O'REILLY®

Flume

构建高可用、可扩展的海量日志采集系统

Using Flume

[美] Hari Shreedharan 著

马延辉 史东杰 译

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Flume

构建高可用、可扩展的海量日志采集系统

Using Flume

阜阳师范学院图书馆
藏书

[美] Hari Shreedharan 著
马延辉 史东杰 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书从 Flume 的基本概念和设计原理开始讲解，分别介绍了不同种类的组件、如何配置组件、如何运行 Flume Agent 等。同时，分别讨论 Source、Channel 和 Sink 三种核心组件，不仅仅阐述每个组件的基本概念，而且结合实际的编程案例，深入、全面地介绍每个组件的详细用法，并且这部分内容也是整个 Flume 框架的重中之重。之后，讲解拦截器、Channel 选择器、Sink 组和 Sink 处理器等内容，它们为 Flume 提供灵活的扩展支持。最后，介绍了 Flume 的高级使用，如何使用 Flume 软件开发工具集（SDK）和 Embedded Agent API，如何设计、部署和监控 Flume 生产集群。

总而言之，本书是一本理论结合实战，深度、广度兼备的海量日志采集系统的著作。

©2015 by O'Reilly Media, Inc.
Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2015. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由 O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2015-4168

图书在版编目（CIP）数据

Flume：构建高可用、可扩展的海量日志采集系统/（美）史瑞德哈伦（Shreedharan,H.）著；马延辉，史东杰译. —北京：电子工业出版社，2015.8
书名原文：Using Flume
ISBN 978-7-121-26558-7

Ⅰ.①F… Ⅱ.①史… ②马… ③史… Ⅲ.①数据采集 Ⅳ.①TP274

中国版本图书馆CIP数据核字(2015)第151458号

策划编辑：张春雨
责任编辑：张春雨
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
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：787×980　1/16　印张：14.25　字数：319.2千字
版　　次：2015年8月第1版
印　　次：2015年8月第1次印刷
定　　价：69.00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。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”;创建第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了Make杂志,从而成为DIY革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖,共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程,每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人,他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了:‘如果你在路上遇到岔路口,走小路(岔路)。’回顾过去Tim似乎每一次都选择了小路,而且有几次都是一闪即逝的机会,尽管大路也不错。”

——Linux Journal

译者序

几乎任何规模的公司，每时每刻都在产生大量的数据，收集业务日志数据，供离线和在线的分析系统使用。处理这些日志需要特定的日志系统，一般而言，这些系统需要具有高可用性、高可靠性和可扩展性。

Flume是一个分布式、可靠的和高可用的海量日志采集、聚合和传输的系统。支持在系统中定制各类数据发送方，用于收集数据；同时，Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。Flume初始的发行版本目前被统称为Flume OG（original generation），属于Cloudera。但随着Flume功能的扩展，Flume OG代码工程臃肿、核心组件设计不合理、核心配置不标准等缺点暴露出来，尤其是在Flume OG的最后一个发行版本0.94.0中，日志传输不稳定的现象尤为严重。为了解决这些问题，2011年10月22日，Cloudera完成了Flume-728，对Flume进行了里程碑式的改动：重构核心组件、核心配置以及代码架构，重构后的版本统称为Flume NG（next generation）；改动的另一原因是将Flume纳入Apache旗下，Cloudera Flume改名为Apache Flume。

本书针对具有Java编程语言基础的开发者，需要已经熟悉普通Java开发实践。如果掌握了Java语言基础，那么对于理解Flume原理和开发Flume将更加得心应手。如果打算学习使用Flume收集分布式的海量日志，那么本书正好适合你。

Flume以Agent为最小的独立运行单位。一个Agent就是一个JVM。单Agent由Source、Sink和Channel三大组件构成。本书从Flume的三大核心组件通过配置实例和原理进行讲解，因此学习了本书后，你将能够直接使用Flume进行各种类型日志的收集并保存到Hadoop集群或者HBase集群。

- 认识Apache Hadoop和Apache HBase。本章简单介绍了HDFS和HBase的基本概念。
- 用Apache Flume处理流数据。本章介绍为什么需要一个类似Flume的系统、它的

设计和实现，以及使得 Flume 高扩展、灵活和可靠的不同特征。

- 源（Source）。本章我们将讨论 Flume 中自带的多种 Source 的设计和工作原理，以及如何才能最优化配置；我们也将介绍如何去编写自定义的 Source。
- Channel。Channel 是位于 Source 和 Sink 之间的缓冲区，保存通过 Source 传入到 Flume Agent 的数据，直到它被 Sink 移除。
- Sink。本章我们将讨论不同种类的 Sink，它们的配置和管理，如何针对每个 Sink 序列化数据以使得数据可以写为用户选择的一种格式，以及如何编写自定义 Sink。
- 拦截器、Channel 选择器、Sink 组和 Sink 处理器。Flume Agent 最重要的组件是 Source、Channel 和 Sink。另外，Flume Agent 还有一些使 Flume 更加灵活的组件。本章将讨论拦截器、Channel 选择器、Sink 组和 Sink 处理器。
- 发送数据到 Flume。本章我们将讨论数据从客户端应用程序发送到一个或多个 Flume Agent 的不同方法。Flume 有两类发送数据到 Flume Agent 的程序性方法：Flume SDK 和 Embedded Agent API。Flume 也自带有 log4j appender，可以用来从应用程序发送数据到 Flume Agent。
- 规划、部署和监控 Flume。在这一章，我们将讨论如何规划去部署 Flume，以及如何部署和监控 Flume Agent。我们还将讨论 Flume 项目本身之外的各种可用工具，这些工具会使 Flume 的部署和监控更加简易。

由于本书涉及的许多术语目前尚无固定译法，我们在翻译过程中力求选择一个简洁达意的译法，但限于自身水平有限，必定仍会有诸多不足，我们真诚地希望同行和读者们不吝赐教，我们将不胜感激。

——史东杰

序

过去的几年中，大数据技术的发展和应用有了巨大的增长。Hadoop 和相关平台推动着基于不断增长的巨大数据量的下一波数据分析浪潮。今天产生的数据将用来预测明天发生的事情，伴随着数字革命以不断加快的速度席卷我们生活的所有方面。要跨过这个海量数据新时代的门槛，明显需要注意的一点是：开始处理如何获取所有的数据到你的集群中的数据。显然，这些数据产生的来源，广泛蔓延到整个企业，并且与机器、传感器和社会数据等产生了有趣的互动效应。任何处理类似挑战的操作员毫无疑问会同意，这点是很重要的——如果不是很困难，要建立一个系统，以符合成本效益的方式路由数据到你的集群。

Apache Flume 正是用来应对这个挑战的。

早在 2011 年，当 Flume 进入 Apache 软件基金会孵化状态时，它就是 Cloudera 工程师用来在 Hadoop 上解决大规模日志数据聚合的一个项目。作为一开始就受欢迎的项目，它看到了大量的新需求，从事件排序到语义的可靠传递，并在其初始发行版就提出来了。鉴于它的受欢迎程度和人们对复杂要求的高需求，我们决定重构整个项目，让它在适用性和可管理性方面更简单、更强大，并在必要时允许简易扩展。在这个孵化器项目中，Hari 和我，连同其他几个工程师，都是昼夜不停地工作，与 Flume 社区共同推动这个愿景，使其早日实现。从那时到现在，Flume 已经成型并将自己推入顶级 Apache 项目，推出了多次稳定版本，从而大大丰富了其功能。

如今，Flume 在世界各地的大量数据中心被积极部署和使用，有时跨度超越洲际。它继续有效地提供高弹性、容错、可靠、快速和高效的机制，将大量数据从各种各样的数据源移动到目的地系统，如 HBase、HDFS 等。一个精心设计的 Flume 拓扑，可以在最少或没有干预的前提下运转，实际上它能无限期地自运行。它提供了上下文的路由，当连接恢复时，通过提供可靠存储和重新发送消息的能力，能够处理宕机、网络中断等不可

预知或计划外的中断。这些都是现成可用的，且通过使用已广泛采用的相当稳定和直观的接口，提供了自定义任何组件的灵活性。

在本书中，Hari 概述了 Flume 的各种组件，并在必要时深入细节。操作员将会发现，这本书对于如何轻松设置和部署 Flume 管道的理解是非常有价值的。开发人员将找到一个方便的参考，以在 Flume 内构建或自定义组件，并更好地理解其架构和组件的设计。最重要的是，这本书在你设置连续 HDFS 和 HBase 的输入时，会给出必要的见解——HDFS 和 HBase 是如今最受欢迎的两个存储系统。

通过部署 Flume，可以确保数据——无论数据来自什么企业，或是多大量级——都将安全、及时地到达你的大数据平台。然后你就可以将精力集中在如何洞悉数据上。祝大家好运！

——Arvind Prabhakar, CTO, StreamSets

前言

如今，开发人员能够很容易编写和在“云”中大量服务器上部署应用程序。这些应用程序比以往产生更多的数据，对其存储和分析能提供有价值的见解，可以改善应用程序本身和将应用程序作为自身一部分的企业。这类应用程序生成的数据通常使用 Hadoop 和 HBase 等系统来分析。

只有当你可以从前端服务器获取数据到上述这些系统，分析这些数据才是可能的。通常，随着数据产生的时间变长，这种分析的有效性就变得不那么靠得住了。要想近乎实时地获取到数据，就会用到诸如 Apache Flume 这样的系统。Apache Flume 是一个系统，用于从大量数据生产商那里移动大量级数据到存储、索引或分析数据的系统。这种系统也可以将数据的消费者和生产者解耦，使得在一方不知情的情况下，改变另一方，变得很容易。除了解耦，它们还提供故障隔离，并在生产商和存储系统之间添加一个额外的缓冲区。数据生产商将不知道存储或索引系统无法访问，直到所有的 Flume 缓冲区也被填满——这提供了一个额外的缓冲区，可能足以确保存储系统完成重新上线和清理 Flume 缓冲区中积压的事件。

在本书中，我们将详细讨论为什么需要诸如 Flume 这样的系统、Flume Agent 的内部构造，以及如何配置和部署 Flume Agent。我们还将讨论 Flume 自定义部署的各种方式，以及如何编写 Flume 的插件。

第 1 章给出了 Apache Hadoop、Apache HBase 的基本介绍。本章只是为了向读者介绍 Hadoop、HBase 和它们内部的一些细节。如果读者已经熟悉 Hadoop 和 HBase，这章可以跳过。

第 2 章介绍了 Flume 主要的组件和配置，也解释了如何部署 Flume，以从数据生成服务器推送数据到存储和索引系统。

第 3 章、第 4 章、第 5 章和第 6 章解释了 Flume 内置的不同种类的 Source、Channel 和 Sink，以及编写自定义插件以定制 Flume 接收、修改、格式化和写数据的方式。

第 7 章，我们讨论从你的应用程序发送数据到 Flume Agent 的不同方式。本章主要是为了写推送数据到 Flume Agent 的应用程序的开发人员。

我们还将在第 8 章讨论如何设计、部署和监控 Flume 部署。

本书使用的约定

本书中使用的约定如下：

斜体字

表明新术语、网址、电子邮件地址、文件名和文件扩展名。

`等宽字体`

用于程序清单，以及在段落引用程序元素，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

`等宽加粗字体`

显示用户应该输入的命令或其他文本。

`等宽斜体`

显示用户提供的应该替换或取决于上下文的文本。

配置清单中的粗体

表示配置参数列表中的必需参数。

编写自定义代码*

某些章和节的名称以星号（*）结束。这表明它们讨论 Flume 组件的设计概念相关的开发主题，或解释如何编写可以插入 Flume 的自定义组件。

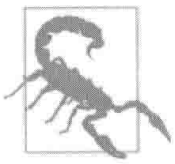
这个元素表示提示或建议。

这个元素表示一般的注意。

这个元素表示警告或注意。

使用代码示例

可以在 *https://github.com/harishreedharan/usingflumecode* 下载可用的补充材料（代码示例、练习题等）。



本书中的代码

本书中所示的代码已经被编辑得尽可能清晰，可能这样的代码不能编译或运行。完整工作版本的代码可以在上面所示的 GitHub 库中找到。请注意，随书代码是为了说明 Flume 的关键概念，可能没有处理所有边缘或错误情况。

本书是帮助你完成工作的。一般来说，如果本书提供了示例代码，你可以在你的程序和文档使用它。你不需要联系我们来获得许可，除非你复制代码的很大一部分。例如，编写一个程序，使用本书几块代码不需要许可。出售或分发 O'Reilly 书籍示例的 CD-ROM 是需要许可的。引用本书回答问题和引用示例代码不需要许可。从本书合并大量的示例代码到你的产品文档需要许可。

我们赞赏，但不要求标明信息出处。信息出处通常包括标题、作者、出版商和 ISBN。例如："*Using Flume* by Hari Shreedharan (O'Reilly)。版权 2015 Hari Shreedharan，978-1-449-36830-2。"

如果你觉得你对示例代码的使用超出合理使用或上面给出的许可，随时与我们联系：*permissions@oreilly.com*。

Safari® Books Online

Safari Books Online 是一个随需应变的数字图书馆，提供技术商业领域世界领先作者的书和视频形式的专业内容。

专业技术人员、软件开发人员、网页设计师和商业创意专业人士使用 Safari Books Online，作为他们研究、解决问题、学习和认证培训的主要资源。

Safari Books Online 提供了一系列为企业、政府、教育和个人的计划和定价。

会员可以访问成千上万的书籍、培训视频和一个出版商正式出版前的手稿的完全可搜索的数据库，这些出版社包括 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press，John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，以及更多出版社。关于 Safari Books Online 的更多信息，请访问我们的主页。

如何联系我们

请将关于本书的评论和问题反馈给出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

本书有一个 web 页面，我们列出勘误表、示例和任何其他的信息。你可以访问这个页面 *http://bit.ly/using-flume*。

要评论本书或询问技术问题，请发送邮件到 *bookquestions@oreilly.com*。

想了解更多关于我们书籍、课程、会议和新闻的信息，请访问我们的网址 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看我们：*http://www.youtube.com/oreillymedia*

致谢

我满怀感激地把这本书献给我的父母，Dr. M. Shreedharan 和 Usha Shreedharan，没有他们的努力和鼓励，我不能够成为一名工程师并最终写出这本书。

我也感谢我的姐姐 Lakshmi Shreedharan 和姐夫 Dinakar Kesavapillai，感谢在我大学和研究生期间他们的所有帮助和支持。我还想谢谢我的侄女 Dhwani，感谢她为我的生活增加的所有乐趣。

我由衷感谢我的妻子，Archana Sastry，感谢她的支持，感谢她在 Flume 发展早期时的每个深夜的陪伴。

感谢 Arvind Prabhakar，作为导师和朋友，他的指导一直帮助我选择正确的道路。

我也感谢 Mike Percy、Brock Noland、Jarcec Cecho、Kathleen Ting、Jeff Lord、Prasad Mujumdar、Will McQueen 和 Ken Choy，是他们帮助我成长为一名工程师和一个真正的人。

我也借此机会感谢 Jon Hsieh 开启 Flume 项目，感谢 Apache Flume 社区的建立和维护。

目录

译者序 v

序 xiii

前言 x

第 1 章　认识 Apache Hadoop 和 Apache HBase 1

分布式文件系统 HDFS 1

HDFS 的数据格式 3

处理 HDFS 中的数据 4

Apache HBase 4

总结 5

参考文献 6

第 2 章　用 Apache Flume 处理流数据 7

我们需要 Flume 7

Flume 是否适合呢？ 9

Flume Agent 内部原理 10

配置 Flume Agent 13

Flume Agent 之间的相互通信 17

复杂的流 17
复制数据到不同目的地 20
动态路由 21
Flume 的无数据丢失保证，Channel 和事务 22
Flume Channel 中的事务 23
Agent 失败和数据丢失 25
批量的重要性 26
重复怎么样？ 27
运行 Flume Agent 27
总结 29
参考文献 30

第 3 章 源（Source） 31

Source 的生命周期 31
Sink-to-Source 通信 33
Avro Source 34
Thrift Source 37
RPC Sources 的失败处理 39
HTTP Source 40
针对 HTTP Source 写处理程序 * 42
Spooling Directory Source 47
使用 Deserializers 读取自定义格式 * 50
Spooling Directory Source 性能 55
Syslog Source 55
Exec Source 59
JMS Source 61
转换 JMS 消息为 Flume 事件 * 63
编写自定义 Source* 65
Event-Driven Source 和 Pollable Source 66
总结 73
参考文献 73

第 4 章　Channel 75
事务工作流 76
Flume 自带的 Channel 78
Memory Channel 78
File Channel 80
总结 86
参考文献 86
第 5 章　Sink 87
Sink 的生命周期 88
优化 Sink 的性能 89
写入到 HDFS：HDFS Sink 89
理解 Bucket 90
配置 HDFS Sink 93
使用序列化器控制数据格式 * 100
HBase Sink 106
用序列化器将 Flume 事件转换成 HBase Put 和 Increment* 108
RPC Sink 113
Avro Sink 113
Thrift Sink 115
Morphline Solr Sink 116
Elastic Search Sink 119
自定义数据格式 * 121
其他 Sink：Null Sink、Rolling File Sink 和 Logger Sink 124
编写自定义 Sink* 125
总结 129
参考文献 129
第 6 章　拦截器、Channel 选择器、Sink 组和 Sink 处理器 131
拦截器 131
时间戳拦截器 132

主机拦截器 133
静态拦截器 133
正则过滤拦截器 134
Morphline 拦截器 135
UUID 拦截器 136
编写拦截器 * 137
Channel 选择器 140
复制 Channel 选择器 140
多路复用 Channel 选择器 141
自定义 Channel 选择器 * 144
Sink 组和 Sink 处理器 146
Load-Balancing Sink 处理器 148
Failover Sink 处理器 151
总结 153
参考文献 154

第 7 章　发送数据到 Flume* 155

构建 Flume 事件 155
Flume 客户端 SDK 156
创建 Flume RPC 客户端 157
RPC 客户端接口 157
所有 RPC 客户端的公共配置参数 158
默认 RPC 客户端 165
Load-Balancing RPC 客户端 168
Failover RPC 客户端 171
Thrift RPC 客户端 172
嵌入式 Agent 173
配置嵌入式 Agent 175
log4j Appender 180
Load-Balancing log4j Appender 181
总结 182
参考文献 183

第 8 章　规划、部署和监控 Flume 185
规划一个 Flume 部署 185
修复时间 185
我的 Flume Channel 需要多少容量？ 186
多少层？ 186
通过跨数据中心链接发送数据 188
层分片 190
部署 Flume 191
部署自定义代码 191
监控 Flume 193
从自定义组件报告度量 196
总结 196
参考文献 196

索引 197

第 1 章

认识Apache Hadoop和Apache HBase

Apache Hadoop 是一个高度可扩展的、容错的分布式系统，这意味着它可以存储海量数据，并且可以恰当地处理这些数据。Hadoop 的设计理念是存储数据的同一个集群上，运行大规模处理系统。Hadoop 的设计哲学是将所有的数据存储在一个地方，并且在同一个地方处理数据——也就是，将处理移动到数据存储，而不是将数据移动到处理系统。Apache HBase 是构建在 Hadoop 上的，提供键值存储的数据库系统，其得益于 Hadoop 提供的分布式框架。

数据一旦写入 Hadoop 分布式文件系统（HDFS）就不可改变。HDFS 中的文件只支持追加的方式。一旦一个文件被创建并写入数据，该文件可以被追加或删除，但是不能改变该文件中的数据。尽管 HBase 运行在 HDFS 上，但是 HBase 支持对任何写入数据的更新操作，非常类似传统的数据库系统。

本章将会简单介绍 Apache Hadoop 和 Apache HBase，但不会阐述很多细节的内容。

分布式文件系统 HDFS

Hadoop 的核心是分布式文件系统 HDFS。HDFS 是一个高度分布式的、容错的文件系统。它被专门设计用于在商用硬件上运行，通过简单添加更多的硬件来扩展数据存储。可以配置 HDFS 复制多份数据到不同的机器，以确保即使保存数据的机器出现异常情况，数据也不会丢失。复制数据使得系统具备高可用性，即使拥有副本的机器出现网络断开或者宕机也不会受到影响。本节将简要介绍 HDFS 的设计以及运行在 HDFS 上的各种处理系统。

2 HDFS 最初的设计是基于谷歌文件系统 [gfs]。HDFS 是一个分布式系统，可以使用上千种现成的商用服务器存储数据，对硬件配置没有特殊要求。这意味着 HDFS 不需要使用存储区域网络（SANs）、昂贵的网络配置或任何特殊的磁盘。HDFS 可以运行在任何安装好的普通数据中心上。HDFS 基于用户配置的复制因子，复制所有写入数据。默认复制因子值为 3，这确保了任何写入 HDFS 的数据会复制到集群内的三个不同的服务器上。这大大降低了写入 HDFS 时，发生数据丢失的可能性。

HDFS 与其他文件系统一样，将数据写入独立的块中。每个 HDFS 上的文件至少包含一个块。根据文件的大小，每个文件会包含多个块。HDFS 设计初衷是存储大文件。因此，HDFS 块大小也比其他文件系统大很多。HDFS 块大小是可配置的，并且在大多数情况下，是 128 MB 到 512 MB 的范围内。HDFS 会根据复制因子对每个块进行复制，从而保证将文件本身复制为复制因子倍数的文件数量。HDFS 是机架感知的，默认的块放置策略将确保块的每个副本位于不同的机架上。

HDFS 有两种服务器：NameNode 和 DataNode。大多数 Hadoop 集群通常包含两个 NameNode 和多个 DataNode。DataNode 用于存储数据。在任意时刻，只能有一个活跃 NameNode 和一个可选的备用 NameNode。活动 NameNode 是服务客户端和其他数据节点提供服务的 NameNode。备用 NameNode 是活动 NameNode 的有效备份，如果活动 NameNode 出现故障或由于某种原因不能访问时，负责接管活动 NameNode 职责。NameNode 负责存储文件系统中文件和块的元数据。NameNode 将每个文件映射到文件所在的块列表。NameNode 还包含块的位置信息——块存储的 DataNode 以及在 DataNode 上的存储位置。

客户端的写操作最初是写到客户机上的本地文件，直到客户端刷新文件、关闭文件或临时文件的大小超过一个块的边界，创建该文件（或者一旦块边界出现交叉时写入新数据需要添加新块，或者现有文件被重新打开用于追加），并且 NameNode 将块分配给该文件。然后，将数据写入到每个块，并依次复制到多台 DataNode 节点上。仅当所有数据节点（与复制因子值相同的数据节点数量）成功地复制该块，写入操作才算成功。

3 HDFS 的文件不能编辑，只能追加。文件一旦关闭，只能追加数据。HDFS 也不能保证写上文件时对其他客户端是可见的，直到写数据的客户端将数据刷新到 DataNode 的内存中，或关闭该文件时，对其他客户端才是可见的。每次新块被请求，NameNode 分配新块给文件，并跟踪块的状态。对于每个读操作，客户端从 NameNode 获取代表文件的块的位置信息，并直接从 DataNode 读取数据。但从用户的角度看，HDFS 是一个单纯的存储系统，事实上，每个文件复制并存储在多个系统上，这点对用户是完全透明的。因此，用户编写代码过程中，不必关心 HDFS 容错或复制方面的事情。即使客户端 API 刷新或关闭调用之前，向本地计算机上写文件的过程对用户也是透明的，不需要在编码中体现

这些操作。

客户端 API 是用户与 HDFS 交互的一种方式。HDFS 还提供了一组可用于操作一般文件的 Shell 命令。HDFS 命令的格式为：

```
hdfs dfs -<command> <options>
```

例如，获取 HDFS 上 /Data/ 目录下的文件列表，可以使用下面的命令：

```
hdfs dfs -ls /Data
```

支持的命令列表可以在 Hadoop 文档 [commands] 中找到。但是，需要在系统环境中正确配置 *HADOOP_HOME* 或者 *HADOOP_PREFIX* 参数，还要在 Hadoop 配置文件中正确配置 *HADOOP_CLASSPATH* 参数，才能运行这些命令。有关 HDFS 架构和配置的详细信息，请参考 Hadoop 权威指南 [hdfs-architecture]。

HDFS 的数据格式

一般情况下，HDFS 的数据格式可以划分为两种：可拆分的和不可拆分的。可拆分格式是指在一个文件可以按照记录的边界拆分为多个切片。可拆分格式可以从任何地方找到记录的开始位置。可拆分格式是对 MapReduce 是友好的，因为 MapReduce 也是切分文件，使用不同的 Mapper 并行从文件读取数据。

写数据到 HDFS 时，使用二进制格式是比纯文本更好的选择。这是因为大多数二进制格式都有一些方法指明记录中损坏或不完整等问题。故障可能导致写入文件时产生不完整记录。例如，如果 HDFS 集群空间已用完或有连接问题：可能是块分配失败导致文件包含不完整或损坏的记录。二进制记录有助于检测这些不正确的记录并忽略它们。例如，Hadoop 中常用的二进制格式是 Avro。这种格式是可拆分的，并且可以检测文件中损坏或不完整的记录。MapReduce、Hive 和 Pig 等都支持 Avro 作为输入格式。Avro 同时也支持使用 *Snappy* 和 *Deflate/BZ2* 压缩编码。

4

HDFS 有很多数据格式。其中最常见的一种数据格式是顺序文件（Sequence file）。顺序文件是可拆分的文件格式，通常用在 MapReduce 作业中。顺序文件是一些键和值的列表，每个键或值都是 *Writable* 的实例，Writable 是一个可序列化的类。

很多压缩格式是可拆分的，像 bz2、预处理 LZO 等。关于 Hadoop 生态系统中文件格式的细节描述，请参见 Hadoop 权威指南 [serialization]。

Flume 支持多种可用的内置数据格式，并且还允许用户插入自己的序列化器，序列化器可以将用户自选的任意格式的数据写入到 HDFS。我们将在第 5 章的“使用序列化器控

制数据格式 *”一节讨论，这里不再赘述。

处理 HDFS 中的数据

正如我们讨论的，Hadoop 提供了用于数据存储的处理系统。因此，存储数据的相同节点也能处理存储在 HDFS 中的数据。MapReduce 一直是处理 HDFS 中数据的经典系统。

MapReduce 是一个分布式处理框架，允许用户编写 Java 代码从 HDFS 读取并处理数据。每个 MapReduce 程序运行在多个节点上，每个节点处理输入数据的一部分，MapReduce 程序有两个阶段：Map 阶段和 Reduce 阶段。每个阶段在多个节点同时运行一段 Java 代码，从而并行处理大量数据。每个 Mapper 从一个在 HDFS 中的指定文件夹读取一个输入分片（一个输入数据的固定子集）并处理为 Key 和相对应的 Value 格式。

目录中文件的数据如何映射为 Key 和 Value 主要取决于解析数据的 input format[input-format]。Map 阶段解析输入数据并生成中间数据的键值对。所有具有相同 key 值的键值对数据交给相同的 Reducer 进行处理。然后 Reducer 输出最终的键值对数据到配置的输出目录中。更多关于在 Hadoop 中 MapReduce 的资料可以参考：Hadoop 权威指南 [mr]。

Apache Hive 和 Cloudera Impala 提供 SQL 接口（实际是 SQL 子集）处理 HDFS 数据。Hive 将 SQL 请求解析为一个 MapReduce 作业处理数据，而 Impala 有自己的处理引擎，用来读取数据并将基于查询的转换应用于处理数据上。这些系统将 HDFS 中的文件映射到运行查询的表中。如果用户使用基于 SQL 的数据库系统处理和存储数据，那么这些系统给用户提供了一个简单的迁移方法。还有一些其他系统，例如 Apache Pig、Apache Spark 等，用于分析和存储在 HDFS 上的数据。

Apache HBase

Apache HBase 是 Hadoop 生态系统中的键值对存储。经常用于实时写入和更新数据。也可以代替传统的数据库用于提供实时服务。它建立在 HDFS 上并且依赖于 HDFS 的副本机制。从逻辑上来讲，HBase 数据模型类似于数据写入到有若干行和列的表的数据库，它的列不是固定模式，而是可以由客户端动态创建生成（每个行键可以有一个不同列的集合而且没有一个表示固定列集合的固定模式）。

每一行用一个被称为行键键来访问，这和标准数据库系统中的主键非常相似。根据需要一个行键可以有多个列，但是每行中的每一个列只能有一个对应的值（不过 HBase 可以保留多个“版本”——该行对应列的最后 *n* 个值）。HBase 将所有列分组为*列族*，一起保存到 HDFS。所以，将数据的写入和访问在相似模式下的列进行分组，是一个好主意。

HBase 客户端 API 允许使用 Java 程序与 HBase 集群进行交互。使用 Put 的方式写入到 HBase，代表一次写入一行。每个 Put 方法通过远程过程调用（RPC）来写入相同行中的多列。HBase 也支持 Increment 方法，可以用在列中的增量值作为计数器。像 Put、Increment 方法也可以在一个 RPC 访问中更新相同行的多个列值。

在 Flume 与 HBase 交互环境中，虽然 HBase 提供了更新和删除数据的 RPC 调用，不过我们只关心 Put 和 Increment。更多的细节可以参考 HBase 权威指南 [hbase-client]。除了 Java 语言可以和 HBase 进行交互外，HBase 还提供了 Thrift API，可以参考 Apache HBase wiki [hbase-thrift]。

除了客户端 API 以外，HBase 提供了一个和 HBase 集群交互的 Shell 客户端。HBase Shell 有写入、获取、增量、删除、扫描等命令，也可以创建、禁用、清空和删除表 [hbase-shell]。可以用下面的命令启动一个 HBase Shell ：

```
hbase shell
```

HBase 提供基于单行数据操作的原子性保证。如果在单个 Put 中对同一行写入多个列，HBase 可以保证读取所有列的旧值或者所有列的新值，不会一部分新值一部分旧值。不过，HBase 没有提供事务或 ACID（原子性、一致性、隔离性、持久性）。由于在多行上没有事务，因此不能给客户端读取多行提供一致性保证。

如上所述，HBase 基于 HDFS 构建。因此在 HBase 中的数据可以自动备份。 HBase 将其行划分为 *Region*。一个 Region 是两个固定值之间行键的集合。 HBase 将整个数据集分为多个 Region，每个 Region 由叫作 *RegionServer* 的服务托管。在任何时候，一个 RegionServer 托管一个特定的 Region，不过单个 RegionServer 也可以托管多个 Region。通过 RegionServer 托管的那个 Region 来进行每行数据的读或写。 决定哪一个服务器加载的 Region 的服务是 *HBase Master*。Master 是 HDFS NameNode 的 HBase 版本。Master 也会决定一个太大的 Region 什么时候进行分裂等。

Flume 允许用户在 HBase 中写入数据或者增量计数。用户可以将自定义代码嵌入到 Flume 中，用于将 Flume 的事件转换为 HBase 的 Put 或 Increment 操作。我们将会在第 5 章“使用序列化器翻译 Flume 事件为 HBase Put 和 Increment*”一节中讲解到。

总结

本章中讨论了 HDFS 和 HBase 的基本概念。尽管 Flume 支持其他系统，不过 HDFS 和 HBase 却是最重要和最常用的系统。在第 5 章“Sink”中，将会讨论如何使用 Flume 以可扩展的方式将数据写入到这些系统。

参考文献

- *http://research.google.com/archive/gfs.html*
- [hdfs-architecture] HDFS architecture, *Hadoop: The Definitive Guide, 3rd Edition*, Chapter 3
- [serialization] Hadoop serialization, *Hadoop: The Definitive Guide, 3rd Edition*, Chapter 4
- [hbase-client] HBase operationsn, *HBase: The Definitive Guide*, Chapter 3
- [commands] Hadoop commands, *http://bit.ly/1p9jalP*
- [hbase-thrift] HBase Thrift API, *http://wiki.apache.org/hadoop/Hbase/ThriftApi*
- [hbase-shell] HBase shell, *http://wiki.apache.org/hadoop/Hbase/Shell*
- [input-format] Hadoop input format, *Hadoop: The Definitive Guide*, Chapter 7
- [mr] MapReduce, *Hadoop: The Definitive Guide*, Chapter 2

第2章 <7>

用Apache Flume处理流数据

用中间系统推送数据到 HDFS 和类似的存储系统是非常普遍的使用案例。例如 Apache Flume、Apache Kafka、Facebook 的 Scribe 等几个系统，都支持这种使用案例。这些系统允许 HDFS 和 HBase 集群处理数据偶然突发的情况而没必要具备处理持续写入速度的能力。这些系统在数据生产者和最终目的地之间担当缓冲的角色。凭借其缓冲能力，它们能够平衡在生产者和消费者之间的阻抗不匹配，因此可以提供一个稳定的流状态。扩展这些系统比扩展 HDFS 或者 HBase 集群更容易。这样的系统也允许应用程序推送数据，而不用担心万一 HDFS 放降，必须缓冲数据和重试等。

大多数这种系统都有一些基本类似之处。通常，这些系统通过 RPC 调用成 HTTP（可以通过客户端 API 展现），接收生产者数据的组件，通过 RPC 调用或 HTTP（可以通过客户端 API 展现）。它们也有作为缓冲区的组件，数据存储在缓冲区直到数据被移动到下一阶段或目的地。在这一章中，我们将会讨论 Flume Agent 的基本架构和如何配置 Flume Agent 来将数据从各种各样的应用中移动到 HDFS 或者 HBase。

在大企业中，Apache Hadoop 正在变成一个标准的数据处理框架。应用程序经常产生大规模数量级的数据，并写入到 HDFS——Hadoop 基础的分布式文件系统。Apache Flume 被构思为这样一个系统，以一种可靠的和可扩展的形式将数据写入 Apache Hadoop 和 Apache HBase。因此，Flume 的 HDFS Sink 和 HBase Sink 提供了一个非常丰富的功能集合，使得它可以用这些系统支持的任意格式和 MapReduce/Hive/Impala/Pig 友好型的方式写入数据。本书将论述我们为什么需要一个类似 Flume 的系统、Flume 的设计和实现，以及使得 Flume 高扩展、灵活和可靠的各种特征。 <8>

我们需要 Flume

为什么我们真的需要一个类似 Flume 的系统呢？为什么不是简单地从每个生产数据的应

用服务器直接写入数据到 HDFS？这一节我们将会论述为什么需要这样的系统，以及它增加了什么内容到架构中。

将系统之间彼此隔离的消息系统已经存在了很长时间，在 Hadoop 环境中 Flume 做了这样的工作。Flume 是专门设计用来从大量的源，推送数据到 Hadoop 生态系统中各种各样存储系统中去的，例如 HDFS 和 HBase。

一般来说，当在 Hadoop 集群上，有足够数据处理的时候，通常会有很多生产数据的服务器。这些服务器的数量是上百甚至是上千的。这样庞大数量的服务器试着将数据写入 HDFS 或者 HBase 集群，会因为多种原因导致重大问题。

HDFS 确切地需要一个客户端写入到文件——因此，在同一时间可能有成千上万的文件写入。每次一个文件被创建或者一个新块被分配，在 NameNode 节点会发生一组复杂的操作。如此庞大数量的操作同时发生在服务节点上，可能会造成服务器承受巨大压力。而且，当成千上万的机器写大量数据到小数量机器的时候，连接这些机器的网络可能会不堪重负并且开始经历严重的延迟。

很多情况下，存在于多个数据中心的应用程序服务器，在托管 Hadoop 集群的单个数据中心聚合数据，这意味着应用程序必须通过广域网（WAN）写数据。在所有这些情况下，应用程序在尝试写入 HDFS 和 HBase 的时候，可能会遇到严重的延迟。如果托管应用程序的服务器数量或写数据的应用程序数量增加，延迟和失败率可能会增加。因此，在设计写入到 HDFS 的软件时，要把 HDFS 集群和网络延迟作为额外考虑的因素。

大多数应用程序以预见的方式查看生产流量，每天高峰流量有几个小时，其余时间的流量很小。为了确保应用程序直接写入 HDFS 或 HBase 时，不丢失数据或不需要缓冲很多数据，需要配置 HDFS 或 HBase 集群，以很少或没有延迟的方式处理峰值流量。所有这些案例可以清楚地看到，从 HDFS 或 HBase 隔离生产应用程序，且保证生产应用程序用可控和良好组织的方式推送数据到这些系统，是很重要的。

9 Flume 被设计成为一个灵活的分布式系统，可以很容易地扩展，而且是高度可定制化的。一个配置正确的 Flume Agent 和由相互连接的 Agent 创建的 Agent 的管道，保证不会丢失数据，提供持久的 Channel。

Flume 部署的最简单元是 Flume *Agent*。一个 Flume Agent 可以连接一个或多个其他的 Agent。一个 Agent 也可以从一个或多个 Agent 接收数据。通过相互连接的多个 Flume Agent，一个流作业被建立。这个 Flume Agent 链条可以用于将数据从一个位置移动到另一个位置——特别是，从生产数据的应用程序到 HDFS、HBase 等。

大量的 Flume Agent 从应用服务器接收数据，然后将数据写入到 HDFS 或者 HBase（无

论是直接或者通过其他 Flume Agent），通过简单增加更多的 Flume Agent 就能够扩展服务器的数量并将大量数据写入到 HDFS。

每个 Flume Agent 有三个组件：*Source*、*Channel* 和 *Sink*。*Source* 负责获取事件到 Flume Agent，而 *Sink* 负责从 Agent 移走事件并转发它们到拓扑结构中的下一个 Agent，或者到 HDFS、HBase、Solr 等。*Channel* 是一个存储 Source 已经接收到的数据的缓冲区，直到 Sink 已经将数据成功写入到下一阶段或者最终目的地。

实际上，在一个 Flume Agent 中的数据流以下面几种方式运行：生产 / 接收的数据源写入数据到一个或者更多 Channel，一个或者多个 Sink 从 Channel 读取这些事件，然后推送它们到下一个 Agent，或者存储和索引系统。

Flume Agent 可以被配置成在数据被写入到目的地之前，从管道的一个 Agent 发送数据到另一个 Agent。一旦数据到达 Flume Agent，数据的持久性完全取决于 Agent 使用的 Channel 的持久性保证。在一般情况下，当一个 Flume agent 被配置成使用任何的内置 Source 或 Sink 以及一个持久的 Channel，Agent 保证不会丢失数据。凭借独立 Agent 不丢失数据的优势，Flume 管道也不会丢失任何数据。

如果 Flume 管道中有意想不到的错误 / 超时并进行了重试，Flume 会产生重复的数据最终被写出。如果托管持久 Channel 的硬盘遇到不可恢复的失败，Flume 可能会因为硬盘故障而丢失数据。虽然可能会引起冗余，Flume 允许用户通过冗余流复写事件，以确保硬盘和 Agent 失败是可控的。所以，用户可能必须通过做一些后期处理来确保妥善处理这些冗余。

Flume 是否适合呢？

Flume 将数据表示为事件。事件是非常简单的数据结构，具有一个主体和一个报头集合。事件的主体是一个字节数组，通常是 Flume 传送的负载。报头被表示为一个 map，其中有字符串 key 和字符串 value。报头并不是用来传输数据的，只为了路由的意图和跟踪发送事件的优先级和严重性。报头也可以用于给事件增加事件 ID 或者 UUID。

每个事件本质上必须是一个独立的记录，而不是记录的一部分。这也要求每个事件要适应 Flume Agent JVM 的内存。如果使用 File Channel，应该有足够的硬盘空间来支持。如果数据不能表示为多个独立记录，Flume 可能不适用于这个用例。

Flume 主要是从大量生产服务器将数据推送到 HDFS、HBase 等。在 Flume 不是很适合的场景下，通常有更容易的方法，例如 Web HDFS 或者 HBase HTTP API，都可以用于写数据。如果只有少量的生产服务器生产数据并且数据不要求实时写出，那么通过 Web

HDFS 或 NFS 将数据移动到 HDFS 也是有意义的，尤其是如果写出的数据量相对小——每几小时有几个 GB 的若干文件不会损害 HDFS。在这种情况下，设计、配置和部署 Flume 可能不值得做。Flume 真正做的是实时推送事件，数据流是持续且量级很大的情况。

如前面所指出的，Flume 最简单的部署单元叫作 *Flume Agent*。Agent 是一个 Java 应用程序，接收或者生成数据并缓冲数据，直到最终写入到 Agent 或者存储或索引系统。我们将在下一节论述 Flume Agent 的三个主要组件（Source、Channel、Sink）。

Flume Agent 内部原理

如前面所说，每个 Flume Agent 包含三个主要组件：Source、Channel、Sink。在本节，我们将描述这些组建以及其他组件，以及它们是如何协同工作的。

Source 是从一些其他产生数据的应用中接收数据的活跃组件。有自己产生数据的 Source，不过这些 Source 通常用于测试目的。Source 可以监听一个或者多个网络端口，用于接收数据或者可以从本地文件系统读取数据。每个 Source 必须至少连接一个 Channel。基于一些标准，一个 Source 可以写入几个 Channel，复制事件到所有或某些 Channel。

11 一般来说，Channel 是被动组件（虽然它们可以为了清理或者垃圾回收运行自己的线程），缓冲 Agent 已经接收，但尚未写出到另一个 Agent 或者存储系统的数据。Channel 的行为像队列，Source 写入到它们，Sink 从它们中读取。多个 Source 可以安全地写入到相同的 Channel，并且多个 Sink 可以从相同的 Channel 进行读取。可是一个 Sink 只能从一个 Channel 读取。如果多个 Sink 从相同的 Channel 读取，它可以保证只有一个 Sink 将会从 Channel 读取（和提交——更多参见第 4 章）一个指定特定的事件。

Sink 连续轮询各自的 Channel 来读取和删除事件。Sink 将事件推送到下一阶段（RPC Sink 的情况下），或到最终目的地。一旦在下一阶段或其目的地中数据是安全的，Sink 通过事务提交通知 Channel，可以从 Channel 中删除这些事件。

图 2-1 展示了拥有一个 Source、Channel 和 Sink 的简单 Flume Agent。

图2-1 带有一个Souce、Channel和Sink的简单Flume Agent

Flume 本身不限制 Agent 中 Source、Channel 和 Sink 的数量。因此 Flume Source 可以接收事件，并可以通过配置将事件复制到多个目的地。这使得 Source 通过 Channel 处理器、拦截器和 Channel 选择器，写入数据到 Channel 成为可能。

每个 Source 都有自己的 Channel 处理器。每次 Source 将数据写入 Channel，它是通过委派该任务到其 Channel 处理器来完成的。然后，Channel 处理器将这些事件传到一个或多个 Source 配置的拦截器中。

拦截器是一段代码，可以基于某些它完成的处理来读取事件和修改或删除事件。基于某些标准，如正则表达式，拦截器可以用来删除事件，为事件添加新报头或移除现有的报头等。每个 Source 可以配置成使用多个拦截器，按照配置中定义的顺序被调用，将拦截器的结果传递给链的下一个单元。这就是所谓的*责任链*（*chain-of-responsibility*）的设计模式。一旦拦截器处理完事件，拦截器链返回的事件列表传递到 Channel 列表，即通过 *Channel 选择器*为每个事件选择的 Channel。

Source 可以通过处理器 - 拦截器 - 选择器路由写入多个 Channel。Channel 选择器是决定每个事件必须写入到 Source 附带的哪个 Channel 的组件。因此拦截器可以用来插入或删除事件中的数据，这样 Channel 选择器可以应用一些条件在这些事件上，来决定事件必须写入哪些 Channel。Channel 选择器可以对事件应用任意过滤条件，来决定每个事件必须写入哪些 Channel，以及哪些 Channel 是必需的或可选的。

写入到必需的 Channel 失败将会导致 Channel 处理器抛出 ChannelException，表明 Source 必须重试该事件（实际上，所有的事件都在那个事务中），而未能写入可选 Channel 失败仅仅忽略它。一旦写出事件，处理器会对 Source 指示成功状态，可能发送确认（ACK）给发送该事件的系统，并继续接受更多的事件。图 2-2 展示了该流程。

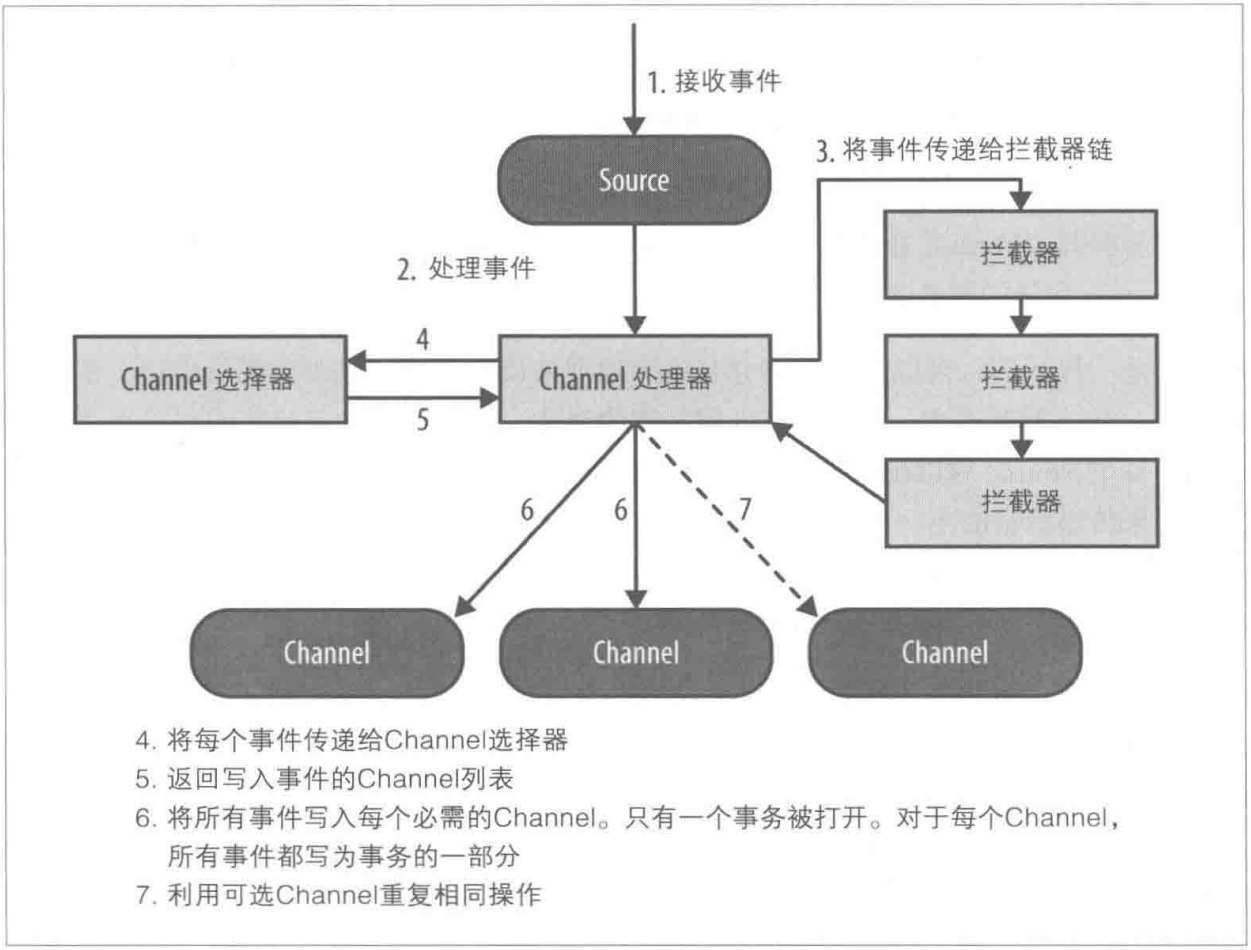


图2-2 Source、Channel处理器、拦截器和Channel选择器之间的交互

Sink 运行器（Sink Runner）运行一个 Sink 组（*Sink Group*），Sink 组可含有一个或多个 Sink。如果组中只存在一个 Sink，那么没有组将更有效率。Sink 运行器仅仅是一个询问 Sink 组（或 Sink）来处理下一批事件的线程。每个 Sink 组有一个 Sink 处理器（Sink Processor），处理器选择组中的 Sink 之一去处理下一个事件集合。每个 Sink 只能从一个 Channel 获取数据，尽管多个 Sink 可以从同一个 Channel 获取数据。选定的 Sink（或如果没有组，唯一的 Sink）从 Channel 中接收事件，并将事件写入到下一阶段或最终目的地。这些流程如图 2-3 所示。

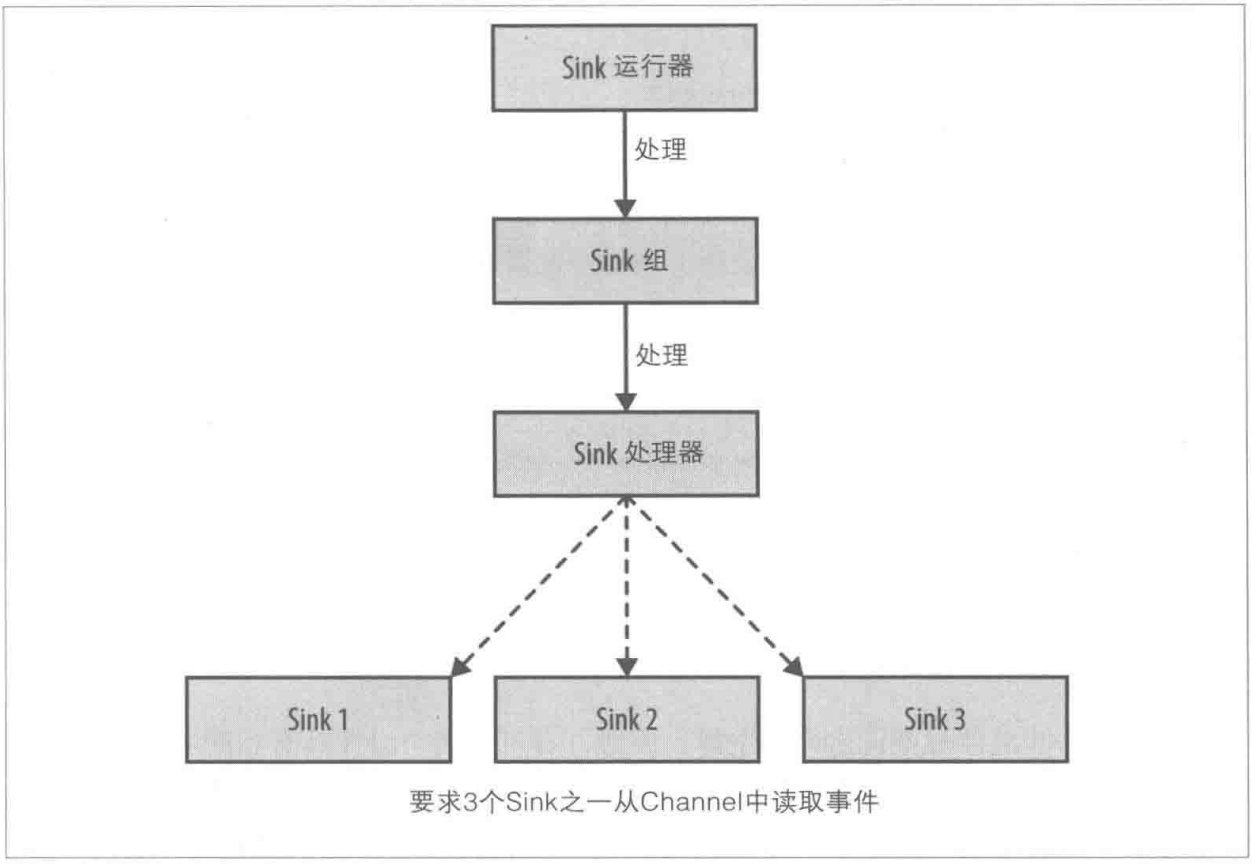


图2-3 Sink、Sink运行器、Sink组和Sink处理器

配置 Flume Agent

Flume Agent 使用纯文本配置文件来配置。Flume 配置使用*属性文件格式*，仅仅是用换行符分隔的键值对的纯文本文件。属性文件的实例如下所示：

```
key1 = value1
key2 = value2
```

通过使用该格式，Flume 可以很容易地将配置传给 Agent 和它的各种组件。在配置文件中，Flume 使用分层结构。每个 Flume Agent 都有一个名称，在 Flume Agent 使用 flume-ng 命令启动的时候设置（本章“运行 Flume Agent”一节中有描述）。配置文件可以包含若干 Flume Agent 的配置，但实际只加载 flume-ng 命令中指定名称的 Agent 配置。

在 Flume Agent 有一些组件可以有若干实例，像 Source、Sink、Channel 等。为了能够识别这些组件的每一个实例的配置，需要对这些组件进行命名。对一个 Agent，配置文件必须使用下面的格式列出 Source、Sink、Channel 和 Sink 组的名称，该列表称为*活跃列表*：

14

```
agent1.sources = source1 source2
```

```
agent1.sinks = sink1 sink2 sink3 sink4
agent1.sinkgroups = sg1 sg2
agent1.channels = channel1 channel2
```

上面的配置片段表示名为 agent1 的 Flume Agent，带有两个 Source、两个 Sink 组、两个 Channel、四个 Sink。即使某些部件罗列出配置参数，如果它们不是在活跃列表中，则它们不创建、配置或启动。其他组件，例如拦截器和 Channel 选择器，不需要存在于活跃列表中。当和它们有关的组件（Source、Sink 等）是活跃的，它们会自动创建并激活。

对于要配置的每个组件，组件的配置使用下面格式的前缀传递：

```
<agent-name>.<component-type>.<component-name>.<configuration-parameter> = \
<value>
```

用于 Source 的 <component-type> 部分是 sources，Sink 是 sinks，Channel 是 channels，Sink 组是 sinkgroups。如拦截器、Channel 选择器和 Sink 处理器等组件是绑定到单个顶级组件，并使用相同的配置模式锚定到这些组件的。

组件名称根据其组件类型设定命名空间。因此，有可能多个组件具有相同名称，只要它们的组件类型不同即可。像拦截器这样的组件也设定独立的 Source 命名空间，多个拦截器具有相同名称是可能的。但是不推荐这种方式。

例如，可以通过下面的格式传递配置到 source1：

```
agent1.sources.source1.port = 4144
agent1.sources.source1.bind = avro.domain.com
```

对于每个组件，配置参数键的前缀被移除（包括组件的名称）。只有实际的参数和它的值通过传递到配置方法的一个 Context 类的实例来传入。Context 是类似 Map 的键值存储，并带有一些稍微复杂的方法。因此，在这种情况下，Source 只在 Context 实例中获取了两个参数，键是 port 和 bind，值是 4144 和 avro.domain.com（并不是整个配置行）。当我们讨论每个组件的配置时，这些表将只显示传递到组件的实际参数，而不是配置文件中的所有行。

15 对于 Source、Sink、Channel 和拦截器，Flume 配置使用 type 参数实例化组件。该 type 参数可以是完整类别名称完全限定类名（FQCN）或内置组件的别名。指定 type 参数的示例如下：

```
agent1.sources.source1.type = avro
```

Flume 配置系统还通过创建 Channel 处理器和为每个 Source 的处理器设定正确的 Channel，确保为每个 Source 设置正确的 Channel。它还负责拦截器的初始化，所以来自

正确 Source 的正确 Channel 处理器可以传递事件到正确的拦截器（虽然从配置来看，拦截器看起来像是 Source 的子组件——Source 实际上并不需要创建或配置拦截器）。类似地，每个 Sink 的 Channel 也由配置系统来设置。配置系统增加 Sink 到正确的 Sink 组，还为 Sink 组配置 Sink 处理器。

需要从 Flume 配置系统获取配置的任何组件，都必须实现 Configurable 接口，如例 2-1 所示。

例2-1 Configurable接口

```
package org.apache.flume.conf;
public interface Configurable {
  public void configure(Context context);
}
```

组件可具有子组件，子组建也可以是可配置的。每个组件必须配置其子组件或传递子组件的配置到子组件，然后子组件必须进行自配置。虽然每个子组件可以用组件指定实现的任何方式进行配置，但是实现 Configurable 接口是一种很好的做法。使用 Context 类的 getSubProperties 方法，可以将指定给子组件的子属性传递给子组件。

配置子组件的一个例子是 HDFS 序列化器，在第 5 章的“使用序列化器控制数据格式 *”一节中有详细描述。序列化器通过使用 Configurable 接口和 getSub Properties 方法，由 HDFS Sink 配置。序列化器可以使用后缀 serializer. 对 hdfsSink 进行配置。在配置文件中，序列化器获取 serializer. 后面的子串作为键。在下面的例子中，序列化器将获取带一个键值对的 Context 实例。键是 bufferSize 且值是 4096：

```
agent.sinks.hdfsSink.serializer.bufferSize = 4096
```

带可配置子组件的所有 Flume 组件都遵循这种模式，每个子组件只获取移除所有前缀的自己的参数。这对于子组件的子组件也适用，前提是有子组件存在。

例 2-2 展示了具有多个组件的 Flume Agent 的一个例子，其中一些组件拥有子组件。在该 Agent 中，有一个 Source、两个 Channel 和两个 Sink。Source 是一个 HTTP Source，命名为 httpSrc。该 Source 写入两个内存 Channel，memory1 和 memory2——这是由配置系统在 Channel 处理器中设置，Source 实现不需要设置 Channel。多个参数——bind、port、ssl、keystore、eystore-password、handler 和 handler.insertTimestamp——它们的值在传递给配置方法的 Context 实例中是可用的。Source 的实现决定了传递给它的任何配置参数都负责什么事情。

对于此配置文件，配置系统还创建了一个拦截器，用于 HTTP Source 接收到的所有事件的转发。在该实例中，HTTP Source 不需要理会拦截器创建或配置的任何特殊处理。拦

截器的创建和配置由 Channel 处理器来处理。同样地，所有其他组件，包括那些子组件，通过 configure 方法获得参数和值。

例2-2 典型Flume Agent配置

```
agent.sources = httpSrc
agent.channels = memory1 memory2
agent.sinks = hdfsSink hbaseSink

agent.sources.httpSrc.type = http
agent.sources.httpSrc.channels = memory1 memory2

# Bind to all interfaces
agent.sources.httpSrc.bind = 0.0.0.0
agent.sources.httpSrc.port = 4353

# Removing this line will disable SSL
agent.sources.httpSrc.ssl = true
agent.sources.httpSrc.keystore = /tmp/keystore
agent.sources.httpSrc.keystore-password = UsingFlume

agent.sources.httpSrc.handler = usingflume.ch03.HTTPSourceXMLHandler
agent.sources.httpSrc.handler.insertTimestamp = true

agent.sources.httpSrc.interceptors = hostInterceptor
agent.sources.httpSrc.interceptors.hostInterceptor.type = host

# Initializes a memory channel with default configuration
agent.channels.memory1.type = memory

17 # Initializes a memory channel with default configuration
agent.channels.memory2.type = memory

# HDFS Sink
agent.sinks.hdfsSink.type = hdfs
agent.sinks.hdfsSink.channel = memory1
agent.sinks.hdfsSink.hdfs.path = /Data/UsingFlume/%{topic}/%Y/%m/%d/%H/%M
agent.sinks.hdfsSink.hdfs.filePrefix = UsingFlumeData

agent.sinks.hbaseSink.type = asynchbase
agent.sinks.hbaseSink.channel = memory2
agent.sinks.hbaseSink.serializer = usingflume.ch05.AsyncHBaseDirectSerializer
agent.sinks.hbaseSink.table = usingFlumeTable
```

Flume Agent 之间的相互通信

在下面几节中我们将看到，几乎总是需要一个 Flume Agent 发送数据到另一个 Agent。为了实现这一目标，Flume 内置了专门的 RPC sink-source 对。对于 agent-to-agent 通信，首选的 RPC sink-RPC source 对是 Avro Sink-Avro Source 对。

要从其他 Flume Agent 或客户端接收数据，Agent 接收数据可以被配置为使用 Avro Source，且发送数据的 Agent 必须配置用来运行 Avro Sink。Avro Sink 是发送事件到 Avro Source 的特定的 Sink。除了 Avro Sink，Flume RPC 客户端也可以发送事件到 Avro Source。Avro 通过 Avro RPC 协议，从其他 Flume Agent 或运行 Flume RPC 客户端的应用程序接收数据。单个 Avro Source 可以从大量的客户端或 Flume Agent 接收数据。即使单个 Avro Sink 只可以发送数据到单独一个 Avro Source，但是使用我们将在第 6 章“Sink 组和 Sink 处理器”一节讨论的 Sink 组和 Sink 处理器，可以从一个 Agent 发送数据到其他多个 Agent。

我们将在第 3 章“Sink-to-Source 通信”一节更详细地讨论这个问题。现在，重要的是要理解，可以从一个 Flume Agent 将数据发送到另一个 Agent，且通过客户端 API 也可以从自定义应用程序将数据发送到 Flume Agent（我们将在第 7 章进行讨论）。

