Native types in Redis modules

Redis modules can access Redis built-in data structures both at high level, by calling Redis commands, and at low level, by manipulating the data structures directly.

By using these capabilities in order to build new abstractions on top of existing Redis data structures, or by using strings DMA in order to encode modules data structures into Redis strings, it is possible to create modules that *feel like* they are exporting new data types. However, for more complex problems, this is not enough, and the implementation of new data structures inside the module is needed.

We call the ability of Redis modules to implement new data structures that feel like native Redis ones **native types support**. This document describes the API exported by the Redis modules system in order to create new data structures and handle the serialization in RDB files, the rewriting process in AOF, the type reporting via the TYPE command, and so forth.

Overview of native types

A module exporting a native type is composed of the following main parts:

- The implementation of some kind of new data structure and of commands operating on the new data structure.
- A set of callbacks that handle: RDB saving, RDB loading, AOF rewriting, releasing of a value associated with a key, calculation of a value digest (hash) to be used with the `DEBUG DIGEST` command.
- A 9 characters name that is unique to each module native data type.
- An encoding version, used to persist into RDB files a module-specific data version, so that a module will be able to load older representations from RDB files.

While to handle RDB loading, saving and AOF rewriting may look complex as a first glance, the modules API provide very high level function for handling all this, without requiring the user to handle read/write errors, so in practical terms, writing a new data structure for Redis is a simple task.

A **very easy** to understand but complete example of native type implementation is available inside the Redis distribution in the `/modules/hellotype.c` file. The reader is encouraged to read the documentation by looking at this example implementation to see how things are applied in the practice.

Registering a new data type

In order to register a new native type into the Redis core, the module needs to declare a global variable that will hold a reference to the data type. The API to register the data type

will return a data type reference that will be stored in the global variable.

```
static RedisModuleType *MyType;
#define MYTYPE_ENCODING_VERSION 0

int RedisModule_OnLoad(RedisModuleCtx *ctx) {
RedisModuleTypeMethods tm = {
    .version = REDISMODULE_TYPE_METHOD_VERSION,
    .rdb_load = MyTypeRDBLoad,
    .rdb_save = MyTypeRDBSave,
    .aof_rewrite = MyTypeAOFRewrite,
    .free = MyTypeFree
};

    MyType = RedisModule_CreateDataType(ctx, "MyType-AZ",
    MYTYPE_ENCODING_VERSION, &tm);
    if (MyType == NULL) return REDISMODULE_ERR;
}
```

As you can see from the example above, a single API call is needed in order to register the new type. However a number of function pointers are passed as arguments. Certain are optionals while some are mandatory. The above set of methods *must* be passed, while `.digest` and `.mem_usage` are optional and are currently not actually supported by the modules internals, so for now you can just ignore them.

The `ctx` argument is the context that we receive in the `OnLoad` function. The type `name` is a 9 character name in the character set that includes from `A-Z`, `a-z`, `0-9`, plus the underscore _ and minus - characters.

Note that **this name must be unique** for each data type in the Redis ecosystem, so be creative, use both lower-case and upper case if it makes sense, and try to use the convention of mixing the type name with the name of the author of the module, to create a 9 character unique name.

NOTE: It is very important that the name is exactly 9 chars or the registration of the type will fail. Read more to understand why.

For example if I'm building a *b-tree* data structure and my name is *antirez* I'll call my type **btree1-az**. The name, converted to a 64 bit integer, is stored inside the RDB file when saving the type, and will be used when the RDB data is loaded in order to resolve what module can load the data. If Redis finds no matching module, the integer is converted back to a name in order to provide some clue to the user about what module is missing in order to load the data.

The type name is also used as a reply for the TYPE command when called with a key holding the registered type.

The `encver` argument is the encoding version used by the module to store data inside the RDB file. For example I can start with an encoding version of 0, but later when I release version 2.0 of my module, I can switch encoding to something better. The new module will register with an encoding version of 1, so when it saves new RDB files, the new version will be stored on disk. However when loading RDB files, the module `rdb_load` method will be called even if there is data found for a different encoding version (and the encoding version is passed as argument to `rdb_load`), so that the module can still load old RDB files.

The last argument is a structure used in order to pass the type methods to the registration function: `rdb_load`, `rdb_save`, `aof_rewrite`, `digest` and `free` and `mem_usage` are all callbacks with the following prototypes and uses:

```
typedef void *(*RedisModuleTypeLoadFunc)(RedisModuleIO *rdb, int encv
typedef void (*RedisModuleTypeSaveFunc)(RedisModuleIO *rdb, void *val
typedef void (*RedisModuleTypeRewriteFunc)(RedisModuleIO *aof, RedisM
typedef size_t (*RedisModuleTypeMemUsageFunc)(void *value);
typedef void (*RedisModuleTypeDigestFunc)(RedisModuleDigest *digest,
typedef void (*RedisModuleTypeFreeFunc)(void *value);
```

- `rdb_load` is called when loading data from the RDB file. It loads data in the same format as `rdb_save` produces.
- `rdb_save` is called when saving data to the RDB file.
- `aof_rewrite` is called when the AOF is being rewritten, and the module needs to tell Redis what is the sequence of commands to recreate the content of a given key.
- `digest` is called when `DEBUG DIGEST` is executed and a key holding this module type is found. Currently this is not yet implemented so the function ca be left empty.
- `mem_usage` is called when the `MEMORY` command asks for the total memory consumed by a specific key, and is used in order to get the amount of bytes used by the module value.
- `free` is called when a key with the module native type is deleted via DEL or in any other mean, in order to let the module reclaim the memory associated with such a value.

