Getting Started with Storm 中文版

韩飞　译
Mail : aaron.han.1986@gmail.com
QQ : 362504281
Blog : http://blog.csdn.net/lonelytrooper

写在前面的话：翻译这本书纯因为个人对 storm 的兴趣，水平有限，希望大家多多指正，我自己也会不断改进其中的内容。 希望这本书能对 storm 的入门爱好者起到一定的帮助作用。希望此版本仅用作学习交流，不要出现挂在论坛赚积分、甚至赚钱的无耻行为。

2013.9.30

第一章 基础

Storm 是一套分布式的、可靠的，可容错的用于处理流式数据的系统。处理工作会被委派给不同类型的组件，每个组件负责一项简单的、特定的处理任务。Storm 集群的输入流由名为 spout 的组件负责。Spout 将数据传递给名为 bolt 的组件，后者将以某种方式处理这些数据。例如 bolt 以某种存储方式持久化这些数据，或者将它们传递给另外的 bolt。你可以把一个 storm 集群想象成一条由 bolt 组件组成的链，每个 bolt 对 spout 暴露出来的数据做某种方式的处理。

为了说明这个概念，这里有一个简单的示例。昨天晚上在我看新闻时，播音员们开始谈论政治家以及他们阵营的各种话题。播音员们一直重复着不同的名字，于是我想知道是否每个名字被提及了相同的次数，或者提到的次数是否有偏重。

把播音员们说的字幕认为成你的数据输入流。你可以让 spout 来从一个文件(或者套接字，通过 HTTP，或者一些其他方法)读取输入。当文本行到达时，spout 将它们交给一个 bolt，该 bolt 将文本行流分隔成单词。单词流被传递到另一个 bolt，在这个 bolt 里，每个单词会被与一个预先定义好的政治家名单列表作比较。每作一次比较，第二个 bolt 会在数据库中增加一次那个名字的计数。当你想查看结果时，你只要查询数据库，该数据库在数据到达时被会实时更新。所有组件的排列(spouts 和 bolts)及它们的连接被称为一个 topology(见图 1-1)。

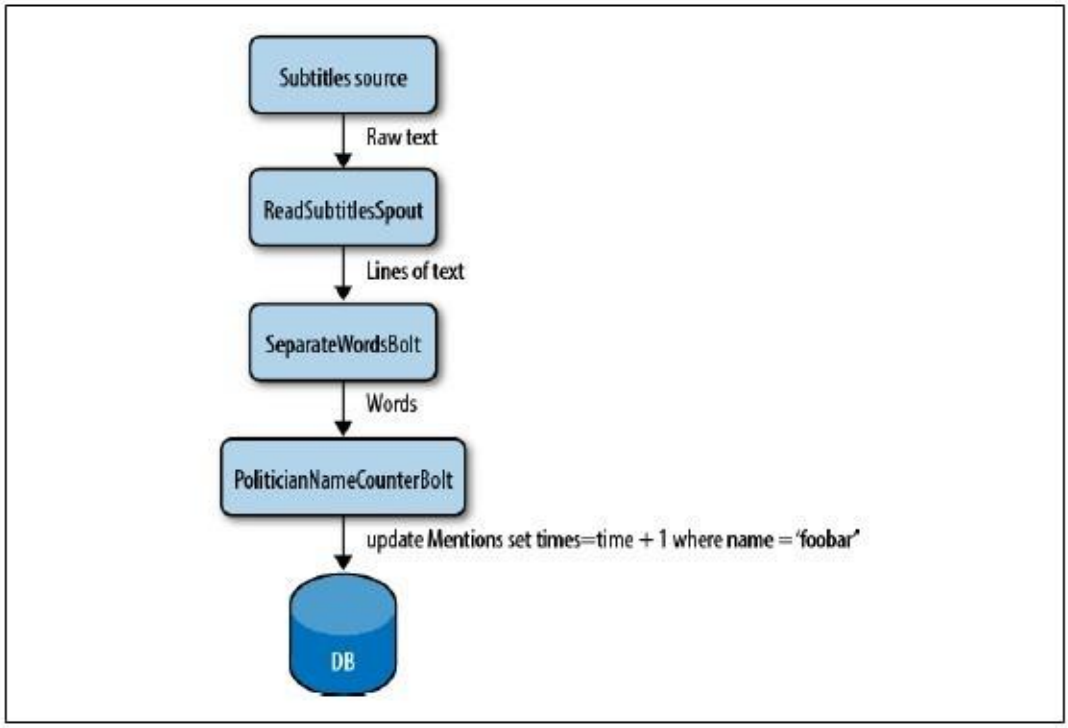

图 1-1. 一个简单的 topology

现在想象你可以简单地定义整个集群中每个 bolt 和 spout 的并行度，这样你就可以无限地扩展你的 topology。很神奇，不是吗？尽管这只是一个简单的例子，你已经可以看到 storm 有多强大。

storm 的典型用例有哪些呢？
流处理
　　正如前述示例中演示的，与其他的流处理系统不同，使用 storm 不需要中间队列。
持续计算
　　向客户端持续发送数据，这样它们可以实时更新并显示结果，例如站点度量。
分布式远程过程调用
　　简单地并行化 cpu 密集型操作。

Storm 组件

在 storm 集群中，结点被一个持续运行的主结点管理。

Storm 集群中有两种结点：主结点和工作结点。主结点运行一个叫做 Nimbus 的守护进程，它负责在集群内分发代码，为每个工作结点指派任务和监控失败的任务。工作结点运行一个叫做 Supervisor 的守护进程，它执行 topology 的一部分。Storm 中的 topology 运行在不同机器的许多工作结点上。

因为 storm 保存所有集群状态在 zookeeper 或者本地磁盘上，因此这些守护进程是无状态的，并且可以在对系统健康无影响的情况下失败或者重启(见图 1-2)。

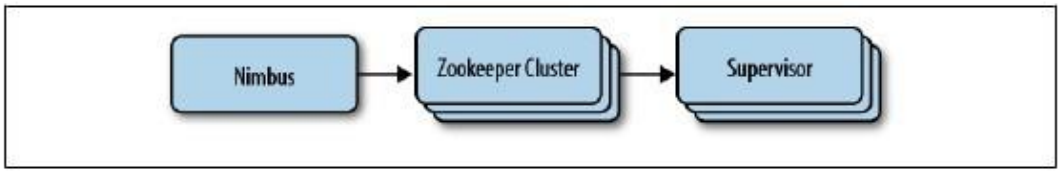

图 1-2 storm 集群的组件

在底层，storm 使用了 zeromq(Omq,zeromq)，一种高级的，可嵌入的、提供极好特性的网络库，这使得 storm 成为可能。我们列举一下 zeromq 的一些特性：
. 充当并行框架的套接字库
. 比 TCP 更快速，支持集群产品和超级计算
. 在进程内、IPC、TCP 和多播之间传递数据
. 异步 IO 支持可扩展的多核消息传递应用
. 通过展开、发布订阅、管道、请求应答连接 N-to-N

Storm 只使用 push/pull 套接字。

Storm 属性

在所有的这些设计理念及决策中，有一些非常好的属性使得 storm 与众不同。
编程简单

如果你尝试过从头开始构建实时处理系统，你会明白它有多痛苦。通过 storm，复杂性被引人注目的减少了。

支持多种编程语言

使用基于 JVM 的开发语言很容易，但是 storm 支持任何语言只要你使用或者实现一个小的中间库。

容错

Storm 关注 worker 数量的下降并在必要时刻对任务进行重分配。

可扩展

对于扩展，你所有要做的只是为集群增加更多的机器。Storm 会为新机器分配任务当它们可用的时候。

可靠

所有的消息都确保至少处理一次。如果有错误，消息可能会被处理不止一次，但是你永远不会丢失数据。

快速

速度曾是驱动 storm 设计的关键因素。

事务

对于几乎任何计算，你可以获取确切的一次消息语义。

第二章 开始

在本章中，我们会建一个 storm 工程和我们的第一个 storm topology。

下述假设你安装了至少1.6版本的Java运行时环境(JRE)。我们推荐使用oracle提供的JRE，可以在这里找到 http://www.java.com/downloads/。

操作模式

在我们开始之前，理解 storm 的操作模式很重要。有两种方式运行 storm。

本地模式

在本地模式中，storm topologies 运行在本地机器一个单独的 JVM 中。由于是最简单的查看所有的 topology 组件一起工作的模式，这种方式被用来开发，测试和调试。在这种模式下，我们可以调整参数，这使得我们可以看到我们的 topology 在不同的 storm 配置环境下是怎么运行的。为了以本地模式运行 topologies，我们需要下载 storm 的开发依赖包，其中包含了我们开发和测试 topology 所需的所有东西。

当我们建立自己的第一个 storm 工程的时候我们很快就可以看到是怎么回事了。

在本地模式运行 topology 与在 storm 集群中运行它是类似的。然而，确保所有的组件线程安全是重要的，因为当它们被部署到远程模式中时，它们可能运行在不同的 JVM 中或者在不同的物理机器上，这样的话，它们之间没有直接的交流或者内存共享。
本章的所有示例，我们都以本地模式运行。

远程模式

在远程模式中，我们提交 topology 到 storm 集群，该集群由许多进程组成，通常运行在不同的机器上。远程模式不显示调试信息，这也是它被认为是生产模式的原因。然而，在一台单独的开发机器上建立 storm 集群是可能的，并且它被认为是在部署至生产前的一个好方法，这可以确保在生产环境中运行 topology 时没有任何问题。
你在第六章中可以了解到更多关于远程模式的内容，我会在附录 B 里展示怎样安装一个集群。

Hello World Storm

在这个工程中，我们会建立一个简单的 topology 来为单词计数。我们可以把这个工程认为是 storm topologies 的“hello world”。然而，它是一个非常强大的 topology，因为它只需要做一些小的改动便可以扩展到几乎无限规模并且，我们甚至可以用它来做一个统计系统。例如，我们可以修改这个项目来找出 Twitter 上的话题趋势。
为了建立这个 topology，我们将使用一个 spout 来负责读取单词，第一个 bolt 来标准化单词，第二个 bolt 来为单词计数，正如我们在图 2-1 中看到的那样。

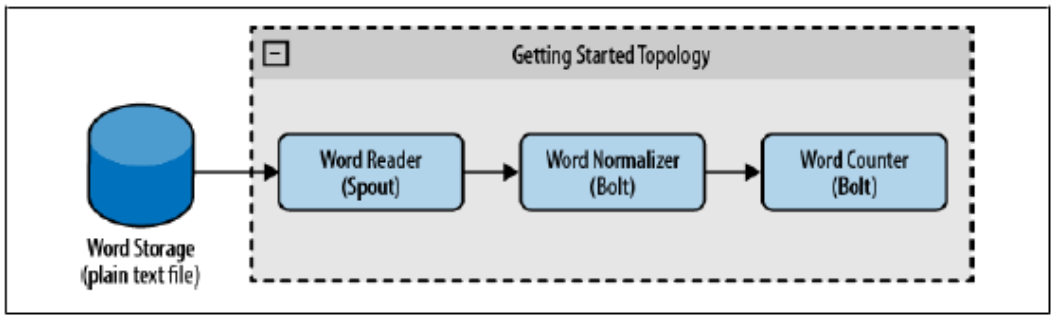

图 2-1 开始 topology
你可以在 https://github.com/storm-book/examples-ch02-getting_started/zipball/master 下载示例源代码的 ZIP 文件。

如果你使用 git(一个分布式的校正控制及源代码管理工具)，你可以在你想要下载的源代码的目录中运行 git clone git@github.com:storm-book/examplesch02-getting_started.git。

验证 Java 安装

安装环境的第一步是验证你正在运行的 java 的版本。打开一个终端窗口，运行命令 java –version。我们可以看到如下类似的信息：

```
java -version
java version "1.6.0_26"
Java(TM) SE Runtime Environment (build 1.6.0_26-b03)
Java HotSpot(TM) Server VM (build 20.1-b02, mixed mode)
```

如果没有，验证下你的 java 安装。(见 http://www.java.com/download/.)

创建工程

为开始这个工程，先建立一个用来存放应用的文件夹(就像对任何的 java 应用一样)。该文件夹包含工程的源代码。
接着我们需要下载 storm 的依赖包：一个我们将添加到应用类路径的 jar 包的集合。你可以用两种方式中的一种做这件事：

. 下载依赖包，解压，添加到类路径。
. 使用 Apache Maven

maven 是一套软件工程管理的工具。它可以被用来管理软件开发周期中的多个方面，从依赖到发布构建过程。在本书中我们会广泛的使用它。为验证是否已安装了 maven，运行命令 mvn。如果没有，可以 从 http://maven.apache.org/download.html 下载。尽管使用 storm 没有必要成为一个 maven 专家，但是知道 maven 是怎样工作的基础知识是有帮助的。你可以找到更多信息在 Apache Maven 的网站(http://maven.apache.org/)。
为了定义工程的结构，我们需要建立一个pom.xml(工程对象模型)文件，该文件描述依赖，包，源码等。我们将使用依赖包及nathanmarz建立的maven库(https://github.com/nathanmarz/)。这些依赖可以在这里找到https://github.com/nathanmarz/storm/wiki/Maven。

storm 的 maven 依赖包引用了在本地模式运行 storm 所需的所有库函数。
使用这些依赖包，我们可以写一个包含运行 topology 基本的必要组件的 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>
```

前几行指定了工程的名字和版本。然后我们添加了一个编译器插件，该插件告诉maven我们的代码应该用Java1.6编译。接下来我们定义库(maven支持同一工程的多个库)。Clojars是storm依赖包所在的库。Maven会自动下载本地模式运行storm所需的所有子依赖包。

应用有如下的结构，典型的maven java工程：

```
our-application-folder/
        ├── pom.xml
        └── src
            └── main
                └── java
                |   ├── spouts
                |   └── bolts
                └── resources
```

Java下的文件夹包含我们的源代码并且我们会将我们的单词文件放到resources文件夹中来处理。


mkdir –p 建立所有所需的父目录。

建立我们第一个 Topology

为建立我们第一个 topology，我们要创建运行单词计数的所有的类。或许示例的一些部分在目前阶段不是很清晰，我们将在后边的章节中解释它们。

Spout

WordReader spout 是实现了 IRichSpout 接口的类。我们在第四章会看到更多的细节。WordReader 负责读文件并且将每行提供给一个 bolt。


一个 spout 发射一个定义的域的列表。这个架构允许你有多种 bolt 读取相同的 spout 流，然后这些 bolt 定义域供其他的 bolt 消费等等。

示例 2-1 包含了这个类的完整代码(我们在示例后分析代码的每个部分)。

Example 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);
}
/**
* The only thing that the methods will do It is emit each
* file line
*/
public void nextTuple() {
/**
* The nextuple it is called forever, so if we have been readed the file
* we will wait and then return
*/
    if(completed){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        //Do nothing
        }
        return;
    }
    String str;
    //Open the reader
    BufferedReader reader = new BufferedReader(fileReader);
    try{
        //Read all lines
        while((str = reader.readLine()) != null){
            /**
            * By each line emmit a new value with the line as a their
            */
            this.collector.emit(new Values(str),str);
        }
    }catch(Exception e){
        throw new RuntimeException("Error reading tuple",e);
    }finally{
        completed = true;
    }
```

```
    }
    /**
     * We will create the file and get the collector object
     */
    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;
    }
    /**
     * Declare the output field "word"
     */
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("line"));
    }
}
```

在任何spout中都调用的第一个方法是void open(Map conf, TopologyContext context, SpoutOutputCollector collector)。方法的参数是 TopologyContext，它包含了所有的 topology 数据；conf 对象，它在 topology 定义的时候被创建；SpoutOutputCollector，它使得我们可以发射将被 bolt 处理的数据。下面的代码是 open 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;
    }
```

在这个方法中，我们也创建了 reader，它负责读文件。接着我们需要实现 public void nextTuple()，在这个方法里我们发射将被 bolt 处理的值。在我们的例子中，这个方法读文件并且每行发射一个值。

