**第十一章**

**P2**

单机模式：Hadoop安装时的默认模式，不对配置文件进行修改，使用本地文件系统，Hadoop不启动NameNode、DataNode、JobTracker、TaskTracker等守护进程，这是一种用来对MapReduce程序进行查错和调试的模式。

虚拟分布模式：在一台机器上用软件模拟多节点集群，每个守护进程（daemon）都以Java进程形式运行。与单机模式比较增加了代码调试功能，允许检查内存使用情况和读写HDFS文件系统。需修改3个配置文件：core-site.xml、hdfs-site.xml、mapred-site.xml并格式化文件系统。

完全分布模式：Hadoop安装运行在多台主机上，构成一个真实的Hadoop集群，在所有的节点上安装JDK和Hadoop，相互通过高速局域网连接。各节点间设置SSH免密码登录，将各个从节点生成的公钥添加到主节点的信任列表。需要修改3个配置文件：core-site.xml、hdfs-site.xml、mapred-site.xml并格式化文件系统。

**P3**

硬件配置：

Hadoop集群内的计算节点类型实际只有两类：NameNode（执行作业调度、资源调配、系统监控等任务）和DataNode（承担具体的数据计算任务），因此节点机器的选型不宜超过两种

Hadoop实际生产系统可根据项目需要进行灵活的硬件系统配置，比如NameNode可以配置2台（另一台称为Secondary NameNode，与Active NameNode保持同步，随时可以进行切换）。

小型集群中NameNode与JobTracker两个程序部署在同一台机器上，但在大型集群中，NameNode与JobTracker则部署在不同机器上以提高运行效率。

实际应用时Hadoop集群的机器数可随着需要增长，这种动态可扩展性正是Hadoop平台的优势之一。

网络配置：

常规的Hadoop集群包含两层网络结构：NameNode到机架（Rack）的网络连接，以及机架内部的DataNode之间的网络连接。每个机架（Rack）内有30-40个DataNode服务器，配置一个1GB的交换机，并向上传输到一个核心交换机或者路由器（1GB或以上）。相同机架内节点间的带宽总和要大于不同机架间的带宽总和。

**P5**

Hadoop的数据存储系统包括：

分布式文件系统HDFS（Hadoop Distributed File System）

分布式非关系型数据库Hbase

数据仓库及数据分析工具Hive和Pig

用于数据采集、转移和汇总的工具Sqoop和Flume。

HDFS文件系统构成了Hadoop数据存储体系的基础，除HDFS外，在Hadoop平台上支持的文件系统还包括：FTP（File Transfer Protocol）文件系统[16]，用于远程文件传输；Amazon S3（Simple Storage Service）文件系统[17]（主要用于Amazon的弹性计算云架构）；微软的Windows Azure Storage Blobs (WASB)文件系统[18]，这是微软开发的在HDFS之上的数据读写层，用于微软的Azure云存储系统的数据读写。

Hadoop的资源调度管理工具包括：

提供分布式协调服务管理的Zookeeper，

负责作业调度的Oozie，

提供集群配置、管理和监控功能的Ambari、

大型集群监控系统Chukwa、

新的集群资源调度管理系统YARN。

Hadoop提供的计算引擎或计算模型包括：

离线批处理MapReduce、

图并行计算框架Hama[23]和Giraph[24]、

流计算Storm[25]、

内存计算Spark[26]、

交互式计算Drill[27]、

基于YARN的有向无环图（DAG）计算框架Tez[28]。

另外Hadoop还提供一系列计算分析工具，如支持数据挖掘与机器学习的Mahout[29]、用于节点间RPC通信支持多语言数据序列化（data serialization）框架Avro[30]、数据可视化分析工具Hue[31]等，上述系统或工具大多为Apache的独立开源项目。

**P11**

1）命名空间的限制：由于管理命名空间的名称节点进程是保存在内存中，因此名称节点能够容纳的对象（文件、块）的个数会受到内存空间大小的限制；

2）性能的瓶颈：整个分布式文件系统的吞吐量，受限于单个名称节点的吞吐量；

3）单点失效（SPOF）问题：一旦这个唯一的名称节点发生故障，会导致整个集群变得不可用

**P13**

有利于大规模文件存储：一个大规模文件可以被分拆成若干个文件块，不同的文件块可以被分发到不同的节点上，因此，一个文件的大小不会受到单个节点存储容量的限制，文件规模可以远远大于网络中任意节点的存储容量

