0.1 redis 使用与进阶

```
有问题或者宝贵意见联系我的QQ,非常希望你的加入!
```

> 目标 （希望大家仔细研究redis.conf配置文件-本文很多基础的一带而过）

```
1.redis分布式锁,zk分布式锁,lua脚本限流,lua分布式锁
2.redis持久化策略
3.redis集群
4.redis的简单操练
整理大部分常用的场景与使用,如果有疑问或者你不懂的地方请联系我!
```

0.0.1 1 redis分布式锁

> 1.分布式系统（单机的使用ReentrantLock或者synchronized代码块来实现） 2.共享资源大并发产生 3.同步访问

redis分布式锁解决什么问题

```
1.一个进程中的多个线程,多个线程并发访问同一个资源的时候,如何解决线程安全问题。
2.一个分布式架构系统中的两个模块同时去访问一个文件对文件进行读写操作
3.多个应用对同一条数据做修改的时候,如何保证数据的安全性
在但一个进程中,我们可以用到synchronized、lock之类的同步操作去解决,但是对于分布式架构下多进程的情况下,
如何做到跨进程的锁。就需要借助一些第三方手段来完成
```

设计一个分布式所需要解决的问题,分布式锁解决方案（数据库方案）

```
数据库解决方式,创建一个表叫做LOCK表
lock(
  id  int(11)
  methodName  varchar(100),--锁定的方法名称
  memo varchar(1000)
  modifyTime timestamp
  unique key mn (method)  --唯一约束
)

在执行方法的时候,获取锁的伪代码 或者for update行锁 或者 乐观锁方式也是可以的
try{
exec  insert into lock(methodName,memo) values('method','desc');
return true;
}Catch(DuplicateException e){
return false;
}

释放锁:
delete from lock where methodName='';
```

（数据库方案）存在的问题以及思考

```
1.锁没有失效时间,一旦解锁操作失败,就会导致锁记录一直在数据库中,其他线程无法再获得到锁
2.锁是非阻塞的,数据的insert操作,一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列
  要想再次获得锁就要再次触发获得锁操作
3.锁是非重入的,同一个线程在没有释放锁之前无法再次获得该锁
```

ZK方案

```
ZK方案实现之前你要先了解ZK关于他的节点的几个特性:

有序节点:假如当前有一个父节点为/lock,我们可以在这个父节点下面创建子节点;zookeeper提供了一个可选的有序特
性
```

例如我们可以创建子节点“/lock/node-”并且指明有序,那么zookeeper在生成子节点时会根据当前的子节点数量自动添加整数序号
也就是说如果是第一个创建的子节点,那么生成的子节点为/lock/node-0000000000,下一个节点则为/lock/node-0000000001,依次类推

临时节点：客户端可以建立一个临时节点,在会话结束或者会话超时后,zookeeper会自动删除该节点

事件监听：在读取数据时,我们可以同时对节点设置事件监听,当节点数据或结构变化时,zookeeper会通知客户端
当前zookeeper有如下四种事件：
1）节点创建
2）节点删除
3）节点数据修改
4）子节点变更

获取分布式锁的流程 ------- 假设所空间的根节点为/lock

1.客户端连接zookeeper,并在/lock下创建临时的且有序的子节点
第一个客户端对应的子节点为/lock/lock-0000000000,第二个为/lock/lock-0000000001,以此类推

2.--避免羊群效应--客户端获取/lock下的子节点列表,判断自己创建的子节点是否为当前子节点列表中序号最小的子节点,
如果是则认为获得锁,否则监听刚好在自己之前一位的子节点删除消息,获得子节点变更通知后重复此步骤直至获得锁

3.实行业务代码

4.流程完成后,删除对应的子节点并释放锁 （Watch机制）
对应分布式开源包Curator

```
public static boolean accquire() throws IOException, URISyntaxException {
    Jedis jedis = new Jedis( host: "39.107.245.253");

    String lua =
            "local key = KEYS[1] " +
                    " local limit = tonumber(ARGV[1]) " +
                    " local current = tonumber(redis.call('get', key) or '0')" +
                    " if current + 1 > limit " +
                    " then  return 0 " +
                    " else "+
                    " redis.call('INCRBY', key,'1')" +
                    " redis.call('expire', key,'2') " +
                    " end return 1 ";
```

