译者序

Storm 入门终于翻译完了。首先感谢并发编程网同意本人在网站上首发本书译文，同时还要感谢并发编程网的各位大牛们的耐心帮助。这是本人翻译的第一本书，其中必有各种不足请诸位读者朋友不吝斧正。

译完此书之后，我已经忘记了是如何知道的 Storm 这个工具了。本人读过的所有技术书籍大部分都是在地铁上完成的，现在已经成了习惯。最近发现自己有一阵子没有看书，那个时候大数据已经相当火热，我就想找一些讲大数据分析的书来读一读，虽然一直没有机会接触大数据的工作，不过做一些技术储备也是好的。于是上谷歌和亚马逊用“大数据”、“实时分析”这类关键词搜索相关的技术文章和书籍。然后就知道了 Storm，可惜一直没有找到中文的相关内容，只找到这一本《Getting Started with Storm》。可惜本人英文词汇量实在太少，书买来之后一直束之高阁，后来突发奇想我为什么不利用业余时间把这本书翻译了呢？于是由本人完成的《Getting Started with Storm》在并发编程网面世了。在本人之前已有人在 CSDN 上完成了本书除附录以外的全部翻译，并且有了 PDF 版。不过既然已经开始就不忍中途放弃了，所以一直坚持把本书译完。再次感谢并发编程网的朋友们的支持。

由于本人是持学习的目的翻译本书，对 Storm 的了解并不丰富，许多专用术语翻译难免不准确，如有谬误还请读者朋友们不吝指正。

本书基于最新的 Storm0.7.1 版本撰写，从 Storm 开发环境的搭建、Storm 工程的组成，到 Storm 各组件功能与开发，一步步的让读者入门并熟练掌握如何基于 Storm 的开发并利用 Storm 完成。

本书共分为八个章节和三个附录：

第一章介绍 Storm 的特性以及可能的应用场景。

第二章讲述了 Storm 的运行模式，Storm 工程包含的组件，以及如何创建一个 Storm 工程。

第三章对 Storm 的拓扑结构，各个组件如何分工协作做了详细介绍，数据流分组是本章重点。

第四章介绍 Storm 的数据源——spouts，Storm 的所有数据都从这里开始。

第五章介绍 Storm 处理数据的组件。

第六章以一个简单的 WEB 应用讲解如何 Storm 进行数据分析。

第七章以 PHP 为例讲述如何使用非 JVM 语言开发 Storm 工程。

第八章讲解支持事务的拓扑，当然不要把这里的事务跟关系型数据库的事务等同起来。

附录 A 安装 Storm 客户端，以及常用命令。

附录 B 安装与部署 Storm 集群。

附录 C 如何运行第六章的例子

全书目录如下：

章节目录

Storm 入门之第一章

译者注：本文翻译自《Getting Started With Storm》，本书中所有 Storm 相关术语都用斜体英文表示。 这些术语的字面意义翻译如下，由于这个工具的名字叫 Storm，这些术语一律按照气象名词解释

- *spout* 龙卷，读取原始数据为 *bolt* 提供数据
- *bolt* 雷电，从 *spout* 或其它 *bolt* 接收数据，并处理数据，处理结果可作为其它 *bolt* 的数据源或最终结果
- *nimbus* 雨云，主节点的守护进程，负责为工作节点分发任务。

下面的术语跟气象就没有关系了

- *topology* 拓扑结构，Storm 的一个任务单元
- *define field(s)* 定义域，由 *spout* 或 *bolt* 提供，被 *bolt* 接收

本文是该书的第一章。

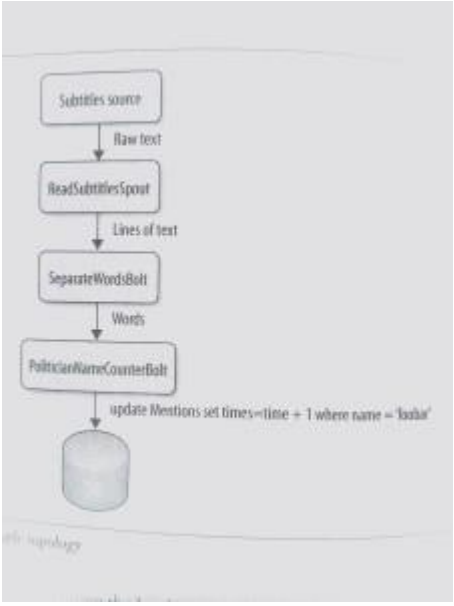
基础知识

Storm 是一个分布式的，可靠的，容错的数据流处理系统。它会把工作任务委托给不同类型的组件，每个组件负责处理一项简单特定的任务。Storm 集群的输入流由一个被称作 *spout* 的组件管理，*spout* 把数据传递给 *bolt*，*bolt* 要么把数据保存到某种存储器，要么把数据传递给其它的 *bolt*。你可以想象一下，一个 Storm 集群就是在一连串的 *bolt* 之间转换 *spout* *传*过来的数据。

这里用一个简单的例子来说明这个概念。昨晚我在新闻节目里看到主持人在谈论政治人物和他们对于各种政治话题的立场。他们一直重复着不同的名字，而我开始考虑这些名字是否被提到了相同的次数，以及不同次数之间的偏差。

想像播音员读的字幕作为你的数据输入流。你可以用一个 *spout* 读取一个文件（或者 socket，通过 HTTP，或者别的方法）。文本行被 *spout* 传给一个 *bolt*，再被 bolt 按单词切割。单词流又被传给另一个 *bolt*，在这里每个单词与一张政治人名列表比较。每遇到一个匹配的名字，第二个

bolt 为这个名字在数据库的计数加 1。你可以随时查询数据库查看结果， 而且这些计数是随着数据到达实时更新的。所有组件（*spouts* 和 *bolts*）及它们之间的关系请参考拓扑图 1-1



现在想象一下，很容易在整个 Storm 集群定义每个 *bolt* 和 *spout* 的并行性级别，因此你可以无限的扩展你的拓扑结构。很神奇，是吗？尽管这是个简单例子，你也可以看到 Storm 的强大。

有哪些典型的 Storm 应用案例？

数据处理流

正如上例所展示的，不像其它的流处理系统，Storm 不需要中间队列。

连续计算

连续发送数据到客户端，使它们能够实时更新并显示结果，如网站指标。

分布式远程过程调用

频繁的 CPU 密集型操作并行化。

Storm 组件

对于一个 Storm 集群，一个连续运行的主节点组织若干节点工作。

在 Storm 集群中，有两类节点：主节点 *master node* 和工作节点 *worker nodes*。主节点运行着一个叫做 *Nimbus* 的守护进程。这个守护进程负责在集群中分发代码，为工作节点分配任务，并监控故障。Supervisor 守护进程作为拓扑的一部分运行在工作节点上。一个 Storm 拓扑结构在不同的机器上运行着众多的工作节点。

因为 Storm 在 Zookeeper 或本地磁盘上维持所有的集群状态，守护进程可以是无状态的而且失效或重启时不会影响整个系统的健康（见图 1-2）

在系统底层，Storm 使用了 zeromq(0mq, zeromq(http://www.zeromq.org))。这是一种先进的，可嵌入的网络通讯库，它提供的绝妙功能使 Storm 成为可能。下面列出一些 zeromq 的特性。

- 一个并发架构的 Socket 库
- 对于集群产品和超级计算，比 TCP 要快
- 可通过 inproc（进程内）, IPC（进程间）, TCP 和 multicast(多播协议)通信
- 异步 I / O 的可扩展的多核消息传递应用程序
- 利用扇出(fanout)，发布订阅（PUB-SUB）,管道（pipeline），请求应答（REQ-REP），等方式实现 N-N 连接

NOTE: Storm 只用了 push/pull sockets

Storm 的特性

在所有这些设计思想与决策中，有一些非常棒的特性成就了独一无二的 Storm。

- 简化编程　如果你曾试着从零开始实现实时处理，你应该明白这是一件多么痛苦的事情。使用 Storm，复杂性被大大降低了。
- 使用一门基于 JVM 的语言开发会更容易，但是你可以借助一个小的中间件，在 Storm 上使用任何语言开发。有现成的中间件可供选择，当然也可以自己开发中间件。
- 容错　　Storm 集群会关注工作节点状态，如果宕机了必要的时候会重新分配任务。

- 可扩展　所有你需要为扩展集群所做的工作就是增加机器。Storm 会在新机器就绪时向它们分配任务。
- 可靠的　所有消息都可保证至少处理一次。如果出错了，消息可能处理不只一次，不过你永远不会丢失消息。
- 快速　速度是驱动 Storm 设计的一个关键因素
- 事务性　You can get exactly once messaging semantics for pretty much any computation.你可以为几乎任何计算得到恰好一次消息语义。

Storm 入门 第二章准备开始

准备开始

在本章，我们要创建一个 Storm 工程和我们的第一个 Storm 拓扑结构。

NOTE: 下面假设你的 JRE 版本在 1.6 以上。我们推荐 Oracle 提供的 JRE。你可以到 http://www.java .com/downloads/下载。

操作模式

开始之前，有必要了解一下 Storm 的操作模式。有下面两种方式。

本地模式

在本地模式下，Storm 拓扑结构运行在本地计算机的单一 JVM 进程上。这个模式用于开发、测试以及调试，因为这是观察所有组件如何协同工作的最简单方法。在这种模式下，我们可以调整参数，观察我们的拓扑结构如何在不同的 Storm 配置环境下运行。要在本地模式下运行，我们要下载 Storm 开发依赖，以便用来开发并测试我们的拓扑结构。我们创建了第一个 Storm 工程以后，很快就会明白如何使用本地模式了。

NOTE：在本地模式下，跟在集群环境运行很像。不过很有必要确认一下所有组件都是线程安全的，因为当把它们部署到远程模式时它们可能会运行在不同的 JVM 进程甚至不同的物理机上，这个时候它们之间没有直接的通讯或共享内存。

我们要在本地模式运行本章的所有例子。

远程模式

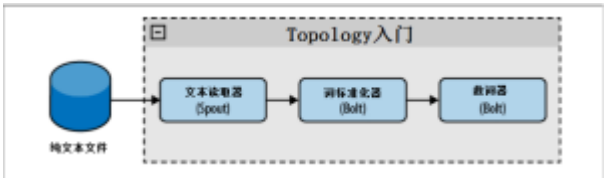
在远程模式下，我们向 Storm 集群提交拓扑，它通常由许多运行在不同机器上的流程组成。远程模式不会出现调试信息， 因此它也称作生产模式。不过在单一开发机上建立一个 Storm 集群是一个好主意，可以在部署到生产环境之前，用来确认拓扑在集群环境下没有任何问题。

你将在第六章学到更多关于远程模式的内容，并在附录 B 学到如何安装一个 Storm 集群。

Hello World

我们在这个工程里创建一个简单的拓扑，数单词数量。我们可以把这个看作 Storm 的“Hello World”。不过，这是一个非常强大的拓扑，因为它能够扩展到几乎无限大的规模，而且只需要做一些小修改，就能用它构建一个统计系统。举个例子，我们可以修改一下工程用来找出 Twitter 上的热点话题。

要创建这个拓扑，我们要用一个 *spout* 读取文本，第一个 *bolt* 用来标准化单词，第二个 *bolt* 为单词计数，如图 2-1 所示。



你可以从这个网址下载源码压缩包， https://github.com/storm-book/examples-ch02-getting_started/zipball/master。

NOTE: 如果你使用 git（一个分布式版本控制与源码管理工具），你可以执行 git clone git@github.com:storm-book/examples-ch02-getting_started.git，把源码检出到你指定的目录。

Java 安装检查

构建 Storm 运行环境的第一步是检查你安装的 Java 版本。打开一个控制台窗口并执行命令：java -version。控制台应该会显示出类似如下的内容：

```
java -version



java version "1.6.0_26"

Java(TM) SE Runtime Enviroment (build 1.6.0_26-b03)



Java HotSpot(TM) Server VM (build 20.1-b02, mixed mode)
```

如果不是上述内容，检查你的 Java 安装情况。（参考 http://www.java.com/download/）

创建工程

开始之前，先为这个应用建一个目录（就像你平常为 Java 应用做的那样）。这个目录用来存放工程源码。

接下来我们要下载 Storm 依赖包，这是一些 jar 包，我们要把它们添加到应用类路径中。你可以采用如下两种方式之一完成这一步：

- 下载所有依赖，解压缩它们，把它 们添加到类路径
- 使用 *Apache Maven*

NOTE: Maven 是一个软件项目管理的综合工具。它可以用来管理项目的开发周期的许多方面，从包依赖到版本发布过程。在这本书中，我们将广泛使用它。如果要检查是否已经安装了 maven，在命令行运行 mvn。如果没有安装你可以从 http://maven.apache.org/download.html 下载。

没有必要先成为一个 Maven 专家才能使用 Storm，不过了解一下关于 Maven 工作方式的基础知识仍然会对你有所帮助。你可以在 Apache Maven 的网站上找到更多的信息（http://maven.apache.org/）。

NOTE: Storm 的 Maven 依赖引用了运行 Storm 本地模式的所有库。

要运行我们的拓扑，我们可以编写一个包含基本组件的 pom.xml 文件。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
         <modelVersion>4.0.0</modelVersion>
         <groupId>storm.book</groupId>
         <artifactId>Getting-Started</artifactId>
         <version>0.0.1-SNAPSHOT</version>
         <build>
             <plugins>
                 <plugin>
                     <groupId>org.apache.maven.plugins</groupId>
                     <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>2.3.2</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                    <compilerVersion>1.6</compilerVersion>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <repositories>
        <!-- Repository where we can found the storm dependencies -->
        <repository>
            <id>clojars.org</id>
            <url>http://clojars.org/repo</url>
        </repository>
    </repositories>
    <dependencies>
        <!-- Storm Dependency -->
        <dependency>
            <groupId>storm</groupId>
            <artifactId>storm</artifactId>
```

```
                    <version>0.6.0</version>
                </dependency>
            </dependencies>
    </project>
```

开头几行指定了工程名称和版本号。然后我们添加了一个编译器插件，告知 Maven 我们的代码要用 Java1.6 编译。接下来我们定义了 Maven 仓库（Maven 支持为同一个工程指定多个仓库）。clojars 是存放 Storm 依赖的仓库。Maven 会为运行本地模式自动下载必要的所有子包依赖。

一个典型的 Maven Java 工程会拥有如下结构：

```
    我们的应用目录/
            ├─── pom.xml
            └─── src
                 └─── main
                    └─── java
                 |  ├─── spouts
                 |  └─── bolts
                 └─── resources
```

java 目录下的子目录包含我们的代码，我们把要统计单词数的文件保存在 resource 目录下。

NOTE：命令 mkdir -p 会创建所有需要的父目录。

创建我们的第一个 Topology

我们将为运行单词计数创建所有必要的类。可能这个例子中的某些部分，现在无法讲的很清楚，不过我们会在随后的章节做进一步的讲解。

Spout

pout WordReader 类实现了 IRichSpout 接口。我们将在第四章看到更多细节。WordReader 负责从文件按行读取文本，并把文本行提供给第一个 *bolt*。

NOTE: 一个 *spout* 发布一个定义域列表。这个架构允许你使用不同的 *bolts* 从同一个 *spout* 流读取数据，它们的输出也可作为其它 *bolts* 的定义域，以此类推。

例 2-1 包含 WordRead 类的完整代码（我们将会分析下述代码的每一部分）。

```
/**
 *  例 2-1.src/main/java/spouts/WordReader.java
 */
package spouts;


import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Map;
import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichSpout;
import backtype.storm.topology.OutputFieldsDeclarer;
```

```
import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;



public class WordReader implements IRichSpout {

    private SpoutOutputCollector collector;

    private FileReader fileReader;

    private boolean completed = false;

    private TopologyContext context;

    public boolean isDistributed() {return false;}

    public void ack(Object msgId) {

            System.out.println("OK:"+msgId);

    }

    public void close() {}

    public void fail(Object msgId) {

         System.out.println("FAIL:"+msgId);

    }

    /**

     * 这个方法做的惟一一件事情就是分发文件中的文本行

     */

    public void nextTuple() {

    /**
```

```
     * 这个方法会不断的被调用，直到整个文件都读完了，我们将等待并返回。
     */
        if(completed){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                //什么也不做
            }
           return;
        }
        String str;
        //创建 reader
        BufferedReader reader = new BufferedReader(fileReader);
        try{
            //读所有文本行
           while((str = reader.readLine()) != null){
            /**
             * 按行发布一个新值
             */
                this.collector.emit(new Values(str),str);
            }
```

```
                }catch(Exception e){
                    throw new RuntimeException("Error reading tuple",e);
                }finally{
                    completed = true;
                }
            }
            /**
             * 我们将创建一个文件并维持一个 collector 对象
             */
            public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector) {
                    try {
                        this.context = context;
                        this.fileReader = new
FileReader(conf.get("wordsFile").toString());
                    } catch (FileNotFoundException e) {
                        throw new RuntimeException("Error reading file
["+conf.get("wordFile")+"]");
                    }
                    this.collector = collector;
            }
            /**
```

```
             * 声明输入域"word"
             */
            public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields("line"));
            }
        }
```

第一个被调用的 *spout* 方法都是 **public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)**。它接收如下参数：配置对象，在定义 topology 对象是创建；TopologyContext 对象，包含所有拓扑数据；还有 SpoutOutputCollector 对象，它能让我们发布交给 *bolts* 处理的数据。下面的代码主是这个方法的实现。

```
    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector collector) {
        try {
            this.context = context;
            this.fileReader = new FileReader(conf.get("wordsFile").toString());
        } catch (FileNotFoundException e) {
            throw new RuntimeException("Error reading file
["+conf.get("wordFile")+"]");
        }
        this.collector = collector;
    }
```

我们在这个方法里创建了一个 FileReader 对象，用来读取文件。接下来我们要实现 **public void nextTuple()**，我们要通过它向 *bolts* 发布待处理的数据。在这个例子里，这个方法要读取文件并逐行发布数据。

```
    public void nextTuple() {
        if(completed){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                //什么也不做
            }
            return;
        }
        String str;
        BufferedReader reader = new BufferedReader(fileReader);
        try{
            while((str = reader.readLine()) != null){
                this.collector.emit(new Values(str));
            }
        }catch(Exception e){
            throw new RuntimeException("Error reading tuple",e);
        }finally{
```

```
            completed = true;
        }
    }
```

NOTE: Values 是一个 ArrarList 实现，它的元素就是传入构造器的参数。

nextTuple()会在同一个循环内被 **ack()**和 **fail()**周期性的调用。没有任务时它必须释放对线程的控制，其它方法才有机会得以执行。因此 nextTuple 的第一行就要检查是否已处理完成。如果完成了，为了降低处理器负载，会在返回前休眠一毫秒。如果任务完成了，文件中的每一行都已被读出并分发了。

NOTE:元组(tuple)是一个具名值列表，它可以是任意 java 对象（只要它是可序列化的）。默认情况，Storm 会序列化字符串、字节数组、ArrayList、HashMap 和 HashSet 等类型。

Bolts

现在我们有了一个 *spout*，用来按行读取文件并每行发布一个*元组*，还要创建两个 *bolts*，用来处理它们（看图 2-1）。*bolts* 实现了接口 **backtype.storm.topology.IRichBolt**。

bolt 最重要的方法是 **void execute(Tuple input)**，每次接收到元组时都会被调用一次，还会再发布若干个元组。

NOTE: 只要必要，*bolt* 或 *spout* 会发布若干元组。当调用 **nextTuple** 或 **execute** 方法时，它们可能会发布 0 个、1 个或许多个元组。你将在第五章学习更多这方面的内容。

第一个 *bolt*，**WordNormalizer**，负责得到并标准化每行文本。它把文本行切分成单词，大写转化成小写，去掉头尾空白符。

首先我们要声明 *bolt* 的出参：