```
    public void nextTuple() {
        if(completed){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
```

```
                //Do nothing
            }
            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;
        }
    }
```

Values 是 ArrayList 的一个实现，其中把 list 的元素传到了构造方法中。

nextTuple()方法在相同的循环中被周期性的调用，正如 ack()和 fail()方法。当没有工作要做时，必须释放对线程的控制这样其他的方法有机会被调用。所以 nextTuple 方法的第一行是检查处理是否完成了。如果已经完成，在返回前它会休眠至少一毫秒来降低处理器的负载。如果有工作要做，那么文件的每一行被读取为一个值并且发射。

元组(Tuple)是一个值的命名列表，它可以是任何类型的 java 对象(只要这个对象是可序列化的)。Storm 在缺省的情况下可以序列化常用的类型例如 strings，byte arrays，ArrayList，HashMap 和 HashSet。

Bolts

现在我们有了一个 spout 来读取文件并且每一行发射一个元组。我们需要建立两个 bolt 来处理元组(见图 2-1)。这些 bolts 实现了 backtype.storm.topology.IRichBolt 接口。
Bolt 最重要的方法是 void execute(Tuple input)，每收到一个元组调用一次。对于每个收到的元组，bolt 会发射出一些元组。

bolt 或者 spout 可以发射如所需一样多的元组。当 nextTuple 或 execute 方法被调用时，它们可能发射 0 个，1 个或多个元组。你将在第五章了解到更多。

第一个 bolt，WordNormalizer，负责获取行并且标准化行。它会将行分隔成单词，将单词转化成小写并且 trim 单词。
首先我们要声明 bolt 的输出参数：

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

这里我们声明 bolt 将输出一个域，名为 word。
接着我们实现 public void execute(Tuple input)方法，在这里输入的元组将被处理：

```
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();
            //Emit the word
            collector.emit(new Values(word));
        }
    }
    // Acknowledge the tuple
    collector.ack(input);
}
```

第一行读取元组的值。值可以通过位置或者名字读取。值被处理然后使用 collector 对象发射。在每个元组被处理完成后，collector 的 ack()方法被调用来表明处理被成功的完成。如果该元组不能被处理，应该调用 collector 的 fail()方法。
示例 2-2 包含这个类的完整代码。
Example 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() {}
    /**
    * The bolt will receive the line from the
    * words file and process it to Normalize this line
```

```
     *
     * The normalize will be put the words in lower case
     * and split the line to get all words in this
     */
    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();
                //Emit the word
                List a = new ArrayList();
                a.add(input);
                collector.emit(a,new Values(word));
            }
        }
        // Acknowledge the tuple
        collector.ack(input);
    }
    public void prepare(Map stormConf, TopologyContext context,
            OutputCollector collector) {
        this.collector = collector;
    }
    /**
     * The bolt will only emit the field "word"
     */
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

在这个类中，我们看到了在一个单独的 execute 方法中发射多个元组的例子。如果方法收到句子This is the Storm book，在一个单独的execute方法中，它将发射五个元组。

下一个 bolt，WordCounter，负责为单词计数。当 topology 结束的时候(cleanup()方法被调用时)，我们会显示每个单词的计数。

这是一个 bolt 不发射任何东西的示例。在这个例子中，数据被加到一个 map 中，

但在实际中，bolt 会将数据存到数据库中。

```
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;
    /**
    * At the end of the spout (when the cluster is shutdown
    * We will show the word counters
    */
    @Override
    public void cleanup() {
        System.out.println("-- Word Counter ["+name+"-"+id+"] --");
        for(Map.Entry<String, Integer> entry : counters.entrySet()){
            System.out.println(entry.getKey()+": "+entry.getValue());
        }
    }
    /**
    * On each word We will count
    */
    @Override
    public void execute(Tuple input) {
        String str = input.getString(0);
        /**
        * If the word dosn't exist in the map we will create
        * this, if not We will add 1
        */
        if(!counters.containsKey(str)){
            counters.put(str, 1);
        }else{
            Integer c = counters.get(str) + 1;
            counters.put(str, c);
        }
        //Set the tuple as Acknowledge
        collector.ack(input);
    }
```

```
    /**
     * On create
     */
    @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 方法是用 map 来收集和计数单词。当 topology 结束的时候，cleanup()方法被调用并且打印出计数的 map。(这只是一个示例，通常来讲你应该在 cleanup()方法里关闭有效的连接和其他资源当 topology 关闭的时候。)

主类

在主类中，你会建立 topology 和 LocalCluster 对象，它使你可以在本地测试和调试 topology。与 Config 对象结合，LocalCluster 允许你尝试不同的集群配置。例如，当一个全局或者类变量被不慎使用时，在使用不同数量的 workers 配置来测试你的 topology 时你会找到这个错误。(在第三章你将看到更多关于 config 对象。)

所有的 topology 结点应该可以在进程间没有数据共享的情形下独立运行(例如，没有全局或类变量)，因为 topology 在实际的集群中运行时，这些进程可能运行在不同的机器上。

你将使用 TopologyBuilder 创建 topology，topology 告诉 storm 结点是怎么安排的并且它们之间怎样交换数据。

```
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("word-reader",new WordReader());
    builder.setBolt("word-normalizer", new WordNormalizer()).shuffleGrouping("wordreader");
    builder.setBolt("word-counter", new WordCounter()).shuffleGrouping("wordnormalizer");
```

spout 和 bolts 通过 shuffleGroupings 连接起来。这种分组(grouping)告诉 storm 在源结点和目标结点之间以随机分布的方式发送消息。

接着，建立包含 topology 配置的 Config 对象，该配置在运行时会被与集群的配置合并并且通过 prepare 方法发送到所有结点。

```
    Config conf = new Config();
    conf.put("wordsFile", args[0]);
    conf.setDebug(true);
```

对将要被 spout 读取的文件名设置属性 wordsFile，因为你在开发过程中所以 debug 属性为

true。当 debug 为 true 时，storm 打印结点间交换的所有消息， debug 数据对于理解 topology 怎样运行是有用的。

正如前边提到的，你将使用 LocalCluster 来运行 topology。在生产环境中，topology 持续的运行，但对于这个例子你可以只运行 topology 几秒以便你可以查看结果。

```
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("Getting-Started-Toplogie", conf, builder.createTopology());
Thread.sleep(2000);
cluster.shutdown();
```

创建和运行 topology 使用 createTopology 和 submitTopology，睡眠两秒(topology 运行在不同的线程中)，然后通过关闭集群来停止 topology。

看示例 2-3 来把它放到一起。

Example 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 {
        //Topology definition
        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"));
        //Configuration
        Config conf = new Config();
        conf.put("wordsFile", args[0]);
        conf.setDebug(false);
        //Topology run
        conf.put(Config.TOPOLOGY_MAX_SPOUT_PENDING, 1);
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("Getting-Started-Toplogie", conf,
        builder.createTopology());
        Thread.sleep(1000);
        cluster.shutdown();
    }
}
```

实践一下

你将要运行你的第一个topology！如果你已经建立了文件src/main/resources/words.txt 并且其中每行有一个单词，你可以用这个命令运行 topology：

```
mvn exec:java -Dexec.mainClass="TopologyMain" -Dexec.args="src/main/resources/words.txt"
```

例如，你可以用下边的 word.txt 文件：

```
Storm
test
are
great
is
an
Storm
simple
application
but
very
powerful
really
Storm
is
great
```

在日志中，你应该看到类似如下的输出：

```
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
```

在这个示例中，你只使用了每个结点的一个单例。但如果你有一个非常大的日志文件呢？你可以轻易的改变系统中结点的数量来并行工作。在这个例子中，你可以建立两个 WordCounter 的实例：

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

如果你重新运行程序，你将看到：

```
-- Word Counter [word-counter-2] --
```

application: 1
is: 1
great: 1
are: 1
powerful: 1
Storm: 3
-- Word Counter [word-counter-3] --
really: 1
is: 1
but: 1
great: 1
test: 1
simple: 1
an: 1
very: 1

牛逼啊！改变并行度这么容易(在实际中，当然，每个实例运行在独立的机器中)。但看起来似乎有个问题：单词 is 和 great 在每个 WordCounter 实例中各被计算了一次。为什么呢？当你使用 shuffleGrouping 的时候，你告诉 storm 以随机分布的方式将每条消息发送至你的 bolt 实例。在这个例子中，总是把相同的单词送到相同的 WordCounter 是更理想的。为了实现这个，你可以将 shuffleGrouping("wordnormalizer")换成 fieldsGrouping("word-normalizer",new Fields("word"))。尝试一下并且重新运行程序来验证结果。你会在后续的章节看到更多关于分组和消息流的内容。

结论

我们已经讨论了 storm 本地操作模式和远程操作模式的不同，以及用 storm 开发的强大和简便。你也学到了更多关于 storm 的基本概念，这些概念我们将在接下来的章节深入解释。

第三章 Topologies

在本章中，你将看到怎样在一个 storm topology 的不同组件之间传递元组，以及怎样在一个运行的 storm 集群上部署 topology。

流分组

在设计一个 topology 的时候，你需要做的最重要的事情是定义数据在组件之间怎样交换(流怎样被 bolts 消费)。流分组指定了每个 bolt 消费哪些流和这些流被怎样消费。

一个结点可以发射不止一条数据流。流分组允许我们选择接收哪些流。

正如我们在第二章看到的，当 topology 被定义的时候流分组就被设置好了：

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

在上述代码块中，在 topology builder 上设置了一个 bolt，然后源被设置为 shuffle 分组。流分组通常使用源组件的 ID 作为参数，并且也会选择性的使用其他参数，取决于流分组的种类。

每个 InputDeclare 可以有不止一个源，同时每个源可以用不同的流分组来分组。

Shuffle 分组

Shuffle 分组是最常用的分组方式。它使用一个参数(源组件)，源组件会发射元组到一个随机选择的 bolt 并确保每个消费者会收到等数量的元组。
Shuffle 分组对于做原子操作例如数学操作是很有用的。然而，如果操作不能被随机分布，就像第二章中的你需要计数单词的示例，你应用考虑使用其他的分组。

Fields 分组

Fields 分组允许你基于元组的一个或多个域来控制元组怎样被发送到 bolts。它确保一个联合域中给定的值集合总是会被送到相同的 bolt。回到单词计数的示例，如果你通过单词域分组流，word-normalizer bolt 总是会将元组和给定的单词一起发送到相同的 word-counter bolt 实例中。

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

fields 分组中的所有的域必须在源组件中被声明。

All 分组

All 分组发送每个元组的一份单独拷贝到接收 bolt 的所有实例上。这种分组被用来向 bolts 发送信号。例如，如果你需要刷新缓存，你可以发送一个刷新缓存信号到所有的 bolts。在单词计数的示例中，你可以通过使用 all 分组来增加清空计数器缓存的功能(见 Topologies 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) {
            //Do nothing
        }
        ...
    }
```

我们增加了一个条件判断来检查流的源。Storm 给了我们声明命名的流的可能性(如果你不发送元组到一个命名的流，则流的名字是”default”)；它是一个非常好的方式来确定元组的源，正如这个例子中我们需要确定信号一样。

在 topology 定义中，你会增加另一个流到单词计数 bolt 来将元组从 signals-spout 流发送到这个 bolt 的所有实例。

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

Signals-spout 的实现可以在 git 库中找到。

自定义分组

你可以通过实现 backtype.storm.grouping.CustomStreamGrouping 接口来实现你的自定义流分组。这给了你决定每个元组将被哪个(些)bolt 收到的权力。

我们修改单词计数示例来对元组进行分组，这样的话相同字母开头的单词将被相同的 bolt 接收。

```
public class ModuleGrouping implements CustomStreamGrouping, Serializable{
    int numTasks = 0;
    @Override
    public List<Integer> chooseTasks(List<Object> values) {
        List<Integer> boltIds = new ArrayList();
        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();
        }
    }
```

你可以看到一个 CustomStreamGrouping 的简单实现，在这里我们使用任务的数量来对单词的第一个字符的整型值取模，由此来决定哪个 bolt 将接收这个元组。

要使用示例中分分组，按照下列方式修改 word-normalizer 分组。

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

Direct 分组

这是一个由源决定哪个组件将接收元组的分组。与前一个示例类似，源将基于单词的第一个字母决定哪个 bolt 接收元组。为使用 direct 分组，在 WordNormalizer 中，使用 emitDirect 方法代替 emit 方法。

```
public void execute(Tuple input) {
    ...
    for(String word : words){
        if(!word.isEmpty()){
            ...
            collector.emitDirect(getWordCountIndex(word),new Values(word));
        }
    }
    // Acknowledge the tuple
    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 方法中算出目标任务的数量：

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

在 topology 的定义中，指定流将被直接分组：

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

Global 分组

Global 分组将所有源实例产生的元组发送到一个单独的目标实例(特别地，ID 最低的任务)中。

None 分组

在写这本著作的时候(storm 版本 0.7.1)，使用这种分组与使用 22 页的”Shuffle 分组”是一样的。换言之，当用这个分组时，流怎样分组是无所谓的。

LocalCluster versus StormSubmitter

到目前为止，你使用了叫做 LocalCluster 的设施来在你本地的电脑中运行 topology。在你的电脑里运行 storm 的基础设施使你可以方便的运行和调试不同的 topologies。但是当你想提交你的 topology 到一个正在运行的 storm 集群时怎么样呢？storm 的一个有趣的特点是提交你的 topology 到一个实际的集群中运行是很容易的。你需要将 LocalCluster 修改为 StormCommitter 并且执行 submitTopology 方法，这个方法负责将 topology 提交到集群。
你可以看到下边代码的改变：

```
//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();
```


当你使用 StormSubmitter 时，你不能像你使用 LocalCluster 那样通过代码控制集群。

接下来，把资源打包成一个 jar，它将在你运行 storm 客户端命令提交 topology 时被发送。因为你使用 maven，你唯一需要做的一件事是到源文件目录下，然后运行下边的命令：

```
mvn package
```

一旦你有了创建好的 jar，使用 storm jar 命令来提交 topology(在附录 A 中你会知道怎样安装 storm 客户端)。语法是 storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3。
这个示例中，在 topologies 的源工程文件夹运行：

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

用这些命令，你可以提交 topology 到集群。
要停止/杀掉它，运行：

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


topology 的名字必须是唯一的。


要安装 storm 客户端，见附录 A。

DPRC Topologies

有一种特殊类型的 topology 被称为是分布式远程过程调用(DRPC)，它通过 storm 分布式的能力来执行远程过程调用(RPC)(见图 3-1)。Storm 提供了一些工具来激活 DRPC 的使用。第一个是一个在客户端和 storm topology 之间作为连接器运行的 DRPC 服务器，作为 topology spouts 的源。它接收一个函数和它的参数来执行。然后对于函数操作的每个数据片，该服务器指定一个在 topology 中使用的请求 ID 来识别 RPC 请求。当 topology 执行最后一个 bolt 时，它必须发射 RPC 的请求 ID 和结果，这使得 DRPC 服务器可以返回结果至正确的客户端。


一个单独的 DRPC 服务器可以执行很多函数。每个函数可以由一个唯一的标识符识别。

Storm提供的第二个工具是LinearDRPCTopologyBuilder(在示例中使用的)，一个来帮助构建DRPC topologies的抽象。构建的topology创建DRPCSpouts---它连接DRPC服务器并且发送数据到topology的剩余部分---topology还包装bolts，这使得结果可以从最后一个bolt返回。所有添加到LinearDRPCTopologyBuilder上的bolts被顺序执行。
作为这种类型topology的一个示例，你可以创建一个累加数的处理过程。这只是一个简单的示例，但是可以继承这个概念来执行复杂的分布式数学操作。

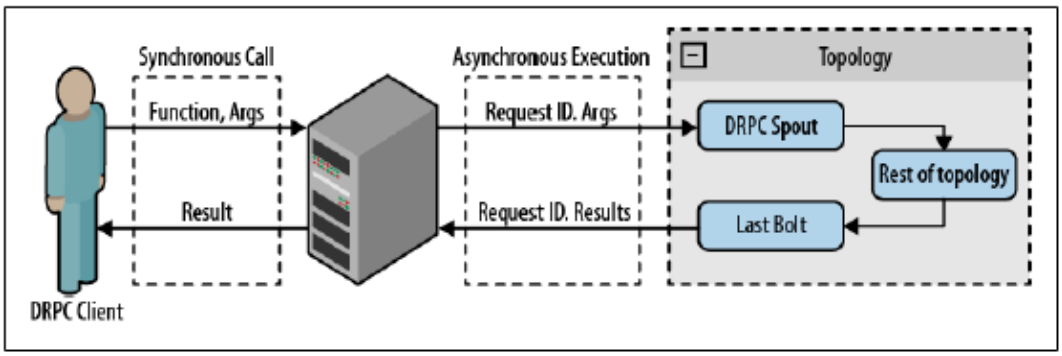

图3-1 DRPC topology模式

Bolt有如下的输出声明：

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

因为这是topology中唯一的bolt，它必须发射RPC ID和结果。

execute方法负责执行增加的操作：

