# 数据处理

# 大数据处理场景

- 中低规模数据的数据处理场景
- 关系型数据库+存储过程
- 使用ORACLE-SQL开发方式，满足对常规，中低数据规划的数据仓库，数据集市处理。
- 低延时应用查询基础数据关联场景
- Redis-noSQL
- 准实时统计分析场景
- HBase
- 基本可以达到准实时统计分析功能
- 没有开源实现，开发成本高
- 大多是自有系统，例如Google的Percolator，淘宝的prom
- 大数据批处理计算
- Pig
- 语法不很普及
- 数据流语言，适合对问题一步步的描述。
- 统计和机器学习（Yahoo, twitter）

# 大数据处理场景

- 大数据汇总简单计算
- Hive-SQL
- SQL熟悉，处理大数据汇总、简单计算的场景
- 典型案例
- 1）用户访问日志处理/互联网广告
- 2）电子商务（淘宝的云梯）
- 复杂高效大数据处理
- 使用MAP-REDUCE处理应对复杂逻辑，要求高效、大数据场景的开发
- 批处理
- 性能比Hive和Pig要高
- 适合算法要求比较高的场景，对mapreduce运行的逻辑进行重写，定制代码，提高效率；适合其他pig和hive无法实现的任务，开发难度大
- 典型案例
- 1） 搜索引擎网页处理
- 2） 典型的ETL（全盘扫描）
- 3） 机器学习/聚类，分类，推荐（百度Ecomm）

# 大数据处理场景

- 复杂高效大数据处理
- 使用HIVE-SQL的开发方式，满足日常大数据汇总、处理、简单计算的场景开发。
- 使用HBASE-NOSQL方式开发，满足应用类，快速查询，快速提取数据，快速检索等需求场景。
- 使用REDIS-NOSQL方式开发，满足低延时应用查询，基础数据关联等场景。
- 使用H2O，满足高效数据统计、机器学习、大数据计算等场景。

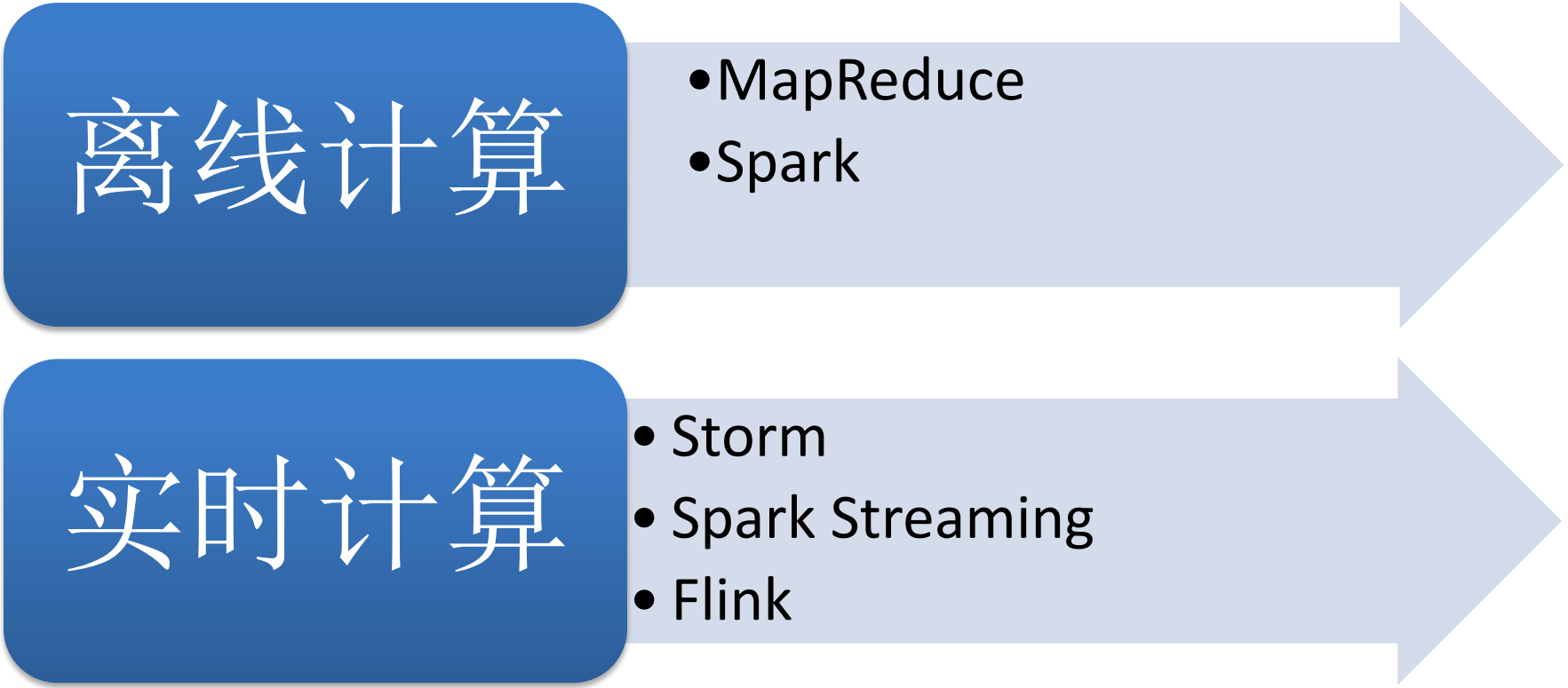
# 数据处理种类

| | |
|---|---|
| 大数据查询分析计算 | HBase,Hive,Cassandra,Impala,Shark,Hana |
| 批处理计算 | Hadoop,MapReduce,Spark |
| 流式计算 | Scribe, Smaza,Flume,Storm,S4,Spark,Streaming |
| 迭代计算 | Hadoop,iMapReduce,Twister,Spark |
| 图计算 | Pregel,Graph,Trinity,PowerGraph,GraphX |
| 内存计算 | Dremel,Hana,Spark |
| 混合计算 | Spark, Flink |

# 数据处理种类

| | |
|---|---|
| 查询分析 | Hive |
| | Impala |
| | Spark SQL |
| | Presto |
| 信息检索 | Solr |
| | Elasticsearch |
| 挖掘分析 | Mahout |
| | Spark MLlib |

# 数据计算分类



# 用户行为分析离线数据处理

- 离线处理平台
- 离线平台基本架构与技术选型
- 采用什么样的系统和平台，从哪些角度考虑?
- 用户行为数据收集
- 利用 Flume 进行分布式数据收集
- 用户行为数据存储
- 利用 HDFS 和 HBase 进行大数据存储
- 用户行为分析
- 利用 MapReduce 和 Hive 进行用户行为分析，为后面数据推荐做准备
- 利用 Spark 进行模型训练，为后面数据推荐做准备

# 用户行为分析实时数据处理

- 实时处理平台
- 实时处理平台基本架构与技术选型
- 采用什么样的系统和平台，从哪些角度考虑？如何与离线平台完成对接。
- 用户行为实时获取
- 利用 kafka 消息队列存储用户行为流式数据
- 用户行为实时分析
- 利用 Storm 实时分析用户行为数据

# 数据处理调度

- 把多个逻辑处理节点的统一调度起来。
- 1、进行节点的类型研究（java程序节点、shell节点、hive节点、mapreduce节点等）
- 2、节点间关系研究（顺序、选择、分支、合并）
- 3、常用的调度方式研究，shell调度方式应用、oozie调度方式应用，其中oozie为基于Hadoop的数据处理调度方式。

# MR

# MapReduce起源：Google搜索

每一次搜索

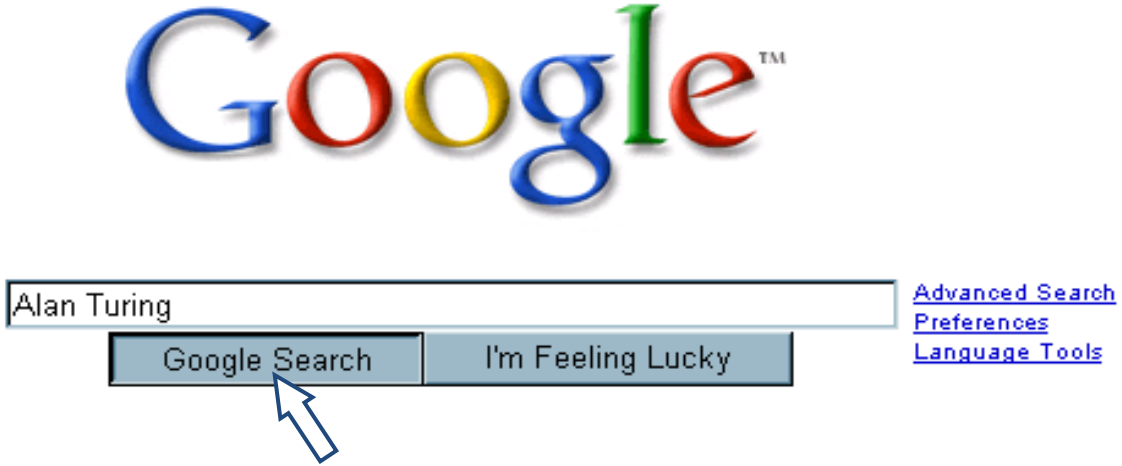
200+ CPU

200TB以上数据

$10^{10}$ CPU周期

0.1秒内响应

5¢广告收入



# 简单的问题，计算并不简单！

- 计算问题简单，但求解困难
- 待处理数据量巨大（PB级），只有分布在成百上千个节点上并行计算才能在可接受的时间内完成
- 如何进行并行分布式计算？
- 如何分发待处理数据？
- 如何处理分布式计算中的错误？

# MapReduce大规模数据处理

- 处理海量数据（>1TB）
- 上百/上千 CPU 实现并行处理
- 简单地实现以上目的
- 对大数据并行处理：分而治之
- 上升到抽象模型：Map与Reduce
- 上升到框架：以统一框架为程序员隐藏系统细节

Google MapReduce
架构设计师
Jeffrey Dean

分而治之

**Divide and Conquer**

# MapReduce起源

- 源自于Google的MapReduce论文
  - 发表于2004年12月
  - Hadoop MapReduce是Google MapReduce的完整版
- MapReduce的特点
  - 分布式计算框架
  - 易于编程：简单的实现一些接口，类似于写一个简单的串行程序，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的PC 机器运行
  - 良好的拓展性：计算资源不够的时候，可以通过简单的增加机器扩展计算能力
  - 高容错性：例如其中一台机器挂了，会自动把上面的计算任务转移到另外一个节点上运行，不至于任务运行失败，整个过程不需要人工参与
  - 适合PB级以上海量数据的离线处理

# 常见MapReduce应用场景

- 简单的数据统计，比如网站pv，uv统计
- 搜索引擎建索引
- 海量数据查找
- 复杂数据分析算法实现
  - √聚类算法
  - √分类算法
  - √推荐算法
  - √图算

# MR处理模式

- 流处理（stream processing）和批处理（batch processing）两种。批处理是先存储后处理（store-process），而流处理则是直接处理（straight-through process）。（有时也分为在线、离线、近线三种）

batch processing

Fact finding with data at rest

stream processing

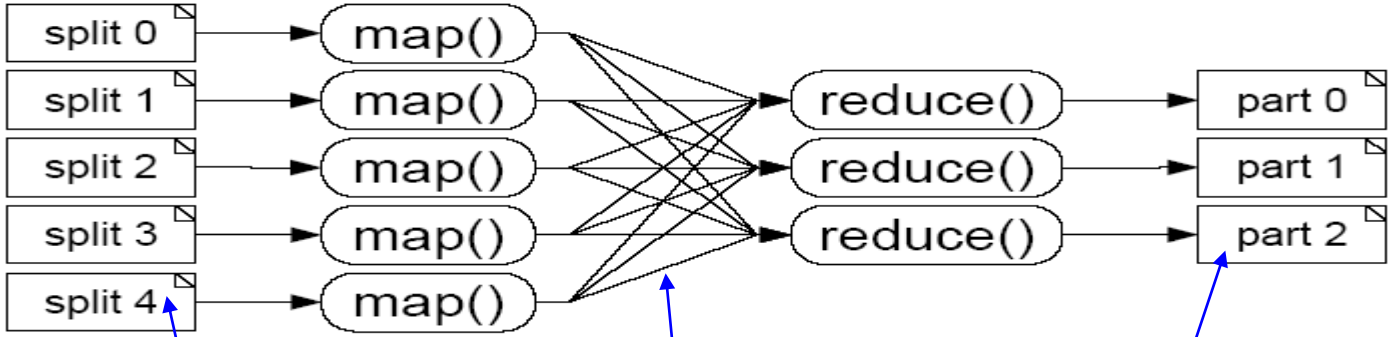
Insights from data in motion

# MapReduce不擅长

- 实时计算：MapReduce无法像Mysql一样，在毫秒或者秒级内返回结果
- 流式计算：MapReduce自身的设计特点决定了数据源必须是静态的，不能动态变化，而流式计算的输入数据是动态的
- DAG计算：涉及多次迭代计算，MapReduce可以做，但是每个MapReduce作业的输出结果都会写入到磁盘，造成大量的磁盘IO，导致性能低下

# 从打扑克与说起

分牌 *input* | 各自齐牌 *map tasks* | 再次理牌 *reduce tasks* | 搞定 *output*




交换

Input split

shuffle

output

# MapReduce编程范例

场景：有大量文件，里面存储了单词，而且一个单词占一行
任务：如何统计每个单词出现的次数
1:整个文件可以加载到内存中；
　√sort datafile | uniq –c
2:文件太大不能加载到内存中，但
　√ <word.count>可以放到内存中：
3:文件太大无法加载到内存中，且
　√ <word.count>也不行
类似应用场景：
搜索引擎中，统计最流行的K个搜索词；
统计搜索词频率，帮助优化搜索词提示

# MapReduce基本流程

- MapReduce是一种编程模型，用于大规模数据集的并行计算，主要思想是Map(映射)和Reduce(化简)。将作业的整个运行过程分成2个阶段：Map阶段和Reduce阶段；
- MapReduce程序将输入数据列表变成输出数据列表。
- MapReduce程序由两个阶段组成：Map和Reduce，用户只需要实现map()和reduce()两个函数，即可实现分布式计算。

# MapReduce编程模型

- map和reduce函数遵循如下常规格式：

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

- map 函数将每个单词转化为key/value对输出，这里key为每个单词，value为词频1。(k2,v2)是 map 输出的 key/value 中间结果对。

- reduce 将相同单词的所有词频进行合并，比如将单词k2，词频为list(v2)，合并为(k3,v3)。reduce 合并完之后，最终输出一系列(k3,v3)键值对

# MapReduce基本流程

**Map阶段由一批同时运行的Map Task 组成，每个 Map Task 由3个部分组成：**

- InputFormat：对输入数据格式进行解析
- Mapper：输入数据处理
- Partitioner：数据分组， Mapper 输出key会经过Partitioner分组选择不同的Reduce。默认Partitioner 会对

map 输出的key进行hash取模，比如有6个Reduce Task， 它就是模（mod）6，如果key的hash值为0，就选择第0个Reduce Task。

# MapReduce基本流程

**map任务处理**

- 对输入文件的每一行，解析成key、value对，每一个键值对调用一次map函数
- 写自己的逻辑，对输入的key、value处理，转换成新的key、value输出
- 对输出的key、value进行分区
- 对不同分区的数据，按照key进行排序、分组，相同key的value放到一个集合中
- (可选)分组后的数据进行归约

