Hadoop开发者

2011年第一期　总第四期

海量数据处理平台架构演变

计算不均衡问题在Hive中的解决办法

ZooKeeper权限管理机制

ZooKeeper服务器工作原理和流程

Hadoop最佳实践

Hadoop类tar命令的实现

《Hadoop开发者》编辑组

《Hadoop 开发者》第四期

分享互助，欢迎投稿

出品

Hadoop 技术论坛

网址

http://www.hadoopor.com

本期主编

何忠育（Spork）

编辑

皮冰锋（若冰）易剑（一见）

贺湘辉（小米）王磊（beyi）

代志远（国宝）柏传杰（飞鸿雪泥）

何忠育（Spork）秘中凯

美工/封面设计

何忠育（Spork）

投稿信箱

hadoopor@foxmail.com

刊首语

《Hadoop 开发者》第四期，在千呼万呼中，终于艰难的出来。这是众多 Hadoopor 期望的一期，也是相对成熟的一期，本期的作者大多都具备在一线的 Hadoop 开发或应用经验，因此实践性较强。

在这里，我要特别感谢所有无私分享经验的作者们，没有你们的支持和奉献就不可能有《Hadoop 开发者》第四期。

本期排版工作全靠何忠育（Spork）独立担当，在他的细心下，《Hadoop 开发者》第四期才得以与大家见面。

《Hadoop 开发者》第四期的诞生是一个艰辛的过程，鲜有人乐意主动撰稿，就好比论坛里，常有人发帖求助，但少有人主动提供帮助。在征集到一期的稿件之后，又遇到了编辑、排版和审核的问题，大家都很忙，所以我要特别感谢《Hadoop 开发者》团队成员中的 Spork 同学主动跳出来担当了排版工作，也要非常感谢皮冰锋（若冰）同学一字一字地审核每篇文章，并将发现的问题逐一标出来。

虽然我们仍很业余，但不管怎样，《Hadoop 开发者》第四期出来了，问题虽然很多，但仍希望可以给每一位 Hadoopor 带来一丝帮助，更希望有更多的技术爱好者加入分享的行列、开源的行列。

Hadoop 技术论坛站长：一见

目 录

mooon

一见*

mooon 取名为“飞越”或“飞月”的意思，也可叫“非月”，但非 moon。在 2009 年，我对 Hadoop mapreduce 源代码进行了一段时间的系统化分析，在这个过程中，发觉 mapreduce 存在两大问题：数据倾斜和并行调度，并探索出一些解决方案。有点想将自己的想法付诸实践，但重实现一个 mapreduce 的工作量是非常大的，而且还依赖于分布式文件系统。我决定动手去做一些工作，但我不想仅仅奔着这个目标而来，而是希望每一点都能做到尽可能的多复用，按层划分是一个比较好的主意。

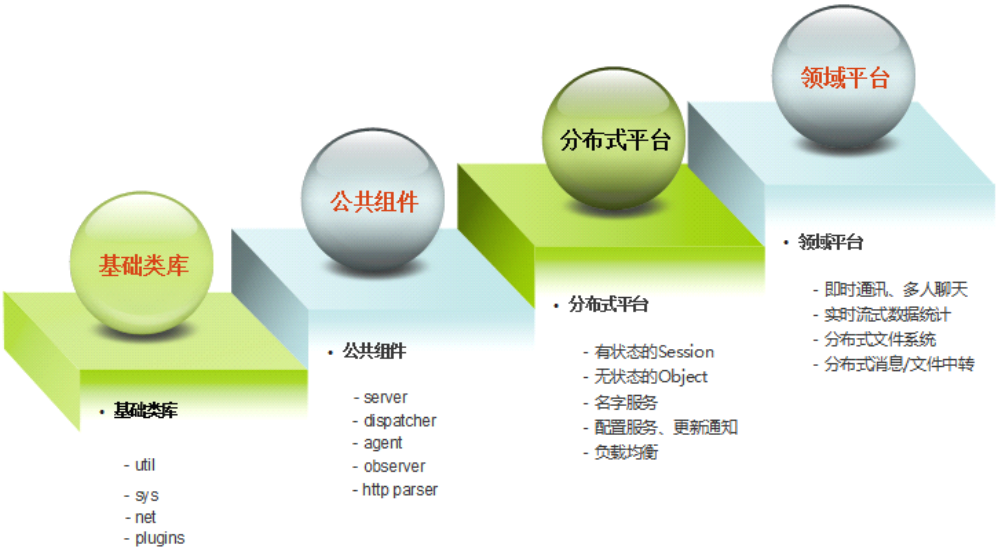
mooon 中的每一点每一步都结合了我近 10 年来的开发实践，特别是多年的分布系式统开发经验，但 mooon 不会参照任何一个现存的系统去做，而是由目标驱动。在这过程中会利用一些开源，并以独立的形式存在，如 plugin_tinyxml 方式，尽量保持第三方代码的独立性，这即是对他人劳动成果的尊重，也是避免系统走向臃肿的必要举措。

本文将分四点对 mooon 做一个简单介绍，希望能对您了解 mooon 起到一点帮助作用：

一、 优势和特点

作者简介：易剑，零二年毕业于湘潭工学院，曾就职于长沙创智、珠海金山和深圳华为。工作前半年的时间主要从事 VC/Delphi 开发，后转入 Linux/C++开发。钟情于软件技术，多年不减，在 2009 年发起开源项目“飞月”。擅长软件架构设计，代码编写严谨，重视软件的可测试性、可观察性和可运营，重视代码的用户体验。掌握方法重要，领悟思想方为根本，超越面向对象和设计模式等方法，“简单”才是最为精髓的思想。
联系方式：eyjian at qq.com

二、 分层结构



三、 基础类库

util	sys	net	plugin_mysql	plugin_tinyxml
提供与系统调用无关的工具类： ①字符串操作 ②位操作 ③可超时对象和超时管理 ④柱状图数组	提供与系统调用相关的工具类，但不涉及网络： ①共享库加载 ②共享内存 ③线程和线程池 ④文件工具 ⑤内存池和对象池 ⑥各类锁 ⑦日期时间 ⑧日志 ⑨事件队列 ⑩系统资源系统	提供与网络相关的工具类： ①可Epoll的队列 ②各类Socket封装 ③兼容IPV4和IPV6 ④字节序转换 ⑤取本地IP地址	基于MySQL的数据库连接和连接池的实现。	基于TinyXML的配置读取功能实现，支持以数组方式按层次读取配置。

四、 公共组件

server	dispatcher	http parser	agent	observer
高性能服务端框架： ①大并发下的高性能 ②抽象了包的解析和处理 ③将配置抽象为接口，不带配置文件	大并发客户端组件： ①支持消息和文件 ②支持可容灾的路由表方式 ③支持即取即用	HTTP协议解析器： ①高效解析 ②结构简单 ③易于使用 ④请求包 ⑤响应包	通用代理实现： ①提供实时的系统资源接口，如CPU、内存和流量等 ②支持连接到指定的Master或Center ③定时心跳 ④内置配置更新处理 ⑤支持可再编程，以支持Master下发的自定义命令字 ⑥提供运行状态数据上报Master接口	观察者，主动采用运行时状态数据： ①定时采集数据

五、 分布式平台

Mooon 的源代码放在 GoogleCode 网站上，可通过 SVN 下载，或直接在浏览器上查看，网址是：http://code.google.com/p/mooon。同时，我也会在 http://www.hadoopor.com 上输出 mooon 的一些情况。

海量数据处理平台架构演变

覃武权*

新入职的小 Q 懵懵懂懂，误打误撞踏上了数据分析的康庄大道，上班第一天，听说自己的导师（王 sir）是鼎鼎大名的数据分析王、业界泰斗，鸡冻不已，欣喜之情溢于言表。

王 sir 果然是位大牛，大会小会开个不停。小 Q 来了一上午，只和王 sir 打了个照面，就再没见着他的影子。刚来也没人指导，小 Q 有点不知所措，于是，翻开自己带来的那本破旧的互联网数据分析专业书，温习下基础知识：

一般网站把用户的访问行为记录以 apach 日志的形式记录下来了，这些日志中包含了下面一些关键字段：

client_ip

user_id

access_time

url

referer

status

page_size

agent

因为需要统一对数据进行离线计算，所以常常把它们全部移到同一个地方。

简单算了一下：

（1） 网站请求数：1kw/天

（2） 每天日志大小：450Byte/行 * 1kw = 4.2G，

（3） 日志存储周期：2 年

一天产生 4.5G 的日志，2 年需要 4.2G * 2 * 365 = 3.0T

为了方便系统命令查看日志，不压缩，总共需要 3.0T 的空间，刚好有一些 2U 的服务器，每台共 1T 的磁盘空间，为了避免系统盘坏掉影响服务器使用，对系统盘做了 raid1；为了避免其他存放数据的盘坏掉导致数据无法恢复，对剩下的盘做了 raid5。做完 raid 后，除去系统盘的空间，每台服务器你大概还有 800G 可以用于存储数据。先装满一台，再顺序装下一台，机器不会有浪费，可以满足需要，先放到这里来吧。于是所有的数据都汇聚到这几台 LogBackup 服务器上来了。

数据从四个地区的 Collector 服务器上，跨 IDC 传输到 LogBackup 服务器上，因为刚起步，你偷了个懒，直接使用 rsync 进行传输，把传输模块的开发也给省下了。

有了 LogBackup 服务器，离线统计就可以全部在这些服务器上进行了。在这套架构上，用 wc、

作者简介：jamesqin(覃武权)，负责各种运营支撑和管理平台的架构及开发，致力于运维支撑体系的数据化、自动化、流程化建设。
联系方式：jamesqin at viq.qq.com

grep、sort、uniq、awk、sed 等系统命令，完成了很多的统计需求，比如统计访问频率较高的 client_ip，某个新上线的的页面的 referer 主要是哪些网站。

嗯，不错，老大如果问起这个网站的一些数据，回答起来绝对是游刃有余。^_^

看书看得小有成就的小 Q 暗自窃喜，这时候王 sir 走过来关心下徒弟，小 Q 一激动，就把刚学的东东向王 sir 汇报了一番。王 sir 边听边点点头，称赞小 Q 懂的还真不少啊！“如果你的网站数据量再翻 10 倍，达到日志总行数 1 亿/天，这个架构还能支撑吗？”“这个，这……”突然一问，问懵了小 Q，露馅了不是？小 Q 赶紧认了，“这个还真不知道，求师傅详解。”

王 sir 看这徒弟如此积极好学，心里很是安慰，拿着笔在小 Q 的笔记本上边划边耐心讲道。

当业务的迅猛发展，网站流量爆发增长，产品经理如果想从中获取更多的用户特征和用户信息，就需要我们这些数据分析人员从不同的日志中找到令他们满意的答案。如果

（1） 日志总行数：1 亿/天

（2） 每天日志大小：450Byte/行 * 1 亿 = 42G，

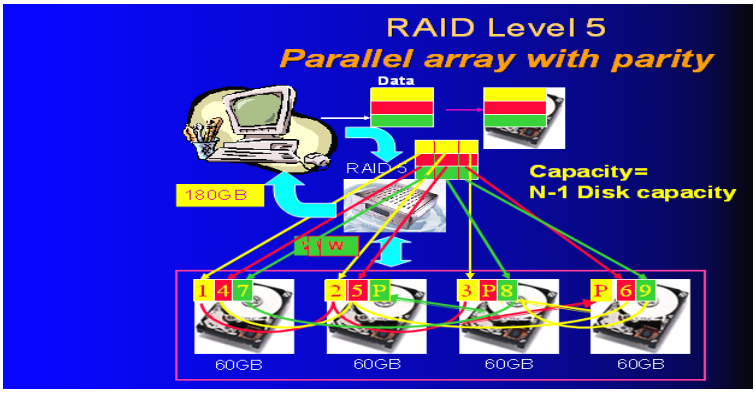
（3） 日志种类：5 种

那么之前采用的 LogBackup 服务器就会出现短板，虽然 LogBackup 服务器不会有空间不足的风险，但是它这样单机独立存储，在一堆数据之中执行一次 grep，都需要等上几分钟，串行操作直接导致性能瓶颈。这时候细心观察 LogBackup 服务器上的 cpu 利用率数据，就会发现日志存储服务器大部分的时间都是闲置状态，而一些临时的 linux 命令或脚本运行的时候，cpu 利用率也不高，如下图：

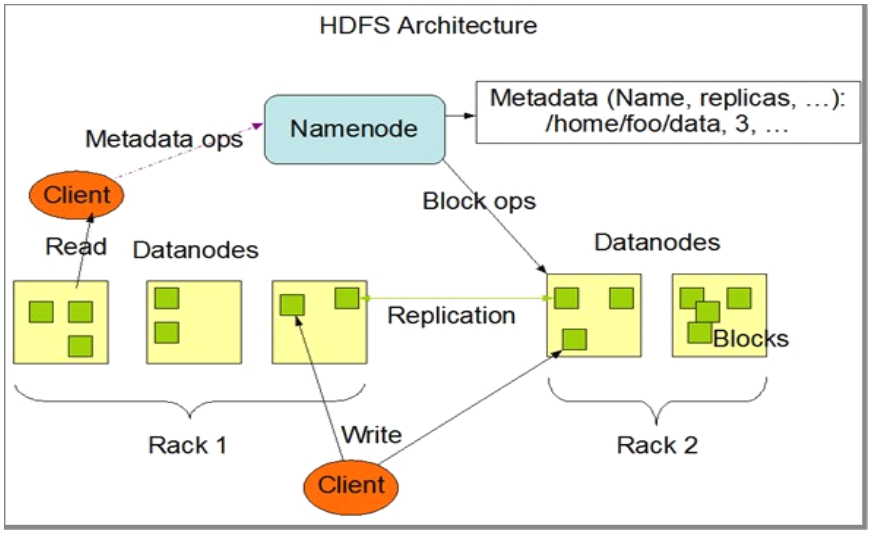
```
top - 20:37:21 up 35 days,  5:14,  2 users,  load average: 1.53, 0.73, 0.29
Tasks: 108 total,   1 running, 107 sleeping,   0 stopped,   0 zombie
Cpu0  : 12.4%us,  3.0%sy, 12.4%ni, 61.9%id,  8.7%wa,  0.3%hi,  1.3%si,  0.0%st
Cpu1  : 17.7%us,  5.4%sy,  3.7%ni, 70.9%id,  2.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  : 24.3%us,  2.0%sy,  4.0%ni, 67.0%id,  2.7%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  : 27.8%us,  5.0%sy,  1.3%ni, 47.0%id, 18.9%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8172248k total,  8126936k used,    45312k free,    56896k buffers
Swap:  2008116k total,      428k used,  2007688k free,  7690112k cached
```

从这里就可以找到突破点，LogBackup 服务器是 4 核 cpu，8G 内存，8 块 SAS 盘，RAID 方案为 2RAID1 + 6RAID5。2RAID1 一般用作系统盘，存放系统文件和用户数据，日志数据就存放在那个 RAID5 盘。

因日志备份服务器一般为顺序写入，且写入后不会修改或删除，因此磁盘读取速度可以达到较高的顺序读取速度。SAS 盘顺序读取的平均速度为 110MB/s，6 个盘做 RAID 后，约有 550MB/s 的速度。下面是 RAID5 的工作原理图：



由此可见，RAID 技术可以增大单个逻辑盘的存储容量，且具备了容错能力，由于使用了条带存储方式，磁盘吞吐也较单盘有很大提升，这点和 hadoop 的 HDFS 的存储模型有异曲同工之妙，可以和下图的 HDFS 架构图对比一下：



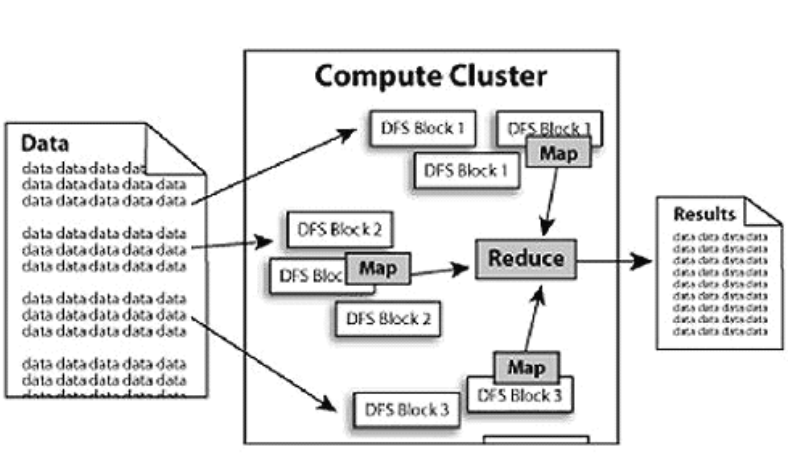
其中，RAID 卡就相当于 HDFS 的 NameNode，独立的物理磁盘就相当于 HDFS 的 DataNode，整个 RAID 盘就相当于一个单机版的 HDFS。

既然多核 cpu 没有吃满，而 raid 盘的 IO 能力又这么强，那么提高脚本的并发量就可以进一步提升 cpu 的利用率。前面已经提到，RAID 盘的 IO 能力比较高，不会因为并发计算而在 IO 方面造成不平衡短板。

另外，你注意到 HDFS 上的文件都是分块存储的，于是你的日志也需要分块，好让他们并发起来吧。那好，来自 4 个地区的日志文件，每隔 5 分钟传输一次，每次都以独立的文件推送过来，这样存储在 LogBackup 服务器上的日志就是打散的了（而不是一整个大文件）。这些文件存在在 RAID 盘上，为后续用脚本并发计算提供了存储基础。

你很快按照 MapReduce 的思路，用 bash 实现了一个简易的并行计算框架，起名为 Bash-MapReduce。

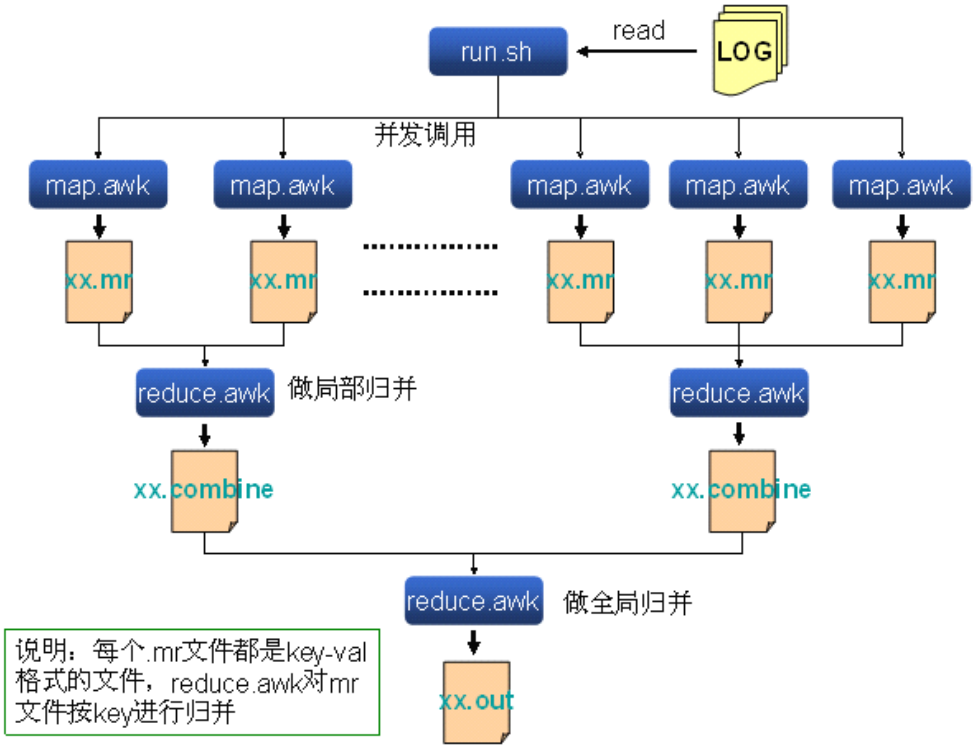
我们还是引用 Hadoop 的 MapReduce 原理图来说明这个并发模型：



我们对照上图，结合一个例子来说明，假设我们要计算某个产品一天的 PV，我们要计算的全量数据（即 web 前端服务器一天切割成的 4 * 24 * 60 / 5 = 1152 个日志文件）如上图《Data 文件》，因全量数据已经分割存储在 1152 个文件中，就如上图 Computer Cluster 中的 DFS Block 1、DFS Block 2……，而我们的 Bash-MapReduce 框架，就是启动 N 个进程（一般地，N 取 cpu 个数的倍数，如 4 核 cpu，N 取 12），分别对各个文件（即各个 DFS Block）进行 PV 计算，每一个原始文件经过 map 模块后分别得到一个中间结果文件。Map 结束后，reduce 模块把所有的中间结果收集起来，进行归并，得到最终的结果（对应于上图的《Results 文件》）。

从上图也可以看到，并行模型中最重要的就是 Map 模块和 Reduce 模块。Bash-MapReduce 框架就是提供了一个任务分发到进程（Map）、中间结果归并（Reduce）的架构，具体的 Map 功能和 Reduce 功能由用户来编写，其实就是写两个脚本而已，这样只需进行简单的单文件计算编程，就可能获得并发能力。需要注意的是，Map 模块输出的，必须是 key-value 类型的格式化数据。

Bash-MapReduce 框架的并行模型图如下所示：



Bash-MapReduce 非常争气，把性能提高了好多倍，下图是我们运行 Bash-MapReduce 框架进行并行计算后的 cpu 利用率，4 个 cpu 都已经逼进极限，而 IO 没有造成瓶颈：

```
top - 09:52:57 up 35 days, 18:29,  2 users,  load average: 9.44, 3.66, 1.43
Tasks: 130 total,  10 running, 120 sleeping,   0 stopped,   0 zombie
Cpu0  : 95.7%us,  3.1%sy,  0.0%ni,  0.6%id,  0.6%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  : 95.1%us,  3.7%sy,  0.0%ni,  0.3%id,  0.6%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu2  : 95.4%us,  3.6%sy,  0.0%ni,  0.3%id,  0.3%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu3  : 97.3%us,  2.1%sy,  0.0%ni,  0.0%id,  0.3%wa,  0.0%hi,  0.3%si,  0.0%st
Mem:   8172248k total,  8126084k used,    46164k free,    44220k buffers
Swap:  2008116k total,      428k used,  2007688k free,  7879604k cached
```

有了这个简易的并行框架，以后产品经理那些个催命鬼再怎么催需求，几秒就搞定他，呵呵，小 case,so easy~~

听得云里雾里的小 Q 第一次亲密接触泰斗，短短几分钟，被他深入浅出的技术讲解给震摄了，佩服得五体投地。王 sir 也从小 Q 朦胧的眼神中找到了强烈自豪感，呵呵，小忽悠一把就能把他整成这样，那后面的，嘿嘿，你懂的……

不知不觉，午饭时间到，王 sir 领着小 Q 去吃饭。饭局间，小 Q 为了怕冷场，又扯回到刚才讨论的 MapReduce，对它的超强并发性赞不绝口。王 sir 也想借此机会考验下小 Q 的反应能力，问了句：“凡事有利必有弊，有没有考虑到它的弊呢？”

小 Q 挠挠头，想了想，弱弱地问道：“它每一个需求都需要去写脚本统计，统计逻辑是相同的，时间长了，也就成了重复性的体力活，换个参数而已。我们是不是可以更智能点，让查起来更方便快捷呢？”

王 sir 惊叹小 Q 的灵性，太给力了，孺子可教也！于是，很兴奋地讲了一些领域特定语言（domain-specific languages，简称 DSL）方面的概念。

LogBackup 服务器上的日志，一般都是经过了初步格式化后，每行都是\n 分隔，每列都是\t 分隔。如果这些数据是存放在 MySQL 数据库中的就好了，这样很多统计就能用 SQL 来代替，比如前面提到的当天访问次数做多的 ip 的 top10 榜单，可以用“SELECT COUNT(1) AS pv FROM tb_mylog_20110302 GROUP BY client_ip ORDER BY pv DESC”统计出来。下一次需求变动，如果只是更换一下日期，或者更换一下 GROUP BY 的字段，改 SQL 显然比改 shell 脚本要简单得多。

呵呵，不过呢，MySQL 是难以扛起“存储海量日志”这面大旗，SQL 的灵活性倒是挺诱人，能不能开发一种类 SQL 的语言直接操作日志文件呢？想想，办法总会有的：）

当被日复一日的简单重复折腾得家不能回，通宵达旦筋疲力尽的时候，绝望中的一缕曙光照亮了整个黑夜，它就是 LogQL（Log Query Language）。LogQL 是一种专门用于日志统计查询的语言，它是一种领域特定语言（DSL），什么是 DSL 呢？比如 AWK 脚本就是一种 DSL，另外，一些软件为了增强应用程序的配置能力，也会设计专门的语言。LogBackup 服务器上的几种日志有固定格式，于是你开发 LogQL 具备了前提条件。

以一个统计需求为例，要设计的 LogQL 的语法是类似这样的：

```
// 需求: 统计百度和 google 给我们的站点带来多少 pv
INIT {
    FIELDS Referer;                 // 扫描到每一行都需要提取 referer 字段进行分析
    pv = 0;                         // 定义变量 pv
}

INPUT {
    DATE 20110212-20110214;         // 选择日志范围
    TYPE 1;                         // 指定日志格式
}

LINE_FILTER {
    referer LIKE “%www.baidu.com%”
```

```
        AND (referrer LIKE “%www.google.com.hk%”
             OR referrer LIKE “%www.google.com%”
             OR referrer LIKE “%www.google.cn%”);
}

LINE_STAT{
    pv = pv + 1;                    // 每一行经过 LINE_FILTER 后，都执行此操作
}
OUTPUT{
    pv;                             // 输出内容，日志本身的 field 或脚本定义的变量
}
```

不经一番寒彻骨，哪来梅花扑鼻香哦，熬夜奋战几宿，基于 lex 和 yacc 把 LogQL 开发出来了，可以解析上面的小示例类似的语法，解决了绝大部分的统计需求。实用、高效，又省心，感谢 LogQL，让我们专注于要提取的数据，而不是提取的逻辑，解救了我们这帮被催命鬼（产品经理）压迫的奴隶。呵呵，悲催吧~谁叫我们是数据分析者，伤得起吗你？！

