

Hadoop

华信培训


课程内容

- Hadoop集群
- HDFS
- MapReduce
- Hbase, Hive, Sqoop

创建虚拟机

1. 安装VMware-workstation  VMware-Workstation-15.0.4-

2. 创建新的虚拟机  **创建新虚拟机(N)**
创建新虚拟机，该虚拟机随后将添加到库的顶部。

3. 镜像文件  ubuntu-16.04.3-desktop-amd64.iso

4. 创建用户
密码: **hadoop**

新建虚拟机向导

简易安装信息
这用于安装 Ubuntu 64 位。

个性化 Linux

全名(E):

用户名(U):

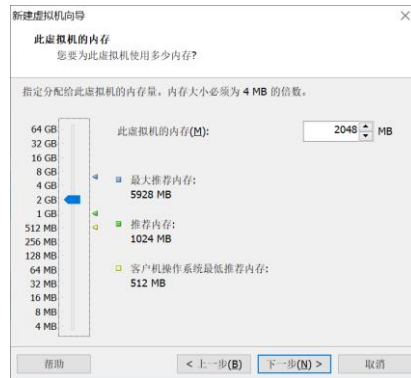
密码(P):

确认(C):

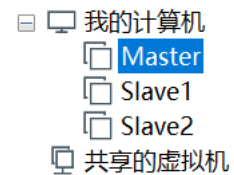
[帮助](#) [< 上一步\(B\)](#) [下一步\(N\) >](#) [取消](#)

创建虚拟机

5.推荐内存2G



6.克隆



修改虚拟机

7. 设置各虚拟机的hostname `$sudo vi /etc/hostname`

- 将文件中的名称改成想要的机器名 `master` , `slave1` , `slave2`。

编辑主机名解析文件, 填入对应的各节点地址和对应的主机名

- `$sudo vi /etc/hosts`
- `192.168.149.128 master`
- `192.168.149.129 slave1`
- `192.168.149.131 slave2`

配置IP

- 8.在各节点配置静态ip
- \$sudo vi /etc/network/interfaces

auto eth0

iface eth0 inet static

address 192.168.149.128

netmask 255.255.255.0

network 192.168.149.0

broadcast 192.168.149.255

gateway 192.168.149.2

dns-nameservers 8.8.8.8 8.8.4.4

设置SSH

- 9.设置公钥、私钥

查看ps -ef | grep ssh

安装sudo apt-get install ssh

三个节点全部生成公钥、私钥对ssh-keygen -t rsa

导入公钥cat .ssh/id_rsa.pub >> .ssh/authorized_keys

查看公钥cd .ssh cat id_rsa.pub

子节点传送密钥

scp .ssh/id_rsa.pub hadoop@master:/home/hadoop/id_rsa1.pub

scp .ssh/id_rsa.pub hadoop@master:/home/hadoop/id_rsa2.pub

设置SSH

- 10.设置免密码登录

```
cat id_rsa1.pub >> .ssh/authorized_keys
```

```
cat id_rsa2.pub >> .ssh/authorized_keys
```

最全公钥查看 `cat .ssh/authorized_keys`

返回子节点

```
scp .ssh/authorized_keys
```

```
hadoop@slave1:/home/hadoop/.ssh/authorized_keys
```

```
scp .ssh/authorized_keys
```

```
hadoop@slave2:/home/hadoop/.ssh/authorized_keys
```

验证: `ssh master`, `ssh slave1`, `ssh slave2`

配置集群

- 11.在hadoop各节点上创建以下文件夹:

/home/hadoop/data/namenode

/home/hadoop/data/datanode

/home/hadoop/temp

Master上 hadoop路径下解压缩hadoop2.7.2,jdk1.8



hadoop-2.7.2.tar.gz



jdk-8u91-linux-x64.tar.gz

配置集群

- 12.修改master
- 修改hadoop-env.sh 文件:
export HADOOP_PREFIX=/home/hadoop/hadoop-2.7.2(新建)
export JAVA_HOME=/home/hadoop/jdk1.8
- 修改yarn-env.sh文件:
export JAVA_HOME=/home/hadoop/jdk1.8

配置集群

- 修改core-site.xml文件:

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

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

```
<configuration>
```

```
  <property>
```

```
    <name>fs.defaultFS</name><value>hdfs://master:9000</value>
```

```
  </property>
```

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

```
    <value>file:/home/hadoop/temp</value>
```

```
  </property>
```

```
</configuration>
```

配置集群

- 修改hdfs-site.xml文件:

```
<configuration>
```

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

```
<value>file:/home/hadoop/data/namenode</value></property>
```

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

```
<value>file:/home/hadoop/data/datanode</value>
```

```
</property><property>
```

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

```
<value>slave1:50090</value></property><property>
```

```
<name>dfs.replication</name><value>2</value></property><property>
```

```
<name>dfs.permissions.enabled</name><value>>false</value></property>
```

```
</configuration>
```

配置集群

- 修改mapred-site.xml文件:

```
<configuration>
```

```
<property><name>mapreduce.framework.name</name>
```

```
<value>yarn</value></property>
```

```
<property><name>mapreduce.jobhistory.address</name>
```

```
<value>master:10020</value></property>
```

```
<property>
```

```
    <name>mapreduce.jobhistory.webapp.address</name>
```

```
    <value>master:19888</value>
```

```
</property>
```

```
</configuration>
```

配置集群

- 修改yarn-site.xml文件:

```
<configuration>
```

```
<property><name>yarn.nodemanager.aux-services</name>
```

```
<value>mapreduce_shuffle</value></property>
```

```
<property>
```

```
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
```

```
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
```

```
</property><property>
```

```
<name>yarn.resourcemanager.scheduler.address</name>
```

```
<value>master:8030</value></property>
```

配置集群

```
<property>
```

```
<name>yarn.resourcemanager.admin.address</name>
```

```
<value>master:8033</value></property>
```

```
<property> <name>yarn.resourcemanager.address</name>
```

```
<value>master:8032</value> </property>
```

```
<property>
```

```
<name>yarn.resourcemanager.resource-tracker.address</name>
```

```
<value>master:8031 </value></property>
```

```
<property><name>yarn.resourcemanager.webapp.address</name>
```

```
<value>master:8088</value>
```

```
</property>
```

```
</configuration>
```

配置集群

- 修改slave文件:

slave1

slave2

修改三个节点的环境变量 `sudo vi /etc/profile`

```
export HADOOP_HOME=/home/hadoop/hadoop-2.7.2
```

```
export
```

```
CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH:$HADOOP_HOME/share/hadoop/common/hadoop-common-2.7.2.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.7.2.jar:$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar
```

```
export
```

```
PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$HADOOP_HOME/sbin:$HADOOP_HOME/bin:$PATH
```


配置集群

```
export JAVA_HOME=/home/hadoop/jdk1.8
```

```
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

```
export PATH=$JAVA_HOME/bin:$PATH
```

```
export JRE_HOME=$JAVA_HOME/jre
```

```
source /etc/profile
```

```
java -version
```

```
hadoop version
```

```
hadoop@slave1:~$ java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
hadoop@slave1:~$ hadoop version
Hadoop 2.7.2
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r b165c4fe8a74265c792ce23f546c64604acf0e41
Compiled by jenkins on 2016-01-26T00:08Z
Compiled with protoc 2.5.0
From source with checksum d0fda26633fa762bff87ec759ebe689c
This command was run using /home/hadoop/hadoop-2.7.2/share/hadoop/common/hadoop-common-2.7.2.jar
```

配置集群

- 13.拷贝hadoop2.7.2到其它子节点
- `$scp -r hadoop-2.7.2/ hadoop@slave1:/home/hadoop`
- `$scp -r hadoop-2.7.2/ hadoop@slave2:/home/hadoop`

启动集群

- 14.格式化hdfs，启动

在master上格式化**HDFS**文件系统：

```
$ hdfs namenode -format
```

启动hadoop集群：\$ start-dfs.sh

启动yarn：\$start-yarn.sh