```
    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));
    }
```

包含做累加的bolt的topology的定义如下：

```
    public static void main(String[] args) {
        LocalDRPC drpc = new LocalDRPC();
        LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("add");
        builder.addBolt(new AdderBolt(),2);
        Config conf = new Config();
        conf.setDebug(true);
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("drpc-adder-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LocalDRPC对象来在本地运行DRPC。接着，创建topology builder来添加bolt到topology。为了测试topology，在你的DRPC对象上使用execute方法。

为连接一个远程的DRPC服务器，需要使用DRPCClient类。DRPC服务器暴露一个可以被多种语言使用的Thrift API，并且在本地或远程运行DRPC服务器的API是一样的。为了提交topology到storm集群，使用builder对象的createRemoteTopology方法来代替createLocalTopology方法，该方法使用storm配置中的DRPC配置。

第四章 Spouts

在本章中，你会看到设计 topology 入口点(spout)的最常用的策略和怎样使得 spouts 容错。

可靠消息 versus 不可靠消息

当设计 topology 的时候，一件要时刻记在脑中的重要事情是消息的可靠性。如果一个消息不能被处理，你需要决定对这条单独的消息该做什么，对于整个 topology 该做什么。例如，当处理银行存款时，不遗漏一条交易消息是重要的。但是如果你处理上百条的 tweet 消息来寻找一些统计指标的话，丢失了一条数据，你可以假设指标是非常准确的。

在 storm 中，根据每个 topology 的需求来确保消息的可靠性是开发者的职责。这其中包含一个权衡问题。一个可靠的 topology 必须处理丢失的消息，而这需要更多的资源。一个不是那么可靠的 topology 可能丢失一些消息，但是资源没那么密集。不论选择的可靠性策略是什么，storm 都提供工具来实现它。

为了管理 spout 的可靠性，你可以发送时和元组一起发送一个消息 ID(collector.emit(new Values(…),tupleId))。ack 方法和 fail 方法在元组被成功处理和处理失败时分别被调用。只有当元组被所有的目标 bolts 和锚定的 bolts 处理完时，元组才被处理成功(在第五章中会了解到怎么为一个元组锚定一个 bolt)。

元组处理失败当：

. collector.fail(tuple)被目标 spout 调用

. 处理时间超出配置的超时时间

我们看一个示例。假设你在处理银行的交易，你有如下的需求：

. 如果一个交易失败，重新发送消息。

. 如果一个交易失败太多次，终止 topology。

你将创建一个发送 100 个随机交易 ID 的 spout 和一个接受到的元组 80%都处理失败的 bolt(你可以在 ch04-spout 示例中找到完成的示例)。你将用 Map 实现的 spout 来发射交易消息元组，这样的话重发消息是很容易的。

```
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();
    }
}
```

如果仍有消息等待被发送，那么获取每个交易消息及它相关联的 ID 并且将它们作为一个元组发射，然后清空消息队列。调用 map 的 clear 是安全的，因为 nextTuple，fail 和 ack 是唯一修改 map 的方法并且它们在相同的线程中运行。

维护两个 map 来记录等待发送的交易消息和每个交易已经失败的次数。ack 方法仅仅从每个列表中删除了交易消息。

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

fail 方法决定是否重发交易消息或者当失败次数太多时终止 topology。

当你在 topology 中使用 all 分组并且任何 bolt 的实例失败的时候，spout 的 fail 方法也会被调用。

```
public void fail(Object msgId) {
    Integer transactionId = (Integer) msgId;
    // Check the number of times the transaction has failed
    Integer failures = transactionFailureCount.get(transactionId) + 1;
    if(fails >= MAX_FAILS){
        // If the number of failures is too high, terminate the topology
        throw new RuntimeException("Error, transaction id ["+
            transactionId+"] has had too many errors ["+failures+"]");
    }
    // If the number of failures is less than the maximum, save the number and re-send
    the message
    transactionFailureCount.put(transactionId, failures);
    toSend.put(transactionId,messages.get(transactionId));
    LOG.info("Re-sending message ["+msgId+"]");
}
```

首先，检查交易失败的次数。如果一个交易失败太多次，抛出一个 RuntimeException 来终止它所在的 worker。否则，保存失败的次数并且将交易消息发到 toSend 队列，这样当 nextTuple 方法被调用时它会被重新发送。

storm 结点不保存状态，所以如果你在内存中保存信息(就像在这个示例中)并且结点失败的话，你将丢失所有已保存的信息。

Storm 是一个快速失败的系统。如果一个异常被抛出，topology 会失败，但是 storm 将以一致的状态重新启动处理，这样它可以正确的恢复。

获取数据

这里你将看到一些常用的设计 spouts 的方法，它们可以从多个源有效的收集数据。

直接连接

在直接连接的架构中，spout 直接连接到一个消息发射器(见图 4-1)。

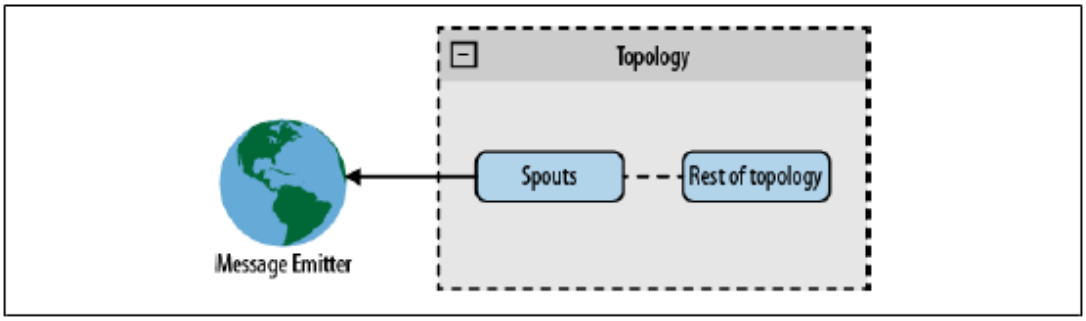

图 4-1 直接连接的 spout

这种架构实现简单，特别是当消息发射器是一个众所周知的设备或设备组时。一个众所周知的设备是指一个在启动时就被知晓并且在整个 topology 生命周期中都保持一致的设备。一个未知的设备是一个 topology 已经运行后添加的设备。一个众所周知的设备组是一个组中所有设备在启动时就被知晓的设备组。

作为一个示例，创建一个 spout 来使用 Twitter Streaming API 读取 Twitter 流。Spout 将直接连接到作为消息发射器的 API。通过过滤流来获取所有公共的与 track 参数(就像 Twitter 开发页面上的文档)相匹配的 tweets。完整的示例可以在 Twitter 示例的 github 页面上找到。

Spout 从配置对象中获取连接参数(track，user，and password)并且创建一个到 API 的连接(在这个示例中，使用 Apache 的 DefaultHttpClient)。它从连接中每次读取行，将该行从 JSON 格式解析为 java 对象，然后发射它。

```
public void nextTuple() {
    //Create the client call
    client = new DefaultHttpClient();
    client.setCredentialsProvider(credentialProvider);
    HttpGet get = new HttpGet(STREAMING_API_URL+track);
    HttpResponse response;
    try {
        //Execute
        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;
            //Read line by line
            while((in = reader.readLine())!=null){
                try{
                    //Parse and emit
                    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) {
            }
        }
    }
```

这里你正在锁定 nextTuple 方法，所以你从未执行 ack 和 fail 方法。在实际的应用中，我们推荐你在一个单独的线程中加锁并且使用内部的队列来交换信息(你可以在 34 页的下一个示例”Enqueued Messages”中学到怎样做)。

这很好！

你使用一个单独的 spout 读取 Twitter。如果你并行你的 topology，你将有多个 spouts 来读取同一个流的不同部分，这没有意义。所以当你有多个流要读时你怎样并行你的处理呢？storm 的一个有趣的特点是你可以从任何组件(spouts/bolts)访问 TopologyContext。通过这个特性，你可以在你的 spout 实例之间划分流。

```
    public void open(Map conf, TopologyContext context,
            SpoutOutputCollector collector) {
        //Get the spout size from the context
        int spoutsSize =
        context.getComponentTasks(context.getThisComponentId()).size();
        //Get the id of this spout
        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("No track found for spout" +
                    " [spoutsSize:"+spoutsSize+", tracks:"+tracks.length+"] the amount" +
                    " of tracks must be more then the spout paralellism");
            this.track =tracksBuffer.substring(1).toString();
        }
```

```
        ...
    }
```

通过这种技术，你甚至可以在数据源间分布收集器。相同的技术可以被应用在其他的场景-例如，从 web 服务器收集日志文件。见图 4-2。

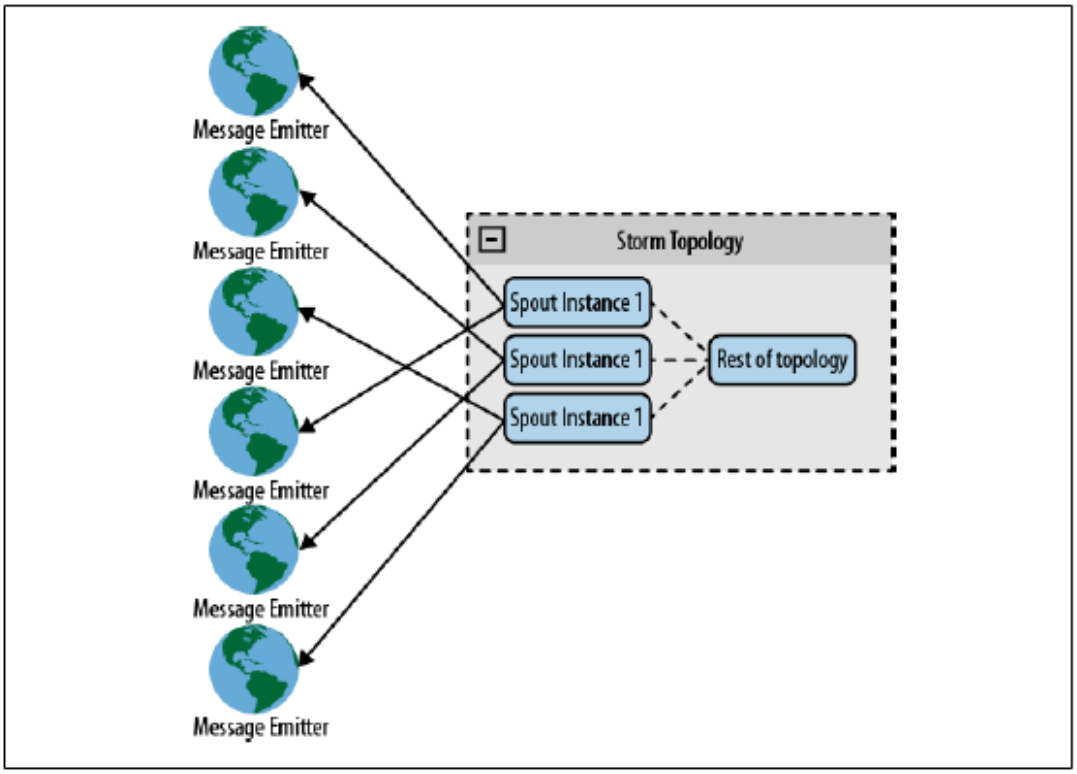

图 4-2 直接哈希连接

在前边的例子中，你连接 spout 到一个众所周知的设备。你可以使用相同的方法来连接到一个未知的设备并听过一个协同系统来维护设备列表。该协调器检测列表的变化并建立和销毁连接。例如，当从 web 服务器收集日志文件时，web 服务器的列表可能随着时间变化。当一个 web 服务器被添加时，协调器检测到变化并为它建立一个新的 spout。见图 4-3。

推荐建立从 spout 到消息发射器的连接，而不是相反的方式。如果运行 spout 的集群挂掉了，storm 会在别的机器上重新启动该 spout，所以从 spout 来定位消息发射器比消息发射器保存 spout 所在的机器的路径简单。

队列消息

第二个方法是连接你的 spouts 到一个排队系统，该系统将从消息发射器接收消息并将消息供 spouts 消费。使用排队系统的优点是它可以作为 spouts 和数据源之间的中间件。很多时候你可以通过排队来做多队列的消息回放。这意味着你不需要知道任何关于消息发射器的

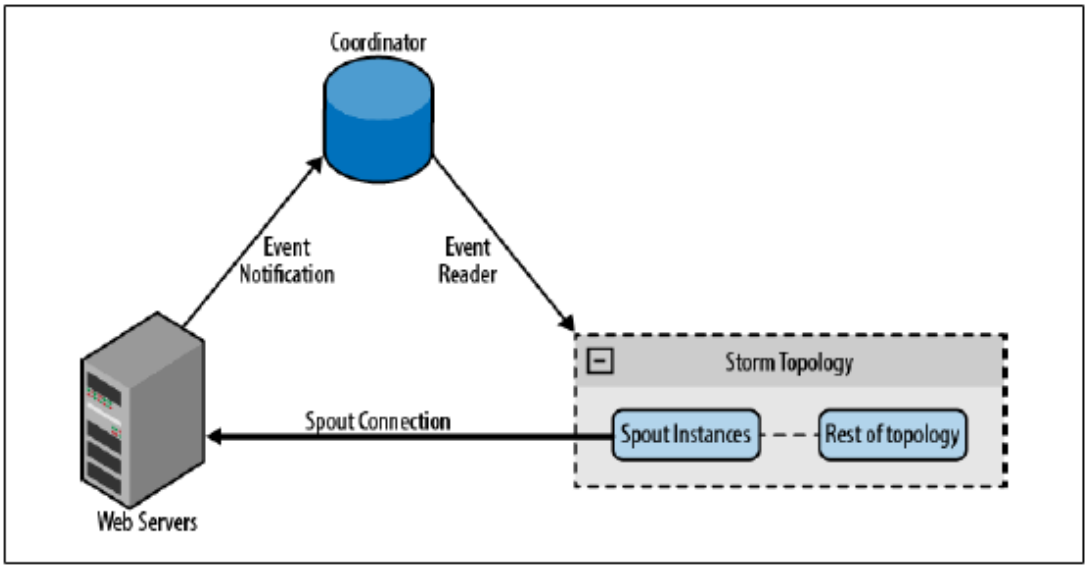

图 4-3 直接连接协调器

东西，添加和删除发射器比直接连接更简单。这种架构的问题是队列将成为你的失败点，并且你将你的处理流增加了一层。

图 4-4 显示了架构的模式。

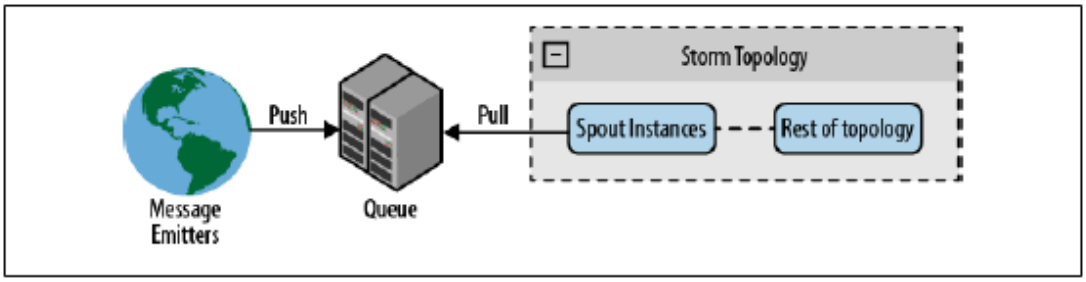

图 4-4 使用排队系统

你可以使用循环拉或者哈希队列(用哈希来划分消息队列并发送到 spout 或者建立多个队列)来在队列间并行的处理，在多个 spouts 间划分消息。

你将建立一个用 Redis 及它们的 Java 库 Jedis 做排队系统的示例。在这个例子中，你将建立一个日志处理器从一个未知的源收集日志，它使用 lpush 插入消息到队列中，使用 blpop 来允许你等待一个消息。如果你有多个处理，使用 blpop 将使你以循环的方式接收消息。

