# Hadoop 大数据实战手册


孙老师

# Hadoop 大数据实战手册

## 目录


Hadoop 入门与实践 1

第一章 前言 2

第二章 hadoop 简介 3

1. Hadoop 版本衍化历史 3
2. Hadoop 生态圈 4

第三章 安装 hadoop 环境 6

第四章 HDFS 文件系统 13

1. HDFS 特点： 13
2. 不适用于 HDFS 的场景： 14
3. HDFS 体系架构 15
4. HDFS 数据块复制 16
5. HDFS 读取和写入流程 17
6. 操作 HDFS 的基本命令 19

第五章 Mapreduce 计算框架 21

1. MapReduce 编程模型 21
2. MapReduce 执行流程 23
3. MapReduce 数据本地化（Data-Local） 26
4. MapReduce 工作原理 27
5. MapReduce 错误处理机制 30

第六章 Zookeeper 32

1. Zookeeper 数据模型 33
2. Zookeeper 访问控制 35
3. Zookeeper 应用场景 36

第七章 HBase 37

1. Hbase 简介 38

2. Hbase 数据模型.....39
3. Hbase 架构及基本组件.....41
4. Hbase 容错与恢复.....43
5. Hbase 基础操作.....45

第八章 Hive.....48

1. Hive 基础原理.....49
2. Hive 基础操作.....52

第九章 流式计算解决方案-Storm.....60

1. Storm 特点.....60
2. Storm 与 Hadoop 区别.....61
3. Storm 基本概念.....63
4. Storm 系统架构.....69
5. Storm 容错机制.....71
6. 一个简单的 Storm 实现.....73
7. Storm 常用配置.....74

第十章 数据挖掘——推荐系统.....75

1. 数据挖掘和机器学习概念.....75
2. 一个机器学习应用方向——推荐领域.....76
3. 推荐算法——基于内容的推荐方法.....77
4. 推荐算法——基于协同过滤的推荐方法.....80

# 第一章 前言

出此书的目的就是为了帮助新人快速进入大数据行业，市面上有很多类似的书籍都是重理论少实践，特别缺少一线企业实践经验的传授，而这个课程会让您少走弯路、快速入门和实践，让您再最短时间内达到一个一线企业大数据工程师的能力标准，因为在课程整理和实践安排上过滤掉很多用不上的知识，直接带领大家以最直接的方式掌握大数据使用方法。

我在知名一线互联网公司从事大数据开发与管理多年，深知业界大数据公司一直对大数据人才的渴望，同时也知道有很多的大数据爱好者想参与进这个朝阳行业，因为平时也是需要参与大数据工程师的招聘与培养的，所以特别想通过一种方式，让广大的大数据爱好者更好的与企业对接，让优秀的人才找到合适的企业，《Hadoop 大数据实战手册》是我根据多年从业经验整理的系列课程，希望让更多的大数据爱好者收益！

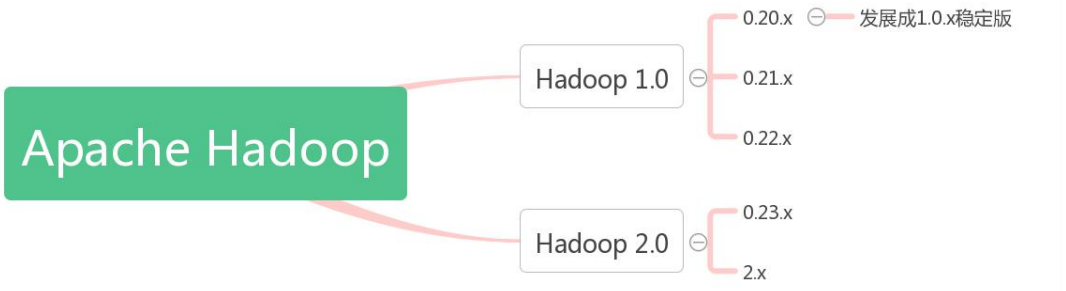
# 第二章 hadoop 简介

Hadoop 是一个由 Apache 基金会所开发的开源分布式系统基础架构。用户可以在不了解分布式底层细节的情况下，开发分布式程序，充分利用集群的威力进行高速运算和存储。解决了大数据（大到一台计算机无法进行存储，一台计算机无法在要求的时间内进行处理）的可靠存储和处理。适合处理非结构化数据，包括 HDFS，MapReduce 基本组件。

## 1. Hadoop 版本衍化历史

由于 Hadoop 版本混乱多变对初级用户造成一定困扰，所以对其版本衍化历史有个大概了解，有助于在实践过程中选择合适的 Hadoop 版本。

**Apache Hadoop 版本分为分为 1.0 和 2.0 两代版本**，我们将第一代 Hadoop 称为 Hadoop 1.0，第二代 Hadoop 称为 Hadoop 2.0。下图是 Apache Hadoop 的版本衍化史：




**第一代 Hadoop 包含三个大版本，分别是 0.20.x，0.21.x 和 0.22.x**，其中，0.20.x 最后演化成 1.0.x，变成了稳定版。

**第二代 Hadoop 包含两个版本，分别是 0.23.x 和 2.x**，它们完全不同于 Hadoop 1.0，是一套全新的架构，均包含 HDFS Federation 和 YARN 两个系统，相比于 0.23.x，2.x 增加了

NameNode HA 和 Wire-compatibility 两个重大特性。


Hadoop 遵从 Apache 开源协议，用户可以免费地任意使用和修改 Hadoop，也正因此，市面上出现了很多 Hadoop 版本，其中比较出名的一是 Cloudera 公司的发行版，该版本称为 CDH（Cloudera Distribution Hadoop）。

截至目前为止，CDH 共有 4 个版本，其中，前两个已经不再更新，最近的两个，分别是 CDH3（在 Apache Hadoop 0.20.2 版本基础上演化而来的）和 CDH4 在 Apache Hadoop 2.0.0 版本基础上演化而来的），分别对应 Apache 的 Hadoop 1.0 和 Hadoop 2.0。

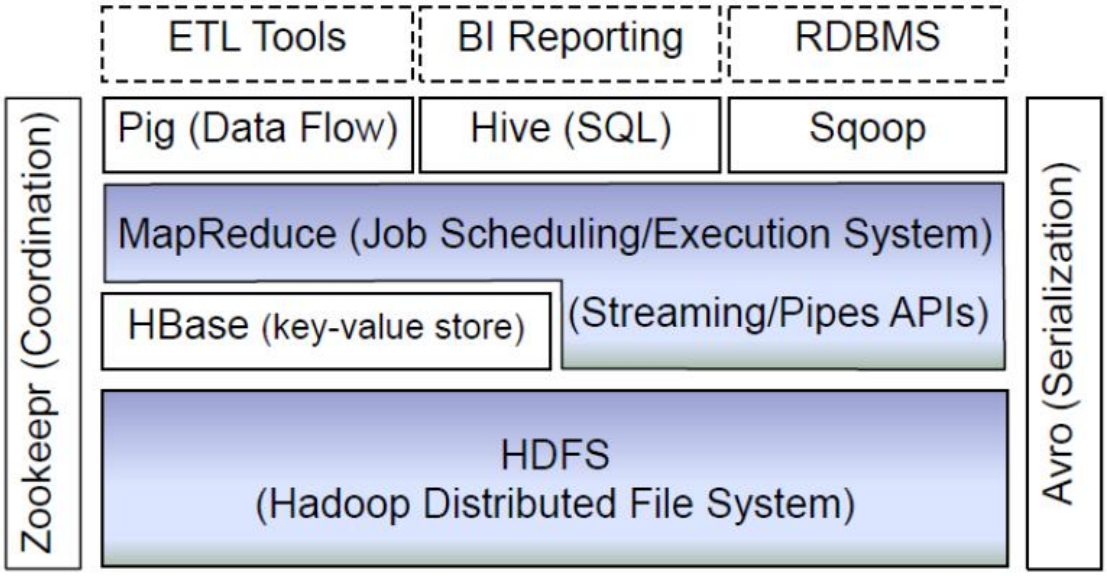
## 2. Hadoop 生态圈

架构师和开发人员通常会使用一种软件工具，用于其特定的用途软件开发。例如，他们可能会说，Tomcat 是 Apache Web 服务器，MySQL 是一个数据库工具。

然而，当提到 Hadoop 的时候，事情变得有点复杂。Hadoop 包括大量的工具，用来协同工作。因此，Hadoop 可用于完成许多事情，以至于，人们常常根据他们使用的方式来定义它。

对于一些人来说，Hadoop 是一个数据管理系统。他们认为 Hadoop 是数据分析的核心，汇集了结构化和非结构化的数据，这些数据分布在传统的企业数据栈的每一层。对于其他人，Hadoop 是一个大规模并行处理框架，拥有超级计算能力，定位于推动企业级应用的执行。还有一些人认为 Hadoop 作为一个开源社区，主要为解决大数据的问题提供工具和软件。因为 Hadoop 可以用来解决很多问题，所以很多人认为 Hadoop 是一个基本框架。

虽然 Hadoop 提供了这么多的功能，但是仍然应该把它归类为多个组件组成的 Hadoop 生态圈，这些组件包括数据存储、数据集成、数据处理和其它进行数据分析的专门工具。




该图主要列举了生态圈内部主要的一些组件，从底部开始进行介绍：

1) HDFS：Hadoop 生态圈的基本组成部分是 Hadoop 分布式文件系统（HDFS）。HDFS 是一种数据分布式保存机制，数据被保存在计算机集群上。数据写入一次，读取多次。HDFS 为 HBase 等工具提供了基础。

2) MapReduce：Hadoop 的主要执行框架是 MapReduce，它是一个分布式、并行处理的编程模型。MapReduce 把任务分为 map(映射)阶段和 reduce(化简)。开发人员使用存储在 HDFS 中数据（可实现快速存储），编写 Hadoop 的 MapReduce 任务。由于 MapReduce 工作原理的特性， Hadoop 能以并行的方式访问数据，从而实现快速访问数据。

3) Hbase：HBase 是一个建立在 HDFS 之上，面向列的 NoSQL 数据库，用于快速读/写大量数据。HBase 使用 Zookeeper 进行管理，确保所有组件都正常运行。

4) ZooKeeper：用于 Hadoop 的分布式协调服务。Hadoop 的许多组件依赖于 Zookeeper，它运行在计算机集群上面，用于管理 Hadoop 操作。

5) Hive：Hive 类似于 SQL 高级语言，用于运行存储在 Hadoop 上的查询语句，Hive 让不熟悉 MapReduce 开发人员也能编写数据查询语句，然后这些语句被翻译为 Hadoop 上面的 MapReduce 任务。像 Pig 一样，Hive 作为一个抽象层工具，吸引了很多熟悉 SQL 而不是 Java 编程的数据分析师。

6) Pig：它是 MapReduce 编程的复杂性的抽象。Pig 平台包括运行环境和用于分析 Hadoop 数据集的脚本语言(Pig Latin)。其编译器将 Pig Latin 翻译成 MapReduce 程序序列。

7) Sqoop：是一个连接工具，用于在关系数据库、数据仓库和 Hadoop 之间转移数据。Sqoop 利用数据库技术描述架构，进行数据的导入/导出；利用 MapReduce 实现并行化运行和

容错技术。

# 第三章 安装 hadoop 环境

由于实践部分主要以 Hadoop 1.0 环境为主，所以这主要介绍如何搭建 Hadoop 1.0 分布式环境。

整个分布式环境运行在带有 linux 操作系统的虚拟机上，至于虚拟机和 linux 系统的安装这里暂不做过多介绍。

安装 Hadoop 分布式环境：

1) 下载 Hadoop 安装包：

在 http://pan.baidu.com/s/1qXSN3hM 地址中可以找到 hadoop-1.2.1-bin.tar.gz 文件

使用 securtCRT 的 rz 功能上传 hadoop-1.2.1-bin.tar.gz 这个文件到虚拟机的系统中。

同样在 securtcrt 中 ll 时，能得到

```
dr-x------  2 hadoop hadoop        0 Feb 19 05:28 .gvfs/
-rw-r--r--  1 hadoop hadoop 38096663 Feb 19 06:05 hadoop-1.2.1-bin.tar.gz
-rw-------  1 hadoop hadoop     3258 Feb 19 05:28 .ICEauthority
drwxr-xr-x  8 hadoop hadoop     4096 Mar 27  2013 jdk1.6.0_45/
```

2) 安装 Hadoop 安装包：

- 首先将安装包解压缩：

```
hadoop@hadoop-VirtualBox:~$ tar zxvf ./hadoop-1.2.1-bin.tar.gz
```

- Linux 终端执行 cd 进入相应目录：

```
hadoop@hadoop-VirtualBox:~$ cd hadoop-1.2.1/
```

- 新增 tmp 目录，mkdir /home/hadoop/hadoop-1.2.1/tmp

```
hadoop@hadoop-VirtualBox:~/hadoop-1.2.1$ mkdir /home/hadoop/hadoop-1.2.1/tmp
```

3) 配置 Hadoop：

- 使用 vim 修改 master 文件内容：

```
hadoop@hadoop-VirtualBox:~/hadoop-1.2.1/conf$ vi ./masters
```

将 localhost 修改成 master：

```
master
~
```

最后保存退出。

- 修改 slaves 文件

注意，这里准备设置几台 slave 机器，就写几个，因为当前分布式环境有四个虚拟机，一台做 master，三台做 slave，所以这里写成了三个 slave

```
slave1
slave2
slave3
~
~
```

- 修改 core-site.xml 文件：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

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

<configuration>
 <property>
        <name>hadoop.tmp.dir</name>
        <value>/home/hadoop/hadoop-1.2.1/tmp</value>
    </property>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://192.168.2.55:9000</value>
    </property>
</configuration>
~
```

【注意】中间的 ip 地址，不要输入 192.168.2.55，根据自己的情况设置。

- 修改 mapred-site.xml 文件：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

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

<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>http://192.168.2.55:9001</value>
    </property>
</configuration>
~
~
```

【注意】记得 value 的内容要以 http 开头。

- 修改 hdfs-site.xml 文件：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

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

<configuration>
        <property>
        <name>dfs.replication</name>
        <value>3</value>
        </property>
</configuration>
~
```

其中，<value>3</value>视情况修改，如果有三台 slave 机器，这里设置成 3，如果只有 1 台或 2 台，修改成对应的值即可。

- 修改 hadoo-env.sh 文件

在`# export JAVA_HOME=/usr/lib/j2sdk1.5-sun`

下新增 export JAVA_HOME=/home/hadoop/jdk1.6.0_45/

```
# The java implementation to use.  Required.
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
export JAVA_HOME=/home/hadoop/jdk1.6.0_45/
```

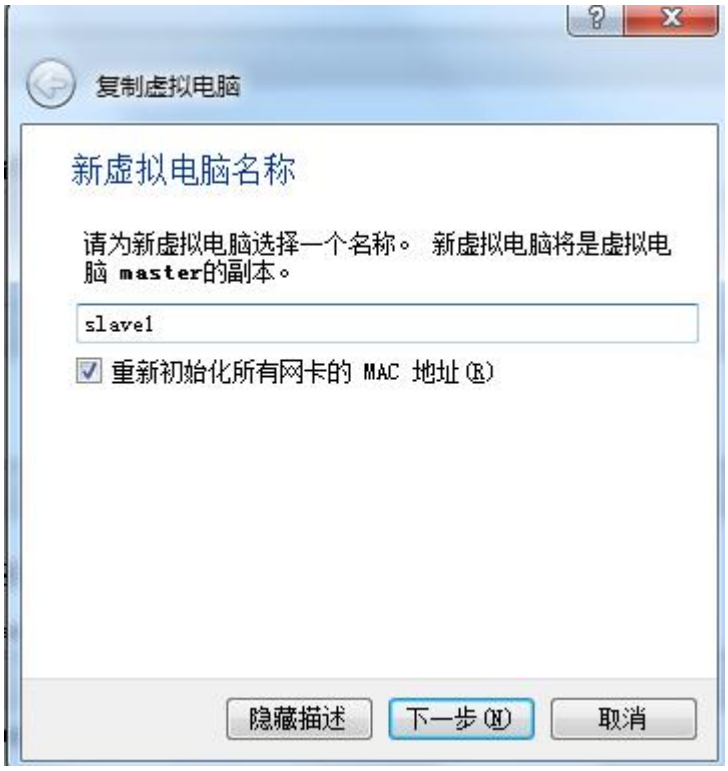
- 修改本地网络配置：编辑/etc/hosts 文件

```
hadoop@hadoop-VirtualBox:~$ sudo vi /etc/hosts
127.0.0.1       localhost
127.0.1.1       hadoop-VirtualBox