```
    public void declareOutputFields(OutputFieldsDeclarer declarer){
```

```
        declarer.declare(new Fields("word"));
    }
```

这里我们声明 *bolt* 将发布一个名为“word”的域。

下一步我们实现 **public void execute(Tuple input)**，处理传入的元组：

```
    public void execute(Tuple input){
        String sentence=input.getString(0);
        String[] words=sentence.split(" ");
        for(String word : words){
            word=word.trim();
            if(!word.isEmpty()){
                word=word.toLowerCase();
                //发布这个单词
                collector.emit(new Values(word));
            }
        }
        //对元组做出应答
        collector.ack(input);
    }
```

第一行从元组读取值。值可以按位置或名称读取。接下来值被处理并用 collector 对象发布。最后，每次都调用 collector 对象的 **ack()**方法确认已成功处理了一个元组。

例 2-2 是这个类的完整代码。

```
//例 2-2 src/main/java/bolts/WordNormalizer.java
package bolts;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
public class WordNormalizer implements IRichBolt{
    private OutputCollector collector;
    public void cleanup(){}
    /**
      * *bolt*从单词文件接收到文本行，并标准化它。
      * 文本行会全部转化成小写，并切分它，从中得到所有单词。
     */
    public void execute(Tuple input){
```

```
            String sentence = input.getString(0);
            String[] words = sentence.split(" ");
            for(String word : words){
                word = word.trim();
                if(!word.isEmpty()){
                    word=word.toLowerCase();
                    //发布这个单词
                    List a = new ArrayList();
                    a.add(input);
                    collector.emit(a,new Values(word));
                }
            }
            //对元组做出应答
            collector.ack(input);
        }
        public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
            this.collector=collector;
        }

        /**
```

```
         * 这个*bolt*只会发布“word”域
         */
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
    }
```

NOTE:通过这个例子，我们了解了在一次 **execute** 调用中发布多个元组。如果这个方法在一次调用中接收到句子“This is the Storm book”，它将会发布五个元组。

下一个 *bolt*，**WordCounter**，负责为单词计数。这个拓扑结束时（**cleanup()**方法被调用时），我们将显示每个单词的数量。

NOTE: 这个例子的 *bolt* 什么也没发布，它把数据保存在 map 里，但是在真实的场景中可以把数据保存到数据库。

```
package bolts;



import java.util.HashMap;
import java.util.Map;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.OutputFieldsDeclarer;
```

```
import backtype.storm.tuple.Tuple;



public class WordCounter implements IRichBolt{

    Integer id;

    String name;

    Map<String,Integer> counters;

    private OutputCollector collector;



    /**

      * 这个 spout 结束时（集群关闭的时候），我们会显示单词数量

      */

    @Override

    public void cleanup(){

        System.out.println("-- 单词数 【"+name+"-"+id+"】 --");

        for(Map.Entry<String,Integer> entry : counters.entrySet()){

            System.out.println(entry.getKey()+": "+entry.getValue());

        }

    }



    /**

     *  为每个单词计数
```

```
     */
    @Override
    public void execute(Tuple input) {
        String str=input.getString(0);
        /**
         * 如果单词尚不存在于 map，我们就创建一个，如果已在，我们就为它加 1
         */
        if(!counters.containsKey(str)){
            conters.put(str,1);
        }else{
            Integer c = counters.get(str) + 1;
            counters.put(str,c);
        }
        //对元组作为应答
        collector.ack(input);
    }

    /**
     * 初始化
     */
    @Override
```

```
    public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector){
        this.counters = new HashMap<String, Integer>();
        this.collector = collector;
        this.name = context.getThisComponentId();
        this.id = context.getThisTaskId();
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {}
}
```

execute 方法使用一个 map 收集单词并计数。拓扑结束时，将调用 **clearup()**方法打印计数器 map。（虽然这只是一个例子，但是通常情况下，当拓扑关闭时，你应当使用 **cleanup()**方法关闭活动的连接和其它资源。）

主类

你可以在主类中创建拓扑和一个本地集群对象，以便于在本地测试和调试。**LocalCluster** 可以通过 **Config** 对象，让你尝试不同的集群配置。比如，当使用不同数量的工作进程测试你的拓扑时，如果不小心使用了某个全局变量或类变量，你就能够发现错误。（更多内容请见第三章）

NOTE：所有拓扑节点的各个进程必须能够独立运行，而不依赖共享数据（也就是没有全局变量或类变量），因为当拓扑运行在真实的集群环境时，这些进程可能会运行在不同的机器上。

接下来，**TopologyBuilder** 将用来创建拓扑，它决定 Storm 如何安排各节点，以及它们交换数据的方式。

```
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("word-reader", new WordReader());

    builder.setBolt("word-normalizer", new
WordNormalizer()).shuffleGrouping("word-reader");

    builder.setBolt("word-counter", new
WordCounter()).shuffleGrouping("word-normalizer");
```

在 *spout* 和 *bolts* 之间通过 **shuffleGrouping** 方法连接。这种分组方式决定了 Storm 会以随机分配方式从源节点向目标节点发送消息。

下一步，创建一个包含拓扑配置的 **Config** 对象，它会在运行时与集群配置合并，并通过 **prepare** 方法发送给所有节点。

```
    Config conf = new Config();

    conf.put("wordsFile", args[0]);

    conf.setDebug(true);
```

由 *spout* 读取的文件的文件名，赋值给 **wordFile** 属性。由于是在开发阶段，设置 **debug** 属性为 **true**，Strom 会打印节点间交换的所有消息，以及其它有助于理解拓扑运行方式的调试数据。

正如之前讲过的，你要用一个 **LocalCluster** 对象运行这个拓扑。在生产环境中，拓扑会持续运行，不过对于这个例子而言，你只要运行它几秒钟就能看到结果。

```
    LocalCluster cluster = new LocalCluster();

    cluster.submitTopology("Getting-Started-Topologie", conf,
builder.createTopology());

    Thread.sleep(2000);
```

```
    cluster.shutdown();
```

调用 **createTopology** 和 **submitTopology**，运行拓扑，休眠两秒钟（拓扑在另外的线程运行），然后关闭集群。

例 2-3 是完整的代码

```
    //例 2-3 src/main/java/TopologyMain.java

    import spouts.WordReader;

    import backtype.storm.Config;

    import backtype.storm.LocalCluster;

    import backtype.storm.topology.TopologyBuilder;

    import backtype.storm.tuple.Fields;

    import bolts.WordCounter;

    import bolts.WordNormalizer;



    public class TopologyMain {

        public static void main(String[] args) throws InterruptedException {

        //定义拓扑

            TopologyBuilder builder = new TopologyBuilder());

            builder.setSpout("word-reader", new WordReader());

            builder.setBolt("word-normalizer", new
WordNormalizer()).shuffleGrouping("word-reader");
```

```
            builder.setBolt("word-counter", new
WordCounter(),2).fieldsGrouping("word-normalizer", new Fields("word"));



        //配置

            Config conf = new Config();

            conf.put("wordsFile", args[0]);

            conf.setDebug(false);



        //运行拓扑

             conf.put(Config.TOPOLOGY_MAX_SPOUT_PENDING, 1);

            LocalCluster cluster = new LocalCluster();

            cluster.submitTopology("Getting-Started-Topologie", conf,
builder.createTopology();

            Thread.sleep(1000);

            cluster.shutdown();

        }

    }
```

观察运行情况

你已经为运行你的第一个拓扑准备好了。在这个目录下面创建一个文件，**/src/main/resources/words.txt**，一个单词一行，然后用下面的命令运行这个拓扑：**mvn exec:java -Dexec.mainClass="TopologyMain" -Dexec.args="src/main/resources/words.txt**。

举个例子，如果你的 *words.txt* 文件有如下内容： **Storm test are great is an Storm simple application but very powerful really Storm is great** 你应该会在日志中看到类似下面的内容：**is: 2 application: 1 but: 1 great: 1 test: 1 simple: 1 Storm: 3 really: 1 are: 1 great: 1 an: 1 powerful: 1 very: 1** 在这个例子中，每类节点只有一个实例。但是如果你有一个非常大的日志文件呢？你能够很轻松的改变系统中的节点数量实现并行工作。这个时候，你就要创建两个 **WordCounter** 实例。

```
    builder.setBolt("word-counter", new
WordCounter(),2).shuffleGrouping("word-normalizer");
```

程序返回时，你将看到： **– 单词数 【word-counter-2】 — application: 1 is: 1 great: 1 are: 1 powerful: 1 Storm: 3 — 单词数 [word-counter-3] — really: 1 is: 1 but: 1 great: 1 test: 1 simple: 1 an: 1 very: 1** 棒极了！修改并行度实在是太容易了（当然对于实际情况来说，每个实例都会运行在单独的机器上）。不过似乎有一个问题：单词 *is* 和 *great* 分别在每个 **WordCounter** 各计数一次。怎么会这样？当你调用 **shuffleGrouping** 时，就决定了 Storm 会以随机分配的方式向你的 *bolt* 实例发送消息。在这个例子中，理想的做法是相同的单词问题发送给同一个 **WordCounter** 实例。你把 **shuffleGrouping(“word-normalizer”)**换成 **fieldsGrouping(“word-normalizer”, new Fields(“word”))**就能达到目的。试一试，重新运行程序，确认结果。 你将在后续章节学习更多分组方式和消息流类型。

结论

我们已经讨论了 Storm 的本地和远程操作模式之间的不同，以及 Storm 的强大和易于开发的特性。你也学习了一些 Storm 的基本概念，我们将在后续章节深入讲解它们。

Storm 入门之第三章拓扑

在这一章，你将学到如何在同一个 Storm 拓扑结构内的不同组件之间传递元组，以及如何向一个运行中的 Storm 集群发布一个拓扑。

数据流组

设计一个拓扑时，你要做的最重要的事情之一就是定义如何在各组件之间交换数据（数据流是如何被 *bolts* 消费的）。一个*数据流组*指定了每个 *bolt* 会消费哪些数据流，以及如何消费它们。

NOTE：一个节点能够发布一个以上的数据流，一个数据流组允许我们选择接收哪个。

数据流组在定义拓扑时设置，就像我们在第二章看到的：

```
...

    builder.setBolt("word-normalizer", new WordNormalizer())

            .shuffleGrouping("word-reader");

...
```

在前面的代码块里，一个 *bolt* 由 **TopologyBuilder** 对象设定， 然后使用随机数据流组指定数据源。数据流组通常将数据源组件的 ID 作为参数，取决于数据流组的类型不同还有其它可选参数。

NOTE：每个 **InputDeclarer** 可以有一个以上的数据源，而且每个数据源可以分到不同的组。

随机数据流组

随机流组是最常用的数据流组。它只有一个参数（数据源组件），并且数据源会向随机选择的 *bolt* 发送元组，保证每个消费者收到近似数量的元组。

随机数据流组用于数学计算这样的原子操作。然而，如果操作不能被随机分配，就像第二章为单词计数的例子，你就要考虑其它分组方式了。

域数据流组

域数据流组允许你基于元组的一个或多个域控制如何把元组发送给 *bolts*。它保证拥有相同域组合的值集发送给同一个 *bolt*。回到单词计数器的例子，如果你用 *word* 域为数据流分组，**word-normalizer** *bolt* 将只会把相同单词的元组发送给同一个 **word-counter***bolt* 实例。

```
• • •

    builder.setBolt("word-counter", new WordCounter(),2)

            .fieldsGrouping("word-normalizer", new Fields("word"));

• • •
```

NOTE: 在域数据流组中的所有域集合必须存在于数据源的域声明中。

全部数据流组

全部数据流组，为每个接收数据的实例复制一份元组副本。这种分组方式用于向 *bolts* 发送信号。比如，你要刷新缓存，你可以向所有的 *bolts* 发送一个*刷新缓存信号*。在单词计数器的例子里，你可以使用一个全部数据流组，添加清除计数器缓存的功能（见拓扑示例）

```
    public void execute(Tuple input) {

        String str = null;

        try{

            if(input.getSourceStreamId().equals("signals")){

                str = input.getStringByField("action");

                if("refreshCache".equals(str))
```

```
                counters.clear();

            }

        }catch (IllegalArgumentException e){

            //什么也不做

        }

        . . .

    }
```

我们添加了一个 if 分支，用来检查源数据流。Storm 允许我们声明具名数据流（如果你不把元组发送到一个具名数据流，默认发送到名为”**default**“的数据流）。这是一个识别元组的极好的方式，就像这个例子中，我们想识别 **signals** 一样。 在拓扑定义中，你要向 **word-counter** *bolt* 添加第二个数据流，用来接收从 **signals-spout** 数据流发送到所有 *bolt* 实例的每一个元组。

```
    builder.setBolt("word-counter", new WordCounter(),2)

            .fieldsGroupint("word-normalizer",new Fields("word"))

            .allGrouping("signals-spout","signals");
```

signals-spout 的实现请参考 git 仓库。

自定义数据流组

你可以通过实现 **backtype.storm.grouping.CustormStreamGrouping** 接口创建自定义数据流组，让你自己决定哪些 *bolt* 接收哪些元组。

让我们修改单词计数器示例，使首字母相同的单词由同一个 *bolt* 接收。

```
    public class ModuleGrouping mplents CustormStreamGrouping, Serializable{
```

```
        int numTasks = 0;



        @Override
        public List<Integer> chooseTasks(List<Object> values) {
            List<Integer> boltIds = new ArrayList<Integer>();
            if(values.size()>0){
                String str = values.get(0).toString();
                if(str.isEmpty()){
                    boltIds.add(0);
                }else{
                    boltIds.add(str.charAt(0) % numTasks);
                }
            }
            return boltIds;
        }



        @Override
        public void prepare(TopologyContext context, Fields outFields, List<Integer>
targetTasks) {
            numTasks = targetTasks.size();
        }
```

```
    }
```

这是一个 **CustomStreamGrouping** 的简单实现，在这里我们采用单词首字母字符的整数值与任务数的余数，决定接收元组的 *bolt*。

按下述方式 **word-normalizer** 修改即可使用这个自定义数据流组。

```
    builder.setBolt("word-normalizer", new WordNormalizer())

            .customGrouping("word-reader", new ModuleGrouping());
```

直接数据流组

这是一个特殊的数据流组，数据源可以用它决定哪个组件接收元组。与前面的例子类似，数据源将根据单词首字母决定由哪个 *bolt* 接收元组。要使用直接数据流组，在 **WordNormalizer** *bolt* 中，使用 **emitDirect** 方法代替 **emit**。

```
    public void execute(Tuple input) {

        ...

        for(String word : words){

            if(!word.isEmpty()){

                ...

                collector.emitDirect(getWordCountIndex(word),new Values(word));

            }

        }

        //对元组做出应答

        collector.ack(input);
```

```
    }


    public Integer getWordCountIndex(String word) {
        word = word.trim().toUpperCase();
        if(word.isEmpty()){
            return 0;
        }else{
            return word.charAt(0) % numCounterTasks;
        }
    }
```

在 **prepare** 方法中计算任务数

```
    public void prepare(Map stormConf, TopologyContext context,
                OutputCollector collector) {
        this.collector = collector;
        this.numCounterTasks = context.getComponentTasks("word-counter");
    }
```

在拓扑定义中指定数据流将被直接分组：

```
    builder.setBolt("word-counter", new WordCounter(),2)
            .directGrouping("word-normalizer");
```

全局数据流组

全局数据流组把所有数据源创建的元组发送给单一目标实例（即拥有最低 ID 的任务）。

不分组

写作本书时（Stom0.7.1 版），这个数据流组相当于随机数据流组。也就是说，使用这个数据流组时，并不关心数据流是如何分组的。

LocalCluster VS StormSubmitter

到目前为止，你已经用一个叫做 **LocalCluster** 的工具在你的本地机器上运行了一个拓扑。Storm 的基础工具，使你能够在自己的计算机上方便的运行和调试不同的拓扑。但是你怎么把自己的拓扑提交给运行中的 Storm 集群呢？Storm 有一个有趣的功能，在一个真实的集群上运行自己的拓扑是很容易的事情。要实现这一点，你需要把 **LocalCluster** 换成 **StormSubmitter** 并实现 **submitTopology** 方法， 它负责把拓扑发送给集群。

下面是修改后的代码：

```
    //LocalCluster cluster = new LocalCluster();
    //cluster.submitTopology("Count-Word-Topology-With-Refresh-Cache", conf,
    //builder.createTopology());
    StormSubmitter.submitTopology("Count-Word-Topology-With_Refresh-Cache", conf,
            builder.createTopology());
    //Thread.sleep(1000);
    //cluster.shutdown();
```

NOTE: 当你使用 **StormSubmitter** 时，你就不能像使用 **LocalCluster** 时一样通过代码控制集群了。

接下来，把源码压缩成一个 jar 包，运行 Storm 客户端命令，把拓扑提交给集群。如果你已经使用了 Maven， 你只需要在命令行进入源码目录运行：**mvn package**。

现在你生成了一个 jar 包，使用 **storm jar** 命令提交拓扑（关于如何安装 Storm 客户端请参考附录 A）。命令格式：**storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3**。

对于这个例子，在拓扑工程目录下面运行：

```
storm jar target/Topologies-0.0.1-SNAPSHOT.jar countword.TopologyMain
src/main/resources/words.txt
```

通过这些命令，你就把拓扑发布集群上了。

如果想停止或杀死它，运行：

```
storm kill Count-Word-Topology-With-Refresh-Cache
```

NOTE：拓扑名称必须保证惟一性。

NOTE：如何安装 Storm 客户端，参考附录 A

DRPC 拓扑

有一种特殊的拓扑类型叫做分布式远程过程调用（DRPC），它利用 Storm 的分布式特性执行远程过程调用（RPC）（见下图）。Storm 提供了一些用来实现 DRPC 的工具。第一个是 DRPC 服务器，它就像是客户端和 Storm 拓扑之间的连接器，作为拓扑的 *spout* 的数据源。它接收一个待执行的函数和函数参数，然后对于函数操作的每一个数据块，这个服务器都会通过拓扑分配一个请求 ID 用来识别 RPC 请求。拓扑执行最后的 *bolt* 时，它必须分配 RPC 请求 ID 和结果，使 DRPC 服务器把结果返回正确的客户端。

NOTE：单实例 DRPC 服务器能够执行许多函数。每个函数由一个惟一的名称标识。

Storm 提供的第二个工具（已在例子中用过）是 **LineDRPCTopologyBuilder**，一个辅助构建 DRPC 拓扑的抽象概念。生成的拓扑创建 **DRPCSpouts**——它连接到 DRPC 服务器并向拓扑的其它部分分发数据——并包装 *bolts*，使结果从最后一个 *bolt* 返回。依次执行所有添加到 **LinearDRPCTopologyBuilder** 对象的 *bolts*。

作为这种类型的拓扑的一个例子，我们创建了一个执行加法运算的进程。虽然这是一个简单的例子，但是这个概念可以扩展到复杂的分布式计算。

bolt 按下面的方式声明输出：

```
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id","result"));
    }
```

因为这是拓扑中惟一的 *bolt*，它必须发布 RPC ID 和结果。**execute** 方法负责执行加法运算。

```
    public void execute(Tuple input) {
        String[] numbers = input.getString(1).split("\\+");
        Integer added = 0;
        if(numbers.length<2){
            throw new InvalidParameterException("Should be at least 2 numbers");
        }
        for(String num : numbers){
            added += Integer.parseInt(num);
```

```
        }

        collector.emit(new Values(input.getValue(0),added));

    }
```

包含加法 *bolt* 的拓扑定义如下：

```
    public static void main(String[] args) {

        LocalDRPC drpc = new LocalDRPC();



        LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("add");

        builder.addBolt(AdderBolt(),2);



        Config conf = new Config();

        conf.setDebug(true);



        LocalCluster cluster = new LocalCluster();

        cluster.submitTopology("drpcder-topology", conf,

            builder.createLocalTopology(drpc));

