一种简单的分布式ID生成器解决方案

转至元数据结尾

创建： 王万彬，最新修改： 牛占军 于 九月 07, 2017 转至元数据起始

一、分布式系统带来ID生成挑战

在复杂的系统中,往往需要对大量的数据如订单,账户进行标识,以一个有意义的有序的序列号来作为全局唯一的ID;

而分布式系统中我们对ID生成器要求又有哪些呢?

全局唯一性：不能出现重复的ID号，既然是唯一标识，这是最基本的要求。

递增：比较低要求的条件为趋势递增,即保证下一个ID一定大于上一个ID,而比较苛刻的要求是连续递增,如1,2,3等等。

高可用高性能：ID生成事关重大,一旦挂掉系统崩溃;高性能是指必须要在压测下表现良好,如果达不到要求则在高并发环境下依然会导致系统瘫痪。

二、业内方案简介

1. UUID方案

优点：

能够保证独立性，程序可以在不同的数据库间迁移，效果不受影响。

保证生成的ID不仅是表独立的，而且是库独立的，这点在你想切分数据库的时候尤为重要。

缺点：

1. 性能为题：UUID太长，通常以36长度的字符串表示,对MySQL索引不利：如果作为数据库主键，在InnoDB引擎下，UUID的无序性可能会引起数据位置频繁变动，严重影响性能

2. UUID无业务含义：很多需要ID能标识业务含义的地方不使用

3.不满足递增要求

2. snowflake方案

snowflake是twitter开源的分布式ID生成系统。 Twitter每秒有数十万条消息的请求，每条消息都必须分配一条唯一的id，这些id还需要一些大致的顺序（方便客户端排序），并且在分布式系统中不同机器产生的id必须不同。

snowflake的结构如下(每部分用-分开):

0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 – 000000000000

第一位为未使用，接下来的41位为毫秒级时间(41位的长度可以使用69年)，然后是5位datacenterId和5位workerId(10位的长度最多支持部署1024个节点） ，最后12位是毫秒内的计数（12位的计数顺序号支持每个节点每毫秒产生4096个ID序号）

一共加起来刚好64位，为一个Long型。(转换成字符串长度为18)

snowflake生成的ID整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞（由datacenter和workerId作区分），并且效率较高。snowflake的缺点是:

强依赖时钟,如果主机时间回拨,则会造成重复ID,会产生

ID虽然有序,但是不连续

snowflake现在有较好的改良方案,比如美团点评开源的分布式ID框架：leaf，通过使用ZooKeeper解决了时钟依赖问题。

snowflake的关键源码如下：

snowflake算法

/\*\*

\* Twitter\_Snowflake<br>

\* SnowFlake的结构如下(每部分用-分开):<br>

\* 0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000 <br>

\* 1位标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0<br>

\* 41位时间截(毫秒级)，注意，41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截)

\* 得到的值），这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的（如下下面程序IdWorker类的startTime属性）。41位的时间截，可以使用69年，年T = (1L << 41) / (1000L \* 60 \* 60 \* 24 \* 365) = 69<br>

\* 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId<br>

\* 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号<br>

\* 加起来刚好64位，为一个Long型。<br>

\* SnowFlake的优点是，整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞(由数据中心ID和机器ID作区分)，并且效率较高，经测试，SnowFlake每秒能够产生26万ID左右。

\*/

public class SnowflakeIdWorker {

// ==============================Fields===========================================

/\*\* 开始时间截 (2015-01-01) \*/

private final long twepoch = 1420041600000L;

/\*\* 机器id所占的位数 \*/

private final long workerIdBits = 5L;

/\*\* 数据标识id所占的位数 \*/

private final long datacenterIdBits = 5L;

/\*\* 支持的最大机器id，结果是31 (这个移位算法可以很快的计算出几位二进制数所能表示的最大十进制数) \*/

private final long maxWorkerId = -1L ^ (-1L << workerIdBits);

/\*\* 支持的最大数据标识id，结果是31 \*/

private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);

/\*\* 序列在id中占的位数 \*/

private final long sequenceBits = 12L;

/\*\* 机器ID向左移12位 \*/

private final long workerIdShift = sequenceBits;

/\*\* 数据标识id向左移17位(12+5) \*/

private final long datacenterIdShift = sequenceBits + workerIdBits;

