



Redis Modules: an introduction to the API

The modules documentation is composed of the following pages:

- Introduction to Redis modules (this file). An overview about Redis Modules system and API. It's a good idea to start your reading here.
- [Implementing native data types](#) covers the implementation of native data types into modules.
- [Blocking operations](#) shows how to write blocking commands that will not reply immediately, but will block the client, without blocking the Redis server, and will provide a reply whenever will be possible.
- [Redis modules API reference](#) is generated from module.c top comments of RedisModule functions. It is a good reference in order to understand how each function works.

Redis modules make it possible to extend Redis functionality using external modules, rapidly implementing new Redis commands with features similar to what can be done inside the core itself.

Redis modules are dynamic libraries that can be loaded into Redis at startup, or using the [MODULE LOAD](#) command. Redis exports a C API, in the form of a single C header file called `redismodule.h`. Modules are meant to be written in C, however it will be possible to use C++ or other languages that have C binding functionalities.

Modules are designed in order to be loaded into different versions of Redis, so a given module does not need to be designed, or recompiled, in order to run with a specific version of Redis. For this reason, the module will register to the Redis core using a specific API version. The current API version is "1".

This document is about an alpha version of Redis modules. API, functionalities and other details may change in the future.

Loading modules

In order to test the module you are developing, you can load the module using the following `redis.conf` configuration directive:

```
loadmodule /path/to/mymodule.so
```

It is also possible to load a module at runtime using the following command:

```
MODULE LOAD /path/to/mymodule.so
```

In order to list all loaded modules, use:

```
MODULE LIST
```

Finally, you can unload (and later reload if you wish) a module using the following command:

```
MODULE UNLOAD mymodule
```

Note that `mymodule` above is not the filename without the `.so` suffix, but instead, the name the module used to register itself into the Redis core. The name can be obtained using [MODULE LIST](#). However it is good practice that the filename of the dynamic library is the same as the name the module uses to register itself into the Redis core.

The simplest module you can write

In order to show the different parts of a module, here we'll show a very simple module that implements a command that outputs a random number.

```
#include "redismodule.h"
#include <stdlib.h>

int HelloworldRand_RedisCommand(RedisModuleCtx *ctx, RedisModuleString
    RedisModule_ReplyWithLongLong(ctx,rand());
    return REDISMODULE_OK;
}

int RedisModule_OnLoad(RedisModuleCtx *ctx, RedisModuleString **argv,
    if (RedisModule_Init(ctx,"helloworld",1,REDISMODULE_APIVER_1)
        == REDISMODULE_ERR) return REDISMODULE_ERR;

    if (RedisModule_CreateCommand(ctx,"helloworld.rand",
        HelloworldRand_RedisCommand, "fast random",
        0, 0, 0) == REDISMODULE_ERR)
        return REDISMODULE_ERR;
```

```
    return REDISMODULE_OK;
}
```

The example module has two functions. One implements a command called `HELLOWORLD.RAND`. This function is specific of that module. However the other function called `RedisModule_OnLoad()` must be present in each Redis module. It is the entry point for the module to be initialized, register its commands, and potentially other private data structures it uses.

Note that it is a good idea for modules to call commands with the name of the module followed by a dot, and finally the command name, like in the case of `HELLOWORLD.RAND`. This way it is less likely to have collisions.

Note that if different modules have colliding commands, they'll not be able to work in Redis at the same time, since the function `RedisModule_CreateCommand` will fail in one of the modules, so the module loading will abort returning an error condition.

Module initialization

The above example shows the usage of the function `RedisModule_Init()`. It should be the first function called by the module `OnLoad` function. The following is the function prototype:

```
int RedisModule_Init(RedisModuleCtx *ctx, const char *modulename,
                    int module_version, int api_version);
```

The `Init` function announces the Redis core that the module has a given name, its version (that is reported by [MODULE LIST](#)), and that is willing to use a specific version of the API.

