**基于Hadoop+Spark集群的大数据系统的设计与实现**

——脉脉网的大数据分析实证研究

**摘要：**在互联网信息爆炸性增长的背景下，互联网数据的重要性也不言而喻，但在海量的互联网数据面前，无论是对数据的存储、清洗还是对数据的处理，传统的数据处理模式显得捉襟见肘。分布式大数据处理平台的诞生为海量数据的分析提供了解决方案。以Hadoop为核心大数据生态也逐渐成为互联网公司追捧的对象。

#### 本次设计通过Docker虚拟机技术，依托centOS操作系统，虚拟含有四台Linux操作系统的集群，在此基础上搭建集Hadoop、Zookeeper、Hbase、Spark为一体的完全分布式的大数据分析平台，然后通过Python爬虫代码爬取新兴的职场社交网站—脉脉网大约100万的用户数据，然后对基础数据进行清洗通过Thrift插件将清洗后的数据写入到Hbase中存储，再通过编写Scala代码利用Spark读取Hbase中的数据，分析脉脉网用户的城市分布、省份分布、职位分布、话题时间分布等数据特征，最后将Spark的数据处理结果整理成JS文件，通过编写HTML和CSS代码，使用echarts图表插件，实现数据处理的可视化。从而实现利用大数据技术发现和挖掘脉脉网用户的潜在特征和价值。

**关键字**：大数据，Hadoop，Spark，Hbase，Zookeeper

Design and implementation of big data system based on Hadoop cluster

——Big data analysis: An Empirical Study on maimai

A**bstract：**In the background of Internet information explosion, the importance of the Internet data is self-evident, but in the face of Internet data, whether the data storage, cleaning or cleaning of the data, the traditional data processing mode is difficult. The birth of distributed large data processing platform provides a solution for the analysis of mass data. To Hadoop as the core, big data ecology has gradually become the object of Internet Co sought after.

This design through the Docker virtual machine technology, relying on the centOS operating system, virtual operating system containing four Linux clusters, large data fully distributed on the basis of Hadoop and Zookeeper, to build a set of Hbase and Spark as one of the analysis platform, then the user data from the maimai about 1 million of the climb through the tenderness Python crawler code, and then on the basis of the data cleaning by Thrift plug after cleaning the data is written to the Hbase store, then by writing Scala code using Spark to read the data in the Hbase, features of city distribution, geographical distribution, position distribution, time distribution of data analysis and topic of network users, and finally the data processing Spark the results into a JS file, written by HTML and CSS code, use the echarts chart plug-ins, realize the visualization of data processing. In order to achieve the use of big data technology and found potential value and characteristics of mining pulse network users.

**Key Words：**BigData,Hadoop,Hbase,Zookeeper,Spark

**目 录**

[1大数据开发技术环境 2](#_Toc482795415)

[1.1 大数据处理的起源 2](#_Toc482795416)

[1.2 Hadoop 2](#_Toc482795417)

[1.3 Hbase 4](#_Toc482795421)

[1.4 Spark 7](#_Toc482795425)

[2系统设计与实现 10](#_Toc482795428)

[2.1 系统基础架构 10](#_Toc482795429)

[2.2 搭建集群 11](#_Toc482795430)

[2.3 大数据组件的安装配置 17](#_Toc482795434)

[3 脉脉网大数据分析实证 28](#_Toc482795440)

[3.1脉脉网及其数据接口 28](#_Toc482795441)

[3.1 数据爬取 30](#_Toc482795442)

[3.2 数据处理 31](#_Toc482795443)

[3.4结果展示 34](#_Toc482795444)

[4 总结 40](#_Toc482795451)

[参考文献 41](#_Toc482795452)

[附录 42](#_Toc482795453)

[HDFS配置文件代码 42](#_Toc482795454)

[core-site.xml代码 44](#_Toc482795455)

[hbase-site.xml代码 45](#_Toc482795456)

[Zookeeper配置文件 50](#_Toc482795457)

[Spark的maven工程pom文件 51](#_Toc482795458)

[CountJob.scala代码 58](#_Toc482795459)

[HTML前端代码 60](#_Toc482795460)

[谢辞 64](#_Toc482795461)

最近20年，互联网在全世界呈现爆炸式的增长，互联网本身也一直在不断地成长与进化，互联网技术的更新速度远远超过其他的行业和工具。

相比较于其他的数据，互联网数据具有数量大，易获取，质量参差不齐等特点，而在这些相对廉价的参差不齐的海量数据中往往隐藏着巨大的价值，经典的啤酒尿布的案例也只是其中的冰山一角的体现，数据的挖掘与分析也已经成为推动社会经济发展的巨大助力。但是在面对动辄TB甚至PB级别的海量数据时候，传统的数据处理的模式就显得非常的捉襟见肘。

首先是数据量巨大的问题，传统的数据处理模式往往是在一个单机系统中运用数据处理工具对数据进行清洗、分析和运算等一些列流程。但互联网海量数据的数据量已经远远超出了现有的单机磁盘容量，数据装载就几乎不可能完成，更不可能说再去进行一系列的数据处理与运算。

其次是数据质量参差不齐的问题，由于互联网数据的来源千奇百怪，加之数据时间的不确定性，其数据质量很难得到保证。但一个高质量的数据集是数据分析成功的关键因素，在面对低质量的数据集的时候，再好的数据分析处理模型也很难达到预期的效果。

因此大数据的分析处理逐渐成为刚需，以国内为例，近三年以Hadoop为代表的大数据处理技术取得了飞速的发展，并且在电商、金融等互联网领域被广泛的使用，大数据人才的需求也在逐年攀升。

本次设计就是在这种时代背景下，通过搭建完全分布式的Hadoop大数据处理平台实现大数据处理技术典型应用场景。

# 1大数据开发技术环境

## 1.1 大数据处理的起源

2003年谷歌公布了Google File System，正式拉开了大数据处理的开源时代。这是一个分布式文件处理系统，其核心原理是将一个大文件分割成一些小文件，然后再讲这些小文件分布式存储在一些分布式集群上。紧随其后的是2004年被Google公布的MapReduce算法，Google最早使用MapReduce算法来计算他们的搜索索引，MapReduce算法被认为是大数据计算的核心。2006谷歌公开了内部存储大数据的数据库——BigTable的论文，NOSQL数据库开始逐渐展现在人们面前。而GFS，MapReduce，以及Bigtable则成为后面大数据处理技术的三大核心[1]。

虽然Google只公布了其内部大数据处理的理论论文，没有开源其实际的代码，但就是这三篇论文在互联网世界掀起了一场巨大的风暴，很多实验室和开源组织开始依照Google的这三篇论文来开发大数据处理的实用框架，这其中Apache基金会成为领先者，成功研发出了Hadoop，Hbase，Spark等顶级大数据项目，并且全部开源，而这三者也逐渐成为大数据处理的主流产品。

## 1.2 Hadoop

Hadoop是大数据处理的主流框架之一，是一个分布式集群系统，一般架构在Linux集群上，能够稳定高效的处理海量数据文件，而且是开源的，这也是其备受追捧的原因之一。

### 1.2.1 Hadoop的起源与历史

2003到2004年间Google先后公布了GFS和MapReduce的相关论文，受此启发，Doug Cutting等两人利用业余时间真正实现了GFS和MapReduce机制，使Nutch的性能得到很大程度上的提升，随后Doug Cutting极其项目组进入Yahoo，开始专门研究GFS和MapReduce。2005年Hadoop正式进入Apache基金会，成为Lucene的子项目的一部分，2006年Hadoop正式独立出来成为Apache的顶级子项目。在后续的高速发展过程中，Hbase，Hiv，Zookeeper等原属于Hadoop的子项目均正式独立出来成为了Apache的顶级项目[2]。整个Hadoop生态也逐渐完善和壮大起来。

### 1.2.2 Hadoop的核心

Hadoop由两个核心组件组成，一个是分布式文件管理系统—HDFS，另一个而是与分布式文件管理系统相对应的分布式计算框架—MapReduce。

HDFS是GFS的一种开源实现，是Hadoop最底层的元素，是一种架构在Linux集群上的抽象的分布式文件处理系统，负责存储和管理Hadoop集群上的文件。它的核心思想是将非常大的文件根据系统的数量，进行拆分然后再将这些小的文件分布式存储在整个集群的各个节点上，然后再由整个HDFS文件系统统一管理。

HDFS主要依靠NameNode和DataNode实现对整个分布式集群文件的管理。其中NameNode类似于管理者的作用，一个Hadoop集群有且仅有一个NameNode节点，其上面存储的是整个分布文件系统的文件目录以及整个集群的元素数据信息。作为整个Hadoop集群的核心，NameNode的稳定性和灾备十分重要，在解决灾备的问题上，Hadoop提供了一种备份NameNode节点的机制，在Hadoop集群中被称为SecondNameNode。DataNode则扮演的是工作者的角色，一个Hadoop集群中可以有多个DataNode节点，其上是真正存储和管理集群文件[3]。

MapReduce是一种并行的分布式计算的算法模型，有两个核心的过程——Map和Reduce。每个过程都是以Key—Value的形式作为输入和输出。下面以图1.1为例来说明MapReduce过程。

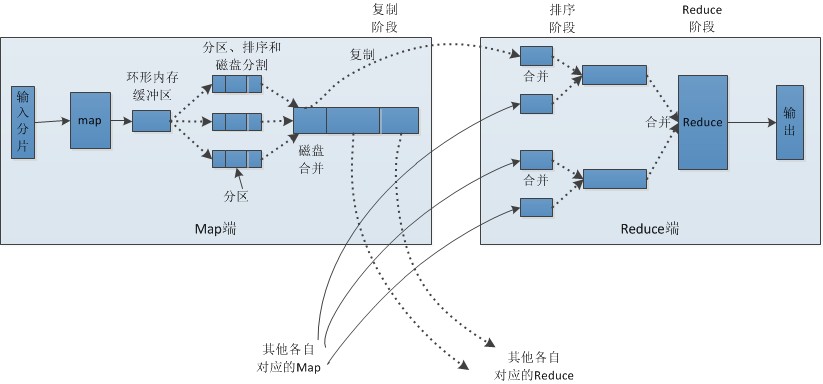


图1.1 MapReduce流程示意图

Map端：在读取数据完成后，HDFS会将整个数据文件切割成一个个的数据块（默认是64M），然后每一个数据块都会被一个单独的Map任务进行处理。Map任务处理完成之后不会立马提交或者写入磁盘，而是会将输出结果放在一个环形缓冲区中，而当数据量过大，环形缓冲区被塞满时，则会将输出结果写入到本地磁盘上。

Reduce端：Reduce会接受到不同Map穿过来的数据，如果接收到的文件不是很大会直接存储在内存中，如果过大溢出，则会将溢出的部分写入到磁盘中，而随着溢出文件的逐渐增大，后台线程会不断的进行合并成更大的文件，合并的过程中会不断的将文件写入到磁盘中， 而最后一次的合并结果不会写入到磁盘中，而是直接输出到reduce函数。

在整个Map和Reduce过程中为了减少写入磁盘文件的大小（从磁盘读取文件很慢），会不断的进行排序然后合并的过程，这也就是为什么会说排序是Hadoop的灵魂。

### 1.2.3 Hadoop的特性

作为一个成熟的大数据处理框架，Hadoop是建立在廉价的Linux集群服务器上的，其对服务器的性能要求不是很高，可以在几台比较廉价的服务器集群上跑起来整个应用服务，因此具有高扩展性，低成本的特点。而且，Hadoop内部对数据是按位存储的，其处理数据的可靠性比较高，加上其Master-Slave的工作模式，即使一台机器宕机也不会对整个集群造成致命的影响。因此对于Hadoop的特点可以简单总结为：高可靠性、高扩展性、高效性、高容错性、低成本。

## 1.3 Hbase

Hbase一种NOSQL数据库，是Google的BigTable的开源实现，其底层依赖于HDFS，是目前应用最为广泛的与大数据处理相对应数据库系统[3]。其与Hadoop之间的关系可用图1.2来表示：

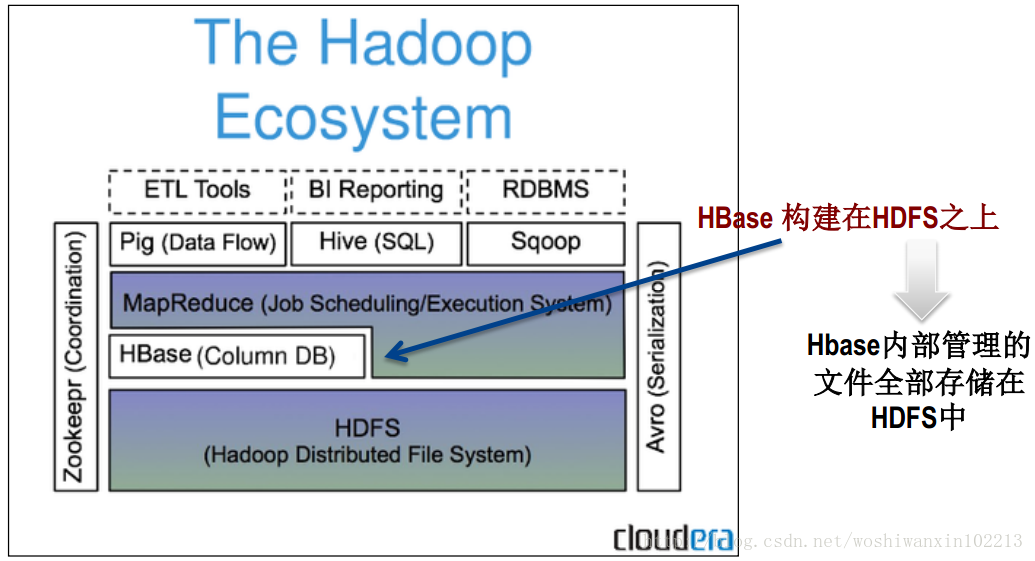


图1.2 Hadoop生态系统示意图

Hbase是Hadoop生态系统中的重要一员，典型的Key—Value架构，Hbase的主要目标是横向扩展，通过增加廉价的商用服务器来加强自身存储和计算数据的能力。

### 1.3.1 Hbase的基本概念

在介绍Hbase的核心原理架构之前，需要了解小Hbase的一些基本概念：

RowKey：Hbase数据库中每一行的主键，是一个字节数组数组（Byte Array），设计的目的是方便快速查找，在整个Hbase系统中RowKey的设计十分重要，会直接影响到整个Hbase的速度[4]。

Column Family：列族，字符串（String），一个列族可以包含有多个列

Column：列，一定属于某一个列族，而且列数目可以动态的创建于添加

Version Number：一个长整型（Long），默认为系统的时间戳，当然也支持自定义

Values：每一列对应的值，也是字符数组型，在Hbase中所有的数据均是以字符数组进行存储的。

### 1.3.2 Hbase的物理模型

Hbase的每一个列族中的数据都会以一个单独的文件存储在HDFS中，而且空值（Java中的Null或者空字符串）不会被存储；每一条数据对应的RowKey和Version Number在列族中都会存储一份；Hbase为每一个数据都维护了一个多级索引，依次是：RowKey、Column Family、Column和TimeStamp。

在实际的物理存储上，Hbase中所有的行都是按照RowKey的字典序进行排序。在数据的存储管理上，Hbase会根据数据量的大小，将整个Table纵向划分成不同的Region，而且是等分。随着数据量的不断增大，这种Region的划分也在持续的进行中。不同的Region会被分配到集群不同的机器上去，然后交由该机器上的RegionServer进行管理，基本示意图如图1.3所示：

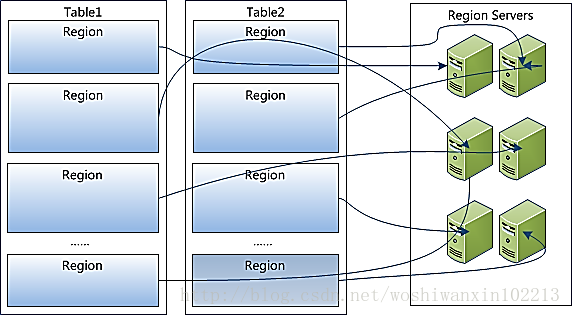


图1.3 Hbase Region和RegionServer

在更细层次的划分上，一个Region由一个或者多个Store组成，每一个Store保存一个Column Family的数据；而一个Store由一个memStore和0个或者多个StoreFile组成，其中memStore是一个内存级别的缓存，其典型应用场景是新插入的数据被读写的频率会高于老的数据[5]；StoreFile包含HFile存储在HDFS上，当memStore中的数据达到一定的阈值的时候会别刷到HFile中[5]。基本关系如图1.4所示：