        String result = drpc.execute("add", "1+-1");

        checkResult(result,0);

        result = drpc.execute("add", "1+1+5+10");

        checkResult(result,17);
```

```
        cluster.shutdown();
        drpc.shutdown();
    }
```

创建一个 **LocalDRPC** 对象在本地运行 DRPC 服务器。接下来，创建一个拓扑构建器（译者注：LineDRpctopologyBuilder 对象），把 *bolt* 添加到拓扑。运行 DRPC 对象（LocalDRPC 对象）的 **execute** 方法测试拓扑。

NOTE：使用 **DRPCClient** 类连接远程 DRPC 服务器。DRPC 服务器暴露了 Thrift API，因此可以跨语言编程；并且不论是在本地还是在远程运行 DRPC 服务器，它们的 API 都是相同的。 对于采用 Storm 配置的 DRPC 配置参数的 Storm 集群，调用构建器对象的 **createRemoteTopology** 向 Storm 集群提交一个拓扑，而不是调用 **createLocalTopology**。

Storm 入门之第四章 Spouts

你将在本章了解到 *spout* 作为拓扑入口和它的容错机制相关的最常见的设计策略。

可靠的消息 VS 不可靠的消息

在设计拓扑结构时，始终在头脑中记着的一件重要事情就是消息的可靠性。当有无法处理的消息时，你就要决定该怎么办，以及作为一个整体的拓扑结构该做些什么。举个例子，在处理银行存款时，不要丢失任何事务报文就是很重要的事情。但是如果你要统计分析数以百万的 tweeter 消息，即使有一条丢失了，仍然可以认为你的结果是准确的。

对于 Storm 来说，根据每个拓扑的需要担保消息的可靠性是开发者的责任。这就涉及到消息可靠性和资源消耗之间的权衡。高可靠性的拓扑必须管理丢失的消息，必然消耗更多资源；可靠性

较低的拓扑可能会丢失一些消息，占用的资源也相应更少。不论选择什么样的可靠性策略，Storm 都提供了不同的工具来实现它。

要在 *spout* 中管理可靠性，你可以在分发时包含一个元组的消息 ID（**collector.emit(new Values(…),tupleId)**）。在一个元组被正确的处理时调用 **ack** 方法，而在失败时调用 **fail** 方法。当一个元组被所有的靶 *bolt* 和锚 *bolt* 处理过，即可判定元组处理成功（你将在第 5 章学到更多锚 *bolt* 知识）。

发生下列情况之一时为元组处理失败：

- 提供数据的 *spout* 调用 **collector.fail(tuple)**
- 处理时间超过配置的超时时间

让我们来看一个例子。想象你正在处理银行事务，需求如下：

- 如果事务失败了，重新发送消息
- 如果失败了太多次，终结拓扑运行

创建一个 *spout* 和一个 *bolt*，*spout* 随机发送 100 个事务 ID，有 80%的元组不会被 *bolt* 收到（你可以在例子 ch04-spout 查看完整代码）。实现 *spout* 时利用 **Map** 分发事务消息元组，这样就比较容易实现重发消息。

```
public void nextTuple() {

    if(!toSend.isEmpty()){

        for(Map.Entry<Integer, String> transactionEntry : toSend.entrySet()){

            Integer transactionId = transactionEntry.getKey();

            String transactionMessage = transactionEntry.getValue();

            collector.emit(new Values(transactionMessage),transactionId);

        }
```

```
            toSend.clear();
        }
    }
```

如果有未发送的消息，得到每条事务消息和它的关联 ID，把它们作为一个元组发送出去，最后清空消息队列。值得一提的是，调用 map 的 **clear** 是安全的，因为 **nextTuple** 失败时，只有 **ack** 方法会修改 map，而它们都运行在一个线程内。

维护两个 map 用来跟踪待发送的事务消息和每个事务的失败次数。**ack** 方法只是简单的把事务从每个列表中删除。

```
public void ack(Object msgId) {
    messages.remove(msgId);
    failCounterMessages.remove(msgId);
}
```

fail 方法决定应该重新发送一条消息，还是已经失败太多次而放弃它。

NOTE:如果你使用全部数据流组，而拓扑里的所有 *bolt* 都失败了，*spout* 的 **fail** 方法才会被调用。

```
public void fail(Object msgId) {
    Integer transactionId = (Integer) msgId;
    //检查事务失败次数
    Integer failures = transactionFailureCount.get(transactionId) + 1;

    if(failes >= MAX_FAILS){
```

```
            //失败数太高了，终止拓扑
            throw new RuntimeException("错误, transaction id 【"+


             transactionId+"】 已失败太多次了 【"+failures+"】");
        }


        //失败次数没有达到最大数，保存这个数字并重发此消息
        transactionFailureCount.put(transactionId, failures);
        toSend.put(transactionId, messages.get(transactionId));
        LOG.info("重发消息【"+msgId+"】");
    }
```

首先，检查事务失败次数。如果一个事务失败次数太多，通过抛出 **RuntimeException** 终止发送此条消息的工人。否则，保存失败次数，并把消息放入待发送队列（**toSend**），它就会再次调用 **nextTuple** 时得以重新发送。

NOTE: Storm 节点不维护状态，因此如果你在内存保存信息（就像本例做的那样），而节点又不幸挂了，你就会丢失所有缓存的消息。

Storm 是一个快速失败的系统。拓扑会在抛出异常时挂掉，然后再由 Storm 重启，恢复到抛出异常前的状态。

获取数据

接下来你会了解到一些设计 *spout* 的技巧，帮助你从多数据源获取数据。

直接连接

在一个直接连接的架构中，spout 直接与一个消息分发器连接（见图 4-1）。

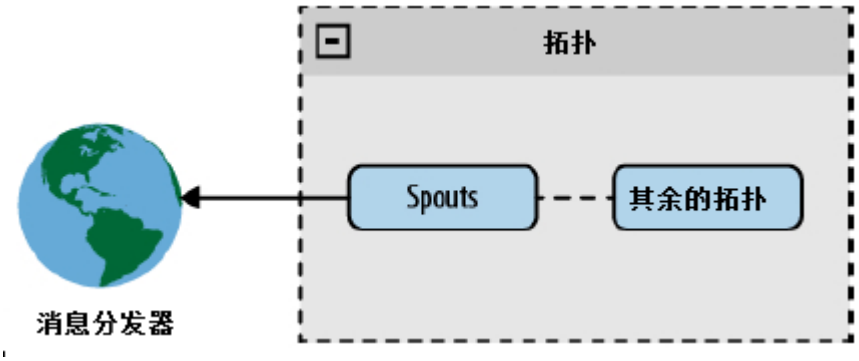


图 4-1 直接连接的 *spout*

这个架构很容易实现，尤其是在消息分发器是已知设备或已知设备组时。已知设备满足：拓扑从启动时就已知道该设备，并贯穿拓扑的整个生命周期保持不变。未知设备就是在拓扑运行期添加进来的。已知设备组就是从拓扑启动时组内所有设备都是已知的。

下面举个例子说明这一点。创建一个 spout 使用 Twitter 流 API 读取 twitter 数据流。spout 把 API 当作消息分发器直接连接。从数据流中得到符合 **track** 参数的公共 tweets（参考 twitter 开发页面）。完整的例子可以在链接 https://github.com/storm-book/examples-ch04-spouts/找到。

spout 从配置对象得到连接参数（**track**，**user**，**password**），并连接到 API（在这个例子中使用 Apache 的 DefaultHttpClient）。它一次读一行数据，并把数据从 JSON 转化成 Java 对象，然后发布它。

```
public void nextTuple() {
    //创建 http 客户端
    client = new DefaultHttpClient();
    client.setCredentialsProvider(credentialProvider);
    HttpGet get = new HttpGet(STREAMING_API_URL+track);
    HttpResponse response;
```

```
    try {
        //执行 http 访问
        response = client.execute(get);
        StatusLine status = response.getStatusLine();
        if(status.getStatusCode() == 200){
            InputStream inputStream = response.getEntity().getContent();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
            String in;
            //逐行读取数据
            while((in = reader.readLine())!=null){
                try{
                    //转化并发布消息
                    Object json = jsonParser.parse(in);
                    collector.emit(new Values(track,json));
                }catch (ParseException e) {
                    LOG.error("Error parsing message from twitter",e);
                }
            }
        }
    } catch (IOException e) {
```

```
        LOG.error("Error in communication with twitter api
["+get.getURI().toString()+"],
           sleeping 10s");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e1) {}
    }
}
```

NOTE:在这里你锁定了 **nextTuple** 方法，所以你永远也不会执行 **ack** 和 **fail** 方法。在真实的应用中，我们推荐你在一个单独的线程中执行锁定，并维持一个内部队列用来交换数据（你会在下一个例子中学到如何实现这一点：消息队列）。

棒极了！

现在你用一个 *spout* 读取 Twitter 数据。一个明智的做法是，采用拓扑并行化，多个 *spout* 从同一个流读取数据的不同部分。那么如果你有多个流要读取，你该怎么做呢？Storm 的第二个有趣的特性（译者注：第一个有趣的特性已经出现过，这句话原文都是一样的，不过按照中文的行文习惯还是不重复使用措词了）是，你可以在任意组件内（*spouts/bolts*）访问 **TopologyContext**。利用这一特性，你能够把流划分到多个 *spouts* 读取。

```
public void open(Map conf, TopologyContext context,
          SpoutOutputCollector collector) {
    //从 context 对象获取 spout 大小
    int spoutsSize =
context.getComponentTasks(context.getThisComponentId()).size();
```

```
    //从这个 spout 得到任务 id
    int myIdx = context.getThisTaskIndex();
    String[] tracks = ((String) conf.get("track")).split(",");
    StringBuffer tracksBuffer = new StringBuffer();
    for(int i=0; i< tracks.length;i++){
        //Check if this spout must read the track word
        if( i % spoutsSize == myIdx){
            tracksBuffer.append(",");
            tracksBuffer.append(tracks[i]);
        }
    }
    if(tracksBuffer.length() == 0) {
        throw new RuntimeException("没有为 spout 得到 track 配置" +
 " [spouts 大小:"+spoutsSize+", tracks:"+tracks.length+"] tracks 的数量必须高于 spout
的数量");
 this.track =tracksBuffer.substring(1).toString();
    }
 ...
 }
```

利用这一技巧，你可以把 collector 对象均匀的分配给多个数据源，当然也可以应用到其它的情形。比如说，从 web 服务器收集日志文件见图 4-2

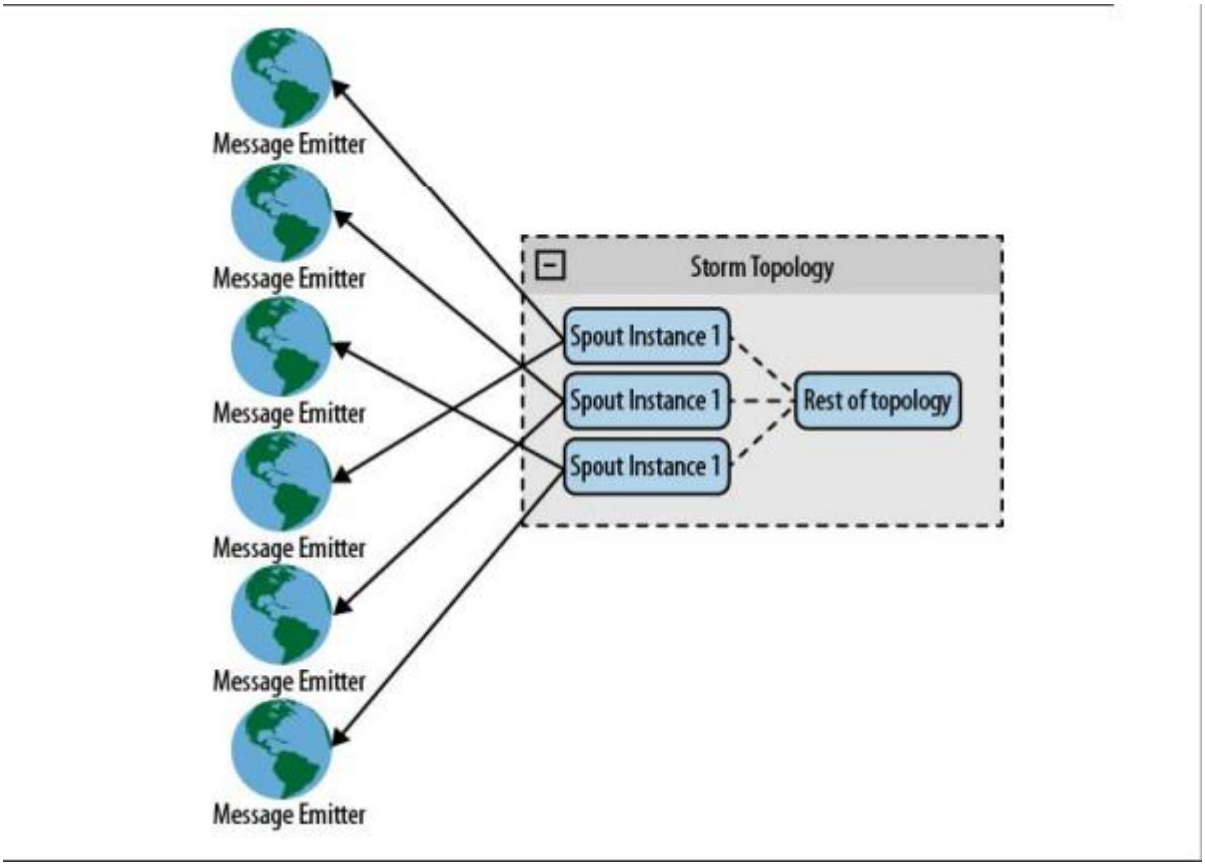


图 4-2 直连 hash

通过上一个例子，你学会了从一个 *spout* 连接到已知设备。你也可以使用相同的方法连接未知设备，不过这时你需要借助于一个协同系统维护的设备列表。协同系统负责探察列表的变化，并根据变化创建或销毁连接。比如，从 web 服务器收集日志文件时，web 服务器列表可能随着时间变化。当添加一台 web 服务器时，协同系统探查到变化并为它创建一个新的 *spout*。见图 4-3

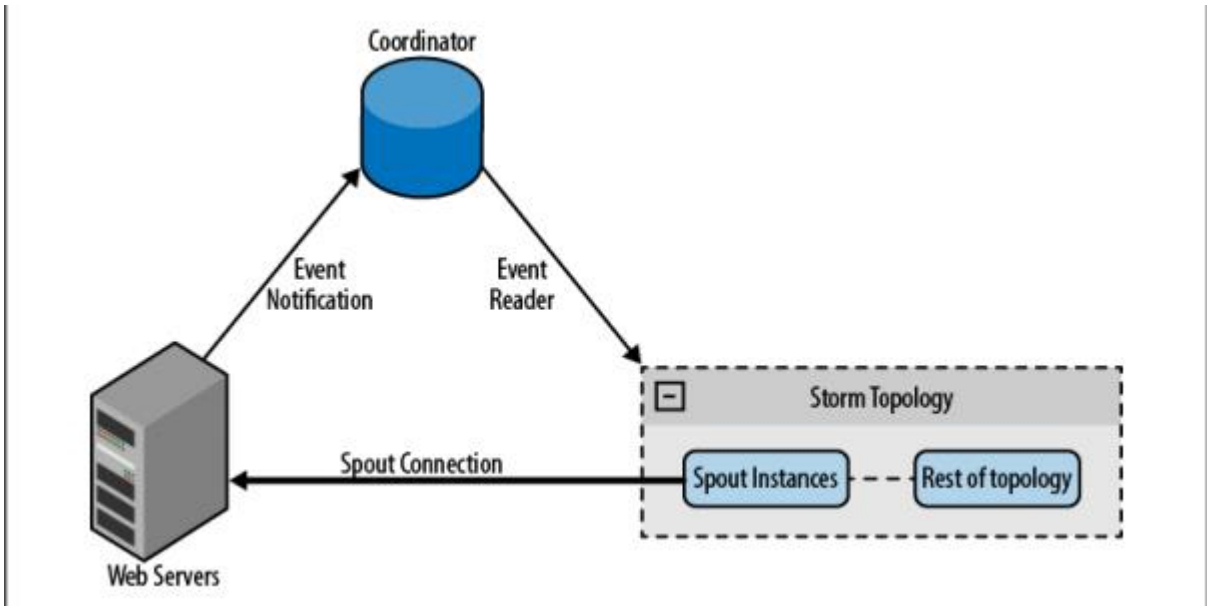


图 4-3 直连协同

消息队列

第二种方法是，通过一个队列系统接收来自消息分发器的消息，并把消息转发给 *spout*。更进一步的做法是，把队列系统作为 *spout* 和数据源之间的中间件，在许多情况下，你可以利用多队列系统的重播能力增强队列可靠性。这意味着你不需要知道有关消息分发器的任何事情，而且添加或移除分发器的操作比直接连接简单的多。这个架构的问题在于队列是一个故障点，另外你还要为处理流程引入新的环节。

图 4-4 展示了这一架构模型

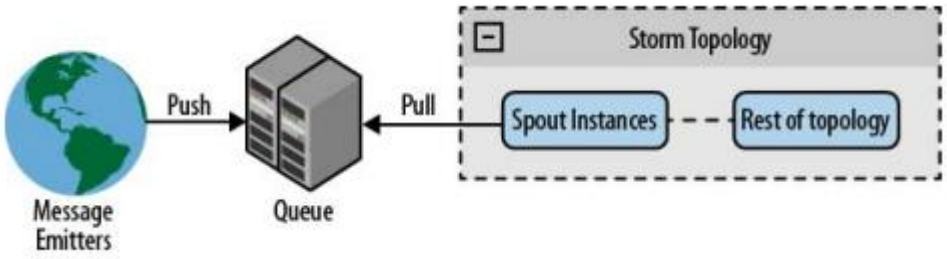


图 4-4 使用队列系统

NOTE:你可以通过轮询队列或哈希队列（把队列消息通过哈希发送给 *spouts* 或创建多个队列使队列 *spouts* 一一对应）在多个 *spouts* 之间实现并行性。

接下来我们利用 Redis 和它的 java 库 Jedis 创建一个队列系统。在这个例子中，我们创建一个日志处理器从一个未知的来源收集日志，利用 **lpush** 命令把消息插入队列，利用 **blpop** 命令等待消息。如果你有很多处理过程，blpop 命令采用了轮询方式获取消息。

我们在 *spout* 的 **open** 方法创建一个线程，用来获取消息（使用线程是为了避免锁定 **nextTuple** 在主循环的调用）：

```
new Thread(new Runnable() {

    @Override

    public void run() {

        try{

           Jedis client= new Jedis(redisHost, redisPort);

           List res = client.blpop(Integer.MAX_VALUE, queues);

           messages.offer(res.get(1));

        }catch(Exception e){

            LOG.error("从 redis 读取队列出错",e);

            try {

                Thread.sleep(100);

            }catch(InterruptedException e1){}

        }

    }
```

```
}).start();
```

这个线程的惟一目的就是，创建 redis 连接，然后执行 **blpop** 命令。每当收到了一个消息，它就被添加到一个内部消息队列，然后会被 **nextTuple** 消费。对于 *spout* 来说数据源就是 redis 队列，它不知道消息分发者在哪里也不知道消息的数量。

NOTE:我们不推荐你在 *spout* 创建太多线程，因为每个 spout 都运行在不同的线程。一个更好的替代方案是增加拓扑并行性，也就是通过 Storm 集群在分布式环境创建更多线程。

在 **nextTuple** 方法中，要做的惟一的事情就是从内部消息队列获取消息并再次分发它们。

