

发件人: 方堃 fangkun119@icloud.com  
主题: CH12 Master-Slave  
日期: 2017年5月2日 下午12:02  
收件人:

方

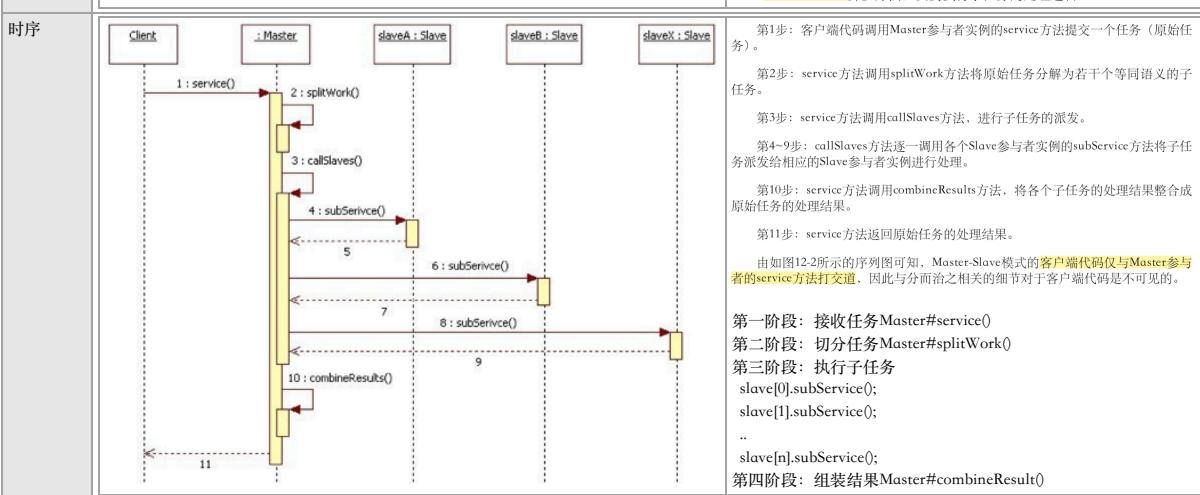
## CH12 Master-Slave

2017年4月30日 星期日  
下午4:51

**用途** Master-Slave模式是一个基于分而治之(Divide and conquer)思想的设计模式。其核心思想是将一个任务(原始任务)分解为若干个语义等同(Semantically-identical)的子任务，并由专门的工作者线程来并行执行这些子任务。原始任务的处理结果是通过整合各个子任务的处理结果而形成的。而这些与分而治之相关的处理细节对于原始任务的提交方来说又是不可见的。因此，Master-Slave模式既提高计算效率，又实现了信息隐藏。



图12-1. Master-Slave模式的类图



- 使用**
- 并行计算 (Parallel Computation)**。该场景使用Master-Slave模式是为了提升计算性能。本案例就属于该运用场景。在此情形下，原始任务的处理结果是通过组合每个Slave实例的处理结果形成的。
  - 容错处理 (Fault Tolerance)**。该场景使用Master-Slave模式是为了提高计算的可靠性。在此情形下，原始任务的处理结果是任意一个Slave实例的成功处理结果(那些处理失败的Slave实例无法为Master返回结果)。
  - 计算精度 (Computational Accuracy)**。该场景使用Master-Slave模式是为了提高计算的精确程度。在此情形下，原始任务的处理结果是所有Slave实例中处理结果不精确性最低的一个结果。

- 好处**
- 可交换性 (Exchangeability) 和可扩展性 (Extensibility)**。如上文所述，如果我们使所有的Slave参与者实现类拥有共同的接口，那么替换某个Slave实例，增加一个Slave实例对Master参与者产生的影响很小。当然，客户端代码也不会受此影响。Master进行任务分解、任务派发和对子任务处理结果的合并所涉及的算法如果有改变也不会影响到Slave参与者和客户端代码。
  - 提升计算性能**。Master-Slave模式实现恰当的话可以提升计算性能。这里，我们需要注意原始任务的分解、子任务的派发、子任务处理结果的合并以及Slave线程的管理都有其自身的时间和空间消耗。

**实现考虑点**

结果收集	Master参与者需要收集各个子任务的处理结果，才能生成原始任务的处理结果。收集子任务的处理结果通常有两种方法。 一种是使用存储仓库 (Repository)。所谓的存储仓库是一个Master参与者和Slave参与者都能够访问的数据结构。Slave参与者实例将其子任务的处理结果存入存储仓库。Master参与者可以等待各个Slave参与者实例处理完毕后从存储仓库中获取各个子任务的处理结果。本章案例就是采用这种方法收集子任务的处理结果。清单12-1中的类Master的实例变量repository (其类型为ConcurrentMap)就是一个用于收集子任务处理结果的存储仓库。 另外一种方法是使用第6章中介绍的Promise模式。这里，我们可以使Slave参与者的subService方法的返回值为Promise模式中的Promise参与者实例。通常可以使subService方法的返回值为一个java.util.concurrent.Future实例。这样，Master参与者可以通过调用各个Slave参与者的subService方法的返回值的get()方法来获取子任务的处理结果。
------	---

