Lua 脚本

Lua 脚本功能是 Reids 2.6 版本的最大亮点， 通过内嵌对 Lua 环境的支持， Redis 解决了长久以来不能高效地处理 CAS （check-and-set）命令的缺点， 并且可以通过组合使用多个命令， 轻松实现以前很难实现或者不能高效实现的模式。

本章先介绍 Lua 环境的初始化步骤， 然后对 Lua 脚本的安全性问题、以及解决这些问题的方法进行说明， 最后对执行 Lua 脚本的两个命令 —— EVAL 和 EVALSHA 的实现原理进行介绍。

初始化 Lua 环境

在初始化 Redis 服务器时， 对 Lua 环境的初始化也会一并进行。

为了让 Lua 环境符合 Redis 脚本功能的需求， Redis 对 Lua 环境进行了一系列的修改， 包括添加函数库、更换随机函数、保护全局变量， 等等。

整个初始化 Lua 环境的步骤如下：

1. 调用 lua_open 函数，创建一个新的 Lua 环境。
2. 载入指定的 Lua 函数库，包括：
    - 基础库（base lib）。
    - 表格库（table lib）。
    - 字符串库（string lib）。
    - 数学库（math lib）。
    - 调试库（debug lib）。
    - 用于处理 JSON 对象的 cjson 库。
    - 在 Lua 值和 C 结构（struct）之间进行转换的 struct 库（http://www.inf.puc-rio.br/~roberto/struct/）。
    - 处理 MessagePack 数据的 cmsgpack 库（https://github.com/antirez/lua-cmsgpack）。
3. 屏蔽一些可能对 Lua 环境产生安全问题的函数，比如 loadfile 。
4. 创建一个 Redis 字典，保存 Lua 脚本，并在复制（replication）脚本时使用。字典的键为 SHA1 校验和，字典的值为 Lua 脚本。
5. 创建一个 redis 全局表格到 Lua 环境，表格中包含了各种对 Redis 进行操作的函数，包括：
    - 用于执行 Redis 命令的 redis.call 和 redis.pcall 函数。
    - 用于发送日志（log）的 redis.log 函数，以及相应的日志级别（level）：
        - redis.LOG_DEBUG
        - redis.LOG_VERBOSE
        - redis.LOG_NOTICE
        - redis.LOG_WARNING
    - 用于计算 SHA1 校验和的 redis.shalhex 函数。
    - 用于返回错误信息的 redis.error_reply 函数和 redis.status_reply 函数。
6. 用 Redis 自己定义的随机生成函数，替换 math 表原有的 math.random 函数和 math.randomseed 函数，新的函数具有这样的性质：每次执行 Lua 脚本时，除非显式地调用 math.randomseed ，否则 math.random 生成的伪随机数序列总是相同的。
7. 创建一个对 Redis 多批量回复（multi bulk reply）进行排序的辅助函数。
8. 对 Lua 环境中的全局变量进行保护，以免被传入的脚本修改。
9. 因为 Redis 命令必须通过客户端来执行，所以需要在服务器状态中创建一个无网络连接的伪客户端（fake client），专门用于执行 Lua 脚本中包含的 Redis 命令：当 Lua 脚本需要执行 Redis 命令时，它通过伪客户端来向服务器发送命令请求，服务器在执行完命令

之后，将结果返回给伪客户端，而伪客户端又转而将命令结果返回给 Lua 脚本。
10. 将 Lua 环境的指针记录到 Redis 服务器的全局状态中，等候 Redis 的调用。

以上就是 Redis 初始化 Lua 环境的整个过程， 当这些步骤都执行完之后， Redis 就可以使用 Lua 环境来处理脚本了。

严格来说， 步骤 1 至 8 才是初始化 Lua 环境的操作， 而步骤 9 和 10 则是将 Lua 环境关联到服务器的操作， 为了按顺序观察整个初始化过程， 我们将两种操作放在了一起。

另外， 步骤 6 用于创建无副作用的脚本， 而步骤 7 则用于去除部分 Redis 命令中的不确定性（non deterministic）， 关于这两点， 请看下面一节关于脚本安全性的讨论。