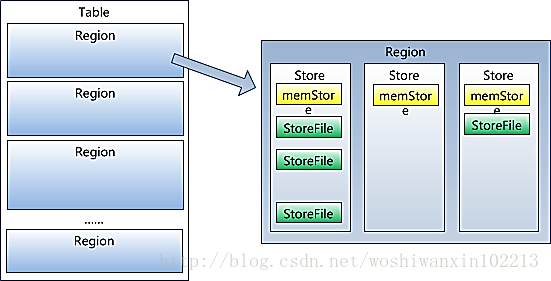


图1.4 Region组成示意图

### 1.3.3 Hbase核心架构及组件

Hbase其核心架构示意图如图1.5所示：

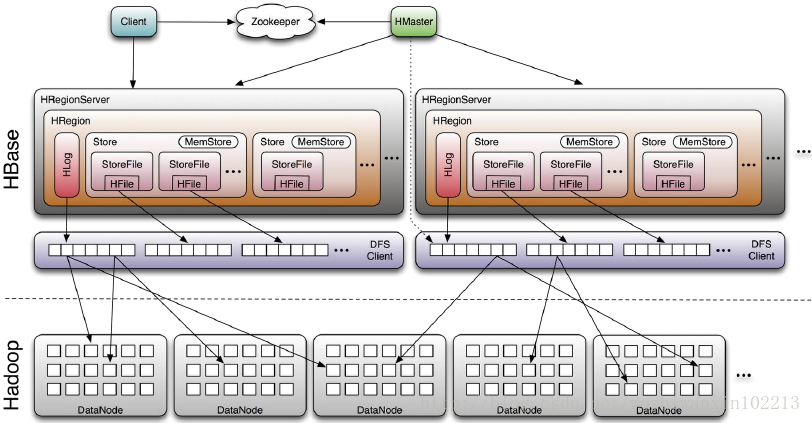


图1.5 Hbase架构图

其中HMaster是整个Hbase集群的管理和调度中心，负责管理RegionServer以及用户对Hbase的一系列操作，同时还负责重新分配发生宕机的RegionServer上的Region，保证整个集群数据的可用性。RegionServer负责管理其所有的Region包括外部的I/O请求，以及分割过大的Region。

图中的Zookeeper也是大数据处理工具的一种，是Hbase集群必不可少的，对于Hbase集群而言Zookeeper主要有以下几个作用：

1. 通过选举机制，保证整个Hbase集群在任何时候有且仅有一个Master，而Master与RegionServer会在启动的时候主动向Zookeeper注册自己的信息；
2. 存储所有的Region寻址入口；
3. 实时监控Region的状态和信息，并且向Master反馈；
4. 存储Hbase的schema以及Table的元数据信息。

## 1.4 Spark

在1.2这一章节中介绍过Hadoop的两大核心HDFS和MapReduce，其中MapReduce是整个Hadoop系统的灵魂，提供Mapper和Reducer两个抽象的编程模型，然后每一个Mapper和Reducer组成一个Job，多个Job传承一个邮箱无环图（DAG）[6]，然后将每个Mapper和Reducer放到Hadoop集群上进行运算得到结果。单这种模式也导致了诸多的问题：

1. 抽象的层次太低，需要手动编写Map和Reduce算法
2. Map和Reduce的中间结果还要存储在HDFS中
3. 高延迟，对于交互式和实时的数据处理支持力度不够

因此，在Hadoop推出之后出现了很多相关的技术对其局限性进行改进，使用比较成熟和广泛有Pig、Spark、Phoenix、Hiv等。这其中Spark是目前最流行的MR数据处理引擎。

### 1.4.1 Spark技术核心

在现有的阶段下，数据处理的模型主要有以下几种——Iterative Algorithms，Relational Queries，MapReduce，Sream Processing。Hadoop就是使用的MapReduce模型，而向Storm则采用的是Sream Processing，而Spark则非常巧妙的将这四者进行了集成，所以Spark几乎可以适用于各种大数据处理的应用场景[7]。

在实际的应用场景下，Spark最为突出的优势就是在海量数据的处理速度上，响应速度在百秒到秒级之间。而相比于Hadoop，在内存处理模式下，Spark的速度几乎是前者的百分之一，即使是在磁盘的读写条件下，速度也能达到前者的十分之一左右。

能让Spark的数据处理速度大幅提升的核心技术点有两个：一是Spark是基于内存计算的，在前面提到过Hadoop在每个Job计算完成后都是讲中间结果放在磁盘中，而Spark则将其放在内存中，大大减少了磁盘读写带来的开销，从而大幅度的提高速度。二是Spark的核心数据结构RDD（(Resilient Distributed Datasets,弹性分布式数据集)，它是一个抽象的数据集合，是一个分区制度记录的集合。一个RDD可以包含有多个分区，一个分区实质上就是一个dataSet，依据不同的RDD操作会产生不同性质的依赖，主要是宽依赖（we dependency）和窄依赖（narrow dependency）。例如map操作会产生narrow dependency，而join操作则产生wide dependency。在基于内存计算的基础上，如果不加处理常常会遇到一个十分现实而且常见的问题——OOM（OutOfMemory，内存溢出），所以这也是Spark这类基于内存计算框架必须解决的问题，Spark的处理方式是，在默认情况下，Spark会通过persist方法将RDD存在内存中，当内存不够的时候就会将RDD写入到磁盘中[8]。

### 1.4.2 Spark生态

从2009年诞生于伯克利大学，到2013年成为Apache的顶级项目，Spark的发展速度令人震惊，而且随着不断的发展Spark自身也在不断的完善中，现在Spark的机器学习、流计算等诸多大数据和流式计算的框架特日趋成熟，并且被广泛的应用。整个Spark生态圈如图1.6所示：

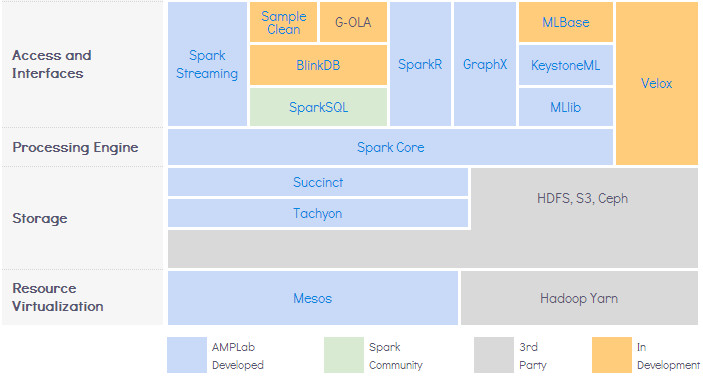


图1.6 Spark生态核心组件

2013年以来，Spark一直都处在高速发展的状态，逐渐成为一个成熟稳定的大数据处理平台，用户的活跃度在不断攀升。Spark的底层实现代码是Scala语言，但是对外提供了包括Java、Python、Scala、R语言在内的种API，能够让不同语言习惯的大数据开发者快速的上手。

# 2系统设计与实现

在了解了大数据的一些基本的理论之后，本次设计结合大数据的基础概念和实际的工作经验，搭建完全分布式的Hadoop大数据处理集群，将Hadoop、Hbase、Spark、Zookeeper等集成起来，再通过Python爬虫代码爬取脉脉网的数据，然后利用已经搭建好的大数据平台进行数据分析。

## 2.1 系统基础架构

本次毕设在一开始准备搭建一个含有5台Linux操作系统的服务器，但由于内存不足等一列的原因（具体原因下文会讲到），后改成4台，虽然服务器减少一台，但基本不会影响整个集群的基本功能。修改后，本次设计的基础架构图如图2.1所示：



图2.1集群架构图

集群的IP地址区段统一设定为192.168.1.71至192.168.1.74，统一的网段是整个集群网络通信的基础，为了方便后续的操作，每台服务器的IP与主机名对应如表2.1所示：

表2.1 集群IP—主机名对应关系

|  |  |
| --- | --- |
| IP地址 | 主机名 |
| 192.168.1.71 | hadoop1 |
| 192.168.1.72 | hadoop2 |
| 192.168.1.73 | hadoop3 |
| 192.168.1.74 | hadoop4 |

hadoop1上的服务主要有：Hadoop的Master—NameNode，Hbase的Master—Hmaster，Spark的Master—Master。

hadoop2上的服务有：Hadoop的备份Master—SecondaryNameNode，Hbase的RejionServer—HRejionServer，Zookeeper。

hadoop3上的服务有：Hadoop的数据节点—DataNode，Hbase的RejionServer—HRejionServer，Zookeeper，Spark的工作节点—Worker。

hadoop4上的服务有：Hadoop的数据节点—DataNode，Zookeeper，Spark的工作节点—Worker。

由于整个大数据集群所涉及到的基础组件比较多，而各个组件的版本也在不断的更新之中，不同版本的组件相互之间可能会出现不兼容的情况，所以在确定好了整个集群的架构之后各个组件的版本选择也尤为重要，本次设计在参考公司的选择后，最后基本信息确定如表2.2所示：

表2.2集群主要组件版本信息

|  |  |
| --- | --- |
| 项目名称 | 版本 |
| JDK | 1.8 |
| Hadoop | 2.7.3 |
| Hbase | 1.2.5 |
| Zookeeper | 3.4.6 |
| Spark | 2.0.2 |
| centOS | 6.6 |
| Python | 2.7.3 |
| Docker | 1.1.2 |

其中，centOS是红帽的一款Linux操作系统，目前最新的版本是7，为了稳定起见最终选择6.6。Docker是类似于虚拟机的软件，后面的文章为对其做一些介绍。

## 2.2 搭建集群

### 2.2.1 Docker介绍

在上文提到，本次设计至少需要四台Linux服务器，由于没有机房，以及没有足够的资金去购买相应的金融云服务器，故本次设计采用的是虚拟机技术， 先购买一个移动硬盘，然后将一个Linux操作系统装入到移动硬盘中，再利用虚拟技术在移动硬盘的Linux构建出整个集群系统。

现今流行的虚拟机技术有很多，而在Linux操作系统上最为流行的是KVM技术，属于一种传统的VM技术。而Docker是从2013年开始流行起来的开源的高级容器引擎，使了开发者可以打包自己的应用然后放在任何流行的Linux操作系统上运行起来。而且容器之间使用的沙箱机制，相互之间完全独立不存在任何接口[9]。是容器技术的代表之作，



图2.2 容器技术与VM技术的对比

如图2.2所示，以KVM为代表的传统VM技术实现资源隔离的方法是使用各自独立的OS，并利用Hypervisor虚拟化CPU、内存、IO设备等实现的。而Docker则不需要对宿主机的硬件资源进行虚拟化的操作，是内核级别的虚拟化技术，运行在Docker上的程序直接使用的就是宿主机的硬件资源，这在很大程度上减少了系统硬件资源的开销。而关于容器虚拟技术（Docker）与传统虚拟机（KVM）的一些简单比较可以用表2.3简单表示：

表2.3 容器技术与虚拟机技术简单比较

|  |  |  |
| --- | --- | --- |
| 特性 | 容器 | 虚拟机 |
| 启动 | 秒级 | 分钟级 |
| 硬盘使用 | MB级 | GB级 |
| 性能 | 接近原生 | 弱于原生 |
| 系统支持量 | 单机上千个 | 一般几十个 |

关于Docker的详细技术原理以及VM技术与容器技术的比较不是本次设计的重点，故也不再详细的阐述比较。需要说明的是，以Docker为代表的容器技术并不会取代以KVM等为代表的虚拟机技术，两者的原理不同导致两者的适用场景也会有所不同。本次设计采用Docker的主要原因是基于它对Linux的完美适应以及对硬件资源的极少消耗。

### 2.2.2 Docker安装

本次设计的硬件资源是一台4核8G的笔记本电脑和一个1T的移动硬盘，为了不影响原有笔记本的硬盘数据，本次设计的所有软件资源要全部部署在移动硬盘上，整个系统是架构在Docker虚拟技术上的，而Docker需要Linux操作系统，所以第一步是给移动硬盘安装Linux操作系统。

Docker宿主机采用的操作系统是centOS7，至于如何给移动硬盘装Linux操作系统，百度上有很多，在此不做详细的讨论。在装完操作系统后，还需要设置本机为静态IP，具体的操作如下：

编辑centOS的配置文件如图2.3所示，



图2.3编辑centOS网络配置文件

编辑文件结果如图2.4所示：

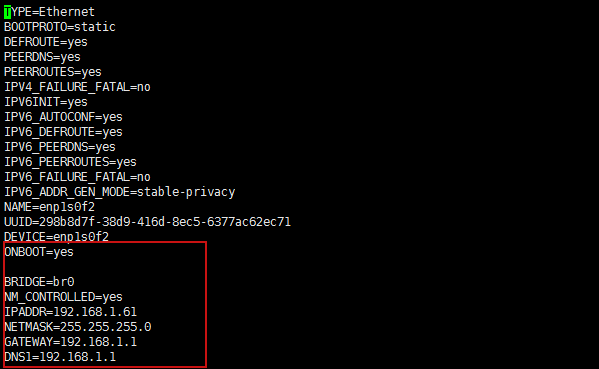


图2.4 centOS网络配置文件编辑结果

图中红色框内是修改或者增加的内容，其中ONBOOT=yes是设置开机启动网络设置，IPADDR=192.168.1.61设置宿主机的IP，NETMASK、GATEWAY、DNS1分贝对应设置子网掩码、网关和DNS服务器。设置完成之后需要重启网络服务，然后可以通过ping命令来检查网络是否通畅。在搞定网络之后，先使用yum update来更新yum的缓存[10]。

一切都做好之后就可以开始安装docker了，具体流程如下：

1) 设定yum，因为系统自带的yum源中的docker版本比较低，所以首先修改yum源，命令代码如图2.5所示：

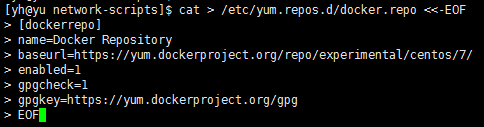


图2.5修改yum源代码

2) 安装docker 引擎代码如图2.6所示：



图2.6 安装docker引擎代码

3) 设置自启动，命令代码如图2.7所示：



图2.7 设置docker开机自启动代码

4) 启动docker服务，命令代码如图2.8所示：



图2.8启动docker代码

5) 查看docker信息，命令代码及结果如图2.9所示：

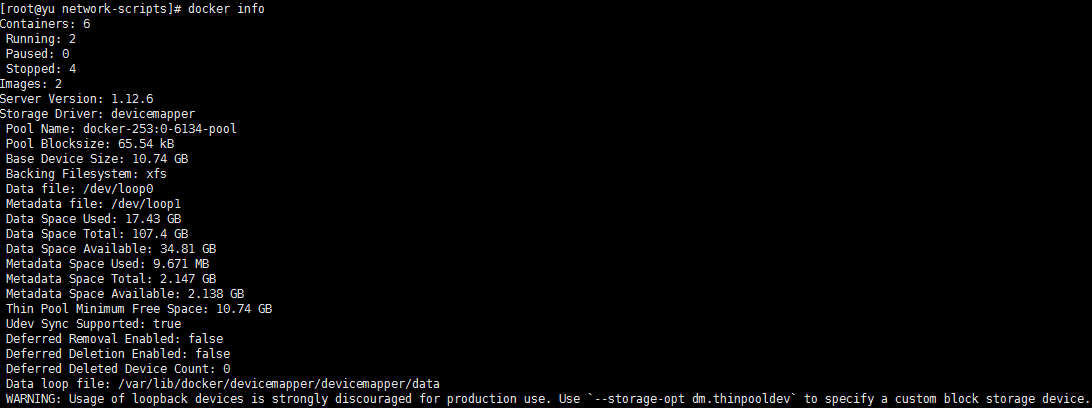


图2.9 docker信息

### 2.2.3 Docker构建集群

依照一开始的设计是需要用docker构建一个含有4台Linux的集群，但在实际的测试和使用过程中发现，8G的内存虽然能构建起集群但后期加上Hadoop、Hbase、Zookeeper、Spark之后内存就会不足，整个集群就会崩掉，所以在实际搭建整个集群中用docker启动两台，然后在另一台电脑上用虚拟机创建两台，然后将主要组件的Master部署在docker开启的两台机器上。