负载均衡的规模和数量是否均衡。

如果原始任务的规模事先可知，那么Master参与者可以根据原始任务的规模和Slave参与者实例的个数算出的平均数进行子任务的派发<sup>11</sup>。这时，每个Slave参与者实例被分配到的子任务的总规模是相等的。在并行计算场景中，一个Slave参与者实例通常就是一个线程。为了避免增加JVM线程调度的负担，Slave线程的数量一般要根据JVM所在主机的CPU个数来定。通常，Slave线程数量只比CPU个数大一点。

如果原始任务的规模事先不可知，则Master参与者在派发子任务的时候可能要采用某种负载均衡算法来使得各个Slave线程所分配到的子任务的规模和数量是均衡的。本章案例就是属于这种情形。流量统计工具的Java模块是从标准输入中读取Linux Shell脚本模块输出的接口日志文件名，我们不想等到Linux Shell脚本模块输出其查找到的所有接口日志文件的文件名后再进行统计，因此对于Java模块而言，原始任务的规模是其事先不可知的。故其使用了简单的轮询(Round-Robin)算法(代码见清单12-1)来保持各个Slave线程的子任务负载均衡。

上述的负载均衡着眼点在于子任务的派发上。Master-Slave模式可以看成Producer-Consumer模式的一个实例。Master-Slave模式的Master参与者和Slave参与者分别相当于Producer-Consumer模式(参见第7章)中的Producer参与者和Consumer参与者。因此，从Slave实例运行的角度来看，我们可以使用第7章中介绍的工作窃取算法来动态调整各个Slave实例的计算负载。

负载均衡的规模和数量是否均衡。

如果原始任务的规模事先可知，那么Master参与者可以根据原始任务的规模和Slave参与者实例的个数算出的平均数进行子任务的派发<sup>11</sup>。这时，每个Slave参与者实例被分配到的子任务的总规模是相等的。在并行计算场景中，一个Slave参与者实例通常就是一个线程。为了避免增加JVM线程调度的负担，Slave线程的数量一般要根据JVM所在主机的CPU个数来定。通常，Slave线程数量只比CPU个数大一点。

如果原始任务的规模事先不可知，则Master参与者在派发子任务的时候可能要采用某种负载均衡算法来使得各个Slave线程所分配到的子任务的规模和数量是均衡的。本章案例就是属于这种情形。流量统计工具的Java模块是从标准输入中读取Linux Shell脚本模块输出的接口日志文件名，我们不想等到Linux Shell脚本模块输出其查找到的所有接口日志文件的文件名后再进行统计，因此对于Java模块而言，原始任务的规模是其事先不可知的。故其使用了简单的轮询(Round-Robin)算法(代码见清单12-1)来保持各个Slave线程的子任务负载均衡。

Master参与者的实现类需要注意处理其与Slave参与者的通信异常以及Slave参与者对子任务处理的异常。

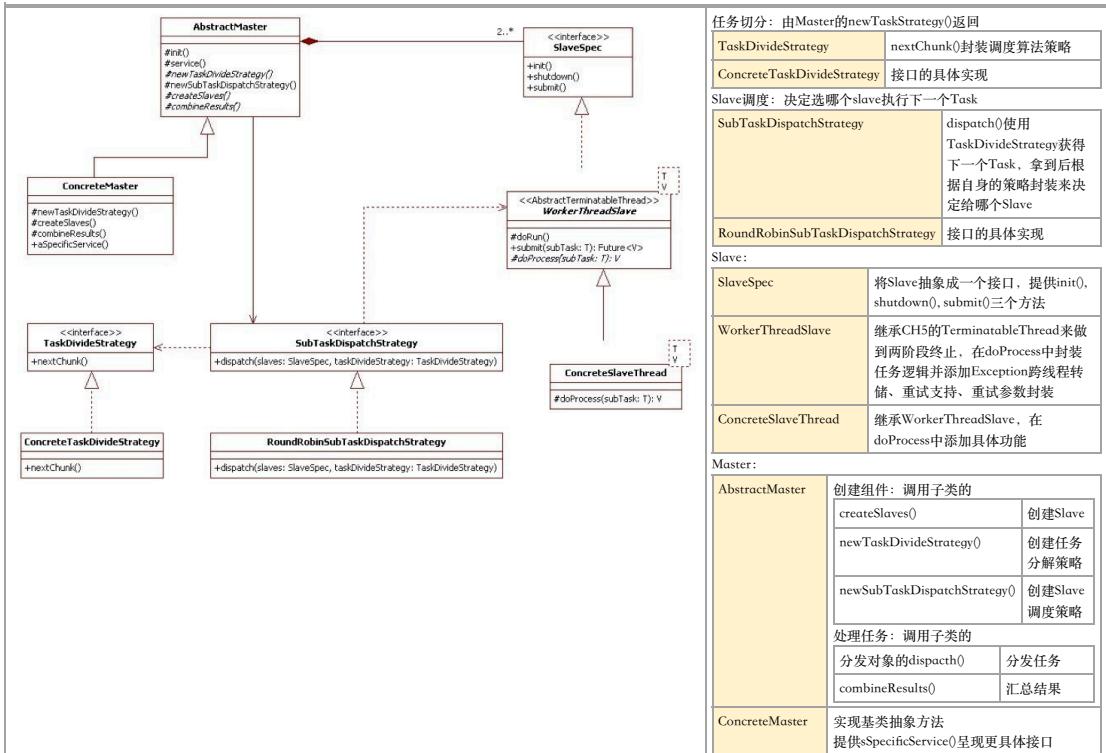
由于Slave参与者实例是运行在工作者线程中的，而Master参与者实例是运行在客户端线程中的，后者要获取前者抛出的异常有点困难(详情参见第6章)。如果Master参与者利用上文所述的基于Promise模式的方法来获取子任务的处理结果，那么获取Slave参与者实例的处理异常就变得非常简单：Master参与者实例调用subService方法的get()方法获取子任务处理结果的时候，如果get()方法抛出异常则说明相应的子任务处理失败。

