

深入理解大数据-大数据处理与编程实践

Ch.3. Google 和Hadoop MapReduce基本构架

南京大学计算机科学与技术系

主讲人：黄宜华

鸣谢：本课程得到Google（北京）与Intel公司
中国大学合作部精品课程计划资助

Ch.3. Google /Hadoop MapReduce基本构架

1. MapReduce的基本模型和处理思想
2. Google MapReduce的基本工作原理
3. Hadoop MapReduce的基本工作原理
4. Hadoop 分布式文件系统HDFS
5. Hadoop HDFS的编程

1. MapReduce的基本模型和处理思想

三个层面上的基本构思

如何对付大数据处理：分而治之

对相互间不具有计算依赖关系的大数据，实现并行最自然的办法就是采取分而治之的策略

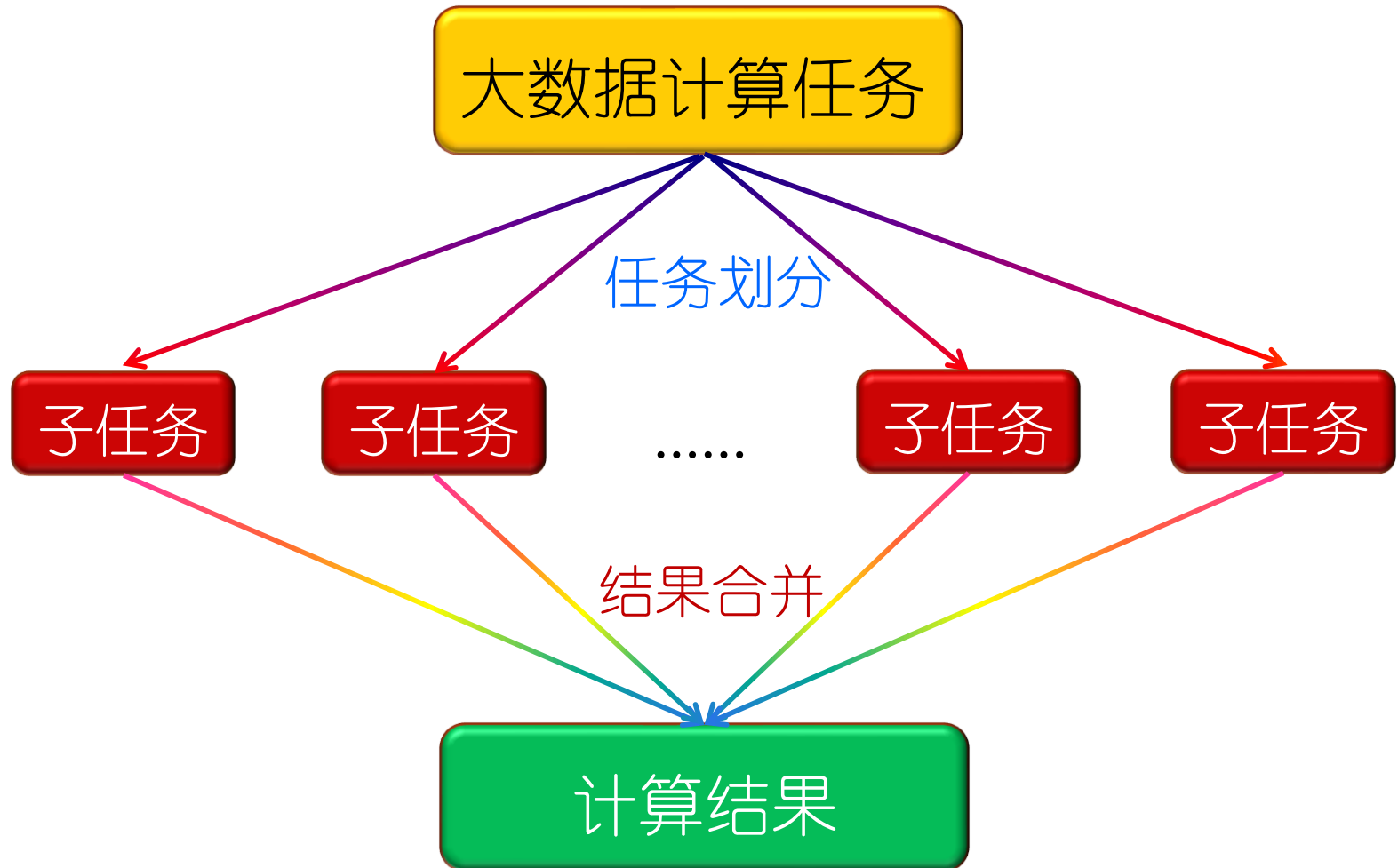
上升到抽象模型：Mapper与Reducer

MPI等并行计算方法缺少高层并行编程模型，为了克服这一缺陷，MapReduce借鉴了Lisp函数式语言中的思想，用Map和Reduce两个函数提供了高层的并行编程抽象模型

上升到构架：统一构架，为程序员隐藏系统层细节

MPI等并行计算方法缺少统一的计算框架支持，程序员需要考虑数据存储、划分、分发、结果收集、错误恢复等诸多细节；为此，MapReduce设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面的处理细节

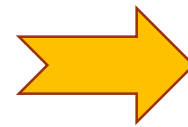
大数据分而治之



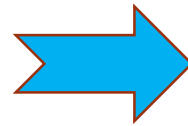
建立Map和Reduce抽象模型

典型的流式大数据问题的特征

- 大量数据记录/元素进行重复处理
- 对每个数据记录/元素作感兴趣的
处理、获取感兴趣的中间结果信息
- 排序和整理中间结果以利后续处理
- 收集整理中间结果
- 产生最终结果输出