在构建docker集群之前还要解决的是docker宿主机与docker容器之间网络通信的问题。在此需要指出的是，依照docker的设计初衷，最理想的是位每一个容器独立开房相应的端口用以跟宿主机以及外部网络通信。但在这里因为是构建一个完整的集群，所以需要开放的网络端口比较多，因此，使用网桥的方式来打通docker容器和外部的网络[11]，具体的操作如下图2.10所示：

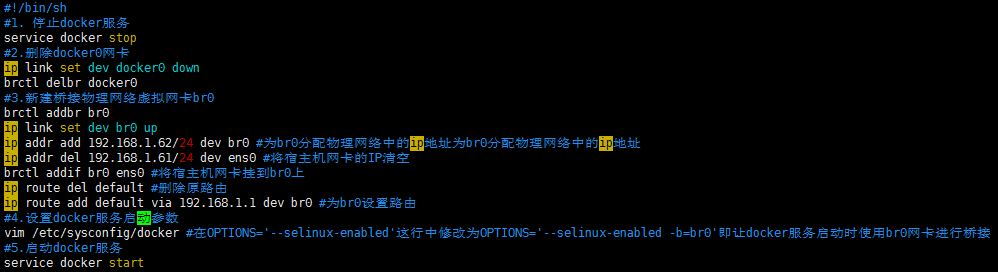


图2.10 设置docker物理网桥

网桥设置完成之后就可以开始使用docker搭建整个集群了，利用docker构建一个Linux虚拟机是很简单的事情，但在实际的应用中，构建大的集群再去一个个的用命令行来解决是很浪费时间的。所以，在本次设计中凡是涉及到集群的相似操作都采用shell脚本的方式来处理。

使用docker构建集群的shell脚本如图2.11所示：

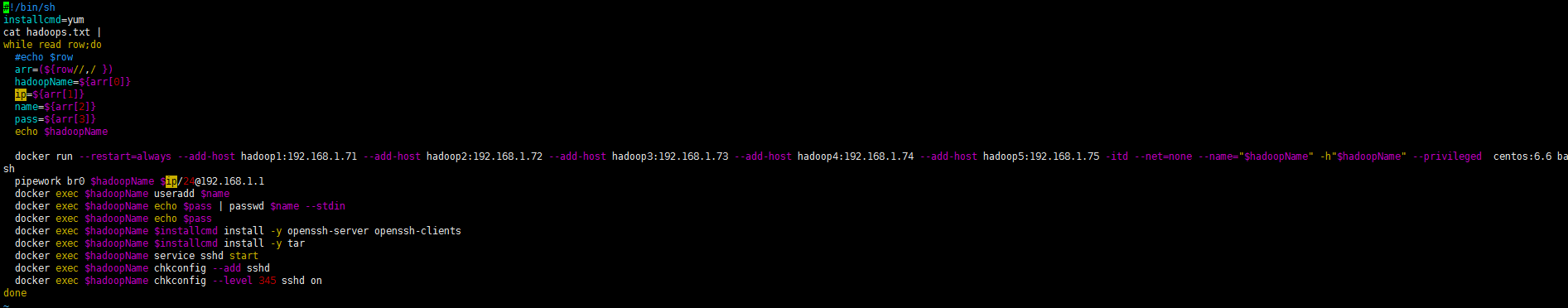


图2.11 docker构建集群shell脚本

其中hadoops.txt文件的内容如图2.12所示：



图2.12 集群配置文件

图中以逗号分割，从左到右依次表示为主机名、IP、用户名、密码。

Shell脚本运行之后docker就会创建出对应数量的容器，每一个都是含有一个最小版本的centOS6.6操作系统，并且已经下载ssh设置为开机启动，同时该脚本还给每一台机器创建一个hadoop用户，初始密码设置为a。虚拟出Linux系统之后还需要做的是关闭每一台机器上的防火墙，以及使用yum下载一些相应的包，具体的命令不再详细列出，基本的思路还是使用shell脚本来循环控制，基本脚本如图2.13所示：

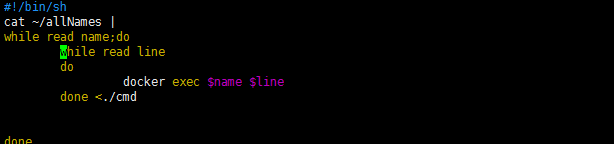


图2.13 集群yum shell脚本

其中allNames就是一个保存docker所有容器name的文件，cmd就是一个保存需要批量执行的shell命令。执行完成脚本之后，可以使用xshell等工具远程登录到hadoop1机器上，后续的操作基本就与宿主机（192.168.1.62）无关了。

至此集群的机器已经准备好，在正式开始搭建大数据平台之前还有一个很重要的工作要做—集群之间的免密登录，由于大数据集群之间需要频繁的通信，集群之间的免密通信就成为关键的一步。ssh免密登录的基本原理就是现在本机生成私钥和公钥，然后将公钥发送到需要免密登录的机器上，这个看似简单，但如果通过手动的一步步操作，一个大小为N的集群需要的操作数是Nx(N-1)次，而且全是重复的工作[12]，为了避免这种无畏的时间浪费，还是采用shell脚本的方式，在hadoop1上编辑脚本如图2.14所示：

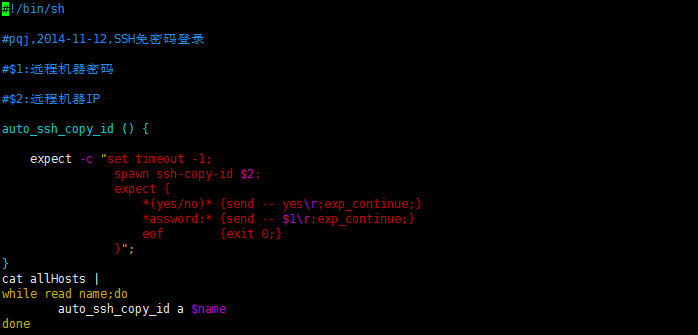


图2.14 集群ssh免密登录脚本

脚本编辑完成之后，通过scp命令将其远程复制到集群的其他机器上，然后即可实现集群之间的免密登录。这里需要说明的是即使是设置了免密登录之后。每两台机器第一次相互通信的时候还是需要进行输入密码验证，之后就可以直接通过ssh命令直接远程登录了，具体操作代码如图2.15所示：



图 2.15 集群免密登录效果

至此，整个集群的基本配置项就已经全部设置完毕，接下来就可以开始依照需要安装对应的大数据组件了。

## 2.3 大数据组件的安装配置

现在成熟的大数据组件相对较多，而且都有相应的适应场景，本次设计依照自己的需求安装Hadoop、Hbase、Zookeeper、Spark几个组件。

### 2.3.1 JDK的安装配置

由于Hadoop组件是需要JDK的支持所以，大数据环境的第一步是安装JDK，这个比较简单，首先去Oracle官网下载对应版本的JDK，本次设计使用的是JDK1.8，下载并上传到相应目录，结果如图2.16所示：



图2.16 集群需要的tar包

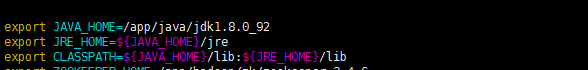
然后使用tar命令解压到相应目录，本次设计为了方便管理所有的相关应用都放在/app/Hadoop/这个目录下，同样的jdk解压到/app/hadoop目录下，解压完成之后还需要设置Java的环境变量，编辑/ect/profile 文件加上如图2.17所示：

图2.17 配置系统profile文件

然后使用source /etc/profile 命令是配置生效，在使用java -version查看jdk信息，结果如图2.18所示：

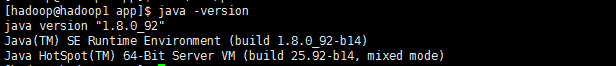


图2.18 查看jdk版本信息

出现jdk版本信息标识JDK已经安装完成，然后再将整个jdk和/etc/profile文件直接通过scp命令复制到相应的Hadoop机器上，这样整个Hadoop集群的jdk的环境已经配置完成，可以进行接下来的工作。

### 2.3.2 Hadoop 的安装配置

先去Apache官网下载相应的Hadoop版本，这里选择的是Hadoop-2.7-3，然后上传到相应目录（/app/hadoop），使用tar命令解压，接下来就需要修改一些配置文件，首先是/app/hadoop/hadoop-2.7.3/etc/hadoop下的core-site.xml文件，是Hadoop的核心配置文件之一，配置文件的最终结果如图2.19所示：



图2.19 Hadoop核心配置文件配置

其中，fs.defaultFS表示NameNode的URL，hadoop.tmp.dir 是Hadoop文件系统依赖的基础配置，这两者最为关键。

然后需要进行编辑的是hdfs-site.xml，主要是针对hdfs的一些配置，这个是整个Hadoop的基础核心，主要配置项也会比较多[13]，主要配置项如图2.20所示：

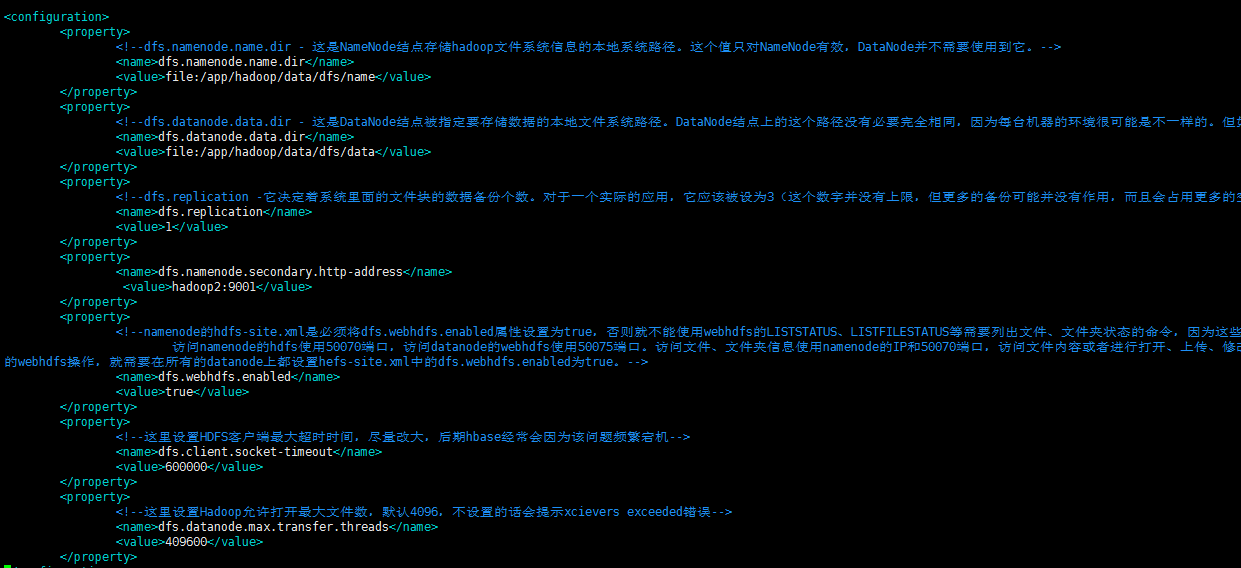


图2.20 hdfs配置文件

接下来需要配置的是yarn-site.xml，yarn是Hadoop0.23以后引入的一个新的MapReduce框架，对Hadoop MR过程中的Job和资源管理有较大的提升，但本设计是直接引入了Spark来替换Hadoop的MR，所以yarn的作用实际上就会减少很多，加上资源紧张，所以在本设计的实际使用过程中不启用yarn，只使用hdfs。但在此需要说明的是，yarn对整个Hadoop的管理和调优的作用还是比较明显的[14]，在实际的生产过程中还是建议开启yarn，根据本设计的实际情况，yarn的配置实际如图2.21所示：



图2.21 yarn配置文件

然后配置hadoop-env.sh、yarn-env.sh的JAVA\_HOME，均是加上如图2.22所示的一句：



图2.22配置JAVA\_HOME

如果没有把JAVA\_HOME追加到path中就需要写实际的绝对路径。

最后是需要配置slaves指定slaves节点，在配置了hosts文件之后慧姐可以配置如图2.23所示：



图2.23 Hadoop集群slave节点配置

在将基本的配置全部完成之后，通过scp名将整个hadoop文件夹复制到其他机器的对应目录（均是/app/hadoop）目录下，然后进入到master（hadoop1）的Hadoop的sbin目录下，使用 ./ start-dfs.sh 命令来启动HDFS，如果需要开启yarn就再使用./strat-yarn命令即可执行，命令代码及结果如图2.24所示：

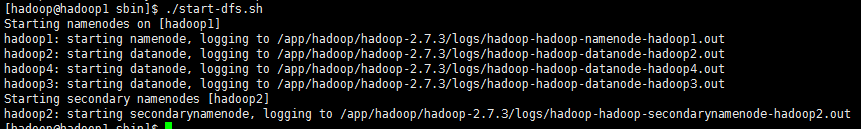


图2.24 启动HDFS

命令行执行时显示一次启动Hadoop的各个节点，hadoop2是备份节点，同时整个Hadoop的日志全部在Hadoop目录下面的logs目录下。集群启动之后，使用jps命令查看集群的进程，命令代码及结果如下所示：



