2856631955

1、面试题

一般实现分布式锁都有哪些方式？使用redis如何设计分布式锁？使用zk来设计分布式锁可以吗？这两种分布式锁的实现方式哪种效率比较高？

2、面试官心里分析

其实一般问问题，都是这么问的，先问问你zk，然后其实是要过度的zk关联的一些问题里去，比如分布式锁。因为在分布式系统开发中，分布式锁的使用场景还是很常见的。

3、面试题剖析

（1）redis分布式锁

官方叫做RedLock算法，是redis官方支持的分布式锁算法。

这个分布式锁有3个重要的考量点，互斥（只能有一个客户端获取锁），不能死锁，容错（大部分redis节点或者这个锁就可以加可以释放）

第一个最普通的实现方式，如果就是在redis里创建一个key算加锁

SET my:lock 随机值 NX PX 30000，这个命令就ok，这个的NX的意思就是只有key不存在的时候才会设置成功，PX 30000的意思是30秒后锁自动释放。别人创建的时候如果发现已经有了就不能加锁了。

释放锁就是删除key，但是一般可以用lua脚本删除，判断value一样才删除：

关于redis如何执行lua脚本，自行百度

if redis.call("get",KEYS[1]) == ARGV[1] then

return redis.call("del",KEYS[1])

else

return 0

end

为啥要用随机值呢？因为如果某个客户端获取到了锁，但是阻塞了很长时间才执行完，此时可能已经自动释放锁了，此时可能别的客户端已经获取到了这个锁，要是你这个时候直接删除key的话会有问题，所以得用随机值加上面的lua脚本来释放锁。

但是这样是肯定不行的。因为如果是普通的redis单实例，那就是单点故障。或者是redis普通主从，那redis主从异步复制，如果主节点挂了，key还没同步到从节点，此时从节点切换为主节点，别人就会拿到锁。

第二个问题，RedLock算法

这个场景是假设有一个redis cluster，有5个redis master实例。然后执行如下步骤获取一把锁：

1）获取当前时间戳，单位是毫秒

2）跟上面类似，轮流尝试在每个master节点上创建锁，过期时间较短，一般就几十毫秒

3）尝试在大多数节点上建立一个锁，比如5个节点就要求是3个节点（n / 2 +1）

4）客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了

5）要是锁建立失败了，那么就依次删除这个锁

6）只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁

（2）zk分布式锁

zk分布式锁，其实可以做的比较简单，就是某个节点尝试创建临时znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能注册个监听器监听这个锁。释放锁就是删除这个znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新枷锁。

/\*\*

\* ZooKeeperSession

\* @author Administrator

\*

\*/

public class ZooKeeperSession {

private static CountDownLatch connectedSemaphore = new CountDownLatch(1);

private ZooKeeper zookeeper;

private CountDownLatch latch;

public ZooKeeperSession() {

try {

this.zookeeper = new ZooKeeper(

"192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181",

50000,

new ZooKeeperWatcher());

try {

connectedSemaphore.await();

} catch(InterruptedException e) {

e.printStackTrace();

}

System.out.println("ZooKeeper session established......");

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 获取分布式锁

\* @param productId

\*/

public Boolean acquireDistributedLock(Long productId) {

String path = "/product-lock-" + productId;

try {

zookeeper.create(path, "".getBytes(),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

return true;

} catch (Exception e) {

while(true) {

try {

Stat stat = zk.exists(path, true); // 相当于是给node注册一个监听器，去看看这个监听器是否存在

if(stat != null) {

this.latch = new CountDownLatch(1);

this.latch.await(waitTime, TimeUnit.MILLISECONDS);

this.latch = null;

}

zookeeper.create(path, "".getBytes(),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

return true;

} catch(Exception e) {

continue;

}

}

// 很不优雅，我呢就是给大家来演示这么一个思路

// 比较通用的，我们公司里我们自己封装的基于zookeeper的分布式锁，我们基于zookeeper的临时顺序节点去实现的，比较优雅的

}

return true;

}

/\*\*

\* 释放掉一个分布式锁

\* @param productId

\*/

public void releaseDistributedLock(Long productId) {

String path = "/product-lock-" + productId;

try {

zookeeper.delete(path, -1);

System.out.println("release the lock for product[id=" + productId + "]......");

