13. The Heap Data Structure and the TTL

13.1 Heap Review

The primary use of Redis is as cache servers, and one way to manage the size of the cache is through explicitly setting TTLs (time to live). TTLs can be implemented using timers. Unfortunately, timers in the last chapter are of fixed value (using linked lists); thus, a sorting data structure is needed for implementing arbitrary and mutable timeouts; and the heap data structure is a popular choice. Compared with the AVL tree we used before, the heap data structure has the advantage of using less space.

A quick review of the heap data structure:

1. A heap is a binary tree, packed into an array; and the layout of the tree is fixed. The parent-child relationship is implicit, pointers are not included in heap elements.
2. The only constraint on the tree is that parents are no bigger than their kids.
3. The value of an element can be updated. If the value changes:
   - Its value is bigger than before: it may be bigger than its kids, and if so, swap it with the smallest kid, so that the parent-child constraint is satisfied again. Now that one of the kids is bigger than before, continue this process until reaching a leave.
   - Its value is smaller: likewise, swap it with its parent until reaching the root.
4. New elements are added to the end of the array as leaves. Maintain the constraint as above.
5. When removing an element from a heap, replace it with the last element in the array, then maintain the constraint as if its value was updated.

13.2 The Heap Definition

The code listing begins:

```
struct HeapItem {
    uint64_t val = 0;
    size_t *ref = NULL;
};

// the structure for the key
struct Entry {
    struct HNode node;
    std::string key;
    std::string val;
    uint32_t type = 0;
    ZSet *zset = NULL;
    // for TTLs
    size_t heap_idx = -1;
};
```

The heap is used to order the timestamps, and the `Entry` is mutually linked with the timestamp. The `heap_idx` is the index of the corresponding `HeapItem`, and the `ref` points to the `Entry`. We are using the intrusive data structure again; the `ref` pointer points to the `heap_idx` field.

The parent-child relationship is fixed:

```
static size_t heap_parent(size_t i) {
    return (i + 1) / 2 - 1;
}

static size_t heap_left(size_t i) {
    return i * 2 + 1;
}

static size_t heap_right(size_t i) {
    return i * 2 + 2;
}
```

13.3 Heap Operations

Swap with the parent when a kid is smaller than its parent. Note the `heap_idx` is updated through the `ref` pointer while swapping.

```c
static void heap_up(HeapItem *a, size_t pos) {
    HeapItem t = a[pos];
    while (pos > 0 && a[heap_parent(pos)].val > t.val) {
        // swap with the parent
        a[pos] = a[heap_parent(pos)];
        *a[pos].ref = pos;
        pos = heap_parent(pos);
    }
    a[pos] = t;
    *a[pos].ref = pos;
}
```

Swapping with the smallest kid is similar.

```c
static void heap_down(HeapItem *a, size_t pos, size_t len) {
    HeapItem t = a[pos];
    while (true) {
        // find the smallest one among the parent and their kids
        size_t l = heap_left(pos);
        size_t r = heap_right(pos);
        size_t min_pos = -1;
        size_t min_val = t.val;
        if (l < len && a[l].val < min_val) {
            min_pos = l;
            min_val = a[l].val;
        }
        if (r < len && a[r].val < min_val) {
            min_pos = r;
        }
        if (min_pos == (size_t)-1) {
            break;
        }
        // swap with the kid
        a[pos] = a[min_pos];
        *a[pos].ref = pos;
        pos = min_pos;
    }
    a[pos] = t;
    *a[pos].ref = pos;
}
```

The heap_update is the heap function for updating a position. It is used for updating, inserting, and deleting.

```
void heap_update(HeapItem *a, size_t pos, size_t len) {
    if (pos > 0 && a[heap_parent(pos)].val > a[pos].val) {
        heap_up(a, pos);
    } else {
        heap_down(a, pos, len);
    }
}
```

13.4 New Timers

Add the heap to our server:

```
// global variables
static struct {
    HMap db;
    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;
    // timers for idle connections
    DList idle_list;
    // timers for TTLs
    std::vector<HeapItem> heap;
} g_data;
```