脚本的安全性

当将 Lua 脚本复制到附属节点， 或者将 Lua 脚本写入 AOF 文件时， Redis 需要解决这样一个问题： 如果一段 Lua 脚本带有随机性质或副作用， 那么当这段脚本在附属节点运行时， 或者从 AOF 文件载入重新运行时， 它得到的结果可能和之前运行的结果完全不同。

考虑以下一段代码， 其中的 get_random_number() 带有随机性质， 我们在服务器 SERVER 中执行这段代码， 并将随机数的结果保存到键 number 上：

```
# 虚构例子，不会真的出现在脚本环境中

redis> EVAL "return redis.call('set', KEYS[1], get_random_number())" 1 number
OK

redis> GET number
"10086"
```

现在， 假如 EVAL 的代码被复制到了附属节点 SLAVE ， 因为 get_random_number() 的随机性质， 它有很大可能会生成一个和 10086 完全不同的值， 比如 65535 ：

```
# 虚构例子，不会真的出现在脚本环境中

redis> EVAL "return redis.call('set', KEYS[1], get_random_number())" 1 number
OK

redis> GET number
"65535"
```

可以看到， 带有随机性的写入脚本产生了一个严重的问题： 它破坏了服务器和附属节点数据之间的一致性。

当从 AOF 文件中载入带有随机性质的写入脚本时， 也会发生同样的问题。

只有在带有随机性的脚本进行写入时， 随机性才是有害的。

如果一个脚本只是执行只读操作， 那么随机性是无害的。

比如说， 如果脚本只是单纯地执行 RANDOMKEY 命令， 那么它是无害的； 但如果在执行 RANDOMKEY 之后， 基于 RANDOMKEY 的结果进行写入操作， 那么这个脚本就是有

害的。

和随机性质类似， 如果一个脚本的执行对任何副作用产生了依赖， 那么这个脚本每次执行所产生的结果都可能会不一样。

为了解决这个问题， Redis 对 Lua 环境所能执行的脚本做了一个严格的限制 —— 所有脚本都必须是无副作用的纯函数（pure function）。

为此，Redis 对 Lua 环境做了一些列相应的措施：

- 不提供访问系统状态状态的库（比如系统时间库）。
- 禁止使用 loadfile 函数。
- 如果脚本在执行带有随机性质的命令（比如 RANDOMKEY ），或者带有副作用的命令（比如 TIME ）之后，试图执行一个写入命令（比如 SET ），那么 Redis 将阻止这个脚本继续运行，并返回一个错误。
- 如果脚本执行了带有随机性质的读命令（比如 SMEMBERS ），那么在脚本的输出返回给 Redis 之前，会先被执行一个自动的字典序排序，从而确保输出结果是有序的。
- 用 Redis 自己定义的随机生成函数，替换 Lua 环境中 math 表原有的 math.random 函数和 math.randomseed 函数，新的函数具有这样的性质：每次执行 Lua 脚本时，除非显式地调用 math.randomseed ，否则 math.random 生成的伪随机数序列总是相同的。

经过这一系列的调整之后， Redis 可以保证被执行的脚本：

1. 无副作用。
2. 没有有害的随机性。
3. 对于同样的输入参数和数据集，总是产生相同的写入命令。

脚本的执行

在脚本环境的初始化工作完成以后， Redis 就可以通过 EVAL 命令或 EVALSHA 命令执行 Lua 脚本了。

其中， EVAL 直接对输入的脚本代码体（body）进行求值：

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

而 EVALSHA 则要求输入某个脚本的 SHA1 校验和， 这个校验和所对应的脚本必须至少被 EVAL 执行过一次：

```
redis> EVAL "return 'hello world'" 0
"hello world"

redis> EVALSHA 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 0     // 上一个脚本的校验和
"hello world"
```

或者曾经使用 SCRIPT LOAD 载入过这个脚本：

```
redis> SCRIPT LOAD "return 'dlrow olleh'"
```

```
"d569c48906b1f4fca0469ba4eee89149b5148092"

redis> EVALSHA d569c48906b1f4fca0469ba4eee89149b5148092 0
"dlrow olleh"
```

