目录

第 1 章 大数据开发与集成概论 1
第 1 节 大数据行业现状及发展 1
第 2 节 大数据人才需求分析 4
第 3 节 大数据典型行业应用场景 5
第 4 节 大数据系统中的职能角色 7
第 5 节 支撑大数据应用的基础技术 10
第 2 章 Hadoop 入门概念及基础环境搭建 15
第 1 节 Hadoop 简介与安装入门 15
第 2 节 MapReduce 基本概念 26
第 3 节 Yarn 简介与安装入门 32
第 4 节 Hadoop 开发插件配置 37
第 5 节 Hadoop 集群配置 47
第 3 章 HDFS 50
第 1 节 HDFS 简介 50
第 2 节 HDFS 的数据分块 53
第 3 节 Namenode 和 Datanode 54
第 4 节 HDFS 的缺点和改进 59
第 5 节 HDFS 的 Shell 接口 60
第 6 节 HDFS 的文件接口 71
第 4 章 MapReduce 开发 85
第 1 节 MapReduce 概述 85
第 2 节 MapReduce 作业入门 91
第 3 节 MapReduce 入门数据分析实例 98
第 4 节 MapReduce 序列化 119
第 5 节 MapReduce 分区 128
第 6 节 Combiner 134
第 7 节 MapReduce 过程进阶 135
第 8 节 MapReduce2 和 MapReduce1 的差异 140
第 9 节 MapReduce 的 IO 操作 142
第 10 节 MapReduce 的数据压缩 145
第 11 节 容器格式-基于文件的数据结构 151
第 12 节 Hadoop 支持的第三方序列化框架 160
第 5 章 数据流脚本语言 Pig 170
第 1 节 Pig 入门 170
第 2 节 安装 Pig 171
第 3 节 Pig 的基本使用 172
第 4 节 Pig Latin 入门 178
第 5 节 Pig 模式 188
第 6 节 Pig 函数 193
第 7 节 更多数据操作 200
第 6 章 基于 HDFS 的数据仓库 Hive 207
第 1 节 Hive 基本介绍 207

第 2 节 Hive 安装.....209
第 3 节 Hive 基本使用.....211
第 4 节 托管表和外部表.....216
第 5 节 HiveQL 特性入门.....217
第 6 节 查询优化-分区和桶.....221
第 7 节 Hive 的存储格式.....225
第 8 节 Hive 查询操作进阶.....226
第 9 节 通过 JDBC 访问 Hive.....238
第 7 章 大数据查询引擎.....240
第 1 节 Presto.....240
第 2 节 Impala.....246
第 8 章 分布式数据库 HBase.....251
第 1 节 HBase 基本介绍.....251
第 2 节 HBase 核心概念.....251
第 3 节 HBase 安装.....255
第 4 节 HBase 的 Shell 操作.....257
第 5 节 HBase 操作示例.....259
第 6 节 HBase 开发 API.....262
第 9 章 Zookeeper.....271
第 1 节 Zookeeper 入门介绍.....271
第 2 节 安装 Zookeeper.....271
第 3 节 深入理解 Zookeeper.....273
第 4 节 Zookeeper 服务.....280
第 5 节 使用 Zookeeper 构建应用.....290
第 6 节 生产环境中的 Zookeeper.....297
第 10 章 数据导入与导出.....299
第 1 节 Sqoop 入门介绍.....299
第 2 节 Sqoop 的基本使用.....299
第 3 节 数据导入的示例.....301
第 4 节 深入 Sqoop 的导入与导出.....304
第 11 章 Flume.....316
第 1 节 Flume 入门介绍.....316
第 2 节 Flume 核心概念.....317
第 3 节 Flume 安装与使用.....319
第 4 节 Flume 配置文件.....324
第 12 章 Kafka.....326
第 1 节 Kafka 入门介绍.....326
第 2 节 Kafka 交互流程.....327
第 3 节 Kafka 安装与使用.....330
第 4 节 搭建多个 broker 的集群.....331
第 5 节 Kafka 开发.....333
第 13 章 内存计算框架 Spark.....341
第 1 节 由 Java 到 Scala.....341
第 2 节 Spark 入门.....344

第 3 节 安装使用 Spark 345
第 4 节 Spark Streaming 353
第 5 节 SparkSQL 355
第 14 章 Oozie 360
第 1 节 Oozie 入门介绍 360
第 2 节 安装 Oozie 361
第 3 节 运行 ooize 示例 365
第 4 节 oozie 工作流程定义 366
第 15 章 全文检索工具 372
第 1 节 Solr 372
第 2 节 Elasticsearch 387
第 16 章 图数据库 391
第 1 节 Titan 391
第 2 节 Neo4J 396
第 17 章 基于 CDH 的大数据平台运维 408
第 1 节 准备工作 408
第 2 节 支持库安装 411
第 3 节 MySQL 安装 412
第 4 节 集群管理组件安装配置 414
第 5 节 集群管理维护工具的基本使用 415

第1章 大数据开发与集成概论

第1节 大数据行业现状及发展

大数据已经上升为国家基础性战略资源，是新世纪的数字钻石矿。国家高度重视大数据在经济社会发展中的作用，国务院提出“实施国家大数据战略”，出台《促进大数据发展行动纲要》，全面推进大数据发展，加快建设数据强国。十三五时期是我国新旧产业和发展驱动转换接续的关键时期，全球新一代信息技术产业正处于加速变革期，国内市场应用需求处于爆发期，我国大数据产业发展面临重要的发展机遇。培养出足够的、合格的数据人才，对我国在未来掌握大数据的核心价值起着至关重要的作用。大数据是一种人类社会生产活动信息化的产物，它的内涵不止是海量的数据存储，还包括其相关的技术、领域应用、社会学和道德法律等等交叉学科的内容。受益于大数据市场规模快速扩大，对相关 IT 支持的需求出现了爆发式增长，其中为提供大数据基础设施的企业、大数据软件技术服务的企业、行业大数据内容咨询服务的企业带来空前的客户群体。

随着云时代的来临，移动互联网、电子商务、物联网以及社交媒体的快速发展，全球的数据正在以几何速度呈爆炸性增长，大数据（Big Data）也吸引了越来越多的关注。无处不在的信息感知和收集终端为我们收集了海量的数据，而以云计算为代表的计算技术的不断进步，为我们提供了强大的计算能力，构建起了一个与物质世界相平行的数字世界。“大数据”可能带来的巨大价值正渐渐被人们认可，它通过技术的创新与发展，以及数据的全面感知、收集、分析、共享，为人们提供了一种全新的看待世界的方法。

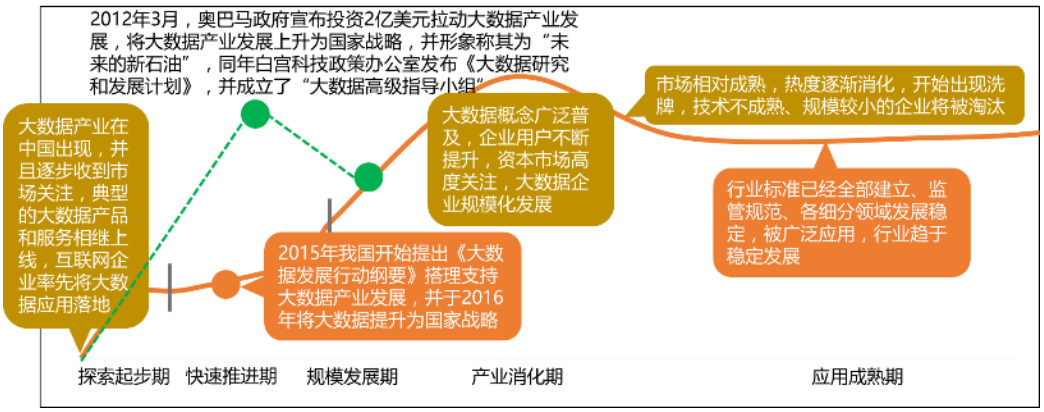
大数据，指无法在可承受的时间范围内用常规软件工具进行捕捉、管理和处理的巨量数据集合，是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力来适应海量、高增长率和多样化的信息资产。大数据具有 5V 特点（IBM 提出）：Volume（大量）、Velocity（高速）、Variety（多样）、Value（价值）、Veracity（真实性）。

从技术上看，大数据与云计算的关系就像一枚硬币的正反面一样密不可分。大数据必然无法用单台的计算机进行处理，必须采用分布式架构。它的特色在于对海量数据进行分布式数据挖掘。大量须依托云计算的分布式处理、分布式数据库和云存储、虚拟化技术。

大数据通常用来形容一个公司创造的大量非结构化数据和半结构化数据，这些数据在下载到关系型数据库用于分析时会花费过多时间和金钱。大数据分析常和云计算联系到一起，因为实时的大型数据集分析需要向数十、数百或甚至数千的电脑分配工作。

大数据需要特殊的技术，以有效地处理大量的数据。适用于大数据的技术，包括大规模并行处理（MPP）数据库、数据挖掘、分布式文件系统、分布式数据库、云计算平台、互联网和可扩展的存储系统。

随着全球范围内个人电脑、智能手机等设备的普及和新兴市场内不断增长的互联网访问量，以及监控摄像机或 RFID、传感器、GPS 等物联网相关设备产生的数据爆增，使数字宇宙的规模在 2012 到 2013 两年间翻了一番，达到惊人的 2.8ZB。IDC 预计，到 2020 年，数字宇宙规模将达到 40ZB。与此同时，数据的形式也极其丰富，既有社交网络、多媒体等应用所主动产生的数据，也有搜索引擎、网页浏览等被动行为过程中被记录、搜集的数据。在数据的这种爆炸式增长的背景下，“大数据”的概念逐渐在科技界、学术界、产业界引起热议。



大数据产业发展历程图

中国市场的规模之大也为这一产业发展提供了大空间、大平台。大数据对于中国企业不仅仅是信息技术的更新，更是企业发展战略的变革。随着对大数据的获取、处理、管理等各个角度研究的开展，企业逐渐认识数据已经逐渐演变成“数据资产”。为了充分利用数据资产，大数据产业呼之欲出。大数据处理的基础设施数据仓库、以物联网为代表的数据收集环节、实时性强的在线数据分析工具，以及数据可视化的产品呈现，数据挖掘的应用在营销、销售、人力资源、电子商务等各个商业领域广泛开展，大数据为个性化营销和精准化推荐提供了充足的养分和可持续发展的沃土。同时大数据研究会给企业管理变革带来巨大冲击。对现代企业的管理运作理念、组织业务流程、市场营销决策以及消费者行为模式等产生巨大影响，使得企业商务管理决策越来越依赖于数据分析而非经验甚至直觉。大数据将催生由信息驱动的商业模式，在企业的价值链中发挥中间作用，通过商业交易创建极具价值的“排出数据”；数据驱动的决策制定，利用可控实验，企业能够验证假设、分析结果以指导投资决策及运作改变；利用大数据进一步提高算法和机器分析的作用，避免成本高昂的人工干预，节约成本，提高效益。

信息技术与经济社会的交汇融合引发了数据迅猛增长，数据已成为国家基础性战略资源。坚持创新驱动发展，加快大数据部署，深化大数据应用，已成为稳增长、促改革、调结构、惠民生和推动政府治理能力现代化的内在需要和必然选择。

2009 年，美国政府推出 Data.gov，是为了增加政府透明度而设立的一系列网站。2012 年，美国白宫科技政策办公室发布《大数据研究和发展计划》，成立“大数据高级指导小组”。

2011 年开始，英国政府不断对大数据领域进行持续的专项资金投入。英国政府建立了有“英国数据银行”之称的 data.gov.uk 网站，通过这个公开平台发布

政府的公开政务信息。

2011 年 12 月，法国政府推出的公开信息线上共享平台 data.gouv.fr，便于公民自由查询和下载公共数据。2013 年 2 月，法国政府发布《数字化路线图》，明确了大数据是未来要大力支持的战略性高新技术。

2013 年 8 月， 澳大利亚政府发布了公共服务大数据战略。Data.gov.au 是澳大利亚政府信息目录的开放数据平台，用户可以在该网站上简便地搜索、浏览和利用澳大利亚国家、地区政府的公共数据。

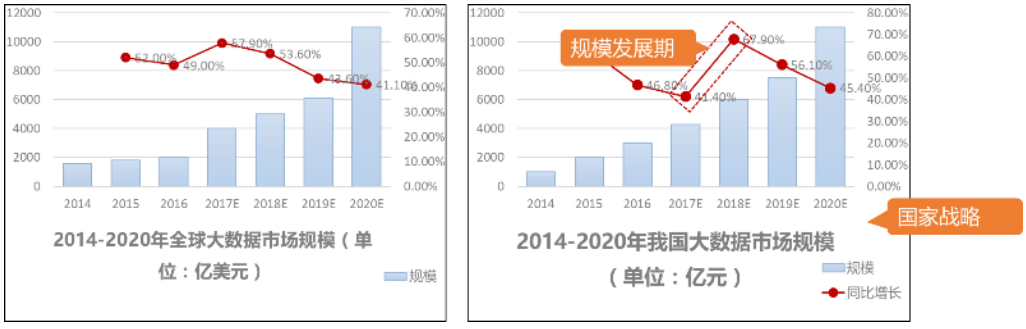
我国十三五规划纲要中明确提出“实施国家大数据战略”，开始着手布局国家大数据平台、数据中心等基础设施。

从国内外经验看，用数据来决策、用数据来管理、用数据来创新，已成为提升国家竞争力的一部分。作为“基础性战略资源”，大数据已经成为促进人类社会发展的“新的石油”、“新的资产”、“新的空间”。

经李克强总理签批，国务院日前印发《促进大数据发展行动纲要》（下称《纲要》），系统部署大数据发展工作。

《纲要》提出，要加强顶层设计和统筹协调，大力推动政府信息系统和公共数据互联开放共享，加快政府信息平台整合，消除信息孤岛，推进数据资源向社会开放，增强政府公信力，引导社会发展，服务公众企业；以企业为主体，营造宽松公平环境，加大大数据关键技术研发、产业发展和人才培养力度，着力推进数据汇集和发掘，深化大数据在各行业创新应用，促进大数据产业健康发展；完善法规制度和标准体系，科学规范利用大数据，切实保障数据安全。《纲要》明确，推动大数据发展和应用，在未来 5 至 10 年打造精准治理、多方协作的社会治理新模式，建立运行平稳、安全高效的经济运行新机制，构建以人为本、惠及全民的民生服务新体系，开启大众创业、万众创新的创新驱动新格局，培育高端智能、新兴繁荣的产业发展新生态。

目前大数据已经上升到我国的国家战略层面。培养出足够的、合格的数据人才，对我国在未来掌握大数据的核心价值起着至关重要的作用。大数据是一种人类社会生产活动信息化的产物，它的内涵不止是海量的数据存储，还包括其相关的技术、领域应用、社会学和道德法律等等交叉学科的内容。受益于大数据市场规模快速扩大，对相关 IT 支持的需求出现了爆发式增长，其中为提供大数据基础设施的企业、大数据软件技术服务的企业、行业大数据内容咨询服务的企业带来空前的客户群体。



大数据产业规模

第2节 大数据人才需求分析

根据领英最近发布的一份研究表明，过去四年从事数据分析的专业人士数量翻了一倍。相关工作的招聘量急剧上升，这一趋势反映了人们对高端数据分析人才的需求在不断增加，因为企业需要他们对数据进行各式处理分析，这必然会对2017年甚至以后的大数据局面产生影响。

另外，全球最著名的管理咨询公司麦肯锡发布大数据人才需求报告并预测大数据人才短缺，到 2018 年，在“具有深入分析能力的人才” 方面，美国可能面临着14万—19万人的缺口，而“可以利用大数据分析来作出有效决策的经理和分析师” 缺口则会达到 150 万。国内大数据人才有多大的缺口，目前尚无权威研究报告指出。在今年 2 月某集团公司的一份“求贤帖” 颇能说明问题：该集团开出50万—70万元年薪吸纳大数据架构师、大数据研发工程师和算法工程师，应者寥寥。同时，从如下研究报告中也可以看出来大数据人才的缺口。大数据将为全球带来440万个 IT 新岗位和上千万个非 IT 岗位。涵盖了大数据的数据开发工程师、大数据分析师、数据架构师、大数据后台开发工程师、算法工程师等多个方向。大数据技术应用岗位需求在京津冀城市群的需求量则超过 20万。

我国大数据产业发展已具备一定基础，但要实现从“数据大国”向“数据强国”转变，还面临诸多挑战。一是对数据资源及其价值认识不足。全社会尚未形成对大数据客观、科学的认识，对数据资源及其在人类生产、生活和社会管理方面的价值利用认识不足，存在盲目追逐硬件设施投资、轻视数据资源积累和价值挖掘利用等现象。二是技术创新与支撑能力不够。大数据需要从底层芯片到基础软件再到应用分析软件等信息产业全产业链的支撑，无论是新型计算平台、分布式计算架构，还是大数据处理、分析和呈现方面与国外均存在较大差距，对开源技术和相关生态系统的影响力仍然较弱，总体上难以满足各行各业大数据应用需求。三是数据资源建设和应用水平不高。用户普遍不重视数据资源的建设，即使有数据意识的机构也大多只重视数据的简单存储，很少针对后续应用需求进行加工整理。数据资源普遍存在质量差，标准规范缺乏，管理能力弱等现象。跨部门、跨行业的数据共享仍不顺畅，有价值的公共信息资源和商业数据开放程度低。数据价值难以被有效挖掘利用，大数据应用整体上处于起步阶段，潜力远未释放。最后，信息安全和数据管理体系尚未建立。数据所有权、隐私权等相关法律法规和信息安全、开放共享等标准规范缺乏，技术安全防范和管理能力不够，尚未建立起兼顾安全与发展的数据开放、管理和信息安全保障体系。

要解决以上几个现实问题的根本思路是加强大数据行业人才队伍建设。培养综合掌握数学、统计学、计算机等相关学科及应用领域知识的综合性数据科学人才，通过先进的具备行业应用底蕴的实践实验环境，满足既熟悉行业业务需求又掌握大数据技术与管理的综合型人才的实践需要。而站在以“新工科”作为高等工程教育改革发展新的历史起点上，更是对高校和企业进行更为融和的协同育人方式在多个方面提出了新的要求，需要通过校企联合制定更科学合理的培养目标和培养方案，校企共同建设课程与开发教程、共建实验室和实训实习基地，企业需要参与到教育教学各个环节中。

第3节 大数据典型行业应用场景

传统的数据应用主要集中在对业务数据的统计分析，作为系统或企业的辅助支撑，应用范围以系统内部或企业内部为主，例如各类统计报表、展示图表等。伴随着各种随身设备、物联网和云计算、云存储等技术的发展，数据内容和数据格式多样化，数据颗粒度也愈来愈细，随之出现了分布式存储、分布式计算、流处理等大数据技术，各行业基于多种甚至跨行业的数据源相互关联探索更多的应用场景，同时更注重面向个体的决策和应用的时效性。因此，大数据时代应用系统的数据形态、处理技术、应用形式都构成了与传统计算机应用系统区别。

整体来看，目前国内大数据应用尚处于从热点行业领域向传统领域渗透的阶段。权威部门的调查显示大数据应用水平较高的行业主要分布在互联网、电信、金融、电商、交通行业，一些传统行业的大数据应用发展较为缓慢，批发零售业甚至有超过 80%的企业并没有大数据应用计划，远低于整体平均水平。

- **运营商业务**

运营商掌握体量巨大的数据资源，单个运营商其用户每天产生的话单记录、上网日志等数据就可达到 PB 级规模。大数据对于运营商而言，意味着利用廉价便捷的大数据技术提升其传统的数据处理能力，聚合更多的数据提升洞察能力，借助大数据加快诊断网络潜在问题的效率，改善服务水平，为客户提供了更好的体验，获得了更多的客户以及更高的业务增长。

大数据系统中可以利用相关技术对运营商用户进行精准画像，形成大量有价值的标签数据，为客户服务和市场营销提供有力支持。如过对消费、通话、位置、浏览、使用和交往圈等数据的分析，利用各种联系记录发现各种圈子，分析影响力及关键人员，用来进行家庭客户、政企客户和关键客户的识别，以实现主动营销和客户维系。

- **金融业务**

金融行业是信息产业之外大数据的又一重要应用领域，大数据在金融三大业务：银行、保险和证券中均具有力广阔的应用前景。总体说来，金融行业的主要业务应用包括企业内外部的风险管理、信用评估、借贷、保险、理财、证券分析等，都可以通过获取、关联和分析更多维度、更深层次的数据，并通过不断发展的大数据处理技术得以更好、更快、更准确的实现。大数据分析应用为金融机构提供统一的客户视图。借助客户统一视图，可以从交易、服务、风险、权益等多个层面获取和分析数据，对客户按照低、中、高价值来进行分类，根据银行整体经营策略积极地提供相应的个性化服务，在降低成本的同时大幅提升精准营销能力。并利用大数据技术整合来自互联网等渠道的更大的外部数据。

- **政务业务**

大数据政务应用获得世界各国政府日益重视。我国政府也非常重视利用大数据提升国家治理能力。《国务院关于印发促进大数据发展行动纲要的通知》（国发〔2015〕50 号）提出“大数据成为提升政府治理能力的新途径”，要“打造精准治理、多方协作的社会治理新模式”。大数据应用中着眼于提升政府提供的公共

产品和服务。基于政务数据共享互通，模拟建设医疗、社保、教育、交通等民生事业大数据平台，提升民生服务，开展创新应用研究，深入发掘公共服务数据，有助于激发社会活力、促进大数据应用市场化服务，通过数据分析实现宏观调控科学化。通过对各部门、社会企业的经济相关数据进行关联分析和融合利用，可以提高宏观调控的科学性、预见性和有效性。比如电商交易、人流、物流、金融等各类信息的融合交汇可以绘出国家经济发展的气象云图，帮助人们了解未来经济走向，提前预知通货膨胀或经济危机。

- **交通领域业务**

交通数据资源丰富、具有实时性特征。在交通领域，数据主要包括各类交通运行监控、服务和应用数据，如公路、航道、客运场站和港口等视频监控数据，城市和高速公路、干线公路的各类流量、气象检测数据，城市公交、出租车和客运车辆卫星定位数据，以及公路和航道收费数据等，这些交通数据类型繁多，而且体积巨大。此外，交通领域的数据采集和应用服务均对实时性要求较高。

大数据应用系统可以基于对大数据的预测性分析，通过梳理影响安全运行的各种原因，发现道路运行安全管理的内在规律，将为交通管理决策、规划、运营、服务、以及主动安全防范带来更加有效的支持。

在出行方面，面向公众出行信息需求，整合交通出行服务信息，在公共交通、出租汽车、道路交通、轨道交通等领域扩大信息服务覆盖面，使公众出行更便捷。可以提供综合性、多层次信息服务，包括交通资讯、人流预测等动态信息服务以及出行路径规划等信息交互服务。

- **电子商务业务**

大数据开启了电子商务行业的时代转型。电商和传统商家最大区别在于：电商构建的各类型数据库能够涵盖商家信息、用户信息、行业资讯、产品使用体验、商品浏览记录、商品成交记录、产品价格动态等海量信息。电商行业大数据背后隐藏的是电商行业的用户需求、竞争情报，蕴藏着巨大的财富价值。借助大数据挖掘与分析技术，电商不仅可以提高营销转化为购买行为的成功率，还能降低营销成本，使产品更契合用户的需求，全面提升企业竞争力。

大数据应用系统可以集中在以下方面积极提升改进电商业务：1. 市场预测。基于大数据预测技术能够实现产品从开发、生产、销售到物流整个链条的智能化和快速反应。通过对海量数据的收集、甄别与分析处理，不仅可以为终端的市场用户勾勒出消费习惯、消费能力的“用户画像”，大数据分析还能获取产品在各区域、各时间段、各消费群的库存和预售情况与发展趋势等，基于大数据预测技术能够实现从产品开发、生产、销售到物流等的整个链条的智能化和快速反应。2. 精准营销。大数据对电商行业的影响最典型就在营销领域。对于电商来说，如何将产品定向推荐给需要的用户，始终是电商的核心关注点。消费数据量的急剧增加为电商企业精确把握用户群体和个体网络行为模式奠定了基础。电商企业通过大数据应用，划分细分群体与单体受众的心理层次，进行个人化、个性化、精确化和智能化广告推送与推广服务的探索。3. 用户体验。作为核心的服务理念，电子商务用户体验很大程度上决定了电子商务未来的成败。要让消费者最大限度的感受消费的归属感、满足感和幸福感，需要电商企业提供更智能、人性与差异化的服务，实现双赢的深度价值创造。在用户体验方面，各大电商包括垂直

电商都形成了各具特色的个性化服务，用以提升用户体验。如基于用户画像为客户量身定做咨询应答策略，如快速理解用户意图、针对性商品评测或商品推荐、个性化关怀等。4. 物流与仓储优化。电子商务与物流业的合作随着云计算、物联网和数据应用等技术的突破越来越紧密，大数据改变了物流业的服务方向和服务内容，对于客户数据的分析也就不仅局限于电商企业单向操作。通过对客户数据的分析企业能够更合理的选择派送方式，优选路径，提供差异化服务，提高物流服务的质量，提升电商物流业的品牌形象。

- **科学研究业务**

科学数据是人类在认识自然、发展科技的活动中产生和积累的数据，是人类长期科学活动的知识积累，是一种重要的基础资源和战略资源。大数据时代，科学模式已经变革为“数据密集型科学”的科研范式阶段，部分学科领域的科研活动已经成为了典型的大数据行为，科学家有机会利用海量的科学数据去探索世界，开展此前无法进行的研究，解决此前难以解决的科学问题，产生突破性进展。

- **教育领域业务**

教育大数据的主要目的是为不同利益相关者提供精准的教育服务，如学生的学习、教师的教学、开发者的资源开发、教育管理者的决策等；其核心是精准获取学习者的需求，为学习者提供精准教育服务；其数据主要来源于各类教育系统，包括学习管理系统 LMS、内容管理系统 CMS、电子学档系统 EPS、智能培训系统 ITS、社会性学习系统 SLS、实时教学系统 LTS、学习设计系统 LDS 和学生信息管理系统 SIS 等。应用和分析的教育大数据技术主要为教育数据挖掘技术和学习分析技术，当前研究热点为学习分析。学习分析是以理解和优化学习及其发生的环境为目的，对学习者及其所处情境的数据进行的测量、收集、分析和报告，其焦点是分析学习行为相关的数据、过程，以及呈现的方式。

- **健康医疗领域业务**

随着公共卫生领域的信息化建设，各地政府和公卫医疗行业都将医疗行业数据处理及共享作为信息化建设的重点之一，而随着国内首轮公卫医疗行业信息建设浪潮而来的是庞大的医疗信息，以及医疗系统间的信息不对称、不共享。同时，卫生数据分散在各医疗机构的信息系统中，与中心平台数据要求存在结构不同、标准不统一的情况，传统的由医疗机构系统开发商完成数据上报的方式存在医疗数据处理工作量大、项目开发进度缓慢、数据质量低、医疗单位协调难、数据监管难等问题。在这样的医疗行业需求背景下，医疗健康大数据处理思想的引入，上述难题将得到有效解决。通过对医疗大数据的获取和分析，并将数据与各级医疗平台进行实时共享，对分散医疗卫生机构的数据以及公众随身的健康医疗传感器数据进行快速、有效、可靠的采集，实现医疗卫生机构卫生数据的有效接入，将对公卫医疗信息化建设起到至关重要的作用。在大数据背景下，海量传感器数据的监管依靠自动化、智能化的方式进行统一的集中监管，以便于及时发现、定位、追溯、跟踪和解决问题，从而实现全民的健康跟踪和处理。

第4节 大数据系统中的职能角色

大数据本身既能形成新兴产业，也能推动其他产业发展。当前全球围绕大数

据采集、存储、管理和挖掘，正在逐渐形成一个小生态，即大数据核心产业。大数据核心产业为全社会大数据应用提供以资源、产品工具和应用服务，支撑各个领域的大数据应用，是大数据在各个领域应用的基石。应该注意到，目前大数据产业仍然围绕信息的采集加工构建，属于信息产业的一部分。

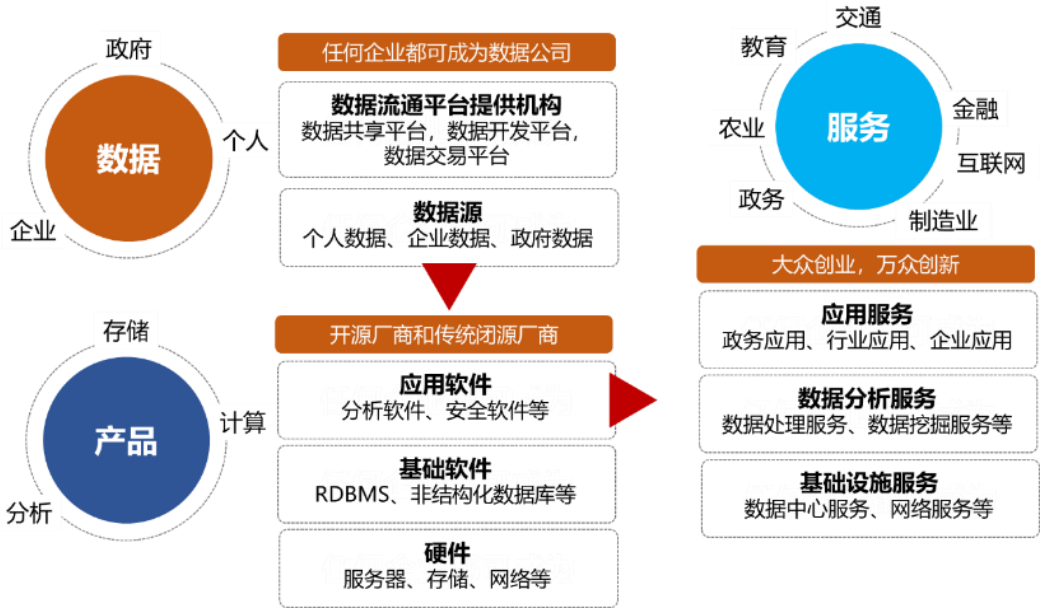
大数据是新资源、新技术和新理念的混合体，而不同的视角也展示了大数据业务系统针对的不同目标职业体系和需要应对的解决方案。

从资源视角来看，大数据体现了一种全新的资源观。从上个世纪 90 年代初，在摩尔定律的推动下，社会 IT 系统计算存储和传输数据的能力在以指数速度增长，数据存储承载设备的价格也在逐年下降，尤其是 2000 年依赖，以 Hadoop 为代表的分布式存储和计算技术迅猛发展，极大的提升了互联网企业数据管理能力，互联网企业对历史数据的挖掘利用大获成功，引发全社会开始重新审视数据的价值，开始把数据当作一种独特的战略资源对待。

从技术视角看，大数据代表了新一代数据管理与分析技术。传统的数据管理与分析技术以结构化数据为管理对象，在小数据集上进行分析，以集中式架构为主，成本高昂。与“贵族化”的数据分析技术相比，源于互联网的，面向多源异构数据、在超大规模数据集（PB 量级）上进行分析、以分布式架构为主的新一代数据管理技术，与开源软件潮流叠加，在大幅提高处理效率的同时，成百倍的降低了数据应用成本。

从理念的视角看，大数据打开了一种全新的思维角度。大数据的应用，赋予了“实事求是”新的内涵，其一是“数据驱动”，即经营管理决策可以自下而上地由数据来驱动，甚至像量化股票交易、实时竞价广告等场景中那样，可以由机器根据数据直接决策；其二是“数据闭环”，观察互联网行业大数据案例，它们往往能够构造起包括数据采集、建模分析、效果评估到反馈修正各个环节在内的完整“数据闭环”，从而能够不断地自我升级，螺旋上升。目前很多“大数据应用”，要么数据量不够大，要么并非必须使用新一代技术，但体现了数据驱动和数据闭环的思维，改进了生产管理效率，这是大数据思维理念应用的体现。

参与到大数据生态的各角色组成如下：



大数据生态角色图

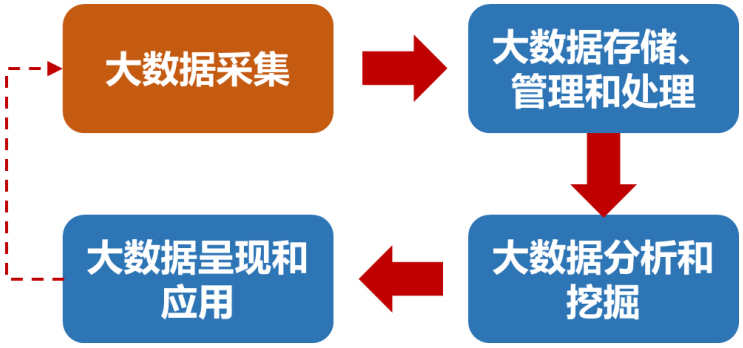
数据资源部分负责原始数据的供给和交换，根据数据来源的不同，可以细分为数据资源提供者和数据交易平台两种角色。

数据基础能力部分负责与数据生产加工相关的基础设施和技术要素供应，根据数据处理的实现流程，数据基础能力部分主要包括数据存储、数据处理和数据库（数据管理）等多个角色。

数据分析/可视化分负责数据隐含价值的挖掘、数据关联分析和可视化展现等，既包括传统意义上的 BI、可视化和通用数据分析工具，也包括面向非结构化数据提供的语音、图像等媒体识别服务。

数据应用部分根据数据分析和加工的结果，面向电商、金融、交通、气象、安全等细分行业提供精准营鞘、信用评估、出行引导、信息防护等企业或公众服务。

在过去的几年中，大数据产业快速增长态势不变，且远快于整个信息和通讯技术市场增长速度。大数据作为新兴领域，已经进入应用发展阶段，基础设施建设带来的规模性高速增长出现逐步放缓的趋势，技术创新和商业模式创新推动各行业应用逐步成熟，应用创造的价值在市场规模中的比重日益增大，并成为新的增长动力。



大数据角色业务闭环

- **大数据采集：**

此类角色收集和掌握着各种数据的产生来源，这些数据将逐渐成为整个业务系统进行处理分析业务的重要数据集资源。

- **大数据存储、管理和处理：**

此类角色从事着从海量数据中挑选出有用数据，并对采集到的数据进行传输、存储和管理。

- **大数据分析和挖掘：**

此类角色根据上游系统的需求，对相关数据进行定制化的分析。

- **大数据呈现和应用：**

此类角色将分析得到的数据进行可视化，结合相关专业知识和商业背景，揭示分析结果所反映的相关行业问题，并将相关数据应用于相关行业中，并通过业务系统的运行反哺数据采集角色。

在综合性大数据业务系统中，上述各个角色并不是严格区分开的，很多情况下都是模糊或者交叠的，例如，一些角色既是大数据的产生者，也是大数据技术和服务的提供者，既能够采集到数据，也能够对相应的数据进行分析，并进一步应用于相应的业务系统中：

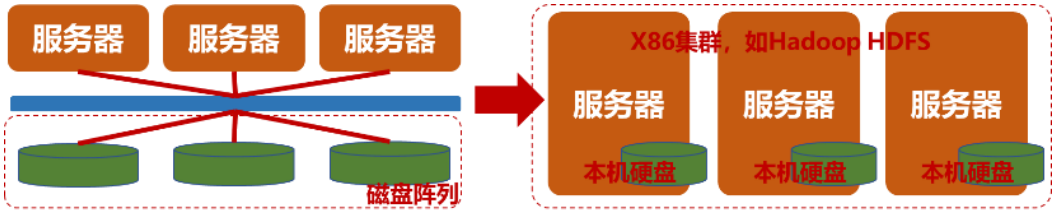
角色能力需要

第5节 支撑大数据应用的基础技术

- **存储和计算**

大数据存储和计算技术是整个大数据系统的基础。在存储方面，2000 年左右谷歌等提出的文件系统（GFS）、以及随后的 Hadoop 的分布式文件系统 IIDFS（Hadoop Distributed File System)奠定了大数据存储技术的基础。与传统系

统相比，GFS/HDFS 将计算和存储节点在物理上结合在一起，从而避免在数据密集计算中易形成的 I/O 吞吐量的制约，同时这类分布式存储系统的文件系统也采用了分布式架构，能达到较高的并发访问能力：



存储架构演进

在计算方面，谷歌在 2004 年公开的 MapReduce 分布式并行计算技术，是新型分布式计算技术的代表。一个 MapReduce 系统由廉价的通用服务器构成，通过添加服务器节点可线性扩展系统的总处理能力，在成本和可扩展性上都有巨大的优势。谷歌的 MapReduce 是其内部网页索引、广告等核心系统的基础。之后出现的 Apache Hadoop MapReduce 是谷歌 MapReduce 的开源实现，目前已经成为应用最广泛的大数据计算软件平台。MapReduce 架构能够满足先存储后处理的离线批量计算需求，但也存在局限性，最大的问题是时延过长，难以适用于机器学习迭代、流处理等实时计算任务，也不适合针对大规模图数据等特定数据结构的快速运算。为此，很多公司在 MapReduce 基础上，提出了多种不同的并行计算技术路线，如 Storm 系统是针对边到达边计算的实时流计算框架，可在一个时间窗口上对数据流进行在线实时分析。而 Spark 内存计算框架则是通过内存 IO 解决 MapReduce 过度使用磁盘 IO 导致的运算延迟。

此外 Apache 还在 Hadoop 基础上提供了很多数据传输、数据分析处理、管理与协同等工具，如 Avro、Hive、Pig、OoZie、ZooKeeper、Mahout、Tez 等等，使 Apache Hadoop 系列成为大数据开源界最有影响力的产品。

很多公司在 Apache Hadoop 的基础上进一步完善、开源自己的产品，最为著名的包括 Cloudera CDH（Cloudera's Distribution Hadoop）版本，Hortonworks HDP(Hortonworks Data Platform) 版本等。

Hadoop MapReduce 是批处理计算模型，其任务运行一次常常长达数小时、甚至几天或更长的周期；即便是几十条记录的小数据，MapReduce 处理一次也需要几秒的时间。也就是说，MapReduce 不适合分析处理实时性要求强的业务。这种情况下，可以采用流计算框架来进行改进。Storm 是 Twitter 正式开源的一个分布式的、容错的实时流计算系统，能够逐条接收和处理数据记录，具有很好的实时响应特性。Storm 实时计算提供了一组通用原语，可被用于“流处理”之中，实时处理消息并更新数据。借助实时的信息交互与通讯组件（如 Kafka、ZeroMQ、Netty 等），Storm 对大数据中的记录进行逐条处理，响应实时性可以达到秒级别甚至更短。

而且 Storm 能与 HDFS、YARN 有效集成，进一步拓展了其在大数据领域的适用范围。目前 Storm 已成为 Apache 下的顶级项目。

很多大数据分析场景需要多步骤的、迭代的分析计算，每次 Map 或 Reduce 都

需要大量文件 IO、排序等处理。虽然采用固态硬盘等硬件可以提升一部分性能，但与内存中的数据访问与计算相比就慢了很多。

Spark 是一个以 Map-Reduce 计算模型为原型实现的高效迭代计算框架，由 Berkeley 计算机系 AMPLab 实验室开发，并于 2010 年发布第一个版本。相比 MapReduce 计算模型，它通过把数据放入内存和有向无环图任务计划安排，大幅度缩减了批量迭代计算中的 IO 耗时呵和任务效率，可大幅度提升计算效率。据官方， Spark 程序在磁盘和内存中分别比同类 Hadoop MapReduce 要快 10 倍和 100 倍。

Spark 除了能与 HDFS、 YARN 有效集成，伯克利大学为其支持流处理能力提供了 Spark Streaming 组件，使具有了更快的分析计算响应速度。目前 Spark 也已成为 Apache 下的顶级项目，并已经被 FaceBook、Twitter、亚马逊、百度、淘宝等多家国际化互联网公司的大数据团队所使用， 是近年来迅速成长起来的大数据分析计算技术体系。

以 Hadoop 为代表的开源软件大幅度降低数据的存储与计算的成本。，新一代计算平台 Spark 进一步把 Hadoop 性能提升了 30 多倍，性能越来越高，技术门槛越来越低。

目前，开源 Hadoop 和 Spark 已经形成了比较成熟的产品供应体系，基本上可以满足大部分企业建设大数据存储和分析平台的需求，为企业提供了低成本解决方案，成为了业界首选。

- **NoSQL 数据库系列**

传统关系型数据库运用通常离不开 SQL。 而 NoSQL 则是 Not Only SQL 的首字母缩写。 NoSQL 数据库通过降低对数据的一致性要求，相比关系数据库而言更适应分布式应用。这一特性使得 NoSQL 数据库更能适合大数据应用。常见 NoSQL 可以按照其数据组织与结构进一步细分为键值(Key-Value)存储数据库、列存储数据库、文档型数据库以及图数据库等。基于 HDFS 的 Apache HBase 就是其中的典型代表。

- **数据采集**

经过行业信息化建设，医疗、交通、金融等领域已经积累了许多内部数据，构成大数据资源的“存量”：而移动互联网和物联网的发展，大大丰富了大数据的采集渠道，来自电商、外部社交网络、可穿戴设备、车联网、物联网及政府公开信息平台的数据将成为大数据增量数据资源的主体。

快速发展的物联网，也成为越来越重要的大数据资源提供者，这也将是企业大数据业务系统内部数据采集的一个重要途径。相对于现有互联网数据杂乱无章和价值密度低的特点，通过定制传感器、可穿戴、车联网等多种数据采集终端，定向采集的数据资源更具利用价值。例如，智能化的可穿戴设备经过几年的发展，智能手环、腕带、手表和各类定制健康传感器正在走向成熟，智能钥匙扣、自行车、等设备层出不穷，国外 Intel、Google、Facebook，国内百度、京东、小米等都有所布局。

- **前沿业务技术**

大数据业务分为联机分析处理（OLAP，Online Analytical Processing)和数据挖掘(Data Mining)两大类。OLAP 技术，一般基于用户的一系列假设，在多维数据集上进行交互式的数据集查询、关联等操作（一般使用 SQL 语句）来验证这些假设，代表了演绎推理的思想方法。

数据挖掘技术，一般是在海量数据中主动寻找模型，自动发展隐藏在数据中的模式，代表了归纳的思想方法。传统的数据挖掘算法主要有：

聚类：

又称群分析，是研究（样品或指标）分类问题的一种统计分析方法，针对数据的相似性和差异性将一组数据分为几个类别。属于同一类别的数据间的相似性很大，但不同类别之间数据的相似性很小，跨类的数据关联性很低。企业通过使用聚类分析算法可以进行客户分群，在不明确客户群行为特征的情况下对客户数据从不同维度进行分群，再对分群客户进行特征提取和分析，从而抓住客户特点推荐相应的产品和服务。

分类：

类似于聚类，但是目的不同，分类可以使用聚类预先生成的模型，也可以通过经验数据找出一组数据对象的共同点，将数据划分成不同的类，其目的是通过分类模型将数据项映射到某个给定的类别中，代表算法是分类与回归树。企业可以将用户、产品、服务等各业务数据进行分类，构建分类模型，再对新的数据进行预测分析，使之归于已有类中。分类算法比较成熟，分类准确率也比较高，对于客户的精准定位、营销和服务有着非常好的预测能力，帮助企业进行决策。

回归：

反映了数据的属性值的特征，通过函数表达数据映射的关系来发现属性值之间的一览关系。它可以应用到对数据序列的预测和相关关系的研究中。企业可以利用回归模型对市场销售情况进行分析和预测，及时作出对应策略调整。在风险防范、反欺诈等方面也可以通过回归模型进行预警。

传统的数据方法，不管是传统的 OLAP 技术还是数据挖掘技术，都难以应付大数据的挑战。首先是执行效率低。传统数据挖掘技术都是基于集中式的底层软件架构开发，难以并行化，因而在处理 TB 级以上数据的效率低。其次是数据分析精度难以随着数据量提升而得到改进，特别是难以应对非结构化数据。在人类全部数字化数据中，仅有非常小的一部分（约占总数据量的 1%）数值型数据得到了深入分析和挖掘（如回归、分类、聚类），大型互联网企业对网页索引、社交数据等半结构化数据进行了浅层分析（如排序），占总量近 60%的语音、图片、视频等非结构化数据还难以进行有效的分析。

因此业务系统在以下几个方面尤为关注。一是对体量庞大的结构化和半结构化数据进行高效率的深度分析，挖掘隐性知识，如从自然语言构成的文本网页中理解和识别语义、情感、意图等：二是对非结构化数据进行分析，将海量复杂多源的语音、图像和视频数据转化为机器可识别的、具有明确语义的信息，进而从中提取有用的知识。在应用系统中往往以深度神经网络等新兴技术为代表的大数据分析技术来应对这一需求。

神经网络是一种先进的人工智能技术，具有自身自行处理、分布存储和高度容错等特性，非常适合处理非线性的以及模糊、不完整、不严密的知识或数据，十分适合解决大数据挖掘的问题。

第2章 Hadoop 入门概念及基础环境搭建

第1节 Hadoop 简介与安装入门

Hadoop 由 Apache Lucene 创始人 Doug Cutting 创建的，Lucene 是一个应用广泛的文本搜索系统库。Hadoop 起源于开源的网络搜索引擎 Apache Nutch，它诞生之初是作为 Luncene 项目的一个重要核心组成部分。

2008 年 1 月，Hadoop 已成为 Apache 的顶级项目，证明了它的成功、多样化和生命力。到目前为止，除雅虎之外，还有很多公司在用 Hadoop，例如 Last.fm、Facebook 和《纽约时报》等。

2008 年 4 月，Hadoop 打破世界纪录，成为最快的 TB 级数据排序系统。在一个 910 节点的群集，Hadoop 在 209 秒内（不到 3.5 分钟）完成了对 1TB 数据的排序，击败了前一年的 297 秒冠军。同年 11 月，谷歌在报告中声称，它的 MapReduce 对 1TB 数据排序只用了 68 秒。2009 年 5 月有报道称雅虎有一个的团队使用 Hadoop 对 1TB 数据进行排序只花了 62 秒。

现在，Hadoop 已经跃升为企业主流的部署系统。在工业界，Hadoop 已经是公认的大数据通用存储和分析平台，这一事实主要体现在大量直接使用或者间接辅助 Hadoop 系统的产品如雨后春笋般大量涌现。一些大公司也发布 Hadoop 发行版本，包括 EMC,IBM,Microsft 和 Oracle 以及一些专注于 Hadoop 的公司，如 cloudera,Hortonworks，和 MapR。

尽管 Hadoop 因 MapReduce 及其分布式文件系统（HDFS）而出名，但 Hadoop 这个名字也用于泛指一组相关的项目，这些相关项目都使用这个基础平台进行分布式计算和海量数据处理，例如：

项目名称	项目描述
Flume-ng	Flume 是一个分布式、高可靠、高可用的海量日志聚合系统
YARN	新一代资源管理框架，允许多个应用集群同时高效地运行在一个物理集群上
HDFS	Apache Hadoop 分布式文件系统 (HDFS) 是 Hadoop 应用程序使用的主要存储系统。HDFS 创建多个数据块副本并将它们分布在整个群集的计算主机上，以启用可靠且极其快速的计算功能
Hbase	非关系型分布式 NoSQL 数据库，与传统数据库相比，采用列的方式进行存储，具有高加载低延迟的特性，用于千亿级数据的快速查询

Hive	基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行。 其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析
Presto	Presto 是一个开源的分布式 SQL 查询引擎，适用于交互式分析查询，数据量支持 GB 到 PB 字节。 Presto 的设计和编写完全是为了解决像 Facebook 这样规模的商业数据仓库的交互式分析和处理速度的问题，它拥有比 Hive 更高的执行效率，并针对不同的数据源提供了对应的连接器 ，用于实现统一的 ETL
Oozie	Oozie 是一个工作流调度引擎，可按时间或数据变化触发运行，是群集中管理数据处理作业的工作流协调服务
Pig	数据流处理语言，按照语法以流程化的方式描述数据处理流程，并以分布式的方式完成数据的处理，适合于使用 Hadoop 和 MapReduce 平台来查询大型半结构化数据集
Sqoop	主要用于在 Hadoop(Hive)与传统的数据库(mysql、postgresql...)间进行数据的传递，可以将一个关系型数据库（例如 ： MySQL ,Oracle ,Postgres 等）中的数据导进到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导进到关系型数据库中
Zookeeper	一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现，是 Hadoop 和 Hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等
Impala	Impala 为存储在 HDFS 和 HBase 中的数据提供了一个实时 SQL 查询接口。Impala 需要 Hive 服务，并共享 Hive Metastore
Solr	一个独立的企业级搜索应用服务器，它对外提供类似于 Web-service 的 API 接口。用户可以通过 http 请求，向搜索引擎服务器提交一定格式的 XML 文件，生成索引；也可以通过 Http Get 操作提出查找请求，并得到 XML 格式的返回结果
Key-Value Store Indexer(Lily)	键/值 Store Indexer 侦听 HBase 中所含表内的数据变化，并使用 Solr 为其创建索引
Titan	基于图的数据库,通过节点和边建立关系网络
Kafka	一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据

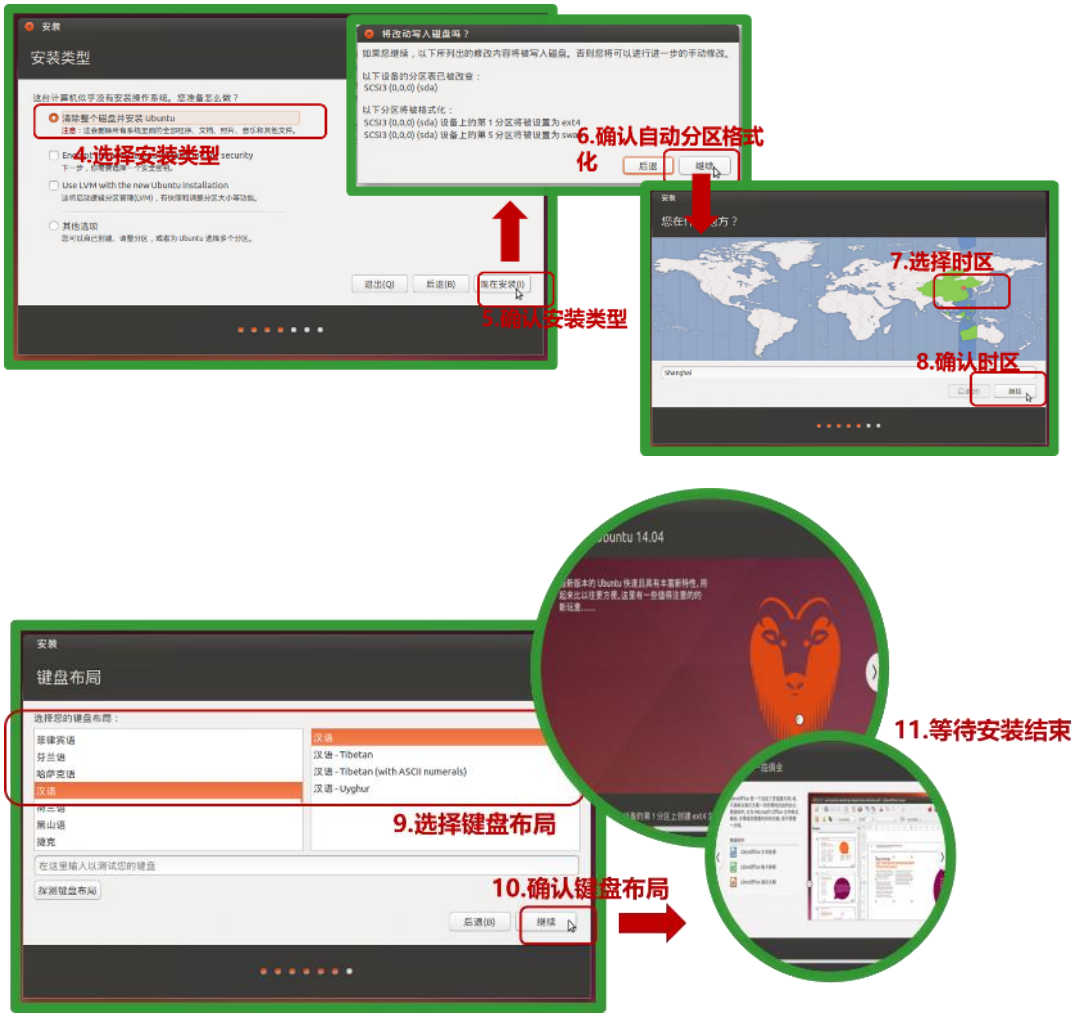
Storm	Apache Storm 是一个分布式的、可靠的、容错的数据流处理系统，适用于实时分析、在线机器学习、连续计算、分布式 RPC、分布式 ETL 等等
Spark SQL	Shark 的前身，给熟悉 RDBMS 但又不理解 MapReduce 的技术人员提供快速上手的工具
Keepalived	主要用作 RealServer 的健康状态检查以及 LoadBalance 主机和 BackUP 主机之间 failover 的实现
Kylin	一个分布式的 OLAP 和多维数据分析工具

安装 Apache Hadoop 的基础步骤如下：

- 安装 Linux 系统（以 Ubuntu 为例）
- 安装配置 JDK
- 安装配置 SSH（Secure Shell，可以用来进行远程控制或在计算机之间传送文件）
- 安装 Hadoop
- 运行示例代码
- 配置 Hadoop 集群

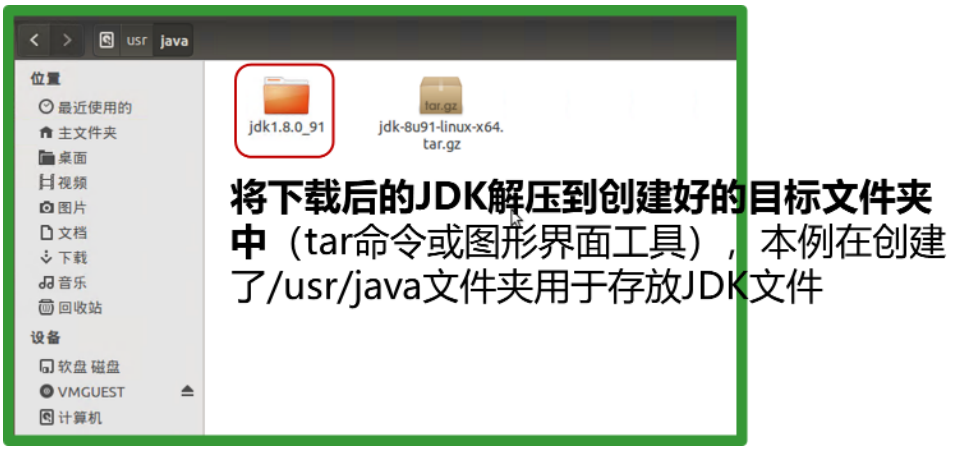
Hadoop 支持主流的 Linux 发行版本，如 RedHat、CentOS、Ubuntu 等。为了便于初学者尽快熟悉 Hadoop 环境，能够快速入门，在搭建开发测试环境时可以选择在虚拟机中安装更为易用的 Ubuntu 发行版本作为 Hadoop 环境的操作系统支持。

利用 Ubuntu 安装盘、ISO 或引导 U 盘引导系统后进入安装界面后按照安装程序的提示完成 Ubuntu 的安装，本步骤大约会消耗 15-20 分钟：



接下来需要在新安装的 Ubuntu 中安装 Java 开发工具 JDK，建议不要使用 Ubuntu 提供的 APT 包管理工具进行安装，因为默认情况下 APT 包管理工具在安装 JDK 时会自动下载 OpenJDK，而 OpenJDK 对 Hadoop 的支持存在问题（CentOS 中的 yum 包管理器也存在类似的问题）。因此可以在 Java 官网可以在 Java 的官方网站上找到 Linux 可用的 JDK 下载后手动完成安装：

将下载后的 JDK 解压到创建好的目标文件夹中：

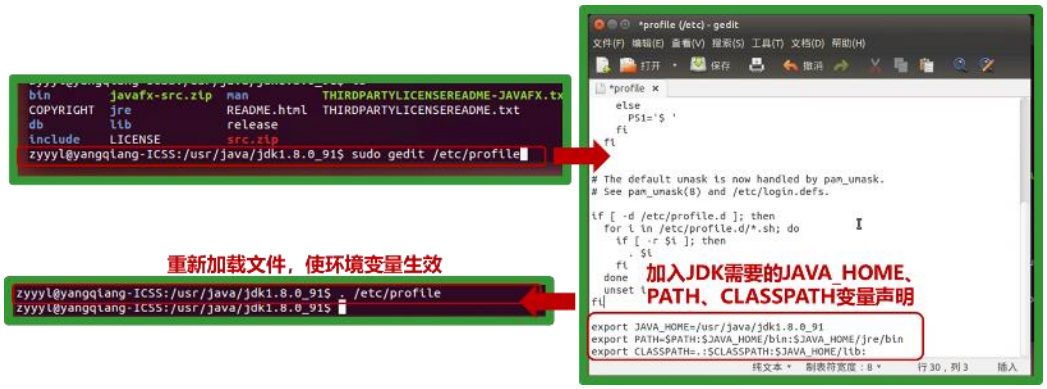


配置 JDK 所需要的环境变量，修改/etc/profile 文件，添加环境变量声明，需要注意 Linux 中环境变量的设置语句和 Windows 的区别（Linux 中设置命令为 export，而 Windows 中为 set，Linux 中每个变量值的分隔符为：而 Windwos 中为;,Linux 中引用其它变量的符号为$,而 Windows 中引用其他变量的符号为%）。

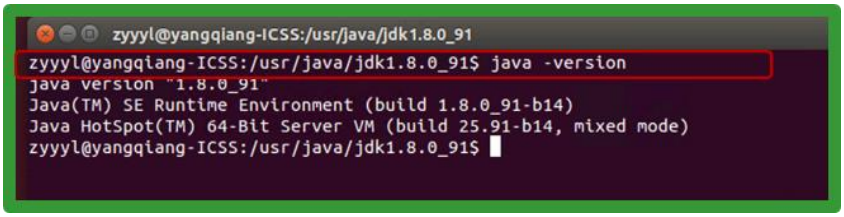
Java 所需的环境变量实例如下：

```
export JAVA_HOME=/usr/jdk/jdk18
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/jre/bin
export CLASSPATH=.:$CLASSPATH:$JAVA_HOME/lib
```

添加环境变量后使其重新生效：



配置成功后可以利用 JAVA 提供的工具命令测试环境是否成功：

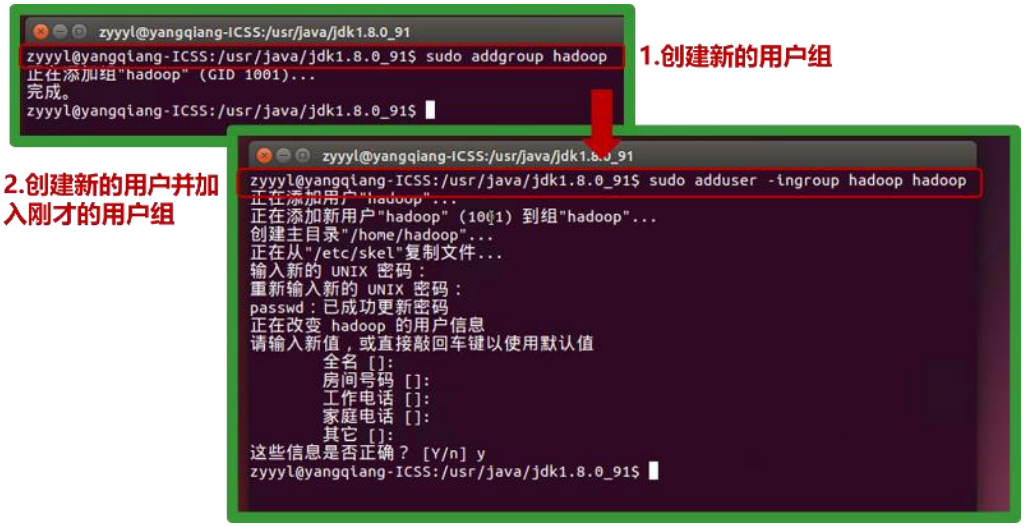


Linux 中可以配置环境变量的文件如下，可以根据需要选择修改不同的文件

来实现同样的目的：

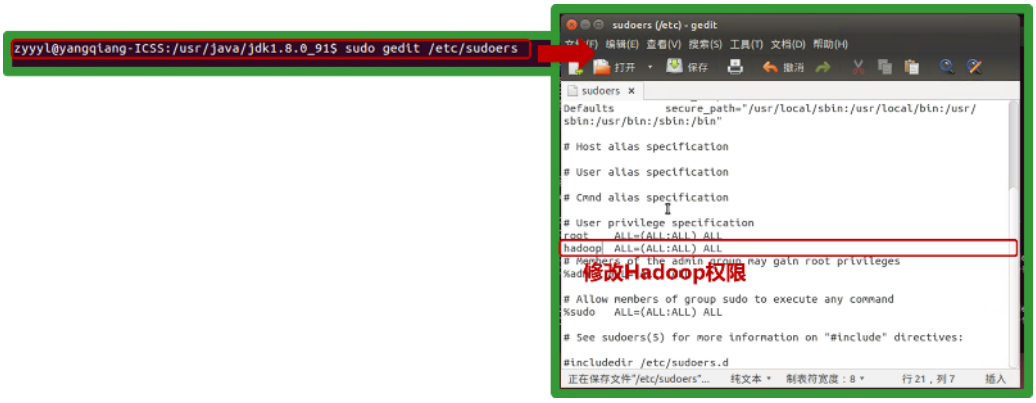
- /etc/profile:此文件为系统的每个用户设置环境信息,当用户第一次登录时,该文件被执行,并从/etc/profile.d 目录的配置文件中搜集 shell 的设置
- /etc/bashrc:为每一个运行 bash shell 的用户执行此文件，当 bash shell 被打开时,该文件被读取
- ~/.bash_profile:每个用户都可使用该文件输入专用于自己使用的 shell 信息,当用户登录时,该文件仅仅执行一次!默认情况下,他设置一些环境变量,执行用户的.bashrc 文件
- ~/.bashrc:该文件包含专用于你的 bash shell 的 bash 信息,当登录时以及每次打开新的 shell 时,该文件被读取。

为了对 Hadoop 各组件进行统一的管理，可以创建一个专门的用户来进行操作。在 Linux 中首先创建一个新的用户组，然后创建一个新的用户，并将该用户加入到刚创建的用户组中：



为新建的用户添加 sudo 权限，sudo 的配置文件是/etc/sudoers 文件，它允许系统管理员集中的管理用户的使用权限和使用的主机。在 sudoers 文件中按照已有的 root 账户模式，为新建的 hadoop 用户提供任意主机操作权限，配置如下：

```
root    ALL=(ALL:ALL) ALL
hadoop  ALL=(ALL:ALL) ALL
```



Hadoop 的操作依赖 SSH，因此需要在服务器中安装 SSH 服务，可以通过 Linux 发行版本的包管理器完成 SSH 的自动安装操作，在 Ubuntu 中，通过命令：

```
sudo apt-get install openssh-server
```

即可实现 SSH 服务的安装，具体步骤如下：

为了在 Hadoop 集群中，主节点能够方便快捷的对从节点进行控制（启动/关闭服务等），可以设置 SSH 的免密访问。

首先创建密钥文件，命令如下：

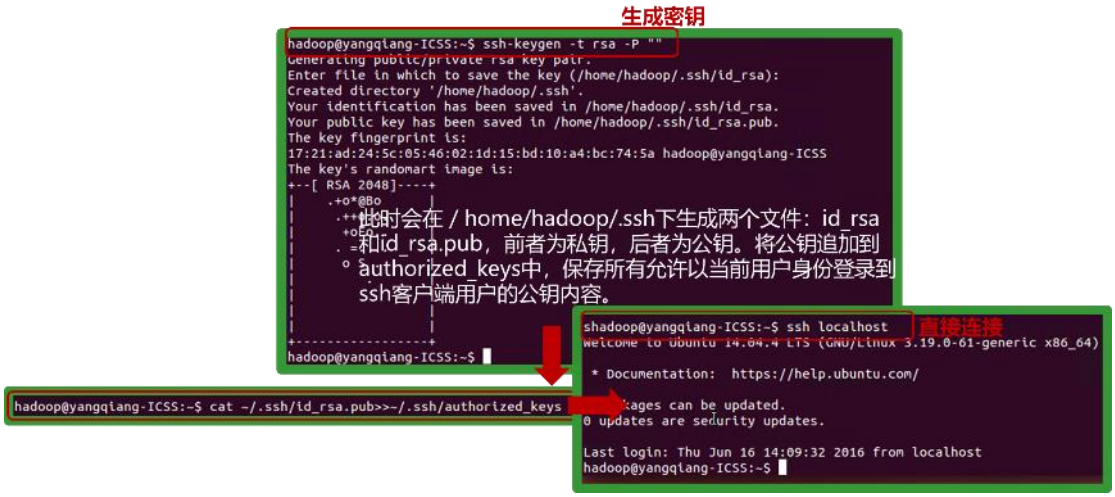
```
ssh-keygen -t rsa -P “”
```

注意参数中的 P 为大写形式。

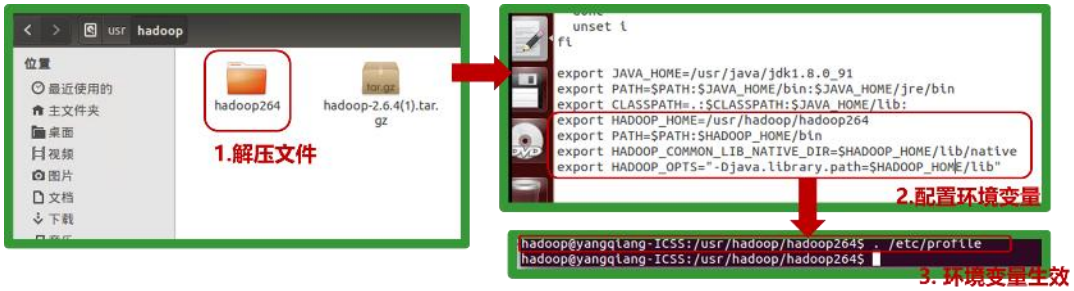
接下来保存密钥信息：

```
cat ~/.ssh/id_rsa.pub>>~/.ssh/authorized_keys
```

具体步骤如下：



接下来下载 Hadoop（http://hadoop.apache.org/releases.html）并解压，配置必要的环境变量。



Hadoop 需要的环境变量参考如下：

```
export HADOOP_HOME=/usr/hadoop/hadoop264
export PATH=$PATH:$HADOOP_HOME/bin
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/native"
```

修改 Hadoop-env.sh 中的 JAVA_HOME 路径，Hadoop-env.sh 文件位于 HADOOP_HOME/etc/hadoop 中：

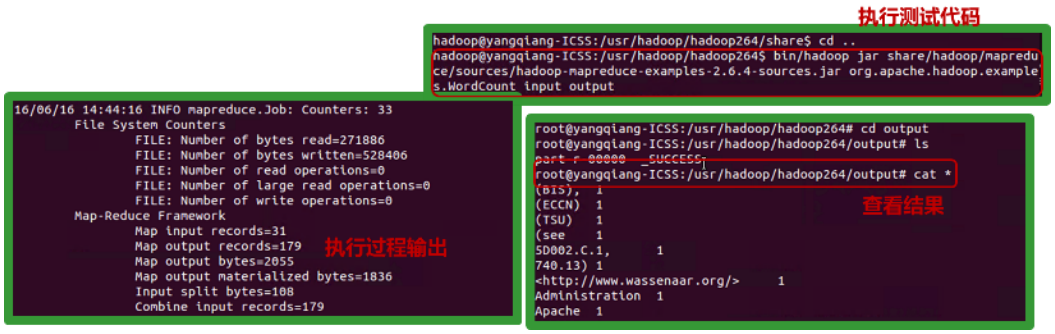
执行完上述步骤后 Hadoop 的单机模式即可使用，可以利用 Hadoop 提供的示例测试环境安装配置是否成功。

测试 MapReduce 领域的 Hello World-单词计数：

创建示例输入目录，并将 Hadoop 自带的 readme 文件作为输入数据拷贝到输入目录中：

```
hadoop@yangqiang-ICSS:/usr/hadoop/hadoop264$ sudo mkdir input
[sudo] password for hadoop:
hadoop@yangqiang-ICSS:/usr/hadoop/hadoop264$ sudo cp README.txt input
hadoop@yangqiang-ICSS:/usr/hadoop/hadoop264$ cd input
hadoop@yangqiang-ICSS:/usr/hadoop/hadoop264/input$ ls
README.txt
hadoop@yangqiang-ICSS:/usr/hadoop/hadoop264/input$
```

执行示例代码：



Hadoop 可以在单节点上以伪分布式的方式运行，Hadoop 进程以分离的 Java 进程来运行，节点既作为 NameNode 也作为 DataNode，同时，读取的是 HDFS 中的文件。

- NameNode：namenode 一是管理文件系统文件的元数据信息（包括文件名称、大小、位置、属性、创建时间、修改时间等等），二是维护文件到块的对应关系和块到节点的对应关系，三是维护用户对文件的操作信息（文件的增删改查）与任务调度
- DataNode：存放数据

Hadoop 的配置文件位于 $HADOOP_HOME/etc/hadoop/ 中，伪分布式需要修改 2 个配置文件 core-site.xml 和 hdfs-site.xml 。Hadoop 的配置文件是 xml 格式，每个配置以声明 property 的 name 和 value 的方式来实现。

修改 core-site.xml 文件，配置默认的临时数据路径和默认的文件系统，默认文件系统中规定了当前 Hadoop 集群的 HDFS 组件对外提供服务的端口号：

core-site.xml 的参考配置如下：

```
<property>
    <name>hadoop.tmp.dir</name>
    <value>file:/usr/hadoop/hadoop264/tmp</value>
</property>
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://192.168.1.1:9000</value>
</property>
```

修改 hdfs-site.xml，确定当前集群中 HDFS 文件系统保证文件高可用性的文件块副本数量（该数量的默认值为 3，且取值不能超过实际 datanode 的数据量，因此在伪分布式环境中，应该始终设置为 1）：

hdfs-site.xml 的参考配置如下：

```
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
<property>
    <name>dfs.namenode.name.dir</name>
    <value>file:/usr/hadoop/hadoop264/tmp/dfs/name</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>file:/usr/hadoop/hadoop264/tmp/dfs/data</value>
</property>
```

Hadoop 的运行方式是由配置文件决定的（运行 Hadoop 时会读取配置文件），因此如果需要从伪分布式模式切换回非分布式模式，需要删除 core-site.xml 中的配置项。

此外，伪分布式虽然只需要配置 fs.defaultFS 和 dfs.replication 就可以运行（官方教程如此），不过若没有配置 hadoop.tmp.dir 参数，则默认使用的临时目录为 /tmp/hadoo-hadoop，而这个目录在重启时有可能被系统清理掉，导致必须重新执行文件系统的格式化才行。所以我们进行了设置，同时也指定 dfs.namenode.name.dir 和 dfs.datanode.data.dir，否则在接下来的步骤中可能会出错。

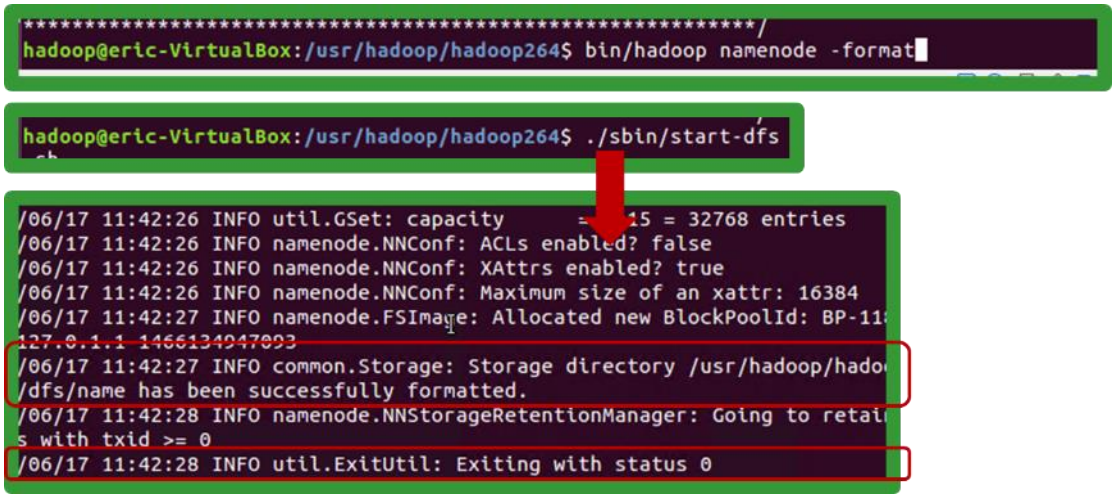
格式化 dfs，执行以下指令（指令位于 HADOOP_HOME/bin 中）：

```
hadoop namenode -format
```

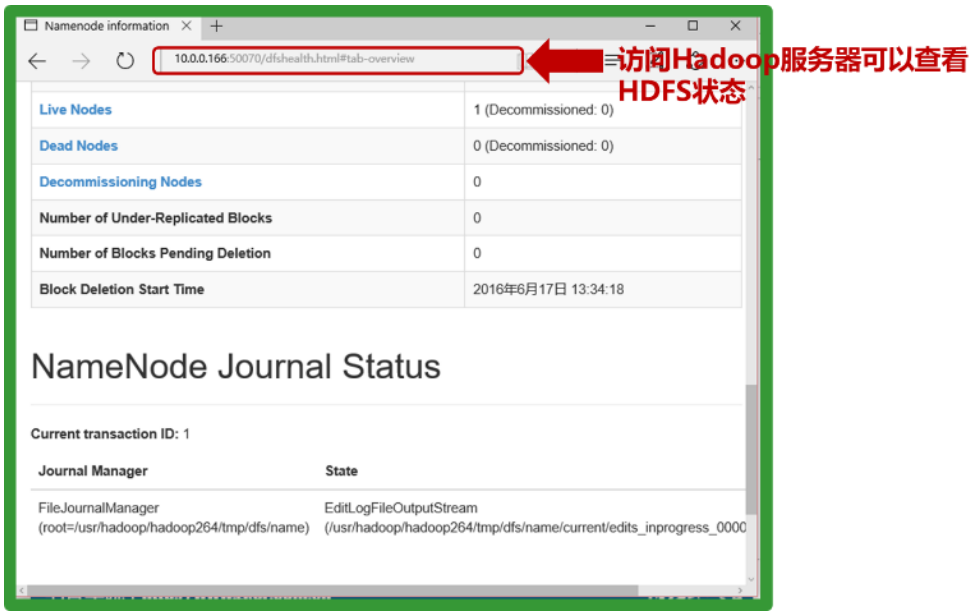
格式化成功，并且状态码为 0，说明配置成功。

启动 HDFS 服务，执行以下指令（指令位于 HADOOP_HOME/sbin 中）：

```
start-dfs.sh
```



启动 HDFS 服务以后，通过 WEB 端可以查看到其运行状态，在能够通过网络访问到服务器的任意主机上通过浏览器访问 http://服务器域名或 IP 地址:50070 即可：



第2节 MapReduce 基本概念

Google 提出并首先在自己的产品中运用 M-R 模型。MapReduce 任务过程分为两个处理阶段：map 阶段和 reduce 阶段。每个阶段都以键值对作为输入和输出，其类型由程序员来选择。程序员还需要写两个函数：map 函数和 reduce 函数。

可以利用果汁生产的流程来简单解释 MapReduce 的执行过程。可以将水果看作业务系统的数据输入，而最终的果汁产品则视作业务系统的结果输出。在早期的普通计算机业务系统中，数据数据集规模较小，对应的是小规模水果的榨汁流程，直接将水果提供给单一的榨汁机完成榨汁即可，而到了现代的大数据应用系统，海量的水果交给单一的榨汁机需要很长的时间才能完全处理，因此可以提供更多的榨汁节点并发分布式的完成榨汁，每一台榨汁机针对一个水果子集并形成

一个中间结果（map），然后将所有榨汁机的成果汇总后形成最终产品（reduce），通过并发的分布式任务调度，解决了针对海量输入的处理能力缺失的问题，极度降低处理延时：

MapReduce 将对计算的计算过程拆分为两个步骤：Map 和 Reduce

- Map 阶段并行处理输入数据
- Reduce 阶段对 Map 结果进行分析处理
- Shuffle 连接 Map 和 Reduce 两个阶段
  - Map Task 将数据写到本地磁盘
  - Reduce Task 从每个 Map Task 上读取一份数据

MapReduce 的特点：

- 由于在数据操作的过程中使用了大量的磁盘 IO 操作，延迟性较高，仅适合离线批处理。
- 具有很好的容错性和扩展性
- 适合简单的批处理任务
- 启动开销大、过多使用磁盘导致效率低下等

MapReduce 的基础结构如下：

- 主从结构
  - 主节点，只有一个：JobTracker
  - 从节点，有很多个：TaskTrackers
- JobTracker 负责：
  - 接收客户提交的计算任务
  - 把计算任务分给 TaskTrackers 执行

- 监控 TaskTracker 的执行情况

- TaskTrackers 负责：
  - 执行 JobTracker 分配的计算任务

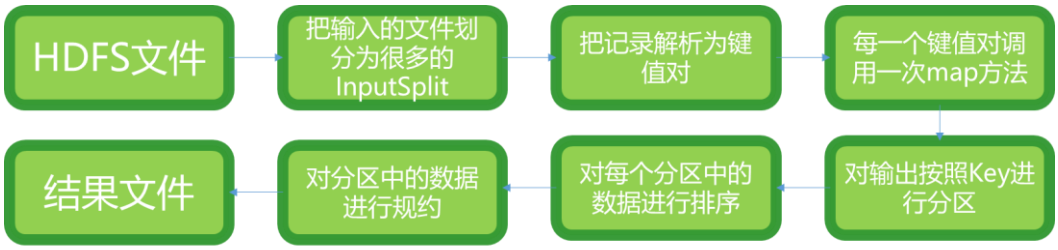
Map 函数和 Reduce 函数是交给用户实现的，这两个函数定义了任务本身。

map 函数：接受一个键值对（ key-value pair ），产生一组中间键值对。MapReduce 框架会将 map 函数产生的中间键值对里键相同的值传递给一个 reduce 函数。

reduce 函数：接受一个键，以及相关的一组值，将这组值进行合并产生一组规模 更小的值（通常只有一个或零个值）。

从业务系统执行的流程来看，Map 负责清洗数据，Reduce 负责数据分析并输出最终结果，而且这两个功能之间并非一对一的关系，可以根据具体业务选择匹配关系。

MapReduce 任务的具体执行流程如下：



利用分块和分区进行分布式并发操作
以提高对海量数据操作的效率

第一阶段是把输入文件按照一定的标准分片(InputSplit)，每个输入片的大小是固定的。默认情况下，输入片(InputSplit)的大小与数据块(Block)的大小是相同的。如果数据块(Block)的大小是默认值 128MB，输入文件有两个，一个是 32MB，一个是 172MB。那么小的文件是一个输入片，大文件会分为两个数据块，那么是两个输入片。一共产生三个输入片。每一个输入片由一个 Mapper 进程处理。这里的三个输入片，会有三个 Mapper 进程处理。

第二阶段是对输入片中的记录按照一定的规则解析成键值对。有个默认规则是把每一行文本内容解析成键值对。“键”是每一行的起始位置(单位是字节)，“值”是本行的文本内容。

第三阶段是调用 Mapper 类中的 map 方法。第二阶段中解析出来的每一个键值对，调用一次 map 方法。如果有 1000 个键值对，就会调用 1000 次 map 方法。每一次调用 map 方法会输出零个或者多个键值对。

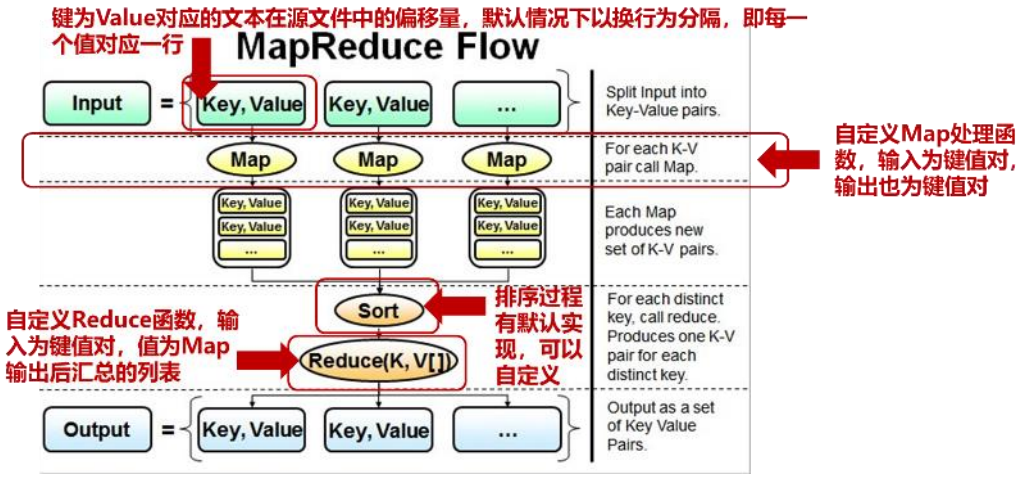
第四阶段是按照一定的规则对第三阶段输出的键值对进行分区。比较是基于键进行的。比如我们的键表示省份(如北京、上海、山东等)，那么就可以按照不同省份进行分区，同一个省份的键值对划分到一个区中。默认是只有一个区。分

区的数量就是 Reducer 任务运行的数量。默认只有一个 Reducer 任务。

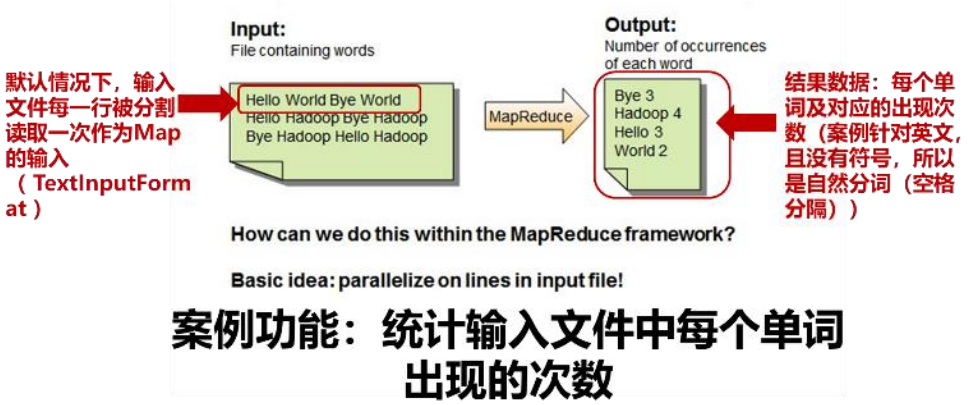
第五阶段是对每个分区中的键值对进行排序。首先，按照键进行排序，对于键相同的键值对，按照值进行排序。比如三个键值对<2,2>、<1,3>、<2,1>，键和值分别是整数。那么排序后的结果是<1,3>、<2,1>、<2,2>。如果有第六阶段，那么进入第六阶段；如果没有，直接输出到本地的 linux 文件中。

第六阶段是对数据进行归约处理，也就是 reduce 处理。键相等的键值对会调用一次 reduce 方法。经过这一阶段，数据量会减少。归约后的数据输出到本地的 linxu 文件中。本阶段默认是没有的，需要用户自己增加这一阶段的代码。

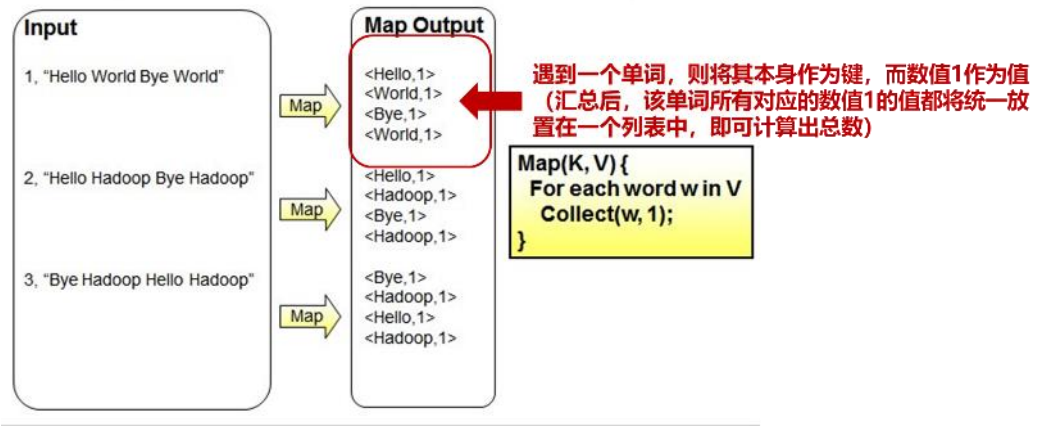
MapReduce 任务的数据从输入到输出经过的流程如下：



接下来通过上一节中已经执行过的入门案例 WordCount 来具体展示 MapReduce 任务的执行过程。该案例实现的功能是统计每一个单词在所有的输入文件中出现的次数：



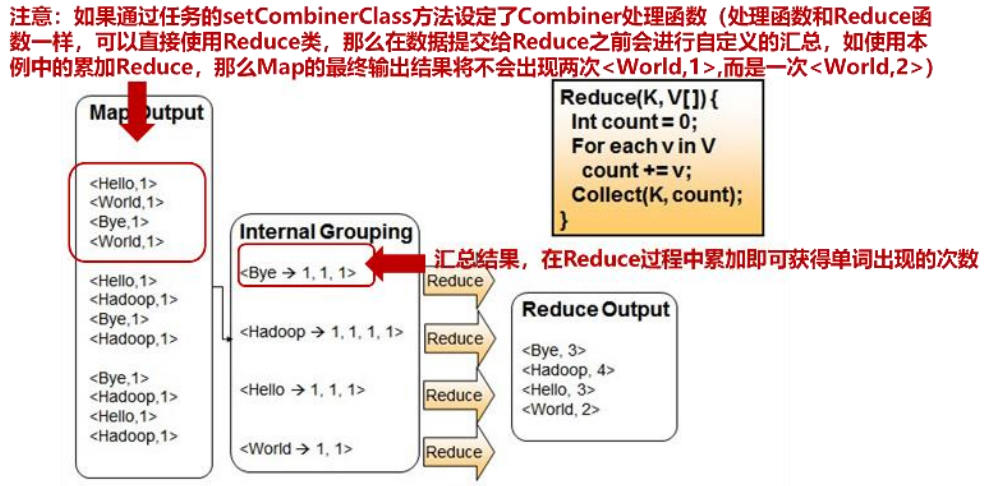
案例的 Map 任务中首先对输入文件进行分词（针对英文的自然分词，中文分词需要借助特殊算法或辅助工具），然后将针对分词得到的每一个单词将单词本身作为键，数字 1 作为值输出（表示在输入文件中，特定单词出现了一次）：



Hadoop 自身提供的 WordCount 案例代码相对复杂，在经过简化并保证功能不变的情况下，Map 任务的参考代码如下：

```
    public static class TokenizerMapper extends
            Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            StringTokenizer itr = new
StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }

    }
```

进入 Reduce 任务之前，Hadoop 将会对所有的 Map 任务中间结果进行汇总，主要操作有两个：首先将相同的键合并，对应的值构成一个集合，其次，根据键对合并后的中间结果进行排序。因此在单词统计案例中，Reduce 任务只需要对每个输入的键（每个单词）对应的值集合进行求和统计，即可汇总每个单词出现的次数：



Reduce 任务的参考代码如下：

```
    public static class IntSumReducer extends
            Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws IOException,
InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }

    }
```

任务调度的代码参考如下：

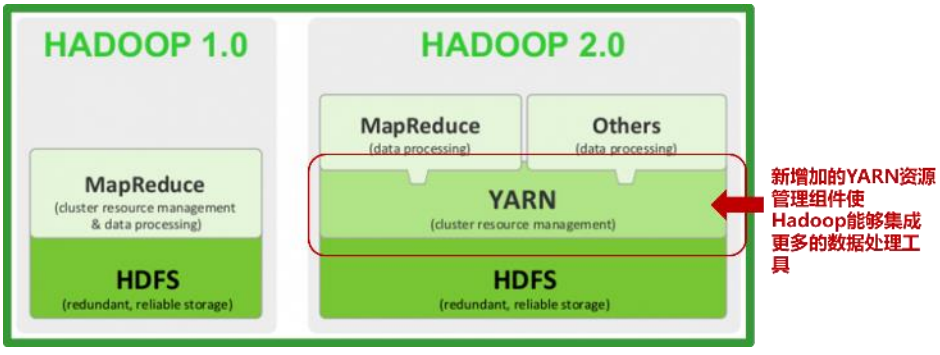
```
Configuration conf = new Configuration();
String[] otherArgs = new GenericOptionsParser(conf, new String[] {
    "hdfs://111.207.49.184:8020/user/cloudera/bdes",
    "hdfs://111.207.49.184:8020/user/cloudera/bdes/output" })
    .getRemainingArgs();
Job job = Job.getInstance(conf, "wc");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
```

```
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
job.setNumReduceTasks(2);
job.setPartitionerClass(WordCountPartitioner.class);

return job.waitForCompletion(true) ? 0 : 1;
```

第3节 Yarn 简介与安装入门

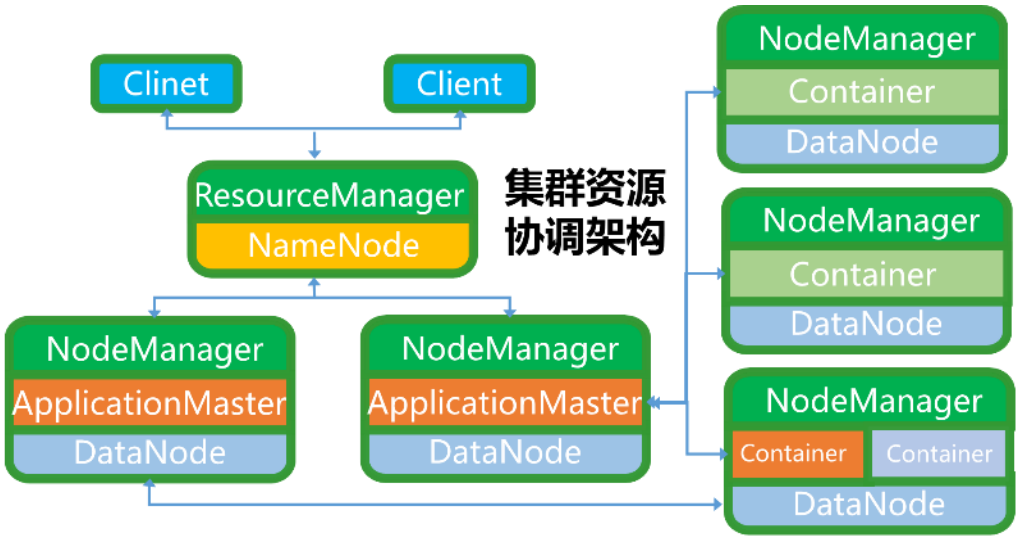
Hadoop1.0 中的任务调度存在以下问题：节点扩展性受限；单点故障，可靠性受限；难以支持 MR 之外的计算：

Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。



引入 Yarn 后，Hadoop 的任务调度和执行框架进行了层次化管理，使得其可以和其他的计算框架进行融合集成。YARN 的基本思想是将 JobTracker 的两个主要功能（资源管理和作业调度/监控）分离，主要方法是创建一个全局的 ResourceManager（RM）和若干个针对应用程序的 ApplicationMaster（AM）。这里的应用程序是指传统的 MapReduce 作业或作业的 DAG（有向无环图）。

Yarn 的结构如下图所示：



YARN 分层结构的本质是 ResourceManager。它控制整个集群并管理应用程序向基础计算资源的分配。

ResourceManager 将各个资源部分精心安排给基础 NodeManager（YARN 的每节点代理），并与 ApplicationMaster 一起分配资源，与 NodeManager 一起启动和监视它们的基础应用程序。

在此上下文中，ApplicationMaster 承担了以前的 TaskTracker 的一些角色，ResourceManager 承担了 JobTracker 的角色。

在整个集群中，ResourceManager 只有一个，它负责集群资源的统一管理和调度，主要负责：

- 处理客户端请求
- 启动/监控 ApplicationMaster
- 监控 NodeManager
- 资源分配与调度

其中资源包括网络、计算、内存等。

ApplicationMaster 管理在 YARN 内运行的应用程序的每个实例。

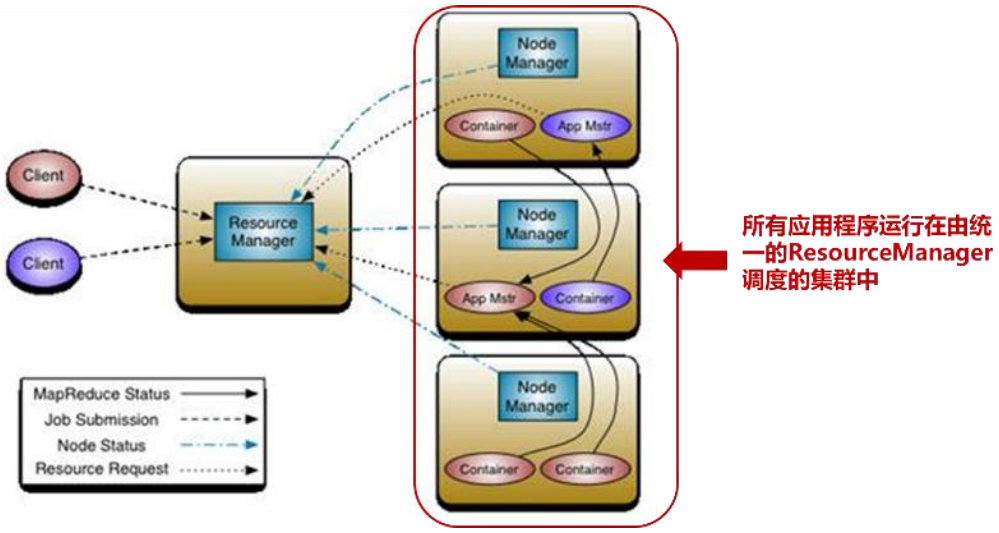
ApplicationMaster 负责协调来自 ResourceManager 的资源，并通过 NodeManager 监视容器的执行和资源使用（CPU、内存等的资源分配）。请注意，尽管目前的资源更加传统（CPU 核心、内存），但未来会带来基于手头任务的新资源类型（比如图形处理单元或专用处理设备）。从 YARN 角度讲，ApplicationMaster 是用户代码，因此存在潜在的安全问题。YARN 假设 ApplicationMaster 存在错误或者甚至是恶意的，因此将它们当作无特权的代码对待。

每个应用有一个 ApplicationMaster，负责应用程序的管理，包括数据切分、为应用程序申请资源，并进一步分配给内部任务以及任务监控与容错。

YARN 集群中的每个节点包含一个 NodeManager。它提供针对集群中每个节点的服务，从监督对一个容器的终生管理到监视资源和跟踪节点健康。MRv1 通过插槽管理 Map 和 Reduce 任务的执行，而 NodeManager 管理抽象容器，这些容器代表着可供一个特定应用程序使用的针对每个节点的资源。YARN 继续使用 HDFS 层。它的主要 NameNode 用于元数据服务，而 DataNode 用于分散在一个集群中的复制存储服务。

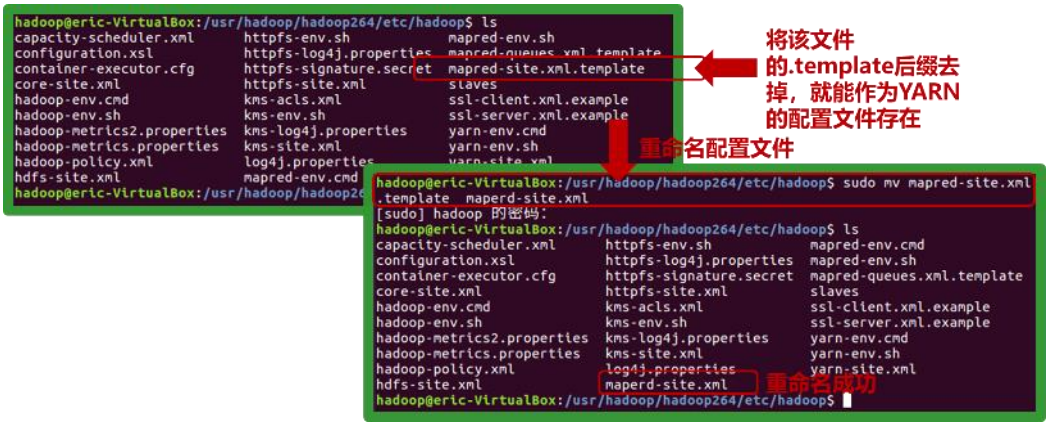
Container 则是对任务运行环境的抽象，它描述一系列信息，包括任务运行资源（节点、内存、CPU）、任务启动命令以及任务运行环境。

Yarn 中节点与节点之间的关系如下：



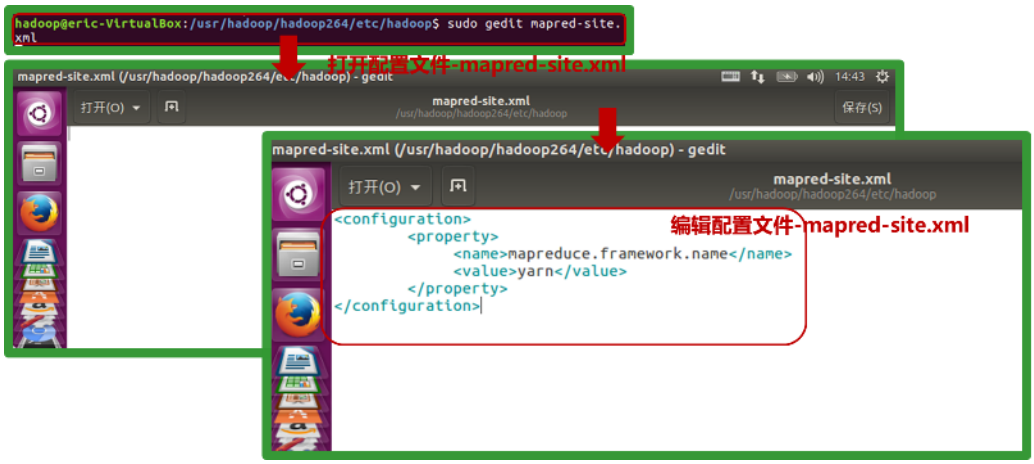
在按照前序章节的介绍安装配置 Hadoop 并启动服务后，系统中找不到见不到 JobTracker 和 TaskTracker 进程。因为在没有进行 YARN 配置时，通过./sbin/start-dfs.sh 启动 Hadoop，仅仅是启动了 MapReduce 环境，可以配置并启动 YARN ，让 YARN 来负责资源管理与任务调度。

Yarn 是 Hadoop2 原生提供的组件，已经包含在 Hadoop 的发行包中，不需要额外下载，要启动 Yarn 服务，只需要正确修改 Yarn 配置文件并启动对应的服务即可。Hadoop 提供了 YARN 配置文件模板，该模板位于 HADOOP_HOME/etc/hadoop 路径中，模板文件的文件名为 mapred-site.xml.template，首先需要将该文件复制一份，并改名为 mapred-site.xml：



修改该配置文件，该文件和 Hadoop 的 core-site.xml 和 hdfs-site.xml 一样，也是标准的 XML 文件，并且以 <property> <name> </name> <value> </value> </property>的结构对属性进行配置，一个 mapred-site.xml 的参考配置如下：

```
<property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>

 </property>
```



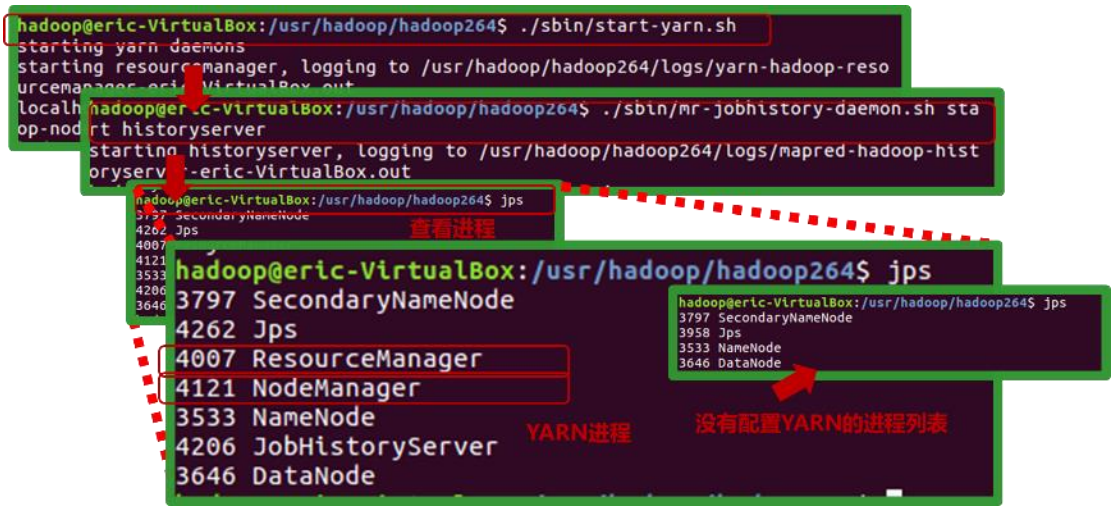
同时，修改相同目录中的 yarn-site.xml，将其内容修改为：

```
<?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing,
software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
  See the License for the specific language governing permissions
and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>
        <property>
             <name>yarn.nodemanager.aux-services</name>
             <value>mapreduce_shuffle</value>
            </property>
```

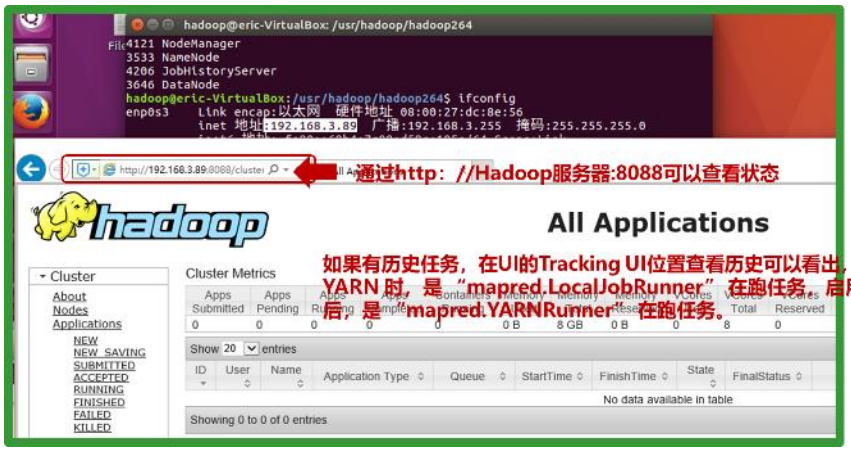
```
</configuration>
```

完成以上几个步骤，即可启动 Yarn 服务作为 Hadoop 的任务调度器，具体步骤如下：

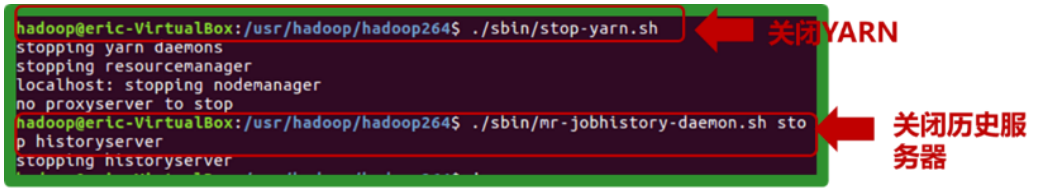
- 启动 MapReduce：./sbin/start-dfs.sh
- 启动 YARN：./sbin/start-yarn.sh
- 启动历史服务器：./sbin/mr-jobhistory-daemon.sh start historyserver



开启历史服务器后，可以在 Web 中查看 Hadoop 任务运行情况，通过能够访问到服务器的任何主机，访问 http://服务器域名或 IP 地址:8088 即可：



YARN 主要是为集群提供更好的资源管理与任务调度，然而这在单机上体现不出价值，反而会使程序执行得稍慢些。因此在单机上是否开启 YARN 需要综合考虑实际情况。关闭 YARN 的脚本如下：



需要注意的是，如果不想启动 YARN，务必把配置文件 mapred-site.xml 重命名，改成 mapred-site.xml.template，需要用时改回来就行。否则在该配置文件存在，而未开启 YARN 的情况下，运行程序会提示 “Retrying connect to server: 0.0.0.0/0.0.0.0:8032” 的错误，这也是为何该配置文件初始文件名为 mapred-site.xml.template。

第4节 Hadoop 开发插件配置

Hadoop 大部分组件的原生开发语言和 API 都基于 Java，因此可以直接使用常见的 Java IDE 作为开发工具。如果使用 Eclipse，还可以安装对应的插件工具方便的在 IDE 中以图形化的方式管理 Hadoop 连接和 HDFS 文件。

由于在 Windows 中利用 Eclipse 插件管理 HDFS 文件会出现很多难以控制的问题（包括 Hadoop 的本地库替换等），因此建议在 Linux 中安装 Eclipse 和对应插件。

在安装 Eclipse 和插件之前，有一个重要的准备工作。需要在主机的 hosts 文件中正确设置 Hadoop 集群中所有服务器的主机名和 IP 地址的映射。如果对应的映射关系设置有误，将会导致客户端和服务器端的握手过程失败，丢失连接。由于 Hadoop 内部会反复执行正向和反向的域名/主机名寻址，因此即便在程序代码中所有提供的地址均直接以 IP 的形式提供，也不能忽略这个准备操作。

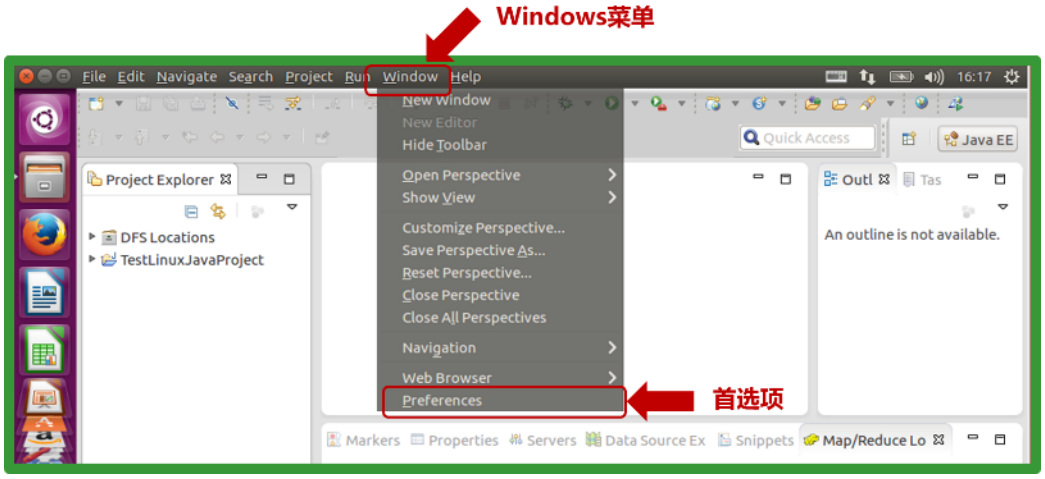
Linux 中 hosts 文件的路径为/etc/hosts。

Eclipse 本身无需任何配置，直接解压到目标文件夹即可：

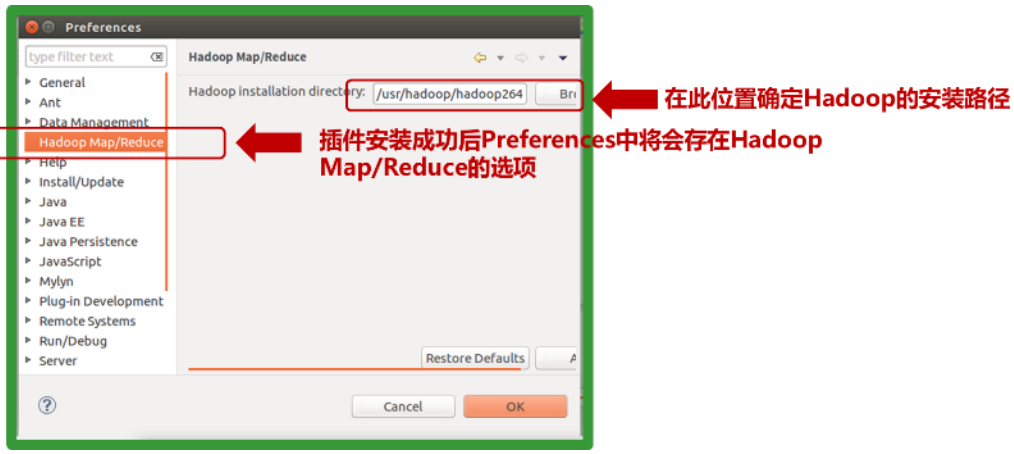
解压后直接执行文件夹中的 eclipse 文件即可运行 Eclipse 开发工具（首先确认在运行 Eclipse 的主机上是否成功安装了 JDK），如果已经安装了 JDK，运行 Eclipse 时仍然提示无法找到 Java 虚拟机，则在 Eclipse 的安装目录中创建一个名为 jre 的子目录，将 JDK 的所有文件拷贝到其中，再次执行 eclipse 即可：

下载对应版本的 Hadoop-Eclipse 插件，解压后是一个标准的 jar 包，保证当前用户对该 jar 包拥有执行的权限，将其直接拷贝到 Eclipse 的插件目录中：

完成以上步骤后，运行 Eclipse 工具，并对 Hadoop 插件进行必要的配置。点击 Eclipse 的【Windows】菜单中的【Preferences】菜单项，进入到设置界面：

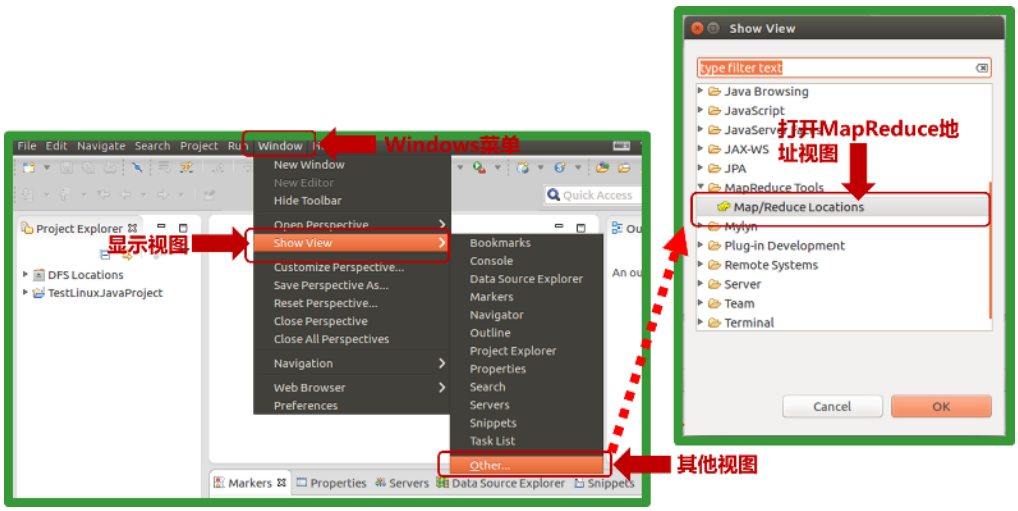


可以看到 Eclipse 的设置中此时已经存在了【Hadoop Map/Reduce】选项卡，进入该选项卡，确定 Hadoop 的安装路径。需要说明的是，Eclipse 并不一定要安装到 Hadoop 服务所在的主机，但是要求运行 Eclipse 的主机中需要存在 Hadoop 的二进制文件：

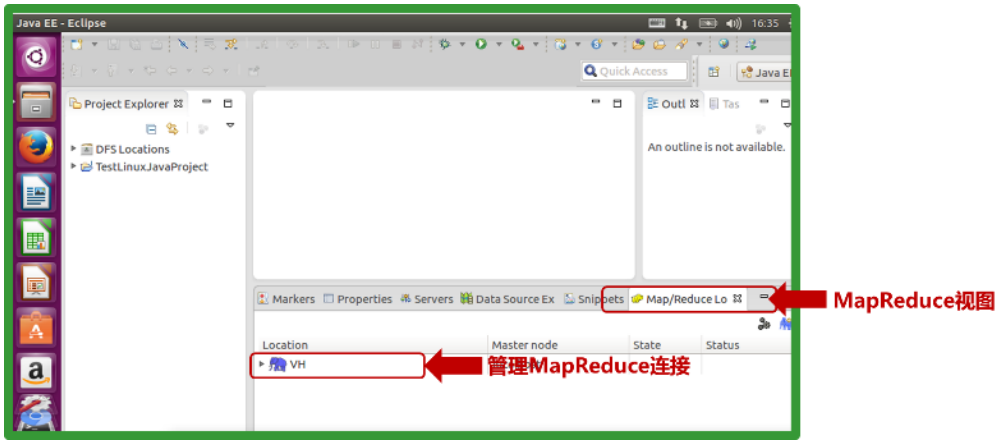


正确配置 Hadoop 插件后，可以通过 MapReduce 视图创建针对 Hadoop 服务器的连接，并通过该连接对 HDFS 文件结构进行管理。

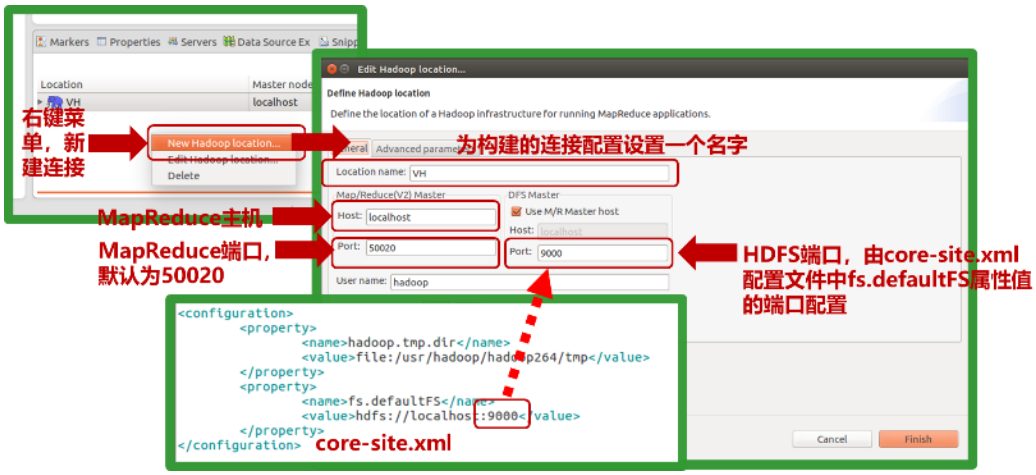
首先在 Eclipse 中打开【MapReduce Location】视图，菜单路径为 Windows -> Show View -> Other…：



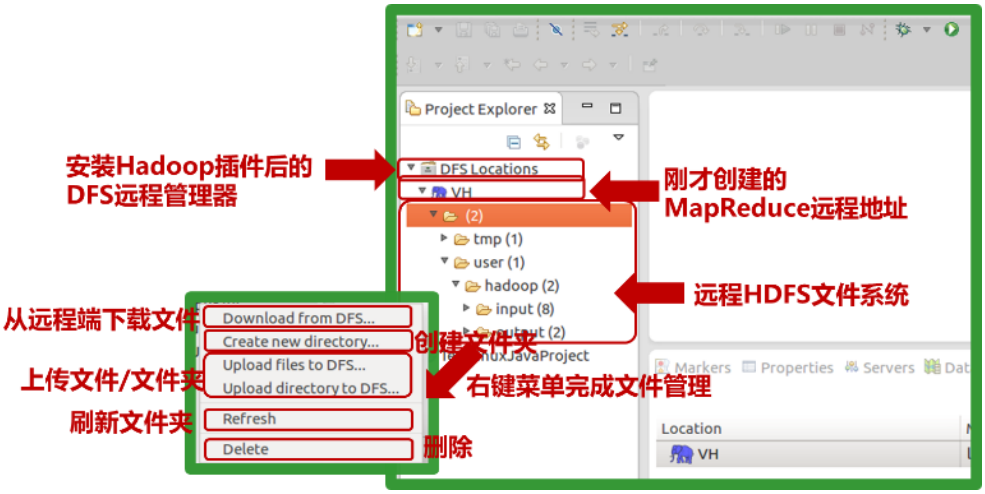
此时可以看到现在已经存在的 Hadoop 连接：



在该视图中也可以创建一个新的 Hadoop 连接：



配置远程 MapReduce 路径后即可通过 Eclipse 方便的查看、管理 Hadoop HDFS 文件：

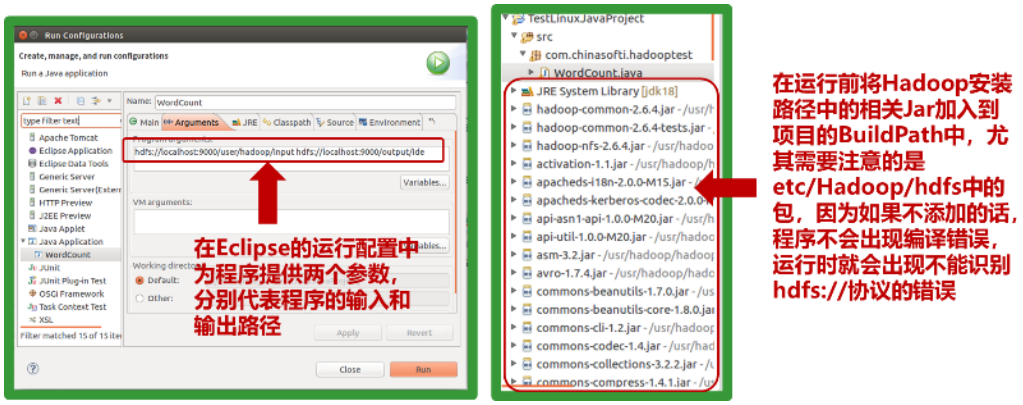


在正确安装配置 Eclipse 插件后，在新建项目对话框中即可找到【MapReduce Project】类型的项目，该项目类型实质上是一个普通的 Java 项目，和传统的 Java 项目不同是，采用这种方式构建项目后，Eclipse 将会自动将 Hadoop 客户端应用（主要是 MapReduce 客户端和 HDFS 客户端）所需要依赖的 Jar 包自动加入项目 Class Path，由于 Hadoop 发行包中依赖 Jar 包的位置路径分布比较复杂，Eclipse 的这一特性能够大大简化开发人员的项目构建工作。

通过 Eclipse 开发的 Hadoop MapReduce 程序有两种调度运行方法：

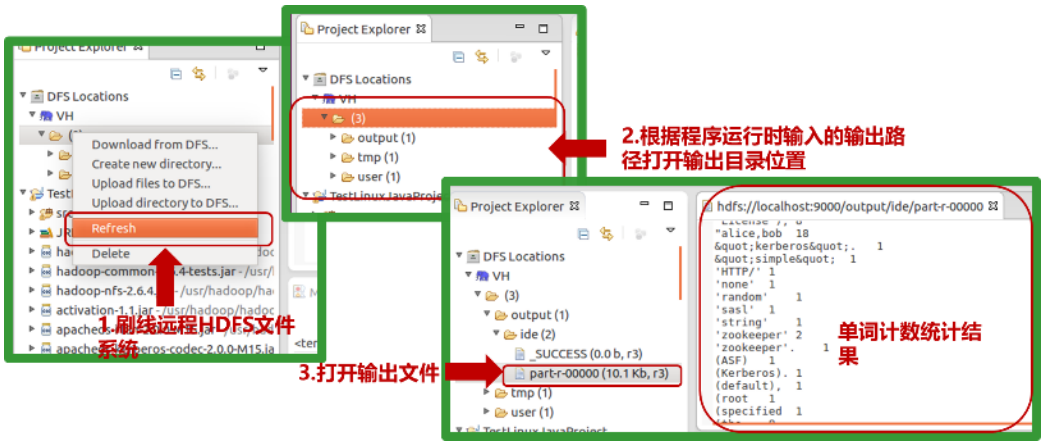
- Eclipse 端直接运行，远程访问 MapReduce
- 打包为 jar 文件，拷贝到 Hadoop 服务器执行

例如前面章节的 WordCount 程序可以直接在 Eclipse 中运行，远程访问 MapReduce，运行时需要为程序提供参数，代表程序的输入和输出。输入和输出不能以文件路径的方式存在，否则将会访问本地资源，应该使用’hdfs://主机:端口/路径’的方式提供:



运行结果如下：

程序执行成功后，可以通过 Eclipse 的 Hadoop 插件对应的视图直接查看程序执行时在 HDFS 上生成的结果文件：



程序运行时请确保输出路径不存在，否则程序将输出异常而不能正常执行（空文件夹也不行），因此如果再次运行的话需要将之前的运行结果删除：

在重复执行之前
先将之前生成的执行结果删除

需要注意的是，按照上述代码方式进行任务调度，默认情况下其真正执行时采用的是本地模式，即 MapReduce 完全没有在后台集群进行 MapReduce 任务，而是在执行代码的本机上完成所有任务指令，不过在执行指令前在 HDFS 上将任务需要的输入文件完整下载到本地临时目录，任务完成后再将结果上传至 HDFS。

在这种情况下会产生很大的问题：

- 数据量较大，在真正执行任务前，下载输入数据文件需要消耗过多的时间，影响任务的执行效率
- 数据量超大的情况下，本地主机无法正常加载所有的数据文件从而导致任务执行失败
- 没有实现分布式并发处理，失去了大数据平台存在的价值

直接运行后任务调度执行的控制台输出也能明确的看出这个问题，即在控制台输出的调试信息中可以看到任务的编号为：job_localXXXXXXXX，说明任务是本地执行。

如果需要完成真正的分布式任务调度，可以选择以下两个方案：

- 将任务代码编译后打包成 jar 文件，发送到 Hadoop NameNode，通过 Hadoop 命令行的形式执行任务，这将在接下来的内容中介绍。

- 将任务代码编译后打包成 jar 文件，在代码中明确任务所需要的 Jar 包，该包将会被自动上传到 HDFS 的临时目录并被当前任务关联的所有 MapReduce Datanode 下载执行：

```
conf.set("mapreduce.job.jar", "myjob.jar");
//此处代码，一定放在 Job 任务前面，否则会报类找不到的异常
```

也可在代码中实现自动的代码打包工具：

```
package com.chinasofti.hadoop.jobutil;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.jar.Manifest;

public class PackageUtil {

    /**
     * 在"java.io.tmpdir"指定的临时目录中创建一个临时的jar文件
     *
     * @param root
     *            需要创建的jar包对应的目录，如"bin"一般指本项目的
编译结果根目录
     * @return 指向该临时jar文件的File对象
     * @throws IOException
     */
    public static File createTempJar(String root) throws
IOException {
        if (!new File(root).exists()) {
            return null;
        }
        Manifest manifest = new Manifest();
        manifest.getMainAttributes().putValue("Manifest-Version",
"1.0");
        final File jarFile = File.createTempFile("MRbyChinasofti-",
".jar",
                new File(System.getProperty("java.io.tmpdir")));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
```

```
                jarFile.delete();
            }
        });

        JarOutputStream out = new JarOutputStream(
                new FileOutputStream(jarFile), manifest);
        createTempJarInner(out, new File(root), "");
        out.flush();
        out.close();
        return jarFile;
    }

    private static void createTempJarInner(JarOutputStream out,
File f,
            String base) throws IOException {
        if (f.isDirectory()) {
            File[] fl = f.listFiles();
            if (base.length() > 0) {
                base = base + "/";
            }
            for (int i = 0; i < fl.length; i++) {
                createTempJarInner(out, fl[i], base +
fl[i].getName());
            }
        } else {
            out.putNextEntry(new JarEntry(base));
            FileInputStream in = new FileInputStream(f);
            byte[] buffer = new byte[1024];
            int n = in.read(buffer);
            while (n != -1) {
                out.write(buffer, 0, n);
                n = in.read(buffer);
            }
            in.close();
        }
    }


}
```

然后利用自动化打包后调度执行任务：

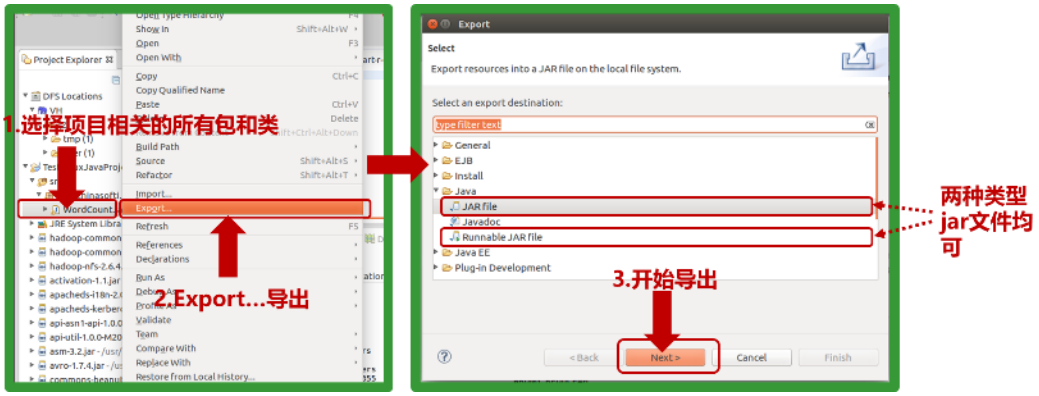
```
File mpjar=PackageUtil.createTempJar("bin");
```

```
conf.set("mapreduce.job.jar", mpjar.getPath());
```

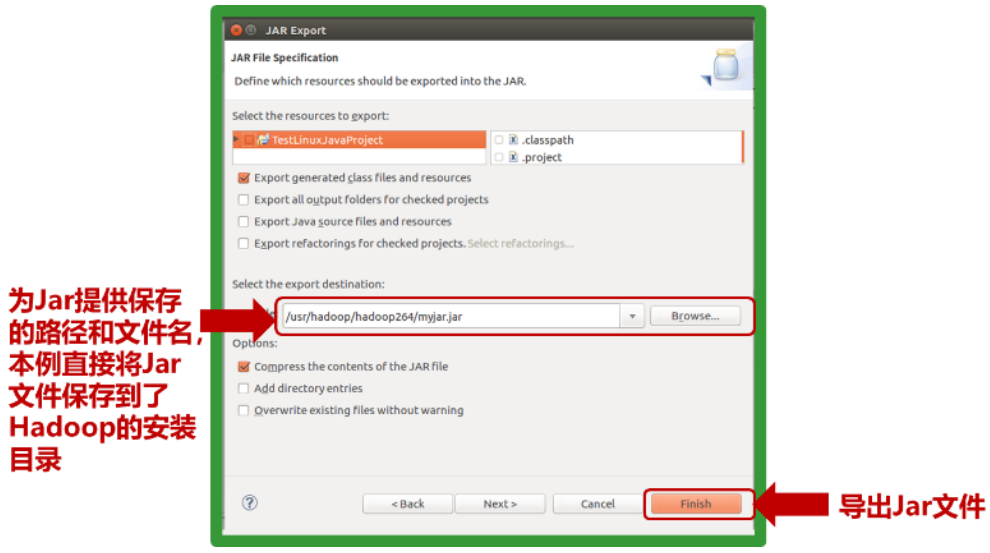
如果任务中涉及到其他第三方类库 jar 包，也需要将 jar 包上传到 HDFS（不同的环境读取的路径不同，可以先不上传，通过 Debug 的错误提示找到 jar 包应该上传的路径）：

```
conf.set("tmpjars","jedis.jar,common-pool-1.6.jar");
//任务需要使用的第三方jar包，jar文件用逗号隔开，可以通过设置后DEBUG的输出找到在HDFS上读取jar包的位置然后手动上传

//也可以利用分布式缓存，如DistributedCache.addArchiveToClassPath(path, job)，此处的path必须是hdfs，即自己讲jar上传到hdfs上，然后将路径加入到分布式缓存中；
```

接下来介绍通过将任务代码打包为 Jar 并上传到 Hadoop 服务器执行的方式。Hadoop 并不会读取 Jar 包的属性文件以查找主类，而是在运行任务 jar 包的时候通过参数显式指出，因此打包的时候选择 JAR 或 Runnable JARfile 均可：



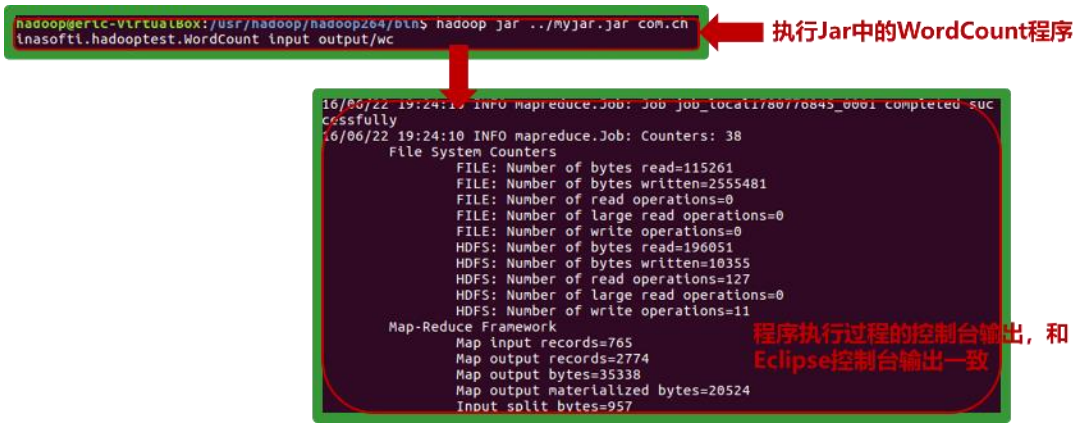
为 Jar 提供路径和文件名后直接导出即可：



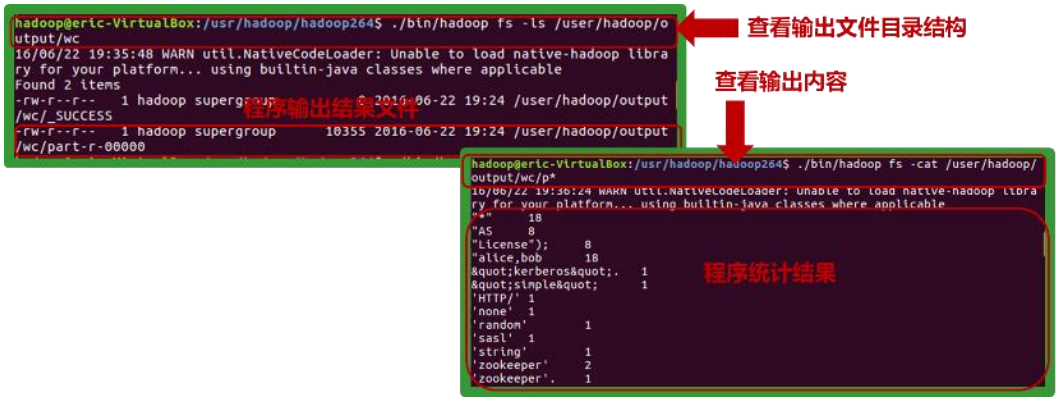
导出 jar 包后将其通过网络工具传输到 Hadoop 服务器主节点中，然后就可

以直接通过命令行执行其中包含的 MapReduce 任务了。执行 Jar 程序的指令格式是：

```
Hadoop jar jar路径 程序入口类名 输入 输出
```

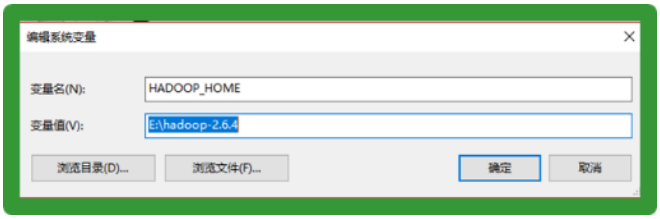


使用 Hadoop shell 的 ls 命令可以查看文件系统结构，cat 命令可以查看结果内容：

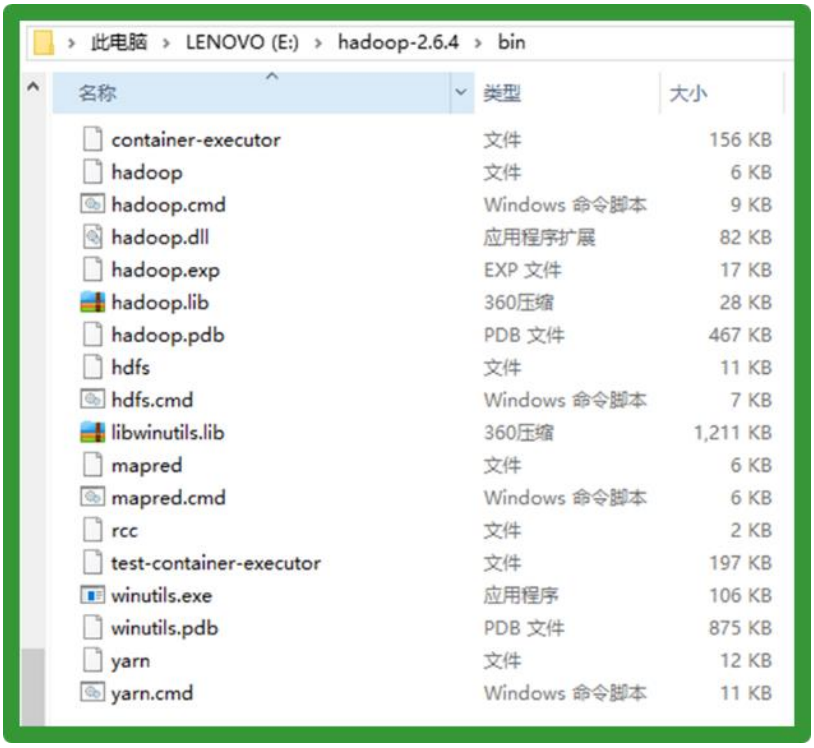


如果由于开发环境的限制，必须要在 Windows 中搭建 Hadoop 的开发环境，则首先需要确认 Windows 版本为 Win8/Win10 64 位版本。在 Windows7 的设备上，通过插件能够管理远程 Hadoop HDFS 文件系统中的文件，但不能正常通过 API 代码进行 MapReduce 任务调度。

修改 Windows 系统的主机，添加远程 Hadoop 集群的主节点和从节点的 IP 及主机名映射列表，Windows 的 hosts 文件位于：C:\Windows\System32\drivers\etc\hosts。然后在 windows 系统环境变量中创建新的环境变量：HADOOP_HOME，该变量的值为 Hadoop 发行包在 Windows 中解压的目录，Hadoop 插件在连接服务时需要使用 Hadoop 发行包中的二进制文件，如将 Hadoop 解压到了 E:\hadoop-2.6.4 文件夹中，则环境变量应该如下设置：



下载 Hadoop 的 windows dll 包，将其中所有文件解压到 hadoop 解压目录的 bin 子文件夹中，成功后 HADOOP_HOM\bin 的目录结构如下：



最后按照 Linux 平台同样的方式配置 Eclipse 和 Hadoop 插件即可。

第5节 Hadoop 集群配置

伪分布式的 Hadoop 集群功能和分布式集群一致，能够很好的承担开发测试任务，但如果要进行长时间的功能验证测试，还是需要搭建一个更接近生产环境的分布式集群，因为伪分布式环境中，Datanode 只有一个，因此 HDFS 的数据块副本数量只能设置为 1，一旦该数据节点出现存储异常，整个系统就无法从软件系统层面保障系统数据的高可用性。

Hadoop 的主节点通过 SSH 对整个集群中的所有其他节点进行调度，包括节点本地服务的启动和停止等动作。因此对集群内部节点有以下几个要求：

- 为了保证网络数据交换的正常进行，因此集群中每个节点的 hosts 文件均需要包含本集群中所有节点的主机名和 IP 地址的映射关系。
- Hadoop 主节点中使用 shell 脚本通过 SSH 对所有从节点中的服务进行控制，服务调度脚本通过获取自身所在路径的方式确定从节点中服务脚本

所在的位置，因此所有节点中 Hadoop 组件安装的路径需要保持一致。

- 集群中所有节点操作 Hadoop 组件的用户和角色需要保持一致。
- 由于主节点需要通过 SSH 调度从节点的服务脚本，因此主节点需要能够免密访问所以的从节点。

基于以上要求，如果使用虚拟机搭建 Hadoop 集群，建议在主节点中把所有的组件正确配置（包括整个集群中节点的主机名/ip 地址映射）后，克隆虚拟机作为从节点使用。

搭建 Hadoop 集群的基本步骤如下：

首先规划 Hadoop 集群的结构，建议在开发测试环境中使用一主 2 从结构（主节点也能作为运算节点存在，在这种结构下，一共存在 3 个运算节点），如：

主机名	IP 地址	角色
ICSS_BDES_Master	192.168.1.101	主节点/从节点
ICSS_BDES_S1	192.168.1.102	从节点
ICSS_BDES_S2	192.168.1.103	从节点

根据规划在已经搭建好伪分布式环境的主节点的 hosts 文件中添加所有节点的主机名-IP 地址的映射关系。

如果使用虚拟机搭建集群环境，则将主控节点克隆 2 次作为从节点虚拟机，如使用物理机器搭建集群环境，则额外准备 2 台主机，按照主控节点的配置方式（路径等保持一致）进行配置。然后分别按照集群规划，正确修改两个从节点的主机名和 IP 地址。

为了保证主节点能够免密访问两个从节点，将主节点的 SSH 免密访问签名信息复制到两台从节点主机，在主节点的终端中执行 ssh-copy-id 命令，将本主机的密钥信息拷贝到目标节点：

```
ssh-copy-id ICSS_BDES_S1

ssh-copy-id ICSS_BDES_S2
```

接下来修改主节点中 Hadoop 的配置文件 hdfs-site.xml，设置 Hadoop 数据高可用性的副本数，副本数量不能大于从节点的数量，由于目前实际存在 3 个计算节点，因此可以将该值设置为默认的 3，即每个上传到 HDFS 的文件拆分成的数据块都将保存 3 个副本，用冗余的方式来保障数据的高可用性。

修改后的 replication 属性如下：

```
<property>

        <name>dfs.replication</name>

        <value>3</value>

</property>
```

最后修改主节点中 Hadoop 的配置文件，设置 Hadoop 的从节点列表，该配置文件位于 HADOOP_HOME/etc/hadoop/slaves，该配置文件是一个标准的文本文件，每一行列举一个从节点的主机名，如按照上述规划，该文件的内容如下：

```
ICSS_BDES_S1
ICSS_BDES_S2
```

所有配置成功后，删除 Hadoop 的所有临时文件（临时文件位置在 Hadoop 的 core-site.xml 配置文件中规定），在主节点中重新执行格式化命令

```
hadoop namenode -format
```

然后在主节点中启动 HDFS 服务：

```
start-dfs.sh
```

此时再次访问 HDFS 的 WEB 端界面，可以看到活动的 datanode 已经变成了 3 个，说明集群配置成功。

第3章 HDFS

第1节 HDFS 简介

当数据集的大小超过一台独立的物理计算机的存储能力时，就有必要对它进行分区并存储到若干台单独的计算机上。管理网络中跨多台计算机存储的文件系统称为分布式文件系统（Distributed Filesystem）。该系统架构于网络之上，势必会引入网络编程的复杂性，因此分布式文件系统比普通磁盘文件系统更为复杂。例如，使文件系统能够容忍节点故障且不丢失任何数据，就是一个极大的挑战。

Hadoop 有一个称为 HDFS 分布式系统，即 Hadoop Distributed Filesystem。在非正式文档或旧文档以及配置文件中，有时也简称为 DFS，它们是同一个概念。

HDFS 是 Hadoop 的旗舰级文件系统，但实际上 Hadoop 是一个综合性的文件系统抽象，因此也能将 Hadoop 集成其他文件系统（如本地文件系统和 AmazonS3 系统）。

HDFS 被设计成适合运行在通用硬件上的分布式文件系统。它和其他的分布式文件系统有很多共同点。但同时，它和其他的分布式文件系统的区别也是很明显的。HDFS 是一个高度容错性的系统，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS 放宽了一部分 POSIX 约束，来实现流式读取文件系统数据的目的。HDFS 在最开始是作为 Apache Nutch 搜索引擎项目的基础架构而开发的。HDFS 是 Apache Hadoop Core 项目的一部分。

HDFS 特性如下：

- **硬件故障**

硬件故障是常态，而不是异常。整个 HDFS 系统将由数百或数千个存储着文件数据片段的服务器组成。实际上它里面有非常巨大的组成部分，每一个组成部分都很可能出现故障，这就意味着 HDFS 里的总是有一些部件是失效的，因此，故障的检测和自动快速恢复是 HDFS 一个很核心的设计目标。

- **数据访问**

运行在 HDFS 之上的应用程序必须流式地访问它们的数据集，它不是运行在普通文件系统之上的普通程序。HDFS 被设计成适合批量处理的，而不是用户交互式的。重点是在数据吞吐量，而不是数据访问的反应时间，POSIX 的很多硬性需求对于 HDFS 应用都是非必须的，去掉 POSIX 一小部分关键语义可以获得更好的数据吞吐率。

- **大数据集**

运行在 HDFS 之上的程序有很大量的数据集。典型的 HDFS 文件大小是 GB 到 TB 的级别。所以，HDFS 被调整成支持大文件。它应该提供很高的聚合数据带宽，一个集群中支持数百个节点，一个集群中还应该支持千万级别的文件。

- **简单一致性模型**

大部分的 HDFS 程序对文件操作需要的是一次写多次读取的操作模式。一个文件一旦创建、写入、关闭之后就不需要修改了。这个假定简单化了数据一致的问题，并使高吞吐量的数据访问变得可能。一个 MapReduce 程序或者网络爬虫程序都可以完美地适合这个模型。

- **移动计算比移动数据更经济**

在靠近计算数据所存储的位置来进行计算是最理想的状态，尤其是在数据集特别巨大的时候。这样消除了网络的拥堵，提高了系统的整体吞吐量。一个假定就是迁移计算到离数据更近的位置比将数据移动到程序运行更近的位置要更好。HDFS 提供了接口，来让程序将自己移动到离数据存储更近的位置。

- **异构软硬件平台间的可移植性**

HDFS 被设计成可以简便地实现平台间的迁移，这将推动需要大数据集的应用更广泛地采用 HDFS 作为平台。

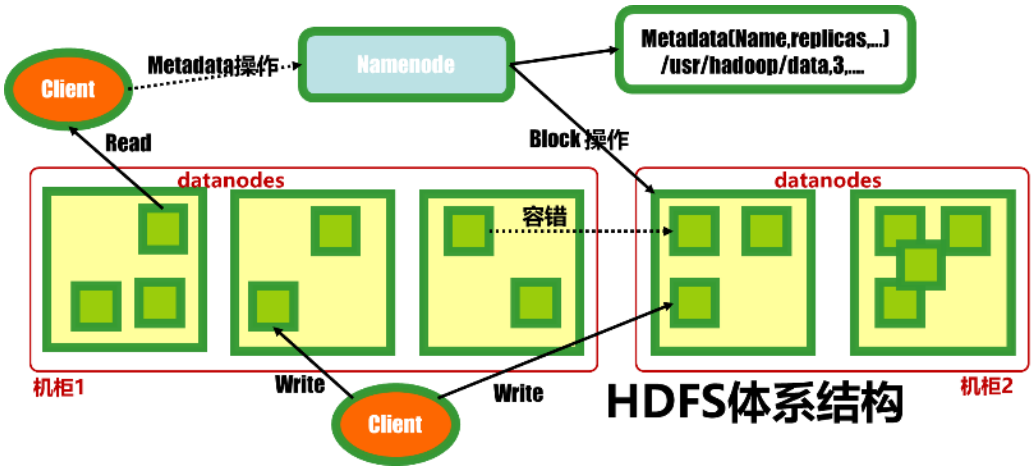
- **名字节点和数据节点**

HDFS 是一个主从结构，一个 HDFS 集群是由一个名字节点，它是一个管理文件命名空间和调节客户端访问文件的主服务器，当然还有一些数据节点，通常是一个节点一个机器，它来管理对应节点的存储。HDFS 对外开放文件命名空间并允许用户数据以文件形式存储。

内部机制是将一个文件分割成一个或多个块，这些块被存储在一组数据节点中。名字节点用来操作文件命名空间的文件或目录操作，如打开，关闭，重命名等等。它同时确定块与数据节点的映射。数据节点负责来自文件系统客户的读写请求。数据节点同时还要执行块的创建，删除，和来自名字节点的块复制指令。

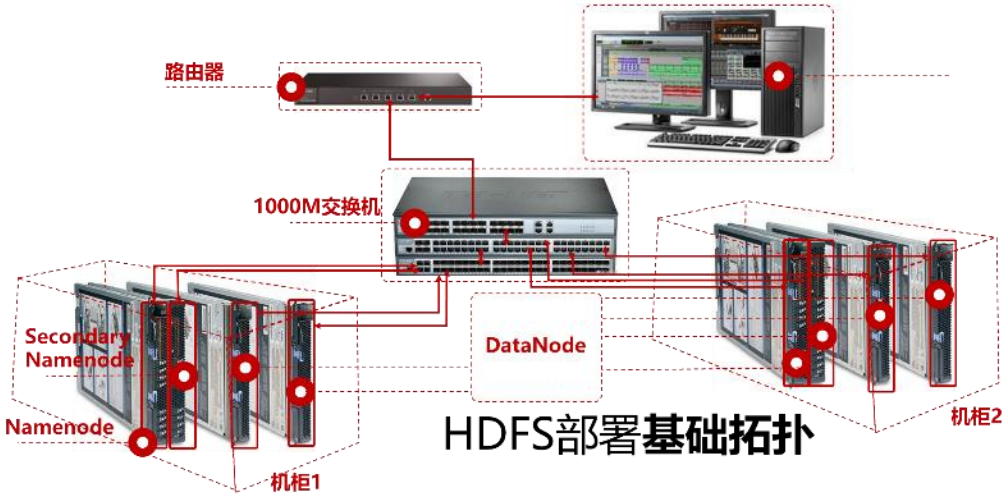
名字节点和数据节点都是运行在普通的机器之上的软件，机器典型的都是 GNU/Linux，HDFS 是用 Java 编写的，任何支持 Java 的机器都可以运行名字节点或数据节点，利用 Java 语言的超轻便性，很容易将 HDFS 部署到大范围的机器上。典型的部署是由一个专门的机器来运行名字节点软件，集群中的其他每台机器运行一个数据节点实例。体系结构不排斥在一个机器上运行多个数据节点的实例，但是实际的部署不会有这种情况。

HDFS 的体系结构如下：



集群中只有一个名字节点极大地简单化了系统的体系结构。名字节点是仲裁者和所有 HDFS 元数据的仓库，用户的实际数据不经过名字节点。

HDFS 的网络拓扑结构如下：



HDFS 适用于一次写入多次查询的情况，因此在非必要情况下，需要重点关注 HDFS 的限制：

上面的内容提到了，HDFS 以流式数据访问模式来存储超大文件，运行于商用硬件集群上。

超大文件

“超大文件”在这里指有几百 MB、几百 GB 甚至几百 TB 大小的文件。目前已经有存储 PB 级数据的 Hadoop 集群了

流式数据访问

HDFS 的构建思路是这样的：一次写入、多次读取是最高效的访问模式。数据集通常由数据源生成或从数据源复制而来，接着长时间在此数据集上进行各种分析。每次分析都将涉及该数据集的大部分数据甚至全部，因此读取整个数据集的时间延迟比读取第一条记录的时间延迟更重要。

商用硬件

Hadoop 并不需要运行在昂贵且高可靠的硬件上。它是设计运行在商用硬件（在各种零售店都能买到的普通硬件）的集群上的，因此至少对于庞大的集群来说，节点故障的几率还是非常高的。HDFS 遇到上述故障时，被设计成能够继续运行且不让用户察觉到明显的中断。

同样，那些不适合在 HDFS 上运行的应用也值得研究。目前某些应用领域并不适合在 HDFS 上运行，不过以后可能会有所改进。

不适合低时间延迟的数据访问

要求低时间延迟数据访问的应用，例如几十毫秒范围，不适合在 HDFS 上运行。记住，HDFS 是为高数据吞吐量应用优化的，这可能会以提高时间延迟为代价。目前，对于低延迟的访问需求，HBase 是更好的选择。

不适合大量的小文件的环境

由于 namenode 将文件系统的元数据存储在内存中，因此该文件系统所能存储的文件总数受限于 namenode 的内存容量。根据经验，每个文件、目录和数据块的存储信息大约占 150 字节。因此，举例来说，如果有一百万个文件，且每个文件占一个数据块，那至少需要 300MB 的内存。尽管存储上百万个文件是可行的，但是存储数十亿个文件就超出当前硬件的能力了。在目前的应用环境中，如果要将 HDFS 作为海量静态文件服务器，应该使用 Hadoop 的归档文件（后续章节会详细介绍）结合反向代理实现。

不适合多用户写入，任意修改文件

HDFS 中的文件可能只有一个 Writer,而且写操作总是将数据添加在文件的末尾。它不支持其有多个写入者的操作，也不支持在文件的任意位置进行修改。可能以后会支持这些操作，但它们相对比较低效。

第2节 HDFS 的数据分块

每个磁盘都有默认的数据块大小，这是磁盘进行数据读 / 写的最小单位。构

建于单个磁盘之上的文件系统通过磁盘块来管理该文件系统中的块，该文件系统块的大小可以是磁盘块的整数倍。文件系统块一般为几千字节，而磁盘块一般为512 字节。文件系统块大小对于需要读 / 写文件的文件系统用户来说是透明的。HDFS 同样也有块（Block）的概念，但是大得多，早期 Hadoop 默认为 64MB，而 Hadoop2 已经提高为默认 128M。与单一磁盘上的文件系统相似，HDFS 上的文件也被划分为块大小的多个分块，作为独立的存储单元。但与其他文件系统不同的是，HDFS 中小于一个块大小的文件不会占据整个块的空间。

为什么 HDFS 中的分块如此之大?

