# Hadoop-分布式系统基础架构

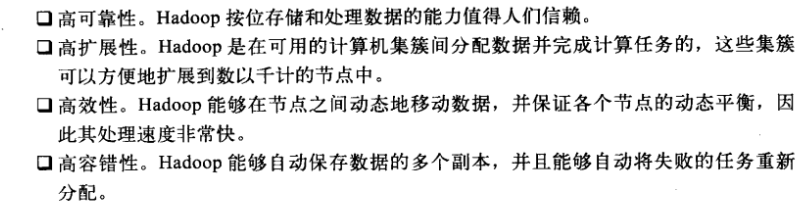
网易云课堂《大数据开发Hadoop全集视频教程》

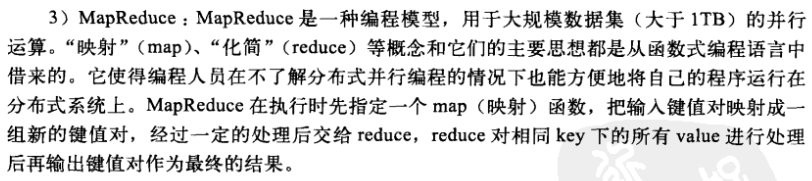
目标：海量数据的存储(HDFS-分布式文件系统)和海量数据的分析计算(MapReduce-并行编程)问题

优势：

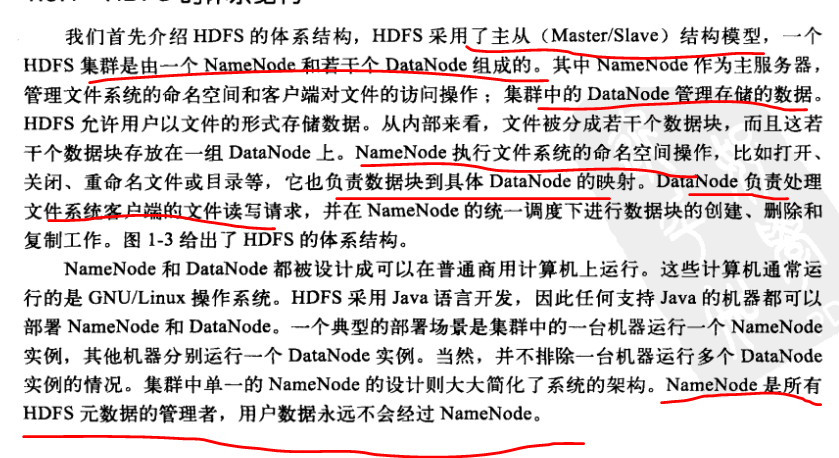
高可靠性：底层维护多个数据副本(3个)