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

192.168.2.55 master
192.168.2.56 slave1
192.168.2.57 slave2
192.168.2.58 slave3
~
~
```

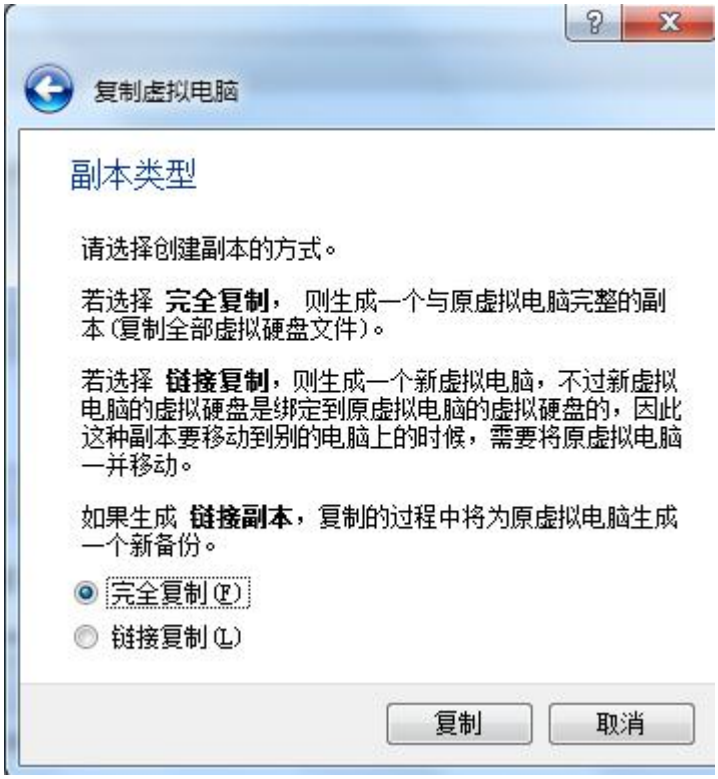
【注意】Ip 地址根据具体的情况要进行修改。

4) 复制虚拟机

- 关闭当前虚拟机，并复制多份



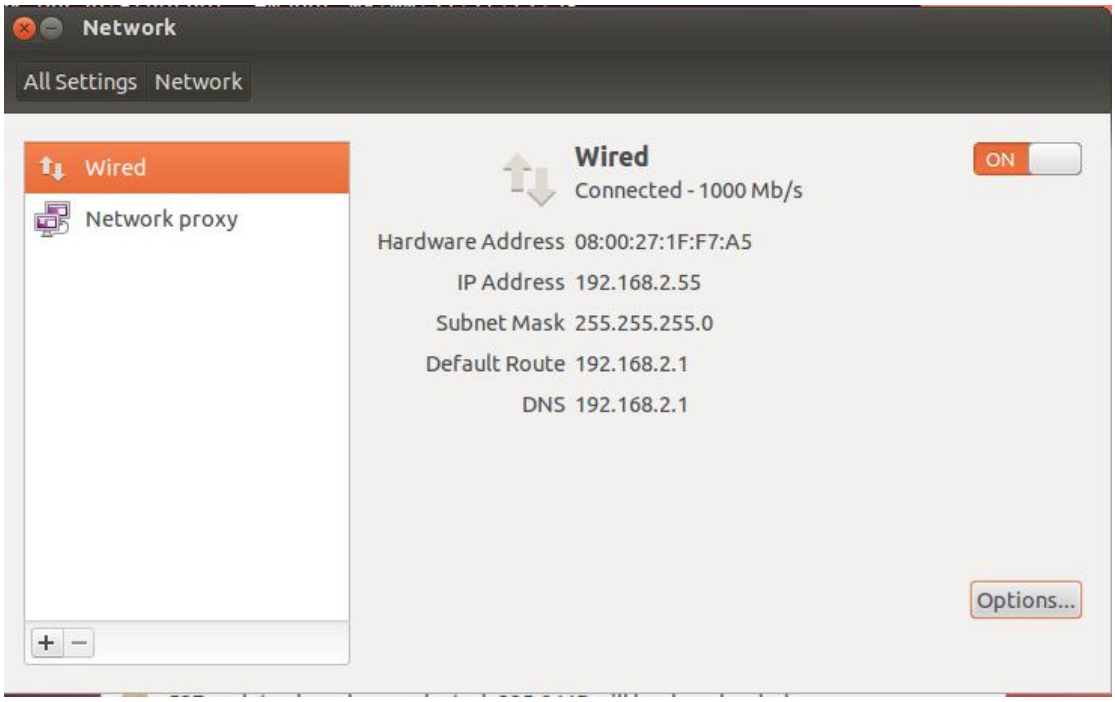

【注意】要选择初始化所有网卡的 mac 地址




根据自己需求，复制 2 到 3 台虚拟机作为 slave，同样要确认网络连接方式为桥接。

- 设置所有机器的 IP 地址

分别启动虚拟机，修改机器的 ip 地址，在虚拟机的图形界面里，选择设置  单击打开，在弹出来的窗口里，选择 


打开 

，修改成如下的形式，选择 ipv4 ,分配方式选择成 manual。

【注意】具体的 ip 地址，根据实际的情况来设置，因为培训教室里都是 192.168.2.x 的网段，所以我这里设置成了 192.168.2.x，每个人选择自己的一个 ip 地址范围，注意不要和其它人冲突了。


5) 建立互信关系

- 生成公私钥，在 master 机器的虚拟机命令行下输入 ssh-keygen，一路回车，全默认

```
hadoop@master:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh/id_rsa):
Created directory '/home/hadoop/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hadoop/.ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/id_rsa.pub.
The key fingerprint is:
51:cc:20:47:4a:b6:ef:f6:2b:81:78:5a:26:9a:ec:a3 hadoop@master
The key's randomart image is:
+--[ RSA 2048]----+
|       +.++.     |
|      o = .o     |
|       o .       |
|        . .      |
|       . .S      |
|      o =..      |
|   . o *  o.     |
|    = .  ...     |
|Eo..      .o.    |
+-----------------+
hadoop@master:~$
```

- 复制公钥

复制一份 master 的公钥文件，cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

```
hadoop@master:~/.ssh$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

同样，在所有的 slave 机器上，也在命令行中输入 ssh-keygen，一路回车，全默认

在所有的 salve 机器上，从 master 机器上复制 master 的公钥文件：

```
hadoop@hadoop-VirtualBox:~$ scp master:~/.ssh/authorized_keys /home/hadoop/.ssh/
hadoop@master's password:
authorized_keys
hadoop@hadoop-VirtualBox:~$
```

- 测试连接

在 master 机器上分别向所有的 slave 机器发起联接请求：

如：ssh slave1

```
hadoop@master:~/.ssh$ ssh slave1
The authenticity of host 'slave1 (192.168.2.56)' can't be established.
ECDSA key fingerprint is 0b:c6:fc:6a:80:b8:be:0b:6f:0e:21:ae:19:7a:ae:88.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'slave1' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-29-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/

537 packages can be updated.
217 updates are security updates.

Last login: Thu Feb 20 06:59:20 2014 from master
hadoop@hadoop-VirtualBox:~$ exit
logout
Connection to slave1 closed.
```

【注意】记得一旦联接上，所有的操作，就视同在对应的 slave 上操作，所以一定要记得使用 exit 退出联接。

6) 启动 Hadoop：

- 初始化：在 master 机器上，进入/home/hadoop/hadoop-1.2.1/bin 目录

```
hadoop@master:~/hadoop-1.2.1/bin$ cd /home/hadoop/hadoop-1.2.1/bin
```

在安装包根目录下运行./hadoop namenode –format 来初始化 hadoop 的文件系统。

```
hadoop@master:~/hadoop-1.2.1/bin$ ./hadoop  namenode -format
14/02/20 07:06:11 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = master/192.168.2.55
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 1.2.1
STARTUP_MSG:   build = https://svn.apache.org/repos/asf/hadoop/common/branches/branch-1.2 -r 1503152; compiled by 'mattf' on Mon Jul 22 15:23:09 PDT 2013
STARTUP_MSG:   java = 1.6.0_45
************************************************************/
Re-format filesystem in /home/hadoop/hadoop-1.2.1/tmp/dfs/name ? (Y or N) y
Format aborted in /home/hadoop/hadoop-1.2.1/tmp/dfs/name
14/02/20 07:06:14 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master/192.168.2.55
************************************************************/
hadoop@master:~/hadoop-1.2.1/bin$
```

- 启动

执行./start-all.sh，如果中间过程提示要判断是否，需要输入 yes

```
hadoop@master:~/hadoop-1.2.1/bin$ ./start-all.sh
starting namenode, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-namenode-master.out
slave3: starting datanode, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-datanode-hadoop-VirtualBox.out
slave1: starting datanode, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-datanode-hadoop-VirtualBox.out
slave2: starting datanode, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-datanode-hadoop-VirtualBox.out
master: starting secondarynamenode, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-secondarynamenode-master.out
starting jobtracker, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-jobtracker-master.out
slave1: starting tasktracker, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-tasktracker-hadoop-VirtualBox.out
slave2: starting tasktracker, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-tasktracker-hadoop-VirtualBox.out
slave3: starting tasktracker, logging to /home/hadoop/hadoop-1.2.1/libexec/../logs/hadoop-hadoop-tasktracker-hadoop-VirtualBox.out
hadoop@master:~/hadoop-1.2.1/bin$
```

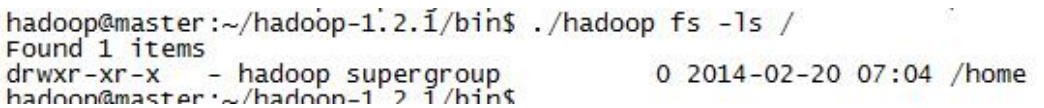
输入 jps，查看进程是否都正常启动。

```
hadoop@master:~/hadoop-1.2.1/bin$ jps
5409 SecondaryNameNode
5630 Jps
5152 NameNode
5489 JobTracker
hadoop@master:~/hadoop-1.2.1/bin$
```

如果一切正常，应当有如上的一些进程存在。

7) 测试系统

输入./hadoop fs –ls /




能正常显示文件系统。

如此，hadoop 系统搭建完成。否则，可以去/home/hadoop/hadoop-1.2.1/logs 目录下，查看缺少的进程中，对应的出错日志。

# 第四章 HDFS 文件系统

Hadoop 附带了一个名为 HDFS(Hadoop 分布式文件系统)的分布式文件系统，专门存储超大数据文件，为整个 Hadoop 生态圈提供了基础的存储服务。

本章内容：

1) HDFS 文件系统的特点，以及不适用的场景

2) HDFS 文件系统重点知识点：体系架构和数据读写流程

3) 关于操作 HDFS 文件系统的一些基本用户命令

## 1. HDFS 特点：

HDFS 专为解决大数据存储问题而产生的，其具备了以下特点：

1) HDFS 文件系统可存储超大文件

每个磁盘都有默认的数据块大小，这是磁盘在对数据进行读和写时要求的最小单位，

文件系统是要构建于磁盘上的，文件系统的也有块的逻辑概念，通常是磁盘块的整数倍，通常文件系统为几千个字节，而磁盘块一般为 512 个字节。

HDFS 是一种文件系统，自身也有块（block）的概念，其文件块要比普通单一磁盘上文件系统大的多，默认是 64MB。

HDFS 上的块之所以设计的如此之大，其目的是为了最小化寻址开销。

HDFS 文件的大小可以大于网络中任意一个磁盘的容量，文件的所有块并不需要存储在一个磁盘上，因此可以利用集群上任意一个磁盘进行存储，由于具备这种分布式存储的逻辑，所以可以存储超大的文件，通常 G、T、P 级别。

2) 一次写入，多次读取

一个文件经过创建、写入和关闭之后就不需要改变，这个假设简化了数据一致性的问题，同时提高数据访问的吞吐量。

3) 运行在普通廉价的机器上

Hadoop 的设计对硬件要求低，无需昂贵的高可用性机器上，因为在 HDFS 设计中充分考虑到了数据的可靠性、安全性和高可用性。

## 2. 不适用于 HDFS 的场景：

1) 低延迟

HDFS 不适用于实时查询这种对延迟要求高的场景，例如：股票实盘。往往应对低延迟数据访问场景需要通过数据库访问索引的方案来解决，Hadoop 生态圈中的 Hbase 具有这种随机读、低延迟等特点。

2) 大量小文件

对于 Hadoop 系统，小文件通常定义为远小于 HDFS 的 block size（默认 64MB）

的文件，由于每个文件都会产生各自的 MetaData 元数据，Hadoop 通过 Namenode 来存储这些信息，若小文件过多，容易导致 Namenode 存储出现瓶颈。

3) 多用户更新

为了保证并发性，HDFS 需要一次写入多次读取，目前不支持多用户写入，若要修改，也是通过追加的方式添加到文件的末尾处，出现太多文件需要更新的情况，Hadoop 是不支持的。

针对有多人写入数据的场景，可以考虑采用 Hbase 的方案。

4) 结构化数据

HDFS 适合存储半结构化和非结构化数据，若有严格的结构化数据存储场景，也可以考虑采用 Hbase 的方案。

5) 数据量并不大

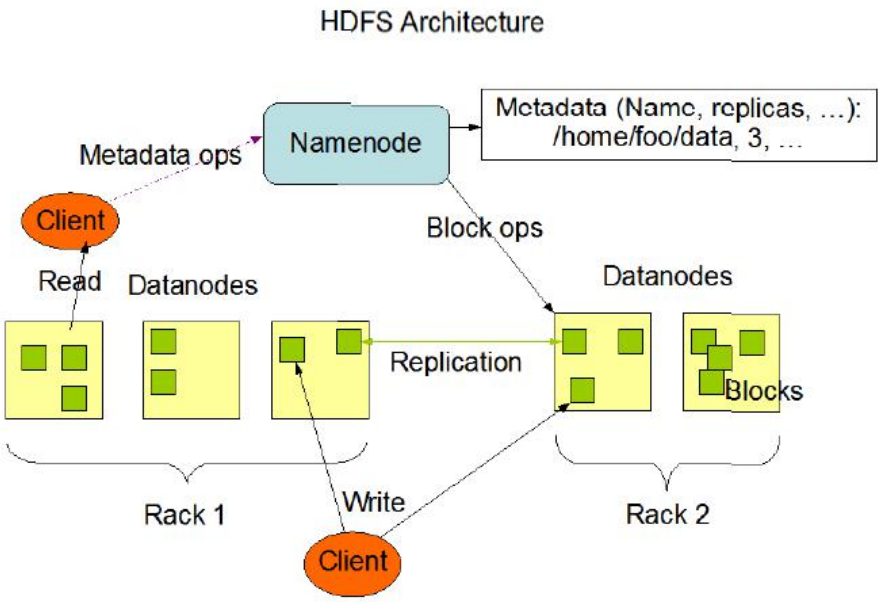
通常 Hadoop 适用于 TB、PB 数据，若待处理的数据只有几十 GB 的话，不建议使用 Hadoop，因为没有任何好处。

## 3. HDFS 体系架构

HDFS 是一个主/从（Master/Slave）体系架构，由于分布式存储的性质，集群拥有两类节点 NameNode 和 DataNode。

NameNode（名字节点）：系统中通常只有一个，中心服务器的角色，管理存储和检索多个 DataNode 的实际数据所需的所有元数据。

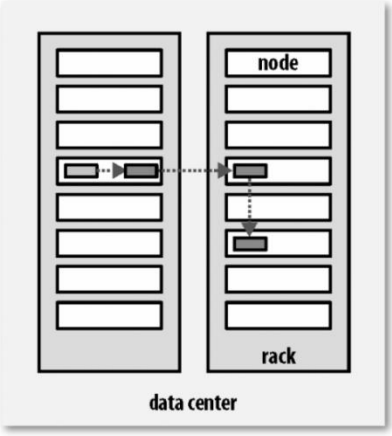
DataNode（数据节点）：系统中通常有多个，是文件系统中真正存储数据的地方，在 NameNode 统一调度下进行数据块的创建、删除和复制。




图中的 Client 是 HDFS 的客户端，是应用程序可通过该模块与 NameNode 和 DataNode 进行交互，进行文件的读写操作。

## 4. HDFS 数据块复制

为了系统容错，文件系统会对所有数据块进行副本复制多份，Hadoop 是默认 3 副本管理。



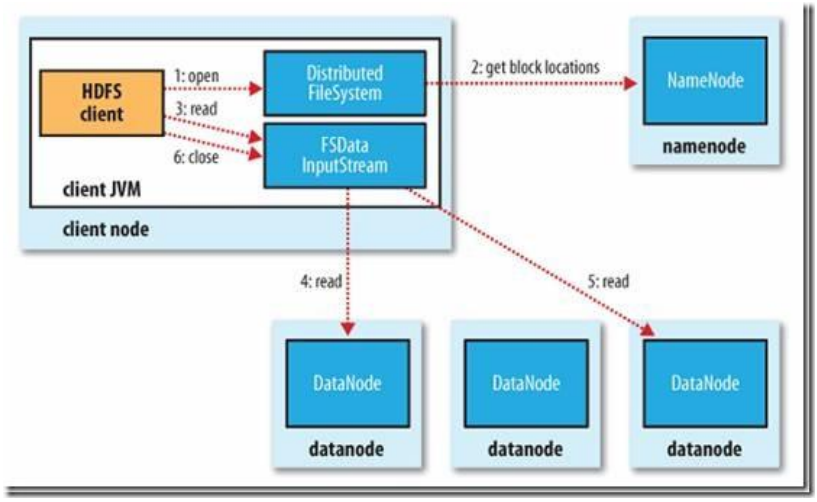

复本管理策略是运行客户端的节点上放一个复本（若客户端运行在集群之外，会随机选择一个节点），第二个复本会放在与第一个不同且随机另外选择的机架中节点上，第三个复本与第二个复本放在相同机架，切随机选择另一个节点。所存在其他复本，则放在集群中随

机选择的节点上，不过系统会尽量避免在相同机架上放太多复本。

所有有关块复制的决策统一由 NameNode 负责，NameNode 会周期性地接受集群中数据节点 DataNode 的心跳和块报告。一个心跳的到达表示这个数据节点是正常的。一个块报告包括该数据节点上所有块的列表。

## 5. HDFS 读取和写入流程

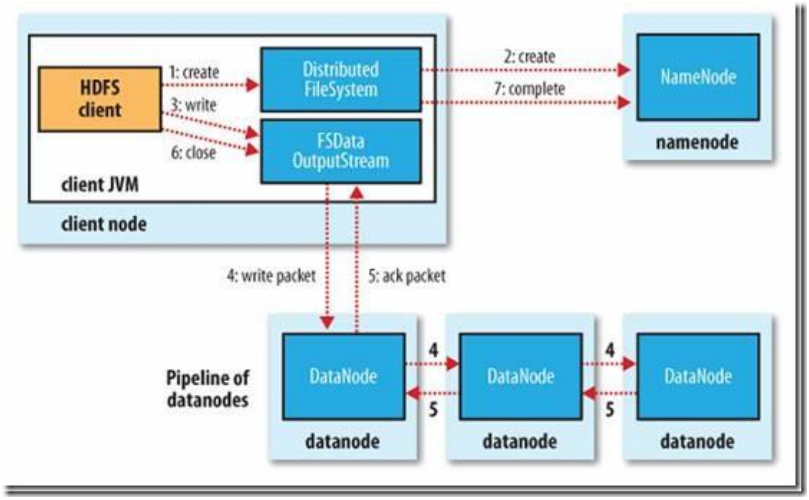
1) 读文件的过程：




首先 Client 通过 File System 的 Open 函数打开文件，Distributed File System 用 RPC 调用 NameNode 节点，得到文件的数据块信息。对于每一个数据块，NameNode 节点返回保存数据块的数据节点的地址。Distributed File System 返回 FSDataInputStream 给客户端，用来读取数据。客户端调用 stream 的 read()函数开始读取数据。DFSInputStream 连接保存此文件第一个数据块的最近的数据节点。DataNode 从数据节点读到客户端(client)，当此数据块读取完毕时，DFSInputStream 关闭和此数据节点的连接，然后连接此文件下一个数据块的最近的数据节点。当客户端读取完毕数据的时候，调用 FSDataInputStream 的 close 函数。

在读取数据的过程中，如果客户端在与数据节点通信出现错误，则尝试连接包含此数据块的下一个数据节点。失败的数据节点将被记录，以后不再连接。

2) 写文件的过程：




客户端调用 create()来创建文件，Distributed File System 用 RPC 调用 NameNode 节点，在文件系统的命名空间中创建一个新的文件。NameNode 节点首先确定文件原来不存在，并且客户端有创建文件的权限，然后创建新文件。

Distributed File System 返回 DFSOutputStream，客户端用于写数据。客户端开始写入数据，DFSOutputStream 将数据分成块，写入 Data Queue。Data Queue 由 Data Streamer 读取，并通知 NameNode 节点分配数据节点，用来存储数据块(每块默认复制 3 块)。分配的数据节点放在一个 Pipeline 里。Data Streamer 将数据块写入 Pipeline 中的第一个数据节点。第一个数据节点将数据块发送给第二个数据节点。第二个数据节点将数据发送给第三个数据节点。

