EVAL script numkeys key [key ...] arg [arg ...]

> **Available since 2.6.0.**
>
> **Time complexity:** Depends on the script that is executed.

Introduction to EVAL

EVAL and EVALSHA are used to evaluate scripts using the Lua interpreter built into Redis starting from version 2.6.0.

The first argument of EVAL is a Lua 5.1 script. The script does not need to define a Lua function (and should not). It is just a Lua program that will run in the context of the Redis server.

The second argument of EVAL is the number of arguments that follows the script (starting from the third argument) that represent Redis key names. The arguments can be accessed by Lua using the KEYS global variable in the form of a one-based array (so `KEYS[1]`, `KEYS[2]`, ...).

All the additional arguments should not represent key names and can be accessed by Lua using the ARGV global variable, very similarly to what happens with keys (so `ARGV[1]`, `ARGV[2]`, ...).

The following example should clarify what stated above:

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

Note: as you can see Lua arrays are returned as Redis multi bulk replies, that is a Redis return type that your client library will likely convert into an Array type in your programming language.

It is possible to call Redis commands from a Lua script using two different Lua functions:

- `redis.call()`
- `redis.pcall()`

`redis.call()` is similar to `redis.pcall()`, the only difference is that if a Redis command call will result in an error, `redis.call()` will raise a Lua error that in turn will

force EVAL to return an error to the command caller, while `redis.pcall` will trap the error and return a Lua table representing the error.

The arguments of the `redis.call()` and `redis.pcall()` functions are all the arguments of a well formed Redis command:

```
> eval "return redis.call('set','foo','bar')" 0
OK
```

The above script sets the key `foo` to the string `bar`. However it violates the EVAL command semantics as all the keys that the script uses should be passed using the KEYS array:

```
> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
OK
```

All Redis commands must be analyzed before execution to determine which keys the command will operate on. In order for this to be true for EVAL, keys must be passed explicitly. This is useful in many ways, but especially to make sure Redis Cluster can forward your request to the appropriate cluster node.

Note this rule is not enforced in order to provide the user with opportunities to abuse the Redis single instance configuration, at the cost of writing scripts not compatible with Redis Cluster.

Lua scripts can return a value that is converted from the Lua type to the Redis protocol using a set of conversion rules.

Conversion between Lua and Redis data types

Redis return values are converted into Lua data types when Lua calls a Redis command using `call()` or `pcall()`. Similarly, Lua data types are converted into the Redis protocol when calling a Redis command and when a Lua script returns a value, so that scripts can control what EVAL will return to the client.

This conversion between data types is designed in a way that if a Redis type is converted into a Lua type, and then the result is converted back into a Redis type, the result is the same as the initial value.

In other words there is a one-to-one conversion between Lua and Redis types. The following table shows you all the conversions rules:

Redis to Lua conversion table.

- Redis integer reply -> Lua number
- Redis bulk reply -> Lua string

- Redis multi bulk reply -> Lua table (may have other Redis data types nested)
- Redis status reply -> Lua table with a single `ok` field containing the status
- Redis error reply -> Lua table with a single `err` field containing the error
- Redis Nil bulk reply and Nil multi bulk reply -> Lua false boolean type

Lua to Redis conversion table.

- Lua number -> Redis integer reply (the number is converted into an integer)
- Lua string -> Redis bulk reply
- Lua table (array) -> Redis multi bulk reply (truncated to the first nil inside the Lua array if any)
- Lua table with a single `ok` field -> Redis status reply
- Lua table with a single `err` field -> Redis error reply
- Lua boolean false -> Redis Nil bulk reply.

There is an additional Lua-to-Redis conversion rule that has no corresponding Redis to Lua conversion rule:

- Lua boolean true -> Redis integer reply with value of 1.

Lastly, there are three important rules to note:

- Lua has a single numerical type, Lua numbers. There is no distinction between integers and floats. So we always convert Lua numbers into integer replies, removing the decimal part of the number if any. **If you want to return a float from Lua you should return it as a string**, exactly like Redis itself does (see for instance the ZSCORE command).
- There is no simple way to have nils inside Lua arrays, this is a result of Lua table semantics, so when Redis converts a Lua array into Redis protocol the conversion is stopped if a nil is encountered.
- When a Lua table contains keys (and their values), the converted Redis reply will **not** include them.