# MapReduce基本流程

**Reduce 阶段由一批同时运行的 Reduce Task 组成，每个Reduce Task由4个部分组成：**

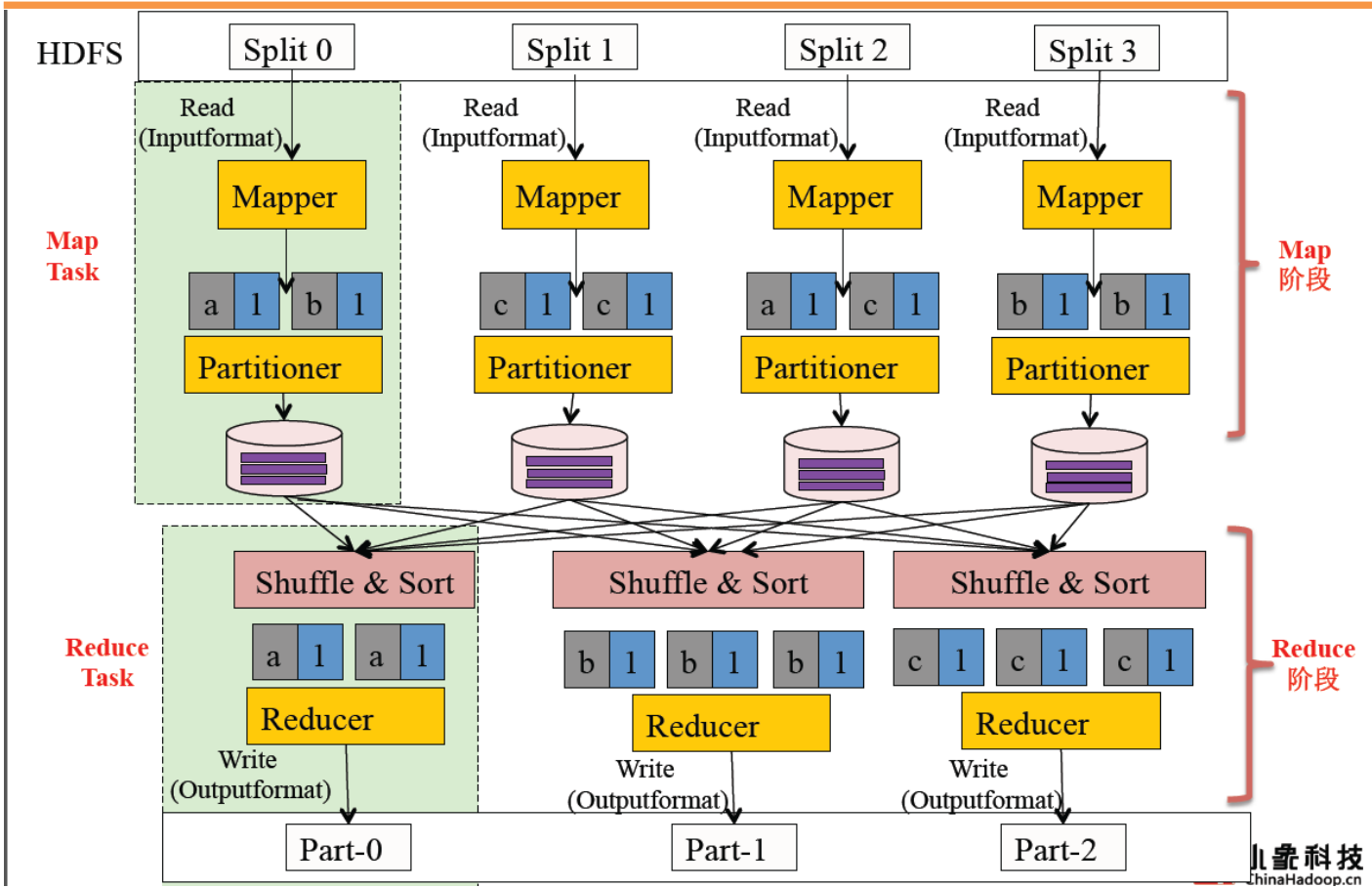
- Shuffle： Reduce Task 远程拷贝每个 map 处理的结果， 从每个 map 中读取一部分结果，Partitioner 决定每个 Reduce Task 拷贝哪些数据
- Sort：读取完数据后，会按照key排序，相同的key被分到一组
- Reducer：数据处理，以WordCount为例，对相同的key统计词频数
- OutputFormat：数据输出格式， 默认为 TextOutputFormat ，以WordCount为例，这里的key为单词，value为词频数

# MapReduce基本流程

**reduce任务处理**

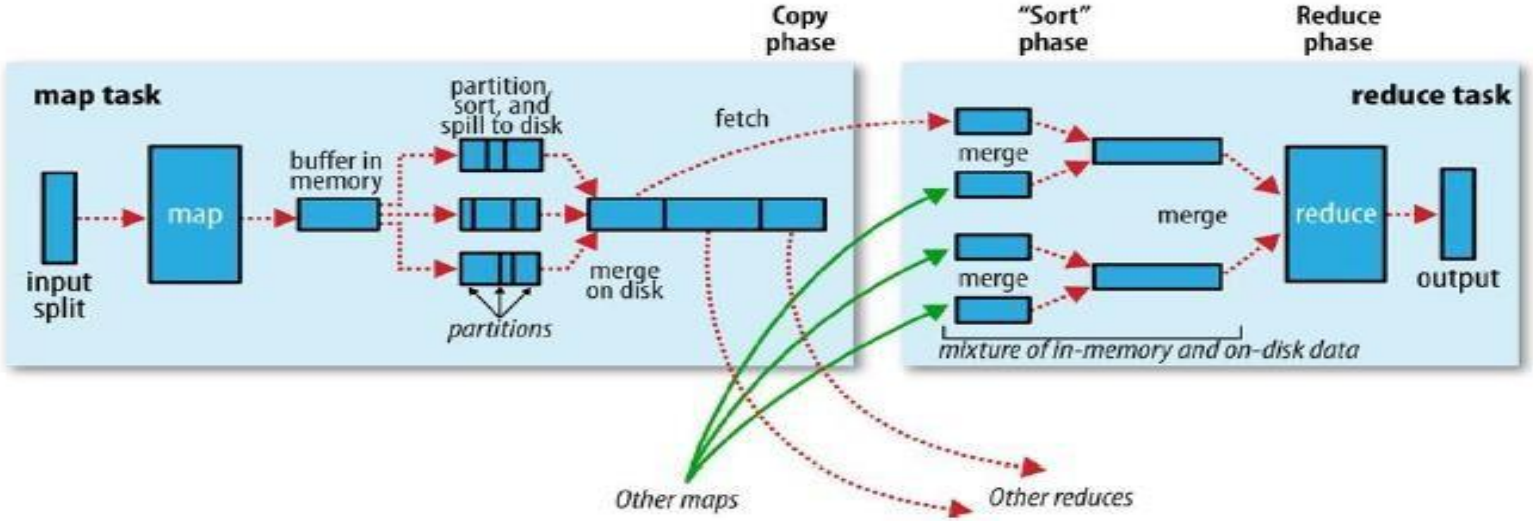
- 对多个map任务的输出，按照不同的分区，通过网络拷贝到不同的reduce节点
- 对多个map任务的输出进行合并、排序
- 写reduce函数逻辑，对输入的key、value处理，转换成新的key、value输出
- 把reduce的输出保存到文件中

# MapReduce的编程模型-内部逻辑




- Split：HDFS 中的数据以 Split 方式作为 MapReduce 的输入
- Block 是 HDFS 术语，Split 是 MapReduce 术语
- 通常1个 Split 对应1个 block，也可能对应多个block，具体是由 InputFormat 和压缩格式决定的
- 默认情况下，使用的是 TextInputFormat，这时1个Split对应1个block
- Mapper解析出的数据输出到本地磁盘上

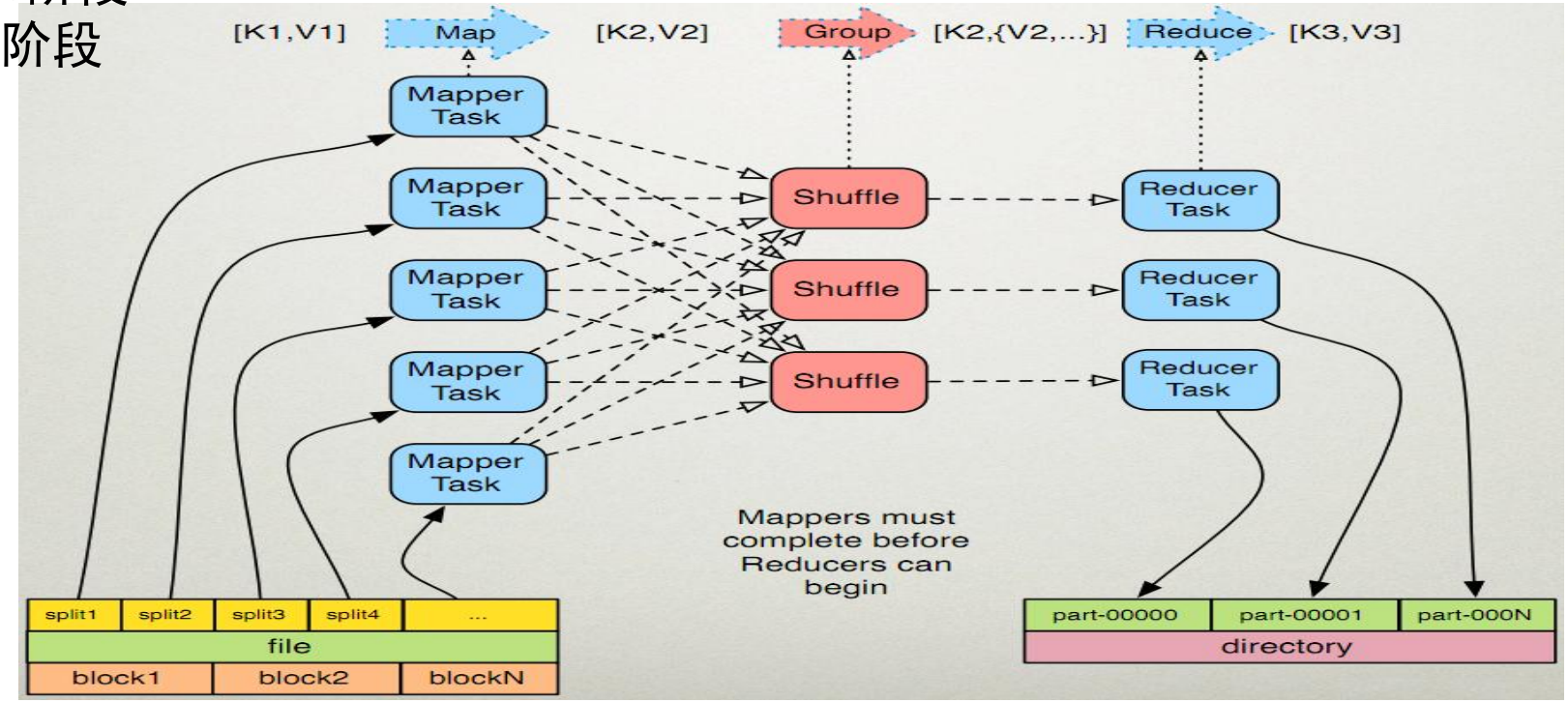
# MapReduce的编程模型-内部逻辑




- 缓冲区默认为100M，由io.sort.mb属性控制
- 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），写磁盘，最后合并，reduce端也一样，reduce端拿到的map端数据是按key排序

# MapReduce四大阶段

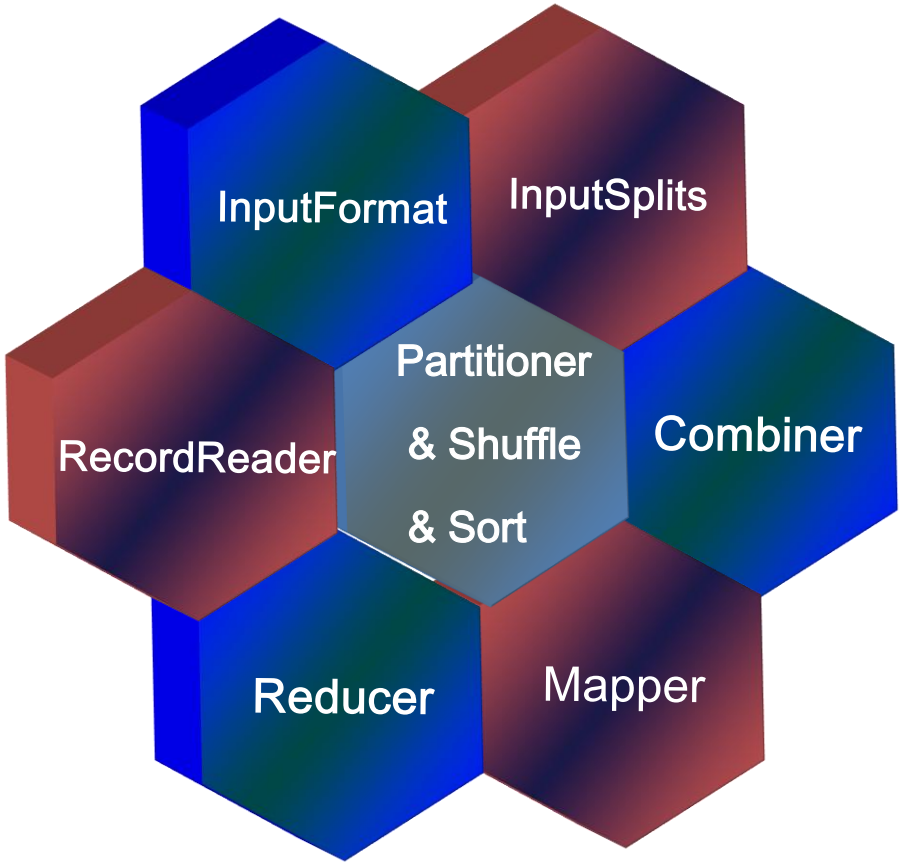
- 输入分片（input split）阶段
- Map阶段
- shuffle阶段
- Reduce阶段



# MapReduce编程模型

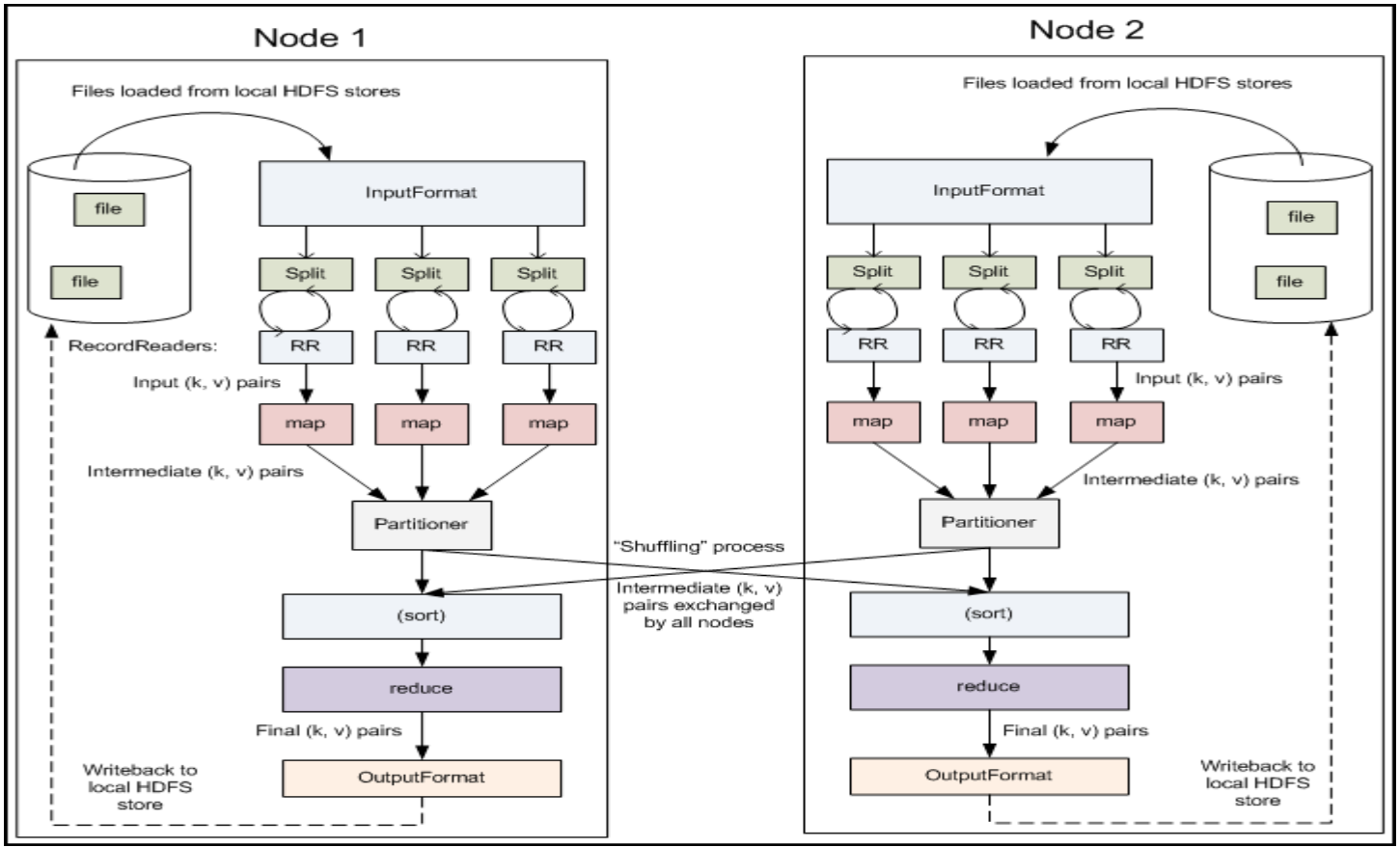
- Map阶段
  - InputFormat(默认TextInputFormat)
  - Mapper
  - Combiner(local reduvcer)
  - Partitioner
- Reduce阶段
  - Reducer
  - OutputFormat(默认TextOutputFormat)