起来，不愿做奴隶的人们~~被鬼催的感觉，痛不欲生啊，我们要当自己的主人，救世主就是我们自己。

当被需求驱动着，自己感觉很被动，不愿做需求响应者，想翻身做需求管理者。于是搭建了一个 web 前台，写了个网页，让每个人都可以从网页提交 LogQL 到 LogBackup 服务器去执行统计任务。另外，还稍上一份详尽的 LogQL 的使用说明，并召集产品经理进行 LogQL 的使用培训，由于 LogQL 是类 SQL 的语言，简单易用，产品经理们很快掌握了你发明的 LogQL，并通过你的 web 前台，独立完成数据的统计与提取。

嗯嗯，不用再亲自出马，难得片刻闲，嘿嘿，喝杯咖啡，品一品这种久违的悠然自得的惬意~

小 Q 望着师傅那一脸的自豪，对他的钦佩犹如滔滔江水连绵不绝，又如黄河泛滥一发不可收拾！“这么好用的 LogQL 我居然都没听说过，简直太肤浅了”，这不比不知道，一比吓几跳，泰斗，果然名不虚传呐。

王 sir 看到小 Q 那傻乎乎一愣愣的表情，心里知道是咋回事，窃喜;) 不由得惊叹自己的忽悠功！呵呵，侃了半天，饿了，菜也凉了，还是边吃边聊……（此处略去1万字……）

小 Q 吃了差不多，请教师傅一个问题：莫非 LogQL 就是江湖流传上的“葵花宝典”？

王 sir 大囧，口里的饭都快喷了……捂嘴大笑，哈哈！兴致来了，接上：

做事就要做到极致，光凭 LogQL 这一把刷子，是远远不够的：

（1） 计算性能：LogQL 只能运行在单机上，受到 cpu 限制，并发能力有限；

（2） 内存消耗：LogQL 只能运行在单机上，受到内存限制，做类似 DISTINCT 的时候，会导致大量数据积压在内存，最后导致语言解析引擎 core 掉；最典型的案例，就是统计 UV 的时候，需要在内存中维护 user_id 的列表，那会让内存受到极大挑战。

（3） 日志连续性：日志文件分布在多台服务器上，存满一台，则存下一台。如果需要分析的日志刚好跨了两台机器，单机运行的 LogQL 就傻眼了；

（4） 日志格式定义：LogQL 要支持新格式的日志，需要进行一些配置，就好比 SQL 的 Create Table 一样，只不过，在 LogQL 中进行有点麻烦；

（5） 迭代计算：有时候 LogQL 计算出来的数据，需要作为下一步的输出，但是 LogQL 不支持管理中间数据；

（6） 自定义函数：某些字段需要加工一下再输出来，好比 SQL 中的 SUBSTR(access_time, 1, 10)，很无奈，不支持。

（7） LogQL 只支持对未压缩的文本进行统计，对磁盘空间的占用是一个较大的挑战。

随口一说，暴露了 LogQL 的大把缺点，师傅也愁啊，小 Q 不忍心师傅这等郁闷，自己苦下功夫，经常上 bbs.hadoopor.com 论坛，求仙问道。得知名师指点，一款名为 hive 的工具，灰常牛叉，把 LogQL 缺的地方全补上了。它是基于 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，并提供完整的 sql 查询功能，可以将 sql 语句转换为 MapReduce 任务进行运行。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。

通过阅读 hive 的 manual，现在能很申请到一些预算用于 hadoop 平台的搭建，按照官方网站上的操作手册，可以很快把集群搭建起来，现在是考虑迁移到 hadoop 平台的时候了。接下来，Hadoop 平台搭建后的第一件事情，就是数据迁移。

考虑到日志的安全性，日志备份机还是需要继续保留，以便有需要的时候取出；考虑到日常的统计分析需求处理的日志一般都在最近的 60 天，因此最近的 60 天日志可以放到 hadoop 平台中进行存储及计算；考虑到 MapReduce 编程的易操作性，使用 hive 对 hadoop 平台上的数据进行管理。因此，既然 hadoop 平台定位于并行计算，而 LogBackup 定位于备份，那么 LogBackup 仍需保留。

因为 hadoop 能自动识别压缩文件，使用 gzip 对日志文件进行压缩，默认的压缩级别，压缩比可以达到 5：1，为了节省存储空间及网络带宽，日志文件在 Collector 服务器直接压缩，一式两份，一份传给 LogBackup 服务器，一份传给 hadoop 集群。然后你可以通过 hive 客户端，创建 external 表来管理这些格式化的日志。

```
CREATE EXTERNAL TABLE IF NOT EXISTS tb_weblog (
    client_ip      STRING,
    user_id        STRING,
    access_time    STRING,
    url            STRING,
    referer        STRING,
    status         INT,
    page_size      INT,
    agent          STRING)
PARTITIONED BY (statdate STRING, channel STRING)
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
    LOCATION '/user/dw/weblog';
```

其中，statdate 是当天的日期，格式为“YYYYMMDD”；channel 是频道名称，因为你的网站有主频道、新闻频道、博客频道等，进行频道切分有助于后续按频道进行有针对性的流量分析。

借助于 hive，进行日志统计就非常简单，比如我们要处理这样一个需求：计算 1 月份各频道每

天的回弹率，其中，回弹率=回弹用户数/总用户数，回弹用户定义为 pv=1 的那些用户。输出的形式“日期 频道名 回弹率”。这个需求的目的是想了解各频道对用户吸引力如何。

根据需求，我们需要分别统计当天 pv=1 的用户数和当天总用户数，很明显我们需要统计每个用户每天的 pv 数，在此基础上才能提取到 pv=1 的用户数。为了简化操作，我们需要先建立一张中间表：用户访问行为宽表。有了这张宽表记录，以后针对用户级别的需求都可以基于此宽表进行。这张宽表结构设计如下：

```
CREATE EXTERNAL TABLE IF NOT EXISTS tb_weblog_user (
        user_id         STRING,
        channel         STRING,
        pv              INT)
PARTITIONED BY (statdate STRING)
    CLUSTERED BY(suid) INTO 8 BUCKETS
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
    LOCATION '/user/dw/tb_weblog_user';
```

我们期待这张宽表按天生成，内容类似下面：

```
hive> select * from tb_weblog_user where statdate='20110101' limit 10;
OK
1000389296      www     3       20110101
100040834       www     2       20110101
1001614708      image   1       20110101
1002153202      news    3       20110101
1002205781      news    1       20110101
1002459159      www     3       20110101
1002495573      blog    1       20110101
1002567168      blog    2       20110101
1002616118      news    3       20110101
1002647394      image   2       20110101
```

建好表结构之后，我们写一个脚本，逐天把数据跑出来

```
#!/bin/sh
function init()
{
    export HADOOP_HOME=/usr/local/hadoop
    export HIVE_HOME=/usr/local/hive
    export PATH=$PATH:$HADOOP_HOME/bin:$HIVE_HOME/bin
}
function do_one_day()
{
        local _date=$1
        hql="ALTER TABLE tb_weblog_user ADD  PARTITION (statdate='${_date}') LOCATION
'/user/dw/tb_weblog_user/${_date}';"
        hive -e "$hql"
        hql="INSERT  OVERWRITE  TABLE  tb_weblog_user  PARTITION(statdate='${_date}')
```

```
SELECT user_id,channel,COUNT(1) AS pv FROM tb_weblog WHERE statdate = '${_date}' GROUP
BY statdate,channel,user_id"
        hive -e "$hql"
}

Init
for the_day in `seq -w 1 31`
do
     do_one_day 201101${the_day}
done
```

这个脚本的逻辑有几个关键点：

1、对于每天，用户宽表都需要增加分区 statdate，这个是建表的时候规定的；

2、从原始的网站流量数据表 tb_weblog 中进行查询，把汇总结果充入到我们的用户宽表 tb_weblog_user 中。

有了用户宽表，我们再回过头来统计回弹用户率这个需求，那就轻而易举了。

```
SELECT channel, IF(pv=1, 1, 0) AS reject_uv, COUNT(1) AS uv
FROM tb_weblog_user
WHERE statdate like '201101%'
GROUP BY statdate, channel;
```

输出的结果是“频道名 回弹用户数 当天用户数 统计日期”的格式，从开发到数据统计完毕，几分钟就可以完成，非常方便。如果我们使用以前的方式，使用常规的编程语言来开发实现，恐怕在开发上需要耗费的各种开销会非常可观。

这个需求实现的关键点在于中间表-用户访问行为宽表的引入，试想，如果需求中“回弹用户”的定义发生了改变，pv<=2 的用户都算入回弹用户，那我们需要修改的，仅仅是那个基于宽表提取数据的 SQL，影响不大。

鉴于原始的流水表数据量太大，而中间表在这次统计需求中发挥了很重要的作用，你很自然的想到，针对原始流水表进行一个初步的汇总，会后续的统计分析是非常有帮助的。于是，类似用户访问行为宽表，你根据日常的需求，扩展了宽表，上百个字段的宽表也是常有的事。在时间维度上也增加了一些新的宽表，如用户周宽表、月宽表，这样就能很方便对各种周期的数据进行总览，这些宽表一次生成，多次使用，因为是汇总数据，规模比流水表小了不少，流水表一般是一月一删，而汇总数据体积小，可以保留更长的时间。同时也避免了临时数据提取时产生的计算资源消耗，从而提高了数据提取的效率。

支持类 SQL 语法的计算平台和宽表概念的出现是一个很重要的转折点，开发人员开始从重复性的数据提取工作中脱离出来，有更多的时间去透过数据思考业务，数据工作从面向数据，开始转向面向业务。工作思路和方法上，都有很多转变。

一开始在 hive 上执行的任务不多，也都比较简单，我们在 linux 下简单的使用 shell 脚本进行开发，再使用 crontab 来定时启动，基本上都能满足需要。但是由于每日日志量大小变化及 hadoop 集群的性能等因素影响，同一个 hive 任务的每次执行耗时并不固定，如果写死在 crontab 中的定时任务的启动时间安排不得当，很容易导致下一个定时任务启动了，但它所依赖的上一个定时任务的

输出结果还没有生成，进而导致该任务执行失败。更糟糕的是，后续还有一大堆任务依赖该任务的输出结果（中间宽表），并且都是定时执行的，于是纷纷执行失败，出现了可怕的多米诺骨牌效应。

最常见的情况是，我们每天早上 10 点前会从宽表中提取一些数据，作为日报邮件发出。前一天的数据最早顶多凌晨 0 点就绪，到 10 点必须生成宽表，集群的工作时间只有 10 个小时。

为了避免前面提及的多米诺骨牌效应，在任务少的时候，我们可以通过适当拉宽任务的定时启动的时间间隔来保证每一个环节都有充足的时间完成计算任务，但是在任务较多的情况下，任务排期就会出现挑战。另外，任务之间有依赖关系，crontab 并不是一个很好的管理任务间依赖关系的工具。除了依赖关系管理，我们还需要失败重试、告警通知。当然，把这些需求留给 crontab 似乎太过苛刻，，因此我们设计一个任务调度系统，统一对计算任务进行管理。结合前面遇到的问题，我们一起来梳理一下，任务调度系统需要提供哪些特性呢？梳理一下，列个清单：

NO.	特性	描述
1	扩展能力	提供水平扩展能力，可动态增加或减少任务数量，并调整任务之间的依赖关系；
2	并行能力	对于已经满足依赖条件的任务，能控制一定数量的任务进入执行状态，防止并发过高压垮集群，也避免串行执行，不能充分发挥集群的性能
3	前置检查	允许提交自定义脚本，编写自定义的条件检查是否满足任务执行条件。比如任务运行前需要通过检查前一天的数据是否齐全，可以约定数据传输完毕后，生成一个check文件，则前置检查脚本则只需简单检查check文件存在与否，便知数据完整性；
4	重试机制	任务执行失败时，提供错误重试机制，指定次数内重试不成功需要告警。有时候hadoop集群并行处理的任务较多，会导致临时文件撑满datanode，这种情况下一般重试会凑效；
5	垃圾清理	允许提交自动以脚本，用于任务异常终止时进行垃圾清理。某些任务执行过程中需要建立临时表，生成一些中间数据，如果任务异常退出，则临时表、中间数据就会成为垃圾，调度系统需要负责垃圾清理；
6	性能监控	每个任务的启动时间、完成时间、执行状态、执行耗时均做记录，对于耗时超过设置值的需要告警。除了告警监控的用途外，通过耗时记录，也可以直观的找到耗时大户，进行任务性能调优的时候，这些耗时数据将成为重要的参考；
7	任务控制	提供定时调度、手工调度两种控制方式。其中，手工调度是指能通过前台页面控制任务的start、stop、restart操作。在调试任务、重跑数据、手工恢复数据等场合都会用到该功能；
8	依赖管理	能描述任务间的依赖关系，保证任务按照依赖性的先后顺序进行调度。如果某个节点任务执行失败，系统自动取消所有依赖于该节点的后续任务，并调用告警机制。比如日宽表计算失败，则当日的日报邮件能停发或改发故障通知，而不是发送携带错误数据空白数据的邮件；
9	柔性机制	能标注任务的重要级别，在集群资源吃紧等情况下进行服务降级，优先保证重要任务的调度权。比如当天日报所依赖数据的运算任务优先级较高，而月汇总表的运算任务可以降权；

用不严谨的工作流程和粗糙的计算逻辑来提供的数据，是非常危险的，因为运维故障随处可见，磁盘可能损坏、服务器可能崩溃、机架可能断电、机房可能着火、网络可能瘫痪，各种不确定的因

素太多，一旦出现故障，没有任何防范措施的数据提取只能是提供错误的数据，错误的数据将面临信任危机，久而久之，不光数据服务部门丧失数据公信力，整日疲于做数据核查、数据恢复，也会成为消耗时间的苦差事。不过现在借助于任务调度系统，我们的任务管理能力将得到大大增强，现在可以高枕无忧了。

路慢慢其修远兮，吾将上下而求索，共勉！

咆哮体后记：

深夜，夜凉如水有木有!!!!!!!!!!!!!

泥马疲惫困睡!!!!!!! 亲!!!!!!!!!!

偶生噩梦，老板发飙：把这些个数据在几个 hadoop 集群间飞会儿!!!!!

尼玛真 BT 有木有!!!!!!!!

好环躁啊!!!!!!!!

又来一 SB 催命鬼：简单配置就实现数据在不同数据平台间流转啊!!!!!!

哎，还要搞个通用数据传输模块，有木有!!!

架构演变!!!!!!!

永无止境，伤不起呀!!!!!!!!!!!!!

你伤不起啊啊啊啊!!!!!!!

计算不均衡问题在 Hive 中的解决办法

李均*

一、 Hive 简介

由于 facebook 在数据处理领域的影响力以及 Hive 开源社区非常活跃，Hive 慢慢的被国内很多大型互联网公司采用作为替代传统关系型数据仓库的解决方案。

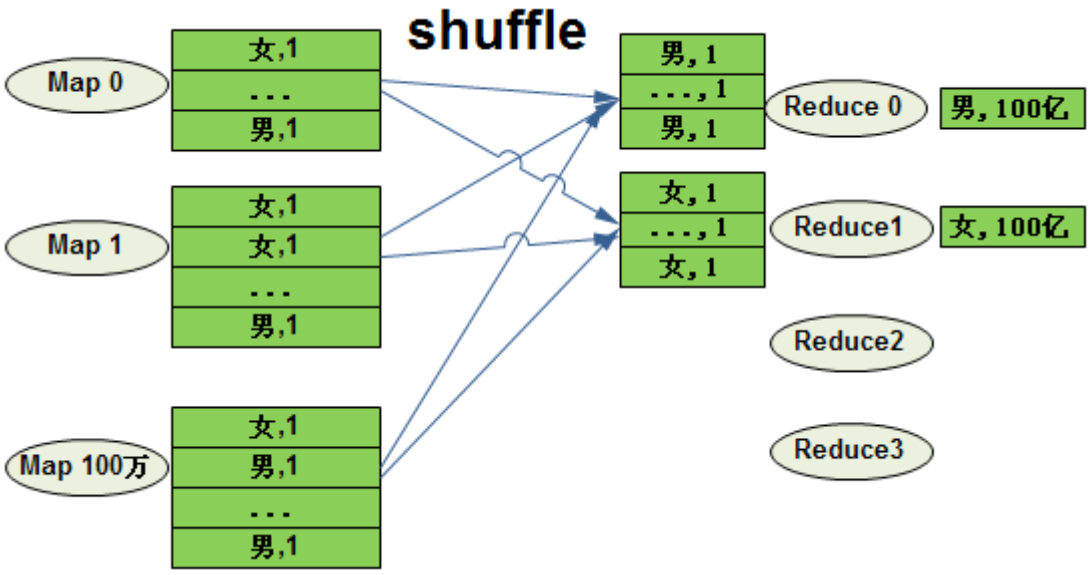
本文主要介绍在 hive 使用过程中计算不均衡产生的原因，以及相应的解决办法。

二、 Group By 中的计算均衡优化

1.1 Map 端部分聚合

先看看下面这条 SQL，由于用户的性别只有男和女两个值。如果没有 map 端的部分聚合优化，map 直接把 groupby_key 当作 reduce_key 发送给 reduce 做聚合，就会导致计算不均衡的现象。虽然 map 有 100 万个，但是 reduce 只有两个在做聚合，每个 reduce 处理 100 亿条记录。

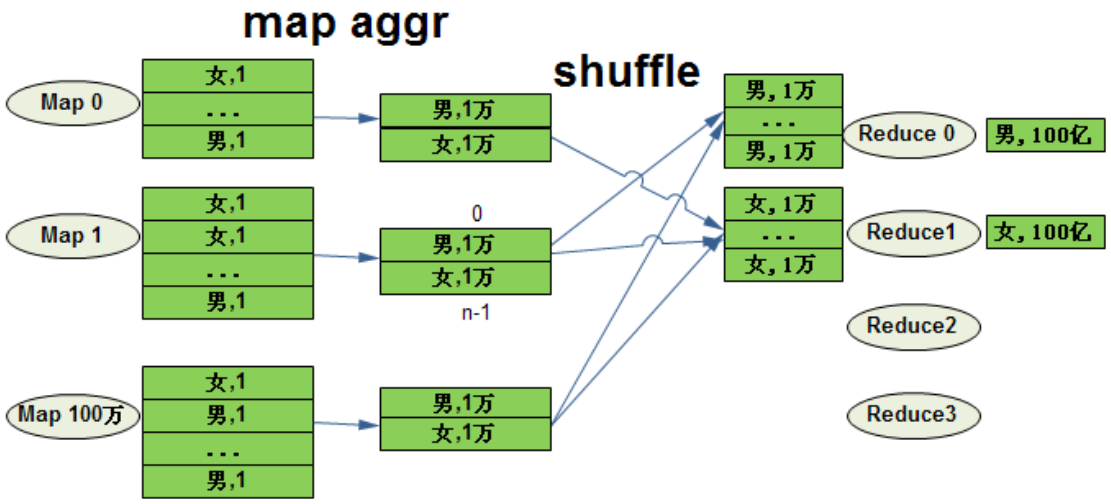
select user.gender,count(1) from user group by user.gender



没开 map 端聚合产生的计算不均衡现象

hive.map.aggr=true 参数控制在 group by 的时候是否 map 局部聚合，这个参数默认是打开的。参数打开后的计算过程如下图。由于 map 端已经做了局部聚合，虽然还是只有两个 reduce 做最后的聚合，但是每个 reduce 只用处理 100 万行记录，相对优化前的 100 亿小了 1 万倍。

作者简介：李均，目前从事分布式数据仓库技术架构和开发工作，对 hdfs+mapreduce+hive 的技术框架和源码有深入的研究。积累了丰富的海量数据处理平台和业务流程建设经验。个人职业目标是希望能成为国内顶尖的云计算领域专家。
联系方式：fiberlijun at yahoo.com.cn



map 端聚合打开

map 聚合开关缺省是打开的，但是不是所有的聚合都需要这个优化。考虑下面的 sql，如果 groupby_key 是用户 ID，因为用户 ID 没有重复的，因此 map 聚合没有太大意义，并且浪费资源。

select user..id,count(1) from user group by user.id

hive.groupby.mapaggr.checkinterval = 100000

hive.map.aggr.hash.min.reduction=0.5

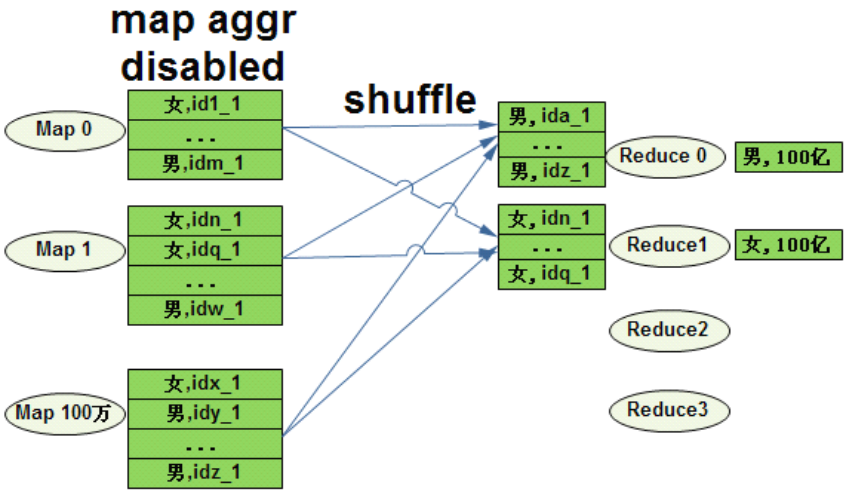
上面这两个参数控制关掉 map 聚合的策略。Map 开始的时候先尝试给前 100000 条记录做 hash 聚合，如果聚合后的记录数/100000>0.5 说明这个 groupby_key 没有什么重复的，再继续做局部聚合没有意义，100000 以后就自动把聚合开关关掉，在 map 的 log 中会看到下面的提示：

2011-02-23 06:46:11,206 WARN org.apache.hadoop.hive.ql.exec.GroupByOperator: Disable Hash Aggr: #hash table = 99999 #total = 100000 reduction = 0.0 minReduction = 0.5

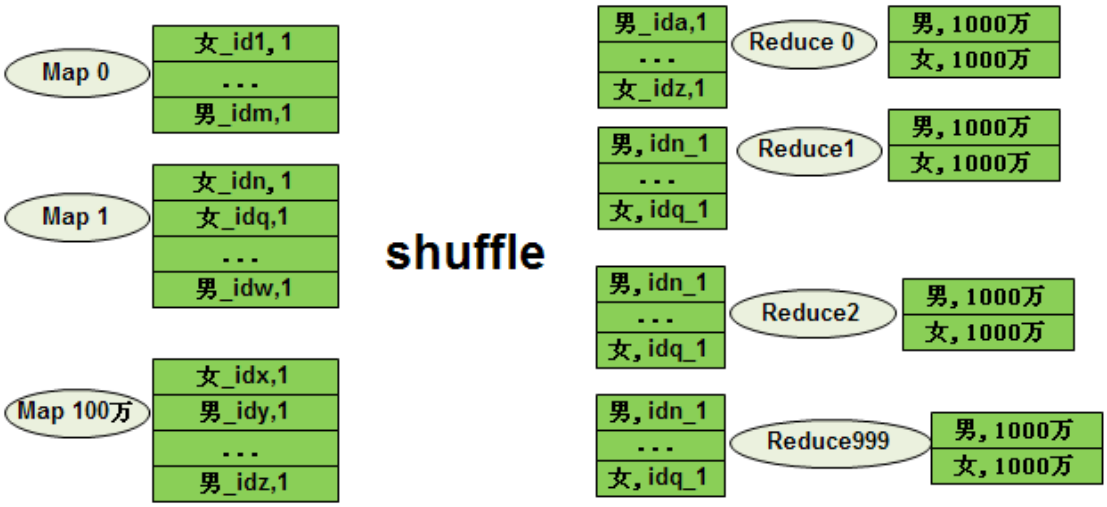
1.2 数据倾斜

通常这种情况都是在有 distinct 出现的时候，比如下面的 sql，由于 map 需要保存所有的 user.id，map 聚合开关会自动关掉，导致出现计算不均衡的现象，只有 2 个 redcue 做聚合，每个 reduce 处理 100 亿条记录。

select user.gender,count(distinct user.id) from user group by user.gender

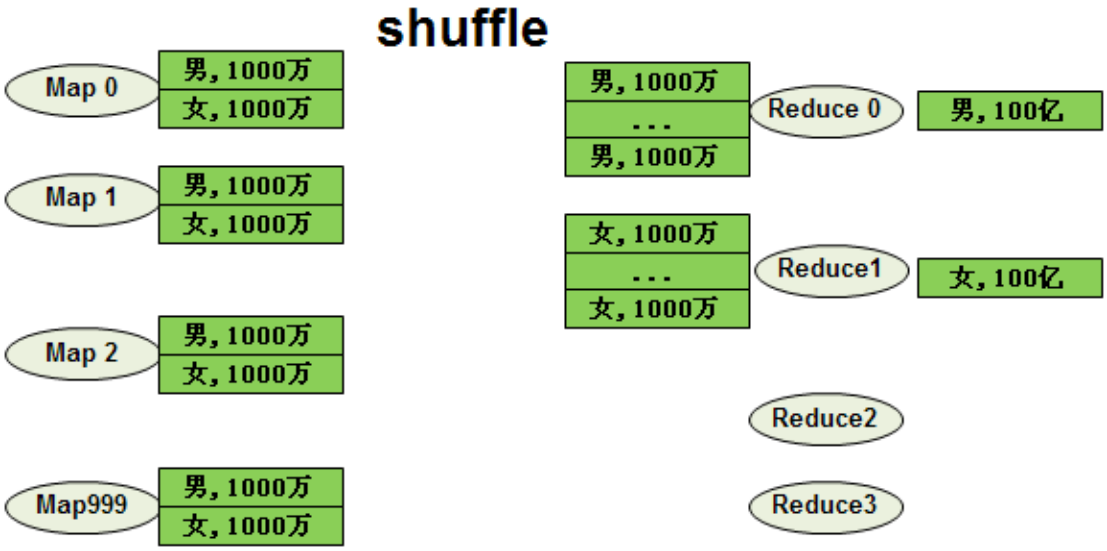


hive.groupby.skewindata = true 参数会把上面的 sql 翻译成两个 MR，第一个 MR 的 reduce_key 是 gender+id。因为 id 是一个随机散列的值，因此这个 MR 的 reduce 计算是很均匀的，reduce 完成局部聚合的工作。



MR1

第二个 MR 完成最终的聚合，统计男女的 distinct id 值，数据流如下图所示，每个 Map 只输出两条记录，因此虽然只有两个 redcue 计算也没有关系，绝大部分计算量已经在第一个 MR 完成了。



