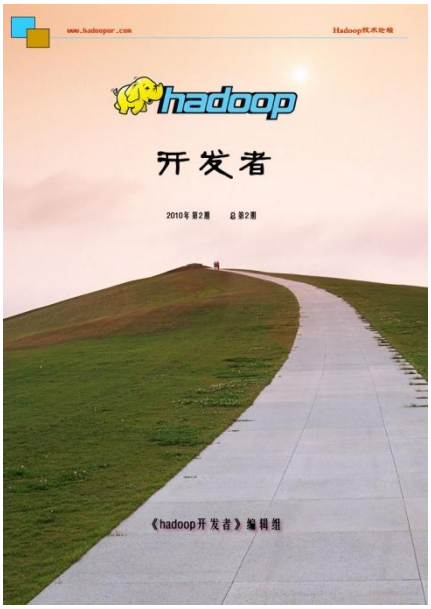
开发者

2010年 第2期 总 第2期

《hadoop开发者》编辑组

分享 自由 开放



《Hadoop 开发者》第二期

2010 年 3 月 30 日发布

欢迎投稿

出品

Hadoop 技术论坛

总编辑

易剑(一见)

副总编辑

Barry(beyi)　　代志远(国宝)

本期执行主编

Barry(beyi)

编辑

皮冰锋(若冰)　　易剑(一见)
贺湘辉(小米)　　Barry(beyi)
代志远(国宝)　　柏传杰(飞鸿雪泥)
何忠育(Spork)　　秘中凯
陈炬

排版/美工/封面设计

Barry(beyi)

网址

http://www.hadoopor.com

投稿信箱

hadoopor@foxmail.com

刊首语

《Hadoop 开发者》的又一期与大家见面了。

万事开头难，在主编一见成功地推出创刊号后，短短的几天内下载量过千，发布在我个人博客的《hadoop 开发者》就有上百人下载，可见，Hadoop 如此受到大家的钟爱，也给予了我们继续下去的动力。Hadoop 开发者的第二期继续延续着分享、自由、开放这一开源社区的精神传统，分享给大家 Hadoop 学习和应用的心得与体会。

Hadoop 应用一直是大家关注而又热衷的话题，这一期里原本打算推出 Hadoop 与搜索引擎这一主题，但遗憾的是收到的相关稿件较少，难以成刊，只好作罢。从稿件质量看来，《Hadoop 开发者》需要一些更高水平的稿件，而不能仅仅局限于 Getting Started，更需要 Deeply Involved。办好《Hadoop 开发者》，任重而道远，需要 Hadoop 的爱好者的广泛参与，我们在期待大师级文章的出现。

《Hadoop 开发者》第二期成刊过程中，我认识了国内某知名猎头公司的“人才猎手”Syvia，得知到业界的很多著名 IT 公司近几年都在物色 Hadoop 相关的优秀技术人才，Hadoop 正在或已经引起业界的广泛关注。有理由可以期待，Hadoop 的未来和应用前景光明。

《Hadoop 开发者》编辑组

本期执行主编：Barry

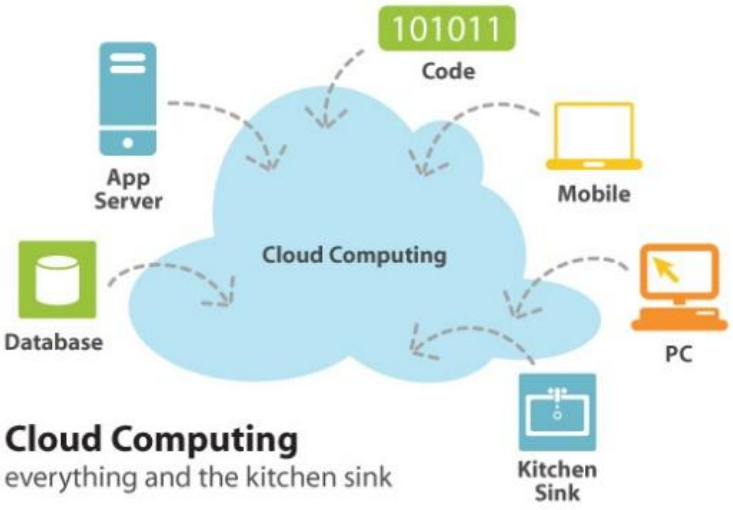
2010-3

目录

Hadoop 业界资讯

1.InfoWorld 授予 Apache Hadoop 年度技术创新奖章

今年 1 月，InfoWorld 授予 Apache Hadoop 年度技术创新奖章，获奖理由就是 Apache Hadoop 公司创造了使用商用硬件上数千兆数据来运行大规模分析计算功能的可能性。有了 Hadoop 和开源 NoSQL 数据库来大幅度减少数据处理时间， 我们就有机会带来游戏行业的变革，因为游戏程序的改动需要进行高度专业的分析。另外，昂贵的硬件和软件资源也可以供更多的专业人才使用。



有可能很多数据的处理过程都将在云上完成，云服务将提供给那些不需要全天候运行大规模计算处理的企业用户。事实上，云本身会有两种方式和开源对接。其一，多用户租赁开源软件成为缺省的软件即服务产品，其二，由谷歌，亚马逊等提供商提供的开放式应用编程接口也会被很多用户用在开源代码的研发上。虽然这还有一段 适应的过程，但行业中的很多用户看似都能接受这种重新定义。

2.网友观点：SQL 和关系型数据库---它们并不适合云计算

我参加了在加利福尼亚州圣克拉拉市举行的 2010 年 Cloud Connect 大会，这是今年最早举行的云计算重大会议之一。到目前为止，会议一个较大的议题是“不使用关系型数据库来保持数据的持久性”。这被称为 “NoSQL”运动，其宗旨是使用其他形式的数据库，更有效地处理大规模的数据。而关于围绕云计算出现的“大规模数据”，我已经写过一些文章，但是这一运动更为重要，它将推动数据回归到以更简单、但却可能更有效的模型进行物理存储的方式。

NoSQL 系统在运行时一般会把数据存放在内存中，或者是并行地从许多磁盘上读取数据。

其中就有一个问题，“传统”的关系数据库不提供这种模式，因此也没法提供同样的性能。在过去那种数据库中，如果只有几个 GB 数据，这一问题还不是很明显，但是许多云计算的数据库已经超过了 1TB，还会有更多的大规模数据库会被用来支撑不断发展的云计算系统。在关系型数据库上对大规模数据进行操作是兵家大忌，因为在处理数据时 SQL 请求会占用大量的 CPU 周期，并且会导致大量的磁盘读写。

如果你觉得以前好像在哪里听过这种说法，那么我告诉你其实你是对的。早在上世纪 90 年代，对象数据库和 XML 数据库就取得过一些进展，尽管那时许多非关系型数据库确实能提供更好的性能，但很多企业却守住了关系型数据库的江山，如 Oracle、Sybase 和 Informix。然而，由于从关系型数据库上迁移出去的花费和风险太高，而且数据的规模也相对较小，使得关系型数据库几乎一统天下。

不过，云计算改变了一切。在云计算中需要对大量的数据进行处理，这一需求导致新的数据库处理方法运用在了旧模型上。MapReduce 是 Hadoop 处理数据的基本方法，它是基于几年前的“无共享”(share-nothing)数据库处理模型，但现在我们有了实现它的处理能力、磁盘空间以及带宽。

我估计云计算的发展将会减少对关系型数据库的使用。这并非新鲜事物，但这回我们却实实在在需要改变了。

3. Twitter：用 Cassandra 取代 MySQL？

甲骨文收购 Sun 之后，MySQL 的发展前景一直受到各方的密切关注。最近，一些 MySQL 长期用户向其他系统迁移的做法，为 MySQL 的未来增加了悲观的预期。

前段时间，Twitter 宣布，将淘汰既有的 MySQL 系统，改用 Cassandra 管理信息。Cassandra 是一个由 Apache 基金资助的分布式开源数据库，主要用于将海量数据分布到大量廉价服务器，进而拼凑出一个无单点故障的信息管理集群。而在 Twitter 之前，Facebook、Digg 已经开始使

用 Cassandra，思科的 WebEx 也已使用 Cassandra 来收集用户反馈。

曾几何时，MySQL 作为互联网的宠儿和开源软件旗手，备受各类互联网应用的青睐。但是，随着 Sun 收购 MySQL，它曾经耀眼的光芒慢慢褪去。对许多用户而言，MySQL 已成为商业性盈利产品，其未来发展存在很大不确定性。另外，随着 Web 2.0 应用的不断扩展，很多企业发现，使用 MySQL 的成本将伴随数据量的膨胀呈指数级增长，集中式数据存储越来越难于达到效率与效益的有机平衡。

除此之外，近期兴起的 No-SQL 运动也给 IT 行业带来了新的选择。除 Cassandra 外，No- SQL 运动的代表还包括 Hadoop、Google 的 Big Table、MemCacheDB、Voldemort、CouchDB 和 MongoDB。在这样的背景下，不仅大型互联网企业开始放弃包括 MySQL 在内的关系型数据库，即便是一些企业的内部应用，考虑到多媒体、电子邮件、空间和地理信息的增多，也开始采用非关系型数据库方案。

Twitter 官方对 Cassandra 的一些评论似乎更能说明问题。Twitter 称：Cassandra 不存在单点故障；出身于 Facebook，天生为海量数据设计；适用于大量分布式写操作；依托于一个健康的支持社区。”对于那些建立在关系型数据库之上的系统而言，Cassandra 还提供从其他关系型数据库加载数据的手段，这意味着那些潜在用户可以考虑尝试将其系统用于 Cassandra，而 Twitter 也正在这么做的。Twitter 计划让两套系统先并行一段时间，待确定新系统稳定运行后再将 MySQL 淘汰掉。

今天的 MySQL 不得不面临许多问题，它在大型应用领域的采用率较低，同时面临 PostgreSQL 的强有力竞争者。另外，MySQL 的草根版本正在茁壮成长，相对于官方的企业版和社区版，MySQL 的分支产品似乎得到了更多的社区支持。不仅如此，收购案并没有真正尘埃落定。尽管目前甲骨文对 Sun 的收购已经获得了美国和欧盟的认可，但能否通过我国和俄罗斯反垄断部门的审查仍需时日。

4. 互联网两巨头 PK“云计算”

3 月 28 日消息，由深圳市政府与数字中国联合会共同主办的 2010 中国(深圳)IT 领袖峰会今日在深圳五洲宾馆举行，百度 CEO 李彦宏和阿里巴巴主席董事局主席马云就云计算展开了交锋。

李彦宏认为，云计算的理念已经产生了很多年，是新瓶装旧酒，没有新东西。早期的时候，15 年前大家讲客户端跟服务器这个关系，再往后大家讲基于互联网 web 界面的服务，现在讲云计算，实际上本身都是一样，主要活都是在服务器这端来做，客户端所需要做的事情越来越简单。

对于传统软件产业向云计划靠拢，李彦宏表示担忧，他认为这会存在左手打右手的问题。你说你是微软的 office，你想弄成所有东西都在云端来做，在客户端什么都不要了，这个多多少少有点左手打右手，吃力不讨好的一个情形。

马云不认同李彦宏的观点，他认为云计算最后会是一种分享，数据的处理、存储然后跟分享的机制。他警告说不能小瞧这种机制，云计算可能蕴藏颠覆性力量。我最怕的是老酒装新瓶的东西，你看不清他在玩什么，突然爆发出来最可怕。假如从来没有听说的，这个不可怕。雅虎当年做搜索引擎，然后 Google 出来了，雅虎很多人认为跟我们也差不多，后来几乎把他们搞死。

马云表示，阿里巴巴对云计算充满了信心，能够为社会创造出更大的价值。我们不是觉得这又找到一个新的矿产，我们阿里巴巴拥有大量消费数据、支付宝交易数据，我们觉得这些数据对我们有用，但是可能对社会更有用，比如我们从小企业的信息掌握到整个中国经济、世界经济的问题，从消费者数据给制造业数据，让他们生产出更好产品卖给消费者。

马云同时暗示，云计算是大势所趋，是阿里巴巴必须要实施的战略。如果能够把这个数据分享给社会，是一个很有用的。如果有一天我们不做这个，百度、腾讯就会把我们赶出电子商务门口。所以这是客户需要，如果我们不做，将来会死掉。

（声明：以上文章均来自互联网，由 Barry 编辑）

Nutch + Hadoop 构建商用分布式搜索引擎的问题探究

（作者：Barry）

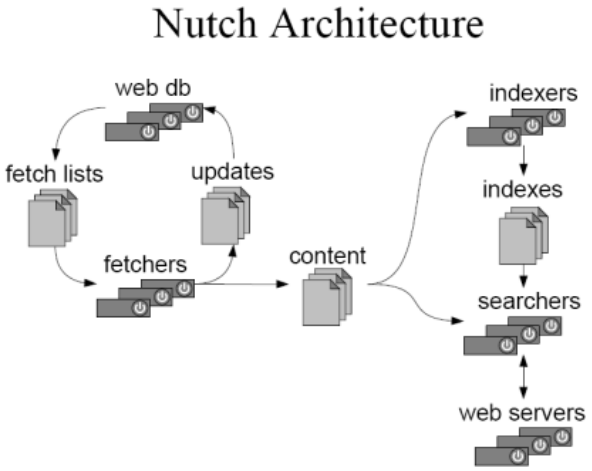
1. 题记

众所周知，Nutch 和 hadoop 本是一家，从 0.X 版本开始，Hadoop 从 Nutch 中剥离出来成为一个开源子项目，Hadoop 的初衷是为解决 Nutch 的海量数据爬取和存储的需要。相信 Hadoop 的 fans 都很清楚，Hadoop 其实并非一个单纯用于存储的分布式文件系统，而是一个被设计用来在由普通硬件设备组成的大型集群上执行分布式应用的框架（Framework）。Hadoop 包含两个部分：一个分布式文件系统 HDFS (Hadoop Distributed File System)，和一个 MapReduce 实现。因此，Hadoop 的目标是为开发分布式应用提供一个框架，而不是像 OpenAFS, Coda 那样为存储提供一个分布式文件系统。搜索引擎就是一种典型的分布式程序，而 Nutch 正是基于 Hadoop 开发的一个应用，Nutch 发展到目前的 1.0 版，确实成熟了不少，这里请允许笔者下一疏浅的定论，Nutch 离商用的海量分布式搜索引擎应用尚有一定的距离。