复杂的流

如前所述，Flume Agent 可以有几个 Source、Sink 和 Channel，但是单个 Agent 中，这些组件的数量必须谨慎配置，以确保硬件不会崩溃。由于每个 Source 实际上可以写入到多个 Channel，事件可以很容易被复制，以确保每个事件到达多个目的地。然后，Sink 可以从 Channel 中删除数据，将数据推送到不同的目的地。

流是一个或多个的一系列 Agent，用来推送数据到另一个 Agent 并最终推送到存储或索引系统。在现实中，流可以任意地复杂，每三个组件作为多个流的一部分，且流包括多个 source-channel-sink 三件套。

一般来说，Flume 是为了从大量的服务器把数据发送到一个单独的 HDFS 集群。Flume 内置的 Avro 和 Thrift sink-source 对，可以用于从一个 Flume Agent 将数据发送到另一个。这允许用户设计从大量产生数据的应用程序的 fan-in-style 流。重要的是，要限制写数据到任意存储系统的应用程序的数量，以确保带大量数据写入的存储系统规模，并且可以处理写操作的突发性。

在集群内，有多种方法可以组织 Flume Agent。第一个且最简单的一个是，部署 Flume Agent 的单个的层，以接收从应用程序服务器来的数据，且用相同的 Agent 直接写入数据到存储系统。这样的系统从存储系统中隔离应用程序的错误，并允许存储系统通过吸纳不断增长的输入速度，处理周期性爆发的数据。每次写入失败时，通过回退增长的时间量（直到一个最大周期），Flume 将调整存储系统的写入速率，如果能力低于处理当前写入速度，不至于使得存储系统崩溃。

单层中 Flume Agent 的数量通常只需要占应用程序服务器总数的一小部分，因为 Flume 的 Avro Source 和 Thrift Source 被设计用来接收大量的服务器的大量数据，尽管每个 Agent 的最大容量取决于具体的硬件部署、网络、延迟需求等。图 2-4 展示了设计为这样的拓扑。

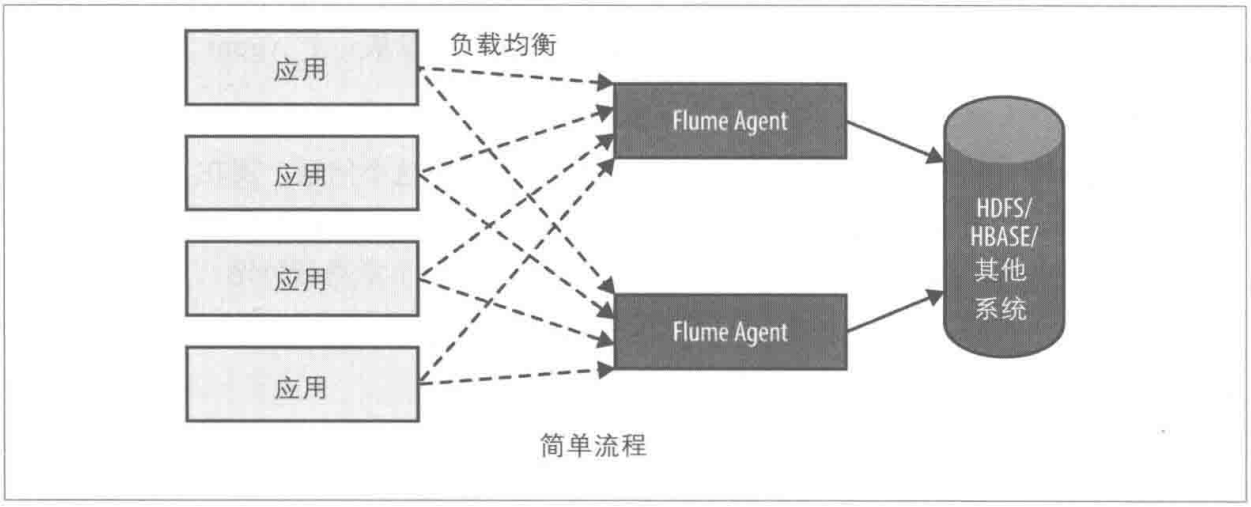


图2-4 使用Flume从大量应用程序服务器将数据聚合到HDFS

19 要设计一个 fan-in 拓扑，需要大量的 Flume Agent 接收产生数据的应用程序的数据，而少量 Agent 将数据写到存储系统。基于有多少服务器生产多少数据，Agent 可以组织到一个、两个或者更多的层，每一层的 Agent 使用 RPC sink-RPC source 的组合，从一层转发数据到另一层。

如图 2-5 所示，最外面的层有最大数量的 Agent 接收应用程序的数据，尽管 Flume Agent 的数量通常只占应用服务器数量的一小部分，但具体的数量取决于多种因素，包括网络、硬件和数据总量。当应用程序产生更多的数据或添加了更多服务器，很容易通过在最外层添加更多的 Agent，并配置它们向第二层的机器写数据来扩展。

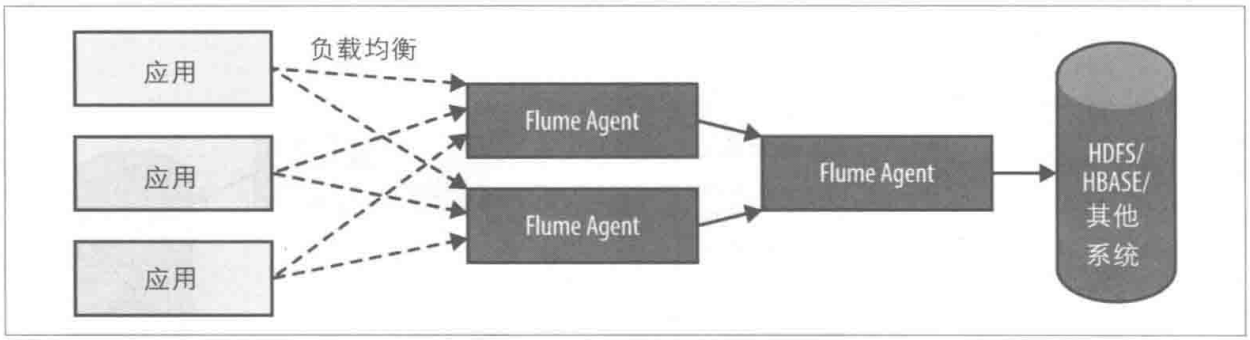


图2-5 fan-in流

通常第二层比最外层扩展 Agent 要慢得多，因为写数据到第二层的服务器数量（最外层的 Flume Agent 的数量）比应用程序服务器数量的增长速度要缓慢得多。这是因为 Flume Agent 的第一层将承受增加应用服务器造成的大部分影响。

这种拓扑结构允许 Flume 根据需要回退，来控制写入到存储系统的速率，同时还允许应用程序无须担心地写入数据。当生产数据的应用程序部署在许多不同的数据中心时，这种拓扑结构也经常使用，并聚合数据到一个集群。通过在同一个数据中心写入到 Flume Agent，应用程序可以避免跨数据中心广域网链接写数据，从而确保数据最终会持久化到存储系统。Flume Agent 之间的通信可以配置为允许更高的跨数据中心的延迟，以确保 agent-to-agent 通信能够没有超时地成功完成。

通过不过度地使用任何一个 Agent 或层，并尽快移除每一层的数据，可以让更多的层承受更长和更大的峰值负荷。因此，层需要的数量主要取决于被推送到 Flume 部署的数据总量。因为最外层接收来自大量机器的数据，那么这层应该有最大数量的 Agent 来扩展网络架构。随着我们进一步讨论 Flume 拓扑结构，Agent 的数量可以明显减少。

如果生产数据的服务器数量持续增加，该层从应用程序服务器接收数据的 Flume Agent 数量也需要增加。这意味着在某种程度上，它可能需要增加后续层，尽管后续层上 Agent 的数量可以比外部层增加的速度更慢。这也确保在 Flume 设置内能增加缓冲容量，以适应数据产量的增加。图 2-6 展示了这样一个流的例子。

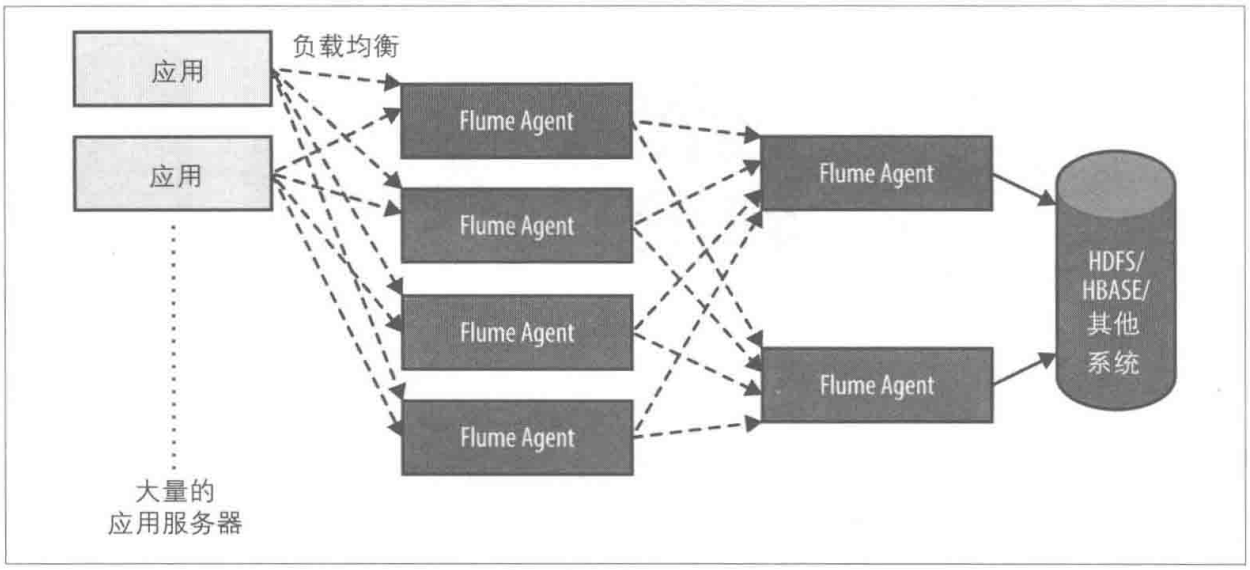


图2-6 一个支持大量应用服务器、提供大量缓冲的巨大的复杂的拓扑结构

在大多数情况下，应用程序和 Flume Agent 之间、Flume Agent 本身之间的通信，对于 Agent 或机器失败必须是弹性的。即使当机器宕机，失败机器上的数据是不可用的，如果容量尚未耗尽应该不会引起积压。Flume 具备针对这样一类应用程序的特性，该类应用程序允许使用 Flume API 自动在多个 Flume Agent 之间负载均衡（这将是 Flume Agent 最外层的一个子集），也允许 Sink 通过 Sink 处理器，在下一层的多个 Agent 之间负载均衡。如果仍有容量存在于失败的 Agent 上，这两个组合能确保数据流持续。

复制数据到不同目的地

通常，在 HBase 中将事件计数器和一些元数据聚合在一起，来用于用户应用程序的实时查询，同时实际数据写入 HDFS 做详细的处理和分析。Flume 也允许这样的拓扑。图 2-7 展示了这样一个例子。

21

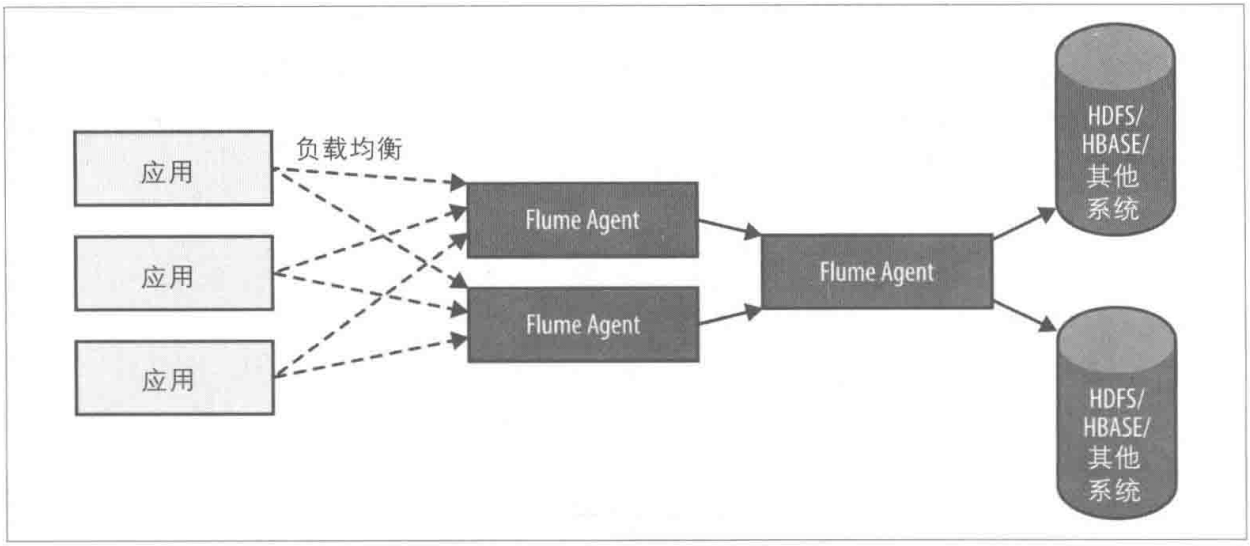


图2-7 复制数据到不同目的地

完成这项工作是相当简单的。要做到这一点，Avro Source 应该配置为，写入到 HDFS 和 HBase 读取的 Channel。因为多个 Sink 可以从一个 Channel 读取事件，那么可以配置更多的 HDFS 和 HBase Sink，来自每个 Channel 中的读操作将分别输送到 HDFS 和 HBase。这里显示了这样一个 Agent 的示例配置（每个组件的特定配置参数都忽略了这一点）：

```
agent.sources.avro.type = avro
# The following line causes the Avro Source
# to replicate data to channels feeding HDFS and HBase
agent.sources.avro.channels = hdfsChannel hbaseChannel
agent.channels.hdfsChannel.type = file
agent.channels.hbaseChannel.type = file
agent.sinks.hdfsSink1.type = hdfs
agent.sinks.hdfsSink1.channel = hdfsChannel
agent.sinks.hdfsSink2.type = hdfs
agent.sinks.hdfsSink2.channel = hdfsChannel
agent.sinks.hbaseSink2.type = hbase
agent.sinks.hbaseSink2.channel = hbaseChannel
```

动态路由

Flume 的一个重要特性是*动态路由*。传入的事件数据往往具有不同的优先级，或不需要到相同的数据存储系统——一些可能只需要到 HDFS，而其他事件可能需要到 HDFS 和 HBase，或基于优先级或其他标准数据，可能到不同的集群。在这些情况下，事件必须根据一些标准进行路由。Flume 支持使用多路复用 Channel 选择器。多路复用 Channel

22

选择器是一种，通过特定报头的值，检查每一个经过选择器传入事件的 Channel 选择器；基于此值，它选择一组事件必须写入的 Channel，图 2-8 中有举例说明。这是 Flume 内置的，选择的报头、值和 Channel 却是可配置的。配置动态路由涉及更多，所以，我们将在第 6 章“Channel 选择器”一节更详细地讨论它。

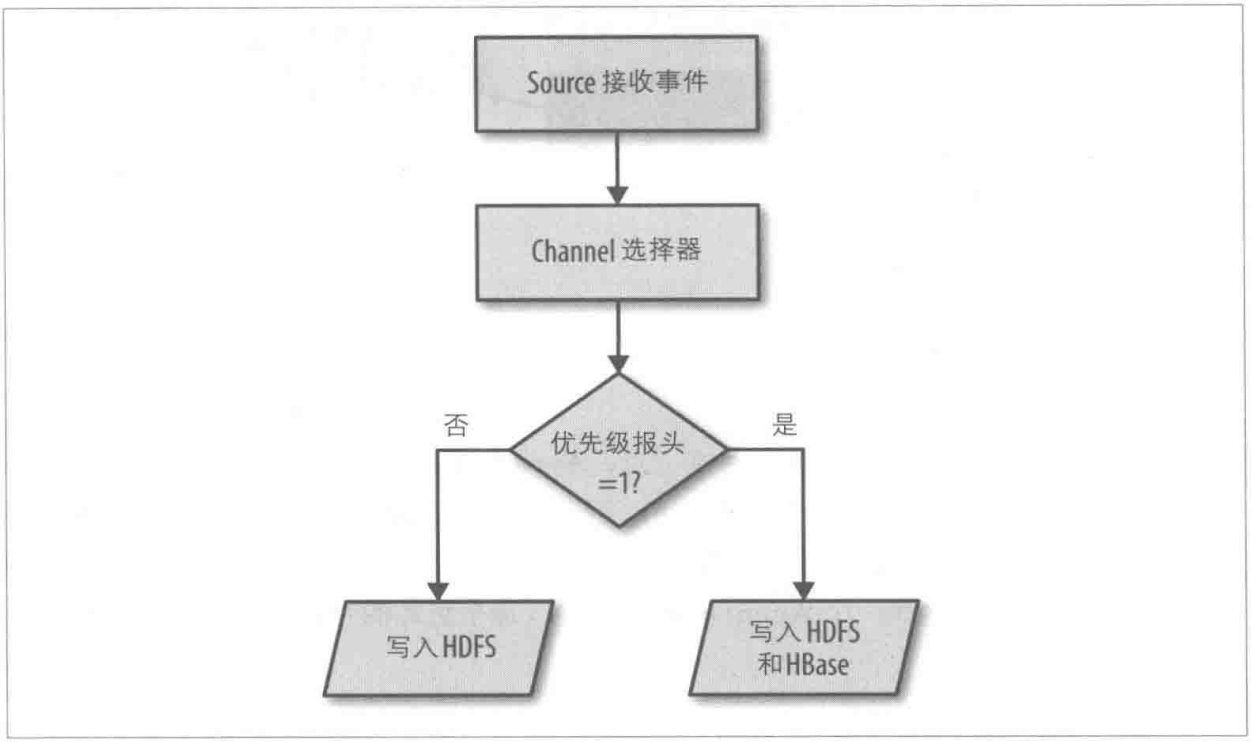


图2-8 动态路由

当配置动态路由时，中间层是很重要的。在某一层之后有一个额外的层来做动态路由的工作，以保证一旦流分为两部分，每一个新的流不是直接去到目的地，而是在更多的层都有缓冲。

Flume 的无数据丢失保证，Channel 和事务

如果配置正确，Flume 提供了无数据丢失的保证。当然，一旦管道中所有 Flume Agent 的容量之和被使用完，Flume 将不再接受来自客户端的数据。此时，客户端需要缓冲数据，否则数据可能会丢失。因此，配置管道能够处理最大预期的停机时间是非常重要的。我们将在第 8 章讨论 Flume 管道的配置。

Flume 的持久性保证依赖于使用的持久性 Channel 的保证。Flume 自带两类 Channel：*Memory Channel* 和 *File Channel*。Memory Channel 是一个内存缓冲区，因此如果 Java 虚拟机（JVM）或机器重新启动，任何缓冲区中的数据将丢失。另一方面，File Channel

是在磁盘上的。即使 JVM 或机器重新启动，File Channel 也不丢失数据，只要磁盘上存储的数据仍然是起作用的和可访问的。机器和 Agent 一旦开始运行，任何存储在 File Channel 中的数据将最终被访问。

Channel 本质上是事务性的。此处的事务不同于数据库事务。每个 Flume 事务代表一批自动写入到 Channel 或从 Channel 删除的事件。无论是当 Source 将事件写入 Channel 时，或 Sink 从 Channel 读取事件时，它必须在事务的范围之内进行操作。

Flume 保证事件至少一次被送到它们的目的地。Flume 只有一次倾力写数据，且不存在任何类型的故障事件只被写一次。但是像网络超时或部分写入存储系统的错误，可能导致事件不止被写一次，因为 Flume 将重试写操作直到它们完全成功。网络超时可能表示写操作的失败，或者只是机器运行缓慢。如果是机器运行缓慢，当 Flume 重试这将导致重复。因此，确保每个事件都有某种形式的唯一标识符通常是一个好主意，如果需要，最终可以用来删除事件数据。

Flume Channel 中的事务

事务性语义是由 Flume 保证的“没有数据丢失”的关键。当每个 Source（或 Sink）写入到 Channel 或读取来自一个 Channel 的数据，会利用 Channel 启动一个事务。对于所有 Flume 自带的 Channel，每个事务是本地线程的。出于这个原因，不同类型 Source 和 Sink 的事务处理略有不同，但基本原理是一样的：每个线程应该运行它自己的事务。对于所有 Pollable Source 和所有 Sink——如前所述，是由运行线程驱动的——每个过程调用只应该启动一个事务，并且如果事务回滚则抛出一个异常，以告知运行线程回退。即使 Source 或 Sink 生成多个 I/O 的新线程，也最好遵循这个协议，如果 `process` 方法失败引发了众多事务之一的事务，以避免歧义。

当 Source 写事件到 Channel 时，事务由 Channel 处理器所处理，所以 Source 无须处理自己的事务。只有当事件被成功写入 Channel，Channel 处理器才提交事务；否则，Channel 处理器将回滚该事务并关闭它。因为每个 Source 可以写入到多个 Channel，所以 Channel 处理器可以为 Source 依次写事件并提交它们。因此，数据被写出并被提交到一些 Channel 而不是其他 Channel 是可能的。在这种情况下，Flume 不能回滚已经被提交的事务，但能确保数据成功写出到所有 Channel，Flume 将重试写数据到所有 Channel，包括之前成功写入的那些；这可能会导致重复。

24

对于终端 Sink，只有当数据被安全写出到存储系统，事务才应该被提交。一旦数据在最终目的地是安全的，那么可以提交事务，且 Channel 可以删除该事务中的事件。如果写操作失败，Flume 必须回滚事务，以确保事件不会丢失。所有 Flume 自带的 Sink 都以这种方式工作，并在事务提交前确保数据已经在 HDFS、HBase、Solr、Elastic Search 等。

如果写操作失败或超时，那么，该事务将被回滚，然后从 Channel 读取事件的 Sink 或另一个 Sink 将再次试着写事件。

前面讨论的技术能确保终端 Sink 以一种持久的方式写出数据，但 agent-to-Flume Agent 或 client-to-Flume Agent 是如何通信的呢？对于 RPC Sink 和 RPC Source 之间的通信，RPC Sink 批量发送事件，这些都从 Channel 读取同一事务的一部分，在单个事务中通过 Source 写入到 Channel。一旦 Source 利用 Channel 成功提交事务，它发送一个确认（ACK）给发送事件的 Sink，表明事件现在在接收事件 Agent 的 Channel 中是安全的。当 Sink 收到这个 ACK，从 Channel 读取事件时，Sink 就提交它打开的事务，表明事件现在可以从 Channel 中被删除。

如果 Source 花费太长时间发送 ACK，或有一些网络问题，导致 Sink 超时，然后 Sink 就假设写操作失败，它回滚事务并再次重复整个过程（连接到该 Channel 的这个或另一个 Sink 可以读取相同的事件）。这种重叠事务的方法保证了在任何时候，在一个 Channel 中的事件是安全的。图 2-9 展示了 RPC Sink 和 RPC Source，在任何时候是如何保证事件在至少一个 Agent 上是安全的时间线。

25

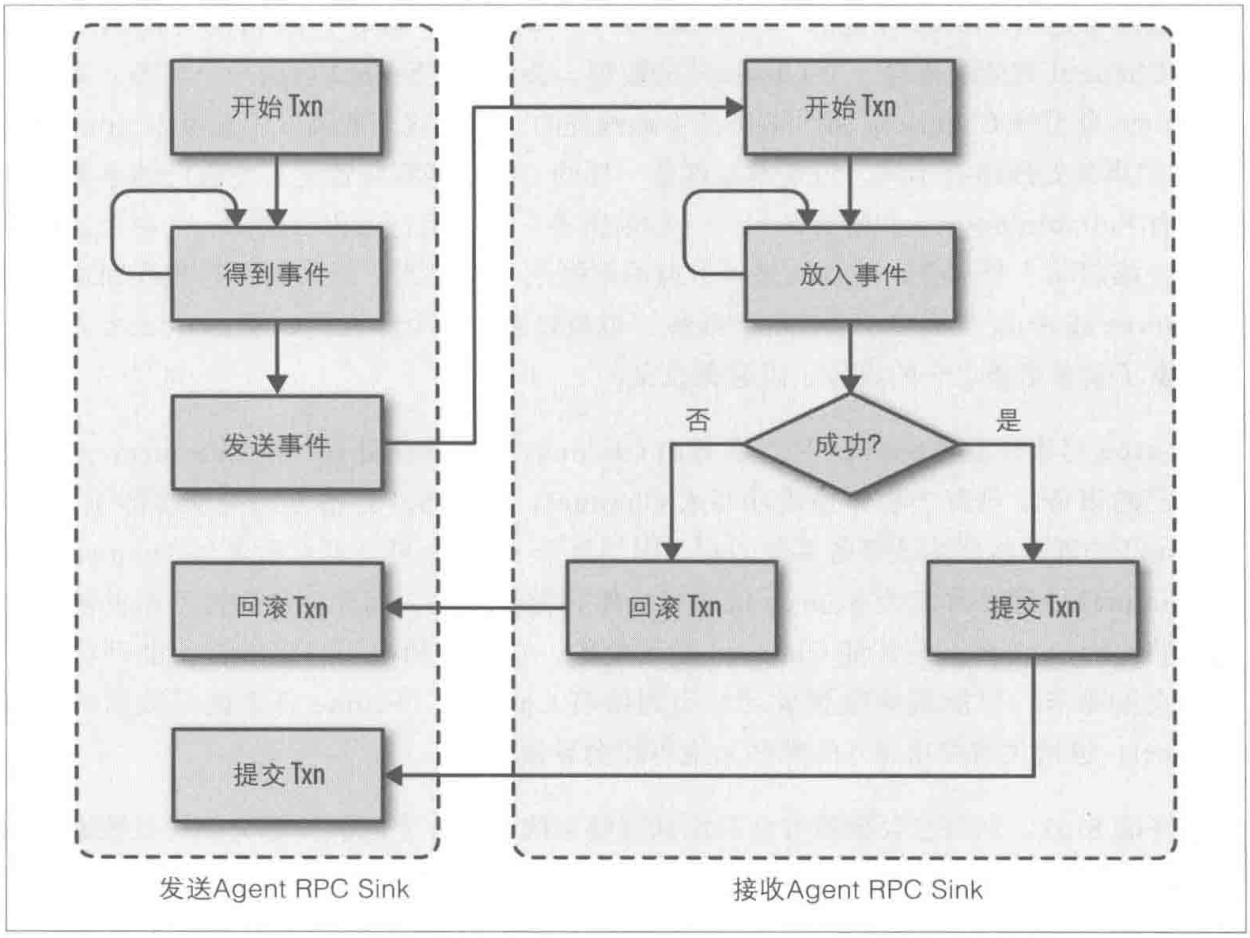


图2-9 通过重叠事务，RPC Sink和RPC Source保证每个事件至少在一个Agent的Channel中

Agent 失败和数据丢失

如果一个 Agent 失败，或存储系统不可达，会发生什么？当一个 Agent 或最终目的地变得不可达，Agent 会在 Sink 看到错误的位置写入数据时结束。Sink 会看到连接失败，或下一阶段或存储系统的 ACK 丢失（或存储系统的客户端 API 抛出异常）。当 Sink 遇到异常，表明数据可能没有被写入，Sink 利用 Channel 回滚事务。因为传入的数据流并没有停止，这导致 Channel 大小增加，最终填满 Channel。当这个 Agent 的 Channel 被填满，Avro Source（或任何其他写数据到该 Channel 的 Source）开始在 Channel 的 Put 操作上得到 ChannelExceptions 异常，通过返回一个错误导致 Source 回退到前一阶段。进而导致前一阶段的 Sink 回滚得到事务，导致那个阶段的 Channel 大小增长。这个场景如图 2-10 所示。

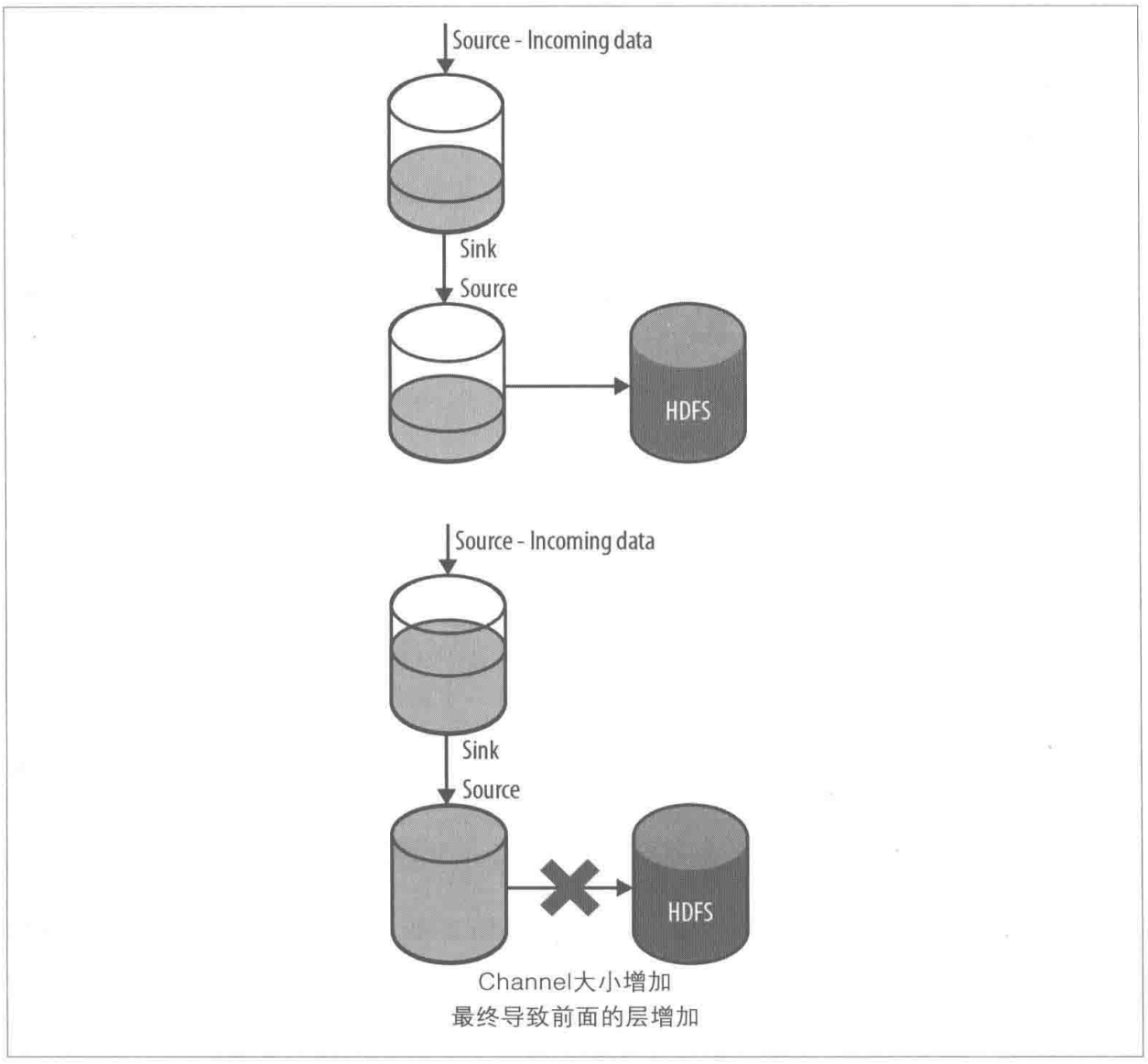


图2-10 失败导致Channel被填满

很明显，失败导致每一层都缓冲数据，直到 Channel 填满，此时它开始回退到前一层，直到所有层都被填满。此时，客户端开始看到错误，那么客户端必须现在通过缓冲数据或删除数据进行处理，从而导致数据丢失。在这种情况下，容量必须进行设计，停机时不应该创建发生这种情况的情景。我们将在第 8 章谈论容量的设计。

批量的重要性

当事件被发送到 Source，Source 通过网络从 RPC Sink 或远程客户端接收数据，Source 通过委托这个任务给 Channel 处理器，在单个事务中成批写出所有事件。

27 对于 File Channel，这是持久的 Channel 实现，每个事务的提交导致一个**文件同步**操作 [fsync]。**文件同步**是 POSIX 标准定义的系统调用，用于告知操作系统为一个特定的文件描述符刷新内部缓冲区的所有数据到磁盘。如果每个事务写入的数据量很小，启动一个系统调用（在时间和资源消耗方面），切换到内核空间，在实际同步到磁盘之前刷新所有缓冲数据，将占据**文件同步**调用本身总成本非常高的分额。

对于 Memory Channel，这是内存 Channel 的实现，与整个 Channel 同步相关的成本，在提交期间发挥作用，但这远远小于**文件同步**调用的开销。

由于实际调用相关的元数据和所有额外的 TCP 开销，RPC 调用有额外开销。当发送的数据量非常小时，这些开销是每个 RPC 调用成本的一大部分，导致不必要的网络利用率等。为了避免这种开销，单个 RPC 调用中批量处理几个事件（当然，除非每个事件本身就很大）或从远程客户端写入，总是一个好主意。

尽管 Flume 的 RPC 客户端和 RPC Sink 支持没有批量或批处理大小为 1 来写事件，为避免比必要的开销支付更多的成倍的开销成本，捆绑事件到合理大小的批次几乎总是一个好主意。理想的批量大小将取决于具体的用例，但是对于最大为几个字节的事件，批量大小在 100 和 1000 之间通常性能很好（虽然特定的硬件、网络和其他因素影响该值，在测试过不同的值，且找到一个匹配所需性能的值，才最后设置）。

批处理影响 RPC Sink 和其他通过网络写数据的 Sink 的性能。如前所述，RPC Sink 性能受多个因素影响。甚至对于 HDFS Sink，当每一批量提交，Flume 刷新事件到所有数据节点的内存。因此，为所有 Sink 使用合理的批量大小总是一个好主意。

有控制批量大小的 Source，例如 Exec Source、JMS Source 等。出于同样的原因，这些 Source 为了性能也应该批处理事件。Source 写事件到 Channel，且它们应该以合理大小的批量写出，以避免在这一节之前讨论的文件同步或同步问题。所以，即使对于能控制自己批量大小的 Source，它们被配置为使用合理大小的批量也是很重要的。

一个合理的批量大小是多少？

这取决于部署、硬件和其他几个因素。没有大量的试错测试，批量大小不应该最终确定，因为批量太大也会导致自己的问题，如网络上太多的碎片等。同时，批量太大会增加事件重复的风险，因为每一批失败最终可能会导致大量的事件再次写入，如果有些事件已经成功写入 HDFS，这些事件最终会重新写入。

要为 RPC 和终端 Sink 选择正确的批量大小，开始是相当于几百 KB 到 1MB 的批量大小，然后基于你看到的超时和重复率，从那里向上或向下调整。如果有太多的重复或超时太多，你必须减小你的批量大小；在相反的情况下，增加批量大小直到开始出现超时。一旦你看到超时，说明已经达到阈值，应该减少几个百分点。

重复怎么样？

Flume 提供至少一次保证，这基本上意味着通过 Flume 到存储系统的发送事件至少将存储一次。但是最终 Flume 可能会不止一次地存储数据。有很多场景会导致重复，有些是由于错误，另一些是由于配置。

因为每个 agent-to-agent RPC 调用有一个可配置的超时，如果没有在超时时间内得到响应，即使 RPC 没有失败，发送事件的 Agent 也有可能认为 RPC 失败，从而引发重试。如果 RPC 没有失败，重试将导致相同事件再次发送，造成重复。这种情况可能发生在终端 Sink，如 HDFS 或 HBase Sink。

而且，由于 Flume Source 可以写入到多个 Channel，如果相同的 Source 配置了多个 Channel，同样的事件基本上可以重复。如果 Sink 从 Channel 读取事件并最终推送事件到相同的存储系统，这也可能会导致重复。

如果用例是重复敏感型的，在事件中插入唯一标识符通常是一个好主意。后续的处理工作可以使用这些标识符删除重复的数据，如使用 Spark、MapReduce 等。

运行 Flume Agent

本节假设 Flume 目录结构没有改变，当前工作目录是 Flume 目录结构的顶层。每个 Flume Agent 使用 flume-ng 命令从命令行启动。这个命令需要几个参数：要启动的 Flume Agent 名称、使用的配置文件、使用的配置目录。

Flume 配置文件可以包含多个 Flume Agent 的配置，每个由一个唯一名称确定。当启动

一个 Flume Agent 的时候，这个名字被传递到 flume-ng 脚本，作为 -n 命令行选项的值。Flume 的配置系统将加载只与特定 Agent 名称相关的配置参数。因为多个 Agent 的配置可以在一个文件中，那么很容易将相同的文件部署到多个层，每一层都有不同的配置。在大多数情况下，因为一层的所有 Agent 与下一层 Agent 的相同集合通信，那层的每个 Agent 可以有一个完全相同的配置。在一层中，所有 Agent 使用相同名称和相同配置文件使得层的部署非常容易实现自动化。配置文件可以使用 -f 命令行选项传递到 Flume Agent。

对于配置目录，可使用 -c 命令行选项将其路径传递给 Flume Agent，它是两个重要文件 *flume-env.sh* 和 *log4j.properties* 的主目录。当启动一个 Flume Agent 时，Agent 初始化脚本将寻找 *flume-env.sh* 脚本。该文件必须包含需要传递给 flume-ng 脚本的任何环境变量。使用 *flume-env.sh* 文件初始化最常用的环境变量都列在表 2-1 中。

表2-1 使用*flume-env.sh*初始化的环境变量

环境变量	描述
FLUME_CLASSPATH	除了 Flume lib 和 *plugins.d* 目录外，传递到 Flume Agent 的类路径，这是自动添加的
JAVA_OPTS	任何传递给 JVM 的 Java 指定选项，包括 -XX 和 -D 参数，用来控制内存和从命令行传递参数
HADOOP_PREFIX (or) HADOOP_HOME	Hadoop 的安装目录（*$HADOOP_PREFIX/bin* 包含 Hadoop 可执行文件）
HBASE_HOME	HBase 的安装目录（*$HBASE_HOME/bin* 包含 HBase 可执行文件）

FLUME_CLASSPATH 环境变量是 Flume 的 *lib* 和 *plugins.d* 目录之外的目录的列表(以:分隔)，这两个目录被添加到 Flume Agent 的类路径中。*plugins.d* 目录是存放自定义组件 JARs (Java Archive files) 的目录，所以 Flume 可以使用它们来加载自定义组件。

JAVA_OPTS 是直接传递给 JVM 的参数集合。最常使用的是用来修改 JVM 分配的 heap size 的参数：-Xms 和 -Xmx。你可以在 Java 文档的 JAVA_OPTS 中找到它们和其他的选项 [Java-commandline]。

HADOOP_PREFIX（在旧的 Hadoop 版本 1 中，等价于 HADOOP_HOME）是 Hadoop 的安装目录。如果 hadoop 命令是 PATH，且 HADOOP_PREFIX 没有设置，Flume 将使用 PATH。对于 HBase，HBASE_HOME 实现相同的功能。

要运行一个名为 usingFlumeAgent 的 Flume Agent，其使用 Flume 配置 */home/usingflume/flume/flume.conf* 和配置目录 */home/usingflume/flume/conf*，则可使用下面的命令（假设目

前的工作目录是 Flume 的安装目录）：

```
bin/flume-ng agent -n usingFlumeAgent -f /home/usingflume/flume/flume.conf
-c /home/usingflume/flume/conf
```

Agent 将在配置目录中读取 *log4j.properties* 文件，并且根据文件中的规范记录日志。log4j 配置细节可以在 log4j 文档中找到 [log4j]。一旦 Agent 启动，它继续运行，直到被 SIGTERM 或其等价命令所杀，导致 Agent 关闭。Agent 记录所有日志到 log4j 日志（可能看起来像 Agent 没有运行，尽管它实际上在运行和做它应该做的工作；没有信息被记录到控制台）。

flume-ng 脚本接受很多参数，在表 2-2 中有所描述。

表2-2 flume-ng脚本接受的命令行参数

参数	描述
-n	使用的 Agent 名字。它必须放置在命令行上的 flume-ng agent 之后
-f	配置文件（没有这项，Agent 将不会运行）
-c	使用的配置目录（如果没有指定，使用 *./conf*）
-C	添加到类路径的目录列表。该项也可以在 FLUME_CLASSPATH 中指定
-d	只是运行。如果运行时没有该选项，将打印出 Flume 使用的整个命令
--plugins-path	如果没有使用 *./plugins.d* 目录作为包含自定义类的 JARs 的目录，那么该参数的值用于插件检查
-h	将打印出详细的帮助

31

尽管 Flume 可以接受参数通过命令行传递给 JVM，但是推荐使用 JAVA_OPTS 设置这些参数，因此它们没有列在这里。

为了找出确切的版本和正在使用的 Flume 版本的修订信息，运行：

```
bin/flume-ng version
```

总结

本章我们讨论了 Flume 的基本概念和设计原理，它的不同种类的组件以及如何配置。最后，我们了解了当 Flume 的配置和组件已经确定时，该如何运行 Flume Agent。

第 3 章、第 4 章和第 5 章将分别讨论 Source、Channel 和 Sink。第 6 章将讨论其他组件——拦截器、Channel 选择器、Sink 组和 Sink 处理器。第 7 章将解释如何使用 Flume 软件开

发工具集（SDK）和 Embedded Agent API。最后一章，第 8 章，将讨论如何设计、部署和监控 Flume 的细节。

参考文献

- [Java-commandline] Java command-line arguments, *http://bit.ly/1p9NzQX*
- [log4j] log4j documentation, *https://logging.apache.org/log4j/1.2/manual.html*
- [fsync] fsync system call, *http://bit.ly/1p9Nzk3*

第3章 源（Source）

33

Source 是负责接收数据到 Flume Agent 的组件。Source 可以从其他系统中接收数据，像 Java 消息服务（JMS）或者其他处理的输出结果。Source 也可以用于接收其他 Flume Agent 的 Sink 通过 RPC 发送的数据。Source 甚至可以生产数据。不夸张地说，Source 可以接收任何来源的数据。

数据源从外部系统或者其他 Agent（或者自身生产）接收数据，并将数据写入到一个或多个 Channel 中，这些 Channel 是提前为 Source 配置过的。这也是 Source 最基本的职责。

本章我们将讨论 Flume 中封装的多种 Source 的设计和工作原理，以及如何才能最优化配置；我们也将介绍如何去编写自定义的 Source。

Source 的生命周期

Source 被命名为像任何其他部件一样通过配置文件进行配置的组件。Flume 的配置系统验证每个 Source 的配置和屏蔽错误配置的 Source。通过配置系统完成的验证相当少，不过，Flume 配置系统可以确保：

- 每个 Source 至少有一个正确配置的 Channel 连接它。
- 每个 Source 有一个定义的 type 参数。
- Source 是在 Agent 中 Sources 里面的活跃列表。

配置系统一旦验证通过一个 Source，就会实例化并且由 `ConfigurationProvider` 进行配置。如果 Source 缺少配置或者缺少需要的参数，这个 Source 将被屏蔽。Source 一旦成功配置，34
Flume 的生命周期管理系统将会尝试启动 Source。只有 Agent 自身停止或被杀死、或者 Agent 被用户重新配置，Source 才会停止。

Source，像所有的 Flume 组件一样，需要在配置文件中指定它们的类型。可以是 FQCN，或者内置 Source 的别名。所有的 Source 都需要至少一个用于写入的正确配置的 Channel。因此，Channel 的列表也是对于 Source 合理配置需要考虑的必需参数。表 3-1 里展示了所需的参数。

表3-1 所有Source的必需配置参数

参数	描述
Type	Source 的类型。可以是 FQCN 或者 Source 的别名（仅为 Flume 自带的 Source）。类必须安装使用“plugins”一节中描述的 *plugins.d* 框架。每个 Flume 封装 Source 的别名在相关部分都会提及
channels	Source 写入 Channel 的时候应该以空格分隔。Channel 选择器基于事件和路由参数会选择出用于写入的 Channel，Source 会按照配置文件中指定的顺序将事件写入这些 Channel。在第 6 章“channel 选择器”一节中有更多的细节描述

类型 avro 的名为 usingFlumeSource 的 Source，运行在一个以 usingFlume 开头的 Agent 中，在文件中将会如下配置：

```
usingFlume.sources = usingFlumeSource
usingFlume.channels = memory

usingFlume.sources.usingFlumeSource.type = avro
usingFlume.sources.usingFlumeSource.channels = memory
usingFlume.sources.usingFlumeSource.port = 7877
usingFlume.sources.usingFlumeSource.bind = 0.0.0.0
```

有一些参数可以任意传递给所有的 Source。这些参数旨在为 Source 配置拦截器和 Channel 选择器，表 3-2 中有详细的说明。这些参数就像其他配置参数一样传递给 Source。

表3-2 所有Source的可选配置参数

参数	描述
Interceptors	代表一连串拦截器的名单
interceptors.<interceptor_name>.*	传递给指定名称拦截器的参数
Selector	Channel 选择器使用的别名或 FQCN；如果没有指定选择器，使用复制 Channel 选择器
selector.*	传递给 Channel 选择器的配置参数

例 3-1 展示了一个实例，即为名为 avro 的 Source 配置两个拦截器。配置两个拦截器，*host* 和 *static*（分别命名为 i1 和 i2），是用来拦截 Source 收到的事件。如上所述，拦截器可以接受 interceptors.<interceptor_name>. 前缀的配置。

例3-1 配置拦截器

```
agent.sources.avro.interceptors = i1 i2
agent.sources.avro.interceptors.i1.type = host
agent.sources.avro.interceptors.i1.preserveExsiting = true
agent.sources.avro.interceptors.i2.type = static
agent.sources.avro.interceptors.i2.key = header
agent.sources.avro.interceptors.i2.value = staticValue
```

每个 Source 恰好有一个 Channel 选择器（这就是为什么它不是一个指定的组件，并且可以使用 selector 配置后缀进行配置）。尽管 Channel 选择器的配置看起来像 Source 的子组件的配置，但是 Source 不需要配置选择器——这是由配置系统完成的。在例 3-2 中，名为 avro 的 Source 的 Channel 处理器配置了一个多路复用 Channel 选择器，用来分流 Source 事件。我们将在第 6 章中讨论多路复用 Channel 选择器的具体配置参数，但是正如这个例子的示范，选择器可以接受配置参数，它们会返回给 Source 写入具体事件的 Channel。

例3-2 Channel选择器配置

```
agent.sources.avro.selector.type = multiplexing
agent.sources.avro.selector.header = priority
agent.sources.avro.selector.mapping.1 = channel1
agent.sources.avro.selector.mapping.2 = channel2
agent.sources.avro.selector.default = channel2
```

如果一个 Agent 被重新配置，相同的 Source 类的实例将不能被重用。因此，所有 Flume 自带的 Source 都是无状态的。我们也期望任何自定义的被嵌入到 Flume 的 Source 是无状态的，以避免数据丢失。

我们已经了解了 Source 的基础概念，现在我们来讨论 Flume 封装的各种 Source。

Sink-to-Source 通信

36

Flume 最重要的特性之一就是 Flume 部署水平扩展的简单性。可以很容易完成扩展的原因是，很容易为 Flume 调度添加新的 Agent，也很容易配置新的这些 Agent 发送数据给其他 Flume Agent。类似地，一旦添加了新的 Agent，仅仅通过更新配置文件，就能很简单地配置已经运行的 Agent 来写入这个新的 Agent。

Flume 灵活性的重点就是 Flume 的 RPC sink-source 的结合。在第 2 章的“Flume Agent

之间的相互通信”一节中已经讨论过，RPC Sink 被设计用来给 RPC Source 发送事件——Thrift Sink 发送给 Thrift Source，Avro Sink 发送给 Avro Source。RPC Sink 和 Source 是高扩展性的，因为 Source 能接收大量的 Sink 或者 RPC 客户端发出的数据。尽管每个 RPC Sink 只能发送数据给一个 RPC Source，但是每个 Agent 可以配置使用 Sink 组和 Sink 处理器，来发送数据给多个其他的 Agent；详情见第 6 章的“Sink 组和 Sink 处理器”。

在这部分中，我们将要讨论 Flume 的 RPC Source，包括配置的各个方面以及如何部署它们。

Avro Source

Flume 主要的 RPC Source 是 *Avro Source*。Avro Source 被设计为高扩展的 RPC 服务器端，能从其他的 Flume Agent 的 Avro Sink 或者使用 Flume 的 SDK 发送数据的客户端应用，接收数据到一个 Flume Agent 中。Avro Source 和 Avro Sink 的组合代表了 Flume 内部的通信原理（在 Flume Agent 之间）。Avro Source 的可扩展性结合 Channel 担当了缓冲器的角色，使得 Flume Agent 能够处理重要的负载峰值。

Flume 的 Avro Source 使用 Netty-Avro inter-process 的通信（IPC）协议来通信。因此可以用 Java 或 JVM 语言发送数据到 Avro Source。如果你想从应用使用 Avro Source 发送事件给一个 Agent，你可以利用 Flume SDK（详情见第 7 章的“Flume 客户端 SDK”一节）或者嵌入式 Agent（详情见第 7 章的“嵌入式 Agent”节）来完成。

Avro Source 可以配置用来从配置好输出压缩事件的 Avro Sink 中接收压缩的事件；它也可以配置来确保接收任何客户端或 Sink 发送的使用 SSL 加密的数据。Avro Source 的配置参数在表 3-3 中将详细列出。

37

表3-3 Avro Source配置

配置参数	默认值	描述
type	-	Avro Source 的别名是 avro，也可以使用完整类别名称，org.apache.flume.source.AvroSource（大小写敏感）
bind	-	绑定的 IP 地址或主机名。使用 0.0.0.0 绑定机器所有的接口
port	-	绑定的端口
threads	infinity	接收从客户端或 Avro Sink 传入的数据的最大工作线程的数量
ssl	false	SSL 应该启用吗？如果设置为 true，所有连接到 Source 的客户端都需要使用 SSL。如果启用了 SSL，keystore 和 keystore-password 参数是必需的

续表

配置参数	默认值	描述
keystore	-	使用 SSL 的 keystore 的路径。如果启用了 SSL 这是一个必需的参数
keystore-password	-	打开 keystore 使用的密码。如果启用了 SSL 这是一个必需的参数
keystore-type	JKS0	正在使用的 keystore 的类型
compression-type	-	用于解压缩传入数据的压缩格式。唯一支持的压缩格式是 zlib。如果要接收 zlib 压缩的数据，设置该参数为 deflate

以最简单的方式配置 Avro Source 需要参数的最小集合。最少的情况下，Source 需要两个除了必需参数以外的参数：bind 和 port。这两个参数指明了 Source 使用的 socket 地址。如果有多个网络接口，Avro Source 可以绑定一个或所有接口。如果只是绑定其中的一个接口，简单地设置那个接口的 IP 地址或域地址为 bind 参数的值即可；如果要绑定所有的接口，设置 bind 参数的值为 0.0.0.0。port 参数为配置的绑定地址指明了 Source 应该监听的端口号。

Avro Source 使用 Netty 服务器 [netty] 来处理传入的请求。Netty 服务器使用 Java 的非阻塞 I/O（NIO）[nio]，这保证了当 Netty 服务器使用相对较少的线程来处理请求的高性能。Avro Source 允许用户配置线程的最大数量，即 Source 使用 threads 参数处理传入的数据的线程数量。这使得用户能监控 Source 所消耗的资源。尽管线程数量理论上可以是无穷大的，但是实际的线程数量受 JVM、操作系统和硬件的限制。

38

SSL Keystores

Keystore 是用 Java 标准定义的加密秘钥和证书的集合 [keystore]。每个 keystore 受可以用来加载 keystore 的密码保护。在 Flume 实例中，密码以纯文本存储在 Flume 配置文件中；配置文件必须以恰当的权限来保护，以避免密码落入他人手中。

如果 Avro Sink 或者 RPC 客户端发送数据给 Flume 时，配置了使用 SSL 发送数据给 Avro Source，Avro Source 必须配置 SSL 相关的参数。ssl 参数必须设置为 true，keystore 和 keystore-password 参数也相应要设置。keystore 参数需要指向一个合法的 keystore 文件，keystore-password 用来打开 keystore 的密码。

keystore-type 参数是可选的，如果需要可以被设置为轮流的 keystore 类型 [keystore-type]。使用的加密算法通过 Java 安全属性文件里的 ssl.KeyManagerFactory.algorithm

定义。如果 Java 安全属性文件这个参数没有设置，就会使用 *SunX509* 算法。更多关于 Java 安全属性文件的细节可在 Java 安全指南（Java Security Guide）中查找到 [Java-security]。

Avro Sink 和 Flume RPC 客户端可以配置用于在发送数据给 Avro Source 之前压缩数据。如果数据在广域网或数据中心之间传输时，这是非常有用的，能够减少使用的带宽。目前，Avro Source 仅对 RPC 支持 zlib 压缩。为了使 Avro Source 能以压缩的形式接收数据，需要将 `compression-type` 参数设置为 `deflate`。如果这个参数没有设置或者设置为 `none`，Flume 将不会解压数据；这可能导致事件积压在前一阶段，因为 Source 将不能解析压缩数据，会给前一阶段返回错误，将导致该阶段一直重试。

39

Avro Source 和压缩

如果 `compression-type` 参数设置为 `deflate`，那么传入的数据必须被压缩，否则 Source 不能解析传入的数据。Sink 或 Flume 客户端发送数据时，必须配置为压缩将要发送的数据。因此，如果压缩和未压缩的数据被传输到相同的 Flume Agent，agent 应该能运行两种 Avro Source，一种用来接收压缩的数据，另一种用来接收未压缩的数据。

还有需要注意的是，当 Avro Sink 或 Flume RPC 客户端发送数据到 Avro Source 时，数据是分批被压缩的，而不是按照每个事件，因此这可能会提供更好的压缩率，减少压缩时内存的使用情况。

下面是一个配置了 SSL 和压缩的 Avro Source 的例子。如果要禁用 SSL，移除 `ssl` 参数即可，剩下的 SSL 相关的参数将会被忽略：

```
agent.sources = avroSrc
agent.channels = memChannel

agent.sources.avroSrc.type = avro
agent.sources.avroSrc.channels = memChannel

# Bind to all interfaces// 绑定到所有的接口
agent.sources.avroSrc.bind = 0.0.0.0
agent.sources.avroSrc.port = 4353

# Removing the next line will disable SSL
// 删除下一行将禁用 SSL
agent.sources.avroSrc.ssl = true
agent.sources.avroSrc.keystore = /tmp/keystore.jks
agent.sources.avroSrc.keystore-password = UsingFlume
agent.sources.avroSrc.keystore-type = jks
```

```
agent.sources.avroSrc.compression-type = deflate

# Initializes a memory channel with default configuration
// 使用默认配置初始化 memory channel
agent.channels.memChannel.type = memory
```

写入这个 source 的 Avro Sink 将会有如下类似的配置：

```
agent.channels = avroSinkChannel
agent.sinks = avroSink

agent.channels.avroSinkChannel.type = memory

agent.sinks.avroSink.type = avro
agent.sinks.avroSink.channel = memory
agent.sinks.avroSink.hostname = avrosrchost.example.com
agent.sinks.avroSink.port = 4353

# SSL properties
//SSL 属性
agent.sinks.avroSink.ssl = true
agent.sinks.avroSink.trust-all-certs = true
agent.sinks.avroSink.truststore = /path/to/keystore
agent.sinks.avroSink.truststore-password = UsingFlume
agent.sinks.avroSink.truststore-type = JKS

agent.sources.avroSink.compression-type = deflate
```


Thrift Source

在上一节的“Avro Source”中提及过，Flume 对 Avro 的 Java 特定 RPC 原理的使用，使得 Avro Source 不能接收非 JVM 语言的数据。随着 Flume 变得更受欢迎，这种情况亟待解决。因此，Apache Thrift RPC 的支持 [thrift_ch3] 被加入到 Flume 中。Thrift 是 Apache 软件基金会的顶级项目，它支持跨语言通信，这使它非常流行。Flume 中 Thrift Sink-Thrift Source 在 Flume 中的组合，设计目标是像 Avro Sink–Avro Source 的组合一样出色地工作。Flume 也有 Java Thrift RPC 客户端，是 Flume SDK 的一部分。用最简单的术语来说，Thrift Source 就是多线程、高性能的 Thrift 服务器。Flume 使用的 Thrift 接口定义语言（IDL）如下所示：

```
namespace java org.apache.flume.thrift

struct ThriftFlumeEvent {
  1: required map <string, string> headers,
```

```
  2: required binary body,
}

enum Status {
  OK,
  FAILED,
  ERROR,
  UNKNOWN
}

service ThriftSourceProtocol {
  Status append(1: ThriftFlumeEvent event),
  Status appendBatch(1: list<ThriftFlumeEvent> events),
}
```

IDL 可以用来以 Thrift 支持的任何语言生成 Thrift 客户端。生成的代码可以用来将数据发送到 Flume 的 Thrift Source。

Thrift Source 的配置非常简单，类似于 Avro Source（见表 3-4）。

41 表3-4 Thrift Source配置

配置参数	默认值	描述
type	-	Thrift Source 的别名是 Thrift。也可以使用 FQCN，org.apache.flume.source.ThriftSource（大小写敏感）
bind	-	绑定的 IP 地址 / 主机名。使用 0.0.0.0 绑定机器所有的接口
port	-	绑定的端口号
threads	-	Source 处理请求使用的最大工作线程的数量

bind 参数指明要绑定的接口的主机名或 IP 地址；使用 0.0.0.0 绑定所有的接口。port 参数指明使用的端口号——客户端将使用这个端口发送事件给 Source。这些是必须配置的参数。

Thrift Source 的 threads 参数与 Avro Source 相比，工作方式稍有不同。Flume（自 1.4.0 版本起）构建且包含 0.7.0 版本的 Thrift。这意味着支持客户端（用 Flume SDK 写的程序）以相同的程序写入到 0.92 版本的 HBase（或更早的版本）。如果对该 HBase 版本的支持没有要求，那么建议 Flume 附带的 Thrift 版本替换为更新的版本，尽管由于 Thrift-generated 代码不兼容，Flume 可能也需要对新版本进行重新编译。

当使用低于 0.8.0 版本的 Thrift，Flume 使用 Thrift 的 TThreadPoolServer，每个客户端连接使用一个线程，threads 参数控制 Source 创建的线程的最大数，从而间接地控制客

户端连接 Agent 的数量。在这种情况下不建议设置 threads 这个参数。如果使用更新版本的 Thrift，Flume 使用 Thrift 的 TThreadedSelectorServer，使用 Java 非阻塞式 I/O，因此可以支持比可用的线程更多的客户端。在这种情况下，threads 参数和 Avro Source 的 threads 参数工作类似，可以用来控制资源利用率。

Thrift Source 不同于 Avro Source，目前不支持压缩或 SSL。因此 Thrift Source 只能用来从非 JVM 语言系统推送数据到 Flume 中，或者从用于别的用途的 Thrift 的写数据的应用推送数据到 Flume 中。对于 Flume Agent 到 Flume Agent 的通信，推荐使用 Avro Sink-Avro Source 对。

下面是 Thrift Source 配置的例子：

```
agent.sources = thriftSrc
agent.channels = memChannel

agent.sources.thriftSrc.type = thrift
agent.sources.thriftSrc.channels = memChannel

# Bind to all interfaces
// 绑定到所有的接口
agent.sources.thriftSrc.bind = 0.0.0.0
agent.sources.thriftSrc.port = 4564

# Initializes a memory channel with default configuration
// 使用默认配置初始化 memory channel
agent.channels.memChannel.type = memory
```


写入这个 Source 的 Thrift Sink 将会有如下类似的配置：

```
agent.channels = memChannel
agent.sinks = thriftSink

agent.channels.memChannel.type = memory

agent.sinks.thriftSink.type = thrift
agent.sinks.thriftSink.channel = memory
agent.sinks.thriftSink.hostname = thriftsrchost.example.com
agent.sinks.thriftSink.port = 4564
```

RPC Sources 的失败处理

在 Avro Source 和 Thrift Source 中失败的处理都有些棘手。这是因为 RPC Source 尽管看起来像是本地方法的调用，但实际是被另一边网络链路上的客户端或 Sink 调用。由于参

数错误，RPC Source 都不会启动，例如绑定不到端口号，当 Source 尝试启动的时候会抛出异常。因为 Flume 的配置系统将每隔几秒重启组件，所以如果它的错误不再存在，配置成功的情况下，Source 将启动——例如，其他绑定到该端口的进程被杀或者释放端口。