为从 Redis 中接收消息，你将使用一个在 open spout 中创建的线程(使用一个线程来避免锁定 nextTuple 方法所在的主循环)。

```
new Thread(new Runnable() {
    @Override
    public void run() {
        while(true){
            try{
```

```
                    Jedis client = new Jedis(redisHost, redisPort);
                    List<String> res = client.blpop(Integer.MAX_VALUE, queues);
                    messages.offer(res.get(1));
                }catch(Exception e){
                    LOG.error("Error reading queues from redis",e);
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e1) {}
                }
            }
        }
    }).start()
```

这个线程的唯一目的是建立连接并且执行 blpop 命令。当消息被接收到时，它被添加到一个内部的会被 nextTuple 方法消费的消息队列。这里你可以看到 Redis 队列是源并且你不知道谁是消息的发射者以及它们的数量。

我们推荐不要在 spout 中创建许多线程，因为每个 spout 运行在一个不同的线程中。与其创建很多线程，更好的方法是增加并行度。这将以分布式的方式在 storm 集群中创建更多的线程。

在 nextTuple 方法中，你唯一要做的是接受消息并再次发射它们。

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

你可以通过修改 spout 使其通过 Redis 具有重发消息的能力来使整个 topology 变得更加可靠。

DRPC

DPRCSpout 是实现了接收来自 DRPC 服务器的函数调用流并且处理它的 spout(见第三章中的示例)。在最常用的情况下，backtype.storm.drpc.DRPCSpout 是足够的，但有可能使用来自 storm 包的 DRPC 类来创建你自己的实现。

结论

你已经看到了最常用的 spout 实现的模型，它们的优点，以及怎样使得消息可靠。基于你致力解决的问题来设计 spout 通信是重要的。没有一种适合所有 topologues 的架构。如果你了

解源并且可以控制这些源，你可以使用直接连接，如果你需要添加未知源的能力或者从多个源接收消息的能力，使用列队连接是更好的。如果你需要在线处理，你将需要使用DRPCSpouts 或者类似的实现。
尽管你已经了解了三种主要类型的连接，仍有无限的实现它的方式，这取决于你的需求。

第五章 Bolts

正如你看到的，bolts 是一个 storm 集群中的关键组件。在本章中，你将看到一个 bolt 的生命周期，bolt 的设计策略，以及怎样实现它们的一些示例。

Bolt 生命周期

Bolt 是一种将元组作为输入并且制造元组作为输出的组件。当你实现一个 bolt 的时候，你通常实现 IRichBolt 接口。Bolts 在客户端机器被创建，序列化至 topology 中，并提交至集群中的 master 机器。集群运行 workers 来反序列 bolt，调用它上的 prepare 方法，然后开始处理元组。


要自定义 bolt，你需要在它的构造方法中设置参数并且将它们保存为实例变量，这样它们可以在 bolt 被提交至集群时序列化。

Bolt 结构

Bolts 包含如下方法：
declareOutputFields(OutputFieldsDeclarer declarer)
　　定义该bolt的输出模式。
prepare(java.util.Map stormConf, TopologyContext context, OutputCollector col
lector)
　　在bolt开始处理元组前被调用。
execute(Tuple input)
　　处理一个单独的输入元组。
cleanup()
　　当 bolt 将要关闭时被调用。
来看一个将句子分隔成单词的 bolt 示例：

```
class SplitSentence implements 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(new Values(word));
            }
        }
        public void cleanup() {
        }
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
}
```

正如你看到的，该bolt是非常直观的。值得一提的是这个例子中没有消息担保。这意味着如果该bolt由于某种原因丢弃了一条信息---因为该消息失败了或者以编程的方式故意丢弃了---产生该消息的spout是永远不会被提醒的，并且其间的任何的bolts和spouts也不会。
在许多场景下，你是希望在整个topology中消息是被确保处理的。

可靠 Bolts versus 不可靠 Bolts

如前述，storm 确保由 spout 发送的每条消息都会被所有 bolts 完全处理。这是一种设计考虑，意味着你需要决定是否你的 bolts 是确保消息被处理的。
Topology 是一颗消息(元组)经过一条或多条分支形成的的结点树。每个结点将 ack(元组)或 fail(元组)，这样 storm 知道一个消息是什么时候失败的并且通知产生该消息的 spout 或 spouts。因为 storm topology 运行在一个高度并行的环境，因此记录原始的 spout 实例的最好方式是在消息元组中包含一个原始 spout 的引用。这个技术叫做锚定。修改你刚刚看到的 SplitSentence bolt，这样它可以确保消息处理。

```
class SplitSentence implements 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) {
            declarer.declare(new Fields("word"));
        }
}
```

锚定发生的具体行是在 collector.emit() 语句。就像前边提到的，传递元组使得 storm 可以记录原始 spout 的踪迹。collector.ack(tuple) 和 collector.fail(tuple)告诉 spout 每条消息发生了什

么。当树中每条消息都被处理后，storm 认为由 spout 发出的元组被完全处理了。当它的消息树没有在配置的超时时间内被全部处理时，一个元组的处理被认为是失败的。缺省时间为 30 秒。


你可以通过修改topology的Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS配置来修改超时。

当然，spout需要考虑消息失败并且重试或者相应的丢弃消息的场景。


每个你处理的元组都必须被应答或者宣告失败。Storm使用内存来保存每个元组的踪迹，所以如果你没有确认/宣告失败每个元组，该任务会逐渐的耗尽内存。

多流

Bolt 可以使用 emit(streamId, tuple)发射元组到多条流，每条流由字符串 streamId 来识别。然后，在 TopologyBuilder 中，你可以决定订阅哪条流。

多锚定

使用 bolt 来连接或者聚合流时，你需要在内存中缓存元组。为了这种情形下的消息担保，你不得不锚定流到不止一个元组。这通过调用包含元组列表的 emit 方法实现。

```
...
List<Tuple> anchors = new ArrayList<Tuple>();
anchors.add(tuple1);
anchors.add(tuple2);
_collector.emit(anchors, values);
...
```

那样，任何时候一个 bolt 被确认或者失败，它会通知树，并且由于流被锚定给不止一个元组，被包含的所有 spouts 都会被通知。

通过 IBasicBolt 来自动 Ack

正如你可能注意到的，在许多用例中，你需要消息担保。为简化它，storm 为 bolt 提供了另一个叫做 IBasicBolt 的接口，它封装了一种在 execute 方法之后调用 ack 方法的模式。BaseBasicBolt，一个该接口的实现，被用来做自动确认。

```
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"));
        }
}
```

被发射到 BasicOutputCollector 的元组被自动锚定到输入元组。

第六章 一个真实的示例

本章要阐述一个典型的 web 分析解决方案，一个经常使用 hadoop 批量作业来解决的问题。不同于 hadoop 的实现，基于 storm 的解决方案将显示实时刷新的结果。

我们的示例包含三个主要的组件(见图 6-1)：

- 一个 Node.js 网络应用，用来测试系统
- 一个 Redis 服务器，用来持久化数据。
- 一个 storm topology，用于实时分布式数据处理。

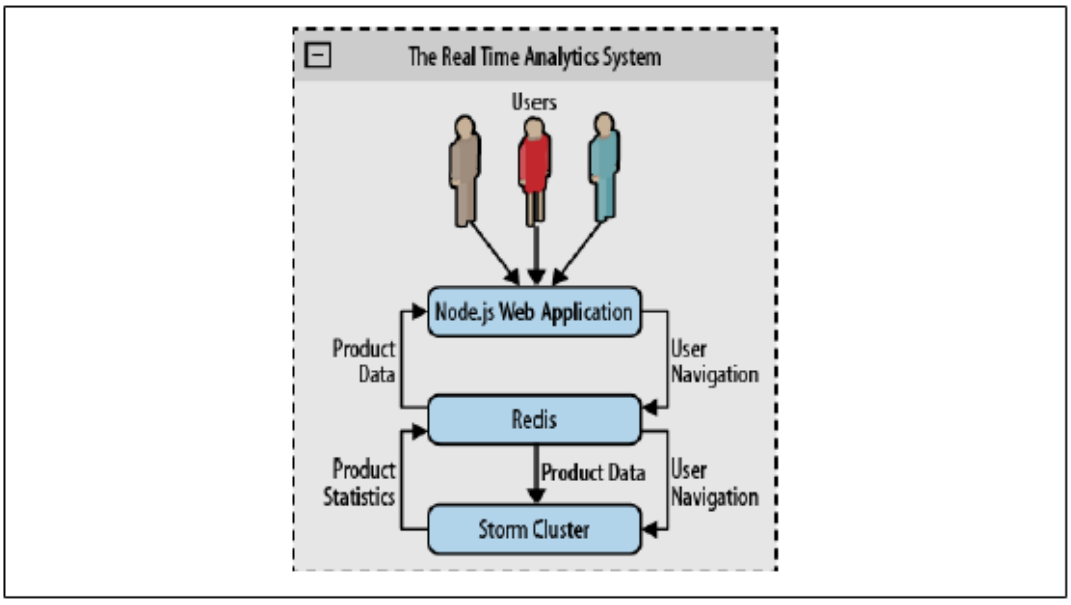

图 6-1. 架构总述

如果你希望通过实现示例来学习这一章，你应该先看看附录 C。

Node.js Web 应用

我们使用三个页面来模拟了一个电子商务网站：一个主页，一个产品页以及一个产品统计页。这个应用通过使用 Express Framework 和 Socket.io Framework 向浏览器推送更新来实现。该应用的目的是使你运行集群并看到结果，但它并不是本书的关注点，因此我们除了页面本身的描述没有其他任何的细节深入。

主页

本页提供平台所有可用商品的链接来简化他们的导航。它从 Redis 服务器读取并列举了所有的项。本页的 URL 地址是 http://localhost:3000/。(见图 6-2)。

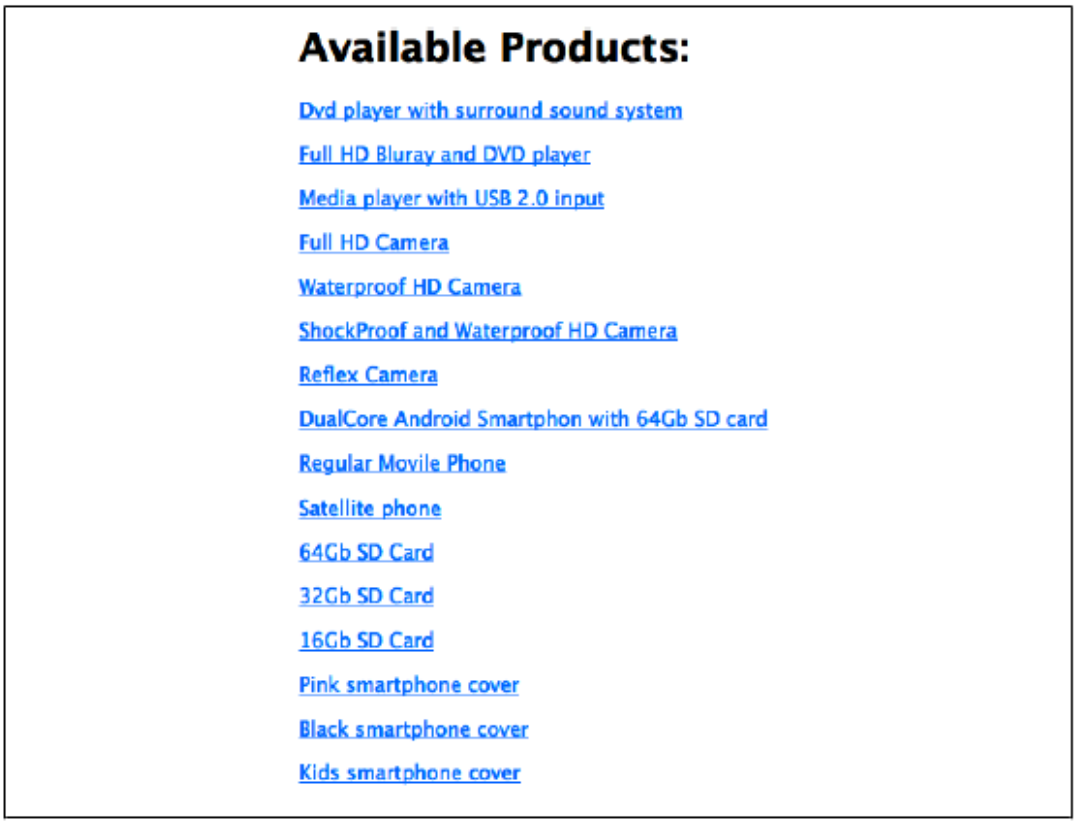

图 6-2. 主页

产品页

产品页显示了一个与特定产品相关的信息，例如价格，标题及分类。本页的 URL 地址是 http://localhost:3000/product/:id。(见 6-3)。

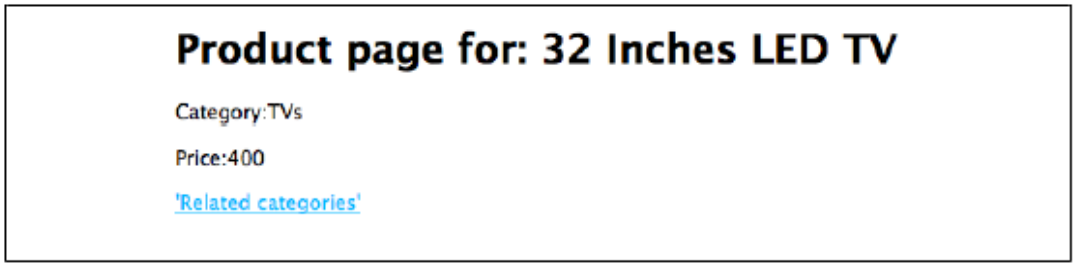

图 6-3. 产品页

产品统计页

本页展示 storm 集群计算的信息，这些信息是用户浏览页面时被收集的。可以如此总结：

浏览了这件产品的用户同时观察了 N 次该类别的产品。本页的 URL 地址是 http://localhost:3000/product/:id/stats。(见图 6-4)。