DFSOutputStream 为发出去的数据块保存了 Ack Queue，等待 Pipeline 中的数据节点告知数据已经写入成功。

# 6. 操作 HDFS 的基本命令

1) 打印文件列表（ls）

```
标准写法：
    hadoop fs -ls hdfs:/      #hdfs: 明确说明是 HDFS 系统路径

简写：
    hadoop fs -ls /                  #默认是 HDFS 系统下的根目录

打印指定子目录：
    hadoop fs -ls /package/test/ #HDFS 系统下某个目录
```

2) 上传文件、目录（put、copyFromLocal）

put 用法：

```
上传新文件：
    hdfs fs -put file:/root/test.txt hdfs:/        #上传本地 test.txt 文件到 HDFS 根目录 ,HDFS
    根目录须无同名文件，否则“File exists”
    hdfs fs -put test.txt /test2.txt    #上传并重命名文件。
    hdfs fs -put test1.txt test2.txt hdfs:/        #一次上传多个文件到 HDFS 路径。

上传文件夹：
    hdfs fs -put mypkg /newpkg          #上传并重命名了文件夹。

覆盖上传：
    hdfs fs -put -f /root/test.txt /    #如果 HDFS 目录中有同名文件会被覆盖
```

copyFromLocal 用法：

```
上传文件并重命名：
    hadoop fs -copyFromLocal file:/test.txt hdfs:/test2.txt

覆盖上传：
    hadoop fs -copyFromLocal -f test.txt /test.txt
```

3) 下载文件、目录（get、copyToLocal）

get 用法：

```
拷贝文件到本地目录：
    hadoop fs -get hdfs:/test.txt file:/root/

拷贝文件并重命名，可以简写：
    hadoop fs -get /test.txt /root/test.txt
```

copyToLocal 用法

```
拷贝文件到本地目录：
    hadoop fs -copyToLocal hdfs:/test.txt file:/root/

拷贝文件并重命名，可以简写：
    hadoop fs -copyToLocal /test.txt /root/test.txt
```

4) 拷贝文件、目录（cp）

```
从本地到 HDFS，同 put
    hadoop fs -cp file:/test.txt hdfs:/test2.txt

从 HDFS 到 HDFS
    hadoop fs -cp hdfs:/test.txt hdfs:/test2.txt
    hadoop fs -cp /test.txt /test2.txt
```

5) 移动文件（mv）

```
hadoop fs -mv hdfs:/test.txt hdfs:/dir/test.txt
hadoop fs -mv /test.txt /dir/test.txt
```

6) 删除文件、目录（rm）

```
删除指定文件
    hadoop fs -rm /a.txt

删除全部 txt 文件
    hadoop fs -rm /*.txt

递归删除全部文件和目录
    hadoop fs -rm -R /dir/
```

7) 读取文件（cat、tail）

```
hadoop fs -cat /test.txt   #以字节码的形式读取
hadoop fs -tail /test.txt
```

8) 创建空文件（touchz）

```
hadoop fs - touchz /newfile.txt
```

9) 创建文件夹（mkdir）

```
hadoop fs -mkdir /newdir /newdir2                    #可以同时创建多个
hadoop fs -mkdir -p /newpkg/newpkg2/newpkg3    #同时创建父级目录
```

10) 获取逻辑空间文件、目录大小（du）

```
hadoop fs - du /    #显示 HDFS 根目录中各文件和文件夹大小
hadoop fs -du -h /       #以最大单位显示 HDFS 根目录中各文件和文件夹大小
```

```
hadoop fs -du -s /        #仅显示 HDFS 根目录大小。即各文件和文件夹大小之和
```

# 第五章　Mapreduce 计算框架

如果将 Hadoop 比做一头大象 ,那么 MapReduce 就是那头大象的电脑。MapReduce 是 Hadoop 核心编程模型。在 Hadoop 中 ,数据处理核心就是 MapReduce 程序设计模型。

本章内容：

1) MapReduce 编程模型

2) MapReduce 执行流程

3) MapReduce 数据本地化

4) MapReduce 工作原理

5) MapReduce 错误处理机制

## 1. MapReduce 编程模型

Map 和 Reduce 的概念是从函数式变成语言中借来的，整个 MapReduce 计算过程分为 Map 阶段和 Reduce 阶段，也称为映射和缩减阶段，这两个独立的阶段实际上是两个独立的过程，即 Map 过程和 Reduce 过程，在 Map 中进行数据的读取和预处理，之后将预处理的结果发送到 Reduce 中进行合并。

我们通过一个代码案例，让大家快速熟悉如何通过代码，快速实现一个我们自己的 MapReduce。

案例：分布式计算出一篇文章中的各个单词出现的次数，也就是 WordCount。

1) 创建 map.py 文件，写入以下代码：

```
#!/usr/bin/env python
import sys
word_list = []
for line in sys.stdin:
    word_list = line.strip().split(' ')
    if len(word_list) <= 0:
        continue
    for word in word_list:
        w = word.strip()
        if len(w) <= 0:
            continue
        print '\t'.join([w, "1"])
```

该代码主要工作是从文章数据源逐行读取，文章中的单词之间以空格分割，

word_list = line.strip().split(' ')这块代码是将当前读取的一整行数据按照空格分割，将分割后的结果存入 word_list 数组中，然后通过 for word in word_list 遍历数组，取出每个单词，后面追加“1”标识当前 word 出现 1 次。

2) 创建 reduce.py，写入以下代码：

```
#!/usr/bin/env python

import sys

cur_word = None
sum_of_word = 0

for line in sys.stdin:
    ss = line.strip().split('\t')
    if len(ss) != 2:
        continue
    word = ss[0].strip()
    count = ss[1].strip()

    if cur_word == None:
        cur_word = word

    if cur_word != word:
        print '\t'.join([cur_word, str(sum_of_word)])
        sum_of_word = 0
        cur_word = word
```

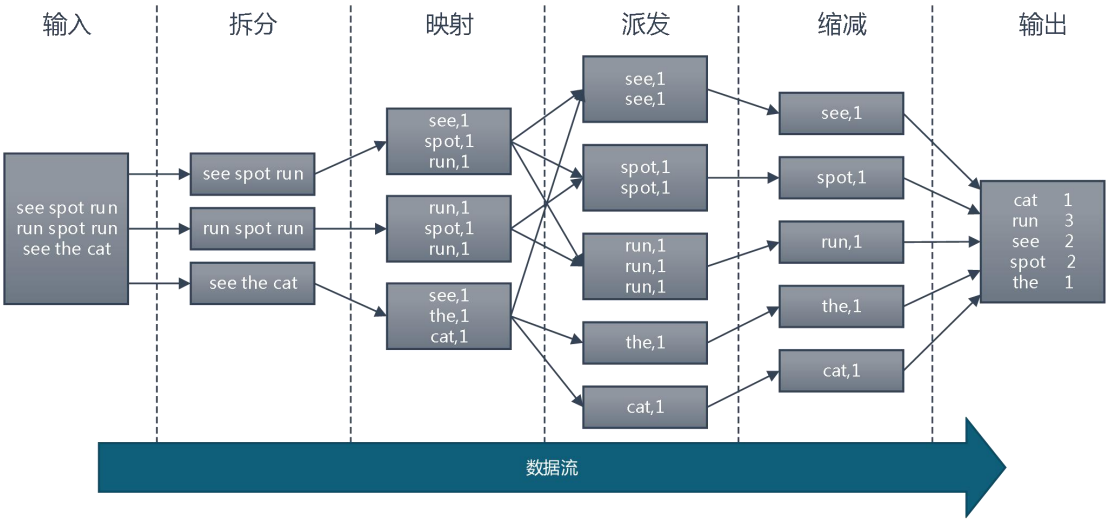
```
        sum_of_word += int(count)

print '\t'.join([cur_word, str(sum_of_word)])
sum_of_word = 0
```

该代码针对 map 阶段的数组进行汇总处理，map 到 reduce 过程中默认存在 shuffle partition 分组机制，保证同一个 word 的记录，会连续传输到 reduce 中，所以在 reduce 阶段只需要对连续相同的 word 后面的技术进行累加求和即可。

3) 本地模拟测试脚本：

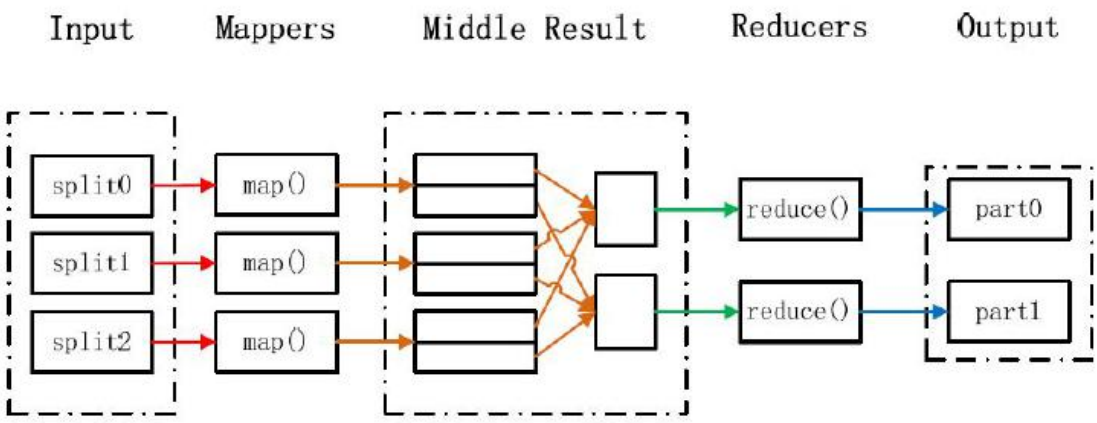
```
]$ cat big.txt | python map.py | sort -k1 | python reduce.py
cat       1
run       3
see       2
spot      2
the       1
```

6) 脚本执行流程：




## 2. MapReduce 执行流程

上面的例子属于 MapReduce 计算框架的一般流程，经过整理总结：




1) 输入和拆分：

不属于 map 和 reduce 的主要过程，但属于整个计算框架消耗时间的一部分，该部分会为正式的 map 准备数据。

**分片（split）操作：**

split 只是将源文件的内容分片形成一系列的 InputSplit，每个 InputSpilt 中存储着对应分片的数据信息（例如，文件块信息、起始位置、数据长度、所在节点列表…），并不是将源文件分割成多个小文件，每个 InputSplit 都由一个 mapper 进行后续处理。

每个分片大小参数是很重要的，splitSize 是组成分片规则很重要的一个参数，该参数由三个值来确定：

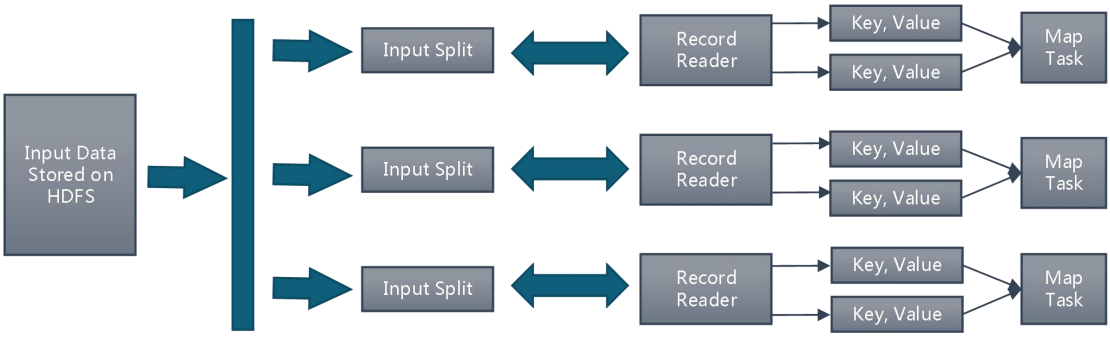
- minSize：splitSize 的最小值，由 mapred-site.xml 配置文件中 mapred.min.split.size 参数确定。
- maxSize：splitSize 的最大值，由 mapred-site.xml 配置文件中 mapreduce.jobtracker.split.metainfo.maxsize 参数确定。
- blockSize：HDFS 中文件存储的快大小，由 hdfs-site.xml 配置文件中 dfs.block.size 参数确定。

splitSize 的确定规则：splitSize=max{minSize，min{maxSize，blockSize}}

**数据格式化（Format）操作：**

将划分好的 InputSplit 格式化成键值对形式的数据。其中 key 为偏移量，value 是每一行的内容。

值得注意的是，在 map 任务执行过程中，会不停的执行数据格式化操作，每生成一个键值对就会将其传入 map，进行处理。所以 map 和数据格式化操作并不存在前后时间差，而是同时进行的。




2) Map 映射：

是 Hadoop 并行性质发挥的地方。根据用户指定的 map 过程，MapReduce 尝试在数据所在机器上执行该 map 程序。在 HDFS 中，文件数据是被复制多份的，所以计算将会选择拥有此数据的最空闲的节点。

在这一部分，map 内部具体实现过程，可以由用户自定义。

3) Shuffle 派发：

Shuffle 过程是指 Mapper 产生的直接输出结果，经过一系列的处理，成为最终的 Reducer 直接输入数据为止的整个过程。这是 mapreduce 的核心过程。该过程可以分为两个阶段：

**Mapper 端的 Shuffle：**由 Mapper 产生的结果并不会直接写入到磁盘中，而是先存储在内存中，当内存中的数据量达到设定的阀值时，一次性写入到本地磁盘中。并同时进行 sort（排序）、combine（合并）、partition（分片）等操作。其中，sort 是把 Mapper 产生的结果按照 key 值进行排序；combine 是把 key 值相同的记录进行合并；partition 是把

数据均衡的分配给 Reducer。

**Reducer 端的 Shuffle：**由于 Mapper 和 Reducer 往往不在同一个节点上运行，所以 Reducer 需要从多个节点上下载 Mapper 的结果数据，并对这些数据进行处理，然后才能被 Reducer 处理。

4) Reduce 缩减：

Reducer 接收形式的数据流，形成形式的输出，具体的过程可以由用户自定义，最终结果直接写入 hdfs。每个 reduce 进程会对应一个输出文件，名称以 part-开头。

## 3. MapReduce 数据本地化（Data-Local）

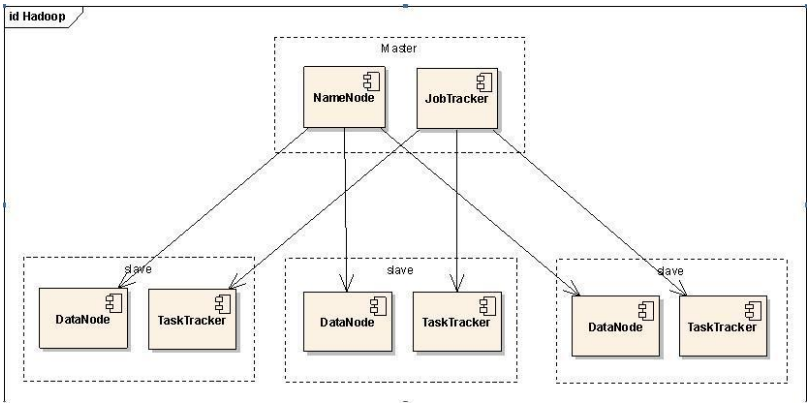
首先，HDFS 和 MapReduce 是 Hadoop 的核心设计。对于 HDFS，是存储基础，在数据层面上提供了海量数据存储的支持。而 MapReduce，是在数据的上一层，通过编写 MapReduce 程序对海量数据进行计算处理。

在前面 HDFS 章节中，知道了 NameNode 是文件系统的名字节点进程，DataNode 是文件系统的数据节点进程。

MapReduce 计算框架中负责计算任务调度的 JobTracker 对应 HDFS 的 NameNode 的角色，只不过一个负责计算任务调度，一个负责存储任务调度。

MapReduce 计算框架中负责真正计算任务的 TaskTracker 对应到 HDFS 的 DataNode 的角色，一个负责计算，一个负责管理存储数据。

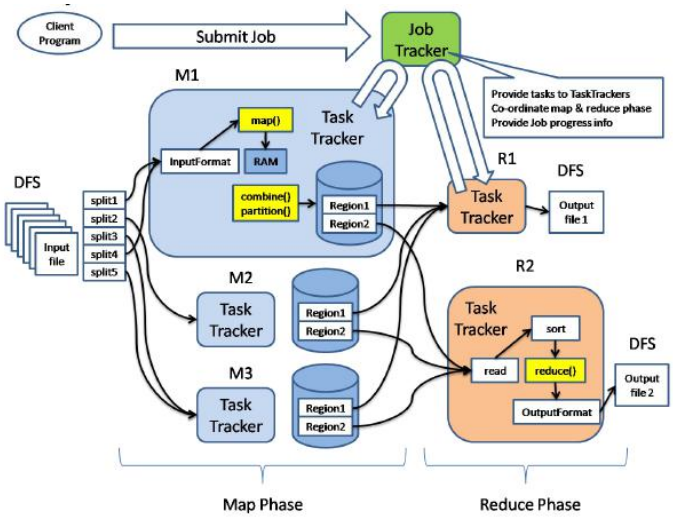
考虑到“本地化原则”，一般地，将 NameNode 和 JobTracker 部署到同一台机器上，各个 DataNode 和 TaskNode 也同样部署到同一台机器上。




这样做的目的是将 map 任务分配给含有该 map 处理的数据块的 TaskTracker 上，同时将程序 JAR 包复制到该 TaskTracker 上来运行，这叫“运算移动，数据不移动”。而分配 reduce 任务时并不考虑数据本地化。

## 4. MapReduce 工作原理

我们通过 Client、JobTrask 和 TaskTracker 的角度来分析 MapReduce 的工作原理：

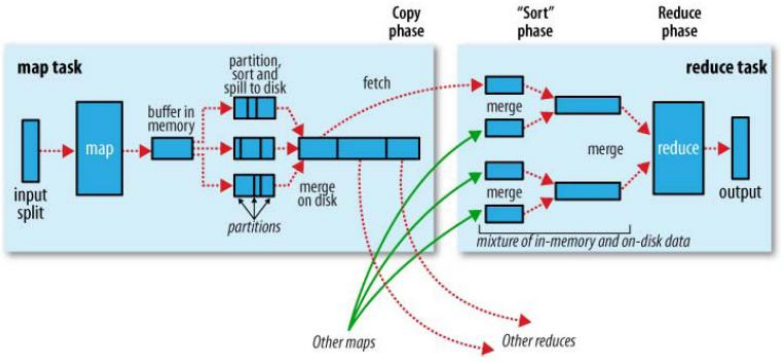



首先在客户端（Client）启动一个作业（Job），向 JobTracker 请求一个 Job ID。将运行作业所需要的资源文件复制到 HDFS 上，包括 MapReduce 程序打包的 JAR 文件、配置文件和客户端计算所得的输入划分信息。这些文件都存放在 JobTracker 专门为该作业创建的文件夹中，文件夹名为该作业的 Job ID。JAR 文件默认会有 10 个副本（mapred.submit.replication 属性控制）;输入划分信息告诉了 JobTracker 应该为这个作