图2.25 查看Hadoop的nameNodej节点

可以看到hadoop1上的nameNode节点已经启动，在ssh到集群的其他节点上查看即可。

还可以访问50070端口来查看整个集群的状态，结果如图2.26所示：

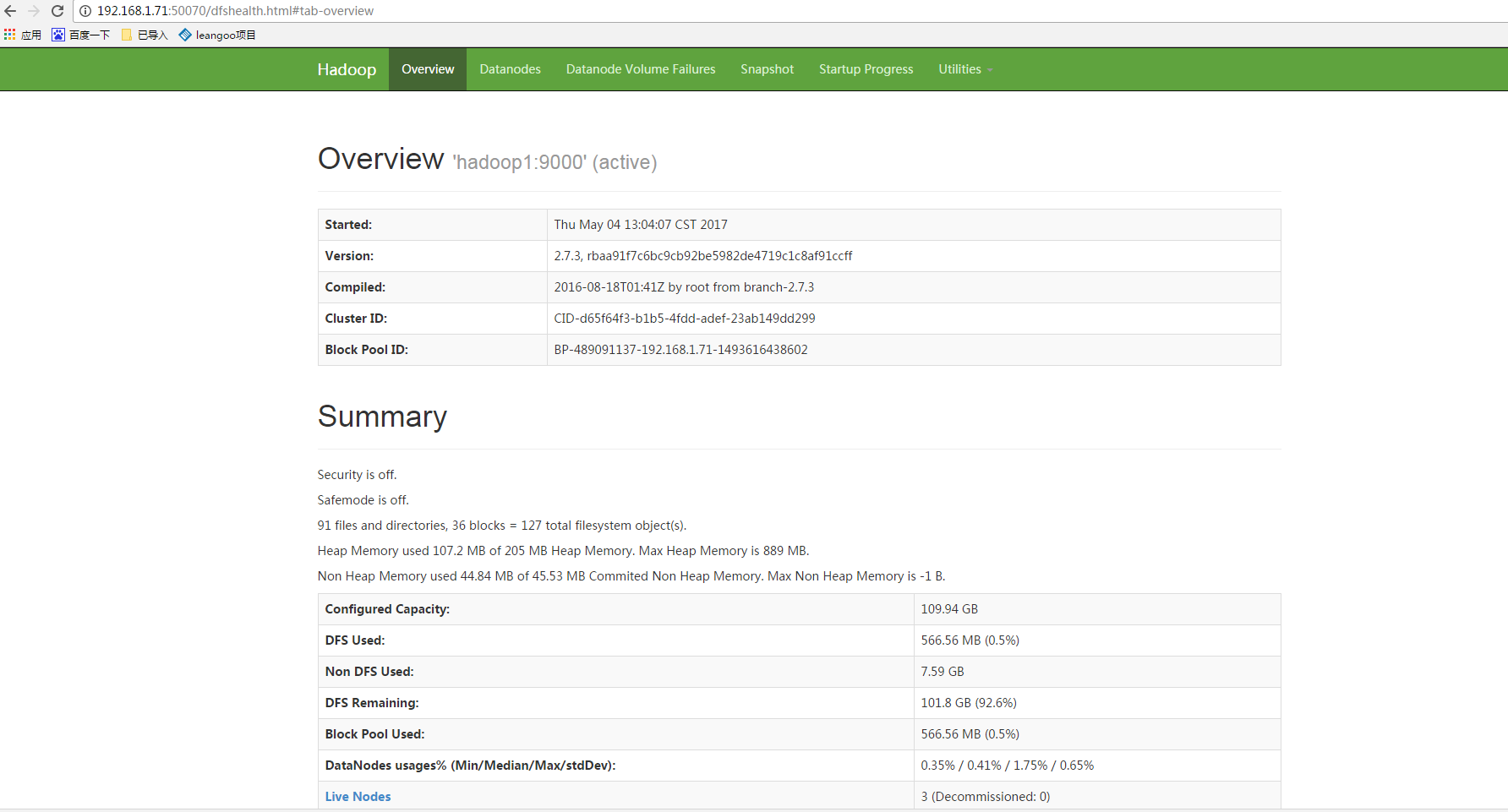


图2.26 Hadoop集群状态Web界面

如果需要查看集群中各个节点的状态则可以点击Web界面上的DataNodes，结果如图2.27所示：

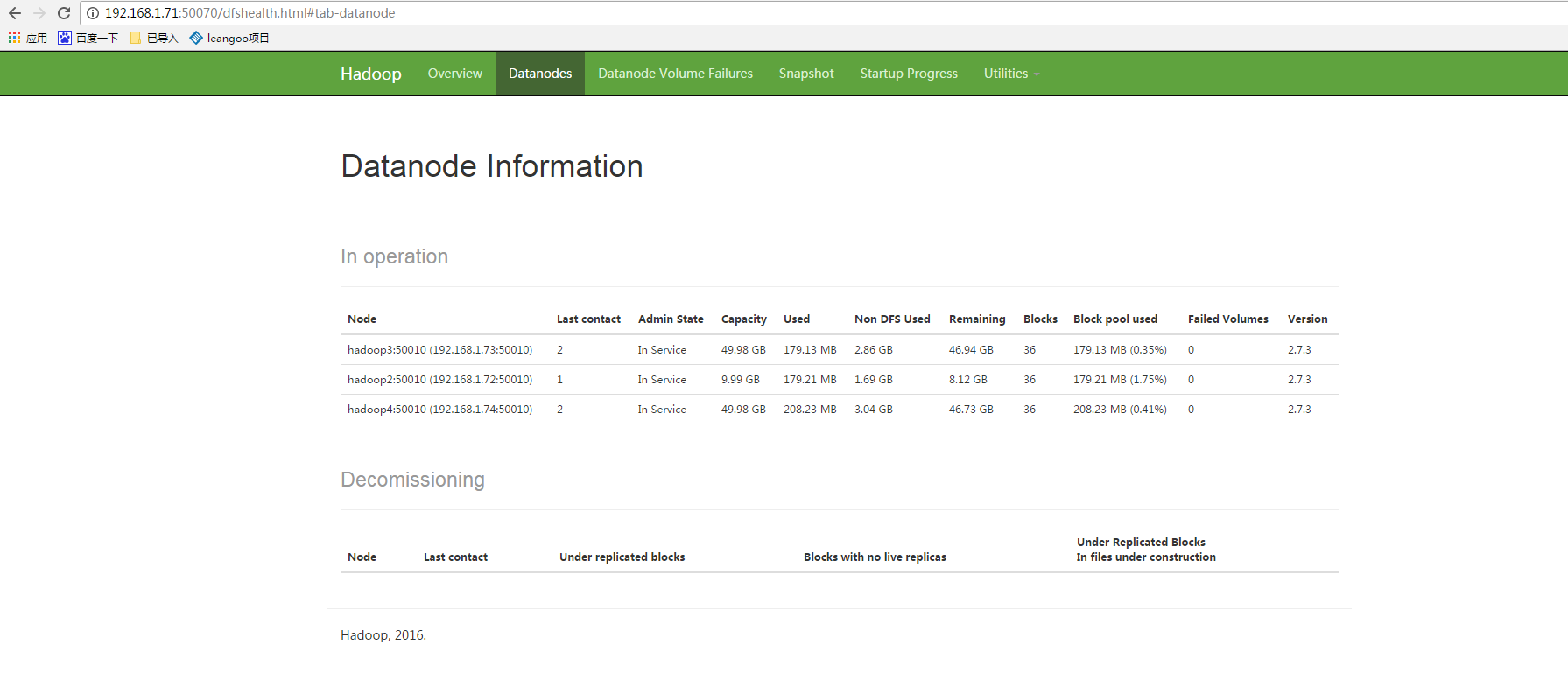


图2.27 Hadoop集群DataNode节点状态Web界面

### 2.3.3 Zookeeper安装配置

因为Hbase依赖于Zookeeper进行管理，所以Zookeeper是Hbase集群必不可少的，Hbase自带的有Zookeeper，但相对于外部的Zookeeper其本身自带的不是很灵活，而且相对来说版本也会比较低，所以一般在完全分布式的集群中都会使用独立的Zookeeper，相对而言Zookeeper的配置比较简单。

首先依旧是到Apache官网下载对应版本的Zookeeper的tar包，本次设计采用的是3.3.6，然后上传到master机器上（hadoop1），解压完成后进入到conf目录开始一些相关配置：

将zoo\_sample.cfg文件复制一份为zoo.cfg，然后编辑信息如图2.28所示：

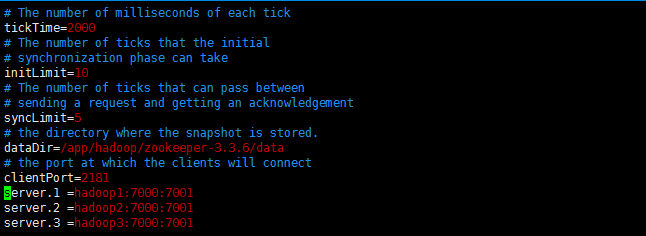


图2.28 Zookeeper配置文件

其中比较容易出错的是dataDir目录，这个目录需要自己手动创建，并且需要在该目录下，新建myid文件（名字必须交myid），根据下面的server.X和自身主机的IP来确定myid文件所填写的数字，例如hadoop1对应的及时配置文件的server.1，那么hadoop1的dataDir目录下的myid文件里面的内容为数字1，而且有且仅能有这一个数字[20]。

以上内容配置完成之后Zookeeper的主要配置即完成，然后同样使用scp命令将其复制到相应的机器上，本设计即为hadoop2和hadoop3，然后修改hadoop2和hadoop3上的myid里面的数字分别为2和3，然后在hadoop1、hadoop2和hadoop3分别在Zookeeper的bin目录下使用./zkServer.sh start启动Zookeeper，命令代码及结果如图2.29所示：

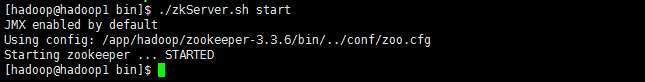


图2.29 启动Zookeeper节点

完成之后使用jps命令查看进程，看到有QuorumPeerMain进程即表示启动成功，命令代码及结果如图2.30所示：

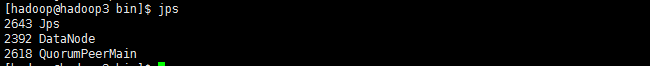


图2.30 查看Zookeeper进程

### 2.3.4 Hbase安装配置

由于Hbase几乎完全依赖于Hadoop，所以Hadoop和Hbase的版本之间的兼容性就显得尤为重要，各个Hadoop版本与之对应的Hbase版本可以在官网上查到，本设计的Hadoop版本是2.7.3可以兼容Hbase1.0以上的版本，最终采用的Hbase版本是1.2.5，同Hadoop的流程一样，下载上传到master（hadoop1）机器上，然后解压，开始配置。

首先是配置hbase-env.sh文件，来配置Hbase的基本环境，重要的四个参数极其配置结果如图2.31所示：

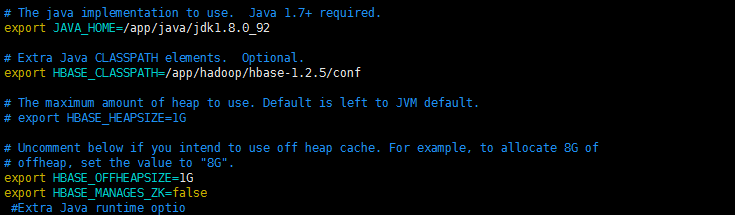


图2.31 Hbase基础环境配置文件

其中JAVA\_HOME是配置jdk的目录，HBASE\_CLASSPATH配置Hbase的自身的conf目录，HBASE\_OFFHEAPSIZE用于指定HbaseRegionServer的最大内存，由于HbaseRejionServer对呢粗是贪婪性的，在实际的应用中8G几乎是基本的配置，但本次设计的内存本身就比较吃紧，加之数据量不会很大，所以被迫设置为1G。将HBASE\_MANAGES\_ZK设置为false代表的是禁用Hbase自带的Zookeeper。

接下来配置的是hbase-site.xml文件，为Hbase的核心配置，由于Hbase本身对内存的要求较高，而本次设计又可以削减了很大的内存，这会在一定程度上降低Hbase的稳定性，因此需要修改多个配置项来维持Hbase本身的稳定。Hbase配置文件结果如图2.32，图2.33和图2.34所示：



图2.32 Hbase核心配置文件



图2.33 Hbase核心配置文件

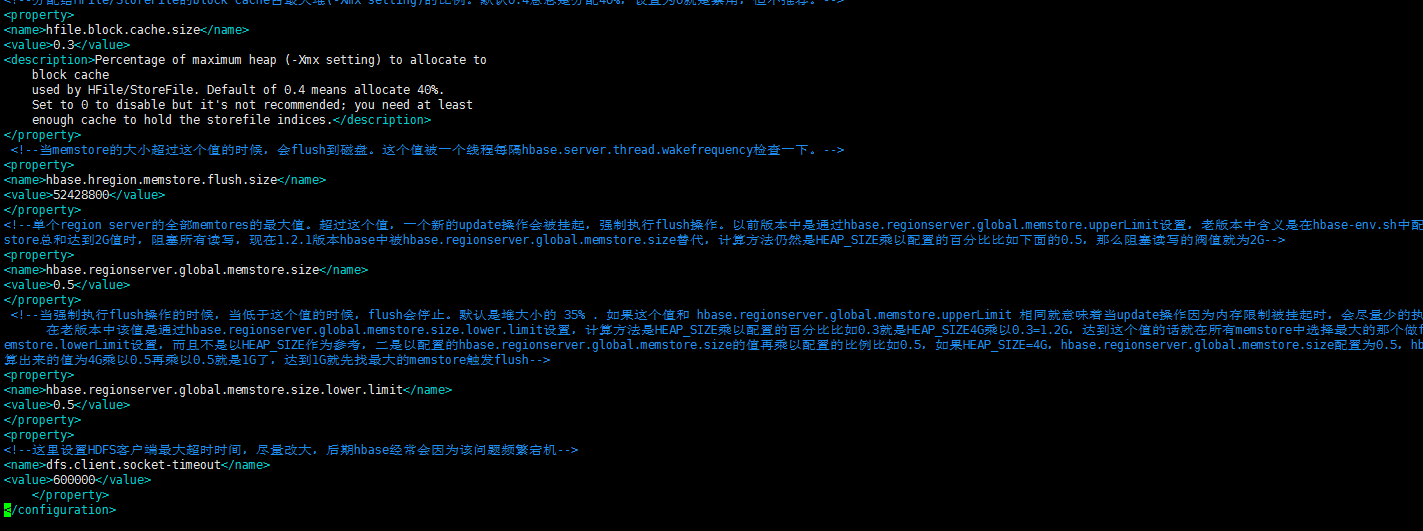


图2.34 Hbase核心配置文件

具体的每一项的含义以及作用在这里不再详细的列出（详细信息参加附录）。

然后配置regionserver文件，输入regionserver所在的机器（hadoop2，hadoop3）。配置完成之后将整个Hbase拷贝到regionserver机器上，然后在master上使用./start-hbase.sh即可启动整个Hbase集群，命令代码及结果如图2.35所示：

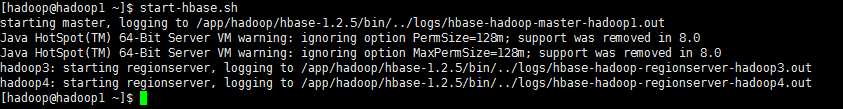


图2.35 启动Hbase集群

图中的两个警告是由于jdk的心特性导致的，对整个环境没有影响；然后使用jps命令查看，命令及结果如图2.36所示：

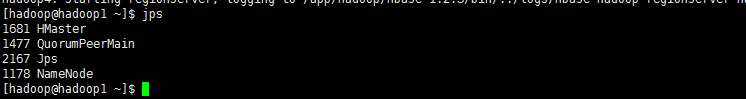


图2.36 查看Hbase进程

master（hadoop1）机器上出现HMaste进程表明Hbase的Master进程已经启动，然后可以ssh到regionServer节点上通过jps命令查看。同样的Hbase也有自己Web监控界面，默认的端口是16010，在浏览器中输入Master地址即可查看[15]，界面如图2.37所示：

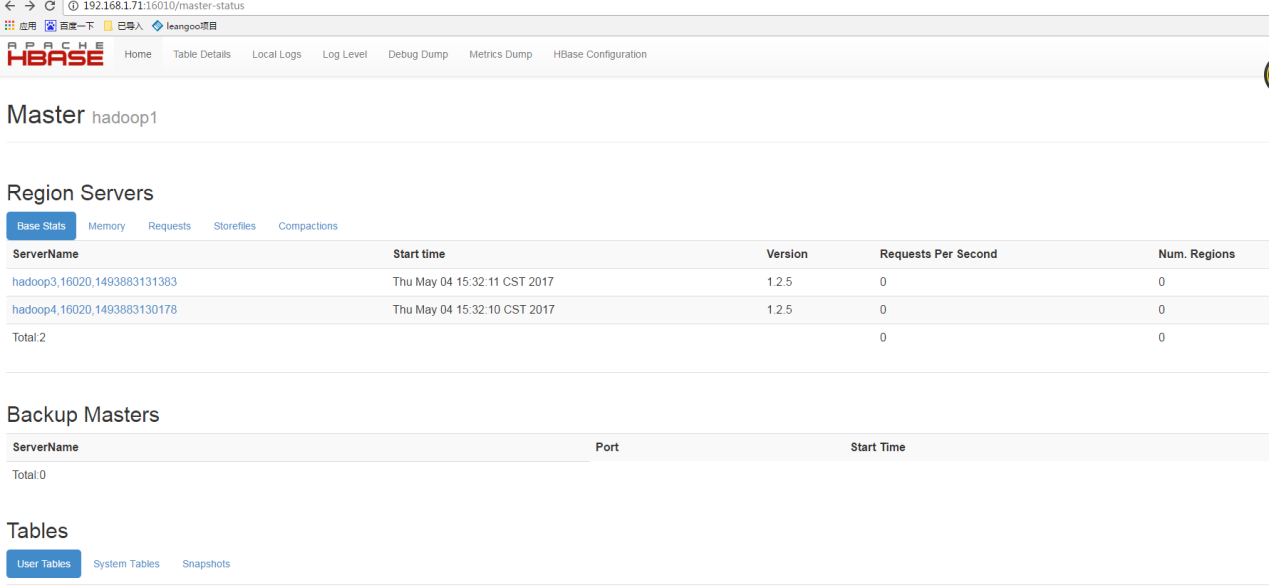


图2.37 Hbase集群Web监页面

### 2.3.5 Spark的安装配置

Spark本身是用Scala语言写的，所以其依赖于Scala语言，Scala语言简单来说就是一种运行在JVM上的函数式编程语言。所以第一步是安装Scala，这个跟jdk的很相似，选择版本在官网上下载相应的版本之后解压，添加到环境变量即可，代码如图2.38所示：