本文就作者在工作中如何改造 Nutch 以实现支持海量数据检索的分布式应用所遇到的问题一一分享给各位，一些思考或许会贻笑大方，但也无妨，毕竟作者的一些粗浅理解，也是日常工作的一些实践总结，期盼对后来者有所启发。

2. Nutch 工作原理解释

在这里，很容易明确，Nutch 的架构如下图所示。



基于 Nutch 的分布式搜索引擎，其构架可以分割为：分布式爬虫器（Crawler），分布式文件存储系统（HDFS）、检索服务系统（Searcher）四个部分，下面对其进行简要概述，不明白的同仁可详细借鉴 Hadoop 或 Nutch 的 Wiki（在 Hadoop 技术交流群里，我一直提到刚刚接触

hadoop 的新手们，应该先把这两个 wiki 全部读一遍，这里我是这么读过来的，并无他意）。

（1）分布式爬虫器的工作流程为:

首先 Crawler 根据 WebDB 生成一个待抓取网页的 URL 集合叫做 Fetchlist，接着下载线程 Fetcher 开始根据 Fetchlist 将网页抓取回来，如果下载线程有很多个，那么就生成很多个 Fetchlist，也就是一个 Fetcher 对应一个 Fetchlist。然后 Crawler 根据抓取回来的网页 WebDB 进行更新，根据更新后的 WebDB 生成新的 Fetchlist，里面是未抓取的或者新发现的 URLs，然后下一轮抓取循环重新开始。这个循环过程可以叫做“产生/抓取/更新”循环[1]。

（[1] http://www.mysoo.com.cn/news/2007/200711676.shtml）

在 Nutch 中，Crawler 操作的实现是通过一系列子操作的实现来完成的。这些子操作 Nutch 都提供了子命令行可以单独进行调用。下面就是这些子操作的功能描述以及命令行，命令行在括号中。

1) 创建一个新的 WebDb (admin db -create).
2) 将抓取起始 URLs 写入 WebDB 中 (inject).
3) 根据 WebDB 生成 fetchlist 并写入相应的 segment(generate).
4) 根据 fetchlist 中的 URL 抓取网页 (fetch).
5) 根据抓取网页更新 WebDb (updatedb).
6) 循环进行 3－5 步直至预先设定的抓取深度。
7) 根据 WebDB 得到的网页评分和 links 更新 segments (updatesegs).
8) 对所抓取的网页进行索引(index).
9) 在索引中丢弃有重复内容的网页和重复的 URLs (dedup).
10) 将 segments 中的索引进行合并生成用于检索的最终 index(merge).

（2）分布式文件系统

Nutch 爬取的文件按按块存放在你搭建好的 HDFS 上，其文件目录及其结构如下。

- crawldb 目录下面存放下载的 URL,以及下载的日期，用来页面更新检查时间。
- linkdb 目录存放 URL 的关联关系，是下载完成后分析时创建的，通过这个关联关系可以实现类似 google 的 pagerank 功能。
- segments 目录存储抓取的页面，下面子目录的个数与获取页面的层数有关系。Lucene 中的 segment 和 Nutch 中的不同，Lucene 中的 segment 是索引 index 的一部分，但是 Nutch 中的 segment 只是 WebDB 中 各个部分网页的内容和索引，最后通过其生成的 index 跟这些 segment 已经毫无关系了。

内含有 6 个子目录 ：

content: 下载页面的内容

crawl_fetch: 下载 URL 的状态内容

crawl_generate: 待下载的 URL 的集合，在 generate 任务生成时和下载过程中持续分析出来

crawl_parse: 存放用来更新 crawldb 的外部链接库

parse_data: 存放每个 URL 解析出来的外部链接和元数据

parse_text: 存放每个解析过的 URL 的文本内容。

- index 目录存放符合 lucene 格式的索引目录，是 indexs 里所有的索引内容合并后的完整内容，这里的索引文件的内容基本与 lucene 的索引文件一致。
- indexs 目录存放每次下载的索引目录，存放 part-0000 到 part-0003

（3）分布式检索服务系统

实际上 Nutch 的分布式检索是将索引块分别存放在不同的机器上，每个目录中存放完整的一块索引或几块索引，注意这里的一块或几块，实际上你可以将（2）采集得到的 Segments 和 Index 拷贝出来就可以成为独立的一个索引块，这样的索引块在作为一个检索的节点的机器上是可以多块存放的。

Nutch 的分布式检索服务与 HDFS 是没有关系的，提供检索服务的索引块存放在 local 文件系统中，而不是 HDFS 上，根据 Nutch 官方 WIki 的解释认为，是效率决定，不能将索引文件和快照服务文件放在 HDFS 上。其分布式检索的服务是基于 RPC 通讯进行的的，当然你可以用 RMI 机制+Lucene 构建自己的分布式搜索引擎，具体文章请见本期另一篇文章。

其实要真正理解 Nutch 的工作细节，还得阅读其源码，限于篇幅，作者不在一一 叙述一些细节。

3. 利用 Nutch 构建商用分布式搜索引擎存在的问题思考

1）爬虫效率问题

笔者测试的 Nutch 集群构建的爬虫方式，效率不高，爬取页面的速度远比不上 Heritrix 等爬虫工具，这跟 Nutch 过滤链接和抓取过程中的一系列操作有较大关系，也许是作者的配置有问题，但作者试过不下几十次，没有一次让作者获得满意的抓取速度。

2）爬虫的个性化问题

Nutch 的爬虫提供两种方式，一种是站内爬虫、一种是全网爬虫，如果对上百个网站进行站内抓取就是一个很麻烦的问题，链接过滤采取正则匹配的方式，效率不是很高，如果上千个网站，正则怎么写？

3）结构化文本的抽取问题

Nutch 没有提供文本的正文抽取等组件，仅仅是去掉标签进行索引，结构化的标题、时间、正文抽取还需你自己写程序解决。

4）网站重访爬虫问题

Nutch 的官方 Wiki 虽然给出了相关网站重访爬取的 Shell 脚本，看看这脚本，无异于重来一次，对于海量数据估计没人能忍受完全重爬一次。

5）索引分发与更新问题

如果说 Nutch 的爬虫还基本可以满足商用搜索引擎的基本需求的话，Nutch 的索引分发和

索引更新就完全是不能满足商用搜索引擎的需要了，之前提到 Nutch 分布式搜索的实现是本地文件索引+RPC 通信+检索 API 实现的，这里就有几个问题需要解决:

- 放在每台机器提供索引服务的 Local 的索引每块放多大合适？如果是放几块 Segments+index，那每块多大合适？放几块？按照 Nutch 的爬虫方式，如何控制爬的数据按要求在数据量到一定程度时就调到下一块？Nutch 的爬虫是一次出去抓，抓到规定的层数和规定页面为止才回来，当然你可以启用分步抓取命令，就像上一节的 1）-10）一样，手动去解决，但是最后形成索引块的大小也是一个难以控制的问题。
- 按照分布式检索的要求，Nutch 的抓回来的数据自然是放在 HDFS 之上，Index 的过程自然是借助 Hadoop 平台的 Map/Reduce 能力进行分布式的索引构建，但是在 Hadoop 构建好的索引后如何分布到本地去，很多人肯定很简单的回答，CopeToLocal 呗！但是，如果上百 GB 的索引拷贝是非常耗时的！尤其是索引更新要求时效性较高时，这个拷贝的时间太长了。这个问题解决的关键点在于，Segments 与 Index 的合并，什么时候合并，以什么方式合并，在本地还是平台合并。
- 索引更新后如何重启检索服务。Nutch 考虑了在一个索引服务宕机时是可以提供检索服务。但是如下几点却使得 Nutch 难以满足商业搜索的索引更新:（1）在一块新索引加上后，并启动该索引服务后，更新 Search-servers.txt, Nutch 随后会有一个检索服务加载中断的过程的，而在商用搜索引擎中，这个中断是不能忍受的。（2）如果在启动检索 Web 工程时，如果 Search-servers.txt 中某个端口的索引服务没有开启，整个 Nutch 检索是不能服务的。（3）Nutch 是惰性读取 Search-servers.txt，就是说你或许更新了，检索它也不一定马上知道。解决这个问题的关键点在与改变 Nutch 读取 Search-servers.txt 的方式。

6）大数据的排重与检索排序问题

在大数据量的情况下，Nutch 的 Segments 合并是非常耗时的工作，有时候几乎不可能去合并它，这样可能使得每块 Index+segments 是单独的一块索引服务，这样，这些单独的块之间的数据只是相对块内的排序，对检索结果是有一定影响的。

而 Nutch 的排重是在 Index 阶段进行的，如何不 Merge Index,全局数据的排重也就没法做。

7）搜索服务问题

Nutch 的分页、搜索提示、查询联想、站内搜索、摘要等都需要改进。

8）负载均很与检索 cache 问题

Nutch 的检索的 Cache 方式是非常低效的，商用的搜索引擎必须改进其 Cache 机制，另外，还需要考虑 Nutch 检索时的负载均衡问题。

作者成文匆匆，有些问题论述不够准确之处，请大家指正。

4.基于 Nutch+Hadoop 构建商用分布式检索构架方式

作者认为，可以利用 Nutch 和 Hadoop 的方式来构建一个适合商用的分布式搜索引擎，但

其关键点是解决以上的几个问题。Nutch 的优势在与 Hadoop 的结合，提供了一个支持海量数据搜索的分布式检索构架，基于此构建，我们能做出一些合理的改变，以适应构建商用搜索或垂直搜索的需要。

对 Nutch+hadoop 的功能组件进行重组。其主要流程为:

（1） 采用 Heritrix 爬取网页文本;

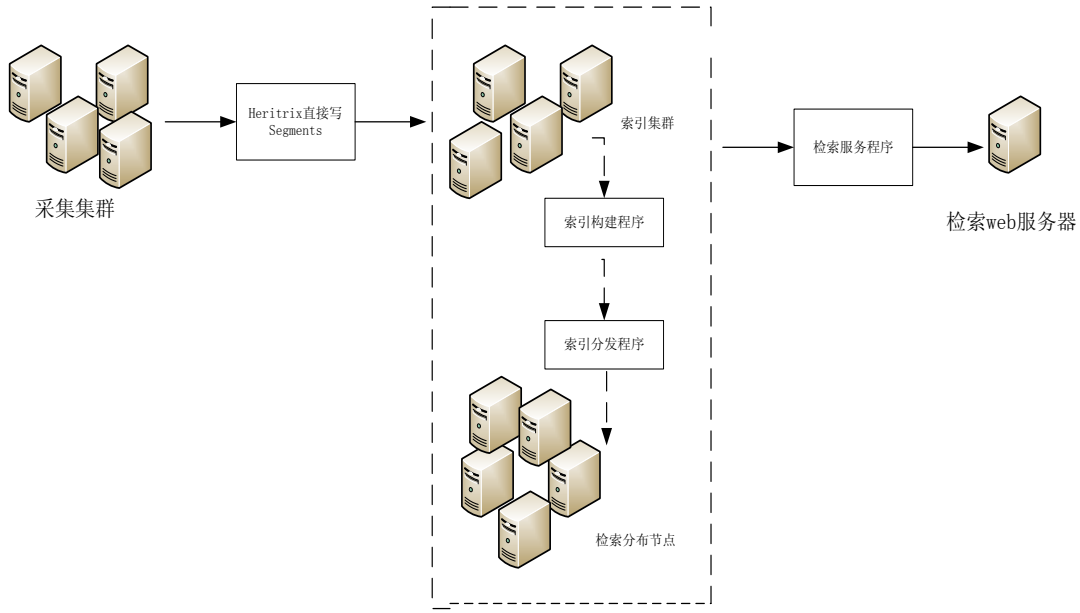
（2） 得到的数据写入 Nutch 的 Segments，交由 HDFS 存储;

（3） 在 Segments 的基础上做链接分析和文本抽取工作;

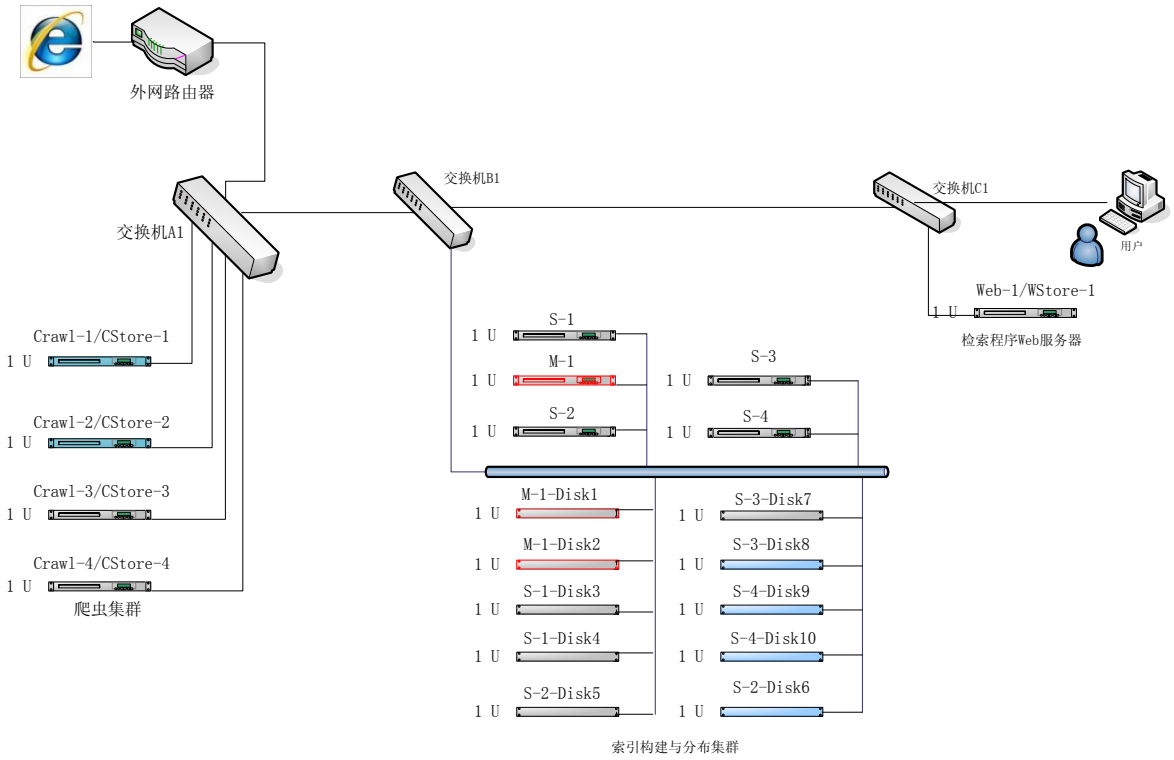
（4） 构建分布式索引分发机制和更新机制;

