阿里巴巴、雅虎、Twitter、Groupon等众多知名互联网企业
正在使用的大数据实时计算最新利器

从零开始学 Storm

赵必厦 程丽明 编著

- 国内首本Storm技术教程
- 技术基础、平台部署与应用案例完全解析

清华大学出版社

从零开始学 Storm
赵必厦 程丽明 编著
清华大学出版社

从零开始学

Storm

赵必厦 程丽明 编著

清华大学出版社
北 京

内 容 简 介

本书由基础知识、安装与部署、研发与维护、进阶知识、企业应用 5 个模块构成，并细分为 20 个章节，其中“基础知识”6 章、“安装与部署”4 章、“研发与维护”4 章、“进阶知识”5 章、“企业应用”1 章，分别介绍了 Storm 的安装与配置、Storm 的基本原理、Topology 组件、Spout 组件、Bolt 组件、ZooKeeper 集群、实战环节等内容，包括理论基础、环境搭建、研发准备、应用案例等。

本书理论联系实际，通过大量实例分析，让读者在较短的时间内掌握 Storm 的使用，搭建并研发出自己的基于 Storm 的大数据处理平台。

本书适合所有大数据处理、实时流数据处理、Storm 的开发者或爱好者，也适合高等院校和培训学校相关专业的师生参考使用。

图书在版编目（CIP）数据

从零开始学 Storm/赵必厦，程丽明编著. —北京：清华大学出版社，2014
ISBN 978-7-302-37245-5

Ⅰ. ①从… Ⅱ. ①赵… ②程… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字（2014）第 154066 号

责任编辑：王金柱
封面设计：王　翔
责任校对：闫秀华
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqboOK 按钮.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　张**：23.75　**字　数**：608 千字
版　　次：2014 年 9 月第 1 版　　**印　次**：2014 年 9 月第 1 次印刷
印　　数：1~3000 册
定　　价：59.00 元

产品编号：056158-01

前 言

Storm是一个免费开源的分布式实时计算系统。Storm能轻松可靠地处理无界的数据流，就像Hadoop批处理一样对数据进行实时处理；但是Storm能持续运作下去，并且Storm的使用十分简单，开发人员可以使用任何编程语言对它进行操作，得到满意的结果。

本书以Storm官方网站最新的Release版本Storm 0.9.0.1 进行讲解，从零开始，使读者在较短的时间内系统掌握Storm的理论基础，面向Linux平台搭建与研发自己基于Storm的大数据处理平台。全书分为 5 个模块，共 20 章内容，其中“基础知识”6 章、“安装与部署”4 章、“研发与维护”4 章、“进阶知识”5 章、“企业应用”1 章，分别介绍了Storm的安装与配置、Storm的基本原理、Topology组件、Spout组件、Bolt组件、ZooKeeper集群、实战环节等内容，包括理论基础、环境搭建、研发准备、应用案例等。

作为国内第一本介绍Storm方面的书籍，本书的Storm理论部分主要参考了Storm Wiki；另外，为了更好地理解Storm，也参考了不少Storm爱好者的中文翻译文章；而Storm的应用部分，是本书编者的实战应用经验与理论相结合的结晶。本书在编写过程中，得到了多位Storm开发者/爱好者的帮助以及对本书的校验和建议，在此表示感谢，特别要感谢赵鸿、陈险峰两位同志。

由于编者水平有限，且本书涉及的知识点较多，书中难免有不妥和错误之处，敬请广大读者批评指正。

书中的源代码主要来自storm-starter项目，下载地址为：https://github.com/nathanmarz/storm-starter。

为了更有效地学习，建议读者在开始学习或阅读本书前先了解Linux的基本操作、Shell脚本的基本语法、Java语言的基本语法等内容。

编者

2014 年 4 月

目 录

第 1 章 Storm 简介

第 2 章　Storm 的基本知识

第 3 章　拓扑详解

第 4 章 组件详解

第 5 章 Spout 详解

第 8 章　基础软件的安装与使用

第 9 章　Storm 的安装与配置

第 10 章　Storm 集群搭建实践

第 11 章　准备 Storm 的开发环境

第 12 章 开发自己的 Storm 应用

第 13 章 storm-starter 详解

第 14 章　研发与集群管理技巧

第 15 章　DRPC 详解

第 16 章　事务拓扑详解

第 17 章　Trident 详解

第1章

Storm简介

本章主要介绍 Storm 的起源，Storm 的基本概念、特征和优势，从哪里可以找到 Storm 的学习和源代码参考的网站，以及 Storm 的应用现状和发展趋势。通过本章的学习，读者将对 Storm 有一个基本的了解。

1.1 什么是 Storm

Storm 是一个免费开源的分布式实时计算系统。Storm 能轻松可靠地处理无界的数据流，就像 Hadoop 批处理一样对数据进行实时处理；但是 Storm 能持续地运作下去，并且要掌握 Storm 十分简单，开发人员可以使用任何编程语言对它进行操作，得到满意的结果。

Storm 能用到很多场景中，包括实时分析、在线机器学习、连续计算、分布式 RPC、ETL 等。Storm 的处理速度非常快，每个节点每秒钟可以处理 100 万条消息。同时，Storm 是可伸缩的、容错的，并且能保证数据根据用户的设定被妥善处理，便于进行设置和操作。Storm 的抽象示意图如图 1.1 所示。

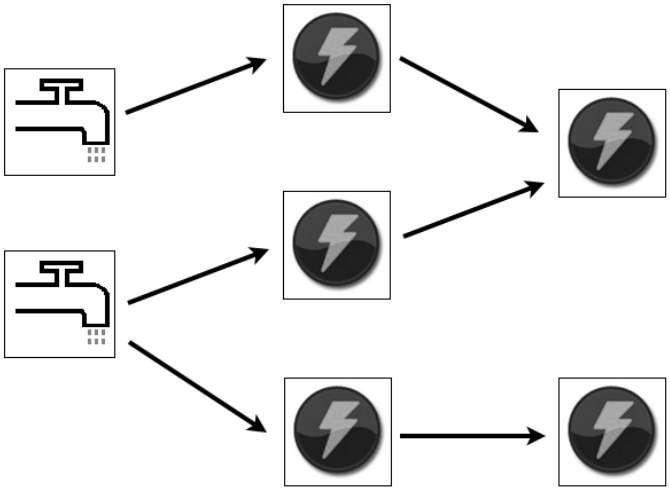


图 1.1　Storm 的抽象示意图

Storm 集成了许多消息队列和数据库技术。Storm 的 Topology 消耗数据流，以任意复杂的方式处理这些流，并且在每个需要计算的阶段以这些流进行重新划分。Storm 的开发语言主要是 Java 和 Clojure，其中 Java 定义骨架，Clojure 编写核心逻辑。Storm 具有多语言功能，Java 是 Storm 的主要使用语言，但 Python 也是 Storm 的常用语言之一（本书的一些例子将使用 Python 语言）。

Storm 公开一组原语进行实时计算。使用这些简单的原语，能实现类似 MapReduce 的效果。Storm 的原语极大地简化了并行实时计算的编写，最终使并行批处理的编写变得易于实现。

Storm 具有的关键特征如下。

（1）用例非常广泛（Extremely broad set of use cases）

Storm 可以用于处理消息和更新数据库（即流处理），完成数据流的持续查询任务，然后把结果流送到客户端（持续计算）。Storm 并行密集查询能持续进行，完成搜索查询（分布式 RPC）和更多的业务功能。仅一个小小的 Storm 的原语集便可以处理惊人数量的用例。

（2）可伸缩性（Scalable）

Storm 每秒处理大量的消息。要对一个拓扑进行扩展，用户所需要做的仅仅是向拓扑中添加主机，然后增加拓扑的并行设置。让我们来看下 Storm 规模化的一个例子，最初的 Storm 应用程序在一个 10 节点集群（作为拓扑的一部分）上每秒处理 1 000 000 条消息，这还包括每秒数以百计的数据库调用。Storm 使用 ZooKeeper 协调集群，使其可以扩展到更大的集群规模。

（3）保证没有数据丢失（Guarantees no data loss）

一个实时系统必须强有力地保证数据能按照用户设定而被成功处理。一个丢失数据的系统是不优秀的，仍拥有非常有限的用例。Storm 能保证每条消息都会被妥善处理，这与其他系统（例如 S4）形成了非常直接的对比。

（4）强壮的鲁棒性（Extremely robust）

不同于鲁棒性差的甚至难以管理的 Hadoop 系统，Storm 集群非常健壮，能有效地运行。强壮的鲁棒性是 Storm 项目的一个明确目标，旨在让用户尽可能简单地管理 Storm 集群。

（5）容错性好（Fault-tolerant）

如果在执行用户设定的计算时有错误发生，Storm 只需要重新分配任务再执行。Storm 能保证计算可以永远运行，直到用户结束计算进程为止。

（6）编程语言无关（Programming language agnostic）

健壮的、可伸缩的实时处理不应该局限于单一的平台。Storm 的拓扑和处理组件可以使用任何语言定义。这一编程语言无关性使得 Storm 可以被几乎任何一个用户轻松使用。

1.2　Storm 的诞生

Storm 是随着实时大数据处理的需求而生的，最早用在微博社交工具 Twitter 上，在分布式的环境下不间断地实时处理少量的数据。Storm 获得了极大的成功。之后由 Twitter 开源，成为处理实时大数据的最实用工具之一。

1.2.1　从 Twitter 说起

Twitter（中文译名：推特）是美国的一个社交网络及微博客服务的网站，由利兹·斯通、埃文·威廉姆斯和杰克·多尔西于 2006 年共同创建，总部位于美国加州旧金山。Twitter 利用无线网络、有线网络等通信技术进行即时通信，是微博的鼻祖。它允许用户将自己的最新动态

和想法以短消息的形式（推文）发布（发推），可绑定 IM 即时通信软件。所有的 Twitter 消息都被限制在 140 个字符之内。

1.2.2 Twitter 需要处理大批实时性要求高的大数据业务

Twitter 业务的实时处理需求极高。几十亿的用户在不同高峰段通过手机、平板电脑、个人计算机等终端发布大量（说海量真是名副其实）的信息，同时大量的用户在 Twitter 上从这些海量的信息中搜索、查看关心的话题并且对它们进行转发和评论。这一切都需要强大的实时处理海量数据的能力。

除基本的推文发布功能之外，Twitter 还有大量的统计类应用，主要包括 doesfollow（判断某用户是否 follow 了另一用户）、TweetStats（分析用户自身或其他用户的 Twitter 内容，如发推常见时间段、主要交流用户等）、Twitterholic（统计出用户的一些基本数据，更能将这些数据与其他 Twitter 用户进行比较，而得出一个排名）、TwitterRank（根据用户所有推文和其他用户互动等信息，进行 Twitter 的细分排名）、Tweetwasters（在 Twitter 上总的花费时间及细项统计）和 monitter（根据用户设定选择三个关键词进行监控）。从以上的简要介绍来看，这些业务都需要实时对最新的大量数据进行统计。

Twitter 本身还有一些需要进行大数据处理的业务，另外还向第三方开发者提供开放 API（Twitter 将自己的网站服务封装成一系列 API 开放出去，供第三方开发者使用以开发 Twitter 相关的应用，通过 Twitter 获得收益）。Twitter 需要对这些应用提供实时的大数据服务，主要有：TwitterMap（用户根据 Twitter 的 Username 进行地理位置的搜索，显示用户公开的 Twitter 留言以及地理位置等相关信息），TwitterBar（将用户当前浏览的网站地址收藏到自己的 Twitter 账号中，还可见当前访问网站的相关信息等），Twittervision（在 Google Map 上实时显示用户更新 Twitter 的内容），Twitter tools（WordPress 插件，用户可以在自己的 WP 平台上发送以及显示自己的 Twitter 留言等），Twitteroo（安装在 Windows 下的桌面软件，允许用户在不登入 Twitter 的情况下向自己的账号中发送信息），Twitter Badges（Twitter 提供的一款小型应用，用户可以通过它将自己的心情放在自己的博客中）。

1.2.3 Storm 帮助 Twitter 解决实时海量大数据处理问题

过去的十年经历了数据处理的革命。MapReduce、Hadoop 以及相关技术，已经能够对先前难以想象的大规模数据进行存储和处理。然而，这些数据处理技术并不是实时的，也注定不是实时的。没有高手会把 Hadoop 变成一个实时系统，实时数据处理相对于批处理有一个从本质上不同的需求集。

然而，企业越来越要求超大规模的实时数据处理。缺乏实时处理能力的 Hadoop 已经成为数据处理生态系统中最大的缺陷。

Storm 填补了这一缺陷。Storm 是一个分布式实时计算系统。类似于提供了一组通用原语进行批处理的 Hadoop，Storm 提供了一组通用原语进行实时计算。Storm 是简单的，可以使用

任何编程语言，很多公司使用它并且有很浓厚的兴趣去使用它。

在 Storm 出现之前，通常需要手工搭建一个消息队列和 Worker 组成的网络进行实时处理。工作节点会处理从消息队列弹出的消息，然后更新数据库，并发送新消息到其他队列进行进一步处理。可惜，这种方法具有严重的局限性。

（1）很乏味

用户用大部分的开发时间来配置向何处发送消息、部署工作节点、部署中间队列等工作，而用户关心的实时处理逻辑只占用户代码库中一个相对小的百分比。

（2）系统很脆弱

有很小的容错功能，用户的职责是保证每个工作节点和消息队列运行起来。

（3）很难对规模进行扩展

当一个工作节点或队列的消息吞吐量达到高点，用户需要决定对数据如何传播进行划分。用户需要重新配置其他工作节点，让它们知道发送消息的新位置。这就引入了移动组件和新的可以失败的组件。

尽管可将队列和工作节点范例分解为大量的消息，但是消息处理显然是实时计算的基本范例。问题是如何在不丢失数据，扩展到很大规模，并且可以简单地使用和操作的前提下，做到这些？而 Storm 可满足这一目标。

2011 年 7 月，Twitter 正式收购了 BackType 公司。BackType 公司提供的服务主要为：分析一个组织在 Twitter 发布的信息的影响力，并且总结分析 Twitter 消息为他人重复转发的频率。同年 8 月 4 日 Twitter 将 Storm 正式开源。具有实时、快速地处理海量大数据的能力，Storm 能帮助 Twitter 和其他有此类需求的企业解决实时大数据处理问题。

1.3　Storm 的成长

Storm 自开源后，得到了原项目小组和其他热心贡献者的帮助，其组成技术不断地发展，拥有了广大的支持者和技术兴趣者。这些支持者和技术兴趣者中也成立了许多技术支持网站，以向学习者提供帮助及技术爱好者之间相互交流。

1.3.1　Storm 正式开源

Twitter 将 Storm 正式开源了，这是一个分布式的、容错的实时计算系统，它被托管在 GitHub 上，遵循 Eclipse Public License 1.0。Storm 是由 BackType 开发的实时处理系统，BackType 现在已在 Twitter 麾下。GitHub 上的最新版本是 Storm 0.9.0.1，基本是用 Clojure 写的。

Storm 为分布式实时计算提供了一组通用原语，可用于“流处理”之中，实时处理消息并更新数据库。这是管理队列及工作者集群的另一种方式。Storm 也可被用于“连续计算”

（Continuous Computation），对数据流做连续查询，在计算时就将结果以流的形式输出给用户。它还可用于“分布式 RPC”，以并行的方式运行昂贵的运算。

Storm 可以方便地在一个计算机集群中编写与扩展复杂的实时计算，Storm 之于实时处理，就好比 Hadoop 之于批处理。Storm 保证每个消息都会得到处理，而且它速度很快——在一个小集群中，每秒可以处理数以百万计的消息。更棒的是可以使用任意编程语言来进行开发。

1.3.2 Storm 的核心技术和基本组成

Storm 框架的核心由 7 个部分组成，如图 1.2 所示，它们同时也是 Storm 的基本组成部分。

图 1.2 Storm 的核心技术组成

- Topology（拓扑）

一个拓扑是一个图的计算。用户在一个拓扑的每个节点包含处理逻辑，节点之间的链接显示数据应该如何在节点之间传递。Topology 的运行是很简单的。

- Stream（流）

流是 Storm 的核心抽象。一个流是一个无界 Tuple 序列，Tuple 可以包含整形、长整形、短整形、字节、字符、双精度数、浮点数、布尔值和字节数组。用户可以通过定义序列化器，在本机 Tuple 使用自定义类型。

- Spout（喷口）

Spout 是 Topology 流的来源。一般 Spout 从外部来源读取 Tuple，提交到 Topology（如 Kestrel 队列或 Twitter API）。Spout 可分为可靠的和不可靠的两种模式。Spout 可以发出超过一个流。

- Bolt（螺栓）

Topology 中的所有处理都在 Bolt 中完成。Bolt 可以完成过滤、业务处理、连接运算、连接、访问数据库等业务。Bolt 可以做简单的流的转换，发出超过一个流，主要方法是 execute 方法。完全可以在 Bolt 中启动新的线程做异步处理。

- Stream grouping（流分组）

流分组在 Bolt 的任务中定义流应该如何分区。Storm 有 7 个内置的流分组接口随机分组（Shuffle grouping）、字段分组（Fields grouping）、全部分组（All grouping）、全局分组（Global grouping）、无分组（None grouping）、直接分组（Direct grouping）、本地或者随机分组（Local

or shuffle grouping）。

- Task（任务）

每个 Spout 或者 Bolt 在集群执行许多任务。每个任务对应一个线程的执行，流分组定义如何从一个任务集到另一个任务集发送 Tuple。可通过 TopologyBuilder 类的 setSpout()和 setBolt()方法来设置每个 Spout 或者 Bolt 的并行度。

- Worker（工作进程）

Topology 跨一个或多个 Worker 节点的进程执行。每个 Worker 节点的进程是一个物理的 JVM 和 Topology 执行所有任务的一个子集。

1.3.3　Storm 的项目小组

Storm 项目小组的构成如图 1.3 所示。

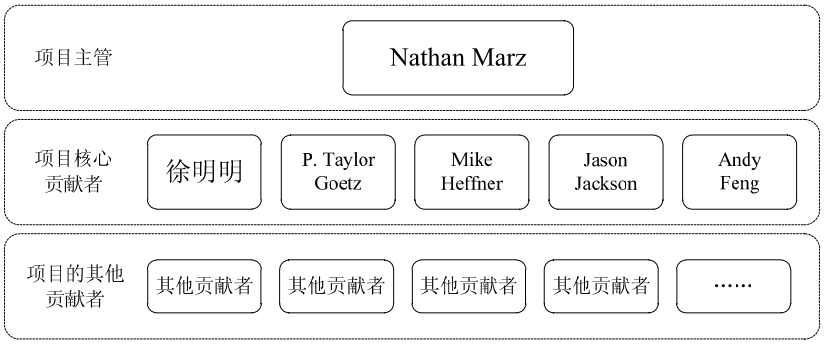


图 1.3　Storm 项目小组的构成

1. 项目主管

Nathan Marz，Storm 的作者和主工程师，Twitter 首席工程师。

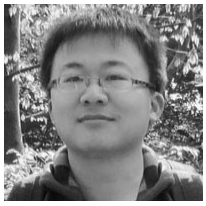
Nathan Marz 原是 BackType 的主力工程师。BackType 于 2011 年 7 月被 Twitter 收购。收购后不久，Storm 对外开源，许多互联网公司纷纷采纳这一系统。

Nathan Marz 依然带领其团队继续从事 Storm 的相关工作。过去很长时间内 Nathan Marz 热衷于研究 Java、Ruby、Python，现在则使用 Clojure。Clojure 非常有趣，它给 Storm 项目组带来了巨大的效能。

Nathan Marz 坚信开源的力量，并发表了一些开源项目。Cascalog 是用于 MapReduce 的高度抽象的库，用 Clojure 编写，拥有活跃的用户社区，并有大量企业使用。Storm 用十分流行的 Java、Scala 或 Clojure 语言编写。目前 Nathan Marz 正在编写 *Big Data* 一书，主要介绍可扩展的实时数据系统的原则与实践。

2. 项目核心贡献者

（1）徐明明



中国阿里巴巴的徐明明是 Storm 项目的核心贡献者。他的英文名为 James Xu，Twitter 账号为 xumingming。

徐明明于 2007 年毕业，一直从事互联网方面的工作，目前在阿里巴巴做交易相关的开发工作。当 Storm 开源的时候他正好在做数据统计相关的项目，对 Storm 产生了浓厚的兴趣，并且为 Storm 的发展做出了巨大的贡献，属于 Storm 的学术型使用者。图 1.4 显示了徐明明对 Storm 的贡献。

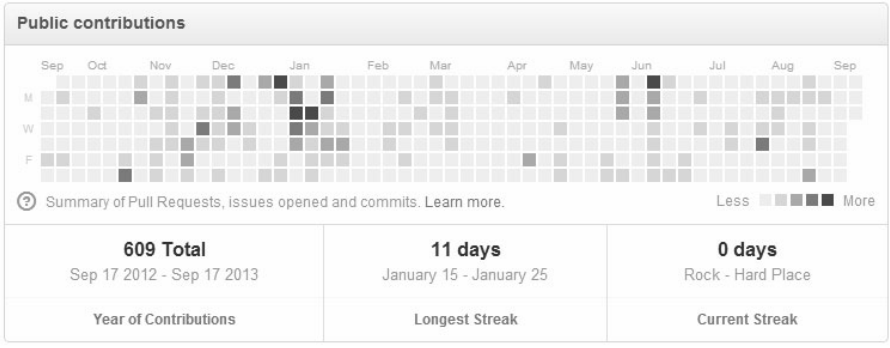


图 1.4　徐明明的贡献图

徐明明主要关注 java、Clojure、开源、大数据相关的技术。他翻译了《Clojure 编程》一书（http://book.douban.com/subject/21661495/）。