更麻烦的部分是关于实际接收数据和往 Channel 写事件的代码。如果 Source 配置用来写入的其中一个 Channel 由于写满，会抛出 `ChannelException`，或这次事务量太大，Source 将会给客户端或 Sink 返回一个失败的状态，用来回调它并期望它重试。因为 RPC Source 通过线程池中的线程接收数据，线程异常只能导致线程终止。

在所有这些情况下，失败的真正提示只在这些记录异常的日志文件中。有时这些异常可能指明了一个大问题，例如在程序运行过程中耗尽资源（正如 `OutOfMemoryError`）。因此，监控 Flume Agent 生成的日志是很重要的，它能保证进程平稳运行。频繁抛出

`ChannelException` 意味着 Channel 分配远低于写入的速率，或者 Sink 清理 Channel 中数据的速度不够快。如果数量不足的 Sink 读取数据，可以增加 Sink 的数量，但是如果最终目的地本身不能处理该负载，则需要重新考虑能力。不管在什么样的情况下，错误可能导致程序重复执行但不会真的造成数据丢失，因为只有当数据真正成功写入下一阶段，事件才会从管道中移除。

HTTP Source

Flume 自带的 HTTP Source 可以通过 HTTP POST 接收事件。对于应用程序环境，它可能不能部署 Flume SDK 及它的依赖项，或客户端代码倾向于通过 HTTP 而不是 Flume 的 RPC 发送数据的情况，HTTP Source 可以用来将数据接收到 Flume 中。从客户端的角度来看，HTTP Source 表现得像 web 服务器一样能接收 Flume 事件。

如表 3-5 中展示的，HTTP Source 需要配置一些参数。这类 Source 的配置极其简单，还允许用户配置嵌入式的处理程序。

表3-5 HTTP Source配置

配置参数	默认值	描述
type	-	HTTP Source 的别名是 `http`，也可以使用 FQCN，`org.apache.flume.source.HttpSource`（大小写敏感）
bind	-	绑定的 IP 地址 / 主机名。使用 0.0.0.0 绑定机器所有的接口
port	-	绑定的端口号

续表

配置参数	默认值	描述
enableSSL	false	要启用 SSL，设置该参数值为 true
keystore	-	使用的 keystore 文件的路径
keystorePassword	-	能够进入 keystore 的密码
handler	JSONHandler	HTTP Source 使用的处理程序类的 FQCN，可以转换 HTTP 请求为 Flume 事件。见下一节“针对 HTTP Source 写处理程序 *”
handler.*	-	必须传给处理程序的任何参数可以通过使用 handler. 前缀在配置中传入

可以预料到的是，bind 和 port 参数定义 Source 绑定的接口和端口号，即客户端发送数据到的主机名和端口号。

44

为了安全传输 HTTP Source 也支持 SSL。默认情况下，Source 不启用 SSL。要启用 SSL，设置 enableSSL 参数为 true 即可。如果启用了 SSL，keystore 和 keystorePassword 参数是必需的。keystore 参数是使用 SSL 的 keystore 的完整路径，keystorePassword 是能使用 keystore 的密码。

例 3-3 展示了如何配置使用自定义处理程序的 HTTP Source 的示例。

例3-3 HTTP Source配置示例

```
agent.sources = httpSrc
agent.channels = memChannel

agent.sources.httpSrc.type = http
agent.sources.httpSrc.channels = memChannel

# Bind to all interfaces
// 绑定到所有的接口
agent.sources.httpSrc.bind = 0.0.0.0
agent.sources.httpSrc.port = 4353

# Removing this line will disable SSL
// 删除下一行将禁用 SSL
agent.sources.httpSrc.ssl = true
agent.sources.httpSrc.keystore = /tmp/keystore
agent.sources.httpSrc.keystore-password = UsingApacheFlume

agent.sources.httpSrc.handler = usingflume.ch03.HTTPSourceXMLHandler
```

```
agent.sources.httpSrc.handler.insertTimestamp = true

# Initializes a memory channel with default configuration
// 使用默认配置初始化 memory channel
agent.channels.memChannel.type = memory


What parameters the handler requires and how they are used by the handler
depends on the specific handler implementation. Please consult the handler' s
documentation for details on this.
```

处理程序需要什么参数，以及这些参数是如何被处理程序使用的，取决于特定的处理程序实现。详情请查阅处理程序的文档。

针对 HTTP Source 写处理程序 *

对用户来说，开发和嵌入处理程序（handler）来转换接收到的数据为 Flume 事件是很简单的。HTTP Source 可以接收从客户端发来的能被处理程序处理的任何格式的数据。HTTP Source 处理程序是一个继承自简单接口 HTTPSourceHandler 的类：

```
package org.apache.flume.source.http;
public interface HTTPSourceHandler extends Configurable {
  public List<Event> getEvents(HttpServletRequest request) throws
          HTTPBadRequestException, Exception;
}
```


处理程序的接口很简单，只有一个方法，getEvents，该方法能接收客户端发送的 HTTPServletRequest，并且返回 Flume 事件的列表。尽管处理程序接口很简单，但它基本能将 HTTPServletRequest 中的输入数据做任意处理从而转化成 Flume 事件。但是处理的总量应该有所限制，否则客户端给 Source 发送数据可能导致超时。处理程序通过 Flume 的标准配置机制进行配置。因为不管是什么处理程序，HTTP Source 都是一个请求，使用恰好一个事务，发送方必须谨慎只发送管道能支持的尽量多的事件。

处理程序负责确保传递给它的配置参数是有效的。HTTP Source 将在启动时实例化和配置处理程序。因为 HTTP Source 将处理程序抛出的异常传递给配置系统，所以处理程序必须校验参数并且在 configure 方法中应用该参数。通过上下文实例将参数传递给 configure 方法。Context 实例是简单的键值对，包含各种配置键和它们的值。如果传来的配置是不合法的或不能被成功应用，HTTP Source 将再次通过处理程序将异常抛给配置系统，这会反过来禁用 HTTP Source 并将它从 Agent 中移除。

当处理传入的数据时，HTTP Source 通过返回客户端失败状态来操作处理程序抛出的异

常。如果传入的数据是不规范的并且不能被转换为 Flume 事件，HTTP Source 期望处理程序抛出 HTTPBadRequestException 异常。这个操作必须是幂等的，并且处理程序每次针对相同的输入必须抛出相同的异常。如果处理程序抛出 HTTPBadRequestException 异常，HTTP Source 将返回 HTTP 错误代码 400 来通知客户端这个请求是不规范的。如果处理程序抛出其他的异常，Source 将返回 HTTP 错误代码 500 来通知客户端 HTTP Source 有内部错误。然后由客户来决定在这种情况下如何重试。如果其中一个 Source 正在写入的管道抛出一个管道异常，Source 将返回错误代码 503，以提示管道暂时满负荷，客户端应该稍后重试。

JAR 文件包含的处理程序(或处理程序的 *.class* 文件)和所有的依赖，应该通过第 8 章的“部署自定义代码”一节中讨论的 plugins.d 机制将它们添加到 Flume 的环境变量中。

如果配置中没有指定处理程序，HTTP Source 将使用与 Flume 绑定的处理程序，它能处理 JSON 格式的事件。每个事件可以包含包装为数组的几个事件，尽管 Source 写入的管道可能有限制的事务能力。处理程序接受 UTF-8、UTF-16 或 UTF-32 编码的 JSON 格式的数据，并且将它转换成一个列表的事件。事件的正文是原始 HTTP 请求序列化的字符集。处理程序接受的格式如下：

46

```
[{
    "headers" : {
       "event1Header1" : "event1Value1",
       "event1header2" : "event1Value2"
    },
    "body" : "This is the body of the first event."

},
{
    "headers" : {
       "event2Header1" : "event2Value1",
       "event2Header2" : "event2Value2"
    },
    "body" : "This is the body of the second event"
}]
```

在例 3-5 里展示的 HTTPSourceXMLHandler 是对 HTTP Source 适用的另一个处理程序的例子。处理程序将 XML 格式的数据转换为 Flume 事件。这类处理程序很简单，接受例 3-4 里展示的 XML 格式的数据。处理程序期望的格式很简单，只有处于 <events> 和 </events> 之间的数据才被处理。每个事件被期望处于 <events> 和 </events> 标签之间。唯一限制每个请求事件的数量的是 Source 写入的管道的事务处理能力。每个事件可以有一或多个节段，位于 <headers> 和 </headers> 标签间。每个 header 被用作 header 名字

的标签所表示；位于开放的 header 名字标签的和结束标签之间的值作为 header 的值。正文位于 <body> 和 </body> 标签之间。

例3-4 HTTPSourceXMLHandler期望的格式

```
<events>
  <!-- This can contain as many events
  as the channel can support in a transaction -->

  <event>
    <headers>
      <header1>value1</header1>
      <header2>value2</header2>
    </headers>

    <body>This is a test.
        This input should show up in an event.
    </body>
  </event>

  <event>
    <!-- There can be zero or more headers sections.
     They are merged together, so each header name
     must be unique even between sections. -->
    <headers>
      <event2Header1>event2Value1</event2Header1>
    </headers>

    <!-- Each event can have only one body -->

    <body>This is the 2nd event.</body>

    <headers>
      <event2Header2>event2Value2</event2Header2>
    </headers>
  </event>
</events>
```


处理程序将 XML 格式的事件解析为 Flume 事件并将它们返回给 HTTP Source，进而将它们写入 Channel。当解析事件的时候，处理程序要确认每个事件至少有一个 header 和一个 body；如果不是，处理程序会抛出 HTTPBadRequestException 异常来提醒客户端传入的数据是不规范的。可以配置将时间戳插入 Flume 事件的 header 中，这样 HDFS Sink 可以用来批处理事件。

例3-5 HTTP Source的XML处理程序

```
package usingflume.ch03;

/**
 * A handler for the HTTP Source that accepts XML-formatted data.
 * Each event can contain multiple header nodes,
 * but exactly one body node. If there is
 * more than one body tag, the first one in the event is picked up.
 */
public class HTTPSourceXMLHandler implements HTTPSourceHandler {

  private final String ROOT = "events";
  private final String EVENT_TAG = "event";
  private final String HEADERS_TAG = "headers";
  private final String BODY_TAG = "body";

  private final String CONF_INSERT_TIMESTAMP = "insertTimestamp";
  private final String TIMESTAMP_HEADER = "timestamp";

  private final DocumentBuilderFactory documentBuilderFactory
    = DocumentBuilderFactory.newInstance();

  private final ThreadLocal<DocumentBuilder> docBuilder
    = new ThreadLocal<DocumentBuilder>();

  private boolean insertTimestamp;

  @Override
  public List<Event> getEvents(HttpServletRequest
    httpServletRequest) throws HTTPBadRequestException, Exception {
    if (docBuilder.get() == null) {
      docBuilder.set(documentBuilderFactory.newDocumentBuilder());
    }
    Document doc;
    final List<Event> events;
    try {
      doc = docBuilder.get().parse(
        httpServletRequest.getInputStream());
      Element root = doc.getDocumentElement();
      root.normalize();

      // Verify that the root element is "events"
      Preconditions.checkState(
        ROOT.equalsIgnoreCase(root.getTagName()));
```

48

```
      NodeList nodes = root.getElementsByTagName(EVENT_TAG);
      int eventCount = nodes.getLength();
      events = new ArrayList<Event>(eventCount);
      for (int i = 0; i < eventCount; i++) {
        Element event = (Element) nodes.item(i);
        // Get all headers. If there are multiple header sections,
        // combine them.
        NodeList headerNodes
          = event.getElementsByTagName(HEADERS_TAG);
        Map<String, String> eventHeaders
          = new HashMap<String, String>();
        for (int j = 0; j < headerNodes.getLength(); j++) {
          Node headerNode = headerNodes.item(j);
          NodeList headers = headerNode.getChildNodes();
          for (int k = 0; k < headers.getLength(); k++) {
            Node header = headers.item(k);

            // Read only element nodes
            if (header.getNodeType() != Node.ELEMENT_NODE) {
              continue;
            }
            // Make sure a header is inserted only once,
            // else the event is malformed
            Preconditions.checkState(
              !eventHeaders.containsKey(header.getNodeName()),
              "Header expected only once " + header.getNodeName());
            eventHeaders.put(
              header.getNodeName(), header.getTextContent());
          }
        }
        Node body = event.getElementsByTagName(BODY_TAG).item(0);
        if (insertTimestamp) {
          eventHeaders.put(TIMESTAMP_HEADER, String.valueOf(System
            .currentTimeMillis()));
        }
        events.add(EventBuilder.withBody(
          body.getTextContent().getBytes(
            httpServletRequest.getCharacterEncoding()),
          eventHeaders));
      }
    } catch (SAXException ex) {
      throw new HTTPBadRequestException(
        "Request could not be parsed into valid XML", ex);
    } catch (Exception ex) {
      throw new HTTPBadRequestException(
```

49

```
        "Request is not in expected format. " +
          "Please refer documentation for expected format.", ex);
    }
    return events;
  }

  @Override
  public void configure(Context context) {
    insertTimestamp = context.getBoolean(CONF_INSERT_TIMESTAMP,
      false);
  }
}
```

例 3-3 展示了 XML 处理程序的配置示例。配置指示了处理程序为每个事件插入处理时的时间戳。通常，任意数量的参数都能以这种方式传递给 HTTP Source 处理程序。

HTTPHandler 接口是 flume-ng-core Maven 组件的一部分，可以在你的应用中的 *pom.xml* 的 deperdency 部分添加例 3-6。

例3-6 在你的应用中包含以下的flume-ng-core组件

```
    <dependency>
      <groupId>org.apache.flume</groupId>
      <artifactId>flume-ng-core</artifactId>
      <version>1.5.0</version>
</dependency>
```

使用第 8 章的“部署自定义代码”一节里展示的 *pluins.d* 框架给 Flume Agent 部署自定义的 HTTP 处理程序。

Spooling Directory Source

50

在很多场景中，应用程序生成被写入文件的数据。通常这些文件不是简单的文本文件，或者每一行转换为一个事件没有意义，而是若干行的组共同组成一个事件。一个例子就是堆栈跟踪。这通常很难或不可能修改这些应用，来使用 Flume Client API 直接发送数据给 Flume。在这种情况下，Flume 的 Spooling Directory Source 可以被用来从这些文件中读取数据。

Spooling Directory Source 监视读取事件的目录。尽管新文件可以被实时地添加到该目录，Source 还是期望目录中的文件是不变的。文件一旦被移入到该目录，它不应该被写入。如果你在处理日志文件，这样做的一个好方法是配置日志系统当日志滚动时移动文件。Source 也期望文件名是不重名的。如果这两种情况发生一种，Source 将会抛出异常并终止。这时重启 Source 的唯一方式是重启 Agent 自身。

Spooling Directory Source是使用tail -F的Exec Source（将稍后在本章讨论）的一种好的替换方案，因为这类Source能保证数据传送，通常比使用tail -F的Exec Source更加可靠。唯一的缺点是数据不是实时跟踪的，并且只能在文件关闭或移入到相关目录时才能读取文件。文件一旦被Source完全使用完且所有的事件被成功写入Source的Channel中，Source就可以基于配置重命名文件或删除文件。当文件被重命名，Source只是给文件名添加一个后缀，而不是完全改变它。这个后缀也是可以配置的。

Spooling Directory Source使用追踪器持久化到磁盘，以定位每个文件在哪个位置成功将事件写入Channel，这样如果Agent或机器失败和重启，Source就能从这个位置开始读取数据。这也能保证Source实时定位任一位置的处理，并且保证当Source重启时，处理是接着上次处理的位置。这正是Source不允许文件名重用的原因之一。

表3-6展示了Spooling Directory Source接受的各种配置参数。

表3-6 Spooling Directory Source配置

51

参数	默认值	描述
type	-	Spooling Directory Source的别名是spooldir，FQCN是org.apache.flume.source.SpoolDirectorySource
spoolDir	-	监控和读取文件的目录。该目录的子目录不会被扫描
batchSize	100	每个事件写入Channel的最大事件数量
ignorePattern	^$	文件名称匹配这个正则表达式被忽略和数据也不会从中读取
deletePolicy	never	如果要删除使用完的文件——该参数的值为never或immediate
fileSuffix	.COMPLETED	被使用完的文件使用的后缀。如果这是扩展“.”是必需的
fileHeader	false	如果设置为true，文件名会被添加到事件的header
fileHeaderKey	file	如果文件名被添加到header，此参数为在header中使用的秘钥
trackerDir	.flumespool	Spooling Directory Source存储元数据的目录，用来source中断时来重启Source
deserializer	line	Builder类的别名或完整类别名称，用来创建读取自定义格式数据的反序列化器。下节解释了如何写反序列化器
deserializer.*	-	任何传递给反序列化器的参数
inputCharset	UTF-8	当反序列化器调用readChar方法时使用的编码

Spooling Directory Source 的类型是 spooldir。在前面提及过，Source 读取指定目录中的所有文件并且逐个处理它们。处理目录的完整路径应该通过 spoolDir 参数来传递。由于性能的原因，Source 批次写事件，批处理的最大数量通过 batchSize 参数定义。Source 尝试尽可能多地从文件中读取事件，直到达到指定的批量大小。如果文件中有更少的可用事件，一旦文件中所有事件都读取完，它就尽快提交事务。

有时，写文件到相同的目录中，实际上可能不包含数据，如元数据文件。为了避免使用此类文件，即不包含有效数据的文件，可以通过使用 ignorePattern 参数指定 ignore 模式。 52
这个参数需要一个正则表达式，任何匹配正则表达式文件名的文件将被忽略。

前面提及过，文件一旦完全使用完，Flume 就能重命名或删除文件。如果要立即删除文件，设置 deletePolicy 参数为 immediate 即可。如果 deletePolicy 设置为 never（默认值），文件一旦使用完就重命名，用 fileSuffix 参数指定的后缀追加到原始文件名的后面。任何针对已完成的文件使用的该后缀将被忽略，所以要谨慎，不要使用这样的文件后缀，即写入该目录的新文件的后缀。

当文件被处理完并且从文件生成事件时，通常有利于处理系统知道事件来自哪些文件（例如，在搜索 UI 中展示属于堆栈跟踪的文件名）。通过设置 fileHeader 参数为 true 可以包含完整路径和文件名。使用 fileHeaderKey 参数可以设置 header 中使用的秘钥（默认值为 file）。

Spooling Directory Source 能够从它中断的位置恢复，所以能避免重复消耗文件中的数据。这使得当 Source 启动的时候，持久化文件处理和读取的信息到磁盘是可以实现的。信息持久化到追踪目录中，追踪目录一直在 Source 监控的目录中。追踪目录默认的名字是 .flumespool，这可以通过 trackerDir 参数来改变。要注意的是，目录内创建的子目录将会被读取，trackerDir 参数的值被用作 Source 监控目录的相对路径。一旦追踪目录的名字设置好，如果这个参数的值发生改变（甚至关闭 Flume 之后），Source 将不能再定位文件处理的位置，可能在开始的位置结束处理导致重复。所以，该参数一旦设置就不能修改。

例 3-7 展示了一个例子，关于使用 Spooling Directory Source 从目录中以每批次 250 个事件的速度读取数据到磁盘。当文件完全使用完的时候 Source 会尽快删除文件。通过 UsingFlumeFiles 的密钥也能向 header 插入文件名。

例3-7 Spooling Directory Source配置示例

```
agent.sources = spool
agent.channels = memChannel

agent.sources.spool.type = spooldir
```

```
agent.sources.spool.channels = memChannel

agent.sources.spool.spoolDir = /data/flume/spool
agent.sources.spool.batchSize = 250

agent.sources.spool.deletePolicy = immediate
agent.sources.spool.fileHeader = true
agent.sources.spool.fileHeaderKey = usingFlumeFiles
agent.sources.spool.deserializer = \
usingflume.ch03.ProtobufDeserializer$ProtobufDeserializerBuilder

agent.channels.memChannel.type = memory
agent.channels.memChannel.capacity = 10000
agent.channels.memChannel.transactionCapacity = 500
```


使用 Deserializers 读取自定义格式 *

Source 使用嵌入式的反序列化器转换目录中文件的数据，这允许 Source 以不同的方式从文件中读取数据到事件中。例如，一个“理解”Avro 的反序列化器能读取 Avro 格式的文件，并且转为每个 Avro 信息发送给 Flume 事件。否则，一次就要读取几行数据直到整个堆栈读取完并转换为 Flume 事件。文件中所有的数据一旦都读取完，Source 可以删除文件或者用新的扩展名重命名文件，这样相同的文件就不会被再次处理。

如果要使用自定义反序列化器，将 deserializer 参数的值设置为 EventDeserializer$Builder 的一个实现，EventDeserializer$Builder 可以构建 EventDeserializer 实现来使用。反序列化器可以通过传入参数 deserializer. 前缀进行配置。基于文本的反序列化器可以调用 readChar 方法读取字符。不同字符集的字符表示的方式不同。要告知 Source 用什么字符集，则设置 character 参数的值为输入字符集的名称，默认情况下是 UTF-8。

反序列化器实现 EventDeserializer 接口，还应该提供一个 Builder 类，它必须实现 EventDeserializer$Builder 接口。Builder 必须有一个无参数的公共构造函数，Flume 框架可以使用它来实例化构造器。Builder 的 build 方法必须创建并返回一个完全反序列化器的配置实例。

Context 实例和 ResettableInputStream 实例被传递给该方法。可以使用 Context 实例配置反序列化器。反序列化器将从输入流反序列化事件。ResettableInputStream 是一个从流读取数据的接口，而且还可以回滚到流以前的位置。ResettableInputStream 类的实例，调用 reset 方法时，不管上次 mark 方法调用之后，流使用 read 或 readChar 方法读取了多少字节的数据，能保证从上次 mark 方法调用发生时候的流的位置，重置从流

中读取的数据。如果写入 Channel 失败，并且事件不能被写入，流 Directory Source 就能重读事件。反序列化器可以使用 mark 和 reset 方法的这个功能，以确保它在能回滚到 stream 正确的位置。

54

反序列化器实现另外两个方法，即 Source 从流读取事件要调用的方法——readEvent 和 readEvents 方法。readEvent 方法必须从流中返回一个事件，而 readEvents 方法接受一个参数，这个参数是从流中读取事件的最大值。

例 3-9 展示了一个反序列化器，基于例 3-8 所示的格式，反序列化消息序列化为 Protocol Buffer (Protobuf) 消息。每个 Protobuf 先将它的 4 字节整数的长度写入文件，再将消息写入文件。

例3-8 ProtobufDeserializer使用的Protobuf格式

```
option java_package = "usingflume.ch03";
option java_outer_classname = "UsingFlumeEvent";

message Event {
  repeated Header header = 1;
  required bytes body = 2;
}

message Header {
  required string key = 1;
  required string val = 2;
}
```

例3-9 ProtobufDeserializer:反序列化写入的数据为Protobuf消息

```
package usingflume.ch03;

public class ProtobufDeserializer implements EventDeserializer {
  private final ResettableInputStream stream;
  private boolean isOpen;

  private ProtobufDeserializer(ResettableInputStream stream) {
    // No configuration to do, so ignore the context.
    this.stream = stream;
    isOpen = true;
  }

  @Override
  public Event readEvent() throws IOException {
    throwIfClosed();
    // To not create an array each time or copy arrays multiple times,
```

```
    // read the data to an array that backs byte buffers,
55  // then wrap that array in a stream and pass it to the Protobuf
    // parseDelimitedFrom method.
    // The format is expected to be:
    // <length of message> - int
    // <protobuf message (written using writeTo (not delimited)>
    // We assume here that the file is well-formed and the length
    // or the
    // message are not partially cut off.
    byte[] sz = new byte[4];
    if (stream.read(sz, 0, 4) != -1) {
      int length = ByteBuffer.wrap(sz).getInt();
      byte[] data = new byte[length];
      stream.read(data, 0, data.length);
      UsingFlumeEvent.Event protoEvent =
        UsingFlumeEvent.Event.parseFrom(new ByteArrayInputStream(data));
      List<UsingFlumeEvent.Header> headerList
        = protoEvent.getHeaderList();
      Map<String, String> headers = new HashMap<String, String>(
        headerList.size());
      for (UsingFlumeEvent.Header hdr : headerList) {
        headers.put(hdr.getKey(), hdr.getKey());
      }
      return EventBuilder.withBody(protoEvent.getBody().toByteArray(), headers);
    }
    return null;
  }

  @Override
  public List<Event> readEvents(int count) throws IOException {
    throwIfClosed();
    List<Event> events = new ArrayList<Event>(count);
    for (int i = 0; i < count; i++) {
      Event e = readEvent();
      if (e == null) {
        break;
      }
      events.add(e);
    }
    return events;
  }

  @Override
  public void mark() throws IOException {
    throwIfClosed();
```

```
    stream.mark();
  }

  @Override
  public void reset() throws IOException {
    throwIfClosed();
    stream.reset();
  }

  @Override
  public void close() throws IOException {
    isOpen = false;
    stream.close();
  }

  private void throwIfClosed() {
    Preconditions.checkState(isOpen, "Serializer is closed!");
  }

  public static class ProtobufDeserializerBuilder implements Builder {

    @Override
    public EventDeserializer build(Context context,
      ResettableInputStream resettableInputStream) {
      // The serializer does not need any configuration,
      // so ignore the Context instance. If some configuration has
      // to be passed to the serializer, this Context instance can be used.
      return new ProtobufDeserializer(resettableInputStream);
    }
  }
}
```


ProtobufDeserializer 类从文件中读取 Protobuf-serialized 事件，并在 readEvent 方法中将它们转换为 Flume 事件，返回为 null 时表示没有可用的事件读取。如果 readEvents 方法没有事件可以读取，将返回一个空列表，就如 EventDeserializer 接口强制要求的那样。

由于文件是不可变的，一旦我们达到一个阶段，即文件中没有更多的可用的事件，这意味着所有的事件已经从文件读取完，此时 Source 通过调用 close 方法关闭反序列化器。如果序列化器保持任何内部状态或有一些清理工作要做，这种方法有望做到这一点。在这种情况下，我们只是简单地关闭流。mark 和 reset 方法只是检查反序列化器是否打开和转发调用方法给流。序列化器特定的实现不需要任何配置，但反序列化器可以通过

Context 实例接收配置传递给 builder，进而可以通过构造函数传递到反序列化器实例。例 3-7 展示了 Spooling Directory Source 使用 ProtobufDeserializer 进行配置。

Flume 自带了一部分反序列化器。默认的反序列化器是 LineDeserializer[line-deserializer]。这是反序列化器的一个示例配置。如果 Spooling Directory Source 没有设置反序列化器或 deserializer 参数的值设置为 line，行反序列化器将启用。行反序列化器逐行读取文件，基于一个可配置的字符集（默认 UTF - 8）将每一行转换为一个事件。表 3-7 列出了配置参数。

表3-7 行反序列化器配置

参数	默认值	描述
outputCharset	UTF-8	转换从文件中读取的字符为字节数组要使用的字符集，这是为事件主体设置的
maxLineLength	2048	每行返回的最大字符数。如果行比这个参数大，则截断来返回

另一个与 Flume 自带的反序列化器是 AvroEventDeserializer。要使用这个反序列化器，设置 deserializer 参数的值为 avro。Avro 反序列化器可以读取 Avro 容器文件并且发送 Avro 格式事件的数据。这个反序列化器只有一个配置参数，如表 3-8 中描述。

表3-8 Avro反序列化器配置

参数	默认值	描述
schemaType	flume.avro.schema.hash	该项参数可以设置为 flume.avro.schema.hash 或 flume.avro.schema.literal。设置为 flume.avro.schema.hash 将导致模式的 64 位 Rabin 指纹使用 flume.avro.schema.hash 的关键词插入 header。如果该参数的值设置为 flume.avro.schema.literal，整个 JSON 化模式将使用 flume.avro.schema.literal 关键词插入到 header

解释数据，重要的是要知道所使用的模式。虽然这些信息包含在文件本身中，保持每个事件的模式也是很重要的，这样可以从事件中读取数据。反序列化器支持在 header 插入 Avro 模式，或简单地将该模式的 64 位 Rabin 指纹（正如 Avro 规范 [schema-fp] 中所指定的）插入 header，这在后面可以用来查找被模式指纹索引的模式注册表（Avro 格式的数据将存入几个已知的模式之一，这是最常见的场景。在 header 中使用指纹允许用户指明读取信息时应该使用的模式，这样当写数据的时候就能被反序列化器所使用）。要想在事件的 header 中设置指纹，则设置 schemaType 参数的值为 flume.avro.schema.hash。

使用 flume.avro.schema.hash 的关键词将模式指纹插入到 header 中。

如果整个 JSON 化模式应该写入每个事件，设置 schemaType 参数的值为 flume.avro.schema.literal。在这种情况下，整个模式使用关键 flume.avro.schema.literal 来写。在每一个事件的 header 中写模式是非常低效的，因为它增加了事件的大小，特别是如果存在数量有限的模式类型。 <58>

要从目录中读取文件作为二进制大对象 (BLOB)[blob]，则要使用 blob 反序列化器。blob 反序列化器的 FQCN 是 org.apache.flume.sink.solr.morphline.BlobDeserializer。这类反序列化器有 blob 能接受的最大大小，每个文件会试图以尽可能多的字节从 blob 中文件读取数据。这类反序列化器只有一个配置参数，maxBlobLength，是每个 blob 以字节为单位的最大容量。如果文件比该值大，就要分为几个 blob，每个 blob 的大小应小于或等于配置的最大值。反序列化器在内存中分批缓存所有的 blob，所以批量的大小和 blob 容量的最大值应该要配置，来确保反序列化器不会比预期使用更多内存导致终止。

因为 Spooling Directory Source 是 flume-ng-core 工件的一部分，确保你在序列化器的 pom.xml 文件中添加了 flume-ng-core 的工件，在例 3-6 中提及过。自定义反序列化器可以通过使用第 8 章“部署自定义代码”一节中提到的 *plugins.d* 框架部署到 Flume Agent。

Spooling Directory Source 性能

Spooling Directory Source 是 I/O 密集型的。为了避免复杂的反序列化器实现，Source 被专门设计成单线程的。这意味着有可能通过使用多个线程读取数据、更多地使用可用的 CPU 来提高性能。提高文件读取的性能的一种方法是轮流写文件到不同的目录，并有一个 Spooling Directory Source 处理每一个目录 (如果所有数据传到相同的目的地，则写入到相同的 Channel)。这意味着更多从磁盘读取数据的线程和更多的可以用来反序列化的 CPU。

Syslog Source

Syslog 是很多应用写日志消息的众所周知的格式。集成了 syslog 的 Flume 可以接收 TCP 和 UDP 消息。Flume 提供了两种 syslog Source: Syslog UDP Source 和 Multiport Syslog Source。Syslog UDP Source 用 UDP 接收 syslog 消息，而 Multiport Syslog Source 可以在多个端口用 TCP 接收 syslog 消息。这两种 Source 都能解析 syslog 消息，抽取几个字段到 Flume 事件的 header，可以用 HDF Sink 分桶。如果 syslog 消息不符合 Syslog RFCs，RFC-3164 或 RFC-5424，事件将包含一个带有值为 Invalid flume.syslog.status 的 header。 <59>

Syslog UDP Source 认为整个 UDP 数据报文是一个 syslog 事件，并将其转换为一个 Flume 事件，而 Multiport Syslog Source 每次遇到一个换行符（\n）字符创建一个新的消息。这些 Source 创建两个 header，Facility 和 Severity，在每个 Flume 事件的 header 中指明每个消息的 Facility 和 Severity。这可以用于分桶或多路复用 Channel 选择器（第 6 章中讨论）。

表 3-9 列出了两种 Source 的公共配置参数。

表3-9 Syslog Source配置

参数	默认值	描述
type	-	Syslog UDP Source 类型是 syslogudp，Multiport Syslog Source 类型是 multiport_syslogtcp
host	-	绑定的 IP 地址或主机名。使用 0.0.0.0 绑定机器上所有的接口
keepFields	false	如果设置为 true，syslog 消息的所有字段会保存在事件除了 header 之外的 body

Syslog UDP Source 可以使用 syslogudp 别名，而 Multiport Syslog Source 可以使用 multiport_syslogtcp 别名。两类 Source 都需要用户使用 host 参数的值来指定绑定的主机名。如果要绑定机器上所有的端口，将 host 参数的值设置为 0.0.0.0。如果 keepFields 参数设置为 true，syslog 消息的字段通常会移动到事件的 header（或整个移除），例如 Priority、Timestamp 和 Hostname 字段会保留在事件 body，同时也复制到事件的 header。Multiport Syslog Source 也允许 syslog 消息将每个端口接收的数据编码为不同的字符集。本质上这能使单类 Source 接收不同种类的 Source 的数据，并将消息编码为不同的字符集。

除了公共参数，Syslog UDP Source 只有一个另外的参数（见表 3-10）。

表3-10 Syslog UDP Source配置

参数	默认值	描述
port	-	绑定的端口号

Port 参数用来指定 Source 应该绑定的端口号。

60 Multiport Syslog Source 可以在主机上绑定多个端口。除了公共参数，Multiport Syslog Source 还定义了表 3-11 中的参数。

表3-11 Multiport Syslog Source配置

参数	默认值	描述
ports	-	以空格分隔的要绑定的端口列表
portHeader	-	接收事件的端口包含在 header 中，作为这个 header 指定的键的值。如果该参数没有设置，端口信息不包括在内
charset.default	UTF-8	设置要使用的默认的字符
charset.port.<port>	-	为特定端口设置字符
eventSize	2500	一个事件以字节为单位的最大容量
batchSize	100	写入 Channel 之前在内存缓冲的事件的数量
readBufferSize	1024	底层服务器使用的缓冲区大小
numProcessors	-	使用的处理器的数量。这使得 Source 能提高并行的程度

Multiport Syslog Source 可以接收多个端口的数据，这些端口要以空格分隔罗列作为 ports 参数的值。每个接收到的事件可以带上端口号注释，表明事件是从哪个端口号接收的。配置参数 portHeader 的值用作 Flume 的事件中 header 的键，该 header 的值是端口号。

Source 还允许 syslog 消息将每个端口接收的数据编码为不同的字符集。为每个端口配置字符集，可以使用 charset.port. 前缀后面跟上端口号作为配置参数，值为字符集的名字。使用 charset.default 参数来设置默认的字符。当一个端口没有指定字符集则使用这个参数的值。

前面提及过，Source 假设每个事件是由一个换行符分隔的。有时，重要的是每个事件并不是固定大小的。每个事件最大的大小可以使用 eventSize 参数设置，以字节表示。如果事件的大小大于这个值，事件就被截断为该值的长度，并将 flume.syslog.status 的键的值设为 Incomplete 插入到 header。Multiport Syslog Source 还在内存中缓冲事件，以避免影响 Channel 的性能。批处理大小可以使用 batchSize 参数指定。

61

Source 使用一个名为 *Apache MINA* 的框架来接收消息。MINA 服务器使用一个内部缓冲区读取网络中的数据。这个缓冲区的大小是可配置的，通过配置参数 readBufferSize 指定字节数。MINA 对并行性支持良好。配置并行的程度，可以通过使用 numProcessors 参数设置使用的处理器的数量。如果没有定义，使用的处理器数量的值是自动检测来的。对于每个可以使用的处理器，MINA 将产生两个线程。如果要减少使用的线程（不经常需要），则减小 numProcessors 指定的值。

例 3-10 展示了 Multiport Syslog Source 的示例，配置了所有的接口和 4353、4565、4553 三个端口来接收数据。Source 被配置为将 4565 端口接收到的数据编码为 ISO-8859-1 字符集，而其他两个端口接收到的所有数据都假设编码为 UTF-8 格式的字符集。Source 还在每个 header 中增加了关键端口，接收事件的端口号作为值。Source 每次批量写 1000 个事件，最大事件大小是 1092 字节。接收数据时 Source 还配置了 MINA 来使用 2048 字节的缓冲区。

例3-10 Multiport Syslog Source配置示例

```
agent.sources = syslog
agent.channels = memChannel

agent.sources.syslog.type = multiport_syslogtcp
agent.sources.syslog.channels = memory

# Bind to all interfaces
agent.sources.syslog.host = 0.0.0.0
agent.sources.syslog.ports = 4353 4565 4553

agent.sources.syslog.charset.default = UTF-8
agent.sources.syslog.charset.port.4565 = ISO-8859-1

agent.sources.syslog.batchSize = 1000
agent.sources.syslog.portHeader = port
agent.sources.syslog.readBufferSize = 2048
agent.sources.syslog.eventSize = 1092

agent.channels.memChannel.type = memory
agent.channels.memChannel.capacity = 10000
agent.channels.memChannel.transactioncapacity = 5000
```

Syslog 数据丢失

Syslog 通常被认为是一个“fire and forget”协议。RFCs 没有定义从接收方到发送方应答的方法，也没有指定超时后重新发送消息的方法。因此，如果 Flume Source 无法往 Channel 写事件，或者网络中断导致信息丢失（特别是在 UDP 的情况下），那么对于 Flume 没有实际的方式通知发送方的发送失败，或者告知发送方有错误情况发生并且重新发送消息。这实际上将导致数据无声地丢失，且没有恢复丢失数据的可能性。这导致 Flume 的 no-data-loss 保证对 syslog 不适用，因此，如果没有其他选择，并且 Flume RPC 客户机或嵌入式 Agent 不能使用，才建议使用 syslog。

Exec Source

Exec Source 执行用户配置的命令，且基于命令的标准输出来生成事件。它还可以从命令中读取错误流，将事件转换为 Flume 事件，并将它们写入 Channel。Source 希望命令不断生产数据，并且吸收其输出和错误流。只要命令开始运行，Source 就要不停地运行和处理，不断读取处理的输出流 (如果进行了配置，还有错误流)。

接着，输出流中的每一行将被编码为字节数组。要使用的编码是可配置的，默认为 UTF-8。然后每个字节数组用作 Flume 事件的 body。由于性能原因，Source 批量预配置事件的数量 (或直到超时)，并把它们写入 Channel。如果 Channel 已满 ,Source 可以配置成停止读取流程的输出和错误流 (因此暂时阻塞流程)，或放弃当前的批处理，继续读取输出和错误流 (因此可以让处理继续)。表 3-12 举例说明了 Exec Source 的配置参数。

表3-12 Exec Source配置

配置参数	默认值	描述
type	-	Exec Source 的别名是 exec，FQCN 是 org.apache.flume.source.ExecSource（大小写敏感）
command	-	Source 运行的命令
restart	false	如果设置为 true，当流程死亡时 Source 将重启流程
restartThrottle	10000	重启流程前要等待的毫秒值时间。如果 restart 参数值为 false 或没有设置，则该参数无影响
logStdErr	false	如果设置为 true，流程的错误流也会被读取并转换为 Flume 事件
batchSize	20	配置的在写入 Channel 之前事件的最大数量
batchTimeout	3000	配置的在将缓存事件写入 Channel 前要等待的毫秒值时间
charset	UTF-8	编码输出和错误流为 Flume 事件要使用的字符集
shell	-	用于运行该命令的 shell 或命令处理器

通过 command 参数传递要运行的命令。通过设置 restart 参数为 true，Exec Source 可以配置为随着命令的开始重启过程。为了确保有足够的时间区别命令的执行，可以设置 restartThrottle 参数。这个过程一旦死亡，Source 将在命令再次运行之前等待这么长的时间间隔。

当与 File Channel 同时使用时，Exec Source 通过缓存数据来确保好的性能。在第 4 章“file-channel”一节中会提到，当每个事务的事件数量很大时，Channel 有更好的性能。Exec Source 中的 batchSize 参数控制了批处理的大小。Source 也能配置在配置的

时间段结束时写入缓存的事件，这可以通过 batchTimeout 参数设置。如果 batchSize 和 batchTimeout 参数都设置好了，一旦达到批处理大小或批处理超时，就批量写入到 Channel 中。

Exec Source 可以设置为在一个单独的 shell 中运行配置命令，这可能不同于 Flume 处理所运行的方式。要在不同的 shell 中运行该处理，传递可执行 shell 的完整路径给 shell 参数。如果命令需要 shell 突出通配符或管道的替换功能，那么 shell 参数必须设置，因为 Flume 不会执行替换。

64 下面是 Exec Source 配置文件的示例，使用 shell 参数来运行一个完整的命令：

```
agent.sources = execSrc
agent.channels = memChannel

agent.sources.execSrc.type = exec
agent.sources.execSrc.shell = /bin/bash -c
agent.sources.execSrc.command = tail -F /var/log/flume/flume.log | grep
"error:"

agent.sources.execSrc.channels = memChannel

# Initializes a memory channel with default configuration
// 使用默认配置初始化 memory channel
agent.channels.memChannel.type = memory
```

Exec Source 丢失数据的可能性

Exec Source 是异步 Source 的一个例子，即如果有失败可能通知不到数据生产者。因此，重新启动 Agent 或机器会导致数据丢失，同上面解释的一样。

Exec Source 在 Flume 中最常用来追踪文件。利用 tail -F 命令使用 Exec Source 追踪文件，将近乎实时地将数据放入 Flume，但存在数据丢失的风险。如果 Flume Agent 死亡或机器重启，Exec Source 将在它启动的时候运行该命令；在这种情况下，它将运行 tail -F <file_name> 命令。因为 tail 命令只会拉取新数据写入到文件中的新数据，任何在 Agent 死亡和 Source 启动期间的写入文件的数据都会丢失。由于这个原因，建议使用本章前面所讨论的 Spooling Directory Source 处理写入文件的数据。尽管更严格一些，但因为它追踪的是正在从文件中读取的数据，所以 Source 不会丢失数据。

即使使用其他一些命令，Exec Source 在将事件写入 Channel 之前，也能缓冲和 batch 大小尽可能多的事件。如果 Agent 或机器重启，这些事件也可能会在批处理超时或达到 batch 最大容量之前丢失。

JMS Source

Flume 自带的 Source 可以获取来自 Java 消息服务队列或主题的数据。使用 JMS Source，可以接受支持 JMS 的消息传递系统中的消息，例如 ActiveMQ。安装和使用 JMS Source 比起大多数其他 Source 有点棘手。因为它有可能集成了支持 JMS 的多种消息传递系统，各自的系统提供的客户端库必须都安装到 *plugins.d* 目录，这在第 8 章“部署自定义代码”一节中会讨论。

表 3-13 中列出了 JMS Source 的配置参数。

表3-13 JMS Source配置

配置参数	默认值	描述
type	-	JMS Source 的别名是 jms，FQCN 是 `org.apache.flume.source.jms.JMSSource`（大小写敏感）
initialContextFactory	-	供应商初始上下文工厂的类名。例如 ActiveMQ，将会是 `org.apache.activemq.jndi.ActiveMQInitialContextFactory`
destinationName	-	JMS 消耗信息的目的地的名称
destinationType	-	JMS 目的地、队列、主题的类型
providerURL	-	JMS broker 的 URL
connectionFactory	ConnectionFactory	连接工厂呈现出的 JNDI 名称
messageSelector	-	如果需要，该项为过滤消息的消息选择器类的 FQCN
userName	-	登录到 JMS 提供者的用户名
passwordFile	-	包含到 JMS 提供者密码的文件
batchSize	100	每个事务写入 Channel 的事件的数量
converter.type	DEFAULT	实现 JMSMessageConverter 类的 FQCN。详情见下一节介绍
converter.charset	UTF-8	默认转换器使用的字符集。该项参数可能不会被其他转换器接受
converter.*	-	其他传递给转换器的配置参数

供应商提供的与 JMS broker 通信的库必须放入 Flume 的 *plugins.d* 目录。库的初始上下文工厂和连接工厂类应该作为值传递给 `initialContextFactory` 和 `connectionFactory` 配置参数。供应商提供的文档库中文件的信息应该是可用的。`connectionFactory` 参数的默认值是 `ConnectionFactory`，这应该能为类似 ActiveMQ 等流行的 JMS 系统工作。其他 JMS 相关参数，如目的地名称、目的地类型和消息选择器（如果消息需要过滤），也应该通过配置文件进行传递。Source 还需要 JMS broker 的 URL 作为 `providerURL` 参数的值。

Source 可以使用 `userName` 指定的用户名和文件中指定的密码登录到安全的 JMSbroker，文件的路径是由 `passwordFile` 参数指定的。

默认情况下，每次 JMS 会话 Source 将从 broker 中拉取多达 100 个消息，但这可以使用 `batchSize` 参数调整。

JMS Source 可以通过转换器（converter）将数据从 JMS 格式的消息转换为 Flume 事件。如果在配置中没有指定转换器，则使用默认的转换器处理大多数标准的 JMS 消息。如果使用的是自定义或专用的格式，则可以使用 `converter.type` 参数部署自定义转换器，值为转换器类的 FQCN。任何需要传递给转换器的参数可以使用 `converter.` 前缀进行传递。例如，默认的转换器要使用非默认 UTF-8 格式的字符集，需要设置 `converter.charset` 的值为字符集的标准名称，例如 `ISO-8859-1`。下一节详细讨论如何编写自定义转换器。例 3-11 展示了 JMS Source 配置的示例。

例3-11 JMS Source配置示例

```
agent.sources = jmsSrc
agent.channels = memChannel

agent.sources.jmsSrc.type = jms
agent.sources.jmsSrc.channels = memory

# Bind to all interfaces
// 绑定到所有的接口
agent.sources.jmsSrc.initialContextFactory = \
org.apache.activemq.jndi.ActiveMQInitialContextFactory
agent.sources.jmsSrc.destinationName = UsingFlume

agent.sources.jmsSrc.charset.providerURL = tcp://usingflume.oreilly.com:61616
agent.sources.jmsSrc.destinationType = QUEUE

agent.sources.jmsSrc.batchSize = 1000
agent.sources.jmsSrc.converter.type = usingflume.ch03.JsonMessageConverter
agent.sources.jmsSrc.converter.charset = iso-8859-1
```

```
agent.channels.memChannel.type = memory
agent.channels.memChannel.capacity = 10000
agent.channels.memChannel.transactioncapacity = 5000
```

配置文件配置了 Source 从 ActiveMQ 消息队列拉取数据，ActiveMQ 消息队列在主机 usingflume.oreilly.com:61616 读取 JMS 队列。队列信息通过很多参数来传递。对于每个 ActiveMQ 文件，初始化上下文工厂设置为 `org.apache.activemq.jndi.ActiveMQInitialContextFactory`，`JsonMessageConverter` 使用的是 ISO-8859-1 字符集。

67

转换 JMS 消息为 Flume 事件 *

JMS Source 可以配置为使用自定义代码将 JMS 消息转换成 Flume 事件，就像 HTTP Source 的处理程序。这使得 JMS Source 极其灵活，允许用户将 JMS 消息解析为专有格式的数据。与 Flume 绑定在一起的是 JMS 消息转换器（JMS Message Converter），支持默认 JMS 格式。JMS Source 可以传递配置到消息转换器，就像 HTTP Source。这个配置可以用来设置任何初始配置或转换器所需的状态。Source 处理所有的 JMS 事务；转换器不需要任何担心。

与 Flume 一起绑定的默认 JMS 消息转换器，基于 JMS 消息的格式和内容，分别将它们转换成 Flume 事件。`JMS ByteMessage` 只是读取，从消息中读取的字节被放入 Flume 事件的 body。对于 `JMS TextMessage`，转换器编码文本为字节数组并把它放入 Flume 事件的 body。通过 Source 配置，要使用的编码是可配置的。如果 `ObjectMessage` 从 JMS 队列读出，转换器在 `ObjectOutputStream` 封装了对象并且写到 `ByteArrayOutputStream`。然后会从流中读取到字节作为 Flume 事件的 body。在大多数情况下，这就是用户所需要的内容。

有时候，从消息中解析出为任何模式，并将其转换为一种可以在终端 Sink 更容易被序列化的格式，这也许更有意义。例如，如果有几个不同的应用程序以不同的序列化格式（例如 JSON 或 XML）写入 JMS 队列，那么可以编写一个转换器将这些转换成统一的格式，这样可以更容易在终端 Sink 被解析。

转换器必须实现 `JMSMessageConverter` 接口，在例 3-12 中有提及。JMS Source 实例化 `JMSMessageConverter$Builder` 的类，然后通过一个 Context 实例传递配置给 Builder 类的 `build` 方法，这个方法会返回一个完全配置好的转换器实例。

68

例3-12 JMS转换器接口

```
package org.apache.flume.source.jms;
public interface JMSMessageConverter {
  public List<Event> convert(Message message)
```

```
        throws JMSException;
  /**
   * Implementors of JMSMessageConverter must either provide
   * a suitable builder or implement the Configurable interface.
   */
  public interface Builder {
    public JMSMessageConverter build(Context context);
  }
}
```

例 3-13 列出了 MS Message Converter 的一个示例，用来从读取格式为 JSON 的 JMS 消息并转换成 Flume 事件。转换器可以实现 Configurable 接口来接受配置，JMS Source 通过 converter. 前缀传递任何指定的配置给转换器。转换器接收了配置就可以知道转换 JSON 事件为 Flume 事件是使用什么字符集的。

例3-13 JMS Message Converter的示例

```
package usingflume.ch03;

public class JsonMessageConverter implements JMSMessageConverter,
  Configurable {

  private static final Logger LOGGER =
    LoggerFactory.getLogger(JsonMessageConverter.class);
  private final Type listType
    = new TypeToken<List<JSONEvent>>() {
  }.getType();
  private final Gson gson
    = new GsonBuilder().disableHtmlEscaping().create();
  private String charset = "UTF-8";

  @Override
  public List<Event> convert(javax.jms.Message message)
    throws JMSException {

    Preconditions.checkState(message instanceof TextMessage,
      "Expected a text message, but the message received " +
        "was not Text");
    List<JSONEvent> events =
      gson.fromJson(((TextMessage) message).getText(), listType);
    return convertToNormalEvents(events);
  }

69  private List<Event> convertToNormalEvents(List<JSONEvent> events) {
    List<Event> newEvents = new ArrayList<Event>(events.size());
    for(JSONEvent e : events) {
```

```
      e.setCharset(charset);
      newEvents.add(EventBuilder.withBody(e.getBody(),
        e.getHeaders()));
    }
    return newEvents;
  }

  @Override
  public void configure(Context context) {
    try {
    charset = context.getString("charset", "UTF-8");
    } catch (Exception ex) {
      LOGGER.warn("Charset not found. Using UTF-8 instead", ex);
      charset = "UTF-8";
    }

  }
}
```

例 3-13 中展示的转换器期望是 JMS TextMessage 的消息。该消息被期望是 JSON 格式转换为一系列 Flume 事件。一旦 JSON 事件被创建，一系列“simple”的 Flume 事件也随之创建，以避免每次读取的时候将字符串转换为字节数组产生的额外开销。要为 JMS Source 配置转换器，可以使用例 3-11。

要部署一个自定义的转换器，首先要确保 *plugins.d* 目录包含这个类的 JAR 文件，然后要使用 converter.type 参数指定类的 FQCN。当写你的反序列化器时，要在你应用程序的 *pom.xml* 文件中包含 flume-jms-source 工件。

```
        <dependency>
          <groupId>org.apache.flume.flume-ng-sinks</groupId>
          <artifactId>flume-hdfs-sink</artifactId>
          <version>1.5.0</version>
    </dependency>
```

自定义的转换器可以像第 8 章“部署自定义代码”一节中叙述的一样安装到 *plugins.d* 目录。

编写自定义 Source*

Source 是数据进入到 Flume Agent 的要点。用户很可能会需要使用定制的或专有的通信格式用于写数据到 Flume。通过 Flume SDK 推送数据通常是更高效、更容易的。如果要与其他系统集成，用户可以编写自己的 Flume Source，并使用 Flume 的 *plugins.d* 机制部署它们。

70

每个 Source“生成”事件，并为了 Source 转发事件到 Channel 处理器中。Source 每次生成一个事件，调用 Channel 处理器的 processEvent 方法将事件写入 Channel 处理器，或者使用 Channel 处理器的 sprocessEventBatch 方法来发送事件。使用 processEventBatch 方法来处理一批事件总是更好的。每次 processEventBatch 调用启动一个 Channel 事务，每次事务写入整个批次再提交。另一方面，processEvent 只为一个事务创建事务，这可能会导致严重的开销，影响 Channel 的性能。这就是为什么建议使 Source 使用 processEventBatch 方法，除非已知每个事务都很大（近似数百千字节到几兆字节）。要访问 Source 的 Channel 处理器，Source 可以调用 AbstractSource 类中定义的 getChannelProcessor 方法。

如果来自外部 Source 的数据需要认证后被发送，重要的是这只在 processEventBatch 方法返回后被发送。如果对 channel 的提交失败，我们必须通知原始数据源，需要再次发送数据。在 processEventBatch 方法返回后发送 ACK 可以避免这个问题。这个方法抛出的 ChannelExceptions 可以被捕获，且失败可以报告给数据源，所以数据就可以重新发送。

通过 Flume 框架每个 Source 的 Channel 处理器被创建和启用，所以 Source 不需要处理 Channel 处理器的创建或配置工作。在这一部分，我们将了解如何写自定义 Source。

这部分所有被描述为依赖的类都是 flume-ng-core 工件或它的依赖的一部分。例 3-6 描述了如何在你插件的 *pom.xml* 文件中包含这个工件。就像第 8 章“部署自定义代码”一节中展示的，自定义 Source 可以像其他插件一样被部署到 Flume。

71

Event-Driven Source 和 Pollable Source

每个 Source 在称为 SourceRunner 的自身线程上运行。Source 运行器运行单独的线程来操作 Source。Flume 有两种类型的 Source：*Event-driven Source* 和 *Pollable Source*。在 Source 类型的基础上，Flume 框架会创建 EventDrivenSourceRunner 或 PollableSourceRunner 来运行 Source。

开发 Pollable Source

Pollable Source 不运行它们自己的线程；它们反而受 Flume 框架的控制，即 Flume 框架会循环调用 Source 的 process 方法。这些 Source 继承了 AbstractPollableSource 类并实现了 process 方法。Pollable Source 可以通过 Flume 配置系统接收用户的配置，这种功能除了要继承 AbstractPollableSource 类还要实现 Configurable 接口。

Pollable Source 运行一个循环来生成数据或轮询外部系统来接收数据，而不是运行一个服务器。一旦配置提供者实例化并且配置了 Pollable Source，Flume 框架会创建一个

PollableSourceRunner 来运行 Source。

Flume 框架通过重复调用 process 方法，为每个 Pollable Source 运行一个线程。每次调用 process 方法，Source“生成”事件并将它们传递到 Channel 处理器。Source 负责通知框架是否能够成功生成数据。如果 Source 能够成功生成事件，就会返回 PollableSource.Status.READY 给调用它的运行线程，该线程将立即再次调用 process 方法。否则，Source 返回 PollableSource.Status.BACKOFF。在这种情况下，Flume 框架只能在短暂超时后利用被调用的 process 方法发起一个补救措施，即每次 Source 返回失败时增加 1/2 的超时时间（最大超时是 5 秒）。Pollable Source 将生成自己的数据，或者轮询其他的 Source。

如果 processEventBatch 方法抛出异常，Source 可以捕获异常然后报告错误给恢复数据的系统。对于 JMS Source，这可能会导致 JMS 事务回滚。否则，成功会被报告给外部系统，例如 JMS 事务的提交。

很容易就可以自己编写 Pollable Source。例 3-14 展示了 Pollable Source 的例子。StockTickerSource 定期轮询一个外部服务，用来获取一组预配置的股票行情自动收录器的价格，并从接收到的数据创建 Flume 事件。这类 Source 仅仅将股票价格转换为简单的字符串，以字符串的 UTF-8 表示创建事件的 body。

例3-14 pollable source的示例

72

```
package usingflume.ch03;

public class StockTickerSource extends AbstractPollableSource {

  private static final String CONF_TICKERS = "tickers";
  private static final String CONF_REFRESH_INTERVAL = "refreshInterval";
  private static final int DEFAULT_REFRESH_INTERVAL = 10;

  private int refreshInterval = DEFAULT_REFRESH_INTERVAL;

  private final List<String> tickers = new ArrayList<String>();
  private final QuoteProvider server = new RandomQuoteProvider();

  private volatile long lastPoll = 0;

  @Override
  protected Status doProcess() throws EventDeliveryException {
    Status status = Status.BACKOFF;
    if(TimeUnit.MILLISECONDS.toSeconds(System.currentTimeMillis() - lastPoll) >
      refreshInterval) {
      final List<Event> events = new ArrayList<Event>(tickers.size());
```

```
      Map<String, Float> prices = server.getQuote(tickers);
      lastPoll = System.currentTimeMillis();
      // Convert each price into ticker = price form in UTF-8 as event body
      for(Map.Entry<String, Float> e: prices.entrySet()) {
        StringBuilder builder = new StringBuilder(e.getKey());
        builder.append(" = ").append(e.getValue());
        events.add(EventBuilder.withBody(builder.toString().getBytes(Charsets
          .UTF_8)));
      }
      getChannelProcessor().processEventBatch(events);
      status = Status.READY;
    }
    return status;
  }

  @Override
  protected void doConfigure(Context context) throws FlumeException {
    refreshInterval = context.getInteger(CONF_REFRESH_INTERVAL,
      DEFAULT_REFRESH_INTERVAL);
    String tickersString = context.getString(CONF_TICKERS);
    Preconditions.checkArgument(tickersString != null && !tickersString
      .isEmpty(), "A list of tickers must be specified");
    tickers.addAll(Arrays.asList(tickersString.split("\\s+")));
  }

  @Override
  protected void doStart() throws FlumeException {
    server.start();
  }
```



```
  @Override
  protected void doStop() throws FlumeException {
    server.stop();
  }
}
```

Source 继承了 AbstractPollableSource 类，它的父类 BasicSourceSemantics 实现了 start、stop、configure 方法。调用这些方法即授权这个 Source 的 doStart、doStop 和 doConfigure 方法。

当 Agent 启动时，框架就会配置这个 Source。任何配置必须通过调用 doConfigure 方法来创建。在这个例子中，自动收录机报价列表必须检索的引用要从配置文件获取。连续事件生成之间的时间间隔也从配置中读取。

在这之后，Source 的 doStart 方法被调用。这种方法可以用于建立任何网络客户端。在这种情况下，获取股票报价的服务就启动了 (因为这只是一个例子，在这种情况下，服务只是返回随机值)。

然后 Pollable Source 运行器调用 process 方法（在 BasicSourceSemantics 实现过），这也表示调用了 doProcess 方法。doProcess 方法利用外部服务获取的数据来生成事件。当 Source 成功将传入的引用转换为 Flume 事件，并且通过 Channel 处理器的 processEventBatch 方法将事件写入 Channel 时，它会返回 Status.READY；否则，它将返回 Status.BACKOFF。在这种情况下，如果在比刷新间隔更少的时间连续调用 doProcess，Source 只会返回 Status.BACKOFF，这表明 Source 运行器再次调用 process 方法时应该等待一段时间。

最终，在 Agent 停止时，doStop 方法就被调用。这个方法可以做任何清理，如关闭网络连接。一旦停止代理，该实例可以被垃圾回收。

下面是 Pollable Source 一个简单的例子。如果想看更真实的例子，请查看 Flume[jms_src_code] 的 JMS Source。

创建 Event-driven Source

Event-driven sources 实现了 EventDrivenSource 接口，对于 Flume 框架这仅仅是选择 SourceRunner 实现来运行 Source 的一个标记接口。当 Flume 框架调用 start 方法，Event-driven Source 通常会运行它们自己的线程。这类 Source 控制了它们写数据到 Channel 的速率。

74

例如，Flume 自带的 HTTP Source 就是一个 event-driven Source，它运行一个 web 监听特定端口的服务器。它基于发送给 HTTP 请求的事件来生成 Flume 的事件，并将这些事件写入与之关联的 Channel。Event-driven Source 通常运行它们自己的线程或线程池，用来处理事件生成和事件写入到 Channel。因为这些 Source 要对一些外部刺激做反应，Flume 框架创建一个新的 EventDrivenSourceRunner，通过在新线程调用 start 方法来启动这些 Source，并允许它们自己来管理。当 Agent 停止或重新配置时，调用 stop 方法来停止 Source。

Event-driven Source 响应外部事件来产生数据。大多数从外部实体来接收数据的 Source 都属于这一类。Event-driven Source 运行自己的线程用于接收数据并生成事件。大部分 Flume 自带的 Source，例如 Avro Source、HTTP Source、Exec Source 等，都是 Event-driven Source。

Event-driven Source 比 Pollable Source 稍微更复杂一些，因为 Source 必须追踪产生数据的外部程序，并且不借助 Flume 框架处理传入的数据。如果 Event-driven Source 实现了 Configurable 接口，也可以通过配置文件进行配置。

要写 Event-driven Source，需要继承 AbstractEventDrivenSource 类，其已经是实现了 Configurable 接口。或者，可以只是简单地实现标记接口 EventDrivenSource，其继承了 Source 接口。因此，Source 需要实现的唯一的方法就是 Source 接口定义的方法，如果需要，还要实现 Configurable 接口的方法。

