

提供各种IT类书籍pdf下载，如有需要，请 QQ: 2011705918

注：链接至淘宝，不喜者勿入！整理那么多资料不容易，请多多见谅！非诚勿扰！

更多资源请点击

实时

海量

高效

可靠

# Storm实战

## 构建大数据实时计算

|| 阿里巴巴集团数据平台事业部商家数据业务部 编著 ||



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

## 融入淘宝技术团队的丰富实践经验与应用架构 快速掌握Storm技术精髓

Storm被视为一个实时版本的Hadoop，填补了大数据处理生态系统的巨大缺失。随着大规模数据实时处理需求的强劲增长，Storm的重要性日益凸显，并被越来越多的公司所采用。

非常高兴听到用户对Storm赞誉有加：简单，灵活，极其健壮。这正是Storm所追求的一些核心设计目标。

Nathan Marz, Storm之父

Storm是一个开源的分布式实时计算系统。它简单有趣，可以用任何编程语言来使用。

Storm可以适用于很多场景：实时分析，在线机器学习，持续计算，分布式RPC，ETL等。

Storm速度惊人：每个节点每秒能处理超过百万条tuples。扩展性强，容错率高，能保证数据的及时处理，而且很容易构建和操作。

Storm集成了常用的队列和数据库技术。Storm的拓扑能够以任意复杂的方式消费和处理数据流，也可以在计算的各个阶段重新分流。

Storm官网



博文视点 Broadview



新浪微博  
weibo.com  
@博文视点Broadview



策划编辑：刘皎 [@皎丫子](#)

责任编辑：徐津平

封面设计：侯士卿

欢迎投稿

Ljiao@phei.com.cn

010-88254395

上架建议：大数据

ISBN 978-7-121-22649-6



9 787121 226496 >

定价：59.00元

## 内 容 简 介

本书是一本系统并且具有实践指导意义的Storm工具书和参考书，对Storm整个技术体系进行了全面的讲解，不仅包括对基本概念、特性的介绍，也涵盖了一些原理说明。本书的实战性很强，各章节都提供了一些小案例，同时对于本地，以及集群环境的部署有详细介绍，易于理解，操作性强。

全书一共10章：第1章全面介绍了Storm的特性、能解决什么问题，以及和其他流计算系统的对比；第2章通过实际运行一个简单的例子，以及介绍本地环境和集群环境的搭建，让读者对Storm有了直观的认识；第3章深入讲解了Storm的基本概念，同时实现一个Topology运行；第4章和第5章阐述了Storm的并发度、可靠处理的特性；第6章~第8章详细而系统地讲解了几个高级特性：事务、DRPC和Trident；第9章以实例的方式讲解了Storm在实际业务场景中的应用；第10章总结了几个在大数据场景应用过程中遇到的经典问题，以及详细的排查过程。

本书既适合没有Storm基础的初学者系统地学习，又适合有一定Storm基础但是缺乏实践经验的读者实践和参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

### 图书在版编目 (CIP) 数据

Storm 实战：构建大数据实时计算 / 阿里巴巴集团数据平台事业部商家数据业务部编著. —北京：电子工业出版社，2014.8

（大数据丛书. 阿里巴巴集团技术丛书）

ISBN 978-7-121-22649-6

I. ①S… II. ①A… III. ①数据处理 IV. ①TP274

中国版本图书馆 CIP 数据核字(2014)第 050578 号

策划编辑：刘 豪

责任编辑：徐津平

印 刷：三河市鑫金马印装有限公司

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：900×1280 1/32 印张：5.75 字数：91千字

版 次：2014年8月第1版

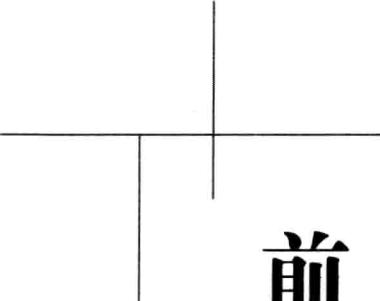
印 次：2014年8月第1次印刷

印 数：4000册 定价：59.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。



# 前 言

## 实时流计算

互联网从诞生的一刻起，对世界的最大改变就是让信息能够实时交互，从而大大提高各个环节的效率。正因为大家对信息实时响应、实时交互的需求，软件行业除了个人操作系统之外，数据库（更精确地说是关系型数据库）应该是软件行业发展最快、收益最为丰厚的产品了。记得 20 世纪 90 年代，很多银行别说实时转账，连实时查询都做不到，但是数据库和高速网络改变了这个情况。

SNS: Social Network Service

互联网的进一步发展，从 Portal 信息浏览型到 Search 信息搜索型再到 SNS 关系交互传递型，以及电子商务、互联网旅游生活产品等，将人们生活中的流通环节在线化。对效率的要求让大家对实时性的要求进一步提升，而信息的交互和沟通正在从点对点向信息链甚至信息网的方向发展，这样必然带来数据在各个维度的交叉关联，数据爆炸已不可避免。因此流式处理加NoSQL 产品应运而生，分别解决实时处理框架和数据大规模存储计算的问题。

早在 2000 年初，诸如 UC 伯克利、斯坦福等大学就开始了对流式数据处理的研究，但是由于更多地关注于金融行业的业务场景或者互联网流量监控的业务场景，以及当时互联网数据场景的限制，造成了研究多是基于对传统数据库处理的流式化，对流式框架本身的研究偏少。目前这样的研究逐渐没有了声音，工业界将更多的精力转向了实时数据库。

2010 年 Yahoo! 对 S4 的开源，2011 年 Twitter 对 Storm 的开源，改变了这个现状。以前互联网的开发人员在做一个实时应用的时候，除了要关注应用逻辑计算本身，还要

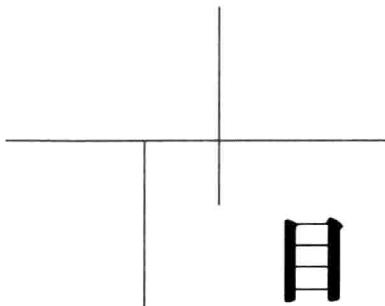
为数据的实时流转、交互、分布大伤脑筋。现在的情况却大为不同，以 Storm 为例，开发人员可以快速搭建一套健壮、易用的实时流处理框架，配合 SQL 产品或者 NoSQL 产品或者 MapReduce 计算平台，就可以以低成本做出很多以前很难想象的实时产品，比如量子恒道品牌旗下的多个产品就是构建在 Storm 实时流处理平台上的。

本书是一本对 Storm 的基础介绍手册，但是我们也希望它不仅仅是一本 Storm 的使用手册，我们会在其中加入更多在实际数据生产过程中的经验和应用架构，最终的目的是帮助所有愿意使用实时流处理框架的技术同仁，同时也默默地改变这个世界。

在本书即将出版之际，Storm 已经成功发布了 0.9.0 版本，追加了一些新的特性，如使用 Netty 作为新的消息传输层、提供日志查看 UI 等，同时修复了大量跟稳定性相关的 BUG。本次发布对茁壮成长的 Storm 来说是一次巨大的进步。新版本的 Storm 在系统结构及使用方式方面，并没有太多变化，本书可以帮助你快速掌握应用 Storm 的知识和技能。

本书由团队中多位同学先后参与合作完成，为体现阿里巴巴的文化，这里列出所有涉及同学的花名：张中、太奇、鸣世、曦轩、鸣珂、民瞻、九翎、渊虹、国相、晨炫、木晗、毅山、宋智、澄苍，是大家的合作与努力才让此书得以成行。同时感谢刘皎等编辑的辛苦工作，是你们把这本书呈献给读者，感谢你们！

同样要感谢支持我们工作的同事们：冰夷、王贲，有你们的帮助和支持才让我们有决心和毅力来完成这项工作。



# 目 录

---

## 第 1 章 Storm 基础 1

1.1	Storm 能做什么 .....	2
1.2	Storm 特性 .....	3
1.3	其他流计算系统 .....	8
1.4	应用模式 .....	13

---

## 第 2 章 Storm 初体验 17

2.1	本地环境搭建 .....	18
2.2	Storm 集群 .....	25

---

## 第 3 章 构建 Topology

41

3.1	Storm 基本概念	42
3.2	构建 Topology	53
3.3	小结	61

---

## 第 4 章 Topology 的并行度

62

4.1	并行元素	63
4.2	配置并行度	65
4.3	一个运行中 Topology 的例子	68
4.4	如何更新运行中的 Topology 的并行度	71

---

## 第 5 章 消息的可靠处理

73

5.1	简介	74
5.2	理解消息被完整处理	74
5.3	消息的生命周期	76
5.4	可靠相关的 API	79
5.5	高效地实现 tuple tree	84

5.6 选择合适的可靠性级别 .....	87
5.7 集群的各级容错 .....	89
5.8 小结 .....	91
<b>第 6 章 一致性事务</b>	<b>92</b>
6.1 简单设计一：强顺序流 .....	93
6.2 简单设计二：强顺序 batch 流 .....	95
6.3 CoordinateBolt 的原理 .....	96
6.4 Transactional Topology .....	98
<b>第 7 章 DRPC</b>	<b>105</b>
7.1 Storm DRPC .....	106
7.2 总体概述 .....	106
7.3 LinearDRPCTopologyBuilder .....	108
7.4 本地模式 DRPC .....	110
7.5 远程模式 DRPC .....	111
7.6 一个复杂的例子 .....	113

7.7 非线性 DRPC 拓扑 .....	117
7.8 LinearDRPCTopologyBuilder 工作过程 .....	117
7.9 高级进阶 .....	118

---

第 8 章 Trident 的特性 119

8.1 理解 Trident .....	120
8.2 结合多个 Trident 任务 .....	124
8.3 消费和生产 Field .....	126
8.4 State（状态保存） .....	128
8.5 Trident Topology 的执行过程 .....	136
8.6 总结 .....	137

---

第 9 章 Storm 实例 138

9.1 一个简单的实例 .....	139
9.2 复杂一点的实例 .....	150
9.3 其他 .....	161

## 第 10 章 常见应用问题分析

162

10.1	<u>性能问题排查与定位</u>	163
10.2	系统中常见的问题与排查	167
10.3	业务问题的定位与排查	170

# 第1章

## Storm 基础

## 1.1 Storm 能做什么

HDFS:Hadoop Distributed File System

在大数据处理方面，相信大家已经对 Hadoop 耳熟能详了，Hadoop 处理的是存放在其分布式文件系统 HDFS 上的数据，Hadoop 使用磁盘作为中间交换的介质，在对海量数据进行离线分析时得心应手，但处理实时数据流却是力有未逮。

Storm 是一个开源的分布式实时计算系统，可以简单、可靠地处理大量的数据流。Storm 有很多使用场景，如实时分析、在线机器学习、持续计算、分布式 RPC、ETL，等等。Storm 支持水平扩展，具有高容错性，保证每个消息都会得到处理，而且处理速度很快（在一个小集群中，每个节点每秒可以处理数以百万计的消息）。Storm 的部署和运维都很便捷，而且更为重要的是可以使用任意编程语言来开发应用。

## 1.2 Storm 特性

Storm 有如下特点。

### 1. 编程模型简单

基于 Google Map/Reduce 来实现的 Hadoop 为开发者提供了 map、reduce 原语，使并行批处理程序变得非常简单和优美。同样，Storm 也为大数据的实时计算提供了一些简单优美的原语，这大大降低了开发并行实时处理任务的复杂性，帮助你快速、高效的开发应用。

### 2. 可扩展

在 Storm 集群中真正运行 Topology 的主要有三个实体：工作进程、线程和任务。Storm 集群中的每台机器上都可以运行多个工作进程，每个工作进程又可创建多个线程，每个线程可以执行多个任务，任务是真正进行数据处理的实体，Spout、Bolt 被开发出来就是作为一个或者多个任务的方式执行的。



因此，计算任务在多个线程、进程和服务器之间并行进行，支持灵活的水平扩展。

### 3. 高可靠性

Storm 可以保证 Spout 发出的每条消息都能被“完全处理”，这也是直接区别于其他实时系统的地方，如 Yahoo! S4。

请注意，Spout 发出的消息后续可能会触发产生成千上万条消息，可以形象地理解为一棵消息树 其中 Spout 发出的消息为树根，Storm 会跟踪这棵消息树的处理情况，只有当这棵消息树中的所有消息都被处理了，Storm 才会认为 Spout 发出的这个消息已经被“完全处理”。如果这棵消息树中的任何一个消息处理失败了，或者整棵消息树在限定的时间内没有“完全处理”，那么 Spout 发出的消息就会重发。

考虑到尽可能减少对内存的消耗，Storm 并不会跟踪消息树中的每个消息，而是采用了一些特殊的策略，它把消息树当作一个整体来跟踪，对消息树中所有消息的唯一 id 进行异或计算，通过是否为零来判定 Spout 发出的消息是否被“完全处理”，这极大地节约了内存和简化了判定逻辑，后面会对

这种机制进行详细介绍。

这种模式，每发送一个消息，都会同步发送一个 ack 或 fail，对于网络的带宽会有一定的消耗，如果对于可靠性要求不高，可通过使用不同的 emit 接口关闭该模式。

上面所说的，Storm 保证了每个消息至少被处理一次，但是对于有些计算场合，会严格要求每个消息只被处理一次，幸而 Storm 的 0.7.0 版引入了事务性拓扑，解决了这个问题，本书后面会有详述。

#### 4. 高容错性

如果在消息处理过程中出现了一些异常，Storm 会重新部署这个出问题的处理单元。Storm 保证一个处理单元永远运行（除非你显式的结束这个处理单元）。

当然，如果处理单元中存储了中间状态，那么当处理单元重新被 Storm 启动时，需要应用自己处理中间状态的恢复。

## 5. 支持多种编程语言

除了用 Java 实现 Spout 和 Bolt，你还可以使用任何你熟悉的编程语言来完成这项工作，这一切得益于 Storm 所谓的多语言协议。多语言协议是 Storm 内部的一种特殊协议，允许 Spout 或者 Bolt 使用标准输入和标准输出来进行消息传递，传递的消息为单行文本或者是 JSON 编码的多行。

Storm 支持多语言编程主要是通过 ShellBolt、ShellSpout 和 ShellProcess 这些类来实现的，这些类都实现了 IBolt 和 ISpout 接口，以及让 Shell 通过 Java 的 ProcessBuilder 类来执行脚本或者程序的协议。

可以看到，采用这种方式，每个 Tuple 在处理的时候都需要进行 JSON 的编解码，因此在吞吐量上会有较大影响。

## 6. 支持本地模式

Storm 有一种“本地模式”，也就是在进程中模拟一个 Storm 集群的所有功能，以本地模式运行 Topology 跟在集群上运行 Topology 类似，这对于我们开发和测试来说非常有用。

## 7. 高效

用 ZeroMQ 作为底层消息队列，保证消息能快速被处理。

## 8. 运维和部署简单

Storm 计算任务是以“拓扑”为基本单位的，每个拓扑完成特定的业务指标，拓扑中的每个逻辑业务节点实现特定的逻辑，并通过消息相互协作。

实际部署时，仅需要根据实际情况配置逻辑节点的并发数，而不需要关心部署到集群中的哪台机器。所有部署仅需通过命令提交一个 jar 包，全自动部署。停止一个拓扑，也只需通过一个命令操作。

Storm 支持动态增加节点，新增节点自动注册到集群中，但现有运行的任务不会自动负载均衡。

## 9. 图形化监控

图形界面，可以监控各个拓扑的信息，包括每个处理单元的状态和处理消息的数量。

## 1.3 其他流计算系统

这里主要将 Yahoo! S4 和 IBM InfoSphere Streams 与 Storm 进行对比。

以易用性（开发效率）、性能、通用性为基准，从系统模型、应用开发环境、系统性能、高可用性和现有代码的复用等方面进行对比。

列出对某些功能的支持与否并不表示此产品本身的优势，首先应该考虑的是在特定场景下这些功能是否有必要。

### 1.3.1 Yahoo! S4

Yahoo! S4 (Simple Scalable Streaming System) 是一个通用的、分布式的、可扩展的、分区容错的、可插拔的流式系统，基于开源协议 Apache License 2.0。

## 1. 系统模型

S4 系统由多个处理单元（Processing Elements，PEs）相互配合进行计算，PE 之间通过消息的形式传递，PE 消费事件，同时发出一个或多个可能被其他 PE 处理的事件，或者直接产出结果。

通过把任务分解为尽可能小的处理单元，各处理单元之间形成流水线，从而提高并发度和吞吐量。各处理单元的粒度及逻辑均由开发者自行掌握，所以并不能保证各处理单元分解得足够合理，进而影响并发性。这点与 Storm 的方式一样。

与 Storm 不同的是，S4 内置的 PE 还可以处理 count、join 和 aggregate 等常见任务需求。

## 2. 开发

S4 系统使用 Java 开发，采用了极富层次的模块化编程，每个通用功能点都尽量抽象出来作为通用模块，而且尽可能让各模块实现可定制化。

### 3. 通信协议

S4 节点间通信采用 “Plain Old Java Objects” (POJOs) 模式，应用开发者不需要写 Schemas 或用哈希表在节点间发送消息元组 (tuples)。但底层通信协议采用 UDP，在要求一定可靠性的系统中，这点颇受诟病。

### 4. 高可用

S4 集群中的所有处理节点都是等同的。这种架构将使得集群的扩展性很好，处理节点的总数理论上无上限；同时，S4 没有单点的问题。一旦处理单元崩溃，可以转而利用可用的其他处理单元继续处理，但和 Storm 一样，处理单元中的状态数据无法依靠系统本身恢复，需要借助外部的系统。

### 5. 运维与部署

S4 不支持动态部署，也不支持动态增删节点。这点 Storm 都已支持。

### 1.3.2 IBM InfoSphere Streams

IBM 的商业化的实时流处理系统，有较为完善的 IDE 支持，以及自定义业务描述语言。

#### 1. 系统模型

IBM InfoSphere Streams 同样也是把任务分解为尽可能小的处理单元，各处理单元之间形成流水线，从而提高并发度和吞吐量。不同的是，各处理单元只能完成预定的操作，这些操作组合起来完成一个整体的功能。从机制上保证了处理单元的粒度，有助于系统整体性能的提升。

系统由很多 PE 节点组成，PE 仅仅实现规定的一些操作（如 join、merge 等），强制限定了每个处理单元的粒度，从而可以提高每个单元的处理速度，使得流水线更流畅。多个 PE 可能集成到一个线程中，降低了系统中线程的数量。在设计阶段无须关心 PE 与主机的对应关系，此关系在运行阶段根据集群的配置自动部署。

## 2. 开发

定制的开发环境 Eclipse-SPL (Stream Programming Language)。为流处理定制的 SPL 语言可以简洁地描述出整个系统的业务，也可以使用其他语言（如 C++）来定制特定的流处理模块（operator）。

## 3. 运维与部署

部署半自动化，程序设计、编译阶段无须指定主机信息。只需指定约束关系（如 PE1 与 PE2 不能再同一台主机上），在拓扑部署的时候根据集群的实际情况和预定的约束来确定 PE 在哪台主机上执行。

动态增加节点，新节点自动注册到集群中，与 Storm 不同的是，InfoSphere Streams 可以将现有业务根据负载自动均衡。而 Storm 已经运行的业务不会自动负载均衡。

## 4. 高可用

InfoSphere Streams 与 Storm、S4 类似，可把失败节点上的任务自动转移到其他可用节点上，但同样也不保证数据的恢复。

## 1.4 应用模式

### 1. 海量数据处理

Storm 由于其高效、可靠、可扩展、易于部署、高容错及实时性高等特点，对于海量数据的实时处理非常合适。例如，对于统计网站的页面浏览量（如 Page View，简称 PV）指标，Storm 可以做到实时接收到点击数据流，并实时计算出结果。

### 2. 中间状态存储与查询

这里的中间状态分为两种，一种指的是 Storm 处理流数据实时计算出的结果，是在实时、快速地变化着的。Storm 提供了实时处理数据流的平台，但是并未提供现成的取得实时处理结果的接口，查询这些实时处理的结果，即所谓的中间状态，就需要 Storm 与一些存储服务相结合，比如 MySQL 和 HBase。可以将 Storm 实时计算的中间结果实时地写入 MySQL 或者 HBase，用户就能直接通过数据库的接口取得实