因为 EVALSHA 是基于 EVAL 构建的， 所以下文先用一节讲解 EVAL 的实现， 之后再讲解 EVALSHA 的实现。

EVAL 命令的实现

EVAL 命令的执行可以分为以下步骤：

1. 为输入脚本定义一个 Lua 函数。
2. 执行这个 Lua 函数。

以下两个小节分别介绍这两个步骤。

定义 Lua 函数

所有被 Redis 执行的 Lua 脚本， 在 Lua 环境中都会有一个和该脚本相对应的无参数函数： 当调用 EVAL 命令执行脚本时， 程序第一步要完成的工作就是为传入的脚本创建一个相应的 Lua 函数。

举个例子， 当执行命令 EVAL "return 'hello world'" 0 时， Lua 会为脚本 "return 'hello world'" 创建以下函数：

```
function f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91()
    return 'hello world'
end
```

其中， 函数名以 f_ 为前缀， 后跟脚本的 SHA1 校验和（一个 40 个字符长的字符串）拼接而成。 而函数体（body）则是用户输入的脚本。

以函数为单位保存 Lua 脚本有以下好处：

- 执行脚本的步骤非常简单，只要调用和脚本相对应的函数即可。
- Lua 环境可以保持清洁，已有的脚本和新加入的脚本不会互相干扰，也可以将重置 Lua 环境和调用 Lua GC 的次数降到最低。
- 如果某个脚本所对应的函数在 Lua 环境中被定义过至少一次，那么只要记得这个脚本的 SHA1 校验和，就可以直接执行该脚本 —— 这是实现 EVALSHA 命令的基础，稍后在介绍 EVALSHA 的时候就会说到这一点。

在为脚本创建函数前，程序会先用函数名检查 Lua 环境，只有在函数定义未存在时，程序才创建函数。重复定义函数一般并没有什么副作用，这算是一个小优化。

另外，如果定义的函数在编译过程中出错（比如，脚本的代码语法有错）， 那么程序向用户返回一个脚本错误， 不再执行后面的步骤。

执行 Lua 函数

在定义好 Lua 函数之后， 程序就可以通过运行这个函数来达到运行输入脚本的目的了。

不过， 在此之前， 为了确保脚本的正确和安全执行， 还需要执行一些设置钩子、传入参数之类的操作， 整个执行函数的过程如下：

1. 将 EVAL 命令中输入的 KEYS 参数和 ARGV 参数以全局数组的方式传入到 Lua 环境中。
2. 设置伪客户端的目标数据库为调用者客户端的目标数据库： fake_client->db = caller_client->db ，确保脚本中执行的 Redis 命令访问的是正确的数据库。
3. 为 Lua 环境装载超时钩子，保证在脚本执行出现超时时可以杀死脚本，或者停止 Redis 服务器。
4. 执行脚本对应的 Lua 函数。
5. 如果被执行的 Lua 脚本中带有 SELECT 命令，那么在脚本执行完毕之后，伪客户端中的数据库可能已经有所改变，所以需要对调用者客户端的目标数据库进行更新： caller_client->db = fake_client->db 。
6. 执行清理操作：清除钩子；清除指向调用者客户端的指针；等等。
7. 将 Lua 函数执行所得的结果转换成 Redis 回复，然后传给调用者客户端。
8. 对 Lua 环境进行一次单步的渐进式 GC 。

以下是执行 EVAL "return 'hello world'" 0 的过程中， 调用者客户端（caller）、Redis 服务器和 Lua 环境之间的数据流表示图：

```
          发送命令请求
          EVAL "return 'hello world'" 0
Caller ----------------------------------------> Redis

          为脚本 "return 'hello world'"
          创建 Lua 函数
Redis  ----------------------------------------> Lua

          绑定超时处理钩子
Redis  ----------------------------------------> Lua

          执行脚本函数
Redis  ----------------------------------------> Lua

          返回函数执行结果（一个 Lua 值）
Redis  <---------------------------------------- Lua

          将 Lua 值转换为 Redis 回复
          并将结果返回给客户端
Caller <---------------------------------------- Redis
```