HDFS 的块比磁盘的块大，其目的是为了最小化寻址开销。如果块设里得足够大，从磁盘传输数据的时间会明显大于定位这个块开始位置所需的时间。因而，传愉一个由多个块组成的文件的时间取决于磁盘传愉速率。我们来做一个速算，如果寻址时间约为 10ms，而传输速率为 100MB/s，为了使寻址时间仅占传输时间的 1%，我们要将块大小设置约为 100MB. 默认的块大小实际为 64MB，但是很多情况下 HDFS 使用 128MB 的块设置。以后随着新一代磁盘驱动器传输速率的提升，块的大小将被设定得更大。但是这个数也不会设置得过大。MapReduce 中的 Map 任务通常一次只处理一个块中的数据，因此如果任务数太少（少于集群中的节点数量），作业的运行速度就会比较慢。

对分布式文件系统中的块进行抽象会带来很多好处。

第一个最明显的好处是，一个文件的大小可以大于网络中任意一个磁盘的容量。文件的所有块并不需要存储在同一个磁盘上，因此它们可以利用集群上的任意一个磁盘进行存储。事实上，尽管不常见，但对于整个 HDFS 集群而言，也可以仅存储一个文件，该文件的块占满集群中所有的磁盘。

第二个好处是，使用抽象块而非整个文件作为存储单元，大大简化了存储子系统的设计。简化是所有系统的目标，但是这对于故障种类繁多的分布式系统来说尤为重要。将存储子系统控制单元设置为块，可简化存储管理（由于块的大小是固定的，因此计算单个磁盘能存储多少个块就相对容易）。同时也消除了对元数据的顾虑（块只是存储数据的一部分，而文件的元数据，如权限信息，并不需要与块一同存储，这样一来，其他系统就可以单独管理这些元数据）。不仅如此，块还非常适合用于数据备份进而提供数据容错能力和提高可用性。将每个块复制到少数几个独立的机器上（默认为 3 个），可以确保在块、磁盘或机器发生故障后数据不会丢失。如果发现一个块不可用，系统会从其他地方读取另一个复本，而这个过程对用户是透明的。一个因损坏或机器故障而丢失的块可以从其他候选地点复制到另一台可以正常运行的机器上，以保证复本的数量回到正常水平。同样，有些应用程序可能选择为一些常用的文件块设置更高的复本数量进而分散集群中的读取负载。

第3节 Namenode 和 Datanode

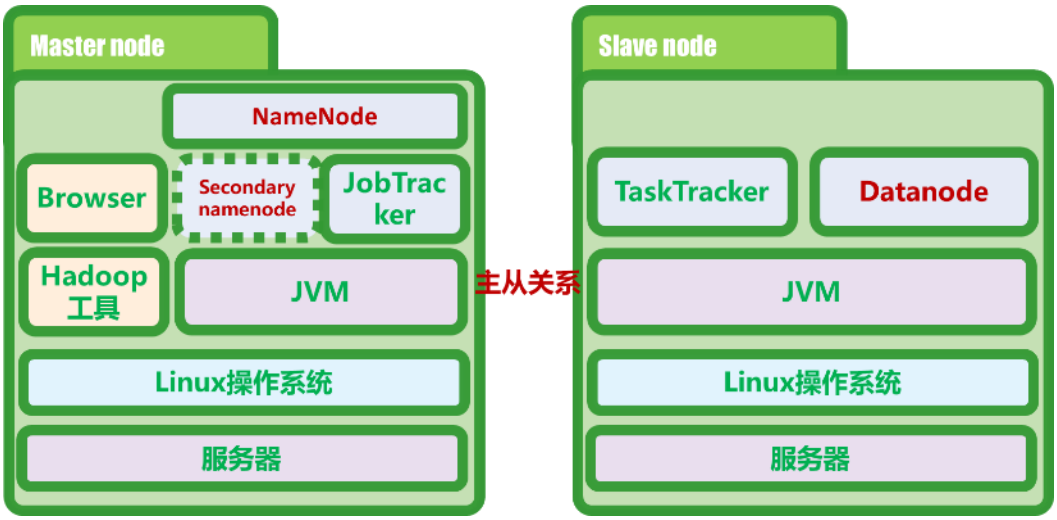
HDFS 集群有两类节点以管理者-工作者模式运行，即一个 namenode（管理者）和多个 datanode（工作者）。namenode 管理文件系统的命名空间。它维护着文件系统树及整棵树内所有的文件和目录。这些信息以两个文件形式永久保存在本地

磁盘上：命名空间镜像文件和编辑日志文件。namenode 也记录着每个文件中各个块所在的数据节点信息，但它并不永久保存块的位置信息，因为这些信息会在系统启动时由数据节点重建。

客户端（client）代表用户通过与 namenode 和 datanode 交互访问整个文件系统。客户端提供一个类似于 POSIX（可移植操作系统界面）的文件系统接口，因此用户在编程时无需知道 namenode 和 datanode 也可实现其功能。

datanode 是文件系统的工作节点。它们根据需要存储并检索数据块（受客户端或 namenode 调度），并且定期向 namenode 发送它们所存储的块的列表。

HDFS 的单节点结构如下：



节点的核心功能如下：

namenode（元数据节点）-管理系统命名空间：

- 其将所有的文件和文件夹的元数据保存在一个文件系统树中。
- 这些信息也会在硬盘上保存成以下文件：命名空间镜像及修改日志
- 其还保存了一个文件包括哪些数据块，分布在哪些数据节点上。然而这些信息并不存储在硬盘上，而是在系统启动的时候从数据节点收集而成的。

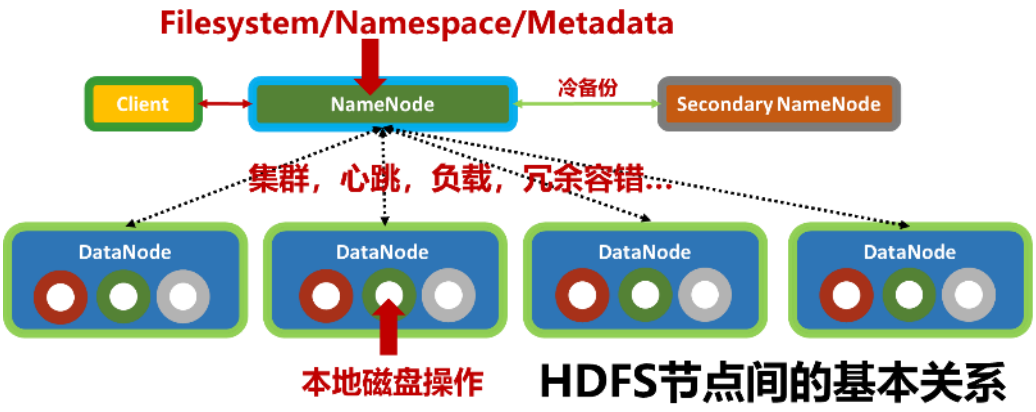
datanode（数据节点）-文件系统中真正存储数据的地方：

- 客户端或者元数据信息可以向数据节点请求写入或者读出数据块。
- 其周期性的向元数据节点回报其存储的数据块元数据信息。

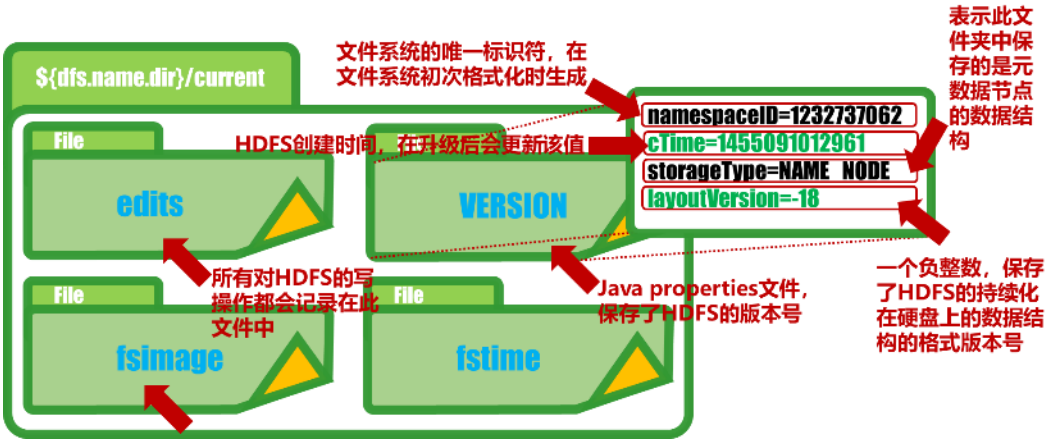
从元数据节点（secondary namenode） -namenode 冷备份：

- 并不是元数据节点出现问题时候的备用节点，它和元数据节点负责不同的事情。
- 其主要功能就是周期性将元数据节点的命名空间镜像文件和修改日志合并，以防日志文件过大。合并过后的命名空间镜像文件也在从元数据节

点保存了一份，以防元数据节点失败的时候，可以恢复。



Namenode 在集群中扮演了至关重要的角色。它的节点组成结构如下：



没有 namenode，文件系统将无法使用。事实上，如果运行 namenode 服务的机器毁坏，文件系统上所有的文件将会丢失，因为在这种情况下，系统完全不知道如何根据 datanode 的块重建文件。因此，对 namenode 实现容错非常重要，Hadoop 为此提供两种机制。

第一种机制是备份那些组成文件系统元数据持久状态的文件。Hadoop 可以通过配置使 namenode 在多个文件系统上保存元数据的持久状态。这些写操作是实时同步的，是原子操作。一般的配置是，将持久状态写入本地磁盘的同时，写入一个远程挂载的网络文件系统（NFS）。

另一种更常用的就是之前提到的 namenode 冷备份。

热备份：

b 是 a 的热备份，如果 a 坏掉。那么 b 马上运行代替 a 的工作。

冷备份：

b 是 a 的冷备份，如果 a 坏掉。那么 b 不能马上代替 a 工作。但是 b 上存储 a 的一些信息，减少 a 坏掉之后的损失。

在 HDFS 集群中，namenode 冷备份主要涉及两个主要文件核心：fsimage 和

edits。利用这两个元素进行备份的策略更类似于传统 RDBMS 的增量备份策略：在某一个特定时间点上完成数据库的完整备份，其后的时间段内保存数据的操作日志（记录了所有对数据库的修改），通过基础的完整备份+后续的日志即可完整恢复数据库的内容。

在 HDFS 中，fsimage 保存了元数据镜像文件（文件系统的目录树），而 edits 中完整记录了元数据的操作日志（针对文件系统做的修改操作记录）。namenode 内存中存储的元数据可以用 fsimage+edits 来表达。而 SecondaryNameNode 负责定时默认 1 小时，从 namenode 上，获取 fsimage 和 edits 来进行合并，然后再发送给 namenode。减少 namenode 的工作量。这是 HA（高可用性）的一个常用解决方案，但不支持热备。要使用这种方式来完成高可用性，直接修改配置即可。默认情况下在它和 namenode 在同一个节点上，但这样不安全

SeecondaryNamenode 的冷备份流程描述如下：

当文件系统客户端进行写操作时，首先把它记录在修改日志中(edit log)。

元数据节点在内存中保存了文件系统的元数据信息。在记录了修改日志后，元数据节点则修改内存中的数据结构。

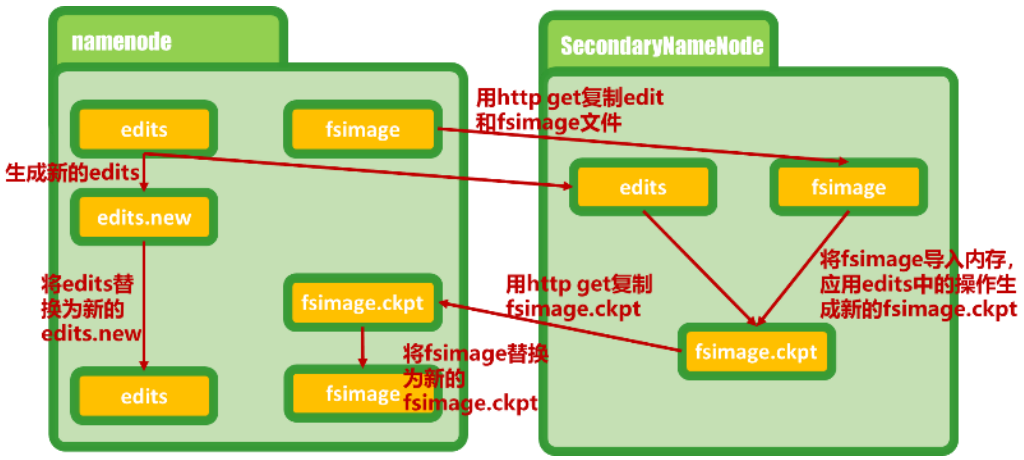
每次的写操作成功之前，修改日志都会同步(sync)到文件系统。

fsimage 文件，也即命名空间映像文件，是内存中的元数据在硬盘上的 checkpoint，它是一种序列化的格式，并不能够在硬盘上直接修改。

同数据的机制相似，当元数据节点失败时，则最新 checkpoint 的元数据信息从 fsimage 加载到内存中，然后逐一重新执行修改日志中的操作。

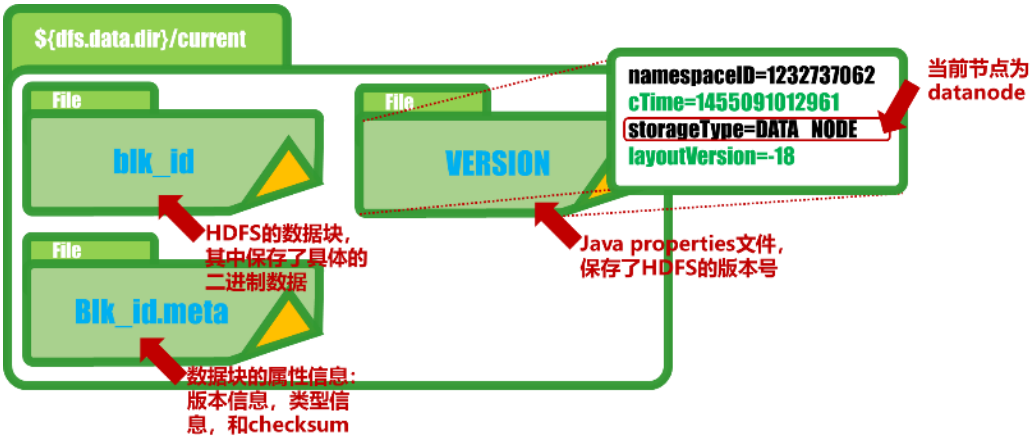
从元数据节点就是用来帮助元数据节点将内存中的元数据信息 checkpoint 到硬盘上的。HDFS 会定期（dfs.namenode.checkpoint.period，默认 3600 秒）的对最近的 fsimage 和一批新 edits 文件进行 Checkpoint（也可以手工命令方式），Checkpoint 发生后会将前一次 Checkpoint 后的所有 edits 文件合并到新的 fsimage 中，HDFS 会保存最近两次 checkpoint 的 fsimage。Namenode 启动时会把最新的 fsimage 加载到内存中。

checkpoint 的过程如下：



- 通知元数据节点生成新的日志文件，以后的日志都写到新的日志文件中。
- 从元数据节点用 http get 从元数据节点获得 fsimage 文件及旧的日志文件。
- 从元数据节点将 fsimage 文件加载到内存中，并执行日志文件中的操作，然后生成新的 fsimage 文件。
- 从元数据节点将新的 fsimage 文件用 http post 传回元数据节点
- 元数据节点可以将旧的 fsimage 文件及旧的日志文件，换为新的 fsimage 文件和新的日志文件(第一步生成的)，然后更新 fstime 文件，写入此次 checkpoint 的时间。
- 元数据节点中的 fsimage 文件保存了最新的 checkpoint 的元数据信息，日志文件也重新开始，不会变的很大了。

Datanode 用于存放实际的数据分块文件，节点结构如下：



由于存在 Namenode 和 Datanode 的主从结构，因此使用 HDFS 读写数据时的基本流程如下：

文件写入：

- Client 向 NameNode 发起文件写入的请求。

- NameNode 根据文件大小和文件块配置情况，返回给 Client 它所管理部分 DataNode 的信息。
- Client 将文件划分为多个 Block，根据 DataNode 的地址信息，按顺序写入到每一个 DataNode 块中。

文件读取：

- Client 向 NameNode 发起文件读取的请求。
- NameNode 返回文件存储的 DataNode 的信息。
- Client 读取文件信息。

后续的章节还将针对这两个流程进行更详细的讲解。

第4节 HDFS 的缺点和改进

HDFS 不适合低延迟数据访问。

如果要处理一些用户要求时间比较短的低延迟应用请求，则 HDFS 不适合。HDFS 是为了处理大型数据集分析任务的，主要是为达到高的数据吞吐量而设计的，这就可能要求以高延迟作为代价。

对于那些有低延时要求的应用程序，HBase 是一个更好的选择。通过上层数据管理项目来尽可能地弥补这个不足。在性能上有了很大的提升，它的口号就是 goes real time。使用缓存或多 master 设计可以降低 client 的数据请求压力，以减少延时。还有就是对 HDFS 系统内部的修改，这就得权衡大吞吐量与低延时了，HDFS 不是万能的银弹。

HDFS 无法高效存储大量小文件。

因为 Namenode 把文件系统的元数据放置在内存中，所以文件系统所能容纳的文件数目是由 Namenode 的内存大小来决定。一般来说，每一个文件、文件夹和 Block 需要占据 150 字节左右的空间，所以，如果你有 100 万个文件，每一个占据一个 Block，你就至少需要 300MB 内存。当前来说，数百万的文件还是可行的，当扩展到数十亿时，对于当前的硬件水平来说就没法实现了。还有一个问题就是，因为 Map task 的数量是由 splits 来决定的，所以用 MR 处理大量的小文件时，就会产生过多的 Maptask，线程管理开销将会增加作业时间。举个例子，处理 10000M 的文件，若每个 split 为 1M，那就会有 10000 个 Maptasks，会有很大的线程开销；若每个 split 为 100M，则只有 100 个 Maptasks，每个 Maptask 将会有更多的事情做，而线程的管理开销也将减小很多。

利用 SequenceFile、MapFile、Har 等方式归档小文件，这个方法的原理就是把小文件归档起来管理，HBase 就是基于此的。对于这种方法，如果想找回原来的小文件内容，那就必须得知道与归档文件的映射关系。

横向扩展，一个 Hadoop 集群能管理的小文件有限，那就把几个 Hadoop 集群拖在一个虚拟服务器后面，形成一个大的 Hadoop 集群。google 也是这么干过的。

多 Master 设计，这个作用显而易见了。正在研发中的 GFS II 也要改为分布式多 Master 设计，还支持 Master 的 Failover，而且 Block 大小改为 1M，有意识要调优处理小文件。

Alibaba DFS 的设计，也是多 Master 设计，它把 Metadata 的映射存储和管理分开了，由多个 Metadata 存储节点和一个查询 Master 节点组成。

第5节 HDFS 的 Shell 接口

HDFS 提供了一套完善的 Shell 接口来对分布式文件系统中的文件进行管理。为了方便传统的 Linux 运维人员能够快速上手，HDFS 的文件管理 Shell 命令最大程度上保持了和原生 Linux 文件操作命令的一致性。

通过 HDFS Shell 命令可以对文件系统进行一下操作：读取文件；新建目录；移动文件；删除数据；列出目录；上传下载等。

HDFS 的 Shell 命令通用格式如下：

```
$HADOOP_HOME/bin/hadoop fs 或$HADOOP_HOME/bin/hadoop dfs
```

fs 是个比较抽象的层面，在分布式环境中，fs 就是 dfs，但在本地环境中，fs 是 local file system，这个时候 dfs 就不能用。所有的 FS shell 命令使用 URI 路径作为参数。URI 格式：

```
scheme://authority/path
```

HDFS 的 scheme 是 hdfs，对本地文件系统，scheme 是 file。其中 scheme 和 authority 参数都是可选的，如果未加指定，就会使用配置中指定的默认 scheme。例如：/parent/child 可以表示成 hdfs://namenode:namenodePort/parent/child，或者更简单的/parent/child（假设配置文件 core-site.xml 中配置的是 namenode:namenodePort）。

大多数 FS Shell 命令的行为和对应的 Unix Shell 命令类似。

针对文件和目录，HDFS 的权限模式与 POSIX 非常相似。一共提供三类权限模式：只读权限（r）、写入权限和可执行权限（x）。读取文件或列出目录内容时需要只读权限，写入一个文件或是在一个目录上新建及删除文件或目录，需要写入权限。对于文件而言，可执行权限可以忽略，因为你不能在 HDFS 中执行文件（与 POSlX 不同），但在访问一个目录的子项时需要该权限。

每个文件和目录都有所属用户（owner）、所属组别（group）及模式（mode）。这个模式是由所属用户的权限、组内成员的权限及其他用户的权限组成的。

在默认情况下，可以通过正在运行进程的用户名和组名来唯一确定客户端的标识. 但由于客户端是远程的，任何用户都可以简单地在远程系统上以其名义新建一个账户来进行访问。因此，作为共享文件系统资源和防止数据意外损失的一种机制，权限只能供合作团体中的用户使用，而不能用于在一个不友好的环境中

保护资源。注意，最新版的 Hadoop 已经支持 Kerberos 用户认证，该认证去除了这些限制。但是，除了上述限制之外，为防止用户或自动工具及程序意外修改或删除文件系统的重要部分，启用权限控制还是很重要的（这也是默认的配置：dfs.permissions 属性）。如果在开发测试环境中希望简化控制操作，可以将 HDFS 的权限控制属性关闭，即在 Hadoop 的配置文件 hdfs-site.xml 中加入配置项：

```
<property>
    <name>dfs.permissions</name>
    <value>false</value>
</property>
```

如果启用权限检查，就会检查所属用户权限，以确认客户端的用户名与所属用户是否匹配，另外也将检查所属组别权限，以确认该客户端是否是该用户组的成员；若不符，则检查其他权限。

这里有一个超级用户（super-user）的概念，超级用户是 namenode 进程的标识。对于超级用户，系统不会执行任何权限检查。

常用的 HDFS Shell 命令如下：

命令	功能
-help [cmd]	显示命令的帮助信息
-ls(r) <path>	显示当前目录下所有文件
-du(s) <path>	显示目录中所有文件大小
-count[-q] <path>	显示目录中文件数量
-mv <src> <dst>	移动多个文件到目标目录
-cp <src> <dst>	复制多个文件到目标目录
-rm(r)	删除文件(夹)
-put <localsrc> <dst>	本地文件复制到 hdfs
-copyFromLocal	同 put
-moveFromLocal	从本地文件移动到 hdfs
-get [-ignoreCrc] <src>	复制文件到本地，可以忽略 crc 校验

<localdst>

-getmerge <src> <localdst>	将源目录中的所有文件排序合并到一个文件中
-cat <src>	在终端显示文件内容
-text <src>	在终端显示文件内容
-copyToLocal [-ignoreCrc] <src> <localdst>	复制到本地
-moveToLocal <src> <localdst>	移动到本地
-mkdir <path>	创建文件夹
-touchz <path>	创建一个空文件

一下是部分命令的具体使用方法：

cat

使用方法：hadoop fs -cat URI [URI …]

将路径指定文件的内容输出到 stdout 。

示例：

```
hadoop fs -cat hdfs://host1:port1/file1 hdfs://host2:port2/file2
hadoop fs -cat file:///file3 /user/hadoop/file4
```

返回值：

成功返回 0，失败返回-1。

chgrp

使用方法：hadoop fs -chgrp [-R] GROUP URI [URI …]改变文件所属的组。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是 文件的所有

者或者超级用户。更多的信息请参见 HDFS 权限用户指南 。

chmod

使用方法：hadoop fs -chmod [-R] <MODE[,MODE]... | OCTALMODE> URI [URI …]

改变文件的权限。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是文件的所有者或者超级用户。

chown

使用方法：hadoop fs -chown [-R] [OWNER][:[GROUP]] URI [URI]

改变文件的拥有者。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是超级用户。

copyFromLocal

使用方法：hadoop fs -copyFromLocal <localsrc> URI

除了限定源路径是一个本地文件外，和 put 命 令相似。

copyToLocal

使用方法：hadoop fs -copyToLocal [-ignorecrc] [-crc] URI <localdst>

除了限定目标路径是一个本地文件外，和 get 命 令类似。

cp

使用方法：hadoop fs -cp URI [URI …] <dest>

将文件从源路径复制到目标路径。这个命令允许有多个源路径，此时目标路径必须是一个目录。

示例：

```
hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2
hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir
```

返回值：

成功返回 0，失败返回-1。

du

使用方法：hadoop fs -du URI [URI …]

显示目录中所有文件的大小，或者当只指定一个文件时，显示此文件的大小。

示例：

```
hadoop fs -du /user/hadoop/dir1 /user/hadoop/file1
```

```
hdfs://host:port/user/hadoop/dir1
```

返回值：

成功返回 0，失败返回-1。

dus

使用方法：hadoop fs -dus <args>

显示文件的大小。

expunge

使用方法：hadoop fs -expunge

清空回收站。

get

使用方法：hadoop fs -get [-ignorecrc] [-crc] <src> <localdst>

复制文件到本地文件系统。可用-ignorecrc 选项复制 CRC 校验失败的文件。使用-crc 选项复制文件以及 CRC 信息。

示例：

```
hadoop fs -get /user/hadoop/file localfile

hadoop fs -get hdfs://host:port/user/hadoop/file localfile
```

返回值：

成功返回 0，失败返回-1。

getmerge

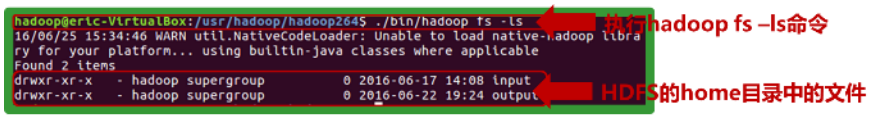
使用方法：hadoop fs -getmerge <src> <localdst> [addnl]

接受一个源目录和一个目标文件作为输入，并且将源目录中所有的文件连接成本地目标文件。addnl 是 可选的，用于指定在每个文件结尾添加一个换行符。

ls

使用方法：hadoop fs -ls <args>

需要注意：在 HDFS 中未带参数的"-ls"命名没有返回任何值，它默认返回 HDFS 的"home"目录下的内容。在 HDFS 中，没有当前目录这样一个概念，也没有 cd 这个命令。



如果是文件，则按照如下格式返回文件信息：

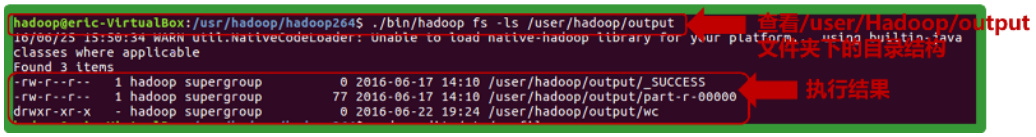
```
文件名 <副本数> 文件大小 修改日期 修改时间 权限 用户 ID 组 ID
```

如果是目录，则返回它直接子文件的一个列表，就像在 Unix 中一样。目录返回列表的信息如下：

```
目录名 <dir> 修改日期 修改时间 权限 用户 ID 组 ID
```

示例：

```
hadoop fs -ls /user/hadoop/file1 /user/hadoop/file2
hdfs://host:port/user/hadoop/dir1 /nonexistentfile
```



返回值：

成功返回 0，失败返回-1。

lsr

使用方法：hadoop fs -lsr <args>

ls 命令的递归版本。类似于 Unix 中的 ls -R 。

mkdir

使用方法：hadoop fs -mkdir <paths>

接受路径指定的 uri 作为参数，创建这些目录。其行为类似于 Unix 的 mkdir -p，它会创建路径中的各级父目录。

示例：

```
hadoop fs -mkdir /user/hadoop/dir1 /user/hadoop/dir2

hadoop fs -mkdir hdfs://host1:port1/user/hadoop/dir hdfs://host2:port2/user/hadoop/dir
```

返回值：

成功返回 0，失败返回-1。

movefromLocal

使用方法：dfs -moveFromLocal <src> <dst>

输出一个“not implemented”信息。

mv

使用方法：hadoop fs -mv URI [URI …] <dest>

将文件从源路径移动到目标路径。这个命令允许有多个源路径，此时目标路径必须是一个目录。不允许在不同的文件系统间移动文件。

示例：

```
hadoop fs -mv /user/hadoop/file1 /user/hadoop/file2

hadoop fs -mv hdfs://host:port/file1 hdfs://host:port/file2 hdfs://host:port/file3 hdfs://host:port/dir1
```

返回值：

成功返回 0，失败返回-1。

put

使用方法：hadoop fs -put <localsrc> ... <dst>

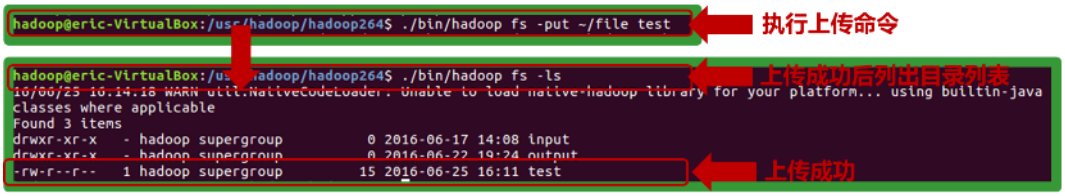
从本地文件系统中复制单个或多个源路径到目标文件系统。也支持从标准输入中读取输入写入目标文件系统。

hadoop fs -put localfile /user/hadoop/hadoopfile

hadoop fs -put localfile1 localfile2 /user/hadoop/hadoopdir

hadoop fs -put localfile hdfs://host:port/hadoop/hadoopfile

从标准输入中读取输入：hadoop fs -put - hdfs://host:port/hadoop/hadoopfile



返回值：

成功返回 0，失败返回-1。

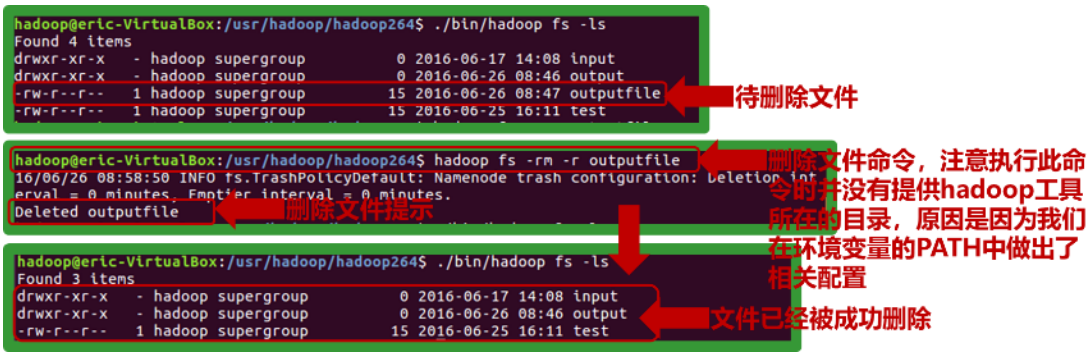
rm

使用方法：hadoop fs -rm URI [URI …]

删除指定的文件。只删除非空目录和文件。请参考 rmr 命令了解递归删除。

示例：

```
hadoop fs -rm hdfs://host:port/file /user/hadoop/emptydir
```



返回值：

成功返回 0，失败返回-1。

rmr

使用方法：hadoop fs -rmr URI [URI …]

delete 的递归版本。

示例：

```
hadoop fs -rmr /user/hadoop/dir
hadoop fs -rmr hdfs://host:port/user/hadoop/dir
```

返回值：

成功返回 0，失败返回-1。

setrep

使用方法：hadoop fs -setrep [-R] <path>

改变一个文件的副本系数。-R 选项用于递归改变目录下所有文件的副本系数。

示例：

```
hadoop fs -setrep -w 3 -R /user/hadoop/dir1
```

返回值：

成功返回 0，失败返回-1。

stat

使用方法：hadoop fs -stat URI [URI …]

返回指定路径的统计信息。

示例：

```
hadoop fs -stat path
```

返回值：

成功返回 0，失败返回-1。

tail

使用方法：hadoop fs -tail [-f] URI

将文件尾部 1K 字节的内容输出到 stdout。支持-f 选项，行为和 Unix 中一致。

示例：

```
hadoop fs -tail pathname
```

返回值：

成功返回 0，失败返回-1。

test

使用方法：hadoop fs -test -[ezd] URI

选项：

-e 检查文件是否存在。如果存在则返回 0。

-z 检查文件是否是 0 字节。如果是则返回 0。

-d 如果路径是个目录，则返回 1，否则返回 0。

示例：

```
hadoop fs -test -e filename
```

text

使用方法：hadoop fs -text <src>

将源文件输出为文本格式。允许的格式是 zip 和 TextRecordInputStream。

touchz

使用方法：hadoop fs -touchz URI [URI …]

创建一个 0 字节的空文件。

示例：

```
hadoop -touchz pathname
```

“hadoop fs”的命令远不止这些，之前介绍的命令已可以在 HDFS 上完成大多数常规操作。对于其他操作，可以通过"-help commandName"命令所列出的清单来进一步学习与探索。

除了文件管理外，HDFS 也提供了一些重要的服务级别的管理控制命令，以下是部分示例：

报告 HDFS 的基本使用情况：

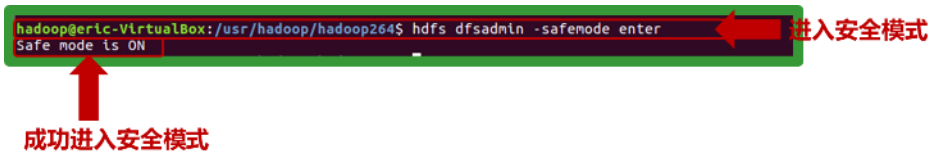
```
hdfs dfsadmin -report
```

NameNode 在启动时会自动进入安全模式。安全模式是 NameNode 的一种状态，

在这个阶段，文件系统不允许有任何修改。安全模式的目的是在系统启动时检查各个 DataNode 上数据块的有效性，同时根据策略对数据块进行必要的复制或删除，当数据块最小百分比数满足的最小副本数条件时，会自动退出安全模式。需要注意的是，HDFS 进入安全模式后会导致 Hive 和 HBase 的启动异常。

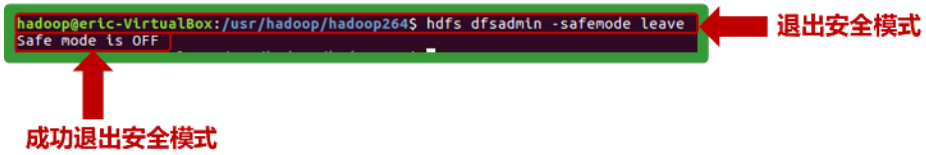
可以使用下面的命令手动进入安全模式：

```
hdfs dfsadmin -safemode enter
```



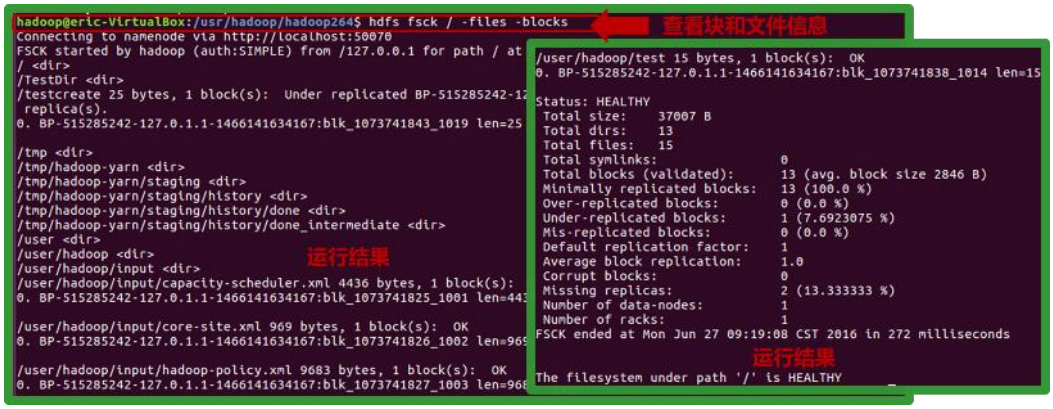
可以通过下面的命令退出安全模式：

```
hdfs dfsadmin -safemode leave
```



与磁盘文件系统相似，HDFS 中 fsck 指令可以显示块信息。例如，执行以下命令将列出文件系统中各个文件由哪些块构成：

```
hdfs fsck / -files -blocks
```



HDFS 的数据在各个 DataNode 中的分布肯能很不均匀，尤其是在 DataNode 节点出现故障或新增 DataNode 节点时。新增数据块时 NameNode 对 DataNode 节点的选择策略也有可能导致数据块分布的不均匀。用户可以使用命令重新平衡 DataNode 上的数据块的分布：

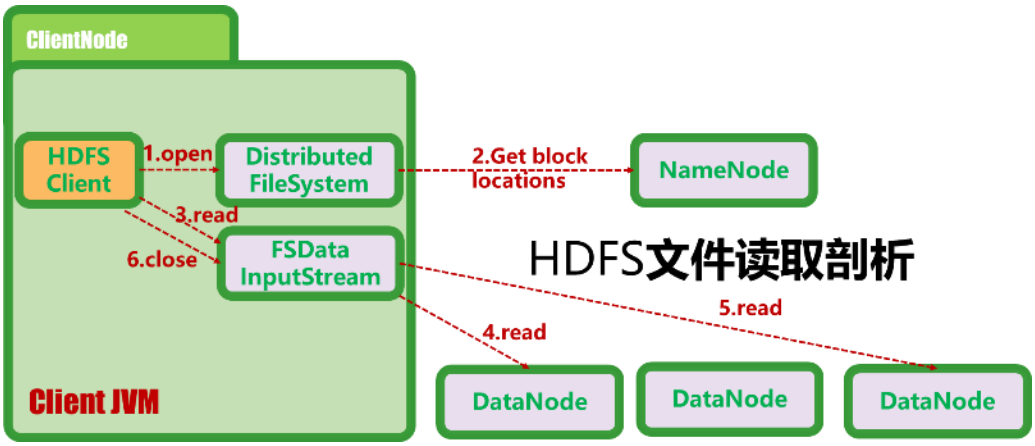
```
start-balancer.sh
```

第6节 HDFS 的文件接口

Hadoop 整合了众多文件系统，在其中有一个综合性的文件系统抽象，它提供了文件系统实现的各类接口，HDFS 只是这个抽象文件系统的一个实例。提供了一个高层的文件系统抽象类 org.apache.hadoop.fs.FileSystem，这个抽象类展示了一个分布式文件系统，并有几个具体实现：

文件系统	URI 方案	Java 实现（org.apache.hadoop）	定义
Local	file	fs.LocalFileSystem	支持有客户端校验和本地文件系统。带有校验和的本地系统文件在 fs.RawLocalFileSystem 中实现。
HDFS	hdfs	hdfs.DistributionFileSystem	Hadoop 的分布式文件系统。
HFTP	hftp	hdfs.HftpFileSystem	支持通过 HTTP 方式以只读的方式访问 HDFS，distcp 经常用在不同的 HDFS 集群间复制数据。
HSFTP	hsftp	hdfs.HsftpFileSystem	支持通过 HTTPS 方式以只读的方式访问 HDFS。
HAR	har	fs.HarFileSystem	构建在 Hadoop 文件系统之上，对文件进行归档。Hadoop 归档文件主要用来减少 NameNode 的内存使用。
KFS	kfs	fs.kfs.KosmosFileSystem	Cloudstore（其前身是 Kosmos 文件系统）文件系统是类似于 HDFS 和 Google 的 GFS 文件系统，使用 C++编写。
FTP	ftp	fs.ftp.FtpFileSystem	由 FTP 服务器支持的文件系统。
S3（本地）	s3n	fs.s3native.NativeS3FileSystem	基于 Amazon S3 的文件系统。
S3（基于块）	s3	fs.s3.NativeS3FileSystem	基于 Amazon S3 的文件系统，以块格式存储解决了 S3 的 5GB 文件大小的限制。

HDFS 文件的读取操作流程如下：



在此过程中，HDFS 内部机制执行如下：

- 客户端(client)用 FileSystem 的 open()函数打开文件。
- DistributedFileSystem 用 RPC 调用元数据节点，得到文件的数据块信息。
- 对于每一个数据块，元数据节点返回保存数据块的数据节点的地址。
- DistributedFileSystem 返回 FSDataInputStream 给客户端，用来读取数据。
- 客户端调用 stream 的 read()函数开始读取数据。
- DFSInputStream 连接保存此文件第一个数据块的最近的数据节点。
- Data 从数据节点读到客户端(client)。
- 当此数据块读取完毕时，DFSInputStream 关闭和此数据节点的连接，然后连接此文件下一个数据块的最近的数据节点。
- 当客户端读取完毕数据的时候，调用 FSDataInputStream 的 close 函数。
- 如果客户端在与数据节点通信出现错误，尝试连接包含此数据块的下一个数据节点。
- 失败的数据节点将被记录，以后不再连接。

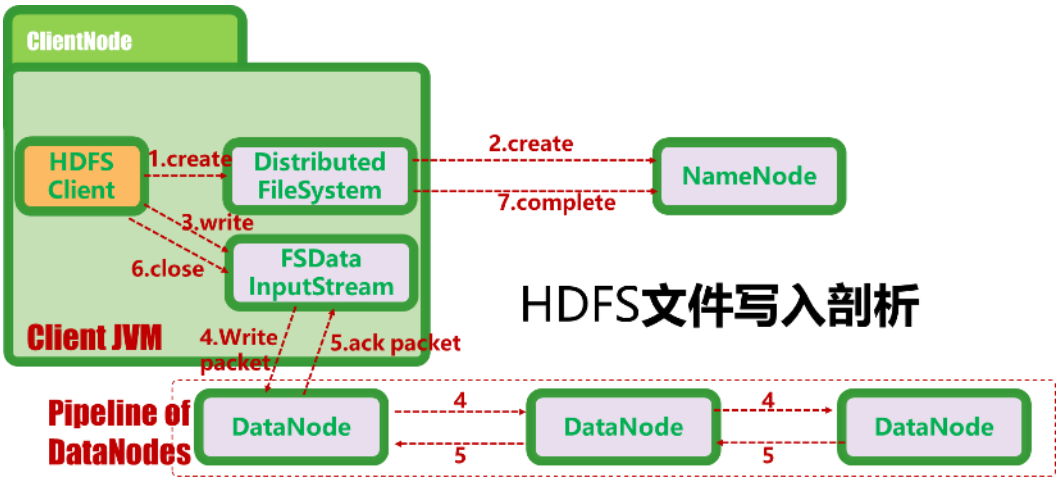
客户端 API 机制执行如下：

- 使用 HDFS 提供的客户端开发库，向远程的 Namenode 发起 RPC 请求；
- Namenode 会视情况返回文件的部分或者全部 block 列表，对于每个 block，Namenode 都会返回有该 block 拷贝的 datanode 地址；
- 客户端开发库会选取离客户端最接近的 datanode 来读取 block；
- 读取完当前 block 的数据后，关闭与当前的 datanode 连接，并为读取下一个 block 寻找最佳的 datanode；
- 当读完列表的 block 后，且文件读取还没有结束，客户端开发库会继续

向 Namenode 获取下一批的 block 列表。

- 读取完一个 block 都会进行 checksum 验证，如果读取 datanode 时出现错误，客户端会通知 Namenode，然后再从下一个拥有该 block 拷贝的 datanode 继续读。

HDFS 文件的写入操作流程如下：



在此过程中，HDFS 内部机制执行如下：

- 客户端调用 create()来创建文件。
- DistributedFileSystem 用 RPC 调用元数据节点，在文件系统的命名空间中创建一个新的文件。
- 元数据节点确定文件原来不存在，并且客户端有创建文件的权限后创建新文件
- DistributedFileSystem 返回 DFSOutputStream，客户端用于写数据。
- 客户端开始写入数据，DFSOutputStream 将数据分成块，写入 data queue。D
- ata queue 由 Data Streamer 读取，并通知元数据节点分配数据节点，用来存储数据块(每块默认复制 3 块)。分配的数据节点放在一个 pipeline 里。
- Data Streamer 将数据块写入 pipeline 中的第一个数据节点。第一个数据节点将数据块发送给第二个数据节点。第二个数据节点将数据发送给第三个数据节点。
- DFSOutputStream 为发出去的数据块保存了 ack queue，等待 pipeline 中的数据节点告知数据已经写入成功。
- 如果数据节点在写入的过程中失败：
  - 关闭 pipeline，将 ack queue 中的数据块放入 data queue 的开始。
  - 当前的数据块在已经写入的数据节点中被元数据节点赋予新的标示，

则错误节点重启后能够察觉其数据块是过时的，会被删除。

- 失败的数据节点从 pipeline 中移除，另外的数据块则写入 pipeline 中的另外两个数据节点。
- 元数据节点则被通知此数据块是复制块数不足，将来会再创建第三份备份。

- 当客户端结束写入数据，则调用 stream 的 close 函数。此操作将所有的数据块写入 pipeline 中的数据节点，并等待 ack queue 返回成功。最后通知元数据节点写入完毕。

客户端 API 机制执行如下：

- 使用 HDFS 提供的客户端开发库，向远程的 Namenode 发起 RPC 请求；
- Namenode 会检查要创建的文件是否已经存在，创建者是否有权限进行操作，成功则会为文件创建一个记录，否则会让客户端抛出异常；
- 当客户端开始写入文件的时候，开发库会将文件切分成多个 packets，并在内部以“data queue”的形式管理 packets，并向 Namenode 申请新的 blocks，获取用来存储 replicas 的合适的 datanodes 列表，其大小根据在 Namenode 中对 replication 的设置而定。
- 开始以 pipeline（管道）的形式将 packet 写入所有的 replicas 中。以流的方式写入第一个 datanode，该 datanode 把该 packet 存储之后，再将其传递给在此 pipeline 中的下一个 datanode，直到最后一个 datanode，写数据的方式呈流水线的形式。
- 最后一个 datanode 成功存储之后会返回一个 ack packet，在 pipeline 里传递至客户端，在客户端的开发库内部维护着"ack queue"，成功收到 datanode 返回的 ack packet 后会从"ack queue"移除相应的 packet。
- 如果传输过程中，有某个 datanode 出现了故障，那么当前的 pipeline 会被关闭，出现故障的 datanode 会从当前的 pipeline 中移除，剩余的 block 会继续剩下的 datanode 中继续以 pipeline 的形式传输，同时 Namenode 会分配一个新的 datanode，保持 replicas 设定的数量。

Hadoop 中关于文件操作类基本上全部是在"org.apache.hadoop.fs"包中，这些 API 能够支持的操作包含：打开文件，读写文件，删除文件等。

Hadoop 类库中最终面向用户提供的接口类是 FileSystem，该类是个抽象类，只能通过来类的 get 方法得到具体类。get 方法存在几个重载版本，常用的是这个：

```
static FileSystem get(Configuration conf);
```

该类封装了几乎所有的文件操作，例如 mkdir，delete 等。基本上可以得出操作文件的程序库框架：

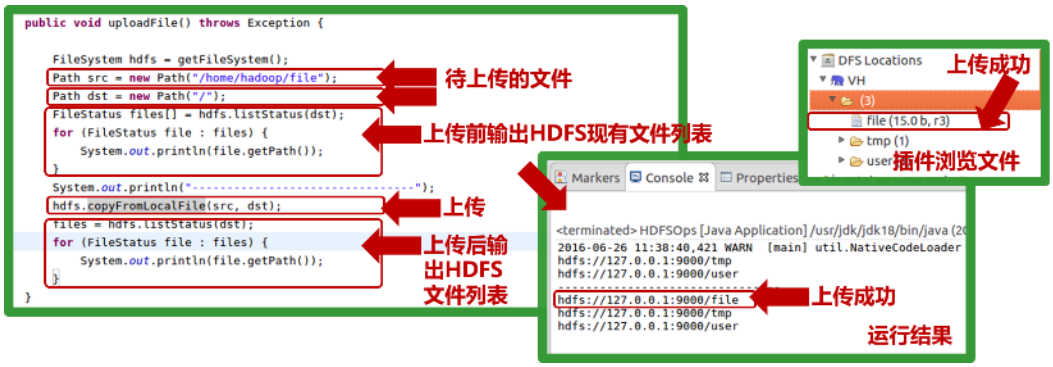
```
operator()
    {
                得到 Configuration 对象
                得到 FileSystem 对象
                进行文件操作
    }
```

在进行实际的文件操作之前，均需要获取到抽象的 FileSystem 对象。构建获取 FileSystem 接口 API 的方法供以后的文件操作使用的具体方式如下：

```
FileSystem getFileSystem() throws Exception {
    URI uri = new URI("hdfs://127.0.0.1:9000/");
    //使用HDFS文件系统并提供服务器路径，端口号在core-site.xml中配置
    FileSystem fileSystem = FileSystem.get(uri, new
Configuration());
    return fileSystem;
}
```

使用 FileSystem 的 copyFromLocalFile(本地文件， HDFS 路径)方法可以将本地文件上传到 HDFS 中：

```
    public void uploadFile() throws Exception {

        FileSystem hdfs = getFileSystem();
        Path src = new Path("/home/hadoop/file");
        Path dst = new Path("/");
        FileStatus files[] = hdfs.listStatus(dst);
        for (FileStatus file : files) {
            System.out.println(file.getPath());
        }
        System.out.println("--------------------------------");
        hdfs.copyFromLocalFile(src, dst);
        files = hdfs.listStatus(dst);
        for (FileStatus file : files) {
            System.out.println(file.getPath());
        }
    }
```

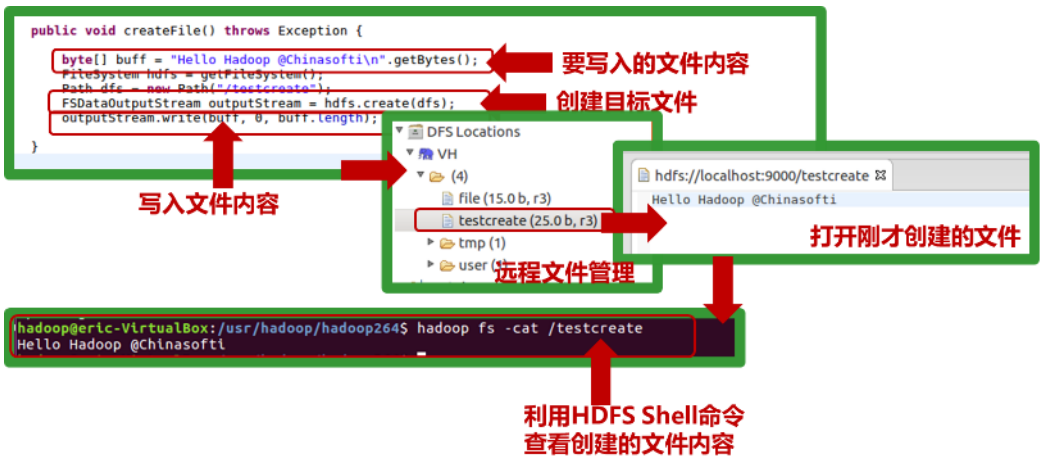


通过"FileSystem.create（Path f）"可在 HDFS 上创建文件，其中 f 为文件的完整路径。具体实现如下：

```
public void createFile() throws Exception {

    byte[] buff = "Hello Hadoop @Chinasofti\n".getBytes();
    FileSystem hdfs = getFileSystem();
    Path dfs = new Path("/testcreate");
    FSDataOutputStream outputStream = hdfs.create(dfs);
    outputStream.write(buff, 0, buff.length);


}
```

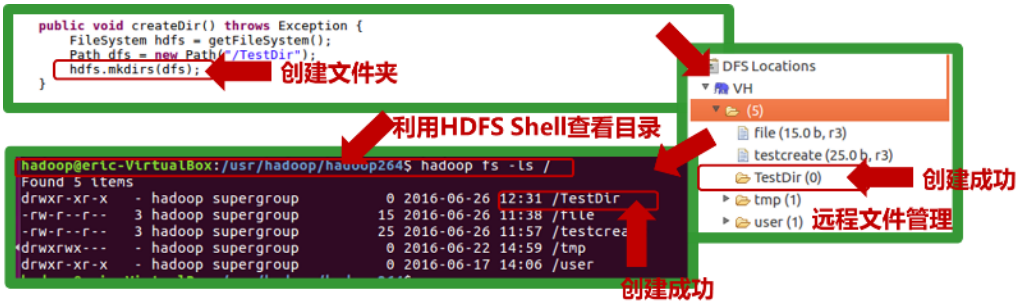
如果运行程序时出现 AccessControlException: Permission denied 异常，则说明访问 HDFS 时出现权限问题，在开发测试环境中可以取消 HDFS 对权限的判定，根据前序章节的介绍，可以在 conf/hdfs-site.xml 增加：

```
<property>
        <name>dfs.permissions</name>
        <value>false</value>
</property>
```



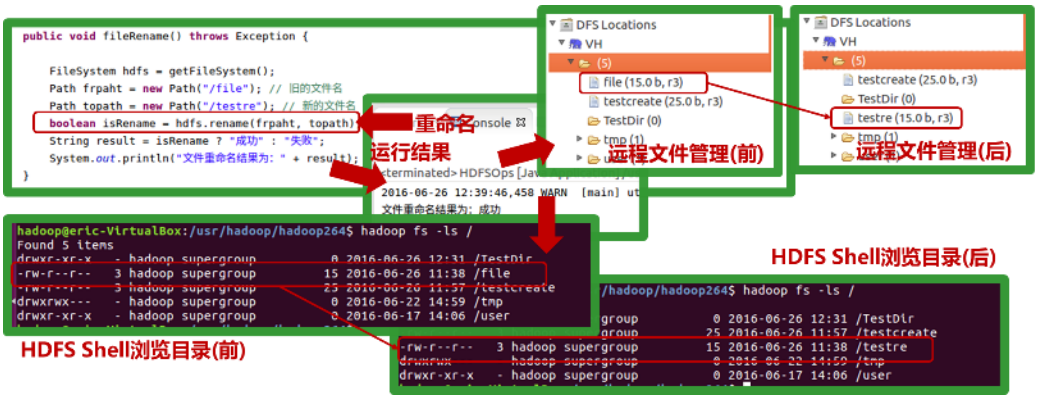
通过"FileSystem.mkdirs（Path f）"可在 HDFS 上创建文件夹，其中 f 为文件夹的完整路径。具体实现如下：

```
public void createDir() throws Exception {
    FileSystem hdfs = getFileSystem();
    Path dfs = new Path("/TestDir");
    hdfs.mkdirs(dfs);

}
```



通过"FileSystem.rename（Path src，Path dst）"可为指定的 HDFS 文件重命名，其中 src 和 dst 均为文件的完整路径。具体实现如下：

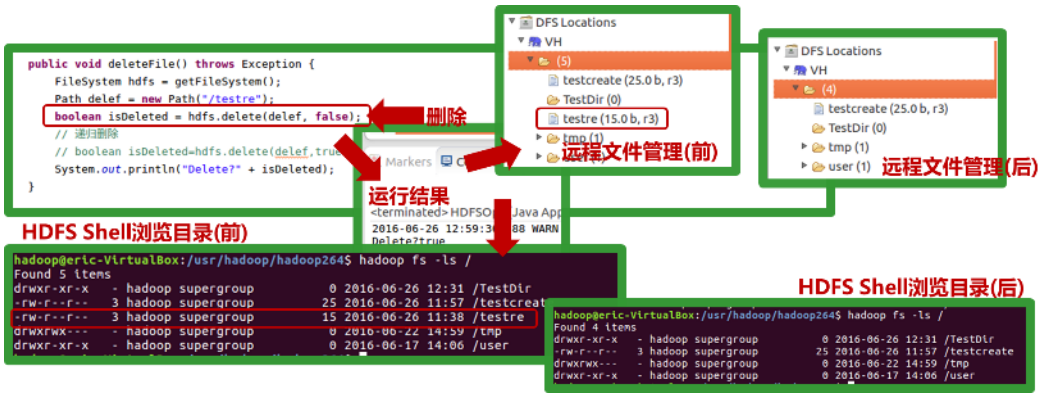
```
public void fileRename() throws Exception {

    FileSystem hdfs = getFileSystem();
    Path frpaht = new Path("/file");
    Path topath = new Path("/testre");
    boolean isRename = hdfs.rename(frpaht, topath);
    String result = isRename ? "成功" : "失败";
    System.out.println("文件重命名结果为：" + result);
}
```



通过"FileSystem.delete(Path f,Boolean recursive)"可删除指定的 HDFS 文件，其中 f 为需要删除文件的完整路径，recuresive 用来确定是否进行递归

删除。具体实现如下：

```
    public void deleteFile() throws Exception {
        FileSystem hdfs = getFileSystem();
        Path delef = new Path("/testre");
        boolean isDeleted = hdfs.delete(delef, false);
        // 递归删除
        // boolean isDeleted=hdfs.delete(delef,true);
        System.out.println("Delete?" + isDeleted);
    }
```



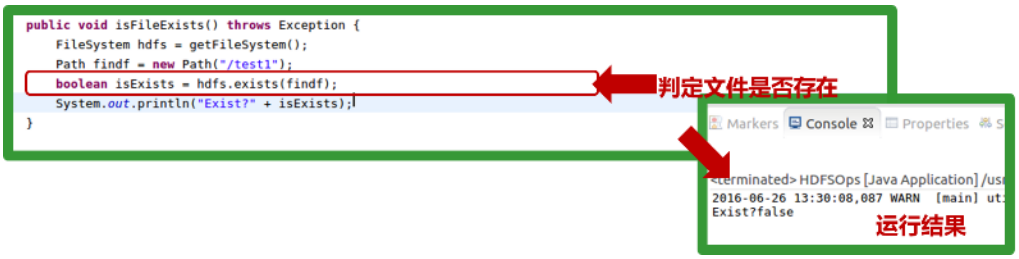
删除文件夹同删除文件代码一样，只是换成删除目录路径即可，如果目录下有文件，要进行递归删除。

通过“FileSystem.open（Path f)”可读取指定的 HDFS 文件。具体实现如下：

```
    public void readFile() throws Exception {
        FileSystem fileSystem = getFileSystem();
        FSDataInputStream openStream = fileSystem.open(new
Path("/testcreate"));
        IOUtils.copyBytes(openStream, System.out, 1024, false);
        IOUtils.closeStream(openStream);

    }
```

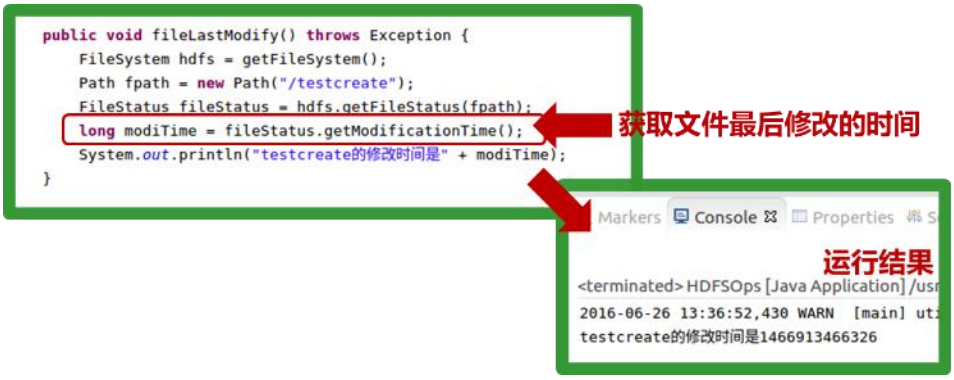
通过"FileSystem.exists（Path f）"可查看指定 HDFS 文件是否存在，其中 f 为文件的完整路径。具体实现如下：

```
public void isFileExists() throws Exception {
    FileSystem hdfs = getFileSystem();
    Path findf = new Path("/test1");
    boolean isExists = hdfs.exists(findf);
    System.out.println("Exist?" + isExists);
}
```



通过"FileSystem.getModificationTime()"可查看指定 HDFS 文件的修改时间。具体实现如下：

```
public void fileLastModify() throws Exception {
    FileSystem hdfs = getFileSystem();
    Path fpath = new Path("/testcreate");
    FileStatus fileStatus = hdfs.getFileStatus(fpath);
    long modiTime = fileStatus.getModificationTime();
    System.out.println("testcreate的修改时间是" + modiTime);
}
```

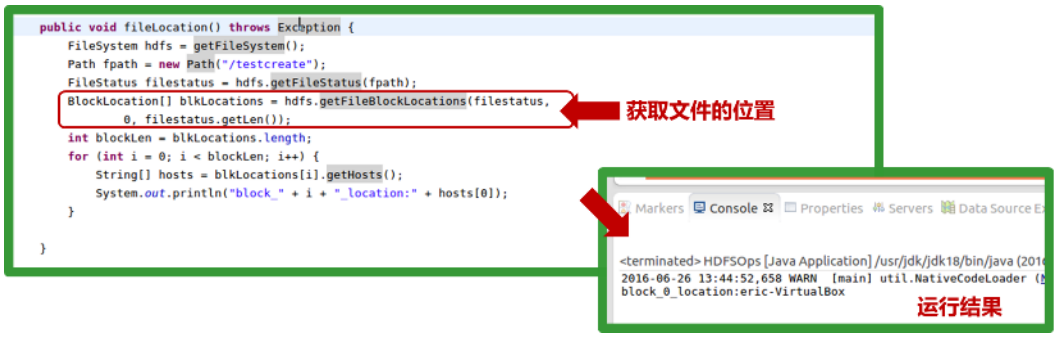


通过“FileStatus.getPath（）”可查看指定 HDFS 中某个目录下所有文件。该功能在上传文件的示例中已经实现。

通过"FileSystem.getFileBlockLocation（FileStatus file，long start，long len）"可查找指定文件在 HDFS 集群上的位置，其中 file 为文件的完整路径，start 和 len 来标识查找文件的路径。具体实现如下：

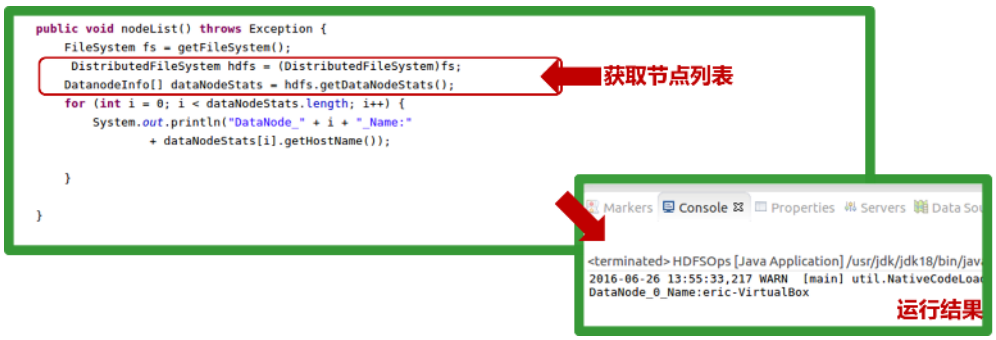
```
public void fileLocation() throws Exception {
    FileSystem hdfs = getFileSystem();
    Path fpath = new Path("/testcreate");
```

```
        FileStatus filestatus = hdfs.getFileStatus(fpath);
        BlockLocation[] blkLocations =
hdfs.getFileBlockLocations(filestatus,
                0, filestatus.getLen());
        int blockLen = blkLocations.length;
        for (int i = 0; i < blockLen; i++) {
            String[] hosts = blkLocations[i].getHosts();
            System.out.println("block_" + i + "_location:" +
hosts[0]);
        }

    }
```



通过"DatanodeInfo.getHostName()"可获取 HDFS 集群上的所有节点名称。具体实现如下：

```
    public void nodeList() throws Exception {
        FileSystem fs = getFileSystem();
        DistributedFileSystem hdfs = (DistributedFileSystem) fs;
        DatanodeInfo[] dataNodeStats = hdfs.getDataNodeStats();
        for (int i = 0; i < dataNodeStats.length; i++) {
            System.out.println("DataNode_" + i + "_Name:"
                    + dataNodeStats[i].getHostName());

        }

    }
```



hadoop 的版本 1.0.4 以后，API 中已经有了追加写入的功能，但不建议在生

产环境中使用，官方文档解释了原因如下：Does HDFS allow appends to files? This is currently set to false because there are bugs in the "append code" and is not supported in any prodction cluster.

如果要测试的话，需要把在 hdfs-site.xml 中设置 dfs.support.appen 的参数设置为 true，不然客户端写入的时候会异常：Exception in thread "main" org.apache.hadoop.ipc.RemoteException: java.io.IOException: Append to hdfs not supported. Please refer to dfs.support.append configuration parameter.

文件系统的一致模型（coherency model）描述了文件读 / 写的数据可见性。HDFS 为性能牺牲了一些 POSIX 要求，因此一些操作与期望的可能不同。

新建一个文件之后，它能在文件系统的命名空间中立即可见，如下所示：

```
Path p=new Path(“p”);
hdfs.create(p);
```

但是，写入文件的内容并不保证能立即可见，即使数据流已经刷新并存储。所以文件长度显示为 0:

```
Path p = new Path(“p”);
OutputStream out = hdfs.create(p);
out.print(“Hello Hadoop!”.getBytes());
System.out.println(hdfs.getFileStatus(p).getLen());
```

写入的数据超过一个块后，第一个数据块对新的 reader 就是可见的。之后的块也不例外。总之，当前正在写入的块对其他 reader 不可见。

HDFS 提供一个方法来使所有缓存与数据节点强行同步，即对 FSDataOutputStream 调用 sync() 方法。当 sync() 方法返回成功后，对所有新的 reader 而言，HDFS 能保证文件中到目前为止写入的数据均到达所有 datanode 的写入管道并且对所有新的 reader 均可见。

每个文件均按块方式存储，每个块的元数据存储在 namenode 的内存中，因此 Hadoop 存储小文件会非常低效。因为大量的小文件会耗尽 namenode 中的大部分内存。但注意，存储小文件所需的磁盘容量和存储这些文件原始内容所需要的磁盘空间相比也不会增多。例如，一个 1MB 的文件以大小为 128MB 的块存储，使用的是 1MB 的磁盘空间，而不是 128MB。

Hadoop 存档文件或 HAR 文件，是一个更高效的文件存档工具，它将文件存入 HDFS 块，在减少 namenode 内存使用的同时，允许对文件进行透明地访问。具体说来，Hadoop 存档文件可以用作 MapReduce 的输入。

在 HDFS 作为高访问量 WEB 系统后台存储解决方案时，Hadoop 存档文件结合类似 Squid 的反向代理能够合理的绕过 HDFS 针对海量小文件进行操作的弊端，

也是现代海量数据 WEB 系统的一个常用解决方案。

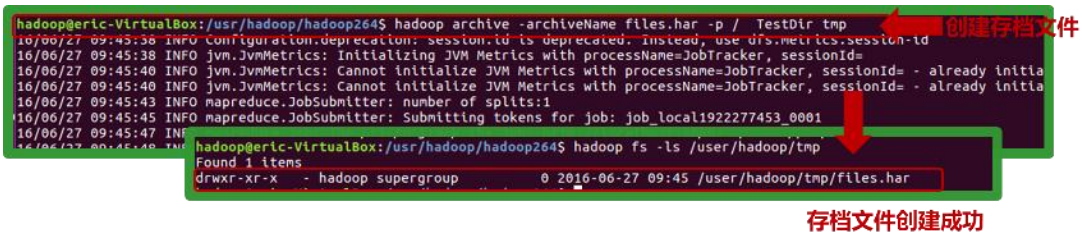
Hadoop 存档是通过 archive 工具根据一组文件创建而来的。该存档工具运行一个 MapReduce 作业来并行处理所有的输入文件，因此你需要一个 MapReduce 集群来运行和使用它。

例如 HDFS 中有一些文档我们希望对它们进行存档：

```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264$ hadoop fs -ls /TestDir
Found 3 items
-rw-r--r--   1 hadoop supergroup      15429 2016-06-27 09:37 /TestDir/LICENSE.txt
-rw-r--r--   1 hadoop supergroup        101 2016-06-27 09:37 /TestDir/NOTICE.txt
-rw-r--r--   1 hadoop supergroup       1366 2016-06-27 09:37 /TestDir/README.txt
```

现在可以运行 archive 指令：

```
hadoop archive -archiveName files.har（存档文件名称） -p /（源文件父路径） TestDir（源文件路径） tmp（目标路径）
```



可以通过 Shell 命令查看 HAR 文件的构造：

```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264$ hadoop fs -ls /user/hadoop/tmp/files.har
Found 4 items
-rw-r--r--   1 hadoop supergroup          0 2016-06-27 09:45 /user/hadoop/tmp/files.har/_SUCCESS
-rw-r--r--   5 hadoop supergroup        396 2016-06-27 09:45 /user/hadoop/tmp/files.har/_index
-rw-r--r--   5 hadoop supergroup         23 2016-06-27 09:45 /user/hadoop/tmp/files.har/_masterindex
-rw-r--r--   1 hadoop supergroup      16896 2016-06-27 09:45 /user/hadoop/tmp/files.har/part-0
```

这个目录列表显示了 HAR 文件的组成部分：两个索引文件以及部分文件的集合，本例中只有一个。这些部分文件中包含已经链接在一起的大量原始文件的内容，并且我们通过索引可以找到包含在存档文件中的部分文件，它的起始点和长度。但所有这些细节对于使用 har URI 方案与 HAR 文件交互的应用都是隐式的，并且 HAR 文件系统是建立在基础文件系统上的。

以下命令以递归方式列出了存档文件中的部分文件：

```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264$ hadoop fs -ls -R har:///user/hadoop/tmp/files.har
drwxr-xr-x   - hadoop supergroup          0 2016-06-27 09:37 har:///user/hadoop/tmp/files.har/TestDir
-rw-r--r--   1 hadoop supergroup      15429 2016-06-27 09:37 har:///user/hadoop/tmp/files.har/TestDir/LICENSE.txt
-rw-r--r--   1 hadoop supergroup        101 2016-06-27 09:37 har:///user/hadoop/tmp/files.har/TestDir/NOTICE.txt
-rw-r--r--   1 hadoop supergroup       1366 2016-06-27 09:37 har:///user/hadoop/tmp/files.har/TestDir/README.txt
```

如果 HAR 文件所在的文件系统是默认的文件系统，这就非常直观易懂。另一方面，如果想在其他文件系统中引用 HAR 文件，则需要使用一个不同于正常情况的 URI 路径格式。以下两个指令作用相同，示例如下：

```
hadoop fs -ls -R har：///user/Hadoop/tmp/files.har
```

```
hadoop fs -ls -R har：//127.0.0.1:9000/user/Hadoop/tmp/files.har
```

如果要在 Java 代码中访问 Hadoop 存档文件，还需要提供一个特殊的 URL 供 HDFS 客户端定位到需要读取的文件位置：

```
    public static void readHarFile() throws URISyntaxException,
IOException {
        //提供特殊的URL形式
        URI uri = new URI("har://hdfs-
192.168.1.1:9000/tmphar/files.har");
        FileSystem fileSystem = FileSystem.get(uri, new
Configuration());
        System.out.println(fileSystem);
        FileStatus[] listStatus = fileSystem.listStatus(new
Path("input"));
         for (FileStatus fileStatus : listStatus) {
             System.out.println(fileStatus.getPath().toString());
         }

        FSDataInputStream openStream = fileSystem.open(new
Path("input/cp.txt"));
        IOUtils.copyBytes(openStream, System.out, 1024, false);
        IOUtils.closeStream(openStream);

    }
```

要想删除 HAR 文件，需要使用递归格式进行删除，因为对于基础文件系统来说，HAR 文件是一个目录：

```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264$ hadoop fs -rm -R /user/hadoop/tmp/files.har
16/06/27 10:12:23 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval = 0 mi
Deleted /user/hadoop/tmp/files.har
```

HDFS 的存档文件也存在一些不足，例如：

存储容量

新建一个存档文件会创建原始文件的一个副本，因此至少需要与要存档（尽管创建了存档文件后可以删除原始文件）的文件容量相同大小的磁盘空间。虽然存档文件中源文件能被压缩（HAR 文件在这方面更接近于 tar 文件），但目前还不支持存档文件的压缩。

更新

一旦创建，存档文件便不能再修改。要想从中增加或删除文件，必须重新创建存档文件。事实上，一般不会再对存档后的文件进行修改，因为它们是定期成批存档的，比如每日或每周。

InputFormat 局限

如前所述，HAR 文件可以作为 MapReduce 的输入。然而，InputFormat 类并不知道文件已经存档，尽管该类可以将多个文件打包成一个 MapReduce 分片，所以即使在 HAR 文件中处理许多小文件，也仍然低效的。

第4章 MapReduce 开发

第1节 MapReduce 概述

MapReduce 最早是由 Google 公司研究提出的一种面向大规模数据处理的并行计算模型和方法。Google 公司设计 MapReduce 的初衷主要是为了解决其搜索引擎中大规模网页数据的并行化处理。Google 公司发明了 MapReduce 之后首先用其重新改写了其搜索引擎中的 Web 文档索引处理系统。但由于 MapReduce 可以普遍应用于很多大规模数据的计算问题，因此自发明 MapReduce 以后，Google 公司内部进一步将其广泛应用于很多大规模数据处理问题。到目前为止，Google 公司内有上万个各种不同的算法问题和程序都使用 MapReduce 进行处理。

2003 年和 2004 年，Google 公司在国际会议上分别发表了两篇关于 Google 分布式文件系统和 MapReduce 的论文，公布了 Google 的 GFS 和 MapReduce 的基本原理和主要设计思想。

Hadoop 的思想来源于 Google 的几篇论文，Google 的那篇 MapReduce 论文里说：Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages。这句话提到了 MapReduce 思想的渊源，大致意思是，MapReduce 的灵感来源于函数式语言（比如 Lisp）中的内置函数 map 和 reduce。简单来说，在函数式语言里，map 表示对一个列表(List)中的每个元素做计算，reduce 表示对一个列表中的每个元素做迭代计算。它们具体的计算是通过传入的函数来实现的，map 和 reduce 提供的是计算的框架。不过从这样的解释到现实中的 MapReduce 还太远，仍然需要一个跳跃。再仔细看，reduce 既然能做迭代计算，那就表示列表中的元素是相关的，比如我想对列表中的所有元素做相加求和，那么列表中至少都应该是数值吧。而 map 是对列表中每个元素做单独处理的，这表示列表中可以是杂乱无章的数据。这样看来，就有点联系了。在 MapReduce 里，Map 处理的是原始数据，自然是杂乱无章的，每条数据之间互相没有关系；到了 Reduce 阶段，数据是以 key 后面跟着若干个 value 来组织的，这些 value 有相关性，至少它们都在一个 key 下面，于是就符合函数式语言里 map 和 reduce 的基本思想了。

这样我们就可以把 MapReduce 理解为，把一堆杂乱无章的数据按照某种特征归纳起来，然后处理并得到最后的结果。Map 面对的是杂乱无章的互不相关的数据，它解析每个数据，从中提取出 key 和 value，也就是提取了数据的特征。经过 MapReduce 的 Shuffle 阶段之后，在 Reduce 阶段看到的都是已经归纳好的数据了，在此基础上我们可以做进一步的处理以便得到结果。

2004 年，开源项目 Lucene（搜索索引程序库）和 Nutch（搜索引擎）的创始人 Doug Cutting 发现 MapReduce 正是其所需要的解决大规模 Web 数据处理的重要技术，因而模仿 Google MapReduce，基于 Java 设计开发了一个称为 Hadoop 的开源 MapReduce 并行计算框架和系统。自此，Hadoop 成为 Apache 开源组织下最重要的项目，自其推出后很快得到了全球学术界和工业界的普遍关注，并得到推广和普及应用。

MapReduce 的推出给大数据并行处理带来了巨大的革命性影响，使其已经成为事实上的大数据处理的工业标准。尽管 MapReduce 还有很多局限性，但人们普遍公认，MapReduce 是到目前为止最为成功、最广为接受和最易于使用的大数据并行处理技术。MapReduce 的发展普及和带来的巨大影响远远超出了发明者和开源社区当初的意料，以至于马里兰大学教授、2010 年出版的《Data-Intensive Text Processing with MapReduce》一书的作者 Jimmy Lin 在书中提出：MapReduce 改变了我们组织大规模计算的方式，它代表了第一个有别于冯·诺依曼结构的计算模型，是在集群规模而非单个机器上组织大规模计算的新的抽象模型上的第一个重大突破，是到目前为止所见到的最为成功的基于大规模计算资源的计算模型。

MapReduce 是面向大数据并行处理的计算模型、框架和平台，它隐含了以下三层含义：

- MapReduce 是一个基于集群的高性能并行计算平台。它允许用市场上普通的商用服务器构成一个包含数十、数百至数千个节点的分布和并行计算集群。

- MapReduce 是一个并行计算与运行软件框架。它提供了一个庞大但设计精良的并行计算软件框架，能自动完成计算任务的并行化处理，自动划分计算数据和计算任务，在集群节点上自动分配和执行任务以及收集计算结果，将数据分布存储、数据通信、容错处理等并行计算涉及到的很多系统底层的复杂细节交由系统负责处理，大大减少了软件开发人员的负担。

- MapReduce 是一个并行程序设计模型与方法。它借助于函数式程序设计语言 Lisp 的设计思想，提供了一种简便的并行程序设计方法，用 Map 和 Reduce 两个函数编程实现基本的并行计算任务，提供了抽象的操作和并行编程接口，以简单方便地完成大规模数据的编程和计算处理。

MapReduce 的编程模型用于大规模数据集(大于 1TB)的并行运算。概念"Map"和"Reduce"，是它的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 Hadoop 中的实现是指定一个 Map 函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce 函数，用来保证所有映射的键值对中的每一个共享相同的键组。

MapReduce 通过把对数据集的大规模操作分发给网络上的每个节点实现可靠性；每个节点会周期性的返回它所完成的工作和最新的状态。如果一个节点保持沉默超过一个预设的时间间隔，主节点记录下这个节点状态为死亡，并把分配给这个节点的数据发到别的节点。每个操作使用命名文件的原子操作以确保不会发生并行线程间的冲突；当文件被改名的时候，系统可能会把他们复制到任务名以外的另一个名字上去（避免副作用）。

Reduce 操作工作方式与之类似，但是由于 Reduce 操作的可并行性相对较差，主节点会尽量把 Reduce 操作只分配在一个节点上，或者离需要操作的数据尽可能近的节点上。

MapReduce 提供了以下的主要功能：

- **数据划分和计算任务调度：**

系统自动将一个作业（Job）待处理的大数据划分为很多个数据块，每个数据块对应于一个计算任务（Task），并自动 调度计算节点来处理相应的数据块。作业和任务调度功能主要负责分配和调度计算节点（Map 节点或 Reduce 节点），同时负责监控这些节点的执行状态，并 负责 Map 节点执行的同步控制。

- **数据/代码互定位：**

为了减少数据通信，一个基本原则是本地化数据处理，即一个计算节点尽可能处理其本地磁盘上所分布存储的数据，这实现了代码向 数据的迁移；当无法进行这种本地化数据处理时，再寻找其他可用节点并将数据从网络上传送给该节点（数据向代码迁移），但将尽可能从数据所在的本地机架上寻 找可用节点以减少通信延迟。

- **系统优化：**

为了减少数据通信开销，中间结果数据进入 Reduce 节点前会进行一定的合并处理；一个 Reduce 节点所处理的数据可能会来自多个 Map 节点，为了避免 Reduce 计算阶段发生数据相关性，Map 节点输出的中间结果需使用一定的策略进行适当的划分处理，保证相关性数据发送到同一个 Reduce 节点；此外，系统还进行一些计算性能优化处理，如对最慢的计算任务采用多备份执行、选最快完成者作为结果。

- **出错检测和恢复：**

以低端商用服务器构成的大规模 MapReduce 计算集群中，节点硬件（主机、磁盘、内存等）出错和软件出错是常态，因此 MapReduce 需要能检测并隔离出错节点，并调度分配新的节点接管出错节点的计算任务。同时，系统还将维护数据存储的可靠性，用多备份冗余存储机制提 高数据存储的可靠性，并能及时检测和恢复出错的数据。

MapReduce 设计上具有以下主要的技术特征：

- **向“外”横向扩展，而非向“上”纵向扩展：**

即 MapReduce 集群的构建完全选用价格便宜、易于扩展的低端商用服务器，而非价格昂贵、不易扩展的高端服务器。

对于大规模数据处理，由于有大 量数据存储需要，显而易见，基于低端服务器的集群远比基于高端服务器的集群优越，这就是为什么 MapReduce 并行计算集群会基于低端服务器实现的原 因。

- **失效被认为是常态：**

MapReduce 集群中使用大量的低端服务器，因此，节点硬件失效和软件出错是常态，因而一个良好设计、具有高容错性的并行计算系统不能因为节点 失效而影响计算服务的质量，任何节点失效都不应当导致结果的不一致或不确定性；任何一个节点失效时，其他节点要能够无缝接管失效节点的计算任务；当失效节

点恢复后应能自动无缝加入集群，而不需要管理员人工进行系统配置。

MapReduce 并行计算软件框架使用了多种有效的错误检测和恢复机制，如节点自动重 启技术，使集群和计算框架具有对付节点失效的健壮性，能有效处理失效节点的检测和恢复。

- **把处理向数据迁移**

传统高性能计算系统通常有很多处理器节点与一些外存储器节点相连，如用存储区域网络连接的磁盘阵列，因此，大规模数据处理时外存文件数据 I/O 访问会成为一个制约系统性能的瓶颈。

为了减少大规模数据并行计算系统中的数据 通信开销，代之以把数据传送到处理节点（数据向处理器或代码迁移），应当考虑将处理向数据靠拢和迁移。MapReduce 采用了数据/代码互定位的技术方法，计算节点将首先尽量负责计算其本地存储的数据，以发挥数据本地化特点，仅当节点无法处理本地数据时，再采用就近原则寻找其他可用计算节点，并把数据传送到该可用计算节点。

- **顺序处理数据、避免随机访问数据**

大规模数据处理的特点决定了大量的数据记录难以全部存放在内存，而通常只能放在外存中进行处理。由于磁盘的顺序访问要远比随机访问快得多，因此 MapReduce 主要设计为面向顺序式大规模数据的磁盘访问处理。

为了实现面向大数据集批处理的高吞吐量的并行处理，MapReduce 可以利用集群中 的大量数据存储节点同时访问数据，以此利用分布集群中大量节点上的磁盘集合提供高带宽的数据访问和传输。

- **为应用开发者隐藏系统层细节**

软件工程实践指南中，专业程序员认为之所以写程序困难，是因为程序员需要记住太多的编程细节（从变量名到复杂算法的边界情况处理），这对大脑记忆是 一个巨大的认知负担，需要高度集中注意力；而并行程序编写有更多困难，如需要考虑多线程中诸如同步等复杂繁琐的细节。由于并发执行中的不可预测性，程序的 调试查错也十分困难；而且，大规模数据处理时程序员需要考虑诸如数据分布存储管理、数据分发、数据通信和同步、计算结果收集等诸多细节问题。

MapReduce 提供了一种抽象机制将程序员与系统层细节隔离开来，程序员仅需描述需要计算什么，而具体怎么去计算就交由系统的执行框架处理，这样程序员可从系统层细节中解放出来，而致力于其应用本身计算问题的算法设计。

- **平滑无缝的可扩展性**

这里指出的可扩展性主要包括两层意义上的扩展性：数据扩展和系统规模扩展性。

理想的软件算法应当能随着数据规模的扩大而表现出持续的有效性，性能上的下降程度应与数据规模扩大的倍数相当；在集群规模上，要求算法的计算性能应能随着节点数的增加保持接近线性程度的增长。绝大多数现有的单机算法都达不到 以上理想的要求；把中间结果数据维护在内存中的单机算法在大规模数据处理时很快失效；从单机到基于大规模集群的并行计算从根本上需要完全不同的

算法设计。奇妙的是，MapReduce 在很多情形下能实现以上理想的扩展性特征。

多项研究发现，对于很多计算问题，基于 MapReduce 的计算性能可随节点数目增长保持近似于线性的增长。

Hadoop 自带的单词统计案例是一个 MapReduce 的典型应用，该应用可以用来阐述数据处理方式随着数据集规模扩大的演变。如果想统计下过去 10 年计算机论文出现最多的几个单词，看看大家都在研究些什么，那收集好论文后，统计词频的方法有以下几种：

方法一：可以写一个小程序，把所有论文按顺序遍历一遍，统计每一个遇到的单词的出现次数，最后就可以知道哪几个单词最热门了。

这种方法在数据集比较小时，是非常有效的，而且实现最简单，用来解决这个问题很合适。

方法二：写一个多线程程序，并发遍历论文。

这个问题理论上是可以高度并发的，因为统计一个文件时不会影响统计另一个文件。当执行统计程序的设备是多核或者多处理器，方法二肯定比方法一高效。但是写一个多线程程序要比方法一困难多了，必须自己同步共享数据，比如要防止两个线程重复统计文件。

方法三：把作业交给多个计算机去完成。

可以使用方法一的程序，部署到 N 台机器上去，然后把论文集分成 N 份，一台机器跑一个作业。这个方法跑得足够快，但是部署起来很麻烦，要人工把程序 copy 到别的机器，要人工把论文集分开，最痛苦的是还要把 N 个运行结果进行整合（当然我们也可以再写一个程序）。

方法四：让 MapReduce 来完成。

MapReduce 本质上就是方法三，但是如何拆分文件集，如何 copy 程序，如何整合结果这些都是框架定义好的。开发人员只要定义好这个任务（用户程序），其它都交给 MapReduce。

Map 函数：

接受一个键值对（key-value pair），产生一组中间键值对。MapReduce 框架会将 map 函数产生的中间键值对里键相同的值传递给一个 reduce 函数。

Reduce 函数：

接受一个键，以及相关的一组值，将这组值进行合并产生一组规模更小的值（通常只有一个或零个值）。

在统计词频的例子里，map 函数接受值是文件的内容，map 逐个遍历单词，每遇到一个单词 w，就产生一个中间键值对<w, "1">，这表示单词 w 又找到了一个；MapReduce 将键相同（都是单词 w）的键值对传给 reduce 函数，这样 reduce 函数接受的键就是单词 w，值是一串"1"（默认基本的实现，但可以优化），个数等于键为 w 的键值对的个数，然后将这些“1”累加就得到单词 w 的出现次数。最后这些单词的出现次数会被写到用户定义的位置，存储在底层的分布式存储系统

（HDFS）。

最后详细解释一下 MapReduce 程序的执行流程：

- MapReduce 库先把用户程序的输入文件划分为 M 份（M 为用户定义），然后将用户进程拷贝到集群内其它机器上。
- 用户程序的副本中有一个称为 master，其余称为 worker，master 是负责调度的，为空闲 worker 分配作业(Map 作业或者 Reduce 作业)，worker 的数量也是可以由用户指定的。
- 被分配了 Map 作业的 worker，开始读取对应分片的输入数据，Map 作业数量是由 M 决定的；Map 作业从输入数据中抽取出键值对（Hadoop 在默认情况下将输入数据的每一行作为一个值传递给 Map 作业），每一个键值对都作为参数传递给 map 函数，map 函数产生的中间键值对被缓存在内存中。
- 缓存的中间键值对会被定期写入本地磁盘，而且被分为 R 个区，R 的大小是由用户定义的，将来每个区会对应一个 Reduce 作业；这些中间键值对的位置会被通报给 master，master 负责将信息转发给 Reduce worker。
- master 通知分配了 Reduce 作业的 worker 它负责的分区在什么位置（肯定不止一个地方，每个 Map 作业产生的中间键值对都可能映射到所有 R 个不同分区），当 Reduce worker 把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个 Reduce 作业，所以排序是必须的。
- reduce worker 遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给 reduce 函数，reduce 函数产生的输出会添加到这个分区的输出文件中。
- 当所有的 Map 和 Reduce 作业都完成了，master 唤醒用户程序，MapReduce 函数调用返回用户程序的调度代码。

所有执行完毕后，MapReduce 输出放在了 R 个分区的输出文件中（分别对应一个 Reduce 作业）。用户通常并不需要合并这 R 个文件，而是将其作为输入交给另一个 MapReduce 程序处理。整个过程中，输入数据是来自底层分布式文件系统（HDFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（HDFS）的。而且要注意 Map/Reduce 作业和 map/reduce 函数的区别：Map 作业处理一个输入数据的分片，可能需要调用多次 map 函数来处理每个输入键值对；Reduce 作业处理一个分区的中间键值对，期间要对每个不同的键调用一次 reduce 函数，Reduce 作业最终也对应一个输出文件。

Hadoop MapReduce 的局限性：

Hadoop 有些属性不适合一般企业的需求，特别是移动应用程序。下面是其中的一些特点：

- Hadoop 的最大价值在于数据集，而 Hadoop 所用的数据集是移动应用程

序所用数据库的 10 到 1000 倍。对于许多人来说，使用 Hadoop 就是杀鸡用牛刀。

- Hadoop 有显著的设置和处理开销。 Hadoop 工作可能会需要几分钟的时间，即使相关数据量不是很大。
- Hadoop 在支持具有多维上下文数据结构方面不是很擅长。例如，一个定义给定地理变量值的记录，然后使用垂直连接，来连续定义一个比 hadoop 使用的键值对定义更复杂的数据结构关系。
- Hadoop 必须使用迭代方法处理的问题方面用处不大——尤其是几个连续有依赖性步骤的问题。

第2节 MapReduce 作业入门

MapReduce 作业(Job)是客户端需要执行的一个工作单元,它包括输入数据、MapReduce 程序和配置信息。

Hadoop 将作业分成若干个小任务（task）来执行，其中包括两类任务：map 任务和 reduce 任务。

有两类节点控制着作业执行过程：

一个 JobTracker,通过调度 TaskTracker 上运行的任务来协调所有运行在系统上的作业。

一系列 TaskTracker，TaskTracker 在运行任务的同时将运行进度报告发送给 JobTracker，JobTracker 由此记录每项作业任务的整体进度情况。如果其中一个任务失败，JobTracker 可以在另外一个 TaskTracker 节点上重新调度该任务。

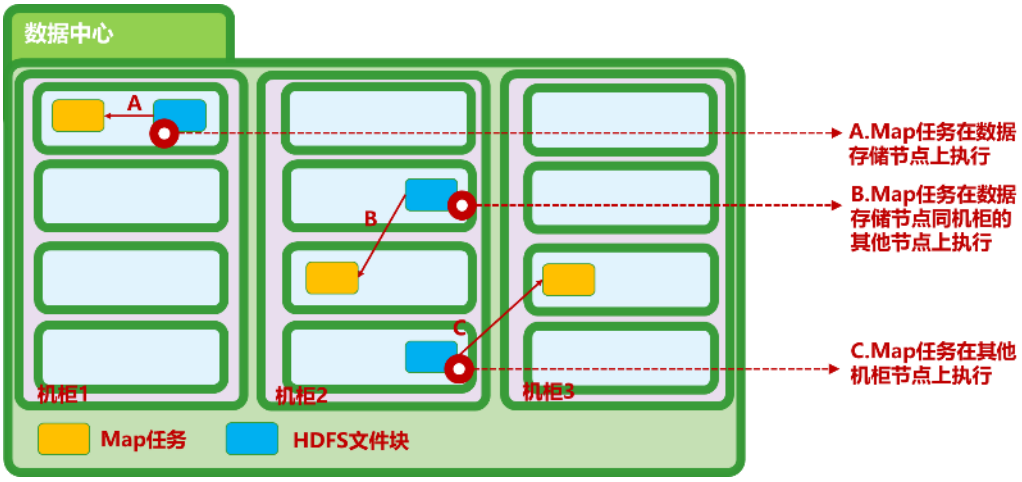
Hadoop 将 MapReduce 的输入数据划分成等长的小数据块，称为输入分片(InputSplit)。

Hadoop 为每个分片构建一个 map 任务，并由该任务来运行用户自定义的 map 方法从而处理分片中的每条记录

拥有许多分片，意味着处理每个分片所需要的时间少于处理整个输入数据所花的时间。因此，如果并行处理每个分片，且每个分片数据比较小，那么整个处理过程将获得更好的负载平衡,因为一台较快的计算机能够处理的数据分片比一台较慢的计算机更多。即使使用相同的机器，失败的进程或其他同时运行的作业能够实现满意的负载平衡，并且如果分片被切分得更细，负载平衡的会更高。另一方面，如果分片切分得太小，那么管理分片的总时间和构建 map 任务的总时间将决定作业的整个执行时间。对于大多数作业来说，一个合理的分片大小趋向于 HDFS 的一个块的大小，默认是 64MB，不过可以针对集群调整这个默认值（对新建的所有文件），或对新建的每个文件具体指定。

Hadoop 在存储有输入数据（HDFS 中的数据）的节点上运行 map 任务，可以获得最佳性能。这就是所谓的“数据本地化优化”(Data Locality Optimization)，

因为它无需使用宝贵的集群带宽资源。但是，有时对于一个 map 任务的输入来说，存储有某个 HDFS 数据块备份的三个节点可能正在运行其他 map 任务，此时作业调度需要在三个备份中的某个数据寻求同个机架中空闲的机器来运行该 map 任务。仅仅在非常偶然的情况下（该情况基本上不会发生），会使用其他机架中的机器运行该 map 任务，这将导致机架与机架之间的网络传输：



在 MapReduce 任务中，最佳分片大小应该与块大小相同，这是一个典型的数据本地优化。这种做法是确保可以存储在单个节点上的最大输入块的大小。如果分片跨越两个数据块，那么对于任何一个 HDFS 节点，基本上都不可能同时存储这两个数据块，因此分片中的部分数据需要通过网络传输到 map 任务节点。与使用本地数据运行整个 map 任务相比，这种方法显然效率更低。

需要注意的是，map 任务将其输出写入本地硬盘，而非 HDFS。

因为 map 的输出是中间结果：该中间结果由 reduce 任务处理后才产生最终输出结果，而且一旦作业完成，map 的输出结果就可以删除。因此，如果把它存储在 HDFS 中并实现备份，难免有些小题大做。如果该节点上运行的 map 任务在将 map 中间结果送给 reduce 任务之前失败，Hadoop 将在另一个节点上重新运行这个 map 任务以再次构建 map 中间结果。

reduce 任务并不具备数据本地化的优势-单个 reduce 任务的输入通常来自于所有 mapper 的输出。排过序的 map 输出需通过网络传输发送到运行 reduce 任务的节点。数据在 reduce 端合并，然后由用户定义的 reduce 方法处理。reduce 的输出通常存储在 HDFS 中以实现可靠存储。

对于每个 reduce 输出的 HDFS 块，第一个复本存储在本地节点上，其他复本存储在其他机架节点中。因此，将 reduce 的输出写入 HDFS 确实需要占用网络带宽，但这与正常的 HDFS 流水线写入的消耗一样。

MapReduce 编程遵循一个特定的流程：

首先写 map 函数和 reduce 函数，最好使用单元测试来确保函数的运行符合预期。

接下来写一个驱动程序来运行作业，看这个驱动程序是否可以正确运行，可以先从本地 IDE 中用一个小的数据集来运行它。如果驱动程序不能正确运行，就

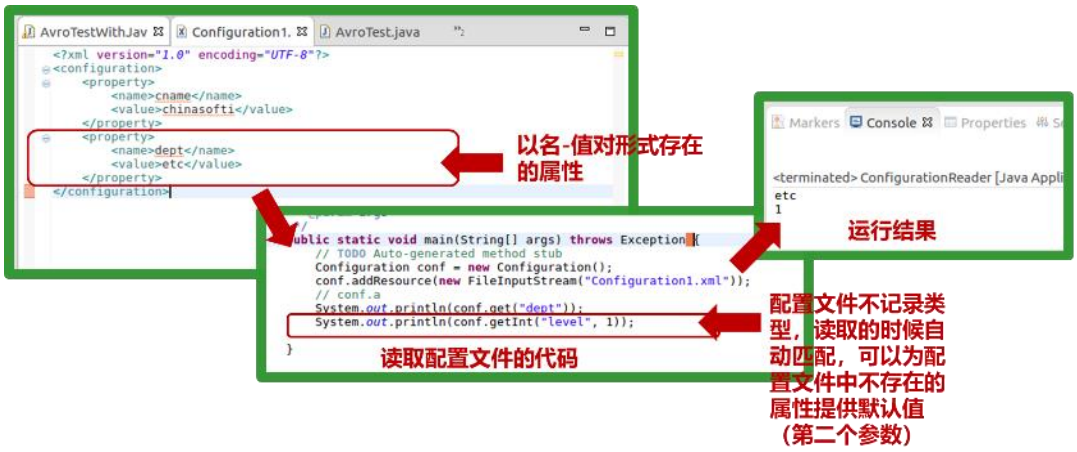
用本地 IDE 调试器来找出问题根源。

根据这些调试信息，可以通过扩展单元测试来覆盖这一测试用例，从而改进 mapper 或 reducer，使其能正确处理类似输入。

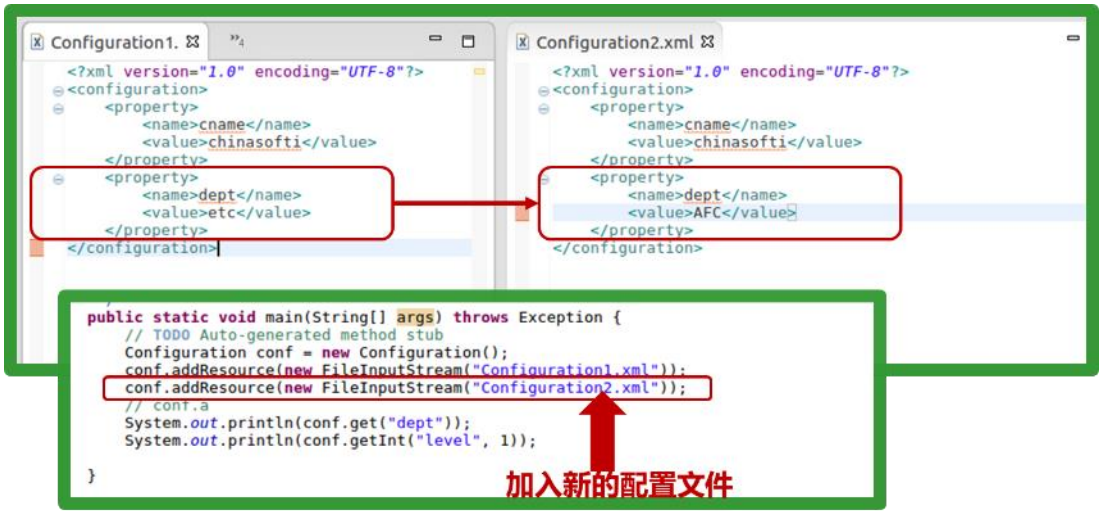
Hadoop 中的组件是通过 Hadoop 自己的配置 API 来配置的。一个 Configuration 类的实例（可以在 org.apache.Hadoop.conf 包中找到）代表配置属性及其取值的一个集合。

每个属性由一个 String 来命名，而值的类型可以是多种类型之一，包括 Java 基本类型（如 boolean、int、iong 和 float）和其他有用的类型（如 String、Class、java.io.File 和 String 集合）。

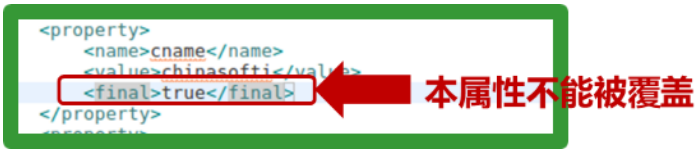
Configuration 从资源（即使用简单结构定义名一值对的 XML 文件）中读取其属性值。在项目文件中创建配置文件，并写入名-值对属性后读取：



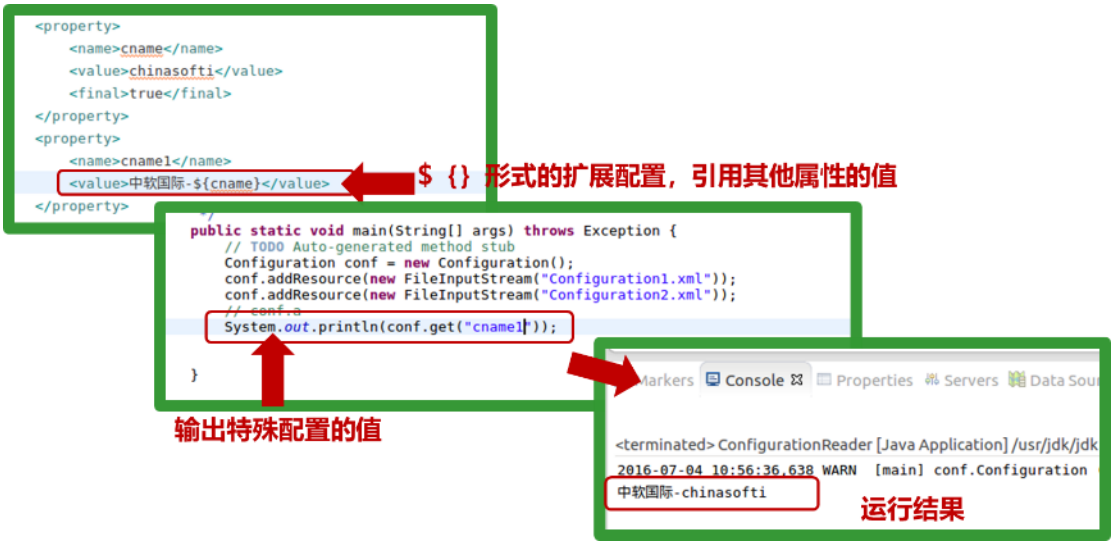
后来添加到资源文件的属性会覆盖之前定义的属性：



被标记为 final 的属性不能被后面的定义所覆盖：



配置属性可以用其他属性或系统属性进行定义：



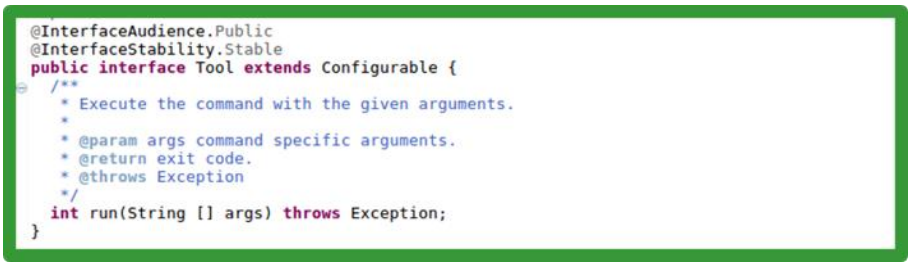
为了简化命令行方式运行作业，Hadoop 自带了一些辅助类：

- GenericOptionsParser
- Tool
- ToolRunner

GenericOptionsParser 是一个类，用来解释常用的 Hadoop 命令行选项，并根据需要，为 Configuration 对象设置相应的取值。

通常不直接使用 GenericOptionsParser。更方便的方式是实现 Tool 接口，通过 ToolRunner 来运行应用程序。

ToolRunner 内部调用 GenericOptionsParser:



如果需要输出当前 Hadoop 的一些核心配置信息，可以使用如下代码：

```
public class ConfigurationPrinter extends Configured implements
Tool {
```

```
    static {

        Configuration.addDefaultResource("hdfs-default.xml");
        Configuration.addDefaultResource("hdfs-site.xml");
        Configuration.addDefaultResource("mapred-default.xml");
        Configuration.addDefaultResource("mapred-site.xml");

    }

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {

        int code=ToolRunner.run(new ConfigurationPrinter(), args);
        System.exit(code);

    }

    @Override
    public int run(String[] arg0) throws Exception {
        // TODO Auto-generated method stub
        Configuration conf = getConf();
        for (Entry<String, String> entry : conf) {
            System.out.println(entry.getKey() + "----" +
entry.getValue());
        }

        return 0;
    }

}
```

Configured 是 Configurable 接口的一个实现。

Tool 的所有实现都需要实现 Configurable（因为 Tool 继承于 Configurable),configured 子类通常是一种最简单的实现方式。

run() 方法通过 Configurable 的 getConf()方法获取 Configuration，然后重复执行，将每个属性打印到标准输出。

静态代码部分确保核心配置外的 HDFS 和 MapReduce 配置能够被获取（因为 Configuration 已经获取了核心配置）。

程序运行结果如下：

可以在 Hadoop 安装路径的 docs 目录中，查看所有公共属性的默认设置，相关文件包括 coredefault.html,hdfs-default.html 和 mapred－default.html 这几个 HTML 文件。

每个属性都有用来解释属性作用和取值范围的描述。

在客户端配五中设置某些属性，将不会产生影响。例如，如果在作业提交时想通过设置 mapred.tasktracker.map.tasks.maximum 来改变运行作业的 tasktracker 的任务槽）数，结果会令你失望，因为这个属性只能在 tasktracke 的 mapred-site.xml 文件中进行设置。

一般情况下，可以通过属性名的组成部分来获知该属性应该在哪里进行设置。由于 mapred.tasktracker.map.tasks.maximum 以 mapred.tasktracker 开头，因此，我们知道它只能为 tasktracker 守护进程进行设置。但是，这不是硬性的，在有些情况下，需要进行尝试，甚至去阅读源代码。

GenericoptionsParser 和 ToolRunner 支持的其他选项可参见表:

选项名称	描述
-D property=value	将指定值赋值给某个 Hadoop 配置属性，覆盖配置文件里的默认属性或站点属性，或通过-conf 选项设置的任务属性
-conf filename …	将指定稳健添加到配置的资源列表中，这是设置站点属性或同事设置一组属性的渐变方法
-fs uri	设置默认的文件系统
-jt host:port	设置 jobtracker 的主机和端口
-files file1，file2…	使用本地文件系统复制到共享文件系统
-archive file1,file2…	使用本地文件系统复制到共享文件系统(打包文件)

-libjars jar1，jar2　　提供本地的 jar 运行库文件

早期版本的 Hadoop 使用一种非常简单的方法来调度用户的作业：按照作业提交的顺序，使用 FIFO（先进先出）调度算法来运行作业。

典型情况下，每个作业都会使用整个集群，因此作业必须等待直到轮到自己运行。虽然共享集群极有可能为多个用户提供大量资源，但问题在于如何公平地在用户之间分配资源，这要求有更好的调度器。

生产作业需要及时完成，同时还要保证正在进行较小临时查询的用户能够在合理时间内得到返回结果。

后来的 Hadoop 中增加了设置作业优先级的功能。

可以通过设置 mapred.job.priority 属性或 JobClient 的 setJobPriority()方法来设置优先级（在这两种方法中，可以选择 VERY_HIGH，HIGH，NORMAL，LOW，VERY_LOW 中的任何值作为优先级）。

在作业调度器选择要运行的下一个作业时，选择的是优先级最高的作业。然而，在 FIFO 调度算法中，优先级并不支持抢占（preemption),所以高优先级的作业仍然受阻于此前已经开始的、长时间运行的低优先级的作业。

在 Hadoop 中，可以选择 MapReduce 的调度器。

MapReducel 的默认调度器是最初基于队列的 FIFO 调度器，还有两个多用户调度器

- 公平调度器（FairScheduler）
- 容量调度器（CapacityScheduler）。

公平调度器的目标是让每个用户公平共享集群能力。

如果只有一个作业在运行，就会得到集群的所有资源。

随着提交的作业越来越多，闲置的任务槽会以“让每个用户公平共享集群”这种方式进行分配。

某个用户的耗时短的作业将在合理的时间内完成，即便另一个用户的长时间作业正在运行而且还在运行过程中。

作业都放在作业池中，在默认情况下，每个用户都有自己的作业池。

提交作业数较多的用户，不会因此而获得更多的集群资源。

公平调度器支持抢占机制，所以，如果一个池在特定的一段时间内未能公平共享资源，就会中止运行池中得到过多资源的任务，把空出来的任务槽让给运行资源不足的作业池。

公平调度器是一个后续模块。要使用它，需要将其 JAR 文件放在 Hadoop 的类路径（classpath)，即将它从 Hadoop 的 contrib/fairscheduler 目录复制到 Iib 目录，然后设置 mapred.jobtracker.taskscheduler 属性：

org.apache.hadoop.mapred.FalrScheduler

针对多用户调度，容量调度器采用的方法稍有不同。集群由很多队列组成（类似于公平调度器的任务池），这些队列可能是层次结构的（因此，一个队列可能是另一个队列的子队列），每个队列被分配有一定的容量。这一点与公平调度器类似，只不过在每个队列内部，作业根据 FIFO 方式（考虑优先级）进行调度。

本质上，容量调度器允许用户或组织（使用队列进行定义）为每个用户或组织模拟出一个使用 FIFO 调度策略的独立 MapReduce 集群。相比之下，公平调度器（实际上也支持作业池内的 FIFO 作业调度，使其类似于容量调度器）强制每个池内公平共享，使运行的作业共享池的资源。

一个最基本的任务调度代码如下，其中比较重要的部分是设置 Map 任务和 Reduce 任务的输入输出类型和数据路径，通过方法的名称可以方便的获知其实现的功能：

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args)
            .getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }

    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);

    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    job.setNumReduceTasks(2);
    job.setPartitionerClass(WordCountPartitioner.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);

}
```

第3节 MapReduce 入门数据分析实例

在进行入门程序编写调试时，往往需要进行一定程度上的调试，由于 MapReduce 是一个分布式框架，任务分散在不同的计算节点运算，因此传统的调

试方法可能无法准确提供开发人员期望获取的信息。在这种情况下，在进行入门案例分析时，可以使用下面的两种。

- 在 Map 中以结果的形式输出调试信息，在 Reduce 中判断，如键或值以“debug”开头。
- 使用 Counter（MapReduce 内置的计数器）：

```
Counter counter = context.getCounter("Map中循环strScore", “输出信息”);

counter.increment(11);
```

实例 1：分析获取历年的最高气温。

在该实例中，需要使用使用 MapReduce 对通过温度传感器获取保存的历年温度数据进行清洗分析，最终获取每一年中出现过的最高气温。为了简化分析操作，可以对温度数据的结构进行简化，例如针对保存在 HDFS 上的如下的数据格式：

通过对源数据格式和最终的目标结果的分析可以发现，原始数据的前四个字符保存的年份信息和第 9 个以后的字符保存的温度信息对程序的运行有支撑作用，剩下的代表月份和日期的字符不在统计范围之列，因此在 Map 函数中对每一行数据进行处理时，可以通过 Java 的字符串操作 API 获取年份部分和温度部分。由于需要统计每一年出现的最高气温，因此在结果中，每一年的数据仅需出现一次，对应的温度是该年份温度集合中的最大值，因此 Map 函数可以将年份作为键，对应的温度作为值输出，Map 函数的参考实现如下：

```
static class TempMapper extends

    Mapper<LongWritable, Text, Text, IntWritable> {

        @Override
        public void map(LongWritable key, Text value, Context
context)

        throws IOException, InterruptedException {
            // 打印样本: Before Mapper: 0, 2000010115
```

```
            System.out.print("Before Mapper: " + key + ", " +
value);
            String line = value.toString();
            String year = line.substring(0, 4);
            int temperature = Integer.parseInt(line.substring(8));
            context.write(new Text(year), new
IntWritable(temperature));
            // 打印样本: After Mapper:2000, 15
            System.out.println("======" + "After Mapper:" + new
Text(year)
                    + ", " + new IntWritable(temperature));
        }