# MR核心组件



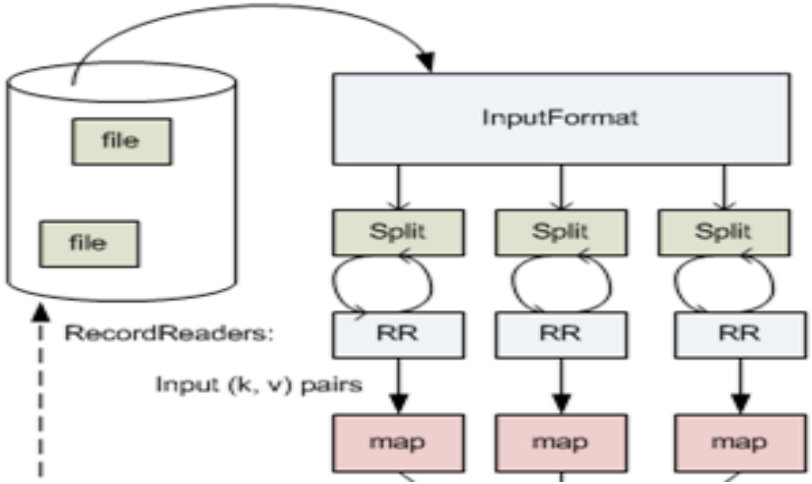

- 组件就是对象，是对数据和方法的简单封装。组件有自己的属性和方法。属性是组件数据的简单访问者。方法则是组件的一些简单而可见的功能。
- 类是实现了源代码级的重用，是静态重用组件是要实现二进制的重用，动态重用，并最终实现搭积木式的系统构造的梦想。
- 在组件发展中，也发展了很多的强大特性，如：分布式组件，分布式组件极大的提高了系统的灵活性，伸缩性（负载伸缩）和可维护性。

# MR核心组件


Node 1
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store
Node 2
Files loaded from local HDFS stores
InputFormat
file
file
Split
Split
Split
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

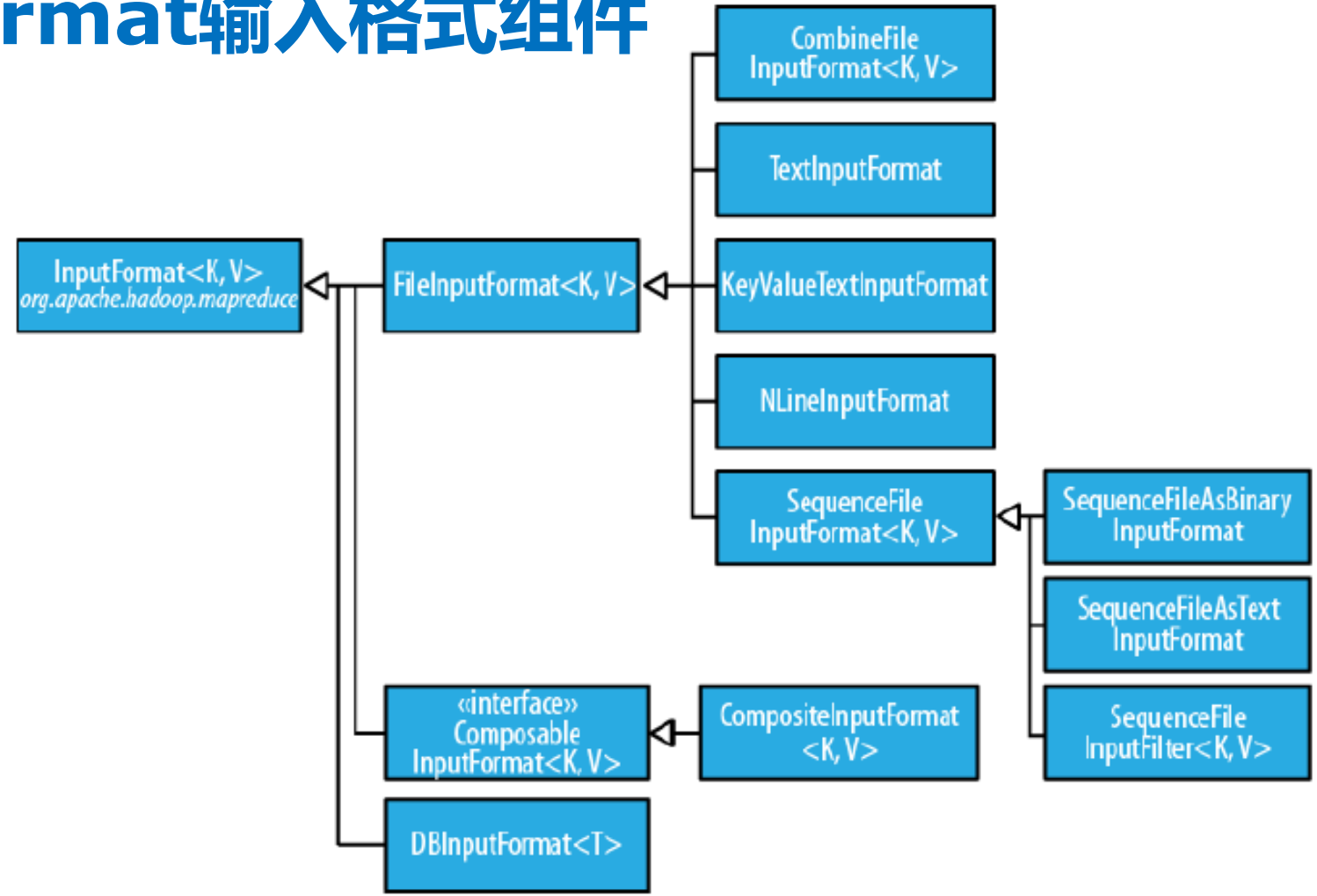
# InputFormat输入格式组件

- 文件分片（Input Split）方法：
  - 处理跨行问题
- 将分批数据解析成key/value对
  - 默认实现是TextInputformat
- TextInputformat
  - Key是行在文件中的偏移量；
  - 若行被阶段，则读取下一个block的前几个字符；




定义了数据文件如何分割和读取

# InputFormat输入格式组件



# InputFormat文件输入格式组件

| InputFormat: | Description: | Key: | Value: |
| --- | --- | --- | --- |
| TextInputFormat | Default format; reads lines of text files | The byte offset of the line | The line contents |
| KeyValueText Input Format | Parses lines into key-val pairs | Everything up to the first tab character | The remainder of the line |
| SequenceFileInput Format | A Hadoop-specific high-performance binary format | user-defined | user-defined |

# MR输入格式——TextInputFormat

- 默认的 InputFormat
- 键是存储该行在Block中的字节偏移量
- 值是这行的内容，不包括任何行终止符

```
Rich learning form
Intelligent learning engine
Learning more convenient
From the real demand for more close to the enterprise
```

```
(0,Rich learning form)
(19,Intelligent learning engine)
(47,Learning more convenient)
(72,From the real demand for more close to the enterprise)
```

# MR输入格式——FileInputFormat

- 所有使用文件作为数据源的 InputFormat 的基类
- 设置作业的输入文件位置
- FileInputFormat 提供了四种静态方法来设定 Job 的输入路径.

```
public static void addInputPath(Job job, Path path)
public static void addInputPaths(Job job, String commaSeparatedPaths)
public static void setInputPaths(Job job, Path... inputPaths)
public static void setInputPaths(Job job, String commaSeparatedPaths)
```

- setInputPathFilter()方法设置一个过滤器，排除特定文件

```
public static void setInputPathFilter(Job job, Class<? extends PathFilter> filter)
```

# MR输入格式——KeyValueTextInputFormat

- 每一行被分隔符（缺省是\t）分割为key和value
- key和value都是Text类型

```
line1 —>Rich learning form
line2 —>Intelligent learning engine
line3 —>Learning more convenient
line4 —>From the real demand for more close to the enterprise
```

```
(line1,Rich learning form)
(line2,Intelligent learning engine)
(line3,Learning more convenient)
(line4,From the real demand for more close to the enterprise)
```

# MR输入格式——NLineInputFormat

- 每个Mapper 收到固定行数的输入
- 键是存储该行在Block中的字节偏移量
- 值是这行的内容，不包括任何行终止符
- N值由mapreduce.input.lineinputformat.linespermap 属性设定。

```
Rich learning form
Intelligent learning engine
Learning more convenient
From the real demand for more close to the enterprise
```

```
(0,Rich learning form)
(19,Intelligent learning engine)
```

```
(47,Learning more convenient)
(72,From the real demand for more close to the enterprise)
```

# MR输入格式——SequenceFileInputFormat

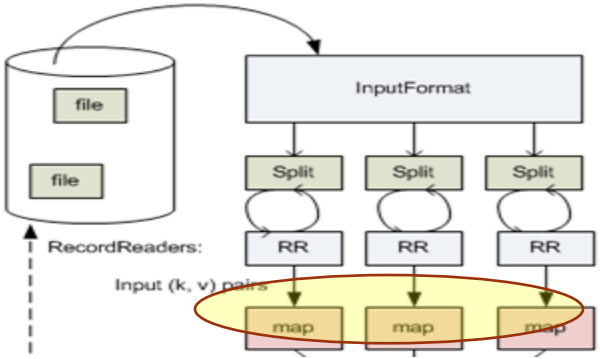
- 用于读取 sequence file
- sequence file为 Hadoop 专用的压缩二进制文件格式
- 键和值由用户定义

# MR多个输入

- 一般MapReduce 作业的多个输入文件，都由同一个InputFormat 和 同一个 Mapper 来解释
- 数据格式往往会随时间演变，或者有些数据源会提供相同的数据， 但格式不同
- MultipleInputs允许为每条输入路径指定 InputFormat 和 Mapper
- 下面代码取代 FileInputFormat.addInputPath()和 job.setMapperClass() 的常规调用

```
MultipleInputs.addInputPath(job,ncdcInputPath,TextInputFormat.class,NCDCTemperatureMapper.class);
MultipleInputs.addInputPath(job,metofficeInputPath,TextInputFormat.class,MetofficeTemperatureMapper.class);
```

# InputSplits输入数据分块组件




定义了输入到单个Map任务的输入数据

- 一个MapReduce程序被统称为一个Job，可能有上百个任务构成
- InputSplit将文件分为64MB的大小
- 配置文件hadoop-site.xml的mapred.min.split.size参数控制这个大小
- mapred.tasktracker.map.taks.maximum用来控制某一个节点上所有map任务的最大数目
- InputSplit定义了一个数据分块，但是没有定义如何读取数据记录

# InputSplits输入数据分块组件

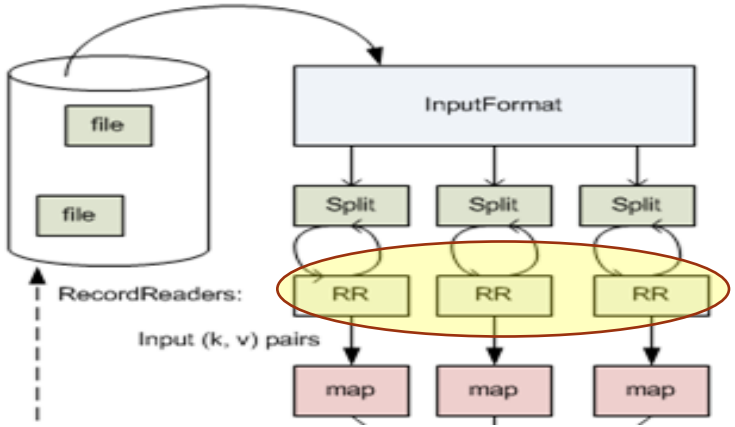
- 在进行Map计算之前计算输入分片
- 输入分片存储内容为分片长度和数据位置的数组，而非数据本身
- 分片数量由文件大小和HDFS块数量决定,分片数量决定Map的个数
- 例，块大小为64M，文件大小分别为3M、65M和127M，则分片分别为：3M为3/64=1分片,65M为65/64=2分片，
  127mb为127/64=2分片

# Split与Block

- Block：
  - HDFS是最小的数据存储单位；
  - 默认是128MB；
- Split
  - Mapreduce中最小的计算单位；
  - 默认与block一一对应；
- Block与Split
  - Block与split的对应关系是任意的，可以用户控制；

# RecordReader数据记录读入组件

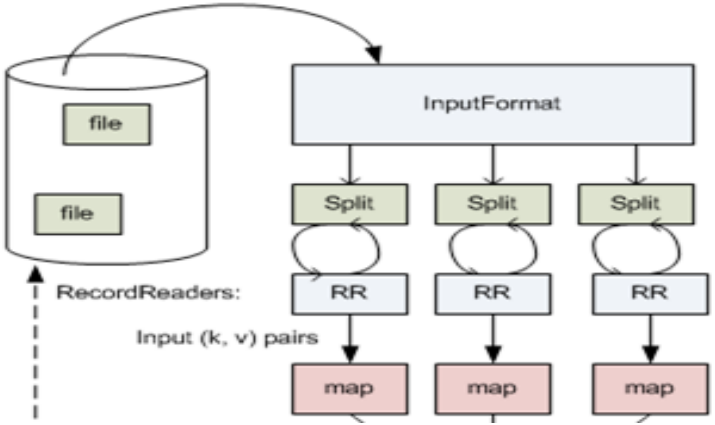
- 定义了如何将数据记录转化为一个(key,value)对的详细方法，并将数据记录传给 Mapper类
- TextInputFormat提供LineRecordReader，读入一个文本行数据记录