/\*\* 时间截向左移22位(5+5+12) \*/

private final long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

/\*\* 生成序列的掩码，这里为4095 (0b111111111111=0xfff=4095) \*/

private final long sequenceMask = -1L ^ (-1L << sequenceBits);

/\*\* 工作机器ID(0~31) \*/

private long workerId;

/\*\* 数据中心ID(0~31) \*/

private long datacenterId;

/\*\* 毫秒内序列(0~4095) \*/

private long sequence = 0L;

/\*\* 上次生成ID的时间截 \*/

private long lastTimestamp = -1L;

//==============================Constructors=====================================

/\*\*

\* 构造函数

\* @param workerId 工作ID (0~31)

\* @param datacenterId 数据中心ID (0~31)

\*/

public SnowflakeIdWorker(long workerId, long datacenterId) {

if (workerId > maxWorkerId || workerId < 0) {

throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0", maxWorkerId));

}

if (datacenterId > maxDatacenterId || datacenterId < 0) {

throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0", maxDatacenterId));

}

this.workerId = workerId;

this.datacenterId = datacenterId;

}

// ==============================Methods==========================================

/\*\*

\* 获得下一个ID (该方法是线程安全的)

\* @return SnowflakeId

\*/

public synchronized long nextId() {

long timestamp = timeGen();

//如果当前时间小于上一次ID生成的时间戳，说明系统时钟回退过这个时候应当抛出异常

if (timestamp < lastTimestamp) {

throw new RuntimeException(

String.format("Clock moved backwards. Refusing to generate id for %d milliseconds", lastTimestamp - timestamp));

}

//如果是同一时间生成的，则进行毫秒内序列

if (lastTimestamp == timestamp) {

sequence = (sequence + 1) & sequenceMask;

//毫秒内序列溢出

if (sequence == 0) {

//阻塞到下一个毫秒,获得新的时间戳

timestamp = tilNextMillis(lastTimestamp);

}

}

//时间戳改变，毫秒内序列重置

else {

sequence = 0L;

}

//上次生成ID的时间截

lastTimestamp = timestamp;

//移位并通过或运算拼到一起组成64位的ID

return ((timestamp - twepoch) << timestampLeftShift) //

| (datacenterId << datacenterIdShift) //

| (workerId << workerIdShift) //

| sequence;

}

/\*\*

\* 阻塞到下一个毫秒，直到获得新的时间戳

\* @param lastTimestamp 上次生成ID的时间截

\* @return 当前时间戳

\*/

protected long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {

timestamp = timeGen();

}

return timestamp;

}

/\*\*

\* 返回以毫秒为单位的当前时间

\* @return 当前时间(毫秒)

\*/

protected long timeGen() {

return System.currentTimeMillis();

}

//==============================Test=============================================

/\*\* 测试 \*/

public static void main(String[] args) throws InterruptedException {

SnowflakeIdWorker idWorker = new SnowflakeIdWorker(0, 0);

for (int i = 0; i < 100; i++) {

long id = idWorker.nextId();

//System.out.println(Long.toBinaryString(id));

Thread.sleep(1);

System.out.println(id);

}

}

}

3. 基于数据库方案

利用数据库生成ID是最常见的方案。能够确保ID全数据库唯一。其优缺点如下：

优点：

非常简单，利用现有数据库系统的功能实现，成本小，有DBA专业维护。

ID号单调自增，可以实现一些对ID有特殊要求的业务。

缺点:

不同数据库语法和实现不同，数据库迁移的时候或多数据库版本支持的时候需要处理。

在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。

在性能达不到要求的情况下，比较难于扩展。

如果涉及多个系统需要合并或者数据迁移会比较麻烦。

分表分库的时候会有麻烦。

4.其他方案简介

通过Redis生成ID、ZooKeeper生成ID、MongoDB的ObjectID等均可实现唯一性的要求

三、我们在实际应用中经历的方案

1. 方案简介

实际业务中，除了分布式ID全局唯一之外，还有是否趋势/连续递增的要求。根据具体业务需求的不同，有两种可选方案。

一是只保证全局唯一，不保证连续递增。二是既保证全局唯一，又保证连续递增。

2. 基于ZooKeeper和本地缓存的方案

基于zookeeper分布式ID实现方案有很多种,本方案只使用ZooKeeper作为分段节点协调工具。每台服务器首先从zookeeper缓存一段,如1-1000的id,

