文档修订历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **作者** | **工作描述** | **修订历史** | **修改日期** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[文档修订历史 1](#_Toc522787485)

[CBSS多线程编程学习摘录 3](#_Toc522787486)

[1 线程框架 3](#_Toc522787487)

[1.1 程序代码 3](#_Toc522787488)

[1.1.1 Parallel.java 3](#_Toc522787489)

[1.2 Ctrl+2 7](#_Toc522787490)

[1.2.1 Ctrl+3 7](#_Toc522787491)

[附录 7](#_Toc522787492)

[附录一、Java volatile关键字 7](#_Toc522787493)

[1.2.2 基本概念 7](#_Toc522787494)

[1.2.3 Volatile原理 10](#_Toc522787495)

CBSS多线程编程学习摘录

# 线程框架

负责将一批可以并行处理的任务，切分成多个子任务，并行处理

## 程序代码

### Parallel.java

|  |
| --- |
| /\*\*  \* Created by qiupq on 2017/7/29.  \* 负责将一批可以并行处理的任务，切分成多个子任务，并行处理  \*/  public class Parallel {  /\*\*  \* 将dataList切分threadNum份，启动threadNum个线程，送给worker加工  \*/  public static <DATA> void parallelDo(int threadNum, List<DATA> dataList, WorkerBase<DATA> worker) {  List<List<DATA>> sliceDataList = new ArrayList<>();  int nData = dataList.size();  if (nData == 0) {  Logger logger = LoggerFactory.getLogger(Parallel.class);  logger.warn("Parallel: NO DATA to process. task finished immediately");  return;  }  threadNum = threadNum > nData ? nData : threadNum;  for (int i = 0; i < threadNum; i++) {  sliceDataList.add(new ArrayList<>());  }  //切分数据  for (int i = 0; i < dataList.size(); i++) {  int j = i % threadNum;  sliceDataList.get(j).add(dataList.get(i));  }  parallelDo(sliceDataList, worker);  }  /\*\*  \* 调用者切分好任务，逐个方法负责并行启动  \*  \* @param sliceDataList 已经切分好的任务列表的列表，每个任务列表会对应启动一个线程，  \* 并将任务列表中的每个元素送给该线程逐个处理  \* @param worker 线程核心  \* @param <DATA> 送给线程核心的工作元素数据  \*/  public static <DATA> void parallelDo(List<List<DATA>> sliceDataList, WorkerBase<DATA> worker) {  List<WorkerThread<DATA>> myThreads = new ArrayList<>();  List<Thread> threads = new ArrayList<>();  int threadNum = sliceDataList.size();  Logger logger = LoggerFactory.getLogger(Parallel.class);  logger.warn("Parallel: task start, threads=" + threadNum);  //准备线程  for (int i = 0; i < threadNum; i++) {  WorkerThread<DATA> workerThread = new WorkerThread<>(i);  myThreads.add(workerThread);  threads.add(new Thread(workerThread));  }  //为这组线程准备共用的Stop标志  Stop stop = new Stop();  //逐个线程分发数据并启动  for (int j = 0; j < threadNum; j++) {  // System.out.println("Parallel: thread id=" + j + ",tasks=" + sliceDataList.get(j).size());  myThreads.get(j).setData(sliceDataList.get(j));  myThreads.get(j).setWorker(worker);  myThreads.get(j).setStop(stop);  threads.get(j).start();  }  //等待完成  for (Thread thread : threads) {  try {  thread.join();  } catch (InterruptedException e) {  }  }  if (stop.stopNow) {  logger.error("Parallel: task finished failure.");  throw new BizException("并行任务执行失败。");  } else {  logger.warn("Parallel: task finished successfully.");  }  }  }  class Stop {  volatile boolean stopNow = false;  }  class WorkerThread<DATA> implements Runnable {  private List<DATA> data;  private WorkerBase<DATA> worker;  private Stop stop;  private int threadId;  void setData(List<DATA> data) {  this.data = data;  }  void setWorker(WorkerBase<DATA> worker) {  this.worker = worker;  }  public void setStop(Stop stop) {  this.stop = stop;  }  WorkerThread(int threadId) {  this.threadId = threadId;  }  @Override  public void run() {  try {  int i = 0;  int n = data.size();  for (DATA data : this.data) {  if (stop.stopNow) {  //Logger logger = LoggerFactory.getLogger(Parallel.class);  //logger.error("Parallel: threadId=" + threadId + ",task be stoped");  break;  }  i++;  long t1 = System.currentTimeMillis();  //work就是worker  worker.work(data);  long t2 = System.currentTimeMillis();  // System.out.println("threadId=" + threadId + ",progress=" + i + "/" + n + "=" + (i \* 1000 / n / 10.0) + "%,this step use:" + (t2 - t1) + "ms");  }  } catch (Throwable e) {  if (e instanceof RateException) {  ((RateException) e).printErrAndStack();  } else {  e.printStackTrace();  }  Logger logger = LoggerFactory.getLogger(Parallel.class);  logger.error("Parallel: threadId=" + threadId + ",catch exception:"  + RateException.formatMsg(e));  stop.stopNow = true;//别的线程看见这个标志会主动退出  }  }  } |

### WorkerBase.java

|  |
| --- |
| /\*\*  \* Created by qiupq on 2017/7/29.  \* 并行工作时，每个子任务的基类  \*/  public interface WorkerBase<DATA> {  //每个子任务在此处执行，执行时只需要专注当前id如何处理  void work(DATA data);  } |