```
$ start-all.sh $ stop-all.sh
```

查看进程：\$jps

查看hdfs的状态报告：hdfs dfsadmin -report

查看yarn的基本信息：yarn node -list

测试

- 15.

创建文件夹: `hadoop fs -mkdir /input`

上传文件: `hadoop fs -put etc/hadoop/*.xml /input`

`hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar
grep /input /output 'you'`

导出: `hadoop fs -get /output /home/hadoop/output`

课程内容

- Hadoop集群
- HDFS
- MapReduce
- Hbase, Hive, Sqoop

GFS (Google文件系统)

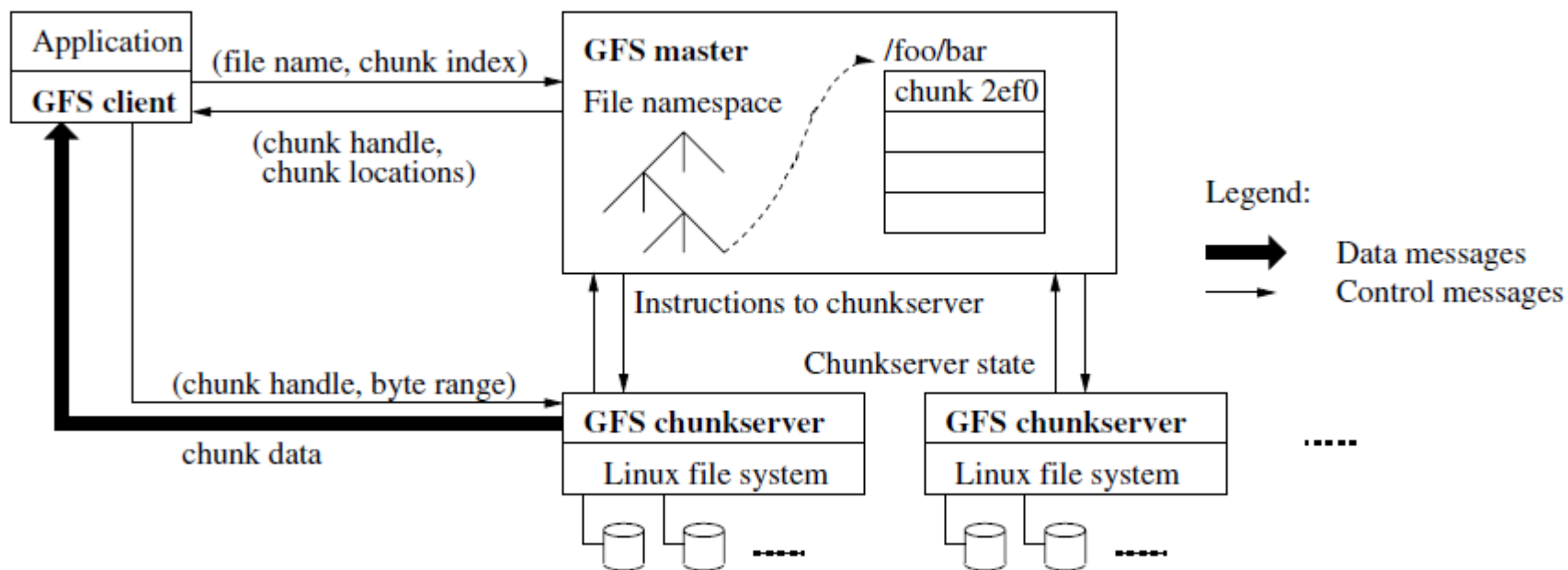


Figure 1: GFS Architecture

HDFS

- **HDFS**是分布式文件系统，**HDFS**有高容错性的特点，并且设计用来部署在低廉的硬件上；而且它提供高吞吐量（**high throughput**）来访问应用程序的数据，适合那些有着超大数据集的应用程序。

特性：

高容错性

高吞吐量

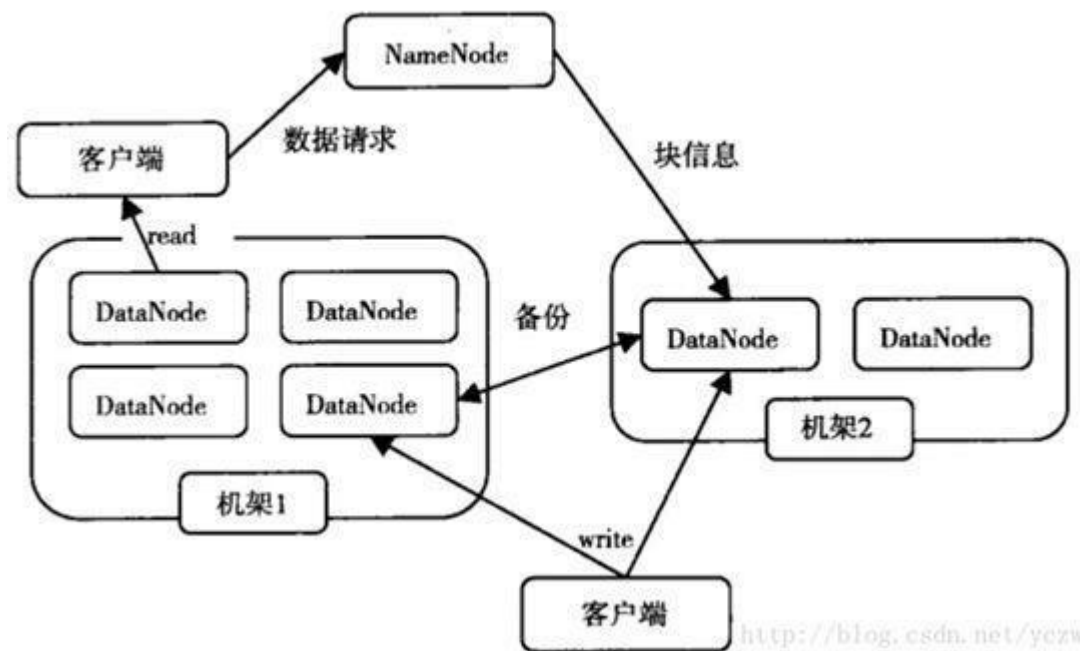
适用于大文件存储

适用于流式文件数据访问

故障检测和自动恢复

低成本——基于普通硬件集群构建

HDFS体系结构



Namenode和Datanode

- HDFS采用master/slave架构。
 - 一个HDFS集群是由一个Namenode和一组Datanodes构成
- Namenode是一个中心服务器，负责管理文件系统的命名空间(namespace)以及处理客户端对文件的访问请求，设置HDFS保存的文件的副本数目。
- Datanode是数据节点，负责管理它所在节点上的数据存储，在Namenode的统一调度下进行数据块的创建、删除和复制。
- 为了容错，文件的所有数据块都会有副本

基于块block的文件存储

- HDFS中，文件被切分成固定大小的数据块进行存储
- 默认的数据块大小为128M
- 较大的文件块减少了寻址开销，提升了读写效率
- 通过文件块，实现了对超大文件的支持
- 文件块信息单独保存在元数据中，实现了文件的控制信息和数据存储的分离

HDFS的元数据

- 元数据（**Metadata**）：维护HDFS文件系统中文件和目录的信息，分为内存元数据和元数据文件两种。
- 元数据包含了数据块到文件的映射信息以及文件系统的属性信息
- **NameNode**维护整个元数据。
- HDFS采用元数据镜像文件（**FSImage**）+日志文件（**edits**）的备份机制。
- **Namenode**使用事务日志**Editlog**记录系统元数据的修改
例如：创建新文件、修改文件的副本设置
- HDFS会定期的对最近的**fsimage**和一批新**edits**文件进行**Checkpoint**

基于数据块的数据存储

- **HDFS**被设计成能够在一个大集群中跨机器可靠地存储超大文件。
- 它将每个文件存储成一系列的数据块，除了最后一个，所有的数据块都是同样大小的。
- 为了容错，文件的所有数据块都会有副本。每个文件的数据块大小和副本系数都是可配置的。
- 应用程序可以指定某个文件的副本数目。
- 副本系数可以在文件创建的时候指定，也可以在之后改变

数据存储理念

- **Namenode**全权管理数据块的复制
- **Namenode**周期性地从集群中的每个**Datanode**接收心跳信号和块状态报告(**Blockreport**)
- 接收到心跳信号意味着该**Datanode**节点工作正常
- 块状态报告包含了一个该**Datanode**上所有数据块的列表

副本放置策略

- 一个文件划分成多个block，每个block存多份，如何为每个block选择节点存储这几份数据？
- Block副本放置策略：
 - 副本1: 同Client的节点上
 - 副本2: 同机架内其它节点上
 - 副本3: 不同机架中的节点上
另一个节点上
 - 其他副本: 随机挑选

HDFS负载均衡

- Hadoop的HDFS集群非常容易出现机器与机器之间磁盘利用率不平衡的情况，例如：当集群内新增、删除节点，或者某个节点机器内硬盘存储达到饱和值。
- 当HDFS负载不均衡时，需要对HDFS进行数据的负载均衡调整，即对各节点机器上数据的存储分布进行调整。从而，让数据均匀的分布在各个DataNode上，均衡IO性能，防止热点的发生。
- 进行数据的负载均衡调整，必须要满足如下原则：
 - 数据平衡不能导致数据块减少，数据块备份丢失
 - 管理员可以中止数据平衡进程
 - 每次移动的数据量以及占用的网络资源，必须是可控的
 - 数据均衡过程，不能影响namenode的正常工作

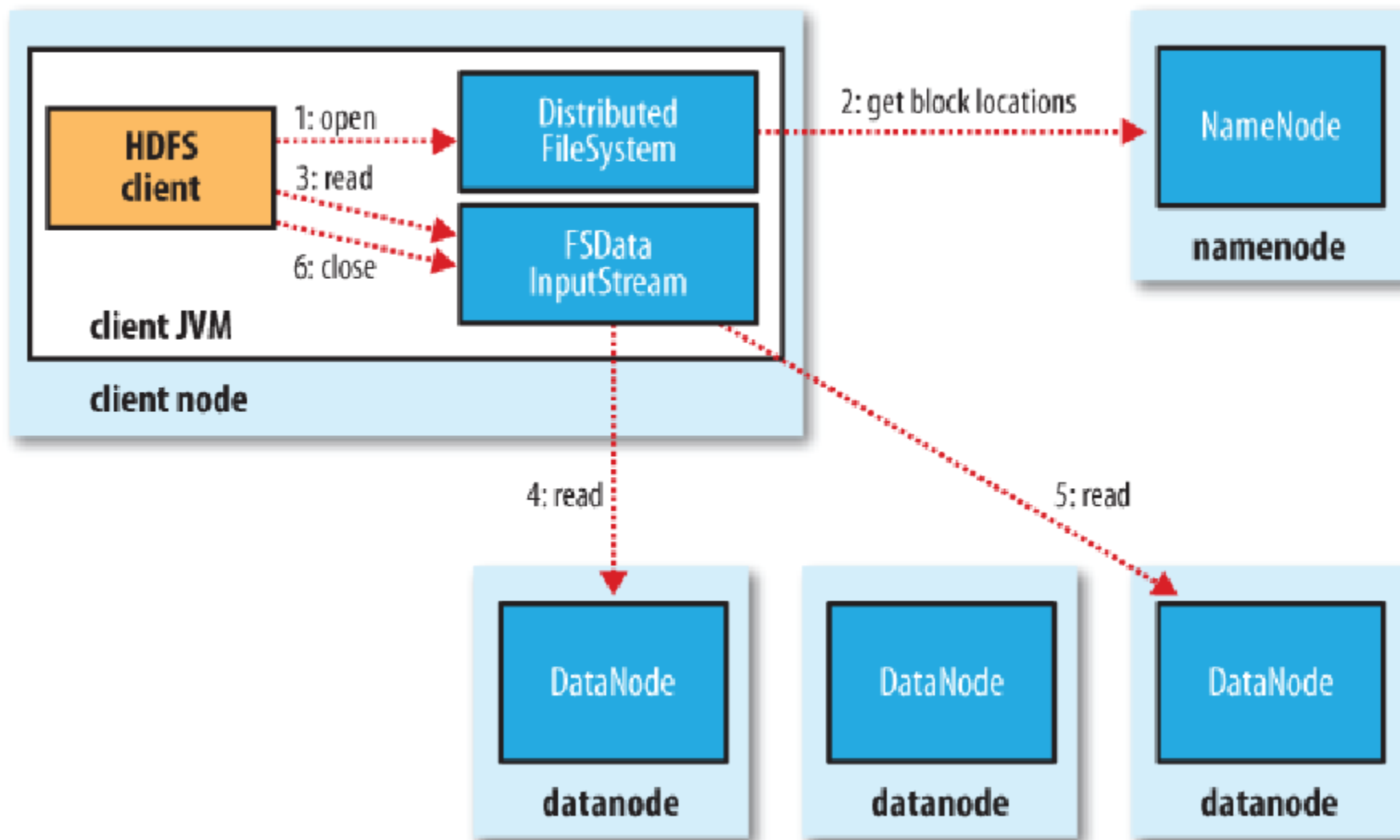
安全模式

- 当集群启动的时候，会首先进入到安全模式
 - 安全模式是hadoop集群的一种保护方式
 - 每个Datanode检查心跳信息和块报告Blockreport信息
 - Namenode校验每个数据块是否有足够的副本
- 安全模式下不允许客户端进行修改文件的操作
- 通过系统自检之后，Namenode退出安全模式
 - Namenode会按自检结果生成一份数据块复制列表
 - 然后按列表执行数据备份
 - 判断通过自检的最小副本率是可以设置的

存储空间回收

- 文件删除
 - 当客户端删除一个文件时，**HDFS**将文件移动到`/trash`文件夹，并保存一段时间
 - 上述时间段内，客户可以恢复这个文件
 - 在该时间段之后，文件被彻底删除，并且磁盘空间得到释放
- 减少数据副本系数
 - 将数据复制因子降低，**Namenode**会选择多余的可以删除的数据副本
 - 下一个心跳信息`heartbeat`发送时，删除信息会发送到数据节点，然后相应的数据块会被删除

存储空间回收



文件读取流程

- 文件读取基本流程如下
 1. 客户端发出打开文件请求
 2. 通过**Namenode**查询到数据块的位置地址信息
 3. 开始读取文件数据，从第一个数据块开始，顺序读取每个数据块的数据信息，直到完成读取
- 备注
 - 如果遇到错误，自动尝试读取其它数据块副本
 - 读取之后会对每个数据块进行校验，如果校验错误，会自动尝试读取其它数据块副本
 - 多次错误后，会对出错数据节点进行标注，避免再次访问
- 上述流程实现了数据的分布式读取，**Namenode**仅负责查询数据块地址，因此，**HDFS**支持高并发和高数据吞吐量

HDFS数据块读取策略

- HDFS将网络看作一棵树
- 网络中两个节点的距离是距离他们最近共同祖先的总和
- 为了降低整体的带宽消耗和读取延时，HDFS会尽量让读取程序读取离它最近的副本
- HDFS的数据块读取优先级是：
 1. 本节点数据块
 2. 同一机架不同节点数据块
 3. 同一数据中心不同机架不同节点的数据块
 4. 其它数据中心的数据块

文件读操作流程

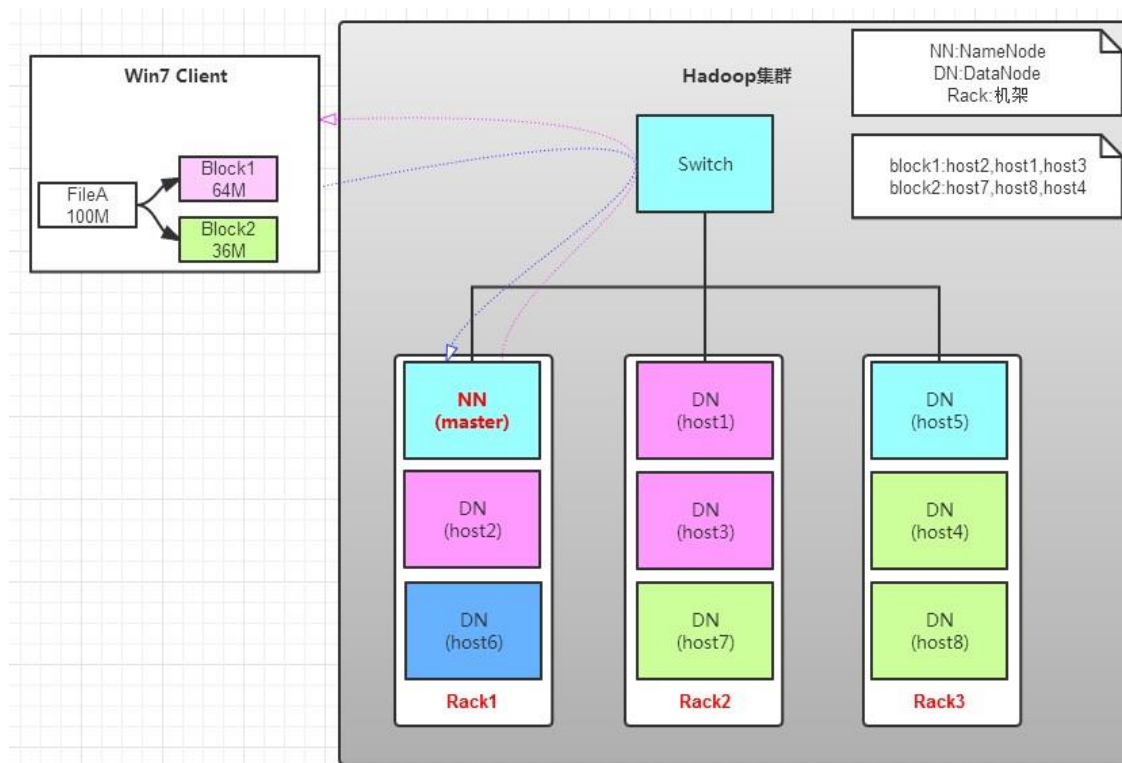
- 文件读取基本流程如下

1. 客户端发出打开文件请求

2. 通过Namenode查询到数据块的位置地址信息

3. 开始读取文件数据，从第一个数据块开始

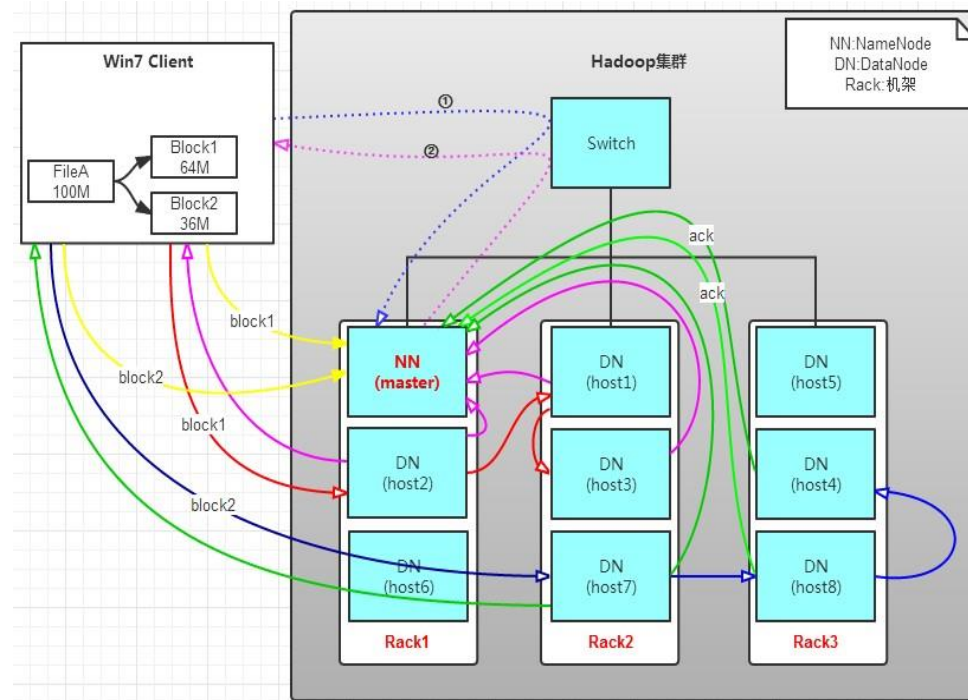
顺序读取每个数据块的数据信息，直到完成读取



文件写入流程

- 文件写入基本流程如下

1. 客户端发出创建文件请求
2. 向Namenode请求创建新文件
3. 如果请求通过，Namenode生成新文件记录



4. 将文件数据分块写入内部数据队列，然后按数据副本因子顺序写入到每个数据节点
5. 数据块的数据已经写入全部数据节点之后，将会被移出队列
6. 直到全部数据块写入完毕，文件写入完成

HDFS的健壮性

- HDFS的主要目标就是即使在出错的情况下也要保证数据存储的可用性及可靠性
- 常见的三种出错情况：
 - Namenode出错
 - Datanode出错
 - 网络割裂(network partitions)
- 一般情况下，Datanode出错或者局部网络故障，不会影响HDFS的正常工作，一旦Namenode出现问题，HDFS系统将无法正常工作

Namenode故障

- HDFS主节点Namenode保存FsImage和Editlog数据
- 一旦主节点的FsImage和Editlog数据出现故障，HDFS系统将无法正常工作
- 为提升可用性，Namenode可以通过配置保存多份FsImage和Editlog数据，数据同步更新
- 在HDFS2 中使用HA机制保证Namenode 的高可用

Datanode异常

- 每个Datanode节点周期性地向Namenode发送心跳信号
- 网络故障（或其它故障）可能导致一组Datanode无法连接到Namenode
- Namenode一旦发现没有按时收到Datanode的心跳Heartbeat信息，就对这些Datanode进行宕机标注
- Namenode不再向这些节点发送I/O请求，Datanode上的数据暂时变成不可用的状态
- Datanode的异常也会导致数据块的副本数量不足，这时会触发数据重新复制的过程

数据完整性校验

- 有时候客户端读取的数据可能已经损坏
 - 数据损坏可能由存储设备故障、网络故障和软件故障导致
- HDFS的客户端计算数据块的checksum值，并与HDFS命名空间中存储的数值进行对比，以校验文件完整性
- 如果checksum值校验错误，客户端将从其它数据副本重新读取数据

HDFS可靠性策略

- 文件完整性

通过CRC32校验，来确定文件是否损坏；如损坏，用其他副本取代损坏文件。

- Heartbeat

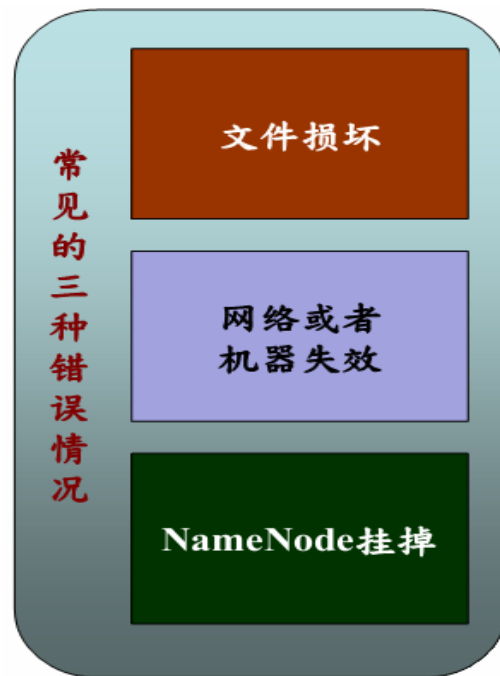
Datanode 定期向Namenode发heartbeat

- 元数据信息

- FSImage（文件系统镜像）、Editlog

(操作日志)

- 多份存储
- 主备NameNode实时切换



HDFS shell常用命令

- HDFS有很多shell命令，利用该命令可以查看HDFS文件系统的目录结构、上传和下载数据、创建文件等。
- Hadoop中有三种Shell命令方式：
 - `hadoop fs`适用于任何不同的文件系统，比如本地文件系统和HDFS文件系统
 - `hadoop dfs`只能适用于HDFS文件系统
 - `hdfs dfs`跟`hadoop dfs`的命令作用一样，也只能适用于HDFS文件系统
- 命令的用法为：
 - `hdfs dfs [genericOptions] [commandOptions]`
 - `hadoop fs [genericOptions] [commandOptions]`

文件操作命令

- 显示指定目录下文件的详细信息 `hdfs dfs -ls /`
- 创建目录 `hdfs dfs -mkdir /music`
- 上传文件
 - `hdfs dfs -put test.txt /music`
 - `hdfs dfs -copyFromLocal /data/a.mp3 /music/b.mp3`
- 下载文件 `hdfs dfs -copyToLocal /music/b.mp3 /data/bbb.mp3`
- 查看文件内容 `hdfs dfs -cat /music/test.txt`
- 追加内容 `hdfs dfs -appendToFile README.txt /music/test.txt`
- 复制文件 `hdfs dfs -cp /music/test.txt /music/cxx.txt`
- 重命名文件 `hdfs dfs -mv /music/cxx.txt /music/b.txt`
- 删除文件 `hdfs dfs -rm /music/c.txt`

管理命令

dfsadmin 命令用来管理HDFS集群，管理员使用。

- 显示集群状态信息 `hdfs dfsadmin -report`
- 将集群置于安全模式 `hdfs dfsadmin -safemode enter`
- 设置每个数据节点在HDFS块平衡时的网络带宽

`hdfs dfsadmin -setBalancerBandwidth 1000`

- 显示拓扑 `hdfs dfsadmin -printTopology`
- Hadoop提供一个fsck工具来检查HDFS中文件的健康情况。该工具将查找所有数据节点中丢失的块；副本不足或者副本过多的块；查看一个文件的所有数据块位置；删除损坏的数据块。

`hdfs fsck /`

`hdfs fsck /music -locations`

动态新增节点

- clone 一份 slave2 作为新增的节点 slave3, 修改静态 IP , 同时修改 master、slave1 和 slave2 的 /etc/hosts 文件
- 修改 master 的 etc/hadoop/slaves 文件, 添加新增的节点 slave3, 并将其更新到各个节点。
- 在新增的 slave3 节点执行命令启动 datanode
- ./sbin/hadoop-daemon.sh start datanode
- 在新增的 slave3 节点执行命令启动 nodemanager
- ./sbin/yarn-daemon.sh start nodemanager

动态删除节点

- 在etc/hadoop/ 目录下添加 excludes 文件，配置需要删除的节点
- 修改 etc/hadoop/hdfs-site.xml