为了提高计算的可靠性，Master参与者在侦测到上述异常时可以考虑由其自身重新执行处理失败的子任务，即让处理失败的子任务运行在客户端线程中，而不是在Slave参与者的工作者线程中；类似于ThreadLocalExecutor框架的newUntilSuccessful(ThreadLocalExecutor callerRunsFinally)方法。但要注意的是，如果Master-Slave模式的Master参与者在侦测到上述异常时，直接在客户端线程中重新执行处理失败的子任务，那么可能会导致线程泄漏，从而造成内存泄漏。

复用代码中涉及的SubTaskFailureException异常类（代码见清单12-8）和RetryInfo类（代码见清单12-9）为这种重试机制提供了支持。

Slave线程 在并行计算场景中，每个Slave参与者实例就是一个线程。通常，这些线程会使用阻塞队列（BlockingQueue）来接收子任务，而线程则从阻塞队列中取出子任务（即调用BlockingQueue的take()方法）进行执行。而BlockingQueue的take()在队列为空时会使当前线程一直处于等待状态，因此当Slave线程处理完分配给它的所有子任务时，Slave线程仍然未停止。此时，我们可以使用Two-phase Termination模式（参见第5章）来实现Slave线程在处理完分配给它的子任务后停止。本章案例就是采用这种方法来停止Slave线程。

可复用 Master-Slave模式的可复用代码实现起来有些复杂，需要借助Template（模板）模式、Strategy（策略）模式和Factory Method（工厂方法）模式。



任务切分：由Master的newTaskDivideStrategy()返回

TaskDivideStrategy nextChunk()封装调度算法策略

ConcreteTaskDivideStrategy 接口的具体实现

Slave调度：决定选哪个slave执行下一个Task

SubTaskDispatchStrategy dispatch()使用 TaskDivideStrategy获得下一个Task，拿到后根据自身的策略封装来决定给哪个Slave

RoundRobinSubTaskDispatchStrategy 接口的具体实现

Slave：

SlaveSpec 将Slave抽象成一个接口，提供init(), shutdown(), submit()三个方法

WorkerThreadSlave 继承CH5的TerminatableThread做到两阶段终止，在doProcess中封装任务逻辑并添加Exception跨线程转储、重试支持、重试参数封装

ConcreteSlaveThread 继承WorkerThreadSlave，在doProcess中添加具体功能

Master:

AbstractMaster	创建组件：调用子类的 createSlaves() newTaskDivideStrategy() newSubTaskDispatchStrategy()	创建Slave
	处理任务：调用子类的 分发对象的dispatch() combineResults()	创建任务 分解策略 汇总结果
ConcreteMaster	实现基类抽象方法 提供#specificService()呈现更具体接口	

接口

任务切分	<pre>public interface TaskDivideStrategy&lt;T&gt; {     /**      * 返回下一个子任务。若返回值为null，则表示无后续子任务。      *      * @return 下一个子任务      */     T nextChunk(); }</pre>	
------	--	--

Slave调度

```
* @param <T> 子任务类型
* @param <V> 子任务处理结果类型
*/
public interface SubTaskDispatchStrategy<T,V> {
    /**
     * 根据指定的原始任务分解策略，将分解得来的各个子任务派发给一组Slave参与者实例。
     *
     * @param slaves 可以接受子任务的一组Slave参与者实例
     * @param taskDivideStrategy 原始任务分解策略
     * @return iterator, 遍历该iterator可得到用于获取子任务处理结果的Promise (参见第6章,
     * Promise模式) 实例。
     * @throws InterruptedException 当Slave工作者线程被中断时抛出该异常。
     */
    Iterator<Future<V>> dispatch(Set<? extends SlaveSpec<T, V>> slaves,
        TaskDivideStrategy<T> taskDivideStrategy) throws InterruptedException;
}
```

任务切分结果被封装在taskDivideStrategy中，  
调用taskDivideStrategy的nextChunk()可以得到下一个任务

public class RoundRobinSubTaskDispatchStrategy<T, V> implements SubTaskDispatchStrategy<T, V> {
 Future<Rst>通过SlaveSpec<Task,Rst>的submit(TaskSpec)得到
 to
}

```
@SuppressWarnings("unchecked")
@Override
public Iterator<Future<V>> dispatch(Set<? extends SlaveSpec<T, V>> slaves,
    TaskDivideStrategy<T> taskDivideStrategy) throws InterruptedException {
    final List<Future<V>> subResults = new LinkedList<Future<V>>();
    T subTask;
    Object[] arrSlaves = slaves.toArray();
    int i = -1;
    final int slaveCount = arrSlaves.length;
    Future<V> subTaskResultPromise;

    while (null != (subTask = taskDivideStrategy.nextChunk())) {
        i = (i + 1) % slaveCount;
        subTaskResultPromise = ((WorkerThreadSlave<T, V>) arrSlaves[i])
            .submit(subTask);
        subResults.add(subTaskResultPromise);
    }
}
```