MR2

hive.groupby.skewindata 默认是关闭的，因此如果确定有不均衡的情况，需要手动打开这个开关。当然，并不是所有的有 distinct 的 group by 都需要打开这个开关，比如下面的 sql。因为 user.id 是一个散列的值，因此已经是计算均衡的了，所有的 reduce 都会均匀计算。**只有在 groupby_key 不散列，而 distinct_key 散列的情况下才需要打开这个开关，其他的情况 map 聚合优化就足矣。**

select id,distinct(gender) from user group by user.id;

三、 Join 中的计算均衡优化

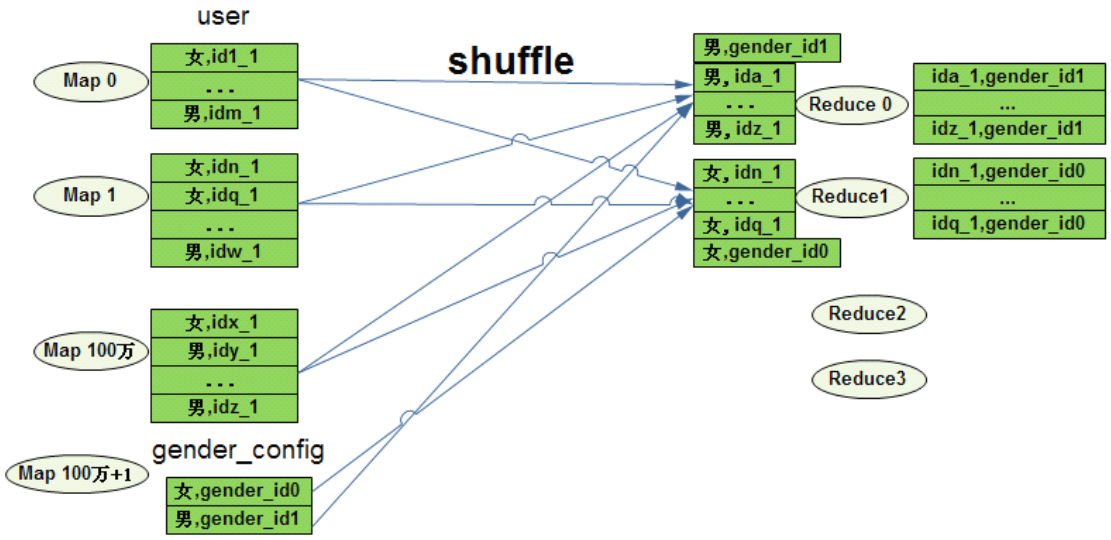
在 hive 中，join 操作一般都是在 reduce 阶段完成的，写 sql 的时候要注意把小表放在 join 的左

边，原因是在 Join 操作的 Reduce 阶段，位于 Join 操作符左边的表的内容会被加载进内存，将条目少的表放在左边，可以有效减少发生 out of memory 错误的几率。

一个大表和一个配置表的 reduce join 经常会引起计算不均衡的情况。比如配置表 gender_config(gender string,gender_id int)。把“男”“女”字符串映射成一个 id。配置表和上面的 user 表 join 的 sql 如下：

select user.id gender_config.gender_id from gender_config join user on gender_config.gender=user.gender

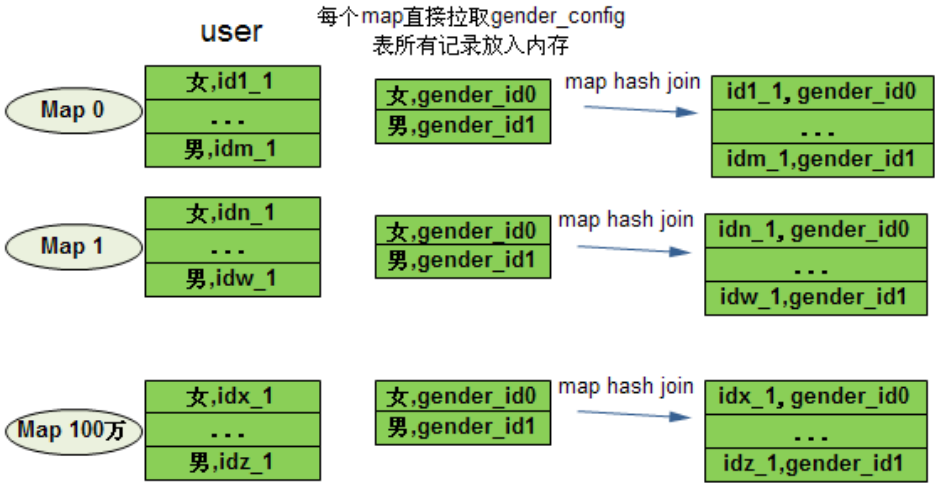
gender 只有男女两个值，hive 处理 join 的时候把 join_key 作为 reduce_key,因此会出现和 group by 类似的 reduce 计算不均衡现象，只有两个 reduce 参与计算，每个 reduce 计算 100 亿条记录。



一个大表和一个小配置表的 **reduce join** 流程图

这种大表和配置表通常采用 mapjoin 的方式来解决这种不均衡的现象。目前 hive 是采用**/*+ MAPJOIN(gender_config) */**提示的方式告诉翻译器把 sql 翻译成 mapjoin，提示里必须指明配置表是哪个。

select /*+ MAPJOIN(gender_config) */ user.id gender_config.gender_id from gender_config join user on gender_config.gender=user.gender



一个大表和一个小配置表的 **map join** 流程图

每个 map 会把小表读到 hash table，然后和大表做 hash join。因此 map join 的关键是小表能放入 map 进程的内存，如果内存放不下会序列化到硬盘，效率会直线下降。

成千上万个 map 从 hdfs 读这个小表进自己的内存，使得小表的读操作变成成个 join 的瓶颈，甚至有些时候有些 map 读这个小表会失败（因为同时有太多进程读了），最后导致 join 失败。临时解决办法是增加小表的副本个数。**Hive-1641** 已经解决了这个问题，主要思路是把小表放入 Distributed Cache 里，map 读本地文件即可。

Hive 目前的 mapjoin 实现还不是很完美，开源社区有一些 patch 都是解决 mapjoin 的问题，具体可以参考这个链接 http://wiki.apache.org/hadoop/Hive/JoinOptimization。

Join 算子在 Hadoop 中的实现

宫振飞*

我们可以使用 MapReduce 做大型数据集的 Join，但是这些 Join 操作的代码我们都需要从头开始。

对于一条语句中有多个 Join 的情况，如果 Join 的条件相同，比如查询：

```
INSERT OVERWRITE TABLE pv_users
  SELECT pv.pageid, u.age FROM page_view p
  JOIN user u ON (pv.userid = u.userid)
  JOIN newuser x ON (u.userid = x.userid);
```

如果 Join 的 key 相同，不管有多少个表，都会合并为一个 Map-Reduce 任务，而不是‘n’个。在做 OUTER JOIN 的时候也是一样。

如果 Join 的条件不相同，比如：

```
INSERT OVERWRITE TABLE pv_users
  SELECT pv.pageid, u.age FROM page_view p
  JOIN user u ON (pv.userid = u.userid)
  JOIN newuser x on (u.age = x.age);
```

则 Map-Reduce 的任务数目和 Join 条件的数目是对应的，上述查询和以下查询是等价的：

```
  INSERT OVERWRITE TABLE tmptable
    SELECT * FROM page_view p JOIN user u
    ON (pv.userid = u.userid);
  INSERT OVERWRITE TABLE pv_users
    SELECT x.pageid, x.age FROM tmptable x
    JOIN newuser y ON (x.age = y.age);
```

下面我们来分别介绍一下使用 MapReduce 如何解决两表 Join 的情况，多表 Join 的情况可以依据上述情况等价转化。

一、 Distribute cache

DistributedCache 是框架提供的用于缓存文件的工具类。应用程序可以在 JobConf 通过 urls (hdfs://) 指定，DistributedCache 类认为通过 hdfs:// urls 指定的文件已经存储在文件系统中。作业运行的时候，如果指定了 DistributedCache ，框架会把每个所需要的文件拷贝到需要执行任务的节点上。当作业执行的时候，DistributedCache 不应该在程序中被修改。

文件可以通过设置 mapred.cache.files 属性被缓存。如果缓存的文件有多个，可以通过逗号来分

作者简介：宫振飞，主要从事分布式计算的研发工作，对 Hadoop 也有深入的研究和应用。
联系方式：gzfhit at gmail.com

割。也可以在程序通过 DistributedCache.addCacheFile(URI,cof)来设置。我们可以在程序运行时使用 DistributedCache ，在 Map 或者 Reduce 任务中通过 DistributedCache.getLocalCacheFiles(jobConf)来获得作业设置的缓存文件。

如果我们需要 Join 的两个表，其中一个表是小表，可以一次装载到内存，我们就可以使用 DistributedCache 来完成 Join 操作。

有两张表 URL 和 QQLogin，其字段如下，现在需要做的连接是把 QQLogin 表中每个 Md5 替换成 URL 自身。其中 URL 表是小表，一次可以加载到内存，登录信息表是一个 QQ 登录的流水账，每天的记录数很多。

表 URL 对应的字段

Url	每个页面的 URL
Md5	URL 对应的 MD5 编码

表 QQLogin 对应的字段

Md5	页面 Md5 编码
QQ	登录页面的 QQ 号码
Time	登录时间

这种情况下我们就可以考虑使用 DistributedCache，首先我们在 run 函数里面通过 API DistributedCache.addCacheFile(newPath(), jobConf)把 URL 表对应的文件作为程序的缓存文件。在 Map 任务我们来做 Join 操作，首先在 Map 任务的 configure() 方法中 DistributedCache.getLocalCacheFiles(job)得到程序的缓存文件，然后找出 URL 表对应的 md5_url.txt 文件，把它一次放入到 HashMap 里面。然后在 map 函数中我们对 QQLogin 表的每条记录的 Md5 进行查表操作，找出对应的 URL，然后输出，这些 Join 全部是在内存中完成。

```
public class JoinWithDistribute extends Configured implements Tool {

   public static class JoinMapper extends MapReduceBase
   implements Mapper<LongWritable, Text, Text, Text> {

   private  Map<String,String> dic = new HashMap<String,String>();
   private    Text url = new Text();
   private    Text others = new Text();

   public void configure(JobConf job) {
       Path[] cacheFiles = new Path[0];
       try {
           cacheFiles = DistributedCache.getLocalCacheFiles(job);
           for (Path cacheFile : cacheFiles){
               if (cacheFile.getName().indexOf("md5_url.txt") != -1 ){
                   BufferedReader    fis    =    new    BufferedReader(new
FileReader(cacheFile.toString()));
                   String line = null;
                   while ((line = fis.readLine()) != null) {
```

```
                    String[] field = line.split(",");
                    if ( field.length == 2 ){
                        dic.put(field[0], field[1]);
                    }
                }
            }
        }
    }catch(Exception e){
        System.err.println(e.toString());
    }

}
public void map(LongWritable key, Text value,
            OutputCollector<Text, Text> output,
            Reporter reporter)
  throws IOException {

    String line = value.toString();

    String[] record = line.split(",",2);

    if ( record.length == 2 ){
        String other = record[1];
        String md5 = record[0];

        if ( dic.get(md5) != null ){
            url.set(dic.get(md5));
            others.set(other);
            output.collect(url,others);
        }
    }
  }
}
  static int printUsage() {
    System.out.println("JoinWithDistribute      <input>      <output>
<cachefile>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
  }
  @Override
  public int run(String[] arg0) throws Exception {
    // TODO Auto-generated method stub
    JobConf jobConf  =                                          new
JobConf(getConf(),JoinWithDistribute.class);
```

```
        jobConf.setJobName("join with distribute");
        jobConf.setMapperClass(JoinMapper.class);

        jobConf.setInputFormat(TextInputFormat.class);
        jobConf.setOutputFormat(TextOutputFormat.class);

        jobConf.setMapOutputKeyClass(Text.class);
        jobConf.setMapOutputValueClass(Text.class);

        jobConf.setNumReduceTasks(0);
        // Make sure there are exactly 2 parameters left.
        if (arg0.length != 3) {
          System.out.println("ERROR: Wrong number of parameters: " +
                arg0.length + " instead of 2.");
          return printUsage();
        }

        DistributedCache.addCacheFile(new          Path(arg0[2]).toUri(),
jobConf);

        FileInputFormat.setInputPaths(jobConf,new Path(arg0[0]));
        FileOutputFormat.setOutputPath(jobConf, new Path(arg0[1]));

        JobClient.runJob(jobConf);
        return 0;
    }
    /**
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        int     res     =     ToolRunner.run(new     Configuration(),     new
JoinWithDistribute(), args);
        System.exit(res);
    }
}
```

二、 Join with secondary sort

试想一下上面的案例，如果我们以 Md5 为 Key，把两个表的记录输出到同一个 Reducer 上，我们也是可以做 Join 操作的，需要在 Reduce 函数里面遍历 Iterator，找出 url,然后替换登录表中的各个记录，这样做需要两次遍历登录表。我们还有一种更为简单的方法就是使用框架提供的二次排序的功能，把 URL 表中对应的记录在 reduce 函数中排在登录表记录之前，这样就避免的多次遍历的

过程。

首先使用<Md5，标记>作为联合 Key，使用我们在附录中给出的 TextPair 类型，由于 TextPair 的 HashCode 是第一个字段的编码，所以所有相同 Md5 的记录都会分发到同一个 Reduce 中，标记怎么设置呢？我们设置 URL 表记录的标记是 0，登录表的标记是 1。TextPair 注册的比较器是两个字段全比较，0 比 1 小，所以 URL 表的记录会排在登录表记录之前。这时候由于联合 key 中标记的不同，相同标记的两个表记录也不会在同一个分组内。我们需要设置分组的比较器。

jobConf.setOutputValueGroupingComparator(TextPair.FirstComparator.class);FirstComparator 只比较第一个字段，所以 Md5 相同的记录又会在同一个分组。这样 reduce 函数中 Value 迭代器第一个记录就是来自 URL 表，之后用它替换分组其他来自登录表的记录的 Md5 就行了。

需要说明的是在 map 函数中我们可以通过 jobConf.get("map.input.file")来取得 map 处理的文件名称。详细代码如下：

```
public class JoinWiteVlueGroupComparator extends Configured implements
Tool {
   public static class Map extends MapReduceBase implements
          Mapper<LongWritable, Text, TextPair, Text> {

      private boolean flag = true;
      private Text other = new Text();
      private TextPair record = new TextPair();

      public void configure(JobConf job) {
          String input = job.get("map.input.file");
          if ( input.indexOf("md5_url.txt") != -1 ){
              flag = true;
          }
          else
              flag = false;

      }
      public void map(LongWritable key, Text value,
              OutputCollector<TextPair, Text> output, Reporter reporter)
              throws IOException {

          String line = value.toString();
          String []field = line.split(",",2);
          if ( field.length == 2) {
              other.set(field[1]);
              if (flag) {
                  record = new TextPair( field[0],"0");
              } else {
                  record = new TextPair( field[0],"1");
              }
```

```
                output.collect(record, other);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements
            Reducer<TextPair, Text, Text, Text> {

        @Override
        public void reduce(TextPair key, Iterator<Text> values,
                OutputCollector<Text, Text> output, Reporter reporter)
                throws IOException {

            Text url = new Text(values.next());
            while (values.hasNext()) {
                output.collect(url, values.next());
            }
        }
    }

    static int printUsage() {
        System.out.println("JoinWiteVlueGroupComparator <input>
<output>");
        ToolRunner.printGenericCommandUsage(System.out);
        return -1;
    }

    public int run(String[] arg0) throws Exception {
        // TODO Auto-generated method stub
        JobConf jobConf = new
JobConf(getConf(),JoinWiteVlueGroupComparator.class);
        jobConf.setJobName("join with value group");

        jobConf.setMapperClass(Map.class);

        jobConf.setReducerClass(Reduce.class);

        jobConf.setInputFormat(TextInputFormat.class);
        jobConf.setOutputFormat(TextOutputFormat.class);

        jobConf.setMapOutputKeyClass(TextPair.class);
        jobConf.setMapOutputValueClass(Text.class);

        jobConf.setOutputKeyClass(Text.class);
```

```
        jobConf.setOutputValueClass(Text.class);

        JobClient client = new JobClient(jobConf);
        ClusterStatus cluter = client.getClusterStatus();
        int numReduce = (int) (cluter.getTaskTrackers()*
Integer.parseInt(jobConf.get("mapred.tasktracker.reduce.tasks.maximum")
) * 0.95);

        jobConf.setNumReduceTasks(numReduce);
        //jobConf.setNumReduceTasks(1);

    jobConf.setOutputValueGroupingComparator(TextPair.FirstComparator.c
lass);

        // Make sure there are exactly 2 parameters left.
        if (arg0.length != 2) {
            System.out.println("ERROR: Wrong number of parameters: "
                    + arg0.length + " instead of 2.");
            return printUsage();
        }

        FileInputFormat.setInputPaths(jobConf, new Path(arg0[0]));
        FileOutputFormat.setOutputPath(jobConf, new Path(arg0[1]));

        JobClient.runJob(jobConf);
        return 0;

    }

    /**
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        int res = ToolRunner.run(new Configuration(),new
JoinWiteVlueGroupComparator(), args);
        System.exit(res);
    }
}
```

三、 Reduce side join

针对“多对多”连接，我们来探讨一下起解决思路。

要解决的问题可以简化描述一下：

有两组数据，input1 { P1, U1; P1, U2; P2, U3; P3, U4; P4, U4 }，

input2 { P1, C1; P1, C2; P2, C3; P3, C3; P3, C4; P4, C4 }。

要求执行类似于数据库两表 Inner Join 的操作，以 P 为 key，建立起 U 和 C 直接的对应关系，即最终结果为 output { U1, C1; U1, C2; U2, C1; U2, C2; U3, C3; U4, C3; U4, C4 }。

在数据库里，使用类似的 SQL 可以达到要求：SELECT U, C FROM input1 INNER JOIN input2 ON input1.P=input2.P。但如果要放在 Hadoop 里面求解，就需要动些脑筋了。

研究这个问题，首先需要理解 Hadoop 的运行机制。简单来讲，Hadoop 分为 Map 和 Reduce 两个操作：Map 操作将输入（如一行数据）格式化为 <key: value1><key: value2><key: value3> … <key: valueN>这样的一组结果，作为 Map 的输出。Hadoop 在 Map 和 Reduce 之间，会自动把 Map 的输出按照 key 合并起来，作为 Reduce 的输入。Reduce 得到这样一个 {key: [value1, value2, value3, ..., valueN]} 的输入之后，就可以进行自己的处理，完成最终计算了。

针对于我们这里要解决的问题，步骤如下。

将 Map 的输入构造为下面的格式：来自于 input1 的输入格式化为 {<input1, P1>: U1, U2}；来自于 input2 的输入格式化为 {<input2, P1>: C1, C2}。

在 Map 操作内，将数据转化为 {P1: <input1, U1>}，{P1: <input1, U2>}，{P1: <input2, C1>}，{P1: <input2, C2>}，作为 Reduce 操作的输入。

经过 Hadoop 内部自己的操作，实际 Reduce 操作的输入为：{P1: <input1, U1>, <input1, U2>, <input2, C1>, <input2, C2>}。

Reduce 里操作会复杂一下。首先需要执行一次 regroup，得到如下的结果 {<input1>: <input1, U1>, <input1, U2>; <input2>: <input2, C1>, <input2, C2>}。把这个结果拆开，可以得到两个集合：{<input1>, <input2>} 与 {[<input1, U1>, <input1, U2>], [<input2, C1>, <input2, C2>]}。

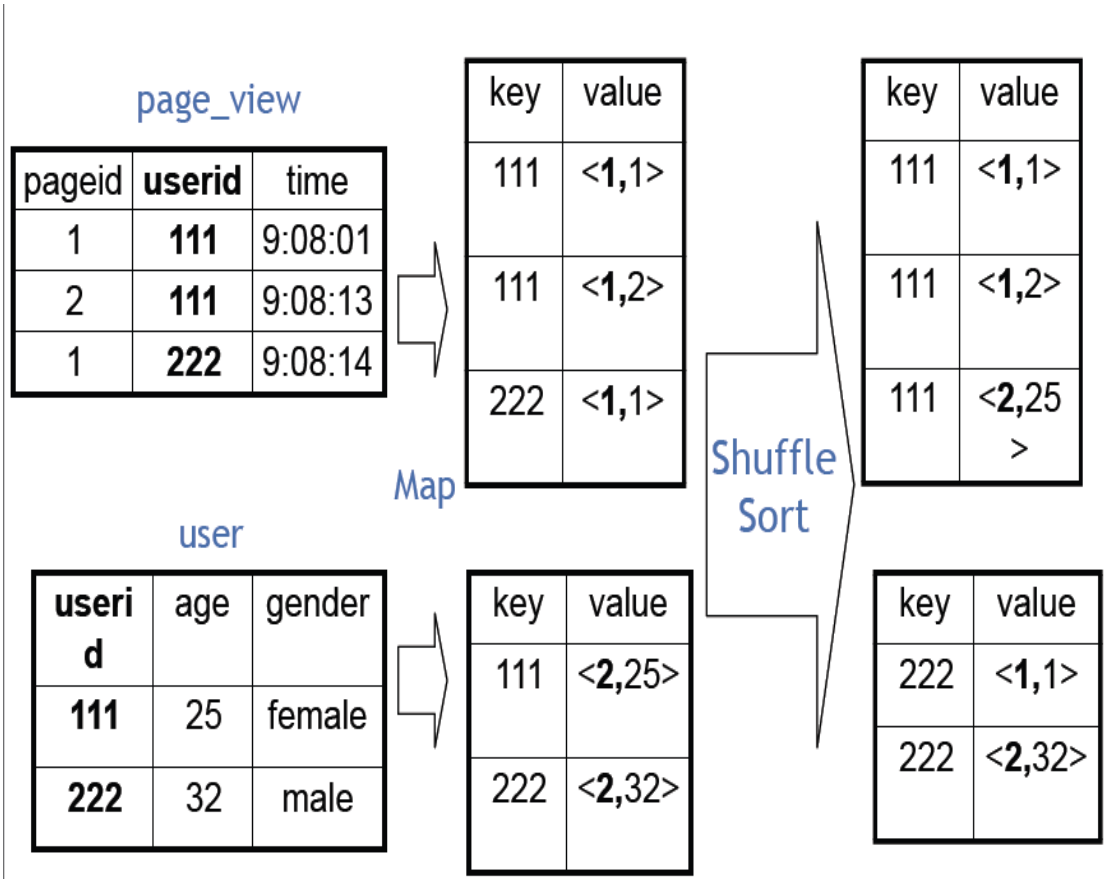
循环遍历集合{[<input1, U1>, <input1, U2>], [<input2, C1>, <input2, C2>]}，把来自 input1 和 input2 的记录进行连接操作。

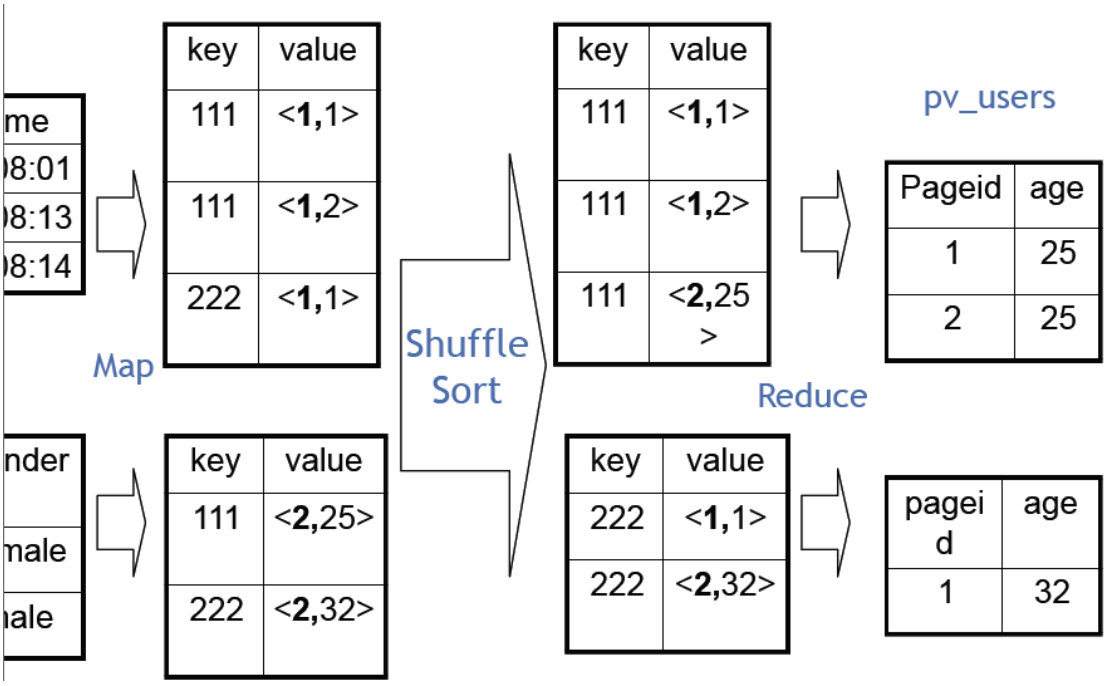
简单的说，可以描述如下：

在 Map 函数中为连接字段作为 Key，以记录和标记（来自的文件）作为 value,经过 reduce 相同连接 Key 的记录会分发到一个 Reduce 中。在 Reduce 我们首先进行分组，把来自于同一个文件的记录放在一个集合里面，最后循环遍历这两个集合，做两个集合的笛卡尔积。

下面的例子，可以很好的说明这个过程：

```
SELECT pv.pageid, u.age
FROM page_view pv JOIN user u ON (pv.userid = u.userid);
```


page_view
pageid userid time
1 111 9:08:01
2 111 9:08:13
1 222 9:08:14
user
userid age gender
111 25 female
222 32 male
Map
key value
111 <1,1>
111 <1,2>
222 <1,1>
111 <2,32>
222 <2,32>
Shuffle Sort
key value
111 <1,1>
111 <1,2>
111 <2,25>
222 <1,1>
222 <2,32>


Map
key value
111 <1,1>
111 <1,2>
222 <1,1>
111 <2,25>
222 <2,32>
Shuffle Sort
key value
111 <1,1>
111 <1,2>
111 <2,25>
222 <1,1>
222 <2,32>
Reduce
pv_users
Pageid age
1 25
2 25
pageid age
1 32