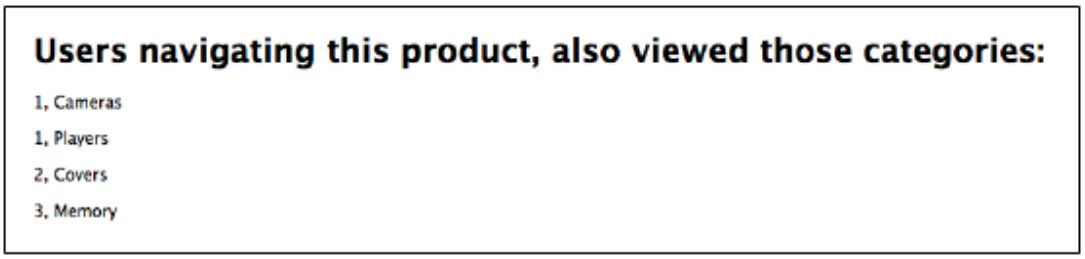

图 6-4. 产品状态视图

开始 Node.js Web 应用

在开始 Redis 服务器之前，通过运行工程路径上的如下命令来启动 web 应用：

```
Node webapp/app.js
```

该 web 应用会自动植入包含示例产品的 Redis 供你实践。

The Storm Topology

Storm topology 在本系统中的目的是当用户浏览页面时实时的更新产品统计数据。产品统计页显示一个关联了计数器的分类列表，显示访问了同一类目下其他产品的用户个数。这有助于卖家了解他们客户的需求。Topology 收到一条浏览日志并且更新产品的统计数据，正如在图 6-5 中显示的。

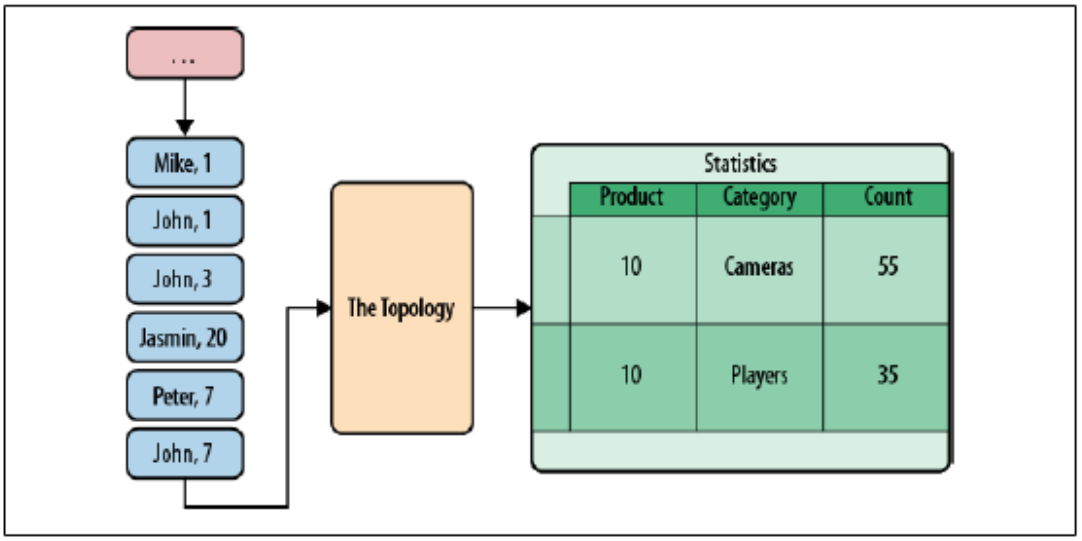

图 6-5. Storm topology 输入和输出

我们的 storm topology 包含五个组件：一个用于供给它的 spout 及四个用于完成工作的 bolts。

UsersNavigationSpout

读取用户浏览队列并发送至 topology

GetCategoryBolt

从 Redis 服务器读取产品信息并添加其类目到流

UserHistoryBolt

读取用户之前的产品浏览信息并发送 Product:Category 对来更新下一阶段的计数器

ProductCategoriesCounterBolt

记录用户浏览的某一特定分类的某产品的次数。

NewsNotifierBolt

通知 web 应用立即更新用户界面。

这里是 topology 的构建(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());
    }
}
```

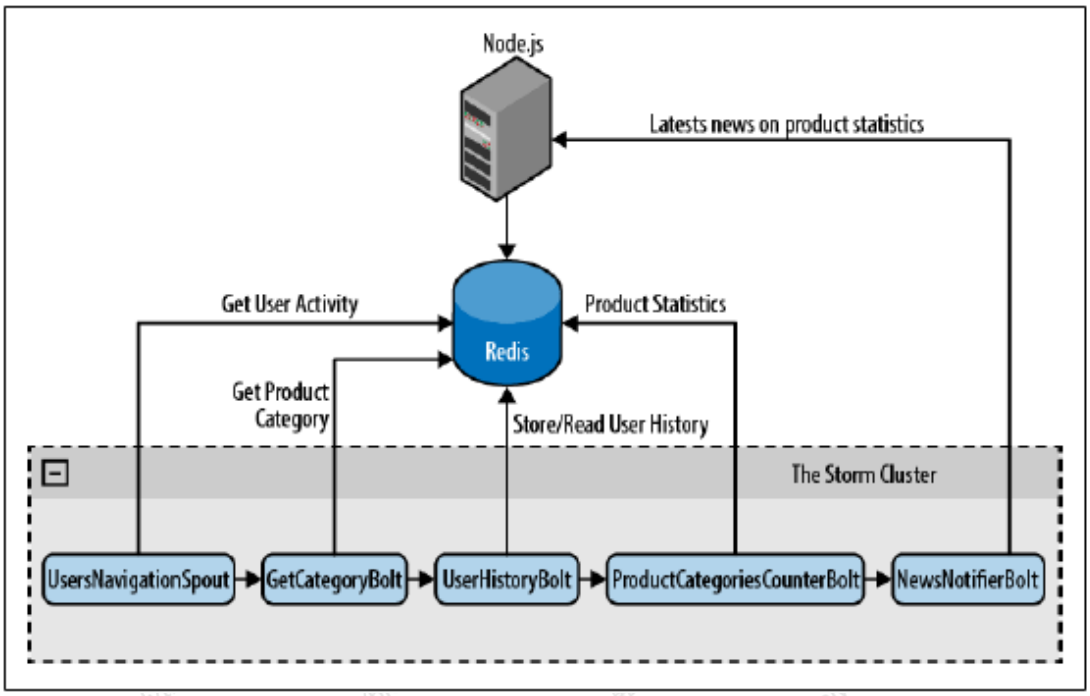

图 6-6. Storm topology

UsersNavigationSpout

UsersNavigationSpout 负责向 topology 输送浏览记录。每条浏览记录是一条某用户浏览某产品页的引用。它们通过 web 应用被存储在 Redis 服务器中。我们一会儿将深入它的更多细节。为了从 Redis 服务器读取记录，你需要使用 https://github.com/xetorthio/jedis，一个极小极简的 Redis Java 客户端。

接下来的代码块只显示相关部分的代码。

```
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"));
        }
}
```

首先 spout 调用 jedis.rpop("navigation")来删除并返回 Redis 服务器"浏览"列表中最右端的元素。如果列表已空，睡眠 0.3 秒以此来避免忙等循环将服务器阻塞。如果找到一条记录，则解析内容(内容是 JSON)并将它映射到一个 NavigationEntry 对象，该对象仅仅是一个包含记录信息的 POJO：

. 浏览的用户
. 用户浏览的页面类型
. 页面附加信息取决于类型属性。"产品"类型页面包含一个被浏览的产品 ID 项。

Spout 通过调用 collector.emit(new Values(user,entry))发射包含该信息的元组。元组的内容是 topology 中下一个 bolt 的输入：GetCategoryBolt。

GetCategoryBolt

这是一个非常简单的 bolt。它唯一的职责是反序列化由前边 spout 发送的元组的内容。如果消息记录是关于产品页，它会使用 ProductsReader 帮助类来从 Redis 服务器加载产品信息。然后，对于每个输入的元组，它发送一个新的包含更进一步特定产品信息的元组：

. 用户
. 产品
. 产品的分类

```
package storm.analytics;
public class GetCategoryBolt extends BaseBasicBolt {
    private ProductsReader 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");
                // Call the items API to get item information
                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();
                }
            }
        }
        ...
}
```

正如前边所提到的，使用 ProductReader 帮助类来读取特定的产品信息。

```
package storm.analytics.utilities;
...
public class ProductsReader {
        ...
        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);
            JSONObject product=(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 是应用的核心。它负责保存每个用户浏览的产品的踪迹并决定哪些是应该被增加的结果对。

你将使用 Redis 服务器来按用户存储产品的历史记录，出于性能原因，你需要在本地维护一个存储的拷贝。你通过 getUserNavigationHistory(user) 和 addProductToHistory(user,prodKey) 方法分别来读写，以此隐藏了数据的访问细节。

```
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);
            // Product key will have category information embedded.
```

```
            String prodKey = prod1+":"+cat1;
            Set<String> productsNavigated = getUserNavigationHistory(user);
            // If the user previously navigated this item -> ignore it
            if(!productsNavigated.contains(prodKey)) {
                // Otherwise update related items
                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);
            }
        }
}
```

注意该 bolt 的期望输出是发射分类关系应该被增加的产品。

看一下源代码。Bolt 保存了每个用户浏览的产品的集合。注意该集合包含的是'产品：分类'对而不仅仅是产品。那是因为在以后的调用中你需要分类信息并且如果这些信息不需要每次都从数据库读取的话效率会更高。这是可能的，因为产品只属于一个分类并且分类在产品的生命周期中不会改变。

在读取用户之前浏览的产品集合后(包含它们的分类)，检查是否当前产品之前已经被访问过。如果是，本条记录被忽略。如果这是用户第一次浏览该产品，遍历用户的历史浏览记录并使用collector.emit(new Values(prod1, cat2))方法发送一个包含当前正在被浏览的产品及历史浏览记录中所有产品的分类信息的元组，使用collector.emit(new Values(prod2, cat1))发送包含其他产品及正在被浏览的产品所属分类的另一个元组。最后，添加产品及它的分类到集合。

例如，假设用户John有如下的浏览记录:

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

以及下边的需要被处理的浏览记录：

User	#	Category
John	8	Phones

用户还未浏览 8 号产品，所以你需要处理它。

所以发射的元组将是：

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

注意左边产品与右边分类的关系应该在同一个单元中被增加。
现在，我们探索下 bolt 使用的持久化。

```
public class UserHistoryBolt extends BaseRichBolt{
    ...
    private Set<String> getUserNavigationHistory(String user) {
        Set<String> userHistory = usersNavigatedItems.get(user);
        if(userHistory == null) {
            userHistory = jedis.smembers(buildKey(user));
            if(userHistory == null)
                userHistory = new HashSet<String>();
            usersNavigatedItems.put(user, userHistory);
        }
        return userHistory;
    }
    private void addProductToHistory(String user, String product) {
        Set<String> userHistory = getUserNavigationHistory(user);
        userHistory.add(product);
        jedis.sadd(buildKey(user), product);
    }
    ...
}
```

getUserNavigationHistory 方法返回用户已访问的产品的集合。首先，尝试使用 userNavigatedItems.get(user)方法从本地内存中读取用户的历史浏览记录，如果读取不到，通过 jedis.smembers(buildKey(user))方法从 Redis 服务器读取并将读取数据添加到本地内存结构 usersNavigatedItems。
用户浏览新产品时，调用 addProductToHistory 方法来同时更新内存结构与 Redis 服务器。通过 userHistory.add(peoduct)更新内存结构，通过 jedis.sadd(buildKey(user),product)更新 Redis 服务器。
值得注意的是既然 bolt 在内存中按用户保存信息，那么当你并行化 bolt 时，在第一级对用户使用 FieldGrouping 是非常重要的，否则用户历史记录的不同拷贝将不同步。

ProductCategoriesCounterBolt

ProductCategoriesCounterBolt 类负责记录所有的产品-分类关系。它接收由 UsersHistoryBolt 发射的产品-分类对并更新计数器。
每个产品-分类对的事件信息被存放在 Redis 服务器。出于性能的原因，使用一个本地的用于读的缓存和一个写缓存。事件信息被通过一个后台进程发送到 Redis。
对于输入，该 bolt 也会发送一个包含了更新的计数器的元组来供 topology 中下一个 bolt 消费，NewsNotifierBolt，它用于广播消息到最终用户来用于实时更新。

```
public class ProductCategoriesCounterBolt extends BaseRichBolt {
    ...
    @Override
    public void execute(Tuple input) {
        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;
    }
    ...
}
```

该 bolt 中的持久化被隐藏在 getProductCategoryCount 和 storeProductCategoryCount 方法中。我们进一步看下：

```
package storm.analytics;
...
public class ProductCategoriesCounterBolt extends BaseRichBolt {
    // ITEM:CATEGORY -> COUNT
    HashMap<String, Integer> counter = new HashMap<String, Integer>();
    // ITEM:CATEGORY -> COUNT
    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);
            }
        }
        ...
}
```

getProductCategoryCount 首先查询内存缓存计数器。如果查不到相应信息，它将从 Redis 服务器中获取它。

storeProductCategoryCount 更新计数器缓存和 pendingToSave 缓冲区。该缓冲区被下边的后台进程持久化：

```
package storm.analytics;
public class ProductCategoriesCounterBolt extends BaseRichBolt {
        ...
        private void startDownloaderThread() {
            TimerTask t = new TimerTask() {
            @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，当它发送旧的 buffer 到 Redis 的同时建立新的缓冲区来供其他线程使用。该代码块每隔 downloadTime 毫秒运行一次并且可以通过 down-load topology 配置参数来配置。download-time 越长，执行 Redis 写的次数越少，因为连续的添加被一次写入。
再次牢记，正如在前边的 bolt 中一样，当分配资源到该 bolt 时，应用正确的域分组是非常重要的，在该例中根据产品分组。那是因为它按产品存储该信息的内存拷贝，并且如果一些缓存和 buffer 的拷贝存在，将出现不一致。

NewsNotifierBolt

NewsNotifierBolt 负责通知 web 应用统计数据的变化，目的是为了使用户可以实时的看到变化。通知使用 Apache HttpClient 构建 HTTP POST，发送到 topology 中配置的 web 服务器的 URL 地址。POST 体被编码成 JSON。
当测试的时候，该 bolt 被从 topology 中删除。

```
package storm.analytics;
...
public class NewsNotifierBolt extends BaseRichBolt {
    ...
    @Override
    public void execute(Tuple input) {
        String product = input.getString(0);
        String categ = input.getString(1);
        int visits = input.getInteger(2);
        String content = "{ \"product\": \""+product+"\", \"categ\":\""+categ+"\",
            \"visits\":"+visits+" }";
        HttpPost post = new HttpPost(webserver);
        try {
            post.setEntity(new StringEntity(content));
            HttpResponse response = client.execute(post);
            org.apache.http.util.EntityUtils.consume(response.getEntity());
        } catch (Exception e) {
            e.printStackTrace();
            reconnect();
        }
    }
...
}
```

Redis 服务器

Redis 是一套高级的用于持久化的内存 Key Value 存储系统(见 http://redis.io)。使用它来存储下述信息：
. 产品信息，用于服务网站。
. 用户导航队列，用于供给 Storm Topology。

· Storm Topology 中间数据，用于 Topology 从失败中恢复。
· Storm Topology 结果，用于存储预期的结果。

生产信息

Redis 服务器存储产品，它使用产品 ID 作为键，包含所有产品信息的 JSON 对象作为值。

```
> redis-cli
redis 127.0.0.1:6379> get 15
"{\"title\":\"Kids smartphone cover\",\"category\":\"Covers\",\"price\":30,\"id\":
15}"
```

用户导航队列

用户导航队列被存储在一个命名为导航的 Redis 列表中并且被组织成一个先进先出(FIFO)的队列。每次用户访问一个产品页，服务器添加一个项到列表左端，以此来表明哪个用户访问了哪个商品。Storm 集群不时地从列表的右端删除元素来处理信息。