Slave

```
public interface SlaveSpec<T, V> {
    /**
     * 用于Master向其提交一个子任务。
     */
}
```

T: TaskType

V: ResultType

```

        * @param task
        *          子任务
        * @return 可借以获取子任务处理结果的Promise (参见第6章, Promise模式) 实例。
        * @throws InterruptedException
     */
    Future<V> submit(final T task) throws InterruptedException;

    /**
     * 初始化Slave实例提供的服务
     */
    void init();

    /**
     * 停止Slave实例对外提供的服务
     */
    void shutdown();
}

public abstract class WorkerThreadSlave<T, V> extends AbstractTerminatableThread implements SlaveSpec<T, V> {
    private final BlockingQueue<Runnable> taskQueue;

    public WorkerThreadSlave(BlockingQueue<Runnable> taskQueue) {
        this.taskQueue = taskQueue;
    }

    public Future<V> submit(final T task) throws InterruptedException {
        FutureTask<V> ft = new FutureTask<V>(new Callable<V>() {
            @Override
            public V call() throws Exception {
                V result;
                try {
                    result = doProcess(task);
                } catch (Exception e) {
                    SubTaskFailureException stfe = newSubTaskFailureException(task, e);
                    throw stfe;
                }
                return result;
            }
        });
        taskQueue.put(ft);
        terminationToken.reservations.incrementAndGet();
        return ft;
    }

    private SubTaskFailureException newSubTaskFailureException(final T subTask,
                                                               Exception cause) {
        RetryInfo<T, V> retryInfo = new RetryInfo<T, V>(subTask, new Callable<V>() {
            @Override
            public V call() throws Exception {
                V result;
                result = doProcess(subTask);
                return result;
            }
        });
        return new SubTaskFailureException(retryInfo, cause);
    }

    /**
     * 留给子类实现。用于实现子任务的处理逻辑。
     *
     * @param task
     *          子任务
     * @return 子任务的处理结果
     * @throws Exception
     */
    protected abstract V doProcess(T task) throws Exception;

    @Override
    protected void doRun() throws Exception {
        try {
            Runnable task = taskQueue.take();
            task.run();
        } finally {
            terminationToken.reservations.decrementAndGet();
        }
    }

    @Override
    public void init() {
        start();
    }

    @Override
    public void shutdown() {
        terminate(true);
    }
}

Master public abstract class AbstractMaster<T, V, R> {
    protected volatile Set<? extends SlaveSpec<T, V>> slaves;

    // 子任务派发算法策略
    private volatile SubTaskDispatchStrategy<T, V> dispatchStrategy;

    public AbstractMaster() {
    }
}

```

AbstractTerminatableThread参考CH5, 实现两阶段终止  
FutureTask<ReturnType>参考JDoc

```

protected void init() {
    slaves = createSlaves();
    dispatchStrategy = newSubTaskDispatchStrategy();
    for (SlaveSpec<T, V> slave : slaves) {
        slave.init();
    }
}

/**
 * 对子类暴露的服务方法。该类的子类需要定义一个比该方法命名更为具体的服务方法（如
 * downloadfileService）。
 * 由命名含义具体的服务方法（如downloadFileService）调用该方法。
 * 该方法使用了Template（模板）模式、Strategy（策略）模式。
 *
 * @param params
 *          客户端代码传递的参数列表
 */
protected R service(Object... params) throws Exception {
    final TaskDivideStrategy<T> taskDivideStrategy =
newTaskDivideStrategy(params);

    /*
     * 对原始任务进行分解，并将分解得来的子任务派发给Slave参与者实例。这里使用了Strategy
     * （策略）模式：原始任务分解和子任务派发。
     * 这两个具体的计算是通过调用需要的算法策略（对象）实现的。
     */
    Iterator<Future<V>> subResults = dispatchStrategy.dispatch(slaves,
        taskDivideStrategy);

    // 等待Slave实例处理结果
    for (SlaveSpec<T, V> slave : slaves) {
        slave.shutdown();
    }

    // 合并子任务的处理结果
    R result = combineResults(subResults);
    return result;
}

/**
 * 留给子类实现。用于创建原始任务分解算法策略。
 *
 * @param params
 *          客户端代码调用service方法时传递的参数列表
 */
protected abstract TaskDivideStrategy<T> newTaskDivideStrategy(
    Object... params);

/**
 * 用于创建子任务派发算法策略。默认使用轮询（Round-Robin）派发算法。
 *
 * @return 子任务派发算法策略实例。
 */
protected SubTaskDispatchStrategy<T, V> newSubTaskDispatchStrategy() {
    return new RoundRobinSubTaskDispatchStrategy<T, V>();
}

/**
 * 留给子类实现。用于创建slave参与者实例。
 *
 * @return 一组Slave参与者实例。
 */
protected abstract Set<? extends SlaveSpec<T, V>> createSlaves();

/**
 * 留给子类实现。用于合并子任务的处理结果。
 *
 * @param subResults
 *          各个子任务处理结果
 * @return 原始任务的处理结果
 */
protected abstract R combineResults(Iterator<Future<V>> subResults);
}