四、 附录

TextPair 类的实现代码

```
public class TextPair implements WritableComparable<TextPair> {

    private Text first;
    private Text second;

    public TextPair() {
        set(new Text(), new Text());
    }

    public TextPair(String first, String second) {
        set(new Text(first), new Text(second));
    }

    public TextPair(Text first, Text second) {
        set(first, second);
    }
    public void set(Text first, Text second) {
        this.first = first;
        this.second = second;
    }

    public Text getFirst() {
        return first;
    }
    public Text getSecond() {
        return second;
    }
    @Override
    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        first.readFields(in);
        second.readFields(in);
    }
    @Override
    public int hashCode() {
        //return first.hashCode() * 163 + second.hashCode();
        return first.hashCode();
```

```
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof TextPair) {
            TextPair tp = (TextPair) o;
            return first.equals(tp.first) && second.equals(tp.second);
        }
        return false;
    }

    @Override
    public String toString() {
        return first + "\t" + second;
    }
    @Override
    public int compareTo(TextPair tp) {
        int cmp = first.compareTo(tp.first);
        if (cmp != 0) {
            return cmp;
        }
        return second.compareTo(tp.second);
    }

    public static class Comparator extends WritableComparator {

        private static final Text.Comparator TEXT_COMPARATOR = new
Text.Comparator();

        public Comparator() {
            super(TextPair.class);
        }

        @Override
        public int compare(byte[] b1, int s1, int l1,
                           byte[] b2, int s2, int l2) {
            try {
                int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) +
readVInt(b1, s1);
                int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) +
readVInt(b2, s2);
                int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2,
firstL2);
                if (cmp != 0) {
```

```
                return cmp;
            }
            return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,
                                           b2, s2 + firstL2, l2 - firstL2);
            } catch (IOException e) {
                throw new IllegalArgumentException(e);
            }
        }
    }
    static {
        WritableComparator.define(TextPair.class, new Comparator());
    }
    public static class FirstComparator extends WritableComparator {

        private static final Text.Comparator TEXT_COMPARATOR = new
Text.Comparator();
        public FirstComparator() {
            super(TextPair.class);
        }
        @Override
        public int compare(byte[] b1, int s1, int l1,
            byte[] b2, int s2, int l2) {
            try {
                int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) +
readVInt(b1, s1);
                int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) +
readVInt(b2, s2);
                return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2,
firstL2);
            } catch (IOException e) {
                throw new IllegalArgumentException(e);
            }
        }
        @Override
        public int compare(WritableComparable a, WritableComparable b)
{               if (a instanceof TextPair && b instanceof TextPair) {
                return ((TextPair) a).first.compareTo(((TextPair)
b).first);
            }

            return super.compare(a, b);
        }
    }
}
```

配置 Hive 元数据 DB 为 PostgreSQL

jov*

HIVE 的元数据默认使用 derby 作为存储 DB，derby 作为轻量级的 DB，在开发、测试过程中使用比较方便，但是在实际的生产环境中，还需要考虑易用性、容灾、稳定性以及各种监控、运维工具等，这些都是 derby 缺乏的。MySQL 和 PostgreSQL 是两个比较常用的开源数据库系统，在生产环境中比较多的用来替换 derby。配置 MySQL 在网上的文章比较多，这里不再赘述，本文主要描述配置 HIVE 元数据 DB 为 PostgreSQL 的方法。

HIVE 版本：HIVE 0.7-snapshot，HIVE 0.8-snapshot

步骤 1：在 PG 中为元数据增加用户的 DB

首先在 PostgreSQL 中为 HIVE 的元数据建立帐号和 DB。

```
--以管理员身份登入 PG：
psql postgres -U postgres

--创建用户 hive_user:
Create user hive_user;

--创建 DB metastore_db，owner 为 hive_user:
Create database metastore_db with owner=hive_user;

--设置 hive_user 的密码：
\password hive_user
```

完成以上步骤以后，还要确保 PostgreSQL 的 pg_hba.conf 中的配置允许 HIVE 所在的机器 ip 可以访问 PG。

步骤 2：下载 PG 的 JDBC 驱动

在 HIVE_HOME 目录下创建 auxlib 目录：

mkdir auxlib

此时 HIVE_HOME 目录中应该有 bin，lib，auxlib，conf 等目录。

作者简介：jov，主要从事 Hive 相关的开发工作。
联系方式：jovz at vip.qq.com

下载 PG 的 JDBC 驱动

wget http://jdbc.postgresql.org/download/postgresql-9.0-801.jdbc4.jar

将下载到的 postgresql-9.0-801.jdbc4.jar 放到 auxlib 中。

步骤 3：修改 HIVE 配置文件

在 HIVE_HOME 中新建 hive-site.xml 文件，内容如下，蓝色字体按照 PG server 的相关信息进行修改。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:postgresql://pg_server_ip:pg_server_port/metastore_db?</value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>org.postgresql.Driver</value>
  <description>Driver class name for a JDBC metastore</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>hive_user</value>
  <description>username to use against metastore database</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>hive_user_pass</value>
  <description>password to use against metastore database</description>
</property>
</configuration>
```

步骤 4：初始化元数据表

元数据库 metastore 中默认没有表，当 HIVE 第一次使用某个表的时候，如果发现该表不存在就会自动创建。对 derby 和 mysql，这个过程没有问题，因此 derby 和 mysql 作为元数据库不需要这一步。

PostgreSQL 在初始化的时候，会遇到一些问题，导致 PG 数据库死锁。例如执行以下 HIVE 语句：

>Create table kv (key,int,value string) partitioned by (ds string);

OK

>Alter table kv add partition (ds = '20110101');

执行这一句的时候，HIVE 会一直停在这。

查看 PG 数据库，发现有两个连接在进行事务操作，其中一个是：

<IDLE> in transaction

此时处于事务中空闲，另外一个是：

ALTER TABLE "PARTITIONS" ADD CONSTRAINT "PARTITIONS_FK1" FOREIGN KEY ("SD_ID") REFERENCES "SDS" ("SD_ID") INITIALLY DEFERRED

处于等待状态。

进一步查看日志，发现大致的过程是这样的：

HIVE 发起 Alter table kv add partition (ds = '20110101')语句，此时 DataNucleus 接口发起第一个 isolation 为 SERIALIZABLE 的事务，锁定了 TBLS 等元数据表。在这个的事务进行过程中，DataNucleu 发现 PARTITIONS 等表没有，则要自动创建。于是又发起了另外一个 isolation 为 SERIALIZABLE 的事务，第一个事务变为<IDLE> in transaction。第二个事务创建了 PARTITIONS 的表后，还要给它增加约束条件，这时，它需要获得它引用的表 SDS 的排他锁，但这个锁已经被第一个事务拿到了，因此需要等待第一个事务结束。而第一个事务也在等待第二个事务结束。这样就造成了死锁。

类似的情况出现在：

>create test(key int);

OK

>drop table test;

当 drop table 时会去 drop 它的 index，而此时没有 index 元数据表，它去键，然后产生死锁。

有三种方法可以解决这个死锁问题：

第一种方法：

使用 PG 的 pg_terminate_backend()将第一个事务结束掉，这样可以保证第二个事务完成下去，将元数据表键成功。

第二种方法：

使 HIVE 将创建元数据表的过程和向元数据表中添加数据的过程分离：

>Create table kv (key,int,value string) partitioned by (ds string);

OK

>show partitions kv;

OK

>Alter table kv add partition (ds = '20110101');

OK

执行以上语句时就不会发生死锁，因为在执行 show partitions kv 语句时，它是只读语句，不会加锁。当这个语句发现 PARTITIONS 等表不在时，创建这些表不会发生死锁。

同样对于 index 表，使用

>Show index on kv;

可以将 IDXS 表建好。

第三种方法：

使用 DataNucleu 提供的 SchemaTool，将 HIVE 的 metastore/src/model/package.jdo 文件作为输入，这个工具可以自动创建元数据中的表。具体的使用方法见：

http://www.datanucleus.org/products/accessplatform_2_0/rdbms/schematool.html

小结

本文给出了使用 PostgreSQL 作为 HIVE 元数据 DB 的配置方法，以及遇到的死锁问题的解决办法，希望对使用 HIVE 和 PostgreSQL 的朋友有帮助。

原文链接：http://blog.csdn.net/zhao6014/archive/2011/02/26/6210252.aspx，欢迎围观。

ZooKeeper 权限管理机制

高正*

一、 ZooKeeper 权限管理机制介绍

本节将简要介绍 ZooKeeper ACL 权限管理的几种方式。ZooKeeper 的权限管理亦即 ACL 控制功能通过 Server、Client 两端协调完成：

a) Server 端：

一个 ZooKeeper 的节点（znode）存储两部分内容：数据和状态，状态中包含 ACL 信息。创建一个 znode 会产生一个 ACL 列表，列表中每个 ACL 包括：权限(perms)、验证模式(scheme)、具体内容(Ids)（当 scheme="digest" 时，Ids 为用户名密码，例如"root：J0sTy9BCUKubtK1y8pkbL7qoxSw="）。ZooKeeper 提供了如下几种验证模式：

a) Digest Client 端由用户名和密码验证，譬如 user:pwd
b) Host Client 端由主机名验证，譬如 localhost
c) Ip Client 端由 IP 地址验证，譬如 172.2.0.0/24
d) World 固定用户为 anyone，为所有 Client 端开放权限

权限许可集合如下，注意的是,exists操作和getAcl操作并不受ACL许可控制，因此任何客户端可以查询节点的状态和节点的ACL：

a) Create 允许对子节点 Create 操作
b) Read 允许对本节点 GetChildren 和 GetData 操作
c) Write 允许对本节点 SetData 操作
d) Delete 允许对子节点 Delete 操作
e) Admin 允许对本节点 setAcl 操作

Znode ACL 权限用一个 int 型数字 perms 表示，perms 的 5 个二进制位分别表示 set**a**cl、**d**elete、**c**reate、**w**rite、**r**ead。比如 adcwr=0x1f，----r=0x1，a-c-r=0x15。

b) Client 端：

Client 通过调用 `zoo_add_auth()` 函数设置当前会话的 `Author` 信息（针对 `Digest` 验证模式）。`Server` 收到 `Client` 发送的操作请求（除 `exists`、`getAcl` 之外），需要进行 `ACL` 验证：对该请求携带的 `Author` 明文信息加密，并与目标节点的 `ACL` 信息进行比较，如果匹配则具有相应的权限，否则请求被 `Server` 拒绝。

作者简介：高正，在南京大学获得计算机科学与技术学士和硕士学位。爱好领域有分布式系统，Web 服务，事务处理，曾在计算机类中文核心期刊《计算机科学》发表论文一篇。目前从事 ZooKeeper 研究与开发的相关工作，对 ZooKeeper 的实际应用颇有经验。
联系方式：gaozheng0123 at 163.com

二、 ZooKeeper 使用接口介绍（C API）

基于上一节的内容，本节介绍 Digest 验证模式下，通过用户名、密码的方式进行节点权限管理需要的相关接口。

1、设置节点权限

Znode 存储 ACL 的内容为密文，所以在 setAcl 时必须将明文的用户名、密码(user：pwd)加密，结果为：EYJny+H3eleOv6O/G6jy9vuSCq8=（28 位），其加密方式为 SHA1。使用方式如下：

```
//ACL参数设置
char szUserPwd[] = "user:pwd";
char szEncUserPwd[32];
char szDigestIds[64] = {0};

//EncryptSHA1()为SHA1加密函数
EncryptSHA1(szUserPwd, strlen((char*) szUserPwd), szEncUserPwd,
        sizeof(szEncUserPwd));
snprintf(szDigestIds,sizeof(szDigestIds),"user:%s", szEncUserPwd);
struct ACL stMyACL[]={{0x1f,{"digest",szDigestIds}},{0x01, {"world",
"anyone"}}};
struct ACL_vector vecMyACL = {2, stMyACL};