此时zk上保存最大值1000,每次获取的时候都会进行判断,如果id<=1000,则更新本地的当前值,如果为1001,则会将zk上的最大值更新至2000,本地缓存

段更新为1001-2000,更新的时候使用curator的分布式锁来实现。

由于ID是从本机获取，因此本方案的优点是性能非常好。缺点是如果多主机负载均衡,则会出现不连续的id,当然将递增区段设置为1也能保证连续的id,

但是效率会受到很大影响。实现关键源码如下：

缓存式ZK方案

import org.apache.curator.framework.CuratorFramework;

import org.apache.curator.framework.CuratorFrameworkFactory;

import org.apache.curator.framework.recipes.locks.InterProcessSemaphoreMutex;

import org.apache.curator.retry.ExponentialBackoffRetry;

import org.apache.zookeeper.CreateMode;

import org.apache.zookeeper.data.Stat;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.io.UnsupportedEncodingException;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

/\*\*

\* 根据开源项目mycat实现基于zookeeper的递增序列号

\* <p>

\* 只要配置好ZK地址和表名的如下属性

\* MINID 某线程当前区间内最小值

\* MAXID 某线程当前区间内最大值

\* CURID 某线程当前区间内当前值

\*

\* @author wangwanbin

\* @version 1.0

\* @time 2017/9/1

\*/

public class ZKCachedSequenceHandler extends SequenceHandler {

protected static final Logger LOGGER = LoggerFactory.getLogger(ZKCachedSequenceHandler.class);

private static final String KEY\_MIN\_NAME = ".MINID";// 1

private static final String KEY\_MAX\_NAME = ".MAXID";// 10000

private static final String KEY\_CUR\_NAME = ".CURID";// 888

private final static long PERIOD = 1000;//每次缓存的ID段数量

private static ZKCachedSequenceHandler instance = new ZKCachedSequenceHandler();

/\*\*

\* 私有化构造方法,单例模式

\*/

private ZKCachedSequenceHandler() {

}

/\*\*

\* 获取sequence工具对象的唯一方法

\*

\* @return

\*/

public static ZKCachedSequenceHandler getInstance() {

return instance;

}

private Map<String, Map<String, String>> tableParaValMap = null;

private CuratorFramework client;

private InterProcessSemaphoreMutex interProcessSemaphore = null;

public void loadZK() {

try {

this.client = CuratorFrameworkFactory.newClient(zkAddress, new ExponentialBackoffRetry(1000, 3));

this.client.start();

} catch (Exception e) {

LOGGER.error("Error caught while initializing ZK:" + e.getCause());

}

}

public Map<String, String> getParaValMap(String prefixName) {

if (tableParaValMap == null) {

try {

loadZK();

fetchNextPeriod(prefixName);

} catch (Exception e) {

LOGGER.error("Error caught while loding configuration within current thread:" + e.getCause());

}

}

Map<String, String> paraValMap = tableParaValMap.get(prefixName);

return paraValMap;

}

public Boolean fetchNextPeriod(String prefixName) {

try {

Stat stat = this.client.checkExists().forPath(PATH + "/" + prefixName + SEQ);

if (stat == null || (stat.getDataLength() == 0)) {

try {

client.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT)

.forPath(PATH + "/" + prefixName + SEQ, String.valueOf(0).getBytes());

} catch (Exception e) {

LOGGER.debug("Node exists! Maybe other instance is initializing!");

}

}

if (interProcessSemaphore == null) {

interProcessSemaphore = new InterProcessSemaphoreMutex(client, PATH + "/" + prefixName + SEQ);

}

interProcessSemaphore.acquire();

if (tableParaValMap == null) {

tableParaValMap = new ConcurrentHashMap<>();

}

Map<String, String> paraValMap = tableParaValMap.get(prefixName);

if (paraValMap == null) {

paraValMap = new ConcurrentHashMap<>();

tableParaValMap.put(prefixName, paraValMap);

}

long now = Long.parseLong(new String(client.getData().forPath(PATH + "/" + prefixName + SEQ)));

client.setData().forPath(PATH + "/" + prefixName + SEQ, ((now + PERIOD) + "").getBytes());

if (now == 1) {

paraValMap.put(prefixName + KEY\_MAX\_NAME, PERIOD + "");

paraValMap.put(prefixName + KEY\_MIN\_NAME, "1");

paraValMap.put(prefixName + KEY\_CUR\_NAME, "0");

