Redis RDB Dump File Format

Redis RDB File Format

Redis *.rdb file is a binary representation of the in-memory store. This binary file is sufficient to completely restore Redis' state.

The rdb file format is optimized for fast read and writes. Where possible LZF compression is used to reduce the file size. In general, objects are prefixed with their lengths, so before reading the object you know exactly how much memory to allocate.

Optimizing for fast read/writes means the on-disk format should be as close as possible to the in-memory representation. This is the approach taken by the rdb file. As a consequence, you cannot parse the rdb file without some understanding of Redis' in-memory representation of data structures

High Level Algorithm to parse RDB

At a high level, the RDB file has the following structure

```
----------------------------# RDB is a binary format. There are no new
lines or spaces in the file.
52 45 44 49 53             # Magic String "REDIS"
30 30 30 37                # 4 digit ASCCII RDB Version Number. In this
case, version = "0007" = 7
----------------------------
FE 00                      # FE = code that indicates database selector. db
number = 00
----------------------------# Key-Value pair starts
FD $unsigned int           # FD indicates "expiry time in seconds". After
that, expiry time is read as a 4 byte unsigned int
$value-type                # 1 byte flag indicating the type of value - set,
map, sorted set etc.
$string-encoded-key        # The key, encoded as a redis string
$encoded-value             # The value. Encoding depends on $value-type
----------------------------
FC $unsigned long          # FC indicates "expiry time in ms". After that,
expiry time is read as a 8 byte unsigned long
```

```
$value-type                # 1 byte flag indicating the type of value - set,
map, sorted set etc.
$string-encoded-key         # The key, encoded as a redis string
$encoded-value              # The value. Encoding depends on $value-type
----------------------------
$value-type                # This key value pair doesn't have an expiry.
$value_type guaranteed != to FD, FC, FE and FF
$string-encoded-key
$encoded-value
----------------------------
FE $length-encoding         # Previous db ends, next db starts. Database
number read using length encoding.
----------------------------
...                        # Key value pairs for this database, additonal
database

FF                         ## End of RDB file indicator
8 byte checksum             ## CRC 64 checksum of the entire file.
```

Magic Number

The file starts off with the magic string "REDIS". This is a quick sanity check to know we are dealing with a redis rdb file.

```
52 45 44 49 53 # "REDIS"
```

RDB Version Number

The next 4 bytes store the version number of the rdb format. The 4 bytes are interpreted as ascii characters and then converted to an integer using string to integer conversion.

```
00 00 00 03 # Version = 3
```

Database Selector

A Redis instance can have multiple databases.

A single byte `0xFE` flags the start of the database selector. After this byte, a variable length field indicates the database number. See the section "Length Encoding" to understand how to read this database number.

Key Value Pairs

After the database selector, the file contains a sequence of key value pairs.

Each key value pair has 4 parts -

1. Key Expiry Timestamp. This is optional
2. One byte flag indicating the value type
3. The key, encoded as a Redis String. See "Redis String Encoding"
4. The value, encoded according to the value type. See "Redis Value Encoding"

Key Expiry Timestamp

This section starts with a one byte flag. A value of `FD` indicates expiry is specified in seconds. A value of `FC` indicates expiry in specified in milliseconds.
If time is specified in ms, the next 8 bytes represent the unix time This number is an unix time stamp in either seconds or milliseconds precision, and represents the expiry of this key.

See the section "Redis Length Encoding" on how this number is encoded.

During the import process, keys that have expired must be discarded.

Value Type

A one byte flag indicates encoding used to save the Value.

1. 0 = "String Encoding"
2. 1 = "List Encoding"
3. 2 = "Set Encoding"
4. 3 = "Sorted Set Encoding"
5. 4 = "Hash Encoding"
6. 9 = "Zipmap Encoding"
7. 10 = "Ziplist Encoding"
8. 11 = "Intset Encoding"
9. 12 = "Sorted Set in Ziplist Encoding"
10. 13 = "Hashmap in Ziplist Encoding" (Introduced in rdb version 4)

Key

The key is simply encoded as a Redis string. See the section "String Encoding" to learn how the key is encoded.

Value

The encoding of the value depends on the value type flag.

- When value type = 0, the value is a simple string.
- When value type is one of 9, 10, 11 or 12, the value is wrapped in a string. After reading the string, it must be parsed further.
- When value type is one of 1, 2, 3 or 4, the value is a sequence of strings. This sequence of strings is used to construct a list, set, sorted set or hashmap.