可以从以下网站访问徐明明的最新成果：https://github.com/xumingming，http://stackoverflow.com/users/238546/james-xu，http://weibo.com/64398966。

（2）P. Taylor Goetz

P. Taylor Goetz 是 Storm 项目在 GitHub 网站上排名第 2 的核心贡献者，主要的贡献有 storm-signals、storm-jms、storm-cassandra、storm-maven-plugin 和 duke。他对 storm 项目特别是最新的 0.9 版本做出了极大的贡献，图 1.5 显示了 P. Taylor Goetz 对 Storm 的贡献。

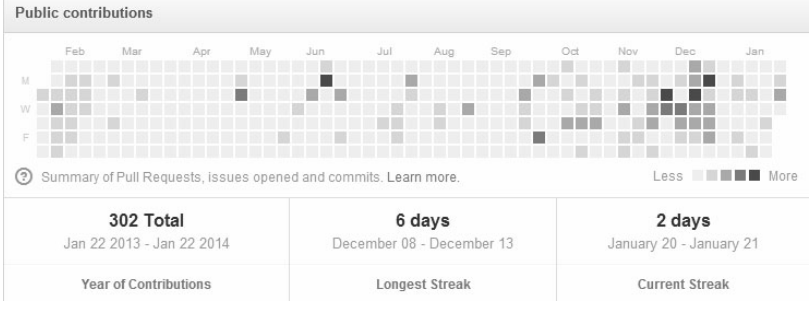


图 1.5　P. Taylor Goetz 的贡献图

可以从以下网站访问 P. Taylor Goetz 的最新成果：https://github.com/ptgoetz，http://ptgoetz.github.io/blog/archives/。

（3）Mike Heffner

Mike Heffner 是 Storm 项目在 GitHub 网站上排名第 3 的核心贡献者，主要的贡献有 awsam、rails_datatables、octoglassformtastic-sass 和 ofxsh。他对 Storm 的贡献如图 1.6 所示。

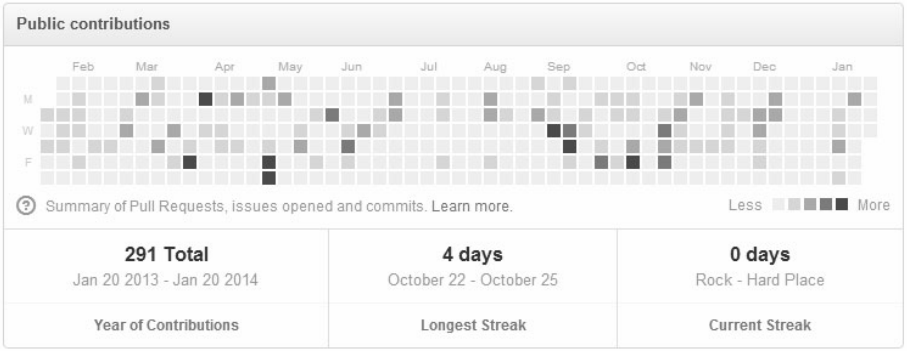


图 1.6　Mike Heffner 的贡献图

Mike Heffner 的 Twitter 主页是：http://twitter.com/mheffner。您可以通过 Twitter 与他进行 Storm 技术的交流。

（4）Jason Jackson

Jason Jackson 在 Twitter、BackType、Google、UWaterloo 担任过软件工程师。Jason Jackson 的 Twitter 主页是：http://twitter.com/jason_j。您可以通过 Twitter 与他进行 Storm 技术的交流。

（5）Andy Feng

Andy Feng 是 Storm 项目的核心贡献者。Andy Feng 主要关注 Java、Shell、JavaScript 语言。同样，Andy Feng 对 Storm 做出了巨大的贡献，他对 Storm 的贡献如图 1.7 所示。

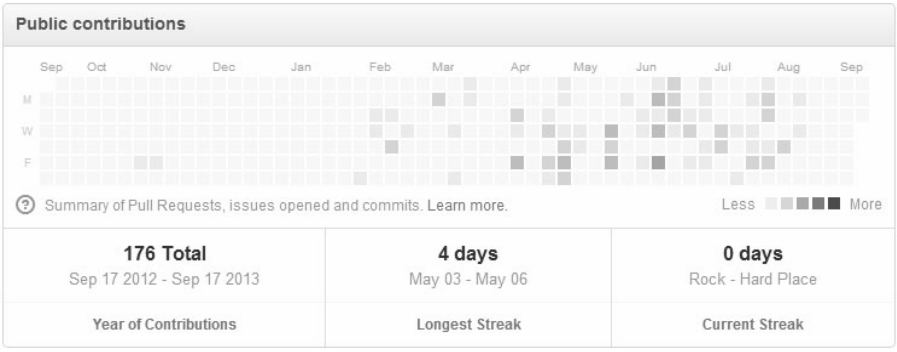


图 1.7　Andy Feng 的贡献图

可以从以下网站位置访问 Andy Feng 的最新成果：https://github.com/anfeng。

3. 项目的其他贡献者

以下是 Storm 项目的其他贡献者以及他们的个人/贡献成果介绍。

Christopher Bertels

http://twitter.com/bakkdoor

Michael Montano

http://twitter.com/michaelmontano

Dennis Zhuang

https://github.com/killme2008

Trevor Smith

https://github.com/trevorsummerssmith

Ben Hughes

https://github.com/schleyfox

Alexey Kachayev

https://github.com/kachayev

Haitao Yao

https://github.com/haitaoyao

Dan Dillinger

https://github.com/ddillinger

Kang Xiao

https://github.com/xiaokang

Gabriel Grant

https://github.com/gabrielgrant

Travis Wellman

https://github.com/travisfw
Kasper Madsen
https://github.com/KasperMadsen
Michael Cetrulo
https://github.com/git2samus
Thomas Jack
https://github.com/tomo
Nicolas Yzet
https://github.com/nicoo
Fabian Neumann
https://github.com/hellp
Soren Macbeth
https://github.com/sorenmacbeth
Ashley Brown
https://github.com/ashleywbrown
Guanpeng Xu
https://github.com/herberteuler
Vinod Chandru
https://github.com/vinodc
Martin Kleppmann
https://github.com/ept
Evan Chan
https://github.com/velvia
Sjoerd Mulder
https://github.com/sjoerdmulder
Yuta Okamoto
https://github.com/okapies
Barry Hart
https://github.com/barrywhart
Sergey Lukjanov
https://github.com/Frostman
Ross Feinstein
https://github.com/rnfein
Junichiro Takagi
https://github.com/tjun

Bryan Peterson
https://github.com/Lazyshot
Sam Ritchie
https://github.com/sritchie
Stuart Anderson
https://github.com/emblem
Robert Evans
https://github.com/revans2
Lorcan Coyle
https://github.com/lorcan
Derek Dagit
https://github.com/d2r
Andrew Olson
https://github.com/noslowerdna
Gavin Li
https://github.com/lyogavin
Tudor Scurtu
https://github.com/tscurtu
Homer Strong
https://github.com/strongh
Sean Melody
https://github.com/srmelody
Jake Donham
https://github.com/jaked
Ankit Toshniwal
https://github.com/ankitoshniwal

1.3.4 Storm 的技术支持网站

1. Storm 项目的官方网址

Storm 项目的官方网址为：

http://storm-project.net/

使用浏览器打开官网网址，其首页如图 1.8 所示。

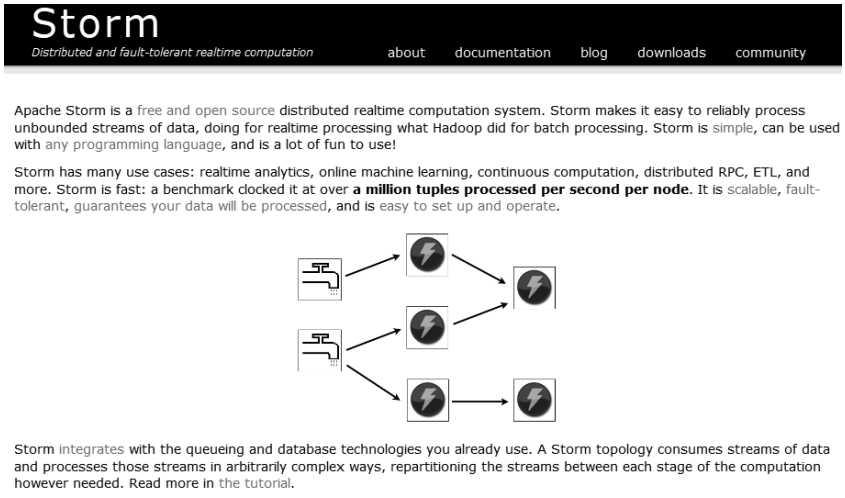


图 1.8　Storm 项目的官方网址图

2. GitHub 上的 Storm 项目

Storm 项目在 GitHub 上的网址为：

https://github.com/nathanmarz/storm

在 GitHub 主页 https://github.com/上可以搜索到 GitHub 上与 Storm 相关的所有项目，如图 1.9 和图 1.10 所示。

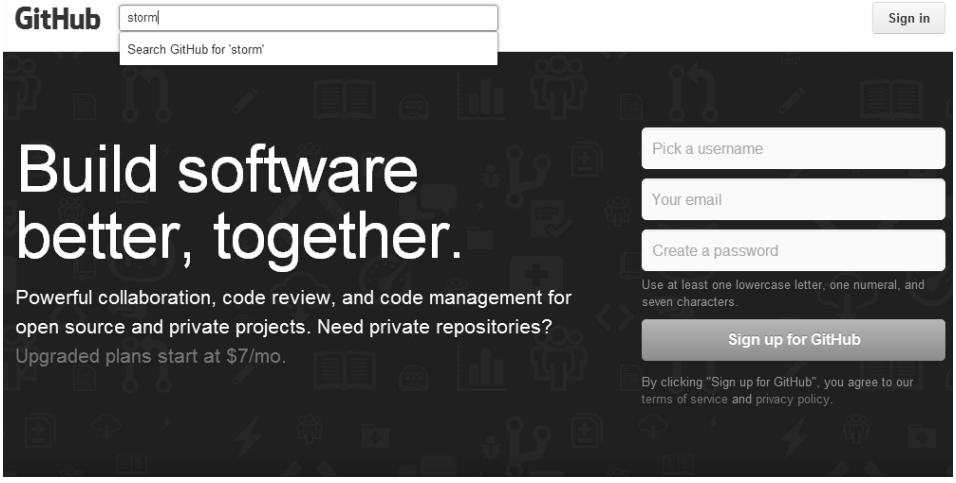


图 1.9　在 GitHub 网站上搜索 Storm 项目

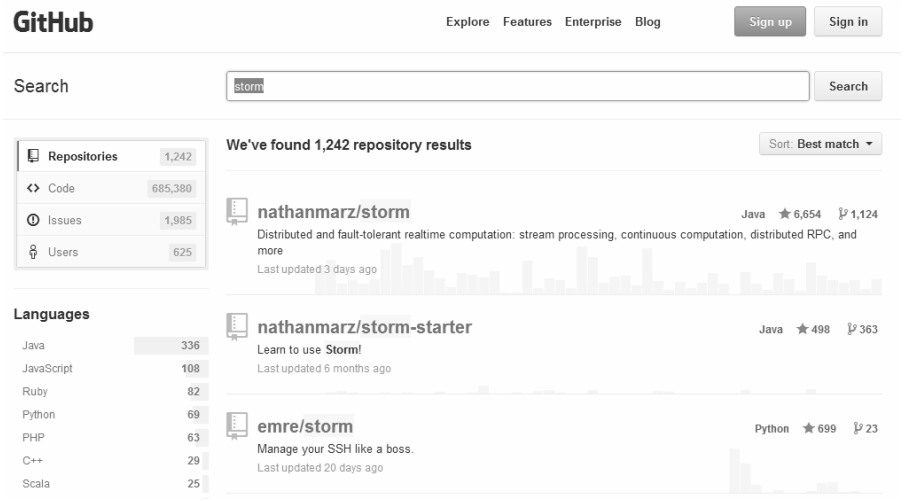


图 1.10　GitHub 网站上的 Storm 项目

在这个网站上，可以找到 Storm 的相关代码和相关项目。

特别说明，GitHub 网站上的 Storm Wiki 有 storm-contrib 项目外的其他值得注意的 Storm 相关项目的链接。

此外，还可以在 Storm Wiki 上找到 Storm 的相关文档和教程，Storm Wiki 的网址为

https://github.com/nathanmarz/storm/wiki

GitHub 网站上的 Storm Wiki 如图 1.11 所示。

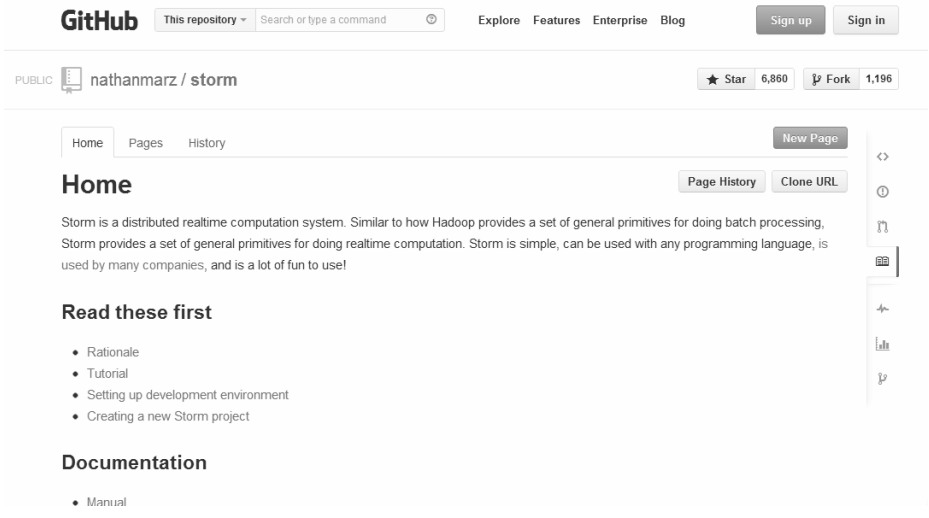


图 1.11　GitHub 网站上的 Storm wiki

3. 在 Wiki 上的 Storm 介绍

在 Wiki 上可以对 Storm 项目有一个总体的了解，可以找到 Storm 的一些相关网站链接，如图 1.12 所示。

图 1.12　Storm Wiki 的网站图

4. Storm 的邮件列表

Storm 的邮件列表地址如下：

http://groups.google.com/group/storm-user

用户可在 Storm 的邮件列表中进行提问。

用户也可以到 freenode 网站（http://freenode.net/）的#storm-user 的空间找到 Storm 开发者来帮助你。

1.4　Storm 的优势

Storm 的主要优势如下。

1.4.1　集成多种技术

Storm 集成了一些消息队列系统和数据库系统，Storm 的 Spout 抽象使得它很容易集成一些新的消息队列系统。消息队列集成主要有：

- Kestrel
- RabbitMQ/AMQP
- Kafka
- JMS

同样，Storm 也可以很容易地和数据库系统集成。简单打开一个数据库连接，就可以像平常一样进行读写，Storm 会自动完成并行化、分区、在失败时尝试连接等操作。

1.4.2 简单的 API

Storm 有简单且易于使用的 API。当使用 Storm 编程的时候，用户可以使用相应的 API 来操纵和变换 Tuple 的流。一个 Tuple 是一个命名的值列表。Tuple 可以包含任何类型的对象，如果想使用一个 Storm 不知道的类型，可以通过序列化注册该类型然后在程序中进行应用。

Storm 只有三个抽象类型：Spout、Bolt 和拓扑。Spout 是计算流的来源。通常在系统中 Spout 从 Kestrel、RabbitMQ 和 Kafka 等消息队列进行读取，但 Spout 还可以生成自己的流或者从 Twitter 流的 API 的某个地方读取。Spout 实现了现有的大多数队列系统。

Bolt 处理任意数量的输入流，并且产生任意数量的新的输出流。大多数的逻辑计算进入 Bolt，如功能、过滤、流连接与数据库交互等。

Topology 是一个由很多 Spout 和 Bolt 组成的网络，网络上的每条边代表了一个 Bolt 订阅的数据流，这些数据流包括从 Spout 或从 Bolt 输出的数据流。一个 Topology 实际上就是一个任意复杂的多级流计算过程。当 Topology 在服务器上部署完之后，它就会一直运行下去，直到用户禁止相应的进程。

Storm 有一个“本地模式”，用户可以在进程里面模拟一个 Storm 集群，然后进行类似实际集群上的开发工作。这种模式对于开发和测试十分有用。当用户准备好在一个真正的集群上提交 Topology 执行的时候，可以使用 Storm 命令行方便地从客户端提交一个 Topology 到集群上运行。

1.4.3 可扩展的

Storm 的 Topology 是并行计算的，它运行在一个集群主机上面。可以对 Topology 的不同部分单独调整它们的并行扩展。Storm 命令行客户端的 rebalance 命令可以调整并行运行的 Topology。

Storm 内在的并行度意味着它能以低延迟速度来处理高吞吐量的消息。根据 Storm 官方网站的资料介绍，Storm 的一个节点（Intel E5645@2.4Ghz 的 CPU，24 GB 的内存）在 1 秒钟能够处理 100 万个 100 字节的消息。

1.4.4 容错的

Storm 是容错的。当 Worker 死亡，Storm 会自动重新启动它们。如果一个节点死亡，Worker 将在另一个节点上重新启动。

Storm 的守护进程 Nimbus 和 Supervisor 被设计为无状态和快速失败的，所以如果它们死亡，它们会重启，就像什么都没发生过。这意味着可以使用 kill -9 命令来强制杀死 Storm 守护进程而不影响集群或者 Topology 的健康。

1.4.5　保证数据处理

Storm 保证每一个 Tuple 都能被完全处理。Storm 的核心机制之一是能够通过一种非常有效的 Topology 方式来跟踪 Tuple。

Storm 保证每个消息至少能得到一次完整处理。任务失败时，它会负责从消息源重试消息。

使用 Trident（一种在 Storm 的基本抽象上更高层次的抽象），用户可以实现一次且仅一次处理的语义。

1.4.6　可以使用任何语言

Storm 可与任何编程语言一起使用。Storm 的核心是一个定义和提交 Topology 的 Thrift 定义。因为 Thrift 可以用于任何语言，所以 Topology 可以用任何语言进行定义并提交。

同样，用户可以使用任何语言来定义 Spout 和 Bolt。非 JVM 的 Spout 和 Bolt 可通过一个基于 JSON 协议的 stdin/stdout 与 Storm 通信。目前有 Ruby、Python、Javascript、Perl 和 PHP 实现这个协议的适配器。

Storm-starter 有一个使用 Python 实现一个 Bolt 的 Topology 的例子。

1.4.7　部署和操作简单

Storm 集群是易于部署的，仅需要少量的安装和配置就可以运行，Storm 的“开箱即用”配置十分适合生产环境。

如果用户在 EC2 环境上开展业务，通过 storm-deploy 项目提供的简单方式就可以完成准备、配置和安装一个 Storm 集群的工作——用户仅仅需要点击一个按钮！

此外，Storm 部署之后也很容易操作。现在 Storm 被设计得极为健壮，使用 Storm 的集群会日复一日持续地稳定运行。

1.4.8　自由开源

Storm 是 Eclipse Public License 许可证下的一个免费开放源码项目，目前被托管在 GitHub 上。Eclipse Public License（EPL）是一个非常宽容的许可，允许用户使用 Storm 用于开源的或者专有的目的。

Storm 项目使用的是 Eclipse Public License 1.0 授权，官方原文如下：

> The use and distribution terms for this software are covered by the Eclipse Public License 1.0（http://opensource.org/licenses/eclipse-1.0.php） which can be found in the file LICENSE.html at the root of this distribution. By using this software in any fashion，you are agreeing to be bound by the terms of this license. You must not remove this notice， or any other，from this software.