13.4.1 Maintain TTL Timers

Updating, adding, and removing a timer to the heap. Just call the heap_update after updating an element of the array.

```
// set or remove the TTL
static void entry_set_ttl(Entry *ent, int64_t ttl_ms) {
    if (ttl_ms < 0 && ent->heap_idx != (size_t)-1) {
        // erase an item from the heap
        // by replacing it with the last item in the array.
        size_t pos = ent->heap_idx;
        g_data.heap[pos] = g_data.heap.back();
        g_data.heap.pop_back();
        if (pos < g_data.heap.size()) {
            heap_update(g_data.heap.data(), pos, g_data.heap.size());
        }
        ent->heap_idx = -1;
    } else if (ttl_ms >= 0) {
        size_t pos = ent->heap_idx;
        if (pos == (size_t)-1) {
            // add an new item to the heap
            HeapItem item;
            item.ref = &ent->heap_idx;
            g_data.heap.push_back(item);
            pos = g_data.heap.size() - 1;
        }
        g_data.heap[pos].val = get_monotonic_usec() + (uint64_t)ttl_ms *
1000;
        heap_update(g_data.heap.data(), pos, g_data.heap.size());
    }
}
```

Removing the possible TTL timer when deleting an `Entry`:

```
static void entry_del(Entry *ent) {
    switch (ent->type) {
    case T_ZSET:
        zset_dispose(ent->zset);
        delete ent->zset;
        break;
    }
    entry_set_ttl(ent, -1);
    delete ent;
}
```

13.4.2 Find the Nearest Timer

The `next_timer_ms` function is modified to use both idle timers and TTL timers.

```
static uint32_t next_timer_ms() {
    uint64_t now_us = get_monotonic_usec();
    uint64_t next_us = (uint64_t)-1;

    // idle timers
    if (!dlist_empty(&g_data.idle_list)) {
        Conn *next = container_of(g_data.idle_list.next, Conn, idle_list);
        next_us = next->idle_start + k_idle_timeout_ms * 1000;
    }

    // ttl timers
    if (!g_data.heap.empty() && g_data.heap[0].val < next_us) {
        next_us = g_data.heap[0].val;
    }

    if (next_us == (uint64_t)-1) {
        return 10000;   // no timer, the value doesn't matter
    }

    if (next_us <= now_us) {
        // missed?
        return 0;
    }
    return (uint32_t)((next_us - now_us) / 1000);
}
```

13.4.3 Fire Timers

Adding TTL timers to the `process_timers` function:

```
static void process_timers() {
    // the extra 1000us is for the ms resolution of poll()
    uint64_t now_us = get_monotonic_usec() + 1000;

    // idle timers
    while (!dlist_empty(&g_data.idle_list)) {
        // code omitted...
    }

    // TTL timers
    const size_t k_max_works = 2000;
    size_t nworks = 0;
    while (!g_data.heap.empty() && g_data.heap[0].val < now_us) {
        Entry *ent = container_of(g_data.heap[0].ref, Entry, heap_idx);
        HNode *node = hm_pop(&g_data.db, &ent->node, &hnode_same);
        assert(node == &ent->node);
        entry_del(ent);
```

```
            if (nworks++ >= k_max_works) {
                // don't stall the server if too many keys are expiring at once
                break;
            }
        }
    }
```

This is just checking the minimal value of the heap and removing keys. Note that we put a limit on the number of keys expired per event loop iteration; the limit is needed to prevent the server from stalling should there are too many keys expiring at once.

13.5 New Commands

The command for updating and querying TTLs is straightforward to add:

```
static void do_expire(std::vector<std::string> &cmd, std::string &out) {
    int64_t ttl_ms = 0;
    if (!str2int(cmd[2], ttl_ms)) {
        return out_err(out, ERR_ARG, "expect int64");
    }

    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (node) {
        Entry *ent = container_of(node, Entry, node);
        entry_set_ttl(ent, ttl_ms);
    }
    return out_int(out, node ? 1: 0);
}


static void do_ttl(std::vector<std::string> &cmd, std::string &out) {
    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (!node) {
        return out_int(out, -2);
    }

    Entry *ent = container_of(node, Entry, node);
    if (ent->heap_idx == (size_t)-1) {
        return out_int(out, -1);
    }

    uint64_t expire_at = g_data.heap[ent->heap_idx].val;
    uint64_t now_us = get_monotonic_usec();
    return out_int(out, expire_at > now_us ? (expire_at - now_us) / 1000 :
        0);
}
```

Exercises:

1. The heap-based timer adds `O(log(n))` operations to the server, which might be a bottleneck for a sufficiently large number of keys. Can you think of optimizations for a large number of timers?
2. The real Redis does not use sorting for expiration, find out how it is done, and list the pros and cons of both approaches.

Source code:

- 13_server.cpp
- avl.cpp
- avl.h
- common.h
- hashtable.cpp
- hashtable.h
- heap.cpp
- heap.h
- list.h
- test_heap.cpp
- zset.cpp
- zset.h

See also:

codecrafters.io offers “Build Your Own X” courses in many programming languages. Including Redis, Git, SQLite, Docker, and more.

Check it out

← prev Contents next →

Subscribe Twitter About
Free Books Blogs Home