Map



Reduce

关键思想：为大数据处理过程中的两个主要处理阶段
提炼为一种抽象的操作机制

建立Map和Reduce抽象模型

借鉴函数式程序设计语言Lisp中的思想，定义了Map和Reduce两个抽象的操作函数：

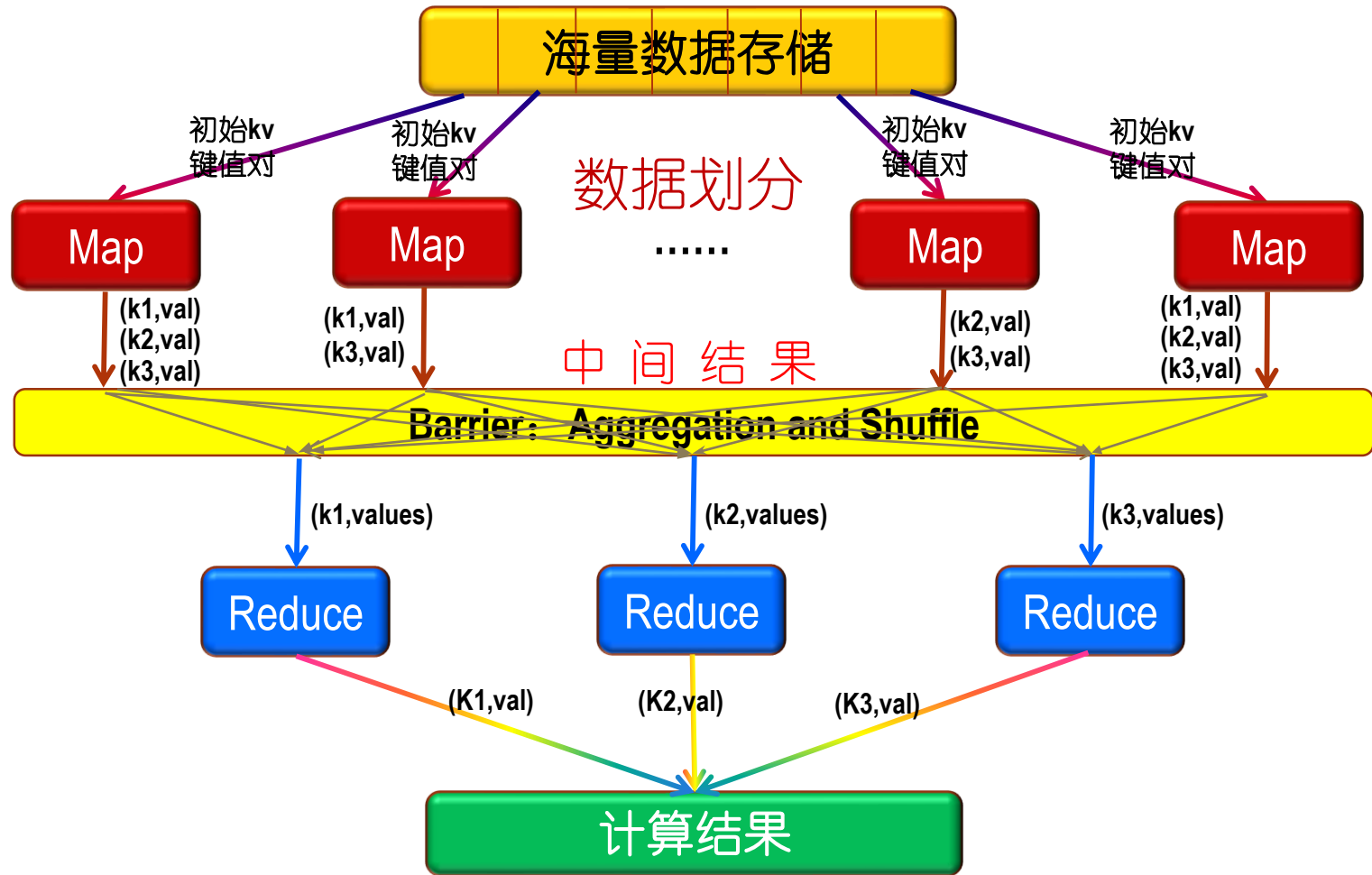
- $\text{map}: (k1; v1) \rightarrow [(k2; v2)]$
- $\text{reduce}: (k2; [v2]) \rightarrow [(k3; v3)]$