If the API version is wrong, the name is already taken, or there are other similar errors, the function will return `REDISMODULE_ERR`, and the module `OnLoad` function should return ASAP with an error.

Before the `Init` function is called, no other API function can be called, otherwise the module will segfault and the Redis instance will crash.

The second function called, `RedisModule_CreateCommand`, is used in order to register commands into the Redis core. The following is the prototype:

```
int RedisModule_CreateCommand(RedisModuleCtx *ctx, const char *name,
                             RedisModuleCmdFunc cmdfunc, const char
                             int firstkey, int lastkey, int keystep)
```

As you can see, most Redis modules API calls all take as first argument the context of the module, so that they have a reference to the module calling it, to the command and client executing a given command, and so forth.

To create a new command, the above function needs the context, the command's name, a pointer to the function implementing the command, the command's flags and the positions of key names in the command's arguments.

The function that implements the command must have the following prototype:

```
int mycommand(RedisModuleCtx *ctx, RedisModuleString **argv, int argc
```

The command function arguments are just the context, that will be passed to all the other API calls, the command argument vector, and total number of arguments, as passed by the user.

As you can see, the arguments are provided as pointers to a specific data type, the `RedisModuleString`. This is an opaque data type you have API functions to access and use, direct access to its fields is never needed.

Zooming into the example command implementation, we can find another call:

```
int RedisModule_ReplyWithLongLong(RedisModuleCtx *ctx, long long inte
```

This function returns an integer to the client that invoked the command, exactly like other Redis commands do, like for example [INCR](#) or [SCARD](#).

Module cleanup

In most cases, there is no need for special cleanup. When a module is unloaded, Redis will automatically unregister commands and unsubscribe from notifications. However in the case where a module contains some persistent memory or configuration, a module may include an optional `RedisModule_OnUnload` function. If a module provides this function, it will be invoked during the module unload process. The following is the function prototype:

```
int RedisModule_OnUnload(RedisModuleCtx *ctx);
```

The `OnUnload` function may prevent module unloading by returning `REDISMODULE_ERR`. Otherwise, `REDISMODULE_OK` should be returned.

Setup and dependencies of a Redis module

Redis modules don't depend on Redis or some other library, nor they need to be compiled with a specific `redismodule.h` file. In order to create a new module, just copy a recent version of `redismodule.h` in your source tree, link all the libraries you want, and create a dynamic library having the `RedisModule_OnLoad()` function symbol exported.

The module will be able to load into different versions of Redis.

Passing configuration parameters to Redis modules

When the module is loaded with the `MODULE LOAD` command, or using the `loadmodule` directive in the `redis.conf` file, the user is able to pass configuration parameters to the module by adding arguments after the module file name:

```
loadmodule mymodule.so foo bar 1234
```

In the above example the strings `foo`, `bar` and `123` will be passed to the module `OnLoad()` function in the `argv` argument as an array of `RedisModuleString` pointers. The number of arguments passed is into `argc`.

The way you can access those strings will be explained in the rest of this document.

Normally the module will store the module configuration parameters in some `static` global variable that can be accessed module wide, so that the configuration can change the behavior of different commands.

Working with RedisModuleString objects

The command argument vector `argv` passed to module commands, and the return value of other module APIs functions, are of type `RedisModuleString`.

Usually you directly pass module strings to other API calls, however sometimes you may need to directly access the string object.

There are a few functions in order to work with string objects:

```
const char *RedisModule_StringPtrLen(RedisModuleString *string, size_
```

The above function accesses a string by returning its pointer and setting its length in `len`. You should never write to a string object pointer, as you can see from the `const` pointer qualifier.

However, if you want, you can create new string objects using the following API:

```
RedisModuleString *RedisModule_CreateString(RedisModuleCtx *ctx, const
```

The string returned by the above command must be freed using a corresponding call to `RedisModule_FreeString()`:

```
void RedisModule_FreeString(RedisModuleString *str);
```

However if you want to avoid having to free strings, the automatic memory management, covered later in this document, can be a good alternative, by doing it for you.

