# 一 Redis与Lua脚本

## 1 Lua脚本调用redis

redis.call(‘set’, ‘foo’, ‘bar’);

local result = redis.call(‘get’, ‘foo’); //定义局部变量result接收结果

redis的返回结果有5种类型，call函数会将这些类型转换成Lua的数据类型，对照如下：

REDIS Lua

整数 数字类型

字符串 字符串类型

多行字符串的回复 表（即数组）

状态回复 表（只有1个ok字段）

错误回复 表（只有1个err字段）

pcall函数作用和call一致，但是pcall在命令出错时会记录错误并继续执行，calll则直接返回错误，不会继续执行。

在脚本中可以使用return语句将值返回给客户端，没有返回则默认返回nil。redis会自动转换返回值类型！

## 2 EVAL命令

Redis提供了EVAL命令可以使开发者像调用其他Redis内置命令一样调用脚本。EVAL命令的格式是：

EVAL脚本内容 key参数的数量 key key ... [arg ...]

可以通过key和arg这两类参数向脚本传递数据，他们的值在脚本中分别使用KEYS和ARGV两个代表类型的全局变量访问，比如希望脚本执行SET：

return redis.call(‘SET’,KEYS[1],ARGV[1])

现在打开redis-cli执行此脚本：

redis> EVAL “return redis.call(‘SET’, KEYS[1], ARGV[1])” 1 foo bar

## 3 EVALSHA命令

由于脚本较长，会影响带宽，Redis提供了EVALSHA命令允许开发者通过脚本内容的SHA1摘要来执行脚本，该命令的用法和EVAL一样，只不过是将脚本内容替换成脚本内容的SHA1摘要。

Redis在执行EVAL命令时会计算脚本的SHA1摘要并记录在脚本缓存中，执行EVALSHA命令时Redis会根据提供的摘要从脚本缓存中查找对应的脚本内容，如果找到了则执行脚本，否则发布会错误:NOSCRIPT..

在程序中使用EVALSHA命令流程如下：

1 先计算脚本的SHA1摘要，使用EVALSHA命令执行脚本

2 获得返回值，如果返回值NOSCRIPT错误则使用EVAL命令重新执行脚本

大部分Redis客户端会代替开发者完成上述过程。

# 二 脚本实例Node

对象JSON实例化后可以存储到字符串类型键中，如果要对这些对象进行计算，可以使用脚本在服务端完成计算后再返回，既节省了网络带宽，又保证了操作的原子性。

实例：使用脚本统计多个学生的课程分数总和

*//学生类，包括姓名和该学生的所有课程分数*class Student {  
 constructor(name){  
 this.name = name;  
 this.courses = {};  
 }  
 *//添加一个课程，参数为课程名和分数* addCourese(name,score){  
 this.courses[name] = score;  
 }  
}  
  
*//创建两个学生对象*let bob = new Student('Bob');  
bob.addCourese("Math",80);  
bob.addCourese("English",97);  
  
let jeff = new Student('Jeff');  
jeff.addCourese('Math',70);  
jeff.addCourese('History',77);  
  
  
const Redis = require('ioredis');  
const redisClient = new Redis(6379,'119.23.75.123');  
  
*//将对象JSON序列化后存入数据库中*redisClient.mset(  
 'user:1',JSON.stringify(bob),  
 'user:2',JSON.stringify(jeff)  
);  
  
*//编写lua脚本:使用了unpack函数将keys表展开，所以执行脚本可以传入  
//任意数量的键参数*let luaStr = " \  
 local sum = 0 \  
 local users = redis.call('mget',unpack(KEYS)) \  
 for \_, user in ipairs(users) do \  
 local courses = cjson.decode(user).courses \  
 for \_, score in pairs(courses) do \  
 sum = sum + score \  
 end \  
 end \  
 return sum \  
";  
  
*//执行脚本:会先计算SHA1摘要，并尝试使用EVALSHA调用，失败就使用EVAL，建议遵守下列书写规则*redisClient.eval(luaStr, 2, 'user:1', 'user:2', function (err, data) {  
 if(err){  
 console.log(err)  
 }  
 console.log(data); *//输出324*});