时的结果了。

Storm 处理单元中存储的中间状态可以不单单是计算结果，还可以是计算逻辑类的快照或者“还原点”，这样有一个好处就是错误恢复。Storm 的容错机制确保了如果一个处理单元崩溃，则重启一个处理单元继续处理该数据流。这意味着 Storm 中每个处理单元的处理逻辑应该是无状态的，这样每次重启后，依然能基本正确地处理业务逻辑。但是，对于大部分计算场合，譬如 PV 这样的累计指标，如果没有中间状态，一旦重启，这个处理单元就会重新从 0 开始累计 PV，显然这样的结果是不正确的。因此，如果利用 MySQL 或者 HBase 实时存储处理的中间状态，就可以减少处理单元崩溃后的损失，从最近的中间状态恢复。

另外，Storm 错误恢复还需要数据源的配合，处理单元恢复后能够从数据源继续读取尚未处理的数据处理，或者跳过，前进一定的跨度继续处理。Storm 本身提供了与 kestrel 队列交互的 Spout 来支持这些特性。在实践中，也可以将数据实时写入其他存储介质，如 HBase，然后由 Storm 实时读

取 HBase 的数据，利用 HBase 的特性支持数据的前后定位。

### 3. 数据增量更新

在实际业务中，PV 这种指标的计算，以 HBase 为例，最简单的想法是每次增加都实时修改 HBase。容错策略是，处理单元崩溃恢复后，继续累计 HBase 中的值，但每次计数修改 HBase 的方式，对 HBase 的压力太大。

可以考虑数据在 Storm 内计算短暂的一段时间后，增量地合并到 HBase，以牺牲一定查询结果的实时性换取 HBase 压力的减轻。容错上，每次崩溃，也只是丢失在内存中未合并到 HBase 的那部分计数，尽可能做到崩溃时数据的快速恢复和误差可控。

### 4. 结合概率算法实时计算复杂指标

Storm 实时处理数据，相对于离线处理，需要更多的空间来存储中间状态。对于复杂的指标，可能中间状态的存储最后成为瓶颈，导致内存无法容下。业务上对于实时计算的指标，有时并不需要完全精确，因此，可以利用一些概率算

法来解决这种问题。

例如，对于网站来说访客数（Unique Visitor, UV）指标是指某一时段访问的访客数量，需要对访客的唯一标识去重并计数才能算出。这样，就需要存储从开始计数到现在所有访客的唯一标识，用以去重计算。在访问量巨大、指标交叉维度繁多时，很容易形成瓶颈。采用 Hyper LogLog 这样的概率算法，只需对 UV 指标存储一个位图信息，就能估计出 UV 值，而位图的大小取决于算法需要达到的精度，可以根据业务调节。对于所有类似去重的指标，都可以采用这样的概率算法。

## 第2章

# Storm 初体验

## 2.1 本地环境搭建

由于 Storm 的分布式特性，用户提交的 Topology 可能会分布到多台物理机器上运行，这对 Topology 的开发、测试和调试造成了一定困难。Storm 提供本地集群机制，允许用户将 Topology 提交到本地集群，而且所有的 Bolt、Spout 都运行在一个进程中，能很方便地对 Topology 进行调试。

### 2.1.1 环境准备

#### Eclipse 的环境准备

Eclipse 需要安装 Maven 插件，即 m2e。可在 Eclipse 主界面菜单中单击 Help→About Eclipse 按钮，打开 About Eclipse 对话框，查看是否已安装 m2e 插件。

如果未安装，可通过单击 Help→Eclipse Marketplace... 按钮，找 Maven Integration for Eclipse 选项进行安装。

## 2.1.2 Storm jar 下载、配置

### 1. 本地下载 jar 包

Storm 的官方主页为 <http://storm-project.net/>，在此处可  
下载最新的 jar 包。

源代码地址为 <https://github.com/nathanmarz/storm>，此处  
可获取最新代码，从源代码编译 jar 包（注：目前 Storm 已经  
成为 Apache 的一个孵化项目，官方的 git 仓库由 Apache 维  
护，github 上有一个镜像，参见链接 <https://github.com/apache/incubator-storm>）。

获取 jar 包后，在 Eclipse 的 Package Explorer 项目中单  
击鼠标右键，选择 Properties 按钮，在弹出的对话框中选择  
Java Build Path 选项，再选择 Libraries 选项卡，通过 Add  
External JARs...选项将 Storm jar 文件加入编译路径。

### 2. 使用 Maven

如果你的项目是 Maven 项目，可以简单地在 pom.xml  
文件中添加对 Storm jar 的依赖。

推荐使用 Maven 项目进行 Storm 开发。

在你的 pom.xml 中添加以下代码：

```
<repository>
    <id>clojars.org</id>
    <url>http://clojars.org/repo</url>
</repository>
<dependency>
    <groupId>storm</groupId>
    <artifactId>storm</artifactId>
    <version>0.7.2</version>
</dependency>
```

### 2.1.3 运行一个简单的实例

运行一个本地集群实例很简单，只需要把 Topology 提交给 backtype.storm.LocalCluster 类的对象就可以了。用户还可以像调试普通 Java 程序一样在本地集群中调试 Blot 和 Spout。

storm-starter 项目包含一系列简单的 Storm 实例代码，下面介绍如何利用 storm-starter 来运行本地集群，下文假设读者使用 Linux、Eclipse 和 Maven。

首先，获取 storm-starter 代码，在命令行运行：

```
$ git clone https://github.com/nathanmarz/storm-starter.git
```

在命令行运行经典的 WordCount 程序：

```
$ cd storm-starter  
$ mvn -f m2-pom.xml compile exec:java -Dstorm.topology= \  
  > storm-starter.WordCountTopology
```

将 storm-starter 项目代码导入到 Eclipse IDE 中：

```
$ cd storm-starter  
$ git checkout 0.7.0  
$ mv m2-pom.xml pom.xml
```

打开 Eclipse，单击菜单中 File→Import 选项，选择获取的 storm-starter 目录。

一直单击 Next 按钮，直到最后单击 Finish 按钮，完成导入。

找到 src/jvm/storm/WordCountTopology.java，右键单击 Run As→Java Application 选项，即可以在 Eclipse 中运行 WordCount 实例。

注意，在运行 storm-starter 的过程中可能会遇到以下问题。

1) twitter4j-core 和 twitter4j-stream 这两个包无法下载。

这是由于网络原因导致 twitter Maven 仓库无法使用。

解决方法：从 Maven 中央仓库手动下载这两个包，注释掉 storm-starter/pom.xml 的<repositories>...</repositories>，然后再试。

2) storm 和 clojure 这两个包的下载速度太慢，难以等待。

这是由于 <http://clojars.org/repo> 这个仓库太慢，国内访问困难，导致 Maven 难以下载。

解决方案如下。

(1) 从官方下载完整的 storm-0.7.0，地址 <https://www.dropbox.com/s/pfz2xsy3om6g9eo/storm-0.7.0.zip>。

(2) 解压 storm-0.7.0.zip 到 storm-starter 目录下。

(3) 将 storm-starter/pom.xml 中 clojure、clojure-contrib、storm 这 3 个包的依赖注释掉。

(4) 手动将 storm-0.7.0 的所有 jar 包加入 Eclipse。在

Package Explorer 视图中找到 storm-starter 工程，右键单击 Properties→Java Build Path→Libraries→Add Library 选项，单击 User Library→Next→User Libraries→New 按钮新建一个 Library，起名为 storm070，然后选择 storm070，单击 Add External JARs... 按钮，将 storm-starter/storm-0.7.0/lib 和 storm-starter/storm-0.7.0/下的所有 jar 文件加入其中。

3) 可以运行，但产生名为“when launching multilang subprocess”的异常。

这是由于 storm-starter/multilang/resuources 未导入工程，或者 Python 版本过高导致。

解决方案如下。

(1) 命令行执行 python –version，查看 Python 是否为 3.x 版本，如果是，需要安装 Python 2.x 版本替换。

(2) 检查 storm-starter/multilang/resuources 是否导入工程，如果没有，需要手动 import 此文件夹到工程中。

最后，如果正确运行了 storm.starter. WordCountTopology，将会有类似于如下代码的输出：

```
8370 [Thread-30] INFO backtype.storm.daemon.task - Emitting: split default ["am"]
8370 [Thread-19] INFO backtype.storm.daemon.task - Emitting: count default [1, 50]
8370 [Thread-30] INFO backtype.storm.daemon.task - Emitting: split default ["at"]
8370 [Thread-17] INFO backtype.storm.daemon.task - Emitting: count default [am, 49]
8370 [Thread-28] INFO backtype.storm.daemon.task - Emitting: split default ["at"]
8370 [Thread-17] INFO backtype.storm.daemon.task - Emitting: count default [am, 50]
8370 [Thread-30] INFO backtype.storm.daemon.task - Emitting: split default ["two"]
8370 [Thread-19] INFO backtype.storm.daemon.task - Emitting: count default [at, 49]
8370 [Thread-28] INFO backtype.storm.daemon.task - Emitting: split default ["two"]
8371 [Thread-19] INFO backtype.storm.daemon.task - Emitting: count default [at, 50]
8371 [Thread-30] INFO backtype.storm.daemon.task - Emitting: split default ["with"]
8371 [Thread-21] INFO backtype.storm.daemon.task - Emitting: count default [two, 49]
8371 [Thread-28] INFO backtype.storm.daemon.task - Emitting: split default ["with"]
```

运行大概会持续 10 秒钟，然后退出。storm-starter 可以帮助我们学习使用 Storm 和利用本地集群模式进行调试。

#### 2.1.4 本地集群原理

本地集群模式通过多线程来运行 Topology，把 Spout 和 Blot 集中到一个进程的多个线程中执行，来模拟真实的运行情况。

在本地集群模式中，许多耗时的工作都使用 Thread.sleep 方法模拟，有时会导致运行速度过慢，这点需要注意。

## 2.2 Storm 集群

### 2.2.1 Storm 集群组件

Storm 集群中包含两类节点：主控节点（Master Node）和工作节点（Worker Node）。它们对应的角色如下。

- (1) 主控节点上运行一个被称为 Nimbus 的后台程序，它负责在 Storm 集群内分发代码，分配任务给工作机器，并负责监控集群运行状态。Nimbus 的作用类似于 Hadoop 中 JobTracker 的角色。
- (2) 每个工作节点上运行一个被称为 Supervisor 的后台程序。Supervisor 负责监听从 Nimbus 分配给它执行的任务，据此启动或停止执行任务的工作进程。每一个工作进程执行一个 Topology 的子集，一个运行中的 Topology 由分布在不同工作节点上的多个工作进程组成。

Nimbus 和 Supervisor 节点之间所有的协调工作是通过 Zookeeper 集群来实现的，如图 2-1 所示。

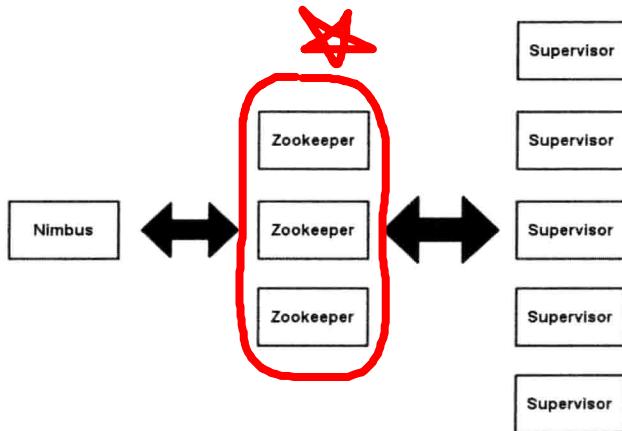


图 2-1

此外，Nimbus 和 Supervisor 进程都是快速失败 (fail-fast) 和无状态 (stateless) 的；Storm 集群中所有的状态要么在 Zookeeper 集群中，要么存储在本地磁盘上。这意味着你可以用 kill -9 来结束 Nimbus 和 Supervisor 进程，它们在重启后可以继续工作。这个设计使得 Storm 集群拥有不可思议的稳定性。

## 2.2.2 安装 Storm 集群

本节将详细描述如何搭建一个 Storm 集群。下面是需要依次完成的安装步骤。

- 搭建 Zookeeper 集群。
- 安装 Storm 依赖库。
- 下载并解压 Storm 发布版本。
- 修改 storm.yaml 配置文件。
- 启动 Storm 的各个后台进程。

### 1. 搭建 Zookeeper 集群

Storm 使用 Zookeeper 协调集群，由于 Zookeeper 并不用于消息传递，所以 Storm 给 Zookeeper 带来的压力相当小。大多数情况下，单个节点的 Zookeeper 集群足够胜任，不过为了确保故障恢复或者部署大规模 Storm 集群，可能需要更大规模节点的 Zookeeper 集群（对于 Zookeeper 集群，官方推荐的最小节点数为 3 个）。在 Zookeeper 集群的每台机器上完成以下安装部署步骤。

(1) 下载安装 Java JDK，官方下载链接为 <http://java.sun.com/javase/downloads/index.jsp>，JDK 版本为 JDK 6 或以上。

(2) 根据 Zookeeper 集群的负载情况，合理设置 Java 堆的大小，尽可能避免发生 swap，导致 Zookeeper 性能下降。保守起见，4GB 内存的机器可以为 Zookeeper 分配 3GB 最大堆空间。

(3) 下载后解压安装 Zookeeper 包，官方下载链接为 <http://hadoop.apache.org/zookeeper/releases.html>。

(4) 根据 Zookeeper 集群节点的情况，在 conf 目录下创建 Zookeeper 配置文件 zoo.cfg，代码如下。

```
tickTime=2000  
dataDir=/var/zookeeper/  
clientPort=2181  
initLimit=5  
syncLimit=2  
server.1=zoo1:2888:3888  
server.2=zoo2:2888:3888  
server.3=zoo3:2888:3888
```

其中， dataDir 指定 Zookeeper 的数据文件目录，  
server.id=host:port:port，id 是每个 Zookeeper 节点的编号，保

存在 dataDir 目录下的 myid 文件中，zoo1~zoo3 表示各个 Zookeeper 节点的 hostname，第一个 port 是用于连接 leader 的端口，第二个 port 是用于 leader 选举的端口。

(1) 在 dataDir 目录下创建 myid 文件，文件中只包含一行，且内容为该节点对应的 server.id 中的 id 编号。

(2) 启动 Zookeeper 服务。

```
$ java -cp zookeeper.jar:lib/log4j-1.2.15.jar:conf \
> org.apache.zookeeper.server.quorum.QuorumPeerMain zoo.cfg
```

或者

```
$ bin/zkServer.sh start
```

(3) 通过 Zookeeper 客户端测试服务是否可用。

```
$ java -cp zookeeper.jar:src/java/lib/log4j-1.2.15.jar:conf:src/ \
> java/lib/jline-0.9.94.jar \
> org.apache.zookeeper.ZooKeeperMain -server 127.0.0.1:2181
```

或者

```
$ bin/zkCli.sh -server 127.0.0.1:2181
```

注意事项如下。

(1) 由于 Zookeeper 是快速失败的，且遇到任何错误情况，进程均会退出，因此，最好能通过监控程序将 Zookeeper 管理起来，保证 Zookeeper 退出后能被自动重启<sup>1</sup>。

(2) Zookeeper 运行过程中会在 dataDir 目录下生成很多日志文件和快照文件，而 Zookeeper 进程并不负责定期清理或合并这些文件，导致占用大量磁盘空间，因此，需要通过 cron 等方式定期清除没用的日志和快照文件<sup>2</sup>。具体命令格式如下。

```
$ java -cp zookeeper.jar:log4j.jar:conf org.apache.zookeeper \
> .server.PurgeTxnLog <dataDir> <snapDir> -n <count>
```

---

1 具体方法请参考 [http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc\\_supervision](http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc_supervision)

2 具体方法请参考 [http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc\\_maintenance](http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc_maintenance)

## 2. 安装 Storm 依赖库

接下来，需要在 Nimbus 和 Supervisor 机器上安装 Storm 的依赖库，具体步骤如下。

- (1) 安装 ZeroMQ 2.1.7。请勿使用 ZeroMQ 2.1.10 版本，因为该版本的一些严重 BUG 会导致 Storm 集群运行时出现奇怪的问题。少数用户在 ZeroMQ 2.1.7 版本会遇到 "IllegalArgumentException" 的异常，此时将 ZeroMQ 降为 2.1.4 版本可修复这一问题。
- (2) 安装 JZMQ。
- (3) 安装 Java 6。
- (4) 安装 Python 2.6.6。
- (5) 安装 unzip。

以上依赖库的版本是经过 Storm 测试的，Storm 并不能保证在其他版本的 Java 或 Python 库下可运行。

- (1) 安装 ZeroMQ 2.1.7。

## 下载后编译安装 ZeroMQ：

```
$ wget http://download.zeromq.org/zeromq-2.1.7.tar.gz  
$ tar -xzf zeromq-2.1.7.tar.gz  
$ cd zeromq-2.1.7  
$ ./configure  
$ make  
$ sudo make install
```

**注意事项** 如果安装过程报错 uuid 找不到，则通过如下方法安装 uuid 库：

```
$ sudo yum install e2fsprogs -b current  
$ sudo yum install e2fsprogs-devel -b current
```

## (2) 安装 JZMQ。

### 下载后编译安装 JZMQ：

```
$ git clone https://github.com/nathanmarz/jzmq.git  
$ cd jzmq  
$ ./autogen.sh  
$ ./configure  
$ make  
$ sudo make install
```

为了保证 JZMQ 正常工作，可能需要完成以下配置。

- ① 正确设置 JAVA\_HOME 环境变量
- ② 安装 Java 开发包

③ 升级 autoconf

④ 如果你用的是 Mac OS X 系统，参考本页脚注1的链接

**注意事项** 如果运行`./configure` 命令出现问题，参考本页脚注2的链接。

(3) 安装 Java 6。

① 下载并安装 JDK 6，参考本页脚注3的链接

② 配置 JAVA\_HOME 环境变量

③ 运行`java`、`javac`命令，测试 Java 正常安装

(4) 安装 Python 2.6.6。

① 下载 Python 2.6.6

---

1 <http://cbcgnet.blog/2011/07/30/getting-zeromq-and-jzmq-running-on-mac-os-x/>

2 <http://stackoverflow.com/questions/3522248/how-do-i-compile-jzmq-for-zeromq-on-osx>

3 <http://www.oracle.com/technetwork/java/javase/index-137561.html#linux>

```
$ wget http://www.python.org/ftp/python/2.6.6/Python-2.6.6.tar.bz2
```

### ② 编译安装 Python 2.6.6

```
$ tar -jxvf Python-2.6.6.tar.bz2  
$ cd Python-2.6.6  
$ ./configure  
$ make  
$ make install
```

### ③ 测试 Python 2.6.6

```
$ python -V  
Python 2.6.6
```

### (5) 安装 unzip。

- ① 如果使用 RedHat 系列的 Linux 系统，执行以下命令  
安装 unzip：

```
$ yum install unzip
```

- ② 如果使用 Debian 系列的 Linux 系统，执行以下命令  
安装 unzip：

```
$ apt-get install unzip
```

### 3. 下载并解压 Storm 发布版本

下一步，需要在 Nimbus 和 Supervisor 机器上安装 Storm 发行版本。

(1) 下载 Storm 发行版本，推荐使用 Storm 0.8.1。

```
$ wget https://github.com/downloads/nathanmarz/storm/storm-0.8.1.zip
```

(2) 解压到安装目录下。

```
$ unzip storm-0.8.1.zip
```

### 4. 修改 storm.yaml 配置文件

Storm 发行版本解压后，目录下有一个 conf/storm.yaml 文件，用于配置 Storm。默认配置可以在 <https://github.com/nathanmarz/storm/blob/master/conf/defaults.yaml> 中的配置选项覆盖 defaults.yaml 中的默认配置。以下配置选项是必须在 conf/storm.yaml 中进行配置的。