Note that the strings provided via the argument vector `argv` never need to be freed. You only need to free new strings you create, or new strings returned by other APIs, where it is specified that the returned string must be freed.

Creating strings from numbers or parsing strings as numbers

Creating a new string from an integer is a very common operation, so there is a function to do this:

```
RedisModuleString *mystr = RedisModule_CreateStringFromLongLong(ctx, 1
```

Similarly in order to parse a string as a number:

```
long long myval;  
if (RedisModule_StringToLongLong(ctx, argv[1], &myval) == REDISMODULE_0  
    /* Do something with 'myval' */  
}
```

Accessing Redis keys from modules

Most Redis modules, in order to be useful, have to interact with the Redis data space (this is not always true, for example an ID generator may never touch Redis keys). Redis modules have two different APIs in order to access the Redis data space, one is a low level API that provides very fast access and a set of functions to manipulate Redis data structures. The other API is more high level, and allows to call Redis commands and fetch the result, similarly to how Lua scripts access Redis.

The high level API is also useful in order to access Redis functionalities that are not available as APIs.

In general modules developers should prefer the low level API, because commands implemented using the low level API run at a speed comparable to the speed of native Redis commands. However there are definitely use cases for the higher level API. For example often the bottleneck could be processing the data and not accessing it.

Also note that sometimes using the low level API is not harder compared to the higher level one.

Calling Redis commands

The high level API to access Redis is the sum of the `RedisModule_Call()` function, together with the functions needed in order to access the reply object returned by `Call()`.

`RedisModule_Call` uses a special calling convention, with a format specifier that is used to specify what kind of objects you are passing as arguments to the function.

Redis commands are invoked just using a command name and a list of arguments. However when calling commands, the arguments may originate from different kind of strings: null-terminated C strings, `RedisModuleString` objects as received from the `argv` parameter in the command implementation, binary safe C buffers with a pointer and a length, and so forth.

For example if I want to call `INCRBY` using a first argument (the key) a string received in the argument vector `argv`, which is an array of `RedisModuleString` object pointers, and a C string representing the number "10" as second argument (the increment), I'll use the following function call:

```
RedisModuleCallReply *reply;  
reply = RedisModule_Call(ctx,"INCRBY","sc",argv[1],"10");
```

The first argument is the context, and the second is always a null terminated C string with the command name. The third argument is the format specifier where each character corresponds to the type of the arguments that will follow. In the above case "sc" means a `RedisModuleString` object, and a null terminated C string. The other arguments are just the two arguments as specified. In fact `argv[1]` is a `RedisModuleString` and "10" is a null terminated C string.

This is the full list of format specifiers:

- **c** -- Null terminated C string pointer.
- **b** -- C buffer, two arguments needed: C string pointer and `size_t` length.
- **s** -- `RedisModuleString` as received in `argv` or by other Redis module APIs returning a `RedisModuleString` object.
- **l** -- Long long integer.
- **v** -- Array of `RedisModuleString` objects.

- **!** -- This modifier just tells the function to replicate the command to replicas and AOF. It is ignored from the point of view of arguments parsing.
- **A** -- This modifier, when **!** is given, tells to suppress AOF propagation: the command will be propagated only to replicas.
- **R** -- This modifier, when **!** is given, tells to suppress replicas propagation: the command will be propagated only to the AOF if enabled.

The function returns a `RedisModuleCallReply` object on success, on error `NULL` is returned.

`NULL` is returned when the command name is invalid, the format specifier uses characters that are not recognized, or when the command is called with the wrong number of arguments. In the above cases the `errno` var is set to `EINVAL`. `NULL` is also returned when, in an instance with Cluster enabled, the target keys are about non local hash slots. In this case `errno` is set to `EPERM`.

Working with `RedisModuleCallReply` objects.

`RedisModuleCall` returns reply objects that can be accessed using the `RedisModule_CallReply*` family of functions.