RESP3 mode conversion rules: note that the Lua engine can work in RESP3 mode using the new Redis 6 protocol. In this case there are additional conversion rules, and certain conversions are also modified compared to the RESP2 mode. Please refer to the RESP3 section of this document for more information.

Here are a few conversion examples:

```
> eval "return 10" 0
(integer) 10

> eval "return {1,2,{3,'Hello World!'}}" 0
1) (integer) 1
2) (integer) 2
3) 1) (integer) 3
   2) "Hello World!"

> eval "return redis.call('get','foo')" 0
"bar"
```

The last example shows how it is possible to receive the exact return value of `redis.call()` or `redis.pcall()` from Lua that would be returned if the command was called directly.

In the following example we can see how floats and arrays containing nils and keys are handled:

```
> eval "return {1,2,3.3333,somekey='somevalue','foo',nil,'bar'}" 0
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) "foo"
```

As you can see 3.333 is converted into 3, *somekey* is excluded, and the *bar* string is never returned as there is a nil before.

Helper functions to return Redis types

There are two helper functions to return Redis types from Lua.

- `redis.error_reply(error_string)` returns an error reply. This function simply returns a single field table with the `err` field set to the specified string for you.
- `redis.status_reply(status_string)` returns a status reply. This function simply returns a single field table with the `ok` field set to the specified string for you.

There is no difference between using the helper functions or directly returning the table with the specified format, so the following two forms are equivalent:

```
return {err="My Error"}
return redis.error_reply("My Error")
```

Atomicity of scripts

Redis uses the same Lua interpreter to run all the commands. Also Redis guarantees that a script is executed in an atomic way: no other script or Redis command will be executed while a script is being executed. This semantic is similar to the one of MULTI / EXEC. From the point of view of all the other clients the effects of a script are either still not visible or already completed.

However this also means that executing slow scripts is not a good idea. It is not hard to create fast scripts, as the script overhead is very low, but if you are going to use slow scripts you should be aware that while the script is running no other client can execute commands.

Error handling

As already stated, calls to `redis.call()` resulting in a Redis command error will stop the execution of the script and return an error, in a way that makes it obvious that the error was generated by a script:

```
> del foo
(integer) 1
> lpush foo a
(integer) 1
> eval "return redis.call('get','foo')" 0
(error) ERR Error running script (call to f_6b1bf486c81ceb7edf3c093f4
```

Using `redis.pcall()` no error is raised, but an error object is returned in the format specified above (as a Lua table with an `err` field). The script can pass the exact error to the user by returning the error object returned by `redis.pcall()`.

Running Lua under low memory conditions

When the memory usage in Redis exceeds the `maxmemory` limit, the first write command encountered in the Lua script that uses additional memory will cause the script to abort (unless `redis.pcall` was used). However, one thing to caution here is that if the first write

command does not use additional memory such as DEL, LREM, or SREM, etc, Redis will allow it to run and all subsequent commands in the Lua script will execute to completion for atomicity. If the subsequent writes in the script generate additional memory, the Redis memory usage can go over `maxmemory`.

Another possible way for Lua script to cause Redis memory usage to go above `maxmemory` happens when the script execution starts when Redis is slightly below `maxmemory` so the first write command in the script is allowed. As the script executes, subsequent write commands continue to generate memory and causes the Redis server to go above `maxmemory`.

In those scenarios, it is recommended to configure the `maxmemory-policy` not to use `noeviction`. Also Lua scripts should be short so that evictions of items can happen in between Lua scripts.

Bandwidth and EVALSHA

The EVAL command forces you to send the script body again and again. Redis does not need to recompile the script every time as it uses an internal caching mechanism, however paying the cost of the additional bandwidth may not be optimal in many contexts.

On the other hand, defining commands using a special command or via `redis.conf` would be a problem for a few reasons:

- Different instances may have different implementations of a command.
- Deployment is hard if we have to make sure all instances contain a given command, especially in a distributed environment.
- Reading application code, the complete semantics might not be clear since the application calls commands defined server side.

In order to avoid these problems while avoiding the bandwidth penalty, Redis implements the EVALSHA command.

EVALSHA works exactly like EVAL, but instead of having a script as the first argument it has the SHA1 digest of a script. The behavior is the following:

- If the server still remembers a script with a matching SHA1 digest, the script is executed.
- If the server does not remember a script with this SHA1 digest, a special error is returned telling the client to use EVAL instead.

