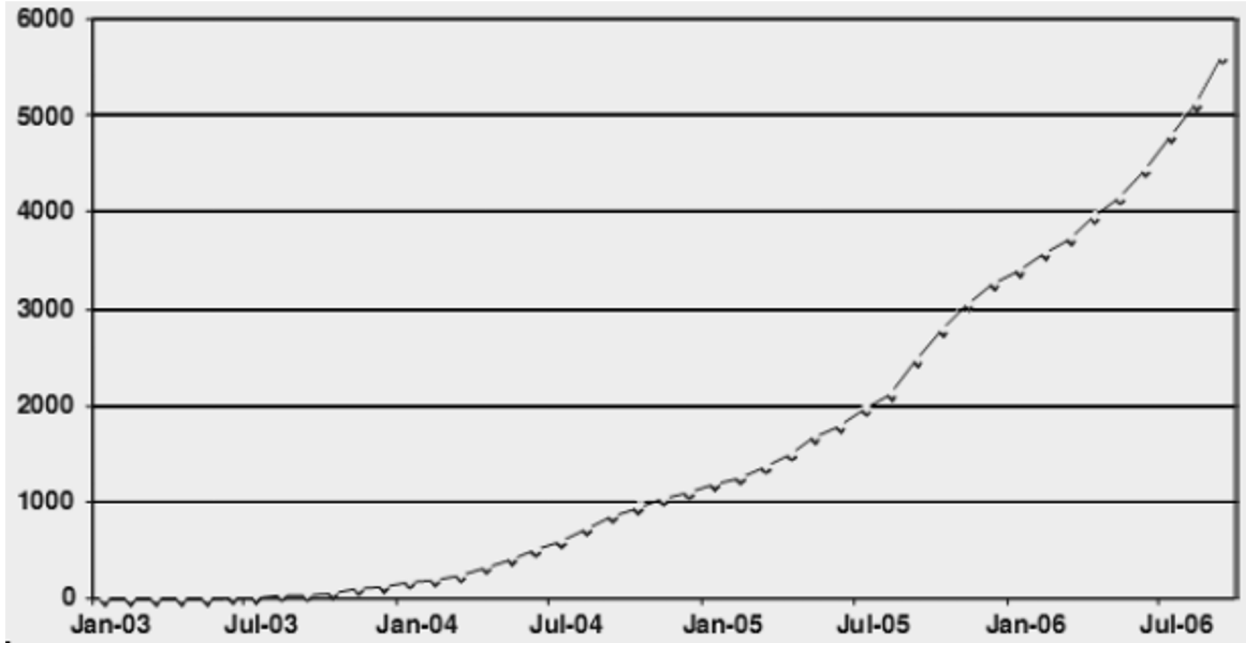
MapReduce基础算法程序设计 (I)

摘要

- MapReduce可解决哪些算法问题?
- 回顾：MapReduce流水线
- MapReduce WordCount1.0
- MapReduce WordCount2.0
- MapReduce矩阵乘法
- MapReduce关系代数运算

应用范围

- 自MapReduce发明后，Google大量用于各种海量数据处理。目前Google内部有7千以上的程序基于MapReduce实现。MapReduce可广泛应用于搜索引擎（文档倒排索引，网页链接图分析与页面排序等）、Web日志分析、文档分析处理、机器学习、机器翻译等各种大规模数据并行计算应用领域各类大规模数据并行处理算法。



基本算法

各种全局数据相关性小、能适当划分数据的计算任务，如：

- 分布式排序
- 分布式GREP(文本匹配查找)
- 关系代数操作

  如：选择，投影，求交集、并集，连接，成组，聚合…
- 矩阵向量相乘、矩阵相乘
- 词频统计(word count)，词频重要性分析(TF-IDF)
- 单词同现关系分析

  典型的应用如从生物医学文献中自动挖掘基因交互作用关系
- 文档倒排索引
- ……

复杂算法及应用

- Web搜索引擎
  - 网页爬取、倒排索引、网页排序、搜索算法
- Web访问日志分析
  - 分析和挖掘用户在Web上的访问、购物行为特征、以定制个性化用户界面或投放用户感兴趣的产品广告
- 数据/文本统计分析
  - 如科技文献引用关系分析和统计、专利文献引用分析和统计
- 图算法
  - 并行化宽度优先搜索(最短路径问题，可克服Dijkstra串行算法的不足)，最小生成树，子树搜索、比对
  - Web链接图分析算法PageRank，垃圾邮件连接分析
- 聚类(clustering)
  - 文档聚类、图聚类、其它数据集聚类

复杂算法及应用

- 相似性比较分析算法
  - 字符序列、文档、图、数据集相似性比较分析
- 基于统计的文本处理
  - 最大期望(EM)统计模型，隐马可夫模型(HMM)，……
- 机器学习
  - 监督学习、无监督学习、分类算法(决策树、SVM…)
- 数据挖掘
- 统计机器翻译
- 生物信息处理
  - DNA序列分析比对算法Blast：双序列比对、多序列比对
  - 生物网络功能模块(Motif)查找和比对
- 广告推送与推荐系统
- ……

MapReduce算法应用专著

1.Mining of Massive Datasets

2010, Jure Leskovec (Stanford Univ.), Anand Rajaraman (Kosmix, Inc), Jeffrey D. Ullman (Stanford Univ.)

主要介绍基于MapReduce的大规模数据挖掘相关的技术和算法，尤其是Web或者从Web导出的数据

Ch3. Similarity search, including the key techniques of minhashing and locality-sensitive hashing.

Ch4. Data-stream processing and specialized algorithms for dealing with data that arrives so fast it must be processed immediately or lost.

Ch5. The technology of search engines, including Google's PageRank, link-spam detection, and the hubs-and-authorities approach(a link analysis algorithm: Hyperlink-Induced Topic Search (HITS)).

Ch6. Frequent-itemset mining, including association rules, market-baskets, the A-Priori Algorithm and its improvements (a classic algorithm for learning association rules).

Ch7. Algorithms for clustering very large, high-dimensional datasets.

Ch8. Two key problems for Web applications: managing advertising and recommendation systems.

MapReduce算法应用专著

2. Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer，2010，University of Maryland, College Park

主要介绍基于MapReduce的大规模文档数据处理技术和算法

Ch4. Inverted Indexing for Text Retrieval

Ch5. Graph Algorithms

- Parallel Breadth-First Search
- PageRank

Ch6. EM Algorithms for Text Processing

- EM, HMM
- Case Study: Word Alignment for Statistical Machine Translation

回顾：MapReduce流水线

MapReduce Pipeline

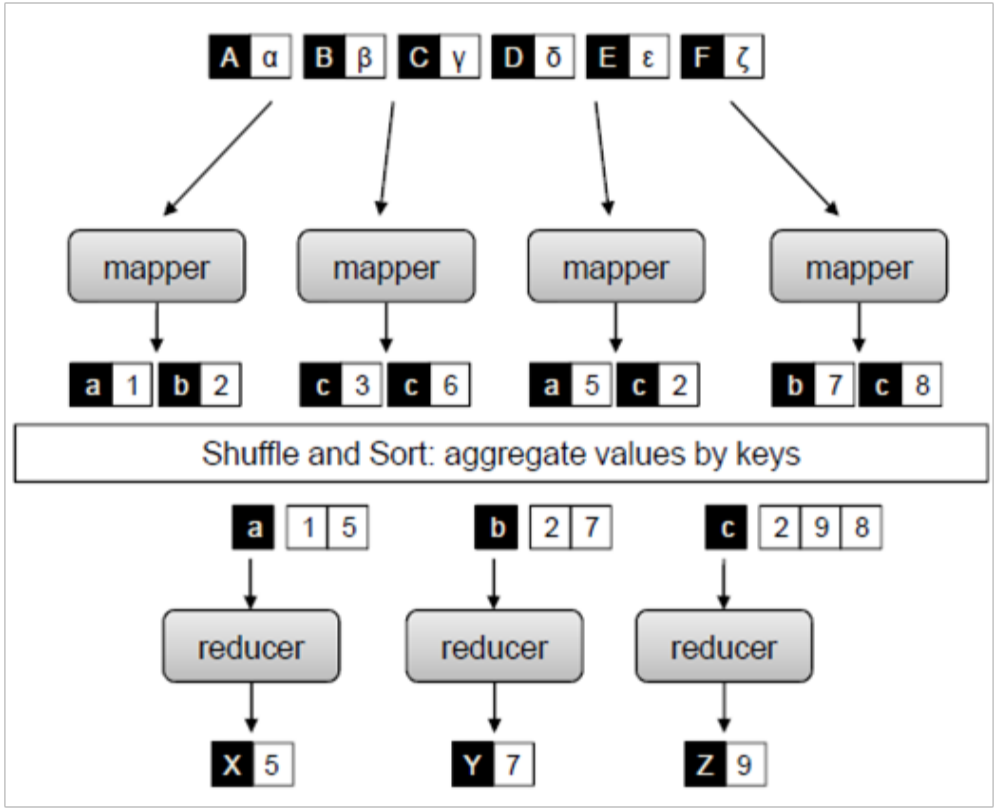
map(K1,V1)->[(K2,V2)]

shuffle and sort

reduce(K2,[V2]) -> [(K3,V3)]

([…] denotes a list)

Any algorithm that you wish to develop must be expressed in terms of such rigidly-defined components



回顾：MapReduce流水线

- Mapper
  - Initialize: setup()
  - map(): It is called once for each key/value pair in the input split. The default is the identity function.
  - Close: cleanup()
- Shuffle
  - Shuffle phase needs the *Partitioner* to route the output of mapper to reducer.
  - *Partitioner* controls the partitioning of the keys of the intermediate map-outputs. The key is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job.
  - *HashPartitioner* is the default Partitioner.

