# 什么是多线程

多线程为了能够提高应用程序的运行效率，在一个进程中有多条不同的执行路径，同时并行执行，互不影响。

# 什么是线程安全

当多个线程同时共享，同一个**全局变量或静态变量**，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作是不会发生数据冲突问题。

## 解决办法

使用同步代码块或者Lock锁机制，保证在多个线程共享同一个变量只能有一个线程进行操作

# 什么是Java内存模型

共享内存模型指的就是Java内存模型(简称JMM)，**JMM决定一个线程对共享变量的写入时,能对另一个线程可见**。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：**线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本**。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

下面通过示意图来说明这两个步骤：   


如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

**总结：什么是Java内存模型：java内存模型简称jmm，定义了一个线程对另一个线程可见。共享变量存放在主内存中，每个线程都有自己的本地内存，当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题。**

# 分布式锁解决办法

# 传统方式生成订单号ID

## 业务场景

在分布式情况，生成全局订单号ID

## 生成订单号方案

1. 使用时间戳
2. 使用UUID
3. 推特 (Twitter) 的 Snowflake 算法——用于生成唯一 ID

## 生成订单类

|  |
| --- |
| //生成订单类  **public** **class** OrderNumGenerator {  //全局订单id  **public** **static** **int** *count* = 0;  **public** String getNumber() {  **try** {  Thread.*sleep*(200);  } **catch** (Exception e) {  }  SimpleDateFormat simpt = **new** SimpleDateFormat("yyyy-MM-dd-HH-mm-ss");  **return** simpt.format(**new** Date()) + "-" + ++*count*;  }  } |

## 使用多线程情况模拟生成订单号

|  |
| --- |
| //使用多线程模拟生成订单号  **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  **for** (**int** i = 0; i < 100; i++) {  **new** Thread(**new** OrderService()).start();  }  }  } |

## 多线程生成订单号，线程安全问题解决

使用synchronized或者loca锁

### Synchronized同步代码块方式

|  |
| --- |
| //使用多线程模拟生成订单号  **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  **synchronized** (**this**) {  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  }  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  OrderService orderService = **new** OrderService();  **for** (**int** i = 0; i < 100; i++) {  **new** Thread(orderService).start();  }  }  } |

### Lock锁方式

|  |
| --- |
| **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  // 使用lock锁  **private** java.util.concurrent.locks.Lock lock = **new** ReentrantLock();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  **try** {  // synchronized (this) {  lock.lock();  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  // }  } **catch** (Exception e) {  } **finally** {  lock.unlock();  }  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  OrderService orderService = **new** OrderService();  **for** (**int** i = 0; i < 100; i++) {  **new** Thread(orderService).start();  }  }  } |

# 分布式场景下生成订单ID

## 业务场景

在分布式情况，生成全局订单号ID

## 产生问题

在分布式(集群)环境下，每台JVM不能实现同步，在分布式场景下使用时间戳生成订单号可能会重复

## 分布式情况下，怎么解决订单号生成不重复

1. 使用分布式锁
2. 提前生成好，订单号，存放在redis取。获取订单号，直接从redis中取。

## 使用分布式锁生成订单号技术

1.使用数据库实现分布式锁

缺点:性能差、线程出现异常时，容易出现死锁

2.使用redis实现分布式锁

缺点:锁的失效时间难控制、容易产生死锁、非阻塞式、不可重入

**3.使用zookeeper实现分布式锁**

**实现相对简单、可靠性强、使用临时节点，失效时间容易控制**

## 什么是分布式锁

分布式锁一般用在分布式系统或者多个应用中，用来控制同一任务是否执行或者任务的执行顺序。在项目中，部署了多个tomcat应用，在执行定时任务时就会遇到同一任务可能执行多次的情况，我们可以借助分布式锁，保证在同一时间只有一个tomcat应用执行了定时任务