    }
```

数据输出到 Reduce 函数之前，已经经过了必要的汇总操作，将相同的键合并，该键对应的所有值被放置到一个集合中。具体到本案例中，Reduce 拿到的数据中，同一年份的键只会出现一次，而统计获取到的该年所有温度数据都被存档到了以年份为键对应的值-一个可迭代的集合中，因此只需要对该集合进行遍历，找到集合中的最大值（当年的最高气温），并以年份为键输出即可。Reduce 函数的参考实现如下：

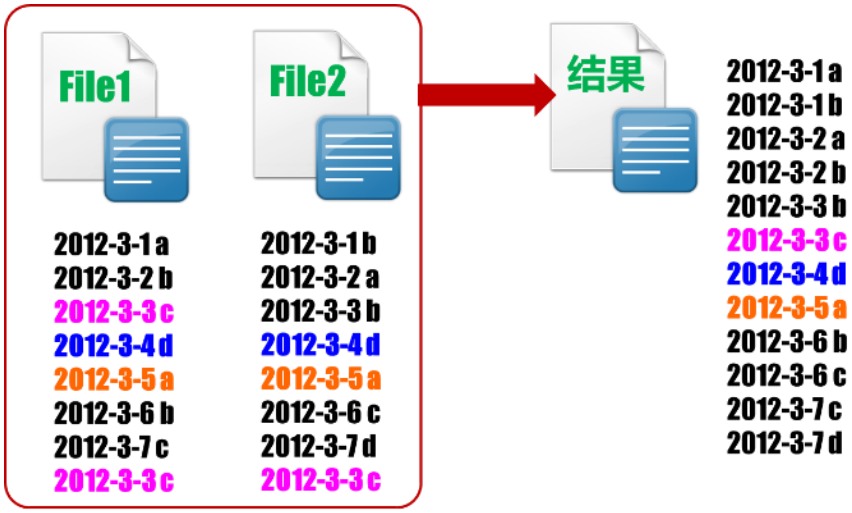
```
    static class TempReducer extends
            Reducer<Text, IntWritable, Text, IntWritable> {
        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws IOException,
InterruptedException {
            int maxValue = Integer.MIN_VALUE;
            StringBuffer sb = new StringBuffer();
            // 取values的最大值
            for (IntWritable value : values) {
                maxValue = Math.max(maxValue, value.get());
                sb.append(value).append(", ");
            }
            // 打印样本： Before Reduce: 2000, 15, 23, 99, 12, 22,
            System.out.print("Before Reduce: " + key + ", " +
sb.toString());
            context.write(key, new IntWritable(maxValue));
            // 打印样本： After Reduce: 2000, 99
            System.out.println("======" + "After Reduce: " + key +
", "
                    + maxValue);
        }
    }
```

完整代码实现参考本教材的随书代码。

实例 2：数据去重。

MapReduce 计算模型内部的一些处理机制为大数据应用系统中部分常见处理提供了便利，如 Map 函数计算的中间结果传输到 Reduce 函数执行之前会执行的 Key 汇总操作，Map 输出的重复 Key 会被合并，开发人员可以利用这一特性直接完成数据的去重操作。

例如要求对 HDFS 上存在的原始数据做如下的操作：



数据去重的最终目标是让原始数据中出现次数超过一次的数据在输出文件中只出现一次。我们自然而然会想到将同一个数据的所有记录都交给一台 reduce 机器，无论这个数据出现多少次，只要在最终结果中输出一次就可以了。

具体就是 reduce 的输入应该以数据作为 key，而对 value-list 则没有要求。当 reduce 接收到一个<key，value-list>时就直接将 key 复制到输出的 key 中，并将 value 设置成空值。

在 MapReduce 流程中，map 的输出<key，value>经过 shuffle 过程聚集成<key，value- list>后会交给 reduce。所以从设计好的 reduce 输入可以反推出 map 的输出 key 应为数据，value 任意。继续反推，map 输出数 据的 key 为数据，而在这个实例中每个数据代表输入文件中的一行内容，所以 map 阶段要完成的任务就是在采用 Hadoop 默认的作业输入方式之后，将 value 设置为 key，并直接输出（输出中的 value 任意）。map 中的结果经过 shuffle 过程之后交给 reduce。reduce 阶段不会管每 个 key 有多少个 value，它直接将输入的 key 复制为输出的 key，并输出就可以了（输出中的 value 被设置成空了）。

对应的 Map 参考实现如下：

```
// map将输入中的value复制到输出数据的key上，并直接输出
public static class Map extends Mapper<Object, Text, Text,
```

```
Text> {

        private static Text line = new Text();// 每行数据

        // 实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            line = value;
            context.write(line, new Text(""));
        }

    }
```

对应的 Reduce 参考实现如下：

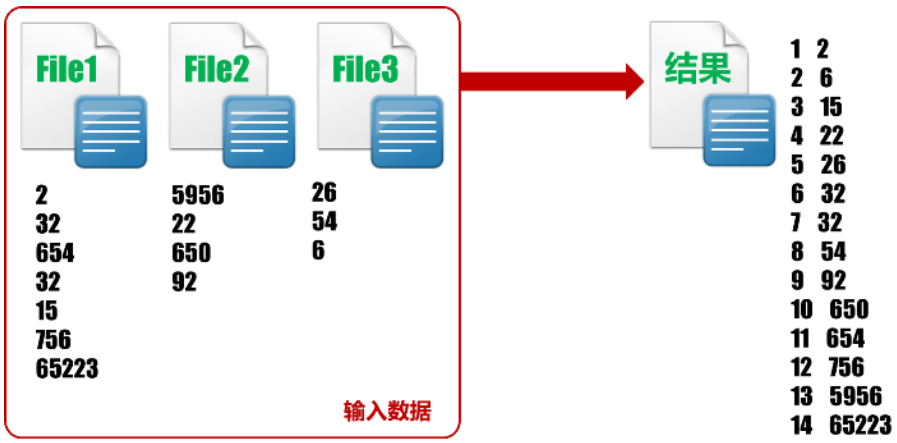
```
    // reduce将输入中的key复制到输出数据的key上，并直接输出
    public static class Reduce extends Reducer<Text, Text, Text,
Text> {
        // 实现reduce函数
        public void reduce(Text key, Iterable<Text> values, Context
context)
                throws IOException, InterruptedException {
            context.write(key, new Text(""));
        }

    }
```

完整代码实现参考本教材的随书代码。

实例 3：数据排序。

“数据排序”是许多实际任务执行时要完成的第一项工作，比如学生成绩评比、数据建立索引等。这个实例和数据去重类似，都是先对原始数据进行初步处理，为进一步的数据操作打好基础。

例如要求对 HDFS 上存在的原始数据做如下的操作：



这个实例仅仅要求对输入数据进行排序，可以利用 MapReduce 默认的排序，而不需要自己再实现具体的排序规则(自定义的排序规则将在后续的章节详细介绍)。

MapReduce 排序过程是按照 key 值进行排序的，如果 key 为封装 int 的 IntWritable 类型，那么 MapReduce 按照数字大小对 key 排序，如果 key 为封装为 String 的 Text 类型，那么 MapReduce 按照字典顺序对字符串排序。

应该使用封装 int 的 IntWritable 型数据结构。也就是在 map 中将读入的数据转化成 IntWritable 型，然后作为 key 值输出（value 任意）。reduce 拿到<key，value-list>之后，将输入的 key 作为 value 输出，并根据 value-list 中元素的个数决定输出的次数。输出的 key（即代码中的 linenum）是一个全局变量，它统计当前 key 的位次。需要注意的是这个程序 MapReduce 过程中不使用 Combiner，使用 map 和 reduce 就已经能够完成任务了。

参考的 Map 实现如下：

```
    // map将输入中的value化成IntWritable类型，作为输出的key
    public static class Map extends
            Mapper<Object, Text, IntWritable, IntWritable> {

        // 实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString().trim();
            context.write(new IntWritable(Integer.parseInt(line)),
                    new IntWritable(1));
        }

    }
```

对应的 Reduce 参考实现如下：

```
// reduce将输入中的key复制到输出数据的key上，然后根据输入的value-
list中元素的个数决定key的输出次数用全局linenum来代表key的位次
    public static class Reduce extends
            Reducer<IntWritable, IntWritable, IntWritable,
IntWritable> {
        static int lineNum = 1;

        // 实现reduce函数
        public void reduce(IntWritable key, Iterable<IntWritable>
values,
                Context context) throws IOException,
InterruptedException {
            for (IntWritable val : values) {
                context.write(new IntWritable(lineNum++), key);
```

```
            }
        }
    }
```

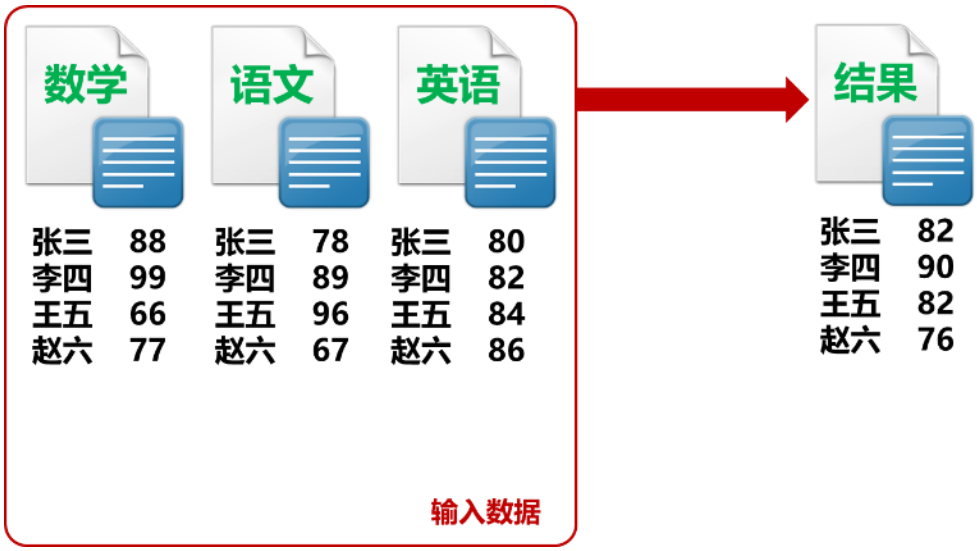
完整代码实现参考本教材的随书代码。

实例 4：平均成绩。

“平均成绩”主要目的还是在重温经典"WordCount"例子，可以说是在基础上的微变化版，该实例主要就是实现一个计算学生平均成绩的例子。

对输入文件中数据进行就算学生平均成绩。输入文件中的每行内容均为一个学生的姓名和他相应的成绩，如果有多门学科，则每门学科为一个文件。要求在输出中每行有两个间隔的数据，其中，第一个代表学生的姓名，第二个代表其平均成绩。

例如要求对 HDFS 上存在的原始数据做如下的操作：



求平均成绩的操作和词频统计相比，Map 操作输出的值不再是 1，而是每个学生对应的考试成绩值，在 Reduce 函数中，除了针对每个 Key（学生）对应的值集合进行累加之外，还要统计当前学生考试的克木属，并在最后实现平均值的计算。

对应的 Map 参考实现如下：

```
    public static class Map extends
            Mapper<LongWritable, Text, Text, IntWritable> {
        // 实现map函数
        public void map(LongWritable key, Text value, Context
context)
                throws IOException, InterruptedException {
            // 将输入的纯文本文件的数据转化成String
            String line = value.toString();
```

```
            String[] datas = line.split(" ");
            context.write(new Text(datas[0]),
                    new IntWritable(Integer.parseInt(datas[1])));

        }


    }
```

对应的 Reduce 参考实现如下：

```
    public static class Reduce extends
            Reducer<Text, IntWritable, Text, IntWritable> {

        // 实现reduce函数
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws IOException,
InterruptedException {
            int sum = 0;
            int count = 0;
            Iterator<IntWritable> iterator = values.iterator();
            while (iterator.hasNext()) {
                sum += iterator.next().get();// 计算总分
                count++;// 统计总的科目数
            }
            int average = (int) sum / count;// 计算平均成绩
            context.write(key, new IntWritable(average));
        }


    }
```

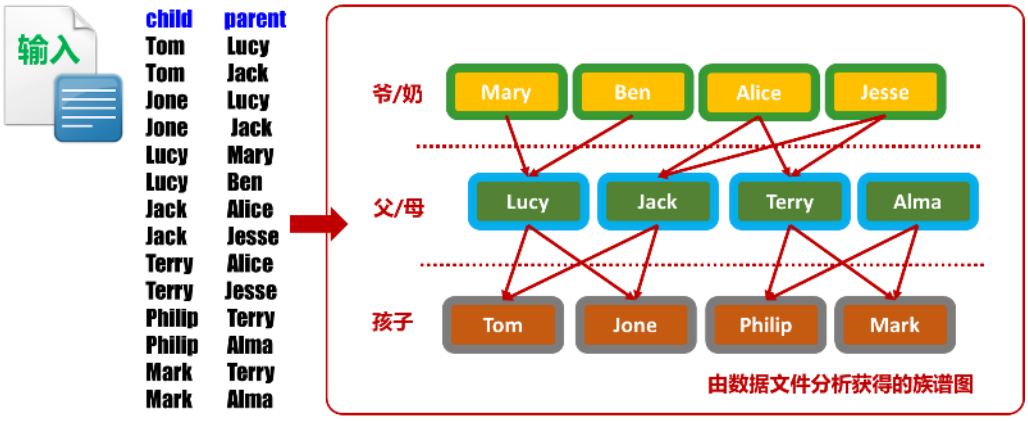
完整代码实现参考本教材的随书代码。

实例 5：单表关联。

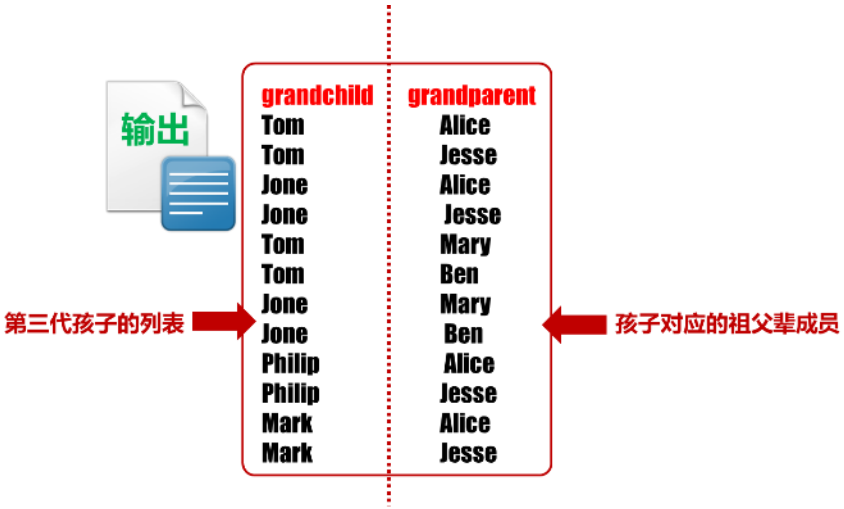
MapReduce 对数据处理的优势在于海量数据的并发处理，在一般情况下，通过不同渠道获取到的海量数据可能不会像传统 RDBMS 数据库中的数据一样被抽象成标准的 Schema 结构并建立关联关系。因此无法像 SQL 语句一样直接处理数据与数据之间的关联关系。然而在很多业务环境中，无法完全规避类似的关联处理。接下来的内容就展示了在 MapReduce 中处理类似问题的一般方法。首先是一个类似于 RDBMS 中单表自关联的问题。

"单表关联"这个实例要求从给出的数据中寻找所关心的数据，它是对原始数据所包含信息的挖掘。

实例中给出 child-parent（孩子——父母）映射表，要求输出 grandchild-grandparent（孙子——爷奶）映射表。原始数据所展示的继承关系如下图所示：



问题要求输出的结果如下：



分析这个实例，显然需要进行类似 RDBMS 中的单表自连接操作，连接的是同一个表中左表（逻辑）的 parent 列和右表（逻辑）的 child 列。

连接结果中除去连接的两列就是所需要的结果:"grandchild--grandparent"表。

要用 MapReduce 解决这个实例，首先应该考虑如何将数据看作表，并实现表的自连接；其次就是连接列的设置；最后是结果的整理。

考虑到 MapReduce 的 shuffle 过程会将相同的 key 会连接在一起，所以可以将 map 结果的 key 设置成待连接的列，然后列中相同的值就自然会连接在一起了。

要连接的是左表的 parent 列和右表的 child 列,且左表和右表是同一个表，所以在 map 阶段将读入数据分割成 child 和 parent 之后，会将 parent 设置成 key，child 设置成 value 进行输出，并作为左表；再将同一对 child 和 parent 中的 child 设置成 key，parent 设置成 value 进行输出，作为右表。为此需要在输出的 value 中再加上左右表的信息，比如在 value 的 String 最开始处加上字符 1 表示左表，字符 2 表示右表。这样在 map 的结果中就形成了左表和右表，然后在 shuffle 过程中完成连接。reduce 接收到连接的结果，其中每个 key 的

value-list 就包含了"grandchild--grandparent"关系。取出每个 key 的 value-list 进行解析，将左表中的 child 放入一个数组，右表中的 parent 放入一个数组，然后对两个数组求笛卡尔积就是最后的结果了。

对应的 Map 参考实现如下：

```
/* *
     * map将输出分割child和parent，然后正序输出一次作为右表，
     * 反序输出一次作为左表，需要注意的是在输出的value中必须加上左右表的区别标识。
     */

    public static class Map extends Mapper<Object, Text, Text, Text> {
        // 实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            System.out.println("value:--------------"+value);
            String childname = "";// 孩子名称
            String parentname = "";// 父母名称
            String relationtype = "";// 左右表标识
            // 输入的一行预处理文本
            String[] values = value.toString().split(" ");
            if (!values[0].equals("child")) {// 忽略表头
                childname = values[0];
                parentname = values[1];
                // 输出左表
                relationtype = "1";
                context.write(new Text(values[1]), new Text(relationtype + "+"
                        + childname + "+" + parentname));
                // 输出右表
                relationtype = "2";
                context.write(new Text(values[0]), new Text(relationtype + "+"
                        + childname + "+" + parentname));
            }
        }

    }
```

由于在结果输出中要求输出表头，因此定义全局变量用于判定当前是否是输出结果文件的表头：

```
public static int time = 0;
```

执行判断并在 Reduce 中输出表头：

```
if (0 == time) {// 输出表头
    context.write(new Text("grandchild"), new Text("grandparent"));
                time++;

}
```

对应的 Reduce 参考实现如下：

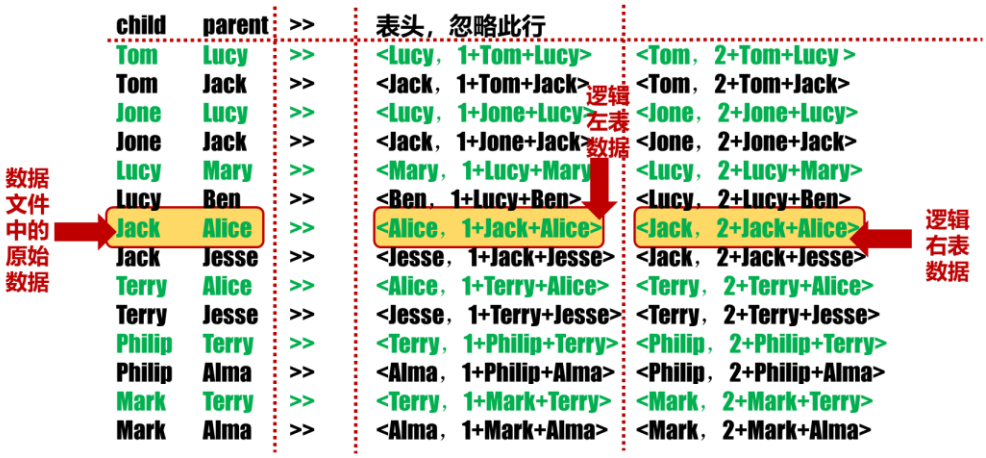
```
    public static class Reduce extends Reducer<Text, Text, Text,
Text> {
        // 实现reduce函数
        public void reduce(Text key, Iterable<Text> values, Context
context)
                throws IOException, InterruptedException {
            if (0 == time) {// 输出表头
                context.write(new Text("grandchild"), new
Text("grandparent"));
                time++;
            }
            int grandchildnum = 0, grandparentnum = 0;
            String[] grandchild = new String[10], grandparent = new
String[10];
            Iterator ite = values.iterator();
            while (ite.hasNext()) {

                String record = ite.next().toString();
                System.out.println("key+----------------
"+key+"record+-----------"+record);
                if (0 == record.length()) {
                    continue;
                }
                String[] datas = record.split("\\+");
                char relationtype = datas[0].charAt(0); // 取得左右表
标识
                String childname = "", parentname = ""; // 定义孩子和
父母变量
                childname = datas[1];// 获取value-list中value的child
                parentname = datas[2];// 获取value-list中value的
parent
                if ('1' == relationtype) {  // 左表，取出child放入
grandchildren
                    grandchild[grandchildnum++] = childname;

                }
```

```
                if ('2' == relationtype) {// 右表，取出parent放入
grandparent
                    grandparent[grandparentnum++] = parentname;

                }
            }
            // grandchild和grandparent数组求笛卡尔儿积
            if (0 != grandchildnum && 0 != grandparentnum) {
                for (int m = 0; m < grandchildnum; m++) {
                    for (int n = 0; n < grandparentnum; n++) {
                        context.write(new Text(grandchild[m]), new
Text(
                                grandparent[n]));
                    }
                }
            }
        }

    }
```

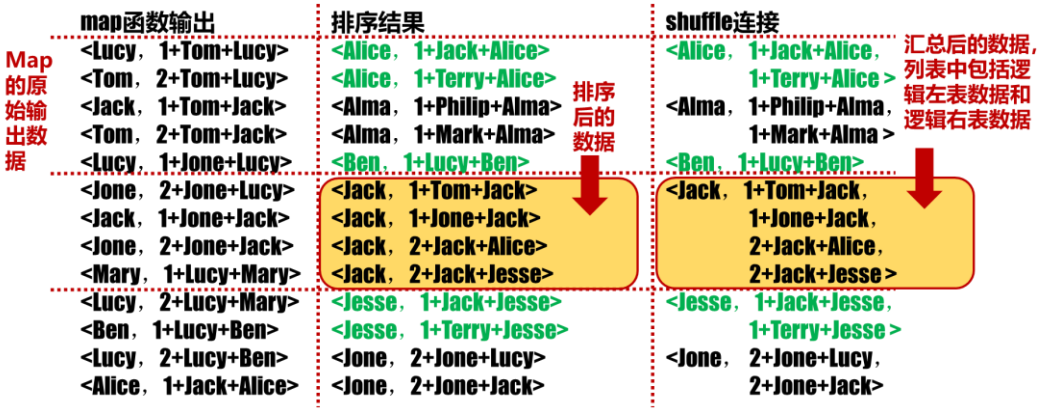
完整代码实现参考本教材的随书代码。

接下来对上述代码的执行过程进行详细解析，更深入的探讨在 MapReduce 中处理数据关联的一般方式。

完成 Map 函数的处理后，中间输出结果如下；map 函数输出结果如下所示：

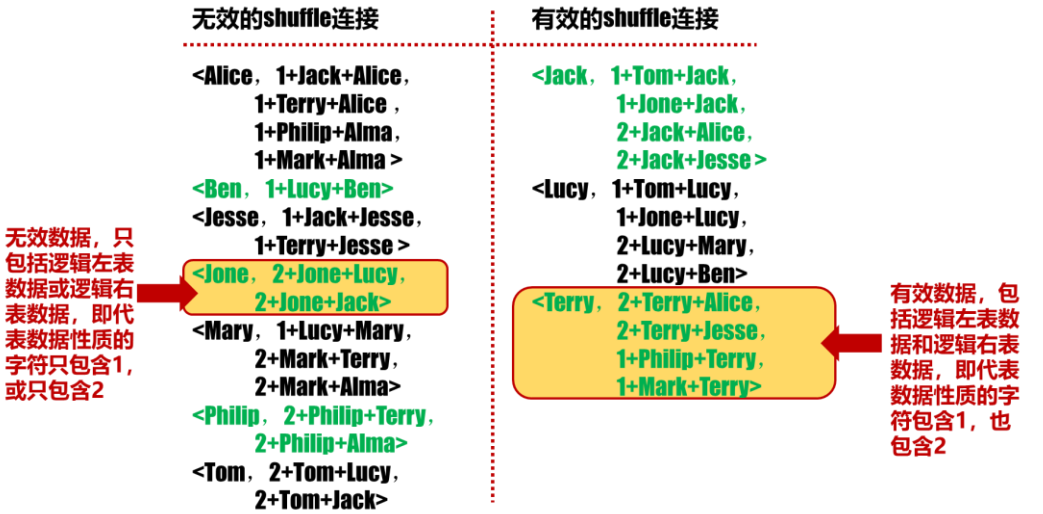


在 shuffle 过程中完成汇总，部分汇总结果如下：



由语句"0 != grandchildnum && 0 != grandparentnum"得知，只要在"value-list"中没有左表或者右表，则不会做处理

可以根据这条规则去除无效的 shuffle 连接：



根据以下代码处理有效 shuffle，分别获取父母辈的姓名和孩子的姓名：

```
if ('1' == relationtype) {  // 左表，取出child放入grandchildren
        grandchild[grandchildnum++] = childname;
}
if ('2' == relationtype) {// 右表，取出parent放入grandparent
        grandparent[grandparentnum++] = parentname;

}
```

针对一条数据进行分析：

<Jack，1+Tom+Jack，
1+Jone+Jack，
2+Jack+Alice，
2+Jack+Jesse>

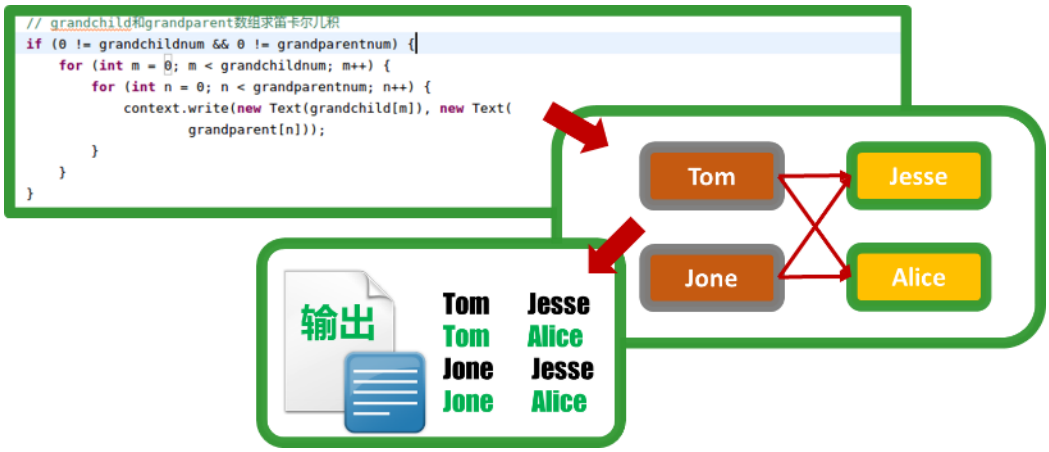
分析结果：左表用"字符 1"表示，右表用"字符 2"表示，上面的<key，value-list>中的"key"表示左表与右表的连接键。而"value-list"表示以"key"连接的左表与右表的相关数据。

根据上面针对左表与右表不同的处理规则，取得两个数组的数据如下所示：

grandchild	Tom、Jone（grandchild[grandchildnum] = childname;）
grandparent	Alice、Jesse（grandparent[grandparentnum] = parentname;）

处理代码：

```
if (0 != grandchildnum && 0 != grandparentnum) {
    for (int m = 0; m < grandchildnum; m++) {
        for (int n = 0; n < grandparentnum; n++) {
            context.write(new Text(grandchild[m]), new Text(
                    grandparent[n]));
        }
    }

}
```



其他的有效 shuffle 连接处理都是相同的流程。

实例 6：多表关联。

多表（逻辑）关联和单表关联类似，它也是通过对原始数据进行一定的处理，从其中挖掘出关心的信息。

案例要求：输入是两个文件，一个代表工厂表，包含工厂名列和地址编号列；另一个代表地址表，包含地址名列和地址编号列。要求从输入数据中找出工厂名和地址名的对应关系，输出“工厂名——地址名”映射表，如下图所示：

多表关联和单表关联相似，都类似于数据库中的自然连接。相比单表关联，多表关联的左右表和连接列更加清楚。所以可以采用和单表关联的相同的处理方式，map 识别出输入的行属于哪个表之后，对其进行分割，将连接的列值保存在 key 中，另一列和左右表标识保存在 value 中，然后输出。reduce 拿到连接结果之后，解析 value 内容，根据标志将左右表内容分开存放，然后求笛卡尔积，最后直接输出。

对应的 Map 参考实现如下：

```
    /**
     * 在map中先区分输入行属于左表还是右表，然后对两列值进行分割，
保存连接列在key值，剩余列和左右表标志在value中，最后输出
     */
    public static class Map extends Mapper<Object, Text, Text,
Text> {
        // 实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();// 每行文件
            String relationtype = "";// 左右表标识
            // 输入文件首行，不处理
            if (line.contains("factoryname") ||
line.contains("addressed")) {
                return;
            }
            // 输入的一行预处理文本
            String[] datas = line.split(" ");
            String mapkey = datas[0].charAt(0) >= '0'&&
datas[0].charAt(0) <= '9' ? datas[0] : datas[1];
            String mapvalue = datas[0].charAt(0) >= '0'   &&
datas[0].charAt(0) <= '9' ? datas[1] : datas[0];
            relationtype = datas[0].charAt(0) >= '0'&&
datas[0].charAt(0) <= '9' ? "2" : "1";
            // 输出左右表
            context.write(new Text(mapkey), new Text(relationtype +
```

```
"+"
                    + mapvalue));
        }
    }
```

对应的 Reduce 参考实现如下：

```
    /**
     * reduce解析map输出，将value中数据按照左右表分别保存，然后求出笛卡尔积，并输出。
     */
    public static class Reduce extends Reducer<Text, Text, Text,
Text> {
        // 实现reduce函数
        public void reduce(Text key, Iterable<Text> values, Context
context)
                throws IOException, InterruptedException {
            // 输出表头
            if (0 == time) {
                context.write(new Text("factoryname"), new
Text("addressname"));
                time++;
            }
            int factorynum = 0, addressnum = 0;
            String[] factory = new String[10], address = new
String[10];
            Iterator ite = values.iterator();
            while (ite.hasNext()) {
                String record = ite.next().toString();
                int i = 2;
                if (0 ==  record.length()) {
                    continue;
                }
                // 取得左右表标识
                char relationtype = record.charAt(0);
                // 左表
                if ('1' == relationtype) {
                    factory[factorynum] = record.substring(i);
                    factorynum++;
                }
                // 右表
                if ('2' == relationtype) {
                    address[addressnum] = record.substring(i);
```

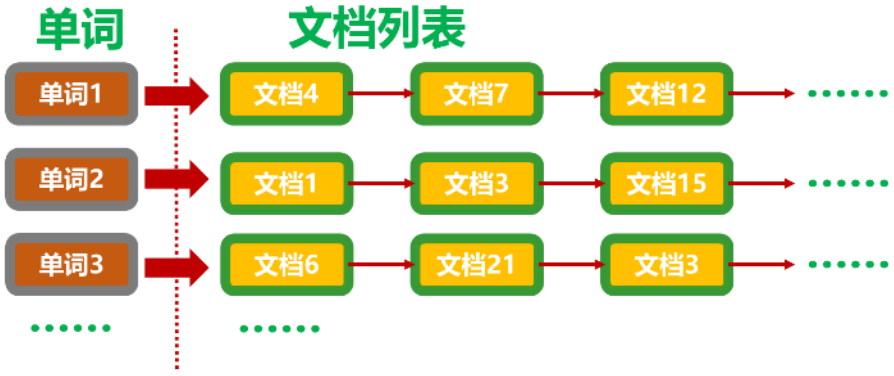
```
                    addressnum++;
                }
            }
            // 求笛卡尔积
            if (0 != factorynum && 0 != addressnum) {
                for (int m = 0; m < factorynum; m++) {
                    for (int n = 0; n < addressnum; n++) {// 输出结果
                        context.write(new Text(factory[m]),    new
Text(address[n]));
                    }
                }
            }
        }
    }
```

完整代码实现参考本教材的随书代码。

实例 7：搜索引擎及全文检索中的倒排索引。

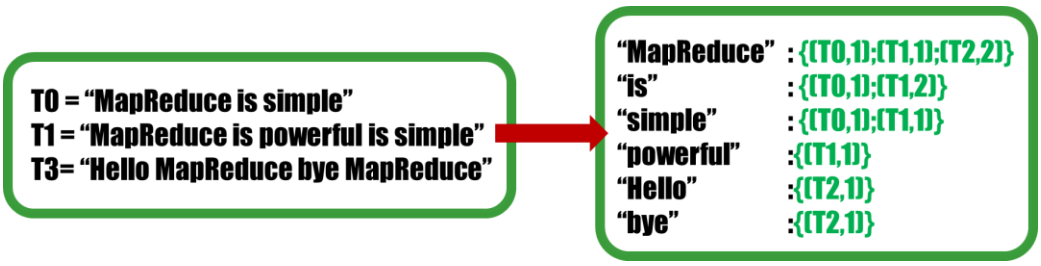
“倒排索引”是文档检索系统中最常用的数据结构，被广泛地应用于全文搜索引擎。它主要是用来存储某个单词（或词组）在一个文档或一组文档中的存储位置的映射，即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容，而是进行相反的操作，因而称为倒排索引（Inverted Index）。

通常情况下，倒排索引由一个单词（或词组）以及相关的文档列表组成，文档列表中的文档或者是标识文档的 ID 号，或者是指文档所在位置的 URL，如图：



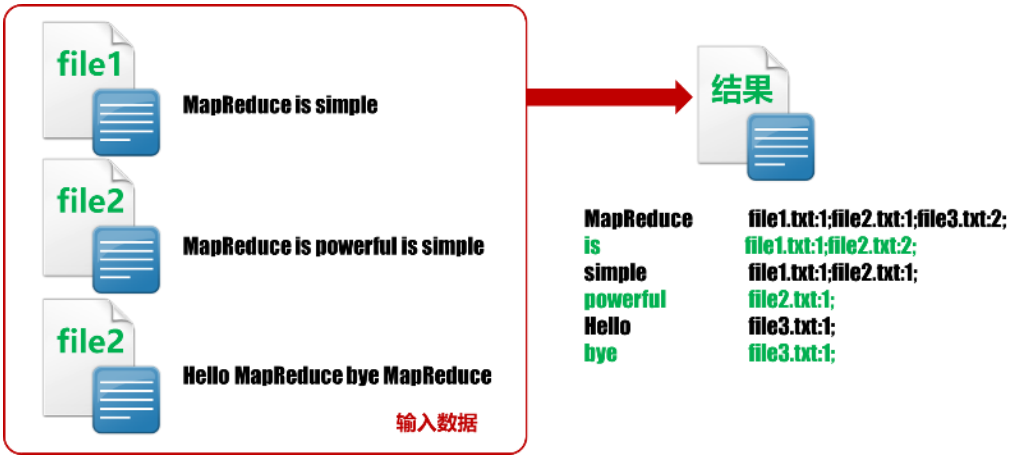
单词 1 出现在{文档 4，文档 7，文档 12，……}中，单词 2 出现在{文档 1，文档 3，文档 15，……}中，而单词 3 出现在{文档 6，文档 21，文档 3，……}中。在实际应用中，还需要给每个文档添加一个权值，用来指出每个文档与搜索内容的相关度：

最常用的是使用词频作为权重，即记录单词在文档中出现的次数。以英文为例，如下所示，索引文件中的"MapReduce"一行表示："MapReduce"这个单词在文本 T0 中出现过 1 次，T1 中出现过 1 次，T2 中出现过 2 次。当搜索条件为"MapReduce"、"is"、"Simple"时，对应的集合为：{T0，T1，T2}∩{T0，T1}∩{T0，T1}={T0，T1}，即文档 T0 和 T1 包含了所要索引的单词，而且只有 T0 是连续的：



更复杂的权重还可能要记录单词在多少个文档中出现过，以实现 TF-IDF（Term Frequency-Inverse Document Frequency）算法，或者考虑单词在文档中的位置信息（单词是否出现在标题中，反映了单词在文档中的重要性）等。

本案例对倒排索引的生成进行了简化，实现基本的索引功能，如下所示：



实现“倒排索引”只要关注的信息为：单词、文档 URL 及词频，但是在实现过程中，索引文件的格式与之前的格式会略有所不同，以避免重写 OutPutFormat 类。下面根据 MapReduce 的处理过程给出倒排索引的设计思路。

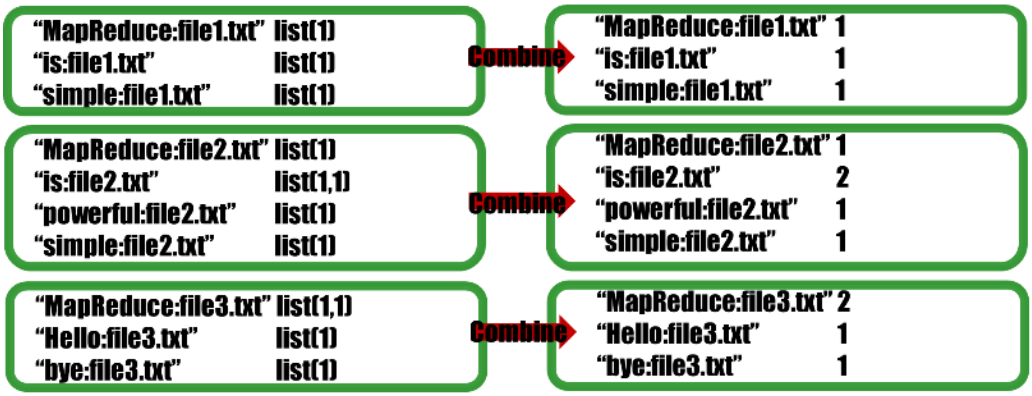
首先使用默认的 TextInputFormat 类对输入文件进行处理，得到文本中每行的偏移量及其内容。显然，Map 过程首先必须分析输入的<key,value>对，得到倒排索引中需要的三个信息：单词、文档 URL 和词频，如下：

这里存在两个问题：

- <key,value>对只能有两个值，在不使用 Hadoop 自定义数据类型的情况下，需要根据情况将其中两个值合并成一个值，作为 key 或 value 值；
- 通过一个 Reduce 过程无法同时完成词频统计和生成文档列表，所以必须增加一个 Combine 过程完成词频统计。

这里将单词和 URL 组成 key 值（如"MapReduce：file1.txt"），将词频作为 value，这样做的好处是可以利用 MapReduce 框架自带的 Map 端排序，将同一文档的相同单词的词频组成列表，传递给 Combine 过程，实现类似于 WordCount 的功能。

经过 map 方法处理后，Combine 过程将 key 值相同的 value 值累加，得到一个单词在文档在文档中的词频。如果直接作为 Reduce 过程的输入，在 Shuffle 过程时将面临一个问题：所有具有相同单词的记录（由单词、URL 和词频组成）应该交由同一个 Reducer 处理，但当前的 key 值无法保证这一点，所以必须修改 key 值和 value 值。这次将单词作为 key 值，URL 和词频组成 value 值（如"file1.txt：1"）。这样做的好处是可以利用 MapReduce 框架默认的 HashPartitioner 类完成 Shuffle 过程，将相同单词的所有记录发送给同一个 Reducer 进行处理。

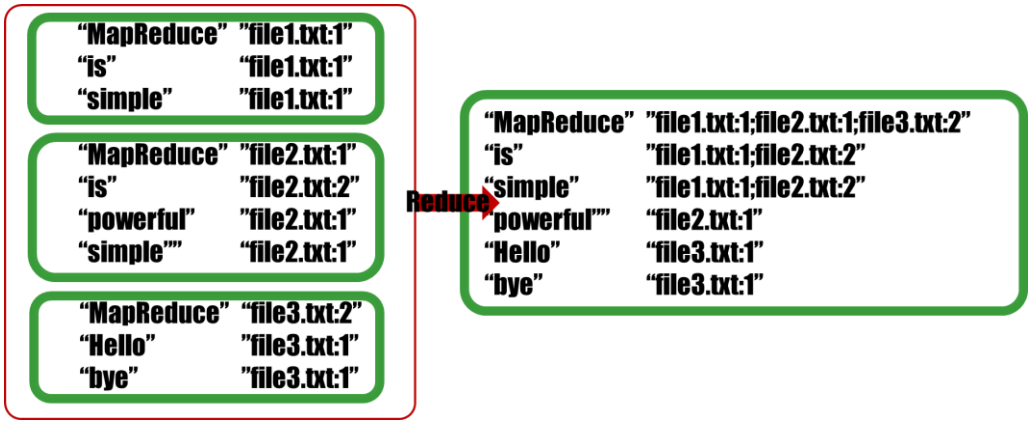


Reduce 过程只需将相同 key 值的 value 值组合成倒排索引文件所需的格式即可，剩下的事情就可以直接交给 MapReduce 框架进行处理了。

本实例设计的倒排索引在文件数目上没有限制，但是单词文件不宜过大（具体值与默认 HDFS 块大小及相关配置有关），要保证每个文件对应一个 split。否则，由于 Reduce 过程没有进一步统计词频，最终结果可能会出现词频未统计完

全的单词。可以通过重写 InputFormat 类将每个文件为一个 split，避免上述情况。或者执行两次 MapReduce，第一次 MapReduce 用于统计词频，第二次 MapReduce 用于生成倒排索引。除此之外，还可以利用复合键值对等实现包含更多信息的倒排索引。

Reduce 过程的输入与输出如下：



对应的 Map 参考实现如下：

```
    public static class Map extends Mapper<Object, Text, Text,
Text> {
        private Text keyInfo = new Text(); // 存储单词和URL组合
        private Text valueInfo = new Text(); // 存储词频
        private FileSplit split; // 存储Split对象
        // 实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            // 获得<key,value>对所属的FileSplit对象
            split = (FileSplit) context.getInputSplit();
            String[] datas = value.toString().split(" ");
            for (String word : datas) {
                int splitIndex =
split.getPath().toString().indexOf("file");//输入文件名以file为前
缀，如file1.txt
                keyInfo = new Text(word + ":"
                        +
split.getPath().toString().substring(splitIndex));
                valueInfo = new Text("1");
                context.write(keyInfo, valueInfo);
            }
        }
    }
```

对应的 Reduce 参考实现如下：

```
    public static class Combine extends Reducer<Text, Text, Text,
Text> {
        private Text info = new Text();
        // 实现reduce函数
        public void reduce(Text key, Iterable<Text> values, Context
context)
                throws IOException, InterruptedException {
            // 统计词频
            int sum = 0;
            for (Text value : values) {
                sum += Integer.parseInt(value.toString());
            }
            int splitIndex = key.toString().indexOf(":");
            // 重新设置value值由URL和词频组成
            info.set(key.toString().substring(splitIndex + 1) + ":"
+ sum);
            // 重新设置key值为单词
            key.set(key.toString().substring(0, splitIndex));
            context.write(key, info);
        }
    }

    public static class Reduce extends Reducer<Text, Text, Text,
Text> {
        private Text result = new Text();

        // 实现reduce函数
        public void reduce(Text key, Iterable<Text> values, Context
context)
                throws IOException, InterruptedException {
            // 生成文档列表
            String fileList = new String();
            for (Text value : values) {
                fileList += value.toString() + ";";
            }
            result.set(fileList);
            context.write(key, result);
        }
    }
```

完整代码实现参考本教材的随书代码。

第4节 MapReduce 序列化

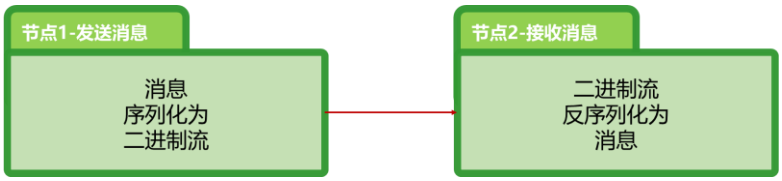
序列化是数据处理程序，尤其是分布式数据处理程序一定会面临的一个重要内容。序列化是指将结构化对象转化为字节流以便在网络上传输或写到磁盘进行永久存储的过程。

对应的，还有反序列化的概念，它是指将字节流转回结构化对象的逆过程。

序列化在分布式数据处理的两大领域经常出现：

- 进程间通信
- 永久存储。

在 Hadoop 中，系统中多个节点上进程间的通信是通过“远程过程调用”(remote procedure call,RPC）实现的。RPC 协议将消息序列化成二进制流后发送到远程节点，远程节点接着将二进制流反序列化为原始消息。



通常情况下，RPC 序列化格式有如下的要求：

紧凑

紧凑格式能充分利用网络带宽（数据中心中最稀缺的资源）。

快速

进程间通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是最基本的。

可扩展

为了满足新的需求，协议不断变化。所以在控制客户端和服务器的过程中，需要直接引进相应的协议。例如，需要能够在方法调用的过程中增添新的参数，并且新的服务器需要能够接受来自老客户端的老格式的消息（无新增的参数）。

支持互操作

对于某些系统来说，希望能支持以不同语言写的客户端与服务器交互，所以需要设计需要一种特定的格式来满足这一需求。

Hadoop 在标准的 Java 序列化方法基础上优化形成了自己的序列化格式 Writable。事实上，Hadoop 认为传统 Java 的对象序列化方式过于复杂，开销较大，因此 Writable 实现时选择了更为轻量化的 DataInput 和 DataOutput，使用基本数据类型和字符串的串行处理实现内部的序列化规范。

MapReduce 的任意 Key 和 Value 必须实现 Writable 接口，以保证任务执行过

程中的中间结果和 Reduce 函数生成的最终结果能够以正确的方式在节点本地磁盘或 HDFS 上写入。

再此基础上，Hadoop 还抽象了 WritableComparable 接口，从名字可以看出，该接口实际是一个可提供比较排序的序列化接口，由于 MapReduce 的中间汇总操作会对 Key 进行排序，因此 MapReduce 的任意 key 必须实现 WritableComparable 接口。这也为复杂的自定义数据排序规则的实现提供了思路。

Writable 接口位于 org.apache.hadoop.io 包中。

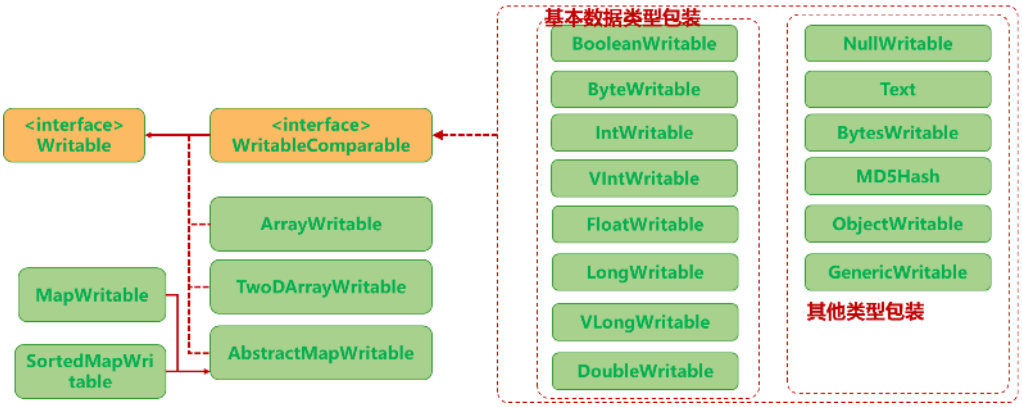
Writable 接口定义了两个方法：

- 一个将其状态写到 DataOutput 二进制流
- 另一个从 DataInput 二进制流读取状态

WritableComparable 接口声明如下：

Hadoop 自带的 org.apache.Hadoop.io 包中有广泛的内置 Writable 类可供选择。

内置 Writable 结构如下：



Writable 类对 Java 基本类型了提供封装，char 除外（可以存储在 IntWritable 中）。所有的封装包含 get() 和 set() 两个方法用于读取或存储封装的值。

Java 基本类型	Writable 实现	序列化大小（字节）
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1-5
float	FloatWetable	4
long	LongWritable	8
	VLongWritable	1-9
double	DoubleWritable	8

对整数进行编码时，有两种选择，即定长格式 IntWritale 和 LongWritable）和变长格式（VIntWritable 和 VLongWritable）。

需要编码的数值如果相当小（在-127 和 127 之间，包括-127 和 127），变长格式就是只用一个字节进行编码。否则，使用第一个字节来表示数值后跟多少个字节，从第二个字节开始保存实际的值。

如何在定长格式和变长格式之间进行选择呢？定长格式非常适合对整个值域空间中分布非常均匀的数值进行编码，如精心设计的哈希函数。大多数数值变量的分布都不均匀，而且变长格式一般更节省空间。

变长编码的另一个优点是可以在 WIntWritable 和 VLongWritable 转换，因为它们的编码实际上是一致的。所以选择变长格式之后，便有增长的空间，不必一开始就用 8 字节的 long 表示。

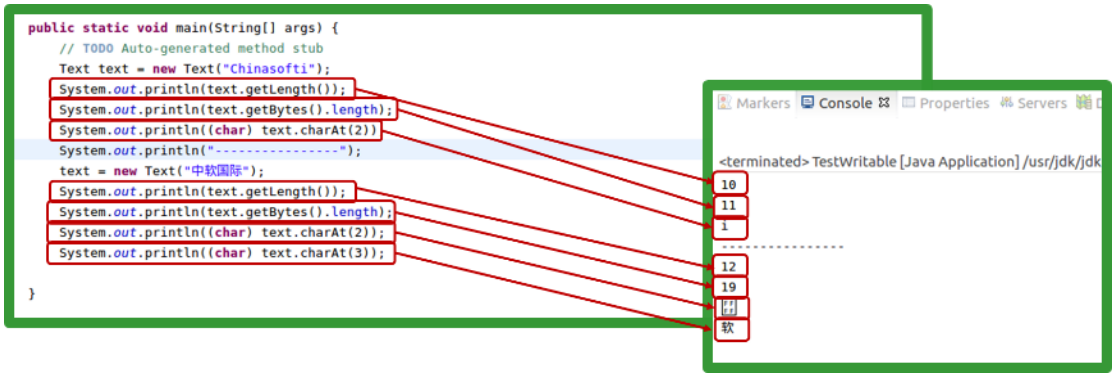
Hadoop 中 Text 类型本质是针对 UTF-8 序列的 Writable 类。一般可以认为它等价与 java.lang.String 的 Writable。

Text 使用整型（通过变长编码的方式）来存储字符串编码中所需的字节数，

最大值为 2GB。它使用标准 UTF-8 编码，这使得能够更简便地与其他理解 UTF-8 编码的工具进行交互操作。

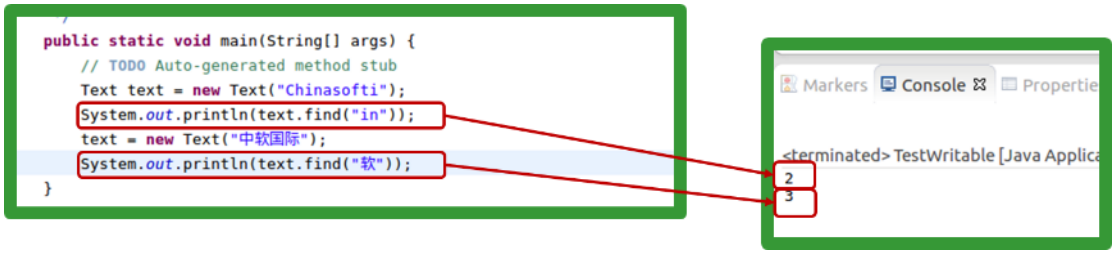
由于着重使用标准的 UTF-8 编码，因此 Text 类和 Java String 类之间存在一定的差别。对 Text 类的索引是根据编码后字节序列中的位置实现的，并非字符串中的 Unicode 字符，也不是 Java char 的编码单元。对于 ASCII 字符串，这三个索引位置的概念是一致的。

通过以下代码可以发现，Text 类的相关 API 均操作转换后的 byte 数组，和 String 的操作差异较大：



注意 charAt() 方法返回的是一个表示 Unicode 编码位置的 int 类型值，而 String 返回一个 char 类型值。

Text 还有一个 find() 方法，该方法类似于 String 的 indexof() 方法：



再一次证明一旦开始使用需要多个字节来编码的字符（比如例子中的中文）时，Text 和 String 之间的区别就昭然若揭了。

处理字符串时 Text 和 String 的区别如下：

Unicode 编码点	U+0041	U+00DF	U+6771	U+10400
名称	拉丁大写字母 A	拉丁小写字母#	一个汉字	候补字符
UTF-8 编码单元	41	c39f	e69db1	F0909080
Java 表示	\u0041	\u00DF	\u6771	\uuD801\uDC00

所有字符（除了表中最后一个字符 U+10400），都可以使用单个 Java char 类型来表示。U+10400 是一个候补字符，并且需要两个 Java char 来表示，称为“字

符代理对”(SurrogatePair)。可以观察到 String 和 Text 之间的表示差别。

由于有缓存机制的存在，因此在有的情况下，Text 的 getBytes（）方法返回的字节数组可能比 getLength（）方法返回的长度更长，因此在调用 getBytes()方法之前始终要调用 getLength（）方法，可以由此知道字节数组中多少字符是有效的。

通过上面实验代码的执行结果，可以总结出 Hadoop 中的 Text 序列化类型和 Java 中的标准 String 字符串有如下区别：

String：

- 长度是其所含 char 编码单元的个数
- indexof() 方法返回 char 编码单元中的索引位置

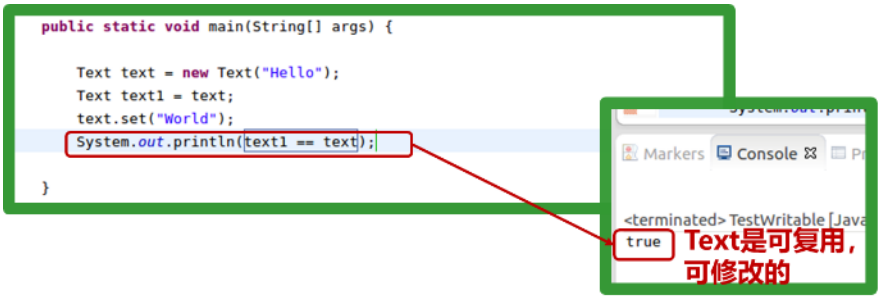
Text：

- 对象的长度却是其 UTF-8 编码的字节数（使用不定长方式编码）
- find() 方法则返回字节偏移量。

之前的这个测试证实：

利用字节偏移量实现的位置索引，对 Text 类中的 Unicode 字符进行迭代是非常复杂的，因为无法通过简单地增加索引值来实现该迭代。同时迭代的语法有些模糊：将 Text 对象转换为 java.nio.ByteBuffer 对象，然后利用缓冲区对 Text 对象反复调用 bytesToCodePoint()静态方法。该方法能够获取下一代码的位置，并返回相应的 int 值，最后更新缓冲区中的位置。当 bytesToCodePoint() 返回 -1 时，则检测到字符串的末尾。

与 String 相比，Text 的另一个区别在于它是可变的（与所有 Hadoop 的 Writable 接口实现相似，NullWritable 除外，它是单实例对象）。可以通过调用其中一个 set() 方法来重用 Text 实例。例如：



Text 类并不像 java.lang.String 类那样有丰富的字符串操作 API。所以，在多数情况下需要将 Text 对象转换成 String 对象。这一转换通常通过调用 toString() 方法来实现。

BytesWitable 是对二进制数据数组的封装。它的序列化格式是一个指定所含数据字节数的整数域（4 字节），后跟数据内容本身。例如，长度为 2 的字节数组包含数值 3 和 5，序列化形式为一个 4 字节的整数（00000002）和该数组中的两个字节（03 和 05）

和 Text 一样，BytesWritable 是可变的，其值可以通过 set() 方法进行修改。和 Text 相似，BytesWritable 类的 getBytes() 方法返回的字节数组长度（容量）可能无法体现 BytesWritable 所存储数据的实际大小。可以通过 getLength() 方法来确定 BytesWritable 的大小。

在 Hadoop 序列化框架中还有一个特殊类型 NullWritable,，它的序列化长度为 0。它并不从数据流中读取数据，也不写入数据。它充当占位符。

例如，在 MapReduce 中，如果不需要使用键或值的序列化地址，就可以将键或值声明为 NullWritable，结果是高效的存储常量空值。如果希望存储一系列数值，与键 / 值对相对，NullWritable 也可以用作在 SequenceFile 中的键。

NullWritable 是一个不可变的单实例类型，通过调用 NullWritable.get() 方法可以获取这个实例。

ObjectWritable 是对 Java 类型（String、enum、Writable、null 或这些类型组成的数组）的一个通用封装。它在 Hadoop RPC 中用于对方法的参数和返回类型进行封装和解封装。

当一个字段中包含多个类型时，ObjectWritable 非常有用：例如，如果 SequenceFile 中的值包含多个类型，就可以将值类型声明为 ObjectWritable，并将每个类型封装在一个 ObjectWritable 中。

SequenceFile 是 Hadoop 中的一种特定数据存储格式，它的存储类似于 Log 文件，所不同的是 Log File 的每条记录的是纯文本数据，而 SequenceFile 的每条记录是可序列化的字符数组。后续内容将对 SequenceFile 进行详细介绍。

作为一个通用的机制，每次序列化都写封装类型的名称，这非常浪费空间。如果封装的类型数量比较少并且能够提前知道，那么可以通过使用静态类型的数组，并使用对序列化后的类型的引用加入位置索引来提高性能。GenericWritable 类采取的就是这种方式，所以需要在继承的子类中显式指定支持类型。

Hadoop 对集合同样提供了默认的序列化支持，org.apache.hadoop.io 包中一共有 6 个 Writable 集合类：

ArrayWritable

ArrayPrimitiveWritable

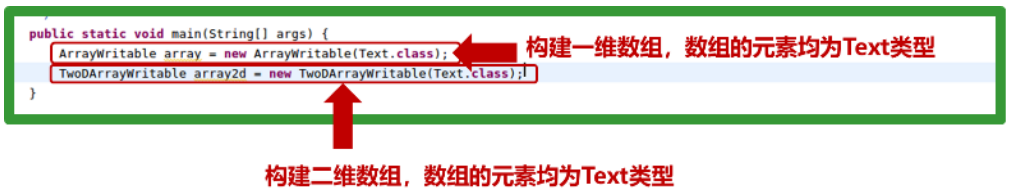
TwoDArrayWritable

MapWritable

SortedMapWritable

EnumMapWritable。

ArrayWritable 和 TwoDArrayWritable 是对 Writable 的数组和二维数组(数组的数组）的实现。ArrayWritable 或 TwoDArrayWritable 中所有元素必须是同一类的实例（在构造函数中指定）



两个类型都有 get()、set()和 toArray()方法。toArray（）方法用于新建该数组（或二维数组）的一个“浅拷贝(shallow copy)”。

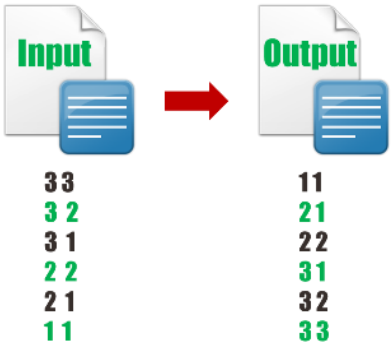
ArrayPrimitiveWritable 是对 Java 基本数据类型数组的一个封装。调用 set()方法时，可以识别相应组件类型，因此无需通过继承该类来设置类型。

MapWritable 和 SortedMapWritable 分别实现了 java.util.Map＜Writable,Writable ＞和 java.util.SortedMap ＜ WritableComparable ，Writable＞。每个键/值字段使用的类型是相应字段序列化形式的一部分。类型存储为单个字节（充当类型数组的索引）。在 org.apache.hadoop.io 包中，数组经常与标准类型结合使用，而定制的 Writable 类型也通常结合使用，但对于非标准类型，则需要在包头指明所使用的数组类型。根据实现，MapWritable 类和 SortedMapWritable 类通过正 byte 值来指示定制的类型，所以在 MapWritable 和 SortedMapWritable 实例中最多可以使用 127 个不同的非标准 Wirtable 类。

在很多应用场景中，Hadoop 默认提供的序列化类型并不能完全匹配开发者的需求，这时候就需要进行定制化的序列化类型设计。当自定义的序列化类型作为 MapReduce 任务的 Value 使用时，实现 Writable 接口即可，而如果自定义序列化类型要作为任务中的 Key 使用，则需要实现 WritableComparable 接口，这还可以帮助开发人员实现自定义的复杂数据排序规则。

MapReduce 框架对处理结果的输出会根据 key 值进行默认的排序，这个默认排序可以满足一部分需求，但是也是十分有限的。在我们实际的需求当中，往往有要对 reduce 输出结果进行二次排序的需求。

自定义二次排序的执行其本质是自定义 Map 处理输出的 Key 类型，因为 Map 执行后的输出将会依据 Key 进行排序，程序只需要自行定义 Key 类型的排序依据即可，例如有如下需求，将输入文件中的数据首先按照第一列升序排列，当第一列相同时，第二列升序排列：



这个需求的实现思路比较简单，将源输入文件中的两列数据均保存在一个自定义的 Key 类型中，然后根据标准 Java 对象的自然排序的方式实现 Key 的比较方法。

首先构建自定义的序列化类型，根据之前内容的描述，该类型需要用作 MapReduce 任务的 Key 中，因此需要实现 WritableComparable 接口，在该类型中需要保存输入文件每一行数据的两个字段，因此需要定义这两个字段对应的属性：

```
public class CustomerSortKey implements
WritableComparable<CustomerSortKey> {
    public long customerKey;

    public long customerValue;
```

实现该类型的序列化方法，通过 DataOutput 的方式将类型中的属性数据写入对应的流中：

```
@Override
public void write(DataOutput out) throws IOException {
    out.writeLong(customerKey);
    out.writeLong(customerValue);

}
```

接下来实现该类型的反序列化方法，通过 DataInput 的方式将类型中的属性值从给定的流中还原：

```
@Override
public void readFields(DataInput in) throws IOException {
    this.customerKey = in.readLong();
    this.customerValue = in.readLong();

}
```

根据 Java 中的对象自然排序的方式实现比较方法：

```
@Override
public int compareTo(CustomerSortKey other) {
```

```
    long temp = this.customerKey - other.customerKey;
    if (temp > 0) {
        return 1;
    } else if (temp < 0) {
        return -1;
    }
    return (int) (other.customerValue-this.customerValue);
}
```

实现比较方法后，根据 Java 对象比较的约定，还应该正确实现该序列化类型的 equals 方法和 hash 方法。在完成序列化类型的声明后，即可实现对应的 MapReduce 任务开发，和之前所编写的 MapReduce 任务不同，本任务的 Map 函数 Key 输出类型和 Reduce 函数的 Key 输入类型不再使用 Hadoop 内置的序列化类型，而是使用刚才已经声明好的自定义序列化类型来代替。

对应的 Map 参考实现如下：

```
    static class MyMapper extends
            Mapper<LongWritable, Text, CustomerSortKey,
LongWritable> {
        @Override
        protected void map(LongWritable key, Text value, Context
context)
                throws IOException, InterruptedException {
            String[] split = value.toString().split(" ");
            CustomerSortKey ck = new
CustomerSortKey(Long.parseLong(split[0]),
                    Long.parseLong(split[1]));
            context.write(ck, new
LongWritable(Long.parseLong(split[1])));
        }

    }
```

对应的 Reduce 函数参考实现如下：

```
    static class MyReduce extends
            Reducer<CustomerSortKey, LongWritable, LongWritable,
LongWritable> {
        @Override
        protected void reduce(CustomerSortKey ck,
Iterable<LongWritable> v2s,
                Context context) throws IOException,
InterruptedException {
            context.write(new LongWritable(ck.customerKey), new
LongWritable(
                    ck.customerValue));
```

```
        }
    }
```

完整代码实现参考本教材的随书代码。

第5节 MapReduce 分区

在默认情况下，MapReduce 认为 Reduce 函数处理的是数据汇总操作，因此其针对的必定是一个 Map 函数清洗处理后的相对规模较小的数据集，且需要对整个集群中所有 Map 的中间输出结果进行统一处理，因此只会启动一个 Reduce 计算节点来处理。这和某些特殊的应用需求并不相匹配，在某些特定的时刻，开发人员希望启动更多的并发 Reduce 节点来优化最终结果统计的性能，减小数据处理的延迟，这通过简单的设置代码即可完成，而在更定制化的环境中，开发人员希望符合特定规则的 Map 中间输出交由特定的 Reduce 节点进行处理，这就需要使用 MapReduce 分区，开发人员还可以提供自定义的分区规则。

如果有好多个 reduce 任务，每个 map 任务就会针对输出进行分区(Partition)，即为每个 reduce 任务建一个分区。每个分区有许多键（及其对应的值），但每个键对应的键 / 值对记录都在同一分区中。如果不给定自定义的分区规则，则 Hadoop 将会使用默认的哈希函数来分区，效率较高。

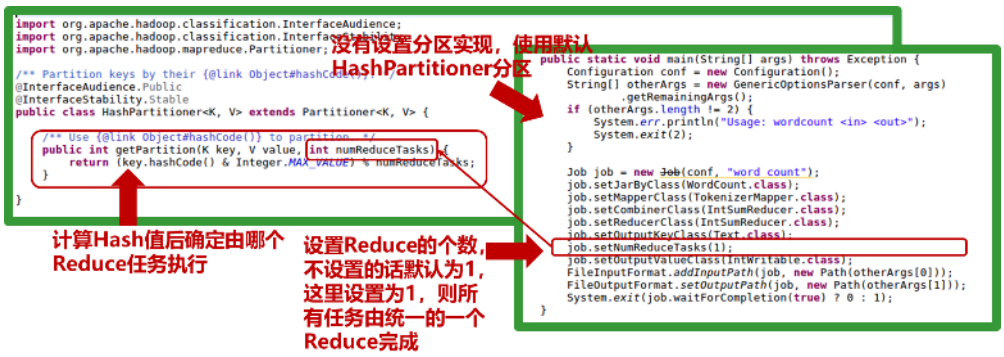
如果希望在执行任务时提供多个 Reduce 计算节点并发处理，只需要在任务调度代码中执行：

```
Job.setNumReduceTasks(自定义 Reduce 任务数量);
```

如果希望实现自定义的分区规则，需要继承 Hadoop 提供的分区类：org.apache.hadoop.mapreduce.Partitioner，该类的声明如下：

HashPartitioner 是 MapReduce 的默认 Partitioner。

在这个分区规则中，选择 Reduce 节点的对应计算方法是 which reducer=(key.hashCode() & Integer.MAX_VALUE) % numReduceTasks，得到当前的目的 reducer。



MapReduce 任务分区的注意事项：

如果 Reduce 一共有 3 个，则 getPartition 方法返回值应该是 0，1，2。

如果 setNumReduceTasks(m);而 getPartition 方法的返回值有 n 种可能，则：

- 如果 m>n，则会开启 m 个 Reduce，但是只有 n 个 Reduce 能够获取数据并处理。
- 如果 m=n，理想状况，这样可以合理利用资源，负载更均衡。
- 如果 m<n，则会抛出异常。

m 个 Reduce 会产生 m 个 part 输出结果文件。

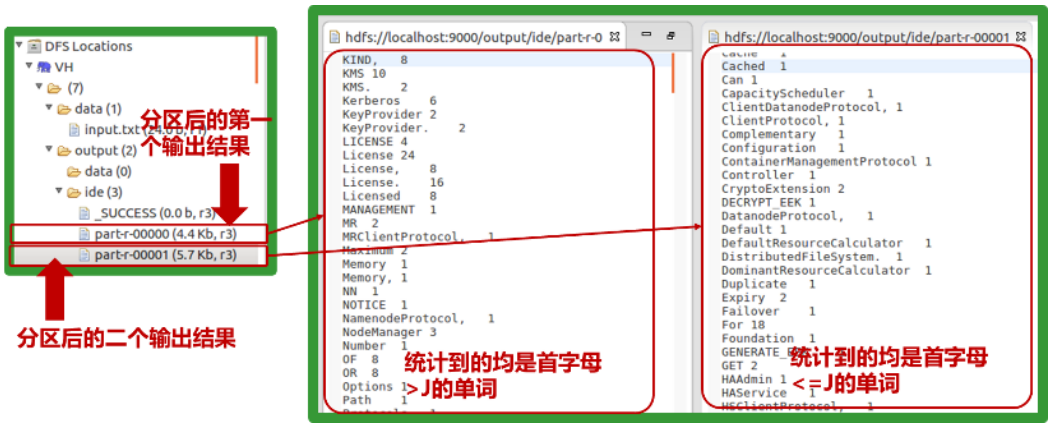
例如可以为 WordCount 案例提供自定义的分区规则，规定整个 WordCount 计算词频的 Reduce 操作由两个 Reduce 节点共同实现，Map 函数输出的中间结果中的键（即统计到的单词本身）的第一个字母作为分区的依据，如果第一个字母的编码大于字母 J 的编码，则交由第 1 个 Reduce 节点处理，反之则交给第 2 个 Reduce 节点处理，对应的分区规则参考实现如下：

```
public static class WordCountPartitioner extends
        Partitioner<Text, IntWritable> {
```

```
    @Override
    public int getPartition(Text arg0, IntWritable arg1, int arg2)
{
        // TODO Auto-generated method stub
        char ch = arg0.toString().charAt(0);
        if (ch > 'J') {
            return 0;
        } else {
            return 1;
        }
    }

}
```

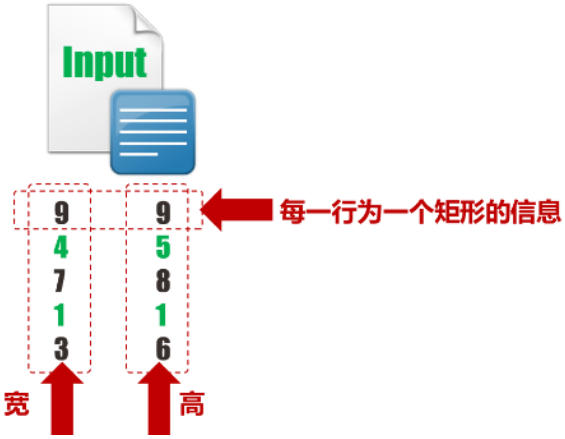
接下来在任务调度中添加设置 Reduce 数量和分区规则的代码：

```
job.setNumReduceTasks(2);

job.setPartitionerClass(WordCountPartitioner.class);
```



完整代码实现参考本教材的随书代码。

分区和序列化可以同时使用，实现更为灵活的任务调度控制。例如有如下的原始数据：



要求按照如下的规则进行统计处理：实现矩形按面积升序排列，长方形和正方形分区统计，并存入两个不同的结果文件。

因为要按照矩形的面积进行排序，因此矩形的信息必须定义为一个自定义的序列化类型并作为 MapReduce 任务中 Map 函数的输出 Key 类型和 Reduce 函数的输入 Key 类型，在该序列化类型中需要定义矩形的宽和高、序列化、反序列化的方法以及对应的比较方法，该序列化类型的参考实现如下：

```
class RectangleWritable implements WritableComparable {

    int length, width;

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public RectangleWritable() {
        super();
        // TODO Auto-generated constructor stub
    }

    public RectangleWritable(int length, int width) {
```

```
        super();
        this.length = length;
        this.width = width;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        // TODO Auto-generated method stub
        out.writeInt(length);
        out.writeInt(width);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        // TODO Auto-generated method stub
        this.length = in.readInt();
        this.width = in.readInt();

    }

    @Override
    public int compareTo(Object o) {
        // TODO Auto-generated method stub
        RectangleWritable to = (RectangleWritable) o;
        if (this.getLength() * this.getWidth() > to.getLength() *
to.getWidth())
            return 1;
        if (this.getLength() * this.getWidth() < to.getLength() *
to.getWidth())
            return -1;
        return 0;
    }


}
```

接下来实现判定一个矩形是否为正方形并交由特定的 Reduce 任务处理的分区规则：

```
class MyPatitioner extends Partitioner<RectangleWritable,
NullWritable> {
    @Override
    public int getPartition(RectangleWritable k2, NullWritable v2,
            int numReduceTasks) {
```

```
        if (k2.getLength() == k2.getWidth()) {
            return 0; // 正方式在任务0中汇总
        } else
            return 1;// 长方式在任务1中汇总
    }

}
```

Map 函数的参考实现如下：

```
    static class MyMapper extends
            Mapper<LongWritable, Text, RectangleWritable,
NullWritable> {

        protected void map(LongWritable k1, Text v1, Context
context)
                throws IOException, InterruptedException {
            String[] splites = v1.toString().split(" ");
            RectangleWritable k2 = new RectangleWritable(
                    Integer.parseInt(splites[0]),
Integer.parseInt(splites[1]));
            context.write(k2, NullWritable.get());
        };

    }
```

Reduce 函数的参考实现如下：

```
    static class MyReducer extends
            Reducer<RectangleWritable, NullWritable, IntWritable,
IntWritable> {
        protected void reduce(RectangleWritable k2,
Iterable<NullWritable> v2s,
                Context context) throws IOException,
InterruptedException {
            context.write(new IntWritable(k2.getLength()),
                    new IntWritable(k2.getWidth()));
        };

    }
```

最后注意在任务调度代码中确定 Reduce 任务的数量和设置分区规则：

```
job.setPartitionerClass(MyPatitioner.class);

job.setNumReduceTasks(2);
```

程序的最终运行结果如下：

完整代码实现参考本教材的随书代码。

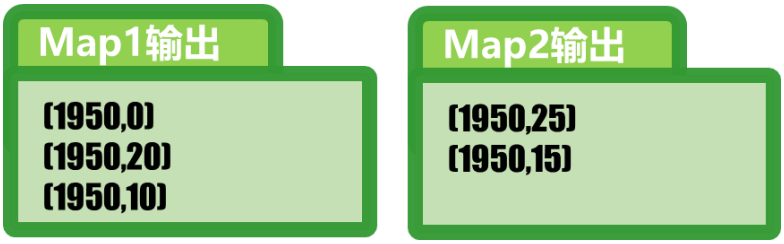
第6节 Combiner

集群上的可用带宽限制了 MapReduce 作业的数量，因此尽量避免 map 和 reduce 任务之间的数据传输是有利的。

Hadoop 允许用户针对 map 任务的输出指定一个 combiner（就像 mapper 和 reducer 一样）-combiner 函数的输出作为 reduce 函数的输入。

由于 combiner 属于优化方案，所以 Hadoop 无法确定要对 map 任务输出记录调用多少次 combiner（如果需要）。换而言之，不管调用 combiner 多少次，0 次、1 次或多次，reducer 的输出结果都必须是一样的。

以获取每年最高气温的应用为例，假设两个 map 的输出如下：



Reduce 方法执行时，输入如下：

因为其中 25 最大，因此最终输出的结果是(1950，25)。

可以像使用 reduce 函数那样，使用 Combiner 找出每个 map 任务输出结果中的最高气温。如此一来，reduce 函数调用时将被传入以下数据：

reduce 输出的结果和以前一样。可以通过下面的表达式来说明气温数值的方法调用：

```
max(0,20,10,25,15）=max(max(0,20,10)，max(25,15)）=max(20,25)=25
```

在一般情况下,Combiner 可以直接使用 Reduce 函数,因为它实现了和 Reduce 一样的数据汇总操作，差异只在于 Reduce 函数在专门的 Reduce 计算节点中完成，而 Combiner 在 Map 节点中完成。要设置 Combiner 处理器，在 MapReduce 的任务调度代码中调用一下代码即可：

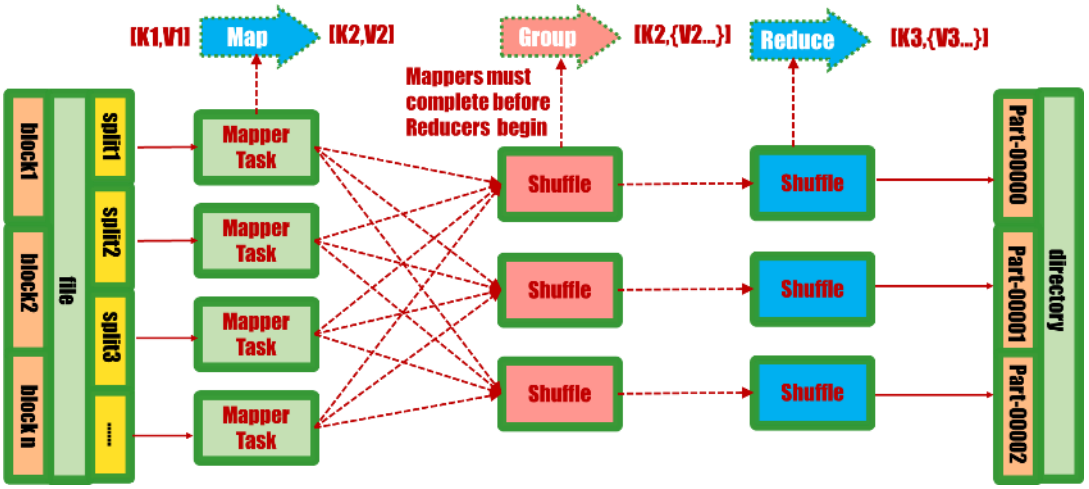
```
Job.setCombinerClass(Combiner 处理器类，一般情况下直接使用 Reduce);
```

注意：并非所有的场景都适用于这种方法，如计算平均气温，就不能用平均数作为 combiner，因为 avg(0,20,10,25,15）=14，而 avg(avg(0,20,10)，avg(25,15)）=avg(10,20)=15。

第7节 MapReduce 过程进阶

MapReduce 中，Map 阶段处理的数据如何传递给 Reduce 阶段，是整个 MapReduce 框架中最关键的一个流程，这个流程就叫 shuffl。它的核心机制包括数据分区，排序和缓存等。

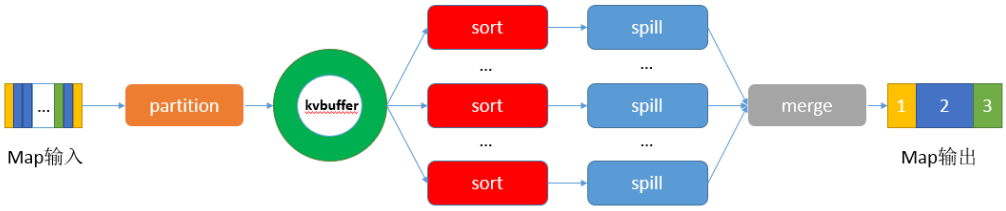
Map 是映射，负责数据的过滤分法，将原始数据转化为键值对；Reduce 是合并，将具有相同 key 值的 value 进行处理后再输出新的键值对作为最终结果。为了让 Reduce 可以并行处理 Map 的结果，必须对 Map 的输出进行一定的排序与分割,然后再交给对应的 Reduce,而这个将 Map 输出进行进一步整理并交给 Reduce 的过程就是 Shuffle。整个 MR 的大致过程如下：



Map 和 Reduce 操作需要开发人员自己定义相应 Map 类和 Reduce 类，以完成所需要的化简、合并操作，而 shuffle 则是系统自动实现的，了解 shuffle 的具体流程能帮助开发人员编写出更加高效的 MapReduce 程序。

Shuffle过程包含在Map和Reduce两端，即Map shuffle和Reduce shuffle。

在 Map 端的 shuffle 过程是对 Map 的结果进行分区、排序、分割，然后将属于同一划分（分区）的输出合并在一起并写在磁盘上，最终得到一个分区有序的文件，分区有序的含义是 map 输出的键值对按分区进行排列，具有相同 partition 值的键值对存储在一起，每个分区里面的键值对又按 key 值进行升序排列（默认），其流程大致如下：



对于 map 输出的每一个键值对，系统都会给定一个 partition，partition 值默认是通过计算 key 的 hash 值后对 Reduce task 的数量取模获得。如果一个键值对的 partition 值为 0，意味着这个键值对会交给第一个 Reducer 处理。

每一个 Reduce 的输出都是有序的，但是将所有 Reduce 的输出合并到一起却并非是全局有序的，如果要做到全局有序，该怎么做呢？最简单的方式，只设置一个 Reduce 计算节点，但是这样完全发挥不出集群的优势，而且能应对的数据量也很受限。最佳的方式是自己定义一个 Partitioner，用输入数据的最大值除以系统 Reduce 任务数量的商作为分割边界，也就是说分割数据的边界为此商的 1 倍、2 倍至 numPartitions-1 倍，这样就能保证执行分区后的数据是整体有序的。

另一种需要自己定义一个 Partitioner 的情况是各个 Reduce 任务处理的键值对数量极不平衡。对于某些数据集，由于很多不同的 key 的 hash 值都一样，

导致这些键值对都被分给同一个 Reducer 处理，而其他的 Reducer 处理的键值对很少，从而拖延整个任务的进度。当然，编写自己的 Partitioner 必须要保证具有相同 key 值的键值对分发到同一个 Reducer。

Map 的输出结果是由 collector 处理的，每个 Map 任务不断地将键值对输出到在内存中构造的一个环形数据结构中。使用环形数据结构是为了更有效地使用内存空间，在内存中放置尽可能多的数据。

这个数据结构其实就是个字节数组，叫 Kvbuffer，名如其义，但是这里面不光放置了数据，还放置了一些索引数据，给放置索引数据的区域起了一个 Kvmeta 的别名，在 Kvbuffer 的一块区域上穿了一个 IntBuffer（字节序采用的是平台自身的字节序）的马甲。数据区域和索引数据区域在 Kvbuffer 中是相邻不重叠的两个区域，用一个分界点来划分两者，分界点不是亘古不变的，而是每次 Spill 之后都会更新一次。初始的分界点是 0，数据的存储方向是向上增长，索引数据的存储方向是向下增长。

Kvbuffer 的存放指针 bufindex 是一直闷着头地向上增长，比如 bufindex 初始值为 0，一个 Int 型的 key 写完之后，bufindex 增长为 4，一个 Int 型的 value 写完之后，bufindex 增长为 8。

索引是对在 kvbuffer 中的键值对的索引，是个四元组，包括：value 的起始位置、key 的起始位置、partition 值、value 的长度，占用四个 Int 长度，Kvmeta 的存放指针 Kvindex 每次都是向下跳四个“格子”，然后再向上一个格子一个格子地填充四元组的数据。比如 Kvindex 初始位置是-4，当第一个键值对写完之后，(Kvindex+0)的位置存放 value 的起始位置、(Kvindex+1)的位置存放 key 的起始位置、(Kvindex+2)的位置存放 partition 的值、(Kvindex+3)的位置存放 value 的长度，然后 Kvindex 跳到-8 位置，等第二个键值对和索引写完之后，Kvindex 跳到-12 位置。

Kvbuffer 的大小可以通过 io.sort.mb 设置，默认大小为 100M。但不管怎么设置，Kvbuffer 的容量都是有限的，键值对和索引不断地增加，加着加着，Kvbuffer 总有不够用的那天，那怎么办？把数据从内存刷到磁盘上再接着往内存写数据，把 Kvbuffer 中的数据刷到磁盘上的过程就叫 Spill，多么明了的叫法，内存中的数据满了就自动地 spill 到具有更大空间的磁盘。

关于 Spill 触发的条件，也就是 Kvbuffer 用到什么程度开始 Spill，还是要讲究一下的。如果把 Kvbuffer 用得死死得，一点缝都不剩的时候再开始 Spill，那 Map 任务就需要等 Spill 完成腾出空间之后才能继续写数据；如果 Kvbuffer 只是满到一定程度，比如 80%的时候就开始 Spill，那在 Spill 的同时，Map 任务还能继续写数据，如果 Spill 够快，Map 可能都不需要为空闲空间而发愁。两利相衡取其大，一般选择后者。Spill 的门限可以通过 io.sort.spill.percent，默认是 0.8。

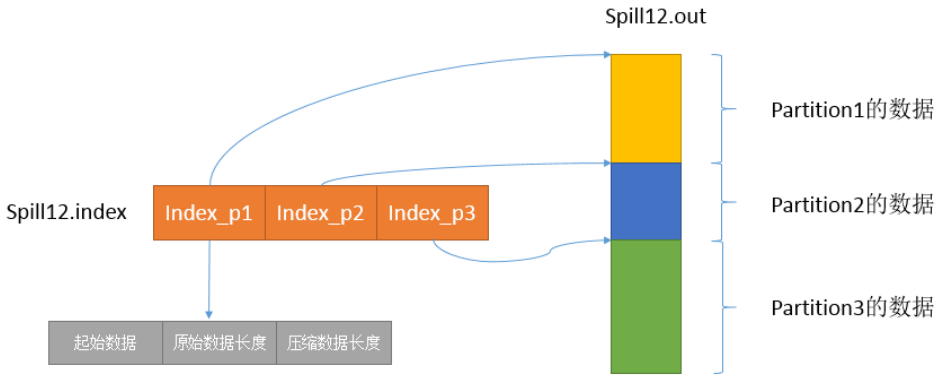
Spill 这个重要的过程是由 Spill 线程承担，Spill 线程从 Map 任务接到“命令”之后就开始正式干活，干的活叫 SortAndSpill，原来不仅仅是 Spill，在 Spill 之前还有个颇具争议性的 Sort。

当 Spill 触发后，SortAndSpill 先把 Kvbuffer 中的数据按照 partition 值

和 key 两个关键字升序排序，移动的只是索引数据，排序结果是 Kvmeta 中数据按照 partition 为单位聚集在一起，同一 partition 内的按照 key 有序。

Spill 线程为这次 Spill 过程创建一个磁盘文件：从所有的本地目录中轮训查找能存储这么大空间的目录，找到之后在其中创建一个类似于“spill12.out”的文件。Spill 线程根据排过序的 Kvmeta 挨个 partition 的把数据吐到这个文件中，一个 partition 对应的数据吐完之后顺序地吐下个 partition，直到把所有的 partition 遍历完。一个 partition 在文件中对应的数据也叫段(segment)。在这个过程中如果用户配置了 combiner 类，那么在写之前会先调用 combineAndSpill()，对结果进行进一步合并后再写出。Combiner 会优化 MapReduce 的中间结果，所以它在整个模型中会多次使用。那哪些场景才能使用 Combiner 呢？Combiner 的输出是 Reducer 的输入，Combiner 绝不能改变最终的计算结果。所以从我的想法来看，Combiner 只应该用于那种 Reduce 的输入 key/value 与输出 key/value 类型完全一致，且不影响最终结果的场景。比如累加，最大值等。Combiner 的使用一定得慎重，如果用好，它对 job 执行效率有帮助，反之会影响 reduce 的最终结果。

所有的 partition 对应的数据都放在这个文件里，虽然是顺序存放的，但是怎么直接知道某个 partition 在这个文件中存放的起始位置呢？强大的索引又出场了。有一个三元组记录某个 partition 对应的数据在这个文件中的索引：起始位置、原始数据长度、压缩之后的数据长度，一个 partition 对应一个三元组。然后把这些索引信息存放在内存中，如果内存中放不下了，后续的索引信息就需要写到磁盘文件中了：从所有的本地目录中轮训查找能存储这么大空间的目录，找到之后在其中创建一个类似于“spill12.out.index”的文件，文件中不光存储了索引数据，还存储了 crc32 的校验数据。spill12.out.index 不一定在磁盘上创建，如果内存（默认 1M 空间）中能放得下就放在内存中，即使在磁盘上创建了，和 spill12.out 文件也不一定在同一个目录下。每一次 Spill 过程就会最少生成一个 out 文件，有时还会生成 index 文件，Spill 的次数也烙印在文件名中。索引文件和数据文件的对应关系如下图所示：



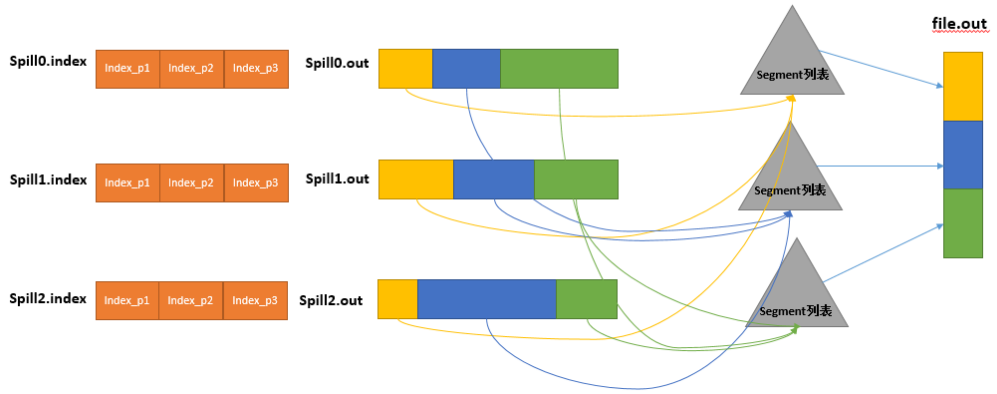
在 Spill 线程如火如荼的进行 SortAndSpill 工作的同时，Map 任务不会因此而停歇，而是一无既往地进行着数据输出。Map 还是把数据写到 kvbuffer 中，那问题就来了：只顾着闷头按照 bufindex 指针向上增长，kvmeta 只顾着按照 Kvindex 向下增长，是保持指针起始位置不变继续跑呢，还是另谋它路？如果保

持指针起始位置不变，很快 bufindex 和 Kvindex 就碰头了，碰头之后再重新开始或者移动内存都比较麻烦，不可取。Map 取 kvbuffer 中剩余空间的中间位置，用这个位置设置为新的分界点，bufindex 指针移动到这个分界点，Kvindex 移动到这个分界点的-16 位置，然后两者就可以和谐地按照自己既定的轨迹放置数据了，当 Spill 完成，空间腾出之后，不需要做任何改动继续前进。

Map 任务总要把输出的数据写到磁盘上，即使输出数据量很小在内存中全部能装得下，在最后也会把数据刷到磁盘上。

Map 任务如果输出数据量很大，可能会进行好几次 Spill，out 文件和 Index 文件会产生很多，分布在不同的磁盘上。最后把这些文件进行合并的 merge 过程发挥了重要的作用。

Merge 过程怎么知道产生的 Spill 文件都在哪了呢？从所有的本地目录上扫描得到产生的 Spill 文件，然后把路径存储在一个数组里。Merge 过程又怎么知道 Spill 的索引信息呢？没错，也是从所有的本地目录上扫描得到 Index 文件，然后把索引信息存储在一个列表里。到这里，又遇到了一个值得纳闷的地方。在之前 Spill 过程中的时候为什么不直接把这些信息存储在内存中呢，何必又多了这步扫描的操作？特别是 Spill 的索引数据，之前当内存超限之后就把数据写到磁盘，现在又要从磁盘把这些数据读出来，还是需要装到更多的内存中。之所以多此一举，是因为这时 kvbuffer 这个内存大户已经不再使用可以回收，由内存空间来装这些数据了。



然后为 merge 过程创建一个叫 file.out 的文件和一个叫 file.out.Index 的文件用来存储最终的输出和索引，一个 partition 一个 partition 的进行合并输出。对于某个 partition 来说，从索引列表中查询这个 partition 对应的所有索引信息，每个对应一个段插入到段列表中。也就是这个 partition 对应一个段列表，记录所有的 Spill 文件中对应的这个 partition 那段数据的文件名、起始位置、长度等等。

然后对这个 partition 对应的所有的 segment 进行合并，目标是合并成一个 segment。当这个 partition 对应很多个 segment 时，会分批地进行合并：先从 segment 列表中把第一批取出来，以 key 为关键字放置成最小堆，然后从最小堆中每次取出最小的输出到一个临时文件中，这样就把这一批段合并成一个临时的

段，把它加回到 segment 列表中；再从 segment 列表中把第二批取出来合并输出到一个临时 segment，把其加入到列表中；这样往复执行，直到剩下的段是一批，输出到最终的文件中。最终的索引数据仍然输出到 Index 文件中。

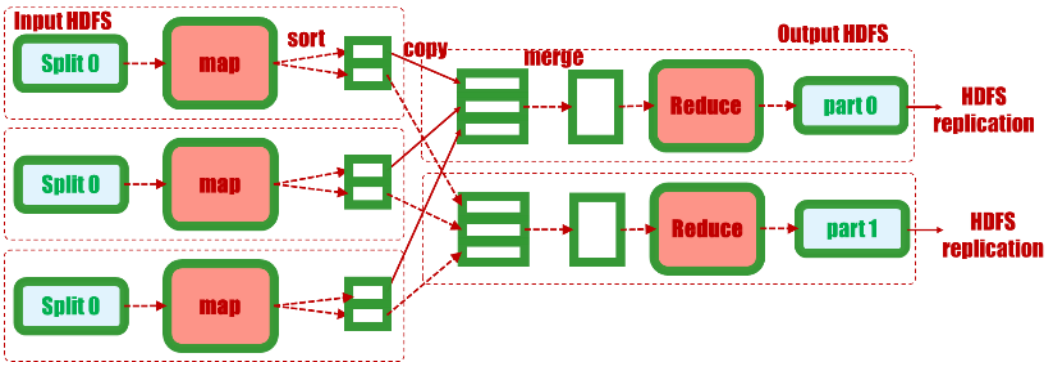
在 Reduce 端，shuffle 主要分为复制 Map 输出、排序合并两个阶段。

Reduce 任务通过 HTTP 向各个 Map 任务拖取它所需要的数据。Map 任务成功完成后，会通知父 TaskTracker 状态已经更新，TaskTracker 进而通知 JobTracker（这些通知在心跳机制中进行）。所以，对于指定作业来说，JobTracker 能记录 Map 输出和 TaskTracker 的映射关系。Reduce 会定期向 JobTracker 获取 Map 的输出位置，一旦拿到输出位置，Reduce 任务就会从此输出对应的 TaskTracker 上复制输出到本地，而不会等到所有的 Map 任务结束。

获取过来的数据会先放入内存缓冲区中，如果内存缓冲区中能放得下这次数据的话就直接把数据写到内存中，即内存到内存 merge。Reduce 要向每个 Map 去拖取数据，在内存中每个 Map 对应一块数据，当内存缓存区中存储的 Map 数据占用空间达到一定程度的时候，开始启动内存中 merge，把内存中的数据 merge 输出到磁盘上一个文件中，即内存到磁盘 merge。在将 buffer 中多个 map 输出合并写入磁盘之前，如果设置了 Combiner，则会化简压缩合并的 map 输出。Reduce 的内存缓冲区可通过 mapred.job.shuffle.input.buffer.percent 配置，默认是 JVM 的 heap size 的 70%。内存到磁盘 merge 的启动门限可以通过 mapred.job.shuffle.merge.percent 配置，默认是 66%。

当属于该 reducer 的 map 输出全部拷贝完成，则会在 reducer 上生成多个文件（如果拖取的所有 map 数据总量都没有内存缓冲区，则数据就只存在于内存中），这时开始执行合并操作，即磁盘到磁盘 merge，Map 的输出数据已经是有序的，Merge 进行一次合并排序，所谓 Reduce 端的 sort 过程就是这个合并的过程。一般 Reduce 是一边 copy 一边 sort，即 copy 和 sort 两个阶段是重叠而不是完全分开的。最终 Reduce shuffle 过程会输出一个整体有序的数据块。

由此，MapReduce 的整体执行流程可以总结如下：



第8节 MapReduce2 和 MapReduce1 的差异

新旧版本的 MapReduce API 有如下几个明显的差异：

编程接口

在旧 API 中使用 Mapper 和 Reducer 接口，新 API 倾向于使用抽象类，而不是接口，因为更有利于扩展。这意味着用不着修改类的实现，即可在抽象类中添加一个方法（即默认的实现）。

API 位置

新 API 放在 org.apache.hadoop.mapreduce 包（和子包）中,之前版本的 API 依旧放在 org.apache.hadoop .hadoop.mapred 中。

上下文

新 API 充分使用上下文对象，使用户代码能与 MapReduce 系统通信。例如，新的 context 基本统一了旧 API 中的 JobConf、OutputCollector 和 Reporter 的功能。

流程控制

键 / 值对记录在这两类 API 中都被推给 Mapper 和 Reducer，但除此之外，新的 API 通过重写 run() 方法允许 mapper 和 reducer 控制执行流程。例如，既可以批处理记录，也可以在处理完所有的记录之前停止。在旧 API 中可以通过写 MapRunnable 类在 Mapper 中实现上述功能,但是在 Reducer 中没有对等的实现。

作业控制

新的 API 中作业控制由 Job 类实现，而非旧 API 中的 Jobclient 类，新的 API 中删除了 JobClient 类。

配置

新增的 API 实现了配置的统一。旧 API 通过一个特殊的 JobConf 对象配置作业，该对象是 Hadoop 配置对象的一个扩展（用于配置守护进程）。在新 API 中，作业的配置由 Configuration（或许通过 Job 类中的一些辅助方法）来完成。

输出

输出文件的命名方式稍有不同。在旧的 API 中 map 和 reduce 的输出被统一命名为 part 一 nnmm，但是在新 API 中 map 的输出文件名为 part-m-nnnnn，而 reduce 的输出文件名为 part-r-nnnnn（其中 nnnn 是从 0 开始的表示分块序号的整数）。

异常

新 API 中的用户重载函数被声明为抛出异常 java.lang.InterruptedException。这意味着可以用代码来实现中断响应，从而使该框架在必要时可以优雅地取消需长时间运行的作业。

列表值

在新的 API 中，reduce() 传递的值是 java.lang.lterable 类型的，而非 java.lang.lterator 类型（旧 API 中传递该类型的值）。这一改变使我们更容易通过 Java 的 for-each 循环结构来迭代这些值：

```
for（元素类型：列表）{…}
```

将 Mapper 和 Reducer 类转换为新 API 时，记住将 map() 和 reduce() 的签名转换为新形式。如果只是将类的继承修改为对新的 Mapper 和 Reducer 类的继承，编译的时候不会报错或显示等告信息，因为新的 Mapper 和 Reducer 类也同样提供了等价的 map() 和 reduce() 方法。但是，自己写的 mapper 或 reducer 代码是不会被调用的，这会导致难以诊断的错误。

对 map() 和 reduce() 方法添加@Override 注释，Java 编译器会发现这些错误。

第9节 MapReduce 的 IO 操作

Hadoop 自带一套原子操作用于数据 I/O 操作。其中有一些技术比 Hadoop 本身更常用，如数据完整性保持和压缩，但在处理多达好几个 TB 的数据集时，特别值得关注。其他一些则是 Hadoop 工其或 API，它们所形成的构建模块可用于开发分布式系统，比如序列化操作和在盘（on-disk）数据结构。

Hadoop 用户肯定都希望系统在存储和处理数据时不会丢失或损坏任何数据。尽管磁盘或网络上的每个 I/O 操作不太可能将错误引入自己正在读 / 写的数据中，但是如果系统中需要处理的数据量大到 Hadoop 的处理极限时，数据被损坏的概率还是很高的。

检测数据是否损坏的常见措施是，在数据第一次引入系统时计算校验和（checksum）并在数据通过一个不可靠的通道进行传输时再次计算校验和，这样就能发现数据是否损坏。如果计算所得的新校验和与原来的校验和不匹配，我们就认为数据已损坏。但该技术并不能修复数据，它只能检测出数据错误。（这正是不使用低端硬件的原因。具体说来一定要使用 ECC 内存。）注意，校验和也是可能损坏的，不只是数据，但由于校验和比数据小得多，所以损坏的可能性非常小。常用的错误检测码是 CRC-32（循环冗余校验），任何大小的数据输入均计算得到一个 32 位的整数校验和。

HDFS 会对写入的所有数据计算校验和，并在读取数据时验证校验和。它针对每个由 io.bytes.per.checksum 指定字节的数据计算校验和。默认情况下为 512 个字节，由于 CRC-32 校验和是 4 个字节，所以存储校验和的额外开销低于 1%。

HDFS 的数据校验在两个位置处发挥作用。

datanode 负责在收到数据后存储该数据及其验证校验和。它在收到客户端的数据或复制其他 datanode 的数据时执行这个操作。正在写数据的客户端将数据及其校验和发送到由一系列 datanode 组成的管线，管线中最后一个 datanode 负责验证校验和。如果 datanode 检测到错误，客户端便会收到一个 ChecksumException 异常，它是 IOException 异常的一个子类，后者应以应用程序特定的方式来处理，比如重试这个操作。

客户端从 datanode 读取数据时，也会验证校验和，将它们与 datanode 中存储的校验和进行比较。每个 datanode 均持久保存有一个用于验证的校验和日志

（persistent log of checksum verification），所以它知道每个数据块的最后一次验证时间。客户端成功验证一个数据块后，会告诉这个 datanode,datanode 由此更新日志。保存这些统计信息对于检测损坏的磁盘很有价值。

不只是客户端在读取数据块时会验证校验和，每个 datanode 也会在一个后台线程中运行一个 DataBlockscanner，从而定期验证存储在这个 datanode 上的所有数据块。该项措施是解决物理存储煤体上位损坏的有力措施。

由于 HDFS 存储着每个数据块的复本（replica），因此它可以通过数据复本来修复损坏的数据块，进而得到一个新的、完好无损的复本。基本思路是：客户端在读取数据块时，如果检测到错误，首先向 namenode 报告已损坏的数据块及其正在尝试读操作的这个 datanode，再抛出 checksumException 异常。namenode 将这个数据块复本标记为已损坏，因此，它不会将处理请求直接发送到这个节点，或尝试将这个复本复制到另一个 datanode。之后，它安排这个数据块的一个复本复制到另一个 datanode，如此一来，数据块的复本因子（replication factor）又回到期望水平。此后，已损坏的数据块复本便被删除。

Hadoop 的 LocalFileSystem 执行客户端的校验和验证。这意味着在写入一个名为 filename 的文件时，文件系统客户端会明确地在包含每个文件块校验和的同一个目录内新建一个名为 filename.crc 的隐藏文件。就像 HDFS 一样，文件块的大小由属性 io.bytes.per.checksum 控制，默认为 512 个字节。文件块的大小作为元数据存储在．crc 文件中，所以即使文件块大小的设置已经发生变化，仍然可以正确读回文件。在读取文件时需要验证校验和，并且如果检测到错误，LocalFileSystem 还会抛出一个 checksumException 异常：

校验和的计算代价是相当低的（在 Java 中，它们是用本地代码实现的），一般只是增加少许额外的读 / 写文件时间。对大多数应用来说，付出这样的额外开销以保证数据完整性是可以接受的。

也可以禁用校验和计算，特别是在底层文件系统本身就支持校验和的时候。在这种情况下，使用 RawLocalFilesystem 替代 LocalFilesystem。

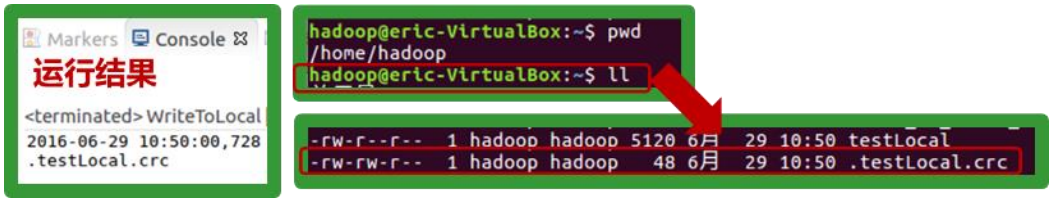
要想在一个应用中实现全局校验和验证，需要将 fs.file.impl 属性设置为 org.apache.hadoop.fs.RawLcalFilesystem 进而实现对文件 URI 的重新映射。还有一个可选方案可以直接新建一个 RawLocalFileSystem 实例。如果想针对一些读操作禁用校验和，这个方案非常有用。

通过 Hadoop LocalFileSystem 向本地文件系统写入带有校验的文件的示例如下：

```
public class WriteToLocal {

    public static void main(String[] args) throws IOException,
```

```
URISyntaxException {
        Configuration conf = new Configuration();
       //利用LocalFileSystem构建本地文件，构建成功后会添加.crc文件
        LocalFileSystem fs =  new LocalFileSystem(new
RawLocalFileSystem());
        fs.initialize(new URI("file:///home/hadoop/testLocal"),
conf); // put the conf object to filesystem instance
        OutputStream out = fs.create(new
Path("file:///home/hadoop/testLocal"));
        for(int i = 0; i < 512*10;i++){
            out.write('a');
        }
        out.close();
        //获取校验文件信息
        Path file = fs.getChecksumFile(new
Path("file:///home/hadoop/testLocal"));
        System.out.println(file.getName());
        fs.close();
    }

}
```

程序执行结果如下：



通过 Hadoop LocalFileSystem 向读取该文件时即可实现校验功能，如果校验失败，将会抛出异常。

LocalFileSystem 通过 ChecksumFilesystem 来完成自己的任务，有了这个类，向其他文件系统（无校验和系统）加入校验和就非常简单，因为 ChecksumFilesystem 类继承自 Filesystem 类。

一般用法如下：

```
//构建一个原生的不带有校验的文件系统接口对象
FileSystem rawFs = …
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

底层文件系统称为“源”(raw）文件系统，可以使用 ChecksumFileSystem 实例的 getRawFileSystem() 方法获取它。ChecksumFileSystem 类还有其他一些与校验和有关的有用方法，比如上一个案例中的 getChecksumFile() 可以获得任意一个文件的校验和文件路径。

如果 ChecksumFileSystem 类在读取文件时检测到错误，会调用自己的 reportChecksumFailure() 方法。默认实现为空方法，但 LocalFileSystem 类会将这个出错的文件及其校验和移到同一存储设备上一个名为 bad_files 的边际文件夹(side directory)中。管理员应定期检查这些坏文件并采取相应的行动。

第10节　MapReduce 的数据压缩

在处理大量数据时对数据进行压缩相当重要，它能够减少存储文件所需要的磁盘空间并加速数据在网络和磁盘上的传输。在 Hadoop 中支持很多常见的压缩格式、工具和算法，它们各有千秋：

压缩格式	工具	算法	文件扩展名	是否可切分
DEFLATE	无	DEFLATE	.deflate	否
Gzip	gzip	DEFLATE	.gz	否
bzip2	bzip2	bzip2	.bz2	是
LZO	lzop	LZO	.lzo	否
LZ4	无	LZ4	.lz4	否
Snappy	无	Snappy	.snappy	否

DEFLATE 是一个标准压缩算法。标准实现是 zlib。没有可用于生成 DEFLATE 文件的常用命令行工具，因为通常都用 gzip 格式。gzip 文件格式只是在 DEFLATE 格式上增加了文件头和一个文件尾。.deflate 文件扩展名是 Hadoop 约定的。

所有压缩算法都需要权衡空间 / 时间：压缩和解压缩速度更快，其代价通常是只能节省少量的空间。压缩工具都提供 9 个不同的选项来控制压缩时必须考虑的权衡：选项-1 为优化压缩速度，-9 为优化压缩空间。

例如，下述命令通过最快的压缩方法创建一个名为 file.gz 的压缩文件：

```
gzip -1 file
```

不同压缩工具有不同的压缩特性。gzip 是一个通用的压缩工具，在空间 / 时间性能的权衡中，居于其他两个压缩方法之间。bzip2 的压缩能力强于 gzip，但压缩速度更慢一点。尽管 bzip2 的解压速度比压缩速度快，但仍比其他压缩格式要慢一些。另一方面，LZO、LZ4 和 Snappy 均优化压缩速度，其速度比 gzip 快一个数量级，但压缩效率稍逊一筹。Snappy 和 LZ4 的解压缩速度比 LZO 高出很多

“是否可切分”列表示对应的压缩算法是否支持切分(splitable)，也就是说，是否可以搜索数据流的任意位置并进一步往下读取数据。可切分压缩格式尤其适合 MapReduce。

codec 实现了一种压缩一解压缩算法。

在 Hadoop 中，一个对 CompressionCodec 接口的实现代表一个 codec。例如，GzipCodec 包装了 gzip 的压缩和解压缩算法。以下列举了 Hadoop 实现的 codec：

压缩格式	Hadoop CompressionCodec

DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.Bzip2Codec
LZO	org.apache.hadoop.io.compress.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

CompressionCodec 包含两个函数，可以轻松用于压缩和解压缩数据。

createOutputStream(OutputStream out)

如果要对写入输出数据流的数据进行压缩，可用 createOutputStream(OutputStream out）方法在底层的数据流中对需要以压缩格式写入在此之前尚未压缩的数据新建一个 CompressionOutputStream 对象。

createInputStream(InputStream in)

对输入数据流中读取的数据进行解压缩的时候，则调用 createInputStream(InputStream in)获取 CompressionInputStream，可通过该方法从底层数据流读取解压缩后的数据。

CompressionOutputStream 和 CompressionInputStream，类似于 java.util.zip.DeflaterOutputStream 和 java.util.zip.DeflaterInputStream，只不过前两者能够重置其底层的压缩或解压缩方法，对于某些将部分数据流（Section of Data Stream）压缩为单独数据块（Bock）的应用-例如 SequenceFile，这个能力是非常重要的。

以下的代码展示了如何将原有的 MapReduce 输入文件内容写入一个压缩后的文件中：

```
public class StreamCompressor {
    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub

        Configuration conf = new Configuration();
        CompressionCodec codec = (CompressionCodec) ReflectionUtils

    .newInstance(org.apache.hadoop.io.compress.GzipCodec.class,
                        conf);
        LocalFileSystem fs = new LocalFileSystem(new
RawLocalFileSystem());
        fs.initialize(new
URI("file:///home/hadoop/testCompression"), conf);
        CompressionOutputStream out = codec.createOutputStream(fs
                .create(new
```

```
Path("file:///home/hadoop/testCompression.gz")));
        out.write("9 9\n".getBytes());
        out.write("4 5\n".getBytes());
        out.write("7 8\n".getBytes());
        out.write("1 1\n".getBytes());
        out.write("3 6\n".getBytes());
        out.flush();
        out.finish();
        out.close();

    }
}
```

程序运行结果如下：

对于大型文件，不要对整个文件使用不支持分割的压缩格式，因为这样会损失本地性优势，从而使降低 MapReduce 应用的性能。

如果输入的文件是压缩过的，那么在被 MapReduce 读取时，它们会被自动解压，根据文件扩展名来决定应该使用哪一个压缩解码器。

如果要压缩 MapReduce 作业的输出，请在作业配置文件中将 mapred.output.compress 属性设置为 true。将 mapred.output.compression.codec 属性设置为自己打算使用的压缩编码/解码器的类名：

```
FileOutputFormat.setCompressOutput(job, true); // job使用压缩
FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class); // 设置压缩格式
```

即使 MapReduce 应用使用非压缩的数据来读取和写入，我们也可以受益于压缩 map 阶段的中间输出。因为 map 作业的输出会被写入磁盘并通过网络传输到 reducer 节点，所以如果使用 LZO 之类的快速压缩，能得到更好的性能，因为传输的数据量大大减少了。

以下代码显示了启用 map 输出压缩和设置压缩格式的配置属性。

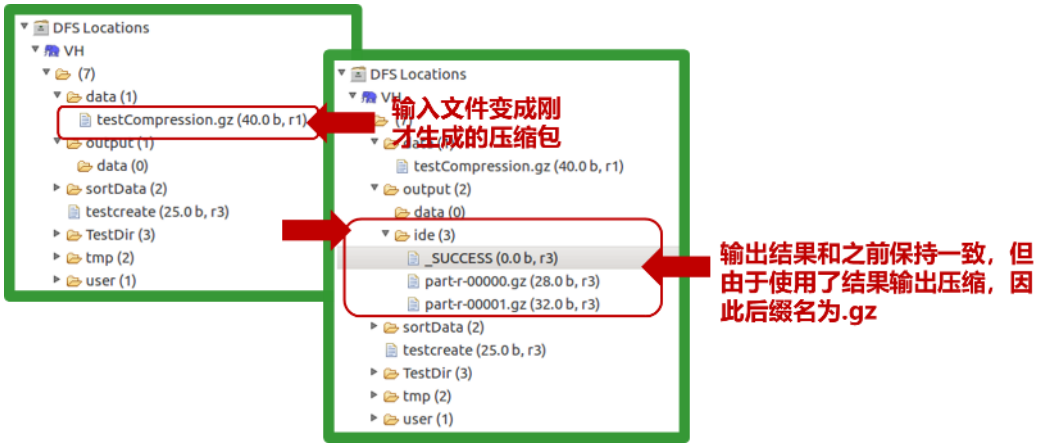
```
conf.setBoolean("mapred.compress.map.output", true);
conf.setClass("mapred.map.output. compression.codec", GzipCodec.class,
        CompressionCodec.class);
```

在旧的 API 中，可以使用更简单的方法来进行 map 作业输出结果的压缩：

```
conf.setCompressMapOutput(true);

conf.setMapOutputCompressorClass(GzipCodec.class);
```

当输入文件是压缩文件时，Hadoop 会根据后缀名自动识别压缩方式并解压，然后按照正常的 MR 流程运行。

将刚才生成的压缩文件作为前序章节中分区统计矩形并按照矩形面积排序案例的输入数据后直接执行，结果与直接使用非压缩的文本数据时执行的结果保持一致：

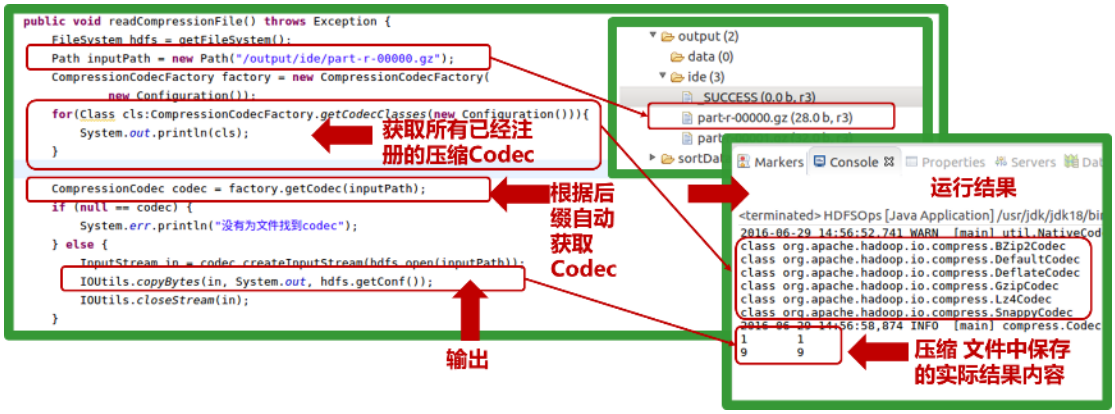


可以利用 Hadoop 提供的 Codec 读取案例生成的压缩结果文件，在解压过程中能够利用后缀名自动识别压缩方法：

```
    public void readCompressionFile() throws Exception {
        FileSystem hdfs = getFileSystem();
        Path inputPath = new Path("/output/ide/part-r-00000.gz");
        CompressionCodecFactory factory = new
CompressionCodecFactory(
                new Configuration());
        for (Class cls : CompressionCodecFactory
                .getCodecClasses(new Configuration())) {
            System.out.println(cls);
        }

        CompressionCodec codec = factory.getCodec(inputPath);
        if (null == codec) {
```

```
            System.err.println("没有为文件找到codec");
        } else {
            InputStream in =
codec.createInputStream(hdfs.open(inputPath));
            IOUtils.copyBytes(in, System.out, hdfs.getConf());
            IOUtils.closeStream(in);
        }

    }
```

调用该程序后的执行结果及代码解析如下：



为了性能，最好使用“原生”(native）类库来实现压缩和解压缩。例如，在一个测试中，使用原生 gzip 类库可以减少约一半的解压缩时间和约 10%的压缩时间（与内置的 Java 实现相比）。并非所有格式都有原生实现（如 bzip2），有些则只有原生类库实现（如 LZO）：

压缩格式	是否有 Java 实现	是否有原生实现
DEFLATE	是	是
gzip	是	是
bzip2	是	否
LZO	否	是
LZ4	否	是
Snappy	否	是

可以通过 Java 系统的 java.library.path 属性指定原生代码库。

bin 文件夹中的 hadoop 脚本可以设置该属性，但如果不用这个脚本，则需要在应用中手动设置该属性。

默认情况下，Hadoop 会根据自身运行的平台搜索原生代码库，如果找到相应的代码库就会自动加载。这意味着，你无需为了使用原生代码库而修改任何设置。但是，在某些情况下，例如调试一个压缩相关问题时，可能需要禁用原生代码库。将属性 hadoop.native.lib 的值设置成 false 即可，这可确保使用内置的 Java 代码库（如果有的话）。

如果使用的是原生代码库并且需要在应用中执行大量压缩和解压缩操作，可

以考虑使用 CodecPool，它支持反复使用压缩和解压缩，以分摊创建这些对象的开销。

```
    public void readCompressionFileWithCodecPool() throws Exception
{
        FileSystem hdfs = getFileSystem();
        CompressionCodec codec =
ReflectionUtils.newInstance(GzipCodec.class,
                hdfs.getConf());
        Compressor compressor = CodecPool.getCompressor(codec);
        InputStream in = codec.createInputStream(hdfs.open(new Path(
                "/output/ide/part-r-00000.gz")));
        IOUtils.copyBytes(in, System.out, hdfs.getConf());
        IOUtils.closeStream(in);
        CodecPool.returnCompressor(compressor);

    }
```

在考虑如何压缩将由 MapReduce 处理的数据时，理解这些压缩格式是否支持切分（splitting）是非常重要的。

以一个存储在 HDFS 文件系统中且压缩前大小为 1GB 的文件为例，如果 HDFS 的块大小设置为 64M，那么该文件将被存储在 16 个块中，把这个文件作为输入数据的 MapReduce 作业将创建 16 个数据块，其中每个数据块作为一个 Map 任务的输入。

经过 gzip 压缩后，文件大小为 1GB。与以前一样，HDFS 将这个文件保存为 16 个数据块。但是，将每个数据块单独作为一个输入分片是无法实现工作的，因为无法实现从 gzip 压缩数据流的任意位置读取数据，所以让 map 任务独立于其他任务进行数据读取是行不通的。

gzip 并不支持文件切分的原因主要有以下两个：

算法，gzip 格式使用 DEFLATE 算法来存储压缩后的数据，而 DEFLATE 算法将数据存储在一系列连续的压缩块中。

读取，从每个块的起始位置进行读取与从数据流的任意位置开始读取时一致并接着往后读取下一个数据块，因此需要与整个数据流进行同步。

在这种情况下，MaReduce 会采用正确的做法：

它不会尝试切分 gzip 压缩文件，因为它知道输入是 gzip 压缩文件（通过文件扩展名看出）且 gzip 不支持切分。这就牺牲了数据的本地性：一个 map 任务处理 16 个 HDFS 块，而其中大多数块并没有存储在执行该 map 任务的节点。而且，map 任务数越少，作业的粒度就较大，因而运行的时间可能会更长。

另一种压缩格式 bzip2 文件提供不同数据块之间的同步标识（Pi 的 48 位近似值）,因而它支持切分。

Hadoop 应用处理的数据集非常大，因此需要借助于压缩。使用哪种压缩格式

与待处理的文件的大小、格式和所使用的工具相关。下面有一些建议，大致是按照效率从高到低排列的：

使用容器文件格式

例如顺序文件、RCFile 或者 Avro 数据文件，所有这些文件格式同时支持压缩和切分。通常最好与一个快速压缩工具联合使用，例如 LZO,LZ4，或者 Snappy。

使用支持切分的压缩格式

例如 bzip2，或者使用通过索引实现切分的压缩格式。

在应用中将文件切分成块，并使用任意一种压缩格式为每个数据块建立压缩文件（不论它是否支持切分）

这种情况下，需要合理选择数据块的大小，以确保压缩后数据块的大小近似于 HDFS 块的大小。

第11节　容器格式-基于文件的数据结构

对于某些应用，开发人员需要一种特殊的数据结构来存储自己的数据。对于基于 MapReduce 的数据处理，将每个二进制数据大对象（blob）单独放在各自的文件中不能实现可扩展性，所以，Hadoop 为此开发了很多更高层次的容器。

如果要村塾日志文件，那么在一般情况下日志文件格式为：每一行文本代表一条日志记录。

但是标准的日志文件格式存在一个难以绕过的限制条件，那就是纯文本不合适记录二进制类型的数据。

在这种情况下，Hadoop 的 SequenceFile 类非常合适，它是二进制持久数据结构。将它作为日志文件的存储格式时，开发人员可以自己选择键（比如键 / 值对提供了一个 LongWritable 类型所表示的时间戳），以及值可以是 Writable 类型（用于表示日志记录的数量）。

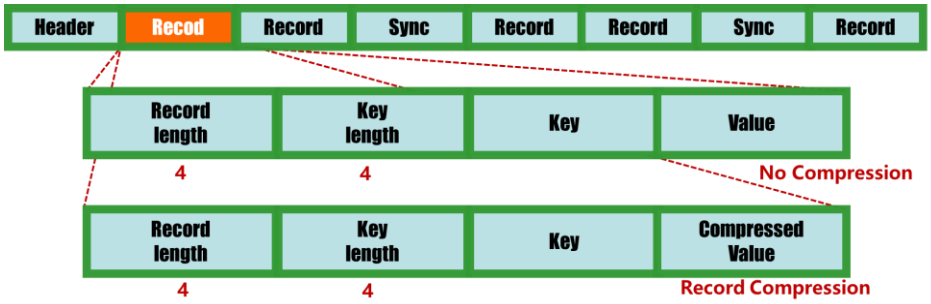
SequenceFiles 也可以作为小文件的容器。HDFS 和 MapReduce 是针对大文件优化的，所以通过 sequenceFile 类型将小文件包装起来，可以获得更高效率的存储和处理。

SequenceFile 文件格式描述：

- SequenceFile 由文件头和随后的一条或多条记录组成。
- 文件的前三个字节为 SEQ（顺序文件代码），紧随其后的一个字节表示顺序文件的版本号。
- 文件头还包括其他字段，例如键和值类的名称、数据压缩细节、用户定义的元数据以及同步标识。
- 同步标识用于在读取文件时能够从任意位置开始识别记录边界。每个文件都有一个随机生成的同步标识，其值存储在文件头中。

- 同步标识位于顺序文件中的记录与记录之间。同步标识的额外存储开销要求小于 1%，所以没有必要在每条记录末尾添加该标识（特别是比较短的记录）。

SequenceFile 文件格式可以用如下的图形来表示：

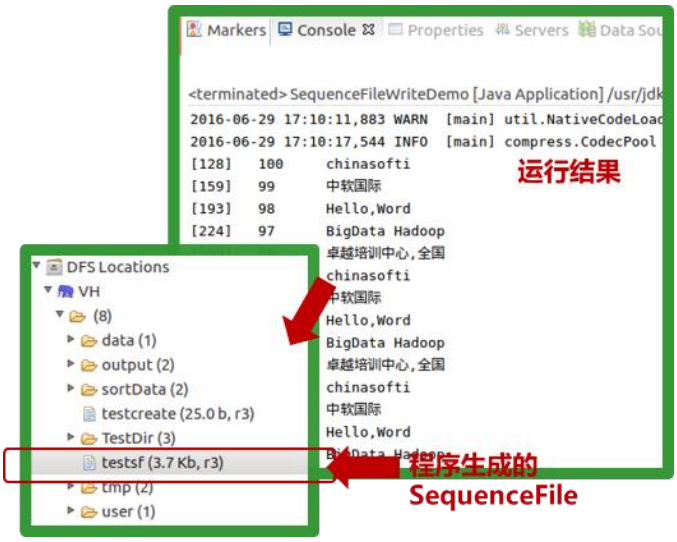


在 Hadoop 的 Java API 中，通过通过 SequenceFiles.createWriter() 静态方法可以创建 SequenceFile 对象，并返回 SequenceFile.Writer 实例。一旦拥有 SequenceFile. Writer 实例，就可以通过 append() 方法在文件末尾附加键 / 值对。写完后，可以调用 close() 方法（SequenceFile.Writer 实现了 java.io.Closeable 接口）。

通过 Java API 写于一个 SequenceFile 文件的参考方法如下：

```
public class SequenceFileWriteDemo {

    private static final String[] DATA = { "chinasofti", "中软国际
", "Hello,Word",
            "BigData Hadoop", "卓越培训中心,全国" };

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://127.0.0.1:9000/tt/testsf2";
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        Path path = new Path(uri);
        IntWritable key = new IntWritable();
        Text value = new Text();
        SequenceFile.Writer writer = null;
        try {
            writer = SequenceFile.createWriter(fs, conf, path,
key.getClass(),
                    value.getClass());
            for (int i = 0; i < 100; i++) {
                key.set(100 - i);
                value.set(DATA[i % DATA.length]);
```

```
                System.out.printf("[%s]\t%s\t%s\n",
writer.getLength(), key,
                        value);
                writer.append(key, value);
            }
        } finally {
            IOUtils.closeStream(writer);
        }
    }
}
```



生成的 SequenceFile 中存储的键 / 值对,键是从 100 到 1 降序排列的整数,表示为 IntWritable 对象，值是 Text 对象。在将每条记录追加到 SequenceFiie. Writer 实例末尾之前，调用 getLength() 方法来获取文件的当前位置。如果不按序读取文件，则使用这一信息作为记录的边界。

SequenceFile 文件的两种读取方法：

- 顺序读取
- 随机读取

顺序读取 SequenceFile 的步骤如下：

- 创建 Sequencerile.Reader 实例
- 反复调用 next() 方法迭代读取记录
- 读取的是哪条记录与你使用的序列化框架相关。如果使用的是 Writable 类型，那么通过键和值作为参数的 next() 方法可以将数据流中的下一条键 / 值对读入变量中：

```
public Boolean next (Writable key, Writable value)
```

- 如果键 / 值对成功读取，则返回 true，如果已读到文件末尾，则返回 false。

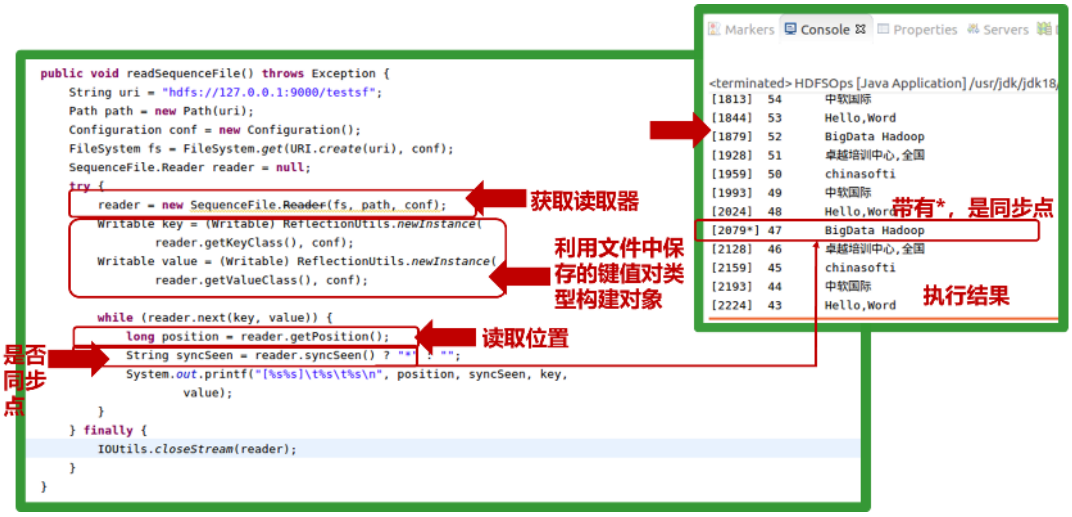
对于其他非 Writable 类型的序列化框架应该使用下面两个方法：

```
public Object next(Object key)throws IOException
Public Object getCurrentValue(Object value)throws IOException
```

这种情况下，需要确保 io.serializations 属性已经设置了你想使用的序列化框架，如果 next() 方法返回的是非 null 对象，则可以从数据流中读取键、值对，并且可以通过 getCurrentvalue() 方法读取该值。否则，如果 next()返回 null 值，则表示已经读到文件末尾。

顺序读取 SequenceFile 的参考实现如下：

```
    public void readSequenceFile() throws Exception {
        String uri = "hdfs://127.0.0.1:9000/tt/testsf2";
        Path path = new Path(uri);
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        SequenceFile.Reader reader = null;
        try {
            reader = new SequenceFile.Reader(fs, path, conf);
            Writable key = (Writable) ReflectionUtils.newInstance(
                    reader.getKeyClass(), conf);
            Writable value = (Writable) ReflectionUtils.newInstance(
                    reader.getValueClass(), conf);
            //reader.seek(1939);
            reader.sync(1959);
            while (reader.next(key, value)) {
                long position = reader.getPosition();
                String syncSeen = reader.syncSeen() ? "*" : "";
                System.out.printf("[%s%s]\t%s\t%s\n", position,
syncSeen, key,
                        value);
            }
        } finally {
            IOUtils.closeStream(reader);
        }
    }
```



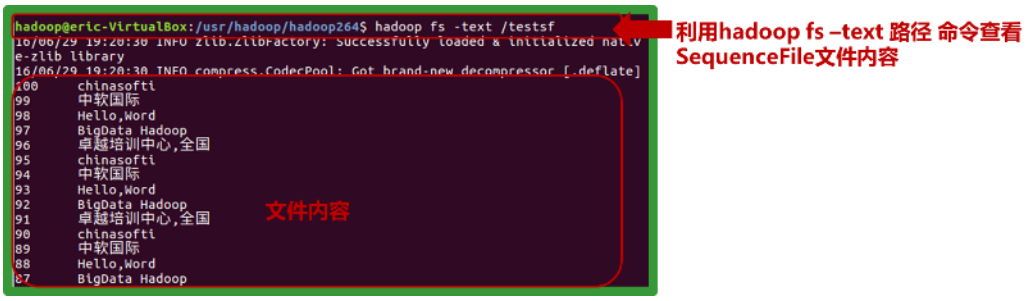
上面程序的另一个特性是能够显示顺序文件中同步点的位置信息。同步点是指数据读取的实例出错后能够再一次与记录边界同步的数据流中的一个位置，例如，在数据流中搜索到任意位置后。同步点是由 SequenceFile.Writer 记录的，后者在顺序文件写入过程中插入一个特殊项以便每隔几个记录便有一个同步标识。这样的特殊项非常小，因而只造成很小的存储开销，不到 1%。