适合数据备份：每个文件块都可以冗余存储到多个节点上，大大提高了系统的容错性和可用性

系统设计简化：首先简化了存储管理，因为文件块大小是固定的，这样就可以很容易计算出一个节点可以存储多少文件块；其次方便了元数据的管理，元数据不需要和文件块一起存储，可以由其他系统负责管理元数据

**P15**

FsImage文件：包含所有目录和文件inode的序列化形式，每个inode是一个文件或目录的元数据的内部表示

文件元数据：包含诸如文件复制等级、修改和访问时间、访问权限、块大小以及组成文件的块等信息

目录元数据：则包含修改时间、权限和配额元数据等。

FsImage文件并没有把文件 -> block -> 节点的映射表静态存储在某个节点，而是由名称节点进程把这个映射关系表装载并保留在内存中。当一个数据节点加入HDFS集群时，数据节点会把自己所包含的块列表通知给名称节点，由后者对内存的映射表进行更新，以确保名称节点掌握的块映射表是最新的。

**P16**

1）Roll edits：SecondaryNameNode进程会定期和NameNode通信，请求其停止使用EditLog文件，暂时将新的写操作写到一个新的文件edit.new上来，这个操作是瞬间完成，上层写日志的函数完全感觉不到差别；

2）Retrieve FsImage and edits from NameNode: SecondaryNameNode进程通过HTTP GET方式从NameNode取得FsImage和EditLog文件，并下载到本地相应目录下；

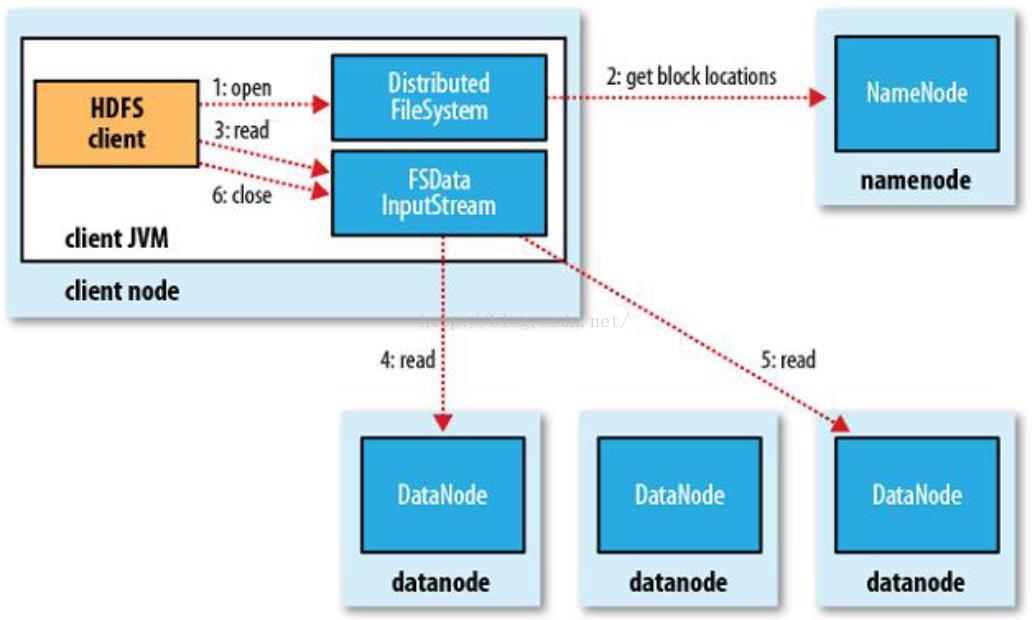
3）Merge: SecondaryNameNode进程将下载的FsImage载入到内存，然后一条一条地执行EditLog文件中的各项更新操作，使得内存中的FsImage保持最新，这个过程就是EditLog和FsImage文件的合并；

4）Transfer checkpoint to NameNode: SecondaryNameNode执行完3）操作之后，会通过post方式将新的FsImage文件发送到NameNode节点上；

5） Roll again: NameNode将从SecondaryNameNode接收到的新的FsImage替换旧的FsImage文件，同时将edit.new替换原来的EditLog文件，这个过程使得EditLog文件重新变小。

**P18**

**HDFS读文件流程**



1）打开文件（open file）：Java客户端通过FileSystem打开文件，执行代码如下：

import org.apache.hadoop.fs.FileSystem

Configuration conf = new Configuration();