Example:

```
> set foo bar
OK
> eval "return redis.call('get','foo')" 0
"bar"
> evalsha 6b1bf486c81ceb7edf3c093f4c48582e38c0e791 0
"bar"
```

```
"bar"
> evalsha ffffffffffffffffffffffffffffffffffffffff 0
(error) `NOSCRIPT` No matching script. Please use [EVAL](/commands/ev
```

The client library implementation can always optimistically send EVALSHA under the hood even when the client actually calls EVAL, in the hope the script was already seen by the server. If the `NOSCRIPT` error is returned EVAL will be used instead.

Passing keys and arguments as additional EVAL arguments is also very useful in this context as the script string remains constant and can be efficiently cached by Redis.

Script cache semantics

Executed scripts are guaranteed to be in the script cache of a given execution of a Redis instance forever. This means that if an EVAL is performed against a Redis instance all the subsequent EVALSHA calls will succeed.

The reason why scripts can be cached for long time is that it is unlikely for a well written application to have enough different scripts to cause memory problems. Every script is conceptually like the implementation of a new command, and even a large application will likely have just a few hundred of them. Even if the application is modified many times and scripts will change, the memory used is negligible.

The only way to flush the script cache is by explicitly calling the SCRIPT FLUSH command, which will *completely flush* the scripts cache removing all the scripts executed so far.

This is usually needed only when the instance is going to be instantiated for another customer or application in a cloud environment.

Also, as already mentioned, restarting a Redis instance flushes the script cache, which is not persistent. However from the point of view of the client there are only two ways to make sure a Redis instance was not restarted between two different commands.

- The connection we have with the server is persistent and was never closed so far.
- The client explicitly checks the `runid` field in the INFO command in order to make sure the server was not restarted and is still the same process.

Practically speaking, for the client it is much better to simply assume that in the context of a given connection, cached scripts are guaranteed to be there unless an administrator explicitly called the SCRIPT FLUSH command.

The fact that the user can count on Redis not removing scripts is semantically useful in the context of pipelining.

For instance an application with a persistent connection to Redis can be sure that if a script was sent once it is still in memory, so EVALSHA can be used against those scripts in a pipeline without the chance of an error being generated due to an unknown script (we'll see this problem in detail later).

A common pattern is to call SCRIPT LOAD to load all the scripts that will appear in a pipeline, then use EVALSHA directly inside the pipeline without any need to check for errors resulting from the script hash not being recognized.

The SCRIPT command

Redis offers a SCRIPT command that can be used in order to control the scripting subsystem. SCRIPT currently accepts three different commands:

- SCRIPT FLUSH

  This command is the only way to force Redis to flush the scripts cache. It is most useful in a cloud environment where the same instance can be reassigned to a different user. It is also useful for testing client libraries' implementations of the scripting feature.

- `SCRIPT EXISTS sha1 sha2 ... shaN`

  Given a list of SHA1 digests as arguments this command returns an array of 1 or 0, where 1 means the specific SHA1 is recognized as a script already present in the scripting cache, while 0 means that a script with this SHA1 was never seen before (or at least never seen after the latest SCRIPT FLUSH command).

- `SCRIPT LOAD script`

  This command registers the specified script in the Redis script cache. The command is useful in all the contexts where we want to make sure that EVALSHA will not fail (for instance during a pipeline or MULTI/EXEC operation), without the need to actually execute the script.

- SCRIPT KILL

  This command is the only way to interrupt a long-running script that reaches the configured maximum execution time for scripts. The SCRIPT KILL command can only be used with scripts that did not modify the dataset during their execution (since stopping a read-only script does not violate the scripting engine's guaranteed atomicity). See the next sections for more information about long running scripts.

Scripts as pure functions

Note: starting with Redis 5, scripts are always replicated as effects and not sending the script verbatim. So the following section is mostly applicable to Redis version 4 or older.

A very important part of scripting is writing scripts that are pure functions. Scripts executed in a Redis instance are, by default, propagated to replicas and to the AOF file by sending the script itself -- not the resulting commands.

The reason is that sending a script to another Redis instance is often much faster than sending the multiple commands the script generates, so if the client is sending many scripts to the master, converting the scripts into individual commands for the replica / AOF would result in too much bandwidth for the replication link or the Append Only File (and

also too much CPU since dispatching a command received via network is a lot more work for Redis compared to dispatching a command invoked by Lua scripts).