同步点始终位于记录的边界处。

可以方便的通过 Hadoop 提供的控制台命令查看 SequenceFile 的文件内容：

```
hadoop fs  -text 路径
```

例如使用命令行查看刚才创建的 SequenceFile：

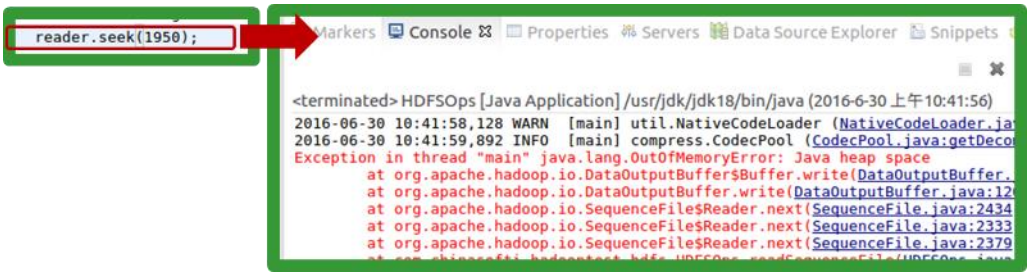


在 SequenceFile 中搜索给定位置有两种方法:调用 seek() 方法，该方法将读指针指向文件中指定的位置或通过同步点查找记录边界。

可以按如下方式搜查记录边界：

```
reader.seek(1939);
```

但如果给定位置不是记录边界，调用 next() 方法时就会出错：



第二种方法通过同步点查找记录边界。

SequenceFile.Reader 对象的 sync(Iong position) 方法可以将读取位置定位到 position 之后的下一个同步点。

如果 position 之后没有同步了，那么当前读取位置将指向文件末尾。这样，我们对数据流中的任意位置调用 sync() 方法（例如非记录边界）而且可以重新定位到下一个同步点并继续向后读取：

```
reader.sync(1959);
```

可以将加入同步点的顺序文件作为 MapReducc 的输入，因为该类顺序文件允许切分，由此该文件的不同部分可以由独立的 map 任务单独处理。由于 SequenceFile 本身存储的就是键值映射对，因此 Map 的输入不需要额外处理，可以直接使用键和值。如果使用 SequenceFile 作为 MapReduce 的输入，则需要在任务调度代码中设置输入格式为 SequenceFile：

```
job.setInputFormatClass(SequenceFileInputFormat.class);
```

读取 SequenceFile 作为输入的 Map 参考实现如下：

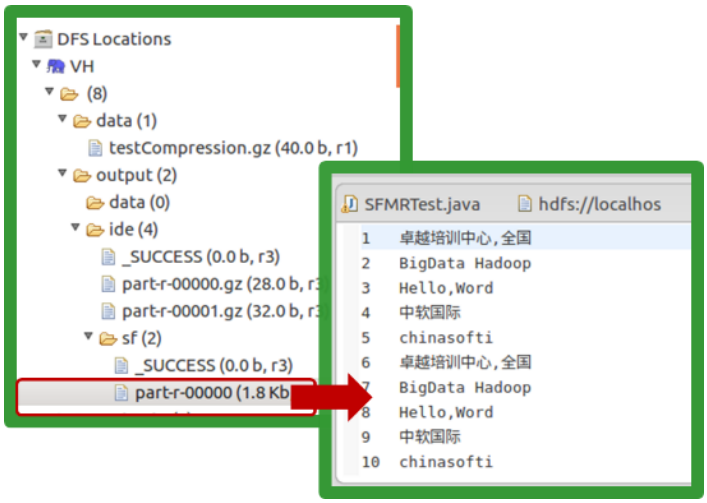
```
public static class SFMapper extends
        Mapper<IntWritable, Text, IntWritable, Text> {
```

```
        public void map(IntWritable key, Text value, Context
context)
                throws IOException, InterruptedException {
            System.out.println(key + "-----" + value);
            context.write(key, value);
        }

    }
```

读取 SequenceFile 作为输入的 Reduce 参考实现如下：

```
    public static class SFReducer extends
            Reducer<IntWritable, Text, IntWritable, Text> {

        public void reduce(IntWritable key, Iterable<Text> values,
                Context context) throws IOException,
InterruptedException {
            for (Text value : values) {
                context.write(key, value);
            }
        }

    }
```

程序运行结果如下：



Hadoop 中另一种容器格式为 MapFile。MapFile 是已经排过序的 sequenceFile，它有索引，所以可以按键查找。可以将 MapFile 视为 java.util.Map 的持久化形式（尽管它并没有实现该接口），它的大小可以超过保存在内存中一个 map 的大小。MapFile 的写操作类似于 SequenceFile 的写操作：

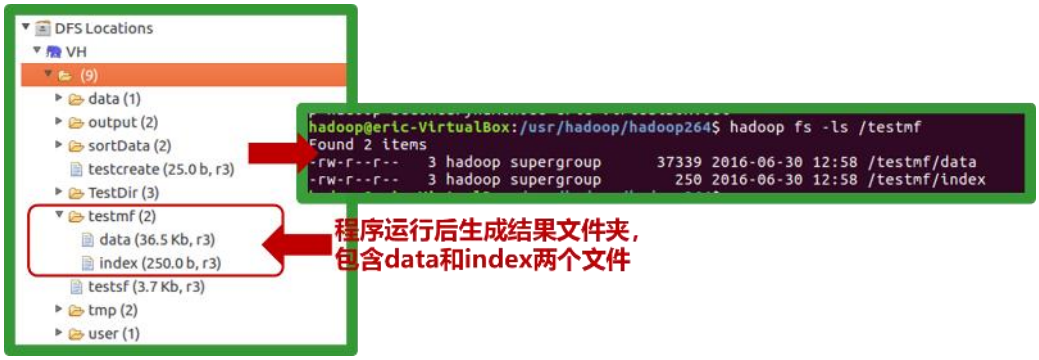
- 新建一个 MapFile.Writer 实例

- 调用 append() 方法顺序写入文件内容
- 如果不按顺序写入，就抛出一个 IOException 异常。
- 键必须是 WritableComparable 类型的实例，值必须是 Writable 类型的实例(这与 SequenceFile 正好相反，后者可以为其条目使用任意序列化框架)

可以利用 SequenceFile 文件的写入代码稍事修改，即可成功写入 MapFile 文件：

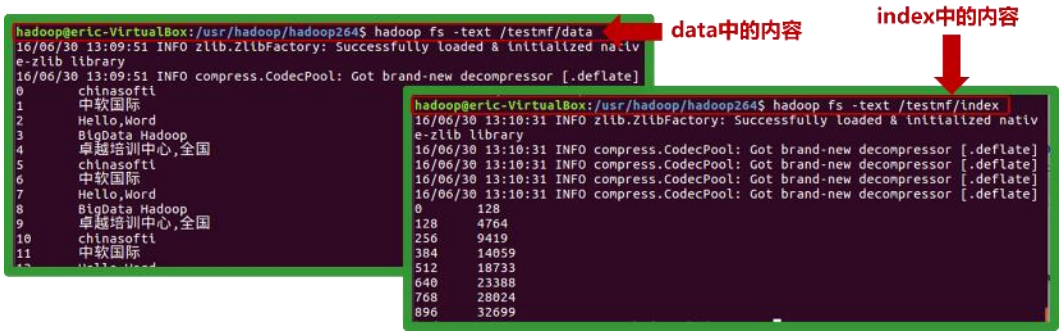
```
public class MapFileWriteDemo {

    private static final String[] DATA = { "chinasofti", "中软国际
", "Hello,Word",
            "BigData Hadoop", "卓越培训中心,全国" };

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://127.0.0.1:9000/testmf";
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        IntWritable key = new IntWritable();
        Text value = new Text();
        MapFile.Writer writer = null;
        try {
            writer = new MapFile.Writer(conf, fs, uri,
key.getClass(),
                    value.getClass());
            for (int i = 0; i < 1024; i++) {
                key.set(i);
                value.set(DATA[i % DATA.length]);
                writer.append(key, value);
            }
        } finally {
            IOUtils.closeStream(writer);
        }
    }

}
```

程序执行后输出的结果文件结构如下：



仔细观察这个 MapFile，会发现它实际上是一个包含 data 和 index 这两个文件的文件夹。这两个文件都是 SequenceFile。文件包含的内容如下：

- data 文件包含所有记录
- index 文件包含一部分键和 data 文件中键到其偏移量的映射



从 MapFile 的内容结构输出结果中可以看到：

默认情况下只有每隔 128 个键才有一个包含在 index 文件中。当然也可以调整，调用 MapFile.Writer 实例的 setIndexIntervai() 方法来设置 io.map.index.interval 属性即可。增加索引间隔大小可以有效减少 MapFile 存储索引所需要的内存。降低该间隔可以提高随机访问效率，代价是消耗了更多内存。因为索引只保留一部分键，所以 MapFile 无法枚举所有键甚至计算它自己有多少键。唯一的办法是读取整个文件。

在 MapFile 依次遍历文件中所有条目的过程类似于 SequenceFile 中的过程：

- 首先新建一个 MapFile.Reader 实例
- 调用 next() 方法，直到返回值为 false，表示没有条目返回，因为已经读到文件末尾
- 通过调用 get() 方法可以随机访问文件中的数据
- 返回值用于确定是否在 MapFile 中找到相应的条目，如果是 null，说明指定 key 没有相应的条目。如果找到相应的 key，则将该键对应的值读入 value 输入参数，通过方法调用返回。

第12节　Hadoop 支持的第三方序列化框架

尽管大多数 MapReduce 程序使用的都是 Writable 类型的键和值，但这并不是 MapReduce API 强制使用的。事实上，可以使用任何类型，只要能有一种机制对每个类型进行类型与二进制表示的来回转换。

为了支持这一机制，Hadoop 有一个针对可替换序列化框架（serialization framework）的 API。序列化框架用一个 Serialization 实现（包含在 org.apache.hadoop.io.serializer 包）来表示。例如，WritableSerialization 类是对 Writable 类型的 Serialization 实现。

Serialization 对象定义了：从类型到 Serializer 实例（将对象转换为字节流）和 Deserializer 实例（将字节流转换为对象）的映射方式。

将 io.serizalizations 属性设置为一个由逗号分隔的类名列表，即可注册 Serialization 实现。它的默认值包括：

- org.apache.hadoop.io・serializer.Writableserialization
- 序列化框架 Avro 指定序列化和自反序列化类

这意味着只有 Writable 对象和 Avro 对象才可以在外部序列化和反序列化。

Hadoop 包含一个名为 JavaSerialization 的类，该类使用 Java Object Serialization。尽管它方便了我们在 MapReduce 程序中使用标谁的 Java 类型，如 Integer 或 String，但不如 Writable 高效。

还有许多其他序列化框架从不同的角度来解决该问题：不通过代码来定义类型，而是使用“接口定义语言”(Interface Description Language,IDL）以不依赖干具体语言的方式进行声明。由此，系统能够为其他语言生成类型，这种形式能有效提高互操作能力。

Apache 和 Google 的 ProtocolBuffers 是两个比较流行的序列化框架，但它们常用作二进制数据的永久存储格式。MapReduce 格式对该类的支持有限。但在 Hadoop 内部，部分组件仍使用上述两个序列化框架来实现 RPC 和数据交换。

Apache Avro 是一个独立于编程语言的数据序列化系统。该项目由 Doug Cutting(Hadoop 之父）创建，旨在解决 Hadoop 中 Writable 类型的不足，缺乏语言的可移植性。拥有一个可被多种语言（当前是 c、c++＋、c＃、Java、PHP、Python 和 Ruby）处理的数据格式。

与绑定到单一语言的数据格式相比，Avro 更易于与公众共享数据集同时也更具生命力，该语言将使得数据具有更长的生命周期，即使原先用于读 / 写该数据的语言已经不再使用。

Avro 数据是用语言无关的模式定义的。但与其他系统不同的是，在 Avro 中，代码生成是可选的，这意味着你可以对遵循指定模式的数据进行读 / 写操作，即使在此之前代码从来没有见过这个特殊的数据模式。为此，Avro 假设数据模式总是存在的（在读 / 写数据时）形成的是非常精简的编码，因为编码后的数值不

需要用字段标识符来打标签。

Avro 模式通常用 Json 来写，数据通常采用二进制格式来编码。但也有其他选择。还有一种高级语言称为 AvroIDL，可以使用开发人员更熟悉的类 C 语言来写模式。

Avro 规范精确定义所有实现都必须支持的二进制格式。同时它还指定这些实现还需要支持的其他 Avro 特性。规范并没有给 API 制定规范，实现可以根据自己的需求操作 Avro 数据并给出相应的 API，因为每个 API 都与语言相关。事实上，只有一种二进制格式比较重要，这表明绑定一种新的编程语言来实现是比较容易的，可以避免语言和格式组合爆炸问题（否则将对互操作性造成一定的问题）。

Avro 有丰富的模式解析（schema resolution）能力。

Avro 是支持模式演化的，在精心定义的约束条件下，读数据所用的模式不必与写数据所用的模式相同。

例如，如果有一个新的、可选择的字段要加入一个记录中，那么需要在用于读取老数据的模式中声明它即可。新客户端和以前的客户端非常相似，均能读取按旧模式存储的数据，同时新的客户端可以使用新字段写入的新内容。相反，如果老客户端读取新客户端写入的数据，会忽略新加入的字段并按照先前的数据模式处理。

Avro 为一系列对象指定了一个对象容器格式：类似于 Hadoop 的顺序文件。

Avro 数据文件包含元数据项（模式数据存储在其中），使此文件可以自我声明。Avro 数据文件支持压缩，并且是可切分的，这对 MapReduce 的输入格式至关重要。Avro 本身是为 MapReduce 设计的，所以在不久的将来可能使 Avro 用于顶层的 MapReduce API（即比 Streaming 更丰富的 API，就像 Java API 或 C++管道一样）像使用其他编程语言一样使用。

Avro 定义了少量数据基本类型，它们可用于以写模式的方式来构建应用特定的数据结构。考虑到互操作性，其实现必须支持所有的 Avro 类型：

类型	描述	模式示例
Null	空值	“null”
boolean	布尔值	“boolean”
Int	32 位带符号整数	“int”
long	64 位带符号整数	“long”
float	单精度（32 位）IEEE754 浮点数	“float”
double	双精度（64 位）IEEE754 浮点数	“double”
bytes	8 位无符号字节序列	“bytes”
string	Unicode 字符序列	“string”

每个基本类型还可以使用更冗长的形式和使用 type 属性来指定，示例如下：

```
{“type”:” null”}
```

Avro 中的复杂数据类型如下：

类型	描述	模式示例

array	一个排过序的对象集合，特定数组中的所有对象必须模式相同	{
		“type”:”array”,
		“items”:”long”
		}

map	未排过序的键值对，键必须是字符串，值可以是任何类型，但一个特定 map 中的所有值必须模式相同	{
		“type”:”map”,
		“values”:”string”
		}

record	一个任意类型的命名字段集合	{
		“type”:”record”,
		“name”:”WeatherRecord”,
		“doc”:”A Weahter reading”,
		“fields”:{
		{“name”:”year”,”
		type”:”int”},
		{“name”:”
		temperature”,”type”:”int”}
		{“name”:”stationId”,”
		type”:”string”}
		}
		}

enum	一个命名的值集合	{
		“type”:”enum”,
		“name”:”Cutlery”,
		“doc”:”An eating utensil.”,
		“symbols”:[“KNIFE”,”
		FORK”,”SPOON”]
		}

fixed	一组固定数量的 8 位无符号字节	{
		“type”:”fixed”,
		“name”:”Md5Hash”,
		“size”:16
		}

union	模式的并集，并集可用 JSON 数组表示，其中每个元素为一个模式。并集表示的数据必须与其中一个模式相匹配	[“null”, “string”, {“type”:”map”,”values”:” string”}]

每个 Avro 语言 API 都包含该语言特定的 Avro 类型表示。例如，Avro 的

double 类型可以用 C、C++和 Java 语言的 double 类型，Python 的 float 类型以及 Ruby 的 Float 类型来表示。

对于一个特定的数据类型而言，一种语言可能有多种表示或映射。

所有的语言都支持动态映射，即使运行前并不知道具体模式，也可以使用动态映射。对此，Java 称为“通用”(generic）映射。

Java 和 C++实现可以自动生成代码来表示符合某种 Avro 模式的数据。

代码生成（code generation 在 Java 中称为“特殊映射”）能优化数据处理，如果读 / 写数据之前就有一个模式备份。那么，为用户代码生成的类和为通用代码生成的类相比，前者提供的 API 更贴近问题域。

Java 拥有第三类映射，即自反映射（reflectnlapping，将 Avro 类型映射到已有的 Java 类型）。它的速度比通用映射和特殊映射都慢，所以不推荐在新应用中使用。

如下所示，特殊映射和通用映射相同，除非有特别说明（同样的，自反映射与特殊映射也相同，除非特别说明）。特殊映射与通用映射仅在 record,enum 和 fixed 三个类型方面有区别，其他所有类型均有自动生成的类（类名由 name 属性和可选的 namesapce 属性决定）。

Avro 类型	Java 通用映射	Java 特殊映射	Java 自反映射
null	null 类型		
boolean	boolean		
int	int		int 或 short
long	long		
float			
double	double		
string	org.apache.avro.util.Utf8		Java.lang.String
array	org.apache.avro.generic.GenericArray		数组或 Jiava.util.Collection
map	Java.util.map		
record	org.apache.avro.generic.GenericRecord	生成实现 org.apache.avro.specific.SpecificRecord 类的实现	具有零参数构造方法的任意用户类，继承了所有不传递的示例字段
enum	Java.lang.String	生成 Java enum 类型	任意 Java enum 类型
fixed	org.apache.avro.generic.Genericfixed	生成实现 org.apache.avro.specific.SpecificFixedb	org.apache.avro.specific.SpecificFixed
bytes	Java.nio.ByteBuffer		字节数组

union　　Java.lang.Object

Avro string 既可以通过 Java String 表示，也可以通过 Avro Utf8 Java 类型表示。相较之下，使用 Utf8 表示比较高效。Avro Utf8 类型是易变的，单个 UTF8 实例可以重用，并对一系列值进行读 / 写操作。

Java String 在新建对象时进行 UTF-8 解码，但 Avro 执行 UTF-8 解码更晚一些，某些情况下这样做可以提高系统性能。Utf8 实现了 Java 的 java.lang.Charsequence 接口，该接口可以与 Java 类库实现互操作。在其他情况下可能需要通过调用 toString() 方法将 utf8 实例转化成 String 对象。

Avro 为序列化和反序列化提供了 API，如果想把 Avro 集成到现有系统（比如已经定义帧格式的消息系统），这些 API 就很有用。其他情况，应该考虑使用 Avro 数据 Schema 文件格式。该文件以传统的 JSON 文档的格式存在，例如，要保存 left 和 right 两个字段的数据进行序列化，那么 Schema 的格式如下：

```
{
"namespace":"com.chinasofti.hadooptest.avro",
"type":"record",
"name":"StringPair",
"doc":"A pair of Strings.",
"fields":[
    {"name":"left","type":"string"},
    {"name":"right","type":"string"}
]

}
```

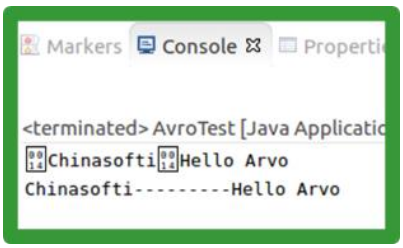
特别需要注意的是，使用 Java 特殊映射，Avro 模式文件一定要添加 namespace 说明，值为生成代码所处的 Java 包，否则在序列化时会抛出 NullPointerException。

在内存中利用该 Schema 文件进行数据序列化和反序列化的参考代码如下：

```
public class AvroTest {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        //加载模式文件
        Schema.Parser parser = new Schema.Parser();
        Schema schema = parser.parse(new File("StringPair.avsc"));
        //映射为Record类型，并写入两个字符串
        GenericRecord datum = new GenericData.Record(schema);
        datum.put("left", new Utf8("Chinasofti"));
        datum.put("right", new Utf8("Hello Arvo"));
        //序列化
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        DatumWriter<GenericRecord> writer = new
```

```
GenericDatumWriter<GenericRecord>(
                schema);
        Encoder encoder = EncoderFactory.get().binaryEncoder(out,
null);
        writer.write(datum, encoder);
        encoder.flush();
        out.close();
        System.out.println(new String(out.toByteArray()));
        //反序列化
        DatumReader<GenericRecord> reader = new
GenericDatumReader<GenericRecord>(
                schema);
        BinaryDecoder decoder =
DecoderFactory.get().createBinaryDecoder(
                out.toByteArray(), null);
        GenericRecord record = reader.read(null, decoder);
        System.out.println(record.get("left") + "---------"
                + record.get("right"));
    }

}
```

程序执行结果如下：



上述示例代码中有两个重要的对象：

DatumWrlter，DatumWriter 对象将数据对象翻译成 Encoder 对象可以理解的类型，然后由后者写到输出流。

Encoder，使用 GenericDatumWriter 对象，它将 GenericRecord 字段的值传递给 Encoder 对象。将 null 传给 Encode 工厂，因为这里没有重用先前构建的 Encoder。

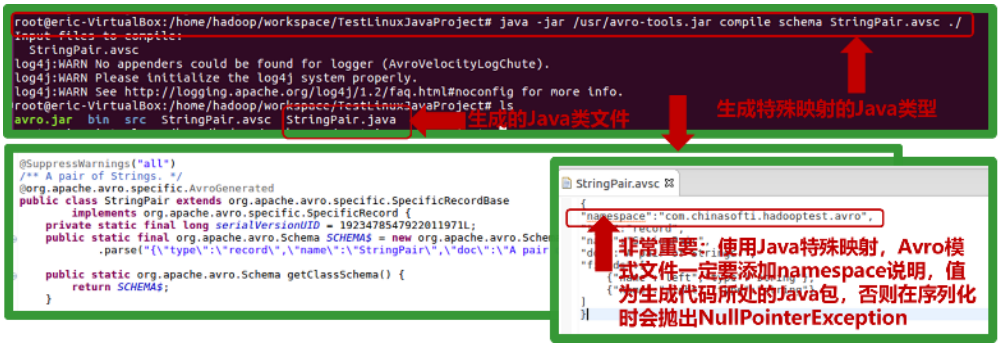
在本例中，只有一个对象被写到输出流，但如果需要写若干个对象，可以多次调用 write() 方法，然后再关闭输入流。需要向 GenericDatumWriter 对象传递模式，因为它要根据模式来确定将数据对象中的哪些数值写到输出流。

如果使用特殊映射，即将需要序列化的数据抽象为一个 Java 类型，这样在使用的时候更符合传统的 Java 面向对象编码习惯，则需要利用 Avro Java 工具生成特殊映射代码。

首先在 Avro 官方网站下载 avro-tools-*.jar，然后在.avsc 模式文件所在文件夹中执行 AvroTools 的 compile 方法自动生成 Java 特殊映射类型，生成代码的命令格式如下：

```
java -jar AvroJava 工具 jar compile schema Schema 文件 路径
```

例如：



使用生成代码完成序列化和反序列化的代码和之前类似，只是在创建读写器的时候将类型映射改变为特殊映射：

```
public class AvroTestWithJavaType {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        //直接使用生成的Java类型
        StringPair datum = new StringPair();
        // GenericRecord datum = new GenericData.Record(schema);
        datum.setLeft("Chinasofti");
        datum.setRight("HelloWorld");

        System.out.println(datum.getLeft() + "\t" +
datum.getRight());
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        //使用特殊映射写入工具
        DatumWriter<StringPair> writer = new
SpecificDatumWriter<StringPair>(StringPair.class);
        Encoder encoder = EncoderFactory.get().binaryEncoder(out,
null);
        writer.write(datum, encoder);
        encoder.flush();
        out.close();
        System.out.println(new String(out.toByteArray()));
        //使用特殊映射读取工具
        DatumReader<StringPair> reader = new
SpecificDatumReader<StringPair>(StringPair.class);
```

```
        BinaryDecoder decoder =
DecoderFactory.get().createBinaryDecoder(
                out.toByteArray(), null);
        StringPair record = reader.read(null, decoder);
        System.out.println(record.getLeft() + "---------" +
record.getRight());
    }
}
```

该代码执行结果和使用通用类型映射的代码执行结果保持一致。

在实际的应用系统中，单纯的在内存中执行序列化和反序列化明显不能满足需要，为了能够在不同的系统中正常的交换数据，在项目的开发过程中往往需要将数据序列化后的结果保存为一个独立的数据文件用于网络传输交换，Avro 同样支持这种数据交换方式。

Avro 数据文件作为 Avro 的对象容器，主要用于存储 Avro 对象序列。这与 Hadoop 顺序文件的设计非常相似，主要区别在于 Avro 数据文件主要是面向跨语言使用而设计的，所以，可以用 Java 写入文件，而用 C 语言来读取文件。

Avro 数据文件的头部分包含元数据，包括一个 Avro 模式和一个 syncmarker（同步标识），紧接着是一系列包含序列化 Avro 对象的数据块（压缩可选）。数据块由 syncmarker 来分隔，它对该文件而言，是唯一的（特定文件的标识信息存储在文件头部），并且允许在文件中搜索到任意位置之后通过块边界快速地重新进行同步。

Avro 数据文件是可切分的，适合 MapReduce 快速处理

Avro 的对象写到数据文件与写到数据流类似，在写入数据时，使用 DatumWriter 作为写入工具。不用 Encoder，而是用 DatumWriter 来创建一个 DataFileWriter 实例。然后便可以新建一个数据文件（该文件一般有. avro 扩展名）并向它附加新写入的对象。

从数据文件中读取对象与前面例子中在内存数据流中读取数据类似，只有一个重要的区别：不需要指定模式，因为可以从文件元数据中读取它。事实上，还可以对 DataFileReader 实例调用 getschema() 方法来获取该模式，并验证该模式是否和原始写入对象的模式相同

基于 Avro 数据文件的数据写入与读取参考实现如下：

```
public class WriteAvroDatafile {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        Schema.Parser parser = new Schema.Parser();
        Schema schema = parser.parse(new File("StringPair.avsc"));
        GenericRecord datum = new GenericData.Record(schema);
        datum.put("left", new Utf8("Chinasofti"));
        datum.put("right", new Utf8("Hello Arvo"));
```

```
        DatumWriter<GenericRecord> writer = new
GenericDatumWriter<GenericRecord>(
                schema);
        File dataFile = new File("test.avro");
        DataFileWriter<GenericRecord> dataFilewriter = new
DataFileWriter<>(
                writer);

        dataFilewriter.create(schema, dataFile);
        dataFilewriter.append(datum);
        dataFilewriter.close();

        DatumReader<GenericRecord> reader = new
GenericDatumReader<GenericRecord>(
                schema);
        DataFileReader<GenericRecord> dataFileReader = new
DataFileReader<>(
                dataFile, reader);
        GenericRecord record = dataFileReader.next();
        System.out.println(dataFileReader.getSchema());
        System.out.println(record.get("left") + "---------"
                + record.get("right"));
    }
}
```

程序执行结果如下：

之前提到了，Avro 文件本身支持切分，因此它使用用于作为 MapReduce 任务的数据输入，利用 Avro 文件作为 MapReduce 的输入，需要首先在任务调度代码中设定 MapReduce 任务的输入格式：

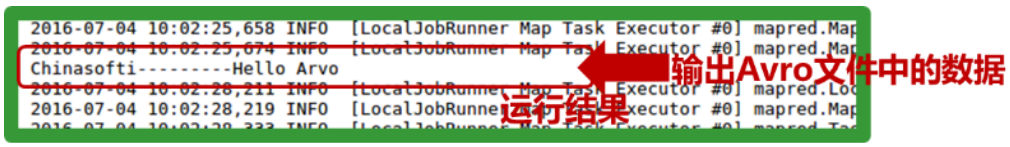
```
job.setInputFormatClass(AvroKeyInputFormat.class);
```

对应的 Map 和 Reduce 函数的参考实现如下：

```
    public static class AvroMapper extends
```

```
            Mapper<AvroKey<GenericRecord>, NullWritable, Text,
IntWritable> {

        public void map(AvroKey<GenericRecord> key, NullWritable
value,
                Context context) throws IOException,
InterruptedException {
            //将之前生成的Avro数据文件作为输入，因此输出Avro数据记录
中的left和right内容
            System.out.println(key.datum().get("left") + "---------"
                    + key.datum().get("right"));
        }
    }

    public static class AvroReducer extends
            Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws IOException,
InterruptedException {
            System.out.println("Reduce!");
        }

    }
```

程序执行的结果如下：



完整代码实现参考本教材的随书代码。

第5章 数据流脚本语言 Pig

第1节 Pig 入门

MapReduce 使程序员能够自己定义一个 map 函数和一个紧跟其后的 reduce 函数。但是，必须使数据处理过程与这一连续的 map 和 reduce 模式相匹配。很多时候，数据处理需要多个 MapReduce 过程才能实现。而使得数据处理过程与该模式匹配可能很困难。

Pig 为大型数据集的处理提供了更高层次的抽象。

有了 Pig，就能使用更为丰富的数据结构。这些数据结构往往都是多值和嵌套的。Pig 还提供了一套更强大的数据变换操作，包括在 MapReduce 中被忽视的连接（Join）操作。

从概念上来说，Pig 包含了两个组成部分：

- 用于描述数据流的语言，称为 Pig Latin
- 用于运行 Pig Latin 程序的执行环境。

当前的 Pig 版本存在两个脚本运行环境，用于不同的开发、测试和生产部署场景：

- 单 JVM 中的本地执行环境
- Hadoop 集群上的分布式执行环境。

Pig Latin 程序由一系列的“操作”(operation)或“变换”(transformation)组成。每个操作或变换对输入进行数据处理，并产生输出结果。从整体上看，这些操作描述了一个数据流。Pig 执行环境把数据流翻译为可执行的内部表示并运行它。在 Pig 内部，这些变换操作被转换成一系列 MapReduce 作业。

Pig 是一种探索大规模数据集的脚本语言。

MapReduce 的一个缺点是开发周期太长：首先需要编写 mapper 和 reducer 函数，然后对代码进行编译和打包并提交作业，最后获取结果。

即便使用 streaming 能在这一过程中去除代码的编译和打包步骤，仍不能改善这一情况。

Pig 的诱人之处在于仅用控制台上的五六行 Pig Latin 代码就能够处理 TB 级的数据。作为程序员，多数情况下并不需要知道这些转换是如何进行的。这样一来，便可以将精力集中在数据上，而非执行细节。简单的说，Pig 将程序员从繁重的 MapReduce 代码开发任务中解救出来，通过简短的脚本描述，就能完成常见的数据处理任务。

Pig 提供了多个命令来检查和处理程序中已有的数据结构。因此，它能够很好地支持程序员撰写查询。

Pig 的一个更有用的特性是它支持在输入数据的一个有代表性的子集上试运行。这样一来，用户可以在处理整个数据集前检查程序执行时是否会有错误。

Pig 被设计为可扩展的。处理路径中的几乎每个部分，包括载入、存储、过滤、分组、连接，都可以定制。这些操作都可以使用用户定义函数(user-defined function,UDF)进行修改。这些用户定义函数作用于 Pig 的嵌套数据模型。因此，它们可以在底层与 Pig 的操作集成。UDF 的另一个好处是，相较于为了写 MapReduce 程序而开发的库，它们更易于重用。

当然，Pig 并不适合所有的数据处理任务。

和 MapReduce 一样，它是为数据批处理而设计的。如果想执行的查询只涉及一个大型数据集中的一小部分数据，Pig 的表现并不会很好。这是因为它要扫描整个数据集或其中的很大一部分。

在有些情况下，Pig 的表现不如 MapReduce 程序。但随着新版本的发布，Pig 的开发团队使用了复杂、精巧的算法来实现 Pig 的关系型操作，二者的差距不断缩小。公平地说，除非你愿意花大量时间来优化 Java MapReduce 程序，否则使用 Pig Latin 来写查询的确能够帮你节约时间。

第2节 安装 Pig

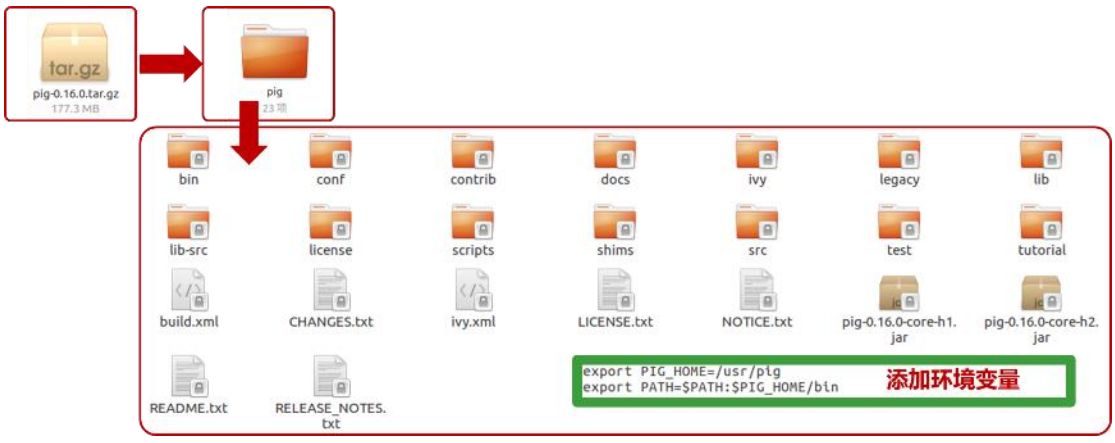
Pig 是作为一个客户端应用程序运行的。即使准备在 Hadoop 集群上运行 Pig，也无需在集群上额外安装什么东西。Pig 从工作站上发出作业，并在工作站上和 HDFS（或其他 Hadoop 文件系统）进行交互。

Pig 的安装非常简单，只需要在 Pig 的官网上下载其发行包，并解压到期望安装的目标文件夹中，最后设置必要的环境变量即可。

Pig 需要的基本环境变量如下：

```
export PIG_HOME=/usr/pig
export PATH=$PATH:$PIG_HOME/bin
```

安装 Pig 的基本步骤如下图：



第3节 Pig 的基本使用

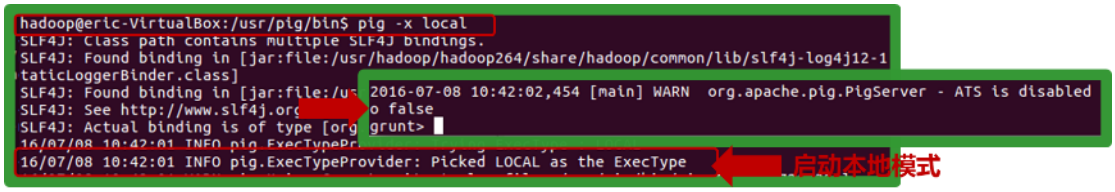
上之前的章节中已经提到，Pig 具有两种执行类型：本地模式和 MapReduce 模式。

在本地模式下，Pig 运行在单个 JVM 中，访问本地文件系统。该模式只适用于处理小规模数据集或试用 Pig 时。执行类型可用-x 或-exectype 选项进行设置。

如果要使用本地模式运行，应把该选项设置为 local:

```
pig  -x local
```

这样就能够启动 Grunt。Grunt 是与 Pig 进行交互的外壳程序（shell）：



从控制台的日志输入可以看出，当前启动的交互界面运行在本地模式之上。

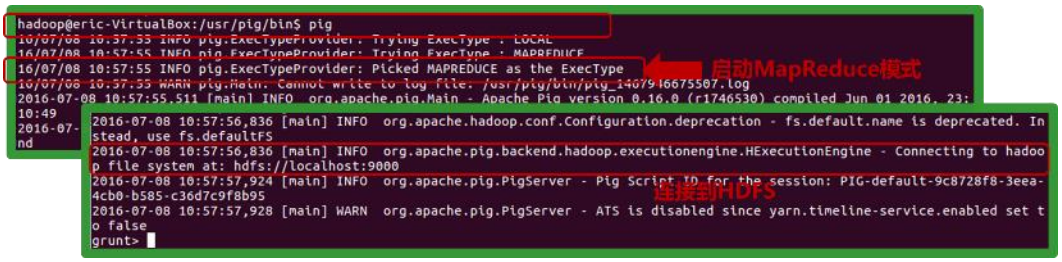
在 MapReduce 模式下，Pig 将查询翻译为 MapReduce 作业，然后在 Hadoop 集群上执行。集群可以是伪分布的，也可以是全分布的。如果要用 Pig 处理大规模数据集，应该使用（全分布集群上的）MapReduce 模式。

Pig 根据 HADOOP_HOME 环境变量来寻找并运行对应的 Hadoop 客户端。但是，如果该环境变量没有被设置过，那么 Pig 会运行捆绑在 Pig 中的 Hadoop 库。注意，捆绑的库的版本可能和集群上的 Hadoop 版本不一样。所以，最好明确指定 HADOOP_HOME。

然后，需要将 Pig 指向集群的 namenode 和 jobtracker。如果安装在 HADOOP_HOME 中的 Hadoop 已经进行了设置，那么不需要进行额外的配置。否则，可以把 HADOOP_CONF__DIR 设置为一个目录，其中包含 fs.default.name 和 mapred.job.tracker 定义的 Hadoop 站点文件。另一种办法是，在 Pig 的 conf 目录下的 pig.properlies 文件中设置这两个属性。

一旦设置好 Pig 到 Hadoop 集群的连接，就可以设置-x 选项为 mapreduce 或忽略该选项来运行 Pig。

可以忽略该选项的原因是 MapReduce 模式是 Pig 的默认执行模式：



有 3 种方法可以运行 Pig 程序：

脚本

Pig 可以运行包含 Pig 命令的脚本文件。例如，pig script.pig 运行在本地文件 script.pig 中的命令。或者，对于很短的脚本，也可以使用-e 选项直接在命令行中以字符串形式输入脚本。

Grunt

Grunt 是运行 Pig 命令的交互式外壳环境（shell），如果没有指明 Pig 要运行的文件，而且也没有使用-e 选项，Pig 就会启动 Grunt。在 Grunt 环境中，还可以通过 run 和 exec 命令运行 Pig 脚本。

嵌入式方式

还可以在 Java 中通过 PigServer 类运行 Pig 程序。这就像能够在 Java 中使用 JDBC 运行 SQL 程序。如果要以编程的方式访问 Grunt，则需要使用 PigRunner。

Grunt 包含的行编辑功能和 GNU Readline（在 bash 外壳环境等命令行应用中使用）类似。例如，组合键 Ctrl 十 E 将光标移到行末。Grunt 也记录过去执行过的命令。可以使用 Ctrl-P 和 Ctrl-N 或上下光标键，回显命令历史缓存中

Grunt 的另一个有用特性是自动补全机制。它能够在你按 Tab 键时试图自动补全 Pig Latin 关键词和函数。

可以创建一个名为 autocomplete 的文件并把它放在 Pig 的类路径（如 Pig 的 install 目录的 conf 目录中）中或者启动 Grunt 的目录中，以此来定制自动补全的单词。这个文件中每个单词占一行，且单词中不能出现空白字符。自动补全的匹配是大小写敏感的。在这个文件中列出常用的文件路径特别有用（因为 Pig 并不提供文件名自动补全）。在该文件中列出你创建的用户自定义函数也能带来很多便利。可以使用 help 命令列出命令列表。结束一个 Grunt 会话时，可以使用 quit 命令退出。

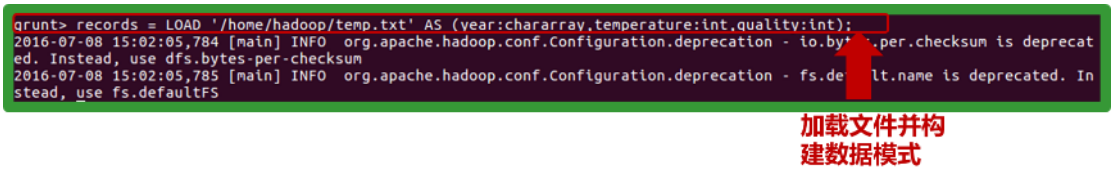
PigPen 是一个提供了 Pig 程序开发环境的 Eclipse 插件。

它包含 Pig 脚本编辑器、示例生成器（等同于 ILLUSTRATE 命令）以及用于在 Hadoop 集群上运行脚本的按钮。它还提供一个操作图窗口，能够以图形方式显示脚本，将数据流可视化。

接下来通过与在 MapReduce 入门数据分析一节中实现过的汇总统计每一年最高气温相似的案例来展示 Pig 是如何简化 MapReduce 任务开发工作的。

首先利用本地模式加载本地文件，内容如下：

加载文件并创建模式：



为了简单起见，程序假设输入是由制表符分割的文本，每行只包含年度、气温和质量三个字段。（事实上，Pig 的输入格式处理能力比这个灵活得多）。

这行代码描述要处理的输入数据。year:chararray 描述字段的名称和类型。chararray 和 Java 字符串类似，int 和 Java 的 int 类似。

LOAD 操作接受一个 URI 参数作为输入。在这个示例中，只使用了一个本地文件。在 LOAD 中，也可以引用 HDFS URI。

可选的 AS 子句设定了字段的名称，以便在随后的语句中更方便地引用它们。

LOAD 操作的结果和 Pig Latin 其他所有操作的结果一样，都是一个关系（relation），即一个元组集合。一个元组类似于数据库表中的一行数据，包含按照特定顺序排列的多个字段。

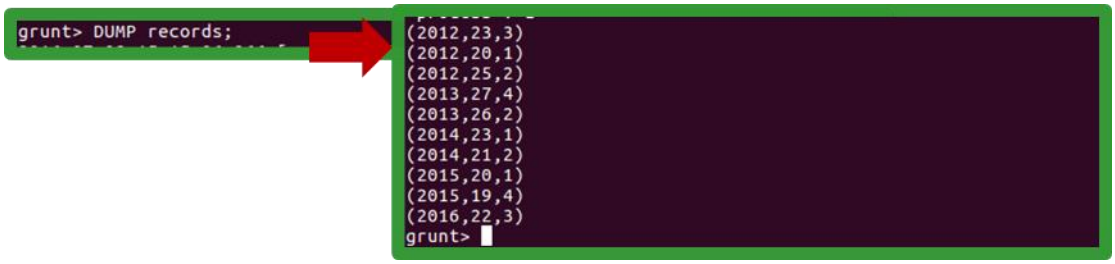
在这个示例中，LOAD 函数根据输入文件，生成一个（年份，气温，质量）元组的集合。关系写成每个元组一行，每行由括号括起，每项字段由逗号分隔：

```
（2012,23,3）
（2012,20,1）
（2012,25,2）
（2013,27,4）
（2013,26,2）
  …
```

关系被赋予名称或别名（alias）以便于引用。示例关系的别名是 records。

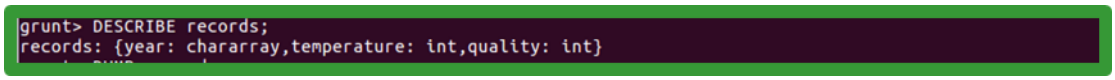
可以使用 DUMP 操作来查看某个别名所对应关系的内容：

```
DUMP 关系别名
```



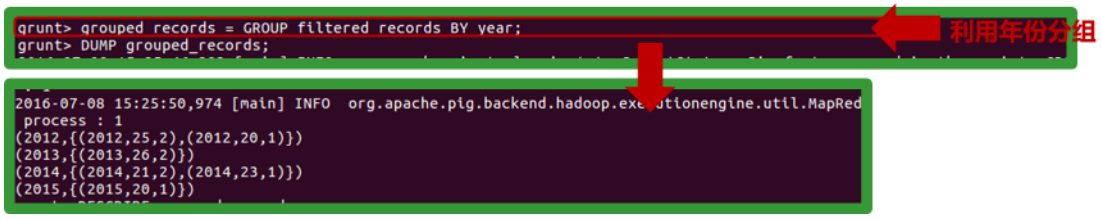
另一个重要的 DESCRIBE 操作通过关系的别名来查看该关系的结构，即关系的模式：

```
DESCRIBE 关系别名
```



可以通过类似与 SQL 的条件表达式对数据进行过滤，数据过滤使用 FILTER 变换，该操作将会生成一个满足过滤条件的新关系集合：

通过 GROUP 函数通过年度值对数据进行分组：



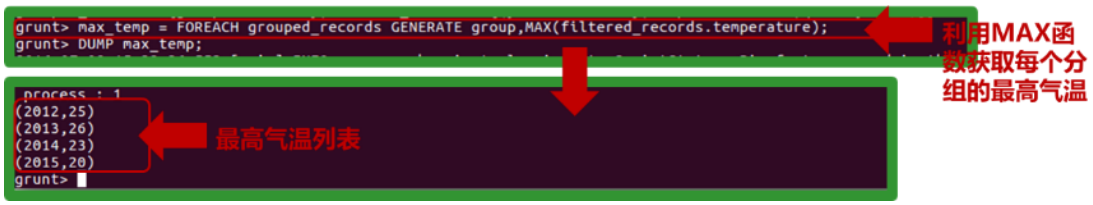
完成分组后的每个元组对应于输入数据中的一个年度。

每个元组的第一个字段是用于进行分组的字段（即年度）。第二个字段是该年度的元组的包（bag）。包（bag）是元组的无序集。在 Pig Latin 里，包用大括号表示。可以查看分组后的数据模式：

```
grunt> DESCRIBE grouped_records;
grouped_records: {group: chararray,filtered_records: {(year: chararray,temperature: int,quality: int)}}
grunt>
```

从输出结果可以看到，Pig 给分组字段起了别名 group。第二个字段和被分组的 filtered_records 关系的结构相同。

分组后可以获取每组中的最高气温：



FOREACH 对每一行数据进行处理，并生成一组导出的行。导出的行的字段由 GENERAT 子句定义。在这个示例中，第一个字段是 group，也就是年度。第二个字段稍微复杂一点。filtered_records.temperature 引用了 grouped_records 关系中的 filtered_records 包中的 temperature 字段。MAX 是计算包中字段的最大值的内置函数。在这个例子中，它计算了 filtered_records 包中的最高温度。

在示例已经使用了一个仅包含少数行的抽样数据集，以简化数据流跟踪和调试。

创建一个精简的数据集是一门艺术。理想情况下，这个数据集的内容应该足够丰富，能够覆盖查询中可能碰到的各种情况（满足完备性[completeness]条件），同时，这个数据集应该足够小，能够被程序员直观理解（满足简明性[conciseness]条件）。

通常情况下，使用随机取样并不能满足要求，因为连接和过滤这两个操作往往会去除掉所有的随机取样的数据，而导致产生一个空的结果集。这样是无法描述典型的数据流的。

Pig 通过 ILLUSTRATE 操作，提供了生成相对完备和简明的数据集的工具：

ILLUSTRATE 关系别名

```
grunt> ILLUSTRATE max_temp;
```

```
-------------------------------------------------------------------------
| records     | year:chararray     | temperature:int     | quality:int     |
-------------------------------------------------------------------------
|             | 2012               | 25                  | 2               |
|             | 2012               | 20                  | 1               |
|             | 2013               | 27                  | 4               |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| filtered_records     | year:chararray     | temperature:int     | quality:int     |
-------------------------------------------------------------------------
|                      | 2012               | 25                  | 2               |
|                      | 2012               | 20                  | 1               |
-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
| grouped_records     | group:chararray     | filtered_records:bag{:tuple(year:chararray,temperature:int,quality:int)}
-------------------------------------------------------------------------------------------------------------
|                     | 2012                | {}
|                     | 2012                | {}
-------------------------------------------------------------------------------------------------------------
-----------------------------------------
| max_temp     | group:chararray     | :int     |
-----------------------------------------
|              | 2012                | 25       |
-----------------------------------------
```

从上面的结果可以看出，Pig 既使用了部分的原始数据（这对于保持生成数据集的真实性很重要），也创建了一些新的数据。Pig 注意到查询中的过滤条件值，会创建一个包含该值的元组来测试 FILTER 语句。

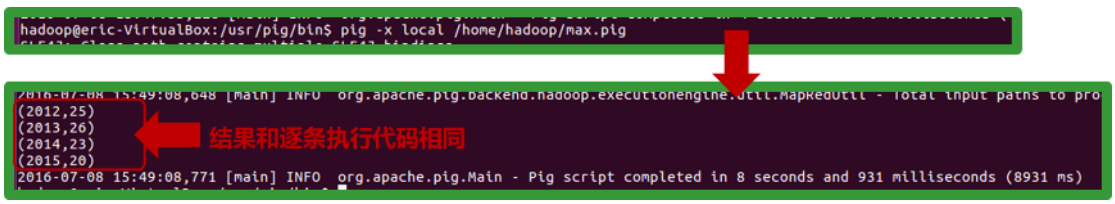
ILLUSTRATE 的输出本身易于理解，而且也能帮助开发人员理解查询的执行过程。

可以直接将示例中的数据分析过程代码编写为一个脚本，然后一次性利用脚本运行模式获取数据分析结果（如果不希望安装其他的开发工具，可以直接使用 Linux 中的 gedit，它支持 Pig 脚本的语法高亮显示）：

```
records = LOAD '/home/hadoop/temp.txt' AS (year:chararray,temperature:int,quality:int);

filtered_records = FILTER records BY temperature != 1 AND (quality==1 OR quality==2);

grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group,MAX(filtered_records.temperature);

DUMP max_temp;
```

接下来执行该脚本：

```
pig -x 执行模式 脚本文件路径
```



看上去，Pig Latin 和 SQL 很相似，GROUP BY 和 DESCRIBE 这样的操作更加强了这种感觉。

但是，两种语言之间，以及 Pig 和关系数据库管理系统（RDBMS）之间，有几个方面是不同的。

它们之间最显著的不同是：Pig Latin 是一种数据流编程语言，而 SQL 是种声明式编程语言。换句话说，一个 Pig Latin 程序是一组针对输入关系的一步步操作。其中每一步都是对数据的简单变换。相反，SQL 语句是一个约束的集合。这些约束结合在一起，定义了输出。

从很多方面看，用 Pig Latin 编程更像在 RDBMS 中查询规划器（query planner）这一层对数据进行操作。查询规划器决定了如何将声明式语句转化为一系列系统化执行的步骤。

RDBMS 把数据存储在严格定义了模式的表内。Pig 对它所处理的数据要求则

宽松得多：你可以在运行时定义模式，而且这是可选的。本质上，Pig 可以在任何来源的元组上进行操作（当然，数据源必须支持并行的读操作，例如存放在多个文件中）。它使用 UDF 从原始格式中读取元组。

最常用的输入格式是用制表符分隔的字段组成的文本文件。Pig 为这种输入格式提供了内置加载函数。

和传统的数据库不同,Pig 并不提供专门的数据导入过程将数据加载到 RDBMS。从文件系统（通常是 HDFS）中加载数据是处理的第一个步骤。

Pig 对复杂、嵌套数据结构的支持也使其不同于只能处理平面数据类型的 SQL。

Pig 的语言能够和 UDF 以及流式操作紧密集成。Pig Latin 的这一能力及其嵌套数据结构，使它比大多数 SQL 的变种具有更强的定制能力。

RDBMS 具有一些支持在线和低延迟查询的特性，如事务和索引。而这些特性是 Pig 所缺乏的。如前所述，Pig 并不支持随机读和几十毫秒级别的查询。它也不支持针对一小部分数据的随机写。和 MapReduce 一样，所有的写都是批量的、流式的写操作。

后续章节将会介绍的 Hive 介于 Pig 和传统的 RDBMs 之间和 Pig 一样，Hive 也被设计为用 HDFS 作为存储。但是它们之间有着显著的区别：

Hive 的查询语言 HiveQL 是基于 SQL 的。任何熟悉 SQL 的人都可以轻松使用 HiveQL 写查询。

和 RDBMS 相同，Hive 要求所有数据必须存储在表中，而表必须有模式，且模式由 Hive 进行管理。但是，Hive 允许为预先存在于 HDFS 的数据关联一个模式。所以，数据的加载步骤是可选的。

和 Pig 样，Hive 也不支持低延迟查询。

第4节 Pig Latin 入门

Pig Latin 是用于描述数据流的语言。一个 Pig Latin 程序由一组语句构成。一个语句可以理解为一个操作，或一个命令。例如，GROUP 操作是这样一种语句：

```
Grouped_records = GROUP records BY year;
```

另一个语句的例子是列出 Hadoop 文件系统中文件的命令：

```
ls /
```

Pig Latin 并没有正式的语言定义。但通过 pig 网站 http://pig.apache.org 可以找到 Pig Latin 语言的完整指南。

如前面的 GROUP 语句所示，一条语句通常用分号结束。实际上，那是一条必须用分号表示结束的语句。如果省略了分号，它会产生一个语法错误。

而另一方面，ls 命令可以不用分号结束。

一般的规则是：在 Grunt 中交互使用的语句或命令不需要表示结束的分号。这包括交互式的 Hadoop 命令以及用于诊断的操作，例如 DFSCRIBE。

加上表示结束的分号总是不会错。因此，如果不确定是否需要分号，把它加上是最简单的解决办法。

必须用分号表示结束的语句可以被分割成多行以便于阅读：

```
records = LOAD ‘/home/Hadoop/temp.txt’
   AS (year:chararray,temperature:int,quality:int);
```

PigLatin 有两种注释方法：

双减号表示单行注释，Pig Latin 解释器会忽略从第一个减号开始到行尾的所有内容：

```
-- My program
DUMP A; -- What’s in A?
```

C 语言风格的注释更灵活。这是因为它使用/＊和＊/符号表示注释块的开始和结束。这样，注释既可以跨多行，也可以内嵌在某一行内：

```
/*
* Descriptlon of my program spanning
* multiple lines.
*/
```

和大部分的编程语言一样，Pig Latin 有一个关键词列表，其中的单词在 Pig Latin 中有特殊含义，不能用作标识符。这些单词包括：

- 操作（LOAD,ILLUSTRATE 等）
- 命令（cat,ls 等）
- 表达式（matches,FLATTEN 等）
- 函数（DIFF,MAX 等）

Pig Latin 采用混合的大小写敏感规则。操作和命令是大小写无关的（这样能使得交互式操作更“宽容”），而别名和函数名是大小写敏感的。

在 Pig Latin 程序执行时，每个命令按次序进行解析。

如果遇到语法错误或其他（语义）错误，例如未定义的别名，解释器会终止运行，并显示一条错误消息。

解释器会给每个关系操作建立一个“逻辑计划”(Logical Plan)。逻辑计划构成了 Pig Latin 程序的核心。解释器把为一个语句创建的逻辑计划加到到目前为止已经解析完的程序的逻辑计划上，然后继续处理下一条语句。

特别需要注意的是，在构造整个程序的逻辑计划时，Pig 并不处理数据。仍然以前面的 Pig Latin 程序为例：

```
records = LOAD '/home/hadoop/temp.txt' AS (year:chararray,temperature:int,quality:int);

filtered_records = FILTER records BY temperature != 1 AND (quality==1 OR quality==2);

grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group,MAX(filtered_records.temperature);

DUMP max_temp;
```

Pig Latin 解释器看到第一行 LOAD 语句时，首先确认它在语法和语义上是正确的，然后再把这个操作加入逻辑计划。但是，解释器并不真的从文件加载数据（它甚至不去检查该文件是否存在）。

在这里可能会存在几个疑问：

Pig 到底要把文件加载到哪里呢？

数据是加载到内存吗？

即使这些数据可以放入内存，Pig 又如何处理数据呢？

应用程序中可能并不需要所有的数据(因为后续的语句可能会过滤数据)，因此加载数据没有意义。关键问题在于，在没有定义整个数据流之前，开始任何处理都是没有意义的。与此类似，Pig 验证 GROUP 和 FOREACH … GENERATE 语句，并把它们加入逻辑计划中，但并不执行这两条语句。让 Pig 开始执行的是 DUMP 语句。此时，逻辑计划被编译成物理计划，并执行。

由于 DUMP 是一个诊断工具，因此它总是会触发语句的执行。

STORE 命令与 DUMP 不同：

- 在交互模式下,STORE 和 DUMP 一样，总是会触发语句的执行（这一过程包含了 run 命令）
- 在批处理模式下，它不会触发执行（此时包含了 exec 命令）

这是为了性能考虑而进行的设计，在批处理模式下，Pig 会解析整个脚本，看看是否能够为减少写或读磁盘的数据量进行优化。

考虑如下的简单示例：

```
A = LOAD ‘/home/hadoop/simple.txt’;
B = FILTER A BY $1 == ‘banana’;
C = FILTER A BY $1 != ‘banana’;
STORE B INTO ‘output/b’;
STORE C INTO ‘output/c’;
```

关系 B 和 C 都是从 A 导出的,因此，为了防止读两遍 A,Pig 可以用一个 MapReduce 作业从 A 读取数据，并把结果写到两个输出文件中去：一个给 B，一个给 C。这一特性称为“多查询执行”(multiquery execution)。

Pig 的早期版本不包括多查询执行特性：批处理模式下脚本中的每个 STORE 语句都会触发语句的执行，从而每个 STORE 语句都有一个时应的作业。

可以在执行时对 Pig 使用-M 或-no_multiquery 选项来禁用多查询执行，从而恢复使用以前的设置。

Pig 的物理计划是一系列的 MapReduce 作业。

在本地模式下，这些作业在本地 JVM 中运行，而在 MapReduce 模式下，它们在 Hadoop 集群上运行。可以用 EXPLAIN 命令对一个关系查看 Pig 所创建的逻辑和物理计划（例如 EXPLAIN max_temp;)。

EXPLAIN 也会显示 MapReduce 计划，即显示物理操作是如何组成 MapReduce 作业的。这是一个查看 Pig 为查询运行多少个 MapReduce 作业的好办法。

下表概括了能够作为 Pig 逻辑计划一部分的关系操作：

类型	操作	描述
加载与存储	LOAD	从文件系统或其他存储加载数据，存入关系
	STORE	将一个关系存放到文件系统或其他存储中
	DUMP	将关系打印到控制台
过滤	FILTER	从关系中删除不需要的行
	DISTINCT	从关系中删除重复的行
	FOREACH…GENERATE	在关系中对每一行进行处理
	MAPREDUCE	以一个关系作为输入运行某个 MapReduce 作业
	STREAM	使用外部程序对一个关系进行变换
	SIMPLE	对一个关系进行随机取样

表概括了能够作为 Pig 逻辑计划一部分的关系操作：

类型	操作	描述
分组与连接	JOIN	连接两个或多个关系
	COGROUP	对两个或多个关系中的数据进行分组
	GROUP	在一个关系中对数据进行分组

	CROSS	创建两个或更多关系的乘积
排序	ORDER	根据一个或多个字段对某个关系进行排序
	LIMIT	将一个关系的元祖个数限定在一定数量内
组合和	UNION	合并两个或多个关系为一个关系
切分	SPLIT	把某个关系切分两个或多个关系

有些种类的语句并不会被加到逻辑计划中去。例如，诊断操作 DESCRIBE、EXPLAIN 以及 ILLUSTRATE。

这些操作是用来让用户能够与逻辑计划进行交互以进行调试的。DUMP 也是一种诊断操作。它只能用于与很小的结果集进行交互调试，或与 LIMIT 结合使用，来获得某个较大的关系的一小部分行。

当输出包含较多行的时候，应该使用 STORE 语句。这是因为 STORE 语句把结果存入文件而不是在控制台显示。

Pig Latin 的诊断操作：

类型	操作	描述
诊断操作	DESCRIBE	打印关系的模式
	EXPLAIN	打印逻辑和物理计划
	ILLUSTRATE	使用生成的输入子集显示逻辑计划的试运行结果

为了在 Pig 脚本中使用宏和用户自定义函数，Pig Latin 提供了 REGISTER、DEFINE 和 IMPORT 这三个语句：

类型	操作	描述
诊断操作	REGISTER	在 Pig 运行环境中注册一个 JAR 文件
	DEFINE	为宏、UDF、流式脚本或命令规范新建别名
	IMPORT	把在另一个文件中定义的宏导入脚本

因为这些命令不处理关系，所以它们不会被加入逻辑计划。相反，这些命令会被立即执行。

Pig 提供了与 Hadoop 文件系统和 MapReduce 进行交互的命令及其他工具：

类型	操作	描述
Hadoop 交互命令（与 Hadoop 文件系统进行交互的命令对在 Pig 处理前和处理后移动数据	cat	打印一个或多个文件的内容
	cd	改变当前目录
	copyFromLocal	复制本地文件或目录
	copyToLocal	将一个文件或目录从 Hadoop 文件系统复制到本地文件系统
	cp	把一个文件或目录复制到另一个目录
	fs	访问 Hadoop 文件系统外壳程序
	ls	打印文件列表信息
	mkdir	创建新目录
	mv	讲一个文件或目录移动到另一个目录

非常有用）	pwd	打印当前目录的路径
	rm	删除一个文件或目录
	rmf	强制删除文件或目录（即使文件或目录不存在也不会失败）
	kill	终止某个 MapReduce 作业
Hadoop MapReduce 工具	exec	在新的 Grunt 外壳程序中以批处理模式运行脚本
	help	显示可用的命令或选项
	quit	退出解释器
	run	在当前 Grunt 外壳程序中运行脚本
	set	设置 Pig 选项和 MapReduce 作业属性
	sh	在 Grunt 中运行外壳命令

文件系统相关的命令可以对任何 Hadoop 文件系统的文件或目录进行操作。这些命令和 Hadoop fs 命令很像，因为两者都是 Hadoop Filesystem 接口的简单封装。

可以使用 Pig 的 fs 命令访问所有的 Hadoop 文件系统外壳命令。例如，fs -ls 显示文件列表，fs -help 则显示所有可用命令的帮助信息。准确地说，使用哪个 Hadoop 文件系统是由 Hadoop 主节点配置文件文件中的 fs.defaultFS.name 属性决定的。

set 命令用于设置控制 Pig 行为的选项，包括所有 MapReduce 作业的属性。

debug 选项用于在脚本中打开或关闭调试日志（也可以在启动 Pig 时用-d 或-debug 选项控制日志的级别）：

```
set debug on
```

job.name 是另一个很有用的选项。它为 Pig 作业设定一个有意义的名称。这样，即可方便地知道在共享的 Hadoop 集群上有哪些 Pig MapReduce 作业是自己的。如果 Pig 正在运行某个脚本（而不是通过 Grunt 运行交互式查询），默认作业名称是基于脚本的名称的。

有两个命令可以运行 Pig 脚本：

- exec
- run

它们的区别是：

- exec 在一个新的 Grunt 外壳程序中以批处理方式运行脚本。因此，所有脚本中定义的别名在脚本运行结束后不能在外壳程序中再被访问。
- 如果用 run 运行脚本，那么效果就和在外壳中手工输入脚本的内容是一样的。因此，运行该脚本的外壳的命令历史中包含脚本的所有语句。
- 只能用 exec 进行多查询执行，即 Pig 以批处理方式一次执行一批语句。不能用 run 命令进行多查询执行。

Pig Latin 的设计中缺少原生的控制流语句。

如果要写需要条件逻辑或循环结构的程序，建议把 Pig Latin 嵌入其他语言之中，如 Python、Javascripl 或 Java,由该语言管理控制流。

在这一模型下，宿主脚本使用 comile-bind-run API 来执行 Pig 脚本，并获得脚本的状态。要想获得 API 的详细信息可以参考 Pig 的帮助文档。

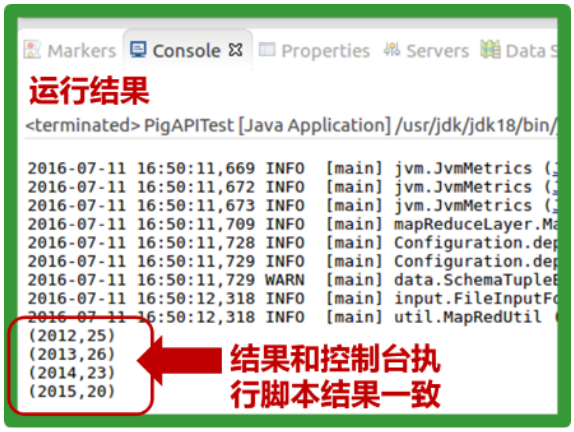
可以使用 PigServer API 轻松加载 Pig 脚本执行，仍然以之前的 Pig 脚本程序为例：

```
records = LOAD '/home/hadoop/temp.txt' AS (year:chararray,temperature:int,quality:int);

filtered_records = FILTER records BY temperature != 1 AND (quality==1 OR quality==2);

grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group,MAX(filtered_records.temperature);

DUMP max_temp;
```

在 Java 中调用该脚本的参考实现如下：

```
public class PigAPITest {

    public static void main(String[] args) {

        try {
            //构建API工具并加载执行脚本文件
            PigServer pigServer = new PigServer("local");
             pigServer.registerScript(new FileInputStream(new File(
             "/home/hadoop/max.pig")));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

程序执行结果如下：

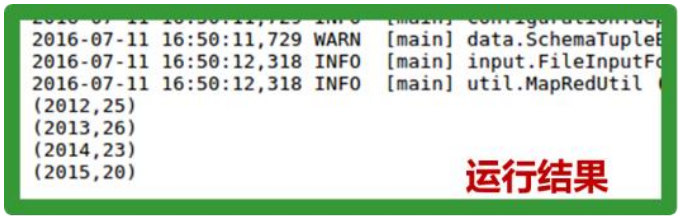


也可以在程序中动态定义执行的指令：

```
public class PigAPITest {

    public static void main(String[] args) {

        try {
            PigServer pigServer = new PigServer("local");
            // pigServer.registerScript(new FileInputStream(new
File(
            // "/home/hadoop/max.pig")));

            //也可以利用反复调用registerQuery方法的形式执行命令（每
次提供一行命令），但是该方法不支持DUMP，可以使用store（）方法将结果
保存在文件中
            // pigServer.registerQuery("records = LOAD
'/home/hadoop/temp.txt' AS
(year:chararray,temperature:int,quality:int);");
            String script = "records = LOAD '/home/hadoop/temp.txt'
AS (year:chararray,temperature:int,quality:int);";
            script += "filtered_records = FILTER records BY
temperature != 1 AND (quality==1 OR quality==2);";
            script += "grouped_records = GROUP filtered_records BY
year;";
            script += "max_temp = FOREACH grouped_records GENERATE
group,MAX(filtered_records.temperature);";
            script += "DUMP max_temp;";
            //构建API工具并利用内存缓冲流加载执行脚本文件
            byte[] srciptByte = script.getBytes();
            pigServer.registerScript(new
ByteArrayInputStream(srciptByte));

        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }

}
```

程序执行结果如下：



Pig 中也支持表达式的计算。可以通过计算表达式得到某个值。在 Pig 中，表达式可以作为包含关系操作的语句的一部分。

Pig 可以使用丰富的表达式类型。Pig 中的很多表达式类型和其他编程语言中的表达式相像。

Pig Latin 支持的表达式如下表所示：

类别	表达式	描述	示例
常数	文字	常量值	1.0,’a’
字段（位置指定）	$n	第 n 个字段（以 0 为基数）	$0
字段（名字指定）	f	字段名 f	year
字段（消除歧义）	r::f	分组或连接后关系 r 中名为 f 的字段	A::year
投影	c.$n,c.f	在容器 c（关系、包或元组）中的字段	records.$0,records.year
Map 查找	m#k	在映射 m 中键 k 所对应的值	items#’Coat’
类型转换	（t）f	将字段 f 转换为类型 t	（int）year
算数	x+y,x-y	加法和减法	$1+$2,$1-$2
	x*y,x/y	乘法和除法	$1*$2,$1/$2
	x%y	取模运算（取余数）	$1%$2
	+x,-x	正和负	+1,-1
条件	x？y：z	和 Java 中的条件三元运算符一致	quality==0?0:1
	x==y,x!=y	相等和不等	quality==0,quality!=1
	x>y,x<y	大于和小于	quality>0, quality<10
	x>=y,x<=y	大于等于额小于等于	quality>=1, quality<=3
	x matches y	正则表达式匹配	quality matches ‘[01459]’

	x is null	是空值	temperature is null
	x is not null	不是空值	temperature is not null
算数	x or y	逻辑或	q==0 or q==1
	x and y	逻辑与	q==0 and r==0
	not x	逻辑非	not q matches '[01459]'
函数型	fn(f1,f2…)	在 f1,f2 等字段上应用函数 fn	isGood(quality)
平面化	FLATTEN(f)	从包和元组中去除嵌套	FLATTEN(group)

Pig 有四种数值类型：int、long、float 和 double。它们和 Java 中对应的数值类型相同。Pig 还有 bytearray 类型，这类似于于表示二进制大对象（blob）的 Java 的 byte 数组。chararray 类型则类似于用 UTF-16 格式表示文本数据的 java.lang.String。chararray 也可以被加载或存储为 UTF-8 格式。

Pig 没有任何一种数据类型对应于 Java 的 boolean、byte、short，或 char 类型。这些数据类型都能方便地使用 Pig 的 int 类型或 chararray 类型表示。

在 Pig 中，数值、文本与二进制类型都是原子类型。Pig Latin 有三种用于表示嵌套结构的复杂类型：

- 元组（tuple）
- 包（bag）
- 映射（map）

下表列出 Pig 中的所有类型：

类别	数据类型	描述	示例
数值	int	32 位有符号整数	1
	long	64 位有符号整数	1L
	float	32 位浮点数	1.0F
	double	64 位浮点数	1
文本	chararray	UTF-16 格式的字符数组	'a'
二进制	bytearray	字节数组	
复杂类型	tuple	任何类型的字段序列	(1,'pomegranate')
	bag	元组的无序多重集合（允许重复元组）	{(1,'pomegranate'),(2)}
	map	一个键-值对的集合。键必须是字符数组，值可以使任何数据类型的数据	['a'#'pomegranate']

复杂类型通常从文件加载数据或使用关系操作进行构建。

PigStorage 用于约束从文件加载数据的分割（默认为制表符），如：

```
rs = LOAD ‘/home/hadoop/temp.txt’ USING PigStorage(’ ’);
```

则使用空格作为分隔符。

Pig 提供了内置的 TOTUPLE、TOBAG 以及 TOMAP 函数用来将表达式转化为元组、包以及映射。

虽然关系和包在概念上是相同的(本质上都是元组的无序多重集合),但事实上 Pig 对它们的处理稍有不同。

关系是顶层构造结构，而包必须在某个关系中。正常情况下，不必操心它们的区别。但是，对它们的使用有一些限制，这些限制可能导致错误。例如，不能根据包文字直接创建一个关系。因此，如下语句会运行失败：

```
A = {(1, 2), (3, 4)}; --Error
```

针对这种情况，最简单的解决办法是使用 LOAD 语句从文件加载数据。

另外,对待关系,不能像处理包那样把一个字段投影为一个新的关系(例如，使用位置符号，用 $0 指向 A 的第一个字段）：

```
B=A.$0
```

为此，必须使用关系操作将一个关系 A 转换为另一个关系 B:

```
B=FOREACH A GENERATE $0
```

Pig Latin 将来的版本可能采用相同的方法处理关系和包，以消除这种不一致性。

第5节 Pig 模式

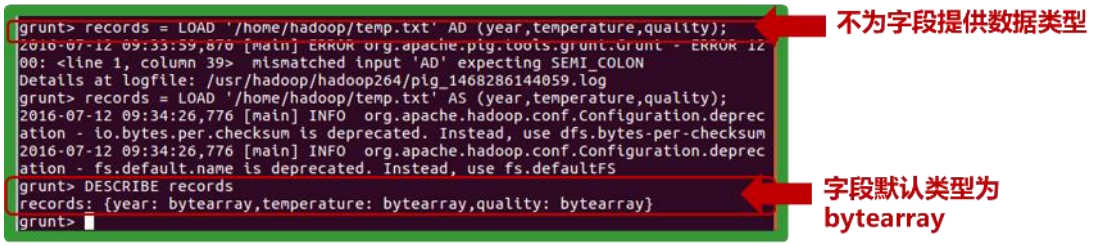
Pig 中的一个关系可以关联一个模式。模式为关系的字段指定名称和类型。可以通过 LOAO 语句的 AS 子句如何在关系上附以模式：

```
records = LOAD ’/home/hadoop/temp.txt’ AS ( year:chararray , temperature:int , quality:int);
```

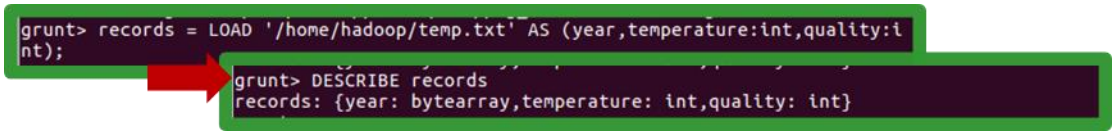
以上的模式声明中，年份也可以声明为整数类型，而不是 chararray 类型。如果要对年份这一字段进行算术操作（例如将它变为一个时间戳），那么用整数类型更为合适；相反，如果只是想把它作为一个简单的标识符，那么表示为 chararray 类型就更合适。Pig 的这种模式声明方式提供了很大的灵活性。这和传统 SQL 数据库要求在数据加载前必须先声明模式的方式截然不同。Pig 的设计目的是用它来分析不包含数据类型信息的纯文本输入文件，因此，它为字段确定类型的时机比 RDBMS 要晚也是理所当然的。

也可以在声明模式是忽略类型声明：

```
records = LOAD '/home/hadoop/temp.txt' AS ( year , temperature , quality);
```



在上面这个例子中，模式中只确定了字段的名称：year、temperature 和 quality。默认的数据类型为最通用的 bytearray， 即二进制串。还有更灵活的模式声明方法，不必为每一个字段都给出类型，可以让某些字段的类型为默认的 bytearray，就像如下模式声明示例中的 year 字段：

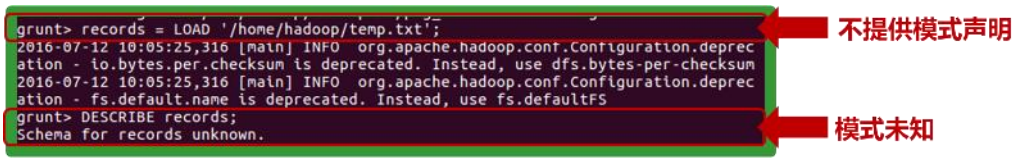


如果要用忽略类型声明方式确定模式，必须在模式中定义每 •个字段。同样，不能只确定字段的类型而不给出其名称。

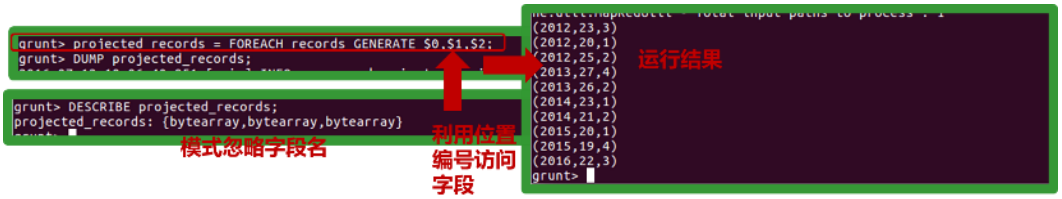
模式本身是可选的，可以省略 AS 子句，如下所示：

```
records = LOAD '/home/hadoop/temp.txt';
```

如果使用这种方法加载数据，在后续的脚本中无法获取该关系对应的明确模式：



对于没有对应模式的关系中的字段，只能使用位置符号进行引用：$0 表示关系中的第一个字段，$1 表示第二个，依此类推。它们的类型都是默认的 bytearray：



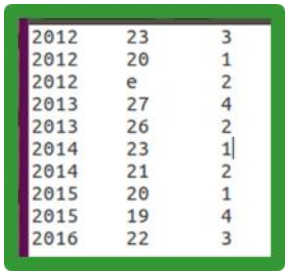
虽然不为字段指明类型很省事（特别是在撰写查询的开始阶段），但如果指定了字段的类型，我们能使 Pig Latin 程序更清晰，也使程序运行得更高效。因此，

一般情况下，建议指明字段的数据类型。

虽然在查询中声明模式的方式是灵活的，但这种方式并不利于模式重用。处理相同输入数据的一组 pig 查询常常使用相同的模式。如果一个查询要处理很多字段，那么在每个查询中维护重复出现的模式会很困难。

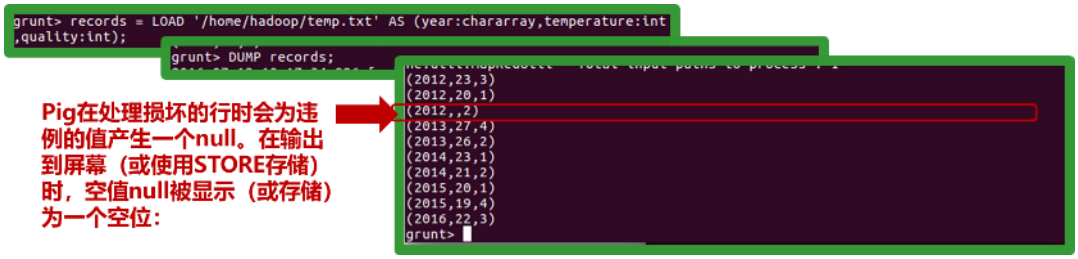
SQL 数据库在加载数据时，会强制检查表模式中的约束。例如，试图将一个字符串加载到声明为数值型的列会失败。

在 Pig 中，如果一个值无法被强制转换为模式中声明的类型，Pig 会用空值 null 替代。如果有如下天气数据输入，在定义为整数型的地方出现了一个字符“e”，接下来来看一下这一验证机制是如何工作的：



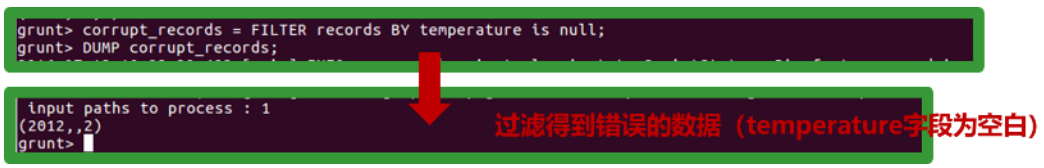
利用修改后的资源文件创建模式并显示：

```
records = LOAD '/home/hadoop/temp.txt' AS (year , temperature , quality);

DUMP records;
```



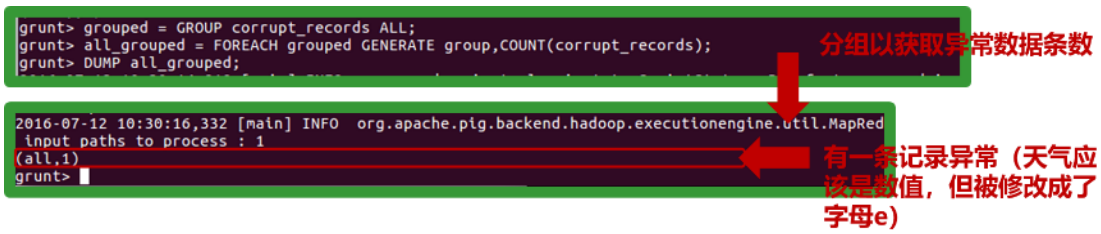
Pig 会为非法字段产生一个警告，但是不会终止处理。大的数据集普遍都有损坏的值、无效值或意料之外的值，因而逐步修正每一条无法解析的记录一般都不太现实。作为一种替代方法，可以一次性地把所有的非法记录都找出来，然后再一起处理它们。

可以使用 is null 表达式修正程序(因为这些记录表示写程序时犯了错误)，或把这些记录过滤掉（因为这些数据无法使用）：



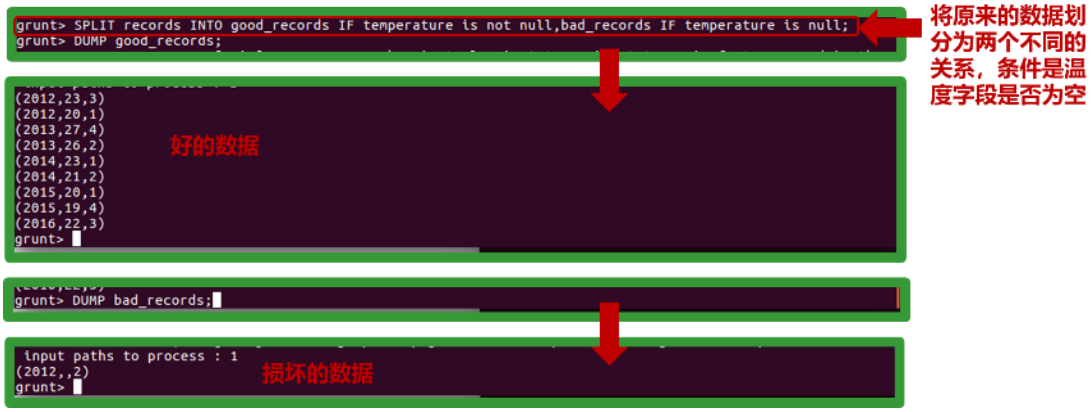
这里对 is null 操作的使用和 SQL 中类似。

可以在原始记录中获得更多的信息（例如标识符和无法被解析的值等），以帮助分析问题数据。可以使用如下对关系中的行进行计数的常用语句获得损坏记录的条数：

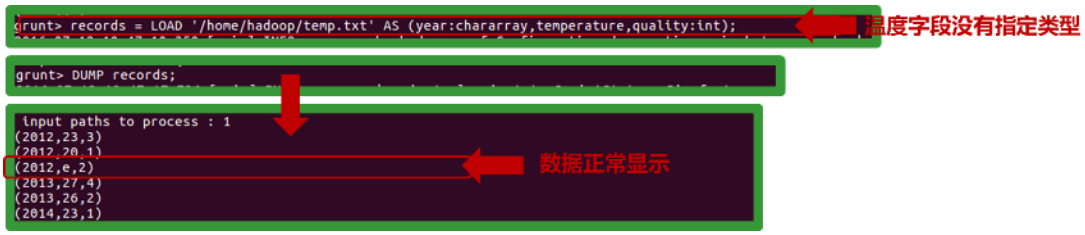


一个有用的技巧是使用 SPLIT 操作把数据划分成“好”和“坏”两个关系，然后再分别对它们进行分析：

```
SPLIT records INTO good_record IF temperature is no null,bad_records
IF temperature is null;

DUMP good_record;
```



需要注意的是，前面 temperature 数据类型未声明的情况下无法轻松检测到损坏的数据，因为它没有被当作 null 值：



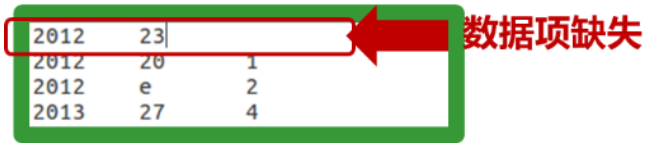
可以按照之前的方式对这个数据集合进行后续操作：

```
grunt> filtered_records = FILTER records BY temperature != 20;
grunt> grouped_records = GROUP filtered_records BY year;
grunt> max_temp = FOREACH grouped_records GENERATE group,MAX(filtered_records.temperature);
grunt> DUMP max_temp;
input paths to process : 1
(2012,23.0)
(2013,27.0)
(2014,23.0)
(2015,19.0)
(2016,22.0)
grunt>
```

在上面情况下，temperature 字段被解释为 bytearoay，因此在数据加载时，损坏的字段并没有被检测出来。在传输给 MAX 函数时，因为 MAX 只能处理数值类型，所以 temperature 字段被强制转换为 double 类型。损坏的字段不能被表示为 double，所以它被当作 null 处理，MAX 则会忽略这个值。

通常，最好的解决办法是在加载数据时声明数据类型，并在进行主要的处理前查看关系中缺失的值或损坏的值。

有时，因为有些字段缺失，损坏的数据被显示为比较短的元组。可以用 SIZE 函数对它们进行过滤，如下修改的数据文件所示：



利用 Pig 加载该数据文件：

```
A = LOAD '/home/hadoop/temp.txt';
```

显示该关系的内容：

```
DUMP A;
```

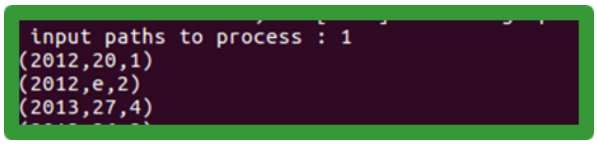
```
input paths to process : 1
(2012,23)
(2012,20,1)
(2012,e,2)
(2013,27,4)
```

利用 Size 函数过滤元素不足的元组来保证数据的一致性：

```
B = FILTER A BY SIZE(TOTUPLE(*))>2;
```

显示过滤后的结果：

```
DUMP B;
```



在Pig中,不用为数据流中每一个新产生的关系声明模式。在大多数情况下，Pig能够根据关系操作的输入关系的模式来确定输出结果的模式。

那么模式是如何传播到新关系的呢？有些关系操作并不改变模式。因此，LIMIT操作（对一个关系的最大元组数进行限制）产生的关系就和它所处理的关系其有相同的模式。

对于其他操作，情况可能更复杂一些。例如，UNION操作将两个或多个关系合并成一个，并试图同时合并输入关系的模式。如果这些模式由于数据类型或字段个数不同而不兼容，那么UNION产生的模式是未知的。

针对数据流中的任何关系，都可以使用DESCRIBE操作来获取它们的模式。

如果要重新定义一个关系的模式,可以使用带AS子句的FOREACH…GENERATE操作来定义输入关系的一部分或全部字段的模式。

第6节 Pig 函数

Pig Latin中的函数分为四种类型：

- 计算函数
- 过滤函数
- 加载函数
- 存储函数

计算函数获取一个或多个表达式作为输入，并返回另一个表达式。

MAX就是一个内置计算函数的例子，它返回一个包内所有项中的最大值。

有些计算函数是“聚集函数”(aggregate function)，这意味着它们作用于数据的“包”(bag)，并产生一个标量值(scalarvalue)。MAX就是一个聚集函数。

此外，很多聚集函数是“代数的”(algebraic)。也就是说这些函数的结果可以增量计算。用MapReduce的术语来表示，通过使用combiner进行计算，代数函数的计算效率可以提高很多。MAX是一个代数函数，而计算一组值的“中位数”(median）的函数则不是代数函数。

过滤函数是一类特殊的计算函数。这类函数返回的是逻辑布尔值。正如其名，过滤函数被FILTER操作用于移除不需要的行。它们也可以用于其他以布尔条件作为输入的关系操作或用于使用布尔或条件表达式的表达式。

IsEmpty就是一个内置过滤函数。它测试一个包或映射是否包含有元素。

加载函数指明如何从外部存储加载数据到一个关系。存储函数指明如何把一

个关系中的内容存到外部存储。通常，加载和存储函数由相同的类型实现。例如，PigStorage 从分隔的文本文件中加载数据，也能以相同的格式存储数据。

Pig 有很多内置的函数，下表列出了其中的一部分。完整的内置函数列表包括很多标准数学和字符串函数。Pig 每个发布版本文档都包含这些函数列表：

类型	操作	描述
计算	AVG	计算包中项的平均值
	CONCAT	把两个字节数组或字符数组连接成一个
	COUNT	计算一个包中非空值的项的个数
	COUNT_STAR	计算一个包中的项的个数，包括空值
	DIFF	计算两个包的差。如果两个参数不是包，那么如果它们相同，则返回一个包含这两个参数的包，否则返回一个空包
	MAX	计算一个包中的最大值
	MIN	计算一个包中的最小值
	SIZE	计算一个类型的大小。数值型的大小总是 1，对于字符数组，它返回字符数组的个数；对于容器，它返回其中项的个数。
	SUM	计算一个包中的值的总和
	TOBAG	把一个或多个表达式转换为单独的元组，然后把这个元组放入包
	AVG	计算包中项的平均值
	TOKENIZE	对一个字符数组进行标记解析，并把结果放入一个包
	TOMAP	将偶数个数表达式转换为一个键-值对的映射
	TOP	计算包中最前面的 n 个元组
	TOTUPLE	将一个或多个表达式转换为一个元组
过滤	IsEmpty	判断一个包或映射是否为空
加载/存储	PigStorage	用字段分隔文本格式加载或存储关系。用一个可设置的分隔符（默认为一个制表符）把每一行分隔为字段后，将它们分别存储于元组的各个字段。这是不指定加载 l 存储方式时的默认存储函数
	BinStorage	从二进制文件加载一个关系或把关系存储到二进制文件中。该函数使用基于 HadoopWritable 对象的 Pig 内部格式
	TextLoader	从纯文本格式加载一个关系。每一行对应于一个元组。每个元组只包含一个字段，即该行文本
	JsonLoader,JsonStorage	从（Pig 定义的）JsoN 格式加载关系，或将关系存储为 JsON 格式。每个元组存储为一行

HBaseStroage	从 HBase 表中加载关系，或将关系存储至 HBase 表中

如果表中没有需要的函数，可以自己写，但在此之前，可以看一下 Piggy Bank。这是一个 Pig 社区共享 Pig 函数的库。例如，Piggy Bank 中有 Avro 数据文件、CSV 文件、Hive RCFiles、Sequence Files，以及 XML 文件的加载和存储函数。Pig 网站有关于如何浏览和获取 Piggy Bank 函数的指导。

如果 Piggy Bank 也没有所需要的函数，可以撰写自己的函数（如果函数很通用，可以考虑把它贡献给 Piggy Bank，好让其他人也能从中受益）。这些函数就是“用户自定义函数”(user-defined function,UDF)。

Pig 设计者认识到以插件形式提供使用用户定制代码的能力是一个关键问题，但这也是数据处理中最琐碎的工作。因此，他们的设计简化了用户自定义函数的定义与使用。

推荐编写 Java UDF。也可以使用 Python 或 Javascript 来撰写 UDF，它们都使用 JavaScripting API 来运行。

接下拉通过编写一个过滤不满足气温质量读数要求的天气记录的函数来演示如何写过滤函数。实现的基本思路是修改下面的代码：

```
filtered_records = FILTER records BY temperature !=20 AND (quality==1
or quality==2);
```

修改后的代码：

```
filtered_records = FILTER records BY temperature !=20 AND
isGood(quality);
```

这样写有两个好处：使 Pig 脚本更精简；而且还封装了处理逻辑，以便轻松重用于其他脚本。如果只是编写一个即时查询，我们可能不需要麻烦地写一个 UDF。只有需要不断进行相同的处理时，才需要如此写可重用的 UDF。

所有的过滤函数都是 FilterFunc 的子类，而 FilterFunc 本身是 EvalFunc 的子类，EvalFunc 类结构如下：

```
public abstract class EvalFunc<T> {
    public abstract T exec(Tuple input) throws IOException;
}
```

EvalFunc 只有一个抽象方法 exec()。它的输入是一个元组，输出则只有一个值，其（参数化）类型为 T。

输入元组的字段包含传递给函数的表达式，在这个例子里，它是一个整数。对于 FilterFunc,T 是 Boolean 类型的，对于那些不应该被过滤掉的元组，该方法应该返回 true。

对于案例中的气温质量过滤器，我们要写一个 IsGoodQuality 类，扩展

FilterFunc 并实现 exec()。Tuple 类本质上是一个与某个类型关联的对象列表。我们只关心第一个字段(因为函数只有一个参数)。我们用 Tuple 的 get()方法，根据序号来获取这个字段。该字段的类型是整型。因此，如果它非空，我们就对它进行类型转换，检查它是否表示气温读数是正确的，并根据检查结果返回相应的值：true 或 false。

过滤天气质量的过滤函数示例代码如下：

```
public class IsGoodQuality extends FilterFunc {

    @Override
    public Boolean exec(Tuple tuple) throws IOException {

        if (null == tuple || 0 == tuple.size()) {
            return false;
        }
        Object obj = tuple.get(0);
        if (null == obj) {
            return false;
        }
        int quality = ((Integer) obj).intValue();
        //满足条件则返回true表示本条数据应该要保留
        return quality == 1 || quality == 2;
    }


}
```

为了使用新函数，首先需要进行编译，并把它打包到一个 JAR 文件。然后，我们通过 REGISTER 操作指定文件的本地路径（不带引号），告诉 Pig 这个 JAR 文件的信息：

```
REGISTER 函数 jar 包路径;
```

使用自定义函数：

```
filtered_records = FILTER records BY temperature !=20 AND
com.chinasofti.hadooptest.pig. IsGoodQuality(quality);

DUMP filtered_records;
```

Pig 把函数名作为 Java 类名并试图用该类名来加载类以完成函数调用。(这也就是为什么函数名是大小写敏感的，因为 Java 类名是大小写敏感的。)在搜索类的时候，Pig 会使用包含已注册 JAR 文件的类加载器。运行于分布式模式时，Pig 会确保将 JAR 文件传输到集群。

内置函数的解析也使用同样的方式处理。

内置函数和 UDF 的处理有两个区别：Pig 会搜索一组内置包。因此，对于内置函数的调用并不一定要提供完整的名称。例如，函数 MAX 实际上是由包 org.apache.pig.builtin 中的类 MAX 实现的，是 Pig 搜索的内置包，所以在 Pig 程序中可以使用 MAX 而不需要用 org.apache.pig.builtin.MAX。

可以使用 DEFINE 操作为函数定义别名，以缩短函数名：

```
DEFINE isGood com.chinasofti.hadooptest.pig. IsGoodQuality();

filtered_records = FILTER records BY temperature !=20 AND
isGood(quality);

DUMP filtered_records;
```

执行这个脚本将会得到和上面直接使用类名作为函数名的调用相同的结果。需要在一个脚本里多次使用一个函数时，为函数定义别名是一个好办法。如果要向 UDF 的实现类传递参数，必须定义别名。

只有在气温质量字段的类型定义为 int 时,前面定义的过滤器才能正常工作。如果没有类型信息，这个 UDF 就不能正常处理。因为此时该字段的类型是默认类型 bytearray，表示为 DataByteArray 类。因为 DataByteArray 不是整型，因此类型转换失败。

修正这一问题最直接的办法是在 exec() 方法中把该字段转换成整型。但更好的办法是告诉 Pig 该函数所期望的各个字段的类型。EvalFunc 为此提供了 getArgToFuncMapping() 方法，可以重载这个方法来告诉 Pig 第一个字段应该是整型。

在自定义函数类中加入明确类型示例代码：

```
    @Override
    public List<FuncSpec> getArgToFuncMapping() throws
FrontendException {
        // TODO Auto-generated method stub
        List<FuncSpec> funcSpecs = new ArrayList<FuncSpec>();
        funcSpecs.add(new FuncSpec(this.getClass().getName(), new
Schema(
                new Schema.FieldSchema(null, DataType.INTEGER))));
        return funcSpecs;

    }
```

这个方法为传递给 exec()方法的那个元组的每个字段返回一个 Funcspec 对象。在这个例子里，只有一个字段。我们构造一个匿名 Fieldschema（因为 Pig 在进行类型转换时忽略其名称，其名称以 null 传递）。该类型使用 Pig 的 DataType 类的常量 INTEGER 进行指定。

使用这个修改后的函数，Pig 将尝试把传递给函数的参数转换成整型。如果无法转换这个字段，则把这个字段传递为 null。如果字段为 null,exec()方法返回的结果总是 fase。在这个应用中，因为想要在过滤掉质量字段无法理解的记录，所以这样做很合适。

同样的，开发人员也可以自定义实现计算 UDF，例如，要实现一个在字段值后串联 Chinasofti 的计算 UDF 示例：

```
public class AppendUDF extends EvalFunc<String> {
    @Override
    public String exec(Tuple tuple) throws IOException {
        if (null == tuple || 0 == tuple.size()) {
            return null;
        }
        Object obj = tuple.get(0);
        if (null == obj) {
            return null;
        }
        return obj.toString() + "Chinasofti";
    }
    @Override
    public List<FuncSpec> getArgToFuncMapping() throws
FrontendException {
        // TODO Auto-generated method stub
        List<FuncSpec> funcSpecs = new ArrayList<FuncSpec>();
        funcSpecs.add(new FuncSpec(this.getClass().getName(), new
Schema(
                new Schema.FieldSchema(null, DataType.CHARARRAY))));
        return funcSpecs;
    }
}
```

计算 UDF 的使用结果如下：

之前已经探讨 Pig 如何从外部存储加载数据来进行处理。相似，存储处理结果也是非常直观的。下面的例子使用 PigStorage 存储为以冒号分隔的纯文本值：

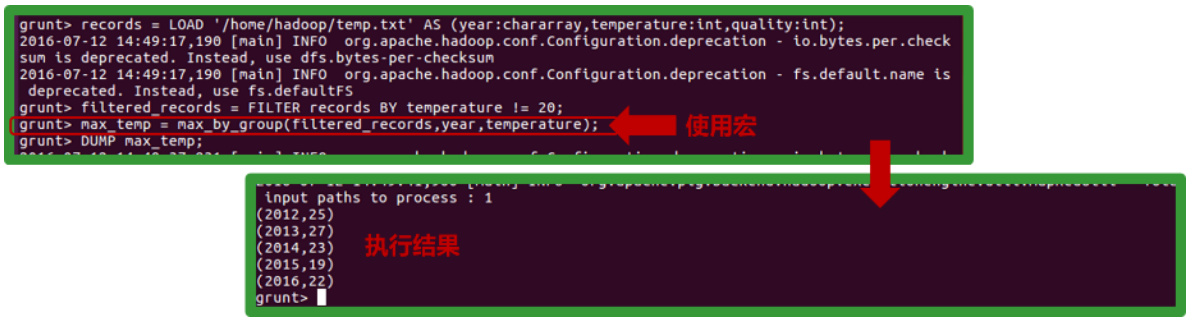
```
STORE A INTO ‘out‘ USING PigStorage(’ :’ );
```

另一种脚本代码复用的方法是使用宏。宏提供了在 Pig Latin 内对可重用的 Pig Laiin 代码进行打包的功能。例如，我们可以把对关系进行分组，然后在每一组内查找最大值的 Pig Laiin 程序抽出来，定义成如下的宏：

```
DEFINE max_by_group(x,group_key,max_field) RETURNS Y{
A = GROUP $X BY $group_key;
$Y = FOREACH A GENERATE group,MAX($X.$max_field);
};
```

这个宏的名字为 max_by_group。它包含三个参数：一个关系 X 以及两个字段名：group_key 和 max_field。它返回一个关系 Y。在宏的体内，参数和返回别名通过使用 $ 前缀进行引用，例如$X。

宏的使用方法如下：



在运行时，Pig 将使用宏定义展开宏。一般情况下看不到程序展开后的形式，这是因为 Pig 是在内部进行这 •操作的。但是，有些情况下，在撰写和调试宏时，如果能看到展开后的形式会比较有益。可以通过给 pig 命令传递-dryrun 参数让 Pig 只进行宏扩展（而不执行脚本）

运行时传递给宏的参数（filtered_records、year，以及 temperature)会取代宏定义中的参数名。

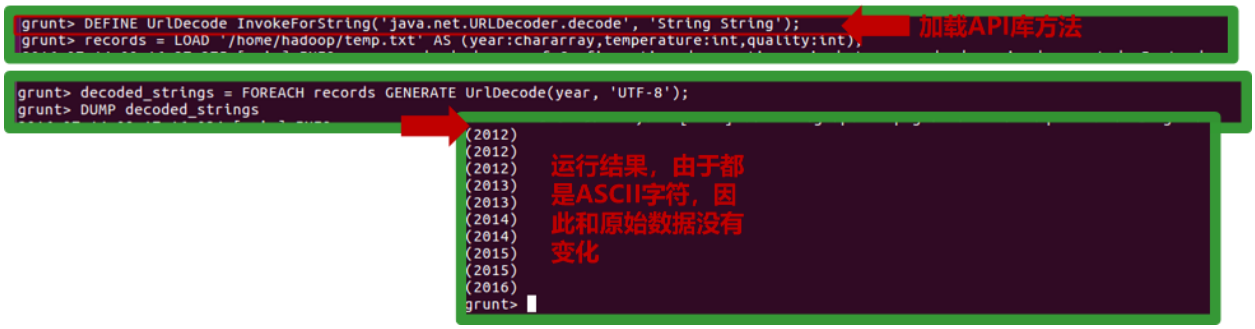
宏定义中不使用 $ 前缀的别名，例如示例中的 A，仅在宏定义的局部有效。在展开时，它们被重写，以避免和程序的其他部分的别名冲突。A 在展开后的形式可能为 macro_max_by_group_A_0。

为了方便重用，宏可以定义在 Pig 脚本以外的文件中。这种情况下，需要在使用宏的脚本中导入这些文件。导入（import）语句的形式如下：

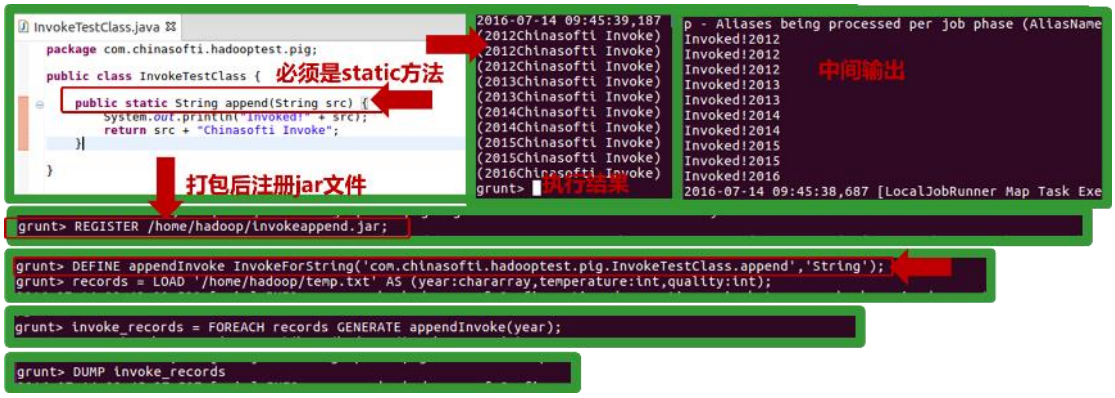
```
IMPORT  ‘/home/hadoop/max.macro’;
```

有时希望使用由某个 Java 库提供的方法，而不是自己编写 UDF。

动态调用器（invoker）可以在 Pig 脚本中直接调用 Java 静态方法。使用这种方法的代价是，对于方法的调用是通过 Java 的反射（reflection）机制进行的。如果要为一个很大的数据集的每行记录进行调用，则会导致巨大的处理开销。所以，对于需要重复运行的脚本，最好使用专用的 UDF。



可以自己编写静态方法供 Pig 调用，编译后打包供 Pig 使用：



第7节 更多数据操作

FOREACH…GENERATE 操作

FOREACH…GENERATE 操作用于逐个处理一个关系中的行。它可用于移除字段或创建新的字段。

PARALLEL

使用关键字 PARALLEL 来设置要使用多少个 reducer，如：

```
grouped_records = GROUP records BY year PARALLEL 30;
```

STREAM 操作

STREAM 操作用外部程序或脚本对关系中的数据进行变换。这一操作的命名对应于 Hadoop 的 Streaming，后者为 MapReduce 的提供类似能力。STREAM 可以使用内置命令作为参数。下面的例子使用 Unix cut 命令从 A 中每个元组抽取第二个字段。注意，命令及其参数要用反向撇号引用：

```
C = STREAM A THROUGH `cut -f 2`;
```

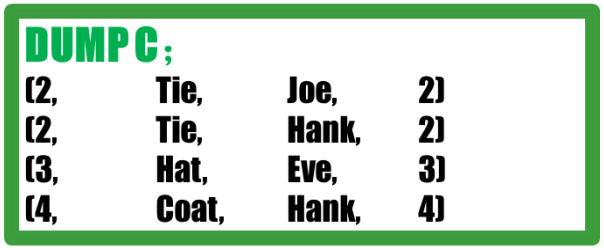
在 MapReduce 中对数据集进行连接操作需要程序员写不少程序，Pig 为连接操作提供很好的内置支持，简化了数据集的连接。因为只有非规范化的大规模数据集才最适宜使用 Pig（或 MapReduce）这样的工其进行分析，因此连接在 Pig 中的使用频率远小于在 SQL 中的使用频率。

如果有如下关系：

可以在两个关系的数值型（标识符）属性上对它们进行连接操作：

```
C = JOIN A BY $0,B BY $1;
```

操作结果如下：



这是一个典型的“内连接”(inner join)操作：两个关系元组的每次匹配都和结果中的一行相对应。因为连接谓词为相等，所以这其实是一个等值连接。结果中的字段由所有输入关系的所有字段组成。

如果要进行连接的关系太大，不能全部放在内存中，则应该使用通用的连接

操作。如果有一个关系小到能够全部放在内存中，则可以使用特殊的分段复制连接,它把小的输入关系发送到所有 mapper，并在 map 端使用内存查找表对分段的较大的关系进行连接：

```
C = JOIN A BY $0,B BY $1 USING “replicated”;
```

Pig 也支持通过使用类似于 SQL 的语法进行外连接。例如：

```
C = JOIN A BY $0 LEFT OUTER,B BY $1;
```

操作结果如下：

```
DUMP C;
(1,     Scarf,    ,       )
(2,     Tie,      Joe,    2)
(2,     Tie,      Hank,   2)
(3,     Hat,      Eve,    3)
(4,     Coat,     Hank,   4)
```

JOIN 结果的结构总是“平面”的，即一组元组。COGROUP 语句和 JOIN 类似，但是不同点在于，它会创建一组嵌套的输出元组集合。如果希望利用如下语句中输出结果那样的结构，那么 COGROUP 将会有用：

```
D = COGROUP A BY $0 ,B BY $1;
```

操作结果如下：

```
DUMP D;
(0,{},{(Ali,0)} )
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Joe,2),(Hank,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

COGROUP 为每个不同的分组键值生成一个元组。每个元组的第一个字段就是那个键值。其他字段是各个关系中匹配该键值的元组所组成的包。第一个包中包含关系 A 中有该键值的匹配元组。同样，第二个包中包含关系 B 中有该键值的匹配元组。如果某个键值在一个关系中没有匹配的元组，那么对应于这个关系的包就为空。

这是一个外连接的例子。COGROUP 的默认类型是外连接。可以使用关键词 OUTER 来显式指明使用外连接，COGROUP 产生的结果和前一个语句相同：

```
D = COGROUP A BY $0 OUTER ,B BY $1;
```

也可以使用关键词 INNER 让 COGROUP 使用内连接的语义,剔除包含空包的行。

INNER 关键词是针对关系进行使用的，因此如下语句只去除关系 A 中不匹配的行：

```
E = COGROUP A BY $0 INNER ,B BY $1;
```

操作结果如下：

```
DUMP E;
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Joe,2),(Hank,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

可以把这个结构平面化：

```
F = FOREACH E GENERATE FLATTEN (A), B.$0
```

操作结果如下：

```
DUMP F;
(1,Scarf),{})
(2,Tie,{(Joe,2),(Hank,2)})
(3,Hat,{(Eve,3)})
(4,Coat,{(Hank,4)})
```

把 COGROUP、INNER 和 FLATTEN（消除嵌套）组合起来使用相当于实现了内连接：

```
G = COGROUP A BY $0 INNER ,B BY $1 INNER;
H = FOREACH E GENERATE FLATTEN ($1), GLATTEN($2);
```

操作结果如下：

```
DUMP H;
(2,     Tie,    Joe,    2)
(2,     Tie,    Hank,   2)
(3,     Hat,    Eve,    3)
(4,     Coat,   Hank,   4)
```

如果要连接的键由多个字段组成，则可以在 JOIN 或 COGROUP 语句的 BY 子句中把它们都列出来。这时要保证每个 BY 子句中的字段个数相同。如：

```
max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY
```

```
(usaf, wban) PARALLEL 30;
```

Pig Latin 包含叉乘，也称笛卡儿积操作。这一操作把一个关系中的每个元组和第二个中的所有元组进行连接（如果有更多的关系，那么这个操作就进一步把结果逐一和这些关系的每一个元组进行连接）。

这个操作的输出结果的大小是输入关系的大小的乘积。输出结果可能会非常大：

```
I = CROSS A, B;
```

GROUP 语句则一个关系中的数据进行分组。GROUP 不仅支持对字段键值进行分组,还可以使用表达式或用户自定义函数作为分组键。例如，有如下关系 A：

```
DUMP A;
(Joe,    cherry)
(Ali,    apple)
(Joe,    banana)
(Eve,    apple)
```

根据这个关系的第二个字段的字符个数进行分组：

```
B = GROUP A BY SIZE($1);
```

操作结果如下：

```
DUMP B;
(5,{(Ali,apple)},{Eve,apple})
(6,{(Joe,cherry)},{(Joe,banana),})
```

有两种特殊的分组操作：

ALL,ALL 把一个关系中的所有元组放入一个包。这和使用某个常量函数作为分组函数所获得的结果一样：

```
C = GROUP A ALL;
```

操作结果如下：

```
DUMP C;
(all,{(Joe,cherry)},{Ali,apple},{Joe,banana},{Eve,apple})
```

ANY，关键词 ANY 用于对关系中的元组随机分组。它对于取样非常有用。

Pig 中的关系是无序的。

Pig 按什么顺序来处理关系中的行是不一定的。特别是在使用 DUMP 或 STORE 检索关系中的内容时 Pig 可能以任何顺序输出结果。

如果想设置输出的顺序，可以使用 ORDER 操作按照某个或某几个字段对关系中的数据进行排序。

默认的排序方式是对具有相同类型的字段值使用自然序进行排序，而不同类型字段值之间的排序则特定的，例如，一个元组总是小于一个包。

排序示例：

```
B = ORDER A BY $0,S1 DESC;
```

排序后关系的后续处理并不保证能够维持已排好的顺序。例如：

```
C = FOREACH B GENERATE *;
```

即使关系 C 和关系 B 有相同的内容，关系 C 用 DUMP 或 STORE 仍然可能产生以任意顺序排列的输出结果。正是由于这样，通常只在获取结果前一步才使用 ORDER 操作。

LIMIT 语句对于限制结果的大小以快速获得一个关系样本，非常有用。取原型化操作（即 ILLUSTRATE 命令）则更适用于根据数据产生有代表性的样本。

LIMIT 语句可紧跟 ORDER 语句使用，来获得排在最前而的 n 个元组。通常，LIMIT 会随意选择一个关系中的 n 个元组。但是，当它紧跟 ORDER 语句使用时，OROER 产生的次序会继续保持（这和其他操作不保持输入关系数据顺序的规则不同，是一个例外）：

```
D = LIMIT B 2;
```

如果所给的限制值远远大于关系中所有元组个数的总数，则返回所有元组(LIMIT 操作没有作用)。

使用 LIMIT 能够提升系统的性能。因为 Pig 会在处理流水线中尽早使用限制操作，以最小化需要处理的数据总量。因此，如果不需要所有输出数据，就应该用 LIMIT 操作。

把几个关系组合在一起，可以使用 UNION 语句：

```
C = UNION A, B;
```

C 是关系 A 和 B 的并集。因为关系本身是无序的，因此 C 中元组的顺序是不确定的。可以对两个模式不同或字段个数不同的关系进行并操作。Pig 会试图合并正在执行 UNION 操作的两个关系的模式。如果两个模式是不兼容的（字段数或类型不同），结果 C 没有模式。如果输出关系没有模式，脚本就需要能够处理字段个数和数据类型都不同的元组。

SPLIT 操作是 UNION 操作的反操作。它把一个关系划分成两个或多个关系。在之前数据空值和验证阶段已经使用过 SPLIT 将有效数据和错误数据划分为两个不同的关系。

第6章 基于 HDFS 的数据仓库 Hive

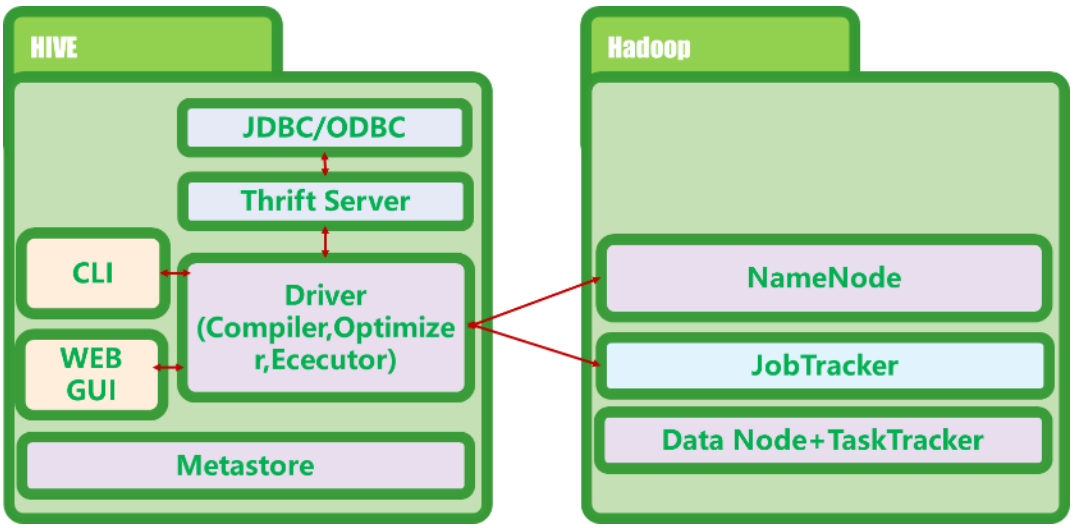
第1节 Hive 基本介绍

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载（ETL ），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL ，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

Hive 一般在工作站上运行，它提供了一个 SQL 解析引擎，把 SQL 查询转换为一系列在 Hadoop 集群上运行的 MapReduce 作业。Hive 把数据组织为表，通过这种方式为存储在 HDFS 的数据赋予结构。元数据（如表模式）存储在 metastore 数据库中。

Hive 的表其实就是 HDFS 的目录，按表名把文件夹分开。如果是分区表，则分区值是子文件夹，可以直接在 M/R Job 里使用这些数据。

Hive 的基本系统架构如下图所示：



Hive 的用户接口主要有三个：CLI，JDBC/ODBC 和 WebUI。

- CLI，即 Shell 命令行
- JDBC/ODBC 与使用传统数据库 JDBC 的方式类似
- WebGUI 是通过浏览器访问 Hive

Hive 将元数据存储在数据库中(metastore)，目前只支持 mysql、derby。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。

Hive 提供的解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用执行

Hive 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成。它没有专门的数据存储格式。Hive 数据的逻辑存储结构主要包括：数据库、文件、表、视图。Hive 默认可以直接加载文本文件，还支持 Hadoop 的 SequenceFile，创建表时，指定 Hive 数据的列分隔符与行分隔符，Hive 即可解析数据。

为了方便后续的数据操作，Hive 也定义了对应的数据类型，Hive 中的数据类型包括：

基本数据类型

- tinyint/smallint/int/bigint
- float/double
- boolean
- string

复杂数据类型

- Array/Map/Struct

需要注意的是，Hive 中没有 RDBMS 中常用的代表日期的 date/datetime 类型。

下表中列举出了 Hive 和传统数据产品的主要差异：

查询语言	HiveQL	SQL
数据存储位置	HDFS	Raw Device or 本地 FS
数据格式	用户定义	系统决定
数据更新	不支持	支持
索引	新版本有，但弱	有
执行	MapReduce	Executor
执行延迟	高	低
可扩展性	高	低
数据规模	大	小

Hive 的表在逻辑上由存储的数据和描述表中数据形式的相关元数据组成。数据一般存放在 HDFS 中，但它也可以放在其他任何 Hadoop 文件系统中，包括本地文件系统或 S3。

Hive 把元数据存放在关系数据库中，而不是放在 HDFS 中。

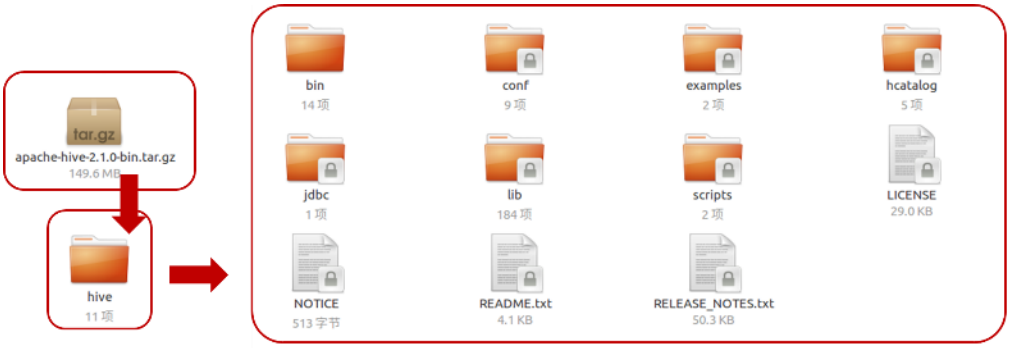
很多关系数据库提供了多个“命名空间”(namespace）的支持，用户和应用可以被隔离到不同的数据库或模式中。

Hive 也对此提供了同样的支持，可用的命令包括 CREATE DATABASE dbname、USE dbname 以及 DROP DATABASE dbname 这样的语句，可以通过

dbname.tablename 来完全限定某一张表。如果没有指明数据库，那么所指的是在 default 数据库中的表。

第2节 Hive 安装

Hive 的安装也很简单，在官方网站上下载 Hive 的发行包，解压后添加配置文件即可启动 Hive：



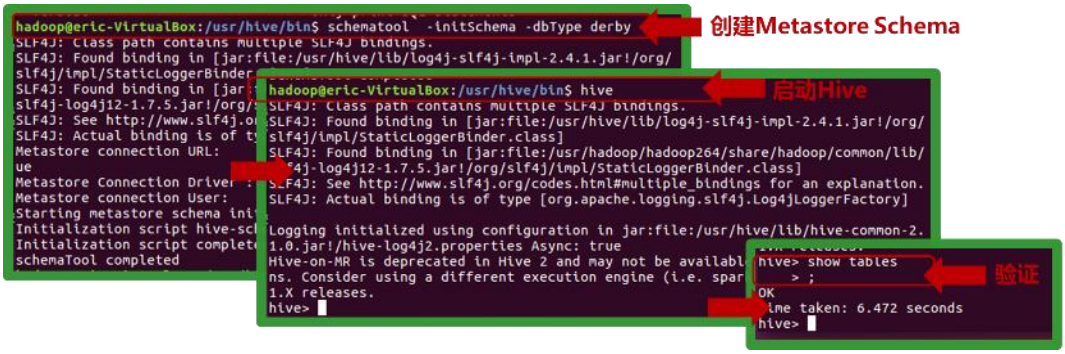
Hive 解压成功后向系统中添加必要的环境变量：

```
export HIVE_INSTALL=/usr/hive/
export PATH=$PATH:$HIVE_INSTALL/bin
```

默认启动 Hive 将使用 derby 作为 Metastore。在启动 Hive 之前，需要对 Hive 的 Metastore 数据库进行初始化，运行如下命令：

```
schematool -initSchema -dbType derby
```

Meta 初始化成功后，就可以直接使用 Hive 启动脚本来启动 Hive 的交互环境了（如果启动不成功，注意将 Hive 所在的目录及存放 Metastore 的目录权限修改为可写入），Hive 的启动脚本为 HIVE_HOME/bin/hive，启动 Hive 后可以利用类似 MySQL 的列举数据表的命令 show tables 来验证 Hive 是否启动成功：



如果 Hive 因为 HDFS Namenode 处于安全模式而不能启动，使用如下命令离开安全模式：

```
hdfs dfsadmin -safemode leave
```

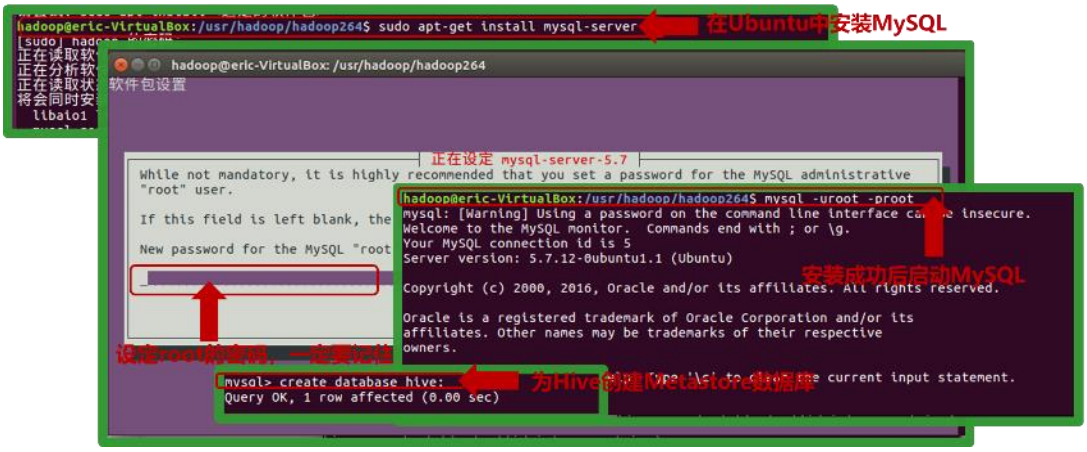
如果 Hive 成功启动，接下来就可以创建一个测试表，用于验证 Hive 数据的存放。Hive 的基本建表语句和 SQL 一致，使用如下格式可以构建一个测试表：

```
create table test(id int,name string);
```

Metastore 是 Hive 元数据的集中存放地。Metastore 默认使用内嵌的 derby 数据库作为存储引擎。但事实上，默认的 Derby Metastore 仅适用于测试验证环境，因为使用它作为 Metastore 存在一个非常重要的缺陷：一次只能打开一个会话，因此，在开发和生产环境中都会使用 MySQL 作为外置存储引擎，以支持多用户同时访问。

要使用 MySQL 作为 Metastore 数据库，首先需要安装 MySQL 服务器，以 Ubuntu 为例，APT 包管理器可以自动安装配置 MySQL 服务，在终端中执行如下命令：

```
sudo apt-get install mysql-server
```



将 Hive 安装目录中 conf 文件夹中的 hive_default.xml.template 拷贝一份并重名为 hive-site.xml，加入 MySQL Metastore 连接配置：

```
<?xml version="1.0"?>
<configuration>
    <property>
```

```
        <name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://127.0.0.1:3306/hive?characterEncoding=UTF-
8</value>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>root</value>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>root</value>
    </property>
</configuration>
```

接下来重新执行 Hive 的 Metastore 初始化操作，注意将数据库类型参数变更为 mysql：

```
schematool -initSchema -dbType mysql
```

Metastore 初始化成功后，再次启动 Hive 交互环境，此时 Hive 的 Metastore 服务已经成功迁移到了 MySQL。

第3节 Hive 基本使用

在上一节中，成功安装 Hive 后利用 hive 脚本开启的 Hive Shell 外壳程序是和 Hive 交互、发出 HiveQL 命令的主要方式。在交互环境中操作数据的查询语言被称为 HiveQL，是 SQL 的一种“方言”。它的设计在很大程度上深受 MySQL 的影响。因此，如果开发人员熟悉 MySQL，会觉得 Hive 的数据操作很亲切。

第一次启动 Hive 时，可以通过之前使用的列出 Hive 的表来检查 Hive 是否正常工作(由于 Hive 刚安装，没有执行任何的建表操作，此时应该没有任何表)：

```
show tables;
```

和 MySQL 一样，Hive 交互环境中的命令必须以分号结束，解释器只有遇到分号时才会进行指令解释，因此，交互环境中的单条命令是可以换行的。

和 SQL 类似，HiveQL 一般是大小写不敏感的(除了字符串比较以外)。和 Linux 的 Shell 命令行一样，制表符（Tab）会自动补全 Hive 的关键字和函数以提高开发人员的操作效率。另外，也可以以非交互式模式运行 Hive 外壳环境。使用-f 选项可以运行指定文件中的命令。如运行脚本文件 script.q:

```
hive  -f script.q
```

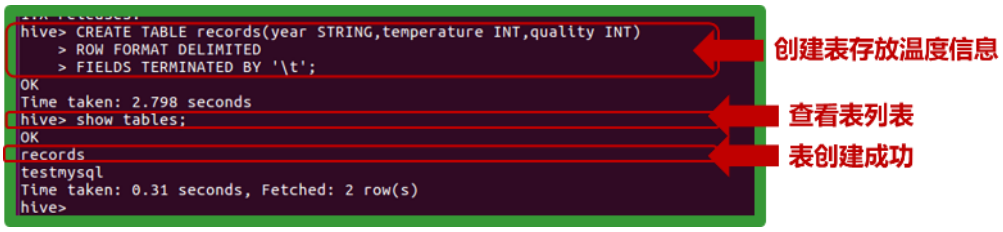
对于较短的脚本而言，还可用-e 选项在行内嵌入命令。此时不需要表示结束的分号:

```
hive  -e  ‘select * from test’
```

无论是在交互式还是非交互式模式下，Hive 都会把操作运行时的信息打印输出到标准错误输出，例如运行一个查询所花的时间。可以在启动程序的时候使用-S 选项强制不显示这些消息，只输出查询结果。

和 RDBMS 一样，Hive 把数据组织成表。接下来展示使用 CREATE TABLE 语句为数据新建一个表，并显式规定每一行数据的字段与字段之间分隔符（这在 SQL 语句中是不存在的操作）：

```
CREATE TABLE records(year STRING , temperature INT , quality INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

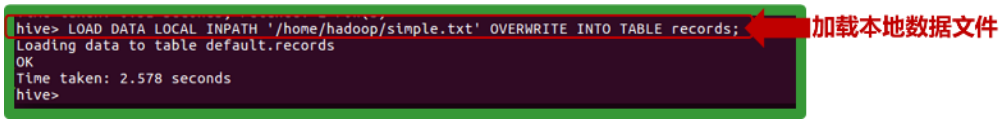


第一行声明一个 records 表，包含三列：year,temperature 和 quality。还必须指明每一列的数据类型，在这里，年为字符串类型，另外两列为整数型。

但是接下来的 ROW FORMAT 子句是 HiveQL 所特有的。这个子句所声明的是数据文件的每一行是由制表符分隔的文本。Hive 按照这一格式读取数据：每行三个字段，分别对应于表中的三列，字段间以制表符分隔，每行以换行符分隔。

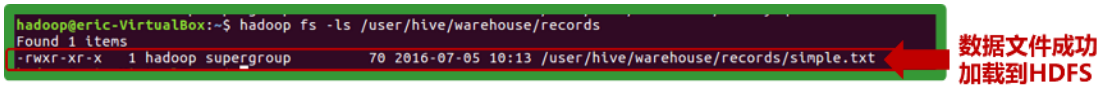
可以使用 LOAD 操作将本地的数据文件加载到 Hive 中：

```
LOAD DATA LOCAL INPATH 本地数据文件路径 OVERWRITE INTO TABLE 表名;
```



这一命令告诉 Hive 把指定的本地文件放入其仓库目录中。这只是一个简单的文件系统操作。这个操作并不解析文件或把它存储为内部数据库格式，因为 Hive 并不强制使用任何特定文件格式。文件以原样逐字存储，Hive 并不会对文件进行任何的修改。

数据文件加载成功后，可以查看该表在 HDFS 上对应的数据仓库位置，可以发现文件已经被上传到该路径：



在这个示例中，现在只有一个文件 simple.txt，但是在一般情况下，可以有多个文件，而且 Hive 会在查询表的时候读入所有这些文件。

LOAD DATA 语句中的 OVERWRITE 关键字告诉 Hive 删除表对应目录中已有的所有文件。如果省去这一关键字，Hive 就简单地把新的文件加入目录（除非目录下正好有同名的文件，此时将替换掉原有的同名文件）。

现在可以针对数据文件进行一次简单的查询操作（类似 SQL），例如如下的查询示例：

```
select year,max(temperature) from records where quality=1 or quality=2 group by year;
```

这个查询没什么特别的：它是一个带 Group By 子句的 select 语句。这个查询根据年份对行进行分组，然后使用 max() 聚集函数在每个年份组中找到址高气温。

Hive 的优势在于把这个查询转化为一个 MapReduce 作业并为我们执行这个

作业，然后把结果打印输出到控制台。

可以看到，示例通过一个和 SQL 语言保持高度一致的简单查询语句就实现了需要编写大量代码的 MapReduce 任务才能实现的统计每一年出现的最高气温的功能。虽然 Hive 和其他数据库有一些细微的差别，例如 Hive 支持的 SQL 的结构以及可以查询中数据的格式等，但能够在原始数据上执行 SQL 查询，才彰显出 Hive 的强大功能。

和 Hadoop 类似，Hive 使用 XML 配置文件进行设置。配置文件为 hive-site.xml，它在 Hive 的 conf 目录下，这在之前的安装配置过程中已经对其进行了修改，确定了 Metastore 的连接方式。

通过这个文件，可以设置每次运行 Hive 时希望 Hive 使用的选项。该目录下还包括 hive-defaultult.xml（其中记录 Hive 使用的选项及其默认值）。

传递--config 选项参数给 hive 命令，可以通过这种方式重新定义 Hive 查找 hive-sile.xml 文件：

```
hive --config /home/hadoop/devhive-conf
```

注意，这个选项指定的是包含配置文件的目录，而不是配置文件 hive-site.xml 本身。

Hive 外壳环境只是 hive 命令提供的其中一项服务。可以在运行时使用--service 选项指明要使用哪种服务。键入 hive --service help 可以获得可用服务列表。

下面介绍一些最有用的服务：

cli

Hive 的命令行接口（外壳环境）。这是默认的服务。

hiveserver2

让 Hive 以提供 Thrift 服务的服务器形式运行，允许用不同语言编写的客户端进行访问。使用 Thrift、JDBC 和 ODBC 连接器的客户端需要运行 Hive 服务器来和 Hive 进行通信。通过设置 HIVE_PORT 环境变量来指明服务器所监听的端口号（默认为 10000）。

hwi

Hive 的 Web 接口。

Jar

与 Hadoop jar 等价。这是运行类路径中同时包含 Hadoop 和 Hive 类 Java 应用程序的简便方法。

metastore

默认情况下，metastore 和 Hive 服务运行在同一个进程里运行。使用这个服

务，可以让 metastore 作为一个单独的（远程）进程运行，通过设置 METASTORE_PORT 环境变量可以指定服务器监听的端口号。

如果以服务器方式运行 Hive(hive --service hiveserver2)，可以在应用程序中以不同机制连接到服务器。Hive Thrift 客户端简化了在多种编程语言中运行 Hive 命令。Hive 的 Thrift 绑定支持 C++、Java、PHP、Python 和 Ruby。在 Hive 发布版本的 src/service/src 子目录下可以找到对这些语言的 Thrift 绑定。

Hive 提供了（Type4)（纯 Java）的 JDBC 驱动，定义在 org.apache.hadoop.hive.jdbc.HiveDriver 类中。在以 jdbc:hive2://host:port/dbname 形式配置 JDBC URI 以后，Java 应用程序可以在指定的主机和端口连接到在另一个进程中运行的 Hive 服务器。(驱动使用 Java 的 Thrift 绑定来调用由 HiveThrift 客户端实现的接口。)

还可以通过 URI jdbc:hive2:/ / 用 JDBC 内嵌模式来连接 Hive。在这个模式下，Hive 和发出调用的应用程序在同一个 JVM 中运行。这时不需要以独立服务器方式运行 Hive,这是因为此时应用程序并不使用 Thrift 服务或 Hive 的 Thrift 客户端。