AbstractEventDrivenSource 继承了 BasicSourceSemantics 类，其实现了 start、stop 和 configure 方法。这些调用获得了这个类的 doStart、doStop 和 doConfigure 方法的授权。

框架一旦启动了 Source，Source 就自行工作，直到被框架停止。这完全取决于 Source 的实现者，它创建线程处理生成 Flume 事件的外部事件。

例 3-15 展示了 Event-driven Source 经过 Netty-Avro IPC 接收外部服务数据的一个例子。我们在这里不会详细讨论 IPC 协议，但基本思想是，处理程序类，在这种情况下 TransactionSource 类必须实现一个 FlumeCreditCardAuth 接口，其由协议定义文件中的 Avro 生成，如下所示：

75

```
@namespace("usingflume.ch03")

protocol FlumeCreditCardAuth {

  record CreditCardTransaction {
    string cardNumber;
    string location;
    float amount;
  }

  enum Status {
    OK,
    FAILED
  }

  Status transactionsCompleted(array<CreditCardTransaction> transactions);
}
```

Avro 编译器生成代表 CreditCardTransaction 的类和表示回调的接口，当客户端发送数据给这台主机时，调用回调的 transactionCompleted 方法。TransactionSource 实现了这个接口。关于 Avro IDL 的信息可以在 Avro 文档 [avro-idl] 中查找。

例3-15 Event-driven Source的一个示例

```
package usingflume.ch03;

public class TransactionSource extends AbstractEventDrivenSource implements
  FlumeCreditCardAuth {

  private static final String CONF_HOST = "host";
  private static final String CONF_PORT = "port";
  private static final String DELIMITER = ";";

  private String host;
  private Integer port;
  private Server srv;

  @Override
  protected void doConfigure(Context context) throws FlumeException {
    host = context.getString(CONF_HOST);
    Preconditions.checkArgument(host != null && !host.isEmpty(),
      "Host must be specified");
    port = Preconditions.checkNotNull(context.getInteger(CONF_PORT),
      "Port must be specified");
  }

  @Override
  protected void doStart() throws FlumeException {
    srv = new NettyServer(
      new SpecificResponder(FlumeCreditCardAuth.class, this),
      new InetSocketAddress(host, port));
    srv.start();
    super.start();
  }

  @Override
  protected void doStop() throws FlumeException {
    srv.close();
    try {
      srv.join();
    } catch (InterruptedException e) {
      throw new FlumeException("Interrupted while waiting for Netty Server to" +
        " shut down.", e);
    }
  }

  @Override
  public Status transactionsCompleted(List<CreditCardTransaction> transactions)
```

76

```
    throws AvroRemoteException {
    Status status;
    List<Event> events = new ArrayList<Event>(transactions.size());
    for (CreditCardTransaction txn : transactions) {
      StringBuilder builder = new StringBuilder();
      builder.append(txn.getCardNumber()).append(DELIMITER)
        .append(txn.getLocation()).append(DELIMITER)
        .append(txn.getAmount()).append(DELIMITER);
      events.add(EventBuilder.withBody(builder.toString().getBytes(
        Charsets.UTF_8)));
    }
    try {
      getChannelProcessor().processEventBatch(events);
      status = Status.OK;
    } catch (Exception e) {
      status = Status.FAILED;
    }
    return status;
  }
}
```

Source需要一些配置参数，如主机名和服务器必须绑定的端口。当Source初始化且这个Source所有的指定配置都通过Context实例传递完，configure方法被Flume框架调用。在这个方法中，所有相关配置必须被验证并被保存到相关字段。例如，TransactionSource的doConfigure方法验证配置文件中用户提供的主机名和端口，如果它们没有指定则抛出异常。

配置一旦完成，框架通过调用start方法启动Source，授权该类的doStart方法。在这个方法中，Source启动服务器进程。一旦开始，框架不与Source进行交互直到它被停止。任何服务器或将要接收数据的线程都在这个方法中被启动。对于TransactionSource，在这个方法中启动NettyServer。

NettyServer管理多个通过网络接收数据的线程。当接收到一个完整Avro消息时，它调用transactionCompleted方法，来传递数据。该方法将Avro格式的数据转换为Flume事件。作为演示代码，翻译只是将字符串格式版本的数据表示为字节。这种转换可以包括任意逻辑，尽管保持逻辑简单是一个好主意，因为代码将执行收到的每个事件，并直接影响Source的性能。

传入的数据一旦被转换为Flume事件，整个Flume的批量事件通过processEventBatch方法传递到Channel处理器。Channel处理器处理关于Channel的事务，如果提交到任何Channel失败了就抛出一个异常，在这种情况下，Source会给发送者返回Status.FAILED，发送者可能会再次发送相同的数据，以确保数据被持久化到Flume Channel。

如果事件被成功地写出，Source 返回 `Status.OK` 状态，表明数据被提交给 Channel。

当 Agent 停止或被重新配置时，调用 `stop` 方法。这个调用授权了 `doStop` 方法，这个方法必须做一些所需的清理工作。在这种情况下，该方法停止 `NettyServer` 实例，并在返回前等待它终止。如果 Event-driven Source 启动线程或创建线程池，或打开套接字或文件，它们必须在 `stop` 方法中分别关闭。

总结

本章中，我们讨论了两种类型的 Flume Source（Pollable 和 Event-driven）。我们还了解了 Flume 自带的各种 Source，以及它们的配置和它们的插件等。我们还研究了一些关于 Source 的最佳实践，主要是关于批处理。最后，我们通过编写自定义 Event-driven 和 Pollable Source，对如何写 Source 有了更好的理解。

在下一章，我们将讨论 Channel 的基本概念、Flume 自带的两类 Channel，以及如何配置它们。

参考文献

- [netty] The Netty Project, *http://netty.io*
- [nio] Java New I/O API, *http://bit.ly/1wxlcQC*
- [zlib_ch3] zlib compression library, *http://www.zlib.net*
- [keystore] Java KeyStore API, *http://bit.ly/1wxlgQg*
- [keystore-type] Java KeyStore types, *http://bit.ly/1wxll6y*
- [Java-security] Customizing Java security, *http://bit.ly/1wxlmrj*
- [thrift_ch3] Apache Thrift project, *http://thrift.apache.org*
- [jms_src_code] Apache Flume JMS Source, *http://bit.ly/1wxlreC*
- [line-deserializer] Line deserializer, *http://bit.ly/1wxlyXC*
- [schema-fp] Avro Schema Fingerprints, *http://bit.ly/1wxlF5s*
- [blob] Binary large object, *http://en.wikipedia.org/wiki/Binary_large_object*
- [avro-idl] Avro IDL documentation, *http://avro.apache.org/docs/current/idl.html*

第 4 章 Channel

79

Channel 是位于 Source 和 Sink 之间的缓冲区。因此，Channel 允许 Source 和 Sink 运作在不同的速率上。Channel 是 Flume 保证数据不丢失的关键（当然，这是在正确配置的情况下）。Source 写入数据到一个或多个 Channel 中，再由一个或多个 Sink 读取。Sink 只能从一个 Channel 读取数据，而多个 Sink 可以从相同的 Channel 读取以获得更好的性能。Channel 有事务语义，允许 Flume 为写进 Channel 的数据提供明确的保证。

位于 Source 和 Sink 之间作为缓冲区的 Channel 操作，有几个优点。Channel 允许 Source 在同一 Channel 上操作，以拥有自己的线程模型而不必担心 Sink 从 Channel 读取数据，反之亦然。位于 Source 和 Sink 之间的缓冲区也允许它们工作在不同的速率，因为写操作发生在缓冲区的尾部，读取发生在缓冲区的头部。这也使得 Flume Agent 能处理 Source“高峰小时”的负载，即使 Sink 无法立即读取 Channel。

Channel 允许多个 Source 和 Sink 在它们上面进行操作。Channel 本质上是事务性的。每次从 Channel 中写入和读取数据，都在事务的上下文中发生。只有当写事务被提交，事务中的事件才可以被任意 Sink 读取。同样，如果一个 Sink 已经成功读取了一个事件，该事件对于其他 Sink 是不可用的，除非该 Sink 回滚事务。

在大多数实时应用程序临时负载出现峰值是很常见的，Flume 被设计用来处理这些情况。事件将缓冲到 Channel 直到 Sink 移除它们，让 Agent 处理传入负载的变化。每个 Agent 可以处理多少额外的数据取决于 Channel 的容量。分配 Channel 的容量应根据预期的所有写入 Channel 的 Source 最大峰值负载和所有 Sink 的读取速度的总和。这个设计还允许 Source 和 Sink 对失败具备重试逻辑。对于失败，Source 可以尝试再次写入 Channel，Sink 可以尝试再次读取。

80

事务工作流

在第 2 章“Flume Channel 中的事务”一节中讨论过，Flume Channel 是事务性的。事务本质上是原子性写入 Channel 的批量事件。事件要么全部批量地存在在 Channel 要么全都不存在。事务提供了重要的保证，它能知晓什么时候事件被写入 Channel 或从 Channel 移除。例如，Sink 可以从 Channel 中读取一个事件，试图把它写到 HDFS 但是失败了，在这种情况下，事件应该回滚到 Channel，这样就可以被这个可用的 Sink 或另外一个 Sink 读取并写到 HDFS。

只有在事务提交后事件才被移除能保证事件不会丢失，即使写失败一次，此时 Sink 可以回滚该事务。事务可以有一个或多个事件，但由于性能的原因总是推荐每个事务有相当大数量的事件。

批量写入 Channel 是很重要的，尤其是持久的 Channel。甚至 Agent 或机器重启的情况下，持久 Channel 也能保证没有数据丢失，所以它们必须在事务提交期间刷新和同步所有缓冲事件的数据到磁盘，每批量发生一次。同步到磁盘是昂贵和耗时的操作，应该只在相当大部分的数据写入页面缓存时完成。另外，同步到磁盘需要时间，包括在实际同步之前重要的系统调用的消耗，这一切都随着时间的推移而增加。每一个这样的批量也表示为一个事务，使得事务对于性能以及可靠性越来越重要。

每个 Channel 可以有多个 Source 和 Sink，分别写入 Channel 和从 Channel 读取。Source 和 Sink 关于事务以稍微不同的方式工作。Source 不直接处理事务；相反，Source 的 Channel 处理器代表它处理事务。Channel 处理器处理事务的工作方式与 Sink 几乎是相同的（除了 Sink 是从 Channel 读取数据，而 Channel 处理器是将数据放入 Channel）。

Sink 用 Channel 发起事务是通过调用 Channel 的 `getTransaction` 方法，这个方法返回 `Transaction` 的一个实例。然后 Sink 开始调用事务对象，它允许 Channel 设置任何事务所需的内部状态。通常，这包括队列的创建，以用来暂时托管事件直到事务完成。

81

事务一旦开始，Sink 在 Channel 上调用 `take` 方法（Channel 处理器的情况下是 put 方法），直到 Sink 准备提交事务。一旦 Sink 读取一个事件，该事件将不会被用于相同的或另一个 Sink，除非事务回滚。

由于性能的原因 Sink（和 Channel 处理器）通常会将一些事件批量放入一个事务中。一旦 Sink 完成了它的批量任务，Sink 就对事务调用 `commit` 方法。一旦 sink-side 事务（只进行读取的事务）被提交，该事务中的事件被 Channel 标记为删除，也不能被其他 Sink 再次使用。一旦 source-side 事务（Channel 处理器所拥有的事务）被提交，Channel 中的事件就是安全的。另外，这意味着只有当 Sink 读取完事件并提交，这些事件才能从

Channel 中被删除。

我们需要注意的是，如果 Sink 读取完一个事件，该事件对于其他 Sink 就是不可用的，除非该 Sink 回滚事务，导致该 Sink 中的事务能被再次读取。这是专门设计用来当多个 Sink 操作相同的 Channel 时，避免重复的。Channel 的每一个事件实际上可以读取和提交一次，之后事件从 Channel 中移除。

取决于所使用的特定的 Channel，即使机器或 JVM 重启，Channel 中的事件也可能是可用的。Sink 写所有的事件到任意它所支持的地方也有可能失败，因此必须重试。在这种情况下，Sink 在事务中使用 rollback 方法回滚整个事务。一旦事务被回滚到 Sink 这边，Channel 重新存储事件到 Channel 并使它们对 Sink 可用以用来读取。在 source-side 事务回滚的情况下，就好像这个事务从来没有发生过，在事务期间写入的事件从来没有写入 Channel。当回滚是由超时或其他失败引起的，这时事件可能已经提交到下一阶段的 Channel 中，回滚可能会造成重复。

当事务提交或回滚之后，通过调用 close 方法关闭事务，来清理事务使用的任何资源。图 4-1 说明了事务的工作流。

82

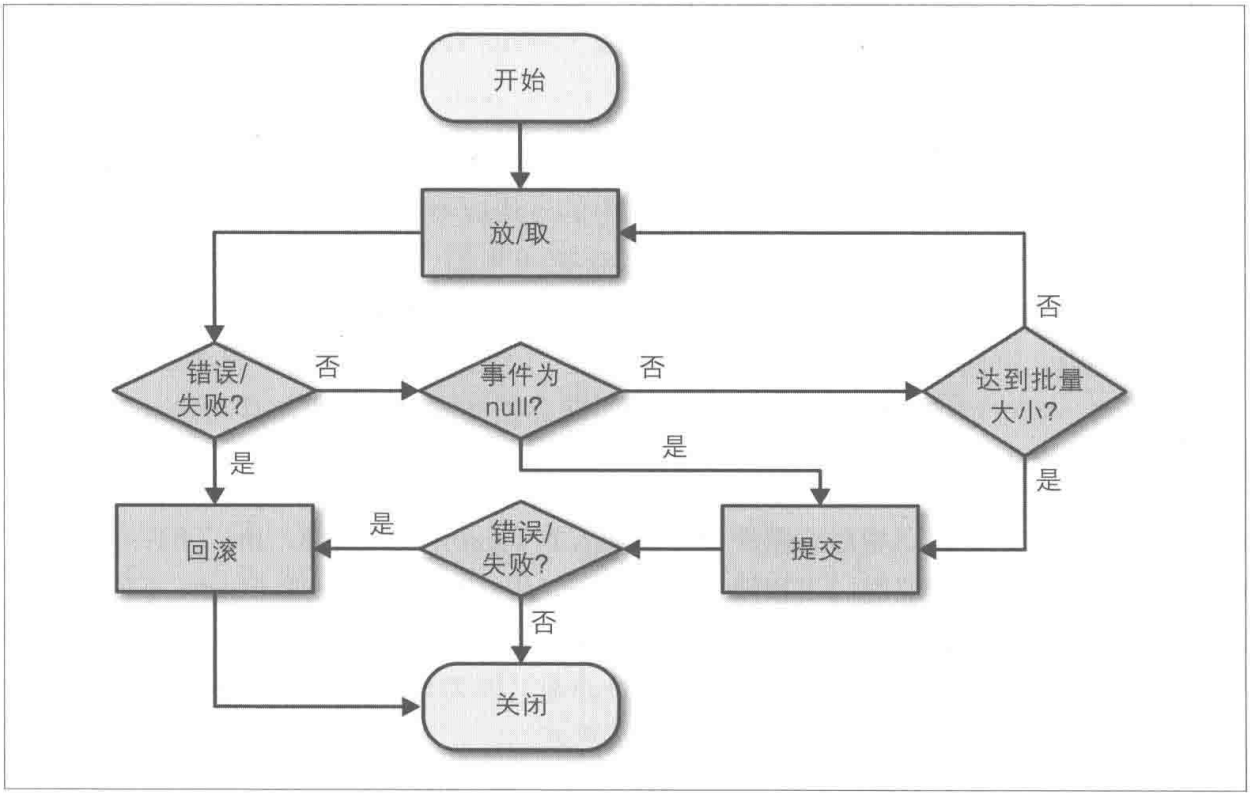


图4-1 事务工作流

单个事务不能同时写入和读取事件。这保证了 Source 只能往 Channel 中放入事件，Sink

只能从 Channel 中取走事件。

Flume 自带的 Channel

Flume 自带两种 Channel：*Memory Channel* 和 *File Channel*。这里介绍的两种 Channel 以相同的基本原理工作。两种 Channel 都是完全线程安全的，并且可以操作多个 Source 和 Sink。顾名思义，Memory Channel 在主存中存储提交的事件，而 File Channel 将事件写出到磁盘上的文件。在本节中我们将讨论这两种 Channel 和在两者之间选择时不同的考虑因素。

Memory Channel

Memory Channel 是内存中的 Channel，在堆上存储写入的事件。实际上，Memory Channel 是内存中的队列——Source 从它的尾部写入，Sink 从它的头部读取。Memory Channel 支持很高的吞吐量，因为它在内存中保存所有的数据。如前面所说，Channel 是线程安全的，可以同时处理几个 Source 的写入操作和几个 Sink 的读取操作。Memory Channel 在不需要关心数据丢失的情景下适用，因为该类 Channel 没有将数据持久化到磁盘。如果需要关心数据丢失，那么 Memory Channel 就不应该使用，因为程序死亡、机器宕机或重启都会导致数据丢失。

83

Memory Channel 支持 Flume 的事务性模型，并为每个程序中的事务维护单独的队列。一旦一个 source-side 事务提交，该事务队列中的事件就被自动移入到 Channel 的主队列中。如果提交完全成功，事务中的事件对 Sink 是可用的。如果失败了，Source 将回滚事务且事件被忽略。对于 sink-side 事务，每当 Sink 完成一个读操作事件就会被移入到事务的队列中。这保证 Sink 正确地读取了事件。当 Sink 提交事务时，事务队列被忽略，事件被废弃，等待垃圾回收。因此，Sink 的实现必须谨慎，只有当事件被成功写入目的地时，Sink 才能提交事务。

如果事务失败，事件将以相反的顺序被重新插入到 Channel 的头部，所以事件将以相同的顺序被再次读取，就像它们当初被插入时一样。用这种方法，尽管 Flume 不能保证顺序性，但是 Memory Channel 能保证事件以它们被写入的顺序进行读取。然而，当某些事务回滚，后写入的事件有可能更早写出到目的地（因为另一个 Sink 可能已经提交包含事件的事务，这些事件比回滚事务中的事件“更新”）。

Memory Channel 不需要太多精力就可以配置好，它是最容易配置的 Flume 组件之一。表 4-1 列出了 Channel 的配置参数。

表4-1 Memory Channel配置

配置参数	默认值	描述
type	-	Memory Channel 的别名是 memory。FQCN 是 org.apache.flume.channel.MemoryChannel (大小写敏感)
capacity	100	Channel 能保存的提交事件的最大数量
transactionCapacity	100	可以在单个事务被放入或取走的事件的最大数量。
byteCapacity	进程中可用堆空间总量的 80%	Channel 允许使用最大的堆空间 (字节)
byteCapacityBufferPercentage	20	byteCapacity的百分比作为一个缓冲区，大小应保持在 Channel 的字节容量和目前 Channel 中的所有事件主体的总大小之间
keep-alive	3	每次放入或取走等待完成的最大时间周期 (秒)

84

Memory Channel 在内存中保存所有的事件——因此，Channel 的容量是有限制的，由 capacity 参数确定。该参数确定了 Channel 在任何时候都能保存的被提交事件的总数。任何时候进入到 Channel 的被提交事件的总数量和从 Channel 取走的事件的数量之间的差值，都应该小于或等于 Channel 的容量。如果 Channel 达到最大容量，任何试图插入数据到 Channel 的操作都会失败并抛出 ChannelException 异常，直到最少相等数量的事件从该 Channel 中取走。

transactionCapacity 参数控制了可以在单个事务放入或取走事件的最大数量。这个参数也是一个好的防御，来抵御流氓客户端推送大量事件到 Source，导致 Agent 内存溢出。这个参数强制指定批量的大小限制，从而限制了每个 RPC 调用的事件数量，并且是一个简单拒绝服务 (DoS) 攻击的防御。

byteCapacity 参数限定了 Channel 中事件使用的内存总量。byteCapacityBufferPercentage 参数代表了字节容量的百分比，为事件 header 做了保留。当事件将要被提交到 Channel 时，只有当 Channel 中所有事件主体和当前事件主体的和，小于或等于内存中没有用于事件 header 保留的可用总量时，事件才被插入到 Channel 中。

当事件将要被插入到 Channel 时，Channel 有可能是满的。在这种情况下，插入事件的线程在失败之前将等待 keep-alive 的最大值秒数。从 Channel 取走事件的线程也需要等待 keep-alive 秒数，因为该事件需要在 Channel 中是可用的。大多数情况下，没必要设

85 置这个参数。它抑制写入或读取速率的安全值。当读取速率远大于写入速率，节流速率是很有用的，反之亦然。

下面的配置展示了一个配置能保存多达 100000 个事件的 Memory Channel，每个事务能保存 1000 个事件。Channel 中所有事件占有的内存可以是大约 5GB 空间的最大值。在这 5GB 当中，Channel 将 10% 为事件 header 保留（正如 byteCapacityBufferPercentage 参数所指明的），使得事件主体可以利用 4.5GB 大小。

```
agent.channels = mc
agent.sources = sq

agent.channels.mc.type = memory
agent.channels.mc.capacity = 100000
agent.channels.mc.transactionCapacity = 1000
agent.channels.mc.byteCapacity = 5000000000
agent.channels.mc.byteCapacityBufferPercentage = 10

agent.sources.sq.type = seq
agent.sources.sq.channels = mc
```

File Channel

File Channel 是 Flume 的持久 Channel。它将所有事件写到磁盘，因此在程序关闭或机器宕机的情况下不会丢失数据。File Channel 保证了即使机器或 Agent 宕机或重启，只有当 Sink 取走了事件并提交给事务时，任何提交到 Channel 的事件才从 Channel 移除。它被设计为高并发且可以同时处理多个 Source 和 Sink。File Channel 的设计大致上是基于 Rosenblum 和 Ousterhout 的关于日志结构文件系统的论文 [lfs-paper]。更多关于该设计的细节将在本节稍后讨论。

File Channel 被设计用于数据需要持久化和不容忍数据丢失的场景下。因为 Channel 将数据写到磁盘，它不会由于宕机或失败造成数据丢失。一个额外的好处，因为它写数据到磁盘，Channel 可以有非常大的容量，尤其是和 Memory Channel 相比。

只要磁盘空间可用，File Channel 可以有非常大的容量，能保存多达几十或几百个成千上万的事件。当预计的 Sink 从 Channel 取走事件的速率无法跟上一个有限的高峰期时，File Channel 对可能存在的大量待处理事件是特别有用的。因为一旦事件被提交，Channel 就不在内存中保存它们了，File Channel 比相等容量的 Memory Channel 需要更少的堆栈空间。

86 File Channel 保证每个写入的事件，经过 Agent 和机器故障或重新启动，仍然是可用的。它通过将每个放入 Channel 的事件写出到磁盘上。一旦提交一个事务，事务中的事件对

于读取操作就是可用的。当事件从 Channel 取走时，它们就从磁盘被读取并被传递到 Sink，一旦读取的事务被提交，事件对于移动就是完全间接引用和符合条件的。实现的更多细节将在本节稍后讨论。

File Channel 通过把多个磁盘挂载到不同的挂载点，允许用户配置多个磁盘的使用。当配置使用多个磁盘时，Channel 在磁盘之间循环，因此当更多磁盘可用的时候，Channel 性能更好。建议（但不是必需的）为 File Channel 检查点使用一个单独的磁盘。检查点反映了检查点写出瞬间 Channel 的确切状态。File Channel 无须读取所有的数据文件，就能使用检查点迅速重启。当操作期间，将检查点写出到磁盘上。重启的时候，Channel 加载最后写入的检查点，只重新进行这个检查点之后的写入和读取操作，允许 Channel 快速启动为正常运行做好准备。连续两个检查点之间的间隔默认设置为 30s，但它是可配置的。

File Channel 允许用户传递几个配置参数，允许它们基于硬件调整 Channel 的性能。表 4-2 中描述了 File Channel 的配置参数。

表4-2 File Channel配置

配置参数	默认值	描述
type	-	File Channel 的别名是 `file`。FQCN 是 `org.apache.flume.channel.file.FileChannel`（大小写敏感）
`capacity`	`1000000`	Channel 可以保存的提交事件的最大数量
`transactionCapacity`	`1000`	单个事务中可以写入或读取的事务的最大数量
`checkpointDir`	*~/flume/filechannel/checkpoint*	Channel 写出到检查点的目录
`dataDirs`	*~/flume/filechannel/data*	写入事件到以逗号分隔的列表的目录。配置多个目录，每个挂载不同的磁盘，通过并行写入磁盘可以显著提高性能
`useDualCheckpoints`	`false`	告诉 Channel 一旦它被完全写出是否支持检查点。参数的值必须为 `true` 或 `false`。如果设置为 `true`，`backupCheckpointDir` 参数则必须设置
`backupCheckpointDir`	-	支持检查点的目录。如果主检查点损坏或不完整，Channel 可以从备份中恢复从而避免数据文件的完整回放。这个参数必须指向不同于 `checkpointDir` 的目录

续表

配置参数	默认值	描述
checkpointInterval	30	连续检查点之间的时间间隔（以秒为单位）
maxFileSize	1623195647	每个数据文件的最大大小（以字节为单位）。一旦文件达到这个大小（或一旦写入下一个事件将达到该大小），该文件保存关闭并在那个目录下创建一个新的数据文件。如果将此值设置为高于默认值，Channel 仍然使用默认值为最大值
minimumRequiredSpace	524288000	Channel 继续操作时每个卷所需的最少的空间（以字节为单位）。如果任何一个挂载数据目录的卷只有这么多空间剩余，Channel 将停止操作来防止损坏和避免不完整的数据被写入。这个参数的最小值是 1048576（1MB）
keep-alive	3	每次写入或读取应该等待完成的最大的时间周期（以秒为单位）

File Channel，作为 Flume 主要的持久化 Channel，通常整体上表现了 Agent 的性能。可以通过配置对 Channel 的几个方面进行调整。File Channel 有 capacity 和 transactionCapacity 参数，这些和 Memory Channel 中的参数完全相同，但是它们的默认值更高，分别是 1000000 和 1000。

前面已经讨论过，File Channel 可以往多个磁盘写数据，但是 Channel 没有直接意识到这一点。不同磁盘可以挂载到不同的挂载点，Channel 可以配置以循环的方式往这些目录写数据。一个数据目录 Channel 总是会追加到一个文件，尽管 Channel 会按照需要从所有的文件中读取数据。因为多个 Source 可以写入到一个 Channel（并且多个线程可以写入一个 Source），Channel 将以不同的线程并行写到不同的数据目录（很多情况下，每个 Source 和 Sink 至少运行一个线程，例如 Avro Source 运行不止一个线程），因此并行磁盘的使用会有更好的性能。所以，如果你分配给 File Channel 更多起作用的磁盘，那么 Channel 的性能有可能有所改善。

88

File Channel 将逗号分隔的一系列数据目录作为 dataDirs 参数的值。默认值只能用来测试不建议用于生产环境使用。即使一个磁盘或有限数量的磁盘，每个磁盘可以使用多个数据目录以获得更好的性能，但是能提高性能的目录数量，随着磁盘的不同有所差异。

File Channel 定期写出到检查点以快速重启和恢复。checkpointDir 参数的值指定了检查点写入的目录。如果 Channel 正在写入检查点时停止，检查点可能是不完整或损坏的。

损坏或不完整的检查点会导致 Channel 重启得非常慢，因为 Channel 需要从所有的数据文件读取和重新进行。

为了避免这个问题，推荐将 useDualCheckpoints 参数值设置为 true，backupCheckpointDir 参数也要设置，建议这个目录在不同的磁盘上而不是在最初检查点存储的磁盘上。当这些参数设置了，在检查点完成之后 Flume 将尽快备份它到 backupCheckpointDir。这将确保一旦 Channel 在一个短暂的时间内操作（第一个检查点写出和备份所需的足够时间），它将能够从检查点重新启动，即使最新的一个是损坏或不完整的，这大大减少了启动时间。连续检查点之间的时间间隔是由 checkpointInterval 参数控制的。

剩下的参数是用来调整 File Channel 的性能和磁盘使用的。Flume 在数据文件的结尾追加每个事件，并且只要文件包含的事件没有被读取和提交完毕，或事件一直被写入，Flume 就一直保留这个文件。在 Flume 关闭文件并将它作为只读文件之前，maxFileSize 参数控制了文件增加的最大值。该参数默认为大约 1.6GB，也是该参数的最大值。如果设置的值比默认值更大，文件仍会在它达到默认大小时保存关闭。需要注意的是，当且仅当所有写入 Channel 的事件被读取提交，文件才被删除（实际上，当检查点追踪的最后一个事件被读取和提交时文件才被删除）。如果文件需要更快地被删除，该参数则应设置为更小的值，这样所有的事件会更快地从单独的文件被读取（因为文件更小，那么它比大文件包含更少的事件）。但是该值设置得过小将导致磁盘创建过多的文件，所以最好不要从默认值减小该值。关于文件的删除 Channel 也是保守的。即使文件没有任何将要被读取和提交的事件，Channel 也总是在每个数据目录保留两个文件。Channel 也不会删除正在写入的文件。

为了确保 Channel 不写入低空间的磁盘，则可以设置 minimumRequiredSpace 参数。一旦指定磁盘的磁盘空间低于该参数的值（或 1MB，这是更高的），Channel 就停止操作。为了保存系统资源且不影响性能，Channel 并没有在每次写入的时候检查磁盘空间，而是周期性检查，它维护内部计数器来计算磁盘的可用空间。*假设没有其他 Channel 或程序写入这个磁盘，这将使可用的磁盘空间减少得更快。*

89

前面已经讨论过，File Channel 定期写检查点到磁盘上。因为检查点反映了 Channel 写入时候的状态，所以写入检查点时，操作不能正常进行。keep-alive 参数与 Memory Channel 中的工作相似，指明了等待容量可用的时间，或检查点完成一个写入事件的等待时间，或从 Channel 取走的事件等待其可用的时间。

通常当 Channel 有更多可用磁盘时，会有更好的性能。如果足够的磁盘空间可用，Channel 应该配置一个大的容量来适应下游故障或通信高峰。基于每个 Channel 预想的

吞吐率和下游 Agent 或目的地预想的最大故障时间，Channel 容量可以配置用来处理大量积压的工作。Channel 能在几分钟之内足够快地清除数以千万计的事件，但是，很大程度上性能取决于底层的硬件。

下面的配置文件展示了一个名为 fc 的 File Channel，配置能承担 100 万个事件。这个 Channel 保存数据到三个磁盘，以轮循的方式写入到它们。Channel 也配置了备份检查点到不同的目录用来从失败中快速恢复，配置了支持多达 10000 个事件的事务。为了垃圾回收文件更快，Channel 设置数据文件大小的最大值近似于 900MB。

```
agent.channels = fc
agent.sources = sq

agent.channels.fc.type = file
agent.channels.fc.capacity = 1000000
agent.channels.fc.transactionCapacity = 10000
agent.channels.fc.checkpointDir = /data1/fc/checkpoint
agent.channels.fc.dataDirs = /data1/fc/data,/data2/fc/data,/data3/fc/data
agent.channels.fc.useDualCheckpoints = true
agent.channels.fc.backupCheckpointDir = /data4/fc/backup
agent.channels.fc.maxFileSize = 900000000

agent.sources.sq.type = seq
agent.sources.sq.channels = fc
```

File Channel 的最佳实践

你可以通过在配置中指定几个数据目录让 File Channel 写入到不同的磁盘。增加更多磁盘会直接提升性能，因为 File Channel 在磁盘之间顺序写入。

File Channel 可以备份检查点，如果检查点本身是不完整或损坏的，File Channel 可以从备份检查点启动（当 Channel 正在写入检查点时，如果机器或 Agent 宕机，这是有可能发生的）。这使得备份检查点允许 Channel 快速重启，即使检查点本身是损坏的。

如果写入磁盘失败，File Channel 会丢失数据。即使很多磁盘中的一块磁盘 Channel 写入数据失败，Channel 可能不会恢复任何的数据——即使磁盘上没有失败的事件也可能不会恢复。为了避免这种情况，File Channel 使用 RAID-ed 磁盘是一个好主意。

File Channel 使用 NFS-mounted 磁盘不是一个好主意，因为 NFS 不能提供像本地磁盘一样的保证。特别是，文件同步系统调用不能保证数据持久化到物理磁盘上，所以如果机器或 Agent 宕机，File Channel 可能会丢失数据。

File Channel 的设计和实现 *

前面已经讨论过，File Channel 持久化每个事件到磁盘，这保证了即使 Agent 或机器宕机重启时的事件也是可用的。File Channel 也持久化每个执行到磁盘的操作。这意味着当 Channel 关闭的时候，Channel 可以以相同的顺序重新回放每个记录，就像让它自己回到相同的状态一样。当 Channel 完成回放记录，就为正常的操作做好了准备。在这一节，我们将更加详细地探讨 File Channel 的内部原理。

File Channel 维护了两个独立的数据结构：Flume 事件队列（将被称为队列）和 *write-ahead* 日志（WAL）。每次写入、读取、提交和回滚都表示为一个事务事件记录（后面称为记录），记录的类型表示了操作——写入、读取、提交或回滚。每个 File Channel 在 WAL 中记录为一条记录。每次一个事件被写入 Channel，写入的记录就会写入 WAL 中，即使该事务实际上并没有被提交。

类似地，对于读取操作，会写出到一条读取记录。每条记录有一个唯一的单条递增的 ID，写入 ID，记录了该条记录是什么时候写入到 WAL 中的。每条记录也包含了事务的唯一 ID，记录是事务的一部分。因为每条写入（或读取）记录包含事务 ID，那么可以 91 通过映射事务对应的提交或回滚记录，得知哪些事件被提交，哪些事务被回滚。

通过读取 WAL 并以实际发生的顺序执行每条操作（可以从操作的写入 ID 推算得到），我们可以在任何时候重新构建 Channel 的文件状态。当 Channel 通过读取所有的数据文件被完整重构，这称为全回放（*full replay*）。全回放通常是消耗时间和破坏性的，因为需要读取所有的数据文件，WAL 中的每个操作必须执行。这时尤其如此，当 WAL 包含数百万条写入和读取记录——即使最后的状态包含很少的事件，每条记录都必须被读取并存入内存，直到我们读取到相应的提交或回滚记录。

每次写入发生时，写记录写入到磁盘。使用文件中记录的文件 ID 和偏移，Channel 为该记录构建一个唯一的 Flume 事件指针。每次事件写入到 Channel，指针向事务指出了在本地内存队列存储的记录。当事务被提交，本地队列的指针被复制到 File Channel 主队列的尾部：Flume 事件队列。因此，那个队列代表了任何时候 Channel 的实时状态。

当 Sink 从 Channel 读取事件时，队列的头部被移除，指针失效。该事件随后被存储到该事务的本地队列。对于提交，本地队列被忽略，因为事件已经被完全移除。对于回滚，事件被后推到队列。实际上队列是内存映射文件——映射缓冲区更新，并且在检查点中被推送到磁盘。

在启动时，Flume 开始一个被称为回放的程序，可以让 Channel 回到之前停止时的确切状态。启动时仅仅通过内存映射检查点文件，队列就可以被加载到内存。然后从检查点的时刻从偏移量读取数据文件（偏移量被记录在检查点时刻数据文件的元数据中），并且

写入和读取操作将按顺序应用到队列。

所有代表现在不完整事务称为 inflight 的指针也被写入到磁盘，所以检查点时刻的任何被读取但还没提交的事件，都可以在加载检查点后重新插入到队列。检查点时刻仍然运行中的写入事务也可以通过使用 inflight 文件进行恢复（可能有检查点之前但是提交之后的数据文件，它们在检查点时刻没有被插入到队列，因此没有从数据文件回放，因为数据文件只能在检查点时刻的偏移量中回放）。

92 一旦回放完成，Channel 就为正常操作做好了准备。当没有队列存在或当它是不完整时，Channel 就进行全回放。前面讨论过，Channel 可以在检查点完成之后立即备份，所以如果现在的检查点损坏，或者 Agent 在完成之前被杀死，那么可以加载备份检查点来避免全回放。

总结

本章我们讨论的 *Channel*，是位于 Source 和 Sink 之间的缓冲区，保存通过 Source 传入到 Flume Agent 的数据，直到它被 Sink 移除。Channel 可以在内存或磁盘，使用内存的 *Memory Channel* 拥有更好的性能，同时基于磁盘的 *File Channel* 保证了持久化，Agent 和机器通过持久化所有操作和数据到磁盘进行重启。

在下一章，我们将讨论 Sink 是如何设计的，以及 Flume 自带的不同种类的 Sink。我们也将讨论如何实现自定义 Sink。

参考文献

- [lfs-paper] “The design and implementation of a log-structured file system,” *http://bit.ly/1wxCsoM*

第5章 93

Sink

Flume 被设计为带有插入几乎每个组件的能力，包括那些写数据到最终目的地的组件——大多数情况下，目的地是一些数据存储。

从 Flume Agent 移除数据并写入到另一个 Agent 或数据存储或一些其他系统的组件被称为 *Sink*。为了推动这个过程，Flume 允许用户配置 Sink，可以是 Flume 内置的 Sink 之一或是由用户自己写的 Sink（对于非 Flume 内置的自定义 Sink，JAR 包应该放入 Flume 的 *plugins.d* 目录）。

Sink 是 Flume Agent 中的组件，能移除 Channel 中的数据，所以 Source 可以持续接收事件并写入到 Channel。Sink 不断地轮询 Channel 中的事件且批量地移除它们。这些事件批量写入到存储或索引系统，或者被发送到另一个 Flume Agent。

Sink 是完全事务性的。在从 Channel 批量移除数据之前，每个 Sink 用 Channel 启动一个事务。批量事件一旦成功写出到存储系统或下一个 Flume Agent，Sink 就利用 Channel 提交事务。事务一旦被提交，该 Channel 从自己的内部缓冲区删除事件。

Flume 封装了许多 Sink，可以写到诸如 HDFS、HBase、Solr、Elastic Search 等存储和索引系统。这些 Sink 通常被称为*终端* Sink，因为它们通常出现在 Flume 管道的终点。Flume 管道通过发送数据到另一个 Flume Agent 的 Agent 创建。通过 RPC sink-source 对发生这种通信。Flume 封装的 Avro- 和 Thrift-based RPC Sink，可以用于将数据发送给远程 Flume Agent 上各自的 RPC Source。

在本章中，我们将讨论不同种类的 Sink，它们的配置和管理，如何针对每个 Sink 序列化数据，以使得数据可以被写为用户选择的一种格式，以及该如何编写自定义 Sink。 94

Sink 的生命周期

Sink 使用标准的 Flume 配置系统进行配置。每个 Agent 可以没有 Sink 或若干 Sink。每个 Sink 只能从一个 Channel 中读取事件。如果 Sink 没有配置 Channel，那么 Sink 就会从 Agent 中被移除。配置系统保证：

- 每个 Sink 至少有一个正确配置的 Channel 连接它。
- 每个 Sink 有一个定义的 type 参数。
- Sink 是在 Agent 中 Sink 活跃列表中的。

类似于 Source，每个 Sink（如果配置正确）被 ConfigurationProvider 实例化然后被配置。配置错误和 configure 方法抛出异常的 Sink 会从 Agent 中被移除。Sink 一旦配置好，就会被配置系统启动。Flume 可以聚合线程到 Sink 组，这个我们将在后面讨论。每个 Sink 组可以包含一个或多个 Sink。如果一个 Sink 没有定义 Sink 组，那么该 Sink 可以被认为是在一个组内，且该 Sink 是组内的唯一成员。

每个 *Sink* 运行器运行一个 Sink 组，Sink 运行器是一个针对 Sink 组循环调用 process 方法的线程，依次转发调用到组里其中一个 Sink 的 process 方法。如果 Sink 可以立即处理更多的事件，通常是当前调用至少处理了一个事件的时候，process 方法就返回 Status.READY。如果 Sink 不能处理任何事件，Sink 返回 Status.BACKOFF，表明运行线程在再次尝试之前应该回退一段时间。为 Sink 定义 Sink 组不是强制的。如果一个 Sink 是活跃的，但是不在 Sink 组里，那么它就获取了一个针对自身的 Sink 运行器，调用 Sink 的 process 方法。针对该 Sink 的回退语义和 Sink 组里的相同。我们将第 6 章“Sink 组和 Sink 处理器”一节讨论什么时候使用 Sink 组和 Sink 处理器。

Sink 可以在配置中指定它们的 FQCN 或内置 Sink 的别名来指定。每个 Sink 必须有一个配置的 Channel。只有当 Channel 配置正确且启动成功，Sink 才会启动。表 5-1 列出了需要的参数。

95

表5-1 所有Sink必需的配置参数

参数	描述
type	Sink 的类型。可以是 FQCN 或 Sink 的别名（这只针对 Flume 内置的 Sink）。该类必须在 Flume 的环境变量中。Flume 内置 Sink 的别名在相关部分将被提及
channel	读取事件的 Channel

类似于其他所有的组件（除了 Memory Channel），当 Flume 被重新配置，Sink 的实例是不会被重用的。因此，Sink 不应该缓冲任何已被读取和提交的事件。这是因为提交事

务表明 Channel 可以删除在事务的上下文中读取的事件。当 Agent 重启时，Sink 中缓冲的这些事件可能会造成数据丢失，因为数据没有在持久的存储或缓冲中，而是在可能不能重用的 Sink 的内部数据结构中。一旦提交，Sink 必须保证事件已经被提交到下一个 Agent 的 Channel（在 RPC Sink 的情况下）或已经持久化到最终目的地。

优化 Sink 的性能

通常一个线程运行一个 Sink：Sink 运行器。因为一个线程运行一个 Sink，且大多数 Sink 倾向于 I/O 密集，可能会有这样的情况产生，当 Sink 等待 I/O 完成且 Channel 中没有将要移除的事件。因此，在大多数情况下，当个别 Sink 比写入到指定 Channel 的速率要慢很多时，那就值得添加多个 Sink 用来写入到相同目的地和从相同 Channel 读取事件。这对于 HDFS 和 HBase 的 Sink 尤其有用，能用来提升写入到 HDFS 和 HBase 的速率。

性能优化所需的 Sink 数量取决于不同的因素，包括正被使用的 Sink、目的地（Avro Source/HDFS/HBase）、网络吞吐量、Channel、Channel 使用的磁盘 I/O 性能。因此，在确定数量之前尝试各种不同数量的 Sink 配置通常是一个好主意。当增加 Sink 数量来提升性能时，必须确保的一点是资源不会因过度使用而结束，创建有太多上下文切换或网络的情况下将会导致阻塞。当达到这一点，说明了机器上的硬件被最大化利用；如果 HDFS 或 HBase 集群仍然有能力，可以在其他机器上添加更多的 Flume Agent。

写入到 HDFS：HDFS Sink

Flume 主要被设计为以可扩展的方式写入数据到 HDFS。通常用户有成百上千的数据源写数据到 HDFS。如果所有的 Source 直接将数据写入 HDFS，那么通常会给 NameNode 和 HDFS 集群造成很大压力。因此建议使用 Flume Agent 的多个层写入数据。这在第 2 章的“复杂流”一节已经解释过。Agent 将使用 HDFS Sink 最终将数据写入到 HDFS。HDFS Sink 是很灵活的，可以基于事件的报头、事件的时间戳（或 Sink 的当前时间）等，配置写入到不同的目录。我们将在稍后一节中看到 Sink 关于批量事件到不同目录的工作细节。

HDFS Sink 允许用户通过编写和部署的序列化器，自定义写入到 HDFS 事件的格式。HDFS Sink 支持 Hadoop 1 和 Hadoop 2，但是代码必须针对 Hadoop 的相关版本进行特定的编译。例如，Flume 针对 Hadoop 2 进行的编译将不能将数据写入到 Hadoop 1 HDFS 集群。这是因为在 Hadoop 1 和 Hadoop 2 之间二进制不兼容性变化。因此，针对 HDFS 特定版本编译的二进制文件必须在特定的 HDFS 中使用。Apache Flume 目前只发布针对 Hadoop 1 编译的二进制，但是有一些供应商，基于 Hadoop 2 的发行版，发布可编制的

二进制（但是它们可能不会影响 Apache Hadoop 工程发布的二进制）。

HDFS Sink 将数据写到 HDFS 的 *bucket*。实际上，一个 bucket 就是一个目录（尽管多个 Sink 可以写入到相同目录，但是用户需要谨慎配置一个 Sink 写入到一个指定文件的情况）。因此基于配置中指定的条件，我们可以将 bucket 认为是一个 HDFS Sink 写数据的目录。

一个 HDFS Sink 可以同时将数据写到多个 bucket 中，但是一个事件将只进入到一个 bucket 中。无论任何时候每个 bucket 最多有一个文件打开，但是每个 Sink 在不同的 bucket 里可以打开若干文件。Flume 允许用户基于配置文件中不同的指定参数动态创建 bucket，例如事件报头中的时间戳、指定的报头的值等。然后每个事件基于这些参数进行计算且写入到一个 bucket 中。HDFS Sink 批量处理事件，在一个批处理中刷新所有的事件到 HDFS 中。

如果 Flume 正在写入的 HDFS 版本支持 Flume，Flume 刷新整个批量到 DataNode 内存中，然后数据对其他读取的进程将是可用的。数据可视化是 HDFS 处理的一个重要方面，还有比较重要的是，要理解 Flume 写入的数据什么时候对其他进程可见。默认情况下，Flume 不会将数据写入到 HDFS 的隐藏文件中，而是写入到以 *.tmp* 扩展名结尾的文件。这可以在配置中变更。因为 Flume 每次批量写出事件并刷新数据到 HDFS，HDFS 上其他运行的进程将会在批处理成功刷新数据到 HDFS 后，立即看到 Flume 写入的数据。如果 Flume 写入事件到 HDFS 失败或写超时，Sink 利用 Channel 回滚事务且关闭将要写入的文件。只有当写入成功，Sink 才利用 Channel 提交事务，因此保证了只有当数据成功被写入 HDFS，数据才从 Channel 中被移除。如果一个事件再次被写入到同一个 bucket，那么 Sink 将会创建一个新的文件。

理解 Bucket

前面已经讨论过，Sink 可以决定事件写入到 HDFS 的哪个目录。HDFS Sink 使用几个参数创建 bucket 路径。为了动态创建这些路径，配置文件必须指明路径为 `hdfs.path` 参数。该路径可以指定一个或多个 Sink 将取代的转义序列，用来构造写事件的真实路径。

为了动态生成文件名，`hdfs.filePrefix` 参数的值也可以是转义的。简单地说，转义是将配置中指定的字符序列替换为另一个序列。在 HDFS Sink 中，所有的转义序列都是以 % 字符开始的。HDFS Sink 能识别几个转义序列，并将它们替换为另外的值。我们将举例说明，展示 Flume 是如何插入指定事件报头的值到路径中的。

考虑添加如下配置参数到 HDFS Sink：

```
agent.sinks.hdfsSink.hdfs.path = /Data/Flume/%{topic}
```

当 HDFS Sink 从 Channel 读取一个事件，它读取主题报头的值，并利用配置参数，将路径的转义序列替换为指定主题报头的值。例如，一个主题报头的值为 inputData 的事件，将写出到 HDFS 路径为 */Data/Flume/inputData* 目录中的文件。另一个主题报头的值为 logData 的事件，将写出到 HDFS 路径为 */Data/Flume/logData* 目录中的文件。该配置参数经常用来替换主机拦截器插入的主机信息，或者更经常的是，用来替换时间戳拦截器或生成事件的系统插入的时间戳。

HDFS Sink 提供了很强大的时间戳的转义。它支持几个使用时间戳事件报头值的转义序列。为了使用时间戳用于转义，Sink 对于报头为 timestamp 的键，预期为一个数字的值（或配置中 hdfs.useLocalTimestamp 参数必须设置为 true）。HDFS Sink 将时间戳的值认为是 Epoch，定义为自 1970 年 1 月 1 日以来 00:00:00 UTC 格式总的秒数，不包括闰秒。基于该报头的值，Sink 可以替换多个转义序列为相应的值。基于时间戳的转义序列在表 5-2 中列出。 98

表5-2 HDFS Sink时间戳转义序列

转义序列	替换值
%t	毫秒值的 Epoch 时间
%s	秒值的 Epoch 时间
%H	24 小时时钟指定的一天当中的小时值（00..23）
%I	12 小时时钟指定的小时值（01..12）
%M	每小时的分钟数（00..59）
%S	每分钟的秒数（00..59）
%k	和 %H 一样，但是最开始移除一个 0（0..23）
%p	AM 或 PM（或者当前区域它们的相等值）
%z	时区指定为自 UTC 以来用 +hhmm 格式表示的差值（例如，+0530 表示印度标准时间，比 GMT 时间提前 5 个小时 30 分钟）
%a	当前区域的周的星期简称（例如 Mon、Tue 等）
%A	当前区域的周的星期全称（例如 Monday、Tuesday 等）
%b	当前区域的月的简称（例如 Jan、Feb 等）
%B	当前区域的月的全称（例如 January、February 等）
%d	月的天数（例如 01、02 等）

续表

转义序列	替换值
%c	当前区域指定的天，日期和时间（例如，Sun Feb 9 14:05:45 2014）
%m	月的数值表示（例如，January 将会是 01）
%D	以 mm/dd/yy 表示的日期
%y	世纪开始以来的年（00..99）
%Y	当前年份（例如 2013、2014 等）

多个转义序列可以在相同的路径中使用。这实质上意味着用户可以根据下面展示的日期、小时、分钟等，将它们的消息分桶（bucket）：

```
agent.sinks.hdfsSink.hdfs.path = /Data/Flume/%{topic}/%Y/%m/%d/%H
```

这个例子对于每一个主题，将把所有时间戳为相同小时的事件写到相同目录内。如果一个事件主题名为 UsingFlume，时间戳是 2:35 PM on September 1, 2014，它将写入到 */Data/Flume/UsingFlume/2014/09/01/14* 目录。

通常基于时间的分桶，在该小时后的几分钟，通常使用 Oozie 或 cron job（在 Flume 管道中引发延时）运行加载数据到系统中的查询，用来加载数据到像 Hive 或 Impala 的系统中。这种分桶的一个重要方面是，有更老时间戳的数据有可能被写入到 bucket，因为 Sink 使用的时间戳是基于事件的报头——在 Flume Agent 中，如果时间戳在事件创建时或通过一个拦截器被插入，Agent 故障或网络延时会造成事件到达 HDFS Sink 的时间比时间戳晚很多。如果一个 Hive 查询是基于时间分桶加载数据，一定会造成写入的延时。如果只有最新的目录被扫描，那么目录中更老时间戳的新数据可能从不会被加载。

HDFS Sink 有一个配置参数，`hdfs.useLocalTimestamp`，如果设置为 `true`，那么强制 Sink 使用运行 Agent 机器的系统时间；它可以用来确保数据基于当地时间戳写入。这个参数需要谨慎使用，因为 HDFS 集群和 Flume Agent 可能不同步并且可能不会有精确的相同时间。

这里描述的基于时间分桶的配置有一个问题，可以按照每年、天、小时、分钟或秒分桶，但是每几分钟创建一个新 bucket 是不可能的；例如，每十分钟一个新 bucket。如果用户想每十分钟进行分桶，那么他需要每分钟有一个 bucket，意味着他将以 HDFS 上众多数量的小文件而收场。但是 HDFS Sink 确实提供了一种通过配置参数的组合来实现的方法。所需的三个配置参数列在表 5-3 中。

表5-3 HDFS Sink round-down参数

参数	默认值	描述
hdfs.round	False	表明事件的时间戳是否应向下取整
hdfs.roundValue	1	时间戳将取整这个参数多个值中的最大值，以 hdfs.roundUnit 参数指定的值为单位
hdfs.roundUnit	Second	hdfs.roundValue 配置参数的单位（可以是秒、分钟或小时）

如果 hdfs.round 参数设置为 true，那么 HDFS Sink 从事件报头中读取时间戳，并将它向下取整到 byhdfs.roundValue 指定的多个值中最高的一个，以 hdfs.roundUnit 指定的值为单位。当使用了 %t 转义序列，向下取整的值就不能使用。让我们看看这里展示的 Sink 的配置参数：

```
agent.sinks.hdfsSink.hdfs.path = /Data/Flume/%{topic}/%Y/%m/%d/%H/%M
agent.sinks.hdfsSink.hdfs.round = true
agent.sinks.hdfsSink.hdfs.roundUnit = minute
agent.sinks.hdfsSink.hdfs.roundValue = 10
```

如果一个事件的时间戳是 2:35 PM on September 1, 2014，HDFS Sink 向下取整的时间戳为 2:30 PM on September 1, 2014，并将这个事件写出到 */Data/Flume/UsingFlume/2014/09/01/14/30* 目录。实际上，位于 2:30:00:00:000 PM 和 2:39:59:59:999 之间的事件都是写出到相同的目录。使用类似的配置原理（就像下面展示的），用户可以每两小时就有 bucket。

```
agent.sinks.hdfsSink.hdfs.path = /Data/Flume/%{topic}/%Y/%m/%d/%H/%M
agent.sinks.hdfsSink.hdfs.round = true
agent.sinks.hdfsSink.hdfs.roundUnit = hour
agent.sinks.hdfsSink.hdfs.roundValue = 2
```

配置 HDFS Sink

HDFS Sink 有很多配置参数，允许用户在一个很好的粒度级别控制 Sink 的表现。表 5-4 列出了配置 HDFS Sink 使用的参数。

101 表5-4 HDFS Sink配置

参数	默认值	描述
type	-	HDFS Sink 的别名是 hdfs。也可以使用 FQCN org.apache.flume.sink.hdfs.HDFSEventSink
hdfs.path		Sink 应该写入的目录路径。允许转义序列
hdfs.filePrefix	FlumeData	文件名的前缀。文件名包含的这个前缀在数字和名字之后
hdfs.fileSuffix	-	文件名使用的后缀。如果需要扩展，则应该明确指定“.”
hdfs.inUsePrefix	-	HDFS Sink 正在写入的文件使用的文件名前缀
hdfs.inUseSuffix	.tmp	HDFS Sink 正在写入的文件使用的文件名后缀
hdfs.timeZone	-	创建 bucket 路径使用的时区
hdfs.rollInterval	30	文件保存之前的秒值时间。设置为 0 禁用基于时间的文件保存
hdfs.rollSize	1024	在文件保存之前写入文件的事件的最大值。设置为 0 禁用基于事件计算的文件保存
hdfs.batchSize	100	每批次写入事件的最大值
hdfs.idleTimeout	0	连续事件到一个未关闭的文件要等待的最大时间周期的秒值。设置为 0 禁用该项
hdfs.fileType	SequenceFile	使用的文件格式。可以是 SequenceFile、DataStream 或 CompressedStream 的一种
hdfs.codeC	-	用来压缩文件的压缩编码。可以是 gzip、bzip2、lzo、lzop 或 snappy 中的一种
hdfs.maxOpenFiles	5000	HDFS Sink 一次可以保持打开文件的最大数量
hdfs.callTimeout	10000	每个 HDFS 操作超时前等待的毫秒延时
hdfs.threadsPoolSize	10	线程池中执行 HDFS 操作的线程数量
102 hdfs.rollTimerPoolSize	1	基于 hdfs.rollInterval 和 hdfs.idleTimeout 参数的线程池中保存 HDFS 文件的线程数量
hdfs.kerberosPrincipal	-	登录到 Kerberos key 分布中心（KDC）（使用的安全 HDFS）的所用的 Kerberos 主体

续表

参数	默认值	描述
hdfs.kerberosKeytab	-	使用 hdfs.kerberosPrincipal 登录到 KDC（使用的安全 HDFS）的 keytab 文件的路径
hdfs.proxyUser	-	Flume 应该模仿的用户，如果需要。如果设置为 none，Flume 以当前用户写入数据
hdfs.useLocalTimeStamp	false	如果设置为 true，HDFS Sink 将使用现在 Agent 的时间戳做基于时间的分桶
hdfs.round	false	表明事件的时间戳是否应向下取整
hdfs.roundUnit	second	hdfs.roundValue 配置参数的单位（可以是秒、分钟或小时）
hdfs.roundValue	1	hdfs.roundUnit 参数指定的时间戳将舍去的多个参数的单位的最大值
serializer	TEXT	使用的序列化器。其他内置序列化器可以通过设置该值为 AVRO_EVENT 或 HEADER_AND_TEXT 来使用。为了使用自定义类，该值应该为序列化器 Builder 类的完整类别名称，Builder 类实现了 EventSerializer$Builder。序列化器本身必须实现 EventSerializer。见下一节介绍
serializer.*	-	传递给序列化器的配置参数。见下一节介绍

为了使用 HDFS Sink，设置 Sink 的 type 参数的值为 hdfs 或使用 FQCN org.apache.flume.sink.hdfs.HDFSEventSink。

HDFS Sink 需要用户提供一个文件写入的目录路径。目录路径应该设置为 hdfs.path 参数的值。该参数支持分桶，在第 4 章“理解 bucket”一节中解释过。因此，该路径实际上不需要指定一个确切的目录，而是基于指定的分桶使用一组目录。HDFS Sink 创建的每个文件的名字，是根据 HDFS 配置中一系列参数指定的特定模式来的。

hdfs.filePrefix 参数指定了文件名使用的前缀。在该参数中分桶是允许的，但是只有当 103 第一个文件创建，基于时间的分桶才会完成，并对 bucket 中所有的文件保持相同，所以最好要避免使用基于时间的分桶，并且在文件前缀中使用基于报头的分桶。

hdfs.fileSuffix 参数可以用来指明文件名的后缀。通常，通过使用这个参数指定文件的扩展。Flume 将不会自动在扩展之前添加句点（.），所以如果需要例如“txt”的扩展名，那么该参数的值需要被指定为 .txt。最终，文件名将会由文件前缀、文件后缀和中间的

计数器组成。考虑下面的配置：

```
agent.sinks.hdfsSink.hdfs.filePrefix = UsingFlume
agent.sinks.hdfsSink.hdfs.fileSuffix = .oreilly
```

配置将最终生成例如以 *UsingFlume.33434321.oreilly* 和 *UsingFlume.33434322.oreilly* 等命名的文件。

当文件仍然被写入时，建议类似Hive或MapReduce的系统忽略该文件直到Flume关闭它。这能确保当 MapReduce 或 Hive 读取文件的时候，文件内容不会更新。不幸的是，确定文件是否正被写入和内容是否更新是不容易的。为了解决这个问题，一旦 Flume 关闭了文件，HDFS Sink 允许用户添加后缀和前缀到将要移除的文件名。

通过使用这些参数，MapReduce 作业或 Hive 查询可以过滤一些文件。当文件正被写入，文件名将会以 in-use 前缀开始，以 in-use 后缀结尾，中间是最终的文件名（基于前面解释过的文件前缀和文件后缀参数）。我们再次看一下配置例子：

```
agent.sinks.hdfsSink.hdfs.filePrefix = UsingFlume
agent.sinks.hdfsSink.hdfs.fileSuffix = .oreilly
agent.sinks.hdfsSink.hdfs.inUsePrefix = .
agent.sinks.hdfsSink.hdfs.inUseSuffix = .temp
```

这将创建以 *.UsingFlume.33434321.oreilly.temp*、*.UsingFlume.33434322.oreilly.temp* 等命名的文件，一旦关闭将会重命名为 *UsingFlume.33434321.oreilly*、*UsingFlume.33434322.oreilly* 等。前面的配置确保了文件是隐藏的直到它们最终被关闭，重命名了文件名，文件名是前面描述过的 `hdfs.filePrefix` 和 `hdfs.fileSuffix` 参数指定的。

HDFS Sink 可以基于时间分桶，在第 4 章“理解 bucket”一节中描述过。Sink 基于 `hdfs.timeZone` 参数转换 Epoch 时间戳为分桶的日期和时间。如果没有指定的时区，那么就使用运行 Agent 机器的当地时区。时区按照 Internet Assigned Numbers Authority (IANA) 标准格式 [tz-list]。

104

为了确保文件是关闭，且对于处理数据的系统数据是可用的，HDFS Sink 可以被配置用基于时间、事件计数或写入到文件的预压缩大小来滚动文件。`hdfs.rollInterval` 参数控制基于时间的文件滚动。每个文件在该参数指定的时间（以秒为单位）之后被刷新、关闭和重命名。设置为 0 禁用基于时间的滚动。

HDFS Sink 也可以基于写入事件的数量来滚动文件。`hdfs.rollCount` 参数控制了这一点。设置为 0 表示禁用基于计数的滚动文件。最终，有可能使用 `hdfs.rollSize` 参数（即使写入的数据是压缩的格式，但这是未压缩的大小），基于文件中事件主体的总大小来滚动文件。该参数的值指定为以字节为单位。只要滚动文件的参数之一达到了阈值，文件