```
<property>
```

```
  <name>dfs.hosts.exclude</name>
```

```
  <value>/home/hadoop/hadoop-2.7.2/etc/hadoop/excludes</value>
```

```
</property>
```

- 修改 mapred-site.xml

```
<property>
```

```
  <name>mapred.hosts.exclude</name>
```

```
  <value>/home/hadoop/hadoop-2.7.2/etc/hadoop/excludes</value>
```

```
  <final>true</final>
```

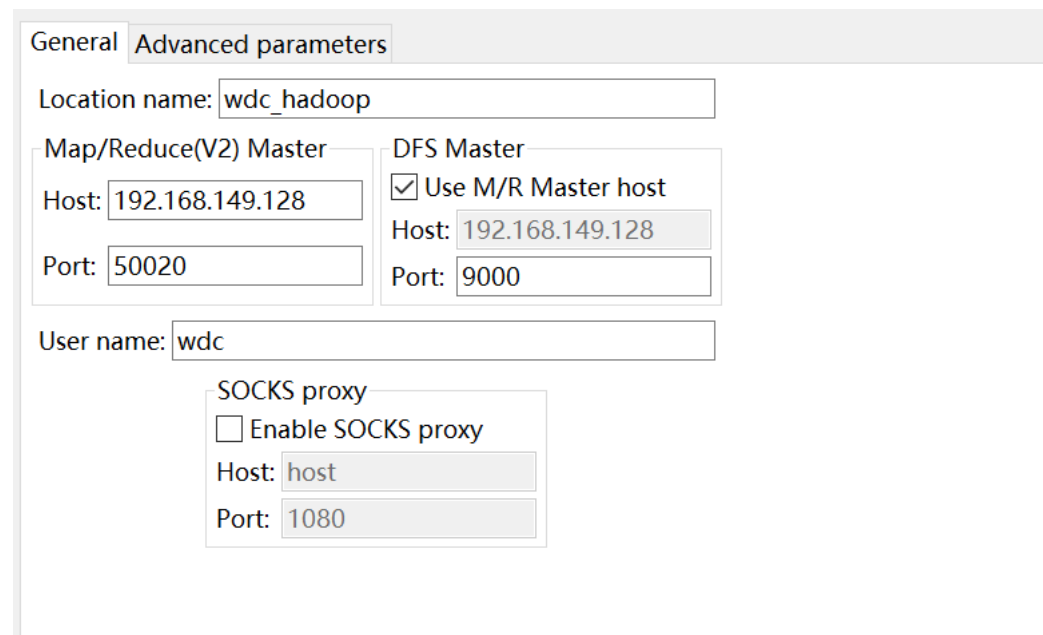
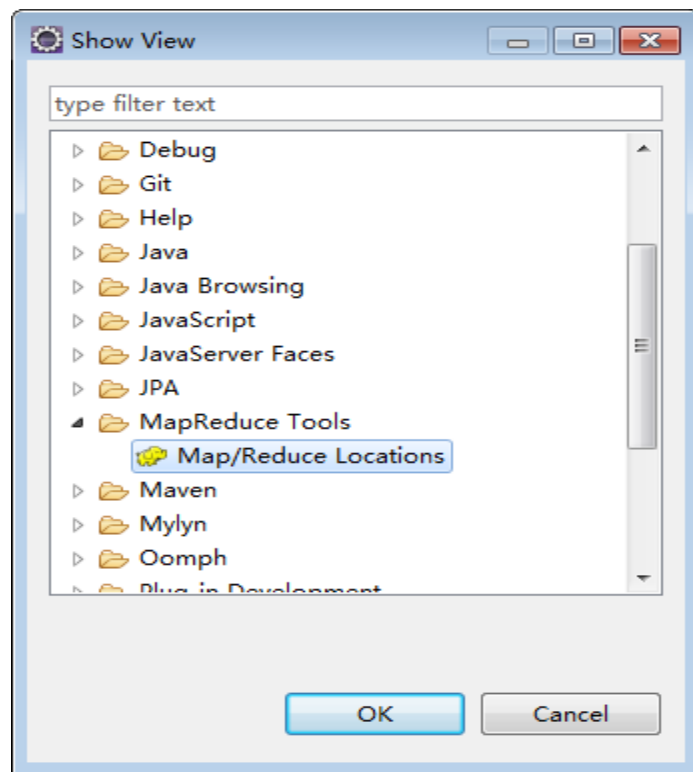
```
</property>
```


动态删除节点

- 对节点进行强制刷新
 - hdfs dfsadmin -refreshNodes
- 查看节点状态，该节点的状态为Decommissioned（退役）
- 在 slave3 节点上关闭 datanode 和 nodemanager 进程
 - `hadoop-daemon.sh stop datanode`
 - `yarn-daemon.sh stop nodemanager`
- 运行start-balancer.sh均衡 block
- `start-balancer.sh`

Java API—使用hadoop插件搭建开发环境

- windows下载hadoop-2.7.2解压到某目录下
- 把hadoop-eclipse-plugin-2.7.2.jar放到eclipse的plugins目录下。
- 在eclipse下配置插件， windows->show view->other
- 创建mapreduce项目



Java API—上传文件到HDFS

```
Configuration conf = new Configuration();
conf.set("fs.defaultFS", "hdfs://192.168.149.128:9000");
FileSystem hdfs = FileSystem.get(conf);
Path src = new Path("d:/abc.txt");
Path dst = new Path("/music/test_hdfs.text");
hdfs.copyFromLocalFile(src, dst);
System.out.println("Upload to :"+conf.get("fs.default.name"));
FileStatus files[] = hdfs.listStatus(dst);
for(FileStatus file:files){
    System.out.println(file.getPath());
}
```

Java API—新建文件

```
Configuration conf = new Configuration();  
conf.set("fs.defaultFS", "hdfs://192.168.149.128:9000");  
FileSystem hdfs = FileSystem.get(conf);  
byte[] buff = "hello wdc".getBytes();  
Path dst = new Path("/music/hello.txt");  
FSDataOutputStream outputStream = hdfs.create(dst);  
outputStream.write(buff,0,buff.length);  
outputStream.close();  
boolean isExists = hdfs.exists(dst);  
System.out.println(isExists);
```

Java API—显示文件的内容

```
Configuration conf = new Configuration();  
conf.set("fs.defaultFS", "hdfs://192.168.149.128:9000");  
FileSystem hdfs = FileSystem.get(conf);  
FSDataInputStream is = hdfs.open(new Path("/music/tmp.txt"));  
int i = is.read();  
while(i != -1){  
    System.out.print((char)i);  
    i = is.read();  
}  
is.close();
```

Java API—列出目录下的所有文件

```
Configuration conf = new Configuration();  
String uri = "hdfs://192.168.149.128:9000";  
FileSystem hdfs = FileSystem.get(URI.create(uri),conf);  
FileStatus[] status = hdfs.listStatus(new Path("/music"));  
for(FileStatus sta : status){  
    System.out.println(sta);  
}
```

Java API—删除HDFS上的文件及目录

```
Configuration conf = new Configuration();  
conf.set("fs.defaultFS", "hdfs://192.168.149.128:9000");  
FileSystem hdfs = FileSystem.get(conf);  
Path dst = new Path("/music/hello.txt");  
boolean isdeleted = hdfs.delete(dst, false);  
System.out.println(isdeleted);
```

课程内容

- Hadoop集群
- HDFS
- MapReduce
- Hbase, Hive, Sqoop

MapReduce介绍

- MapReduce是分布式计算框架，MapReduce是Google的一项重要技术，它是一个编程模型，用以进行大数据量的计算。
- MapReduce的名字源于这个模型中的两项核心操作：Map和Reduce。
- Map 阶段并行处理输入数据； Reduce阶段对Map结果进行汇总；
- MapReduce适合PB级以上海量数据的离线处理，性能比较低，但吞吐率很高。
- Mapper负责“分”
 - 把复杂的任务分解为若干个“简单的任务”来处理。“简单的任务”包含三层含义：
 - 数据或计算的规模相对原任务要大大缩小
 - 就近计算原则，任务会分配到存放着所需数据的节点上进行计算
 - 这些小任务可以并行计算彼此间几乎没有依赖关系
- Reducer负责对map阶段的结果进行汇总

词频统计

- 统计一组文档中每个单词的出现次数
 - 例如: "About DHC is the IT solutions...DHC"
→ (DHC,2);(About ,1);(is,1);(the,1).....
- 伪代码:

```
定义wordCount 列表: 由<word, count>键值对组成
对每个文档D in 文档集DS {
    对每个单词 W in D{
        wordCount[W]++;}
    }
返回wordCount;
```

词频统计

- 使用单台PC循序执行将会运行很长时间
 - 可以考虑将一亿文档分散到一组PC来执行程序
 - 每个PC负责处理一部分文档；全部处理完成之后，再合并各个PC的处理结果

- 伪代码：