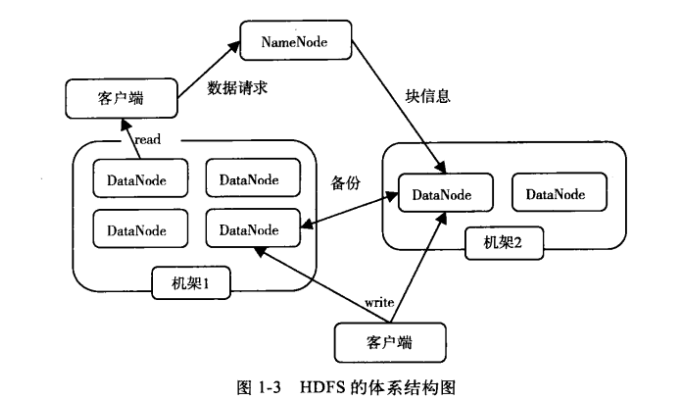
高扩展性：在集群间分配任务数据，可方便的扩展数以千计的节点



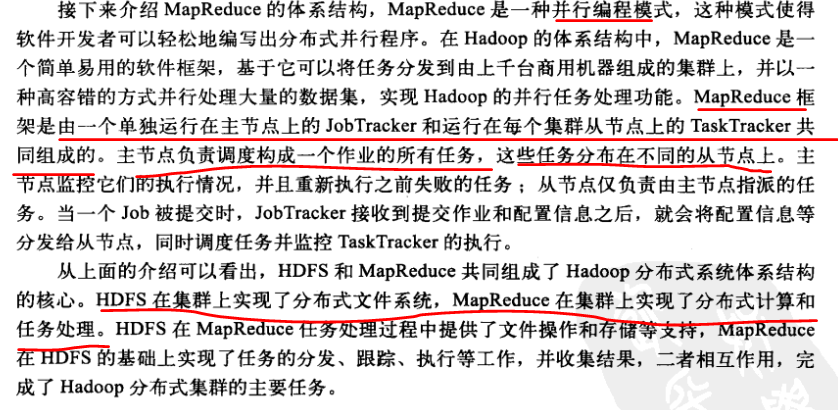


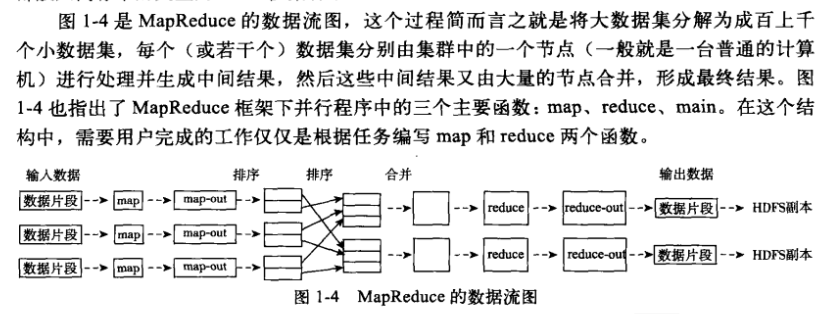
## HDFS体系架构



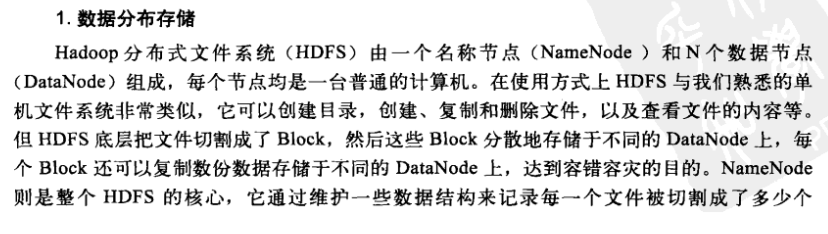


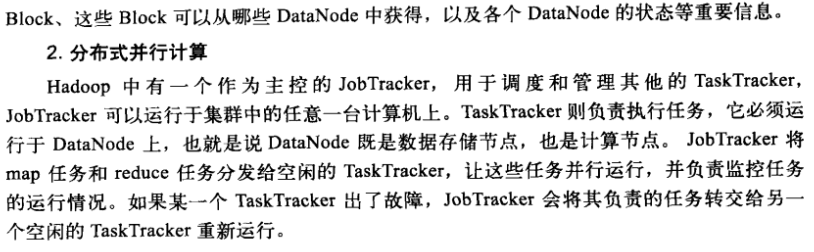
## MapReduce体系架构



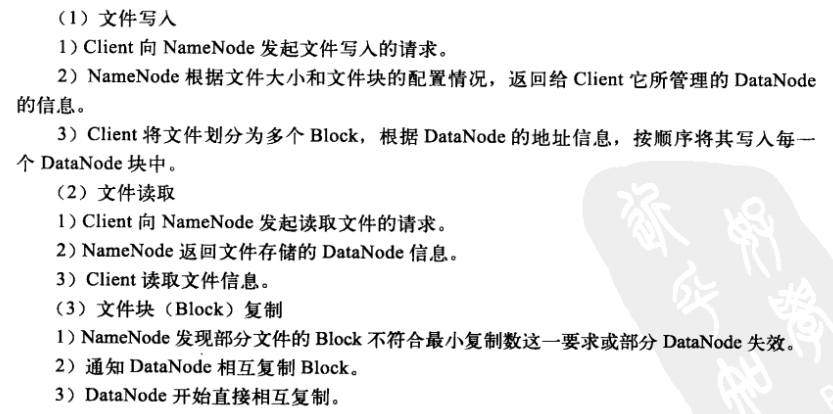


简单原理：





## HDFS对数据的管理



**Hadoop模块：**  
 common: 工具类 rpc框架->远程过程调用

**hdfs: 将文件切分为小文件存储在不同节点**

架构：主从架构

主：namenode(名称节点) 一个集群唯一，负责分配调度一个作业的所有任务

**Namenode作用**

1. 存储元数据：抽象目录树、存储数据和block块的对应关系、block存储的位置
2. 处理客户端读写请求

Hdfs目录结构和linux操作系统的类似，以/为根节点，其目录结构代表的是所有数据节点的抽象出来的目录，不代表任何一个节点。

从：datanode(数据节点) 负责存储数据及计算

**Datanode作用：**

1. 负责真正的数据存储，存储数据的block
2. 真正处理读写

助理：secodarynamenode 负责分担主节点压力，当namenode宕机时secondarynamenode不能主动切换为namenode,但可以手动切换，但secondarynamenode中存储的数据和namenode中相同(基本相同)

**Secondarynamendoe作用：**

1. namenode宕机时帮助namenode恢复
2. 帮助namenode进行元数据合并

**hdfs优点:**

1. 可构建在廉价机上，成本低
2. 高容错性->保存副本
3. 适合批处理(离线数据处理，非实时数据处理)
4. 适合大数据处理
5. 流式文件访问(一次性写入、多次读取)，不支持数据修改

**hdfs缺点：**

1. 不支持实时/近实时数据访问
2. 不擅长存储大量的小文件(kb级别)
3. 寻址时间太长，可能会大于读取数据的时间(需要先访问元数据，元数据和block对应)
4. 会造成元数据存储量过大，增加namenode(集群中唯一)的压力
5. 不支持文件内容修改(因为修改时必须修改多个副本，不如重新写入)

#### hdfs命令方式：->文件访问只能以绝对路径访问方式

1. hadoop fs：开启原生hadoop或hdfs(文件系统)客户端，以与hdfs打交道
2. hadoop fs –ls /: 查看根目录下文件或目录
3. hadoop fs –mkdir /ss: 在根目录下创建文件夹ss
4. hadoop fs –mkdir –p /aa/bb/cc/dd：级联创建多级目录