就会被滚动（例如，即使在保存间隔达到之前事件的数量达到了计数的数量，那么文件会被滚动）。所以，同时可以启用不止一个参数。

当使用基于时间的分桶，有可能在某个固定的时间点之后，就不会有事件再写入到bucket中。如果对于文件关闭启用了滚动间隔，它将至少需要滚动间隔的时间。如果滚动间隔没有启用，这样的文件可能从不会被关闭。所以，总是推荐设置hdfs.idleTimeout，它表示在最后一个事件写入文件之后关闭文件要等待的秒数值时间。大多数情况下，将这个值设置比滚动间隔小得多更有意义，所以当没有数据再写入到bucket时，比关闭间隔更短的时间的数据就是可用的。但是它应该设置得比事件写入bucket的平均时间稍微大一些，这样文件不会被要求更频繁地关闭文件——不然，HDFS上会有很多小文件，给HDFS NameNode造成压力。

当有多个HDFS Sink写入HDFS，在同一个Agent或不同的Agent上，每个Agent写入到不同的目录或使用不同的文件前缀是很重要的。重要的原因是确切地说一个HDFS Sink只能写入一个文件；其他试图写入相同文件的HDFS Sink可能会遇到异常且不能再写数据。为了避免这个问题，每个HDFS Sink拥有它正在写入的目录且没有其他写入的Sink或进程，建议使用这种方式分桶数据。这可以通过使用主机名和Sink名作为bucket名字的一部分来完成（主机名可以通过Flume自带的主机拦截器来插入），或者使用多路复用Channel选择器来确保每个HDFS Sink写的数据属于一个确定的主题等。

HDFS Sink允许用户以顺序文件或压缩文件，或者任何二进制或文本的格式写数据。hdfs.fileType参数控制了文件格式。如果要以顺序文件写数据，就将该值设置为SequenceFile。对于顺序文件，Flume写事件主体作为数值键的对应值，写为LongWritable类型。如果事件报头中有时间戳，可以使用时间戳作为键；否则，就是用当前事件的毫秒值。事件主体基于hdfs.writeFormat参数将它本身写出为ByteWritable或TextWritable类型。如果要使用textWritable，该参数应该被设置为text；对于ByteWritable则应设置为writable。当使用顺序文件，serializer和serializer.*参数将被忽略。当使用数据流或压缩流时，hdfs.writeFormat参数会被忽略。

HDFS Sink允许用户指定序列化器来转换事件为用户适用的数据格式。只有当文件类型设置为DataStream或CompressedStream，序列化器才会启用。然后序列化器可以以它选择的格式将数据写到磁盘。序列化器内部可以做任何压缩，但是Flume本身适用数据流的时候不压缩写入的数据。为了压缩数据，文件类型必须设置为CompressedStream，hdfs.codeC参数必须设置为gzip、bzip2、lzo、lzop或snappy中的一种，指定了写入到HDFS使用的压缩编码。当使用数据流时压缩编码不需要设置。当使用压缩流时，如果文件后缀没有指定，对于配置好压缩编码的写入文件将使用恰当的扩展名（文件后缀

将覆盖压缩编码默认的扩展名)。

如果需要，HDFS 可以使用 Kerberos[Kerberos] 来确保 HDFS 的安全。HDFS Sink 可以使用在配置文件中指定的认证信息写入到安全 HDFS 集群上。hdfs.kerberosPrincipal 参数指明了登录到 KDC 使用的主体。hdfs.kerberosKeytab 参数应该指明到达 Kerberos keytab 文件的完整路径。该文件对于运行 Flume Agent 的用户应该是可读的，而且必须包含使用的委托人对应的 keytab。HDFS 允许用户模拟其他的用户。Flume 通过 hdfs.proxyUser 参数支持该特征。为了模拟其他用户，指定用户的用户名为这个参数的值即可。Flume 将以那个用户写数据到 HDFS。详细的配置见 HDFS 文档 [impersonation]。为了使用这个功能，运行 Flume Agent 的用户必须在 HDFS 配置中被授权给可以模拟 Flume 写入的用户。

Kerberos、HDFS 和 HBase Sink

相同 Agent 中的所有 HDFS Sink 必须使用相同的 Kerberos 认证信息去登录。另外，所有的 HBase Sink 也必须使用相同的认证信息。对不同的 HDFS 和 HBase Sink 使用不同的认证信息，会导致它们中的一个或多个不能写入到 HDFS 或 HBase。如果多个 HDFS 或 HBase 集群需要相同的数据但是有不同的认证信息，那么数据必须从起源的 Flume Agent（接收或生成数据的 Agent）路由到不同的 Flume Agent，每次写入使用多个认证信息中的一个。这可以通过以下方法来简单实现，将接收数据的 Source 绑定多个 Channel，由 Avro Sink 每个 Channel 拉取数据，并且将它们发送到不同的写 HDFS 的 Flume Agent。

在配置的时间周期之后，HDFS Sink 可以对每个 HDFS 操作超时。这能确保 Sink 不会停止处理事件以防 DataNode 挂起。hdfs.callTimeout 参数值就是配置好的毫秒值的超时时间。该参数的最佳值取决于用户具体的部署，需要谨慎设置以避免太多的超时发生或 Sink 等待时间过长，这可能会影响吞吐量。这些操作使用独立的线程池执行，线程池的大小可以使用 hdfs.threadsPoolSize 参数进行设置，但是该值很少需要改变。

为了触发滚动和闲置超时，Flume 使用单独的线程池，它的大小也可以配置，虽然很少需要改变。这可以使用 hdfs.rollTimerPoolSize 参数调整。

HDFS 客户端 API 对于打开的文件保持了内部缓冲区，尤其对于压缩的文件。为了限制打开文件的数量，从而节约使用的资源，一旦打开文件的数量达到了上限，Sink 就会自动关闭最早写入的文件。这个上限通过 hdfs.maxOpenFiles 参数指定。

hdfs.useLocalTimestamp 参数如果设置为 true，会使用用于基于时间分桶的托管 Agent 的机器上的本地时间。

hdfs.batchSize控制了每个事务写出事务的最大数量（提供在Channel中有足够的事件可用——如果Channel中有更少的事件，会认为批处理立即完成）。每次批处理利用Channel提交一个事务。

让我们看一些HDFS配置的真实例子。下面的配置将事件写到10分钟bucket，使用基于时间的分桶和向下取整，Snappy格式。如果没有配置文件后缀，文件将会自动添加后缀*.snappy*。每2分钟或者当100000个事件写到一个文件，无论哪个条件先达到，Sink会关闭文件，如果文件已经打开30秒没有写入，且使用闲置超时就关闭文件。

Sink使用内置的TEXT序列化器以纯文本的格式写事件。在任何时候当100个文件打开，Sink也会关闭最早写入的文件。它使用Flume主体来做Kerberos认证信息登录，以UsingFlume用户写数据。Flume用户必须在HDFS配置中被授权可以模拟UsingFlume用户。当写入到该Sink的文件使用“.”前缀使之成为隐藏文件，文件的后缀名是*.temporary*。一旦文件关闭，文件会被重命名为最终的文件名。

```
agent.sinks = hdfsSink
agent.channels = memoryChannel

agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000

agent.sinks.hdfsSink.type = hdfs
agent.sinks.hdfsSink.channel = memory
agent.sinks.hdfsSink.hdfs.path = /Data/UsingFlume/%{topic}/%Y/%m/%d/%H/%M
agent.sinks.hdfsSink.hdfs.filePrefix = UsingFlumeData
agent.sinks.hdfsSink.hdfs.inUsePrefix = .
agent.sinks.hdfsSink.hdfs.inUseSuffix = .temporary
agent.sinks.hdfsSink.hdfs.fileType = CompressedStream
agent.sinks.hdfsSink.hdfs.codeC = snappy
agent.sinks.hdfsSink.hdfs.rollSize = 128000000
agent.sinks.hdfsSink.hdfs.rollCount = 100000
agent.sinks.hdfsSink.hdfs.rollInterval = 120
agent.sinks.hdfsSink.hdfs.idleTimeout = 30
agent.sinks.hdfsSink.hdfs.maxOpenFiles = 100
agent.sinks.hdfsSink.hdfs.round = true
agent.sinks.hdfsSink.hdfs.roundUnit = minute
agent.sinks.hdfsSink.hdfs.roundValue = 10
agent.sinks.hdfsSink.hdfs.kerberosPrincipal = flume/_HOST@OREILLY.COM
agent.sinks.hdfsSink.hdfs.kerberosKeytab = /etc/flume/conf/UsingFlume.keytab
agent.sinks.hdfsSink.hdfs.proxyUser = UsingFlume
```

108

写入到 Hadoop 1 和 Hadoop 2*

Flume 可以写入到 Hadoop 1 或 Hadoop 2 的 HDFS。Apache 软件基金会发布的二进制工件是针对 Hadoop 1 的。如果数据被写入到 Hadoop 2.x 集群，那么用户必须使用命令 `mvn clean install -Dhadoop.profile=2` 重新编译 Flume（该命令要求基于 Hadoop 2.x 编译的 HBase 0.94.2 在 Maven 缓存中是可用的），或使用命令 `mvn clean install -Dhadoop.profile=hbase-98`，用 Hadoop 2.4.0 和 HBase 0.98 编译。Flume 与 Flume 的二进制文件中没有打包 Hadoop 及其依赖关系。如果 Hadoop 已经安装在机器上，Flume 会自动获得依赖；否则，用户必须在 Flume 的环境变量中添加包含 HDFS 客户端库及其依赖关系的目录。Hadoop 供应商通常发布 HDFS 客户端库的版本号，它们会使用该版本号发布 Flume 二进制文件。

使用序列化器控制数据格式 *

写出到 HDFS 的数据最终会被各种各样的其他系统所使用。因此，HDFS Sink 足够灵活去支持可以被这些系统理解的数据格式是很重要的。HDFS Sink 通过允许用户嵌入序列化器来转换 Flume 事件为处理事件的系统可以理解的格式，并将它们写出到最终会刷新到 HDFS 的流中，这样就能允许用户以适用的的数据格式写入 HDFS。Flume 自带了一些序列化器，支持常见的格式 text 和 Avro 等。需要记住的是，只有当 HDFS Sink 配置好了事件使用数据流或压缩流，序列化器才启用。本节将说明如何编写一个序列化器，以及如何告诉 HDFS Sink 使用序列化器。

文件格式

当写数据到HDFS，或数据可能被应用处理且并不意味着人类可读的任何其他系统，使用例如 Avro、Protobuf 等二进制格式更有意义。就磁盘占用的空间量和写出数据花费的时间量而论，二进制格式往往更有效率。这是因为二进制格式可以更有效地编码数据，在 Avro 中一个整数四个字节，而如果使用 text，如果数量超过四位数，它将超过四个字节（在普通的旧的 ASCII 中）。多种二进制格式在写之前还能压缩数据。

Flume 的二进制格式还有另一个特别大的优势。Flume 批量写出事务到 HDFS，在
109 需要时分配块。如果由于其他任何原因一个块分配失败或者写失败，可能会写入部分事件。当这样的失败发生，Flume 将在一个新文件重试该事件，但部分事件仍将在HDFS 文件里。如果这部分事件代表Hive消费的一行数据，Hive查询可能会失败，因为数据可能没有意义，也可能是不完整的。甚至更糟糕的是，如果解析没有失

> 败那么没有办法知道数据是否是有效的，因为可能在一行中的最后一列的中间发生写失败。这样的失败会导致异常数据被处理。另一方面，二进制格式将无法读取一个不完整的写入，通常会用一个异常来允许读取数据的应用程序指导数据是异常的且可以跳过。在这种情况下，读取数据的应用程序可以直接移动到下一个文件。为了被更好地检测到错误，二进制格式通常被用于从 Flume 中写出数据。
>
> 如 RCFile、ORCFile、Parquet 等二进制柱状格式怎么样？大多数列式格式对于批量写是优化了的，即在一个写操作中写大批量的数据。另一方面，Flume 按照它传入的数据写数据。可能对于这样的格式不是非常适合。在 Flume 中使用像 Parquet 的格式的一个好方法是，以 Avro 格式写数据，然后使用 Parquet 自带的工具或使用 Impala 把它转换成 Parquet。

HDFS 序列化器通常是一个很简单的类，使用 Flume 配置系统进行配置。序列化器类本身必须实现 `org.apache.flume.serialization.EventSerializer` 接口。Flume 需要使用继承自 `org.apache.flume.serialization.EventSerializer.Builder` 的 builder 类创建该类。所有的 builder 类必须有一个无参的公共构造方法，当实例化 builder 类时，Sink 使用该构造方法。

例 5-1 中展示了 `EventSerializer` 接口。

例5-1 EventSerializer接口

```
package org.apache.flume.serialization;
public interface EventSerializer {
  public static String CTX_PREFIX = "serializer.";
  public void afterCreate() throws IOException;
  public void afterReopen() throws IOException;
  public void write(Event event) throws IOException;
  public void flush() throws IOException;
  public void beforeClose() throws IOException;
  public boolean supportsReopen();
  public interface Builder {
    public EventSerializer build(Context context, OutputStream out);
  }
}
```

HDFS Sink 传递一个 `OutputStream` 实例和 `Context` 实例给 builder，会依次创建和配置返回给 Sink 的 `Serializer` 实例。序列化器被期待转换传递到它 `write` 方法的事件为需要的格式，然后将数据写出到提供的输出流中。序列化器可以使用传递的 `Context` 实例进行配置。

110

文件打开之后会立即调用 afterOpen 方法，这个方法可以用来写文件级别的 header 信息，例如序列化器中的高级标签，可以在 XML 中写数据。

每次 Sink 读取事件时，调用 write 方法。该方法主要负责转换 Flume 事件为所需的格式并将它写到输出流。Sink 一旦完成了整个批处理，就调用 flush 方法，该方法必须刷新序列化器的任何内部缓冲区的数据到流中。如果流是被序列化器的 BufferedOutputStream 包装，序列化器必须刷新缓冲流，这样所有传递到序列化器的数据才能刷新到输出流中。序列化器不需要刷新传入流，因为这通过 HDS Sink 本身已经完成。

HDFS Sink 关闭文件前，调用 beforeClose 方法。该方法可以用来编写任何文件的追踪（如关闭顶级报头）。如果 supportsReopen 方法返回 true，HDFS Sink 可以附加到文件。所以，只有当文件可以关闭然后为写操作可以再次打开，那么该方法必须返回 true。

例 5-2 展示的序列化器，序列化 Flume 事件到一个文件中，文件中包含使用例 3-8 展示过的 Protobuf 定义序列化的 Protobuf。在 Protobuf 序列化事件之后，每个事件被表示为一个整数，表示事件的长度。

例5-2 Protobuf事件序列化器

```
package usingflume.ch05;

public class ProtobufSerializer implements EventSerializer {
  private final boolean writeHeaderAndFooter;
  private final BufferedOutputStream stream;
  private static final byte[] footer = ("End Using Flume protobuf " +
    "file").getBytes();
  private static final byte[] header = ("Begin Using Flume protobuf" +
    " file").getBytes();


  private ProtobufSerializer(Context ctx, OutputStream stream) {
    writeHeaderAndFooter = ctx.getBoolean("writeHeaderAndFooter",
      false);
    this.stream = new BufferedOutputStream(stream);
  }

  @Override
111 public void afterCreate() throws IOException {
    if(writeHeaderAndFooter) {
      stream.write(header);
    }
  }

  @Override
```

```
public void afterReopen() throws IOException {

}

@Override
public void write(Event event) throws IOException {
  UsingFlumeEvent.Event.Builder
    builder = UsingFlumeEvent.Event.newBuilder();
  for (Map.Entry<String, String> entry : event.getHeaders()
    .entrySet()) {
    builder.addHeader(UsingFlumeEvent.Header.newBuilder()
      .setKey(entry.getKey())
      .setVal(entry.getValue()).build());
  }
  builder.setBody(ByteString.copyFrom(event.getBody()));
  UsingFlumeEvent.Event e = builder.build();
  stream.write(ByteBuffer.allocate(Integer.SIZE / 8).putInt(e
    .getSerializedSize()).array());
  e.writeTo(stream);
}

@Override
public void flush() throws IOException {
  stream.flush();
}

@Override
public void beforeClose() throws IOException {
  if (writeHeaderAndFooter) {
    stream.write(footer);
  }
}

@Override
public boolean supportsReopen() {
  return false;
}

public static class Builder implements EventSerializer.Builder {

  @Override
  public EventSerializer build(Context context,
    OutputStream outputStream) {
    return new ProtobufSerializer(context, outputStream);
  }
```

```
  }
}
```

任何传递给 HDFS Sink 带有 serializer. 后缀的配置参数，会被传递给序列化器。例如，需要字符集的序列化器可以如下配置：

```
agent.sinks.hdfsSink.serializer.bufferSize = 4096
agent.sinks.hdfsSink.serializer.charset = UTF-8
```

这些参数通过 Context 实例进行传递（只有参数和值被传递——前面的任何东西是不会传递的），且用来配置序列化器。简单来讲，序列化器假设事件主体已经编码为合适的格式；它不会改变且按照该格式写出。

Flume 内置了三类序列化器：TEXT 序列化器、HEADER_AND_TEXT 序列化器和 AVRO_EVENT 序列化器。TEXT 序列化器以事件主体本身写入到文件，可以根据需要在事件之间插入新的一行。HEADER_AND_TEXT 序列化器也做相同的事情，但也包括“key=value”格式的事件报头。默认会在事件之间插入新的一行。这可以通过将 appendNewline 的值设置为 false 来禁用，如下面的展示：

```
agent.sinks.hdfsSink.fileType = DataStream
agent.sinks.hdfsSink.serializer = HEADER_AND_TEXT
agent.sinks.hdfsSink.serializer.appendNewline = false
```

AVRO_EVENT 序列化器使用下面所示的模式，将事件写出为 Avro container 文件 [container-files]。

```
{
  "type": "record",
  "name": "Event",
  "fields": [
    {
      "name": "headers",
      "type": {
        "type": "map",
        "values": "string"
        }
    },
    {
      "name": "body",
      "type": "bytes"
    }
  ]
}
```

MapReduce 作业可以直接使用 Avro MapReduce 模块读取 Avro container 文件。Hive、Pig、Cloudera Impala 等其他系统也支持 Avro container 格式。为了压缩 Avro container 文件，应该使用 Avro 的本地压缩而不是使用 Flume 的压缩流。为了使用 Avro 的本地压缩，将 `compressionCodec` 参数的值设置为对应的压缩编码 `deflate` 或 `snappy`，再将该参数传递给序列化器。Avro container 格式允许用户指定同步标志之间的数据总量，可以使用 `syncIntervalBytes` 设置，该参数指定了以字节为单位。默认大小是 `2048000`（2 MB）。下面展示了一个 Avro 序列化器配置的例子：


```
agent.sinks.hdfsSink.fileType = DataStream
agent.sinks.hdfsSink.serializer = AVRO
agent.sinks.hdfsSink.serializer.syncIntervalBytes = 4096000
agent.sinks.hdfsSink.serializer.compressionCodec = snappy
```

`EventSerializer` 接口是 `flume-ng-core` 工件的一部分，可以像例 3-6 所示的那样，添加到你的序列化器的 *pom.xml* 文件中。

Ingest 格式和 Output 格式的不同 *

当使用 Flume 时混乱的主要来源是格式之间的关系，包括写入到最终目的地的数据格式和接收的数据格式。**接收格式**可以是任何格式——取决于使用的 Source，输入数据转化为 Flume 事件的方式会有所不同。例如，如果通过 RPC 客户端和 RPC Source 接收数据，应用程序将数据转换成 Flume 事件。对于其他 Source，如 spooling directory，一个可插件化的组件完成从原始数据格式到 Flume 事件的转换。

一旦事件在 Flume 里，对于 Flume 事件数据就像一个黑匣子，直到它到达目的 Sink。这条规则的一个异常是什么时候用**拦截器**，拦截器是实际上可以修改事件的组件。我们将在第 6 章讨论拦截器。在写数据到存储系统时，Flume 事件本身需要转换成处理系统使用的格式，处理系统用于从存储系统中读取数据。这是**输出格式**。大多数扮演终端 Sink 作用的 Sink，例如 HDFS、HBase 和 Morphline Solr Sink，都接受可以将 Flume 事件转换为最终目的地格式的插件。

正如你所看到的，在 Source 中有一个从原始格式到 Flume 事件的初始转换，第二个转换成最终目的地格式发生在目的地 Sink 中。如果最初和最终的格式都是相同的，那么只是将原来的格式转换为一个字节数组，然后以预编码的数据写字节数组，这种方式是有意义的。

编写预编码数据的一个很好的例子是 Flume 自带的 AvroEventSerializer 的工作方式。如果传入的事件是 Avro 格式，可以使用 Avro API 将它编码为一个字节数组，且设置值为事件主体。可以使用这种序列化器写入 HDFS，只是简单地写数据，从而使 HDFS 上 Avro container 文件中的原始 Avro 格式的数据可用。

HBase Sink

HBase 累积实时数据已越来越受欢迎，Flume 支持写入到 HBase。Flume 有两类 HBase Sink，*HBase Sink* 和 *Async HBase Sink*，它们的实现略有不同，但配置非常相似。HBase Sink 使用 HBase 客户端 API 来写数据到 HBase。因此，HBase Sink 更容易与 HBase 有线协议变化去同步的。

HBase 客户端 API 是阻塞的，所以 HBase Sink 逐个向 HBase 集群发送事件。另一方面，Async HBase Sink 使用 AsyncHBase API[asynchbase]，该 API 是非阻塞的且使用多线程写数据到 HBase。所以，大多数情况下，Async HBase Sink 有可能给出更好的性能。但是 HBase Sink 支持安全 HBase，这个是 Async HBase Sink 所不支持的。这两类 Sink 都支持序列化器，允许用户使用自定义逻辑转换 Flume 事件为 HBase 友好型对象。在这一节，我们将讨论 HBase sink 的两种类型，以及对于这两类 Sink 如何实现序列化器。

连接到一个或多个 HBase 集群的两类 HBase sink，在 Flume 配置文件或环境变量中的第一个 *hbase-site.xml* 文件中指明了它们的 Quorum。比起 HDFS Sink，HBase Sink 配置参数更少。两类 Sink 公共的参数展示在表 5-5 中。

表5-5 两类HBase sink的配置

参数	默认值	描述
type	-	HBase Sink 的别名是 hbase，也可以使用 FQCN org.apache.flume.sink.hbase.HBaseSink。Async HBase Sink 的别名是 asynchbase，也可以使用 FQCN org.apache.flume.sink.hbase.AsyncHBaseSink
table	-	Sink 写入事件的表。该表必须在 HBase 中已存在——Flume 不会创建表
columnFamily	-	创建的列所在的列族。Flume 也不会创建列族，列族在 HBase 中必须存在

续表

参数	默认值	描述
batchSize	100	每次批处理写入事件的数量
zookeeperQuorum	-	HBase 集群使用的 quorum 中 Zookeeper 服务器的列表
znodeParent	/hbase	Zookeeper quorum 上 HBase 集群使用的父节点
serializer	HBase Sink 的 SimpleHbaseEventSerializer, Async HBase Sink 的 SimpleAsyncHbaseEventSerializer	使用的序列化器的 FQCN。在下一节讨论如何写序列化器的更多细节
serializer.*	-	传递给序列化器的配置参数的列表

115

这两类 Sink 接受这些参数，且关于这些参数有相同的特性。像我们至今见到多的所有其他组件，Sink 的 type 参数可以是别名或者 Sink 的实际 FQCN：HBase Sink 是 hbase 或 org.apache.flume.sink.hbase.HBaseSink，Async HBase Sink 是 asynchbase 或 org.apache.flume.sink.hbase.AsyncHBaseSink。这两类 HBase sinks 只能写入 table 和 columnFamily 参数所指定的一张表和一个列族。

这两类 HBase sink 都是批量写事件。batchSize 参数控制了每个事务写出事件的最大数量（在 Channel 中提供了足够可用的事件，如果 Channel 中有更少的事件，批处理会被认为立即完成）。每次批处理提交 Channel 的每个事务。当且仅当该批次中所有的事件被成功写入 HBase，该事务才被提交。为批处理的大小确定一个数值之前，尝试不同的值是很重要的，因为最合适的值取决于传入的确切数据、HBase 集群拓扑、模式设计、表是如何分裂为不同的 Region 服务器和网络体系结构的。在 HBase 文档或 *HBase：The Definitive Guide* [hbase-book] 中可以查找到 HBase 集群和 HBase 模式的设计细节。

默认情况下，两类 Sink 将在环境变量中查找客户端配置文件 *hbase-site.xml*，并且使用其中的信息连接 HBase。这带来了一个有趣的问题——Agent 中的每个 Sink 看到的是相同的环境变量，这意味着它不可能写数据到多个 HBase 集群。FLume 允许用户通过配置覆盖环境变量。

Sink 接受 zookeeperQuorum 参数，该参数接受逗号分隔的主机名和端口列表。主机名和端口指定为下面的格式：hostname1:port1、hostname2:port1（所有的服务器必须使用相同端口，这是 HBase 要求的）。Sink 也接受另外一个参数 znodeParent，为 HBase 集群使用的父节点（这通常不需要改变）。

116

每种类型的 HBase Sink 都有一些特定的参数。正如前面所讨论的，HBase Sink 有能力写入到安全 HBase 集群。因此，需要在配置中添加安全相关的参数。表 5-6 是 HBase Sink 接受的参数，而 Async HBase Sink 不使用这些参数（与其他组件一样，为 Async HBase Sink 在配置中添加这些参数没有造成危害——只是被忽略）[hbase-security]。

表5-6 HBase Sink安全配置

参数	默认值	描述
kerberosPrincipal	-	用来登录到 KDC 使用的 Kerberos 主体
kerberosKeytab	-	登录到 KDC 的 Kerberos Principal 使用的 keytab 文件的路径

这些参数和 hdfs.kerberosPrincipal、hdfs.kerberosKeytab 参数有完全相同的含义。

除了两类 Sink 共同的参数之外，Async HBase Sink 还接受一个参数：timeout，是整个批处理成功写出到 HBase 的时间周期。如果整个批处理在该时间周期内没有成功写出，那么 Sink 利用 Channel 回滚整个事务，且使这些事件对于该 Sink 和其他 Sink 都是可用的。该参数的默认值是 60000。

例 5-3 中展示了配置有自定义序列化器的 Async HBase Sink 的一个例子，在配置中指定了 zookeeper quorum。

例5-3 Async HBase Sink配置的一个例子

```
agent.sinks = asynchbase
agent.channels = memory

agent.sinks.asynchbase.type = asynchbase
agent.sinks.asynchbase.channel = memory
agent.sinks.asynchbase.zookeeperQuorum = zk1.usingflume.com:2181,
zk2.usingflume.com:2181,zk3.usingflume.com:2181
agent.sinks.asynchbase.znodeParent = /hbase
agent.sinks.asynchbase.table = usingFlumeTable
agent.sinks.asynchbase.columnFamily = usingFlumeFamily
agent.sinks.asynchbase.batchSize = 1000
agent.sinks.asynchbase.timeout = 60000
agent.sinks.asynchbase.serializer = usingflume.ch05.AsyncHBaseDirectSerializer

agent.channels.memory.type = memory
agent.channels.memory.size = 100000
```


用序列化器将 Flume 事件转换成 HBase Put 和 Increment*

HBase Sink 可以在 HBase 中写数据或增量计数器。就像 HDFS Sink，HBase Sink 允许用

户转换 Flume 事件为目的地系统所需的格式：假设为 HBase *Put* 和 *Increment*。这可以借助序列化器来实现。每个 Sink 有自己的序列化器接口；它们彼此之间稍有不同，主要是使用的表示写入 HBase 的 API 略有不同。我们先讨论嵌入到 Async HBase Sink 的序列化器接口。例 5-4 展示了 Async HBase Sink 接口，每个方法后有解释。HBase Sink 的序列化器与此很相似。

例5-4 Async HBase Sink序列化器接口

```
package org.apache.flume.sink.hbase;
public interface AsyncHbaseEventSerializer extends Configurable,
ConfigurableComponent {
  public void initialize(byte[] table, byte[] cf);
  public void setEvent(Event event);
  public List<PutRequest> getActions();
  public List<AtomicIncrementRequest> getIncrements();
  public void cleanUp();
}
```

尽管在接口中不明显，AsyncHbaseEventSerializer（和 HbaseEventSerializer）有能力从 Flume 配置系统接收配置。任何通过配置文件传递的配置参数都会传给序列化器的 configure 方法（继承自 Configurable 接口）。当 Sink 启动时，Sink 会创建序列化器的实例然后调用 initialize 方法，传递给它配置中设置的 table 和 columnFamily 值。

一旦 Sink 从 Channel 中读完一个事件，就调用序列化器的 setEvent 方法并且传给事件。之后，Sink 调用 getActions 方法，会返回 PutRequest 对象列表 [put-request]（Async HBase 相当于 HBase Put），然后调用 getIncrements，会返回 AtomicIncrementRequest 对象列表 [increment-request]（Async HBase 相当于 HBase Increment）。

很明显，每个 Flume 事件可以产生零或多个 HBase Put 和零或多个 HBase Increment。这允许用户非常灵活地解析事件，数据是基于写入多个行或列和多个计数器递增的。例 5-5 所示的 AsyncHBaseDirectSerializer 确实是做到了。这个序列化器查找三个报头：rowKey、incrementColumns 和 payloadColumn。如果 rowKey 报头不存在，那么就忽略事件。如果存在 incrementColumn 报头，它被看作为一个以逗号分隔的字符串列表，按照 rowKey 报头指定的行递增这些列。如果存在 payloadColumn 报头，事件主体写入到 rowKey 报头指定的行的列中。这个序列化器不接受任何配置，但是可以通过配置文件传递这些配置给序列化器。配置传递给序列化器的 configure 方法。

118

例5-5 Async HBase Sink序列化器的例子

```
package usingflume.ch05;

public class AsyncHBaseDirectSerializer
```

```
implements AsyncHbaseEventSerializer {
private byte[] table;
private byte[] columnFamily;
private Event currentEvent;
private static final String ROWKEY_HEADER = "rowKey";
private static final String INCREMENTCOLUMNS_HEADER
  = "incrementColumns";
private static final String PAYLOADCOLUMN_HEADER = "payloadColumn";
private final ArrayList<PutRequest> putRequests
  = Lists.newArrayList();
private final ArrayList<AtomicIncrementRequest> incrementRequests
  = Lists
  .newArrayList();
private byte[] currentRow;
private String incrementColumns;
private String payloadColumn;
private boolean shouldProcess = false;

@Override
public void initialize(byte[] table, byte[] cf) {
  this.table = table;
  this.columnFamily = cf;
}

@Override
public void setEvent(Event event) {
  this.currentEvent = event;
  Map<String, String> headers = currentEvent.getHeaders();
  String rowKey = headers.get(ROWKEY_HEADER);
  if (rowKey == null) {
    shouldProcess = false;
    return;
  }
  currentRow = rowKey.getBytes();
  incrementColumns = headers.get(INCREMENTCOLUMNS_HEADER);
  payloadColumn = headers.get(PAYLOADCOLUMN_HEADER);
  if (incrementColumns == null && payloadColumn == null) {
    shouldProcess = false;
    return;
  }
  shouldProcess = true;
}

@Override
public List<PutRequest> getActions() {
```

119

```
    putRequests.clear();
    if (shouldProcess && payloadColumn != null) {
      putRequests.add(new PutRequest(table, currentRow, columnFamily,
        payloadColumn.getBytes(), currentEvent.getBody()));
    }
    return putRequests;
  }

  @Override
  public List<AtomicIncrementRequest> getIncrements() {
    incrementRequests.clear();
    if (shouldProcess && incrementColumns != null) {
      String[] incrementColumnNames = incrementColumns.split(",");
      for (String column : incrementColumnNames) {
        incrementRequests.add(
          new AtomicIncrementRequest(table, currentRow,
            columnFamily, column.getBytes()));
      }
    }
    return incrementRequests;
  }

  @Override
  public void cleanUp() {
    // Help garbage collection
    putRequests.clear();
    incrementColumns = null;
    payloadColumn = null;
    currentEvent = null;
    currentRow = null;
    table = null;
    columnFamily = null;
  }

  @Override
  public void configure(Context context) {
    // No configuration required
  }

  @Override
  public void configure(ComponentConfiguration conf) {
    // No configuration required
  }
}
```

120

为具有相同行为的 HBase Sink 写一个序列化器是可能的。例 5-6 展示了需要实现的接口。

例5-6 HBase Sink的序列化器接口

```
package org.apache.flume.sink.hbase;
public interface HbaseEventSerializer extends Configurable,
    ConfigurableComponent {
  public void initialize(Event event, byte[] columnFamily);
  public List<Row> getActions();
  public List<Increment> getIncrements();
  public void close();
}
```

HBase Sink 的序列化器没有全局的初始化方法；HBase Put 和 Increment 不需要关于表的信息，这是由 Flume 直接处理的。相反，`HbaseEventSerializer` 中的 `initialize` 方法，和 `AsyncHbaseEventSerializer` 的 `setEvent` 方法做同样的事情，传递下一个将被序列化的事件和需要序列化的列族。`getActions` 和 `getIncrements` 方法和 `AsyncHbaseEventSerializer` 中同名的方法做同样的事情，唯一的区别是使用的 API。`initialize` 方法被调用传给事件后，Sink 立即调用这些方法。`close` 方法可以做任何需要清理的工作。只有当 Sink 即将停止才调用这个方法。

这两类序列化器都是 `flume-ng-hbase-sink` 包的一部分，你可以按照如下所示在 *pom.xml* 的依赖中添加它。

```
<dependency>
  <groupId>org.apache.flume.flume-ng-sinks</groupId>
  <artifactId>flume-ng-hbase-sink</artifactId>
  <version>1.5.0</version>
</dependency>
```

HBase 版本

HBase 0.96 之前，Apache 没发布针对 Hadoop 2 编译的 HBase JARs。Apache Flume 中的 hadoop-2 配置文件默认针对 HBase 0.94.2 编译。不幸的是，这个针对 Hadoop 2 建立的 HBase 版本在 Maven Central 上是不可用的。所以，用户必须使用 Hadoop 2 配置文件构建 HBase，以确保 Flume 针对 HBase 的正确版本建立。默认的配置文件针对 HBase 0.92.1 构建。Flume 也可以使用 hbase-98 配置文件针对 HBase 0.98.x 和 Hadoop 2.4.0 构建。如果使用任何其他版本的 HBase，必须要确保 HBase 的正确版本在环境变量中，Flume *pom.xml* 文件被修改为指向 HBase 的正确版本。

RPC Sink

正如前面章节解释的，集群拓扑需要 Flume Agent 发送数据到其他 Flume Agent。如果从一个 Flume Agent 发送数据到另一个 Agent，那么要使用 RPC Sink。RPC Sink 和对应的 RPC Source 使用相同的 RPC 协议。在第 3 章“Sink-to-Source 通信”一节中有关于 RPC Source 的更多细节。因此，RPC Sink 可以发送数据到 RPC Source。这种方法可以用于将数据从一个 Flume Agent 发送到另一个 Agent。因为 RPC Source 作为服务器监听指定的端口，那么可以实现几个 Flume Agent 发送数据到一个或多个使用对应的 RPC Sink 的 Flume Agent。

Flume 支持两种 RPC 系统，就像在第 3 章“Sink-to-Source 通信”一节解释的：Avro 和 Thrift [thrift_ch5]。Avro 被认为是 Flume 主要的 RPC 格式，所以我们在这一节中将花更多的时间讨论 Avro Sink。我们还将简要查看 Thrift Sink，但因为 Avro Sink 更加成熟，它仍然是 Flume Agent 之间通信的推荐方法。两个 RPC Sink 共享一些基本配置参数，但是 Avro Sink 有更多的功能，因此有更多的配置选项。值得注意的是，Avro Sink 只能发送数据给 Flume Avro Source（或是建立在 Avro 的 Netty RPC 格式上的 Java 服务器），而 Thrift Sink 可以发送数据到 Flume Thrift Source 和任何语言编写的服务器进程，进程有 Thrift 服务器监听发送数据的 Sink 端口。

Avro Sink

Avro Sink 使用 Avro 的 Netty-based RPC 协议发送数据到 Avro Source。它被实现为一个感知事务的 Flume `NettyAvroRpcClient` 的包装器。因此，它们共享一些配置参数。Avro Sink 可以批量发送事件到 Avro Source。这很重要，因为 Avro Source 不控制每个事务写出事件的数量——它将写出整个批次的事件。对于 File Channel，重要的是要以合理的批量写出，因为 Channel 将同步每个事务的数据文件到磁盘。为了避免太多昂贵的同步操作，批量大小必须适度大。

122

大型批量大小的另一个原因是，发送的每个批处理作为 RPC 调用。小批量可能意味着 RPC 开销在有效负载占据了很大比例，这应该避免。最好的批处理大小因为每个部署会有所不同。与 HDFS Sink 类似，它将取决于所使用的硬件、网络，甚至配置（如果在持有 Sink 和持有写入 Source 的两个 Agent 上批量大小都是大的，File Channel 将表现得更好）。

表 5-7 列出了 Avro Sink 的配置参数。

表5-7 Avro Sink配置

参数	默认值	描述
type	-	Avro Sink 的别名是 avro。也可以使用 FQCN org.apache.flume.sink.AvroSink
hostname	-	托管 Agent 机器的主机名，Sink 应该连接的 Source 在该 Agent 上
port	-	Sink 连接的 Source 的监听端口
batch-size	100	每个 RPC 调用发送的事件数量。这也是在单次事务中 Sink 从 Channel 中读取的事件数量
compression-type	-	用来解压传入数据的压缩格式。Flume 1.5.0 只支持 zlib[zlib_ch5] 的压缩格式。如果要接收 zlib 压缩的数据，那么将该参数设置为 deflate
compression-level	6	如果通过 compression-type 参数启用了压缩，那么使用压缩级别有效值是 1-9，数值越高压缩越好
connect-timeout	20000	最初的连接和握手协议完成的超时，以毫秒为单位
request-timeout	20000	整个 RPC 调用成功完成的超时，以毫秒为单位
ssl	false	如果要将加密数据发送给服务器，该参数应该设置为 true
trust-all-certs	false	如果设置为 true，Sink 接受所有的 SSL 证书
truststore	-	使用的 trust store 的路径。如果该值没有设置，默认使用 Java trust store
truststore-password	-	用来打开 trust store 的密码
truststore-type	JKS	Java trust store 类型
reset-connection-interval	-	Sink 从 Source 断开连接并重新连接之后的间隔

Avro Sink的别名是avro，FQCN是org.apache.flume.sink.AvroSink；这两个都可以使用。

hostname参数指定了Sink应该连接的主机的主机名。Source从应该运行Sin的机器上接收数据。port参数指定了Source监听的端口。Sink连接到hostname参数指定主机上的端口。

batch-size参数指定了每次RPC调用发送事件的最大数量（在Channel中提供了足够可用的事件，如果Channel中有更少的事件，批处理会认为立即完成），默认是100。每

次批处理提交 Channel 中的一个事务。

Avro Sink 在通过线路发送数据之前可以压缩数据。为了启用压缩，设置 compression-type 的值为 deflate。可以使用 compression-level 参数控制压缩的级别，该参数的值可以是 1 到 9 之间的任意整数，1 代表最差的压缩，9 代表最好的压缩。值得注意的是，级别越高，在 Sink 压缩和 Source 解压缩所花费的时间越多。默认值是 6。当启用了压缩，Avro Sink 发送压缩数据到的 Avro Source，也应该要设置 compression-type 参数的值为 deflate。压缩发生在批处理级别，所以在批量被发送之前，事件的整个批量立即被压缩。

Avro Sink 支持使用 SSL 进行加密通信。Source 接收数据也必须被配置为从 Avro Sink 接收加密数据。为了启用加密，应设置 ssl 参数的值为 true。默认情况下该参数值为 false。除非另有指定，Flume 将使用 Java 默认的 JSSE 证书授权文件，jssecacerts/ cacerts，来判断 Avro Source 的 SSL 证书是否可信。

124

如果要使用自定义 trust store，设置 truststore 参数的值为各自的 trust store 文件的路径。运行 Agent 的用户应该已经读取访问完文件。truststore-password 参数的值为打开该文件的密码。truststore-type 参数设置了 trust store 的类型。默认为 JKS，但可以是任何支持 Java trust store 的类型。如果 Sink 发送数据时不需要检查 Source 的 SSL 证书，设置 trust-all-certs 值为 true。除了测试该值不应被设为 true。一味地信任 Source 的 SSL 证书意味着任何 Source 都可以读取发送的数据——这影响了 SSL 提供的安全。Java 中更多关于 SSL 的细节，以及如何创建 trust store 都可以在 Oracle Java CAPS 文档 [truststores] 查找到。

大多数情况下，Agent 之间在中间利用负载均衡器相互通信。因为 Agent 之间的连接是黏性的，在负载均衡器之后添加托管 Avro Source 的新 Agent，需要托管 Avro Sink 的 Agent 重新启动，以确保新 Agent 接收连接。为了避免必须这样做，用户可以通过设置 reset-connection-interval 参数，强制 Avro Sink 定期终止它们到 Agent 的连接并重新连接。第 3 章在“Avro Source”一节中展示了 Avro Sink 配置的一个例子。

Thrift Sink

Thrift Sink 可以用来在 Flume Agent 之间使用 Thrift RPC 进行通信。通常 Thrift Sink 与 Avro Sink 工作方式很像，但是它缺少压缩和 SSL 功能。在 Flume Agent 之间推荐使用 Avro RPC 进行通信。Thrift Sink 应该用来写数据到已经运行的 Thrift Source，或许从使用 Thrift RPC 客户端的其他语言写的 RPC 客户端接收数据。

表 5-8 展示了 Thrift Sink 的配置参数。

表5-8 Thrift Sink配置

参数	默认值	描述
type	-	Thrift Sink 的别名是 thrift，也可以使用 FQCN `org.apache.flume.sink.ThriftSink`
hostname	-	托管 Agent 机器的主机名，Sink 应该连接的 Source 在该 Agent 上
port	-	Sink 连接的 Source 的监听端口
batch-size	100	每个 RPC 调用发送的事件数量。这也是在单次事务中 Sink 从 Channel 中读取的事件数量
request-timeout	20000	整个 RPC 调用成功完成的超时，以毫秒为单位
reset-connection-interval	-	Sink 从 Source 断开连接并重新连接之后的间隔
maxConnections	5	Sink 开放给 Source 连接的最大数量

125

除了 `maxConnections` 其余的所有参数都和 Avro Sink 中同名的参数有相同的意义。`maxConnections` 参数控制了每个 Sink 应该开放给 Source 连接的最大数量。如果需要，该参数可以用来调整该 Sink 使用的资源。第 3 章中“Thrift Source”一节展示了 Thrift Sink 配置的一个例子。

Morphline Solr Sink

Morphlines 是一个高度可扩展的 ETL（提取、转换和加载）框架，作为 Kite SDK[morphlines_ch5] 的一部分发布。命令操作独立的记录。命令对输入记录做一个特定的转换，生成零或多个输出记录。几个命令可以连接在一起，且使一个命令的输出是下一个的输入，来执行数据的重量级转换。这样一组命令被称为 *Morphline*。Morphline Solr Sink 整合 Morphline 框架和提供 Sink 的 Flume，可以从 Flume 管道传输数据到这样一个 Morphline（命令的组合）。最重要的 Morphline Solr Sink 的使用，是关于 morphlines 与 Apache Solr[solr] 搜索的集成。

Morphline Solr Sink 通过移动事件主体到记录的 *_attachment_body* 字段，转换每个 Flume 事件为一条记录。报头被填充到字段，命名和报头键一样，值为 Flume 事件设置的对应值。每一批的事件从 Channel 被读取，作为单个 morphline 事务的一部分被处理，所以如果一个事件在 morphline 任何地方失败，那么整个事务被回滚。

126

正如前面所提到的，Morphline Solr Sink 最重要的作用是，实际加载 Flume 事件到 Solr 索引。Morphline 库提供了一个称为 `loadSolr` 的命令，可以加载记录到 Solr。使用该

Sink 加载 Flume 事件到 Solr，使用的 morphline 配置文件必须将 `loadSolr` 列为最后一个命令。每个 Flume 事件经过每个配置命令的转换，并最终通过 loadSolr 的命令被加载到 Solr。在加载可搜索的文本到 Solr 之前，转换可以用来解析文本为复杂格式表示的数据，例如 PDF、XML 等。在我们进入详细谈论 Solr 的用例之前，让我们看看表 5-9 中 Morphline Solr Sink 的配置参数。

表5-9 Morphline Solr Sink配置

配置	默认值	描述
type	-	Morphline Solr Sink 的别名是 `morphlinesolr`，也可以使用 FQCN `org.apache.flume.sink.solr.morphline.MorphlineSolrSink`
morphlineFile	-	从 morphline 读取文件的完整路径
morphlineId	-	如果在相同文件中有多个 morphline，Sink 应该使用来处理事件的 morphline 的 ID
batchSize	1000	每次批处理写入事件的数量
batchDurationMillis	1000	批处理被认为完成（如果有 `loadSolr` 命令，且加载到 Solr）之后的毫秒值时间

Morphline Solr Sink 的别名是 `morphlinesolr`，FQCN 是 `org.apache.flume.sink.solr.morphline.MorphlineSolrSink`，这两个都可以使用。`morphlineFile` 参数指定了 Sink 使用的 morphline 读取文件的路径。

Morphlines 框架允许 morphline 文件包含多个 morphline，每个用一个 ID 表示。如果文件中不止有一个 morphline，`morphlineId` 参数指定了文件使用的 morphline 的 ID。

`batchSize` 参数指定了一个事务中，Sink 从 Channel 读取事件的最大数量（在 Channel 中读取完最后一个事件，如果 Channel 中有更少的可用事件，批处理被认为立即完成）。这也是一个 morphline 会话中事件的总数量。在固定的持续时间之后也可以关闭 morphline 会话。该持续时间为 `batchDurationMillis` 参数指定的毫秒值。

127

为了在 Solr 上索引数据，每个插入到 Solr 的记录需要一个唯一的键来识别记录。对于 Solr 集合 [solr-unique-key]，作为键的字段使用 *schema.xml* 文件中 `<uniqueKey>` 标签所指定的值。在这个例子中，我们假设指定的唯一键的字段是 `id`。我们首先看一下用来加载数据到 Solr 的 morphline 文件（在例 5-7 中展示）。

例5-7 一个morphline配置文件

```
SOLR_LOCATOR: {
  collection : usingFlumeCollection
  zkHost : "com.usingflume.solrZk:2181/solr"
}

morphlines : [
  {
                id : usingFlumeMorphline

    importCommands : ["org.kitesdk.**", "org.apache.solr.**"]
    # Convert the event into a UTF-8 encoded string
commands : [
                {
                        readClob {
                           charset : UTF-8
                        }
                }

    # Generate a UUID that can be used for the Solr unique key field
    {
      generateUUID {
          field : id
      }
    }

    # Load the record into Solr
    {
      loadSolr: {
          solrLocator : ${SOLR_LOCATOR}
      }
    }
        ]
  }
]
```

Morphline Solr Sink 启动 morphline 会话，从 Channel 中读取事件，并将它们转换为记录，
128 事件主体设置为 _attachment_body 字段。然后 Sink 依次传递事件到 morphline。每条记录传递给 readClob 命令，用来读取主体，并将它转换为 UTF-8 字符串，再将该字符串设置为 message 字段的值。

generateUUID 生成 UUID，并将它设置为 id 字段的值，为 Solr 使用的唯一键（前面讨论过）。然后为了索引事件通过 loadSolr 命令加载到 Solr 索引。为了扩展 loadSolr 命令也允许加载数据到 SolrCloud 集群。

添加更多的 Morphline Sink 可以增加加载到 Solr 的事件数量，直到 Solr 中可用容量用完。一旦读取事件的总数量达到了批量大小或时间，因为批量启动后时间超过 batchDurationMillis 参数的值（或者，像其他所有的 Sink、Channel 中没有事件了），Sink 提交完 Sink 事务接着提交 morphline 事务。任何失败将会导致 morphline 事务和 Channel 有关的事务回滚。

Sink 可以配置为使用配置文件，类似于例 5-8 展示的例子。

例5-8 Morphline Solr Sink配置的一个例子

```
agent.sinks = morphline
agent.channels = solrChannel

agent.sinks.morphline.type = morphlinesolr
agent.sinks.morphline.morphlineFile = /etc/morphline/conf/morphline.conf
agent.sinks.morphline.morphlineId = usingFlumeMorphline
agent.sinks.morphline.batchSize = 1000
agent.sinks.morphline.batchDurationMillis = 2500
agent.sinks.morphline.channel = solrChannel

agent.channels.solrChannel.type = memory
agent.channels.solrChannel.capacity = 100000
```

这个配置文件描述了一个 Agent，该 Agent 通过 Avro Source 从其他 Flume Agent 或应用程序接收数据，然后通过两个 Memory Channel 复制数据到 HDFS 和 Solr。这是一个设置的典型例子，用来保证使用 Flume 实时地搜索索引。

事件被复制到 HDFS 是很常见的，Source 将相同事件写到 Channel 以供给该 Sink 和 HDFS Sink 也是常见的。为了合并 Solr 分片和用 Solr 调解 HDFS 上的数据，HDFS 上的数据可以用作 MapReduce 作业的输入 [solr-on-hdfs]。

在这个 Agent 中，morphline sink 使用 morphline 配置文件 */etc/morphline/conf/morphline.conf*，且使用 FlumeMorphline 读取 morphline 的 ID。Sink 使用 1000 的批量大小，但是 129 如果花费多于 2.5 秒的时间达到这个数量，Sink 就提交该批量操作。

应该使用第 8 章“部署自定义代码”一节展示的 plug-in 部署系统，将 morphline JARs 和 morphline 配置文件添加到 Flume 的环境变量中。

Elastic Search Sink

Flume 也可以使用 Elastic Search Sink 实时加载数据到 Elastic Search 来创建索引。Elastic Search Sink 需要用户安装 Elastic Search 客户端 JAR 的正确版本，且在环境变量中安装

依赖，因为 Elastic Search 需要客户端 API 版本来匹配服务器版本。该 Sink 在每天午夜动态创建 Elastic Search 集群上的新索引。默认情况下，该 Sink 和 Kibana UI[kibana] 是兼容的。

表 5-10 展示了 Elastic Search Sink 的配置参数。

表5-10 Elastic Search Sink配置

参数	默认值	描述
type	-	Elastic Search Sink 的别名是 elasticsearch，也可以使用 FQCN org.apache.flume.sink.elasticsearch.ElasticSearchSink
hostNames	-	Elastic Search 主机以逗号分隔的主机名和端口列表
batchSize	100	每个事务中事件的最大数量
ttl	-	过期文件删除的时间周期，以天为单位
indexName	flume	索引名字的前缀
indexType	logs	Elastic Search 上索引的类型
clusterName	elasticsearch	要写入的 Elastic Search 集群的名字
serializer	ElasticSearchLogStashEventSerializer	实现 ElasticSearchEventSerializer 或 ElasticSearchIndexRequestBuilderFactory 类的完整类别名称。下一节将讨论如何使用该 Sink 以自定义格式写数据
serializer.*	-	传递给序列化器的参数

130 Elastic Search Sink 可以使用 elasticsearch 别名或 FQCN org.apache.flume.sink.elasticsearch.ElasticSearchSink 来初始化。hostNames 配置参数的值指定了 Sink 写数据的 Elastic Search 服务器列表。主机名被指定为 hostname:port 格式的逗号分隔的列表。如果集群使用默认端口 9300，那么可以忽略 port 参数。batchSize 参数控制每个事务中事件的最大数量，如果 Channel 中事件可用，该参数也指定每个事务 Sink 移除事件的最大数量。

Elastic Search 允许用户在索引文件上设置逾时日期。为了自动删除这些文件，用户可以配置文件的 ttl 参数（time to live，以天为单位）。

indexName 参数指定了索引名字的前缀。序列化器负责决定每个事件该去往哪个索引，但是它被期望使用这个参数创建索引。当使用默认的序列化器时（或者如果自定义序列化器继承自 AbstractElasticSearchIndexRequestBuilderFactory 类），Sink 就每

天创建一个新索引，使用 indexName 参数值为前缀，当前的日期为后缀。索引名字是 <indexName>-DAY, MONTH DD, YYYY 的格式。该参数的默认值是 flume，所以一个索引名字的示例可以是 flume-Monday, September 22, 2014。用户也可以指定索引中数据的类型为 indexType 参数的值。该类型可以被 Elastic Search 索引，可以用来搜索文件的指定类型。clusterName 参数的值指定了要写入索引的 Elastic Search 集群的名字。

就像 Morphline Solr Sink 一样，Elastic Search Sink 可以配置用来索引数据（在 Elastic Search 的情况下），而 HDFS Sink 并行写数据到 HDFS。例 5-9 的配置文件描述了一个 Elastic Search Sink，配置写入到索引，索引名字以 usingFlume 开始，索引类型是 books。

例5-9 Elastic Search配置示例

```
agent.sinks = elasticsearch
agent.channels = esChannel

agent.sinks.elasticsearch.type = elasticsearch
agent.sinks.elasticsearch.hostNames = es1.usingflume.com:5400,es2.usingflume.com,
es3.usingflume.com
agent.sinks.elasticsearch.batchSize = 1000
agent.sinks.elasticsearch.ttl = 2
agent.sinks.elasticsearch.indexName = usingFlume
agent.sinks.elasticsearch.indexType = books
agent.sinks.elasticsearch.clusterName = usingFlumeCluster
agent.sinks.elasticsearch.serializer =
usingflume.ch05.HeaderAndBodyIndexRequestBuilderFactory
agent.sinks.elasticsearch.serializer.writeHeaders = true
agent.sinks.elasticsearch.channel = esChannel

agent.channels.esChannel.type = memory
agent.channels.esChannel.capacity = 100000
```


自定义数据格式 *

使用实现了 ElasticSearchIndexRequestBuilderFactory 的插件化的类，为 Elastic Search 创建一个 IndexRequest，来自定义写出到 Elastic Search 的数据格式也是可能的。这是添加事件到 Elastic Search 的首选方法。也有一个弃用的方法可以实现这个，该方法实现了 ElasticSearchEventSerializer；尽管将来可能移除这个方法，但现在仍是支持的。

为了写数据到 Elastic Search 时使用自定义类序列化数据，需要指定该类的 FQCN 为 serializer 参数的值。该类必须实现 ElasticSearchIndexRequestBuilderFactory 或

ElasticSearchEventSerializer。如果这个参数没有指定值，事件会被序列化为 Kibana 友好型格式。使用 serializer. 前缀可以传递任何参数给序列化器。例 5-10 展示了 ElasticSearchIndexRequestBuilderFactory 接口。

例5-10 ElasticSearchIndexRequestBuilderFactory接口

```
package org.apache.flume.sink.elasticsearch;
public interface ElasticSearchIndexRequestBuilderFactory extends Configurable,
    ConfigurableComponent {
  IndexRequestBuilder createIndexRequest(Client client,
      String indexPrefix, String indexType, Event event) throws IOException;
}
```

为了更简单地实现这个接口，Flume 提供了一个抽象类，该类提供了一个简单的实现，AbstractElasticSearchIndexRequestBuilderFactory，能像前面一节介绍的那样每天创建一个新的索引。继承自该类的序列化器将每天自动创建一个新的索引（当然是在该类没有覆盖 createIndexRequest 方法来改变行为的情况下）。

为了使用自动创建索引的功能，而且也能以某些自定义的格式插入数据，那么继承自该类的一个类的 FQCN 应该设置为 serializer 参数的值。为了定制插入到索引中的数据，需要实现 prepareIndexRequest 方法。例 5-11 可以用来将事件插入到 Elastic Search 索引。
132 如果配置了写报头，那么报头插入它们各自的键和体（这个序列化器假设报头没有 body 键）。如果没有配置写 header，那就只是简单地写入体。

例5-11 IndexRequestBuilderFactory实现的一个例子

```
package usingflume.ch05;

public class HeaderAndBodyIndexRequestBuilderFactory extends
  AbstractElasticSearchIndexRequestBuilderFactory {

  private String CONFIG_WRITE_HEADERS = "writeHeaders";
  // By default, don' t write the headers.
  private boolean DEFAULT_WRITE_HEADERS = false;
  private boolean writeHeaders = false;
  private static final String BODY_HEADER = "body";

  public HeaderAndBodyIndexRequestBuilderFactory() {
    this(FastDateFormat.getDateInstance(FastDateFormat.FULL));
  }

  protected HeaderAndBodyIndexRequestBuilderFactory(
    FastDateFormat dateFormat) {
    super(dateFormat);
  }
```

```
  @Override
  public void configure(Context context) {
    writeHeaders = context.getBoolean(CONFIG_WRITE_HEADERS,
      DEFAULT_WRITE_HEADERS);
  }

  @Override
  public void configure(
    ComponentConfiguration componentConfiguration) {
  }

  @SuppressWarnings("unchecked")
  @Override
  protected void prepareIndexRequest(
    IndexRequestBuilder indexRequestBuilder,
    String indexName, String indexType, Event event)
    throws IOException {
    indexRequestBuilder.setIndex(indexName).setType(indexType);
    if (writeHeaders) {
      Map source = (Map) event.getHeaders();
      source.put(BODY_HEADER,
        new String(event.getBody(), Charsets.UTF_8));
      indexRequestBuilder.setSource((Map<String, Object>) source);
    } else {
      indexRequestBuilder.setSource(event.getBody());
    }
  }
}
```

133

例 5-9 展示了配置的序列化器的一个例子。

这里展示的所有序列化器基本类别都是 flume-ng-elasticsearch-sink 工件的一部分。为了在你创建的序列化器中包含它，添加以下内容到 *pom.xml* 文件的依赖部分。

```
<dependency>
  <groupId>org.apache.flume.flume-ng-sinks</groupId>
  <artifactId>flume-ng-elasticsearch-sink</artifactId>
  <version>1.5.0</version>
</dependency>
```

Elastic Search Client API

因为 Elastic Search 需要 Flume 使用完全相同 Elastic Search 版本作为集群，Flume 没有打包 Elastic Search 客户端库。用户必须在 Agent 的环境变量中部署库和它们的所有依赖，以使得 Sink 能够写数据。

其他 Sink：Null Sink、Rolling File Sink 和 Logger Sink

Flume 内置了多种 Sink 可以用于测试目的。但是我们不会详细研究它们，只是了解一下它们的功能和配置。Null Sink 是非常简单的 Sink，用于从 Channel 读取事件并删除它们。这类 Sink 的目的是检测其余 Agent 的功能和性能。Null Sink 批量地从 Channel 移除事件。每次批处理的大小被 batchSize 参数控制。该 Sink 也能在每次删除一定数量事件时更新日志文件。logEveryNEvents 参数控制了 Sink 记录到日志文件后的事件数量。表 5-11 列出了 Null Sink 的几个参数。

表5-11 Null Sink配置

参数	默认值	描述
type	-	Null Sink 的别名是 null，也可以使用 FQCN org.apache.flume.sink.NullSink
batchSize	100	在事务提交之前 Sink 从 Channel 移除的事件的数量
logEveryNEvents	10000	Sink 记录到日志文件之后的事件数量

Rolling File Sink 写事件到本地文件系统的文件上。sink.directory 参数指明了事件写入的目录。该 Sink 也支持和 HDFS Sink 一样的序列化器。sink.serializer 配置参数指明了序列化器。就像 HDFS Sink，使用 sink.serializer. 前缀可以将配置传递给序列化器。该 Sink 可以基于一个时间间关闭文件。sink.rollInterval 参数指明了这个间隔(以秒为单位)。该类 Sink 也支持批处理每个事务的事件。sink.batchSize 参数指明了批处理的大小。表 5-12 展示了 Rolling File Sink 的配置参数。

表5-12 Rolling File Sink配置

参数	默认值	描述
type	-	Rolling File Sink 的别名是 file_roll，也可以使用 FQCN org.apache.flume.sink.RollingFileSink
sink.batchSize	100	在事务提交之前 Sink 从 Channel 移除的事件的数量
sink.directory	-	Sink 应该写入事件的目录
sink.rollInterval	30	文件应该关闭之后的时间间隔，以秒为单位
sink.serializer	TEXT	用来写事件的序列化器。该值可以是内置的序列化器的别名，或者自定义类的 FQCN
sink.serializer.*	-	传递给序列化器的配置参数

Logger Sink 记录到 *log4j.properties* 文件中配置的 Flume Agent 的 log4j 日志文件。该类 Sink 所有的配置都从 *log4j.properties* 文件中获取，且它不需要任何其他的参数（除了 type 和 channel 参数）。表 5-13 展示了 Logger Sink 的配置参数。

表5-13 Logger Sink配置

135

参数	默认值	描述
type	-	Logger Sink 的别名是 logger，也可以使用 FQCN org.apache.flume.sink.LoggerSink