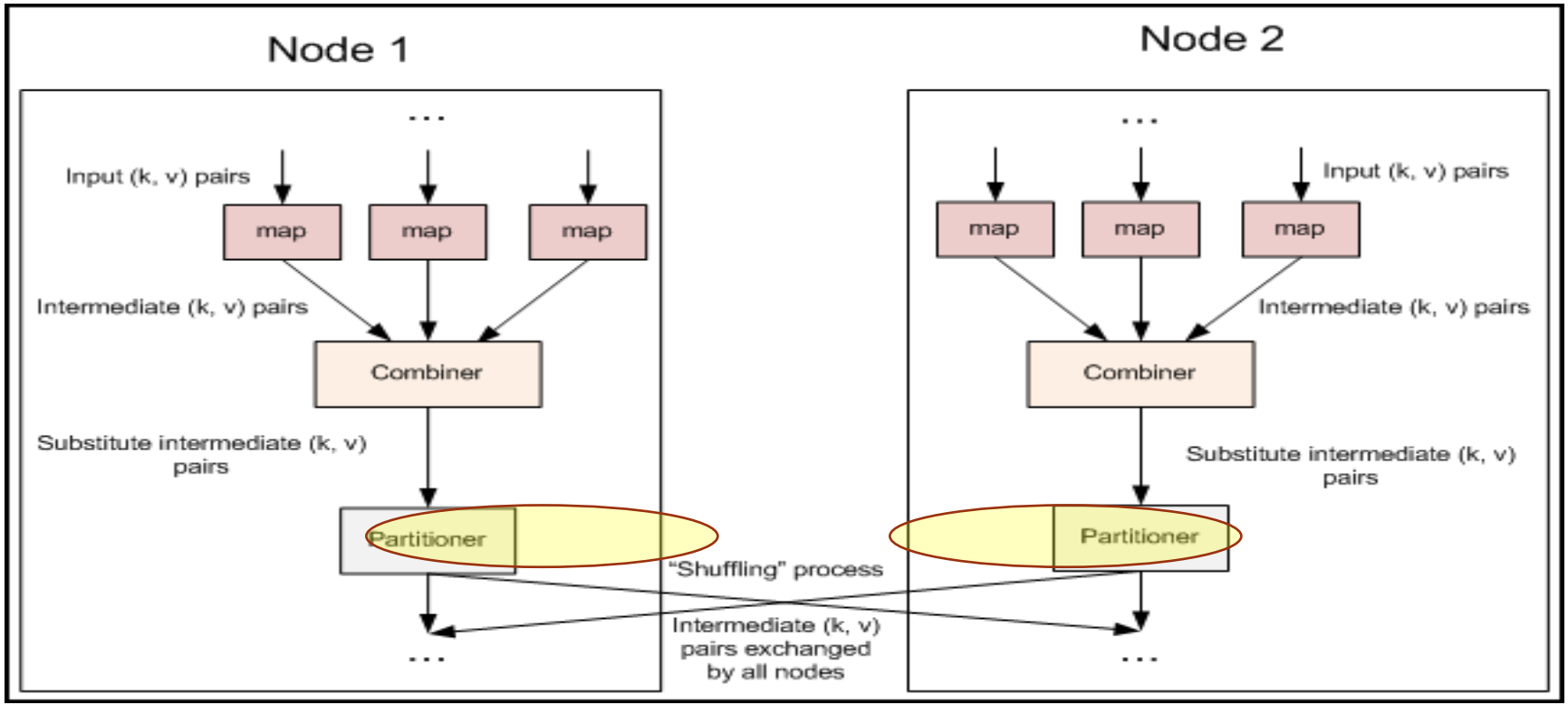
# Mapper组件

每一个Mapper类的实例生成了一个Java进程，负责处理某一个InputSplit上的数据

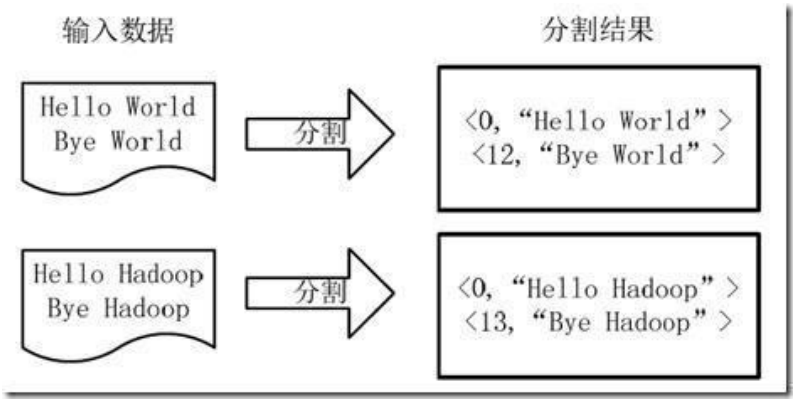
- OutputCollector收集中间结果，Reporter用来获得环境参数以及设置当前执行的状态
- Mapper负责“分”，即把复杂的任务分解为若干个“简单的任务”执行“简单的任务”有几个含义：
- 数据或计算规模相对于原任务要大大缩小；
- 就近计算，即会被分配到存放了所需数据的节点进行计算；
- 这些小任务可以并行计算，彼此间几乎没有依赖关系

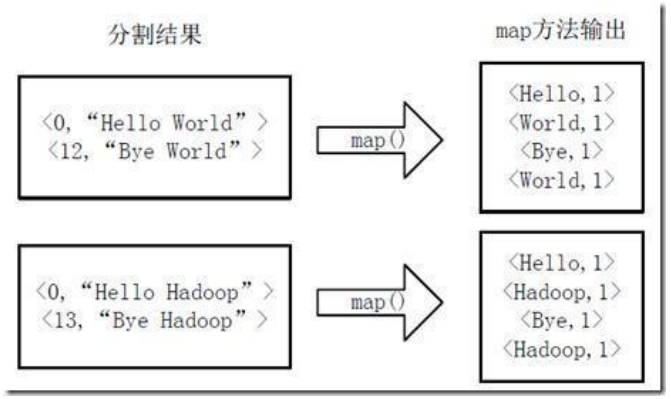


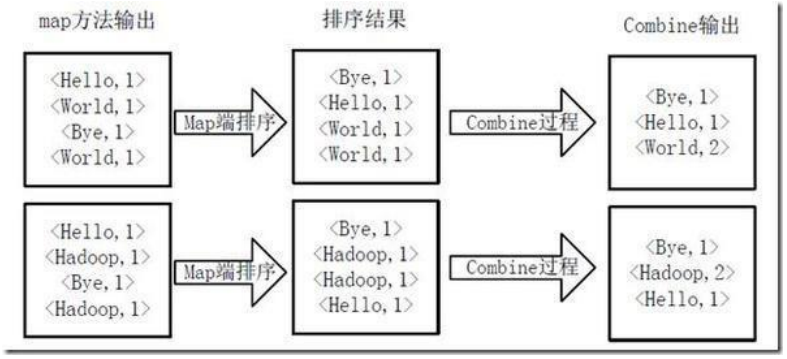
# Combiner组件

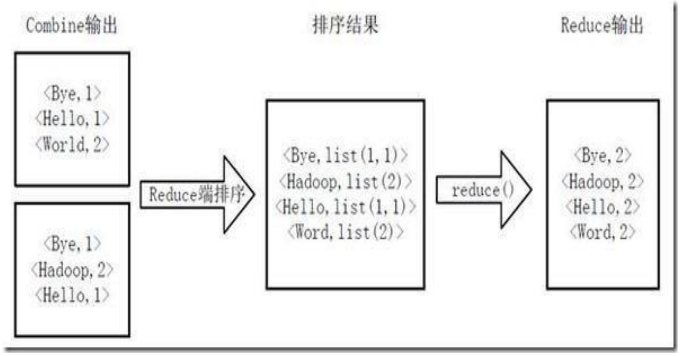

Node 1
Node 2
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes

# Combiner组件

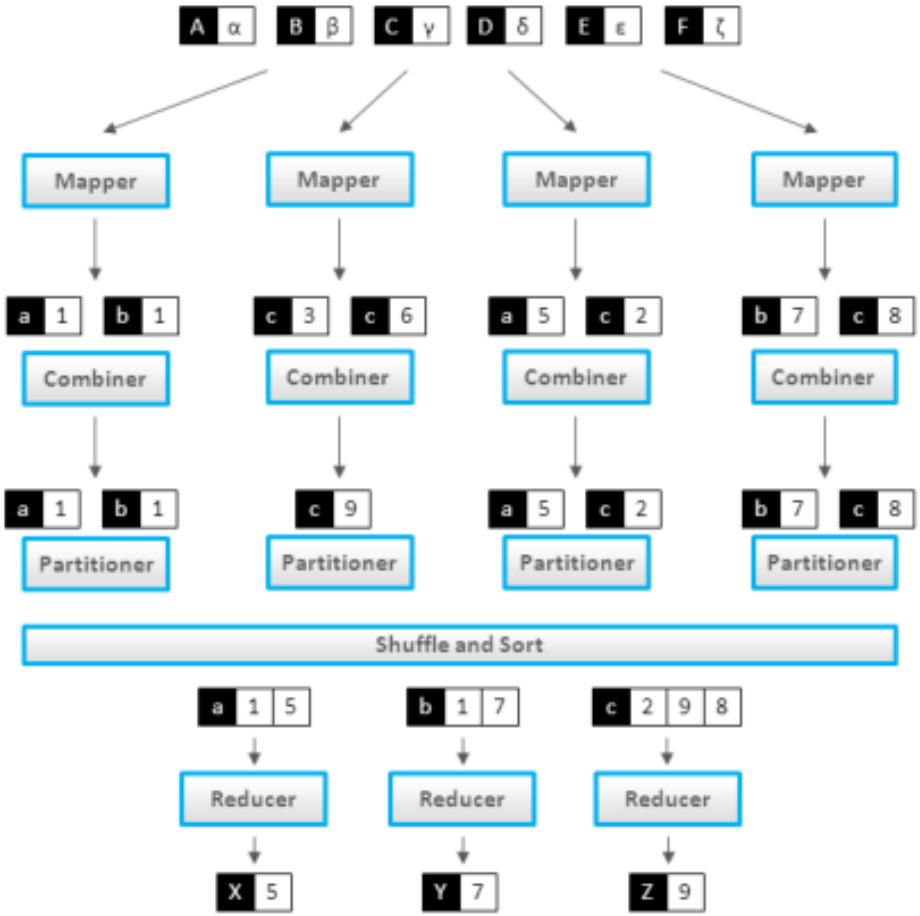

输入数据
分割结果
Hello World
Bye World
分割
〈0, “Hello World”〉
〈12, “Bye World”〉
Hello Hadoop
Bye Hadoop
分割
〈0, “Hello Hadoop”〉
〈13, “Bye Hadoop”〉


分割结果
map方法输出
〈0, “Hello World”〉
〈12, “Bye World”〉
map()
〈Hello, 1〉
〈World, 1〉
〈Bye, 1〉
〈World, 1〉
〈0, “Hello Hadoop”〉
〈13, “Bye Hadoop”〉
map()
〈Hello, 1〉
〈Hadoop, 1〉
〈Bye, 1〉
〈Hadoop, 1〉


map方法输出
排序结果
Combine输出
〈Hello, 1〉
〈World, 1〉
〈Bye, 1〉
〈World, 1〉
Map端排序
〈Bye, 1〉
〈Hello, 1〉
〈World, 1〉
〈World, 1〉
Combine过程
〈Bye, 1〉
〈Hello, 1〉
〈World, 2〉
〈Hello, 1〉
〈Hadoop, 1〉
〈Bye, 1〉
〈Hadoop, 1〉
Map端排序
〈Bye, 1〉
〈Hadoop, 1〉
〈Hadoop, 1〉
〈Hello, 1〉
Combine过程
〈Bye, 1〉
〈Hadoop, 2〉
〈Hello, 1〉


Combine输出
排序结果
Reduce输出
〈Bye, 1〉
〈Hello, 1〉
〈World, 2〉
〈Bye, 1〉
〈Hadoop, 2〉
〈Hello, 1〉
Reduce端排序
〈Bye, list(1, 1)〉
〈Hadoop, list(2)〉
〈Hello, list(1, 1)〉
〈Word, list(2)〉
reduce()
〈Bye, 2〉
〈Hadoop, 2〉
〈Hello, 2〉
〈Word, 2〉

# Combiner组件




- Combiner可看作是local reducer：
  - 合并相同的key对应的value（wordcount例子）；
  - 通常与reducer逻辑 一样；
- 好处
  - 减少Map Task输出数据量（磁盘IO）；
  - 减少reducer-map网络传输数据量（网络IO）；
- 如何正确使用
  - 结果可叠加；
  - Sun(YES!),Average(NO!)

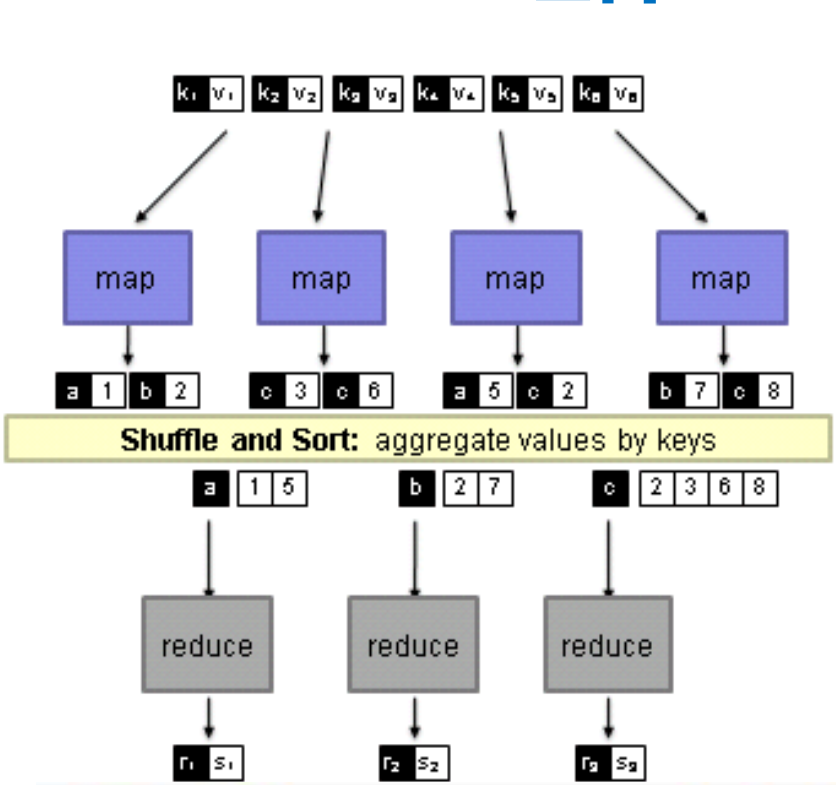
# Combiners组件

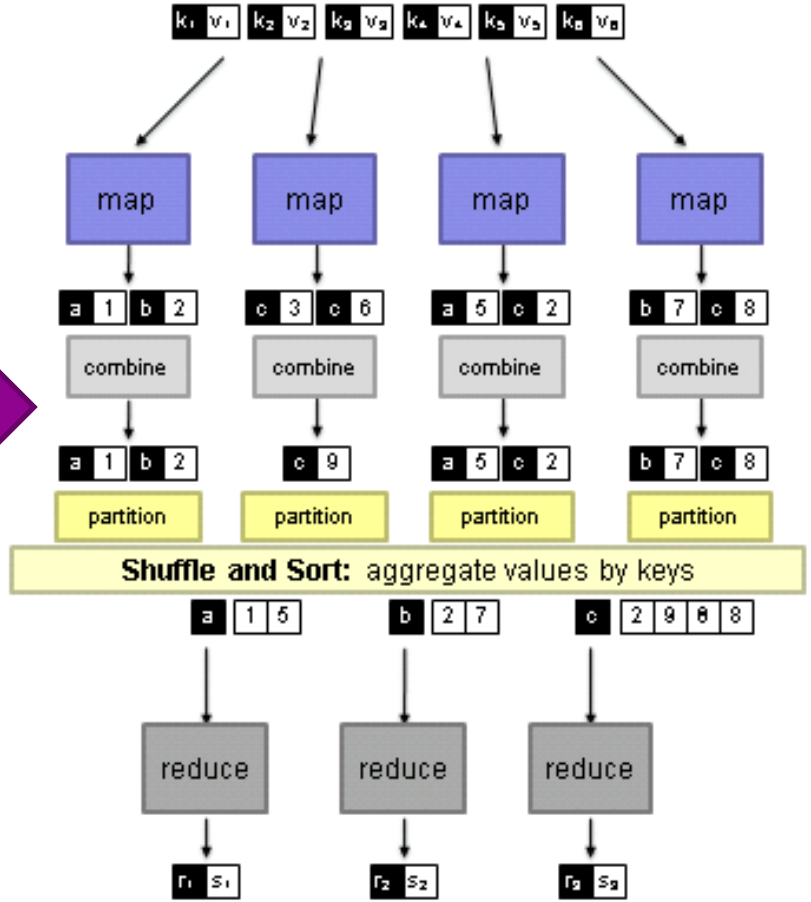
- combiner最基本是实现本地key的聚合，对map输出的key排序，value进行迭代。
  map: (K1, V1) → list(K2, V2)
  combine: (K2, list(V2)) → list(K2, V2)
  reduce: (K2, list(V2)) → list(K3, V3)
- combiner还具有类似本地的reduce功能，例如hadoop自带的wordcount的例子和找出value的最大值的程序，combiner和reduce完全一致。如下所示：
  map: (K1, V1) → list(K2, V2)
  combine: (K2, list(V2)) → list(K3, V3)
  reduce: (K3, list(V3)) → list(K4, V4)