} else {

paraValMap.put(prefixName + KEY\_MAX\_NAME, (now + PERIOD) + "");

paraValMap.put(prefixName + KEY\_MIN\_NAME, (now) + "");

paraValMap.put(prefixName + KEY\_CUR\_NAME, (now) + "");

}

} catch (Exception e) {

LOGGER.error("Error caught while updating period from ZK:" + e.getCause());

} finally {

try {

interProcessSemaphore.release();

} catch (Exception e) {

LOGGER.error("Error caught while realeasing distributed lock" + e.getCause());

}

}

return true;

}

public Boolean updateCURIDVal(String prefixName, Long val) {

Map<String, String> paraValMap = tableParaValMap.get(prefixName);

if (paraValMap == null) {

throw new IllegalStateException("ZKCachedSequenceHandler should be loaded first!");

}

paraValMap.put(prefixName + KEY\_CUR\_NAME, val + "");

return true;

}

/\*\*

\* 获取自增ID

\*

\* @param sequenceEnum

\* @return

\*/

@Override

public synchronized long nextId(SequenceEnum sequenceEnum) {

String prefixName = sequenceEnum.getCode();

Map<String, String> paraMap = this.getParaValMap(prefixName);

if (null == paraMap) {

throw new RuntimeException("fetch Param Values error.");

}

Long nextId = Long.parseLong(paraMap.get(prefixName + KEY\_CUR\_NAME)) + 1;

Long maxId = Long.parseLong(paraMap.get(prefixName + KEY\_MAX\_NAME));

if (nextId > maxId) {

fetchNextPeriod(prefixName);

return nextId(sequenceEnum);

}

updateCURIDVal(prefixName, nextId);

return nextId.longValue();

}

public static void main(String[] args) throws UnsupportedEncodingException {

long startTime = System.currentTimeMillis(); //获取开始时间

final ZKCachedSequenceHandler sequenceHandler = getInstance();

sequenceHandler.loadZK();

new Thread() {

public void run() {

long startTime2 = System.currentTimeMillis(); //获取开始时间

for (int i = 0; i < 5000; i++) {

System.out.println("线程1 " + sequenceHandler.nextId(SequenceEnum.ACCOUNT));

}

long endTime2 = System.currentTimeMillis(); //获取结束时间

System.out.println("程序运行时间1： " + (endTime2 - startTime2) + "ms");

}

}.start();

for (int i = 0; i < 5000; i++) {

System.out.println("线程2 " + sequenceHandler.nextId(SequenceEnum.ACCOUNT));

}

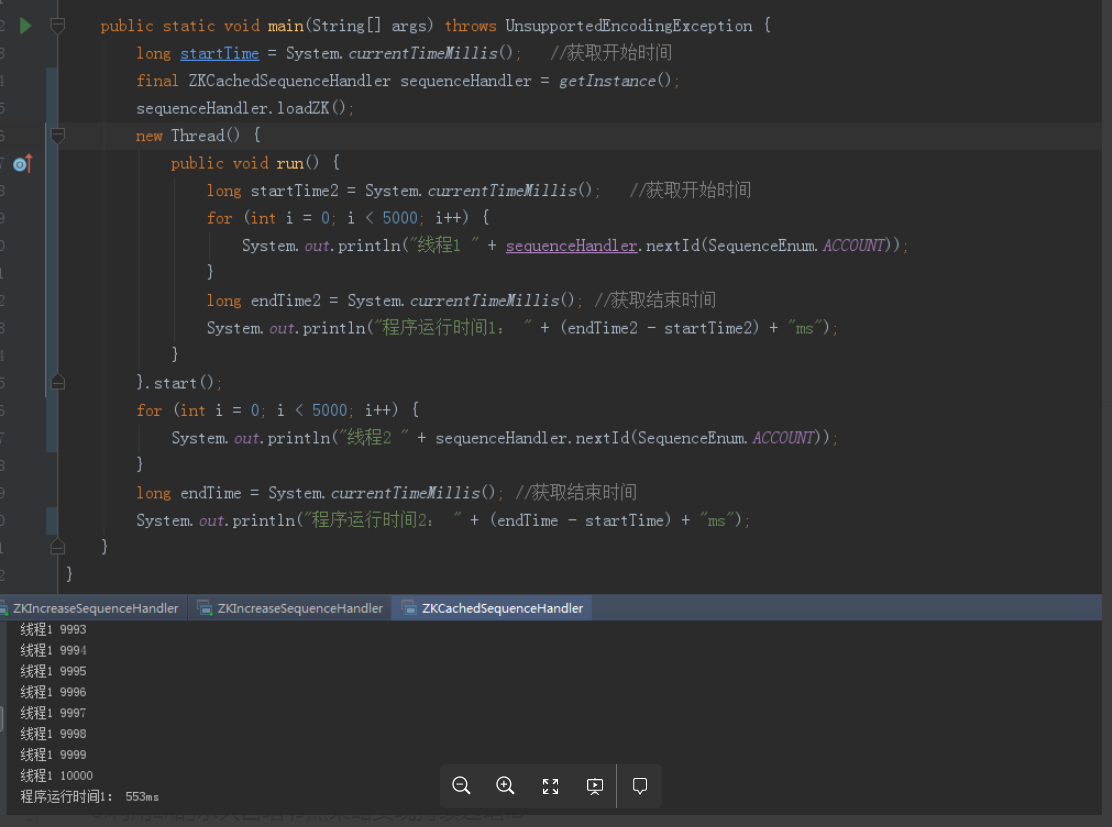
long endTime = System.currentTimeMillis(); //获取结束时间