回顾：MapReduce流水线

- Sort
  - We can controls how the keys are sorted before they are passed to the Reducer by using a customized comparator.
- Reducer
  - Initialize: setup()
  - reduce(): It is called once for each key. The default implementation is an identity function.
  - Close: cleanup()

常用数据类型

- 这些数据类型都实现了WritableComparable接口，以便进行网络传输和文件存储，以及进行大小比较。

Class	Description
BooleanWritable	Wrapper for a standard Boolean variable
ByteWritable	Wrapper for a single byte
DoubleWritable	Wrapper for a Double
FloatWritable	Wrapper for a Float
IntWritable	Wrapper for a Integer
LongWritable	Wrapper for a Long
Text	Wrapper to store text using the UTF8 format
NullWritable	Placeholder when the key or value is not needed

Hadoop API文档

- **Apache Hadoop Main 3.3.6 API**
  - Common; HDFS; MapReduce; YARN
  - https://hadoop.apache.org/docs/stable/api/index.html

MapReduce User Interface

- Mapper
  - map
    - How many maps?
  - combine
- Reducer
  - shuffle
  - sort/secondary sort
  - Reduce
    - How many reduces?
- Partitioner
- Counter

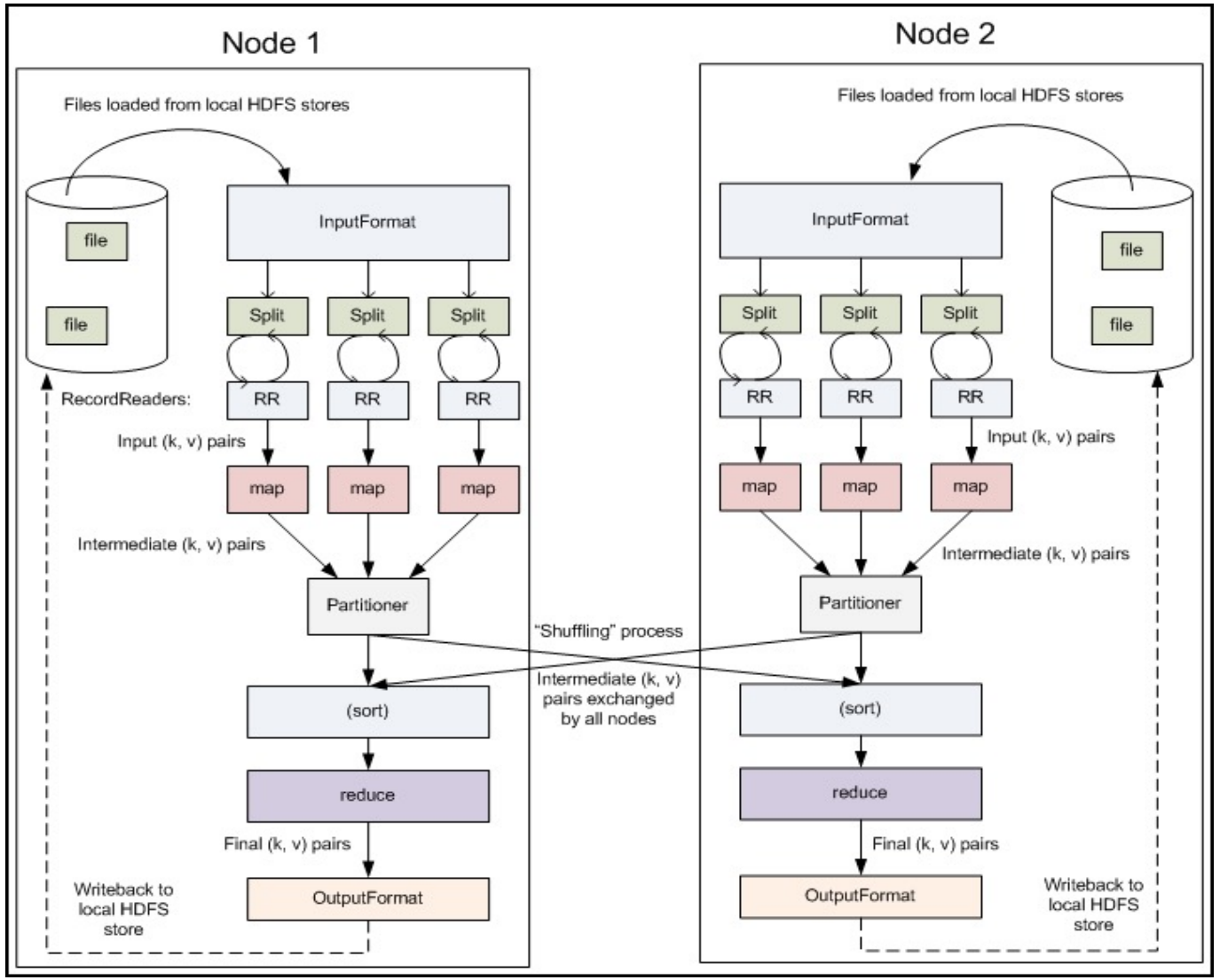
MapReduce User Interface

- Job Configuration
  - Job represents a MapReduce job configuration
- Task Execution & Environment
  - The MRAppMaster executes the Mappper/Reducer task as a child process in a separate JVM.
- Job Submission and Monitoring
  - Job.submit() or Job.waitForCompletion(boolean)
- Job Input
- Job Output

MapReduce User Interface

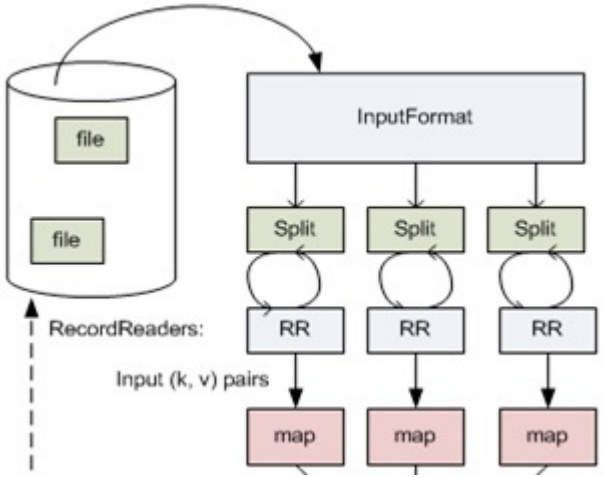
- Other Useful Features
  - Submitting Jobs to Queues
  - Counters: global counters
  - DistributedCache: distribute application-specific, large, read-only files efficiently
  - Profiling
  - Debugging
  - Data Compression
  - Skipping Bad Records
  - ……

MapReduce基本工作过程



主要组件

文件输入格式InputFormat



- 定义了数据文件如何分割和读取
- InputFormat提供了以下一些功能
  - 选择文件或者其它对象，用来作为输入
  - 定义InputSplits，将一个文件分开成为任务
  - 为RecordReader提供一个工厂，用来读取这个文件
- 有一个抽象的类FileInputFormat，所有的输入格式类都从这个类继承这个类的功能以及特性。当启动一个Hadoop任务的时候，一个输入文件所在的目录被输入到FileInputFormat对象中。FileInputFormat从这个目录中读取所有文件。然后FileInputFormat将这些文件分割为一个或者多个InputSplits。
- 通过在JobConf对象上设置JobConf.setInputFormat设置文件输入的格式

