transfer an RDB file from any Redis instance to the local computer running `redis-cli`. To use this mode, call the CLI with the `--rdb <dest-filename>` option:

```
$ redis-cli --rdb /tmp/dump.rdb
SYNC sent to master, writing 13256 bytes to '/tmp/dump.rdb'
Transfer finished with success.
```

This is a simple but effective way to make sure you have disaster recovery RDB backups of your Redis instance. However when using this options in scripts or `cron` jobs, make sure to check the return value of the command. If it is non zero, an error occurred like in the following example:

```
$ redis-cli --rdb /tmp/dump.rdb
SYNC with master failed: -ERR Can't SYNC while not connected with my
$ echo $?
1
```

Slave mode

The slave mode of the CLI is an advanced feature useful for Redis developers and for debugging operations. It allows to inspect what a master sends to its slaves in the replication stream in order to propagate the writes to its replicas. The option name is simply `--slave`. This is how it works:

```
$ redis-cli --slave
SYNC with master, discarding 13256 bytes of bulk transfer...
SYNC done. Logging commands from master.
"PING"
"SELECT","0"
"set","foo","bar"
"PING"
"incr","mycounter"
```

The command begins by discarding the RDB file of the first synchronization and then logs each command received as in CSV format.

If you think some of the commands are not replicated correctly in your slaves this is a good way to check what's happening, and also useful information in order to improve the bug report.

Performing an LRU simulation

Redis is often used as a cache with LRU eviction. Depending on the number of keys and the amount of memory allocated for the cache (specified via the `maxmemory` directive), the amount of cache hits and misses will change. Sometimes, simulating the rate of hits is very useful to correctly provision your cache.

The CLI has a special mode where it performs a simulation of GET and SET operations, using an 80-20% power law distribution in the requests pattern. This means that 20% of keys will be requested 80% of times, which is a common distribution in caching scenarios.

Theoretically, given the distribution of the requests and the Redis memory overhead, it should be possible to compute the hit rate analytically with with a mathematical formula. However, Redis can be configured with different LRU settings (number of samples) and LRU's implementation, which is approximated in Redis, changes a lot between different versions. Similarly the amount of memory per key may change between versions. That is why this tool was built: its main motivation was for testing the quality of Redis' LRU implementation, but now is also useful in for testing how a given version behaves with the settings you had in mind for your deployment.

In order to use this mode, you need to specify the amount of keys in the test. You also need to configure a `maxmemory` setting that makes sense as a first try.

IMPORTANT NOTE: Configuring the `maxmemory` setting in the Redis configuration is crucial: if there is no cap to the maximum memory usage, the hit will eventually be 100% since all the keys can be stored in memory. Or if you specify too many keys and no maximum memory, eventually all the computer RAM will be used. It is also needed to configure an appropriate *maxmemory policy*, most of the times what you want is `allkeys-lru`.

In the following example I configured a memory limit of 100MB, and an LRU simulation using 10 million keys.

WARNING: the test uses pipelining and will stress the server, don't use it with production instances.

```
$ ./redis-cli --lru-test 10000000
156000 Gets/sec | Hits: 4552 (2.92%) | Misses: 151448 (97.08%)
153750 Gets/sec | Hits: 12906 (8.39%) | Misses: 140844 (91.61%)
159250 Gets/sec | Hits: 21811 (13.70%) | Misses: 137439 (86.30%)
151000 Gets/sec | Hits: 27615 (18.29%) | Misses: 123385 (81.71%)
145000 Gets/sec | Hits: 32791 (22.61%) | Misses: 112209 (77.39%)
157750 Gets/sec | Hits: 42178 (26.74%) | Misses: 115572 (73.26%)
154500 Gets/sec | Hits: 47418 (30.69%) | Misses: 107082 (69.31%)
151250 Gets/sec | Hits: 51636 (34.14%) | Misses: 99614 (65.86%)
```

The program shows stats every second. As you see, in the first seconds the cache starts to be populated. The misses rate later stabilizes into the actual figure we can expect in the long time:

```
120750 Gets/sec | Hits: 48774 (40.39%) | Misses: 71976 (59.61%)
122500 Gets/sec | Hits: 49052 (40.04%) | Misses: 73448 (59.96%)
127000 Gets/sec | Hits: 50870 (40.06%) | Misses: 76130 (59.94%)
124250 Gets/sec | Hits: 50147 (40.36%) | Misses: 74103 (59.64%)
```

A miss rage of 59% may not be acceptable for our use case. So we know that 100MB of memory is not enough. Let's try with half gigabyte. After a few minutes we'll see the output stabilize to the following figures:

```
140000 Gets/sec | Hits: 135376 (96.70%) | Misses: 4624 (3.30%)
141250 Gets/sec | Hits: 136523 (96.65%) | Misses: 4727 (3.35%)
140250 Gets/sec | Hits: 135457 (96.58%) | Misses: 4793 (3.42%)
140500 Gets/sec | Hits: 135947 (96.76%) | Misses: 4553 (3.24%)
```

So we know that with 500MB we are going well enough for our number of keys (10 millions) and distribution (80-20 style).