## 使用Zookeeper实现分布式锁

### Zookeeper实现分布式锁原理

使用zookeeper创建临时序列节点来实现分布式锁，适用于顺序执行的程序，大体思路就是创建临时序列节点，找出最小的序列节点，获取分布式锁，程序执行完成之后此序列节点消失，通过watch来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理，依次类推……

### Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.10</version>  </dependency>  </dependencies> |

### 创建Lock接口

|  |
| --- |
| **public** **interface** Lock {  //获取到锁的资源  **public** **void** getLock();  // 释放锁  **public** **void** unLock();  } |

### 创建ZookeeperAbstractLock抽象类

|  |
| --- |
| **//将重复代码写入子类中..**  **public abstract class ZookeeperAbstractLock implements Lock {**  **// zk连接地址**  **private static final String *CONNECTSTRING* = "127.0.0.1:2181";**  **// 创建zk连接**  **protected ZkClient zkClient = new ZkClient(*CONNECTSTRING*);**  **protected static final String *PATH* = "/lock";**  **public void getLock() {**  **if (tryLock()) {**  **System.*out*.println("##获取lock锁的资源####");**  **} else {**  **// 等待**  **waitLock();**  **// 重新获取锁资源**  **getLock();**  **}**  **}**  **// 获取锁资源**  **abstract boolean tryLock();**  **// 等待**  **abstract void waitLock();**  **public void unLock() {**  **if (zkClient != null) {**  **zkClient.close();**  **System.*out*.println("释放锁资源...");**  **}**  **}**  **}** |

### ZookeeperDistrbuteLock类

|  |
| --- |
| public class ZookeeperDistrbuteLock extends ZookeeperAbstractLock {  private CountDownLatch countDownLatch = null;  @Override  boolean tryLock() {  try {  zkClient.createEphemeral(*PATH*);  return true;  } catch (Exception e) {  // e.printStackTrace();  return false;  }  }  @Override  void waitLock() {  IZkDataListener izkDataListener = new IZkDataListener() {  public void handleDataDeleted(String path) throws Exception {  // 唤醒被等待的线程  if (countDownLatch != null) {  countDownLatch.countDown();  }  }  public void handleDataChange(String path, Object data) throws Exception {  }  };  // 注册事件  zkClient.subscribeDataChanges(*PATH*, izkDataListener);  if (zkClient.exists(*PATH*)) {  countDownLatch = new CountDownLatch(1);  try {  countDownLatch.await();  } catch (Exception e) {  e.printStackTrace();  }  }  // 删除监听  zkClient.unsubscribeDataChanges(*PATH*, izkDataListener);  }  } |

### 使用Zookeeper锁运行效果

|  |
| --- |
| **public** **class** OrderService **implements** Runnable {  **private** OrderNumGenerator orderNumGenerator = **new** OrderNumGenerator();  // 使用lock锁  // private java.util.concurrent.locks.Lock lock = new ReentrantLock();  **private** Lock lock = **new** ZookeeperDistrbuteLock();  **public** **void** run() {  getNumber();  }  **public** **void** getNumber() {  **try** {  lock.getLock();  String number = orderNumGenerator.getNumber();  System.***out***.println(Thread.*currentThread*().getName() + ",生成订单ID:" + number);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  lock.unLock();  }  }  **public** **static** **void** main(String[] args) {  System.***out***.println("####生成唯一订单号###");  // OrderService orderService = new OrderService();  **for** (**int** i = 0; i < 100; i++) {  **new** Thread( **new** OrderService()).start();  }  }  } |

# 使用Zookeeper实现负载均衡原理

## 思路

使用Zookeeper实现负载均衡原理，服务器端将启动的服务注册到，zk注册中心上，采用临时节点。客户端从zk节点上获取最新服务节点信息，本地使用负载均衡算法，随机分配服务器。

## 创建项目工程

### Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.8</version>  </dependency>  </dependencies> |