```
定义wordCount 列表和totalWordCount列表： 由
<word, count>键值对组成
对每个文档D in 文档子集DSS {
    对每个单词 W in D{
        wordCount[W]++;}
}
保存wordCount为中间结果;
对每个 wordCount 中间结果{
    合并结果(totalWordCount, wordCount );}
返回totalWordCount;
```

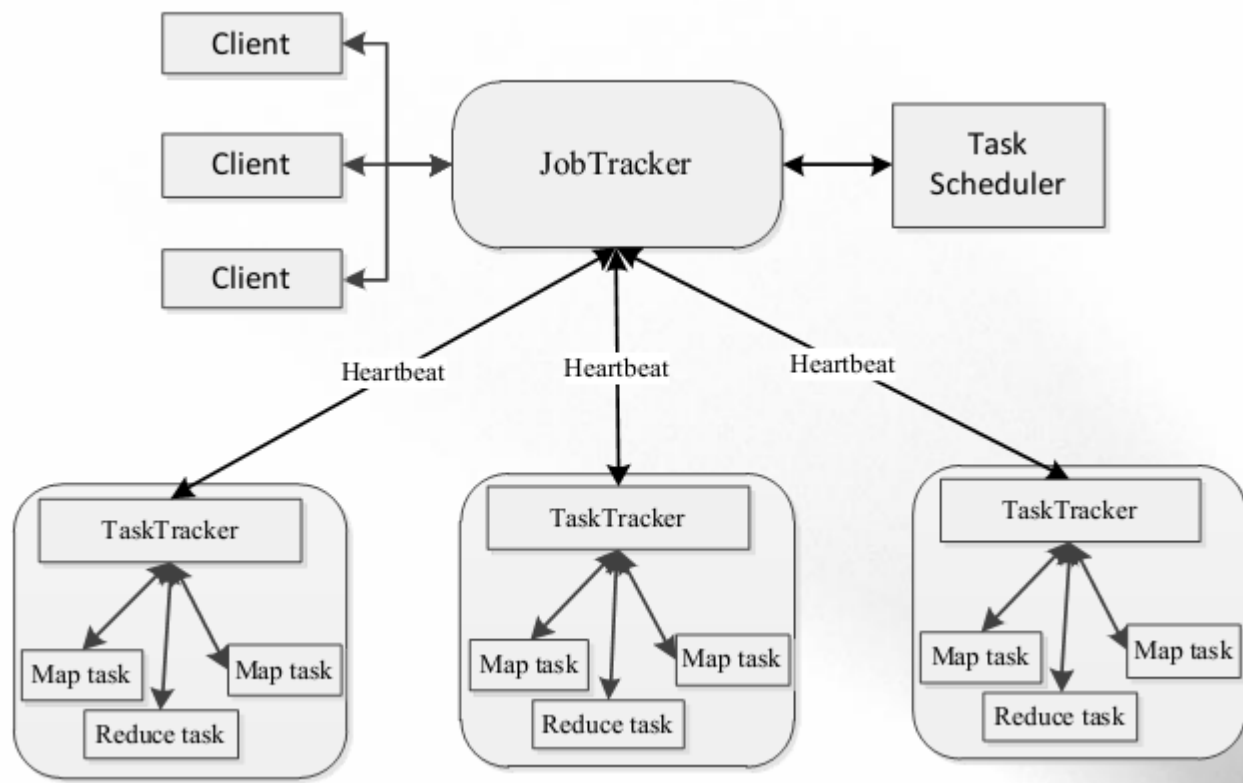
词频统计案例的问题

- 文件存储问题
 - 海量数据如何来存储
- 内存容量问题
 - 如果不同单词的数量超出了单机内存，程序无法执行
 - 在合并阶段，负责执行合并程序的**PC**将会成为性能瓶颈
- 解决方案：
 - 海量文档集分散存储到不同数据节点，然后每个数据节点只需要处理本机存储的文档子集。
 - 设定每个节点处理文件数量
 - 设置多个节点执行合并程序，多个节点仅需要合并某个字母开头的单词

MapReduce功能

- MapReduce程序分为两个阶段执行
 - mapping阶段和reducing阶段
- 在mapping阶段，MapReduce将输入数据拆分，并且将数据元素传送给mapper，过滤输入数据，将其转换为中间结果
- 在reducing阶段，reducer处理来自全部mapper的输出，得到最终结果，实现了结果合并和整合
- 主要数据类型：列表和键值对

Hadoop1.x MapReduce框架



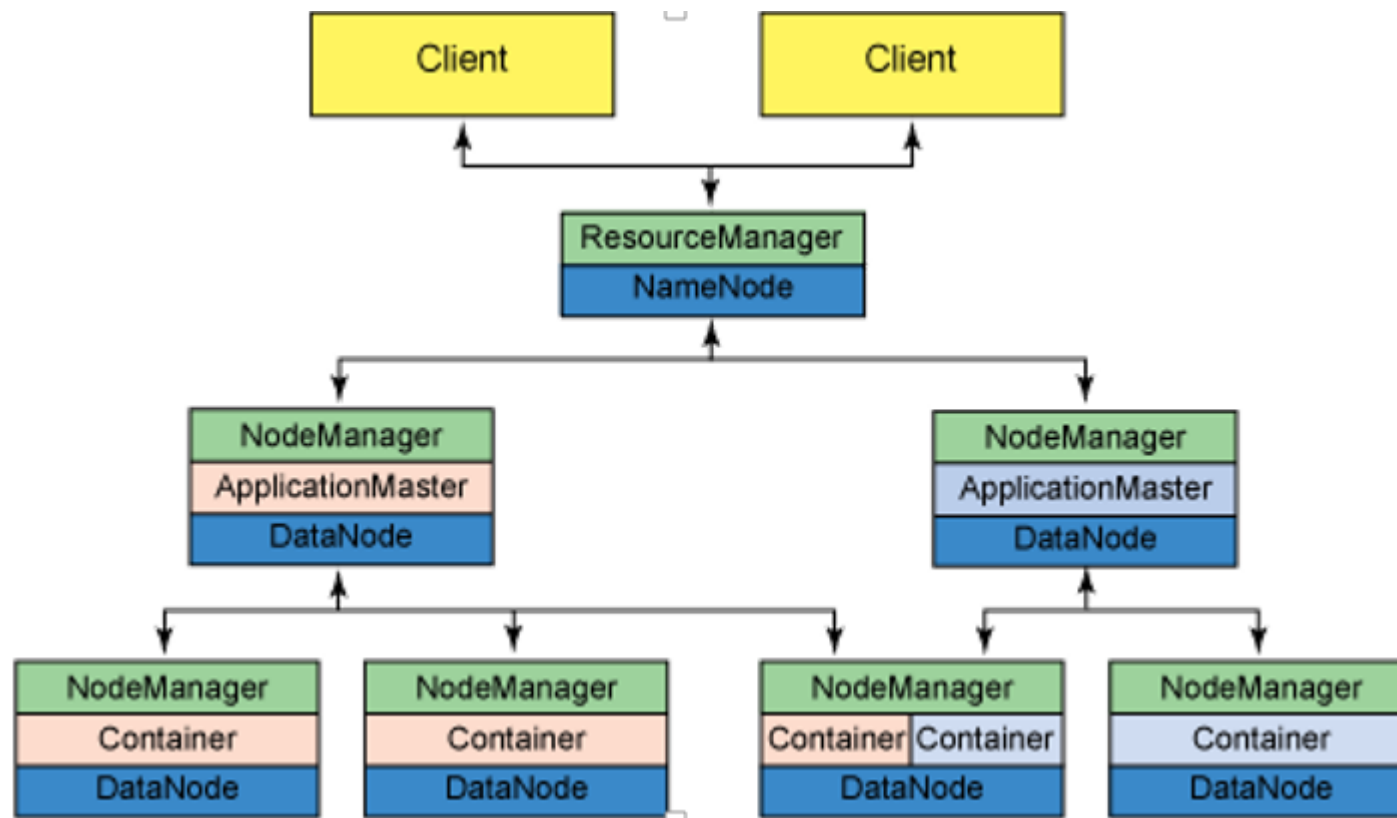
Hadoop1.x MapReduce缺欠

- JobTacker是整个Hadoop1.x MapReduce（MapReduce v1）框架的中心，其承担的任务有：接受任务、计算资源、分配资源、与DataNode进行交流等功能。随着需求越来越大，MapReduce v1已经不能够胜任现在的需求，主要表现在以下几个方面：
 - JobTracker是整个MapReduce v1的核心，存在单点故障
 - JobTracker管理整个MapReduce作业的任务，造成资源消耗，当map/reduce task过多的时候，JobTracker将会耗费大量内存，同时也增加Job Tracker fail的风险。
 - JobTracker对DataNode进行资源询问时，使用Task的个数，未考虑内存和CPU的使用率等，如果将两个大内存的Map/reduce Task放在一个节点上执行，可能会出现内存溢出。
 - 代码层中的类有些超过3000行，导致整个类的任务不够明确，并且进行修改时任务量也巨大，因此增加了维护、开发人员进行修改的难度。

YARN产生的背景

- Yarn是一个分布式的资源管理系统，用以提高分布式的集群环境下的资源利用率，这些资源包括内存、IO、网络、磁盘等。其产生的原因是为了解决原MapReduce框架的不足。最初MapReduce的committer们还可以周期性的在已有的代码上进行修改，可是随着代码的增加以及原MapReduce框架设计的不足，在原MapReduce框架上进行修改变得越来越困难，所以MapReduce的committer们决定从架构上重新设计MapReduce,使下一代的MapReduce(MRv2/Yarn)框架具有更好的扩展性、可用性、可靠性、向后兼容性和更高的资源利用率以及能支持除了MapReduce计算框架外的更多的计算框架。

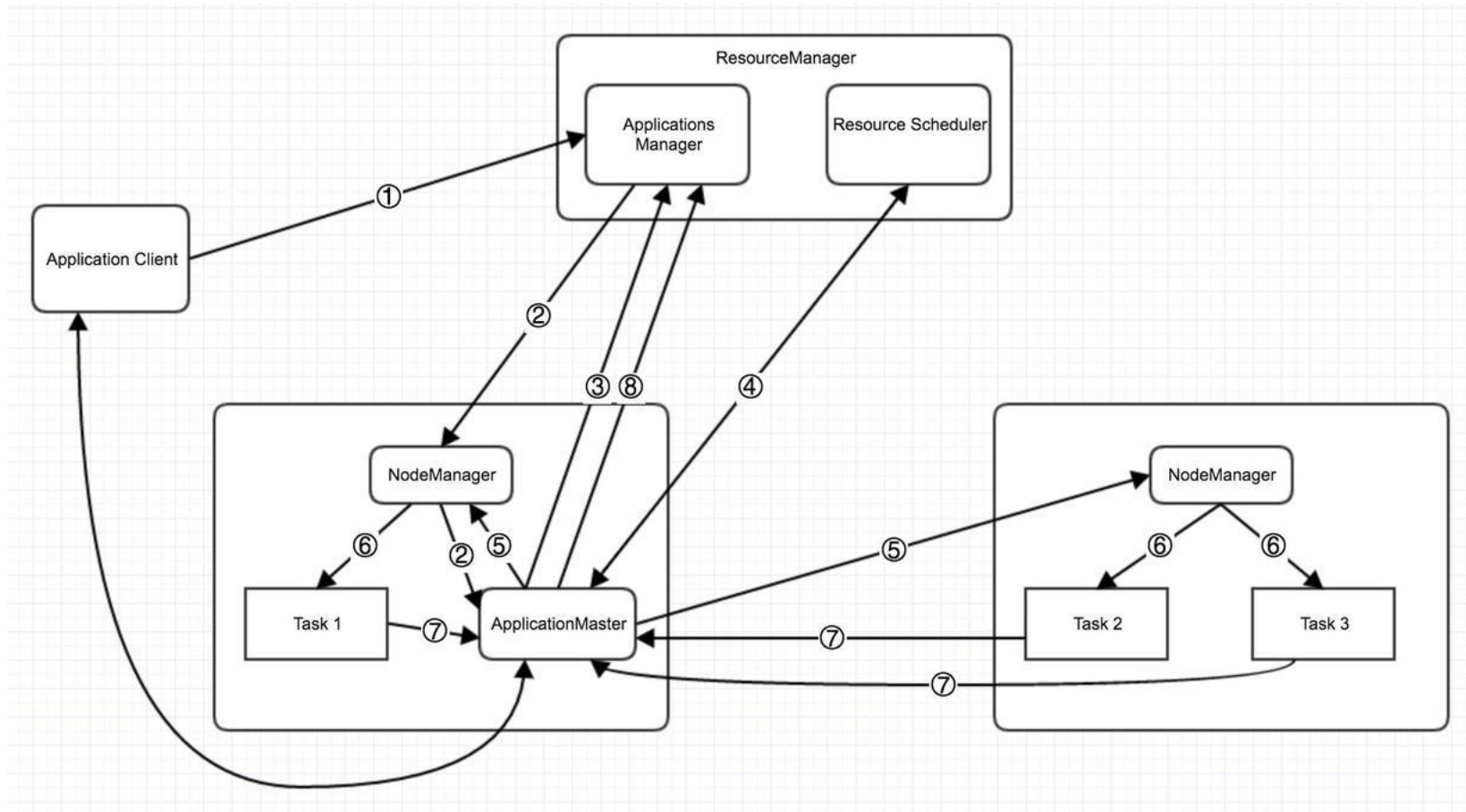
YARN基本架构



YARN基本架构

- **ResourceManager** 整个集群只有一个，负责集群资源的统一管理和任务调度。
- **Node Manager** 整个集群有多个，负责本节点的资源管理以及任务管理，**Node Manager**会定期向**Resource Manager**汇报本节点的资源使用情况；**Resource Manager**可以通过心跳应答方式向**Node Manager**下达命令或者分发新的任务。
- **Resource Manager** 将资源分配给某个应用程序后，应用程序会启动一个 **ApplicationMaster**，**ApplicationMaster**会为应用程序向**Resource Manager**申请资源，申请完资源后，再与对应的节点通信，运行自己内部的任务。

YARN工作流程



YARN工作流程

- 1、用户向YARN中提交应用程序
- 2、ResourceManager为该应用程序分配第一个Container，并与对应的Node-Manager通信，要求它在这个Container中启动应用程序的ApplicationMaster。
- 3、ApplicationMaster首先向ResourceManager注册，这样用户可以直接通过ResourceManage查看应用程序的运行状态。
- 4、ApplicationMaster通过RPC协议向ResourceManager申请和领取资源。
- 5、一旦ApplicationMaster申请到资源后，便与对应的NodeManager通信，要求NodeManager 在 Container 中启动任务。
- 6、NodeManager 收到 ApplicationMaster 的请求后，为任务设置好运行环境（包括环境变量、JAR 包、二进制程序等），启动任务。
- 7、各个任务通过RPC协议向ApplicationMaster汇报自己的状态和进度，以让ApplicationMaster随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。在应用程序运行过程中，用户可随时通过RPC向ApplicationMaster查询应用程序的当前运行状态。
- 8、应用程序运行完成后，ApplicationMaster向ResourceManager注销并关闭自己。

ResourceManager

- ResourceManager是Yarn上的主进程，部署在集群的主节点上，负责集群资源的统一管理和任务调度，ResourceManager主要功能：
 - 处理客户端请求：响应client端的web、命令行程序请求
 - RTS（ResourceTrackerService）：监控所有slave资源状况，将各NodeManager上传的资源汇总成一个资源清单（以Container形式）供调度器使用。
 - 资源分配与调度：将各个机器上的资源进行统筹安排，灵活的将资源进行分配和调度。
 - ApplicationsManager：负责管理所有的ApplicationMaster，每个应用程序对应一个ApplicationMaster，负责应用程序的资源申请、任务调度、任务容错等。

NodeManager

- NodeManager是yarn上的从进程，集群中每个slave节点上都要部署一个NodeManager，主要功能包括：

负责管理本机资源

定时地向RM汇报本节点上的资源使用情况

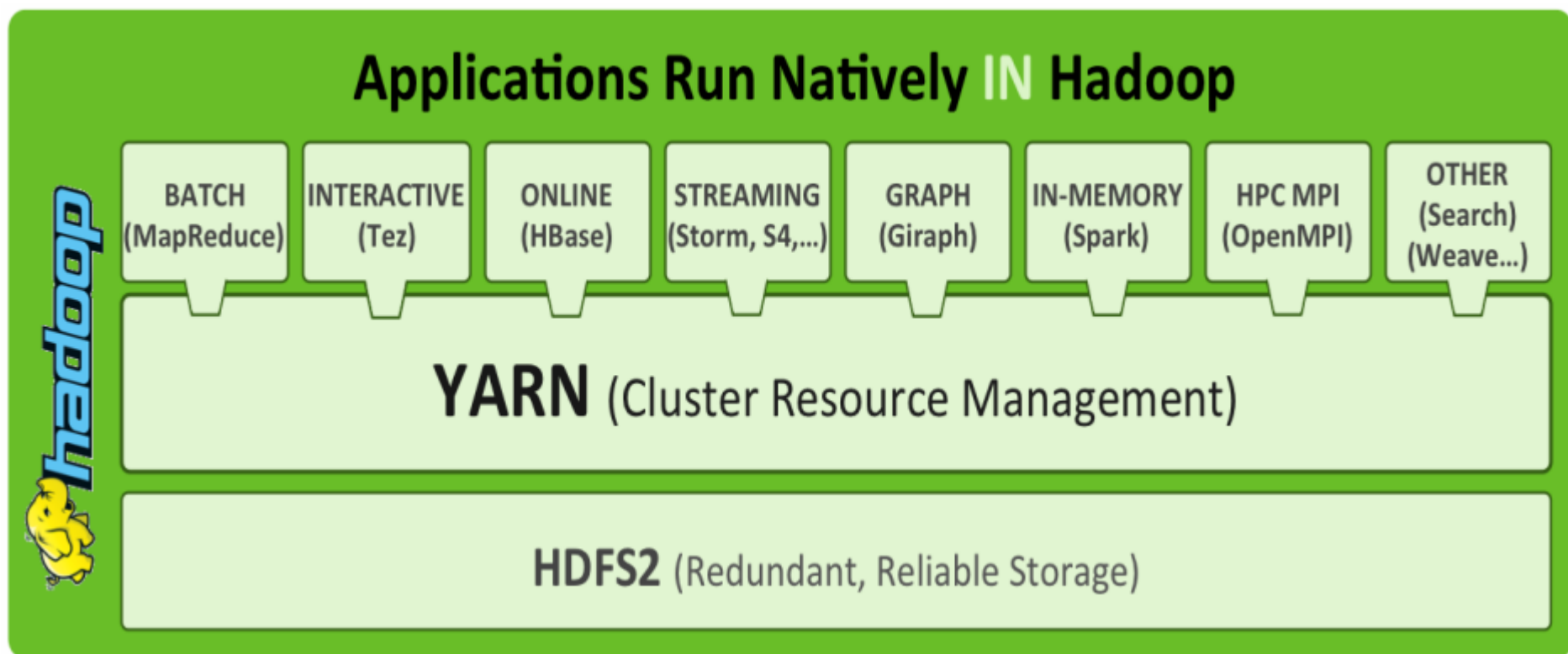
处理来自ResourceManager的命令

处理来自ApplicationMaster的命令

其它进程及资源

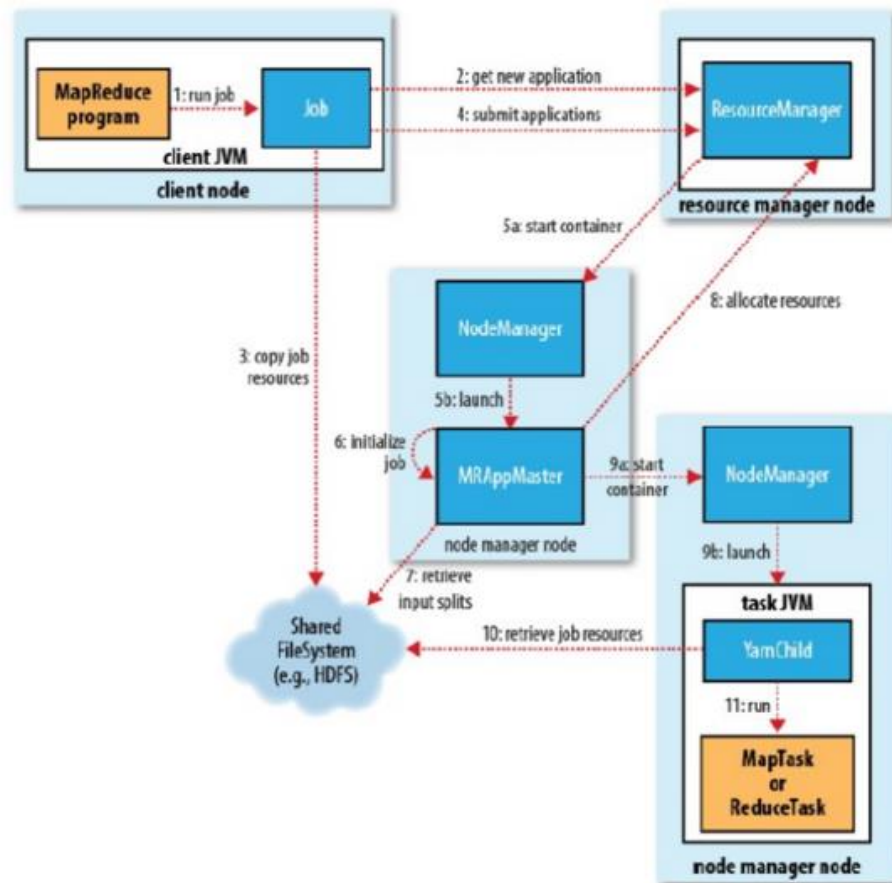
- ApplicationsManager
- 应用程序管理器负责管理整个系统中所有应用程序，包括应用程序提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重新启动它等。
- ApplicationMaster
- 每一个应用都对应着一个AM，AM负责向ResourceManager索要执行任务所需要的资源容器，将任务分配给各个Container去执行，监控各个Container的执行情况。
- Container:
- Container是Yarn中的资源抽象，每个Container包含一定的内存、CPU、磁盘、网络等资源，从而限定每个任务使用的资源量，Yarn会为每个任务分配一个Container。

在YARN上的计算框架



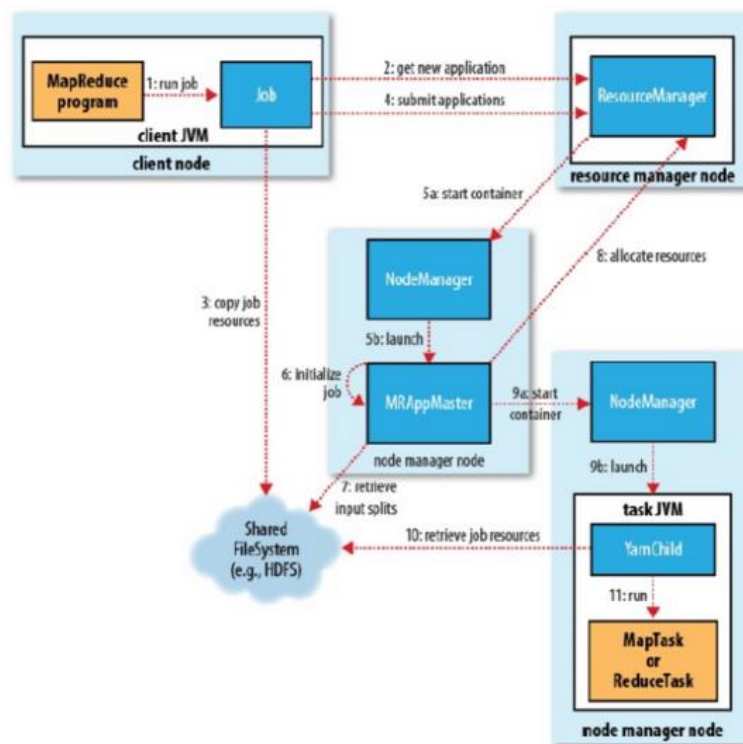
MapReduce on Yarn 工作流程

- 1、启动一个Job
- 2、当RMClient向RM提交MR-App后，RM会将AppID返回给MRClient。
- 3、MRClient检查作业的输出目录、计算输入分片、将作业资源复制到HDFS上。
- 4、客户端MRClient通过调用submitApplication()方法向集群提交作业
- 5、将请求传递给调度器，调度器分配一个容器，container上运行MRAppMaster。
- 6、MRAppMaster进程对整个作业进行初始化
- 7、MRAppMaster进程获取输入数据的分片信息
- 8、MRAppMaster对每个分片申请一个container，按系统指定的Reduce个数，再申请相应数量的container。



MapReduce on Yarn 工作流程

- 9、在申请到一定数量的container后，MRAppMaster内部的scheduler会逐个启动这些container。
- 10、在map或reduce类型的container执行前，scheduler模块需要为这些Container配置完备的上下文信息。
- 11、运行map/reduce任务