**核心功能：上传(本地文件系统到hdfs)、下载(hdfs到本地文件系统)**

1. **hadoop fs –put 本地路径 hdfs路径(绝对路径)**
2. **hadoop fs –get hdfs路径 本地路径**
3. hadoop fs –getmerge /ss/aa.txt /ss/bb.txt /home/cc.txt: 合并下载(会将最后一个路径之前的当做需要合并的文件，最后一个为合并生成的文件)
4. hadoop fs –cat /ss/aa.txt: 查看文件内容
5. hadoop fs –rm /ss/aa.txt: 删除文件
6. hadoop fs –rm –r /ss : 递归删除目录及其下内容
7. hadoop fs –f /ss/aa.txt: 强制删除文件
8. hadoop fs –mv /aa/bb.txt /ss/test.txt：改名
9. hadoop fs –cp /text.txt /aa.txt : 复制
10. hadoop fs –appendToFile 本地文件 hdfs文件：在hdfs文件末尾追加，若文件超过128M,才会分块。
11. hadoop fs –tail /aa.txt: 显示文件aa.txt末尾

文件权限3种：

可读：r 4

可写：w 2

可执行：x 1

故最大为7

1. hadoop fs –chmod –R 777 hdfs目录/文件：修改目录及目录下文件权限为7(本用户可读可写可执行)、7(本组成员可读可写可执行)、7(其他成员可读可写可执行)
2. hadoop fs chown –R 用户名：组名 文件/目录🡪修改文件所属用户和组

每个namenode进行数据的管理依靠的Block Pool ID，同一集群中块的Block Pool ID是相同的。

### Hadoop与Eclipse的配置：

1. 配置后，可以在Eclispe中配置HDFS的**可视化界面，**能在Eclipse中显示HDFS目录结构及文件
2. 配置hadoop的通信地址，通过hdfs的主节点与hdfs通信
3. 配置链接Hadoop的用户名，必须关闭Hadoop节点防火墙

**FileSystem(抽象类):** **这个对象是HDFS抽象目录树的一个实例，如果用户想要操作HDFS，首先需要获取到这个实例。**

FileSystem fs=FileSystem.get(conf)://获取的是本地文件系统

FileSystem fs=FileSystem.get(url,conf)://获取分布式文件系统抽象目录树

url: HDFS主节点url

conf: 集群配置文件

**Configuration对象：**加载hadoop集群配置文件对象,默认加载的配置文件来自于jar包中的hdfs-default.xml，实际上配置文件加载有顺序：

1. Jar包下hdfs-default.xml文件(默认备份为3)
2. 工程的classpath(src)下配置文件，只识别两种名字(hdfs-default.xml和hdfs-site.xml文件，其他名字识别不了)
3. 通过代码设置备份数：conf.set(“dfs.replication”,”5”)
4. **优先级递增**

Configuration conf=new Configuration();

读取四个配置文件：

1. core-default.xml
2. hdfs-default.xml
3. maprred-default.xml
4. yarn-default.xml

**文件上传目录：底层是上传到某一个节点上**

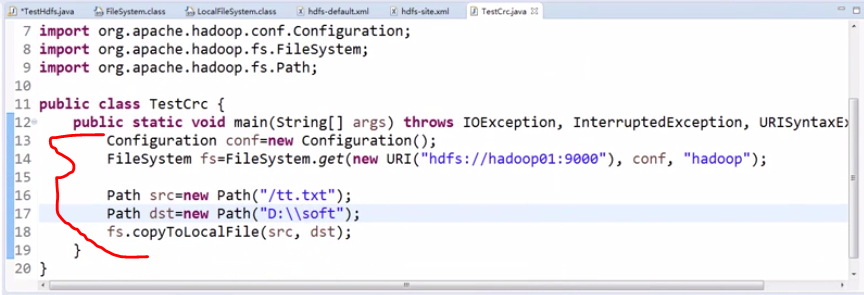
数据真实的存储目录;datanode负责

真实数据的存储目录配置文件的目录：/home/hadoop/data/hadoopdata/data下

current:存储真实数据

in\_use\_lock:锁文件，用于标识datanode进程，一个节点上只允许开启一个datanode进程，这个文件用于锁定datanode进程，只允许一个datanode进程启动

**下载HDFS根目录下文件tt.txt到本地D:\\soft目录：从HDFS下载文件会进行校验(crc文件校验下载数据额完整性，与上传文件时产生的.meta文件进行比较)，根据文件起始偏移量和结尾偏移量进行校验，所以在文件末尾追加内容(中间内容不发生改变)时校验通过，但如果在文件中间插入内容，则校验不通过，报错CheckSumException**



**问题：**从本地上传文件到HDFS遇见问题：没有权限

解决：两种配置方式

1. 代码提交的时候配置：右键->run as->run configuration->arguments(配置程序运行参数)
2. 写在代码中：FileSystem fs=FileSystem.get(url,conf,”hadoop”)://Hadoop指定的代码提交用户
3. 设置用户名后再上传文件：System.setProperty(“HADOOP\_USER\_NAME”,”hadoop”)