图2.38 Scala环境变量配置

然后直接在控制台输入scala出现版本信息即可，接着同给集群安装jdk一样将整个scala和/etc/profile拷贝包集群其他机器上即可。

Scala安装完成之后就可以安装Spark了，Spark同样在Apache官网上下载，上传到master（hadoop对应的目录下），接着根据实际情况进行配置。首先是spark-env.sh文件主要配置一些spark运行所需要的基础环境，代码及配置结果如图2.39所示：

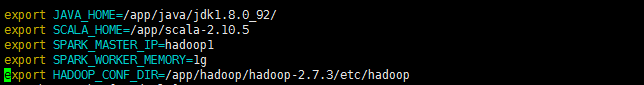


图2.39 Spark环境变量配置

其中SPARK\_MASTER\_IP用来指定Spark的Master节点，而SPARK\_WORKER\_MEMORY则用来限定Spark每个work的内存，这里设置为1G，接着是配置slaves文件用来指定Spark的Worker，本次设计指定的Spark worker节点是hadoop2和hadoop4。配置完成之后通过scp将spark复制到对应的Spark节点上，然后在Spark的sbin目录下直接./start-all.sh就可以了，启动之后通过jps命令即可查看，命令代码及结果如图2.40所示：

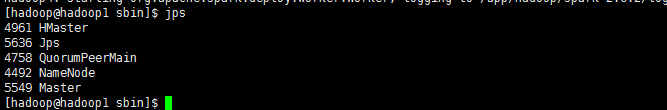


图2.40 启动Spark

master节点上有Master进程，slaves有Worker节点即表示整个Spark集群已经启动完成。同样的可以通过Spark的Web端口查看相应的Spark状态信息，默认的端口是8080，这里修改为10000，在sbin/spark-master.sh文件中配置，配置结果如图2.41所示：

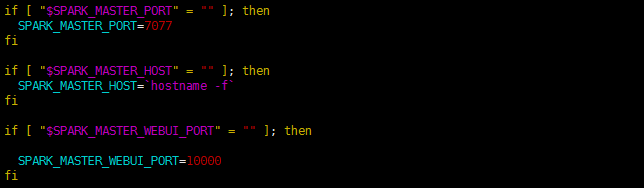


图2.41 配置Spark Master的Web界面端口

端口配置完成之后就可以通过访问Spark Master的10000端口查看界面信息了，结果如图2.42所示：

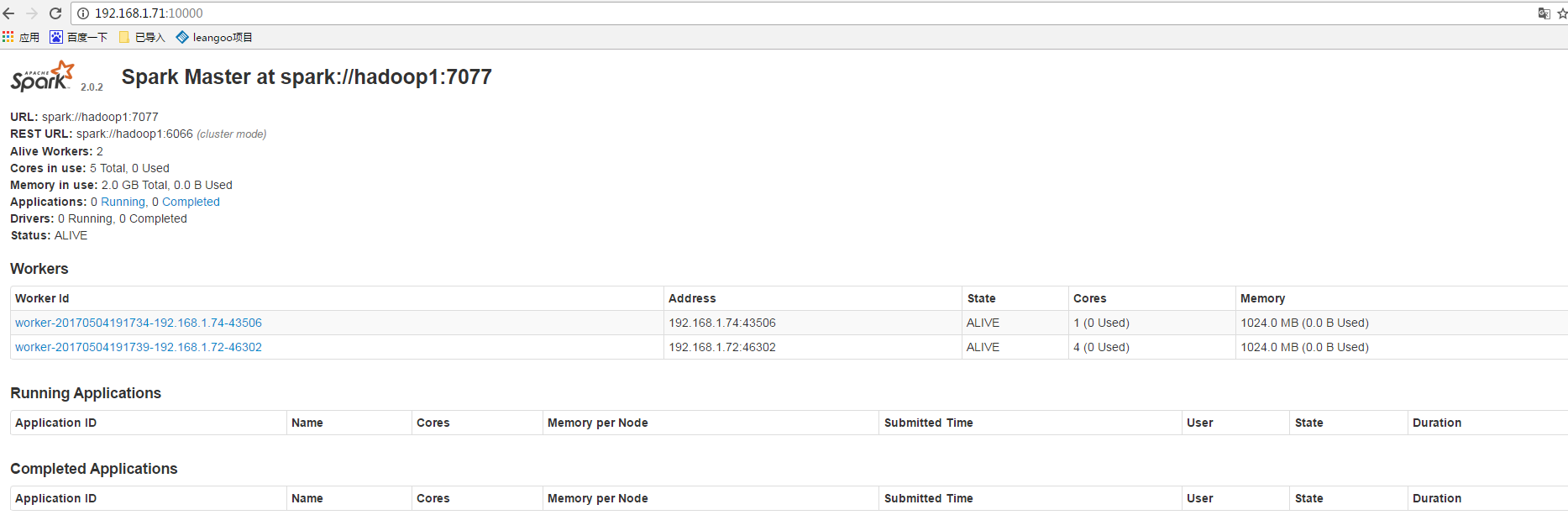


图2.42 Spark的Web监控界面

至此，整个Hadoop的大数据处理平台就已经全部搭建完毕，属于完全分布式的架构，将Hadoop、Zookeeper、Hbase、Spark等主流的大数据处理框架全部整合在一起，接下来就可以向Hbase中写入数据，再利用Spark进行数据分析了。

# 3 脉脉网大数据分析实证

## 3.1脉脉网及其数据接口

数据是大数据分析的灵魂，只有向大数据框架中灌入海量的数据才能真正体现那些大数据处理工具的作用。

脉脉网是专业的行业交流平台，专注于职场的社交，涵盖互联网、金融、社交媒体、房产等多个行业，截止目前脉脉的日活率已经达到30%，搜狗CEO王小川、新东方创始人徐小平、分众传媒CEO江南春、36氪CEO刘成城、今日头条CEO张一鸣、果壳网CEO姬十三等行业大咖已先后成为脉脉用户。

作为新兴的职场社交平台，截止2017年5月，脉脉用户量已经突破2500万，汇聚了各色的白领精英，支持个人专栏、聊天、分布职位以及投递简历等功能，用户数据的价值比较高。脉脉网的界面如图3.1所示：.



图3.1 脉脉网信息页面

脉脉网非常重视用户的联系人关系网络，所以每一位用户信息中都有“看了他的人还看了”这一栏，用来标识用户的一个关系网络。这个结果可以通过一个Https请求拿到一个JSON串。在浏览器输入

https://maimai.cn/contact/interest\_contact/28964?jsononly=1

即可以看到ID为28964的用户感兴趣的人的信息，返回的JSON串如图3.2所示：



图3.2 脉脉网返回JSON串

使用bejson等在线格式化JSON的工具就可以将这个数格式化结果如图3.3所示：



图3.3 格式化之后的JSON串

可以很清楚的看到这是一个十分标准的JSON数组，每一个数组包含一个用户的多个信息，数组中的每一英文字段对应一些信息，由于是英文字段，起具体的含义只能根据经验来推理，现已经确定的字段极其对应的中文含义如表3.1所示：

表3.1 脉脉字段意义对照表

|  |  |
| --- | --- |
| 英文字段 | 含义 |
| id | 编号 |
| name | 名称 |
| company | 公司 |
| city | 城市 |
| province | 省份 |
| position | 职位 |
|  |  |

得到数据返回的JSON字符串后就可以通过爬虫技术来爬取到这些JSON格式的数据然后再解析这些JSON数据，就得到需要的脉脉网用户数据了。在爬虫工作开始之前还需要创建一张表，建表语句如图3.3所示：

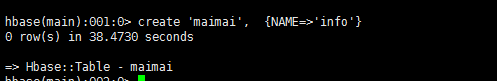


图3.3 Hbase建表语句

## 3.1 数据爬取

现在主流的编程语言几乎都可以开发爬虫程序，但相对于Java这些比较笨重的编译型语言，Python等解释型语言在开发上更为灵巧，而且更加适合于在服务器上直接开发。综合实际情况，本次爬虫采用Python语言。

通过Python爬取的数据最终还是要写入到Hbase中去的，类似于Java向mysql中写数据一样，Python向Hbase中写入数据也需要一个中间工具来进行桥接[16]。本次设计选择的是thrift工具，是由Facebook开发的一款跨语言跨服务的软件，具体的thrift安装全部集成了一个shell脚本如图3.4所示：



图3.4 thrift安装脚本

在安装过程中可能会遇到Python版本的问题，可以根据实际情况做相应的处理。

thrift安装完成之后，并且经过测试之后就可以开始Python爬虫代码的设计工作了。去脉脉的官网查询得知现在的活跃用户有2500万，排除一部分的数据水分，估计1000万应该是有的，面对这么大量的数据只能依赖于多线程才能到达预期的效果[17]。经过多次试验和设计之后，最终定稿的Python脚本代码如图3.5所示：



图3.5 Python爬虫代码

脚本中192.168.1.71的是Hbase的Master节点，getData是爬虫的核心代码，maimai是事先在Hbase创建的Hbase表，info是列族，在创建表的时候创建。虽然使用了多线程但这段代码还是存在一些不足，核心问题是没有使用线程池，Python的线程属于比较鸡肋的存在，所以在实际的使用过程中开了十个线程速度大概也只是提高了一点，不是十分明显，Python也有比较成熟的线程池框架，这里不做研究了。只好采取这种比较慢的操作。然后将其复制到另外一台Hbase的RejionServer上运行，最终的结果是花了一天的时间爬取了大概100万的数据量。由于时间的限制，爬虫的工作就先停下来，开始数据数据的一些处理工作。

## 3.2 数据处理

由于是在Spark上进行跑批任务，Spark虽然支持多种语言，但由于其本身就是用Scala语言编写的，所以本次设计也采用Scala作为基础语言，使用的工具是IDEA。在IDEA上安装Scala插件后可以完全依赖于maven管理所需要的包，然后既就可以进行开发了[19]。工程的主体架构和一般的maven工程没有很大的区别，核心pom.xml配置文件比较多，详情见附录。

工程的基础环境搭建好了之后，在开始编辑任务代码之前需要，准备一些基础的公共函数放在util包里面，方便后面调用；base包存放封装的代码框架；constant包则存放一些基础的静态变量；job则存放整个工程的任务代码。每个包的具体代码详见附录。工程整体代码结构如图3.6所示：

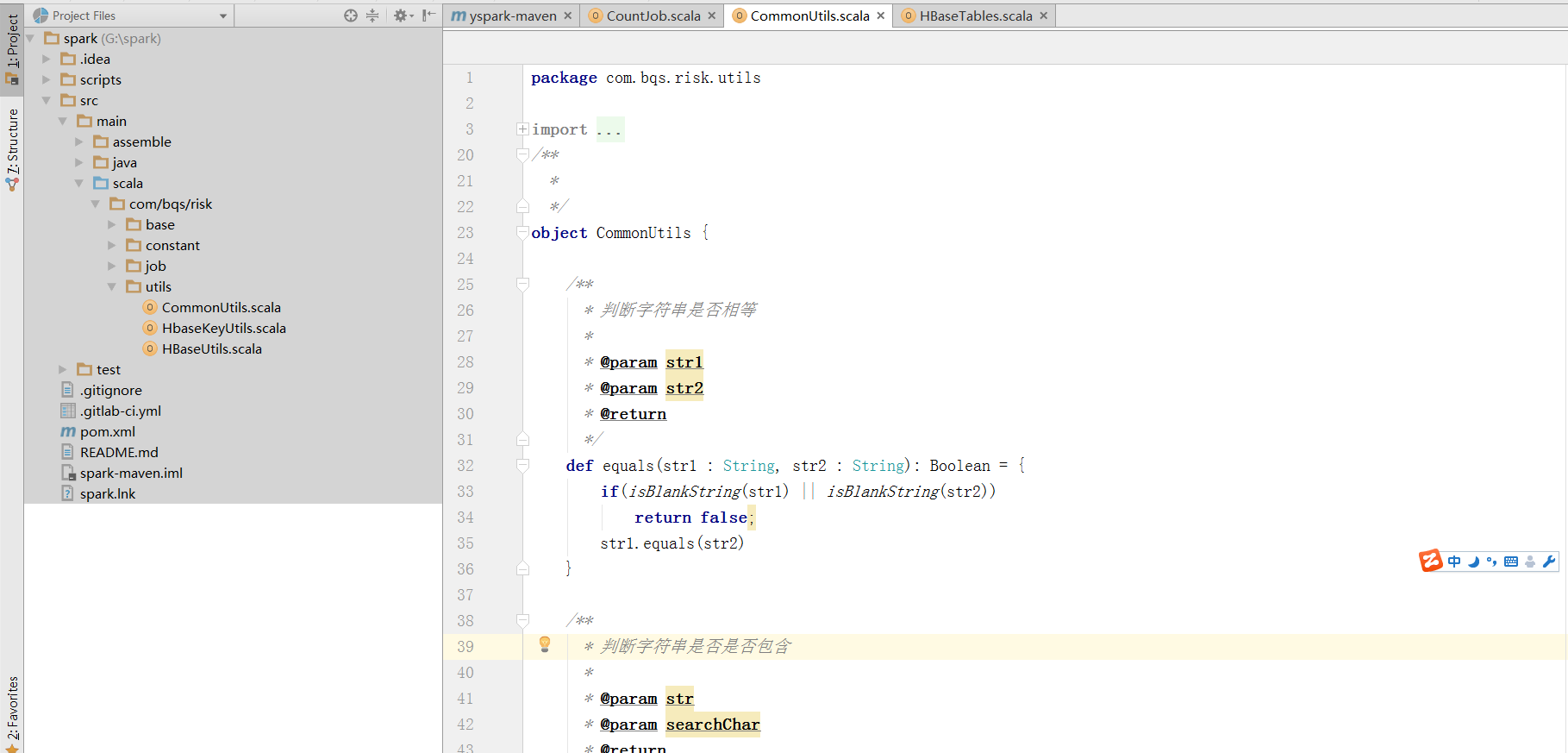


图3.6 spark工程代码结构

基础的代码环境准备好之后就可以开始根据自己的需要来编写相应的Scala代码从而完成所需要的数据分析与计算。由于时间的关系，在本次设计中，只开发了几个基础的数据分析，分别是地域分布、公司分布、行业分布、话题信息、积分分布、匿名与实名认证信息、性别和职位几个特征属性的分布情况。以低于分布中的城市分布为例，其Scala代码（部分）如图3.7所示：

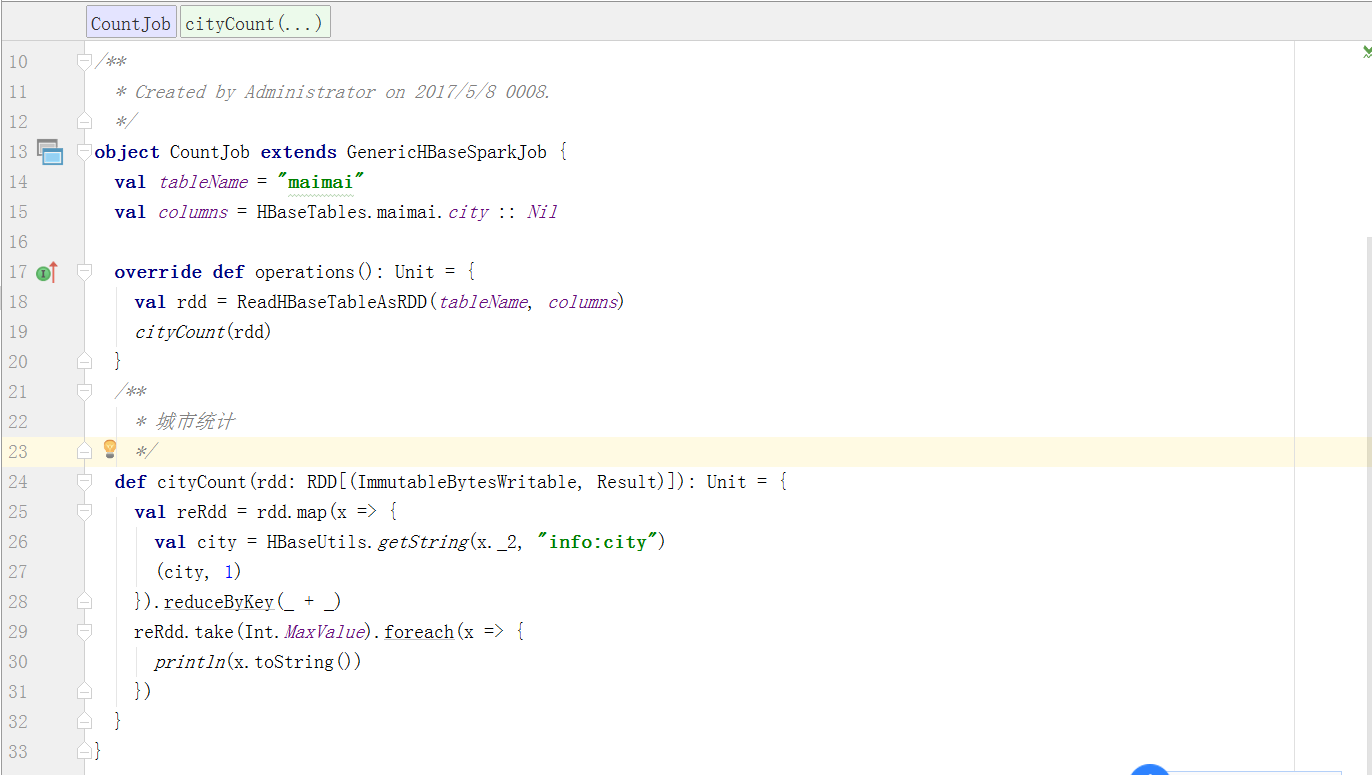


图3.7 城市统计代码

图中的代码很简单，却也突出了Scala语言的几个典型特征，首先Scala的代码中一般使用的是val作为变量的关键字，表示的是常量，一旦定义就不会被改变，如果一定要使用变量的话关键字是var；其次是十分简洁的语言风格，整个任务是由简单的几行代码，却将大数据的map和reduce过程全部囊括其中。上面代码中结果核心部分一个是ReadHBaseTableAsRDD函数，这是自己封装后的函数，在父类GenericHBaseSparkJob中，详细代码如图3.8所示：

