Java多线程并发编程实战之百万级数据计算性能优化实战

# 出自:

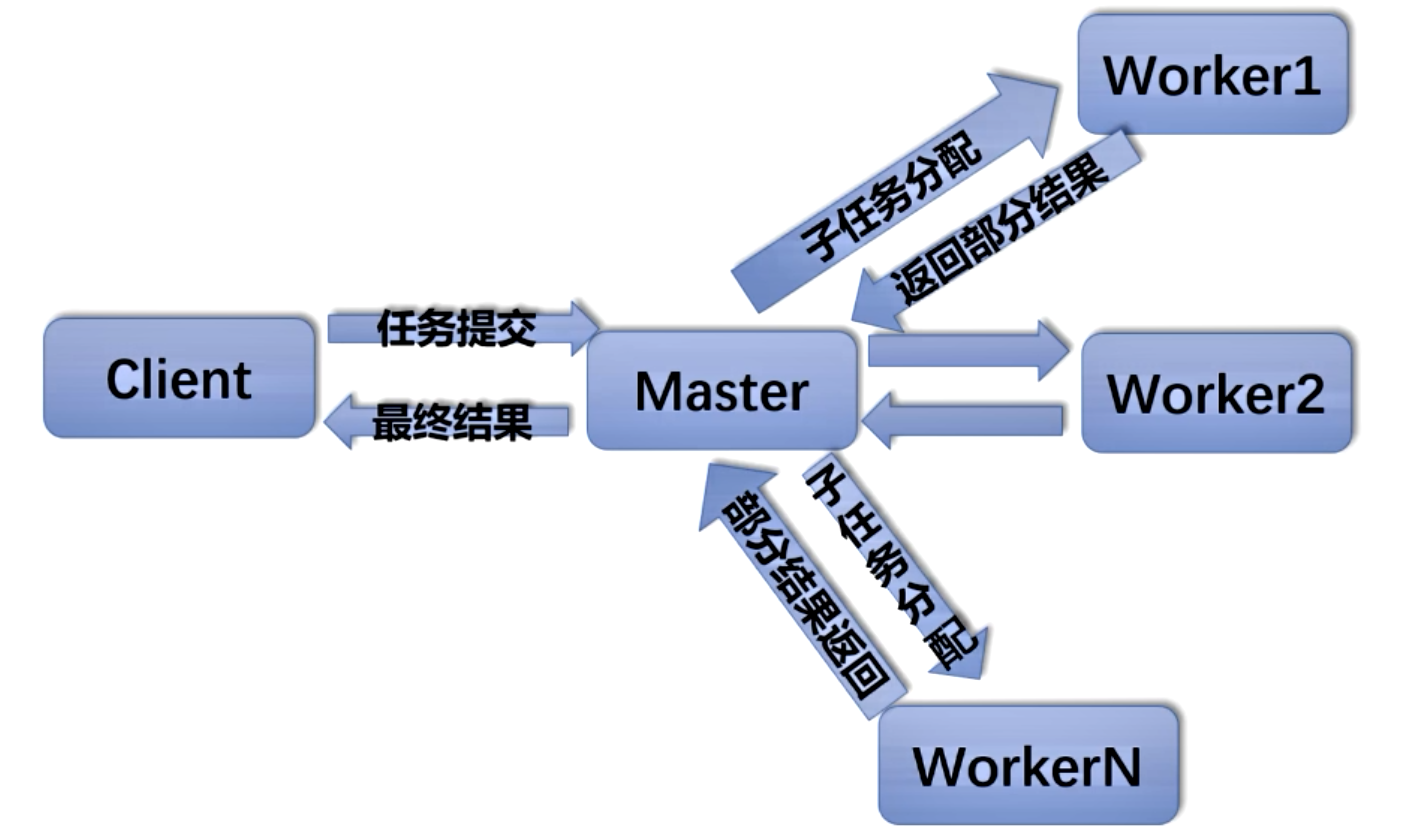
出自 <腾讯课堂 700多分钟干货实战Java多线程高并发高性能实战全集> , 我学习完了之后, 我给 老师在课上说的话做了个笔记,以及视频的内容,还有代码敲了一遍,然后添加了一些注释,把执行结果也整理了一下, 做了个笔记

# 案例背景

某应用程序有100万条数据,且每条数据在计算前的业务校验逻辑平均耗时5毫秒,请运用多线程高并发变成等相关的基础知识,实现1分钟以内,完成计算100万条数据的平方和.

