redis 实现可重入分布式锁

作者：zsr251
原文链接：https://ld246.com/article/1513007414553
来源网站：链滴

redis实现可重入分布式锁

网上有很多用redis实现分布式锁的例子，但是很多都是有问题的，不能保证命令的原子性，例如:

```
if (jedis.setnx("key","线程标识")>0){
    jedis.expire("key",30);
}
```

  利用setnx命令先进行加锁，如果加锁成功，则设置锁的超时时间，防止线程掉后锁不释放。但是这样的写法不能完全解决这个问题！如果线程在获得锁后立刻挂掉，也就是说还有进行设置超时时间，那么就会出现锁永不释放的情况。解决这个问题可以使用redis的set命令，添参数的方式设置nx的同时设置超时时间，获得通过lua脚本解决。

  这个文章的实现就是通过lua脚本实现的。实现的原理：使用HASH类型当作锁ey，用redis的lua脚本保证原子性，锁key中记录锁获取次数和线程标识，使用BLPOP实现释放锁时通知，防止等待线程自旋浪费cpu资源。暂时还没有看redisson的源码，等看完之后再做优化或重写

  GitHub：https://github.com/zsr251/jedis-distribute-lock
目前项目是研究性质的，使用到生产环境上需谨慎，分布式信号量可直接忽略。

redis的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命令， 轻松实现以前很难实现或者不能高效实现的模式。

lua官方文档：https://www.lua.org/pil/contents.html

脚本的原子性

  Redis 使用单个 Lua 解释器去运行所有脚本，并且， Redis 也保证脚本会以子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行

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。

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 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）：
- 用于计算 SHA1 校验和的 redis.sha1hex 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 ，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，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

脚本的执行

- 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"
```

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：

```
Caller <---------------------------------------- Redis
```

● 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"
```

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
```

EVAL命令

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

  键名参数 key [key ...] 从 EVAL 的第三个参数开始算起，表示在脚本中所用到

那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 KEYS 数组，用 1 为基址的形式访问(EYS[1]， KEYS[2]，以此类推)

  在命令的最后，那些不是键名参数的附加参数 arg [arg ...]，可以在 Lua 中通全局变量 ARGV 数组访问，访问的形式和 KEYS 变量类似(ARGV[1] 、 ARGV[2]，诸如此类)

```
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

lua中调用redis

```
> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
OK
```

- redis.call()
- redis.pcall()

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误

  redis.call() 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚错误，错误的输出信息会说明错误造成的原因

  redis.pcall() 出错时并不引发(raise)错误，而是返回一个带 err 域的 Lua 表(tab e)，用于表示错误

分布式锁使用的脚本

加锁的lua脚本 返回：0获取锁失败 返回1获取锁成功。

持有锁的线程可以再次获得锁，无需等待排队

```
--[[
lock 加锁过程 eval调用
三个参数：key、线程标识、超时时间
--]]
local f = redis.call('HGET',KEYS[1],'flag')
-- 如果线程标识不是空 且不是当前线程 则返回加锁失败
if type(f) == 'string' and f ~= KEYS[2] then
    return 0
end
-- 设置线程标识
redis.call('HSET',KEYS[1],'flag',KEYS[2])
-- 设置超时时间
redis.call('EXPIRE',KEYS[1],KEYS[3])
local c = redis.call('HGET',KEYS[1],'count')
if type(c) ~= 'string' or tonumber(c) < 0 then
    redis.call('HSET',KEYS[1],'count',1)
else
-- 如果是重入，记录获取次数
    redis.call('HSET',KEYS[1],'count',c+1)
end
return 1
```

释放锁的lua脚本 返回：0释放失败 1释放成功，不再持有锁 2单次释放成功，依然持有锁

只有持有锁的线程才能释放锁

```
--[[
unlock 解锁过程 eval调用
两个参数：key、线程标识
--]]
local f = redis.call('HGET',KEYS[1],'flag')
-- 如果线程标识不是空 且不是当前线程 则返回解锁失败
if type(f) ~= 'string' or (type(f) == 'string' and f ~= KEYS[2]) then
    return 0
end
local c = redis.call('HGET',KEYS[1],'count')
if type(c) ~= 'string' or tonumber(c) < 2 then
    redis.call('DEL',KEYS[1])
    -- 释放成功 不再持有
    return 1
else
    redis.call('HSET',KEYS[1],'count',c-1)
    -- 释放成功 但依然持有 即同一个线程多次获得锁的情况
    return 2
end
```