```

异常转@SuppressWarnings("serial")  
及重试 public class SubTaskFailureException extends Exception {

```

    /**
     * 对处理失败的子任务进行重试所需的信息
     */
    @SuppressWarnings("rawtypes")
    public final RetryInfo retryInfo;

    @SuppressWarnings("rawtypes")
    public SubTaskFailureException(RetryInfo retryInfo, Exception cause) {
        super(cause);
        this.retryInfo = retryInfo;
    }
}

```

```

public class RetryInfo<T, V> {
    public final T subTask;
    public final Callable<V> redoCommand;
    public RetryInfo(T subTask, Callable<V> redoCommand) {
        this.subTask = subTask;
        this.redoCommand = redoCommand;
    }
}

```

例子1

用上述可复用代码实现Master-Slave模式

事项 1. 【必需】 创建TaskDivideStrategy接口的实现类，在该类中实现原始任务分解算法。

2. 【必需】创建AbstractMaster的子类。该子类除了实现其父类定义的几个抽象方法外，还要定义服务方法，该服务方法的名字比其父类的service方法含义更为具体。
3. 【必需】创建WorkerThreadSlave的子类。在该子类中实现其父类的doProcess方法。当然，我们也可以自己编写SlaveSpec接口的实现类。
4. 【可选】创建SubTaskDispatchStrategy的实现类。在该类中实现子任务派发算法。AbstractMaster默认使用RoundRobinTaskDispatchStrategy。

```

代码 public class ParallelPrimeGenerator {
    public static void main(String[] args) throws Exception {
        PrimeGeneratorService primeGeneratorService = new PrimeGeneratorService();
        Set<BigInteger> result = primeGeneratorService.generatePrime(Integer
            .valueOf(args[0]));
        System.out.println("Generated " + result.size() + " prime:");
        System.out.println(result);
    }
}

class PrimeGeneratorService extends AbstractMaster<Range, Set<BigInteger>, Set<BigInteger>> {
    public PrimeGeneratorService() {
        this.init();
    }

    // 创建子任务分解算法实现类
    @Override
    protected TaskDivideStrategy<Range> newTaskDivideStrategy(
        final Object... params) {

        final int numSlaves = slaves.size();
        final int originalTaskScale = (Integer) params[0];
        final int subTaskScale = originalTaskScale / numSlaves;
        final int subTasksCount = (0 == (originalTaskScale % numSlaves)) ?
            numSlaves : numSlaves + 1;
        TaskDivideStrategy<Range> tds = new TaskDivideStrategy<Range>() {
            private int i = 1;

            @Override
            public Range nextChunk() {
                int upperBound;
                if (i < subTasksCount) {
                    upperBound = i * subTaskScale;
                } else if (i == subTasksCount) {
                    upperBound = originalTaskScale;
                } else {
                    return null;
                }

                int lowerBound = (i - 1) * subTaskScale + 1;
                i++;

                return new Range(lowerBound, upperBound);
            }
        };
        return tds;
    }

    // 创建Slave线程
    @Override
    protected Set<? extends SlaveSpec<Range, Set<BigInteger>>> createSlaves() {
        Set<PrimeGenerator> slaves = new HashSet<PrimeGenerator>();
        for (int i = 0; i < Runtime.getRuntime().availableProcessors(); i++) {
            slaves.add(new PrimeGenerator(new ArrayBlockingQueue<Runnable>(2)));
        }
        return slaves;
    }

    // 组合子任务的处理结果
    @Override
    protected Set<BigInteger> combineResults(
        Iterator<Future<Set<BigInteger>>> subResults) {

        Set<BigInteger> result = new TreeSet<BigInteger>();

        while (subResults.hasNext()) {
            try {
                result.addAll(subResults.next().get());
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                Throwable cause = e.getCause();
                if (SubTaskFailureException.class.isInstance(cause)) {
                    @SuppressWarnings("rawtypes")
                    RetryInfo retryInfo = ((SubTaskFailureException)
                        cause).retryInfo;
                    Object subTask = retryInfo.subTask;
                    Debug.info("failed subTask:" + subTask);
                    e.printStackTrace();
                }
            }
        }

        return result;
    }

    // 使Master子类对外保留一个含义具体的方法名
    public Set<BigInteger> generatePrime(int upperBound) throws Exception {
        return this.service(upperBound);
    }
}

