大数据hw4报告

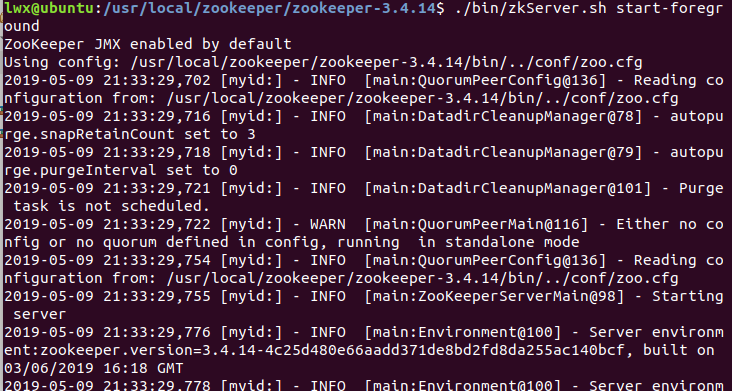
单人作业。根据 ZooKeeper 的指导文档完成安装与基本使用，根据PPT所讲和代码示例实现共享锁。

    报告内容及要求：

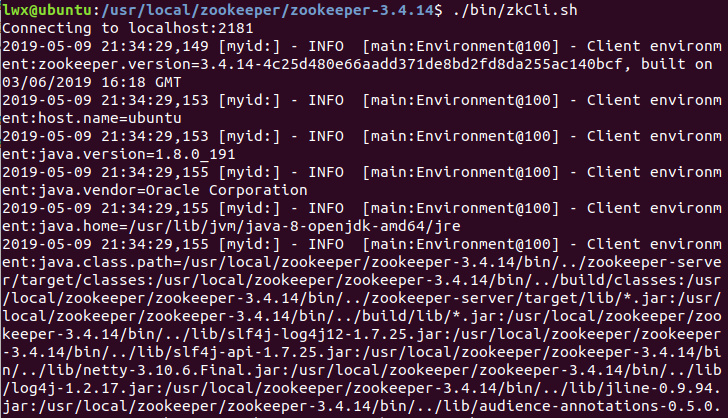
    1. 请在报告中详细写明你的实验步骤、技术方法、实习体会等，必要时附上相应的代码段或截图。  
    2. 截止时间：请同学们于北京时间2019年5月13日23:59前，在北大教学网课程页面，提交pdf形式的上机报告，文件名格式为："报告4-张三-1500012345.pdf"。  
    3. 每延迟一天当次作业成绩扣除20%，三天以上记为0。

