# 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();  }  }  } |