### HDFS常用API

**最主要的两个：**

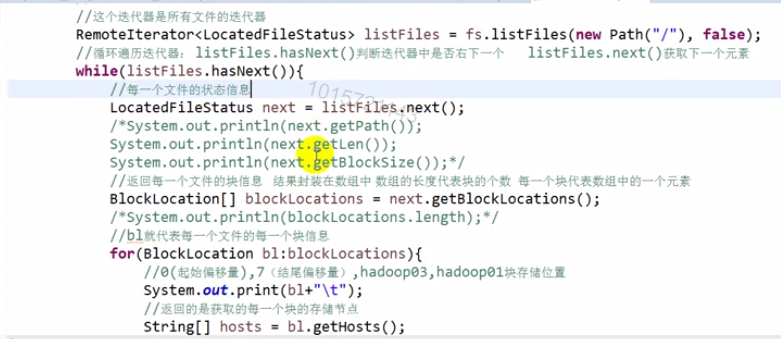
**fs.copyFromLocalFile()://从本地上传到hdfs**

**fs.copyToLocalFile();//从hdfs下载到本地**

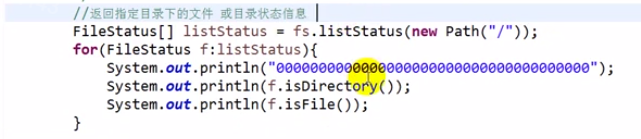
1. 创建文件夹->可创建多级文件夹
   1. Path p=new Path(“”test01“”);
   2. Fs.mkdirs(p);
2. 删除文件夹->可删除多级文件夹(递归删除)
   1. fs.delete(new Path(“/test01”));

fs.delete(new Path(“/ss”),true);🡪设定递归删除

1. 判断目录或文件是否存在
   1. boolean ss=fs.exists(new Path(“/ff”));
2. 重命名
   1. fs.rename(new Path(“/aa.txt”),new Path(/aa1803.txt));
3. 获取指定目录下文件列表(不会获取目录信息)



1. 返回指定目录下的文件或目录状态信息



1. IO流操作文件，必须指定一个文件名(目标hdfs下文件名)
   1. 文件上传，本地(输入流)----->hdfs(输出流)



* 1. 文件下载 hdfs（输入流）--🡪本地（输出流）



### HDFS(提供高容错性的分布式的数据存储方案)的四大机制、两大核心

Hadoop启动时，各进程启动顺序：namenode、datanode、secondarynamenode

1. 四大机制
   1. 心跳机制(heart): 集群节点之间必须做时间同步，namenode作为集群老大，负责集群上任务的额分工，如果要进行分工，必须知道各从节点的存活状况，datanode节点每3秒(配置文件中配置)向namenode发送一次心跳报告，报告是否存活，当namenode连续10次未能接受到datanod的心跳报告，则认为该datanode可能死了，并没有断定死了，这时，namenode会主动向datanode发送一次检查(检查时间为5分钟),若第一次检查未能返回信息，则namenode会再进行第二次检查，若第二次仍未能收到检查信息，仍判断datanode宕机(10次心跳+2次检查=630s)
   2. 安全模式：元数据：1.抽象目录树 2.数据和块的映射关系 3.数据块存储的位置信息，存储在内存(1+2+3)和磁盘(1+2)中

集群第一次启动时，首先将磁盘中元数据加载到内存中，如果磁盘元数据过大会造成加载时间过长，所以磁盘中只存储1、2；而数据块存储的位置信息是由集群启动时namenode接收datanode的心跳报告中获取的

集群启动后(不允许外界对集群进行操作，处于安全模式)：

1. namenode将元数据加载到内存
2. datanode启动，namenode接收datanode心跳报告，获取datanode存活状况、获取块的的存储信息(数据块存储的位置信息)
3. 启动secondarynamenode

安全模式下可进行操作：不修改元数据的操作(查看、下载)

* 1. 机架策略：副本存放机制，默认下每个数据块有3个副本，
     1. 第一个副本一般存储在客户端所在的节点上
     2. 第二个副本放在另外一个机架上的任意一个节点上🡪防止第一个断电
     3. 第三个副本存储在和第一个副本相同机架上的不同节点上
  2. 负载均衡：每个节点存储的数据百分比差别不大 ，在文件上传的时候回优先选择客户端所在的节点，集群会**自动进行负载均衡**(集群空闲时默认1M/S)，适用于小规模集群;**手动负载均衡** start-balancer.sh，不会立即执行，会在hadoop集群空闲时执行。

### HDFS文件上传流程

命令：hadoop fs –put word /

1. 客户端向namenode发送数据上传请求，包含数据长度信息，从而可计算出数据的块数
2. Namenode接收到客户端请求后检查文件是否已经存在、文件上传父目录是否存在、检查权限
3. Namenode检查通过，向客户端返回存储的节点信息，原则：
4. 就近原则，优先返回客户端所在节点
5. 返回同机架节点
6. 返回不同机架节点
7. 客户端接收到namenode返回信息，进行逻辑切块，为以后物理切块(0-127,128-)做准备.
8. 准备文件上传
9. 构建文件上传通道pipeline，根据块id以此进行构建，将同一个块的所有存储节点构建成一个数据流通道（客户端-01节点-02节点-04节点）.
10. 上传文件，边上传边切分，上传的过程是以package为单位(512kb)进行，上传郭长城中会先写到缓存中，每接收到一个package就可以往下一个节点传递，同时缓存中数据还会持续向磁盘中写入。
11. 当第一个块上传完成，通道关闭
12. 开始上传第二块，重复6、7、8动作
13. 所有块上传完成后向客户端返回结果