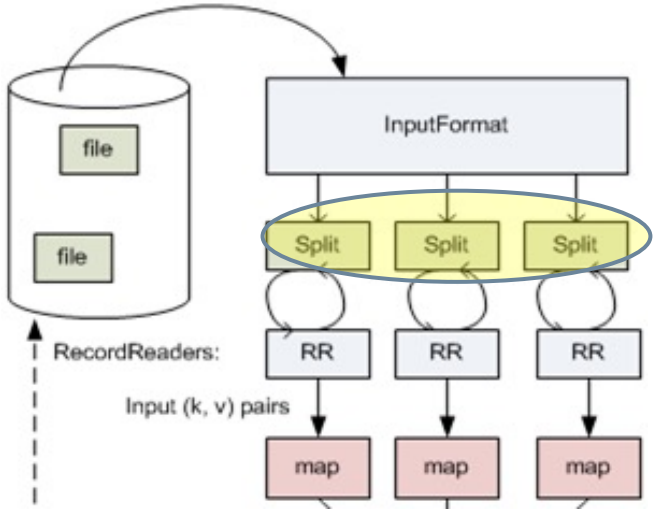
主要组件

- 文件输入格式InputFormat

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

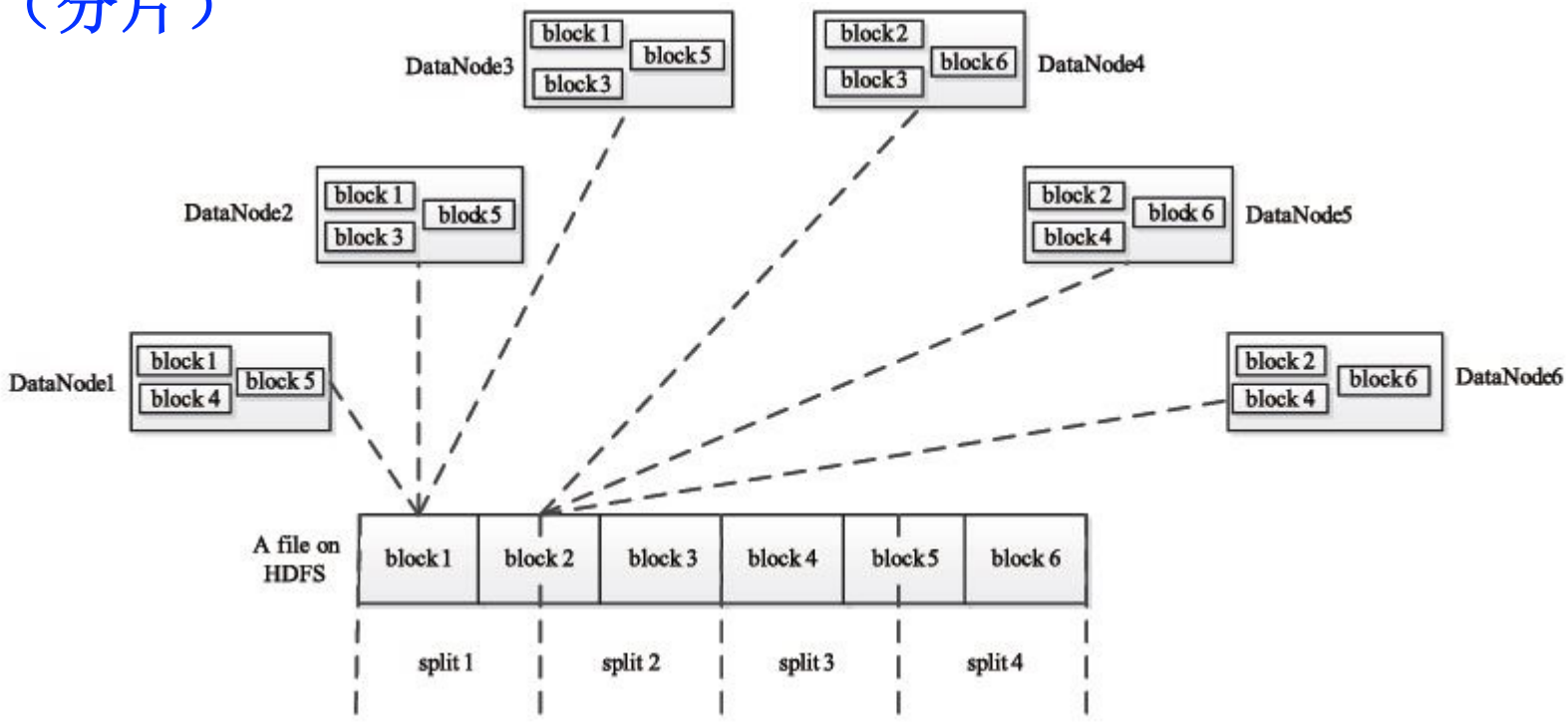
主要组件

- 输入数据分块InputSplits
  - InputSplit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中的mapred.min.split.size参数控制这个大小
  - mapred.tasktracker.map.task.maximum用来控制某一个节点上所有map任务的最大数目



主要组件

关于Split（分片）

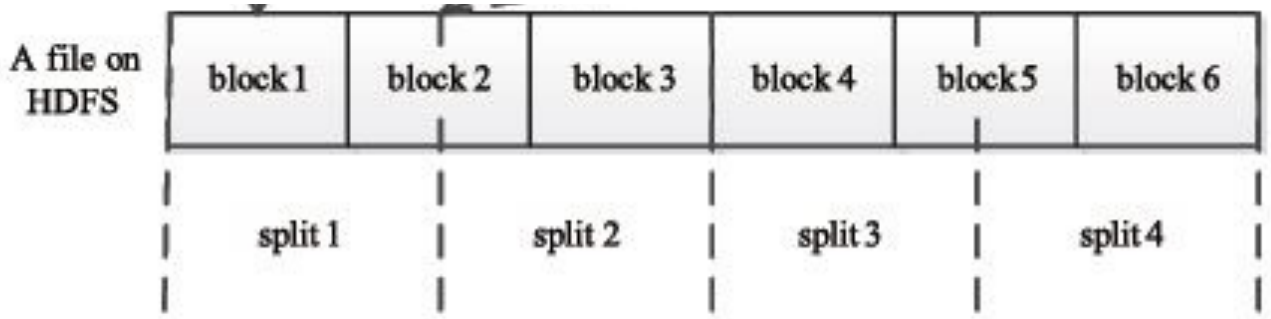


HDFS 以固定大小的block 为基本单位存储数据，而对于MapReduce 而言，其处理单位是split。split是一个逻辑概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。

主要组件

Map任务的数量

- Hadoop为每个split创建一个Map任务，split 的多少决定了Map任务的数目。大多数情况下，理想的分片大小是一个HDFS块

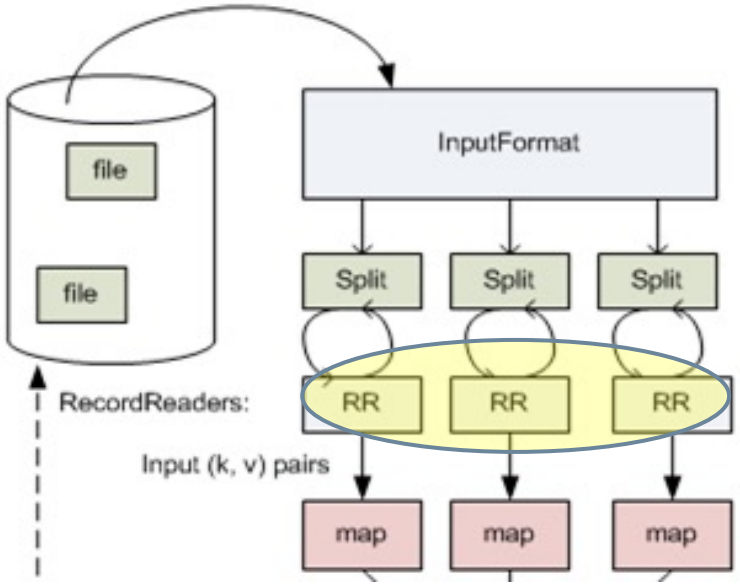


Reduce任务的数量

- 最优的Reduce任务个数取决于集群中可用的reduce任务槽(slot)的数目
- 通常设置比reduce任务槽数目稍微小一些的Reduce任务个数（这样可以预留一些系统资源处理可能发生的错误）

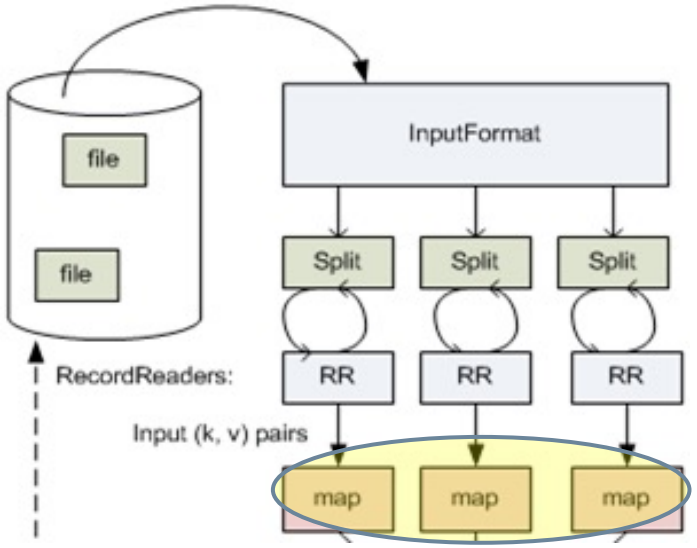
主要组件

- 数据记录读入RecordReader
  - InputSplit定义了一项工作的大小，但是没有定义如何读取数据
  - RecordReader实际上定义了如何从数据上转化为一个(key,value)对的详细方法，并将数据输出到Mapper类中
  - TextInputFormat提供了LineRecordReader



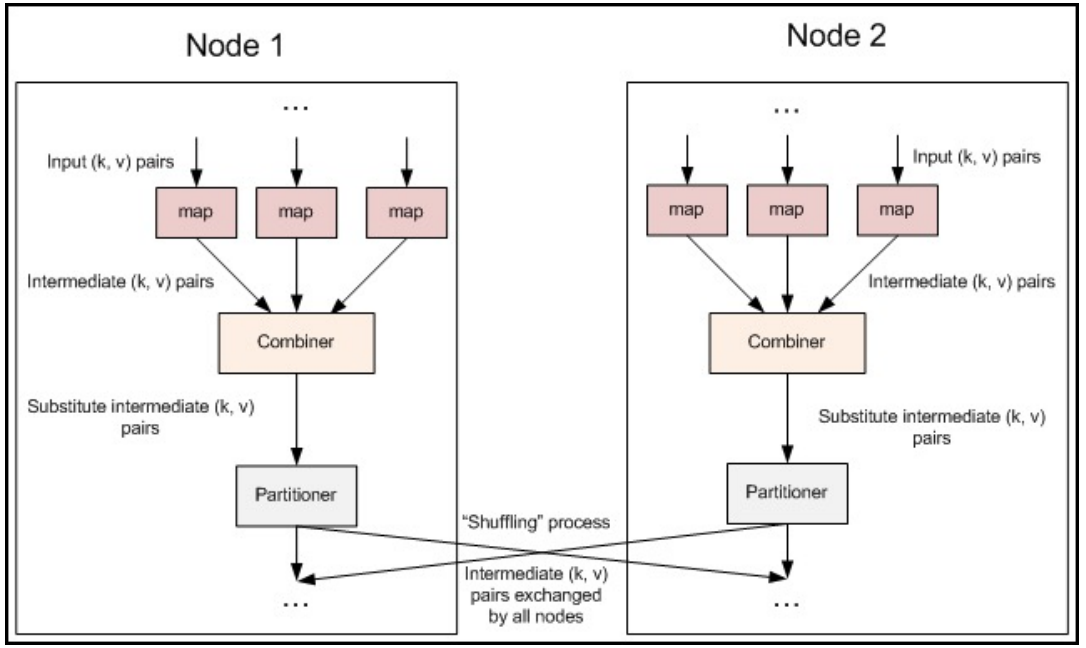
主要组件

- Mapper
  - 每一个Mapper类的实例生成了一个Java进程（在某一个InputSplit上执行）
  - 有两个额外的参数OutputCollector以及Reporter，前者用来收集中间结果，后者用来获得环境参数以及设置当前执行的状态。