(1) **storm.zookeeper.servers**: Storm 集群使用的 Zookeeper 集群地址，其格式如下。

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

如果 Zookeeper 集群使用的不是默认端口，那么还需要 storm.zookeeper.port 选项。

(2) **storm.local.dir**: Nimbus 和 Supervisor 进程用于存储少量状态，如 jars、confs 等本地磁盘目录，需要提前创建该目录并给予足够的访问权限。然后在 storm.yaml 中配置该目录，如：

```
storm.local.dir: "/home/demo/storm/workdir"
```

(3) **java.library.path**: Storm 使用的本地库（ZMQ 和 JZMQ）加载路径，默认为"/usr/local/lib:/opt/local/lib:/usr/lib"，一般来说 ZMQ 和 JZMQ 默认安装在/usr/local/lib 下，因此不需要配置即可。

(4) **nimbus.host**: Storm 集群 Nimbus 机器地址，各个 Supervisor 工作节点需要知道哪个机器是 Nimbus，以便下载 Topologies 的 jars、confs 等文件，如：

```
nimbus.host: "111.222.333.444"
```

(5) **supervisor.slots.ports**: 对于每个 Supervisor 工作节点, 需要配置该工作节点可以运行的 Worker 数量。每个 Worker 占用一个单独的端口用于接收消息, 该配置选项用于定义哪些端口是可被 Worker 使用的。默认情况下, 每个节点上可运行 4 个 Workers, 分别在 6700、6701、6702 和 6703 端口, 如:

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

## 5. 启动 Storm 各个后台进程

最后一步, 启动 Storm 的所有后台进程。和 Zookeeper 一样, Storm 也是快速失败的系统, 这样 Storm 才能在任意时刻被停止, 并且当进程重启后被正确地恢复执行。这也是为什么 Storm 不在进程内保存状态的原因, 即使 Nimbus 或 Supervisors 被重启, 运行中的 Topologies 也不会受到影响。

以下是启动 Storm 各个后台进程的方式。

(1) **Nimbus**: 在 Storm 主控节点上运行 "bin/storm  
nimbus >/dev/null 2>&1 &"，启动 Nimbus 后台程序，并放到后台执行。

(2) **Supervisor**: 在 Storm 各个工作节点上运行 "bin/storm  
supervisor >/dev/null 2>&1 &"，启动 Supervisor 后台程序，并放到后台执行。

(3) **UI**: 在 Storm 主控节点上运行 "bin/storm ui >/dev/null  
2>&1 &"，启动 UI 后台程序，并放到后台执行，启动后可以通过 `http://{nimbus host}:8080` 观察集群的 Worker 资源使用情况、Topologies 的运行状态等信息。

注意事项如下。

(1) 启动 Storm 后台进程时，需要对 `conf/storm.yaml` 配置文件中设置的 `storm.local.dir` 目录具有写权限。

(2) Storm 后台进程被启动后，将在 Storm 安装部署目录下的 `logs/` 子目录下生成各个进程的日志文件。

(3) 经测试, Storm UI 必须和 Storm Nimbus 部署在同一台机器上, 否则 UI 无法正常工作, 因为 UI 进程会检查本机是否存在 Nimbus 链接。

(4) 为了方便使用, 可以将 bin/storm 加入系统环境变量中。

至此, Storm 集群已经部署、配置完毕, 可以向集群提交 Topology 运行。

### 2.2.3 向集群提交任务

#### 1. 启动 Storm Topology

```
$ storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3
```

其中, allmycode.jar 是包含 Topology 实现代码的 jar 包, org.me.MyTopology 的 main 方法是 Topology 的入口, arg1、arg2 和 arg3 为 org.me.MyTopology 执行时需要传入的参数。

#### 2. 停止 Storm Topology

```
$ storm kill [toponame]
```

其中，{toponame}为 Topology 提交到 Storm 集群时指定的 Topology 任务名称。

#### 2.2.4 本章参考资料

[1] <https://github.com/nathanmarz/storm/wiki/Tutorial>

[2] <https://github.com/nathanmarz/storm/wiki/Setting-up-a-Storm-cluster>

# 第3章

## 构建 Topology

### 3.1 Storm 基本概念

在运行一个 Storm 任务之前，需要了解如下概念。

- (1) Topologies
- (2) Streams
- (3) Spouts
- (4) Bolts
- (5) Stream Groupings
- (6) Reliability
- (7) Tasks
- (8) Workers
- (9) Configuration

Storm 集群和 Hadoop 集群表面上看很类似，但是 Hadoop

上运行的是 MapReduce job，而 Storm 上运行的是拓扑 (Topology)，这两者之间是非常不一样的。一个关键的区别是：MapReduce job 最终会结束，而 Topology 会永远运行（除非你手动结束）。

在 Storm 的集群里有两种节点：控制节点和工作节点。控制节点上面运行一个叫 Nimbus 的后台程序，它的作用类似 Hadoop 里面的 JobTracker。Nimbus 负责在集群里分发代码，分配计算任务给机器并监控状态。

每一个工作节点上面运行一个叫作 Supervisor 的节点。Supervisor 会监听分配给它那台机器的工作，根据需要启动/关闭工作进程。每一个工作进程执行一个 Topology 的一个子集；一个运行的 Topology 由运行在很多机器上的很多工作进程组成。

Nimbus 和 Supervisor 之间的所有协调工作都是通过 Zookeeper 集群完成的。另外，Nimbus 进程和 Supervisor 进程都是快速失败和无状态的。所有的状态要么在 Zookeeper 里面，要么在本地磁盘上。这也就意味着你可以用 kill -9 来

结束 Nimbus 和 Supervisor 进程，然后再重启它们，就好像什么都没有发生过。这个设计使得 Storm 异常稳定。

### 3.1.1 Topologies

通过 Stream Groupings 将 Spouts 和 Bolts 连接起来构成一个 Topology，如图 3-1 所示。

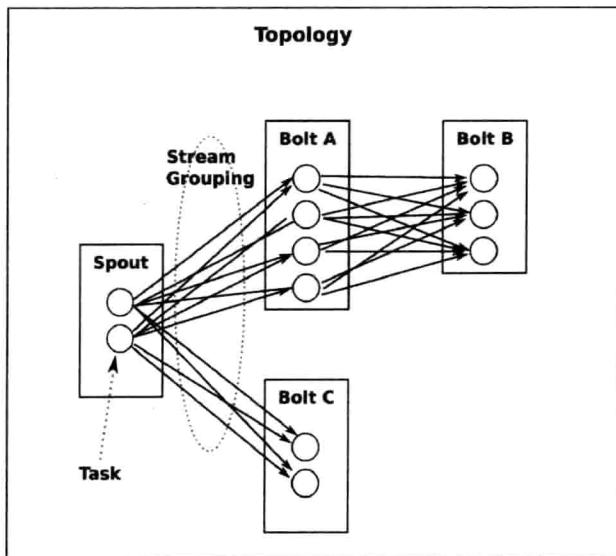


图 3-1

一个 Topology 会一直运行，直到你手动结束，Storm 自

动重新分配执行失败的任务，并且 Storm 可以保证你不会有数据丢失（如果开启了高可靠性的话）。如果一些机器意外停机它上面的所有任务会被转移到其他机器上。

运行一个 Topology 很简单。首先，把你所有的代码及所依赖的 jar 打进一个 jar 包。然后运行类似下面的这个命令。

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

这个命令会运行主类 `backtype.strom.MyTopology`，参数是 `arg1` 和 `arg2`。这个类的 `main` 函数定义这个 Topology 并且把它提交给 Nimbus。Storm jar 负责连接到 Nimbus 并且上传 jar 包。

Topology 的定义是一个 Thrift 结构，并且 Nimbus 就是一个 Thrift 服务，你可以提交由任何语言创建的 Topology。上面讲述的方法是用 JVM-based 语言提交的最简单的方法。

### 3.1.2 Streams

消息流 Stream 是 Storm 里的关键抽象。一个消息流是一

个没有边界的 tuple 序列，而这些 tuple 序列会以一种分布式的方式并行地创建和处理。通过对 Stream 中 tuple 的每个字段命名来定义 Stream。在默认情况下，tuple 的字段类型可以是 integer、long、short、byte、string、double、float、boolean 和 byte array。你也可以自定义类型（只要实现相应的序列化器）。

每个消息流在定义时会被分配一个 id，因为单一消息流使用得相当普遍，OutputFieldsDeclarer 定义了一些方法让你可以定义一个 Stream 而不用指定这个 id。在这种情况下，这个 Stream 会被分配一个值为 ‘default’ 的默认 id。

Storm 提供的最基本的处理 Stream 的原语是 Spout 和 Bolt。你可以实现 Spout 和 Bolt 提供的接口来处理你的业务逻辑。

### 3.1.3 Spouts

消息源 Spout 是 Storm 里面一个 Topology 中的消息生产者。一般来说消息源会从一个外部源读取数据并且向

Topology 里面发出消息 tuple。Spout 可以是可靠的也可以是不可靠的。如果这个 tuple 没有被 Storm 成功处理，可靠的消息源 Spout 可以重新发射一个 tuple，但是不可靠的消息源 Spout 一旦发出一个 tuple 就不能重发了。

消息源可以发射多条消息流 Stream。使用 OutputFieldsDeclarer.declareStream 来定义多个 Stream，然后使用 SpoutOutputCollector 来发射指定的 Stream。

Spout 类里面最重要的方法是 nextTuple。要么发射一个新的 tuple 到 Topology 里面，要么简单地返回（如果没有新的 tuple）。要注意的是 nextTuple 方法不能阻塞，因为 Storm 在同一个线程上面调用所有消息源 Spout 的方法。

另外两个比较重要的 Spout 方法是 ack 和 fail。Storm 在检测到一个 tuple 被整个 Topology 成功处理的时候调用 ack，否则调用 fail。Storm 只对可靠的 Spout 调用 ack 和 fail。

### 3.1.4 Bolts

所有的消息处理逻辑都被封装在 Bolts 里面。Bolts 可以

做很多事情：过滤、聚合、查询数据库，等等。

可以简单地理解 Bolts 为消息流的传递。复杂的消息流处理往往需要很多步骤，从而也就需要经过很多 Bolts。比如算出一堆图片里面被转发最多的图片就至少需要两步：第一步是算出每个图片的转发数量；第二步是找出转发最多的前 10 个图片（如果要把这个过程做得更具有扩展性，可能需要更多的步骤）。

Bolts 可以发射多条消息流，使用 `OutputFieldsDeclarer.declareStream` 定义 Stream，使用 `OutputCollector.emit` 选择要发射的 Stream。

Bolts 的主要方法是 `execute`，它以一个 tuple 作为输入，Bolts 使用 `OutputCollector` 来发射 tuple，Bolts 必须要为它处理的每一个 tuple 调用 `OutputCollector` 的 `ack` 方法，以通知 Storm 这个 tuple 被处理完成了，从而通知这个 tuple 的发射者 Spouts。一般的流程是：Bolts 处理一个输入 tuple，发射 0 个或者多个 tuple，然后调用 `ack` 通知 Storm 自己已经处理过这个 tuple 了。Storm 提供了一个 `IBasicBolt` 会自动调用 `ack`。

### 3.1.5 Stream Groupings

定义一个 Topology 的其中一步是定义每个 Bolt 接收什么样的流作为输入。Stream Grouping 就是用来定义一个 Stream 应该如何分配数据给 Bolts 上面的多个 tasks 的。

Storm 里面有 7 种类型的 Stream Grouping。

- **Shuffle Grouping:** 随机分组。随机派发 Stream 里面的 tuple，保证每个 Bolt 接收到的 tuple 数目大致相同。
- **Fields Grouping:** 按字段分组。比如按 userid 分组，相同 userid 的 tuple 会被分到同一个 Bolts 的同一个 task 中，而不同的 userid 则会被分配到不同的 task 中。
- **All Grouping:** 广播发送。对于每一个 tuple，所有的 Bolts 都会收到。
- **Global Grouping:** 全局分组。这个 tuple 被分配到 Storm 中的一个 Bolt 的其中一个 task。再具体一点就是分配给 id 值最低的那个 task。
- **Non Grouping:** 不分组。这个分组的意思是说 Stream

不关心到底谁会收到它的 tuple。目前这种分组和 Shuffle Grouping 是一样的效果，有一点不同的是 Storm 会把这个 Bolt 放到这个 Bolt 的订阅者的同一个线程里面去执行。

- Direct Grouping：直接分组。这是一种比较特别的分组方法，用这种分组意味着消息的发送者需要指定由 消息接收者的哪个 task 处理这个消息。只有被声明为 Direct Stream 的消息流可以声明这种分组方法。而且这种消息 tuple 必须使用 emitDirect 方法发射。消息处理器可以通过 TopologyContext 获取处理它的消息的 task 的 id (OutputCollector.emit 方法也会返回 task 的 id)。
- Local or Shuffle Grouping：如果目标 Bolt 有一个或者多个 task 在同一个工作进程中，tuple 将会被随机发生给这些 tasks。否则，和普通的 Shuffle Grouping 行为一致。

### 3.1.6 Reliability

Storm 保证每个 tuple 会被 Topology 完整执行。Storm 会追踪由每个 Spout tuple 所产生的 tuple 树（一个 Bolt 处理一个 tuple 之后可能会发射别的 tuple 从而形成树状结构），并且跟踪这棵 tuple 树什么时候成功处理完。每个 Topology 都有一个消息超时的设置，如果 Storm 在这个超时的时间内检测不到某个 tuple 树到底有没有执行成功，那么 Topology 会把这个 tuple 标记为执行失败，并且过一会儿重新发射这个 tuple。

为了利用 Storm 的可靠性特性，在即将发出一个新的 tuple 之前，以及完成处理一个 tuple 之后，你必须要通知到 Storm。这一切是由 OutputCollector 来完成的，它主要是 Bolt 用来发送 tuple 的。通过 emit 方法通知一个新的 tuple 产生，通过 ack 方法通知一个 tuple 处理完成了。

Storm 的可靠性我们会在本书后面章节深入介绍。

### 3.1.7 Tasks

每一个 Spout 和 Bolt 会被当作很多 task 在整个集群里执行。每一个 executor 对应到一个线程，在这个线程上运行多个 task，而 Stream Grouping 则是定义怎么从一堆 task 发射 tuple 到另外一堆 task。你可以调用 TopologyBuilder 类的 setSpout 和 setBolt 来设置并行度（也就是有多少个 task）。

### 3.1.8 Workers

一个 Topology 可能会在一个或者多个 Worker（工作进程）里面执行，每个 Worker 是一个物理 JVM 并且执行整个 Topology 的一部分。比如，对于并行度是 300 的 Topology 来说，如果我们使用 50 个工作进程来执行，那么每个工作进程会处理其中的 6 个 tasks。Storm 会尽量均匀地将工作分配给所有的 Worker。

### 3.1.9 Configuration

Storm 里面有一堆参数可以配置来调整 Nimbus、

Supervisor, 以及正在运行的 Topology 的行为, 一些配置是系统级别的, 一些配置是 Topology 级别的。default.yaml 里面有所有的默认配置。你可以通过定义一个 storm.yaml 在你的 classpath 里覆盖这些默认配置。你也可以在代码里设置一些与 Topology 相关的配置信息 (使用 StormSubmitter)。

## 3.2 构建 Topology

### 3.2.1 实现目标

我们将设计一个 Topology, 用以实现对一个句子里面的单词出现的频率进行统计。这是一个简单的例子, 目的是让大家对 Topology 快速上手, 有一个初步的理解。

### 3.2.2 设计 Topology 结构

开发 Storm 项目的第一步, 就是要设计 Topology。我们来看一个简单的例子, 整个 Topology 如图 3-2 所示。



图 3-2

整个 Topology 分为三个部分。

- KestrelSpout：数据源，负责发送 sentence。
- SplitSentence：负责将 sentence 切分。
- WordCount：负责对单词的频率进行累加。

### 3.2.3 设计数据流

这个 Topology 从 kestrel 队列读取句子，并把句子划分成单词，然后汇总每个单词出现的次数，一个 tuple 负责读取句子，每一个 tuple 分别对应计算每一个单词出现的次数。

### 3.2.4 代码实现

#### 1. 构建 Maven 环境

为了开发 Storm Topology，你需要把 Storm 相关的 jar 包添加到 classpath 里去——要么手动添加所有相关的 jar 包，

要么使用 Maven 来管理所有的依赖。Storm 的 jar 包发布在 Clojars (一个 Maven 库)，如果你使用 Maven 的话，把下面的配置添加在你项目的 pom.xml 里面。

```
<repository>
  <id>clojars.org</id>
  <url>http://clojars.org/repo</url>
</repository>
<dependency>
  <groupId>storm</groupId>
  <artifactId>storm</artifactId>
  <version>0.5.3</version>
  <scope>test</scope>
</dependency>
```

## 2. 定义 Topology

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(1, new KestrelSpout(
    "kestrel.backtype.com",
    22133,
    "sentence_queue",
    new StringScheme()));
builder.setBolt(2, new SplitSentence(), 10).
shuffleGrouping(1);
builder.setBolt(3, new WordCount(), 20).
fieldsGrouping(2, new Fields("word"));
```

这个 Topology 的 Spout 从句子队列中读取句子，句子队列在 `kestrel.backtype.com` 服务器上，对应的端口为 22133。

Spout 用 `setSpout` 方法插入一个独特的 id 到 Topology。Topology 中的每个节点必须给予一个 id，id 是其他 Bolts 用于

订阅该节点的输出流。KestrelSpout 在 Topology 中的 id 为 1。

setBolt 适用于在 Topology 中插入 Bolts。在 Topology 中定义的第一个 Bolts 是切割句子的 Bolts。这个 Bolts 将句子流转成单词流。

让我们看看 SplitSentence 的实现：

```
public class SplitSentence implements IBasicBolt {  
    public void prepare(Map conf, TopologyContext context) {  
    }  
    public void execute(Tuple tuple, BasicOutputCollector collector) {  
        String sentence = tuple.getString(0);  
        for(String word: sentence.split(" ")) {  
            collector.emit(new Values(word));  
        }  
    }  
    public void cleanup() {  
    }  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("word"));  
    }  
}
```

关键的方法是 execute 方法。正如你看到的，它将句子拆分成单词，并发出每个单词作为一个新的消息。另一个重要的方法是 declareOutputFields，其中定义 Bolts 输出元组的结构，它发出一个字段为 word 的元组。



```
        count++;
        _counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    public void cleanup() {
    }

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

SplitSentence 对句子里面的每个单词发送一个新的 tuple，WordCount 在内存里面维护一个单词到→次数的映射表，WordCount 每收到一个单词，就更新内存里面的统计状态。

### 3.2.5 运行 Topology

Storm 的运行有两种模式：本地模式和分布式模式。

#### 1. 本地模式

Storm 用一个进程里面的线程来模拟所有的 Spout 和 Bolt。本地模式对开发和测试来说比较有用。当你运行 storm-starter 里面的 Topology 时，它们就是以本地模式运行的，你可以看到 Topology 里面的每一个组件在发射什么消息。

## 2. 分布式模式

Storm 由多台机器组成。当你提交 Topology 给 Master 时，你同时也把 Topology 的代码提交了。Master 负责分发你的代码，同时负责给你的 Topology 分配工作进程。如果一个工作进程挂掉了，Master 会把任务重新分配给其他节点。