文件上传过程中有一个节点块上传失败，则HDFS会立即进行一次重试，若仍然失败，会将失败节点从pipeline中剔除，并将失败节点报告给namenode

### HDFS文件下载过程

1. 客户端向namenode发送文件下载请求
2. namenode在自己的元数据库中查询，若查询到则返回客户端数据的块及副本存储节点
3. 客户端收到了存储节点信息，会先进行第一个数据块的下载（就近原则）
4. 第一个块下载完成后会生成一个crc文件，和上传时产生的.meta文件进行文件完整性校验，校验通过则下载成功
5. 进行第二块的下载，重复3、4操作
6. 所有块下载成功后向namenode发送数据下载完毕

文件下载过程中产生异常：数据块的某一节点读取不到数据，此时会向namenode汇报，namenode标记该节点，可能为问题节点，接着读取该块所在的其他节点。

### HDFS元数据详解

元数据存储目录下文件有4大类：

1. 历史日志文件(编辑完成的日志文件)🡪客户端对元数据操作日志，仅记录操作信息
2. 正在编辑的日志文件：目前元数据修改的日志文件
3. 镜像文件：真实的元数据信息经过序列化之后的文件
4. 合并点记录文件：seen\_txid,记录下一次需要合并的日志文件

### Checkpoint（合并）

真实的硬盘上存储的完整的元数据：fsimage+正在编辑的日志文件(需要合并)

内存中元数据永远是最新的最完整的元数据。

如果fsimage不和日志文件进行合并，则fsimage会和内存中元数据差别越来越大，故需要定期fsimage和日志文件的合并，由secondarynamenode完成。

触发合并的情况：

1. 每1h进行一次合并
2. 元数据条数达到100万 条

Checkpoint过程：

1. Secondarynamenode向namenode询问是否checkpoint
2. Namenode检查合并情况，返回响应需要checkpoint
3. Secondarynamenode请求Namenode进行chackpoint
4. Namenode将正在编辑的日志文件进行回滚(将正在编辑的日志文件切换为编辑完成的状态)，同时生成一个新的正在编辑的日志文件。
5. Secondarynamenode将fsimage和edits(编辑完成的历史日志文件)拉取到自己的节点上(合并：即根据edits文件操作fsimage的元数据信息-🡪内存中合并)
6. Secondarynamenode将合并完成的fsimage文件发给namenode（secondarynamenode中会保留合并完成的fsiamge文件）

在没有达到chackpoint过程的时间段内，若集群正常关闭，则集群关闭之前内存中元数据会持久化到硬盘中，保证磁盘中元数据和内存中一致。

Namenode作用：

1. 保存元数据
2. 处理客户端的读写请求
3. 负责分配数据块的存储节点
4. 负载均衡

Secondarynamenode作用：

1. 帮助namenode做元数据备份
2. 进行checkpoint，帮助namenode进行数据合并

Datanode作用：

1. 存储数据块
2. 向namenode发送心跳报告(状态、块位置信息)

Yarn: 集群的资源调度框架，负责集群的资源管理-🡪MapReduce计算需要yarn包

架构：主从架构

主：resourcemanager 负责统筹资源

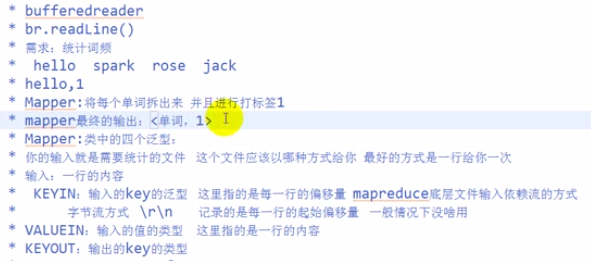
从：nodemanager

Mapreduce: 分布式计算框架实现并行计算，包括Map(映射)、reduce(化简)

# MapReduce并行计算

两个类：mapper类(分布计算)、reducer类(计算结果合并)，只需要重写mapper类的map方法和reducer类的reduce方法。

##### map()方法解析如下：



Mapreduce的example例子wordcount，mapper输入为一行数据，map过程记录每行单词出现记为1，等到reduce过程再将所有次数相加起来。

**序列化反序列化**：当数据需要持久化到磁盘或者网络传输时必须进行序列化和反序列化。

序列化：原始数据--->二进制

反序列化：二进制---->原始数据

Int----->IntWritable

long---->LongWritable

double--->DoubleWritable

String----->Text

Null------>NullWritable

Java中数据类型转换为hadoop中类型：new hadoop类型(java中值)

Hadoop类型转Java中数据类型：hadoop值.get()