System.out.println("程序运行时间2： " + (endTime - startTime) + "ms");

}

}

可以看到,由于不需要进行过多的网络消耗,缓存式的zk协调方案性能相当了得,生成10000个ID仅需553ms(两个线程耗时较长者) , 平均每个ID=0.05ms



3.利用zk的永久自增节点策略实现持续递增ID

使用zk的永久sequence策略创建节点,并获取返回值,然后删除前一个节点,这样既防止zk服务器存在过多的节点,又提高了效率;节点删除采用线程池来统一处理,提高响应速度

优点:能创建连续递增的ID,又能降低ZK消耗。关键实现代码如下：

zk自增

import org.apache.curator.framework.CuratorFramework;

import org.apache.curator.framework.CuratorFrameworkFactory;

import org.apache.curator.retry.ExponentialBackoffRetry;

import org.apache.zookeeper.CreateMode;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

/\*\*

\* 基于zk的永久型自增节点PERSISTENT\_SEQUENTIAL实现

\* 每次生成节点后会使用线程池执行删除节点任务,以减小zk的负担

\* Created by wangwanbin on 2017/9/5.

\*/

public class ZKIncreaseSequenceHandler extends SequenceHandler {

protected static final Logger LOGGER = LoggerFactory.getLogger(ZKCachedSequenceHandler.class);

private static ZKIncreaseSequenceHandler instance = new ZKIncreaseSequenceHandler();

private CuratorFramework client;

private ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

/\*\*

\* 私有化构造方法,单例模式

\*/

private ZKIncreaseSequenceHandler() {

loadZK();

}

private void loadZK() {

try {

this.client = CuratorFrameworkFactory.newClient(zkAddress, new ExponentialBackoffRetry(1000, 3));

this.client.start();

} catch (Exception e) {

LOGGER.error("Error caught while initializing ZK:" + e.getCause());

}

}

/\*\*

\* 获取sequence工具对象的唯一方法

\*

\* @return

\*/

public static ZKIncreaseSequenceHandler getInstance() {

return instance;

}

@Override

public long nextId(final SequenceEnum sequenceEnum) {

String result = createNode(sequenceEnum.getCode());

final String idstr = result.substring((PATH + "/" + sequenceEnum.getCode() + SEQ).length());

final long id = Long.parseLong(idstr);

final String format = "%0" + idstr.length() + "d";

//删除上一个节点

fixedThreadPool.execute(new Runnable() {

@Override

public void run() {

String preIdstr = String.format(format, id - 1);

String prePath = PATH + "/" + sequenceEnum.getCode() + SEQ + preIdstr;

try {

client.delete().forPath(prePath);

} catch (Exception e) {

LOGGER.error("delete preNode error", e);

}

}

});

return id;

}

private String createNode(String prefixName) {

try {

String result = client.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT\_SEQUENTIAL)

.forPath(PATH + "/" + prefixName + SEQ, String.valueOf(0).getBytes());

return result;

} catch (Exception e) {

throw new RuntimeException("create zookeeper node error", e);