### 创建Server服务端

### ZkServerScoekt服务

|  |
| --- |
| //##ServerScoekt服务端  **public** **class** ZkServerScoekt **implements** Runnable {  **private** **int** port = 18080;  **public** **static** **void** main(String[] args) **throws** IOException {  **int** port = 18080;  ZkServerScoekt server = **new** ZkServerScoekt(port);  Thread thread = **new** Thread(server);  thread.start();  }  **public** ZkServerScoekt(**int** port) {  **this**.port = port;  }  **public** **void** run() {  ServerSocket serverSocket = **null**;  **try** {  serverSocket = **new** ServerSocket(port);  System.***out***.println("Server start port:" + port);  Socket socket = **null**;  **while** (**true**) {  socket = serverSocket.accept();  **new** Thread(**new** ServerHandler(socket)).start();  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if** (serverSocket != **null**) {  serverSocket.close();  }  } **catch** (Exception e2) {  }  }  }  } |

### ZkServerClient

|  |
| --- |
| **public** **class** ZkServerClient {  **public** **static** List<String> *listServer* = **new** ArrayList<String>();  **public** **static** **void** main(String[] args) {  *initServer*();  ZkServerClient client= **new** ZkServerClient();  BufferedReader console = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **while** (**true**) {  String name;  **try** {  name = console.readLine();  **if** ("exit".equals(name)) {  System.*exit*(0);  }  client.send(name);  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  // 注册所有server  **public** **static** **void** initServer() {  *listServer*.clear();  *listServer*.add("127.0.0.1:18080");  }  // 获取当前server信息  **public** **static** String getServer() {  **return** *listServer*.get(0);  }    **public** **void** send(String name) {  String server = ZkServerClient.*getServer*();  String[] cfg = server.split(":");  Socket socket = **null**;  BufferedReader in = **null**;  PrintWriter out = **null**;  **try** {  socket = **new** Socket(cfg[0], Integer.*parseInt*(cfg[1]));  in = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));  out = **new** PrintWriter(socket.getOutputStream(), **true**);  out.println(name);  **while** (**true**) {  String resp = in.readLine();  **if** (resp == **null**)  **break**;  **else** **if** (resp.length() > 0) {  System.***out***.println("Receive : " + resp);  **break**;  }  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (out != **null**) {  out.close();  }  **if** (in != **null**) {  **try** {  in.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  } |

### 改造ZkServerScoekt

|  |
| --- |
| **public** **class** ZkServerScoekt **implements** Runnable {  **private** **static** **int** *port* = 18081;  **public** **static** **void** main(String[] args) **throws** IOException {  ZkServerScoekt server = **new** ZkServerScoekt(*port*);  Thread thread = **new** Thread(server);  thread.start();  }  **public** ZkServerScoekt(**int** port) {  **this**.*port* = port;  }  **public** **void** regServer() {  // 向ZooKeeper注册当前服务器  ZkClient client = **new** ZkClient("127.0.0.1:2181", 60000, 1000);  String path = "/test/server" + *port*;  **if** (client.exists(path))  client.delete(path);  client.createEphemeral(path, "127.0.0.1:" + *port*);  }  **public** **void** run() {  ServerSocket serverSocket = **null**;  **try** {  serverSocket = **new** ServerSocket(*port*);  regServer();  System.***out***.println("Server start port:" + *port*);  Socket socket = **null**;  **while** (**true**) {  socket = serverSocket.accept();  **new** Thread(**new** ServerHandler(socket)).start();  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if** (serverSocket != **null**) {  serverSocket.close();  }  } **catch** (Exception e2) {  }  }  }  } |