# Combiners组件

- 对于hadoop自带的wordcount的例子，value就是一个叠加的数字，所以map一结束就可以进行reduce的value叠加，而不必要等到所有的map结束再去进行reduce的value叠加

# Combiners组件




不是所有MapReduce任务都适合使用combiner

# Combiners组件

第一个map的输出为：
（1950,0）（1950,20）（1950,10）
第二个map输出为：
（1950,25）（1950,15）（1950,30）
Reduce函数被调用是，输入如下：
（1950，[0,20,10,25,15,30]）
因为30是最大的值，所以输出（1950,30）
如果使用 combiner：那么reduce调用的时候传入的数据如下：
（1950，[20,30]）--　（1950,30）
Max(0,20,10,25,15,30)=max(max(0,20,10),max(25,15,30))=max(20,30)=30
计算最大值可以使用Combiners能提高效率。

# Combiners组件

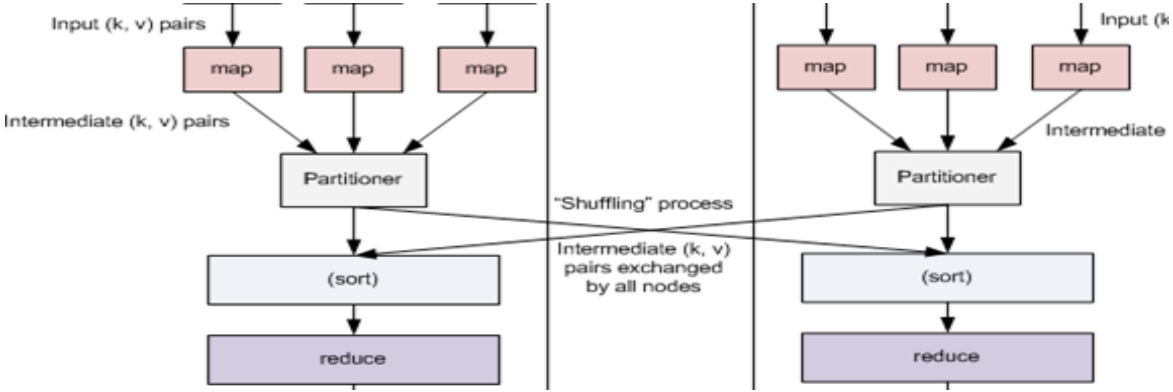
第一个map的输出为：
（1950,0）（1950,20）（1950,10）
第二个map输出为：
（1950,25）（1950,15）（1950,30）
Reduce函数被调用是，输入如下：
（1950，[0,20,10,25,15,30]）
因为30是最大的值，所以输出（1950,30）
如果使用 combiner：那么reduce调用的时候传入的数据如下：
（1950，[20,30]）--　（1950,30）
Max(0,20,10,25,15,30)=max(max(0,20,10),max(25,15,30))=max(20,30)=30
计算最大值可以使用Combiners能提高效率。

# Combiners组件

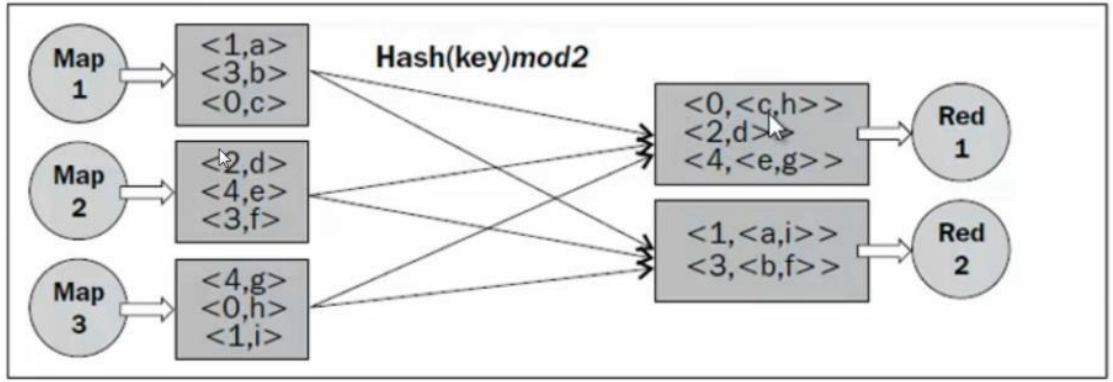
求平均值Avg（0,20,10,25,15,30） = 15
如果使用Combiner会得到什么样的结果呢？
第一个map输出为：
avg(0,20,10) = 10
第二个map输出为：
Avg（25,15,30） = 23
输入到reduce出来结果为：
Avg(10,23) = 17.5
17.5和15？

# Partitioner组件

- Map工作完成之后，每一个 Map 函数会将结果传到对应的Reducer所在的节点，此时用户提供一个Partitioner类，用来决定一个给定的(key,value)对传给哪个节点
- Partitioner 处于 Map阶段
- Mapper处理的数据，由Partitioner进行分区
- Mapper结果均匀分布到Reducer上面执行
- Partitioner 的默认实现：hash(key) mod R
- 自定义 Partitioner，继承Partitioner类，实现下面方法，其中numPartitions 为 Reduce 的个数

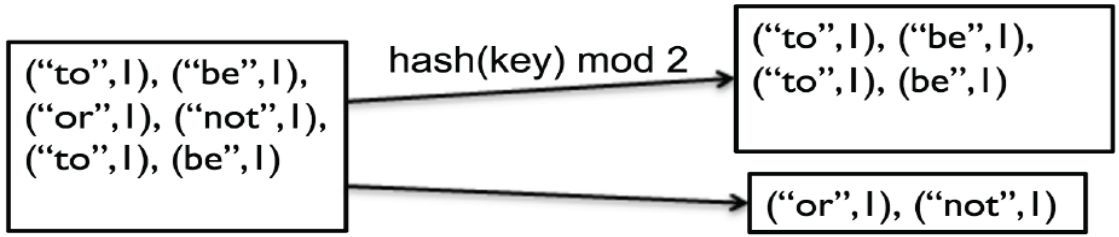



```
getPartition(Text key, Text value, int numPartitions);
```



# Partitioner组件

- Partitioner决定了Map Task输出的每条数据交给了哪个Reducer Task处理：
- 默认实现：hash(key)mod R
  - R是Reduce Task数目；



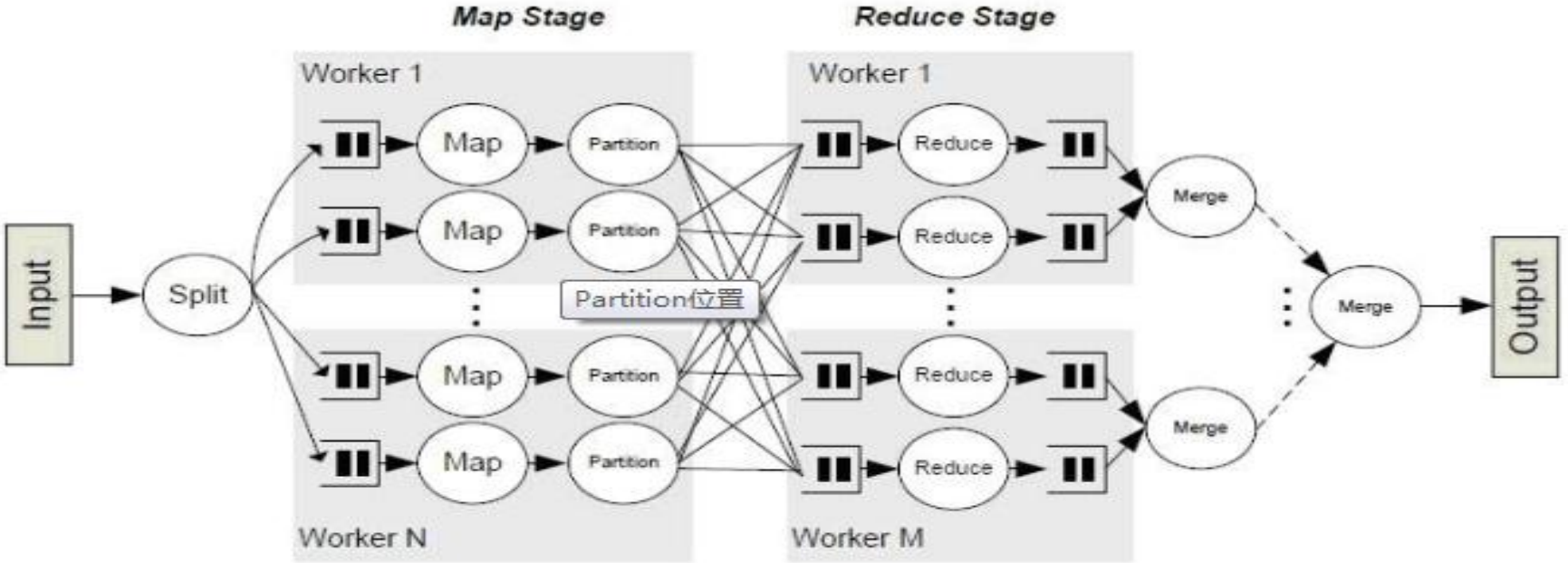

- 很多情况需自定义partitioner
  - 比如“hash(hostname(URL))mod R”确保相同域名的网页交给同一个reducer task 处理

# Partitioner组件

Partition主要作用就是将map的结果发送到相应的reduce。这就对partition有两个要求：

1）均衡负载，尽量的将工作均匀的分配给不同的reduce。

2）效率，分配速度一定要快。



# Partitioner

Partition主要是根据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 hash后再以reduce数据取模。

主要作业是均衡负载，尽量的将工作均匀的分配给不同的reduce。

MapReduce提供Partition编程接口。

例一个应用5个map，3个reduce（ 编号默认为0，1，2），如果一个数据key的hash值为1，则这个数据会分给编号为1 的reduce。Reduce的总数可以设置，也可以自动计算出来。

# Partitioner实例

## 根据成绩数据，统计出每个年龄段的男、女学生的最高分

```
Alice<tab>23<tab>female<tab>45
Bob<tab>34<tab>male<tab>89
Chris<tab>67<tab>male<tab>97
Kristine<tab>38<tab>female<tab>53
Connor<tab>25<tab>male<tab>27
Daniel<tab>78<tab>male<tab>95
James<tab>34<tab>male<tab>79
Alex<tab>52<tab>male<tab>69
Nancy<tab>7<tab>female<tab>98
Adam<tab>9<tab>male<tab>37
Jacob<tab>7<tab>male<tab>23
Mary<tab>6<tab>female<tab>93
Clara<tab>87<tab>female<tab>72
Monica<tab>56<tab>female<tab>92
```

| Size | Replication | Block Size | Name |
|---|---|---|---|
| 0 B | 1 | 128 MB | _SUCCESS |
| 67 B | 1 | 128 MB | part-r-00000 |
| 71 B | 1 | 128 MB | part-r-00001 |
| 71 B | 1 | 128 MB | part-r-00002 |

```
Nancy age-7 gender-female score-98
Adam  age-9 gender-male score-37
```

# Combiner + Partitioner实例

**根据明星搜索指数，分别统计出搜索指数最高的男明星和女明星**

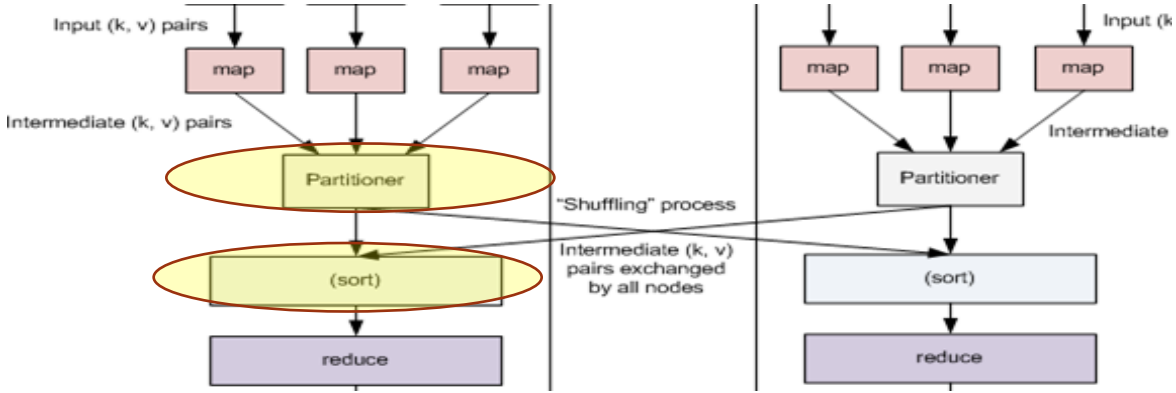
```
李易峰  male  32670
朴信惠  female  13309
林心如  female  5242
黄海波  male  5505
成龙  male  7757
刘亦菲  female  14830
angelababy  female  55083
王宝强  male  9472
郑爽  female  9279
周杰伦  male  42020
莫小棋  female  13978
朱一龙  male  10524
宋智孝  female  12494
吴京  male  6684
赵丽颖  female  24174
尹恩惠  female  5985
```

| Size | Replication | Block Size | Name |
|---|---|---|---|
| 0 B | 1 | 128 MB | _SUCCESS |
| 24 B | 1 | 128 MB | part-r-00000 |
| 21 B | 1 | 128 MB | part-r-00001 |

```
周杰伦  male  42020
```

```
angelababy  female  55083
```