In order to obtain the type or reply (corresponding to one of the data types supported by the Redis protocol), the function `RedisModule_CallReplyType()` is used:

```
reply = RedisModule_Call(ctx,"INCRBY","sc",argv[1],"10");
if (RedisModule_CallReplyType(reply) == REDISMODULE_REPLY_INTEGER) {
    long long myval = RedisModule_CallReplyInteger(reply);
    /* Do something with myval. */
}
```

Valid reply types are:

- `REDISMODULE_REPLY_STRING` Bulk string or status replies.
- `REDISMODULE_REPLY_ERROR` Errors.
- `REDISMODULE_REPLY_INTEGER` Signed 64 bit integers.
- `REDISMODULE_REPLY_ARRAY` Array of replies.
- `REDISMODULE_REPLY_NULL` `NULL` reply.

Strings, errors and arrays have an associated length. For strings and errors the length corresponds to the length of the string. For arrays the length is the number of elements. To obtain the reply length the following function is used:


```
size_t reply_len = RedisModule_CallReplyLength(reply);
```

In order to obtain the value of an integer reply, the following function is used, as already shown in the example above:

```
long long reply_integer_val = RedisModule_CallReplyInteger(reply);
```

Called with a reply object of the wrong type, the above function always returns `LLONG_MIN`. Sub elements of array replies are accessed this way:

```
RedisModuleCallReply *subreply;  
subreply = RedisModule_CallReplyArrayElement(reply, idx);
```

The above function returns `NULL` if you try to access out of range elements.

Strings and errors (which are like strings but with a different type) can be accessed using in the following way, making sure to never write to the resulting pointer (that is returned as `const` pointer so that misusing must be pretty explicit):

```
size_t len;  
char *ptr = RedisModule_CallReplyStringPtr(reply, &len);
```

If the reply type is not a string or an error, `NULL` is returned.

`RedisCallReply` objects are not the same as module string objects (`RedisModuleString` types). However sometimes you may need to pass replies of type string or integer, to API functions expecting a module string.

When this is the case, you may want to evaluate if using the low level API could be a simpler way to implement your command, or you can use the following function in order to create a new string object from a call reply of type string, error or integer:

```
RedisModuleString *mystr = RedisModule_CreateStringFromCallReply(myre
```

If the reply is not of the right type, `NULL` is returned. The returned string object should be released with `RedisModule_FreeString()` as usually, or by enabling automatic memory

management (see corresponding section).

Releasing call reply objects

Reply objects must be freed using `RedisModule_FreeCallReply`. For arrays, you need to free only the top level reply, not the nested replies. Currently the module implementation provides a protection in order to avoid crashing if you free a nested reply object for error, however this feature is not guaranteed to be here forever, so should not be considered part of the API.

If you use automatic memory management (explained later in this document) you don't need to free replies (but you still could if you wish to release memory ASAP).

Returning values from Redis commands

Like normal Redis commands, new commands implemented via modules must be able to return values to the caller. The API exports a set of functions for this goal, in order to return the usual types of the Redis protocol, and arrays of such types as elements. Also errors can be returned with any error string and code (the error code is the initial uppercase letters in the error message, like the "BUSY" string in the "BUSY the sever is busy" error message).

All the functions to send a reply to the client are called `RedisModule_ReplyWith<something>`.

To return an error, use:

```
RedisModule_ReplyWithError(RedisModuleCtx *ctx, const char *err);
```

There is a predefined error string for key of wrong type errors:

```
REDISMODULE_ERRORMSG_WRONGTYPE
```

Example usage:

```
RedisModule_ReplyWithError(ctx,"ERR invalid arguments");
```

We already saw how to reply with a long long in the examples above:

```
RedisModule_ReplyWithLongLong(ctx,12345);
```

To reply with a simple string, that can't contain binary values or newlines, (so it's suitable to send small words, like "OK") we use:

```
RedisModule_ReplyWithSimpleString(ctx,"OK");
```

It's possible to reply with "bulk strings" that are binary safe, using two different functions:

```
int RedisModule_ReplyWithStringBuffer(RedisModuleCtx *ctx, const char  
  
int RedisModule_ReplyWithString(RedisModuleCtx *ctx, RedisModuleStrin
```

The first function gets a C pointer and length. The second a RedisModuleString object. Use one or the other depending on the source type you have at hand.