# 基本使用

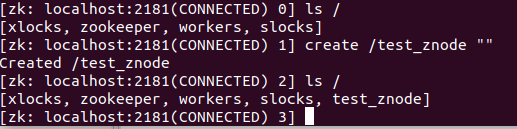
启动zookeeper服务



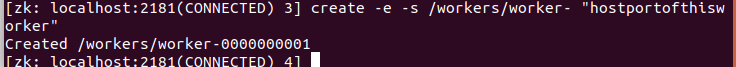
客户端运行



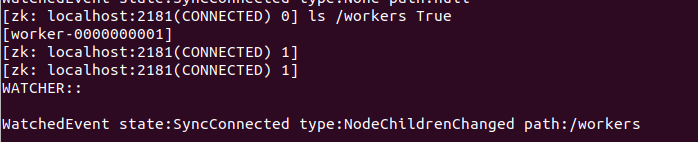
查看znode，新建znode



创建临时、有序的节点



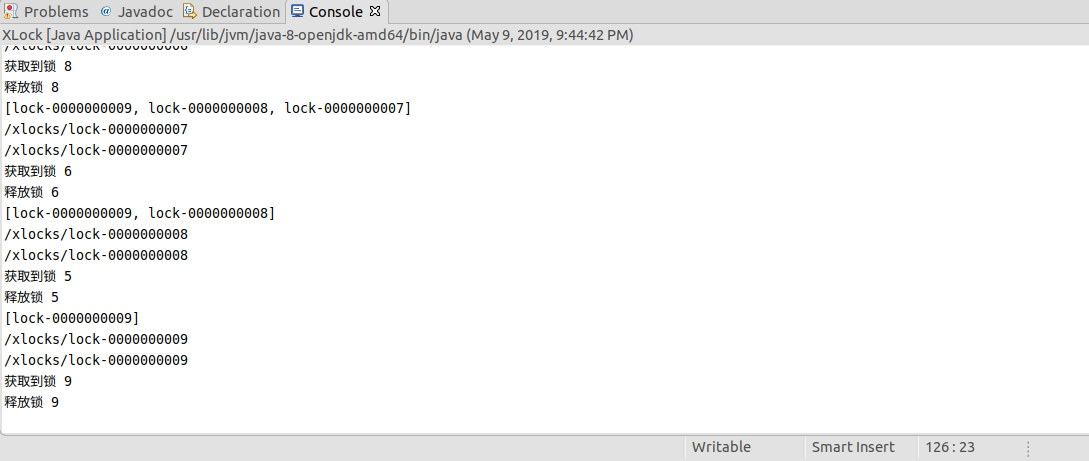
监视子节点变化



# 排他锁

只有最多一个线程可以拿到排他锁

10个线程，每个线程创建一个节点，只有在比这个线程的节点数值小的节点全部删除后，该节点才可以拿到排他锁，等待一段时间后删除该节点，下一个节点即可拿到排他锁。



# 共享锁

可以有多个读锁，只能有最多一个写锁，读锁和写锁不能共存。

写锁仍然为String，读锁为ArrayList<String>。5个线程是读者，5个线程是写者。

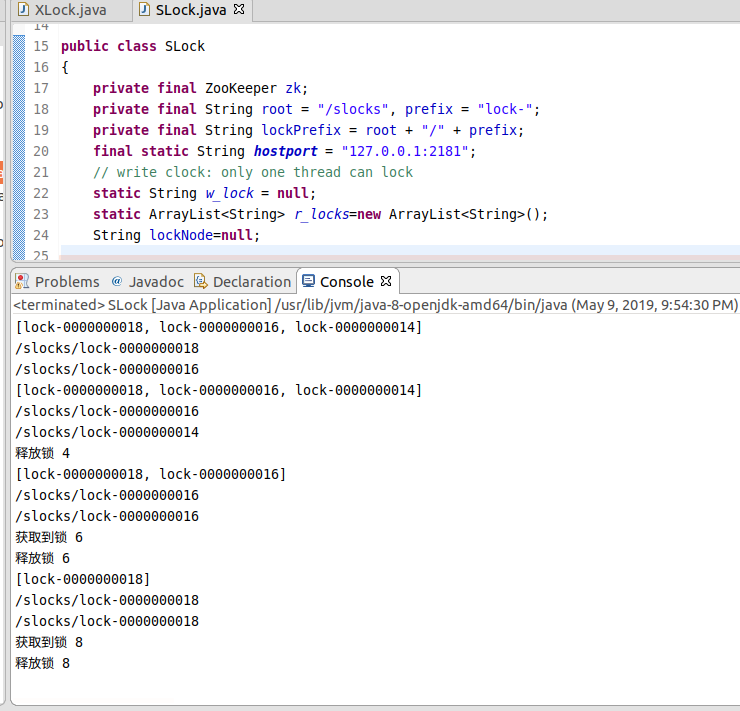
获得读锁的前提条件：存在读锁或者不存在读锁也不存在写锁。

否则说明此时有写锁，需要等待写锁节点删除。

获得写锁的前提条件：不存在读锁也不存在写锁。为与排他锁的实现保持一致，节点按从小到大的顺序获得写锁。即需要等待比该节点小的所有节点删除和所有读节点删除。

删除写锁时删除节点，将写锁值设为null

删除读锁时删除节点，从list中将读锁删除。



# 心得

实现互斥锁时只需按顺序等待即可，而实现共享锁时写锁仍然遵循按顺序等待的原则，读锁只需要存在已有读锁即可获得。

实现共享锁时判断有无读锁或写锁不仅仅通过节点的存在来判断，还需要根据读锁的ArrayList<String>和写锁的String判断。

# 附件：共享锁代码

**package** zk;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.concurrent.CountDownLatch;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooKeeper;

**import** org.apache.zookeeper.data.Stat;

**import** org.apache.zookeeper.ZooDefs.Ids;

**public** **class** SLock

{

**private** **final** ZooKeeper zk;

**private** **final** String root = "/slocks", prefix = "lock-";

**private** **final** String lockPrefix = root + "/" + prefix;

**final** **static** String ***hostport*** = "127.0.0.1:2181";

// write clock: only one thread can lock

**static** String *w\_lock* = **null**;

**static** ArrayList<String> *r\_locks*=**new** ArrayList<String>();

String lockNode=**null**;

**boolean** is\_reader;

**public** SLock(**boolean** \_is\_reader) **throws** IOException

{

zk = **new** ZooKeeper(***hostport***, 15000, (event) -> {});

**this**.is\_reader=\_is\_reader;

}

**public** **void** writer\_lock() **throws** KeeperException, InterruptedException

{

**this**.lockNode = zk.create(lockPrefix, " ".getBytes(), Ids.***OPEN\_ACL\_UNSAFE***,

CreateMode.***EPHEMERAL\_SEQUENTIAL***);

**while** (**true**)

{

// 不断循环直到自己是最小的节点

List<String> children = zk.getChildren(root, **false**);