（5） 利用 Nutch 提供分布式检索。

其中（1）、（2）点将在本人的同事 mingyuan 的文章中进行论述，（3）、（4）、（5）涉及的问题及其关键点在前文中基本进行了论述。其中设计的技术组件和程序流程见下图所述,其中几个方框标明的是需要读者去实现的。



Heritrix + Nutch +Hadoop 的网络拓扑结构图，如下所示。



5. 后记

因为成文时间仓促，持续研究 Hadoop 相关技术一路走来，有收获，也有感触，跟大家一样在摸索中前进。有诸多细节在此难以一一叙述，本文粗浅地描述了一些个人体验，意在抛砖引玉，各位 Hadoop&Nutch 的爱好者尽可表述观点，个人邮箱 beyiwork@gmail.com,疏漏之处，还望多多指正，敬候来音。

支持自定义爬虫的 Nutch segment 文件存储接口改写

作者：mingyuan Email：cn.mingyuan@foxmail.com

本文的初衷是所在的项目组前期采用 Heritrix 方式爬取数据，数据的存储形式是 ARC 格式，需要解决的是在不改变用户爬虫程序的情况下，让用户的爬虫程序采用 Nutch 的文件存储形式，以便利用 Nutch 的分布式索引构建、检索功能，从而可以实现前端 Heritrix 或其他形式爬取数据，后端 Nutch 统一处理、索引、并提供检索。

Nutch 自带的爬虫有时不能满足我们的抓取需求，需要使用自定义的爬虫对网页进行抓取。如何对抓取的数据进行存储并建立索引、提供检索是本文将要探讨的主要内容。

Nutch 使用的是 Lucene 的索引机制，关键数据存储在 segment 中，其余的数据皆可从 segment 中生成。能否生成 segment 关系到方案成败。本方案主要由以下三部分构成:

生成 segment

修复 segment

建立索引

1 生成 segment

Nutch 的 1.0 版本中有一个 Arc 文件转换为 segment 的工具类：ArcSegmentCreator，使用此类可以生成 segment。我们可以对此类加以改造进行使用。

由于 ArcSegmentCreator 是 MapReduce 程序，而我们需要将抓取得数据即时的追加到文件中，所以需要将它改造成非 MapReduce 的程序供自定义的爬虫使用。

（1）CreateSegment(Path arcFiles, Path segmentOutDir)

此方法相当于程序的控制部分，给我们提供了程序处理流程的主要信息。此方法的主要部分摘录如下:

```
job.setInputFormat(ArcInputFormat.class);//指定输入数据格式
job.setMapperClass(ArcSegmentCreator.class);//MapperClass
FileOutputFormat.setOutputPath(job, new Path(segmentOutDir, segName));//输出路径
job.setOutputFormat(FetcherOutputFormat.class);//输出格式
job.setOutputKeyClass(Text.class);//输出的 Key Class
job.setOutputValueClass(NutchWritable.class);//输出的 Value Class
```

通过对以上几行代码的分析可知: ArcSegmentCreator 中的 map()即为 MapReduce 中的 Map 部分；输出的路径由 FileOutputFormat.setOutputPath() 指定；结果的输出格式在 FetcherOutputFormat 中指定。下面对 Map 方法进行分析。

(2) map()方法

map(Text key, BytesWritable bytes,OutputCollector<Text, NutchWritable> output, Reporter reporter)

key 为 url，bytes 为网页源码，output 收集处理之后的数据。在 map 方法的结尾我们可以看到，在 map 方法进行一系列处理之后 map 过程并没有结束而是交给 output 方法继续执行未完的处理。

(3) output()方法

output(output,segmentName, url, datum, null, null, CrawlDatum.STATUS_FETCH_RETRY)

;Output 方法最终生成 Content、CrawlDatum、ParseImpl 交给 FetcherOutputFormat 进行持久化处理。

(4) FetcherOutputFormat.java

getRecordWriter()返回的 RecordWriter 实现了对 Content、CrawlDatum、ParseImpl 的处理，将数据分别写入到 content、crawl_fetch、crawl_parse、parse_data、parse_text 目录生成 segment。

2 修改程序为非 mapreduce 程序

ArcSegmentCreator 是针对已经抓取的数据进行批量处理的，不能满足数据即时写入的需求，需要对其加以改造，使之脱离 Hadoop 环境也能运行。

在 ArcSegmentCreator 基础上修改代码比较简单，将原来的程序看作黑盒，我们只需关注程序的输入与输出即可。流程大概是这样的：数据首先输入到 map()中，然后由 map()流向 output(),数据经由 output()方法处理之后，由 write()方法完成持久化操作，生成 segment。

ArcSegmentCreator 的输入由 ArcInputFormat.class 确定，从 map()方法的参数及其内容来分析，参数中的 key 为网页的 url，而 bytes 则为网页源码。这两项数据我们可以从爬虫那里轻易获得，这样就得到了输入。

输出方面需要注意的问题是怎样将由 hadoop 控制的输出转化为我们自己控制的输出。通过上一节的分析可知 FetcherOutputFormat 中的 RecordWriter 负责数据的持久化操作，由此可将 RecordWriter 的 write()方法提取出来，在 output()方法结束时，将结果交给 write()方法进行处理，完成持久化操作，从而解决输出问题。需要注意的是 ArcSegmentCreator 使用 MapReduce 进行数据处理，其中包含了对 url 的排序过程。我们知道 segment 中的 content、crawl_fetch、parse_data、parse_text 目录存放的是 MapFile，crawl_parse 目录存放的是

SequenceFile；由于数据实时性等方面的限制，我们不可能在抓取大量数据之后再将它们写入到 segment 中，而需要在抓取到网页之后马上写入。由于 MapFile 文件存储有序数据，不满足追加的需求，从而只能使用 SequenceFile 保存数据。这样在实现时就需要将 RecordWriter 中涉及到的 MapFile.Writer 用 SequenceFile.Writer 来替换。

3 修复 segment

现在生成的 segment 还不是最终的 segment，因为 content、crawl_fetch、parse_data、parse_text 这些目录下的数据要求是 MapFile，而现在只是 SequenceFile，需要将其转化为 MapFile 之后才能使用。

转化使用 Hadoop 平台，分两步进行：

第一步为将 SequenceFile 按照 url 进行排序。排序可以使用 hadoop-*-examples.jar 中的 Sort.class 完成：

```
Path seqFilePath = new Path(filePath, "data");
Path seqSortedFilePath = new Path(filePath, "data.sorted");
String[] contentArgs = { "-r", "1", "-inFormat", "org.apache.hadoop.mapred.SequenceFileInputFormat", "-outFormat", "org.apache.hadoop.mapred.SequenceFileOutputFormat", "-outKey", keyClass.getName(), "-outValue", valueClass.getName(), seqFilePath.toString(),seqSortedFilePath.toString() };
FileSystem fs = FileSystem.get(filePath.toUri(), conf);
Sort.main(contentArgs);
```

第二步为使用 MapFile 的 fix()方法重建索引，生成 MapFile。

```
// 将 SequenceFile 转化为 MapFile
MapFile.fix(fs, contentPath, Text.class, Content.class, false, conf);
MapFile.fix(fs, fetchPath, Text.class, CrawlDatum.class, false, conf);
MapFile.fix(fs, textPath, Text.class, ParseText.class, false, conf);
MapFile.fix(fs, dataPath, Text.class, ParseData.class, false, conf);
```

4 建立索引

建立索引的过程比较简单，主要是以下几步：

（1） 根据 segment 生成 crawldb：bin/nutch updatedb crawldb –dir segment

（2） 根据 segment 生成 linkdb：bin/nutch invertlinks linkdb –dir segment

（3） 生成 indexes：bin/nutch index indexes crawldb linkdb segment/*

（4） 如果需要还可以进行排重以及合并索引的操作。

Nutch 中 mapreduce 应用的几个特殊点

author: 皮冰锋 email: pi.bingfeng@gmail.com

Nutch 重在网络爬虫，主要包括 injector, generator, fetcher, crawldb updater, linkdb updater, indexer 及 merge index。除了最后的 merge index 过程之外，其他的都用到了 mapreduce 的计算思想，是很值得我们学习的优秀源码。

在这篇文章中，我重点讲解几个特别值得我们学习的地方，也许在实际的分布式计算中都有可能用到。

1.SpeculativeExecution 的 job

Mapreduce 的精髓是分割任务交给多个 datanode 去执行，在同一时刻，可能有多个 datanode 在执行同一个 task，谁的 task 先完成，namenode 就采用谁的结果，并命令其他 datanode 停止对这个 block 的计算。

我归结这样的好处有两点：

(1) “多劳者多得”，计算快的节点总是能得到更多的计算任务，保证了整个计算任务在最短的时间内完成；

(2) 进一步体现了 hadoop 的抗灾冗余思想：即使某个 datanode 失效，还有其他的 datanode 在执行同样的 task，计算不会终止。

但是在某些情况下，出于某种特殊原因，我们不希望有多个 datanode 执行同一个 task，比如爬虫中为了保持爬虫的礼节性，不能有多个机器同时去抓取一个站点。这时就可以设置 job 的 SpeculativeExecution 属性：

job.setSpeculativeExecution(false);

默认情况下，SpeculativeExecution 是 turn on 的状态，即多个 datanode 同时执行一个 task。

还可以分开控制 mapper 及 reducer 的 SpeculativeExecution 状态：

```
Job.setMapSpeculativeExecution(false);
Job.setReduceSpeculativeExecution(false);
```

2. Multiple Outputs

Mapreduce 默认允许有多个 input，但只有一个 output 的，从定义 inputpath 及 outputpath 的方法名就可以看出：

定义输入： FileInputFormat.addInputPath(job, input);

定义输出： FileOutputFormat.setOutputPath(job, output);

但在实际应用中，根据需求经常会遇到输出结果为多个文件的情况。

在 nutch 中， fetcher 的 FetcherOutputFormat 给了我们很好的示例。

FetcherOutputFormat 继承了 OutputFormat 接口，实现了自定义的 RecordWriter,接收在 fetcher 的主线程中 collect 的所有 NutchWritable 对象,包括 Content, CrawlDatum, ParseImpl 等。

```
public class FetcherOutputFormat implements OutputFormat<Text, NutchWritable> {
public RecordWriter<Text, NutchWritable> getRecordWriter(
final FileSystem fs,
final JobConf job,
final String name,
final Progressable progress) throws IOException {
        //TODO something
  }
```

RecordWriter是我们关注的重点，在它的write函数中，将上述封装的NutchWritable又还原成封装之前的Content, CrawlDatum, ParseImpl对象，再根据对象的不同写到不同的文件中去。

```
    public void write(Text key, NutchWritable value)
            throws IOException {
            Writable w = value.get();
            if (w instanceof CrawlDatum)
                fetchOut.append(key, w);
            else if (w instanceof Content)
                contentOut.append(key, w);
            else if (w instanceof Parse)
                parseOut.write(key, (Parse)w);
    }
```

这种做法比较独特，我很欣赏。如果想了解更多细节，大家可以去仔细看看FetcherOutputFormat这个类的源码。

3. Special Inputformat

Mapreduce 中提供了几种较为通用的文件格式，包括SequenceFileInputFormat, MapFileInputFormat, TextFileInputFormat等，这些都有自定义的split方法，所以不用太在意很多。

但对于有些自定义的输入文件格式，不好切分也不好采用传统的数据读取方法，可以将整个文件作为一个split，自定义RecordReader，实现next方法。

拿Indexer的DeleteDuplicates 类来说，它需要读取lucene的索引文件，但lucene的索引格式很复杂，不易切分，所以采取了读取完整的Index方法，如下:

```
    public static class InputFormat extends FileInputFormat<Text, IndexDoc> {
        private static final long INDEX_LENGTH = Integer.MAX_VALUE;

        /** Return each index as a split. */
        public InputSplit[] getSplits(JobConf job, int numSplits)
          throws IOException {
          FileStatus[] files = listStatus(job);
          InputSplit[] splits = new InputSplit[files.length];
          for (int i = 0; i < files.length; i++) {
            FileStatus cur = files[i];
            splits[i] = new FileSplit(cur.getPath(), 0, INDEX_LENGTH, (String[])null);   //该
split 的长度为无限长，所以不会切分索引文件
          }
          return splits;
    }
```

```
    public class DDRecordReader implements RecordReader<Text, IndexDoc> {
        //TODO: 自定义的读取索引文件方法，遍历得到 Index 的每个 document。
    }
```

Nutch中还有很多值得我们借鉴的地方，有兴趣的同学可以直接分析源代码。

Java RMI + Lucene 构建分布式检索应用初探

（作者：Barry）

Nutch 给出了分布式搜索引擎的一套解决方案，我们可不可以用 java 自己写一个分布式搜索，答案当然是肯定的。最近在解决这样一个问题，Web 工程调用检索的问题，前期我们采用的 web 与索引放在同一个服务器上解决，但近期发现索引在达到数十 GB 级别的时候，索引更新过快是一间很耗资源的事情，想弄个异地索引，远程服务的形式，很想写一个轻量级远程索引和检索调用模块。这里给出一个简单的远程应用的文本搜索 Demo。