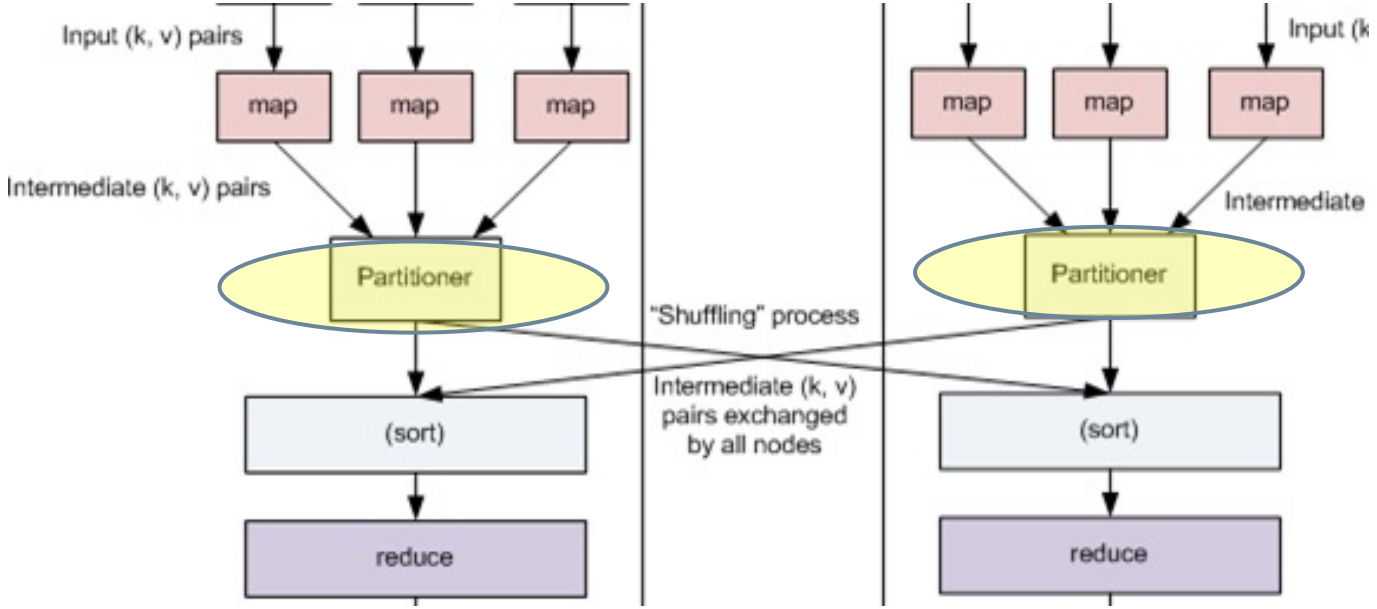
主要组件

- Combiner
  - 合并相同key的键值对，减少partitioner时候的数据通信开销；
  - 是在本地执行的一个Reducer，满足一定的条件才能够执行。
    - conf.setCombinerClass(Reduce.class);



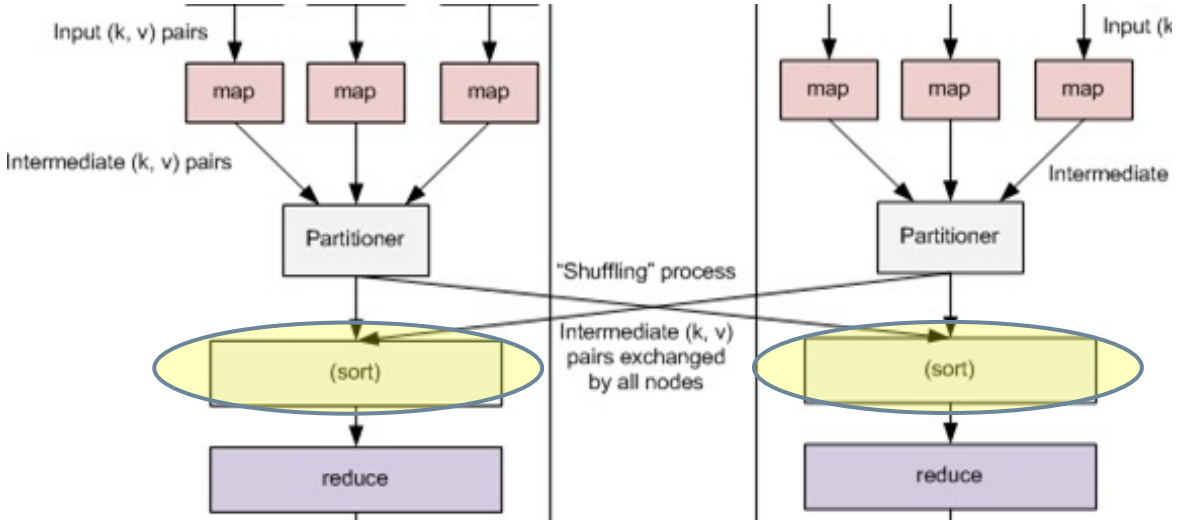
主要组件

- Partitioner & Shuffle
  - 在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，用户可以提供一个Partitioner类，用来决定一个给定的(key,value)对传输的具体位置。



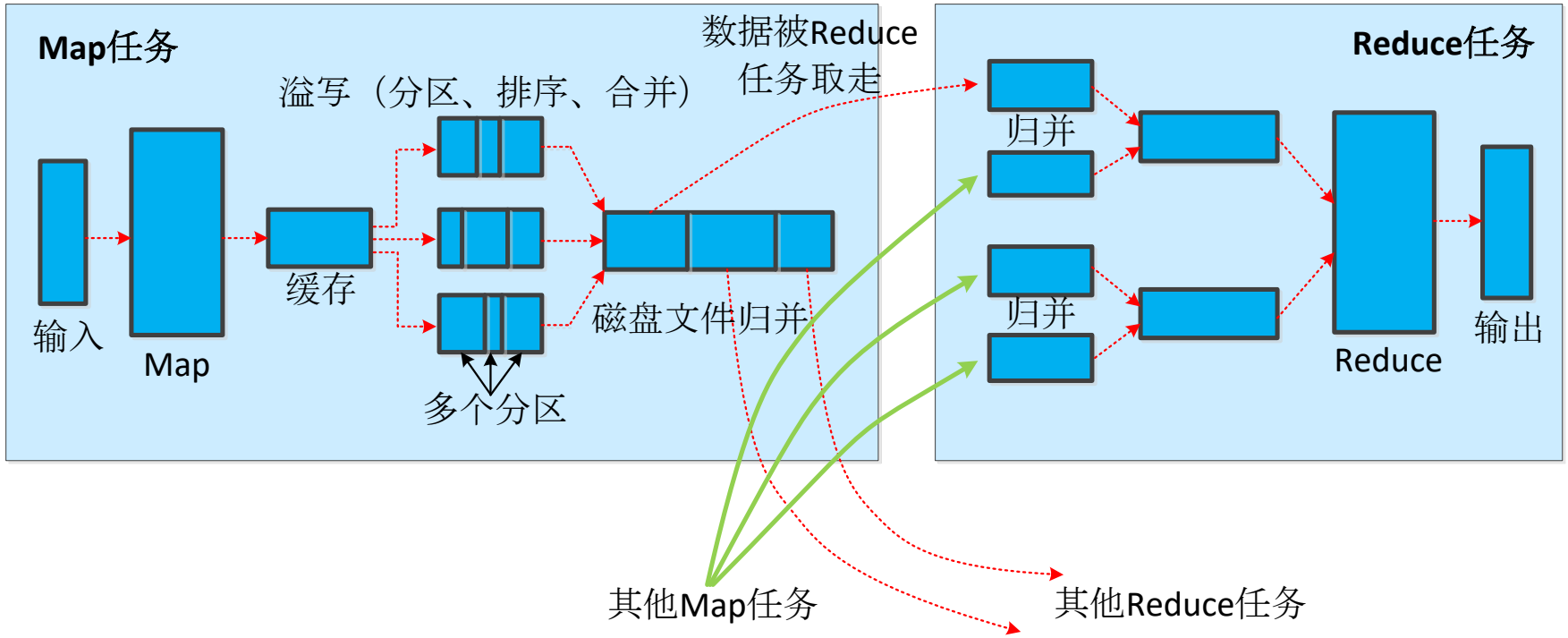
主要组件

- Sort
  - 传输到每一个节点上的所有的Reduce函数接收到的(key,value)都会被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）



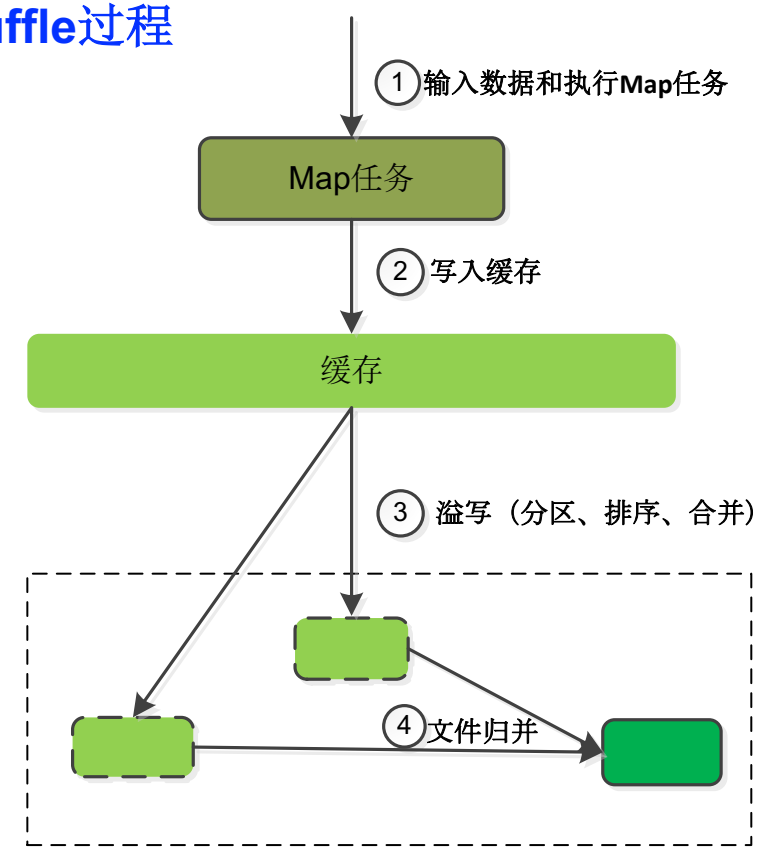
主要组件

1. Shuffle过程简介



主要组件

2. Map端的Shuffle过程



- 每个Map任务分配一个缓存
- MapReduce默认100MB缓存

- 设置溢写比例0.8
- 分区默认采用哈希函数
- 排序是默认的操作
- 排序后可以合并（Combine）
- 合并不能改变最终结果

- 在Map任务全部结束之前进行归并
- 归并得到一个大的文件，放在本地磁盘
- 文件归并时，如果溢写文件数量大于预定值（默认是3）则可以再次启动Combiner，少于3不需要
- JobTracker会一直监测Map任务的执行，并通知Reduce任务来领取数据