Hive 的 ODBC 驱动允许支持 ODBC 协议的应用程序连接到 Hive。和 JDBC 驱动类似，ODBC 驱动使用 Thrift 和 Hive 服务器进行通信。

https://cwiki.apache.org/confluence/display/Hive/HiveClient 提供了各类客户端的使用说明

Hive 在很多方面和传统数据库类似(例如支持 SQL 接口),但是其底层对 HDFS 和 MapReduce 的依赖意味着它的体系结构有别于传统数据库,而这些区别又影响着 Hive 所支持的特性，进而影响着 Hive 的使用。

在传统数据库里，表的模式是在数据加载时强制确定的。如果在加载时发现数据不符合模式，则拒绝加载数据。因为数据是在写入数据库时对照模式进行检查，因此这一设计有时被称为“写时模式”(schema on write)。

Hive 对数据的验证并不在加载数据时进行，而在查询时进行。这称为“读时模式”(schema on read)。

- **读时模式可以使数据加载非常迅速**

这是因为它不需要读取数据，进行“解析”(parse)，再进行序列化以数据库内部格式存入磁盘。数据加载操作仅仅是文件复制或移动。

- **写时模式有利于提升查询性能。**

因为数据库可以对列进行索引，并对数据进行压缩。但是作为权衡，此时加载数据会花更多时间。此外，在很多加载时模式未知的情况下，因为查询尚未确定，因此不能决定使用何种索引。

更新、事务和索引都是传统数据库最重要的特性。Hive 还没有考虑支持这些特性，因为 Hive 被设计为用 MapReduce 操作 HDFS 数据。在这样的环境下，“全表扫描”(full-table scan）是常态操作，而表更新则是通过把数据变换后放入

新表实现的。对于在大规模数据集上运行的数据仓库应用，这一方式很见效。

Hive 不支持更新（或删除），但支持 INSERT INTO，所以可以向现有表中增加新的行。

在 0.7.0 发布版本中，Hive 引入了索引，以在某些情况下加快查询的速度。对于 SELECT * from t where x=a 这样的查询，因为只需要扫描表文件的一小部分，因此可以利用在列 x 上的索引。目前 Hive 的索引分成两类：

- 紧凑（compact）索引
- 位图（bitmap）索引

索引的实现被设计为可插拔的（PIuggable)，所以为了其他目的而设计的索引实现会陆续出现。

紧凑索引存储每个值的 HDFS 块号，而不是存储文件内偏移量。因此存储不会占用过多的磁盘空间，且对于值被聚簇（clustered）存储于相近行的情况，索引仍然有效。

位图索引使用压缩的位集合（bitset）来高效存储具有某个特殊值的行。这种索引一般适合于具有较少取值可能(low-cardinality)的列(如性别或国别)。

Hive 的 0.7.0 发布版本还引入了表级(table-level)和分区级(partition-level）的锁。有了锁，就可以防止一个进程删除正在被另一个进程读取的表。

锁由 ZooKeeper 透明管理，因此用户不必执行获得和释放锁的操作，但仍然可以通过 SHOW LOCKS 语句获取已经获得了哪些锁的信息。

默认情况下，并未启用锁的功能。

第4节 托管表和外部表

在 Hive 中创建表时，默认情况下 Hive 负责管理数据。这意味着 Hive 把数据移入它的“仓库目录”(warehouse directory)。另一种选择是创建一个“外部表”(external table)。这会让 Hive 到仓库目录以外的位置访问数据。

这两种表的区别表现在 LOAD 和 DROP 命令的语义上。

在之前的案例中已经看出，托管表加载数据时会将数据文件上载到 HDFS 的位置。

由于加载操作就是文件系统中的文件移动或文件重命名，因此它的执行速度很快。即使是托管表，Hive 也并不检查表目录中的文件是否符合为表所声明的模式。如果有数据和模式不匹配，只有在查询时才会知道。通常要通过查询为缺失字段返回的空值 NULL 才知道存在不匹配的行。可以发出一个简单的 SELECT 语句来查询表中的若干行数据，从而检查数据是否能够被正确解析。

如果随后要丢弃某个表，可使用以下语句：

```
DROP TABLE tablename;
```

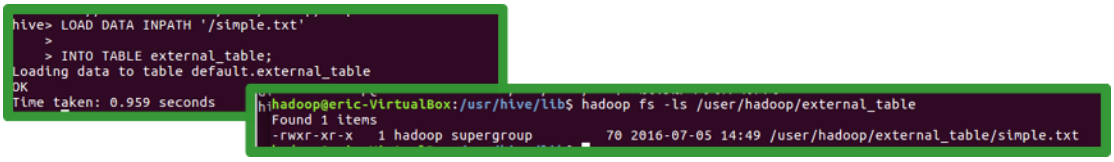
这个表，包括它的元数据和数据，会被一起删除。

重复强调：因为最初的 LOAO 是一个移动操作，而 DROP 是一个删除操作，所以数据会彻底消失。这就是 Hive 所谓的“托管数据”的含义。

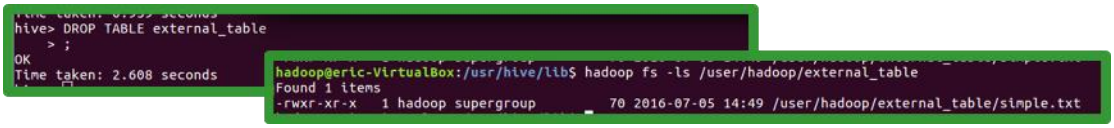
对于外部表而言，加载数据和删除表操作的结果不一样：手动来控制数据的创建和删除。外部数据的位置需要在创建表的时候指明：

```
CREATE EXTERNAL TABLE 表名（模式说明） LOCATION 数据文件路径;
```

加载数据的命令和托管表一致：



丢弃外部表时，Hive 不会碰数据，而只会剧除元数据：



应该如何选择使用哪种表呢？

在多数情况下，这两种方式没有太大的区别（当然 DROP 语义除外），因此这只是个人喜好问题。

作为一个经验法则，如果所有处理都由 Hive 完成，应该使用托管表。但如果要用 Hive 和其他工具来处理同一个数据集，应该使用外部表。普遍的用法是把存放在 HDFS（由其他进程创建）的初始数据集用作外部表进行使用，然后用 Hive 的变换功能把数据移到托管的 Hive 表。这一方法反之也成立-外部表（未必在 HDFS 中）可以用于从 Hive 导出数据供其他应用程序使用。

需要使用外部表的另一个原因是希望为同一个数据集关联不同的模式。

第5节 HiveQL 特性入门

Hive 的 SQL“方言”，称为 HiveQL，并不完全支持 SQL-92 标准，因为 SQL-

92 兼容本来就不是 Hive 项目的目标。作为一个开源项目，开发者不断增加新的功能，以满足用户的需要，从而使得 Hive 的 SQL 支持越来越丰富。

此外，Hive 还有一些 SQL-92 所没有的扩展。这些扩展受到其他数据库系统（特别是 MySQL）语法的启发。事实上，HiveQL 可以勉强看作是对 MySQL 的 SQL 方言的模仿。

下表描述了 SQL 标准与 HiveQL 的主要区别：

特性	SQL	HiveQL
更新	UPDATE,INSERT,DELETE	INSERT
事务	支持	支持（表级和分区级）
索引	支持	支持
延迟	亚秒级	分钟级
数据类型	整数、浮点数、定点数、文本和二进制串、时间	整数、浮点数、布尔型、文本和二进制串、时间戳、数组、映射、结构
函数	数百个内置函数	几十个内置函数
多表插入	不支持	支持
Create table as select	SQL-92 中不支持，但有些数据库支持	支持
选择	SQL-92	FROM 子句中只能有一个表或视图。支持偏序的 SORT BY。可限制返回数量的 LIMIT
子查询	在任何子句中支持相关和不相关子查询	只能在 FROM 子句中，不支持相关子查询
视图	可更新	只读
扩展点	用户自定义函数 存储过程	用户自定义函数 MapReduce 脚本

在数据类型方面，Hive 提供的支持如下：

- 原子
  - 原子数据类型包括数值型、布尔型、字符串类型和时间戳类型。
- 复杂数据类型
  - 复杂数据类型包括数组、映射和结构。

下表描述了 Hive 中的基本类型：

类型	描述	文字示例
TINYINT	1 字节（8 位）有符号整数，从-128 到 127	1
SMALLINT	2 字节（16 位）有符号整数	1
INT	4 字节（32 位）有符号整数	1
BIGINT	8 字节（64 位）有符号整数	1
FLOAT	4 字节（32 位）单精度浮点数	1

DOUBLE	8 字节（64 位）双精度浮点数	1
BOOLEAN	true/false	TRUE
STRING	字符串	‘a’,“a”
BINARY	字节数组	
TIMESTAMP	精确到纳秒的时间戳	

虽然有些 Hive 的原子数据类型的命名受到 MySQL 数据类型名称（其中有些和 SQL-92 相同）的影响，但这些数据类型基本对应于 Java 中的类型。

有四种有符号整数类型：TINYINT、SMALLINT、INT 以及 BIGINT，分别等价于 Java 的 byte、short、int 和 long 原子数据类型。它们分别为 1 字节、2 字节、4 字节和 8 字节有符号整数。

Hive 的浮点数据类型 FLOAT 和 DOUBLE 对应于 Java 的 float 和 double 类型，分别为 32 位和 64 位浮点数。和有些数据库不同，它不提供控制浮点数值有效数字或小数位位数的选项。

Hive 提供 BOOLEAN 数据类型用于存储真值（true）和假值（false）。

Hive 只提供了一种存储文本的数据类型 STRING。该类型是一个变长字符串。Hive 的 STRING 类似于其他数据库的 VARCHAR，但不能声明其中最多能存储多少个字符。（理论上其中最多能存储 2GB 的字符数，但如果真要物化存储那么大的值，效率肯定很低。

BINARY 数据类型用于存储变长的二进制数据。

TIMESTAMP 数据类型存储精度为纳秒的时间戳。Hive 提供了在 Hive 时间戳、Unix 时间戳（从 UNIX 纪元开始的秒数）、字符串之间进行转换的 UDF，这样就使得常用的日期操作较为容易进行处理。但是 TIMESTAMP 中并未封装时区信息。可以使用 to_utc_timestamp 和 from_utc_timestamp 函数来进行时区转换。

下表描述 Hive 中的复合类型：

类型	描述	文字示例
ARRAY	一组有序字段，字段的类型必须相同	array（1，2）
MAP	一组无序的键/值对。键的类型必须是原子的，值可以是任何类型，同一个映射的键和值的类型必须相同	map（’a’,1,’b’,2）
STRUCT	一组命名的字段，字段的类型可以不同	struct(‘a’,1,1.0)

Hive 有三种复杂数据类型：ARRAY、MAP 和 STRUCT。ARRAY 和 MAP 与 Java 中的同名数据类型类似，而 STRUCT 是一种记录类型，它封装了一个命名的字段集合。复杂数据类型允许任意层次的嵌套。

数组、映射和结构的文字形式可以通过函数得到：array()、map()、struct()。这三个函数都是 Hive 的内置函数。

复杂数据类型声明必须使用尖括号符号指明其中数据字段的类型。如下所示

的表定义有三列，每一列对应一种复杂数据类型：

```
CREATE TABLE complex (
    col1 ARRAY<INT>,
    col2 MAP<STRING,INT>,
    col3 STRUCT<a:STRING,b:INT,c:DOUBLE>
) ;
```

如下的查询展示了每种类型的字段访问操作：

```
SELECT col1[0],col2[ ‘b’ ],col3.c FROM complex;
```

Hive 提供了普通 SQL 运算符操作，包括：

- 关系操作（例如等值判断 x=”a“，空值判断 x is NULL，模式匹配 x LIKE ’a%’）
- 算术操作（例如加法 x+1）
- 逻辑操作（例如逻辑或（OR） x OR y））

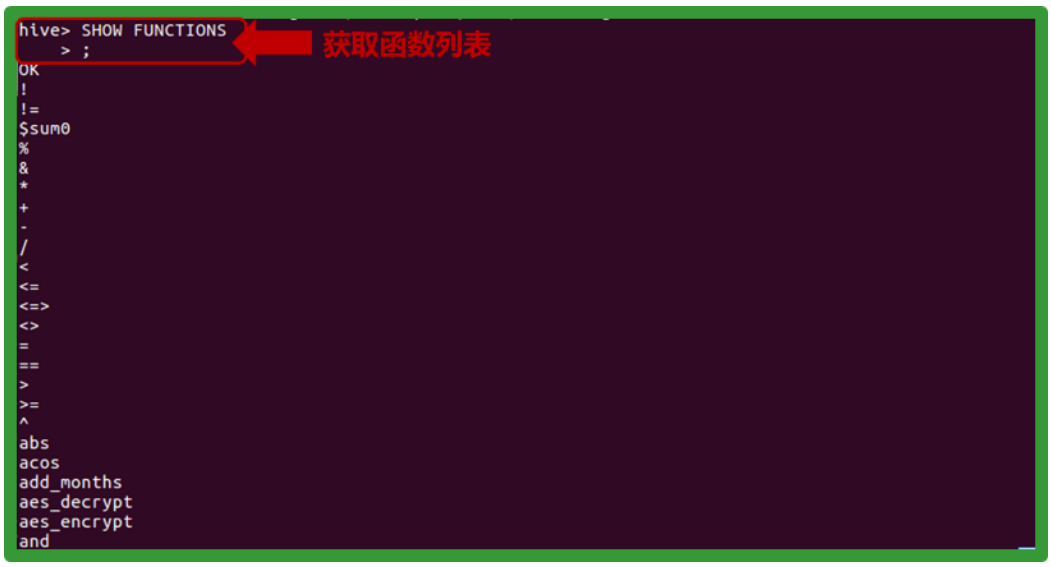
这些操作和 MySQL 的操作一样，而和 SQL-92 不同:||是逻辑或（OR），而不是字符串连接。在 MySQL 和 Hive 中，字符串连接应该用 concat 函数。

Hive 提供的内置函数分成几个大类:

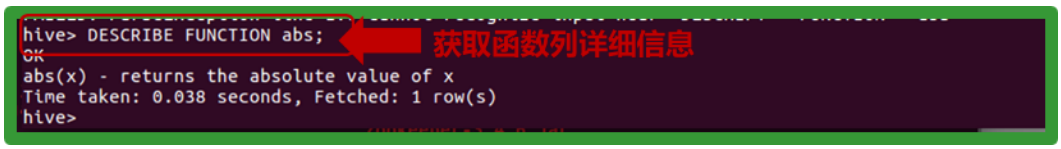
- 数学和统计函数
- 字符串函数
- 日期函数（用于操作表示日期的字符串）
- 条件函数
- 聚集函数
- 处理 XML 以使用的 xpath 函数和 JSON 的函数

可以在 Hive 外壳环境中输入 SHOW FUNCT 工 ONS 以获取函数列表：

```
SHOW FUNCTIONS;
```



要想了解某个特定函数的使用帮助，可以使用 DESCRIBE FUNCTION 命令：



Hive 中和 RDBMS 一样也提供了数据类型转换的方法。原子数据类型形成了一个 Hive 函数和操作表达式进行隐式类型转换的层次。例如，如果某个表达式要使用 INT，那么 TINYINT 会被转换为 INT。但是，Hive 不会进行反向转换，它会返回错误，除非使用 CAST 操作。

隐式类型转换规则概述如下：

- 任何整数类型都可以隐式地转换为一个范围更广的类型。
- 所有整数类型、FLOAT 和（可能令人惊讶的）STRING 类型都能隐式转换为 DOUBLE。
- TINYINT、SMALLINT 和 INT 都可以转换为 FLOAT。
- BOOLEAN 类型不能转换为其他任何数据类型
- TIMESTAMP 可以被隐式转换为 STRING。

可以使用 CAST 操作显式进行数据类型转换。例如， CAST（‘1’ AS INT）把字符串’1”转换成整数值 1。如果强制类型转换失败，例如执行 CAST（‘X’ AS INT)，表达式就会返回空值 NULL。

第6节 查询优化-分区和桶

Hive 把表组织成“分区”(partition)。这是一种根据“分区列”(partition column，如日期）的值对表进行粗略划分的机制。使用分区可以加快数据分片的查询速度。由于 Hive 使用的是读时模式，查询时无论是否带有附加的查询条件，

Hive 都需要对数据仓库中的数据文件进行全扫描，而对表进行分区则表示在加载数据时就会根据分区条件将对应的数据放置到不同的子文件夹中，此时如果利用分区条件作为查询条件使用，那么进行全扫描的数据文件数量将会大大缩小，从而优化查询的效率。

表或分区可以进一步分为“桶”(bucket)。它会为数据提供额外的结构以获得更高效的查询处理。例如，通过根据用户 ID 来划分捅，我们可以在所有用户集合的随机样本上快速计算基于用户的查询。

以分区的常用情况为例：

考虑日志文件，其中每条记录包含一个时间戳。如果根据日期来对它进行分区，那么同一天的记录就会被存放在同一个分区中。

这样做的优点是：对于限制到某个或某些特定日期的查询，它们的处理可以变得非常高效。因为它们只需要扫描查询范围内分区中的文件。

使用分区并不会影响大范围查询的执行，仍然可以查询跨多个分区的整个数据集。

一个表可以以多个维度来进行分区。例如，在根据日期对日志进行分区以外，我们可能还要进一步根据国家对每个分区进行子分区（sub partition)，以加速根据地理位置进行的查询。

分区是在创建表的时候用 PARTITION BY 子句定义的。该子句需要定义列的列表。例如，对前面提到的假想的日志文件，我们可能要把表记录定义为由时间戳和日志行构成。

创建分区表，利用日志日期和国家分区：

```
CREATE TABLE logs (ts BIGINT,line STRING) PARTITIONED BY (dt STRING,country STRING);
```

将文件加载到表对应的分区中（需要指定分区条件)：

```
LOAD DATA LOCAL INPATH '/home/hadoop/simple.txt' INTO TABLE logs PARTITION (dt='2016-7-4',contry='GB');
```

```
hive> LOAD DATA LOCAL INPATH '/home/hadoop/simple.txt' INTO TABLE logs PARTITION (dt='2016-7-4',c
ontry='GB');
Loading data to table default.logs partition (dt=2016-7-4, contry=GB)
OK
Time taken: 1.84 seconds
```

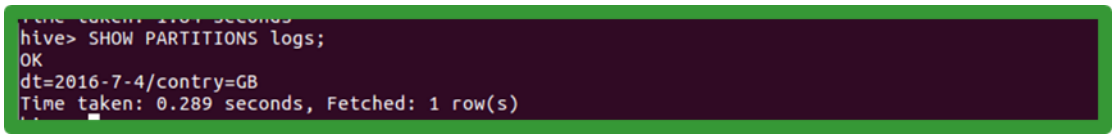
数据加载后的结果（文件夹结构)：

```
hadoop@eric-VirtualBox:/usr/hive/lib$ hadoop fs -ls /user/hive/warehouse/logs
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2016-07-05 15:15 /user/hive/warehouse/logs/dt=2016-7-4
hadoop@eric-VirtualBox:/usr/hive/lib$ hadoop fs -ls /user/hive/warehouse/logs/dt=2016-7-4
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2016-07-05 15:15 /user/hive/warehouse/logs/dt=2016-7-4/contry=GB
```

分区实质上是文件夹结构

可以用 SHOW PARTITIONS 命令让 Hive 告诉我们表中有哪些分区：

```
SHOW PARTITIONS 表名;
```



PARTITIONED BY 子句中的列定义是表中正式的列，称为“分区列”(partition column)，但是，数据文件并不包含这些列的值，因为它们源于目录名。

可以在 SELECT 语句中以通常的方式使用分区列。Hive 会对输入进行修剪，从而只扫描相关的分区，例如：

```
SELECT * FROM logs WHERE contry = 'GB';
```

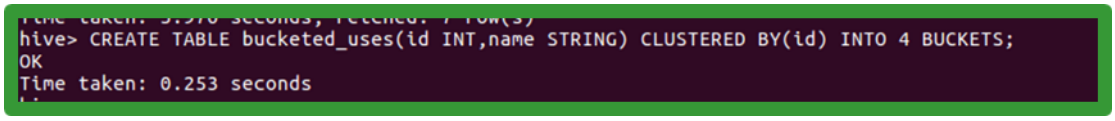
把表（或分区）组织成桶（bucket）有两个理由。

第一个理由是获得更高的查询处理效率。桶为表加上了额外的结构。Hive 在处理有些查询时能够利用这个结构。具体而言，连接两个在（包含连接列的）相同列下划分了桶的表，可以使用 map 端连接（map-side join）高效地实现。

第二个理由是使“取样”(sampling)更高效。在处理大规模数据集时，在开发和修改查询的阶段，如果能在数据集的一小部分数据上试运行查询，会带来很多方便。

使用 CLUSTERED BY 子句来指定划分桶所用的列和要划分的桶的个数：

```
CREATE TABLE bucketed_uses(id INT,name STRING) CLUSTERED BY(id) INTO 4 BUCKETS;
```

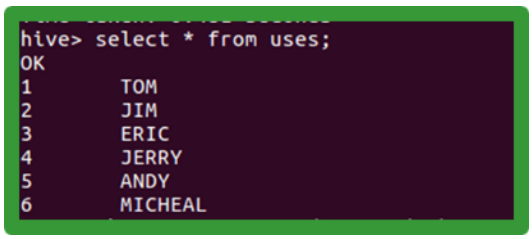


上例中使用用户 ID 来确定如何划分桶（Hive 对值进行哈希计算将结果除以桶的个数取余数。这样，任何一桶里都会有一个随机的用户集合。）

桶中的数据可以根据一个或多个列另外进行排序。以下语法声明一个表使其使用排序桶：

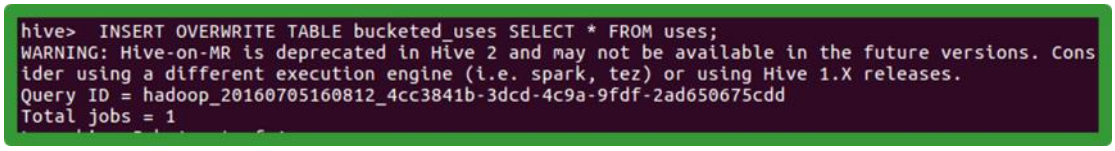
```
CREATE TABLE bucketed_uses(id INT,name STRING) CLUSTERED BY(id)
SORTED BY(id ASC) INTO 4 BUCKETS;
```

创建分桶表后，即可将数据写入分桶表中获取更好的数据查询和取样优化，例如可以将下表的数据写入新建的分桶表：



执行插入操作：

```
INSERT OVERWRITE TABLE bucketed_uses SELECT * FROM uses;
```

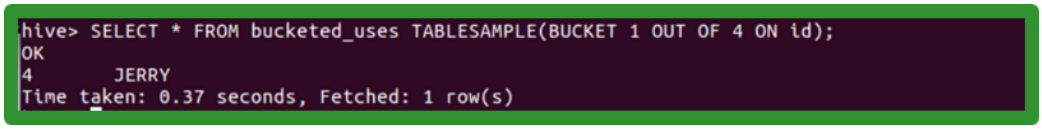


插入操作执行成功后可以查看 HDFS 上生成的分桶文件：

第一个桶里包括用户 4，因为一个 INT 的哈希值就是这个整数本身，在这里即除以桶数（4）以后的余数：

用 TABLESAMPLE 子句对表进行操作，可以获得数据采样的结果。这个子句会将查询限定在表的一部分桶内，而不是使用整个表：

```
SELECT * FROM bucketed_uses TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
```

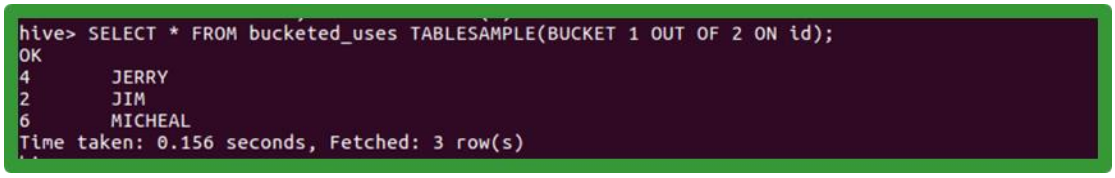


桶的个数从 1 开始计数。因此，前面的查询从 4 个桶的第一个中获取所有的

用户。对于一个大规模的、均匀分布的数据集，这会返回表中约 1/4 的数据行。

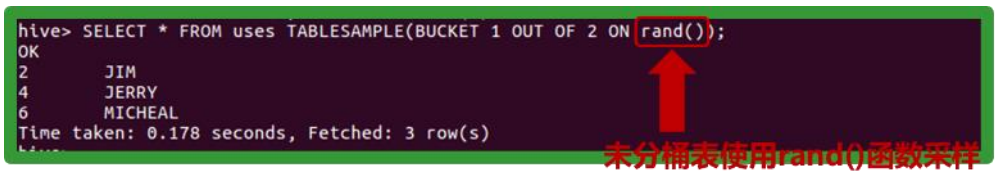
也可以用其他比例对若干个桶进行取样（因为取样并不是一个精确的操作，因此这个比例不一定是桶数的整数倍）。例如，下面的查询返回 1/2 的捅：

```
SELECT * FROM bucketed_uses TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
```



前例查询只需要读取和 TABLESAMPLE 子句匹配的桶，所以取样分桶表是非常高效的操作。如果使用 rand() 函数对没有划分成桶的表进行取样，即使只需要读取很小一部分样本，也要扫描整个输入数据集：

```
SELECT * FROM uses TABLESAMPLE(BUCKET 1 OUT OF 2 ON rand());
```



第7节 Hive 的存储格式

Hive 从两个维度对表的存储进行管理：

- **行格式(Row Format)**

行格式指行和一行中的字段如何存储。按照 Hive 的术语，行格式的定义由 SerDe 定义。SerDe 是“序列化和反序列化工具”（Serializer-Deserializer）的合成词。

- **文件格式(File Format)**

文件格式指一行中字段容器的格式。最简单的格式是纯文本文件，但是也可以使用面向行的和面向列的二进制格式。

如果在创建表时没有用 ROW FORMAT 或 STORED AS 子句，那么 Hive 所使用的默认格式是分隔的文本，每行存储一个数据行（row）。

默认的行内分隔符不是制表符，而是 ASCll 控制码集合中的 Control-A（它的 ASCll 码为 1）。

选择 Control-A（在文档中有时记作^A）作为分隔符是因为和制表符相比，它出现在字段文本中的可能性比较小。在 Hive 中无法对分隔符进行转义，因此，

挑选一个不会在数据字段中用到的字符作为分隔符非常重要。

“集合”(collection）元素的默认分隔符为字符 Control-B。它用于分隔 ARRAY 或 STRUCT 或 MAP 的键 / 值对中的元素。默认的映射键（Map Key）分隔符为字符 Control-C。它用于分隔 MAP 的键和值。

表中各行之间用换行符分隔。

最基础的创建表的语句：

```
CREATE TABLE …;
```

等价于下面显式说明的语句：

```
CREATE TABLE …
    ROW FORMAT DELIMITED FIELDS TERMINATED BY  ‘\001’
    COLLECTION ITEMS TERMINATED BY  ‘\002’
    MAP KEYS TERMINATED BY  ‘\003’
    LINES TERMINATED BY  ‘\n’
STORED AS TEXTFILE;
```

其中 STORED AS TEXTFILE 表示该表使用的数据文件以文本文件的方式存在。

Hive 在内部使用一个名为 LazySimpleSerDe 的 SerDe 来处理这种分隔格式

使用前缀“lazy”的原因是这个 SerDe 对字段的反序列化是延迟处理的-只有在访问字段时才进行反序列化。但是，由于文本以冗长的形式进行存放，所以这种存储格式并不紧凑。比如，一个布尔值事实上是以文本字符串 true 或 false 的形式存放的。

二进制 SerDe 不应该和默认的 TEXTFILE 格式(或显式的 STORED AS TEXTFILE 子句）一起使用。二进制数据行中几乎总是包含换行符，这会导致 Hive 把行截断，从而在反序列化时失败。

第8节 Hive 查询操作进阶

- 在 Hive 中，向一个表中导入数据的方法有以下几种：
- 使用 LOAD DATA 操作，通过把文件复制或移到表的目录中，从而把数据导入 Hive 的表（或分区）。
- 用 INSERT 语句把数据从一个 Hive 表填充到另一个。
- 在新建表的时候使用 CTAS 结构，CTAS 是 CREATE TABLE … AS SELECT 的缩写。

使用 LOAD 操作为表加载数据在前面的章节已经看到过，接下来展示 INSERT 语句的使用。以下是 INSERT 语句的一个示例:

```
INSERT OVERWRITE TABLE target SELECT col1,col2 FROM source;
```

对于分区表，可以使用 PARTITION 子句来指明数据要插入哪个分区：

```
INSERT OVERWRITE TABLE target PARTITION(dt=‘2016-7-4’) SELECT col1,col2 FROM source;
```

OVERWRITE 关键字在以上两种情况下都是强制的。这意味着目标表（对于前面的第一个例子）或‘2016-7-4’分区（对于第二个例子）中的内容会被 SELECT 语句的结果替换掉。

如果要向已经填充了内容的非分区表或分区添加记录，那么可以使用 INSERT INTO TABLE。

在针对分区表进行操作时，还可以在 SELECT 语句中通过使用分区值来动态指明分区：

```
INSERT OVERWRITE TABLE target PARTITION(dt) SELECT col1,col2,dt FROM source;
```

这种方法称为“动态分区插入”(dynamic-partition insert)。需要注意的是，这一特性默认是关闭的，所以在使用前需要先在 Hive 的配置文件中把属性 hive.exec.dynamic.partition 设为 true。

和其他数据库不同，Hive（现在）不支持在 INSERT 语句中直接以文字形式给出一组记录的形式。也就是说. Hive 不允许 INSERT INTO … VALUES…形式的语句。

在 HiveQL 中，可以把 INSERT 语句倒过来，把 FROM 子句放在最前而查询的效果是一样的：

```
FROM source INSERT OVERWRITE TABLE target SELECT col1,col2;
```

可以在同一个查询中使用多个 INSERT 子句。这样的语法会让查询的含义更清楚。这种“多表插入”方法比使用多个单独的 INSERT 语句效率更高，因为只需要扫描一遍源表就可以生成多个不相交的输出。

下面的例子根据气象数据集来计算多种不同的统计数据：

```
FROM records

INSERT OVERWRITE TABLE stations_by_year SELECT year,COUNT(DISTINCT station) GROUP BY year

INSERT OVERWRITE TABLE records_by_year SELECT year,COUNT(1) GROUP BY
```

```
year

INSERT OVERWRITE TABLE good_records_by_year SELECT year,COUNT(1)
WHERE temperature!=32 AND quality=1 OR quality=2 GROUP BY year
```

这里只有一个源表(records2)，但有三个表用于存放针对这个源表的三个不同查询所产生的结果。

把 Hive 查询的输出结果存放到一个新的表往往非常方便，这是因为输出结果太多，不适宜于显示在控制台上或基于输出结果还有其他后续处理。新表的列的定义是从SELECT子句所检索的列导出的。在下而的查询中，target表有两列，分别名为col1和col2，它们的数据类型和源表中对应的列相同：

```
CREATE TABLE target AS SELECT col1,col2 FROM source;
```

CTAS操作是原子的，因此如果SELECT查询由于某种原因失败，新表就不会被创建。

由于Hive使用“读时模式”灵活地支持在创建表以后对表定义的修改。但需要警惕，在很多情况下，要确保修改数据以符合新的结构。可以使用ALTER TABLE语句来重命名表：

```
ALTER TABLE source RENAME TO target;
```

在更新表的元数据以外，ALTER TABLE语句还把表目录移到新名称所对应的目录下。在这个示例中，/user/hive/warehouse/source被重命名为/user/hive/warehouse/target。对于外部表，这个操作只更新元数据，而不会移动目录。

Hive允许修改列的定义，添加新的列，甚至用一组新的列替换表内已有的列。例如，考虑添加一个新列：

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

新的列col3添加在已有（非分区）列的后面。数据文件并没有被更新，因此原来的查询会为col3的所有值返回空值null（当然，除非文件中原来就已经有额外的字段）。

因为 Hive 并不允许更新已有的记录，所以需要使用其他机制来更新底层的文件。为此，更常用的做法是创建一个定义了新列的新表，然后使用SELECT语句把数据填充进去。如果我们设想原来的数据类型可以用新的数据类型来进行解释，那么修改一个表的元数据（如列名或数据类型）就变得更为直观。

要想更进一步了解如何修改表的结构，包括添加或丢弃分区、修改和替换列，修改表和SerDe的属性，可访问Hive的英文维基页面，网址为https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL。

DROP TABLE语句用于删除表的数据和元数据。如果是外部表，就只删除元数

据－数据不会受到影响。如果要删除表内的所有数据，但要保留表的定义（如MySQL 的 DELETE 或 TRUNCATE),删除数据文件即可。例如：

```
dfs  - rmr  /user/hive / warehouse/my_table;
```

Hive 把缺少文件（或根本没有表对应的目录）的表认为是空表。

另一种达到类似目的的方法是使用 LIKE 关键字创建一个与第一个表模式相同的新表：

```
CREATE TABLE new__table LIKE existing_table;
```

在 Hive 中可以使用标准的 ORDER BY 子句对数据进行排序。但这里有一个潜在的不利因素。ORDER BY 能够预期产生完全排序的结果，但它是通过只用一个 reducer 来做到这一点的。所以对于大规模的数据集，它的效率非常低。

在很多情况下，并不需要结果是全局排序的。此时，可以换用 Hive 的非标准的扩展 SORT BY。SORT BY 为每个 reducer 产生一个排序文件。

在有些情况下，需要控制某个特定行应该到哪个 reducer，其目的通常是为了进行后续的聚集操作。这就是 Hive 的 DlSTRIBUTE BY 子句所做的事情。

下面的例子根据年份和气温对气象数据集进行排序，以确保所有具有相同年份的行最终都在同一个 reducer 分区中：

```
FROM records2 SELECT year,temperature DISTRIBUTE BY year SORT BY year ASC,temperature DESC
```

如果 SORT BY 和 DISTRIBUTE BY 中所用的列相同，可以缩写为 CLUSTER BY 以便同时指定两者所用的列。

和直接使用 MapReduce 相比，使用 Hive 的一个好处在于 Hive 简化了常用操作。对比在 MapReduce 中实现“连接” 要做的事情,在 Hive 中进行连接操作就能充分体现这个好处。

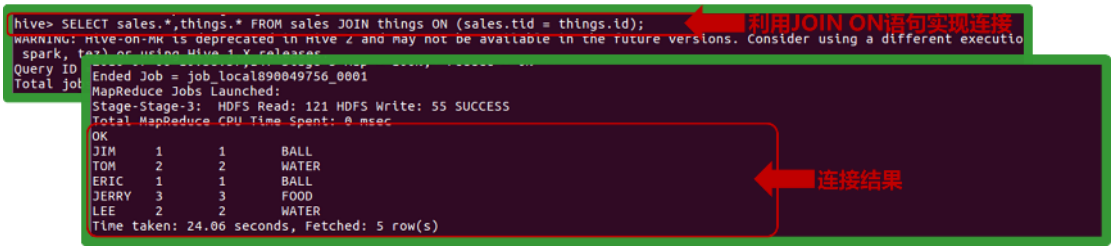
内连接是最简单的一种连接。输入表之间的每次匹配都会在输出表里生成一行。首先创建两个演示用的小表并加载相应的测试数据：

- sales 列出了人名及其所购商品的 ID
- Things 列出商品的 ID 和名称

```
hive> SELECT * FROM sales;
OK
JIM     1
TOM     2
ERIC    1
JERRY   3
LEE     2
Time taken: 0.194 seconds, Fetched: 5 row(s)
hive> SELECT * FROM things;
OK
1       BALL
2       WATER
3       FOOD
Time taken: 0.176 seconds, Fetched: 3 row(s)
hive>
```

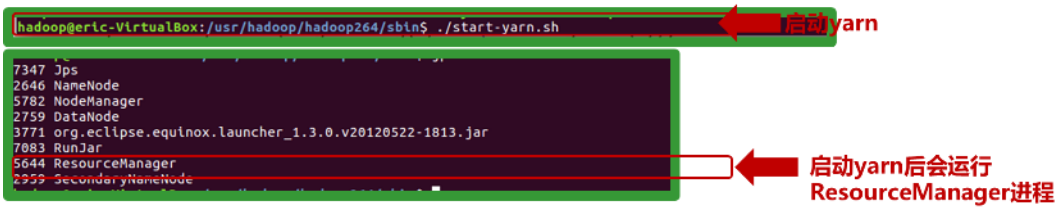
可以使用下面的查询语句完成内连接（类似 SQL）：

```
SELECT sales.*,things.* FROM sales JOIN things ON (sales.tid = things.id);
```



需要注意的是，有些数据库，例如 MySQL 和 Oracle,允许在 SELECT 语句的 FROM 子句中列出要连接的表，而在 WHERE 子句中指定连接条件，但是 Hive 中并不支持这种语法。

如果在 Hadoop 中启用了 yarn 作为任务资源调度器，则需要将 yarn 启动，启动脚本位于：$Hadoop_HOME\sbin\start-yarn.sh,否则执行连接查询会出现卡在：hive Number of reduce tasks is set to 0 since there ‘s no reduce operator 日志输出的情况：

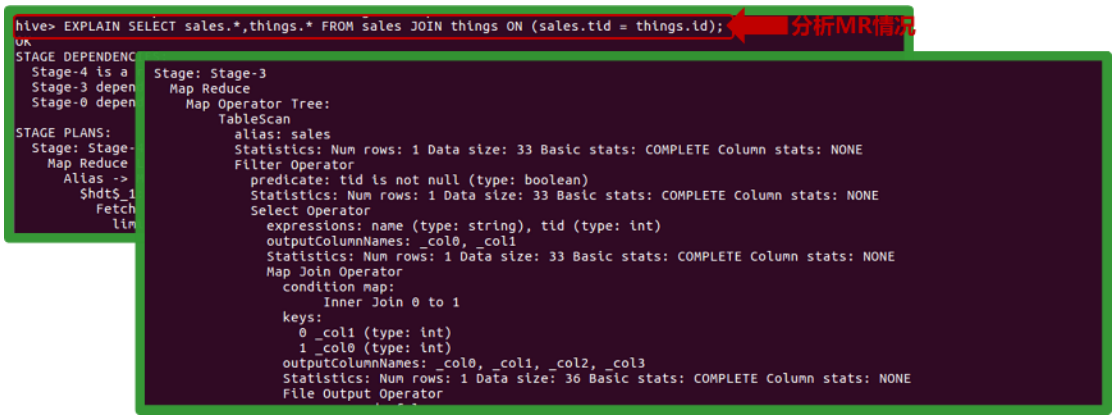


在 Hive 中，可以在连接谓词中使用 AND 关键字分隔的一系列表达式来连接多个列。

还可以在查询中使用多个 JOIN…ON…子句来连接多个表。Hive 会智能地以最少 MapReduce 作业数来执行连接。单个的连接用一个 MapReduce 作业实现。但是，如果多个连接的连接条件中使用了相同的列，那么平均每个连接可以用少于一个 MapReduce 作业来实现。

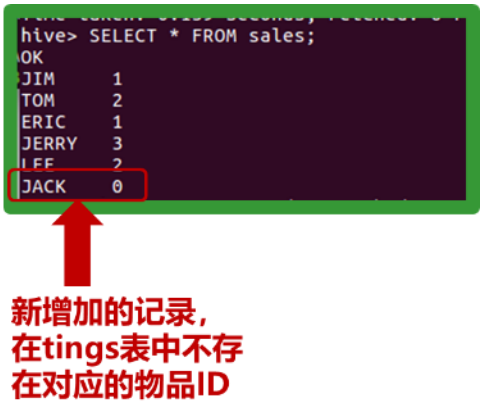
可以在查询前使用 EXPLAIN 关键字来查看 Hive 将为某个查询使用多少个 MapReduce 作业：

```
EXPLAIN SELECT sales.*,things.* FROM sales JOIN things ON (sales.tid = things.id);
```



外连接可以让你找到连接表中不能匹配的数据行。

修改 sales 表，添加一个不存在的物品 ID，把连接的类型改为 LEFT OUTER JOIN,即使左侧表（sales）中的有些行无法与所要连接的表（things）中的任何数据行对应，查询还是会返回这个表中的每一个数据行：

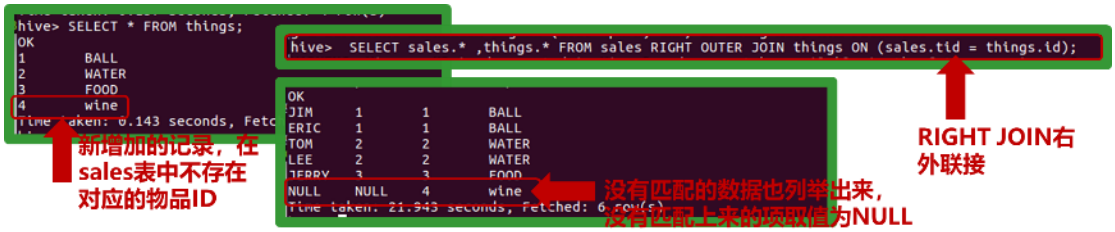



```
SELECT sales.*,things.* FROM sales LEFT OUTER JOIN things ON (sales.tid = things.id);
```

Hive 也支持“右外连接” ，交换两个表的角色：

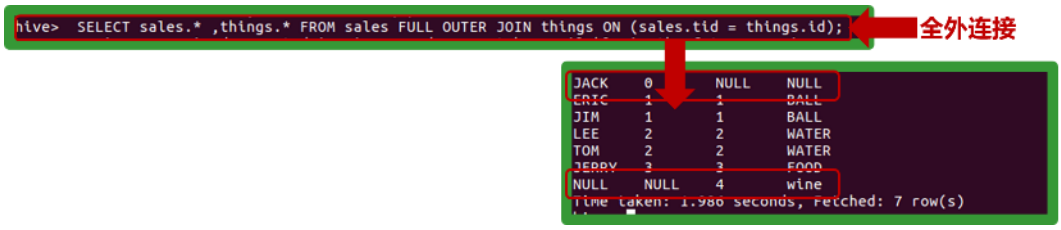
```
SELECT sales.*,things.* FROM sales RIGHT OUTER JOIN things ON
```

```
(sales.tid = things.id);
```



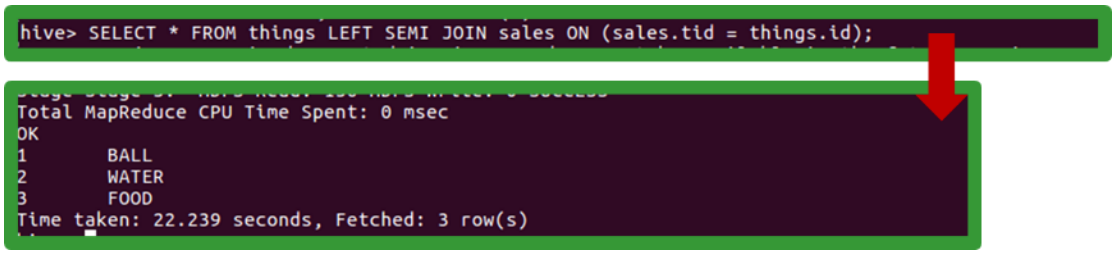
另外还有全外连接，两个表中无法匹配条件的数据都将被显示：

```
SELECT sales.*,things.* FROM sales FULL OUTER JOIN things ON (sales.tid = things.id);
```



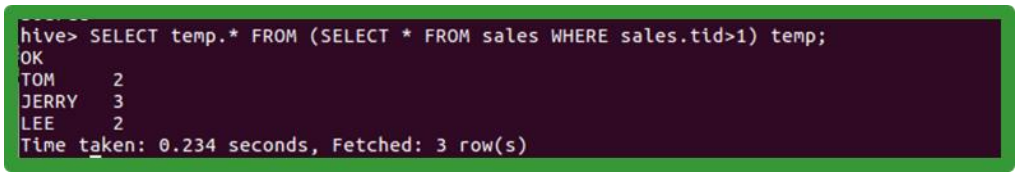
最后，Hive 可以使用 LEFT SEMI JOIN 来达到 IN 子查询相同的效果。如下例能够查找 things 表中在 sales 表中出现过的所有商品：

```
SELECT * FROM things LEFT SEMI JOIN sales ON (sales.tid = things.id);
```



子查询是内嵌在另一个 SQL 语句中的 SELECT 语句。Hive 对子查询的支持很有限，它只允许子查询出现在 SELECT 语句的 FROM 子句中。

```
SELECT temp.* FROM (SELECT * FROM sales WHERE sales.tid>1) temp;
```



和 RDBMS 一样，Hive 的视图是一种用 SELECT 语句定义的“虚表”。

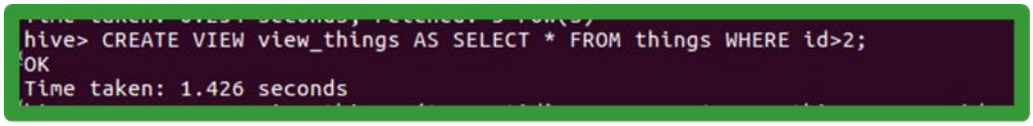
视图可以用来以一种不同于磁盘实际存储的形式把数据呈现给用户。现有表中的数据常常需要以一种特殊的方式进行简化和聚集以便于后期处理。视图也可

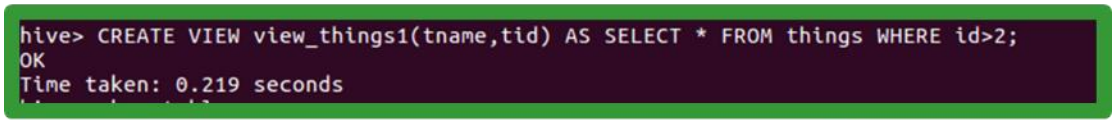
以用来限制用户，使其只能访问被授权可以看到的表的子集。

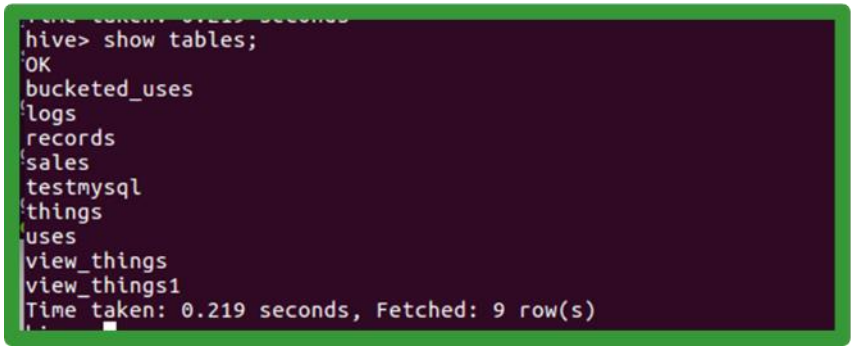
在 Hive 中，创建视图时并不把视图“物化” 存储到磁盘上。相反，视图的 SELECT 语句只是在执行引用视图的语句时才执行。如果一个视图要对“基表”进行大规模的变换，或视图的查询会频繁执行，可以选择新建一个表，并把视图的内容存储到新表中，以此来进行优化。

创建视图可以直接使用基表原生的字段名，也可以自定义字段名（适用于字段由聚集函数生成的环境，如果不提供字段名，Hive 会生成无意义的字段名）：

```
CREATE VIEW view_things AS SELECT * FROM things WHERE id>2;

CREATE VIEW view_things1(tname,tid) AS SELECT * FROM things WHERE id>2;
```







创建视图时并不执行查询，查询只是存储在 metastore 中。可以使用 DESCRIBE EXTENDED view_name 命令来查看某个视图的详细信息，包括用于定义它的那个查询：

```
DESCRIBE EXTENDED 视图名;
```

在业务系统中的查询有时无法轻松（或根本不能）使用 Hive 提供的内置函数来表示，通过写“用户定义函数”(user-defined function,UDF),Hive 可以方便地插入用户写的处理代码并在查询中调用它们。Hive 用户自定义函数必须使用 Java 编写。

Hive 中有三种 UDF:

- **普通 UDF**

UDF 操作作用于单个数据行，且产生一个数据行作为输出。大多数函数（例如数学函数和字符串函数）都属于这一类。

- **用户定义聚集函数**

UDAF 接受多个输入数据行，并产生一个输出数据行。像 COUNT 和 MAX 这样的函数都是聚集函数。

- **用户定义表生成函数**

UDTF 为输出操作作用于单个数据行，且产生多个数据行-一个表作为输出。

这三种不同的用户自定义函数所接受的输入和产生的输出的数据行的数量是不同的。

一个 UDF 必须满足下面两个条件：

- 一个 UDF 必须是 org.apache.hadoop.hive.ql.exec.UDF 的子类；
- UDF 必须至少实现了 evaluate() 方法。

evaluate() 方法不是由接口定义的，因为它可接受的参数的个数、它们的数据类型及其返回值的数据类型都是不确定的。Hive 会检查 UDF，看能否找到和函数调用相匹配的 evaluate() 方法。

一个在表字段数据后面串联字符串的 UDF 的参考实现如下：

```
public class StringAppend extends UDF {

    public String evaluate(String str) {
        return str == null ? "chinasofti" : str + "chinasofti";
    }

    public String evaluate(String str, String addChars) {
        return str == null ? addChars : str + addChars;
    }


}
```

和在 Pig 中使用自定函数类似，要想在 Hive 中使用以上编写的自定义函数，也需要将其编译后打包并加载到 Hive 环境中。

首先将 Java 代码导出为 JAR 文件，然后使用如下命令将其加入到 Hive 环境中：

```
ADD JAR 自定义函数类所在 jar 包路径;
```

```
hive> ADD JAR /usr/hive/addstring.jar
    > ;
Added [/usr/hive/addstring.jar] to class path
Added resources: [/usr/hive/addstring.jar]
```

接下来创建 UDF 声明：

```
CREATE TEMPORARY FUNCTION addstring AS '自定义函数完整类名';
```

```
hive> CREATE TEMPORARY FUNCTION addstring AS 'com.chinasofti.hadooptest.hive.udf.StringAppend';
OK
Time taken: 1.553 seconds
```

执行成功后，就可以使用 UDF 了：

```
hive> SELECT addstring(sales.name),sales.tid FROM sales;
OK
JIMchinasofti   1
TOMchinasofti   2
ERICchinasofti  1
```

UDF 是非大小写敏感的。

要想在开始时调用 ADD JAR，可以在 Hive 启动时指定查找附加 JAR 文件的路径，这个路径会被加入 Hive 的类路径（也包括 MapReduce 的类路径）。这种技术对于每次运行 Hive 时自动添加 UDF 库是很有用的。

有两种指明路径的办法，第一种方式是在 hive 命令后传递--auxpath 选项：

```
hive --auxpath /usr/hive/addstring.jar
```

另一种方法是在运行 Hive 前设置 HIVE_AUX__JARS_PATH 环境变量。附加路径可以是一个用逗号分隔的 JAR 文件路径列表或包含 JAR 文件的目录。

聚集函数比普通的 UDF 难写。因为值是在块内进行聚集的（这些块可能分布在很多 map 或 reduce 任务中），从而实现时要能够把部分的聚集值组合成最终结果。

以下是计算平均 ID 值的 UDAF 代码示例：

```
public class AvgID extends UDAF {
    public static class AvgIDIntUDAFEvaluator implements
UDAFEvaluator {
        public static class PartialResult {
            int sum, count;
```

```
        }

        private PartialResult partial;

        public void init() {
            partial = null;
        }

        public boolean iterate(int value) {
            partial = partial == null ? new PartialResult() :
partial;
            partial.sum += value;
            partial.count++;
            return true;
        }

        public PartialResult terminatePartial() {
            return partial;
        }

        public boolean merge(PartialResult other) {
            if (other == null) {
                return true;
            }
            partial = partial == null ? new PartialResult() :
partial;
            partial.sum += other.sum;
            partial.count += other.count;
            return true;
        }

        public int terminate() {
            return partial == null ? 0 : partial.sum /
partial.count;
        }
    }
}
```

一个计算函数必须实现下面 5 个方法：

- `init()`

方法 `init()` 方法负责初始化计算函数并重设它的内部状态。

- `iterate()`

每次对一个新值进行聚集计算时都会调用 iterate()方法。计算函数要根据聚集计算的结果更新其内部状态。iterate() 接受的参数和 Hive 中被调用函数的参数是对应的。

- terminatePartial()

Hive 需要部分聚集结果时会调用 terminatepartiai() 方法。这个方法必须返回一个封装了聚集计算当前状态的对象。

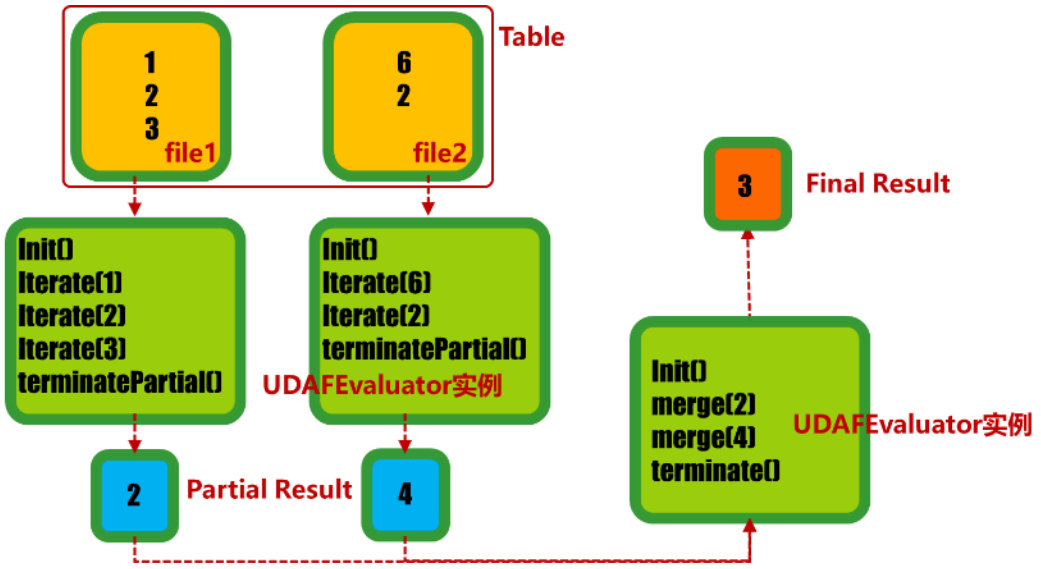
- merge()

在 Hive 决定要合并一个部分聚集值和另一个部分聚集值时会调用 merge() 方法。该方法接受一个对象作为输入。这个对象的类型必须和 terminatePartial() 方法的返回类型一致。

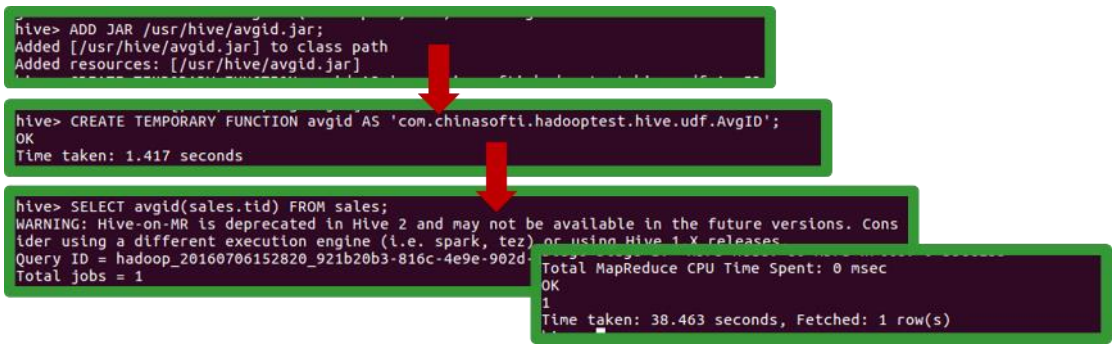
- terminate()

Hive 需要最终聚集结果时会调用方法。计算函数需要把状态作为一个值返回。

UDAF 的执行流程描述如下图所示：



UDAF 函数的执行步骤和 UDF 一致：



第9节 通过 JDBC 访问 Hive

Hive 能够通过和 SQL 语言保持高度一致的 HQL 对数据进行查询访问，这极大的降低了 Hive 作为业务系统数据仓库支撑的入门难度，因为在绝大部分的情况下，通过 Hive 的查询引擎即可完成业务需求而无须开发人员学习复杂的 MapReduce 计算框架。在开发接口方面，Hive 也提供了传统 Java 开发人员的低成本迁移方案：通过 JDBC 集成 Hive 和 Java 应用。

要使用 JDBC 连接 Hive，首先需要修改 Hadoop 配置，允许 Hive 用户通过远程访问（在 Hadoop 的 core-site.xml 添加用户控制属性）：

```
<property>
      <name>hadoop.proxyuser.hadoop.groups</name>
      <value>*</value>
      <description>Allow the superuser oozie to impersonate any members of the group group1 and group2</description>
</property>

<property>
      <name>hadoop.proxyuser.hadoop.hosts</name>
      <value>*</value>
      <description>The superuser can connect only from host1 and host2 to impersonate a user</description>
</property>
```

接下来启动 Hive 的服务模式：

```
hive --service hiveserver2
```

Hive JDBC 的必要参数如下：

- 驱动名：org.apache.hive.jdbc.HiveDriver
- 连接字符串：jdbc:hive2://主机:端口（默认 10000）/数据库名（默认数据库是 default）

编写和传统 RDBMS 连接相同的 JDBC 代码即可连接 Hive 进行数据查询操作，参考实现如下：

```
public class TestJDBCConnection {
```

```
    public static void main(String[] args) {
        try {
            Class.forName("org.apache.hive.jdbc.HiveDriver");
            Connection con = DriverManager.getConnection(
                    "jdbc:hive2://127.0.0.1:10000/default",
"username", "password");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM sales");
            while (rs.next()) {
                System.out.println(rs.getString(1) + "\t" +
rs.getInt(2));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

程序执行结果如下：

第7章 大数据查询引擎

第1节 Presto

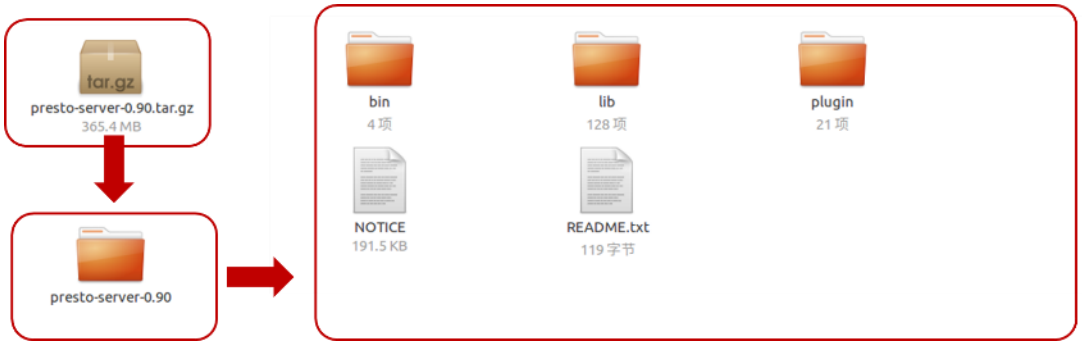
Presto 是一个开源的分布式 SQL 查询引擎，适用于交互式分析查询，数据量支持 GB 到 PB 字节。Presto 的设计和编写完全是为了解决像 Facebook 这样规模的商业数据仓库的交互式分析和处理速度的问题。

Presto 支持在线数据查询，包括 Hive，Cassandra，关系数据库以及专有数据存储。 一条 Presto 查询可以将多个数据源的数据进行合并，可以跨越整个组织进行分析。它以分析师的需求作为目标，他们期望响应时间小于 1 秒到几分钟。 Presto 终结了数据分析的两难选择，要么使用速度快的昂贵的商业方案，要么使用消耗大量硬件的慢速的“免费”方案。

Facebook 使用 Presto 进行交互式查询，用于多个内部数据存储，包括 300PB 的数据仓库。 每天有 1000 多名 Facebook 员工使用 Presto，执行查询次数超过 30000 次，扫描数据总量超过 1PB。

Presto 的安装需要两个步骤：解压发行包和修改配置文件，和 Hive 等组件不同，Presto 的不同功能的配置文件存放在不同的路径中，在对 Presto 进行配置时需要注意配置文件的名字是否正确。

如果设备上安装的是 JDK1.7，则在 Presto 官网上下载 0.90 版本的发行包，更高版本的 Presto 需要至少 1.8 版本的 JDK 支持：



完成解压后需要在 Presto 文件夹中构建其配置文件夹并在 Presto 文件夹或其他任何能够访问到的地方构建 Presto 使用的数据文件夹：

Presto 配置文件所在文件夹名为 etc，需要手工创建，并且位于 Presto 安装目录中，Presto 运行时的数据文件，可以建立在任何有权限能够访问到的地方，通过 Presto 的配置文件进行定位：

创建 etc 配置文件目录后，在其中创建以下配置文件：

- node.properties：每个节点的环境配置
- jvm.config：jvm 参数
- config.properties：配置 Presto Server 参数
- log.properties：配置日志等级
- Catalog Properties：Catalog 的配置

etc/node.properties 示例配置如下：

```
node.environment=prestoicss
node.id=icssnode1
node.data-dir=/usr/presto-server-0.90/data
```

上述配置文件中各参数的含义如下：

node.environment：

环境名称。一个集群节点中的所有节点的名称应该保持一致。

node.id：

节点唯一标识的名称。

node.data-dir：

数据和日志存放路径。

需要注意的是，Presto 对于环境和节点的名称要求比较严格，在一般编程环境中能够使用的某些特殊字符，如下划线_都不允许出现在名称字符串中，否则服务不能正常启动，因此建议环境名称和节点的标识名称都只包含英文字母和数字。

etc/jvm.config 用于配置 Presto 运行时的 Java 虚拟机环境参数，如内存大小等，示例配置如下：

```
-server
-Xmx16G
-XX:+UseConcMarkSweepGC
-XX:+ExplicitGCInvokesConcurrent
-XX:+CMSClassUnloadingEnabled
-XX:+AggressiveOpts
-XX:+HeapDumpOnOutOfMemoryError
-XX:OnOutOfMemoryError=kill -9 %p
-XX:ReservedCodeCacheSize=150M
```

etc/config.properties 包含 Presto Server 相关的配置，每一个 Presto Server 可以通时作为 coordinator 和 worker 使用。可以将他们配置在一个节点上，但是，在一个大的集群上建议分开配置以提高性能。参考配置如下：

```
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8080
task.max-memory=1GB
discovery-server.enabled=true
discovery.uri=http://127.0.0.1:8080
```

如果节点是 coordinator ，则它的最小配置如下：

```
coordinator=true
node-scheduler.include-coordinator=false
http-server.http.port=8080
task.max-memory=1GB
discovery-server.enabled=true
discovery.uri=http://server:8080
```

而 worker 的最小配置如下：

```
coordinator=false
http-server.http.port=8080
```

```
task.max-memory=1GB
discovery.uri=http://server:8080
```

config.properties 配置文件中各个参数的含义如下：

coordinator：

Presto 实例是否以 coordinator 对外提供服务。

node-scheduler.include-coordinator：

是否允许在 coordinator 上进行调度任务。

http-server.http.port：

HTTP 服务的端口。

task.max-memory=1GB：

每一个任务（对应一个节点上的一个查询计划）所能使用的最大内存。

discovery-server.enabled：

是否使用 Discovery service 发现集群中的每一个节点。

discovery.uri：

Discovery server 的 url。

etc/log.properties 可以设置某一个 java 包的日志等级，例如：

```
com.facebook.presto=INFO
```

关于 Catalog 的配置，首先需要创建 etc/catalog 目录，然后根据想使用的连接器来创建对应的配置文件，比如，在接下来的业务环境中想使用 Hive 连接器（在 Presto 中多个不同类型的连接器可以协同工作），则在 catalog 目录中创建 hive.properties，并设置 Hive 对应的连接参数：

```
#连接器名称，本例使用 Hive 连接器
connector.name=hive-hadoop2
# Hive 的 metastore 服务地址（因此使用 Presto 之前需要开启 Hive 的 Metastore 服务）
hive.metastore.uri=thrift://127.0.0.1:9083
```

为了能够正常使用 Presto 连接 Hive 的 Metastore 进行数据查询，需要按照以下几个步骤来启动对应的依赖服务：

```
start-dfs.sh
```

```
hive  -service metastore

PRESTO_HOME/bin/launcher start
```

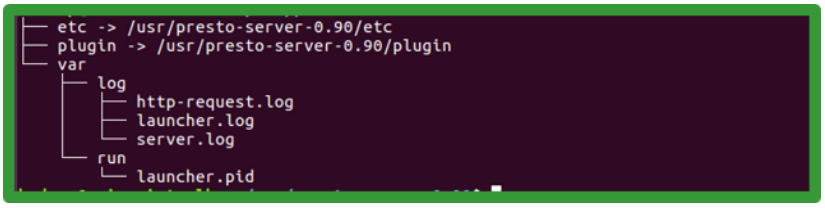
Presto 也可以前台启动，观察输出日志：

```
PRESTO_HOME/bin/launcher run
```

如果需要停止 Presto 服务，则执行脚本：

```
PRESTO_HOME/bin/launcher stop
```

Presto 服务启动后，观察之前设置的数据文件 data 目录：



在这个目录结构的/var/log 目录中可以查看日志，Presto 的日志根据记录的主体目标不同被分割成了不同的文件保存，其中包含：

launcher.log：

启动日志。

server.log：

Presto Server 输出日志。

http-request.log：

HTTP 请求日志。

启动 Presto 后可以通过 WEB 界面查看状态：

Presto 本身提供了控制台交互工具，该工具和 Hive 的交互界面一样，可以直接在控制台终端中进行数据的查询操作。不过该工具并没有和 Presto 的发行包打包在一起，而是作为了一个单独的工具提供给用户下载。在官网中下载 presto-cli-版本号-executable.jar 工具，并将其重命名为 presto-cli（也可以重命名为 presto 等任何符合 linux 命名规范的名称），然后添加执行权限，接下来以下面的命令格式启动交互程序：

```
./presto-cli(自行重命名的名字) --server 服务器地址:端口 --catalog 连接器名称 --schema 操作的 Schema 名称
```

然后可以使用类似 Hive 的操作指令和 Presto 进行交互：

Presto 同样提供了 JDBC 接口来降低 Java 开发人员的上手难度，使用 jdbc 连接 Presto，需要下载 jdbc 驱动 presto-jdbc-版本号.jar 并将其加到应用程序的 classpath 中。

Presto 支持以下几种 JDBC URL 格式：

```
jdbc:presto://host:port

jdbc:presto://host:port/catalog

jdbc:presto://host:port/catalog/schema
```

例如要连接 hive 数据库中 sales 库，示例如下：

```
jdbc:presto://cdh1:8080/hive/sales
```

第2节 Impala

Impala 是 Cloudera 公司主导开发的新型查询系统，它提供 SQL 语义，能查询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。已有的 Hive 系统虽然也提供了 SQL 语义，但由于 Hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程，难以满足查询的交互性。相比之下，Impala 的最大特点也是最大卖点就是它的快速。

Impala 不需要把中间结果写入磁盘，省掉了大量的 I/O 开销。它省掉了 MapReduce 作业启动的开销。MapReduce 启动 task 的速度很慢（默认每个心跳间隔是 3 秒钟），Impala 直接通过相应的服务进程来进行作业调度，速度快了很多。

Impala 完全抛弃了 MapReduce 这个不太适合做 SQL 查询的范式，而是像 Dremel 一样借鉴了 MPP 并行数据库的思想另起炉灶，因此可做更多的查询优化，从而省掉不必要的 shuffle、sort 等开销。它通过使用 LLVM 来统一编译运行时代码，避免了为支持通用编译而带来的不必要开销。

在系统层面，Impala 使用 C++实现，做了很多有针对性的硬件优化，例如使用 SSE 指令。Impala 还使用了支持 Data locality 的 I/O 调度机制，尽可能地将数据和计算分配在同一台机器上进行，减少了网络开销。

在性能方面，Impala 宣称在性能上比 Hive 高出 3~30 倍，甚至预言说在将来的某一天可能会超过 Hive 的使用率而成为 Hadoop 上最流行的实时计算平台，它的目的不在于替换现有的 MapReduce 工具，如 Hive，而是提供一个统一的平台用于实时查询。事实上 Impala 的运行也是依赖 Hive 的元数据。

与 Hive 类似，Impala 也可以直接与 HDFS 和 HBase 库直接交互。只不过 Hive 和其它建立在 MapReduce 上的框架适合需要长时间运行的批处理任务。例如批量提取，转化，加载（ETL）类型的 Job。而 Impala 主要用于实时查询

Impala 由以下几个组件组成：

State Store，对应进程为 statestored，它用于协调各个运行 impalad 的实例之间的信息关系，Impala 正是通过这些信息去定位查询请求所要的数据。换句话说，state store 的作用主要为跟踪各个 impalad 实例的位置和状态，让各个 impalad 实例以集群的方式运行起来

与 HDFS 的 NameNode 不一样，虽然 State Store 一般只安装一份，但一旦 State Store 挂掉了，各个 impalad 实例却仍然会保持集群的方式处理查询请求，只是无法将各自的状态更新到 State Store 中，如果这个时候新加入一个 impalad 实例，则新加入的 impalad 实例不为现有集群中的其他 impalad 实例所识别，然而，State Store 一旦重启，则所有 State Store 所服务的各个 impalad 实例（包括 state store 挂掉期间新加入的 impalad 实例）的信息（由 impalad 实例发给 state store）都会进行重建。

Impalad,对应进程为 impalad(核心进程,数据的计算靠这个进程来执行),该进程应运行在 DataNode 机器上，每个 impalad 实例会接收、规划并调节来自 ODBC 或 Impala Shell 等客户端的查询。每个 impalad 实例会充当一个 Worker，处理由其它 impalad 实例分发出来的查询片段(query fragments)。客户端可以随便连接到任意一个 impalad 实例,被连接的 impalad 实例将充当本次查询的协调者（Ordinator)，将查询分发给集群内的其它 impalad 实例进行并行计算。当所有计算完毕时，其它各个 impalad 实例将会把各自的计算结果发送给充当 Ordinator 的 impalad 实例，由这个 Ordinator 实例把结果返回给客户端。每个 impalad 进程可以处理多个并发请求。

Impala shell，这是一个客户端工具，该客户端工具提供一个交互接口，供使用者发起数据查询或管理任务，比如连接到 impalad。这些查询请求会传给 ODBC 这个标准查询接口。

在安装 Impala 之前，需要注意几个重要的限制条件。首先 Impala 能使用的内存无法超过系统的硬件可用内存（查询需要的内存如果超出硬件内存，则查询将失败)，Impala 对内存要求非常高，典型的硬件内存为：32~48G，如果内存不足，甚至会导致服务不能正常启动。

Impala 只支持 redhat 5.7/centos 5.7 或 redhat 6.2/centos 6.2 以上（建议安装在 64 位系统上)，不支持 Ubuntu，而且安装 Impala 的节点上需要已经安装了 CDH（后续的章节会进行介绍)。还需要已经安装了 Hive，并配置一个外部数据库（如 MySQL）供 Hive 存储元数据。可通过执行下面的命令来判断 Hive 是否安装正常：

```
$ hive

hive> show tables;

OK

Time taken: 2.809 seconds
```

Impala 不支持以下特性：

- 查询流数据
- 删除数据
- 索引
- YARN 集成
- 全文搜索
- 不具有像 Hive SerDe 的可扩展机制
- 不支持线上查询容错，如果查询出错，如机器宕机，Impala 将会丢弃本次查询
- 不支持表和列级别的授权

- impalad 实例之间的传输没有加密
- 不支持 Hive UFS
- 当前版尚不支持 JDBC，计划未来支持

要安装 Imapla，首先下载它的 yum repository （考虑到内存和性能问题，如果机器数允许，建议 Impalad 实例不要跟 NameNode 运行在同一台机，但却需与 DataNode 安装在同一台机，以免影响 Impala 整体性能），因 Impala 的 rpm 包比较大，且需要在多部机器上安装，故建议直接下载 rpm 包，然后通过 rpm -ivh 的方式安装。

rpm 包列表如下：

RPM 包文件名
impala-0.5-1.p0.491.el6.x86_64.rpm
impala-debuginfo-0.5-1.p0.491.el6.x86_64.rpm
impala-server-0.5-1.p0.491.el6.x86_64.rpm
impala-shell-0.5-1.p0.491.el6.x86_64.rpm
impala-state-store-0.5-1.p0.491.el6.x86_64.rpm

其中，除了 impala-debuginfo-0.5-1.p0.491.el6.x86_64.rpm 可以不下载之外，其它几个包都是必须的，尤其是 impala-0.5-1.p0.491.el6.x86_64.rpm。

上述各个包的作用如下：

impala-state-store-0.5-1.p0.491.el6.x86_64.rpm 是运行 statestored 实例所需要的包，该包依赖 impala-0.5-1.p0.491.el6.x86_64.rpm（核心包）。

impala-0.5-1.p0.491.el6.x86_64.rpm（核心包）和 impala-server-0.5-1.p0.491.el6.x86_64.rpm（扩展包，需首先安装核心包）是运行 impalad 实例所需要的包。

impala-shell-0.5-1.p0.491.el6.x86_64.rpm 是运行 impala-shell 客户端所需要的包。

如果选择用 yum 的方式来安装，则请将下面的 repo 文件拷贝到 /etc/yum.repos.d/ 目录下：

```
[cloudera-impala]

name=Impala

baseurl=http://beta.cloudera.com/impala/redhat/6/x86_64/impala/0/

gpgkey                                                          =
http://beta.cloudera.com/impala/redhat/6/x86_64/impala/RPM-GPG-KEY-
cloudera

gpgcheck = 1
```

在 DataNode 节点安装 impalad 后台实例：

```
sudo yum install -y impala-server（会自动安装 impala 核心包）
```

安装 statestored 实例：

```
sudo yum install -y impala-state-store（会自动安装 impala 核心包）
```

安装 impala-shell 客户端：

```
sudo yum install -y impala-shell
```

进入 Impala 安装目录，默认为/usr/lib/impala（可通过 rpm -ql impala 查看），查看是否存在 conf 目录，如果不存在，则需要手动创建。创建 conf 目录是为了存放 impalad 的配置文件。impalad 的配置文件路径由环境变量 IMPALA_CONF_DIR 指定，默认为/usr/lib/impala/conf。

拷贝 hive-site.xml、core-site.xml、hdfs-site.xml（只需从 Hadoop 和 Hive 配置文件目录中拷贝过来)至/usr/lib/impala/conf 目录下(假设 impalad 的配置文件路径为/usr/lib/impala/conf)，并作下面修改：

在 core-site.xml 文件中添加如下内容（如果不存在的话）：

```
<property>
    <name>dfs.client.read.shortcircuit</name>
    <value>true</value>
</property>
<property>
    <name>dfs.client.read.shortcircuit.skip.checksum</name>
    <value>false</value>
</property>
```

在 hdfs-site.xml 文件中添加如下内容（如果不存在的话）：

```
<property>
    <name>dfs.datanode.data.dir.perm</name>
    <value>755</value>
</property>
<property>
    <name>dfs.block.local-path-access.user</name>
```

```
    <value>hadoop</value>
</property>
<property>
    <name>dfs.datanode.hdfs-blocks-metadata.enabled</name>
    <value>true</value>
</property>
```

以下为 impala 服务启动命令，先启动 statestored（默认端口为 24000）：

```
statestored -state_store_port=24000
```

再启动 impalad 实例：

```
impalad HADOOP_CONF_DIR= ” /usr/lib/impala/conf ” impalad -state_store_host=192.168.22.30 -nn=192.168.22.30 -nn_port=12900 -hosame=192.168.22.34 -ipaddress=192.168.22.34
```

需要加上 HADOOP_CONF_DIR，否则在 impala 查询数据，可能会报类似 Wrong FS 。。。expect 。。。的错误。

命令中的-nn 和-nn_port，表示 NameNode 的主机和端口，因 Hadoop 2 以上的版本对 NameNode 采用 HA 的方式，对外提供 NameService 而不是某个具体的 NameNode，然而这里 impalad 启动时却依然需要知道某个具体的 NameNode 的主机和端口。

在第一次启动 impalad 的时候，可能会遇到 impalad 报类似找不到 JDBC 数据库驱动（假设为 MYSQL)的问题，这是因为 impalad 默认使用的数据库驱动包的位置为：/usr/share/java/mysql-connector-java.jar，该配置默认由/etc/default/impala 文件中的 MYSQL_CONNECTOR_JAR 项指定，可以根据实际情况修改。

Impala 启动和可以使用 Hive 相似的语法进行数据操作。

第8章 分布式数据库 HBase

第1节 HBase 基本介绍

在介绍 HBase 之前，首先关注以下 NoSQL 的定义，NoSQL 就是“Not Only SQL”缩写，泛指用来解决大数据相关问题而创建的数据库技术，NoSQL 技术不会完全替代关系型数据库。

如果需要实时地随机访问超大规模数据集，就可以使用 HBase 这一 Hadoop 应用。它是一个在 HDFS 上开发的面向列的分布式数据库。

虽然数据库存储和检索的实现可以选择很多不同的策略，但是绝大多数解决办法-特别是关系数据库技术的变种－不是为大规模可伸缩的分布式处理设计的。很多厂商提供了复制（replication）和分区（partitioning）解决方案，让数据库能够从单个节点上扩展出去，但是这些附加的技术大都属于“事后”的解决办法，而且非常难以安装和维护。并且这些解决办法常常要牺牲一些重要的 RDBMS 特性。

在一个“扩展的”RDBMS 上，连接、复杂查询、触发器、视图以及外键约束这些功能要么运行开销大，要么根本无法用。HBase 从另一个方向来解决可伸缩性的问题。它自底向上地进行构建，能够简单地通过增加节点来达到线性扩展。

HBase 并不是关系型数据库，它不支持 SQL。但在特定的问题空间里，它能够做 RDBMS 不能做的事：在廉价硬件构成的集群上管理超大规模的稀疏表。

HBase 是 Apache 的顶级开源项目，本质上是 bigtable 的开源山寨版本。建立的 HDFS 之上，提供高可靠性、高性能、列存储、可伸缩、实时读写的数据库系统， 它介于 NoSQL 和 RDBMS 之间，仅能通过主键(row key)和主键的 range 来检索数据，仅支持单行事务(可通过 hive 支持来实现多表 join 等复杂操作)。主要用来存储非结构化和半结构化的松散数据。

第2节 HBase 核心概念

HBase 的数据存放在带标签的表中。表由行和列组成。表格的“单元格”(cell)由行和列的坐标交叉决定，是有版本的。默认情况下，版本号是自动分配的，为 HBase 插入单元格时的时间戳。单元格的内容是未解释的字节数组。

表中行的键也是字节数组。所以理论上，任何东西都可以通过表示成字符串或将二进制形式转化为长整型或直接对数据结构进行序列化，来作为键值。表中的行根据行的键值（也就是表的主键）进行排序。排序根据字节序进行。所有对表的访问都要通过表的主键。

Hbase 表中的每个列，都归属与某个列族。列族是表的 Schema 的一部分(而列不是)，必须在使用表之前定义。列名都以列族作为前缀。例如 courses:history ， courses:math 都属于 courses 这个列族，列族中可以动

态添加新的列。

HBase 中通过 row 和 columns 确定的为一个存贮单元称为 cell。每个单元都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 HBase 在数据写入时自动赋值，此时时间戳是精确到毫秒的当前系统时间。

时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell 中，不同版本的数据按照时间倒序排序。

为了节省空间应对稀疏的数据，HBase 列可以被赋空值来节省空间。

与 NoSQL 数据库们一样,row key 是用来检索记录的主键。访问 Hbase table 中的行，只有三种方式：

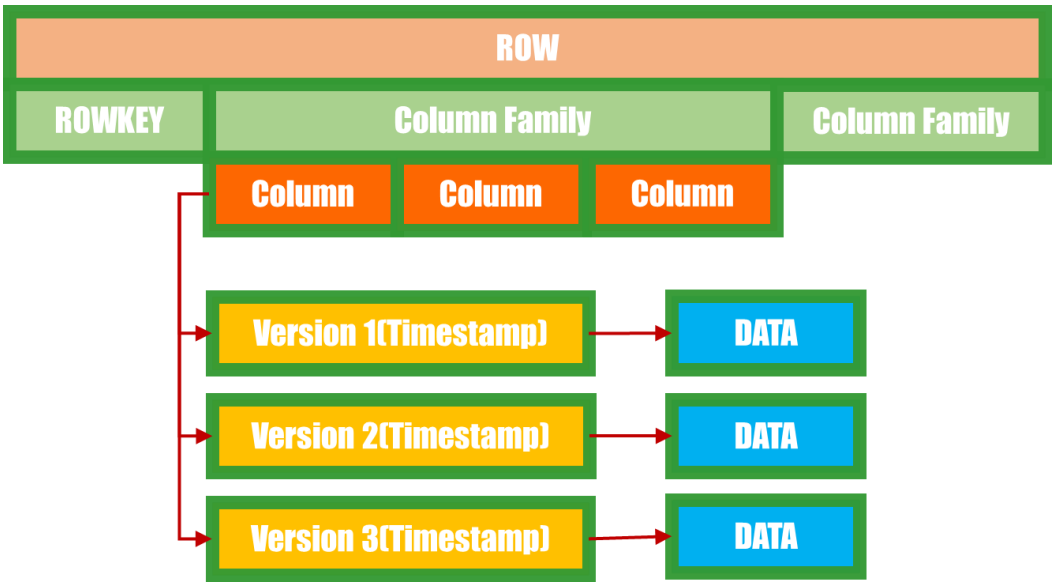
- 通过单个 row key 访问
- 通过 row key 的 range
- 全表扫描

Row key 行键 (Row key)可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在 Hbase 内部，row key 保存为字节数组。存储时，数据按照 Row key 的字典序(byte order)排序存储。设计 key 时，要充分利用排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)。

物理上，所有的列族成员都一起存放在文件系统中。所以，虽然把 HBase 描述为一个面向列的存储器，但实际上更准确的说法是：

HBase 是个面向列族的存储器。由于调优和存储都是在列族这个层次进行的，所以最好使所有列族成员都有相同的“访问模式”(access pattern）和大小特征。简而言之，Hbase 表和 RDBMS 中的表类似，单元格有版本，行是排序的，而只要列族预先存在，客户端随时可以把列添加到列族中去。

Hbase 存储格式如下：



通过分析，HBase 和传统 RDBMS 在存储方面的差异如下描述，一个典型的 RDBMS 二维表格式如下：

Primary key	Last Name	First Name	Account Number	Type of Account	Timestamp
1234	Smith	John	abcd1234	Checking	20120118
1235	Johnson	Michael	wxyz1234	Checking	20120118
1235	Johnson	Michael	aabb1234	Checking	20111123

对应的数据在 HBase 中的存储格式如下所示：

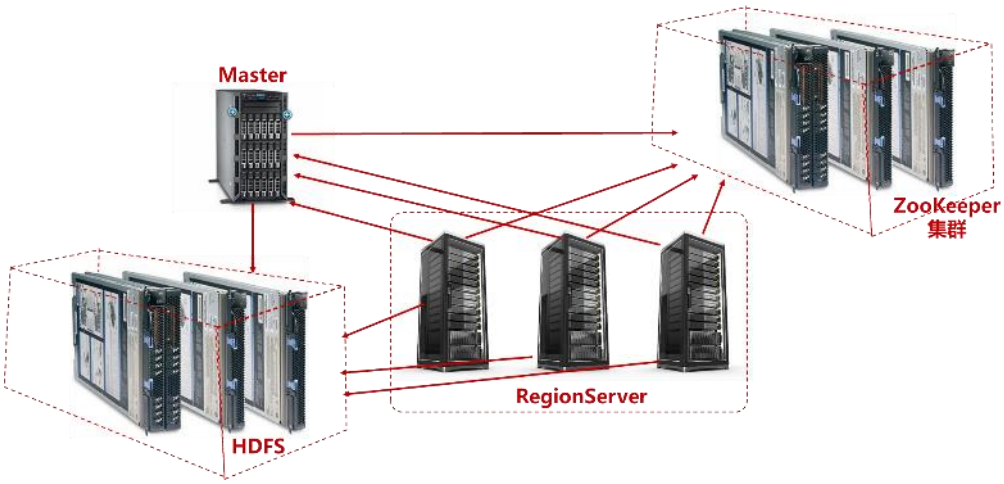
Row key	Value (CF, Column, Version, Cell)
1234	info: { ‘lastName’ : ‘Smith’ , ‘firstName’ : ‘John’ }
	acct: { ‘checking’ : ‘abcd1234’ }
1235	info: { ‘lastName’ : ‘Johnson’ , ‘firstName’ : ‘Michael’ }
	acct: { ‘checking’ : ‘wxyz1234’ @ts=2012, ‘checking’ : ‘aabb1234’ @ts=2011

HBase 自动把表水平划分成“区域”(region)。每个区域由表中行的子集构成。每个区域由它所属于的表、它所包含的第一行及其最后一行（不包括这行）来表示。

一开始，一个表只有一个区域。但是随着区域开始变大，等到它超出设定的大小阈值，便会在某行的边界上把表分成两个大小基本相同的新分区。在第一次划分之前，所有加载的数据都放在原始区域所在的那台服务器上。

随着表变大，区域的个数也会增加。区域是在 HBase 集群上分布数据的最小单位。用这种方式，一个因为太大而无法放在单台服务器上的表会被放到服务器集群上，其中每个节点都负责管理表所有区域的一个子集。表的加载也是使用这种方法把数据分布到各个节点。在线的所有区域按次序排列就构成了表的所有内容。

常规的 HBase 服务器部署结构如下：



各个角色的功能如下：

Region：

表中一部分数据组成的子集，当 Region 内的数据过多时能够自动分裂，过少时会合并。

Region Server：

维护 Master 分配给它的 region，处理对这些 region 的 IO 请求，负责切分在运行过程中变得过大的 region。

Master：

为 Region server 分配 region，负责 region server 的负载均衡，发现失效的 region server 并重新分配其上的 region，执行 HDFS 上的垃圾文件回收。

Zookeeper

保证任何时候，集群中只有一个 master，存储所有 Region 的寻址入口。实时监控 Region Server 的状态，将 Region server 的上线和下线信息实时通知给 Master。存储 Hbase 的 schema,包括有哪些 table，每个 table 有哪些 column family，处理 Region 和 Master 的失效。

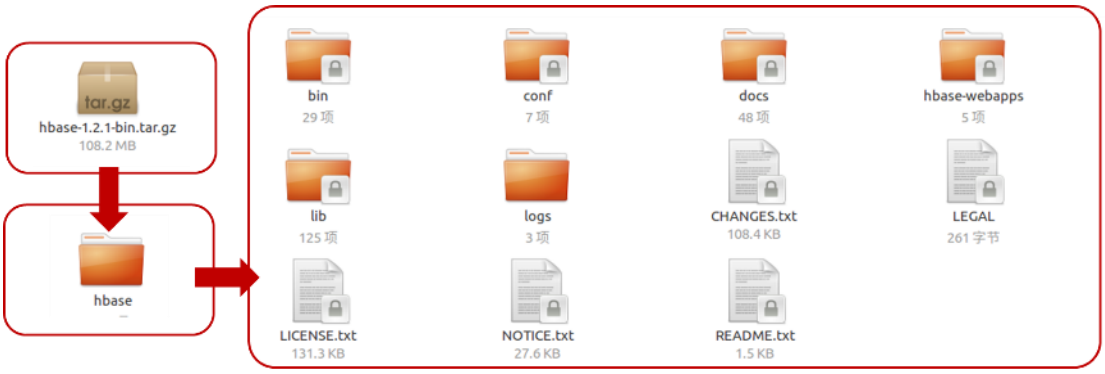
下表列出 Hbase 与 RMDBS 的主要差异：

	HBase	RDBMS
数据类型	只有字符串	丰富的数据类型
数据操作	简单的增删改查	各种各样的函数，表连接
存储模式	基于列存储	基于表格结构和行存储
数据保护	更新后旧版本仍然会保留	替换
可伸缩性	轻易的进行增加节点，兼容性高	需要中间层，牺牲功能
典型的数据大小	TB -PB - 上亿到数十亿条记录	GB-TB - 几十万到几百万条记录

吞吐量	每秒百万次查询	每秒数千次查询

第3节 HBase 安装

HBase 的安装同样只需要 2 个简单的步骤：在官方网站上下载 HBase 的发行包，解压后修改配置文件即可启动 HBase。



在 HBASE_HOME/conf 文件夹中修改配置文件 hbase-site.xml，指定 HDFS 的位置：

```
<property>
    <name>hbase.rootdir</name>
    <value>hdfs://192.168.1.1:9000/hbase</value>
</property>
```

然后为 Hbase 添加环境变量（/etc/profile）：

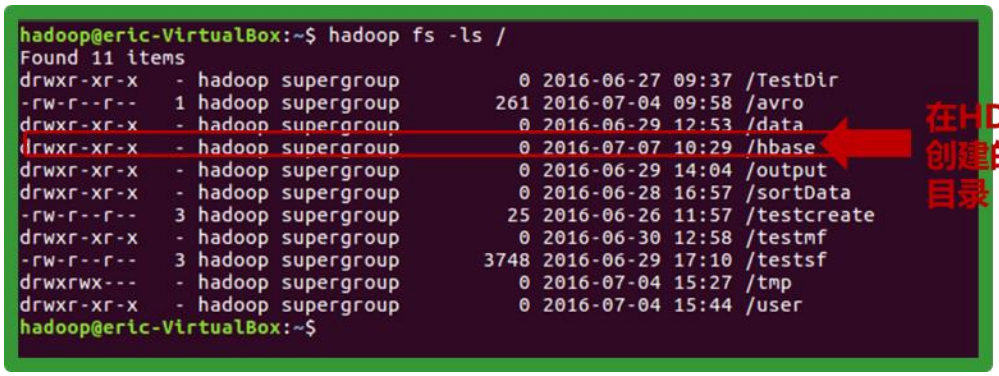
```
export HBASE_HOME=/usr/hbase
export PATH=$PATH:$HBASE_HOME/bin
```

启动 HDFS 后启动 Hbase：

```
start-hbase.sh
```

```
hadoop@eric-VirtualBox:/usr/hbase/bin$ start-hbase.sh
starting master, logging to /usr/hbase/logs/hbase-hadoop-master-eric-VirtualBox.
out
```

启动 HBase 后会在 HDFS 创建 HBase 的数据目录：

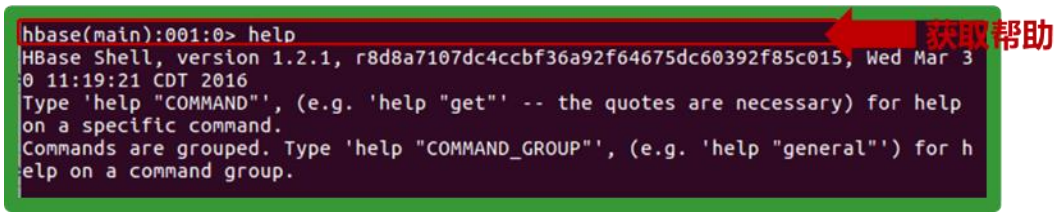


接下来可以启动 HBase 的交互环境 HBase Shell 来验证 HBase 的安装是否成功：

```
hbase shell
```

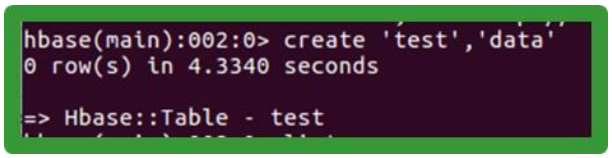
在交互环境中，可以获取帮助：

```
help
```



接下来创建一个测试表：

```
create 'test','data'
```



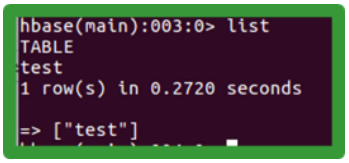
要新建一个表，首先必须为表起一个名字，并为其定义模式。

一个表的模式包含表的属性和列族的列表。列族本身也有属性。可以在定义模式时依次定义它们。例如，列族的属性包括列族是否应该在文件系统中被压缩存储，一个单元格要保存多少个版本等。模式可以被修改，需要修改时把表设为“离线”(offline) 即可。在外壳环境中使用 disable 命令可以把表设为离线，使用 alter 命令可以进行必要的修改，而 enable 命令则可以把表重新设为“在线”(online)。

刚才的命令新建了一个名为 test 的表，使其只包含一个名为 data 的列族，(如果需要多个列族，使用逗号分隔的列表)表和列族属性都为默认值。

列出 HBase 中现有的表：

```
list
```



第4节 HBase 的 Shell 操作

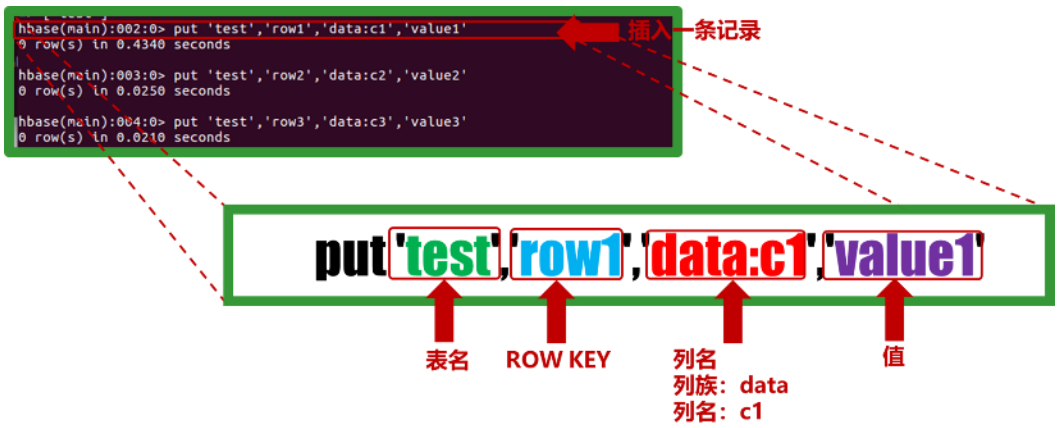
HBase 并不支持 SQL 语句进行数据的查询，事实上，由于 HBase 的数据存取特性，导致针对其进行操作的语言远远不及 SQL 这类结构化的查询语言复杂，通过几条简单的命令指令就能完成对数据的更新和查询操作。

下表列举了常见的 HBase Shell 命令：

名称	命令表达式
创建表	create ’表名称’, ’列族名称 1’,’列族名称 2’,’列族名称 N’
添加记录	put ‘表名称’, ‘行键’, ’列名称:’, ’值’
查看记录	get ’表名称’, ’行键’
查看表中的记录总数	count ’表名称’
删除记录	delete ’表名’,’行键’, ’列名称’
删除一张表	先要屏蔽该表，才能对该表进行删除，第一步 disable ’表名称’ 第二步 drop ’表名称’
查看所有记录	scan ″表名称″
查看某个表某个列中所有数据	scan ″表名称″, {COLUMNS=>’列族名称:列名称’}
更新记录	就是重写一遍进行覆盖

要在上一节中构建的示例表 test 的列族 data 中三个不同的行和列上插入数据，然后列出表的内容，输入如下：

```
put ’test’,’row1’,’data:c1’,’value1’
put ’test’,’row2’,’data:c2’,’value2’
put ’test’,’row3’,’data:c3’,’value3’
```



接下来可以通过 scan 扫描表，查看刚才插入的 3 条记录：

```
scan  ‘test’
```

```
hbase(main):005:0> scan 'test'
ROW                   COLUMN+CELL
 row1                 column=data:c1, timestamp=1467871101377, value=value1
 row2                 column=data:c2, timestamp=1467871108996, value=value2
 row3                 column=data:c3, timestamp=1467871118096, value=value3
```

还可以通过 describe 命令显示表的描述信息：

```
describe ’表名’
```

```
hbase(main):006:0> describe 'test'
Table test is ENABLED
test
COLUMN FAMILIES DESCRIPTION
{NAME => 'data', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KE
EP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', CO
MPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65
536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0910 seconds
```

同样在交互环境中可以查看是否存在某一个特定的表：

```
exists ’表名’
```

查看一个表是否启用：

```
is_enabled ’表名’
```

查看一个表是否被禁用：

```
is_disabled ’表名’
```

如果希望想删除一个表，需要两个步骤，首先要禁用该表：

```
disable ’表名’
```

然后删除该表：

```
drop ’表名’
```

```
hbase(main):010:0> disable 'test'
0 row(s) in 2.4960 seconds
```

```
hbase(main):011:0> drop 'test'
0 row(s) in 1.3950 seconds

hbase(main):012:0> list
TABLE
0 row(s) in 0.0200 seconds

=> []
```

第5节 HBase 操作示例

首先创建一个用于进行示例操作的表：

```
create  ‘users’, ‘user_id’, ‘address’, ‘info’
```

在这个表中插入测试数据：

```
put ’users’,’xiaoming’,’info:age’,’24’
put ’users’,’xiaoming’,’info:birthday’,’1987-06-17’
put ’users’,’xiaoming’,’info:company’,’alibaba’
put ’users’,’xiaoming’,’address:contry’,’china’
put ’users’,’xiaoming’,’address:province’,’zhejiang’
put ’users’,’xiaoming’,’address:city’,’hangzhou’
put ’users’,’zhangyifei’,’info:birthday’,’1987-4-17’
put ’users’,’zhangyifei’,’info:favorite’,’movie’
put ’users’,’zhangyifei’,’info:company’,’alibaba’
put ’users’,’zhangyifei’,’address:contry’,’china’
put ’users’,’zhangyifei’,’address:province’,’guangdong’
put ’users’,’zhangyifei’,’address:city’,’jieyang’
put ’users’,’zhangyifei’,’address:town’,’xianqiao’
```

接下来利用这个数据集完成一些基本的业务操作，首先可以取得一个 id 的所有数据：

```
get ‘users’, ‘xiaoming’
```

```
hbase(main):030:0> get 'users','xiaoming'
COLUMN                          CELL
 address:city                   timestamp=1467872797177, value=hangzhou
 address:contry                 timestamp=1467872778597, value=china
 address:province               timestamp=1467872787686, value=zhejiang
 info:age                       timestamp=1467872754328, value=24
 info:birthday                  timestamp=1467872761164, value=1987-06-17
 info:company                   timestamp=1467872769618, value=alibaba
6 row(s) in 0.1060 seconds
```

接下来可以提供更为精准的查询目标，例如获取一个 id，一个列族的所有数据：

```
get ‘users’, ‘xiaoming’, ‘info’
```

```
hbase(main):031:0> get 'users','xiaoming','info'
COLUMN                          CELL
 info:age                       timestamp=1467872754328, value=24
 info:birthday                  timestamp=1467872761164, value=1987-06-17
 info:company                   timestamp=1467872769618, value=alibaba
3 row(s) in 0.0620 seconds
```

更进一步规定需要查询的列，即获取一个 id，一个列族中一个列的所有数据：

```
get ‘users’, ‘xiaoming’, ‘info:age’
```

```
hbase(main):032:0> get 'users','xiaoming','info:age'
COLUMN                          CELL
 info:age                       timestamp=1467872754328, value=24
1 row(s) in 0.0310 seconds
```

在 HBase 中，不存在类似与 RDBMS 中专门用于数据更新的 update 操作，无论是数据插入还是更新，都使用统一的 put 指令。当使用 put 指令向一个表中的某个列插入数据时，如果该数据原来不存在，则执行插入操作，而如果原始数据存在，则执行更新操作。和 RDBMS 不同，在 HBase 执行更新操作时并不会将原有的数据删除替换，而是直接以一个新的版本号额外将新的数据插入到单元格中，这就意味着 HBase 允许通过查询将某一个单元中曾经存在过的所有历史版本数据统一查询出来。例如要想一个特定的单元格中进行多次 put 操作：

```
put ‘users’, ‘xiaoming’, ‘info:age’, ‘29’
get ‘users’, ‘xiaoming’, ‘info:age’
put ‘users’, ‘xiaoming’, ‘info:age’, ‘30’
get ‘users’, ‘xiaoming’, ‘info:age’
```

```
hbase(main):033:0> put 'users','xiaoming','info:age' ,'29'
0 row(s) in 0.0260 seconds

hbase(main):034:0> get 'users','xiaoming','info:age'
COLUMN                          CELL
 info:age                       timestamp=1467873444577, value=29
1 row(s) in 0.0390 seconds

hbase(main):035:0> put 'users','xiaoming','info:age' ,'30'
0 row(s) in 0.0310 seconds

hbase(main):036:0> get 'users','xiaoming','info:age'
COLUMN                          CELL
 info:age                       timestamp=1467873467412, value=30
1 row(s) in 0.0140 seconds
```

值和时间戳都发生了变化

有两种方法可获取单元格中不同版本的数据，第一种是直接利用版本号（每次数据更新，版本号自动会执行+1 操作）：

```
get ‘users’, ‘xiaoming’, {COLUMN=> ‘info:age’,VERSIONS=>3}
```

```
hbase(main):039:0> get 'users','xiaoming',{COLUMN=>'info:age',VERSIONS=>3}
COLUMN                          CELL
 info:age                       timestamp=1467873467412, value=30
1 row(s) in 0.0620 seconds
```

利用版本号获取数据

另一种方法是使用插入数据时的时间戳：

```
get ‘users’, ‘xiaoming’, {COLUMN=> ‘info:age’,TIMESTAMP=>1467873467412}
```

```
hbase(main):040:0> get 'users','xiaoming',{COLUMN=>'info:age',TIMESTAMP=>1467873467412}
COLUMN                          CELL
 info:age                       timestamp=1467873467412, value=30
1 row(s) in 0.0420 seconds
```

利用时间戳获取数据

接下来演示对数据的删除操作，首先删除 xiaoming 值的‘info:age’字段：

```
delete ‘users’, ‘xiaoming’, ‘info:age’
```

```
hbase(main):041:0> delete 'users','xiaoming','info:age'
0 row(s) in 0.0490 seconds

hbase(main):042:0> get 'users','xiaoming'
COLUMN                          CELL
 address:city                   timestamp=1467872797177, value=hangzhou
 address:contry                 timestamp=1467872778597, value=china
 address:province               timestamp=1467872787686, value=zhejiang
 info:birthday                  timestamp=1467872761164, value=1987-06-17
 info:company                   timestamp=1467872769618, value=alibaba
5 row(s) in 0.0590 seconds
```

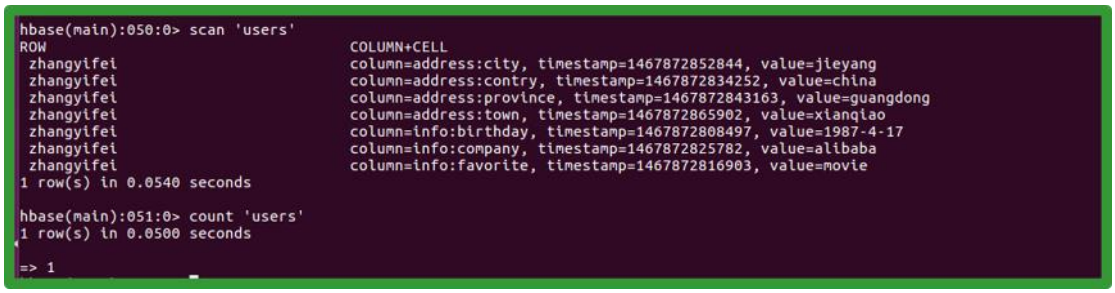
通过扩大参数目标，删除整行：

```
delete ‘users’, ‘xiaoming’
```

```
hbase(main):043:0> deleteall 'users','xiaoming'
0 row(s) in 0.0250 seconds
```

统计表的行数：

```
count  'users'
```



最后是清空表的操作：

```
truncate  'users'
```

第6节 HBase 开发 API

由于 HBase 本身对数据提供的操作命令不多，因此使用对应的 API 对数据进行操作也比较简单。在本节中将会对日常业务中常见的数据操作进行简单的封装，在后续的实际开发工作中，可以直接使用封装好的工具完成对应的数据操作。

在进行数据操作之前，需要明确当前 HBase 的环境参数，即服务器的地址等。可以定义 HBase 的环境配置信息对象和连接对象，并在初始化代码中直接获取：

```
    private static Configuration conf;
    private static Connection con;
    //private static long counter=0;
    // 初始化连接
    static {
        conf = HBaseConfiguration.create(); // 获得配制文件对象
        conf.set("hbase.rootdir",
"hdfs://111.207.49.184:8020/hbase");
        conf.set("hbase.zookeeper.quorum", "111.207.49.184");
        try {
            con = ConnectionFactory.createConnection(conf);// 获得连
接对象
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
```

为了在后续的操作中，保证获取到的连接对象是可用的并且能够正常在操作完成后释放资源，还可以实现统一获取连接和释放连接的方法：

```
// 获得连接
    public static Connection getCon() {
        if (con == null || con.isClosed()) {
            try {
                con = ConnectionFactory.createConnection(conf);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return con;
    }

    // 关闭连接
    public static void close() {
        if (con != null) {
            try {
                con.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
```

接下来封装的是利用表明和列族名称列表作为参数在 HBase 中构建一个新标的方法，构建新表时应该保证提供的是一个目前系统中不存在的表名，因此在执行实际的建表操作之前先判断提供的表名是否存在，如果已经存在，则跳出本次建表操作：

```
    // 创建表
    public static void createTable(String tableName, String...
FamilyColumn) {
        TableName tn = TableName.valueOf(tableName);
        try {
            Admin admin = getCon().getAdmin();
            if (admin.tableExists(tn)) {
                return;
            }
            HTableDescriptor htd = new HTableDescriptor(tn);
            for (String fc : FamilyColumn) {
                HColumnDescriptor hcd = new HColumnDescriptor(fc);
                htd.addFamily(hcd);
            }
```

```
            admin.createTable(htd);
            admin.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

另外一个针对表结果的重要操作是删除一个已经存在的表，其参考实现如下：

```
    // 删除表
    public static void dropTable(String tableName) {
        TableName tn = TableName.valueOf(tableName);
        try {
            Admin admin = getCon().getAdmin();
            // Admin admin = con.getAdmin();
            admin.disableTable(tn);
            admin.deleteTable(tn);
            admin.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

构建好合适的表结构，接下来的操作就是向表中插入数据，插入或更新数据的 API 调用参考实现如下：

```
    // 插入或者更新数据
    public static boolean insert(String tableName, String rowKey,
            String family, String qualifier, String value) {

        System.out.println("insert:------------------------------
"+tableName+"----"+value);
            try {
            Table t =
getCon().getTable(TableName.valueOf(tableName));

    System.out.println(t.getClass()+"############################");
            //t.setWriteBufferSize(128*1024);
            Put put = new Put(Bytes.toBytes(rowKey));
            put.addColumn(Bytes.toBytes(family),
Bytes.toBytes(qualifier),
                    Bytes.toBytes(value));
            t.put(put);
            t.close();
            return true;
```

```
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //HBaseUtil.close();
        }
        return false;
    }
```

根据 HBase 的操作特性，针对标的删除操作可以有不同的删除范围，如删除一个单元格、删除一个列族，删除一行等等。首先展示的是如何删除一个单元格的数据：

```
// 删除
    public static boolean del(String tableName, String rowKey,
String family,
            String qualifier) {
        try {
            Table t =
getCon().getTable(TableName.valueOf(tableName));
            Delete del = new Delete(Bytes.toBytes(rowKey));

            if (qualifier != null) {
                del.addColumn(Bytes.toBytes(family),
Bytes.toBytes(qualifier));
            } else if (family != null) {
                del.addFamily(Bytes.toBytes(family));
            }
            t.delete(del);
            return true;
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            HBaseUtil.close();
        }
        return false;
    }
```

接下来演示如何删除一行中的一个列族：

```
// 删除一行下的一个列族
    public static boolean del(String tableName, String rowKey,
String family) {
        return del(tableName, rowKey, family, null);
    }
```

可以看出，删除一个列族的操作实际上调用了删除单元格的方法，不过在列名的参数中提供了 null，同理，要删除一行也可以使用这个方法来完成：

```
    // 删除一行
    public static boolean del(String tableName, String rowKey) {
        return del(tableName, rowKey, null, null);
    }
```

查询数据也有很多不同的需求，例如直接查询一个单元格的数据，查询一行的数据，根据 row key 的范围来检索数据等，接下来对这些需求的实现方法进行一一展示，首先是读取一个单元格中的数据：

```
    // 数据读取
    // 取到一个值
    public static String byGet(String tableName, String rowKey,
String family,
            String qualifier) {
        try {
            Table t =
getCon().getTable(TableName.valueOf(tableName));
            Get get = new Get(Bytes.toBytes(rowKey));
            get.addColumn(Bytes.toBytes(family),
Bytes.toBytes(qualifier));
            Result r = t.get(get);
            return
Bytes.toString(CellUtil.cloneValue(r.listCells().get(0)));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

然后是获取一个列族的值：

```
    // 取到一个族列的值
    public static Map<String, String> byGet(String tableName,
String rowKey,
            String family) {
        Map<String, String> result = null;
        try {
            Table t =
getCon().getTable(TableName.valueOf(tableName));
            Get get = new Get(Bytes.toBytes(rowKey));
            get.addFamily(Bytes.toBytes(family));
            Result r = t.get(get);
            List<Cell> cs = r.listCells();
            result = cs.size() > 0 ? new HashMap<String, String>() :
result;
            for (Cell cell : cs) {
```

```
            result.put(Bytes.toString(CellUtil.cloneQualifier(cell)),
                    Bytes.toString(CellUtil.cloneValue(cell)));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return result;
    }
```

接下来是通过 row key 获取一行数据的方法：

```
    // 取到多个族列的值
    public static Map<String, Map<String, String>> byGet(String
tableName,
            String rowKey) {
        Map<String, Map<String, String>> results = null;
        try {
            Table t =
getCon().getTable(TableName.valueOf(tableName));
            Get get = new Get(Bytes.toBytes(rowKey));

            Result r = t.get(get);
            List<Cell> cs = r.listCells();
            results = cs.size() > 0 ? new HashMap<String,
Map<String, String>>()
                    : results;
            for (Cell cell : cs) {
                String familyName =
Bytes.toString(CellUtil.cloneFamily(cell));
                if (results.get(familyName) == null) {
                    results.put(familyName, new HashMap<String,
String>());
                }
                results.get(familyName).put(
                        Bytes.toString(CellUtil.cloneQualifier(cell)),
                        Bytes.toString(CellUtil.cloneValue(cell)));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return results;
    }
```

扫描表，获取所有数据的参考实现如下：

```
    // 显示所有数据
    public static List<String> scan(String tableName) {
        try {
            List<String> list = new ArrayList<String>();
            Admin admin = getCon().getAdmin();
            TableName tn = TableName.valueOf(tableName);
            if (!admin.tableExists(tn)) {
                return list;
            }

            Table table =
getCon().getTable(TableName.valueOf(tableName));
            // HTable table = new HTable(getConfiguration(),
tableName);

            Scan scan = new Scan();
            ResultScanner scanner = table.getScanner(scan);
            for (Result result : scanner) {
                Cell[] cells = result.rawCells();
                for (Cell cl : cells) {
                    // kv.getRow();
                    System.out
                            .println(String
                                    .format("row:%s, family:%s,
qualifier:%s, qualifiervalue:%s, timestamp:%s.",

    Bytes.toString(result.getRow()),
                                    Bytes.toString(CellUtil
                                            .cloneFamily(cl)),
                                    Bytes.toString(CellUtil
                                            .cloneQualifier(cl)),
Bytes
                                            .toString(CellUtil

    .cloneValue(cl)),
                                    cl.getTimestamp()));

    list.add(Bytes.toString(CellUtil.cloneValue(cl)));
                }

            }
            scanner.close();
            admin.close();
            return list;
```

```
        } catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
    }
```

最后是根据行健范围为索引查询数据：

```
    // 根据行健范围为索引查询数据
    public static List<String> scanScope(String tableName, String
start,
            String stop) throws IOException {
        try {
            List<String> list = new ArrayList<String>();
            // HBaseAdmin admin = new
HBaseAdmin(getConfiguration());
            Admin admin = getCon().getAdmin();
            TableName tn = TableName.valueOf(tableName);
            if (!admin.tableExists(tn)) {
                return list;
            }

            // HTable table = new HTable(getConfiguration(),
tableName);
            Table table =
getCon().getTable(TableName.valueOf(tableName));
            Scan scan = new Scan();
            scan.setStartRow(Bytes.toBytes(start));
            scan.setStopRow(Bytes.toBytes(stop));
            scan.setBatch(1000);
            ResultScanner scanner = table.getScanner(scan);
            for (Result result : scanner) {
                Cell[] cells = result.rawCells();
                for (Cell cl : cells) {

    list.add(Bytes.toString(CellUtil.cloneValue(cl)));
                }
            }
            scanner.close();
            return list;
        } catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
```

```
    }
```

如果在查询过程中需要获取历史版本的数据，则调用 Scan 对象的如下方法即可：

```
scan.setMaxVersions(版本数);//设置读取的最大的版本数
```

完整代码实现参考本教材的随书代码。

为了简化 Java 开发人员的操作，一些组织也为 HBase 提供了 JDBC 接口，开发人员可以通过习惯的方式对 HBase 数据进行操作：Phoenix 是由 saleforce.com 开源的一个项目，后又捐给了 Apache。它相当于一个 Java 中间件，帮助开发者，像使用 JDBC 访问关系型数据库一样访问 HBase。

可以把 Phoenix 只看成一种代替 HBase 的语法的一个工具。Phoenix 官网上，对 Phoenix 的讲解非常详细，如果希望使用 JDBC 访问 HBase，可以参考其上的文档内容。

第9章 Zookeeper

第1节 Zookeeper 入门介绍

在上一章的 HBase 中就已经接触过 Zookeeper 的概念，原因是 HBase 依赖 Zookeeper 作为节点之间的调度，由于 HBase 发行包中自带 Zookeeper 服务器，而且在默认的基础环境中无需对其进行特使配置，直接使用缺省参数即可，所以并没有对该组件进行详细说明，本章将针对 Zookeeper 的执行机制进行详细的阐述。

那么究竟什么是 Zookeeper 呢？Zookeeper 是 Hadoop 项目的一个子项目，它是一个针对大型分布式系统的可靠协调系统。Zookeeper 在集群中提供的主要功能包括：

- 配置维护
- 名字服务
- 分布式同步
- 组服务

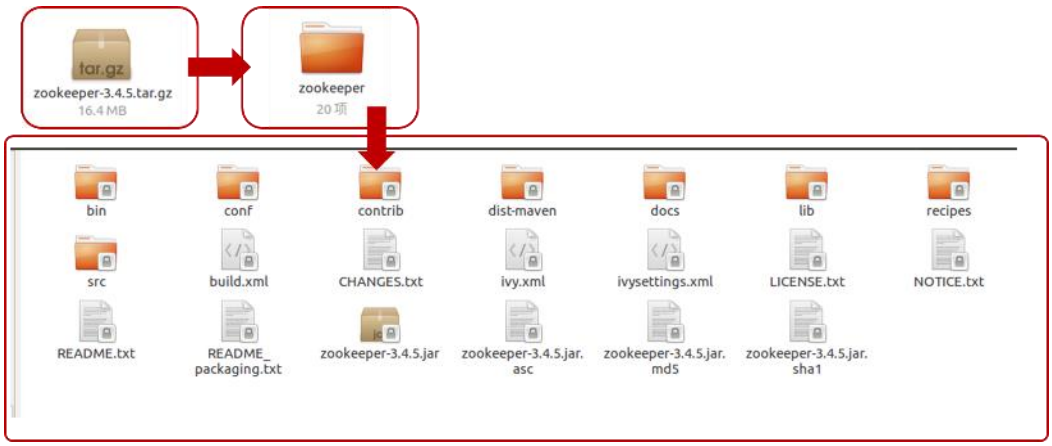
Zookeeper 诞生的原因是大部分分布式应用需要一个主控、协调器或控制器来管理物理分布的子进程。另外，大部分应用需要开发私有的协调程序，缺乏一个通用的机制。

协调程序的反复编写浪费成本，且难以形成通用、伸缩性好的协调器。ZooKeeper 可提供通用的分布式锁服务，用以协调分布式应用。

ZooKeeper 成功地应用于大量的工业程序中。例如，在 Yahoo！被用作雅虎消息代理(Yahoo! Message Broker)的协调和故障恢复服务。那么在普通的业务系统，Zookeeper 能够承担什么常见的角色呢？在 Hadoop 中，它能够确保整个集群只有一个 NameNode，能够存储配置信息。在 HBase 中它能够确保整个集群只有一个 HMaster 并察觉 HRegionServer 联机和宕机以及存储访问控制列表，在 Storm 中它保存 Storm 集群和作业的所有信息并负责 nimbus(master)和 supervisor(slave)的全部通信。

第2节 安装 Zookeeper

首先在官网上下载 Zookeeper 的发行包，解压到安装目标目录：



在/etc/profile 中设置 Zookeeper 环境变量并应用：

```
export ZOOKEEPER_HOME=/usr/zookeeper

export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

接下来根据模版文件修改 Zookeeper 配置文件，配置模版文件的路径位于$ZOOKEEPER_HOME/conf/zoo_sample.cfg，需要将其复制一份，并改名为 zoo.cfg，修改其中的配置参数：

```
tickTime=2000

dataDir=/usr/zookeeper/data

clientPort=2181
```

这三个属性是以独立模式运行 ZooKeeper 所需的最低要求，他们代表的参数含义如下：

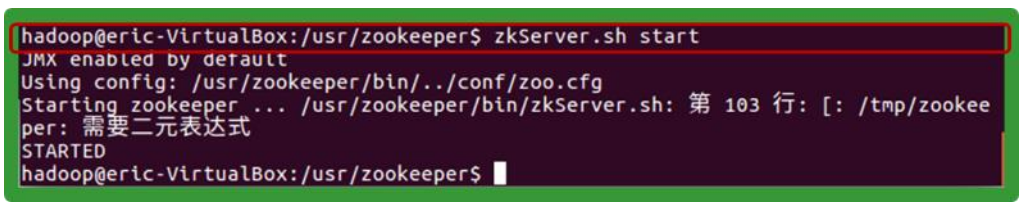
tickTime 属性指定了基本时间单元（以毫秒为单位）

dataDir 属性指定了 ZooKeeper 存储持久数据的本地文件系统位置

clientPort 属性指定了 ZooKeeper 用于监听客户端连接的端口（通常使用 2181 端口）

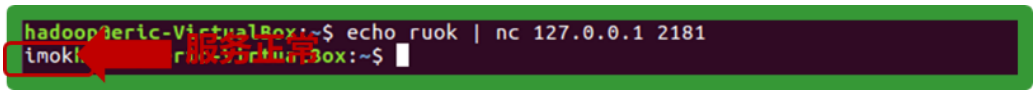
定义好合适的配置文件之后（需要创建配置文件中存放数据的文件夹）可以启动一个本地 Zookeeper 服务器：

```
zkServer.sh start
```



可以使用 nc(telnet 也可以）发送 ruok 命令（Are you OK?）到监听端口，检查 ZooKeeper 是否正在运行：

```
echo ruok | nc 127.0.0.1 2181
```



返回的 imok 是 Zookeeper 在说“I'm OK”。其他一些用于管理 zooKeeper 的命令，都采用类似的四字母组合，如下：

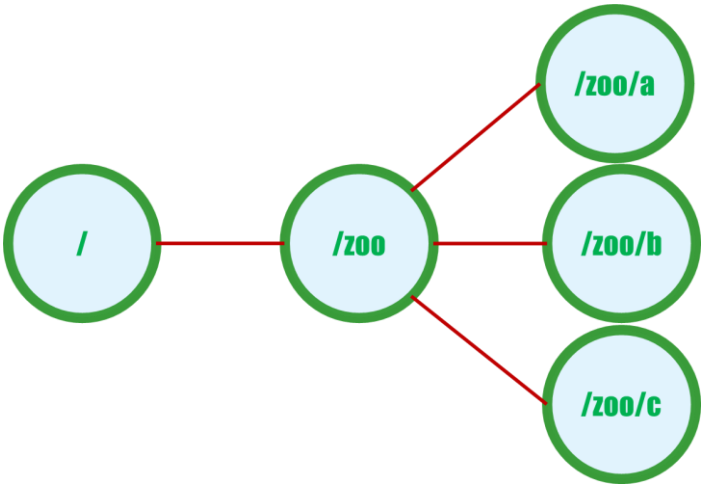
类型	命令	描述
服务器状态	ruok	如果服务器正在运行并且未处于出错状态，则输出 imok
	conf	输出服务器的配置信息（根据配置文件 zoo.cfg）
	envi	输出服务器的环境信息，包括 Zookeeper 版本、Java 版本和其他系统属性
	srvr	输出服务器的统计信息，包括延迟统计、znode 的数量和服务器运行模式
	stat	输出服务器的统计信息和已连接的客户端
	sfst	重置服务器的统计信息
	isro	显示服务器是否处于只读模式或读写模式
客户端连接	dump	列出集合体中所有会话和短暂 znode，必须连接到 leader 才能使用此命令
	cons	列出所有服务器客户端的连接统计信息
	crst	重置连接统计信息
观察	wchs	列出服务器上所有观察的摘要信息
	wchc	按连接列出服务器是那个所有的观察，如果观察数量较多，此命令会影响服务器的性能
	wchp	按 znode 路径列出服务器上所有的观察，如果观察数量较多，此命令会影响服务器的性能
监控	mntr	按 Java 属性格式列出服务器统计信息

第3节 深入理解 Zookeeper

理解 ZooKeeper 的方法就是将其看作一个具有高可用性特征的文件系统。

这个文件系统中没有文件和目录，而是统一使用“节点”(node）的概念，称为 znode。znode 既可以作为保存数据的容器（如同文件），也可以作为保存其他 znode 的容器（如同目录）。

所有的 znode 构成了一个层次化的命名空间，一种自然的建立组成员列表的方式就是利用这种层次结构，创建一个以组名为节点名的 znode 作为父节点，然后以组成员名（服务器名）为节点名来创建作为子节点的 znode。在一个真实的应用中，可以想象将成员相关的数据存储在它们的 znode 中，例如主机名：



可以通过 JavaAPI 构建 Zookeeper 中的组，参考实现如下：

```
public class CreateGroup implements Watcher {
    private static final int SESSION_TIMEOUT = 5000;
    private ZooKeeper zk;
    private CountDownLatch connectedSignal = new CountDownLatch(1);
    public void connect(String hosts) throws IOException,
InterruptedException {
        zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
        connectedSignal.await();
    }
    public void process(WatchedEvent event) { // Watcher interface
        if (event.getState() == KeeperState.SyncConnected) {
            connectedSignal.countDown();
        }
    }
    public void create(String groupName) throws
KeeperException,InterruptedException {
        String path = "/" + groupName;
        String createdPath = zk.create(path, null/* data */,
                Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
        System.out.println("Created " + createdPath);
    }
    public void close() throws InterruptedException {
        zk.close();
    }
    public static void main(String[] args) throws Exception {
        CreateGroup createGroup = new CreateGroup();
        createGroup.connect("127.0.0.1");
        createGroup.create("zoo");
        createGroup.close();
    }
```

```
}
```

程序运行结果如下：

main() 方法执行时，创建了一个 CreateGroup 的实例然后调用了这个实例的 connect() 方法。

connect 方法实例化了一个新的 ZooKeeper 类的对象，这个类是客户端 API 中的主要类，用于维护客户端和 ZooKeeper 服务之间的连接。

ZooKeeper 类的构造函数有三个参数：

第一个参数是 zooKeeper 服务的主机地址（可指定端口，默认端口是 2181）

第二个参数是以毫秒为单位的会话超时参数（案例设成 5 秒）

第三个参数是一个 Watcher 对象的实例。Watcher 对象接收来自于 ZooKeeper 的回调，以获得各种事件的通知。

当一个 ZooKeeper 的示例被创建时，会启动一个线程连接到 Zookeeper 服务。由于对构造函数的调用是立即返回的，因此在使用新建的 Zookeeper 对象前一定要等待其与 Zookeeper 服务之间成功建立连接。

使用 Java 的 CountDownLatch 类（位于 java.util.concurrent 包中）来阻止使用新建的 ZooKeeper 对象，直到这个 Zookeeper 对象已经准备就绪。

Watcher 类被用于获取 Zookeeper 对象是否准备就绪的信息，在它的接口中只有一个方法：

```
public void process(WatchedEvent event)
```

当客户端已经与 Zookeeper 服务建立连接后，Watcher 的 process()方法会被调用，参数是一个用于表示该连接的事件。

在接收到一个连接事件（以 Watcher.Event.Keeperstate 的枚举型值 SyncConnected 来表示）时，通过调用 CountDownLatch 的 countDown() 方法来递减它的计数器。

锁存器（latch）被创建时带有一个值为 1 的计数器，用于表示在它释放所有等待线程之前需要发生的事件数。在调用一次 countDown() 方法之后，计数器的值变为 0，则 await() 方法返回。

connect() 方法返回后下一个执行的是 CreateGroup 的 create()方法。

在这个方法中，使用 ZooKeeper 实例中的 create() 方法来创一个新的 Zookeeper 的 znode。所需的参数包括：

- 路径（用字符串表示）
- node 的内容（字节数组，示例代码中使用空值）
- 访问控制列表（简称 ACL，示例代码中使用了完全开放的 ACL，允许任何客户端对 znode 进行读写）
- 创建 znode 的类型。有两种类型的 znode：
  - 短暂的（ephemeral）
    - 创建 znode 的客户端断开连接时，无论客户端是明确断开还是因为任何原因而终止，短暂 znode 都会被 ZooKeeper 服务删除。
  - 持久的（persistent）。
    - 当客户端断开连接时，持久 znode 不会坡删除。

create() 方法的返回值是 ZooKeeper 所创建的节点路径，可以用这个返回值来打印一条表示节点路径被成功创建的消息。

接下来是一段用于注册组成员的程序。

每个组成员将作为一个程序运行，并且加入到组中。当程序退出时，这个组成员应当从组中被删除。为了实现这一点，在 ZooKeeper 的命名空间中使用短暂 znode 来代表一个组成员。

```
public class ConfigWatcher implements Watcher {

    private ActiveKeyValueStore store;

    public ConfigWatcher(String hosts) throws IOException,
InterruptedException {
        store = new ActiveKeyValueStore();
        store.connect(hosts);
    }
    public void displayConfig() throws InterruptedException,
KeeperException {
        String value = store.read(ConfigUpdater.PATH, this);
        System.out.printf("Read %s as %s\n", ConfigUpdater.PATH,
value);
    }
    public void process(WatchedEvent event) {
        if (event.getType() == EventType.NodeDataChanged) {
            try {
                displayConfig();
            } catch (InterruptedException e) {
                System.err.println("Interrupted. Exiting.");
                Thread.currentThread().interrupt();
            } catch (KeeperException e) {
```

```
                System.err.printf("KeeperException: %s. Exiting.\n",
e);
            }
        }
    }
    public static void main(String[] args) throws Exception {
        ConfigWatcher configWatcher = new
ConfigWatcher("127.0.0.1");
        configWatcher.displayConfig();
        Thread.sleep(Long.MAX_VALUE);
    }

}
```

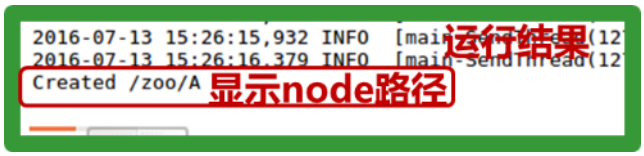
该程序将成员加入组：

```
public class JoinGroup extends ConnectionWatcher {

    public void join(String groupName, String memberName)
            throws KeeperException, InterruptedException {
        String path = "/" + groupName + "/" + memberName;
        String createdPath = zk.create(path, null/* data */,
                Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
        System.out.println("Created " + createdPath);
    }

    public static void main(String[] args) throws Exception {
        JoinGroup joinGroup = new JoinGroup();
        joinGroup.connect("127.0.0.1");
        joinGroup.join("zoo", "A");

        // 保持短暂znode
        Thread.sleep(Long.MAX_VALUE);
    }

}
```

JoinGroup 的代码与 CreateGroup 的非常相似。在它的 join() 方法中，创建短暂 znode 作为组 znode 的子节点，然后通过休眠来模拟正在做某种工作，直到该进程被强行终止。

随着进程终止，这个短暂 znode 被 Zookeeper 删除。

程序的执行结果如下：



以下的代码用于列出组成员：

```
public class ListGroup extends ConnectionWatcher {

    public void list(String groupName) throws KeeperException,
            InterruptedException {
        String path = "/" + groupName;
        try {
            List<String> children = zk.getChildren(path, false);
            if (children.isEmpty()) {
                System.out.printf("No members in group %s\n",
groupName);
                System.exit(1);
            }
            for (String child : children) {
                System.out.println(child);
            }
        } catch (KeeperException.NoNodeException e) {
            System.out.printf("Group %s does not exist\n",
groupName);
            System.exit(1);
        }
    }

    public static void main(String[] args) throws Exception {
        ListGroup listGroup = new ListGroup();
        listGroup.connect("127.0.0.1");
        listGroup.list("zoo");
        listGroup.close();
    }

}
```

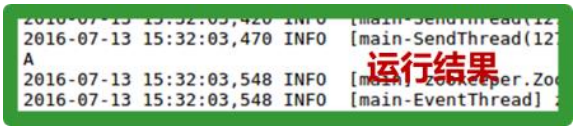
list() 方法中调用了 getchildren() 方法来检索并打印输出一个 znode 的子节点列表，调用参数为该 znode 的路径和设为 false 的观察标志。

如果在一个 znode 上设置了观察标志，那么一旦该 znode 的状态改变，关联的观察（Watcher）会被触发。虽然示例没有使用观察，但在查看一个 znode 的子节点时，通过设置观察可以让应用程序接收到组成员加入、退出和组被删除的有关通知。

要注意：组成员关系管理并不能解决与节点通信过程中出现的网络问题。在

与一个组中的成员节点进行通信的过程中可能会出现故障，这些故障必须以一种合适的方式来解决（重试、使用组中另外一个成员等）。

程序运行结果如下：



Zookeeper 提供了一个用于与其命名空间进行交互的命令行工具。可以使用这个工具列出 / zoo znode 之下的 znode 列表，如下所示：

```
zkCli.sh -server 服务器地址 ls 根节点
```

不使用任何参数直接运行这个命令行工具，可以显示该工其的使用帮助。

Zookeeper 类提供了一个 delete() 方法，该方法有两个参数：

- 节点路径
- 版本号

如果所提供的版本号与 znode 的版本号一致，ZookeePer 会删除这个 znode。

这是一种乐观的加锁机制，使客户端能够检测出对 znode 的修改冲突。

通过将版本号设置为-1，可以绕过这个版本检测机制，不管 znode 的版本号是什么而直接将其删除。

删除组的代码：