Storm 有巨大的并且日益增长的生态系统库，以及与 Storm 协同使用的工具。storm-contrib 项目是一个为社区贡献的中央知识库，storm-contrib 内所有的内容都是在 Eclipse Public License

许可证下的。当然，许多社区成员也会选择 storm-contrib 外的 Storm 相关的项目。

1.5 Storm 的应用现状和发展趋势

Storm 作为最成功的实时大数据处理工具之一，在实时处理海量数据的领域中得到了充分的应用，以下作简单的介绍，使用 Storm 的公司及其相关项目的更多内容可参考本书的第 20 章。

1.5.1 应用现状

1. Storm 应用方向

目前 Storm 主要应用在以下这 3 个方面。

（1）流处理（Stream Processing）

Storm 最基本的用例是“流处理”。Storm 可以用来处理源源不断流进来的消息，处理之后将结果写入到某个存储中去。

（2）连续计算（Continuous Computation）

Storm 的另一个典型用例是“连续计算”。Storm 能保证计算可以永远运行，直到用户结束计算进程为止。

（3）分布式 RPC（Distributed RPC）

Storm 的第三个典型用例是“分布式 RPC”，由于 Storm 的处理组件是分布式的，而且处理延迟极低，所以可以作为通用的分布式 RPC 框架来使用。当然，其实搜索引擎本身也是一个分布式 RPC 系统。

2. 使用 Storm 的企业和组织

自 2011 年 8 月 Twitter 将 Storm 正式开源以来，Storm 在全球众多互联网企业得到了广泛的应用。表 1.1 是使用 Storm 的主要企业和组织。

表 1.1 使用 Storm 的主要企业和组织

企业/项目名称	简单介绍
Twitter	社交网络及微博客服务的网站，利用无线网络、有线网络等通信技术进行即时通信，是微博的典型应用。 Twitter 正式收购了 BackType 公司，并在不久后将 Storm 开源。Storm 大大提升了 Twitter 的众多系统的性能，包括探索发现、实时分析、个性化、搜索、收入优化等

（续表）

企业/项目名称	简单介绍
阿里巴巴集团	阿里巴巴集团旗下的企业包括企业对企业的网上贸易市场平台阿里巴巴，亚太最大的网络零售商圈淘宝网，国内领先的独立第三方支付平台支付宝公司。 阿里巴巴是国内最早采用 Storm 技术的企业，也为 Storm 的发展和开源做出了极大的贡献。阿里巴巴使用 Storm 来处理应用程序日志和数据库中的数据变化，搭建了基于 Storm 的实时计算架构，为数据应用程序提供实时统计
Groupon	国外知名团购公司。Groupon 使用 Storm 构建实时数据集成系统
The Weather Channel	公司成立于 1982 年，是美国一家基本有线和卫星电线节目提供商，向公众提供世界各地天气的实时信息。公司利用多个 Storm Topology 来采集并持久化天气数据
FullContact	公司旨在解决世界联系信息的难题，使用 Storm 作为系统的骨干支柱，用以将第三方提供的服务与公司的云通信录同步及向公司的联系图分析和联邦名片搜索系统提供实时的支持
雅虎	雅虎公司是美国著名的互联网门户网站，利用 Storm 和 Hadoop 开发一个支持大数据的融合和低延迟处理的下一代平台

3. Storm 的相关项目

- storm-website 项目

这是 storm-project.net 网站的源代码，使用 Jekyll（https://github.com/mojombo/jekyll）编写。storm-website 项目的网址为：https://github.com/nathanmarz/storm-website。

- storm-starter 项目

storm-starter 项目的网址为：https://github.com/nathanmarz/storm-starter。

storm-starter 项目包含了大量使用 Storm 的例子。

如果读者是第一次接触 Storm，建议首先了解以下 3 个主要的 Storm 的 Topology。

  - ExclamationTopology：全由 Java 编写的基本 Topology;
  - WordCountTopology：使用 Python 实现 Bolt 的使用 multilang 的基本 Topology;
  - ReachTopology：在 Storm 上面的复杂的 DRPC 例子。

- storm-deploy 项目

storm-deploy 项目的网址为：https://github.com/nathanmarz/storm-deploy

如果读者在 AWS 上使用 Storm，那么可以考虑使用 storm-deploy 项目。storm-deploy 项目使得我们能够非常简单地在 AWS 上部署 Storm 集群。storm-deploy 在 EC2 上完全自动化地供给、配置和安装 Storm 集群，它还为你设置 Ganglia，你可以使用 Ganglia 监视 CPU、磁盘和网络。

- storm-contrib 项目

storm-contrib 项目的网址为：https://github.com/nathanmarz/storm-contrib

storm-contrib 是一个使用 Storm 模块的社区库，其中包含了大量的集成了 Redis、Kafka、MongoDB 等其他系统的 Spout/Bolt，以及 Storm 开发者遇到的常见任务的代码。

storm-contrib 是被组织成每个模块一个子文件夹的“超级项目”，每个模块是分别独立的，模块所有者负责分配。

一些 storm-contrib 模块是 git 子模块（链接到外部的 github 库）。这允许 storm-contrib 子项

目可以被外部维护（所以那些项目可以保持独立的分支和标签），但也包含在 storm-contrib 项目中，以便增加社区可见度。

storm-contrib 内所有的内容都是在 Eclipse Public License 许可证下的。storm-contrib 包括：

- Spouts：Spouts 集成 JMS、Kafka、Redis pub/sub 等消息队列系统。
- storm-state：storm-state 使得用户在其计算过程中能以一种可靠的方式来很容易地管理大量的内存状态——通过持久地使用一个分布式文件系统。
- 数据库集成：有许多 Bolt 帮助集成各种数据库，如 MongoDB、关系数据库、Cassandra 等。
- 其他繁杂的公用组件。

- storm-mesos 项目

storm-mesos 项目在 Mesos 集群资源管理器上运行 Storm。Storm 与 Mesos 集群资源管理器整合在一起，这个项目已用于 Twitter 产品。

用户可以像将 Topology 提交给正常的 Storm 集群的方式来将 Topology 提交给 Storm/Mesos。

Storm/Mesos 提供了在 Topology 之间进行隔离的技术。通过这种技术，用户无需为 Topology 之间可能的干扰而担心。

- storm-yarn 项目

storm-yarn 项目的网址为：https://github.com/yahoo/storm-yarn。

storm-yarn 是 Yahoo！公司的开源项目，使 Storm 集群可以被部署到 Hadoop YARN 管理的主机上，这仍然是进行中的工作。

本项目的贡献者有：Andy Feng （@anfeng）、Robert Evans （@revans2）、Derek Dagit（@d2r）、Nathan Roberts （@ynroberts）。

- 其他 storm 相关的项目

可以在 GitHub 的官方网站（https://github.com/）搜索 Storm 相关的开源项目。

1.5.2 发展趋势

1. Storm 技术发展情况

2012 年，Storm 各系列的版本已经在各大公司得到广泛使用；2013 年 12 月，Storm 0.9.0 已经成功发布，当前最新版本为 0.9.0.1。现已有不少的企业组织或个人开始使用这一新的版本。但目前主要使用的版本为 0.8 版本。

2012 年 8 月发布的 0.8 版本中引入了 State，使得其从一个纯计算框架演变为一个包含存储和计算的实时计算利器。

另外 0.8 版本也引入了 Trident——一个高级抽象实时计算系统，这是实时计算领域的巨大跨越。Trident 提供更加友好的接口，同时可定制 Scheduler 的特性也为其针对不同的应用场景做优化提供了更便利的手段，也有人已经在基于 Storm 的实时 QL（Query Language）上迈出了一步。

在服务化方面，Storm 一直在朝着融入 Mesos 框架的方向努力。同时，Storm 也在实现细节上不断地优化，使用很多优秀的开源产品以进行产品功能的扩展，包括 Kryo、Disruptor 和 Curator 等。

Storm 集成了许多库，支持包括 Kestrel、Kafka、JMS、Cassandra、Memcached 以及更多系统。随着支持的库越来越多，Storm 更容易与现有的系统协作。

Storm 的性能提升超过 10 倍。经在 Twitter 内部集群测试，单节点每秒完成 100 万条信息。

Storm 团队正在积极地为推动 Storm 而努力。目前，团队正在开发一项监测功能，用于观察用户拓扑网络中的实时状态。Storm 团队的另外一项庞大的计划用于提高 Trident 的性能，集成更多的数据存储输入源。

2. Storm 研究人员发展情况

Storm 开源已经两年有余，在过去的两年中得到了飞速的发展，2012 年就已经有 27 个公司宣布正在产品中使用 Storm，现在使用 Storm 的公司的数量还在继续增加。2012 年，Storm 的邮件列表已有 1300 多个成员，每月超过 500 条信息；在 GitHub 上超过 4000 个项目负责人；同期 Twitter@stormprocessor 上的粉丝超过 1200 个；在旧金山，超过 230 名会员定期参加讨论活动。目前，世界各地有各种各样针对 Storm 的讨论活动和兴趣小组；在中国也有很多 Storm 相关的学习、讨论网站和社交兴趣组。

目前，除项目主管 Nathan Marz，项目核心贡献者徐明明、P. Taylor Goetz、Mike Heffner、Jason Jackson、Andy Feng 外，全球共有 41 名代码贡献者。有关 Storm 的图书已经出版，Amazon 网站上的英文版 Storm 书籍已有十多本（其中包括 O'Reilly 出版的 Storm 图书），但目前尚未有中文版 Storm 书籍面世。

3. Storm 版本发展情况

Storm 0.7 和 Storm 0.8 系列的版本已在各大公司得到了广泛使用。目前最新的版本是 Storm0.9.0.1。

2013 年 12 月，Storm 0.9.0 成功发布。这个版本较之前版本新增的功能如下。

- 消息通知机制变更：引入了使用纯 Java 语言编写的 Netty 作为我们的传输层，也就是说消息通知机制支持 Netty。这一特性是这个版本最大的变化，也使得 Storm 具有更好的跨平台能力。
- UI 变更：支持 Topology 控制命令，支持查看 Nimbus 和 Topology 的配置值，支持 UI 上查看各个 work 的日志。
- 安全性增强：Storm 的安全保障能力较弱，此版本在这方面有了一定的加强——提供了 API 用来实现可插拔的 Tuple 序列化，并且有一个基于 BlowFish 的加密算法用于加密 Tuple 的实现。

Storm 在实现细节上不断地优化，使用很多优秀的开源产品，包括 Kryo、Disruptor 和 Curator 等。

可以想像，当 Storm 发展到 1.0 版本时，一定是一款无比杰出的产品，让我们拭目以待。

1.6 如何学习 Storm

Storm 官方建议的学习 Storm 的过程如下。

1. Storm 的基本知识

- Javadoc 文档
- 概念
- 配置
- 保证消息处理（Guaranteeing message processing）
- 容错（Fault-tolerance）
- 命令行客户端（Command line client）
- 理解 Storm 拓扑的并行度
- 常见问题解答 FAQ

2. Trident

Trident 是 Storm 的高层抽象，它提供了仅一次处理，“事务性”数据存储持久性和共同流集合的分析操作。

3. 安装和部署

- 安装 Storm 集群
- 本地模式
- 故障排除
- 在生产集群运行拓扑
- 使用 Maven 或 Leiningen 创建 Storm

4. Storm 的进阶知识

- 序列化（Serialization）
- 常见的模式（Common patterns）
- Clojure DSL（Clojure DSL）
- 使用非 JVM 语言与风暴（Using non-JVM languages with Storm）
- 分布式 RPC（Distributed RPC）
- 事务拓扑（Transactional topologies）
- Kestrel 与 Storm（Kestrel and Storm）
- 直接分组（Direct groupings）

- 钩子（Hooks）
- 衡量指标（Metrics）
- Trident 元组的生命周期（Lifecycle of a trident tuple）

5. Storm 的高阶知识

- 为 Storm 定义一个非 JVM 语言的 DSL
- 多语言协议（Multilang protocol）：如何为另一种语言提供支持
- 实现文档（Implementation docs）

1.7　本书的章节安排及学习建议

1.7.1　本书的章节安排

本书一共 20 章，其中“基础知识”6 章，“安装与部署”4 章，“研发与维护”4 章，“进阶知识”5 章，“企业应用”1 章。本书的章节安排如表 1.2 所示。

表 1.2　本书的章节安排

所属模块	包含章节
基础知识	Storm 简介 Storm 的基础知识 拓扑详解 组件详解 Spout 详解 Bolt 详解
安装与部署	Zookeeper 详解 基础软件的安装与使用 Storm 的安装与配置 Storm 集群搭建实践
研发与维护	准备 Storm 的开发环境 开发自己的 Storm 应用 storm-starter 详解 研发与集群管理技巧
进阶知识	DRCP 详解 事务拓扑详解 Trident 详解 Storm 的内部实现 Storm 相关的其他项目
企业应用	企业应用实例

1.7.2 关于如何阅读本书的建议

1. 为了调研 Storm

建议优先阅读本书的首末两章，即“第 1 章　Storm 简介”和“第 20 章　企业应用案例”，再阅读“第 2 章　Storm 的基本知识”，也可以阅读完第 1 章到第 6 章“基础知识”模块的全部 6 个章节，最后阅读本书的其他章节。

2. 为了学习 Storm 的理论基础

建议优先阅读第 1 章到第 6 章“基础知识”模块的全部 6 个章节，然后阅读第 15 章到第 19 章“进阶知识”模块的全部 5 个章节，最后阅读本书的其他章节。

3. 为了学习 Storm 的应用实践

建议优先阅读第 7 章到第 10 章“安装与部署”模块的全部 4 个章节，然后阅读第 11 章到第 14 章“研发与维护”模块的全部 4 个章节，最后阅读本书的其他章节。

在开始相关章节的学习之前，请读者预先准备安装好如下的软硬件环境。

（1）生产环境（集群环境）节点

建议使用 64 位的 CentOS 6.0 以上的英文版 Linux 操作系统。

（2）开发环境（试验环境）主机

32 位或者 64 位，Linux 或者 Windows 操作系统都可以，无特殊要求。

但是，为了更好地与集群环境进行交互，建议使用 64 位的 CentOS 6.0 以上的 Linux 操作系统。

1.8 本章小结

本章简单介绍了 Storm 的基本概念，然后介绍 Storm 的起源、特征和优势。同时介绍了 Storm 的项目小组以及从何处可以获得 Storm 技术支持、技术讨论，以便于读者找到 Storm 的学习和源代码参考的网站。之后，本章还重点介绍了 Storm 的应用方向、使用 Storm 的企业和组织，以及 Storm 相关的项目，例如 storm-starter、storm-deploy 和 storm-contrib 等。在本章末尾给出了学习 Storm 的官方建议、本书的章节安排及学习建议。通过本章的学习，读者将对 Storm 有一个基本的了解。

第 2 章

Storm的基本知识

本章将学习 Storm 的基本知识，包括 Storm 的基本概念、配置、序列化、容错机制、可靠性机制、消息机制、开发环境与生产环境、并行度和命令行客户端等内容。

2.1 概念

2.1.1 元组（Tuple）

元组（Tuple），是消息传递的基本单元，是一个命名的值列表，元组中的字段可以是任何类型的对象。Storm 使用元组作为其数据模型，元组支持所有的基本类型、字符串和字节数组作为字段值，只要实现类型的序列化接口就可以使用该类型的对象。元组本来应该是一个 key-value 的 Map，但是由于各个组件间传递的元组的字段名称已经事先定义好，所以只要按序把元组填入各个 value 即可，所以元组是一个 value 的 List。

拓扑的每个节点必须声明 emit（发射）的元组的输出字段。例如，下面的 DoubleAndTripleBolt 类在 declareOutputFields 方法中声明输出字段为["double", "triple"]二元组。

```
    public class DoubleAndTripleBolt extends BaseRichBolt {
        private OutputCollectorBase _collector;

        @Override
        public void prepare(Map conf, TopologyContext context, OutputCollectorBa
se collector) {
            _collector = collector;
        }

        @Override
        public void execute(Tuple input) {
            int val = input.getInteger(0);
            _collector.emit(input, new Values(val*2, val*3));
            _collector.ack(input);
        }

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

首先，在 declareOutputFields 方法中声明了输出为["double", "triple"]的二元组。接着，在 prepare 方法中把 OutputCollectorBase 对象赋给私有成员变量_collector。最后，使用_collector

发射元组，输出元组是一个二元组，第 1 个值为原值的 2 倍，第 2 个值为原值的 3 倍，并使用 _collector 的 ack 方法，保证消息处理，提供可靠性机制。

2.1.2　流（Stream）

流（Stream）是 Storm 的核心抽象，是一个无界的元组序列。源源不断传递的元组就组成了流，在分布式环境中并行地进行创建和处理。

流被定义成一个为流中元组字段进行命名的模式，默认情况下，元组可以包含整形（integer）、长整形（long）、短整形（short）、字节（byte）、字符（string）、双精度数（double）、浮点数（float）、布尔值（boolean）和字节数组（byte array），还可以自定义序列化器，以便本地元组可以使用自定义类型。

流由元组组成，使用 OutputFieldsDeclarer 声明流及其模式。Serialization 是 Storm 的动态元组类型的信息，声明自定义序列化。自定义序列化必须实现 ISerialization 接口，自定义序列化可以注册使用 CONFIG.TOPOLOGY_SERIALIZATIONS 这个配置。

Storm 提供可靠的方式把原语转换成一个新的分布式的流，执行流转换的基本元素是 Spout 和 Bolt。

Spout 是流的源头，通常从外部数据源读取元组，并 emit 到拓扑中。例如，Spout 从 Kestrel 队列中读取元组，并作为一个流提交到拓扑。

Bolt 接收任何数量的输入流，执行处理后，可能提交新的流。复杂流的转换，如从 tweets 流中计算一个热门话题，需要多个步骤，因此需要多个 Bolt。Bolt 可以执行运行函数、过滤元组、连接流和连接数据库等任何事情。

如图 2.1 所示，Spout 和 Bolt 的网络被打包成一个“拓扑”，即顶级抽象，之后提交这个拓扑到 Storm 集群中执行。拓扑是一个图的流转换，节点表示 Spout 或 Bolt，弧边指示哪些 Bolt 订阅了该流。当一个 Spout 或 Bolt 发射一个元组到一个流时，它会发射元组到每一个订阅该流的 Bolt。

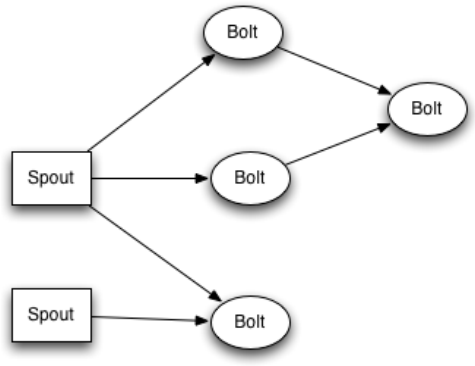


图 2.1　拓扑示例

拓扑节点之间的链接表示元组应该如何传递。例如，如果从 Spout A 到 Bolt B 有一个链接，

从 Spout A 到 Bolt C 有一个链接，从 Bolt B 到 Bolt C 有一个链接，那么每次 Spout A 将同时发射元组到 Bolt B 与 Bolt C，Bolt B 也会发射元组到 Bolt C。

一个 Storm 拓扑中的每个节点均可以并行执行，也可以指定拓扑中跨集群执行的线程数量。

一个拓扑会一直运行，直到你杀死它。Storm 会自动重新分配任何失败的任务。此外，即使主机已经关闭，消息已经被删除，Storm 也能保证不会有数据丢失。

每个流被声明后都会赋予一个 id。因为单个流的 Spout 和 Bolt 是如此普遍，OutputFieldsDeclarer 有方便的方法来声明一个不指定 id 的单流。在这种情况下，流被赋予默认的 id 值。

2.1.3 喷口（Spout）

喷口（Spout）是拓扑的流的来源，是一个拓扑中产生源数据流的组件。通常情况下，Spout 会从外部数据源（例如 Kestrel 队列或 Twitter API）中读取数据，然后转换为拓扑内部的源数据。Spout 可以是可靠的，也可以是不可靠的。如果 Storm 处理元组失败，可靠的 Spout 能够重新发射，而不可靠的 Spout 就尽快忘记发出的元组。Spout 是一个主动的角色，其接口中有个 nextTuple()函数，Storm 框架会不停地调用此函数，用户只要在其中生成源数据即可。

Spout 可以发出超过一个流。为此，使用 OutputFieldsDeclarer 类的 declareStream 方法来声明多个流，使用 SpoutOutputCollector 类的 emit 执行流的提交。

Spout 的主要方法是 nextTuple()。nextTuple()会发出一个新 Tuple 到拓扑，如果没有新的元组发出则简单地返回。nextTuple()方法不阻止任何 Spout 的实现，因为 Storm 在同一线程调用所有的 Spout 方法。

Spout 的其他主要方法是 ack()和 fail()。当 Storm 检测到一个元组从 Spout 发出时，ack()和 fail()会被调用，要么成功完成通过拓扑，要么未能完成。ack()和 fail()仅被可靠的 Spout 调用。

IRichSpout 是 Spout 必须实现的接口。

2.1.4 螺栓（Bolt）

在拓扑中的所有处理都在 Bolt 中完成，Bolt 是流的处理节点，从一个拓扑接收数据然后执行进行处理的组件。Bolt 可以完成过滤、业务处理、连接运算、连接与访问数据库等任何操作。Bolt 是一个被动的角色，其接口中有一个 execute()方法，在接收到消息后会调用此方法，用户可以在其中执行自己希望的操作。

Bolt 可以完成简单的流的转换，而完成复杂的流的转换通常需要多个步骤，因此需要多个 Bolt。例如，将 tweet 数据流转换为热门图片流至少需要两个步骤：首先一个 Bolt 为每个图像的转发做一个滚动的计数，然后一个或多个 Bolt 流找出排名前 X 的图片（也可以使用 3 个 Bolt 代替使用 2 个 Bolt，一个更具有伸缩性的方式，来做这种特殊的流转换）。

Bolt 可以发出超过一个的流。为此，使用 OutputFieldsDeclarer 类的 declareStream()方法声明多个流，并使用 OutputCollector 类的 emit()方法指定发射的流。

当声明一个 Bolt 的输入流时，总是订阅另一个组件的特定流。如果想订阅另一个组件的所有的流，必须订阅每一个过程。InputDeclarer()方法有语法以订阅声明默认流 id 的流：declarer.shuffleGrouping("1")是指订阅组件 1 的默认流，相当于 declarer.shuffleGrouping("1", DEFAULT_STREAM_ID)。

Bolt 的主要方法是 execute()方法，该方法将一个新的元组作为输入。Bolt 使用 OutputCollector 对象发射新的元组。Bolt 必须为它们处理的每个元组调用 OutputCollector 类的 ack()方法，以便 Storm 知道什么时候元组会完成（并最终确定它是安全应答原始的 Spout 的元组）。对于处理一个输入元组的常见情况是发射 0 个或者更多基于该元组的元组，然后应答输入元组（Storm 提供了一个自动应答的 IBasicBolt 接口）。

我们完全可以在 Bolt 中启动新的线程进行异步处理。OutputCollector 类是线程安全的，可以在任何时间被调用。

IRichBolt 是 Bolt 的通用接口。IBasicBolt 是定义 Bolt 进行过滤或简单功能的一个很方便的接口。使用 OutputCollector 类的实例可以让 Bolt 发送元组到其输出流。

2.1.5　拓扑（Topology）

拓扑（Topology）是 Storm 中运行的一个实时应用程序，因为各个组件间的消息流动而形成逻辑上的拓扑结构。

把实时应用程序的运行逻辑打成 jar 包后提交到 Storm 的拓扑（Topology）。Storm 的拓扑类似于 MapReduce 的作业（Job）。其主要的区别是，MapReduce 的作业最终会完成，而一个拓扑永远都在运行直到它被杀死。一个拓扑是一个图的 Spout 和 Bolt 的连接流分组。

在 Java 中，使用 TopologyBuilder 类来构造拓扑。

2.1.6　主控节点与工作节点

Storm 集群中有两类节点：主控节点（Master Node）和工作节点（Worker Node）。其中，主控节点只有一个，而工作节点可以有多个。

2.1.7　Nimbus 进程与 Supervisor 进程

主控节点运行一个称为 Nimbus 的守护进程，类似于 Hadoop 的 JobTracker。Nimbus 负责在集群中分发代码，对节点分配任务，并监视主机故障。

每个工作节点运行一个称为 Supervisor 的守护进程。Supervisor 监听其主机上已经分配的主机的作业，启动和停止 Nimbus 已经分配的工作进程。

2.1.8 流分组（Stream grouping）

流分组，是拓扑定义中的一部分，为每个 Bolt 指定应该接收哪个流作为输入。流分组定义了流/元组如何在 Bolt 的任务之间进行分发。

Storm 内置了 7 种流分组方式，通过实现 CustomStreamGrouping 接口可以实现自定义的流分组。

2.1.9 工作进程（Worker）

Worker 是 Spout/Bolt 中运行具体处理逻辑的进程。拓扑跨一个或多个 Worker 进程执行。每个 Worker 进程是一个物理的 JVM 和拓扑执行所有任务的一个子集。例如，如果合并并行度的拓扑是 300，已经分配 50 个 Worker，然后每个 Worker 将执行 6 个任务（作为在 Worker 内的线程）。Storm 尝试在所有 Worker 上均匀地发布任务。

可使用 Config.TOPOLOGY_WORKERS 配置项设置执行拓扑时分配的 Worker 的数量。

2.1.10 任务（Task）

Worker 中每一个 Spout/Bolt 的线程称为一个任务（Task）。每个 Spout 或者 Bolt 在集群中执行许多任务。每个任务对应一个线程的执行，流分组定义如何从一个任务集到另一个任务集发射元组。可以通过 TopologyBuilder 类的 setSpout()方法和 setBolt()方法来设置每个 Spout 或者 Bolt 的并行度（parallelism）。

2.1.11 执行器（Executor）

在 Storm 0.8 以后，Task 不再与物理线程对应，同一个 Spout/Bolt 的 Task 可能会共享一个物理线程，该线程称为执行器（Executor）。

2.1.12 可靠性（Reliability）

Storm 保证每一个 Spout 元组将被拓扑完全可靠地处理。它跟踪每个 Spout 元组的元组树，检测树中的元组什么时候可以成功完成。每个拓扑都有“消息超时时间”，如果 Storm 在超时之前未能检测到 Spout 元组已经完成，那么会设元组为失败并在之后重新发射它。

要利用 Storm 的可靠性优势，当元组树的新边缘被创建时必须通知 Storm，并告之 Storm 什么时候处理完该元组。这些都是通过使用 OutputCollector 对象实现的，Bolt 使用 OutputCollector 对象发射元组。在 emit()方法中完成 Anchoring，声明已经使用 ack()方法完成一个元组。

2.2　Storm 的配置

2.2.1　Storm 的配置类型

Storm 有大量的配置，可以调整 Nimbus、Supervisor、拓扑运行的参数，其中有些配置是不能修改的系统配置，而其他配置是可以修改的。

每个配置会有一个默认值，该值定义在 Storm 代码库的 defaults.yaml 文件中。在 Nimbus 和 Supervisor 的类路径中定义一个 storm.yaml 文件，可以覆盖这些配置值。使用 StormSubmitter 提交拓扑的时候，可以定义一个指定拓扑的配置，但是只能覆盖前缀为 TOPOLOGY 的配置项。

Storm 0.7.0 以后的版本开始允许在 Spout/Bolt 中覆盖配置，可以修改的配置主要有：

- "topology.debug"。
- "topology.max.spout.pending"。
- "topology.max.task.parallelism"。
- "topology.kryo.register"。

topology.kryo.register 与其他的配置有所不同，它的序列化会应用到拓扑上的所有组件。

Storm 的 Java API 也提供了两种方式指定组件的配置。

- 内部的（Internally）

在 Spout 或者 Bolt 类中，覆盖 getComponentConfiguration 方法，返回组件配置的 Map 对象。getComponentConfiguration 方法定义如下：