# Master-Worker设计模式

## 介绍



1.Client发起任务给Master

2.Master将部分任务分配给Worker

3.Worker处理完了之后将结果返回给Master

4.所有Worker处理完了之后,Master将结果累加之后返回给Client

## 使用场景

可以分解大的任务并行化处理

## 角色

Master线程: 分配任务,合并Worker线程处理的结果

Worker线程: 处理具体的一个任务

## 模式优点

串行任务并行化处理,提高效率

# 头脑风暴

## Master-Worker设计模式是如何提高并发处理能力的?

答: 多个Worker线程并行的处理,处理完了返回给Master线程合并结果, 这样并发能力就变强了.

## 线程状态类有哪些应用?

用于线程状态判断的,比如说Master线程会判断Worker线程的任务是否结束,如果Worker线程结束了,Master就会将这个结束的线程的结果拿出来进行合并.

## 1.百万级别数据计算的特点是什么?

1.并发量很高,是百万级别计算,性能比较高,

2.要求性能高,要求一分钟内计算出来百万数据的平方和.

# 测试结果

100工作线程的时候测试:

当前工作线程有: 100 个 ,执行计算结果为:333333833333500000 耗时: 60061毫秒
  
当前工作线程有: 100 个 ,执行计算结果为:333333833333500000 耗时: 60635毫秒

200工作线程的时候测试:

当前工作线程有: 200 个 ,执行计算结果为:333333833333500000 耗时: 31753毫秒

300工作线程的时候测试:

当前工作线程有: 300 个 ,执行计算结果为:333333833333500000 耗时: 22459毫秒

500工作线程的时候测试:

当前工作线程有: 500 个 ,执行计算结果为:333333833333500000 耗时: 16198毫秒

1000工作线程的时候测试:

当前工作线程有: 1000 个 ,执行计算结果为:333333833333500000 耗时: 17717毫秒  
当前工作线程有: 1000 个 ,执行计算结果为:333333833333500000 耗时: 17370毫秒

2000工作线程的时候测试:

当前工作线程有: 2000 个 ,执行计算结果为:333333833333500000 耗时: 18081毫秒  
  
当前工作线程有: 2000 个 ,执行计算结果为:333333833333500000 耗时: 18968毫秒

3000工作线程的时候测试:

当前工作线程有: 3000 个 ,执行计算结果为:333333833333500000 耗时: 18705毫秒
  
当前工作线程有: 3000 个 ,执行计算结果为:333333833333500000 耗时: 19129毫秒

5000工作线程的时候测试:

电脑直接卡死,程序没跑出来...我直接手动给关闭了.

# 代码

## 操作说明

执行MultiThreadCompute这个类的testCompute测试类

## Master

package com.yrxy.thread.case5;
  
   
import java.util.HashMap;
  
import java.util.Map;
  
import java.util.Queue;
  
import java.util.concurrent.ConcurrentHashMap;
  
import java.util.concurrent.ConcurrentLinkedQueue;
  
   
/\*\*
  
 \* Master是协调用的,给计算任务分给不同的worker线程来处理.
  
 \*/
  
public class Master {
  
 // 放任务的队列
  
 protected Queue<Object> workQueue = new ConcurrentLinkedQueue<Object>();
  
 //装载Worker线程
  
 protected Map<String, Thread> workerThreadMap = new HashMap<String, Thread>();
  
 //每个worker计算结果放到这个map里面
  
   
 protected Map<String, Object> resultMap = new ConcurrentHashMap<String, Object>();
  
   
 /\*\*
  
 \* 构造方法
  
 \*
  
 \* @param worker 工作线程
  
 \* @param countWorker 工作线程数量
  
 \*/
  
 public Master(Worker worker, int countWorker) {
  
 // 设置工作队列
  
 worker.setWorkQueue(workQueue);
  
 // 设置 存放计算结果的Map
  
 worker.setResultMap(resultMap);
  
   
 for (int i = 0; i < countWorker; i++) {
  
 // 装载Worker线程 ,!! 需要注意的是,这里如果工作线程设置的过大的话,这里装在Worker线程会比较多
  
 // 如果你电脑配置不高的话,这里可能会消耗很多时间去实例化线程,并且装到workerThreadMap 里面
  
 // 所以设置工作线程数量的时候要量力而行.
  
 workerThreadMap.put(Integer.toString(i), new Thread(worker, Integer.toString(i)));
  
 }
  
 }
  
   
 /\*\*
  
 \* 会判断每一个线程状态是否有结束
  
 \*
  
 \* @return 没结束就返回false, 结束了就返回true
  
 \*/
  
 public boolean isComplete() {
  
 for (Map.Entry<String, Thread> entry : workerThreadMap.entrySet()) {
  
 // Thread.State.TERMINATED : 终止线程的线程状态。线程已完成执行
  
   
 if (entry.getValue().getState() != Thread.State.TERMINATED) {
  
 // 没结束就返回false
  
   
 return false;
  
 }
  
 }
  
 // 结束了就返回true
  
 return true;
  