MapReduce的任务调度与执行

MapReduce任务由一个主进程MRAppMaster和多个从进程YarnChild共同协作完成。

MRAppMaster主要负责流程控制，通常运行在NodeManager的某个Container上。

MRAppMaster将Mapper和Reduce分配给空闲的Container后，由NodeManager负责启动和监管这些Container。

MRAppMaster还负责监控任务的运行状态，如果某个Container发生故障，MRAppMaster就会将其负责的任务分配给其他空闲的Container重新执行。

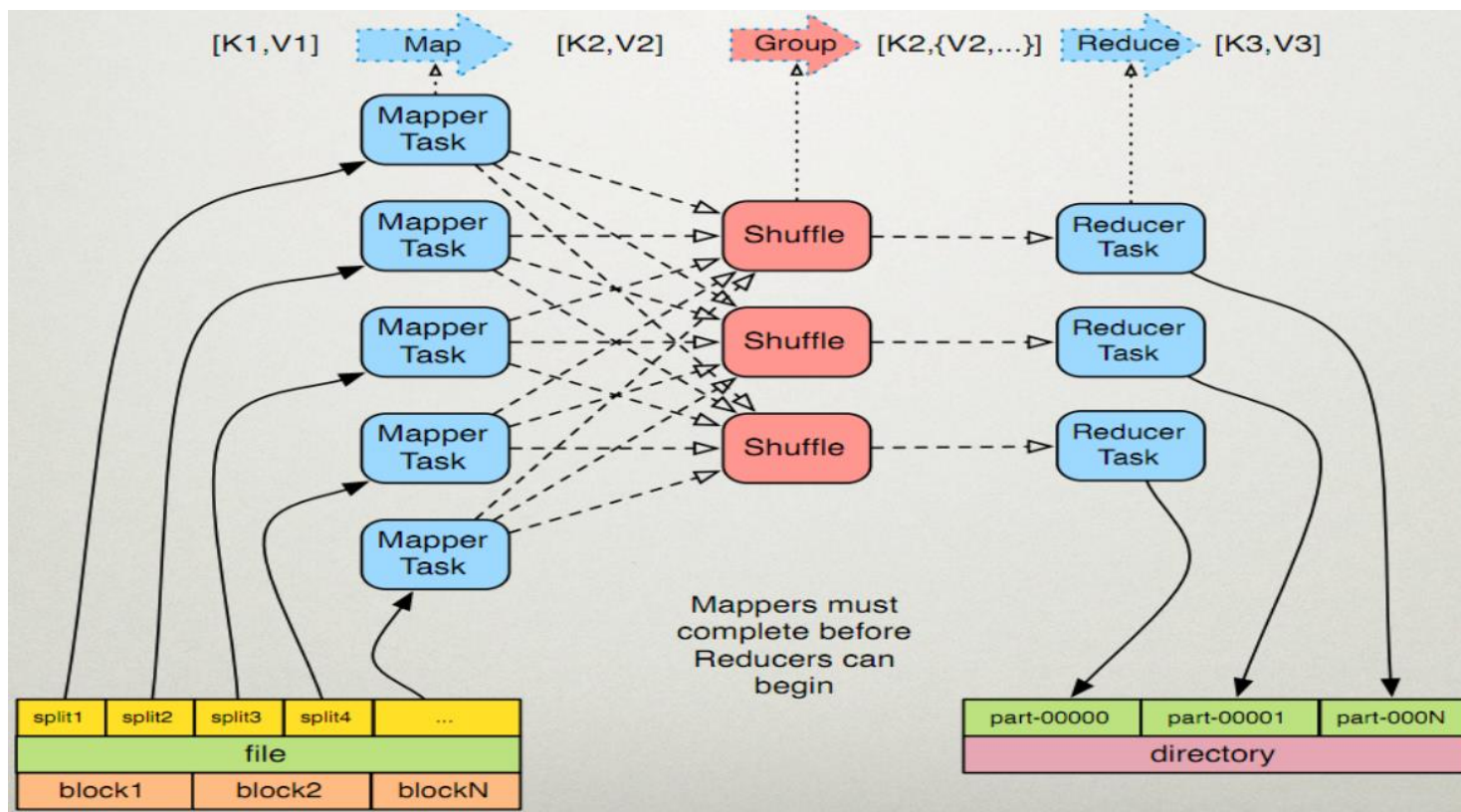
MR V1和MR V2的区别

在MR V2版本中，大部分的API接口都是兼容的保留下来，MR V1中的JobTracker和TaskTracker被替换成相应的Resource Manager、Node Manager。

对比于MR V1中的Task的分配、监控、重启等内容都交由App Master来处理，Resource Manager提供中心服务，负责资源的分配与调度。Node Manager负责维护Container的状态，并将收集的信息上报给Resource Manager，以及负责和Resource Manager维持心跳。

MapReduce的执行过程

- 在MR-App执行过程中，主要包含Data input->Map->Shuffle->Reduce->Store Result阶段

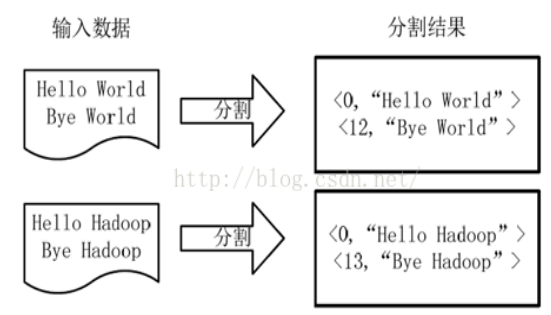


输入分片 (Input Split)

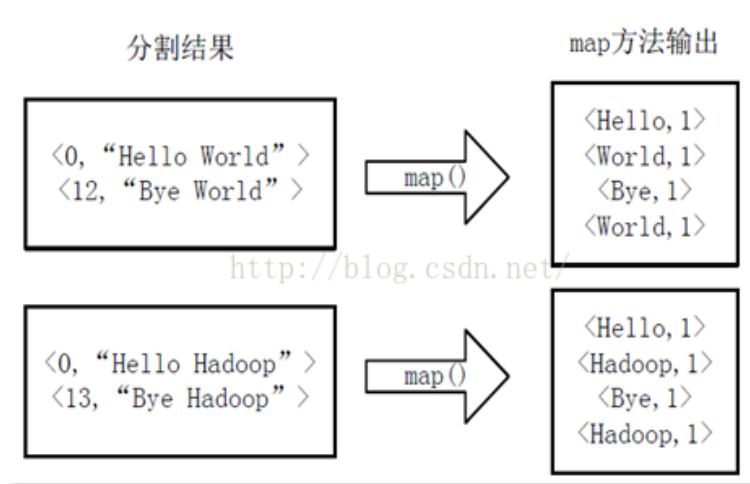
- 在进行map计算之前，mapreduce会根据输入文件计算输入分片（input split），每个输入分片对应一个map任务，输入分片存储的并非数据本身，而是一个分片长度和一个记录数据位置的数组。
- 对于获取的每一个输入文件，根据它的block信息产生输入分片，文件之间不能产生分片。我们可以设置输入分片的数据大小，最小字节数由mapreduce.input.fileinputformat.split.minsize设置，默认是1，最大字节数由mapreduce.input.fileinputformat.split.maxsize设置，默认是Long.MAX_VALUE。
- 由用户定义的分片大小的设置及每个文件block大小的设置，可以计算得分片的大小。
$$\text{Math.max}(\text{minSize}, \text{Math.min}(\text{maxSize}, \text{blockSize}));$$

MapReduce执行过程分解

- 将文件拆分成splits(片), 并将每个split按行分割形成<key,value>对, 这一步由MapReduce框架自动完成, 其中偏移量即key值。

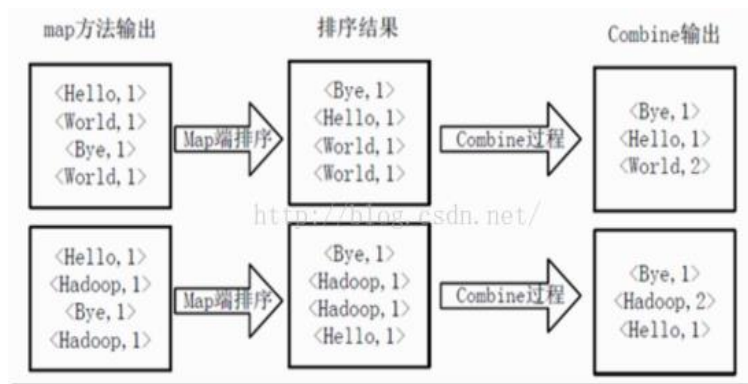


- 将分割好的<key,value>对交给用户定义的map方法进行处理, 生成新的<key,value>对。

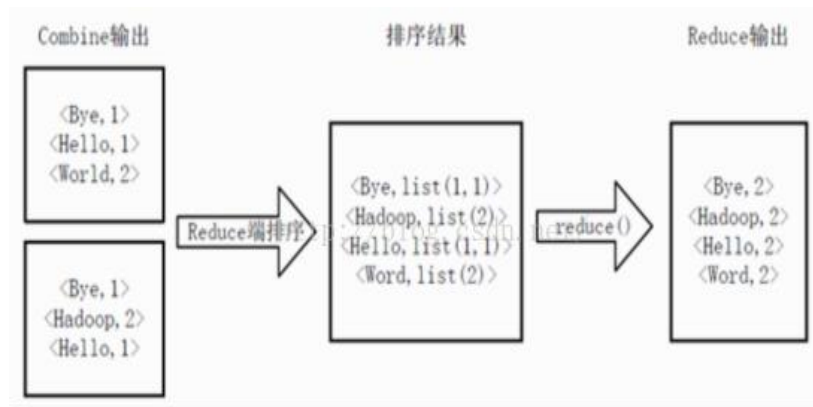


MapReduce执行过程分解

- 得到map方法输出的<key,value>对后，Mapper会将它们按照key值进行排序，并执行Combiner过程，将key相同value值累加，得到Mapper的最终输出结果。



- Reducer先对从Mapper接收的数据进行排序，再交由用户自定义的reduce方法进行处理，得到新的<key,value>对，并作为WordCount的输出结果。



Yarn自带的web接口

- Web接口主要为管理员使用，只支持读操作，不支持写操作。
- 通过Web接口，管理员可以查看集群统计信息、调度器、应用程序列表等功能。
- Yarn web的默认地址：RMIP: 8080



Logged in as: dr.who

All Applications

Cluster

[About](#)
[Nodes](#)
[Node Labels](#)
[Applications](#)
[NEW](#)
[NEW SAVING](#)
[SUBMITTED](#)
[ACCEPTED](#)
[RUNNING](#)
[FINISHED](#)
[FAILED](#)
[KILLED](#)
[Scheduler](#)

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	1	0	19	20 GB	24 GB	0 B	19	24	0	3	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:8>

Show 20 entries

Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes
application_1496627918377_0001	hadoop	word count	MAPREDUCE	default	Mon Jun 5 16:17:15 +0800 2017	N/A	RUNNING	UNDEFINED	<div></div>	ApplicationMaster	0

Showing 1 to 1 of 1 entries

First Previous 1 Next Last

MapReduce Web接口

- MapReduce Web的默认地址是：jobhistoryIP:19888



MapReduce Job job_1497057336012_0001

Logged in as: dr.who

Application

Job

- Overview
- Counters
- Configuration
- Map tasks
- Reduce tasks

Tools

		Job Overview
Job Name:	word count	
User Name:	hadoop	
Queue:	default	
State:	SUCCEEDED	
Uberized:	false	
Submitted:	Sat Jun 10 09:18:30 CST 2017	
Started:	Sat Jun 10 09:19:14 CST 2017	
Finished:	Sat Jun 10 09:20:21 CST 2017	
Elapsed:	1mins, 7sec	
Diagnostics:		
Average Map Time	40sec	
Average Shuffle Time	20sec	
Average Merge Time	0sec	
Average Reduce Time	0sec	

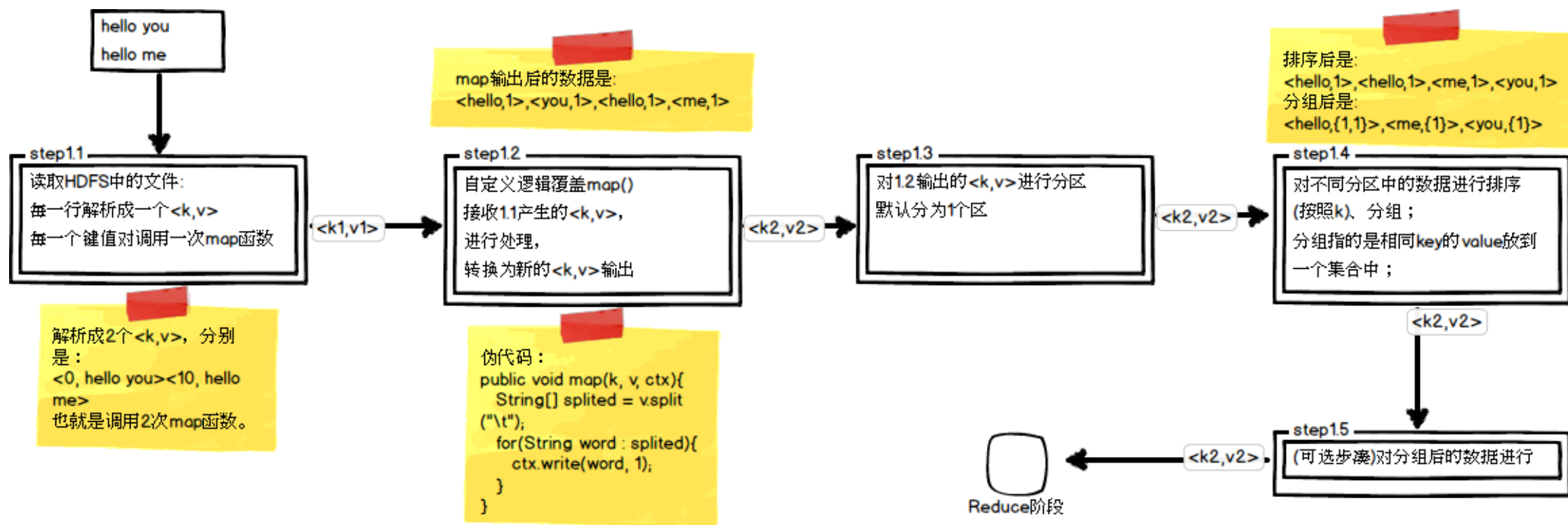
ApplicationMaster			
Attempt Number	Start Time	Node	Logs
1	Sat Jun 10 09:18:44 CST 2017	slave2:8042	logs

Task Type	Total		Complete	
Map	28		28	
Reduce	1		1	
Attempt Type	Failed		Killed	Successful
Maps	0	0		28
Reduces	0	0		1

Combiner

- 如果有10亿个数据，Mapper会生成10亿个键值对在网络间进行传输，但如果我们只是对数据求最大值，那么很明显的Mapper只需要输出它所知道的最大值即可。这样做不仅可以减轻网络压力，同样也可以大幅度提高程序效率。
- 每一个map都可能会产生大量的本地输出，Combiner的作用是把一个map产生的多个<KEY,VALUE>合并成一个新的<KEY,VALUE>,然后再将新<KEY,VALUE>作为reduce的输入，以减少在map和reduce节点之间的数据传输量，以提高网络IO性能。
- Combiner适用场景
 - 对记录汇总的场景（如求和）
 - 求最大值、最小值
- 不适用场景
 - 求平均数
- 在实际的Hadoop集群操作中，由多台主机一起进行MapReduce，如果加入规约操作，每一台主机会在reduce之前进行一次对本机数据的规约，然后再通过集群进行reduce操作，这样就会大大节省reduce的时间，从而加快MapReduce的处理速度

Partitioner操作



Partitioner操作

- Mapper最终处理的键值对<key, value>，是需要送到Reducer去合并的，合并的时候，有相同key的键/值对会送到同一个Reducer节点中进行归并。
- 当存在多个Reducer任务时，由Partitioner决定Reducer的分配过程。
- 用户在中间key上使用分区函数来对数据进行分区
- 在一些集群应用中，例如分布式缓存集群中，缓存的数据大多都是靠哈希函数来进行数据均匀分布的。
- 默认的分区函数使用hash方法（比如常见的： $\text{hash}(\text{key}) \bmod R$ ）进行分区。hash方法能够产生非常平衡的分区。

默认的分区分类HashPartitioner

- Hadoop中自带了一个默认的分区分类HashPartitioner，它继承了Partitioner类，提供了一个getPartition的方法