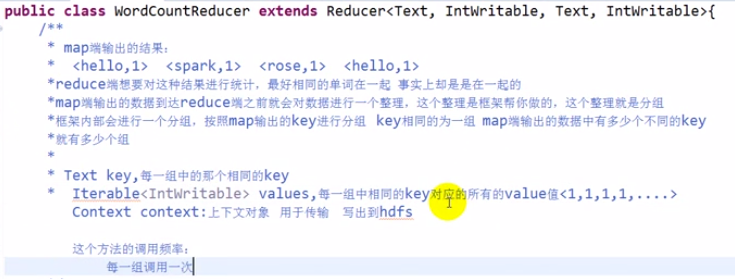
Hadoop提供了序列化和反序列化接口：Writable-->轻便

且已经实现好了8中数据类型的序列化和反序列化接口。

##### reduce()方法解析如下：

**Map()中的Context context:上下文对象，用于写给reduce()数据，且是序列化的数据类型。**

**Reducer接口的reduce()函数的输入时map()端的输出，且必须为序列化的数据类型。**



reduce()方法的调用频率为每一组调用一次，从Mapper的map()方法中调用context()传来的是key-value对组，Mapreduce框架会自动整理，将相同的key值的键值对列为一组。每组调用一次reduce()方法。reduce()方法中context向hdfs中写入。

## Mapreduce编程

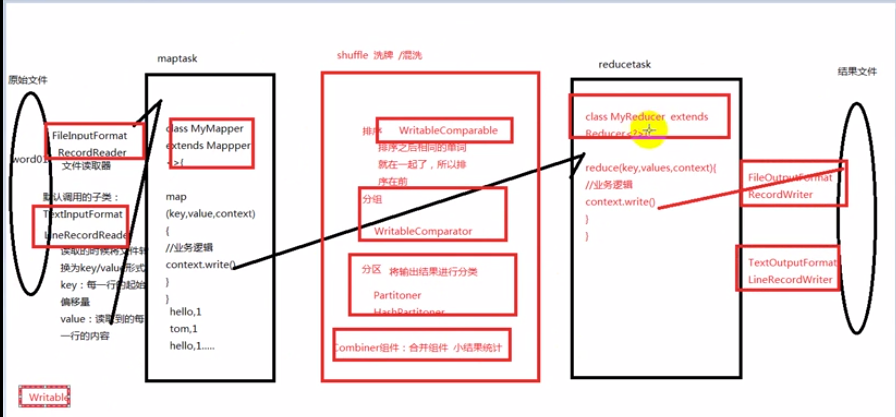
驱动类：代码提交类，将实现的Mapper类和Reducer类封装，设置对应的Mapper-Reducer

**Mapreduce的一个计算程序对应一个Job对象**，设置Job对象的Mapper和Reducer类，将Mapper和Reducer类联系起来；并设置Job对象的mapper的输出类型、redcuer的输出类型；设置输入路径和输出路径。

Mapreduce运行的三种方式：

1. 将代码打成jar包，提交到集群中运行(真实生产中使用)--->不便于代码的调试和修改

**Mapreduce例子wordcount执行流程：**



### Mapreduce的并行度🡪默认一个数据块对应一个切片，对应一个maptask任务

提交代码时：

1. MRAppmaster: 整个运行程序的管理者，，管理整个程序的运行进度
2. Yarnchild: maptask和reducetask运行的进程，一个yarnchild进程对应一个maptask任务或reducetask任务

**Maptask任务并行度：**-->即分布式运行了多少个map任务

Hdfs数据存储是按块进行划分的，故任务划分应和块有直接关系。

在抽象类FileInputFormat中有个方法**getSplits()**，返回包含所有文件切片的List集合，决定每个maptask的任务划分。

**任务划分其实就是对原始要处理的数据进行切分，让不同的数据切片跑在不同的计算节点上，一个逻辑数据切片对应一个maptask，一个maptask对应一个yarnchild**

**底层存储块---128M,不同数据块可能存储在不同的数据节点上，默认情况下切片大小就是数据块的大小，一个maptask任务最终处理的数据就是一个切片的数据，**

**切片和块的关系：**

**切片是数据计算逻辑的划分，块是数据存储的物理划分，默认情况下切片大小和块大小一致。**

**300M\*3的文件最终启动9个maptask任务：**

**每个300M的文件存3个块，3个300M文件分9个块--->对应9个切片—>对应9个maptask任务。**

修改并行度—>需要修改切片大小：

1. 需要修改配置文件
2. 修改代码

**Reducetask任务：reduce端进行任务分配时的每个小任务**

1. Reducetask的并行度和分区个数有关，分区可以对关键字key进行Hash，将相同key的key-value 哈希到同一reducetask上，便于计算最终结果。
2. 设置reducetask并行度：job.setNumReduceTasks(num);,最终结果输出是num个结果文件，再将num个结果文件合并。

**自定义分区**：--->定义类MyPartition继承Partitioner<Text,IntWritable>,实现getPartition(Text key,IntWritable value,int numPartitions)方法，指定分区规则。其中key：map输出key；value:map输出值value；numPartitions：reducetask任务数

1. 必须在代码中指定设置setNumReduceTasks(num)--->决定reducetask任务数
2. 必须在代码中指定设置setPartitionerClass(自定义分区类)，**规划每个reducetask需要处理的数据范围(哈希方法)**
3. 自定义分区getPartitions()中返回值和reducetask的id相对应，返回为2时，对应reducetask02，生成结果文件为part-r-00002，setNumReduceTasks(num)中num值必须至少比getPartitions()返回的最大值大1：
4. 当num=2，而getPartitions()中分区数为2时，可正常进行，每个reducetask负责执行一个分区
5. 当num=2，而getPartitions()中分区大于2时，报错，两个reducetask任务不知道执行哪两个分区
6. 当num=2, 而getPartitions()中分区数小于num时，前面对应的reducetask执行对应的分区任务，剩余的reducetask不执行，但仍产生空的结果文件。