```
public void nextTuple(){
    while(!messages.isEmpty()){
        collector.emit(new Values(messages.poll()));
    }
}
```

NOTE:你还可以借助 redis 在 *spout* 实现消息重发，从而实现可靠的拓扑。（译者注：这里是相对于开头的**可靠的消息 VS 不可靠的消息**讲的）

DRPC

DRPCSpout 从 DRPC 服务器接收一个函数调用，并执行它（见第三章的例子）。对于最常见的情况，使用 backtype.storm.drpc.DRPCSpout 就足够了，不过仍然有可能利用 Storm 包内的 DRPC 类创建自己的实现。

小结

现在你已经学习了常见的 *spout* 实现模式，它们的优势，以及如何确保消息可靠性。不存在适用于所有拓扑的架构模式。如果你知道数据源，并且能够控制它们，你就可以使用直接连接；然而如果你需要添加未知数据源或从多种数据源接收数据，就最好使用消息队列。如果你要执行在线过程，你可以使用 DRPCSpout 或类似的实现。

你已经学习了三种常见连接方式，不过依赖于你的需求仍然有无限的可能。

Storm 入门之第五章 Bolts

正如你已经看到的，*bolts* 是一个 Storm 集群中的关键组件。你将在这一章学到 *bolt* 生命周期，一些 *bolt* 设计策略，以及几个有关这些内容的例子。

Bolt 生命周期

Bolt 是这样一种组件，它把元组作为输入，然后产生新的元组作为输出。实现一个 *bolt* 时，通常需要实现 **IRichBolt** 接口。*Bolts* 对象由客户端机器创建，序列化为拓扑，并提交给集群中的主机。然后集群启动工人进程反序列化 *bolt*，调用 **prepare**，最后开始处理元组。

NOTE:要创建一个 bolt 对象，它通过构造器参数初始化成员属性，*bolt* 被提交到集群时，这些属性值会随着一起序列化。

Bolt 结构

Bolts 拥有如下方法：

```
declareOutputFields(OutputFieldsDeclarer declarer)
    为 bolt 声明输出模式
prepare(java.util.Map stormConf, TopologyContext context, OutputCollector collector)
```

```
    仅在 bolt 开始处理元组之前调用

execute(Tuple input)

    处理输入的单个元组

cleanup()

    在 bolt 即将关闭时调用
```

下面看一个例子，在这个例子中 *bolt* 把一句话分割成单词列表：

```
class SplitSentence implements IRichBolt {

    private OutputCollector collector;

    publlic void prepare(Map conf, TopologyContext context, OutputCollector collector)
{

        this.collector = collector;

    }



    public void execute(Tuple tuple) {

        String sentence = tuple.getString(0);

        for(String word : sentence.split(" ")) {

            collector.emit(new Values(word));

        }

    }
```

```
    public void cleanup(){}


    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

正如你所看到的，这是一个很简单的 *bolt*。值得一提的是在这个例子里，没有消息担保。这就意味着，如果 bolt 因为某些原因丢弃了一些消息——不论是因为 *bolt* 挂了，还是因为程序故意丢弃的——生成这条消息的 *spout* 不会收到任何通知，任何其它的 *spouts* 和 *bolts* 也不会收到。

然而在许多情况下，你想确保消息在整个拓扑范围内都被处理过了。

可靠的 *bolts* 和不可靠的 *bolts*

正如前面所说的，Storm 保证通过 *spout* 发送的每条消息会得到所有 *bolt* 的全面处理。基于设计上的考虑，这意味着你要自己决定你的 *bolts* 是否保证这一点。

拓扑是一个树型结构，消息（元组）穿过其中一条或多条分支。树上的每个节点都会调用 **ack(tuple)** 或 **fail(tuple)**，Storm 因此知道一条消息是否失败了，并通知那个/那些制造了这些消息的 *spout(s)*。既然一个 Storm 拓扑运行在高度并行化的环境里，跟踪始发 *spout* 实例的最好方法就是在消息元组内包含一个始发 spout 引用。这一技巧称做 *锚定(译者注：原文为Anchoring)*。修改一下刚刚讲过的 *SplitSentence*，使它能够确保消息都被处理了。

```
class SplitSentence implenents IRichBolt {

    private OutputCollector collector;
```

```
    public void prepare(Map conf, TopologyContext context, OutputCollector collector)
{
        this.collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for(String word : sentence.split(" ")) {
            collector.emit(tuple, new Values(word));
        }
        collector.ack(tuple);
    }

    public void cleanup(){}

    public void declareOutputFields(OutputFieldsDeclarer declarer){
        declar.declare(new Fields("word"));
    }
}
```

锚定发生在调用 **collector.emit()**时。正如前面提到的，Storm 可以沿着元组追踪到始发 *spout*。**collector.ack(tuple)**和 **collector.fail(tuple)**会告知 spout 每条消息都发生了什么。当树上的每

条消息都已被处理了，Storm 就认为来自 *spout* 的元组被全面的处理了。如果一个元组没有在设置的超时时间内完成对消息树的处理，就认为这个元组处理失败。默认超时时间为 30 秒。

NOTE:你可以通过修改 Config.TOPOLOGY_MESSAGE_TIMEOUT 修改拓扑的超时时间。

当然了 spout 需要考虑消息的失败情况，并相应的重试或丢弃消息。

NOTE:你处理的每条消息要么是确认的（译者注：collector.ack()）要么是失败的（译者注：collector.fail()）。Storm 使用内存跟踪每个元组，所以如果你不调用这两个方法，该任务最终将耗尽内存。

多数据流

一个 *bolt* 可以使用 **emit(streamId, tuple)**把元组分发到多个流，其中参数 **streamId** 是一个用来标识流的字符串。然后，你可以在 **TopologyBuilder** 决定由哪个流订阅它。

多锚定

为了用 *bolt* 连接或聚合数据流，你需要借助内存缓冲元组。为了在这一场景下确保消息完成，你不得不把流锚定到多个元组上。可以向 **emit** 方法传入一个元组列表来达成目的。

```
...

List anchors = new ArrayList();

anchors.add(tuple1);

anchors.add(tuple2);

collector.emit(anchors, values);

...
```

通过这种方式，bolt 在任意时刻调用 **ack** 或 **fail** 方法，都会通知消息树，而且由于流锚定了多个元组，所有相关的 *spout* 都会收到通知。

使用 IBasicBolt 自动确认

你可能已经注意到了，在许多情况下都需要消息确认。简单起见，Storm 提供了另一个用来实现 *bolt* 的接口，**IBasicBolt**。对于该接口的实现类的对象，会在执行 **execute** 方法之后自动调用 **ack** 方法。

```
class SplitSentence extends BaseBasicBolt {

    public void execute(Tuple tuple, BasicOutputCollector collector) {

        String sentence = tuple.getString(0);

        for(String word : sentence.split(" ")) {

            collector.emit(new Values(word));

        }

}



    public void declareOutputFields(OutputFieldsDeclarer declarer) {

        declarer.declare(new Fields("word"));

    }

}
```

NOTE:分发消息的 BasicOutputCollector 自动锚定到作为参数传入的元组。

Storm 入门之第 6 章一个实际的例子

本章要阐述一个典型的网络分析解决方案，而这类问题通常利用 Hadoop 批处理作为解决方案。

与 Hadoop 不同的是，基于 Storm 的方案会实时输出结果。

我们的这个例子有三个主要组件（见图 6-1）

- 一个基于 Node.js 的 web 应用，用于测试系统
- 一个 Redis 服务器，用于持久化数据
- 一个 Storm 拓扑，用于分布式实时处理数据

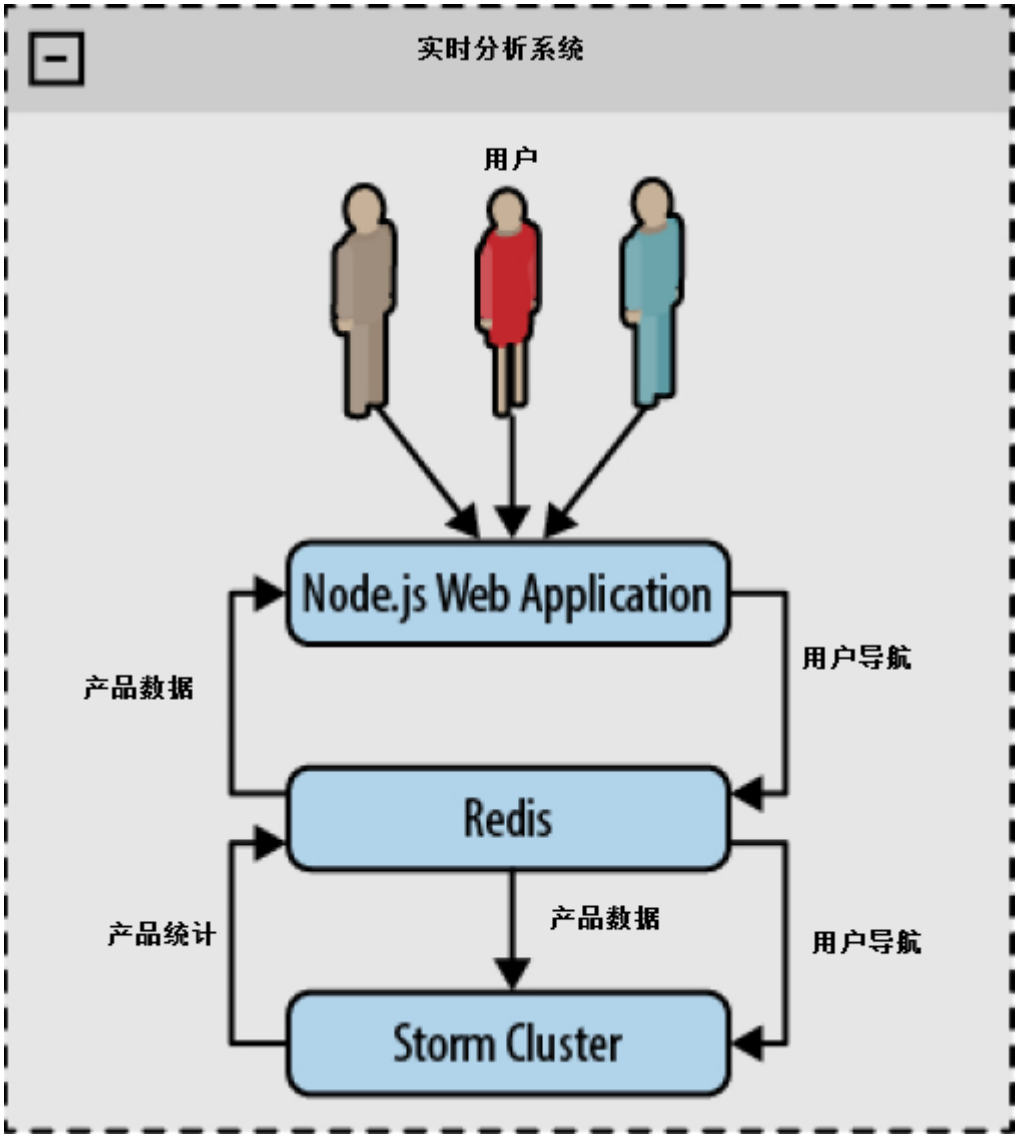


图 6-1　架构概览

NOTE:你如果想先把这个例子运行起来，请首先阅读附录 C

基于 Node.js 的 web 应用

我们已经伪造了简单的电子商务网站。这个网站只有三个页面：一个主页、一个产品页和一个产品统计页面。这个应用基于 Express 和 Socket.io 两个框架实现了向浏览器推送内容更新。制作这个应用的目的是为了让你体验 Storm 集群功能并看到处理结果，但它不是本书的重点，所以我们不会对它的页面做更详细描述。

主页

这个页面提供了全部有效产品的链接。它从 Redis 服务器获取数据并在页面上把它们显示出来。这个页面的 URL 是 http://localhost:3000/。（见图 6-2，译者注，图 6-2 翻译如下，全是文字就不制图了）

有效产品：

DVD 播放器（带环绕立体声系统）

全高清蓝光 dvd 播放器

媒体播放器（带 USB 2.0 接口）

全高清摄像机

防水高清摄像机

防震防水高清摄像机

反射式摄像机

双核安卓智能手机（带 64GB SD 卡）

普通移动电话

卫星电话

64GB SD 卡

32GB SD 卡

16GB SD 卡

粉红色智能手机壳

黑色智能手机壳

小山羊皮智能手机壳

图 6-2 首页

产品页

产品页用来显示指定产品的相关信息，例如，价格、名称、分类。这个页面的 URL 是：http://localhost:3000/product/:id。（见图 6-3，译者注：全是文字不再制图，翻译如下）

产品页：32 英寸液晶电视

分类：电视机

价格：400

相关分类

图 6-3，产品页

产品统计页

这个页面显示通过收集用户浏览站点，用 Storm 集群计算的统计信息。可以显示为如下概要：浏览这个产品的用户，在那些分类下面浏览了 n 次产品。该页的 URL 是：http://localhost:3000/product/:id/stats。（见图 6-4，译者注：全是文字，不再制图，翻译如下）

浏览了该产品的用户也浏览了以下分类的产品：

1.摄像机

2.播放器

3.手机壳

4.存储卡

图 6-4. 产品统计视图

启动这个 Node.js web 应用

首先启动 Redis 服务器，然后执行如下命令启动 web 应用：

```
node webapp/app.js
```

为了向你演示，这个应用会自动向 Redis 填充一些产品数据作为样本。

Storm 拓扑

为这个系统搭建 Storm 拓扑的目标是改进产品统计的实时性。产品统计页显示了一个分类计数器列表，用来显示访问了其它同类产品的用户数。这样可以帮助卖家了解他们的用户需求。拓扑接收浏览日志，并更新产品统计结果（见图 6-5）。

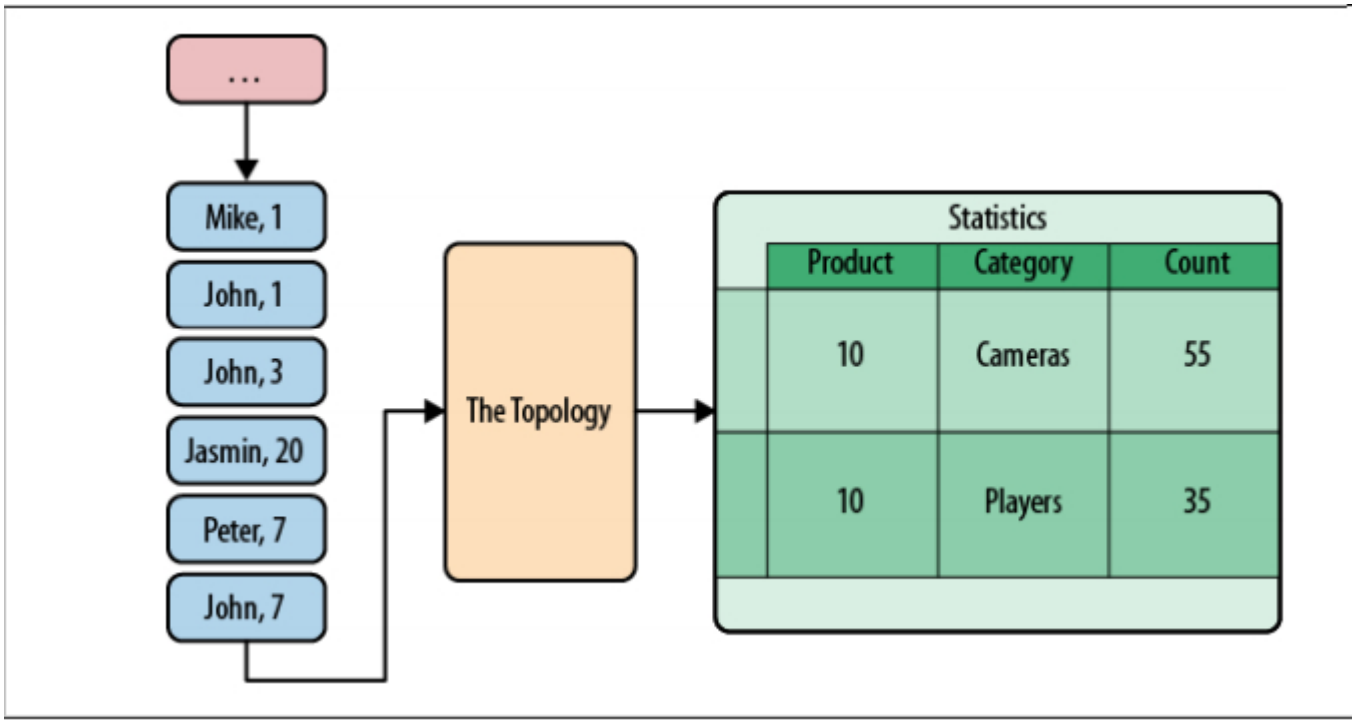


图 6-5　Storm 拓扑的输入与输出

我们的 Storm 拓扑有五个组件：一个 *spout* 向拓扑提供数据，四个 *bolt* 完成统计任务。

UsersNavigationSpout

从用户浏览数据队列读取数据发送给拓扑

GetCategoryBolt

从 Redis 服务器读取产品信息，向数据流添加产品分类

UserHistoryBolt

读取用户以前的产品浏览记录，向下一步分发 Product:Category 键值对，在下一步更新计数器

ProductCategoriesCounterBolt

追踪用户浏览特定分类下的商品次数

NewsNotifierBolt

通知 web 应用立即更新用户界面

下图展示了拓扑的工作方式（见图 6-6）

```
package storm.analytics;

...

public class TopologyStarter {

    public static void main(String[] args) {

        Logger.getRootLogger().removeAllAppenders();

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("read-feed", new UsersNavigationSpout(),3);

        builder.setBolt("get-categ", new GetCategoryBolt(),3)

                .shuffleGrouping("read-feed");

        builder.setBolt("user-history", new UserHistoryBolt(),5)

                .fieldsGrouping("get-categ", new Fields("user"));

        builder.setBolt("product-categ-counter", new
ProductCategoriesCounterBolt(),5)

                .fieldsGrouping("user-history", new Fields("product"));

        builder.setBolt("news-notifier", new NewsNotifierBolt(),5)

                .shuffleGrouping("product-categ-counter");



        Config conf = new Config();
```

```
        conf.setDebug(true);

        conf.put("redis-host",REDIS_HOST);

        conf.put("redis-port",REDIS_PORT);

        conf.put("webserver", WEBSERVER);



        LocalCluster cluster = new LocalCluster();

        cluster.submitTopology("analytics", conf, builder.createTopology());

    }

}
```