```
Map<String, Object> getComponentConfiguration()
```

- 外部的（Externally）

使用 TopologyBuilder 类的 setSpout 方法返回 SpoutDeclarer 对象，使用 setBolt 方法返回 BoltDeclarer 对象。SpoutDeclarer 与 BoltDeclarer 实现了 ComponentConfigurationDeclarer 接口，该接口有 addConfiguration 方法和 addConfigurations 方法，可以通过调用这两个方法来覆盖组件的配置。

SpoutDeclarer 接口的定义代码如下：

```
public interface SpoutDeclarer extends
ComponentConfigurationDeclarer<SpoutDeclarer> {

}
```

BoltDeclarer 接口的定义代码如下：

```
public interface BoltDeclarer extends InputDeclarer<BoltDeclarer>,
        ComponentConfigurationDeclarer<BoltDeclarer> {
```

```
}
```

ComponentConfigurationDeclarer 接口的定义代码如下：

```
public interface ComponentConfigurationDeclarer<T
extends ComponentConfigurationDeclarer> {
    T addConfigurations(Map conf);
    T addConfiguration(String config, Object value);
    T setDebug(boolean debug);
    T setMaxTaskParallelism(Number val);
    T setMaxSpoutPending(Number val);
    T setNumTasks(Number val);
}
```

Storm 配置值的优先顺序为：

```
defaults.yaml < storm.yaml < 特定拓扑的配置 < 内部特定组件的配置 < 外部特定组件的配置
```

2.2.2 defaults.yaml 文件

在 Storm 的源代码中，文件夹 conf 下有一个名为 defaults.yaml 的文件，即 conf/defaults.yaml，defaults.yaml 文件记录着所有配置的默认值。

defaults.yaml 文件的详细清单如下：

```
############ defaults.yaml 文件存储的是默认值
############可以在 storm.yaml 文件中写入额外的配置

java.library.path: "/usr/local/lib:/opt/local/lib:/usr/lib"

### storm.* 配置是普通的配置
# storm.local.dir 是保存 jar 包的目录
storm.local.dir: "storm-local"
storm.zookeeper.servers:
    - "localhost"
storm.zookeeper.port: 2181
storm.zookeeper.root: "/storm"
storm.zookeeper.session.timeout: 20000
storm.zookeeper.connection.timeout: 15000
storm.zookeeper.retry.times: 5
storm.zookeeper.retry.interval: 1000
storm.zookeeper.retry.intervalceiling.millis: 30000
storm.cluster.mode: "distributed" # can be distributed or local
storm.local.mode.zmq: false
storm.thrift.transport: "backtype.storm.security.auth.SimpleTransportPlugin"
storm.messaging.transport: "backtype.storm.messaging.netty.Context"

### nimbus.* 配置应用于主节点
```

```
nimbus.host: "localhost"
nimbus.thrift.port: 6627
nimbus.thrift.max_buffer_size: 1048576
nimbus.childopts: "-Xmx1024m"
nimbus.task.timeout.secs: 30
nimbus.supervisor.timeout.secs: 60
nimbus.monitor.freq.secs: 10
nimbus.cleanup.inbox.freq.secs: 600
nimbus.inbox.jar.expiration.secs: 3600
nimbus.task.launch.secs: 120
nimbus.reassign: true
nimbus.file.copy.expiration.secs: 600
nimbus.topology.validator: "backtype.storm.nimbus.DefaultTopologyValidator"

### ui.* 配置应用于主节点
ui.port: 8080
ui.childopts: "-Xmx768m"

logviewer.port: 8000
logviewer.childopts: "-Xmx128m"
logviewer.appender.name: "A1"


drpc.port: 3772
drpc.worker.threads: 64
drpc.queue.size: 128
drpc.invocations.port: 3773
drpc.request.timeout.secs: 600
drpc.childopts: "-Xmx768m"

transactional.zookeeper.root: "/transactional"
transactional.zookeeper.servers: null
transactional.zookeeper.port: null

### supervisor.* 配置应用于 Supervisor 节点
# 定义可以在本机上运行的工作进程的数量，每个工作进程分配一个通信端口
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
supervisor.childopts: "-Xmx256m"
# Supervisor 等待多长时间确保工作进程已经启动
supervisor.worker.start.timeout.secs: 120
# Supervisor 认为工作进程死亡并试图进行重启的心跳时间的大小
supervisor.worker.timeout.secs: 30
# Supervisor 检查工作进程的状态的频率，Supervisor 会在必要时重启工作进程
```

```
    supervisor.monitor.frequency.secs: 3
    # Supervisor 到 Nimbus 的心跳的频率
    supervisor.heartbeat.frequency.secs: 5
    supervisor.enable: true

    ### worker.* 配置应用于工作进程的任务
    worker.childopts: "-Xmx768m"
    worker.heartbeat.frequency.secs: 1

    task.heartbeat.frequency.secs: 3
    task.refresh.poll.secs: 10

    zmq.threads: 1
    zmq.linger.millis: 5000
    zmq.hwm: 0

    storm.messaging.netty.server_worker_threads: 1
    storm.messaging.netty.client_worker_threads: 1
    storm.messaging.netty.buffer_size: 5242880 #5MB 缓存
    storm.messaging.netty.max_retries: 30
    storm.messaging.netty.max_wait_ms: 1000
    storm.messaging.netty.min_wait_ms: 100

    ### topology.* 配置应用于具体执行的 Storm
    topology.enable.message.timeouts: true
    topology.debug: false
    topology.optimize: true
    topology.workers: 1
    topology.acker.executors: null
    topology.tasks: null
    # 消息超时时间，如果超时，消息被认为是失败的
    topology.message.timeout.secs: 30
    topology.skip.missing.kryo.registrations: false
    topology.max.task.parallelism: null
    topology.max.spout.pending: null
    topology.state.synchronization.timeout.secs: 60
    topology.stats.sample.rate: 0.05
    topology.builtin.metrics.bucket.size.secs: 60
    topology.fall.back.on.java.serialization: true
    topology.worker.childopts: null
    topology.executor.receive.buffer.size: 1024
    topology.executor.send.buffer.size: 1024
    topology.receiver.buffer.size: 8 #设置过高会导致很多问题，如心跳线程饿死、吞吐量大幅
下跌
    topology.transfer.buffer.size: 1024
    topology.tick.tuple.freq.secs: null
    topology.worker.shared.thread.pool.size: 4
```

```
    topology.disruptor.wait.strategy: "com.lmax.disruptor.BlockingWaitStrategy"
    topology.spout.wait.strategy: "backtype.storm.spout.SleepSpoutWaitStrategy"
    topology.sleep.spout.wait.strategy.time.ms: 1
    topology.error.throttle.interval.secs: 10
    topology.max.error.report.per.interval: 5
    topology.kryo.factory: "backtype.storm.serialization.DefaultKryoFactory"
    topology.tuple.serializer: "backtype.storm.serialization.types.ListDelegate
Serializer"
    topology.trident.batch.emit.interval.millis: 500

    dev.zookeeper.path: "/tmp/dev-storm-zookeeper"
```

2.2.3　storm.yaml 文件

修改 STORM_HOME 目录下的 conf/storm.yaml 文件，可以覆盖 Storm 配置的默认值。storm.yaml 会覆盖 defaults.yaml 中的任何内容。以下几个配置是必须要设置的。

（1）storm.zookeeper.servers

这是一个为 Storm 集群服务的 ZooKeeper 集群的主机列表，它的配置应该类似于：

```
    storm.zookeeper.servers:
    - "111.222.333.444"
    - "555.666.777.888"
```

如果你的 ZooKeeper 集群使用的端口和默认的端口不相同，应该也设置“storm.zookeeper.port”的值。

（2）storm.local.dir

Nimbus 和 Supervisor 守护进程需要在本地硬盘的一个目录存储少量的状态（如 jars、confs 等）。你应该在每台主机上创建该目录，赋予它适当的权限，然后使用该配置填写目录位置。例如：

```
    storm.local.dir: "/mnt/storm"
```

（3）java.library.path

这是加载 Storm 使用的本地库的路径，例如 ZeroMQ 库和 JZMQ 库。默认的 /usr/local/lib:/opt/local/lib:/usr/lib 适用于大多数安装，一般不需要设置此配置。

（4）nimbus.host

工作节点为了下载拓扑的 jar 和 confs 文件，需要知道哪些主机是主控节点。例如：

```
    nimbus.host: "111.222.333.44"
```

（5）supervisor.slots.ports

对于每个工作节点，可以通过该配置项来设置该节点上运行多少个 Worker。每个 Worker 使用一个端口接收消息，此设置定义为使用哪些端口是打开的。如果在这里定义 5 个端口，然

后 Storm 将分配到 5 个 Worker 在这台主机上运行。如果定义 3 个端口，Storm 只会分配 3 个 Worker。此设置默认为在端口 6700、6701、6702 和 6703 上配置运行 4 个 Worker。例如：

```
supervisor.slots.ports:
- 6700
- 6701
- 6702
- 6703
```

2.2.4 Config 类

Config 类，即 backtype.storm.Config，是一个所有配置的清单，一个创建拓扑特定配置的 helper 类。

Config 类提供了方便的方法来创建一个拓扑配置 Map，为所有可以设置的配置项提供了 setter 方法，这也使得它很容易实现序列化。

Config 类还提供了 Storm 集群和 Storm 拓扑上所有可能的配置的常量，可以在 defaults.yaml 文件中找到默认值。

可以把其他的配置项添加到 Config 类中。Storm 会忽略任何它不能识别的配置项，但是拓扑可以在 Bolt 的 prepare()方法或者 Spout 的 open()方法中自由地使用这些配置项。

Config 类的继承关系如图 2.2 所示。

backtype.storm

Class Config

java.lang.Object
  └ java.util.AbstractMap<K,V>
    └ java.util.HashMap<java.lang.String,java.lang.Object>
      └ **backtype.storm.Config**

All Implemented Interfaces:
java.io.Serializable, java.lang.Cloneable, java.util.Map<java.lang.String,java.lang.Object>

图 2.2　Config 的继承关系图

Config 类的 setter 方法如下：

```
public void setDebug(boolean isOn)
public void setFallBackOnJavaSerialization(boolean fallback)
public void setKryoFactory(Class<? extends IKryoFactory> klass)
public void setMaxSpoutPending(int max)
public void setMaxTaskParallelism(int max)
public void setMessageTimeoutSecs(int secs)
public void setNumAckers(int numExecutors)
public void setNumWorkers(int workers)
public void setOptimize(boolean isOn)
public void setSkipMissingKryoRegistrations(boolean skip)
public void setStatsSampleRate(double rate)
```

2.3　序列化（Serialization）

本节将介绍 Storm 的序列化系统如何运行，适合 0.6.0 之后的版本。在 0.6.0 之前，Storm 使用不同的序列化系统。

元组可以包含任何类型的对象。因为 Storm 是一个分布式系统，它需要知道当元组在任务之间传递时如何序列化和反序列化对象。

Storm 使用 Kryo 序列化。Kryo 是一个灵活和快速的序列化库，产生很小的序列化。

默认情况下，Storm 可以序列化原始类型、字符串、字节数组、ArrayList、HashMap、HashSet 和 Clojure 集合类型。如果想在元组中使用另一种类型，则需要注册一个自定义序列化器。

2.3.1　动态类型

Storm 在一个元组中没有为字段声明类型。当把对象放置到字段，Storm 动态地找出它并序列化。在我们开始序列化接口前，先花一分钟了解为什么 Storm 的元组是动态类型的。

首先添加静态类型到元组字段会给 Storm 的 API 增加复杂性。例如，Hadoop 是静态类型的键和值，但在用户部分需要大量的注释。对于 Hadoop 的 API 使用这些是一种负担，“类型安全”不值得花费如此多的资源，而动态类型则简单并且容易使用。

进一步来说，以任何合理的方式使 Storm 的元组变成静态类型都是不可能的。假设一个 Bolt 订阅多个流，从所有这些流的元组可能有不同的类型穿过字段。当一个 Bolt 在 execute 方法中接收到一个元组，元组可能来自任何流，所以可能有任何类型的组合。可能你可以做一些反射原理，为 Bolt 订阅的每一个元组流声明不同的方法，但 Storm 选择动态类型是一种更简单和更直接的方法。

最后，使用动态类型的另一个原因是，Storm 可以被动态类型语言（如 Clojure 和 Jruby）以一种简单易懂的方式使用。

2.3.2　自定义序列化

正如前面所提到的，Storm 使用 Kryo 来序列化。要实现自定义序列化器，需要用 Kryo 注册新的序列化器。强烈推荐读者阅读 Kryo 主页以理解它是如何处理自定义序列化的。

添加自定义序列化器是通过拓扑配置的 topology.kryo.register 属性完成的。它需要一个注册的列表，每个注册项可以采取两种形式：

- 类名注册。在这种情况下，Storm 将使用 Kryo 的 FieldsSerializer 来序列化该类。这可能是也可能不是该类最好的选择——更多的细节可以查看 Kryo 文档。
- 实现了 com.esotericsoftware.kryo.Serializer 接口的类名注册的映射。

让我们看下面的例子。