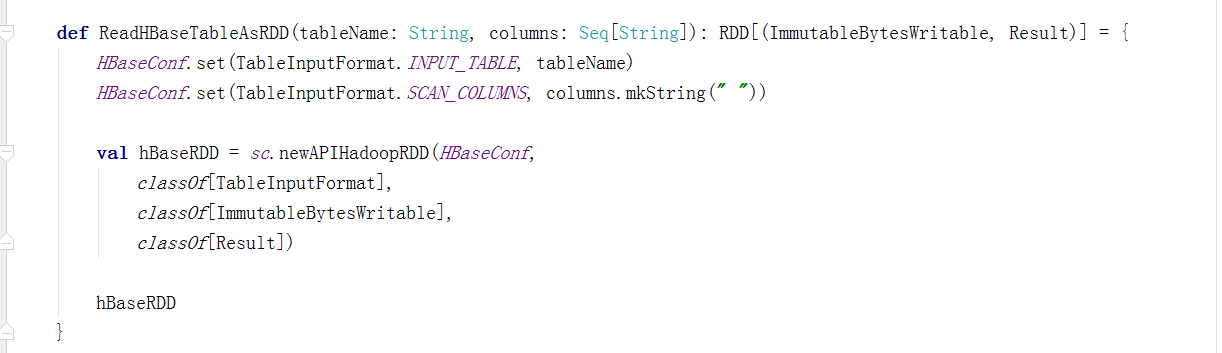


图3.8 ReadHBaseTableAsRDD代码

主要的流程是通过 HBaseConfiguration设置Hbase的基本信息，然后调用Spark的API来实现读取Hbase并且转换成RDD；另外一个则是RDD的map和reduce函数，分别执行map和reduce过程。

Scala代码完成后需要做的是运行这些代码来进行数据分析的计算，一般来说有两种方式，一种是通过IDEA建立Spark和Hbase的操作连接，然后直接在IDEA中运行代码。另一种是讲代码打包成jar然后提交到Spark集群上去，通过spark-submit命令来进行执行。后者在真正的企业级开发中是主流的形式，因为在企业级的数据集群中，数据仓库是极其核心的，不可能随便对外开放，一般是打开几个特定的端口用于进行出数和入数，但需要提前将Spark运行所需要的jar包上传到Spark集群上，然后配置Spark制定的jar路径。本次设计采用的是利用IDEA的方式，更加的快捷简单。首先是要将整个工程打包上传到Spark集群上，然后是由spark-submit命令提交，图3.9显示的是一个完整的spark-submit的提交命令：

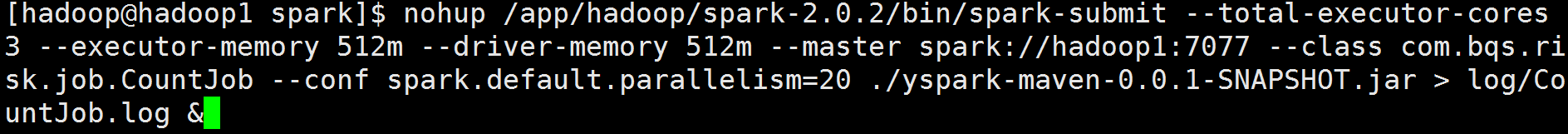


图3.9 spark-submit提交命令

其中 /app/hadoop/spark-2.0.2/bin/spark-submit是spark-submit的路径，--total-executor-cores设置这个任务的核数，--executor-memory设置worker的内存，--driver-memory设置driver的内存，--master设置Spark的master，--class设置该任务的源码路径，--conf 设置基本配置yspark-maven-0.0.1-SNAPSHOT.jar设置jar包的路径，后面的这是重定向log的地址。输入提交命令之后就可以在spark的Web界面上看执行动态。结果如图3.10所示：

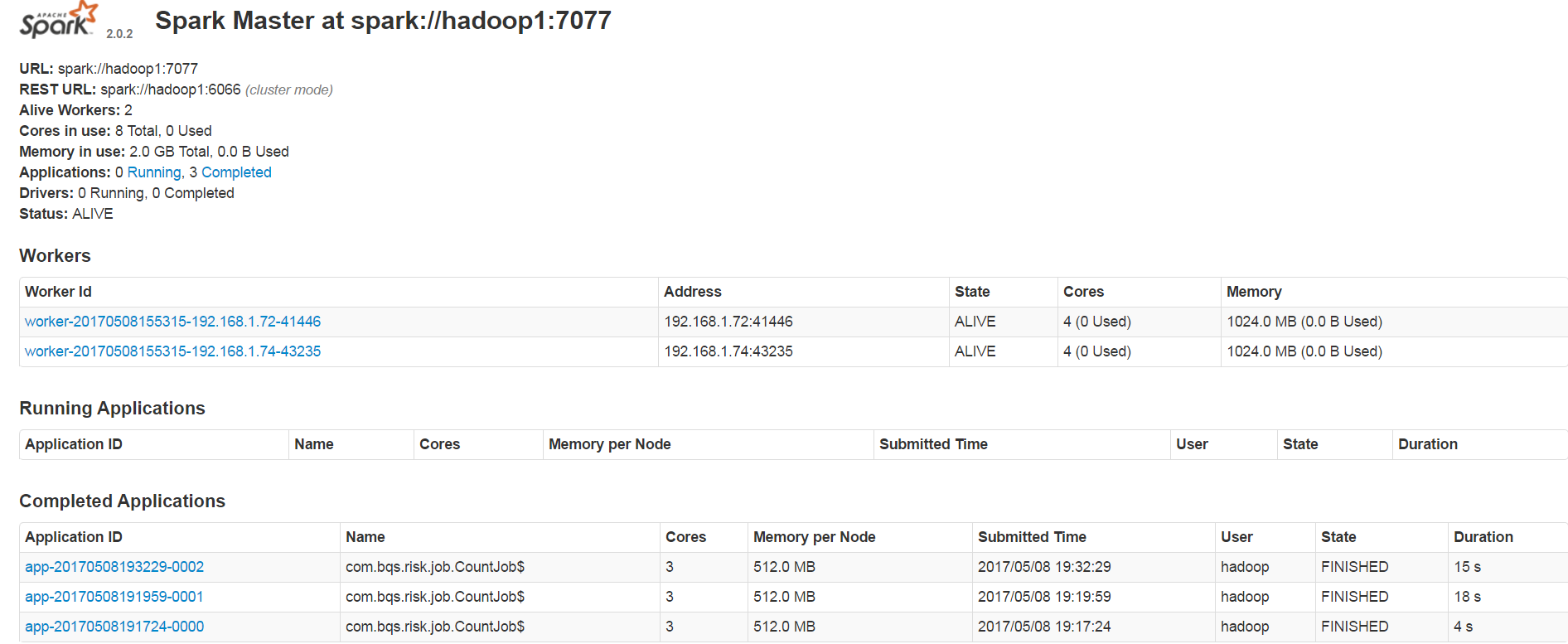


图3.10 spark提交任务后的Web界面

界面中Workers列出了Spark集群中所有Worker的信息，从左到右分别是地址，状态，核数和内存大小；Running Applications列出的是正在运行的spark任务；Completed Applications列出的是已经完成的spark任务（无论是失败还是成功）。

在本次设计中将Spark的计算结果输出到日志文件中，输出结果是key—value形式，然后将这些数据复制到js文件中后面的数据前端展示做准备。

完成第一个任务的计算之后，会发现用户的城市、省份、职业等基本信息分布的计算逻辑都是一样的，所以可以在原来代码的基础上抽象出函数通过传参数和循环来完成相应的计算操作。最终的计算代码见附录。

## 3.4结果展示

数据展示是大数据处理过程的最后一步，但同样至关重要，说到底前面的数据预算处理得到的结果往往是一个文件或者一段数据而已，然而仅仅是做到输出数据这一步是不够的，展示给客户或者上级看的结果应该是更加清晰和直观的，因此一个优秀的数据展示结果在数据处理环节是必不可少的。

### 3.4.1数据展示的前端框架

数据展示的前端框架有很多，本次设计选用的是echarts，是由百度开发的一款前端框架，能满足基本的前端图表展示需求。基本的思路是根据前端展示的需要，将统计的结果转成相应的js格式，然后在HTML页面引入即可。最终的整体结构如图3.11所示：

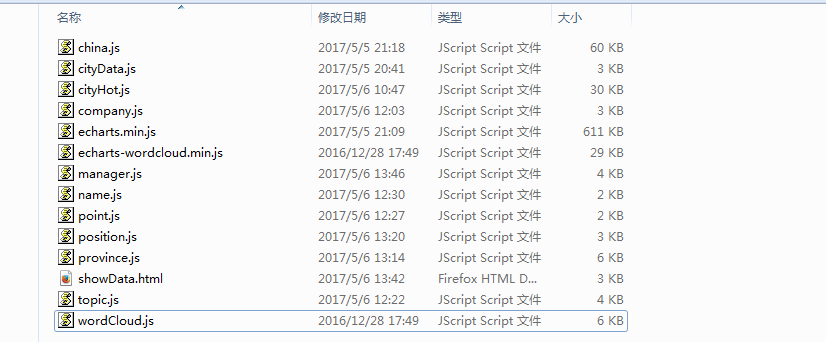


图3.11 数据展示工程js

其中china.js、echarts.js和echarts-wordcloud.min.js是界面展示所需要的基本js。showData.html是整个数据输出的界面，代码如图3.12所示：



图3.12 数据展示HTML代码

整体的结构比较简单，通过不断的引入js来实现效果，避免了所有的数据都在一个HTML界面中显得代码十分的臃肿。

### 3.4.2城市分布

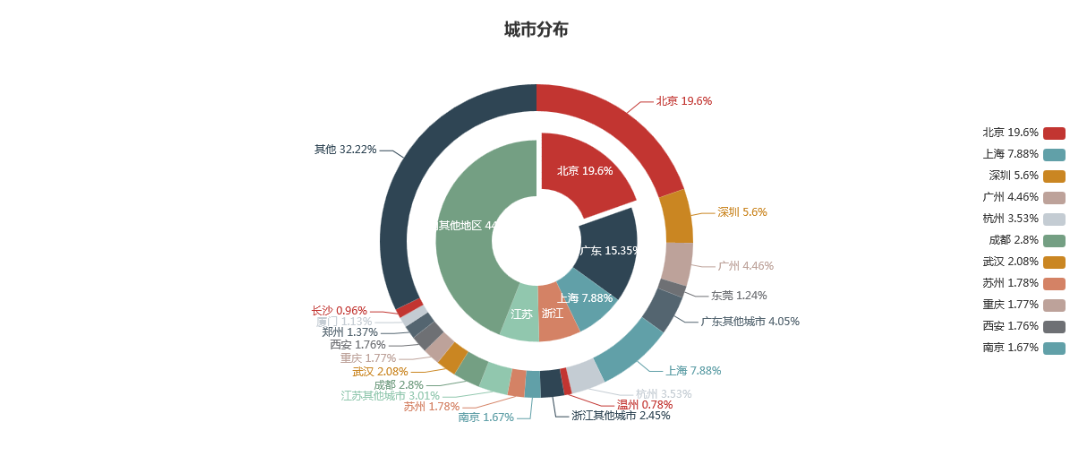


图3.13出城市分布饼图

从图3.13可以看到，脉脉网的用户主要集中在北京、上海、广州、深圳等经济比较发达的地区。（其他地区之所以会占到如此高的比例，主要是由于长尾效用，因为展示界面的限制，当数量低于10000的时候就统一归为其他）。除了饼图之外，城市分布的热力图也能很直观的展示统计的结果，如图3.14所示：

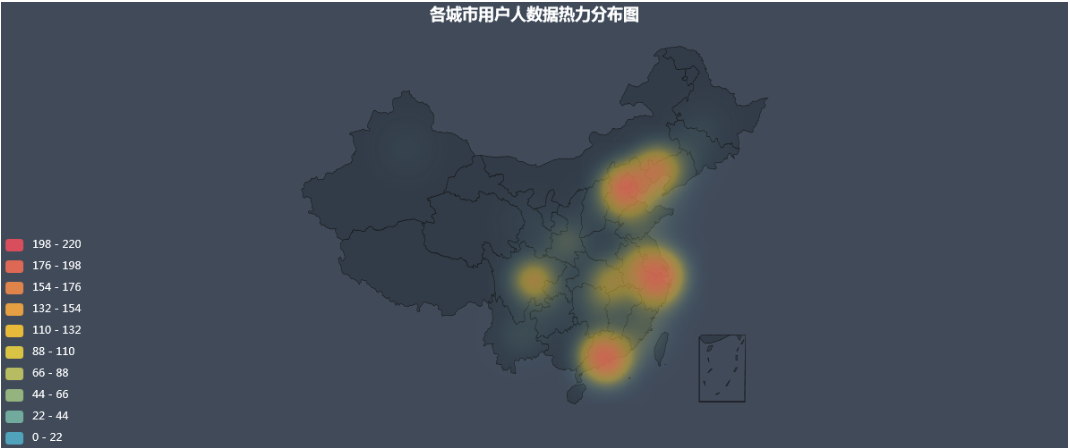


图3.14 城市分布热力图

需要说明的是在热力图中由于显示的问题需要对数据再做进一步的处理，主要是因为整个数据分布的极差和方差比较大，如果不做数据的缩放处理整个热力图就会显得很不协调，连整体的基本特征显示都会有问题。

