36 为多线程们安排一位经理—Master/Slave模式详解

更新时间：2019-12-24 14:13:05

> 没有引发任何行动的思想都不是思想，而是梦想。
>
> —— 马丁

前文我们讲过 ForkJoinPool 是分而治之的思想。今天我们将要学习的 Master/Slave 也是同样的思想。其中 Master 负责承接一个大的任务，然后它会根据一定策略把大任务拆散为若干个小任务，然后随机分发给一组 Slave。每个 Slave 完成任务后上报自己的任务完成情况。当所有 Slave 都完成了自己的任务时，Master 也就完成了自己的任务。Master 就像是 Slave 的经理，把自己的任务分发下去，而 Slave 则在完成工作后向它汇报。

1、Master/Slave 模式设计

1.1 Master 设计

在 Master/Slave 模式中，一个 Master 持有一组 Slave 的引用。Master 对外暴露一个承接任务的方法 startTask。这是 Master 的主要方法，在内部做了如下事情：

创建 slave

由于创建 Slave 线程并启动的操作比较重，所以放到提交任务的时候才真正去做；

分发任务

把 Task 进行拆分，然后分发给每个 Slave；

等待处理结果

轮循检查任务是否全部完成，全部完成结束轮循；

返回处理结果

返回任务执行结果。

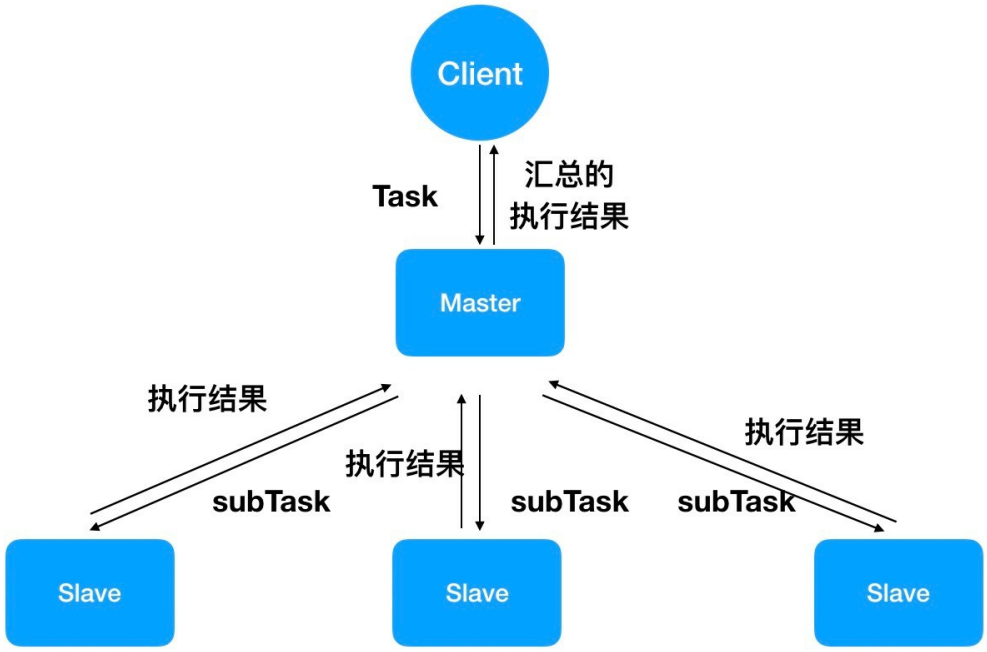
可以看到这四个方法逻辑十分的清晰。

1.2 Slave 设计

下面我们再看看 Slave 的设计：

Slave 继承自 Thread。内部通过阻塞队列 BlockingQueue 保存 Task。这样在取任务时候如果已经没有，则会阻塞等待。它有一个 submitTask 用来提交子任务，这个方法在 Master 分发任务时会被调用。此外还有 run 方法从 BlockingQueue 中取得任务执行。执行结束后通知 Master。

以上的设计并不是固定的模式。但 Master 接收任务，分割任务，派发任务这些功能是要有的，此外 Master 要有能力知道所有子任务都被执行完毕。而 Slave 则需要不断承接子任务，并且执行。执行完毕能够把执行结果回写给 Master。设计如下图：



其实说这么多，不如直接看代码。下面我们就通过一个小例子，来感受一下 Master/Slave 模式。

2、Master/Slave 代码示例

2.1 Client 代码

不知道你是否还记得本专栏开始几节反复用来举例的单词抄写的需求。本节是正文最后一篇，正好我们回到最初的例子，用 Master/Slave 方式来实现它。我们这次先看 Client 的代码：

```
public class Client {
    public static void main(String[] args) throws InterruptedException {
        Task task = new Task(123,"internationalization");

        Master master = new Master();

        master.startTask(task);

        master.printResult();
    }
}
```