编写自定义 Sink*

在许多情况下，用户将很可能需要编写自定义的 Sink。这种情况下的一个例子是，如果用户需要将数据写入专有数据存储或自定义格式。在本节中，我们将介绍 Sink 的基本工作流，以及编写一个自定义 Sink 的例子。

自定义 Sink 必须实现 Sink 接口，如果 Sink 需要接受来自配置系统的配置，那么可以选择实现 Configurable 接口。为了更好地理解如何编写一个 Sink，很重要的是要理解 Flume 框架如何与 Sink 交互。

当 Agent 启动时，框架检查来确保每个 Sink 有一个指定的 type，且有一个 Channel 参数的值代表了 Agent 中已经存在的配置合适的 Channel。然后 Sink 被实例化且配置传递给它的 configure 方法。如果 configure 方法失败且抛出异常，那么该 Sink 会从 Agent 中移除且删除它的实例。一旦 Sink 成功配置，它就连接到应该从其中读取事件的 Channel。

从那时起，Sink 由一个 Sink 运行器管理。Sink 运行器只是一个负责运行该 Sink 的线程。框架通过调用 start 方法框架来启动 Sink。如果 start 方法失败，该框架将反复重试启动 Sink。

一旦 Sink 启动，Sink 运行器线程就循环调用 process 方法。该方法负责从 Channel 读取数据并写出到下一阶段或最终目的地。每个 process 调用必须处理整个事务——启动事务，从 Channel 读取事件，提交或回滚事务，并最终关闭事务。如果 Channel 不包含任何 Sink 可以移除的事件，process 方法必须返回 Status.BACKOFF，这使得 Sink 运行器只在一个时间间隔后重试，增加每个连续时间 Sink 返回 Status.BACKOFF 时。这种机制减缓了 Sink 在没有足够的数据。如果 Sink 成功，它必须返回 Status.READY 状态，且运行器将立即再次调用 process 方法。

当从 Channel 读取数据或写入到目的地时，如果遇到一些异常，Sink 必须返回 Status.BACKOFF 状态或抛出异常以报告失败。这使 Sink 运行器减慢速度，如果不能清除下流

136

Agent或最终目的地的数据，这也是避免发送数据到下一阶段的调节机制。在Flume框架中，为了正确地运行，process方法必须是线程安全的。例5-12给出一个自定义Sink的例子。

例5-12 一个自定义Sink的例子

```
package usingflume.ch05;

public class S3Sink extends AbstractSink implements Configurable {
  private String objPrefix;
  private final AtomicLong suffix = new AtomicLong(System
    .currentTimeMillis());
  private String awsAccessKeyId;
  private String awsSecretKey;
  private String bucket;
  private int batchSize;
  private String endPoint;
  private int bufferSize;
  private AmazonS3 connection;

  // 64K buffer
  public static final int DEFAULT_BUFFER_SIZE = 64 * 1024;
  public static final int DEFAULT_BATCH_SIZE = 1000;
  public static final String DEFAULT_OBJECT_PREFIX = "flumeData-";

  @Override
  public void start() {
    // Set up Amazon S3 client
    AWSCredentials credentials = new BasicAWSCredentials(
      awsAccessKeyId, awsSecretKey);
    ClientConfiguration config = new ClientConfiguration();
    config.setProtocol(Protocol.HTTP);
    connection = new AmazonS3Client(credentials, config);
    connection.setEndpoint(endPoint);
    if (!connection.doesBucketExist(bucket)) {
      connection.createBucket(bucket);
    }
    super.start();
  }

  @Override
  public synchronized void stop() {
    super.stop();
  }

  @Override
```

```
public Status process() throws EventDeliveryException {
  Status status = Status.BACKOFF;
  Transaction tx = null;
  final ByteArrayOutputStream data
    = new ByteArrayOutputStream(bufferSize);
  try {
    tx = getChannel().getTransaction();
    tx.begin();
    int i = 0;
    for (; i < batchSize; i++) {
      Event e = getChannel().take();
      if (e == null) {
        break;
      }
      byte[] body = e.getBody();
      data.write(
        ByteBuffer.allocate(Integer.SIZE / 8).putInt(body.length).array());
      data.write(body);
    }
    if (i != 0) {
      connection.putObject(bucket,
        objPrefix + suffix.incrementAndGet(),
        new ByteArrayInputStream(data.toByteArray()),
        new ObjectMetadata());
      status = Status.READY;
    }
    tx.commit();
  } catch (Exception e) {
    if (tx != null) {
      tx.rollback();
    }
    throw new EventDeliveryException("Error while processing " +
      "data", e);
  } finally {
    if (tx != null) {
      tx.close();
    }
  }
  return status;
}

@Override
public void configure(Context context) {
  awsAccessKeyId = context.getString("awsAccessKeyId");
  Preconditions.checkArgument(!Strings.isNullOrEmpty(awsAccessKeyId),
```

```
                                "AWS Key Id is required");

    awsSecretKey = context.getString("awsSecretKey");
    Preconditions.checkArgument(!Strings.isNullOrEmpty(awsSecretKey),
                                "AWS Secret Key must be specified");

    bucket = context.getString("bucket");
    Preconditions.checkArgument(!Strings.isNullOrEmpty(bucket),
                                "Bucket name must be specified");

    endPoint = context.getString("endPoint");
    Preconditions.checkArgument(!Strings.isNullOrEmpty(endPoint),
                                "Endpoint cannot be null");

    batchSize = context.getInteger("batchSize", DEFAULT_BATCH_SIZE);
    objPrefix = context.getString("objectPrefix", DEFAULT_OBJECT_PREFIX);
    bufferSize = context.getInteger("bufferSize", DEFAULT_BUFFER_SIZE);
  }
}
```


该例子描述了写数据到 Amazon S3 bucket 的 Sink[s3]。该 Sink 是终端 Sink 的一个例子，用于写数据到存储系统。它从 Channel 读取数据且批量写出到 Amazon S3 的文本文件。

S3 Sink 继承自 AbstractSink 类实现了 Configurable 接口。当 Agent 启动时，框架通过调用 configure 方法配置 Sink。该 Sink 创建需要登录到 S3 服务的认证。Sink 连接的 bucket 名字和端点也从配置文件中读取。因为需要这些参数，这个方法使这些通过配置文件传递的参数值有效且不为空。可以选择性地传递例如 batchSize 参数和对象名前缀。如果没有传递它们，那就使用默认值。

然后 Flume 框架通过调用 start 方法启动 Sink。在这个方法中，Sink 建立需要的连接和 bucket 信息。

一旦 Sink 启动，Sink 运行器就循环调用 process 方法。在 process 方法中，Sink 通过调用 Channel 的 getTransaction 方法创建事务。使用 Transaction.begin 方法启动事务，然后 Sink 通过调用 take 方法尽可能多地以批量的大小从 Channel 中读取事件（或者直到 Channel 中没有更多可用的事件——此时 take 方法返回 null），并将它们写出到 S3。Sink 也会给每个事件预先添加上每个事件的长度，以确保我们可以从相同文件中读取多个事件。一旦数据写入成功，事务就会提交；否则事务回滚。最终，必须关闭事务（这应该总是在最后一个块中完成的）。如果没有可读的事件，Sink 运行器被要求在 process 方法中通过返回 Status.BACKOFF 来减慢速度；否则，该方法返回 Status.READY。如果抛出任何异常，Sink 运行器就捕获异常且自动后退。

最终，当 Agent 停止时，就调用 `stop` 方法。`stop` 方法必须关闭任何网络连接，关闭任何已经创建的线程或线程池。

使用第 8 章“部署自定义代码”一节展示的插件化部署框架，自定义 Sink 可以被部署到一个 Flume Agent。 139

总结

在本章中，我们讨论了 Sink 的基本概念和 Flume 内置的不同种类的 Sink，包括 RPC Sink 和推送数据到存储和索引系统的 Sink。我们也讨论了如何编写自定义 Sink，以移动数据到 Flume 目前不支持的系统。

在下一章中，我们将讨论 Flume 体系结构中的其他组件：拦截器、Channel 选择器、Sink 组和 Sink 处理器，这些组件使得 Flume 更具灵活性和可扩展性。

参考文献

- [tz-list] IANA Time Zone Data, *http://www.iana.org/time-zones*
- [kerberos] MIT Kerberos, *http://web.mit.edu/~kerberos/*
- [impersonation] Hadoop impersonation configuration, *http://bit.ly/1ARZHsp*
- [container-files] Avro container file specification, *http://bit.ly/1ARZKoe*
- [asynchbase] AsyncHBase, *https://github.com/OpenTSDB/asynchbase*
- [hbase-book] HBase: The Definitive Guide, O’Reilly Media
- [hbase-security] HBase client-side security configuration, *http://bit.ly/1AS0qKj*
- [put-request] AsyncHBase PutRequest, *http://bit.ly/1ARZVjm*
- [increment-request] AsyncHBase AtomicIncrementRequest, *http://bit.ly/1ARZYLX*
- [thrift_ch5] Apache Thrift, *http://thrift.apache.org/docs/concepts*
- [zlib_ch5] zlib compression library, *http://www.zlib.net*
- [truststores] Oracle Java trust store, *http://bit.ly/1AS0bPj*
- [morphlines_ch5] Morphlines: Kite SDK, *http://bit.ly/1AS0g5I*
- [solr] Apache Solr, *https://lucene.apache.org/solr/*
- [solr-unique-key] Apache Solr unique key, *http://bit.ly/1AS0fi8*
- [solr-on-hdfs] Apache Solr on HDFS, *http://bit.ly/1AS0jhN*
- [kibana] Kibana Elastic Search UI, *http://www.elasticsearch.org/overview/kibana/*
- [s3] Amazon S3, *http://aws.amazon.com/s3/*

第 6 章 141

拦截器、Channel选择器、Sink组和Sink处理器

正如前面章节讨论的，Flume Agent 最重要的组件是 Source、Channel 和 Sink。另外，Flume Agent 还有一些使 Flume 更加灵活的组件。本章将讨论拦截器、Channel 选择器、Sink 组和 Sink 处理器。

拦截器

拦截器（Interceptor）是简单插件式组件，设置在 Source 和 Source 写入数据的 Channel 之间。Source 接收到的事件在写入到对应的 Channel 之前，拦截器都可以转换或删除这些事件。每个拦截器实例只处理同一个 Source 接收到的事件。拦截器可以基于任意标准删除或转换事件，但是拦截器必须返回尽可能多（或尽可能少）的事件，如同原始传递过来的事件。

在一个链条中，可以添加任意数量的拦截器去转换从单个 Source 中来的事件。Source 将同一个事务的所有事件传递给 Channel 处理器，进而传递给拦截器链条，然后事件被传递给链条中的第一个拦截器。通过拦截器转换事件产生的一系列事件，传递到链条的下一个拦截器，以此类推。链条中最后一个拦截器返回的最终事件列表写入到 Channel 中。

因为拦截器必须在事件写入 Channel 之前完成转换操作，只有当拦截器已成功转换事件后，RPC Source（和任何其他可能产生超时的 Source）才会响应发送事件的客户端或 Sink。因此，在拦截器中进行大量重量级的处理并不是一个好主意。如果拦截器中的处 142
理是重量级的、耗时的，那么需要相应地调整超时时间属性。

所有拦截器通用的唯一配置参数是 type 参数，该参数必须是拦截器的别名或者 Builder 类的全完全限定类名（FQCN），该 Builder 类用于创建拦截器。正如前面提到的，可以设置任意数量的拦截器连接到单个的 Source。

拦截器是需要命名的组件，每个拦截器实例必须限定一个名字。为了给 Source 添加拦截器，需要列出 Source 应该连接的拦截器名字，这些拦截器就是 Source 应该连接到 Source 配置中 interceptors 参数的值代表的拦截器。源配置中以 interceptors. 开头的、后面跟着拦截器名称和参数的所有值都传递给拦截器。下面的实例展示了如何配置拦截器：

```
agent.sources.avroSrc.interceptors = hostInterceptor timestampInterceptor
agent.sources.avroSrc.interceptors.hostInterceptor.type = host
agent.sources.avroSrc.interceptors.hostInterceptor.preserveExisting = true
agent.sources.avroSrc.interceptors.timestampInterceptor.type = timestamp
agent.sources.avroSrc.interceptors.timestampInterceptor.preserveExisting =
false
```

Flume 内置了很多拦截器，并且会定期地添加一些拦截器。本节将讨论一些 Flume 内置的、经常使用的拦截器。

时间戳拦截器

Flume 中一个最常用的拦截器是时间戳拦截器（timestamp interceptor），该拦截器将时间戳插入到 Flume 的事件报头中，附带的键 timestamp 是 HDFS Sink 用来分桶的报头。除非 preserveExisting 参数设置为 false，如果时间戳报头已经存在，拦截器都会替换该时间戳报头。添加时间戳拦截器，使用别名 timestamp。该拦截器经常用在第一层 Agent，用于从客户端接收数据，所以，HDFS Sink 可以使用时间戳进行分桶。时间戳拦截器的配置参数如表 6-1 所示。

表6-1 时间戳拦截器配置

参数	默认值	描述
type	-	类型名称是 timestamp。也可以使用 Builder 类的 FQCN，org.apache.flume.interceptor.TimestampInterceptor$Builder
preserveExisting	false	如果设置为 true，若事件中报头已经存在，不会替换时间戳报头的值

143 下面是 Source 连接到时间戳拦截器的 Agent 样例：

```
agent.sources.avro.interceptors = timestampInterceptor
agent.sources.avro.interceptors.timestampInterceptor.type = timestamp
agent.sources.avro.interceptors.timestampInterceptor.preserveExisting =
false
```

主机拦截器

主机拦截器（host interceptor）插入服务器的 IP 地址或者主机名，Agent 将这些内容注入到 Flume 的事件报头中。事件报头中的键使用 hostHeader 配置，默认值是 host。如果拦截器经过配置使用的事件报头在事件中存在，若 preserveExisting 参数设置为 false（或者不特意设定该参数），那么将会替换事件报头。将 useIP 参数设置为 false，插入的主机名会替换 IP 地址。主机拦截器的配置参数如表 6-2 所示。

表6-2 主机拦截器配置

参数	默认值	描述
type	-	类型名称是 host。也可以使用 Builder 类的 FQCN，org.apache.flume.interceptor.HostInterceptor$Builder
hostHeader	host	事件头的键，用于插入 IP 地址或主机名
useIP	true	如果设置为 true，host 键插入的值是 IP 地址
preserveExisting	false	如果设置为 true，若事件报头已经存在于事件中，不替换 host 报头的值

下面的样例展示了配置主机拦截器将主机名写入事件报头，如果事件报头已经存在，则不替换其值。

```
agent.sources.avro.interceptors = hostInterceptor
agent.sources.avro.interceptors.hostInterceptor.type = host
agent.sources.avro.interceptors.hostInterceptor.useIP = false
agent.sources.avro.interceptors.hostInterceptor.preserveExisting = true
```

静态拦截器

静态拦截器简单地将固定报头的键和值插入拦截的每个事件中。报头键和值是可配置的，键的默认值是 key，值的默认值是 value。该拦截器也有 preserveExisting 参数，在键已经存在的情况下，该参数用于维护报头中已经存在的键值对。该参数默认值是 true（不像时间戳拦截器和主机拦截器）。静态拦截器的配置参数如表 6-3 所示。

144

表6-3 静态拦截器

参数	默认值	描述
type	-	类型名是 static。org.apache.flume.interceptor.HostInterceptor$Builder 是 Builder 类的 FQCN，也可以使用
key	key	报头的键
value	value	针对报头中键插入的值
preserveExisting	true	如果设置为 false，若事件报头中键已经存在，特定键的值被 value 参数的值替换

下面配置将触发报头中包含以 book 为键、以 usingFlume 为值的事件：

```
agent.sources.avro.interceptors = staticInterceptor
agent.sources.avro.interceptors.hostInterceptor.type = static
agent.sources.avro.interceptors.staticInterceptor.key = book
agent.sources.avro.interceptors.staticInterceptor.value = usingFlume
agent.sources.avro.interceptors.staticInterceptor.preserveExisting = false
```

正则过滤拦截器

正则过滤拦截器可以用于过滤事件。过滤操作基于配置文件中提供的正则表达式(regex)。每个正则过滤拦截器将事件体转换为 UTF-8 的字符串，并根据提供的正则表达式匹配字符串。一旦匹配成功，则允许或者丢掉事件。可以配置拦截器丢掉匹配正则的事件或者允许匹配正则的事件通过。

很多这样的拦截器可以添加到单个 Source 实现更复杂的过滤，只有匹配特定模式的事件才会被写入 Channel，如果这些事件也匹配了其他模式，那么就会丢掉这些事件。正则过滤拦截器可以用于确保只有最重要的事件才能被传递，通过 Flume Agent 聚合数据并推送到 HDFS 或 HBase。表 6-4 罗列了正则过滤拦截器相关的配置参数。

145

表6-4 正则过滤拦截器配置

参数	默认值	描述
type	-	类型名字是 regex_filter。org.apache.flume.interceptor.RegexFilteringInterceptor$Builder 是 Builder 类 FQCN，也是可以使用的
regex	.*	匹配事件体的正则表达式
excludeEvents	false	如果设置为 true，匹配上正则的事件会丢弃掉；反之，只有匹配上正则的事件才会通过

当事件体匹配上正则后，excludeEvents 参数决定需要做什么。如果该参数设置为 true，所有匹配上正则的事件会被丢弃掉，所有剩余的事件都允许通过。如果该参数设置为 false，匹配正则的事件才允许通过，其他的都需要丢弃掉。

下面的配置展示了两个正则过滤拦截器的集合，消息中出现单词“flume”，并且没出现单词“DEBUG”的事件允许通过。这样的组合可以用于确保只有从特定 Source 来的，并且匹配一些标准的消息才能通过：

```
agent.sources.avroSrc.interceptors = include exclude
agent.sources.avroSrc.interceptors.include.type = regex_filter
agent.sources.avroSrc.interceptors.include.regex = .*flume.*
agent.sources.avroSrc.interceptors.include.excludeEvents = false
agent.sources.avroSrc.interceptors.exclude.type = regex_filter
agent.sources.avroSrc.interceptors.exclude.regex = .*DEBUG.*
agent.sources.avroSrc.interceptors.exclude.excludeEvents = true
```

Morphline 拦截器

前面在第 5 章“Morphline Solr Sink”一节讨论了 Morphline Solr Sink，在该节中描述了 morphline 如何用于处理事件和加载事件到 Solr 中。也可以使用相同的 morphline 命令去转换事件。该拦截器获取使用哪些 morphline 文件和使用文件中的哪些 morphline 去处理事件的信息。

如果需要重量级的处理，最好使用 Morphline Solr Sink，因为拦截器中的事件处理时间不应该引起 Source 的超时或者 Avro Sink 写入 Source 时超时。拦截器中不应该使用复杂 morphline，例如 loadSolr。如何使用 morphline 的详细信息，请参考 Kite SDK 文档 [morphlines_ch6]。表 6-5 概述了 morphline 拦截器的配置参数。

表6-5 Morphline拦截器配置

146

参数	默认值	描述
type	-	必须使用 org.apache.flume.sink.solr.morphline.MorphlineInterceptor$Builder 这个 FQCN
morphlineFile	-	包含要使用的 morphline 的文件
morphlineId	-	要使用的 morphline 的 ID。如果文件中存在多个 morphline，需要使用该参数

morphlineFile 参数必须指定文件的全路径，该文件包含转换事件的 morphline。morphlineId 参数指定了文件中用于转换事件的 morphline 的 ID。

下面的代码是 morphline 拦截器的配置示例。该示例中，从文件 */etc/flume/conf/morphline.conf* 去加载 ID 是 usingFlume 的 morphline，并去处理传送过来的事件：

```
agent.sources.avroSrc.interceptors = morphlineInterceptor
agent.sources.avroSrc.interceptors.morphlineInterceptor.type = \
org.apache.flume.sink.solr.morphline.MorphlineInterceptor$Builder
agent.sources.avroSrc.interceptors.morphlineInterceptor.morphlineFile = \
/etc/flume/conf/morphline.conf
agent.sources.avroSrc.interceptors.morphlineInterceptor.morphlineId =
usingFlume
```

Morphline Solr Sink 的 morphline 配置文件和包含 morphline 的 JAR 包必须使用插件部署框架部署，该插件部署框架在第 8 章“部署自定义代码”一节里有描述。

UUID 拦截器

类似 Solr 这样的系统，需要每个文档写入时都带有唯一 ID。UUID（通用唯一标示符）拦截器可以用于为每个事件生成这样的唯一标识符。生成的 UUID 可以设置为可配置参数的值。它也可以随意地以预配置前缀字符串为前缀。表 6-6 罗列了 UUID 拦截器的配置参数。

表6-6 UUID拦截器配置

参数	默认值	描述
type	-	org.apache.flume.sink.solr.morphline.UUIDInterceptor$Builder 是必须使用的 Builder 类的 FQCN
headerName	id	需要插入 UUID 作为值的事件报头的键
prefix	-	添加到 UUID 的前缀
preserveExisting	true	如果设置为 false，前缀 +UUID 组合产生的字符串将重写 headerName 限定的事件报头的值

147

下面是 UUID 拦截器的示例，该示例添加 UUID（以 usingFlume- 作为前缀）作为 eventId 事件报头的值，并替换已经存在的 eventId：

```
agent.sources.avroSrc.interceptors = uuidInterceptor
agent.sources.avroSrc.interceptors.uuidInterceptor.type = \
org.apache.flume.sink.solr.morphline.UUIDInterceptor$Builder
agent.sources.avroSrc.interceptors.uuidInterceptor.headerName = eventId
agent.sources.avroSrc.interceptors.uuidInterceptor.prefix = usingFlume-
agent.sources.avroSrc.interceptors.uuidInterceptor.preserveExising = false
```

UUID 拦截器生成的 UUID

UUID 拦截器生成版本 4 的 UUID，实际上是伪随机方式。对于那些需要很强的 UUID 唯一性保证的应用，最好编写一个自定义拦截器来保证强唯一性。

编写拦截器 *

拦截器是 Flume 中最容易编写的组件。编写拦截器的时候，实现者只需要写一个实现 `Interceptor` 接口的类。该接口本身非常简单，虽然 Flume 要求所有的拦截器必须有一个实现了 `Interceptor$Builder` 接口的 `Builder` 类。所有的 Builder 类必须有一个公共无参的构造方法，Flume 使用该方法来进行实例化。可以使用传递到 `Builder` 类的 Context 实例配置拦截器。所有需要的参数都要传递到 Context 实例。

拦截器一般用于分析事件以及在需要的时候丢弃事件。通常情况是，拦截器给事件插入事件报头，这些事件后续用于 HDFS Sink（用于时间戳或者用于基于报头的分桶）、HBase Sink（用于行键）等。这些事件报头也经常在复杂 Channel 处理器中用于将流分为多个流的分支，或者基于优先级将事件发送到不同的 Sink 中，这些事件报头是拦截器分析的内容。这种进行正则匹配和检测优先级（基于类似日志级别这样优先级）的字符串处理的方式，可以从创建数据的应用中卸载下来。

例 6-1 展示了所有拦截器必须实现的 Interceptor 接口。

148

例6-1 Interceptor接口

```
package org.apache.flume.interceptor;
public interface Interceptor {
  public void initialize();
  public Event intercept(Event event);
  public List<Event> intercept(List<Event> events);
  public void close();
  /** Builder implementations MUST have a no-arg constructor */
  public interface Builder extends Configurable {
    public Interceptor build();
  }
}
```

当实现拦截器的时候，有两种处理事件的方法，都命名为 `intercept`，它们具有不同的参数和返回类型。第一种方法只接受一个事件并返回一个事件（或者 `null`），第二种方法可接受一个事件列表并返回一个事件列表。这两种情况下，都包含伴随 Channel 的一个事务。这两种方法必须是线程安全的，因为如果 Source 运行在多线程的情况下，这些方法可能被多个线程调用。

如果处理一个事件的方法被调用，那么相关的事务将只有一个事件，然后 Channel 处理器的 processEvent 方法将调用该事务，processEvent 方法是被 Source 调用来处理事件的。当第二个方法被调用，Channel 处理器的 processEventBatch 方法被 Source 调用，拦截器返回列表中的所有事件都写入一个单个事务中。请参见第 3 章“编写自定义 Source”一节理解 processEvent 方法和 rocessEventBatch 方法的不同之处。

例 6-2 展示了一个简单拦截器，并说明拦截器是如何工作的。Channel 处理器实例化 Builder 类，然后调用 Builder 对象的 configure 方法，该方法用于传递包含用于配置拦截器的配置参数的 Context 实例。然后，Channel 处理器调用 build 方法，该方法返回拦截器。Channel 处理器通过调用拦截器实例的 initialize 方法来初始化拦截器。通过构造方法传递配置参数给拦截器是一种非常好的方法，所以拦截器可以基于最终的配置完成所有的状态，正如 CounterInterceptor 类中实现的一样。

例6-2 简单拦截器

```
package usingflume.ch06;

public class CounterInterceptor implements Interceptor {
  private final String headerKey;
  private static final String CONF_HEADER_KEY = "header";
  private static final String DEFAULT_HEADER = "count";
  private final AtomicLong currentCount;

  private CounterInterceptor(Context ctx) {
    headerKey = ctx.getString(CONF_HEADER_KEY, DEFAULT_HEADER);
    currentCount = new AtomicLong(0);
  }

  @Override
  public void initialize() {
    // No op
  }

  @Override
  public Event intercept(final Event event) {
    long count = currentCount.incrementAndGet();
    event.getHeaders().put(headerKey, String.valueOf(count));
    return event;
  }

  @Override
  public List<Event> intercept(final List<Event> events) {
    for (Event e : events) {
      intercept(
```

```
        e); // Ignore the return value; the event is modified in place
    }
    return events;
  }

  @Override
  public void close() {
    // No op
  }

  public static class CounterInterceptorBuilder
    implements Interceptor.Builder {

    private Context ctx;

    @Override
    public Interceptor build() {
      return new CounterInterceptor(ctx);
    }

    @Override
    public void configure(Context context) {
      this.ctx = context;
    }
  }
}
```

CounterInterceptor 类的拦截方法是线程安全的，因为实例访问的唯一变量要么是 final 类型（基础配置初始化的所有变量），要么是使用线程安全的类（作为计数器使用的 AtomicLong 实例）。处理事件列表的 intercept 方法循环调用处理单个事件的 intercept 方法。建议所有的定制拦截器都遵循这种模式。在这种情况下，由于事件恰好是在变换过程中，所以不创建新列表，仅仅返回包含修改后事件的原始列表。另外，也可以创建一个新的列表，如果需要，则添加新的事件到这个列表。事件可以丢弃，可以通过从返回的原始列表中删除，或者通过不添加事件到返回的新列表中实现。

150

单个拦截器可以返回多少事件？

拦截器不允许返回比原先传递给它的更多的事件，但它可以返回较少的事件。这背后的逻辑是，拦截器添加更多的事件可能会导致写入 Channel 的事件超过吞吐量，即使 Avro Sink 发送数据到 Avro Source，每个批次发送的事件也比吞吐量小。如果拦截器丢弃传递给它的所有事件，拦截器仍必须返回一个列表，如果所有的事件都将被丢弃，该列表可能是空的。

定制拦截器可以部署在 *plugins.d* 目录，如在第 8 章“部署自定义代码”一节的说明。

Channel 选择器

Channel 选择器是决定 Source 接收的一个特定事件写入哪些 Channel 的组件。它们告知 Channel 处理器，然后由其将事件写入到每个 Channel。

由于 Flume 并不是两阶段提交，事件被写入到一个 Channel，然后在事件被写入到下一个 Channel 之前提交。如果写入一个 Channel 时出现故障，可能已经发生在其他 Channel 的相同事件的写入不能被回滚。当这样的故障发生时，Channel 处理器抛出 `ChannelException` 并且事务失败。如果 Source 试图再次写入相同的事件（在大多数情况下，它会；只有类似 Syslog、Exec 等 Source 不能重试，因为没有办法再次生成相同的数据），重复的事件将写入到 Channel，而先前的提交实际上是成功的。这是在 Flume 管道中发生重复的一种方式。

Channel 选择器配置是通过 Channel 处理器完成的，虽然配置看起来像 Source 子组件的配置。传递到 Channel 选择器的所有参数作为 Source 的上下文中的参数使用 `selector` 后缀传递。对于每个 Source，选择器通过使用一个配置参数 `type` 指定。Channel 选择器可以指定一组 Channel 是必需的（`requied`），另一组是可选的（`optional`），如第 2 章的描述。所有 Channel 选择器的一个公共参数在表 6-7 中有描述。

151

表6-7 Channel选择器配置

配置参数	默认值	描述
type	replicating	用于 Source 的 Channel 选择器的别名或 FQCN

Flume 内置两种 Channel 选择器：`replicating` 和 `multiplexing`。如果 Source 的配置中没有指定选择器，那么会自动使用复制 Channel 选择器。一个 Channel 选择器配置例子如下所示：

```
agent.sources.avroSrc.selector.type = multiplexing
agent.sources.avroSrc.selector.header = priority
agent.sources.avroSrc.selector.mapping.1 = channel1
agent.sources.avroSrc.selector.mapping.2 = channel2
agent.sources.avroSrc.selector.default = channel2
```

复制 Channel 选择器

如果 Source 没有指定 Channel 选择器，则该 Source 使用复制 Channel 选择器。该复制 Channel 选择器复制每个事件到通过 Source 的 `channels` 参数所指定的所有的 Channel 中。

除了表 6-7 指定的参数外，复制 Channel 选择器还有一个配置参数 optional，该参数是空格分隔的 Channel 名字列表。该参数是可选的。此参数指定的所有 Channel 都认为是可选的，所以如果当事件写入这些 Channel 时有失败发生，则仅仅忽略这些失败。写入任何其他 Channel 的失败将导致对 Source 抛出异常，指明失败并要求 Source 重试。

下面的 Source 配置说明了复制 Channel 选择器的使用，该配置没有可选 Channel ：

```
agent.sources.avroSrc.type = avro
agent.sources.avroSrc.channels = c1 c2 c3
```

这种配置使得由 Avro Source 接收的每一个事件都写入三个 Channel，c1、c2 和 c3。如果它们中的任何一个是满的，或者因任何其他原因不能写入，则 Avro Source 得到 ChannelException，这会导致 Source 通知失败发生的前一阶段（发送消息的 Avro Sink 或 RPC 客户端），使其重试（虽然使用 RPC 客户端的应用程序的行为依赖于应用程序，但在 Avro Sink 的情况下能保证重试）。

如果该配置改为 C3 是可选的，如下所示，未能写入 C3 不会导致 ChannelException 抛出到 Source，并且 Source 将通知前一阶段写入是成功的 ： 152

```
agent.sources.avroSrc.type = avro
agent.sources.avroSrc.channels = c1 c2 c3
agent.sources.avroSrc.selector.optional = c3
```

可选 Channel 也必须罗列在 Source 的 channels 参数中，但必须使用传递到选择器的 optional 参数标记为可选。请注意，即使我们没有在这里指定选择器的类型（因为它有默认值），仍然需要传递配置参数。复制 Channel 选择器不做任何其他处理或流的分叉；它只是复制数据。通过去往不同目的地的 Sink 从每个 Channel 读取事件，可以使得事件写入到多个目的地。

多路复用 Channel 选择器

多路复用 Channel 选择器是一种专门用于动态路由事件的 Channel 选择器，通过选择事件应该写入的 Channel，基于一个特定的事件头的值进行路由。结合拦截器，可以在事件上做一些分析，然后决定应该写入哪一个 Channel。

多路复用 Channel 选择器寻找一个特定的报头，该报头通过选择器的配置指定。基于该报头的值，选择器返回事件写入的 Channel 的一个子集。被写入的 Channel 列表由配置中的每个报头的值指定。如果配置未指定一个特定事件中报头的值，则该事件写入到默认 Channel。

图 6-1 展示出基于报头的值到一个或者多个 Channel 的事件流。在这种情况下，选择器检查优先级报头的值。无论是优先级为 1 或 2 事件都被写入 HDFS，而优先级为 1 的事件也被写入到 HBase。这种路由甚至可以在接收数据层完成，为了更低的延时，通过更快的、不可靠的流（使用 Memory Channel）发送更高优先级的事件。同时如果必要，通过慢但可靠的流（使用 File Channel）来发送所有的事件和后续的去重（包括高优先级的）。

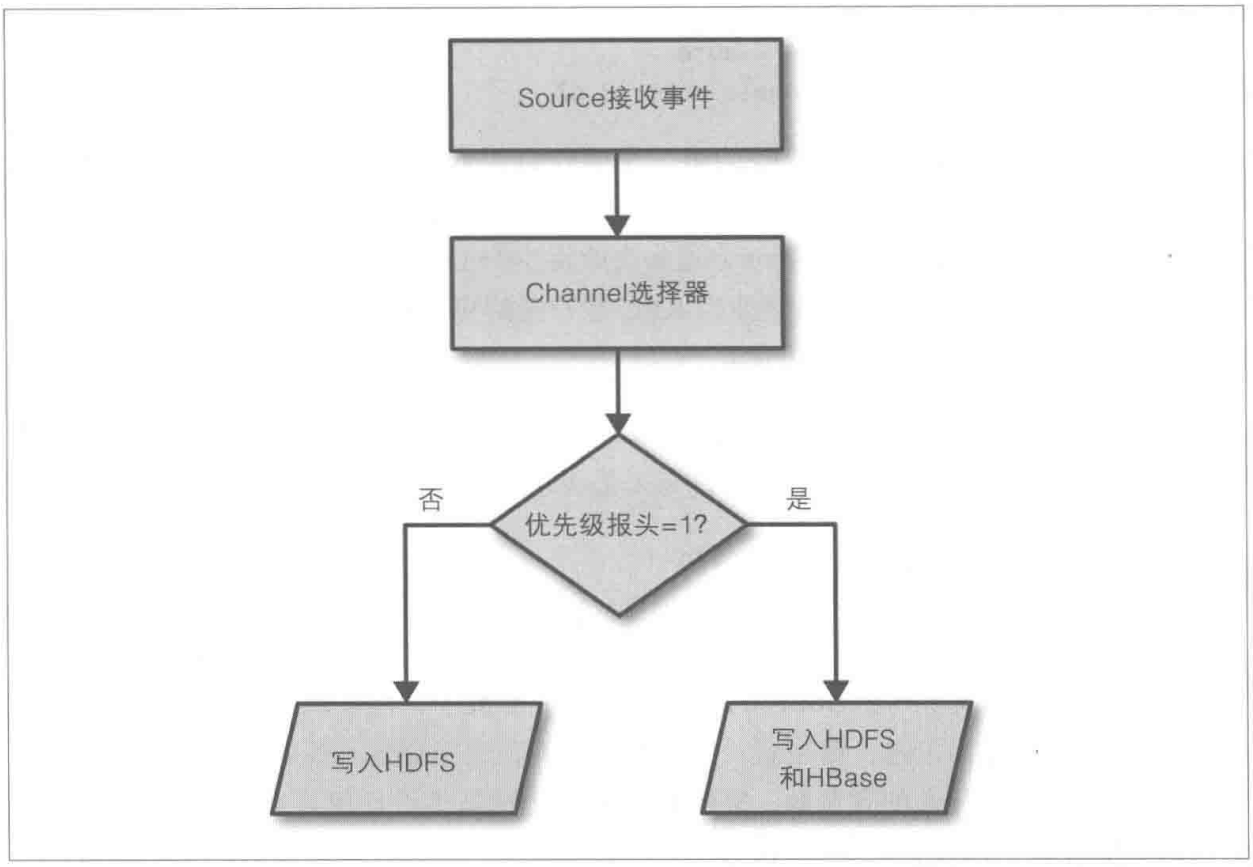


图6-1 多路复用Channel选择器

表 6-8 显示了多路复用 Channel 选择器的配置参数。请注意，这里显示的所有配置参数，在 Source 的上下文中必须使用 `selector.` 前缀。

表6-8 多路复用Channel选择器配置

配置参数	默认值	描述
type	-	multiplexing
header	flume.selector.header	值必须被检查且用于路由事件的报头
mapping.<hdr-value>	-	报头映射列表。每个映射是事件写入的 Channel 列表，如果报头的值匹配该参数的值（<hdr-value>）

续表

配置参数	默认值	描述
optional.<hdr-value>	-	与 mapping 参数相同，但是在这个列表中指定的 Channel 是可选的，写入失败会忽略
default	-	事件必须写入的 Channel 列表，如果报头不存在或者报头的值没有指定映射列表

配置多路复用 Channel 选择器与配置复制选择器颇有几分不同。与往常一样，Source 写入的所有 Channel 必须在 Source 的 channels 参数中指定。为了使 Source 的多路复用 Channel 选择器生效，该 Source 的 selector.type 参数必须设置为 multiplexing。传递到 Channel 选择器的所有参数使用 Source 前缀，该 Source 后面跟着 selector.，如下所示：

154

```
agent.sources.avroSrc.type = avro
agent.sources.avroSrc.channels = c1 c2 c3
agent.sources.avroSrc.selector.type = multiplexing
agent.sources.avroSrc.selector.default = c3
```

对于每个事件，选择器查找配置中 header 参数指定的键的报头。接下来，它检查报头的值是否是配置中使用 mapping. 前缀指定值的其中一个。如果是 *mapping* 的一个匹配，那么它通过映射将事件写出到 Channel。可选的映射也可以使用 optional. 前缀而不是用 mapping. 前缀进行指定。任何指定为可选的 Channel 列表写失败，将会被忽略。如果选择器没有找到匹配或报头本身不存在，那么它写事件到 default 参数指定的 Channel 中。如果一个事件没有映射到任何所需的 Channel，但是映射到一个或多个可选的 Channel，那么该事件被写出到可选 Channel 和默认 Channel，任何写出到默认 Channel 的失败将会导致抛出 ChannelException 异常。

下面展示了一个配置了多路复用 Channel 选择器的 Source 的实例：

```
agent.sources.avroSrc.type = avro
agent.sources.avroSrc.channels = c1 c2 c3 c4 c5
agent.sources.avroSrc.selector.type = multiplexing
agent.sources.avroSrc.selector.header = priority
agent.sources.avroSrc.selector.mapping.1 = c1 c2
agent.sources.avroSrc.selector.mapping.2 = c2
agent.sources.avroSrc.selector.optional.1 = c3
agent.sources.avroSrc.selector.optional.2 = c4
agent.sources.avroSrc.selector.optional.3 = c4
agent.sources.avroSrc.selector.default = c5
```

在这个例子中，Avro Source 写事件到四个 Channel。但是不像复制 Channel 选择器，不是所有的事件都被写入到所有的四个 Channel。对于每个事件，Channel 选择器查找使用

priority 键的报头。

对于每个优先级为 1 的事件写入到三个 Channel，c1、c2 和 c3，c3 被标记为是可选的。所以，如果写入到 c1 或 c2 失败，Source 会从 Channel 处理器的 processEvent 或 processEventBatch 方法得到一个异常，且该 Source 必须重试。但是因为 c3 标记为可选的，如果写入到 c3 失败，Source 不会得到异常且意识不到失败，因为这个失败被 Channel 处理器所忽略。

类似地，任何优先级为 2 的事件写入到 c2 和可选的 c4。可以清楚地看出，Channel 可以出现在多个映射中，例如这个例子中的 c2。优先级为 1 或 2 的 Channel 写入到 c2——这就是图 6-1 展示的例子是如何实现的。

缺少优先级报头的事件，或值不是 1 或 2 的事件，写入到默认的 Channel，在这个例子中是 c5。如果一个事件没有找到所需的 Channel，那么事件将写入到该事件可选的 Channel 和默认的 Channel 中。在这个例子中，优先级为 3 的事件将写入到 Channel c4 和 c5。如果写入到 c4 失败，那么该失败被忽略，但是如果写入到 c5 失败，Source 会得到一个异常，且该事件必须被重新写入。

自定义 Channel 选择器 *

可以编写和部署自定义 Channel 选择器，允许你使用指定部署逻辑来控制事件流。为了实现一个自定义 Channel 选择器，该选择器需要实现 ChannelSelector 接口或继承 AbstractChannelSelector 类。例 6-3 展示了 AbstractChannelSelector 类。

对于每个事件，Channel 处理器调用 Channel 选择器的 getRequiredChannels 和 getOptionalChannels 方法，返回需要的和可选的将要写入事件的 Channel 列表。如果写入到任何所需的 Channel 的操作失败，Channel 处理器将会抛出 ChannelException 异常，并使 Source 重试。任何写入到可选的 Channel 的操作失败都会被忽略。

例6-3 自定义选择器继承的AbstractChannelSelector类

```
package org.apache.flume.channel;

public abstract class AbstractChannelSelector implements ChannelSelector {

  private List<Channel> channels;
  private String name;

  @Override
  public List<Channel> getAllChannels() {
    return channels;
```

```
  }

  @Override
  public void setChannels(List<Channel> channels) {
    this.channels = channels;
  }

  @Override
  public synchronized void setName(String name) {
    this.name = name;
  }

  @Override
  public synchronized String getName() {
    return name;
  }

  protected Map<String, Channel> getChannelNameMap() {
    Map<String, Channel> channelNameMap = new HashMap<String, Channel>();
    for (Channel ch : getAllChannels()) {
      channelNameMap.put(ch.getName(), ch);
    }
    return channelNameMap;
  }

  protected List<Channel> getChannelListFromNames(String channels,
          Map<String, Channel> channelNameMap) {
    List<Channel> configuredChannels = new ArrayList<Channel>();
    if(channels == null || channels.isEmpty()) {
      return configuredChannels;
    }
    String[] chNames = channels.split(" ");
    for (String name : chNames) {
      Channel ch = channelNameMap.get(name);
      if (ch != null) {
        configuredChannels.add(ch);
      } else {
        throw new FlumeException("Selector channel not found: "
                + name);
      }
    }
    return configuredChannels;
  }

}
```

156

Channel 处理器调用 setChannels 方法，该方法传递给所有的 Channel，即选择器必须为每个事件选择的 Channel。该类实现了 Configurable 接口，所以当选择器初始化时调用 configure 方法。当每个事件将要处理时，调用 getRequiredChannels 和 getOptionalChannels 方法。getAllChannels 方法必须返回在创建期间 Channel 处理器设置的所有 Channel。

这个类也提供了一对便捷的方法：一个方法返回 Channel 名字的一个 map，给实际的 Channel 实例；另一个方法返回 Channel 实例的列表，该列表是以空格分隔的字符串表示的 Channel 名字。可以使用 FQCN 部署 Channel 选择器：

157

```
agent.sources.avroSrc.type = avro
agent.sources.avroSrc.channels = c1 c2 c3 c4 c5
agent.sources.avroSrc.selector.type = com.usingflume.selector.RandomSelector
agent.sources.avroSrc.selector.default = c5
agent.sources.avroSrc.selector.random.seed = 4532
```

在这种情况下，自定义选择器获取通过 agent.sources.avro.selector. 传递的所有参数，就像其他组件一样。在这个例子中，选择器将会在键为 default 和 random.seed，值分别为 c5 和 4532 的 configure 方法中，得到一个 Context 实例。

自定义 Channel 选择器应该放入 plugins.d 目录，就像第 8 章“部署自定义代码”一节描述的那样。

Sink 组和 Sink 处理器

第 5 章我们已经讨论了 Sink 是如何工作的，以及 Flume 自带的不同种类的 Sink 处理器。我们也简要讨论过 Sink 组和 Sink 处理器。正如我们前面讨论的，Flume 配置框架为每个 Sink 组实例化一个 Sink 运行器，来运行 Sink 组。每个 Sink 组可以包含任意数量的 Sink。Sink 运行器持续请求 Sink 组，要求其中的一个 Sink 从自己的 Channel 中读取事件。Sink 组通常用于 RPC Sink，在层之间以负载均衡或故障转移方式发送数据。

因为 RPC Sink 被设计成只连接一个 RPC Source，从一个 Flume Agent 发送数据到下一层的一组 Agent，至少需要和发送事件到 Agent 一样多的 Sink。为了确保每个 Agent 发送事件到下一层的多个目的地 Agent，且一层上的每层发送数据到下一层的所有 Agent，不需要压倒性地占据网络或那些 Agent，那么每个 Agent 可以在下一层所有机器之间负载均衡。

很重要的是要理解 Sink 组中所有 Sink 是不会在同时被激活的；任何时候只有它们中的一个用来发送数据。因此，Sink 组不应该用来更快地清除 Channel，在这种情况下，多个 Sink 应该只是设置为自己操作自己，而没有 Sink 组，且它们应该配置从相同的 Channel 进行读取。

每个 Sink 组在活跃列表中被声明为一个组件，就像 Source、Sink 和 Channel 一样，使用 `sinkgroups` 关键字。每个 Sink 组是一个需要命名的组件，因为每个 Agent 可以有多个 Sink 组。Sink 组以下面的方式进行定义：

```
agent.sinkgroups = sg1 sg2
```

该配置展示了两个定义的 Sink 组：`sg1` 和 `sg2`。然后每个 Sink 组配置了一些 Sink 作为组的一部分。在 Sink 活跃集合中的 Sink 列表，优先于指定为 Sink 组一部分的 Sink 列表。因此，所有作为 Sink 组的一部分的 Sink，也必须分别定义在 Sink 的活跃集中，以用来激活它们。下面展示用 Sink 集合配置 `sg1` 和 `sg2`:

158

```
agent.sinks = s1 s2 s3 s4
agent.sinkgroups.sg1.sinks = s1 s2
agent.sinkgroups.sg2.sinks = s3 s4
```

Sink 组中的每个 Sink 必须单独进行配置。这包括以下这些配置，Sink 从哪个 Channel 读取，它写数据到哪些主机或集群等。如果 Sink 组表示一组 RPC Sink，意味着传达到下一层，每个连接的主机必须配置一个 Sink，用来发送数据给它。据推测，它们都从同一 Channel 读取，因为这是层和层之间的通信。在理想情况下，如果 Sink 组中建立了几个 Sink，所有的 Sink 将从相同的 Channel 读取，这将有助于在当前层以合理的速度清除数据，确保将要被发送到多台机器的数据，以一种支持负载均衡和故障转移的方式进行发送。

清除 Channel 的速度比单个 Sink 组这样做的速度要快很重要，但也要求当建立的 Agent 发送数据到多个主机时，可以添加多个 Sink 组，每个组中的 Sink 有相似的配置。例如，前一个示例中的 `sg1` 和 `sg2` 分别有 Sink `s1`、`s2` 和 `s3`、`s4`。`s1` 和 `s3` 可以有相同的配置（从同一 Channel 将数据推送到相同的主机和端口），同时 `s2` 和 `s4` 可以有相似的配置。这将确保更多的对于目的地 Agent 的连接是开放每一个 Agent 的，同时如果需要，也允许数据被推到不止一个的 Agent。这使得 Channel 被清除得更快，同时确保负载均衡和故障转移自动发生。

目前为止，我们已经讨论了 Sink 组如何创建流去做负载均衡和故障转移，但是我们还没有讨论如何实际地告知 Sink 组它们应该进行负载均衡或故障转移。这是通过使用 Sink 处理器完成的。Sink 处理器是任何时候决定哪个 Sink 是活跃的组件。

注意，Sink 处理器与 Sink 运行器不同。Sink 运行器实际上是运行 Sink 的，而 Sink 处理器决定了哪个 Sink 应该从自己的 Channel 中拉取事件。当 Sink 运行器要求 Sink 组告知其中一个用于从 Channel 中拉取事件的 Sink，且将事件写到下一阶段（或存储系统）时，Sink 处理器就是实际用来选择完成这个过程的 Sink 组件。Flume 自带了两类 Sink 处理器：*load-balancing sink* 处理器和 *failover sink* 处理器。

159 对于指定的 Sink 组的一部分，Sink 处理器使用 `processor.type` 后缀进行配置。配置可以使用 `processor.` 前缀传递给 Sink 处理器。这里是如何配置的一个例子。

```
agent.sinks = s1 s2
agent.sinkgroups = sg1
agent.sinkgroups.sg1.sinks = s1 s2
agent.sinkgroups.sg1.processor.type = load_balance
agent.sinkgroups.sg1.processor.backoff = false
```

Load-Balancing Sink 处理器

假设你有一个拓扑结构，在推送数据到 HDFS 之前，其中第一层从成千上万的应用程序服务器接收数据，第二层通过 Avro RPC 从第一层接收数据。为简单起见，我们假设第一层有 100 个 Agent，第二个层有 4 个 Agent。在尽可能简单的拓扑中，第一层每个 Agent 将有四个 Avro Sink 用来推送数据到第二层的每个 Agent。该工作正常运行，直到其中第二层的一个 Agent 失败。此时，配置发送数据的 Sink 将不会发送任何数据，直到第二层失败的 Agent 重新上线。

除了该 Sink 耗尽了 Agent 上的几个线程这样一个事实（一个用于 Sink Runner，另一个用于使用 Netty 发送数据的线程池）浪费了 CPU 周期，直到第二层 Agent 启动并运行，通过创建删除事件的事务且回滚，该 Sink 也会给 Channel 造成额外的压力。对于 File Channel，尽管事务没有被提交，许多将要写入文件的读取操作（即使事务没有提交，这些读取操作也要写入文件），也造成了 I/O 成本和磁盘空间成本。如图 6-2 所示。

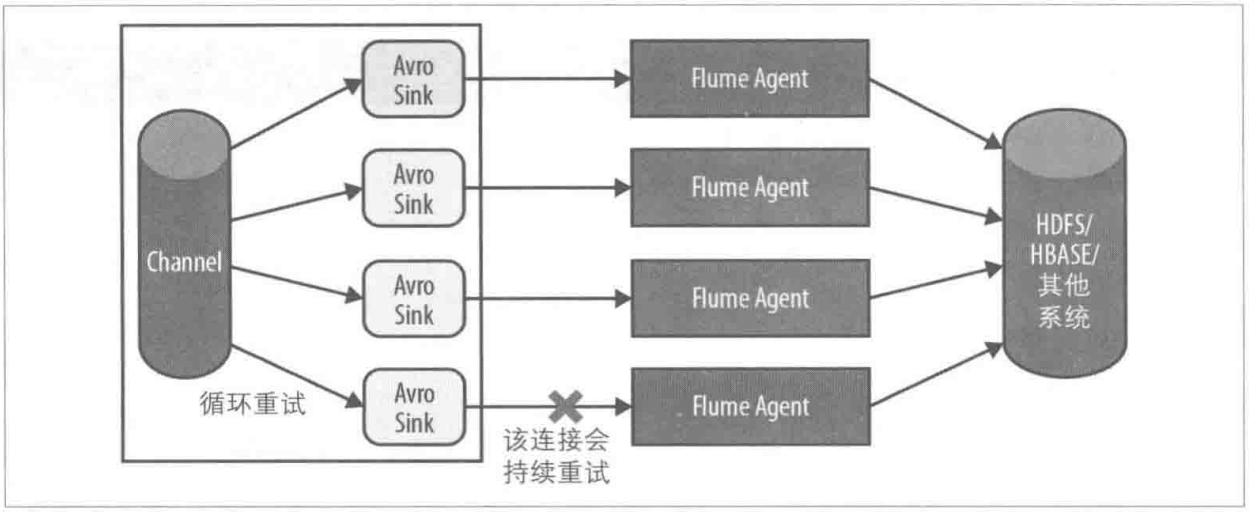


图6-2 为什么我们需要load-balancing sink处理器

正如你所看到的，有这样一个拓扑结构不仅能够充分利用网络，也不会浪费 CPU 周期和较高的 I/O 开销。为了避免这样的问题，Sink 组使用 load-balancing sink 处理器是一个好主意，它将从 Sink 组所有的 Sink 中选择一个 Sink，处理来自 Channel 的事件。

Sink 选择的顺序可以配置为 random 或 round-robin。如果顺序被设置为 random，那么将随机从 Sink 组的 Sink 中选择一个，用来从自己的 Channel 中移除事件并将它们写出。round-robin 选项使 Sink 以循环的方式被选择：每个过程循环调用定义 Sink 组中指定顺序 Sink 的 process 方法。如果 Sink 写入到一个失败的 Agent 或速度太慢的 Agent，会导致超时，Sink 处理器会选择另一个 Sink 写数据。

Sink 处理器可以配置将失败的 Sink 加入黑名单，回退时间以指数方式增长直到达到上限值。这能确保相同的 Sink 不会循环重复尝试且不浪费资源，直到回退时间过期。

表 6-9 中展示了 load-balancing sink 处理器的配置参数。所有的参数必须以 Sink 组前缀后面紧跟着的 processor. 为前缀，以确保 Sink 处理器能获取到正确的参数。

表6-9 load-balancing sink处理器配置

配置参数	默认值	描述
type	-	必须设置为 load_balance
selector	round_robin	可以设置为 round_robin 或 random，或实现了 LoadBalancingSinkProcessor$SinkSelector 接口的类的 FQCN
backoff	false	如果设置为 true，失败的 Sink 将加入到黑名单，加入黑名单的时间以指数形式增长的时间周期

续表

配置参数	默认值	描述
selector.maxTimeOut	30000	毫秒值时间，该时间之后黑名单时间周期不再增长

load-balancing sink 处理器用下面的方式进行配置：

```
agent.sinks = s1 s2 s3 s4
agent.sinkgroups = sg1
agent.sinkgroups.sg1.sinks = s1 s2 s3 s4
agent.sinkgroups.sg1.processor.type = load_balance
agent.sinkgroups.sg1.processor.selector = random
agent.sinkgroups.sg1.processor.backoff = true
agent.sinkgroups.sg1.processor.selector.maxTimeOut = 10000
```


该配置设置 Sink 组使用 load-balancing sink 处理器，随机选择 s1、s2、s3 或 s4 中的一个。如果一个 Sink（或者更准确地，Sink 发送数据到的 Agent）失败，那么该 Sink 会被加入到黑名单，回退时间从 250 毫秒开始，然后以指数形式增长直到达到 10 秒。在这之后，每次写操作失败，Sink 就回退 10 秒，直到它能够成功写入数据，此时回退时间被重置为 0。如果 selector 参数值设置为 round_robin，那么 s1 被首先用来处理数据，然后是 s2，然后是 s3，接下来是 s4，然后再次是 s1。

该配置意味着，在任何时候每个 Agent 只有一个 Sink 写数据。可以通过添加多个有相似配置的 load-balancing sink 处理器的 Sink 组进行修改。注意，可能会有多个 Agent 尝试写入数据到第二层每个 Agent。

拥有太多 Sink 发送数据到相同 Agent 的风险

因为每个 Avro Sink 对 Avro Source 保持持续开放的连接，拥有写入到相同 Agent 的多个 Sink 会增加更多的 socket 连接，且在第二层 Agent 上占据更多的资源。对相同 Agent 增加大量 Sink 之前必须要谨慎考虑。

编写 Sink 选择器 *

每次 Sink 运行器调用 process 方法时，可以使用自定义逻辑的 load-balancing sink 处理器，用来选择激活使用哪个 Sink。自定义选择器必须实现这里展示的 LoadBalancingSinkProcessor$SinkSelector 接口：

```
public interface SinkSelector extends Configurable, LifecycleAware {
  void setSinks(List<Sink> sinks);
  Iterator<Sink> createSinkIterator();
```

```
    void informSinkFailed(Sink failedSink);
  }
```

当 Sink 处理器启动时，用传递过来的 Sink 的列表，初始化 Sink 选择器且调用 setSinks 方法。在配置文件中该列表以相同的顺序指定。每次 Sink 处理事件时，就调用 createSinkIterator 方法。该方法必须返回一个迭代器，这个迭代器以 Sink 必须要求拉取数据入 Sink 的顺序，返回 Sink。

一旦 Sink 成功处理事件且返回成功，那么现在的迭代器就被废弃，再次调用该方法用来获取新的迭代器，这可能会返回不同顺序的 Sink。当 Sink 发送事件失败时（由抛出的异常所指示），那么调用 informSinkFailed 方法。如果需要，这可以用来将该 Sink 暂时加入黑名单。 162

为了创建一个自定义 Sink 选择器，必须在你的 *pom.xml* 文件的依赖部分包含 flume-ng-core 工件，像例 3-6 展示的那样。

Failover Sink 处理器

图 6-2 中展示的相同问题可以用一种略微不同的方式来解决。load-balancing sink 处理器的问题是，因为每个 Sink 组决定哪个 Sink 在大量的 Agent 上是活跃的，所以可能第二层的 Agent 将不能接收到相同总量的数据，尽管当使用循环顺序时，它们应该是平均的。然而，像前面描述的一样，可以配置 Sink 组使用硬连接方式的写操作，直到失败实际发生。大多数时候，通过允许 Sink 组写入一致的数据到相同的 Sink，可以预测多少数据被写入到每个 Agent。这种情况可以使用 *failover sink* 处理器来实现。

failover sink 处理器从 Sink 组中以优先级的顺序选择 Sink。拥有最高优先级的 Sink 先写数据直到它失败（在 RPC Sink 的情况下，Sink 的失败甚至可以是下游 Agent 的死亡），然后选择组中其他 Sink 中拥有最高优先级的 Sink。只有当前 Sink 写入数据失败时，才会选择另一个不同的 Sink 写数据。这能确保当没有失败时，每台机器上只有一个 Sink 写入到第二层的所有 Agent，只有当失败时某些机器才会看到更多传入的数据。

但是除非现在的 Sink 失败，failover 机制不会选择一个新的 Sink。这意味着，即使已经失败的最高优先级的 Agent 重新上线，failover sink 处理器也不会让写入该 Agent 的 Sink 激活，直到目前活跃的 Sink 遇到一个错误。图 6-3 展示了 failover sink 处理器的工作流。

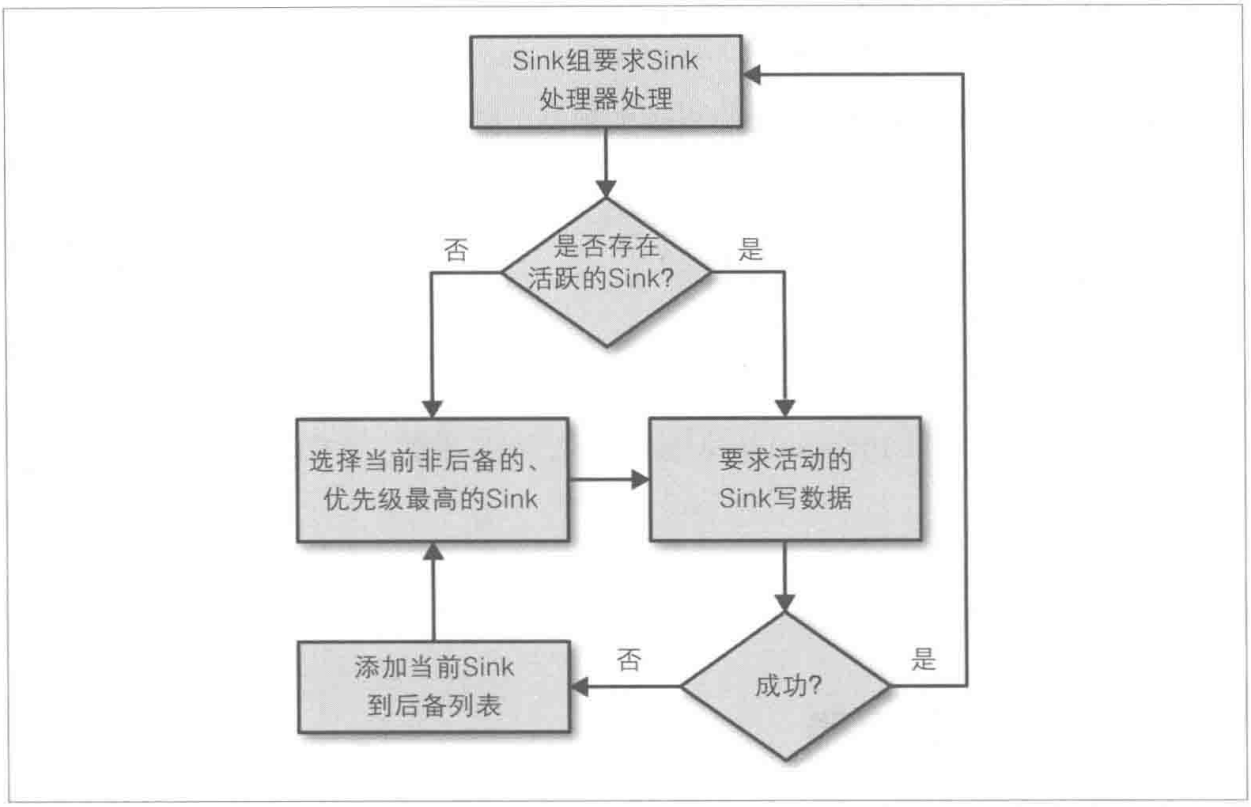


图6-3 failover sink处理器工作流

表 6-10 展示了可以用来配置 failover sink 处理器的配置参数列表。所有的参数必须以 Sink 组的 processor. 为前缀，以确保参数传递给 Sink 处理器。