### 改造ZkServerScoekt

|  |
| --- |
| **public** **class** ZkServerClient {  **public** **static** List<String> *listServer* = **new** ArrayList<String>();  **public** **static** **void** main(String[] args) {  *initServer*();  ZkServerClient client = **new** ZkServerClient();  BufferedReader console = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **while** (**true**) {  String name;  **try** {  name = console.readLine();  **if** ("exit".equals(name)) {  System.*exit*(0);  }  client.send(name);  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  // 注册所有server  **public** **static** **void** initServer() {  **final** String path = "/test";  **final** ZkClient zkClient = **new** ZkClient("127.0.0.1:2181", 60000, 1000);  List<String> children = zkClient.getChildren(path);  *listServer*.clear();  **for** (String p : children) {  *listServer*.add((String) zkClient.readData(path + "/" + p));  }  // 订阅节点变化事件  zkClient.subscribeChildChanges("/test", **new** IZkChildListener() {  **public** **void** handleChildChange(String parentPath, List<String> currentChilds) **throws** Exception {  *listServer*.clear();  **for** (String p : currentChilds) {  *listServer*.add((String) zkClient.readData(path + "/" + p));  }  System.***out***.println("####handleChildChange()####listServer:" + *listServer*.toString());  }  });  }  // 请求次数  **private** **static** **int** *count* = 1;  // 服务数量  **private** **static** **int** *serverCount*=2;  // 获取当前server信息  **public** **static** String getServer() {  String serverName = *listServer*.get(*count*%*serverCount*);  ++*count*;  **return** serverName;  }  **public** **void** send(String name) {  String server = ZkServerClient.*getServer*();  String[] cfg = server.split(":");  Socket socket = **null**;  BufferedReader in = **null**;  PrintWriter out = **null**;  **try** {  socket = **new** Socket(cfg[0], Integer.*parseInt*(cfg[1]));  in = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));  out = **new** PrintWriter(socket.getOutputStream(), **true**);  out.println(name);  **while** (**true**) {  String resp = in.readLine();  **if** (resp == **null**)  **break**;  **else** **if** (resp.length() > 0) {  System.***out***.println("Receive : " + resp);  **break**;  }  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (out != **null**) {  out.close();  }  **if** (in != **null**) {  **try** {  in.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  } |

# 使用Zookeeper实现选举策略

## 场景

  有一个向外提供的服务，服务必须7\*24小时提供服务，不能有单点故障。所以采用集群的方式，采用master、slave的结构。一台主机多台备机。主机向外提供服务，备机负责监听主机的状态，一旦主机宕机，备机要迅速接代主机继续向外提供服务。从备机选择一台作为主机，就是master选举。

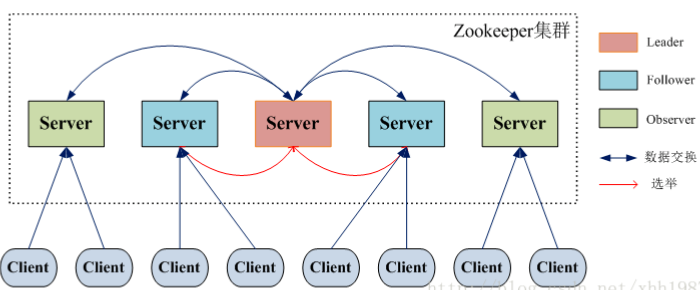
## 原理分析

 右边三台主机会尝试创建master节点，谁创建成功了，就是master，向外提供。其他两台就是slave。

所有slave必须关注master的删除事件（临时节点，如果服务器宕机了，Zookeeper会自动把master节点删除）。如果master宕机了，会进行新一轮的master选举。本次我们主要关注master选举，服务注册、发现先不讨论。

# 使用Zookeeper原理

» 领导者（leader），负责进行投票的发起和决议，更新系统状态  
　　» 学习者（learner），包括跟随者（follower）和观察者（observer），follower用于接受客户端请求并想客户端返回结果，在选主过程中参与投票  
　　» Observer可以接受客户端连接，将写请求转发给leader，但observer不参加投票过程，只同步leader的状态，observer的目的是为了扩展系统，提高读取速度  
　　» 客户端（client），请求发起方





　　• Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协  
　　   议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者  
　　　崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后  
　　  ，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

　　• 为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（  
　　　proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识  
　　   leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的  
　　　统治时期。低32位用于递增计数。  
　　• 每个Server在工作过程中有三种状态：  
　　　　LOOKING：当前Server不知道leader是谁，正在搜寻  
　　　　LEADING：当前Server即为选举出来的leader  
　　　　FOLLOWING：leader已经选举出来，当前Server与之同步