# 三 深入脚本

## 1 KEYS与ARGV

有时候键名是根据脚本某部分的执行结果生成的，这时就无法在执行前将键名明确标出。比如一个集合类型键存储了用户ID列表，每个用户使用散列键存储，其中有一个字段是年龄，下面的脚本可以计算某个集合中用户的平均年龄：

local sum = 0

local users = redis.call(‘SMEMBERS, KEYS[1])

for \_, user\_id in ipairs(users) do

local user\_age = redis.call (‘HGET’, ‘user:’ .. user\_id, ‘age’)

sum = sum + user\_age

end

return sum / #users

该脚本无法兼容集群，因为第4行访问了KEYS变量中没有的键，但是避免了数据往返客户端和副段的开销，为了兼容集群，可以在客户端获取集合中的用户ID列表，然后将用户ID组装成键名列表传给脚本并计算平均年龄。

## 2 沙盒与随机数

Redis脚本禁止使用Lua标准库中与文件、系统调用相关的函数，在脚本中只允许对数据进行处理，并且通过禁用全局变量方式保证每个脚本都是相对隔离的。

这样既保证了服务器的拿权，还确保了脚本执行交过只和脚本自身以及传递参数与有关，不依赖外部条件。因为在执行 复制 和 AOF持久化时记录的是脚本的内容而不是脚本的命令，所以必须保证在脚本和内容参数一样的前提下，执行的脚本结果一致。

为了确保执行结果可以重现，Redis还对随机数和会产生随机结果的命令进行了特殊处理：

对随机数而言，Redis替换了math.random和math.randomseed函数，使得每次执行脚本时生成的随机数列都相同，如果希望获得不同的随机数序列，最简单的方法是由程序生成随机数并通过参数传递给脚本，或采用更灵活的方法，即在程序中生成随机数传给脚本作为随机数种子（通过math.randomseed(tonumber(Argv[种子参数索引]))）,这样在脚本中再调用math.random产生的随机数就不同了，由随机数种子决定。

对于会产生随机结果的命令如SMEMBERS或者HJEYS等（因为无序）等Redis会对结果按照字典顺序排序。内部是通过调用Lua标准库的table.sort函数实现的，大致代码如下：

function \_\_redis\_\_compare\_helper(a,b)

if a == false then a = ‘’ + end

if b == false then b = ‘’ + end

return a < b

end

table.sort(result\_array, \_\_redis\_\_compaire\_helper)

对于会产生随机结果但无法排序的命令（比如只会产生一个元素），Redis会在这类命令执行后将该脚本状态标记为lua\_random\_dirty，此后只允许调用只读命令，不允许修改数据库的值，否则会返回错误：

Write commands not allowed after non deterministic commands

属于此类的Redis命令有：SPOP SRANDMEMBER RANDOMKEY TIME

## 3 原子性和执行时间

Redis的脚本执行时原子的，即在脚本执行期间不会执行其他命令。为了防止某个脚本执行时间过长导致Redis无法提供服务，Redis提供了lua-time-limit参数限制脚本的最长运行时间，默认是5秒钟。

当时间超过设定限制时间后，Redis将开始接受其他命令但不会执行（以确保原子性），而是返回 BUSY 错误。但是此时其实还是会执行 SCRIPT KILL 和 SHUTDOWN NOSAVE，执行后脚本终止，并报错： Error running script....

需要注意的是如果当前脚本对Redis的数据进行了修改，则SCRIPT KILL命令不会终止脚本运行以防止脚本只执行了一部分，因为不能违背原子性要求，要么都执行，要呢都不执行。

这时只能通过SHUTDOWN NOSAVE强行终止Redis（NOSAVE 不会进行持久化操作）。

# 四 持久化

## 1 Redis持久化简介

Redis数据存放在内存中，重启后会造成数据丢失，某些情况不允许这种现象：

1 作为数据库使用

2 作为缓存服务器会，缓存被穿透后会对性能造成较大影响，所有缓存同时失效会导致缓存雪崩，从而使服务无法响应。

我们希望Redis能将数据从内存中以某种形式同步到磁盘中，使得重启后仍然可以根据硬盘的记录回复数据。

Redis持久化有两种方式：

1 RDB： 根据指定规则定时将内存中的数据存储在硬盘上

2 AOF： 每次执行命令后都将命令本身记录下来

大多情况下都是两种持久化方式共同使用。

## 2 RDB

### 2.1 简介

RDB通过快照方式完成，当符合一定条件时，Redis会自动将内存中的数据生成一份副本并存储在硬盘上，这个过程即为快照。

快照的条件：

根据配置自动快照；

用户执行SAVE或者BGSAVE命令

执行FLUSHALL命令

执行复制（replication）

### 2.2 配置自动快照

在配置文件中配置：时间窗口M和改动的键个数N

save 900 1

save 300 10

....