1.服务器端接口

SearchInterf.java

```
package rmi.test;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;
/**
 * 这是一个远程的检索接口
 *  @author beyiwork
 */
public interface SearchInterf extends Remote{

    // 声明根据关键词和检索字段，检索索引，并返回 NewsSearchResult 集合
    /**
     * @param filed :检索字段集，比如"title,content"
     * @param keywords：按域检索的关键词集合,对应每个域的提交的关键字
     */
    public List<NewsSearchResult> queryList(String[] fileds,String[] keywords) throws RemoteException;
}
```

2.服务器端得检索业务实现类

SearchService.java

```
package rmi.test;
/**
 * @author beyiwork
 *  此类是服务器端的检索业务实现类
 */
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.Term;
import org.apache.lucene.search.BooleanClause;
import org.apache.lucene.search.BooleanQuery;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.TermQuery;

public class SearchService extends UnicastRemoteObject implements SearchInterf{

    private static final long serialVersionUID = -7810768762217713731L;
    //空的构造函数
    public SearchService() throws RemoteException {
        super();
        // TODO Auto-generated constructor stub
    }

    //真正的检索业务处理类
    public List<NewsSearchResult> queryList(String[] fields, String[] keywords)
            throws RemoteException {
        // TODO Auto-generated method stub
        IndexSearcher searcher = null;
        Hits hits = null;
```

```
//采用布尔检索
BooleanQuery query = new BooleanQuery();

//构建检索条件
for(int i = 0;i<fields.length;i++)
{
    Term term = new Term(fields[i],keywords[i]);
    TermQuery tq = new TermQuery(term);
    query.add(tq, BooleanClause.Occur.SHOULD);
}
try {
    //指定读取索引的存储路径
    String indexDir = "D:\\project\\IndexCreator\\newfandongIndex";
    //初始化 search
    searcher = new IndexSearcher(indexDir);
    //查询
    hits = searcher.search(query);

} catch (CorruptIndexException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
//读取 Hits 包装检索结果
ArrayList<NewsSearchResult> newsList = new ArrayList<NewsSearchResult>();
try {

    for (int i = 0; i < hits.length(); i++) {
        Document doc = hits.doc(i);
        String id = doc.get("id");
        String title = doc.get("title");
        String site = doc.get("site");
        String pubtime = doc.get("pubtime");
        String sourceurl = doc.get("sourceurl");
```

```
                String content = doc.get("content");
                String genus = doc.get("genus");
                String tempid = doc.get("tempid");
                String type = doc.get("type");
                newsList.add(new NewsSearchResult(id, title, site, pubtime,
                        sourceurl, content, genus, tempid, type));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return newsList;
    }

    //这是一个测试用例，略过
    public static void main(String[] args) {
        String[] fields = new String[]{"title","content"};
        String[] keywords = new String[]{"中国","中国"};
        try {
            SearchService searchService = new SearchService();
            List<NewsSearchResult> newsList =   searchService.queryList(fields, keywords);
            for(NewsSearchResult newsSearchResult : newsList)
            {
                System.out.println(newsSearchResult.getTitle());
                System.out.println(newsSearchResult.getId());
                System.out.println("------------------------------------");
            }
        } catch (RemoteException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}
```

3.注册 RMI 应用的服务器端程序

绑定到 Localhost 的默认端口

SimpleSearchServer.java

```
package rmi.test;
/**
 * 这是一个将检索服务绑定到RMI 应用的类
 * @author beyiwork
 *
 */
import javax.naming.Context;
import javax.naming.InitialContext;

public class SimpleSearchServer {
    public static void main(String[] args) {
        try {
            //实例化一个应用
            SearchInterf service1 = new SearchService();
            //将应用绑定到 RMI 的端口
            Context namingContext = new InitialContext();
            namingContext.rebind("rmi:SearchService1", service1);
            System.out.println("服务器注册了一个 SearchService");
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

4.Client 端的检索提交与实现

```
package rmi.test;
/**
 * 这是一个 client 提交检索的实现
 * @author beyiwork
 */
import java.util.List;
```

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class SimpleSearchClient {

    public static void main(String[] args) {
        String url = "rmi://localhost/";
        try {
            //声明应用的 RMI 服务
            Context namingContext = new InitialContext();
            SearchInterf service1 = (SearchInterf)namingContext.lookup(url+"SearchService1");
            //System.out.println("aaaaaaaa");
            //构建检索条件
            String[] fields = new String[]{"title","content"};
            String[] keywords = new String[]{"中国","中国"};
            //调用服务进行检索
            List<NewsSearchResult> newsList =   service1.queryList(fields, keywords);
            int count = 0;
            for(NewsSearchResult newsSearchResult : newsList)
            {
                System.out.println(newsSearchResult.getId());
                System.out.println(newsSearchResult.getTitle());
                System.out.println("-----------------------------------");
                if(count++>10)
                {
                    break;
                }
            }
        } catch (NamingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
```

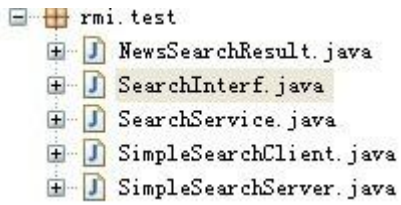
```
            }
        }
    }
```

5.NewsSearchResult.java 的源码

```
package rmi.test;

import java.io.Serializable;

/**
 * 此类是检索结果的POJO，一定要序列化哦
 * @author beyiwork
*/

public class NewsSearchResult implements Serializable{
    private static final long serialVersionUID = -2685508592396115327L;

    private String id;//id
    private String title;//标题
    private String site;//站点
    private String pubtime;//发布时间
    private String sourceurl;//来源 URL
    private String content;//内容
    private String genus;// 分类号
    private String tempid;//分类号
    private String type;//类型

    public NewsSearchResult() {
        // TODO Auto-generated constructor stub
    }
    public String getSourceurl() {
        return sourceurl;
    }
    public void setSourceurl(String sourceurl) {
        this.sourceurl = sourceurl;
```

```
        }
    .... Get set
    }
```

6.程序的结构



7.命令行下程序的运行（一定要注意路径）

（1）Classpath 设定

将 lucene 的 包 放 进 去 ， D:\project\distributedSearch\bin>set classpath=".;D:\project\distributedSearch\lib\lucene-core-2.3.jar"

（2）注册 rmi 应用

D:\project\distributedSearch\bin>start rmiregistry

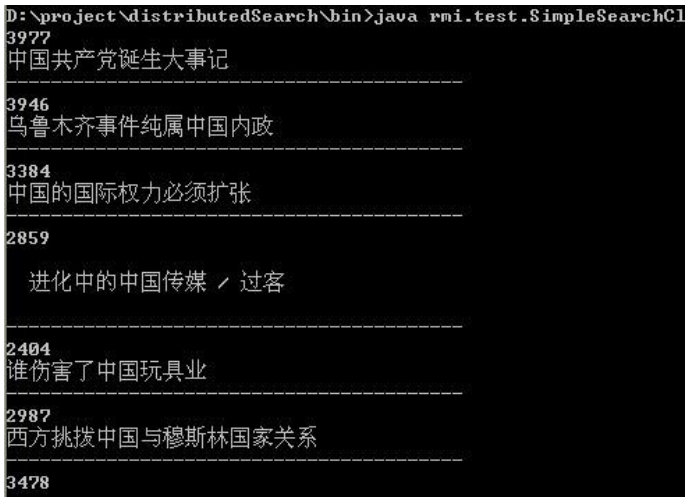
（3）启动服务器端服务：

D:\project\distributedSearch\bin>start java rmi.test.SimpleSearchServer

（4）检索，试试：

D:\project\distributedSearch\bin>java rmi.test.SimpleSearchClient

（5）检索结果页面



8.总结

这不能算一个严格意义上的分布式检索，但是基于此 Demo 是可以开发出一个分布式检索应用的，笔者将继续改进到一个成熟的 Release 版本，提交给各位同仁，继续改进。

一对多的表关联在 Mapreduce 中的应用(续)

Author: 皮冰锋 Email: pi.bingfeng@gmail.com

在《Hadoop 开发者入门专刊》中，我们有一篇《表关联在 mapreduce 中的应用》介绍，以“商品表”和“支付表”为例，讲述了如何使用 mapreduce 将两个表链接起来。也许部分读者已经发现在具体的实现中存在一个假设，即同一个“tradeID”只能匹配一个“payID”，是“一对一”的关系。但在很多情况下，我们需要处理“一对多”的关系，即同一系列的“tradeID”可以匹配多个“payID”。

本文还是以“商品表”和“支付表”为例，详细介绍如何实现“一对多”的表关联。

1. 问题描述

假设有两个表格:

（1）商品表(trade table)，数据格式如下:

```
product1"trade1
product2"trade2
product3"trade3
```

（2）支付表(pay table)，数据格式如下:

```
product1"pay1
product2"pay2
product2"pay4
product3"pay3
```

目标：将这两个表格根据相同的 productID 关联起来，如下:

```
trade1    pay1
trade2    pay2
trade2    pay4
trade3    pay3
```

2. 最容易的实现方法

为了实现“一对多”的匹配，我们一开始想到的办法，肯定是在 reduce 阶段用一个数组缓存多个“payID”，然后再将“tradeID”与多个“payID”分别对应，类似如下操作:

```
public class CommonReduce extends MapReduceBase
implements Reducer<Text,Text,Text,Text> {
```

```
public void reduce(Text key, Iterator<Text>values,
OutputCollector<Text,Text> output, Reporter reporter) throws IOException {
        String tradeID = "";
        ArrayList<String> payIDlist =new ArrayList<String>();
        while( values.hasNext()){
            String value = values.next().toString();
            int index = value.indexOf("'");
            if( index == -1)      //skip bad record
                continue;
String subValue =   value.substring(index + 1 ,
value.length());
            if( value.startsWith("supid")){ //is trade                    tradeID      =
subValue;
            }
            else if( value.startsWith("buyid")){
                payIDlist.add(subValue);
            }
        }
        for( String payID: payIDlist){
            output.collect(new Text(tradeID), new Text(payID));
    }
}
```

问题分析：如果数据量较小，这种实现方式不会有问题；但如果数据量偏大，payIDlist 会膨胀很快，极有可能出现“out of memory”问题。

3. 比较好的解决办法

思路：借鉴“secondary sort”的思想(详细介绍见文章《mapreduce 中 value 的二次排序》)，对 reduce 端的 values 集合排序，使 values 的第一个元素始终为“tradeID”，紧随其后的才是多个“payID”。

对前一个方式的实现方式做稍微调整，如下

（1）Mapper 的实现

```
public class PreMapper extends MapReduceBase
implements Mapper<LongWritable, Text, TextPair, Text> {
public void map(LongWritable key, Text val,
OutputCollector<TextPair, Text> output, Reporter reporter)
```

```
                        throws IOException {
            //input file path
    String path=
    ((FileSplit)reporter.getInputSplit()).getPath().toString();

    String[] line = val.toString().split("\"");
            if( line.length <2){    //skip bad value
                return;
            }
            TextPair kr =new TextPair();
            String productID = line[0];
            kr.setText(productID);      //set product ID;

            Text kv =new Text();
            if(path.indexOf("action")>=0){ //trade table
                kr.setID(0);        //用于 TextPair 的排序
                String tradeID = line[1];
                kv.set(tradeID);        //value is tradeID
            }
    else if(path.indexOf("alipay")>=0){ // pay table
                kr.setID(1);       //用于 TextPair 的排序
                String payID = line[1];
                kv.set(payID);     //value is   payID
            }
            output.collect(kr, kv);
    }
```

Mapper 阶段的 key 换成了封装的 TextPair 对象，保存“productID”及一个用于排序的 int，对于“tradeID”，int 赋值为 0，对于“payID”,int 赋值为“1”。Mapper 的 value 为“商品 ID”或“支付 ID”，不需要组装。

（2）TextPair 的实现代码如下，重点要关注 compareTo 的实现:

```
public class TextPair implements WritableComparable{
    private String text ;
    private int id;
    public TextPair(){}
    public TextPair(String text, int id){
        this.text = text;
```

```
        this.id = id;
    }
    public void setText(String text){
        this.text = text;
    }
    public void setID(int id){
        this.id = id;
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        this.text = in.readUTF();
        this.id = in.readInt();
    }
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(text);
        out.writeInt(id);
    }
    @Override
    public int compareTo(Object o) { //先比较 text，在比较 id
        TextPair that = (TextPair)o;
        if( !this.text.equals(that.text)){
            return this.text.compareTo(that.text);
        }
        else{
            return this.id - that.id;
        }
    }
}
```

（3）Reducer 的实现就相对简单些了：

```
public class CommonReduce extends MapReduceBase
implements Reducer<TextPair,Text,Text,Text> {
public void reduce(TextPair key, Iterator<Text>values,OutputCollector<Text,Text> output, Reporter reporter)
throws IOException {

        String tradeId= values.next().toString();//first value is tradeID
```

```
            while( values.hasNext()){     //next is payID
                String payID = values.next().toString();                output.collect(new
Text(tradeId), new Text(payID));
            }
    }
```

（4）调用函数的配置

```
public class Join {
public static void main(String[] args) throws IOException   {
            if( args.length <3){
                System.err.println("Usage: Join <tradeTableDir>
<payTableDir> <output>");
                System.exit(-1);
            }
            String tradeTableDir = args[0];
            String payTableDir = args[1];
            String joinTableDir = args[2];
            //定义 Job
            JobConf conf = new JobConf(Join.class);
            conf.setJobName("join two tables");
            //add inputpath:    trade table && pay table
            FileInputFormat.addInputPath(conf, new Path(tradeTableDir));
            FileInputFormat.addInputPath(conf, new Path(payTableDir));
            conf.setInputFormat(TextInputFormat.class);
            conf.setMapperClass(PreMapper.class);
            conf.setMapOutputKeyClass(TextPair.class);
            conf.setPartitionerClass(KeyPartitioner.class);
            conf.setOutputValueGroupingComparator(FirstComparator.class);
            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(Text.class);
            conf.setReducerClass(CommonReduce.class);
            conf.setOutputFormat(TextOutputFormat.class);
            conf.setNumReduceTasks(1);

 //output
FileOutputFormat.setOutputPath(conf, new Path(joinTableDir));
JobClient.runJob(conf);
```

```
}
```