In order to reply with an array, you just need to use a function to emit the array length, followed by as many calls to the above functions as the number of elements of the array are:

```
RedisModule_ReplyWithArray(ctx,2);  
RedisModule_ReplyWithStringBuffer(ctx,"age",3);  
RedisModule_ReplyWithLongLong(ctx,22);
```

To return nested arrays is easy, your nested array element just uses another call to `RedisModule_ReplyWithArray()` followed by the calls to emit the sub array elements.

Returning arrays with dynamic length

Sometimes it is not possible to know beforehand the number of items of an array. As an example, think of a Redis module implementing a FACTOR command that given a number outputs the prime factors. Instead of factorializing the number, storing the prime factors into an array, and later produce the command reply, a better solution is to start an array reply where the length is not known, and set it later. This is accomplished with a special argument to `RedisModule_ReplyWithArray()`:

```
RedisModule_ReplyWithArray(ctx, REDISMODULE_POSTPONED_ARRAY_LEN);
```

The above call starts an array reply so we can use other `ReplyWith` calls in order to produce the array items. Finally in order to set the length, use the following call:

```
RedisModule_ReplySetArrayLength(ctx, number_of_items);
```

In the case of the `FACTOR` command, this translates to some code similar to this:

```
RedisModule_ReplyWithArray(ctx, REDISMODULE_POSTPONED_ARRAY_LEN);
number_of_factors = 0;
while(still_factors) {
    RedisModule_ReplyWithLongLong(ctx, some_factor);
    number_of_factors++;
}
RedisModule_ReplySetArrayLength(ctx, number_of_factors);
```

Another common use case for this feature is iterating over the arrays of some collection and only returning the ones passing some kind of filtering.

It is possible to have multiple nested arrays with postponed reply. Each call to `SetArray()` will set the length of the latest corresponding call to `ReplyWithArray()`:

```
RedisModule_ReplyWithArray(ctx, REDISMODULE_POSTPONED_ARRAY_LEN);
... generate 100 elements ...
RedisModule_ReplyWithArray(ctx, REDISMODULE_POSTPONED_ARRAY_LEN);
... generate 10 elements ...
RedisModule_ReplySetArrayLength(ctx, 10);
RedisModule_ReplySetArrayLength(ctx, 100);
```

This creates a 100 items array having as last element a 10 items array.

Arity and type checks

Often commands need to check that the number of arguments and type of the key is correct. In order to report a wrong arity, there is a specific function called `RedisModule_WrongArity()`. The usage is trivial:

```
if (argc != 2) return RedisModule_WrongArity(ctx);
```

Checking for the wrong type involves opening the key and checking the type:

```
RedisModuleKey *key = RedisModule_OpenKey(ctx,argv[1],
    REDISMODULE_READ|REDISMODULE_WRITE);

int keytype = RedisModule_KeyType(key);
if (keytype != REDISMODULE_KEYTYPE_STRING &&
    keytype != REDISMODULE_KEYTYPE_EMPTY)
{
    RedisModule_CloseKey(key);
    return RedisModule_ReplyWithError(ctx,REDISMODULE_ERRORMSG_WRONGT
}
}
```

Note that you often want to proceed with a command both if the key is of the expected type, or if it's empty.

Low level access to keys

Low level access to keys allow to perform operations on value objects associated to keys directly, with a speed similar to what Redis uses internally to implement the built-in commands.

Once a key is opened, a key pointer is returned that will be used with all the other low level API calls in order to perform operations on the key or its associated value.

