19 从零实现 KV 存储—Redis 数据结构设计

理论讲解

本节无代码部分

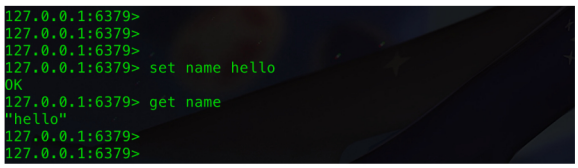
前面我们实现的存储引擎，只实现了几个简单的 KV 接口，例如 `Put(K, V)，Get(K) -> V，Delete(K)`，这几个接口只支持 Key/Value 的数据格式，在更加多样化的实际使用需求中，纯粹的 KV 接口能够满足的需求比较有限。

我们可以参考 NoSQL 数据库的行业标准 Redis，Redis 是一个成熟强大的 KV 数据库，它支持的数据类型非常的多样化，例如字符串（String）、哈希表（Hash）、列表（List）、集合（Set）、有序集合（sorted set）、位图（bitmap）等等，能够满足很多的需求。

因此我们可以在这基础之上，将我们实现的 KV 存储引擎加上更加多样的数据类型，这一节将会依次介绍 Redis 数据结构设计的大致思路，包含 Redis 最常用的五种数据结构，String、Hash、Set、List、ZSet（Sorted Set）。

在设计上，我们遵循的一个总体理念就是在 KV 的接口之上，去实现 Redis 的这几种数据结构，主要是将这几种结构进行转化和编码，然后使用我们 bitcask 存储引擎的 KV 接口来进行存储。

String




```
1         +----------+------------+--------------------+
2 key =>  |   type   |  expire    |       payload      |
3         | (1byte)  | (Ebyte)    |       (Nbyte)      |
4         +----------+------------+--------------------+
```

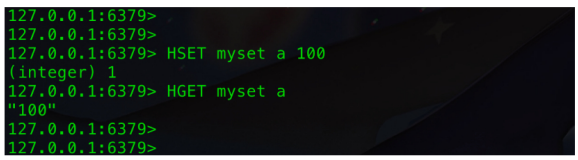
type：数据类型

```
1 0-String
2 1-Hash
3 2-Set
4 3-List
5 4-ZSet
```

expire：过期时间，unix 时间戳

payload：原始 value 部分

Hash



元数据

```
1         +----------+------------+-----------+-----------+
2 key =>  |   type   |  expire    |  version  |  size     |
3         | (1byte)  | (Ebyte)    |  (8byte)  | (Sbyte)   |
4         +----------+------------+-----------+-----------+
```

type：数据类型

expire：过期时间，unix 时间戳

version：版本，用于快速删除

size：此 key 下有多少数据

version 有什么作用?

Version 的作用主要是用于快速删除一个 key，假设我们有一个 hash 类型的 key，例如叫 `myhash_key`，这个 key 下面有这样几条数据：

```
1 key          field   value
2 myhash_key    f1     val1
3 myhash_key    f2     val2
4 myhash_key    f3     val3
```

假如我们删除这个 key `myhash_key`，那么这个 key 下的所有数据都是不可访问的了，在没有任何其他保证的情况下，我们需要遍历这个 key 下面的所有数据，依次执行删除。

但如果一个 key 下面有非常多的数据，我们一条一条的去删除，容易造成性能瓶颈，所以我们可以维护一个元数据 version，记录数据的版本，这个 key 下面的每一条数据在存储的时候，都记录了这个 version。

version 是递增的，假如 key 被删除之后，我们又重新添加了这个 key，这时候会分配一个新的 version，和之前的 version 不一样，所以我们就查不到之前的旧的数据。

数据部分

```
                    +--------------+
key|version|field => |     value    |
                    +--------------+
```

Set

```
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> SADD mys a b c
(integer) 3
127.0.0.1:6379> SMEMBERS mys
1) "c"
2) "b"
3) "a"
127.0.0.1:6379>
127.0.0.1:6379>
```

元数据

```
        +----------+------------+-----------+-----------+
key =>  |   type   |  expire    |  version  |  size     |
        | (1byte)  | (8byte)    |  (8byte)  | (8byte)   |
        +----------+------------+-----------+-----------+
```

和 Hash 完全一致。

数据部分

```
                                +--------------+
key|version|member|member size => |     NULL     |
                                +--------------+
```

List

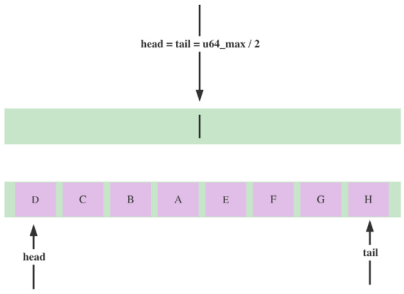
```
127.0.0.1:6379>
127.0.0.1:6379> LPUSH mylist a b c
(integer) 3
127.0.0.1:6379> RPUSH mylist e d f
(integer) 6
127.0.0.1:6379> LPOP mylist
"c"
127.0.0.1:6379> RPOP mylist
"f"
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379>
```

元数据

```
        +----------+------------+-----------+-----------+-----------+-----------+
key =>  |   type   |  expire    |  version  |  size     |  head     |  tail     |
        | (1byte)  | (8byte)    |  (8byte)  | (8byte)   | (8byte)   | (8byte)   |
        +----------+------------+-----------+-----------+-----------+-----------+
```

List 结构的元数据部分和 Hash、Set 比较类似，只是多了两个字段 head 和 tail。

List 可以看做是一个队列，可以在队列的头尾进行 Push、Pop 操作，因此我们可以使用一个标识来表示头尾，在初始情况下，`head = tail = U64_MAX / 2`。



数据部分

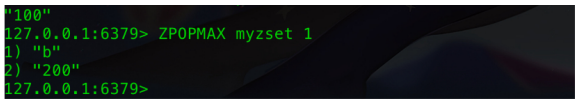
```
                    +--------------+
key|version|index => |     value    |
                    +--------------+
```

Index 就是 head 或者 tail 的值，当在左边 Push 的时候，index 的值是 `head - 1`，当在右边 push 的时候，index 的值就是 `tail`。

ZSet

ZSet 是有序集合 sorted set。

```
127.0.0.1:6379>
127.0.0.1:6379> ZADD myzset 100 a
(integer) 1
127.0.0.1:6379> ZADD myzset 200 b
(integer) 1
127.0.0.1:6379> ZSCORE myzset a
```



元数据

```
        +----------+------------+-----------+-----------+
key ->  |  flags   |  expire    |  version  |  size     |
        | (1byte)  | (Ebyte)    |  (8byte)  | (Sbyte)   |
        +----------+------------+-----------+-----------+
```

和 Hash、Set 一致。

数据部分

```
                          +--------------+
key|version|member =>  |     score      |   (1)
                          +--------------+


                          +--------------+
key|version|score  =>  |     member     |   (2)
                          +--------------+
```

分为两部分存储，第一部分主要是通过 key + member 获取到对应的 score，第二部分将 member 按照 score 分值进行排列，主要是为了能够按照 score 的顺序获取 member。