```
topology.kryo.register:
- com.mycompany.CustomType1
- com.mycompany.CustomType2: com.mycompany.serializer.CustomType2Serializer
- com.mycompany.CustomType3
```

com.mycompany.CustomType1 和 com.mycompany.CustomType3 使用 FieldsSerializer 来进行序列化，而 com.mycompany.CustomType2 使用 com.mycompany.serializer.CustomType2Serializer 来进行序列化。

Storm 为拓扑配置里的注册序列化提供了帮助。Config 类中有一个名为 registerSerialization 的方法，可以把注册添加到配置中。

另外，还有一个 Config.TOPOLOGY_SKIP_MISSING_KRYO_REGISTRATIONS 的高级配置。如果设置为 true，Storm 会忽略任何已经注册但在类路径中没有其代码的序列化；否则，当 Storm 找不到一个序列化，它将抛出错误。如果你运行很多各有不同的序列化的拓扑集群，这是有用的，除非想在 storm.yaml 文件中声明所有的序列化。

2.3.3 Java 序列化

如果 Storm 遇到一种没有序列化注册的类型，如果可能的话，它将使用 Java 序列化。如果对象不能被 Java 序列化所处理，那么 Storm 将抛出一个错误。

注意，Java 序列化所耗资源是极其昂贵的，无论是从 CPU 成本还是序列化对象的大小来看。当你把拓扑放在生产上面使用时，强烈建议注册自定义序列化器。Java 序列化的行为，很容易原型化新的拓扑。

可以通过设置 Config.TOPOLOGY_FALL_BACK_ON_JAVA_SERIALIZATION 配置项为 false，关掉依靠 Java 序列化行为。

2.3.4 特定组件序列化注册

Storm 0.7.0 允许设置特定组件的配置。当然，如果一个组件定义了一个序列化，序列化需要对于其他 Bolt 可用，否则它们无法从这些组件接收到任何消息！

当一个拓扑被提交，单一的序列化集会被选择用于拓扑中的所有组件发送消息。这是通过合并特定组件的序列化器注册与常规的序列化注册集来完成的。如果两个组件对同一个类定义序列化器，其中一个序列化器将是任意选择的。

如果两个特定组件的注册有冲突，Storm 会强制对一个特定的类进行序列化。只需要在特定于拓扑配置定义你想要使用的序列化器。对于序列化注册，这个特定拓扑的配置优先于特定组件的配置。

2.4　容错机制

本节将介绍 Storm 作为一个容错系统的设计细节。

2.4.1　Worker 进程死亡

当一个 Worker（工作进程）死亡，Supervisor 会尝试重启它。如果它在启动时连续失败了一定的次数，无法发送心跳信息到 Nimbus，Nimbus 将在另一台主机上重新分配 Worker。

2.4.2　节点死亡

如果节点死亡，分配给该节点主机的任务会暂停，Nimbus 会把这些任务重新分配给其他节点主机。

2.4.3　Nimbus 或者 Supervisor 守护进程死亡

Nimbus 和 Supervisor 守护进程被设计成快速失败的（每当遇到任何意外的情况，进程自动毁灭）和无状态的（所有状态都保存在 ZooKeeper 或者磁盘上）。如创建 Storm 集群中所述，Nimbus 和 Supervisor 守护进程应该使用 daemontools 或者 monit 工具监控运行。所以，如果 Nimbus 或 Supervisor 守护进程死亡，它们重启并会像什么事情都没发生一样正常工作。

Nimbus 或者 Supervisor 的死亡不影响 Worker 进程的工作。这与 Hadoop 不同，在 Hadoop 中，如果 JobTracker 进程死亡，所有正在运行的 Job 都会丢失。

2.4.4　Nimbus 是否是“单点故障”的

如果失去了 Nimbus 节点，Worker 也会继续执行。另外，如果 Worker 死亡，Supervisor 也会继续重启它们。但是，没有 Nimbus，Worker 不会在必要时（例如，失去一个 Worker 的主机）被安排到其他主机。

所以 Nimbus“在某种程度上”是单点故障（Single Point Of Failure，SPOF）。在实践中，这不是一个大问题，因为 Nimbus 守护进程死亡，不会发生灾难性的问题。Storm 官方计划让 Nimbus 在未来实现高可用性。

2.5 可靠性机制——保证消息处理

Storm 保证每个来自 Spout 的消息将被完全地处理。

2.5.1 消息被“完全处理”的含义

如同蝴蝶效应一样，一个来自 Spout 的元组可以引发基于它所创建的数以千计的元组。例如，流的单词统计的拓扑定义如下：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentences", new KestrelSpout("kestrel.backtype.com",
                              22133, "sentence_queue",new StringScheme()));
builder.setBolt("split", new SplitSentence(), 10)
       .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 20)
       .fieldsGrouping("split", new Fields("word"));
```

此拓扑从一个 Kestrel 队列读取句子，分割句子为单词，然后对每个单词在 emit 之前进行计数。来自 Spout 的一个元组会触发很多基于它创建的元组：一个是句子中每个单词的元组，一个是为每个单词更新计数的元组。如图 2.3 所示为句子 the cow jumped over the moon 的元组树。

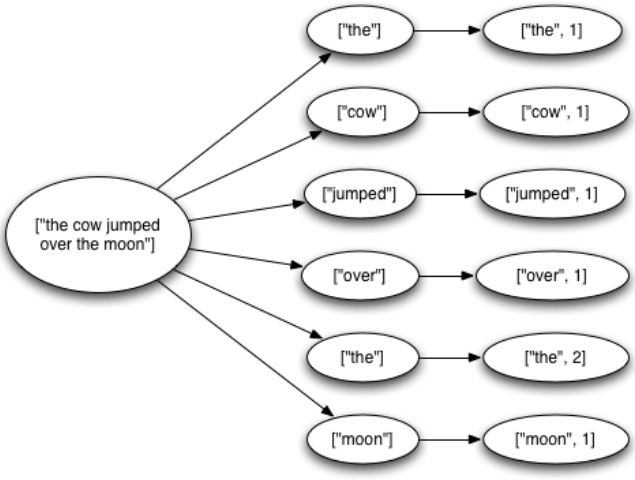


图 2.3 WordCount 的某个句子的元组树

当树创建完毕，并且树中的每一个消息都已经被处理时，Storm 认为来自 Spout 的元组是“完全处理”的。当一个元组的消息树在指定的超时范围内不能被完全处理，则元组被认为是失败的。超时的默认值为 30 秒，对于一个特定的拓扑，可以使用 Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS 来配置修改。

2.5.2　如果一个消息被完全处理或完全处理失败会发生什么

首先，让我们看看 Spout 的元组的生命周期。ISpout 接口的定义如下：

```
    public interface ISpout extends Serializable {
        void open(Map conf, TopologyContext context, SpoutOutputCollector colle-
ctor);
        void close();
        void nextTuple();
        void ack(Object msgId);
        void fail(Object msgId);
    }
```

首先，Storm 通过调用 Spout 的 nextTuple()方法从 Spout 请求一个元组。Spout 使用 open()方法提供的 SpoutOutputCollector 对象发射一个元组到它的输出流。当发射元组时，Spout 会提供一个“消息 id”，以便用来识别元组。例如，KestrelSpout 从 Kestrel 消息队列中读取一个消息时，会发射 Kestrel 提供的“消息 id”。下面发射一个消息到 SpoutOutputCollector 对象：

```
    _collector.emit(new Values("field1", "field2", 3) , msgId);
```

接下来，元组被发送到 Bolt，同时 Storm 负责跟踪创建的消息树。如果 Storm 检测到一个元组是完全处理的，Storm 将调用原 Spout 任务的 ack()方法，把 Spout 提供给 Storm 的消息 id 作为输入参数。同样，如果元组超时，Storm 将调用 Spout 的 fail()方法。注意，一个元组将由 Spout 任务来确认成功或失败，这个 Spout 任务是创建此元组的完全相同的 Spout 任务。如果一个 Spout 跨集群执行很多任务，元组是不会被创建它的那个任务外的其他任务确认成功或失败的。

当 KestrelSpout 从 Kestrel 队列取出一个消息时，它将“打开”消息。这意味着消息实际上还没有从队列取出，而是放在“待定”状态等待确认消息已经完成。当在“待定”状态时，一个消息将不会被发送到其他消费者队列。如果客户端断开连接，所有“待定”状态的消息会放回到队列。当消息被打开时，Kestrel 向客户端提供信息的数据以及消息的惟一 id。当发射元组到 SpoutOutputCollector 时，KestrelSpout 使用精确的 id 作为元组的“消息 id”。之后，当 KestrelSpout 的 ack()或者 fail()方法被调用时，KestrelSpout 会把一个 ack 或 fail 消息以及此消息的 id 一起发送到 Kestrel，以便从队列取出或恢复。

2.5.3　Storm 如何保证可靠性

作为受益于 Storm 可靠性功能的用户，有两件事情必须完成：一是当在树的元组上创建一个新链接时需要通知 Storm，二是处理完成一个单一元组时需要通知 Storm。通过这两件事情，当树上的元组被完全处理时 Storm 可以检测到，并且可以相应地对 Spout 的元组进行确认。Storm 的 API 提供了一个简洁的方法来完成这些任务。

在元组树中指定一个链接，此链接被称为锚定（Anchoring）。Anchoring 在发射一个新的元组的同一时间完成。让我们使用以下 Bolt 为例进行介绍，这个 Bolt 将包含一个句子的元组划分为一个包含每个单词的锚定：

```
    public class SplitSentence extends BaseRichBolt {
        OutputCollector _collector;

        public void prepare(Map conf, TopologyContext context, OutputCollector
collector) {
            _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 declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
    }
```

通过指定输入元组作为第一个参数来发射，每个单词元组被锚定（anchored）。因为这个单词元组是被锚定的，如果单词元组未能被下游处理，树的根的 Spout 元组将在稍后重发。相反，如果单词元组的发射操作如下，让我们看看会发生什么：

```
    _collector.emit(new Values(word));
```

这种方式发射的单词元组导致未被锚定（unanchored）。如果元组未被下游处理，根元组将不会重发。这取决于你需要的 Topology 的容错保证，有时候需要相应地发射一个未被锚定的元组。

一个输出元组可以被锚定到多个输入元组。在做流媒体连接或聚合时，这是有用的。一个复合锚定（multi-anchore）元组未能被处理将导致来自 Spout 的多个元组重发。复合锚定是通过指定一个元组列表而不仅仅是一个元组来完成的。例如：

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

```
_collector.emit(anchors, new Values(1, 2, 3));
```

复合锚定添加输出元组到多个元组树。请注意，对于复合锚定还可以打破树结构，创建元组的有向无环图，如图 2.4 所示。

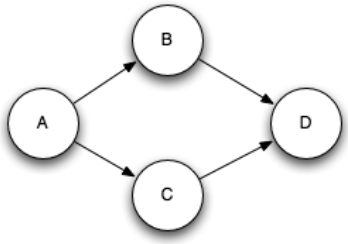


图 2.4　创建元组的有向无环图

Storm 的实现适用于有向无环图以及元组树。

锚定是你如何指定元组树。当完成处理元组树中的单个元组时，Storm 的可靠 API 将指定下一个元组和最后一个元组。这是通过使用 OutputCollector 类的 ack 和 fail 方法完成的。如果回顾 SplitSentence 示例，可以看到输入元组处于 acked 状态，毕竟这个单词元组已经被发射。

可以使用 OutputCollector 类的 fail()方法立即失败在元组树的根部的 Spout 的元组。例如，应用程序可以选择从数据库客户端捕获到一个异常，明确失败输入元组。由于没有明确的元组，Spout 元组回放的速度比等待 tuple 超时的速度更快。

你处理的每一个元组必须 ack 或者 fail。Storm 使用内存来追踪每个元组，所以如果不 ack/fail 每个元组，任务最终会耗尽内存。

很多 Bolt 遵循一个读取一个输入元组，发射元组，在 execute 方法确认元组的通用模式。这些 Bolt 具有类别过滤器和简单的功能。Storm 有一个接口称为 BasicBolt，为你封装这个模式。SplitSentence 的例子可以使用 BasicBolt 写成：

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

这个实现与之前的实现相比，语义上是相同的，但是更简单。元组发射到

BasicOutputCollectorare 是自动 Anchoring 到输入元组，当 execute 方法完成时，输入元组自动为你确认。

与此相反，做聚合或者连接操作的 Bolt 可能会延迟确认一个元组，直到它基于一群元组计算完结果。聚合和连接一般也会 multi-anchore 它们输出元组。这些东西不属于简单的 IBasicBolt 模式。

2.5.4 Storm 如何实现可靠性

一个 Storm 拓扑有一组特殊的 Acker 任务，对于每一个 Spout 元组，跟踪元组的有向无环图。当一个 Acker 看到有向无环图是完整的，它发送一个消息到 Spout 任务，创造 Spout 元组来证实消息。可以在拓扑配置中使用 Config.TOPOLOGY_ACKERS 为一个拓扑设置 Acker 任务的数量。Storm 默认 TOPOLOGY_ACKERS 是 1 个，对于拓扑处理大量的信息，需要增加这个数字。

理解 Storm 可靠性实现的最好方式，是看元组的生命周期和元组的有向无环图。当元组在拓扑中被创建，无论是在一个 Spout 或 Bolt 中，它是给定一个随机抽取的 64 位 id。这些 id 被 Acker 用来对每一个 Spout 元组跟踪元组的有向无环图。

每个元组知道所有 Spout 元组的 id（因为它在元组树中存在）。当你在一个 Bolt 发射一个新的元组时，来自元组的锚的 Spout 元组的 id 都复制到新的 tuple。当一个元组是 acked 时，它发射一个消息以及关于为什么元组树被改变的信息到相应的 Acker 任务；特别是它告诉 Acker“我现在在这个 Spout 元组的元组树里面完成了，而在树上的新的元组被锚定到我这里。”

例如，如果基于元组 C 创建元组 D 和 E，当 C 是 acked 时，元组树就改变了，如图 2.5 所示。

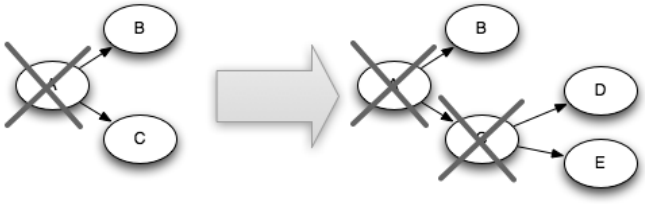


图 2.5　元组树的例子

从树中移除 C 的同时，D 和 E 被添加到其中，因此树永远不会过早地完成。

之前已经提到，在一个 Topology 里可以拥有任意数量的 Acker 任务。这导致了以下问题：当一个元组在 Topology 里面是 acked 时，它如何知道是哪一个 Acker 任务发送这些信息的呢？

Storm 使用取模哈希映射一个 Spout 元组 id 到一个 Acker 任务。由于知道每一个元组与其存在于所有树的 Spout 的元组的 id，它们知道该与哪些 Acker 任务沟通。

Storm 的另一个细节是 Acker 任务如何追踪了解哪个 Spout 任务是负责它们跟踪的 Spout 元组：当一个 Spout 任务发送一个新的元组时，它只是将消息发送给适当的 Acker，告知其任

务 id 负责 Spout 元组；然后当一个 Acker 看到一棵树已经完成，它知道发送完成消息到哪个任务 id。

Acker 任务不显式跟踪元组树。对于大量的元组树与成千上万的节点（或更多），跟踪所有的元组树可能会超过 Acker 所使用的内存容量。相反，Acker 采取不同的策略，只需要每个 Spou 元组一个固定的空间量（约 20 个字节）。这种跟踪算法是 Storm 如何工作的关键，这是 Storm 的一个重大突破。

一个 Acker 任务存储来自 Spout 元组 id 的 map 的一对值。第一个值是创建 Spout 元组的任务 id，用于稍后发送完成消息。第二个值是一个 64 位的值，称为 ack val。ack val 是整个元组树的表示状态，不管多大多小；它仅仅是所有元组 id 的 XOR，已经在树上被创建和/或 acked。

当一个 Acker 任务看到 ack val 已经成为 0，它就知道元组树已经完成了。因为元组 id 是随机的 64 位数字，ack val 意外地成为 0 的可能性是非常小的。如果每秒 1 万的 ack，直到一个错误发生，它需要五百万年。另外，即使元组在这个 Topology 发生失败，只会导致数据丢失。

既然了解了可靠性算法，让我们复习一下所有的失败案例，看看在每种情况下 Storm 如何避免数据丢失。

（1）任务挂了导致元组没有被 ack:

在这种情况下，在树根的失败元组的 Spout 元组 id 会超时并被重发。

（2）Acker 任务挂了:

在这种情况下，所有的 Spout 元组跟踪的 Acker 会超时并被重发。

（3）Spout 任务挂了:

在这种情况下，Spout 任务通信的来源负责重放消息。例如，当客户端断开连接时，Kestrel 和 RabbitMQ 等消息队列会将所有等待的消息放回队列中。

如你所见，Storm 的可靠性机制是完全分布式、可伸缩的、容错的。

2.5.5　调节可靠性

Acker 任务是轻量级的，所以不需要很多的拓扑。可以通过 Storm UI（组件 id 为“__acker”）跟踪它们的性能。如果吞吐量看起来不太对，需要添加更多的 Acker 任务。

如果可靠性对你来说不是很重要——也就是说，你不在乎在失败的情况下失去元组，那么可以不跟踪 Spout 元组的元组树以提高 Storm 性能。没有跟踪一个元组树部分转移的消息数量，通常对于元组树的每一个元组存在一个 ack 消息。此外，它需要更少的 id 保存在每个下游的消息，减少带宽使用情况。

有三种方法可以删除可靠性。

第一种是设置 Config.TOPOLOGY_ACKERS 为 0。在这种情况下，Storm 会在 Spout 发射一个元组后，立即调用 Spout 的 ack 方法。元组树不会被跟踪。

第二种方法是通过消息基础删除消息的可靠性。可以在 SpoutOutputCollector.emit 方法中忽略消息 id，关掉对于个别的 Spout 元组的追踪。

最后一种方法是，如果你不介意是否一个拓扑下游元组的特定子集无法被处理，可以作为非固定元组发射它们。因为它们没有固定到任何 Spout 元组，所以，如果它们没有 acked，不会造成任何 Spout 元组失败。

2.6 消息传输机制

2.6.1 ZeroMQ

ØMQ 也称为 ZeroMQ、0MQ 或者 zmq，是一个简单好用的传输层，一个像框架一样的 Socket 库，一个消息处理队列库。它使得 Socket 编程更加简单，性能更高，可以跨进程内、进程间、TCP、多播等进行传输，支持 C、C++、Java、NET、Python 等 40 多种语言，支持 Linux、Windows、OS X 等大多数操作系统，以 LGPLv3 协议进行开源。

ZeroMQ 是 Storm 一直都在使用的消息传输机制。但是，ZeroMQ 有其局限性。主要体现在如下几个方面。

- ZeroMQ 是一个本地化的消息库，它过度依赖操作系统环境;
- ZeroMQ 的安装比较麻烦;
- ZeroMQ 的稳定性在不同版本之间差异巨大，并且目前只有 2.1.7 版本的 ZeroMQ 能与 Storm 协调工作。

2.6.2 Netty

Netty 是 Storm 0.9 版本新引入的传输机制。Netty 提供了一个纯 Java 的消息通信解决方案，消除了 Storm 对本地库的依赖。Netty 的传输性能是 ZeroMQ 的两倍，并且它使得工作进程之间的授权和认证成为可能。

Storm 默认使用 ZeroMQ 作为传输层，如果要在 Storm 中使用 Netty，只需要把下面的内容加入到 storm.yaml 文件中，并根据实际情况作适当调整即可。

```
storm.messaging.transport: "backtype.storm.messaging.netty.Context"
storm.messaging.netty.server_worker_threads: 1
storm.messaging.netty.client_worker_threads: 1
storm.messaging.netty.buffer_size: 5242880
storm.messaging.netty.max_retries: 100
storm.messaging.netty.max_wait_ms: 1000
storm.messaging.netty.min_wait_ms: 100
```

2.6.3　自定义消息通信机制

如果不喜欢 Netty 或者 ZeroMQ，只要通过实现 backtype.storm.messaging.IContext 接口，就可以自定义消息通信机制。

2.7　Storm 的开发环境与生产环境

Storm 的环境一般分为生产环境与开发环境两种。

2.7.1　开发环境与本地模式

Storm 的开发环境是指供 Storm 开发人员进行 Storm 程序开发与测试的环境。Storm 的硬件环境只需要一台普通 PC 主机，软件环境需要主机安装 Linux 或者 Windows 操作系统，另外需要安装与配置 JDK、Maven、Eclipse 等软件。

在开发环境上面开发 Storm 应用，称为本地模式。

本地模式在进程里面模拟了一个 Storm 集群，用于开发和测试拓扑。在本地模式下运行拓扑类似于在集群上运行拓扑。

2.7.2　生产环境与远程模式

Storm 的生产环境，也就是 Storm 集群环境，需要 3 台或者更多的主机作为节点，安装 Linux 操作系统，同时需要安装与配置 JDK、ZooKeeper、Storm、ZeroMQ、JZMQ、Python 和 unzip 等。

在生产环境上运行 Storm 应用，称为远程模式。

从集群的角度来看，Storm 的生产环境主要包括 ZooKeeper 集群和 Storm 集群。Storm 集群的拓扑结构如图 2.6 所示。

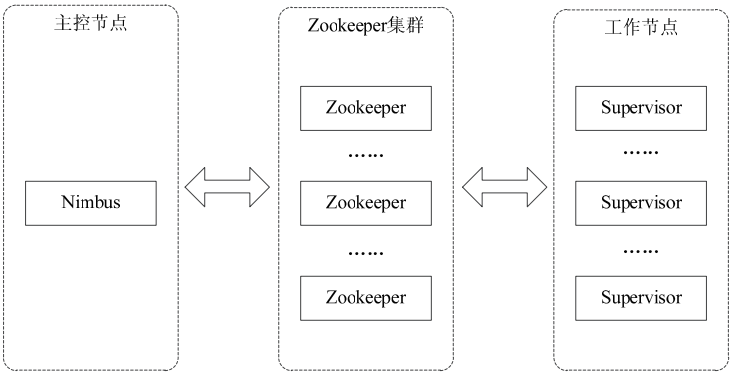


图 2.6　Storm 集群的拓扑结构

Storm 集群类似于 Hadoop 集群，Hadoop 上运行的是“MapReduce 作业”，而在 Storm 上运行的是“拓扑”。“作业”和“拓扑”本身是非常不同的，一个关键的区别是，MapReduce 作业最后是完成的，而拓扑永不休止地处理消息（或直到你杀死它）。Hadoop 与 Storm 的对比如表 2.1 所示。

表 2.1　Hadoop 与 Storm 的对比

对比项	Hadoop	Storm
系统角色	JobTracker	Nimbus
	TaskTracker	Supervisor
	Child	Worker（工作进程）
应用名称	Job	Topology（拓扑）
组件接口	Mapper/Reducer	Spout/Bolt

在 Storm 集群中有两种类型的节点，即主控节点（the master node）和工作节点（the worker nodes）。其中，主控节点只有一个，而工作节点可以有多个。

主控节点运行一个守护进程，称为 Nimbus，类似于 Hadoop 的 JobTracker。Nimbus 负责在集群中分发代码，对节点分配任务，并监视主机故障。

