09. Data Serialization

For now, our server protocol response is an error code plus a string. What if we need to return more complicated data? For example, we could add the `keys` command, which returns a list of strings. We have already encoded the list-of-strings data in the request protocol. In this chapter, we will generalize the encoding to handle different types of data. This is often called "serialization".

9.1 The Command Interface

Our serialization protocol consists of five data types:

```
enum {
    SER_NIL = 0,    // Like `NULL`
    SER_ERR = 1,    // An error code and message
    SER_STR = 2,    // A string
    SER_INT = 3,    // A int64
    SER_ARR = 4,    // Array
};
```

The array type can contain any type of data, even nested arrays.

The code listing starts with the `try_one_request` function:

```
static bool try_one_request(Conn *conn) {
    // code omitted...

    // parse the request
    std::vector<std::string> cmd;
    if (0 != parse_req(&conn->rbuf[4], len, cmd)) {
```

```
            msg("bad req");
            conn->state = STATE_END;
            return false;
        }

        // got one request, generate the response.
        std::string out;
        do_request(cmd, out);

        // pack the response into the buffer
        if (4 + out.size() > k_max_msg) {
            out.clear();
            out_err(out, ERR_2BIG, "response is too big");
        }
        uint32_t wlen = (uint32_t)out.size();
        memcpy(&conn->wbuf[0], &wlen, 4);
        memcpy(&conn->wbuf[4], out.data(), out.size());
        conn->wbuf_size = 4 + wlen;

        // code omitted...
    }
```

For convenience, `std::string` is used to hold the response data. Production-grade projects often have more sophisticated ways to manage buffers.

The new command `keys` is added to the `do_request` handler:

```
    static void do_request(std::vector<std::string> &cmd, std::string &out) {
        if (cmd.size() == 1 && cmd_is(cmd[0], "keys")) {
            do_keys(cmd, out);
        } else if (cmd.size() == 2 && cmd_is(cmd[0], "get")) {
            do_get(cmd, out);
        } else if (cmd.size() == 3 && cmd_is(cmd[0], "set")) {
            do_set(cmd, out);
        } else if (cmd.size() == 2 && cmd_is(cmd[0], "del")) {
            do_del(cmd, out);
        } else {
            // cmd is not recognized
            out_err(out, ERR_UNKNOWN, "Unknown cmd");
        }
    }
```

9.2 Data Encoding Scheme

The code for our serialization protocol:

```
static void out_nil(std::string &out) {
    out.push_back(SER_NIL);
}

static void out_str(std::string &out, const std::string &val) {
    out.push_back(SER_STR);
    uint32_t len = (uint32_t)val.size();
    out.append((char *)&len, 4);
    out.append(val);
}

static void out_int(std::string &out, int64_t val) {
    out.push_back(SER_INT);
    out.append((char *)&val, 8);
}

static void out_err(std::string &out, int32_t code, const std::string &msg)
        {
    out.push_back(SER_ERR);
    out.append((char *)&code, 4);
    uint32_t len = (uint32_t)msg.size();
    out.append((char *)&len, 4);
    out.append(msg);
}

static void out_arr(std::string &out, uint32_t n) {
    out.push_back(SER_ARR);
    out.append((char *)&n, 4);
}
```

As we can see, our serialization protocol starts with a byte of data type, followed by various types of payload data. Arrays come first with their size, then their possibly nested elements.

The serialization scheme can be summarized as “type-length-value” (TLV): “Type” indicates the type of the value; “Length” is for variable length data such as strings or arrays; “Value” is the encoded at last.

TLV is the basis of many real-world serialization protocols. It has many advantages:

1. It can be decoded without a schema, like JSON or XML, which enables some types of mddleware.
2. It can encode arbitrarily nested data.

The Thrift RPC framework includes 2 serialization schemes, both derived from the TLV scheme. You can learn more by reading the specification and comparing it to the popular Protobuf scheme.

9.3 Command Responses

The `do_keys` function generates a response consisting of a list of strings:

```
static void h_scan(HTab *tab, void (*f)(HNode *, void *), void *arg) {
    if (tab->size == 0) {
        return;
    }
    for (size_t i = 0; i < tab->mask + 1; ++i) {
        HNode *node = tab->tab[i];
        while (node) {
            f(node, arg);
            node = node->next;
        }
    }
}

static void cb_scan(HNode *node, void *arg) {
    std::string &out = *(std::string *)arg;
    out_str(out, container_of(node, Entry, node)->key);
}

static void do_keys(std::vector<std::string> &cmd, std::string &out) {
    (void)cmd;
    out_arr(out, (uint32_t)hm_size(&g_data.db));
    h_scan(&g_data.db.ht1, &cb_scan, &out);
    h_scan(&g_data.db.ht2, &cb_scan, &out);
}
```

The `del` command responds with an integer indicating whether the deletion took place.

```
static void do_del(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_pop(&g_data.db, &key.node, &entry_eq);
    if (node) {
        delete container_of(node, Entry, node);
    }
    return out_int(out, node ? 1 : 0);
}
```

The code for the other commands is of nothing interesting, so we'll skip them.

9.4 The Client and Testing

Listing the client "deserialization" code:

```
static int32_t on_response(const uint8_t *data, size_t size) {
    if (size < 1) {
        msg("bad response");
        return -1;
    }
    switch (data[0]) {
    case SER_NIL:
        printf("(nil)\n");
        return 1;
    case SER_ERR:
        if (size < 1 + 8) {
            msg("bad response");
            return -1;
        }
        {
            int32_t code = 0;
            uint32_t len = 0;
            memcpy(&code, &data[1], 4);
            memcpy(&len, &data[1 + 4], 4);
            if (size < 1 + 8 + len) {
                msg("bad response");
                return -1;
            }
            printf("(err) %d %.*s\n", code, len, &data[1 + 8]);
            return 1 + 8 + len;
        }
```

```
    case SER_STR:
        // code omited...
    case SER_INT:
        // code omited...
    case SER_ARR:
        if (size < 1 + 4) {
            msg("bad response");
            return -1;
        }
        {
            uint32_t len = 0;
            memcpy(&len, &data[1], 4);
            printf("(arr) len=%u\n", len);
            size_t arr_bytes = 1 + 4;
            for (uint32_t i = 0; i < len; ++i) {
                int32_t rv = on_response(&data[arr_bytes], size -
        arr_bytes);
                if (rv < 0) {
                    return rv;
                }
                arr_bytes += (size_t)rv;
            }
            printf("(arr) end\n");
            return (int32_t)arr_bytes;
        }
    default:
        msg("bad response");
        return -1;
    }
}
```

Testing our new server/client:

```
$ ./client asdf
(err) 1 Unknown cmd
$ ./client get asdf
(nil)
$ ./client set k v
(nil)
$ ./client get k
(str) v
$ ./client keys
```

```
(arr) len=1
(str) k
(arr) end
$ ./client del k
(int) 1
$ ./client del k
(int) 0
$ ./client keys
(arr) len=0
(arr) end
```

Source code:

- 09_client.cpp
- 09_server.cpp
- hashtable.cpp
- hashtable.h