特点：

- 描述了对一组数据处理的两个阶段的抽象操作
- 仅仅描述了需要做什么，不需要关注怎么做

MapReduce的基本模型与处理思想

上升到构架--自动并行化并隐藏底层细节



上升到构架--自动并行化并隐藏低层细节

海量数据存储

初始kv键值对

Map

Combiner

Partitioner

Barrier

Reduce

计算结果

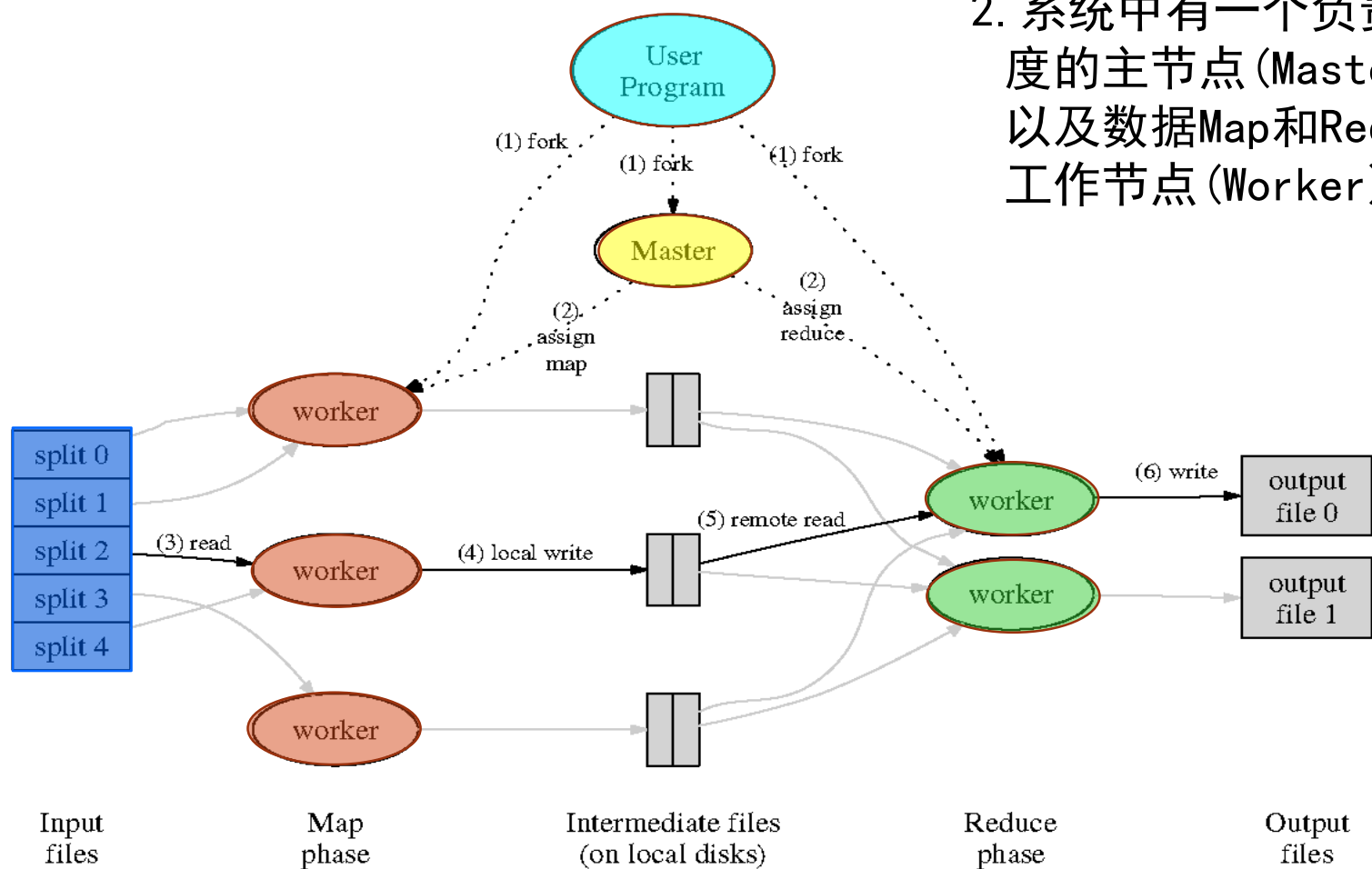
数据划分

中间结果

2. Google MapReduce的基本工作原理

Google MapReduce并行处理的基本过程

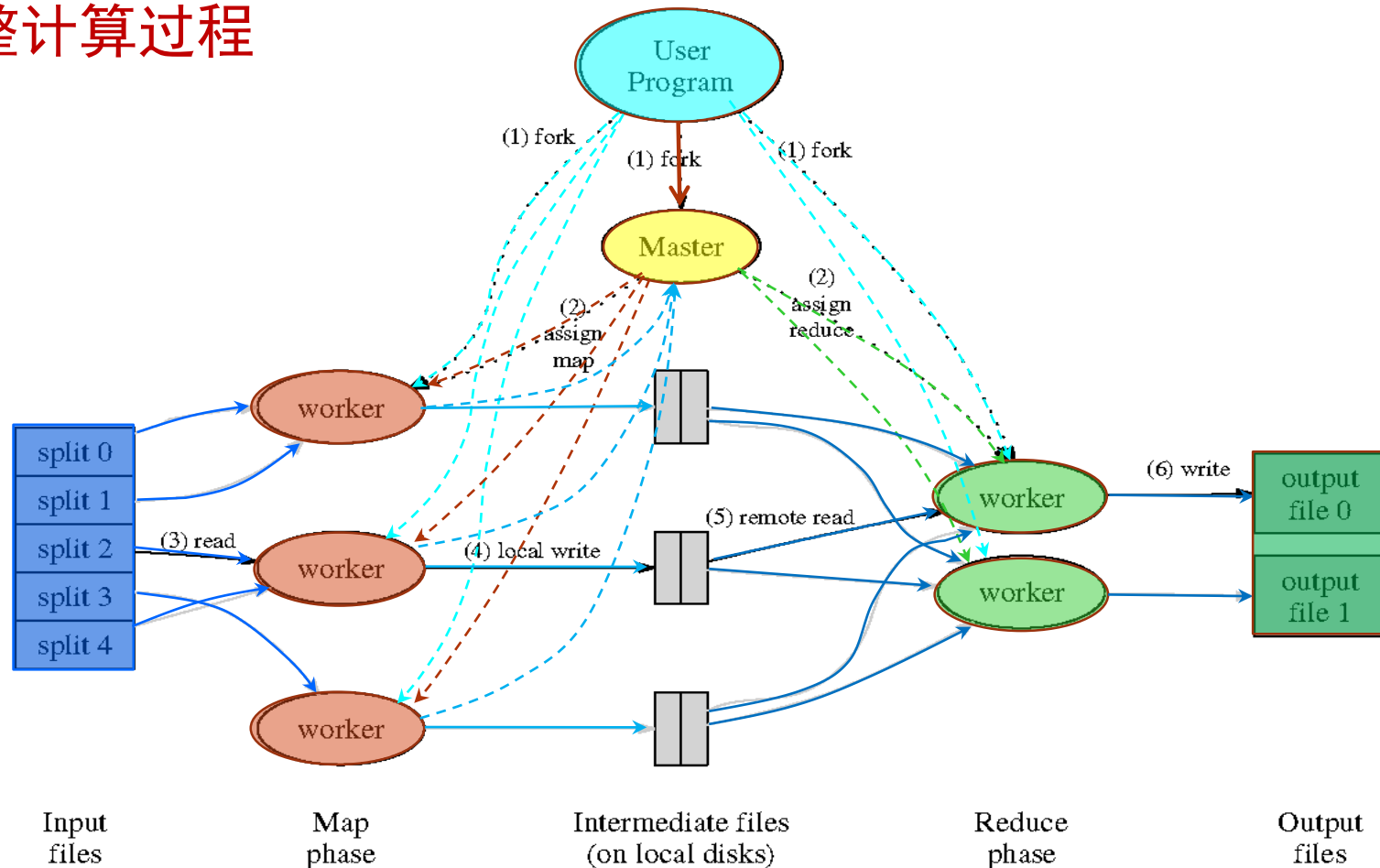