```
redis 127.0.0.1:6379> llen navigation
(integer) 5
redis 127.0.0.1:6379> lrange navigation 0 4
1) "{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"1\",\"type\":
\"PRODUCT\"}"
2) "{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"1\",\"type\":
\"PRODUCT\"}"
3) "{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"2\",\"type\":
\"PRODUCT\"}"
4) "{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"3\",\"type\":
\"PRODUCT\"}"
5) "{\"user\":\"59c34159-0ecb-4ef3-a56b-99150346f8d5\",\"product\":\"5\",\"type\":
\"PRODUCT\"}"
```

中间数据

集群需要分别存储每个用户的历史记录。为达到这个目的，它在 Redis 服务器中保存了一个集合，该集合存储了每个用户浏览的所有产品及它们的分类。

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

结果

集群产生用户访问特定产品的有用数据并且将它们存储在命名为”procnt”的 Redis Hash 中：后边紧跟着产品 ID。

```
redis 127.0.0.1:6379> hgetall prodcnt:2
1) "Players"
2) "1"
3) "Cameras"
4) "2"
```

测试 Topology

为了测试 topology，使用提供的 LocalCluster 和一个本地的 Redis 服务器(见图 6-7)。你将在初始化时填充产品数据库并且在 Redis 服务器上模拟浏览日志的插入。我们的断言将通过读取 topology 输出到 Redis 服务器来执行。测试用 Java 和 Groovy 编写。

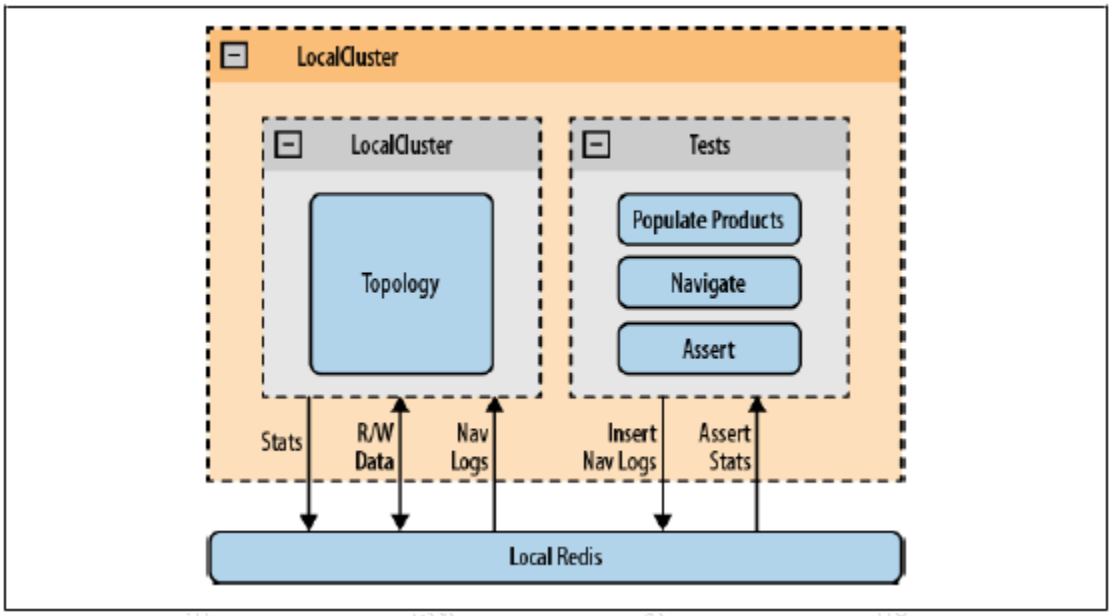

图 6-7. 测试架构

测试初始化

初始化由三步组成：

启动 LocalCluster 并提交 Topology. 初始化在 AbstractAnalyticsTest 中被执行，该类被所有的测试继承。一个叫做 topologyStarted 的静态标志被用来避免当多个 AbstractAnalyticsTest 子类初始化时 AbstractAnalyticsTest 本身被初始化不止一次的情况。

注意那里 sleep 的目的是允许 LocalCluster 在尝试从中恢复结果之前正确的启动。

```
public abstract class AbstractAnalyticsTest extends Assert {
    def jedis
    static topologyStarted = false
    static sync= new Object()
    private void reconnect() {
        jedis = new Jedis(TopologyStarter.REDIS_HOST, TopologyStarter.REDIS_PORT)
    }
    @Before
    public void startTopology(){
```

```
            synchronized(sync){
                reconnect()
                if(!topologyStarted){
                    jedis.flushAll()
                    populateProducts()
                    TopologyStarter.testing = true
                    TopologyStarter.main(null)
                    topologyStarted = true
                    sleep 1000
                }
            }
        }
        ...
        public void populateProducts() {
            def testProducts = [
                [id: 0, title:"Dvd player with surround sound system",
                    category:"Players", price: 100],
                [id: 1, title:"Full HD Bluray and DVD player",
                    category:"Players", price:130],
                [id: 2, title:"Media player with USB 2.0 input",
                    category:"Players", price:70],
                ...
                [id: 21, title:"TV Wall mount bracket 50-55 Inches",
                    category:"Mounts", price:80]
            ]
            testProducts.each() { product ->
            def val =
            "{ \"title\": \"${product.title}\" , \"category\": \"${product.category}\"," +
                " \"price\": ${product.price}, \"id\": ${product.id} }"
            println val
            jedis.set(product.id.toString(), val.toString())
            }
        }
        ...
}
```

在 AbstractAnalyticsTest 类中实现一个叫做 navigate 的方法。为了使不同的测试有一种来模拟用户导航页面行为的方式，该步在 Redis 服务器导航队列中插入导航项。

```
public abstract class AbstractAnalyticsTest extends Assert {
        ...
        public void navigate(user, product) {
            String nav =
                "{\"user\": \"${user}\", \"product\": \"${product}\", \"type\": \"PRODUCT
                \"}".toString()
            println "Pushing navigation: ${nav}"
```

```
            jedis.lpush('navigation', nav)
        }
        ...
}
```

在 AbstractAnalyticsTest 中提供一个叫做 getProductCategory 的方法来从 Redis 服务器中读取特定的关系。不同的测试也需要对统计的结果进行断言来确保 topology 按预期的运行。

```
public abstract class AbstractAnalyticsTest extends Assert {
        ...
        public int getProductCategoryStats(String product, String categ) {
            String count = jedis.hget("prodcnt:${product}", categ)
            if(count == null || "nil".equals(count))
                return 0
            return Integer.valueOf(count)
        }
        ...
}
```

一个测试用例

在下边的小片段断中，你将模拟用户”1”的一些产品浏览记录，然后核实结果。注意在断言确定结果已经被存储到 Redis 之前你要等待两秒钟。(需要记住的是 ProductCategoriesCounterBolt 包含一个计数器的内存拷贝并且在后台将他们发送至 Redis)。

```
package functional
class StatsTest extends AbstractAnalyticsTest {
        @Test
        public void testNoDuplication(){
            navigate("1", "0") // Players
            navigate("1", "1") // Players
            navigate("1", "2") // Players
            navigate("1", "3") // Cameras
            Thread.sleep(2000) // Give two seconds for the system to process the data.
            assertEquals 1, getProductCategoryStats("0", "Cameras")
            assertEquals 1, getProductCategoryStats("1", "Cameras")
            assertEquals 1, getProductCategoryStats("2", "Cameras")
            assertEquals 2, getProductCategoryStats("0", "Players")
            assertEquals 3, getProductCategoryStats("3", "Players")
        }
}
```

扩展性和可用性的说明

为适应本书一个单独章节的大小，该方案的架构被简化了。由于这个原因，你规避了一些对于这个方案的扩展和高可用性来说所必要的复杂性。该架构有两个主要的问题。

该架构中 Redis 服务器不仅仅是单点失败的而且是一个瓶颈。你只能获取 Redis 服务器所能

处理的数据量。Redis 层可以通过使用分片来扩展，并且它的可用性可以通过使用一个 Master/Slave 配置来改进，这需要 topology 和 web 应用资源都做出改变。
另一个弱点是以循环的方式添加机器时，web 应用并没有成比例的扩展。这是因为当产品的统计改变的时候它需要被通知，并且是通知所有相应的浏览器。这个"通知浏览器"的桥接使用 Socket.io 实现，但它需要监听器和通知器被部署在同一台 web 服务器上。这只有在你共享 GET /product/:id/stats 通信和 the POST /news 通信，并且都以相同的标准，确保引用相同产品的请求在相同的服务器上结束的情况下才做的到。

第七章 在 Storm 中使用非 JVM 语言

有时候你需要使用不是基于 JVM 的语言来实现一个 Storm 工程，或者你用其他的语言会感到更舒服，抑或你想使用由其他语言开发的库。
Storm 用 Java 实现，并且本书中你看到的所有的 spouts 和 bolts 都是用 Java 实现的。所以使用例如 Python，Ruby 或者甚至 JavaScript 语言来实现 spouts 和 bolts 是可能的吗？ 答案的可能的！通过使用名为 multilang protocol 的东西实现了这个可能。
multilang protocol 是 Storm 实现的一个特殊的协议，它使用标准的输入输出作为和做 spout 和 bolt 工作的进程的通信通道。消息被编码成 JSON 或者简单的文本通过这个通道进行传递。我们看一个非 JVM 语言的简单的 spout 和 bolt 示例。你将有一个用来产生 1 到 10000 之间的数字的 spout 和一个用来过滤质数的 bolt，都用 PHP 开发的。

在本示例中，我们以一种幼稚的方式来检查质数。有更好的实现方式，但是它们也更复杂并且超出了本示例的范围。

有一个针对 storm 的 PHP DSL 官方实现。在本章中，我们将展示我们的实现来作为示例。首先，定义 topology。

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

有一种方式来指定非 JVM 语言中的 topologies。因为 Storm topologies 是 Thrift 架构，并且 Nimbus 是一个 Thrift 守护进程，因此你可以使用任何你想用的语言来建立和提交 topologies。但这超出了本书的范围。

这里没有新东西。我们看一下 NumbersGeneratorSpout 的实现。

```
public class NumberGeneratorSpout extends ShellSpout implements IRichSpout {
```

```
    public NumberGeneratorSpout(Integer from, Integer to) {
        super("php", "-f", "NumberGeneratorSpout.php", from.toString(), to
                .toString());
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("number"));
    }
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

正如你可能已经注意到的，该 spout 继承自 ShellSpout。这是 Storm 提供的帮助你运行和控制用其他语言编写的 spouts 的特殊类。在这个例子中，它告诉 Storm 怎样执行你的 PHP 脚本。

NumberGeneratorSpout PHP 脚本发射元组到标准输出，读取标准输入来处理 acks 或 fails。在查看 NumberGeneratorSpout.php 脚本的实现之前，更详细地看一下 mutilang protocol 是怎样工作的。

Spout 产生从 from 参数到 to 参数的序列号，然后传递给构造方法。

接下来看一下 PrimeNumbersFilterBolt。该类实现了之前提到的 shell。它告诉 Storm 怎样执行你的 PHP 脚本。Storm 为该目的提供了一个特殊的叫做 ShellBolt 的类，你唯一要做的事情指出怎样运行脚本和在发射时声明域。

```
public class PrimeNumbersFilterBolt extends ShellBolt implements IRichBolt {
    public PrimeNumbersFilterBolt() {
        super("php", "-f", "PrimeNumbersFilterBolt.php");
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("number"));
    }
}
```

在构造方法中，只需告诉 Storm 怎样运行 PHP 脚本。这与下述命令是等价的：

```
php –f PrimeNumbersFilterBolt.php
```

PrimeNumbersFilterBolt PHP 脚本从标准输入读取元组，处理，发射，acks 或 fails 到标准输出。在查看 PrimeNumbersFilterBolt.php 脚本的实现前，我们先详细查看下 mutilang protocol 是怎样工作的。

Mutilang 协议

该协议依赖标准输入输出作为进程间通信的信道。一个脚本需要按下述的步骤来准备工作：

1. 初始握手
2. 启动循环
3. 读写元组


有一个特殊的从你的脚本记录日志的方式，它使用 Storm 内部的日志记录机制，这样你无需实现你自己的日子记录系统。

我们看一下每个步骤的细节及怎样用 PHP 脚本实现它。

初始握手

为了控制进程(启动和停止它)，Storm需要知道脚本正在执行的进程ID(PID)。根据mutilang protocol，当你的进程启动时第一件事是Storm将发送一个包含storm配置，topology上下文，PID目录的JSON对象到标准输入。它看起来有点像下边的代码块：

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

进程必须在 pidDir 指定的目录建一个空文件，文件名为进程ＩＤ，并且以一个 JSON 对象将 PID 写到标准输出。

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

例如，如果你接收到/tmp/example\n 并且你的脚本的 PID 是 123，你应该建立一个空文件在 /tmp/example/123 并且打印{"pid": 123}和结束\n 到标准输出。这就是 Storm 如何记录 PID 及关闭时杀死进程的。我们看一下用 PHP 怎么做：

```
$config = json_decode(read_msg(), true);
$heartbeatdir = $config['pidDir'];

$pid = getmypid();
fclose(fopen("$heartbeatdir/$pid", "w"));
storm_send(["pid"=>$pid]);
flush();
```

你建立了一个叫做 read_msg 的函数来从标准输入读取消息。Mutilang protocol 声明消息是可以被编码成 JSON 的单行或者多行。当 Storm 发送一个以\n 结束的单行时，一个消息就完成了。

```
function read_msg() {
    $msg = "";
    while(true) {
        $l = fgets(STDIN);
        $line = substr($l,0,-1);
        if($line=="end") {
            break;
        }
        $msg = "$msg$line\n";
    }
    return substr($msg, 0, -1);
}
function storm_send($json) {
    write_line(json_encode($json));
    write_line("end");
}
function write_line($line) {
    echo("$line\n");
}
```

flush()的使用很重要；可能存在这样一个直到字符数累计到一定数量才会被刷新的缓冲区。这意味着你的脚本被一直挂起来等待来自 Storm 的输入，它永远不会接收到因为 Storm 转而在等待来自你脚本的输出。因此确保当你的脚本有输出时被及时的刷新是重要的。

启动循环并读写元组

这是最重要的步骤，所有的工作在这里完成。本步骤的实现取决于是否你正在开发一个 spout 或 bolt。

如果是 spout，你应该启动发送元组。如果是 bolt，循环并读取元组，处理并发射，ack 或 fail。

我们看一下发射数字的 spout 的实现：

```
$from = intval($argv[1]);
$to = intval($argv[2]);

while(true) {
    $msg = read_msg();
    $cmd = json_decode($msg, true);
    if ($cmd['command']=='next') {
        if ($from<$to) {
            storm_emit(array("$from"));
            $task_ids = read_msg();
```

```
                $from++;
            } else {
                sleep(1);
            }
        }
        storm_sync();
    }
```

从命令行参数获取 from 和 to 并且开始迭代。每次你从 storm 获取一条 next 消息都意味着你准备发射一个新的元组。

一旦你发射了所有的数字并且没有更多的元组要发射，就休眠一段时间。

为了确保脚本为下一个元组准备好了，Storm 在发送下一个之前等待行 sync\n。为读取命令，只需调用 read_msg()并用 JSON 解码它。

如果是 bolts，有一些小不同。

```
    while(true) {
        $msg = read_msg();
        $tuple = json_decode($msg, true, 512, JSON_BIGINT_AS_STRING);
        if (!empty($tuple["id"])) {
            if (isPrime($tuple["tuple"][0])) {
                storm_emit(array($tuple["tuple"][0]));
            }
            storm_ack($tuple["id"]);
        }
    }
```