### 3.4.3省份分布

在城市的基础上省份分布采用同样的方式，界面效果如图3.15所示：

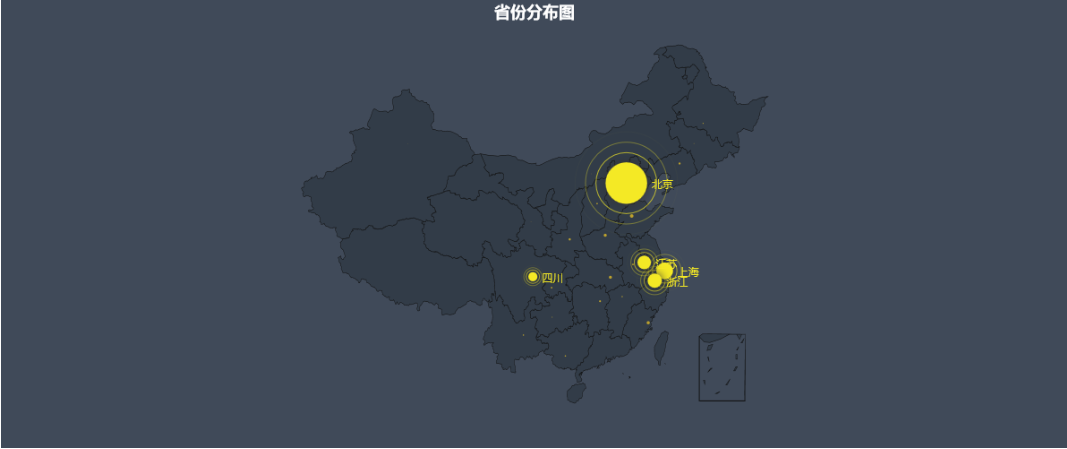


图3.15 省份分布图

可以看到的是数据的整体体现跟城市的基本一致，北京地区最大，其次是浙江和上海一带，另外值得注意的是四川，显得比较突出，很有可能是脉脉网下一个流量的爆发点[18]。

### 3.4.4公司分布

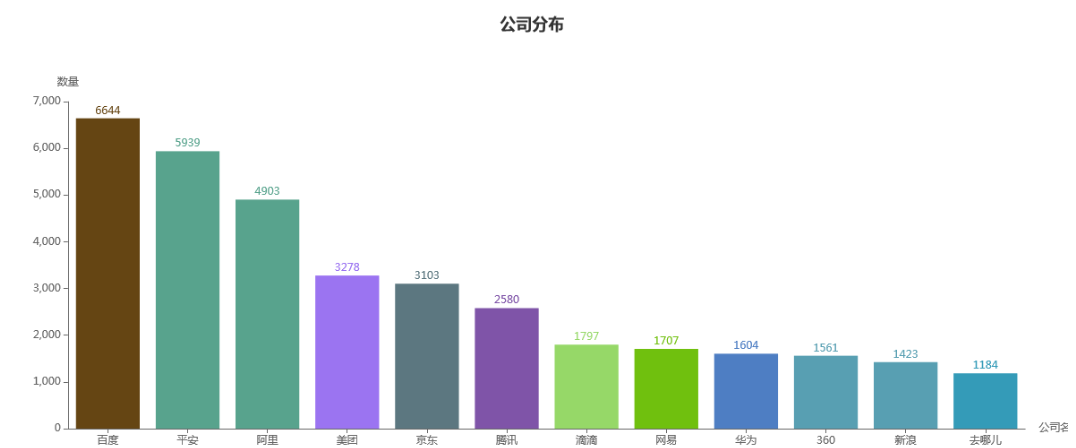


图3.16公司分布

从图3.16可以很直观的看到，百度独占鳌头，平安紧接而至，阿里、美团、京东、腾讯分列3至6位，这就很明显的看出脉脉的主要人群是来自互联网公司，这是一个很重要的特征，由此可以扩展到用户群的基本消费特征和行为偏好，这就属于一些更深层次的用户画像分析了。由于篇幅和时间的限制这里就不再详细的研究。

### 3.4.5职位的分布

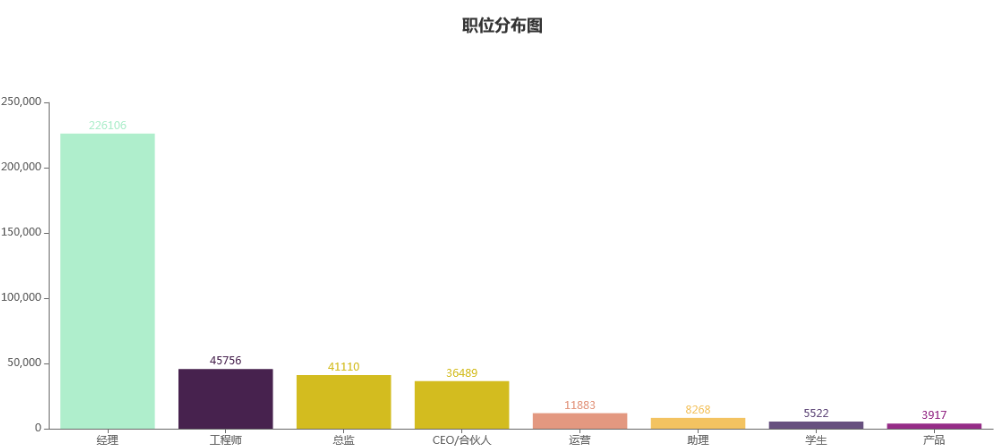


图3.17职位分布

从图3.17可以看到，很让人意外的是经理占到了将近一半的量，工程师排在第二不让人意外，毕竟互联网企业最多的就是工程师，而总监和合伙人的量也很大。这对脉脉网来说绝对是一个令人惊喜的数据，这说明脉脉网不仅仅是一个交流的平台，其用户结构明显偏中高层的管理者，这使得其用户的特征会更加的明显。

为了弄清楚更深一层次的职位分布情况，接下来特地选了经理做进一步的跟进分类分析，结果如图3.18所示：

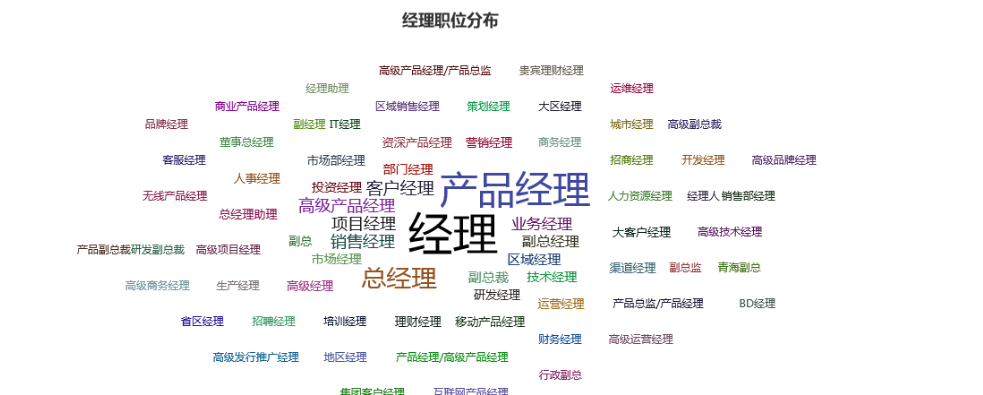


图3.18经理职位分布图

从图中的结果来看，经理还是很多，说明很多用户并没有详细自己的职位信息，但还是可以看到产品经理、项目经理和客户经理还是比较多的。

### 3.4.6时间分布

在前面的分析过程中可以很明显的感觉到脉脉的用户集中在互联网公司，为了验证这个，选择话题时间做进一步的分析，因为以互联网公司的规律，网上活动的时间应该集中在后半夜，而且应该是1到2点最为活跃，最终的统计结果如图3.19所示：

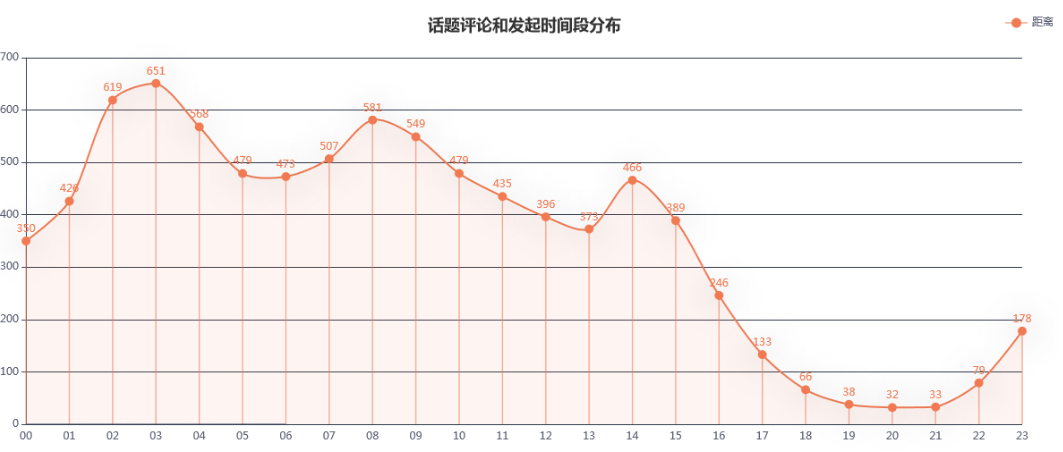


图3.19 话题评论和时间分布

结果也验证了之前的设想。12点以后的活动量一直居高不下，四点钟开始下滑，在8点钟又达到一个次高峰。突显出了脉脉网用户的活跃时间特征。

# 4 总结

至此，整个工程全部完结了。首先是利用Docker的虚拟技术构建了四台Linux服务器，然后在这四台机器上安装和配置Hadoop、Hbase、Zookeeper、Spark等大数据计算的相关组件，之后利用Python爬去了脉脉网近100万的用户数据，最后利用Spark对这些数据进行分析并且通过HTML进行数据展示。整个流程基本上是企业级大数据开发的基本流程，论证了整个大数据平台的可用性与实战性，同时通过对脉脉网用户数据的分析对脉脉网的用户特征有了进一步的明晰。

通过这次的毕设实战，我对整个大数据系统也有了更进一步的熟悉，同时Linux和shell脚本也更加熟练。当然还是存在一些不足，首先是由于初始的估计不到位，导致系统的内存不够，不得不使用虚拟机搭建另外两台服务器。其次是，整个大数据系统的稳定性还没检验，承受能力也还没有详细的说明；还有是对整个系统的整体认知上，由于工作的进一步的深入，发现利用Spark读取Hbase的做法可能不是很明智，拖累的整体的速度，从HDFS直接捞取数据可能是更加明智的选择。最后是在数据的分析上，还是相对浅显的分析，还可以利用Spark自带的图计算功能进一步分析整个脉脉网的用户关系，获得脉脉网更加精准的有用户画像。

# 参考文献

[1] Mark G.Sobell.Red Hat Linux指南[M].北京:人民邮电出版社,2010

[2] James Turnbull.The Docker Book[M].北京:人民邮电出版社,2014

[3] Arun Murthy,Vinod Kumar,Vavilapalli,Doug Eadline.Apache Hadoop YARN[M]. Washington :Addison-Wesley Professional,2014

[4] Tom White.Hadoop The Definitive Guide, 4th Edition[M]. Washington :O'Reilly Media,2015

[5] 徐鹏. Hadoop 2.X HDFS源码剖析[M].杭州:电子工业出版社,2016

[6] Lars George. HBase[M]. Washington:O'Reilly Media,2011

[7] John Ratey. Spark[M]. Washington : Your Coach In A Box,2009

[8] 陈云亮.分级存储系统中基于进化算法的数据管理与保护关键技术研究[D].华中科技大学,2013

[9] Nick Dimiduk,Amandeep Khurana.Hbase实战[M].人民邮电出版社,2013

[10] Tom White.Hadoop权威指南[M].北京:清华大学出版社,2015

[11] Lars George.Hbase权威指南[M].北京:人民邮电出版社,2013

[12] 吴伟.海量存储系统元数据管理的研究[D].华中科技大学,2010

[13] Liping Xiang,Yinlong Xu,John C. S. Lui,Qian Chang,Yubiao Pan,Runhui Li.  A Hybrid Approach to Failed Disk Recovery Using RAID-6 Codes[J]. ACM Transactions on Storage (TOS),2011:26~36

[14] Liping Xiang,Yinlong Xu,John C.S. Lui,Qian Chang.  Optimal recovery of single disk failure in RDP code storage systems[J]. ACM SIGMETRICS Performance Evaluation Review . 2010:34~39

[15] Bianca Schroeder,Garth A. Gibson.  Understanding disk failure rates[J]. ACM Transactions on Storage (TOS),2007:27~29

[16] 王玉林. 多节点容错存储系统的数据与缓存组织研究[D]. 电子科技大学,2010

[17] 杨保华，戴王剑，曹亚仑. Docker技术入门与实战[M].北京:机械工业出版社,2017

[18] 王健宗. 云存储服务质量的若干关键问题研究[D]. 华中科技大学,2012

[19] Holden Karau. Spark快速大数据分析[M].北京:人民邮电出版社,2015

[20] Flavio Junqueira，Benjamin Reed .ZooKeeper：分布式过程协同技术详解[M].北京:机械工业出版社,2008

# 附录

## HDFS配置文件代码

<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

<!--dfs.namenode.name.dir - 这是NameNode结点存储hadoop文件系统信息的本地系统路径。这个值只对NameNode有效，DataNode并不需要使用到它。-->

<name>dfs.namenode.name.dir</name>

<value>file:/app/hadoop/data/dfs/name</value>

</property>

<property>

<!--dfs.datanode.data.dir - 这是DataNode结点被指定要存储数据的本地文件系统路径。DataNode结点上的这个路径没有必要完全相同，因为每台机器的环境很可能是不一样的。但如果每台机器上的这个路径都是统一配置的话，会使工作变得简单一些。默认的情况下，它的值-->

<name>dfs.datanode.data.dir</name>

<value>file:/app/hadoop/data/dfs/data</value>

</property>

<property>

<!--dfs.replication -它决定着系统里面的文件块的数据备份个数。对于一个实际的应用，它应该被设为3（这个数字并没有上限，但更多的备份可能并没有作用，而且会占用更多的空间）。少于三个的备份，可能会影响到数据的可靠性(系统故障时，也许会造成数据丢失)-->

<name>dfs.replication</name>

<value>1</value>

</property>

<property>

<name>dfs.namenode.secondary.http-address</name>

<value>hadoop2:9001</value>

</property>

<property>

<!--namenode的hdfs-site.xml是必须将dfs.webhdfs.enabled属性设置为true，否则就不能使用webhdfs的LISTSTATUS、LISTFILESTATUS等需要列出文件、文件夹状态的命令，因为这些信息都是由namenode来保存的。

访问namenode的hdfs使用50070端口，访问datanode的webhdfs使用50075端口。访问文件、文件夹信息使用namenode的IP和50070端口，访问文件内容或者进行打开、上传、修改、下载等操作使用datanode的IP和50075端口。要想不区分端口，直接使用namenode的IP和端口进行所有的webhdfs操作，就需要在所有的datanode上都设置hefs-site.xml中的dfs.webhdfs.enabled为true。-->

<name>dfs.webhdfs.enabled</name>

<value>true</value>

</property>

<property>

<!--这里设置HDFS客户端最大超时时间，尽量改大，后期hbase经常会因为该问题频繁宕机-->

<name>dfs.client.socket-timeout</name>

<value>600000</value>

</property>

<property>

<!--这里设置Hadoop允许打开最大文件数，默认4096，不设置的话会提示xcievers exceeded错误-->

<name>dfs.datanode.max.transfer.threads</name>

<value>409600</value>

</property>

</configuration>

## core-site.xml代码

<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<!-- 开启垃圾回收站功能,HDFS文件删除后先进入垃圾回收站，垃圾回收站最长保留数据时间为1天，超过一天后就删除 -->

<property>

<name>fs.trash.interval</name>

<value>1440</value>

</property>

<property>

<!--NameNode的URI。格式:【hdfs://主机名/】-->

<name>fs.defaultFS</name>

<value>hdfs://hadoop1:9000</value>

</property>

<property>

<!--hadoop.tmp.dir 是hadoop文件系统依赖的基础配置，很多路径都依赖它。如果hdfs-site.xml中不配置namenode和datanode的存放位置，默认就放在这个路径中-->

<name>hadoop.tmp.dir</name>

<value>file:/app/hadoop/tmp</value>

</property>

<property>

<!--hadoop访问文件的IO操作都需要通过代码库。因此，在很多情况下，io.file.buffer.size都被用来设置SequenceFile中用到的读/写缓存大小。不论是对硬盘或者是网络操作来讲，较大的缓存都可以提供更高的数据传输，但这也就意味着更大的内存消耗和延迟。这个参数要设置为系统页面大小的倍数，以byte为单位，默认值是4KB，一般情况下，可以设置为64KB（65536byte）,这里设置128K-->

<name>io.file.buffer.size</name>

<value>4048</value>

</property>

<property>

<name>dfs.namenode.handler.count</name>

<value>100</value>

<description>The number of server threads for the namenode.</description>

</property>

<property>

<name>dfs.datanode.handler.count</name>

<value>50</value>

<description>The number of server threads for the datanode.</description>

</property>

</configuration>

## hbase-site.xml代码

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<!--hbase存储在HADOOP HDFS上文件根目录路径-->