//设置ACL到Node
zoo_set_acl(zk, pszPath, -1, &vecMyACL);
```

2、用户验证方式

在 Client 连接 ZooKeeper Server 之后，用如下的接口设置 session 的用户信息：

zoo_add_auth(zk, "digest", "user:pwd", 8, 0, 0);设置 author 信息之后，该 session 的每次操作都带有该 author 标识。如果想用不同的 author 信息操作，只需再调用一次 zoo_add_auth，Client 端以最后一次设置的信息为有效 author 信息。

三、 ZooKeeper SuperDigest

1、 一次 Client 对 znode 进行操作的验证 ACL 的方式为：

（a）遍历znode的所有ACL：

i) 对于每一个 ACL，首先操作类型与权限（perms）匹配

ii) 只有匹配权限成功才进行 session 的 auth 信息与 ACL 的用户名、密码匹配

（a）如果两次匹配都成功，则允许操作；否则，返回权限不够error（rc=-102）

2、 如果 znode ACL List 中任何一个 ACL 都没有 setAcl 权限，那么就算 superDigest 也修改不了它的权限；再假如这个 znode 还不开放 delete 权限，那么它的所有子节点都将不会被删除。唯一的办法是通过手动删除 snapshot 和 log 的方法，将 ZK 回滚到一个以前的状态，然后重启，当然这会影响到该 znode 以外其它节点的正常应用。

3、 superDigest 设置的步骤：

a) 启动 ZK 的时候（zkServer.sh），加入参数：Java

"-Dzookeeper .DigestAuthenticationProvider.superDigest=super:D/InIHSb7yEEbrWz8b9l71RjZJU="（中间没有空格）。

a) 在客户端使用的时候，zoo_add_auth(zh, "digest", "super:test", 10, 0, 0); “super:test”为“super:D/InIHSb7yEEbrWz8b9l71RjZJU=”的明文表示，加密算法同 setAcl。

ZooKeeper 服务器工作原理和流程

陈旭日*

一、 Zookeeper 基本原理

1.1 Zookeeper 的保证

a) 顺序性，client的updates请求都会根据它发出的顺序被顺序的处理；

b) 原子性， 一个update操作要么成功要么失败，没有其他可能的结果；

c) 一致的镜像，client不论连接到哪个server，展示给它都是同一个视图；

d) 可靠性，一旦一个update被应用就被持久化了，除非另一个update请求更新了当前值；

e) 实时性，对于每个client它的系统视图都是最新的。

1.2 Zookeeper server 角色

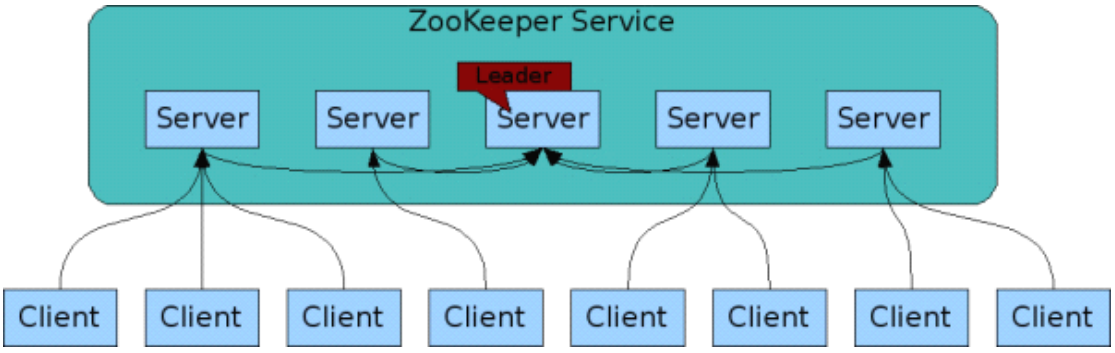
领导者（Leader）：领导者不接受client的请求，负责进行投票的发起和决议，最终更新状态。

跟随者（Follower）：Follower用于接收客户请求并返回客户结果。参与Leader发起的投票。

观察者（observer）：Oberserver可以接收客户端连接，将写请求转发给leader节点。但是Observer不参加投票过程，只是同步leader的状态。Observer为系统扩展提供了一种方法。

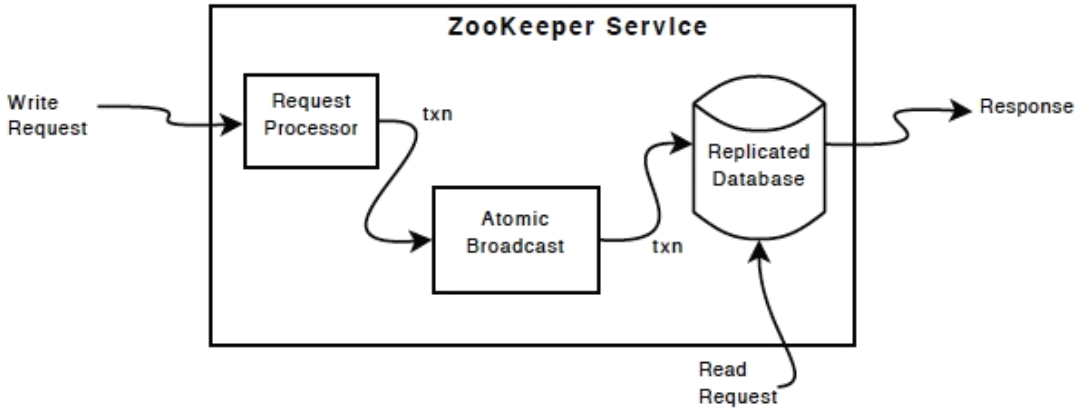
学习者（Learner）：和leader进行状态同步的server统称Learner，上述Follower和Observer都是Learner。

1.3 Zookeeper 集群



通常Zookeeper由2n+1台servers组成，每个server都知道彼此的存在。每个server都维护的内存状态镜像以及持久化存储的事务日志和快照。对于2n+1台server，只要有n+1台（大多数）server可用，整个系统保持可用。

作者简介：陈旭日，吉林大学硕士，兴趣领域：数据库技术、分布式系统、负载均衡技术。对 ZooKeeper 的原理和应用有教深入研究。联系方式：julianchen888 at gmail.com
联系方式：julianchen888 at gmail.com



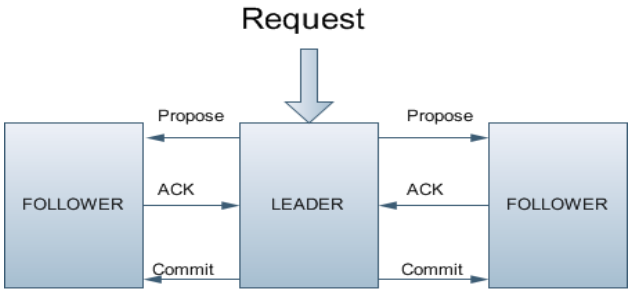
系统启动时，集群中的server会选举出一台server为Leader，其它的就作为follower（这里先不考虑observer角色）。接着由follower来服务client的请求，对于不改变系统一致性状态的读操作，由follower的本地内存数据库直接给client返回结果；对于会改变系统状态的更新操作，则交由Leader进行提议投票，超过半数通过后返回结果给client。

二、 Zookeeper server 工作原理

Zookeeper的核心是原子广播，这个机制保证了各个server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式和广播模式。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和server具有相同的系统状态。

一旦leader已经和多数的follower进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个server加入zookeeper服务中，它会在恢复模式下启动，发现leader，并和leader进行状态同步。待到同步结束，它也参与消息广播。Zookeeper服务一直维持在Broadcast状态，直到leader崩溃了或者leader失去了大部分的followers支持。

Broadcast模式极其类似于分布式事务中的2pc（two-phrase commit 两阶段提交）：即leader提起一个决议，由followers进行投票，leader对投票结果进行计算决定是否通过该决议，如果通过执行该决议（事务），否则什么也不做。

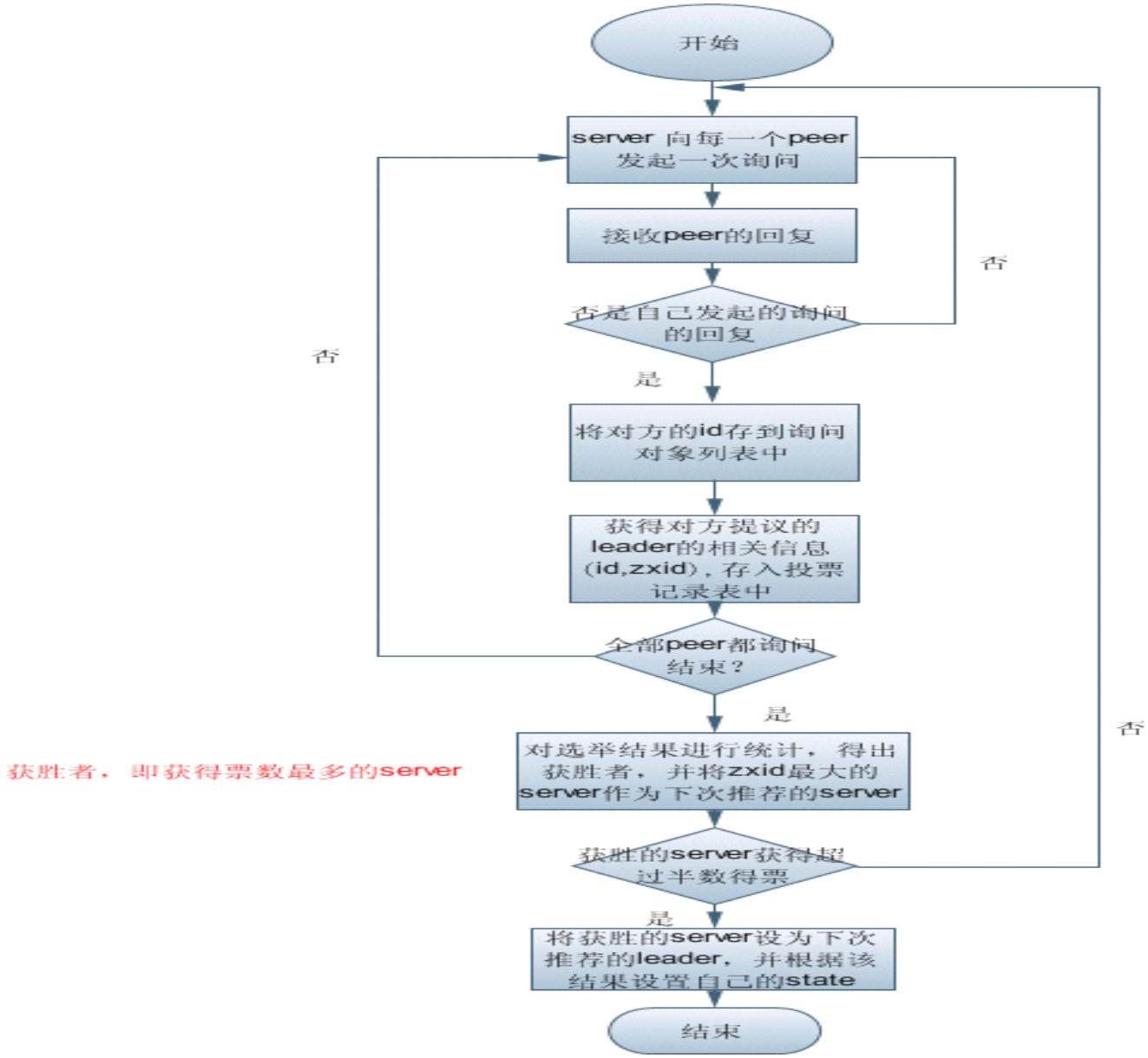


广播模式需要保证proposal被按顺序处理，因此zk采用了递增的事务id号(zxid)来保证。所有的提议(proposal)都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch。低32位是个递增计数。

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个

新的leader，让所有的server都恢复到一个正确的状态。

首先看一下选举的过程，zk的实现中用了基于paxos算法（主要是fastpaxos）的实现。具体如下：



1）每个Server启动以后都询问其它的Server它要投票给谁；

2）对于其他server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都会推荐自己）；

3）收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

4）计算这过程中获得票数最多的的sever为获胜者，如果获胜者的票数超过半数，则改server被选为leader。否则，继续这个过程，直到leader被选举出来；

此外恢复模式下，如果是重新刚从崩溃状态恢复的或者刚启动的的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。

选完leader以后，zk就进入状态同步过程。

1）leader就会开始等待server连接；

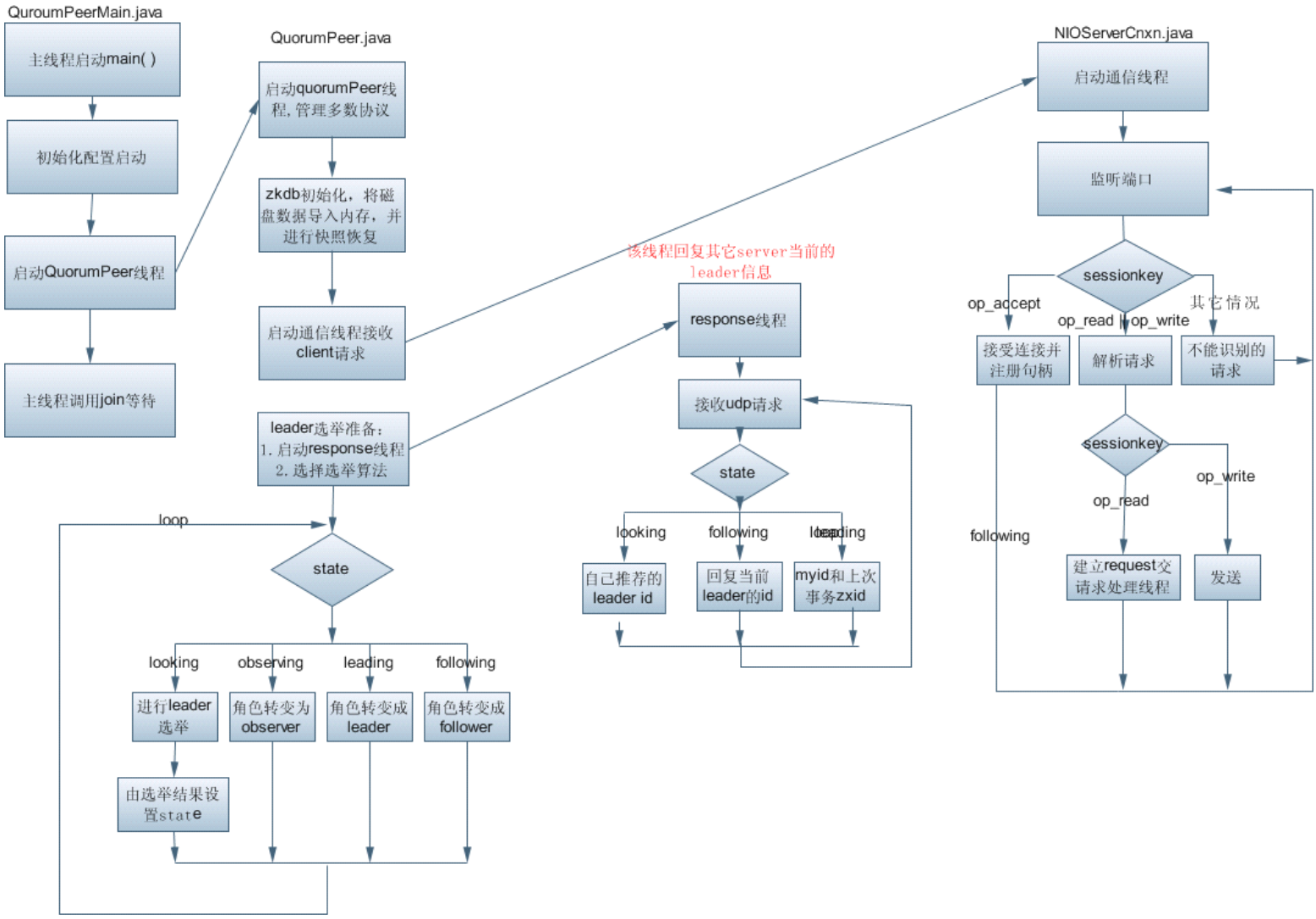
2）Follower连接leader，将最大的zxid发送给leader；

3）Leader根据follower的zxid确定同步点；

4）完成同步后通知follower 已经成为uptodate状态；

5）Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。

三、 ZookeeperServer 工作流程



3.1 主线程的工作

1）刚开始时各个Server处于一个平等的状态peer；

2）主线程加载配置后启动；

3）主线程启动QuorumPeer线程，该线程负责管理多数协议（Quorum），并根据表决结果进行角色的状态转换；

4）然后主线程等待QuorumPeer线程。

3.1.1 QuorumPeer 线程

1）首先会从磁盘恢复zkdatabase（内存数据库），并进行快照恢复；

2）然后启动server的通信线程，准备接收client的请求；

3）紧接着该线程进行选举leader准备，选择选举算法，启动response线程（根据自身状态）向其他server回复推荐的Leader；

4）刚开始的时候server都处于looking状态，进行选举根据选举结果设置自己的状态和角色。

3.1.2 quorumPeer 的状态

1）Looking: 寻找状态，这个状态不知道谁是leader，会发起leader选举；

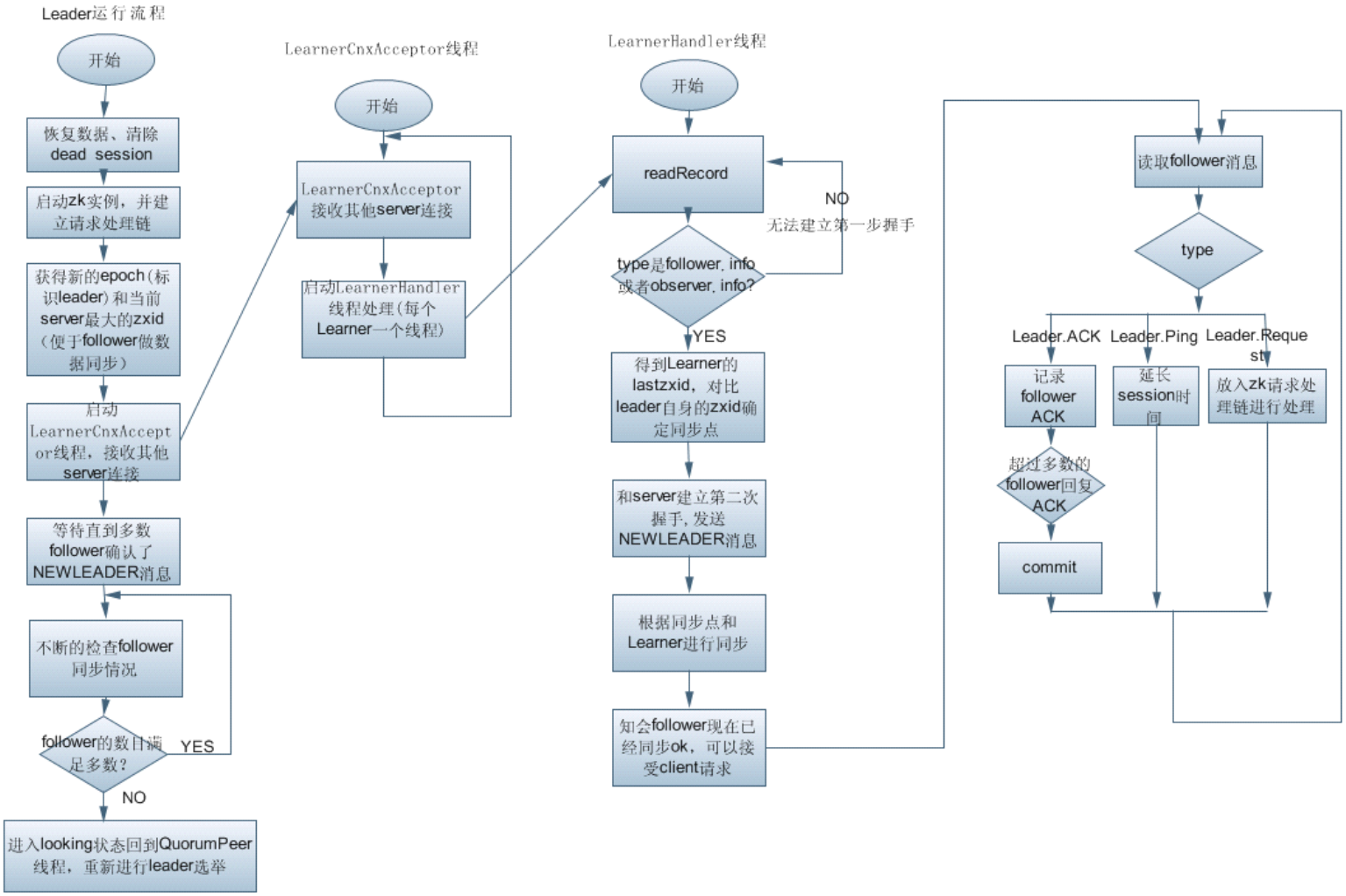
2）Observing: 观察状态，这时候observer会观察leader是否有改变，然后同步leader的状态；

3）Following: 跟随状态，接收leader的proposal ，进行投票。并和leader进行状态同步；

4）Leading: 领导状态，对Follower的投票进行决议，将状态和follower进行同步。

当一个Server发现选举的结果自己是Leader把自己的状态改成Leading，如果Server推荐了其他人为Server它将自己的状态改成Following；做Leader的server如果发现拥有的follower少于半数时，它重新进入looking状态，重新进行leader选举过程。（Observing状态是根据配置设置的）。

3.2 Leader 的工作流程



3.2.1 Leader 主线程

1）首先leader开始恢复数据和清除session。启动zk实例，建立请求处理链(Leader的请求处理链)：PrepRequestProcessor->ProposalRequestProcessor->CommitProcessor->Leader. ToBeAppliedRequestProcessor ->FinalRequestProcessor；

2）得到一个新的epoch，标识一个新的leader , 并获得最大zxid（方便进行数据同步）；

3）建立一个学习者接受线程（来接受新的followers的连接，follower连接后确定followers的zxvid号，来确定是需要对follower进行什么同步措施，比如是差异同步(diff)，还是截断（truncate）同步，还是快照同步）；

4）向follower建立一个握手过程leader->follower NEWLEADER消息，并等待直到多数server发送了ack；

5）Leader不断的查看已经同步了的follower数量，如果同步数量少于半数，则回到looking状态重新进行leaderElection过程，否则继续step5。

3.2.2 LearnerCnxAcceptor 线程

1）该线程监听Learner的连接；

2）接受Learner请求，并为每个Learner创建一个LearnerHandler来服务。

3.2.3 LearnerHandler 线程的服务流程

1）检查server来的第一个包是否为follower.info或者observer.info，如果不是则无法建立握手；

2）得到Learner的zxvid，对比自身的zxvid，确定同步点；

3）和Learner建立第二次握手，向Learner发送NEWLEADER消息；

4）与server进行数据同步；

5）同步结束，知会server同步已经ok，可以接收client的请求；

6）不断读取follower消息判断消息类型：

- 如果是LEADER.ACK,记录follower的ack消息，超过半数ack，将proposal提交(Commit)；
- 如果是LEADER.PING，则维持session（延长session失效时间）；
- 如果是LEADER.REQEST，则将request放入请求链进行处理--Leader写请求发起proposal，然后根据follower回复的结果来确定是否commit的。最后由FinallRequestProcessor来实际进行持久化，并回复信息给相应的response给server。

3.3 Follower 的工作流程

1）启动zk实例，建立请求处理链:FollowerRequestProcessor->CommitProcessor->FinalProcessor；

2）follower首先会连接leader，并将zxid和id发给leader；

3）接收NEWLEADER消息，完成握手过程；

4）同leader进行状态同步；

5）完成同步后，follower可以接收client的连接；

6）接收到client的请求，根据请求类型

- 对于写操作, FollowerRequestProcessor会将该操作作为LEADER.REQEST发给LEADER由LEADER发起投票。
- 对于读操作，则通过请求处理链的最后一环FinalProcessor将结果返回给客户端。

对于observer的流程不再赘述，observer流程和Follower的唯一不同的地方就是observer不会参加leader发起的投票。

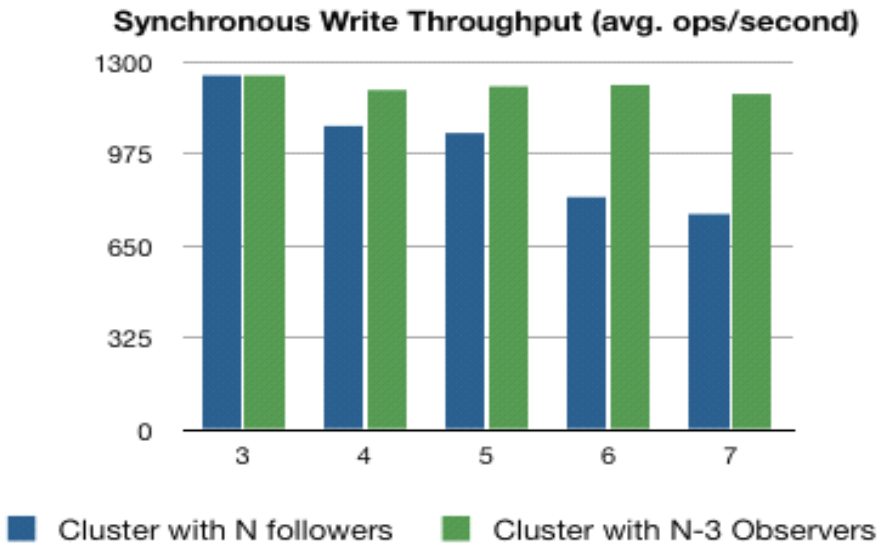
四、 关于 Zookeeper 的扩展

为了提高吞吐量通常我们只要增加服务器到Zookeeper集群中。但是当服务器增加到一定程度，会导致投票的压力增大从而使得吞吐量降低。因此我们引出了一个角色：Observer。

Observers 的需求源于ZooKeeper follower服务器在上述工作流程中实际扮演了两个角色，它们从客户端接受连接与操作请求，之后对操作结果进行投票。这两个职能在 ZooKeeper集群扩展的时候彼此制约。如果我们希望增加 ZooKeeper 集群服务的客户数量（我们经常考虑到有上万个客户端的情况），那么我们必须增加服务器的数量，来支持这么多的客户端。然而，从一致性协议的描述可以看到，增加服务器的数量增加了对协议的投票部分的压力。领导节点必须等待集群中过半数的服务器响应投票。于是，节点的增加使得部分计算机运行较慢，从而拖慢整个投票过程的可能性也随之提高，投票操作的会随之下降。这正是我们在实际操作中看到的问题——随着 ZooKeeper 集群变大，投票操作的吞吐量会下降。

所以需要增加客户节点数量的期望和我们希望保持较好吞吐性能的期望间进行权衡。要打破这一耦合关系，引入了不参与投票的服务器，称为 Observers。 Observers 可以接受客户端的连接，将写请求转发给领导节点。但是，领导节点不会要求 Observers 参加投票。相反，Observers 不参与投票过程，仅仅和其他服务节点一起得到投票结果。

这个简单的扩展给 ZooKeeper 的可伸缩性带来了全新的镜像。我们现在可以加入很多 Observers节点，而无须担心严重影响写吞吐量。规模伸缩并非无懈可击——协议中的一步（通知阶段）仍然与服务器的数量呈线性关系。但是，这里的串行开销非常低。因此可以认为在通知服务器阶段的开销无法成为主要瓶颈。



上图显示了一个简单评测的结果。纵轴是从一个单一的客户端发出的每秒钟同步写操作的数量。横轴是 ZooKeeper 集群的尺寸。蓝色的是每个服务器都是 voting 服务器的情况，而绿色的则只有三个是 voting 服务器，其它都是 Observers。图中看到，扩充 Observers，写性能几乎可以保持不变，但如果同时扩展 voting 节点的数量的话，性能会明显下降。显然 Observers 是有效的。因此Observer可以用于提高Zookeeper的伸缩性。

此外Observer还可以成为特定场景下，广域网部署的一种方案。原因有三点：1.为了获得更好的读性能，需要让客户端足够近，但如果将投票服务器分布在两个数据中心，投票的延迟太大会大幅降低吞吐，是不可

取的。因此希望能够不影响投票过程，将投票服务器放在同一个IDC进行部署，Observer可以跨IDC部署。2.投票过程中，Observer和leader之间的消息、要远小于投票服务器和server的消息，这样远程部署对带宽要求就较小。3.由于Observers即使失效也不会影响到投票集群，这样如果数据中心间链路发生故障，不会影响到服务本身的可用性。这种故障的发生概率要远高于一个数据中心中机架间的连接的故障概率，所以不依赖于这种链路是个优点。

ZooKeeper 实现共享锁

陈旭日*

大家也许都很熟悉了多个线程或者多个进程间的共享锁的实现方式了，但是在分布式场景中我们会面临多个Server之间的锁的问题。

假设有这样一个场景：两台server ：serverA，serverB需要向C机器上的/usr/local/a.txt文件，如果两台机器同时写该文件，那么该文件的最终结果可能会产生乱序等问题。最先能想到的是serverA在写文件前告诉ServerB “我要开始写文件了，你先别写”，等待收到ServerB的确认回复后ServerA开始写文件，写完文件后再通知ServerB“我已经写完了”。假设在我们场景中有100台机器呢，中间任意一台机器通信中断了又该如何处理？容错和性能问题呢？要能健壮，稳定，高可用并保持高性能，系统实现的复杂度比较高，从头开发这样的系统代价也很大。幸运的是，我们有了基于google chubby原理开发的开源的ZooKeeper系统。接下来本文将介绍两种ZooKeeper实现分布式共享锁的方法。

一、 利用节点名称的唯一性来实现共享锁

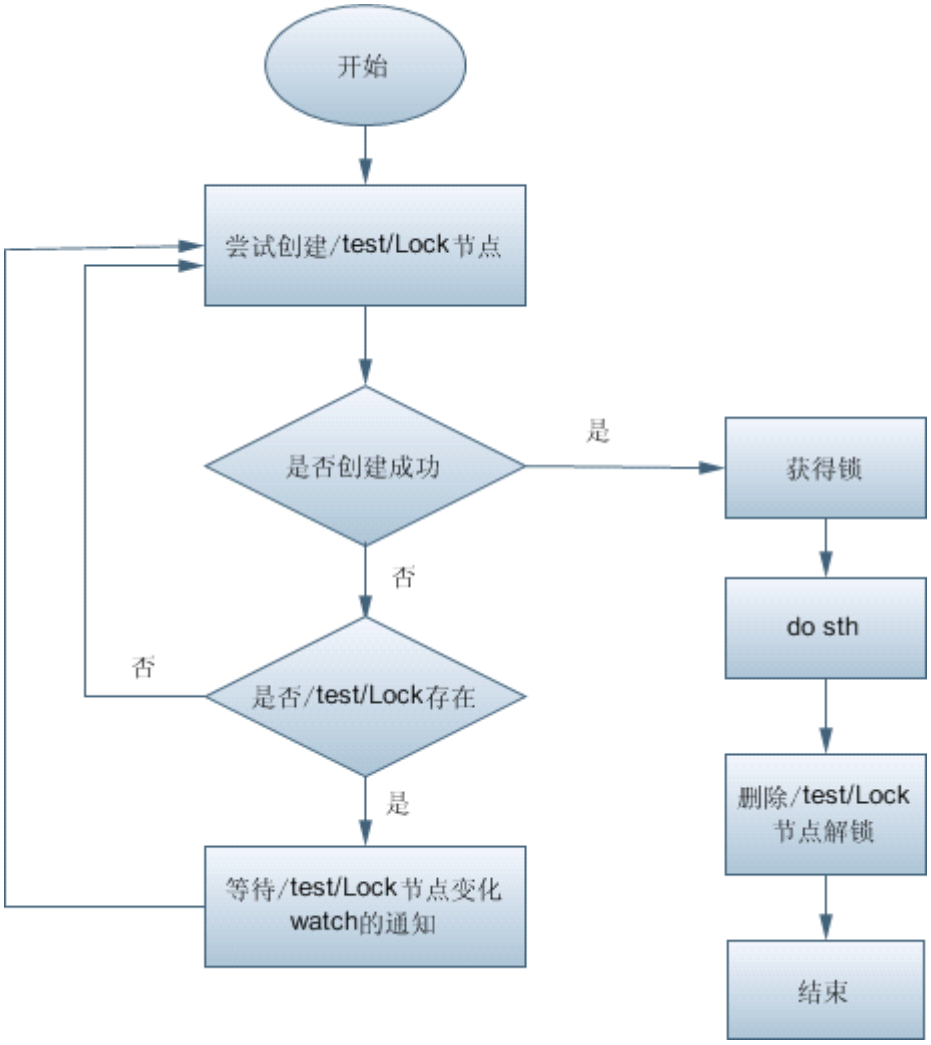
ZooKeeper表面上的节点结构是一个和unix文件系统类似的小型的树状的目录结构，ZooKeeper机制规定：同一个目录下只能有一个唯一的文件名。

例如：我们在Zookeeper目录/test目录下创建，两个客户端创建一个名为Lock节点，只有一个能够成功。

算法思路：利用名称唯一性，加锁操作时，只需要所有客户端一起创建/test/Lock节点，只有一个创建成功，成功者获得锁。解锁时，只需删除/test/Lock节点，其余客户端再次进入竞争创建节点，直到所有客户端都获得锁。

基于以上机制，利用节点名称唯一性机制的共享锁算法流程如图所示：

作者简介：陈旭日，吉林大学硕士，兴趣领域：数据库技术、分布式系统、负载均衡技术。对 ZooKeeper 的原理和应用有较深入研究。联系方式：julianchen888 at gmail.com



使用该方法进行测试锁的性能列表如下：

Lock机制的互斥测试(5PC)		
互斥进程数	平均进程完成时间（ms）	所有进程都完成所用时间（ms）
5	14.4	23
10	45.6	71
20	141.85	207
50	612	882

二、 利用顺序节点实现共享锁

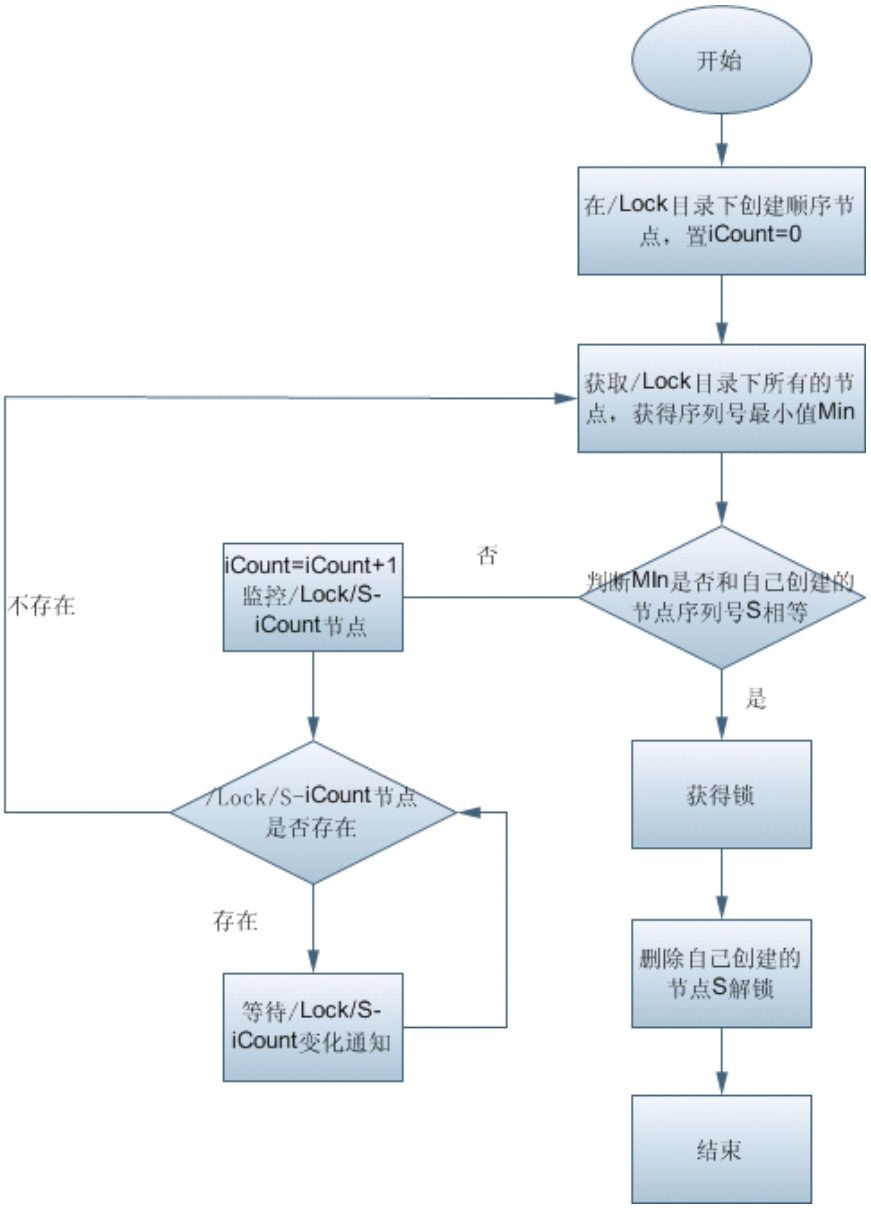
首先介绍一下，Zookeeper中有一种节点叫做顺序节点，故名思议，假如我们在/lock/目录下创建节3个点，ZooKeeper集群会按照提起创建的顺序来创建节点，节点分别为/lock/0000000001、/lock/0000000002、/lock/0000000003。

ZooKeeper中还有一种名为临时节点的节点，临时节点由某个客户端创建，当客户端与ZooKeeper集群断开连接，则开节点自动被删除。

算法思路：对于加锁操作，可以让所有客户端都去/lock目录下创建临时、顺序节点，如果创建的客户端发现自身创建节点序列号是/lock/目录下最小的节点，则获得锁。否则，监视比自己创建

节点的序列号小的节点（当前序列在自己前面一个的节点），进入等待。解锁操作，只需要将自身创建的节点删除即可。

具体算法流程如下图所示:



Lock机制的互斥测试		
互斥进程数	平均进程完成时间（ms）	所有进程都完成所有时间(ms)
5	3.56	7
10	5.32	8.6
50	47.24	78.4

该算法只监控比自身创建节点序列号的次小节点，在当前获得锁的节点释放锁的时候,没有“惊群”效应。

Hadoop 最佳实践

万乐*

一、 简介

Hadoop是Apache自由软件基金会资助的顶级项目，致力于提供基于map-reduce计算模型的高效、可靠、高扩展性分布式计算平台。

二、 Map-Reduce 应用场景

作为一种受限的分布式计算模型，Map-Reduce计算模型有其擅长的领域，也有其不擅长的方面：

条款1：map-reduce计算模型适用于批处理任务，即在可接受的时间内对整个数据集计算某个特定的查询的结果，该计算模型不适合需要实时反映数据变化状态的计算环境。

条款2：map-reduce计算模型是以“行”为处理单位的，无法回溯已处理过的“行”，故每行日志都必须是一个独立的语义单元，行与行之间不能有语义上的关联。

条款3：相对于传统的关系型数据库管理系统，Map-Reduce计算模型更适合于处理半结构化或无结构话的数据。

因为Map-Reduce计算模型是在处理的时候对数据进行解释的，这就意味着输入的Key和Value可以不是数据本身固有的属性，Key、Value的选择完全取决于分析数据的人。

条款4：Map-Reduce是一个线性可扩展模型，服务器越多，处理时间越短。

以下是同一个任务在不同机器数下获得的测试结果：

机器数	30 台	40 台	50 台
200G 任务运行时间 (分)	101	77	64

三、 任务调度优化

首先对一些术语进行一下说明。Job是一组客服端想要完成的工作，包括输入数据，map-reduce程序以及配置信息，Hadoop通过将Job划分为一些task来执行，task又分为map task和reduce task。

如何调度Hadoop任务才能充分发挥集群中所有服务器的能力呢？

条款5：每个Job的输入文件不宜过大，也不宜过小。文件过大会造成reduce任务分布不均匀，导致reduce time的不可预知性，而大量的小文件则会严重影响Hadoop的性能。

Hadoop会将Job的输入文件分割成64M固定大小的split，每个split启动一个map task处理，这个split中的每个record都经过用户定义的map函数处理生成中间结果。若输入文件小于64M，则此文件单独作为一个split处理。故当输入文件中有大量的小文件时，那么管理这些小文件的开销以及map task的创建开销会占据绝大多数的Job执行时间。

作者简介：万乐，2008 年硕士毕业，主要从事高性能服务器和分布式系统开发，对一切有挑战的事情感兴趣。
联系方式：wanle0626 at 163.com

为了找到Hadoop合适的Job文件大小，我们在一个有50台退役机器组成的集群做了一组性能测试，结果如下表：

文件大小	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
reduce shuffle time（分钟）	9	18	27	37	45	54	62	69	77	86
reduce time(分钟)	12	16	23	27	39	46	62	89	99	109
计算总时间（分钟）	21	34	50	64	84	100	124	158	176	195

我们把一个任务的计算时间分为两部分：reduce shuffle time和reduce time。

- reduce shuffle time是reduce任务把map输出的<key，value>对copy到本地的时间，即reduce shuffle time=map时间+<key，value>对网络传输时间。
- reduce time就是reduce处理这些<key，value>对的时间。

从上表我们可以得出结论：

- 各个任务的reduce shuffle time是完全线性的（随着任务量增加，时间线性增加）。
- 任务量在300G以内，reduce time基本线性增长，之后随着任务量增加，reduce time呈现随机性加大的趋势。在任务量达到550G后这种随机性更加明显，先后运行同样的任务时间可能会相差一个小时。可以推断，随着任务量增加，reduce任务分布不均匀的机率提高，导致了reduce time的不可预知性。
- 上面两个时间的叠加影响下，在300G以内退役机器处理任务的时间是线性增加的。300G以上的任务需要分成若干个小任务串行运行，保证reduce处理在线性可控的区间内。

条款6：多个大输入的Job建议使用串行执行，多个小输入的Job建议使用并行执行。

Hadoop的任务处理分为map阶段以及reduce阶段，当集群的task slots足够支持多个任务同时执行时，建议使用多任务并行执行，反之，建议使用串行执行，且当一个Job开始执行reduce task时，可以开始执行下一个Job的map task。

以下是我们在50台退役机器上分别并行和串行运行2个100G，200G，300G的任务的测试结果：

文件大小	50G	100G	200G	300G
并行运行二个任务总时间（分）	26	53	140	211
串行运行二个任务总时间（分）	41	68	128	200
并行串行时间差（分）	-15	-15	+12	+11

条款7：reducer的个数应该略小于集群中全部reduce slot的个数。

map task的个数由输入文件大小决定，所以选择合适的reducer的个数对充分利用Hadoop集群的性能有重要的意义。

Hadoop中每个task均对应于tasktracker中的一个slot，系统中mapper slots总数与reducer slots总数的计算公式如下：

mapper slots总数 = 集群节点数 × mapred.tasktracker.map.tasks.maximum

reducer slots总数 = 集群节点数 × mapred.tasktracker.reduce.tasks.maximum

设置reducer的个数比集群中全部的reducer slot略少可以使得全部的reduce task可以同时进行，而且可以容忍一些reduce task失败。

条款8：多个简单串行的Job优于一个复杂的Job。

将复杂的任务分割成多个简单的任务，这是典型的分治的思想。这样不仅可以使得程序变得更简单，职责更单一，而且多个串行的任务还可以在上一个任务的正在执行reduce任务的时候，利用空闲的map资源来执行下一个任务。

四、 Key-Value 权衡

Map-Reduce 算法的核心过程如下：

map (k1,v1) --> list(k2,v2)

reduce (k2,list(v2)) --> list(v2)

即通过用户定义的 map 函数将输入转换为一组<Key，Value>对，而后通过用户定义的 reduce 函数将<Key，List<Value>>计算出最后的结果。

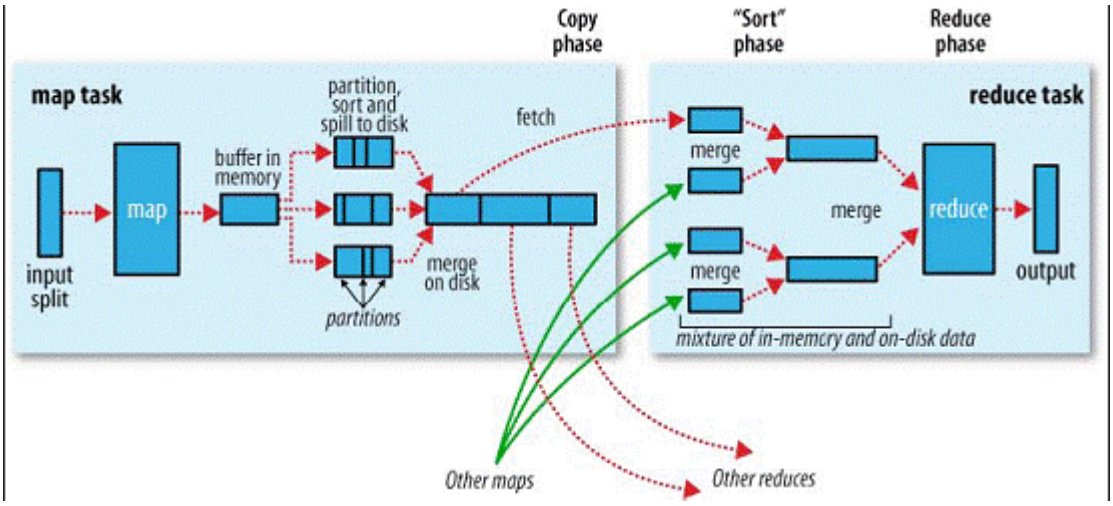
如何选择合适的 map 和 reduce 函数才能充分利用 Hadoop 平台的计算能力呢？换句话说，如何选择上式中合适的 K2 和 V2 呢？

条款 9：map task 或 reduce task 的大小应该适中，以一个 task 运行 2-3 分钟为宜，且 task 不能超出计算节点的运算能力。

虽然 Hadoop 平台帮助我们将数据分割成为小任务来执行，但我们也应当意识到，每个 task 都是在一个计算节点运行的，若一个 task 对机器资源（CPU、内存、磁盘空间等）的需求超出了计算节点的能力的话，任务将会失败。而如果 task 过小的话，虽然计算节点能够快速的完成 task 的执行，但过多的 task 的管理开销，以及中间结果频繁的网络传输将占据任务执行的绝大部分时间，这样同样会严重影响性能。建议的 task 大小最好是以能够运行 2-3 分钟为宜。

条款 10：map 产生的中间结果不宜过大。

输入数据经过用户定义的 map 函数后生成的<Key，Value>对是 Map-Reduce 模型的中间计算结果，如下图所示：



Map task将计算的中间结果保存在本地磁盘，而后通过Reduce task拉去所有当前任务所需的中间结果，并将中间结果按Key排序。显然若map产生的中间结果过大，网络传输时间以及中间结果排序将占据大部分的Job执行时间。

通过 Hadoop 的 API 管理 Job

江舟*

一、 背景

前些时候写了一篇这样的文档，由于当时的时间比较紧，且有些细节内容不太好细说，于是写的比较粗。之后也有些朋友发邮件给我，问我要详细的过程以及管理方式。于是，今天就像把这个内容细化，写在这里，供大家参考。

二、 环境简述

- 操作系统 Linux、JDK1.6
- Hadoop 0.20.2
- 开发工具选择 eclipse 3.3.2（可配合使用 hadoop 的插件）

三、 需求

首先还是要说一下需求，为什么要用 hadoop 的 API 来对 Job 进行管理。对此，我列举出了以下需求内容：

1、Job 之间有先后的顺序执行问题（一个业务可能是多个 Job 才能完成）。

2、需要对每个 Job 的状态进行监控（是否完成以及失败情况处理等等）

3、有些无先后依赖关系的 Job 可以并发执行。

4、每个 Job 执行时的信息采集和记录。

5、能够灵活添加或删除要执行的 Job 队列。

如果以上需求去掉 2 和 4，那么，我们通过脚本就可以做到（如 shell）。但是如果要获取 Job 的详细信息以及 Job 运行时的状态，那么还是需要调用 Hadoop 的 API 来实现。所以，这里我选择了 Java 语言来实现以上需求。

四、 设计思路

这里的设计必须要满足以上的需求（名字暂定为 pipeline），设计内容大体如下：

a) 通过周期的遍历时间，获得Job队列启动时间，并在启动之前加载Job配置文件。

b) 通过配置Job的列表来确定Job执行的先后顺序以及哪些可以并发哪些不能并发。

c) 通过JobClinet来采集相关的Job信息。

d) 对失败的Job有重新尝试执行的机制。

作者简介：江舟，五年 Java 工作经验，并做过 Web 和一些后台应用的开发。从 2010 年 2 月开始专注于与 Hadoop 相关的技术领域，目前正致力于 Hadoop 集群的部署、应用以及内部推广工作。
联系方式：dajuezhao at gmail.com

因为考虑到 pipeline 是和 Job 的 MR 代码是剥离的，不能存在于一个工程下，这样，才能实现 MR 的灵活增删。那么，我们还要设计如何通过 pipeline 来管理 MR 生成好的 JAR 包。下面，我们将就以上思路来逐步设计。

五、 配置文件

首先是配置文件，如果要满足以上的设计思路，那么需要 2 个配置文件。一个是 pipeline 自身的配置文件，包含了周期遍历时间、pipeline 启动时间、任务失败尝试次数、最大并发任务数、调度模式（FIFO 还是 FAIR）以及日志输出目录。将这个配置文件命名为 pipeline.ini，见下图：