```
public class DeleteGroup extends ConnectionWatcher {

    public void delete(String groupName) throws KeeperException,
            InterruptedException {
        String path = "/" + groupName;

        try {
            List<String> children = zk.getChildren(path, false);
            for (String child : children) {
                zk.delete(path + "/" + child, -1);
            }
            zk.delete(path, -1);
        } catch (KeeperException.NoNodeException e) {
            System.out.printf("Group %s does not exist\n",
groupName);
```

```
            System.exit(1);
        }
    }

    public static void main(String[] args) throws Exception {
        DeleteGroup deleteGroup = new DeleteGroup();
        deleteGroup.connect("127.0.0.1");
        deleteGroup.delete("zoo");
        deleteGroup.close();
    }
}
```

Zookeeper 不支持递归删除，因此在删除父节点之前必须先删除子节点。

第4节 Zookeeper 服务

Zookeeper 是一个具有高可用性的高性能协调服务。接下来将从三个方面来了解这个服务：模型、操作和实现。

Zookeeper 维护着一个树形层次结构，树中的节点被称为 znode。znode 可以用于存储数据，并且有一个与之相关联的 ACL。

Zookeeper 被设计用来实现协调服务（这类服务通常使用小数据文件），而不是用于大容量数据存储，因此一个 znode 能存储的数据被限制在 1MB 以内。

Zookeeper 的数据访问具有原子性。客户端在读取一个 znode 的数据时，要么读到所有的数据，要么读操作失败，不会只读到部分数据。同样，一个写操作将替换 znode 存储的所有数据。Zookeeper 会保证写操作不成功就失败，不会出现部分写之类的情况，也就是不会出现只保存客户端所写部分数据的情况。

Zookeeper 不支持添加操作。这些特征都是与 HDFS 所不同的。HDFS 被设计用于大容量数据存储，支持流式数据访问和添加操作。

znode 通过路径被引用，路径体系与 Unix 路径的对比：

像 Unix 中的文件系统路径一样，在 Zookeeper 中路径被表示成用斜杠分隔的 Unicode 字符串。

与 Unix 中的文件系统路径不同的是，ZooKeeper 中的路径必须是绝对路径，也就是说每条路径必须从一个斜杠字符开始。此外，所有的路径表示必须是规范的，即每条路径只有唯一的一种表示方式，不支持路径解析。例如，在 Unix 中，一个具有路径/a/b 的文件也可以通过路径 / a/./b 来表示，原因在于“.”在 Unix 的路径中表示当前目录（“..”表示当前目录的上级目录）。在 Zookeeper 中，不具有这种特殊含义，这样表示的路径名是不合法的。

在 Zookeeper 中，路径由 Unicode 字符串构成，并且有一些限制。

字符串“zookeeper”是一个保留词，不能将它作为路径表示中的一部分。需

要特别指出的是，Zookeeper 使用/zookeeper 子树来保存管理信息，例如关于配额的信息。

Zookeeper 的路径与 URI 不同，前者在 Java API 中通过 java.lang.String 来使用，而后者是通过 Hadoop Path 类（或 java.net.URI 类）来使用。

znode 有两种类型：

- 短暂的
- 持久的

znode 的类型在创建时被确定并且之后不能再修改。

在创建短暂 znode 的客户端会话结束时，Zookeeper 会将该短暂 znode 删除。相比之下，持久 znode 不依赖于客户端会话，只有当客户端（不一定是创建它的那个客户端）明确要删除该持久 znode 时才会被删除。

短暂 znode 不可以有子节点，即使是短暂子节点。虽然每个短暂 znode 都会被绑定到一个客户端会话，但它们对所有的客户端还是可见的（当然，还是要符合其 ACL 的定义）。对干那些需要知道特定时刻有哪些分布式资源可用的应用来说，使用短暂 znode 是一种理想的选择。

顺序 znode 是指名称中包含 Zookeeper 指定顺序号的 znode。如果在创建 znode 时设置了顺序标识，那么该 znode 名称之后便会附加一个值，这个值是由一个单调递增的计数器（由父节点维护）所添加的。例如，如果一个客户端请求创建一个名为 / a/b-的顺序 znode，则所创建 znode 的名字可能是 / a/b-3。如果稍后，另外一个名为 / a/b-的顺序 znode 被创建，计数器会给出一个更大的值来保证 znode 名称的唯一性，例如：/a/b-5。

在 Java 的 API 中，顺序 znode 的实际路径会作为 create() 调用的返回值被传回客户端。

在一个分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端就可以通过顺序号来推断事件的顺序。后续课程将介绍如何使用顺序 znode 来实现共享锁。

znode 以某种方式发生变化时，“观察”(watch)机制可以让客户端得到通知。

可以针对 Zookeeper 服务的操作来设置观察，该服务的其他操作可以触发观察。例如，客户端可以对一个 znode 调用 exists 操作，同时设定一个观察。如果这个 znode 不存在，则客户端所调用的 exists 操作将返回 false。如果一段时间之后，另外一个客户端创建 r 这个 znode，则这个观察会被触发，通知前一个客户端这个 znode 被创建。

观察只能被触发一次（对连接事件的回调除外，这种观察不需要重新注册）。为了能够多次收到通知，客户端需要重新注册所需的观察。在前面的例子中，如果客户端希望收到更多 znode 是否存在的通知（例如在这个：node 被删除时也能收到通知），则需要再次调用 exists 操作来设定一个新的观察。

Zookeeper 中有以下几种基本操作：

操作	描述
create	创建一个 znode（必须要有父节点）
delete	删除一个 znode（该 znode 不能有任何子节点）
exists	测试一个 znode 是否存在并且查询它的元数据
getACL,setACL	获取/设置一个 znode 的 ACL
getChildren	获取一个 znode 的子节点列表
getData,setData	获取/设置一个 znode 所保存的数据
sync	将客户端的 znode 视图与 Zookeeper 同步

Zookeeper 中的更新操作是有条件的。在使用 delete 或 setData 操作时必须提供被更新 znode 的版本号（可以通过 exists 操作获得）。如果版本号不匹配，则更新操作会失败。更新操作是非阻塞操作，因此一个更新失败的客户端（由于其他进程同时在更新同一个 znode）可以决定是否重试，或执行其他操作，并不会因此而阻塞其他进程的执行。

虽然 Zookeeper 可以被看作是一个文件系统，但出于简单性的需要，有一些文件系统的基本操作被它摒弃了。由于 Zookeeper 中的文件较小并且总是被整体读写，因此没有必要提供打开、关闭或查找操作。

Zookeeper 中的写操作具有原子性，一个成功的写操作会保证将数据写到 Zookeeper 服务器的持久存储介质中。然而，Zookeeper 允许客户端读到的数据滞后于 Zookeeper 服务的最新状态，因此客户端可以使用 sync 操作来获取数据的最新状态。

Zookeeper 中有一个被称为 multi 的操作，用于将多个基本操作集合成一个操作单元，并确保这些基本操作同时被成功执行，或者同时失败，不会发生其中部分基本操作被成功执行而其他基本操作失败的情况。

集合更新可以被用于在 Zookeeper 中构建需要保持全局一致性的数据结构，例如构建一个无向图。在 Zookeeper 中用一个 znode 来表示无向图中的一个顶点，为了在两个顶点之间添加或删除一条边，需要同时更新两个顶点所分别对应的两个 znode，因为每个 znode 中都有指向对方的引用。如果只用 Zookeeper 的基本操作来实现边的更新，可能会让其他客户端发现无向图处于不一致的状态，即一个顶点其有指向另一个顶点的引用而对方却没有对应的引用。将针对两个 znode 的更新操作集合到一个 multi 操作中可以保证这组更新操作的原子性，也就保证了一对顶点之间不会出现不完整的连接。

对于 Zookeeper 客户端来说，主要有两种语言绑定可以使用：Java 和 C。

在执行操作时都可以选择同步执行或异步执行。之前已经展示过同步执行的 Java API。下面是 exists 操作的签名，它返回一个封装有 znode 元数据的 Stat 对象（如果 znode 不存在，则返回 null):

```
public Stat exists(String path,Watcher watcher) throws KeeperException,InterruptedException
```

在 ZooKeeper 类中同样可以找到异步执行的签名：

```
public void exists(String path,Watcher watcher,StatCallback cb,Object ctx)
```

因为所有异步操作的结果都是通过回调来传送的，因此在 Java API 中异步方法的返回类型都是 void。调用者传递一个回调的实现，当 Zookeeper 响应时，该回调方法被调用。在这种情况下，回调采用 StatCallback 接口，它有以下方法：

```
public void processResult(int rc,String path,Object ctx,Stat stat)
```

其中 rc 参数是返回代码，对应于 KeeperException 的代码。每个非零代码都代表一个异常，在这种情况下，stat 参数是 null。path 和 ctx 参数对应于客户端传递给 exists() 方法的参数，用于识别这个回调所响应的请求。ctx 参数可以是任意对象，当 path 参数不能提供足够的信息时，客户端可以通过 ctx 参数来区分不同请求。如果 path 参数提供了足够的信息，可以将 ctx 参数设成 null。

两种类型的 API 提供相同的功能，因此选择哪一种只是风格问题。

例如，如果习惯于事件驱动的编程模型，则异步 API 更合适一些。异步 API 允许以流水线方式处理请求，这在某些情况下可以提供更好的吞吐量。想象一下，打算读取一大批 znod，并且分别对它们进行处理。如果使用同步 API，每一个读操作都会阻塞进程，直到该读操作返回；但如果使用异步 API，则可以非常快地启动所有的异步读操作并在另外一个单独的线程中来处理读操作的返回。

在 exists、getchildren 和 getData 这些读操作上可以设置观察，这些观察可以被写操作 create、delete 和 setData 触发。ACL 相关的操作不参与触发任何观察。当一个观察被触发时会产生一个观察事件，这个观察和触发它的操作共同决定着观察事件的类型。

观察被触发的情况如下：

- 当所观察的 znode 被创建、删除或其数据被更新时，设置在 exists 操作上的观察将被触发。
- 当所观察的 znode 被删除或其数据被更新时，设置在 getData 操作上的观察将被触发。创建 znode 不会触发 getData 操作上的观察，因为 getData 操作成功执行的前提是 znode 必须已经存在。
- 当所观察的 znode 的一个子节点被创建或删除时，或所观察的 znode 自己被删除时，设置在 getChildren 操作上的观察将会被触发。可以通过观察事件的类型来判断被删除的是 znode 还是其子节点：
  - NodeDelete 类型代表 znode 被删除
  - NodeChildrenChanged 类型代表一个子节点被删除。

下表列出了观察及其触发操作所对应的事件类型：

设置观察的操作	创建 znode	创建子节点	删除 znode	删除子节点	setData
exists	NodeCreated		NodeDeleted		NodeDataChanged
getData			NodeDeleted		NodeDataChanged
getChildren		NodeChildrenChanged	NodeDeleted	NodeChildrenChanged	

一个观察事件中包含涉及该事件的 znode 的路径，因此对于 NodeCreated 和 NodeDeleted 事件来说，可通过路径来判断哪一个节点被创建或删除。为了能够在 NodeChildrenChonged 事件发生之后判断是哪些子节点被修改，需要重新调用 getChildren 来获取新的子节点列表。与之类似，为了能够在 NodeDataChanged 事件之后获取新的数据，需要调用 getData。在这两种情况下，从收到观察事件到执行读操作（getChildren 或 getData）期间,znode 的状态可能会发生改变。

每个 znode 被创建时都会带有一个 ACL 列表，用于决定谁可以对它执行何种操作。ACL 依赖于 Zookeeper 的客户端身份验证机制。Zookeeper 提供了以下几种身份验证方式：

- Digest 通过用户名和密码来识别客户端
- Sasl 通过 Kerberos 来识别客户端
- IP 通过客户端的 IP 地址来识别客户端

在建立一个 Zookeeper 会话之后，客户端可以对自己进行身份验证。虽然 znode 的 ACL 列表会要求所有的客户端是经过验证的，但 Zookeeper 的身份验证过程却是可选的，客户端必须自己进行身份验证来支持对 znode 的访问。这里有一个使用 digest 方式（用户名和密码）进行身份验证的例子：

```
zk.addAuthInfo("digest","chinasofti:pass".getBytes());
```

每个 ACL 都是身份验证方式、符合该方式的一个身份和一组权限的组合。例如，如果打算给 IP 地址为 10.0.0.1 的客户端对某个 znode 的读权限，可以使用 ip 验证方式、10.0.0.1 和 READ 权限在该 znode 上设置一个 ACL。在 Java 语言中，可以如下所示来创建这个 ACL 对象：

```
new ACL(Perms.READ,new Id("ip","10.0.0.1"));
```

下表列出了一个完整的权限集合：

ACL 权限	允许的操作
CREATE	create（子节点）
READ	getChildren getData
WRITE	setData
DELETE	delete（子节点）
ADMIN	setACL

注意，exists 操作并不受 ACL 权限的限制，因此任何客户端都可以调用 exists 来检索一个 znode 的状态或查询一个 znode 是否存在。

在类 ZooDefs.Ids 中有一些预定义的 ACL,OPEN_ACL_UNSAFE 是其中之一，它将所有的权限（不包括 ADMIN 权限）授予每个人。Zookeeper 还支持插入式身份验证机制，如果需要的话，它可以集成第三方的身份验证系统。

Zookeeper 服务有两种不同的运行模式：

- **独立模式**

只有一个 Zoopeeper 服务器。这种模式较为简单，比较适合于测试环境（甚至可以在单元测试中采用），但是不能保证高可用性和可恢复性。

- **复制模式**

在生产环境中的 Zookeeper 通常以复制模式运行于一个计算机集群上，这个计算机集群被称为一个集合体 。Zookeeper 通过复制来实现高可用性，只要集合体中半数以上的机器处于可用状态，它就能够提供服务。例如，在一个有 5 个节点的集合体中，任意 2 台机器出现故障，都可以保证服务继续，因为剩下的 3 台机器超过了半数。注意，6 个节点的集合体也只能够容忍 2 台机器出现故障，因为如果 3 台机器出现故障，剩下的 3 台机器没有超过集合体的半数。出于这个原因，一个集合体通常包含奇数台机器。

从概念上来说，Zookeeper 是非常简单的：它所做的就是确保对 znode 树的每一个修改都会被复制到集合体中超过半数的机器上。如果少于半数的机器出现故障，则至少有一台机器会保存最新的状态，其余的副本最终也会更新到这个状态。然而，这个简单想法的实现却不简单。ZooKeeper 使用了 Zab 协议。

Zab 协议包括两个可以无限重复的阶段：

- **阶段一：领导者选举**

集合体中的所有机器通过一个选择过程来选出一台被称为“领导者”(leader)的机器，其他的机器被称为“跟随者”(follower)。一旦半数以上（或指定数量）的跟随者已经将其状态与领导者同步，则表明这个阶段已经完成。

- **阶段二：原子广播**

所有的写请求都会被转发给领导者，再由领导者将更新广播给跟随者。当半数以上的跟随者已经将修改持久化之后，领导者才会提交这个更新，然后客户端才会收到一个更新成功的响应。这个用来达成共识的协议被设计成其有原子性，因此每个修改要么成功要么失败。这类似于数据库中的两阶段提交协议。

如果领导者出现故障，其余的机器会选出另外一个领导者，并和新的领导者一起继续提供服务。随后，如果之前的领导者恢复正常，会成为一个跟随者。

领导者选举的过程是非常快的，根据一个已公布的结果来看，只需要大约 200 毫秒，因此在领导者选举的过程中不会出现系统性能的明显降低。在更新内存中的 znode 树之前，集合体中的所有机器都会先将更新写入磁盘。任何一台机器都可以为读请求提供服务，并且由于读请求只涉及内存检索，因此非常快。

Leader 选举步骤：

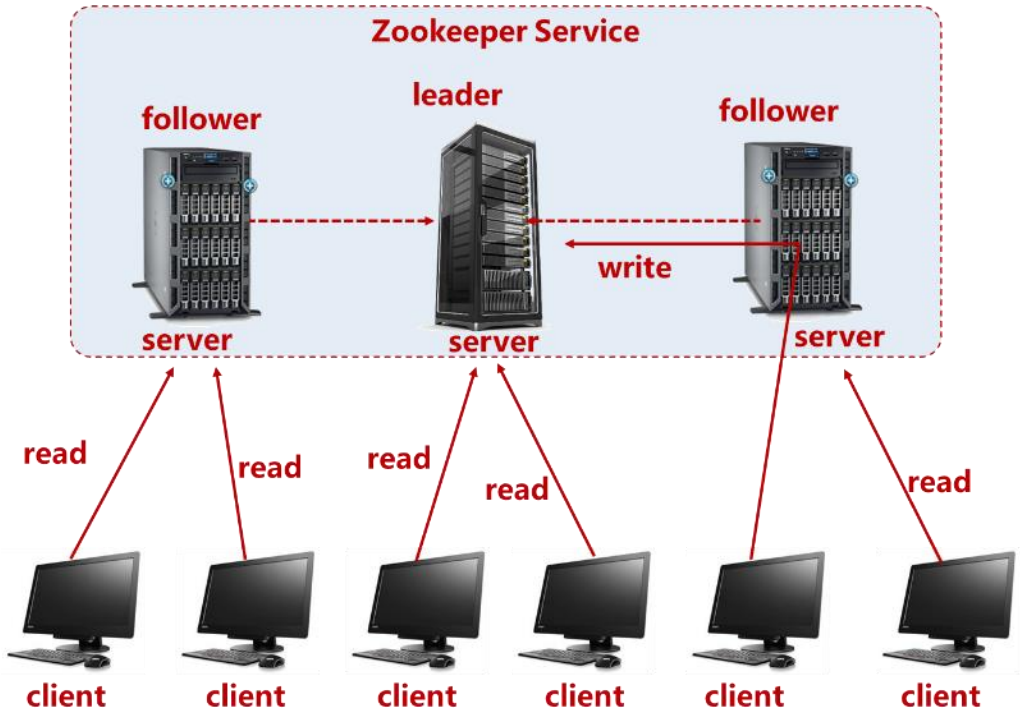
- 每个Server服务器启动以后都询问其它的Server服务器它要投票给谁。
- 对于其他 Server 服务器的询问，Server 每次根据自己的状态都回复自己推荐的 Leader 的 id 和上一次处理事务的 zxid，（系统启动时每个 Server 服务器都会推荐自己）
- Server 服务器收到所有 Server 回复以后，就计算出 zxid 最大的哪个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server。
- 计算的过程中获得票数最多，并且票数超过半数的 Server 服务器就被选为 Leader。否则，继续这个过程，直到 Leader 被选举出来。
- 选出的 Leader 开始等待 Follower 服务器连接。
- Follower 连接 Leader，将最大的 zxid 发送给 Leader。
- Leader 根据 Follower 的 zxid 确定同步点，完成同步后通知 Follower 已经成为 uptodate(现时)状态。
- Follower 收到 uptodate 消息后，就可以接受 Client 的请求进行服务了。

理解 Zookeeper 的实现有助于理解其服务所提供的一致性保证。

在集合体中所使用的术语“领导者”和“跟随者”是恰当的，它们表明一个跟随者可能滞后于领导者几个更新。这也表明在一个修改被提交之前，只需要集合体中半数以上机器已经将该修改持久化即可。

对 Zookeeper 来说，理想的情况就是将客户端都连接到与领导者状态一致的服务器上。每个客户端都有可能被连接到领导者，但客户端对此无法控制，甚至它自己都无法知道是否连接到领导者。

Zookeeper 体系角色可以描述如下：



每一个对 znode 树的更新都被赋予一个全局唯一的 ID，称为 zxid（代表”ZooKeeper Transaction ID”）。Zookeeper 要求对所有的更新进行编号并排序，它决定了分布式系统的执行顺序，如果 zxid z1 小于 z2，则 z1 一定发生在 z2 之前。

在 Zookeeper 的设计中，以下几点考虑保证了数据的一致性：

顺序一致性

来自任意特定客户端的更新都会按其发送顺序被提交。也就是说，如果一个客户端将 znode z 的值更新为 a，在之后的操作中，它又将 z 的值更新为 b，则没有客户端能够在看到 z 的值是 b 之后再看到值 a(如果没有其他对 z 的更新)。

原子性

每个更新要么成功，要么失败。这意味着如果一个更新失败，则不会有客户端看到这个更新的结果。

持久性

一个更新一旦成功，其结果就会持久存在并且不会被撤销。这表明更新不会受到服务器故障的影响。

单一系统映像

一个客户端无论连接到哪一台服务器，它看到的都是同样的系统视图。这意味着，如果一个客户端在同一个会话中连接到一台新的服务器，它所看到的系统状态不会比在之前服务器上所看到的更老。当一台服务器出现故障，导致它的一个客户端需要尝试连接集合体中其他的服务器时，所有状态滞后于故障服务器的服务器都不会接受该连接请求，除非这些服务器将状态更新至故障服务器的水平。

及时性

任何客户端所看到的滞后系统视图都是有限的，不会超过几十秒。这意味着与其允许一个客户端看到非常陈旧的数据，还不如将服务器关闭，强迫该客户端连接到一个状态较新的服务器。

出于性能的原因，所有的读操作都是从 Zookeeper 服务器的内存获得数据，它们不参与写操作的全局排序。如果客户端之间通过 Zookeeper 之外的机制进行通信，则客户端可能会发现它们所看到的 Zookeeper 状态是不一致的。

例如，客户端 A 将 znode z 的值从 x 更新为 y，接着 A 告诉 B 去读 z 的值，而 B 读到的值是 x 而不是 y。这与 Zookeeper 的一致性保证是完全兼容的（这种情况称为“跨客户端视图的同时一致性”）。为了避免这种情况发生，B 应该在读 z 的值之前对 z 调用 sync 操作。sync 操作会强制 B 所连接的 Zookeeper 服务器“赶上”领导者，这样当 B 读 z 的值时，所读到的将会是 A 所更新的（或后来更新的）。

sync 操作只能以异步的方式被调用。不需要等待 sync 调用的返回，Zookeeper 会保证任何后续的操作都在服务器的 sync 操作完成后才执行，哪怕这些操作是在 sync 操作完成之前发出的。

每个 Zookeeper 客户端的配置中都包括集合体中服务器的列表。在启动时，客户端会尝试连接到列表中的一台服务器。如果连接失败，它会尝试连接另一台服务器，以此类推，直到成功与一台服务器建立连接或因为所有 Zookeeper 服务器都不可用而失败。

一旦客户端与一台 Zookeeper 服务器建立连接，这台服务器就会为该客户端创建一个新的会话。每个会话都会有一个超时的时间设置，这个设置由创建会话的应用来设定。如果服务器在超时时间段内没有收到任何请求，则相应的会话会过期。一旦一个会话已经过期，就无法重新被打开，并且任何与该会话相关联的短暂 znode 都会丢失。会话通常都会长期存在，而会话过期则是一种比较罕见的事件，但对于应用来说，如何处理会话过期仍是非常重要的。

只要一个会话空闲超过一定时间，都可以通过客户端发送 ping 请求（也称为心跳）来保持会话不过期。

Ping 请求是由 Zookeeper 的客户端库自动发送，因此在代码中不需要考虑如何维护会话。这个时间长度的设置应当足够低，以便能够检测出服务器故障（由读超时体现），并且能够在会话超时的时间段内重新连接到另外一台服务器。

Zookeeper 客户端可以自动地进行故障切换，切换至另一台 Zookeeper 服务器，并且关健的是，在另一台服务器接替故障服务器之后，所有的会话（和相关的短暂 znode）仍然是有效的。

在故障切换过程中，应用程序将收到断开连接和连接至服务的通知。

当客户端断开连接时，观察通知将无法发送，但是当客户端成功恢复连接后，这些延迟的通知还会被发送。在客户端重新连接至另一台服务器的过程中，如果应用程序试图执行一个操作，这个操作将会失败。这充分说明在真实的 Zookeeper 应用中处理连接丢失异常的甫要性

在 Zookeeper 中“滴答”（tick time）参数的概念，它定义了 Zookeeper 中的基本时间周期，并被集合体中的服务器用来定义相互交互的时间表。

其他设置都是根据滴答参数来定义的，或至少受它限制。例如，会话超时参数的值不可以小于 2 个滴答并且不可以大于 20 个滴答。如果试图将会话超时参数设置在这个范围之外，它将会被自动修改到这个范围之内。

通常将滴答参数设置为 2 秒（2000 毫秒），对应于允许的会话超时范围是 4 到 40 秒。

在选择会话超时设置时有几点需要考虑：

较短的会话超时设置会较快地检测到机器故障，但要避免将会话超时时间设得太低，因为繁忙的网络会导致数据包传输延迟，从而可能会无意中导致会话过期。在这种情况下，机器可能会出现“振动”（flap）现象：在很短的时间内反复出现离开后又重新加入组。

对于那些创建较复杂暂时状态的应用程序来说，由于重建的代价较大，因此比较适合设置较长的会话超时。

在某些情况下，可以对应用程序进行设计，使它能够在会话超时之前重启，从而避免出现会话过期的情况（这适合于对应用进行维护或升级）。

服务器会为每个会话分配一个唯一的 ID 和密码，如果在建立连接的过程中将它们传递给 Zookeeper，可以用于恢复一个会话（只要该会话没有过期）。将会话 ID 和密码保存在稳定存储器中之后，可以将一个应用程序正常关闭，然后在重启应用之前凭借所保存的会话 ID 和密码来恢复会话环境。

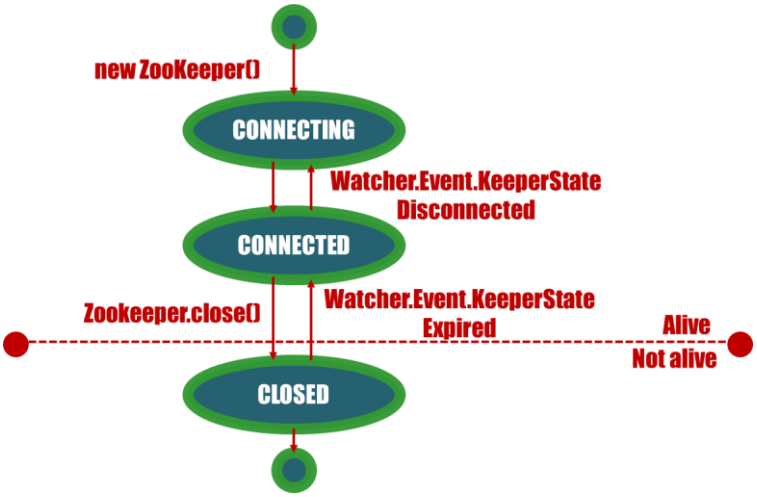
可以将这个特征看成是一种用来帮助避免会话过期的优化技术，但不能因此忽略对会话过期异常的处理，因为机器的意外故障也会导致会话过期，或者，即使应用程序是正常关闭，也有可能因任何原因导致它没有在会话未过期之前完成重启。

一般的规则是，Zookeeper 集合体中的服务器越多，会话超时的设置应越大。

连接超时、读超时和 ping 周期都被定义为集合体中服务器数量的函数，因此集合体中服务器数量越多，这些参数的值反而越小。如果频繁遇到连接丢失的情况，应考虑增大超时的设置。可以使用 JMX 来监控 ZooKeeper 的度量指标，例如请求延迟的统计信息。

Zookeeper 对象在其生命周期中会经历几种不同的状态，可以在任何时刻通过 getState() 方法来查询对象的状态：

```
public States getState()
```



通过注册观察对象，使用了 Zookeeper 对象的客户端可以收到状态转换通知。在进入 CONNECTED 状态时，观察对象会收到一个 WatchedEvent 通知，其中 KeeperState 的值是 SyncConnected。

ZooKeeper 的 Watcher 对象肩负着双重责任：

- 可以被用于获得 Zookeeper 状态变化的相关通知
- 还可以被用于获得 znode 变化的相关通知

传递给 Zookeeper 对象构造函数的(默认的)观察被用于监视其状态的变化。监视 znode 的变化可以使用一个专用的观察对象（将其传递给适当的读操作），也可以通过读操作中的布尔标识来设定是否共享使用默认的观察。

ZooKeeper 实例可以断开然后重新连接到 Zookeeper 服务，此时它的状态就在 CONNECTED 和 CONNECTING 之间转换。如果它断开连接，观察会收到一个 Disconnected 事件。

这些状态转换都是由 ZooKeeper 实例自己发起的，如果连接丢失，它会自动尝试重新连接。如果 close() 方法被调用或出现会话超时（观察事件的 KeeperState 值为 Expired）时，ZooKeeper 实例会转换到第三个状态 CLOSED。

一旦处于 CLOSED 状态，ZooKeeper 对象不再被认为是活跃的(可以对 States 使用 isAlive()方法来测试)，并且不能再用。为了重新连接到 Zookeeper 服务，客户端必须创建一个新的 ZooKeeper 实例。

第5节 使用 Zookeeper 构建应用

配置服务是分布式应用所需要的基本服务之一，它使集群中的机器可以共享配置信息中那些公共的部分。Zookeeper 可以作为一个具有高可用性的配置存储器，允许分布式应用的参与者检索和更新配置文件。

使用 ZooKeeper 中的观察机制，可以建立一个活跃的配置服务，使那些感兴趣的客户端能够获得配置信息修改的通知。

在这里通过两个假设来简化所需实现的服务（稍加修改就可以取消这两个假设）：

- 唯一需要存储的配置数据是字符串，关键字是 znode 的路径，因此在每个 znode 上存储了一个键/值对。
- 在任何时候只有一个客户端会执行更新操作。除此之外，这个模型看起来就像是有一个主人（类似于 HDFS 中的 namenode）在更新信息，而他的工人则需要遵循这些信息。

读写配置信息的示例代码如下：

```
//辅助基类参见之前章节
public class ActiveKeyValueStore extends ConnectionWatcher {

    private static final Charset CHARSET = Charset.forName("UTF-
8");

    //写数据的方法
    public void write(String path, String value) throws
InterruptedException,
            KeeperException {
        Stat stat = zk.exists(path, false);
        if (stat == null) {
            zk.create(path, value.getBytes(CHARSET),
Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT);
        } else {
            zk.setData(path, value.getBytes(CHARSET), -1);
        }
    }
    //读数据的方法
    public String read(String path, Watcher watcher)
            throws InterruptedException, KeeperException {
        byte[] data = zk.getData(path, watcher, null/* stat */);
        return new String(data, CHARSET);
    }

}
```

Write() 方法的任务是将一个关键字及其值写入 Zookeeper。它隐藏了创建一个新的 znode 和用一个新值更新现有 znode 之间的区别，而是使用 exists 操作来检测 znode 是否存在，然后再执行相应的操作。其他值得一提的细节是需要将字符串值转换为字节数组，因为我们只用了 UTF-8 编码的 getBytes() 方法。

Zookeeper 的 getData() 方法有三个参数：

- 路径

- 一个观察对象
- 一个 Stat 对象。

Stat 对象由 getData() 方法返回的值填充，用来将信息回传给调用者。通过这个方法，调用者可以获得一个 znode 的数据和元数据，但在这个例子中，由于我们对元数据不感兴趣，因此将 Stat 参数设为 null。

下例用于说明 ActiveKeyvaluestore 更新配置的用法：

```
public class ConfigUpdater {
    //代表配置信息的节点路径
    public static final String PATH = "/config";

    private ActiveKeyValueStore store;
    private Random random = new Random();

    public ConfigUpdater(String hosts) throws IOException,
InterruptedException {
        store = new ActiveKeyValueStore();
        store.connect(hosts);
    }

    public void run() throws InterruptedException, KeeperException
{
        while (true) {
            //写入随机数据
            String value = random.nextInt(100) + "";
            store.write(PATH, value);
            System.out.printf("Set %s to %s\n", PATH, value);
            TimeUnit.SECONDS.sleep(random.nextInt(10));
        }
    }

    public static void main(String[] args) throws Exception {
        ConfigUpdater configUpdater = new
ConfigUpdater("127.0.0.1");
        configUpdater.run();
    }

}
```

这个程序很简单，ConfigUpdater 中定义了一个 ActiveKeyValueStore,它在 ConfigUpdater 的构造方法中连接到 Zookeeper。

run() 方法永远在循环，在随机时间以随机值更新/config znode。

可以利用如下代码演示如何读取刚才写入的配置信息：

```
//本类作为观察存在，可以接受各事件
public class ConfigWatcher implements Watcher {

    private ActiveKeyValueStore store;

    public ConfigWatcher(String hosts) throws IOException,
InterruptedException {
        store = new ActiveKeyValueStore();
        store.connect(hosts);
    }
    public void displayConfig() throws InterruptedException,
KeeperException {
        String value = store.read(ConfigUpdater.PATH, this);
        System.out.printf("Read %s as %s\n", ConfigUpdater.PATH,
value);
    }
    //处理事件
    public void process(WatchedEvent event) {
        if (event.getType() == EventType.NodeDataChanged) {
            try {
                displayConfig();
            } catch (InterruptedException e) {
                System.err.println("Interrupted. Exiting.");
                Thread.currentThread().interrupt();
            } catch (KeeperException e) {
                System.err.printf("KeeperException: %s. Exiting.\n",
e);
            }
        }
    }
    public static void main(String[] args) throws Exception {
        ConfigWatcher configWatcher = new
ConfigWatcher("127.0.0.1");
        configWatcher.displayConfig();
        Thread.sleep(Long.MAX_VALUE);
    }

}
```

作为配置服务的用户，ConfigWatcher 创建了 ActiveKeyValueStore 对象 store，并且在启动之后调用了 store 一个的 read() 方法（在 displayConfig() 方法中），将自身作为观察传递给 store。

displayConfig() 方法用于显示它所读到的配置信息的初始值。

当 ConfigUpdater 更新 znode 时，Zookeeper 产生一个类型为

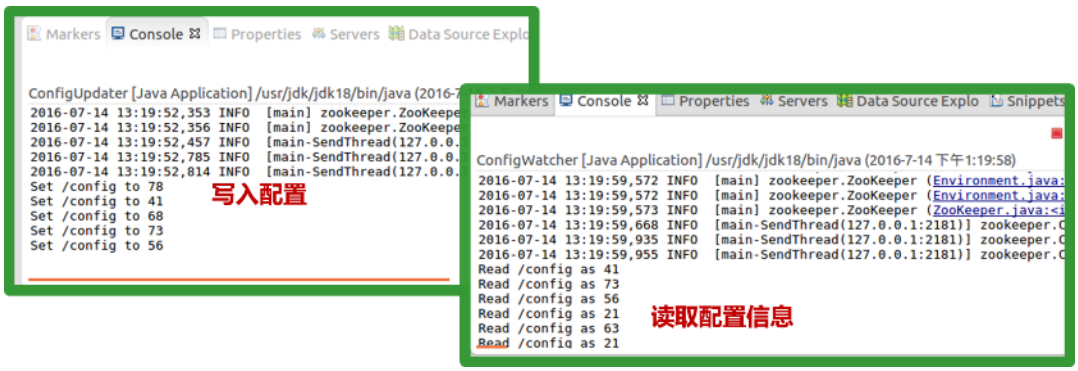
EventType.NodeDataChanged 的事件，从而触发观察。ConfigWatcher 在它的 process() 方法中对这个事件做出反应，读取并显示配置的最新版本。

由于观察仅发送单次信号，因此每次我们调用 ActiveKeyValueStore 的 read() 方法时，都将一个新的观察告知 ZookeePer，以确保我们可以看到将来的更新。

尽管如此，还是不能保证接收到每一个更新，因为在收到观察事件通知与下一次读之间，znode 可能已经被更新过，而且可能是很多次更新，由于客户端在这段时间没有注册任何观察，因此不会收到通知。

对于示例中的配置服务，这不是问题，因为客户端只关心属性的最新值，最新值优先于之前的值。但在一般情况下，这个潜在的问题是不容忽视的。

写入和读取配置信息的运行结果如下：



关于分布式计算的第一个误区是“网络是可靠的”。按照这个观点，程序总是有一个可靠的网络，因此当程序运行在真正的网络中时，往往会出现各种各样的故障。接下来介绍各种可能的故障模式，以及能够解决故障的措施，使程序在面对故障时能够及时复原。

在 Java API 中的每一个 Zookeeper 操作都在其 throws 子句中声明了两种类型的异常，分别是：

- InterruptedException
- KeeperException

如果操作被中断，则会有一个 InterruptedException 异常被抛出。

在 Java 语言中有一个取消阻塞方法的标准机制，即针对存在阻塞方法的线程调用 interrupt()。一个成功的取消操作将产生一个 InterruptedException 异常。Zookeeper 也遵循这一机制，因此你可以使用这种方法来取消一个 Zookeeper 操作。使用了 Zookeeper 的类或库通常会传播 InterruptedException 异常，使客户端能够取消它们的操作。

InterruptedException 异常并不意味着有故障，而是表明相应的操作已经被取消，所以在之前的示例中，可以通过传播异常来中止应用程序的运行。

如果 Zookeeper 服务器发出一个错误信号或与服务器存在通信问题，抛出的则是 KeeperException 异常。

针对不同的错误情况，KeeperException 异常存在不同的子类。例如，KeeperException.NoNodeException 是 KeeperException 的一个子类，如果你图针对一个不存在的 znode 执行操作，抛出的则是该异常。

每一个 KeeperException 异常的子类都对应一个关于错误类型信息的代码。例如，KeeperException.NoNodeException 异常的代码是 KeeperException.Code.NONODE（一个枚举值）。

有两种方法被用来处理 KeeperException 异常：

- 一种是捕捉 KeeperException 异常并且通过检测它的代码来决定采取何种补救措施
- 另一种是捕捉等价的 KeeperException 子类并且在每段捕捉代码中执行相应的操作

KeeperException 异常分为 3 大类型：

- **状态异常**

当一个操作因不能被应用于 znode 树而导致失败时，就会出现状态异常。状态异常产生的原因通常是在同一时间有另外一个进程正在修改 znode。例如，如果一个 znode 先被另外一个进程更新了，根据版本号执行 setData 操作的进程就会失败，并收到一个 KeeperException.BadVersionException 异常，这是因为版本号不匹配。程序员通常都知道这种冲突总是存在的，也都会写代码来进行处理。

一些状态异常会指出程序中的错误，例如 KeeperException·NoChildenForEphemeralsException 异常，试图在短暂 znode 下创建子节点时就会抛出该异常。

- **可恢复的异常**

可恢复的异常是指那些应用程序能够在同一个 Zookeeper 会话中恢复的异常。

一个可恢复的异常是通过 KeeperException.ConnectionLossException 来表示的，它意味着已经丢失了与 Zookeeper 的连接。ZooKeeper 会尝试重新连接，并且在大多数情况下重新连接会成功，并确保会话是完整的。但是 Zookeeper 不能判断与 KeeperException.ConnectionLossException 异常相关的操作是否成功执行。这就是部分失败的一个例子。有必要解决这种不确定性，并且根据应用的情况来采取适当的操作。

需要对幂等操作和非幂等操作进行区分。幂等操作是指那些一次或多次执行都会产生相同结果的操作，例如读请求或无条件执行的 setData 操作。对于幂等操作，只需要简单地进行重试即可。对于非幕等操作，就不能盲目地进行重试，因为它们多次执行的结果与一次执行是完全不同的。程序可以通过在 znode 的路径和它的数据中编码信息来检测是否非幂等操作的更新已经完成。

- **不可恢复的异常**

在某些情况下，Zookeeper 会话会失效，也许因为超时或因为会话被关闭（两种情况下都会收到 KeeperException.SessionExpiredException 异常），或因为

身份验证失败(KeeperException.AuthFailedException 异常)。无论上述哪种情况，所有与会话相关联的短暂 znode 都将丢失，因此应用程序需要在重新连接到 Zookeeper 之前重建它的状态。

分布式锁能够在一组进程之间提供互斥机制，使得在任何时刻只有一个进程可以持有锁。分布式锁可以用于在大型分布式系统中实现领导者选举，在任何时间点，持有锁的那个进程就是系统的领导者。

不能将 Zookeeper 自己的领导者选举和使用 Zookeeper 基本操作实现的一般的领导者选举服务混为一谈（事实上，Zookeeper 中包含有一个领导者选举服务的实现)。Zookeeper 自己的领导者选举机制是不对外公开的，接下来所描述的一般领导者选举服务则不同,它是为那些需要所有进程与主进程保持一致的分布式系统所设计的。

为了使用 Zookeeper 来实现分布式锁服务，可以使用顺序 znode 来为那些竞争锁的进程强制排序。

思路很简单：

- 首先指定一个作为锁的 znode，通常用它来描述被锁定的实体，称为 /leader
- 希望获得锁的客户端创建一些短暂顺序 znode,作为锁 znode 的子节点。
- 在任何时间点，顺序号最小的客户端将持有锁。例如，有两个客户端差不多同时创建 znode，分别为/Ieader/Iock-1 和 / Ieader/Iock-2，那么创建 / leader/lock-1 的客户端将会持有锁，因为它的 znode 顺序号最小。
- Zookeeper 服务是顺序的仲裁者，因为它负责分配顺序号。

通过删除 znode/leader/lock-1 即可简单地将锁释放,如果客户端进程死亡，对应的短暂 znode 也会被删除。接下来，创建 / leader/lock-2 的客户端将持有锁，因为它的顺序号紧跟前一个。通过创建一个关于 znode 删除的观察，可以使客户端在获得锁时得到通知。

以下是申请锁的伪代码：

1.在锁 znode 下创建一个名为 Iock-的短暂顺序 znode，并几记住它的实际路径名（create 操作的返回值)。

2.查询锁 znode 的子节点并且设置一个观察。

3.如果步骤 1 中所创建的 znode 在步骤 2 返回的所有子节点中具有最小的顺序号，则获取到锁，退出。

4.等待步骤 2 中所设观察的通知，转到步骤 2。

虽然获取锁算法是正确的，但还是存在一些问题。第一个问题是这种实现会受到“羊群效应”的影响：

在有成百上千客户端的情况，所有的客户端都在尝试获得锁，所以每个客户

端都会在锁 znode 上设置一个观察，用于捕捉子节点的变化。每次锁被释放或一个新进程开始申请锁的时候，观察都会被触发并且每个客户端都会收到一个通知。

“羊群效应”就是指这种大量客户端收到同一事件的通知，但实际上只有很少一部分需要处理这一事件。在这种情况下，只有一个客户端会成功地获取锁，但是维护的过程以及向所有客户端发送观察事件会产生峰值流量，这会对 Zookeeper 服务器造成压力。

为了避免出现羊群效应，需要优化发送通知的条件。关键在于仅当前一个顺序号的子节点消失时才需要通知下一个客户端，而不是删除（或创建）任何子节点时都进行通知。在示例中，如果客户端创建了 znode/Ieader/lock-1、/leader/Iock-2 和 / leader/lock3，那么只有当 / leader/lock-2 消失时才需要通知 / Ieader/lock-3 对应的客户端；/leader/lock-1 消失或有新的 znode /leader/Iock-4 加入时，不需要通知该客户端。

正确地实现一个分布式锁是一件棘手的事，因为很难对所有类型的故障都进行正确的解释处理。Zookeeper 带有一个 Java 语言编写的生产级别的锁实现，名为 WriteLock，客户端可以很方便地使用它。

第6节 生产环境中的 Zookeeper

Zookeeper 服务器的集合体中，每个服务器都有一个数值型的 ID，服务器 ID 在集合体中是唯一的，并且取值范围在 1 到 255 之间。

可以通过一个名为 myid 的纯文本文件设定服务器的 ID，这个文件保存在 dataDir 参数所指定的目录中。

还需要将集合体中其他服务器的 ID 和网络位置告诉所有的服务器。

在 ZooKeeper 的配置文件中必须为每台服务器添加下面这行配置：

```
server.n=hostname:port:port
```

n 是服务器的 ID。这里有 2 个端口设置：

- 第一个是跟随者用来连接领导者的端口
- 第二个端口被用干领导者选举

这里有一个包含三台机器的复制模式下 Zookeeper 集合体的配置例子：

```
tickTime=2000
dataDir=/usr/pig/data
dataLogDir=/usr/pig/logdata
clientPort=2181
initLimit=5
```

```
syncLimit=2
server.1=192.168.0.1:2888:3888
server.2=192.168.0.2:2888:3888
server.3=192.168.0.3:2888:3888
```

利用该配置文件启动服务后，服务器在 3 个端口上进行监听：

- 2181 端口被用于客户端连接
- 对于领导者来说，2888 端口被用于跟随者连接
- 3888 端口被用于领导者选举阶段的其他服务器连接

当一个 ZooKeeper 服务器启动时，它读取 myid 文件用于确定自己的服务器 ID，然后通过读取配置文件来确定应当在哪个端口进行监听，同时确定集合体中其他服务器的网络地址。

连接到这个 Zookeeper 集合体的客户端在 Zookeeper 对象的构造方法中应当使用 192.168.0.1:2181、192.168.0.2:2181 和 192.168.0.3:2181 作为主机字符串。

在复制模式下，有两个额外的强制参数：initLimit 和 syncLimit，两者都是以滴答参数的倍数进行度量。

- **initLimit 参数**

设定了所有跟随者与领导者进行连接并同步的时间范围。如果在设定的时间段内，半数以上的跟随者未能完成同步，领导者便会宣布放弃领导地位，然后进行另外一次领导者选举。如果这种情况经常发生（可以通过日志中的记录发现这种情况），则表明设定的值太小。

- **syncLimit 参数**

设定了允许一个跟随者与领导者进行同步的时间。如果在设定的时间段内，一个跟随者未能完成同步，会自己重启。所有关联到跟随者的客户端将连接到另一个跟随者。

第10章　数据导入与导出

第1节 Sqoop 入门介绍

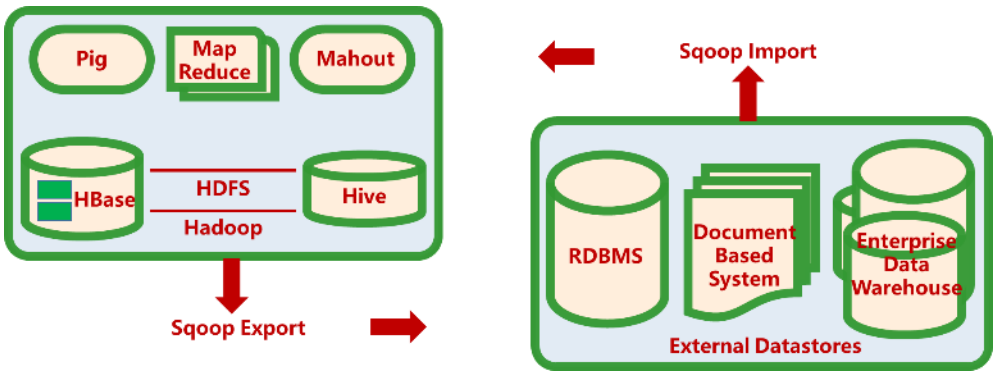
Sqoop(发音：skup)是一款开源的工具，主要用于在 Hadoop(Hive)与传统的数据库(mysql、postgresql...)间进行数据的传递，可以将一个关系型数据库(例如 ： MySQL ，Oracle ，Postgres 等）中的数据导进到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导进到关系型数据库中。

Hadoop 平台的最大优势在于它支持使用不同形式的数据。

HDFS 能够可靠地存储日志和来自不同渠道的其他数据，MapReduce 程序能够解析多种“即席”(ad hoc）数据格式，抽取相关信息并将多个数据集组合成非常有用的结果。但是为了能够和 HDFS 之外的数据存储库进行交互，MapReduce 程序需要使用外部 API 来访问数据。

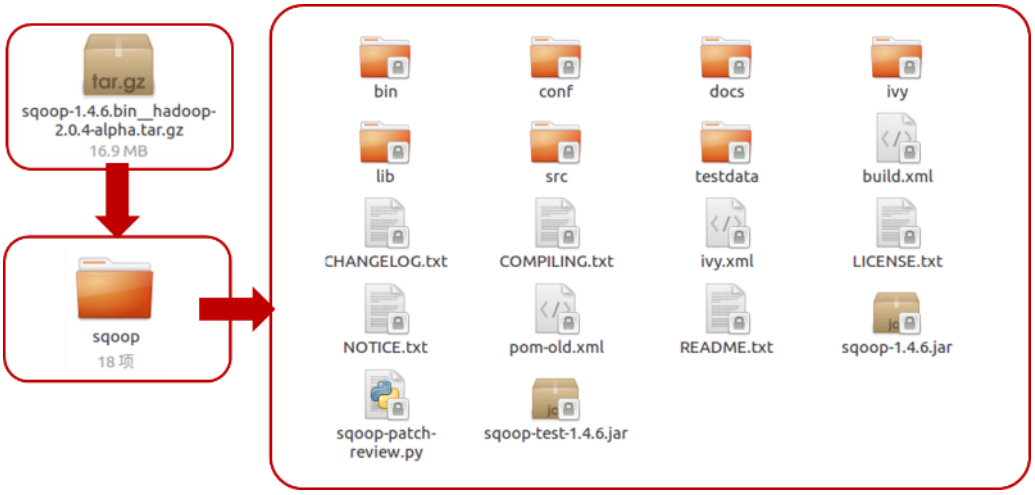
通常，一个组织中有价值的数据都存储在关系型数据库系统等结构化存储器中。Apache Sqoop 允许用户将数据从结构化存储器抽取到 Hadoop 中，用于进一步的处理。抽取出的数据可以被 MapReduce 程序使用，也可以被其他类似于 Hive 的工具使用。(甚至可以使用 Sqoop 将数据从数据库转移到 HBase)

一旦生成最终的分析结果，Sqoop 便可以将这些结果导回数据存储器，供其他客户端使用。Sqoop 解决的问题如下所示：



第2节 Sqoop 的基本使用

Sqoop 的安装仍然满足解压后配置环境变量的模式。首先从官方网站下载发行包，解压后到安装的目标路径：



然后配置对应的环境变量：

```
export SQOOP_HOME=/usr/sqoop

export PATH=$PATH:$SQOOP_HOME/bin
```

接下来就可以启动 Sqoop 工具了，当然不带参数运行 Sqoop 是没有什么意义的：

```
hadoop@eric-VirtualBox:/usr/sqoop/bin$ sqoop
Warning: /usr/sqoop/../hcatalog does not exist! HCatalog jobs will fail.
Please set $HCAT_HOME to the root of your HCatalog installation.
Warning: /usr/sqoop/../accumulo does not exist! Accumulo imports will fail.
Please set $ACCUMULO_HOME to the root of your Accumulo installation.
Try 'sqoop help' for usage.
```

Sqoop 被组织成一组工具或命令。不选择工具，Sqoop 便无所适从。help 是其中一个工具的名称，它能够打印出可用工具的列表，如下所示：

```
hadoop@eric-VirtualBox:/usr/sqoop/bin$ sqoop help
Warning: /usr/sqoop/../hcatalog does not exist! HCatalog jobs will fail.
Please set $HCAT_HOME to the root of your HCatalog installation.
Warning: /usr/sqoop/../accumulo does not exist! Accumulo imports will fail.
Please set $ACCUMULO_HOME to the root of your Accumulo installation.
16/07/14 15:56:43 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen            Generate code to interact with database records
  create-hive-table  Import a table definition into Hive
  eval               Evaluate a SQL statement and display the results
  export             Export an HDFS directory to a database table
```

通过将特定工具的名称作为参数，help 还能够提供该工具的使用说明：

```
hadoop@eric-VirtualBox:/usr/sqoop$ sqoop help import | more
Warning: /usr/sqoop/../hcatalog does not exist! HCatalog jobs will fail.
Please set $HCAT_HOME to the root of your HCatalog installation.
Warning: /usr/sqoop/../accumulo does not exist! Accumulo imports will fail.
Please set $ACCUMULO_HOME to the root of your Accumulo installation.
16/07/14 16:04:58 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]

Common arguments:
   --connect <jdbc-uri>                         Specify JDBC connect
                                                string
   --connection-manager <class-name>            Specify connection manager
                                                class name
   --connection-param-file <properties-file>    Specify connection
                                                parameters file
   --driver <class-name>                        Manually specify JDBC
                                                driver class to use
   --hadoop-home <hdir>                         Override
                                                $HADOOP_MAPRED_HOME_ARG
   --hadoop-mapred-home <dir>                   Override
                                                $HADOOP_MAPRED_HOME_ARG
   --help                                       Print usage instructions
-P                                              Read password from console
   --password <password>                        Set authentication
```

运行 Sqoop 工具的另外一种方法是使用与之对应的特定脚本。这样的脚本一般被命名为 sqoop-toolname，例如：

```
sqoop-help

sqoop-import
```

运行这两个脚本与运行 sqoop help 或 sqoop import 命令是一样的。

Sqoop 拥有一个可扩展的框架，使得它能够从（向）任何支持批量数据传输的外部存储系统导入（导出）数据。

一个 Sqoop 连接器（connector）就是这个框架下的一个模块化组件，用于支持 Sqoop 的导入和导出操作。Sqoop 附带的连接器能够支持大多数常用的关系数据库系统，包括 MySQL、PostgreSQL、Oracle、SQLserver 和 DB2。同时还有一个通用的 JDBC 连接器，用于连接支持 JDBC 协议的数据库。

Sqoop 所提供的 MySQL 和 PostgreSQL 连接器都是经过优化的，通过使用数据库特定的 API 来提供高效率的批量数据传输。

除了内置的 Sqoop 连接器外，还有很多针对各种数据存储器的第三方连接器可用，能够支持对企业级数据仓库（包括 Netezza、Teradata 和 Oracle）和 NoSQL 存储器（例如 Couchbase）的连接。

这些连接器必须另外单独下载，可以根据连接器所附带的安装说明将其添加到已有的 Sqoop 安装中。

第3节 数据导入的示例

在安装了 Sqoop 之后，可以用它将数据导入 Hadoop。接下来所有示例中都使用支持很多平台的易用数据库系统 MySQL 作为外部数据源。首先在 MySQL 中构建示例数据库：

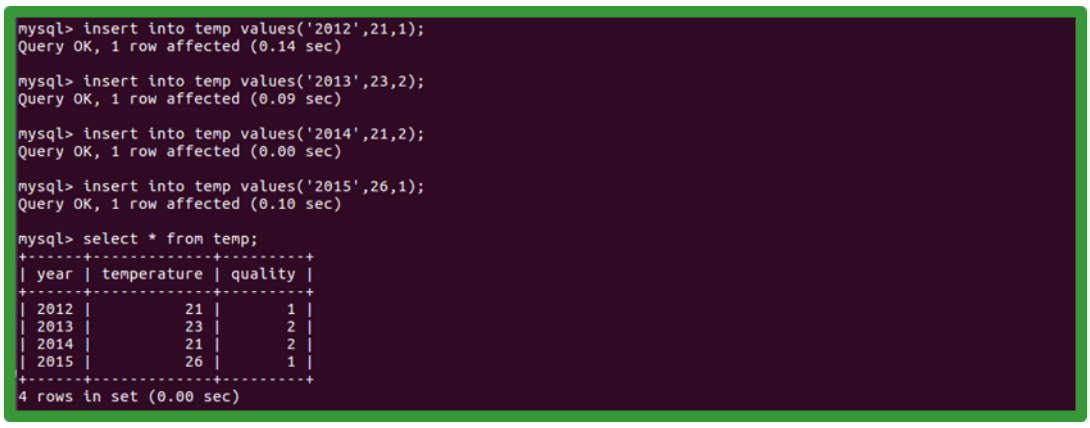
```
create database sqooptest;
```

在该示例数据库中创建表：

```
use sqooptest
create table temp(year varchar(30),temperature int,quality int);
```

向表中写入测试数据：

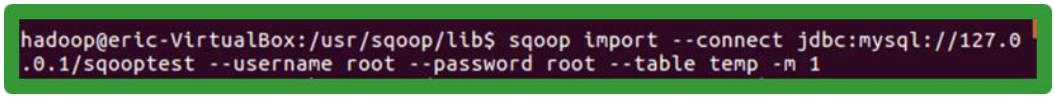
```
insert into temp values('2012',21,,1);
insert into temp values('2013',23,,2);
insert into temp values('2014',21,,2);
insert into temp values('2015',216,,1);
```



将 MySQL JDBC 驱动包拷贝到$SQOOP_HOME/lib。

将数据导入 HDFS：

```
sqoop import --connect jdbc:mysql://服务器地址[:端口号]/数据库名 --username 数据库用户名 --password 数据库密码 --table 表名 -m Map 任务数量
```



Sqoop 的 import 工具会运行一个 MapReduce 作业，该作业会连接 MySQL 数据库并读取表中的数据。默认情况下，该作业会并行使用 4 个 map 任务来加速导入过程。每个任务都会将其所导入的数据写到一个单独的文件，但所有 4 个文件都位于同一个目录中。在本例中，由于知道只有 4 行可以导入的数据，因此指定 Sqoop 只使用一个 map 任务（-m 1），这样就只得到一个保存在 HDFS 中的文件。

上述导入示例命令中的参数含义解释如下：

组成	描述
sqoop import	运行 Sqoop 并执行导入操作
--connect	提供连接器
--username	连接数据库的用户名
--password	连接数据库的密码
--table	要导出的表
--fields-terminated-by '\t'	导出文件的分隔符（不提供的话默认为逗号分割）
-m	Map 任务的个数
--hive-import	把数据复制到 hive 空间中。如果不使用该选项，意味着复制到 HDFS 中

查看数据导入结果：

```
hadoop@eric-VirtualBox:/usr/sqoop/lib$ hadoop fs -cat /user/hadoop/temp/part-m-00000
2012,21,1
2013,23,2
2014,21,2
2015,26,1
hadoop@eric-VirtualBox:/usr/sqoop/lib$
```

在本例中使用了连接字符串 jdbc:mysql：//127.0.0.1/sqooptest，表明需要从本地机器上的数据库中读取数据。如果使用了分布式 Hadoop 集群，则在连接字符串中不能使用 127.0.0.1 或 Localhost；否则，与数据库不在同一台机器上运行的 map 任务都将无法连接到数据库。即使是从数据库服务器所在主机上运行 Sqoop，也需要为数据库服务器指定完整的主机名。

Sqoop 可以将数据导入成几种不同的格式。文本文件（默认）是一种直接可读的数据表示形式，并且是平台独立和最简单的数据格式。但是，文本文件不能保存二进制字段（例如数据库中类型为 VARBINARY 的列），并且在区分 null 值和字符串 null 时可能会出现问题（尽管使用--null-string 选项可以控制空值的表示方式）。

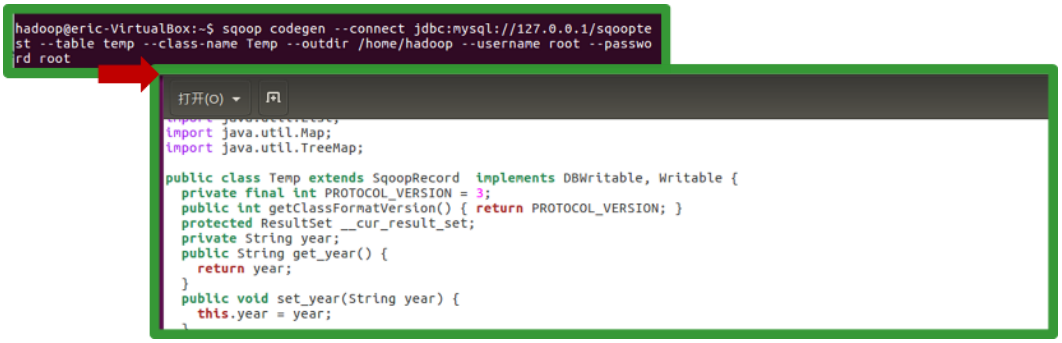
为了处理这些情况，应该使用 Sqoop 的 SequenceFile 格式或 Avro 格式。

Avro 和 SequenceFile 格式的文件能够为导入的数据提供最精确的表示方式，同时还允许对数据进行压缩，并支持 MapReduce 并行处理同一文件的不同部分。

Sqoop 的当前版本还不能将 Avro 或 SequenceFile 文件加载到 Hive 中（可以手动完成）。SequenceFile 文件格式的最后一个缺点是它只支持 Java 语言，而 Avro 数据文件却可以被很多种语言处理。

可以利用 sqoop codegen 工具为导入的数据生成辅助访问代码：

```
sqoop codegen --connect jdbc:mysql://服务器地址[:端口号]/数据库名 --table 表名  --class-name 生成的类名 --outdir 输出目录 --username 数据库用户名 --password 数据库密码
```



生成的类中能够保存从被导入表中取出的记录。该类可以在 MapReduce 中使用这条记录，也可以将这条记录保存在 HDFS 中的一个 SequenceFile 文件中。

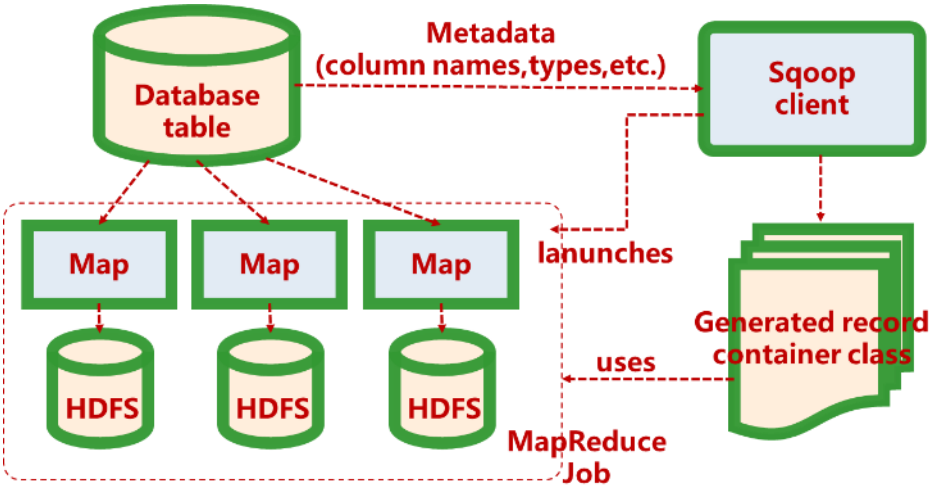
在导入过程中，由 Sqoop 生成的类会将每一条被导入的行保存在 SequenccFile 文件的键 / 值对格式中值的位置。

codegen 工具只是简单地生成代码，它不执行完整的导入操作。

如果计划使用导入到 SequenceFile 文件中的记录，将不可避免地用到生成的类(对 SequenceFile 文件中的数据进行反序列化)。在后续课程中将看到 Sqoop 生成的代码有助于解决数据处理过程中的一些繁琐问题。

第4节 深入 Sqoop 的导入与导出

Sqoop 是通过一个 MapReduce 作业从数据库中导入一个表，这个作业从表中抽取一行行记录，然后将记录写入 HDFS：



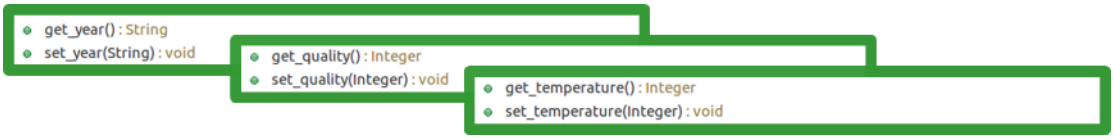
和 Hadoop 一样，Sqoop 是用 Java 语言编写的。Java 提供了一个称为 JDBC(Java Database Connectivity) 的 API，应用程序可以使用这个 API 来访问存储在 RDBMS 中的数据，同时可以检查数据类型。

大多数数据库厂商都提供了 JDBC 驱动程序，其中实现了 JDBC API 并包含用于连接其数据库服务器的必要代码。

根据用于访问数据库的连接字符串中的URL,Sqoop尝试预测应该加载哪个驱动程序，必须事先下载所需的JDBC驱动并且将其安装在Sqoop客户端。在Sqoop无法判定使用哪个JDBC驱动程序时，用户可以明确指定如何将JDBC驱动程序加载到Sqoop，这样Sqoop就能够支持大多数的数据库平台。

在导入开始之前，Sqoop使用JDBC来检查将要导入的表。它检索出表中所有的列以及列的SQL数据类型。这些SQL类型（VARCHAR、INTEGER等）被映射到Java数据类型（String、Integer等），在MapReduce应用中将使用这些对应的Java类型来保存字段的值。

Sqoop的代码生成器使用这些信息来创建对应表的类，用于保存从表中抽取的记录。例如，之前提到的Temp类包含下列方法，这些方法用于从抽取的记录中检索所有的列：



对于导入来说，更关键的是DBWritable接口的序列化方法，这些方法能使Temp类和JDBC进行交互：

```
public void readFields(ResultSet __dbResults) throws SQLException
public void write(PreparedStatement __dbStmt) throws SQLException
```

JDBC的Resultset接口提供了一个用于从查询结果中检索记录的游标,这里的readFields()方法将用ResultSet中一行数据的列来填充Temp对象的字段。write()方法允许Sqoop将新的Temp行插入表，这个过程称为“导出”

Sqoop启动的MapReduce作业用到一个InputFormat，它可以通过JDBC从一个数据库表中读取部分内容。Hadoop提供的DataDBInputFormat能够为几个map任务对查询结果进行划分。使用一个简单的查询通常就可以读取一张表的内容，例如：

```
SELECT col1,col2,col3，… FROM tablename
```

但是，为了获得更好的导入性能，经常将这样的查询划分到多个节点上执行。查询是根据一个“划分列” 来进行划分的。根据表的元数据，Sqoop会选择一个合适的列作为划分列(通常是表的主键)。主键列中的最小值和最大值会被读出，与目标任务数一起用来确定每个map任务要执行的查询。

例如，假设表中有100,000条记录，其id列的值为0-99,999。

在导入这张表时，Sqoop会判断出id是表的主键列。启动MapReduce。作业时，用来执行导入的DataDrivenDBInputFormat便会发出一条类似于SELECT MIN(id),MAX(id) FROM table的查询语句。检索出的数据将用于对整个数据集进行划分。

假设指定并行运行 5 个 map 任务（使用-m 5），这样便可以确定每个 map 任务要执行的查询数据条件。

划分列的选择是影响并行执行效率的重要因素。如果 id 列的值不是均匀分布的（也许在 id 值 50,000 到 75,000 的范围内没有记录），那么有一部分 map 任务可能只有很少或没有工作要做，而其他任务则有很多工作要做。

在运行一个导入作业时，用户可以指定一个列作为划分列，从而调整作业的划分使其符合数据的真实分布。

如果使用-m 1 参数来让一个任务执行导入作业，就不再需要这个划分过程。

在生成反序列化代码和配置 InputFormat 之后，Sqoop 将作业发送到 MapReduce 集群。map 任务执行查询并且将 ResultSet 中的数据反序列化到生成类的实例，这些数据要么被直接保存在 SequenceFile 文件中，要么在写到 HDFS 之前被转换成分隔的文本。

Sqoop 不需要每次都导入整张表。可以指定仅导入表的部分列。用户也可以在导入中通过—where 参数加入 WHERE 子句，以此来限定需要导入的记录。例如，如果上个月已经将 id 为 0-99,999 的记录导入，而本月供应商的产品目录中增加了 1000 种新部件，那么导入时在查询中加入子句 WHERE id>=100000，就可以实现只导入所有新增的记录。

用户提供的 WHERE 子句会在任务分解之前执行，并且被下推至每个任务所执行的查询中。

在向 HDFS 导入数据时，重要的是要确保访问的是数据源的一致性快照。从一个数据库中并行读取数据的 Map 任务分别运行在不同的进程中，因此它们不可能共享同一个数据库事务。保证一致性的最好方法就是在导入时不允许运行任何对表中现有数据进行更新的进程。

Sqoop 的架构支持它在多种可用的导入方法中进行选择，而多数数据库都使用上述基于 DataDrivenDBInputFormat 的方法。

一些数据库提供了能够快速抽取数据的特定工具，例如 MySQL 的 mysqldump 能够以大于 JDBC 的吞吐率从表中读取数据。

在 Sqoop 的文档中将这种使用外部工具的方法称为“直接模式” 。由于直接模式并不像 JDBC 方法那样通用（例如，MySQL 的直接模式不能处理大对象数据类型为 CLOB 或 BLOB 的列，Sqoop 需要使用 JDBC 专用的 API 将这些列载入 HDFS）所以使用直接模式导入时必须由用户明确地启动（通过-direct 参数）。

对于那些提供了此类特定工具的数据库，Sqoop 使用这些工具能够得到很好的效果。采用直接模式从 MySQL 中导入数据通常比基于 JDBC 的导入更加高效（就 map 任务和所需时间而言）。

Sqoop 仍然并行启动多个 map 任务，接着这些任务将分别创建 mysqldump 程序的实例并且读取它们的运行结果。

Sqoop 也支持采用直接模式从 PostgreSQL 中导入数据。

即使是用直接模式来访问数据库的内容，元数据的查询仍然是通过 JDBC 来实现的。

一旦数据被导入 HDFS，就可以供定制的 MapReduce 程序使用。导入的文本格式数据可以供 HadoopStreaming 中的脚本或以 TextlnputFormat 为默认格式运行的 MapReduce 作业使用。

为了使用导入记录的个别字段，必须对字段分隔符（以及转义 / 包围字符）进行解析，抽出字段的值并转换为相应的数据类型。例如，在文本文件中，气候质量被表示成字符串“1”，但必须被解析为 Java 的 Integer 或 int 类型的变量。

Sqoop 生成的表类能够自动完成这个过程，使开发人员可以将精力集中在真正要运行的 MapReduce 作业上。

每个自动生成的类都有几个名为 parse() 的重载方法，这些方法可以对表示为 Text、Charsequence、char[]或其他常见类型的数据进行操作。

将 Sqoop 生成的代码拷贝到 Eclipse 项目中，并构建 MapReduce 应用 MaxTemp 获取最高温度，Map 函数参考实现如下：

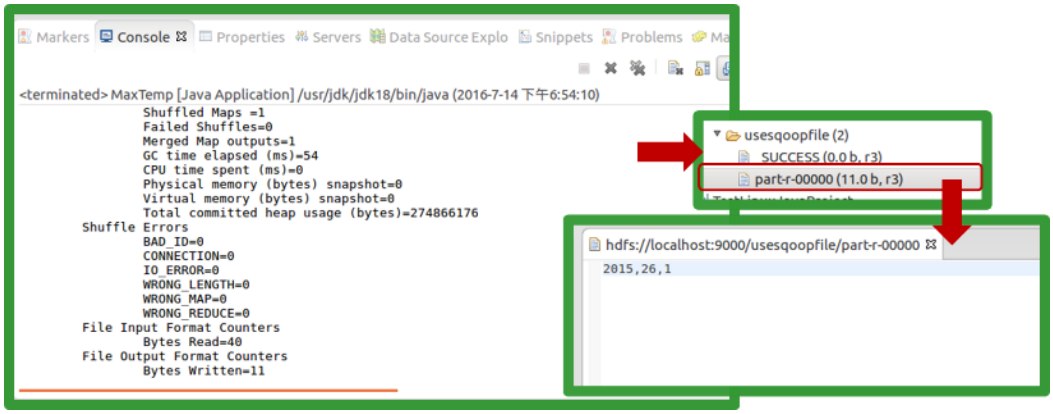
```
    public static class MaxWidgetMapper extends
            Mapper<LongWritable, Text, LongWritable, Temp> {
        private Temp maxTemp = null;
        public void map(LongWritable k, Text v, Context context) {
            Temp temp = new Temp();
            try {
                temp.parse(v); // Auto-generated: parse all fields
from text.
            } catch (ParseError pe) {
                return;
            }
            Integer temperature = temp.get_temperature();
            if (null == temperature) {
                return;
            } else {
                if (maxTemp == null
                        || temperature.intValue() >
maxTemp.get_temperature()
                                .intValue()) {
                    maxTemp = temp;
                }
            }
        }
        //本Map任务所有数据处理完成后执行一次
        public void cleanup(Context context) throws IOException,
                InterruptedException {
            if (null != maxTemp) {
```

```
            context.write(new LongWritable(0), maxTemp);
        }
    }

}
```

对应的 Reduce 任务参考实现如下：

```
public static class MaxWidgetReducer extends
        Reducer<LongWritable, Temp, Temp, NullWritable> {

    public void reduce(LongWritable k, Iterable<Temp> vals,
Context context)
            throws IOException, InterruptedException {
        Temp maxTemp = null;

        for (Temp t : vals) {
            if (maxTemp == null
                    || t.get_temperature().intValue() > maxTemp
                            .get_temperature().intValue()) {
                try {
                    maxTemp = (Temp) t.clone();
                } catch (CloneNotSupportedException cnse) {
                    // Shouldn't happen; Sqoop-generated classes
support
                    // clone().
                    throw new IOException(cnse);
                }
            }
        }

        if (null != maxTemp) {
            context.write(maxTemp, NullWritable.get());
        }
    }
}
```

注意在任务调度代码中将任务的输入指定为刚才由 Sqoop 从 MySQL 中导入得到的文件，程序执行结果如下：



完整代码实现参考本教材的随书代码。

对于很多类型的分析任务来说，使用类似于 Hive 的系统来处理关系操作有利于加快分析任务的开发。特别是对于那些来自于关系数据源的数据，使用 Hive 是非常有帮助的。Hive 和 Sqoop 共同构成了一个强大的服务于分析任务的工具链。

将 MySQL 的表导入 Hive：

```
sqoop import --connect jdbc:mysql://服务器地址[:端口号]/数据库名 --username 数据库用户名 --password 数据库密码 --table 表名 --hive-import -m Map 任务数量 --fields-terminated-by ','
```

```
hadoop@eric-VirtualBox:~$ sqoop import --connect jdbc:mysql://127.0.0.1/sqooptest --username root --password root --table temp --hive-import -m 1 --fields-terminated-by ','
```

执行结果如下：

```
hive> show tables;
OK
bucketed_uses
logs
records
sales
temp
testmysql
things
uses
view_things
view_things1
w_sales
Time taken: 0.577 seconds, Fetched: 11 row(s)
hive> SELECT * FROM temp
    > ;
OK
2012    21      1
2013    23      2
2014    21      2
2015    26      1
Time taken: 3.624 seconds, Fetched: 4 row(s)
hive>
```

成功创建表

数据成功导入

导入 hive 已经创建好的表中：

```
sqoop import --connect jdbc:mysql://ip/db_name --username user_name
--table table_name  --hive-import -m 5 --hive-table hive_table_name
(--hive-partition-key     partition_name     --hive-partition-value
partititon_value);
```

使用 query 导入 hive 表:

```
sqoop import --connect jdbc:mysql://ip/db_name --username user_name
--query "select ,* from retail_tb_order where \$CONDITIONS"  --hive-
import  -m  5  --hive-table  hive_table_name   (--hive-partition-key
partition_name --hive-partition-value partititon_value);
```

注意：$CONDITIONS 条件必须有，query 子句若用双引号，则$CONDITIONS 需要使用\转义，若使用单引号，则不需要转义。

在为一个特定的已导入数据集创建相应的 Hive 表定义时，我们需要指定该数据集所使用的分隔符。否则，Sqoop 将允许 Hive 使用它自己的默认分隔符（与 Sqoop 的默认分隔符不同）。

Hive 的数据类型不如大多数 SQL 系统的丰富。很多 SQL 类型在 Hive 中都没有直接对应的类型。当 Sqoop 为导入操作生成 Hive 表定义时，它会为数据列选择最合适的 Hive 类型，这样可能会导致数据精度的下降。一旦出现这种情况，Sqoop 就会提供一条警告信息。

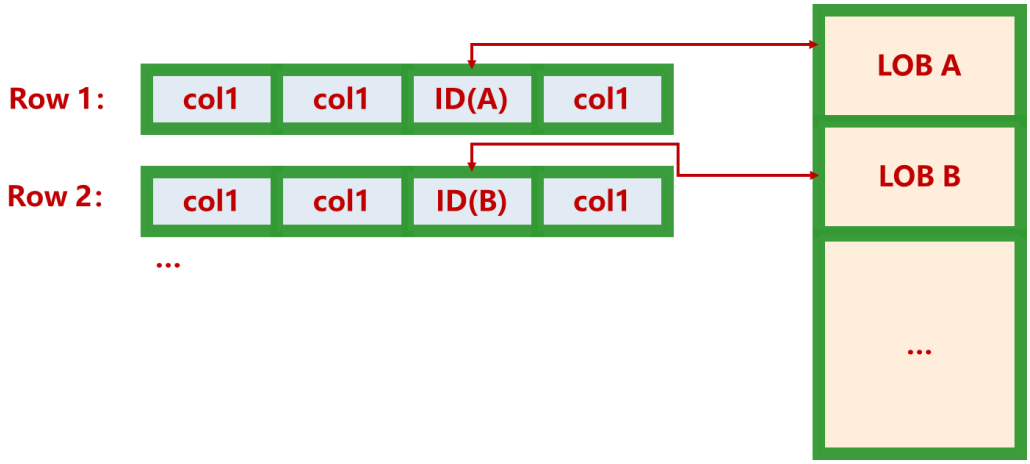
Sqoop 能够根据一个关系数据源中的表来生成一个 Hive 表而不导入数据，共用户自行加载数据文件：

```
sqoop  create-hive-table  --connect  jdbc:mysql://ip/databasename  --
table tablename --fields-terminated-by ','
```

很多数据库都支持在一个字段中保存大量的数据并根据数据是文本还是二进制类型，通常将其保存在表中 CLOB 或 BLOB 类型的列中。

数据库一般会对这些大对象进行特殊处理。大多数的表在磁盘上的物理存储都通过行扫描来确定哪些行匹配特定的查询条件时，需要从磁盘上读出每一行的所有列。如果也是以这种方式内联存储大对象，它们会严重影响扫描的性能。

数据库一般将大对象与它们的行分开存储，在访问大对象时，需要通过行中包含的引用来打开它:



在数据库中使用大对象的困难表明，像 Hadoop 这样的系统更适合于处理大型的、复杂的数据对象，也是存储此类信息的理想选择。Sqoop 能够从表中抽取大对象数据，并且将它们保存在 HDFS 中供进一步处理。同在数据库中一样，MapReduce 在将每条记录传递给 mapper 之前一般要对其进行物化。如果单条记录真的很大，物化操作将非常低效。

Sqoop 所导入的记录在磁盘上的存储格式与数据库的内部数据结构非常相似：

将每条记录的所有字段放在一起组成的一个记录数组。在导入的记录上运行一个 MapReduce 程序时，每个 map 任务必须将所读记录的所有字段完全物化。如果 MapReduce 程序所处理的输入记录中仅有很小一部分的大对象字段的内容，那么将所有记录完全物化将导致程序效率低下。此外，从大对象的大小来看，在内存中进行完全物化也许是无法实现的。

为了克服这些困难，Sqoop 将导入的大对象数据存储在 LobFile 格式的单独文件中。LobFile 格式能够存储非常大的单条记录（使用了 64 位的地址空间），每条记录保存一个大对象。LobFile 格式允许客户端持有对记录的引用，而不访问记录内容；对记录的访问是通过 java.io.InputStream（用于二进制对象）或 java.io.Reader（用于字符对象）来实现的。

在导入一条记录时，所有的“正常”字段会在一个文本文件中一起被物化，同时还生成一个指向保存 CLOB 或 BLOB 列的 LobFIle 文件的引用。

存在大对象的表导入的记录可能像下面这样：

```
2,100,Beijing,externalLob(lf,lobfile0,100,5011714)
```

externalLob（…）是对外部存储大对象的一个引用，这个大对象以 LobFile 格式（lf）存储在名为 lobfile0 的文件中，同时还给出了该对象在文件中的字节位移和长度。

在使用这条记录时，temp.getImage_() 方法会返回一个类型为 BlobRef 的对象，用于引用 image 列，但这个对象并不真正包含记录的内容。BlobRef.getDataStream() 方法实际会打开 LobFile 文件并返回一个 InputStream，用于访问 image 字段的内容。

在使用一个 MapReduce 作业来处理许多 temp 记录时，可能只需要访问少数几条记录的 image 字段。单个图片数据可能有几兆大小或更大，使用这种方式时，只需要承担访问所需大对象的 I/O 开销。

在一个 map 任务中，BlobRef 和 ClobRef 类会缓存对底层 LobFile 文件的引用。如果你访问几个顺序排列记录的 image 字段，就可以利用现有文件指针来定位下一条记录。

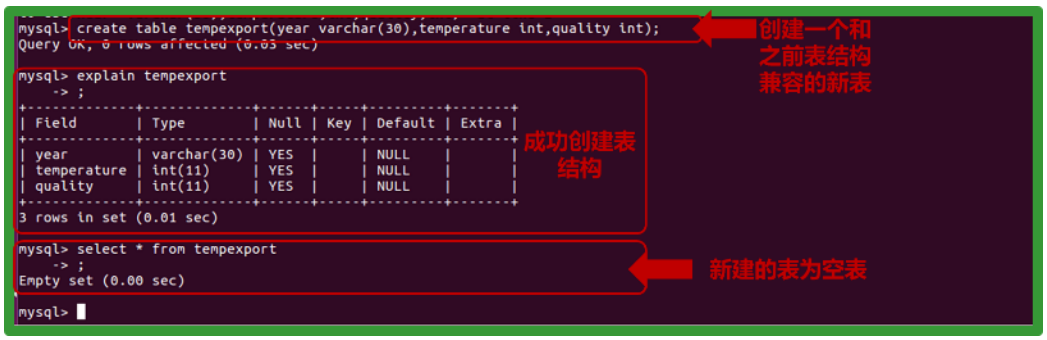
和导入操作相反，“导出”(export）是将 HDFS 作为数据源，而将一个远程数据库作为目标。之前曾经导入了一些数据并且使用 Hive 对数据进行了分析。接下来可以将分析的结果导出到一个数据库中，供其他工具使用。

将一张表从 HDFS 导出到数据库时，必须在数据库中创建一张用于接收数据的目标表。

虽然 Sqoop 可以推断出哪个 Java 类型适合存储 SQL 数据类型，但反过来却是行不通的（例如，有几种 SQL 列的定义都可以用来存储 Java 的 String 类型，如 CHAR(64)、VARCHAR(200）或其他一些类似定义。因此，必须由用户来确定哪些类型是最合适的。

将之前的 temp 表重新导入到一个新的 MySQL 表中，首先创建新的兼容的 MySQL 表：

```
use sqooptest

create table tempexport(year varchar(30),temperature int,quality int);
```



执行导出操作：

```
sqoop export --connect jdbc:mysql://服务器地址[:端口号]/数据库名 -m Map任务数量 --table 表名 --export-dir 要导出的数据文件路径 --input-fields-terminated-by ',' --username 数据库用户名 --password 数据库密码
```

Sqoop 导出功能的架构与其导入功能的非常相似。在执行导出操作之前，Sqoop 会根据数据库连接字符串来选择一个导出方法。对于大多数系统来说，Sqoop 都会选择 JDBC，然后，Sqoop 会根据目标表的定义生成一个 Java 类，这个生成的类能够从文本文件中解析出记录，并能够向表中插入类型合适的值，接着会启动一个 MapReduce 作业，从 HDFS 中读取源数据文件，使用生成的类解析出记录，并且执行选定的导出方法。

基于 JDBC 的导出方法会产生一批 INSERT 语句，每条语句会向目标表中插入多条记录。在大部分的数据库系统中，通过一条语句插入多条记录的执行效率要高于多次执行插人单条记录的 INSERT 语句。

多个独立的线程被用干从 HDFS 读取数据并与数据库进行通信，以确保涉及不同系统的 I/O 操作能够尽量重叠执行。

对于 MySQL 数据库来说，Sqoop 可以采取使用 mysqlimport 的直接模式方法。每个 map 任务会生成一个 mysqlimport 进程，该进程通过本地文件系统上的一个命名 FIFO 通道进行通信，数据通过这个 FIFO 通道流入 mysqlimport，然后再被写入数据库。

虽然从 HDFS 读取数据的 MapReduce 作业大多根据所处理文件的数量和大小来选择并行度（map 任务的数量），但 Sqoop 的导出工具允许用户明确设定任务的数量。

由于导出性能会受并行的数据库写入线程数量的影响，所以 Sqoop 使用 CombineFilelnputFormat 类将输入文件分组分配给少数几个 map 任务去执行。

进程的并行特性决定了导出操作往往不是原子操作。Sqoop 会生成多个并行执行的任务，分别导出数据的一部分。这些任务的完成时间各不相同，即使在每个任务内部都使用事务，不同任务的执行结果也不可能同时提交。

此外，数据库系统经常使用固定大小的缓冲区来存储事务数据，这使得一个任务中的所有操作不可能在一个事务中完成。

Sqoop 每导入几千条记录便会执行一次提交，以确保不会出现内存不足的情况。在导出操作进行过程中，提交过的中间结果都是可见的。因此在导出过程完成前，不要启动那些要使用导出结果的应用程序，否则这些应用会看到不完整的导出结果。

为了解决这个问题，Sqoop 可以将数据先导出到一个临时阶段表中，然后在导出任务完成前（即导出操作成功执行后）在一个事务中将临时阶段表中的数据

全部移动到目标表中：

- 可以通过--staging-table 选项来指定一个临时阶段表，这个临时阶段表必须是已经存在的，并且和目标表具有相同的模式定义。
- 如果不使用--clear-staging-table 选项，则这个临时阶段表必须是空表。

之前的导出示例是从一个 Hive 表中读取源数据，该 Hive 表以分隔文本文件形式保存在 HDFS 中。

Sqoop 也可以从非 Hive 表的分隔文本文件中导出数据。例如，Sqoop 可以导出 MapReduce 作业结果的文本文件。

Sqoop 还可以将存储在 SequenceFile 中的记录导出到输出表，不过有一些限制。SequenceFile 中可以保存任意类型的记录，但由于 Sqoop 的导出工具从 SequenceFile 中读取对象后直接发送到 OutputCollector，由它将这些对象传递给数据库导出 OutputFormat，因此为了能让 Sqoop 使用，记录必须被保存在 SequenceFile 键/值对格式的“值”部分，并且必须继承抽象类 com.cloudera.sqoop.lib.SqoopRecord（像 Sqoop 生成的所有类那样）。

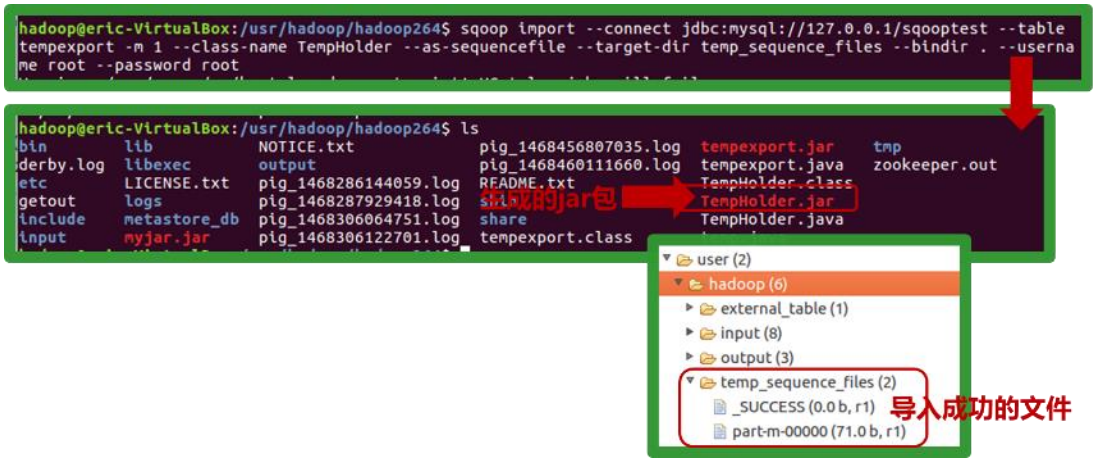
如果基于导出目标表，使用 codegen 工具（sqoop-codegen）为记录生成一个 SqoopRecord 的实现，就可以写一个 MapReduce 程序，填充这个类的实例并将它们写入 SequenceFile，接着 sqoop-export 就可以将这些 SequenceFile 文件导出到表中。

还有另外一种方法，即将数据放入 SqoopRecord 实例中，然后保存到 SequenceFile 中。

如果数据是从数据库表导入 HDFS 的，那么在经过某种形式的修改后，可以将结果保存在持有相同数据类型记录的 SequenceFIle 中。

在这种情况下，Sqoop 应当利用现有的类定义从 SequenceFIle 中读取数据而不是像导出文本记录时所做的那样，为执行导出生成一个新的（临时的）记录容器类。通过为 Sqoop 提供--class-name 和--jar-file 参数，可以禁止它生成代码，而使用现有的记录类和 jar 包。

在导出记录时，Sqoop 将使用指定 Jar 包中指定的类。执行导入：



执行导出操作：

在导入过程中，指定使用 SequenceFile 格式，并且将 JAR 文件放入本地当前目录（使用-bindir），这样将来便可以重用它，否则它将会被保存在一个临时目录中。

然后我们创建一个用于导出的目标表，该表在模式上与源数据兼容；在接下来的导出操作中，使用现有的生成代码从 SequenceFile 中读取记录并将它们写入数据库。

第11章 Flume

第1节 Flume 入门介绍

Flume 是 Cloudera 提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume 支持在日志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。

当前 Flume 有两个版本 Flume 0.9X 版本的统称 Flume-og，Flume1.X 版本的统称 Flume-ng。由于 Flume-ng 经过重大重构，与 Flume-og 有很大不同，使用时请注意区分。

- Flume-og

采用了多 Master 的方式。为了保证配置数据的一致性，Flume 引入了 ZooKeeper，用于保存配置数据，ZooKeeper 本身可保证配置数据的一致性和高可用，另外，在配置数据发生变化时，ZooKeeper 可以通知 Flume Master 节点。Flume Master 间使用 gossip 协议同步数据。

- Flume-ng

最明显的改动就是取消了集中管理配置的 Master 和 Zookeeper，变为一个纯粹的传输工具。Flume-ng 另一个主要的不同点是读入数据和写出数据现在由不同的工作线程处理（称为 Runner）。 在 Flume-og 中，读入线程同样做写出工作（除了故障重试）。如果写出慢的话（不是完全失败），它将阻塞 Flume 接收数据的能力。这种异步的设计使读入线程可以顺畅的工作而无需关注下游的任何问题。

Flume 的主要功能如下：

- **日志收集**

Flume 最早是 Cloudera 提供的日志收集系统，目前是 Apache 下的一个孵化项目，Flume 支持在日志系统中定制各类数据发送方，用于收集数据。

- **数据处理**

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 提供了从 console（控制台）、RPC（Thrift-RPC）、text（文件）、tail（UNIX tail）、syslog（syslog 日志系统，支持 TCP 和 UDP 等 2 种模式），exec（命令执行）等数据源上收集数据的能力。

Hadoop 是一个分布式系统，跟 Hadoop 配合的一般也是分布式系统，分布式系统带来的就是分布式日志，分布式日志带来两个问题：日志数量多并且日志数据量大。所以无论是采集分布式的日志还是存储海量的日志到 Hadoop，都需要一个日志收集系统，这就是 Flume。

Flume 是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。支持在日志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(比如文本、HDFS、Hbase 等)的能力 。

Flume 的数据流由事件(Event)贯穿始终。事件是 Flume 的基本数据单位，它携带日志数据(字节数组形式)并且携带有头信息，这些 Event 由 Agent 外部的 Source 生成，当 Source 捕获事件后会进行特定的格式化，然后 Source 会把事件推入(单个或多个)Channel 中。可以把 Channel 看作是一个缓冲区，它将保存事件直到 Sink 处理完该事件。Sink 负责持久化日志或者把事件推向另一个 Source。

当节点出现故障时，Flume 保证日志能够被传送到其他节点上而不会丢失。Flume 提供了三种级别的可靠性保障，从强到弱依次分别为：

end-to-end,收到数据 agent 首先将 event 写到磁盘上,当数据传送成功后，再删除；如果数据发送失败，可以重新发送。

Store on failure，这也是 scribe 采用的策略，当数据接收方 crash 时，将数据写到本地，待恢复后，继续发送。

Besteffort，数据发送到接收方后，不会进行确认。

数据的可恢复性依赖 Channel，推荐使用 FileChannel，事件持久化在本地文件系统里(但性能较差)。

第2节 Flume 核心概念

接下来介绍 Flume 的一些核心概念：

- Agent

使用 JVM 运行 Flume。每台机器运行一个 agent，但是可以在一个 agent 中包含多个 sources 和 sinks。

- Client

生产数据，运行在一个独立的线程。

- Source

从 Client 收集数据，传递给 Channel。

- Sink

从 Channel 收集数据，运行在一个独立线程。

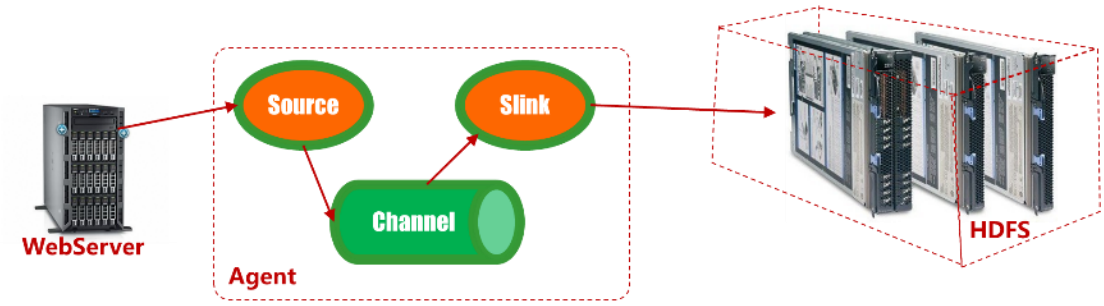
- Channel

连接 sources 和 sinks ，这个有点像一个队列。

- Events

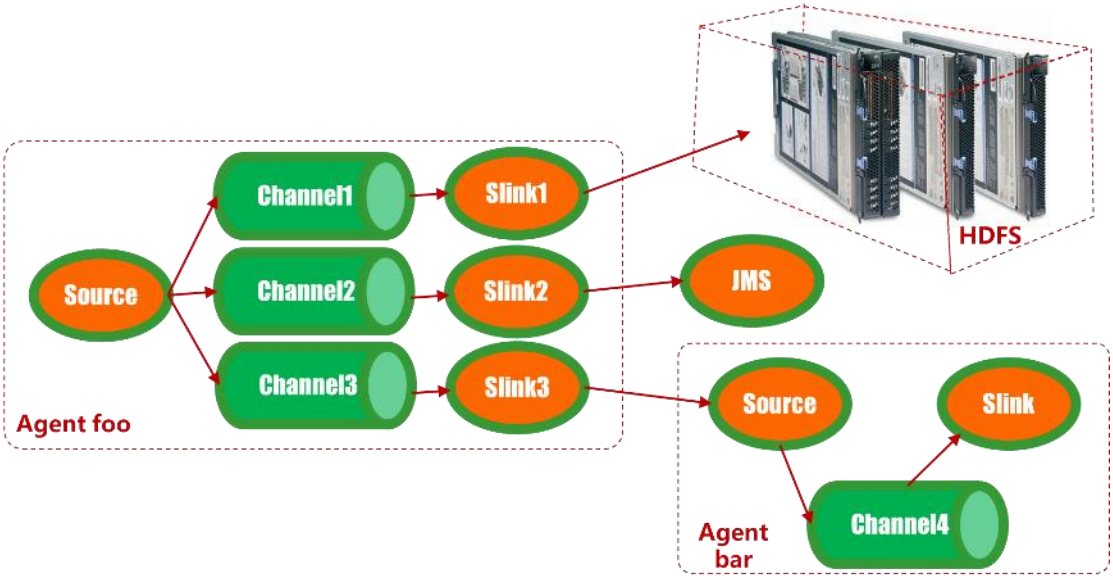
可以是日志记录、 avro 对象等。

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，如下图：



Flume 提供了大量内置的 Source、Channel 和 Sink 类型。不同类型的 Source,Channel 和 Sink 可以自由组合。

组合方式基于用户设置的配置文件，非常灵活。比如：Channel 可以把事件暂存在内存里，也可以持久化到本地硬盘上。Sink 可以把日志写入 HDFS, HBase，甚至是另外一个 Source 等等。Flume 支持用户建立多级流，也就是说，多个 agent 可以协同工作，并且支持 Fan-in、Fan-out、Contextual Routing、Backup Routes。下图描述了内置组件的组合：



与其他流处理框架相比，Flume 具备如下的特点：

丰富的缺省实现，用户只需简单文件配置即可完成部署，无须编写 Java 代码。

优雅的框架结构，各个模块之间界限分明。用户可以按相关接口非常方便的自行实现相关数据源、 Channel 选择算法等，即可获得满足用户特殊需求的逻辑。

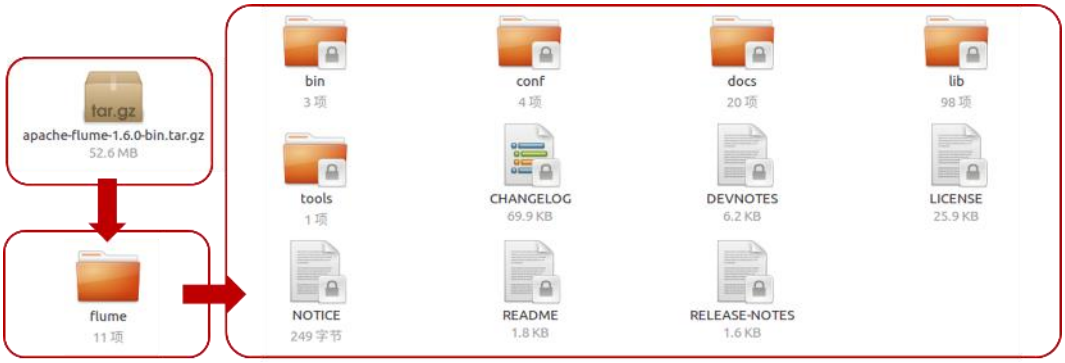
支持负载均衡。

Flume-OG 通过 Zookeeper 能支持自动 FailOver，Flumn-NG 尚未支持。

几乎包含在所有 Hadoop 发行版中，与 Hadoop 集成方便。

第3节 Flume 安装与使用

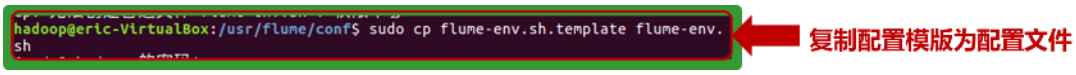
安装 Flume 需要在官网下载发行包并解压：



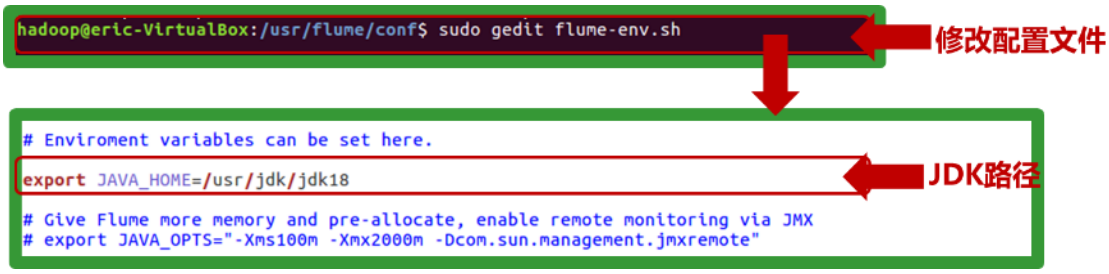
然后设置相应的环境变量：

```
export FLUME_HOME=/usr/flume
export PATH=$PATH:$FLUME_HOME/bin
```

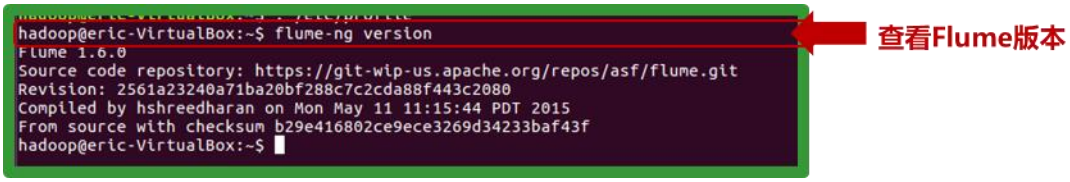
接下来修改 Flume 配置文件，将 FLUME_HOME/conf 中的配置模板文件 flume-env.sh.template 复制一份并命名为 flume-env.sh：



修改配置文件，显式指定 JDK 的安装路径：



测试安装是否成功：



安装成功后，可以使用 Flume 监听 NetCat，接下来使用如下的 Flume 的内置组件来监听网络数据交换：

NetCat Source，这个 source 会监听一个端口，然后把每行通过这个端口发过来的文本当做一个事件传给 channel。

Logger Sink，采用日志的形式来消费 Event。日志级别为 INFO。

Memory Channel，使用内存来暂存 Event。

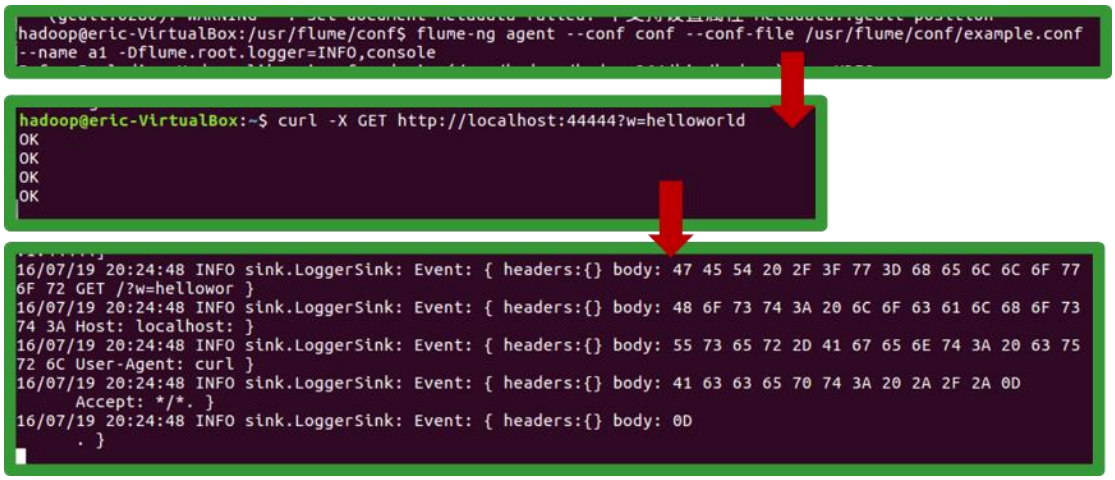
要完成这个需求，首先创建配置文件 example.conf：

```
# example.conf: A single-node Flume configuration

# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
#监听 NetCat，端口 44444
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Describe the sink
a1.sinks.k1.type = logger

# Use a channel which buffers events in memory
#使用内存缓存
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
```

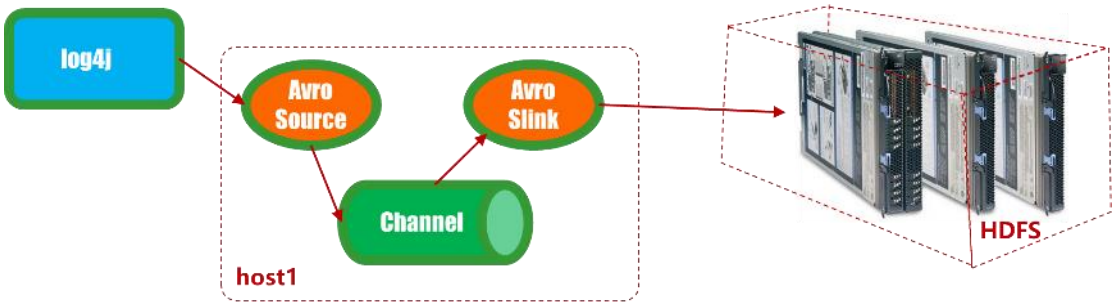
```
a1.sinks.k1.channel = c1
```

启动 Flume 如果没有错误的话，shell 会保持监听状态，此时监听 44444 端口：

```
flume-ng agent --conf conf --conf-file 创建的配置文件路径 --name 任务名称 -Dflume.root.logger=INFO,console
```



由本机将日志发送到 host1 上，并由 host1 上的 flume 将日志存储到 hdfs 上 。架构如下：



要想实现 Flume 收集 log4j 日志文件并写入 hdfs，首先创建配置文件：

```
# source, channel, sink definition
agent.channels = mem-channel
agent.sources = log4j-avro-source
agent.sinks = hdfs-sink

# Channel
# Define a  memory channel called mem-channel on agent
```

```
agent.channels.mem-channel.type = memory


# Source
# Define an Avro source called log4j-avro-channel on agent and tell
it
# to bind to host1:12345. Connect it to channel mem-channel.
agent.sources.log4j-avro-source.type = avro
agent.sources.log4j-avro-source.bind = 127.0.0.1
agent.sources.log4j-avro-source.port = 12345
agent.sources.log4j-avro-source.channels = mem-channel


# Sink
# Define a logger sink that simply logs all events it receives
# and connect it to the other end of the same channel.
agent.sinks.hdfs-sink.type = hdfs
agent.sinks.hdfs-sink.hdfs.path                                    =
hdfs://127.0.0.1:9000/flume/events/
agent.sinks.hdfs-sink.channel = mem-channel
```

接下来编写 log4j.properties:

```
log4j.rootLogger=INFO,flume
log4j.appender.flume=org.apache.flume.clients.log4jappender.Log4
jAppender
log4j.appender.flume.Hostname=127.0.0.1
log4j.appender.flume.Port=12345
```

接下来编写产生日志的程序：

```
public class FlumeLog {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger(FlumeLog.class);
        logger.info("hello world");
        logger.info("My name is alex");
        logger.info("How are you?");
    }
```

```
}
```

程序执行结果如下：

Source 作为 Client 端操作消费数据的来源，Flume 支持 Avro，log4j，syslog 和 http post(body 为 json 格式)。 可以让应用程序同已有的 Source 直接打交道，如 AvroSource， SyslogTcpSource。也可以写一个 Source， 以 IPC(进程间通信协议) 或 RPC(远程进程间通信协议) 的方式接入自己的应用，Avro 和 Thrift 都可以 (分别有 NettyAvroRpcClient 和 ThriftRpcClient 实现了 RpcClient 接口)， 其中 Avro 是默认的 RPC 协议。

对现有程序改动最小的使用方式是使用是直接读取程序原来记录的日志文件， 基本可以实现无缝接入， 不需要对现有程序进行任何改动。

对于直接读取文件 Source,有两种方式：

- **Exec source**

可通过写 Unix command 的方式组织数据，最常用的就是 tail -F [file]

可以实现实时传输，但在 Flume 不运行和脚本错误时，会丢失数据，也不支持断点续传功能。因为没有记录上次文件读到的位置,从而无法知道下次再读时，从什么地方开始读。特别是在日志文件一直在增加的时候。Flume 的 Source 失败后再次开启的这段时间内，增加的日志内容，没办法被 source 读取。

Flume 有一个 execStream 的扩展，可以编写一个监控日志增加情况，通过自己写的工具把增加的内容传送给 Flume 的 node 和 sink 的 node。

- **Spooling Directory Source**

SpoolSource 监测配置的目录下新增的文件，并将文件中的数据读取出来，可实现准实时。

需要注意两点：拷贝到 spool 目录下的文件不可以再打开编辑，spool 目录下不可包含相应的子目录。

在实际使用的过程中，可以结合 log4j 使用，使用 log4j 的时候，将 log4j 的文件分割机制设为 1 分钟一次，将文件拷贝到 spool 的监控目录。log4j 有一个 TimeRolling 的插件，可以把 log4j 分割的文件到 spool 目录。基本能实时的监控。

Flume 在传完文件之后，将会修改文件的后缀，变为.COMPLETED（后缀也可

以在配置文件中灵活指定)。

ExecSource 可以实现对日志的实时收集,但是存在 Flume 不运行或者指令执行出错时，将无法收集到日志数据，无法保证日志数据的完整性。SpoolSource 虽然无法实现实时的收集数据，但是可以使用以分钟的方式分割文件，趋近于实时。如果应用无法实现以分钟切割日志文件的话,可以两种收集方式结合使用。

Flume 中的 Channel 有多种方式：

- MemoryChannel

MemoryChannel 可以实现高速的吞吐，但是无法保证数据的完整性。 Memory Channel 是一个不稳定的隧道，其原因是由于它在内存中存储所有事件。如果 Java

进程死掉， 任何存储在内存的事件将会丢失。 另外， 内存的空间受到 RAM 大小的限制,

- JDBC Channel
- MemoryRecoverChannel

MemoryRecoverChannel 在官方文档的建议上已经建义使用 FileChannel 来替换。

- FileChannel。

FileChannel 保证数据的完整性与一致性。在具体配置 FileChannel 时，建议 FileChannel 设置的目录和程序日志文件保存的目录设成不同的磁盘,以便提高效率。FileChannel 是一个持久化的隧道， 它持久化所有的事件， 并将其存储到磁盘中。 因此， 即使 Java 虚拟机当掉， 或者操作系统崩溃或重启，再或者事件没有在管道中成功地传递到下一个代理,这一切都不会造成数据丢失只要磁盘空间足够，它就可以将所有事件数据存储到磁盘上。

Sink 在设置存储数据时，可以向文件系统中，数据库中，hadoop 中储数据，在日志数据较少时，可以将数据存储在文件系中，并且设定一定的时间间隔保存数据。在日志数据较多时，可以将相应的日志数据存储到 Hadoop 中，便于日后进行相应的数据分析。

第4节 Flume 配置文件

Flume 配置文件特点：

- 单配置文件(Java property 文档格式)
- 基于流(flow)配置
- 流包括一个 Client 以及其运行的 Source，Channel 和 Sink
- 一个流可以包含多个 Channel
- 流以唯一标示符(identifier)命名

- 每行定义一个属性/值对：

```
<flow_identifier>.type.subtype.parameter.config=value
```

以下列出 Flume 标准的配置文件组成模版：

```
# list the sources, sinks and channels for the agent
 <Agent>.sources = <Source>
<Agent>.sinks = <Sink>
<Agent>.channels = <Channel1> <Channel2>
 # set channel for source
<Agent>.sources.<Source>.channels = <Channel1> <Channel2> …
 # set channel for sink
<Agent>.sinks.<Sink>.channel = <Channel1>
# Configure individual components
# properties for sources
<Agent>.sources.<Source>.<someProperty> = <someValue>
 # properties for channels
<Agent>.channel.<Channel>.<someProperty> = <someValue>
# properties for sinks
 <Agent>.sinks.<Sink>.<someProperty> = <someValue>
```

第12章　Kafka

第1节 Kafka 入门介绍

消息队列技术是分布式应用间交换信息的一种技术。消息队列可驻留在内存或磁盘上, 队列存储消息直到它们被应用程序读走。通过消息队列，应用程序可独立地执行--它们不需要知道彼此的位置、或在继续执行前不需要等待接收程序接收此消息。

在分布式计算环境中，为了集成分布式应用，开发者需要对异构网络环境下的分布式应用提供有效的通信手段。为了管理需要共享的信息，对应用提供公共的信息交换机制是重要的。

常用的消息队列技术是 Message Queue。Message Queue 的通讯模式如下：

- **点对点通讯**

点对点方式是最为传统和常见的通讯方式，它支持一对一、一对多、多对多、多对一等多种配置方式，支持树状、网状等多种拓扑结构。

- **多点广播**

MQ 适用于不同类型的应用。其中重要的，也是正在发展中的是"多点广播"应用，即能够将消息发送到多个目标站点 (Destination List)。可以使用一条 MQ 指令将单一消息发送到多个目标站点，并确保为每一站点可靠地提供信息。MQ 不仅提供了多点广播的功能，而且还拥有智能消息分发功能，在将一条消息发送到同一系统上的多个用户时，MQ 将消息的一个复制版本和该系统上接收者的名单发送到目标 MQ 系统。目标 MQ 系统在本地复制这些消息，并将它们发送到名单上的队列，从而尽可能减少网络的传输量。

- **发布/订阅 (Publish/Subscribe) 模式**

发布/订阅功能使消息的分发可以突破目的队列地理指向的限制，使消息按照特定的主题甚至内容进行分发，用户或应用程序可以根据主题或内容接收到所需要的消息。发布/订阅功能使得发送者和接收者之间的耦合关系变得更为松散，发送者不必关心接收者的目的地址，而接收者也不必关心消息的发送地址，而只是根据消息的主题进行消息的收发。

- **群集 (Cluster)**

为了简化点对点通讯模式中的系统配置，MQ 提供 Cluster(群集) 的解决方案。群集类似于一个域 (Domain)，群集内部的队列管理器之间通讯时，不需要两两之间建立消息通道，而是采用群集 (Cluster) 通道与其它成员通讯，从而大大简化了系统配置。此外，群集中的队列管理器之间能够自动进行负载均衡，当某一队列管理器出现故障时，其它队列管理器可以接管它的工作，从而大大提高系统的高可靠性。

Kafka 是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据。 这种动作（网页浏览，搜索和其他用户的行动）是在现代网络上的许多社会功能的一个关键因素。 这些数据通常是由于吞吐量的要求而通过处理日志和日志聚合来解决。对于像 Hadoop 的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

Kafka 的目的是通过 Hadoop 的并行加载机制来统一线上和离线的消息处理，也是为了通过集群机来提供实时的消费。

Kafka 具有如下特性：

- 通过磁盘数据结构提供消息的持久化，这种结构对于即使数以 TB 的消息存储也能够保持长时间的稳定性能。
- 高吞吐量，即使是非常普通的硬件 Kafka 也可以支持每秒数百万的消息。
- 支持通过 Kafka 服务器和消费机集群来分区消息。
- 支持 Hadoop 并行数据加载。

要使用 Kafka，首先需要理解它包含的几个重要术语：

Broker，Kafka 集群包含一个或多个服务器，这种服务器被称为 broker。

Topic，每条发布到 Kafka 集群的消息都有一个类别，这个类别被称为 Topic。（物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 broker 上但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存于何处）。

Partition，Partition 是物理上的概念，每个 Topic 包含一个或多个 Partition。

Producer，负责发布消息到 Kafka broker。

Consumer，消息消费者，向 Kafka broker 读取消息的客户端。

Consumer Group，每个 Consumer 属于一个特定的 Consumer Group（可为每个 Consumer 指定 group name，若不指定 group name 则属于默认的 group）。

第2节 Kafka 交互流程

Kafka 是一个基于分布式的消息发布-订阅系统，它被设计成快速、可扩展的、持久的。

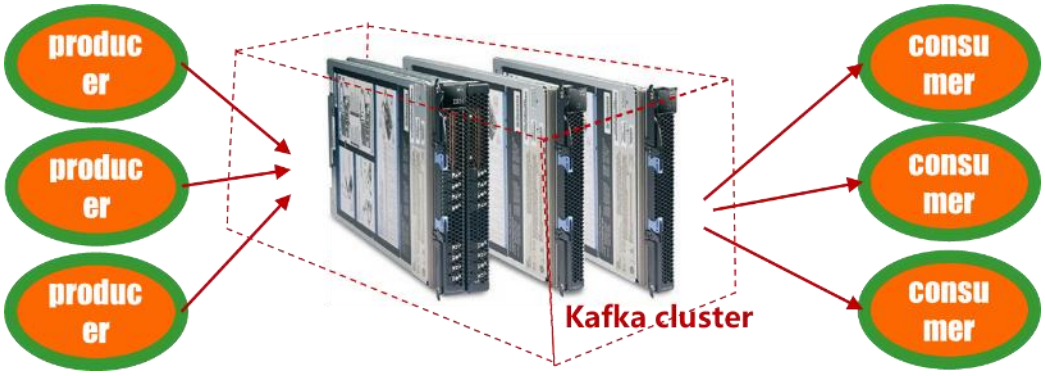
与其他消息发布-订阅系统类似，Kafka 在主题当中保存消息的信息。生产者向主题写入数据，消费者从主题读取数据。由于 Kafka 的特性是支持分布式，同时也是基于分布式的，所以主题也是可以在多个节点上被分区和覆盖的。

信息是一个字节数组，程序员可以在这些字节数组中存储任何对象，支持的数据格式包括 String、JSON、Avro。Kafka 通过给每一个消息绑定一个键值的方式来保证生产者可以把所有的消息发送到指定位置。属于某一个消费者群组的

消费者订阅了一个主题，通过该订阅消费者可以跨节点地接收所有与该主题相关的消息，每一个消息只会发送给群组中的一个消费者，所有拥有相同键值的消息都会被确保发给这一个消费者。

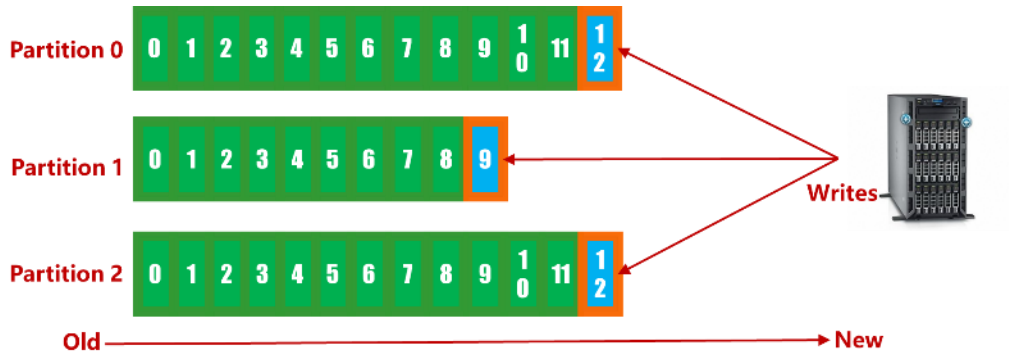
Kafka 设计中将每一个主题分区当作一个具有顺序排列的日志。同处于一个分区中的消息都被设置了一个唯一的偏移量。Kafka 只会保持跟踪未读消息，一旦消息被置为已读状态，Kafka 就不会再去管理它了。Kafka 的生产者负责在消息队列中对生产出来的消息保证一定时间的占有，消费者负责追踪每一个主题（可以理解为一个日志通道） 的消息并及时获取它们。基于这样的设计，Kafka 可以在消息队列中保存大量的开销很小的数据，并且支持大量的消费者订阅。

Kafka 将消息以 topic 为单位进行归纳。将向 Kafka topic 发布消息的程序成为 producers，将预订 topics 并消费消息的程序成为 consumer。Kafka 以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个 broker.
producers 通过网络将消息发送到 Kafka 集群，集群向消费者提供消息，如下图所示：



Kafka 提供了 Java 客户端，并且对多种语言都提供了支持，客户端和服务端通过 TCP 协议通信。

一个 topic 是对一组消息的归纳。对每个 topic，Kafka 对它的日志进行了分区，如下图所示：



每个分区都由一系列有序的、不可变的消息组成，这些消息被连续的追加到分区中。分区中的每个消息都有一个连续的序列号叫做 offset,用来在分区中唯一的标识这个消息。

在一个可配置的时间段内，Kafka 集群保留所有发布的消息，不管这些消息

有没有被消费。比如，如果消息的保存策略被设置为 2 天，那么在一个消息被发布的两天时间内，它都是可以被消费的。之后它将被丢弃以释放空间。Kafka 的性能是和数据量无关的常量级的，所以保留太多的数据并不是问题。

实际上每个 consumer 唯一需要维护的数据是消息在日志中的位置，也就是 offset.这个 offset 有 consumer 来维护：

一般情况下随着 consumer 不断的读取消息，这 offset 的值不断增加，但其实 consumer 可以以任意的顺序读取消息，比如它可以将 offset 设置成为一个旧的值来重读之前的消息。

Kafka consumers 非常的轻量级,它们可以在不对集群和其他 consumer 造成影响的情况下读取消息。可以使用命令行来"tail"消息而不会对其他正在消费消息的 consumer 造成影响。

将日志分区可以达到以下目的：这使得每个日志的数量不会太大，可以在单个服务上保存。每个分区可以单独发布和消费，为并发操作 topic 提供了一种可能。

每个分区在 Kafka 集群的若干服务中都有副本，这样这些持有副本的服务可以共同处理数据和请求，副本数量是可以配置的。副本使 Kafka 具备了容错能力。每个分区都由一个服务器作为“leader”，零或若干服务器作为“followers”,leader 负责处理消息的读和写，followers 则去复制 leader。如果 leader 宕机了，followers 中的一台则会自动成为 leader。

集群中的每个服务都会同时扮演两个角色：作为它所持有的一部分分区的 leader，同时作为其他分区的 followers，这样集群就会具有较好的负载均衡。

Producer 将消息发布到它指定的 topic 中,并负责决定发布到哪个分区。通常简单的由负载均衡机制随机选择分区，但也可以通过特定的分区函数选择分区。使用的更多的是第二种。

发布消息通常有两种模式：

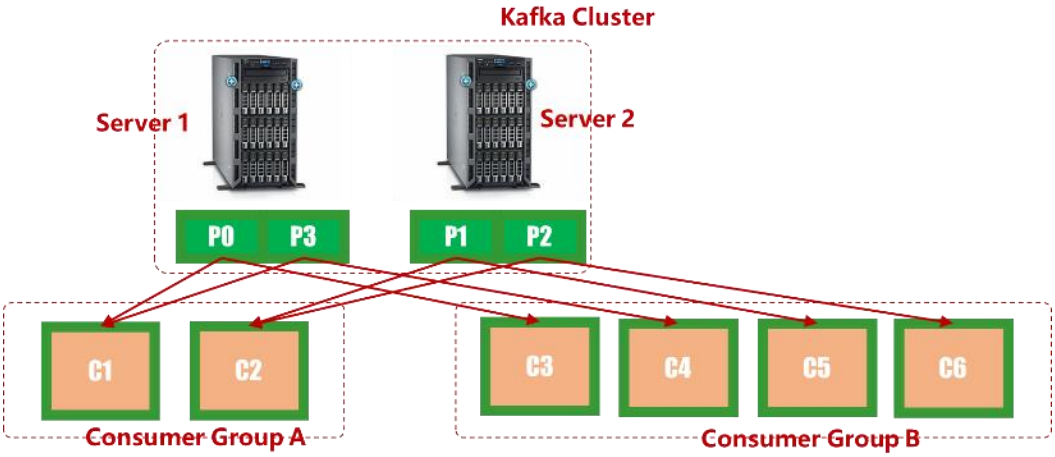
队列模式（queuing），consumers 可以同时从服务端读取消息，每个消息只被其中一个 consumer 读到。

发布-订阅模式(publish-subscribe)，消息被广播到所有的 consumer 中。

Consumers 可以加入一个 consumer 组，共同竞争一个 topic，topic 中的消息将被分发到组中的一个成员中。同一组中的 consumer 可以在不同的程序中，也可以在不同的机器上。如果所有的 consumer 都在一个组中，这就成为了传统的队列模式，在各 consumer 中实现负载均衡。如果所有的 consumer 都不在不同的组中，这就成为了发布-订阅模式，所有的消息都被分发到所有的 consumer 中。

更常见的是，每个 topic 都有若干数量的 consumer 组，每个组都是一个逻辑上的“订阅者”，为了容错和更好的稳定性，每个组由若干 consumer 组成。这其实就是一个发布-订阅模式，只不过订阅者是个组而不是单个 consumer。

由两个机器组成的集群拥有 4 个分区 (P0-P3) 2 个 consumer 组. A 组有两个 consumerB 组有 4 个：



相比传统的消息系统，Kafka 可以很好的保证有序性。传统的队列在服务器上保存有序的消息，如果多个 consumers 同时从这个服务器消费消息，服务器就会以消息存储的顺序向 consumer 分发消息。虽然服务器按顺序发布消息，但是消息是被异步的分发到各 consumer 上，所以当消息到达时可能已经失去了原来的顺序，这意味着并发消费将导致顺序错乱。为了避免故障，这样的消息系统通常使用“专用 consumer”的概念，其实就是只允许一个消费者消费消息，当然这就意味着失去了并发性。

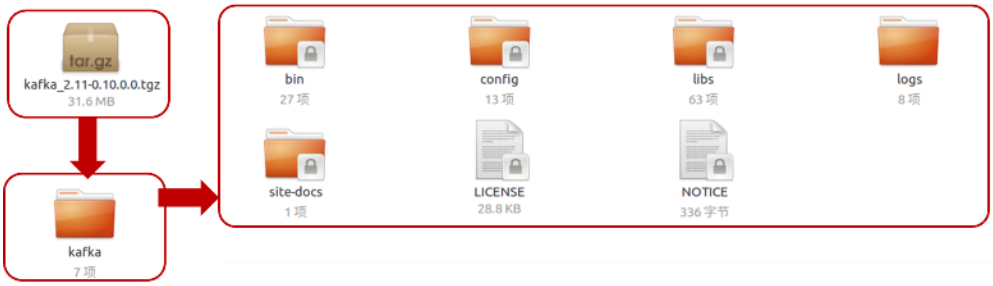
通过分区的概念，Kafka 可以在多个 consumer 组并发的情况下提供较好的有序性和负载均衡。将每个分区分只分发给一个 consumer 组，这样一个分区就只被这个组的一个 consumer 消费，就可以顺序的消费这个分区的消息。因为有多个分区，依然可以在多个 consumer 组之间进行负载均衡。

注意 consumer 组的数量不能多于分区的数量，也就是有多少分区就允许多少并发消费。

Kafka 只能保证一个分区之内消息的有序性，在不同的分区之间是不可以的，这已经可以满足大部分应用的需求。如果需要 topic 中所有消息的有序性，那就只能让这个 topic 只有一个分区，当然也就只有一个 consumer 组消费它。

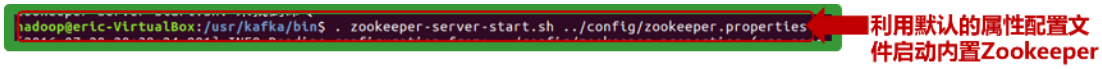
第3节 Kafka 安装与使用

首先，在官网下载发行包并解压：



Kafka 的运行依赖于 Zookeeper，因此在启动 Kafka 之前，需要先启动 Zookeeper 服务：

```
zookeeper-server-start.sh ../config/zookeeper.properties
```

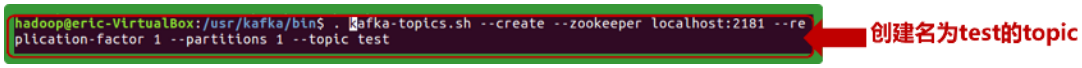


然后启动 Kafka 服务：

```
kafka-server-start.sh ../config/server.properties
```

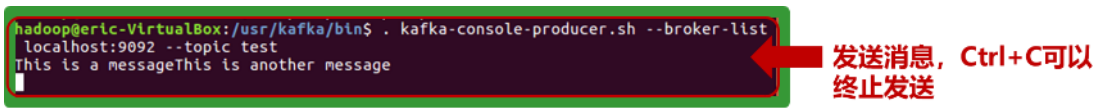
创建 topic：

```
kafka-topics.sh --create --zookeeper Zookeeper服务器地址:端口 -
replication-factory 数量 --partitions 数量 --topic 名称
```



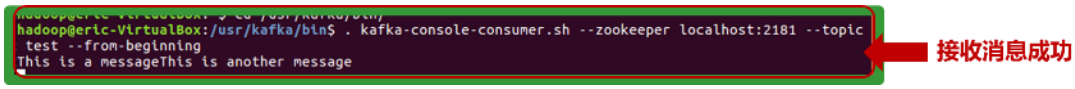
接下来可以使用命令行发送一条消息：

```
kafka-console-producer.sh --broker-list 服务器：端口号 --topic
topic名称
消息内容
```



同样可以使用命令行接收数据：

```
kafka-console-consumer.sh --broker-list 服务器：端口号 --topic
topic名称 --from-begining
```



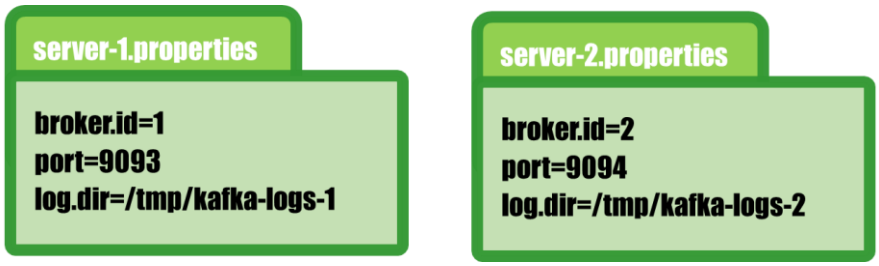
第4节 搭建多个 broker 的集群

刚才只是启动了单个 broker，现在启动有 3 个 broker 组成的集群，这些 broker 节点也都是在本机上的：

首先为每个节点复制配置文件：

```
sudo cp server.properties server-1.properties
sudo cp server.properties server-2.properties
```

在新复制的配置文件中加入：



broker.id 在集群中唯一的标注一个节点，因为在同一个机器上，所以必须指定不同的端口和日志文件，避免数据被覆盖。启动 Zookeeper 后依次启动 3 个 Kafka 进程：

```
kafka-server-start.sh ../config/server.properties
kafka-server-start.sh ../config/server-1.properties
kafka-server-start.sh ../config/server-2.properties
```

接下来创建一个拥有 3 个副本的 topic 来验证集群是否配置成功:

```
kafka-topics.sh --create --zookeeper Zookeeper服务器地址:端口 --replication-factory 数量 --partitions 数量 --topic 名称
```

```
hadoop@eric-VirtualBox:/usr/kafka/bin$ kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions
 1 --topic my-replicated-topic
Created topic "my-replicated-topic".
```

查看集群中的节点信息：

```
kafka-topics.sh --describe --zookeeper Zookeeper服务器:端口 --topic 名称
```

```
hadoop@eric-VirtualBox:/usr/kafka/bin$ kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic
Topic:my-replicated-topic       PartitionCount:1        ReplicationFactor:3     Configs:
        Topic: my-replicated-topic      Partition: 0    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
```

第一行是对所有分区的一个描述，然后每个分区都会对应一行，因为只有一个分区所以下面就只加了一行。

leader：负责处理消息的读和写，leader 是从所有节点中随机选择的.

replicas：列出了所有的副本节点，不管节点是否在服务中.

isr：是正在服务中的节点。在例子中，节点 2 是作为 leader 运行。

接下来测试消息的发送和接收是否成功：

```
hadoop@eric-VirtualBox:/usr/kafka/bin$ kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic
my test message 1my test message 2
^Chadoop@eric-VirtualBox:/usr/kafka/bin$
```

向集群中的任意节点发送消息均可

```
^Chadoop@eric-VirtualBox:/usr/kafka/bin$ kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-repl
ated-topic
my test message 1my test message 2
```

第5节 Kafka 开发

首先创建 Kafka 项目，然后将 KAFKA_HOME/lib 文件夹下的 lib 加入项目 build path。然后构建一个接口声明 Kafka 中常见的属性取值：