```
public class HashPartitioner<K2, V2> implements Partitioner<K2, V2> {  
    public void configure(JobConf job) {}  
  
    /** Use {@link Object#hashCode()} to partition. */  
  
    public int getPartition(K2 key, V2 value,int numReduceTasks)  
    {  
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;  
    }  
}
```

自定义Partitioner

```
public static class MyPartition extends Partitioner<Text, IntWritable>{  
  
    public int getPartition(Text key, IntWritable value, int num) {  
  
        .....  
  
    }  
}
```

- 添加设置Partitioner的代码:

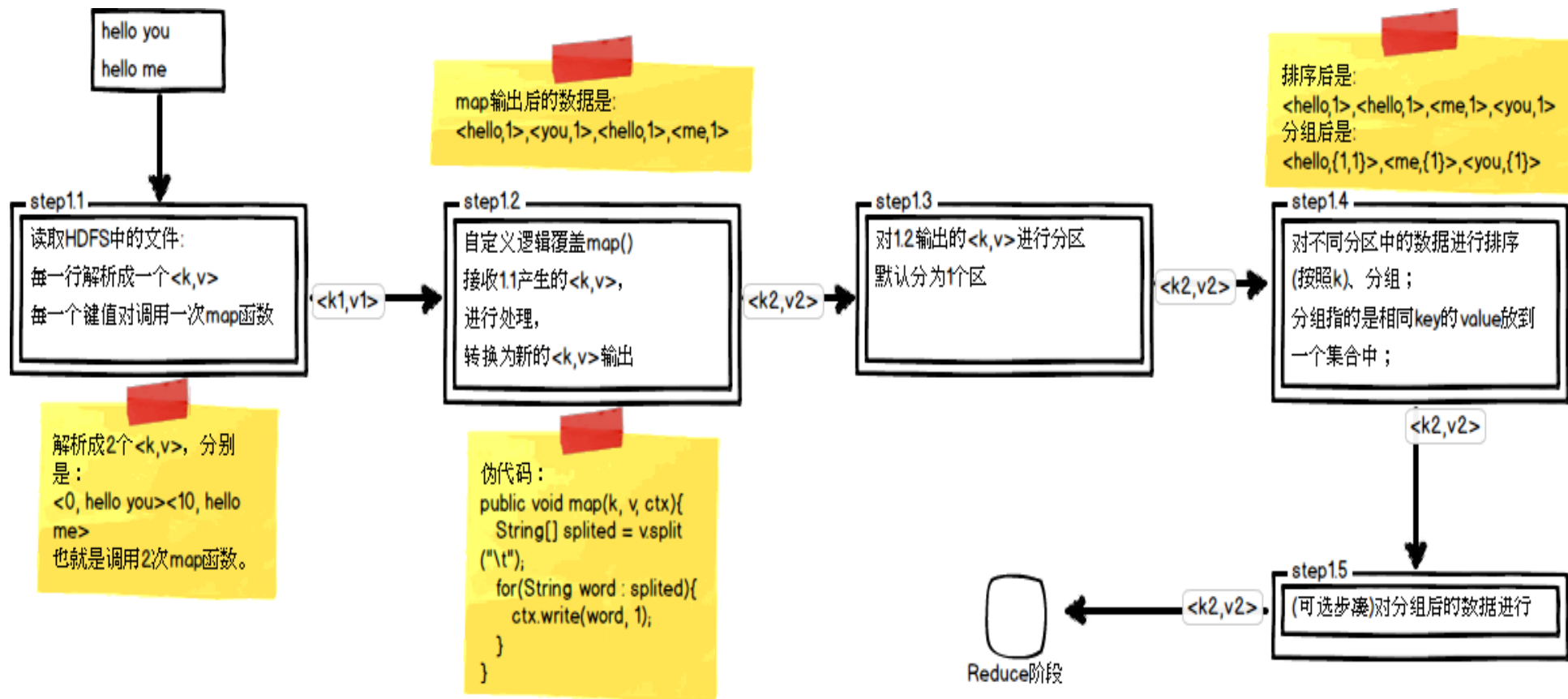
```
job.setPartitionerClass(MyPartition.class);  
  