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 建立zk session的watcher

\* @author Administrator

\*

\*/

private class ZooKeeperWatcher implements Watcher {

public void process(WatchedEvent event) {

System.out.println("Receive watched event: " + event.getState());

if(KeeperState.SyncConnected == event.getState()) {

connectedSemaphore.countDown();

}

if(this.latch != null) {

this.latch.countDown();

}

}

}

/\*\*

\* 封装单例的静态内部类

\* @author Administrator

\*

\*/

private static class Singleton {

private static ZooKeeperSession instance;

static {

instance = new ZooKeeperSession();

}

public static ZooKeeperSession getInstance() {

return instance;

}

}

/\*\*

\* 获取单例

\* @return

\*/

public static ZooKeeperSession getInstance() {

return Singleton.getInstance();

}

/\*\*

\* 初始化单例的便捷方法

\*/

public static void init() {

getInstance();

}

}

（3）redis分布式锁和zk分布式锁的对比

redis分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能

zk分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小

另外一点就是，如果是redis获取锁的那个客户端bug了或者挂了，那么只能等待超时时间之后才能释放锁；而zk的话，因为创建的是临时znode，只要客户端挂了，znode就没了，此时就自动释放锁

redis分布式锁大家每发现好麻烦吗？遍历上锁，计算时间等等。。。zk的分布式锁语义清晰实现简单

所以先不分析太多的东西，就说这两点，我个人实践认为zk的分布式锁比redis的分布式锁牢靠、而且模型简单易用

（4）zookeeper分布式锁实现代码

public class ZooKeeperDistributedLock implements Watcher{

private ZooKeeper zk;

private String locksRoot= "/locks";

private String productId;

private String waitNode;

private String lockNode;

private CountDownLatch latch;

private CountDownLatch connectedLatch = new CountDownLatch(1);

private int sessionTimeout = 30000;

public ZooKeeperDistributedLock(String productId){

this.productId = productId;

try {

String address = "192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181";

zk = new ZooKeeper(address, sessionTimeout, this);

connectedLatch.await();

} catch (IOException e) {

throw new LockException(e);

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

}

public void process(WatchedEvent event) {

if(event.getState()==KeeperState.SyncConnected){

connectedLatch.countDown();

return;

}

if(this.latch != null) {

this.latch.countDown();

}

}

public void acquireDistributedLock() {

try {

if(this.tryLock()){

return;

}

else{

waitForLock(waitNode, sessionTimeout);

}

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

}

public boolean tryLock() {

try {

// 传入进去的locksRoot + “/” + productId

// 假设productId代表了一个商品id，比如说1

// locksRoot = locks

// /locks/10000000000，/locks/10000000001，/locks/10000000002

lockNode = zk.create(locksRoot + "/" + productId, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);

// 看看刚创建的节点是不是最小的节点

// locks：10000000000，10000000001，10000000002

List<String> locks = zk.getChildren(locksRoot, false);

Collections.sort(locks);

if(lockNode.equals(locksRoot+"/"+ locks.get(0))){

//如果是最小的节点,则表示取得锁

return true;

}

//如果不是最小的节点，找到比自己小1的节点

int previousLockIndex = -1;

for(int i = 0; i < locks.size(); i++) {

if(lockNode.equals(locksRoot + “/” + locks.get(i))) {

previousLockIndex = i - 1;

break;

}

}

this.waitNode = locks.get(previousLockIndex);

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

return false;

}

private boolean waitForLock(String waitNode, long waitTime) throws InterruptedException, KeeperException {

Stat stat = zk.exists(locksRoot + "/" + waitNode, true);

if(stat != null){

this.latch = new CountDownLatch(1);

this.latch.await(waitTime, TimeUnit.MILLISECONDS); this.latch = null;

}

return true;

}

public void unlock() {

try {

// 删除/locks/10000000000节点

// 删除/locks/10000000001节点

System.out.println("unlock " + lockNode);

zk.delete(lockNode,-1);

lockNode = null;

zk.close();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (KeeperException e) {

e.printStackTrace();

}

}

public class LockException extends RuntimeException {

private static final long serialVersionUID = 1L;

public LockException(String e){

super(e);

}

public LockException(Exception e){

super(e);

}

}

// 如果有一把锁，被多个人给竞争，此时多个人会排队，第一个拿到锁的人会执行，然后释放锁，后面的每个人都会去监听排在自己前面的那个人创建的node上，一旦某个人释放了锁，排在自己后面的人就会被zookeeper给通知，一旦被通知了之后，就ok了，自己就获取到了锁，就可以执行代码了

}