Ok, but *why* modules types require a 9 characters name?

Oh, I understand you need to understand this, so here is a very specific explanation.

When Redis persists to RDB files, modules specific data types require to be persisted as well. Now RDB files are sequences of key-value pairs like the following:

```
[1 byte type] [key] [a type specific value]
```

The 1 byte type identifies strings, lists, sets, and so forth. In the case of modules data, it is set to a special value of `module data`, but of course this is not enough, we need the information needed to link a specific value with a specific module type that is able to load and handle it.

So when we save a `type specific value` about a module, we prefix it with a 64 bit integer. 64 bits is large enough to store the informations needed in order to lookup the module that can handle that specific type, but is short enough that we can prefix each module value we store inside the RDB without making the final RDB file too big. At the same time, this solution of prefixing the value with a 64 bit *signature* does not require to do strange things like defining in the RDB header a list of modules specific types. Everything is pretty simple.

So, what you can store in 64 bits in order to identify a given module in a reliable way? Well if you build a character set of 64 symbols, you can easily store 9 characters of 6 bits, and you are left with 10 bits, that are used in order to store the *encoding version* of the type, so that the same type can evolve in the future and provide a different and more efficient or updated serialization format for RDB files.

So the 64 bit prefix stored before each module value is like the following:

```
6|6|6|6|6|6|6|6|6|10
```

The first 9 elements are 6-bits characters, the final 10 bits is the encoding version.

When the RDB file is loaded back, it reads the 64 bit value, masks the final 10 bits, and searches for a matching module in the modules types cache. When a matching one is found, the method to load the RDB file value is called with the 10 bits encoding version as argument, so that the module knows what version of the data layout to load, if it can support multiple versions.

Now the interesting thing about all this is that, if instead the module type cannot be resolved, since there is no loaded module having this signature, we can convert back the 64 bit value into a 9 characters name, and print an error to the user that includes the module type name! So that she or he immediately realizes what's wrong.

Setting and getting keys

After registering our new data type in the `RedisModule_OnLoad()` function, we also need to be able to set Redis keys having as value our native type.

This normally happens in the context of commands that write data to a key. The native types API allow to set and get keys to module native data types, and to test if a given key is already associated to a value of a specific data type.

The API uses the normal modules `RedisModule_OpenKey()` low level key access interface in order to deal with this. This is an example of setting a native type private data structure to a Redis key:

```
RedisModuleKey *key = RedisModule_OpenKey(ctx,keyname,REDISMODULE_WRI
struct some_private_struct *data = createMyDataStructure();
RedisModule_ModuleTypeSetValue(key,MyType,data);
```

The function `RedisModule_ModuleTypeSetValue()` is used with a key handle open for writing, and gets three arguments: the key handle, the reference to the native type, as obtained during the type registration, and finally a `void*` pointer that contains the private data implementing the module native type.

Note that Redis has no clues at all about what your data contains. It will just call the callbacks you provided during the method registration in order to perform operations on the type.

Similarly we can retrieve the private data from a key using this function:

```
struct some_private_struct *data;
data = RedisModule_ModuleTypeGetValue(key);
```

We can also test for a key to have our native type as value:

```
if (RedisModule_ModuleTypeGetType(key) == MyType) {
    /* ... do something ... */
}
```

However for the calls to do the right thing, we need to check if the key is empty, if it contains a value of the right kind, and so forth. So the idiomatic code to implement a command writing to our native type is along these lines:

```
RedisModuleKey *key = RedisModule_OpenKey(ctx,argv[1],
    REDISMODULE_READ|REDISMODULE_WRITE);
int type = RedisModule_KeyType(key);
if (type != REDISMODULE_KEYTYPE_EMPTY &&
```

```
if (type != REDISMODULE_KEYTYPE_EMPTY &&
    RedisModule_ModuleTypeGetType(key) != MyType)
{
    return RedisModule_ReplyWithError(ctx,REDISMODULE_ERRORMSG_WRONGT
}
```

Then if we successfully verified the key is not of the wrong type, and we are going to write to it, we usually want to create a new data structure if the key is empty, or retrieve the reference to the value associated to the key if there is already one:

```
/* Create an empty value object if the key is currently empty. */
struct some_private_struct *data;
if (type == REDISMODULE_KEYTYPE_EMPTY) {
    data = createMyDataStructure();
    RedisModule_ModuleTypeSetValue(key,MyTyke,data);
} else {
    data = RedisModule_ModuleTypeGetValue(key);
}
/* Do something with 'data'... */
```

Free method

As already mentioned, when Redis needs to free a key holding a native type value, it needs help from the module in order to release the memory. This is the reason why we pass a `free` callback during the type registration:

```
typedef void (*RedisModuleTypeFreeFunc)(void *value);
```

A trivial implementation of the free method can be something like this, assuming our data structure is composed of a single allocation:

```
void MyTypeFreeCallback(void *value) {
    RedisModule_Free(value);
}
```

However a more real world one will call some function that performs a more complex memory reclaiming, by casting the void pointer to some structure and freeing all the resources composing the value.