job.setNumReduceTasks(3);
```

- 分区Partitioner主要作用:

- 根据业务需要, 产生多个输出文件
- 多个reduce任务并发运行, 提高整体job的运行效率

排序和分组

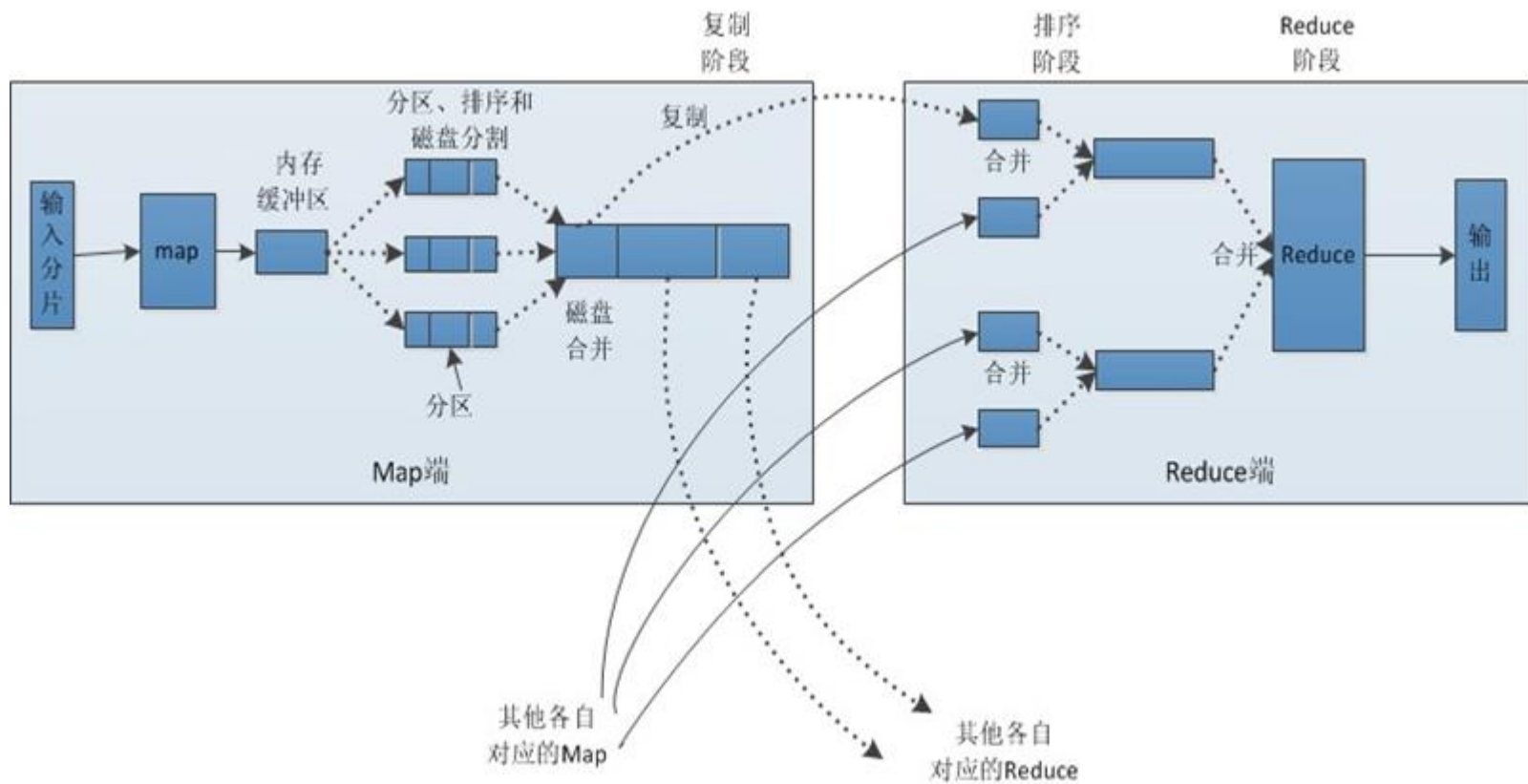
- Step1.4第四步中需要对不同分区中的数据进行排序和分组，默认情况按照key进行排序和分组



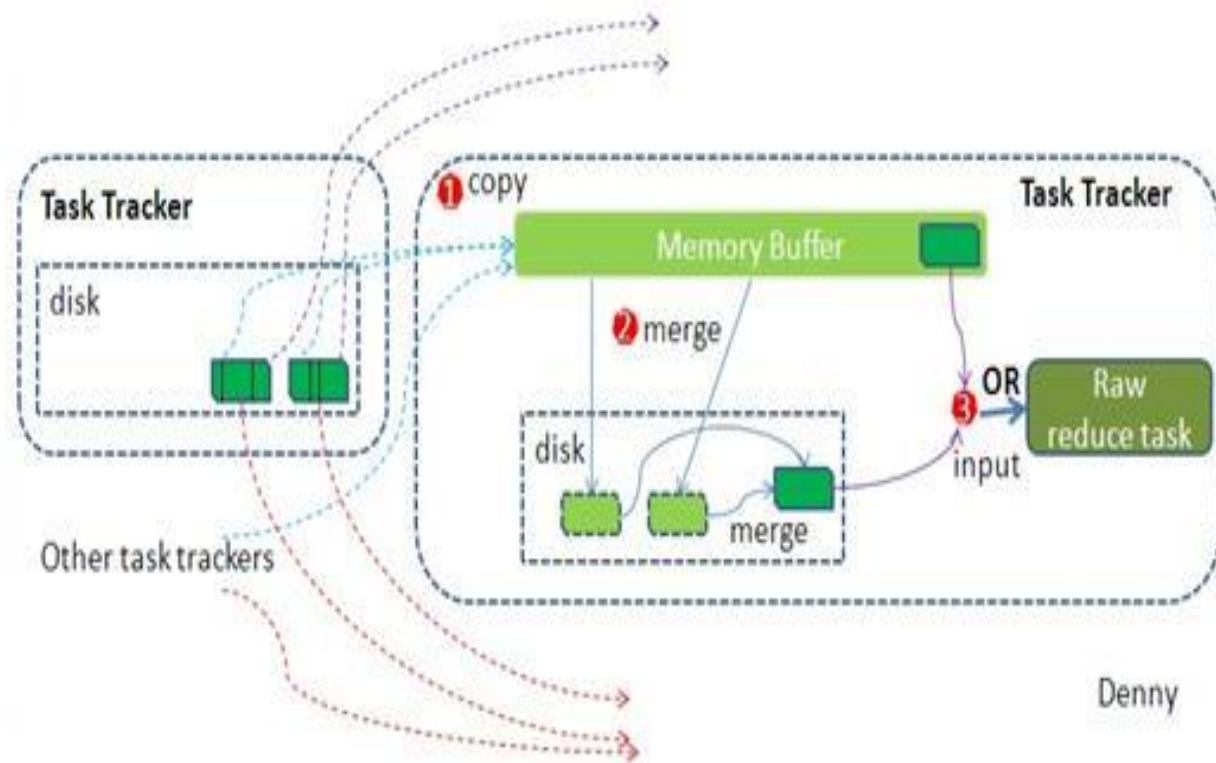
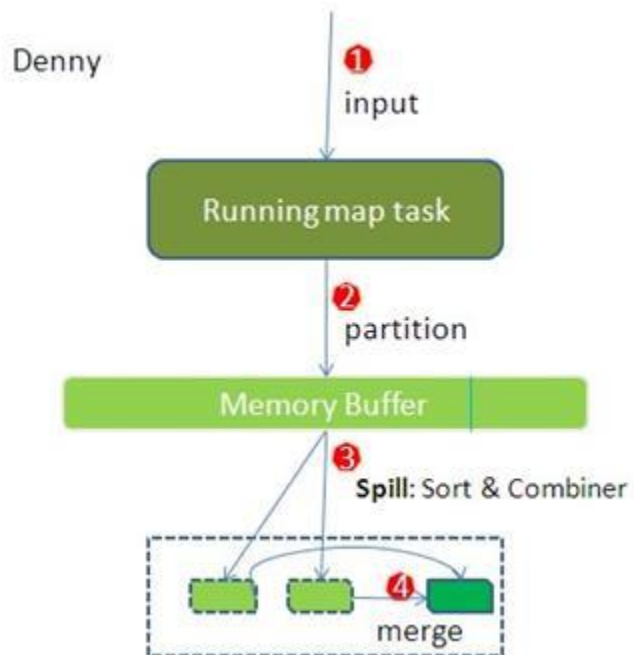
排序

- 在Hadoop默认的排序算法中，只会针对key值进行排序
- 任务：数据文件中，按照第一列升序排列，当第一列相同时，在对第二列升序排列。
- Hadoop中的默认分组规则是基于Key进行的，会将相同key的value放到一个集合中去。
- 任务：求出第一列相同时第二列的最小值
- // 设置自定义分组规则
- `job.setGroupingComparatorClass(MyGroup.class);`

shuffle过程



Map端和Reduce端



课程内容

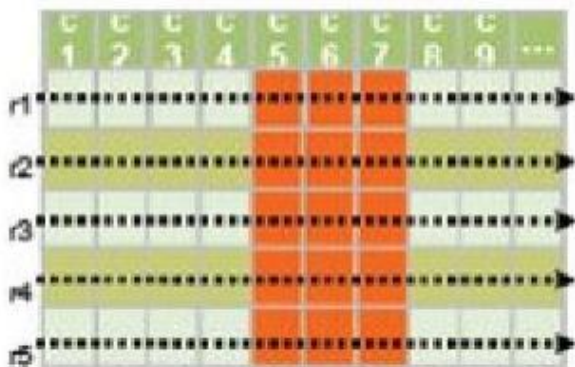
- Hadoop集群
- HDFS
- MapReduce
- Hbase, Hive, Sqoop

Hbase

- HBase是Hadoop Database，是基于Google BigTable模型开发的，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，是Apache Hadoop生态系统中的重要一员；
- Hbase是运行在Hadoop上的NoSQL数据库
- 利用HBase技术可在廉价PC Server上搭建起大规模存储集群。
- 从逻辑上讲，HBase将数据按照表、行和列进行存储

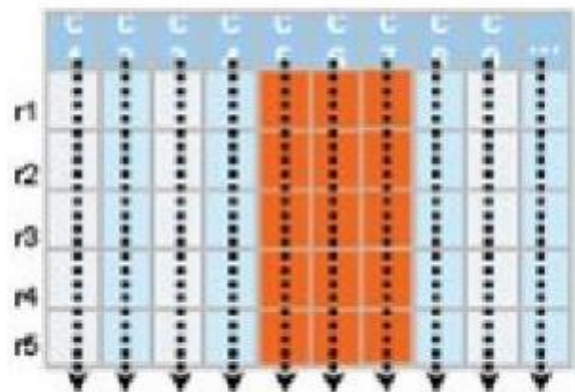
行存储与列存储

传统行式数据库



- 数据是按行存储的
- 没有索引的查询使用大量I/O
- 建立索引和物化视图需要花费大量时间和资源
- 面向查询的需求，数据库必须被大量膨胀才能满足性能要求

列式数据库



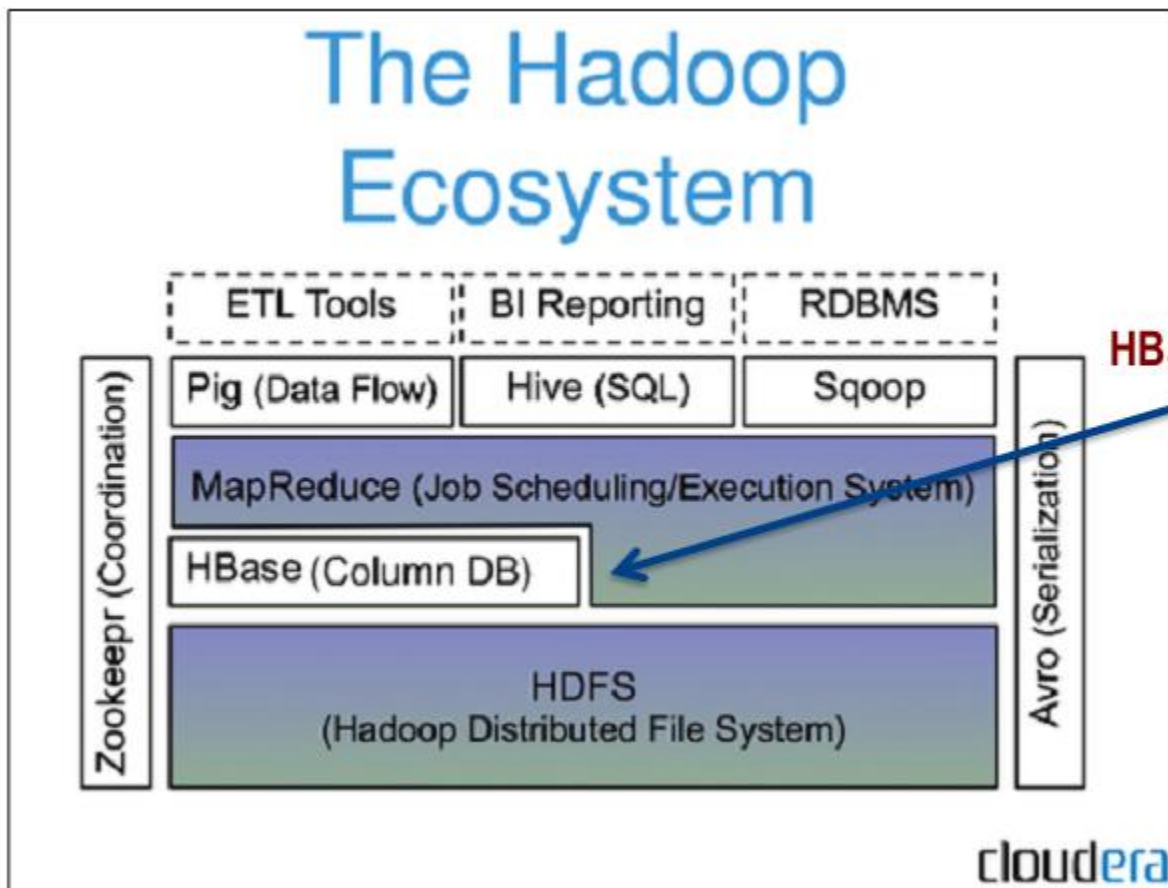
- 数据是按列存储-每一列单独存放
- 数据即是索引
- 指访问查询涉及的列-大量降低系统I/O
- 每一列由一个线索来处理-查询的并发处理
- 数据类型一致，数据特征相似-高效压缩

特点

- 数据量大：一个表可以有上亿行，上百万列；
- 无模式：每行都有一个可排序的主键和任意多的列，列可以根据需要动态的增加，同一张表中不同的行可以有截然不同的列；
- 面向列：面向列（族）的存储和权限控制，列（族）独立检索；
- 稀疏：对于空（null）的列，并不占用存储空间，表可以设计的非常稀疏；
- 数据多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳；
- 强一致性：同一行数据的读写只在一台region server上执行

特点

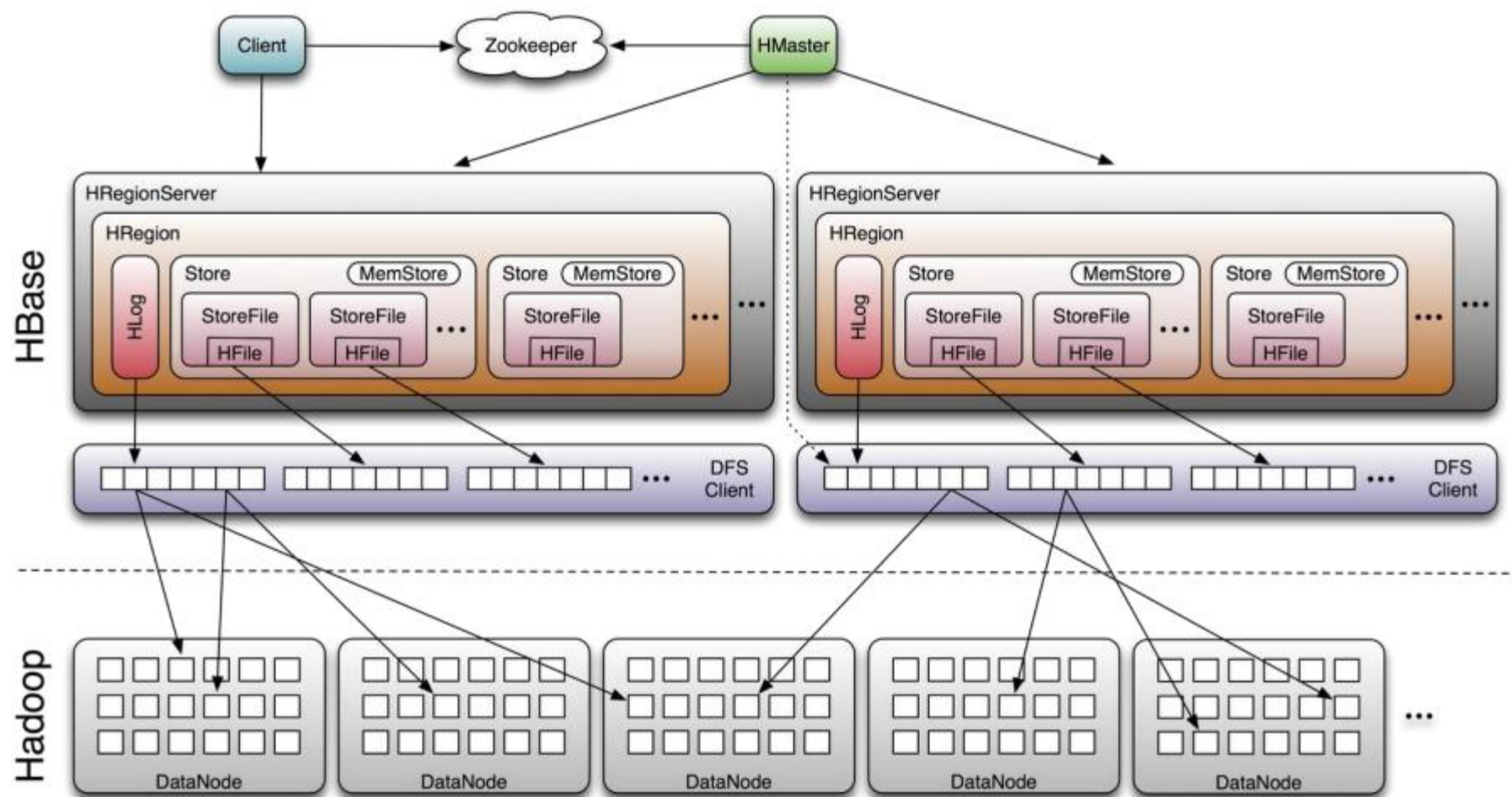
- 需要对数据进行随机读操作或者随机写操作；
- 大数据上高并发操作，比如每秒对**PB**级数据进行上千次操作；
- 读写访问均是非常简单的操作



HBase 构建在HDFS之上

Hbase内部管理的
文件全部存储在
HDFS中

HBase基本架构



与Zookeeper的关系

- Hbase中包含3个主要组件：客户端库、一台主服务器、多台region服务器。
- Zookeeper是一个可靠的、高可用的、持久化的分布式协调系统。它提供了类似文件系统一样的访问目录和文件的功能，通常分布式系统利用它协调所有权、注册服务、监听更新。
- 每台region服务器在Zookeeper中注册一个自己的临时节点，主服务器会利用这些临时节点来发现可用服务器，还可以利用临时节点来跟踪及其故障和网络分区。这些临时节点相当于一个“会话”，会话是客户端链接上Zookeeper服务器之后自动生成的。每个会话有一个唯一的id，客户端会用这个id不断向Zookeeper服务器发送“心跳”，一旦客户端发生故障，发送心跳则会停止，当超过限定时间后，Zookeeper服务器会判定会话超时，并自动删除属于它的临时会话。
- Hbase还可以利用Zookeeper确保只有一个主服务器在运行，存储用于发现region的引导位置，作为一个region服务器的注册表，以及实现其他目的。

与Zookeeper的关系

- HDFS
 - 数据以文件方式直接存储
 - 连续访问效率高
 - 随机访问效率低
- HBase
 - 数据以索引格式特殊存储
 - 连续访问效率低
 - 随机访问效率高

Hive

- Hive是基于Hadoop的数据仓库平台，由Facebook贡献，Hive提供了一系列的工具，可以用来进行数据提取、转化、加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。
- Hive 定义了简单的类 SQL 查询语言，称为 HQL，hive设计目的是让SQL技能良好，但Java技能较弱的分析师可以查询海量数据。
- Hive是SQL解析引擎，它将SQL语句转译成M/R Job，可以认为hive是HQL到MR的语言翻译器。Hive中数据计算使用MR，数据存储使用HDFS
- Hive的HQL表达能力有限，有些复杂运算还需要编写MR程序，hive允许熟悉 MapReduce 开发人员开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

Hadoop的缺点

- 过于底层
 - 不够灵活，受语言约束笨重不堪
 - 数据操作很不清晰，代码量大，难维护
 - 常见操作如排序、连接繁琐，且不高效
- 缺乏高级抽象