```

Slave的客户代码实现

	<pre> WorkerThreadSlave&lt;Range, Set&lt;BigInteger&gt;&gt; {      public PrimeGenerator(BlockingQueue&lt;Runnable&gt; taskQueue) {         super(taskQueue);     }      @Override     protected Set&lt;BigInteger&gt; doProcess(Range range) throws Exception {         Set&lt;BigInteger&gt; result = new TreeSet&lt;BigInteger&gt;();         BigInteger start = BigInteger.valueOf(range.lowerBound);         BigInteger end = BigInteger.valueOf(range.upperBound);          while (-1 == (start = start.nextProbablePrime()).compareTo(end)) {             result.add(start);         }          return result;     } } </pre>	
	<pre> class Range {     public final int lowerBound;     public final int upperBound;     public Range(int lowerBound, int upperBound) {         if (upperBound &lt; lowerBound) {             throw new IllegalArgumentException(                 "upperBound should not be less than lowerBound!");         }         this.lowerBound = lowerBound;         this.upperBound = upperBound;     }      @Override     public String toString() {         return "Range [" + lowerBound + ", " + upperBound + "]";     } } </pre>	辅助类

例子2	<p><b>背景</b> 某基于Web Service的电信系统需要一个系统流量统计工具。该工具的统计依据是该系统运行过程中产生的接口日志文件。接口日志文件记录了该系统接收到的请求、该系统返回给客户端的响应以及该系统调用外部系统时涉及的请求和响应的相关数据。其格式如下：</p> <p>操作时间戳 协议类型(SOAP/REST/HTTP) 记录类型(请求/响应) 接口名称 操作名称 源设备名 目标设备名 消息唯一标识 本机IP地址 主叫号码 被叫号码</p> <p>图12-3展示了一个示例接口日志文件。</p> <p>2015-02-01 14:13:45.039 SOAP request SMS sendSms OSG ESB 00200033375 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.067 SOAP response SMS sendSmsRsp ESB OSG 00200033375 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.100 SOAP request SMS sendSms ESB NIC 00210033376 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.396 SOAP response SMS sendSmsRsp NIC ESB 00210033379 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.047 REST request Location getLocation OSG ESB 00200008326 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.060 REST request Location getLocation ESB NIC 00210008327 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.113 REST response Location getLocationRsp NIC ESB 00210008330 192.168.1.102 13612345678 136712345670  2015-02-01 14:13:45.234 REST response Location getLocationRsp ESB OSG 00200008328 192.168.1.102 13612345678 136712345670</p> <p>图12-3. 示例接口日志文件</p> <p>另外，流量统计工具要支持统计指定时间段内的系统流量。因此，我们把该工具分为两个模块：Linux Shell脚本模块和Java模块。Linux Shell脚本模块根据指定的时间段查找相应的接口日志文件，并将所有符合要求的接口日志文件的文件名通过标准输出传递给Java模块。Java模块根据其标准输入(System.in)中指定的接口日志文件名读取相应的接口日志文件进行流量统计。</p> <p><b>思路</b> 首先，该系统的接口日志文件被统一存储在日志文件服务器上。有时我们可能需要在日志服务器上直接统计系统流量。因此，我们希望统计工具占用的系统资源(CPU时间和内存)尽可能地低，以免其运行干扰了日志服务器。</p> <p>其次，统计工具的统计依据——接口日志文件可能包含大量的记录。以该系统单节点的流量为100TPS(Transaction per Second, 即1s内系统能够处理的请求数量)，平均某个请求的处理会产生4条接口日志记录为例，那么1h会产生上千万条接口日志记录(<math>100 \times 3600 \times 4 = 1440000</math>)。因此，统计工具的统计速率要尽可能快，以免在急需统计结果的时候等待过久。</p> <p>当然，上述两个需求是矛盾的。我们只能在统计速率和资源消耗之间寻求一个平衡点。这里，我们可以使用Master-Slave模式：某个时间段内涉及的接口日志记录总数可能达上千万，因此原始任务的规模比较大。考虑到每个接口日志文件包含的记录个数最多为N条(如N为10000)，我们可以以文件为单位进行任务分解。将若干个日志文件合为一个子任务，并利用专门的工作者线程去执行这些子任务。再汇总各个子任务的统计结果便可得到我们所需要的最终结果。</p>
前提	<p>清单12-1中的类Master相当于Master-Slave模式中的Master参与者实例，它将接口日志文件派发给Slave线程处理。类Master的calculate方法相当于图12-1中的service方法。类Master的dispatchTask方法对原始任务进行分解，并将分解得来的子任务派发给Slave线程处理。因此，calculate方法相当于图12-1中的splitWork方法和callSlaves方法。清单12-1中的类Slave相当于Master-Slave模式中的Slave参与者实例，它负责对接口日志文件中的记录进行统计。每个Slave实例都是一个可停止的线程，其submitWorkload方法相当于图12-1中的subService方法。本案例中，我们使用了一个java.util.concurrent.ConcurrentMap实例，用于存储各个子任务的处理结果。因此，Slave实例往ConcurrentMap实例存储其处理结果的过程也正是原始任务的处理结果的形成过程。</p> <p>本案例的主要计算集中在Slave线程上。为了避免Slave线程占用过多的资源，我们主要采取了两个措施。一个是，Master类所创建的Slave线程数量为JVM所在主机的CPU个数(通过Runtime.getRuntime().availableProcessors()获取)；另一个是Slave线程的doRun方法每处理完10万条记录就会休眠80ms，这是为了避免其使得CPU过于繁忙。</p> <p>清单12-1中的类Slave继承自AbstractTerminatableThread类。它使用了第5章介绍的Two-phase Termination模式。相关源码请见清单5-3。</p> <p>中的subService方法。本案例中，我们使用了一个java.util.concurrent.ConcurrentMap实例，用于存储各个子任务的处理结果。因此，Slave实例往ConcurrentMap实例存储其处理结果的过程也正是原始任务的处理结果的形成过程。</p> <p>本案例的主要计算集中在Slave线程上。为了避免Slave线程占用过多的资源，我们主要采取了两个措施。一个是，Master类所创建的Slave线程数量为JVM所在主机的CPU个数(通过Runtime.getRuntime().availableProcessors()获取)；另一个是Slave线程的doRun方法每处理完10万条记录就会休眠80ms，这是为了避免其使得CPU过于繁忙。</p> <p>清单12-1中的类Slave继承自AbstractTerminatableThread类。它使用了第5章介绍的Two-phase Termination模式。相关源码请见清单5-3。</p>
代码	<pre> public class TPSStat {     public static void main(String[] args) throws Exception {         // 接口日志文件所在目录         String logBaseDir = args[0];          // 忽略的操作名列表         String excludedOperationNames = "";          // 指定要统计在内的操作名列表         String includedOperationNames = "***";          // 指定要统计在内的目标设备名         String destinationSysName = "***";     } } </pre>