下面是以本地模式运行的代码：

```
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，这个代码定义通过定义一个 LocalCluster 对象来定义一个进程内的集群。提交 Topology 给这个虚拟的集群和提交 Topology 给分布式集群是一样的。通过调用 submitTopology 方法来提交 Topology，它接受三个参数：要运行的 Topology 的名字，一个配置对象，以及要运行的 Topology 本身。

Topology 的名字是用来唯一标识一个 Topology 的，这样你可以用这个名字来结束这个 Topology。前面已经说过了，你必须显式的结束一个 Topology，否则它会一直运行。

conf 对象可以配置很多东西，下面两个是最常见的。

TOPOLOGY\_WORKERS(setNumWorkers) 定义你希望集群分配多少个工作进程给你来执行这个 Topology，Topology 里面的每个组件都需要线程来执行。每个组件到底用多少个线程是通过 setBolt 和 setSpout 来指定的。这些线程都运行在工作进程里面，每个工作进程包含一些节点的一些工作线程。比如，如果你指定 300 个线程，60 个进程，那么每个工作进程里面要执行 6 个线程，而这 6 个线程可能属于不同的组件（Spout 或者 Bolt）。你可以通过调整每个组件的并行度及这些线程所在的进程数量来调整 Topology 的性能。

当 TOPOLOGY\_DEBUG(setDebug) 被设置成 true 时，Storm 会记录下每个组件发射的每条消息。这在本地环境调试 Topology 时很有用，但是在线上这么做会影响性能。

### 3.3 小结

本章从 Storm 的基本对象的定义讲起，讲到广泛地介绍 Storm 的开发环境，用一个简单的例子讲解了 Topology 的构建和定义。希望大家可以从本章的内容中对 Storm 有一个基本的理解，并且可以构建一个简单的 Topology！

# 第4章

## Topology 的 并行度

## 4.1 并行元素

在 Storm 集群中真正运行 Topology 的主要有三个实体：工作进程、Executor（线程）和任务。图 4-1 是一个简单的例子，表示它们之间的关系。

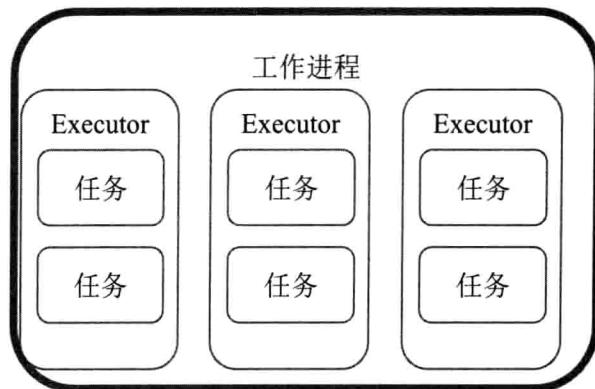


图 4-1

### 4.1.1 工作进程

Storm 集群中的一台机器会为一个或多个 Topology 运行

一个或多个工作进程，每个工作进程执行 Topology 的一个子集。一个工作进程属于一个特定的 Topology，工作进程会为该 Topology 启动一个或者多个 Executor。

#### 4.1.2 Executor

Executor 是一个由工作进程启动的线程，在一个工作进程中，可能会有一个或者多个 Executor。一个 Executor 会为某一个 component (Spout 或者 Bolt) 运行一个或多个任务，每一个线程只会为同一个 component 服务。

#### 4.1.3 任务

任务是真正进行数据处理的实体，你代码中实现的每一个 Spout 或 Bolt 都在集群间运行多个任务。

在 Topology 的生命周期中，每个 component (Spout 或 Bolt) 运行的任务数都是不变的，但是每个组件 (Spout 或 Bolt) 的 Executor 数是可以修改的。

这意味着下述条件是成立的：线程数  $\leq$  任务数。默认情况下，任务数被设置为和线程数相同，即 Storm 的每个线程执行一个任务。

## 4.2 配置并行度

在 Storm 里面，并行度描述了工作进程数据，以及 Executor 的数量和 Storm 中的任务数量。

本节将简单介绍几个配置项，以及如何在代码中修改它们。有很多方法可以设置这些选项，本节介绍的只是其中的一部分。目前 Storm 的配置优先级如下。

`defaults.yaml < storm.yaml < Topology` 的具体配置 < 内部组件的具体配置 < 外部组件的具体配置。

### 4.2.1 配置工作进程数

工作进程数表示可以为一个 Topology 指定在集群中启

动几个工作进程。

`backtype.storm.Config` 类中有一个选项 `TOPOLOGY_WORKERS`，就是用来指定一个 Topology 启动的工作进程数。

每个工作进程都会通过它内部的线程来执行一定数量的任务，因此这个参数应该结合 Topology 中每个组件的并行度来使用，以便调整 Topology 的性能。

配置 `TOPOLOGY_WORKERS` 选项，可以调用 `Config` 类中的 `setNumWorkers` 接口完成，设置的代码参考本节关于 Topology 运行的例子。

#### 4.2.2 配置 Executor 数

Executor 数描述每个 component (Spout 或 Bolt) 启动几个 Executor，每个 component 的 Executor 都需要各自单独配置。

在 TopologyBuilder 类中有 setSpout 和 setBolt 两个接口，分别用来设置 Spout 和 Bolt 初始化时 Executor 的数量，其中 parallelism\_hint 参数表示 Executor 的数量：

```
setSpout(String id, IRichSpout spout, Number parallelism_hint);  
setBolt(String id, IRichBolt bolt, Number parallelism_hint);
```

假如使用不带 parallelism\_hint 参数的 setSpout 或 setBolt 接口，则默认只启动一个 Executor。

### 4.2.3 配置任务数

任务数描述了为每个 component 创建多少个任务。每个 component 启动多少个任务都需要单独配置。

backtype.storm.Config 类中有一个选项 TOPOLOGY\_TASKS，用于配置每个 component (Bolt 或 Spout) 启动多少个任务。一个 Executor 会为相同的 component (Bolt 或 Spout) 运行一个或多个任务。在 Topology 的生命周期里，component 的任务数总是相同的，但是 component 的 Executor 的数量是可以修改的。这允许一个 Topology 可以调整使用更多或者更少的资源，而不用重新部署 Topology 或者违反 Storm 的约束。

(如按字段分组时，任务数不变，可以保证相同的值到相同的任务)。

在 ComponentConfigurationDeclarer 接口里有一个函数 setNumTasks 可以用来配置 component 的任务数。

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

### 4.3 一个运行中 Topology 的例子

下面展示了一个简单的 Topology。Topology 里包括了三个 component：一个是 Blue Spout 的 Spout，另外两个分别是 Green Bolt 和 Yellow Bolt 的 Bolt。Blue Spout 发送它的输出给 Green Bolt，Green Bolt 将自己的输出发送给 Yellow Bolt，如图 4-2 所示。

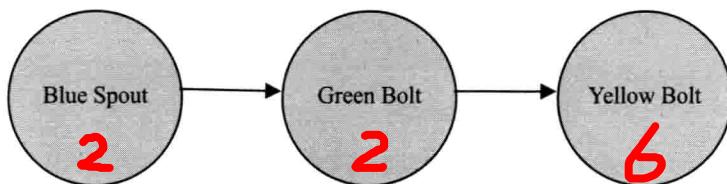


图 4-2

其中，Blue Spout 的 parallelism\_hint (Executor 数) 设置为 2，Green Bolt 的 parallelism\_hint (Executor 数) 设置为 2，Yellow Bolt 的 parallelism\_hint (Executor 数) 设置为 6，parallelism\_hint 表示每个 component 初始化时启动的线程数。因此总的线程数为 10，Topology 中有 2 个工作进程，因此每个工作进程将会处理启动 5 个线程。其中 Green Bolt 设置为启动 4 个任务，因此 Green Bolt 的每个线程启动 2 个任务。

整个 Topology 的并行度如图 4-3 所示。

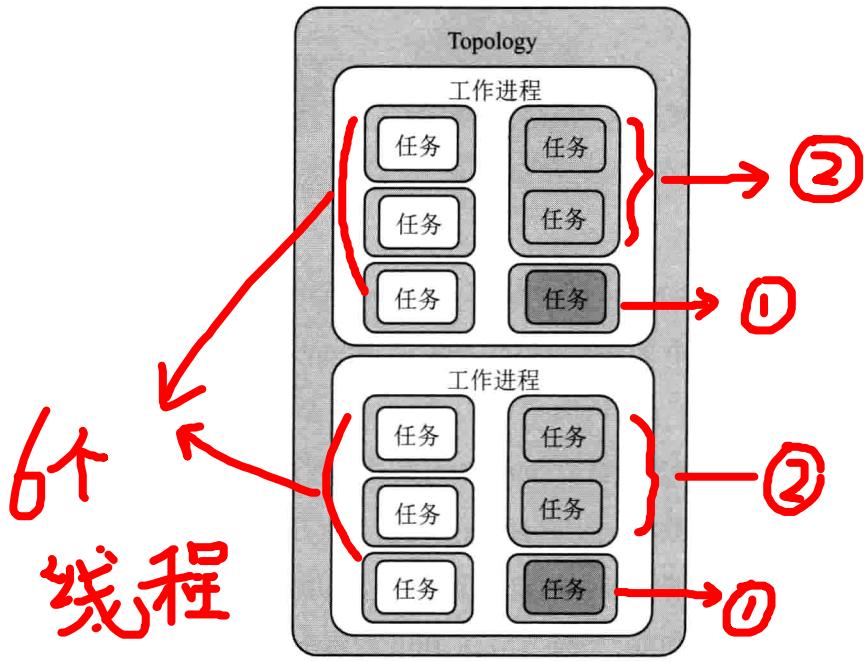


图 4-3

设置的代码如下，Blue Spout 和 Yellow Bolt 只是设置了 parallelism\_hint（初始化时的线程数），而 Green Bolt 设置了任务数为 4。

设置的代码如下：

```
Config conf = new Config();
conf.setNumWorkers(2); // 将工作进程数设置为2

// 设置Blue Spout的线程数为2
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2);

// 设置Green Bolt的线程数为2，并将任务数设置为4
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
.setNumTasks(4)
.shuffleGrouping("blue-spout");

// 设置Yellow Bolt的线程数为6
topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
.shuffleGrouping("green-bolt");

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

## 4.4 如何更新运行中的 Topology 的并行度

Storm 的一个非常好的功能是：你可以动态地增加或减少工作进程数或 Executor 数量，而不用重启集群或者 Topology，这叫作 rebalancing。

你可以通过两种方式去 rebalance 一个 Topology。

- (1) 使用 Storm 的 Web 界面去 rebalance Topology。
- (2) 使用命令行工具 storm rebalance，下面是一个使用命令行工具的例子。

将名为 mytopologyTopology 的工作进程数设置为 5，将名为 blue-spout 的 Spout 的 Executor 数设置为 3，将名为 yellow-bolt 的 Bolt 的 Executor 数设置为 10。

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

# 第5章

## 消息的 可靠处理

## 5.1 简介

Storm 可以确保 Spout 发送出来的每个消息都会被完整处理。本章将描述 Storm 体系是如何达到这个目标的，并将详述开发者应该如何使用 Storm 的这些机制来实现数据的可靠处理。

## 5.2 理解消息被完整处理

一个消息（tuple）从 Spout 发送出来，可能导致成百上千的消息基于此消息而创建。

我们来思考一下流式的“单词统计”的例子。

Storm 任务从数据源（kestrel 队列）每次读取一个完整的英文句子；将这个句子分解为独立的单词，最后，实时地输出每个单词及它出现过的次数。

本例中，每个从 Spout 发送出来的消息（每个英文句子）都会触发很多消息被创建，那些从句子中分隔出来的单词就是被创建出来的新消息。

这些消息构成一个树状结构，我们称之为“tuple tree”，如图 5-1 所示。

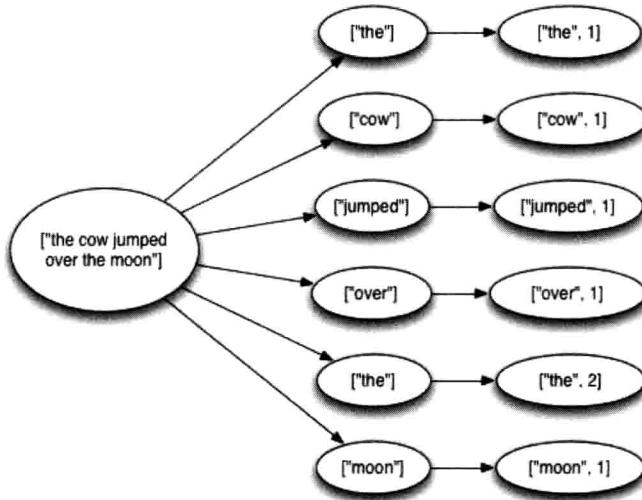


图 5-1

在什么条件下，Storm 才会认为一个从 Spout 发送出来的消息被完整处理了呢？答案就是以下条件被同时满足。



(1) tuple tree 不再生长。

(2) tuple tree 中的任何消息被标识为“已处理”。

如果在指定的时间内，一个消息衍生出来的 tuple tree 未被完全处理成功，则认为此消息未被完整处理。这个超时值可以通过任务级参数 Config.TOPOLOGY\_MESSAGE\_TIMEOUT\_SECS 进行配置，默认为 30 秒。

### 5.3 消息的生命周期

如果消息被完整处理或者未被完整处理，Storm 会如何进行接下来的操作呢？为了弄清这个问题，我们来研究一下从 Spout 发出来的消息的生命周期。Spout 应该实现的接口如下：

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

首先，Storm 使用 Spout 实例的 nextTuple()方法从 Spout 请求一个 tuple。收到请求以后，Spout 使用 open 方法中提供的 SpoutOutputCollector 向它的输出流发送一个或多个消息。每发送一个消息，Spout 会给这个消息提供一个 id，它将被用来标识这个消息。

假设我们从 kestrel 队列中读取消息，Spout 会将 kestrel 队列为这个消息设置的 id 作为此消息的 id。向 SpoutOutputCollector 中发送消息的格式如下：

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

接下来，这些消息会被发送到后续业务处理的 Bolts，并且 Storm 会跟踪由此消息产生出来的新消息。当检测到一个消息衍生出来的 tuple tree 被完整处理后，Storm 会调用 Spout 中的 ack 方法，并将此消息的 id 作为参数传入。同理，如果某消息处理超时，则此消息对应的 Spout 的 fail 方法会被调用，调用时此消息的 id 会被作为参数传入。

**注意：**一个消息只会由发送它的那个 Spout 任务调用 ack 或 fail。如果系统中某个 Spout 由多个任务运行，消息也只会

由创建它的 Spout 任务来应答（ack 或 fail），决不会由其他的 Spout 任务来应答。

我们继续使用从 `kestrel` 队列中读取消息的例子来阐述高可靠性下 Spout 需要做些什么（假设这个 Spout 的名字是 `KestrelSpout`）。

我们先简述一下 `kestrel` 消息队列。

当 `KestrelSpout` 从 `kestrel` 队列中读取一个消息，表示它“打开”了队列中某个消息。这意味着，此消息并未从队列中真正删除，而是被设置为“pending”状态，它等待来自客户端的应答，被应答以后，此消息才会被真正从队列中删除。处于“pending”状态的消息不会被其他客户端看到。另外，如果一个客户端意外断开连接，则由此客户端“打开”的所有消息都会被重新加入到队列中。当消息被“打开”的时候，`kestrel` 队列同时会为这个消息提供一个唯一的标识。

`KestrelSpout` 使用这个唯一的标识作为这个 tuple 的 id。当 ack 或 fail 被调用时，`KestrelSpout` 会把 ack 或者 fail 连同 id 一起发送给 `kestrel` 队列，`kestrel` 会将消息从队列中真正删

除或者将它重新放回队列中。

## 5.4 可靠相关的 API

为了使用 Storm 提供的可靠处理特性，我们需要做两件事情。

(1) 无论何时，只要在 tuple tree 中创建了一个新的节点，我们就要明确地通知 Storm。

(2) 当处理完一个单独的消息时，我们需要告诉 Storm 这棵 tuple tree 的变化状态。

通过上面的两步，Storm 就可以检测到一个 tuple tree 何时被完全处理了，并且会调用相关的 ack 或 fail 方法。Storm 提供了简单明了的方法来完成上述两步。

为 tuple tree 中指定的节点增加一个新的节点，我们称之为锚定 (anchoring)。锚定是在我们发送消息的同时进行的。为了更好地说明问题，我们使用下面的代码作为例子。本示

例的 Bolt 将包含整句话的消息分解为一系列子消息，每个子消息包含一个单词。

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

每个消息都通过这种方式被锚定：把输入消息作为 emit 方法的第一个参数。因为 word 消息被锚定在了输入消息上，这个输入消息是 Spout 发送过来的 tuple tree 的根节点，如果任意一个 word 消息处理失败，派生这个 tuple tree 的那个 Spout 消息将会被重新发送。

与此相反，我们来看看使用下面代码所示的方式 emit 消息时，Storm 会如何处理。

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

如果以这种方式发送消息，将会导致这个消息不会被锚定。如果此 tuple tree 中的消息处理失败，派生此 tuple tree 的根消息不会被重新发送。根据任务的容错级别，有时候很适合发送一个非锚定的消息。

一个输出消息可以被锚定在一个或者多个输入消息上，这在做 join 或聚合时是很有用的。一个被多重锚定的消息处理失败，会导致与之关联的多个 Spout 消息被重新发送。多重锚定通过在 emit 方法中指定多个输入消息来实现，如：

```
List<Tuple> anchors = new ArrayList<Tuple>();
anchors.add(tuple1);
anchors.add(tuple2);
_collector.emit(anchors, new Values(1, 2, 3));
```

多重锚定会将被锚定的消息加到多棵 tuple tree 上。

**注意：**多重绑定可能会破坏传统的树形结构，从而构成一个 DAGs（有向无环图），如图 5-2 所示。

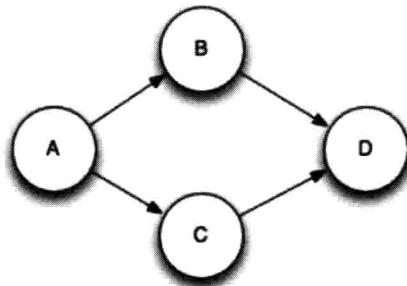


图 5-2

Storm 的实现可以像处理树那样来处理 DAGs。

锚定表明了如何将一个消息加入指定的 tuple tree 中，高可靠处理 API 的接下来部分将向我们描述当处理完 tuple tree 中一个单独的消息时我们该做些什么。这些是通过 OutputCollector 的 ack 和 fail 方法来实现的。回头看一下例子 SplitSentence 可以发现当所有的 word 消息被发送完成后，输入的表示句子的消息会被应答（acked）。

每个被处理的消息必须表明成功或失败（acked 或者 failed）。Storm 是使用内存来跟踪每个消息的处理情况的，如果被处理的消息没有应答的话，迟早内存会被耗尽！

很多 Bolt 遵循特定的处理流程：读取一个消息、发送它

派生出来的子消息、在 `execute` 结尾处应答此消息。一般的过滤器(filter)或者是简单的处理功能都是这类的应用。Storm有一个 `BasicBolt` 接口封装了上述流程。`SplitSentence` 可以使用 `BasicBolt` 来重写，代码如下：

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

使用这种方式，代码比之前稍微简单一些，但是实现的功能是一样的。发送到 `BasicOutputCollector` 的消息会被自动锚定到输入消息中，并且，当 `execute` 执行完毕时，会自动应答输入消息。

很多情况下，一个消息需要延迟应答，例如聚合或者是 `join`。只有根据一组输入消息得到一个结果之后，才会应答之前所有的输入消息。并且聚合和 `join` 大部分时候对输出消息都是多重锚定。然而，这些特性不是 `IBasicBolt` 所能处

理的。

## 5.5 高效地实现 tuple tree

Storm 系统中有一组叫作“acker”的特殊任务，它负责跟踪 DAG 中的每个消息。每当发现一个 DAG 被完全处理，它就向创建这个根消息的 Spout 任务发送一个信号。拓扑中 acker 任务的并行度可以通过配置参数 Config.TOPOLOGY\_ACKERS 来设置。默认的 acker 任务并行度为 1，当系统中有大量的消息时，应该适当提高 acker 任务的并发度。

为了理解 Storm 可靠性处理机制，我们从研究一个消息的生命周期和 tuple tree 的管理入手。当一个消息被创建时（无论是在 Spout 还是 Bolt 中），系统都为该消息分配一个 64bit 的随机值作为 id。这些随机的 id 是 acker 用来跟踪由 Spout 消息派生出来的 tuple tree 的。

每个消息都知道它所在的 tuple tree 对应的根消息的 id。每当 Bolt 新生成一个消息，对应的 tuple tree 中的根消息的



id 就复制到这个消息中。当这个消息被应答的时候，它就把关于 tuple tree 变化的信息发送给跟踪这棵树的 acker。例如，他会告诉 acker：“本消息已经处理完毕，但是我派生出了一些新的消息，帮忙跟踪一下吧”。