每个工作节点运行一个守护进程，称为 Supervisor。Supervisor 监听已经分配到它的节点的作业，启动和停止 Nimbus 已经分配给它的工作进程。每个工作进程执行 Topology 的一个子集；一个运行的 Topology 包含扩散到许多主机的许多工作进程。Nimbus 和 Supervisor 之间的所有的协调工作是由 ZooKeeper 集群完成的。另外，Nimbus 守护进程与 Supervisor 守护进程是快速失败和无状态的。所有状态都保存在 ZooKeeper 中或者本地磁盘上。这意味着可以使用“kill -9”强制杀死 Nimbus 或者 Supervisor，之后他们会开始备份，像什么都没发生一样。这种设计使 Storm 集群变得非常稳定。

2.7.3　开发环境与生产环境的对比

表 2.2 列出了搭建 Storm 所需的硬件环境，表 2.3 展示了搭建 Storm 生产环境所需的主要软件及下载方式。

表 2.2　搭建 Storm 所需的硬件环境

比较项	开发环境	生产环境
操作系统	Linux/Windows	Linux
主机数量	1 台	3 台或者更多

表 2.3　搭建 Storm 生产环境所需的主要软件及下载方式

软件	环境	下载地址	推荐版本
JDK	开发/生产	http://www.oracle.com/technetwork/java/javase/downloads/index.html	1.6 以上
ZooKeeper	生产	http://zookeeper.apache.org/releases.html	最新稳定版

（续表）

软件	环境	下载地址	推荐版本
Storm	生产	http://storm-project.net/downloads.html	最新稳定版
ZeroMQ	生产	http://www.zeromq.org/area:download	2.1.7
JZMQ	生产	https://github.com/nathanmarz/jzmq	最新稳定版
Python	生产	http://www.python.org/download/	2.6.6 以上
unzip	生产	http://www.info-zip.org/UnZip.html#Downloads	最新稳定版
Maven	开发	http://maven.apache.org/download.cgi	最新稳定版
Eclipse	开发	http://www.eclipse.org/downloads/	for Java EE Developers 版本

2.8　Storm 拓扑的并行度（parallelism）

2.8.1　工作进程、执行器和任务

Storm 区分以下 3 个主要实体，用来在 Storm 群集中运行拓扑。

- 工作进程（Worker Process）
- 执行器（Executor），即线程（Thread）
- 任务（Task）

工作进程、执行器、任务三者之间关系的一个简单例子如图 2.7 所示。

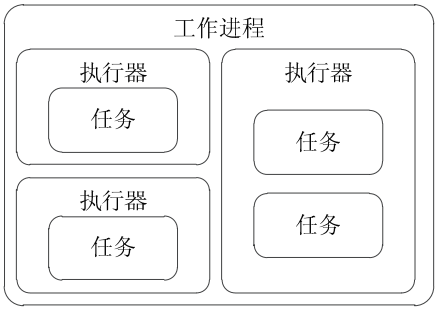


图 2.7　工作进程、执行器、任务三者之间的关系

Storm 集群的一个节点可能有一个或者多个工作进程运行在一个或者多个拓扑上，一个工作进程执行拓扑的一个子集。工作进程属于一个特定的拓扑，并可能为这个拓扑的一个或多个组件（Spout 或 Bolt）运行一个或多个执行器。一个运行中的拓扑包括多个运行在 Storm 集群内多个节点的进程。

一个或者多个执行器可能运行在一个工作进程内，执行器是由工作进程产生的一个线程，它可能为相同的组件（Spout 或 Bolt）运行一个或多个任务。

任务执行真正的数据处理，代码中实现的每个 Spout 或 Bolt，作为很多任务跨集群执行。

一个组件的任务数量始终贯穿拓扑的整个生命周期，但一个组件的执行器（线程）数量可以随时间而改变。这意味着，下述条件成立：#threads ≤ #tasks。默认情况下，任务的数量被设定为相同的执行器的数量，即 Storm 会使用每个线程执行一个任务。

2.8.2 配置拓扑的并行度

请注意，Storm 的术语“并行度（parallelism）”专门用于描述所谓的并行度暗示，表示一个组件的执行器（线程）的初始数量。在这个文件中，虽然我们使用“并行度”在更一般的意义上描述了如何配置执行器的数量、工作进程的数量和 Storm 拓扑的任务数量；但我们将特别指出，“并行度”是用在 Storm 中正常的、狭窄的定义。

以下给出各种配置选项的概述以及如何在代码中进行设置。虽然设置这些选项的方法不止一种，但是表格中只列出了其中的一些。Storm 目前有如下配置设置的优先级顺序：defaults.yaml<storm.yaml<特定拓扑的配置<内部特定组件的配置<外部特定组件的配置。

1. 工作进程的数量

描述：集群中不同节点的拓扑可以创建多少个工作进程。
配置选项：TOPOLOGY_WORKERS
如何在代码中设置示例：

```
Config#setNumWorkers
```

2. 执行器/线程的数量

描述：每个组件产生多少个执行器。
配置选项：暂无
如何在代码中设置示例：

```
TopologyBuilder#setSpout()
TopologyBuilder#setBolt()
```

Storm 的 parallelism_hint 参数现在指定为 Bolt 的执行者初始数量，而不是任务的初始数量。

3. 任务的数量（Number of tasks）

描述：每个组件创建多少个任务。
配置选项：TOPOLOGY_TASKS
（http://nathanmarz.github.io/storm/doc-0.8.1/backtype/storm/Config.html#TOPOLOGY_TASKS）
如何在代码中设置示例：

```
ComponentConfigurationDeclarer#setNumTasks()
T setNumTasks(java.lang.Number val)
```

下面是一个例子的代码片断，在实践中显示这些设置：

```
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
             .setNumTasks(4)
             .shuffleGrouping("blue-spout");
```

在上面的代码中，我们配置 Storm 运行 GreenBolt，并初始化两个执行器及 4 个相关任务。Storm 会为每个执行器/线程运行两个任务。如果不显式配置任务的数量，Storm 默认每个执行器运行一个任务。

2.8.3　拓扑示例

下面定义一个名为 mytopology 的拓扑，由一个 Spout 组件（BlueSpout）、两个 Bolt 组件（GreenBolt 和 YellowBolt）共 3 个组件构成。

```
Config conf = new Config();
conf.setNumWorkers(2); // 使用两个工作进程

topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // 设并行度为 2

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
        .setNumTasks(4) // 使用 4 个任务
        .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
        .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology(
        "mytopology",
        conf,
        topologyBuilder.createTopology()
);
```

mytopology 拓扑的描述如下：

- 拓扑将使用两个工作进程（Worker）；
- Spout 是 id 为 blue-spout、并行度为 2 的 BlueSpout 实例（产生两个执行器和两个任务）；
- 第一个 Bolt 是 id 为 green-bolt、并行度为 2、任务数为 4、使用随机分组方式接收 blue-spout 所发射元组的 GreenBolt 实例（产生两个执行器和 4 个任务）；
- 第二个 Bolt 是 id 为 yellow-bolt、并行度为 6、使用随机分组方式接收 green-bolt 所发射元组的 YellowBolt 实例（产生 6 个执行器和 6 个任务）。

所以，该拓扑一共有两个工作进程（Worker），2+2+6=10 个执行器（Executor），2+4+6=12 个任务。因此，每个工作进程可以分配到 10/2=5 个执行器，12/2=6 个任务。默认情况下，一

个执行器执行一个任务，但是如果指定了任务的数目，则任务会平均分配到执行器中，因此，GreenBolt 的实例 green-bolt 的一个执行器将会分配到 4/2=2 个任务。mytopology 的拓扑及其对应的资源分配如图 2.8 所示。

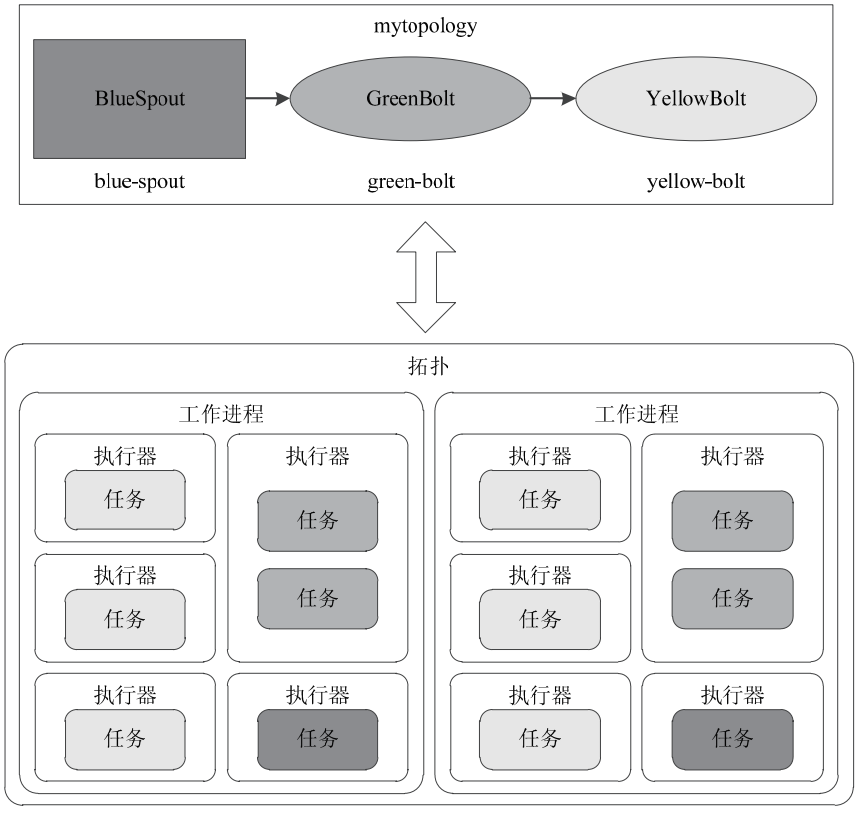


图 2.8　mytopology 的拓扑及其对应的资源分配

Storm 提供了设置拓扑的并行度的其他方法，目前主要有 TOPOLOGY_MAX_TASK_PARALLELISM，即设置单个组件产生的执行器的最大数量。通常在测试阶段使用，限制在本地模式运行拓扑时产生的线程数量，可以通过 Config#setMaxTaskParallelism()方法来配置该选项。

2.8.4　如何改变运行中拓扑的并行度

Storm 一个很好的特性是，可以增加或减少工作进程（Worker）和/或执行器（Executor）的数量而不需要重新启动群集或拓扑，这样的行为被称为再平衡（rebalancing）。

有两种方式可实现拓扑再平衡：

- 使用 Storm Web UI。
- 使用 CLI 工具。

下面是使用 CLI 工具实现拓扑再平衡的一个示例：

```
# 重新配置拓扑
# "mytopology" 拓扑使用 5 个 Worker 进程
# "blue-spout" Spout 使用 3 个 Executor
# "yellow-bolt" Bolt 使用 10 个 Executor

$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10
```

2.9 Storm 命令行客户端

在 Linux 终端直接输入 storm，不带任何参数信息，或者输入 storm help，可以查看 Storm 命令行客户端（Command line client）提供的帮助信息。

Storm 0.9.0.1 版本在 Linux 终端直接输入 storm 后的输出内容如下：

```
Commands:
    activate
    classpath
    deactivate
    dev-zookeeper
    drpc
    help
    jar
    kill
    list
    localconfvalue
    logviewer
    nimbus
    rebalance
    remoteconfvalue
    repl
    shell
    supervisor
    ui
    version

Help:
    help
    help <command>
```

```
    Documentation for the storm client can be found at https://github.com/nathan
marz/storm/wiki/Command-line-client

    Configs can be overridden using one or more -c flags, e.g. "storm list -c nim
bus.host=nimbus.mycompany.com"
```

由此可知，新版 Storm 的命令行客户端提供了 19 个命令。

1. activate

激活指定的拓扑 Spout。语法如下：

```
storm activate topology-name
```

2. classpath

打印出 Storm 客户端运行命令时使用的类路径（classpath）。语法如下：

```
storm classpath
```

3. deactivate

禁用指定的拓扑 Spout。语法如下：

```
storm deactivate topology-name
```

4. dev-zookeeper

以 dev.zookeeper.path 配置的值作为本地目录，以 storm.zookeeper.port 配置的值作为端口，启动一个新的 ZooKeeper 服务，仅用来开发/测试。语法如下：

```
storm dev-zookeeper
```

5. drpc

启动一个 DRPC 守护进程。语法如下：

```
storm drpc
```

该命令应该使用 daemontools 或者 monit 工具监控运行。

6. help

打印一条帮助消息或者可用命令的列表。语法如下：

```
storm help
storm help <command>
```

直接输入不带参数的 storm，也可以启动 storm help 命令。

7. jar

运行类的指定参数的 main 方法。语法如下：

```
storm jar topology-jar-path class ...
```

把 Storm 的 jar 文件和“~/.storm”的配置放到类路径（classpath）中，以便当拓扑提交时，StormSubmitter 会上传 topology-jar-path 的 jar 文件。

8. kill

杀死名为 topology-name 的拓扑。语法如下：

```
storm kill topology-name [-w wait-time-secs]
```

Storm 首先会在拓扑的消息超时时间期间禁用 Spout，以允许所有正在处理的消息完成处理。然后，Storm 将会关闭 Worker 并清理它们的状态。可以使用-w 标记覆盖 Storm 在禁用与关闭期间等待的时间长度。

9. list

列出正在运行的拓扑及其状态。语法如下：

```
storm list
```

10. localconfvalue

打印出本地 Storm 配置的 conf-name 的值。语法如下：

```
storm localconfvalue conf-name
```

本地 Storm 配置是~/.storm/storm.yaml 与 defaults.yaml 合并的结果。

11. logviewer

启动 Logviewer 守护进程。语法如下：

```
storm logviewer
```

Logviewer 提供一个 Web 接口查看 Storm 日志文件。该命令应该使用 daemontools 或者 monit 工具监控运行。

12. nimbus

启动 Nimbus 守护进程。语法如下：

```
storm nimbus
```

该命令应该使用 daemontools 或者 monit 工具监控运行。

13. rebalance

语法如下：

```
storm rebalance topology-name [-w wait-time-secs]
```

有时你可能希望扩散一些正在运行的拓扑的 Worker。例如，假设你有一个 10 个节点的集群，每个节点运行 4 个 Worker，然后假设需要添加另外 10 个节点到集群中。你可能希望有 Spout 扩散正在运行中的拓扑的 Worker，这样每个节点运行两个 Worker。解决的一种方法是杀死拓扑并重新提交拓扑，但 Storm 提供了一个 rebalance 的命令，我们可以用一种更简单的方法来做到这一点。

rebalance 首先会在消息超时时间内禁用拓扑，使用-w 可以覆盖超时时间，然后重新均衡分配集群的 Worker，拓扑会返回到它原来的状态，即禁用的拓扑仍将禁用，激活的拓扑继续激活。

14. remoteconfvalue

打印出远程集群 Storm 配置的 conf-name 的值。语法如下：

```
storm remoteconfvalue conf-name
```

集群 Storm 配置是$STORM-PATH/conf/storm.yaml 与 defaults.yaml 合并的结果。该命令必须在集群节点上运行。

15. repl

打开一个包含类路径（classpath）中的 jar 文件和配置的 Clojure REPL，以便调试时使用。语法如下：

```
storm repl
```

Clojure 可以作为一种脚本语言内嵌到 Java 中，但是 Clojure 的首选编程方式是使用 REPL，REPL 是一个简单的命令行接口。使用 REPL，可以输入命令并执行，然后查看结果。

16. shell

执行 Shell 脚本。语法如下：

```
storm shell resourcesdir command args
```

17. supervisor

启动 Supervisor 守护进程。语法如下：

```
storm supervisor
```

该命令应该使用 daemontools 或者 monit 工具监控运行。

18. ui

启动 UI 守护进程。语法如下：

```
storm ui
```

UI 为 Storm 集群提供了一个 Web 界面并显示运行拓扑的详细统计信息。该命令应该使用 daemontools 或者 monit 工具监控运行。

19. version

打印 Storm 发布的版本号。语法如下：

```
storm version
```

2.10　Javadoc 文档

对于最新的 Storm 提供的在线 Javadoc 文档信息，可以参考如下网址：

```
http://nathanmarz.github.io/storm/
```

目前，Storm 提供的在线 Javadoc 文档如表 2.4 所示。

表 2.4　Storm 提供的在线 Javadoc 文档

Storm 版本	Javadoc 文档的链接地址
Storm 0.8.1	http://nathanmarz.github.io/storm/doc-0.8.1/index.html
Storm 0.8.0	http://nathanmarz.github.io/storm/doc-0.8.0/index.html
Storm 0.7.1	http://nathanmarz.github.io/storm/doc-0.7.1/index.html
Storm 0.7.0	http://nathanmarz.github.io/storm/doc-0.7.0/index.html
Storm 0.6.2	http://nathanmarz.github.io/storm/doc-0.6.2/index.html

2.11　本章小结

本章对 Storm 的基础知识作了一系列的描述。通过本章的学习，读者应该了解 Storm 的基本概念，流由元组组成，Spout 是流的来源，Bolt 是流的处理单元，拓扑是 Spout 和 Bolt 的有向图；了解 Storm 的配置、序列化、容错机制、可靠性机制、消息传输机制、开发环境与生产环境；理解 Storm 拓扑的并行度（parallelism）；熟悉 Storm 命令行客户端、Javadoc 文档等内容。

第 3 章

拓扑详解

在这一章，你将学会如何在 Storm 拓扑的不同组件之间传输元组，以及如何部署拓扑到一个运行中的 Storm 集群。

3.1　什么是拓扑

要使用 Storm 做实时计算，首先需要创建所谓的“拓扑（Topology）”。一个拓扑是一个有向图的计算。在一个拓扑中的每个节点包含处理逻辑，节点之间的连接显示数据应该如何在节点之间传递。

拓扑的运行是很简单的。首先，打包所有的代码和依赖到一个单独的大 jar 包中。然后，运行如下命令：

```
storm jar all-my-code.jar backtype.storm.MyTopology arg1 arg2
```

该命令使用参数 arg1 和 arg2 来运行 all-my-code.jar 包里面的类 backtype.storm.MyTopology。类的主要功能是定义了拓扑，并将它提交到 Nimbus。storm jar 命令部分负责连接 Nimbus 和上传 jar 包。

因为拓扑的定义是 Thrift 结构，而 Nimbus 是一个 Thrift 服务，所以可以使用任何编程语言来创建并提交 Topology。上面的命令是基于 JVM 语言实现的最简单方法。

3.2　TopologyBuilder

TopologyBuilder 是构建拓扑的类，用于指定执行的拓扑。拓扑底层是 Thrift 结构，由于 Thrift API 非常冗长，使用 TopologyBuilder 可以极大地简化建立拓扑的过程。

TopologyBuilder 的公有方法如图 3.1 所示。

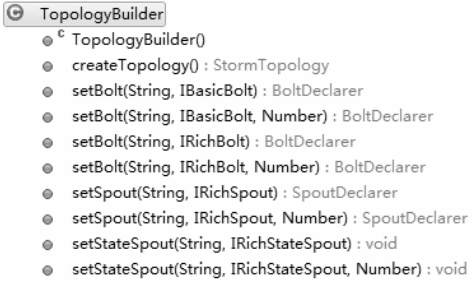


图 3.1　TopologyBuilder 的公有方法

创建和提交拓扑的过程如下：首先，使用 new 关键字创建一个 TopologyBuilder 对象，然后调用 setSpout 方法设置 Spout，接着调用 setBolt 方法设置 Bolt，最后调用 createTopology 方法返回 StormTopology 对象给 submitTopology 方法作为输入参数。

创建并提交 Topology 到 Storm 集群的完整代码如下：

```
// 创建 TopologyBuilder 对象
TopologyBuilder builder = new TopologyBuilder();

// 添加一个 id 为“1”，并行度为 5 的 TestWordSpout 对象
builder.setSpout("1", new TestWordSpout(true), 5);
// 添加一个 id 为“2”，并行度为 3 的 TestWordSpout 对象
builder.setSpout("2", new TestWordSpout(true), 3);
// 添加一个 id 为“3”，并行度为 3 的 TestWordCounter 对象
// 对 id 为“1”的组件按“word”字段进行分组
// 对 id 为“2”的组件按“word”字段进行分组
builder.setBolt("3", new TestWordCounter(), 3)
        .fieldsGrouping("1", new Fields("word"))
        .fieldsGrouping("2", new Fields("word"));
// 添加一个 id 为“4”，并行度为 1 的 TestGlobalCount 对象
// 对 id 为“1”的组件按全局分组
builder.setBolt("4", new TestGlobalCount(),1)
        .globalGrouping("1");

Map conf = new HashMap();                  // 创建 HashMap 对象
conf.put(Config.TOPOLOGY_WORKERS, 4);  // 设置 Worker 的数量为 4

// 提交拓扑
StormSubmitter.submitTopology("mytopology", conf, builder.createTopology());
```

