**准备开始**

在本章，我们要创建一个Storm工程和我们的第一个Storm拓扑结构。

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

**操作模式**

开始之前，有必要了解一下Storm的操作模式。有下面两种方式。

**本地模式**

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

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

我们要在本地模式运行本章的所有例子。

**远程模式**

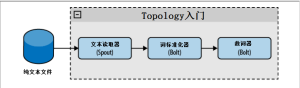
在远程模式下，我们向Storm集群提交拓扑，它通常由许多运行在不同机器上的流程组成。远程模式不会出现调试信息， 因此它也称作生产模式。不过在单一开发机上建立一个Storm集群是一个好主意，可以在部署到生产环境之前，用来确认拓扑在集群环境下没有任何问题。

你将在[第六章](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter6/A RealLife Example.md)学到更多关于远程模式的内容，并在[附录B](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/appendix/B.md)学到如何安装一个Storm集群。

### [Hello World](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "hello-world)

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

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

[](http://ifeve.com/storm%e5%85%a5%e9%97%a8-%e7%ac%ac%e4%ba%8c%e7%ab%a0%e5%87%86%e5%a4%87%e5%bc%80%e5%a7%8b/figure-2-1-getting-started-topology/)

你可以从这个网址下载源码压缩包，[https://github.com/storm-book/examples-ch02-getting\_started/zipball/master](https://github.com/storm-book/examples-ch02-getting_started/zipball/master" \t "http://ifeve.com/getting-started-with-storm-2/_blank)。

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

### [Java安装检查](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "java%E5%AE%89%E8%A3%85%E6%A3%80%E6%9F%A5)

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

java -version

java version "1.6.0\_26"

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

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

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

### [创建工程](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "%E5%88%9B%E5%BB%BA%E5%B7%A5%E7%A8%8B)

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

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

* 下载所有依赖，解压缩它们，把它 们添加到类路径
* 使用[Apache Maven](http://maven.apache.org/)

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

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

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

要运行我们的拓扑，我们可以编写一个包含基本组件的pom.xml文件。

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>storm.book</groupId>

<artifactId>Getting-Started</artifactId>

<version>0.0.1-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.3.2</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

<compilerVersion>1.6</compilerVersion>

</configuration>

</plugin>

</plugins>

</build>

<repositories>

<!-- Repository where we can found the storm dependencies -->

<repository>

<id>clojars.org</id>

<url>http://clojars.org/repo</url>

</repository>

</repositories>

<dependencies>

<!-- Storm Dependency -->

<dependency>

<groupId>storm</groupId>

<artifactId>storm</artifactId>

<version>0.6.0</version>

</dependency>

</dependencies>

</project>

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

一个典型的Maven Java工程会拥有如下结构：

我们的应用目录/

├── pom.xml

└── src

└── main

└── java

| ├── spouts

| └── bolts

└── resources

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

**NOTE**：命令mkdir -p 会创建所有需要的父目录。

### [创建我们的第一个Topology](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "%E5%88%9B%E5%BB%BA%E6%88%91%E4%BB%AC%E7%9A%84%E7%AC%AC%E4%B8%80%E4%B8%AA%E6%8B%93%E6%89%91)

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

### [Spout](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "spout)

pout WordReader类实现了IRichSpout接口。我们将在[第四章](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter4/Spouts.md)看到更多细节。WordReader负责从文件按行读取文本，并把文本行提供给第一个bolt。

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

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

/\*\*

\* 例2-1.src/main/java/spouts/WordReader.java

\*/

package spouts;

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.Map;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichSpout;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

public class WordReader implements IRichSpout {

private SpoutOutputCollector collector;

private FileReader fileReader;

private boolean completed = false;

private TopologyContext context;

public boolean isDistributed() {return false;}

public void ack(Object msgId) {

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

}

public void close() {}

public void fail(Object msgId) {

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

}

/\*\*

\* 这个方法做的惟一一件事情就是分发文件中的文本行

\*/

public void nextTuple() {

/\*\*

\* 这个方法会不断的被调用，直到整个文件都读完了，我们将等待并返回。

\*/

if(completed){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

//什么也不做

}

return;

}

String str;

//创建reader

BufferedReader reader = new BufferedReader(fileReader);

try{

//读所有文本行

while((str = reader.readLine()) != null){

/\*\*

\* 按行发布一个新值

\*/

this.collector.emit(new Values(str),str);

}

}catch(Exception e){

throw new RuntimeException("Error reading tuple",e);

}finally{

completed = true;

}

}

/\*\*

\* 我们将创建一个文件并维持一个collector对象

\*/

public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {

try {

this.context = context;

this.fileReader = new FileReader(conf.get("wordsFile").toString());

} catch (FileNotFoundException e) {

throw new RuntimeException("Error reading file ["+conf.get("wordFile")+"]");

}

this.collector = collector;

}

/\*\*

\* 声明输入域"word"

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

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

}

}

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

public void open(Map conf, TopologyContext context,

SpoutOutputCollector collector) {

try {

this.context = context;

this.fileReader = new FileReader(conf.get("wordsFile").toString());