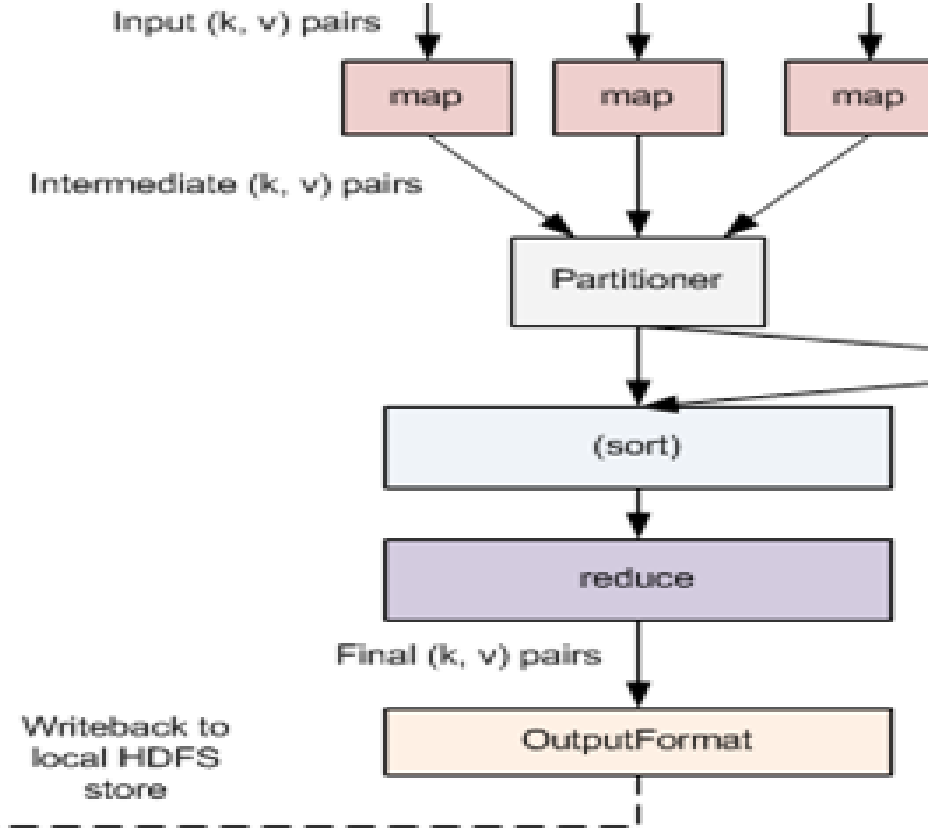
# Sort组件




传输到每一个Reducer节点上的、将被所有的Reduce函数接收到的Key,value对会被Hadoop自动排序（即Map生成的结果传送到某一个节点的时候，会被自动排序）

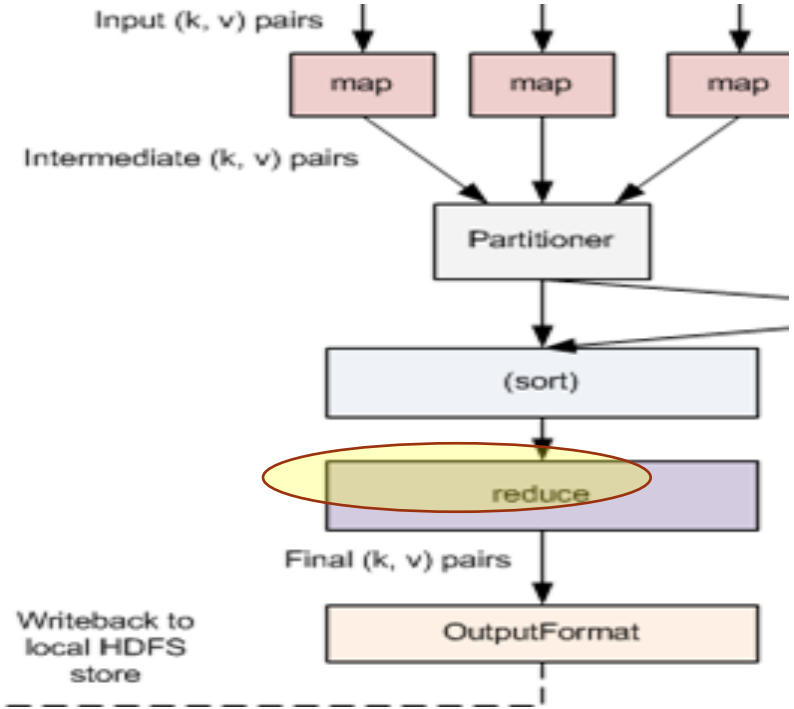
# Reduce组件

- 对map阶段的结果进行汇总
- Reducer的数目由mapred-site.xml配置文件里的项目mapred.reduce.tasks决定。缺省值为1，用户可以覆盖之
- 做用户定义的Reduce操作
- 接收到一个OutputCollector的类作为输出

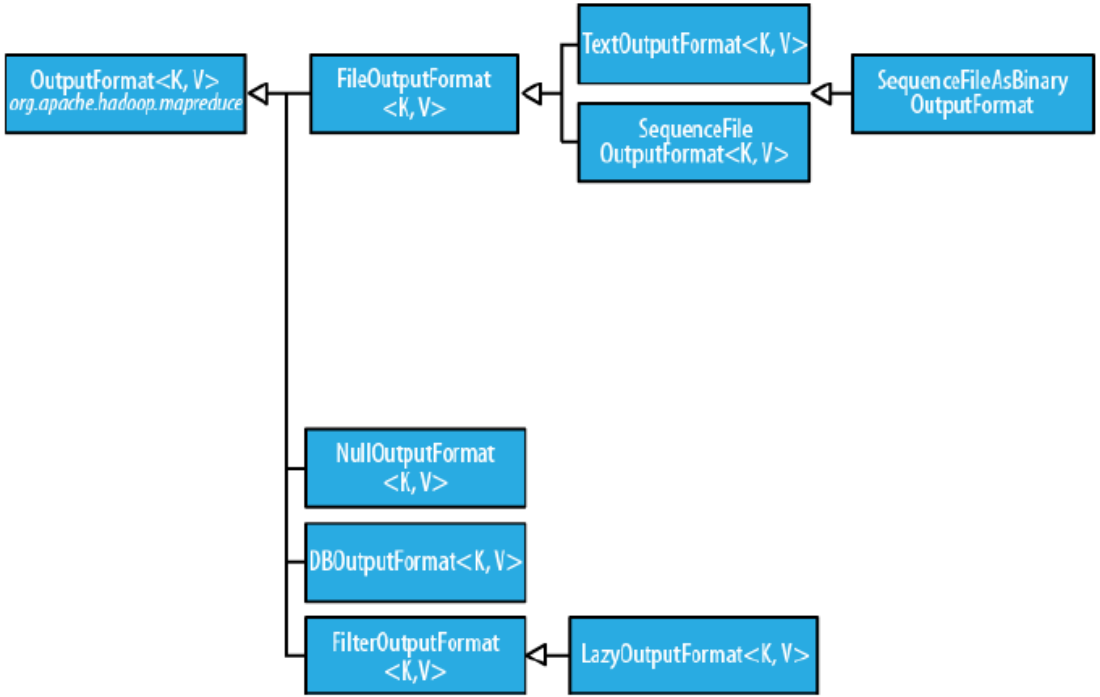


# OutputFormat文件输出格式组件

- 写入到HDFS的所有OutputFormat都继承自FileOutputFormat
- 每一个Reducer都写一个文件到一个共同的输出目录，文件名是part-nnnnn，nnnnn是与每一个reducer相关的一个号（partition id）
- FileOutputFormat.setOutputPath()
- JobConf.setOutputFormat()



# OutputFormat文件输出格式组件



# OutputFormat文件输出格式组件

| OutputFormat: | Description |
| --- | --- |
| TextOutputFormat | Default; writes lines in "key \t value" form |
| SequenceFileOutputFormat | Writes binary files suitable for reading into subsequent MapReduce jobs |
| NullOutputFormat | Disregards its inputs |

# MR输出格式—— 文本输出

- 默认输出格式是 TextOutputFormat
- 每条记录写为文本行
- 每个键/值对由制表符进行分割
- 设定 mapreduce.output.textoutputformat.separator 属性改变默认的分隔符
- NullWritable 可省略输出的键或值
- 两者都省略，相当于 NullOutputFormat ， 什么也不输出

# MR输出格式—— 多个输出

- 默认情况下只有一个 Reduce，输出只有一个文件
- 有时需要对输出的文件名进行控制或让每个 Reduce 输出多个文件
- 实现方式：MultipleOutputs
- 统计邮箱次数按邮箱类别输出到不同文件中

```
wolys@21cn.com
zss1984@126.com
294522652@qq.com
simulateboy@163.com
zhoushigang_123@163.com
sirenxing424@126.com
lixinyu23@qq.com
chenlei1201@gmail.com
370433835@qq.com
cxx0409@126.com
viv093@sina.com
q62148830@163.com
65993266@qq.com
summeredison@sohu.com
zhangbao-autumn@163.com
diduo_007@yahoo.com.cn
fxh852@163.com
```

```
/output/126-r-00000
/output/163-r-00000
/output/21cn-r-00000
/output/gmail-r-00000
/output/part-r-00000
/output/qq-r-00000
/output/sina-r-00000
/output/sohu-r-00000
/output/yahoo-r-00000
```

# MR多个输出——练习

- 将明星微博数据排序后按粉丝数 关注数 微博数 分别输出到不同文件中

数据格式：
明星 明星微博名称 粉丝数 关注数 微博数

```
黄晓明  黄晓明  22616497  506 2011
张靓颖  张靓颖  27878708  238 3846
张成龙2012  张成龙2012  9813621 199 744
羅志祥  羅志祥  30763518  277 3843
劉嘉玲  劉嘉玲  12631697  350 2057
吳君如大美女  吳君如大美女  18490338  190 412
柯震東Kai 柯震東Kai 31337479  219 795
李娜  李娜  23309493  81  631
徐小平  徐小平  11659926  1929  13795
唐嫣  唐嫣  24301532  200 2391
有斐君  有斐君  8779383 577 4251
孙燕姿  孙燕姿  21213839  68  342
成龙  成龙  22485765  5 758
...
```

| Size | Replication | Block Size | Name |
|---|---|---|---|
| 0 B | 1 | 128 MB | _SUCCESS |
| 3.72 KB | 1 | 128 MB | follower-r-00000 |
| 2.79 KB | 1 | 128 MB | friend-r-00000 |
| 0 B | 1 | 128 MB | part-r-00000 |
| 2.98 KB | 1 | 128 MB | statuses-r-00000 |

# RecordWriter组件

TextOutputFormat实现了缺省的LineRecordWriter，以”key\t value”形式输出一行结果。

# CompressionCode组件

- 数据压缩目的
  - 降低磁盘IO
  - 降低网络IO
  - 降低存储成本

Codec为压缩，解压缩的算法实现。
在Hadoop中，codec由CompressionCode的实现来表示。

| Compression format | Hadoop CompressionCodec |
|---|---|
| DEFLATE | org.apache.hadoop.io.compress.DefaultCodec |
| gzip | org.apache.hadoop.io.compress.GzipCodec |
| bzip2 | org.apache.hadoop.io.compress.BZip2Codec |
| LZO | com.hadoop.compression.lzo.LzopCodec |

Map任务输出的压缩属性：

| Property name | Type | Default value | Description |
|---|---|---|---|
| mapred.compress.map.output | boolean | false | Compress map outputs. |
| mapred.map.output.compression.codec | Class | org.apache.hadoop.io.compress.DefaultCodec | The compression codec to use for map outputs. |

# CompressionCode组件

| 压缩格式 | split | native | 压缩率 | 速度 | 是否hadoop自带 | linux命令 | 换成压缩格式后，原来的应用程序是否要修改 |
|---|---|---|---|---|---|---|---|
| gzip | 否 | 是 | 很高 | 比较快 | 是，直接使用 | 有 | 和文本处理一样，不需要修改 |
| lzo | 是 | 是 | 比较高 | 很快 | 否，需要安装 | 有 | 需要建索引，还需要指定输入格式 |
| snappy | 否 | 是 | 比较高 | 很快 | 否，需要安装 | 没有 | 和文本处理一样，不需要修改 |
| bzip2 | 是 | 否 | 最高 | 慢 | 是，直接使用 | 有 | 和文本处理一样，不需要修改 |

# CompressionCode组件

1）在程序中运用：
Configuration conf = **new Configuration();**
conf.setBoolean("mapred.output.compress",**true);**
conf.setClass("mapred.output.compression.codec",GzipCodec.**class,CompressionCodec.class);**
2）mapred.xml
name :mapred.output.compress . value:true
name:mapred.output.compression.codec“ value:
GzipCodec.**class,CompressionCodec.class**

# MapReduce中使用压缩

## Map输出结果压缩

单个job设置：

```
conf.setBoolean("mapreduce.map.output.compress",true);
conf.setClass("mapreduce.map.output.compress.codec",GzipCodec.class,CompressionCodec.class);
```

全局设置(mapred-site.xml)：

mapreduce.map.output.compress设置为true

mapreduce.map.output.compress.codec设置为相应的codec

## Reduce输出结果压缩

单个job设置：

```
conf.setBoolean("mapreduce.output.fileoutputformat.compress",true);
conf.setClass("mapreduce.output.fileoutputformat.compress.codec",GzipCodec.class,CompressionCodec.class);
```

全局设置(mapred-site.xml)：

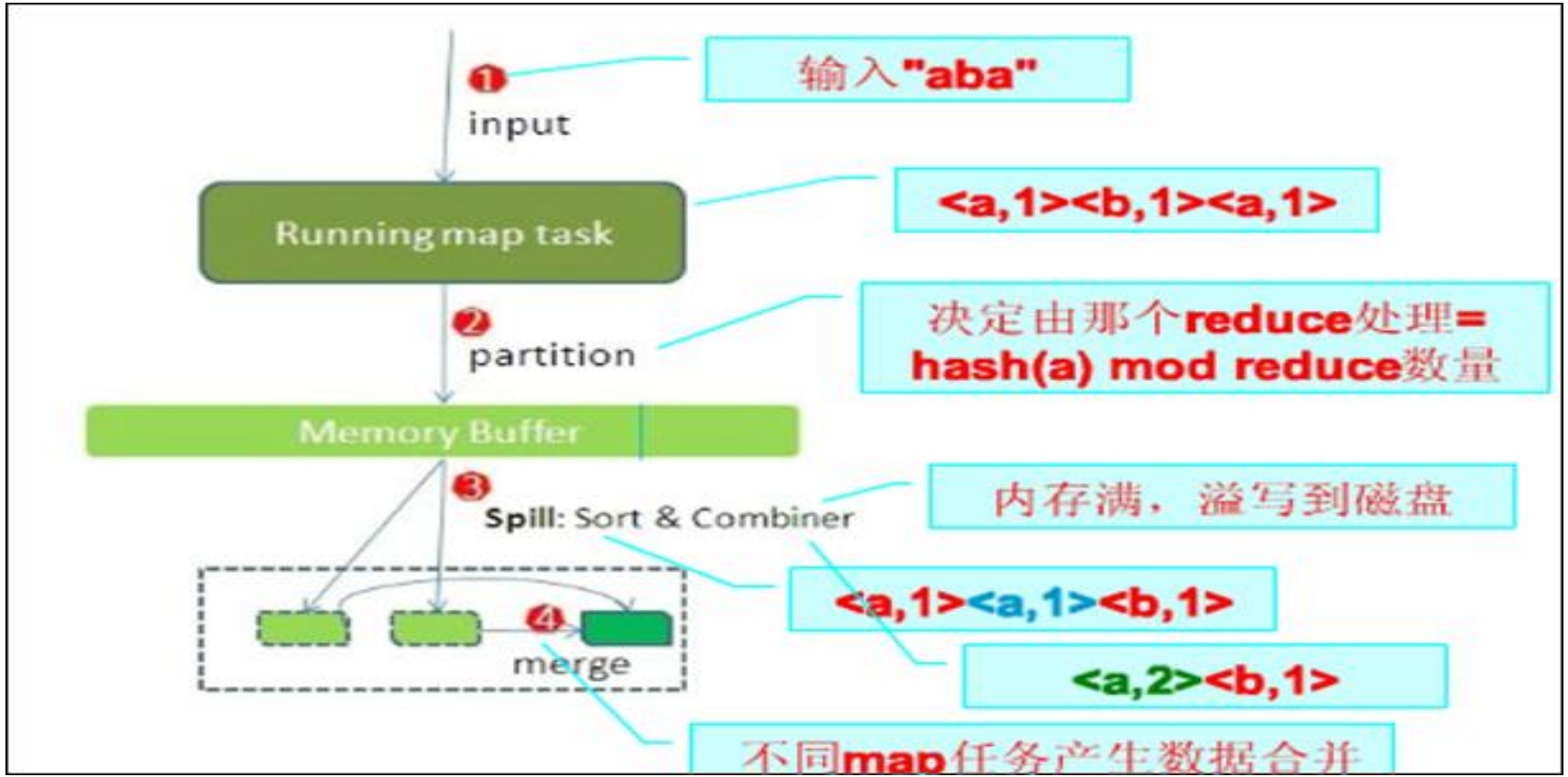
mapreduce.output.fileoutputformat.compress属性设置为true。

mapreduce.output.fileoutputformat.compress.codec设置为相应的codec

# Shuffler

- Shuffler是描述数据从Map输出到Reduce输入的过程，可以把mapper的输出按照某种key值重新切分和组合成n份，把key值符合某种范围的输出送到特定的reducer那里去处理
- 可以简化reducer过程
- Map端的Shuffler过程。
  - Patitioner
  - Spill
  - Combiner
  - Merge
- Reduce端的Shuffler过程。
  - Copy
  - Merge