业启动多少个 map 任务等信息。

JobTracker 接收到作业后，将其放在一个作业队列里，等待作业调度器对其进行调度当作业调度器根据自己的调度算法调度到该作业时，会根据输入划分信息为每个划分创建一个 map 任务，并将 map 任务分配给 TaskTracker 执行。对于 map 和 reduce 任务，TaskTracker 根据主机核的数量和内存的大小有固定数量的 map 槽和 reduce 槽。这里需要强调的是：map 任务不是随随便便地分配给某个 TaskTracker 的，这里就涉及到上面提到的数据本地化（Data-Local）。

TaskTracker 每隔一段时间会给 JobTracker 发送一个心跳，告诉 JobTracker 它依然在运行，同时心跳中还携带着很多的信息，比如当前 map 任务完成的进度等信息。当 JobTracker 收到作业的最后一个任务完成信息时，便把该作业设置成“成功”。当 JobClient 查询状态时，它将得知任务已完成，便显示一条消息给用户。




如果具体从 map 端和 reduce 端分析，可以参考上面的图片，具体如下：

**Map 端流程：**

1) 每个输入分片会让一个 map 任务来处理，map 输出的结果会暂且放在一个环形内存缓冲区中（该缓冲区的大小默认为 100M，由 io.sort.mb 属性控制），当该缓冲区快要溢出时（默认为缓冲区大小的 80%，由 io.sort.spill.percent 属性控制），会在本地文件系统中创建一个溢出文件，将该缓冲区中的数据写入这个文件。

2) 在写入磁盘之前 ,线程首先根据reduce任务的数目将数据划分为相同数目的分区，也就是一个 reduce 任务对应一个分区的数据。这样做是为了避免有些 reduce 任务分配到大量数据，而有些 reduce 任务却分到很少数据，甚至没有分到数据的尴尬局面。其实分区就是对数据进行 hash 的过程。然后对每个分区中的数据进行排序，如果此时设置了 Combiner，将排序后的结果进行 Combine 操作，这样做的目的是让尽可能少的数据写入到磁盘。

3) 当 map 任务输出最后一个记录时，可能会有很多的溢出文件，这时需要将这些文件合并。合并的过程中会不断地进行排序和 Combine 操作，目的有两个：

- 尽量减少每次写入磁盘的数据量；
- 尽量减少下一复制阶段网络传输的数据量。

最后合并成了一个已分区且已排序的文件。为了减少网络传输的数据量，这里可以将数据压缩，只要将 mapred.compress.map.out 设置为 true 就可以了。

4) 将分区中的数据拷贝给相对应的 reduce 任务。分区中的数据怎么知道它对应的 reduce 是哪个呢？其实 map 任务一直和其父 TaskTracker 保持联系，而 TaskTracker 又一直和 JobTracker 保持心跳。所以 JobTracker 中保存了整个集群中的宏观信息。只要 reduce 任务向 JobTracker 获取对应的 map 输出位置就可以了。

**Reduce 端流程：**

1) Reduce 会接收到不同 map 任务传来的数据 ,并且每个 map 传来的数据都是有序的。如果 reduce 端接受的数据量相当小，则直接存储在内存中（缓冲区大小由 mapred.job.shuffle.input.buffer.percent 属性控制 ,表示用作此用途的堆空间的百分比），如果数据量超过了该缓冲区大小的一定比例（由

mapred.job.shuffle.merge.percent 决定），则对数据合并后溢写到磁盘中。

2) 随着溢写文件的增多，后台线程会将它们合并成一个更大的有序的文件，这样做是为了给后面的合并节省时间。其实不管在 map 端还是 reduce 端，MapReduce 都是反复地执行排序，合并操作，所以排序是 hadoop 的灵魂。

3) 合并的过程中会产生许多的中间文件（写入磁盘了），但 MapReduce 会让写入磁盘的数据尽可能地少，并且最后一次合并的结果并没有写入磁盘，而是直接输入到 reduce 函数。

在 Map 处理数据后，到 Reduce 得到数据之前，这个流程在 MapReduce 中可以看做是一个 Shuffle 的过程。

在经过 mapper 的运行后，我们得知 mapper 的输出是这样一个 key/value 对。到底当前的 key 应该交由哪个 reduce 去做呢，是需要现在决定的。 MapReduce 提供 Partitioner 接口，它的作用就是根据 key 或 value 及 reduce 的数量来决定当前的这对输出数据最终应该交由哪个 reduce task 处理。默认对 key 做 hash 后再以 reduce task 数量取模。默认的取模方式只是为了平均 reduce 的处理能力，如果用户自己对 Partitioner 有需求，可以订制并设置到 job 上。

## 5. MapReduce 错误处理机制

MapReduce 任务执行过程中出现的故障可以分为两大类 :硬件故障和任务执行失败引发的故障。

1) 硬件故障

在 Hadoop Cluster 中，只有一个 JobTracker，因此，JobTracker 本身是存在单点故障的。如何解决 JobTracker 的单点问题呢？我们可以采用主备部署方式，启动 JobTracker

主节点的同时，启动一个或多个 JobTracker 备用节点。当 JobTracker 主节点出现问题时，通过某种选举算法，从备用的 JobTracker 节点中重新选出一个主节点。

机器故障除了 JobTracker 错误就是 TaskTracker 错误。TaskTracker 故障相对较为常见，MapReduce 通常是通过重新执行任务来解决该故障。

在 Hadoop 集群中，正常情况下，TaskTracker 会不断的与 JobTracker 通过心跳机制进行通信。如果某 TaskTracker 出现故障或者运行缓慢，它会停止或者很少向 JobTracker 发送心跳。如果一个 TaskTracker 在一定时间内（默认是 1 分钟）没有与 JobTracker 通信，那么 JobTracker 会将此 TaskTracker 从等待任务调度的 TaskTracker 集合中移除。同时 JobTracker 会要求此 TaskTracker 上的任务立刻返回。如果此 TaskTracker 任务仍然在 mapping 阶段的 Map 任务，那么 JobTracker 会要求其他的 TaskTracker 重新执行所有原本由故障 TaskTracker 执行的 Map 任务。如果任务是在 Reduce 阶段的 Reduce 任务，那么 JobTracker 会要求其他 TaskTracker 重新执行故障 TaskTracker 未完成的 Reduce 任务。比如:一个 TaskTracker 已经完成被分配的三个 Reduce 任务中的两个，因为 Reduce 任务一旦完成就会将数据写到 HDFS 上，所以只有第三个未完成的 Reduce 需要重新执行。但是对于 Map 任务来说，即使 TaskTracker 完成了部分 Map，Reduce 仍可能无法获取此节点上所有 Map 的所有输出。所以无论 Map 任务完成与否，故障 TaskTracker 上的 Map 任务都必须重新执行。

2) 任务执行失败引发的故障

在实际任务中，MapReduce 作业还会遇到用户代码缺陷或进程崩溃引起的任务失败等情况。用户代码缺陷会导致它在执行过程中抛出异常。此时,任务 JVM 进程会自动退出，并向 TaskTracker 父进程发送错误消息，同时错误消息也会写入 log 文件，最后 TaskTracker 将此次任务尝试标记失败。对于进程崩溃引起的任务失败，TaskTracker 的监听程序会发现

进程退出，此时 TaskTracker 也会将此次任务尝试标记为失败。对于死循环程序或执行时间太长的程序，由于 TaskTracker 没有接收到进度更新，它也会将此次任务尝试标记为失败，并杀死程序对应的进程。

在以上情况中，TaskTracker 将任务尝试标记为失败之后会将 TaskTracker 自身的任务计数器减 1，以便想 JobTracker 申请新的任务。 TaskTracker 也会通过心跳机制告诉 JobTracker 本地的一个任务尝试失败。JobTracker 接到任务失败的通知后，通过重置任务状态，将其加入到调度队列来重新分配该任务执行（JobTracker 会尝试避免将失败的任务再次分配给运行失败的 TaskTracker）。如果此任务尝试了 4 次（次数可以进行设置）仍没有完成，就不会再被重试，此时整个作业也就失败了。

# 第六章 Zookeeper

Zookeeper 是一种分布式的，开源的，应用于分布式应用的协作服务。它提供了一些简单的操作，使得分布式应用可以基于这些接口实现诸如同步、配置维护和分集群或者命名的服务。Zookeeper 很容易编程接入，它使用了一个和文件树结构相似的数据模型。可以使用 Java 或者 C 来进行编程接入。

众所周知，分布式的系统协作服务很难有让人满意的产品。这些协作服务产品很容易陷入一些诸如竞争选择条件或者死锁的陷阱中。Zookeeper 的目的就是将分布式服务不再需要由于协作冲突而另外实现协作服务。

本章内容：

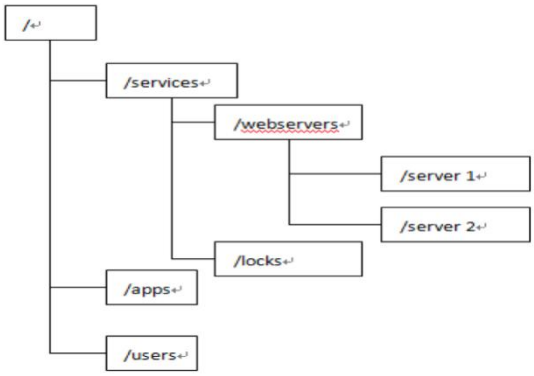
1) Zookeeper 数据模型

2) Zookeeper 访问控制

3) Zookeeper 应用场景

# 1. Zookeeper 数据模型

ZooKeeper 拥有一个层次的命名空间，这个和标准的文件系统非常相似




从图中我们可以看出 ZooKeeper 的数据模型，在结构上和标准文件系统的非常相似，都是采用这种树形层次结构，ZooKeeper 树中的每个节点被称为—Znode。和文件系统的目录树一样，ZooKeeper 树中的每个节点可以拥有子节点。但也有不同之处：

1) 引用方式:

Zonde 通过路径引用，如同 Unix 中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。在 ZooKeeper 中，路径由 Unicode 字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。

2) Znode 结构

ZooKeeper 命名空间中的 Znode，兼具文件和目录两种特点。既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分。图中的每个节点称为一个 Znode。 每个 Znode 由 3 部分组成:

- stat：此为状态信息, 描述该 Znode 的版本, 权限等信息

- data：与该 Znode 关联的数据
- children：该 Znode 下的子节点

ZooKeeper 虽然可以关联一些数据，但并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以 KB 为大小单位。ZooKeeper 的服务器和客户端都被设计为严格检查并限制每个 Znode 的数据大小至多 1M，但常规使用中应该远小于此值。

3) 数据访问

ZooKeeper 中的每个节点存储的数据要被原子性的操作。也就是说读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的 ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。

4) 节点类型

Persistent Nodes：永久有效地节点,除非 client 显式的删除,否则一直存在。

Ephemeral Nodes：临时节点,仅在创建该节点 client 保持连接期间有效,一旦连接丢失,zookeeper 会自动删除该节点。

Sequence Nodes：顺序节点,client 申请创建该节点时, ZooKeeper 会自动在节点路径末尾添加递增序号,这种类型是实现分布式锁,分布式 queue 等特殊功能的关键。

5) 监控

客户端可以在节点上设置 watch，我们称之为监视器。当节点状态发生改变时(Znode 的增、删、改)将会触发 watch 所对应的操作。当 watch 被触发时，ZooKeeper 将会向客户端发送且仅发送一条通知，因为 watch 只能被触发一次，这样可以减少网络流量。

ZooKeeper 可以为所有的读操作设置 watch ,这些读操作包括 :exists()、getChildren()及 getData()。watch 事件是一次性的触发器，当 watch 的对象状态发生改变时，将会触发此对象上 watch 所对应的事件。watch 事件将被异步地发送给客户端，并且 ZooKeeper 为 watch 机制提供了有序的一致性保证。理论上，客户端接收 watch 事件的时间要快于其看到 watch 对象状态变化的时间。

## 2. Zookeeper 访问控制

传统的文件系统中，ACL 分为两个维度，一个是属组，一个是权限，子目录/文件默认继承父目录的 ACL。而在 Zookeeper 中，node 的 ACL 是没有继承关系的，是独立控制的。Zookeeper 的 ACL，可以从三个维度来理解：一是 scheme; 二是 user; 三是 permission，通常表示为 scheme:id:permissions, 下面从这三个方面分别来介绍：

1) scheme: scheme 对应于采用哪种方案来进行权限管理，zookeeper 实现了一个 pluggable 的 ACL 方案，可以通过扩展 scheme，来扩展 ACL 的机制。

| 模式 | 描述 |
| --- | --- |
| World | 它下面只有一个 id, 叫 anyone, world:anyone 代表任何人 ,zookeeper 中对所有人有权限的结点就是属于 world:anyone 的 |
| Auth | 已经被认证的用户 |
| Digest | 通过 username：password 字符串的 MD5 编码认证用户 |
| Host | 匹配主机名后缀，如，host:corp.com 匹配 host:host1.corp.com, host:host2.corp.com，但不能匹配 host:host1.store.com |
| IP | 通过 IP 识别用户，表达式格式为 addr/bits |

2) User：与 scheme 是紧密相关的，具体的情况在上面介绍 scheme 的过程都已介

绍，这里不再赘述。

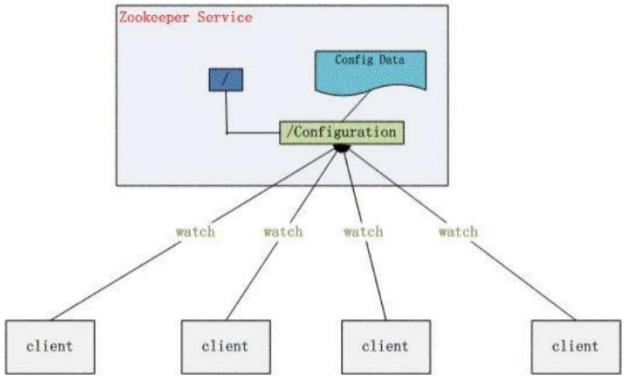
3) permission: zookeeper 目前支持下面一些权限：

| 权限 | 描述 | 备注 |
| --- | --- | --- |
| Create | 有创建子节点的权限 | |
| Read | 有读取节点数据和子节点列表的权限 | |
| Write | 有修改节点数据的权限 | 无创建和删除子节点的权限 |
| Delete | 有删除子节点的权限 | |
| Admin | 有设置节点权限的权限 | |

# 3. Zookeeper 应用场景

1) 数据发布与订阅（配置中心）

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到 ZK 节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，服务式服务框架的服务地址列表等就非常适合使用。




2) 分布式锁服务

分布式锁，这个主要得益于 ZooKeeper 为我们保证了数据的强一致性。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

3) 分布式队列

队列方面，简单地讲有两种，一种是常规的先进先出队列，另一种是要等到队列成员聚齐之后的才统一按序执行。对于第一种先进先出队列，和分布式锁服务中的控制时序场景基本原理一致，这里不再赘述。 第二种队列其实是在 FIFO 队列的基础上作了一个增强。通常可以在 /queue 这个 znode 下预先建立一个/queue/num 节点，并且赋值为 n（或者直接给/queue 赋值 n），表示队列大小，之后每次有队列成员加入后，就判断下是否已经到达队列大小，决定是否可以开始执行了。这种用法的典型场景是，分布式环境中，一个大任务 Task A，需要在很多子任务完成（或条件就绪）情况下才能进行。这个时候，凡是其中一个子任务完成（就绪），那么就去 /taskList 下建立自己的临时时序节点（CreateMode.EPHEMERAL_SEQUENTIAL），当 /taskList 发现自己下面的子节点满足指定个数，就可以进行下一步按序进行处理了。

# 第七章 HBase

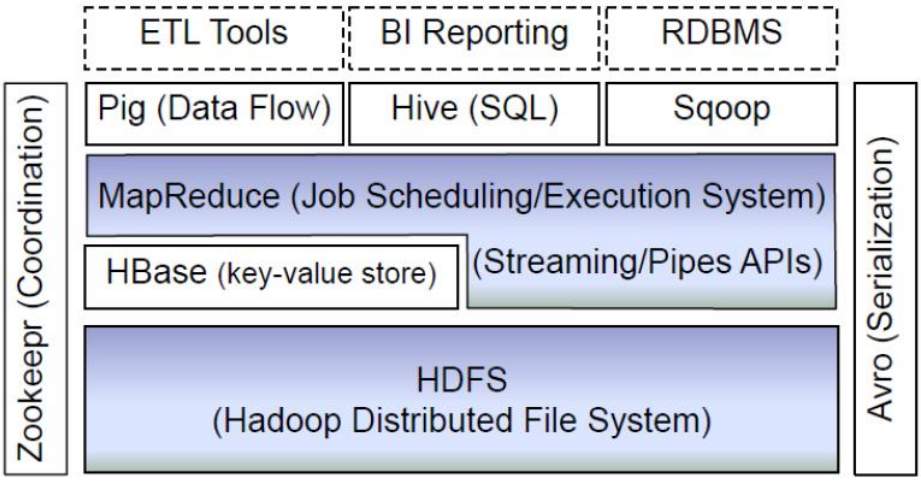
HBase – Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群。

本章内容：

1) Hbase 简介
2) Hbase 数据模型
3) Hbase 基础原理
4) Hbase 系统架构
5) Hbase 基础操作

# 1. Hbase 简介

HBase 是 Apache Hadoop 中的一个子项目，Hbase 依托于 Hadoop 的 HDFS 作为最基本存储基础单元，通过使用 hadoop 的 DFS 工具就可以看到这些这些数据存储文件夹的结构,还可以通过 Map/Reduce 的框架(算法)对 HBase 进行操作。




上图描述了 Hadoop EcoSystem 中的各层系统，其中 HBase 位于结构化存储层，Hadoop HDFS 为 HBase 提供了高可靠性的底层存储支持，Hadoop MapReduce 为 HBase 提供了高性能的计算能力，Zookeeper 为 HBase 提供了稳定服务和 failover 机制。

**Hbase 适用场景：**

1) 大数据量存储，大数据量高并发操作

2) 需要对数据随机读写操作

3) 读写访问均是非常简单的操作

**Hbase 与 HDFS 对比：**

| | Plain HDFS/MR | HBase |
|---|---|---|
| Write pattern | Append-only | Random write, bulk incremental |
| Read pattern | Full table scan, partition table scan | Random read, small range scan, or table scan |
| Hive (SQL) performance | Very good | 4-5x slower |
| Structured storage | Do-it-yourself / TSV / SequenceFile / Avro / ? | Sparse column-family data model |
| Max data size | 30+ PB | ~1PB |