Normally replicating scripts instead of the effects of the scripts makes sense, however not in all the cases. So starting with Redis 3.2, the scripting engine is able to, alternatively, replicate the sequence of write commands resulting from the script execution, instead of replication the script itself. See the next section for more information. In this section we'll assume that scripts are replicated by sending the whole script. Let's call this replication mode **whole scripts replication**.

The main drawback with the *whole scripts replication* approach is that scripts are required to have the following property:

- The script must always evaluates the same Redis *write* commands with the same arguments given the same input data set. Operations performed by the script cannot depend on any hidden (non-explicit) information or state that may change as script execution proceeds or between different executions of the script, nor can it depend on any external input from I/O devices.

Things like using the system time, calling Redis random commands like RANDOMKEY, or using Lua random number generator, could result into scripts that will not always evaluate in the same way.

In order to enforce this behavior in scripts Redis does the following:

- Lua does not export commands to access the system time or other external state.
- Redis will block the script with an error if a script calls a Redis command able to alter the data set **after** a Redis *random* command like RANDOMKEY, SRANDMEMBER, TIME. This means that if a script is read-only and does not modify the data set it is free to call those commands. Note that a *random command* does not necessarily mean a command that uses random numbers: any non-deterministic command is considered a random command (the best example in this regard is the TIME command).
- In Redis version 4, commands that may return elements in random order, like SMEMBERS (because Redis Sets are *unordered*) have a different behavior when called from Lua, and undergo a silent lexicographical sorting filter before returning data to Lua scripts. So `redis.call("smembers",KEYS[1])` will always return the Set elements in the same order, while the same command invoked from normal clients may return different results even if the key contains exactly the same elements. However starting with Redis 5 there is no longer such ordering step, because Redis 5 replicates scripts in a way that no longer needs non-deterministic commands to be converted into deterministic ones. In general, even when developing for Redis 4, never assume that certain commands in Lua will be ordered, but instead rely on the documentation of the original command you call to see the properties it provides.
- Lua pseudo random number generation functions `math.random` and `math.randomseed` are modified in order to always have the same seed every time a new script is executed. This means that calling `math.random` will always generate the

same sequence of numbers every time a script is executed if `math.randomseed` is not used.

However the user is still able to write commands with random behavior using the following simple trick. Imagine I want to write a Redis script that will populate a list with N random integers.

I can start with this small Ruby program:

```
require 'rubygems'
require 'redis'

r = Redis.new

RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,[:mylist],[10,rand(2**32)])
```

Every time this script executed the resulting list will have exactly the following elements:

```
> lrange mylist 0 -1
 1) "0.74509509873814"
 2) "0.87390407681181"
 3) "0.36876626981831"
 4) "0.6921941534114"
 5) "0.7857992587545"
 6) "0.57730350670279"
 7) "0.87046522734243"
 8) "0.09637165539729"
 9) "0.74990198051087"
10) "0.17082803611217"
```

In order to make it a pure function, but still be sure that every invocation of the script will result in different random elements, we can simply add an additional argument to the script that will be used in order to seed the Lua pseudo-random number generator. The new script is as follows:

```
RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    math.randomseed(tonumber(ARGV[2]))
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,1,:mylist,10,rand(2**32))
```

What we are doing here is sending the seed of the PRNG as one of the arguments. This way the script output will be the same given the same arguments, but we are changing one of the arguments in every invocation, generating the random seed client-side. The seed will be propagated as one of the arguments both in the replication link and in the Append Only File, guaranteeing that the same changes will be generated when the AOF is reloaded or when the replica processes the script.

Note: an important part of this behavior is that the PRNG that Redis implements as `math.random` and `math.randomseed` is guaranteed to have the same output regardless of the architecture of the system running Redis. 32-bit, 64-bit, big-endian and little-endian systems will all produce the same output.

Replicating commands instead of scripts

Note: starting with Redis 5, the replication method described in this section (scripts effects replication) is the default and does not need to be explicitly enabled.

Starting with Redis 3.2, it is possible to select an alternative replication method. Instead of replication whole scripts, we can just replicate single write commands generated by the script. We call this **script effects replication**.

In this replication mode, while Lua scripts are executed, Redis collects all the commands executed by the Lua scripting engine that actually modify the dataset. When the script

execution finishes, the sequence of commands that the script generated are wrapped into a MULTI / EXEC transaction and are sent to replicas and AOF.