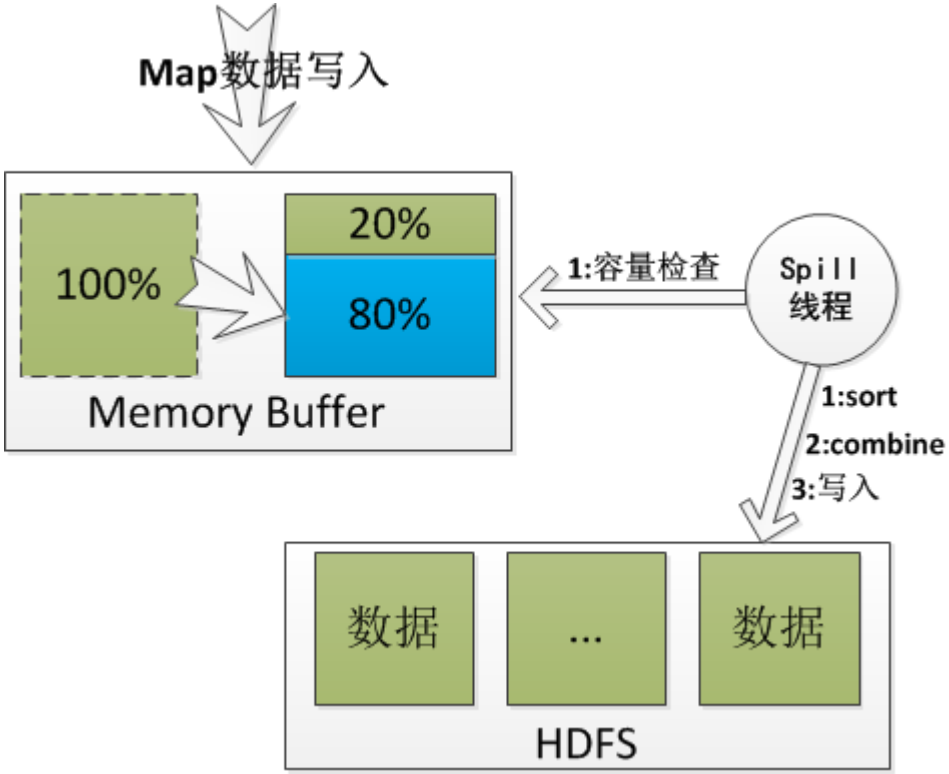
# Map端的shuffle过程



# Map端的shuffle过程

- 每一个Map Task把输入写到内存的buffer中：
  - Map阶段和Reduce阶段
- 当内存buffer达到一定阈值时，后台线程会把数据写到磁盘
  - 根据Partition，把数据写入到不同的partition；
  - 对于每个partition的数据进行排序;
  - 运行Combiner;

# Map端的shuffle过程




1)Map中的数据会先写入到内存buffer中。
2)Spill线程会检查buffer，当达到80%时启动spill程序。
3)Spill会将原80%内存中的数据按key进行sort。
4)再将数据combine(相当于在本地执行一次reduce操作)
5)最后将数据写入HDFS。
6)在这个过程中，Map中的数据会继续向剩余20%的内存buffer中写入。

# Map端的shuffle过程

- 随着Map Task的运行，磁盘上的spill越来越多：
  - 对于一个partition下的spill分片，会进行合并（把多个spill合并）；
  - 运行Combiner
- Redece Task启动后，从每个Map Task上远程拷贝属于它的partition文件；
  - 把这些文件归并排序;

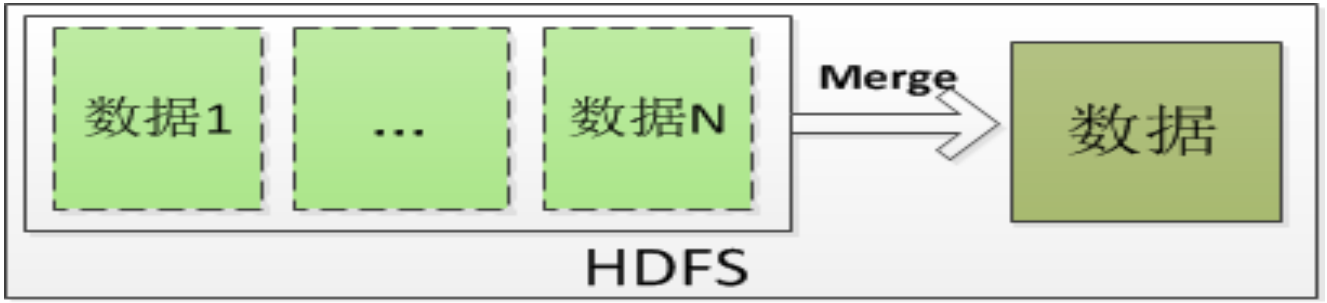
# Merge

- Map结束时，会将本分区的所有Spill产生文件合并在一起，形成一个已分区已排序的文件，这个过程称为Merge。合并后结果形式，大致如下：

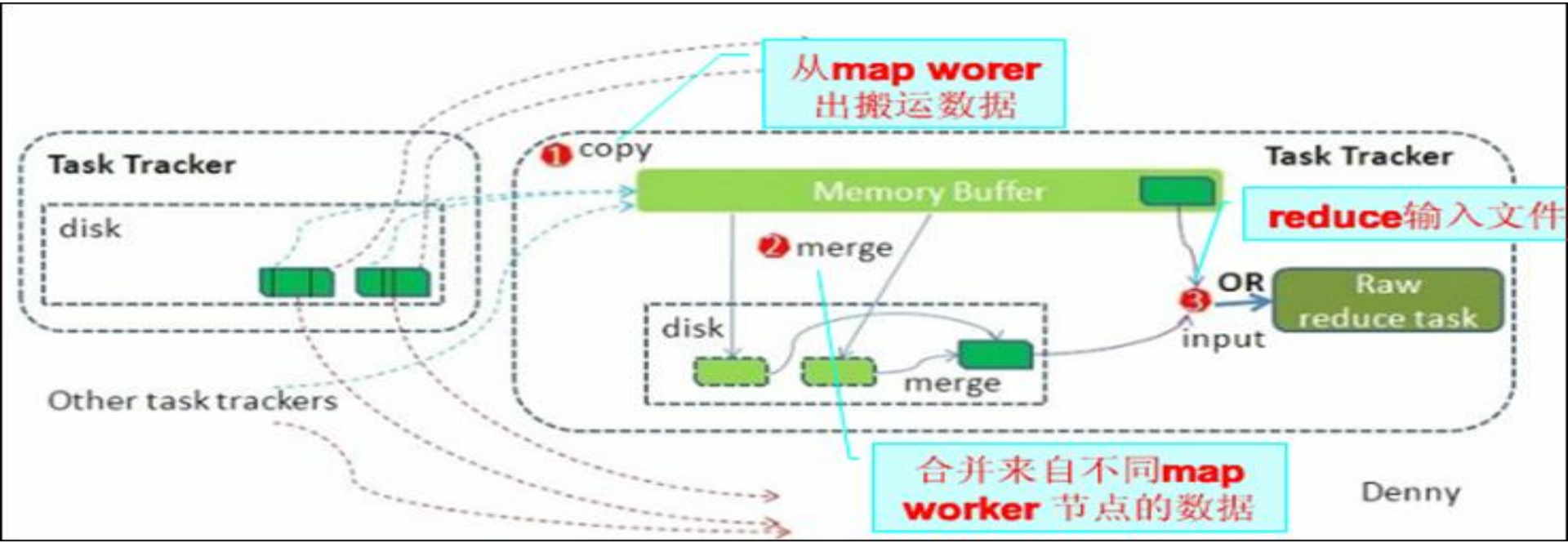
| key | value |
| --- | --- |
| aaa | 4 |
| aaa | 2 |
| bbb | 8 |
| bbb | 6 |

→

| key | value |
| --- | --- |
| aaa | [4, 2] |
| bbb | [8, 6] |



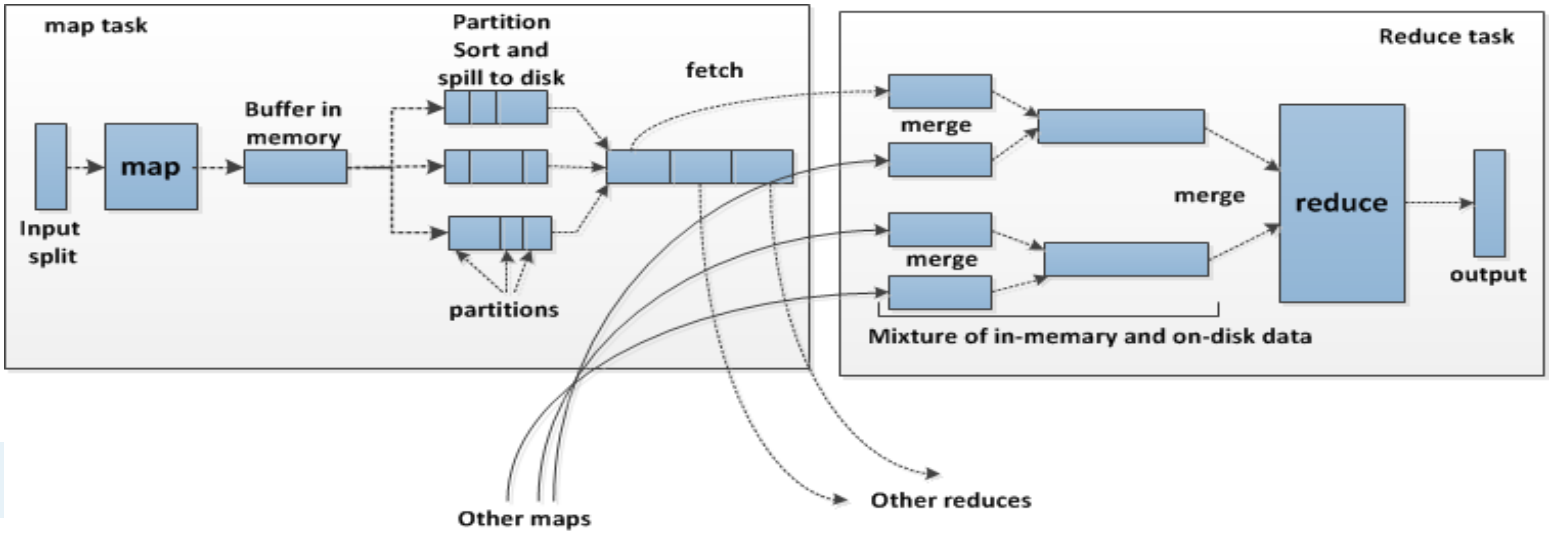
# reduce端的shuffle过程




- Copy过程，从各Map中获取属于自己的文件。
- Merge阶段，不停的对获取的文件进行Merge操作，这里也会执行Combiner。
- 形成Reduce的输入 文件。

# shuffle过程

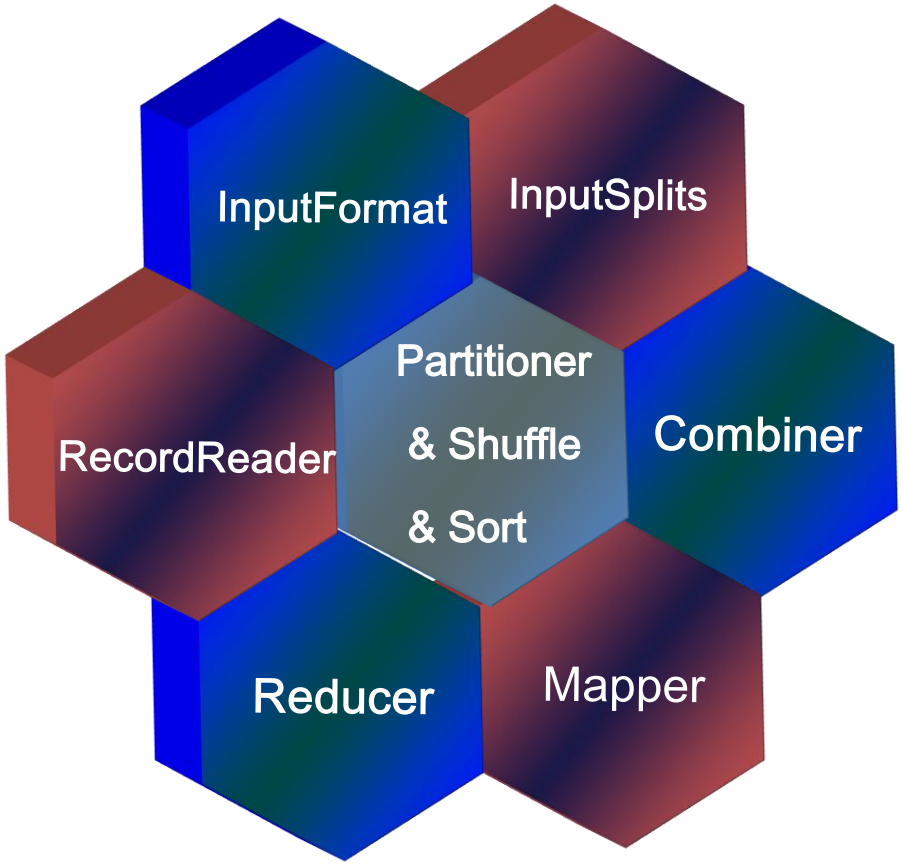
- shuffle指将map输出做为输入传给reduce的过程
- 每个输入分片会让一个map任务来处理，默认以HDFS的一个块大小为一个分片。map输出结果暂且放在一个环形内存缓冲区中（默认大小为100M，由io.sort.mb属性控制），当该缓冲区快要溢出时（默认为缓冲区大小的80%，由io.sort.spill.percent属性控制），会在本地文件系统中创建一个溢出文件，将该缓冲区中的数据写入这个文件。