每条快照占一行，可以存在多个条件，条件之间的关系是 或。上述示例中第一个快照条件是：900秒内有1个或者1个以上的键被更改则进行快照。

### 2.3 手动快照

如果服务器需要重启、手动迁移、备份等操作时，需要手动快照。

SAVE：

此时会同步进行快照，且会阻塞所有客户端的请求。数据较多时，会导致Redis长时间不能响应，应该尽量避免在生产环境中使用该命令。

BGSAVE：

推荐使用该命令，BGSAVE可以异步进行快照操作，同时还能响应客户端请求，执行该命令后立即返回OK，如果想知道快照是否完成，可以通过LASTSAVE命令查看表示最后一次成功执行快照的时间戳

FLUSHALL：

该命令会清空Redis，如果自动快照条件不为空，则始终会执行一次快照操作。

执行复制：

设置了主从模式后，Redis会在复制初始化时候进行自动快照，即使没有定义自动快照条件，并且没有手动执行快照操作，也会生成RDB快照文件。

### 2.4 快照原理

Redis会默认将快照文件存储在Redis当前进程的工作目录中的dump.rdb文件中，可以通过配置dir和dbfilename两个参数分别指定快照文件的存储路径和文件名，快照过程如下：

1 Redis使用for函数复制一份当前进程的副本（子进程）

2 父进程继续接受客户端命令，子进程开始讲内存中的数据写入硬盘的临时文件

3 子进程写入完所有数据后会用该临时文件替换旧的RDB文件，快照完成

注意：新的RDB文件存储的是执行fork那一刻的内存数据，因为fork函数运行时父子共享内容数据，父进程如果要更改数据，操作系统会将该欧安数据复制一份以保证子进程的数据不受影响。

写时复制策略保证了fork的时候，虽然看上去生成了两份内存副本，但是时间上内存占用量不会增加一倍，即系统内容有2G，Redis有1.5G，fork后内存不会增加到3G，为此需要确保Linux允许应用程序申请超过可用内存（物理内存和交换分区）的空间，方法是：

执行：sysctl vm.overcommit\_memory=1

或者 /etc/sysctl.conf 加入 vm.overcommit\_memory=1 然后重启。

注意：如果在快照时，写入操作较多，造成fork前后数据差异较大，会使内存使用量显著超过实际数据大小，因为内存中不仅保存了当前的数据库数据，而且还保存着fork时候的内存数据，进行内存用量估算时很容易忽略这个问题，造成内存用量超限。

Redis启动后会读取RDB快照文件，将数据从硬盘载入到内存。

RDB方式持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据，如果数据很重要，推荐AOF方式。

## 3 AOF

### 3.1 开启AOF

AOF更加注重数据安全，AOF可以将Redis每一条写命令追加到硬盘文件中，但是会造成性能降低。默认Redis没有开启AOF，开启命令：

appendonly yes

AOF文件和RDB文件保存位置相同，都是通过dir参数设置，默认文件名是appendonly.aof，修改参数命令：

appendfilename appendonly.aof

### 3.2 AOFk的实现

AOF以纯文本的形式记录了Redis执行的写命令。AOF文件的内容就是Redis客户端发送的原始通信协议内容。

比如对一个键输入三次 SET 命令，那么AOF文件会记录三次，这时也会造成AOF文件越来越大，因为前两条SET是无用命令，Redis每当达到一定条件会自动重写AOF文件，配置命令如下：

auto-aof-rewrite-percentage 100 //当前AOF文件大小超过上一次文件百分之多少重写

auto-aof-rewrite-min-size 64mb //限制允许重写的最小文件大小

当然也可以手动执行重写： BGREWRITEAOF

Redis在启动时，会逐个执行AOF文件的命令，所以启动速度比RDB慢。

### 3.3 同步硬盘数据

虽然AOF将命令记录在了AOF文件找那个，但是由于操作系统的缓存机制，数据并没有真正写入硬盘，而是进入了系统的硬盘缓存中。

在操作系统中，每30秒执行一次同步操作，如果在这30秒内系统异常，那么会造成硬盘数据丢失，既然开启了AOF肯定是无法容忍的，我们希望Redis在写入AOF时，主动要求系统将缓存同步到硬盘中：

#appendfsync always

appendfsync everysec

#appendfsync no

默认情况下Redis采用everysec规则，每秒执行一次同步操作，always表示每次执行写入都会执行同步，这是最安全也是最慢的方式。no表示不主动进行同步操作，而是完全交给操作系统来做，这是最快但是最不安全的方式。

一般情况下使用everysec。