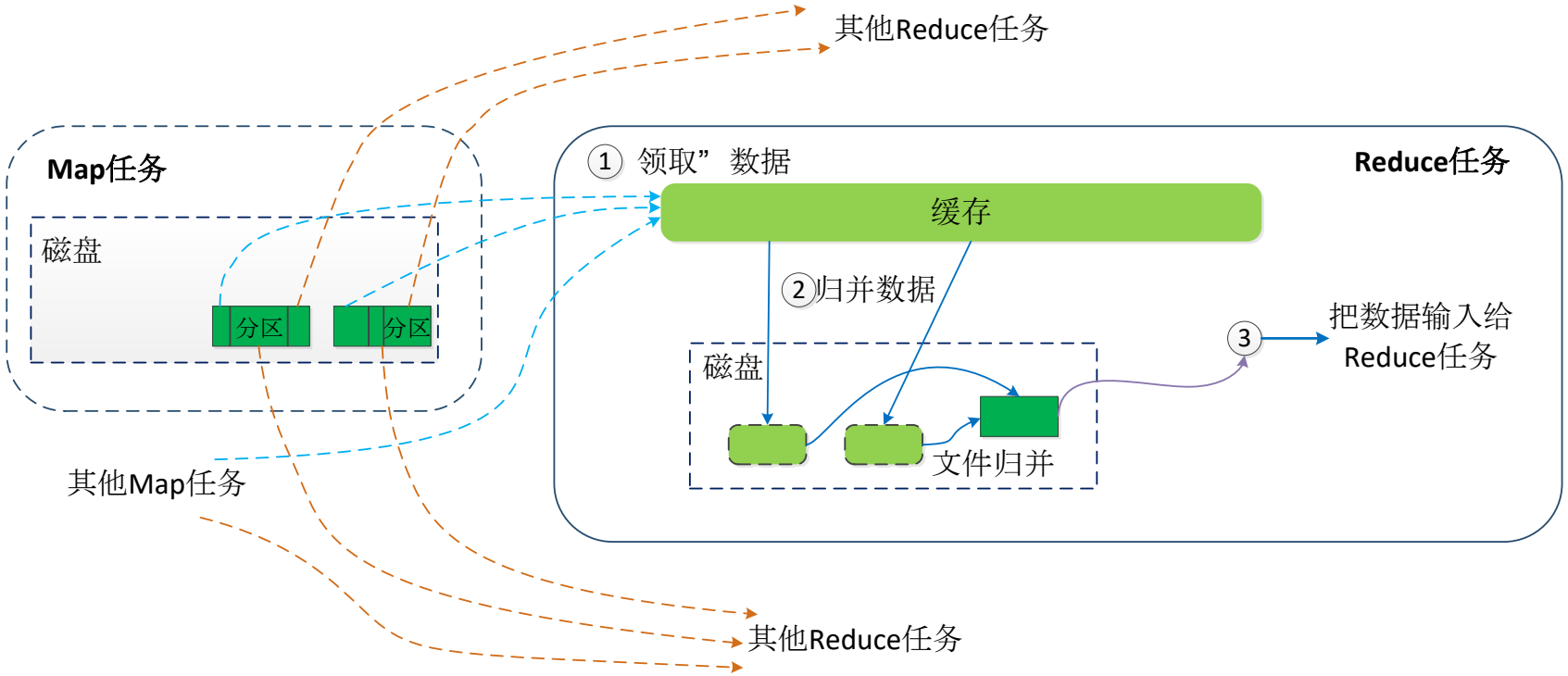
合并（Combine）和归并（Merge）的区别：

两个键值对<“a”,1>和<“a”,1>，如果合并，会得到<“a”,2>，如果归并，会得到<“a”,<1,1>>

主要组件

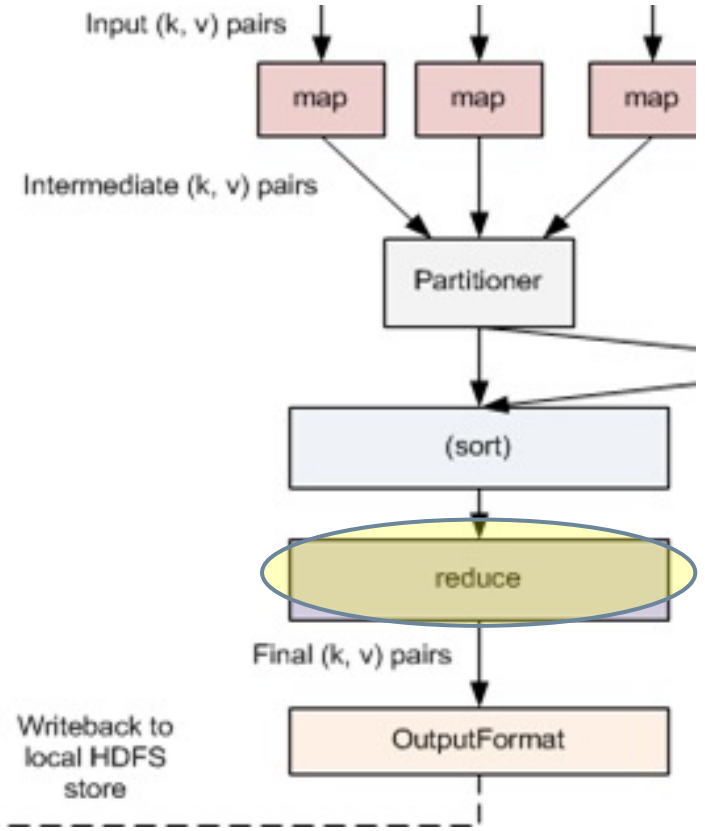
3. Reduce端的Shuffle过程

- Reduce任务向JobTracker询问Map任务是否已经完成，若完成，则领取数据
- Reduce领取数据先放入缓存，来自不同Map机器，先归并，再合并，写入磁盘
- 多个溢写文件归并成一个或多个大文件，文件中的键值对是排序的
- 当数据很少时，不需要溢写到磁盘，直接在缓存中归并，然后输出给Reduce



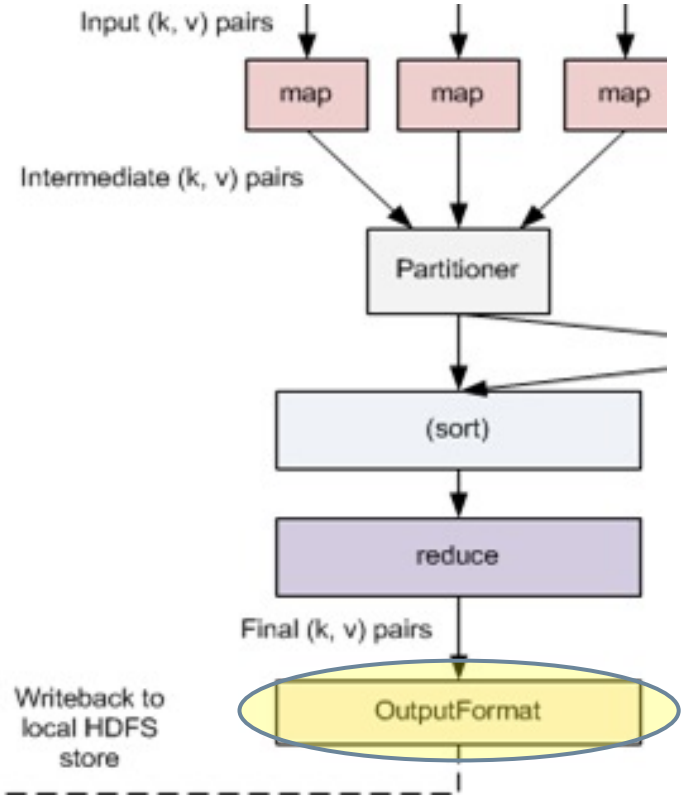
主要组件

- Reducer
  - 执行用户定义的Reduce操作
  - 接收到一个OutputCollector的类作为输出



主要组件

- 文件输出格式OutputFormat
  - 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，其中nnnnn是与每一个reducer相关的一个号（partition id）
  - FileOutputFormat.setOutputPath()
  - JobConf.setOutputFormat()