Because the API is meant to be very fast, it cannot do too many run-time checks, so the user must be aware of certain rules to follow:

- Opening the same key multiple times where at least one instance is opened for writing, is undefined and may lead to crashes.
- While a key is open, it should only be accessed via the low level key API. For example opening a key, then calling DEL on the same key using the `RedisModule_Call()` API will result into a crash. However it is safe to open a key, perform some operation with the low level API, closing it, then using other APIs to manage the same key, and later opening it again to do some more work.

In order to open a key the `RedisModule_OpenKey` function is used. It returns a key pointer, that we'll use with all the next calls to access and modify the value:

```
RedisModuleKey *key;
key = RedisModule_OpenKey(ctx,argv[1],REDISMODULE_READ);
```

The second argument is the key name, that must be a `RedisModuleString` object. The third argument is the mode: `REDISMODULE_READ` or `REDISMODULE_WRITE`. It is possible to use `|` to bitwise OR the two modes to open the key in both modes. Currently a key opened for writing can also be accessed for reading but this is to be considered an implementation detail. The right mode should be used in sane modules.

You can open non existing keys for writing, since the keys will be created when an attempt to write to the key is performed. However when opening keys just for reading, `RedisModule_OpenKey` will return `NULL` if the key does not exist.

Once you are done using a key, you can close it with:

```
RedisModule_CloseKey(key);
```

Note that if automatic memory management is enabled, you are not forced to close keys. When the module function returns, Redis will take care to close all the keys which are still open.

Getting the key type

In order to obtain the value of a key, use the `RedisModule_KeyType()` function:

```
int keytype = RedisModule_KeyType(key);
```

It returns one of the following values:

```
REDISMODULE_KEYTYPE_EMPTY  
REDISMODULE_KEYTYPE_STRING  
REDISMODULE_KEYTYPE_LIST  
REDISMODULE_KEYTYPE_HASH  
REDISMODULE_KEYTYPE_SET  
REDISMODULE_KEYTYPE_ZSET
```

The above are just the usual Redis key types, with the addition of an empty type, that signals the key pointer is associated with an empty key that does not yet exists.

Creating new keys

To create a new key, open it for writing and then write to it using one of the key writing functions. Example:

```
RedisModuleKey *key;  
key = RedisModule_OpenKey(ctx,argv[1],REDISMODULE_WRITE);  
if (RedisModule_KeyType(key) == REDISMODULE_KEYTYPE_EMPTY) {  
    RedisModule_StringSet(key,argv[2]);  
}
```

Deleting keys

Just use:

```
RedisModule_DeleteKey(key);
```

The function returns `REDISMODULE_ERR` if the key is not open for writing. Note that after a key gets deleted, it is setup in order to be targeted by new key commands. For example `RedisModule_KeyType()` will return it is an empty key, and writing to it will create a new key, possibly of another type (depending on the API used).

Managing key expires (TTLs)

To control key expires two functions are provided, that are able to set, modify, get, and unset the time to live associated with a key.

One function is used in order to query the current expire of an open key:

```
mstime_t RedisModule_GetExpire(RedisModuleKey *key);
```

The function returns the time to live of the key in milliseconds, or `REDISMODULE_NO_EXPIRE` as a special value to signal the key has no associated expire or does not exist at all (you can differentiate the two cases checking if the key type is `REDISMODULE_KEYTYPE_EMPTY`).

In order to change the expire of a key the following function is used instead:

```
int RedisModule_SetExpire(RedisModuleKey *key, mstime_t expire);
```

When called on a non existing key, `REDISMODULE_ERR` is returned, because the function can only associate expires to existing open keys (non existing open keys are only useful in order to create new values with data type specific write operations).

Again the `expire` time is specified in milliseconds. If the key has currently no expire, a new expire is set. If the key already have an expire, it is replaced with the new value.

If the key has an expire, and the special value `REDISMODULE_NO_EXPIRE` is used as a new expire, the expire is removed, similarly to the Redis [PERSIST](#) command. In case the key was already persistent, no operation is performed.