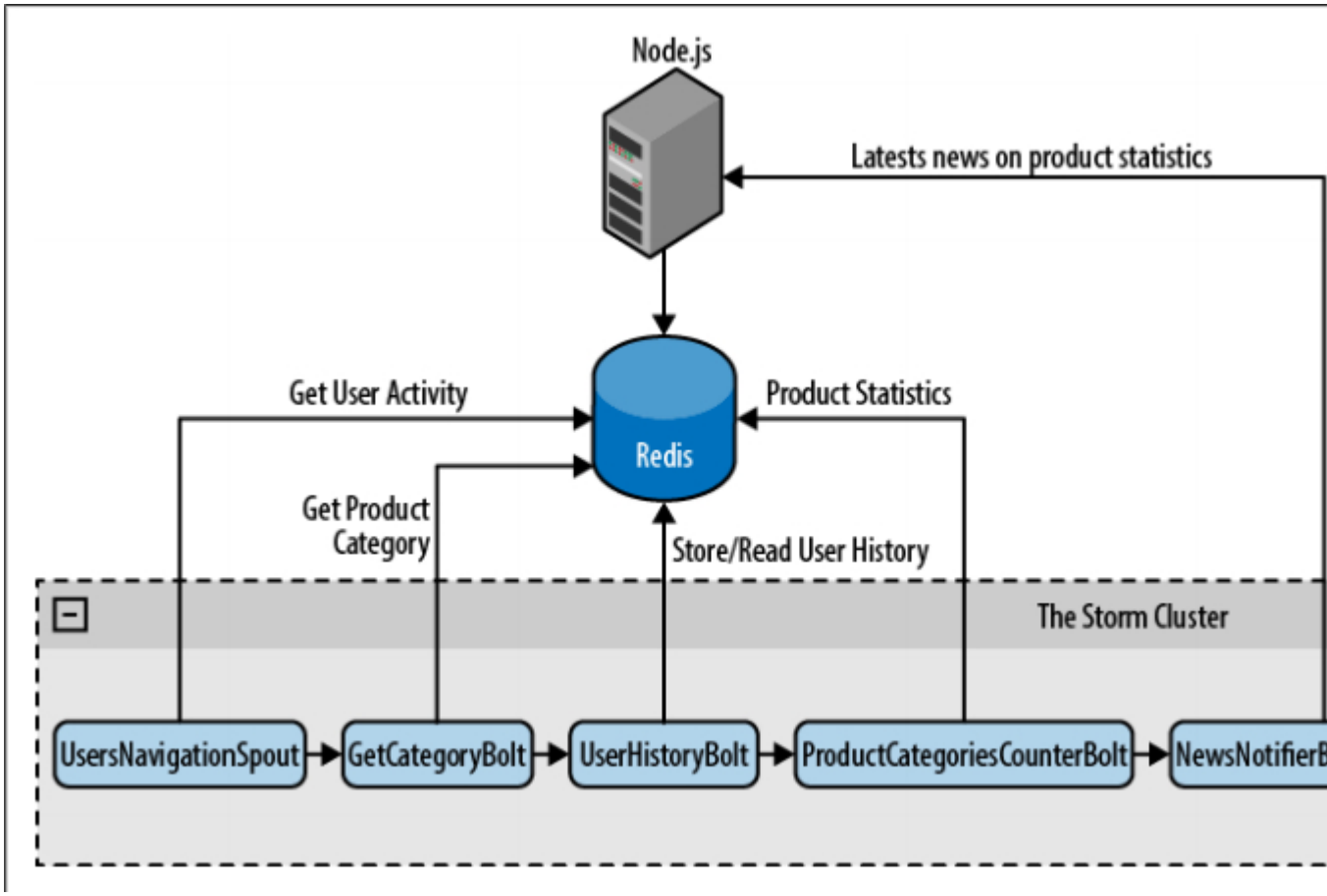


Figure 6-6 Storm 拓扑

UsersNavigationSpout

UsersNavigationSpout 负责向拓扑提供浏览数据。每条浏览数据都是一个用户浏览过的产品页的引用。它们都被 web 应用保存在 Redis 服务器。我们一会儿就要看到更多信息。

你可以使用 https://github.com/xetorthio/jedis 从 Redis 服务器读取数据，这是个极为轻巧简单的 Java Redis 客户端。

NOTE: 下面的代码块就是相关代码。

```
package storm.analytics;

public class UsersNavigationSpout extends BaseRichSpout {
```

```
Jedis jedis;



...



@Override

public void nextTuple() {

    String content = jedis.rpop("navigation");

    if(content==null || "nil".equals(content)){

        try { Thread.sleep(300); } catch (InterruptedException e) {}

    } else {

        JSONObject obj=(JSONObject)JSONValue.parse(content);

        String user = obj.get("user").toString();

        String product = obj.get("product").toString();

        String type = obj.get("type").toString();

        HashMap<String, String> map = new HashMap<String, String>();

        map.put("product", product);

        NavigationEntry entry = new NavigationEntry(user, type, map);

        collector.emit(new Values(user, entry));

    }

}
```

```
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("user", "otherdata"));
    }
}
```

spout 首先调用 **jedis.rpop("navigation")**从 Redis 删除并返回"navigation"列表最右边的元素。如果列表已经是空的，就休眠 0.3 秒，以免使用忙等待循环阻塞服务器。如果得到一条数据（数据是 JSON 格式），就解析它，并创建一个包含该数据的 **NavigationEntry** POJO：

- 浏览页面的用户
- 用户浏览的页面类型
- 由页面类型决定的额外页面信息。"产品"页的额外信息就是用户浏览的产品 ID。

spout 调用 **collector.emit(new Values(user, entry))**分发包含这些信息的元组。这个元组的内容是拓扑里下一个 *bolt* 的输入。

GetCategoryBolt

这个 *bolt* 非常简单。它只负责反序列化前面的 *spout* 分发的元组内容。如果这是产品页的数据，就通过 ProductsReader 类从 Redis 读取产品信息，然后基于输入的元组再分发一个新的包含具体产品信息的元组：

- 用户
- 产品
- 产品类别

```
package storm.analytics;



public class GetCategoryBolt extends BaseBasicBolt {
```

```
private ProductReader reader;



...

@Override

public void execute(Tuple input, BasicOutputCollector collector) {

    NavigationEntry entry = (NavigationEntry)input.getValue(1);

    if("PRODUCT".equals(entry.getPageType())){

        try {

            String product = (String)entry.getOtherData().get("product");



            //调用产品条目API，得到产品信息

            Product itm = reader.readItem(product);

            if(itm == null) {

                return;

            }

            String categ = itm.getCategory();

            collector.emit(new Values(entry.getUserId(), product, categ));

        } catch (Exception ex) {

            System.err.println("Error processing PRODUCT tuple"+ ex);

            ex.printStackTrace();

        }
```

```
        }
    }
    ...
}
```

正如前面所提到的， 使用 ProductsReader 类读取产品具体信息。

```
package storm.analytics.utilities;
...
public class ProductReader {
    ...
    public Product readItem(String id) throws Exception{
        String content = jedis.get(id);
        if(content == null || ("nil".equals(content))){
            return null;
        }
        Object obj = JSONValue.parse(content);
        JSONObjectproduct = (JSONObject)obj;
        Product i = new Product((Long)product.get("id"),
                                (String)product.get("title"),
                                (Long)product.get("price"),
                                (String)product.get("category"));
```

```
        return i;
    }
    ...
}
```

UserHistoryBolt

UserHistoryBolt 是整个应用的核心。它负责持续追踪每个用户浏览过的产品，并决定应当增加计数的键值对。

我们使用 Redis 保存用户的产品浏览历史，同时基于性能方面的考虑，还应该保留一份本地副本。我们把数据访问细节隐藏在方法 **getUserNavigationHistory(user)**和 **addProductToHistory(user,prodKey)**里，分别用来读/写访问。它们的实现如下

```
package storm.analytics;
...
public class UserHistoryBolt extends BaseRichBolt{
    @Override
    public void execute(Tuple input) {
        String user = input.getString(0);
        String prod1 = input.getString(1);
        String cat1 = input.getString(2);

        //产品键嵌入了产品类别信息
```

```
        String prodKey = prod1+":"+cat1;



        Set productsNavigated = getUserNavigationHistory(user);



        //如果用户以前浏览过->忽略它
        if(!productsNavigated.contains(prodKey)) {
            //否则更新相关条目
            for (String other : productsNavigated) {
                String[] ot = other.split(":");
                String prod2 = ot[0];
                String cat2 = ot[1];
                collector.emit(new Values(prod1, cat2));
                collector.emit(new Values(prod2, cat1));
            }
            addProductToHistory(user, prodKey);
        }
    }
}
```

需要注意的是，这个 *bolt* 的输出是那些类别计数应当获得增长的产品。

看一看代码。这个 *bolt* 维护着一组被每个用户浏览过的产品。值得注意的是，这个集合包含产品：类别键值对，而不是只有产品。这是因为你会在接下来的调用中用到类别信息，而且这样也比每次从数据库获取更高效。这样做的原因是基于以下考虑，产品可能只有一个类别，而且它在整个产品的生命周期当中不会改变。

读取了用户以前浏览过的产品集合之后（以及它们的类别），检查当前产品以前有没有被浏览过。如果浏览过，这条浏览数据就被忽略了。如果这是首次浏览，遍历用户浏览历史，并执行 **collector.emit(new Values(prod1,cat2))**分发一个元组，这个元组包含当前产品和所有浏览历史类别。第二个元组包含所有浏览历史产品和当前产品类别，由 **collectior.emit(new Values(prod2,cat1))**。最后，将当前产品和它的类别添加到集合。

比如，假设用户 John 有以下浏览历史：

User	#	Category
John	0	Players
John	2	Players
John	17	TVs
John	21	Mounts

下面是将要处理的浏览数据

User	#	Category
John	8	Phones

该用户没有浏览过产品 8，因此你需要处理它。

因此要分发以下元组：

#	Category
8	Players
8	Players
8	TVs
8	Mounts
0	Phones
2	Phones
17	Phones
21	Phones

注意，左边的产品和右边的类别之间的关系应当作为一个整体递增。

现在，让我们看看这个 *Bolt* 用到的持久化实现。

```
public class UserHistoryBolt extends BaseRichBolt{

    ...

    private Set getUserNavigationHistory(String user) {

        Set userHistory = usersNavigatedItems.get(user);

        if(userHistory == null) {

            userHistory = jedis.smembers(buildKey(user));

            if(userHistory == null)

                userHistory = new HashSet();

            usersNavigatedItems.put(user, userHistory);

        }

        return userHistory;

    }
```

```
    private void addProductToHistory(String user, String product) {
        Set userHistory = getUserNavigationHistory(user);
        userHistory.add(product);
        jedis.sadd(buildKey(user), product);
    }
    ...
}
```

getUserNavigationHistory 方法返回用户浏览过的产品集。首先，通过 **usersNavigatedItems.get(*user*)**方法试图从本地内存得到用户浏览历史，否则，使用 **jedis.smembers(buildKey(user))**从 Redis 服务器获取，并把数据添加到本地数据结构 **usersNavigatedItems**。

当用户浏览一个新产品时，调用 **addProductToHistory**，通过 **userHistory.add(product)**和 **jedis.sadd(buildKey(user),product)**同时更新内存数据结构和 Redis 服务器。

需要注意的是，当你需要做并行化处理时，只要 *bolt* 在内存中维护着用户数据，你就得首先通过用户做域数据流分组（译者注：原文是 fieldsGrouping，详细情况请见第三章的域数据流组），这是一件很重要的事情，否则集群内将会有用户浏览历史的多个不同步的副本。

ProductCategoriesCounterBolt

该类持续追踪所有的产品-类别关系。它通过由 UsersHistoryBolt 分发的产品-类别数据对更新计数。

每个数据对的出现次数保存在 Redis 服务器。基于性能方面的考虑，要使用一个本地读写缓存，通过一个后台线程向 Redis 发送数据。

该 *Bolt* 会向拓扑的下一个 *Bolt*——NewsNotifierBolt——发送包含最新记数的元组，这也是最后一个 *Bolt*，它会向最终用户广播实时更新的数据。

```
public class ProductCategoriesCounterBolt extends BaseRichBolt {

    ...

    @Override

    public void execute(){

        String product = input.getString(0);

        String categ = input.getString(1);

        int total = count(product, categ);

        collector.emit(new Values(product, categ, total));

    }

    ...

    private int count(String product, String categ) {

        int count = getProductCategoryCount(categ, product);

        count++;

        storeProductCategoryCount(categ, product, count);

        return count;

    }

    ...

}
```

这个 *bolt* 的持久化工作隐藏在 **getProductCategoryCount** 和 **storeProductCategoryCount** 两个方法中。它们的具体实现如下：

```
package storm.analytics;

...

public class ProductCategoriesCounterBolt extends BaseRichBolt {

    // 条目：分类 -> 计数

    HashMap<String,Integer> counter = new HashMap<String, Integer>();



    //条目：分类 -> 计数

    HashMap<String,Integer> pendingToSave = new HashMap<String,Integer>();



    ...

    public int getProductCategoryCount(String categ, String product) {

        Integer count = counter.get(buildLocalKey(categ, product));

        if(count == null) {

            String sCount = jedis.hget(buildRedisKey(product), categ);

            if(sCount == null || "nil".equals(sCount)) {

                count = 0;

            } else {

                count = Integer.valueOf(sCount);

            }
```

```
        }

        return count;

    }

    ...

    private void storeProductCategoryCount(String categ, String product, int count) {

        String key = buildLocalKey(categ, product);

        counter.put(key, count);

        synchronized (pendingToSave) {

            pendingToSave.put(key, count);

        }

    }

    ...

}
```

方法 **getProductCategoryCount** 首先检查内存缓存计数器。如果没有有效令牌，就从 Redis 服务器取得数据。

方法 **storeProductCategoryCount** 更新计数器缓存和 pendingToSae 缓冲。缓冲数据由下述后台线程持久化。

```
package storm.analytics;



public class ProductCategoriesCounterBolt extends BaseRichBolt {
```

```
...

    private void startDownloaderThread() {

        TimerTask t = startDownloaderThread() {

            @Override

            public void run () {

                HashMap<String, Integer> pendings;

                synchronized (pendingToSave) {

                    pendings = pendingToSave;

                    pendingToSave = new HashMap<String,Integer>();

                }



                for (String key : pendings.keySet) {

                    String[] keys = key.split(":");

                    String product = keys[0];

                    String categ = keys[1];

                    Integer count = pendings.get(key);

                    jedis.hset(buildRedisKey(product), categ, count.toString());

                }

            }

        };

        timer = new Timer("Item categories downloader");
```

```
        timer.scheduleAtFixedRate(t, downloadTime, downloadTime);
    }
    ...
}
```

下载线程锁定 pendingToSave， 向 Redis 发送数据时会为其它线程创建一个新的缓冲。这段代码每隔 downloadTime 毫秒运行一次，这个值可由拓扑配置参数 download-time 配置。download-time 值越大，写入 Redis 的次数就越少，因为一对数据的连续计数只会向 Redis 写一次。

NewsNotifierBolt

为了让用户能够实时查看统计结果，由 NewsNotifierBolt 负责向 web 应用通知统计结果的变化。通知机制由 Apache HttpClient 通过 HTTP POST 访问由拓扑配置参数指定的 URL。POST 消息体是 JSON 格式。

测试时把这个 *bolt* 从拜年中删除。

```
01 package storm.analytics;
02 ...
03 public class NewsNotifierBolt extends BaseRichBolt {
04 ...
05 @Override
06 public void execute(Tuple input) {
07 String product = input.getString(0);
08 String categ = input.getString(1);
09 int visits = input.getInteger(2);</code>
10
11 String content ="{\"product\":\"+product+"\",\"categ\":\""+categ+"\",\"visits\":"+visits+"}";
12 HttpPost post =new HttpPost(webserver);
```

```
13 try {
14 post.setEntity(new StringEntity(content));
15 HttpResponse response = client.execute(post);
16 org.apache.http.util.EntityUtils.consume(response.getEntity());
17 } catch (Exception e) {
18 e.printStackTrace();
19 reconnect();
20 }
21 }
22 ...
23 }
```

Redis 服务器

Redis 是一种先进的、基于内存的、支持持久化的键值存储（见 http://redis.io）。本例使用它存储以下信息：

- 产品信息，用来为 web 站点服务
- 用户浏览队列，用来为 Storm 拓扑提供数据
- Storm 拓扑的中间数据，用于拓扑发生故障时恢复数据
- Storm 拓扑的处理结果，也就是我们期望得到的结果。

产品信息

Redis 服务器以产品 ID 作为键，以 JSON 字符串作为值保存着产品信息。

```
1 redis-cli
2 redis 127.0.0.1:6379&gt; get 15
3 "{\"title\":\"Kids smartphone cover\",\"category\":\"Covers\",\"price\":30,\"id\":
4 15}"
```

用户浏览队列

用户浏览队列保存在 Redis 中一个键为 navigation 的先进先出队列中。用户浏览一个产品页时，服务器从队列左侧添加用户浏览数据。Storm 集群不断的从队列右侧获取并移除数据。

```
01 redis 127.0.0.1:6379&gt; llen navigation
02 (integer) 5
```

```
03 redis 127.0.0.1:6379&gt; lrange navigation 0 4
04 1)
"{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"1\",\"type\":
05 \"PRODUCT\"}"
06 2)
"{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"1\",\"type\":
07 \"PRODUCT\"}"
08 3)
"{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"2\",\"type\":
09 \"PRODUCT\"}"
10 4)
"{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"3\",\"type\":
11 \"PRODUCT\"}"
12 5)
"{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"5\",\"type\":
13 \"PRODUCT\"}"
```

中间数据

集群需要分开保存每个用户的历史数据。为了实现这一点，它在 Redis 服务器上保存着一个包含所有用户浏览过的产品和它们的分类的集合。

```
1 redis 127.0.0.1:6379&gt; smembers history:59c34159-0ecb-4ef3-a56b-99150346f8d5
2 1)"1:Players"
3 2)"5:Cameras"
4 3)"2:Players"
5 4)"3:Cameras"
```

结果

Storm 集群生成关于用户浏览的有用数据，并把它们的产品 ID 保存在一个名为“prodcnt”的 Redis hash 中。

```
1 redis 127.0.0.1:6379&gt; hgetall prodcnt:2
2 1)"Players"
3 2)"1"
4 3)"Cameras"
5 4)"2"
```

测试拓扑

使用 LocalCluster 和一个本地 Redis 服务器执行测试（见图 6-7）。向 Redis 填充产品数据，伪造访问日志。我们的断言会在读取拓扑向 Redis 输出的数据时执行。测试用户用 java 和 groovy 完成。

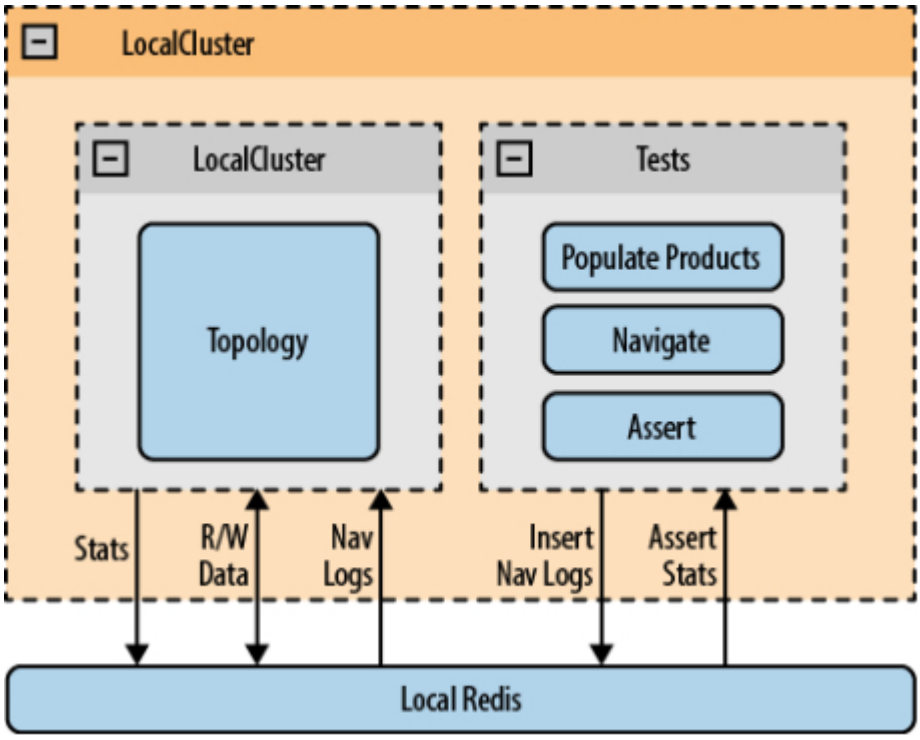


图 6-7. 测试架构

初始化测试

初始化由以下三步组成：

启动 LocalCluster 并提交拓扑。初始化在 AbstractAnalyticsTest 实现，所有测试用例都继承该类。当初始化多个 AbstractAnalyticsTest 子类对象时，由一个名为 topologyStarted 的静态标志属性确定初始化工作只会进行一次。