主要组件

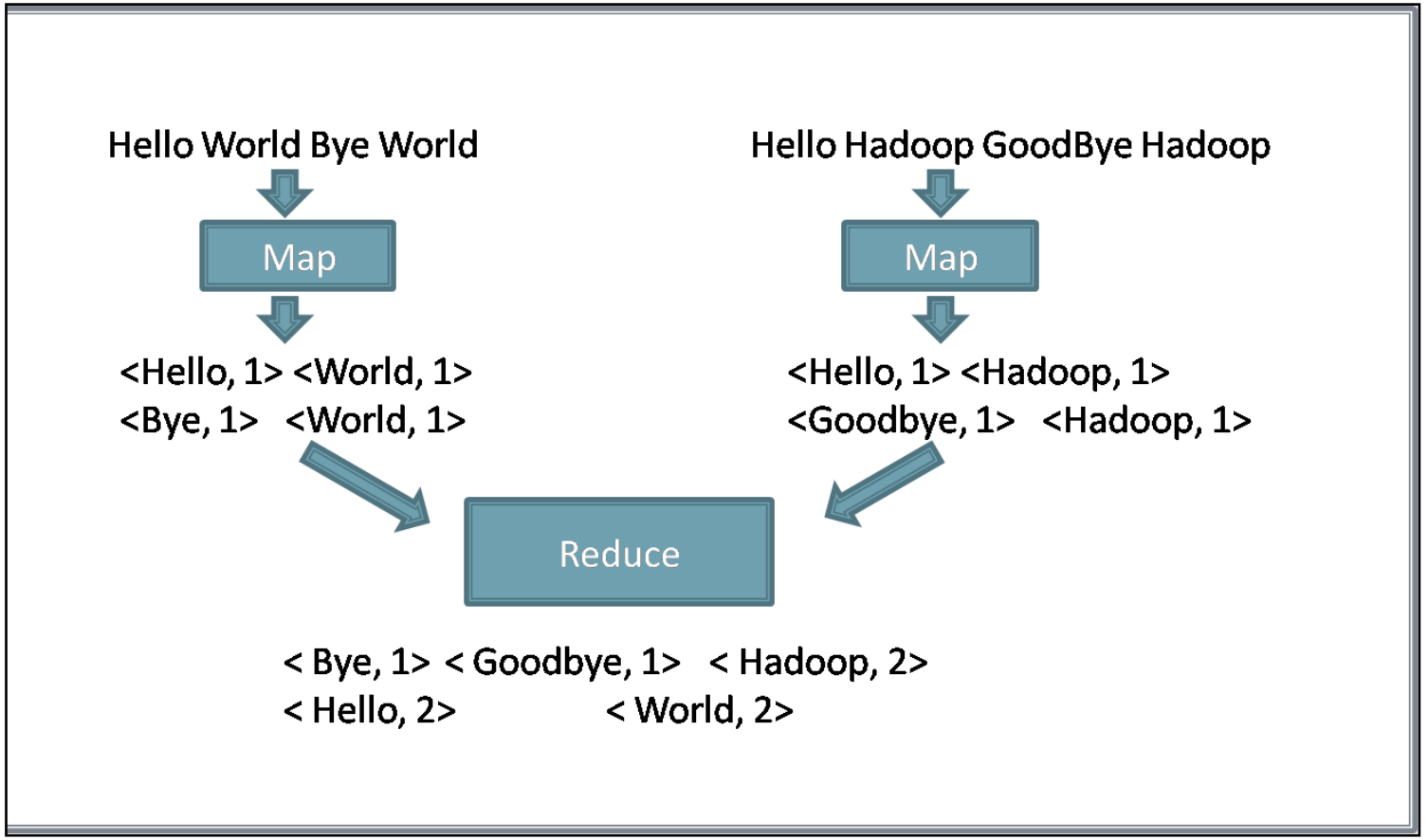
- 文件输出格式OutputFormat

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

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

MapReduce WordCount1.0

- 基本数据处理流程



MapReduce WordCount1.0

- 程序员主要的编码工作如下：
  - 实现Map类
  - 实现Reduce类
  - 实现main函数（运行Job）

MapReduce WordCount1.0

- 实现Map类
  - 这个类实现 org.apache.hadoop.mapreduce.Mapper中的 map 方法，输入参数中的 value 是文本文件中的一行，利用 StringTokenizer 将这个字符串拆成单词，然后通过**context.write**收集<key, value>对。
  - 代码中 LongWritable, IntWritable, Text 均是 Hadoop 中实现的用于封装 Java 数据类型的类，这些类都能够被串行化从而便于在分布式环境中进行数据交换，可以将它们分别视为 long, int, String 的替代。

MapReduce WordCount1.0

- **Map类代码**

```
public static class TokenizerMapper        //定义Map类实现字符串分解
        extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    //实现map()函数
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        //将字符串拆解成单词
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());          //将分解后的一个单词写入word
            context.write(word, one);           //收集<key, value>
        }
    }
}
```

MapReduce WordCount1.0

- 实现Reduce类
  - 这个类实现org.apache.hadoop.mapreduce.Reducer中的 reduce 方法，输入参数中的(key, values) 是由 Map 任务输出的中间结果，values 是一个Iterator，遍历这个 Iterator，就可以得到属于同一个 key的所有value。
  - 此处key 是一个单词，value 是词频。只需要将所有的 value 相加，就可以得到这个单词的总的出现次数。

MapReduce WordCount1.0

- **Reduce类代码**

```
//定义Reduce类规约同一key的value
public static class IntSumReducer  extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    //实现reduce()函数
    public void reduce(Text key, Iterable<IntWritable> values, Context context )
                    throws IOException, InterruptedException {
      int sum = 0;
      //遍历迭代器values，得到同一key的所有value
      for (IntWritable val : values) {  sum += val.get();      }
      result.set(sum);
      //产生输出对<key, value>
      context.write(key, result);
    }
}
```

MapReduce WordCount1.0

- 实现main函数（运行Job）
  - 在 Hadoop 中一次计算任务称之为一个 Job，main函数主要负责新建一个Job对象并为之设定相应的Mapper和Reducer类，以及输入、输出路径等。

MapReduce WordCount1.0

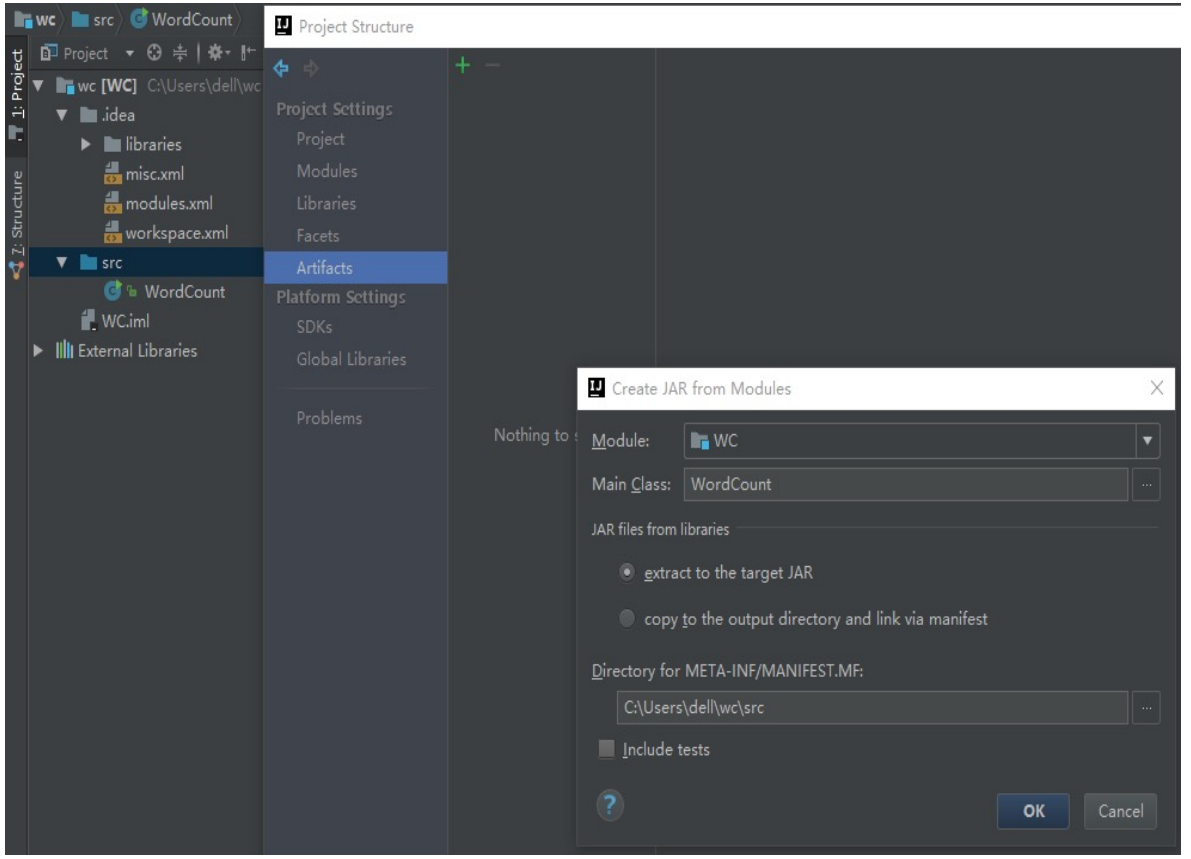
- **main函数代码**

```
public static void main(String[] args) throws Exception{
    //为任务设定配置文件
    Configuration conf = new Configuration();
    //命令行参数
   String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2){
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, “word count”);                    //新建一个用户定义的Job
    job.setJarByClass(WordCount.class);                       //设置执行任务的jar
    job.setMapperClass(TokenizerMapper.class);                //设置Mapper类
    job.setCombinerClass(IntSumReducer.class);                //设置Combine类
    job.setReducerClass(IntSumReducer.class);                 //设置Reducer类
    job.setOutputKeyClass(Text.class);                        //设置job输出的key
    //设置job输出的value
    job.setOutputValueClass(IntWritable.class);
    //设置输入文件的路径
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    //设置输出文件的路径
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    //提交任务并等待任务完成
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

MapReduce WordCount1.0

- 编译源代码
  - 完成编译
  - 导出jar文件
    - 导出wordcount程序的jar包
    - 导出jar文件的时候可以指定一个主类MainClass，作为默认执行的一个类

IntelliJ IDEA CE



MapReduce WordCount1.0

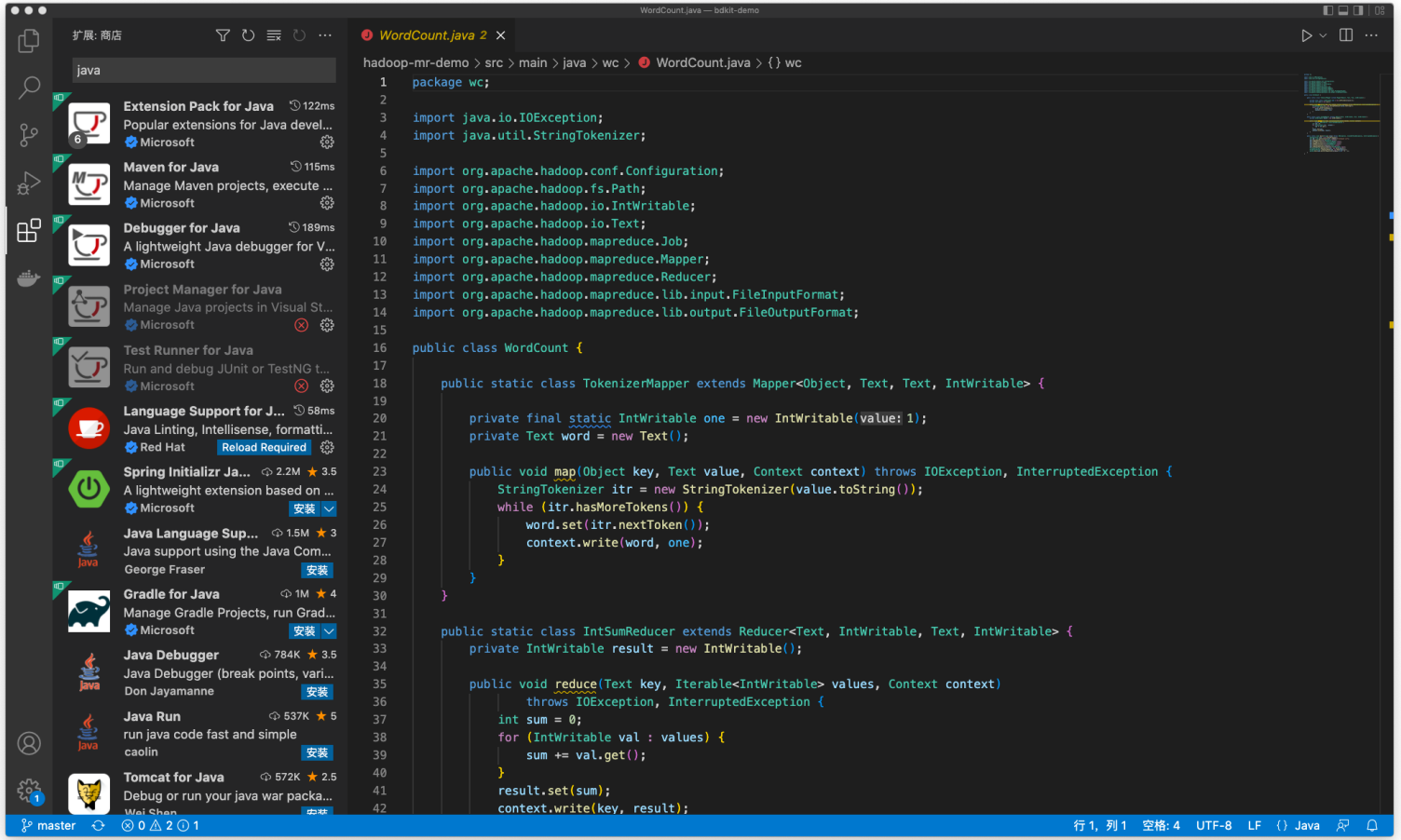
- 本地运行调试
  - 将程序复制到本地Hadoop系统的执行目录，并准备一个小的测试数据，即可通过hadoop的安装包进行运行调试