Obtaining the length of values

There is a single function in order to retrieve the length of the value associated to an open key. The returned length is value-specific, and is the string length for strings, and the number of elements for the aggregated data types (how many elements there is in a list, set, sorted set, hash).

```
size_t len = RedisModule_ValueLength(key);
```

If the key does not exist, 0 is returned by the function:

String type API

Setting a new string value, like the Redis [SET](#) command does, is performed using:

```
int RedisModule_StringSet(RedisModuleKey *key, RedisModuleString *str
```

The function works exactly like the Redis [SET](#) command itself, that is, if there is a prior value (of any type) it will be deleted.

Accessing existing string values is performed using DMA (direct memory access) for speed. The API will return a pointer and a length, so that's possible to access and, if needed, modify the string directly.

```
size_t len, j;  
char *myptr = RedisModule_StringDMA(key,&len,REDISMODULE_WRITE);  
for (j = 0; j < len; j++) myptr[j] = 'A';
```

In the above example we write directly on the string. Note that if you want to write, you must be sure to ask for `WRITE` mode.

DMA pointers are only valid if no other operations are performed with the key before using the pointer, after the DMA call.

Sometimes when we want to manipulate strings directly, we need to change their size as well. For this scope, the `RedisModule_StringTruncate` function is used. Example:

```
RedisModule_StringTruncate(mykey,1024);
```

The function truncates, or enlarges the string as needed, padding it with zero bytes if the previous length is smaller than the new length we request. If the string does not exist since key is associated to an open empty key, a string value is created and associated to the key. Note that every time `StringTruncate()` is called, we need to re-obtain the DMA pointer again, since the old may be invalid.

List type API

It's possible to push and pop values from list values:

```
int RedisModule_ListPush(RedisModuleKey *key, int where, RedisModuleS  
RedisModuleString *RedisModule_ListPop(RedisModuleKey *key, int where
```

In both the APIs the `where` argument specifies if to push or pop from tail or head, using the following macros:

```
REDISMODULE_LIST_HEAD  
REDISMODULE_LIST_TAIL
```

Elements returned by `RedisModule_ListPop()` are like strings created with `RedisModule_CreateString()`, they must be released with `RedisModule_FreeString()` or by enabling automatic memory management.

Set type API

Work in progress.

Sorted set type API

Documentation missing, please refer to the top comments inside `module.c` for the following functions:

- `RedisModule_ZsetAdd`

- `RedisModule_ZsetIncrby`
- `RedisModule_ZsetScore`
- `RedisModule_ZsetRem`

And for the sorted set iterator:

- `RedisModule_ZsetRangeStop`
- `RedisModule_ZsetFirstInScoreRange`
- `RedisModule_ZsetLastInScoreRange`
- `RedisModule_ZsetFirstInLexRange`
- `RedisModule_ZsetLastInLexRange`
- `RedisModule_ZsetRangeCurrentElement`
- `RedisModule_ZsetRangeNext`
- `RedisModule_ZsetRangePrev`
- `RedisModule_ZsetRangeEndReached`

Hash type API

Documentation missing, please refer to the top comments inside `module.c` for the following functions:

- `RedisModule_HashSet`
- `RedisModule_HashGet`

Iterating aggregated values

Work in progress.

Replicating commands

If you want to use module commands exactly like normal Redis commands, in the context of replicated Redis instances, or using the AOF file for persistence, it is important for module commands to handle their replication in a consistent way.

When using the higher level APIs to invoke commands, replication happens automatically if you use the `!"` modifier in the format string of `RedisModule_Call()` as in the following example:

```
reply = RedisModule_Call(ctx,"INCRBY","!sc",argv[1],"10");
```

As you can see the format specifier is `!"sc`. The bang is not parsed as a format specifier, but it internally flags the command as "must replicate".

If you use the above programming style, there are no problems. However sometimes things are more complex than that, and you use the low level API. In this case, if there are no side effects in the command execution, and it consistently always performs the same

work, what is possible to do is to replicate the command verbatim as the user executed it. To do that, you just need to call the following function:

```
RedisModule_ReplicateVerbatim(ctx);
```

When you use the above API, you should not use any other replication function since they are not guaranteed to mix well.