需要注意的是，sleep 语句是为了确保在试图获取结果之前 LocalCluster 已经正确启动了。

```
01 public abstract class AbstractAnalyticsTest extends Assert {
02   def jedis
03   static topologyStarted = false
```

```
04  static sync= new Object()
05  private void reconnect() {
06  jedis = new Jedis(TopologyStarter.REDIS_HOST, TopologyStarter.REDIS_PORT)
07  }
08  @Before
09  public void startTopology(){
10  synchronized(sync){
11  reconnect()
12  if(!topologyStarted){
13  jedis.flushAll()
14  populateProducts()
15  TopologyStarter.testing = true
16  TopologyStarter.main(null)
17  topologyStarted = true
18  sleep 1000
19  }
20  }
21  }
22  ...
23  public void populateProducts() {
24  def testProducts = [
25  [id: 0, title:"Dvd player with surround sound system",
26  category:"Players", price: 100],
27  [id: 1, title:"Full HD Bluray and DVD player",
28  category:"Players", price:130],
29  [id: 2, title:"Media player with USB 2.0 input",
30  category:"Players", price:70],
31  ...
32  [id: 21, title:"TV Wall mount bracket 50-55 Inches",
33  category:"Mounts", price:80]
34  ]
35  testProducts.each() { product -&gt;
36  def val =
37  "{ \"title\": \"${product.title}\" , \"category\": \"${product.category}\"," +
38  " \"price\": ${product.price}, \"id\": ${product.id} }"
39  println val
```

```
40    jedis.set(product.id.toString(), val.toString())
41    }
42    }
43    ...
44 }
```

在 AbstractAnalyticsTest 中实现一个名为 navigate 的方法。为了测试不同的场景，我们要模拟用户浏览站点的行为，这一步向 Redis 的浏览队列(译者注：就是前文提到的键是 navigation 的队列)插入浏览数据。

```
01 public abstract class AbstractAnalyticsTest extends Assert {
02    ...
03 public void navigate(user, product) {
04    String nav =
05    "{\"user\":    \"${user}\",    \"product\":    \"${product}\",    \"type\":
   \"PRODUCT\"}".toString()
06    println "Pushing navigation: ${nav}"
07    jedis.lpush('navigation', nav)
08    }
09    ...
10 }
```

实现一个名为 getProductCategoryStats 的方法，用来读取指定产品与分类的数据。不同的测试同样需要断言统计结果，以便检查拓扑是否按照期望的那样执行了。

```
01 public abstract class AbstractAnalyticsTest extends Assert {
02    ...
03    public int getProductCategoryStats(String product, String categ) {
04    String count = jedis.hget("prodcnt:${product}", categ)
05    if(count == null || "nil".equals(count))
06    return 0
07    return Integer.valueOf(count)
08    }
09    ...
10 }
```

一个测试用例

下一步，为用户“1”模拟一些浏览记录，并检查结果。注意执行断言之前要给系统留出两秒钟处理数据。（记住 **ProductCategoryCounterBolt** 维护着一份计数的本地副本，它是在后台异步保存到 Redis 的。）

```
01 package functional
02 class StatsTest extends AbstractAnalyticsTest {
03   @Test
04   public void testNoDuplication(){
05           navigate("1", "0") // Players
06           navigate("1", "1") // Players
07           navigate("1", "2") // Players
08           navigate("1", "3") // Cameras
09           Thread.sleep(2000) // Give two seconds for the system to process the data.
10           assertEquals 1, getProductCategoryStats("0", "Cameras")
11           assertEquals 1, getProductCategoryStats("1", "Cameras")
12           assertEquals 1, getProductCategoryStats("2", "Cameras")
13           assertEquals 2, getProductCategoryStats("0", "Players")
14           assertEquals 3, getProductCategoryStats("3", "Players")
15   }
16 }
```

对可扩展性和可用性的提示

为了能在一章的篇幅中讲明白整个方案，它已经被简化了。正因如此，一些与可扩展性和可用性有关的必要复杂性也被去掉了。这方面主要有两个问题。

Redis 服务器不只是一个故障的节点，还是性能瓶颈。你能接收的数据最多就是 Redis 能处理的那些。Redis 可以通过分片增强扩展性，它的可用性可以通过主从配置得到改进。这都需要修改拓扑和 web 应用的代码实现。

另一个缺点就是 web 应用不能通过增加服务器成比例的扩展。这是因为当产品统计数据发生变化时，需要通知所有关注它的浏览器。这一“通知浏览器”的机制通过 Socket.io 实现，但是它要求监听器和通知器在同一主机上。这一点只有当 **GET /product/:id/stats** 和 **POST /news** 满足

以下条件时才能实现，那就是这二者拥有相同的分片标准，确保引用相同产品的请求由相同的服务器处理。

Storm 入门之第 7 章使用非 JVM 语言开发

有时候你可能想使用不是基于 JVM 的语言开发一个 Storm 工程，你可能更喜欢使用别的语言或者想使用用某种语言编写的库。

Storm 是用 Java 实现的，你看到的所有这本书中的 *spout* 和 *bolt* 都是用 java 编写的。那么有可能使用像 Python、Ruby、或者 JavaScript 这样的语言编写 *spout* 和 *bolt* 吗？答案是当然

可以！可以使用*多语言协议*达到这一目的。

多语言协议是 Storm 实现的一种特殊的协议，它使用标准输入输出作为 *spout* 和 *bolt* 进程间的通讯通道。消息以 JSON 格式或纯文本格式在通道中传递。

我们看一个用非 JVM 语言开发 *spout* 和 *bolt* 的简单例子。在这个例子中有一个 *spout* 产生从 1 到 10,000 的数字，一个 *bolt* 过滤素数，二者都用 PHP 实现。

NOTE: 在这个例子中，我们使用一个很笨的办法验证素数。有更好当然也更复杂的方法，它们已经超出了这个例子的范围。

有一个专门为 Storm 实现的 PHP DSL(译者注：领域特定语言)，我们将会在例子中展示我们的实现。首先定义拓扑。

```
1...
2 TopologyBuilder builder =  new  TopologyBuilder();
3 builder.setSpout("numbers-generator",  new  NumberGeneratorSpout(1, 10000));
4 builder.setBolt("prime-numbers-filter",  new
5 PrimeNumbersFilterBolt()).shuffleGrouping("numbers-generator");
6 StormTopology topology = builder.createTopology();
```

7 . . .

NOTE：有一种使用非 JVM 语言定义拓扑的方式。既然 Storm 拓扑是 Thrift 架构，而且 *Nimbus* 是一个 Thrift 守护进程，你就可以使用任何你想用的语言创建并提交拓扑。但是这已经超出了本书的范畴了。

这里没什么新鲜了。我们看一下 **NumbersGeneratorSpout** 的实现。

```
01 public class NumberGeneratorSpout extends ShellSpout implements IRichSpout
   {
02         public NumberGeneratorSpout(Integer from, Integer to) {
                super("php", "-f", "NumberGeneratorSpout.php", from.toString(),
03 to.toString());
04         }
05         public void declareOutputFields(OutputFieldsDeclarer declarer) {
06                 declarer.declare(new Fields("number"));
07         }
08         public Map<String, Object> getComponentConfiguration() {
09                 return null;
10         }
11 }
```

你可能已经注意到了，这个 *spout* 继承了 **ShellSpout**。这是个由 Storm 提供的特殊的类，用来帮助你运行并控制用其它语言编写的 *spout*。在这种情况下它告诉 Storm 如何执行你的 PHP 脚本。

NumberGeneratorSpout 的 PHP 脚本向标准输出分发元组，并从标准输入读取确认或失败信号。

在开始实现 NumberGeneratorSpout.php 脚本之前，多观察一下多语言协议是如何工作的。

spout 按照传递给构造器的参数从 **from** 到 **to** 顺序生成数字。

接下来看看 **PrimeNumbersFilterBolt**。这个类实现了之前提到的壳。它告诉 Storm 如何执行你的 PHP 脚本。Storm 为这一目的提供了一个特殊的叫做 **ShellBolt** 的类，你惟一要做的事就是指出如何运行脚本以及声明要分发的属性。

```
1 public class PrimeNumbersFilterBolt extends ShellBolt implements IRichBolt
  {
2         public PrimeNumbersFilterBolt() {
3                 super("php", "-f", "PrimeNumbersFilterBolt.php");
4         }
5         public void declareOutputFields(OutputFieldsDeclarer declarer) {
6                 declarer.declare(new Fields("number"));
7         }
8 }
```

在这个构造器中只是告诉 Storm 如何运行 PHP 脚本。它与下列命令等价。

```
1 php -f PrimeNumbersFilterBolt.php
```

PrimeNumbersFilterBolt.php 脚本从标准输入读取元组，处理它们，然后向标准输出分发、确认或失败。在开始这个脚本之前，我们先多了解一些多语言协议的工作方式。

1. 发起一次握手
2. 开始循环
3. 读/写元组

NOTE：有一种特殊的方式可以使用 Storm 的内建日志机制在你的脚本中记录日志，所以你不需要自己实现日志系统。

下面我们来看一看上述每一步的细节，以及如何用 PHP 实现它。

发起握手

为了控制整个流程（开始以及结束它），Storm 需要知道它执行的脚本进程号（PID）。根据多语言协议，你的进程开始时发生的第一件事就是 Storm 要向标准输入（译者注：根据上下文理解,本章提到的标准输入输出都是从非 JVM 语言的角度理解的,这里提到的标准输入也就是 PHP 的标准输入）发送一段 JSON 数据，它包含 Storm 配置、拓扑上下文和一个进程号目录。它看起来就像下面的样子：

```
{
```

```
    "conf": {
        "topology.message.timeout.secs": 3,
        // etc
    },
    "context": {
        "task->component": {
            "1": "example-spout",
            "2": "__acker",
            "3": "example-bolt"
        },
        "taskid": 3
    },
    "pidDir": "..."
}
```

脚本进程必须在 **pidDir** 指定的目录下以自己的进程号为名字创建一个文件，并以 JSON 格式把进程号写到标准输出。

```
{"pid": 1234}
```

举个例子，如果你收到**/tmp/example\n** 而你的脚本进程号是 123，你应该创建一个名为 **/tmp/example/123** 的空文件并向标准输出打印文本行 **{“pid”: 123}\n**（译者注：此处原文只有

一个 n，译者猜测应是排版错误）和 **end\n**。这样 Storm 就能持续追踪进程号并在它关闭时杀死脚本进程。下面是 PHP 实现：

```
1 $config = json_decode(read_msg(), true);
2 $heartbeatdir = $config['pidDir'];
3 $pid = getmypid();
4 fclose(fopen("$heartbeatdir/$pid", "w"));
5 storm_send(["pid"=>$pid]);
6 flush();
```

你已经实现了一个叫做 **read_msg** 的函数，用来处理从标准输入读取的消息。按照多语言协议的声明，消息可以是单行或多行 JSON 文本。一条消息以 **end\n** 结束。

```
01 function read_msg() {
02         $msg = "";
03         while(true) {
04                 $l = fgets(STDIN);
05                 $line = substr($l,0,-1);
06                 if($line=="end") {
07                         break;
08                 }
09                 $msg = "$msg$line\n";
10         }
11         return substr($msg, 0, -1);
12 }
13 function storm_send($json) {
14         write_line(json_encode($json));
15         write_line("end");
16 }
17 function write_line($line) {
18         echo("$line\n");
19 }
```

NOTE：flush()方法非常重要；有可能字符缓冲只有在积累到一定程度时才会清空。这意味着你的脚本可能会为了等待一个来自 Storm 的输入而永远挂起，而 Storm 却在等待来自你的脚本的输出。因此当你的脚本有内容输出时立即清空缓冲是很重要的。

开始循环以及读/写元组

这是整个工作中最重要的一步。这一步的实现取决于你开发的 *spout* 和 *bolt*。

如果是 *spout*，你应当开始分发元组。如果是 *bolt*，就循环读取元组，处理它们，分发它发，确认成功或失败。

下面我们就看看用来分发数字的 *spout*。

```
01 $from  =  intval($argv[1]);
02 $to  =  intval($argv[2]);
03 while(true) {
04         $msg  = read_msg();
05         $cmd  = json_decode($msg, true);
06         if  ($cmd['command']=='next') {
07                 if  ($from<$to) {
08                         storm_emit(array("$from"));
09                         $task_ids  = read_msg();
10                         $from++;
11                 }  else  {
12                         sleep(1);
13                 }
14         }
15         storm_sync();
16 }
```

从命令行获取参数 **from** 和 **to**，并开始迭代。每次从 Storm 得到一条 **next** 消息，这意味着你已准备好分发下一个元组。

一旦你发送了所有的数字，而且没有更多元组可发了，就休眠一段时间。

为了确保脚本已准备好发送下一个元组，Storm 会在发送下一条之前等待 **sync\n** 文本行。调用 **read_msg()**，读取一条命令，解析 JSON。

对于 *bolts* 来说，有少许不同。

```
01 while(true) {
02         $msg  = read_msg();
03         $tuple  = json_decode($msg, true, 512, JSON_BIGINT_AS_STRING);
04         if  (!empty($tuple["id"])) {
05                 if  (isPrime($tuple["tuple"][0])) {
06                         storm_emit(array($tuple["tuple"][0]));
07                 }
08                 storm_ack($tuple["id"]);
09         }
10 }
```

循环的从标准输入读取元组。解析读取每一条 JSON 消息，判断它是不是一个元组，如果是，再检查它是不是一个素数，如果是素数再次分发一个元组，否则就忽略掉，最后不论如何都要确认成功。

NOTE：在 **json_decode** 函数中使用的 **JSON_BIGINT_AS_STRING** 是为了解决一个在 JAVA 和 PHP 之间的数据转换问题。JAVA 发送的一些很大的数字，在 PHP 中会丢失精度，这样就会导致问题。为了避开这个问题，告诉 PHP 把大数字当作字符串处理，并在 JSON 消息中输出数字时不使用双引号。PHP5.4.0 或更高版本要求使用这个参数。

emit，ack，fail，以及 **log** 消息都是如下结构：

emit

```
{
    "command": "emit",
    "tuple": ["foo", "bar"]
}
```

其中的数组包含了你分发的元组数据。

ack

```
{
    "command": "ack",
    "id": 123456789
}
```

其中的 id 就是你处理的元组的 ID。

fail

```
{
    "command": "fail",
    "id": 123456789
}
```

与 **ack**（译者注：原文是 **emit** 从上下 JSON 的内容和每个方法的功能上判断此处就是 **ack**，可能是排版错误）相同，其中 id 就是你处理的元组 ID。

log

```
{
    "command": "log",
    "msg": "some message to be logged by storm."
}
```

下面是完整的的 PHP 代码。

```
001 //你的 spout：
002 <?php
```

```
003 function  read_msg() {
004         $msg  =  "";
005         while(true) {
006                 $l  =  fgets(STDIN);
007                 $line  =  substr($l,0,-1);
008                 if  ($line=="end") {
009                         break;
010                 }
011                 $msg  =  "$msg$line\n";
012         }
013         return  substr($msg, 0, -1);
014 }
015 function  write_line($line) {
016         echo("$line\n");
017 }
018 function  storm_emit($tuple) {
019         $msg  =  array("command"  =>  "emit",  "tuple"  =>  $tuple);
020         storm_send($msg);
021 }
022 function  storm_send($json) {
023         write_line(json_encode($json));
024         write_line("end");
025 }
026 function  storm_sync() {
027         storm_send(array("command"  =>  "sync"));
028 }
029 function  storm_log($msg) {
030         $msg  =  array("command"  =>  "log",  "msg"  =>  $msg);
031         storm_send($msg);
032         flush();
033 }
034 $config  = json_decode(read_msg(), true);
035 $heartbeatdir  =  $config['pidDir'];
036 $pid  =  getmypid();
037 fclose(fopen("$heartbeatdir/$pid",  "w"));
038 storm_send(["pid"=>$pid]);
```

```
039 flush();
040 $from  =  intval($argv[1]);
041 $to  =  intval($argv[2]);
042 while(true) {
043         $msg  = read_msg();
044         $cmd  = json_decode($msg, true);
045         if  ($cmd['command']=='next') {
046                 if  ($from<$to) {
047                         storm_emit(array("$from"));
048                         $task_ids  = read_msg();
049                         $from++;
050                 }  else  {
051                         sleep(1);
052                 }
053         }
054         storm_sync();
055 }
056 ?>
057 //你的 bolt:
058 <?php
059 function  isPrime($number) {
060         if  ($number  < 2) {
061                 return  false;
062         }
063         if  ($number==2) {
064                 return  true;
065         }
066         for  ($i=2;  $i<=$number-1;  $i++) {
067                 if  ($number  %  $i  == 0) {
068                         return  false;
069                 }
070         }
071         return  true;
072 }
073 function  read_msg() {
074         $msg  =  "";
```

```
075         while(true) {
076                 $l  =  fgets(STDIN);
077                 $line  =  substr($l,0,-1);
078                 if  ($line=="end") {
079                         break;
080                 }
081                 $msg  =  "$msg$line\n";
082         }
083         return  substr($msg, 0, -1);
084 }
085 function  write_line($line) {
086         echo("$line\n");
087 }
088 function  storm_emit($tuple) {
089         $msg  =  array("command"  =>  "emit",  "tuple"  =>  $tuple);
090         storm_send($msg);
091 }
092 function  storm_send($json) {
093         write_line(json_encode($json));
094         write_line("end");
095 }
096 function  storm_ack($id) {
097         storm_send(["command"=>"ack",  "id"=>"$id"]);
098 }
099 function  storm_log($msg) {
100         $msg  =  array("command"  =>  "log",  "msg"  =>  "$msg");
101         storm_send($msg);
102 }
103 $config  = json_decode(read_msg(), true);
104 $heartbeatdir  =  $config['pidDir'];
105 $pid  =  getmypid();
106 fclose(fopen("$heartbeatdir/$pid",  "w"));
107 storm_send(["pid"=>$pid]);
108 flush();
109 while(true) {
110         $msg  = read_msg();
```

```
111         $tuple  = json_decode($msg, true, 512, JSON_BIGINT_AS_STRING);
112         if  (!empty($tuple["id"])) {
113                 if  (isPrime($tuple["tuple"][0])) {
114                         storm_emit(array($tuple["tuple"][0]));
115                 }
116                 storm_ack($tuple["id"]);
117         }
118 }
119 ?>
```

NOTE：需要重点指出的是，应当把所有的脚本文件保存在你的工程目录下的一个名为 **multilang/resources** 的子目录中。这个子目录被包含在发送给工人进程的 jar 文件中。如果你不把脚本包含在这个目录中，Storm 就不能运行它们，并抛出一个错误。

Storm 入门之第 8 章事务性拓扑

正如书中之前所提到的，使用 Storm 编程，可以通过调用 ack 和 fail 方法来确保一条消息的处理成功或失败。不过当元组被重发时，会发生什么呢？你又该如何砍不会重复计算？

Storm0.7.0 实现了一个新特性——事务性拓扑，这一特性使消息在语义上确保你可以安全的方式重发消息，并保证它们只会被处理一次。在不支持事务性拓扑的情况下，你无法在准确性，可扩展性，以空错性上得到保证的前提下完成计算。

NOTE:事务性拓扑是一个构建于标准 Storm *spout* 和 *bolt* 之上的抽象概念。

设计

在事务性拓扑中，Storm 以并行和顺序处理混合的方式处理元组。*spout* 并行分批创建供 *bolt* 处理的元组（译者注：下文将这种分批创建、分批处理的元组称做批次）。其中一些 *bolt* 作为提