FileSystem fs = FileSystem.get(conf);

FSDataInputStream in = fs.open(new Path(uri));

2）获取块信息（get block location）：通过ClientProtocal.getBlockLocations()远程调用名称节点，获得文件开始部分数据块的位置。名称节点还返回保存该数据块的所有数据节点的地址，并根据距离客户端远近进行排序；

3）读取请求（read request）：客户端获得输入流FSDataInputStream以后调用read()函数开始读取数据；输入流根据前面的排序结果，选择距离客户端最近的数据节点建立连接并读取数据：

FileSystem fs = FileSystem.get(conf);

FSDataInputStream in = fs.open(new Path(uri));

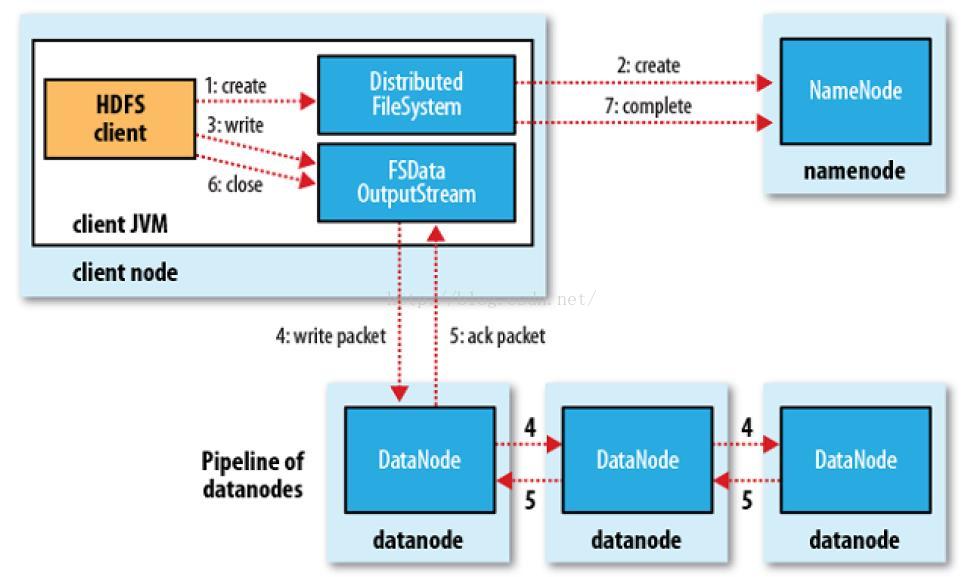
4）读取数据（read data）：从数据节点把数据块读到客户端，当该数据块读取完毕时FSDataInputStream关闭和该数据节点的连接；

5）读取下一个数据节点上的数据块。。。直到该文件的所有数据块读取完毕；

6）关闭文件（close file）：客户端通过FileSystem关闭文件，整个读取文件流程完成。

**P18**

**HDFS写文件流程**



1）创建文件（create file）：客户端通过FileSystem创建文件，执行代码如下：

import org.apache.hadoop.fs.FileSystem

Configuration conf = new Configuration();

FileSystem fs = FileSystem.get(conf);

FSDataOutputStream out = fs.create(new Path(uri));

2）建立文件元数据（create metadata）：RPC远程调用名称节点方法在文件系统的命名空间中新建一个文件，名称节点会执行相应的检查（文件是否存在，客户端权限等）；

3）写入请求（write request）：向系统发出写入数据请求;

4）写入数据包（write packets）：数据被分成一个个分包（packet）放入DFSOutputStream的内部队列，DFSOutputStream向名称节点申请存放数据块的若干数据节点，然后按照申请到的数据节点把文件分包分别传送到节点上存储。传送方式为：申请到的数据节点形成一个数据流管道，队列中的分包最后被打包成数据包第一个数据节点，第一个数据节点再将数据包发送到第二个节点。。。依此类推，形成“流水线复制”，直到数据包存储完成；

5）接收确认包（ack packet）：为了保证节点数据准确，完成接收的数据节点要向发送者发送“确认包”（ack packet），确认包沿着数据流管道逆流而上，经过各个节点最终到达客户端。客户端收到确认包后，将对应的分包从内部队列移除；

