小象科技 ChinaHadoop.cn

Redis 从入门到精通

黄健宏（huangz）

Lua 脚本

在服务器端执行复杂的操作

前面学习的附加功能

流水线:打包发送多条命令，并在一个回复里面接收所有被 执行命令的结果。

事务:一次执行多条命令，被执行的命令要么就全部都被 执行，要么就一个也不 执行。并且事务执行过程中不会被其他工作打断。

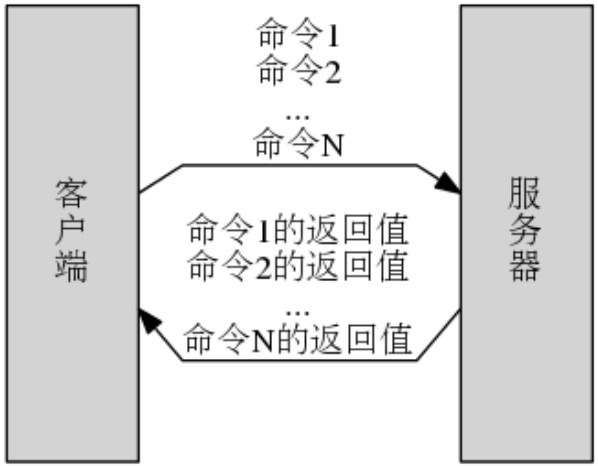
乐观锁:监视特定的键，防止事务出现竞争条件。

虽然这些附加功能都非常有用，但它 们也有一些缺陷。

流水线的缺陷

小象科技 ChinaHadoop.cn

尽管使用流水线可以一次发送多个命令，但是对于一个由多个命令组成的复杂操作来说，为了执行该操作而不断地重复发送相同的命令，这并不是最高效的做法，会对网络资源造成浪费。



如果我们有办法避免重复地发送相同的命令，那么客户端就可以减少花在网络传输方面的时间，操作就可以执行得更快。

事务和乐观锁的缺陷

虽然使用事务可以一次执行多个命令，并且通过乐观锁可以防止事务产生竞争条件，但是在实际中，要正确地使用事务和乐观锁并不是一件容易的事情。

1. 对于一个复杂的事务来说，通常需要仔细思考才能知道应该对哪些键进行加锁：锁了不应该锁的键会增加事务失败的机会，甚至可能会造成程序出错；而忘了对应该锁的键进行加锁的话，程序又会产生竞争条件。

2. 有时候为了防止竞争条件发生，即使操作本身不需要用到事务，但是为了让乐观锁生效，我们也会使用事务将命令包裹起来，这增加了实现的复杂度，并且带来了额外的性能损耗。

误用示例

《事务》一节介绍的 ZDECRBY 命令的实现，这里的事务仅仅是为了让 WATCH 生效而用的：

```
def ZDECRBY(key, decrment, member):
    # 监视输入的有序集合
    WATCH key
    # 取得元素当前的分 值
    old_score = ZSCORE key member
    # 使用当前分值减去指定的减量，得出新的分 值
    new_score = old_score - decrment
    # 使用事务包裹 ZADD 命令
    # 确保 ZADD 命令只会在有序集合没有被修改的情况下 执行
    MULTI
    ZADD key new_score member    # 为元素设置新分值，覆盖现有的分值
    EXEC
```

避免事务被误用的办法

如果有一种方法，可以让我们以事务方式来执行多个命令，并且这种方法不会引入任何竞争条件，那么我们就可以使用这种方法来代替事务和乐观锁。

扩展 Redis 功能时的麻烦

Redis 针对每种数据结构都提供了相应的操作命令，也对数据库本身提供了操作命令，但如果我们需要对数据结构进行一些 Redis 命令不支持的操作，那么就需要使用客户端取出数据，然后由客户端对数据进行处理，最后再将处理后的数据储存回 Redis 服务器。

举个简单的例子，因为 Redis 没有提供删除列表里面所有偶数数字的命令，所以为了执行这一操作，客户端需要取出列表里面的所有项，然后在客户端里面进行过滤，最后将过滤后的项重新推入到列表里面：

```
lst = LRANGE lst 0 -1    # 取出列表包含的所有元素
DEL lst                  # 删除现有的列表
for item in lst:         # 遍历整个列表
    if item % 2 != 0:    # 将非偶数元素推入到列表里面
        RPUSH lst item
```