实现

锁释放通知常用的三种方式：

- 一种是自旋获得锁，浪费redis连接和cpu
- 第二种是使用BLPOP监听一个list，当锁释放时往list中插入一个值通知等待线程
- 第三种是使用redis的发布订阅功能通知等待线程

这里选用的是第二种，但是随之而来的有两个问题还没有很好的解决：

1. 可能会出现redis key超时时，锁通知队列中没有通知，造成假性死锁，需要等待下一个获得锁的程进行通知

2. 锁释放通知列表键 在所有请求处理完成后 不会自动删除 但在实际场景中可以接受

针对第一个问题有两个简单的解决方式：

1. 使用BLPOP设置超时时间，使锁定时间可控，同时控制线程饥饿时间

2. 另起一个线程检测redis中所有的锁释放通知队列的长度，如果对应的锁标识为未赋值则通知释放消息

加锁：

```
/**
 * 获得锁 lua脚本
 * 三个参数：key、线程标识、超时时间
 */
public static String LOCK_SCRIPT = "local f = redis.call('HGET',KEYS[1],'flag');if type(f) == 'strin
```

```
' and f ~= KEYS[2] then return 0;end redis.call('HSET',KEYS[1],'flag',KEYS[2]);redis.call('EXPIRE',
EYS[1],KEYS[3]);local c = redis.call('HGET',KEYS[1],'count');if type(c) ~= 'string' or tonumber(c)
< 0 then redis.call('HSET',KEYS[1],'count',1);else redis.call('HSET',KEYS[1],'count',c+1);end retur
 1";
/**
 * 获得锁
 *
 * @param jedis        redis连接
 * @param expireSecond 持有锁超时秒数
 * @param waitSecond   等待锁超时秒数
 * @param flag         线程标识
 * @return
 */
private boolean tryLockInner(Jedis jedis, int expireSecond, int waitSecond, String flag) {
    // 尝试获得锁 如果自身持有锁则可以再次获得
    if ((Long) jedis.eval(LOCK_SCRIPT, 3, redisLockKey, flag, "" + expireSecond) > 0) {
        return true;
    }
    //阻塞等待释放锁通知
    List<String> lp = jedis.blpop(waitSecond, redisListKey);
    if (lp == null || lp.size() < 1) {
        //如果超时则返回锁定失败
        return false;
    }
    return tryLockInner(jedis, expireSecond, waitSecond, flag);
}
```

释放锁

```
/**
 * 释放锁 lua脚本
 * 两个参数：key、线程标识
 */
public static String UNLOCK_SCRIPT = "local f = redis.call('HGET',KEYS[1],'flag');if type(f) ~= 'st
ing' or (type(f) == 'string' and f ~= KEYS[2]) then return 0;end local c = redis.call('HGET',KEYS[
],'count');if type(c) ~= 'string' or tonumber(c) < 2 then redis.call('DEL',KEYS[1]);return 1;else re
is.call('HSET',KEYS[1],'count',c-1);return 2;end";
/**
 * 释放锁
 *
 * @param flag 线程标识
 * @return
 */
public boolean tryUnlock(String flag) {
    Jedis jedis = jedisPool.getResource();
    try {
        //删除锁定的key
        Long l = (Long) jedis.eval(UNLOCK_SCRIPT, 2, redisLockKey, flag);
        if (l < 1) {
            return false;
        }
        // 因为是可重入锁 所以释放成功不一定会释放锁
        if (l.intValue() == 2) {
            return true;
```

```
        }
        //如果锁释放消息队列里没有值 则释放一个信号
        if (l.intValue() == 1 && jedis.llen(redisListKey).intValue() == 0) {
            //通知等待的线程可以继续获得锁
            jedis.rpush(redisListKey, "ok");
        }
        return true;
    } finally {
        jedis.close();
    }
}
```

详细实现请查看源码

参考

- http://redisbook.readthedocs.io/en/latest/feature/scripting.html
- http://redisdoc.com/script/eval.html