6）关闭文件（close file）：客户端通过FileSystem关闭文件；

7）结束过程（complete）：DFSOutputStream调用ClientProtocal.complete()方法

通知名称节点关闭文件，整个改文件流程结束。

**P27**



Master：HBase集群的主控制服务器，负责集群状态的管理维护。

1） 为Region Server分配region；

2 ）管理Region Server的负载均衡, 调整region分布；

3 ）发现失效的Region Server并重新分配其上的region；

4 ）处理schema更新请求。

Region Server：HBase具体对外提供服务的进程。一个Region Server一般是一台单独的计算机。一个物理节点一般只运行一个Region Server，但是它可以管理多个region，这些regions可以是来自于不同的表。具体任务如下：

1 ）维护Master分配给它的region，处理对这些region的访问请求，负责向HDFS文件系统读写数据；

2 ）负责切分在运行过程中变得过大的region。

Zookeeper：分布式协调服务器。提供可靠的锁服务并保证集群中所有的机器看到是视图是一致的，HBase使用Zookeeper服务来进行节点管理以及表数据定位。具体职责如下：

1） 保证任何时候集群中只有一个Master运行；

2 ）存储所有Region的寻址入口；

3 ）实时监控Region Server的状态，将状态信息实时通知给Master；

4 ）存储Hbase的schema, 包括有哪些table，每个table有哪些column family。

Client：包含访问HBase的接口。Client维护着一些cache来加快对HBase的访问，比如region的位置信息。

**P35**

表（Table）

一个HBase的表由很多行构成。

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RowKey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分段访问，或全表扫描)。

列族（Column Family）

列族必须在表定义时给出。每个列族可以包含一个或多个列(由列限定符Column Qualifier标志)，列成员不需要在定义时给出，可以随后按需要动态加入。

  数据按Column Family分开存储，HBase所谓的列式存储就是基于Column Family分散存储(每一个Column Family对应一个Store)。

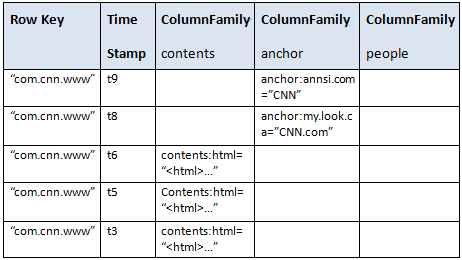
单元格（Cell）

单元位置由行键、列族、列限定符唯一确定，单元中存储的数据是没有类型的，全部以字节码的形式贮存。

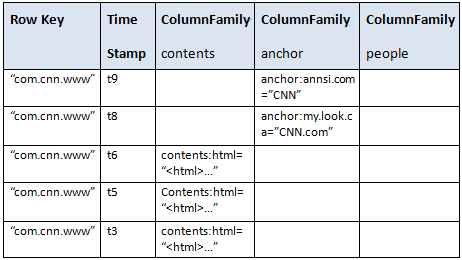
时间戳（Timestamp）

每个单元存储的数据随时间戳不同可以有多个版本，它们用时间戳来加以区分

**P36**



**P37**



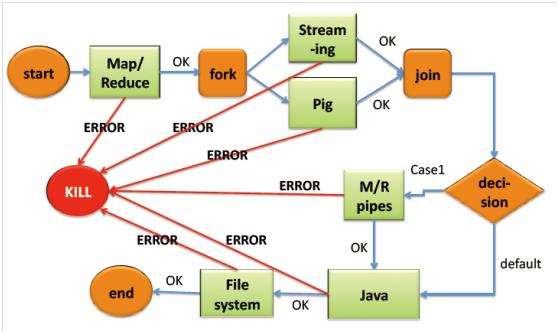


**P45**

我们的问题是：如何根据单元C11（红色格）找到单元C21（蓝色格）？也即，按SQL语句形式：查找符合条件“F:C1 = C11”的F:C2的值。由于不知道C11所对应的RowKey值，我们只有进行对主表的全局扫描，找到F:C1 = C11的单元，然后确定其行键RowKey，然后再根据RowKey进行数据查询，找到C21所在Region位置，读取C21的值。这是一个非常耗时耗力的低效过程。

如果我们构建了一个二级索引表（图中右表），索引表从主表中取出F:C1为RowKey，而以主表的行键（RK1，RK2，RK3）为列（索引数据），这样，针对“查找符合条件F:C1 = C11的F:C2的值”这样的查询，我们只需根据C11的值（RowKey）在二级索引表中找到对应的列值RK1（索引表中两个红色格子），然后再用RK1作为RowKey回到主表就可快速定位C21（图中左下表的红色格和蓝色格）。