Redis分布式锁方案

Redis 中有许多的命令都可以实现分布式锁,但是比较常用的是SETNX这个命令来实现
有多种方案代码：
1.获取锁,释放锁 代码在redismanager 里面 （简单版）


```
public static boolean accquire() throws IOException, URISyntaxException {
    Jedis jedis = new Jedis( host: "39.107.245.253");

    String lua =
            "local key = KEYS[1] " +
                    " local limit = tonumber(ARGV[1]) " +
                    " local current = tonumber(redis.call('get', key) or '0')" +
                    " if current + 1 > limit " +
                    " then  return 0 " +
                    " else "+
                    " redis.call('INCRBY', key,'1')" +
                    " redis.call('expire', key,'2') " +
                    " end return 1 ";
```

```
public static boolean accquire() throws IOException, URISyntaxException {
    Jedis jedis = new Jedis( host: "39.107.245.253");

    String lua =
            "local key = KEYS[1] " +
                    " local limit = tonumber(ARGV[1]) " +
                    " local current = tonumber(redis.call('get', key) or '0')" +
                    " if current + 1 > limit " +
                    " then  return 0 " +
                    " else "+
                    " redis.call('INCRBY', key,'1')" +
                    " redis.call('expire', key,'2') " +
                    " end return 1 ";
```

```
closeOrder也有 不过是另一种！比较复杂！！
加入时间对比如果当前时间已经大与释放锁的时间
说明已经可以释放这个锁重新在获取锁,setget方法可以把之前的锁去掉在重新获取,旧值在于之前的
值比较,如果无变化说明这个期间没有人获取或者操作这个redis锁,则可以重新获取
```

```
public static boolean accquire() throws IOException, URISyntaxException {
    Jedis jedis = new Jedis( host: "39.107.245.253");

    String lua =
            "local key = KEYS[1] " +
                    " local limit = tonumber(ARGV[1]) " +
                    " local current = tonumber(redis.call('get', key) or '0')" +
                    " if current + 1 > limit " +
                    " then  return 0 " +
                    " else "+
                    " redis.call('INCRBY', key,'1')" +
                    " redis.call('expire', key,'2') " +
                    " end return 1 ";
```

```
redis有成熟的框架redission
```

Redis多路复用机制（看不看都行）

```
 linux的内核会把所有外部设备都看作一个文件来操作,对一个文件的读写操作会调用内核提供的系统命令,
 返回一个 file descriptor（文件描述符）。对于一个socket的读写也会有响应的描述符,称为socketfd(socket
描述符)。
 而IO多路复用是指内核一旦发现进程指定的一个或者多个文件描述符IO条件准备好以后就通知该进程
 IO多路复用又称为事件驱动,操作系统提供了一个功能,当某个socket可读或者可写的时候,它会给一个通知。
 当配合非阻塞socket使用时,只有当系统通知我哪个描述符可读了,我才去执行read操作,可以保证每次read都能读到有
效数据。
 操作系统的功能通过select/pool/epoll/kqueue之类的系统调用函数来使用,这些函数可以同时监视多个描述符的读
写就绪情况
 ,这样多个描述符的I/O操作都能在一个线程内并发交替完成,这就叫I/O多路复用,这里的复用指的是同一个线程
 多路复用的优势在于用户可以在一个线程内同时处理多个socket的 io请求。达到同一个线程同时处理多个IO请求的目
的。
 而在同步阻塞模型中,必须通过多线程的方式才能达到目的
```