```
#周期遍历时间，单位分钟，1-5
cycleTime=5
#任务启动时间。24小时制。2-23
startTime=20
#任务失败之后尝试次数。1-3
tryTime=2
#0是FIFO,1是FAIR
isFIFO=1
#同时并发的任务数。只有在FAIR模式下生效。1-3
maxJob=2
#日志输出目录
logDir=/run/log/
```

上面的是 pipeline 自己的配置文件，那么，接下来我们还要考虑，如何将 JAR 加载到 pipeline 中，从而按照我们制定的顺序启动。我们设计了一下配置文件，暂定 pipeline.joblist，这个配置文件通过序号来区分哪些任务是需要顺序执行，哪些任务是可以并发执行的。例如当前有 5 个任务，分别是 A,B,C,D,E。AB 是一个业务，CDE 是一个业务。那么配置文件如下（字段间以\t 划分）：

```
01001 A 后面是输入、输出以及自定义的一些参数。
01002 B 后面是输入、输出以及自定义的一些参数。
02001 C 后面是输入、输出以及自定义的一些参数。
02002 D 后面是输入、输出以及自定义的一些参数。
02003 E 后面是输入、输出以及自定义的一些参数。
```

可以通过编号看出，01 是一个业务，02 是一个业务。01 或者 02 开头的 job 是需要顺序执行的。但是开头不同的，是可以并发执行的。

六、 启动 Job

到这里为止，基本配置文件就设计完成了，后面我们需要考虑代码如何编写。其实，在这个过程中，就是一个 Timer 类，然后根据配置参数来做不停的时间遍历，直到达到了配置的启动时间，那么加载 pipeline.joblist 文件，然后执行 Job。以上的过程如果是 FIFO 模式，是很好实现的，直接的顺序执行就可以了。启动 job 实际上还是调用的 org.apache.hadoop.util.RunJar.main(args[])方法。

由于是在 Timer 下调用，所以如果执行 FIFO 模式的 Job，需要用到线程阻塞的模式来控制。关键代码如下：

```
// 线程阻塞，直到执行结束
jobClient.getJob(jobID).waitForCompletion();
```

上面的代码用了到 JobClient 和 JobID，后面我会说明。OK，到这里为止，FIFO 的调度模式已经比较清晰了。接下来我们说说 FAIR 的模式。由于 FAIR 涉及到了并发，所以，这里需要考虑采用多线程的模式来启动任务。在这里，我采用了 Semaphore 类。代码直接贴出来：

```
private void randRunJob(final ArrayList<ArrayList<String>> serJobList) {
        // 线程池
        ExecutorService exec_es = Executors.newCachedThreadPool();
        // 启动信号器
        for (int i = 0; i < serJobList.size(); i++) {
            final ArrayList<String> temp_list = serJobList.get(i);
            Runnable maxJob_runnable = new Runnable() {
                public void run() {
                    try {
                        ArrayList<String> job_list = temp_list;
                        // 注册一个信号计数器
                        semaphore.acquire();
                        Utils.RunJob(job_list, conf, jobStatus, jobClient);
                        // 释放这个信号计数器
                        semaphore.release();
                    } catch (InterruptedException e) {
                        Log.writeLog(e.getMessage(), Constant.systemLog_int);
                        return;
                    }
                }
            };
            exec_es.execute(maxJob_runnable);
        }
        // 退出线程池
        exec_es.shutdown();
    }
```

可以看到，输入的参数是两层list嵌套的结构。第一层list包含了2个list，里面分别存放的是A,B/C,D,E。OK，这样可以比较清晰的看到通过不同的信号量来启动2个第一层list里面的任务。

七、 Job 的监控

前面我们都说了如何加载 Job 列表以及采用不同的方式来启动 Job。这里，我将说下如何采用 JobClinet 来获取 Job 的一些信息。前提是已知 Job 的 name。那么，接下来我贴上几个函数。

```
    /**
     * 根据JobName获取JobID
     *
     * @param jobName_str
     * @return JobID
     */
    public static ~~JobID~~ getJobIDByJobName(JobClient jobClient, JobStatus[]
jobStatus, String jobName_str) {
        ~~JobID~~ jobID = null;
        try {
            for (int i = 0; i < jobStatus.length; i++) {
                RunningJob rj = jobClient.getJob(jobStatus[i].getJobID());
                if (rj.getJobName().trim().equals(jobName_str)) {
                    jobID = jobStatus[i].getJobID();
                    break;
                }
            }
        } catch (IOException e) {
            Log.writeLog(e.getMessage(), Constant.systemLog_int);
        }
        return jobID;
    }
```

```
    /**
     * 根据JobID获取Job状态
     *
     * @param jobClient
     * @param jobStatus
     * @param jobID
     * @return RUNNING = 1,SUCCEEDED = 2,FAILED = 3,PREP = 4,KILLED = 5
     * @throws IOException
     */
    public static String getStatusByJobID(JobClient jobClient, JobStatus[]
jobStatus, ~~JobID~~ jobID) throws IOException {
```

```
        int status_int = 0;
        jobStatus = jobClient.getAllJobs();
        for (int i = 0; i < jobStatus.length; i++) {
            if (jobStatus[i].getJobID().getId() == jobID.getId()) {
                status_int = jobStatus[i].getRunState();
                break;
            }
        }

        String desc_str = "";
        switch (status_int) {
        case 1:
            desc_str = "RUNNING";
            break;
        case 2:
            desc_str = "SUCCEEDED";
            break;
        case 3:
            desc_str = "FAILED";
            break;
        case 4:
            desc_str = "PREP";
            break;
        case 5:
            desc_str = "KILLED";
            break;
        default:
            break;
        }
        return desc_str;
    }
```

```
/**
     * 获取正在运行的JobID的列表
     *
     * @param jobClient
     * @return ArrayList<JobID>
     */
    public static ArrayList<JobID> getRunningJobList(JobClient jobClient) {
        ArrayList<JobID> runningJob_list = new ArrayList<JobID>();
        JobStatus[] js;
        try {
            js = jobClient.getAllJobs();
            for (int i = 0; i < js.length; i++) {
                if (js[i].getRunState() == JobStatus.RUNNING ||
js[i].getRunState() == JobStatus.PREP) {
                    runningJob_list.add(js[i].getJobID());
                }
            }
        } catch (IOException e) {
            Log.writeLog(e.getMessage(), Constant.systemLog_int);
        }
        return runningJob_list;
    }
```

上面的类还有很多的其他的API可以调用，查看API文档就可以看到。其实通过API还可以做到对某一个TASK的监控，不仅仅是Job的监控。

Hadoop 集群的配置调优

江舟*

一、 背景

Hadoop 的集群使用也有些时候了，不过都是小集群的使用（数量小于 30 台）。在这个过程中不断的进行着配置调优的操作。最早的时候，在网上也没有找到一些合适的配置调优方法，于是，我在这里列举一些基本的调优配置，以供参考。最后，还有一个我自己的测试环境的配置说明，也可以参看一下。

二、 环境和配置

1、 版本和环境

- Hadoop 版本：apache 发布的 0.21
- 操作系统：Linux
- JDK：1.6
- 网络环境为千兆网络

2、 hdfs-site.xml 配置文件

- **dfs.block.size**

这个是块大小的设置，也就是说文件按照多大的 size 来切分块。一般来说，块的大小也决定了你 map 的数量。举个例子：我现在有一个 1T 的文件，如果我的块 size 设置是默认的 64M，那么在 HDFS 上产生的块将有 1024000/64=16000 块。

如果我们以 TextInputFormat 来处理该 1T 的文件，那么将会产生 16000 个 map 来处理。这样的多的 map 明显是不合理的。所以，如果我们将 block 的 size 设置成 512M，那么，将 1T 的文件作为输入文件，将产生 2000 个 map，计算的时候效率将提升不少。

因此，block size 的大小是需要根据输入文件的大小以及计算时产生的 map 来综合考量的。一般来说，文件大，集群数量少，还是建议将 block size 设置大一些的好。

- **dfs.replication**

复制数量的设置，不能为 0。这个名字说实话很迷惑人，最开始的时候我以为就是备份数量，后来才发现，原来就是存放数据文件的份数。

作者简介：江舟，五年 Java 工作经验，并做过 Web 和一些后台应用的开发。从 2010 年 2 月开始专注于与 Hadoop 相关的技术领域，目前正致力于 Hadoop 集群的部署、应用以及内部推广工作。
联系方式：dajuezhao at gmail.com

设置为 1，就是在集群中存一份。如果设置为 2，即做一份备份，也就是说数据在集群中有 2 份。还是以 1T 的数据为例，如果设置 1，集群中就存在 1T 的文件，如果设置为 2，那么集群占用空间为 2T。

当然，这个备份还有个基于机架感知的备份机制（本地存放、同机架存放、异机架存放）如果不配置机架，默认都在一个机架上，之所以做机架感知的备份就是为了做到异地容灾。因为我现在的集群都在一个机房，就是做了机架配置，也不能做异地容灾。要是机房断电，谁也不能容灾谁。但是如果你的集群分别存放在 2 个机房，还是可以考虑做机架配置，然后将备份数量设置为 3。

备份数量有利有弊，备份数量多，节点挂个几个没影响，数据依然完整。但是你的冗余数据会增加。如果只设置为 1，那就是节点一旦有挂掉的，就是 block miss。一般来说，测试情况下建议设置为 1，如果实际使用，所有机器都在一个机房，建议配置为 2，如果集群分布在不同机房，还是试试配置为 3 吧。

3、 mapred-site.xml 配置文件

- **mapred.tasktracker.map.tasks.maximum 和 mapred.tasktracker.reduce.tasks.maximum**

这个 2 个参数分别是用来设置的 map 和 reduce 的并发数量。实际作用就是控制同时运行的 task 的数量。这 2 个参数实际上在配置的时候是需要结合计算节点的硬件配置以及任务调度模式来配置的。

举个例子吧。我现在有 5 台机器，1 台 master，4 台 slave，配置都是 2 个 4 核 CPU，8G 内存，1T 硬盘。我配置任务调度模式是默认的 FIFO 模式。在这样的模式下，我配置并发 map 为 6，并发的 reduce 为 2。 其实这样配置不难看出，map 和 reduce 的并发数就等于 CPU 的总核数。

前段时间看到网上有篇文章说 map 和 reduce 的并发数应该设置为相同。最后我考虑了一下，如果使用 fair 的调度模式，设置成相同，应该是可以的，但是如果是 FIFO 模式，我个人认为在 map 或是 reduce 阶段，CPU 的核数没有得到充分的利用，有些可惜，所以，FIFO 模式下，还是尽量配置的 map 并发数量多于 redcue 并发数量。

因此，我说这个参数的配置不仅仅要考虑硬件配置，还需要考虑到 Job 的调度模式。需要说明的是，这个配置参数不同的节点可以配置不同。适用于硬件异构的集群。见下图：

Host	# running tasks	Max Map Tasks	Max Reduce Tasks	Failures	Seconds since heartbe
hadoopSlave239	3	8	8	0	1
hadoopSlave206	15	16	16	0	1
hadoopSlave240	5	8	8	0	1
hadoopSlave203	9	16	16	0	0
hadoopSlave207	7	16	16	0	0
hadoopSlave205	8	16	16	0	1

- **mapred.child.java.opts**

这个参数是配置每个 map 或 reduce 使用的内存数量。默认的是 200M。对于这个参数，我个人认为，如果内存是 8G，CPU 有 8 个核，那么就设置成 1G 就可以了。实际上，在 map 和 reduce 的过程中对内存的消耗并不大，但是如果配置的太小，则有可能出现”无可分配内存”的错误。所以，

对于这个配置我总结了一个简单的公式：map/reduce 的并发数量(总和不大于 CPU 核数)× mapred.child.java.opts < 该节点机器的总内存。当然也可以等于，不过有点风险而已。

- **mapred.reduce.tasks**

设置 reduce 的数量。一般来说在 job 里面都会通过 conf 来设置 reduce 的数量，不采用这个参数。至于 reduce 的数量，可以根据自己的 reduce 业务逻辑复杂度以及输出的数据量来调整。

4、 core-sit.xml 配置文件

- **webinterface.private.actions**

这个参数实际上就是为了方便测试用。允许在 web 页面上对任务设置优先级以及 kill 任务。需要注意的是，kill 任务是个缓慢的过程，它需要杀掉所有的任务 task 然后才是任务结束。如果 task 数量多，可能有点慢，需要一些耐心等待。

三、 总结

需要说明一下，配置文件的加载顺序是：先加载默认的配置项（看看 default 文件就知道默认配置项了），然后加载 site 文件里的配置项，最后加载 MR 代码里的配置项。所以，个性化的配置还是放在 MR 代码中通过 cong.set 方法来设置比较合适。

其他的还有一些配置，例如 task 心跳响应延迟时间（默认 10 分钟）、task 失败尝试次数等等，这个有兴趣可以改改试下。对于集群，hadoop 还自带一些测试的工具，例如集群的 I/O 测试，这个是最常用的。用法一般为 hadoop jar hadoop-mapred-test-0.21.0.jar TestDFSIO -write -nrFiles 10 -fileSize 1000。执行完成之后会在同级目录下有 TestDFSIO_results.log 文件，查看就行。

四、 附件

这个是我的测试环境，大家可以参考一下：

1）数据量：1.3T文本文件，约31亿条数据，每条数据70个字段。

2）业务：过滤所有数据，然后去重排序。

3）硬件：集群共5台设备（8G内存、2个4核CPU、1T硬盘、千兆网卡，千兆交换机器）。

4）软件配置见下表：

字段	值
blocksize	512M
调度模式	FIFO（就一个任务）
mapred.child.java.opts	1024M
mapred.tasktracker.map.tasks.maximum	6
mapred.tasktracker.reduce.tasks.maximum	2
Reduce 数量	200
dfs.replication	1（硬盘空间不够，就设置了 1）

5）任务耗时：3小时30分钟左右。

Hadoop平台的Java规范及经验

李邕*

一、 设计规范

规范 1-1：Hadoop 平台的 jar 程序调用两种方式：

说明：

1、适合简单逻辑，只有一个 Map/Reduce 形式

示例：

./hadoop org.apache.hadoop.mapred.JobClient -submit ./toolbar_tubi.xml

2、通用的逻辑，自己写 driver 主调函数，适合多个 Map/Reduce 串行调度，及复杂的配置解析程序。

示例：

./hadoop jar HadoopRDT.jar predict completed ./instance/configuration-rdt.xml.gdtday.20110216

规范 1-2：M/R 是基于 key/Value 模式设计的数据处理平台，设计前，必须了解数据的分布，尽量把 key 值打散。

规范 1-3：复杂任务，要先进行任务分解，分成多个 M/R 来计算，封装成不同的 java 类；

二、 使用规范

规范 2-1：在 M/R 里要有过滤和判断脏数据的逻辑，并记录在 counter 中；

示例：

```
if (strs.length != i_allColumnNum[i_tableIndex])
{
    reporter.incrCounter(MapperCounter.BadRow, 1);
    return;
}
```

规范 2-2：在 M/R 里要判断参数初始化是否正常，不正常时根据需要返回异常；

规范 2-3：在 configure 里出错，要在 M/R 里来判断和抛异常；

说明：M/R 系统架构会先执行 configure，再调用 Map/Reduce，即使 configure 返回失败，也会

作者简介：李邕。
联系方式：liyongmnls at gmail.com

继续执行 M/R,为减少 M/R 的执行错误或浪费性能，需要在 M/R 程序初始判断配置是否获取正常。

规范 2-4：用 main 函数接入管理台的情况，不要使用构造函数来初始化，可在 run 方法里初始化；

说明：管理台会执行自己默认的初始化。

规范 2-5：程序尽量不使用 hdfs 上自定义的文件名；

说明：文件名可能会随着系统的异动（如重启或其他平台）而改变

规范 2-6：在 M/R 模块里禁止使用 System.out.println 输出操作，否则可能撑爆磁盘。

说明：M/R 程序多为大数据量处理程序，每条记录输出日志会产生巨大的数据，并且分散在每台 hdfs 机器上，不便于查看。

建议 2-1：使用 cache file 或 File System 来读一个小文件，能提高系统性能；

说明：对于某些小的配置文件或者关联的小表，提前放入每台机器的内存，可以有效的节省计算资源或减少计算复杂度。

建议 2-2：当利用前一个 M/R 的结果作为 cache file 的内容时，最好用目录，而不是指定文件名，具体的文件过滤可在 configure 里判断，或使用 file system 来读

说明：hadoop 生成的文件名可能因为平台不同或者机器配置不同而导致文件名输入规则有变动。

建议 2-3：有些 reduce 的运行时间太长，需要自己添加 reporter.progress 来表明 reduce 的状态，若无法报告，可以把心跳时间根据需要调大；

建议 2-4：Counter 很多的话，可以通过 counter group 来管理；

建议 2-5：map 或 reduce 输出时，使用 set 方法可提高性能。

示例：

```
Text text = new Text（）；
text.set("")；
```

建议 2-6：map/Reduce 函数中，使用合适的 key 写类型，可能提升效率。

示例：

```
Mapper<LongWritable, Text, Text, Text>
```

建议 2-7：数据量大时，使用压缩文件存储，能有效节省 IO 操作时间。

示例：

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

建议 2-8：如果对输入输出性能要求高，map 和 reduce 输出文件时，可以使用 SequenceFile 和 MapFile。

说明：SequenceFile 和 MapFile 都是二进制文件格式。读写速度快，hadoop 提供专门的读写技术。

示例：