特别的简单，创建一个单词抄写的 Task，然后通过 Master 来执行，最后打印执行结果。

2.2 Task 代码

Task 代码如下，省略了 get 方法 ：

```
public class Task {
  //要抄写的次数
    private int copyCount;
  //抄写的序号开始
    private int from;
  //抄写的序号结束
    private int to;
  //要抄写的单词
    private String word;

    public Task(int copyCount, int from, int to, String word) {
        this.copyCount = copyCount;
        this.word = word;
        this.from = from;
        this.to = to;
    }

    public Task(int copyCount, String word) {
        this.copyCount = copyCount;
        this.word = word;
        this.from = 1;
        this.to = copyCount;
    }
}
```

接下来我们看看 Master 代码。

2.3 Master 代码

我们先来看看 Master 有哪些属性：

```
//保存干活的Slave线程
private List<Slave> slaves;
//slave的数量
private static final int SLAVES_COUNT = 8;
//子任务拆分的力度
private static final int SUB_TASK_SIZE = 4;
//完成的任务数量。各个Slave线程都会更新此数量，所以使用Atomic变量
private AtomicInteger finishedTaskCount = new AtomicInteger(0);
//执行结果，key为线程名字，value为此线程完成的数量
private ConcurrentHashMap<String, Integer> results;
```

可以看到 Master 持有一组 slave 线程，用来为它干活。我们的任务是单词抄写，每个子任务由 SUB_TASK_SIZE 来控制单个小任务的抄写次数。子线程抄写完成后会更新 finishedTaskCount 和 results 做任务完成记录。

Master 对外提供了如下方法：

```
//主方法，用于执行任务
public ConcurrentHashMap<String, Integer> startTask(Task task)
//子方法完成后向Master提交完成记录
public void subTaskFinished(String slaveName,int finishedSubTaskCount)
//打印执行结果
public void printResult()
```

这三个方法里最重要的就是 startTask，Master 主要的执行逻辑都在里面，代码如下：

```
public ConcurrentHashMap<String, Integer> startTask(Task task) throws InterruptedException {

    // 1 创建slave
    createSlaves(this);

    // 2 分发任务
    splitAndAssignTask(task);

    // 3 等待结果处理
    checkTaskFinished(task);

    // 4 返回处理结果
    return results;
}
```

startTask 内部主要调用三个方法，最后返回执行结果。由于创建线程成本高，所以在构造 Master 时并没有创建 Slave，而是延迟到 startTask 的时候来创建。splitAndAssignTask 做的事情就是把大的 task 按照拆分逻辑拆开，分发给 slave 去执行。checkTaskFinished 会轮循检查 task 的执行情况，当全部完成时，执行下面的 return 语句。这几个方法都很重要，接下来我们一个个看。

2.3.1 createSlaves 方法

```
private void createSlaves(Master master) {
    if(slaves.size()==0){
        IntStream.range(0, this.SLAVES_COUNT).forEach(count ->
                slaves.add(new Slave("slave " + count, master))
        );

        slaves.forEach(slave -> {
            slave.start();
        });
    }
}
```

这个方法比较简单，就是创建 SLAVES_COUNT 个 slave，然后启动起来。

2.3.2 splitAndAssignTask 方法

```
private void splitAndAssignTask(Task task) throws InterruptedException {
    int count = task.getCopyCount();
    int start = 1;
    List<Task> subTasks = new ArrayList<>();
    //拆分task
    while (start <= count) {
        int end = count + 1;

        if (start + SUB_TASK_SIZE <= count) {
            end = start + SUB_TASK_SIZE;
        }

        subTasks.add(new Task(end-start, start, end, task.getWord()));

        start = end;
    }

  //分发subTask
    for (int i = 0; i < subTasks.size(); i++) {
        int slaveIndex = i % SLAVES_COUNT;
        slaves.get(slaveIndex).submitTask(subTasks.get(i));
    }
}
```

这个方法做了两件事情，一是把 task 拆分为多个 subTask。二是把 subTask 分发给 slave 去执行。subTask 中保存了要 copy 的数量，以及 copy 的 from 序号和 to 序号。当然还有要抄写的单词。