```
bin/hadoop fs -mkdir input
bin/hadoop fs -put docs/*.html input
bin/hadoop jar example.jar wordcount input output
```

  - 当需要用集群进行海量数据处理时，在本地程序调试正确运行后，可按照前述的远程作业提交步骤，将作业提交到远程hadoop集群上运行。

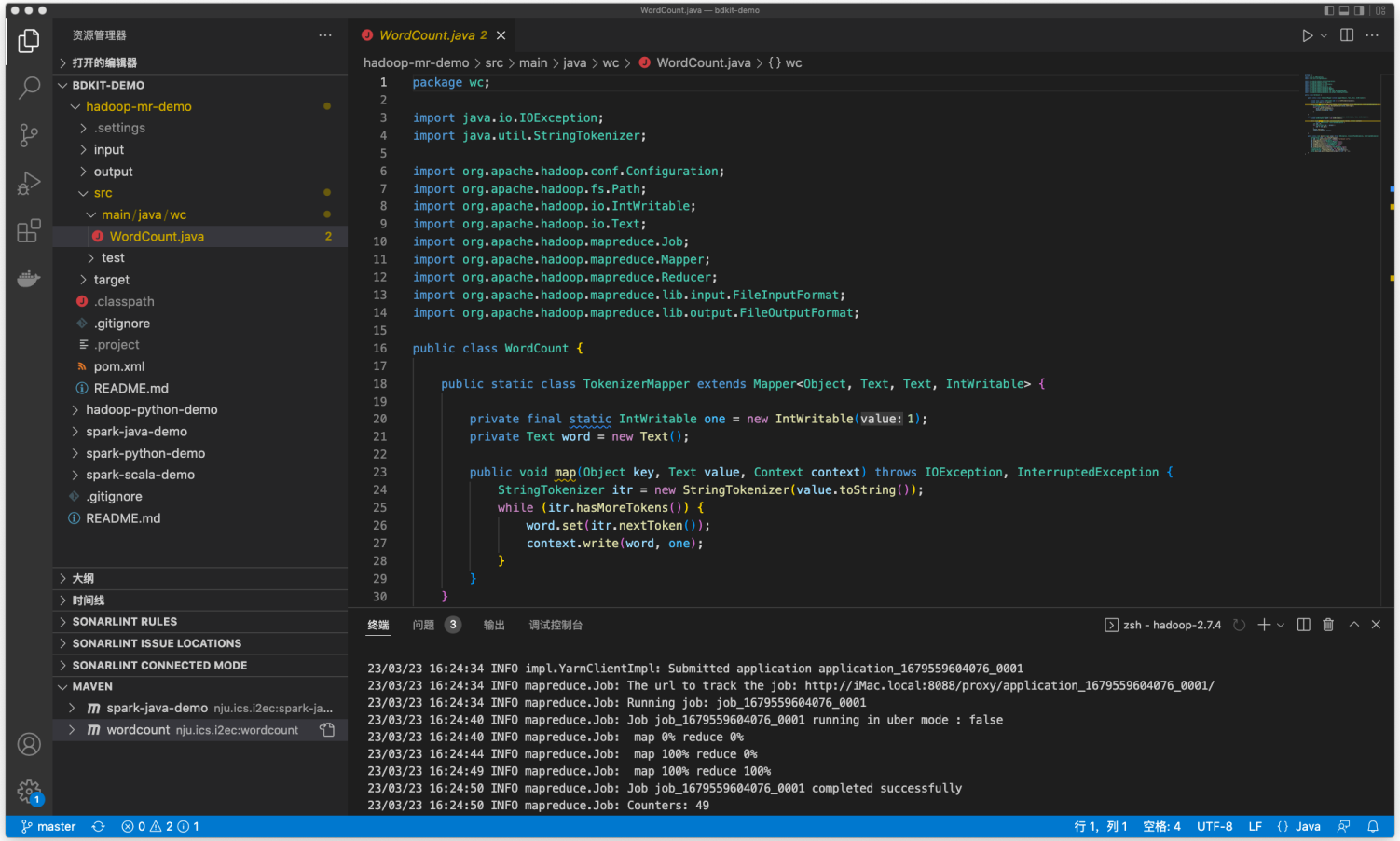
开发环境与工具：VS Code

安装Java开发插件和Maven插件



开发环境与工具：VS Code

编写、编译、打包、运行Java程序



MapReduce WordCount 2.0

- 1. 忽略大小写
- 2. 忽略标点符号

MapReduce WordCount 2.0

- Mapper (new)

```
public void setup(Context context) throws IOException,
    InterruptedException {
  conf = context.getConfiguration();
  caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
  if (conf.getBoolean("wordcount.skip.patterns", true)) {
    URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
    for (URI patternsURI : patternsURIs) {
      Path patternsPath = new Path(patternsURI.getPath());
      String patternsFileName = patternsPath.getName().toString();
      parseSkipFile(patternsFileName);
    }
  }
}
```

MapReduce WordCount 2.0

- Mapper (new)

```
private void parseSkipFile(String fileName) {
  try {
    fis = new BufferedReader(new FileReader(fileName));
    String pattern = null;
    while ((pattern = fis.readLine()) != null) {
      patternsToSkip.add(pattern);
    }
  } catch (IOException ioe) {
    System.err.println("Caught exception while parsing the cached file '"
        + StringUtils.stringifyException(ioe));
  }
}
```

MapReduce WordCount 2.0

- Mapper (updated)

```
@Override
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  String line = (caseSensitive) ?
      value.toString() : value.toString().toLowerCase();
  for (String pattern : patternsToSkip) {
    line = line.replaceAll(pattern, "");
  }
  StringTokenizer itr = new StringTokenizer(line);
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
    Counter counter = context.getCounter(CountersEnum.class.getName(),
        CountersEnum.INPUT_WORDS.toString());
    counter.increment(1);
  }
}
```

MapReduce WordCount 2.0

- Reducer

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

MapReduce WordCount 2.0

- main函数

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
  String[] remainingArgs = optionParser.getRemainingArgs();
  if (!(remainingArgs.length != 2 || remainingArgs.length != 4)) {
    System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
    System.exit(2);
  }
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount2.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  List<String> otherArgs = new ArrayList<String>();
  for (int i=0; i < remainingArgs.length; ++i) {
    if ("-skip".equals(remainingArgs[i])) {
      job.addCacheFile(new Path(remainingArgs[++i]).toUri());
      job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
    } else {
      otherArgs.add(remainingArgs[i]);
    }
  }
  FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));

  System.exit(job.waitForCompletion(true) ? 0 : 1);
```

MapReduce WordCount 2.0

- Input
  - File01: Hello World Bye World
  - File02: Hello Hadoop Goodbye Hadoop
  - File03: Hello World, Bye World!
  - File04: Hello Hadoop, Goodbye to hadoop.
- 运行 $bin/hadoop jar wc2.jar input output2
- 结果:

```
Bye     2
Goodbye 2
Hadoop  2
Hadoop, 1
Hello   4
World   2
World!  1
World,  1
hadoop. 1
to      1
```

MapReduce WordCount 2.0

- 准备patterns文件

```
$ bin/hadoop fs -cat /user/joe/wordcount/patterns.txt
\.
\,
\!
to
```

- 再次运行
  - $ bin/hadoop jar wc.jar -Dwordcount.case.sensitive=true input output3 -skip wordcount/patterns.txt
- 输出：

```
Bye     2
Goodbye 2
Hadoop  3
Hello   4
World   4
hadoop  1
```

- 再次运行
  - $ bin/hadoop jar wc.jar -Dwordcount.case.sensitive=false input output4 -skip wordcount/patterns.txt
- 输出：

```
bye     2
goodbye 2
hadoop  4
hello   4
world   4
```

MapReduce WordCount 2.0

- Demonstrates how applications can access configuration parameters in the setup method of the Mapper (and Reducer) implementations.
- Demonstrates how the DistributedCache can be used to distribute read-only data needed by the jobs. Here it allows the user to specify word-patterns to skip while counting.
- Demonstrates the utility of the GenericOptionsParser to handle generic Hadoop command-line options.
- Demonstrates how applications can use Counters and how they can set application-specific status information passed to the map (and reduce) method.