}

}

public static void main(String[] args) {

long startTime = System.currentTimeMillis(); //获取开始时间

final ZKIncreaseSequenceHandler sequenceHandler = ZKIncreaseSequenceHandler.getInstance();

new Thread() {

public void run() {

long startTime2 = System.currentTimeMillis(); //获取开始时间

for (int i = 0; i < 5000; i++) {

System.out.println("线程1 " + sequenceHandler.nextId(SequenceEnum.ACCOUNT));

}

long endTime2 = System.currentTimeMillis(); //获取结束时间

System.out.println("程序运行时间2： " + (endTime2 - startTime2) + "ms");

}

}.start();

for (int i = 0; i < 5000; i++) {

System.out.println("线程2 " + sequenceHandler.nextId(SequenceEnum.ACCOUNT));

}

long endTime = System.currentTimeMillis(); //获取结束时间

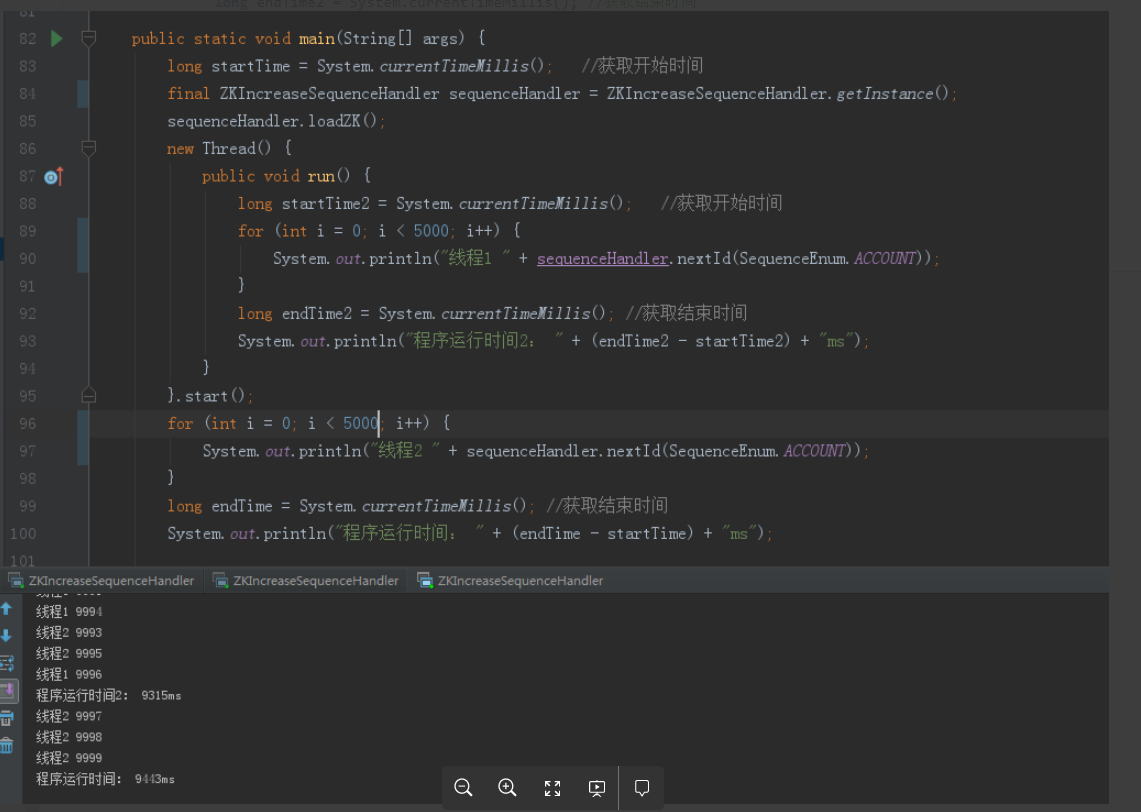
System.out.println("程序运行时间： " + (endTime - startTime) + "ms");

}

}

测试结果如下,生成10000个ID消耗=9443ms(两个线程耗时较长者), 平均每个ID=0.9ms

这还只是单zk连接的情况下,如果使用连接池来维护多个zk的链接,效率将成倍的提升



四、结语

分布式ID生成器的实现有很多种。目前各方案也都各有特点。我们可以根据业务的具体要求，选择实现合适的方案。