	<pre> public class TPSStat {     public static void main(String[] args) throws Exception {         // 接口日志文件所在目录         String logBaseDir = args[0];          // 忽略的操作名列表         String excludedOperationNames = "";          // 指定要统计在内的操作名列表         String includedOperationNames = "***";          // 指定要统计在内的目标设备名         String destinationSysName = "***";     } } </pre>	main()
--	---	--------

```

int argc = args.length;
if (argc > 2) {
    excludedOperationNames = args[1];
if (argc > 3) {
    excludedOperationNames = args[2];
}

if (argc > 4) {
    destinationSysName = args[3];
}

Master processor = new Master(logBaseDir, excludedOperationNames,
    includedOperationNames, destinationSysName);

BufferedReader fileNamesReader = new BufferedReader(new InputStreamReader(
    System.in));

ConcurrentMap<String, AtomicInteger> result = processor
    .calculate(fileNamesReader);

for (String timeRange : result.keySet()) {
    System.out.println(timeRange + "," + result.get(timeRange));
}
}

// 模式角色: Master-Slave.Master
private static class Master {
    private final String logFileBaseDir;
    private final String excludedOperationNames;
    private final String includedOperationNames;
    private final String destinationSysName;

    // 每次派发给某个Slave线程的文件个数
    private static final int NUMBER_OF_FILES_FOR_EACH_DISPATCH = 5;
    private static final int WORKER_COUNT = Runtime.getRuntime()
        .availableProcessors();

    public Master(String logFileBaseDir, String excludedOperationNames,
        String includedOperationNames, String destinationSysName) {
        this.logFileBaseDir = logFileBaseDir;
        this.excludedOperationNames = excludedOperationNames;
        this.includedOperationNames = includedOperationNames;
        this.destinationSysName = destinationSysName;
    }

    public ConcurrentMap<String, AtomicInteger> calculate()
        throws IOException {
        BufferedReader fileNamesReader = new
            ConcurrentMap<String, AtomicInteger> repository =
            new ConcurrentSkipListMap<String, AtomicInteger>();

        // 创建工作者线程
        Worker[] workers = createAndStartWorkers(repository);

        // 指派任务给工作者线程
        dispatchTask(fileNamesReader, workers);

        // 等待工作者线程处理结束
        for (int i = 0; i < WORKER_COUNT; i++) {
            workers[i].terminate(true);
        }
        // 返回处理结果
        return repository;
    }

    private Worker[] createAndStartWorkers(
        ConcurrentMap<String, AtomicInteger> repository) {
        Worker[] workers = new Worker[WORKER_COUNT];
        Worker worker;
        UncaughtExceptionHandler eh = new UncaughtExceptionHandler() {

            @Override
            public void uncaughtException(Thread t, Throwable e) {
                e.printStackTrace();
            }
        };

        for (int i = 0; i < WORKER_COUNT; i++) {
            worker = new Worker(repository, excludedOperationNames,
                includedOperationNames, destinationSysName);
            workers[i] = worker;
            worker.setUncaughtExceptionHandler(eh);
            worker.start();
        }

        return workers;
    }

    private void dispatchTask(BufferedReader fileNamesReader, Worker[] workers)
        throws IOException {
        String line;
        Set<String> fileNames = new HashSet<String>();

        int fileCount = 0;
        int workerIndex = -1;
        BufferedReader logFileReader;
        while ((line = fileNamesReader.readLine()) != null) {
            fileNames.add(line);
            fileCount++;
            if (0 == (fileCount % NUMBER_OF_FILES_FOR_EACH_DISPATCH)) {
                // 工作者线程间的负载均衡: 采用简单的轮询选择worker
                workerIndex = (workerIndex + 1) % WORKER_COUNT;
                logFileReader = makeReaderFrom(fileNames);
                Debug.info("Dispatch " + NUMBER_OF_FILES_FOR_EACH_DISPATCH
                    + " files to worker:" + workerIndex);
                workers[workerIndex].submitWorkload(logFileReader);
                fileNames = new HashSet<String>();
                fileCount = 0;
            }
        }
        if (fileCount > 0) {
    }
}