在本地模式（进程中）下运行完全相同的拓扑的代码如下：

```
TopologyBuilder builder = new TopologyBuilder(); // 创建 TopologyBuilder 对象

// 添加一个 id 为“1”，并行度为 5 的 TestWordSpout 对象
builder.setSpout("1", new TestWordSpout(true), 5);
// 添加一个 id 为“2”，并行度为 3 的 TestWordSpout 对象
builder.setSpout("2", new TestWordSpout(true), 3);
// 添加一个 id 为“3”，并行度为 3 的 TestWordCounter 对象
// 对 id 为“1”的组件按“word”字段进行分组
// 对 id 为“2”的组件按“word”字段进行分组
builder.setBolt("3", new TestWordCounter(), 3)
        .fieldsGrouping("1", new Fields("word"))
        .fieldsGrouping("2", new Fields("word"));
// 添加一个 id 为“4”，并行度为 1 的 TestGlobalCount 对象
// 对 id 为“1”的组件按全局分组
```

```
builder.setBolt("4", new TestGlobalCount(),1)
        .globalGrouping("1");

Map conf = new HashMap();                    // 创建 HashMap 对象
conf.put(Config.TOPOLOGY_WORKERS, 4);   // 设置 Worker 的数量为 4
conf.put(Config.TOPOLOGY_DEBUG, true); // 设置调试模式为 true

LocalCluster cluster = new LocalCluster();  // 创建 LocalCluster 对象
// 提交拓扑
cluster.submitTopology("mytopology", conf, builder.createTopology());
Utils.sleep(10000);                          // 线程睡眠 10 秒，即拓扑可以运行 10 秒
cluster.shutdown();                          // 关闭拓扑
```

3.3 流分组

3.3.1 什么是流分组

流分组是拓扑定义的一部分，为每个 Bolt 指定应该接收哪个流作为输入。流分组定义了流/元组如何在 Bolt 的任务之间进行分发。

在集群中，Spout 和 Bolt 并行执行许多任务。如果你在任务层看拓扑是怎样执行的，则会看到流分组的示意图，如图 3.2 所示。

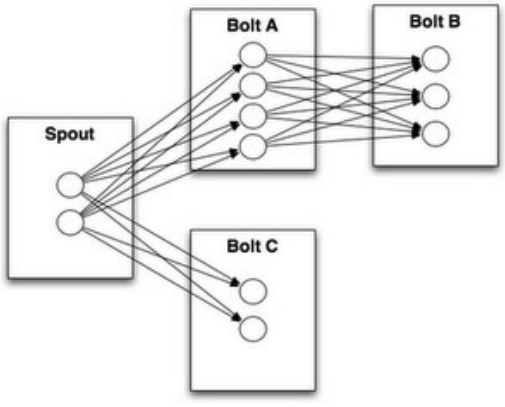


图 3.2　流分组的示例图

当 Bolt A 的任务发送元组到 Bolt B 时，它应该发送给 Bolt B 的哪一个任务呢？“流分组”回答了这个问题，告诉 Storm 如何在任务集之间发送元组。

在设计拓扑的时候，需要做的一件最重要的事情，就是定义数据如何在组件之间进行交换

（流如何被 Bolt 消耗）。一个流分组指定每个 Bolt 消耗哪个流，流将如何被消耗。

一个节点可以发出多个数据流，流分组允许我们有选择地接收流。

在深入研究不同类型的流分组之前，让我们先来看看 storm-starter 项目的 WordCountTopology 拓扑。WordCountTopology 从一个 Spout 中读取句子，WordCountBolt 统计单词的总次数。WordCountTopology 的定义代码如下：

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8)
       .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 12)
       .fieldsGrouping("split", new Fields("word"));
```

SplitSentence 为它接收的每个句子的每个单词发送一个元组，WordCount 就在内存中保持一个单词到计数的映射。WordCount 每次收到一个词，就会更新其状态并发送新单词的计数。

Storm 内置了 7 种流分组方式，通过实现 CustomStreamGrouping 接口可以实现自定义的流分组。

3.3.2 不同的流分组方式

InputDeclarer 接口定义了不同的流分组方式。每当 TopologyBuilder 的 setBolt 方法被调用就返回该对象，用于声明一个 Bolt 的输入流，以及这些流应该如何分组。InputDeclarer 接口的完整定义代码如下：

```
public interface InputDeclarer<T extends InputDeclarer> {
    // 字段分组
    public T fieldsGrouping(String componentId, Fields fields);
    public T fieldsGrouping(String componentId, String streamId, Fields fields);

    // 全局分组
    public T globalGrouping(String componentId);
    public T globalGrouping(String componentId, String streamId);

    // 随机分组
    public T shuffleGrouping(String componentId);
    public T shuffleGrouping(String componentId, String streamId);

    // 本地或者随机分组
    public T localOrShuffleGrouping(String componentId);
```

```
        public T localOrShuffleGrouping(String componentId, String streamId);

        // 无分组
        public T noneGrouping(String componentId);
        public T noneGrouping(String componentId, String streamId);

        // 广播分组
        public T allGrouping(String componentId);
        public T allGrouping(String componentId, String streamId);

        // 直接分组
        public T directGrouping(String componentId);
        public T directGrouping(String componentId, String streamId);

        // 自定义分组
        public T customGrouping(String componentId, CustomStreamGrouping grouping);
        public T customGrouping(String componentId, String streamId, CustomStre
amGrouping grouping);
        public T grouping(GlobalStreamId id, Grouping grouping);
    }
```

从 InputDeclarer 接口中可以看出，流分组的方式主要有 fieldsGrouping（字段分组）、globalGrouping（全局分组）、shuffleGrouping（随机分组）、localOrShuffleGrouping（本地或者随机分组）和 noneGrouping（无分组）、allGrouping（广播分组）、directGrouping（直接分组）、customGrouping（自定义分组）这 8 种不同的流分组方式。每个 InputDeclarer 实例可以有不止一个源，每个源可以用不同的流分组方式来分组。

1. 随机分组

随机分组（Shuffle Grouping）是最常用的流分组方式，它随机地分发元组到 Bolt 上的任务，这样能保证每个任务得到相同数量的元组。

随机分组执行原子操作，这是非常有用的，例如数学运算。但是，如果操作不能被随机分发的话，应该考虑使用其他的分组方式，例如，在单词统计（WordCount）例子中，需要计算单词，就不适合使用随机分组。

2. 字段分组

字段分组（Fields Grouping）根据指定字段对流进行分组。例如，如果流是按 user-id 字段进行分组，具有相同 user-id 的元组总是被分发到相同的任务，具有不同 user-id 的元组可能被分发到不同的任务。

字段分组是实现流连接和关联，以及大量其他的用例的基础。在实现上，字段分组使用取模散列来实现。

3. 广播分组

广播分组（All Grouping）是指流被发送到所有 Bolt 的任务中。使用这个分组方式时要小心。

4. 全局分组

全局分组（Global Grouping）是指全部流都发送到 Bolt 的同一个任务中，再具体一点，是发送给 ID 最小的任务。

5. 无分组

假定你不关心流是如何分组的，则可以使用这种分组方式。目前这种分组和随机分组是一样的效果，有一点不同的是 Storm 会把这个 Bolt 放到 Bolt 的订阅者的同一个线程中执行。

6. 直接分组

直接分组（Direct Grouping）是一种特殊的分组。这种方式的流分组意味着由元组的生产者决定元组的消费者的接收元组的任务。直接分组只能在已经声明为直接流（Direct Stream）的流中使用，并且元组必须使用 emitDirect 方法来发射。Bolt 通过 TopologyContext 对象或者 OutputCollector 类的 emit 方法的返回值，可以得到其消费者的任务 id 列表（List<Integer>）。

7. 本地或者随机分组

如果目标 Bolt 在同一工作进程存在一个或多个任务，元组会随机分配给这些任务。否则，该分组方式与随机分组方式是一样的。

8. 自定义分组

可以自定义流分组的方式，通过实现 CustomStreamGrouping 接口来创建自定义的流分组。CustomStreamGrouping 接口的定义如下：

```
    public interface CustomStreamGrouping extends Serializable {
        void prepare(WorkerTopologyContext context, GlobalStreamId stream, List
<Integer> targetTasks);
        List<Integer> chooseTasks(int taskId, List<Object> values);
    }
```

CustomStreamGrouping 接口主要有两个方法：prepare 和 chooseTasks。CustomStreamGrouping 接口的具体实现，可以参考如下的代码类：

- storm.trident.partition.GlobalGrouping
- storm.trident.partition.IdentityGrouping
- storm.trident.partition.IndexHashGrouping
- backtype.storm.testing.NGrouping

让我们来看一个简单的自定义流分组的实现，它来自 storm.trident.partition.GlobalGrouping。GlobalGrouping 是 Trident 中实现全局分组功能的自定义流分组类。GlobalGrouping 的类定义代码如下：

```
public class GlobalGrouping implements CustomStreamGrouping {

    List<Integer> target;

    @Override
    public void prepare(WorkerTopologyContext context, GlobalStreamId stream,
List<Integer> targets) {
        List<Integer> sorted = new ArrayList<Integer>(targets);
        Collections.sort(sorted);
        target = Arrays.asList(sorted.get(0));
    }

    @Override
    public List<Integer> chooseTasks(int i, List<Object> list) {
        return target;
    }
}
```

自定义流分组的使用是很简单的。假设对 ExclamationTopology 使用自定义流分组。ExclamationTopology 的“exclaim2”Bolt 原来是对“exclaim1”Bolt 使用随机分组，代码如下：

```
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```

现在，修改为“exclaim2”Bolt 对“exclaim1”Bolt 使用自定义流分组 GlobalGrouping，代码如下：

```
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .customGrouping("exclaim1", new GlobalGrouping());
```

3.4 一个简单的拓扑

下面来看一个简单的拓扑，这是来自 storm-starter 项目的 ExclamationTopology 类。ExclamationTopology 的定义如下：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
       .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
       .shuffleGrouping("exclaim1");
```

ExclamationTopology 包含一个 Spout 和两个 Bolt。Spout 发射单词，每个 Bolt 在输入处追加字符串 “!!!”。节点排列在一条线上：Spout 发送给第一个 Bolt，第一个 Bolt 发送给第二个 Bolt。如果 Spout 发送 Tuple["bob"]和["john"]，那么第二个 Bolt 将发送单词["bob!!!!!!"]和["john!!!!!!"]。

这段代码使用 setSpout 和 setBolt 方法定义节点。这些方法将一个用户指定的 id、一个包含了处理逻辑的对象、大量的并行度需要节点作为输入。在这个例子中，Spout 给定 id“words”和 Bolt 给定 id“exclaim1”和“exclaim2”。

对象包含了处理逻辑，实现了 Spout 的 IRichSpout 接口，Bolt 的 IRichBolt 接口。

最后一个参数确定想要多少个节点并行，是可选的。它表明有多少线程应该执行跨集群的组件。如果忽略它，Storm 只会对该节点分配一个线程。

setBolt 返回一个 InputDeclarer 对象，用于定义 Bolt 的输入。在这里，组件“exclaim1”声明，它想读取“words”组件随机分组发送的所有 Tuple，“exclaim2”组件声明，它想要读取“exclaim1”组件随机分组发送的所有 Tuple。“随机分组”意味着 Tuple 应该从输入任务到 Bolt 的任务进行随机分配。有很多方法可对组件之间的数据进行分组。

如果希望组件“exclaim2”读取“words”和“exclaim1”两个组件发送的所有的 Tuple，可以编写组件“exclaim2”的定义如下：

```
builder.setBolt("exclaim2", new ExclamationBolt(), 5)
          .shuffleGrouping("words")
          .shuffleGrouping("exclaim1");
```

正如你可以看到的，输入声明可以被链接到指定 Bolt 的多个来源。

让我们研究一下这个拓扑的 Spout 和 Bolt 的实现。Spout 负责发送新消息到 Topology。在此 Topology 的 TestWordSpout 每 100 毫秒从列表["nathan"， "mike"， "jackson"， "golda"，"bertels"]中随机发送一个的单词作为一个元组。在 TestWordSpout 中的 nextTuple()的实现类似这样：

```
    public void nextTuple() {
        Utils.sleep(100);
        final String[] words = new String[] {"nathan", "mike", "jackson", "golda",
 "bertels"};
        final Random rand = new Random();
        final String word = words[rand.nextInt(words.length)];
        _collector.emit(new Values(word));
    }
```

正如可以看到的，实现非常简单。

ExclamationBolt 追加字符串“!!!”作为它的输入。让我们看看 ExclamationBolt 类的完整实现：

```
    public static class ExclamationBolt implements IRichBolt {
        OutputCollector _collector;

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

        public void execute(Tuple tuple) {
            _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
            _collector.ack(tuple);
        }

        public void cleanup() {
        }

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

        public Map getComponentConfiguration() {
            return null;
        }
    }
```

prepare 方法为 Bolt 提供一个 OutputCollector 对象，用于从这个 Bolt 发射 Tuple。Tuple 可以随时从 Bolt 发射，从 Bolt 的 prepare、execute 或者 cleanup 方法，甚至是在另一个线程的异

步方法发射。这个 prepare 方法实现简单地把 OutputCollector 对象作为一个实例变量保存，使得稍后可以在 execute 方法中使用。

execute 方法从一个 Bolt 的输入接收一个 Tuple。这个 ExclamationBolt 从元组中取出第一个字段，把字符串“!!!”追加到它后面，然后发射出一个新的元组。如果要实现一个订阅多个输入来源的 Bolt，通过使用 Tuple.getSourceComponent 方法，可以找出 Tuple 来自哪个组件。

在 execute 方法中有一些其他的东西，即输入元组作为第一个传递参数来发射，输入元组在最后一行确认。这些是 Storm 的可靠 API 的一部分，保证没有数据丢失。

当 Bolt 被关闭时，cleanup 方法被调用来清理任何已经打开的资源。但不能保证这个方法会被集群调用，例如，如果节点上运行的任务被取消了，就没有办法调用该方法。cleanup 方法适用于当在本地模式（一个 Storm 集群是在进程中模拟）下运行拓扑，希望能够在没有遭受资源泄漏的情况下，运行和杀死一些拓扑。

declareOutputFields 方法声明 ExclamationBolt 类发射一个字段名为“word”的一元组。

getComponentConfiguration 方法允许你配置关于这个组件如何运行的很多参数。

cleanup 和 getComponentConfiguration 方法往往不需要在一个 Bolt 中实现。可以通过使用一个基类 BaseRichBolt 更简洁地定义 Bolt，这个基类在适当的地方提供了默认的实现。ExclamationBolt 可以通过继承 BaseRichBolt 类，写得更简洁一些：

```
    public static class ExclamationBolt extends BaseRichBolt {
        OutputCollector _collector;

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

        public void execute(Tuple tuple) {
            _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
            _collector.ack(tuple);
        }

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

3.5　在本地模式下运行拓扑

下面，我们看看如何在本地模式下运行 ExclamationTopology，以及它的工作原理。

Storm 有两种操作模式，即本地模式和分布式模式。在本地模式下，Storm 通过模拟工作节点的线程在进程中执行完成。本地模式对于测试和开发拓扑是有用的。当你在 storm-starter 中运行拓扑，它们将在本地模式下运行，可以看到每个组件发射的消息。

在分布式模式下，Storm 如同一个集群一样运转。当你提交一个拓扑到主控节点时，也提交了运行拓扑所需的所有代码。主控节点会好好分发你的代码，分配 Worker 去运行你的拓扑。如果 Worker 们瘫痪了，主控节点会在别的地方重新分配它们。

在本地模式下运行 ExclamationTopology 的代码如下：

```
Config conf = new Config();
conf.setDebug(true);
conf.setNumWorkers(2);

LocalCluster cluster = new LocalCluster();
cluster.submitTopology("test", conf, builder.createTopology());
Utils.sleep(10000);
cluster.killTopology("test");
cluster.shutdown();
```

首先，创建了一个 LocalCluster 对象，定义了一个进程内的集群。然后提交拓扑到这个虚拟集群，这等同于提交拓扑到分布的集群。它通过调用 submitTopology 方法提交一个拓扑到 LocalCluster，并把运行中的拓扑的名称、配置和拓扑本身作为输入参数。

名称用来识别拓扑，以便你可以在以后杀死它。一个拓扑将不停地运行下去，直到你杀死它。

配置用于优化运行的拓扑的各个方面。下面指定的两个配置是很常见的。

（1）TOPOLOGY_WORKERS（*参考 setNumWorkers 方法*）

该配置指定想要分配多少进程到集群去执行拓扑。拓扑中的每个组件将执行尽可能多的线程。分配到一个特定组件的线程的数量是通过 setBolt 和 setSpout 方法配置的。这些线程存在于工作进程中。每个工作流程包含它自身在内的某些组件的某些线程。例如，可能在所有组件指定有 300 个线程，在配置里面指定了 50 个工作进程。每个工作进程将执行 6 个线程，每个线程都可能属于一个不同的组件。你可以调优 Storm 拓扑的性能，通过调整每个组件的并行度，以及线程应该在其里面运行的工作进程的数量。

（2）TOPOLOGY_DEBUG（*参考 setDebug 方法*）

当该值设置成真时，每一个组件发射的每一个消息都让 Storm 记录到日志中。这在本地模

式下测试拓扑是非常有用的，但当在集群上运行拓扑时，你可能想保持这个选项关闭。

还可以为拓扑设置许多其他的配置。可以在 Config 的 Java 文档中找到各种配置的详细信息。

3.6 在生产集群上运行拓扑

在生产集群上运行拓扑，与在本地模式下运行拓扑不同，其步骤如下。

步骤 01 定义拓扑。如果使用的是 Java 语言，可以使用 TopologyBuilder 类来定义。

步骤 02 使用 StormSubmitter 提交拓扑到集群。StormSubmitter 需要拓扑的名称、拓扑配置 Config 对象、TopologyBuilder 对象作为输入参数。示例代码如下:

```
Config conf = new Config();
conf.setNumWorkers(20);
conf.setMaxSpoutPending(5000);
StormSubmitter.submitTopology("mytopology", conf, topology);
```

步骤 03 创建一个包含代码和所有依赖包(除 Storm 之外，因为 Storm 的 jar 包会添加到 Worker 节点的类路径上) 的 jar 包。

如果使用 Maven，则只需要在 pom.xml 文件中添加如下几行代码，就可以使用 maven-assembly-plugin 打包插件的功能，把所有依赖的 jar 都一起打包。

```
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <mainClass>com.path.to.main.Class</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

然后运行 mvn assembly:assembly 命令进行打包，并得到打包好的 jar 文件。

注意，请确认已经排除了 Storm 的 jar 包，因为集群的类路径上已经存在 Storm 的 jar 包了。

步骤 04 使用 Storm 客户端提交拓扑到集群。

一般，使用 Storm 客户端的 storm jar 命令提交 jar 包到集群。

storm jar 命令需要指定 jar 包的路径、执行拓扑类的名称与输入参数，命令用法如下：

```
storm jar path/to/allmycode.jar org.me.MyTopology arg1 arg2 arg3
```

其中，path/to/allmycode.jar 是 jar 包的路径，org.me.MyTopology 是拓扑类的完整类名，arg1、arg2、arg3 是拓扑类的输入参数。

为了使得 Storm 客户端能够与 Storm 集群进行通信，需要在~/.storm/storm.yaml 文件中配置 Nimbus 节点的主机名或者 IP 地址。假如 Nimbus 节点的主机名为 node200，则~/.storm/storm.yaml 里面的内容如下：

```
nimbus.host: "node200"
```

3.6.1　常见的配置

可以为每个拓扑设置大量的配置。一个可以设置的所有配置的列表可以在这个类（backtype.storm.Config）找到。那些前缀为 TOPOLOGY 的属性可以被特定拓扑所覆盖，其他的是集群配置，不能被覆盖。下面是一些常见的拓扑设置。

（1）Config.TOPOLOGY_WORKERS

这个设置执行 topology 的工作进程的数量。例如，如果你将这个参数设置为 25，则将会有 25 个 Java 进程跨集群执行所有的任务。如果你有一个跨拓扑中的所有组件的组合 150 并行度，每个工作进程将有 6 个任务作为线程运行。

（2）Config.TOPOLOGY_ACKERS

这是设置任务的数量，该任务将跟踪元组树，当 Spout 元组已经完全处理时进行检测。Acker 是 Storm 的可靠性模型不可或缺的一部分，你可以在 2.5 小节“可靠性机制——保证消息处理”阅读到关于它们的更多信息。

（3）Config.TOPOLOGY_MAX_SPOUT_PENDING

这是设置一次可以在 Spout 任务等待 Spout 元组的最大数量。等待意味着元组还尚未确认（acked）或失败（failed）。强烈推荐设置这个配置项防止队列溢出。

（4）Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS

这是一个 Spout 元组在它被认为是失败前必须完全完成的最大超时时间。这个值默认为 30 秒，这对于大多数拓扑来说是足够的。关于 Storm 的可靠性模型如何工作可查阅“可靠性机制——保证消息处理”小节以获得更多信息。

（5）Config.TOPOLOGY_SERIALIZATIONS

可以使用这个配置注册更多的序列化器，这样就可以在元组里面使用自定义类型。

3.6.2 杀死拓扑

希望杀死一个拓扑，只需要运行如下的命令：