### 数据倾斜：

说明：每个reducetask处理的数据不均匀

后果：影响代码整体执行效率

避免：合理进行分区

### Combiner组件 map->combiner->reduce

**作用：**减少reduce段端数据量，在map端进行了一次合并，减少shuffle过程的数据量，提高分布式计算系统性能，实现逻辑和reduce端相似，不参与业务逻辑，仅相当于map到reduce中间的一个优化组件。

自定义conbiner组件：

1. 继承reducer类
2. 重写reduce()方法

由于Combiner类和Reducer类代码一样，故通常情况下直接用Reducer类。

注意：**Combiner组件针对的是单个maptask（切片），不能对多个maptask的结果进行合并**

**适用情况：求和、最大值、最小值**

**不适应情况：平均值**

### 排序和分组 map与reduce之间

Maptask到redcuetask之间框架默认按照map端输出key的字典顺序排序

需求：将单词统计中词频升序排序

解决：由于默认的是对key进行排序，所以需要将词频放在key的位置。(在Map端发送key-value前将key、value换位置，在reduce端将接收到的key-value位置换回来)，

在自定义类中排序，自定义类实现WriteableComparable接口，并重写 **compareTo()**方法进行比较。

### 全局计数器

作用：统计job运行过程中的进度和状态，类似于job运行的一个报告/日志

### 多Job的串联

程序中有多个job，并且多个job之要相互依赖，job3需要用到job1、job2的数据，则用JobControl对象添加Job间的依赖关系。

步骤：

1. 声明JobControl(串联组)对象
2. 将Job对象变换为可控制的Job
3. 添加Job间依赖关系
4. 将Job对象添加到同一组中
5. 启动线程执行Job组



### 分组

Map—>分组--->reduce，reduce接收到的数据是按照map输出的key进行分组的，分组的时候按照key相同的为一组，需要实现writablecomparable接口，重写compareto()方法，comparato()返回为0的时候默认为一组。

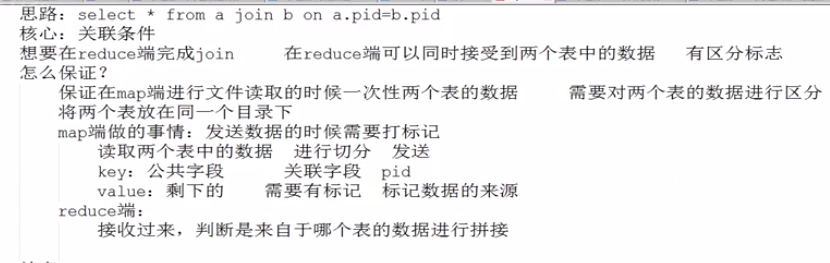
**既有分组又有排序时：先排序、再分组**，且排序时排序字段包含分组字段内容，并且分组字段内容在排序字段的前部分，即先依据分组字段排序，再依据排序字段剩余字段排序，这样使能同一分组的记录紧邻。

Redcue函数中两个坑：

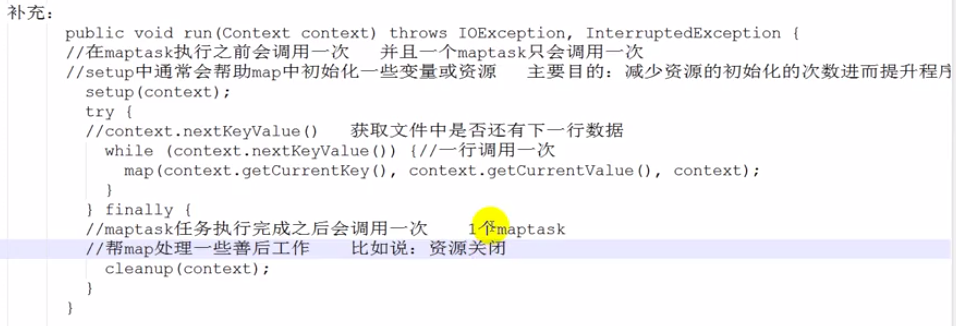
1. Reduce函数中迭代器Iterable<Text> values只能遍历访问一次（指针一定到最后一个）
2. 迭代器中所有对象公用一个地址

### Join

**Redcue端Join：**



**Mapper中实际运行的地方：**



Reduce端Join缺陷：

1. Reducetask并行度问题
2. 容器性能，reducedaunt接收的数据可能会很大
3. Redcuetask容易产生数据倾斜(根据pid分组)

### Map端Join

为了提升map端join过程，需要将**小表数据加载到每个运行maptask的内存**中，如果小表被加载到了内存中，每次在map端只需要读取大表，当读取到大表的每行数据可以直接和内存中的小表关联，就可在map端完成join操作。

大表Join小表：

1. 将小表加载到内存中：job.addCacheFile(url);//将指定url文件加载到内存
2. 在Mapper类的setup()方法中获取缓存文件内容（小表url路径），并利用url路径获取小表文件内容，可以将小表中关联字段放在Map对象的key值
3. 大表文件作为map()的输入文件，查看2中Map对象是否包含关联字段，若包含则组合后写入即可。