RDB load and save methods

The RDB saving and loading callbacks need to create (and load back) a representation of the data type on disk. Redis offers an high level API that can automatically store inside the RDB file the following types:

- Unsigned 64 bit integers.
- Signed 64 bit integers.
- Doubles.
- Strings.

It is up to the module to find a viable representation using the above base types. However note that while the integer and double values are stored and loaded in an architecture and *endianness* agnostic way, if you use the raw string saving API to, for example, save a structure on disk, you have to care those details yourself.

This is the list of functions performing RDB saving and loading:

```
void RedisModule_SaveUnsigned(RedisModuleIO *io, uint64_t value);
uint64_t RedisModule_LoadUnsigned(RedisModuleIO *io);
void RedisModule_SaveSigned(RedisModuleIO *io, int64_t value);
int64_t RedisModule_LoadSigned(RedisModuleIO *io);
void RedisModule_SaveString(RedisModuleIO *io, RedisModuleString *s);
void RedisModule_SaveStringBuffer(RedisModuleIO *io, const char *str,
RedisModuleString *RedisModule_LoadString(RedisModuleIO *io);
char *RedisModule_LoadStringBuffer(RedisModuleIO *io, size_t *lenptr)
void RedisModule_SaveDouble(RedisModuleIO *io, double value);
double RedisModule_LoadDouble(RedisModuleIO *io);
```

The functions don't require any error checking from the module, that can always assume calls succeed.

As an example, imagine I've a native type that implements an array of double values, with the following structure:

```
struct double_array {
    size_t count;
    double *values;
};
```

My `rdb_save` method may look like the following:

```
void DoubleArrayRDBSave(RedisModuleIO *io, void *ptr) {
    struct dobule_array *da = ptr;
    RedisModule_SaveUnsigned(io,da->count);
    for (size_t j = 0; j < da->count; j++)
        RedisModule_SaveDouble(io,da->values[j]);
}
```

What we did was to store the number of elements followed by each double value. So when later we'll have to load the structure in the `rdb_load` method we'll do something like this:

```
void *DoubleArrayRDBLoad(RedisModuleIO *io, int encver) {
    if (encver != DOUBLE_ARRAY_ENC_VER) {
        /* We should actually log an error here, or try to implement
           the ability to load older versions of our data structure.
        return NULL;
    }

    struct double_array *da;
    da = RedisModule_Alloc(sizeof(*da));
    da->count = RedisModule_LoadUnsigned(io);
    da->values = RedisModule_Alloc(da->count * sizeof(double));
    for (size_t j = 0; j < da->count; j++)
        da->values[j] = RedisModule_LoadDouble(io);
    return da;
}
```

The load callback just reconstruct back the data structure from the data we stored in the RDB file.

Note that while there is no error handling on the API that writes and reads from disk, still the load callback can return NULL on errors in case what it reads does not look correct. Redis will just panic in that case.

AOF rewriting

```
void RedisModule_EmitAOF(RedisModuleIO *io, const char *cmdname, cons
```

Handling multiple encodings

```
WORK IN PROGRESS
```

Allocating memory

Modules data types should try to use `RedisModule_Alloc()` functions family in order to allocate, reallocate and release heap memory used to implement the native data structures (see the other Redis Modules documentation for detailed information).

This is not just useful in order for Redis to be able to account for the memory used by the module, but there are also more advantages:

- Redis uses the `jemalloc` allocator, that often prevents fragmentation problems that could be caused by using the libc allocator.
- When loading strings from the RDB file, the native types API is able to return strings allocated directly with `RedisModule_Alloc()`, so that the module can directly link this memory into the data structure representation, avoiding an useless copy of the data.

Even if you are using external libraries implementing your data structures, the allocation functions provided by the module API is exactly compatible with `malloc()`, `realloc()`, `free()` and `strdup()`, so converting the libraries in order to use these functions should be trivial.

In case you have an external library that uses libc `malloc()`, and you want to avoid replacing manually all the calls with the Redis Modules API calls, an approach could be to use simple macros in order to replace the libc calls with the Redis API calls. Something like this could work:

```
#define malloc RedisModule_Alloc
#define realloc RedisModule_Realloc
#define free RedisModule_Free
#define strdup RedisModule_Strdup
```

However take in mind that mixing libc calls with Redis API calls will result into troubles and crashes, so if you replace calls using macros, you need to make sure that all the calls are correctly replaced, and that the code with the substituted calls will never, for example, attempt to call `RedisModule_Free()` with a pointer allocated using libc `malloc()`.