循环，从标准输入读取元　组。一旦获取消息，就用 JSON 解码它。如果是一个元组，处理，检查是否是质数。

如果是质数，发射该数；否则忽略它。在任何情况下，ack 元组。

json_decode 函数内 JSON_BIGINT_AS_STRING 的使用是 Java 和 PHP 转换的变通方案。Java 发送非常大的数字，它们被以较低的精度编码成 PHP，这会导致问题。为解决这个问题，当在 JSON 消息中打印数字时，告诉 PHP 解码大数字为字符串并避免使用双引号。该参数工作需要 PHP5.4.0 或更高的版本。

类似 emit，ack，fail 和 log 的消息有如下结构：

Emit

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

这里的数组包含你要为元组发送的值。

Ack

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

这里的 id 是你正在处理的元组的 ID。

Fail

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

同 emit，id 是你正在处理的元组的 ID。

Log

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

把它放到一起使得你有了下边的 PHP 脚本。

对于你的 spout：

```
<?php
    function read_msg() {
        $msg = "";
        while(true) {
            $l = fgets(STDIN);
            $line = substr($l,0,-1);
            if ($line=="end") {
                break;
            }
            $msg = "$msg$line\n";
        }
        return substr($msg, 0, -1);
    }
    function write_line($line) {
        echo("$line\n");
    }
    function storm_emit($tuple) {
        $msg = array("command" => "emit", "tuple" => $tuple);
        storm_send($msg);
```

```
    }
    function storm_send($json) {
        write_line(json_encode($json));
        write_line("end");
    }
    function storm_sync() {
        storm_send(array("command" => "sync"));
    }
    function storm_log($msg) {
        $msg = array("command" => "log", "msg" => $msg);
        storm_send($msg);
        flush();
    }
    $config = json_decode(read_msg(), true);
    $heartbeatdir = $config['pidDir'];
    $pid = getmypid();
    fclose(fopen("$heartbeatdir/$pid", "w"));
    storm_send(["pid"=>$pid]);
    flush();
    $from = intval($argv[1]);
    $to = intval($argv[2]);
    while(true) {
        $msg = read_msg();
        $cmd = json_decode($msg, true);
        if ($cmd['command']=='next') {
            if ($from<$to) {
                storm_emit(array("$from"));
                $task_ids = read_msg();
                $from++;
            } else {
                sleep(1);
            }
        }
        storm_sync();
    }
?>
```

对于你的 bolt:

```
<?php
    function isPrime($number) {
        if ($number < 2) {
            return false;
        }
        if ($number==2) {
            return true;
```

```
        }
        for ($i=2; $i<=$number-1; $i++) {
            if ($number % $i == 0) {
                return false;
            }
        }
        return true;
}
function read_msg() {
        $msg = "";
        while(true) {
            $l = fgets(STDIN);
            $line = substr($l,0,-1);
            if ($line=="end") {
                break;
            }
            $msg = "$msg$line\n";
        }
        return substr($msg, 0, -1);
}
function write_line($line) {
        echo("$line\n");
}
function storm_emit($tuple) {
        $msg = array("command" => "emit", "tuple" => $tuple);
        storm_send($msg);
}
function storm_send($json) {
        write_line(json_encode($json));
        write_line("end");
}
function storm_ack($id) {
        storm_send(["command"=>"ack", "id"=>"$id"]);
}
function storm_log($msg) {
        $msg = array("command" => "log", "msg" => "$msg");
        storm_send($msg);
}
$config = json_decode(read_msg(), true);
$heartbeatdir = $config['pidDir'];
$pid = getmypid();
fclose(fopen("$heartbeatdir/$pid", "w"));
storm_send(["pid"=>$pid]);
flush();
```

```
    while(true) {
        $msg = read_msg();
        $tuple = json_decode($msg, true, 512, JSON_BIGINT_AS_STRING);
        if (!empty($tuple["id"])) {
            if (isPrime($tuple["tuple"][0])) {
                storm_emit(array($tuple["tuple"][0]));
            }
            storm_ack($tuple["id"]);
        }
    }
?>
```

把所有的这些脚本放在工程路径下一个特殊的叫做 mutilang/resources 的文件夹下是重要的。该文件夹被包含在发送的 worker 的 jar 文件中。如果你没有将脚本放置在那个文件夹下，Storm 没法运行它们并且会报告一个错误。

第八章 事务性 Topologies

在 Storm 中，正如本书前边提到的，你可以通过使用 ack 和 fail 策略来确保消息处理。但是如果元组被重放了会发生什么？你怎样确保你不会计数过多？
事务性 Topologies 是包含在 Storm0.7.0 版本中的新特性，它激活消息语义来确保你以一种安全的方式重放元组并且它们只会被处理一次。没有事务性 topologies 的支持，你不可能以一种完全精确、可扩展和容错的方式计数。

事务性 Topologies 是建立标准 Storm spout 和 bolts 之上的一个抽象。

设计

在事务性 topology 中，Storm 使用并行和顺序元组处理的混合模式。Spout 产生的批量的元组被 bolts 并行的处理。这些 bolts 中的一部分被认为是提交者，它们以某种严格排序的方式提交处理过的批量元组。这意味着如果你有两个批量，每个批量包含五个元组，两边的元组会被 bolts 以并行的方式处理，但是提交者 bolts 直到第一个元组被提交成功后才会提交第二个元组。

当处理事务性 Topology 时，可以从源重放批量的元组，甚至有时候是多次重放是非常重要的。所以确保你的数据源--你的 spout 将要连接的那个--具备这个能力。

这可以被描述成两个不同的步骤，或者阶段：
处理阶段
　　完全并行的阶段，许多批量被同时执行。
提交阶段
　　严格排序的阶段，第二个批量直到第一个批量被提交成功后才提交。
把这两个阶段称为 Storm 事务。

storm 使用 zookeeper 来保存事务元数据。缺省的情况下就使用为 topology 服务的那个 zookeeper 来保存元数据。你可以通过覆盖配置键 transactional.zookeeper.servers 和 transactional.zookeeper.port.来更改。

事务实战

为看清事务怎样工作，你将建立一个 Twitter 分析工具。你会读取存储在 Redis 中的 tweets，通过一系列 bolts 处理他们，然后存储--在另一个 Redis 数据库中--所有标签和它们在 tweets 中的频率的列表，所有用户和他们出现在 tweets 中的总计的列表和一个用户及他们标签和频率的列表。
你即将创建的这个工具的 topology 由图 8-1 描述.

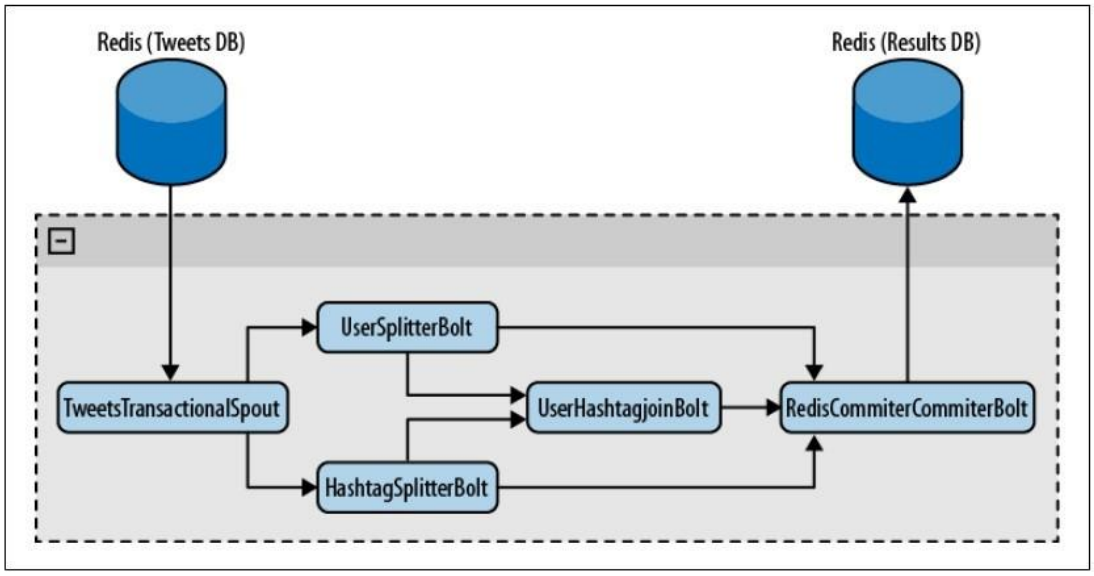

图 8-1 Topology 视图
正如你看到的，TweetsTransactionalSpout 会连接到你的 tweets 数据库并且在 topology 中发射批量的元组。两个不同的 bolts，UserSplitterBolt 和 HashtagSplitterBolt，将从 spout 接收元组。UserSplitterBolt 会分析 tweet 并且查找用户--@后边的单词--并发送这些词一个叫做 users 的自定义流。HashatagSplitterBolt 也分析 tweet，查找#以前的单词，并且发送这些单词到一个叫做 hashtags 的自定义流。第三个 bolt，UserHashtagJoinBolt，会接收两个流并且计算在一个命名用户的 tweet 中一个 hashtag 出现了多少次。为了计数和发送结果，该 bolt 会是 BaseBatchBolt(后边介绍更多)。
最终，最后一个叫做 RedisCommitterBolt 的 bolt，接收这三个--由 UserSplitterBolt，

HashtagSplitterBolt 和 UserHashtagJoinBolt 产生的流。在同一个事物中，它会完成所有计数，并且一旦完成了元组批次的处理就会发送到 Redis。该 bolt 被认为是一种特殊的 bolt，叫做提交者 bolt，后续的章节中会解释它。

为了构建这个 topology，使用 TransactionalTopologyBuilder，类似下边的代码块：

```
TransactionalTopologyBuilder builder =
    new TransactionalTopologyBuilder("test", "spout", new TweetsTransactionalSpout());
builder.setBolt("users-splitter", new UserSplitterBolt(), 4).shuffleGrouping("spout");
builder.setBolt("hashtag-splitter",
    new HashtagSplitterBolt(), 4).shuffleGrouping("spout");
builder.setBolt("user-hashtag-merger", new UserHashtagJoinBolt(), 4)
    .fieldsGrouping("users-splitter","users", new Fields("tweet_id"))
    .fieldsGrouping("hashtag-splitter", "hashtags", new Fields("tweet_id"));
builder.setBolt("redis-committer", new RedisCommiterCommiterBolt())
    .globalGrouping("users-splitter","users")
    .globalGrouping("hashtag-splitter", "hashtags")
    .globalGrouping("user-hashtag-merger");
```

我们看一下怎么在事务topology中实现spout.

The Spout

事务 topology 中的 spout 与标准的 spout 完全不同。

```
public class TweetsTransactionalSpout extends
BaseTransactionalSpout<TransactionMetadata> {
```

正如你在类定义中看到的，TweetsTransactionalSpout继承自BaseTransactionalSpout并带一个泛型类型。你设置在这里的类型被认为是事务元数据。它将在后边从源发送批量的元组时被使用。

在这个例子中，TransactionMetadata被定义为：

```
public class TransactionMetadata implements Serializable {
    private static final long serialVersionUID = 1L;
    long from;
    int quantity;
    public TransactionMetadata(long from, int quantity) {
        this.from = from;
        this.quantity = quantity;
    }
}
```

这里你存储了from和quantity，它们会告诉你具体怎样产生批量的元组。

为完成spout的实现，你需要实现如下方法：

```
@Override
public ITransactionalSpout.Coordinator<TransactionMetadata> getCoordinator(
Map conf, TopologyContext context) {
        return new TweetsTransactionalSpoutCoordinator();
}
@Override
```

```
public backtype.storm.transactional.ITransactionalSpout.Emitter<TransactionMetadata>
getEmitter(
Map conf, TopologyContext context) {
    return new TweetsTransactionalSpoutEmitter();
}
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("txid", "tweet_id", "tweet"));
}
```

在getCoordinator方法中，你告诉Storm哪个类将协调批量元组的产生。在getEmitter中，你告诉Storm哪个类负责从源读取批量的元组并且发射它们到topology的一个流中。最后，如你之前所做的，你需要声明发射了哪些域。

The RQ class

为了使例子更简单些，我们决定封装所有的 Redis 相关的操作在一个类中。

```
public class RQ {
    public static final String NEXT_READ = "NEXT_READ";
    public static final String NEXT_WRITE = "NEXT_WRITE";
    Jedis jedis;
    public RQ() {
        jedis = new Jedis("localhost");
    }
    public long getAvailableToRead(long current) {
        return getNextWrite() - current;
    }
    public long getNextRead() {
        String sNextRead = jedis.get(NEXT_READ);
        if(sNextRead == null)
            return 1;
        return Long.valueOf(sNextRead);
    }
    public long getNextWrite() {
        return Long.valueOf(jedis.get(NEXT_WRITE));
    }
    public void close() {
        jedis.disconnect();
    }
    public void setNextRead(long nextRead) {
        jedis.set(NEXT_READ, ""+nextRead);
    }
    public List<String> getMessages(long from, int quantity) {
        String[] keys = new String[quantity];
```

```
            for (int i = 0; i < quantity; i++)
                keys[i] = ""+(i+from);
            return jedis.mget(keys);
        }
    }
```

仔细阅读每个方法的实现，确保你明白了他们在做什么。

协调器

我们看一下这个例子中协调器的实现。

```
    public static class TweetsTransactionalSpoutCoordinator implements
    ITransactionalSpout.Coordinator<TransactionMetadata> {
        TransactionMetadata lastTransactionMetadata;
        RQ rq = new RQ();
        long nextRead = 0;
        public TweetsTransactionalSpoutCoordinator() {
            nextRead = rq.getNextRead();
        }
        @Override
        public TransactionMetadata initializeTransaction(BigInteger txid,
        TransactionMetadata prevMetadata) {
            long quantity = rq.getAvailableToRead(nextRead);
            quantity = quantity > MAX_TRANSACTION_SIZE ? MAX_TRANSACTION_SIZE :
            quantity;
            TransactionMetadata ret = new TransactionMetadata(nextRead, (int)quantity);
            nextRead += quantity;
            return ret;
        }
        @Override
        public boolean isReady() {
            return rq.getAvailableToRead(nextRead) > 0;
        }
        @Override
        public void close() {
            rq.close();
        }
    }
```

很重要的需要提醒的一点是在整个 topology 中只有一个协调器实例。当协调器被初始化后，它从 Redis 检索一个时序，该时序告诉协调器下一条要读取的 tweet 是哪条。第一次时，该值为 1，意味着接下来要读取的 tweet 是第一条。

第一个要被调用的方法是isReady。它在initializeTransaction之前总是会被调用来确保源已经准备好被读取数据。你要相应的返回true或者false。在这个例子中，检索tweets的数量并把他们和你读到的tweets数量做比较。它们之间的差异是可以读取的tweets数量。如果它大于0，

说明你还有tweets可读。
最后，initializeTransaction被执行。正如你看到的，你用txid和prevMetadata作为参数。第一个参数是一个由Storm产生的唯一的事务ＩＤ，它代表产生的元组的批次。prevMetadata是前一个事务的协调器产生的元数据。
在这个例子中，首先确保有多少tweets可读取。一旦你整理好了，创建一个新的TransactionMetadata，指明哪个是第一个要读的tweet，要读取的量是多少。
你一返回元数据，Storm就把它和txid存到zookeeper中。这确保了一旦有错误，Storm有能力来使发射器重新发送元组。

发射器

创建事务 spout 的最后一步是实现发射器。
我们从下边的实现开始：

```
public static class TweetsTransactionalSpoutEmitter implements
ITransactionalSpout.Emitter<TransactionMetadata> {
    RQ rq = new RQ();
    public TweetsTransactionalSpoutEmitter() {
    }
    @Override
    public void emitBatch(TransactionAttempt tx,
            TransactionMetadata coordinatorMeta, BatchOutputCollector collector) {
        rq.setNextRead(coordinatorMeta.from+coordinatorMeta.quantity);
        List<String> messages = rq.getMessages(coordinatorMeta.from,
            coordinatorMeta.quantity);
        long tweetId = coordinatorMeta.from;
        for (String message : messages) {
            collector.emit(new Values(tx, ""+tweetId, message));
            tweetId++;
        }
    }
    @Override
    public void cleanupBefore(BigInteger txid) {
    }
    @Override
    public void close() {
        rq.close();
    }
}
```