However this is not the only option. It's also possible to exactly tell Redis what commands to replicate as the effect of the command execution, using an API similar to `RedisModule_Call()` but that instead of calling the command sends it to the AOF / slaves stream. Example:

```
RedisModule_Replicate(ctx,"INCRBY","cl","foo",my_increment);
```

It's possible to call `RedisModule_Replicate` multiple times, and each will emit a command. All the sequence emitted is wrapped between a `MULTI/EXEC` transaction, so that the AOF and replication effects are the same as executing a single command.

Note that `Call()` replication and `Replicate()` replication have a rule, in case you want to mix both forms of replication (not necessarily a good idea if there are simpler approaches). Commands replicated with `Call()` are always the first emitted in the final `MULTI/EXEC` block, while all the commands emitted with `Replicate()` will follow.

Automatic memory management

Normally when writing programs in the C language, programmers need to manage memory manually. This is why the Redis modules API has functions to release strings, close open keys, free replies, and so forth.

However given that commands are executed in a contained environment and with a set of strict APIs, Redis is able to provide automatic memory management to modules, at the cost of some performance (most of the time, a very low cost).

When automatic memory management is enabled:

1. You don't need to close open keys.
2. You don't need to free replies.
3. You don't need to free `RedisModuleString` objects.

However you can still do it, if you want. For example, automatic memory management may be active, but inside a loop allocating a lot of strings, you may still want to free strings no longer used.

In order to enable automatic memory management, just call the following function at the start of the command implementation:

```
RedisModule_AutoMemory(ctx);
```

Automatic memory management is usually the way to go, however experienced C programmers may not use it in order to gain some speed and memory usage benefit.

Allocating memory into modules

Normal C programs use `malloc()` and `free()` in order to allocate and release memory dynamically. While in Redis modules the use of `malloc` is not technically forbidden, it is a lot better to use the Redis Modules specific functions, that are exact replacements for `malloc`, `free`, `realloc` and `strdup`. These functions are:

```
void *RedisModule_Alloc(size_t bytes);  
void* RedisModule_Realloc(void *ptr, size_t bytes);  
void RedisModule_Free(void *ptr);  
void RedisModule_Calloc(size_t nmemb, size_t size);  
char *RedisModule_Strdup(const char *str);
```

They work exactly like their `libc` equivalent calls, however they use the same allocator Redis uses, and the memory allocated using these functions is reported by the [INFO](#) command in the memory section, is accounted when enforcing the `maxmemory` policy, and in general is a first citizen of the Redis executable. On the contrary, the method allocated inside modules with `libc malloc()` is transparent to Redis.

Another reason to use the modules functions in order to allocate memory is that, when creating native data types inside modules, the RDB loading functions can return deserialized strings (from the RDB file) directly as `RedisModule_Alloc()` allocations, so they can be used directly to populate data structures after loading, instead of having to copy them to the data structure.

Pool allocator

Sometimes in commands implementations, it is required to perform many small allocations that will be not retained at the end of the command execution, but are just functional to execute the command itself.

This work can be more easily accomplished using the Redis pool allocator:

```
void *RedisModule_PoolAlloc(RedisModuleCtx *ctx, size_t bytes);
```

It works similarly to `malloc()`, and returns memory aligned to the next power of two of greater or equal to `bytes` (for a maximum alignment of 8 bytes). However it allocates memory in blocks, so the overhead of the allocations is small, and more important, the memory allocated is automatically released when the command returns.


So in general short living allocations are a good candidates for the pool allocator.

Writing commands compatible with Redis Cluster

Documentation missing, please check the following functions inside `module.c`:

```
RedisModule_IsKeysPositionRequest(ctx);  
RedisModule_KeyAtPos(ctx,pos);
```

This website is open source software. See all credits.

Sponsored by  **redislabs**
HOME OF REDIS