举个例子，假设消息 D 和 E 是由消息 C 派生出来的，图 5-3 演示了消息 C 被应答时，tuple tree 是如何变化的。

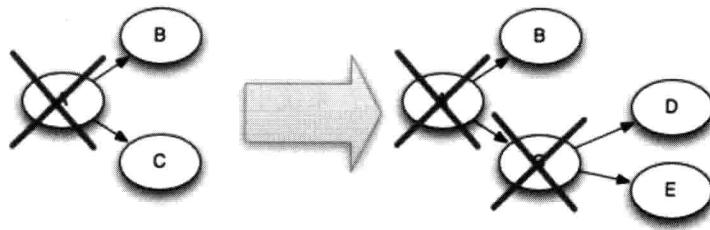


图 5-3

因为在 C 被从树中移除的同时 D 和 E 会被加入到 tuple tree 中，因此 tuple tree 不会被过早地认为已完全处理。

关于 Storm 如何跟踪 tuple tree，我们再深入探讨一下。前面说过系统中可以有任意个数的 acker，那么，每当一个消息被创建或应答时，它怎么知道应该通知哪个 acker 呢？

系统使用一种哈希算法根据 Spout 消息的 id 确定由哪个 acker 跟踪此消息派生出来的 tuple tree。因为每个消息都知道与之对应的根消息的 id，因此它知道应该与哪个 acker 通信。

当 Spout 发送一个消息的时候，它就通知对应的 acker 一个新的根消息产生了，这时 acker 就会创建一个新的 tuple tree。当 acker 发现这棵树被完全处理之后，它就会通知对应的 Spout 任务。

Tuple 是如何被跟踪的呢？系统中有成千上万的消息，如果为每个 Spout 发送的消息都构建一棵树的话，很快内存就会耗尽。所以，必须采用不同的策略跟踪每个消息。由于使用了新的跟踪算法，Storm 只需要固定的内存（大约 20 字节）就可以跟踪一棵树。这个算法是 Storm 正确运行的核心，也是 Storm 最大的突破。

Acker 任务保存了 Spout 消息 id 到一对值的映射。第一个值就是 Spout 的任务 id，通过这个 id，acker 就知道消息处理完成时该通知哪个 Spout 任务。第二个值是一个 64bit 的数字，我们称之为“ack val”，它是树中所有消息的随机 id 的

异或结果。`ack val` 表示了整棵树的状态，无论这棵树多大，只需要这个固定大小的数字就可以跟踪整棵树。当消息被创建和被应答时都会有相同的消息 `id` 发送过来做异或。

每当 `acker` 发现一棵树的 `ack val` 值为 0 时，它就知道这棵树已经被完全处理了。因为消息的随机 `id` 是一个 64bit 的值，因此 `ack val` 在树处理完之前被置为 0 的概率非常小。假设你每秒钟发送一万个消息，从概率上说，至少需要 50,000,000 年才有机会发生一次错误。即使如此，也只有在这个消息确实处理失败的情况下才会有数据丢失！

## 5.6 选择合适的可靠性级别

`Acker` 任务是轻量级的，所以在拓扑中并不需要太多的 `acker` 存在。可以通过 Storm UI 观察 `acker` 任务的吞吐量，如果看上去吞吐量不够的话，说明需要添加额外的 `acker`。

如果你并不要求每个消息必须被处理（你允许在处理过程中丢失一些信息），那么可以关闭消息的可靠处理机制，从

而获取较好的性能。关闭消息的可靠处理机制意味着系统中的消息数会减半（每个消息不需要应答了）。另外，关闭消息的可靠处理机制可以减少消息的大小（不需要每个 tuple 记录它的根 id），从而节省带宽。

有三种方法可以调整消息的可靠处理机制。

- 第一种方法是将参数 Config.TOPOLOGY\_ACKERS 设置为 0，通过此方法，当 Spout 发送一个消息时，它的 ack 方法将立刻被调用。
- 第二种方法是 Spout 发送一个消息时，不指定此消息的 id。当需要关闭特定消息可靠性时，可以使用此方法。
- 第三种方法是，如果你不在意某个消息派生出来的子孙消息的可靠性，则此消息派生出来的子消息在发送时不要做锚定，即在 emit 方法中不指定输入消息。因为这些子孙消息没有被锚定在任何 tuple tree 中，因此他们的失败不会引起任何 Spout 重新发送消息。

## 5.7 集群的各级容错

到现在为止，我们已经理解了 Storm 的可靠性机制，并且知道了如何选择不同的可靠性级别来满足需求。接下来我们研究一下 Storm 如何确保在各种情况下数据都不丢失。

### 5.7.1 任务级失败

- Bolt 任务 crash 引起的消息未被应答。此时，acker 中所有与此 Bolt 任务关联的消息都会因为超时而失败，对应的 Spout 的 fail 方法将被调用。
- acker 任务失败。如果 acker 任务本身失败了，它在失败之前持有的所有消息都将因超时而失败。Spout 的 fail 方法将被调用。
- Spout 任务失败。在这种情况下，与 Spout 任务对接的外部设备（如 MQ）负责消息的完整性。例如，当客

户端异常时，kestrel 队列会将处于 pending 状态的所有消息重新放回队列中。

### 5.7.2 任务槽（slot）故障

- Worker 失败。每个 Worker 中包含数个 Bolt（或 Spout）任务。Supervisor 负责监控这些任务，当 Worker 失败后，Supervisor 会尝试在本机重启它。
- Supervisor 失败。Supervisor 是无状态的，因此 Supervisor 的失败不会影响当前正在运行的任务，只要及时将它重新启动即可。Supervisor 不是自举的，需要外部监控来及时重启。
- Nimbus 失败。Nimbus 是无状态的，因此 Nimbus 的失败不会影响当前正在运行的任务（Nimbus 失败时，无法提交新的任务），只要及时将它重新启动即可。Nimbus 不是自举的，需要外部监控来及时重启。

### 5.7.3 集群节点（机器）故障

- Storm 集群中的节点故障。此时 Nimbus 会将此机器上所有正在运行的任务转移到其他可用的机器上运行。
- Zookeeper 集群中的节点故障。Zookeeper 保证少于半数的机器宕机系统仍可正常运行，及时修复故障机器即可。

## 5.8 小结

本章介绍了 Storm 集群如何实现数据的可靠处理。借助于创新性的 tuple tree 跟踪技术，Storm 高效地通过数据的应答机制保证数据不丢失。

Storm 集群中除 Nimbus 外，没有单点存在，任何节点出故障都可以保证数据不丢失。Nimbus 被设计为无状态的，只要可以及时重启，就不会影响正在运行的任务。

# 第6章

## 一致性事务

Storm 是一个分布式的流处理系统，利用 anchor 和 ack 机制保证所有 tuple 都被成功处理。如果 tuple 出错，则可以被重传，但是如何保证出错的 tuple 只被处理一次呢？Storm 提供了一套事务性组件 Transactional Topology，用来解决这个问题。

Transactional Topology 目前已经不再维护，由 Trident 来实现事务性 Topology，但是原理相同。

Storm 如何实现既对 tuple 并行处理，又保证事务性呢？本章从简单的事务性实现方法入手，逐步引出 Transactional Topology 的原理。

## 6.1 简单设计一：强顺序流

保证 tuple 只被处理一次，最简单的方法就是将 tuple 流变成强顺序的，并且每次只处理一个 tuple。从 1 开始，给每个 tuple 都顺序加上一个 id。在处理 tuple 时，将处理成功的 tuple id 和计算结果存在数据库中。下一个 tuple 到来时，将

其 id 与数据库中的 id 做比较。如果相同，则说明这个 tuple 已经被成功处理过了，那么可以忽略它；如果不同，根据强顺序性，说明这个 tuple 没有被处理过，将它的 id 及计算结果更新到数据库中。

以统计消息总数为例。每来一个 tuple，如果数据库中存储的 id 与当前 tuple id 不同，则数据库中的消息总数加 1，同时更新数据库中的当前 tuple id 值，如图 6-1 所示。

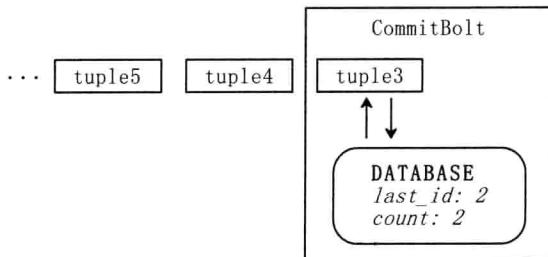


图 6-1

但是这种机制使得系统一次只能处理一个 tuple，无法实现分布式计算。

## 6.2 简单设计二：强顺序 batch 流

为了实现分布式，我们可以每次处理一批 tuple，称为一个 batch。一个 batch 中的 tuple 可以被并行处理。

我们要保证一个 batch 只被处理一次，机制和本书 7.1 节讲述的类似，只不过数据库中存储的是 batch id。batch 的中间计算结果先存在局部变量中，当一个 batch 中的所有 tuple 都被处理完之后，判断 batch id，如果跟数据库中的 id 不同，则将中间的计算结果更新到数据库中。

如何确保一个 batch 里面的所有 tuple 都被处理完了呢？可以利用 Storm 提供的 CoordinateBolt，如图 6-2 所示。

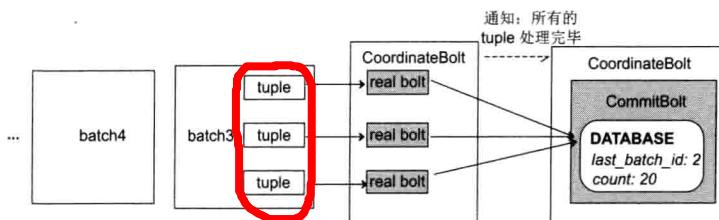


图 6-2

但是强顺序 batch 流也有局限性，每次只能处理一个 batch，batch 之间无法并行。要想实现真正的分布式事务处理，可以使用 Storm 提供的 Transactional Topology。在此之前，我们先详细介绍一下 CoordinateBolt 的原理。

### 6.3 CoordinateBolt 的原理

CoordinateBolt 的具体原理如下。

- 真正执行计算的 Bolt 外面封装了一个 CoordinateBolt。  
我们将真正执行任务的 Bolt 称为 real bolt。
- 每个 CoordinateBolt 记录两个值：有哪些 task 给我发送了 tuple（根据 Topology 的 grouping 信息），以及我要给哪些 task 发送信息（同样根据 grouping 信息）。
- Real bolt 发出一个 tuple 后，其外层的 CoordinateBolt 会记录下这个 tuple 发送给了哪个 task。
- 等所有的 tuple 都发送完之后，CoordinateBolt 通过另

外一个特殊的 Stream 以 emitDirect 的方式告诉所有它发送过 tuple 的 task，它发送了多少 tuple 给这个 task。下游 task 会将这个数字和自己已经接收到的 tuple 数量做对比，如果相等，则说明处理完了所有的 tuple。

- 下游 CoordinateBolt 会重复上面的步骤，通知其下游。

整个过程如图 6-3 所示。

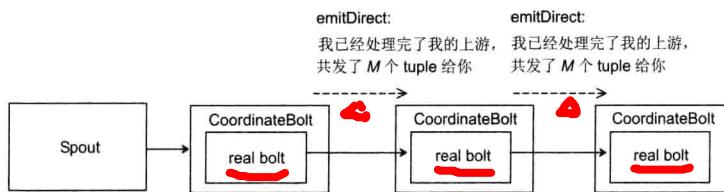


图 6-3

CoordinateBolt 主要用于如下两个场景。

- DRPC (Distributed RPC)
- Transactional Topology

CoordinateBolt 对于业务是有侵入的，要使用 CoordinateBolt 提供的功能，你必须要保证你的每个 Bolt 发

送的每个 tuple 的第一个 field 是 request-id。所谓的“我已经处理完我的上游”的意思是说当前这个 Bolt 对于当前这个 request-id 所需要做的工作做完了。这个 request-id 在 DRPC 里面代表一个 DRPC 请求；在 Transactional Topology 里面代表一个 batch。

## 6.4 Transactional Topology

Storm 提供的 Transactional Topology 将 batch 计算分为 process 和 commit 两个阶段。process 阶段可以同时处理多个 batch, 不用保证顺序性；commit 阶段保证 batch 的强顺序性，并且一次只能处理一个 batch，第 1 个 batch 成功提交之前，第 2 个 batch 不能被提交。

还是以统计消息总数为例，以下代码来自 storm-starter 里面的 TransactionalGlobalCount。

```
MemoryTransactionalSpout spout = new MemoryTransactionalSpout(
    DATA,
    new Fields("word"),
    PARTITION_TAKE_PER_BATCH
);
TransactionalTopologyBuilder builder = new TransactionalTopologyBuilder(
    "global-count",
    "spout",
    spout,
    3
);
builder.setBolt("partial-count", new BatchCount(), 5)
.noneGrouping("spout");
builder.setBolt("sum", new UpdateGlobalCount())
.globalGrouping("partial-count");
```

TransactionalTopologyBuilder 共接收如下 4 个参数。

- 这个 Transactional Topology 的 id，用来在 Zookeeper 中保存当前 Topology 的进度，如果这个 Topology 重启，可以继续之前的进度执行。
- Spout 在这个 Topology 中的 id。
- 一个 TransactionalSpout。一个 Transactional Topology 中只能有一个 TransactionalSpout，在本例中是一个 MemoryTransactionalSpout，从一个内存变量（DATA）中读取数据。
- TransactionalSpout 的并行度（可选）。

下面是 BatchCount 的定义。

```
public static class BatchCount extends BaseBatchBolt {  
    Object _id;  
    BatchOutputCollector _collector;  
    int _count = 0;  
  
    @Override  
    public void prepare(Map conf,  
                        TopologyContext context,  
                        BatchOutputCollector collector, Object id) {  
        _collector = collector;  
        _id = id;  
    }  
  
    @Override  
    public void execute(Tuple tuple) {  
        _count++;  
    }  
  
    @Override  
    public void finishBatch() {  
        _collector.emit(new Values(_id, _count));  
    }  
  
    @Override  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("id", "count"));  
    }  
}
```

BatchCount 的 prepare 方法的最后一个参数是 batch id，在 Transactional Topology 里面，这个 id 是一个 TransactionAttempt 对象。

Transactional Topology 里发送的 tuple 都必须以 TransactionAttempt 作为第一个 field，Storm 根据这个 field 来判断 tuple 属于哪一个 batch。

TransactionAttempt 包含两个值：一个是 transaction id，另一个是 attempt id。transaction id 的作用就是我们上面介绍的对于每个 batch 中的 tuple 是唯一的，不管这个 batch replay 多少次都是一样的。attempt id 是每个 batch 唯一的一个 id，但是对于同一个 batch，它 replay 之后的 attempt id 跟 replay 之前的不一样，我们可以把 attempt id 理解成 replay-times，Storm 利用这个 id 来区别一个 batch 发射的 tuple 的不同版本。

execute 方法会为 batch 里面的每个 tuple 执行一次，你应该把这个 batch 里面的计算状态保持在一个本地变量里面。对于这个例子来说，它在 execute 方法里面递增 tuple 的个数。

最后，当这个 Bolt 接收到某个 batch 的所有 tuple 之后，finishBatch 方法会被调用。这个例子里面的 BatchCount 类会在这个时候发射它的局部数量到它的输出流里面去。

UpdateGlobalCount 类的定义如下列代码所示。

```
public static class UpdateGlobalCount extends BaseTransactionalBolt
implements ICommitter {
    TransactionAttempt _attempt;
    BatchOutputCollector _collector;
    int _sum = 0;

    @Override
    public void prepare(Map conf,
                        TopologyContext context,
                        BatchOutputCollector collector,
                        TransactionAttempt attempt) {
        _collector = collector;
        _attempt = attempt;
    }

    @Override
    public void execute(Tuple tuple) {
        _sum += tuple.getInteger(1);
    }

    @Override
    public void finishBatch() {
        Value val = DATABASE.get(GLOBAL_COUNT_KEY);
        Value newval;

        if (val == null || !val.txid.equals(_attempt.getTransactionId())) {
            newval = new Value();
            newval.txid = _attempt.getTransactionId();

            if (val == null) {
                newval.count = _sum;
            } else {
                newval.count = _sum + val.count;
            }

            DATABASE.put(GLOBAL_COUNT_KEY, newval);
        } else {
            newval = val;
        }

        _collector.emit(new Values(_attempt, newval.count));
    }

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

UpdateGlobalCount 实现了 ICommitter 接口，所以 Storm 只会在 commit 阶段执行 finishBatch 方法，而 execute 方法可以在任何阶段完成。

在 UpdateGlobalCount 的 finishBatch 方法中，将当前的 transaction id 与数据库中存储的 id 做比较。如果相同，则忽略这个 batch；如果不同，则把这个 batch 的计算结果加到总结果中，并更新数据库。

Transactional Topology 的运行示意图如图 6-4 所示。

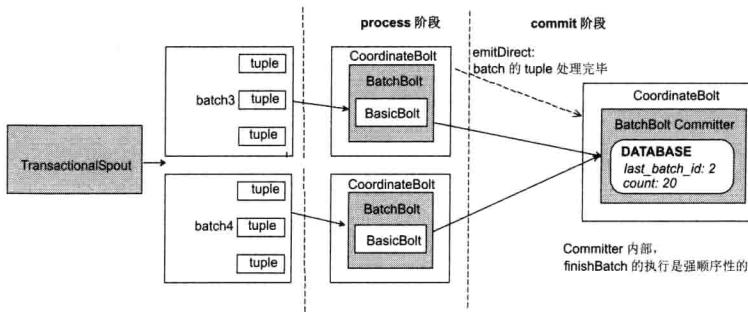


图 6-4

下面总结 Transactional Topology 的一些特性。

Transactional Topology 将事务性机制都封装好，其内部

使用 CoordinateBolt 保证一个 batch 中的 tuple 被处理完。

TransactionalSpout 只能有一个，它将所有 tuple 分组为一个一个的 batch，而且保证同一个 batch 的 transaction id 始终一样。

BatchBolt 处理一个 batch 中所有的 tuples。对于每一个 tuple 调用 execute 方法，而在整个 batch 处理完成时调用 finishBatch 方法。

如果 BatchBolt 被标记成 Committer，则只能在 commit 阶段调用 finishBolt 方法。一个 batch 的 commit 阶段由 Storm 保证只在前一个 batch 成功提交之后才会执行，并且它会重试直到 Topology 里面的所有 Bolt 在 commit 完成提交。

Transactional Topology 隐藏了 anchor/ack 框架，它提供了一个不同的机制来 fail 一个 batch，从而使得这个 batch 被 replay。

# 第7章

---

# DRPC

Distributed RPC (Remote Process Call)

## 7.1 Storm DRPC

DRPC 用于对 Storm 上大量的函数调用进行并行计算。对于每一次函数调用，Storm 集群上运行的拓扑接收调用函数的参数信息作为输入流，将计算结果作为输出流发射出去。

DRPC 本身算不上 Storm 的特性，它是通过 Storm 的基本元素 Streams、Spouts、Bolts、Topologies 衍生的一个模式。DRPC 可以单独作为一个独立于 Storm 的库发布，但由于其重要性，所以还是和 Storm 捆绑在了一起。

## 7.2 总体概述

DRPC 通过 DRPC Server 来实现，DRPC Server 的整体工作过程如下。

- (1) 接收到一个 RPC 调用请求。

(2) 发送请求到 Storm 上的拓扑。

(3) 从 Storm 上接收计算结果。

(4) 将计算结果返回给客户端。

以上过程，在 client 客户端看来，一个 DRPC 调用看起来和一般的 RPC 调用没什么区别。下面的代码是 client 通过 DRPC 调用 “reach” 函数，参数为 “http://twitter.com”。