```
storm kill {stormname}
```

stormname 参数是提交拓扑时使用的名称，使用 storm kill 命令，就可以杀死拓扑。

Storm 不会立即杀死拓扑。反而，它使所有的 Spout 无效，这样它们不会发送新的 Tuple。Storm 等待若干秒后，该时间由 Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS 配置项的值决定，摧毁所有的 Worker。这就给拓扑足够的时间来完成已存在的 Tuple 的处理工作。

3.6.3 更新运行中的拓扑

更新运行中的拓扑，目前唯一的方法是先杀死当前拓扑，然后启动一个新的拓扑。一个计划中的特性是实现一个 storm swap 命令，用一个新的拓扑交换一个运行中的拓扑，保证最小的停机时间，但是两个拓扑不可能在同一时间处理元组。

3.6.4 监控拓扑

监视拓扑的最好方法是使用 Storm UI。Storm UI 提供了有错误发生的任务和吞吐量的细粒度统计、每个运行中拓扑的每个组件的延迟性能等信息。

也可以查阅集群节点上面的工作日志。

3.7 拓扑的常见模式

Storm 拓扑中的一些常见模式主要有：

- 流连接
- 批处理
- BasicBolt
- 内存中缓存与字段的组合
- 计算 top N
- 高效保存最近更新对象缓存的 TimeCacheMap
- 分布式 RPC 的 CoordinatedBolt 与 KeyedFairBolt

3.7.1 流连接（Stream Join）

流连接基于一些常用字段，把两个或者更多的数据流结合到一起，形成一个新的数据流。

拿数据库的表连接与流连接进行对比，一个普通数据库连接有有限的输入和清晰的语义，而一个流连接可以有无限的输入，并且对于应该连接什么在语义上是不明确的。

每个应用的连接类型是不同的，一些应用使用两个流来连接所有元组——不管经过多长时间，另一些应用希望对于每个连接字段每次连接恰好一个元组。在所有这些连接类型中，常见的模式是以相同的方式划分多个输入流。在 Storm 里对输入流在相同的字段使用字段分组，例如：

```
builder.setBolt("join", new MyJoiner(), parallelism)
    .fieldsGrouping("1", new Fields("joinfield1", "joinfield2"))
    .fieldsGrouping("2", new Fields("joinfield1", "joinfield2"))
    .fieldsGrouping("3", new Fields("joinfield1", "joinfield2"));
```

当然，不同的流不需要有相同的字段名字。

3.7.2　批处理（Batching）

通常因为效率或者其他原因，希望对一组元组进行批处理而不是单独地处理。例如，可能想要批量更新到数据库或者做一些排序的流连接。

如果想要在数据处理时具有可靠性，正确的方法是当 Bolt 等待做批处理时，在一个实例变量中保存元组的引用。一旦你做批处理操作，可以 ack 到你已经保存引用的所有元组。

如果 Bolt 发射元组，那么你可能想要使用多锚来保证可靠性。这一切都取决于具体的应用。

3.7.3　BasicBolt

许多 Bolt 遵循读取一个输入元组类似的模式，根据输入元组发射零个或多个元组，然后在 execute 方法的最后立即 ack 输入元组。此模式相匹配的 Bolt 是一些函数和过滤器之类的东西。这是一个常见的模式，Storm 提供了接口，名称为 IBasicBolt，自动为你实现这种模式。

3.7.4　内存中缓存与字段的组合

在 Storm 的 Bolt 内存中保留缓存，是很常见的做法。当你把缓存和一个字段分组进行合并时，缓存就会变得特别大。例如，假如有一个 Bolt，扩展短 URL 到长 URL，如 bit.ly、t.co 等。你可以保留一个短 URL 到长 URL 的扩展的 LRU 缓存，避免反复做相同的 HTTP 请求，从而提高性能。假设组件 urls 发射短 URL，组件 expand 扩展短 URL 到长 URL 并保留一个内部缓存。在以下代码片段中，可以思考一下两者的区别：

```
builder.setBolt("expand", new ExpandUrl(), parallelism)
    .shuffleGrouping(1);
builder.setBolt("expand", new ExpandUrl(), parallelism)
    .fieldsGrouping("urls", new Fields("url"));
```

第二种方法比第一种方法更有效，因为相同的 URL 总是到相同的任务。这样可以避免重复跨任务中的缓存，使短 URL 更有可能命中缓存。

3.7.5 流的 top N

在 Storm 里，一个常见的持续计算是一些排序的“流的 top N”。假设有一个 Bolt 发出元组["value", "count"]，希望另一个 Bolt 发射出基于统计的 top N 元组。最简单的方法是有一个 Bolt 在流中做全局分组，并且在内存中维护 top N 的列表。

这种方法显然不能扩展到很大的流，因为整个流都要通过一个任务，单任务的计算能力是有限的。一个更好的方法是跨流的分区并行做很多 top N 的计算，然后合并那些 top N 到一起，得到全局的 top N。其模式看起来就像这样：

```
builder.setBolt("rank", new RankObjects(), parallellism)
    .fieldsGrouping("objects", new Fields("value"));
builder.setBolt("merge", new MergeObjects())
    .globalGrouping("rank");
```

这种模式行得通，因为第一个 Bolt 进行字段分组，给出需要的分区，这在语义上是正确的。这种模式在 storm-starter 中的一个例子如下：

```
package storm.starter;

import backtype.storm.Config;
import backtype.storm.testing.TestWordSpout;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import storm.starter.bolt.IntermediateRankingsBolt;
import storm.starter.bolt.RollingCountBolt;
import storm.starter.bolt.TotalRankingsBolt;
import storm.starter.util.StormRunner;

public class RollingTopWords {

  private static final int DEFAULT_RUNTIME_IN_SECONDS = 60;
  private static final int TOP_N = 5;

  private final TopologyBuilder builder;
  private final String topologyName;
  private final Config topologyConfig;
  private final int runtimeInSeconds;
```

```
    public RollingTopWords() throws InterruptedException {
      builder = new TopologyBuilder();
      topologyName = "slidingWindowCounts";
      topologyConfig = createTopologyConfiguration();
      runtimeInSeconds = DEFAULT_RUNTIME_IN_SECONDS;

      wireTopology();
    }

    private static Config createTopologyConfiguration() {
      Config conf = new Config();
      conf.setDebug(true);
      return conf;
    }

    private void wireTopology() throws InterruptedException {
      String spoutId = "wordGenerator";
      String counterId = "counter";
      String intermediateRankerId = "intermediateRanker";
      String totalRankerId = "finalRanker";
      builder.setSpout(spoutId, new TestWordSpout(), 5);
      builder.setBolt(counterId, new RollingCountBolt(9, 3), 4).fieldsGrouping
(spoutId, new Fields("word"));
      builder.setBolt(intermediateRankerId, new IntermediateRankingsBolt(TOP_
N), 4).fieldsGrouping(counterId, new Fields("obj"));
      builder.setBolt(totalRankerId, new TotalRankingsBolt(TOP_N)).globalGrou
ping(intermediateRankerId);
    }

    public void run() throws InterruptedException {
      StormRunner.runTopologyLocally(builder.createTopology(), topologyName,
topologyConfig, runtimeInSeconds);
    }

    public static void main(String[] args) throws Exception {
      new RollingTopWords().run();
    }
  }
```

3.7.6 高效保存最近更新缓存对象的 TimeCacheMap（已弃用）

有时程序员会希望在内存中保持最近“活跃”的缓存对象，而已经不活动的对象在一段时间后会自动到期。TimeCacheMap 是实现这种功能的一种高效数据结构，它提供了钩子，这样当一个对象过期失效，可以插入回调函数。

TimeCacheMap 用于清除在配置的秒数内没有更新的到期主键。使用该算法将花费 expirationSecs 到 expirationSecs×(1 + 1 / (numBuckets-1))之间的时间来实际清除到期的消息。运行 get、put、remove、containsKey、size 操作只花费 O(numBuckets)的时间。这种设计的优点是，过期线程只锁定对象 O(1)时间，意味着对象本质上总可以进行 get/put 操作。

目前，该类 backtype.storm.utils.TimeCacheMap<K,V>已经弃用。

3.7.7 分布式 RPC 的 CoordinatedBolt 与 KeyedFairBolt

在 Storm 上构建分布式 RPC 应用有两种常见的模式，它们封装在 CoordinatedBolt 和 KeyedFairBolt 中，属于 Storm 代码库附带的“标准库”的一部分。

CoordinatedBolt 封装了包含你的逻辑的 Bolt，当你的 Bolt 已收到所有元组，就会为任何给定的请求计算出结果。它大量使用直接流（Direct Stream）来做到这一点。

KeyedFairBolt 也封装了包含你的逻辑的 Bolt，并且保证你的拓扑可以同时处理多个 DRPC 调用，而不是串行地一次处理一个。

3.8 本地模式与 StormSubmitter 的对比

现在，已经使用一个名为 LocalCluster 的工具在本地计算机上运行 Topology。在计算机上运行 Storm 基础设施，可以很容易地运行与调试不同的 Topology。但如果你想要提交你的 Topology 到运行中的 Storm 集群呢？Storm 的一个有趣特性是，它很容易发送你的 Topology 去运行在一个真正的集群中。你需要做的是将 LocalCluster 改为 StormSubmitter，实现 submitTopology 方法，submitTopology 方法负责发送 Topology 到集群。

可以在下面的代码中看到变化：

```
// 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();
```

当使用 StormSubmitter 时，不能在代码中控制集群，这和 LocalCluster 是不一样的。

接下来，需要打包源代码到一个 jar 文件中。当运行 Storm 客户端命令提交 Topology 时，会发送该 jar 文件。如果你使用 Maven，唯一需要做的就是到源代码文件夹下运行以下命令：

```
mvn package
```

一旦生成了 jar 文件，就可以使用 storm jar 命令来提交 Topology。语法如下：

```
Storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3
```

在这个例子中，在 Topology 源代码项目文件夹下运行如下命令：

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

使用完这些命令，就会提交 Topology 到集群中。

为了停止或者杀死 Storm，可以运行如下命令：

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

Topology 的名字必须具有唯一性。

本地模式（Local mode）

本地模式在进程中模拟了一个 Storm 集群，用于开发和测试 Topology。在本地模式下运行 Topology 类似于在集群上运行 Topology。

只需使用 LocalCluster 类就可以创建一个进程内的集群，例如：

```
import backtype.storm.LocalCluster;

LocalCluster cluster = new LocalCluster();
```

然后，可以使用 LocalCluster 对象的 submitTopology 方法来提交 Topology。就像在 StormSubmitter 中相应的方法一样，submitTopology 方法需要一个名字、一个 Topology 配置和 Topology 对象。然后，你可以使用 killTopology 方法，将 Topology 名称作为参数，杀死一个 Topology。

关闭一个本地集群，只需要简单地调用：

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

1. 常见的本地模式的配置

acktype.storm.Config 类用来配置 Storm，它的继承关系如下：

```
java.lang.Object
  ┗java.util.AbstractMap<K,V>
```

```
        └java.util.HashMap<java.lang.String,java.lang.Object>
           └backtype.storm.Config
    All Implemented Interfaces:
    java.io.Serializable, java.lang.Cloneable, java.util.Map<java.lang.String,
java.lang.Object>
```

2. Config.TOPOLOGY_MAX_TASK_PARALLELISM

这个配置项是组件产生线程数量的上限。通常生产环境的拓扑并行度很大（数以百计的线程），可以尝试在本地模式下测试拓扑，找出不合理负荷的地方。这个配置项使你可以很容易地控制并行度。

3. Config.TOPOLOGY_DEBUG

当设置为 true 时，每次从 Spout 或者 Bolt 发送元组，Storm 都会写进日志，这对于调试程序是非常有用的。

3.9 多语言协议（Multi-Language Protocol）

本节将解释 Storm 多语言协议，适用于 Storm 0.7.1 及以后的版本，0.7.1 之前的版本使用不同的协议，有兴趣的读者可以参考下列链接中的英文原文：

```
    https://github.com/nathanmarz/storm/wiki/Storm-multi-language-protocol-
(versions-0.7.0-and-below)
```

支持多种语言是通过 ShellBolt、ShellSpout、ShellProcess 类实现的，这些类实现了 IBolt、ISpout 接口，可使用 Java 的 ProcessBuilder 类调用 shell 脚本或程序执行协议。

输出字段（Output field）是 Thrift 定义拓扑的一部分。这意味着当你在 Java 中使用多语言时，需要创建一个继承 ShellBolt 实现 IRichBolt 接口的 Bolt，并且在该 Bolt 的 declareOutputFields 方法中声明字段。ShellSpout 也是一样。

通过 STDIN 和 STDOUT 执行脚本或程序，可以实现一个简单的协议。所有处理的数据交换都是以 JSON 编码的，因此支持几乎任何可能的语言。

在集群上运行一个 Shell 组件提交给主控节点，shell 脚本必须在 jar 包的 resources 目录中。

但是，如果是在本地机器上进行开发或测试，只需要 resources 目录在类路径（classpath）中即可。

1. 协议（Protocol）

协议的两端用行读机制，所以一定要从输入去掉换行符，并把它们添加到输出。

所有 JSON 的输入输出被终止，通过单行含有“end”。请注意，此分隔符本身不是 JSON

编码。

2. 初始握手（Initial Handshake）

初始握手对于这两种类型的 shell 组件是相同的：

STDIN：设置信息。这是一个包含 Storm 配置、拓扑上下文、PID 目录的 JSON 对象。

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

你的脚本应该在这个目录创建一个以 PID 命名的空文件。例如，PID 为 1234，所以在目录中创建一个名为 1234 的空文件。这个文件使 Supervisor 知道 PID，以后它可以用来关闭进程。

STDOUT：PID 的 JSON 对象类似是这样的：

```
{"PID":1234 }
```

shell 组件将记录 PID 到日志中。接下来会发生什么取决于组件的类型。

3. Spouts

Shell Spout 是同步的。其他发生在“while(true)”循环中：

STDIN：next、ack 或者 fail 命令。

“next”是 ISpout 的 nextTuple 相等。它看起来像：

```
{"command": "next"}
```

“ack”看起来像：

```
{"command": "ack", "id": "1231231"}
```

“fail”看起来像：

```
{"command": "fail", "id": "1231231"}
```

STDOUT：Spout 的前面命令的结果。这可以是一个 emit 序列和 log。

“emit”看起来像：

```
    {
        "command": "emit",
        // The id for the tuple. Leave this out for an unreliable emit. The id can
       // be a string or a number.
        "id": "1231231",
        // The id of the stream this tuple was emitted to. Leave this empty to em
it to default stream.
        "stream": "1",
        // If doing an emit direct, indicate the task to send the tuple to
        "task": 9,
        // All the values in this tuple
        "tuple": ["field1", 2, 3]
    }
```

如果不做直接 emit，你将马上在 STDIN 收到发射元组的任务 id 作为 JSON 数组。

“log”会在 Worker 的日志中记录一条消息。它看起来像：

```
    {
        "command": "log",
        // the message to log
        "msg": "hello world!"
    }
```

STDOUT：“sync”命令结束 emit 序列和 log。它看起来像：

```
    {"command": "sync"}
```

在 sync 命令后，ShellSpout 不会读取输出，直到它发送另一个 next、ack 或 fail 命令。

注意，类似于 ISpout，Worker 中所有的 Spout 在 next、ack 或者 fail 命令之后被锁定，直到 sync 命令。对于 ISpout，如果对 next 没有发射元组，你应该在 sync 之前 sleep 少量时间。ShellSpout 不会自动为你 sleep。

4. Bolts

Shell Bolt 协议是异步的。你会在 STDIN 中获得元组只要元组可用，你可以在任何时间 emit、ack、fail 和 log，并写到 STDOUT。

STDIN：一个元组。这是一个 JSON 编码结构，是这样的：

```
    {
       // The tuple's id - this is a string to support languages lacking 64-bit
precision
        "id": "-6955786537413359385",
        // The id of the component that created this tuple
        "comp": "1",
        // The id of the stream this tuple was emitted to
        "stream": "1",
        // The id of the task that created this tuple
        "task": 9,
```

```
        // All the values in this tuple
        "tuple": ["snow white and the seven dwarfs", "field2", 3]
    }
```

STDOUT：ack、fail、emit、log.。

“emit”看起来像：

```
    {
        "command": "emit",
        // The ids of the tuples this output tuples should be anchored to
        "anchors": ["1231231", "-234234234"],
        // The id of the stream this tuple was emitted to. Leave this empty to em
it to default stream.
        "stream": "1",
        // If doing an emit direct, indicate the task to send the tuple to
        "task": 9,
        // All the values in this tuple
        "tuple": ["field1", 2, 3]
    }
```

如果不做直接 emit，你将收到的任务 id，从 STDIN 上发出的元组的任务 id，作为一个 JSON 数组。注意，由于 shell bolt 协议的异步特性，当你在 emit 后读取，你可能接收不到任务 id。然而，你可以为前一次 emit 替换读取任务 id 或者处理一个新的元组。你将收到与相应 emit 相同顺序的任务 id 列表。

“ack”看起来像：

```
    {
        "command": "ack",
        // the id of the tuple to ack
        "id": "123123"
    }
```

“fail”看起来像：

```
    {
        "command": "fail",
        // the id of the tuple to fail
        "id": "123123"
    }
```

“fail”会在 Worker 日志中记录消息。它看起来像：

```
    {
        "command": "log",
        // the message to log
        "msg": "hello world!"
    }
```

Storm 0.7.1 版本不再需要 shell bolt 做“sync”操作。

3.10 使用非 JVM 语言操作 Storm

3.10.1 支持的非 Java 语言

Storm 支持如下的非 Java 的 DSL（Domain Specific Language，领域特定语言）。

- Scala DSL，项目主页为：https://github.com/velvia/ScalaStorm。
- JRuby DSL，项目主页为：http://github.com/colinsurprenant/storm-jruby。
- Clojure DSL，项目主页为：http://github.com/nathanmarz/storm/wiki/Clojure-DSL。
- io-storm，项目主页为：https://github.com/gphat/io-stormPerl。

3.10.2 对 Storm 使用非 Java 语言

对 Storm 使用非 Java 语言分为两部分：使用非 Java 语言创建拓扑以及使用非 Java 语言实现 Spout 和 Bolt。

使用非 Java 创建拓扑是很容易的，因为拓扑是 Thrift 结构的，可以参考 storm.thrift。

使用非 Java 语言实现 Spout 和 Bolt 会调用多语言组件或者 shelling。

这里有一个规范的协议：Multilang protocol。

Thrift 结构允许你显式定义多语言组件作为一个程序和脚本，例如 python 和文件实现你的 Bolt 的文件。

在 Java 中，你会覆盖 ShellBolt 或者 ShellSpout 来创建多语言组件。

注意，输出字段声明（output fields declaration）中发生的 thrift 结构，所以在 Java 中创建多语言组件如下。

在 Java 中声明字段，通过在 ShellBolt 的构造函数中指定它，使用其他语言来处理代码。

多语言在 stdin/stdout 中使用 json 消息与子流程进行通信。

Storm 由 ruby、python 和 fancy 的适配器来实现协议。

python 支持发射（Emitting）、锚定（Anchoring）、确认（Acking）和日志记录（Logging）操作。

“storm shell”命令使得构建 jar 和上传到 Nimbus 变得容易。

可使用 Nimbus 的主机名、端口和 jar 文件的 id 调用你的程序

3.10.3 实现非 Java DSL 的笔记

正确的起点是 src/storm.thrift。因为 Storm 拓扑是 Thrift 结构，Nimbus 是一个 Thrift 守护进程，你可以使用任何语言创建和提交拓扑。

当你为 Spout 和 Bolt 创建 Thrift 结构时，应该在 ComponentObject 结构体中指定 Spout 或者 Bolt。

```
union ComponentObject {
    1: binary serialized_java;
    2: ShellComponent shell;
    3: JavaObject java_object;
}
```

对于一个非 Java 的 DSL，你应该会使用“2”和“3”。ShellComponent 允许你指定一个脚本运行该组件，例如，python 代码。JavaObject 允许你为组件指定本地 java 的 Spout 和 Bolt，Storm 将使用反射来创建 Spout 或者 Bolt。

“storm shell”命令将帮助你提交一个拓扑。它的用法如下：

```
storm shell resources/ python topology.py arg1 arg2
```

storm shell 然后会把“resources/”打包到 jar 文件，并上传 jar 到 Nimbus。调用你的 Topology.py 的脚步命令如下：

```
python topology.py arg1 arg2 {nimbus-host} {nimbus-port} {uploaded-jar-location}
```

然后，可以使用 Thrift API 连接到 Nimbus，提交拓扑，传送{uploaded-jar-location}到 submitTopology 方法。下面是 submitTopology 的定义。

```
void submitTopology(1: string name, 2: string uploadedJarLocation, 3: string
 jsonConf, 4: StormTopology topology) throws (1: AlreadyAliveException e, 2: Inv
alidTopologyException ite);
```

3.11 Hook

Storm 提供了 Hook（钩子），使用它可以在 Storm 内部插入自定义代码来运行任意数量的事件。可以通过继承 BaseTaskHook 类创建一个 Hook，为要捕获的事件重写适当的方法。有两种方法来注册你的钩子：

- 在 Spout 的 open()方法或者 Bolt 的 prepare()方法中，使用 TopologyContext # addTaskHook。
- 通过在 Storm 的配置中使用 topology.auto.task.hooks 配置。这些钩子在每个 Spout 或者 Bolt 中自动注册，它们对于在自定义的监控系统中进行集成是很有用的。

3.12 本章小结

本章介绍了 Storm 拓扑的相关内容，包括什么是 Storm 拓扑、TopologyBuilder 及其使用、流分组、拓扑定义的例子，以及如何在本地模式与生产集群上运行拓扑、拓扑的常见模式等。