说明：在这个调用过程中，我们多设置了一个 Partitioner 及 comparator，这就是用来做“secondary sort”的。

（5）KeyPartitioner 的实现：在 partition 的过程中，只根据 TextPair 的 text 值进行分割，使得相同“productID”的“tradeID”、“payID”放到一起。

```
public static class KeyPartitioner
implements Partitioner<TextPair, Text>{
@Override
public int getPartition(TextPair key, Text value, int numPartitions){
        return (key.text.hashCode()& Integer.MAX_VALUE) % numPartitions;
}
@Override
public void configure(JobConf job) {}
}
```

（6）FirstComparator 的实现：只对 TextPair 对象的 text 值进行比较。

```
public static class FirstComparator extends WritableComparator{
    public FirstComparator(){     //register comparator
        super(TextPair.class, true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        TextPair o1 = (TextPair)a;
        TextPair o2 = (TextPair)b;
        return o1.text.compareTo(o2.text);
    }
}
```

根据上述步骤实现之后，我们在 conf 定义的 output 中，就可以看到类似的结果了：

```
trade1    pay1
trade2    pay2
trade2    pay4
trade3    pay3
```

InputSplit 文件格式分析

作者：一见 email：eyjian@qq.com

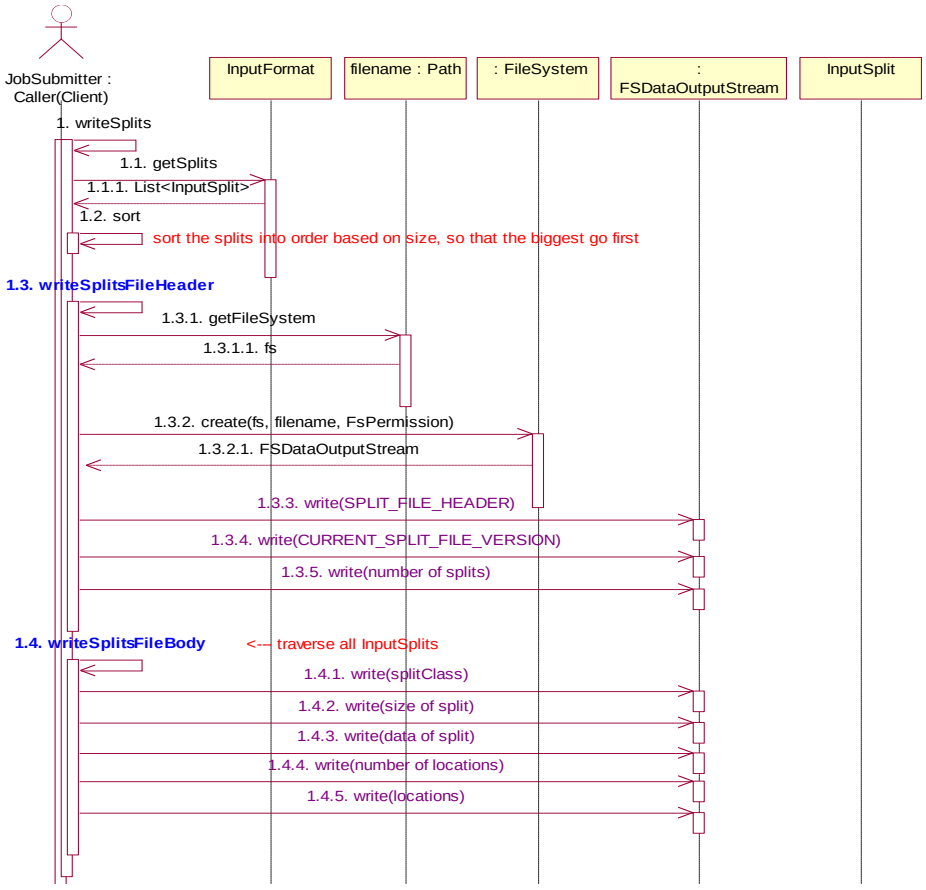
1 什么是 InputSplit 文件？

在 Hadoop 中，Block 和 Split 两个概念，其中 Block 是 HDFS 中的，而 Split 是 MapReduce 中的。HDFS 将一个文件分成若干相等字节数的 Block，也就是一个文件由若干 Block 组成，而 MapReduce 将一个任务(Job)的 Map 阶段工作划分成若干个 Split。

Split 是允许用户重写的，它的抽象接口就是 InputSplit。InputSplit 文件描绘了 MapReduce 是如何将 HDFS 的文件划分成若干 Split，通常一个 Block 对应一个 Split。通过重写 InputSplit，就可以按照自己的逻辑来划分 HDFS 文件了，甚至可以脱离 HDFS，比如计算圆周率并不需要文件。

本文将详细介绍 InputSplit 文件是如何产生的，以及它的二进制格式，所研究的版本是 Hadoop 0.21。

2 生成过程



- **过程描述：**
  （1） 通过 InputFormat，得到 splits 列表；
  （2） 按照 splits 的大小从大到小排序；
  （3） 写 split 文件头；
  （4） 写 split 文件体。

- **写 split 文件头子过程描述：**
  （1） 以二进制格式写文件头标识（一般为 SPL）；
  （2） 写入当前 split 文件版本号；
  （3） 写入 splits 个数。

- **写 split 文件体子过程描述：**
  （1） 写入 split 类名；
  （2） 写入 split 大小（字节数）；
  （3） 写入 split 数据（这个和具体的 split 相关）；
  （4） 写入 locations 个数；
  （5） 循环写入 location。

3 文件格式

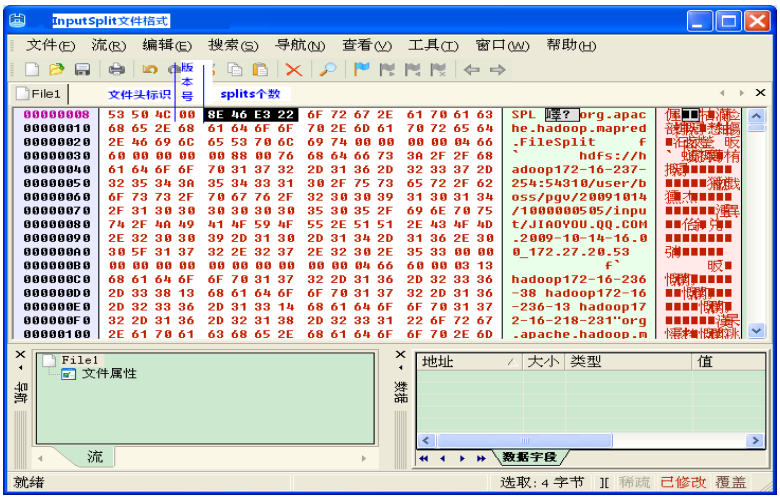
字段名称		字节数	默认值
文件头标识		3	SPL
当前 split 文件版本号		1	0
splits 个数		1-4	
split 类名长度		1-4	
split 类名		由“split 类名长度”决定	org.apache.hadoop.mapred.FileSplit
split 长度		8	
split 内容		由“split 长度”决定	和具体的 split 相关
locations 个数		1-4	
根据 locations 个数个数循环	location 长度	1-4	
	location 内容	由“location 长度”决定	

注：1~4 表示可以为 1，2，3 和 4 中的任何一个值，相关操作封装在 writableUtils.writeVLong 函数中。

4 FileInputSplit 格式

字段名称	字节数	默认值
数据文件名长度	1-4	
数据文件名	由“数据文件名长度”决定	
文件中的偏移位置	8	
大小	8	

5 文件示例



✧ 短评：HDFS、MapReduce 和 HBase 三者相辅相成、各有长处

Hadoop 上的 HDFS、MapReduce 和 HBase 可以说是 Hadoop 的三剑客，它们之间相辅相成、各有长处：

HDFS - 最大化利用磁盘

MapReduce - 最大化利用 CPU

HBase - 最大化利用内存

MapReduce 和 HBase 都将数据存储在 HDFS，而且 HBase 还利用了 MapReduce 的计算能力。

而 Pig 和 Hive 则为更高层的建筑，降低了使用 Hadoop 的门槛，提高了 Hadoop 开发的效率。ZooKeeper 和 Common 成员可以说是地基，是为上层建筑（包括高层）服务的。

Hadoop 的线性扩展性，体现在好几个方面：

（1）存储扩展性，即 HDFS 的扩展能力

（2）计算扩展性，即 MapReduce 的扩展能力，受束于计算均衡性

（3）Master 节点扩展性，主要是 Master 的处理能力和元数据存储能

（作者：一见）

HDFS 在 web 开发中的应用

作者：国宝（代志远）

互联网的应用每时每刻都在产生数据，这些数据长期的积累了长期，使得这些数据文件总量非常庞大，存储这些数据需要投入巨大的硬件资源，但是如果能在已有空闲磁盘集群下可以利用起来，可以不再需要大规模采集服务器存储数据或购买容量庞大的磁盘，减少了硬件成本。在这里就可以使用到分布式存储这种方案来解决这个问题。

HDFS 就是一个开源在 apache 上的分布式文件系统框架，它提供了命令行模式和 api 模式来操作，HDFS 文件系统部署在多台节点上后，我们可以上传任意的文件到 HDFS 中，无需关心文件究竟存储在哪个节点上。只要通过 api 访问文件即可(流操作)。

下面我们用一个实际的产品来解决，在 HDFS 中默认的块大小为 64M，也就是说一个文件在大小不超过 64M 的情况下不会被切割，整个文件会被完整的上传到某个节点中，通常我们在互联网门户中图片存储是个非常常见的应用，大量用户的图片自然占用了大量的存储空间，使用分布式文件系统正好满足相册的技术机构。每个图片一般经过压缩后最大几 M，每个图片在使用 HDFS 的也就是一个对应一个 Block。下面我们考虑如下一个问题:

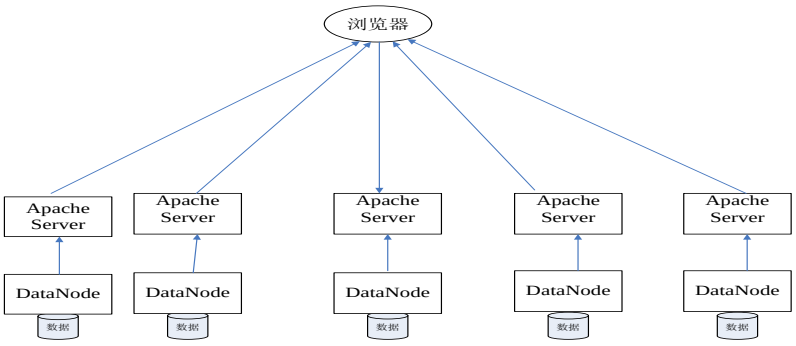
web 应用关心的是个对外的接口，这个接口只需要指定 HDFS 的管理地址，但 web 应用中每个请求访问一个图片，是否需要 web 程序通过 HDFS 将数据从不同节点上拉过来然后再传向客户端，这样是不是影响效率？如果存放数据的节点直接向浏览器发送数据是不是更好？

带着上面的问题我们就从架构开始设计一套系统。

我们可以在每台数据节点上都部署上 apache 服务器，并发布同样的 web 程序。当一个请求发现在某个节点上时，我们只需要直接将 url 重定向到节点所在的机器域名并请求同样的 uri，就可以使得当前这个节点的 apache 服务器同浏览器建立连接，并直接发送数据。

目前比较流行的 web 架构是 J2EE，在这里我们就采用 J2EE 的架构来搭建一个原型。

1 整体架构



我们拥有 100 台机器，每台机器都部署了 DataNode 节点和 Apache Web 服务器，对外提

供一个固定的域名服务器，客户通过浏览器直接访问的是该服务器，在该 server 中发布程序接受来自浏览器的请求，通过调用 HDFS 的接口，来判断请求的文件所存放的节点是否为当前节点，如果不是则根据得到的节点地址去重定向请求到存储该文件的节点的 Apache 服务器上。

由于 HDFS 中默认切割块为 64M，通常图片文件大小不过几 M，文件的存放就在一个 Block 中，如果 Block 数组大于 1，因为浏览器不支持分段获取数据，那么我们需要直接从远程将数据拉到对外服务器中合并成完整的流，然后推送到浏览器中。

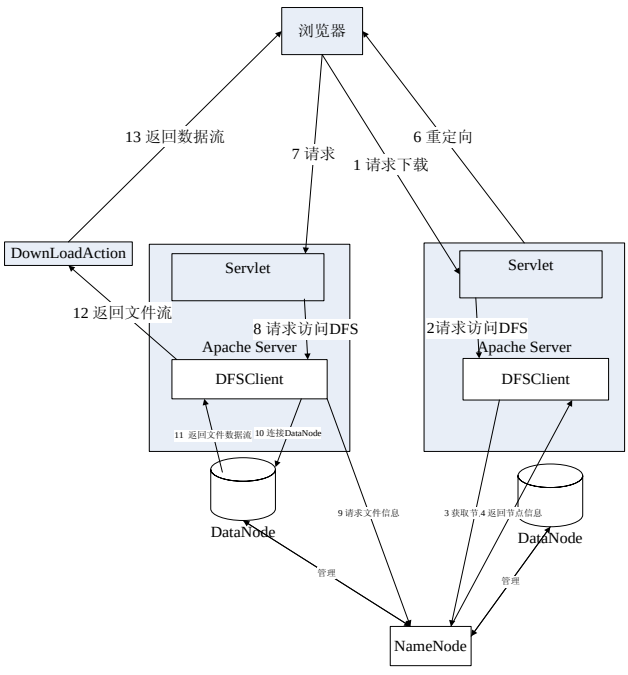
```
String filedisplay = (String) request.getParameter("filePath");
Path path = new Path(filedisplay);
Configuration conf = new Configuration();
FileSystem fs = path.getFileSystem(conf);
long length = file.getLen();
BlockLocation[] blkLocations = fs.getFileBlockLocations(file, 0, length);
```