Length Encoding

Length encoding is used to store the length of the next object in the stream. Length encoding is a variable byte encoding designed to use as few bytes as possible.

This is how length encoding works :

1. One byte is read from the stream, and the two most significant bits are read.
2. If starting bits are `00`, then the next 6 bits represent the length
3. If starting bits are `01`, then an additional byte is read from the stream. The combined 14 bits represent the length
4. If starting bits are `10`, then the remaining 6 bits are discarded. Additional 4 bytes are read from the stream, and those 4 bytes represent the length (in big endian format in RDB version 6)
5. If starting bits are `11`, then the next object is encoded in a special format. The remaining 6 bits indicate the format. This encoding is generally used to store numbers as strings, or to store encoded strings. See String Encoding

As a result of this encoding -

1. Numbers upto and including 63 can be stored in 1 byte
2. Numbers upto and including 16383 can be stored in 2 bytes
3. Numbers upto 2^32 -1 can be stored in 5 bytes

String Encoding

Redis Strings are binary safe - which means you can store anything in them. They do not have any special end-of-string token. It is best to think of Redis Strings as a byte array.

There are three types of Strings in Redis -

1. Length prefixed strings
2. An 8, 16 or 32 bit integer
3. A LZF compressed string

Length Prefixed String

Length prefixed strings are quite simple. The length of the string in bytes is first encoded using "Length Encoding". After this, the raw bytes of the string are stored.

Integers as String

First read the section "Length Encoding", specifically the part when the first two bits are `11`. In this case, the remaining 6 bits are read. If the value of those 6 bits is -

1. 0 indicates that an 8 bit integer follows
2. 1 indicates that a 16 bit integer follows
3. 2 indicates that a 32 bit integer follows

Compressed Strings

First read the section "Length Encoding", specifically the part when the first two bits are `11`. In this case, the remaining 6 bits are read. If the value of those 6 bits is 4, it indicates that a compressed string follows.
The compressed string is read as follows -

1. The compressed length `clen` is read from the stream using "Length Encoding"
2. The uncompressed length is read from the stream using "Length Encoding"
3. The next `clen` bytes are read from the stream
4. Finally, these bytes are decompressed using LZF algorithm

List Encoding

A redis list is represented as a sequence of strings.

1. First, the size of the list `size` is read from the stream using "Length Encoding"
2. Next, `size` strings are read from the stream using "String Encoding"
3. The list is then re-constructed using these Strings

Set Encoding

Sets are encoded exactly like lists.

Sorted Set Encoding

1. First, the size of the sorted set `size` is read from the stream using "Length Encoding"
2. TODO

Hash Encoding

1. First, the size of the hash `size` is read from the stream using "Length Encoding"
2. Next, `2 * size` strings are read from the stream using "String Encoding"
3. Alternate strings are key and values
4. For example, `2 us washington india delhi` represents the map `{"us" => "washington", "india" => "delhi"}`

Zipmap Encoding

NOTE : Zipmap encoding are deprecated starting Redis 2.6. Small hashmaps are now encoded using ziplists.

A Zipmap is a hashmap that has been serialized to a string. In essence, the key value pairs are stored sequentially. Looking up a key in this structure is O(N). This structure is used instead of a dictionary when the number of key value pairs are small.

To parse a zipmap, first a string is read from the stream using "String Encoding". This string is the envelope of the zipmap. The contents of this string represent the zipmap.

The structure of a zipmap within this string is as follows -

```
<zmlen><len>"foo"<len><free>"bar"<len>"hello"<len><free>"world"<zmend>
```

1. *zmlen* : Is a 1 byte length that holds the size of the zip map. If it is greater than or equal to 254, value is not used. You will have to iterate the entire zip map to find the length.
2. *len* : Is the length of the following string, which can be either a key or a value. This length is stored in either 1 byte or 5 bytes (yes, it differs from "Length Encoding" described above). If the first byte is between 0 and 252, that is the length of the zipmap. If the first byte is 253, then the next 4 bytes read as an unsigned integer represent the length of the zipmap. 254 and 255 are invalid values for this field.
3. *free* : This is always 1 byte, and indicates the number of free bytes *after* the value. For example, if the value of a key is "America" and its get updated to "USA", 4 free bytes will be available.
4. *zmend* : Always 255. Indicates the end of the zipmap.