```
DRPCClient client = new DRPCClient("drpc-host", 3772);
String result = client.execute("reach", "http://twitter.com").
```

DRPC 的内部工作流程如图 7-1 所示。

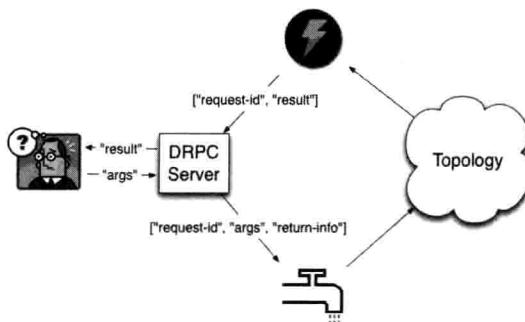


图 7-1

(1) Client 向 DRPC Server 发送被调用执行的 DRPC 函数

名称及参数。

(2) Storm 上的 Topology 通过 DRPCSpout 实现这一函数，从 DRPC Server 接收到函数调用流。

(3) DRPC Server 会为每次函数调用生成唯一的 id。

(4) Storm 上运行的 Topology 开始计算结果，最后通过一个 ReturnResults 的 Bolt 连接到 DRPC Server，发送指定 id 的计算结果。

(5) DRPC Server 通过使用之前为每个函数调用生成的 id，将结果关联到对应的发起调用的 client，将计算结果返回给 client。

### 7.3 LinearDRPCTopologyBuilder

LinearDRPCTopologyBuilder 是 Storm 提供的一个 Topology builder，它可以自动完成几乎所有的 DRPC 步骤。这其中包括构建 Spout、向 DRPC Server 返回结果、为 Bolt 提供函数

用于对 tuples 进行聚集。

下面是一个简单的例子，这个 DRPC 拓扑只是简单地在输入参数后追加 “!” 后返回：

```
public static class ExclaimBolt extends BaseBasicBolt {  
    public void execute(Tuple tuple, BasicOutputCollector collector) {  
        String input = tuple.getString(1);  
        collector.emit(new Values(tuple.getValue(0), input + "!"));  
    }  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("id", "result"));  
    }  
}  
  
public static void main(String[] args) throws Exception {  
    LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("exclamation");  
    builder.addBolt(new ExclaimBolt(), 3);  
    // ...  
}
```

由上述例子可见，我们只须很少的工作即可完成拓扑。当创建 LinearDRPCTopologyBuilder 的时候，需要指定拓扑中 DRPC 函数的名称“exclamation”。一个 DRPC Server 可以协调多个函数，每个函数有不同的函数名称。拓扑中的第一个 Bolt 的输入是两个字段：第一个是请求的 id 号，第二个是请求的参数。

LinearDRPCTopologyBuilder 同时需要最后一个 Bolt 发射一个包含两个字段的输出流：第一个字段是请求 id，第二

个字段是计算结果。因此，所有的中间 tuples 必须包含请求 id 作为第一个字段。

例子中，ExclaimBolt 在输入 tuple 的第二个字段后面追加 “!”，LinearDRPCTopologyBuilder 负责处理其余的协调工作：与 DRPC Server 建立连接，发送结果给 DRPC Server。

## 7.4 本地模式 DRPC

DRPC 可以以本地模式运行，下面的代码是如何在本地模式运行的例子。

```
LocalDRPC drpc = new LocalDRPC();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("drpc-demo", conf, builder.createLocalTopology(drpc));
System.out.println("Results for 'hello': " + drpc.execute("exclamation", "hello"));
cluster.shutdown();
drpc.shutdown();
```

首先创建一个 LocalDRPC 对象，该对象在本地模拟一个 DRPC Server，正如 LocalCluster 在本地模拟一个 Storm 集群一样。然后创建一个 LocalCluster 对象在本地模式下运行拓

扑。LinearDRPCTopologyBuilder 含有单独的方法用于创建本地拓扑和远程拓扑。

本地模式下, LocalDRPC 并不绑定任何端口, 因此 Storm 的拓扑需要了解要通信的对象——这就是为什么 createLocalTopology 方法需要以 LocalDRPC 对象作为输入的原因。

加载完拓扑之后, 通过对 LocalDRPC 调用 execute 方法, 就可以执行 DRPC 函数调用了。

## 7.5 远程模式 DRPC

在实际的 Storm 集群上运行 DRPC 也一样很简单。只须完成以下步骤。

- (1) 启动 DRPC Server(s)。
- (2) 配置 DRPC Server(s)地址。
- (3) 向 Storm 集群提交 DRPC 拓扑。

首先，通过 storm 脚本启动 DRPC Server：

```
$ bin/storm drpc
```

然后，在 Storm 集群中配置 DRPC Server 地址，这就是 DRPCSpout 读取函数调用请求的地方。这一步的配置可以通过 storm.yaml 文件或者拓扑的配置来完成。通过 storm.yaml 文件的配置方式如下：

```
drpc.servers:  
  - "drpc1.foo.com"  
  - "drpc2.foo.com"
```

最后，通过 StormSubmitter 启动 DRPC 拓扑。以远程模式运行上面例子的代码如下：

```
StormSubmitter.submitTopology(  
    "exclamation-drpc",  
    conf,  
    builder.createRemoteTopology()  
,
```

createRemoteTopology 被用于为 Storm 集群创建合适的拓扑。



## 7.6 一个复杂的例子

上面的 `exclamation` 只是一个简单的 DRPC 的例子。下面通过一个复杂的例子介绍如何在 Storm 集群内进行 DRPC——计算 Twitter 上每个 URL 的到达度（reach），也就是每个 URL 暴露给不同人的个数。

为了完成这一计算，需要完成以下步骤。

- (1) 获取所有发布了该 URL 的人。
- (2) 获取步骤 1 中所有人的关注者（followers）。
- (3) 对所有关注者进行去重。
- (4) 对步骤 3 中的关注者人数进行求和。

一个简单的 URL 到达度计算可能涉及成千上万次数据库调用及数以百万的 `followers` 记录，计算量非常大。有了 Storm，将很容易实现这一计算过程。单机上可能需要运行几分钟才能完成，在 Storm 集群上，即使是最难计算的 URL 也

只需要几秒钟。

这个例子的代码如何在 storm-starter<sup>1</sup>创建拓扑的代码抽取如下。

```
LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("reach");
builder.addBolt(new GetTweeters(), 3);
builder.addBolt(new GetFollowers(), 12)
    .shuffleGrouping();
builder.addBolt(new PartialUniquer(), 6)
    .fieldsGrouping(new Fields("id", "follower"));
builder.addBolt(new CountAggregator(), 2)
    .fieldsGrouping(new Fields("id"));
```

拓扑的执行分为以下四步。

(1) **GetTweeters**: 获取所有发布了指定 URL 的用户列表，这个 Bolt 将输入流[id, url]转换成输出流[id, tweeter]，每个 url 元组被映射为多个 tweeter 元组。

(2) **GetFollowers**: 获取步骤 1 中所有用户列表的 follower，这个 Bolt 将输入流[id, tweeter]转换成输出流[id, follower]，当某个人同时是多个人的 follower，而且这些人都发布了指

---

<sup>1</sup> <https://github.com/nathanmarz/storm-starter/blob/master/src/jvm/Storm/starter/ReachTopology.java>

定的 URL，那么将产生重复的 follower 元组。

(3) PartialUniquer: 将所有 follower 按照 follower id 分组，使得同一个 follower 在同一个 task 中被处理。这个 Bolt 接收 follower 并进行去重计数。

(4) CountAggregator: 从各个 PartialUniquer 中接收各部分的计数结果，累加后完成到达度计算。

下面是 PartialUniquer 这个 Bolt 的代码实现：

```
public class PartialUniquer extends BaseBatchBolt {
    BatchOutputCollector _collector;
    Object _id;
    Set<String> _followers = new HashSet<String>();

    @Override
    public void prepare(Map conf,
                        TopologyContext context,
                        BatchOutputCollector collector,
                        Object id) {
        _collector = collector;
        _id = id;
    }

    @Override
    public void execute(Tuple tuple) {
        _followers.add(tuple.getString(1));
    }

    @Override
    public void finishBatch() {
        _collector.emit(new Values(_id, _followers.size()));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id", "partial-count"));
    }
}
```

PartialUniquer 通过继承 BaseBatchBolt 实现了 I<sub>B</sub>atchBolt 接口， batch bolt 提供 API 用于将一批 tuples 作为整体处理。每个请求 id 会创建一个新的 batch bolt 实例，同时 Storm 负责这些实例的清理工作。

当 PartialUniquer 接收到一个 follower 元组时执行 execute 方法，将 follower 添加到请求 id 对应的 HashSet 集合中。

Batch bolt 同时提供了 finishBatch 方法用于当这个 task 已经处理完所有的元组时调用。PartialUniquer 发送一个包含当前 task 所处理的 follower id 子集去重后个数的元组。

在内部实现上，CoordinatedBolt 用于检测指定的 Bolt 是否已经收到指定请求 id 的所有 tuples 元组。CoordinatedBolt 使用 direct streams 管理实现这一协作过程。

拓扑的其他部分易于理解。到达度的每一步的计算过程都是并行进行的，通过 DRPC 实现也是非常容易的。

## 7.7 非线性 DRPC 拓扑

LinearDRPCTopologyBuilder 只能处理“线性的”DRPC 拓扑——正如到达度这样可以通过一系列步骤序列来完成的计算。不难想象，DRPC 调用中包含有更复杂的带有分支和合并 Bolt 的拓扑。目前，必须自己直接使用 CoordinatedBolt 来完成这种非线性拓扑的计算。

## 7.8 LinearDRPCTopologyBuilder 工作过程

- DRPCSpout 发射[args, return-info]，其中 return-info 包含 DRPC Server 的主机和端口号，以及 DRPC Server 为该次请求生成的唯一 id 号。
- 构造一个 Storm 拓扑包含以下部分：
  - DRPCSpout

- PrepareRequest（生成一个请求 id，为 return info 创建一个流，为 args 创建一个流）
- CoordinatedBolt wrappers 和 direct groupings
- Join Result（将结果与 return info 拼接起来）
- Return Result（连接到 DRPC Server，返回结果）
- LinearDRPCTopologyBuilder 是建立在 Storm 基本元素之上的高层抽象。

## 7.9 高级进阶

- KeyedFairBolt 用于组织同一时刻多请求的处理过程。
- 如何直接使用 CoordinatedBolt。

# 第8章

## Trident 的 特性

Trident 是基于 Storm 进行实时流处理的高级抽象，提供了对实时流的聚集、投影、过滤等操作，从而大大简化开发 Storm 任务的工作量。如果你使用 Pig 或 Cascading，对这些接口就不会陌生。另外，Trident 提供了原语处理针对数据库或其他持久化存储的有状态的、增量的更新操作。

## 8.1 理解 Trident

若我们要开发一个对文本中的词频进行统计的程序，使用 Storm 框架，我们需要开发三个 Storm 组件。

- (1) 一个 Spout 收集文本信息并分段，并作为 sentence 字段发送给下游的 Bolt。
- (2) 一个 Bolt 将每段文本分词，将分词结果以 word 字段发送给下游的 Bolt。
- (3) 一个 Bolt 对词频进行统计，将统计结果记录在 count 字段并存储起来。

若使用 Trident，我们用如下代码就可以完成这个任务：

```
1. FixedBatchSpout spout = new FixedBatchSpout(new Fields("sentence"), 3,
    new Values("the cow jumped over the moon"),
    new Values("the man went to the store and bought some candy"),
    new Values("four score and seven years ago"),
    new Values("how many apples can you eat"),
2. spout.setCycle(true);
3. TridentTopology topology = new TridentTopology();
4. TridentState wordCounts =
5. topology.newStream("spout1", spout)
6.     .each(new Fields("sentence"), new Split(), new Fields("word"))
7.     .groupBy(new Fields("word"))
8.     .persistentAggregate(new MemoryMapState.Factory(),
    new Count(),
    new Fields("count"))
9.     .parallelismHint(6);
```

这段代码将被 Trident 框架转换为上面列出的 3 个步骤。

代码的前两行使用 FixedBatchSpout 不断循环生成参数中列出的 5 个句子，第 3 行声明了 TridentTopology 对象，并在第 5 行的 newStream 方法中引入了 FixedBatchSpout。

Trident 是按批处理数据的，FixedBatchSpout 生成的数据是按图 8-1 所示的方式一批一批发送到下一个处理单元的，后续处理单元也是以类似的方式发送数据到其他节点的。

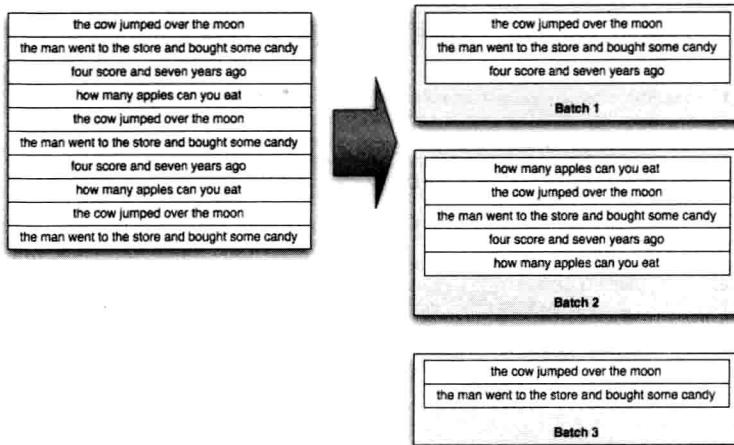


图 8-1

在上述代码的第 6 行使用 Split 将 FixedBatchSpout 生成的句子进行了分词并存储在 word 字段中。Split 的定义如下：

```
public class Split extends BaseFunction {  
    public void execute(TridentTuple tuple, TridentCollector collector) {  
        String sentence = tuple.getString(0);  
  
        for(String word: sentence.split(" ")) {  
            collector.emit(new Values(word));  
        }  
    }  
}
```

在 each 方法中也可以实现过滤操作。

```
stream.each(new Fields("sentence"), new MyFilter())
public class MyFilter extends BaseFilter {
    public boolean isKeep(TridentTuple tuple) {
        return tuple.getString(0).length() >= 10;
    }
}
```

上面的语句会过滤掉所有长度小于 10 个字符的句子。

代码`.groupBy(new Fields("word"))`对 word 进行聚集操作并在之后使用 Count 对象进行计数。之后将统计的结果存储在内存中。Trident 也支持存储到其他介质如数据库、Memcached 等处，比如下面的代码将数据存储到 memcached。

```
.persistentAggregate(MemcachedState.transactional(serverLocations),
    new Count(),
    new Fields("count"))
```

这样数据将以`<word, count>`的 key/value 形式存储到 memcached 中。

实时任务的关键问题是如何处理对数据更新的幂等问题，任务可能失败或重启，因此更新操作可能被重复执行。以上面的例子为例，发送到 Count 的数据可能由于节点的重启或其他网络原因被重复发送，从而引起数据的重复统计。为了避免这个问题，Trident 提供了事务支持。由于数据（例

子中的单词）是按批发送到 Count 节点的，Trident 对每批单词都分配一个 Transaction id。上面的代码中，每完成一批单词的计数，就将这批数据的统计结果连同 Transaction id 存储到 memcached 中。Trident 会比较 memcached 中的 Transaction id 和新到达数据的 Transaction id，如果同一批数据被重复发送，其 Transaction id 就会等于 memcached 中记录的 Transaction id，新数据将会被忽略。另外，每批数据的 Transaction id 是严格有序的，Transaction id 2 的数据如果没有处理完，Transaction id 3 的数据是不会被处理的。

## 8.2 结合多个 Trident 任务

一个任务可能有多个数据源，每个数据源都是以 TridentState 的形式出现在任务定义中的。比如本书 8.1 节中讲到的 wordCount 任务生成的数据就可以被其他任务所使用。

下面的例子使用之前定义的 wordCount 任务和 Storm 的

DRPC 功能，统计一段文本中的所有单词在 wordCount 中的词频总和。

```
1. topology.newDRPCStream("words")
2.     .each(new Fields("args"), new Split(), new Fields("word"))
3.     .groupBy(new Fields("word"))
4.     .stateQuery(wordCounts, new Fields("word"),
new MapGet(), new Fields("count"))
5.     .each(new Fields("count"), new FilterNull())
6.     .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

之后就可以使用下面的代码进行计算。

```
DRPCCClient client = new DRPCCClient("drpc.server.location", 3772);
System.out.println(client.execute("words", "cat dog the man"));
```

代码中第 1 行声明 DRPC 服务 words，第二行对 DRPC 的参数"cat dog the man"分词，第 3 行和第 4 行对单词分组，之后对每个单词查询之前 wordCount 的 TridentState 对象，返回每个单词的词频。第 5 行和第 6 行将这些词频加和。

## 8.3 消费和生产 Field

Trident 的数据模型称作 “TridentTuple” —— 带名字的 Values 列表。在 Topology 中，tuple 是在顺序的操作集合 (Operation) 中增量生成的。Operation 通常包含一组输入字段 (input field) 和提交的功能字段 (function filed)。Operation 的输入字段通常是将 tuple 的一个子集作为操作集合的输入，而功能字段则是命名提交的字段。

### 8.3.1 Field 例子解释

声明一个名为 “students” 的 Stream，可能包含名字、性别、学号、分数等字段。添加一个按 “score” 过滤的 Filter – ScoreFilter，使得 tuples 只过滤出分数大于 60（合格）的学生：

```
students.each(new Fields("score"), new ScoreFilter()),
```

定义 “分数” 过滤器：

```

public class ScoreFilter extends BaseFilter {
    public boolean isKeep(TridentTuple tuple) {
        return tuple.getInteger(0) > 60;
    }
}

```

TridentTuple 输入的参数只含有“score”字段。现在，我们已经过滤出分数合格的学生。当选择输入字段时，Trident 会自动映射出一个 tuple 的子集（操作是非常高效的）。

### 8.3.2 Function Field 例子解释

如果我们想对字段进行运算，并且提交给 TridentTuple，可以模拟以下计算：

```

public class AddAndSubFunction extends BaseFunction {
    public void execute(TridentTuple tuple, TridentCollector collector) {
        int v1 = tuple.getInteger(0);
        int v2 = tuple.getInteger(1);
        int sub = v1 > v2 ? v1 - v2 : v2 - v1;
        collector.emit(new Values(v1 + v2, sub));
    }
}

```

此函数接收两个整型作为参数，并计算差和，作为两个新 Field 进行提交。

```
Calculate.each(new Fields("value_a", "value_b"),
    new AddAndSubFunction (),
    new Fields("Add", "Sub"));
```

新增的功能字段将被追加到输入 tuple 中。每个 tuple 中将比原来多出字段“Add”和“Sub”。

如果我们使用已有的聚合函数，如：

```
// 结果包含"Sum"
Calculate.aggregate(new Fields("value_a"), new Sum(), new Fields("Sum"))

// 结果包含"other_value"和"Sum"
Calculate.groupBy(new Fields("other_value"))
    .aggregate(new Fields("value_a"), new Sum(), new Fields("Sum"))
```

不同的是，经过 aggregate 聚合函数之后的 Fields 将只包含“Sum”字段，或者可以通过 groupBy 添加 grouping fields。

## 8.4 State（状态保存）

Trident 在读写有状态的数据源方面有着较好的抽象。这些状态可以是 Topology 内部的，如基于 HDFS 的内存式（如 HBase 的工作方式），也可以是外部存储，如 Memcached 或者 Cassandra。他们对应的 Trident API 是一致的。

Trident 以容错的方式来管理状态，因此当遇到重试或者错误时状态的更新是幂等，可以把 Trident Topology 理解成每个消息只会被处理一次。

我们来看一个例子，假定你在对一个流做计数处理，同时把计数结果存到数据库中。如果在数据库中用一个值来表示这个计数，然后每处理一个 tuple，就将数据库存储的计数加 1。

当错误发生时，tuple 会被重新处理。这就引发了一个问题，当进行状态更新时，你完全不知道之前是否已经成功处理过这个 tuple。也许之前从来没处理过这个 tuple，那么就应该把计数加 1。也有可能之前已经成功处理过这个 tuple，同时已经把计数加 1 了，但是这个 tuple 在其他处理环节失败了。这种情况下，不能将计数加 1。又或者，之前处理过这个 tuple，但是在更新数据库时失败了，若发生这种情况，则应该更新数据库。