上面这个图可以作为所有 Lua 脚本的基本执行流程图， 不过它展示的 Lua 脚本中不带有 Redis 命令调用： 当 Lua 脚本里本身有调用 Redis 命令时（执行 redis.call 或者 redis.pcall ）， Redis 和 Lua 脚本之间的数据交互会更复杂一些。

举个例子， 以下是执行命令 EVAL "return redis.call('DBSIZE')" 0 时， 调用者客户端（caller）、伪客户端（fake client）、Redis 服务器和 Lua 环境之间的数据流表示图：

```
          发送命令请求
          EVAL "return redis.call('DBSIZE')" 0
Caller ----------------------------------------> Redis

          为脚本 "return redis.call('DBSIZE')"
          创建 Lua 函数
```

```
Redis  ----------------------------------------> Lua

          绑定超时处理钩子
Redis  ----------------------------------------> Lua

          执行脚本函数
Redis  ----------------------------------------> Lua

               执行 redis.call('DBSIZE')
Fake Client <----------------------------------- Lua

               伪客户端向服务器发送
               DBSIZE 命令请求
Fake Client -----------------------------------> Redis

               服务器将 DBSIZE 的结果
               （Redis 回复）返回给伪客户端
Fake Client <----------------------------------- Redis

               将命令回复转换为 Lua 值
               并返回给 Lua 环境
Fake Client -----------------------------------> Lua

          返回函数执行结果（一个 Lua 值）
Redis  <---------------------------------------- Lua

          将 Lua 值转换为 Redis 回复
          并将该回复返回给客户端
Caller <---------------------------------------- Redis
```

因为 EVAL "return redis.call('DBSIZE')" 只是简单地调用了一次 DBSIZE 命令， 所以 Lua 和伪客户端只进行了一趟交互， 当脚本中的 redis.call 或者 redis.pcall 次数增多时， Lua 和伪客户端的交互趟数也会相应地增多， 不过总体的交互方法和上图展示的一样。

EVALSHA 命令的实现

前面介绍 EVAL 命令的实现时说过， 每个被执行过的 Lua 脚本， 在 Lua 环境中都有一个和它相对应的函数， 函数的名字由 f_ 前缀加上 40 个字符长的 SHA1 校验和构成： 比如 f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91 。

只要脚本所对应的函数曾经在 Lua 里面定义过， 那么即使用户不知道脚本的内容本身， 也可以直接通过脚本的 SHA1 校验和来调用脚本所对应的函数， 从而达到执行脚本的目的 —— 这就是 EVALSHA 命令的实现原理。

可以用伪代码来描述这一原理：

```
def EVALSHA(sha1):

    # 拼接出 Lua 函数名字
    func_name = "f_" + sha1

    # 查看该函数是否已经在 Lua 中定义
    if function_defined_in_lua(func_name):
```

```
        # 如果已经定义过的话，执行函数
        return exec_lua_function(func_name)

    else:

        # 没有找到和输入 SHA1 值相对应的函数则返回一个脚本未找到错误
        return script_error("SCRIPT NOT FOUND")
```

除了执行 EVAL 命令之外， SCRIPT LOAD 命令也可以为脚本在 Lua 环境中创建函数：

```
redis> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"

redis> EVALSHA 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 0
"hello world"
```

SCRIPT LOAD 执行的操作和前面《定义 Lua 函数》小节描述的一样。

小结

- 初始化 Lua 脚本环境需要一系列步骤，其中最重要的包括：
  - 创建 Lua 环境。
  - 载入 Lua 库，比如字符串库、数学库、表格库，等等。
  - 创建 redis 全局表格，包含各种对 Redis 进行操作的函数，比如 redis.call 和 redis.log ， 等等。
  - 创建一个无网络连接的伪客户端，专门用于执行 Lua 脚本中的 Redis 命令。
- Reids 通过一系列措施保证被执行的 Lua 脚本无副作用，也没有有害的写随机性：对于同样的输入参数和数据集，总是产生相同的写入命令。
- EVAL 命令为输入脚本定义一个 Lua 函数，然后通过执行这个函数来执行脚本。
- EVALSHA 通过构建函数名，直接调用 Lua 中已定义的函数，从而执行相应的脚本。