如表1所示这里拿到到的BlockLocation包含了该块的信息：

```
private String[] hosts; //hostnames of datanodes
private String[] names; //hostname:portNumber of datanodes
private String[] topologyPaths; // full path name in network topology
private long offset;  //offset of the of the block in the file
private long length;
```

通过此可以拿到 DataNode 节点信息，让后将请求重定向到 DataNode 所在的机器的 Apache 服务器中就行。

2 结构流程



3 Action 实现

```
String filedisplay = (String) request.getParameter("filePath");
Path path = new Path(filedisplay);
Configuration conf = new Configuration();
FileSystem fs = path.getFileSystem(conf);
InputStream ins = fs.open(path);
OutputStream outp = null;
try {
    outp = response.getOutputStream();
byte[] b = new byte[1024];
    int i = 0;
while ((i = ins.read(b)) > 0) {
        outp.write(b, 0, i);
    }
    outp.flush();
} catch (Exception e) {
        e.printStackTrace();
} finally {
        if (ins != null) {
            ins.close();
            ins = null;
        }
    if (outp != null) {
        outp.close();
        outp = null;
    }
}
```

通过以上步骤，jsp 中将请求的 URI 匹配。即可获取文件的流并下载。

Mapreduce 中 value 集合的二次排序

作者：皮冰锋 email: pi.bingfeng@gmail.com

我们都知道，Hadoop 的 MapReduce 模型支持基于 key 的排序，即在一次 MapReduce 之后，结果都是按照 key 的大小排序的。但是在很多应用情况下，我们需要对映射在一个 key 下的 value 集合进行排序，即“secondary sort”。

在《hadoop the definate guide》的 P227 的“secondary sort”章节中，以<year, temperature>为例，在 map 阶段按照 year 来分发 temperature，在 reduce 阶段按照同一 year 对应的 temperature 大小排序。

本文以<String, Int>格式为例，先介绍已知类型的二次排序，再介绍泛型<key,value>集合的二次排序。

设输入为如下的序列对:

```
str1 3
str2 2
str1 1
str3 9
str2 10
```

我们期望的输出结果为:

```
str1 1,3
str2 2,10
str3 9
```

1 value 集合的二次排序

（1）先定义一个 TextInt 类，将 String 及 int 对象封装为一个整体。由于 TextInt 在 mapreduce 中要作为 key 进行比较，必须实现 WritableComparable 接口。如下:

```
public class TextInt implements WritableComparable<TextInt>{
    private String text ;
    private int value;

    public TextInt(){}

    public TextInt(String text , int value){
        this.text = text;
```

```
            this.value = value;
        }

        public String getFirst(){
            return this.text;
        }

        public int getSecond(){
            return this.value;
        }
        @Override
        public void readFields(DataInput in) throws IOException {
            text = in.readUTF();
            value = in.readInt();
        }

        @Override
        public void write(DataOutput out) throws IOException {
            out.writeUTF(text);
            out.writeInt(value);
        }

        @Override
        public int compareTo(TextInt that) {
            return this.text.compareTo(that.text);
        }
    }
```

（2）我们用KeyValueTextInputFormat的方式来读取输入文件，以tab等分割符切分输入的<key,value>对，key，value都是Text类型。所以在mapper阶段需要把将value还原到int数据，同时封装String及int为TextInt;

```
    @Override
    public void map(Text key, Text value,  OutputCollector<TextInt, IntWritable> output, Reporter reporter) throws IOException {

            int intValue = Integer.parseInt(value.toString());
            TextInt ti = new TextInt(key.toString(), intValue);
```

```
            output.collect(ti, new IntWritable(intValue));
    }
```

（3）在 reduce 阶段，为了方便查看输出数据，我们把同一个 string 对应的 int 数据封装在一起，如下：

```
    @Override
    public void reduce(TextInt key, Iterator<IntWritable> values,
                OutputCollector<Text, Text> output, Reporter reporter)
                throws IOException {
            StringBuffer combineValue = new StringBuffer();
            while( values.hasNext()){
                int value = values.next().get();
                combineValue.append(value + ",");
            }
            output.collect(new Text(key.getFirst()),new Text(combineValue.toString()));
    }
```

（4）上面的 map 及 reduce 跟普通的 mapreduce 没什么区别，很容易理解，但真正实现二次排序的是以下两个 comparator 及一个 partitioner。

1)TextIntComparator：先比较 TextInt 的 String，再比较 value;

```
    public static class TextIntComparator extends WritableComparator{
            public TextIntComparator(){
                super(TextInt.class, true);     //注册 comparator
            }

            @Override
            public int compare(WritableComparable a, WritableComparable b) {
                TextInt o1 = (TextInt)a;
                TextInt o2 = (TextInt)b;

                if ( !o1.getFirst().equals(o2.getFirst())){
                    return o1.getFirst().compareTo(o2.getFirst());
                }
                else{
                    return o1.getSecond() - o2.getSecond();
                }
            }
```

```
}
```

2)TextComparator：只比较 TextInt 中的 String。

```
public static class TextComparator extends WritableComparator{
        public TextComparator(){
            super(TextInt.class, true);
        }

        @Override
        public int compare(WritableComparable a, WritableComparable b) {
            TextInt o1 = (TextInt)a;
            TextInt o2 = (TextInt)b;
            return o1.getFirst().compareTo(o2.getFirst());
        }
}
```

3) PartitionByText：根据 TextInt 中的 String 来分割 TextInt 对象：

```
public static class PartitionByText implements Partitioner<TextInt, IntWritable>{

        @Override
        public int getPartition(TextInt key, IntWritable value,  int numPartitions) {
            return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;
        }

        @Override
        public void configure(JobConf job) {}
}
```

（5）OK，基础工作都完成了，现在看实际的 job 调用：

```
//……define input & output
          //定义 Job
          JobConf conf = new JobConf(Join.class);
          conf.setJobName("sort by value");

          //add inputpath:
          FileInputFormat.addInputPath(conf, new Path(input));
          conf.setInputFormat(KeyValueTextInputFormat.class);
          conf.setMapperClass(Mapper.class);
```

```
conf.setMapOutputKeyClass(TextInt.class);
conf.setMapOutputValueClass(IntWritable.class);

conf.setOutputKeyComparatorClass(TextIntComparator.class);
conf.setOutputValueGroupingComparator(TextComparator.class);
conf.setPartitionerClass(PartitionByText.class);

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);

conf.setReducerClass(Reduce.class);
conf.setOutputFormat(TextOutputFormat.class);

//output
FileOutputFormat.setOutputPath(conf, new Path(output));
JobClient.runJob(conf);
```

通过上面的 job 执行，在 output 的文件中，就能看到类似的输出结果了：

```
str1  1,3
str2  2,10
str3  9
```

上面介绍了如何实现，在“知其然”的基础上，我们进一步“知其所以然”。首先，我们需要把相同 string 的 int 数据分发到同一个 reducer，这就是 PartitionByText 的作用；其次，用 TextComparator 控制 reduce 阶段 int 数据集合的 group，即把相同的 string 对应的 int 数据组装在一起；最后，用 TextIntComparator 实现在 reduce 阶段的排列方式。

2 泛型 value 的二次排序

在上一个章节中，将已知类型的 key 和 value 封装成 TextInt 对象，然后做二次排序。在很多情况下，我们需要对不同类型的 key 或 value 做二次排序，或者 value 是一个二元组/多元组，实现多级的排序，这时可以使用泛型的 key 或 value。

定义泛型的 key/value 组合对象，key 和 value 均要实现 WritableComparable 接口。

```
public static class CombinedObject implements WritableComparable{     private static Configuration conf
= new Configuration();
        private Class<? extends WritableComparable> firstClass;
        private Class<? extends WritableComparable> secondClass;
```

```
    private WritableComparable first ;
    private WritableComparable second;

    public CombinedObject(){    }
    public CombinedObject(Class<? extends WritableComparable> keyClass, Class<? extends WritableComparable> valueClass){
        if (keyClass == null || valueClass == null) {
            throw new IllegalArgumentException("null valueClass");
        }
        this.firstClass = keyClass;
        this.secondClass = valueClass;
        first = ReflectionUtils.newInstance(firstClass, conf);
        second = ReflectionUtils.newInstance(secondClass, conf);
    }

    public CombinedObject(WritableComparable f, WritableComparable s){
        this(f.getClass(), s.getClass());
        setFirst(f);
        setSecond(s);
    }

    public void setFirst(WritableComparable key){
        this.first =key;
    }

    public void setSecond(WritableComparable value){
        this.second = value;
    }

    public WritableComparable getFirst(){
        return this.first;
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        String firstClassName = in.readUTF();
```

```
            String secondClassName = in.readUTF();
            firstClass = (Class<? extends WritableComparable>) WritableName.getClass(firstClassName,
conf);
            secondClass = (Class<? extends WritableComparable>)
WritableName.getClass(secondClassName, conf);
            first = ReflectionUtils.newInstance(firstClass, conf);
            second = ReflectionUtils.newInstance(secondClass, conf);
            first.readFields(in);
            second.readFields(in);
        }


        @Override
        public void write(DataOutput out) throws IOException {
            out.writeUTF(firstClass.getName());
            out.writeUTF(secondClass.getName());
            first.write(out);
            second.write(out);
        }

        @Override
        public boolean equals(Object o) {
            if( o instanceof CombinedObject){
                CombinedObject that = (CombinedObject)o;
                return that.first.equals(this.first);
            }
            return false;
        }

        @Override
        public int hashCode() {
            return first.hashCode();
        }

        @Override
        public int compareTo(Object o) {
```

```
            CombinedObject that = (CombinedObject)o;
            return that.first.compareTo(this.first);
        }
    }
```

注意:

在 Writable 接口的 write(DataOutput out)及 readFields(DataInput in)函数中，需要考虑 firstClass 及 secondClass 的序列化及反序列化。

在 comparator 接口的 equal,hashCode,compareTo 三个具体实现中，只考虑 first 的比较即可。

KeyComparator: 只比较 first

```
  public static class KeyComparator extends WritableComparator{
        public KeyComparator(){
            super(CombinedObject.class, true);
        }

        @Override
        public int compare(WritableComparable a, WritableComparable b) {
            CombinedObject o1 = (CombinedObject)a;
            CombinedObject o2 = (CombinedObject)b;
            return o1.first.compareTo(o2.first);
        }
  }
```

KeyValueComparator:比较 first 之后，再比较 second。

```
  public static class KeyValueComparator extends WritableComparator{
        public KeyValueComparator(){
            super(CombinedObject.class, true);
        }

        @Override
        public int compare(WritableComparable a, WritableComparable b) {
            CombinedObject o1 = (CombinedObject)a;
            CombinedObject o2 = (CombinedObject)b;

            if ( !o1.first.equals(o2.first)){
                return o1.first.compareTo(o2.first);
            }
```

```
            else{
                return o2.second.compareTo(o1.second);      //descend
            }
        }
    }
```

KeyPartitioner: 根据 first 来分发 CombinedObject。

```
public static class KeyPartitioner implements Partitioner<CombinedObject, Writable>{

    @Override
    public int getPartition(CombinedObject key, Writable value, int numPartitions) {
        return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;
    }

    @Override
    public void configure(JobConf job) { }
}
```

具体的调用方式:

```
job.setMapperClass(ContentMapper.class);
job.setMapOutputKeyClass(CombinedObject.class);

  // sort value list
job.setOutputKeyComparatorClass(KeyValueComparator.class);
job.setOutputValueGroupingComparator(KeyComparator.class);
job.setPartitionerClass(KeyPartitioner.class);

job.setReducerClass(SameInfoReducer.class);
```

在这个章节介绍了对泛型的key和value的二次排序,可以设计更多组合形式的<key,value>的排序，也可以自定义多元组构成value，实现更多灵活的排序方式。

Hive SQL手册翻译

作者：一见 余姝丹（苹果小巫女）

1 创建表和删除表

创建表语法:

```
   CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
     [(col_name data_type [COMMENT col_comment], ...)]
     [COMMENT table_comment]
     [PARTITIONED BY (col_name data_type [col_comment], col_name data_type
[COMMENT col_comment], ...)]
     [CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name, ...)] INTO
num_buckets BUCKETS]
     [ROW FORMAT row_format]
     [STORED AS file_format]
     [LOCATION hdfs_path]
     [AS select_statement]  (Note: this feature is only available on the latest
trunk or versions higher than 0.4.0.)

   CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
     LIKE existing_table_name
     [LOCATION hdfs_path]

   data_type
     : primitive_type
     | array_type
     | map_type

   primitive_type
     : TINYINT
     | SMALLINT
     | INT
     | BIGINT
     | BOOLEAN
```

```
    | FLOAT
    | DOUBLE
    | STRING

  array_type
    : ARRAY < primitive_type >

  map_type
    : MAP < primitive_type, primitive_type >

  row_format
    : DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char]
        [MAP KEYS TERMINATED BY char]
    | SERDE serde_name [WITH SERDEPROPERTIES property_name=property_value,
property_name=property_value, ...]

  file_format:
    : SEQUENCEFILE
    | TEXTFILE
    | INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname
```

“CREATE TABLE”语句根据指定的表名创建一个表。如果同名的表或表视图已经存在，则报错，但可以使用“IF NOT EXISTS ”跳过这个错误。

如果指定了“EXTERNAL”关键字，并指定了“LOCATION ”，就可以让 Hive 指定的位置，而不是默认的位置创建一个表，当创建表时，如果数据已经存在，这个功能是非常有用的。当删除一个“EXTERNAL”表时，表中的数据并不会被删除。

“LIKE”格式的“CREATE TABLE”允许复制一个已存在表的定义，而不用复制它的数据。

可以使用定制的 SerDe 或自带的 SerDe 创建表。如果没有指定“ROW FORMAT”或指定了“ROW FORMAT DELIMITED”，就使用自带的 SerDe。请参考用户指南中“SerDe”一节了解更多信息。