This is useful in several ways depending on the use case:

- When the script is slow to compute, but the effects can be summarized by a few write commands, it is a shame to re-compute the script on the replicas or when reloading the AOF. In this case to replicate just the effect of the script is much better.
- When script effects replication is enabled, the controls about non deterministic functions are disabled. You can, for example, use the TIME or SRANDMEMBER commands inside your scripts freely at any place.
- The Lua PRNG in this mode is seeded randomly at every call.

In order to enable script effects replication, you need to issue the following Lua command before any write operated by the script:

```
redis.replicate_commands()
```

The function returns true if the script effects replication was enabled, otherwise if the function was called after the script already called some write command, it returns false, and normal whole script replication is used.

Selective replication of commands

When script effects replication is selected (see the previous section), it is possible to have more control in the way commands are replicated to replicas and AOF. This is a very advanced feature since **a misuse can do damage** by breaking the contract that the master, replicas, and AOF, all must contain the same logical content.

However this is a useful feature since, sometimes, we need to execute certain commands only in the master in order to create, for example, intermediate values.

Think at a Lua script where we perform an intersection between two sets. Pick five random elements, and create a new set with this five random elements. Finally we delete the temporary key representing the intersection between the two original sets. What we want to replicate is only the creation of the new set with the five elements. It's not useful to also replicate the commands creating the temporary key.

For this reason, Redis 3.2 introduces a new command that only works when script effects replication is enabled, and is able to control the scripting replication engine. The command is called `redis.set_repl()` and fails raising an error if called when script effects replication is disabled.

The command can be called with four different arguments:

```
redis.set_repl(redis.REPL_ALL) -- Replicate to AOF and replicas.
```

```
redis.set_repl(redis.REPL_AOF) -- Replicate only to AOF.
redis.set_repl(redis.REPL_REPLICA) -- Replicate only to replicas (Red
redis.set_repl(redis.REPL_SLAVE) -- Used for backward compatibility,
redis.set_repl(redis.REPL_NONE) -- Don't replicate at all.
```

By default the scripting engine is always set to REPL_ALL. By calling this function the user can switch on/off AOF and or replicas propagation, and turn them back later at her/his wish.

A simple example follows:

```
redis.replicate_commands() -- Enable effects replication.
redis.call('set','A','1')
redis.set_repl(redis.REPL_NONE)
redis.call('set','B','2')
redis.set_repl(redis.REPL_ALL)
redis.call('set','C','3')
```

After running the above script, the result is that only keys A and C will be created on replicas and AOF.

Global variables protection

Redis scripts are not allowed to create global variables, in order to avoid leaking data into the Lua state. If a script needs to maintain state between calls (a pretty uncommon need) it should use Redis keys instead.

When global variable access is attempted the script is terminated and EVAL returns with an error:

```
redis 127.0.0.1:6379> eval 'a=10' 0
(error) ERR Error running script (call to f_933044db579a2f8fd45d8065f
```

Accessing a *non existing* global variable generates a similar error.

Using Lua debugging functionality or other approaches like altering the meta table used to implement global protections in order to circumvent globals protection is not hard. However it is difficult to do it accidentally. If the user messes with the Lua global state, the consistency of AOF and replication is not guaranteed: don't do it.

Note for Lua newbies: in order to avoid using global variables in your scripts simply declare every variable you are going to use using the *local* keyword.

Using SELECT inside scripts

It is possible to call SELECT inside Lua scripts like with normal clients, However one subtle aspect of the behavior changes between Redis 2.8.11 and Redis 2.8.12. Before the 2.8.12 release the database selected by the Lua script was *transferred* to the calling script as current database. Starting from Redis 2.8.12 the database selected by the Lua script only affects the execution of the script itself, but does not modify the database selected by the client calling the script.

The semantic change between patch level releases was needed since the old behavior was inherently incompatible with the Redis replication layer and was the cause of bugs.

Using Lua scripting in RESP3 mode

Starting with Redis version 6, the server supports two different protocols. One is called RESP2, and is the old protocol: all the new connections to the server start in this mode. However clients are able to negotiate the new protocol using the HELLO command: this way the connection is put in RESP3 mode. In this mode certain commands, like for instance HGETALL, reply with a new data type (the Map data type in this specific case). The RESP3 protocol is semantically more powerful, however most scripts are OK with using just RESP2.