2.3.3 checkTaskFinished

这个方法用来检查 task 是否全部执行完成。

```
private void checkTaskFinished(Task task) throws InterruptedException {
    while (true) {
        if (task.getCopyCount() == finishedTaskCount.get()) {
            finished();
            break;
        }

        TimeUnit.MILLISECONDS.sleep(200);
    }
}
```

方法中使用的轮循的方式来检查 task 的 copy 总数和已完成数量 finishedTaskCount 是否一致，如果一致则说明 task 已经全部完成，那么调用 finished 方法工作做收尾，跳出循环。

2.3.4 subTaskFinished

Master 除了这几个方法还有一个方法用于子线程提交执行结果。代码如下：

```
public void subTaskFinished(String slaveName,int finishedSubTaskCount) {
    Integer count = results.get(slaveName);
    if(count==null){
        results.put(slaveName,finishedSubTaskCount);
    }else{
        results.put(slaveName,count+finishedSubTaskCount);
    }

    finishedTaskCount.getAndAdd(finishedSubTaskCount);
}
```

首先把执行结果放入 results，如果已经存在，则进行累计。此外更新 finishedTaskCount。

Master 的主要方法都已经介绍完毕。下面我们来看看 Slave。

2.4 Slave 代码

Slave 是一个工作的线程，它继承自 Thread 类，

```
public class Slave extends Thread
```

我们先看看 Slave 的属性：

```
//slave的线程名字
private String name;
//持有master引用，因为需要向master提交执行结果
private Master master;
//阻塞队列来保存task
private BlockingQueue<Task> tasks;
```

slave 中提供两个方法，一个是提交 task 的方法 submitTask，代码如下：

```
public void submitTask(Task task) throws InterruptedException {
    tasks.put(task);
}
```

代码很简单，只是向阻塞队列中放入 task。

Slave 执行 task 的逻辑在 run 方法中，Slave 继承自 Thread，当他启动后，run 方法就会被调用。代码如下：

```
@Override
public void run() {
    try {
        while (true) {
            Task task = tasks.take();

            IntStream.range(task.getFrom(), task.getTo()).forEach(
                    count -> System.out.println(String.format("线程%s第%d抄写单词%s", name, count, task.getWord()))
            );

            master.subTaskFinished(name, task.getCopyCount());
        }

    } catch (InterruptedException e) {
        System.out.println(String.format("线程%s被打断", name));
    }
}
```

这段代码不断的从阻塞队列中 take 出 task。如果没有 task，就会阻塞在此。然后根据 task 内容进行输出。执行完成后调用 master 的 subTaskFinished 方法把自己的执行结果提交给 master。如果阻塞的时候被打断，则打印出日志。

3、执行结果分析

在 Client 的 main 方法中我们声明了一个 task = new Task (123,“internationalization”)，抄写 internationalization 单词 123 次。运行后输出如下：

```
线程slave 1第5抄写单词internationalization
线程slave 5第21抄写单词internationalization
线程slave 4第17抄写单词internationalization
线程slave 2第9抄写单词internationalization
线程slave 5第22抄写单词internationalization
线程slave 3第13抄写单词internationalization
线程slave 0第1抄写单词internationalization

.....

线程slave 2第107抄写单词internationalization
线程slave 0第100抄写单词internationalization
线程slave 2第108抄写单词internationalization
任务全部完成!
线程slave 4被打断
线程slave 0被打断
线程slave 6被打断
线程slave 1被打断
线程slave 7被打断
线程slave 3被打断
线程slave 5被打断
线程slave 2被打断
线程slave 0,完成了16次抄写
线程slave 7,完成了12次抄写
线程slave 5,完成了16次抄写
线程slave 6,完成了15次抄写
线程slave 3,完成了16次抄写
线程slave 4,完成了16次抄写
线程slave 1,完成了16次抄写
线程slave 2,完成了16次抄写
```

中间省略了一些输出。可以看到所有任务完成后 slave 线程都被打断。最后结果输出了每个线程抄写的次数，相加总和为 123。我把上面的 slave 打印日志做了统计，也是打印了 123 条。完全符合我们的预期。

4、总结

Master/Slave 模式是常用的多线程设计模式。一般用于大任务的拆分和分发。Master 作为门面对外暴露任务执行的接口，内部则是分发给多个 Slave 线程完成。这一切对于调用者来说是透明的。Master/Slave 模式关键点在于任务的分发和结果的汇总。它的实现方式很灵活，本文只是一种方式，也可以通过线程池来实现。子任务的计算结果也可以使用 Future。此外，分布式系统也有 Master/slave 的设计模式，可以借助 ZooKeeper 来实现。在 Akka 中使用 Actor 也能实现 Master/Slave 模式。实际使用中可以根据业务需求来自己实现。我们只需要掌握模式的核心思想，而不用拘泥于某一种具体的实现方式。