使用自带的 SerDe，必须指定字段列表。字段类型，请参考用户指南的类型部分。定制的 SerDe 的字段列表可以是指定的，但是 Hive 将通过查询 SerDe 决定实际的字段列表。

如果数据需要存储为纯文本文件，请使用“STORED AS TEXTFILE”。如果数据需要压缩，请使用“STORED AS SEQUENCEFILE”。如果打算在 Hive 表中存储压缩的数据，请参考 CompressedStorage。“INPUTFORMAT”和“OUTPUTFORMAT”定义一个 InputFormat 和 OutputFormat 类相应的名字，作为一个字符串。例如:

“org.apache.hadoop.hive.contrib.fileformat.base64”定义为“Base64TextInputFormat”。

“PARTITIONED BY”语句创建分区表格。一个表格可以有一个或多个分区的列，在分区的列中，每个不同值的组合直接创建一个独立的数据。根据某些列的值来确定相应元组的存放地址。进一步说，通过“CLUSTERED BY”列来聚集表或者分区， “SORT BY”列来存储数据，提高查询性能。

表名和列名不区分大小写，但 SerDe 和属性名是区分大小写的。表和列的注释分别是用单引号的字符串。

在选择创建表（CTAS）的语句中，通过查询结果创建和增加表。用 CTAS 创建的表是原始表，即所有的查询结果结束前，其他的用户是不可见的。所以，其他用户要么看到完整的查询结果要么看不到表。

CTAS 有两部分: HiveQL 支持所有的 SELECT 语句;CTAS 的 CREATE 部分从 SELECT 中生成架构和用其它表的属性来创建目标表，例如 SerDe 和存储格式化。CTAS 唯一的限制是目标表不能是一个分区表或者外部表。

例如:

一个创建表的例子:

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User')
 COMMENT 'This is the page view table'
 PARTITIONED BY(dt STRING, country STRING)
 STORED AS SEQUENCEFILE;
```

该语句创建 page_view 表， 包括 viewTime, userid, page_url, referrer_url, and ip 列及注释。在序列文件中分区表,存储数据。用 ctrl+A 分隔字段和用换行分隔行来定义文件中的数据格式。

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User')
 COMMENT 'This is the page view table'
 PARTITIONED BY(dt STRING, country STRING)
 ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '\001'
STORED AS SEQUENCEFILE;
```

该语句用于创建和上表相同的表

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User')
 COMMENT 'This is the page view table'
 PARTITIONED BY(dt STRING, country STRING)
```

```
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\001'
  COLLECTION ITEMS TERMINATED BY '\002'
  MAP KEYS TERMINATED BY '\003'
STORED AS SEQUENCEFILE;
```

在上述示例中，集群 userid 追加到 page_view 表，每个追加数据按照 viewTime 顺序存储。允许用户通过 userid 有效的对集群列进行采样。存储属性使内部操作者在评估查询的同时，更好的了解数据结构，同时提高了效率。如果列是列表或地图可以使用“MAP KEYS” 和“COLLECTION ITEMS”关键字。

到目前为止的所有的例子中，数据都存储在“<hive.metastore.warehouse.dir> / page_view”。在 Hivi 配置文件 hive-site.xml 中 hive.metastore.warehouse.dir 一个指定的值。

```
CREATE EXTERNAL TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User',
    country STRING COMMENT 'country of origination')
COMMENT 'This is the staging page view table'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\054'
STORED AS TEXTFILE
LOCATION '<hdfs_location>';
```

用上述语句创建一个 page_view 表，存储在任何 hdfs 本地文件中。但是必须确保上面查询数据是分隔开的。

```
CREATE TABLE new_key_value_store
  ROW FORMAT SERDE "org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe"
  STORED AS RCFile AS
SELECT (key % 1024) new_key, concat(key, value) key_value_pair
FROM key_value_store
SORT BY new_key, key_value_pair;
```

该 CTAS 语句通过 SELECT 语句的结果(new_key DOUBLE, key_value_pair STRING)架构创建 new_key_value_store 目标表。如果 SELECT 语句不指定列的别名，列的名字将自动分配为 to _col0， _col1， 和 _col2 等。另外，在选择语句中，通过指定的 SerDe 和独立于原表的存储格式来创建一个新的目标表

```
CREATE TABLE empty_key_value_store
LIKE key_value_store;
```

相反，该语句创建一个新表 empty_key_value_store，比起其它表在细节上严格匹配现有的

key_value_store。新表不包含任何行。

2 数据插入到追加表

“CLUSTER BY ”和“SORTED BY”创建命令不会影响数据如何插入到表中，只是影响数据如何读取。这意味着用户必须注意正确插入数据，即减少的数量等于增加的数量，并用”CLUSTER BY“和“SORT BY”命令查询。

一个创建和填充追加表的例子。

3 删除表

```
DROP TABLE table_name
```

“DROP TABLE“删除表的元数据和数据。如果配置了 Trash，实际上这些数据将删除到.Trash/Current 目录。元数据将完全丢失。

当删除一个“EXTERNAL”表，表中的数据不会从文件系统中删除。

当删除视图引用的表，将没有警告，视图视为无效悬挂，必须被用户删除或者重新创建。

如何删除分区看下一节“ALTER TABLE”

4 修改表语句

“ALTER TABLE”语句用于改变一个已经存在表的结构，增加列或者分区，改变 serde，添加表和 serde 的属性，或重命名表。

5 增加分区

```
ALTER TABLE table_name ADD partition_spec [ LOCATION 'location1' ]
partition_spec [ LOCATION 'location2' ] ...

partition_spec:
  : PARTITION (partition_col = partition_col_value, partition_col =
partiton_col_value, ...)
```

“ALTER TABLE ADD PARTITION”增加表的分区。分区值只有是字符串，才被引用。

```
ALTER TABLE page_view ADD PARTITION (dt='2008-08-08', country='us') location
'/path/to/us/part080808' PARTITION (dt='2008-08-09', country='us') location
'/path/to/us/part080809';
```

6 删除分区

```
ALTER TABLE table_name DROP partition_spec, partition_spec,...
```

“ALTER TABLE DROP PARTITION”删除表的分区，将删除分区的数据和元数据。

```
ALTER TABLE page_view DROP PARTITION (dt='2008-08-08', country='us');
```

7 重命名表

```
ALTER TABLE table_name RENAME TO new_table_name
```

该语句改变表的名字，数据的位置和分区的名字不改变。总之，旧的名字不存在，表格写入将改变重命名表的数据

8 改变列名字/类型/位置/注释

```
ALTER TABLE table_name CHANGE [COLUMN] col_old_name col_new_name column_type [COMMENT col_comment] [FIRST|AFTER column_name]
```

该命令允许用户改变列的名字，数据类型，注释，或位置，或其中任意组合。

例如：CREATE TABLE test_change (a int, b int, c int);

"ALTER TABLE CHANGE a a1 INT;" 将列 a 的名字改成 a1.

"ALTER TABLE CHANGE a a1 STRING AFTER b;" 将改列 a 的名字成 a1.a 的数据类型为字符型，并放到列 b 后面。新的表的结构是：b int, a1 string, c int.

"ALTER TABLE CHANGE b b1 INT FIRST;" 将改列 b 的名字为 b1，并作为第一列。新表的结构： b1 int, a string, c int.

注意：列改变命令只是修改 Hive 的元数据，不会触及到数据。用户应该确定实际数据的分布符合元数据的定义。

9 增加/更新列

```
ALTER TABLE table_name ADD|REPLACE COLUMNS (col_name data_type [COMMENT col_comment], ...)
```

“ADD COLUMNS” 在现有列的结尾增加新列，但在分区列前。

“REPLACE COLUMNS” 删除所有现有的列，并增加列新的设置. 这只能对原表进行操作。原表是用“DynamicSerDe” 或 “MetadataTypedColumnsetSerDe serdes”创建.参照用户指南 SerDe 章节了解更多信息。

10 增加表属性

```
ALTER TABLE table_name SET TBLPROPERTIES table_properties

table_properties:
  : (property_key = property_value, property_key = property_value, ... )
```

该语句增加表的元数据。目前 last_modified_user, last_modified_time 的属性是 Hive 自动增加和管理的，Hive 用户可以在列表中增加自己的属性。通过“DESCRIBE EXTENDED TABLE”得到这些信息。

11 增加 Serde 属性

```
ALTER TABLE table_name SET SERDE serde_class_name [WITH SERDEPROPERTIES serde_properties]
ALTER TABLE table_name SET SERDEPROPERTIES serde_properties

serde_properties:
  : (property_key = property_value, property_key = property_value, ... )
```

该语句增加用户定义的元数据到表 SerDe 对象，Hive 初始化 serde 属性为序列化和反序列化数据时，表的 SerDe 通过 serde 属性。所以用户可以存储客户 serde 要求的任何信息。参照用户指南 SerDe 章节了解更多信息。

12 改变表文件格式和组织

```
ALTER TABLE table_name SET FILEFORMAT file_format
ALTER TABLE table_name CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name, ...)] INTO num_buckets BUCKETS
```

该语句改变表的物理存储属性，使用 file_format 选项，请参阅上文关于 CREATE TABLE 部分。

注意：这些命令只是修改 Hive 的元数据，不能重组或格式化现有的数据。用户应该确定实际数据的分布符合元数据的定义。

13 创建/删除视图

注意：只有 Hive 0.6 开始支持视图

创建表视图

```
CREATE VIEW [IF NOT EXISTS] view_name [(column_name [COMMENT column_comment], ...) ]
[COMMENT view_comment]
AS SELECT ...
```

“CREATE VIEW”以指定的名字创建一个表视图。如果表或视图的名字已经存在，则报错，可以使用“IF NOT EXISTS”跳过这个错误。

如果没有提供表名，视图列的名字将由定义的 SELECT 表达式自动生成，如果 SELECT 包括如 x + y 无标量的表达式，视图列的名字将生成_C0，_C1 等形式。当重命名列可选择性提供列注释。注释不会从底层列自动继承。

如果定义 SELECT 表达式的视图是无效的，CREATE VIEW 语句将失败。

注意，没有关联存储的视图是纯粹的逻辑对象。在 Hive 中目前不支持具体化视图。当一个查询引用一个视图，评估视图的定义为了进一步为查询提供行集。这是一种概念的描述，实际上，作为查询优化的一部分，Hive 可以将视图的定义与查询的定义结合起来，例如从查询到视图使用过滤器。

在视图创建的同时视图的架构确定了，随后改变基本表（如添加一列）将不会在视图的架构体现。如果基本表被删除或以不兼容的方式被修改，该无效视图的查询将失败。

视图是只读的，不能用于”LOAD/INSERT/ALTER”的目标。

视图可能包含“ORDER BY” 和 “LIMIT”子句，如果一个参考查询也包含了这些子句，在视图子句和任何其它查询操作后评估查询级子句。例如，如果一个视图指定极限为 5，执行查询语句：select * from v LIMIT 10，最多返回 5 行。

创建视图的例子

```
CREATE VIEW onion_referrers(url COMMENT 'URL of Referring page')
COMMENT 'Referrers to The Onion website'
AS
SELECT DISTINCT referrer_url
FROM page_view
WHERE page_url='http://www.theonion.com';
```

删除表视图

```
DROP VIEW view_name
```

“DROP VIEW”删除指定视图的元数据，在视图中使用“DROP TABLE”是错误的。

例如：

```
DROP VIEW onion_referrers;
```

14 创建/删除函数

创建函数

```
CREATE TEMPORARY FUNCTION function_name AS class_name
```

该语句创建一个由类名实现的函数。在 Hive 中可以持续使用该函数查询。可以使用 Hive 类路径中的任何类。通过执行“ADD FILES”语句添加到类路径。请参阅用户指南 CLI 部分了解有关在 Hive 中如何添加/删除的更多信息。使用该语句注册用户定义函数（UDF’s）。

删除函数

注销用户定义函数如下:

```
DROP TEMPORARY FUNCTION function_name
```

15 显示/描述语句

在 Hive 系统中，该语句提供一种方法对现有的数据和元数据的 Hive 元存储的查询。

显示表

```
SHOW TABLES identifier_with_wildcards
```

SHOW TABLES 列出了所有的基表和给定的相匹配正规表达式名字的视图。正规表达式只能包含'*'作为任意字符[s]或'|'作为选择。例如'page_view'，'page_v *'，'*view|page*'，所有这些将匹配'page_view'表。匹配表按字母顺序排列。在元存储中，如果没有找到匹配的表并不是一个错误。

显示分区

```
SHOW PARTITIONS table_name
```

“SHOW PARTITIONS”列出了给定基表中的所有现有分区。分区按字母顺序排列。

显示表/分区扩展

```
SHOW TABLE EXTENDED [IN|FROM database_name] LIKE identifier_with_wildcards
[PARTITION(partition_desc)]
```

“SHOW TABLE EXTENDED”列出所有给定的匹配正规表达式表的信息。如果一个分区规范存在，用户不能使用正规表达式作为表名。该命令的输出包括基本表信息和文件系统信息例如，文件总数，文件总大小，最大文件大小，最小文件大小，最新存储时间和最新更新时间。如果分区存在，它会输出给定的分区的文件系统信息，而不是表中的文件系统信息。

作为视图，“SHOW TABLE EXTENDED”用于检索视图的定义。提供两个相关的属性:用户指定原视图定义和 Hive 内部使用的扩展定义。

显示函数

```
SHOW FUNCTIONS "a.*"
```

“SHOW FUNCTIONS”列出用户定义和建立所有匹配给定的正规表达式的函数。给所有函数用".*"。

描述表/列

```
    DESCRIBE [EXTENDED] table_name[DOT col_name]
   DESCRIBE [EXTENDED] table_name[DOT col_name ( [DOT field_name] | [DOT '$elem$']
| [DOT '$key$'] | [DOT '$value$'] )* ]
```

“DESCRIBE TABLE”显示列的列表，包括给定表的分区列。如果“EXTENDED”关键字指定，将在序列化形式中显示表的所有元数据。通常只用于调试，而不用于平常使用。

如果表有复杂的列，可以通过指定数组元素 table_name.complex_col_name（和'$ elem $'作为数组元素，'$key$'为图的关键，'$value$'为图的属性）来检查该列的属性 。为探讨复杂列类型可指定该递归。

描述分区