- 两者都具有良好的容错性和扩展性，都可以扩展到成百上千个节点；
- HDFS 适合批处理场景，但不支持数据随机查找，不适合增量数据处理，不支持数据更新

## 2. Hbase 数据模型

HBase 以表的形式存储数据。表由行和列族组成。列划分为若干个列族(row family)，其逻辑视图如下：

| 行健 | 时间戳 | 列族 contents | 列族 anchor | 列族 mime |
|---|---|---|---|---|
| “com.cnn.www” | T9 | | Anchor:cnnsi.com= “CNN” | |
| | T8 | | Anchor:my.look.ca= “CNN.com” | |
| | T6 | Contents:html= “<html>……” | | Mime.type= “text/html” |
| | T5 | Contents:html= “<html>……” | | |
| | T3 | Contents:html= “<html>……” | | |

几个关键概念：

1) 行键(RowKey)

- 行键是字节数组，任何字符串都可以作为行键；
- 表中的行根据行键进行排序，数据按照 Row key 的字节序(byte order)排序存储；
- 所有对表的访问都要通过行键 （单个 RowKey 访问，或 RowKey 范围访问，或全表扫描)

2) 列族（ColumnFamily）

- CF 必须在表定义时给出
- 每个 CF 可以有一个或多个列成员(ColumnQualifier)，列成员不需要在表定义时给出，新的列族成员可以随后按需、动态加入
- 数据按 CF 分开存储，HBase 所谓的列式存储就是根据 CF 分开存储（每个 CF 对应一个 Store），这种设计非常适合于数据分析的情形

3) 时间戳（TimeStamp）

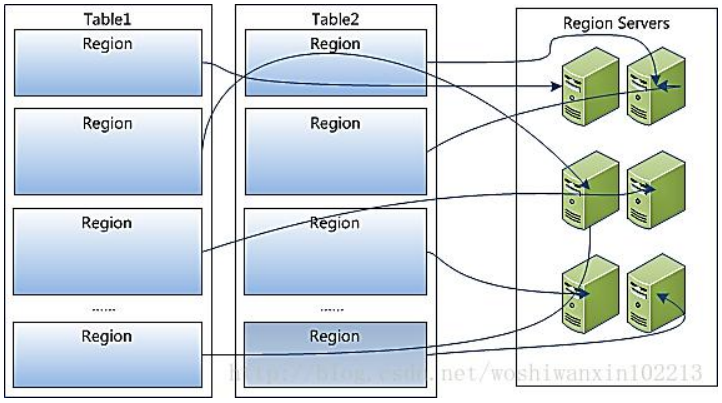
- 每个 Cell 可能又多个版本，它们之间用时间戳区分

4) 单元格（Cell）

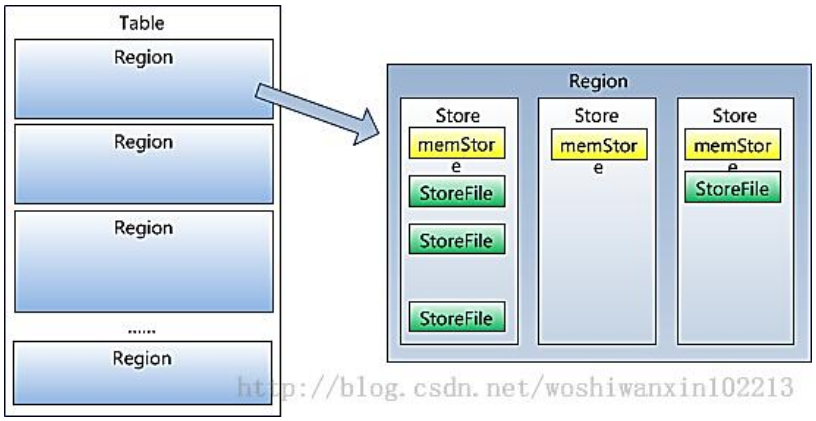
- Cell 由行键，列族:限定符，时间戳唯一决定
- Cell 中的数据是没有类型的，全部以字节码形式存储

5) 区域(Region）

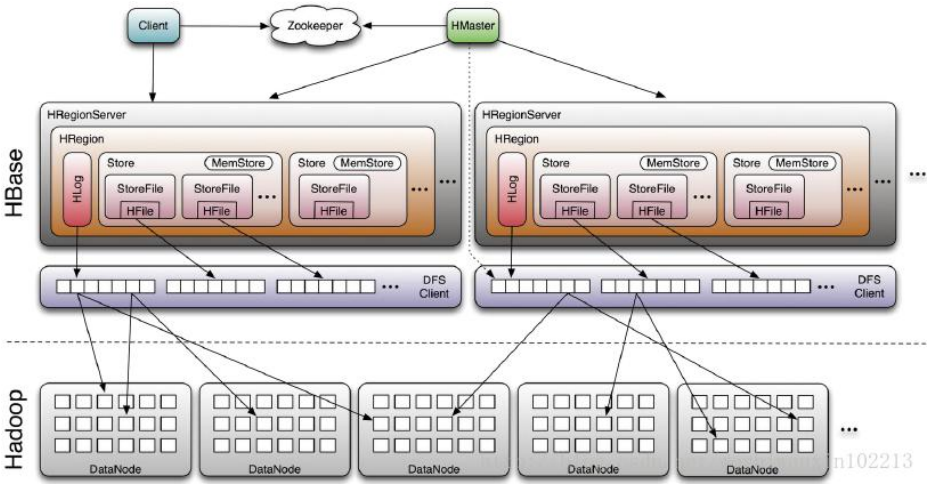
- HBase 自动把表水平（按 Row）划分成多个区域(region)，每个 region 会保存一个表里面某段连续的数据；
- 每个表一开始只有一个 region，随着数据不断插入表，region 不断增大，当增大到一个阀值的时候，region 就会等分会两个新的 region；
- 当 table 中的行不断增多，就会有越来越多的 region。这样一张完整的表被保存在多个 Region 上。




- Region 虽然是分布式存储的最小单元，但并不是存储的最小单元。Region 由一个或者多个 Store 组成，每个 store 保存一个 columns family；每个 Strore 又由一个 memStore 和 0 至多个 StoreFile 组成 ,StoreFile 包含 HFile；memStore 存储在内存中，StoreFile 存储在 HDFS 上。




## 3. Hbase 架构及基本组件



从上图看到 HBase 的基本组件：

1) Client：

- 包含访问 HBase 的接口，并维护 cache 来加快对 HBase 的访问，比如 region 的位置信息。

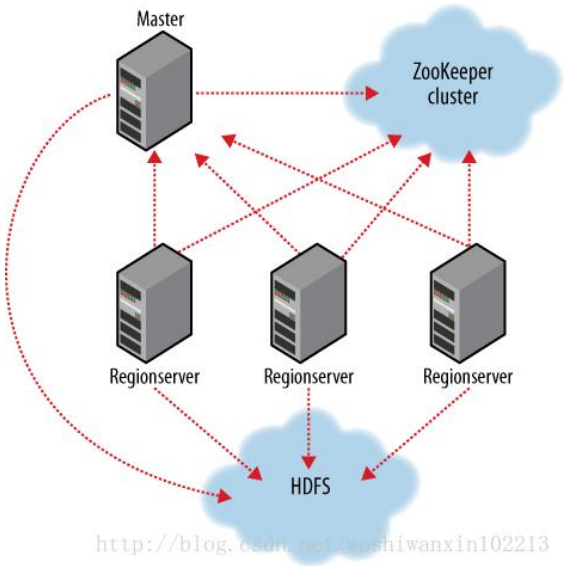
2) Master

- 为 Region server 分配 region
- 负责 Region server 的负载均衡
- 发现失效的 Region server 并重新分配其上的 region
- 管理用户对 table 的增删改查操作

3) Region Server

- Regionserver 维护 region，处理对这些 region 的 IO 请求
- Regionserver 负责切分在运行过程中变得过大的 region

4) Zookeeper 作用

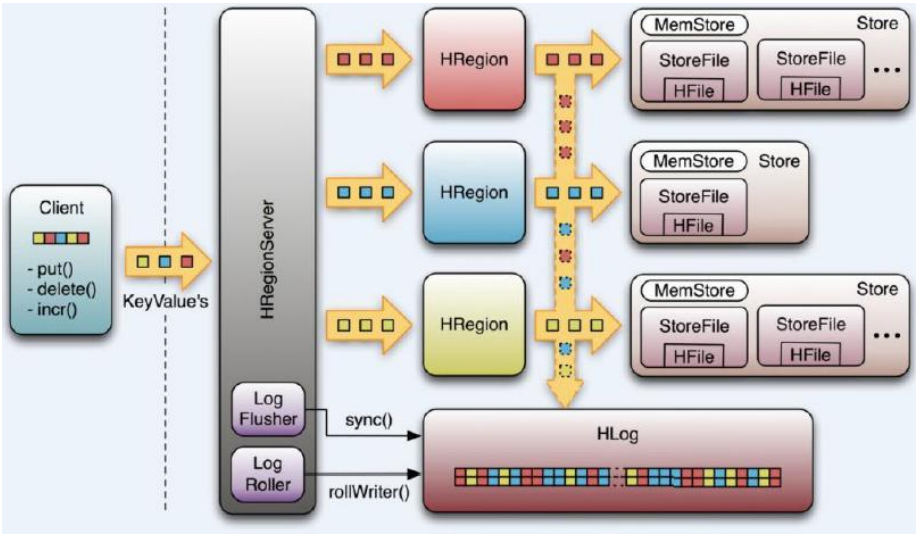




- 通过选举，保证任何时候，集群中只有一个 Master，Master 与 RegionServers 启动时会向 ZooKeeper 注册

- 存储所有 Region 的寻址入口
- 实时监控 Region server 的上线和下线信息，并实时通知给 Master
- 存储 HBase 的 schema 和 table 元数据
- 默认情况下，HBase 管理 ZooKeeper 实例，比如， 启动或者停止 ZooKeeper
- Zookeeper 的引入使得 Master 不再是单点故障

## 4. Hbase 容错与恢复




每个 HRegionServer 中都有一个 HLog 对象，HLog 是一个实现 Write Ahead Log 的类，在每次用户操作写入 MemStore 的同时，也会写一份数据到 HLog 文件中（HLog 文件格式见后续），HLog 文件定期会滚动出新的，并删除旧的文件（已持久化到 StoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 Zookeeper 感知到，HMaster 首先会处理遗留的 HLog 文件，将其中不同 Region 的 Log 数据进行拆分，分别放到相应 region 的目录下，然后再将失效的 region 重新分配，领取到这些 region 的 HRegionServer 在 Load Region 的过程中，会发现有历史 HLog 需要处理，因此会 Replay HLog 中的数据到 MemStore 中，然后 flush 到 StoreFiles，完成数据恢复

**Hbase 容错性：**

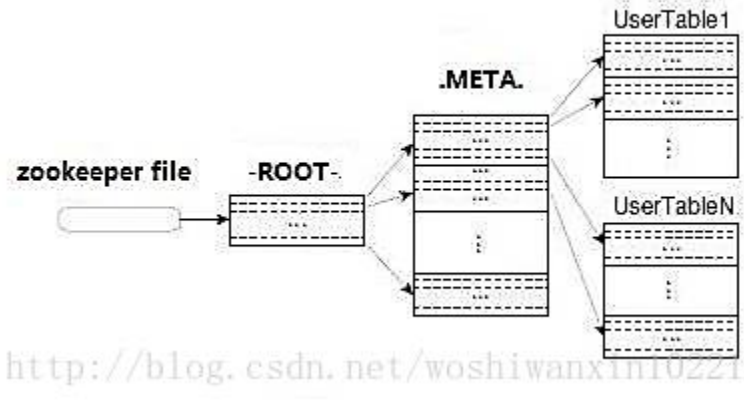
1) Master 容错：Zookeeper 重新选择一个新的 Master

   - 无 Master 过程中，数据读取仍照常进行；
   - 无 master 过程中，region 切分、负载均衡等无法进行；

2) RegionServer 容错：定时向 Zookeeper 汇报心跳，如果一旦时间内未出现心跳，Master 将该 RegionServer 上的 Region 重新分配到其他 RegionServer 上，失效服务器上“预写”日志由主服务器进行分割并派送给新的 RegionServer

3) Zookeeper 容错 :Zookeeper 是一个可靠地服务，一般配置 3 或 5 个 Zookeeper 实例

**Region 定位流程：**





寻找 RegionServer 过程：ZooKeeper--> -ROOT-(单 Region)--> .META.--> 用户表

1) -ROOT-

   - 表包含.META.表所在的 region 列表，该表只会有一个 Region；
   - Zookeeper 中记录了-ROOT-表的 location。

2) .META.

   - 表包含所有的用户空间 region 列表，以及 RegionServer 的服务器地址。

# 5. Hbase 基础操作

1) 进入 hbase shell console

```
$HBASE_HOME/bin/hbase shell
```

**表的管理：**

2) 查看有哪些表

```
list
```

3) 创建表

```
# 语法：create <table>, {NAME => <family>, VERSIONS => <VERSIONS>}

# 例如：创建表 t1，有两个 family name：f1，f2，且版本数均为 2
> create 't1',{NAME => 'f1', VERSIONS => 2},{NAME => 'f2', VERSIONS => 2}
```

4) 删除表

```
# 分两步：首先 disable，然后 drop

# 例如：删除表 t1
> disable 't1'
> drop 't1'
```

5) 查看表的结构

```
# 语法：describe <table>

# 例如：查看表 t1 的结构
> describe 't1'
```

6) 修改表结构

```
# 修改表结构必须先 disable

# 语法：alter 't1', {NAME => 'f1'}, {NAME => 'f2', METHOD => 'delete'}

# 例如：修改表 test1 的 cf 的 TTL 为 180 天
> disable 'test1'
> alter 'test1',{NAME=>'body',TTL=>'15552000'},{NAME=>'meta', TTL=>'15552000'}
> enable 'test1'
```

**权限管理：**

1) 分配权限

```
# 语法 ：grant <user> <permissions> <table> <column family> <column qualifier> 参数后面用逗号分隔
# 权限用五个字母表示： "RWXCA".
# READ('R'), WRITE('W'), EXEC('X'), CREATE('C'), ADMIN('A')

# 例如，给用户 ‘test'分配对表 t1 有读写的权限，
> grant 'test','RW','t1'
```

2) 查看权限

```
# 语法：user_permission <table>

# 例如，查看表 t1 的权限列表
> user_permission 't1'
```

3) 收回权限

```
# 与分配权限类似，语法：revoke <user> <table> <column family> <column qualifier>

# 例如，收回 test 用户在表 t1 上的权限
> revoke 'test','t1'
```

**表数据的增删改查：**

1) 添加数据

```
# 语法：put <table>,<rowkey>,<family:column>,<value>,<timestamp>

# 例如：给表 t1 的添加一行记录：rowkey 是 rowkey001，family name：f1，column name：col1，value：value01，timestamp：系统默认
> put 't1','rowkey001','f1:col1','value01'
```

2) 查询数据——查询某行记录

```
# 语法：get <table>,<rowkey>,[<family:column>,....]

# 例如：查询表 t1，rowkey001 中的 f1 下的 col1 的值
> get 't1','rowkey001', 'f1:col1'
# 或者：
> get 't1','rowkey001', {COLUMN=>'f1:col1'}

# 查询表 t1，rowke002 中的 f1 下的所有列值
hbase(main)> get 't1','rowkey001'
```

3) 查询数据——扫描表

```
# 语法：scan <table>, {COLUMNS => [ <family:column>,.... ], LIMIT => num}
# 另外，还可以添加 STARTROW、TIMERANGE 和 FITLER 等高级功能
```

```
# 例如：扫描表 t1 的前 5 条数据
> scan 't1',{LIMIT=>5}
```

4) 查询表中的数据行数

```
# 语法：count <table>, {INTERVAL => intervalNum, CACHE => cacheNum}
# INTERVAL 设置多少行显示一次及对应的 rowkey ,默认 1000 ;CACHE 每次去取的缓存区大小，默认是 10，调整该参数可提高查询速度

# 例如，查询表 t1 中的行数，每 100 条显示一次，缓存区为 500
> count 't1', {INTERVAL => 100, CACHE => 500}
```

5) 删除数据——删除行中的某个列值

```
# 语法：delete <table>, <rowkey>, <family:column>，<timestamp>,必须指定列名

# 例如：删除表 t1，rowkey001 中的 f1:col1 的数据
> delete 't1','rowkey001','f1:col1'
```

6) 删除数据——删除行

```
# 语法 :deleteall <table>, <rowkey>, <family:column>，<timestamp> ,可以不指定列名，删除整行数据

# 例如：删除表 t1，rowk001 的数据
> deleteall 't1','rowkey001'
```

7) 删除数据——删除表中的所有数据

```
# 语法： truncate <table>
# 其具体过程是：disable table -> drop table -> create table

# 例如：删除表 t1 的所有数据
> truncate 't1'
```

**Region 管理：**

1) 移动 Region

```
# 语法：move 'encodeRegionName', 'ServerName'
# encodeRegionName 指的 regioName 后面的编码，ServerName 指的是 master-status 的 Region Servers 列表

# 示例
>move '4343995a58be8e5bbc739', 'db-41.xxx.xxx.org,60020,139'
```

2) 开启/关闭 region

```
# 语法：balance_switch true|false
hbase(main)> balance_switch
```

3) 手动 split

```
# 语法：split 'regionName', 'splitKey'
```

4) 手动触发 major compaction

```
#语法：
#Compact all regions in a table:
> major_compact 't1'
#Compact an entire region:
> major_compact 'r1'
#Compact a single column family within a region:
> major_compact 'r1', 'c1'
#Compact a single column family within a table:
> major_compact 't1', 'c1'
```

# 第八章 Hive

hive 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，Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据可以将 sql 语句转换为 MapReduce 任务进行运行，不必开发专门的 MapReduce。毕竟会写 SQL 的人比写 JAVA 的人多，这样可以让一大批运营人员直接获取海量数据。在数据仓库建设中，HIVE 灵活易用且易于维护，十分适合数据仓库的统计分析。

本章内容：

1) Hive 简介

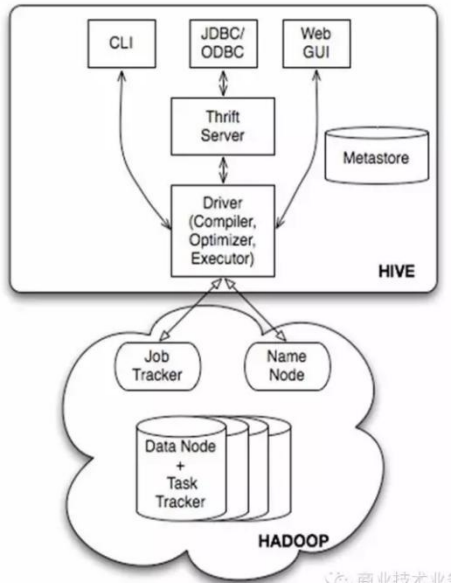
2) Hive 基础组成

3) Hive 执行流程

4) Hive 基础操作

# 1. Hive 基础原理

hive 是建设在 Hadoop 之上，Hive 包括如下组件：CLI（command line interface）、JDBC/ODBC、Thrift Server、WEB GUI、MetaStore 和 Driver(Complier、Optimizer 和 Executor)。