交者以严格有序的方式提交处理过的批次。这意味着如果你有每批五个元组的两个批次，将有两个元组被 *bolt* 并行处理，但是直到提交者成功提交了第一个元组之后，才会提交第二个元组。 **NOTE：** 使用事务性拓扑时，数据源要能够重发批次，有时候甚至要重复多次。因此确认你的数据源——你连接到的那个 *spout*——具备这个能力。这个过程可以被描述为两个阶段：*处理阶段* 纯并行阶段，许多批次同时处理。 *提交阶段* 严格有序阶段，直到批次一成功提交之后，才会提交批次二。 这两个阶段合起来称为一个 Storm 事务。 **NOTE：** Storm 使用 zookeeper 储存事务元数据，默认情况下就是拓扑使用的那个 zookeeper。你可以修改以下两个配置参数键指定其它的 zookeeper——transactional.zookeeper.servers 和 transactional.zookeeper.port。

事务实践

下面我们要创建一个 Twitter 分析工具来了解事务的工作方式。我们从一个 Redis 数据库读取 tweets，通过几个 *bolt* 处理它们，最后把结果保存在另一个 Redis 数据库的列表中。处理结果就是所有话题和它们的在 tweets 中出现的次数列表，所有用户和他们在 tweets 中出现的次数列表，还有一个包含发起话题和频率的用户列表。 这个工具的拓扑见图 8-1。

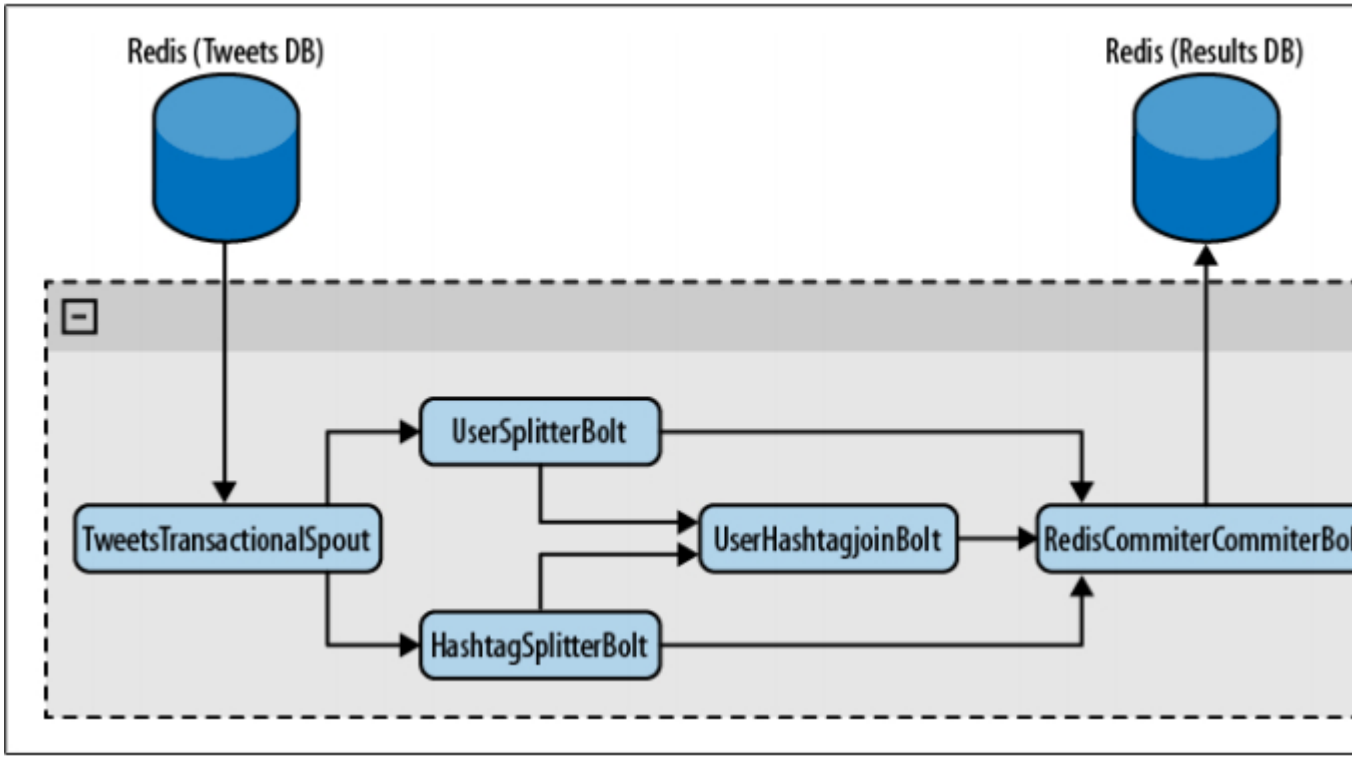


图 8-1 拓扑概览

正如你看到的，**TweetsTransactionalSpout** 会连接你的 tweet 数据库并向拓扑分发批次。**UserSplitterBolt** 和 **HashTagSplitterBolt** 两个 *bolt*，从 *spout* 接收元组。**UserSplitterBolt** 解析 tweets 并查找用户——以@开头的单词——然后把这些单词分发到名为 *users* 的自定义数据流组。**HashtagSplitterBolt** 从 tweet 查找#开头的单词，并把它们分发到名为 *hashtags* 的自定义数据流组。第三个 *bolt*，**UserHashtagJoinBolt**，接收前面提到的两个数据流组，并计算具名用户的一条 tweet 内的话题数量。为了计数并分发计算结果，这是个 **BaseBatchBolt**（稍后有更多介绍）。

最后一个 bolt——**RedisCommitterBolt**——接收以上三个 *bolt* 的数据流组。它为每样东西计数，并在对一个批次完成处理时，把所有结果保存到 redis。这是一种特殊的 *bolt*，叫做提交者，在本章后面做更多讲解。

用 **TransactionalTopologyBuilder** 构建拓扑，代码如下：

```
01 TransactionalTopologyBuilder builder=
0          new  TransactionalTopologyBuilder("test",  "spout",  newTweetsTransactio
2 nalSpout());
03
0 builder.setBolt("users-splitter",  new  UserSplitterBolt(),4).shuffleGrouping("s
4 pout");
0 buildeer.setBolt("hashtag-splitter",  new  HashtagSplitterBolt(),4).shuffleGroup
5 ing("spout");
06
07 builder.setBolt("users-hashtag-manager",  new  UserHashtagJoinBolt(), r)
0               .fieldsGrouping("users-splitter",  "users",  new  Fields("tweet_id
8 "))
0               .fieldsGrouping("hashtag-splitter",  "hashtags",  new  Fields("twe
9 et_id"));
10
11 builder.setBolt("redis-commiter",  new  RedisCommiterBolt())
12              .globalGrouping("users-splitter",  "users")
13              .globalGrouping("hashtag-splitter",  "hashtags")
14              .globalGrouping("user-hashtag-merger");
```

接下来就看看如何在一个事务性拓扑中实现 *spout*。

Spout

一个事务性拓扑的 *spout* 与标准 *spout* 完全不同。

```
1 public class TweetsTransactionalSpout extendsBaseTransactionalSpout<TransactionMetadata>{
```

正如你在这个类定义中看到的，TweetsTransactionalSpout 继承了带范型的 **BaseTransactionalSpout**。指定的范型类型的对象是事务元数据集合。它将在后面的代码中用于从数据源分发批次。

在这个例子中，**TransactionMetadata** 定义如下：

```
01 public class TransactionMetadata implements Serializable {
02         private static final long serialVersionUID = 1L;
03         long from;
04         int quantity;
05
06         public TransactionMetadata(long from, int quantity) {
07                 this.from = from;
08                 this.quantity = quantity;
09         }
10 }
```

该类的对象维护着两个属性 **from** 和 **quantity**，它们用来生成批次。

spout 的最后需要实现下面的三个方法：

```
01 @Override
02 public ITransactionalSpout.Coordinator<TransactionMetadata> getCoordinator(
03                 Map conf, TopologyContext context) {
04         return new TweetsTransactionalSpoutCoordinator();
05 }
06
07 @Override
```

```
0  publicbacktype.storm.transactional.ITransactionalSpout.Emitter<TransactionMetada
8  ta> getEmitter(Map conf, TopologyContext contest) {
09         return  new  TweetsTransactionalSpoutEmitter();
10 }
11
12 @Override
13 public  void  declareOutputFields(OuputFieldsDeclarer declarer) {
14         declarer.declare(new  Fields("txid",  "tweet_id",  "tweet"));
15 }
```

getCoordinator 方法，告诉 Storm 用来协调生成批次的类。**getEmitter**，负责读取批次并把它们分发到拓扑中的数据流组。最后，就像之前做过的，需要声明要分发的域。

RQ 类

为了让例子简单点，我们决定用一个类封装所有对 Redis 的操作。

```
01 public  class  RQ {
02         public  static  final  String NEXT_READ =  "NEXT_READ";
03         public  static  final  String NEXT_WRITE =  "NEXT_WRITE";
04
05         Jedis jedis;
06
07         public  RQ() {
08                 jedis =  new  Jedis("localhost");
09         }
10
11         public  long  getavailableToRead(long  current) {
12                 return  getNextWrite() - current;
13         }
14
15         public  long  getNextRead() {
16                 String sNextRead = jedis.get(NEXT_READ);
17                 if(sNextRead ==  null) {
18                         return  1;
19                 }
20                 return  Long.valueOf(sNextRead);
```

```
21          }
22
23          public  long  getNextWrite() {
24                  return  Long.valueOf(jedis.get(NEXT_WRITE));
25          }
26
27          public  void  close() {
28                  jedis.disconnect();
29          }
30
31          public  void  setNextRead(long  nextRead) {
32                  jedis.set(NEXT_READ,  ""+nextRead);
33          }
34
35          public  List<String> getMessages(long  from,  int  quantity) {
36                  String[] keys =  new  String[quantity];
37                  for  (int  i =  0; i < quantity; i++) {
38                          keys[i] =  ""+(i+from);
39                  }
40                  return  jedis.mget(keys);
41          }
42 }
```

仔细阅读每个方法，确保自己理解了它们的用处。

协调者 Coordinator

下面是本例的协调者实现。

```
0 public  static  class  TweetsTransactionalSpoutCoordinator  implementsITransactio
1 nalSpout.Coordinator<TransactionMetadata> {
0
2           TransactionMetadata lastTransactionMetadata;
03          RQ rq =  new  RQ();
04          long  nextRead =  0;
05
06          public  TweetsTransactionalSpoutCoordinator() {
07                  nextRead = rq.getNextRead();
```

```
08          }
09
10          @Override
11          public  TransactionMetadata initializeTransaction(BigInteger txid,
TransactionMetadata prevMetadata) {
12                  long  quantity = rq.getAvailableToRead(nextRead);
13                  quantity = quantity > MAX_TRANSACTION_SIZE ? MAX_TRANSACTION_SIZE :
quantity;
14                  TransactionMetadata ret =  new  TransactionMetadata(nextRead,
(int)quantity);
15                  nextRead += quantity;
16                  return  ret;
17          }
18
19          @Override
20          public  boolean  isReady() {
21                  return  rq.getAvailableToRead(nextRead) >  0;
22          }
23
24          @Override
25          public  void  close() {
26                  rq.close();
27          }
28 }
```

值得一提的是，*在整个拓扑中只会有一个提交者实例*。创建提交者实例时，它会从 redis 读取一个从 1 开始的序列号，这个序列号标识要读取的 tweet 下一条。

第一个方法是 **isReady**。在 **initializeTransaction** 之前调用它确认数据源已就绪并可读取。此方法应当相应的返回 **true** 或 **false**。在此例中，读取 tweets 数量并与已读数量比较。它们之间的不同就在于可读 tweets 数。如果它大于 0，就意味着还有 tweets 未读。

最后，执行 **initializeTransaction**。正如你看到的，它接收 **txid** 和 **prevMetadata** 作为参数。第一个参数是 Storm 生成的事务 ID，作为批次的惟一性标识。**prevMetadata** 是协调器生成的前一个事务元数据对象。

在这个例子中，首先确认有多少 tweets 可读。只要确认了这一点，就创建一个 TransactionMetadata 对象，标识读取的第一个 tweet（译者注：对象属性 **from**），以及读取的 tweets 数量（译者注：对象属性 **quantity**）。

元数据对象一经返回，Storm 把它跟 **txid** 一起保存在 zookeeper。这样就确保了一旦发生故障，Storm 可以利用分发器(译者注：**Emitter**，见下文)重新发送批次。

Emitter

创建事务性 *spout* 的最后一步是实现分发器（Emitter）。实现如下：

```
0 public  static  class  TweetsTransactionalSpoutEmitterimplements  ITransactionalS
1 pout.Emitter<TransactionMetadata> {
0
2
03 </pre>
04 <pre>        RQ rq =  new  RQ();</pre>
05 <pre>        public  TweetsTransactionalSpoutEmitter() {}</pre>
06 <pre>        @Override
07         public  void  emitBatch(TransactionAttempt tx, TransactionMetadata
   coordinatorMeta, BatchOutputCollector collector) {
08             rq.setNextRead(coordinatorMeta.from+coordinatorMeta.quantity);
               List<String> messages = rq.getMessages(coordinatorMeta.from, <span
09 style="font-family: Georgia, 'Times New Roman', 'Bitstream Charter', Times, serif;
   font-size: 13px; line-height: 19px;">coordinatorMeta.quantity);
10 </span>                long  tweetId = coordinatorMeta.from;
11                for  (String message : messages) {
12                        collector.emit(new  Values(tx,  ""+tweetId, message));
13                        tweetId++;
14                }
15        }
16
17        @Override
18        public  void  cleanupBefore(BigInteger txid) {}
19
20        @Override
```

```
21          public  void  close() {
22                  rq.close();
23          }</pre>
24 <pre>
25 }
```

分发器从数据源读取数据并从数据流组发送数据。分发器应当问题能够为相同的事务 id 和事务元数据发送相同的批次。这样，如果在处理批次的过程中发生了故障，Storm 就能够利用分发器重复相同的事务 id 和事务元数据，并确保批次已经重复过了。Storm 会在 **TransactionAttempt** 对象里为尝试次数增加计数（译者注：**attempt id**）。这样就能知道批次已经重复过了。

在这里 **emitBatch** 是个重要方法。在这个方法中，使用传入的元数据对象从 redis 得到 tweets，同时增加 redis 维持的已读 tweets 数。当然它还会把读到的 tweets 分发到拓扑。

Bolts

首先看一下这个拓扑中的标准 *bolt*：

```
01 public  class  UserSplitterBolt  implements  IBasicBolt{
02          private  static  final  long  serialVersionUID = 1L;
03
04          @Override
05          public  void  declareOutputFields(OutputFieldsDeclarer declarer) {
0                   declarer.declareStream("users",  new  Fields("txid","tweet_id","
6  user"));
07          }
08
09          @Override
10          public  Map<String, Object> getComponentConfiguration() {
11                  return  null;
12          }
13
14          @Override
15          public  void  prepare(Map stormConf, TopologyContext context) {}
16
```

```
17        @Override
18        public  void  execute(Tuple input, BasicOutputCollector collector) {
19                String tweet = input.getStringByField("tweet");
20                String tweetId = input.getStringByField("tweet_id");
21                StringTokenizer strTok =  new  StringTokenizer(tweet,  " ");
22                HashSet<String> users =  new  HashSet<String>();
23
24                while(strTok.hasMoreTokens()) {
25                        String user = strTok.nextToken();
26
27                        //确保这是个真实的用户，并且在这个 tweet 中没有重复
28                        if(user.startsWith("@") && !users.contains(user)) {
29                                collector.emit("users",  new  Values(tx,
  tweetId, user));
30                                users.add(user);
31                        }
32                }
33        }
34
35        @Override
36        public  void  cleanup(){}
37 }
```

正如本章前面提到的，**UserSplitterBolt** 接收元组，解析 tweet 文本，分发@开头的单词

————tweeter 用户。**HashtagSplitterBolt** 的实现也非常相似。

```
01 public  class  HashtagSplitterBolt  implements  IBasicBolt{
02        private  static  final  long  serialVersionUID = 1L;
03
04        @Override
05        public  void  declareOutputFields(OutputFieldsDeclarer declarer) {
0                 declarer.declareStream("hashtags",  newFields("txid","tweet_id",
6  "hashtag"));
07        }
08
09        @Override
10        public  Map<String, Object> getComponentConfiguration() {
```

```
11                return  null;
12        }
13
14        @Override
15        public  void  prepare(Map stormConf, TopologyContext context) {}
16
17        @Oerride
18        public  void  execute(Tuple input, BasicOutputCollector collector) {
19                String tweet = input.getStringByField("tweet");
20                String tweetId = input.getStringByField("tweet_id");
21                StringTokenizer strTok =  new  StringTokenizer(tweet,  " ");
22                TransactionAttempt tx =
(TransactionAttempt)input.getValueByField("txid");
23                HashSet<String> words =  new  HashSet<String>();
24
25                while(strTok.hasMoreTokens()) {
26                        String word = strTok.nextToken();
27
28                        if(word.startsWith("#") && !words.contains(word)){
29                                collector.emit("hashtags",  new  Values(tx,
tweetId, word));
30                                words.add(word);
31                        }
32                }
33        }
34
35        @Override
36        public  void  cleanup(){}
37}
```

现在看看 **UserHashTagJoinBolt** 的实现。首先要注意的是它是一个 **BaseBatchBolt**。这意味着，**execute** 方法会操作接收到的元组，但是不会分发新的元组。批次完成时，Storm 会调用 **finishBatch** 方法。

```
01 public  void  execute(Tuple tuple) {
02        String source = tuple.getSourceStreamId();
```

```
03          String tweetId = tuple.getStringByField("tweet_id");
04
05          if("hashtags".equals(source)) {
06                  String hashtag = tuple.getStringByField("hashtag");
07                  add(tweetHashtags, tweetId, hashtag);
08          }  else  if("users".equals(source)) {
09                  String user = tuple.getStringByField("user");
10                  add(userTweets, user, tweetId);
11          }
12 }
```

既然要结合 tweet 中提到的用户为出现的所有话题计数，就需要加入前面的 *bolts* 创建的两个数据流组。这件事要以批次为单位进程，在批次处理完成时，调用 **finishBatch** 方法。

```
01 @Override
02 public  void  finishBatch() {
03          for(String user:userTweets.keySet()){
04                  Set<String> tweets = getUserTweets(user);
05                  HashMap<String, Integer> hashtagsCounter =  new  HashMap<String,
   Integer>();
06                  for(String tweet:tweets){
07                          Set<String> hashtags=getTweetHashtags(tweet);
08                          if(hashtags!=null){
09                                  for(String hashtag:hashtags){
10                                          Integer
   count=hashtagsCounter.get(hashtag);
11                                          if(count==null){count=0;}
12                                          count++;
13                                          hashtagsCounter.put(hashtag,count);
14                                  }
15                          }
16                  }
17                  for(String hashtag:hashtagsCounter.keySet()){
18                          int  count=hashtagsCounter.get(hashtag);
19                          collector.emit(new  Values(id,user,hashtag,count));
20                  }
21          }
```

```
22 }
```

这个方法计算每对用户-话题出现的次数，并为之生成和分发元组。

你可以在 GitHub 上找到并下载完整代码。（译者注：https://github.com/storm-book/examples-ch08-transactional-topologies 这个仓库里没有代码，谁知道哪里有代码麻烦说一声。）

提交者 *bolts*

我们已经学习了，批次通过协调器和分发器怎样在拓扑中传递。在拓扑中，这些批次中的元组以并行的，没有特定次序的方式处理。

协调者 bolts 是一类特殊的批处理 *bolts*，它们实现了 **IComh mitter** 或者通过 **TransactionalTopologyBuilder** 调用 **setCommiterBolt** 设置了提交者 *bolt*。它们与其它的批处理 *bolts* 最大的不同在于，提交者 *bolts* 的 **finishBatch** 方法在提交就绪时执行。这一点发生在之前所有事务都已成功提交之后。另外，**finishBatch** 方法是顺序执行的。因此如果同时有事务 ID1 和事务 ID2 两个事务同时执行，只有在 ID1 没有任何差错的执行了 **finishBatch** 方法之后，ID2 才会执行该方法。