Redis（2.6以后）--lua脚本

```
1.减少网络开销,在Lua脚本中可以把多个命令放在同一个脚本中运行
2.原子操作,redis会将整个脚本作为一个整体执行,中间不会被其他命令插入。换句话说,编写脚本的过程中无需担心会出
现竞态条件
3.复用性,客户端发送的脚本会永远存储在redis中,这意味着其他客户端可以复用这一脚本来完成同样的逻辑
4.脚本可以通过return 来返回客户端
例子:  利用lua脚本进行电话号或则IP限流 实例 具体请看 redislua类
KEYS[1] ARGV[1] ARGV[2]  key 参数1  参数2
local num=redis.call('incr',KEYS[1])
if tonumber(num)==1 then
   redis.call('expire',KEYS[1],ARGV[1])
   return 1
elseif tonumber(num)>tonumber(ARGV[2]) then
   return 0
else
   return 1
end
```

Redis（2.6以后）--lua--EVALSHA命令

```
考虑到我们通过eval执行lua脚本,脚本比较长的情况下,每次调用脚本都需要把整个脚本传给redis
比较占用带宽。为了解决这个问题,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
1. 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
2. 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
如果找到了就执行脚本,否则返回“NOSCRIPT No matching script,Please use EVAL”
```

Redis（2.6以后）--lua脚本运行限制

```
redis的脚本执行是原子的,即脚本执行期间Redis不会执行其他命令
所有的命令必须等待脚本执行完以后才能执行。为了防止某个脚本执行时间过程导致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
当脚本运行时间超过这个限制后,Redis将开始接受其他命令但不会执行（以确保脚本的原子性）,而是返回BUSY的错误

在第一个窗口中执行lua脚本的死循环
eval “while true do end” 0 在第二个窗口中运行get hello
最后第二个窗口的运行结果是Busy, 可以通过script kill命令终止正在执行的脚本
如果当前执行的lua脚本对redis的数据进行了修改,比如（set）操作,那么script kill命令没办法终止脚本的运行,
因为要保证lua脚本的原子性。如果执行一部分终止了,就违背了这一个原则
在这种情况下,只能通过 shutdown nosave命令强行终止
```

Redis（2.6以后）--lua分布式锁

```
public static boolean accquire() throws IOException, URISyntaxException {
    Jedis jedis = new Jedis( host: "39.107.245.253");

    String lua =
            "local key = KEYS[1] " +
                    " local limit = tonumber(ARGV[1]) " +
                    " local current = tonumber(redis.call('get', key) or '0')" +
                    " if current + 1 > limit " +
                    " then  return 0 " +
                    " else "+
                    " redis.call('INCRBY', key,'1')" +
                    " redis.call('expire', key,'2') " +
                    " end return 1 ";
```

如何利用lua + redis 取代 nigix + lua 脚本进行分布式限流

分布式限流的关键就是把限流做成具有原子性的功能，可以使用redis + lua 来进行技术实现，高并发和高性能，实现时间窗口内的流量控制 操作在lua脚本中又因为redis是单线程的，因此线程安全！Redis 将整个脚本作为一个原子执行, 无需担心并发, 也就无需事务;

```
public static boolean accquire() throws IOException, URISyntaxException {
    Jedis jedis = new Jedis( host: "39.107.245.253");

    String lua =
            "local key = KEYS[1] " +
                    " local limit = tonumber(ARGV[1]) " +
                    " local current = tonumber(redis.call('get', key) or '0')" +
                    " if current + 1 > limit " +
                    " then  return 0 " +
                    " else "+
                    " redis.call('INCRBY', key,'1')" +
                    " redis.call('expire', key,'2') " +
                    " end return 1 ";
```

```
/**
 * 分布式限流
 */
try {
    RedisLimitRateWithLUA.accquire();
} catch (IOException e) {
    result.withError(EXCEPTION.getCode(), REPEATE_MIAOSHA.getMessage());
    return result;
} catch (URISyntaxException e) {
    result.withError(EXCEPTION.getCode(), REPEATE_MIAOSHA.getMessage());
    return result;
}
```

lua脚本一致存在于redis 中可以 限制每秒钟的请求数实现限流！