1) Driver 组件：包括 Complier、Optimizer 和 Executor，它的作用是将我们写的 HiveQL（类 SQL）语句进行解析、编译优化，生成执行计划，然后调用底层的 MapReduce 计算框架。
2) Metastore 组件：元数据服务组件存储 hive 的元数据，hive 的元数据存储在关系数据库里，hive 支持的关系数据库有 derby、mysql。Hive 还支持把 metastore 服务安装到远程的服务器集群里，从而解耦 hive 服务和 metastore 服务。
3) Thrift 服务：thrift 是 facebook 开发的一个软件框架，它用来进行可扩展且跨语言的服务的开发，hive 集成了该服务，能让不同的编程语言调用 hive 的接口。
4) CLI：command line interface，命令行接口。

5) Thrift客户端：hive架构的许多客户端接口是建立在thrift客户端之上，包括JDBC和ODBC接口。

6) WEBGUI：hive客户端提供了一种通过网页的方式访问hive所提供的服务。

用户接口主要有三个：CLI，Client 和 WUI。其中最常用的是CLI，公司内可通过堡垒机连接ssh hdp_lbg_ectech@10.126.101.7，直接输入hive，就可连接到HiveServer。

Hive的metastore组件是hive元数据集中存放地。Metastore组件包括两个部分：metastore服务和后台数据的存储。后台数据存储的介质就是关系数据库，例如hive默认的嵌入式磁盘数据库derby，还有mysql数据库。Metastore服务是建立在后台数据存储介质之上，并且可以和hive服务进行交互的服务组件，默认情况下，metastore服务和hive服务是安装在一起的，运行在同一个进程当中。我也可以把metastore服务从hive服务里剥离出来，metastore独立安装在一个集群里，hive远程调用metastore服务，这样我们可以把元数据这一层放到防火墙之后，客户端访问hive服务，就可以连接到元数据这一层，从而提供了更好的管理性和安全保障。使用远程的metastore服务，可以让metastore服务和hive服务运行在不同的进程里，这样也保证了hive的稳定性，提升了hive服务的效率。

对于数据存储，Hive没有专门的数据存储格式，可以非常自由的组织Hive中的表，只需要在创建表的时候告诉Hive数据中的列分隔符和行分隔符，Hive就可以解析数据。Hive中所有的数据都存储在HDFS中，存储结构主要包括数据库、文件、表和视图。Hive中包含以下数据模型：Table内部表，External Table外部表，Partition分区，Bucket桶。Hive默认可以直接加载文本文件，还支持sequence file 、RCFile。

Hive的数据模型介绍如下：

1) Hive数据库

类似传统数据库的 DataBase，例如 hive >create database test_database;

2) 内部表

Hive 的内部表与数据库中的表在概念上是类似。每一个 Table 在 Hive 中都有一个相应的目录存储数据。例如一个表 hive_test，它在 HDFS 中的路径为 /home/hdp_lbg_ectech/warehouse/hdp_lbg_ectech_bdw.db/hive_test，其中 /home/hdp_lbg_ectech/warehouse 是在 hive-site.xml 中由 ${hive.metastore.warehouse.dir}指定的数据仓库的目录，所有的 Table 数据（不包括外部表）都保存在这个目录中。删除表时，元数据与数据都会被删除。

建表语句示例：

CREATE EXTERNAL TABLE hdp_lbg_ectech_bdw.hive_test

(`userid` string COMMENT'')

ROW FORMAT DELIMITED FIELDS TERMINATED BY'\001';

load data inpath '/home/hdp_lbg_ectech/resultdata/test.txt' overwrite into table hive_test;

3) 外部表

外部表指向已经在 HDFS 中存在的数据，可以创建分区。它和内部表在元数据的组织上是相同的，而实际数据的存储则有较大的差异。内部表在加载数据的过程中，实际数据会被移动到数据仓库目录中。删除表时，表中的数据和元数据将会被同时删除。而外部表只有一个过程，加载数据和创建表同时完成（CREATE EXTERNAL TABLE ……LOCATION），实际数据是存储在 LOCATION 后面指定的 HDFS 路径中，并不会移动到数据仓库目录中。当删除一个外部表时，仅删除该表的元数据，而实际外部目录的数据不会被删除，推荐使用这种模式。

4) 分区

Partition 相当于数据库中的列的索引，但是 Hive 组织方式和数据库中的很不相同。在 Hive 中，表中的一个分区对应于表下的一个目录，所有的分区数据都存储在对应的目录中。

一般是按时间、地区、类目来分区，便于局部查询，避免扫描整个数据源。

5) 桶

Buckets 是将表的列通过 Hash 算法进一步分解成不同的文件存储。它对指定列计算 hash，根据 hash 值切分数据，目的是为了并行，每一个 Bucket 对应一个文件。例如将 userid 列分散至 32 个 bucket ,首先对 userid 列的值计算 hash ,对应 hash 值为 0 的 HDFS 目录为/home/hdp_lbg_ectech/resultdata/part-00000；hash 值为 20 的 HDFS 目录为 /home/hdp_lbg_ectech/resultdata/part-00020。

6) Hive 的视图

视图与传统数据库的视图类似。目前只有逻辑视图，没有物化视图；视图只能查询，不能 Load/Insert/Update/Delete 数据；视图在创建时候，只是保存了一份元数据，当查询视图的时候，才开始执行视图对应的那些子查询；

## 2. Hive 基础操作

1) DDL 操作：包括

- 建表，删除表
- 修改表结构
- 创建/删除视图
- 创建数据库和显示命令
- 增加分区，删除分区

- 重命名表
- 修改列的名字、类型、位置、注释
- 增加/更新列

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
  [(col_name data_type [COMMENT col_comment], ...)]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
  [CLUSTERED BY (col_name, col_name, ...)
  [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS]
  [ROW FORMAT row_format]
  [STORED AS file_format]
  [LOCATION hdfs_path]
```

- CREATE TABLE 创建一个指定名字的表。如果相同名字的表已经存在，则抛出异常；用户可以用 IF NOT EXIST 选项来忽略这个异常
- EXTERNAL 关键字可以让用户创建一个外部表，在建表的同时指定一个指向实际数据的路径（LOCATION）
- LIKE 允许用户复制现有的表结构，但是不复制数据
- COMMENT 可以为表与字段增加描述
- ROW FORMAT

  DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char]

  [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]

  | SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, ...)]

  用户在建表的时候可以自定义 SerDe 或者使用自带的 SerDe。如果没有指定 ROW FORMAT 或者 ROW FORMAT DELIMITED，将会使用自带的 SerDe。在建表的时候，用户还需要为表指定列，用户在指定表的列的同时也会指定自定义的

SerDe，Hive 通过 SerDe 确定表的具体的列的数据。

- STORED AS

  SEQUENCEFILE

  | TEXTFILE

  | RCFILE

  | INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

  如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 STORED AS SEQUENCE 。

**例子 1：**创建简单表

```
CREATE TABLE pokes (foo INT, bar STRING);
```

**例子 2：**创建外部表

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

**例子 3：**创建分区表

```
CREATE TABLE par_table(viewTime INT, userid BIGINT,
     page_url STRING, referrer_url STRING,
     ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(date STRING, pos STRING)
ROW FORMAT DELIMITED  '\t'
FIELDS TERMINATED BY '\n'
STORED AS SEQUENCEFILE;
```

**例子 4：**创建 Bucket 表

```
CREATE TABLE par_table(viewTime INT, userid BIGINT,
```

```
        page_url STRING, referrer_url STRING,
        ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(date STRING, pos STRING)
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
ROW FORMAT DELIMITED   '\t'
FIELDS TERMINATED BY '\n'
STORED AS SEQUENCEFILE;
```

**例子 5：**创建表并创建索引字段 ds

```
CREATE TABLE invites (foo INT, bar STRING) PARTITIONED BY (ds STRING);
```

**例子 6：**复制一个空表

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

**例子 7：**显示所有表

```
SHOW TABLES;
```

**例子 8：**按正则条件（正则表达式）显示表

```
SHOW TABLES '.*s';
```

**例子 9：**表添加一列

```
ALTER TABLE pokes ADD COLUMNS (new_col INT);
```

**例子 10：**添加一列并增加列字段注释

```
ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT 'a comment');
```

**例子 11：**更改表名

```
ALTER TABLE events RENAME TO 3koobecaf;
```

**例子 12：**删除列

```
DROP TABLE pokes;
```

**例子 13：**增加、删除分区

```
增加：
ALTER TABLE table_name ADD [IF NOT EXISTS] partition_spec [ LOCATION 'location1' ]
partition_spec [ LOCATION 'location2' ] ...
         partition_spec:
  : PARTITION (partition_col = partition_col_value, partition_col = partiton_col_value, ...)
删除：
      ALTER TABLE table_name DROP partition_spec, partition_spec,...
```

**例子 14：**重命名表

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

**例子 15：**修改列的名字、类型、位置、注释

```
ALTER TABLE table_name CHANGE [COLUMN] col_old_name col_new_name column_type
[COMMENT col_comment] [FIRST|AFTER column_name]
这个命令可以允许改变列名、数据类型、注释、列位置或者它们的任意组合
```

**例子 16：**创建 / 删除视图

```
CREATE VIEW [IF NOT EXISTS] view_name [ (column_name [COMMENT
column_comment], ...) ][COMMENT view_comment][TBLPROPERTIES (property_name =
property_value, ...)] AS SELECT
增加视图
如果没有提供表名，视图列的名字将由定义的 SELECT 表达式自动生成
如果修改基本表的属性，视图中不会体现，无效查询将会失败
视图是只读的，不能用 LOAD/INSERT/ALTER

DROP VIEW view_name
删除视图
```

**例子 17：**创建数据库

```
CREATE DATABASE name
```

**例子 18：**显示命令

```
show tables;
show databases;
show partitions ;
show functions
describe extended table_name dot col_name
```

2) DML 操作：元数据存储

hive 不支持用 insert 语句一条一条的进行插入操作，也不支持 update 操作。数据是以 load 的方式加载到建立好的表中。数据一旦导入就不可以修改。

DML 包括：

- INSERT 插入
- UPDATE 更新
- DELETE 删除
- 向数据表内加载文件

- 将查询结果插入到 Hive 表中

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

Load 操作只是单纯的复制/移动操作，将数据文件移动到 Hive 表对应的位置。
filepath
相对路径，例如：project/data1
绝对路径，例如： /user/hive/project/data1
包含模式的完整 URI，例如：hdfs://namenode:9000/user/hive/project/data1
```

**例子 1：**向数据表内加载文件

```
LOAD DATA LOCAL INPATH './examples/files/kv1.txt' OVERWRITE INTO TABLE pokes;
```

**例子 2：**加载本地数据，同时给定分区信息

```
LOAD DATA LOCAL INPATH './examples/files/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');
```

**例子 3：**加载本地数据，同时给定分区信息

```
LOAD DATA INPATH '/user/myname/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');
```

**例子 4：**将查询结果插入 Hive 表

```
基本模式：
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement

多插入模式：
FROM from_statement
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1
[INSERT OVERWRITE TABLE tablename2 [PARTITION ...] select_statement2] ...

自动分区模式：
INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select_statement FROM from_statement
```

**例子 3：**将查询结果写入 HDFS 文件系统

```
INSERT OVERWRITE [LOCAL] DIRECTORY directory1 SELECT ... FROM ...

FROM from_statement
INSERT OVERWRITE [LOCAL] DIRECTORY directory1 select_statement1
[INSERT OVERWRITE [LOCAL] DIRECTORY directory2 select_statement2]
```

```
数据写入文件系统时进行文本序列化，且每列用^A 来区分，\n 换行
```

**例子 3：**INSERT INTO

```
INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement
```

3) DQL 操作：数据查询 SQL

DQL 包括：

- 基本的 Select 操作
- 基于 Partition 的查询
- Join

基本 Select 操作：

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list [HAVING condition]]
[ CLUSTER BY col_list
| [DISTRIBUTE BY col_list] [SORT BY| ORDER BY col_list]
]
[LIMIT number]

● 使用 ALL 和 DISTINCT 选项区分对重复记录的处理。默认是 ALL，表示查询所有记录。
  DISTINCT 表示去掉重复的记录
● Where 条件
● 类似我们传统 SQL 的 where 条件
● 目前支持 AND,OR ,0.9 版本支持 between
● IN, NOT IN
● 不支持 EXIST ,NOT EXIST

ORDER BY 与 SORT BY 的不同
● ORDER BY 全局排序，只有一个 Reduce 任务
● SORT BY 只在本机做排序

Limit
● Limit 可以限制查询的记录数
```

**例子 1：**按先件查询

```
SELECT a.foo FROM invites a WHERE a.ds='<DATE>';
```

**例子 2：**将查询数据输出至目录

```
INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE a.ds='<DATE>';
```

**例子 3：**将查询结果输出至本地目录

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out' SELECT a.* FROM pokes a;
```

**例子 4：**选择所有列到本地目录

```
hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a;
INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a WHERE a.key < 100;
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/reg_3' SELECT a.* FROM events a;
INSERT OVERWRITE DIRECTORY '/tmp/reg_4' select a.invites, a.pokes FROM profiles a;
INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT COUNT(1) FROM invites a WHERE a.ds='<DATE>';
INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT a.foo, a.bar FROM invites a;
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/sum' SELECT SUM(a.pc) FROM pc1 a;
```

**例子 5：**将一个表的统计结果插入另一个表中

```
INSERT OVERWRITE TABLE events SELECT a.bar, count(1) FROM invites a WHERE a.foo > 0 GROUP BY a.bar;
FROM pokes t1 JOIN invites t2 ON (t1.bar = t2.bar) INSERT OVERWRITE TABLE events SELECT t1.bar, t1.foo, t2.foo;
```

**例子 6：**将多表数据插入到同一表中

```
FROM src
INSERT OVERWRITE TABLE dest1 SELECT src.* WHERE src.key < 100
INSERT OVERWRITE TABLE dest2 SELECT src.key, src.value WHERE src.key >= 100 and src.key < 200
INSERT OVERWRITE TABLE dest3 PARTITION(ds='2008-04-08', hr='12') SELECT src.key WHERE src.key >= 200 and src.key < 300
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/dest4.out' SELECT src.value WHERE src.key >= 300;
```

**例子 7：**将文件流直接插入文件

```
FROM invites a INSERT OVERWRITE TABLE events SELECT TRANSFORM(a.foo, a.bar) AS (oof, rab) USING '/bin/cat' WHERE a.ds > '2008-08-09';
```

# 第九章 流式计算解决方案-Storm

在 Hadoop 生态圈中，针对大数据进行批量计算时，通常需要一个或者多个 MapReduce 作业来完成，但这种批量计算方式是满足不了对实时性要求高的场景。

Storm 是一个开源分布式实时计算系统，它可以实时可靠地处理流数据。

本章内容：

1) Storm 特点
2) Storm 基本概念
3) Storm 分组模式
4) Storm 系统架构
5) Storm 容错机制
6) 一个简单的 Storm 实现

## 1. Storm 特点

在 Storm 出现之前，进行实时处理是非常痛苦的事情，我们主要的时间都花在关注往哪里发消息，从哪里接收消息，消息如何序列化，真正的业务逻辑只占了源代码的一小部分。一个应用程序的逻辑运行在很多 worker 上，但这些 worker 需要各自单独部署，还需要部署消息队列。最大问题是系统很脆弱，而且不是容错的：需要自己保证消息队列和 worker 进程工作正常。

Storm 完整地解决了这些问题。它是为分布式场景而生的，抽象了消息传递，会自动地在集群机器上并发地处理流式计算，让你专注于实时处理的业务逻辑。

Storm 有如下特点：

1) 编程简单：开发人员只需要关注应用逻辑，而且跟 Hadoop 类似，Storm 提供的编程原语也很简单

2) 高性能，低延迟：可以应用于广告搜索引擎这种要求对广告主的操作进行实时响应的场景。

3) 分布式：可以轻松应对数据量大，单机搞不定的场景

4) 可扩展：随着业务发展，数据量和计算量越来越大，系统可水平扩展

5) 容错：单个节点挂了不影响应用

6) 消息不丢失：保证消息处理

不过 Storm 不是一个完整的解决方案。使用 Storm 时你需要关注以下几点：

1) 如果使用的是自己的消息队列，需要加入消息队列做数据的来源和产出的代码

2) 需要考虑如何做故障处理：如何记录消息处理的进度，应对 Storm 重启，挂掉的场景

3) 需要考虑如何做消息的回退：如果某些消息处理一直失败怎么办？

## 2. Storm 与 Hadoop 区别

1) 定义及架构

Hadoop 是 Apache 的一个项目，是一个能够对大量数据进行分布式处理的软件框架。

Storm 是 Apache 基金会的孵化项目，是应用于流式数据实时处理领域的分布式计算系统。

| | Hadoop | Storm |
|---|---|---|
| 系统角色 | JobTracker | Nimbus |

| | | |
|---|---|---|
| | TaskTracker | Supervisor |
| | Child | Worker |
| 应用名称 | Job | Topology |
| 组件接口 | Mapper/Reducer | Spout/Bolt |

2) 应用方面

Hadoop 是分布式批处理计算，强调批处理，常用于数据挖掘和分析。

Storm 是分布式实时计算，强调实时性，常用于实时性要求较高的地方。

3) 计算处理方式

Hadoop 是磁盘级计算，进行计算时，数据在磁盘上，需要读写磁盘；Hadoop 应用 MapReduce 的思想，将数据切片计算来处理大量的离线数据。Hadoop 处理的数据必须是已经存放在 HDFS 上或者类似 HBase 的数据库中，所以 Hadoop 实现的时候是通过移动计算到这些存放数据的机器上来提高效率的。

Storm 是内存级计算，数据直接通过网络导入内存。Storm 是一个流计算框架，处理的数据是实时消息队列中的，需要写好一个 Topology 逻辑，然后将接收进来的数据进行处理，所以 Storm 是通过移动数据平均分配到机器资源来获得高效率的。

4) 数据处理方面

**数据来源：**Hadoop 是 HDFS 上某个文件夹下的数据，数据量可能以 TB 来计，而 Storm 则是实时新增的某一笔数据。

**处理过程：**Hadoop 是 Map 阶段到 Reduce 阶段的；Storm 是由用户定义处理流程，流程中可以包含多个步骤，每个步骤可以是数据源(SPOUT)，也可以是处理逻辑(BOLT)。

**是否结束：**Hadoop 最后必须要结束；而 Storm 没有结束状态，到最后一步时，就停在那，直到有新数据进入时再重新开始。

**处理速度：**Hadoop 以处理 HDFS 上大量数据为目的，速度慢；Storm 只要处理新增的某一笔数据即可，故此它的速度很快。

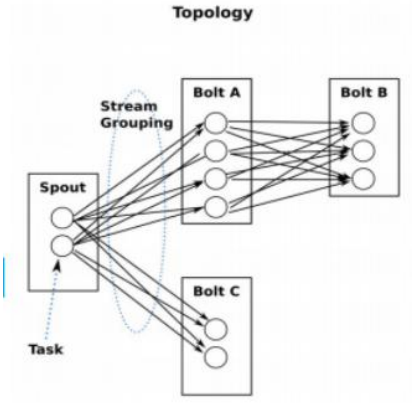
**适用场景：**Hadoop 主要是处理一批数据，对时效性要求不高，需要处理就提交一个 JOB；而 Storm 主要是处理某一新增数据的，故此时效性要求高。