} catch (FileNotFoundException e) {

throw new RuntimeException("Error reading file ["+conf.get("wordFile")+"]");

}

this.collector = collector;

}

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

public void nextTuple() {

if(completed){

try {

Thread.sleep(1);

} catch (InterruptedException e) {

//什么也不做

}

return;

}

String str;

BufferedReader reader = new BufferedReader(fileReader);

try{

while((str = reader.readLine()) != null){

this.collector.emit(new Values(str));

}

}catch(Exception e){

throw new RuntimeException("Error reading tuple",e);

}finally{

completed = true;

}

}

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

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

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

### [Bolts](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "bolts)

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

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

**NOTE:** 只要必要，bolt或spout会发布若干元组。当调用**nextTuple**或**execute**方法时，它们可能会发布0个、1个或许多个元组。你将在[第五章](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter5/Bolts.md)学习更多这方面的内容。

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

首先我们要声明bolt的出参：

public void declareOutputFields(OutputFieldsDeclarer declarer){

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

}

这里我们声明bolt将发布一个名为“word”的域。

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

public void execute(Tuple input){

String sentence=input.getString(0);

String[] words=sentence.split(" ");

for(String word : words){

word=word.trim();

if(!word.isEmpty()){

word=word.toLowerCase();

//发布这个单词

collector.emit(new Values(word));

}

}

//对元组做出应答

collector.ack(input);

}

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

例2-2是这个类的完整代码。

//例2-2 src/main/java/bolts/WordNormalizer.java

package bolts;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichBolt;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Tuple;

import backtype.storm.tuple.Values;

public class WordNormalizer implements IRichBolt{

private OutputCollector collector;

public void cleanup(){}

/\*\*

\* \*bolt\*从单词文件接收到文本行，并标准化它。

\* 文本行会全部转化成小写，并切分它，从中得到所有单词。

\*/

public void execute(Tuple input){

String sentence = input.getString(0);

String[] words = sentence.split(" ");

for(String word : words){

word = word.trim();

if(!word.isEmpty()){

word=word.toLowerCase();

//发布这个单词

List a = new ArrayList();

a.add(input);

collector.emit(a,new Values(word));

}

}

//对元组做出应答

collector.ack(input);

}

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {

this.collector=collector;

}

/\*\*

\* 这个\*bolt\*只会发布“word”域

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

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

}

}

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

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

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

package bolts;

import java.util.HashMap;

import java.util.Map;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichBolt;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Tuple;

public class WordCounter implements IRichBolt{

Integer id;

String name;

Map<String,Integer> counters;

private OutputCollector collector;

/\*\*

\* 这个spout结束时（集群关闭的时候），我们会显示单词数量

\*/

@Override

public void cleanup(){

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

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

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

}

}

/\*\*

\* 为每个单词计数

\*/

@Override

public void execute(Tuple input) {

String str=input.getString(0);

/\*\*

\* 如果单词尚不存在于map，我们就创建一个，如果已在，我们就为它加1

\*/

if(!counters.containsKey(str)){

conters.put(str,1);

}else{

Integer c = counters.get(str) + 1;

counters.put(str,c);

}

//对元组作为应答

collector.ack(input);

}

/\*\*

\* 初始化

\*/

@Override

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector){

this.counters = new HashMap<String, Integer>();

this.collector = collector;

this.name = context.getThisComponentId();

this.id = context.getThisTaskId();

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {}

}

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

### [主类](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "%E4%B8%BB%E7%B1%BB)

你可以在主类中创建拓扑和一个本地集群对象，以便于在本地测试和调试。**LocalCluster**可以通过**Config**对象，让你尝试不同的集群配置。比如，当使用不同数量的工作进程测试你的拓扑时，如果不小心使用了某个全局变量或类变量，你就能够发现错误。（更多内容请见[第三章](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter3/Topologies.md)）

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

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

TopologyBuilder builder = new TopologyBuilder();

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

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

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

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

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

Config conf = new Config();

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

conf.setDebug(true);

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

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

LocalCluster cluster = new LocalCluster();

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

Thread.sleep(2000);

cluster.shutdown();

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

例2-3是完整的代码

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

import spouts.WordReader;

import backtype.storm.Config;

import backtype.storm.LocalCluster;

import backtype.storm.topology.TopologyBuilder;

import backtype.storm.tuple.Fields;

import bolts.WordCounter;

import bolts.WordNormalizer;

public class TopologyMain {

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

//定义拓扑

TopologyBuilder builder = new TopologyBuilder());

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

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

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

//配置

Config conf = new Config();

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

conf.setDebug(false);

//运行拓扑

conf.put(Config.TOPOLOGY\_MAX\_SPOUT\_PENDING, 1);

LocalCluster cluster = new LocalCluster();

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

Thread.sleep(1000);

cluster.shutdown();

}

}

[观察运行情况](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "%E8%A7%82%E5%AF%9F%E8%BF%90%E8%A1%8C%E6%83%85%E5%86%B5)

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

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

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

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

[结论](https://github.com/runfriends/GettingStartedWithStorm-cn/blob/master/chapter2/Hello World Storm.md" \l "%E7%BB%93%E8%AE%BA)

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