若数据库中只存一个计数的话，是区分不出这个 tuple 之前是否被正确处理过的，需要更多的信息来支持。Trident

提供了下面的语义来实现只处理一次的语义。

- (1) tuples 是被分成一组组小的集合 (batch) 来处理的<sup>1</sup>。
- (2) 每一个 batch 会被分配一个唯一的 id (即事务 id, txid)，当 batch 被重新处理时，txid 是不变的。
- (3) batch 之间的状态更新是严格有序的。也就是说，batch2 的状态更新完成了才能做 batch3 的状态更新。

有了这些原语，在处理状态更新时就能知道这个 batch 之前有没有被处理过，然后采取合适的操作即可。下面让我们一起看看每一种 spout 类型能够支持什么样的容错级别。

#### 8.4.1 Transactional Spouts

Trident 是以 batch 的方式来处理 tuple 的，同时每个 batch 会被分配一个唯一的 Transaction id。Transactional Spout 满足以下条件。

---

<sup>1</sup> <https://github.com/nathanmarz/storm/wiki/Trident-tutorial>

(1) 一个 batch 无论重发多少次，只有一个唯一且相同的事务 id，同时所包含的 tuple 都是完全一致的。

(2) 一个 tuple 必须且最多属于一个 batch。

Transactional Spout 很容易理解，但在极端情况下理解起来也会有一些问题。假设一批消息在被 bolt 消费的过程中失败了，需要 Spout 重发，这时，如果刚好消息发送中间件故障（比如节点宕机了或者订阅对应的分区无法访问等），Spout 为了保证重发时每批包含的 tuple 一致，就只能等待消息中间件恢复，整个处理就卡住了。

我们来看下面的例子。

设计一个计算 WordCount 的 Topology，将单词的出现次数以 KV 方式存储到数据库中。key 就是单词，value 就是这个单词出现的次数。可以将 value 和 Transaction id 一起存储到数据库。当某次更新 value 前，先将当前的 Transaction id 和数据里的 id 进行对比。如果一样，则忽略；否则就执行存储。Trident 会保证 State 的更新在 batch 里是顺序的。

Batch (Trasaction id为3)

1. [“man”]
2. [“man”]
3. [“dog”]

数据库中保存了如下信息：

1. man => [count=3, txid=1]
2. dog => [count=4, txid=3]
3. apple => [count=10, txid=2]

单词“man”对应的 Transaction id 是 1，因为当前的 Transaction id 是 3，可以确定没有为这个 batch 中的 tuple 更新过这个单词的数量，所以直接更新 Transaction id 为 3。同时，若单词“dog”的 Transaction id 和当前的 Transaction id 是相同的，则忽略更新。单词“apple”保持不变。更新后的数据如下。

1. man => [count=5, txid=3]
2. dog => [count=4, txid=3]
3. apple => [count=10, txid=2]

#### 8.4.2 Opaque Transactional Spout

Opaque Transactional Spout 并不能确保一个 Transaction id 对应的 batch 的一致性，有如下属性。

tuple 只在一个 batch 中被成功处理，如果 tuple 在一个 batch 中被处理失败，有可能会在另外一个 batch 中被成功处理。也就是说，某个 tuple 可能第一次在 txid=2 的 Batch 中出现，以后有可能在 txid = 4 的 batch 中再次出现。

Opaque Transactional Spout 具有更好的容错性，但是需要额外的“存储空间”。除了 value 和 Transaction id，你还需要在数据库中存储之前的数据（preValue）。我们再来看看数据库中存储一个计数的例子，假设当前数据库中存储的信息如下：

```
{  
    value = 4,  
    preValue = 1,  
    txid = 2  
}
```

下一个 batch 的 Transaction id 是 3，计数值为 2，和数据库中的 Transaction id 不同。这种情况下，将 value 中的值放到 preValue，新增的值加到 value 上去，更新后的数据库信息如下：

```
{  
    value = 6,  
    prevValue = 4,  
    txid = 3  
}
```

如果当前 batch 的 Transaction id 还是 2，与数据库中存的刚好相同，那么怎么操作呢？我们知道，数据库中的 value 值是通过与这次 Transaction id 相同的前一个 batch 更新而来的，但是 batch 可能已经变化了，所以我们要忽略它，这种情况下需要做的就是更新 value 的值为 prevValue 加上本次 batch 的值，结果应该是这样：

```
{  
    value = 3  
    prevValue = 1,  
    txid = 2  
}
```

此方式的正确性基于 Trident 保证了 batch 之间的“强顺序”性。Trident 处理一个新的 batch 时，一定不会重复或者回溯到之前的 batch。每个 tuple 只会在一个 batch 中被“成功”处理，所以更新是“原子”的。

#### 8.4.3 Non-transactional Spout

Non-transactional Spout（非事务 Spout）不保证 batch 中

tuple 的处理情况。因此，如果 batch 处理失败后 tuple 不重发，那么可能“至多处理一次”，当 batch 都处理成功，那么可能“至少处理一次”。但是这种 Spout 不能保证“有且只有一次”的语义。

#### 8.4.4 Spout 和 State 类型的小结

图 8-2 是 Spout 和 State 的组合，可以实现“有且只有一次”的特性（“Yes”代表可以实现），不同的实现在“容错性”和“存储消耗”做着不同的平衡，根据不同的适用场景做出选择。

		State		
		Non-transactional	Transactional	Opaque transactional
Spout	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

图 8-2

## 8.5 Trident Topology 的执行过程

Trident Topology 会被编译成高效的 Storm Topology。只有当数据需要进行重新分区（repartition）时，如 group by 或者 shuffle，才会通过网络传输 Tuples。

如果有这样的 Trident Topology，如图 8-3 所示，那么会被编译成如图 8-4 所示的 Spout 和 Bolt 的集合。

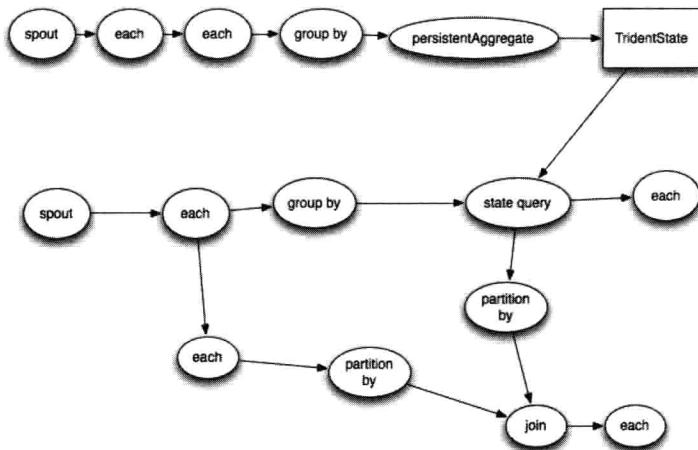


图 8-3

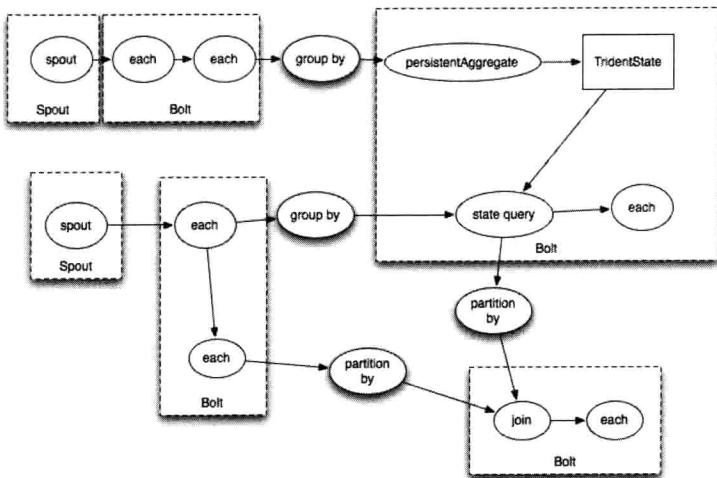


图 8-4

## 8.6 总结

Trident 为我们提供了一种开发 Storm 的更便捷的方式，并且其对事务的支持也使得使用 Storm 开发可靠的分布式应用成为可能。

# 第9章

## Storm 实例

## 9.1 一个简单的实例

我们模拟一个简化的网站统计场景进行实例演示。

在此实例中，Spout 组件（LogReader）模拟用户浏览网站的行为，产生访问日志；然后 Bolt 组件（LogStat）统计每个用户的页面浏览量（PV），最后 Bolt 组件（LogWriter）将统计结果输出（为了简化非 Storm 相关内容，这里我们把结果直接输出到 Storm 日志）。

访问日志的格式如下。

- time 访问时间
- user 访客
- url 被访页面

Topology 结构示意图如图 9-1 所示。

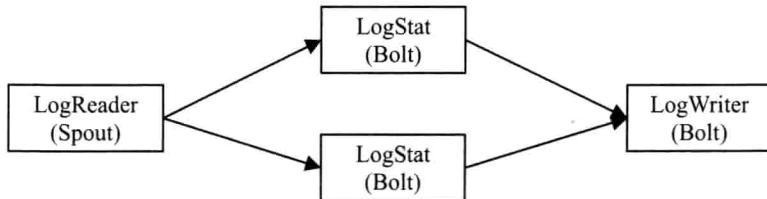


图 9-1

根据图 9-1，我们来定义 Topology 结构。

其代码如下：

```

public static void main(String[] args) throws Exception {
    TopologyBuilder builder=new TopologyBuilder();

    builder.setSpout("log-reader", new LogReader(), 1);

    builder.setBolt("log-stat", new LogStat(), 2).
        fieldsGrouping("log-reader", new Fields("user"));

    builder.setBolt("log-writer", new LogWriter(), 1).
        shuffleGrouping("log-stat");

    Config conf = new Config();
    StormSubmitter.submitTopology(
        "log-topology",
        conf,
        builder.createTopology()
    );
}
  
```

上面代码中，Storm 库提供 TopologyBuilder 类帮助我们在 Storm 集群中构建 Topology。初始化 TopologyBuilder 实例，利用该实例配置一个 Spout (LogReader)，两个 Bolt (LogStat

和 LogWriter)，再实例化 Topology 配置信息，在这里我们使用默认配置。最后利用 StormSubmitter 的静态方法将 Topology 提交到 Storm 集群。

在上面的代码中，setSpout、setBolt 方法的第一个参数是组件 ID，被用来唯一标识同一个 Topology 中的相同组件，其他组件可以通过该 ID 订阅其输出结果。第二个参数是组件的实例对象，第三个参数定义集群为每个组件分配的 Executor 数。fieldsGrouping、shuffleGrouping 声明了子节点订阅上游组件流的方式。

### 9.1.1 LogReader

LogReader 作为 Topology 的 Spout 组件实现，模拟用户访问并输出日志。正式生产环境中，该组件会读取保存用户访问日志的数据源。

其代码如下：

```

public class LogReader extends BaseRichSpout {
    private SpoutOutputCollector _collector;
    private Random _rand = new Random();
    private int _count = 100;
    private String[] _users = {"userA", "userB", "userC", "userD", "userE"};
    private String[] _urls = {"url1", "url2", "url3", "url4", "url5"};

    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
    }

    public void nextTuple() {
        try {
            Thread.sleep(1000);
            while (_count-- > 0) {
                _collector.emit(
                    new Values(
                        System.currentTimeMillis(),
                        _users[_rand.nextInt(5)],
                        _urls[_rand.nextInt(5)]
                    )
                );
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("time", "user", "url"));
    }
}

```

在上面的代码中，我们定义了继承自 BasicRichSpout 的类 LogReader，重写了方法 open、nextTuple 和 declareOutputFilelds。其中 open 方法在集群初始化该组件的一个任务时执行，其包含 3 个参数。第一个参数 conf 包含该 Topology 及集群 Storm 的配置信息，第二个参数 context 包含

该 task 任务的 ID、组件 ID 等 Topology 上下文信息，第三个参数 collector 用来输出 tuple，可以保存该实例，随时（如 open、close 方法中）用来发送 tuple，它是线程安全的。方法 nextTuple 被调用时 Storm 从 Spout 的输出中取出数据向下传递，通常我们都在 nextTuple 方法中输出数据。nextTuple 方法和 ack、fail 在同一个线程中由 Storm 轮询调度，因此它必须是非阻塞的，在示例中，Storm 每次调用 nextTuple 方法时我们模拟生成一条用户访问日志输出，每条日志包含 3 个字段，第一个字段是访问时间，第二个字段是访问用户，第三个字段代表用户访问的页面，这 3 个字段在方法 declareOutputFields 中定义，如示例显示这 3 个字段分别为 time、user 和 url。

### 9.1.2 LogStat

LogStat 作为 Topology 的 Bolt 组件订阅 LogReader 组件的输出结果，这里简单统计了每个用户的 PV 数，然后将统计结果输出（输出结果包含两个字段 user 和 pv）。

其代码如下：

```
public class LogStat extends BaseRichBolt {  
    private OutputCollector _collector;  
    private Map<String, Integer> _pvMap = new HashMap<String, Integer>();  
  
    public void prepare(Map stormConf,  
                        TopologyContext context,  
                        OutputCollector collector) {  
        _collector = collector;  
    }  
  
    public void execute(Tuple input) {  
        String user = input.getStringByField("user");  
  
        if (_pvMap.containsKey(user))  
            _pvMap.put(user, _pvMap.get(user)+1);  
        else  
            _pvMap.put(user, 1);  
  
        _collector.emit(new Values (user, _pvMap. Get (user)));  
    }  
  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("user", "pv"));  
    }  
}
```

在上面代码中定义了继承自 BaseRichBolt 的类 LogStat，重写了方法 prepare、execute 和 declareOutputFilelds。其中 prepare 方法类似于 LogReader 中的 open 方法，在集群初始化该组件的一个任务时执行，参数含义参见 LogReader。方法 execute 被调用时 Storm 从 Spout 的输出中取出一个 tuple 以供处理，在 excute 方法中我们对用户的 PV 进行了累加并将每个用户的最新 PV 输出到下一节点，类似 LogReader 向下输出的字段结构在 declareOutputFields 中定义。

### 9.1.3 LogWriter

LogWriter 作为 Topology 的 Bolt 组件订阅 LogStat 组件的输出结果，这里简单统计结果输出到日志。作为 Topology 中的最后任务组件，它不再往下游输出数据。

其代码如下：

```
public class LogWriter extends BaseRichBolt {  
    public void prepare(Map stormConf,  
                        TopologyContext context,  
                        OutputCollector collector) {  
    }  
    public void execute(Tuple input) {  
        System.out.println(  
            String.format("%s:%d",  
                         input.getStringByField("user"),  
                         input.getIntegerByField("pv"))  
        );  
    }  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
    }  
}
```

因为不需要再往下游输出数据，因此在 `prepare` 方法中未将 `collector` 存储到局部变量。`declareOutputFields` 方法中未定义新的输出字段结构。在 `execute` 方法中只是简单地将结果输出到了 Storm Worker 日志（`worker-****.log`）中。

代码如下：

```

...
2012-12-18 10:55:17 STDIO [INFO] userA:12
2012-12-18 10:55:17 STDIO [INFO] userC:13
2012-12-18 10:55:17 STDIO [INFO] userE:16
2012-12-18 10:55:17 STDIO [INFO] userA:13
2012-12-18 10:55:17 STDIO [INFO] userE:17
2012-12-18 10:55:17 STDIO [INFO] userC:14
...

```

上面例子中，每发生一条访问记录，`LogWriter` 都会在日志中写出用户的 PV。现在我们做一些小小的变更，我们希望 `LogWriter` 只在最后统一输出所有用户的 PV。接下来我们稍微修改一下代码。Topology 结构示意图的变更如图 9-2 所示。

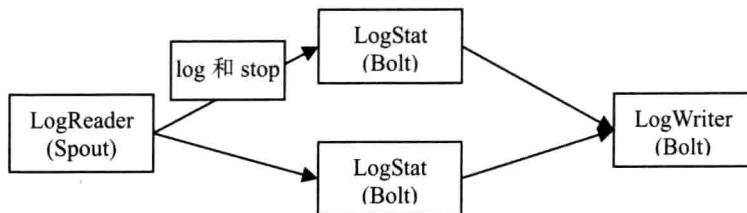


图 9-2

我们为 `LogReader` 输出定义了两个流 `log` 和 `stop`。其中流 `log` 用来向下游输出日志，流 `stop` 用来告诉下游日志全部输出完毕。在第一个示例中，我们没有为流定义名称，Storm

默认使用 default 代替隐藏处理。

相应地，我们将 Topology 定义的代码修改如下：

```
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("log-reader", new LogReader(), 1);
    builder.setBolt("log-stat", new LogStat(), 2)
        .fieldsGrouping("log-reader", "log", new Fields("user"))
        .allGrouping("log-reader", "stop");
    builder.setBolt("log-writer", new LogWriter(), 1)
        .shuffleGrouping("log-stat");
    Config conf = new Config();
    conf.setNumWorkers(5);
    StormSubmitter.submitTopology(
        "log-topology",
        conf,
        builder.createTopology()
    );
}
```

代码的 4~6 行做了修改，可以看到 LogWriter 订阅了两条来自 LogReader 的流，流 log 按字段分组，流 stop 全局广播。在第一个示例中，我们并没有指定流 ID，Storm 默认订阅了 default 流。

#### 9.1.4 LogReader

现在我们修改一下 nextTuple 和 declareOutputFields 方

法，在 `declareOutputFields` 方法中，我们定义两个输出流，并在 `nextTuple` 方法中增加分支判断，日志全部生成后向 stop 流输出一条空记录。

其代码如下：

```
public void nextTuple() {
    try {
        Thread.sleep(1000);
        while (_count-- >= 0) {
            if (_count == 0) {
                _collector.emit("stop", new Values(""));
            } else {
                _collector.emit(
                    "log",
                    new Values(
                        System.currentTimeMillis(),
                        _users[_rand.nextInt(5)],
                        _urls[_rand.nextInt(5)]
                    )
                );
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declareStream("log", new Fields("time", "user", "url"));
    declarer.declareStream("stop", new Fields("flag"));
}
```

### 9.1.5 LogStat

修改 execute 方法，先获取流 ID，如果是流 log，则统计 PV 但不向下游节点发送数据。遇到流 stop 时批量输出结果。

其代码如下：

```
public void execute(Tuple input) {
    String streamId = input.getSourceStreamId();

    if (streamId.equals("log")) {
        String user = input.getStringByField("user");

        if (_pvMap.containsKey(user))
            _pvMap.put(user, _pvMap.get(user) + 1);
        else
            _pvMap.put(user, 1);
    }

    if (streamId.equals("stop")) {
        Iterator<Entry<String, Integer>> it = _pvMap.entrySet().iterator();

        while(it.hasNext()) {
            Entry<String, Integer> entry = it.next();
            _collector.emit(new Values(entry.getKey(), entry.getValue()));
        }
    }
}
```

最终的日志结果如下（每个用户只输出一条记录）：

```
2012-12-18 17:26:37 STDIO [INFO] userB:21
2012-12-18 17:26:37 STDIO [INFO] userD:15
2012-12-18 17:26:37 STDIO [INFO] userA:22
2012-12-18 17:26:37 STDIO [INFO] userC:24
2012-12-18 17:26:37 STDIO [INFO] userE:17
```

## 9.2 复杂一点的实例

上面的应用场景比较简单，理解起来相对容易一些。下面我们再来看一个相对复杂的应用场景实例。此场景中，我们模拟一个简化的电子商务网站用来实时计算来源成交效果的系统。

在此实例中，我们有两个实时数据产生源，即 VSpout 和 BSpout。VSpout 组件（Spout）模拟用户浏览电子商务网站的行为，产生访问日志；BSpout 组件（Spout）模拟用户在电子商务网站的成交行为，产生成交日志。然后 LogMerge 组件（Bolt）负责将两个源头的实时数据，以用户为主键，进行关联合并；最后 LogStat 组件（Bolt）进行来源效果统计。在这个场景中，我们省略结果输出（LogWriter）的环节，留给读者自己去发挥。