总结，Hadoop 和 Storm 并没有真的优劣之分，它们只是在各自的领域上有着独特的性能而已，若是真的把它们进行单纯的比较，反而是有失公平了。事实上，只有在最合适的方面使用最合适的大数据平台，才能够真正体现出它们的价值，也才能够真正为我们的工作提供最为便捷的助力！

## 3. Storm 基本概念

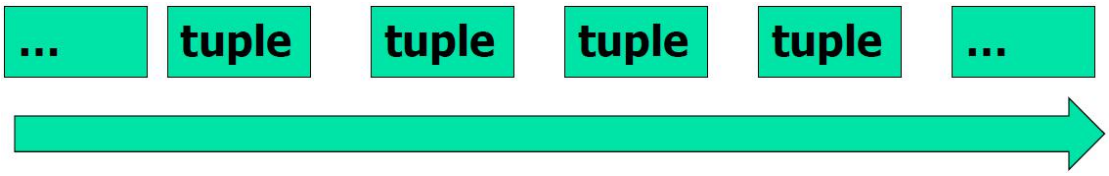
1) Topology

一个 Storm 拓扑打包了一个实时处理程序的逻辑。一个 Storm 拓扑跟一个 MapReduce 的任务(job)是类似的。主要区别是 MapReduce 任务最终会结束，而拓扑会一直运行（当然直到你杀死它)。一个拓扑是一个通过流分组(Stream Grouping)把 Spout 和 Bolt 连接到一起的拓扑结构。图的每条边代表一个 Bolt 订阅了其他 Spout 或者 Bolt 的输出流。一个拓扑就是一个复杂的多阶段的流计算。




2) Tuple

元组是 Storm 提供的一个轻量级的数据格式，可以用来包装你需要实际处理的数据。元组是一次消息传递的基本单元。一个元组是一个命名的值列表，其中的每个值都可以是任意类型的。元组是动态地进行类型转化的—字段的类型不需要事先声明。在 Storm 中编程时，就是在操作和转换由元组组成的流。通常，元组包含整数，字节，字符串，浮点数，布尔值和字节数组等类型。要想在元组中使用自定义类型，就需要实现自己的序列化方式。



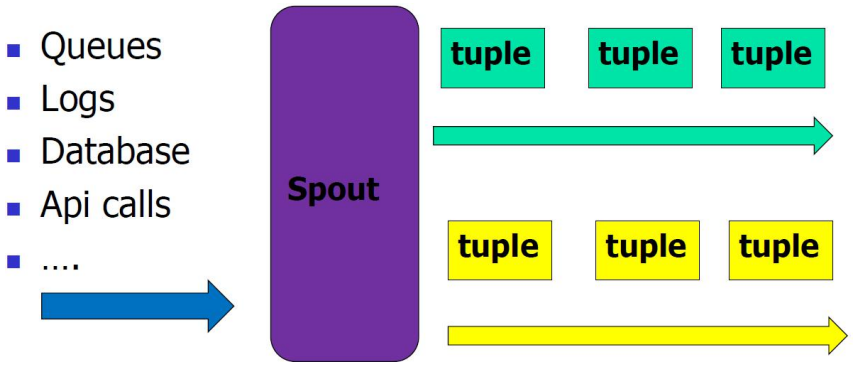

3) Stream

流是 Storm 中的核心抽象。一个流由无限的元组序列组成，这些元组会被分布式并行地创建和处理。通过流中元组包含的字段名称来定义这个流。

每个流声明时都被赋予了一个 ID。只有一个流的 Spout 和 Bolt 非常常见，所以 OutputFieldsDeclarer 提供了不需要指定 ID 来声明一个流的函数(Spout 和 Bolt 都需要声明输出的流)。这种情况下，流的 ID 是默认的“default”。

4) Spout

Spout(喷嘴，这个名字很形象)是 Storm 中流的来源。通常 Spout 从外部数据源，如消息队列中读取元组数据并吐到拓扑里。Spout 可以是可靠的(reliable)或者不可靠(unreliable)的。可靠的 Spout 能够在一个元组被 Storm 处理失败时重新进行处理，而非可靠的 Spout 只是吐数据到拓扑里，不关心处理成功还是失败了。



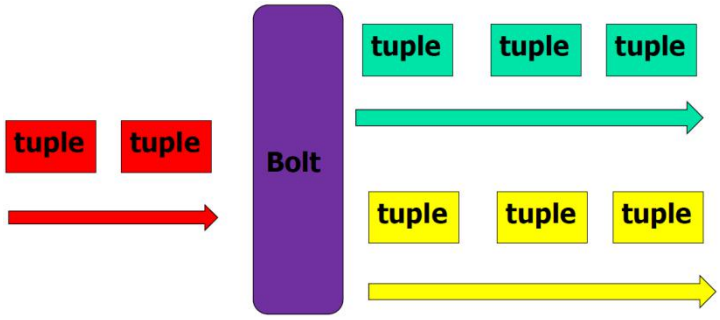

Spout 可以一次给多个流吐数据。此时需要通过 OutputFieldsDeclarer 的 declareStream 函数来声明多个流并在调用 SpoutOutputCollector 提供的 emit 方法时指定元组吐给哪个流。

Spout 中最主要的函数是 nextTuple，Storm 框架会不断调用它去做元组的轮询。如果没有新的元组过来，就直接返回，否则把新元组吐到拓扑里。nextTuple 必须是非阻塞的，因为 Storm 在同一个线程里执行 Spout 的函数。

Spout 中另外两个主要的函数是 Ack 和 fail。当 Storm 检测到一个从 Spout 吐出的元组在拓扑中成功处理完时调用 Ack,没有成功处理完时调用 Fail。只有可靠型的 Spout 会调用 Ack 和 Fail 函数。

5) Bolt

在拓扑中所有的计算逻辑都是在 Bolt 中实现的。一个 Bolt 可以处理任意数量的输入流，产生任意数量新的输出流。Bolt 可以做函数处理，过滤，流的合并，聚合，存储到数据库等操作。Bolt 就是流水线上的一个处理单元，把数据的计算处理过程合理的拆分到多个 Bolt、合理设置 Bolt 的 task 数量，能够提高 Bolt 的处理能力，提升流水线的并发度。




Bolt 可以给多个流吐出元组数据。此时需要使用 OutputFieldsDeclarer 的 declareStream 方法来声明多个流并在使用 [OutputColletor](https://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html)的 emit 方法时指定给哪个流吐数据。

当你声明了一个 Bolt 的输入流，也就订阅了另外一个组件的某个特定的输出流。如果希望订阅另一个组件的所有流，需要单独挨个订阅。InputDeclarer 有语法糖来订阅 ID 为默认值的流。例如 declarer.shuffleGrouping("redBolt")订阅了 redBolt 组件上的默认流，跟 declarer.shuffleGrouping("redBolt", DEFAULT_STREAM_ID)是相同的。

在 Bolt 中最主要的函数是 execute 函数，它使用一个新的元组当作输入。Bolt 使用 OutputCollector 对象来吐出新的元组。Bolts 必须为处理的每个元组调用 OutputCollector 的 ack 方法以便于 Storm 知道元组什么时候被各个 Bolt 处理完了（最终就可以确认 Spout 吐出的某个元组处理完了）。通常处理一个输入的元组时，会基于这个元组吐出零个或者多个元组，然后确认(ack)输入的元组处理完了，Storm 提供了 IBasicBolt 接口来自动完成确认。

必须注意 OutputCollector 不是线程安全的，所以所有的吐数据(emit)、确认(ack)、通知失败(fail)必须发生在同一个线程里。更多信息可以参照问题定位

6) Task

每个 Spout 和 Bolt 会以多个任务(Task)的形式在集群上运行。每个任务对应一个执行

线程，流分组定义了如何从一组任务(同一个 Bolt)发送元组到另外一组任务(另外一个 Bolt)上。可以在调用 TopologyBuilder 的 setSpout 和 setBolt 函数时设置每个 Spout 和 Bolt 的并发数。

7) Component

组件(component)是对 Bolt 和 Spout 的统称

8) Stream Grouping

定义拓扑的时候，一部分工作是指定每个 Bolt 应该消费哪些流。流分组定义了一个流在一个消费它的 Bolt 内的多个任务(task)之间如何分组。流分组跟计算机网络中的路由功能是类似的，决定了每个元组在拓扑中的处理路线。

在 Storm 中有七个内置的流分组策略，你也可以通过实现 CustomStreamGrouping 接口来自定义一个流分组策略:

**洗牌分组(Shuffle grouping):** 随机分配元组到 Bolt 的某个任务上，这样保证同一个 Bolt 的每个任务都能够得到相同数量的元组。

**字段分组(Fields grouping):** 按照指定的分组字段来进行流的分组。例如，流是用字段“user-id”来分组的，那有着相同“user-id”的元组就会分到同一个任务里，但是有不同“user-id”的元组就会分到不同的任务里。这是一种非常重要的分组方式，通过这种流分组方式，我们就可以做到让 Storm 产出的消息在这个”user-id”级别是严格有序的，这对一些对时序敏感的应用(例如，计费系统)是非常重要的。

**Partial Key grouping:** 跟字段分组一样，流也是用指定的分组字段进行分组的，但是在多个下游 Bolt 之间是有负载均衡的，这样当输入数据有倾斜时可以更好的利用资源。这篇论文很好的解释了这是如何工作的，有哪些优势。

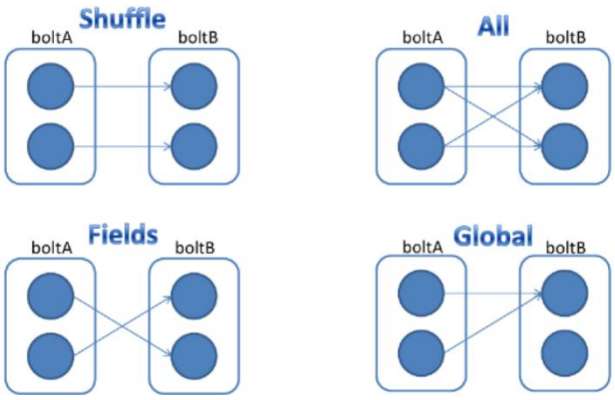
**All grouping:** 流会复制给 Bolt 的所有任务。小心使用这种分组方式。

**Global grouping:** 整个流会分配给 Bolt 的一个任务。具体一点，会分配给有最小 ID 的任务。

**不分组(None grouping):** 说明不关心流是如何分组的。目前，None grouping 等价于洗牌分组。

**Direct grouping :** 一种特殊的分组。对于这样分组的流，元组的生产者决定消费者的哪个任务会接收处理这个元组。只能在声明做直连的流(direct streams)上声明 Direct groupings 分组方式。只能通过使用 emitDirect 系列函数来吐元组给直连流。一个 Bolt 可以通过提供的 TopologyContext 来获得消费者的任务 ID ,也可以通过 OutputCollector 对象的 emit 函数(会返回元组被发送到的任务的 ID)来跟踪消费者的任务 ID。

Local or shuffle grouping：如果目标 Bolt 在同一个 worker 进程里有一个或多个任务，元组就会通过洗牌的方式分配到这些同一个进程内的任务里。否则，就跟普通的洗牌分组一样。




9) Reliability

Storm 保证了拓扑中 Spout 产生的每个元组都会被处理。Storm 是通过跟踪每个 Spout 所产生的所有元组构成的树形结构并得知这棵树何时被完整地处理来达到可靠性。每个拓扑对这些树形结构都有一个关联的“消息超时”。如果在这个超时时间里 Storm 检测到 Spout 产生的一个元组没有被成功处理完，那 Spout 的这个元组就处理失败了，后续会重新处理

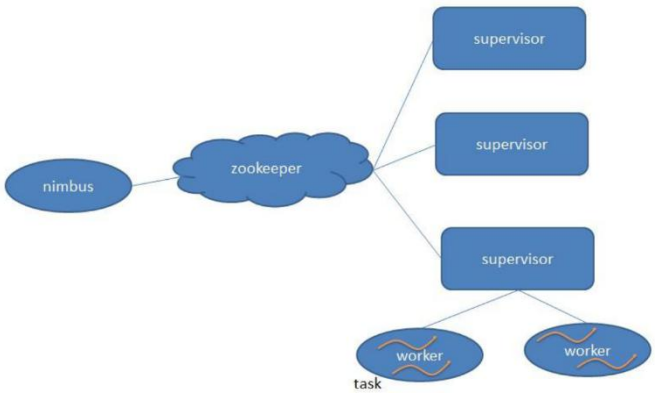
一遍。

为了发挥 Storm 的可靠性，需要你在创建一个元组树中的一条边时告诉 Storm，也需要在处理完每个元组之后告诉 Storm。这些都是通过 Bolt 吐元组数据用的 OutputCollector 对象来完成的。标记是在 emit 函数里完成，完成一个元组后需要使用 Ack 函数来告诉 Storm。

10) Workers

拓扑以一个或多个 Worker 进程的方式运行。每个 Worker 进程是一个物理的 Java 虚拟机，执行拓扑的一部分任务。例如，如果拓扑的并发设置成了 300，分配了 50 个 Worker，那么每个 Worker 执行 6 个任务(作为 Worker 内部的线程)。Storm 会尽量把所有的任务均分到所有的 Worker 上。

## 4. Storm 系统架构




1) 主节点（Nimbus）:

在分布式系统中，调度服务非常重要，它的设计，会直接关系到系统的运行效率，错误恢复(fail over),故障检测(error detection)和水平扩展(scale)的能力。

集群上任务(task)的调度由一个 Master 节点来负责。这台机器上运行的 Nimbus 进程负责任务的调度。另外一个进程是 Storm UI，可以界面上查看集群和所有的拓扑的运行状态。

2) 从节点（Supervisor）

Storm 集群上有多个从节点，他们从 Nimbus 上下载拓扑的代码，然后去真正执行。Slave 上的 Supervisor 进程是用来监督和管理实际运行业务代码的进程。在 Storm 0.9 之后，又多了一个进程 Logviewer,可以用 Storm UI 来查看 Slave 节点上的 log 文件。

3) 协调服务 Zookeeper：

ZooKeeper 在 Storm 上不是用来做消息传输用的，而是用来提供协调服务(coordination service)，同时存储拓扑的状态和统计数据。

- Supervisor，Nimbus 和 worker 都在 ZooKeeper 留下约定好的信息。例如 Supervisor 启动时，会在 ZooKeeper 上注册，Nimbus 就可以发现 Supervisor；Supervisor 在 ZooKeeper 上留下心跳信息，Nimbus 通过这些心跳信息来对 Supervisor 进行健康检测，检测出坏节点
- 由于 Storm 组件(component)的状态信息存储在 ZooKeeper 上，所以 Storm 组件就可以无状态，可以 kill -9 来杀死

  例如：Supervisors/Nimbus 的重启不影响正在运行中的拓扑，因为状态都在 ZooKeeper 上，从 ZooKeeper 上重新加载一下就好了
- 用来做心跳

  Worker 通过 ZooKeeper 把孩子 executor 的情况以心跳的形式汇报给 Nimbus

  Supervisor 进程通过 ZK 把自己的状态也以心跳的形式汇报给 Nimbua
- 存储最近任务的错误情况(拓扑停止时会删除)

4) 进程 Worker

运行具体处理组件逻辑的进程，一个 Topology 可能会在一个或者多个 worker 里面执行，每个 worker 是一个物理 JVM 并且执行整个 Topology 的一部分

例如：对于并行度是 300 的 topology 来说，如果我们使用 50 个工作进程来执行，那么每个工作进程会处理其中的 6 个 tasks，Storm 会尽量均匀的工作分配给所有的 worker

5) Task

Worker 中的每一个 spout/bolt 的线程称为一个 task，每一个 spout 和 bolt 会被当作很多 task 在整个集群里执行，每一个 executor 对应到一个线程，在这个线程上运行多个 task，Stream Grouping 则是定义怎么从一堆 task 发射 tuple 到另外一堆 task，可以调用 TopologyBuilder 类的 setSpout 和 setBolt 来设置并行度（也就是有多少个 task）

## 5. Storm 容错机制

Storm 的容错机制包括架构容错和数据容错。

1) 架构容错：

Nimbus 和 Supervisor 进程被设计成快速失败(fail fast)的(当遇到异常的情况 ,进程就会挂掉)并且是无状态的(状态都保存在 Zookeeper 或者在磁盘上)。

最重要的是，worker 进程不会因为 Nimbus 或者 Supervisor 挂掉而受影响。这跟 Hadoop 是不一样的，当 JobTracker 挂掉，所有的任务都会没了。

**当 Nimbus 挂掉会怎样？**

如果 Nimbus 是以推荐的方式处于进程监管(例如通过 supervisord)之下，那它会被重启，不会有任何影响。

否则当 Nimbus 挂掉后：

- 已经存在的拓扑可以继续正常运行，但是不能提交新拓扑
- 正在运行的 worker 进程仍然可以继续工作。而且当 worker 挂掉，supervisor 会

一直重启 worker。

- 失败的任务不会被分配到其他机器(是 Nimbus 的职责)上了

**当一个 Supervisor(slave 节点)挂掉会怎样？**

如果 Supervisor 是以推荐的方式处于进程监管（例如通过(supervisord)[supervisord.org/])之下，那它会被重启，不会有任何影响

否则当 Supervisor 挂掉：分配到这台机器的所有任务(task)会超时，Nimbus 会把这些任务(task)重新分配给其他机器。

**当一个 worker 挂掉会怎么样？**

当一个 worker 挂掉，supervisor 会重启它。如果启动一直失败那么此时 worker 也就不能和 Nimbus 保持心跳了，Nimbus 会重新分配 worker 到其他机器。

**Nimbus 算是一个单点故障吗？**

如果 Nimbus 节点挂掉，worker 进程仍然可以继续工作。而且当 worker 挂掉，supervisor 会一直重启 worker。但是，没有了 Nimbus，当需要的时候(如果 worker 机器挂掉了)worker 就不能被重新分配到其他机器了。

所以答案是，Nimbus 在“某种程度”上属于单点故障的。在实际中，这种情况没什么大不了的，因为当 Nimbus 进程挂掉，不会有灾难性的事情发生

2) 数据容错：

Storm 中的每一个 Topology 中都包含有一个 Acker 组件。 Acker 组件的任务就是跟踪从某个 task 中的 Spout 流出的每一个 messageId 所绑定的 Tuple 树中的所有 Tuple 的处理情况。如果在用户设置的最大超时时间（timetout 可以通过 Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS 来指定）内这些 Tuple 没有被完全处理，那么 Acker 会告诉 Spout 该消息处理失败，相反则会告知 Spout 该消息处理成功,它会分