<property>

<name>hbase.rootdir</name>

<value>hdfs://hadoop1:9000/hbase</value>

</property>

<!--采用分布式模式-->

<property>

<name>hbase.cluster.distributed</name>

<value>true</value>

</property>

<!--zookeeper地址，端口不指定的话就默认为2181-->

<property>

<name>hbase.zookeeper.quorum</name>

<value>hadoop1,hadoop2,hadoop3</value>

</property>

<!--hbase临时文件存储目录，比如一些数据表的预分区信息等等-->

<property>

<name>hbase.tmp.dir</name>

<value>/app/hadoop/hbase/</value>

</property>

<property>

<name>hbase.master</name>

<value>hdfs://hadoop1:60000</value>

</property>

<!--zookeeper存储数据位置-->

<property>

<name>hbase.zookeeper.property.dataDir</name>

<value>/app/hadoop/zookeeper-3.3.6/zookeeper</value>

</property>

<!--这里设置hbase API客户端侧缓存值，大于此值就进行一次提交，/opt/hbase-1.2.1/conf/hbase-site.xml统一配置为5M，对所有HTable都生效，那么客户端API就可不设置

<property-->

<!--htable.setWriteBufferSize(5242880);//5M -->

<property>

<name>hbase.client.write.buffer</name>

<value>5242880</value>

</property>

<!--这里设置Master并发最大线程数-->

<property>

<name>hbase.regionserver.handler.count</name>

<value>300</value>

<description>Count of RPC Listener instances spun up on

RegionServers.Same property is used by the Master for count of master

handlers.</description>

</property>

<property>

<name>hbase.table.sanity.checks</name>

<value>false</value>

</property>

<!--ZooKeeper 会话超时.HBase把这个值传递改zk集群，向他推荐一个会话的最大超时时间-->

<property>

<!--every 30s,the master will check regionser is working -->

<name>zookeeper.session.timeout</name>

<value>30000</value>

</property>

<!--数据表创建时会预分区，每个预分区最大大小这里设置为30G，防止频繁的split阻塞数据读写，只有当预分区超过30G时才会进行split，正式环境应该首先预测数据存储时间内的大致数据量，然后如果每个预分区为30G，计算出分区数，建表时指定分区设置，防止后期频繁split-->

<property>

<!--every region max file size set to 30G -->

<name>hbase.hregion.max.filesize</name>

<value>32212254720</value>

</property>

<!--默认hbase每24小时会进行一次major\_compact，major\_compact会阻塞读写，这里先禁用，但不代表这个操作不做，可以后期指定linux shell加入到cron定时任务在hbase集群空闲情况下执行-->

<property>

<name>hbase.hregion.majorcompaction</name>

<value>0</value>

</property>

<!--hbase本质上可以说是HADOOP HDFS的客户端，虽然Hadoop的core-site.xml里设置了文件副本数，但是仍然是客户端传值优先，这里设置为2，意思是一个文件，最终在Hadoop上总个数为2，正式环境最好设置为3，目前发现此值小于3时，在遇到All datanodes xxx.xxx.xxx.xxx:port are bad. Aborting...错误信息时，如果某个DataNode宕机，原则上hbase调用的DFSClient会去其他Datanode

上重试写，但发现配置的值低于3就不会去尝试-->

<property>

<name>dfs.replication</name>

<value>3</value>

</property>

<!--

IncreasingToUpperBoundRegionSplitPolicy策略的意思是，数据表如果预分区为2，配置的memstore flush size=128M，那么下一次分裂大小是2的平方然后乘以128MB，即2\*2\*128M=512MB;

ConstantSizeRegionSplitPolicy策略的意思是按照上面指定的region大小超过30G才做分裂

-->

<property>

<name>hbase.regionserver.region.split.policy</name>

<value>org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy</value>

</property>

<!--一个edit版本在内存中的cache时长，默认3600000毫秒-->

<property>

<name>hbase.regionserver.optionalcacheflushinterval</name>

<value>7200000</value>

<description>

Maximum amount of time an edit lives in memory before being automatically

flushed.

Default 1 hour. Set it to 0 to disable automatic flushing.</description>

</property>

<!--分配给HFile/StoreFile的block cache占最大堆(-Xmx setting)的比例。默认0.4意思是分配40%，设置为0就是禁用，但不推荐。-->

<property>

<name>hfile.block.cache.size</name>

<value>0.3</value>

<description>Percentage of maximum heap (-Xmx setting) to allocate to

block cache

used by HFile/StoreFile. Default of 0.4 means allocate 40%.

Set to 0 to disable but it's not recommended; you need at least

enough cache to hold the storefile indices.</description>

</property>

<!--当memstore的大小超过这个值的时候，会flush到磁盘。这个值被一个线程每隔hbase.server.thread.wakefrequency检查一下。-->

<property>

<name>hbase.hregion.memstore.flush.size</name>

<value>52428800</value>

</property>

<!--单个region server的全部memtores的最大值。超过这个值，一个新的update操作会被挂起，强制执行flush操作。以前版本中是通过hbase.regionserver.global.memstore.upperLimit设置，老版本中含义是在hbase-env.sh中配置的HEAP\_SIZE比如4G，那么以该值4G乘以配置的0.5就是2G，意思是所有memstore总和达到2G值时，阻塞所有读写，现在1.2.1版本hbase中被hbase.regionserver.global.memstore.size替代，计算方法仍然是HEAP\_SIZE乘以配置的百分比比如下面的0.5，那么阻塞读写的阀值就为2G-->

<property>

<name>hbase.regionserver.global.memstore.size</name>

<value>0.5</value>

</property>

<property>

<name>hbase.regionserver.global.memstore.size.lower.limit</name>

<value>0.5</value>

</property>

<property>

<!--这里设置HDFS客户端最大超时时间，尽量改大，后期hbase经常会因为该问题频繁宕机-->

<name>dfs.client.socket-timeout</name>

<value>600000</value>

</property>

</configuration>

## Zookeeper配置文件

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

dataDir=/app/hadoop/zookeeper-3.3.6/data

# the port at which the clients will connect

clientPort=2181

server.1 =hadoop1:7000:7001

server.2 =hadoop2:7000:7001

server.3 =hadoop3:7000:7001

5.5 Spark配置文件

spark.master spark://hadoop1:7077

spark.eventLog.enabled true

spark.eventLog.dir hdfs://hadoop1:9000/spark/eventLog

spark.history.fs.logDirectory hdfs://hadoop1:9000/spark/historyLog

spark.serializer org.apache.spark.serializer.KryoSerializer

spark.driver.memory 1g

spark.speculation true

spark.task.maxFailures 3

spark.worker.cleanup.enabled true

spark.local.dir /app/hadoop/spark-2.0.2/data

spark.eventLog.compress true

spark.network.timeout 300

spark.driver.extraClassPath /app/hadoop/spark-2.0.2/executorlib/\*

spark.executor.extraClassPath /app/hadoop/spark-2.0.2/executorlib/\*

## Spark的maven工程pom文件

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>scala-java</groupId>

<artifactId>yspark-maven</artifactId>

<version>0.0.1-SNAPSHOT</version>

<inceptionYear>2008</inceptionYear>

<packaging>jar</packaging>

<properties>

<scala.version>2.10.5</scala.version>

<scala.binary.version>2.10</scala.binary.version>

<hadoop.version>2.7.3</hadoop.version>

<hbase.version>1.2.5</hbase.version>

<spark.version>2.0.2</spark.version>

<java.version>1.8</java.version>

<java.encoding>UTF-8</java.encoding>

<spring.version>4.2.5.RELEASE</spring.version>

</properties>

<dependencies>

<!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-hdfs -->

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.7.3</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-core\_2.10</artifactId>

<version>${spark.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-mllib\_2.10</artifactId>

<version>${spark.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-client</artifactId>

<version>${hbase.version}</version>

</dependency>

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-server</artifactId>

<version>${hbase.version}</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>${hadoop.version}</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-mapreduce-client-jobclient</artifactId>

<version>${hadoop.version}</version>

</dependency>

<dependency>

<groupId>org.scala-lang</groupId>

<artifactId>scala-library</artifactId>

<version>${scala.version}</version>

</dependency>

</dependencies>

<build>

<resources>

<!-- 配置文件默认的targetPath就是target/classes -->

<resource>

<directory>src/main/resource</directory>

<includes>

<include>\*\*/\*.properties</include>

</includes>

<filtering>true</filtering>

</resource>

<resource>

<directory>src/main/resource</directory>

<excludes>

<exclude>\*\*/\*.properties</exclude>

</excludes>

<filtering>false</filtering>

</resource>

<!-- 同时把配置文件拷贝到target/resources目录下，供打包用 -->

<resource>

<directory>src/main/resource</directory>

<targetPath>${basedir}/target/resource</targetPath>

<includes>

<include>\*\*/\*.properties</include>

</includes>

<filtering>true</filtering>

</resource>

<resource>

<directory>src/main/resource</directory>

<targetPath>${basedir}/target/resource</targetPath>

<excludes>

<exclude>\*\*/\*.properties</exclude>

</excludes>

<filtering>false</filtering>

</resource>

</resources>

<sourceDirectory>src/main/scala</sourceDirectory>

<testSourceDirectory>src/test/scala</testSourceDirectory>

<plugins>

<plugin>

<groupId>org.scala-tools</groupId>

<artifactId>maven-scala-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>compile</goal>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

<configuration>

<scalaVersion>${scala.version}</scalaVersion>

<!-- <args>

<arg>-target:jvm-1.8</arg>

</args>-->

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-eclipse-plugin</artifactId>

<configuration>

<downloadSources>true</downloadSources>

<buildcommands>

<buildcommand>ch.epfl.lamp.sdt.core.scalabuilder</buildcommand>

</buildcommands>

<additionalProjectnatures>

<projectnature>ch.epfl.lamp.sdt.core.scalanature</projectnature>

</additionalProjectnatures>

<classpathContainers>

<classpathContainer>org.eclipse.jdt.launching.JRE\_CONTAINER</classpathContainer>

<classpathContainer>ch.epfl.lamp.sdt.launching.SCALA\_CONTAINER</classpathContainer>

</classpathContainers>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>${java.version}</source>

<target>${java.version}</target>

<encoding>${java.encoding}</encoding>

</configuration>

</plugin>

</plugins>

</build>

<profiles>

<profile>

<id>dev</id>

<properties>

<metadata.broker.list>hadoop1:9092,hadoop2:9092,hadoop3:9092</metadata.broker.list>

<exclude.partner>partnerId</exclude.partner>

<p.service.zookeeper.address>192.168.1.212:2181</p.service.zookeeper.address>

<p.alluxio.path>alluxio://hadoop1:19998/</p.alluxio.path>

<mq.host>hadoop1</mq.host>

<mq.port>5672</mq.port>

<mq.user>crawler</mq.user>

<mq.pwd>crawler</mq.pwd>

<p.connection.url>jdbc:mysql://192.168.1.241:3306/risk?useUnicode=true&amp;characterEncoding=utf8</p.connection.url>

<p.connection.username>bqs</p.connection.username>

<p.connection.password>L09OewqYS+9y2oOX0033fyVIFSbOUZUNF8xdmjNyINWhIa7Qfn4XgE+9cZpKUcyDuOUz50IEVkkvrBN5vZznTQ==</p.connection.password>

<p.druid.publickey>MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAI22nXiHLgpfcQdOc0Mbb8t70pZdcCEUR11tJ/ZqH+yX9DMNwEkXAmHI15LgcQD7JpmF1YzL6LLqUx1W9L7jNqsCAwEAAQ==</p.druid.publickey>

<p.ftp.server.host>192.168.1.212</p.ftp.server.host>

<p.ftp.server.username>bmsdvp</p.ftp.server.username>

<p.ftp.server.passwd>123456</p.ftp.server.passwd>

<p.hbase.zookeeper.quorum>hadoop1,hadoop2,hadoop3</p.hbase.zookeeper.quorum>

<p.hbase.zookeeper.property.clientPort>2181</p.hbase.zookeeper.property.clientPort>

</properties>

</profile>

</profiles>

</project>

## CountJob.scala代码

package com.bqs.risk.job

import com.bqs.risk.base.GenericHBaseSparkJob

import com.bqs.risk.constant.HBaseTables

import com.bqs.risk.utils.{CommonUtils, HBaseUtils}

import org.apache.hadoop.hbase.client.Result

import org.apache.hadoop.hbase.io.ImmutableBytesWritable

import org.apache.spark.rdd.RDD

import org.apache.spark.storage.StorageLevel

/\*\*

\* Created by Administrator on 2017/5/8 0008.

\*/

object CountJob extends GenericHBaseSparkJob {

val tableName = "maimai"

val columns =

HBaseTables.maimai.city ::

HBaseTables.maimai.province::

HBaseTables.maimai.position::

HBaseTables.maimai.major::

Nil

override def operations(): Unit = {

val rdd = ReadHBaseTableAsRDD(tableName, columns)

rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_2)

columns.foreach(x=>{

count(rdd,x)

})

}

/\*\*

\* 由于统计的代码逻辑都是相似的，所以完全可以通过传参来控制

\* @param rdd

\* @param column

\*/

def count(rdd: RDD[(ImmutableBytesWritable, Result)], column: String): Unit = {

println(column)

val reRDD = rdd.filter(x => {

val re = HBaseUtils.getString(x.\_2, column)

CommonUtils.isNotBlankString(re)

}).map(x=>{

val re=HBaseUtils.getString(x.\_2,column)

(re,1)

}).reduceByKey(\_+\_)

reRDD.take(Int.MaxValue).foreach(x=>{

println(column+"\t"+x.toString())

})

}

## HTML前端代码

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>ECharts</title>

<!-- 引入 echarts.js -->

<script src="echarts.min.js"></script>

<script src="echarts-wordcloud.min.js"></script>

<script src="china.js" charset ="utf-8"></script>

<script src="cityData.js"></script>

<script src="cityHot.js"></script>

<script src="province.js"></script>

<script src="company.js"></script>

<script src="topic.js"></script>

<script src="point.js"></script>

<script src="name.js"></script>

<script src="position.js"></script>

<script src="manager.js"></script>

</head>

<body>

<!-- 为ECharts准备一个具备大小（宽高）的Dom -->

<div id="data">

<div id="city" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="cityHot" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="province" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="company" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="topic" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="point" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="name" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="position" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

<div id="manager" style="width: 80%;height:500px;margin:0 auto;margin-top:20px;"></div>

</div>

<script type="text/javascript">

var cityChart=echarts.init(document.getElementById('city'));

cityChart.setOption(cityOption);

var cityHotChart=echarts.init(document.getElementById('cityHot'));

cityHotChart.setOption(cityHotOption);

var provinceChart=echarts.init(document.getElementById('province'));

provinceChart.setOption(provinceOption);

var companyChart=echarts.init(document.getElementById('company'));

companyChart.setOption(companyOption);

var topicChart=echarts.init(document.getElementById('topic'));

topicChart.setOption(topicOption);

var topicChart=echarts.init(document.getElementById('point'));

topicChart.setOption(pointOption);

var nameChart=echarts.init(document.getElementById('name'));

nameChart.setOption(nameOption);

var positionChart=echarts.init(document.getElementById('position'));

positionChart.setOption(positionOption);

var managerChart=echarts.init(document.getElementById('manager'));

managerChart.setOption(managerOption);

</script>

</body>

</html>

# 谢辞

毕业设计一开始是准备做一个简单的网站，但在3月份的时候，感觉自己工作这么久，毕设是检验自己真正掌握多少大数据东西的时候，需要认真的对待。当我将换题目的想法告诉陈老师之后，他表示非常支持我的做法，陈老师也希望我们能做出体现自己特色的毕业设计。开始做大数据项目的时候，由于工作的原因实际的有效时间显得很少，这使得我的初稿比较粗糙，在交给指导老师之后，老师很耐心的帮我订正错误，修改论文，直至达到标准。这期间陈老师的专业和孜孜不转给我留下了很深的印象，让我受益匪浅。当然还有其他老师的帮助，无论是平时的言行还是必要时候的帮助都让我获得了很大的成长。在这里也衷心地道一句老师，您们辛苦了！