 - 程序关注点是如何实现，而不是要实现什么
 - 大量时间浪费在调试无用的细节上
- 需要高层工具，通过编译转化为MapReduce任务交给Hadoop执行
 - Hive: 使用类似SQL的语法
 - Pig: 使用PigLatin脚本

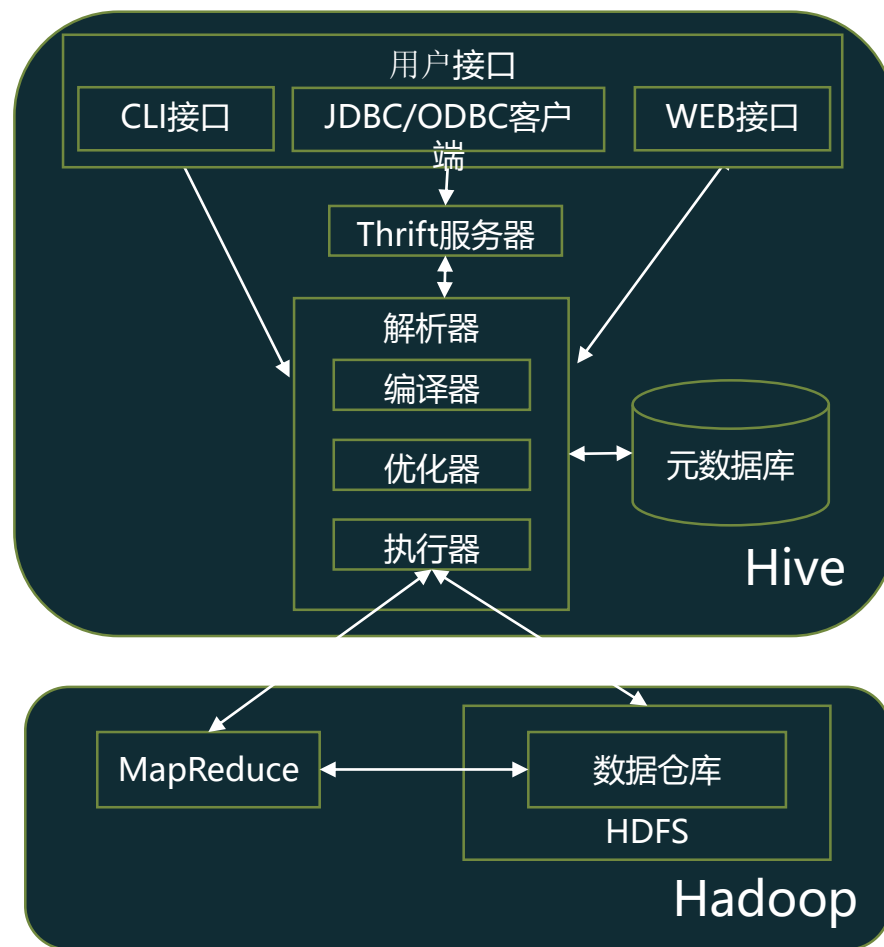
生态图中的Hive

- 适用场景
 - 海量数据的存储处理
 - 数据挖掘
 - 海量数据的离线分析
- 不适用场景
 - 复杂的机器学习算法
 - 复杂的科学计算
 - 联机交互式实时查询



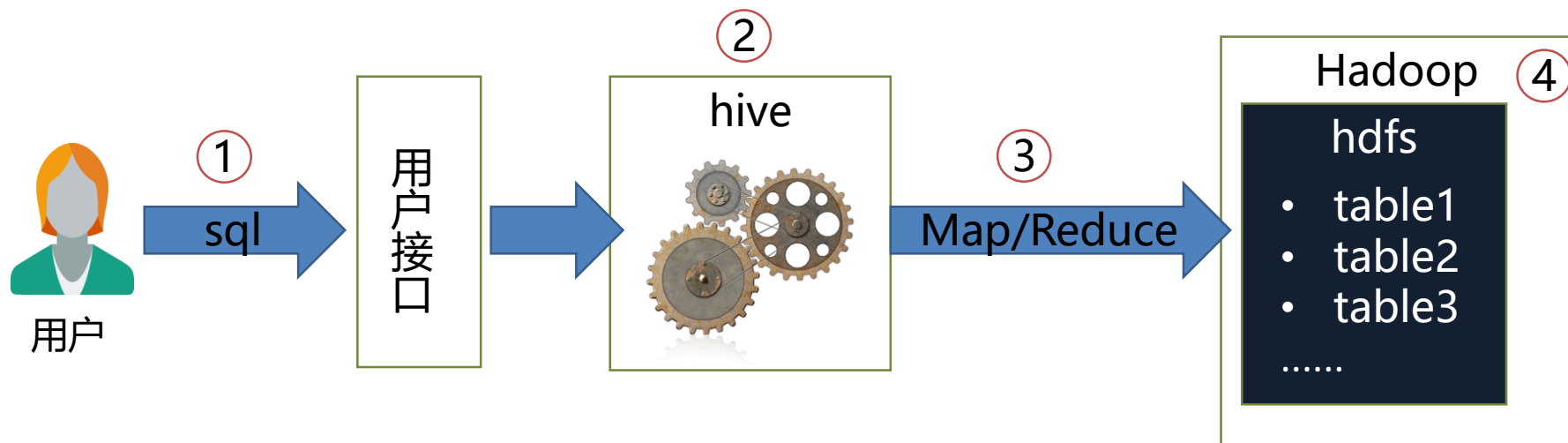
Hive架构

- 用户接口
包括 CLI, JDBC/ODBC, WebUI
- 元数据库
元数据用于存放Hive库的基础信息，它存储在关系数据库中，如 mysql、derby。元数据包括：数据库信息、表的名字，表的列和分区及其属性，表的属性，表的数据所在目录等。
- 解析器
编译器、优化器、执行器
- Hadoop
用 MapReduce 进行计算，用 HDFS 进行存储

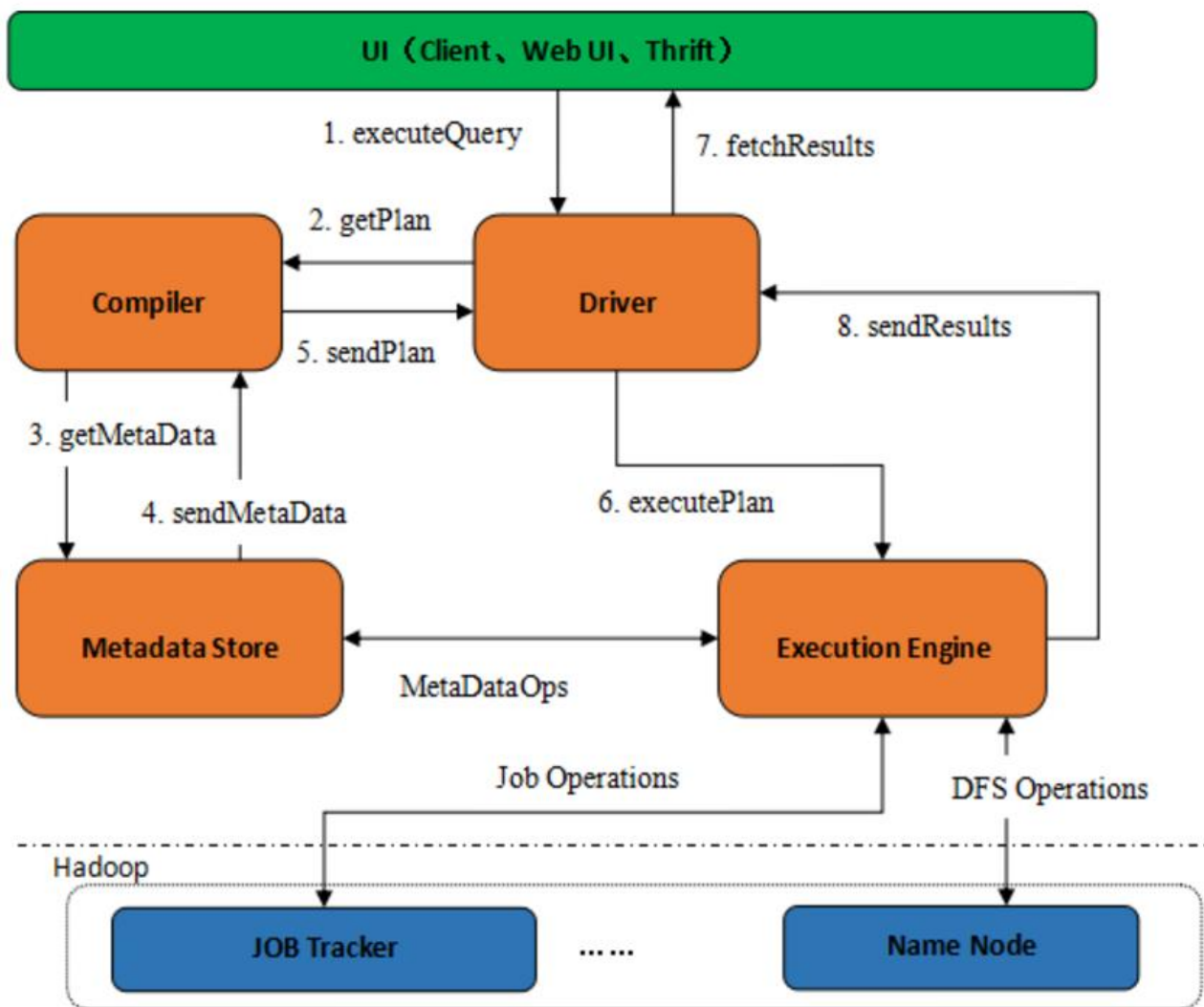


Hive架构

- ① 用户通过用户接口连接Hive,发布Hive SQL
- ② Hive解析查询并制定查询计划
- ③ Hive将查询转换成MapReduce作业
- ④ Hive在Hadoop上执行MapReduce作业



运行机制



Hive 和 Hadoop 关系

- Hive 构建在 Hadoop 之上
- HQL 中对查询语句的解释、优化、生成查询计划是由 Hive 完成的
- 所有的数据都是存储在 Hadoop 中
- 查询计划被转化为 MapReduce 任务，在 Hadoop 中执行
- Hadoop和Hive都是用UTF-8编码的

Hive 和 Hadoop 关系

- Hive 构建在 Hadoop 之上
- HQL 中对查询语句的解释、优化、生成查询计划是由 Hive 完成的
- 所有的数据都是存储在 Hadoop 中
- 查询计划被转化为 MapReduce 任务，在 Hadoop 中执行
- Hadoop和Hive都是用UTF-8编码的

Hive 和传统关系数据库

	Hive	RDBMS
查询语言	HQL	SQL
数据存储	HDFS	Raw Device or Local FS
数据更新	不支持	支持
索引	新版本有，但较弱	有
执行	MapReduce	Excutor
执行延迟	高	低
可扩展性	高	低
数据规模	大	小

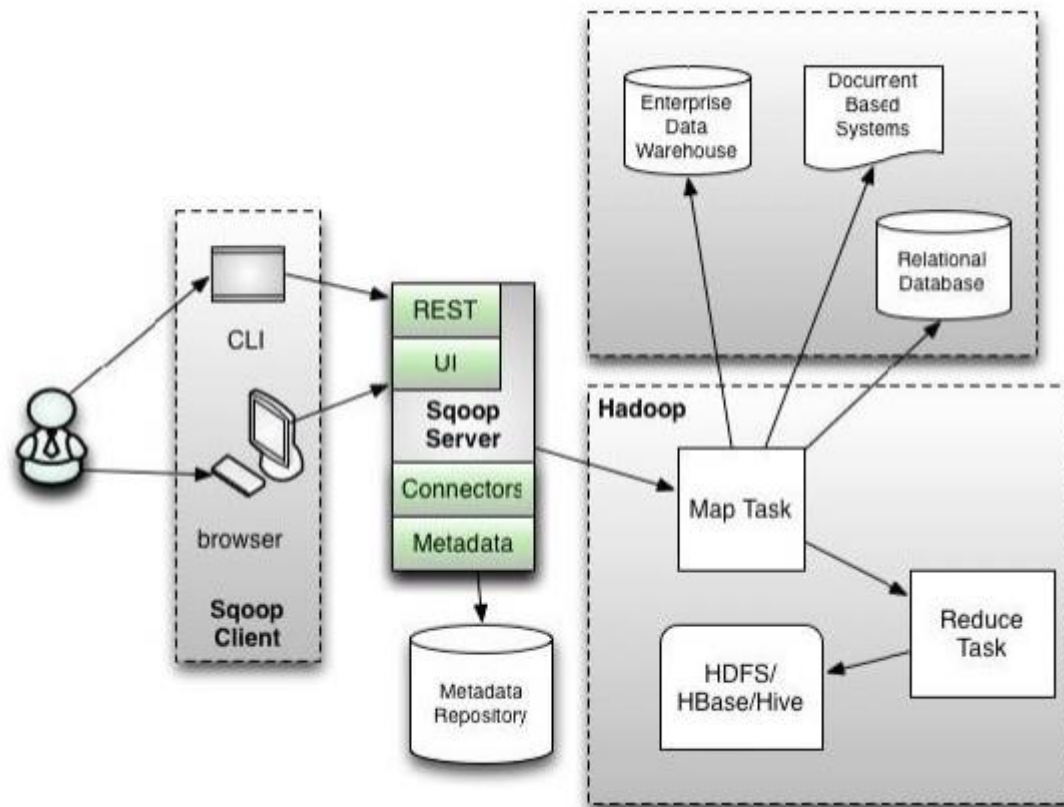
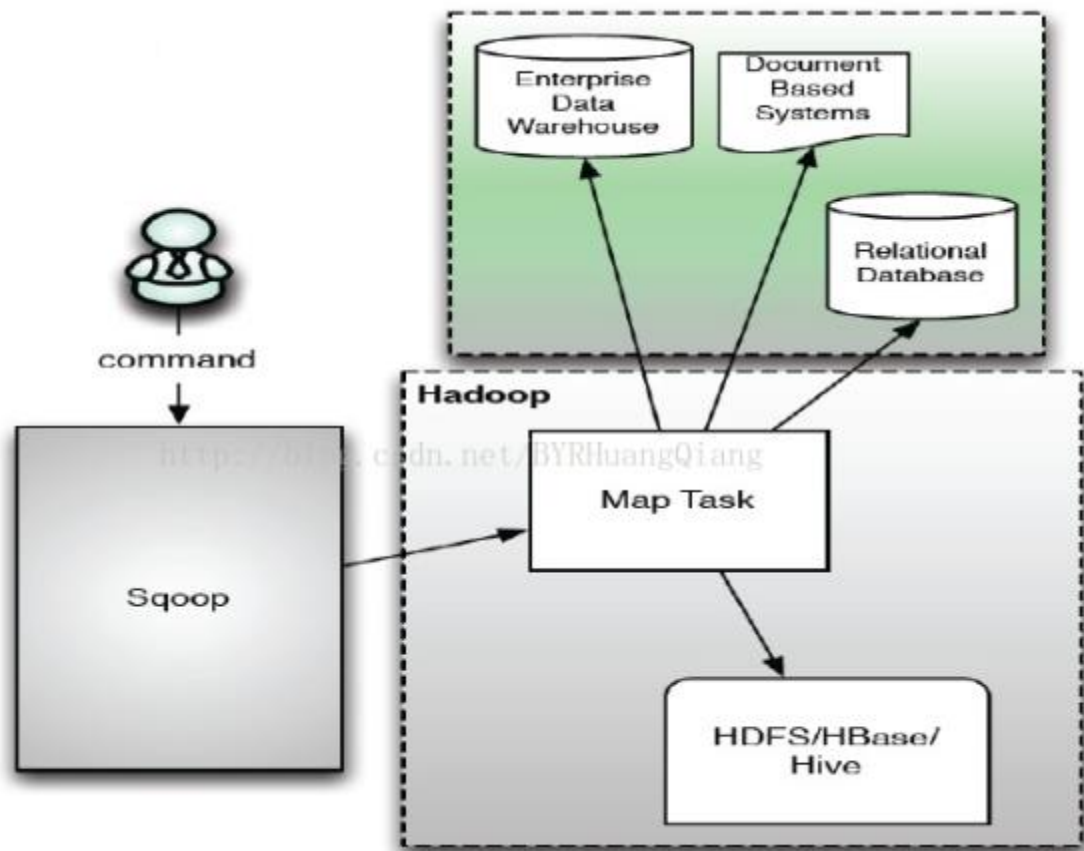
Sqoop 简介

- Sqoop 即 SQL to Hadoop，是一款方便的在传统型数据库与 Hadoop 之间进行数据迁移的工具，充分利用 MapReduce 并行特点以批处理的方式加快数据传输，发展至今主要演化了二大版本，Sqoop1 和 Sqoop2。
- Sqoop 工具是 hadoop 下连接关系型数据库和 Hadoop 的桥梁，支持关系型数据库和 hive、hdfs、hbase 之间数据的相互导入，可以使用全表导入和增量导入。
- 高效可控的利用资源，任务并行度，超时时间。
- 数据类型映射与转化，可自动进行，用户也可自定义
- 支持多种主流数据库，MySQL, Oracle, SQL Server, DB2 等等

Sqoop1 和 Sqoop2 比较

- 两个不同的版本，完全不兼容
- 版本号划分区别
 - Apache 版本： 1.4.x(Sqoop1); 1.99.x(Sqoop2)
 - CDH 版本 :Sqoop-1.4.3-cdh4(Sqoop1) ; Sqoop2-1.99.2-cdh4.5.0 (Sqoop2)
- Sqoop2 比 Sqoop1 的改进
 - 引入 Sqoop server，集中化管理 connector 等
 - 多种访问方式： CLI, Web UI, REST API
 - 引入基于角色的安全机制

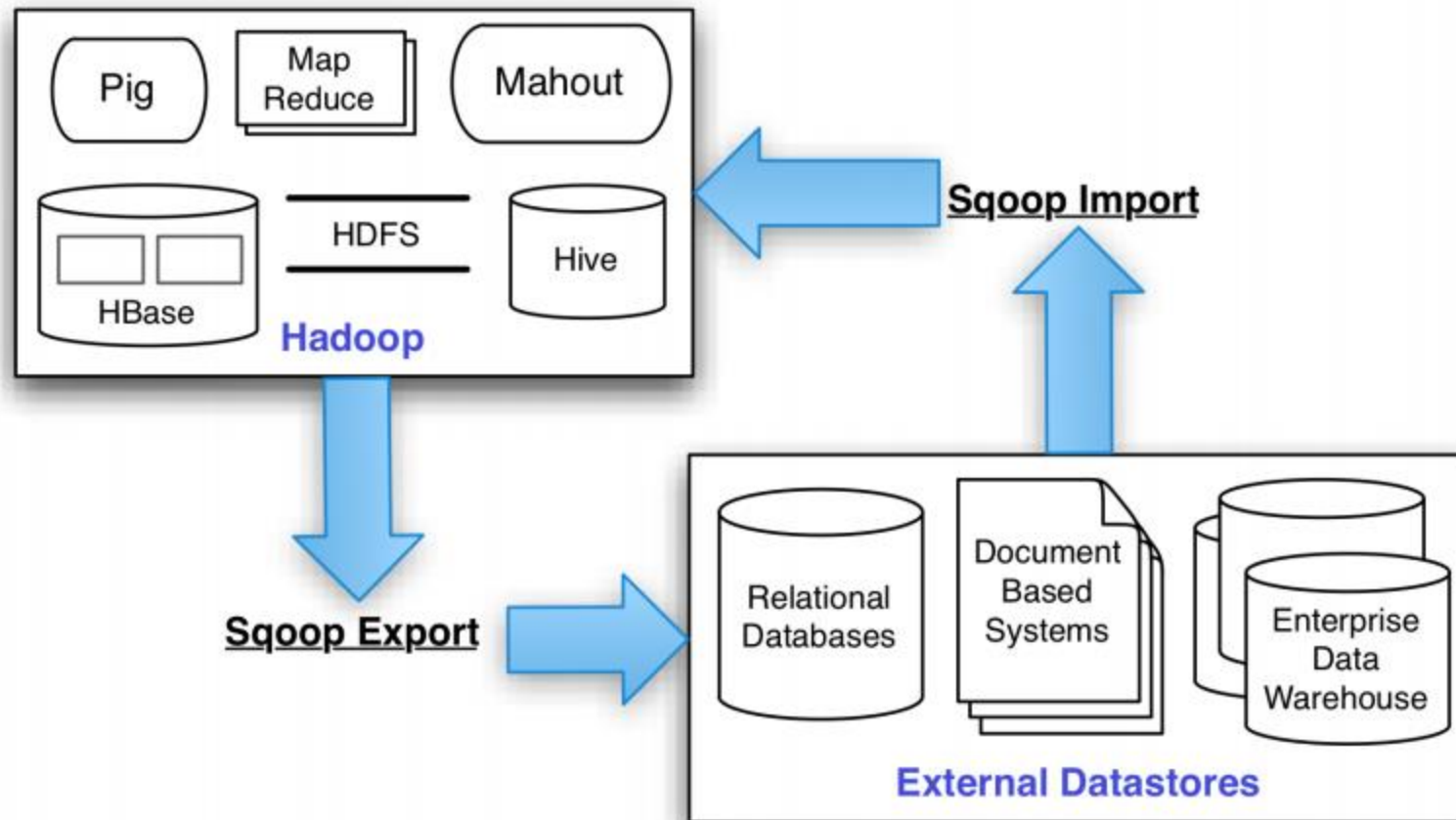
架构



Sqoop1 与 Sqoop2 的优缺点

	Sqoop1	Sqoop2
架构	仅仅使用一个 Sqoop 客户端	引入了 Sqoop server 集中化管理 connector, 以及 rest api、web UI, 并引入权限安全机制
部署	部署简单, 安装需要 root 权限, connector 必须符合 JDBC 模型	架构稍复杂, 配置部署更繁琐
使用	命令行方式容易出错, 格式紧耦合, 无法支持所有数据类型, 安全机制不够完善, 例如密码暴漏	多种交互方式, 命令行, web UI, rest API, connector 集中化管理, 所有的链接安装在Sqoop server 上, 完善权限管理机制, connector 规范化, 仅仅负责数据的读写

Sqoop导入与导出



Sqoop从数据库导入到HDFS的原理

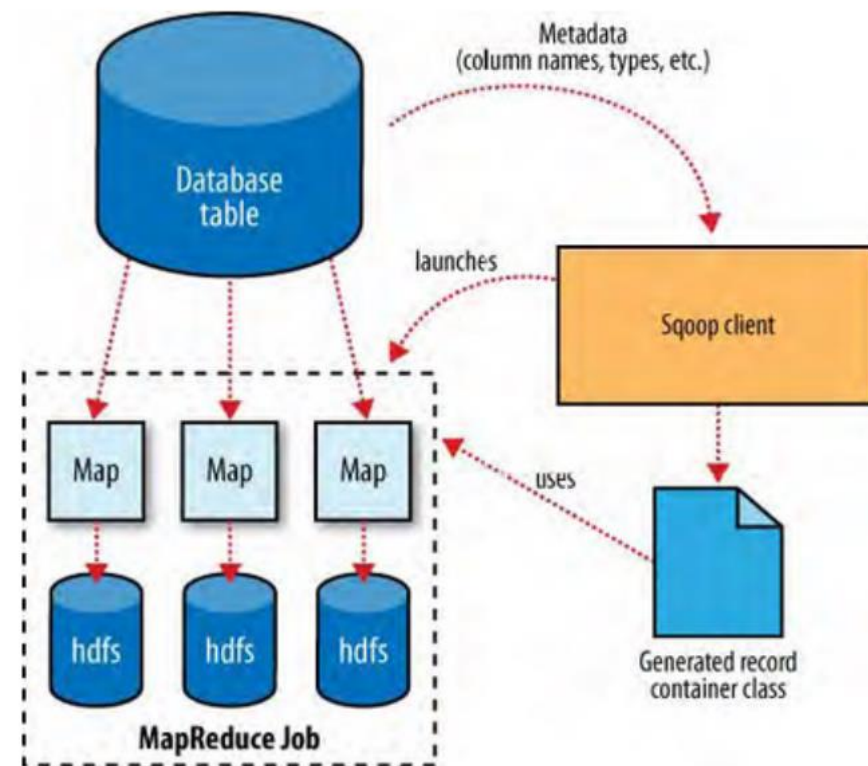
在导入开始之前，Sqoop使用JDBC来检查将要导入的表，检索出表中所有的列以及列的SQL数据类型，这些SQL类型（VARCHAR、INTEGER）被映射到Java数据类型（String、Integer等），在MapReduce应用中将使用这些对应的java类型来保存字段的值。

Sqoop的代码生成器使用这些信息来创建对应表的类，用于保存从表中抽取的记录。

JDBC的ResultSet接口提供了检索记录的游标

Sqoop启动的MapReduce作业，在生成反序列化代码和配置InputFormat之后，Sqoop将作业发送到MapReduce集群。

Map任务将执行查询并将ResultSet中的数据反序列化到生成类的实例，写入到HDFS中。





华信培训