**简单的线程池技术写法及要点**

上一篇文章讲了数据库连接池，这篇主要学习下线程池。

线程，即更小单位的进程，可以更加充分的利用cpu进行多任务操作。

Java中一开始就对多线程有了支持，通常编写另一个线程，都是为了某种特定的目的去的，也就是让他完成

特定的任务，并且通常也只有一个。但是如果有不定的待执行工作数请求执行，并且可能是不同操作的呢？

对于服务端的程序，经常面对的是客户端传入的短小（执行事件短、工作内容较为单一）任务，需要服务端快速处理并

返回结果。如果服务端每次接受到一个任务，创建一个线程，然后执行，这在原型阶段是个不错的选择，但是面对

成千上万的任务递交服务器时，如果还是采用原来的单一模式，将不会是一个很好的选择。因为这会使操作系统频繁的进行线程

上下文切换，无辜增加系统负载，而线程的创建和消亡都是需要耗费系统资源的，也无疑会浪费系统支援。

看如下代码：

/\*\*

\* 一个通用的线程池借口。

\*/

public interface ThreadPool<Job extends Runnable> {

// 执行一个Job，这个Job需要实现Runnable，即一个线程任务

void execute(Job job);

// 关闭线程池

void shutdown();

// 增加工作者线程

void addWorkers(int num);

// 减少工作者线程

void removeWorker(int num);

// 得到正在等待执行的任务数量

int getJobSize();

}

实现类

import java.util.ArrayList;

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

import java.util.concurrent.atomic.AtomicLong;

/\*\*

\* 具体实现线程池的类

\*/

public class DefaultThreadPool<Job extends Runnable> implements ThreadPool<Job> {

// 线程池最大限制数

private static final int MAX\_WORKER\_NUMBERS = 10;

// 线程池默认的数量

private static final int DEFAULT\_WORKER\_NUMBERS = 5;

// 线程池最小的数量

private static final int MIN\_WORKER\_NUMBERS = 1;

// 这是一个工作列表，将会向里面插入工作，类似于数据库连接池

private final LinkedList<Job> jobs = new LinkedList<Job>();

// 工作者列表，一个线程安全的工作队列，在其中去run一个job

private final List<Worker> workers = Collections.synchronizedList(new ArrayList<Worker>());

// 工作者线程的数量

private int workerNum = DEFAULT\_WORKER\_NUMBERS;

// 线程编号生成 ，使用atomiclong

private AtomicLong threadNum = new AtomicLong();

/\*\*

\* 默认构造方法

\*/

public DefaultThreadPool() {

initializeWokers(DEFAULT\_WORKER\_NUMBERS);

}

/\*\*

\* 指定初始时，线程池中线程数量，

\* 不能大于最大，也不能小于最小。

\* @param num

\*/

public DefaultThreadPool(int num) {

workerNum = num > MAX\_WORKER\_NUMBERS ? MAX\_WORKER\_NUMBERS : num < MIN\_WORKER\_NUMBERS ? MIN\_WORKER\_NUMBERS : num;

initializeWokers(workerNum);

}

/\*\*

\* 执行一个job，放在待执行的工作队列中，

\*/

public void execute(Job job) {

if (job != null) {

// 添加一个工作，然后进行通知

synchronized (jobs) {

jobs.addLast(job);

jobs.notify();

}

}

}

/\*\*

\* 停止一个工作，使用volatile标记位实现。

\*/

public void shutdown() {

for (Worker worker : workers) {

worker.shutdown();

}

}

public void addWorkers(int num) {

synchronized (jobs) {

// 限制新增的Worker数量不能超过最大值

if (num + this.workerNum > MAX\_WORKER\_NUMBERS) {

num = MAX\_WORKER\_NUMBERS - this.workerNum;

}

initializeWokers(num);

this.workerNum += num;

}

}

public void removeWorker(int num) {

synchronized (jobs) {

if (num >= this.workerNum) {

throw new IllegalArgumentException("beyond workNum");

}

// 按照给定的数量停止Worker

int count = 0;

while (count < num) {

workers.get(count).shutdown();

count++;

}

this.workerNum -= count;

}

}

public int getJobSize() {

return jobs.size();

}

/\*\*

\* 初始化线程工作者

\* @param num

\*/

private void initializeWokers(int num) {

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

Worker worker = new Worker();

workers.add(worker);

Thread thread = new Thread(worker, "ThreadPool-Worker-" + threadNum.incrementAndGet());

thread.start();

}

}

// 工作者，负责消费任务

class Worker implements Runnable {

// 是否工作

private volatile boolean running = true;

public void run() {

while (running) {

Job job = null;

synchronized (jobs) {

// 如果工作者列表是空的，那么就wait

//每一次按照FIFO的队列顺序拿一个job进行工作。

while (jobs.isEmpty()) {

try {

jobs.wait();

} catch (InterruptedException ex) {

// 感知到外部对WorkerThread的中断操作，返回

Thread.currentThread().interrupt();

return;

}

}

// 取出一个Job

job = jobs.removeFirst();

}

if (job != null) {

try {

job.run();

} catch (Exception ex) {

// 忽略Job执行中的Exception

}

}

}

}

public void shutdown() {

running = false;

}

}

}

因此，线程池技术能够很好地解决这个问题，它预先创建了若干个数量的线程，并且不能由用户直接对线程创建进行控制，在这个前提下重复使用固定或者

较为固定数目的线程来完成任务的执行。

一方面，消除了频繁创建和小王线程的系统资源开销，

另一方面，面对过量任务的提交能够平缓的劣化。

参考：<https://blog.csdn.net/anLA_/article/details/70227386>