**P56**



启动控制节点：工作流作业的入口点

末端控制节点：结束工作流作业时所用的节点，表示工作流执行已成功完成。一个工作流定义必须有一个末端节点。

停止控制节点：使工作流作业自行停止。在到达停止节点（kill node）时，如果工作流作业启动的一个或更多动作正在运行，那么当前运行的所有动作都将停止。工作流定义可以包含零个或多个停止节点。

决策控制节点：使工作流选择执行路径。决策节点的工作原理类似于拥有一组谓词转换对（predicates-transition pair）和一个默认转换的 switch-case 块。谓词是按顺序进行评估的，直至其中一个评估为 ture 为止，同时还会进行相应的转换。如果没有一个谓词被评估为 true，则采用swith的默认转换。

分支-联接控制节点：分支节点将一个执行路径分为多个并发路径。联接节点一直等待，直到前面的分支节点的所有并发执行路径都到达联接节点为止。分叉节点和联接节点必须成对使用。

**P58**

Resource Manager（RM）: 是YARN体系的Master，负责管理整个集群的资源分配，将各种计算资源（计算、内存、带宽等）以抽象资源单位Container的形式分配给 Node Manager（YARN 的节点代理）供Application使用。Resource Manager作为集群资源的管理调度员角色，如果发生单点故障，则整个集群的资源都无法使用。在Hadoop 2.4.0版本之后增加了RM High Availability的特性，增加了RM的可用性。

Node Manager（NM）: 是Resource Manger的Slave，是集群中实际拥有计算资源使用权的工作节点。在Application提交作业以后，YARN会将组成作业的多个task调度分配到多个DataNode上执行计算，而一个DataNode可能需要同时承担多个不同Application的task。这时，就需要NM（与Application Master配合）提供节点上的资源分配和task调度服务。NM的调度是基于抽象资源模型Container（资源容器），它代表着可供一个特定应用程序使用的该节点提供的计算资源（CPU、内存空间、网络带宽等），NM提供资源容器的生命周期管理以及对节点状态的监控。YARN 继续使用 HDFS 层，但主要将 NameNode 用于元数据服务，而 DataNode 用于集群中数据块的复制存储服务。NM组件的组成结构见图11-61.

Application Master（AM）: 主要管理和监控运行在DataNode上的Application task。以MapReduce为例，MapReduce Application是一个用来处理MapReduce计算的服务框架程序，通常用户编写的MapReduce程序包含多个Map task和Reduce task，而各个task的运行管理与监控都是由这个Application Master来负责。比如执行task所需资源的申请由AM向Resource Manager申请；启动/停止NM上某个task对应的Container，也由AM向Node Manager发出请求。从YARN 的角度，Application Master是用户代码，因此存在潜在的安全问题，因此将它们当作无特权的代码对待。

 Container: 是YARN对集群计算资源建立的抽象模型，即将Node Manager上的计算资源（CPU、内存空间）进行封装和量化，根据需要组装成一个个Container，然后服务于已给予资源授权的task。task在完成计算后，系统会回收资源，以供后续计算任务申请使用。Container目前包含两种资源：内存和CPU，以后的Hadoop版本可能会增加硬盘空间、网络带宽等物理资源。

YARN Client: 负责提交Application运行申请到RM，它会首先创建一个Application上下文件对象，并设置AM必需的资源请求信息，然后提交到RM。YARN Client也可以与RM通信，获取一个已经提交并运行的Application的状态信息等。

**P60**

比如Rource Manager使用Container作为资源调度的基本单位，每个Container包含一定量的CPU时间（也可看作虚拟CPU）和一定大小的内存（虚拟内存）；一个CPU需要支持多个Container，一个节点的内存空间也是划分为多个Container共享。用户程序运行在Container中，有点类似虚拟机。RM负责接收用户的资源请求并分配Container，NM负责启动Container并监控资源使用。如果使用的资源（目前只有内存）超出Container的限制，相应进程会被NM杀掉。可见Container这个概念不只用于资源分配，也用于资源隔离。YARN支持内存和CPU两种资源隔离。对于应用程序而言，内存大小是一种决定生死的资源，而CPU时间则是一种影响快慢的资源。YARN提供的内存隔离包括基于线程监控的方案和基于Cgroups（control groups）的方案，CPU隔离则包括默认不隔离和基于Cgroups的CPU隔离方案。