```
public interface KafkaProperties {
    final static String zkConnect = "127.0.0.1:2181";
    final static String groupId = "group1";
    final static String topic = "topic1";
    final static String kafkaServerURL = "127.0.0.1";
    final static int kafkaServerPort = 9092;
    final static int kafkaProducerBufferSize = 64 * 1024;
    final static int connectionTimeOut = 20000;
    final static int reconnectInterval = 10000;
    final static String topic2 = "topic2";
    final static String topic3 = "topic3";
    final static String clientId = "SimpleConsumerDemoClient";
}
```

接下来构建消息生产者代码，一个普通的生产者实现参考如下：

```
public class KafkaProducer extends Thread
{
    private final kafka.javaapi.producer.Producer<Integer, String>
producer;
    private final String topic;
    private final Properties props = new Properties();
    public KafkaProducer(String topic)
    {
        props.put("serializer.class",
"kafka.serializer.StringEncoder");
        props.put("metadata.broker.list", "127.0.0.1:9092");
        producer = new kafka.javaapi.producer.Producer<Integer,
String>(new ProducerConfig(props));
        this.topic = topic;
    }
    @Override
    public void run() {
        int messageNo = 1;
        while (true)
        {
            String messageStr = new String("Message_" + messageNo);
            System.out.println("Send:" + messageStr);
            producer.send(new KeyedMessage<Integer, String>(topic,
messageStr));
```

```
            messageNo++;
            try {
                sleep(3000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

接下来是一个典型的消息消费者参考实现：

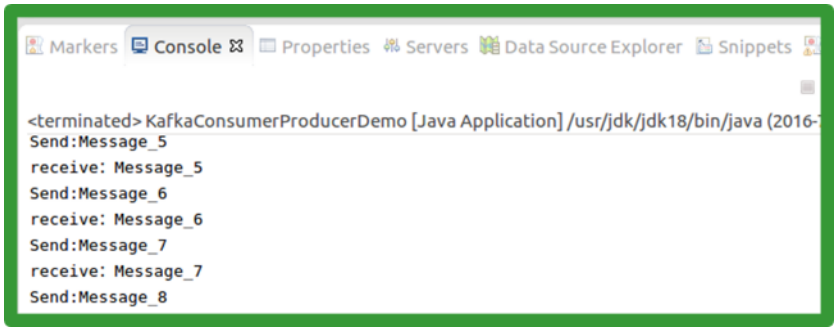
```
public class KafkaConsumer extends Thread {
    private final ConsumerConnector consumer;
    private final String topic;

    public KafkaConsumer(String topic) {
        consumer = kafka.consumer.Consumer
                .createJavaConsumerConnector(createConsumerConfig());
        this.topic = topic;
    }

    private static ConsumerConfig createConsumerConfig() {
        Properties props = new Properties();
        props.put("zookeeper.connect", KafkaProperties.zkConnect);
        props.put("group.id", KafkaProperties.groupId);
        props.put("zookeeper.session.timeout.ms", "40000");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
        return new ConsumerConfig(props);
    }

    @Override
    public void run() {
        Map<String, Integer> topicCountMap = new HashMap<String,
Integer>();
        topicCountMap.put(topic, new Integer(1));
        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer
                .createMessageStreams(topicCountMap);
        KafkaStream<byte[], byte[]> stream =
consumerMap.get(topic).get(0);
        ConsumerIterator<byte[], byte[]> it = stream.iterator();
        while (it.hasNext()) {
```

```
            System.out.println("receive: " + new
String(it.next().message()));
            try {
                sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

最后实现消息生产者和消费者的调度代码来验证功能是否正确：

```
public class KafkaConsumerProducerDemo {
    public static void main(String[] args) {
        KafkaProducer producerThread = new
KafkaProducer(KafkaProperties.topic);
        producerThread.start();
        KafkaConsumer consumerThread = new
KafkaConsumer(KafkaProperties.topic);
        consumerThread.start();
    }
}
```

程序执行结果如下：



Kafka 大量依赖文件系统去存储和缓存消息。

对于硬盘有个传统的观念是硬盘总是很慢，这使很多人怀疑基于文件系统的架构能否提供优异的性能。实际上硬盘的快慢完全取决于使用它的方式。设计良好的硬盘架构可以和内存一样快.

在 6 块 7200 转的 SATA RAID-5 磁盘阵列的线性写速度差不多是 600MB/s，但是随机写的速度却是 100k/s，差了差不多 6000 倍。现代的操作系统都对此做了大量的优化，使用了 read-ahead 和 write-behind 的技巧，读取的时候成块的预读取数据，写的时候将各种微小琐碎的逻辑写入组织合并成一次较大的物理写入。

为了提高性能，现代操作系统往往使用内存作为磁盘的缓存，现代操作系统乐于把所有空闲内存用作磁盘缓存，虽然这可能在缓存回收和重新分配时牺牲一些性能。所有的磁盘读写操作都会经过这个缓存，这不太可能被绕开，除非直接使用 I/O。所以虽然每个程序都在自己的线程里只缓存了一份数据，但在操作系统的缓存里还有一份，这等于存了两份数据。

Java 对象占用空间是非常大的，差不多是要存储的数据的两倍甚至更高。

随着堆中数据量的增加，垃圾回收回变的越来越困难。

如果把数据缓存在内存里，因为需要存储两份，不得不使用两倍的内存空间，Kafka 基于 JVM，又不得不将空间再次加倍，再加上要避免 GC 带来的性能影响，在一个 32G 内存的机器上，不得不使用到 28-30G 的内存空间。并且当系统重启的时候，又必须要将数据刷到内存中（ 10GB 内存差不多要用 10 分钟），就算使用冷刷新（不是一次性刷进内存，而是在使用数据的时候没有就刷到内存）也会导致最初的时候性能非常差。但是使用文件系统，即使系统重启了，也不需要刷新数据。使用文件系统也简化了维护数据一致性的逻辑。

与传统的将数据缓存在内存中然后刷到硬盘的设计不同，Kafka 直接将数据写到了文件系统的日志中。

在大多数的消息系统中，数据持久化的机制往往是为每个 cosumer 提供一个 B 树或者其他的随机读写的数据结构。B 树当然是很棒的，但是也带了一些代价：比如 B 树的复杂度是 O(log N)，O(log N)通常被认为就是常量复杂度了，但对于硬盘操作来说并非如此。磁盘进行一次搜索需要 10ms，每个硬盘在同一时间只能进行一次搜索，这样并发处理就成了问题。虽然存储系统使用缓存进行了大量优化，但是对于树结构的性能的观察结果却表明，它的性能往往随着数据的增长而线性下降，数据增长一倍，速度就会降低一倍。对于主要用于日志处理的消息系统，数据的持久化可以简单的通过将数据追加到文件中实现，读的时候从文件中读就好了。这样做的好处是读和写都是 O(1) 的，并且读操作不会阻塞写操作和其他操作。这样带来的性能优势是很明显的，因为性能和数据的大小没有关系了。

既然可以使用几乎没有容量限制（相对于内存来说）的硬盘空间建立消息系统，就可以在没有性能损失的情况下提供一些一般消息系统不具备的特性。比如，一般的消息系统都是在消息被消费后立即删除，Kafka 却可以将消息保存一段时间（比如一星期），这给 consumer 提供了很好的机动性和灵活性。

数据传输的事务定义通常有以下三种级别：

- 最多一次：消息不会被重复发送，最多被传输一次，但也有可能一次不传输。
- 最少一次：消息不会被漏发送，最少被传输一次，但也有可能被重复传输。
- 精确的一次（Exactly once）：不会漏传输也不会重复传输，每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的。

大多数消息系统声称可以做到“精确的一次”，但是仔细阅读它们的文档可以

看到里面存在误导，比如没有说明当 consumer 或 producer 失败时怎么样，或者当有多个 consumer 并行时怎么样，或写入硬盘的数据丢失时又会怎么样。

kafka 的做法要更先进一些，当发布消息时，Kafka 有一个“committed”的概念，一旦消息被提交了，只要消息被写入的分区的所在的副本 broker 是活动的，数据就不会丢失。

如果 producer 发布消息时发生了网络错误，但又不确定是在提交之前发生的还是提交之后发生的，这种情况虽然不常见，但是必须考虑进去，现在 Kafka 版本还没有解决这个问题，将来的版本正在努力尝试解决。

并不是所有的情况都需要“精确的一次”这样高的级别，Kafka 允许 producer 灵活的指定级别。比如 producer 可以指定必须等待消息被提交的通知，或者完全的异步发送消息而不等待任何通知，或者仅仅等待 leader 声明它拿到了消息（followers 没有必要）。

从 consumer 的方面考虑事务，所有的副本都有相同的日志文件和相同的 offset，consumer 维护自己消费的消息的 offset，如果 consumer 不会崩溃当然可以在内存中保存这个值，当然谁也不能保证这点。如果 consumer 崩溃了，会有另外一个 consumer 接着消费消息，它需要从一个合适的 offset 继续处理。这种情况下可以有以下选择：

consumer 可以先读取消息，然后将 offset 写入日志文件中，然后再处理消息。这存在一种可能就是在存储 offset 后还没处理消息就 crash 了，新的 consumer 继续从这个 offset 处理，那么就会有些消息永远不会被处理，这就是最多一次。

consumer 可以先读取消息，处理消息，最后记录 offset，当然如果在记录 offset 之前就 crash 了，新的 consumer 会重复的消费一些消息，这就是最少一次。

“精确一次”可以通过将提交分为两个阶段来解决：保存了 offset 后提交一次，消息处理成功之后再提交一次。但是还有个更简单的做法：将消息的 offset 和消息被处理后的结果保存在一起。比如用 Hadoop ETL 处理消息时，将处理后的结果和 offset 同时保存在 HDFS 中，这样就能保证消息和 offser 同时被处理了。

Kafka 在提高效率方面做了很大努力。Kafka 的一个主要使用场景是处理网站活动日志，吞吐量是非常大的，每个页面都会产生好多次写操作。读方面，假设每个消息只被消费一次，读的量的也是很大的，Kafka 也尽量使读的操作更轻量化。

磁盘的性能问题-线性读写的情况下影响磁盘性能问题大约有两个方面：

- 太多的琐碎的 I/O 操作
- 太多的字节拷贝

I/O 问题发生在客户端和服务端之间，也发生在服务端内部的持久化的操作中。

Kafka 建立了消息集（message set）的概念，将消息组织到一起，作为处理的单位。以消息集为单位处理消息，比以单个的消息为单位处理，会提升不少性能。Producer 把消息集一块发送给服务端，而不是一条条的发送；服务端把消息集一次性的追加到日志文件中，这样减少了琐碎的 I/O 操作。consumer 也可以一次性的请求一个消息集。

另外一个性能优化是在字节拷贝方面。在低负载的情况下这不是问题，但是在高负载的情况下它的影响还是很大的。为了避免这个问题，Kafka 使用了标准的二进制消息格式，这个格式可以在 producer,broker 和 producer 之间共享而无需做任何改动。

Broker 维护的消息日志仅仅是一些目录文件，消息集以固定队的格式写入到日志文件中，这个格式 producer 和 consumer 是共享的，这使得 Kafka 可以一个很重要的点进行优化：消息在网络上的传递。现代的 unix 操作系统提供了高性能的将数据从页面缓存发送到 socket 的系统函数，在 linux 中，这个函数是 sendfile。

一般将数据从文件发送到 socket 的数据流向：

- 操作系统把数据从文件拷贝内核中的页缓存中
- 应用程序从页缓存从把数据拷贝自己的内存缓存中
- 应用程序将数据写入到内核中 socket 缓存中
- 操作系统把数据从 socket 缓存中拷贝到网卡接口缓存，从这里发送到网络上

上述流程显然是低效率的，因为有 4 次拷贝和 2 次系统调用。

Sendfile 通过直接将数据从页面缓存发送网卡接口缓存，避免了重复拷贝，大大的优化了性能。

在一个多 consumers 的场景里，数据仅仅被拷贝到页面缓存一次而不是每次消费消息的时候都重复的进行拷贝。这使得消息以近乎网络带宽的速率发送出去。这样在磁盘层面你几乎看不到任何的读操作，因为数据都是从页面缓存中直接发送到网络上去了。

很多时候，性能的瓶颈并非 CPU 或者硬盘而是网络带宽，对于需要在数据中心之间传送大量数据的应用更是如此。当然用户可以在没有 Kafka 支持的情况下各自压缩自己的消息，但是这将导致较低的压缩率，因为相比于将消息单独压缩，将大量文件压缩在一起才能起到最好的压缩效果。

Kafka 采用了端到端的压缩：因为有“消息集”的概念，客户端的消息可以一起被压缩后送到服务端，并以压缩后的格式写入日志文件，以压缩的格式发送到 consumer，消息从 producer 发出到 consumer 拿到都被是压缩的，只有在 consumer 使用的时候才被解压缩，所以叫做“端到端的压缩”。

producer 直接将数据发送到 broker 的 leader(主节点)，不需要在多个节点进行分发。为了帮助 producer 做到这点，所有的 Kafka 节点都可以及时的告知：哪些节点是活动的，目标 topic 目标分区的 leader 在哪。这样 producer 就可以

直接将消息发送到目的地了。

客户端控制消息将被分发到哪个分区。可以通过负载均衡随机的选择，或者使用分区函数。Kafka 允许用户实现分区函数，指定分区的 key，将消息 hash 到不同的分区上(当然有需要的话，也可以覆盖这个分区函数自己实现逻辑)。比如如果你指定的 key 是 user id，那么同一个用户发送的消息都被发送到同一个分区上。经过分区之后，consumer 就可以有目的的消费某个分区的消息。

批量发送可以很有效的提高发送效率。Kafka producer 的异步发送模式允许进行批量发送，先将消息缓存在内存中，然后一次请求批量发送出去。这个策略可以配置的，比如可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出去（比如 100 条消息就发送，或者每 5 秒发送一次)。这种策略将大大减少服务端的 I/O 次数。

既然缓存是在 producer 端进行的，那么当 producer 崩溃时，这些消息就会丢失。

Kafa consumer 消费消息时，向 broker 发出"fetch"请求去消费特定分区的消息。consumer 指定消息在日志中的偏移量（offset)，就可以消费从这个位置开始的消息。customer 拥有了 offset 的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的。

Kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计：producer 将消息推送到 broker，consumer 从 broker 拉取消息。

一些消息系统采用了 push 模式，将消息推送到下游的 consumer。这样做有好处也有坏处：由 broker 决定消息推送的速率，对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时，consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式。

Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略。

Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞直到新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发送)。

对消费消息状态的记录也是很重要的。大部分消息系统在 broker 端的维护消息被消费的记录：一个消息被分发到 consumer 后 broker 就马上进行标记或者等待 customer 的通知后进行标记。这样在消息在消费后立马就删除以减少空间占用。如果一条消息发送出去之后就立即被标记为消费过的，一旦 consumer 处理消息时失败了消息就丢失了。很多消息系统提供了另外一个个功能：当消息被

发送出去之后仅仅被标记为已发送状态，当接到 consumer 已经消费成功的通知后才标记为已被消费的状态。这虽然解决了消息丢失的问题，但产生了新问题，如果 consumer 处理消息成功了但是向 broker 发送响应时失败了，这条消息将被消费两次。

broker 必须维护每条消息的状态，并且每次都要先锁住消息然后更改状态然后释放锁。这样且不说要维护大量的状态数据，比如如果消息发送出去但没有收到消费成功的通知，这条消息将一直处于被锁定的状态。

Kafka 采用了不同的策略，Topic 被分成了若干分区，每个分区在同一时间只被一个 consumer 消费。这意味着每个分区被消费的消息在日志中的位置仅仅是一个简单的整数：offset。这样就很容易标记每个分区消费状态，仅仅需要一个整数而已。

这带来了另外一个好处，consumer 可以把 offset 调成一个较老的值，去重新消费老的消息。这对传统的消息系统来说看起来有些不可思议，但确实非常有用。

高级的数据持久化允许 consumer 每个隔一段时间批量的将数据加载到线下系统中比如 Hadoop 或者数据仓库。这种情况下，Hadoop 可以将加载任务分拆，拆成每个 broker 或每个 topic 或每个分区一个加载任务。

Hadoop 具有任务管理功能，当一个任务失败了就可以重启而不用担心数据被重新加载，只要从上次加载的位置继续加载消息就可以了。

Kafka 允许 topic 的分区拥有若干副本，这个数量是可以配置的。Kafka 会自动在每个副本上备份数据，所以当一个节点 down 掉时数据依然是可用的。Kafka 的副本功能不是必须的，可以配置只有一个副本，这样其实就相当于只有一份数据。

创建副本的单位是 topic 的分区，每个分区都有一个 leader 和零或多个 followers，所有的读写操作都由 leader 处理，一般分区的数量都比 broker 的数量多的多，各分区的 leader 均匀的分布在 brokers 中。所有的 followers 都复制 leader 的日志，日志中的消息和顺序都和 leader 中的一致。followers 向普通的 consumer 那样从 leader 那里拉取消息并保存在自己的日志文件中。

许多分布式的消息系统自动的处理失败的请求，它们对一个节点是否有效有着清晰的定义。Kafka 判断一个节点是否活着有两个条件：节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接。如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久。

符合以上条件的节点准确的说应该是“同步中的（in sync）”，而不是模糊的说是“活着的”或是“失败的”。Leader 会追踪所有“同步中”的节点，一旦一个 down 掉了，或是卡住了，或是延时太久，leader 就会把它移除，延时多久算是太久，是由参数 replica.lag.max.messages 决定的而怎样算是卡住了，是由参数 replica.lag.time.max.ms 决定的。

第13章　内存计算框架 Spark

第1节 由 Java 到 Scala

内存计算框架由 Scala 语言编写，其交互式环境也支持直接编写 Scala 表达式来进行数据操作，因此在介绍 Spark 之前，首先来了解一下 Scala 和 Java 的区别。

Scala 是一门多范式的编程语言，一种类似 Java 的编程语言，设计初衷是实现可伸缩的语言、并集成面向对象编程和函数式编程的各种特性。

Scala 具备如下的核心特征：

Scala 中的每个值都是一个对象，包括基本数据类型（即布尔值、数字等）在内，连函数也是对象。

与只支持单继承的语言相比，Scala 具有更广泛意义上的类重用，Scala 允许定义新类的时候重用“一个类中新增的成员定义”。

Scala 还包含了若干函数式语言的关键概念，包括高阶函数、局部套用、嵌套函数、序列解读等等。

Scala 是静态类型的，这就允许它提供泛型类、内部类、甚至多态方法。

Scala 可以与 Java 互操作。用 scalac 这个编译器把源文件编译成 Java 的 class 文件，可以从 Scala 中调用所有的 Java 类库，也同样可以从 Java 应用程序中调用 Scala 的代码。

作为学习 Scala 的第一步，照例首先编写一个标准的 HelloWorld，这个虽然不是很有趣，但是它可以让你对 Scala 有一个最直观的认识而不需要太多关于这个语言的知识。Scala 的 Hello world 看起来像这样：

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

程序的结构对 Java 程序员来说可能很令人怀念：它由一个 main 函数来接受命令行参数，也就是一个 String 数组。这个函数的唯一一行代码把我 们的问候语传递给了一个叫 println 的预定义函数。main 函数不返回值（所以它是一个 procedure method）。所以，也不需要声明返回类型。

对于 Java 程序员比较陌生的是包含了 main 函数的 object 语句。这样的语句定义了一个单例对象：一个有且仅有一个实例的类。object 语 句在定义了一个叫 HelloWorld 的类的同时还定义了一个叫 HelloWorld 的实例。这个实例在第一次使用的时候会进行实例化。

现 main 函数并没有使用 static 修饰符,这是由于静态成员(方法或者变量)在 Scala 中并不存在。Scala 从不定义静态成员，而通过定义单例 object 取而代之。

Scala 的一个强项在于可以很简单的于已有的 Java 代码交互，所有 java.lang 中的类都已经被自动导入了，而其他的类需要显式声明导入。

如果希望对日期进行格式化处理，比如说用法国的格式。Java 类库定义了一系列很有用的类，比如 Date 和 DateFormat。由于 Scala 于 Java 能够进行很好的交互，不需要在 Scala 类库中实现等效的代码，而只需直接吧 Java 的相关类导入就可以了：

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._
object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Scala 的 import 语句看上去与 Java 的非常相似，但是它更加强大。可以使用大括号来导入同一个包里的多个类，就像上面代码中第一行所做的那样。另一个不同点是当导入一个包中所有的类或者符号时，应该使用下划线（_）而不是星号(*)。这是由于星号在 Scala 中是一个有效的标识符(例如作为方法名称)。

第三行的 import 语句导入了 DataFormat 类中的所有成员，这使得静态方法 getDateInstance 和静态变量 LONG 可以被直接引用。

在 main 函数中，首先建立了一个 Java 的 Date 实例。这个实例默认会包含当前时间。接下来使用刚才导入的静态函数 getDateInstance 定义了日期格式。最后将使用 DataFotmat 格式化好的日期打印了出来。最后一行代码显示了 Scala 的一个有趣的语法：只有一个参数的函数可以使用下面这样的表达式来表示：

```
df format now
```

其实就是下面的这个冗长的表达式的简洁写法：

```
df.format(now)
```

Scala 中可以直接继承或者实现 Java 中的接口和类。

Scala 作为一个纯面向对象的语言，于是在 Scala 中万物皆对象，包括数字和函数。在这方面，Scala 于 Java 存在很大不同：Java 区分原生类型（比如 boolean 和 int）和引用类型，并且不能把函数当初变量操纵。

由于数字本身就是对象，所以他们也有方法。事实上平时使用的算数表达式（如下例）：

```
1 + 2 * 3 / x
```

是由方法调用组成的。它等效于下面的表达式，

```
(1).+(((2).*(3))./(x))
```

这也意味着 +，-，*，/ 在 Scala 中也是有效的名称。

函数在 Scala 语言里面也是一个对象，也许这对于 Java 程序员来说这比较令人惊讶。于是把函数作为参数进行传递、把它们存贮在变量中、或者当作另一个函数的返回值都是可能的。把函数当成值进行操作是函数型编程语言的基石。

为了解释为什么吧函数当作值进行操作是十分有用的，可以考虑一个计时器函数。这个函数的目的是每隔一段时间就执行某些操作。那么如何把要做的操作传入计时器呢？可以把它当作一个函数。这种函数对于经常进行用户界面编程的程序员来说是最熟悉的：注册一个回调函数以便在事件发生后得到通知。

在下面的程序中，计时器函数被叫做 oncePerSceond，它接受一个回调函数作为参数。这种函数的类型被写作 () => Unit ，他们不接受任何参数也没有任何返回（Unit 关键字类似于 C/C++中的 void）。程序的主函数调用计时器并传递一个打印某个句子的函数作为回调。换 句话说，这个程序永无止境的每秒打印一个“time flies like an arrow”。

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

可以把这个程序改的更加易于理解。首先定义函数 timeFlies 的唯一目的就是当作传给 oncePerSecond 的参数。这么看来给这种只用一次的函数命名似乎没有什么太大的必要，事实上可以在用到这个函数的时候再定义它。这些可以通过匿名函数在 Scala 中实现，匿名函数顾名 思义就是没有名字的函数。在新版的程序中将会使用一个匿名函数来代替原来的 timeFlise 函数，程序看起来像这样：

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
```

```
    }
}
```

本例中的匿名函数使用了一个箭头（=>）把它的参数列表和代码分开。在这里参数列表是空的，所以在右箭头的左边写上了一对空括号。函数体内容与上面的 timeFlise 是相同的。

详细的 Scala 指导参考 Scala 开发课程教材。

第2节 Spark 入门

Spark 是开源的类 Hadoop MapReduce 的通用并行框架。Spark，拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

Spark 是一种与 Hadoop 相似的开源集群计算环境，但是两者之间还存在一些不同之处，这些有用的不同之处使 Spark 在某些工作负载方面表现得更加优越:Spark 启用了内存分布数据集，除了能够提供交互式查询外，它还可以优化迭代工作负载。

Spark 是在 Scala 语言中实现的，它将 Scala 用作其应用程序框架。与 Hadoop 不同，Spark 和 Scala 能够紧密集成，其中的 Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。

尽管创建 Spark 是为了支持分布式数据集上的迭代作业，但是实际上它是对 Hadoop 的补充，可以在 Hadoop 文件系统中并行运行。

在 Hadoop MR 过程中，中间结果会借助磁盘传递，因此对比计算，大量的 Map-Reduced 作业都受限于 IO。然而对于 ETL、数据整合和清理这样的用例来说，IO 约束并不会产生很大的影响，因为这些场景对数据处理时间往往不会有较高的需求。然而，在现实世界中，同样存在许多对延时要求较为苛刻的用例，比如，对流数据进行处理来做近实时分析。举个例子，通过分析点击流数据做视频推荐，从而提高用户的参与度。在这个用例中，开发者必须在精度和延时之间做平衡。另外就是在大型数据集上进行交互式分析，数据科学家可以在数据集上做 ad-hoc 查询。

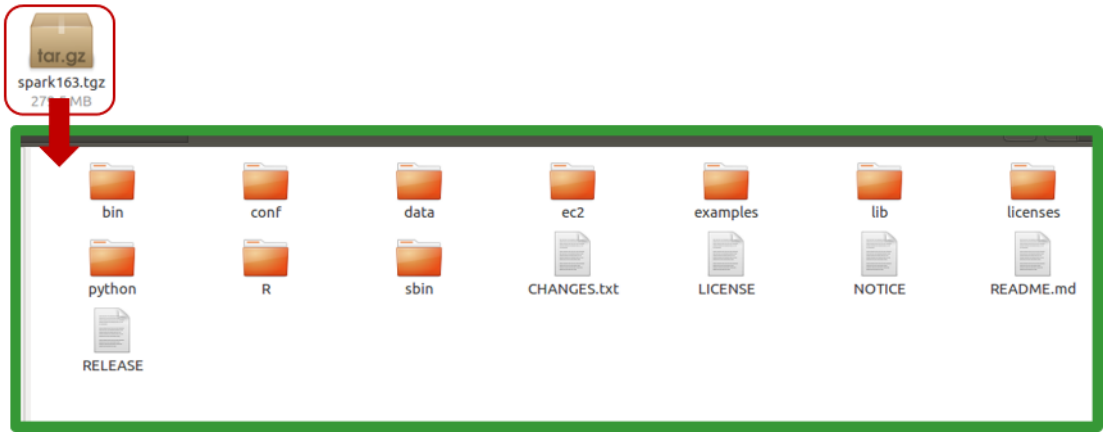
历经数年发展，Hadoop 生态圈中的丰富工具已深受用户喜爱，然而仍然存在的众多问题给使用带来了挑战：

- 每个用例都需要多个不同的技术堆栈来支撑，在不同使用场景下，大量的解决方案往往捉襟见肘。
- 在生产环境中机构往往需要精通数门技术。
- 许多技术存在版本兼容性问题。
- 无法在并行 job 中更快地共享数据。

而通过 Apache Spark，上述问题迎刃而解！

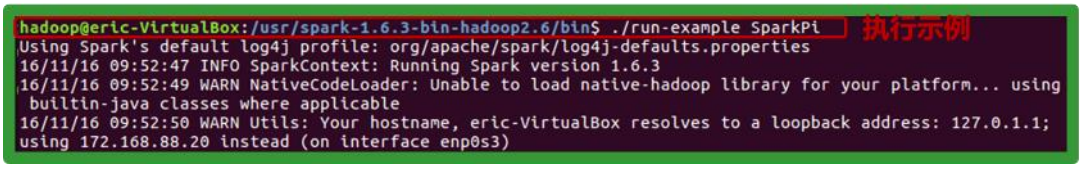
第3节 安装使用 Spark

通过 Spark 官方网站（https://spark.apache.org/downloads.html）下载 Spark 发行包并解压：

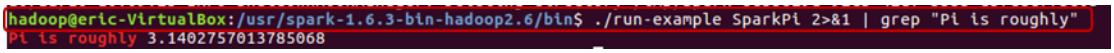


注意：必须安装 Hadoop 才能使用 Spark，但如果使用 Spark 过程中没用到 HDFS，不启动 Hadoop 也是可以的

在 ./examples/src/main 目录下有一些 Spark 的示例程序，有 Scala、Java、Python、R 等语言的版本。可以先运行一个示例程序 SparkPi（即计算 π 的近似值），执行如下命令：



执行时会输出非常多的运行信息，输出结果不容易找到，可以通过 grep 命令进行过滤（命令中的 2>&1 可以将所有的信息都输出到 stdout 中）：

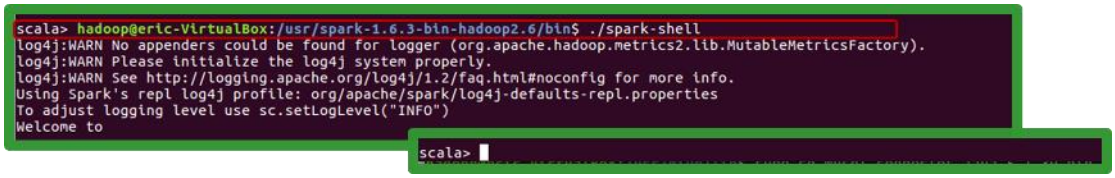


Spark shell 提供了简单的方式来调用 API，也提供了交互的方式来分析数据。Spark Shell 支持 Scala 和 Python，接下来选择使用之前学习的 Scala 进行介绍

Scala 是 Spark 的主要编程语言，如果仅仅是写 Spark 应用，并非一定要用 Scala，用 Java、Python 都是可以的。使用 Scala 的优势是开发效率更高，代码更精简，并且可以通过 Spark Shell 进行交互式实时查询，方便排查问题。

通过执行如下命令启动 Spark Shell：

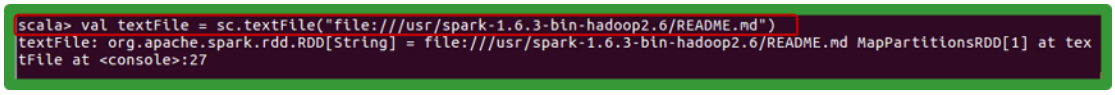
```
spark-shell
```



Spark 的主要抽象是分布式的元素集合，称为 RDD（Resilient Distributed Dataset，弹性分布式数据集），它可被分发到集群各个节点上，进行并行操作。RDDs 可以通过 Hadoop InputFormats 创建（如 HDFS），或者从其他 RDDs 转化而来。

例如从 Spark 的 ./README 文件新建一个 RDD：

```
val textFile = sc.textFile("file:///usr/spark-1.6.3-bin-hadoop2.6/README.md")
```



代码中通过 “file://” 前缀指定读取本地文件。Spark shell 默认是读取 HDFS 中的文件，需要先上传文件到 HDFS 中，否则会有 “org.apache.hadoop.mapred.InvalidInputException: Input path does not exist” 的错误。

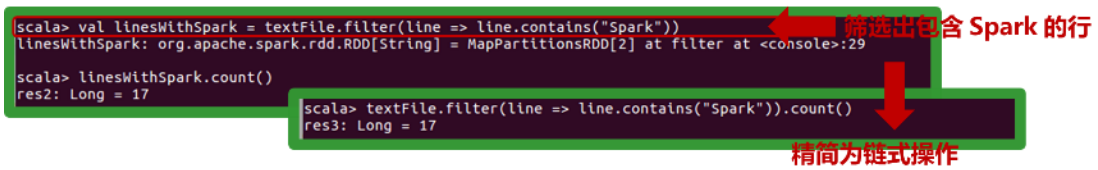
RDDs 支持两种类型的操作：

actions:，在数据集上运行计算后返回值。

transformations:，转换，从现有数据集创建一个新的数据集。

接下来展示一些简单的 Spark 操作。首先是 count() 和 first() actions 操作，count（）操作可以返回数据集的大小，也就是数据集中有多少条数据，和 MapReduce 一样，Spark 默认情况下会将输入源数据文件的每一行数据切分成数据集中的一个元素，而 first（）会直接返回数据集中的第一个元素：

还可以通过 filter transformation 来返回一个新的 RDD，filter 提供了一个元素过滤器，只有符合过滤条件的元素才会被包含到新集合中：



RDD 的 actions 和 transformations 可用在更复杂的计算中，例如通过如下代码可以找到包含单词最多的那一行内容共有几个单词：

```
scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
res4: Int = 14
```

代码首先将每一行内容 map 为一个整数，这将创建一个新的 RDD，并在这个 RDD 中执行 reduce 操作，找到最大的数。map()、reduce() 中的参数是 Scala 的函数字面量，并且可以使用语言特征或 Scala/Java 的库。例如，通过使用 Math.max() 函数（需要导入 Java 的 Math 库），可以使上述代码更容易理解：

```
scala> import java.lang.Math
import java.lang.Math

scala> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
res5: Int = 14
```

Hadoop MapReduce 是常见的数据流模式，在 Spark 中同样可以实现，例如单词统计：

```
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[8] at reduceByKey at <console>:30

scala> wordCounts.collect()
res6: Array[(String, Int)] = Array((package,1), (For,2), (Programs,1), (processing.,1), (Because,1), (The,1), (cluster.
,1), (its,1), ([run,1), (APIs,1), (have,1), (Try,1), (computation,1), (through,1), (several,1), (This,2), (graph,1), (H
ive,2), (storage,1), (["Specifying,1), (To,2), (page](http://spark.apache.org/documentation.html),1), (Once,1), ("yarn"
,1), (prefer,1), (SparkPi,2), (engine,1), (version,1), (file,1), (documentation,,1), (processing,,1), (the,21), (are,1)
, (systems.,1), (params,1), (not,1), (different,1), (refer,2), (Interactive,2), (R,,1), (given.,1), (if,4), (build,3),
(when,1), (be,2), (Tests,1), (Apache,1), (./bin/run-example,2), (programs.,1), (including,3), (Spark.,1), (package.,1),
 (1000).count(),1), (Versions,1), (HDFS,1), (Data.,1), (>>>,1), (programming,1), (T...
scala>
```

如果要在 Spark 中访问 HDFS，首先将文件上传到 HDFS 服务器：

```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264/bin$ ./hadoop fs -put /usr/spark-1.6.3-bin-hadoop2.6/README.md
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264/bin$ ./hadoop fs -ls
Found 8 items
-rw-r--r--   1 hadoop supergroup       3359 2016-11-16 14:14 README.md
drwxr-xr-x   - hadoop supergroup          0 2016-07-05 14:50 external_table
drwxr-xr-x   - hadoop supergroup          0 2016-06-17 14:08 input
drwxr-xr-x   - hadoop supergroup          0 2016-06-28 14:45 output
drwxr-xr-x   - hadoop supergroup          0 2016-07-18 20:13 temp_sequence_files
-rw-r--r--   1 hadoop supergroup         15 2016-06-25 16:11 test
drwxr-xr-x   - hadoop supergroup          0 2016-06-27 10:12 tmp
drwxr-xr-x   - hadoop supergroup          0 2016-11-16 14:14 usr
```

上例中的文本数据文件

接下来加载 HDFS 上的文件构建 RDD：

```
scala> val file = sc.textFile("hdfs://127.0.0.1:9000/user/hadoop/README.md")
file: org.apache.spark.rdd.RDD[String] = hdfs://127.0.0.1:9000/user/hadoop/README.md MapPartitionsRDD[22] at textFile a
t <console>:28

scala> val count = file.flatMap(line=>(line.split(" "))).map(word=>(word,1)).reduceByKey(_+_)
count: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[25] at reduceByKey at <console>:30

scala> count.collect()
res7: Array[(String, Int)] = Array((package,1), (For,2), (Programs,1), (processing.,1), (Because,1), (The,1), (cluster.
,1), (its,1), ([run,1), (APIs,1), (have,1), (Try,1), (computation,1), (through,1), (several,1), (This,2), (graph,1), (H
ive,2), (storage,1), (["Specifying,1), (To,2), (page](http://spark.apache.org/documentation.html),1), (Once,1), ("yarn"
,1), (prefer,1), (SparkPi,2), (engine,1), (version,1), (file,1), (documentation,,1), (processing,,1), (the,21), (are,1)
, (systems.,1), (params,1), (not,1), (different,1), (refer,2), (Interactive,2), (R,,1), (given.,1), (if,4), (build,3),
(when,1), (be,2), (Tests,1), (Apache,1), (./bin/run-example,2), (programs.,1), (including,3), (Spark.,1), (package.,1),
 (1000).count(),1), (Versions,1), (HDFS,1), (Data.,1), (>>>,1), (programming,1), (T...
```

加载HDFS的文件进行操作

Transformations 操作是 Lazy 的，也就是说从一个 RDD 转换生成另一个 RDD 的操作不是马上执行，Spark 在遇到 Transformations 操作时只会记录需要这样的操作，并不会去执行，需要等到有 Actions 操作的时候才会真正启动计算过程进行计算。

Actions 操作会返回结果或把 RDD 数据写到存储系统中。Actions 是触发 Spark 启动计算的动因。

以下是常用的转换，更多转换可以参考 Spark 文档：

转换	作用
reduce(func)	通过函数 func 聚集数据集中的所有元素。Func 函数接受 2 个参数，返回一个值。这个函数必须是关联性的，确保可以被正确的并发执行
collect()	在 Driver 的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用 filter 或者其它操作后，返回一个足够小的数据子集再使用，直接将整个 RDD 集 Collect 返回，很可能会让 Driver 程序 OOM
count()	返回数据集的元素个数
take(n)	返回一个数组，由数据集的前 n 个元素组成。注意，这个操作目前并非在多个节点上，并行执行，而是 Driver 程序所在机器，单机计算所有的元素(Gateway 的内存压力会增大，需要谨慎使用）
first()	返回数据集的第一个元素（类似于 take（1）
saveAsTextFile(path)	将数据集的元素，以 textfile 的形式，保存到本地文件系统，hdfs 或者任何其它 hadoop 支持的文件系统。Spark 将会调用每个元素的 toString 方法，并将它转换为文件中的一行文本
saveAsSequenceFile(path)	将数据集的元素，以 sequencefile 的格式，保存到指定的目录下，本地系统，hdfs 或者任何其它 hadoop 支持的文件系统。RDD 的元素必须由 key-value 对组成，并都实现了 Hadoop 的 Writable 接口，或隐式可以转换为 Writable（Spark 包括了基本类型的转换，例如 Int，Double，String 等等）
foreach(func)	在数据集的每一个元素上，运行函数 func。这通常用于更新一个累加器变量，或者和外部存储系统做交互

以下是常用的操作，更多转换可以参考 Spark 文档：

id	name
1	Charlie Sheen

2	Oliver Stone
3	Michael Douglas
4	Martin Sheen
5	Morgan Freeman

Spark 可以使用 persist 和 cache 方法将任意 RDD 缓存到内存、磁盘文件系统中。缓存是容错的，如果一个 RDD 分片丢失，可以通过构建它的 transformation 自动重构。被缓存的 RDD 被使用的时，存取速度会被大大加速。一般的 executor 内存 60%做 cache， 剩下的 40%做 task。

Spark 中，RDD 类可以使用 cache() 和 persist() 方法来缓存。cache()是 persist()的特例，将该 RDD 缓存到内存中。而 persist 可以指定一个 StorageLevel。

Spark 的不同 StorageLevel ，目的满足内存使用和 CPU 效率权衡上的不同需求。建议通过以下的步骤来进行选择：

如果 RDDs 可以很好的与默认的存储级别(MEMORY_ONLY)契合，就不需要做任何修改了。这已经是 CPU 使用效率最高的选项，它使得 RDDs 的操作尽可能的快；

如果不行，试着使用 MEMORY_ONLY_SER 并且选择一个快速序列化的库使得对象在有比较高的空间使用率的情况下，依然可以较快被访问；

尽可能不要存储到硬盘上，除非计算数据集的函数，计算量特别大，或者它们过滤了大量的数据。否则，重新计算一个分区的速度，和与从硬盘中读取基本差不多快；

如果想有快速故障恢复能力，使用复制存储级别(例如：用 Spark 来响应 web 应用的请求)。所有的存储级别都有通过重新计算丢失数据恢复错误的容错机制，但是复制存储级别可以让你在 RDD 上持续的运行任务，而不需要等待丢失的分区被重新计算；

如果你想要定义你自己的存储级别(比如复制因子为 3 而不是 2)，可以使用

StorageLevel 单例对象的 apply()方法；

在不使用 cached RDD 的时候，及时使用 unpersist 方法来释放它。

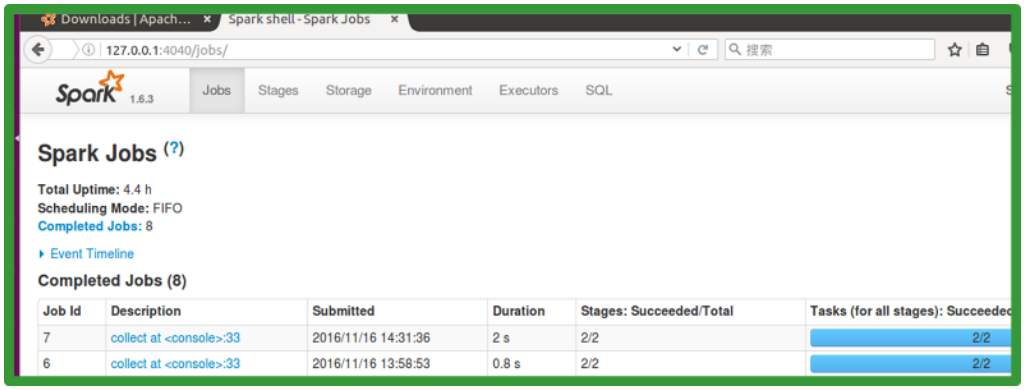
并行化集合是通过调用 SparkContext 的 parallelize 方法，在一个已经存在的 Scala 集合上创建的一个 Seq 对象。集合的对象将会被拷贝，创建出一个可以被并行操作的分布式数据集。例如根据能启动的 executor 的数量来进行切分多个 slice，每一个 slice 启动一个 Task 来进行处理：

```
val rdd = sc.parallelize(Array(1 to 10))
```

指定 partition 的数量

```
val rdd = sc.parallelize(Array(1 to 10), 5)
```

启动 Spark 后可以通过 WEB UI 查看运行状态，默认端口为 4040：



通过 parallelize 方法定义一个从 1~10 的数据集，然后通过 map(_*2)对数据集每个数乘以 2，接着通过 filter(_%3==0)过滤被 3 整除的数字，最后使用 toDebugString 显示 RDD 的 LineAge，并通过 collect 计算出最终的结果：

以下语句和 collect 一样，都会触发作业运行：

```
num.reduce (_ + _)
num.take(5)
num.first
num.count
num.take(5).foreach(println)
```

运行的情况可以通过页面进行监控，在 Spark Stages 页面中可以看到运行

的详细情况，包括运行的 Stage id 号、Job 描述、提交时间、运行时间、Stage 情况等，可以点击作业描述查看更加详细的情况。

通过 parallelize 方法定义 K-V 键值对数据集合，通过 sortByKey()进行按照 Key 值进行排序，然后通过 collect 方法触发作业运行得到结果。groupByKey()为按照 Key 进行归组，reduceByKey(_+_)为按照 Key 进行累加：

```
scala> val kv1=sc.parallelize(List(("A",1),("B",2),("C",3),("A",4),("B",5)))
kv1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[29] at parallelize at <console>:28

scala> kv1.sortByKey().collect
res11: Array[(String, Int)] = Array((A,1), (A,4), (B,2), (B,5), (C,3))

scala> kv1.groupByKey().collect
res12: Array[(String, Iterable[Int])] = Array((B,CompactBuffer(2, 5)), (A,CompactBuffer(1, 4)), (C,CompactBuffer(3)))

scala> kv1.reduceByKey(_+_).collect
res13: Array[(String, Int)] = Array((B,7), (A,5), (C,3))
```

下例进行了 distinct、union、join 和 cogroup 等操作中涉及到 Shuffle 过程：

```
val
kv2=sc.parallelize(List(("A",4),("A",4),("C",3),("A",4),("B",5)))
kv2.distinct.collect
kv1.union(kv2).collect

val kv3=sc.parallelize(List(("A",10),("B",20),("D",30)))
kv1.join(kv3).collect
kv1.cogroup(kv3).collect
```

接下来使用一个日志分析的案例来展示在 Spark 中常见的数据处理过程。嘉定日志的格式如下约束：文件中字段分别为：访问时间\t 用户 ID\t[查询词]\t 该 URL 在返回结果中的排名\t 用户点击的顺序号\t 用户点击的 URL：

打开(O) search.txt /usr

```
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Tom     Spark   4       2       http://abc.com
2016-11-16      Jerry   Spark   1       2       http://abc.com
2016-11-16      Eric    Hadoop  3       1       http://abc.com
2016-11-16      Eric    Hive    5       1       http://abc.com
2016-11-16      Eric    Sqoop   4       1       http://abc.com
2016-11-16      Eric    BigData 3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
2016-11-16      Eric    Spark   3       1       http://abc.com
```

查询搜索结果排名第 1 点击次序排在第 2 的数据：

```
scala> val rdd1 = sc.textFile("hdfs://127.0.0.1:9000/user/hadoop/search.txt")
rdd1: org.apache.spark.rdd.RDD[String] = hdfs://127.0.0.1:9000/user/hadoop/search.txt MapPartitionsRDD[38] at textFile
at <console>:28

scala> val rdd2=rdd1.map(_.split("\t")).filter(_.length==6)
rdd2: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[40] at filter at <console>:30

scala> rdd2.count()
res15: Long = 14

scala> val rdd3=rdd2.filter(_(3).toInt==1).filter(_(4).toInt==2)
rdd3: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[42] at filter at <console>:32

scala> rdd3.count()
res16: Long = 1

scala> rdd3.toDebugString
res17: String =
(1) MapPartitionsRDD[42] at filter at <console>:32 []
 |  MapPartitionsRDD[41] at filter at <console>:32 []
 |  MapPartitionsRDD[40] at filter at <console>:30 []
 |  MapPartitionsRDD[39] at map at <console>:30 []
 |  hdfs://127.0.0.1:9000/user/hadoop/search.txt MapPartitionsRDD[38] at textFile at <console>:28 []
 |  hdfs://127.0.0.1:9000/user/hadoop/search.txt HadoopRDD[37] at textFile at <console>:28 []
```

该命令运行的过程如下：

- 首先使用 textFile()读入 search.txt 文件，读入后由 HadoopRDD 转变为 MadppedRDD；
- 然后通过 rdd1.map(_.split("\t"))对读入数据使用\t 分隔符进行拆分，拆分后 RDD 类型不变即为 MadppedRDD，对这些拆分后的数据使用 filter(_.length==6)过滤每行为 6 个字段的数据，这时数据变为 FilteredRDD；
- 运行 rdd2.count()启动对 rdd2 计数的作业，通过运行结果可以看到该数据集为 100 条；
- rdd2.filter(_(3).toInt==1).filter(_(4).toInt==2)表示对 rdd2 的数据的第 4 个字段搜索结果排名第一，第 5 个字段点击次序排在第二的数据进行过滤，通过 count()方法运行作业得出最终的结果；
- 使用 toDebugString 可以查看 rdd3 的 RDD 详细变换过程

用户查询次数排行榜并把结果保存在 HDFS 中：

```
scala> val rdd4 = rdd2.map(x=>(x(1),1)).reduceByKey(_+_).map(x=>(x._2,x._1)). sortByKey(false).map(x=>(x._2,x._1))
rdd4: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[47] at map at <console>:32

scala> rdd4.toDebugString
res18: String =
(1) MapPartitionsRDD[47] at map at <console>:32 []
 |  ShuffledRDD[46] at sortByKey at <console>:32 []
 +-(1) MapPartitionsRDD[45] at map at <console>:32 []
    |  ShuffledRDD[44] at reduceByKey at <console>:32 []
    +-(1) MapPartitionsRDD[43] at map at <console>:32 []
       |  MapPartitionsRDD[40] at filter at <console>:30 []
       |  MapPartitionsRDD[39] at map at <console>:30 []
       |  hdfs://127.0.0.1:9000/user/hadoop/search.txt MapPartitionsRDD[38] at textFile at <console>:28 []
       |  hdfs://127.0.0.1:9000/user/hadoop/search.txt HadoopRDD[37] at textFile at <console>:28 []

scala> rdd4.saveAsTextFile("hdfs://127.0.0.1:9000/spark/output1")
```

```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264$ ./bin/hadoop fs -cat /spark/output1/part-00000
(Eric,12)
(Tom,1)
(Jerry,1)
```

rdd4 的生成比较复杂：

轴线 map(x=>(x(1),1))是获取每行的第二个字段（用户 Session）计数为 1，然后 reduceByKey(_+_)是安排 Key 进行累和，即按照用户 Session 号进行计数

求查询次数。

其次 map(x=>(x._2,x._1))是把 Key 和 Value 位置互换，为后面排序提供条件，使用 sortByKey(false)对数据进行按 Key 值进行倒排，此时需要注意的是 Key 为查询次数，最后通过 map(x=>(x._2,x._1)再次交换 Key 和 Value 位置，得到了（用户 Session 号，查询次数）结果。

第4节 Spark Streaming

Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括 Kafk、Flume、Twitter、ZeroMQ、Kinesis 以及 TCP Sockets，从数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘。在“One Stack rule them all”的基础上，还可以使用 Spark 的其他子框架，如集群学习、图计算等，对流数据进行处理。

Spark 的各个子框架，都是基于核心 Spark 的，Spark Streaming 在内部的处理机制是，接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，然后通过 Spark Engine 处理这些批数据，最终得到处理后的一批批结果数据。

对应的批数据，在 Spark 内核对应一个 RDD 实例，因此，对应流数据的 DStream 可以看成是一组 RDDs，即 RDD 的一个序列。通俗点理解的话，在流数据分成一批一批后，通过一个先进先出的队列，然后 Spark Engine 从该队列中依次取出一个个批数据，把批数据封装成一个 RDD，然后进行处理，这是一个典型的生产者消费者模型，对应的就有生产者消费者模型的问题，即如何协调生产速率和消费速率。

SparkStream 术语定义如下：

离散流（discretized stream）或 DStream：这是 Spark Streaming 对内部持续的实时数据流的抽象描述。

批数据（batch data）：将实时流数据以时间片为单位进行分批，将流处理转化为时间片数据的批处理。

时间片或批处理时间间隔（ batch interval）：以时间片作为我们拆分流数据的依据。一个时间片的数据对应一个 RDD 实例。

窗口长度（window length）：一个窗口覆盖的流数据的时间长度。必须是批处理时间间隔的倍数。

滑动时间间隔：前一个窗口到后一个窗口所经过的时间长度。必须是批处理时间间隔的倍数。

Input DStream:一个 input DStream 是一个特殊的 DStream，将 Spark Streaming 连接到一个外部数据源来读取数据。

作为构建于 Spark 之上的应用框架，Spark Streaming 承袭了 Spark 的编程

风格，对于已经了解 Spark 的用户来说能够快速地上手。接下来以 Spark Streaming 官方提供的 WordCount 代码为例来介绍 Spark Streaming 的使用方式：

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._

// Create a local StreamingContext with two working thread and
batch interval of 1 second.
// The master requires 2 cores to prevent from a starvation
scenario.
val conf = new
SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

// Create a DStream that will connect to hostname:port, like
localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split(" "))
import org.apache.spark.streaming.StreamingContext._
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this
DStream to the console
wordCounts.print()
ssc.start()                             // Start the computation
ssc.awaitTermination()    // Wait for the computation to terminate
```

上面程序的执行流程解析如下：

- **创建 StreamingContext 对象**

使用 Spark Streaming 就需要创建 StreamingContext 对象。创建 StreamingContext 对象所需的参数与 SparkContext 基本一致，包括指明 Master，设定名称。需要注意的是参数 Seconds(1)，Spark Streaming 需要指定处理数据的时间间隔，如上例所示的 1s，那么 Spark Streaming 会以 1s 为时间窗口进行数据处理。此参数需要根据用户的需求和集群的处理能力进行适当的设置

- **创建 InputDStream**

Spark Streaming 需要指明数据源。如上例所示的 socketTextStream，Spark

Streaming 以 socket 连接作为数据源读取数据。当然 Spark Streaming 支持多种不同的数据源，包括 Kafka、 Flume、HDFS/S3、Kinesis 和 Twitter 等数据源；

- **操作 DStream**

对于从数据源得到的 DStream，用户可以在其基础上进行各种操作，如上例所示的操作就是一个典型的 WordCount 执行流程：对于当前时间窗口内从数据源得到的数据首先进行分割，然后利用 Map 和 ReduceByKey 方法进行计算，当然最后还有使用 print()方法输出结果；

- **启动 Spark Streaming**

之前所作的所有步骤只是创建了执行流程，程序没有真正连接上数据源，也没有对数据进行任何操作，只是设定好了所有的执行计划，当 ssc.start()启动后程序才真正进行所有预期的操作。

第5节 SparkSQL

SparkSQL 的前身是 Shark。为了给熟悉 RDBMS 但又不理解 MapReduce 的技术人员提供快速上手的工具，Hive 应运而生，它是当时唯一运行在 Hadoop 上的 SQL-on-Hadoop 工具。但是 MapReduce 计算过程中大量的中间磁盘落地过程消耗了大量的 I/O，降低的运行效率，为了提高 SQL-on-Hadoop 的效率，大量的 SQL-on-Hadoop 工具开始产生，其中表现较为突出的是：MapR 的 Drill、Cloudera 的 Impala 和 Shark。

其中 Shark 是伯克利实验室 Spark 生态环境的组件之一，它修改了内存管理、物理计划、执行三个模块，并使之能运行在 Spark 引擎上，从而使得 SQL 查询的速度得到 10-100 倍的提升。

随着 Spark 的发展，Shark 对于 Hive 的太多依赖（如采用 Hive 的语法解析器、查询优化器等等），制约了 Spark 的 One Stack Rule Them All 的既定方针，制约了 Spark 各个组件的相互集成，所以提出了 SparkSQL 项目。SparkSQL 抛弃原有 Shark 的代码，汲取了 Shark 的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新开发了 SparkSQL 代码；由于摆脱了对 Hive 的依赖性，SparkSQL 无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。

数据兼容方面，不但兼容 Hive，还可以从 RDD、parquet 文件、JSON 文件中获取数据，未来版本甚至支持获取 RDBMS 数据以及 cassandra 等 NOSQL 数据；

性能优化方面，除了采取 In-Memory Columnar Storage、byte-code generation 等优化技术外、将会引进 Cost Model 对查询进行动态评估、获取最佳物理计划等等；

组件扩展方面，无论是 SQL 的语法解析器、分析器还是优化器都可以重新定义，进行扩展。

SparkSQL 的表数据在内存中存储不是采用原生态的 JVM 对象存储方式，而是

采用内存列存储，该存储方式无论在空间占用量和读取吞吐率上都占有很大优势。

对于原生态的 JVM 对象存储方式，每个对象通常要增加 12-16 字节的额外开销，对于一个 270MB 的 TPC-H lineitem table 数据，使用这种方式读入内存，要使用 970MB 左右的内存空间（通常是 2～5 倍于原生数据空间）；另外，使用这种方式，每个数据记录产生一个 JVM 对象，如果是大小为 200B 的数据记录，32G 的堆栈将产生 1.6 亿个对象，这么多的对象，对于 GC 来说，可能要消耗几分钟的时间来处理（JVM 的垃圾收集时间与堆栈中的对象数量呈线性相关）。显然这种内存存储方式对于基于内存计算的 Spark 来说，很昂贵也负担不起。

对于内存列存储来说，将所有原生数据类型的列采用原生数组来存储，将 Hive 支持的复杂数据类型（如 array、map 等）先序化后并接成一个字节数组来存储。这样，每个列创建一个 JVM 对象，从而导致可以快速的 GC 和紧凑的数据存储；额外的，还可以使用低廉 CPU 开销的高效压缩方法（如字典编码、行长度编码等压缩方法）降低内存开销；更有趣的是，对于分析查询中频繁使用的聚合特定列，性能会得到很大的提高，原因就是这些列的数据放在一起，更容易读入内存进行计算。

在数据库查询中有一个昂贵的操作是查询语句中的表达式，主要是由于 JVM 的内存模型引起的。比如如下一个查询：

```
SELECT a + b FROM table
```

在这个查询里，如果采用通用的 SQL 语法途径去处理，会先生成一个表达式树，在物理处理这个表达式树的时候，将会如图所示的 7 个步骤：

- 调用虚函数 Add.eval()，需要确认 Add 两边的数据类型
- 调用虚函数 a.eval()，需要确认 a 的数据类型
- 确定 a 的数据类型是 Int，装箱
- 调用虚函数 b.eval()，需要确认 b 的数据类型
- 确定 b 的数据类型是 Int，装箱
- 调用 Int 类型的 Add
- 返回装箱后的计算结果

其中多次涉及到虚函数的调用，虚函数的调用会打断 CPU 的正常流水线处理，减缓执行。

Spark1.1.0 在 catalyst 模块的 expressions 增加了 codegen 模块，如果使用动态字节码生成技术（配置 spark.sql.codegen 参数），SparkSQL 在执行物理计划的时候，对匹配的表达式采用特定的代码，动态编译，然后运行。

要使用 SparkSQL，首先需要和 Hive 的 Metastore 集成：将 Hive 的 conf 文件夹下的 hive-site.xml 复制到 Spark 的 conf 文件夹，然后将 MySQL 的 jdbc 驱动复制到 Spark 的 lib 文件夹（用于访问 Hive 的 Matestore）：

```
hadoop@eric-VirtualBox:/usr/hive/conf$ sudo cp hive-site.xml /usr/spark-1.6.3-bi
n-hadoop2.6/conf/
```

```
hadoop@eric-VirtualBox:/usr/hive/lib$ sudo cp mysql-connector-java-5.1.39-bin.ja
r  /usr/spark-1.6.3-bin-hadoop2.6/lib/
```

为了防止在启动时发生无法创建连接的错误，在启动 SparkSQL 时加入 MySQL JDBC 驱动路径：

```
hadoop@eric-VirtualBox:/usr/spark-1.6.3-bin-hadoop2.6$ ./bin/spark-sql --driver-class-path lib/mysql-connector-java-5.1
.39-bin.jar
```

```
16/11/16 16:31:13 INFO SessionState: Created local directory: /tmp/hadoop/ea793a6c-b2da-43bd-9216-73d895951c3d
16/11/16 16:31:14 INFO SessionState: Created HDFS directory: /tmp/hive/hadoop/ea793a6c-b2da-43bd-9216-73d895951c3d/_tmp
_space.db
SET spark.sql.hive.version=1.2.1
SET spark.sql.hive.version=1.2.1
spark-sql>
```

开启 SparkSQL，即可以使用和 Hive 类似的方法进行操作（所不同的是，Hive 的数据操作被翻译成了 MapReduce 任务，而 SparkSQL 的数据操作在 Spark 中执行），例如查询表（可以直接使用以前 Hive 创建的数据）：

```
spark-sql> show tables;
```

```
sales    false
temp    false
testmysql       false16/11/16 16:37:45 IN
67.0 ms 67.0 ms 67.0 ms

things  false
16/11/16 16:37:45 INFO StatsReportListene
00000, min: 1265.000000)
uses    false
view_things     false
view_things1    false
```

查询数据：

```
spark-sql> select * from sales;
```

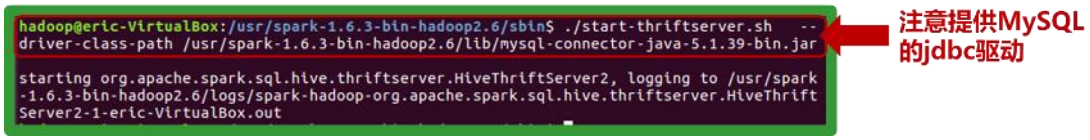
```
JIM     1
TOM     2
ERIC    1
JERRY   3
LEE     2
JACK    0
```

ThriftServer 是一个 JDBC/ODBC 接口，用户可以通过 JDBC/ODBC 连接 ThriftServer 来访问 SparkSQL 的数据。

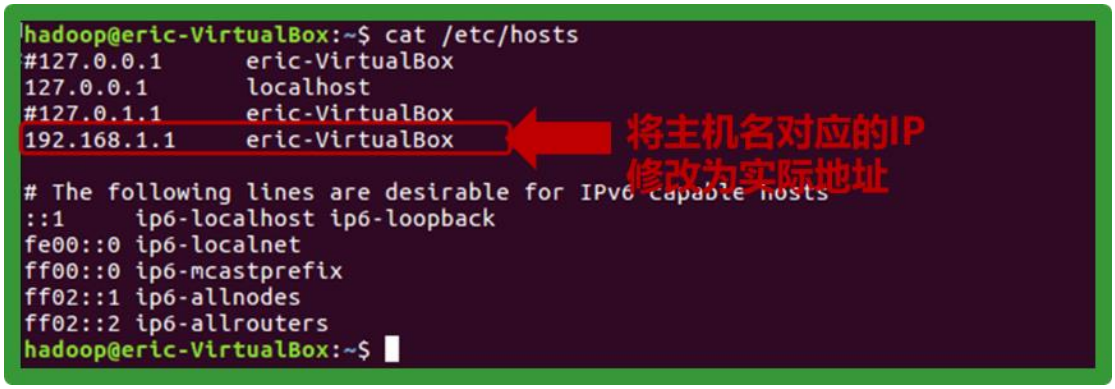
ThriftServer 在启动的时候，会启动了 SparkSQL 的应用程序，而通过 JDBC/ODBC 连接进来的客户端共同分享这个 SparkSQL 应用程序的资源，也就是说不同的用户之间可以共享数据；ThriftServer 启动时还开启一个侦听器，等待 JDBC 客户端的连接和提交查询。所以，在配置 ThriftServer 的时候，至少要配置 ThriftServer 的主机名和端口，如果要使用 Hive 数据的话，还要提供 Hive Metastore 的 uris。

启动 ThriftServer 的步骤如下：

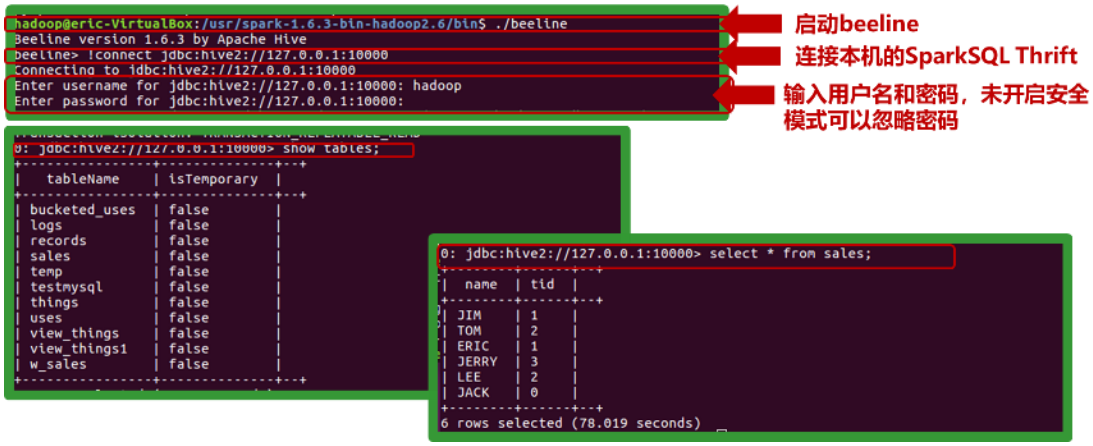
- 启动 HDFS
- 启动 Hive 的 Matestore
- 启动 Spark 服务
- 最后启动 SparkSQL Thrift：



注意，如果系统为 Ubuntu，默认情况下 hosts 文件中会将主机名对应的 IP 设置为 127.0.1.1（此 IP 不可访问），因此会导致服务启动不成功，需要将主机名对应的 IP 修改为实际的 IP 地址：



利用 Beeline 测试 Thrift：



使用和 Hive 相同的 JDBC 代码访问 SparkSQL：

import java.sql.Connection;
public class TestJDBCConnection {
/**
* @param args
*/
public static void main(String[] args) {
try {
Class.forName("org.apache.hive.jdbc.HiveDriver");
Connection con = DriverManager.getConnection(
"jdbc:hive2://127.0.0.1:10000/default", "hadoop", "wall");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM sales");
while (rs.next()) {
System.out.println(rs.getString(1) + "\t" + rs.getInt(2));
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
Markers
Console
Properties
Servers
Data Source Expl
执行结果
<terminated> TestJDBCConnection [Java Application] /usr/jdk/jdk18/bin/java
JIM 1
2016-11-18 10:04:42 DEBUG HiveQueryResultSet:389 - Fetched row string:
TOM 2
2016-11-18 10:04:42 DEBUG HiveQueryResultSet:389 - Fetched row string:
ERIC 1
2016-11-18 10:04:42 DEBUG HiveQueryResultSet:389 - Fetched row string:
JERRY 3
2016-11-18 10:04:42 DEBUG HiveQueryResultSet:389 - Fetched row string:
LEE 2
2016-11-18 10:04:42 DEBUG HiveQueryResultSet:389 - Fetched row string:
JACK 0

第14章 Oozie

第1节 Oozie 入门介绍

Oozie 是一个工作流引擎服务器,用于运行 Hadoop Map/Reduce 和 Pig 任务工作流.同时 Oozie 还是一个 Java Web 程序,运行在 Java Servlet 容器中,如 Tomcat。

Oozie 使用数据库来存储以下内容：工作流定义和当前运行的工作流实例，包括实例的状态和变量。

Oozie 工作流中拥有多个 Action,如 Hadoop Map/Reuce job,Hadoop Pig job 等。所有的 Action 以有向无环图(DAG Direct Acyclic Graph)的模式部署运行.所以在 Action 的运行步骤上是有方向的,只能上一个 Action 运行完成后才能运行下一个 Action。

Oozie 工作流通过 HPDL(一种通过 XML 自定义处理的语言,类似 JBOSS JBPM 的 JPDL)来构造。

Oozie 工作流中的 Action 在远程系统运行 (如 Hadoop,Pig 服务器上)。一旦 Action 完成,远程服务器将回调 Oozie 的接口并通知 Action 已经完成,这时 Oozie 又会以同样的方式执行工作流中的下一个 Action,直到工作流中所有 Action 都完成(完成包括失败)。

Oozie 工作流中包含可控制的工作流节点(control flow node)和 Action 节点(action node)。

- Control flow node

可以理解为 Oozie 的语法,比如可以定义开始(start),结束(end),失败(fail)节点，开始节点就表示从该节点开始运行，同时也提供一种机制去控制工作流的执行过程,如选择(decision),并行(fork),join 节点。

- Action

Oozie 工作流提供各种类型的 Action 用于支持不同的需要,如 Hadoop Map/Reduce,Hadoop File System,Pig,SSH,HTTP,Email,Java,以及 Oozie 子流程，Oozie 也支持自定义扩展以上各种类型的 Action 。

Oozie 工作流允许自定义参数,如${inputDir}。

所有由动作节点触发的计算和处理任务都不在 Oozie 之中--它们是由 Hadoop 的 Map/Reduce 框架执行的。这种方法让 Oozie 可以支持现存的 Hadoop 用于负载平衡、灾难恢复的机制。

任务主要是异步执行的(只有文件系统动作例外，它是同步处理的)。这意味着对于大多数工作流动作触发的计算或处理任务的类型来说，在工作流操作转换到工作流的下一个节点之前都需要等待，直到计算或处理任务结束了之后才能够

继续。

Oozie 可以通过两种不同的方式来检测计算或处理任务是否完成，也就是回调和轮询。当 Oozie 启动了计算或处理任务的时候，它会为任务提供唯一的回调 URL，然后任务会在完成的时候发送通知给特定的 URL。在任务无法触发回调 URL 的情况下（可能是因为任何原因，比方说网络闪断），或者当任务的类型无法在完成时触发回调 URL 的时候，Oozie 有一种机制，可以对计算或处理任务进行轮询，从而保证能够完成任务。

一些工作流是根据需要触发的，但是大多数情况下，有必要基于一定的时间段和(或)数据可用性和(或)外部事件来运行它们。Oozie 协调系统(Coordinator system）让用户可以基于这些参数来定义工作流执行计划。

Oozie 协调程序可以以谓词的方式对工作流执行触发器进行建模，那可以指向数据、事件和（或）外部事件。工作流作业会在谓词得到满足的时候启动。

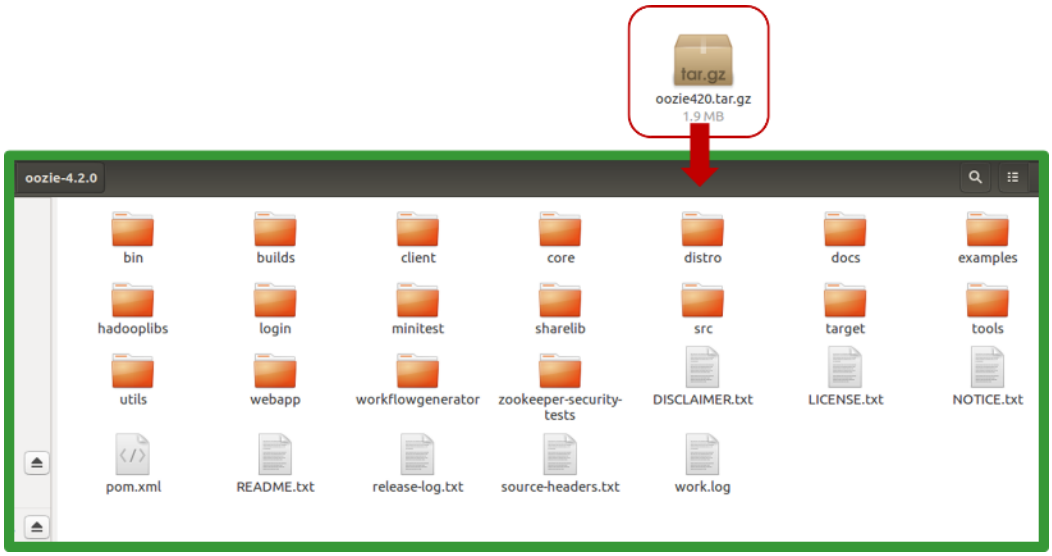
经常需要连接定时运行、但时间间隔不同的工作流操作。多个随后运行的工作流的输出会成为下一个工作流的输入。把这些工作流连接在一起，会让系统把它作为数据应用的管道来引用。Oozie 协调程序支持创建这样的数据应用管道。

第2节 安装 Oozie

安装 Ooize 需要依赖 Maven，在各个 Linux 的发行版中可以使用各自附带的包管理器来安装 Maven，如在 Ubuntu 中，可以执行如下命令：

```
sudo apt-get install maven
```

接下来再官网中下载 Oozie 安装包，并解压：



oozie 的默认 Maven 依赖库中有一个服务商已经停止服务，因此需要将 oozie pom.xml 中的 id 为 CodeHaus repository 的库替换为以下内容：

```
<repository>
  <id>Codehaus repository</id>
  <name>codehaus-mule-repo</name>
  <url>https://repository-master.mulesoft.org/nexus/content/groups/public/
  </url>
  <layout>default</layout>
</repository>
```

编译版本修改为 1.7：

```
<targetJavaVersion>1.7</targetJavaVersion>
```

注意：JDK8 环境中会出现 Javadoc 编译生成错误，可以更换为 1.7 版本的 JDK。需要将修改 Ooize 中的文件夹权限，否则会出现错误。

在 bin 中执行生成 Shell：

```
hadoop@eric-VirtualBox:/usr/oozie-4.2.0/bin$ ./mkdistro.sh -DskipTests
```

JDK8 中可以在 distro 中直接调用 Maven 命令进行生成，需要提供跳过 Javadoc 生成的参数：

```
hadoop@eric-VirtualBox:/usr/oozie-4.2.0/bin$ ./mkdistro.sh  -DskipTests
```

跳过单元测试

利用 Maven 编译后的结果（ooize_home\distro\target）：

生成结果

获取该压缩包并解压：

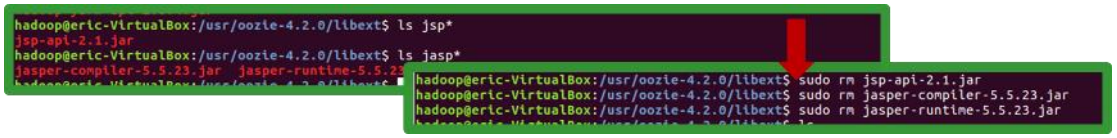
```
hadoop@eric-VirtualBox:/usr/oozie-4.2.0/distro/target$ sudo cp oozie-4.2.0-distr
o.tar.gz /usr/oozie420-dist.tar.gz
```

```
hadoop@eric-VirtualBox:/usr$ sudo tar -zxvf oozie420-dist.tar.gz
```

创建存放库的文件夹（libext）：

将 Hadoop 相关库拷贝到该文件夹：

注意将以下几个包删除（由于 ooize 自带 Tomcat 因此不删除的话可能导致包冲突）：



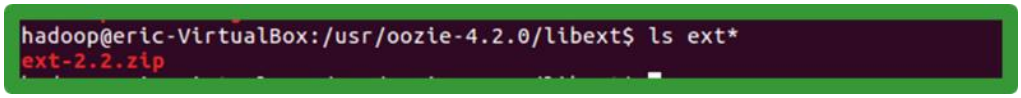
将 MySQL 的 JDBC 驱动包加入库文件夹（用于 ooize 连接 MySQL）：

创建 ooize 的数据库：

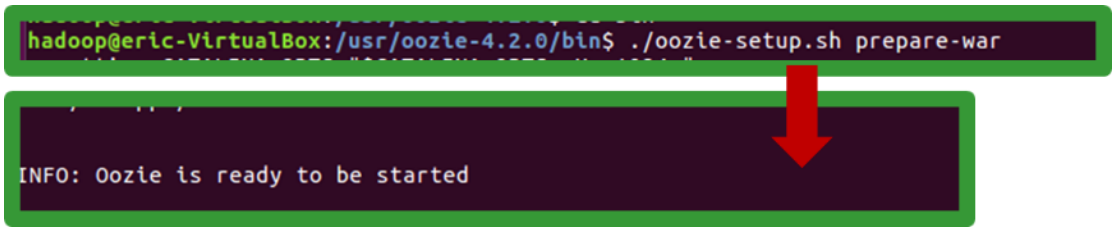
如果不保险，可以使用 ooize 生成库表结构：

```
bin/ooziedb.sh create -sqlfile oozie.sql  -run
```

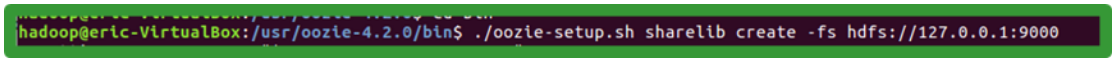
将 EXT2.2 的包加入库文件夹（ooize 在代码中硬编码了 ext 库的文件名，因此一定要是本版本和文件名）：



生成 ooize 的 war 包：



将 ooize 的库上传到 HDFS 对应的位置：



修改 ooize 的 conf/hadoop-conf 中的 core-site.xml，加入 Hadoop 和 yarn

的相关属性配置：

```
 <property>
     <name>mapreduce.framework.name</name>
     <value>yarn</value>
 </property>
 <property>
     <name>mapreduce.jobhistory.webapp.address</name>
     <value>127.0.0.1:19888</value>
 </property>
 <property>
     <name>mapreduce.jobhistory.intermediate-done-dir</name>
     <value>/user/hadoop/tmp</value>
</property>
<property>
     <name>mapreduce.jobhistory.done-dir</name>
     <value>/user/hadoop/done</value>
</property>
<property>
     <name>yarn.resourcemanager.address</name>
     <value>127.0.0.1:8032</value>
</property>
<property>
   <name>yarn.resourcemanager.scheduler.address</name>
    <value>127.0.0.1:8030</value>
</property>
```

修改 Hadoop 的 core-site.xml，加入用户和主机声明：

```
<property>
     <name>hadoop.proxyuser.hadoop.hosts</name>
     <value>*</value>
     <description>The superuser can connect only from host1 and host2 to impersonate a user</description>
 </property>

<property>
       <name>hadoop.proxyuser.root.hosts</name>
       <value>*</value>
   </property>
   <property>
       <name>hadoop.proxyuser.root.groups</name>
       <value>*</value>
   </property>
```

在 mapred-site.xml 中确认 yarn 的相关信息：

```
<configuration>
  <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
<property>
    <name>mapreduce.jobtracker.http.address</name>
    <value>127.0.0.1:50030</value>
</property>
<property>
    <name>mapreduce.jobhistory.address</name>
    <value>127.0.0.1:10020</value>
</property>
<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>127.0.0.1:19888</value>
</property>
    <property>
            <name>mapred.job.tracker</name>
            <value>http://127.0.0.1:9001</value>
    </property>

</configuration>
```

修改 conf/ooize-site.xml 文件，加入 MySQL 及 Hadoop 配置信息：

```
<!--mysql作为元数据存放的数据库-->
<property>
    <name>oozie.service.JPAService.jdbc.driver</name>
    <value>com.mysql.jdbc.Driver</value>
    <description>
        JDBC driver class.
    </description>
</property>
<property>
    <name>oozie.service.JPAService.jdbc.url</name>
    <value>jdbc:mysql://127.0.0.1:3306/oozie</value>
    <description>
        JDBC URL.
    </description>
</property>
<property>
    <name>oozie.service.JPAService.jdbc.username</name>
    <value>root</value>
    <description>
        DB user name.
    </description>
</property>
<property>
    <name>oozie.service.JPAService.jdbc.password</name>
    <value>root</value>
    <description>
        DB user password.
        IMPORTANT: if password is emtpy leave a 1 space s
        if empty Configuration assumes it is NULL.
    </description>
</property>
```

```
<!--设置Hadoop的配置文件的路径-->
<property>
    <name>oozie.service.HadoopAccessorService.hadoop.configurations</name>
    <value>*=/usr/hadoop/hadoop264/etc/hadoop</value>
    <description>
        Comma separated AUTHORITY=HADOOP_CONF_DIR, where AUTHORITY is the HOST:
        the Hadoop service (JobTracker, YARN, HDFS). The wildcard '*' configura
        used when there is no exact match for an authority. The HADOOP_CONF_DIR
        the relevant Hadoop *-site.xml files. If the path is relative is looked
        the Oozie configuration directory; though the path can be absolute (i.e
        to Hadoop client conf/ directories in the local filesystem.
    </description>
</property>
<!--设置Spark的配置文件的路径-->
<property>
    <name>oozie.service.SparkConfigurationService.spark.configurations</name>
    <value>*=/usr/spark-1.6.3-bin-hadoop2.6/conf</value>
    <description>
        Comma separated AUTHORITY=SPARK_CONF_DIR, where AUTHORITY is the HOST:P
        the ResourceManager of a YARN cluster. The wildcard '*' configuration i
        used when there is no exact match for an authority. The SPARK_CONF_DIR
        the relevant spark-defaults.conf properties file. If the path is relati
        the Oozie configuration directory; though the path can be absolute.  Th
        when the Spark master is set to either "yarn-client" or "yarn-cluster".
    </description>
</property>
```

启动 ooize:

```
hadoop@eric-VirtualBox:/usr/oozie-4.2.0/bin$ ./oozied.sh start
```

查看 ooize 启动状态：

```
hadoop@eric-VirtualBox:/usr/oozie-4.2.0/bin$ ./oozie admin -oozie http://localhost:11000/oozie -status
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
System mode: NORMAL
```

NORMAL说明服务启动正常

启动 Hadoop JobHistory 服务：

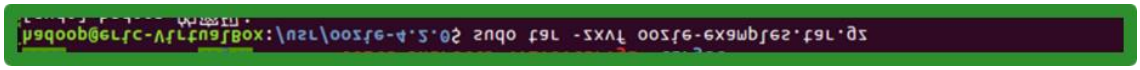
```
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264/sbin$ ./mr-jobhistory-daemon.sh start historyserver
starting historyserver, logging to /usr/hadoop/hadoop264/logs/mapred-hadoop-historyserver-eric-VirtualBox.out
hadoop@eric-VirtualBox:/usr/hadoop/hadoop264/sbin$ jps
6511 Jps
3172 DataNode
3328 SecondaryNameNode
3591 Bootstrap
5528 JobHistoryServer
3063 NameNode
```

可以通过 WEB UI 查看 ooize 状态：

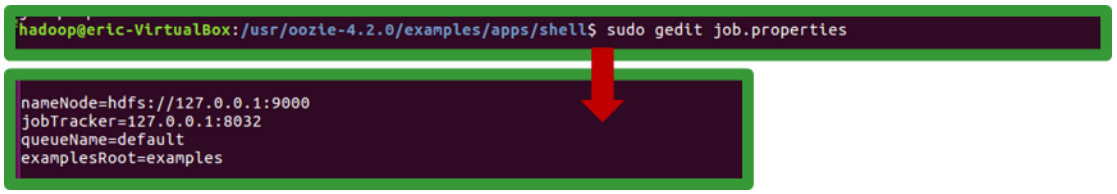
第3节 运行 ooize 示例

Ooize 自带了丰富的示例程序，这些示例可以帮助开发人员理解 Oozie 中工

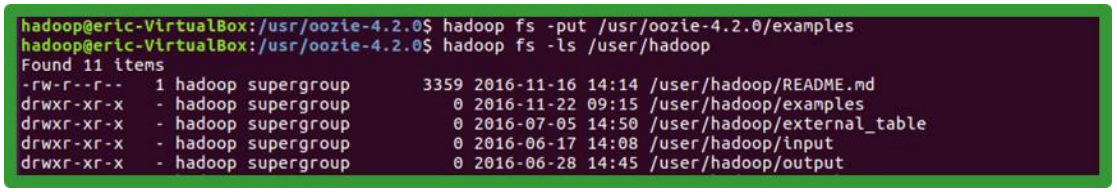
作流的定义、执行和控制的过程。希望运行 Oozie 自带的示例，首先需要解压示例压缩包：



修改需要运行的示例的 job.properties 文件：

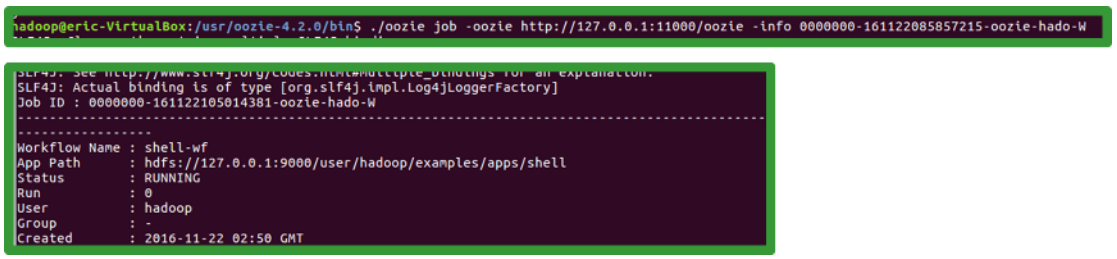


将示例上传到 HDFS：

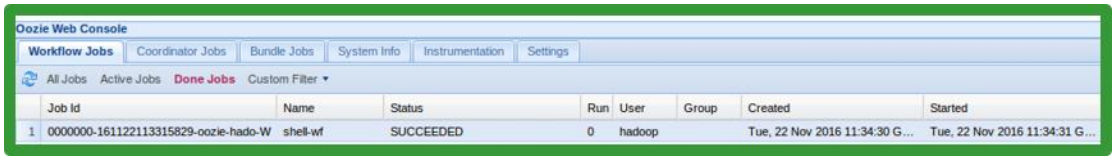


执行示例：

可以通过 job id 查看状态：



可以在 WEB UI 中查看运行状态：



第4节 oozie 工作流程定义

Oozie 工作流程定义是一个 DAG（Directed Acyclical Graphs）图，它由控

制流节点（Control Flow Nodes）或动作节点（Action Nodes）组成，各个节点又是通过表征转移的箭线（transitions arrows）互相连通。对于工作流一般对应存在流程定义语言，例如 jBPM 是 jPDL，大多数都是基于 XML 定义的，Oozie 流程定义语言也是基于 XML 定义的，称为 hPDL（Hadoop Process Definition Language）。

在 Oozie 中，工作流的状态可能存在如下几种：

状态	含义说明
PREP	一个工作流 Job 第一次创建将处于 PREP 状态，表示工作流 Job 已经定义，但是没有运行。
RUNNING	当一个已经被创建的工作流 Job 开始执行的时候，就处于 RUNNING 状态。它不会达到结束状态，只能因为出错而结束，或者被挂起。
SUSPENDED	一个 RUNNING 状态的工作流 Job 会变成 SUSPENDED 状态，而且它会一直处于该状态，除非这个工作流 Job 被重新开始执行或者被杀死。
SUCCEEDED	当一个 RUNNING 状态的工作流 Job 到达了 end 节点，它就变成了 SUCCEEDED 最终完成状态。
KILLED	当一个工作流 Job 处于被创建后的状态，或者处于 RUNNING、SUSPENDED 状态时，被杀死，则工作流 Job 的状态变为 KILLED 状态。
FAILED	当一个工作流 Job 不可预期的错误失败而终止，就会变成 FAILED 状态。

上述各种状态存在相应的转移（工作流程因为某些事件，可能从一个状态跳转到另一个状态），其中合法的状态转移有如下几种，如下表所示：

转移前状态	转移后状态集合
未启动	PREP
PREP	RUNNING、KILLED
RUNNING	SUSPENDED、SUCCEEDED、KILLED、FAILED
SUSPENDED	RUNNING、KILLED

工作流程定义中，控制工作流的开始和结束，以及工作流 Job 的执行路径的节点，它定义了流程的开始（start 节点）和结束（end 节点或 kill 节点），同时提供了一种控制流程执行路径的机制（decision 决策节点、fork 分支节点、join 会签节点）。

start 节点：

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:oozie:workflow:0.1">
    ...
    <start to="[NODE-NAME]" />
    ...
</workflow-app>
```

start 元素的 to 属性，指向第一个将要执行的工作流节点。

End 节点：

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:oozie:workflow:0.1">
    ...
    <end name="[NODE-NAME]" />
    ...
</workflow-app>
```

达到该节点，工作流 Job 会变成 SUCCEEDED 状态，表示成功完成。一个工作流定义必须只能有一个 end 节点。

Kill 节点：

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:oozie:workflow:0.1">
    ...
    <kill name="[NODE-NAME]">
        <message>[MESSAGE-TO-LOG]</message>
    </kill>
    ...
</workflow-app>
```

kill 元素的 name 属性，是要杀死的工作流节点的名称，message 元素指定了工作流节点被杀死的备注信息。达到该节点，工作流 Job 会变成状态 KILLED。

decision 节点：

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:oozie:workflow:0.1">
    ...
    <decision name="[NODE-NAME]">
        <switch>
            <case to="[NODE_NAME]">[PREDICATE]</case>
           ...
            <case to="[NODE_NAME]">[PREDICATE]</case>
            <default to="[NODE_NAME]" />
        </switch>
    </decision>
    ...
</workflow-app>
```

decision 节点通过预定义一组条件，当工作流 Job 执行到该节点时，会根据其中的条件进行判断选择，满足条件的路径将被执行。decision 节点通过 switch…case 语法来进行路径选择，只要有满足条件的判断，就会执行对应的路径，如果没有可以配置 default 元素指向的节点。

fork 节点和 join 节点：

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:oozie:workflow:0.1">
```

```
    ...
    <fork name="[FORK-NODE-NAME]">
        <path start="[NODE-NAME]" />
        ...
        <path start="[NODE-NAME]" />
    </fork>
    ...
    <join name="[JOIN-NODE-NAME]" to="[NODE-NAME]" />
    ...
</workflow-app>
```

fork 元素下面会有多个 path 元素，指定了可以并发执行的多个执行路径。fork 中多个并发执行路径会在 join 节点的位置会合，只有所有的路径都到达后，才会继续执行 join 节点。

动作节点（Action Nodes）是工作流程定义中，能够触发一个计算任务或者处理任务执行的节点。所有的动作都有一些基本的特性：

- **远程执行**

对 Oozie 来说，动作节点的执行都是远程的，因为 Oozie 可能部署在一个单独的服务器上，而工作流 Job 是在 Hadoop 集群的节点上执行的。即使 Oozie 在 Hadoop 集群的某个节点上，它也是处于与 Hadoop 进行独立无关的 JVM 示例之中。

- **异步性**

动作节点的执行，对于 Oozie 来说是异步的。Oozie 启动一个工作流 Job，这个工作流 Job 便开始执行。Oozie 可以通过两种方式来探测工作流 Job 的执行情况：一种是基于回调机制，对每个任务的执行（可以看成是动作节点的执行）都对应一个唯一的 URL，如果任务执行结束或者执行失败，会通过回调这个 URL 通知 Oozie 已经完成；另一种就是轮询，Oozie 不停地去查询任务执行的完成状态，如果由于网络故障回调机制失败，也会使用轮询的方式来处理。

- **执行结果要么成功，要么失败**
- **如果动作节点执行成功，则会转向 ok 节点；如果失败则会转向 error 节点**
- **可恢复性**

如果一个动作节点执行失败，Oozie 提供了一些恢复执行的策略，这个要根据失败的特点来进行：如果是状态转移过程中失败，Oozie 会根据指定的重试时间间隔去重新执行；如果不是转移性质的失败，则只能通过手工干预来进行恢复；如果重试恢复执行都没有解决问题，则最终会跳转到 error 节点。

动作节点包括 MR 动作、Hive、Shell 动作等，详情可以参考 oozie 的开发文档，其中包含节点声明的详细格式。

Oozie 除了可以使用 Properties 文件定义一些属性之外，还提供了一些内置的 EL 函数，能够方便地实现流程的定义和控制。基本 EL 常量如下：

常量名称	含义说明
KB	1KB，类型为 long。
MB	1MB，类型为 long。
GB	1GB，类型为 long。
TB	1TB，类型为 long。
PB	1PB，类型为 long。

基本 EL 函数如下：

函数声明	含义说明
String firstNotNull(String value1, String value2)	返回 value1 和 value2 中不为 null 的值，若都为 null 则返回 null
String concat(String s1, String s2)	连接字符串 s1 和 s2，如果 s1 或 s2 为 null 值，则使用空字符串替换 null 值
String replaceAll(String src, String regex, String replacement)	满足正则表达式 regex，则使用 replace 替换 src 字符串中匹配上的部分
String appendAll(String src, String append, String delimeter)	将 src 中的分隔符 delimeter 替换为 append
String trim(String s)	去掉字符串两边的空格，如果 s 为 null 则返回空字符串

工作流 EL 函数如下：

函数声明	含义说明
String wf:id()	返回当前的工作流 Job 的 ID
String wf:name()	返回当前的工作流 Job 的名称
String wf:appPath()	返回当前的工作流 Job 的应用路径
String wf:conf(String name)	返回当前的工作流 Job 的配置属性
String wf:user()	返回启动当前的工作流 Job 的用户名称

Hadoop EL 常量列表如下：

常量名称	含义说明
RECORDS	Hadoop Record 计数器组名称
MAP_IN	Hadoop Mapper 输入 Record 计数器名称
MAP_OUT	Hadoop Mapper 输出 Record 计数器名称
REDUCE_IN	Hadoop Reducer 输入 Record 计数器名称
REDUCE_OUT	HadoopReducer 输出 Record 计数器名称

HDFS EL 函数列表如下：

选项	含义说明
boolean fs:exists(String path)	path 是否存在

boolean fs:isDir(String path)	path 是否是目录
long fs:dirSize(String path)	如果 path 不是目录或者 path 是一个文件，则返回-1，否则返回该 path 下所有文件的字节数
long fs:fileSize(String path)	如果 path 是目录，则返回-1，否则返回该 path 下所有文件的字节数
long fs:blockSize(String path)	如果 path 不是文件或者不存在则返回-1，否则返回文件的块大小字节数

以下代码展示了如何在 Java 中提交 Oozie 任务：

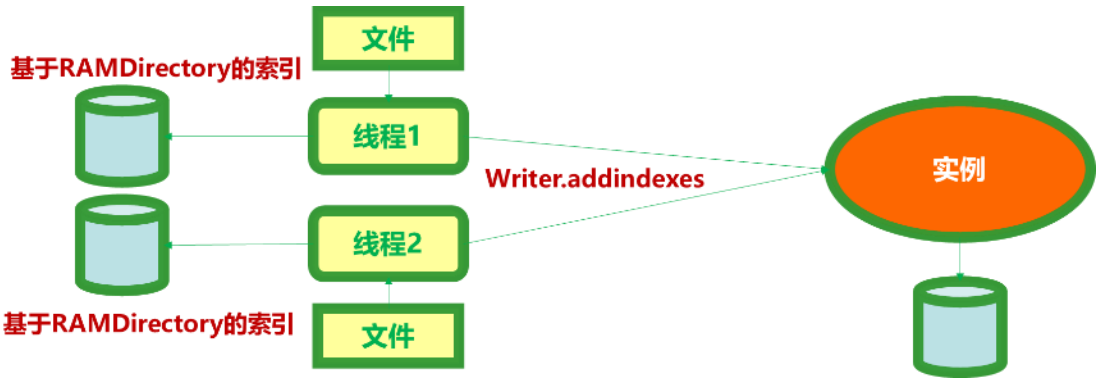
```
public class JavaJob {
    public static void main(String[] args) {
        OozieClient wc = new
OozieClient("http://127.0.0.1:11000/oozie");
        Properties conf = wc.createConfiguration();
        conf.setProperty("nameNode", "hdfs://127.0.0.1:9000");
        conf.setProperty("queueName", "default");
        conf.setProperty("examplesRoot", "examples");
        conf.setProperty("oozie.wf.application.path",

    "${nameNode}/user/hadoop/${examplesRoot}/apps/shell");
        conf.setProperty("jobTracker", "127.0.0.1:8032");

        try {
            String jobId = wc.run(conf);
            System.out.println(jobId);
        } catch (OozieClientException e) {
            e.printStackTrace();
        }
    }
}
```

第15章　全文检索工具

第1节 Solr

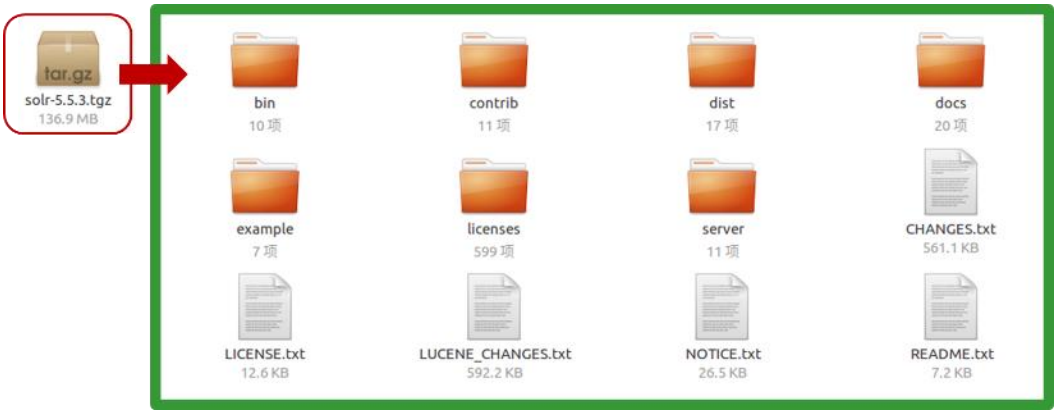
Solr 是一个独立的企业级搜索应用服务器，它对外提供类似于 Web-service 的 API 接口，用户可以通过 http 请求，向搜索引擎服务器提交一定格式的 XML 文件，生成索引；也可以通过 Http Get 操作提出查找请求，并得到 XML 格式的返回结果。

Solr 是一个高性能，采用 Java 开发，基于 Lucene 的全文搜索服务器。同时对其进行了扩展，提供了比 Lucene 更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的全文搜索引擎。



Solr 中的文档通过 Http 利用 XML 加到一个搜索集合中。查询该集合也是通过 http 收到一个 XML/JSON 响应来实现。它的主要特性包括：高效、灵活的缓存功能，垂直搜索功能，高亮显示搜索结果，通过索引复制来提高可用性，提供一套强大 Data Schema 来定义字段，类型和设置文本分析，提供基于 Web 的管理界面等。

最新版的Solr要求JDK为1.8以上且和常用的中文分析组件有兼容性问题，可以选择 Solr5 来在 1.7 版本的 JDK 上运行，下载 Solr 发行包并解压：



解压后就可以直接启动 Solr 了。首先可以运行 Solr 内置的一个实例集合，

通过-e 参数能够启动它：

```
hadoop@eric-VirtualBox:/usr/solr-5.5.3/bin$ ./solr start -e cloud
```

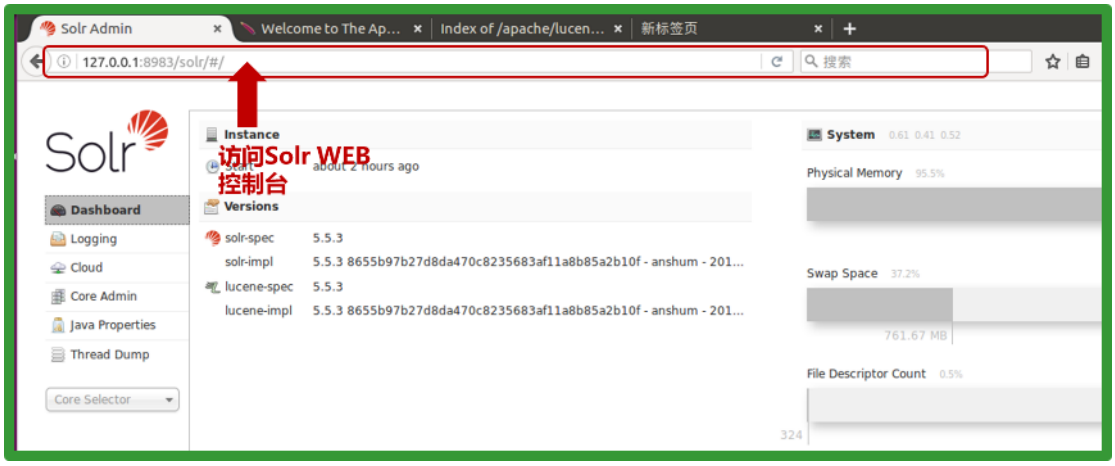
接下来 Solr 会要求设定示例运行的端口，默认为 8983：

```
Please enter the port for node1 [8983]:

Creating Solr home directory /usr/solr-5.5.3/example/cloud/node1/solr
```

还需要设置运行实例程序所使用的集合名，在上传文档时和在 WEB 界面中进行全文检索查询时需要用到这个名字：

```
Now let's create a new collection for indexing documents in your 1-node cluster.
Please provide a name for your new collection: [gettingstarted]
c1
```

启动内置示例后即可通过浏览器访问 Solr：



通过 Solr 的 post 命令上传文件，注意建议只上传运行 post 指令的路径中的文件，上传时需要提供集合名：

```
hadoop@eric-VirtualBox:/usr/solr-5.5.3/bin$ ./post -c c1 mem.xml
/usr/jdk7/bin/java -classpath /usr/solr-5.5.3/dist/solr-core-5.5.3.jar -Dauto=yes -Dc=c1 -Ddata=files org.apache.
solr.util.SimplePostTool mem.xml
SimplePostTool version 5.0.0
Posting files to [base] url http://localhost:8983/solr/c1/update...
Entering auto mode. File endings considered are xml,json,jsonl,csv,pdf,doc,docx,ppt,pptx,xls,xlsx,odt,odp,ods,ott
,otp,ots,rtf,htm,html,txt,log
POSTing file mem.xml (application/xml) to [base]
1 files indexed.
COMMITting Solr index changes to http://localhost:8983/solr/c1/update...
Time spent: 0:00:04.931
```

接下来就可以在 Solr 提供的 WEB 交互见面中进行文档关键字检索。首先在主界面中找到集合选择组件，通过下拉列表选择在启动实例程序时构建的集合名，这时候就会出现该集合对应的操作菜单，在这些菜单项中点击 Query 链接，界面右边将会出现查询对话框，输入查询的关键字，选择生成的结果类型等参数后点击执行查询，将会直接在界面中展示检索结果：

solr 支持丰富的查询语法，比如：现在想搜索字段 name 里面的关键字“Memory”就可以用语法 name:Memory，当然如果搜索 name:xxx 就没有返回结果了，因为文档中没有这样的内容。

导入数据到 Solr 的方式也是多种多样的：

- 可以使用 DIH(DataImportHandler)从数据库导入数据
- 支持 CSV 文件导入，因此 Excel 数据也能轻松导入
- 支持 JSON 格式文档
- 二进制文档比如：Word、PDF
- 还能以编程的方式来自定义导入

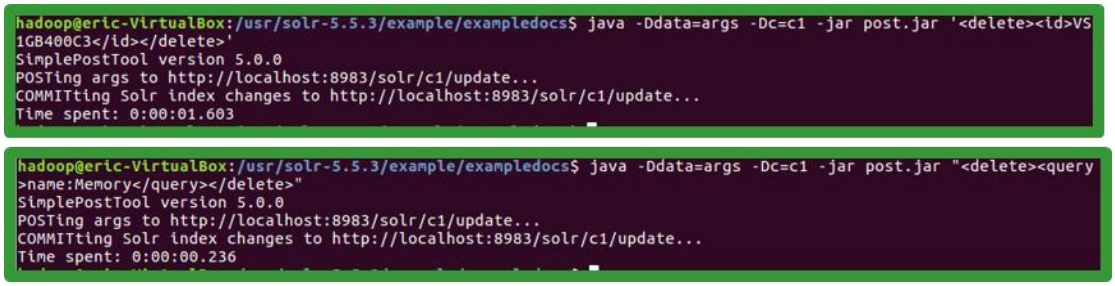
solr 会根据文档的字段 id 来唯一标识文档，如果导入的文档的 id 已经存在 solr 中，那么这份文档就被最新导入的同 id 的文档自动替换。可以自己尝试试验一下，观察替换前后管理界面的几个参数：Num Docs，Max Doc，Deleted Docs 的变化：

numDocs：当前系统中的文档数量，它有可能大于 xml 文件个数，因为一个 xml 文件可能有多个<doc>标签。

maxDoc：maxDoc 有可能比 numDocs 的值要大，比如重复 post 同一份文件后，maxDoc 值就增大了。

deletedDocs：重复 post 的文件会替换掉老的文档，同时 deltedDocs 的值也会加 1，不过这只是逻辑上的删除，并没有真正从索引中移除掉。

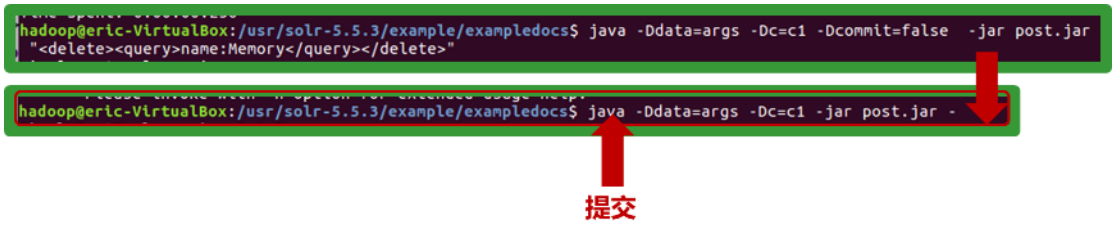
可以通过 solr 的 post.jar 执行文件删除操作，可以通过 id 或一个查询来删除：



文档删除后再次执行之前的搜索操作，发现结果已经发生了变化。还可以使用 solr 提供的 HTTP API 执行相同的数据删除操作：

```
http://127.0.0.1:8983/solr/c1/update?stream.body=<delete><id>TWINX2048-3200PRO</id></delete>&commit=true
```

也可以把 commit 设置为 false，手动提交事务：



Solr 同样提供了灵活的方法来对应用户删除数据的需求，接下来展示不同的数据删除方式。删除所有数据的命令如下：

```
http://127.0.0.1:8983/solr/c1/update?stream.body=<delete><query>*:*</query></delete>&commit=true
```

可以提供条件来指定删除的数据：

```
http://127.0.0.1:8983/solr/c1/update?stream.body=<delete><query>title:abc</query></delete>&commit=true
```

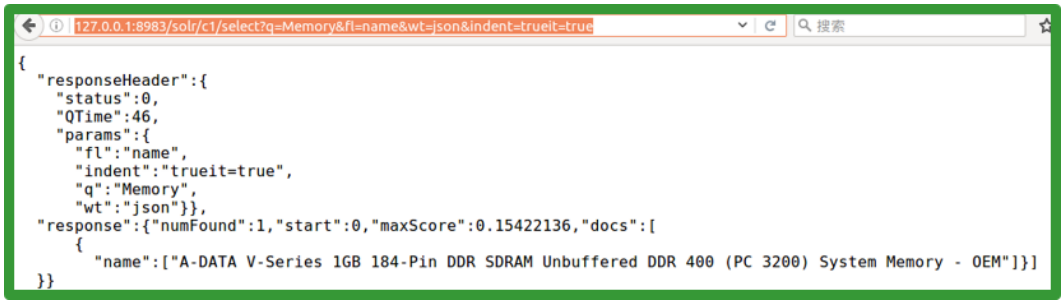
删除时还支持多条件查询：

```
http://127.0.0.1:8983/solr/c1/update?stream.body=<delete><query>title:abc AND name:zhang</query></delete>&commit=true
```

查询数据都是通过 HTTP 的 GET 请求获取的，搜索关键字用参数 q 指定，另外还可以指定很多可选的参数来控制信息的返回，例如：用 fl 指定返回的字段，比如 fl=name，那么返回的数据就只包括 name 字段的内容：

```
http://127.0.0.1:8983/solr/c1/select?q=Memory&fl=name&wt=json&indent=true
```

该查询的执行结果如下：



Solr 提供排序的功能，通过参数 sort 来指定，它支持正序、倒序，或者多个字段排序:

```
q=video&sort=price desc
q=video&sort=price asc
q=video&sort=inStock asc, price desc
```

默认条件下，Solr 根据 socre 倒序排列，socre 是一条搜索记录根据相关度计算出来的一个分数。

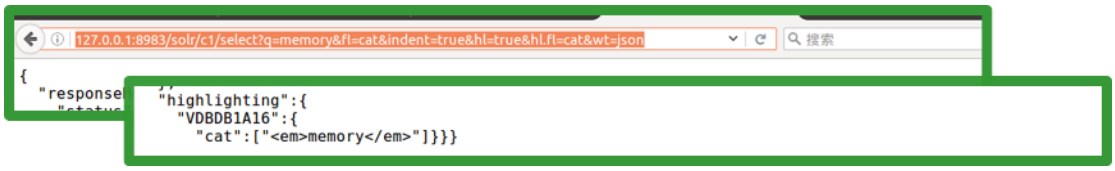
网页搜索中，为了突出搜索结果，可能会对匹配的关键字高亮出来，Solr 提供了很好的支持，只要指定参数：

```
hl=true #开启高亮功能
hl.fl=name #指定需要高亮的字段
```

例如下面的查询：

```
http://127.0.0.1:8983/solr/c1/select?q=memory&fl=cat&indent=true&hl=true&hl.fl=cat&wt=json
```

执行后返回的结果如下：



文本字段通过把文本分割成单词以及运用各种转换方法（如：小写转换、复数移除、词干提取）后被索引，schema.xml 文件中定义了字段在索引中的结构和类型，这些字段将作用于检索过程。例如其中默认情况下搜索“power-shot”是不能匹配“powershot”的，通过修改 schema 文件，把字段替换成“text_en_splitting”类型，就能索引到了

Solr5.5 之后和之前的版本存在较大的不同，不再默认使用 schema.xml 配置字段信息，而是默认使用 managed-schema 文件进行配置，两者的区别是，使用 schema.xml 时，上传的文档格式必须和 schema.xml 中完全匹配，而使用 managed-schema 文件时，如果上传的文件结构中存在没有预先定义的字段，将自动添加。

接下来介绍自定义文字分析方法的过程。首先可以使用正常方式启动 Solr（不使用实例程序）：

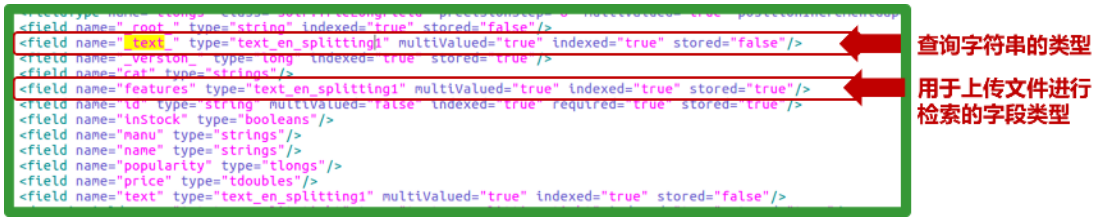
```
hadoop@eric-VirtualBox:/usr/solr-5.5.3/bin$ ./solr start
```

然后创建新的集合。由于本次启动没有使用实例程序，因此集合创建成功后不会包含仍和默认的文档，所有需要进行索引和检索的文档都需要后续手动上传，另一方面，创建新的集合后，定义字符串分析模式的 Schema 文件保持了默认的配置（每一个集合都拥有自己独立的 Schema 文件）。

接下来修改集合中的 schema 配置，Schema 文件位于 SOLR_HOME/server/solr/集合名/conf 路径中：

```
hadoop@eric-VirtualBox:/usr/solr-5.5.3/server/solr/newc/conf$ sudo gedit managed-schema
[sudo] hadoop 的密码:
```

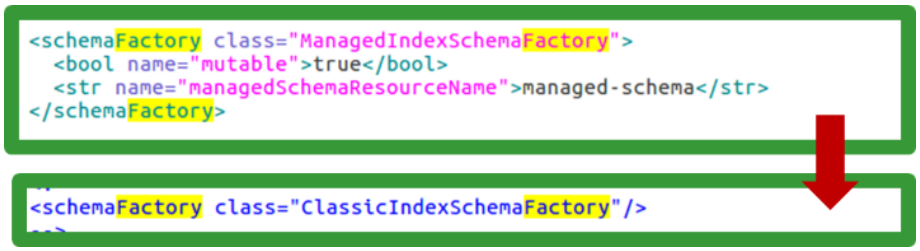
将待查询的字段和默认的文本字段类型修改成 text_en_splitting：



修改后重启 Solr，重新上传文件，然后检索，就能够实现分析匹配：

如果需要使用老版本的 schema.xml 文件，则执行以下的操作：

- 将 managed-schema 文件复制重命名为 schema.xml
- 修改 solrconfig.xml 文件：

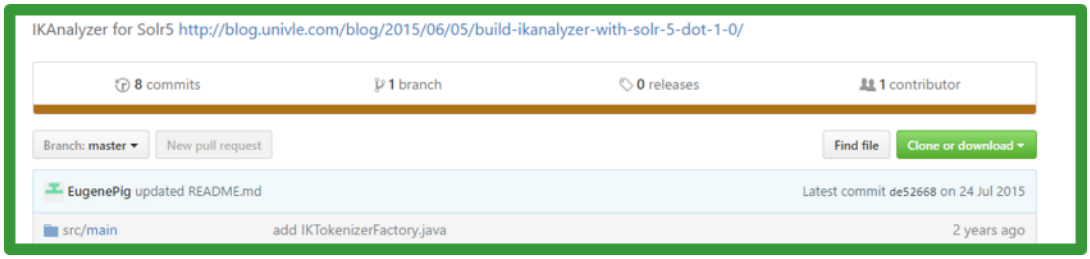


- 重启 solr，上传文件

默认情况下，Solr 无法对中文进行正确的分析，在中文环境下，需要集成专门的分析组件来完成中文的分析和检索。IK Analyzer 是一款结合了词典和文法分析算法的中文分词组件，基于字符串匹配，支持用户词典扩展定义，支持细粒

度和智能切分，比如：针对“张三说的确实在理”这一段字符串，智能分词的结果是：[张三 | 说的 | 确实 | 在理]，而最细粒度分词结果：[张三 | 三 | 说的 | 的确 | 的 | 确实 | 实在 | 在理]。

可以在 https://github.com/EugenePig/ik-analyzer-solr5 下载 IKAnalyzer,需要注意，下载源代码而不是已经打包好的 jar 包，因为其他现成的 jar 对新版本的 Solr 有兼容性问题：



在下载页面上有详细的编译说明：

下载完成后按照网页中的指导步骤执行编译操作：

```
hadoop@eric-VirtualBox:/usr/ik-analyzer-solr5-master$ mvn clean -Djavac.src.version=1.7 -Djavac.target.version=1.7 install
```

编译生成完成后，将生成的 jar 包放入 Solr WebApp 的 WEB-INF/lib 中，注意 jar 包的访问权限，应该保证使用 Solr 的用户对这个 jar 包拥有执行权限：

```
hadoop@eric-VirtualBox:/usr/ik-analyzer-solr5-master/target$ sudo cp ik-analyzer-solr5-5.x.jar /usr/solr-5.5.3/server/solr-webapp/webapp/WEB-INF/lib/
```

修改 Solr schema 配置，加入字段类型声明：

```
hadoop@eric-VirtualBox:/usr/solr-5.5.3/server/solr/newc/conf$ sudo gedit managed-schema
```

```
<fieldType name="text_ik" class="solr.TextField">
      <analyzer type="index" isMaxWordLength="false" class="org.wltea.analyzer.lucene.IKAnalyzer"/>
      <analyzer type="query" isMaxWordLength="true" class="org.wltea.analyzer.lucene.IKAnalyzer"/>
</fieldType>
```

在上面的自定义类型中，提供了两个分析器，仔细观察可以发现，两个分析器的实现类相同，所不同的是一个分析器的 type 类型为 index，最细粒度分词策略属性为 false，而另一个分析器的 type 类型为 query，最细粒度分词策略属性为 true。类型为 index 的分析器将在上传文档并对该文档构建索引时生效，另一个分析器则将在用户执行关键字全文检索操作时生效。

可以通过 Solr 的分词操作界面查验字符分析器是否生效：

除了可以使用通用的 HTTP 协议请求来操作 Solr 外，Solr 还提供了 solrj 来使用 java 进行文档和检索，SolrJ 本质上是封装了 httpClient 包，来操作 solr 的 API。接下来介绍常见的数据操作实现方法。

删除文档数据的参考实现如下：

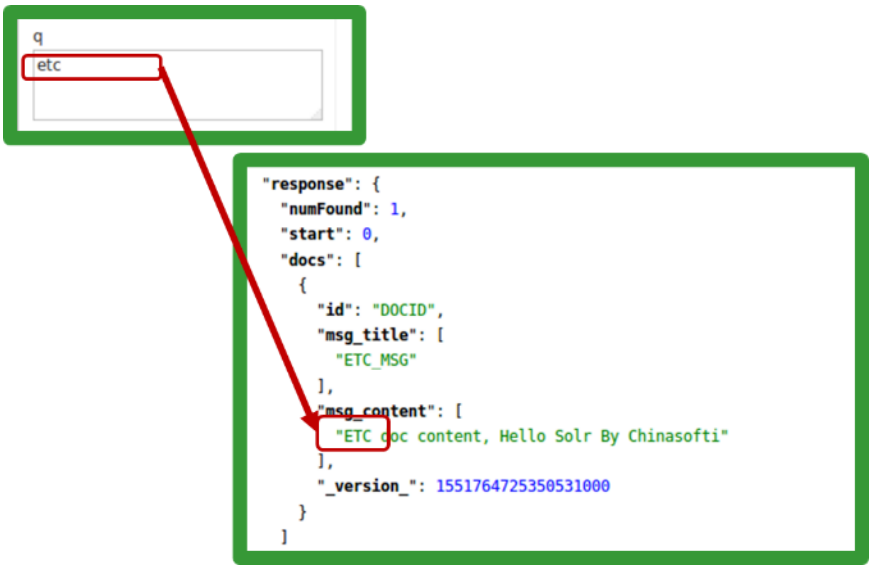
```
public class TestSolr {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //baseURL需要包含集合名
        String baseURL = "http://127.0.0.1:8983/solr/newc";
        SolrClient server = new HttpSolrClient(baseURL);
        try {
            //可以通过ID或查询删除，注意提交
            server.deleteByQuery("*:*");
            //server.deleteById("DOCID");
            server.commit();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

通过 Java 代码向已经存在的集合中添加文档的方法如下：

```
public class AddSolrDoc {
    public static void main(String[] args) {
        String baseURL = "http://127.0.0.1:8983/solr/newc";
        SolrClient server = new HttpSolrClient(baseURL);
        try {
            SolrInputDocument doc = new SolrInputDocument();
```

```
            doc.addField("id", "DOCID");
            doc.addField("msg_title", "ETC_MSG");
            doc.addField("msg_content",
                    "ETC doc content, Hello Solr By Chinasofti");
            server.add(doc);
            server.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

该程序执行成功后会将对应的字符串数据添加到集合文档中，可以在 Solr 的检索界面中进行测试搜索，搜索结果如下：



在刚才的添加文档代码中，使用了类似 Map 的方法添加属性，对于很多开发人员而言，其实更习惯传统的面向对象方式对数据进行操作，即将文档的各个字段保存到一个对象中，然后直接对该对象进行操作，接下来的代码就演示了如何利用一个封装好的 JavaBean 完成文档数据的添加，首先构建封装文档数据的 JavaBean：

```
public class Message {
    @Field
    String id;
    @Field
    String msgTitle;
    @Field
    String[] msgContent;

    public String getId() {
        return id;
```

```
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getMsgTitle() {
        return msgTitle;
    }

    public void setMsgTitle(String msgTitle) {
        this.msgTitle = msgTitle;
    }

    public String[] getMsgContent() {
        return msgContent;
    }

    public void setMsgContent(String[] msgContent) {
        this.msgContent = msgContent;
    }

    public Message(String id, String msgTitle, String[] msgContent)
{
        super();
        this.id = id;
        this.msgTitle = msgTitle;
        this.msgContent = msgContent;
    }

    public Message() {
        super();
    }

}
```

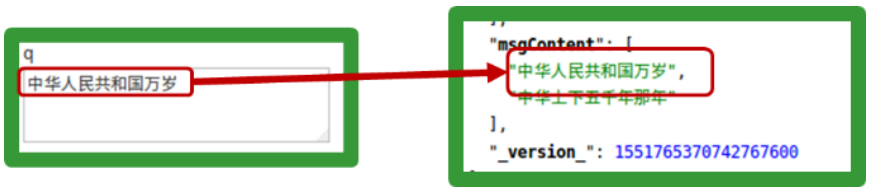
在该 JavaBean 中，需要利用 Field 注解创建字段。

然后编写添加文档的操作代码：

```
public class AddSolrDocWithBean {
    public static void main(String[] args) {
        String baseURL = "http://127.0.0.1:8983/solr/newc";
        SolrClient server = new HttpSolrClient(baseURL);
        try {
            List<Message> msgs = new ArrayList<Message>();
```

```
            msgs.add(new Message("DOC1", "TITLE1", new String[] { "
中华人民共和国万岁",
                    "中华上下五千那年" }));
            msgs.add(new Message("DOC2", "TITLE2", new String[] { "
中软国际是个好公司",
                    "ETC共成长" }));
            server.addBeans(msgs);
            server.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

执行改代码后，文档数据同样被添加到集合中，可以在 Solr 的检索界面中进行测试搜索，搜索结果如下：

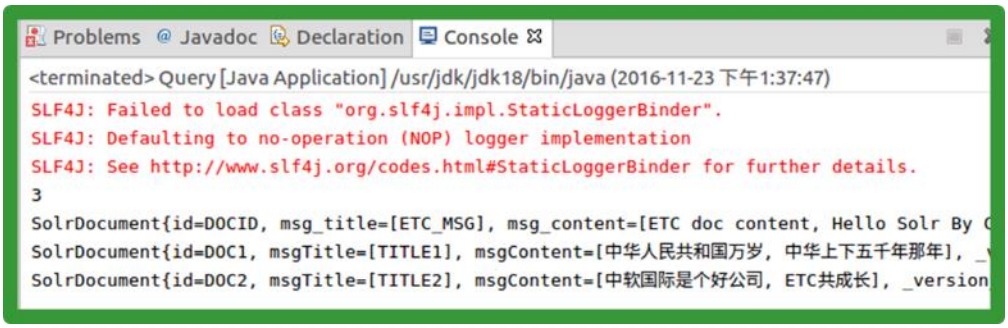


对于全文检索工具而言，最为重要，也是最为常用的 API 调用应该就是在集合中对给出的特定关键字进行检索并获取结果。利用 Java 对 Solr 进行检索操作非常方便，参考实现代码如下：

```
public class Query {
    public static void main(String[] args) {
        String baseURL = "http://127.0.0.1:8983/solr/newc1";
        SolrClient server = new HttpSolrClient(baseURL);
        try {
            // 定义查询内容 * 代表查询所有 这个是基于结果集
            SolrQuery query = new SolrQuery("corsair");
            query.setStart(0);// 起始页
            query.setRows(5);// 每页显示数量
            query.setParam("hl.fl", "manu_id_s");// 设置哪些字段域会
高亮显示
            query.setHighlight(true)
                    .setHighlightSimplePre("<span class='hight'>")
                    .setHighlightSimplePost("</span>");
            QueryResponse rsp = server.query(query);
            SolrDocumentList results = rsp.getResults();
            System.out.println(results.getNumFound());// 查询总条数
            for (SolrDocument doc : results) {
```

```
                System.out.println(doc);

            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

该查询程序运行结果如下：



同样的，查询结果也可以直接填充一个给定的 JavaBean，以便于获取查询结果后对数据的后续操作，首先构建返回结果的 JavaBean，改 Bean 同样需要利用 Field 注解创建字段。：

```
public class ResponseMessage {
    @Field
    String id;
    @Field
    String[] msgTitle;
    @Field
    String[] msgContent;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String[] getMsgTitle() {
        return msgTitle;
    }

    public void setMsgTitle(String msgTitle[]) {
```

```
        this.msgTitle = msgTitle;
    }

    public String[] getMsgContent() {
        return msgContent;
    }

    public void setMsgContent(String[] msgContent) {
        this.msgContent = msgContent;
    }

    public ResponseMessage(String id, String[] msgTitle, String[]
msgContent) {
        super();
        this.id = id;
        this.msgTitle = msgTitle;
        this.msgContent = msgContent;
    }

    public ResponseMessage() {
        super();
    }

}
```

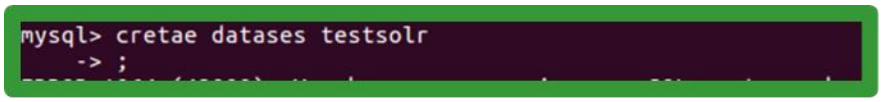
然后利用该 Bean 获取查询结果：

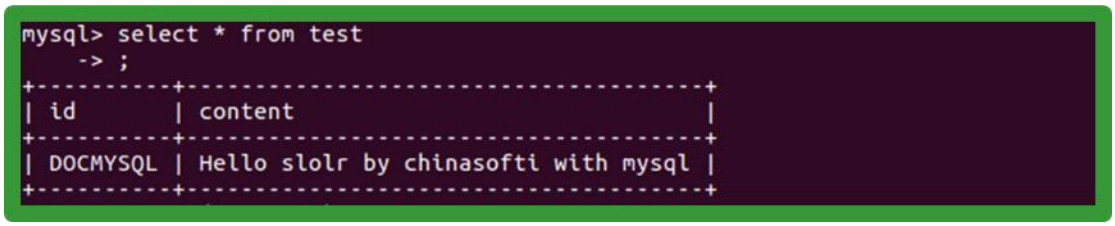
```
public class QueryWithBean {
    public static void main(String[] args) {
        String baseURL = "http://127.0.0.1:8983/solr/newc";
        SolrClient server = new HttpSolrClient(baseURL);
        try {
            SolrQuery query = new SolrQuery("中软国际是个好公司");//
* 号 是查询 所有的数据
            QueryResponse rsp = server.query(query);
            List<ResponseMessage> beans =
rsp.getBeans(ResponseMessage.class);// 这个不能获取查询的总数了,也
不能高亮
            for (ResponseMessage message : beans) {
                for(String msg:message.getMsgTitle()){
                    System.out.println(msg);
                }
            }
            for (ResponseMessage message : beans) {
                for(String msg:message.getMsgContent()){
```

```
                    System.out.println(msg);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序的执行结果如下：

Solr 有一个比较强大的功能，帮助开发人员快速的将老旧的基于 RDBMS 的文档库系统迁移到大数据平台上来，它能够通过一个数据导入处理器将 RDBMS 中的文档数据快速的导入到自己的索引系统中来。为了演示这个功能，首先在 MySQL 中创建数据库表结构，并插入测试数据：





在 schema 文件中增加对应的字段（id 已经存在，增加 content 即可）：

```
<field name="content" type="strings"/>
```

修改 solrconfig.xml,增加导入数据的 requestHandler：

```
<requestHandler name="/dataimport" class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
       <str name="config">data-config.xml</str>
  </lst>
</requestHandler>
```

在同一个目录中添加 data-config.xml,内容如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<dataConfig>
 <dataSource type="JdbcDataSource" driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://127.0.0.1:3306/testsolr" user="root"
password="root"/>
    <document>
        <entity name="check_lawyer" query="select * from test">
            <field column="id" name="id" />
            <field column="content" name="content" />
        </entity>
    </document>
</dataConfig>
```

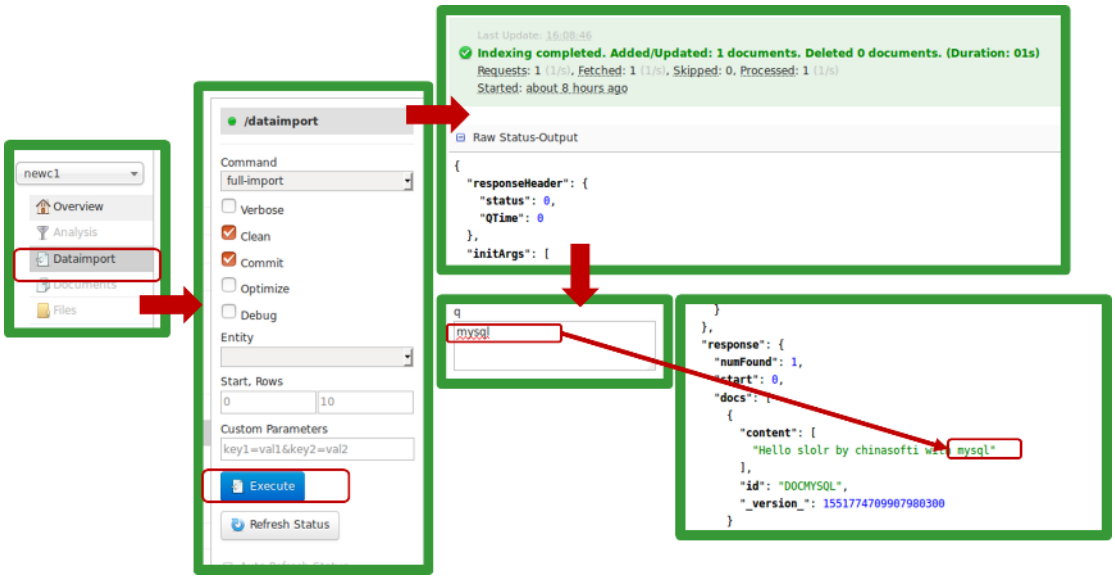
拷贝输入导入的支持库到 WEB-INF/lib，支持库位于 solr 的 dist 目录下，将以下两个文件拷贝到 WEB-INF/lib 中：

```
solr-dataimporthandler-5.5.3.jar  solr-dataimporthandler-extras-5.5.3.jar
```

拷贝 MySQL 的 JDBC 驱动到 solr 的 server/lib 下：

```
javax.servlet-api-3.1.0.jar              jetty-rewrite-9.2.13.v20150730.jar   jetty-webapp-9.2.13.v20150730.jar
jetty-continuation-9.2.13.v20150730.jar  jetty-security-9.2.13.v20150730.jar  jetty-xml-9.2.13.v20150730.jar
jetty-deploy-9.2.13.v20150730.jar        jetty-server-9.2.13.v20150730.jar    mysql-connector-java-5.1.39-bin.jar
jetty-http-9.2.13.v20150730.jar          jetty-servlet-9.2.13.v20150730.jar
jetty-io-9.2.13.v20150730.jar            jetty-servlets-9.2.13.v20150730.jar
hadoop@eric-VirtualBox:/usr/solr-5.5.3/server/lib$
```

完成这些配置后，重新启动 Solr 服务，然后就可以在 Solr 的界面中找到导入数据的处理器，执行导入操作后，MySQL 中原来保存的数据就被抽取到了 Solr 的检索系统，可以通过 Solr 进行关键字检索了：



第2节 Elasticsearch

Elasticsearch 是分布式,REST 风格,搜索和分析系统。具有实时数据，实时分析，分布式，高可用性，多租户，全文搜索，面向文档，冲突管理，自由模式，rest 风格 API，操作持久性，Apache 2 的开源许可证，基于 Apache Lucene 之上的特点：

- 分布式实时文件存储，可将每一个字段存入索引，使其可以被检索到。
- 实时分析的分布式搜索引擎。
- 分布式：索引分拆成多个分片，每个分片可有零个或多个副本。集群中的每个数据节点都可承载一个或多个分片，并且协调和处理各种操作；
- 负载再平衡和路由在大多数情况下自动完成。
- 可以扩展到上百台服务器，处理 PB 级别的结构化或非结构化数据。也可以运行在单台 PC 上（已测试）
- 支持插件机制，分词插件、同步插件、Hadoop 插件、可视化插件等。

Elasticsearch 在很多应用场景发挥着重要的作用：

维基百科使用 Elasticsearch 来进行全文搜索并高亮显示关键词，以及提供 search-as-you-type、did-you-mean 等搜索建议功能。

英国卫报使用 Elasticsearch 来处理访客日志，以便能将公众对不同文章的反应实时地反馈给各位编辑。

StackOverflow 将全文搜索与地理位置和相关信息进行结合，以提供 more-like-this 相关问题的展现。

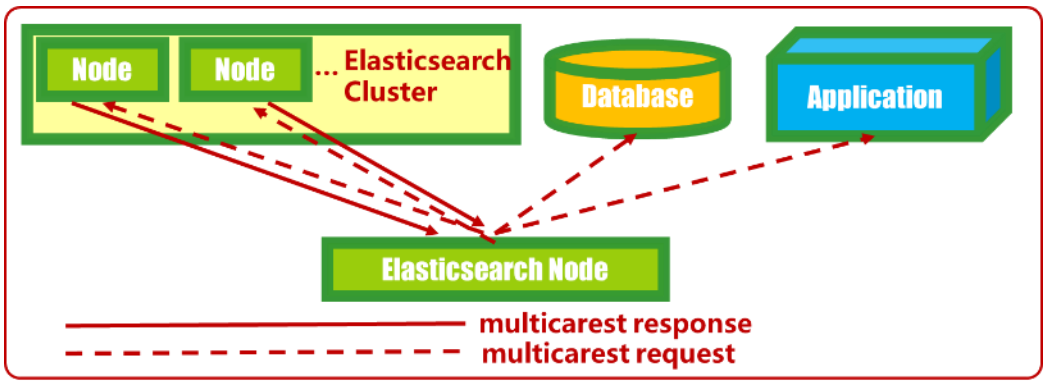
GitHub 使用 Elasticsearch 来检索超过 1300 亿行代码。

每天，Goldman Sachs 使用它来处理 5TB 数据的索引，还有很多投行使用它

来分析股票市场的变动。

Elasticsearch 并不只是面向大型企业的，它还帮助了很多类似 DataDog 以及 Klout 的创业公司进行了功能的扩展。Elasticsearch 可以运行在普通的笔记本上，也可以部署到成千上万的服务器上，处理 PB 级别的数据。

当 ElasticSearch 的节点启动后，它会利用多播(multicast)(或者单播，如果用户更改了配置)寻找集群中的其它节点，并与之建立连接：



ES 可以作为一个独立的单个搜索服务器。不过，为了处理大型数据集，实现容错和高可用性，ES 可以运行在许多互相合作的服务器上。这些服务器的集合称为集群。形成集群的每个服务器称为节点。

当有大量的文档时，由于内存的限制、磁盘处理能力不足、无法足够快的响应客户端的请求等，一个节点可能不够。这种情况下，数据可以分为较小的分片。每个分片放到不同的服务器上。当查询的索引分布在多个分片上时，ES 会把查询发送给每个相关的分片，并将结果组合在一起，而应用程序并不知道分片的存在。即：这个过程对用户来说是透明的。

为提高查询吞吐量或实现高可用性，可以使用分片副本。副本是一个分片的精确复制，每个分片可以有零个或多个副本。ES 中可以有许多相同的分片，其中之一被选择更改索引操作，这种特殊的分片称为主分片。当主分片丢失时，如：该分片所在的数据不可用时，集群将副本提升为新的主分片。

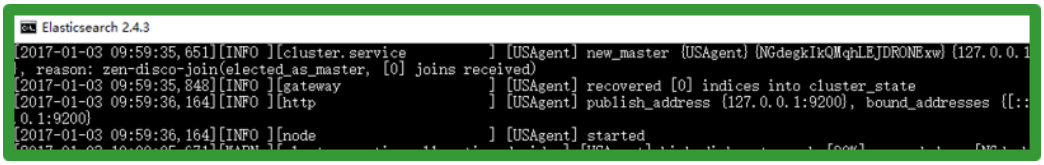
ES 数据架构的主要概念（与关系数据库 Mysql 对比）如下

MySQL	ElasticSearch
Database	Index
Table	Type
Row	Document
Column	Field
Schema	Mapping
Index	Everything is indexes
SQL	Query DSL
Select * from table …	GET http:// …
Update table set …	PUT http:// …

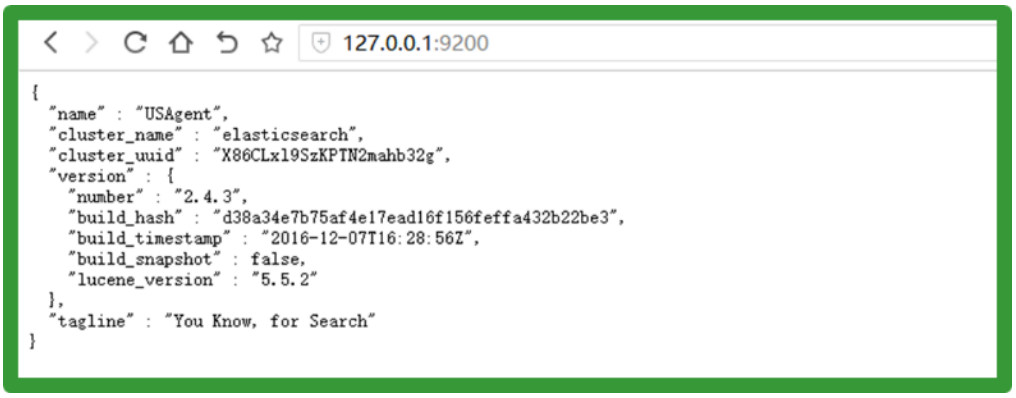
Elasticsearch 的安装非常简单，在官网 https://download.elastic.co/上下载发行包解压即可，零配置，开箱即用。唯一需要注意的是最新版本要求提供 JDK1.8 的支持。

要启动 Elasticsearch，首先要进入到其解压的目标文件夹，然后运行启动服务的脚本，例如：

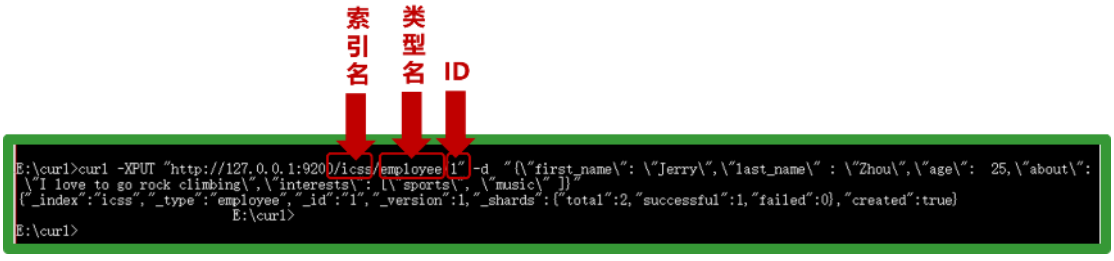
```
cd /usr/local/elasticsearch-2.3.5
./bin/elasticsearch
bin/elasticsearch -d(后台运行)
```



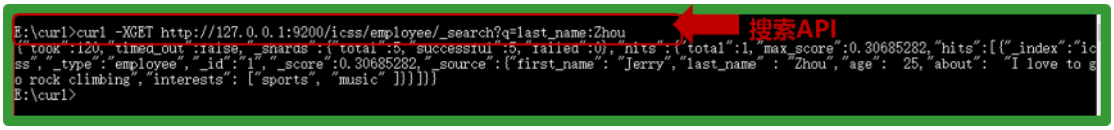
程序启动后，可以通过浏览器验证功能是否正确，直接访问服务器地址会返回对应的环境信息，服务器的默认端口是 9200：



可以通过直接上传文档的方式构建索引：



直接利用 HTTP REST API 进行搜索：



Elasticsearch 与 Solr 的比较：

二者安装都很简单；

Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能；

Solr 支持更多格式的数据，而 Elasticsearch 仅支持 json 文件格式；

Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供；

Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch。

Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。

第16章　图数据库

第1节 Titan

相比起“Hadoop、Spark”这种流行的大数据处理平台，说起“图计算”，可能许多人还比较陌生。甚至有人会误把它当成专门进行“图像”处理的技术。 首先按照互联网上通常的定义来说明一下图计算。

“图计算”是以“图论”为基础的对现实世界的一种“图”结构的抽象表达，以及在这种数据结构上的计算模式。通常，在图计算中，基本的数据结构表达就是：

```
G = （V，E，D） V = vertex （顶点或者节点）       E = edge    （边）
D = data （权重）
```

比如说：对于一个消费者的原始购买行为，有两类节点：用户和产品，边就是购买行为，权重是边上的一个数据结构，可以是购买次数和最后购买时间。对于许多的物理世界的数据问题，都可以利用图结构的来抽象表达：比如社交网络，网页链接关系，用户传播网络，用户网络点击、浏览和购买行为，甚至消费者评论内容，内容分类标签，产品分类标签等等

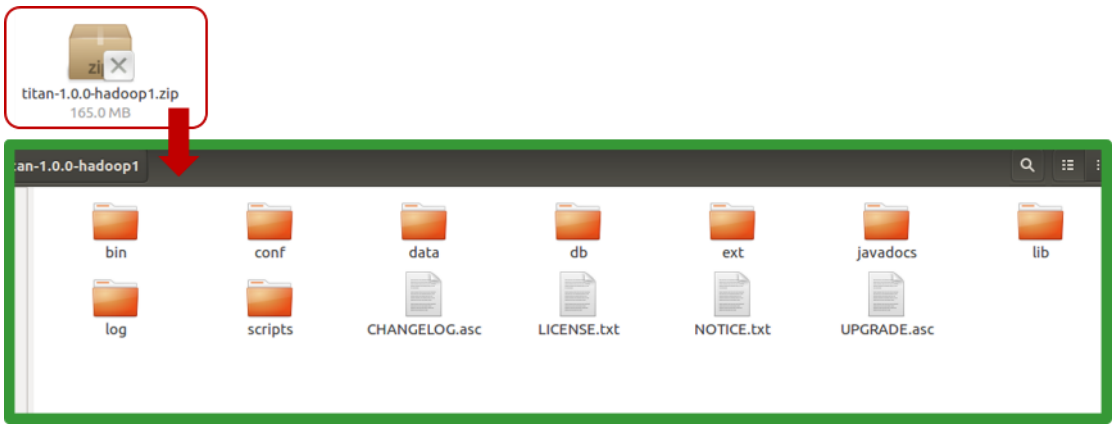
数据结构很好的表达了数据之间的关联性（ dependencies between data ），关联性计算是大数据计算的核心——通过获得数据的关联性，可以从噪音很多的海量数据中抽取有用的信息。比如，通过为购物者之间的关系建模，就能很快找到口味相似的用户，并为之推荐商品；或者在社交网络中，通过传播关系发现意见领袖。但现有的并行计算框架像 MapReduce 还无法满足复杂的关联性计算。比如，曾经有公司利用 MapReduce 进行社交用户推荐，对于 5000 万注册用户，50 亿关系对，利用 10 台机器的集群，需要超过 10 个小时的计算。

最近有许多新型的基于图的计算平台和引擎出现，来应对这种复杂的需求。

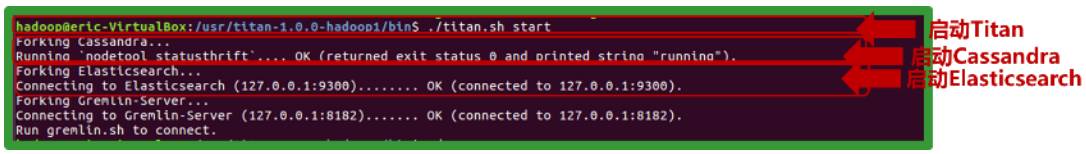
Google 为了应对图计算的需求，推出了新的“计算框架”——Pregel。CMU 给出了一个开源的版本——GraphLab，虽然二者都是对于复杂机器学习计算的处理框架，用于迭代型计算，但是二者的实现方法却采取了不同的路径——Pregel 是基于大块的消息传递机制，GraphLab 是基于内存共享机制。同样的，最近非常火的“Spark”也有支持图计算机器学习的模块——GraphX，可以实现复杂的图数据挖掘。

Titan 也是这样一个产品。技术上，Titan 并不是数据库；它是一个数据库之上的客户端库。它依赖于下面的存储引擎，比如 Cassandra 或者 Hadoop，来存储其数据。它也依赖于索引引擎，比如 Lucene、ElasticSearch 或者 Solr，来执行相关的查询。T

要安装 Titan，从官网下载它的发行包并解压（1.0 版本 Titan 建议下载 Titan-1.0.0-Hadoop1.zip，如果要使用 Hadoop2，下载 0.5.0 版本，1.0 的 Hadoop2 版本存在很大的问题）：



通过 bin 中的 titan.sh 启动 Titan：

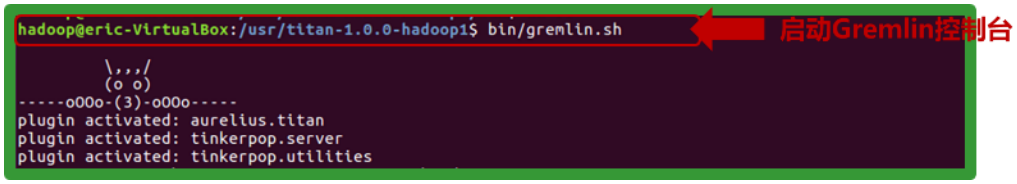


Titan 需要依赖两个基础服务：

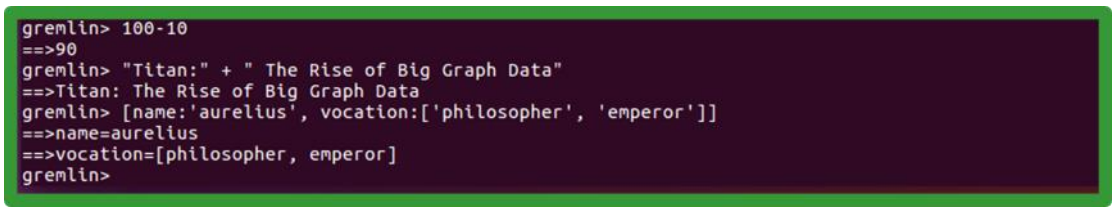
- 数据存储
- 搜索引擎

从控制台输出可以看出，Titan 发行包的默认存储引擎是 Cassandra,默认搜索服务是 Elasticsearch，并且在发行包中已经默认包含两个支持服务。

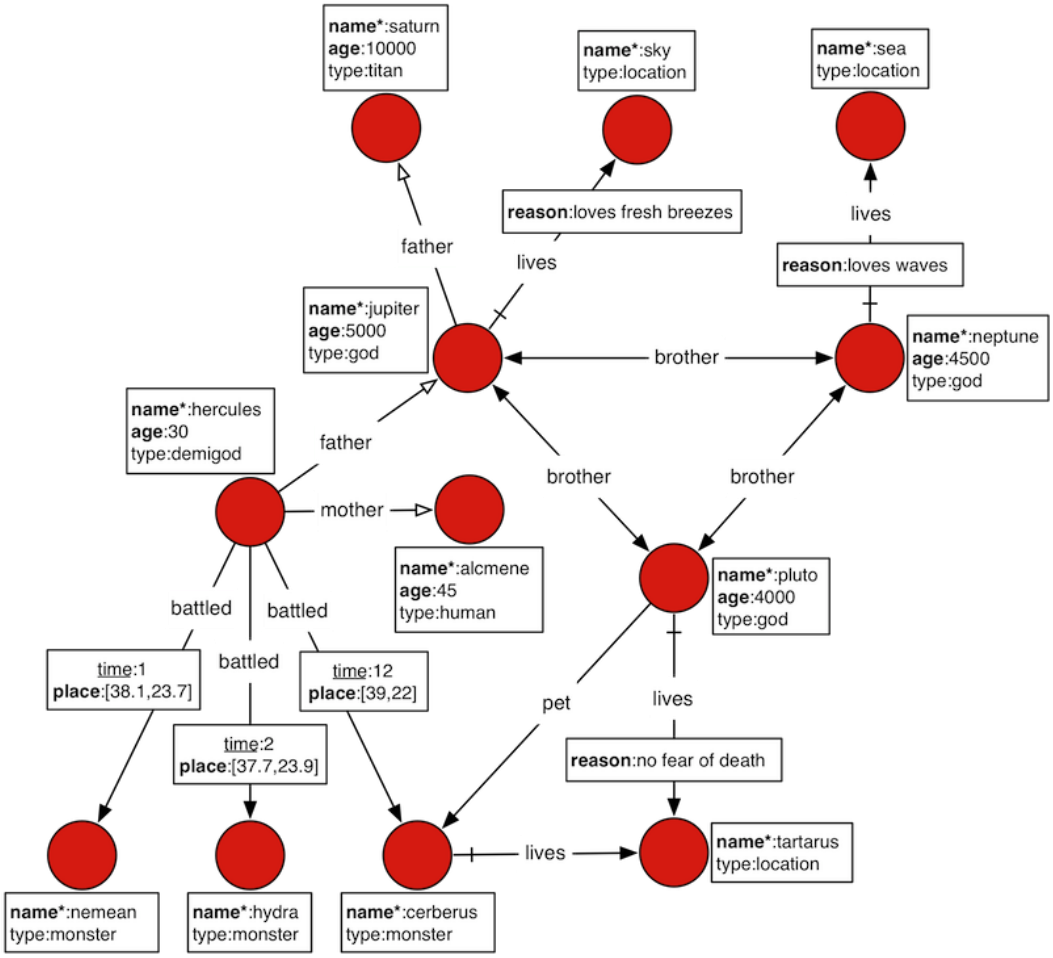
接下来可以启动 Gremlin 控制台用于进行 Titan 操作，注意：一定要在 bin 的上一层目录（即 Titan 的根目录执行 gremlin.sh，否则后续导入数据时会产生路径错误）：



Gremlin 是一个 Groovy 控制台，需要掌握一定的 Groovy 语法以保证使用，详情可以参考 Groovy 文档（http://groovy.codehaus.org/），可以直接在 Gremlin 终端中输入 Groovy 表达式并计算结果：

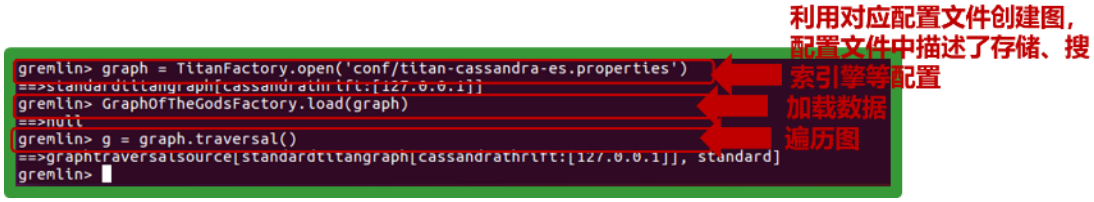


启动 Titan 后即可导入数据并进行操作，Titan 官方提供了名为“众神图谱”的图，内容如下：



图形	含义
粗体键	图形中被索引的键
带有星号的粗体键	图形中被索引的唯一值键
带有下划线的键	顶点被索引的键
空心箭头	不能重复的区域
尾部带有标记的箭头	单向区域

创建图并加载图数据：



TitanFactory.open 方法通过对应属性创建图后，利用 GraphOfTheGodsFactory.load 方法加载众神图谱数据，GraphOfTheGodsFactory 是一个数据操作类，可以通过该类学习图的基本数据加载操作.