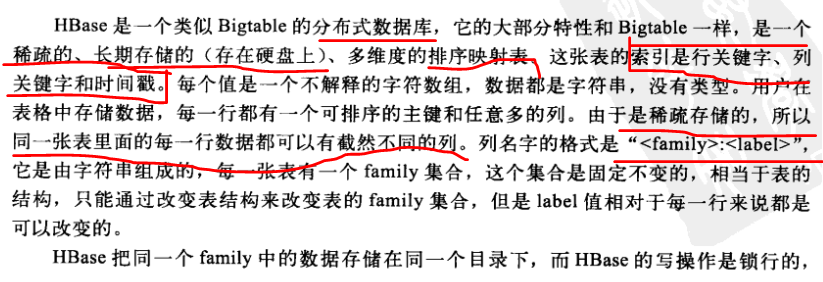
大表Join大表：将一个大表切分成小表，再执行关联

### Map端shuffle过程

**环形缓冲区**：maptask输出结果会先输出到环形缓冲区中(context.write()),环形缓冲区(首尾相连的字节数组)默认大小100M，设有阈值0.8，当达到阈值后环形缓冲区会触发写入磁盘操作,一个maptask任务对应一个环形缓冲区

shuffle过程：

## HBase的数据管理-分布式数据库



### HBase的三个组件管理数据

HDFS介绍：  
 设计思想：文件过大，切分存储

1. 分块存储，每个块叫做block->考虑负载均衡

Hadoop2.x中默认块大小为128M，Hadoop1.x中默认块大小为64M

若一个文件不足默认块的大小，也单独存一个块，块的大小为存储数据的实际大小

1. 备份机制：文件块默认的备份个数是3个，地位相同，没有主次之分

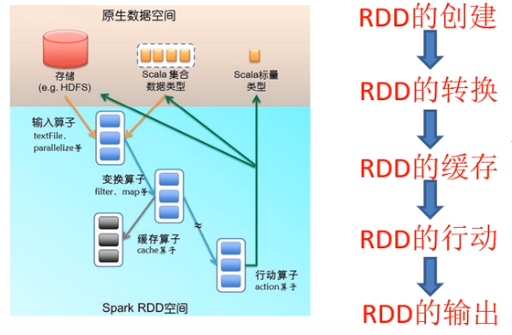
备份的数据块存储在不同的节点上，若节点总共2个，而数据块副本为3个，实际存储2个，另外一个进行记账，当集群节点个数大于等于3时会进行复制副本，最终达到3个。

副本过多带来问题：资源占用过多、维护困难

# Spark基础

B站《尚硅谷大数据Spark》

Hadoop不擅长迭代式计算和交互式数据挖掘，Spark基于内存计算，而Hadoop基于磁盘，Spark多个计算任务可打成一个jar包。



## Spark Core

**RDD：弹性分布式数据集**(分布式内存的抽象),Spark计算基石，本质就是个类，为用户提供了一组方便的数据转换和求值方法，为用户屏蔽了底层复杂的计算和映射环境。

**特性：**

1. 不可变：若需要在一个RDD上进行转换操作，则会生成一个新的RDD.
2. 可分区：RDD里的具体数据分布在多台机器的Executer上，堆内内存和堆外内存+磁盘。
3. 弹性：
   1. 存储的弹性：内存与磁盘的自动切换
   2. 容错的弹性：数据丢失可以自动回复
   3. 计算的弹性：计算出错重试机制--->计算分层，有应用—Job---Stage---TaskSet—Task,每一层都有对应的计算的保障和重复机制，保障计算不会由于一些突发因素而终止。
   4. 分片的弹性：根据需要重新分片(重新调整RDD中数据分布)

Spark Core工作：操作RDD RDD的创建🡪RDD的转换—>RDD的缓存—>RDD的行动🡪RDD的输出。

### RDD的创建方式：

1. 从集合中创建RDD
   1. paralelize():-->将数据并行化分片到节点

val r=sc. parallelize(1 to 10)—>Array[1,2..,10];或者

val r=sc. parallelize(List(“a”,”b”,”c”));

* 1. makeRDD(): -->将数据并行化分片到节点，其内部实现是基于parallelize.

val r=sc. makeRDD(1 to 10);或者

val r=sc.makeRDD(List((1,List(“d”,”e”,”f”)),(2,List(“a”,”b”,”c”))));🡪**指定两个分片**(RDD存放位置)，将(d,e,f)放在分片1中，(a,b,c)放在分片2中。

1. 从外部存储创建RDD
2. 从其他RDD转换

### RDD的操作

1. **转换(transformation)**：通过操作将一个RDD转换成另一个RDD，RDD的所有**转换操作都是懒执行的**，只有当行动操作出现时Spark才会真的去执行（会使Spark更有效率）。
   1. map(func)--->RDD对象中所有元素执行func中操作
   2. mapPartitions(func)--->按照分区迭代执行func方法，执行次数等于分区数
   3. filter(func)--->返回一个新的RDD，该RDD的内容为满足func函数为true的数据集合
   4. glom():将每个分区形成一个数组，形成新的RDD类型时RDD[Array[T]]
   5. flatMap(func)—>返回可迭代类型，可变换类型
   6. mapPartitionsWithIndex()--->传入迭代器
   7. sample()
2. **行动(action)**：将一个RDD进行求值或者输出