The Lua engine always assumes to run in RESP2 mode when talking with Redis, so whatever the connection that is invoking the EVAL or EVALSHA command is in RESP2 or RESP3 mode, Lua scripts will, by default, still see the same kind of replies they used to see in the past from Redis, when calling commands using the `redis.call()` built-in function.

However Lua scripts running in Redis 6 or greater, are able to switch to RESP3 mode, and get the replies using the new available types. Similarly Lua scripts are able to reply to clients using the new types. Please make sure to understand the capabilities for RESP3 before continuing reading this section.

In order to switch to RESP3 a script should call this function:

```
redis.setresp(3)
```

Note that a script can switch back and forth from RESP3 and RESP2 by calling the function with the argument '3' or '2'.

At this point the new conversions are available, specifically:

Redis to Lua conversion table specific to RESP3:

- Redis map reply -> Lua table with a single `map` field containing a Lua table representing the fields and values of the map.
- Redis set reply -> Lua table with a single `set` field containing a Lua table representing the elements of the set as fields, having as value just `true`.
- Redis new RESP3 single null value -> Lua nil.

- Redis true reply -> Lua true boolean value.
- Redis false reply -> Lua false boolean value.
- Redis double reply -> Lua table with a single `score` field containing a Lua number representing the double value.
- All the RESP2 old conversions still apply.

Lua to Redis conversion table specific for RESP3.

- Lua boolean -> Redis boolean true or false. **Note that this is a change compared to the RESP2 mode**, where returning true from Lua returned the number 1 to the Redis client, and returning false used to return NULL.
- Lua table with a single `map` field set to a field-value Lua table -> Redis map reply.
- Lua table with a single `set` field set to a field-value Lua table -> Redis set reply, the values are discarded and can be anything.
- Lua table with a single `double` field set to a field-value Lua table -> Redis double reply.
- Lua null -> Redis RESP3 new null reply (protocol `"_\r\n"`).
- All the RESP2 old conversions still apply unless specified above.

There is one key thing to understand: in case Lua replies with RESP3 types, but the connection calling Lua is in RESP2 mode, Redis will automatically convert the RESP3 protocol to RESP2 compatible protocol, as it happens for normal commands. For instance returning a map type to a connection in RESP2 mode will have the effect of returning a flat array of fields and values.

Available libraries

The Redis Lua interpreter loads the following Lua libraries:

- `base` lib.
- `table` lib.
- `string` lib.
- `math` lib.
- `struct` lib.
- `cjson` lib.
- `cmsgpack` lib.
- `bitop` lib.
- `redis.sha1hex` function.
- `redis.breakpoint and redis.debug` function in the context of the Redis Lua debugger.

Every Redis instance is *guaranteed* to have all the above libraries so you can be sure that the environment for your Redis scripts is always the same.

struct, CJSON and cmsgpack are external libraries, all the other libraries are standard Lua libraries.

struct

struct is a library for packing/unpacking structures within Lua.

```
Valid formats:
> – big endian
< – little endian
![num] – alignment
x – pading
b/B – signed/unsigned byte
h/H – signed/unsigned short
l/L – signed/unsigned long
T   – size_t
i/In – signed/unsigned integer with size `n' (default is size of int)
cn – sequence of `n' chars (from/to a string); when packing, n==0 mea
     the whole string; when unpacking, n==0 means use the previous
     read number as the string length
s – zero-terminated string
f – float
d – double
' ' – ignored
```

Example:

```
127.0.0.1:6379> eval 'return struct.pack("HH", 1, 2)' 0
"\x01\x00\x02\x00"
127.0.0.1:6379> eval 'return {struct.unpack("HH", ARGV[1])}' 0 "\x01\
1) (integer) 1
2) (integer) 2
3) (integer) 5
127.0.0.1:6379> eval 'return struct.size("HH")' 0
(integer) 4
```

CJSON

The CJSON library provides extremely fast JSON manipulation within Lua.

Example:

```
redis 127.0.0.1:6379> eval 'return cjson.encode({["foo"]= "bar"})' 0
"{\"foo\":\"bar\"}"
```

```
redis 127.0.0.1:6379> eval 'return cjson.decode(ARGV[1])["foo"]' 0 "{
"bar"
```

cmsgpack

The cmsgpack library provides simple and fast MessagePack manipulation within Lua.