```

Master#calculate(...)

```

        logFileReader = makeReaderFrom(fileName);
        workerIndex = (workerIndex + 1) % WORKER_COUNT;
        workers[workerIndex].submitWorkload(logFileReader);
    }
}

private BufferedReader makeReaderFrom(final Set<String> logFileNames) {
    BufferedReader logFileReader;

    InputStream in = new SequenceInputStream(new Enumeration<InputStream>() {
        private Iterator<String> iterator = logFileNames.iterator();

        @Override
        public boolean hasMoreElements() {
            return iterator.hasNext();
        }

        @Override
        public InputStream nextElement() {
            String fileName = iterator.next();
            InputStream in = null;
            try {
                in = new FileInputStream(logFileBaseDir + fileName);
            } catch (FileNotFoundException e) {
                throw new RuntimeException(e);
            }
            return in;
        }
    });
    logFileReader = new BufferedReader(new InputStreamReader(in));
    return logFileReader;
}

private static class Worker extends AbstractTerminatableThread { // 模式角色: Master-Slave.Slave
    private static final Pattern SPLIT_PATTERN = Pattern.compile("\\\\|");
    private final ConcurrentMap<String, AtomicInteger> repository;
    private final BlockingQueue<BufferedReader> workQueue;

    private final String selfDevice = "ESB";
    private final String excludedOperationNames;
    private final String includedOperationNames;
    private final String destinationSysName;

    public Worker(ConcurrentMap<String, AtomicInteger> repository,
                 String excludedOperationNames, String includedOperationNames,
                 String destinationSysName) {
        this.repository = repository;
        workQueue = new ArrayBlockingQueue<BufferedReader>(100);
        this.excludedOperationNames = excludedOperationNames;
        this.includedOperationNames = includedOperationNames;
        this.destinationSysName = destinationSysName;
    }

    public void submitWorkload(BufferedReader taskWorkload) {
        try {
            workQueue.put(taskWorkload);
            terminationToken.reservations.incrementAndGet();
        } catch (InterruptedException e) {
        }
    }

    @Override
    protected void doRun() throws Exception {
        BufferedReader logFileReader = workQueue.take();

        String interfaceLogRecord;
        String[] recordParts;
        String timeStamp;
        AtomicInteger reqCounter;
        AtomicInteger existingReqCounter;
        int i = 0;

        try {
            while ((interfaceLogRecord = logFileReader.readLine()) != null) {
                recordParts = SPLIT_PATTERN.split(interfaceLogRecord, 0);

                // 避免CPU占用过高
                if (0 == ((++i) % 100000)) {
                    Thread.sleep(80);
                    i = 0;
                }

                // 跳过无效记录 (如果有的话)
                if (recordParts.length < 7) {
                    continue;
                }

                // 只考虑表示发送请求给selfDevice所指定的系统的记录
                if ("request".equals(recordParts[2]) &&
                    (recordParts[6].startsWith(selfDevice))) {
                    timeStamp = recordParts[0];

                    timeStamp = new String(timeStamp.substring(0,
                        19).toCharArray());

                    String opName = recordParts[4];
                    reqCounter = repository.get(timeStamp);
                    if (null == reqCounter) {

```

```

        reqCounter = new AtomicInteger(0);
        existingReqCounter = repository
            .putIfAbsent(timeStamp, reqCounter);
        if (null != existingReqCounter) {
            reqCounter = existingReqCounter;
        }
    }

    if (isSrcDeviceEEEligible(recordParts[5])) {
        if (excludedOperationNames.contains(openName + ',')) {
            continue;
        }

        if ("*". equals(includedOperationNames)) {
            reqCounter.incrementAndGet();
        } else {
            if (includedOperationNames.contains(openName +
            ',')) {
                reqCounter.incrementAndGet();
            }
        }
    }

} finally {
    terminationToken.reservations.decrementAndGet();
    logFileReader.close();
}

}

// 判断目标设备名是否在待统计之列
private boolean isSrcDeviceEEEligible(String sourceNE) {

    boolean result = false;

    if ("*". equals(destinationSysName)) {

        result = true;
    } else if (destinationSysName.equals(sourceNE)) {

        result = true;
    }

    return result;
}
}

```

中的类Slave相当于Master-Slave模式中的Slave参与者实例，它负责对接口日志文件中的记录进行统计。每个Slave实例都是一个可停止的线程，其submitWorkload方法相当于图12-1中的subService方法。本案例中，我们使用了一个java.util.concurrent.ConcurrentMap实例，用于存储各个子任务的处理结果。因此，Slave实例往ConcurrentMap实例存储其处理结果的过程也正是原始任务的处理结果的形成过程。

本案例的主要计算集中在Slave线程上。为了避免Slave线程占用过多的资源，我们主要采取了两个措施。一个是，Master类所创建的Slave线程数量为JVM所在主机的CPU个数（通过Runtime.getRuntime().availableProcessors() 获取）；另一个是Slave线程的doRun方法每处理完10万条记录就会休眠80ms，这是为了避免其使得CPU过于繁忙。