MapReduce WordCount in Python

- Hadoop Streaming
  - Hadoop Streaming是Hadoop的一个工具， 它帮助用户创建和运行一类特殊的map/reduce作业， 这些特殊的map/reduce作业是由一些可执行文件或脚本文件充当mapper或者reducer。
  - Hadoop Streaming 参考文档
    - https://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html

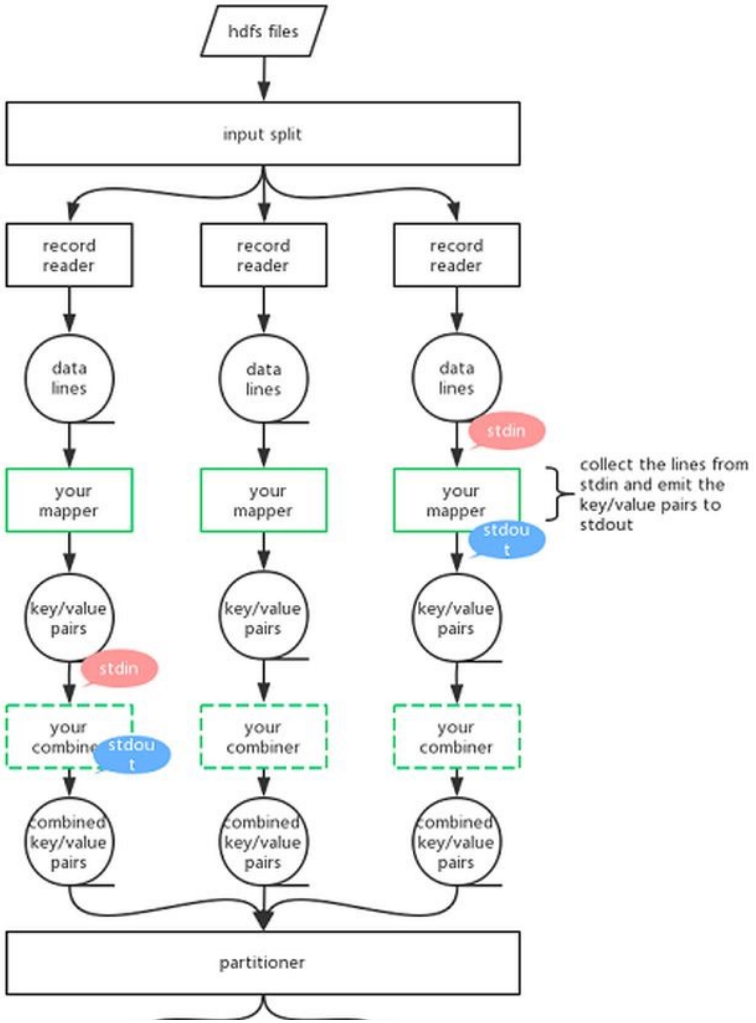
Hadoop Streaming

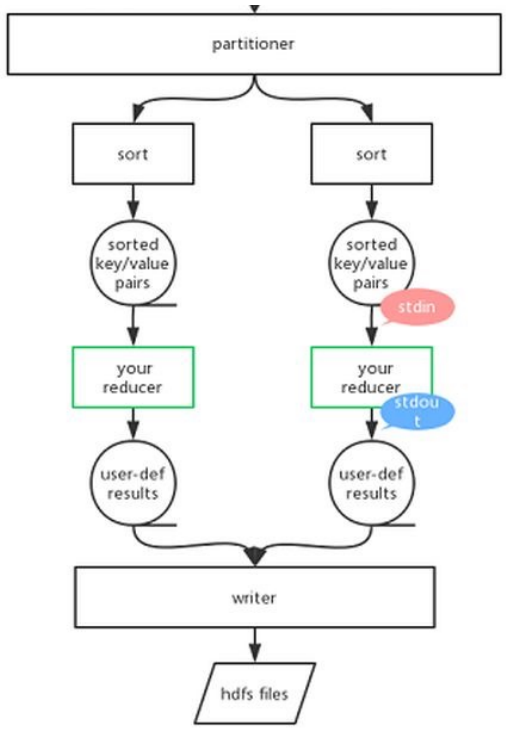
- 使用方式
  - $HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/.../hadoop-streaming.jar [genericOptions] [streamingOptions]
  - $HADOOP_HOME/bin/ mapred streaming [genericOptions] [streamingOptions]
- 常用参数
  - -file
  - -mapper
  - -reducer
  - -input
  - -output

Hadoop Streaming

- 使用 Hadoop Streaming 来让数据在map 和 reduce之间传递数据，使用sys.stdin获得输入数据，sys.stdout 作为标准输出。
- mapper和reducer都是可执行文件，它们从标准输入读入数据（一行一行读），并把计算结果发给标准输出。Streaming工具会创建一个Map/Reduce作业， 并把它发送给合适的集群，同时监视这个作业的整个执行过程。
- 用户可以设定stream.non.zero.exit.is.failure为true 或false 来表明streaming task的返回值非零时是 Failure 还是Success。默认情况，streaming task返回非零时表示失败。

Hadoop Streaming





mapper.py


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

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

reducer.py


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

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

运行


```
#!/bin/bash
bin/hdfs dfs -rm -r python/output
bin/hadoop jar ./share/hadoop/tools/lib/hadoop-streaming-2.7.4.jar \
-D stream.non.zero.exit.is.failure=false \
-D mapred.job.name=python_word_count \
-file ./mapper.py     -mapper  "python mapper.py"  \
-file ./reducer.py    -reducer "python reducer.py"  \
-input python/input/wordcount_data.txt -output python/output
```

MapReduce矩阵乘法

- 并行化矩阵乘法
- 矩阵M_{ab}和矩阵N_{bc}的乘积P=M·N
  - $P_{ik} = (M \cdot N)_{ik} = \sum m_{ij} n_{jk}$
- Map阶段：进行数据准备
  - 定义(key, value)对
  - 矩阵M：<(i, k), (M, j, m_{ij})>
  - 矩阵N：<(i, k), (N, j, n_{jk})>
  - 其中i=1, 2, …a; j=1,2,…b; k=1, 2, …c

MapReduce矩阵乘法

- Reduce阶段： $m_{ij}*n_{jk}$
  - 对于每个键(i, k)相关联的值(M, j, m_{ij})及(N, j, n_{jk})，根据相同j值将m_{ij}和n_{jk}分别存入不同数组中，然后将两者的第j个元素抽取出来分别相乘，最后相加，即可得到p_{ik}的值。

- 代码示例

MapReduce矩阵乘法

- 举例：
- M[1,2] N$\begin{bmatrix} 2 & 1 & 3 \\ 0 & 2 & 4 \end{bmatrix}$
- i=1 j=1,2 k=1,2,3
- Map输出
  - <(1,1),(M,1,1)><(1,1),(N,1,2)><(1,1),(M,2,2)><(1,1),(N,2,0)>
  - <(1,2),(M,1,1)><(1,2),(N,1,1)><(1,2),(M,2,2)><(1,2),(N,2,2)>
  - <(1,3),(M,1,1)><(1,3),(N,1,3)><(1,3),(M,2,2)><(1,3),(N,2,4)>
- Reduce输出
  - <(1,1),2> <(1,2),5><(1,3),11>

关系代数运算

- 选择、投影、并、交、差以及自然连接操作

表1 关系R

ID	NAME	AGE	GRADE
1	张小雅	20	91
2	刘伟	19	87
3	李婷	21	82
4	孙强	20	95

表2 关系S

ID	GENDER	HEIGHT
1	女	165
2	男	178
3	女	170
4	男	175

关系代数运算

- 选择操作
  - Map阶段：对于每个输入的记录判断是否满足条件，将满足条件的记录输出为(Rc, null)。
  - Reduce阶段：无需做额外工作
  - *setNumReduceTasks(0)*
- 投影操作
  - Map 阶段：将每条记录在该属性上的值作为键输出即可。
  - Reduce阶段：将Map端输入的键输出即可。

关系代数运算

- 交运算
  - 同一个模式的关系R和关系T求交集
  - Map阶段：每条记录r输出为(r, 1)
  - Reduce阶段：如果计数为2则输出该记录
  - 注意事项：R和T的相同记录要发送到同一个Reduce节点
    - 重写hashCode()方法使得具有相同域值的记录具有相同的哈希值

关系代数运算

- 差运算
  - 同一个模式的关系R和关系T求差（R-T）
  - Map阶段：R中的记录r输出键值对(r, R)，T中的记录r输出键值对(r, T)
  - Reduce阶段：如果只有R而没有T，则将该记录输出
  - 注意事项：R和T的相同记录要发送到同一个Reduce节点
    - 重写hashCode()方法使得具有相同域值的记录具有相同的哈希值

关系代数运算

- 自然连接
  - 比如在属性ID上做关系R和关系S的自然连接
  - Map阶段：将ID的值作为key，将其余属性的值以及R的名称（S中的记录为S的名称）作为value
  - Reduce阶段：将同一key中所有的值根据它们的来源（R或S）分为两组做笛卡尔乘积然后输出。

- 代码示例