并且为了保证这个操作的安全性，还要用到事务和乐观锁，非常麻烦。

Lua 脚本

为了解决以上提到的问题，Redis 从 2.6 版本开始在服务器内部嵌入了一个 Lua 解释器，使得用户可以在服务器端执行 Lua 脚本。

这个功能有以下好处：

1. 使用脚本可以直接在服务器端执行 Redis 命令，一般的数据处理操作可以直接使用 Lua 语言或者 Lua 解释器提供的函数库来完成，不必再返回给客户端进行处理。

2. 所有脚本都是以事务的形式来执行的，脚本在执行过程中不会被其他工作打断，也不会引起任何竞争条件，完全可以使用 Lua 脚本来代替事务和乐观锁。

3. 所有脚本都是可重用的，也即是说，重复执行相同的操作时，只要调用储存在服务器内部的脚本缓存就可以了，不用重新发送整个脚本，从而尽可能地节约网络资源。

执行 Lua 脚本

EVAL script numkeys key [key ...] arg [arg ...]

script 参数是要执行的 Lua 脚本。

numkeys 是脚本要处理的数据库键的数量，之后的 key [key …] 参数指定了脚本要处理的数据库键，被传入的键可以在脚本里面通过访问 KEYS 数组来取得，比如 KEYS[1] 就取出第一个输入的键，KEYS[2] 取出第二个输入的键，诸如此类。

arg [arg …] 参数指定了脚本要用到的参数，在脚本里面可以通过访问 ARGV 数组来获取这些参数。

显式地指定脚本里面用到的键是为了配合 Redis 集群对键的检查，如果不这样做的话，在集群里面使用脚本可能会出错。

另外，通过显式地指定脚本要用到的数据库键以及相关参数，而不是将数据库键和参数硬写在脚本里面，用户可以更方便地重用同一个脚本。

EVAL 命令使用示例


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

redis> EVAL "return 1+1" 0
(integer) 2

redis> EVAL "return {KEYS[1], KEYS[2], ARGV[1], ARGV[2]}" 2 "msg" "age" 123 "hello world"
1) "msg"    # KEYS[1]
2) "age"    # KEYS[2]
3) "123"    # ARGV[1]
4) "hello world"    # ARGV[2]
```

在 Lua 脚本中执行 Redis 命令

通过调用 redis.call() 函数或者 redis.pcall() 函数，我们可以直接在 Lua 脚本里面执行 Redis 命令。

```
redis> EVAL "return redis.call('PING')" 0    # 在 Lua 脚本里面执行 PING 命令
PONG

redis> EVAL "return redis.call('DBSIZE')" 0    # 在 Lua 脚本里面执行 DBSIZE 命令
(integer) 4

# 在 Lua 脚本里面执行 GET 命令，取出键 msg 的值，并对值进行字符串拼接操作
redis> SET msg "hello world"
OK
redis> EVAL "return 'The message is: ' .. redis.call('GET', KEYS[1]) '" 1 msg
"The message is: hello world"
```

redis.call() 和 redis.pcall() 的区别

redis.call() 和 redis.pcall() 都可以用来执行 Redis 命令，它们的不同之处在于，当被执行的脚本出错时，redis.call() 会返回出错脚本的名字以及 EVAL 命令的错误信息，而 redis.pcall() 只返回 EVAL 命令的错误信息。

```
redis> EVAL "return redis.call('NotExistsCommand')" 0
(error) ERR Error running script (call to f_ddabd662fa0a8e105765181ee7606562c1e6f1ce):
@user_script:1: @user_script: 1: Unknown Redis command called from Lua script

redis> EVAL "return redis.pcall('NotExistsCommand')" 0
(error) @user_script: 1: Unknown Redis command called from Lua script
```

换句话来说，在被执行的脚本出错时，redis.call() 可以提供更详细的错误信息，方便进行查错。

示例:使用 Lua 脚本重新实现 ZDECRBY 命令

创建一个包含以下内容的 zdecrby.lua 文件:

```
local old_score = redis.call('ZSCORE', KEYS[1], ARGV[2])
local new_score = old_score - ARGV[1]
return redis.call('ZADD', KEYS[1], new_score, ARGV[2])
```

然后通过以下命令来执行脚本:

```
$ redis-cli --eval zdecrby.lua salary , 300 peter
(integer) 0
```

这和在 redis-cli 里面执行 EVAL “local … ” 1 salary 300 peter 效果一样, 但先将脚本内容保存到文件里面, 再执行脚本文件的做法, 比起直接在客 户端里面一个个字 输入要容易一些。

另外, 这个脚本实现的 ZDECRBY 也比使用事 务和乐观锁实现的 ZDECRBY 要简单得多。

使用 EVALSHA 来减少网络资源损耗

任何 Lua 脚本，只要被 EVAL 命令执行过一次，就会被储存到服务器的脚本缓存里面，用户只要通过 EVALSHA 命令，指定被缓存脚本的 SHA1 值，就可以在不发送脚本的情况下，再次执行脚本：

EVALSHA sha1 numkeys key [key ...] arg [arg ...]

通过 SHA1 值来重用返回 ‘hello world’ 信息的脚本：