下面是这个类的实现

```
0 public class RedisCommiterCommiterBolt extends BaseTransactionalBolt implem
1 entsICommitter {
02         public static final String LAST_COMMITED_TRANSACTION_FIELD
   = "LAST_COMMIT";
03         TransactionAttempt id;
04         BatchOutputCollector collector;
05         Jedis jedis;
06
07         @Override
08         public void prepare(Map conf, TopologyContext context,
                                                BatchOutputCollector collector,
09 TransactionAttempt id) {
```

```
10              this.id = id;
11              this.collector = collector;
12              this.jedis =  new  Jedis("localhost");
13      }
14
15      HashMap<String, Long> hashtags =  new  HashMap<String,Long>();
16      HashMap<String, Long> users =  new  HashMap<String, Long>();
17      HashMap<String, Long> usersHashtags =  new  HashMap<String, Long>();
18
19      private  void  count(HashMap<String, Long> map, String key,  int  count) {
20              Long value = map.get(key);
21              if(value ==  null){value = (long)0;}
22              value += count;
23              map.put(key,value);
24      }
25
26      @Override
27      public  void  execute(Tuple tuple) {
28              String origin = tuple. getSourceComponent();
29              if("sers-splitter".equals(origin)) {
30                      String user = tuple.getStringByField("user");
31                      count(users, user,  1);
32              }  else  if("hashtag-splitter".equals(origin)) {
33                      String hashtag = tuple.getStringByField("hashtag");
34                      count(hashtags, hashtag,  1);
35              }  else  if("user-hashtag-merger".quals(origin)) {
36                      String hashtag = tuple.getStringByField("hashtag");
37                      String user = tuple.getStringByField("user");
38                      String key = user +  ":"  + hashtag;
39                      Integer count = tuple.getIntegerByField("count");
40                      count(usersHashtags, key, count);
41              }
42      }
43
44      @Override
45      public  void  finishBatch() {
```

```
                String lastCommitedTransaction =
46 jedis.get(LAST_COMMITED_TRANSACTION_FIELD);
47                 String currentTransaction =  ""+id.getTransactionId();
48
49                 if(currentTransaction.equals(lastCommitedTransaction)) {return;}
50
51                 Transaction multi = jedis.multi();
52
53                 multi.set(LAST_COMMITED_TRANSACTION_FIELD, currentTransaction);
54
55                 Set<String> keys = hashtags.keySet();
56                 for  (String hashtag : keys) {
57                         Long count = hashtags.get(hashtag);
58                         multi.hincrBy("hashtags", hashtag, count);
59                 }
60
61                 keys = users.keySet();
62                 for  (String user : keys) {
63                         Long count =users.get(user);
64                         multi.hincrBy("users",user,count);
65                 }
66
67                 keys = usersHashtags.keySet();
68                 for  (String key : keys) {
69                         Long count = usersHashtags.get(key);
70                         multi.hincrBy("users_hashtags", key, count);
71                 }
72
73                 multi.exec();
74         }
75
76         @Override
77         public  void  declareOutputFields(OutputFieldsDeclarer declarer) {}
78 }
```

这个实现很简单，但是在 **finishBatch** 有一个细节。

```
1...
2 multi.set(LAST_COMMITED_TRANSACTION_FIELD, currentTransaction);
3...
```

在这里向数据库保存提交的最后一个事务 ID。为什么要这样做？记住，如果事务失败了，Storm 将会尽可能多的重复必要的次数。如果你不确定已经处理了这个事务，你就会多算，事务拓扑也就没有用了。所以请记住：保存最后提交的事务 ID，并在提交前检查。

分区的事务 *Spouts*

对一个 *spout* 来说，从一个分区集合中读取批次是很普通的。接着这个例子，你可能有很多 redis 数据库，而 tweets 可能会分别保存在这些 redis 数据库里。通过实现 **IPartitionedTransactionalSpout**，Storm 提供了一些工具用来管理每个分区的状态并保证重播的能力。

下面我们修改 **TweetsTransactionalSpout**，使它可以处理数据分区。

首先，继承 **BasePartitionedTransactionalSpout**，它实现了 **IPartitionedTransactionalSpout**。

```
1 public  class  TweetsPartitionedTransactionalSpout  extends
2              BasePartitionedTransactionalSpout<TransactionMetadata> {
3...
4 }
```

然后告诉 Storm 谁是你的协调器。

```
0 public  static  class  TweetsPartitionedTransactionalCoordinator  implementsCoor
1 dinator {
02        @Override
03        public  int  numPartitions() {
04                return  4;
05        }
06
07        @Override
08        public  boolean  isReady() {
09                return  true;
10        }
11
```

```
12          @Override
13          public  void  close() {}
14 }
```

在这个例子里，协调器很简单。numPartitions 方法，告诉 Storm 一共有多少分区。而且你要注意，不要返回任何元数据。对于 **IPartitionedTransactionalSpout**，元数据由分发器直接管理。下面是分发器的实现：

```
01 public  static  class  TweetsPartitionedTransactionalEmitter
02               implements  Emitter<TransactionMetadata> {
03          PartitionedRQ rq =  new  ParttionedRQ();
04
05          @Override
06          public  TransactionMetadata emitPartitionBatchNew(TransactionAttempt tx,
07                                  BatchOutputCollector collector,  int  partition,
08                                  TransactionMetadata lastPartitioonMeta) {
09                  long  nextRead;
10
11                  if(lastPartitionMeta ==  null) {
12                          nextRead = rq.getNextRead(partition);
13                  }else{
14                          nextRead = lastPartitionMeta.from +
lastPartitionMeta.quantity;
15                          rq.setNextRead(partition, nextRead);  //移动游标
16                  }
17
18                  long  quantity = rq.getAvailableToRead(partition, nextRead);
19                  quantity = quantity > MAX_TRANSACTION_SIZE ? MAX_TRANSACTION_SIZE :
quantity;
20                  TransactionMetadata metadata
=  new  TransactionMetadata(nextRead, (int)quantity);
21
22                  emitPartitionBatch(tx, collector, partition, metadata);
23                  return  metadata;
24          }
25
```

```
26          @Override
27          public  void  emitPartitionBatch(TransactionAttempt tx,
BatchOutputCollector collector,
28                          int  partition, TransactionMetadata partitionMeta) {
29                  if(partitionMeta.quantity <=  0){
30                          return;
31                  }
32
33                  List<String> messages = rq.getMessages(partition,
partitionMeta.from,
34                                  partitionMeta.quantity);
35
36                  long  tweetId = partitionMeta.from;
37                  for  (String msg : messages) {
38                          collector.emit(new  Values(tx,  ""+tweetId, msg));
39                          tweetId++;
40                  }
41          }
42
43          @Override
44          public  void  close() {}
45 }
```

这里有两个重要的方法，**emitPartitionBatchNew**，和 **emitPartitionBatch**。对于 **emitPartitionBatchNew**，从 Storm 接收分区参数，该参数决定应该从哪个分区读取批次。在这个方法中，决定获取哪些 tweets，生成相应的元数据对象，调用 **emitPartitionBatch**，返回元数据对象，并且元数据对象会在方法返回时立即保存到 zookeeper。

Storm 会为每一个分区发送相同的事务 ID，表示一个事务贯穿了所有数据分区。通过 **emitPartitionBatch** 读取分区中的 tweets，并向拓扑分发批次。如果批次处理失败了，Storm 将会调用 **emitPartitionBatch** 利用保存下来的元数据重复这个批次。

NOTE: 完整的源码请见：

https://github.com/storm-book/examples-ch08-transactional-topologies（译者注：原文如此，实际上这个仓库里什么也没有）

模糊的事务性拓扑

到目前为止，你可能已经学会了如何让拥有相同事务 ID 的批次在出错时重播。但是在有些场景下这样做可能就不太合适了。然后会发生什么呢？

事实证明，你仍然可以实现在语义上精确的事务，不过这需要更多的开发工作，你要记录由 Storm 重复的事务之前的状态。既然能在不同时刻为相同的事务 ID 得到不同的元组，你就需要把事务重置到之前的状态，并从那里继续。

比如说，如果你为收到的所有 tweets 计数，你已数到 5，而最后的事务 ID 是 321，这时你多数了 8 个。你要维护以下三个值——previousCount=5,currentCount=13，以及 lastTransactionId=321。假设事物 ID321 又发分了一次，而你又得到了 4 个元组，而不是之前的 8 个，提交器会探测到这是相同的事务 ID，它将会把结果重置到 **previousCount** 的值 5，并在此基础上加 4，然后更新 **currentCount** 为 9。

另外，在之前的一个事务被取消时，每个并行处理的事务都要被取消。这是为了确保你没有丢失任何数据。

你的 *spout* 可以实现 **IOpaquePartitionedTransactionalSpout**，而且正如你看到的，协调器和分发器也很简单。

```
0 public  static  class  TweetsOpaquePartitionedTransactionalSpoutCoordinator  impl
1 ementsIOpaquePartitionedTransactionalSpout.Coordinator {
0
2         @Override
03        public  boolean  isReady() {
04                return  true;
05        }
06 }
07
08 public  static  class  TweetsOpaquePartitionedTransactionalSpoutEmitter
0               implements  IOpaquePartitionedTransactionalSpout.Emitter<Transacti
9 onMetadata> {
```

```
10          PartitionedRQ rq   =  new  PartitionedRQ();
11
12          @Override
13          public  TransactionMetadata emitPartitionBatch(TransactionAttempt tx,
14                        BatchOutputCollector collector,  int  partion,
15                        TransactionMetadata lastPartitonMeta) {
16                  long  nextRead;
17
18                  if(lastPartitionMeta ==  null) {
19                          nextRead = rq.getNextRead(partition);
20                  }else{
21                          nextRead = lastPartitionMeta.from +
lastPartitionMeta.quantity;
22                          rq.setNextRead(partition, nextRead);//移动游标
23                  }
24
25                  long  quantity = rq.getAvailabletoRead(partition, nextRead);
26                  quantity = quantity > MAX_TRANSACTION_SIZE ? MAX_TRANSACTION_SIZE :
quantity;
27                  TransactionMetadata metadata
=  new  TransactionMetadata(nextRead, (int)quantity);
28                  emitMessages(tx, collector, partition, metadata);
29                  return  metadata;
30          }
31
32          private  void  emitMessage(TransactionAttempt tx, BatchOutputCollector
collector,
33                                      int  partition, TransactionMetadata
partitionMeta) {
34                  if(partitionMeta.quantity <=  0){return;}
35
36                  List<String> messages = rq.getMessages(partition,
partitionMeta.from, partitionMeta.quantity);
37                  long  tweetId = partitionMeta.from;
38                  for(String msg : messages) {
39                          collector.emit(new  Values(tx,  ""+tweetId, msg));
40                          tweetId++;
```

```
41                  }
42          }
43
44          @Override
45          public  int  numPartitions() {
46                  return  4;
47          }
48
49          @Override
50          public  void  close() {}
51 }
```

最有趣的方法是 **emitPartitionBatch**，它获取之前提交的元数据。你要用它生成批次。这个批次不需要与之前的那个一致，你可能根本无法创建完全一样的批次。剩余的工作由提交器 *bolts* 借助之前的状态完成。

Storm 入门之附录 A

安装 Storm 客户端

Storm 客户端能让我们使用命令管理集群中的拓扑。按照以下步骤安装 Storm 客户端：

1. 从 Storm 站点下载最新的稳定版本（https://github.com/nathanmarz/storm/downloads）当前最新版本是 storm-0.8.1。（译者注：原文是 storm-0.6.2，不过翻译的时候已经是 storm-0.8.1 了）
2. 把下载的文件解压缩到*/usr/local/bin/storm* 的 Storm 共享目录。
3. 把 Storm 目录加入 PATH 环境变量，这样就不用每次都输入全路径执行 Storm 了。如果我们使用了*/usr/local/bin/storm*，执行 *export PATH=$PATH:/usr/local/bin/storm*。
4. 最后，创建 Storm 本地配置文件：*~/.storm/storm.yaml*，在配置文件中按如下格式加入 *nimbus* 主机：

   nimbus.host:"我们的 nimbus 主机"

现在，你可以管理你的 Storm 集群中的拓扑了。

NOTE：Storm 客户端包含运行一个 Storm 集群所需的所有 Storm 命令，但是要运行它你需要安装一些其它的工具并做一些配置。详见附录 B。

有许多简单且有用的命令可以用来管理拓扑，它们可以提交、杀死、禁用、再平衡拓扑。

jar 命令负责把拓扑提交到集群，并执行它，通过 **StormSubmitter** 执行主类。

```
storm jar path-to-topology-jar class-with-the-main arg1 arg2 argN
```

path-to-topology-jar 是拓扑 jar 文件的全路径，它包含拓扑代码和依赖的库。class-with-the-main 是包含 main 方法的类，这个类将由 StormSubmitter 执行，其余的参数作为 main 方法的参数。

我们能够挂起或停用运行中的拓扑。当停用拓扑时，所有已分发的元组都会得到处理，但是 *spouts* 的 **nextTuple** 方法不会被调用。

停用拓扑：

```
storm deactivte topology-name
```

启动一个停用的拓扑：

```
storm activate topology-name
```

销毁一个拓扑，可以使用 **kill** 命令。它会以一种安全的方式销毁一个拓扑，首先停用拓扑，在等待拓扑消息的时间段内允许拓扑完成当前的数据流。

杀死一个拓扑：

```
storm kill topology-name
```

NOTE:执行 kill 命令时可以通过**-w [等待秒数]**指定拓扑停用以后的等待时间。

再平衡使你重分配集群任务。这是个很强大的命令。比如，你向一个运行中的集群增加了节点。

再平衡命令将会停用拓扑，然后在相应超时时间之后重分配工人，并重启拓扑。

再平衡拓扑：

```
storm rebalance topology-name
```

NOTE:执行不带参数的 Storm 客户端可以列出所有的 Storm 命令。完整的命令描述请见：https://github.com/nathanmarz/storm/wiki/Command-line-client。

Storm 入门之附录 B

安装 Storm 集群

译者注：本附录的内容已经有些陈旧了。最新的 Storm 已不再必须依赖 ZeroMQ，各种依赖的库和软件也已经有更新的版本。

有以下两种方式创建 Storm 集群：

- 使用 Storm 部署在亚马逊 EC2 上面创建一个集群，就像你在第 6 章看到的。
- 手工安装（详见本附录）

要手工安装 Storm，需要先安装以下软件

- Zookeeper 集群（安装方法详见管理向导）
- Java6.0
- Python2.6.6
- Unzip 命令

NOTE: Nimbus 和管理进程将要依赖 Java、Python 和 unzip 命令

安装本地库：

安装 ZeroMQ：

```
01 wget http://download.zeromq.org/historic/zeromq-2.1.7.tar.gz
02
03 tar -xzf zeromq-2.1.7.tar.gz
04
05 cd zeromq-2.1.7
06
07 ./configure
08
09 make
10
11 sudo make install
```

安装 JZMQ：

```
1 git clone https://github.com/nathanmarz/jzmq.git
2 cd jzmq
3 ./autogen.sh
4 ./configure
5 make
6 sudo make install
```

本地库安装完了，下载最新的 Storm 稳定版（写作本书时是 Storm0.7.1。译者注：翻译本章时已是 v0.9.1，可从 http://storm.incubator.apache.org/或 https://github.com/apache/incubator-storm/releases 下载），并解压缩。

编辑配置文件，增加 Storm 集群配置（可以从 Storm 仓库的 defaults.yaml 看到所有的默认配置）。

编辑 Storm 目录下的 *conf/storm.yaml*，添加以下参数，增加集群配置：

```
storm.zookeeper.servers:
– "zookeeper addres 1"
– "zookeeper addres 2"
– "zookeeper addres N"
```

storm.local.dir: "a local directory"

nimbus.host: "Nimbus host addres"

supervisor.slots.ports:

– supervisor slot port 1

– supervisor slot port 2

– supervisor slot port N

参数解释：

storm.zookeeper.servers

你的 zookeeper 服务器地址。

storm.local.dir:

Storm 进程保存内部数据的本地目录。（务必保证运行 Storm 进程的用户拥有这个目录的写权限。）

nimbus.host

Nimbus 运行的机器的地址

supervisor.slots.ports

接收消息的工人进程监听的端口号（通常从 6700 开始）；管理进程为这个属性指定的每个端口号运行一个工人进程。

当你完成了这些配置，就可以运行所有的 Storm 进程了。如果你想运行一个本地进程测试一下，就把 *nimbus.host* 配置成 localhost。

启动一个 Storm 进程，在 Storm 目录下执行：*./bin/storm 进程名*。

NOTE：Storm 提供了一个出色的叫做 Storm UI 的工具，用来辅助监控拓扑。

Storm 入门之附录 C

安装实际的例子

译者注：有些软件的最新版本已有变化，译文不会完全按照原文翻译，而是列出当前最新版本的软件。

首先，从下述 GitHub 的 URL 克隆这个例子：

1 > git clone git://github.com/storm-book/examples-ch06-real-life-app.git

src/main

包含拓扑的源码

src/test

包含拓扑的测试用例

webapps 目录

包含 Node.js Web 可以执行拓扑应用

```
.
├── pom.xml
├── src
│   ├── main
```

```
│ │ └── java
│ └── test
│ └── groovy
└── webapp
```

安装 Redis

Redis 的安装是相当简单的：

1. 从 Redis 站点下载最新的稳定版（译者注：翻译本章时最新版本是 2.8.9。）
2. 解压缩
3. 运行 **make**，和 **make install**。

上述命令会编译 Redis 并在 PATH 目录（译者注：/usr/local/bin）创建可执行文件。

可以从 Redis 网站上获取更多信息，包括相关命令文档及设计理念。

安装 Node.js

安装 Node.js 也很简单。从 http://www.nodejs.org/#download 下载最新版本的 Node.js 源码。

当前最新版本是 v0.10.28

下载完成，解压缩，执行

```
1 <b>./configure</b>
2
3 <b>make</b>
4
5 <b>make install</b>
```

可以从官方站点得到更多信息，包括在不同平台上安装 Node.js 的方法。

构建与测试

为了构建这个例子，需要先启动 *redis-server*

>nohup redis-server &

然后执行 mvn 命令编译并测试这个应用。

>mvn package

…

[INFO] ————————————————————————

[INFO] BUILD SUCCESS

[INFO] ————————————————————————

[INFO] Total time: 32.163s

[INFO] Finished at: Sun Jun 17 18:55:10 GMT-03:00 2012

[INFO] Final Memory: 9M/81M

[INFO]

运行拓扑

启动了 *redis-service* 并成功构建之后，在 *LocalCluster* 启动拓扑。

>java -jar target/storm-analytics-0.0.1-jar-with-dependencies.jar

启动拓扑之后，用以下命令启动 Node.js Web 应用：

>node webapp/app.js

NOTE：拓扑和 Node.js 命令会互相阻塞。尝试在不同的终端运行它们。

演示这个例子

在浏览器输入 http://localhost:3000/开始演示这个例子！

关于作者

Jonathan Leibiusky，MercadoLibre 的主要研究与开发人员，已在软件开发领域工作逾 10 年之久。他已为诸多开源项目贡献过源码，包括“Jedis”，它在 VMware 和 SpringSource 得到广泛使用。

Gabriel Eisbruch 一位计算机科学学生，从 2007 年开始在 Mercadolibre(NASDAQ MELI)任架构师。主要负责研究与开发软件项目。去年，他专门负责大数据分析，为 MercadoLibre 实现了 Hadoop 集群。

Dario Simonassi 在软件开发领域有 10 年以上工作经验。从 2004 年开，他专门负责大型站点的操作与性能。现在他是 MercadoLibre(NASDAQ MELI)的首席架构师，领导着该公司的架构师团队。