附完成代码

Master 代码：

```
public class Master {

    private List<Slave> slaves;

    private static final int SLAVES_COUNT = 8;

    private static final int SUB_TASK_SIZE = 4;

    private AtomicInteger finishedTaskCount = new AtomicInteger(0);

    private ConcurrentHashMap<String, Integer> results;

    public Master() {
        results = new ConcurrentHashMap<>();
```

```
        slaves = new ArrayList<>();
    }

    public ConcurrentHashMap<String, Integer> startTask(Task task) throws InterruptedException {

        // 1 创建slave
        createSlaves(this);

        // 2 分发任务
        splitAndAssignTask(task);

        // 3 等待结果处理
        checkTaskFinished(task);

        // 4 返回处理结果
        return results;
    }

    private void createSlaves(Master master) {
        if (slaves.size() == 0) {
            IntStream.range(0, this.SLAVES_COUNT).forEach(count ->
                    slaves.add(new Slave("slave " + count, master))
            );

            slaves.forEach(slave -> {
                slave.start();
            });
        }
    }

    private void splitAndAssignTask(Task task) throws InterruptedException {
        int count = task.getCopyCount();
        int start = 1;
        List<Task> subTasks = new ArrayList<>();
        while (start <= count) {
            int end = count + 1;

            if (start + SUB_TASK_SIZE <= count) {
                end = start + SUB_TASK_SIZE;
            }

            subTasks.add(new Task(end - start, start, end, task.getWord()));

            start = end;
        }


        for (int i = 0; i < subTasks.size(); i++) {
            int slaveIndex = i % SLAVES_COUNT;
            slaves.get(slaveIndex).submitTask(subTasks.get(i));
        }
    }

    public void subTaskFinished(String slaveName, int finishedSubTaskCount) {
        Integer count = results.get(slaveName);

        if (count == null) {
            results.put(slaveName, finishedSubTaskCount);
        } else {
            results.put(slaveName, count + finishedSubTaskCount);
        }

        finishedTaskCount.getAndAdd(finishedSubTaskCount);
    }

    private void checkTaskFinished(Task task) throws InterruptedException {
        while (true) {
            if (task.getCopyCount() == finishedTaskCount.get()) {
                finished();
```

```
                break;
            }

            TimeUnit.MILLISECONDS.sleep(200);
        }
    }

    private void finished() {
        System.out.println("任务全部完成！");
        slaves.forEach(slave -> slave.interrupt());
        slaves.clear();
    }

    public void printResult() {
        results.forEach((key, value) ->
                System.out.println(String.format("线程%s,完成了%d次抄写", key, value)));
    }
}
```

Slave 代码：

```
public class Slave extends Thread {
    private String name;

    private Master master;

    private BlockingQueue<Task> tasks;

    public Slave(String name, Master master) {
        this.name = name;
        this.master = master;
        this.tasks = new ArrayBlockingQueue<Task>(100);
    }

    public void submitTask(Task task) throws InterruptedException {
        tasks.put(task);
    }

    @Override
    public void run() {
        try {
            while (true) {
                Task task = tasks.take();

                IntStream.range(task.getFrom(), task.getTo()).forEach(
                        count -> System.out.println(String.format("线程%s第%d抄写单词%s", name, count, task.getWord()))
                );

                master.subTaskFinished(name, task.getCopyCount());
            }

        } catch (InterruptedException e) {
            System.out.println(String.format("线程%s被打断", name));
        }
    }
}
```

Task 代码：

```
public class Task {
    private int copyCount;
    private int from;
    private int to;
    private String word;

    public Task(int copyCount, int from, int to, String word) {
        this.copyCount = copyCount;
        this.word = word;
        this.from = from;
        this.to = to;
    }

    public Task(int copyCount, String word) {
        this.copyCount = copyCount;
        this.word = word;
        this.from = 1;
        this.to = copyCount;
    }

    public int getCopyCount() {
        return copyCount;
    }

    public String getWord() {
        return word;
    }

    public int getFrom() {
        return from;
    }

    public int getTo() {
        return to;
    }
}
```

Client 代码：

```
public class Client {
    public static void main(String[] args) throws InterruptedException {
        Task task = new Task(123,"internationalization");

        Master master = new Master();

        master.startTask(task);

        master.printResult();
    }
}
```

}

35拆分你的任务—学习使用Fork/Join框架

37 结束语