String closest = closest(children, lockNode.substring(lockNode.lastIndexOf("/") + 1));

System.***out***.println(children);

System.***out***.println(lockNode);

System.***out***.println(closest);

**if** (closest.equals(lockNode))

{

**if**(*r\_locks*.isEmpty())

{

*w\_lock* = lockNode;

**break**;

}

}

**final** CountDownLatch signal = **new** CountDownLatch(1);

Stat exists = zk.exists(closest, **new** Watcher()

{

@Override

**public** **void** process(WatchedEvent event)

{

// 监视的节点被删除

**if** (event.getType() == Event.EventType.***NodeDeleted***)

{

signal.countDown();

}

}

});

// 返回null说明节点不存在，反之存在阻塞等待。

**if** (exists != **null**)

{

signal.await();// 阻塞直到signal.countDown执行。

}

**while**(!*r\_locks*.isEmpty())

{

Stat r\_exists = zk.exists(*r\_locks*.get(0), **new** Watcher()

{

@Override

**public** **void** process(WatchedEvent event)

{

// 监视的节点被删除

**if** (event.getType() == Event.EventType.***NodeDeleted***)

{

signal.countDown();

}

}

});

// 返回null说明节点不存在，反之存在阻塞等待。

**if** (r\_exists != **null**)

{

signal.await();// 阻塞直到signal.countDown执行。

}

}

}

}

**public** **void** reader\_lock() **throws** KeeperException, InterruptedException

{

**this**.lockNode = zk.create(lockPrefix, " ".getBytes(), Ids.***OPEN\_ACL\_UNSAFE***,

CreateMode.***EPHEMERAL\_SEQUENTIAL***);

**while**(**true**)

{

**if**(!*r\_locks*.isEmpty())

{

*r\_locks*.add(lockNode);

**return**;

}

**if**(*w\_lock*==**null**)

{

*r\_locks*.add(lockNode);

**return**;

}

**final** CountDownLatch signal = **new** CountDownLatch(1);

Stat w\_exists = zk.exists(*w\_lock*, **new** Watcher()

{

@Override

**public** **void** process(WatchedEvent event)

{

// 监视的节点被删除

**if** (event.getType() == Event.EventType.***NodeDeleted***)

{

signal.countDown();

}

}

});

// 返回null说明节点不存在，反之存在阻塞等待。

**if** (w\_exists != **null**)

{

signal.await();// 阻塞直到signal.countDown执行。

}

}

}

**private** String closest(List<String> children, String lockNode)

{

// 寻找比自己小的节点，如果没有返回自己。

**int** nodeVal = valueOf(lockNode);

**int** closestVal = -1;

String closestNode = lockNode;

**for** (String child : children)

{

**int** childVal = valueOf(child);

**if** (childVal < nodeVal && childVal > closestVal)

{

closestVal = childVal;

closestNode = child;

}

}

**return** root + "/" + closestNode;

}

**private** **int** valueOf(String node)

{

// System.out.println("enter valueOf");

// System.out.println(node);

// 返回该节点的序号(后缀)

**int** i = node.lastIndexOf(prefix);

// System.out.println(i);

String substring = node.substring(i + prefix.length());

**int** val\_substring=Integer.*valueOf*(substring);

// System.out.println(val\_substring);

**return** val\_substring;

}

**public** **void** lock()**throws** KeeperException, InterruptedException

{

**if**(**this**.is\_reader)

reader\_lock();

**else**

writer\_lock();

}

**public** **void** reader\_unlock() **throws** InterruptedException, KeeperException

{

zk.delete(lockNode, -1);

*r\_locks*.remove(lockNode);

}

**public** **void** writer\_unlock() **throws** InterruptedException, KeeperException

{

zk.delete(lockNode, -1);

*w\_lock*=**null**;

}

**public** **void** unlock() **throws** InterruptedException, KeeperException

{

**if**(**this**.is\_reader)

reader\_unlock();

**else**

writer\_unlock();

}

**public** **static** **void** main(String[] args) **throws** IOException

{

//测试

**for** (**int** i = 0; i < 10; i++)

{

// 并发子线程

**new** Actor(i).start();

}

System.***in***.read();

}

}

**class** Actor **extends** Thread

{

**int** id;

SLock lock;

**public** Actor(**int** id) **throws** IOException

{

**this**.id = id;

**boolean** is\_reader=**true**;

**if**(id%2==0)

{

is\_reader=**false**;

}

lock = **new** SLock(is\_reader);

}

**public** **void** run()

{

**try**

{

System.***out***.println("等待锁 " + id);

lock.lock();

System.***out***.println("获取到锁 " + id);

Thread.*sleep*(1000);

lock.unlock();

System.***out***.println("释放锁 " + id);

}

**catch** (InterruptedException | KeeperException e)

{

e.printStackTrace();

}

}

}