表6-10 failover sink处理器配置

配置参数	默认值	描述
type	-	必须设置为 failover
priority.<sink_name>	-	Sink 组中每个 Sink 的优先级列表
maxpenalty	30000	失败 Sink 的最大回退时间

就如表 6-10 中展示的一样，failover sink 处理器的 type 参数值是 failover。因为每个 Sink 处理器以优先级的顺序激活 Sink，那么 Sink 的优先级必须在配置中使用 priority. 前缀参数进行设置，priority. 后面紧跟着的是 Sink 的名字，参数值设置为期望的优先级。注意，优先级被认为是以增加的顺序递增的顺序，这意味着优先级的绝对值越高，Sink 越早被激活。

例如，一个优先级为 100 的 Sink 在优先级为 90 的 Sink 之前被激活。如果没有为指定的 Sink 设置优先级，那么 Sink 的优先级取决于 Sink 在 Sink 组配置中被指定的顺序。每次 Sink 写数据失败时，Sink 被认为失败且在短暂的一段时间内被加入黑名单。这个黑名单中的时间间隔（与 load-balancing sink 处理器的回退时间类似）在每个连续导致失败的 164 尝试时增加，直到达到 maxpenalty 指定的值（以毫秒为单位）。一旦黑名单时间间隔达到该值，在尝试了这么多毫秒之后，将导致 Sink 进一步的失败。一旦 Sink 在这么多时间之后成功写入数据，那么回退时间被重置为 0。看一下下面的配置：

```
agent.sinks = s1 s2 s3 s4
agent.sinkgroups.sg1.sinks = s1 s2 s3 s4
agent.sinkgroups.sg1.processor.type = failover
agent.sinkgroups.sg1.processor.priority.s2 = 100
agent.sinkgroups.sg1.processor.priority.s1 = 90
agent.sinkgroups.sg1.processor.priority.s4 = 110
agent.sinkgroups.sg1.processor.maxpenalty = 10000
```

在该配置中，四个 Sink 使用了 failover 的配置，Sink s4 有最高的优先级，接下来是 s2 和 s1。Sink s3 没有设置优先级。对于没有指定优先级的 Sink，第一个没有优先级的 Sink 被分配优先级 0，下一个分配优先级 -1，下一个分配 -2，以此类推。这些优先级只分配给没有设置优先级的 Sink。因此，这里展示的配置样本间接分配了 s3 优先级 0，所以 Sink 以 s4、s2、s1、s3 的顺序进行重试。值得注意的是，如果两个 Sink 有相同的优先级（间接或明确分配的），只激活 Sink 组中首先指定的 Sink。也要注意，如果明确和间接的优先级被设置为相同的范围，那么它们的值也和上面一样使用。例如：

```
agent.sinks = s1 s2 s3 s4
agent.sinkgroups.sg1.sinks = s1 s2 s3 s4 s5 s6
agent.sinkgroups.sg1.processor.type = failover
agent.sinkgroups.sg1.processor.priority.s2 = 0
agent.sinkgroups.sg1.processor.priority.s4 = 110
agent.sinkgroups.sg1.processor.priority.s5 = -5
agent.sinkgroups.sg1.processor.priority.s6 = -2
agent.sinkgroups.sg1.processor.maxpenalty = 10000
```

在该配置中，Sink s4 有最高的优先级，所以 s4 最先被激活。Sink s1 将被分配优先级 0——和 s2 相同——这意味着 s2 不会被激活。s3 的优先级是 -1，所以激活的顺序是 s4、s1、s3、s6、s5。虽然 s3 的优先级在配置中没有指定，但是间接指定了它的优先级比 s5 和 s6 的高，所以 s3 在它们两个之前被激活。

165

总结

在本章中我们讨论了拦截器、Channel 选择器、Sink 组和 Sink 处理器。可以使用第 8 章将讨论的 *plugins.d* 框架对它们进行部署。

参考文献

- [morphlines_ch6] Morphlines: Kite SDK,*http://bit.ly/1AS0g5I*

第 7 章 167

发送数据到Flume*

到目前为止，我们已经讨论了 Flume Agent 的内部结构，以及如何配置组成一个 Agent 的不同种类组件。本章我们将讨论数据从客户端应用程序发送到一个或多个 Flume Agent 的不同方法。Flume 有两类发送数据到 Flume Agent 的程序性方法：Flume SDK 和 Embedded Agent API。Flume 也自带有 log4j appender，可以用来从应用程序发送数据到 Flume Agent。

构建 Flume 事件

在我们讨论用来发送数据到 Flume Agent 的 API 之前，先让我们看一下如何创建 Flume 事件。正如我们在第 2 章讨论过的，事件是 Flume 中数据的基本表现形式。每个 Flume 事件包含 header 的一个 map 集合和一个 body，是表示为字节数组的有效负荷。例 7-1 展示了 Event 接口。

例7-1 Event接口

```
package org.apache.flume;
public interface Event {
  public Map<String, String> getHeaders();
  public void setHeaders(Map<String, String> headers);
  public byte[] getBody();
  public void setBody(byte[] body);
}
```

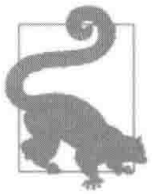
很明显，Event 接口不同实现内的数据内部表示可能不同，只要其显示接口指定格式的 header 和 body 即可。通常大多数应用程序使用 Flume 的 EventBuilder API 创建事件。EventBuilder API 提供了几个静态方法来创建事件。在所有情况下，API 本身对提交的实际事件数据不做修改——不论是 header 还是 body。EventBuilder API 提供了 4 个通 168

常用来创建 Flume 事件的方法。它们如下所示：

```
public class EventBuilder {
  public static Event withBody(byte[] body, Map<String, String> headers);
  public static Event withBody(byte[] body);
  public static Event withBody(String body, Charset charset,
                               Map<String, String> headers);
  public static Event withBody(String body, Charset charset);
}
```

第一个方法只是将 body 看作字节数组，header 看作 map 集合，而第二个方法将 body 看作字节数组但不设置事件 header。第三个和第四个方法可以用来创建 Java `String` 实例的事件，使用提供的字符集转换为编码的字节数组，然后用作 Flume 事件的 body。第三个方法也将事件 header 作为一个参数。

现在我们知道了如何创建 Flume 事件，那么我们可以使用 Flume SDK 或 Embedded Agent API 发送数据到 Flume Agent。



Flume NG SDK 组件

为了使用任何 RPC 客户端或 `Event` 和 `EventBuilder API`，那么确保在你的应用程序中包含了 Flume Client SDK。组件在 Maven 中心仓库是可用的。你可以通过在 *pom.xml* 文件中添加以下内容到依赖部分来包含 `flume-ng-sdk` 组件：

```
<dependency>
  <groupId>org.apache.flume</groupId>
  <artifactId>flume-ng-sdk</artifactId>
  <version>1.5.0</version>
</dependency>
```

Flume 客户端 SDK

一旦应用程序知道，通过 Flume 什么数据将被发送到 HDFS，应用程序就需要发送数据到 Flume Agent。我们已经在第 3 章“HTTP Source”一节讨论过 HTTP Source 和它可插入的处理器。我们也了解了 HTTP Source 接受的 JSON 格式的事件。这是将数据发送到 Flume 的一种方式——使用 HTTP Source 作为接受的 Source，并且有 HTTP 友好的格式化数据的应用程序（如果使用默认的处理器，即 JSON 化数据）。但是关于这方面的问题是，因为有额外的 HTTP 和编码 / 解码的开销，比它需要的更低效。

169 因为 Flume 事件的格式是固定的，发送数据到 Flume 的最好方式是通过 Flume 支持的

RPC 格式之一的 RPC 调用：Avro 或 Thrift。通常，首选的是 Avro RPC，因为通过在 Flume 实例生产环境使用的检测，它更成熟、更优秀。使用 Avro RPC 的缺点是 Flume 使用的 Avro RPC 的版本只支持 Java 和其他 JVM 语言。Flume 通过 Thrift RPC 支持非 JVM 语言，但是在 Flume 的实例中，Thrift RPC 可能缺少一些 Avro RPC 可用的特性。本章我们将讨论 Flume SDK，以及如何使用 SDK 编写程序以发送数据到 Flume。

创建 Flume RPC 客户端

我们首先来看一下如何用 Java 创建 RPC 客户端。通过 RpcClientFactory 类创建 RPC 客户端实例。该类提供了创建不同种类 RPC 客户端的方法。所有用来创建 RPC 客户端的类接收一个 Properties 实例。Properties 实例包含用来配置 RPC 客户端的配置信息。所有的 RPC 客户端使用下面的方法进行创建：

```
public static RpcClient getInstance(Properties properties);
public static RpcClient getInstance(File properties);
```

Properties 实例基本上是一个包含配置参数的 map 集合。第二个方法需要一个 File 实例用来表示属性文件格式中的配置，这在第 2 章“配置 Flume Agent”一节解释过。Factory 类只需要一个必需的参数——client.type 参数，用来指明创建的 RPC 客户端的类型。该参数必须设置为 default、default_failover、default_loadbalance 或 thrift 其中的一个。RPC 客户端一旦创建，就传递 Properties 实例给它。在我们看不同种类可用的 RPC 客户端之前，让我们先来看看应用程序开发者必须针对其编程的 RPC 客户端接口。

RPC 客户端接口

RPC 客户端接口极其简单和简约。因为 RPC 客户端在创建过程中是通过自身进行配置的，应用程序开发者不需要担心配置 RPC 客户端。例 7-2 展示了 RPC 客户端接口。

例7-2 Flume RPC客户端接口

```
package org.apache.flume.api;
public interface RpcClient {
  public int getBatchSize();
  public void append(Event event) throws EventDeliveryException;
  public void appendBatch(List<Event> events) throws EventDeliveryException;
  public boolean isActive();
  public void close() throws FlumeException;
}
```

170

为了使用 RPC 客户端实例发送数据到 Flume Agent，应用程序必须调用 appendBatch 或

append 方法。这些方法接收必须发送到 Flume Agent 的事件，且在返回之前通过网络发送它们。如果方法返回成功，意味着成功将事件写入到目的地 Agent 的 Channel。如果目的地 Agent 不能写出事件到一个或多个 Source 需要的 Channel，或由于网络问题或其他任何问题导致事件不能成功写出，这些方法会抛出 EventDeliveryException 异常。如果抛出了 EventDeliveryException 异常，由使用 RPC 客户端的应用程序决定如何进行处理。应用程序可以回滚和重试，或甚至基于它们自己的内部逻辑移除事件。

如果列表中传递给 appendBatch 的事件数量比批量大小要大，RPC 客户端将把列表分为多个批量且顺次将它们写出，只有当成功写出所有的事件，RPC 客户端才会返回。只要有一个批量操作失败，该方法就抛出一个异常。如果一些批量操作成功而另一些失败，如果应用程序重试则可能导致最终目的地的事件重复。因此，传递和批量大小一样多或更少的事件通常是一个好主意。getBatchSize 方法返回 RPC 客户端使用的批量的最大的大小。

在每个 RPC 调用之前，通过调用 isActive 方法检查 RPC 客户端是否仍然活跃和是否为执行做好准备是一个好主意。一旦不需要 RPC 客户端或 isActive 返回 false，必须通过调用 close 方法关闭 RPC 客户端，以清理资源和避免任何资源的泄漏。这描述为下面的代码段：

```
if (client == null) {
    client = RpcClientFactory.getDefaultInstance(host, port);
} else if (!client.isActive()) {
    client.close();
    client = RpcClientFactory.getDefaultInstance(host, port);
}
```

所有 RPC 客户端的公共配置参数

所有的 RPC 客户端有一些公共的配置参数。接下来我们将讨论这些参数是如何被传递到 RPC 客户端的。表 7-1 列出了 RPC 客户端公共的配置参数。

171

表7-1 RPC客户端公共的配置参数

参数	默认值	描述
client.type	-	该参数值必须设置为 default、default_loadbalance、default_failover 或 thrift。
batch-size	100	每个批量发送事件的最大数量
hosts	-	可以用来指定主机参数的名字列表
hosts.<hostalis>	-	以 hostname:port 格式的主机的配置

client.type参数指定了使用的RPC客户端的类型，该值可以是default、default_loadbalance、default_failover或thrift中的一个。我们将在接下来的一节中一一讨论这些类型。

使用batch-size参数可以设置每个批量发送事件的最大数量。如果多于这个数量的事件被传递到单个appendBatchmethod调用，那么就发送和指定批量大小相同或更少事件的多个批量操作。

hosts参数列出了一些别名，这些别名将用于识别主机，主机与客户端必须连接（默认RPC客户端的情况下，除列表中的第一个主机之外的所有主机被忽略）。该主机名信息必须通过列表中第一个主机的hosts.<hostname>参数来传递，使用hostname:port的格式。这种配置的一个例子如下所示：

```
hosts = h1 h2 h3
hosts.h1 = usingflume1.oreilly.com:5545
hosts.h2 = usingflume2.oreilly.com:5545
hosts.h3 = usingflume3.oreilly.com:5545
```

RPC客户端通过使用RpcClientFactory#getInstance方法进行创建，这在前面“创建Flume RPC客户端”一节解释过。当RPC客户端通过RpcClientFactory返回到应用程序，它是完全配置的且准备好被使用。RPC客户端可以在RPC调用中发送一个事件或一个批量的事件。当RPC客户端实例被创建后，批量的大小可以通过传递到RpcClientFactory的Properties实例进行配置。例7-3展示了一个简单类，该类通过RPC使用随机字符串生成类，然后将批量的事件写到Flume Agent。当我们讨论不同种类RPC客户端时，我们将使用UsingFlumeRPCApp作为基类。

例7-3 Flume RPC客户端使用示例

```
package usingflume.ch07;

public abstract class UsingFlumeRPCApp {

  private static final Logger LOGGER = LoggerFactory.getLogger(
    UsingFlumeRPCApp.class);

  private RpcClient client;
  private final Properties config = new Properties();
  private final ExecutorService executor
    = Executors.newFixedThreadPool(5);
  private int batchSize = 100;

  protected void parseCommandLine(String args[])
    throws ParseException {
```

172

```
setClientTypeInConfig(config);
Options opts = new Options();

Option opt = new Option("r", "remote", true,
  "Remote host to connect " +
    "to");
opt.setRequired(true);
opts.addOption(opt);

opt = new Option("h", "help", false, "Display help");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("b", "batchSize", true, "Batch Size to use");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("c", "compression", false, "If set, " +
  "data is compressed before sending");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("l", "compression-level", false,
  "The compression level " +
    "to use if compression is enabled");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("s", "ssl", false,
  "If set, ssl is enabled using keystore supplied by argument k");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("k", "keystore", true,
  "Keystore to use with SSL");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("d", "keystore-password", true,
  "Password for keystore");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("t", "keystore-type", true,
  "Type keystore");
```

```
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("i", "maxIoWorkers", true,
  "Set the maximum number of " +
    "worker threads to use for network IO");
opt.setRequired(false);
opts.addOption(opt);

opt = new Option("o", "backoff", false,
  "Backoff failed clients?");
opt.setRequired(false);
opts.addOption(opt);

Parser parser = new GnuParser();
CommandLine commandLine = parser.parse(opts, args);

if (commandLine.hasOption("h")) {
  new HelpFormatter().printHelp("UsingFlumeDefaultRPCApp", opts,
    true);
  return;
}

parseHostsAndPort(commandLine, config);

if (commandLine.hasOption("b")) {
  String batchSizeStr = commandLine.getOptionValue("b", "100");
  config.setProperty(CONFIG_BATCH_SIZE, batchSizeStr);
  batchSize = Integer.parseInt(batchSizeStr);

}

if (commandLine.hasOption("c")) {
  config.setProperty(CONFIG_COMPRESSION_TYPE, "deflate");
  if (commandLine.hasOption("l")) {
    config.setProperty(CONFIG_COMPRESSION_LEVEL,
      commandLine.getOptionValue("l"));
  }
}

if (commandLine.hasOption("s") && commandLine.hasOption("k") &&
commandLine.hasOption("d")) {
  config.setProperty(CONFIG_SSL, "true");
  config.setProperty(CONFIG_TRUSTSTORE, commandLine.getOptionValue("k"));
  config.setProperty(CONFIG_TRUSTSTORE_PASSWORD, commandLine.
```

```
getOptionValue("d"));
      if (commandLine.hasOption("t")) {
        config.setProperty(CONFIG_TRUSTSTORE_TYPE, commandLine.
getOptionValue("t"));
      }
    }

    if (commandLine.hasOption("i")) {
      config.setProperty(MAX_IO_WORKERS,
        commandLine.getOptionValue("i"));
    }
    backoffConfig(commandLine, config);
  }

  protected abstract void setClientTypeInConfig(Properties p);

  protected abstract void parseHostsAndPort(CommandLine commandLine,
    Properties config);

  protected abstract void backoffConfig(CommandLine commandLine,
    Properties config);

  @VisibleForTesting
  protected void run(String[] args) throws Exception {
    parseCommandLine(args);

    final UsingFlumeRPCApp app = this;

    for (int i = 0; i < 5; i++) {
      executor.submit(new Runnable() {
        final int total = 100;
        @Override
        public void run() {
          int i = 0;
          while (i++ < total) {
            app.generateAndSend();
          }
        }
      }).get();
      app.closeClient();
    }

    // Set a shutdown hook to shutdown all the threads and the
    // executor itself
    Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
```

```
    @Override
    public void run() {
      executor.shutdown();
      try {
        if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
          executor.shutdownNow();
        }
      } catch (InterruptedException e) {
        LOGGER.warn(
          "Interrupted while attempting to shutdown executor. " +
            "Force terminating the executor now.", e);
        executor.shutdownNow();
      }
      app.closeClient();
    }
  }));

}

private synchronized void reconnectIfRequired() {
  if (client != null && !client.isActive()) {
    closeClient();
  }
  // If client is null, it was either never created or was closed by
  // closeClient above
  if (client == null) {
    try {
      client = RpcClientFactory.getInstance(config);
    } catch (Exception e) {
      e.printStackTrace();
      LOGGER.warn("Client creation failed. Source may not have been started
yet");
    }
  }
}

protected synchronized void closeClient() {
  if(client != null) {
    client.close();
  }
  client = null;
}

protected void generateAndSend() {
  reconnectIfRequired();
```

```
    List<Event> events = new ArrayList<Event>(100);
    for (int i = 0; i < batchSize; i++) {
      events.add(EventBuilder.withBody(
        RandomStringUtils.randomAlphanumeric(100).getBytes()));
    }
    try {
      client.appendBatch(events);
    } catch (Throwable e) {
      e.printStackTrace();
      LOGGER.error(
        "Error while attempting to write data to remote host at " +
          "%s:%s. Events will be dropped!");
      // The client cannot be reused, since we don›t know why the
      // connection
      // failed. Destroy this client and create a new one.
      reconnectIfRequired();
    }
  }
}
```

176

正如你所看到的，UsingFlumeRPCApp 是一个抽象类，它的具体实现将在下面一节进行描述。使用相同的类写数据到更复杂的 RPC 客户端，例如实现了 setClientTypeInConfig 方法的 load-balancing RPC 客户端，它将在配置中设置需要的参数来实例化 RPC 客户端的合适类型。parseHostsAndPort 方法的实现将主机名参数转换为每个 RPC 客户端所需要的格式，backoffConfig 方法在命令行输出的基础上启用回退机制。

parseCommandLine 方法读取几个参数，包括要连接的主机名和端口，是否使用压缩和 SSL，失败时是否回退等，作为命令行参数。然后这些参数被映射到相应的 RPC 客户端配置参数，并且被插入到传递给 getInstance 方法的 Properties 实例，来创建客户端。一个合适的指定参数的安全方法是使用 RpcClientConfigurationConstants 类，用来指明所有的配置参数作为最终的静态字符串。在这个例子中，这些字符串使用静态输入，且当它们传递给 Properties 实例时，用来指明配置参数。

RPC 客户端的方法是线程安全的，但是多线程的应用可能会自己实现一些线程同步，确保当一个线程尝试写数据时，另一个线程不会关闭客户端或设置它为 null，否则会导致 NullPointerException 异常。

例 7-3 展示过的应用程序中，如果任何写数据到远端主机的线程遇到异常，客户端会立即关闭并强制再次连接。如果此时有其他的线程使用该客户端，它们也会遇到异常——这是设计精巧的，因为只要它抛出异常，就认为 Flume 的默认 RPC 客户端是死亡的。

此时，多个线程中的一个必须创建一个新的连接。所有的线程重新连接会导致一次建立

多个客户端，即使只需要一个。因此，我们使用同步的方法 reconnectIfRequired 创建一个新连接，只有当客户端不再活跃或为null才强制重连接(由于没有创建)。通过这样做，我们能确保一旦新的 RPC 客户端创建且可用，其他的线程将不会创建新的连接。当要求关闭时，除了关闭执行器和线程，应用程序也会关闭 RPC 客户端实例以清理资源。

run 方法生成固定数量的事件，且使用 RPC 客户端实例批量发送事件到远程 Flume Agent。简单来说，这个例子只是使用随机生成字符串的字节数组表示方式作为的事件 body。

默认 RPC 客户端

默认 RPC 客户端实例使用 Avro RPC 协议且只能连接到一个 Avro Source。因为 Java 程序写数据到一个 Flume Agent，所以这是推荐的客户端。为了创建默认 RPC 客户端，可以使用 RpcClientFactory 提供的一对便捷的方法。

```
public static RpcClient getDefaultInstance(String hostname, Integer port);
public static RpcClient getDefaultInstance(String hostname, Integer
port,Integer batchSize);
```

因为默认 RPC 客户端写数据到一个 Avro Source，需要连接到该 Avro Source 的参数只是主机名和端口信息，可以被传递到这里展示的第一个方法分别作为第一个和第二个参数。第二个方法还需要另外的一个参数，即批量的大小——在一个 RPC 调用中发送事件的客户端的最大数量。因为第一个方法不需要批量的大小，该方法创建的 RPC 客户端只是使用默认的批量大小，即每个批量 100 个事件。在使用该 RPC 客户端之前就没有其他需要的配置了。

除了在上一节讨论的参数，还有一些更多的参数可以用来加密发送到 Agent 的数据或压缩数据。表 7-2 描述了这些参数。

表7-2 默认RPC客户端配置

参数	默认值	描述
connect-timeout	20	等待初始连接完成的超时时间，以秒为单位
request-timeout	20	等待一个批量成功完成写该批量事件的超时时间，以秒为单位
compression-type	-	使用的压缩算法。该值可以为 deflate 或不用设置
compression-level	6	如果通过使用 compression-type 参数启用压缩，该参数为使用的压缩级别。合法值是 1-9。数值越高，压缩越好

续表

参数	默认值	描述
ssl	false	如果设置为 true，则启用 SSL
trust-all-certs	false	如果设置为 true，所有的 SSL 证书都是可信任的
truststore	-	使用的 trust store。如果 SSL 启用，该参数是必需的
truststore-password	-	用来打开 trust store 的密码
truststore-type	JKS	使用的 trust store 的类型
maxIoWorkers	10	和远程机器通信使用线程最大数量的

178

这里面展示的很多参数对于 Avro Sink 和 RPC 客户端都是公共的。原因是 Avro Sink 实际上就是一个 Channel，且感知事务的包装器包装了默认 RPC 客户端。因此，所有默认 RPC 客户端支持的特性可能 Avro Sink 也都支持（但是对于用户可能是不透明的）。

为了传递这些参数，必须使用 RpcClientFactory 方法接收 Properties 实例（本章前面“创建 Flume RPC 客户端”一节讲述）。

为了使用默认 RPC 客户端更高级的配置，在 Properties 对象中 client.type 参数值必须设置为 default（或 useRpcClientConfigurationConstants.DEFAULT_CLIENT_TYPE）。

batch-size 参数指明了每个 RPC 调用应该发送事件的最大数量。如果传递到单个 appendBatch 调用的事件数量大于批量大小，那么事件将会分为多个 RPC 调用进行发送。

connect-timeout 参数指明了等待初始连接建立完成的时间周期；如果超时，客户端会抛出 EventDeliveryException 异常。这只是在第一个批量事件发送时才做的工作。request-timeout 参数指明了单个 RPC 调用被认为失败前等待的时间周期。如果一个 RPC 调用花费的秒数大于该参数指定的值，则抛出 EventDeliveryException 异常。

在 Agent 期望压缩的数据时，RPC 客户端可以在发送数据到该 Agent 时压缩数据。为了发送压缩数据，设置 compression-type 参数值为 deflate。使用 compression-level 参数设置压缩的级别，参数值的范围在 1 到 9 之间。随着压缩级别的升高，压缩率也会随之提升，数据压缩的时间也会增加。你可以在 zlib 手册中 [zlib-manual] 读到更多关于 deflate 压缩和压缩级别的细节。

179

压缩类型不匹配

如果 RPC 客户端配置使用压缩，接收事件的 Avro Source 必须将 compression-type 参数值设置为 deflate。

要启用 SSL，那么设置 ssl 值为 true。客户端可以通过设置 trust-all-certs 参数值为 true 以信任所有的 SSL 证书——在生产环境中为了避免安全问题不应该这么做。*trust store* 是包含信任公钥的文件。除非另有指示，Flume 将使用 Java 的默认 JSSE 证书授权文件，*jssecacerts/cacerts*，来决定是否信任 Avro Source 的 SSL 证书。

如果使用自定义的 trust store，设置 truststore 参数值为各自 trust store 文件的路径 [truststore]。运行 Agent 的用户应该有读取该文件的权限。truststore-password 参数值必须设置为可以打开 trust store 的密码。truststore-type 参数是可选的，如果需要可以设置为另一个 keystore 类型 [truststore-type] 。

可以通过设置 maxIoWorkers 参数值来限制 RPC 客户端生成的工作线程数量。默认情况下该值为 10，表示执行网络 I/O 操作的最大工作线程是 10 个。

为了配置 UsingFlumeRPCApp 使用默认 RPC 客户端，我们重写例 7-4 展示的 setClientTypeInConfig、parseHostsAndPort 和 backoffConfig 方法。

例7-4 默认Flume RPC客户端使用示例

```
package usingflume.ch07;

public class UsingFlumeDefaultRPCApp extends UsingFlumeRPCApp {
  private String remote;

  @Override
  protected void setClientTypeInConfig(Properties p) {
    p.setProperty(CONFIG_CLIENT_TYPE, DEFAULT_CLIENT_TYPE);
  }

  @Override
  protected void parseHostsAndPort(CommandLine commandLine,
    Properties config) {
    config.setProperty(CONFIG_HOSTS, "h1");

    remote = commandLine.getOptionValue("r").trim();
    Preconditions.checkNotNull(remote, "Remote cannot be null.");
    // This becomes hosts.h1
    config.setProperty(CONFIG_HOSTS_PREFIX + "h1", remote);
  }

  @Override
  protected void backoffConfig(CommandLine commandLine,
    Properties config) {
    // No op
  }
```

```
  public static void main(String args[]) throws Exception {
    // Outsource all work to the app.run method which can be tested
    // more easily
    final UsingFlumeDefaultRPCApp app = new UsingFlumeDefaultRPCApp();
    app.run(args);
  }
}
```

对于该默认 RPC 客户端，在 `setClientTypeInConfig` 方法中将 `client.type` 设置为 `default`。`parseHostsAndPort` 方法只是从命令行读取主机和端口，然后将 `hosts` 参数值设置为 h1，表示该主机使用别名 `h1` 进行了标识。`hosts.h1` 参数值设置为 hostname:port 格式的主机名和端口。

要启用 SSL 或压缩，或指定批量大小、连接超时时间、请求超时时间等，所有默认 RPC 客户端的参数都可以传递到 load-balancing RPC 客户端或 failover RPC 客户端。

Load-Balancing RPC 客户端

load-balancing RPC 客户端与 load-balancing Sink 处理器工作原理类似。load-balancing RPC 客户端可以设置发送事件到多个客户端。对于每个 `append` 或 `appendBatch` 调用，load-balancing RPC 客户端，基于配置，以 *random* 或 *round-robin* 的顺序，选择配置要发送数据的多个 Agent 的其中一个。

当应用程序调用 `append` 或 `appendBatch`，load-balancing RPC 客户端尝试以此发送事件到多个 Flume 的 Agent，直到数据真正发送成功。如果一个远程 Agent 失败且该 RPC 客户端尝试另一个主机，那么应用程序将不会知道发生失败也得不到异常信息，直到所有的主机都试过且都失败。如果所有远程 Agent 都失败，那么 `append` 和 `appendBatch` 将抛出 `EventDeliveryException` 异常。

所有可以传递到默认 RPC 客户端的参数（表 7-2 展示的参数）也可以传递到 load-balancing RPC 客户端，且与在默认 RPC 客户端的参数产生相同的行为。例如，SSL 和压缩可以使用默认 RPC 客户端中相同的参数来启用。

181 为了配置 load-balancing RPC 客户端，表 7-3 列出的参数可以通过 `Properties` 实例进行传递（除了表 7-1 和表 7-2 中展示的参数）。

表7-3 load-balancing RPC客户端配置

参数	默认值	描述
backoff	false	如果设置为 true，一个失败的主机将不再以指数增长的回退周期重连
maxBackoff	-	回退一个失败 Agent 的最大时间，以毫秒为单位
host-selector	round_robin	选择发送数据到达主机的顺序

正如我们前面所讨论的，load-balancing RPC 客户端可以选择写事件到众多主机中的一个。当一个主机连接失败且不接受数据（或太慢，或网络连接失败），load-balancing RPC 客户端可以将该主机加入黑名单以获得成倍增加的回退时间（类似于 load-balancing Sink 处理器）。每次回退时间到期，load-balancing RPC 客户端就尝试写事件到该主机，如果再次失败，回退时间翻倍；否则，后退时间将重置为零且该主机被认为是活跃的。如果 backoff 参数设置为 true，则启用指数后退机制。回退时间没有默认的最大上限，但该最大上限可以使用 maxBackoff 参数设置。一旦一个主机的回退时间已经达到这个值，那么重试该主机的回退时间直到没有任何进一步的增加的可能性。

host-selector 参数指定了客户端选择发送事件到达主机的策略。如果设置为 random，那么将随机选择一个没有处于回退时期的主机。如果设置为 round_robin，那么将基于主机在 hosts 列表指定的顺序循环选择主机。该值也可以设置为实现了 LoadBalancingRpcClient$HostSelector 类的完全限定类名。下一节“编写自己的主机选择器 *”解释如何编写一个自定义主机选择器。

例 7-3 通过改变应用程序接受主机的方式，可以很简单地修改使用 load-balancing RPC 客户端，然后解析主机列表将它们设置为名称为 h1、h2、h3 等的主机。对于每个主机的别名，需要以 hostname:port 的格式指定主机的地址，名称为 hosts.h1、hosts.h2、hosts.h3 等参数的值设置为主机地址。

客户端设置为 default_loadbalance。为了切换到 load-balancing RPC 客户端，那么应该按照如下方式修改 setClient 方法：

```
protected void setClientTypeInConfig(Properties p) {
  p.setProperty(CONFIG_CLIENT_TYPE, "default_loadbalance");
}
```

182

在例 7-5 展示了该方面的内容。

例7-5 load-balancing Flume RPC客户端使用示例

```
package usingflume.ch07;

public class UsingFlumeLBRPCApp extends UsingFlumeRPCApp {

  private String host;
  private String port;

  @Override
  protected void setClientTypeInConfig(Properties p) {
    p.setProperty(CONFIG_CLIENT_TYPE, "default_loadbalance");
  }

  protected void parseHostsAndPort(CommandLine commandLine,
    Properties config) {
    host = commandLine.getOptionValue("r").trim();
    Preconditions.checkNotNull(host, "Remote host cannot be null.");
    StringBuilder hostBuilder = new StringBuilder("");

    String[] hostnames = host.split(",");
    int hostCount = hostnames.length;

    for (int i = 1; i <= hostCount; i++) {
      hostBuilder.append("h").append(i).append(" ");
    }
    config.setProperty(CONFIG_HOSTS, hostBuilder.toString());

    for (int i = 1; i <= hostCount; i++) {
      config.setProperty(
        CONFIG_HOSTS_PREFIX + "h" + String.valueOf(i),
        hostnames[i - 1]);
    }
  }

  @Override
  protected void backoffConfig(CommandLine commandLine,
    Properties config) {
    if (commandLine.hasOption("o")) {
      config.setProperty(CONFIG_BACKOFF, "true");
    }
  }

  public static void main(String args[]) throws Exception {
    // Outsource all work to the app.run method which can be tested
    // more easily
```

```
    final UsingFlumeLBRPCApp app = new UsingFlumeLBRPCApp();
    app.run(args);
  }
}
```

183

因为 load-balancing RPC 客户端支持回退失败的主机，基于命令行配置的 backoffConfig 方法在 Properties 实例中设置了该项。

正如前面提到的，load-balancing RPC 客户端还接受传递给默认 RPC 客户端的参数。因此，所有在 parseCommandLine 方法传递给默认的 RPC 客户端的参数，也用于 load-balancing RPC 客户端的配置。

编写自己的主机选择器 *

round-robin 或 random 选择器满足大部分选择 Sink 用于从 Channel 拉取数据的用例。但是总有一些情况需要你编写自己的算法以选择处于活跃状态的 Sink。如果要编写选择器，必须实现 LoadBalancingRpcClient$HostSelector 接口。例 7-6 展示了该接口。

例7-6 HostSelector接口

```
public interface HostSelector {
  void setHosts(List<HostInfo> hosts);
  Iterator<HostInfo> createHostIterator();
  void informFailure(HostInfo failedHost);
}
```

当 Flume 启动 Sink 处理器时，创建了主机选择器且调用 setHosts 方法，Flume 传递 HostInfo 实例 [hostInfo] 的列表给该方法，HostInfo 实例包含应该发送事件到哪个主机的信息。对于来自客户端的每个 append 或 appendBatch 调用，load-balancing RPC 客户端调用 createHostIterator 方法，该方法必须返回一个 HostInfo 实例的迭代器，HostInfo 实例必须顺序返回客户端应该发送事件的主机。当事件被成功发送到一个主机，该迭代器被丢弃且再次调用 createHostIterator 方法。如果一个主机已经失败或发送数据到主机时失败，那么 RPC 客户端调用 informFailure 方法，传递描述该主机的 HostInfo 实例。这可以暂时用来拉黑主机。

当创建主机选择器时，如前面描述的一样，在你的工程的 *pom.xml* 文件的依赖部分添加 flume-ng-sdk 组件。

Failover RPC 客户端

184

failover RPC 客户端与 failover Sink 处理器的工作原理类似，基于优先级连接 Agent。该 RPC 客户端首先连接优先级最高的 Agent。如果这个 Agent 失败，该客户端就连接下一

个最高优先级的 Agent。与 failover Sink 处理器不同的是，failover RPC 客户端不需要显式地设置优先级，反而优先级是基于 hosts 参数中主机的顺序。hosts 参数中指定的第一个主机有最高的优先级，紧接着是列表中的第二个主机，以此类推。表 7-4 列出了 failover RPC 客户端的配置参数。

表7-4 failover RPC客户端配置

参数	默认值	描述
client.type	default_failover	
max-connections	5	在抛出异常前试图连接主机的最大次数

failover RPC 客户端的 client.type 是 default_failover。除了表 7-1 和表 7-2 中的参数，该客户端唯一的参数是 max-connection。该值为在一个 append 或 appendBatch 调用中，客户端必须故障转移的次数。例如，如果该参数设置为 5，如果有 5 个或更多的主机，那么尝试连接主机的次数最高为 5 次。如果有更少的主机，那么只会尝试连接仅有的主机。

如果要使用 failover Sink 处理器，那么必须重写例 7-3 中的 setClientTypeInConfig 方法。

```
protected void setClient(Properties p) {
  p.setProperty(CONFIG_CLIENT_TYPE, "default_failover");
}
```

Thrift RPC 客户端

Apache Thrift 是一个数据序列化和 RPC 的框架，用来序列化和反序列化来自不同语言的数据。Thrift 通过拥有语言无关的数据格式规范来支持该特性。然后 Thrift 编译器可以生成不同语言的代码，用来读取和写入数据。为了从不同语言的应用程序发送数据到 Flume，Flume Thrift spec[flume-thrift] 可以用来生成在指定语言使用的代码。例 7-7 展示了 Flume Thrift spec 文件的内容。

185

例7-7 Flume Thrift说明

```
namespace java org.apache.flume.thrift

struct ThriftFlumeEvent {
  1: required map <string, string> headers,
  2: required binary body,
}

enum Status {
  OK,
  FAILED,
```

```
  ERROR,
  UNKNOWN
}

service ThriftSourceProtocol {
  Status append(1: ThriftFlumeEvent event),
  Status appendBatch(1: list<ThriftFlumeEvent> events),
}
```

在 Apache Thrift 指导 [thrift-doc] 中可以发现关于生成不同语言的 Thrift 代码的更多细节。

我们不会详细讨论 Thrift RPC 客户端，因为该客户端只用于当 Thrift Source 已经运行在从另一种语言编写的应用程序中接收数据的 Agent 上时（也只是为了避免运行一个 Avro Source）。如果发送的数据只是 Java，那么 Avro Source 应该使用 default、load-balancing 或 failover RPC 客户端。

嵌入式 Agent

在应用程序中使用 Flume RPC 客户端有一个非常明显的问题是在失败和重试的情况下，应用程序必须缓冲数据。这意味着即使使用 load-balancing 或 failover RPC 客户端（可能也会影响延迟），下游失败也可以直接影响应用程序。不能承受丢失消息的应用程序现在不得不缓冲事件。以这种在程序或机器失败中存活下来的方式缓冲事件，是耗时且实现起来令人痛苦不堪的。缓冲，不影响被程序发送数据——听起来像一个已经解决了的问题，不是吗？ Flume 已经被设计来解决这个问题！

为了解决这个问题，Flume 提供了*嵌入式* Agent。嵌入式 Agent 是一类很受限制的 Flume Agent，可以部署在第三方应用程序。比起 RPC 客户端，使用嵌入式 Agent 的优点是该 Agent 有一个 Channel，一个应用程序不需要管理的缓冲区，也可以在磁盘上。这使得应 186
用程序能更长时间地拯救下游失败的 Agent，直到 Channel 满。

当等待下游 HDFS 失败和由此产生的满的 Channel 得到解决时，嵌入式 Agent 通过在应用程序中缓冲数据来缓冲事件。因为每个应用程序过程通常产生的只有一小部分数据被第一层 Flume Agent 处理，所以在每个应用程序的嵌入式 Agent 拥有一个规模小得多的 Flume Channel 大小就足够了。

图 7-1 展示了一个经典的嵌入式 Agent 架构。这个嵌入式 Agent 被应用程序配置且启动。因为 Agent 是嵌入到应用程序中的，所以它存在于应用程序的进程地址空间，且在应用程序中创建和运行线程。因此，比起只是使用 RPC 客户端，应用程序将消耗更多的资源。这就是使用嵌入式 Agent 提供额外缓冲时间所付出的代价。

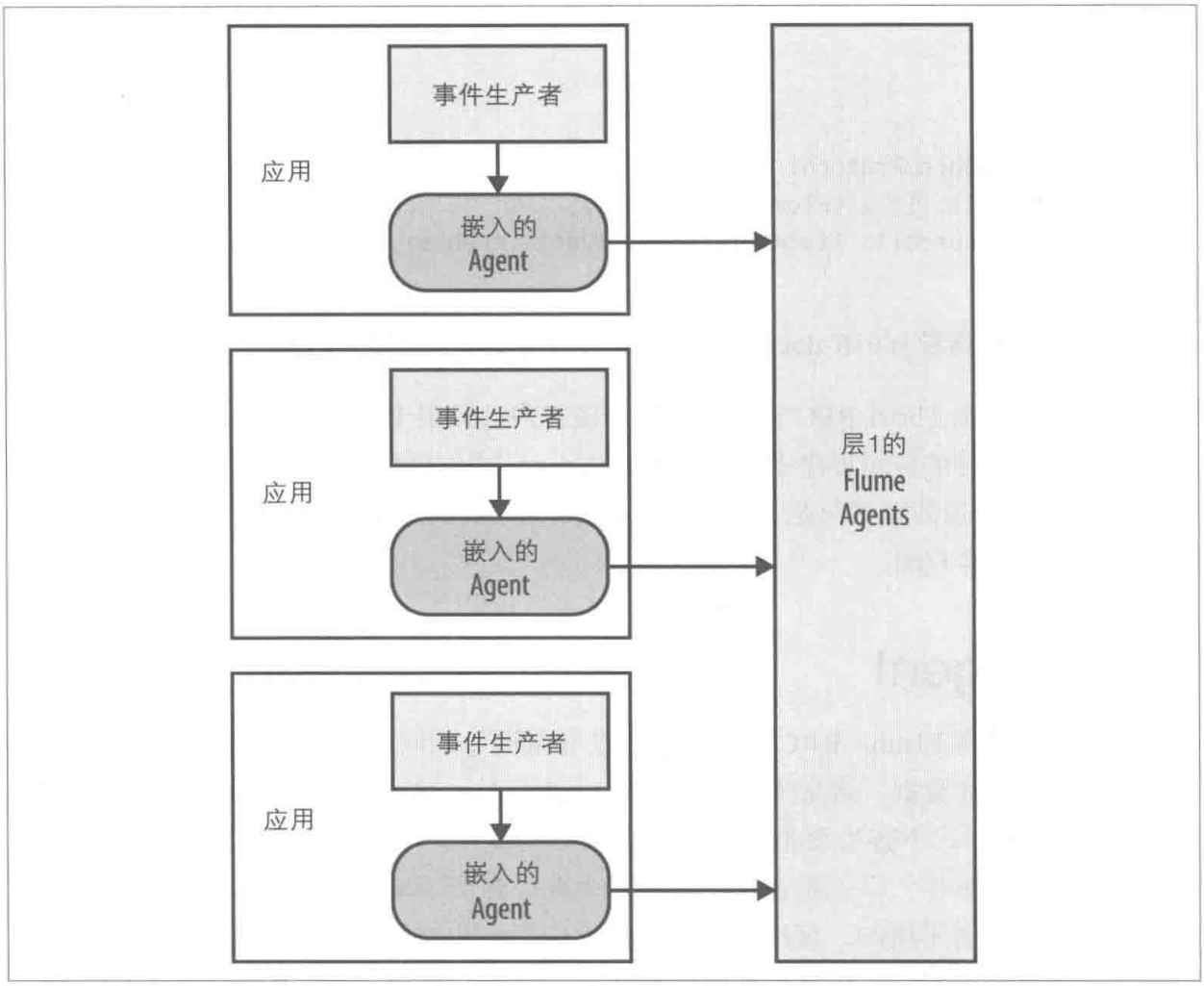


图7-1 嵌入式Agent结构

嵌入式 Agent 也支持使用 File Channel，即使应用程序死亡或机器重启，也能帮助持久
187 化事件。这增加了应用程序执行的 I/O，导致了机器上更多的负载，尤其是 File Channel 使用的磁盘。在第 4 章也讨论过，当没有其他进程使用磁盘时 File Channel 性能较好。在这种情况下，即使嵌入该 Agent 的应用程序也最好不要使用相同的磁盘，以避免不必要的搜索。

嵌入式 Agent 公开了一套非常简单的 API，用于启动 Agent 然后操作本身直到停止。它由一个名为 EmbeddedAgent 的类表示。嵌入式 Agent 与其他的 Flume Agent 具有相同的生命周期。首先通过需要一个名字的公共构造函数创建 EmbeddedAgent 类的实例，进而创建嵌入式 Agent，该名字指的是这个特定 Agent 的名称。异常和日志消息将包含该 Agent 的名字，所以如果不止一个嵌入式 Agent 被部署在同一个应用程序实例中，用户可以识别哪个 Agent 有问题。

嵌入式 Agent 名称中的空格

请注意，目前嵌入式 Agent 名称不能包含空格。名字中的空格将导致嵌入式 Agent 抛出异常无法启动，说明配置是不好的。

一旦它被创建且使用 configure 方法进行了配置，即传递一个包含 Agent 配置的 Map 集合。如果配置失败，这种方法不会有返回，将抛出 FlumeException 异常。

然后开始使用 start 方法启动 Agent。启动失败将导致该方法抛出 FlumeException 异常。该方法初始化所有配置的组件并且启动它们。

事件可以使用 put 或 putAll 方法写入到 Agent，这两个方法分别接受单个事件和批量事件。如果由于任何原因不能写入事件到 Channel，包括但不限于 Channel 已满，将抛出一个 EventDeliveryException 异常。

当 Agent 已经准备好关闭 Agent 时，可以使用 stop 方法停止，如果停止失败这可能会抛出一个 FlumeException 异常。例 7-8 展示了 Embedded Agent API。

例7-8 嵌入式Agent API

```
public class EmbeddedAgent {
        public EmbeddedAgent(String name);
        public void configure(Map<String, String> configuration) throws
        FlumeException;
        public void start() throws FlumeException;
        public void put(Event event) throws EventDeliveryException;
        public void putAll(List<Event> events) throws EventDeliveryException;
        public void stop() throws FlumeException;
}
```

188

配置嵌入式 Agent

一个嵌入式 Agent 只能有一个 Source、一个 Channel 和一个 Sink 组（从技术上讲，受限制的是 Sink 运行器的数量），但是 Sink 组可以包含多个 Sink。表 7-5 列出了嵌入式 Agent 的配置参数。

表7-5 嵌入式Agent配置

参数	默认值	描述
source.type	embedded	嵌入式 Agent 唯一可以使用的 Source 是 embedded Source
channel.type	-	使用的 Channel 类型

续表

参数	默认值	描述
channel.*	-	传递给 Channel 的配置参数
sinks	-	该 Agent 中 Sink 的名字。这和标准 Agent 配置的活跃集合的 <agentname>.sinks 行相等
<sinkname>.type	-	嵌入式 Agent 只能包含 Avro Sink，所以该值必须设置为 avro
<sinkname>.*	-	传递给每个 Sink 的配置参数
processor.type	-	如果指定了多个 Sink，Sink 处理器用来选择活跃 Sink
processor.*	-	传递给 Sink 处理器的配置参数

不需要指定使用的 Source，因为 Agent 自动设置使用嵌入式 Source。即使 Source 通过配置进行设置，也必须设置为 embedded。嵌入式 Source 没有任何必需的参数。

每个嵌入式 Agent 只能有一个 Channel，为 File Channel 或 Memory Channel，所以 channel.type 参数接受 file 或 memory 作为它的值。可以使用 channel. 前缀传递任何可以传递的参数到 Channel。

一个嵌入式 Agent 可以有多个 Sink。如果一个嵌入式 Agent 可以有多个 Sink，这些 Sink 将被分组到一个 Sink 组内，因为一个嵌入式 Agent 只有一个 Sink 运行器。因为嵌入式 Agent 可以有多个 Sink，它们使用 sinks 参数命名，通过该参数指定它们的名字。使
189 用 Sink 名字作为前缀可以传递任何参数到 Sink。嵌入式 Agent 中只能使用 Avro Sink，所以所有 Sink 的 type 参数必须设置为 avro。如果有多个指定的 Sink，这些 Sink 将自动分组到一个 Sink 组，可以使用 processor.type 指定其 Sink 处理器的类型；通过 processor. 前缀可以传递给 Sink 处理器任何的配置。

因为一个嵌入式 Agent 中只有一个 Channel，Source 和 Sink 不需要明确地指定 channels 或 channel 参数，Agent 将自动设置 Source 写数据到唯一的 Channel 且 Sink 从中读取数据。例 7-9 展示了一个使用 Embedded Agent API 的应用程序，可以通过负载均衡写数据到若干 Flume Agent。

例7-9 嵌入式Agent示例

```
package usingflume.ch07;

public class UsingFlumeEmbeddedAgent {
  private static final Logger LOGGER = LoggerFactory.getLogger
    (UsingFlumeEmbeddedAgent.class);
  private final EmbeddedAgent agent = new EmbeddedAgent(
```

```
    "UsingFlume");
  private int batchSize = 100;

  public static void main(String args[]) throws Exception {
    UsingFlumeEmbeddedAgent usingFlumeEmbeddedAgent = new
      UsingFlumeEmbeddedAgent();
    usingFlumeEmbeddedAgent.run(args);
    int i = 0;
    while (i++ < 100) {
      usingFlumeEmbeddedAgent.generateAndSend();
    }
  }

  public void run(String args[]) throws Exception {
    Options opts = new Options();

    Option opt = new Option("r", "remote", true,
      "Remote host to connect " +
        "to");
    opt.setRequired(true);
    opts.addOption(opt);

    opt = new Option("p", "port", true, "Port to connect to");
    opt.setRequired(true);
    opts.addOption(opt);

    opt = new Option("b", "batchSize", true, "Batch Size to use");
    opt.setRequired(false);
    opts.addOption(opt);

    Parser parser = new GnuParser();
    CommandLine commandLine = parser.parse(opts, args);

    if (commandLine.hasOption("h")) {
      new HelpFormatter().printHelp("UsingFlumeEmbeddedAgent", opts,
        true);
      return;
    }

    Map<String, String> config = new HashMap<String, String>();
    parseHostsAndPort(commandLine, config);
    config.put("source.type", "embedded");
    File dcDir = Files.createTempDir();
    dcDir.deleteOnExit();
    config.put("channel.type", "file");
```

190

```
    config.put("channel.capacity", "100000");
    config.put("channel.dataDirs", dcDir.toString() + "/data");
    config.put("channel.checkpointDir", dcDir.toString() + "/checkpoint");
    agent.configure(config);
    agent.start();
    Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
      @Override
      public void run() {
        agent.stop();
      }
    }));
  }

  private void generateAndSend() {
    List<Event> events = new ArrayList<Event>(100);
    for (int i = 0; i < batchSize; i++) {
      events.add(EventBuilder.withBody(
        RandomStringUtils.randomAlphanumeric(1024).getBytes()));
    }
    try {
      agent.putAll(events);
    } catch (Throwable e) {
      LOGGER.error(
        "Error while attempting to write data to remote host at " +
          "%s:%s. Events will be dropped!");
      // The client cannot be reused, since we don›t know why the
      // connection
      // failed. Destroy this client and create a new one.
    }
  }

  private void parseHostsAndPort(CommandLine commandLine,
    Map<String, String> config) {
    String host = commandLine.getOptionValue("r").trim();
    Preconditions.checkNotNull(host, "Remote host cannot be null.");

    String port = commandLine.getOptionValue("p").trim();
    Preconditions.checkNotNull(port, "Port cannot be null.");

191   String[] hostnames = host.split(",");
    int hostCount = hostnames.length;
    final String sinkStr = "sink";
    StringBuilder stringNamesBuilder = new StringBuilder("");
    for (int i = 0; i < hostCount; i++) {
      stringNamesBuilder.append(sinkStr).append(i).append(" ");
```

```
    }
    // this puts sinks = sink0 sink1 sink2 sink 3 etc...
    config.put("sinks", stringNamesBuilder.toString());
    final String parameters[] = {"type", "hostname", "port",
                                 "batch-size"};
    final String avro = "avro";
    for (int i = 0; i < hostCount; i++) {
      final String currentSinkPrefix = sinkStr + String.valueOf(i) +
        ".";
      config.put(currentSinkPrefix + parameters[0], avro);
      config.put(currentSinkPrefix + parameters[1], hostnames[i]);
      config.put(currentSinkPrefix + parameters[2], port);
      config.put(currentSinkPrefix + parameters[3],
        String.valueOf(batchSize));
    }

    if (hostnames.length > 1) {
      config.put("processor.type", "load_balance");
      config.put("processor.backoff", "true");
      config.put("processor.selector", "round_robin");
      config.put("processor.selector.maxTimeout", "30000");
    } else {
      config.put("processor.type", "default");
    }
  }
}
```

应用程序从 `parseHostsAndPort` 方法中的命令行读取主机名和端口号，且配置 Avro Sink 连接它们当中的每一个。如果命令行中指定了若干主机，那么配置中要添加 load balancing Sink 处理器，以确保数据能发送到应用程序连接的所有主机。它也配置带有 100000 容量的 File Channel，且使用检查点和数据目录。这些参数都设置在一个 Map 集合中，稍后将被传递到 EmbeddedAgent 类的 configure 方法。

一旦配置完成，就通过调用 `start` 方法启动 Agent，然后该 Agent 可以通过 `put` 和 `putAll` 方法接收来自应用程序的事件。在这种情况下，使用 `putAll` 方法将预先定义好数量的随机生成的事件传递到该 Agent。

当应用程序完成发送数据，通过调用 `stop` 方法关闭 Agent。

Embedded Agent API 包含在 `flume-ng-embedded-agent` 组件中，可以在你的应用程序的 *pom.xml* 文件的依赖部分添加如下内容： <192>

```
<dependency>
  <groupId>org.apache.flume</groupId>
```

```
        <artifactId>flume-ng-embedded-agent</artifactId>
        <version>1.5.0</version>
    </dependency>
```

log4j Appender

Apache log4j 是一个很流行的日志系统，支持插入自定义的 logger。Flume 提供了两类可以插入到你的应用程序中的 log4j appender：一个可以写数据到一个确切的 Flume Agent；另一个以循环或随机的顺序从若干配置好的 Flume Agent 中选择一个，并将数据写到该 Agent。如果要使用 Flume 的 log4j appender，*flume-ng-log4jappender-1.5.0-jar-with-dependencies.jar* [flume-log4j] 应该放到你的应用程序的类路径中。在创建你的工程的时候不需要这么做，但是在部署的时候需要这么做。

log4j appender 通过 *log4j.properties* 文件进行配置。为了学到更多关于 log4j 的配置，请查阅 log4j 文档 [log4j-doc]。log4 appender 支持 log4j 布局，这可以使用 appender 配置中的 layout 参数进行配置。

表7-6 所有Flume log4j appender的公共配置参数

参数	默认值	描述
UnsafeMode	false	如果设置为 true，如果日志消息不能提交到 Flume Agent 的 Channel 时，log4j appender 将不会抛出任何异常
AvroReflectionEnabled	false	如果设置为 true，appender 尝试将消息的内容解析成 Avro 数据
AvroSchemaUrl	-	存储 Avro Schema 的 URL

UnsafeMode 参数可以设置为 true 用来忽略失败的情况，即当日志消息发送到 Flume Agent 失败的时候。只有当它能接收静默地丢失日志消息时，才可以这么设置。如果不设置或设置为 false，日志消息超时或不能成功提交到 Flume Agent 的 Channel，那么日志方法将抛出异常。

所有的 log4j appender 都可以使用 Avro 序列化进行序列化数据。如果传入的数据是 Avro
193 Generic Record 或 Specific Record 的实例，log4j appender 将使用 Avro 序列化方法进行序列化数据。如果 AvroReflectionEnabled 设置为 true，appender 也将任何数据序列化为 Avro。值得注意的是，只有当数据必须序列化为 Avro 时，该参数才这么使用。如果 AvroSchemaURL 已经设置，那么 appender 将在 Flume 事件中设置键 flume.avro.schema.url，其值为 AvroSchemaURL 参数的值。如果没有进行设置，那么整个 JSON 化模式将使用键 flume.avro.schema.literal 写到 header。

所有的 log4j appender 插入额外的 header 到 Flume 事件的 header，这些 header 给出了关于事件的额外信息。表 7-7 展示了这些额外的 header。

表7-7 Flume log4j appender添加的header

header	描述
flume.client.log4j.logger.name	插入到事件的 logger 实例的名字
flume.client.log4j.log.level	该消息记录的级别
flume.client.log4j.message.encoding	消息的编码（目前总是 UTF-8）
flume.client.log4j.timestamp	消息附加时的时间戳

log4j appender 还需要表 7-8 列出的两个额外配置参数。

表7-8 Flume log4j appender配置

参数	默认值	描述
<appender-name>	-	必须为 org.apache.flume.clients.log4jappender.Log4jAppender
Hostname	-	运行 Flume Agent 的主机名
Port	-	Flume Agent 的 Avro Source 监听的端口

appender 的名字是指定 log4j appender 完全限定类名的键。在这种情况下，必须设置为 org.apache.flume.clients.log4jappender.Log4jAppender。Flume log4j appender 被称为 flumeAppender，可以在你的 log4j.properties 文件中使用下列行进行配置。

```
log4j.rootLogger = INFO, flumeAppender
log4j.appender.flumeAppender = \
org.apache.flume.clients.log4jappender.Log4jAppender
log4j.appender.flumeAppender.Hostname = usingflume-srv-1.domain.com
log4j.appender.flumeAppender.Port = 3343
log4j.appender.flumeAppender.UnsafeMode = true
```

该配置在 INFO 级别或更高级别，将消息写入 flumeAppender，并且写数据到主机和端口为 usingflume-srv-1.domain.com:3343 的 Flume Agent。

194

Load-Balancing log4j Appender

和 RPC 客户端类似，log4j appender 也可以配置使用顺序或随机的策略，在多个 Flume Agent 之间实现负载均衡。使用表 7-9 中的另外的参数进行配置。

表7-9 Load-balancing log4j appender配置

参数	默认值	描述
<appender--name>		必须为org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
Hosts	-	指定为hostname:port格式的主机列表，为Flume Avro Source监听的地址
Selector	ROUND_ROBIN	可以为ROUND_ROBIN，RANDOM或实现了LoadBalancingRpcClient.HostSelector的类的完全限定类名
MaxBackoff	-	连接到已经失败的Flume Agent的最大后退时间（以毫秒为单位）

load-balancing log4j appender可以在几个主机之间实现负载均衡，与load-balancing RPC客户端类似。主机的列表必须指定为逗号分隔的列表，以hostname:port的格式作为键hosts的值。

就像load-balancing RPC客户端，log4j appender可以通过设置Selector参数的值为ROUND_ROBIN或RANDOM，以使用循环或随机的顺序选择主机。如果要使用自定义的策略，那么需要指定实现了LoadBalancingRpcClient$HostSelector的类的完全限定类名。与load-balancing RPC客户端还有相同的地方，log4j appender可以将失败的主机加入到黑名单，使用MaxBackoff参数使增加的时间直到回退时间的最大值。

```
log4j.rootLogger = INFO, flumeAppender
log4j.appender.flumeAppender =
org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
log4j.appender.flumeAppender.Hosts = usingflume-srv-1.domain.com:5545,
usingflume-srv-2.domain.com:3133,usingflume-srv-3.domain.com:4454
log4j.appender.flumeAppender.UnsafeMode = true
log4j.appender.flumeAppender.Selector = RANDOM
log4j.appender.flumeAppender.MaxBackoff = 120000
```

该配置使用LoadBalancingLog4jAppender以随机的顺序连接到三个不同的主机，并且把失败的Flume Agent拉黑120秒的时间。

195

总结

在这一章中，我们介绍了Flume提供的API，以从应用程序获取数据发送到可靠的Flume Agent。我们还介绍了Flume自带的log4j appender，实际上允许用户以不编写任何代码，只是使用一些简单的配置将数据发送给Flume。

在接下来的最后一章，我们将介绍如何规划和部署 Flume，以及一旦部署后该如何监控 Flume 集群。

参考文献

- [truststore] Oracle trust store documentation, *http://bit.ly/1oprwQn*
- [truststore-type] `KeyStore` types, *http://bit.ly/1wxll6y*
- [thrift-doc] Apache Thrift tutorial, *http://thrift.apache.org/tutorial/*
- [flume-thrift] Flume Thrift specification, *http://bit.ly/1oprwjc*
- [hostInfo] `HostInfo` class, *http://bit.ly/1opryrq*
- [zlib-manual] zlib manual, *http://www.zlib.net/manual.html*
- [log4j-doc] log4j manual, *http://logging.apache.org/log4j/1.2/manual.html*
- [flume-log4j] Flume log4j appender on Maven Central, *http://bit.ly/1oprCav*

第 8 章 <197>

规划、部署和监控Flume

到目前为止，我们已经讨论了 Flume 的架构和各种组件及其配置。在这一章，我们将讨论如何规划 Flume 部署，以及如何部署和监控 Flume Agent。我们还将讨论 Flume 项目本身之外的各种可用工具，这些工具会使 Flume 的部署和监控变得更加简易。

规划一个 Flume 部署

规划一个 Flume 的部署可能会非常棘手。在本节中，我们将讨论满足你需求的 Flume 部署的规划中所涉及的步骤。

修复时间

大多数生产部署对出现故障的系统定义了平均修复时间（MTTR），这通常是系统在线恢复将花多长时间的很好评估。在本节中，我们将假定服务器托管的各种服务的 MTTR 是可用的，在大多数情况下恢复所需的时间不会超过该时间。简而言之，在大多数情况下我们认为这是服务器从失败中恢复过来所花费的时间。该值因部署不同会有所不同，如果最大修复时间（失败系统恢复的最长时间）是可用的，应该考虑使用最大修复时间。在这一章，我们假设它是可用的，我们称之为 MaxTTR。