Worked Example

```
18 02 06 4d 4b 44 31 47 36 01 00 32 05 59 4e 4e 58 4b 04 00 46 37 54 49
ff ..
```

1. Start by decoding this using "String Encoding". You will notice that `18` is the length of the string. Accordingly, we will read the next 24 bytes i.e. upto `ff`
2. Now, we are parsing the string starting at `02 06...` using the "Zipmap Encoding"
3. `02` is the number of entries in the hashmap.
4. `06` is the length of the next string. Since this is less than 254, we don't have to read any additional bytes
5. We read the next 6 bytes i.e. `4d 4b 44 31 47 36` to get the key "MKD1G6"
6. `01` is the length of the next string, which would be the value
7. `00` is the number of free bytes
8. We read the next 1 byte(s), which is `0x32`. Thus, we get our value "2"
9. In this case, the free bytes is 0, so we don't skip anything
10. `05` is the length of the next string, in this case a key.
11. We read the next 5 bytes `59 4e 4e 58 4b`, to get the key "YNNXK"
12. `04` is the length of the next string, which is a value
13. `00` is the number of free bytes after the value
14. We read the next 4 bytes i.e. `46 37 54 49` to get the value "F7TI"
15. Finally, we encounter `FF`, which indicates the end of this zip map
16. Thus, this zip map represents the hash `{"MKD1G6" => "2", "YNNXK" => "F7TI"}`

Ziplist Encoding

A Ziplist is a list that has been serialized to a string. In essence, the elements of the list are stored sequentially along with flags and offsets to allow efficient traversal of the list in both directions.

To parse a ziplist, first a string is read from the stream using "String Encoding". This string is the envelope of the ziplist. The contents of this string represent the ziplist.

The structure of a ziplist within this string is as follows -

```
<zlbytes><zltail><zllen><entry><entry><zlend>
```

1. *zlbytes* : This is a 4 byte unsigned integer representing the total size in bytes of the zip list. The 4 bytes are in little endian format - the least signinficant bit comes first.
2. *zltail* : This is a 4 byte unsigned integer in little endian format. It represents the offset to the tail (i.e. last) entry in the zip list
3. *zllen* : This is a 2 byte unsigned integer in little endian format. It represents the number of entries in this zip list
4. *entry* : An entry represents an element in the zip list. Details below
5. *zlend* : Is always equal to `255`. It represents the end of the zip list.

Each entry in the zip list has the following format :

```
<length-prev-entry><special-flag><raw-bytes-of-entry>
```

length-prev-entry : This field stores the length of the previous entry, or 0 if this is the first entry. This allows easy traversal of the list in the reverse direction. This length is stored in either 1 byte or in 5 bytes. If the first byte is less than or equal to 253, it is considered as the length. If the first byte is 254, then the next 4 bytes are used to store the length. The 4 bytes are read as an unsigned integer.

Special flag : This flag indicates whether the entry is a string or an integer. It also indicates the length of the string, or the size of the integer. The various encodings of this flag are shown below :

1. |00pppppp| - 1 byte : String value with length less than or equal to 63 bytes (6 bits).
2. |01pppppp|qqqqqqqq| - 2 bytes : String value with length less than or equal to 16383 bytes (14 bits).
3. |10_____|qqqqqqqq|rrrrrrrr|ssssssss|tttttttt| - 5 bytes : String value with length greater than or equal to 16384 bytes.

4. |1100____| - Read next 2 bytes as a 16 bit signed integer
5. |1101____| - Read next 4 bytes as a 32 bit signed integer
6. |1110____| - Read next 8 bytes as a 64 bit signed integer
7. |11110000| - Read next 3 bytes as a 24 bit signed integer
8. |11111110| - Read next byte as an 8 bit signed integer
9. |1111xxxx| - (with xxxx between 0000 and 1101) immediate 4 bit integer. Unsigned integer from 0 to 12. The encoded value is actually from 1 to 13 because 0000 and 1111 can not be used, so 1 should be subtracted from the encoded 4 bit value to obtain the right value.

Raw Bytes : After the special flag, the raw bytes of entry follow. The number of bytes was previously determined as part of the special flag.