# shuffle过程

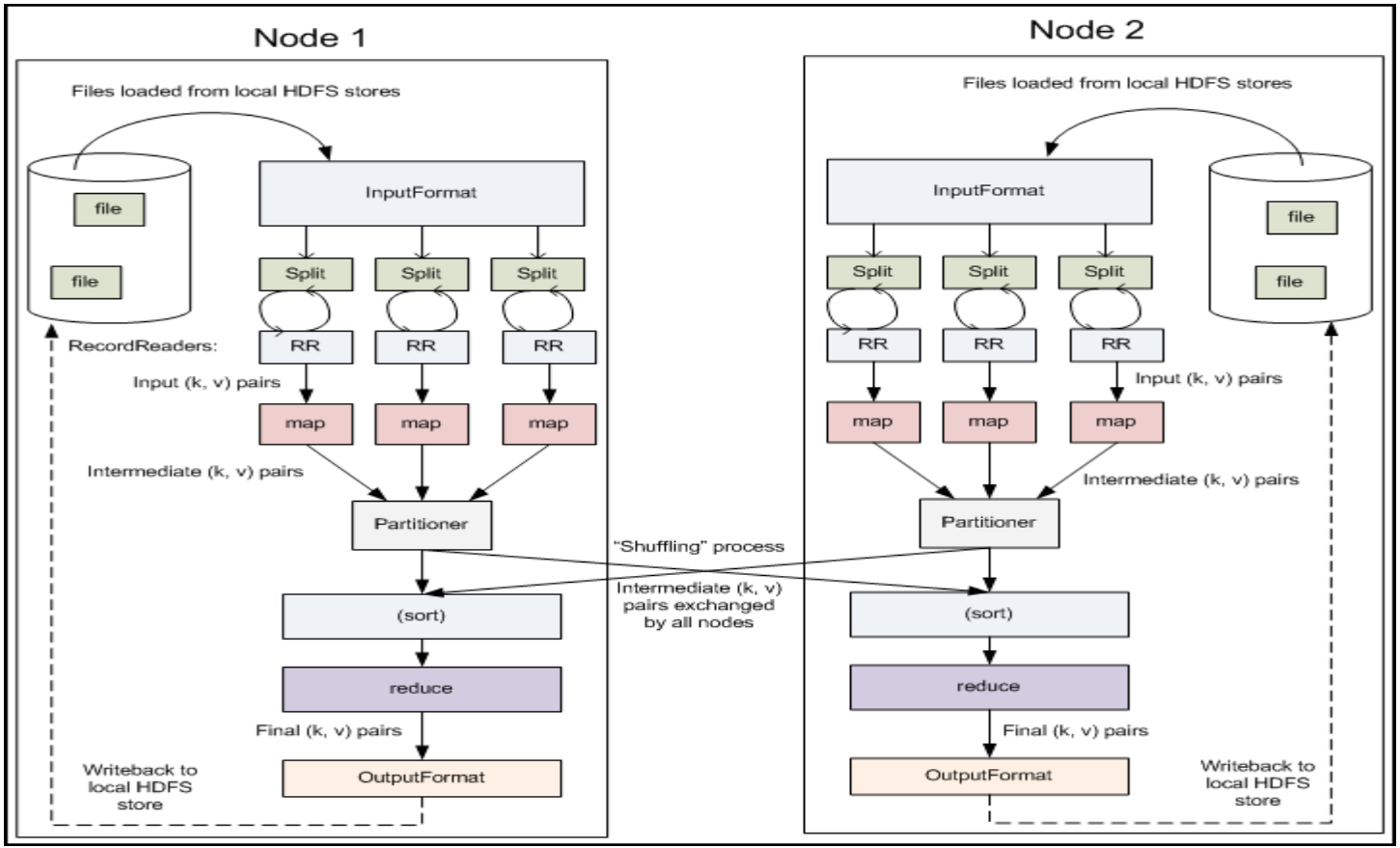
- 写入磁盘前，线程首先根据reduce任务数目将数据划分为相同数目的分区，也就是一个reduce任务对应一个分区的数据。这样做是为了避免有些reduce任务分配到大量数据而有些reduce任务却分到很少数据，甚至没有分到数据的尴尬局面。分区就是对数据进行hash的过程。然后对每个分区中的数据进行排序，如果此时设置了Combiner，将排序后的结果进行Combiner操作，目的是让尽可能少的数据写入到磁盘。
- 当map任务输出最后一个记录时，可能会有很多的溢出文件，这时需要将这些文件合并。合并的过程中会不断地进行排序和Combiner操作，目的有两个：1.尽量减少每次写入磁盘的数据量；2.尽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，将mapred.compress.map.out设置为true。
- 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NM保持联系，而NM又一直和RM保持心跳。所以RM中保存了整个集群中的宏观信息。只要reduce任务向RM获取对应的map输出位置就ok了。

# MR核心组件




- 组件就是对象，是对数据和方法的简单封装。组件有自己的属性和方法。属性是组件数据的简单访问者。方法则是组件的一些简单而可见的功能。
- 类是实现了源代码级的重用，是静态重用；组件是要实现二进制的重用，动态重用，并最终实现搭积木式的系统构造的梦想。
- 在组件发展中，也发展了很多的强大特性，如：分布式组件，分布式组件极大的提高了系统的灵活性，伸缩性（负载伸缩）和可维护性。

# MR核心组件


Node 1
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store
Node 2
Files loaded from local HDFS stores
InputFormat
file
file
Split
Split
Split
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

# 参与计算角色

- JobClient：每一个job都会在用户端通过JobClient类将应用程序以及配置参数打包成jar文件存储在HDFS，并把路径提交到RM ， RM创建每一个Task（MapTask和ReduceTask）并将它们分发到各个NM服务中去执行。
- RM：hadoop的Map/Reduce调度器，负责与NM通信分配计算任务并跟踪任务进度。RM是一个master服务，软件启动之后RM接收job，负责调度job的每一个子任务task运行于AM上，并监控它们，如果发现有失败的task就重新运行它。 RM部署在单独的机器上。
- NM：运行于多个节点上的slaver服务。NM主动与RM通信，接收作业，并负责直接执行每一个任务。AM都需要运行在HDFS的DataNode上，每个节点只有一个NM，但一个NM可以启动多个JVM，用于并行执行map或reduce任务

# 关系型数据库与MapReduce

| | 传统关系型数据库 | MapReduce |
|---|---|---|
| 数据大小 | GB | PB |
| 访问 | 交互型和批处理 | 批处理 |
| 更新 | 多次读写 | 一次写入多次读取 |
| 结构 | 静态模式 | 动态模式 |
| 集成度 | 高 | 低 |
| 伸缩性 | 非线性 | 线性 |

# Mapreduce的编程接口

- Hadoop提供两种编程方式；
  - JAVA（最原始方式）；
  - Hadoop Streaming(支持多语言)；
- JAVA编程接口是所有编程方式的基础；
- 不同的编程接口只是暴露给用户的形式不同而已，内部执行引擎是一样的；
- 不同编程方式效率不同；

# Java序列化与反序列化

- 序列化是将对象状态转换为可保持或传输的格式的过程。与序列化相对的是反序列化，它将流转换为对象。这两个过程结合起来，可以轻松地存储和传输数据。
- 什么情况下需要序列化
- a）把内存中对象保存到一个文件中或者数据库中；
- b）用套接字在网络上传送对象；
- c）通过RMI传输对象；

# Java序列化与反序列化

- 序列化前，每个保存在堆（Heap）中的对象都有相应的状态，即实例变量（instance ariable）。
- 如:
- Foo myFoo = new Foo();
- myFoo .setWidth(37);
- myFoo.setHeight(70);
- 当通过下面的代码序列化之后，MyFoo对象中的width和Height实例变量的值（37，70）都被保存到foo.ser文件，这样以后又可以把它从文件中读出来，重新在堆中创建原来的对象。保存时候不仅仅是保存对象的实例变量的值，JVM还要保存一些小量信息，比如类的类型等以便恢复原来的对象。
- FileOutputStream fs = new FileOutputStream("foo.ser");
  ObjectOutputStream os = new ObjectOutputStream(fs);
  os.writeObject(myFoo);

# Java序列化步骤

a）Make a FileOutputStream

```
FileOutputStream fs = new FileOutputStream("foo.ser");
```

b）Make a ObjectOutputStream

```
ObjectOutputStream os = new ObjectOutputStream(fs);
```

c）write the object

```
os.writeObject(myObject1);
os.writeObject(myObject2);
os.writeObject(myObject3);
```

d) close the ObjectOutputStream

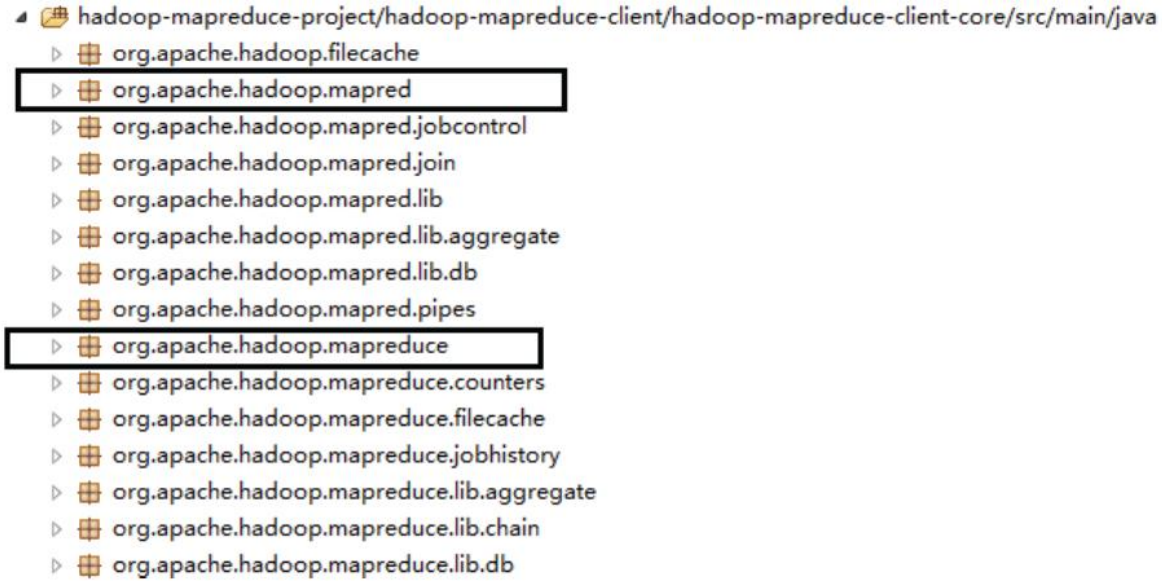
```
os.close();
```

# Java编程接口

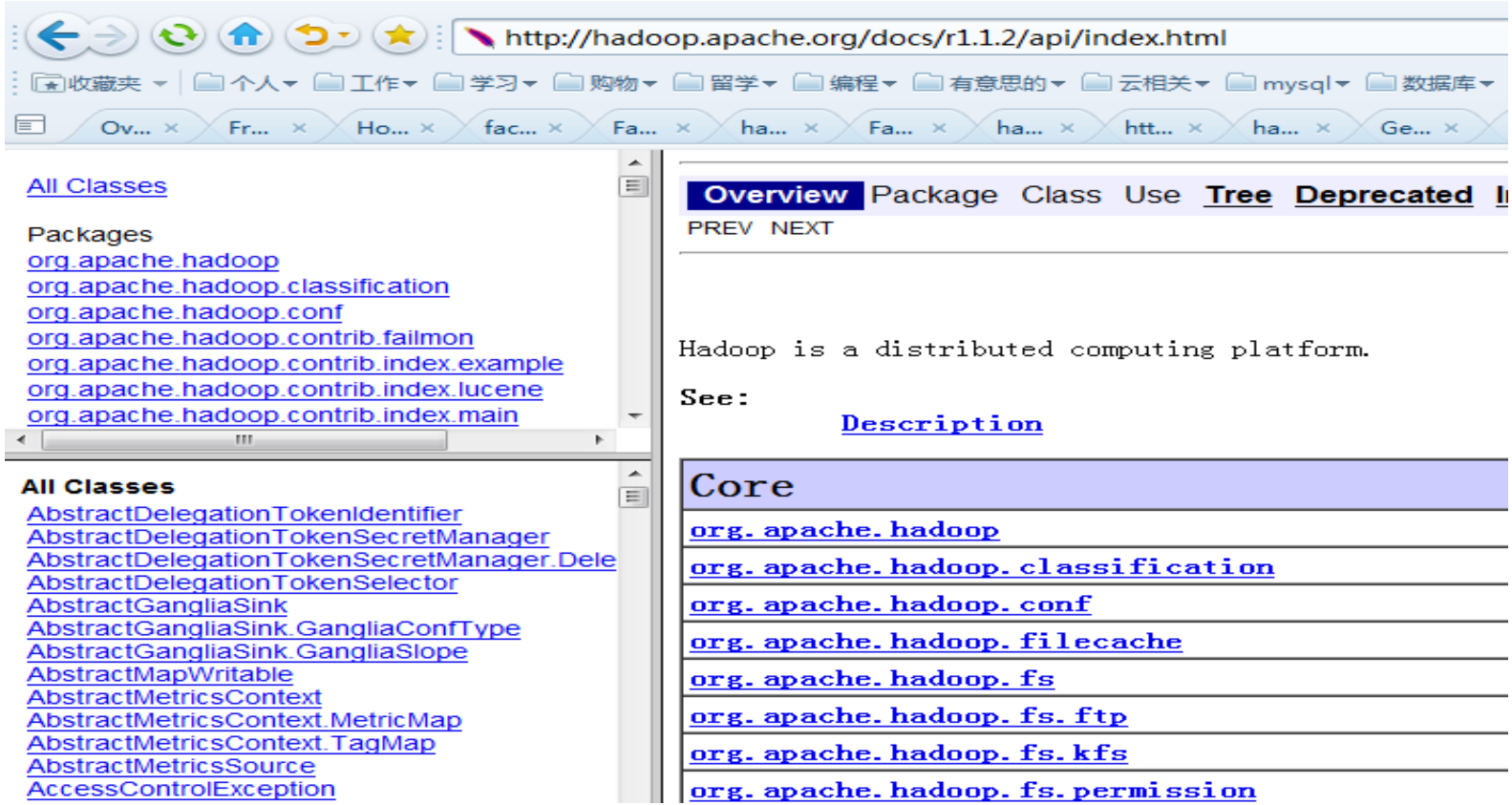
- Java编程接口组成；
  - 旧API：所有java包：org.apache.Hadoop.mapred；
  - 新API：所有java包： org.apache.Hadoop.mapreduce；
- 新API具有良好的扩展性；
- 两种编程接口只是暴露给用户的形式不同而已，内部执行引擎是一样的

# Java新旧API

- 从hadoop1.0.0开始，所有发型版均包含新旧两类API



# Hadoop api


http://hadoop.apache.org/docs/r1.1.2/api/index.html
All Classes
Packages
org.apache.hadoop
org.apache.hadoop.classification
org.apache.hadoop.conf
org.apache.hadoop.contrib.failmon
org.apache.hadoop.contrib.index.example
org.apache.hadoop.contrib.index.lucene
org.apache.hadoop.contrib.index.main
All Classes
AbstractDelegationTokenIdentifier
AbstractDelegationTokenSecretManager
AbstractDelegationTokenSecretManager.Dele
AbstractDelegationTokenSelector
AbstractGangliaSink
AbstractGangliaSink.GangliaConfType
AbstractGangliaSink.GangliaSlope
AbstractMapWritable
AbstractMetricsContext
AbstractMetricsContext.MetricMap
AbstractMetricsContext.TagMap
AbstractMetricsSource
AccessControlException
Overview Package Class Use Tree Deprecated
PREV NEXT
Hadoop is a distributed computing platform.
See:
Description
Core
org.apache.hadoop
org.apache.hadoop.classification
org.apache.hadoop.conf
org.apache.hadoop.filecache
org.apache.hadoop.fs
org.apache.hadoop.fs.ftp
org.apache.hadoop.fs.kfs
org.apache.hadoop.fs.permission

# Hadoop api调用

```
static FileSystem get(Configuration conf)
operator()
    {
       //step1
       得到Configuration对象
       //step2
       得到FileSystem对象
       //step3
       进行文件操作
    }
```

# Hadoop api

org.apache.hadoop.conf 定义了系统参数的配置文件处理API
org.apache.hadoop.fs 定义了抽象的文件系统API。
org.apache.hadoop.dfs Hadoop分布式文件系统（HDFS）模块的实现。
org.apache.hadoop.io 定义了通用的I/O API，用于针对网络，数据库，文件等数据对象做读写操作。
org.apache.hadoop.ipc 用于网络服务端和客户端的工具，封装了网络异步I/O的基础模块。
org.apache.hadoop.mapred Hadoop分布式计算系统（MapReduce）模块的实现，包括任务的分发调度等。
org.apache.hadoop.metrics 定义了用于性能统计信息的API，主要用于mapred和dfs模块。
org.apache.hadoop.record 定义了针对记录的I/O API类以及一个记录描述语言翻译器，用于简化将记录序列化成语言中性的格式（language-neutral manner）。
org.apache.hadoop.tools 定义了一些通用的工具。
org.apache.hadoop.util 定义了一些公用的API。

# 谢谢！