现在用户已经知道（或计算）关于每台机器可以出现故障的最长时间，我们也认为这一章的目的是，存储或索引系统（系统作为一个整体，而不是一台机器）在任何计划内或 <198>
计划外的故障时间都有一个已知的上限，我们称为存储修复的最大时间（MTSR）。

综上所述，我们有两个类型的上限：MaxTTR，单个服务器的最大故障时间；MTSR，存储或索引集群的最大故障时间。为了简单起见，我们也假定这些值以秒为单位来衡量。

计算的有效性

如果客户端应用程序不能写到第一层所有 Agent，且第一层 Agent 不能写到第二层的所有 Agent，这些值不再有任何意义。因为当一个 Agent 无法写入另一个时，合并能力是没有意义的（如果第二层 Agent 出现故障，从客户端到第一层 Agent 的路径数量将减少）。因此，只有当客户端应用程序和第一层之间，层与层之间部署负载均衡或故障转移的策略，在本章中所有的数字才是有效的。

在下面几节中，我们将讨论不同的层之间以及最后一层与 HDFS 之间，Agent 数量的比率。这些数字是基于实验的，不一定对每一个场景都科学有效。当服务器很强大时，以及有几个内核和大量的 RAM，以及更加新的操作系统优化等，就可以大幅度提高整体性能。

这里介绍的比率基于很多因素可能很不一样，包括但不限于硬件、网络配置、使用的操作系统和部署的优化。这里的想法是目前所使用的技术，而不是提供绝对的方法来做这个计划。根据前面提到的所有因素，每个部署真正的比率会有所不同。使用这些数字作为指导，但不要盲目依赖它们。使用它们作为起点，然后，基于反映即将部署的实际硬件和网络的实验，提出合适的数值。

我的 Flume Channel 需要多少容量？

这是团队部署 Flume 最常见的问题之一。Flume Channel 的容量在生产数据的应用程序和存储系统之间提供了一个缓冲区。根据我们之前的定义，无法访问的存储系统的最大时间等于 MaxSTR。这意味着在一段相当于 MaxSTR（加上缓冲时间）的时间里，当 Flume 缓冲数据时，应用程序必须仍然能生产和写入数据到 Flume。

199

计算总缓冲能力很简单。假设所有应用服务器事件的综合生产能力在高峰小时是每秒 p_{max} 个事件。那么在这个时期产生事件的总数可以表示为：

$$T_{max}=p_{max} \times MaxTSR$$

这是应该缓冲在整个 Flume 部署中的事件总数。但这种情况并没有考虑这样一个事实，托管 Flume Agent 的机器也可能失败，这意味着它们提供的任何缓冲将会不可用。考虑到这一点，为 Flume 集群添加一个额外的缓冲是很重要的。T_{max} 额外的 25% 是添加的一个合理的缓冲。所以，在整个 Flume 部署中优异的整体 Channel 容量将表示为：

$$T=1.25 \times T_{max}$$

多少层？

现在我们知道 Flume 部署需要的整体容量，我们还需要了解如何将 Flume 部署分为多层。

当设计分层 Flume 的部署时，就要开始计算从层到 HDFS 通信的能力。

直接写入到 HDFS 集群的机器数量应该是非常有限的，因为 HDFS 集群也必须支持读取和处理数据的应用程序，像 MapReduce、Impala、Spark 等。通常限制写数据到 Flume 的 Flume Agent 的数量为几十个 Agent 是一个好主意。根据 HDFS 集群使用的服务器，集群的负载和网络情况，你的部署可能在任何时候处理更多的客户端写操作。

HDFS 和 Avro Source 扩展

一般来说，保持写数据到 HDFS 集群的 Agent 的数量处于一个比较小的值是一个好主意。所需的 Sink 数量和 Agent 数量，基于特定的用例而有所不同。

因为 Avro Source 接收从 Flume Agent 发送的数据，这些 Agent 实际是运行在一个 JVM 内的服务器，Avro Source 做了大量额外的工作——缓冲到 Channel，Sink 运行几个线程从 Channel 拉取数据，并且写数据到存储系统等。Avro Source 最终将达到不能进一步扩展的情况。它最终会因为 Avro Source 工作线程从 Channel 拉取数据时间太长而结束，导致前一层上 Avro Sink 的超时。此时，该层需要添加更多运行 Avro Source 的 Agent。

我们的目标是确保 Flume 集群的 Channel 中至少有 T 的容量。要做到这一点，我们应该在应用程序和 HDFS 集群之间添加更多的 Agent。一旦写入到 HDFS 的层已满，它将退回到先前的层或应用程序。因为回退到应用程序是很不友好的，当写入到 HDFS 的 Agent 的总容量大于 T，且 HDFS 层 Agent 的数量已经增加到超过 8 个时，在应用程序和写数据到 HDFS 的层之间最好再添加一个额外的层。

这个想法是开始添加 Agent 到写入 HDFS 的层用来增加整个 Channel 的容量。一旦数量很大，HDFS 集群因打开太多的文件或太多的客户端写操作开始面临难题，这表明需要添加更多的容量。此时，应用程序和 HDFS 层之间的一层变得很有必要——我们称该层为 T_n。

一旦 T_n 中 Agent 的数量和写入到 HDFS 层之间的 Agent 数量比率达到 32:1（记住，这些数据需要对单个部署进行测试），在应用程序和 T_n 之间很有必要增加一个额外的层，这样可以增加更多的容量。相同的逻辑可以应用在应用程序和现有的层之间，来进一步增加更多的容量。

类似的逻辑需要适用于扩展从应用程序接收数据的 Agent 数量。添加 Agent 来增加容量和减少应用服务器到第一层 Agent 的比例，直到低于 32:1。如果第一层中 Flume Agent 的数量很小（数以万计机器的订单），则直接从这些机器写入 HDFS；否则，在第一层和第二层之间添加另外一个层，使得比率小于 32:1。现在保持不断增加的层中 Agent 的数量，

以致于连续 Agent 之间的比率就可以小于 32:1。由于最后一层和 HDFS 之间的比率，使得额外的层变得很有必要，因为最后一层不能超过 8 个机器。

你怎么知道 Flume 不能再扩展或目的地存储系统或索引变慢？

前面的比率为集群部署提供了一个很好的起点。但是一旦你部署了 Flume，你怎么知道你可以通过 Flume 写更多的数据呢？在大多数情况下，如果你没有持续在日志中看到异常，且你的 Channel 容量增长没有失控（一些增长是可以的，支持它们最终在大小上有所下降），这意味着一切都好，可以发送更多的数据。一旦你开始看到持续的异常或 Channel 大小只是一直增长，那么你就需要行动了。

如果 Flume Sink 清理 Channel 不够快，而且日志中没有超时，那么添加更多的 Sink 将有所帮助，直到达到 JVM 可以支持线程的最大数量，此时 Channel 被清除的速度开始不变（现在日志仍然没有超时信息）。如果 Channel 清除的速度仍不够快，且 Channel 大小不断增加，这意味着所有 Sink 结合起来清除 Channel 的速度不如 Source 写到 Channel 的速度快，表明需要在这层添加更多的 Flume Agent。在这层添加更多的 Agent 可以分布负载，从而减少 Source 写速率和 Sink 清理速率之间的差值。这将确保 Channel 大小保持一致，不再持续增长。

但是还有一个可能性，你的存储或索引系统可能对所有 Flume 的写负载进行了部署，但 Flume 已经结合了集群上你所做的繁重的处理。这种现象会造成发生在 Sink 端的超时，通常在日志中用 `TimeoutExceptions` 表示。如果是这种情况，增加更多的 Sink 并没什么帮助，因为存储系统本身无法跟上负载的增长。

对于 HDFS，如果有一些 MapReduce 或 Impala 作业运行在同一个集群，且集群的 I/O 设备越来越不堪重负，这种情况有可能发生。

对于 HBase，这种情况可能由多个原因导致，例如太多写到相同 Region 的操作，频繁发生扫描或对行锁定。无论哪种方式，这需要通过扩展存储系统进行修整。

对于每一个系统，如何解决这种情况取决于系统本身。你可以在 HDFS 集群中添加更多的数据节点或在 HBase 集群中添加更多的 RegionServer，或分布式地将数据写入多个 Region。

通过跨数据中心链接发送数据

跨数据中心的链接通常比内部数据中心链接要慢得多，也更加不可靠。如果生产数据的应用程序在一个数据中心，而 HDFS 集群在另一个数据中心，在每一个数据中心都有一个 Flume 层总会是一个好主意，由于跨数据中心的链接更有可能失败或延迟高于本地链

接。应用程序为了避免在链接上的高延迟或失败发生，应该写到 Flume，让 Flume 处理延迟或失败。应用程序写入一个层，该层跨广域网写数据到另一层，然后再写入 HDFS。这种工作方式更好，因为当写操作失败时，Flume 通过调整回退机制进行自动调整。这也使得应用程序可以不管用跨数据中心链接的失败或减速继续工作，依靠 Flume 做缓冲，重试和回退以减少在这些链接上的阻塞。

让 Flume 的写操作跨广域网直接写入 HDFS 不是一个好主意，因为失败的可能性更大。当失败发生时，Flume 可能无法关闭文件或确保数据可以可靠地写入到 HDFS 集群。这可能会导致大量的重复和写入 HDFS 的小文件，因为一旦对文件的单个写入操作超时或失败，Flume 将会关闭这个文件。

如果跨广域网的写操作职责交给 Flume，不太可能发生重复和 HDFS 中的小文件失败的情况，因为 RPC sink-source 组合负责可靠的跨广域网传送数据。当然，超时将基于整个广域网延迟级别进行调整，但偶尔的几个失败不会给 Flume Agent 造成很多压力，如图 8-1 所示。

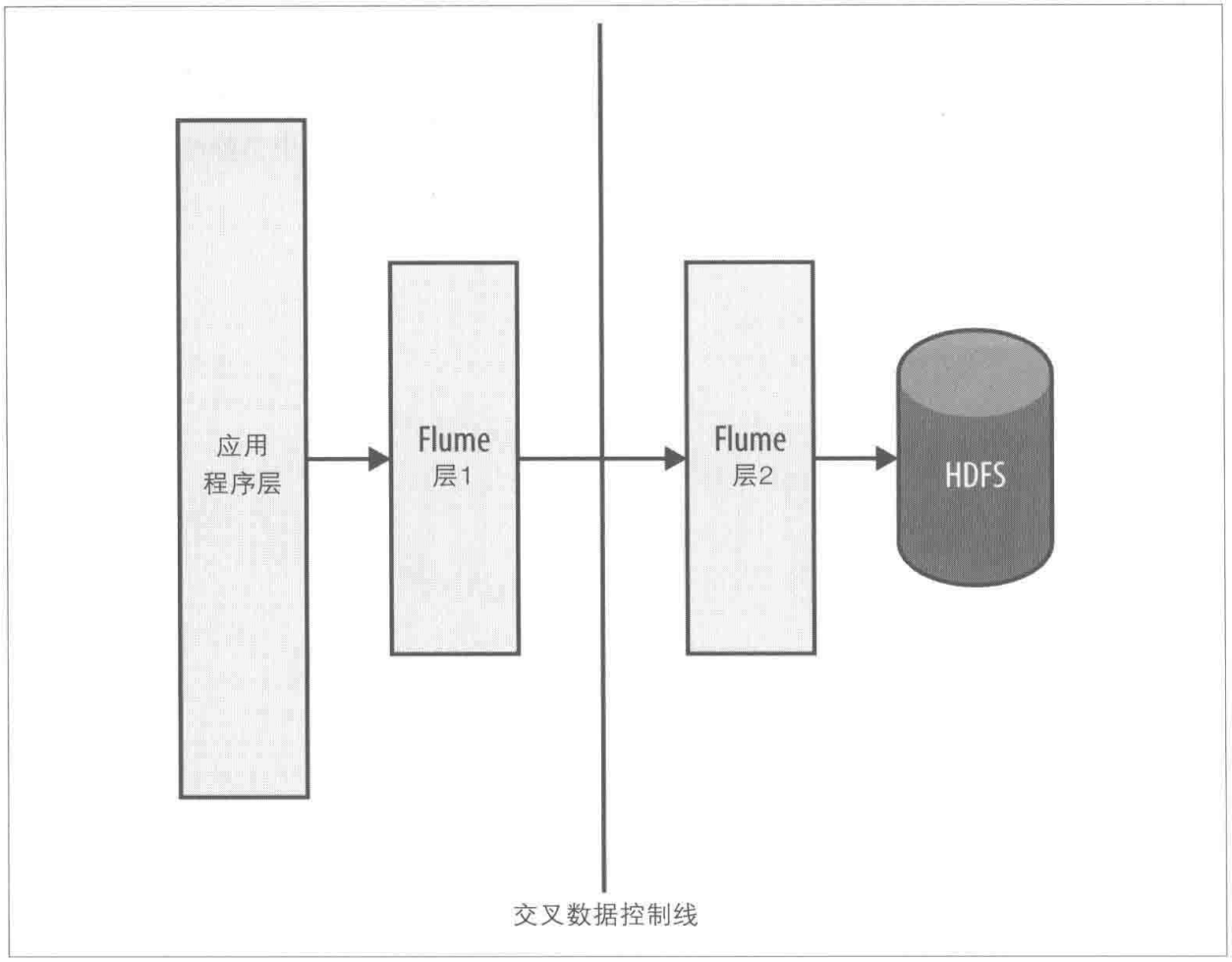


图8-1 使用Flume进行跨数据中心通信

层分片

为了扩展需要缓冲的事件数量，添加层更有意义。但在一层中的每个 Agent 对于下一层的每个 Agent 打开连接，进入第二层的每个 Avro Source 连接的数量仍然与第一层中 Agent 的数量一样多。如此高的绝对数量仍然给这些服务器资源造成一些压力，因为套接字使用文件描述符和 TCP 缓冲区，但是由于前一层 Agent 写入如此多的 Agent，它们的使用可能有所限制。为了避免这种情况，一种好方法可能是层分片。

让我们看一个例子，层 1 有 380 个 Agent，这意味着层 2 必须至少有 12 个 Agent 以保持比率低于 32:1。层 2 的 12 个 Agent 中，每一个 Agent 最终有 380 个传入的开放连接（你也可以将这个替代为 380 个应用程序服务器和 12 个 Flume Agent，作为第一层也是唯一的层）。这给第一层的 Agent 12x 个容错，但这也意味着，12 个 Agent 中的每一个都要
203 为 380 个套接字分配 TCP 缓冲区。通常，12x 容错不是必需的。你可以很容易地得到类似 3-4x 的容错。如果层 1 中的每个 Agent 只发送数据到层 2 的四个 Agent，它们仍有 4x 容错。

因此，你可以从本质上把层 1 和层 2 分割成三个分片，每个分片处理从应用程序发送到 HDFS 的数据总量的一个子集。如果这些层已经分片，那么层 1 就有三个分片，每个都有大约 128 个 Agent，且层 2 也会有相同数量的分片，每片有 4 个 Agent。层 2 的所有分片可以直接写入 HDFS 或另一个层，该层不需要分片是因为层 2 中 Agent 的数量很小。

图 8-2 展示了该技术。

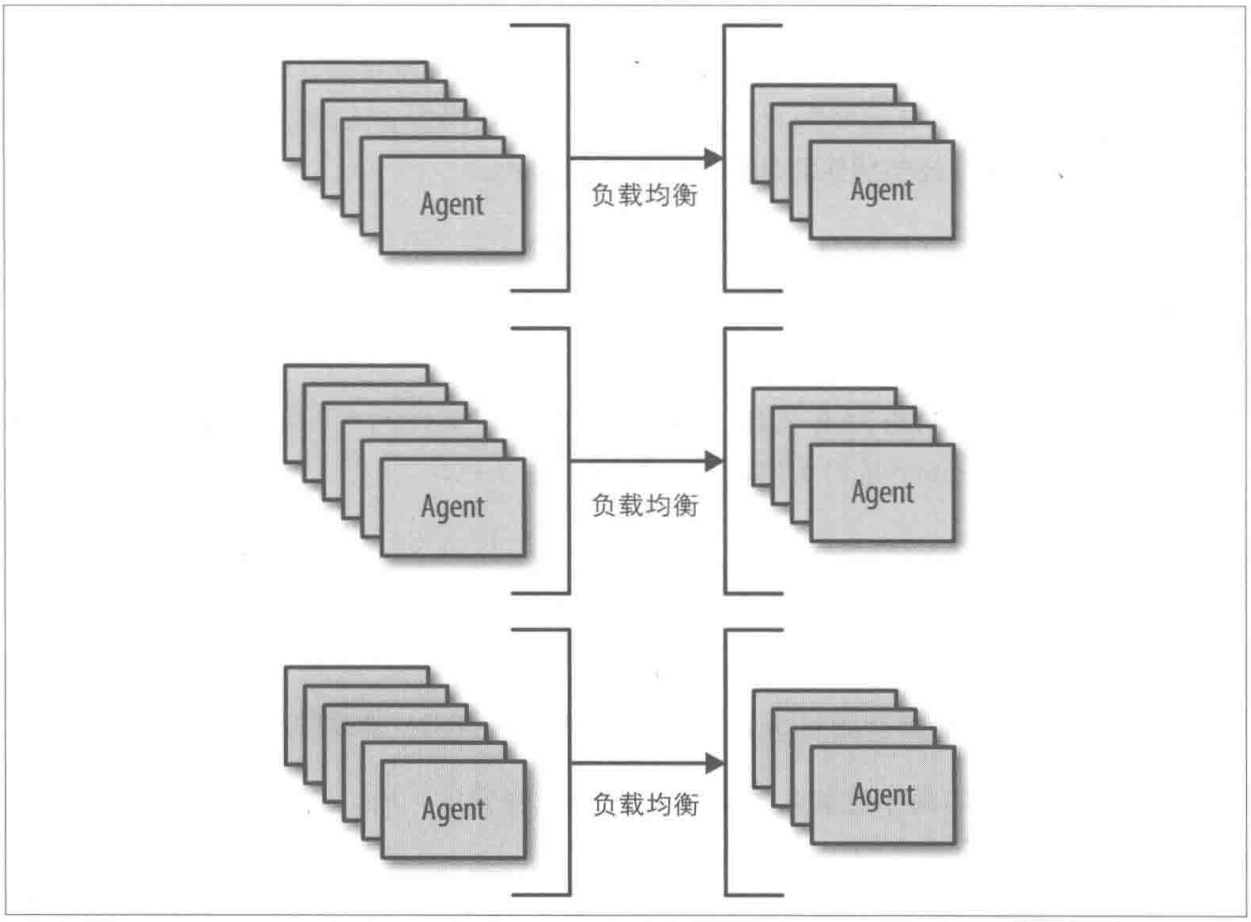


图8-2 层分片以减轻单个服务器的负载

部署 Flume

现在你已经设计好一个 Flume 部署，你需要在这些机器上部署软件和配置它们来运行。Flume 没有集中式部署或管理服务，但在市场上有可用的系统，为你做这些工作。一个例子是 Cloudera Manager[cm]。Cloudera Manager 可以将 Cloudera 的 Flume 部署到不同的服务器，除了部署插件自定义代码也可以部署配置文件。

另一个选择是 Apache Bigtop[Bigtop]；它提供了各种操作系统本地格式的包，例如 Red Hat（和派生系统）的 RPM、Debian 的 deb 等。这些包通常是基于当前 Apache 发布的 Flume。它们可以用于简易部署 Flume 到各种机器，甚至可以使用本地包管理系统安装和移除 Flume。 204

部署自定义代码

自定义代码经常需要部署在 Flume 上。自定义代码可能是为拦截器、序列化器、反序列

化器、HTTP Source处理器等编写的。Flume提供了一种简单部署自定义代码的方式：*plugins.d*框架。*plugins.d*目录是自动添加到Flume环境变量的，因此不需要明确添加到FLUME_CLASSPATH环境变量中。为了添加自定义代码到*plugins.d*框架，你必须理解Flume从*plugins.d*目录将插件添加到环境变量的方式。

图8-3展示了*plugins.d*目录的目录结构。对于每个自定义组件，你会在*plugins.d*目录中创建一个新的子目录（这个名字真的无关紧要）。在每个子目录中，Flume会有三个目录：

lib

该目录包含了实际的JAR文件，JAR文件包含了插件的类和可以进入配置文件的Builder，通过Flume进行实例化。

libext

该目录包含了插件依赖的内部依赖。

native

该目录包含了任何需要通过Java Native Interface（JNI）加载的本地库。

这些目录以不固定的顺序被加入到Flume的环境变量中，所以在任意目录中被添加的JAR文件应该是没有顺序的。

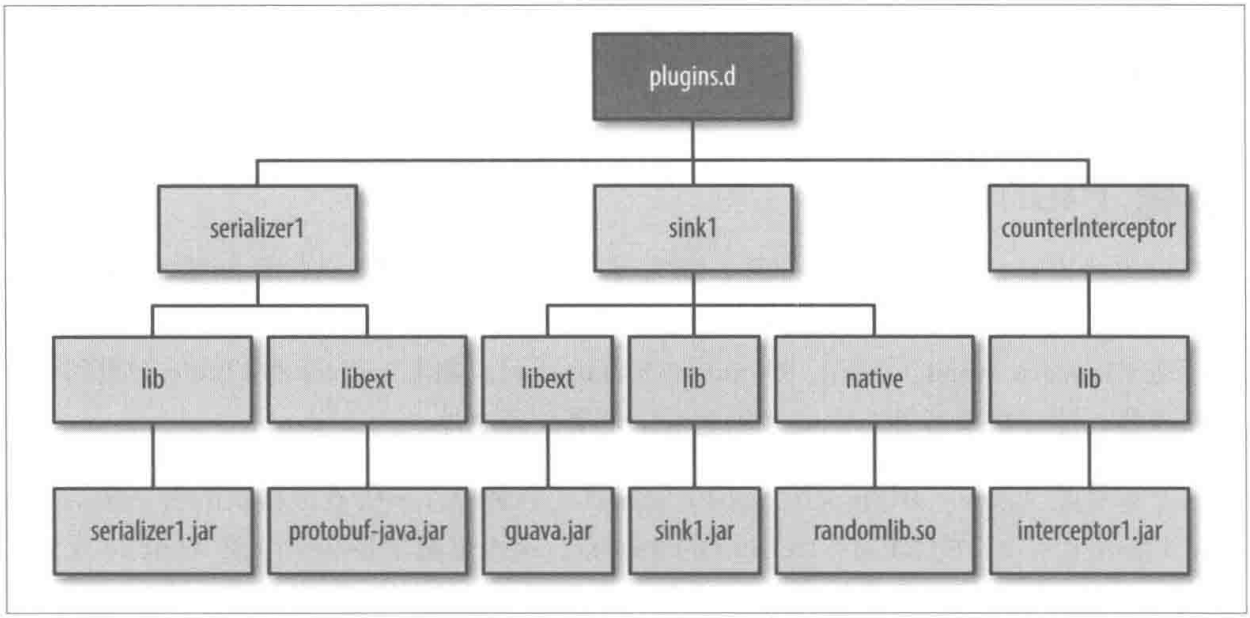


图8-3 *plugins.d*目录结构

监控 Flume

Flume 有一个度量框架，可以通过 Java Management Extensions（JMX）或 HTTP 或 Ganglia 服务器展示出度量。对于每一个组件，有多个展现出来的度量。表 8-1、表 8-2 和表 8-3 分别展示了 Channel、Source 和 Sink 展现的不同度量。

不管什么样的情况下，所有度量都通过 JMX 展现。由于 JMX 可以用来启动或停止 Java 应用程序，所以不建议允许通过 JMX 访问远程计算机。默认情况下该功能禁用，出于安全原因最好不要启用这个功能。

为了给 HTTP 报告度量，当启动 Agent 时传递 `-Dflume.monitoring.type=http` 参数给 Agent：

```
bin/flume-ng agent -f flume.conf -n agent -c conf -Dflume.monitoring.type=http \
-Dflume.monitoring.port=5653
```

这将使得 Flume 在 5653 端口上启动一个 HTTP 服务器。访问 */metrics* 页面，将返回如下 JSON 格式的度量（用来访问指标的 URL 是 *http://usingflume.oreilly.com:5653/metrics* 的形式）：

```
{
"type1.component1" : {"metric1" : "value1", "metric2" : "value2"},
"type2.component2" : {"metric3" : "value3", "metric4" : "value4"}
}
```

对于前面的 Source，类型是 `SOURCE`（Channel 的类型是 `CHANNEL`，Sink 的类型是 `SINK` 等）。下面是一个例子。

206

```
{
"CHANNEL.ch1":{"EventPutSuccessCount":"6645",
               "Type":"CHANNEL",
               "StopTime":"0",
               "EventPutAttemptCount":"6887",
               "ChannelSize":"434",
               "StartTime":"1455782222341,
               "EventTakeSuccessCount":"45200",
               "ChannelCapacity":"600000",
               "EventTakeAttemptCount":"45100"
               }
}
```

Flume 也可以发送度量信息给 Ganglia，用来监控 Flume。在任何时候只能启用一个 Ganglia 或 HTTP 监控。表 8-1、表 8-2 和表 8-3 展示的相同度量通过 HTTP 和 Ganglia 展现。

Flume 周期性地向 Ganglia 报告度量——默认情况是一分钟一次，但是该值可以进行配置用来更频繁地报告度量。为了启用 Ganglia 报告，用下面的命令行启动 Flume（当然，在你的部署中用对应的值替换这些参数）：

```
bin/flume-ng agent -f flume.conf -n agent -c conf \
-Dflume.monitoring.type=ganglia \
-Dflume.monitoring.pollFrequency=45 -Dflume.monitoring.isGanglia3=true
```

为了向 Ganglia 报告度量，当启动 Agent 时给 Agent 传递 `-Dflume.monitoring.type=ganglia` 参数。默认情况下，Flume 以 Ganglia 3.1 的格式报告指标。为了用 Ganglia 3 的格式，启动时在命令行中给 Flume Agent 传递 `-Dflume.monitoring.isGanglia3=true` 参数。为了改变报告间隔，使用 `-Dflume.monitoring.pollFrequency` 作为命令行的参数。传递的值是 Flume 两个连续报告的时间间隔（以秒为单位）。

表8-1 Channel度量

度量	描述
ChannelSize	目前 Channel 中事件的总数量
EventPutAttemptCount	Source 尝试写入 Channel 的事件总数量
EventPutSuccessCount	成功写入 Channel 且提交的事件总数量
EventTakeAttemptCount	Sink 尝试从 Channel 读取的事件总数量。这不意味着每次事件都被返回，因为 Sink 拉取的时候 Channel 可能没有任何数据
EventTakeSuccessCount	Sink 成功读取事件的总数量
StartTime	Channel 启动时自 Epoch 以来的毫秒值时间
StopTime	Channel 停止时自 Epoch 以来的毫秒值时间
ChannelCapacity	Channel 的容量
ChannelFillPercentage	Channel 满时的百分比
Type	对于 Channel，该指标总是返回 CHANNEL

207

表8-2 Source度量

度量	描述
EventReceivedCount	目前为止 Source 已经接收到的事件总数量
EventAcceptedCount	成功写出到 Channel 的事件总数量，且 Source 返回 success 给创建事件的 Sink 或 RPC 客户端或系统

续表

度量	描述
AppendReceivedCount	每批只有一个事件的事件总数量（与 RPC 调用中的一个 append 调用相等）
AppendAcceptedCount	单独传入的事件写入到 Channel 且成功返回的事件总数量
AppendBatchReceivedCount	接收到事件的批次的总数量
AppendBatchAcceptedCount	成功提交到 Channel 的批次的总数量。
StartTime	Source 启动时自 Epoch 以来的毫秒值时间
StopTime	Source 停止时自 Epoch 以来的毫秒值时间
OpenConnectionCount	目前与客户端或 Sink 保持连接的总数量（目前只有 Avro Source 展现该度量）
Type	对于 Source，该指标总是返回 SOURCE

表8-3 Sink度量

度量	描述
ConnectionCreatedCount	下一阶段或存储系统创建的连接数量（例如在 HDFS 创建一个新文件）
ConnectionClosedCount	下一阶段或存储系统关闭的连接数量（例如在 HDFS 关闭一个文件）
ConnectionFailedCount	下一阶段或存储系统由于错误关闭的连接数量（例如 HDFS 上一个新创建的文件因为超时而关闭）
BatchEmptyCount	空的批量的数量，如果数量很大表示 Source 写数据比 Sink 清理数据速度慢得多
BatchUnderflowCount	比 Sink 配置使用的最大批量尺寸更小的批量的数量，如果该值很高也表示 Sink 比 Source 更快
BatchCompleteCount	与最大批量尺寸相等的批量的数量
EventDrainAttemptCount	Sink 尝试写出到存储的事件总数量
EventDrainSuccessCount	Sink 成功写出到存储的事件总数量
StartTime	Sink 启动时自 Epoch 以来的毫秒值时间
StopTime	Sink 停止时自 Epoch 以来的毫秒值时间
Type	对于 Sink，该度量总是返回 SINK

208

这些度量可以用来关注 Flume Agent 的健康状态。例如像 Cloudera Manager 的一些系统可以接受这些 Flume 提供的原始度量，并且提供更多有意义的度量，如多长时间把 Channel 填满。这种系统还提供这些度量按时间分时段（time-bucketed）的图表。当度量遇到至关重要的情况，这些系统可以提醒运维工程师，指明 HDFS 集群或 Flume Agent 规划内容量的故障时间。

从自定义组件报告度量

例如 Source、Sink 或 Channel 的自定义组件，可以使用 `SourceCounter`、`SinkCounter` 和 `ChannelCounter` 类直接向 Flume 框架展现度量，这些类提供了更新 Source、Sink 和 Channel 以前度量的方法。

其他组件，例如拦截器、序列化器等，也可以展现度量，但是没有便捷的类可以直接这样实现，因为每个组件可能展现看起来不一样的度量。这些组件可以使用 `MonitoredCounterGroup` 类直接向 JMX 报告度量。自定义组件应该创建一个继承自该类的 counter 类，对于不同的度量有一个公共的获取方法。为了增加、更新或增量这些度量值，counter 类可以提供一些方法，供自定义类实现来使用。

总结

在本章中，我们讨论了如何规划、部署和监控 Flume。如果在实际部署 Flume 之前考虑到集群故障和期望的缓冲问题，那么就可以用可扩展、灵活性的方式规划和部署 Flume。你也要确保有足够大的缓冲，以确保 HDFS 的故障不会影响写数据到 Flume 的应用程序。

可以使用 Flume 内置的简单技术或允许单个位置部署和配置 Flume 的第三方系统，部署和监控 Flume。

参考文献

- [cm] Cloudera Manager, *http://bit.ly/1opzYiz*
- [bigtop] Apache Bigtop, *http://bigtop.apache.org*

索引†

Symbols

% (percent) character, 97
64-bit Rabin Fingerprint, 57

A

AbstractChannelSelector class, 155
AbstractElasticSearchIndexRequestBuilderFactory class, 131
AbstractPollableSource class, 71, 73
ACID compliance, 5
active name node servers, 2
ActiveMQ, 64
agents (Flume), 9, 10-20
  calculating optimum number of, 200
  communication between, 17
  components of, 9
  configuring, 13-16
  data loss and, 25
  embedded, 185-192
  failure of, 25
  flows, 17-20
  implementation of, 10-13
  Kerberos credentials for, 106
  running, 29-31
aliases for sources, 34
Amazon S3, 139
Apache Bigtop, 209
  deploying Flume with, 204
Apache Flume JMS Source, 78
Apache Hadoop, 1-6
  as data processing framework, 7
  file system for, 1-5
  HBase, 5
  HDFS, 1-5
  HDFS Sink support for, 96
  writing to, 108
Apache HBase, 5
  client API, 5
  client-side security configuration, 139
  HTTP API, 10
  master server, 6
  operations, 5
  regions in, 6
  Shell, 6
  shell commands for, 5
  Thrift API, 6
  timeouts in, 201
  versions, 121
Apache Hive, 4, 99
Apache Kafka, 7
Apache MINA, 61
Apache Pig, 5, 7
  Avro container format, 112
Apache Solr, 139
  on HDFS, 139
  unique key, 139
Apache Solr Search, 125
Apache Spark, 5
  removing duplicates with, 28
Apache Thrift, 139
  generating code in other languages with, 184
  IDL, 40
  Project, 78
  RPC client, 184
  sending data over, 169
  Source configuration, 40
  SSL and, 41

†： 索引所列页码为本书英文版页码，请参照用“▭”表示的原书页码。

Apache Thrift Project, 78
Apache Thrift source, 40-42
  failure handling, 42
append method (RPC client), 170
  for host selector, 183
appendBatch method (RPC client), 170
  batch-size parameter and, 171
  for host selector, 183
applications
  aggregating data from, 8
  embedding agents in, 185-192
Async HBase Sink, 114-121
  configuration, 114
  serializer interface, 117
  serializer, location of, 120
AsyncBase, 139
AsyncHBase API, 114
  atomic increment request, 139
  AtomicIncrementRequest objects, 117
  put request, 139
  PutRequest objects, 117
Avro container file format, 108
  compression support in, 3
  error catching in, 3
  log4j appenders and, 192
  MapReduce and, 112
  sending data over, 169
Avro container file specification, 139
Avro IDL documentation, 78
Avro Schema Fingerprints, 78
Avro Sink, 17, 121-124
  agent to agent communication and, 17
  compression and, 38
  configuration, 122
  configured with SSL and compression, 39
  embedded agents and, 188
  RPC client and, 178
  SSL and, 38
Avro Source, 17, 36-39
  configuration, 36
  configured with SSL and compression, 39
  failure handling, 42
  replicating data with, 21
  scaling, 199
  source scaling for, 199
AvroEventDeserializer (Flume), 57
AvroReflectionEnabled parameter (log4j appender), 193
AvroSchemaURL parameter (log4j appender), 193

B

backoff period, load-balancing RPC clients, 181
backupCheckpointDir parameter (File Channel), 88
batch-size parameter (RPC clients), 171
batches, 26-28
  determining size of, 28
  Rolling Sink and, 134
batchSize parameter
  Async HBase Sink, 115
  Elastic Search Sink, 130
  Exec Source, 63
  HBase Sink, 115
  HDFS Sink, 106
  Morphline Solr Sink, 126
  Null Sink, 133
  Spooling Directory Source, 51
batchTimeout parameter (Exec Source), 63
begin method (Transaction), 80
Bigtop (Apache), 209
  deploying Flume with, 204
binary columnar formats, 109
binary file formats, 108
  HDFS and, 3
Binary Large Object (BLOB) files, 78
  reading, 58
blacklisting failed sinks, 160, 162, 163
blacklisting unresponsive hosts, 181
BLOB (Binary Large Object) files, 78
  reading, 58
blocks (of data), 2
buckets, 97-100
  defined, 96
  time-based, 99
buffers, 185
Builder interface
  EventDeserializer interface, 53
  Interceptor, 147
byteCapacity parameter (Memory Channel), 84
byteCapacityBufferPercentage parameter (Memory Channel), 84
bz2 compression format, 4

C

callTimeout parameter (HDFS Sink), 106
capacity parameter

File Channel, 87
Memory Channel, 84
capacity planning, 198
channel processor, 11, 80
channel selectors, 150-157
configuration, 150
configuring for sources, 35
congestion behavior of, 151
custom, 157
multiplexing, 152-155
replicating, 151
ChannelCounter class, 208
ChannelException, 12
ChannelException error, 25, 42, 150
channels, 11, 22, 79-92
capacity planning, 198
clearing with multiple sinks, 200
clearing with sink groups, 158
component-type parameter for, 14
custom sources and, 70
File, 85-92
for embedded agents, 188
Memory, 82-85
persistent, 85-92
total buffer capacity, calculating, 199
transaction workflow and, 80-82
writing to multiple, 12
channels parameter (multiplex channel selectors), 153
ChannelSelector interface, 155
character sets, naming, 53
checkpointDir parameter, 88
checkpointInterval parameter (File Channel), 88
checkpoints (File Channel), 89
client configuration file
Async HBase Sink, 115
HBase Sink, 115
client writes, 2
close method (Transaction), 81
Cloudera Impala, 4, 7
bucketing and, 99
deploying Flume with, 203
Cloudera Manager, 208, 209
clusterName parameter (Elastic Search Sink), 130
clusters of agents, 18
column families (HBase), 5
command parameter (Exec Source), 63
commands (Morphline Solr Sink), 125
commit method (Trasaction), 81
communication sink-to-source, 36-43
component-type parameter, 14
compression formats, 4
Avro native, 113
for Avro Sink, 123
for HDFS Sink, 105
in RPC client, 178
Configurable interface, 15, 135, 138, 156
configuration file (Flume), 29
ConfigurationProvider
for sinks, 94
for sources, 33
configure method
custom source handlers, 76
EmbeddedAgent class, 187
HTTP Source handlers, 45
sinks, 94
connect-timeout parameter (RPC client), 178
connectionFactory parameter (JMS Source), 65
Context instances, 15, 45
deserializers and, 53
for custom interceptors, 147
converter (JMS Source), 66
configuring, 68
corruption of files, 3
createHostIterator method, 183
cron jobs, 99
cross-data center links, 201
custom code
channel selectors, 155-157
deployment of, 204
deserializing custom formats, 53-58
event-driven sources, 74
for channel selectors, 155-157
for handling HTTP Source, 44-49
for interceptors, 147-150
for sinks, 135-139
host selector, 183
pollable source, 71
reporting metrics from, 208
custom sources, 69-77
event-driven sources, 33
pollable sources, 71-73
Customizing Java Security, 78

D

data

backups, 90
buffering, 63
duplication, 9, 28
types allowed by HDFS Sink, 105
data blocks, 2
replicating, 20
data centers, 19
data compression
Avro Source and, 38
RPC clients and, 38
data flow in Flume agents, 9
data formats
controlling for HDFS Sinks, 108-114
controlling with serializers, 108-114
customizing, 131-133
in HDFS, 3
ingest vs. output, 113
data loss
Flume and, 22
Syslog sources and, 62
with Exec Source, 64
data node servers, 2
data processing in HDFS, 4
dataDirs parameter (File Channel), 88
Deflate/bz2 compression codec
Avro support for, 3
deletePolicy parameter (Spooling Directory Source), 52
delimiters, setting, 60
deployment (Flume)
capacity planning, 198
cross-data center links and, 201
monitoring, 205-208
of custom code, 204
of Log4j appenders, 192
planning, 197-203
scaling, 199
services for, 203
tiers, calculating, 199
tiers, sharding, 202
time to repair, defining, 197
deserializer parameter, 53
deserializers, 53-58
The design and implementation of a log-structured file system (Rosenblum, Ousterhout), 92
disk space, checking, 89
doConfigure method (AbstractPollableSource), 73
durability guarantees, 9
dynamic routing, 21

E

Elastic Search Client API, 133
configuration, 129
Elastic Search Sink, 129-133
customizing data formats, 131-133
Embedded Agent API, 187, 192
embedded agents, 185-192
Avro Sink and, 188
configuring, 188-192
shutting down, 187
EmbeddedAgent class, 187
escape sequences, 97
for timestamps, 98
Event interface, 167
event-driven sources, 33
EventBuilder API, 167
EventDeliveryException, 187
RPC client, 170
EventDeserializer implementation, 53
EventDrivenSource interface, 73-77
events (Flume)
building, 167
converting from JMS Source, 67-69
converting into Morphline records, 125
filtering with regex interceptors, 144
sending from applications, 170
translating into Puts, 117-121
writing to embedded agents, 187
EventSerializer class (HDFS Sink), 109
exception handling in HTTP Sources, 45
excludeEvents parameter (regex interceptors), 145
Exec Source, 62-77
batch sizes, controlling, 27
configuration, 62
Spooling Directory Source vs., 50
expiration dates on indexed documents, 130

F

Facebook Scribe, 7
failed sinks, blacklisting, 160, 162, 163
failover
RPC client, 184
RPC client configuration, 184
sink processor, 162-164
failure handling, 25

Avro sources, 42
binary files and, 108
channel processors and, 150
embedded agents and, 186
sending events from applications, 170
Thrift sources, 42
with load-balancing RPC clients, 180
writing to required channels, 12
fan-in topology, 19
file
corruption, indicating, 3
formats of, 108
limiting number of open, 106
maximum size of, 88
size constraints on, 10
File Channel, 23, 85-92
Avro Sink and, 122
batching and, 27
best practices for, 90
designing, 90-92
embedded agents and, 186
implementing, 90-92
performance of, 89
records, reconstructing, 91
filenames
generating, 97, 103
including, 52
filePrefix parameter (HDFS Sink), 97, 103
fileSuffix parameter
HDFS Sink, 103
Spooling Directory Source, 52
fingerprinting headers, 57
flows, 17-20
of events to channels, 152
reliable vs. fast, 152
Flume, 8-31
agent failure, 25
batches, 26-28
channels, 22
configuration, 15
data duplication, 28
data durability guarantee, 22
data loss and, 22
data replication, 20
dynamic routing between, 21
metrics frameworks, 205-208
startup behavior of, 91
transactions, 23
usage, 10
Flume Client SDK, 168
dependencies for, 168
RPC clients, building, 169
Flume event queue (File Channel), 90
Flume log4j appender on Maven Central, 195
Flume Thrift specification, 184, 195
flume-env.sh, 29
flume-ng-hbase sink package, 120
FlumeException, 187
.flumespool directory, 52
FLUME_CLASSPATH environment variable, 29
FQCN (Fully Qualified Class Name)
for Elastic Search Sink, 130
for Morphline Solr Sink, 126
of Avro Sink, 123
of HBase, 115
of HDFS, 102
of log4j appenders, 193
fsync system call, 27, 31
full replay (File Channel), 91

G

Ganglia
exposing metrics via, 205
sending metrics to, 206
getBatchSize method (RPC client), 170
getChannelProcessor method (AbstractSource), 70
getEvents method (HTTPSourceHandler), 45
getInstance method (RpcClientFactory), 171
getOptionalChannels method (channel selectors), 155
getSubProperties method (Context instances), 15
getTransaction method (channel), 80
Google File System (gfs), 2, 6

H

Hadoop (Apache), 1-6
as data processing framework, 7
file system for, 1-5
HBase, 5
HDFS, 1-5
HDFS Sink support for, 96
writing to, 108
Hadoop Commands, 6
Hadoop impersonation configuration, 139
Hadoop: The Definitive Guide, 3, 6

HADOOP_PREFIX directory, 30
HBase (Apache), 5
  client API, 5
  client-side security configuration, 139
  HTTP API, 10
  master server, 6
  operations, 5
  regions in, 6
  Shell, 6
  shell commands for, 5
  Thrift API, 6
  timeouts in, 201
  versions, 121
HBase Sinks, 114-121
  configuration, 114
  performance of, 95
  Puts, translating events into, 117-121
  serializer, location of, 120
  translating events into Puts, 117-121
HBase: The Definitive Guide, 6, 115, 139
HDFS (Hadoop Distributed File System), 1-5, 7
  calculating tier capacity for, 200
  data formats in, 3
  data processing in, 4
  immutability of data in, 1, 2
  replication factors, 2
  shell commands for, 3
  source scaling for, 199
  SQL interfaces for, 4
  timeouts in, 201
  write constraints on, 8
  writing across a WAN, 201
  writing to, 96-114
HDFS Sink, 96-114
  buckets, 97-100
  configuring, 100-108
  data format, controlling, 108-114
  Hadoop support for, 96
  multiple, configuring, 104
  performance of, 95
  serializers and, 108-114
headers
  generated by interceptors, 147
  generated by log4j appenders, 193
hidden files, writing to, 96
Hive (Apache), 4, 7, 99
  Avro container format, 112
  HDFS Sinks and, 103
host information, replacing, 97
host interceptors, 143
host selector, custom, 183
host-selector parameter (load-balancing RPC client), 181
hostHeader parameter (interceptors), 143
HostInfo class, 183, 195
Hostname interceptor, 97
hosts parameter (RPC client), 171, 184
HostSelector class (LoadBalancingRpcClient), 183, 194
HTTP Source, 43-49
  as event-driven source, 74
  configuring, 43
  custom handlers configuration, 44
  custom handlers for, 44-49
  exception handling in, 45
HTTP, exposing metrics via, 205
HTTPBadRequestException, 45, 47
HTTPHandler interface, 49
HTTPServletRequest, 45
HTTPSourceHandler interface, 44
HTTPSourceXMLHandler, 46

I

IANA Time Zone Data, 139
idle timeouts, triggering, 106
idleTimeout parameter (HDFS Sink), 104
ignorePattern parameter (Spooling Directory Source), 51
immutability of data, 1, 2
increments (HBase), 5
  translating events into, 117-121
indexName parameter (Elastic Search Sink), 130
indexType parameter (Elastic Search Sink), 130
inflight pointers (File Channel), 91
informFailure method (RPC client), 183
informSinkFailed method (sinks), 162
ingest format, 113
initialContextFactory parameter (JMS Source), 65
input format for MapReduce, 4
intercept method (interceptors), 148
Interceptor interface, 147
interceptor parameter (source), 142
interceptors, 11, 141-150
  configuration, 142
  configuring for sources, 35
  custom, 147-150

custom, metrics for, 208
data processing alternatives to, 145
event data and, 113
host, 143
Morphline, 145
regex filtering, 144
static, 143
timestamp, 142
UUID, 146
isActive method (RPC client), 170

J

JAR file, 45
Java, 36
Java command-line arguments, 31
Java KeyStore API, 78
Java KeyStore types, 78
Java Message Service, 33
Java New I/O API, 78
Java String instances, 168
Java trust store, 139
Java Virtual Machines, 36
JAVA_OPTS arguments, 30
JMS ByteMessage converter, 67
JMS ObjectMessage converter, 67
JMS Source, 64-69
batch sizes, controlling, 27
converting into Flume events, 67-69
JMS TextMessage converter, 67
JMSMessageConverter interface, 67
JSON formatted events, 45
JSSE certificate authority files, 123

K

Kafka (Apache), 7
keep-alive parameter
File Channel, 89
Memory Channel, 84
Kerberos, 105
kerberosKeytab parameter
HDFS Sink, 105
kerberosKeytab parameter (HBase Sink), 116
kerberosPrincipal parameter
HDFS Sink, 105
kerberosPrincipal parameter (HBase Sink), 116
keystores, SSL, 38
Kibana Elastic Search UI, 139
Kite SDK, 125

L

lib directory, 29
Line Deserializer, 56, 78
load spikes, handling, 19, 79
load-balancing
between Avro Sinks, 124
log4j appenders, 194
log4j appenders configuration, 194
RPC client, 180-183
RPC client, configuring, 181
sink processor, 159-162
LoadBalancingLog4jAppender, 194
LoadBalancingRpcClient$HostSelector class, 183, 194
loadSolr command (Morphline Solr Sink), 126
log4j
documentation, 31
manual, 195
properties file, 29, 134
log4j appenders, 192-194
configuration, 192
load-balancing, 194
logEveryNEvents parameter (Null Sink), 133
Logger Sink, 134
configuration, 134

M

MapReduce, 4, 7
HDFS Sinks and, 103
input format for, 4
removing duplicates with, 28
splittable files and, 3
max-connections parameter (failover RPC client), 184
maxBackoff parameter (RPC client), 181
maxConnections parameter (Thrift Sink), 125
maxFileSize parameter (File Channel), 88
Maximum Time To Recover (MaxTTR), 197
maximum time to storage repair (MTSR), 197
maxIoWorkers parameter (RPC client), 179
maxOpenFiles parameter (HDFS Sink), 106
MaxTTR (Maximum Time To Recover), 197
mean time to repair (MTTR), 197
validating, 198
Memory Channel, 22, 82-85
batching and, 27
configuration, 83
metrics for Flume, 205-208
from custom components, 208

MINA (Apache), 61
minimumRequiredSpace parameter (File Channel), 89
MIT Kerberos, 139
MonitoredCounterGroup class, 208
monitoring Flume, 205-208
Morphline interceptors, 145
Morphline Solr Sink, 125-129
  configuration, 126
  search indexing with, 125
morphlineFile parameter
  Morphline interceptors, 146
  Morphline Solr Sink, 126
morphlineId parameter
  Morphline interceptors, 146
  Morphline Solr Sink, 126
Morphlines: Kite SDK, 139, 165
mountpoints, multiple (File Channel), 86
MTSR (maximum time to storage repair), 197
MTTR (mean time to repair), 197
  validating, 198
multiplexing channel selector, 22
multiplexing channel selectors, 152-155
Multiport Syslog Source, 58-62

N

name node servers, 2
naming components, 13
Netty Project, 78
Netty servers, 37, 77
Netty-Avro IPC protocol, 36
Netty-based RPC protocol, 121
network latency, 8
NFS-mounted disks, 90
nonblocking I/O (NIO), 37
Null Sink, 133
  configuration, 133
NullPointerException, 176

O

ObjectOutputStream (JMS Source), 67
Oozie jobs, 99
optional parameter (channel selector), 151
Oracle trust store documentation, 195
ORCFile format, 109
OutOfMemoryError, 42
output format, 113

P

parameters, specifying safely, 176
Parquet file format, 109
path parameter (HDFS Sink), 97
paths, defining for sinks, 97
performance
  deployment and, 200
  deployment considerations for, 198
  load-balancing sink processors and, 159-162
  of Async HBase Sink, 114
  of HBase Sink, 114
  of RPC sinks, 27
  of sinks, 95
  sharding tiers and, 202
  Spooling Directory Source, 58
persistent channel, 85-92
Pig (Apache), 5, 7
  Avro container format, 112
plugins.d
  directory, 29
  framework, 34, 204
pollable sources, 71-73
PollableSourceRunner class (Flume), 71
pom.xml file, 49
preprocessed LZO compression format, 4
preserveExisting parameter (interceptors), 142, 143
priorities
  in failover configuration of sinks, 162, 164
  in multiplex channel selectors, 154
process method (sinks), 94
processEvent method (Channel), 70
  custom interceptors and, 148
processEventBatch method (Channel), 70, 71
  custom interceptors and, 148
Properties instance (RPC clients), 169
Protobuf, 108, 110
proxyUser parameter (HDFS Sink), 105
put method (channel processors), 81
  File Channel and, 91
put method (embedded agent), 187
putAll method (embedded agent), 187
PutRequest objects (Async HBase Sink), 117
Puts (HBase), 5
  translating events into, 117-121

R

random sink selection, 160
RCFile format, 109

readChar method, 53
readEvent method (deserializers), 54
records (Morphline Solr Sink), 125
regex filtering interceptors, 144
region server (HBase), 6
regions (HBase), 6
replicating channel selectors, 151
replication factors, 2
reporting metrics from custom code, 208
request-timeout parameter (RPC client), 178
ResettableInputStream instances, 53
restart parameter (Exec Source), 63
rollCount parameter (HDFS Sink), 104
Rolling File Sink, 134
  configuration, 134
rolling timeouts, triggering, 106
rollInterval parameter (HDFS Sink), 104
rollTimerPoolSize parameter (HDFS Sink), 106
round-robin sink selection, 160
row key (HBASE), 5
RPC clients, 169-185
  batching and, 27
  building, 169
  closing, 170
  compression and, 38
  configuration parameters for, 170-176
  creating, 171
  default, 177-180
  failover, 184
  interface, 169
  load-balancing, 180-183
  reconnecting, 176
  sending data over, 169
  SSL and, 38
  Thrift, 184
RPC sinks, 121-125
  agent to agent communication and, 17
  Avro Sink, 121-124
  batching and, 27
  sink groups and, 157
  Thrift Sink, 124
  transaction handling by, 24
RPC sources
  agent to agent communication and, 17
  failure handling, 42
  Thrift, 40-42
  transaction handling by, 24
RpcClientConfigurationConstants class, 176
RpcClientFactory class, 171
  creating default clients with, 177-180

S

SANs (storage area networks), 2
scalability
  of Avro Source, 36
  of Flume agents, 9
  SolrCloud and, 128
Scribe (Facebook), 7
security
  Avro Sink, 123
  Avro Source, 39
  for Avro Sink, 39
  for RPC clients, 38
  HBase Sink and, 116
  in RPC clients, 179
  Kerberos, 105
  keystores, SSL, 38
  SSL, 38
  SSL keystores, 38
  Thrift and, 41
Selector parameter (log4j appenders), 194
sequence files, 4
serializers, 4
  controlling data format with, 108-114
  custom, metrics for, 208
  specifying for HDFS Sinks, 105
  translating events into increments with, 117-121
  translating events into puts with, 117-121
server types (HDFS), 2
setClient method (RPC client), 184
setHosts method (host selector), 183
shell commands
  for HBase, 5
  for HDFS, 3
shell parameter (Exec Source), 63
shell, running Exec Source in, 63
sink groups, 12, 157
  component-type parameter for, 14
  multiple, 158
Sink interface, 135
sink processors, 12, 158-164
  failover, 162-164
  load-balancing, 159-162
sink runner thread, 12, 94, 135
sink-to-source communication, 36-43
  Avro Source, 36-39
  failure handling, 42

Thrift, 40-42
sink.directory parameter (Rolling File Sink), 134
sink.rollInterval parameter (Rolling File Sink), 134
sink.serializer parameter (Rolling File Sink), 134
SinkCounter class, 208
sinks, 11, 93-139
Avro Sink, 121-124
clearing channels with, 200
component-type parameter for, 14
custom, 135-139
Elastic Search Sink, 129-133
embedded agents and, 188
HBase, 114-121
HDFS Sink, 96-114
lifecycle of, 94
Logger Sink, 134
Morphline Solr Sink, 125-129
Null Sink, 133
performance of, 95
Rolling File Sink, 134
RPC sinks, 121-125
specifying for Elastic Search Sink, 130
terminal, 93
Thrift Sink, 124
transaction handling by, 24, 80
Snappy compression codec, 3
Solr (Apache), 139
on HDFS, 139
Search, 125
SolrCloud, 128
unique key, 139
SourceCounter class, 208
sources, 10, 33-77
Avro Source, 36-39
batch sizes, controlling, 27
component-type parameter for, 14
custom, 69-77
Exec Source, 62-77
for embedded agents, 188
HTTP Source, 43-49
JMS Source, 64-69
lifecycle of, 33-35
optional channels in, 152
optional configuration parameters for, 34
required configuration parameters for, 34
sink-to-source communication, 36-43
Spooling Directory Source, 50-58
Thrift RPC Source, 40-42
transaction handling by, 23
Spark (Apache), 5
removing duplicates with, 28
splittable data format (HDFS), 3
spoolDir parameter (Spooling Directory Source), 51
spooldir type (Spooling Directory Source), 51
Spooling Directory Source, 50-58
configuration parameters for, 50
deserializers and, 53-58
Exec Source vs., 50
performance, 58
SQL interfaces, 4
SSL
in Avro Sink, 123
in RPC clients, 179
keystores, 38
Thrift and, 41
standby name node servers, 2
start method (EmbeddedAgent class), 187
static interceptors, 143
stop method (embedded agent), 187
storage area networks (SANs), 2
streaming data, 7-31
with Flume agents, 10-20
subcomponents, configuring, 15
Syslog sources, 58-62
data loss and, 62
Syslog TCP Source, 58-62
Syslog UDP Source, 58-62

T

tail -F code, 50, 64
take method (Transaction), 81
File Channel and, 90
terminal sinks, 93
testing sinks, 133
thread-safety of RPC clients, 176
ThreadedSelectorServer (Thrift), 41
threadsPoolSize parameter (HDFS Sink), 106
Thrift (Apache), 139
documentation, 195
generating code in other languages with, 184
IDL, 40
Project, 78
RPC client, 184

sending data over, 169
Source configuration, 40
SSL and, 41
Thrift Sink, 124
configuration, 124
Thrift source (Apache), 40-42
failure handling, 42
tiers, 18
calculating, 199
capacity planning, 199
sharding, 202
time to repair, 197
time-based buckets, 99
by minute(s), 99
roll intervals and, 104
TimeoutExceptions, 201
timeouts
Async HBase Sink, 116
duplicate events and, 28
Exec Source, 63
HDFS Sink, 104, 106
in RPC clients, 178
interceptors and, 141
transactions and, 24
timestamp interceptor, 142
replacing host information from, 97
timestamps, escaping, 98
timeZone parameter (HDFS Sink), 103
total buffer capacity, calculating, 199
tracker (Spooling Directory Source), 50, 52
trackerDir parameter (Spooling Directory Source), 52
transactionCapacity parameter
File Channel, 87
Memory Channel, 84
transactions, 23
channels and, 80-82
closing, 81
overlapping, 24
trust store, 124
custom, 179
for RPC client, 179
truststore-password parameter (RPC client), 179
truststore-type, 195
type parameter
Async HBase Sink, 115
HBase Sink, 115
types, specifying, 34

U

uniqueKey tag (Solr), 127
universally unique identifier (UUID), 146
configuration, 146
interceptors, 146
UnsafeMode parameter (log4j appender), 192
unsplittable data format (HDFS), 3
useDualCheckpoints parameter (File Channel), 88
useIP parameter, 143
users, impersonating, 105
UUID (universally unique identifier), 146
configuration, 146
interceptors, 146

V

validation of sources, 33
configuration parameters, 45

W

Web HDFS, 10
worker threads, limiting, 179
write-ahead log (WAL), 90
writeFormat parameter (HDFS Sink), 105

X

XML-formatted data, 46

Z

zlib compression, 38
library, 78, 139
manual, 195
zookeeperQuorum parameter for HBase/Async HBase Sink, 115

关于作者

Hari Shreedharan 是 Cloudera 的一名软件工程师，工作在 Apache Spark、Apache Flume 和 Apache Sqoop 等项目上。他也是 Flume 项目的提交者和 PMC 成员，帮助项目的方向做决策。他经常出席各种大数据相关会议和聚会。2010 年，Hari 获得了 Cornell University 的计算机科学硕士学位。

封面介绍

Using Flume 这本书封面上的动物，是江鳕（burbot 或 Lota lota），北方水域的鱼，通常生活在洁净、大型河流和深冷湖中。江鳕也称为玛丽亚、律师和粘鱼，与海生的常见鳕鱼关系密切。

江鳕看起来是与众不同的，头像鲶鱼，身体像鳗鱼，体积很小且摸起来光滑、黏糊糊的。下巴有一个单独的触须是它独特的标志（该鱼的名字来自拉丁语的“胡子”）。它们是侵略性的肉食动物，主要以鱼为食，但有时江鳕也吃昆虫，据知它们也吃青蛙、蛇和鸟类。

江鳕是唯一在冬至期产卵的淡水鱼类。产卵时水温在 32° 至 40°F 之间，通常在冰面以下。它们很难被研究，因为它们在冰下栖息和繁殖，但这为冬季垂钓者提供了巨大机会。事实上，在 Minnesota 的 Walker 镇上，每年冬天的 Leech 湖上都会举办一个国际粘鱼节日。

O’Reilly 封面上的许多动物都濒临灭绝；它们对世界很重要。想了解更多关于你能如何帮助的信息，请访问 *animals.oreilly.com*。

封面图片来自于 Meyers Kleines Lexicon。封面字体是 URW Typewriter 和 Guardian Sans。文本字体是 Adobe Minion Pro，标题字体是 Adobe Myriad Condensed，代码字体是 Dalton Maag 的 Ubuntu Mono。

O'REILLY®

Flume：构建高可用、可扩展的海量日志采集系统

如何能将前端服务器上获得的数据几近实时地放到Hadoop中去？有了这本完整的参考指南，你将学习Flume丰富的功能集合：将大量流数据收集、汇聚和写入到Hadoop分布式文件系统（HDFS）、Apache HBase、SolrCloud、ElasticSearch等系统中。

本书指导运维工程师如何配置、部署和监控Flume群集，并指导开发人员如何编写Flume插件和针对特殊使用情况的自定义组件。你将了解Flume的设计和实现，以及使其具备高度可扩展、灵活和可靠等特性的各种功能。

- **了解Flume如何通过数据生产者和消费者之间的缓冲区为流提供稳定的速率。**
- **深入Flume的关键组件，包括接收数据的Source和写入、转发数据的Sink。**
- **编写自定义插件来定制Flume接收、修改、格式化和写入数据的方式。**
- **探索从自己的应用程序发送数据到Flume Agent的API。**
- **以一种可扩展的、灵活的方式规划和部署Flume——一旦集群运行则监控集群。**

“运维人员将会发现，这本书对于如何轻松设置和部署Flume管道的理解是非常有价值的。开发人员将找到一个方便的参考，以在Flume内部构建或自定义组件，并更好地理解其架构和组件的设计。

“最重要的是，这本书将提供为HDFS和HBase建立持续输入的必要见解。”

——Arvind Prabhakar
StreamSets CTO

Hari Shreedharan是Cloudera的一名软件工程师。他工作于Apache Spark、Apache Flume和Apache Sqoop。他也是Flume项目的一位提交者和PMC成员，帮助决定项目的方向。

DATA

图书分类：大数据

责任编辑：张春雨

Broadview®
WWW.BROADVIEW.COM.CN

ISBN 978-7-121-26558-7

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

定价：69.00元