这一次，我们有了两份日志格式。

### 浏览日志格式 (VSpout)

- time 访问时间
- user 访客
- SRC 访问来源

### 成交日志格式 (BSpout)

- time 访问时间
- user 访客
- pay 成交金额

Topology 的结构如图 9-3 所示。

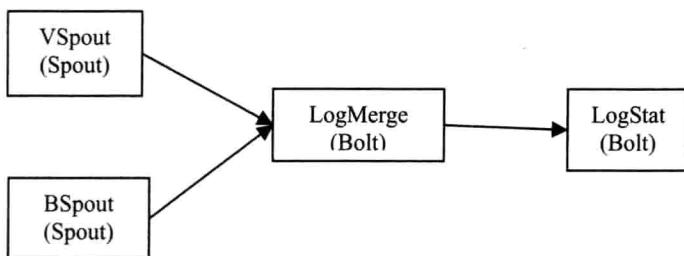


图 9-3

根据图 9-3，我们来定义 Topology 结构。

其代码如下：

```

public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("log-vspout", new VSpout(), 1);
    builder.setSpout("log-bspout", new BSpout(), 1);

    builder.setBolt("log-merge", new LogMergeBolt(), 2)
        .fieldsGrouping("log-vspout", "visit", new Fields("user"))
        .fieldsGrouping("log-bspout", "business", new Fields("user"));

    builder.setBolt("log-stat", new LogStatBolt(), 2)
        .fieldsGrouping("log-merge", new Fields("srcid"));

    Config conf = new Config();

    // 实时计算不需要可靠消息，故关闭acker节省通信资源
    conf.setNumAckers(0);
    // 设置独立Java进程数，一般设为同spout和bolt的总tasks数量相等或更多，
    // 使每个task都运行在独立的Java进程中,
    // 以避免多task集中在同一个jvm里运行产生GC瓶颈
    conf.setNumWorkers(7);
    StormSubmitter.submitTopology(
        args[0],
        conf,
        builder.createTopology()
    );
}

```

上面的代码，读者应该已经很熟悉了。我们继续先初始化 TopologyBuilder 实例，利用该实例配置两个 Spout(VSpout 和 BSpout)，两个 Bolt (LogMergeBolt 和 LogStatBolt)。再实例化 Topology 配置信息，在这里我们使用默认配置。最后利用 StormSubmitter 的静态方法将 Topology 提交到 Storm 集群。

这次我们设置了 VSpout 和 BSpout，组件 ID 分别为 "log-vspout" 和 "log-bspout"，LogMergeBolt 组件可以通过这两个 ID 来订阅其输出结果。我们又对这两个 Spout 组件定义了两个流名称，分别为 "visit" 和 "business"，用来代表流量日志和业务成交日志。LogMergeBolt 和 LogStatBolt 采用 fieldsGrouping 的分组方式按照字段 "user" 进行订阅。这样就构成了来源成交效果系统的 Topology。我们设置了 "conf.setNumAckers(0)"，即关闭 acker 节省通信资源，因为在这个场景中，我们认为不需要确认每个消息都被处理了。我们设置了 "conf.setNumWorkers(7)"，即设置了 Java 独立进程数，充分利用服务的资源来提高系统的效率。

我们再来看一下 Topology 中每个组件的具体实现和作用。

### 9.2.1 VSpout

VSpout 实现了模拟用户访问数据，输出用户的访问时间、来源标识，以及用户名称。正式生产环境中，可能情况会比较复杂，我们这里做了很多简化。

主要代码如下：

```
public class VSpoout extends BaseRichSpout {
    private SpoutOutputCollector _collector;
    private String[] _users = {
        "userA",
        "userB",
        "userC",
        "userD",
        "userE"
    };
    private String[] _srcid = {"s1", "s2", "s3", "s1", "s1"};
    private int count = 5;

    @Override
    public void open(Map conf, TopologyContext context,
                     SpoutOutputCollector collector) {
        _collector = collector;
    }

    @Override
    public void nextTuple() {
        for(int i=0; i<count; i++) {
            try {
                Thread.sleep(1000);
                _collector.emit(
                    "visit",
                    new Values(
                        System.currentTimeMillis(),
                        _users[i],
                        _srcid[i]
                    )
                );
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream (
            "visit",
            new Fields("time", "user", "srcid")
        );
    }
}
```

上面代码中，我们在 `nextTuple()` 方法中模拟输出了 5 条用户访问的名称、来源，以及时间信息。细心的读者可以发现，在发送流量日志前，先暂停了 1000 毫秒。暂停的原因我们在 `BSpout` 的实现中做解答。我们又在 `declareOutputFields` 方法中，定义了三个输出字段：“time”、“user”和“srcid”，传送给下游的 `LogMergeBolt`。

### 9.2.2 BSpout

`BSpout` 实现了模拟用户购买行为数据，输出用户的成交时间、成交金额，以及用户名称。正式生产环境中，情况会比较复杂，我们这里同样做了很多简化。

主要代码如下：

```
public class BSpout extends BaseRichSpout {
    private SpoutOutputCollector _collector;
    private String[] _users = {
        "userA",
        "userB",
        "userC",
        "userD",
        "userE"
    };
    private String[] _pays = {"100", "200", "300", "400", "200"};
    private int count = 5;

    @Override
    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
    }
}
```

```
    @Override
    public void nextTuple() {
        for(int i=0; i<count; i++) {
            try {
                Thread.sleep(1500);
                _collector.emit(
                    "business",
                    new Values(
                        System.currentTimeMillis(),
                        _users[i],
                        _pays[i]
                    )
                );
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream(
            "business",
            new Fields("time", "user", "pay")
        );
    }
}
```

上面的代码中，我们在 `nextTuple()` 方法中模拟输出了 5 条用户成交信息，包括名称、金额和时间信息。当然，在实际生产环境中，并不是每个用户的访问行为都会产生交易，这里为了简化场景，我们假设这 5 个用户都只有一次成交。在发送流量日志前，这里先暂停了 1500 毫秒。这样做就可以使流量日志先于业务成交日志到达下游的 `LogMergeBolt` 组件。只有这样做，我们才能保证成交日志总能关联到流量日

志。(在实际生产环境中，可能会遇到多种情况，需要进行多种处理，这里为了读者容易理解，我们将情况进行了进一步简化。)

### 9.2.3 LogMergeBolt

LogMergeBolt 实现了将成交数据关联到用户的流量来源数据上，并将关联后的数据发送给下游，方便进行来源成交效果的统计。

主要代码如下：

```
public class LogMergeBolt extends BaseRichBolt {
    private transient OutputCollector _collector;

    // 暂时存储用户的来源记录
    private HashMap<String, String> srcmap;

    @Override
    public void prepare(Map stormConf, TopologyContext context,
                        OutputCollector collector) {
        _collector = collector;

        if (srcmap == null) {
            srcmap = new HashMap<String, String>();
        }
    }
```

```

@Override
public void execute(Tuple input) {
    String streamID = input.getSourceStreamId();

    if (streamID.equals("visit")) {
        String user = input.getStringByField("user");
        String srcid = input.getStringByField("srcid");
        srcmap.put(user, srcid);
    }
    else if (streamID.equals("business")) {
        String user = input.getStringByField("user");
        String pay = input.getStringByField("pay");
        String srcid = srcmap.get(user);

        if (srcid != null) {
            _collector.emit(new Values(user, pay, srcid));
            srcmap.remove(user);
        }
        else {
            // 一般只有成交日志快于流量日志时才会发生。
        }
    }
}

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

```

从上面的 execute 方法代码中可以看到，我们按照 Spout 的流名称来判断数据源，如果是用户的流量日志，方法会将用户的名称和来源暂时保存到 Hashmap 中；如果是成交日志，方法会根据成交日志的用户名称，在 Hashmap 中关联到用户的来源信息。

由于上面的一些特殊控制操作，这里成交日志总会从 Hashmap 中关联到用户的来源信息。

最后，将用户的成交信息和来源信息合并为一条信息发送给下游进行统计分析，将发送出去的用户信息从 Hashmap 中移除。

#### 9.2.4 LogStatBolt

LogStatBolt 实现了对用户的成交数据按照来源维度进行统计，这样便可以观察哪个来源的成交金额最高，作为评估来源渠道质量的一个指标。(本例没有将统计结果输出或者进行持久化，读者可以自由发挥。)

主要代码如下：

```
public class LogStatBolt extends BaseRichBolt{
    private transient OutputCollector _collector;
    private HashMap<String, Long> srcpay; // 暂时存储用户的来源记录

    @Override
    public void prepare(Map stormConf, TopologyContext context,
                        OutputCollector collector) {
        _collector = collector;

        if (srcpay == null) {
            srcpay = new HashMap<String, Long>();
        }
    }

    @Override
    public void execute(Tuple input) {
        String pay = input.getStringByField("pay");
        String srcid = input.getStringByField("srcid");

        if(srcpay.containsKey(srcid)) {
            srcpay.put(srcid, Long.parseLong(pay) + srcpay.get(srcid));
        } else {
            srcpay.put(srcid, Long.parseLong(pay));
        }
    }

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

这个组件对于上游发送的数据，按照来源“srcid”分组，保证同一个来源下的成交记录发送到同一个任务中。将每个任务进行统计，并将结果保存在 srcpay 中。

至此第二个应用场景就介绍完毕了，最后补充说明一下，实际生产环境中，计算成交来源效果是一个比较复杂的问题，

计算的方式和衡量的方法有多种。上面的应用场景，只是将其简化为一个简单的模式进行实时计算，作为向读者介绍 Storm 实时计算的一些常用案例。

### 9.3 其他

Storm 作为实时流处理框架，被越来越多地应用到各种业务场景中。作为实时流数据处理的利器，Storm 通常会与其他系统一起运用，以便发挥更大的作用。

比如，目前有些公司正打算将 Storm 整合到 Hadoop 上。当然，这两个技术具备整合可能性极大程度该归于 YARN 这个集群管理层。

还有一些与 Storm 交互的系统，如缓存 Redis、数据库 Hbase 等这类高性能的系统，能将计算的结果缓存到 Redis 或者持久化到 Hbase 系统中，与 Storm 解耦，为后续的高并发查询提供服务，也不失为一个选择。

# 第10章

## 常见应用问 题分析

在日常的 Storm 项目中，遇到各种问题是在所难免的。本章主要借用几个实际的场景，向读者展示如何在项目中定位和排查问题。例子总是特定的，而问题是广泛存在的，这里想向大家展示的主要是一些通用的方法与思想，而不是一些特例或者工具，希望读者可以灵活掌握与运用。

## 10.1 性能问题排查与定位

由于项目需要，我们将平台上的 Storm 升级到当时较新的版本 0.7.2。升级后，Storm 的吞吐量从 10000 tuples/s 下降到约 1000 tuples/s。我们就以这个场景来描述 Storm 体系中问题的定位和排查方法。

### 1. 观察现象

准确地定位问题是解决问题的第一步，也是最关键的第一步。为了准确地定位问题，需要仔细观察，改变程序的运行条件并试跑，不放过任何蛛丝马迹，把这些现象记录下来，

综合分析问题所在。

本例中，提交的是由数十个 Worker 构成的拓扑，这些拓扑分布在十台机器上。首先将这个大拓扑简化成只有一个 Spout、一个 Bolt 的简单拓扑，经过测试，简化后的拓扑吞吐量还是只有 1000 tuples/s，说明这个简化没有掩盖问题。

另外，笔者还观察到在 16 核的 CPU 上，有些 CPU 的使用率为 100%。由于系统中进程较多，不确定是哪些进程占用了 CPU。

## 2. 问题定位

经过排查，服务器的内存使用、网络带宽、磁盘 IO 都没有遇到瓶颈，虽然有些 CPU 核的使用率比较高，但是大部分其他 CPU 核都处于空闲状态。

在 Storm 系统及我们的拓扑中，并没有大规模的计算型任务，因此这个非常高的单核 CPU 占用很值得关注。首先，我们要找出这个线程。

可以通过 ps、top 等工具来查看系统中正在运行的进程，

发现 CPU 占用率高的进程是一个 Java 进程，并且是一个 Storm 的 Worker 进程。

### 3. 找到出问题的线程

Storm 的线程模型较复杂，在一个 Worker 里，包含了 tuple 发送、接收、tasks、heartbeat，等等。

可以通过 jstack 命令检查一个 Java 进程中的所有线程及其运行状态。由于我们要找的是一个 CPU 使用率为 100% 的线程，因此仅仅通过运行状态是很难找出目标线程的。

Java 或者是第三方提供了一些工具，如 JProfiler、Java Mission Control 等工具可以直接检测出占用 CPU 比较多的代码分支，从而从源代码找到所属的线程

然而，这些工具都需要通过 JMX 协议连接到目标 Java 虚拟机，这在很多情况下是不能实现的，这里介绍一种比较简单的定位高 CPU 占用率线程的方法。

只要代码运行路径上有 IO 相关的操作，它的 CPU 使用率就不能达到 100%，因为 IO 操作时，CPU 是处于等待状态的。

我们将拓扑代码进一步简化：Spout 读取数据后立即封装成 tuple 发送出去；Bolt 接收到 tuple 之后，即将这个 tuple 抛弃。

因为并非所有的拓扑都吞吐量低，所以首先初判问题出在与用户代码相关的部分。在 spout.nextTuple() 和 bolt.execute() 方法中加入一段打印代码。经过尝试，发现 CPU 密集型的线程正是与 emit() 方法相关的线程。找到这个线程之后，就需要找到相关的代码段，我们采用的办法是统计每段代码的起止运行时间。最终发现 Storm 在对 byte 数据 byte[] 序列化的时候没有进行优化，而是作为数据单字节进行序列化，因此造成了 CPU 占用率较高，处理速度较慢的情况。

#### 4. 解决问题

找到问题之后，解决起来就简单了。既然 Storm 的序列化组件 kyro 在处理 byte[] 时有性能问题，我们就把 byte[] 转换成 String 类型来规避这个问题。

优化后，Storm 单节点的处理速度超过了 10000 tuples/s，并且 CPU 的使用率大幅下降到 20%~30%。

这里跟大家演示的只是在分布式集群中排查问题的一种思路和方法。**byte[]**序列化的问题在 Storm 0.8.1 中已经解决，大家可以放心使用这个数据类型。

### 方法总结如下

在集群中排查问题，其实只有一个原则：化繁为简。尽量将集群中的问题在单机上重现，然后就可以采用常用的各種手段来诊断问题。另外，工具在排查问题中是必不可少的，不过也不用唯工具论，只要适合特定场景的、可以高效率发现问题的手段都是好工具。

## 10.2 系统中常见的问题与排查

### 1. Zookeeper crash 的问题

Zookeeper 是 Storm 系统的核心，一旦 Zookeeper 无法正常工作，整个 Storm 集群就会停下来，因此应当格外留意 Zookeeper。

Zookeeper 是一个相对比较稳定的系统，最常见的引起故障的问题是快照所在的磁盘分区被用完。因为 Zookeeper 每隔一段时间就会将它的快照备份到 data/ 目录下，这些快照很大，在磁盘较小的系统上，不用太久，磁盘就会被耗尽。

处理的办法也很简单，定时删除多余的快照文件即可，可将下面的代码加到定时任务 crontab 中：

```
* */4 * * * * java -cp /opt/demo/install/zookeeper-3.3.4/zookeeper-3.3.4.jar \
:/opt/demo/install/zookeeper-3.3.4/lib/*:/opt/demo/install/zookeeper-3.3.4/conf \
org.apache.zookeeper.server.PurgeTxnLog /tmp/zookeeper/ /tmp/zookeeper/ -n 3 \
>/dev/null 2>&1
```

## 2. Supervisor 不能启动的问题

在拓扑异常终止后，再次启动这个拓扑时，经常遇到下面的问题：

```
2013-08-12 17:32:09 event [ERROR] Error when processing event
java.io.FileNotFoundException: File '/home/demo/storm/workdir/supervisor/stormdist\star-flow-bridge-15-1375411778/stormconf.ser' does not exist
    at org.apache.commons.io.FileUtils.openInputStream (FileUtils.java:137)
    at org.apache.commons.io.FileUtils.readFileToByteArray (FileUtils.java:1135)
    at backtype.storm.config$read_supervisor_storm_conf.invoke (config.clj:138)
    at backtype.storm.daemon.supervisor$fn_4769.invoke (supervisor.clj:404)
    at clojure.lang.MultiFn.invoke (MultiFn.java:177)
    at backtype.storm.daemon.supervisor$sync_processes$iter__4660__4664$fn_4665.\ninvoke(supervisor.clj:247)
    at clojure.lang.LazySeq.sval (LazySeq.java:42)
    at clojure.lang.LazySeq.seq (LazySeq.java:60)
    at clojure.lang.RT.seq (RT.java:473)
    at clojure.core$seq.invoke (core.clj:183)
    at clojure.core$dorun.invoke (core.clj:2725)
    at clojure.core$doall.invoke (core.clj:2741)
    at backtype.storm.daemon.supervisor$sync_processes.invoke (supervisor.clj:235)
    at clojure.lang.AFn.applyToHelper (AFn.java:161)
    at clojure.lang.AFn.applyTo (AFn.java:151)
    at clojure.core$apply.invoke (core.clj:603)
    at clojure.core$partial$fn_4070.invoke (core.clj:2343)
    at clojure.lang.RestFn.invoke (RestFn.java:397)
    at backtype.storm.event$event_manager$fn_2484.invoke (event.clj:24)
    at clojure.lang.AFn.run (AFn.java:24)
    at java.lang.Thread.run (Thread.java:662)
2013-08-12 17:32:09 util [INFO] Halting process: ("Error when processing an event")
```

即提示“stormconf.ser”不存在，从而造成 Supervisor 启动失败。这是任务异常终止时 Zookeeper 上的拓扑状态与本地拓扑状态不一致造成的。

解决办法就是删除拓扑在本机的临时状态，然后重新从 Zookeeper 上读取最新的状态，步骤如下。

(1) 确保此时 Supervisor 没有运行

(2) cd /home/demo/storm/workdir/supervisor

(3) sudo rm -rf localstate/

### 10.3 业务问题的定位与排查

通常，在业务相关的设计中，也要兼顾系统的效率和稳定性。在一个实时数据分析的项目中，有这样一个场景：使用 Storm 系统从上游的数据源连续不断地读入数据，交给运行在 Storm 上的数据处理单元实时分析，然后为用户提供报表数据。

由于 Spout 中读取外部数据源部分采用单线程设计，当数据量较大时，这部分就成为了系统的瓶颈——不能快速地读入数据，引起数据处理延迟。解决办法很简单，增加读取数据的并发数，不过，这里却面临两个选择。

- C (1) 通过拓扑的参数，靠增加 Spout 的数量来增加并发数。
- C (2) Spout 的并发数不变，而是在每个Spout 中增加专门的线程来提高读取数据的并发数量。

粗看起来，这两种方式没有太大区别，都可以达到增加

并发度的目的。其实不然，仔细分析会发现两者还是有很大差别的。

(1) 内存使用。Java 虚拟机本身的内存开销及 Storm 任务中固定使用的内存都是不可忽略的部分，增加 Java 虚拟机数目，显然会增加系统中总内存的消耗。

(2) Storm framework 的管理成本。Storm 系统中使用 heartbeat 来管理和维护各个 Worker 之间的关系。Worker 越多，系统中的心跳数据包就越多，并且会增加 Storm 处理这些数据包的负担。

(3) 扩展性。在并发度较小时并不明显，当需要较大并发的时候（数十个），单纯增加 Worker 的并发数，第 1 条和第 2 条中提到的额外资源消耗会愈加突出。另外，系统中存在过多的 Java 进程，势必造成 CPU 上下文切换开销增大，反而降低系统的处理效率。

经过上面的对比和评估，我们发现，在需要大量并发的时候，还是将这些并发控制逻辑放到单独的线程中比较合适。

更灵活一点的话，可以将这些并发度的控制作为拓扑的配置参数开放出来，从而得到与 Worker 的并发度一样的灵活控制，又有系统资源开销上的优势。

在项目中，我们采用的是第二种方案，采用这种方案，轻易地实现了 64 个并发的控制，而且没有带来系统开销的明显增加。

上面的分析主要是想跟大家分享在业务逻辑设计中如何兼顾系统性能。现实中的问题不一而足，各有各的场景，重要的是我们可以掌握分析问题的方法和思路，可以从业务逻辑和系统等多个角度综合分析，从而快速准备，找到问题的症结、对症下药！