# 应用示例

## 合帐应用

### 窗口程序（BillServiceImpl.java）

@Service

public class BillServiceImpl

extends SingleDomainServiceSupport<Bill> implements BillService {

### 主干方法acctItemAddToBill()：

|  |
| --- |
| @Override  public int acctItemAddToBill() {  String progName = this.getClass().getSimpleName();  String execMsg = "";  execMsg = "合账开始";  //把无需合账的更新为已合账  acctBackLogService.doLog(progName, AcctConstant.EXEC\_START, "开始把无需合账的更新为已合账...", null);  List<String> acctItemIds = acctItemMapper.queryNoNeedTotalBill();  if (acctItemIds.size() > 0) {  acctItemMapper.updateNoNeedTotalBill(acctItemIds);  }  acctBackLogService.doLog(progName, AcctConstant.EXEC\_INFO, "完成把无需合账的更新为已合账...", null);  //获取需要合帐的合同号  List<String> agreementIds = acctItemMapper.queryNoBillAgreementIds();    //赋执行方法：acctItemAddToBillByAgreementId  WorkerBase<String> worker = this::acctItemAddToBillByAgreementId;    //开启三个线程，同时传入需要合帐的合同列表，并传入合帐的执行方法名，调用了Parallel.java里的方法“public static <DATA> void parallelDo(int threadNum, List<DATA> dataList, WorkerBase<DATA> worker)”  Parallel.parallelDo(3, agreementIds, worker);  return 0;  } |

### 执行方法：acctItemAddToBillByAgreementId()

|  |
| --- |
| //传参是字符型，与上面的Parallel.parallelDo(3, agreementIds, worker)里的“agreementIds“LIST的类型一致  @Override  public int acctItemAddToBillByAgreementId(String agreementId) {  return this.acctItemAddToBillByAgreementId(agreementId,AcctConstant.CSC\_TYPE\_NORMAL);  }  private List<String> queryBillsByAgreementId(String agreementId) {  Condition condition = new Condition(Bill.class);  Example.Criteria criteria = condition.createCriteria();  criteria.andEqualTo("agreementId", agreementId);  criteria.andEqualTo("billState", AcctConstant.BILL\_STATE\_BSN);  List<Bill> bills = queryByCondition(condition);  List<String> billIds = new LinkedList<>();  if (null != bills && bills.size() > 0) {  for (Bill bill : bills) {  billIds.add(bill.getBillId());  }  }  return billIds;  } |

# 附录

## 附录一、Java volatile关键字

转载：https://www.cnblogs.com/zhengbin/p/5654805.html

参考：<https://www.cnblogs.com/shoshana-kong/p/9066888.html>

### 基本概念

先补充一下概念：Java 内存模型中的可见性、原子性和有序性。

#### 1.1可见性：

可见性是一种复杂的属性，因为可见性中的错误总是会违背我们的直觉。通常，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果。另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就是这个操作同样存在线程安全问题。

在 Java 中 volatile、synchronized 和 final 实现可见性。

#### 1.2原子性：

原子是世界上的最小单位，具有不可分割性。比如 a=0；（a非long和double类型） 这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++； 这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

#### 1.3有序性：

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

下面内容摘录自《Java Concurrency in Practice》：

下面一段代码在多线程环境下，将存在问题。

|  |
| --- |
| 1 /\*\*  2 \* @author zhengbinMac  3 \*/  4 public class NoVisibility {  5 private static boolean ready;  6 private static int number;  7 private static class ReaderThread extends Thread {  8 @Override  9 public void run() {  10 while(!ready) {  11 Thread.yield();  12 }  13 System.out.println(number);  14 }  15 }  16 public static void main(String[] args) {  17 new ReaderThread().start();  18 number = 42;  19 ready = true;  20 }  21 } |

NoVisibility可能会持续循环下去，因为读线程可能永远都看不到ready的值。甚至NoVisibility可能会输出0，因为读线程可能看到了写入ready的值，但却没有看到之后写入number的值，这种现象被称为“重排序”。只要在某个线程中无法检测到重排序情况（即使在其他线程中可以明显地看到该线程中的重排序），那么就无法确保线程中的操作将按照程序中指定的顺序来执行。当主线程首先写入number，然后在没有同步的情况下写入ready，那么读线程看到的顺序可能与写入的顺序完全相反。

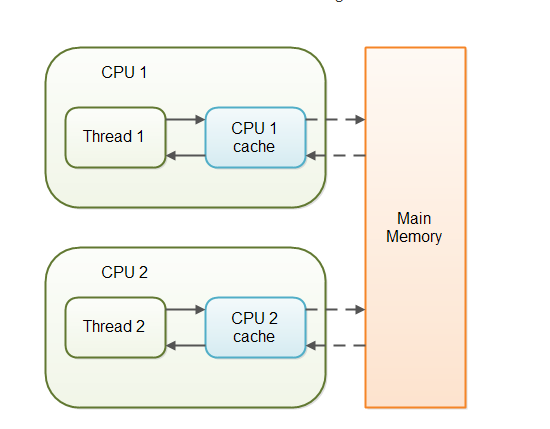
在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。在缺乏足够同步的多线程程序中，要想对内存操作的执行春旭进行判断，无法得到正确的结论。

这个看上去像是一个失败的设计，但却能使JVM充分地利用现代多核处理器的强大性能。例如，在缺少同步的情况下，Java内存模型允许编译器对操作顺序进行重排序，并将数值缓存在寄存器中。此外，它还允许CPU对操作顺序进行重排序，并将数值缓存在处理器特定的缓存中。

### Volatile原理

ava语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。



当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。

而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

#### 2.1当一个变量定义为 volatile 之后，将具备两种特性：

1.保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存（详见：Java内存模型）来完成。

2.禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理）。

#### 2.2volatile 性能：

volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。