```
   DESCRIBE [EXTENDED] table_name partition_spec
```

该语句列出了给定分区的元数据。输出和“DESCRIBE TABLE”类似。目前，当准备工作时不能使用列信息。

Mahout Kmeans 简介

作者： Tony Cui QQ： 20104485

1 走近 Mahout

Mahout 是 apache 组织下，基于 hadoop 的分布式数据挖掘开源项目，目前的最新版本是 0.2 版，已经包括很多主流的数据挖掘算法，如本文将要介绍的 k-means 等。一些高级的数据挖掘算法已经在开发之中，如 SVM，随即森林等，将会发布在接下来的 0.3，0.4 版本，感兴趣的同学可以到 apache 的开发者社区跟进相应的项目。

众所周知，并不是所有的数据挖掘算法都可以并行化，有的算法理论上就不可以并行化，有的直接并行化存在很大的效率问题。简单如 k-means 聚类算法，在并行化的过程中就需要做相应的更改，因此，我在这里先简单介绍下 k-means 的并行化算法。

预先知识：了解 k-means 和 canopy 聚类算法

K-means 并行化策略，共 2 步：

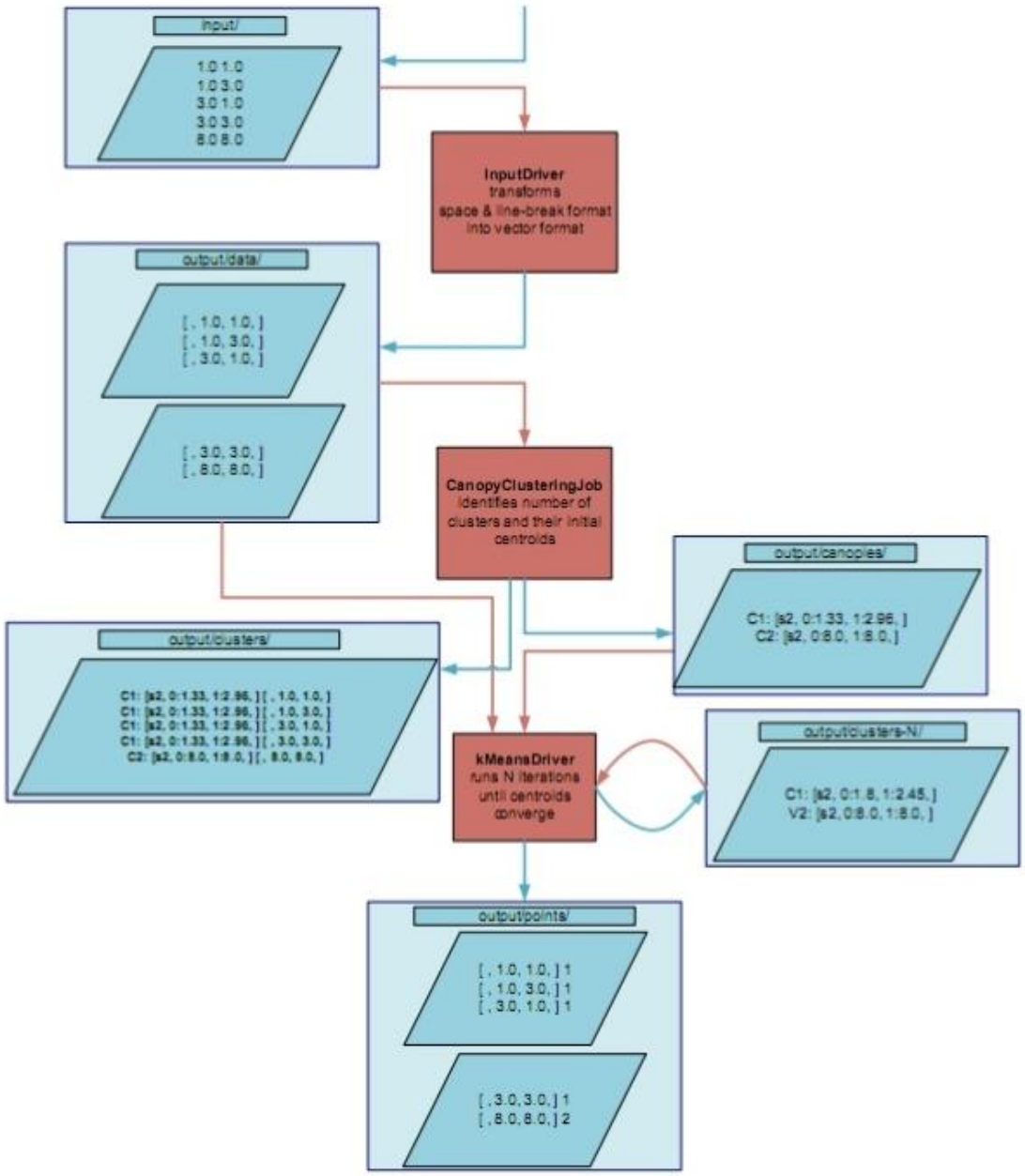
先进行 canopy 聚类，将初始数据聚类成 K 类。Canopy 是一个快速聚类算法，不是很精确，但是算法简单，计算量很小，因此速度非常快

K-means 聚类，在循环计算中心点时，不再计算中心点到所有点的距离，取而代之的是根据 canopy 聚类的结果，计算中心点到所属类的所有的点的距离，取均值，得到下一个循环的中心点。 这样，在 canopy 聚类之后得到的 K 个类就可以分布到各个计算节点，在这些节点计算 K-means 的中心点，最后返回这 K 个中心点，所有数据再计算一遍到各个中心点的距离即可得到聚类结果。

Mahout 中的 k-means 聚类也是依据此策略实现：

```
// 先 canopy 聚类，testdata/point 是 input data, testdata/canopies 是 output data
CanopyDriver.runJob("testdata/points", "testdata/canopies"
ManhattanDistanceMeasure.class.getName(), (float) 3.1, (float) 2.1, "dist/apache-mahout-0.1-dev.jar");
// 再进行 kmeans 聚类，input data 为2部分，一个是 testdata/points,一个是 testdata/canopies
KMeansDriver.runJob("testdata/points", "testdata/canopies",
"output",EuclideanDistanceMeasure.class.getName(), "0.001", "10");
```

可参考此官方提供的流程图：



注：此并行化策略为参考如下视频 【需要翻墙哦~】

http://code.google.com/edu/content/submissions/mapreduce-minilecture/listing.html

2 Mahout k-means 实例

（1）下载数据文件

http://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series.

此文件名为：synthetic_control.data

（2）将其上传到 hdfs

```
[@master tempdata]$ hadoop fs -ls testdata
Found 1 items
-rw-r--r--   1 hadoop supergroup     288374 2010-02-18 14:38 /user/hadoop/testdata/synthetic_control.data
```

（3）运行编译好的 example

所有参数:

```
[@master target]$ hadoop jar mahout-examples-0.3-SNAPSHOT.job org.apache.mahout.clustering.syntheticcontrol.kmeans.Job --help
Usage:
 [--input <input> --output <output> --distance <distance> --convergenceDelta
<convergenceDelta> --maxIter <maxIter> --vectorClass <vectorClass> --t1 <t1>
--t2 <t2> --help]
Options
  --input (-i) input                        Path to job input directory
  --output (-o) output                      The directory pathname for
                                            output.
  --distance (-m) distance                  The Distance Measure to use.
                                            Default is SquaredEuclidean
  --convergenceDelta (-v) convergenceDelta  The convergence delta value.
  --maxIter (-x) maxIter                    The maximum number of iterations.
  --vectorClass (-v) vectorClass            The Vector implementation class
                                            name.  Default is
                                            RandomAccessSparseVector.class
  --t1 (-m) t1                              The t1 value to use.
  --t2 (-m) t2                              The t2 value to use.
  --help (-h)                               Print out help
[@master target]$
```

org.apache.mahout.clustering.syntheticcontrol.kmeans.Job 这个类中没有提供-k 参数，因为聚成几个类是有 canopy 聚类决定的，上图中的 t1,t2 这两个参数就是 canopy 的输入参数。本例中使用默认参数

Command line:

```
Hadoop jar mahout-examples-0.3-SNAPSHOT.job org.apache.mahout.clustering.syntheticcontrol.kmeans.Job -i testdata -o output
```

（4）结果文件

```
[@master target]$
[@master target]$ hadoop fs -ls output
Found 13 items
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:46 /user/hadoop/output/canopies
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:46 /user/hadoop/output/clusters-0
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:47 /user/hadoop/output/clusters-1
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:47 /user/hadoop/output/clusters-2
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:48 /user/hadoop/output/clusters-3
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:48 /user/hadoop/output/clusters-4
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:48 /user/hadoop/output/clusters-5
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:49 /user/hadoop/output/clusters-6
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:49 /user/hadoop/output/clusters-7
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:50 /user/hadoop/output/clusters-8
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:50 /user/hadoop/output/clusters-9
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:46 /user/hadoop/output/data
drwxr-xr-x   - hadoop supergroup          0 2010-02-24 11:51 /user/hadoop/output/points
[@master target]$
[@master target]$
```

本次 k-means 共循环了 10 次，每次循环的结果存放在 clusters-X 文件夹中。因此，最终的中心点可查看 clutsters-9 文件，聚类结果文件村存放在 points 文件中。

Mahout 中 文件都是使用的 seqFile(sequenceFile)格式，seqFile 格式是一个 hadoop 类，支持存储任意格式的键值对。

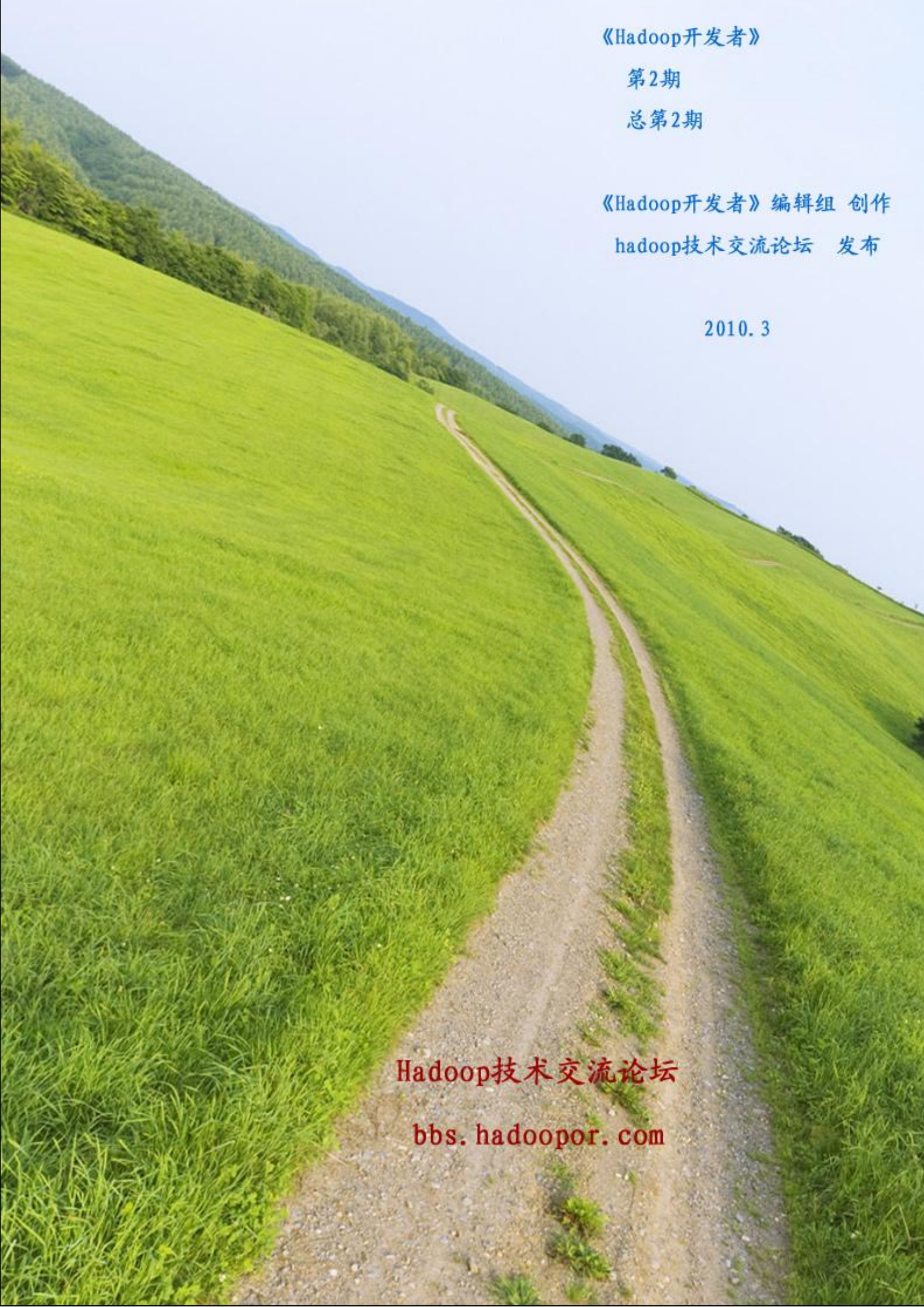
将结果文件导入本地后，转化成为普通文本格式:

```
sh bin/mahout seqdump -s output/clusters-9/part-00000  -o part-0
sh bin/mahout seqdump -s output/points/part-00001  -o part-1
```

part-0 中共有 7 个点，这些点即为中心点。

part-1 中有 300 多个点，每个点对应的 value 值就是所属的分类。其余 300 多个点的数据在 points/part-00000 中，读者可以自行导出。

好了，本文到现在就结束了，希望对大家有所帮助，谢谢。



《Hadoop开发者》

第2期

总第2期

《Hadoop开发者》编辑组 创作

hadoop技术交流论坛 发布

2010.3

Hadoop技术交流论坛

bbs.hadoopor.com