Example:

```
127.0.0.1:6379> eval 'return cmsgpack.pack({"foo", "bar", "baz"})' 0
"\x93\xa3foo\xa3bar\xa3baz"
127.0.0.1:6379> eval 'return cmsgpack.unpack(ARGV[1])' 0 "\x93\xa3foo
1) "foo"
2) "bar"
3) "baz"
```

bitop

The Lua Bit Operations Module adds bitwise operations on numbers. It is available for scripting in Redis since version 2.8.18.

Example:

```
127.0.0.1:6379> eval 'return bit.tobit(1)' 0
(integer) 1
127.0.0.1:6379> eval 'return bit.bor(1,2,4,8,16,32,64,128)' 0
(integer) 255
127.0.0.1:6379> eval 'return bit.tohex(422342)' 0
"000671c6"
```

It supports several other functions: `bit.tobit`, `bit.tohex`, `bit.bnot`, `bit.band`, `bit.bor`, `bit.bxor`, `bit.lshift`, `bit.rshift`, `bit.arshift`, `bit.rol`, `bit.ror`, `bit.bswap`. All available functions are documented in the Lua BitOp documentation

`redis.sha1hex`

Perform the SHA1 of the input string.

Example:

```
127.0.0.1:6379> eval 'return redis.sha1hex(ARGV[1])' 0 "foo"
"0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33"
```

Emitting Redis logs from scripts

It is possible to write to the Redis log file from Lua scripts using the `redis.log` function.

```
redis.log(loglevel,message)
```

`loglevel` is one of:

- `redis.LOG_DEBUG`
- `redis.LOG_VERBOSE`
- `redis.LOG_NOTICE`
- `redis.LOG_WARNING`

They correspond directly to the normal Redis log levels. Only logs emitted by scripting using a log level that is equal or greater than the currently configured Redis instance log level will be emitted.

The `message` argument is simply a string. Example:

```
redis.log(redis.LOG_WARNING,"Something is wrong with this script.")
```

Will generate the following:

```
[32343] 22 Mar 15:21:39 # Something is wrong with this script.
```

Sandbox and maximum execution time

Scripts should never try to access the external system, like the file system or any other system call. A script should only operate on Redis data and passed arguments.

Scripts are also subject to a maximum execution time (five seconds by default). This default timeout is huge since a script should usually run in under a millisecond. The limit is mostly to handle accidental infinite loops created during development.

It is possible to modify the maximum time a script can be executed with millisecond precision, either via `redis.conf` or using the CONFIG GET / CONFIG SET command. The configuration parameter affecting max execution time is called `lua-time-limit`.

When a script reaches the timeout it is not automatically terminated by Redis since this violates the contract Redis has with the scripting engine to ensure that scripts are atomic. Interrupting a script means potentially leaving the dataset with half-written data. For this reasons when a script executes for more than the specified time the following happens:

- Redis logs that a script is running too long.
- It starts accepting commands again from other clients, but will reply with a BUSY error to all the clients sending normal commands. The only allowed commands in this status are SCRIPT KILL and `SHUTDOWN NOSAVE`.
- It is possible to terminate a script that executes only read-only commands using the SCRIPT KILL command. This does not violate the scripting semantic as no data was yet written to the dataset by the script.
- If the script already called write commands the only allowed command becomes `SHUTDOWN NOSAVE` that stops the server without saving the current data set on disk (basically the server is aborted).

EVALSHA in the context of pipelining

Care should be taken when executing EVALSHA in the context of a pipelined request, since even in a pipeline the order of execution of commands must be guaranteed. If EVALSHA will return a `NOSCRIPT` error the command can not be reissued later otherwise the order of execution is violated.

The client library implementation should take one of the following approaches:

- Always use plain EVAL when in the context of a pipeline.
- Accumulate all the commands to send into the pipeline, then check for EVAL commands and use the SCRIPT EXISTS command to check if all the scripts are already defined. If not, add SCRIPT LOAD commands on top of the pipeline as required, and use EVALSHA for all the EVAL calls.

Debugging Lua scripts

Starting with Redis 3.2, Redis has support for native Lua debugging. The Redis Lua debugger is a remote debugger consisting of a server, which is Redis itself, and a client, which is by default `redis-cli`.

The Lua debugger is described in the Lua scripts debugging section of the Redis documentation.

Related commands

- **EVAL**
- EVALSHA
- SCRIPT DEBUG
- SCRIPT EXISTS
- SCRIPT FLUSH

- SCRIPT KILL
- SCRIPT LOAD