GraphOfTheGodsFactory 的源代码可以参考以下地址：

https://github.com/thinkaurelius/titan/blob/titan09/titan-core/src/main/java/com/thinkaurelius/titan/example/GraphOfTheGodsFact

ory.java

```
public static void load(final TitanGraph graph, String mixedIndexName, boolean uniqueNameCompositeIndex) {

    //Create Schema
    TitanManagement mgmt = graph.openManagement();
    final PropertyKey name = mgmt.makePropertyKey("name").dataType(String.class).make();
    TitanManagement.IndexBuilder nameIndexBuilder = mgmt.buildIndex("name", Vertex.class).addKey(name);
    if (uniqueNameCompositeIndex)
        nameIndexBuilder.unique();
    TitanGraphIndex namei = nameIndexBuilder.buildCompositeIndex();
    mgmt.setConsistency(namei, ConsistencyModifier.LOCK);
    final PropertyKey age = mgmt.makePropertyKey("age").dataType(Integer.class).make();
    if (null != mixedIndexName)
        mgmt.buildIndex("vertices", Vertex.class).addKey(age).buildMixedIndex(mixedIndexName);

    final PropertyKey time = mgmt.makePropertyKey("time").dataType(Integer.class).make();
    final PropertyKey reason = mgmt.makePropertyKey("reason").dataType(String.class).make();
```

在图数据库中访问数据的典型模式是首先使用图索引将入口点定位到图中。这个切入点是一个元素（或元素集合） - 即顶点或边。从入口的元素，一个 Gremlin 路径可以描述通过显式图结构如何遍历图中的其他元素。

图中 name 属性上有一个唯一的索引，可以检索到 Saturn 顶点。属性映射（如 Saturn 的键/值对会被检查。在示例中， Saturn 顶点 name 属性“Saturn ”，10000 的 age 值，Titan 的 type 属性。土 Saturn 的孙子元素可以用检索遍历表示：“谁是 Saturn 的孙子？”（“father”的逆检索是“Child”）：

```
gremlin> saturn = g.V().has('name', 'saturn').next()
==>v[4344]
gremlin>
gremlin> g.V(saturn).valueMap()
==>[name:[saturn], age:[10000]]
gremlin>
gremlin> g.V(saturn).in('father').in('father').values('name')
==>hercules
gremlin>
```

place 属性也在图的索引中，它是一个边属性。因此，Titan 可以在图形中索引边。可以查询上帝图谱中所有发生在 Athens （纬度：37.97 和精度：23.72）50 公里内的事件：

```
gremlin> g.E().has('place', geoWithin(Geoshape.circle(37.97, 23.72, 50)))
==>e[54z-6c8-9hx-38g][8216-battled->4192]
==>e[5j7-6c8-9hx-368][8216-battled->4112]
gremlin>
gremlin> g.E().has('place', geoWithin(Geoshape.circle(37.97, 23.72, 50))).as('source').inV().as('god2').select('sour
ce').outV().as('god1').select('god1', 'god2').by('name')
==>[god1:hercules, god2:nemean]
==>[god1:hercules, god2:hydra]
gremlin>
```

图形索引是泰坦中的一种索引结构。图形索引在问询满足一个或多个约束条件的所有顶点(g.V)或所有边(g.E)时由 Titan 自动选择。Titan 中另一个重要的内容是顶点中心索引，顶点中心索引被用来加快图的遍历。

在之前的示例中， Saturn 的孙子是 Hercules，这可以用一个循环来处理，在本质上， Hercules 是一个离 Saturn 节点两步并被名为“father”的边构成的路径连接的节点：

```
gremlin> hercules = g.V(saturn).repeat(__.in('father')).times(2).next()
==>v[8216]
```

Hercules 是一个半神，要想确定 Hercules 有一半是神而另一半是人，那么他的父辈的出身就必须被确定，可以从 Hercules 顶点遍历到他的父亲顶点和母亲顶点。最终可以确定他们的身份：神(god)和人(human)：

```
gremlin> g.V(hercules).out('father', 'mother')
==>v[4168]
==>v[4296]
gremlin>
```

```
gremlin> g.V(hercules).out('father', 'mother').values('name')
==>jupiter
==>alcmene
gremlin>
```

```
gremlin> g.V(hercules).out('father', 'mother').label()
==>god
==>human
gremlin>
```

```
gremlin> hercules.label()
==>demigod
gremlin>
```

到目前为止，示例一直在描述罗马众神的遗传路线，属性图模型的表现力足以代表多个类型的事物和关系，因此，众神图谱还记录了 Hercules 的各种英雄事迹及他的著名的 12 门徒。Hercules 参与了 Athens 附近的两个战斗。可以通过检索 Hercules 的战斗边来遍历这些事件：

```
gremlin> g.V(hercules).out('battled')
==>v[4112]
==>v[4192]
==>v[8392]
```

```
gremlin> g.V(hercules).out('battled').valueMap()
==>[name:[hydra]]
==>[name:[nemean]]
==>[name:[cerberus]]
gremlin>
```

```
gremlin> g.V(hercules).outE('battled').has('time', gt(1)).inV().values('name')
==>cerberus
==>hydra
gremlin>
```

```
gremlin> g.V(hercules).outE('battled').has('time', gt(1)).inV().values('name').toString()
==>[GraphStep([v[8216]],vertex), VertexStep(OUT,[battled],edge), HasStep([time.gt(1)]), EdgeVertexStep(IN), PropertiesStep([name],value)]
gremlin>
```

谁和 pluto 一起居住：

```
gremlin> pluto = g.V().has('name', 'pluto').next()
==>v[4264]
gremlin>
```

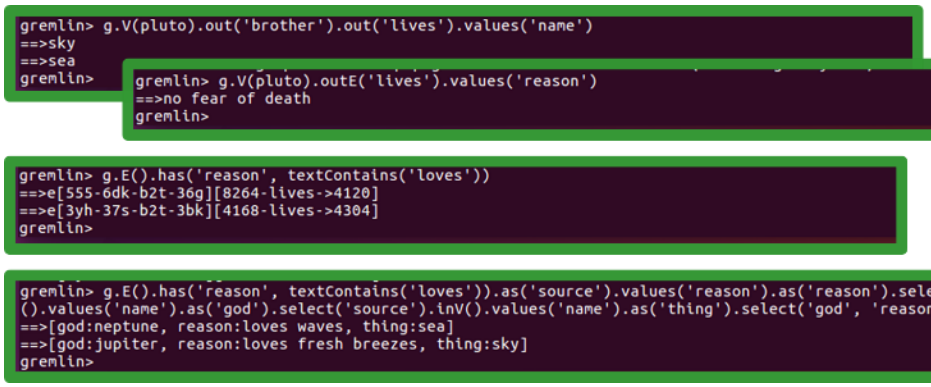
```
gremlin> g.V(pluto).out('lives').in('lives').values('name')
==>pluto
==>cerberus
gremlin>
```

去除掉 pluto 自己：

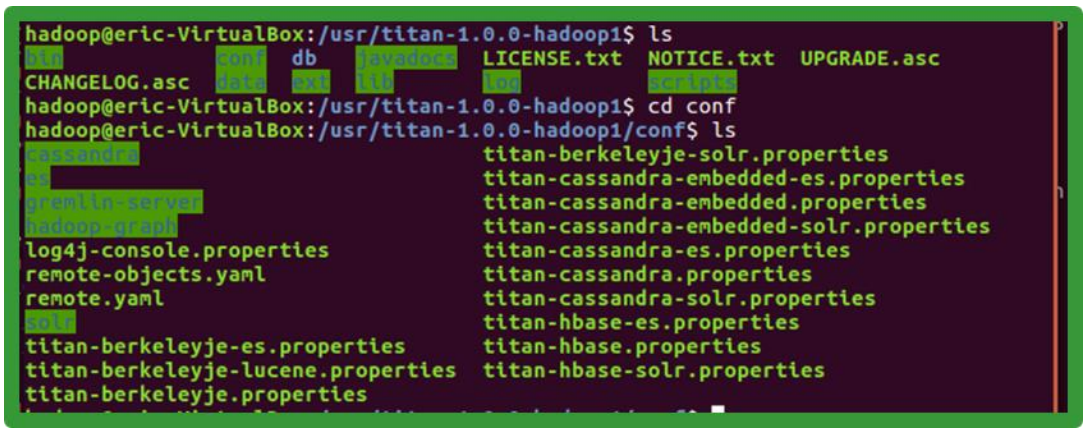
```
gremlin> g.V(pluto).out('lives').in('lives').where(is(neq(pluto))).values('name')
==>cerberus
gremlin>
```

```
gremlin> g.V(pluto).as('x').out('lives').in('lives').where(neq('x')).values('name')
==>cerberus
gremlin>
```

pluto 由于不关心死亡，因此居住在 Tartarus，另一方面，他的兄弟们因为对特定事物的爱好选择了各自的位置：



如果希望改变存储和查询引擎，如将 Titan 连接到 HBase 或 Solr，只需要修改 conf 文件夹中对应的配置文件，提供对应服务的主机等信息：



创建图时加载不同的配置文件即可。

第2节 Neo4J

Neo4j 是一个高性能的 NoSQL 图形数据库。Neo4j 使用图（graph）相关的概念来描述数据模型，把数据保存为图中的节点以及节点之间的关系。很多应用中数据之间的关系，可以很直接地使用图中节点和关系的概念来建模。对于这样的应用，使用 Neo4j 来存储数据会非常的自然，要优于使用关系数据库。

数据存储一般是应用开发中不可或缺的组成部分。应用运行中产生的和所需要的数据被以特定的格式持久化下来。应用开发中很常见的一项任务是在应用本身的领域对象模型与数据存储格式之间进行相互转换。如果数据存储格式与领域对象模型之间比较相似，那么进行转换所需的映射关系更加自然，实现起来也更加容易。

对于一个特定的应用来说，其领域对象模型由应用本身的特征来决定，一般采用最自然和最直观的方式来进行建模。所以恰当的选择数据存储格式就显得很重要。

目前最常见的数据存储格式是关系数据库。关系数据库通过实体 – 关系模型（E-R 模型）来进行建模，即以表和表之间的关系来建模。关系数据库适合用

来存储数据条目的类型同构的表格型数据。如果领域对象模型中不同对象之间的关系比较复杂，则需要使用繁琐的对象关系映射技术来进行转换。

对于很多应用来说，其领域对象模型并不适合于转换成关系数据库形式来存储。这也是非关系型数据库（NoSQL）得以流行的原因。NoSQL 数据库的种类很多，包括键值对数据库、面向文档数据库和图形数据库等。

Neo4j 是最重要的图形数据库之一。Neo4j 使用数据结构中图（graph）的概念来进行建模。Neo4j 中两个最基本的概念是节点和边。节点表示实体，边则表示实体之间的关系。节点和边都可以有自己的属性。不同实体通过各种不同的关系关联起来，形成复杂的对象图。Neo4j 同时提供了在对象图上进行查找和遍历的功能。

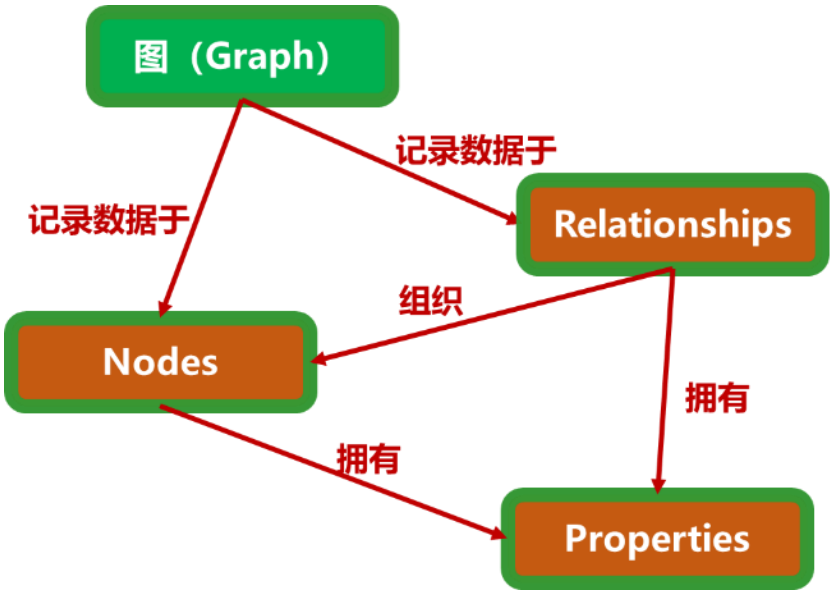
对于很多应用来说，其中的领域对象模型本身就是一个图结构。对于这样的应用，使用 Neo4j 这样的图形数据库进行存储是最适合的，因为在进行模型转换时的代价最小。

以基于社交网络的应用为例，用户作为应用中的实体，通过不同的关系关联在一起，如亲人关系、朋友关系和同事关系等。不同的关系有不同的属性。比如同事关系所包含的属性包括所在公司的名称、开始的时间和结束的时间等。对于这样的应用，使用 Neo4j 来进行数据存储的话，不仅实现起来简单，后期的维护成本也比较低。

Neo4j 使用“图”这种最通用的数据结构来对数据进行建模使得 Neo4j 的数据模型在表达能力上非常强。链表、树和散列表等数据结构都可以抽象成用图来表示。

Neo4j 同时具有一般数据库的基本特性，包括事务支持、高可用性和高性能等。Neo4j 已经在很多生产环境中得到了应用。流行的云应用开发平台 Heroku 也提供了 Neo4j 作为可选的扩展。

一个图由无数的节点和关系组成，关系可以将节点组织成任意的结构，允许一张图被组织成一个列表，一棵树，一张地图，或者一个复杂的实体 - 这个实体本身也是由复杂的，关系高度关联的结构组成：



Neo4J 支持 Linux 和 Windows，在官网下载对应的发行包安装即可：

- Windows：运行安装包进行安装
- Linux：解压发行包

Neo4j 中也定义了自己的查询语言：Cypher。

Cypher 是一个描述性的图形查询语言，允许不必编写图形结构的遍历代码对图形存储有表现力和效率的查询。Cypher 还在继续发展和成熟，这也就意味着有可能会出现语法的变化。同时也意味着作为组件没有经历严格的性能测试。

Cypher 设计的目的是一个类人类查询语言，适合于开发者和在数据库上做点对点模式查询的专业操作人员。它的构念是基于英语单词和灵巧的图解。

Cyper 通过一系列不同的方法和建立于确定的实践为表达查询而激发的。许多关键字如 like 和 order by 是受 SQL 的启发。模式匹配的表达式来自于 SPARQL。正则表达式匹配实现实用 Scala programming language 语言。

Cypher 是一个申明式的语言。对比命令式语言如 Java 和脚本语言如 Gremlin 和 JRuby，它的焦点在于从图中如何找回（what to retrieve），而不是怎么去做。这使得在不对用户公布的实现细节里关心的是怎么优化查询。

Cypher 查询语言包含几个明显的组成部分：

START，在图中的开始点，通过元素的 ID 或所以查找获得。

MATCH，图形的匹配模式，束缚于开始点。

WHERE，过滤条件。

RETURN，返回所需要的结果。

要进行图数据库操作，首先需要创建图结构，创建语句 CREATE 的使用描述如下：

创建单个节点（注意有个空格）：create (n)

创建带标签的节点（ID:n，label:Person）：create (n:Person)

创建带两个标签的节点：create (n:Person:Student)

创建带标签、属性的节点： create (n:Person {name:"weiw",age:23})

返回创建的节点：create (a {name:"Tom"}) return a

创建关系（两个节点之间的关系）：

```
match (a:Person),(b:Person) where a.name="zhangs" and b.name="lisi" create (a)-[r:RELTYPE]->(b) return r
```

创建关系的同时设置属性：

```
match (a:Person),(b:Person) where a.name="zhangs" and b.name="lisi" create (a)-[r:RELTYPE {name:a.name +"<->" + b.name}]->(b) return r
```

完整创建：三个节点两个关系：

```
create p=(an {name:"an"})-[:WORKS_AT]->(neo)<-[:WORKS_AT]-(mach {name:"mach"}) return p;
```

使用 Merge 操作进行创建操作的描述如下：

对不存在的节点创建，存在的节点返回：

```
merge (robert:Critic) return robert,labels(robert);
```

单个属性节点：merge (charlie {name:"Charlie",age:10}) return charlie;

带标签和属性的单个节点：merge (michel:Person {name:"michelDoug"}) return michel;

如果要创建节点就设置属性：merge on create：

```
merge (keanu:Person {name:"Keanu"}) on create set keanu.created=timestamp() return keanu;
```

如果找到节点就设置属性：merge on match：

```
merge (person:Person) on match set person.found=true return person;
```

如果找到就设置属性，没找到创建节点并设置属性：merge on create on match:

```
merge (keanu:Person {name:"Keanu"}) on create set keanu.created=timestamp() on match set keanu.lastSeen=timestamp() return keanu;
```

merge on match 多个属性，如果没有属性则创建：

```
merge (person:Person) on match set
person.found=true,person.lastAccessed=timestamp() return person;
```

merge 关系：

```
match (charlie:Person {name:"Charlie"}),(wall:Movie {title:"Wall"})
merge (charlie)-[r:ACTED_AT]->(wall)  return r;
```

merge 多重关系：

```
MATCH (oliver:Person { name:'Oliver Stone' }),(reiner:Person
{ name:'Rob Reiner' })
MERGE (oliver)-[:DIRECTED]->(movie:Movie)<-[:ACTED_IN]-(reiner)
RETURN movie
```

merge 非直接关系：

```
MATCH (charlie:Person { name:'Charlie Sheen' }),(oliver:Person
{ name:'Oliver Stone' })
MERGE (charlie)-[r:KNOWS]-(oliver) RETURN r
```

merge 上使用唯一性约束：

```
CREATE CONSTRAINT ON (n:Person) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:Person) ASSERT n.role IS UNIQUE;
MERGE (laurence:Person { name: 'Laurence Fishburne' }) RETURN
laurence
```

Set 语句的使用如下：

用于更新一个节点和关系的标签或属性：

```
create (n { name: 'Andres' }) ;
MATCH (n { name: 'Andres' }) SET n.surname = 'Taylor' RETURN n;
```

删除属性：MATCH (n { name: 'Andres' }) SET n.name = NULL RETURN n

在节点和关系之间复制属性：MATCH (at { name: 'Andres' }),(pn { name: 'Peter' }) SET at = pn RETURN at, pn;

从 map 添加属性：MATCH (peter { name: 'Peter' }) SET peter += { hungry:

TRUE , position: 'Entrepreneur' }

设置多个属性：MATCH (n { name: 'Andres' }) SET n.position = 'Developer', n.surname = 'Taylor'

在节点上加标签：

```
MATCH (n { name: 'Stefan' }) SET n :German RETURN n
MATCH (n { name: 'Emil' }) SET n :Swedish:Bossman RETURN n
```

使用 DELETE 删除节点和关系的操作如下：

删除单个节点：MATCH (n:Useless) DELETE n;

删除节点和连接它的关系：MATCH (n { name: 'Andres' })-[r]-() DELETE n, r

删除所有节点和关系：MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,r

使用 REMOVE 删除标签和属性的操作如下：

删除属性：MATCH (andres { name: 'Andres' }) REMOVE andres.age RETURN andres;

删除节点的标签：MATCH (n { name: 'Peter' }) REMOVE n:German RETURN n;

删除多重标签：MATCH (n { name: 'Peter' }) REMOVE n:German:Swedish RETURN n;

其他的常见操作如下：

FOREACH:

为所有节点设置 mark 属性：MATCH p =(begin)-[*]->(END) WHERE begin.name='A' AND END .name='D' FOREACH (n IN nodes(p)| SET n.marked = TRUE)

CREATE UNIQUE:

创建唯一性节点：MATCH (root { name: 'root' }) CREATE UNIQUE (root)-[:LOVES]-(someone) RETURN someone

IMPORT CSV:

LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.3/csv/import/persons.csv" AS csvLine CREATE (p:Person { id: toInt(csvLine.id), name: csvLine.name }) ;

csv 文件的格式如下：

```
id       name
```

```
1        Charlie Sheen
2        Oliver Stone
3        Michael Douglas
4        Martin Sheen
5        Morgan Freeman
```

使用 MATCH 读取数据参考操作如下:

查询所有节点：MATCH (n) RETURN n

查询指定标签的节点：MATCH (movie:Movie) RETURN movie;

关联节点：MATCH (director { name:'Oliver Stone' })--(movie) RETURN movie.title

查询标签：MATCH (charlie:Person { name:'Charlie Sheen' })--(movie:Movie) RETURN movie

关系查询：MATCH (martin { name:'Martin Sheen' })-->(movie) RETURN movie.title

MATCH (martin { name:'Martin Sheen' })-[r]->(movie) RETURN r

通过关系类型查询：MATCH (wallstreet { title:'Wall Street' })<-[:ACTED_IN]-(actor) RETURN actor;

OPTIONAL MATCH 与 match 类似，只是如果没有匹配上，则将使用 null 作为没有匹配上的模式。类似于 SQL 中的外连接。

匹配关系：match (a:Movie {title:"Wall Street"}) optional match (a)-->(x) return x; 如果没有返回 null。

匹配属性：match (a:Movie {title:"Wall Street"}) optional match (a)-->(x) return x,x.name;

和 SQL 类似，Neo4J 中也使用 WHERE 来进行数据的过滤，例如：

```
MATCH (n)  WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = "Tobias") OR NOT (n.name = "Tobias" OR  n.name="Peter")  RETURN n;
```

更多 WHERE 参考操作如下所示：

过滤标签：MATCH (n) WHERE n:Swedish RETURN n;

过滤属性：

```
MATCH (n)  WHERE n.age < 30  RETURN n;

MATCH (n)  WHERE HAS (n.belt)  RETURN n;
```

正则：MATCH (n) WHERE n.name =~ 'Tob.*' RETURN n;

使用 not：MATCH (persons),(peter { name: 'Peter' }) WHERE NOT (persons)-->(peter) RETURN persons

使用属性：MATCH (n) WHERE (n)-[:KNOWS]-({ name:'Tobias' }) RETURN n

关系类型：MATCH (n)-[r]->() WHERE n.name='Andres' AND type(r)=~'K.*' RETURN r

使用 IN：

```
MATCH (a) WHERE a.name IN ["Peter", "Tobias"] RETURN a

MATCH (n) WHERE n.belt = 'white' RETURN n

MATCH (n) WHERE n.belt = 'white' OR n.belt IS NULL RETURN n ORDER BY n.name
```

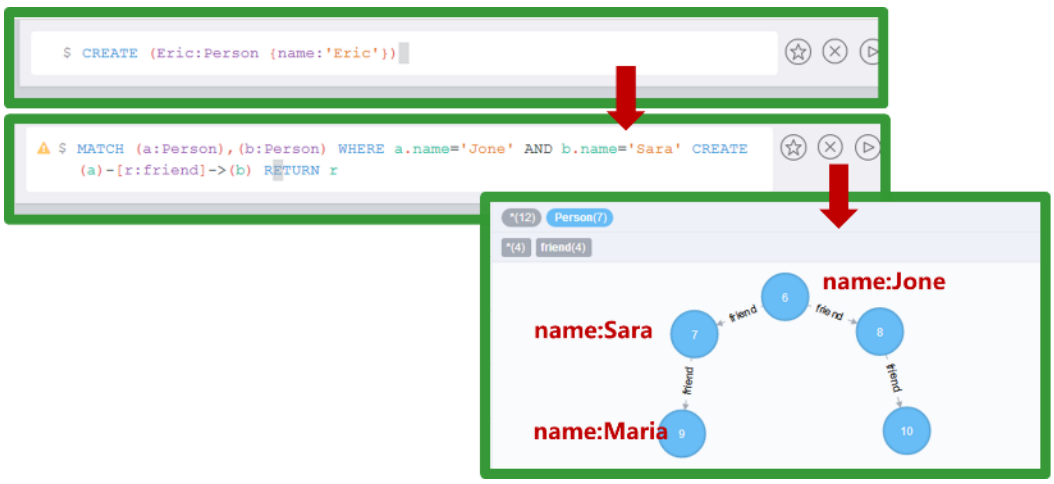
过滤 NULL：MATCH (person) WHERE person.name = 'Peter' AND person.belt IS NULL RETURN person

START 的使用示例如下：

```
START n=node:nodes(name = "A") RETURN n

START r=relationship:rels(name = "Andrés") RETURN r

START n=node:nodes("name:A") RETURN n
```

更多 Cyper 的语法操作参见 Neo4J 官方网站的完整文档说明。

Neo4J 启动后，通过 http://127.0.0.1:7474 访问控制台，能够执行 Cypher 语句并显示图状态：



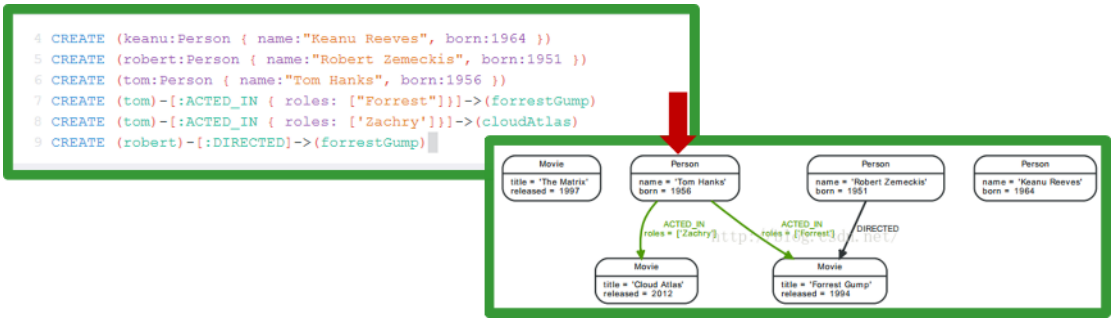
一下是一些 Neo4J 的示例操作，首先查找 Jone 的朋友的朋友：

查找的朋友的名字以 S 开头的节点：

查找第一个节点：

创建并列举现有的索引：

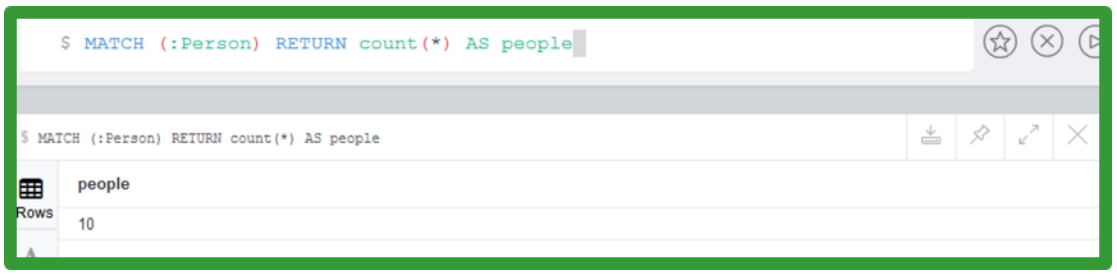
另一个示例是电影和演员的关系图，首先创建示例图：



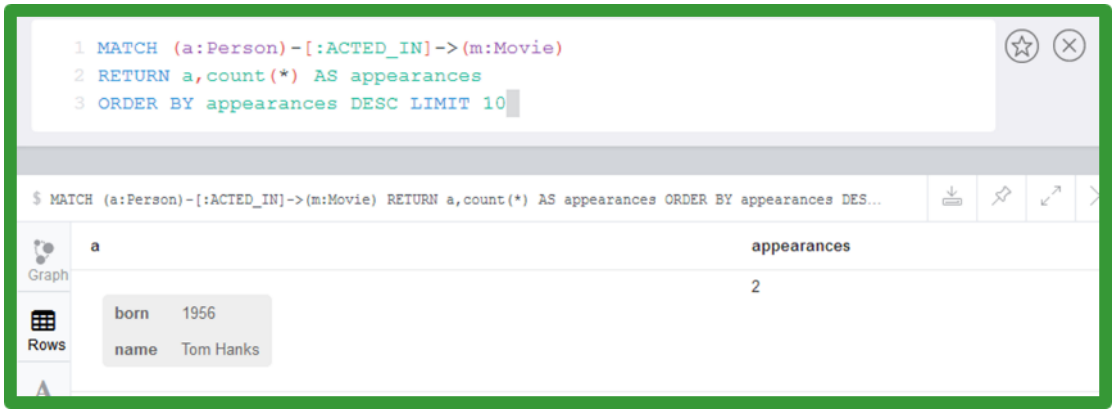
通过属性过滤数据：

定制返回结果：

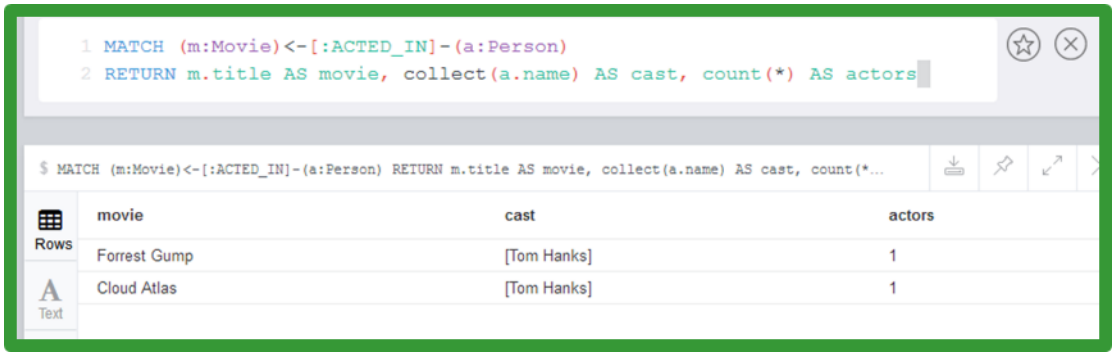
聚合函数使用示例：



排序和分页的用法：

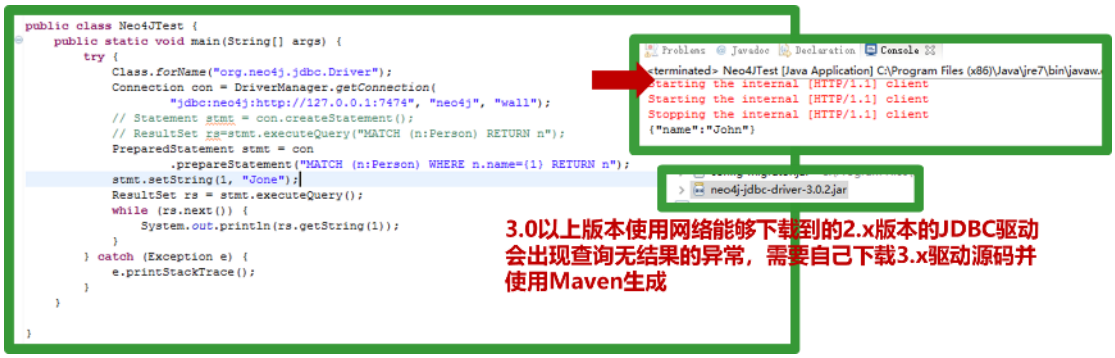


聚集操作如下：



集合操作示例如下：

通过 JDBC 驱动可以方便的访问 Neo4J：



在 JDBC 代码中可以使用普通 Statement 和预编译 Statement，使用预编译 Statement 是，占位符不能是？，而应该是{1}{2}…{n}的形式。Neo4J 还提供了标准的 HTTP API，参见 Neo4J 文档。

第17章　基于 CDH 的大数据平台运维

第1节 准备工作

为了保障生产环境中系统运行的稳定性，在本节中安装 CentOS6.7，它比 Ubuntu 更适合在企业环境中作为服务器使用。安装系统后，使用 root 账户登录系统。

安全增强式 SELinux（SELinux, Security-Enhanced Linux）是一种强制访问控制（mandatory access control）的实现，建议关闭 SELinux，否则将导致：

- 使用在浏览器上无法查看目录文件列表
- 安装 MySQL 后启动失败

先临时关闭 SELinux，设置 SELinux 成为 permissive 模式（setenforce 1 设置 SELinux 成为 enforcing 模式）

```
[root@localhost ~]$ setenforce 0
```

修改系统的配置文件（这个操作将在下次重启机器生效）：

修改 /etc/selinux/config 文件，将 SELINUX=enforcing 改为 SELINUX=disabled。

配置 yum 本地源需要使用 http 服务，所以将安装包拷贝到 http 服务目录 /var/www/html 下：

```
[root@localhost ~]$ mkdir -p /var/www/html
```

使用 ftp 工具，将所有安装组件分发源拷贝到此目录下。

由于此时还未配置离线 yum 源，所有只能使用 rpm 安装 httpd，安装 httpd 服务，需要安装 apr-*.rpm，apr-util-*.rpm，apr-util-ldap-*.rpm，http-tools-*.rpm，http-*.rpm。

如果系统安装 openssl 版本是 1.0.0 版本，需要更新到 1.0.1e，否则忽略此项：

```
[root@localhost ~]$ rpm -Uvh openssl-1.0.1e-15.el6.x86_64.rpm
```

开始安装：

```
[root@localhost ~]$ rpm -ivh apr-1.3.9-5.el6_2.x86_64.rpm
[root@localhost ~]$ rpm -ivh apr-util-1.3.9-3.el6_0.1.x86_64.rpm
```

```
[root@localhost ~]$ rpm -ivh apr-util-ldap-1.3.9-3.el6_0.1.x86_64.rpm

[root@localhost ~]$ rpm -ivh httpd-tools-2.2.15-31.el6.centos.x86_64.rpm

[root@localhost ~]$ rpm -ivh httpd-2.2.15-31.el6.centos.x86_64.rpm
```

安装完毕后，启动并设置开机启动 httpd 服务:

```
[root@localhost ~]$ service httpd start

[root@localhost ~]$ chkconfig httpd on
```

输出如下，不影响使用：

```
Starting httpd: httpd: Could not reliably determine the server's fully qualified domain name, using 127.0.0.1 for ServerName
[  OK  ]
```

在浏览器上访问 http://localhost:80，出现 Apache 页面为正常。然后可以访问安装包目录：

```
http://localhost:80/rpm

http://localhost:80/base

http://localhost:80/common
```

使用 yum 为安装工具，需要选择一个服务器，创建局域网 yum 源，并在客户端配置 yum Repository，指向我们自己创建的 yum 源服务。创建 yum 源仓库描述文件，执行以下操作：

```
[root@localhost ~]$ rpm -ivh /var/www/html/rpm/deltarpm-3.5-0.5.20090913git.el6.x86_64.rpm

[root@localhost ~]$ rpm -ivh /var/www/html/rpm/python-deltarpm-3.5-0.5.20090913git.el6.x86_64.rpm

[root@localhost ~]$ rpm -ivh /var/www/html/rpm/createrepo-0.9.9-18.el6.noarch.rpm
```

拷贝所有安装包以后，进入 yum 源的 rpm 列表目录，执行以下语句创建 yum 源文件：

```
[root@localhost ~]$ chmod 755 /var/www/html/rpm -R

[root@localhost ~]$ createrepo /var/www/html/rpm
```

```
[root@localhost ~]$ chmod 755 /var/www/html/base -R
[root@localhost ~]$ createrepo /var/www/html/base
[root@localhost ~]$ chmod 755 /var/www/html/common/CDH/CDH5.2.0/cm -R
[root@localhost ~]$ createrepo /var/www/html/common/CDH/CDH5.2.0/cm
```

后面每一次变更文件后，需要重新执行以下操作。

移除/etc/yum.repo.d/下面的所有.repo 配置文件或者将该目录下的配置文件重命名（后缀名称不为 repo 即可）：

```
[root@localhost ~]$ rm -rf /etc/yum.repos.d/*
```

添加自己的 yum 源配置文件：

```
[root@localhost ~]$ vim /etc/yum.repos.d/myrepo.repo
```

内容如下（一个目录配置一个地址）：

```
[rpm]
name=rpm
baseurl=http://localhost/rpm
enabled=1
gpgcheck=0

[base]
name=base
baseurl=http://localhost/base
enabled=1
gpgcheck=0

[cdh5.2.0]
name=cdh5.2.0
baseurl=http://localhost/cm
enabled=1
```

```
gpgcheck=0
```

清除旧的 yum list 影响，需清理 yum 缓存(这一步很重要)：

```
[root@localhost ~]$ yum clean all
```

接下来可以用 yum list 查看有效的包信息，如果配置成功，通过 yum list 可以看到安装包列表信息：

```
[root@localhost ~]$ yum list
```

第2节 支持库安装

查询是否已经安装了 OpenJDK：

```
[root@localhost ~]$ rpm -qa | grep java
```

显示如下，说明自带安装了 OpenJDK：

```
java-1.6.0-openjdk-1.6.0.0-1.50.1.11.5.el6_3.x86_64
tzdata-java-2012j-1.el6.noarch
java-1.7.0-openjdk-1.7.0.9-2.3.4.1.el6_3.x86_64
```

如果存在，卸载它们，这里使用 yum 卸载：

```
[root@localhost ~]$ rpm -qa | grep java | xargs rpm -e --nodeps
[root@localhost ~]$ yum clean all
```

安装 Oracle JDK：

```
[root@localhost ~]$ yum -y install oracle-j2sdk1.7
```

配置 Java 环境变量：

```
[root@localhost ~]$ vim /etc/profile
```

在文件最后添加以下内容：

```
export JAVA_HOME=/usr/java/jdk1.7.0_67-cloudera
export PATH=.:$JAVA_HOME/bin:$PATH
export
```

```
CLASSPATH=.:$JAVA_HOME/lib/tools.jar:$JAVA_HOME/lib/dt.jar:$CLASSPATH
```

刷新配置：

```
[root@localhost ~]$ source /etc/profile
```

安装 c 编译器 gcc：

```
[root@localhost ~]$ yum -y install gcc
```

安装 F77 编译器 gcc-gfortran：

```
[root@localhost ~]$ yum -y install gcc-gfortran
```

安装 c++编译器 gcc-c++：

```
[root@localhost ~]$ yum -y install gcc-c++
```

安装 X11 库文件：

```
[root@localhost ~]$ yum -y install libXt-devel
```

安装 readline-devel 库文件：

```
[root@localhost ~]$ yum -y install readline-devel
```

第3节 MySQL 安装

使用包管理器安装：

```
[root@localhost ~]$ yum install -y MySQL-server MySQL-client
```

如果有提示 mysql-libs 冲突问题，则安装 MySQL-shared-compat 包即可：

```
[root@localhost ~]$ yum install -y MySQL-shared-compat
```

拷贝配置文件模板，修改配置文件：

```
[root@localhost ~]$ cp /usr/share/mysql/my-medium.cnf /etc/my.cnf
```

根据实际情况修改 my.cnf 文件，保存退出，数据库初始化：

```
[root@localhost ~]$ mysql_install_db
```

对各个数据目录授权：

```
[root@localhost ~]$ chown -R mysql:mysql /etc/my.cnf
[root@localhost ~]$ chown -R mysql:mysql /var/lib/mysql
```

启动 mysql 服务，并设置开机启动：

```
[root@localhost ~]$ service mysql start && chkconfig mysql on
```

设置初始登陆账户和密码（如果 socket 参数有变化需要指定-S 参数，默认 /var/lib/mysql/mysql.sock，下同）：

```
[root@localhost ~]$ mysqladmin -u root password 'root123456!'
```

登陆 mysql 数据库：

```
[root@localhost ~]$ mysql -uroot -proot123456!
```

集群中的节点有需要访问 Mysql 数据库的需要 mysql-java 驱动的支持，在的主机上执行以下脚本：

```
[root@localhost ~]$ mkdir -p /usr/share/java/
[root@localhost ~]$ cp /var/www/html/common/mysql/driver/mysql-connector-java-5.1.34-bin.jar /usr/share/java/mysql-connector-java.jar
```

注意：

这里的 MySQL 连接器无法使用 yum 安装，使用 yum 安装 MySQL 连接器安装的 JDK 为 openJDK，而且系统会将 JDK 环境修改成 openJDK。

名称必须为 mysql-connector-java.jar。

确保对 mysql-connector-java.jar 包有读取权限，否则会造成比如 cloudera-scm-server 无法读取驱动而启动失败：

```
root@localhost ~]$ chmod 755 /usr/share/java/mysql-connector-java.jar
```

第4节 集群管理组件安装配置

使用 yum 安装：

```
[root@localhost ~]$ yum install -y cloudera-manager-server
```

在 mysql 服务器上创建临时访问权限：

```
mysql>grant all on scm.* to 'scm'@'${本机 IP}' identified by 'scm' with grant option;
```

在 localhost 主机运行以下命令(scm 数据库必须不存在，注意：如果 Cloudera Manager Server 的元数据已存在，仅仅是重装 Cloudera Manager Server 服务，则只需要配置 db.properties 文件即可)：

```
[root@localhost ~]$ /usr/share/cmf/schema/scm_prepare_database.sh mysql -h ${MySQL 主机 IP} -uscm -pscm --scm-host ${本机 IP} scm scm scm
```

- scm(1)为数据库名称
- scm(2)为数据库用户
- scm(3)为数据库密码

执行结果类似“All done, your SCM database is configured correctly”信息，则为配置成功。

查看元数据配置是否正确，在 localhost 主机执行以下命令：

```
[root@localhost ~]$ cat /etc/cloudera-scm-server/db.properties
```

启动集群管理服务：

```
[root@localhost ~]$ service cloudera-scm-server start && chkconfig cloudera-scm-server on
```

查看日志：

```
[root@localhost ~]$ tail -f /var/log/cloudera-scm-server/cloudera-scm-server.log
```

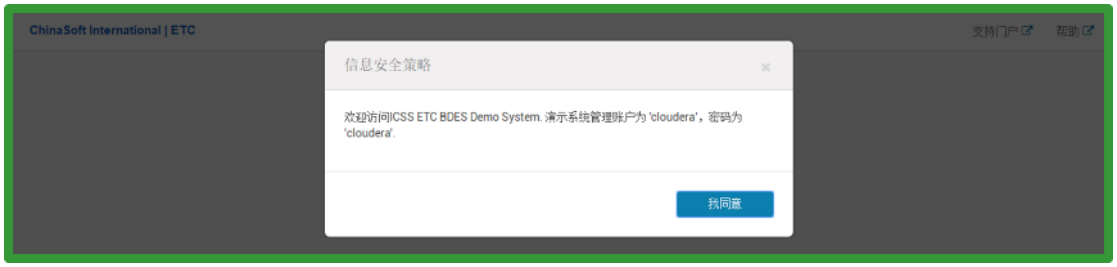
登录集群管理界面，按照界面上的提示添加计算节点和角色。默认的集群 WEB 管理服务器为当前的 MASTER 节点，默认端口为 7180。

第5节 集群管理维护工具的基本使用

集群管理维护工具对 Hadoop 生态中的每个部件都提供了细粒度的可视化和控制，通过集群监控管理工具，运维人员得以提高集群的性能，提升服务质量，提高合规性并降低管理成本。

监控管理工具基本功能主要可以分为四大模块：管理功能、监控功能、诊断功能和集成功能。

在能够通过网络连接到试验系统集群主节点的终端设备上，通过浏览器访问集群监控管理工具的 WEB 管理界面，地址为 http://集群主节点 IP 或域名:7180：

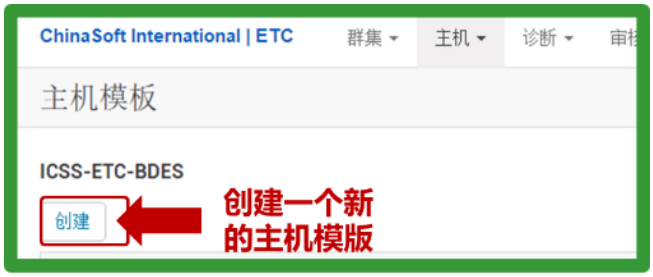


试验系统集群默认的用户名为 cloudera，密码为 cloudera，该账户拥有集群的最高管理权限，登录后可以在管理工具中对用户账户进行管理。

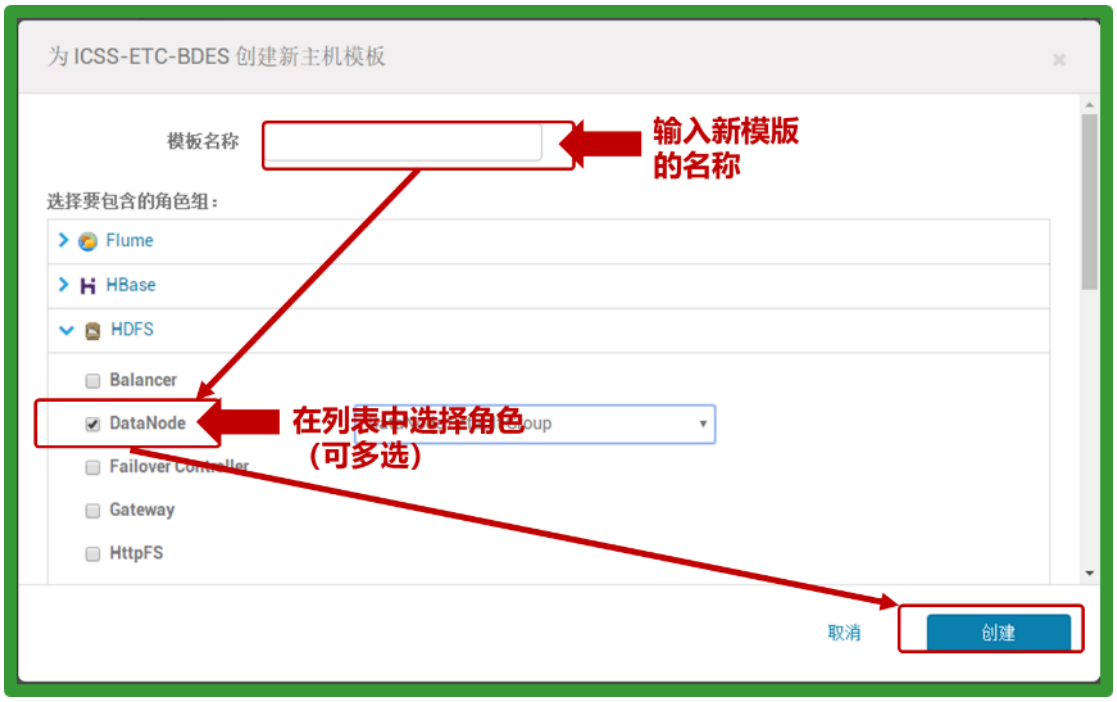
由于 Hadoop 是一个分布式的系统，由多个服务器组成一个集群，相对于传统的单节点运行的应用系统，由于节点数的增多。Hadoop 的管理更加复杂。试验系统集成的监控管理工具提供了强大的 Hadoop 集群部署能力，能够批量的自动化部署节点。安装一个 Hadoop 集群只需要简单的几个步骤。

在管理工具主界面中，选择【主机】菜单，点击【主机模版】链接可以预先定制集群中特定主机能够扮演的角色，如 HDFS Datanode，HBase RegionServer 等：

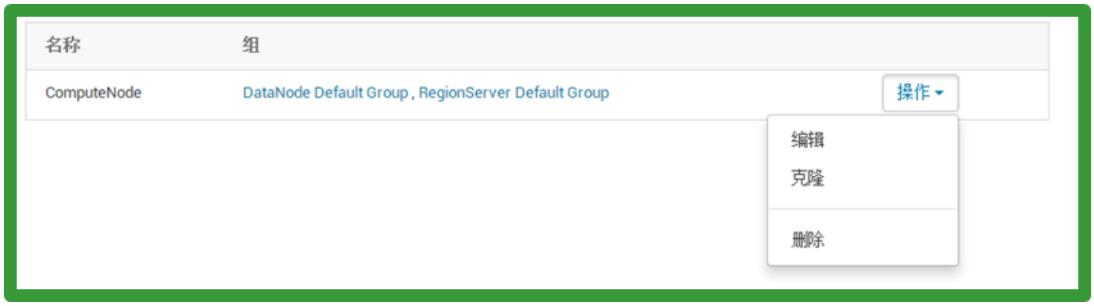
点击【创建】按钮可以创建一个新的主机模版：



在创建主机模板的界面输入模板名称并选择角色列表即可创建一个新的主机角色模板：



在现有的模板列表中，还能对既有的主机模板进行编辑、克隆和删除的操作：

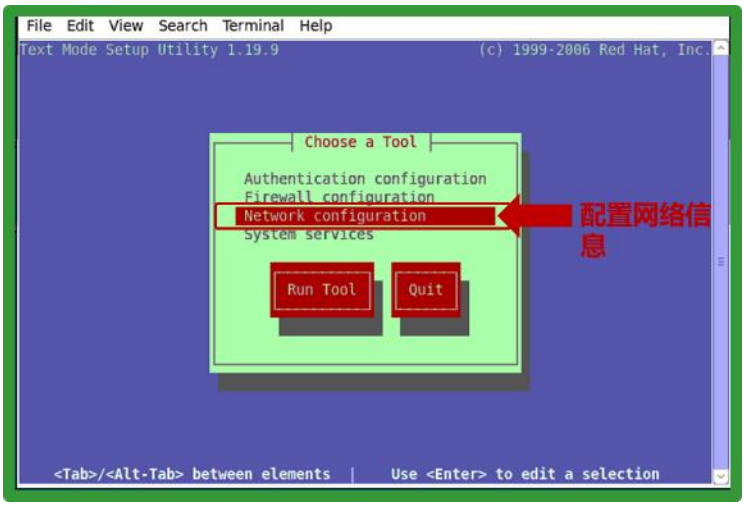


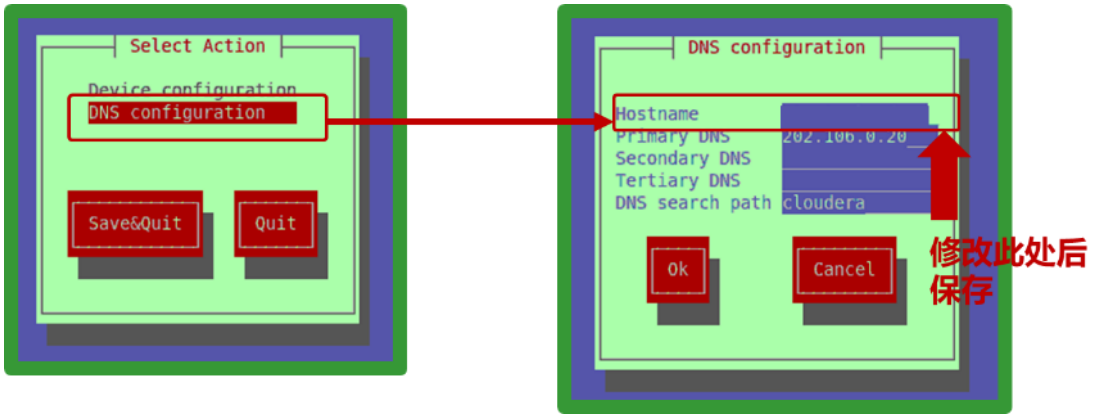
修改节点的主机名，虚拟机的主机名可以使用两种方式：

直接修改配置文件，主机名信息位于如下配置文件中：/etc/sysconfig/network，打开该文件，里面有一行`HOSTNAME=localhost.localdomain，修改 localhost.localdomain 为期望的主机名即可。

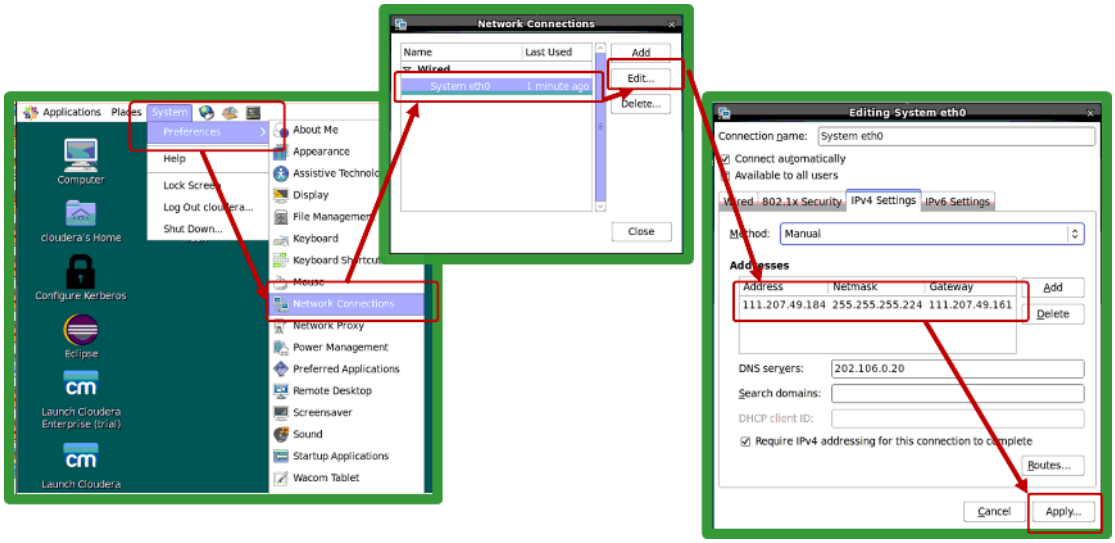
使用“半图形化”的工具设置，在终端中运行命令：sudo setup，进入设置

界面：





修改新节点的 IP 地址，使所有主机保持在同一个网段：

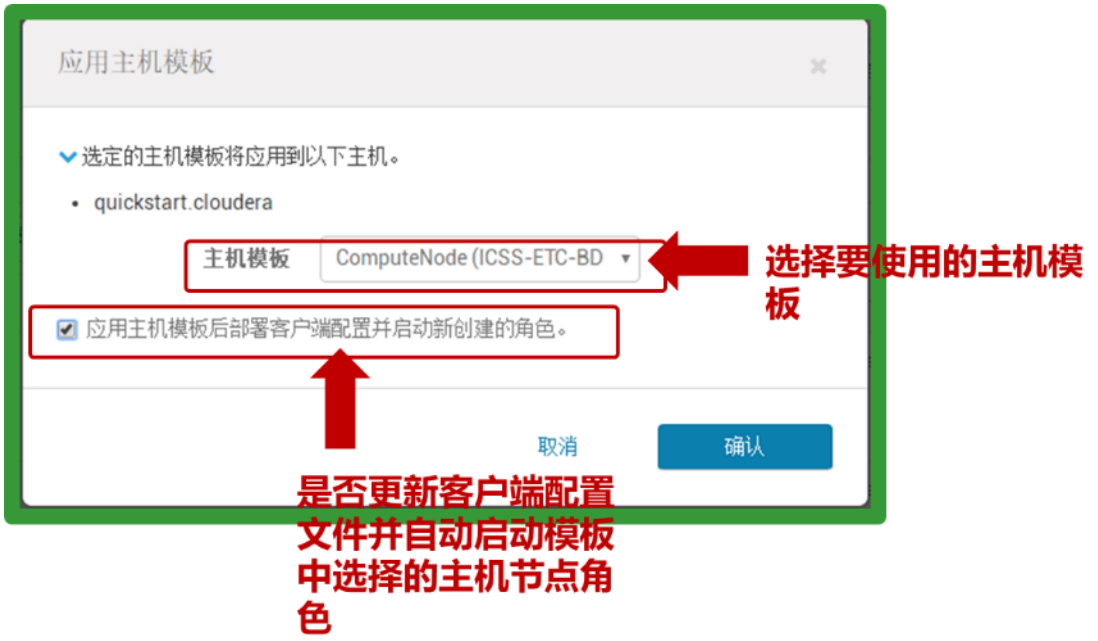


修改集群所有节点的 hosts 文件，保证正确配置集群中所有节点主机名和 IP 的映射关系。

在【主机】界面中点击【添加主机】连接，输入计算节点的 IP 地址搜索后根据提示将主机加入集群：

选中需要应用主机模板的新主机节点，并在【已选定的操作】菜单中选择【应用主机模板】菜单项：

根据向导提示应用主机模板：



Hadoop 包含许多的组件，不同的组件都包含各种各样的配置，并且分布于不同的主机之上。实验系统集成的管理工具针对这种情况提供了界面化的参数配置功能，并且能够自动的部署到每个组件节点。

事实上，应该始终使用图形界面化的参数配置功能来调整集群的参数属性，为了防止没有必要的误修改操作，和节点的 hosts 文件一样，集群管理工具选择了每次重启服务时利用脚本读取数据库中的参数属性取值动态生成 Hadoop 生态组件配置文件的方式来进行参数设置，在主机中能够找到的类似于 HDFS 的 core-site.xml 配置文件实质上是由监控管理工具生成的客户端配置信息，即在进行大数据项目开发时，代码 API 需要读取服务器的核心属性配置，则需要将这些实体配置文件拷贝到项目中作为客户端项目的配置依据。通过界面化的参数配置功能调整了集群的初始属性后，管理工具还提供了对应的功能菜单重新编译这些客户端配置文件。

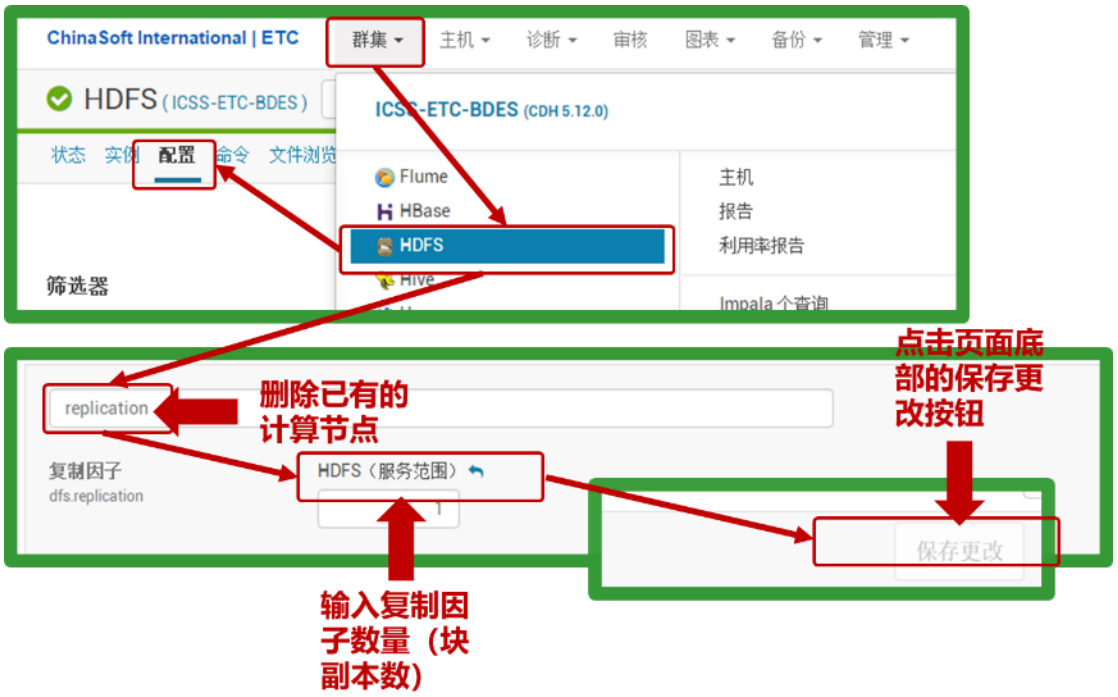
需要配置集群中各组件的属性，首先需要进入集群的组件管理子项，以 HDFS 为例，首先进入 HDFS 的管理界面，在主页的服务列表中选择 HDFS：

也可以点击集群菜单，选择 HDFS。

进入 HDFS 集群管理界面后，选择配置：

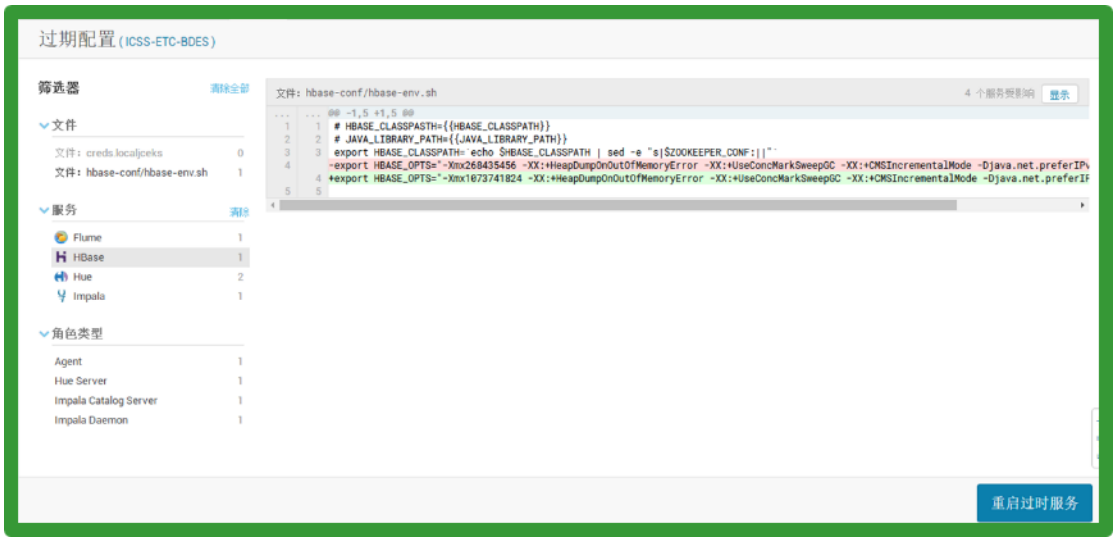
通过该界面即可修改对应的参数属性：

以修改 HDFS 块副本数为例：



修改集群组件节点的参数属性后，监控管理工具会给出提示，要求重新编译客户端配置信息，如修改了 HBase 集群节点对应的 Java 堆内存大小，保存后重启服务，在主界面将会显示：

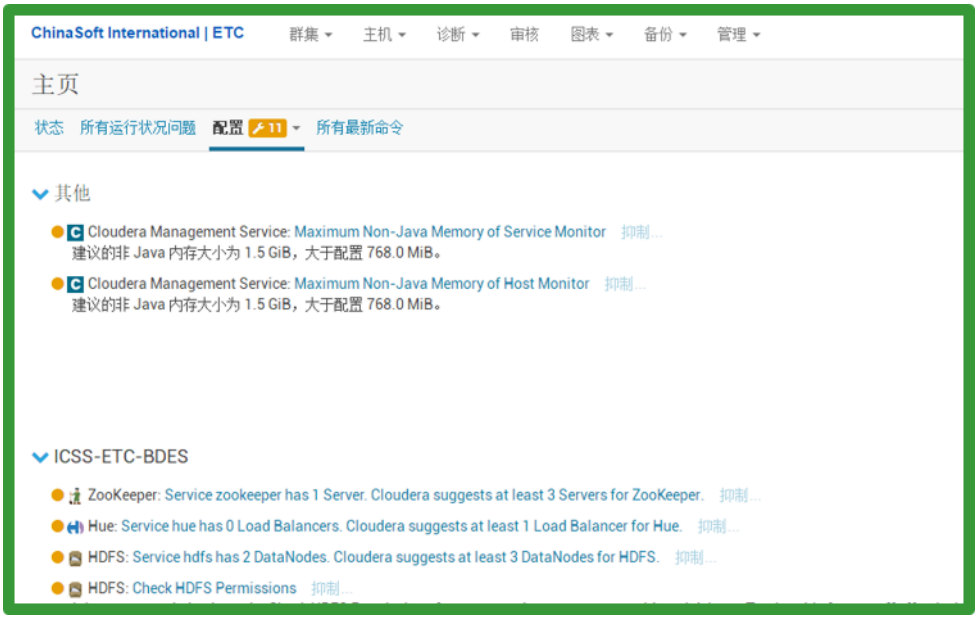
点击该提示图标，将能查看到有哪些配置信息和上次编译客户端配置后发生了改变：



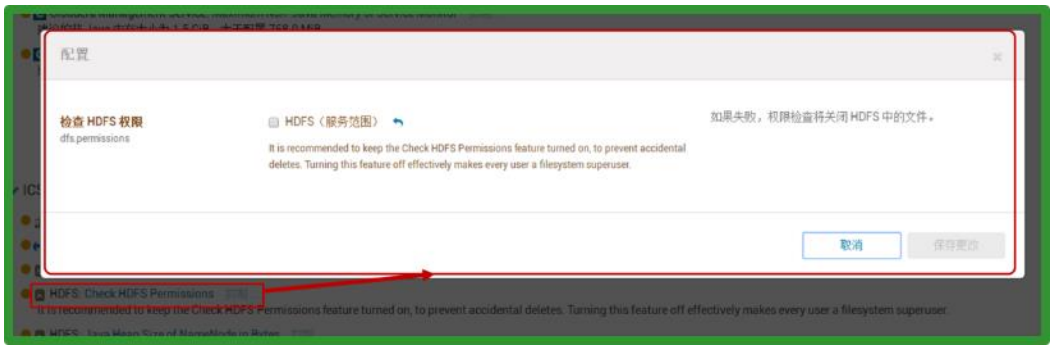
在对应组件的集群管理界面中，可以重新编译部署客户端配置：

当用户在配置过程中，部分参数配置值有问题时，管理工具会产生错误提示，以帮助用户更合理的修改参数，该提示信息在集群管理主页即能看到：

点击【配置问题】菜单项后，会给出详细的问题列表和建议说明：



点击问题说明后的链接即可快捷的对该属性参数进行调整：



集群管理工具对关键的组件使用 HA 部署方案，避免单点失效的发生，同时对于组件的异常错误提供了自动恢复的处理，最大限度的保证服务的可靠性。

在标准配置中，Namenode 是 HDFS 群集中的单点故障来源。每个群集都具有一个 Namenode，如果节点宕机或进程变为不可用，群集整体将变为不可用，直到 Namenode 在新主机上重新启动或上线。HDFS Secondary Namenode 不提供故障转移功能。实验系统集群管理工具支持基于 Quorum 的存储作为唯一的 HA 实施。为了让备用 Namenode 的状态与活动 Namenode 在实验环境中保持同步，两个节点均与一组名为 JournalNode 的独立后台程序进行通信。由活动 Namenode 执行任何元数据修改时，它会持续记录其中大部分 JournalNode 的修改记录。备用 Namenode 能够从 JournalNodes 读取编辑操作，并不断监视它们以了解编辑日志发生的更改。当备用节点发现编辑操作时，它会将这些编辑应用于自己的元数据。在发生故障转移时，备用节点将确保首先从 JournalNodes 读取所有编辑操作，然后才会将自己升级为活动状态。这确保了在发生故障转移之前完全同步 Namespace 状态。为了提供快速故障转移，备用 Namenode 还需要拥有有关群集中的块位置的最新信息。为实现这一目的， Datanode 配置了这两个 Namenode 的位置，它们会将块位置信息和检测信号作为心跳数据发送给这两个 Namenode。

一次只能有其中一个 Namenode 处于活动状态，这一点对于 HA 群集的正常运行来说至关重要。否则，元数据状态会在两者之间快速出现分歧，从而导致数据丢失风险或其他不正确的结果。为了确保此属性并防止元数据分裂，JournalNode 一次只允许一个 Namenode 成为写入程序。在故障转移过程中，要进入活动状态的 NameNode 将接管 JournalNode 的写入角色，这会有效地阻止其他 Namenode 继续保持活动状态，使得新的活动 Namenode 可以安全地继续执行故障转移。

Namenode 的高可靠可以通过监控管理工具的管理界面，根据向导提升启用 HA 配置，主要配置步骤如下 ：

在 HDFS 集群管理界面中在操作菜单中选择启动高可用性：

输入 nameservice 名字:

选择主备 NameNode, Journal Node:

选择对应主机后根据向导提示选择元数据的存放目录，接下来管理工具将自动启动相关服务完成高可用性的配置。

YARN ResourceManager (RM) 负责跟踪群集中的资源并安排应用程序（例如 MapReduce 作业）。RM 高可用性功能以活动/待机 RM 对形式添加冗余，以保证不

出现单点故障。此外，在从待机 RM 到活动 RM 进行故障转移时，应用程序可以从其上次检查点状态恢复 ；例如，在 MapReduce 作业中完成的 map 任务不在后续的尝试中重新运行。这样可以在不对运行中的应用程序产生任何重要性能影响的情况下，处理以下事件 ：

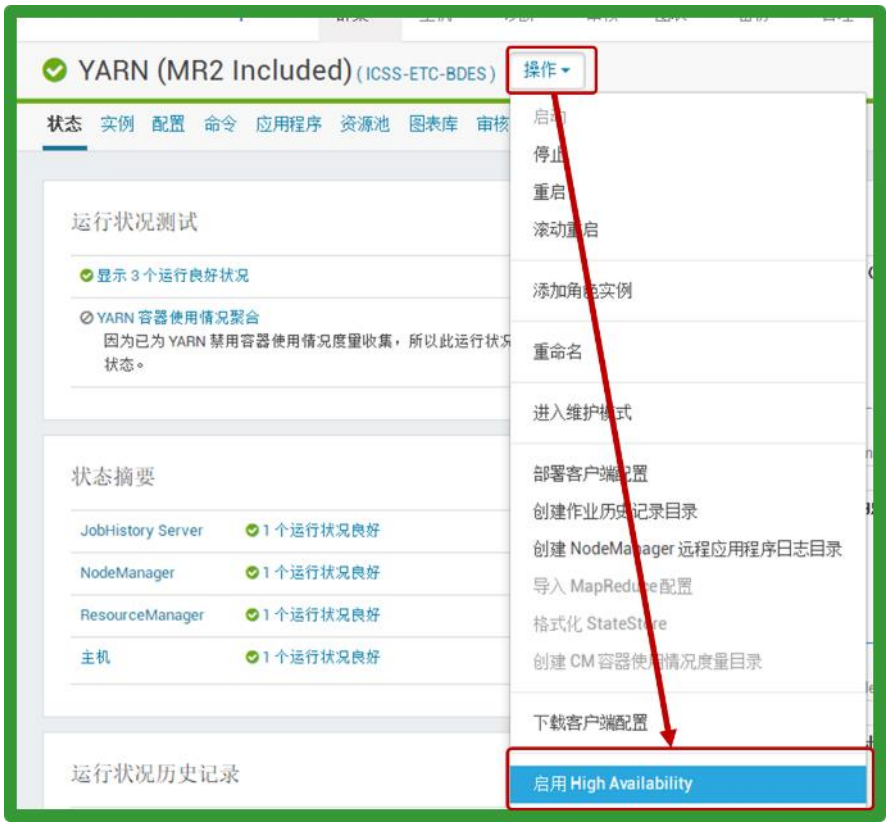
- 计划外事件，如节点宕机。
- 计划内维护事件，如在运行 ResourceManager 的计算机上进行的软件或硬件升级。

RM 高可用性要求 Zookeeper 和 HDFS 服务处于运行状态。其通过活动-待机 RM 对的方式实施。启动时，每个 RM 处于待机状态 ：启动过程，但未加载状态。转换到活动状态时，RM 会从指定的状态存储加载内部状态，并启动所有内部服务。

管理员（通过 CLI 接口）或通过集成的故障转移控制器（启用自动故障转移时）可显示转换为活动状态。

ResourceManager 的高可用性可以通过监控管理工具的的 Web 管理界面，根据向导提示启用 HA 配置，主要配置流程如下：

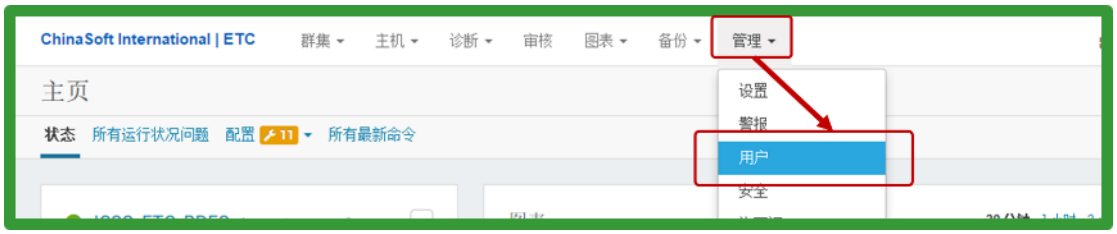
在 YARN 管理界面中在操作菜单中选择启动高可用性：



选择主备 Resource Manager 节点 ：

选择对应主机后管理工具将自动启动相关服务完成高可用性的配置。

对系统管理员、数据库管理员及其他管理员可以赋予不同级别的管理权限，用户权限管理的入口如下：



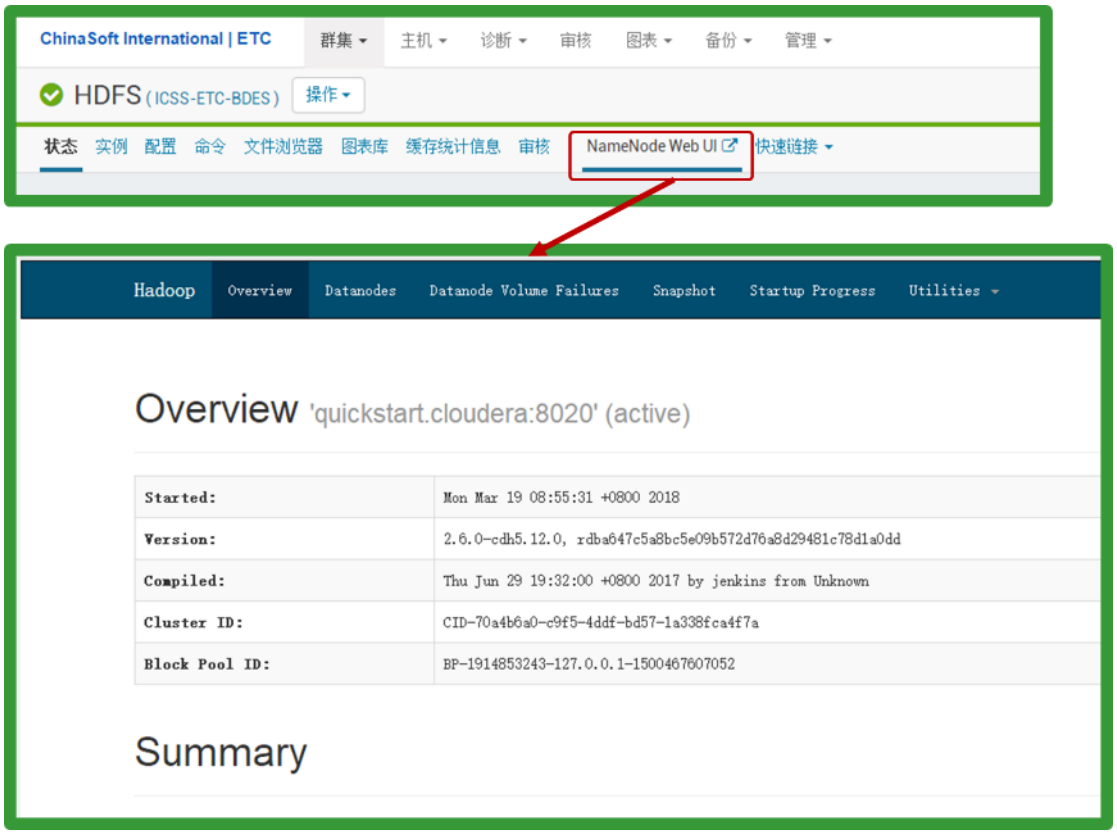
可以添加一个用户并直接赋予特定的身份权限：

也可以对已有用户进行编辑，首先选中用户，然后选择要进行的操作：

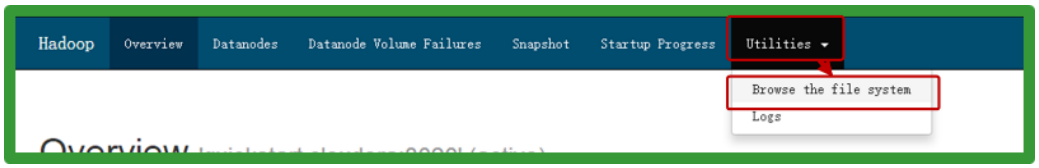
不同角色的用户登录后界面会有所差异，如下只读用户访问监控管理工具的界面，所有服务对应的启停等操作选项都不可用。

通过监控管理工具的 WEB 界面，还能直接查看 HDFS 集群上的文件结构并选择下载。首先进入 HDFS 集群管理界面：

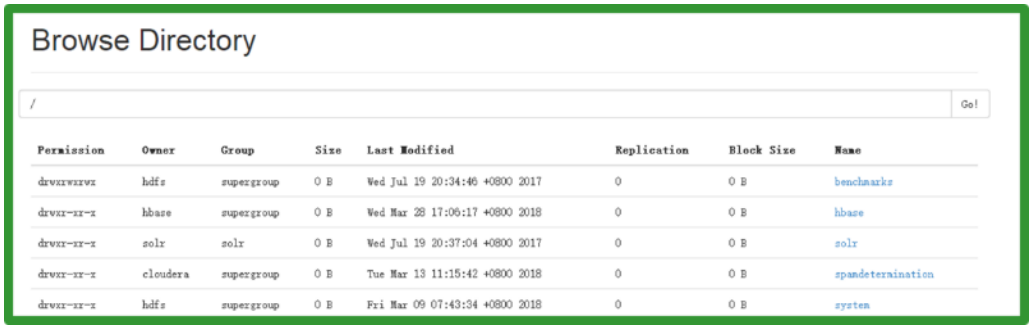
通过链接，进入 Hadoop HDFS 自带的 WEB 控制台界面：



在该界面中选择工具菜单，并选择浏览文件系统：



在浏览文件界面可以直接查看 HDFS 上的文件结构，通过点击文件链接，还能直接完成文件的下载操作：



系统监控管理工具提供了许多监控功能，用于监测群集（主机，服务守护进程）健康、组件性能以及集群中运行的作业的性能和资源需求。具体包括以下监控功能 ：

- **服务监控**

查看服务和角色实例级别健康检查的结果，并通过图表显示，有助于诊断问题。如果健康检查发现某个组件的状态需要特别关注甚至已经出现问题，系统会

对管理员应该采取的行动提出建议。同时，系统管理员还可以查看上的服务或角色上操作的执行历史，也可以查看配置更改的审计日志。

- **主机监控**

监控群集内所有主机的有关信息，包括 ：哪些主机上线或下线，主机上目前消耗的内存，主机上运行的角色实例分配，不同的机架上的主机分配等等。汇总视图中显示了所有主机群集，并且可以进一步查看单个主机丰富的细节，包括显示主机关键指标的直观图表。

- **行为监控**

监控管理工具提供了列表以及图表的方式来查看集群上进行的活动，不仅显示当前正在执行的任务行为，还可以通过仪表盘查看历史活动。同时提供了各个作业所使用资源的许多统计，系统管理员可以通过比较相似任务的不同性能数据以及比较查看同一任务中不同执行的性能数据来诊断性能问题或行为问题。

- **事件活动**

监控界面可以查看事件，并使它们用于报警和搜索，使得系统管理员可以深入了解发生集群范围内所有相关事件的历史记录。系统管理员可以通过时间范围、服务、主机、关键字等字段信息过滤事件。

- **报警**

通过配置监控管理工具可以对指定的事件产生警报。系通过管理员可以针对关键事件配置其报警阈值、启用或禁用报警等，并通过电子邮件或者通过 SNMP 的事件得到制定的警报通知。系统也可以暂时抑制报警事件，此限制可以基于个人角色、服务、主机、甚至整个集群配置，使得进行系统维护/故障排除时不会产生过多的警报流量。

- **审计事件**

监控管理工具记录了有关服务、角色和主机的生命周期的事件，如创建角色或服务、修改角色或服务配置、退役主机和运行集群管理服务命令等等。系统管理员可以通过管理员终端查看，界面提供了按时间范围、服务、主机、关键字等字段信息来过滤审计事件条目。

- **可视化的时间序列数据图表**

系统管理员可以通过搜索度量数据，系统将根据指定规则创建数据，组（方面）数据的图表，并把这些图表保存到用户自定义的仪表板。

- **日志**

介绍如何访问日志中的各种考虑到你正在查看当前上下文的方式。例如，监控服务时，你可以轻松地点击一个链接，查看相关的特定服务的日志条目，通过相同的用户界面。当查看关于用户的活动信息，您可以方便地查看了作业运行时所用的作业的主机上发生的相关日志条目。

- **报告**

监控管理工具可以将收集到的历史监控数据统计生成报表，比如按目录查看集群作业活动的用户、按组或作业 ID 查看有关用户的磁盘利用率，用户组的历史信息等。这些报告可以根据选定的时间段（每小时，每天，每周，等等）汇总数据，并可以导出为 XLS 或 CSV 文件。同时系统管理员还可以管理包括搜索和配额等 HDFS 目录设置。

如上图所示，监控管理工具会实时的对集群的各项服务进行监控，一旦服务出现了异常，将会产生告警。同时，对集群的各项性能数据进行监控，例如 CPU，网络等，一方面能够监控集群的性能状态，一方面能够帮助性能调优。

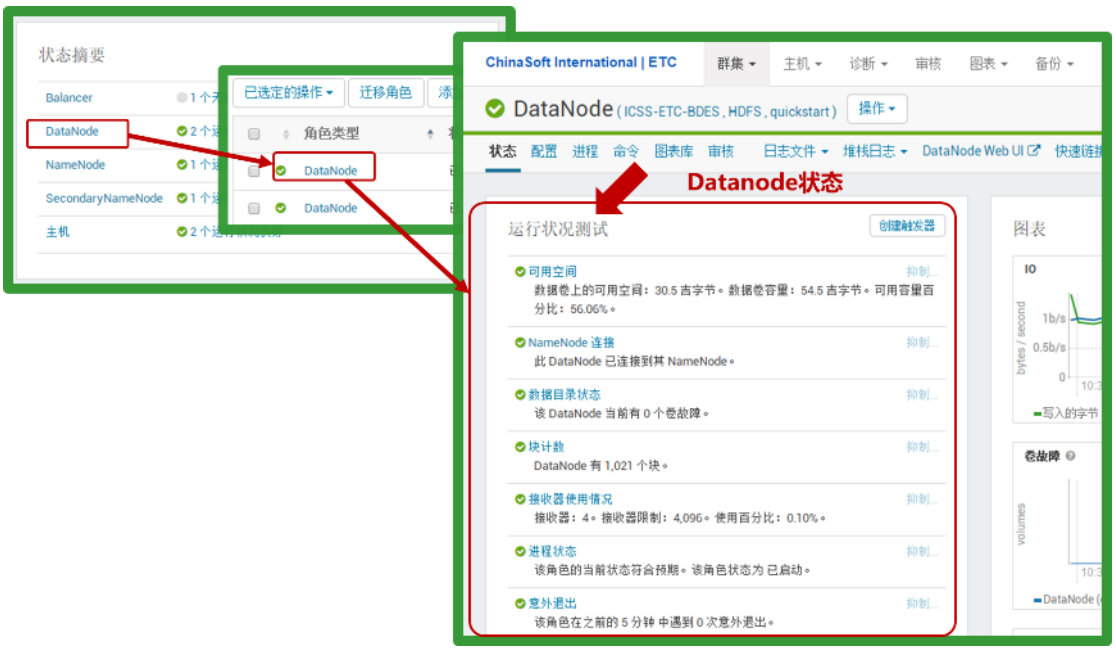
监控管理共几乎能监控 Hadoop 集群中部署的所有生态组件服务，以下以 HDFS 以及 Yarn 为例进行演示说明。

以 HDFS 服务为例， HDFS 服务页面的状态页面中显示了服务状态的汇总信息，包括 HDFS 综述， HDFS 文件系统块总数、总大小，文件总数、剩余量、损坏块，需复制快、 JVM 线程状态等。页面中还包括了各种快捷链接。

服务页面中的“运行状况测试”面板显示了所有的服务进程状态，如下 ：

服务页面中还显示列损坏块、容量、正常服务节点数目等关键的信息，如下图:（由于目前 HDFS 中不存在损坏快，如果存在的话，会在下图中的页面显示。）

点击进入 Datanode 节点详细信息，可查看每个 DataNode 的关键信息，包括服务状态、容量、已使用空间、 Block 数目 版本等：



在 YARN 服务页面也能查看所有 YARN 节点的状态，如下 ：

点击相应的 NodeManager 可以查看对应节点的详细状态和信息，如下 ：

其他生态组件的服务监控参考 HDFS 和 YARN。

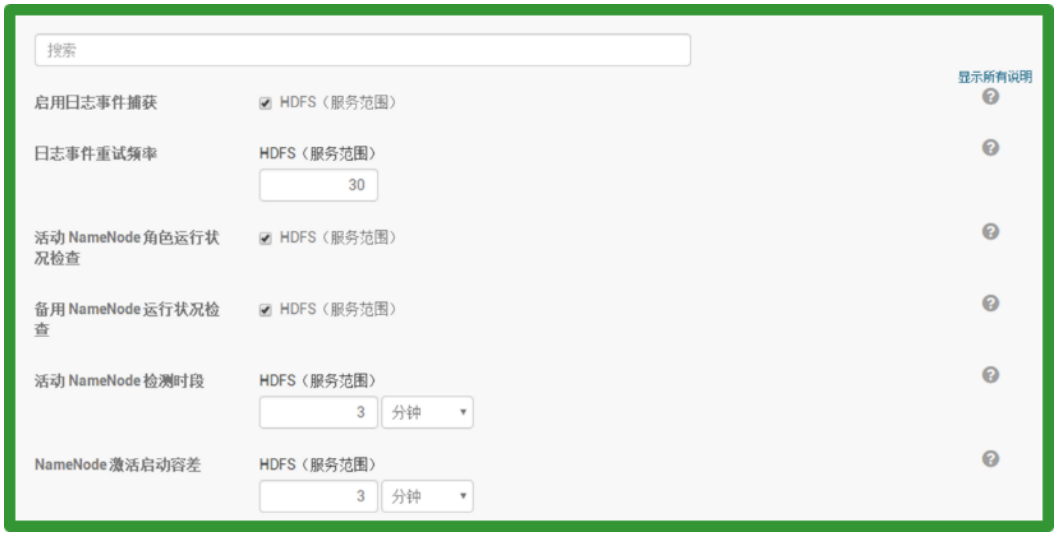
监控管理工具能够支持节点级别的详细信息查看，包括该节点的硬件信息 ：网络，硬盘等。软件信息 ：运行的各项服务的状态，并对这些状态进行实时监控。

点击主机菜单项，进入主机列表，点击特定的主机名后展示该节点对应的监控信息：

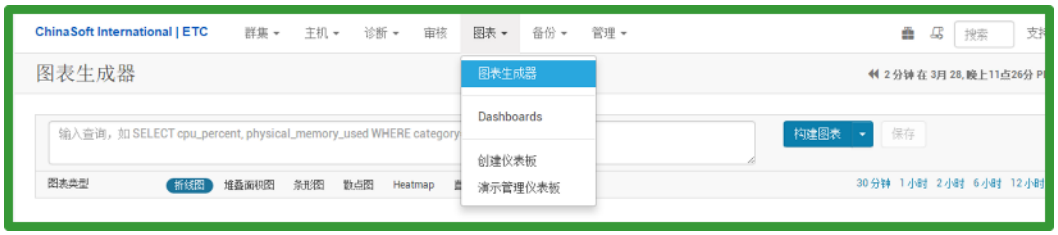
监控管理工具对于每个组件，都提供了监控的参数配置，例如可以设置达到多少阈值的时候，产生 warning 告警，达到多少阈值的时候产生 Critical 告警。

以 HDFS 为例，进入 HDFS 集群管理界面的【配置】界面，并在配置类别中选择【监控】:

此时可以在主界面对监控的参数进行自定义调整：



除了默认提供的性能图表以外，监控管理工具还允许用户自定义图表，使得用户能够查看自己希望的性能图表。 监控管理工具提供了一套类似的 SQL 的语言，用户只需要使用这个语言，简单的编写类似 SQL 的语句，就能够生成自己想要的图表。



监控管理工具主页能够允许用户自定义主界面的风格，可以通过简单的拖拽就能够调整主界面上组件的位置，也能够自定义添加需要的组件。

监控管理工具提供多达 102 类监控指标（见下表），覆盖所有的服务及功能，

包括集群硬件使用情况（网络、CPU、内存以及硬盘等）、服务状态等，同时指标按集群级别、主机级别、用户级别以及表 / 目录级别等等分级统计，总指标项上万，比如：集群指标超过 3000 个、HBase 系统级指标超过 1000 个、HDFS 系统级指标超过 300 个。

部分重要状态指标如下：

HDFS 指标：

KPI 名称	KPI 描述	单位	级别
CPU 占用率	CPU 平均占用率	%	系统级 / 节点级
内存占用率	内存平均占用率	%	系统级 / 节点级
系统空间	总空间	MB	系统级 / 节点级
已用空间	已用空间	MB	系统级 / 节点级
可用空间	剩余空间	MB	系统级 / 节点级
空间使用率	已用空间与系统空间的比值	%	系统级 / 节点级
读流量	统计周期内读流量统计	MB	系统级 / 节点级
写流量	统计周期内写流量统计	MB	系统级 / 节点级
读 IOPS	每秒进行读(I/O)操作的次数	个 /s	系统级 / 节点级
写 IOPS	每秒进行写(I/O)操作的次数	个 /s	系统级 / 节点级

MapReduce 指标：

KPI 名称	KPI 描述	单位	级别
提交作业数	提交作业数	个	系统级
完成作业数	完成作业数	个	系统级
失败作业数	失败作业数	个	系统级
正在运行的作业数	正在运行的作业数	个	系统级
Map 总任务数	Map 总任务数	个	系统级
Reduce 总任务数	Reduce 总任务数	个	系统级
Map 任务完成数	Map 任务完成数	个	系统级
Reduce 任务完成数	Reduce 任务完成数	个	系统级
正在执行的 Map 任务数	正在执行的 Map 任务数	个	系统级
正在执行的 Reduce 任务数	正在执行的 Reduce 任务数	个	系统级
平均 Map 任务执行时间	平均 Map 任务执行时间	秒	系统级
平均 Reduce 任务执行时间	平均 Reduce 任务执行时间	秒	系统级
最小 Map 任务执行时间	最小 Map 任务执行时间	秒	系统级
最小 Reduce 任务执行时间	最小 Reduce 任务执行时间	秒	系统级
最大 Map 任务执行时间	最大 Map 任务执行时间	秒	系统级
最大 Reduce 任务执行时间	最大 Reduce 任务执行时间	秒	系统级
Map 任务执行失败数	Map 任务执行失败数	个	系统级
Reduce 任务执行失败数	Reduce 任务执行失败数	个	系统级

Map 任务执行成功数	Map 任务执行成功数	个	系统级
Reduce 任务执行成功数	Reduce 任务执行成功数	个	系统级

HBase 指标：

KPI 名称	KPI 描述	单位	级别
系统空间	系统总空间	MB	系统级
已用空间	NoSQL 系统已用数据量（系统级）；RegionServer 的 Region 个数（节点级）	MB	系统级 / 节点级
可用空间	NoSQL 系统可用数据量	MB	系统级
空间使用比率	已使用的空间与总空间的比值	%	系统级
CPU 平均使用率	NoSQL 系统平均 CPU 使用率	%	系统级 / 节点级
CPU IOwait 使用率	NoSQL 系统平均 CPU IOwait 使用率	%	系统级 / 节点级
压缩合并队列长度	数据分区服务器中的压缩合并（Compaction）队列大小，当存储文件数量达到该数值时将启动压缩合并。	个	系统级
请求时延 10 毫秒次数	请求响应时间小于或等于 10 毫秒的请求次数	次	系统级
请求时延 2000 毫秒次数	请求响应时间小于或等于 2000 毫秒的请求次数	次	系统级
请求时延 2000 毫秒以上次数	请求响应时间大于 2000 毫秒的请求次数	次	系统级
读 IO 次数	读 IO 次数	次	节点级
写 IO 次数	写 IO 次数	次	节点级
IO 次数	节点 IO 次数（读写 IO 次数总和）	次	节点级

部分重要报警指标如下：

系统级 Hive 指标：

告警名称	告警描述	定位信息
空间使用比率超阈值告警	存储资源空间占用比超过门限	系统标识
	存储系统全部节点 CPU 平均使用率超过门限	系统标识
	存储系统全部节点内存平均使用率超过门限	系统标识
系统状态异常	存储系统不可用、只读告警	系统标识

节点级 Hive 指标：

告警名称	告警描述	定位信息

节点状态异常	节点故障	节点 IP/ 主机名
空间使用比率超阈值告警	存储资源空间占用比超过门限	节点 IP/ 主机名
CPU 使用率超阈值告警	存储系统单节点 CPU 平均使用率超过门限	节点 IP/ 主机名
内存使用率超阈值告警	存储系统单节点内存平均使用率超过门限	节点 IP/ 主机名
进程异常	系统重要进程异常	节点 IP/ 主机名、进程标识
元数据故障	元数据写入异常或丢失	节点 IP/ 主机名、表标识、索引标识、元数据条目

系统级 HDFS 指标：

告警名称	告警描述	定位信息
空间使用比率超阈值告警	存储资源空间占用比超过门限	系统标识
CPU 使用率超阈值告警	存储系统全部节点 CPU 平均使用率超过门限	系统标识
内存使用率超阈值告警	存储系统全部节点内存平均使用率超过门限	系统标识
系统状态异常	存储系统不可用、只读告警	系统标识

节点级 HDFS 指标：

告警名称	告警描述	定位信息
节点状态异常	节点故障	节点 IP/ 主机名
空间使用比率超阈值告警	存储资源空间占用比超过门限	节点 IP/ 主机名
CPU 使用率超阈值告警	存储系统单节点 CPU 平均使用率超过门限	节点 IP/ 主机名
内存使用率超阈值告警	存储系统单节点内存平均使用率超过门限	节点 IP/ 主机名

系统级 MapReduce 指标：

告警名称	告警描述	定位信息
CPU 使用率超阈值告警	存储系统全部节点 CPU 平均使用率超过门限	系统标识
内存使用率超阈值告警	存储系统全部节点内存平均使用率超过门限	系统标识
系统状态异常	系统不可用	系统标识

节点级 MapReduce 指标：

告警名称	告警描述	定位信息
节点状态异常	节点故障	节点 IP/ 主机名
空间使用比率超阈值告警	存储资源空间占用比超过门限	节点 IP/ 主机名
CPU 使用率超阈值告警	存储系统单节点 CPU 平均使用率超过门限	节点 IP/ 主机名
内存使用率超阈值告警	存储系统单节点内存平均使用率超过门限	节点 IP/ 主机名
任务失败	Task 执行失败	节点 IP/ 主机名、用户标识、作业标识、任务标识

系统级 HBase 指标：

告警名称	告警描述	定位信息
空间使用比率超阈值告警	存储资源空间占用比超过门限	系统标识
CPU 使用率超阈值告警	存储系统全部节点 CPU 平均使用率超过门限	系统标识
内存使用率超阈值告警	存储系统全部节点内存平均使用率超过门限	系统标识
系统状态异常	存储系统不可用、只读告警	系统标识

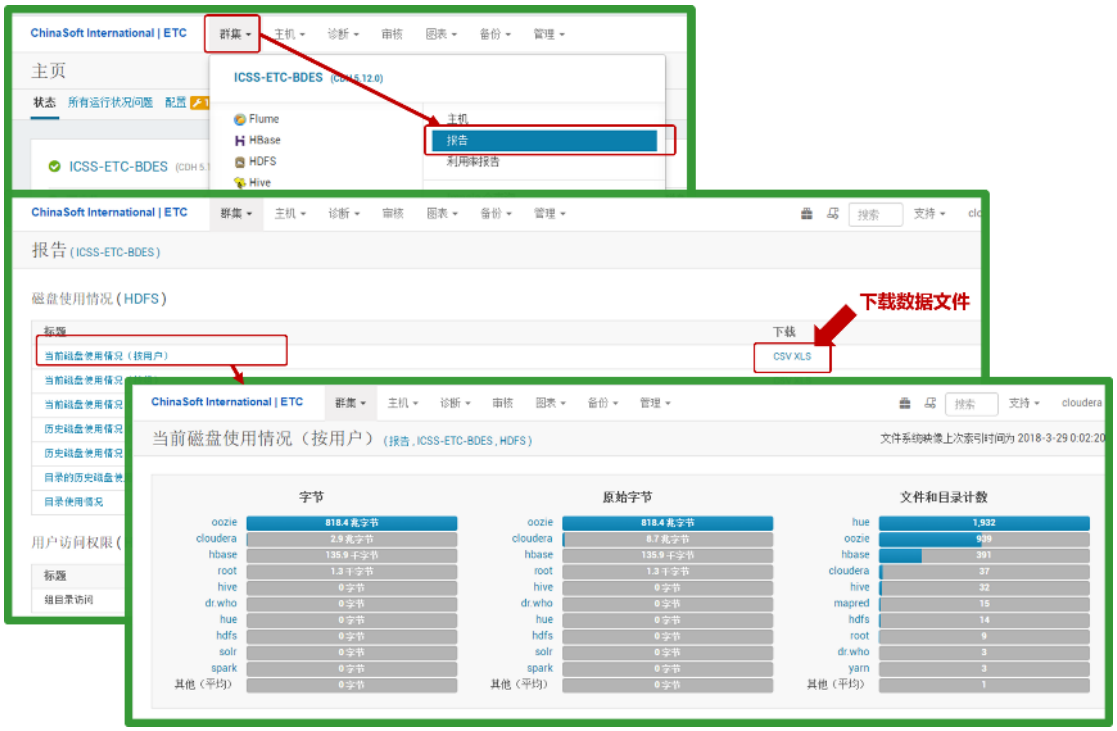
节点级 HBase 指标：

告警名称	告警描述	定位信息
节点状态异常	节点故障	节点 IP/ 主机名
	存储资源空间占用比超过门限	节点 IP/ 主机名
CPU 使用率超阈值告警	存储系统单节点 CPU 平均使用率超过门限	节点 IP/ 主机名
内存使用率超阈值告警	存储系统单节点内存平均使用率超过门限	节点 IP/ 主机名

诊断功能主要是指能够帮助用户根据收集的数据查看潜在的或者存在的风险与问题。

监控管理工具会对集群中运行的服务进行周期性的运行状况测试。以检测这些服务的状态是否正常。如果有异常情况会进行告警，有利于更早的让客户感知集群服务存在的问题。

监控管理工具在 Hadoop 集群运行期间，会记录集群中发生的事件，例如集群服务的启停，用户的操作，性能数据等等。 系统会将这些数据都记录到一个文件中。 系统专门提供了一个集群，能够对这些数据进行分析，查找集群中的潜在风险，以及问题。通过监控管理工具可以下载这些统计数据文件：



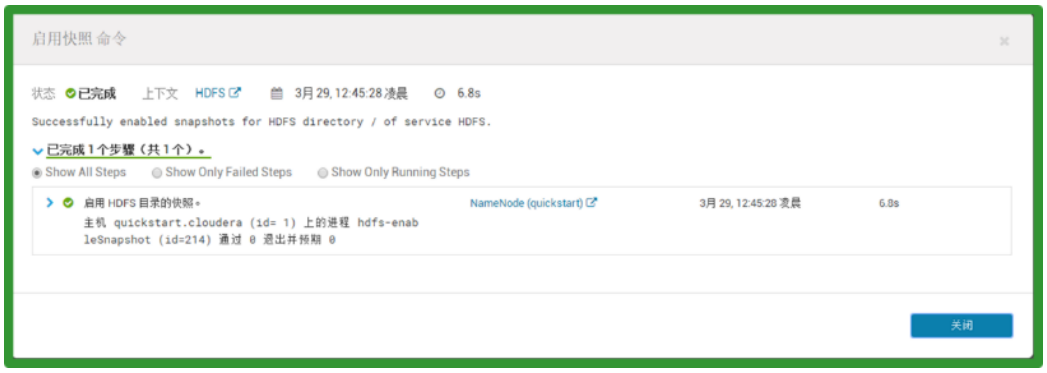
当集群中出现异常状态时，监控管理工具会进行操作提示，例如 hive server 异常退出， 系统会 提示“该角色的进程已退出，预期状态为已启动”。并且，系统会进行操作提示，以及建议，包括针对这种情况，可以采取的操作，如何定位问题的原因等等。

Hadoop 集群的性能调试尤为重要，因此能够方便精确的监控集群额性能尤为重要。 监控管理工具能够生产系统性能使用报告，包括集群的 CPU 使用率等各项性能数据：

如果要完成 HDFS 文件备份，则进入 HDFS 集群管理界面，选择文件浏览器：

在右侧的目录操作栏中选择【启动快照】：

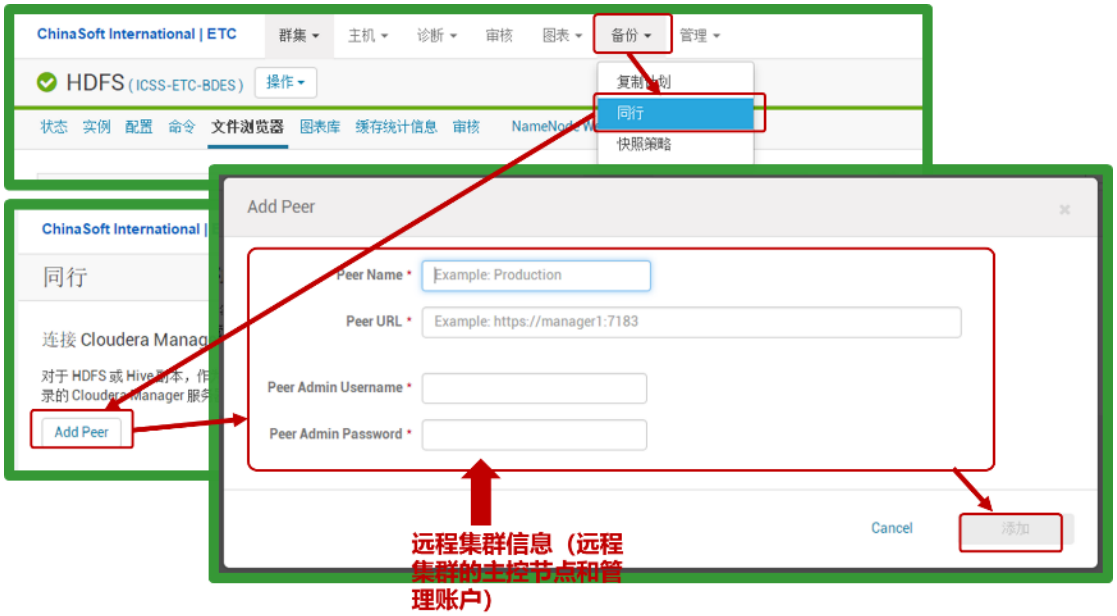
系统会自动为该目录启动快照功能：



需要备份数据时为目录拍摄快照：

需要恢复数据时通过快照恢复即可：

另一种方式是基于对等集群方式备份，首先需要构建一个新的集群用于同步数据，然后再需要备份的集群中设置【同行】，在监控管理工具的主界面中点击【备份】->【同行】，输入远程集群的信息：



设置备份计划：

恢复数据和备份数据的操作相同，只是需要将源集群和目标集群对调，在制定备份计划时，设置为单次执行。

接下来介绍图形化交互界面 Hue 的基本使用。

Hue 是一个可快速开发和调试 Hadoop 生态系统各种应用的一个基于浏览器的图形化用户接口。Hue 的主要功能包括：

- 访问 HDFS 和文件浏览
- 通过 web 调试和开发 hive 以及数据结果展示
- 查询 solr 和结果展示，报表生成
- 通过 web 调试和开发 impala 交互式 SQL Query
- spark 调试和开发
- Pig 开发和调试
- oozie 任务的开发，监控，和工作流协调调度
- Hbase 数据查询和修改，数据展示
- Hive 的元数据（metastore）查询
- MapReduce 任务进度查看，日志追踪

- 创建和提交 MapReduce，Streaming，Java job 任务
- Sqoop2 的开发和调试
- Zookeeper 的浏览和编辑

通过管理工具的 Hue 组件管理界面可以链接到 Hue 交互工具，和平台监控管理工具类似，Hue 也通过 Web 界面进行操作。要使用 Hue，访问如下 URL：

http://主控节点 IP/域名:8888：

在 Hue 中同样可以完成 HDFS 文件的管理，和默认的 HDFS web 管理界面相比，Hue 还能够直接浏览文件内容，并提供了修改文件内容的接口。

进入 Hue 主界面后点击 HDFS 按钮：

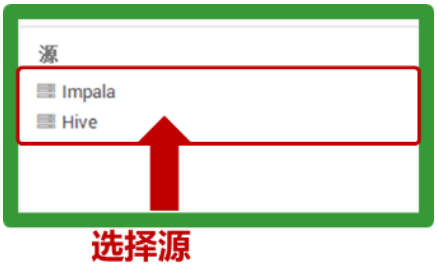
此时可以在下方浏览 HDFS 的文件结构：

通过目录树依次点击，当点击的节点为一个文件是，会自动进行文件内容预览，在文件内容预览界面中还包含了文件的属性摘要以及下载、修改文件对应的超级链接：

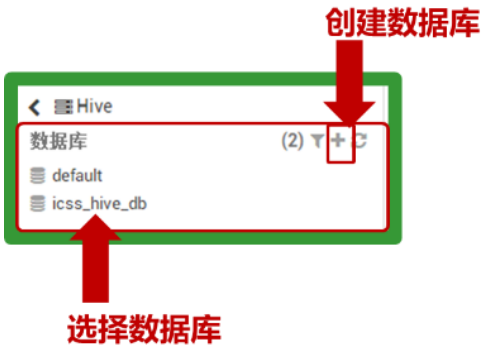
点击编辑文件即可对文件进行在线编辑：

在 Hue 主页面中点击【表】菜单项可以浏览集群中搭建的数据仓库中的表：

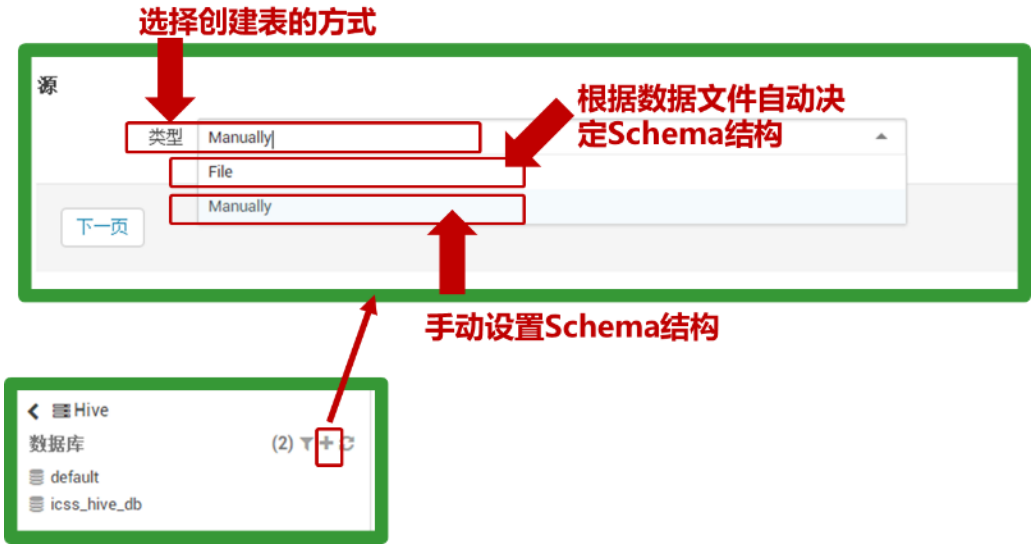
在缺省情况下，能够直接对集群中的 Hive 和 Impala 进行操作，进入表浏览后，选择特定的源：



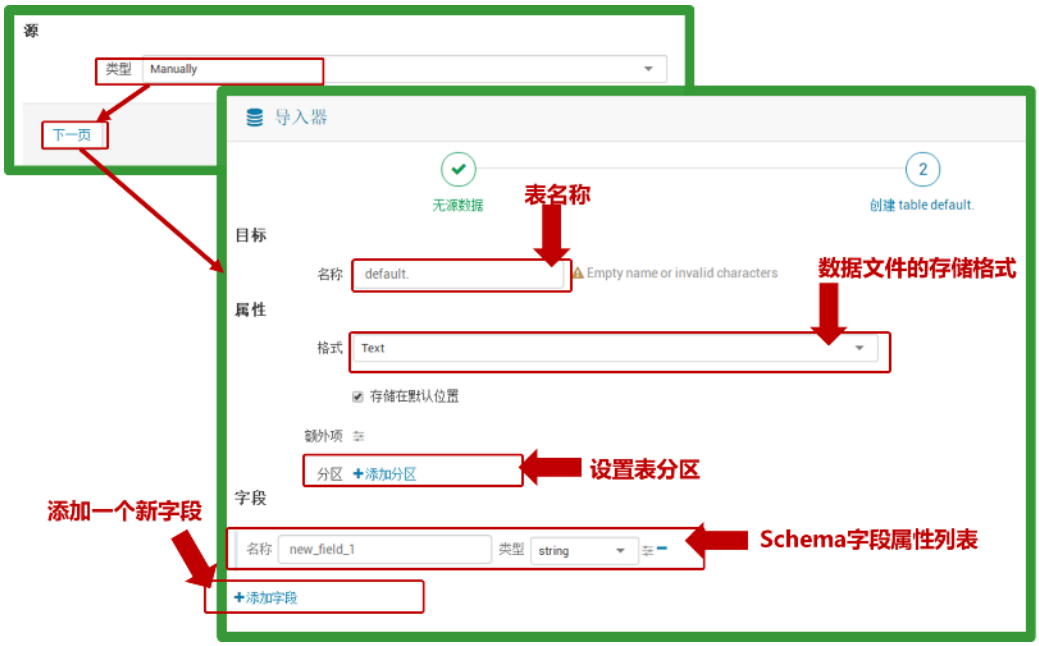
进入特定源后，可以对数据库进行操作：



进入一个特定数据库后，可以利用 WEB 界面直接构建表结构，创建表可以通过数据文件自动生成或是输入 Schema 结构参数手动创建：



手动设置 Schema 的界面设置基本覆盖了构建表所需要的所有参数，包括表的分区、存储类型以及字段属性等：



创建表后可以执行导入数据等操作：

在 Hue 中可以构建针对集群中各类查询引擎的查询分析器，针对不同组件进行查询的方法类似，都是在【查询】菜单中构建对应的查询分析器，以 Hive 查询为例，可以通过如下方式构建查询分析器：

查询分析器界面及功能布局如下：

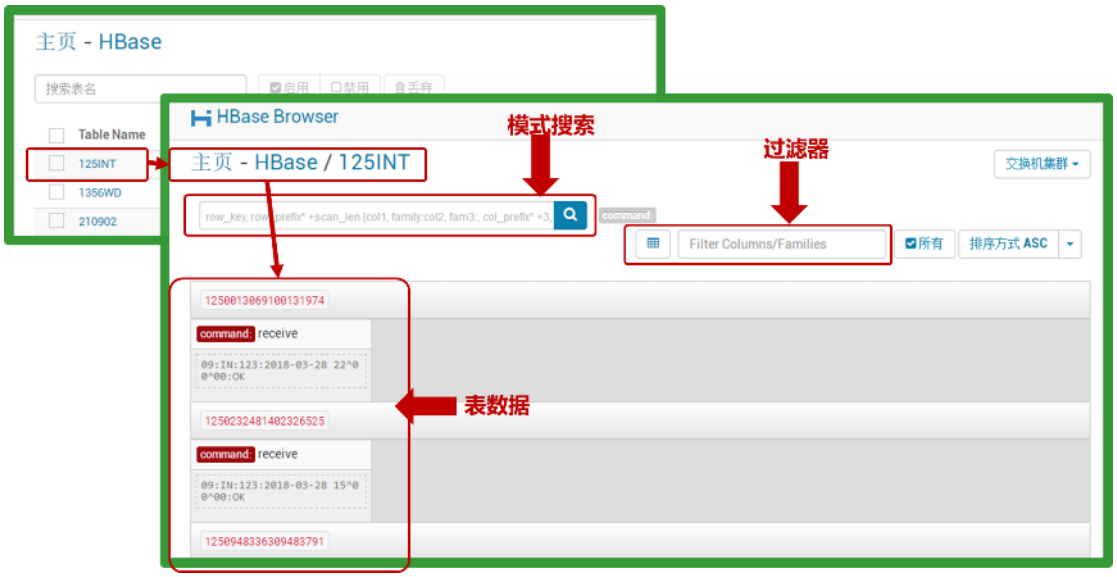
和传统 RDBMS 提供的查询分析工具一样，在编写查询语句时，Hue 也会提供丰富的语法补全提示功能。

通过 Hue 也能方便的对 HBase 的表进行管理和查询，首先打开 HBase 表浏览器：

在该界面中可以对 HBase 表进行操作，如新建表：

点击【新建表】按钮后通过 WEB 界面确定表名和 Schema 结构（列族列表）的方式构建新表：

在现有表列表中点击表名列表，即可查看数据：



在进行搜索时，会自动给出对应的模式提示。

在 Hue 中可以直接执行以 Jar 形式存在的 MapReduce 任务，通过以下步骤进入到 MapReduce 任务调度界面：

MapReduce 的任务调度比较简单，主要需要确认 MapReduce 任务所在的路径（HDFS 上的路径）：

如果担心路径编写错误，还可以直接点击浏览按钮浏览 HDFS 上的文件并选择：