```
SequenceFile.Writer writer =
   new SequenceFile.createWriter(fs,conf,path,key.getClass(),Value.getClass())
writer.append(key,value);
MapFile.Writer writer = new MapFile.Writer(conf,fs,path,key.getClass(),Value.getClass());
writer.append(key,value);
```

建议 2-9：如果 Map 输入中重复 Key 很多时，使用 Combiner 可以提高性能。

说明：使用 combine 的本质是使用本地的 io 和计算来换取网络传输性能。是否使用 combine 与 key 重复量大小有关，如果 key 重复量小，使用 combine 反而会降低系统性能。

建议 2-10：尽量使用大文件作为文件输入，小文件个数太多，会严重降低系统性能。

说明：大量的小文件的打开会增加系统开销。使用 multipleInputs 多个文件一起读，或者前期控制文件大小和个数极大提升系统性能。

建议 2-11：尽量只使用文件夹名区分数据，使用系统自带的输出命令，文件名使用系统默认 part-*。用自定义输出方式，RecordWriter，FsOutputStream 等都有潜在写失败风险。

示例：

```
collect(keyText, valueText);
```

建议 2-12：main 函数最好有判断 M/R 执行成功与否的状态，并根据状态判断是否重做。

示例：

```
try {
    rj = JobClient.runJob(jConf);
    System.out.println(s_joinTime + " join Finished!" + rj.getJobState());
} catch (RuntimeException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

三、 配置规范

规范 3-1：配置文件尽量采用 xml 格式，使用 hadoop 自带的 configure 命令解析。

规范 3-2：map 和 reduce 的个数设置为机器数的 0.95 或 1.75 倍，可以提高性能。

规范 3-3：如果只需要 map，不需要 reduce 时，需要设置 reduce 个数为 0（不设置的默认值是 1）。

示例：

```
jConf.setNumReduceTasks(0);
```

规范 3-4：如果不需要系统输出 part-*格式的文件，自己调用 Writer 写文件，需要设置 output.format.class 为 nullOutputFormat

示例：

```
jConf.set("mapred.output.format.class","org.apache.hadoop.mapred.lib.NullOutputFormat");
```

规范 3-5：对于内存需求大的程序，需要设置内存大小，建议不要高于 1G，留部分给系统使用。

示例：

```
jConf.set("mapred.child.java.opts",-Xmx1000m);
```

四、 测试

规范 4-1：分布式平台运行的通常是大数据量程序，数据、机器存在多种可能，程序发布前需要经过大数据量测试。

规范 4-2：分布式平台运行机器性能差异，可能存在 cpu 和内存消耗过度，导致异常，需要在同类型机器平台上测试才能发布。

建议 4-1：多用 reporter 来测试和 debug，但在正式运营时少用，因为会增加通讯开销；

五、 其他

规范 5-1：分布式处理数据量巨大，机器配置不一，为保证系统正常运行，需要定期清理临时文件或过期历史数据。

MapReduce 开发经验总结

李邕*

一、 MapReduce 开发的优化点和注意事项

1）在开发之前选择一个高级语言，比如：hive，pig，java，streaming，或使用现有的开源code，比如mahout等；

2）map和reduce输出文件时，尽量使用SequenceFile和MapFile，或使用压缩文件，可以节约cpu成本和数据网络传输量；

3）尽量使用combiner（或在map中实现combiner功能），可以提高性能；

4）选定map和reduce的个数，可以提高性能，一般reduce个数为机器数的0.95或1.75倍；

5）尽量使用大文件作为输入，避免使用太多的小文件作为输入，因为会增加shuffle的次数消减性能；

6）使用cache file或File System来读一个小文件（文件大小小于一个block size（如64M））；

7）多用reporter来测试和debug，但在正式运营时少用，因为会增加通讯开销；

8）调整缓冲区大小来避免频繁的IO开销；

9）数据太大，集群无法甲酸时，要分批计算；

10）复杂任务，要先进行任务分解，分成多个M/R来计算；

11）多使用Hadoop的tool参数接口，比如：reduce个数，cache file等等；

12）使用合适的writable，比如：IntWritable等；

13）（高级应用）使用partition来处理方便的处理数据，比如用compositeInputFormat来读，在每个map上就可以做join的操作了；

二、 在 Hadoop 上开发时遇到的问题和解决方法

1）不要使用hard code，特别是和平台相关的地方，因为一旦平台有变动，就会引发bug，我们的应用中出现的取task id；

2）RecordWriter的输出，现有如下的方法输出，可能导致丢失所有结果文件，目前暂可以使用FsOutputStream来输出：

```
String path = outputPath.toString()+Path.SEPARATOR
+FileOutputCommitter.TEMP_DIR_NAME+Path.SEPARATOR+"_" +taskid+Path.SEPARATOR;
 RecordWriter<Text, Text> rw_file = new TextOutputFormat<Text,
Text>().getRecordWriter(null, job, path + pvpath, null);
```

作者简介：李邕。
联系方式：liyongmnls at gmail.com

3）当利用前一个M/R的结果作为cache file的内容时，最好用目录，而不是指定文件名，具体的文件过滤可在configure里判断，或使用file system来读；

4）只有map没有reduce的任务，数据量大的时候会出现很多的文件输出，可能会给后面的流程造成瓶颈（比如入库，另一个map的输入等），可以加一个reduce来类似的合并文件；

5）在configure里出错，要在M/R里来判断和抛异常；

6）在M/R里要有过滤和判断脏数据的逻辑，并记录在counter中；

7）在M/R里不能使用System.out.println的操作，会撑爆磁盘的，搞垮系统的，切记；

8）在M/R里要判断参数初始化是否正常，不正常时根据需要抛异常；

9）在combiner里要注意输入和输出的一致，若不一致，要增加判断的逻辑，因为一个combiner的结果要作为另一个combiner的输入；

10）参数要通过configure来传递，而不是通过main函数里变脸来传递；

11）有些reduce的运行时间太长，需要自己添加reporter.progress来表明reduce的状态，若无法报告，可以把心跳时间根据需要调大；

12）先定义输出类型在用set的方法会提高性能，比如：Text text = new Text（）；text.set("")；

13）自定义文件输出时，要使用参数mapred.output.format.class= org.apache.hadoop.mapred.lib.NullOutputFormat，来屏蔽系统空文件的输出；

14）使用main函数来调度最好能有redo机制、读写done文件的操作等（根据需要而定）；

15）Counter很多的话，可以通过counter group来管理；

16）在main函数里处理input路径时，可以使用mapred.input.dir来处理多个以逗号分隔的的路径，不能用FileInputFormat来直接处理多个逗号的分隔的输入；

17）可以使用MultipleInputs来处理多个map类（每个map类对应一个输入），结果给一个reduce来处理；类似的MultipleOutputs效率不佳；

18）MultiFileInputFormat与设置map数来控制小文件输入过多的问题，可以提高性能；

19）用main函数接入管理台的话，不要使用构造函数来初始化，可在run方法里初始化；

三、 Hadoop0.20 中 MapReduce 的新特性（引自互联网）

在Hadoop0.20更新中，最大的变化是加入了一个叫“ContextObjects”的新JavaAPI。把Mapper和Reduce通过引入ContextObject变成抽象类（不是接口），是为了让API将来更易于演进。

1）JobConf不再存在，Job配置信息由Configuration持有；

2）现在在map()或reduce()方法中，更容易得到job配置信息。仅需要调用 context.getConfiguration()。

3）新的API支持pull形式的迭代。在此之前，如果你想在mapper中遍历一下记录，你不得不把他们保存到Mapper类的实例变量中。在新的API中，只需要调用nextKeyValue()即可。

4）你还能覆盖run()方法，来控制mapper如何运行。

5）新API中不再有IdentityMapper和IdentityReducer两个类，因为默认Mapper和Reducer执行了标识函数。

新的API并非向后兼容，所以你必须重写你的应用。注意，新的API在

org.apache.hadoop.mapreduce包及其子包中，而旧的API在org.apache.hadoop.mapred。

Multipletaskassignment，这一优化让JobTracker在一次心跳周期内能分配给tasktracker多个任务，提高了利用率。同时引入mapred.reduce.slowstart.completed.maps新的配置参数（默认0.05）。

Inputformats 增加了一些有意思的改进。FileInputFormat 在选择那个主机有更多的需要分割的文件上做的更好。在另一方面 0.20 引入了 CombineFileInputFormat 类，他能把许多小文件变成一个 split。

Gridmix2 是第二代 MapReduce 工作量基准模型套件。

Hadoop 中的 tar 命令的实现

开心延年*

是否想过在 hadoop 中实现一个类似 linux 系统中的 tar 命令，可以将本地文件打包导入到 hdfs 中；同时也可以方便的将 hdfs 中的包文件解压到本地，还可以通过 czf，cjf 或者 cvf 选项选择不同的压缩算法。

大家是否因 hadoop namenode 内存瓶颈而不能导入大量的小文件所苦恼？以及在文件压缩以及 splitable 问题的权衡上所困惑？到底如何导入这些小文件？何时该启动压缩？应该选择那种压缩格式？本文将围绕这些问题，讨论 tar 命令的实现；并介绍如何在 mapreduce 和 hive 中使用此格式的数据进行数据分析；在最后，我们给出了实现此功能的完整源码和设计说明。

阅读本文需要有一定的 hadoop 知识，了解 hdfs，sequencefile，mapreduce 等。

一、 设计目标

- 解决因 namenode 内存瓶颈而无法存储大量小文件的问题，也就是说如何合并小文件；
- 文件压缩目标：splitable 可分割，边压缩边存储，不需要二次导入；
- 文件导入和导出，提供过滤筛选功能，以方便直接读取单个文件，而不是遍历整个大文件；
- 提供查看包内文件列表以及文件内容的命令，以方便调试；
- 便于 mapreduce 或者 hive 或其它程序读取以及分析。

二、 设计思路

2.1 小文件问题

如果能够将多个小文件合并成一个大文件，在需要的时候还可以直接将指定的小文件读取出来，那么小文件的问题就解决了。其实 tar 的打包过程就是合并的过程，解包的过程就是读取的过程，那么在这里小文件问题就自然解决了。

2.2 压缩格式与可分割性（splitable）、压缩率的权衡

目前 hadoop 支持几种压缩格式，但都是针对单个文件压缩，有很多目前还不具备可分割性。这些压缩格式还存在的一个问题是需要先将数据在本地压缩完毕后在压入到 hdfs 里，相当于 2 次 IO。

目前 hadoop 还有一种内置的 sequence file 格式，可以实用多种压缩格式，也具备可分割性，但是很多朋友都反映其压缩率较小，但是压缩算法都是一样的，压缩率比较小的问题是因为 key,value 的内容比较小，比如说仅仅是一行字符串或一个数字，而 sequencefile 的压缩是按照单个

作者简介：开心延年，从事数据分析相关工作，先后任职于新浪、腾讯、酷六。当前阶段在研究 mahout 源码，比较喜欢的开源项目有 lucene、hadoop、hive、mahout、hbase 等。
联系方式：myn at 163.com

key,value 来压缩的，并非整个文件压缩，所以可以在 value 里存储一个文件的多行数据(比如说 10M)，这样就可以提高压缩率。

在本文考虑到实现的复杂程度，决定采用 sequence file 格式存储数据，key,存储文件名，value 存储文件的内容，但这个内容“较大”，我们设置成 10M~20M，这样就解决了压缩率的问题。

2.3 文件打包过程

将文件按照固定的大小切割成 1 个或多个数据块，每块数据作为 sequence file 的 value,文件名做 sequence file 的 key,写入到 sequence file 中，这样完成了文件的打包。

这里的数据块通常是 10M~20M，如果文件不足 10M，那么该文件就只有一个数据块。

数据块与数据块之间确保按照\n 分割，这样就不会破坏在 mapreduce 分析时候的数据完整性。

2.4 文件解包过程

解包过程与打包过程正好相反，就是将相同文件的数据块按照原始顺序拼装起来，组合成完整的文件。

三、 概要设计

主要分为三个模块：

1）语法解析（parse）

负责对用户传入的指令参数进行语法解析，然后在调用相关方法响应用户的指令。

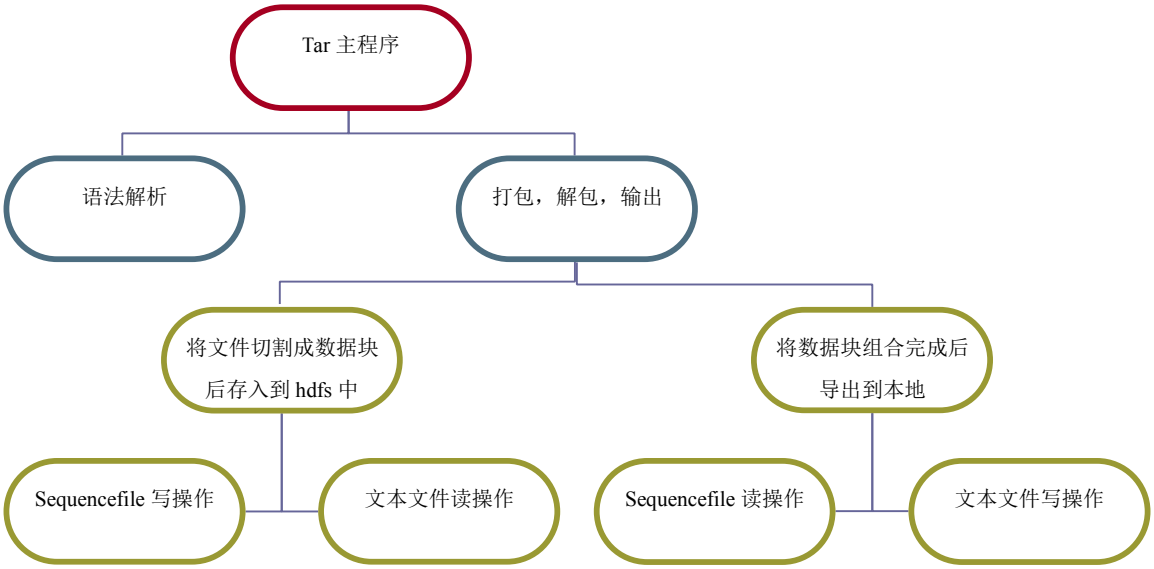
2）存储层 （store）

负责hdfs中的文件和本地文件的读写操作，包括文件文件的读写以及 sequence file文件的读写。

3）文件切割与连接

在打包过程中负责将文件切割成1个或多个数据块，解包过程中负责将相应的数据块组合成原先的文件。

4）程序逻辑结构图如下：



四、 详细设计

4.1 存储程序实现

4.1.1 文件操作公用函数

- 说明

  所有方法均为静态方法，封装了对文件和目录的的基本操作。包括文件或目录的查找，更名，删除、创建；也有包括获取 hadoop 封装的 hdfs 与本地文件系统对象。

- 使用示例

```
// 获取hdfs文件系统
FileSystem fs = TarFsPublic.getFileSystem();
// 获取本地文件系统
FileSystem localFs = TarFsPublic.getLocalFileSystem();

// hdfs中创建目录
TarFsPublic.mkdirs("/home/tartest/", fs);
// hdfs中文件更名
TarFsPublic.rename("/home/tartest/oldname.txt",
    "/home/tartest/newname.txt", fs);
// 删除本地/tmp/yannian/目录和其下所有文件
TarFsPublic.delete("/tmp/yannian", localFs);

// 查看本地/tmp目录下的所有文件和目录，非递归
FileStatus[] list = TarFsPublic.readDirFileStatus("/tmp/*", localFs);
for (FileStatus f : list) {
    System.out.println(f.getPath().toString());
}

// 递归查看/tmp目录下，所有路径包含yannian的文件
ArrayList<FileStatus> listAll = TarFsPublic.getFiles("/tmp/", localFs,
    ".*yannian.*");
for (FileStatus f : listAll) {
    System.out.println(f.getPath().toString());
}
```

实现源码请参见 **TarFsPublic.java**。

4.1.2 基本写操作

- 说明

  所有设计文件写操作的类，均继承于此类，其调用 TarFsPublic，实现文件和目录的的基本操作。

- 使用示例

```
FileSystem fs=TarFsPublic.getLocalFileSystem();
WriterBase writer=new WriterBase();
writer.open(fs, "/tmp/test.txt");
```

```
writer.mkdirs();
if(writer.exists())
{
    writer.delete();
}
writer.close();
```

实现源码请参见 **WriterBase.java**。

4.1.3 文本文件写操作

- 说明

  通过此类，可以向 hdfs 或本地中的文件写入 byte 数据或写入一个字符串。

- 使用示例

```
FileSystem fs = TarFsPublic.getFileSystem();
FileWriter writer = new FileWriter();
writer.open(fs, "/home/yannian/text.txt");
writer.writeString("hello word");
writer.write("hellow word".getBytes());
writer.close();
```

实现源码请参见 **FileWriter.java**。

4.1.4 sequencefile 基本写操作

- 说明

  Sequencefile 基本操作，可以通过 key，value 的方式，向文件中写入数据。

- 示例

```
FileSystem fs = TarFsPublic.getFileSystem();
SeqWriter writer = new SeqWriter();
writer.open(fs, "/home/yannian/text.seq");
writer.append("2010601", "the children has a good day");
writer.close();
```

实现源码请参见 **SeqWriter.java**。

4.1.5 基本读操作

- 说明

  所有的读操作都继承于此类，目前仅提供了判断文件是否存在的方法。

- 使用示例

```
FileSystem fs = TarFsPublic.getFileSystem();
ReaderBase reader = new ReaderBase();
reader.open(fs, "/home/hadoop/yannian.txt");
if (reader.exists()) {
    System.out.println("has found txt file");
}
reader.close();
```

实现源码请参见 **ReaderBase.java**。

4.1.6 文本文件读操作

- 说明
  普通文本文件的读操作，用于从文件中读取一定的字节的数据。
- 使用示例

```
FileSystem fs = TarFsPublic.getLocalFileSystem();
FileReader reader = new FileReader();
reader.open(fs, "/tmp/test/yannian/yannian.txt");
Text str = new Text();
if (reader.exists()) {
    byte[] bytes = new byte[10];
    while (true) {
        int len = reader.read(bytes);
        if (len > 0) {
            str.append(bytes, 0, len);
        }
        if (len < bytes.length) {
            break;
        }
    }
    System.out.println(str.toString());
}
reader.close();
```

实现源码请参见 **FileReader.java**。

4.1.7 sequencefile 文件的读操作

- 说明
  可以读取 sequence file 格式文件的内容。
- 使用示例

```
FileSystem fs = TarFsPublic.getFileSystem();
SeqReader reader = new SeqReader();
reader.open(fs, "/home/hadoop/yannian.seq");
if (reader.exists()) {
    while (reader.next()) {
        System.out.println("key=" + reader.getCurrentKey() + ";value="
                + reader.getCurrentValue());
    }
}
reader.close();
```

实现源码请参见 **SeqReader.java**。

4.2 数据块切割程序实现

- 说明
  用于将一个大文件切割成多个块，块与块之间按照\n 分割，默认的块大小是 10*1024*1024 字节。

每次读取的数据量大小=数据块的大小+数据块位置开始到达换行符这块数据的大小。

- 使用示例

```
FileSystem localfs = TarFsPublic.getLocalFileSystem();
FileReader reader = new FileReader();
reader.open(localfs, "/tmp/test/yannian/yannian.txt");
if (reader.exists()) {
    BlockReader br = new BlockReader(reader, 1024, 10240);
    Text str = new Text();
    int index = 0;
    while (br.readBlock(str) > 0) {
        System.out.println("###########[" + index + "]######");
        System.out.println(str.toString());
    }
}
reader.close();
```

实现源码请参见 **BlockReader.java**。

4.3 打包程序的实现

- 说明

  用于将文件切割，并写入到包文件中，创建 tar 包会直接调用此类，writeFile 就是将文件切割并写入 tar 包的过程。

- 使用示例

```
FileSystem localfs = TarFsPublic.getLocalFileSystem();
FileSystem fs = TarFsPublic.getFileSystem();

SeqWriter wr = new SeqWriter();
wr.open(fs, "/tmp/test/tar.seq");
WriteTar writeTar = new WriteTar(wr, 1024, 10240);

FileReader freader = new FileReader();
freader.open(localfs, "/tmp/test/yannian/yannian.txt");
writeTar.writeFile(freader);
freader.close();

FileReader freader2 = new FileReader();
freader2.open(localfs, "/tmp/test/yannian/yannian2.txt");
writeTar.writeFile(freader2);
freader2.close();

wr.close();
```

实现源码请参见 **WriteTar.java**。

4.4 解包程序的实现

- 说明
  用于将 tar 包的内容读取出来，包括读取文件名列表和文件内容。
- 使用示例

```
FileSystem fs = TarFsPublic.getFileSystem();
FileSystem localFs = TarFsPublic.getLocalFileSystem();
SeqReader readSeq = new SeqReader();
readSeq.open(fs, "/tmp/test/tar.seq");
ReadTar readTar = new ReadTar(readSeq);
while (readTar.nextFile()) {
    String localPath = "/tmp/test/extract/"
            + readTar.getCurrentFileName().toString();
    System.out.println(localPath);
    FileWriter writer = new FileWriter();
    writer.open(localFs, localPath);
    readTar.readFileContents(writer);
    writer.close();
}
readSeq.close();
```

实现源码请参见 **ReadTar.java**。

4.5 Tar 按行读取 readline 的实现

- 说明
  可以将 tar 文件的内容一行一行的读取出来，目前在打印 tar 内容上所用，同时在 mapreduce 和 hive 分析数据中使用。
- 使用示例

```
FileSystem fs = TarFsPublic.getFileSystem();
SeqReader readSeq = new SeqReader();
readSeq.open(fs, "/tmp/test/tar.seq");
ReadTar readTar = new ReadTar(readSeq);
TarLineReader lineReader = new TarLineReader(readTar);
lineReader.setJoinString("\t");

Text line = new Text();
while (lineReader.readLine(line) > 0) {
    System.out.println(line.toString());
}

readSeq.close();
```

实现源码请参见 **TarLineReader.java**。

4.6 语法解析程序

- 说明
  负责对 tar 命令传入的参数的解析。
- 使用示例

```
ParamsParse param = new ParamsParse();
param.Parse(args);
param.printUsage();
System.out.println("isPrintHelp():"+param.isPrintHelp());
System.out.println("isExport():"+param.isExport());
System.out.println("isPrint():"+param.isPrint());
System.out.println("isList():"+param.isList());
System.out.println("isInput():"+param.isInput());
System.out.println("getTop():"+param.getTop());
System.out.println("isOverWrite():"+param.isOverWrite());
System.out.println("isViewDetail():"+param.isViewDetail());
System.out.println("isIgPath():"+param.isIgPath());
System.out.println("getStrMatch():"+param.getStrMatch());
System.out.println("getCompressType():"+param.getCompressType());
System.out.println("getJoinChar():"+param.getJoinChar());
```

实现源码请参见 **ParamsParse.java**。

4.7 主程序实现

- 说明
  Tar 的入口程序，一切由此开始执行。
- 源码参见 **Tar.java**。

4.8 tar.sh 的内容

Tar.sh 是命令行入口的主程序，内部实现是直接调用 hadoop 命令。

使用前需要修改安装 hadoop 的路径和生成的 jar 包的绝对地址。

```
#!/bin/sh
. /etc/profile

/usr/local/hadoop/bin/hadoop jar xxx.jar tar.Tar $@
```

4.9 Mapreduce 中使用 tar 格式的数据

4.9.1 实现思路

Tar 包内的数据实际上存储在 seqencefile 中，仅仅是 value 中存储了多行的数据，我们将他们在重新拆开即可，具体实现见下面的 TarInputFormat，这里使用的是 hadoop0.20.2 的 mapreduce 新版定义。

4.9.2 TarInputFormat 的实现

```
package tar;

import java.io.IOException;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
/**
 * Mapreduce中使用tar格式的数据
 *
 * Tar包内的数据实际上存储在seqencefile中，仅仅是value中存储了多行的数据，我们将他们在重新拆开即可.
 * 这里使用的是hadoop0.20.2的mapreduce新版定义
 * @author muyannian
 *
 */
public class TarInputFormat extends SequenceFileInputFormat<Text, Text> {
    /**
     * 获取reader对象
     */
    public RecordReader<Text, Text> createRecordReader(InputSplit split,
            TaskAttemptContext context) throws IOException {
        return new TarRecordReader();
    }

    /**
     * tar reader的实现
     * @author muyannian
     *
     */
    static public class TarRecordReader extends RecordReader<Text, Text> {
        /**
         * 按行读取
         */
        TarLineReader reader = null;
        /**
         * 当前处理的文件名
         */
        Text value = new Text();
        /**
         * 当前行文件内容
         */
        Text key = new Text();

        /**
         * hadoop sequencefile格式的reader
         */
        SequenceFileRecordReader<Text, Text> seqReader = new SequenceFileRecordReader<Text,
```

```
Text>();

        @Override
        public void initialize(InputSplit split, TaskAttemptContext context)
                throws IOException, InterruptedException {
            seqReader.initialize(split, context);

        }

        @Override
        public void close() throws IOException {
            seqReader.close();

        }

        @Override
        public Text getCurrentKey() throws IOException, InterruptedException {
            return key;
        }

        @Override
        public Text getCurrentValue() throws IOException, InterruptedException {

            return value;
        }

        @Override
        public float getProgress() throws IOException, InterruptedException {
            return seqReader.getProgress();
        }

        @Override
        public boolean nextKeyValue() throws IOException, InterruptedException {
            while (reader == null || reader.readKeyValue(key, value) <= 0) {
                if (!seqReader.nextKeyValue()) {
                    return false;
                }
                reader = new TarLineReader(seqReader.getCurrentKey(),
                        seqReader.getCurrentValue());
            }
            return true;
        }
    }
}
```