```
redis> EVAL "return 'hello world'" 0
"hello world"
redis> EVALSHA 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 0
"hello world"
```

通过 SHA1 值来重用之前实现的 ZDECRBY 命令，这样就不用每次都发送整个脚本了：

```
redis> EVALSHA 918130cae39ff0759b8256948742f77300a91cb2 1 salary 500 peter
(integer) 0
```

脚本管理命令

SCRIPT EXISTS sha1 [sha1 ...]

检查 sha1 值所代表的脚本是否已经被加入到脚本缓存里面，是的话返回 1，不是的话返回 0 。

SCRIPT LOAD script

将脚本储存到脚本缓存里面，等待将来 EVALSHA 使用。

SCRIPT FLUSH

清除脚本缓存储存的所有脚本。

SCRIPT KILL

杀死运行超时的脚本。如果脚本已经执行过写入操作，那么还需要使用 SHUTDOWN NOSAVE 命令来强制服务器不保存数据，以免错误的数据被保存到数据库里面。

函数库

Redis 在 Lua 环境里面载入了一些常用的函数库，我们可以使用这些函数库，直接在脚本里面处理数据，它们分别是标准库：

- base 库：包含 Lua 的核心（core）函数，比如 assert、tostring、error、type 等。
- string 库：包含用于处理字符串的函数，比如 find、format、len、reverse 等。
- table 库：包含用于处理表格的函数，比如 concat、insert、remove、sort 等。
- math 库：包含常用的数学计算函数，比如 abs、sqrt、log 等。
- debug 库：包含调试程序所需的函数，比如 sethook、gethook 等。

以及外部库

- struct 库：在 C 语言的结构和 Lua 语言的值之间进行转换。
- cjson 库：将 Lua 值转换为 JSON 对象，或者将 JSON 对象转换为 Lua 值。
- cmsgpack 库：将 Lua 值编码为 MessagePack 格式，或者从 MessagePack 格式里面解码出 Lua 值。

另外还有一个用于计算 sha1 值的外部函数 redis.sha1hex。

复习

回顾一下本节学到的脚本相关知识

复习(1/2)

Lua 脚本可以像事务一样，不被打扰地一次执行多个命令，并且因为脚本不会引起竞争条件，所以使用脚本来实现复杂的操作会比使用事务和乐观锁更方便。

通过 EVALSHA 命令，我们可以在不发送整个脚本的情况下，直接通过脚本的 SHA1 值来执行脚本，对于由多个命令组成的复杂操作来说，使用 EVALSHA 可以有效地减少客户端与服务器通信时需要发送的数据量。

通过使用 Lua 语言，以及 Lua 环境中内置的各个函数库，脚本可以直接在服务器端执行一些数据处理操作，而不必交给客户端来进行处理，从而减少不必要的通信。

复习(2/2)

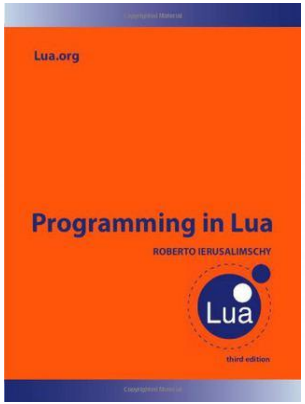
执行脚本的命令:

- EVAL script numkeys key [key ...] arg [arg ...]
- EVALSHA sha1 numkeys key [key ...] arg [arg ...]

管理脚本的命令:

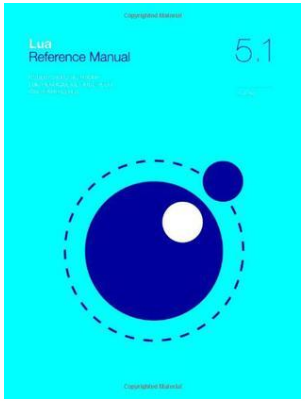
- SCRIPT EXISTS
- SCRIPT LOAD
- SCRIPT FLUSH
- SCRIPT KILL

参考资料



Programming in Lua，第三版

http://www.lua.org/pil/



Lua 5.1 Refernece Manual

www.lua.org/manual/5.1/