别调用 Spout 中的 fail 和 ack 方法。

## 6. 一个简单的 Storm 实现

实现一个拓扑包括一个 spout 和两个 bolt。Spout 发送单词。每个 bolt 在输入数据的尾部追加字符串“!!!”。三个节点排成一条线：spout 发射给首个 bolt，然后，这个 bolt 再发射给第二个 bolt。如果 spout 发射元组“bob”和“john”，然后，第二个 bolt 将发射元组“bob!!!!!!”和“john!!!!!!”。

1) 其中 Topology 代码如下，定义整个网络拓扑图：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
                        .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
                    .shuffleGrouping("exclaim1");
```

2) Spout 实现：

```
public void nextTuple() {
        Utils.sleep(100);
        final String[] words = new String[] {"nathan", "mike", "jackson",
"golda", "bertels"};
        final Random rand = new Random();
        final String word = words[rand.nextInt(words.length)];
        _collector.emit(new Values(word));
}
```

3) Bolt 实现：

```
public static class ExclamationBolt implements IRichBolt {
        OutputCollector _collector;
        public void prepare(Map conf, TopologyContext context, OutputCollector
collector) {
                _collector = collector;
        }
        public void execute(Tuple tuple) {
                _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
                _collector.ack(tuple);
        }
```

```
        public void cleanup() {
        }
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields("word"));
        }
}
```

# 7. Storm 常用配置

1) Config.TOPOLOGY_WORKERS:

这个设置用多少个工作进程来执行这个 topology。比如，如果你把它设置成 25， 那么集群里面一共会有 25 个 java 进程来执行这个 topology 的所有 task。如果你的这个 topology 里面所有组件加起来一共有 150 的并行度 ,那么每个进程里面会有 6 个线程(150 / 25 = 6)。

2) Config.TOPOLOGY_ACKERS:

这个配置设置 acker 任务的并行度。默认的 acker 任务并行度为 1，当系统中有大量的消息时，应该适当提高 acker 任务的并发度。设置为 0，通过此方法，当 Spout 发送一个消息的时候，它的 ack 方法将立刻被调用；

3) Config.TOPOLOGY_MAX_SPOUT_PENDING:

这个设置一个 spout task 上面最多有多少个没有处理的 tuple（没有 ack/failed）回复， 我们推荐你设置这个配置，以防止 tuple 队列爆掉。

4) Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS:

这个配置 storm 的 tuple 的超时时间 – 超过这个时间的 tuple 被认为处理失败了。这个设置的默认设置是 30 秒

# 第十章 数据挖掘——推荐系统

大数据可以认为是许多数据的聚合，数据挖掘是把这些数据的价值发掘出来，比如有过去 10 年的气象数据，通过数据挖掘，几乎可以预测明天的天气是怎么样的，有较大概率是正确的。

机器学习是人工智能的核心，对大数据进行发掘，靠人工肯定是做不来的，那就得靠机器代替人工得到一个有效模型，通过该模型将大数据中的价值体现出来。

本章内容：

1) 数据挖掘和机器学习概念

2) 一个机器学习应用方向——推荐系统

3) 推荐算法——基于内容的推荐方法

4) 推荐算法——基于协同过滤的推荐方法

5) 基于 MapReduce 的协同过滤算法的实现

## 1. 数据挖掘和机器学习概念

机器学习和数据挖掘技术已经开始在多媒体、计算机图形学、计算机网络乃至操作系统、软件工程等计算机科学的众多领域中发挥作用，特别是在计算机视 觉和自然语言处理领域，机器学习和数据挖掘已经成为最流行、最热门的技术，以至于在这些领域的顶级会议上相当多的论文都与机器学习和数据挖掘技术有关。总的来看，引入机器学习和数据挖掘技术在计算机科学的众多分支领域中都是一个重要趋势。

对于数据挖掘，数据库提供数据管理技术，机器学习提供数据分析技术。通常我们要处理的大数据通过 HDFS 云存储平台来进行数据管理，目前 Hadoop 生态圈已经发展成熟，各种工具和接口基本满足大多数数据管理的需要。面对这样庞大的数据资源，需要有一种方

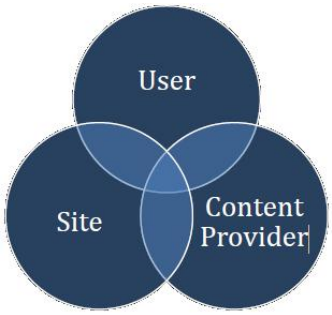
法需要让其中的价值体现出来，机器学习提供了一系列的分析挖掘数据的方法。

Hadoop 生态圈中有一个机器学习开源库的项目——Mahout，提供了丰富的可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序，Mahout 包含许多实现，包括聚类、分类、推荐过滤、频繁子项挖掘。

## 2. 一个机器学习应用方向——推荐领域

推荐算法是最为大众所知的一种机器学习模型。推荐是很多网站背后的核心组件之一，有时也是一个重要的收入来源。

一般来讲，推荐系统试图对用户与某类物品之间的联系建模。比如我们利用推荐系统来告诉用户有哪些电影他们会可能喜欢。如果这一点做的很好的话，就能够吸引更多的用户持续使用我们的服务。这对双方都有好处。同样，如果能准确告诉用户有哪些电影与某一个电影相似，就能方便用户在站点上找到更多感兴趣的信息。这也能提升用户的体验、参与度以及站点内容对用户的吸引力。对于大型网站来说，很多内容是来自于独立的第三方——内容提供商，比如淘宝的商品宝贝基本来自各个店铺、奇艺上的电影很多来自与专业的传媒集团和工作室、微信上制作精良的广告也是来自于各个行业的广告主。




建立一个良好的推荐生态圈，对于用户、网站平台以及内容提供商，都是有好处的，首先用户得到他们想要的物品，平台获得更多的流量和收入，内容提供商售卖其物品的效率也会提高，所以是一个三者共赢的一个场景，所以一个好的推荐系统会带来很大的价值。

## 3. 推荐算法——基于内容的推荐方法

基于内容的推荐（Content Based）应该算最早被使用的推荐方法，它根据用户过去喜欢的产品（本文统称为 item），为用户推荐和他过去喜欢的产品相似的产品。例如，一个推荐饭店的系统可以依据某个用户之前喜欢很多的烤肉店而为他推荐烤肉店。CB 最早主要是应用在信息检索系统当中，所以很多信息检索及信息过滤里的方法都能用于 CB 中。

CB 的过程一般包括以下三步：

1) Item Representation：为每个 item 抽取出一些特征（也就是 item 的 content 了）来表示此 item；
2) Profile Learning：利用一个用户过去喜欢（及不喜欢）的 item 的特征数据，来学习出此用户的喜好特征（profile）；
3) Recommendation Generation：通过比较上一步得到的用户 profile 与候选 item 的特征，为此用户推荐一组相关性最大的 item。

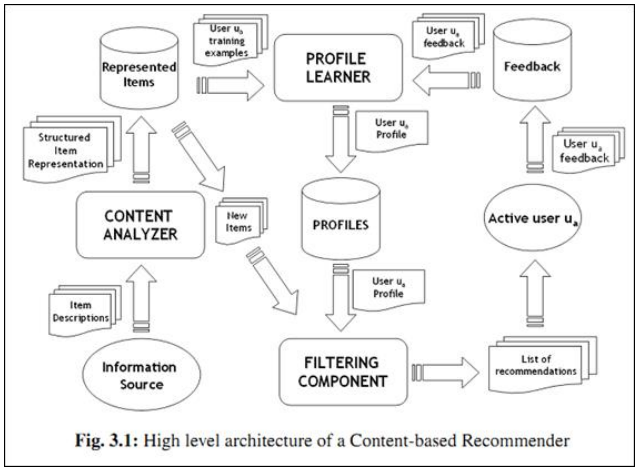


Fig. 3.1: High level architecture of a Content-based Recommender

举个例子说明前面的三个步骤。对于个性化阅读来说，一个 item 就是一篇文章。根据上面的第一步，我们首先要从文章内容中抽取出代表它们的属性。常用的方法就是利用出现在一篇文章中词来代表这篇文章，而每个词对应的权重往往使用信息检索中的 tf-idf 来计算。比如对于本文来说，词“CB”、“推荐”和“喜好”的权重会比较大，而“烤肉”这个词的

权重会比较低。利用这种方法，一篇抽象的文章就可以使用具体的一个向量来表示了。第二步就是根据用户过去喜欢什么文章来产生刻画此用户喜好的 profile 了，最简单的方法可以把用户所有喜欢的文章对应的向量的平均值作为此用户的 profile。比如某个用户经常关注与推荐系统有关的文章，那么他的 profile 中“CB”、“CF”和“推荐”对应的权重值就会较高。在获得了一个用户的 profile 后，CB 就可以利用所有 item 与此用户 profile 的相关度对他进行推荐文章了。一个常用的相关度计算方法是 cosine。最终把候选 item 里与此用户最相关（cosine 值最大）的 N 个 item 作为推荐返回给此用户。

接下来我们详细介绍下上面的三个步骤。

1) Item Representation：

真实应用中的 item 往往都会有一些可以描述它的属性。这些属性通常可以分为两种：结构化的（structured）属性与非结构化的（unstructured）属性。所谓结构化的属性就是这个属性的意义比较明确，其取值限定在某个范围；而非结构化的属性往往其意义不太明确，取值也没什么限制，不好直接使用。比如在交友网站上，item 就是人，一个 item 会有结构化属性如身高、学历、籍贯等，也会有非结构化属性（如 item 自己写的交友宣言，博客内容等等）。对于结构化数据，我们自然可以拿来就用；但对于非结构化数据（如文章），我们往往要先把它转化为结构化数据后才能在模型里加以使用。真实场景中碰到最多的非结构化数据可能就是文章了（如个性化阅读中）。下面我们就详细介绍下如何把非结构化的一篇文章结构化。

如何代表一篇文章在信息检索中已经被研究了很多年了，下面介绍的表示技术其来源也是信息检索，其名称为向量空间模型（Vector Space Model，简称 VSM）。

记我们要表示的所有文章集合为 $D = \{d_1, d_2, ..., d_N\}$，而所有文章中出现的词（对于中文文章，首先得对所有文章进行分词）的集合（也称为词典）为 $T = \{t_1, t_2, ..., t_n\}$。也就是说，我们有 N 篇要处理的文章，而这些文章里包含了 n 个不同的词。我们最终要使用一个向量来表示一篇文章，比如第 j 篇文章被表示为 $d_j = (w_{1j}, w_{2j}, ..., w_{nj})$，其中 $w_{1j}$ 表示第 1 个词 $t_1$ 在文章 j 中的权重，值越大表示越重要；$d_j$ 中其他向量的解释类似。所以，为了表示第 j 篇文章，现在关键的就是如何计算 $d_j$ 各分量的值了。例如，我们可以选取 $w_{1j}$ 为 1，如果词 $t_1$ 出现在第 j 篇文章中；选取为 0，如果 $t_1$ 未出现在第 j 篇文章中。我们也可以选取 $w_{1j}$ 为词 $t_1$ 出现在第 j 篇文章中的次数（frequency）。但是用的最多的计算方法还是信息检索中常用的词频-逆文档频率（term frequency–inverse document frequency，简称 tf-idf）。第 j 篇文章中与词典里第 k 个词对应的 tf-idf 为：

$$\text{TF-IDF}(t_k, d_j) = \underbrace{\text{TF}(t_k, d_j)}_{\text{TF}} \cdot \underbrace{log\frac{N}{n_k}}_{\text{IDF}}$$

其中 $\text{TF}(t_k, d_j)$ 是第 k 个词在文章 j 中出现的次数，而 $n_k$ 是所有文章中包括第 k 个词的文章数量。

最终第 k 个词在文章 j 中的权重由下面的公式获得：

$$w_{k,j} = \frac{\text{TF-IDF}(t_k, d_j)}{\sqrt{\sum_{s=1}^{|T|} \text{TF-IDF}(t_s, d_j)^2}}$$

做归一化的好处是不同文章之间的表示向量被归一到一个量级上，便于下面步骤的操作。

2) Profile Learning

假设用户 u 已经对一些 item 给出了他的喜好判断，喜欢其中的一部分 item，不喜欢其中的另一部分。那么，这一步要做的就是通过用户 u 过去的这些喜好判断，为他产生一

个模型。有了这个模型，我们就可以根据此模型来判断用户 u 是否会喜欢一个新的 item。所以，我们要解决的是一个典型的有监督分类问题，理论上机器学习里的分类算法都可以照搬进这里。

下面我们简单介绍下 CB 里常用的学习算法——KNN：

对于一个新的 item，最近邻方法首先找用户 u 已经评判过并与此新 item 最相似的 k 个 item，然后依据用户 u 对这 k 个 item 的喜好程度来判断其对此新 item 的喜好程度。这种做法和 CF 中的 item-based kNN 很相似，差别在于这里的 item 相似度是根据 item 的属性向量计算得到，而 CF 中是根据所有用户对 item 的评分计算得到。

对于这个方法，比较关键的可能就是如何通过 item 的属性向量计算 item 之间的两两相似度。[2]中建议对于结构化数据，相似度计算使用欧几里得距离；而如果使用向量空间模型（VSM）来表示 item 的话，则相似度计算可以使用 cosine。

3) Recommendation Generation

通过上一步的学习，会得到一个推荐列表，我们直接把这个列表中与用户属性最相关的 n 个 item 作为推荐返回给用户即可。

## 4. 推荐算法——基于协同过滤的推荐方法

俗话说“物以类聚、人以群分”，继续拿看电影这个例子来说，如果你喜欢《蝙蝠侠》、《碟中谍》、《星际穿越》、《源代码》等电影，另外有个人也都喜欢这些电影，而且他还喜欢《钢铁侠》，则很有可能你也喜欢《钢铁侠》这部电影。

所以说，当一个用户 A 需要个性化推荐时，可以先找到和他兴趣相似的用户群体 G，然后把 G 喜欢的、并且 A 没有听说过的物品推荐给 A，这就是基于用户的系统过滤算法。

根据上述基本原理，我们可以将基于用户的协同过滤推荐算法拆分为两个步骤：

1) 发现兴趣相似的用户

通常用 Jaccard 公式或者余弦相似度计算两个用户之间的相似度。设 N(u) 为用户 u 喜欢的物品集合，N(v) 为用户 v 喜欢的物品集合，那么 u 和 v 的相似度是多少呢：

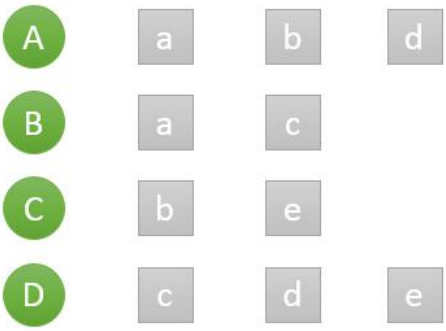
Jaccard 公式：

$$w_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

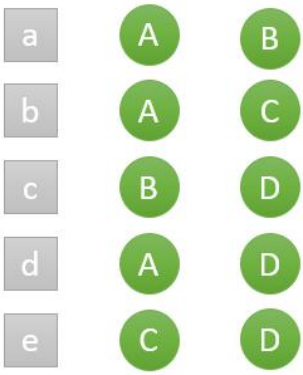
余弦相似度：

$$w_{uv} = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)| \times |N(v)|}}$$

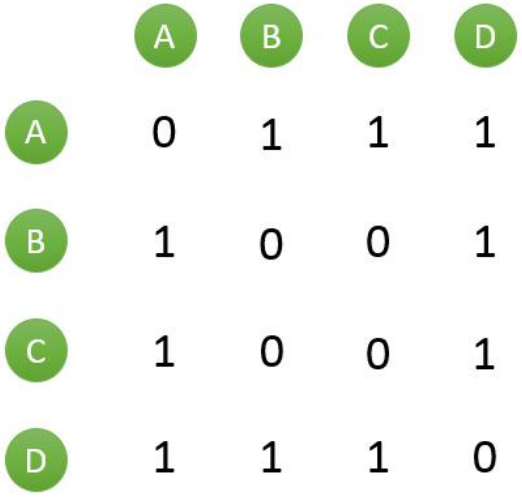
假设目前共有 4 个用户： A、B、C、D；共有 5 个物品：a、b、c、d、e。用户与物品的关系（用户喜欢物品）如下图所示：




如何一下子计算所有用户之间的相似度呢？为计算方便，通常首先需要建立“物品—用户”的倒排表，如下图所示：




然后对于每个物品，喜欢他的用户，两两之间相同物品加 1。例如喜欢物品 a 的用户有 A 和 B，那么在矩阵中他们两两加 1。如下图所示：




计算用户两两之间的相似度，上面的矩阵仅仅代表的是公式的分子部分。以余弦相似度为例，对上图进行进一步计算：

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | $\frac{1}{\sqrt{3\times2}}$ | $\frac{1}{\sqrt{3\times2}}$ | $\frac{1}{\sqrt{3\times3}}$ |
| B | $\frac{1}{\sqrt{3\times2}}$ | 0 | 0 | $\frac{1}{\sqrt{3\times2}}$ |
| C | $\frac{1}{\sqrt{3\times2}}$ | 0 | 0 | $\frac{1}{\sqrt{3\times2}}$ |
| D | $\frac{1}{\sqrt{3\times3}}$ | $\frac{1}{\sqrt{3\times2}}$ | $\frac{1}{\sqrt{3\times2}}$ | 0 |

到此，计算用户相似度就大功告成，可以很直观的找到与目标用户兴趣较相似的用户。

2) 推荐物品

首先需要从矩阵中找出与目标用户 u 最相似的 K 个用户，用集合 S(u, K) 表示，将 S 中用户喜欢的物品全部提取出来，并去除 u 已经喜欢的物品。对于每个候选物品 i ，用户 u 对它感兴趣的程度用如下公式计算：

$$p(u,i)=\sum_{v\in S(u,K)\cap N(i)} w_{uv}\times r_{vi}$$

其中 rvi 表示用户 v 对 i 的喜欢程度，在本例中都是为 1，在一些需要用户给予评分的推荐系统中，则要代入用户评分。

举个例子，假设我们要给 A 推荐物品，选取 K = 3 个相似用户，相似用户则是：B、C、D，那么他们喜欢过并且 A 没有喜欢过的物品有：c、e，那么分别计算 p(A, c) 和 p(A, e)：

$$p(A,c)=w_{AB}+w_{AD}=\frac{1}{\sqrt{6}}+\frac{1}{\sqrt{9}}=0.7416$$

$$p(A,e)=w_{AC}+w_{AD}=\frac{1}{\sqrt{6}}+\frac{1}{\sqrt{9}}=0.7416$$

看样子用户 A 对 c 和 e 的喜欢程度可能是一样的，在真实的推荐系统中，只要按得分排序，取前几个物品就可以了。