Worked Example 1

```
23 23 00 00 00 1e 00 00 00 04 00 00 e0 ff ff ff ff ff ff ff 7f 0a d0 ff ff
00 00 06 c0 fc 3f 04 c0 3f 00 ff ...
```

1. Start by decoding this using "String Encoding". `23` is the length of the string, therefore we will read the next 35 bytes till `ff`
2. Now, we are parsing the string starting at `23 00 00 ...` using "Ziplist encoding"
3. The first 4 bytes `23 00 00 00` represent the total length in bytes of this ziplist. Notice that this is in little endian format
4. The next 4 bytes `1e 00 00 00` represent the offset to the tail entry. `1e` = 30, and this is a 0 based offset. 0th position = `23`, 1st position = `00` and so on. It follows that the last entry starts at `04 c0 3f 00 ..`
5. The next 2 bytes `04 00` represent the number of entries in this list.
6. From now on, we start reading the entries
7. `00` represents the length of previous entry. 0 indicates this is the first entry.
8. `e0` is the special flag. Since it starts with the bit pattern `1110____`, we read the next 8 bytes as an integer. This is the first entry of the list.
9. We now start the second entry
10. `0a` is the length of the previous entry. 10 bytes = 1 byte for prev. length + 1 byte for special flag + 8 bytes for integer.
11. `d0` is the special flag. Since it starts with the bit pattern `1101____`, we read the next 4 bytes as an integer. This is the second entry of the list
12. We now start the third entry
13. `06` is the length of previous entry. 6 bytes = 1 byte for prev. length + 1 byte for special flag + 4 bytes for integer
14. `c0` is the special flag. Since it starts with the bit pattern `1100____`, we read the next 2 bytes as an integer. This is the third entry of the list
15. We now start the last entry

16. `04` is length of previous entry
17. `c0` indicates a 2 byte number
18. We read the next 2 bytes, which gives us our fourth entry
19. Finally, we encounter `ff`, which tells us we have consumed all elements in this ziplist.
20. Thus, this ziplist stores the values `[0x7fffffffffffffff, 65535, 16380, 63]

Intset Encoding

An Intset is a binary search tree of integers. The binary tree is implemented in an array of integers. An intset is used when all the elements of the set are integers. An Intset has support for upto 64 bit integers. As an optimization, if the integers can be represented in fewer bytes, the array of integers will be constructed from 16 bit or 32 bit integers. When a new element is inserted, the implementation takes care to upgrade if necessary.

Since an Intset is a binary search tree, the numbers in this set will always be sorted.

An Intset has an external interface of a Set.

To parse an Intset, first a string is read from the stream using "String Encoding". This string is the envelope of the Intset. The contents of this string represent the Intset.

Within this string, the Intset has a very simple layout :

```
<encoding><length-of-contents><contents>
```

1. *encoding* : is a 32 bit unsigned integer. It has 3 possible values - 2, 4 or 8. It indicates the size in bytes of each integer stored in contents. And yes, this is wasteful - we could have stored the same information in 2 bits.
2. *length-of-contents* : is a 32 bit unsigned integer, and indicates the length of the contents array
3. *contents* : is an array of $length-of-contents bytes. It contains the binary tree of integers

Example

```
14 04 00 00 00 03 00 00 00 fc ff 00 00 fd ff 00 00 fe ff 00 00 ...
```

1. Start by decoding this using "String Encoding". `14` is the length of the string, therefore we will read the next 20 bytes till `00`
2. Now, we start interpreting the string starting at `04 00 00 ...`

3. The first 4 bytes `04 00 00 00` is the encoding. Since this evaluates to 4, we know we are dealing with 32 bit integers
4. The next 4 bytes `03 00 00 00` is the length of contents. So, we know we are dealing with 3 integers, each 4 byte long
5. From now on, we read in groups of 4 bytes, and convert it into a unsigned integer
6. Thus, our intset looks like - `0x0000FFFC, 0x0000FFFD, 0x0000FFFE`. Notice that the integers are in little endian format i.e. least significant bit came first.

Sorted Set as Ziplist Encoding

A sorted list in ziplist encoding is stored just like the Ziplist described above. Each element in the sorted set is followed by its score in the ziplist.

Example

['Manchester City', 1, 'Manchester United', 2, 'Tottenham', 3]

As you see, the scores follow each element.

Hashmap in Ziplist Encoding

In this, key=value pairs of a hashmap are stored as successive entries in a ziplist.

Note : This was introduced in rdb version 4. This deprecates zipmap encoding that was used in earlier versions.

Example

{"us" => "washington", "india" => "delhi"}

is stored in a ziplist as :

["us", "washington", "india", "delhi"]

CRC64 Check Sum

Starting with RDB Version 5, an 8 byte CRC 64 checksum is added to the end of the file. It is possible to disable this checksum via a parameter in redis.conf. When checksum is disabled, this field will have zeroes.