 }
  
   
 //提交
  
 public void submit(Object job) {
  
 workQueue.add(job);
  
 }
  
   
   
 public Map<String, Object> getResultMap() {
  
 return resultMap;
  
 }
  
   
 //发起执行
  
 public void execute() {
  
 for (Map.Entry<String, Thread> entry : workerThreadMap.entrySet()) {
  
 entry.getValue().start();
  
 }
  
 }
  
}

## Worker

package com.yrxy.thread.case5;
  
   
import java.util.Map;
  
import java.util.Queue;
  
   
   
public class Worker implements Runnable {
  
 /\*\*
  
 \* 工作队列
  
 \*/
  
 protected Queue<Object> workQueue;
  
 /\*\*
  
 \* 存放计算结果的Map
  
 \*/
  
 protected Map<String, Object> resultMap;
  
   
 public void setWorkQueue(Queue<Object> workQueue) {
  
 this.workQueue = workQueue;
  
 }
  
   
 public void setResultMap(Map<String, Object> resultMap) {
  
 this.resultMap = resultMap;
  
 }
  
   
 /\*\*
  
 \* 计算相关的逻辑 ,这里子类会实现
  
 \*/
  
 public Object handle(Object input) {
  
 return input;
  
 }
  
   
 /\*\*
  
 \* 从队列里面拉取内容,计算相关的逻辑,把计算的结果放到resultMap里面去.
  
 \*/
  
 @Override
  
 public void run() {
  
 while (true) {
  
 //拉取内容
  
 Object input = workQueue.poll();
  
 if (input == null) {
  
 break;
  
 }
  
 //执行计算逻辑
  
 Object result = handle(input);
  
   
 // 计算的逻辑结果放到这个map里面去.
  
 // key就是hashCode,value就是计算的结果
  
 resultMap.put(Integer.toString(input.hashCode()), result);
  
 }
  
 }
  
}

## ComputeWorker

package com.yrxy.thread.case5;
  
   
/\*\*
  
 \* Worker主要功能类
  
 \*/
  
public class ComputeWorker extends Worker {
  
   
 /\*\*
  
 \* 模拟校验逻辑和计算逻辑
  
 \*
  
 \* @param input 计算的内容
  
 \* @return 计算好的平方和
  
 \*/
  
 @Override
  
 public Object handle(Object input) {
  
 try {
  
 //模拟执行校验逻辑
  
 System.out.println("我开始校验了");
  
 Thread.sleep(5); // 这里模拟校验耗时五毫秒
  
 System.out.println("我校验完了");
  
 } catch (InterruptedException e) {
  
   
 e.printStackTrace();
  
 }
  
 Long i = (Long) input;
  
 return i \* i; // 计算平方和
  
 }
  
}

## MultiThreadCompute

package com.yrxy.thread.case5;
  
   
import org.junit.Test;
  
   
import java.util.Map;
  
import java.util.Set;
  
   
public class MultiThreadCompute {
  
   
 @Test
  
 public void testCompute() {
  
 Long start = System.currentTimeMillis();
  
 // 初始化Master,初始化100个Worker
  
   
 int countWorker = 500;
  
 Master master = new Master(new ComputeWorker(), countWorker);
  
 //for循环提交任务,计算100万条数据
  
 for (long i = 1; i < 1000001; i++) {
  
 master.submit(i);
  
 }
  
 master.execute();
  
   
 long re = 0;
  
 //获取计算结果
  
 Map<String, Object> resultMap = master.getResultMap();
  
   
 //如不满足这个while结果之后,说明计算完成了
  
   
 while (resultMap.size() > 0 || !master.isComplete()) {
  
   
 // 获取存放任务的Map
  
 Set<String> keys = resultMap.keySet();
  
   
 String key = null;
  
 // 获取第一个key然后跳出循环
  
 for (String k : keys) {
  
 key = k;
  
 break;
  
 }
  
   
 Long singleResult = null;
  
 if (key != null) {
  
 // 如果有key的话,就从resultMap里面取出这个key对应的计算完的结果
  
 singleResult = (Long) resultMap.get(key);
  
 }
  
 // 如果计算结果不为空的话就进行累加
  
 if (singleResult != null) {
  
 re += singleResult;
  
 }
  
 // 如果这个key不是null的话,就从计算结果里面删除这个key,原因是因为已经计算完了,留着也没啥用了.
  
 if (key != null) {
  
 resultMap.remove(key);
  
 }
  
 }
  
   
   
 Long end = System.currentTimeMillis();
  
   
 System.out.println("当前工作线程有: " + countWorker + " 个 ,执行计算结果为:" + re + " 耗时: " + (end - start) + "毫秒");
  
 }
  
   
   
}