链滴

多线程设计模式：Master-Worker 模式

作者：jianzh5
原文链接：https://ld246.com/article/1577607190161
来源网站：链滴

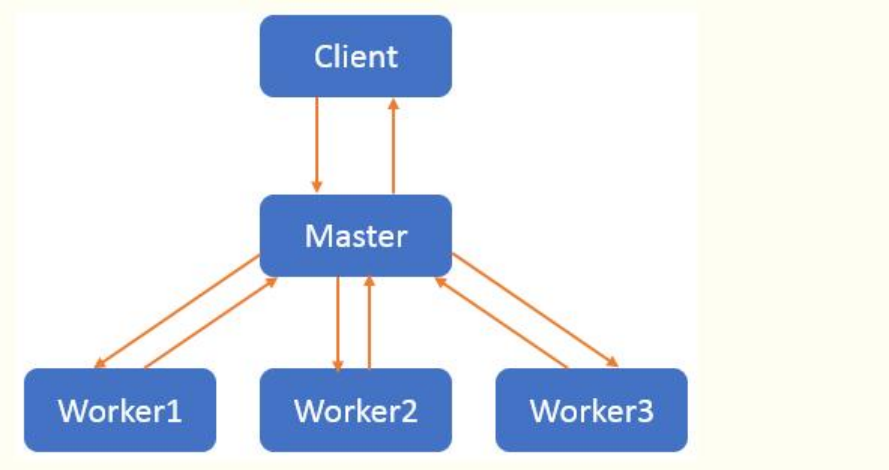
概念剖析

Master-Worker是常用的并行计算模式。它的核心思想是系统由两类进程协作工作：Master进程和Worker进程。

Master负责接收和分配任务，Worker负责处理子任务。当各个Worker子进程处理完成后，会将结果回给Master，由Master作归纳总结。

其好处就是能将一个大任务分解成若干个小任务，并行执行，从而提高系统的吞吐量。

处理过程如下图所示：



Master进程为主要进程，它维护一个Worker进程队列、子任务队列和子结果集。Worker进程队列中Worker进程不停从任务队列中提取要处理的子任务，并将结果写入结果集。

根据上面的思想，我们来模拟一下这种经典设计模式的实现。

过程分析

原文链接：多线程设计模式：Master-Worker 模式

1. 既然Worker是具体的执行任务，那么Worker一定要实现Runnable接口

2. Matser作为接受和分配任务，得先有个容器来装载用户发出的请求，在不考虑阻塞的情况下我们择ConcurrentLinkedQueue作为装载容器

3. Worker对象需要能从Master接收任务，它也得有Master ConcurrentLinkedQueue容器的引用

4. Master还得有个容器需要能够装载所有的Worker，可以使用HashMap<String,Thread>

5. Worker处理完后需要将数据返回给Master，那么Master需要有个容器能够装载所有worker并发理任务的结果集。此容器需要能够支持高并发，所以最好采用ConcurrentHashMap<String,Object>

6. 同理由于Worker处理完成后将数据填充进Master的ConcurrentHashMap，那么它也得有一份ConurrentHashMap的引用

代码实现

Task任务对象

```
public class Task {
    private int id;
    private String name;
    private int price;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }
}
```

Master对象：

```
public class Master {
    //任务集合
    private ConcurrentLinkedQueue<Task> taskQueue = new ConcurrentLinkedQueue<>();
```

```
    //所有的处理结果
    private ConcurrentHashMap<String,Object> resultMap = new ConcurrentHashMap<>();

    //所有的Worker集合
    private HashMap<String,Thread> workerMap = Maps.newHashMap();

    //构造方法，初始化Worker
    public Master(Worker worker,int workerCount){
        //每一个worker对象都需要有Master的引用，taskQueue用于任务的提取，resultMap用于任
的提交
        worker.setTaskQueue(this.taskQueue);
        worker.setResultMap(this.resultMap);
        for(int i = 0 ;i < workerCount; i++){
            //key表示worker的名字,value表示线程执行对象
            workerMap.put("worker"+i,new Thread(worker));
        }
    }

    //用于提交任务
    public void submit(Task task){
        this.taskQueue.add(task);
    }

    //执行方法，启动应用程序让所有的Worker工作
    public void execute(){
        for(Map.Entry<String,Thread> me : workerMap.entrySet()){
            me.getValue().start();
        }
    }

    //判断所有的线程是否都完成任务
    public boolean isComplete() {
        for(Map.Entry<String,Thread> me : workerMap.entrySet()){
          if(me.getValue().getState() != Thread.State.TERMINATED){
              return false;
          }
        }
        return true;
    }

    //总结归纳
    public int getResult(){
        int ret = 0;
        for (Map.Entry<String, Object> entry : resultMap.entrySet()) {
            ret+=(Integer) entry.getValue();
        }
        return ret;
    }
}
```

###Worker对象：

```
public class Worker implements Runnable{
    private ConcurrentLinkedQueue<Task> taskQueue;
```

```
    private ConcurrentHashMap<String, Object> resultMap;

    public void setTaskQueue(ConcurrentLinkedQueue<Task> taskQueue) {
        this.taskQueue = taskQueue;
    }

    public void setResultMap(ConcurrentHashMap<String, Object> resultMap) {
        this.resultMap = resultMap;
    }

    @Override
    public void run() {
        while(true){
            Task executeTask = this.taskQueue.poll();
            if(executeTask == null) break;
            //真正的任务处理
            Object result = handle(executeTask);
            this.resultMap.put(executeTask.getName(),result);
        }
    }

    //核心处理逻辑，可以抽离出来由具体子类实现
    private Object handle(Task executeTask) {
        Object result = null;
        try {
            //表示处理任务的耗时....
            Thread.sleep(500);
            result = executeTask.getPrice();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

客户端调用

```
public class Main {

    public static void main(String[] args) {
        //实际开发中多少个线程最好写成Runtime.getRuntime().availableProcessors()
        Master master = new Master(new Worker(), 10);
        Random random = new Random();
        for(int i = 0 ;i <= 100 ;i++){
            Task task = new Task();
            task.setId(i);
            task.setName("任务"+i);
            task.setPrice(random.nextInt(1000));
            master.submit(task);
        }
        master.execute();
        long start = System.currentTimeMillis();
        while(true){
            if(master.isComplete()){
```

```
                long end  = System.currentTimeMillis() - start;
                int ret = master.getResult();
                System.out.println("计算结果:"+ret+",执行耗时:"+end);
                break;
            }
        }
    }
}
```

在Worker对象中的核心处理业务逻辑handle()方法最好抽象成公共方法，具体实现由子类覆写。