发射器读取源并发射元组到一个流。对于相同的transaction id 和 transaction metadata，发射器总是可以发射相同批次的元组是非常重要的。这样，如果处理一个批次的过程中出错了，Storm可以通过发射器重放相同的transaction id 和 transaction metadata并且确保这个批次被重放了。Storm会增加TransactionAttempt中的attempt id。这样你就可以知道该批次被重放了。

这里emitBatch是一个重要的方法。在这个方法中，使用元数据来作为参数，从Redis中读取tweets。同时增加在Redis中的序列，该序列记录了你到目前为止已读取了多少tweets。当然，还要发射tweets到topology。

The Bolts

首先我们看一下该 topology 中的标准 bolts：

```
public class UserSplitterBolt implements IBasicBolt{
    private static final long serialVersionUID = 1L;
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream("users", new Fields("txid", "tweet_id", "user"));
    }
    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
    @Override
    public void prepare(Map stormConf, TopologyContext context) {
    }
    @Override
    public void execute(Tuple input, BasicOutputCollector collector) {
        String tweet = input.getStringByField("tweet");
        String tweetId = input.getStringByField("tweet_id");
        StringTokenizer strTok = new StringTokenizer(tweet, " ");
        TransactionAttempt tx = (TransactionAttempt)input.getValueByField("txid");
        HashSet<String> users = new HashSet<String>();
        while(strTok.hasMoreTokens()) {
            String user = strTok.nextToken();
            // Ensure this is an actual user, and that it's not repeated in the tweet
            if(user.startsWith("@") && !users.contains(user)) {
                collector.emit("users", new Values(tx, tweetId, user));
                users.add(user);
            }
        }
    }
    @Override
    public void cleanup() {
    }
}
```

正如本章前边提到的，UserSplitterBolt接收元组，解析tweets的文本，并发送@后边的单词或者Twitter用户。HashtagSplitterBolt以一种非常简单的方式工作。

```
public class HashtagSplitterBolt implements IBasicBolt{
    private static final long serialVersionUID = 1L;
```

```
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declareStream("hashtags", new Fields("txid", "tweet_id", "hashtag"));
        }
        @Override
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }
        @Override
        public void prepare(Map stormConf, TopologyContext context) {
        }
        @Override
        public void execute(Tuple input, BasicOutputCollector collector) {
            String tweet = input.getStringByField("tweet");
            String tweetId = input.getStringByField("tweet_id");
            StringTokenizer strTok = new StringTokenizer(tweet, " ");
            TransactionAttempt tx = (TransactionAttempt)input.getValueByField("txid");
            HashSet<String> words = new HashSet<String>();
            while(strTok.hasMoreTokens()) {
                String word = strTok.nextToken();
                if(word.startsWith("#") && !words.contains(word)) {
                    collector.emit("hashtags", new Values(tx, tweetId, word));
                    words.add(word);
                }
            }
        }
        @Override
        public void cleanup() {
        }
    }
```

我们现在看下在UserHashtagJoinBolt中发生了什么。首先要注意到的最重要的事情是它是一个BaseBatchBolt。这意味着会对接收到的元组执行execute方法但不会发送任何新的元组。逐步的，当批次结束的时候，Storm会调用finishBatch方法。

```
    public void execute(Tuple tuple) {
        String source = tuple.getSourceStreamId();
        String tweetId = tuple.getStringByField("tweet_id");
        if("hashtags".equals(source)) {
            String hashtag = tuple.getStringByField("hashtag");
            add(tweetHashtags, tweetId, hashtag);
        } else if("users".equals(source)) {
            String user = tuple.getStringByField("user");
            add(userTweets, user, tweetId);
        }
    }
```

因为你需要将一条tweet中所有的标签与该tweet中提到的用户关联起来并且计数他们出现的次数，你需要对前边bolt的两条流做连接。对整个批次都这样处理，一旦完成了，finishBatch方法会被调用。

```
@Override
public void finishBatch() {
    for (String user : userTweets.keySet()) {
        Set<String> tweets = getUserTweets(user);
        HashMap<String, Integer> hashtagsCounter = new HashMap<String, Integer>();
        for (String tweet : tweets) {
            Set<String> hashtags = getTweetHashtags(tweet);
            if(hashtags != null) {
                for (String hashtag : hashtags) {
                    Integer count = hashtagsCounter.get(hashtag);
                    if(count == null)
                        count = 0;
                    count ++;
                    hashtagsCounter.put(hashtag, count);
                }
            }
        }
        for (String hashtag : hashtagsCounter.keySet()) {
            int count = hashtagsCounter.get(hashtag);
            collector.emit(new Values(id, user, hashtag, count));
        }
    }
}
```

在该方法中，对每一个用户--标签以及它出现的次数，生成并发射一个元组。
你可以在GitHub看到完整的可下载的代码。

提交者 Bolts

正如你已经知道的，在 topology 中批量的元组被协调器和发射器发送。这些批量的元组被并行的处理，并没有特定的顺序。
coordinator bolts 或者是实现了 ICommitter 接口的特殊批量 bolts，或者它在 TransactionalTopologyBuilder 中被用 setCommiterBolt 方法设置过。它与常规的批量 bolts 的主要不同在于当该批次准备好被提交时会执行提交者 bolts 的 finishBatch 方法。这在所有前边的事务被成功的提交后会发生。另外，finishBatch 方法被顺序的执行。所以，当事务 ID 为 1 的批次和事务 ID 为 2 的批次在 topology 中被并行的处理时，正在处理事务 ID 为 2 的批次的提交者 bolt 的 finishBatch 方法只有在事务 ID 为 1 的批次的 finishBatch 方法结束并且没有任何错误的情况下才会被执行。
该类的实现如下：

```
public class RedisCommiterCommiterBolt extends BaseTransactionalBolt
    implements ICommitter {
    public static final String LAST_COMMITED_TRANSACTION_FIELD = "LAST_COMMIT";
```

```
TransactionAttempt id;
BatchOutputCollector collector;
Jedis jedis;
@Override
public void prepare(Map conf, TopologyContext context,
BatchOutputCollector collector, TransactionAttempt id) {
        this.id = id;
        this.collector = collector;
        this.jedis = new Jedis("localhost");
}
HashMap<String, Long> hashtags = new HashMap<String, Long>();
HashMap<String, Long> users = new HashMap<String, Long>();
HashMap<String, Long> usersHashtags = new HashMap<String, Long>();
private void count(HashMap<String, Long> map, String key, int count) {
        Long value = map.get(key);
        if(value == null)
                value = (long) 0;
        value += count;
        map.put(key, value);
}
@Override
public void execute(Tuple tuple) {
        String origin = tuple.getSourceComponent();
        if("users-splitter".equals(origin)) {
                String user = tuple.getStringByField("user");
                count(users, user, 1);
        } else if("hashtag-splitter".equals(origin)) {
                String hashtag = tuple.getStringByField("hashtag");
                count(hashtags, hashtag, 1);
        } else if("user-hashtag-merger".equals(origin)) {
                String hashtag = tuple.getStringByField("hashtag");
                String user = tuple.getStringByField("user");
                String key = user + ":" + hashtag;
                Integer count = tuple.getIntegerByField("count");
                count(usersHashtags, key, count);
        }
}
@Override
public void finishBatch() {
        String lastCommitedTransaction =
  jedis.get(LAST_COMMITED_TRANSACTION_FIELD);
        String currentTransaction = ""+id.getTransactionId();
        if(currentTransaction.equals(lastCommitedTransaction))
                return ;
```

```
            Transaction multi = jedis.multi();
            multi.set(LAST_COMMITED_TRANSACTION_FIELD, currentTransaction);
            Set<String> keys = hashtags.keySet();
            for (String hashtag : keys) {
                Long count = hashtags.get(hashtag);
                multi.hincrBy("hashtags", hashtag, count);
            }
            keys = users.keySet();
            for (String user : keys) {
                Long count = users.get(user);
                multi.hincrBy("users", user, count);
            }
            keys = usersHashtags.keySet();
            for (String key : keys) {
                Long count = usersHashtags.get(key);
                multi.hincrBy("users_hashtags", key, count);
            }
            multi.exec();
        }
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
        }
    }
```

这些都很直观，但是在 finishBatch 方法中有一个非常重要的细节。

```
    ...
    multi.set(LAST_COMMITED_TRANSACTION_FIELD, currentTransaction);
    ...
```

这里你正在存储上一个被提交的事务 ID 到数据库。你为什么要那样做？记住当一个事务失败时，如果有必要的话 Storm 将重放它足够多次。如果你不确定你已经处理过该事务，那么你可以高估，这样整个 topology 的事务性含义都没意义了。所以记住：存储上一个被提交的事务 ID 并且提交前核对它。

分区的事务性 Spouts

对于 spout 来说，从一个分区的集合中读取批量的元组是很常见的。继续这个例子，你可以有几个 Redis 数据库并且 tweets 分散在这些 Redis 数据库中。通过实现 IPartitionedTransactionalSpout，Storm 提供了一些工具来管理每个分区的状态并确保重放的能力。

我们看一下怎样修改前边的 TweetsTransactionalSpout 以使得它可以处理分区。

首先，继承BasePartitionedTransactionalSpout，它实现了IPartitionedTransactionalSpout接口。

```
    public class TweetsPartitionedTransactionalSpout extends
        BasePartitionedTransactionalSpout<TransactionMetadata> {
        ...
    }
```

告诉Storm，哪个是你的协调器。

```
public static class TweetsPartitionedTransactionalCoordinator implements Coordinator {
    @Override
    public int numPartitions() {
        return 4;
    }
    @Override
    public boolean isReady() {
        return true;
    }
    @Override
    public void close() {
    }
}
```

在这个例子中，协调器非常简单。在 numPartitions 方法中，告诉 Storm 你有多少个分区。同时也要注意到你没有返回任何元数据。在一个 IPartitionedTransactionalSpout 中，元数据被发射器直接管理。

我们看一下发射器的实现。

```
public static class TweetsPartitionedTransactionalEmitter
implements Emitter<TransactionMetadata> {
    PartitionedRQ rq = new PartitionedRQ();
    @Override
    public TransactionMetadata emitPartitionBatchNew(TransactionAttempt tx,
    BatchOutputCollector collector, int partition,
    TransactionMetadata lastPartitionMeta) {
        long nextRead;
        if(lastPartitionMeta == null)
            nextRead = rq.getNextRead(partition);
        else {
            nextRead = lastPartitionMeta.from + lastPartitionMeta.quantity;
            rq.setNextRead(partition, nextRead); // Move the cursor
        }
        long quantity = rq.getAvailableToRead(partition, nextRead);
        quantity = quantity > MAX_TRANSACTION_SIZE ? MAX_TRANSACTION_SIZE :
    quantity;
        TransactionMetadata metadata = new TransactionMetadata(nextRead,
    (int)quantity);
        emitPartitionBatch(tx, collector, partition, metadata);
        return metadata;
    }
    @Override
    public void emitPartitionBatch(TransactionAttempt tx, BatchOutputCollector
    collector,int partition, TransactionMetadata partitionMeta) {
        if(partitionMeta.quantity <= 0)
```

```
                return ;
            List<String> messages = rq.getMessages(partition, partitionMeta.from,
            partitionMeta.quantity);
            long tweetId = partitionMeta.from;
            for (String msg : messages) {
                collector.emit(new Values(tx, ""+tweetId, msg));
                tweetId ++;
            }
        }
        @Override
        public void close() {
        }
    }
```

这里有两个重要的方法，emitPartitionBatchNew 和 emitPartitionBatch。在 emitPartitionBatch 中，你从 Storm 接收 partition 参数，它告诉你从哪个分区检索批量的元组。在这个方法中，决定检索哪些 tweets，并产生相应的元数据，调用 emitPartitionBatch，返回元数据，它会被立即存储在 zookeeper 中。

因为事务遍布所有的分区，所以 Storm 会为每个分区发送相同的事务 ID。在 emitPartitionBatch 中，从分区中读取 tweets 并且发送该批次的元组到 topology。如果该批次失败了，Storm 会用存储的元数据来调用 emitPartitionBatch 以实现该批次的重放。

你可以在 ch08-transactional topologies on GitHub 检查代码。

不透明事务 Topologies

目前为止，你可能会假定对于相同的事务 ID，重放一个批次的元组总是可能的。但是在一些场景下，它可能是不可行的。到底发生了什么？

原来，你仍然可以完成明确的一次语义，但是假如事务被 Storm 重放的话，你需要更多的开发工作来保存之前的状态。因为你可以获得相同事务 ID 的不同元组，当在不同时刻发射时，你需要重置到之前的状态并从那里开始。

例如，你要对收到的 tweets 的总数计数，你现在已经数了五个，在最后一个事务 ID 为 321 的事务中，你数了八个。你可以保存这三个值--previousCount=5，currentCount=13 和 lastTransactionId=321。假如事务 ID 为 321 的事务被再次发送并且因为你得到了不同的元组，你数了四个而不是八个，提交者会检测到这是同一个事务 ID，它会将 previousCount 重置为 5，加上新得到的 4 并将 currentCount 更新到 9。

另外，如果前一个事务被取消，那么每个被并行处理的当前事务都会被取消掉。这是为了确保你不会在中间过程丢掉任何东西。

你的 spout 应该实现 IOpaquePartitionedTransactionalSpout 接口并且正如你看到的，协调器和发射器非常简单。

```
    public static class TweetsOpaquePartitionedTransactionalSpoutCoordinator implements
    IOpaquePartitionedTransactionalSpout.Coordinator {
        @Override
```

```
        public boolean isReady() {
            return true;
        }
    }
    public static class TweetsOpaquePartitionedTransactionalSpoutEmitter implements
    IOpaquePartitionedTransactionalSpout.Emitter<TransactionMetadata> {
        PartitionedRQ rq = new PartitionedRQ();
        @Override
        public TransactionMetadata emitPartitionBatch(TransactionAttempt tx,
        BatchOutputCollector collector, int partition,
        TransactionMetadata lastPartitionMeta) {
            long nextRead;
            if(lastPartitionMeta == null)
                nextRead = rq.getNextRead(partition);
            else {
                nextRead = lastPartitionMeta.from + lastPartitionMeta.quantity;
                rq.setNextRead(partition, nextRead); // Move the cursor
            }
            long quantity = rq.getAvailableToRead(partition, nextRead);
            quantity = quantity > MAX_TRANSACTION_SIZE ? MAX_TRANSACTION_SIZE :
        quantity;
            TransactionMetadata metadata = new TransactionMetadata(nextRead,
        (int)quantity);
            emitMessages(tx, collector, partition, metadata);
            return metadata;
        }
        private void emitMessages(TransactionAttempt tx, BatchOutputCollector collector,
        int partition, TransactionMetadata partitionMeta) {
            if(partitionMeta.quantity <= 0)
                return ;
            List<String> messages =
                rq.getMessages(partition, partitionMeta.from, partitionMeta.quantity);
            long tweetId = partitionMeta.from;
            for (String msg : messages) {
                collector.emit(new Values(tx, ""+tweetId, msg));
                tweetId ++;
            }
        }
        @Override
        public int numPartitions() {
            return 4;
        }
        @Override
        public void close() {
```

```
        }
    }
```

最有趣的方法是 emitPartitionBatch，它接收前边提交的元数据。你应该使用该元数据信息来产生一个批次的元组。该批次不必完全相同，正如前边所述，你可能无法重新制造相同的批次。剩余的工作由提交者 bolts 处理，它使用之前的状态。