4.10 Hive里使用tar格式的数据

目前hive是支持自定义格式的，通过设置input format即可，注意hive里的input format使用的旧版的接口。

- HiveTarInputFormat的实现

```
package tar;

import java.io.IOException;
import java.util.Arrays;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * hive中使用tar格式的数据
 *
 * Tar包内的数据实际上存储在seqencefile中，仅仅是value中存储了多行的数据，我们将他们在重新拆开即可.
 * 目前hive是支持自定义格式的，通过设置input format即可，注意hive里的input format使用的是的旧版的接口
 * @author muyannian
 *
 */
public class HiveTarInputFormat extends
        SequenceFileInputFormat<LongWritable, BytesWritable> {
    /**
     * 获取reader对象
     */
    public RecordReader<LongWritable, BytesWritable> getRecordReader(
            InputSplit split, JobConf job, Reporter reporter)
            throws IOException {

        reporter.setStatus(split.toString());
        return new HiveTarRecordReader(job, (FileSplit) split);
    }

    /**
     * tar reader的实现
     * @author muyannian
     *
     */
    static public class HiveTarRecordReader implements
            RecordReader<LongWritable, BytesWritable> {
        /**
         * 按行读取
```

```
 */
TarLineReader linereader = null;
/**
 * 当前处理的文件名
 */
Text key = new Text();
/**
 * 当前行文件内容
 */
Text value = new Text();

/**
 * 数据块的key
 */
Text blockkey = new Text();

/**
 * 数据块的value
 */
Text blockVvalue = new Text();

/**
 * hadoop sequencefile格式的reader
 */

SequenceFileRecordReader seqReader = null;

public HiveTarRecordReader(Configuration conf, FileSplit _split)
        throws IOException {

    seqReader = new SequenceFileRecordReader<Text, Text>(conf, _split);

}

public synchronized boolean next(LongWritable pos, BytesWritable k)
        throws IOException {

    boolean hasNext = this.nextKeyValue();

    if (hasNext) {
        Text line = this.getCurrentValue();
        pos = new LongWritable(seqReader.getPos());
        byte[] textBytes = line.getBytes();
        int length = line.getLength();
```

```
            if (length != textBytes.length) {
                textBytes = Arrays.copyOf(textBytes, length);
            }

            k.set(textBytes, 0, textBytes.length);
        }
        return hasNext;
    }

    private Text getCurrentValue() throws IOException {

        return value;
    }

    private boolean nextKeyValue() throws IOException {

        while (linereader == null
                || linereader.readKeyValue(key, value) <= 0) {
            if (!seqReader.next(blockkey, blockVvalue)) {
                return false;
            }
            linereader = new TarLineReader(blockkey, blockVvalue);
        }
        return true;
    }

    @Override
    public void close() throws IOException {
        seqReader.close();

    }

    @Override
    public LongWritable createKey() {
        return new LongWritable();
    }

    @Override
    public BytesWritable createValue() {
        return new BytesWritable();
    }

    @Override
```

```
        public long getPos() throws IOException {
            return seqReader.getPos();
        }

        @Override
        public float getProgress() throws IOException {
            return seqReader.getProgress();
        }

    }
}
```

五、 命令使用说明

5.1 文件打包命令

tar.sh –czvf [-w] [-m <matcher>] <hdfsFile> <localdir.1>...<localdir.N>

将本地系统中的的一个或多个目录里的所有文件打包到指定的hdfs中的文件里。

1）–cvf –czvf –cjvf：分别对应 不压缩，zip方式压缩和bz2方式压缩；

2）-w 选项：表示如果hdfs已经存在此包，则先删除hdfs中的此包；

3）–m选项：可以指定正则，只有满足条件的文件才导入到包里。

5.2 文件解包命令

tar.sh –xvf [-m <matcher>] [-ig] [-w] <hdfsFile> <localdir>

将hdfs中的指定包中所有文件解压到本地，并创建完整的目录结构。

1）–xvf：将包中的文件解压出来，会自动识别解压编码；

2）-m选项：可以指定正则，只有满足条件的文件才解出来；

3）–ig选项：表示解压后不创建目录结构，所有文件均在同一目录下。

5.3 查看、调试与维护相关命令

1）tar.sh -print [-top N] [-join joinchar] [-m <matcher>] <hdfsPath>

查看tar包内的文件内容，top 选项表示最多打印几行，join表示文件名与文件内容之间分隔符，默认是\t。

2）tar.sh -ls <hdfsPath>

打印tar包中的文件列表。

3）tar.sh –export [-top N] [-w] [-m <matcher>] [-join joinchar] <hdfsPath> <localfile>

将tar包的内容，导出到文件文件中。

六、 基本使用示例

6.1 文件打包

将/tmp/test 内的所有文件 打包到 tmp.sar 里。

./tar.sh -czvf /tmp/test/tmp.sar /tmp/test/

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -czvf /tmp/test/tmp.sar /tmp/test/
file:/tmp/test/access_log.201012040931
11/02/25 10:05:30 INFO util.NativeCodeLoader: Loaded the native-hadoop library
11/02/25 10:05:30 INFO zlib.ZlibFactory: Successfully loaded & initialized nat
11/02/25 10:05:30 INFO compress.CodecPool: Got brand-new compressor
file:/tmp/test/access_log.201012040932
file:/tmp/test/access_log.201012040933
file:/tmp/test/access_log.201012040934
file:/tmp/test/access_log.201012040935
file:/tmp/test/access_log.201012040936
file:/tmp/test/access_log.201012040937
file:/tmp/test/access_log.201012040938
file:/tmp/test/access_log.201012040939
file:/tmp/test/access_log.201012040940
```

6.2 文件解包

将 tmp.sar 包的内容解压到 /tmp/test 目录。

./tar.sh -xzvf /tmp/test/tmp.sar /tmp/test/

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -xzvf /tmp/test/tmp.sar /tmp/test/
11/02/25 10:10:46 INFO util.NativeCodeLoader: Loaded the native-hadoop l
11/02/25 10:10:46 INFO zlib.ZlibFactory: Successfully loaded & initializ
11/02/25 10:10:46 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:10:46 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:10:46 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:10:46 INFO compress.CodecPool: Got brand-new decompressor
/tmp/test//file:/tmp/test/access_log.201012040931
/tmp/test//file:/tmp/test/access_log.201012040932
/tmp/test//file:/tmp/test/access_log.201012040933
/tmp/test//file:/tmp/test/access_log.201012040934
/tmp/test//file:/tmp/test/access_log.201012040935
/tmp/test//file:/tmp/test/access_log.201012040936
/tmp/test//file:/tmp/test/access_log.201012040937
/tmp/test//file:/tmp/test/access_log.201012040938
/tmp/test//file:/tmp/test/access_log.201012040939
/tmp/test//file:/tmp/test/access_log.201012040940
```

6.3 查看包的内容

- 查看 tmp.sar 包的内容

./tar.sh -print /tmp/test/tmp.sar |head -n 3

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -print /tmp/test/tmp.sar |head -n 3
11/02/25 10:13:29 INFO util.NativeCodeLoader: Loaded the native-hadoop library
11/02/25 10:13:29 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib l
11/02/25 10:13:29 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:13:29 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:13:29 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:13:29 INFO compress.CodecPool: Got brand-new decompressor
block:0@file:/tmp/test/access_log.201012040931  0  1291426234046588 GET /yayun2010.gif?t
.ku6.com/lm26/index.shtml?s?=?k?=?u?=0?if?=0?v?=| HTTP/1.1  http://yayun2010.ku6.com/lm2
0
block:0@file:/tmp/test/access_log.201012040931  0  1291423511218607 GET /yayun2010.gif?t
.com/yiyou/index.shtml?p?=http%3A//yayun2010.ku6.com/live/?s?=?k?=?u?=0?if?=1?v?=| HTTP/
.79.66 20101204093100
block:0@file:/tmp/test/access_log.201012040931  0  1291423577578556 GET /yayun2010.gif?t
.com/yiyou/index.shtml?p?=http%3A//yayun2010.ku6.com/live/?s?=?k?=?u?=0?if?=1?v?=| HTTP/
.61.194 20101204093100
```

- 查看包内匹配文件的前 3 条记录

./tar.sh -print -top 3 -m ".*access_log.201012040931.*" /tmp/test/tmp.sar

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -print -top 3 -m ".*access_log.201012040931.*" /tmp/test/tmp.sar
11/02/25 10:15:38 INFO util.NativeCodeLoader: Loaded the native-hadoop library
11/02/25 10:15:38 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
11/02/25 10:15:38 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:15:38 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:15:38 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:15:38 INFO compress.CodecPool: Got brand-new decompressor
block:0@file:/tmp/test/access_log.201012040931  0  1291426234046588 GET /yayun2010.gif?t?=12914262601712
.ku6.com/lm26/index.shtml?s?=?k?=?u?=0?if?=0?v?=| HTTP/1.1  http://yayun2010.ku6.com/lm26/index.shtml  1
0
block:0@file:/tmp/test/access_log.201012040931  0  1291423511218607 GET /yayun2010.gif?t?=12914258419533
.com/yiyou/index.shtml?p?=http%3A//yayun2010.ku6.com/live/?s?=?k?=?u?=0?if?=1?v?=| HTTP/1.1  http://yayu
.79.66 20101204093100
block:0@file:/tmp/test/access_log.201012040931  0  1291423577578556 GET /yayun2010.gif?t?=12914257334683
.com/yiyou/index.shtml?p?=http%3A//yayun2010.ku6.com/live/?s?=?k?=?u?=0?if?=1?v?=| HTTP/1.1  http://yayu
.61.194 20101204093100
block:0@file:/tmp/test/access_log.201012040931  0  1291426244000844 GET /yayun2010.gif?t?=12914262571565
.ku6.com/lmgy/index.shtml?s?=?k?=?u?=0?if?=0?v?=| HTTP/1.1  http://yayun2010.ku6.com/lmgy/index.shtml  1
[hadoop@hadoopSlave36 run]$
```

6.4 导出到文件

./tar.sh -export /tmp/test/tmp.sar /tmp/test/data.txt

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -export /tmp/test/tmp.sar /tmp/test/data.txt
11/02/25 10:17:16 INFO util.NativeCodeLoader: Loaded the native-hadoop library
11/02/25 10:17:16 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
11/02/25 10:17:16 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:17:16 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:17:16 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:17:16 INFO compress.CodecPool: Got brand-new decompressor
[hadoop@hadoopSlave36 run]$ ll /tmp/test/data.txt
-rwxrwxrwx 1 hadoop hadoop 3844230 02-25 10:17 /tmp/test/data.txt
[hadoop@hadoopSlave36 run]$ head /tmp/test/data.txt -n 5
block:0@file:/tmp/test/access_log.201012040931  0  1291426234046588 GET /yayun2010.gif?t?=1291426260171261?r?=?p?=httpX3A//
.ku6.com/lm26/index.shtml?s?=?k?=?u?=0?if?=0?v?=| HTTP/1.1  http://yayun2010.ku6.com/lm26/index.shtml  123.146.134.103 2010
0
block:0@file:/tmp/test/access_log.201012040931  0  1291423511218607 GET /yayun2010.gif?t?=1291425841953367?r?=http%3A//yayu
.com/yiyou/index.shtml?p?=http%3A//yayun2010.ku6.com/live/?s?=?k?=?u?=0?if?=1?v?=| HTTP/1.1  http://yayun2010.ku6.com/live/
.79.66 20101204093100
block:0@file:/tmp/test/access_log.201012040931  0  1291423577578556 GET /yayun2010.gif?t?=1291425733468310?r?=http%3A//yayu
.com/yiyou/index.shtml?p?=http%3A//yayun2010.ku6.com/live/?s?=?k?=?u?=0?if?=1?v?=| HTTP/1.1  http://yayun2010.ku6.com/live/
.61.194 20101204093100
block:0@file:/tmp/test/access_log.201012040931  0  1291426244000844 GET /yayun2010.gif?t?=1291426257156947?r?=?p?=httpX3A//
.ku6.com/lmgy/index.shtml?s?=?k?=?u?=0?if?=0?v?=| HTTP/1.1  http://yayun2010.ku6.com/lmgy/index.shtml  117.136.9.98 2010120
block:0@file:/tmp/test/access_log.201012040931  0  1291426388984160 GET /yayun2010.gif?t?=1291426388984160?r?=?p?=httpX3A//
.ku6.com/lm26/index.shtml?s?=?k?=?u?=0?if?=0?v?=| HTTP/1.1  http://yayun2010.ku6.com/lm26/index.shtml  112.254.107.80 20101
```

6.5 查看包内文件列表

./tar.sh -ls /tmp/test/tmp.sar

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -ls /tmp/test/tmp.sar
11/02/25 10:19:43 INFO util.NativeCodeLoader: Loaded the native-hadoop librar
11/02/25 10:19:43 INFO zlib.ZlibFactory: Successfully loaded & initialized na
11/02/25 10:19:43 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:19:43 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:19:43 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 10:19:43 INFO compress.CodecPool: Got brand-new decompressor
file:/tmp/test/access_log.201012040931
file:/tmp/test/access_log.201012040932
file:/tmp/test/access_log.201012040933
file:/tmp/test/access_log.201012040934
file:/tmp/test/access_log.201012040935
file:/tmp/test/access_log.201012040936
file:/tmp/test/access_log.201012040937
file:/tmp/test/access_log.201012040938
file:/tmp/test/access_log.201012040939
file:/tmp/test/access_log.201012040940
```

七、 Mapreduce 和 hive 示例

7.1 Mapreduce 分析 tar 包数据

- 使用说明

  job 必须设置 TarInputFormat 为格式化函数。

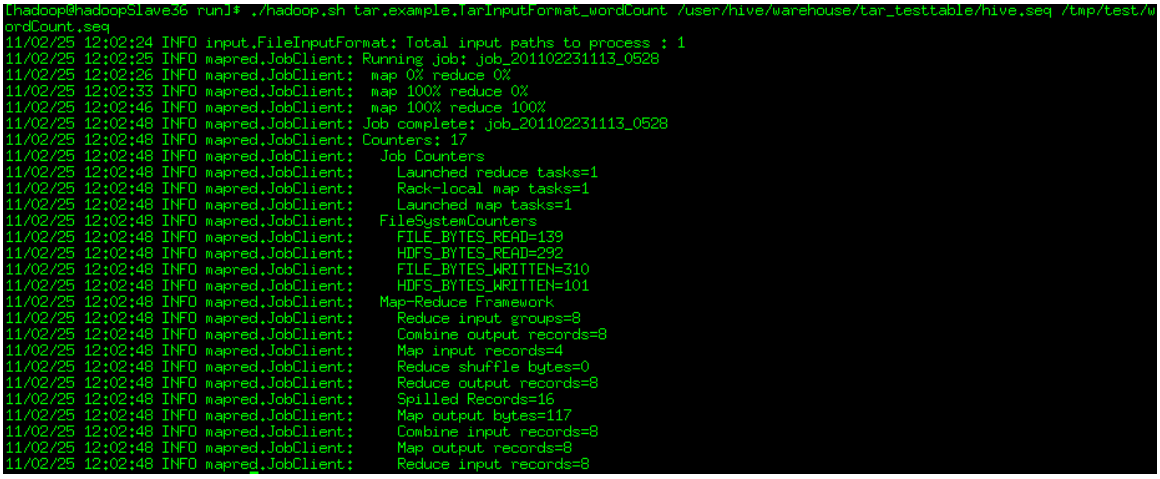
job.setInputFormatClass(TarInputFormat.class);
map 里的 key 是 Text 类型，传入的是文件名称。
public void map(Text key, Text value, Context context)…

- 示例程序
  使用 Hadoop 标准 Word count 程序分析 tar 格式数据。

```
package tar.example;
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

/**
    使用mapreduce分析tar格式的数据示例，需要注意两点


    1.job必须设置TarInputFormat为格式化函数
        job.setInputFormatClass(TarInputFormat.class);
    2.map里的key是Text类型，传入的是文件名称
        public void map(Text key, Text value, Context context)…

 * @author muyannian
 *
 */
public class TarInputFormat_wordCount {

      public static class TokenizerMapper
          extends Mapper<Text, Text, Text, IntWritable>{

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
         //注意这里，map数据的key发生了变化
        public void map(Text 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);
          }
```

```
    }
  }

  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);
    }
  }

  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");

    //注意这里，格式发生了变化
    job.setInputFormatClass(tar.TarInputFormat.class);
    job.setJarByClass(TarInputFormat_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]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

- 执行示例



7.2 使用 hive 来分析 tar 包数据

- 无论是查询，还是创建表必须将编译好的 jar 包导入到 hive 中。

add jar xxx/sar.jar

- 创建表,注意存储类型使用了 HiveTarInputFormat 来格式化。

stored as INPUTFORMAT 'tar.HiveTarInputFormat' OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat'

- 使用示例

```
[hadoop@hadoopSlave36 run]$ ./tar.sh -czvf /tmp/test/hive.seq text.txt
file:/home/rsync/shell/run/text.txt
11/02/25 11:53:21 INFO util.NativeCodeLoader: Loaded the native-hadoop library
11/02/25 11:53:21 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
11/02/25 11:53:21 INFO compress.CodecPool: Got brand-new compressor

[hadoop@hadoopSlave36 run]$ ./tar.sh -print /tmp/test/hive.seq
11/02/25 11:53:26 INFO util.NativeCodeLoader: Loaded the native-hadoop library
11/02/25 11:53:26 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
11/02/25 11:53:26 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 11:53:26 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 11:53:26 INFO compress.CodecPool: Got brand-new decompressor
11/02/25 11:53:26 INFO compress.CodecPool: Got brand-new decompressor
file:/home/rsync/shell/run/text.txt    zhangbingbing   1qaz2wsx
file:/home/rsync/shell/run/text.txt    fengbingjian    123456789
file:/home/rsync/shell/run/text.txt    muyannian       1wdvbhuik
file:/home/rsync/shell/run/text.txt    sixuefeng       plk91lkd

[hadoop@hadoopSlave36 run]$  /usr/local/hive/bin/hive
Hive history file=/tmp/hadoop/hive_job_log_hadoop_201102251153_300103717.txt
hive> add jar /home/rsync/shell/sar.jar;
Added /home/rsync/shell/sar.jar to class path
hive> CREATE TABLE tar_testTable(username STRING,passwd STRING)
```

```
    > ROW FORMAT
    > DELIMITED  FIELDS TERMINATED BY '\t'
    > stored as INPUTFORMAT 'tar.HiveTarInputFormat'
    > OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat'
    > ;
OK
Time taken: 2.628 seconds

hive> LOAD DATA INPATH '/tmp/test/hive.seq' OVERWRITE INTO TABLE tar_testTable;
Loading data to table tar_testtable
OK
Time taken: 0.127 seconds

hive> select * from tar_testTable;
OK
zhangbingbing   1qaz2wsx
fengbingjian    123456789
muyannian       1wdvbhuik
sixuefeng       p1k91lkd
Time taken: 0.34 seconds

hive> select count(*) as cnt from tar_testTable;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_201102231113_0527, Tracking URL =
http://hadoopJobTacker:50030/jobdetails.jsp?jobid=job_201102231113_0527
Kill Command = /usr/local/hadoop/bin/hadoop job
-Dmapred.job.tracker=hadoopJobTacker.ku6.com:9001 -kill job_201102231113_0527
2011-02-25 11:54:53,132 Stage-1 map = 0%,  reduce = 0%
2011-02-25 11:54:56,164 Stage-1 map = 100%,  reduce = 0%
2011-02-25 11:55:14,322 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201102231113_0527
OK
4
Time taken: 35.8 seconds
```

八、 二次开发

命令行模式，并不是在任何时刻都适合，而且有时还需要进一步拓展tar的功能，所以提供相关的接口示例，供二次开发所用。

1）读取包内的文件列表

```
SeqReader readSeq = new SeqReader();
readSeq.open(fs, otherargs.get(0));
ReadTar readTar = new ReadTar(readSeq);
while (readTar.nextFile()) {
    System.out.println(readTar.getCurrentFileName());
}
readSeq.close();
```

2）向tar包中导入文件

```
FileSystem localfs = TarFsPublic.getLocalFileSystem();
FileSystem fs = TarFsPublic.getFileSystem();

SeqWriter wr = new SeqWriter();
wr.open(fs, "/tmp/test/tar.seq");
if(wr.exists())
{
    wr.delete();
}
WriteTar writeTar = new WriteTar(wr, 1024, 10240);

//文件一
FileReader freader = new FileReader();
freader.open(localfs, "/tmp/test/yannian/yannian.txt");
writeTar.writeFile(freader);
freader.close();

//文件二
FileReader freader2 = new FileReader();
freader2.open(localfs, "/tmp/test/yannian/yannian2.txt");
writeTar.writeFile(freader2);
freader2.close();

wr.close();
```

3）将tar包的文件导出到本地

```
FileSystem fs = TarFsPublic.getFileSystem();
FileSystem localFs = TarFsPublic.getLocalFileSystem();
SeqReader readSeq = new SeqReader();
readSeq.open(fs, "/tmp/test/tar.seq");
ReadTar readTar = new ReadTar(readSeq);
```

```
    while (readTar.nextFile()) {
        String localPath = "/tmp/test/extract/"
            + readTar.getCurrentFileName().toString();
        System.out.println(localPath);
        FileWriter writer = new FileWriter();
        writer.open(localFs, localPath);
        readTar.readFileContents(writer);
        writer.close();
    }
    readSeq.close();
```

4）按行读取tar包的内容

```
    FileSystem fs = TarFsPublic.getFileSystem();
    SeqReader readSeq = new SeqReader();
    readSeq.open(fs, "/tmp/test/tar.seq");
    ReadTar readTar = new ReadTar(readSeq);
    TarLineReader lineReader = new TarLineReader(readTar);
    lineReader.setJoinString("\t");

    Text line = new Text();
    while (lineReader.readLine(line) > 0) {
        System.out.println(line.toString());
    }

    readSeq.close();
```

九、 后期拓展

1）hadoop在0.21.0以后将支持append，那么通过重写SeqWriter的getWriter方法，就可以实现append功能。

2）这里的tar都是针对将local文件tar到hdfs中，大家可以通过简单的设置FileSystem来切换到任意hdoop所支持的文件系统中去。

3）由于二进制文件切割的复杂性，目前该命令仅仅支持文本文件，有兴趣的读者可以考虑改造数据切割程序BlockReader，来实现对二进制文件的切割算法或者满足当前业务的二进制需要。

十、 注意事项

1）这里的tar只是类tar命令，文件的格式并不与linux 下的tar格式兼容。

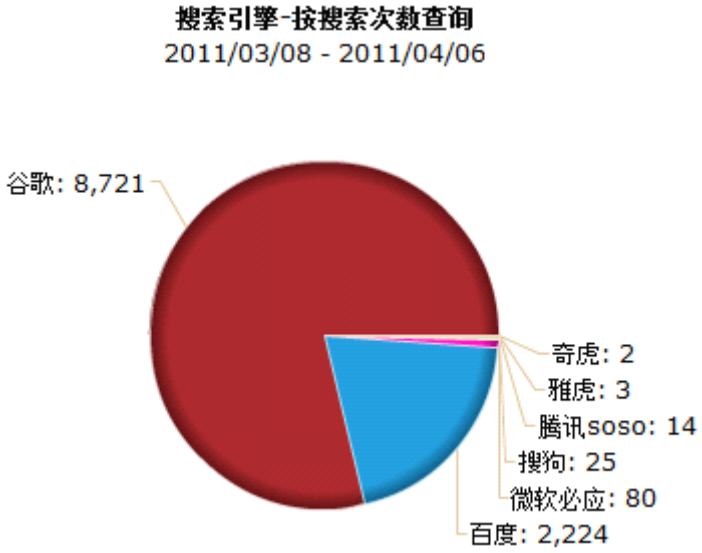
2）本程序是从酷六数据仓库项目组原有程序的基础上提炼而来，时间关系并未对提炼出来的代码进行完整的测试，仅仅作为一种实现思路供大家参考。

3）本文所使用的hadoop版本为hadoop-0.20.2，hive版本为hive-0.6.0。

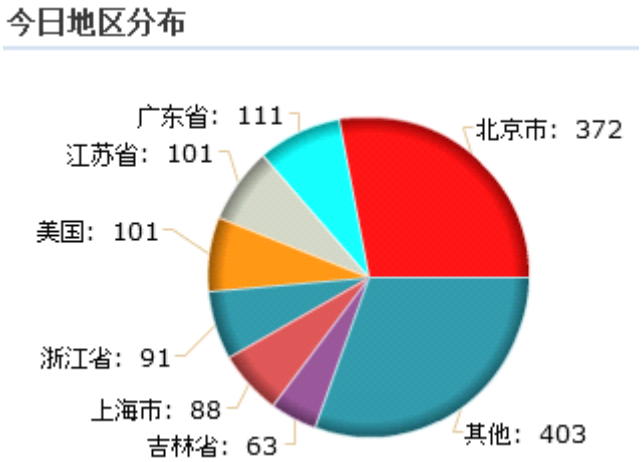
源码下载：http://bbs.hadoopor.com/src/hadoop-tar-src.zip

Hadoop 技术论坛运营数据分享

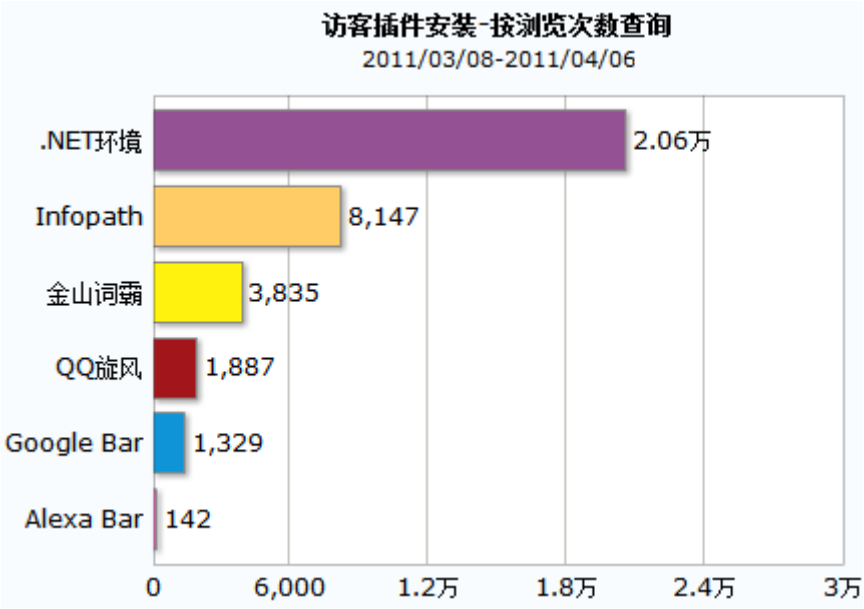
一、 搜索引擎



二、 地区分布



三、 访客插件安装



四、 访客操作系统

操作系统列表（从 2011-03-08 到 2011-04-06）

操作系统	浏览次数	百分比↓	独立访客	IP	人均浏览次数	平均停留时间(秒)
总计	**51755**	100%	**13842**	**10663**	**3.74**	**88.66**
WinXP	26117	50.46%	6880	5193	3.80	86.70
Windows 7	16884	32.62%	4355	3435	3.88	85.92
Linux	5946	11.49%	1820	1390	3.27	104.27
Win2003	1264	2.44%	242	202	5.22	75.73
WinVista	874	1.69%	247	215	3.54	94.56
ApplePC	598	1.16%	260	205	2.30	111.60
Nokia	21	0.04%	3	6	7.00	71.38
Win2000	21	0.04%	19	3	1.11	171.81
Android	15	0.03%	5	5	3.00	82.47
iPhone	13	0.03%	9	7	1.44	144.23
Windows	2	0.00%	2	2	1.00	180.00