2. 系统中有一个负责调度的主节点(Master), 以及数据Map和Reduce工作节点(Worker)



Google MapReduce的基本工作原理

Google MapReduce并行处理的基本过程

完整计算过程



失效处理

- 主节点失效

主节点中会周期性地设置检查点(checkpoint)，检查整个计算作业的执行情况，一旦某个任务失效，可以从最近有效的检查点开始重新执行，避免从头开始计算的时间浪费。

- 工作节点失效

工作节点失效是很普遍发生的，主节点会周期性地给工作节点发送心跳检测，如果工作节点没有回应，这认为该工作节点失效，主节点将终止该工作节点的任务并把失效的任务重新调度到其它工作节点上重新执行

带宽优化

• 问题

大量的键值对数据在传送给Reduce节点时会引起较大的通信带宽开销。

• 解决方案

每个Map节点处理完成的中间键值对将由combiner做一个合并压缩，即把那些键名相同的键值对归并为一个键名下的一组数值。



计算优化

问题

Reduce节点必须要等到所有Map节点计算结束才能开始执行，因此，如果有一个计算量大、或者由于某个问题导致很慢结束的Map节点，则会成为严重的“拖后腿者”。

解决方案

把一个Map计算任务让多个Map节点同时做，取最快完成者的计算结果。

根据Google的测试，使用了这个冗余Map节点计算方法以后，计算任务性能提高40%多！

用数据分区解决数据相关性问题

• 问题

一个Reduce节点上的计算数据可能会来自多个Map节点，因此，为了在进入Reduce节点计算之前，需要把属于一个Reduce节点的数据归并到一起。

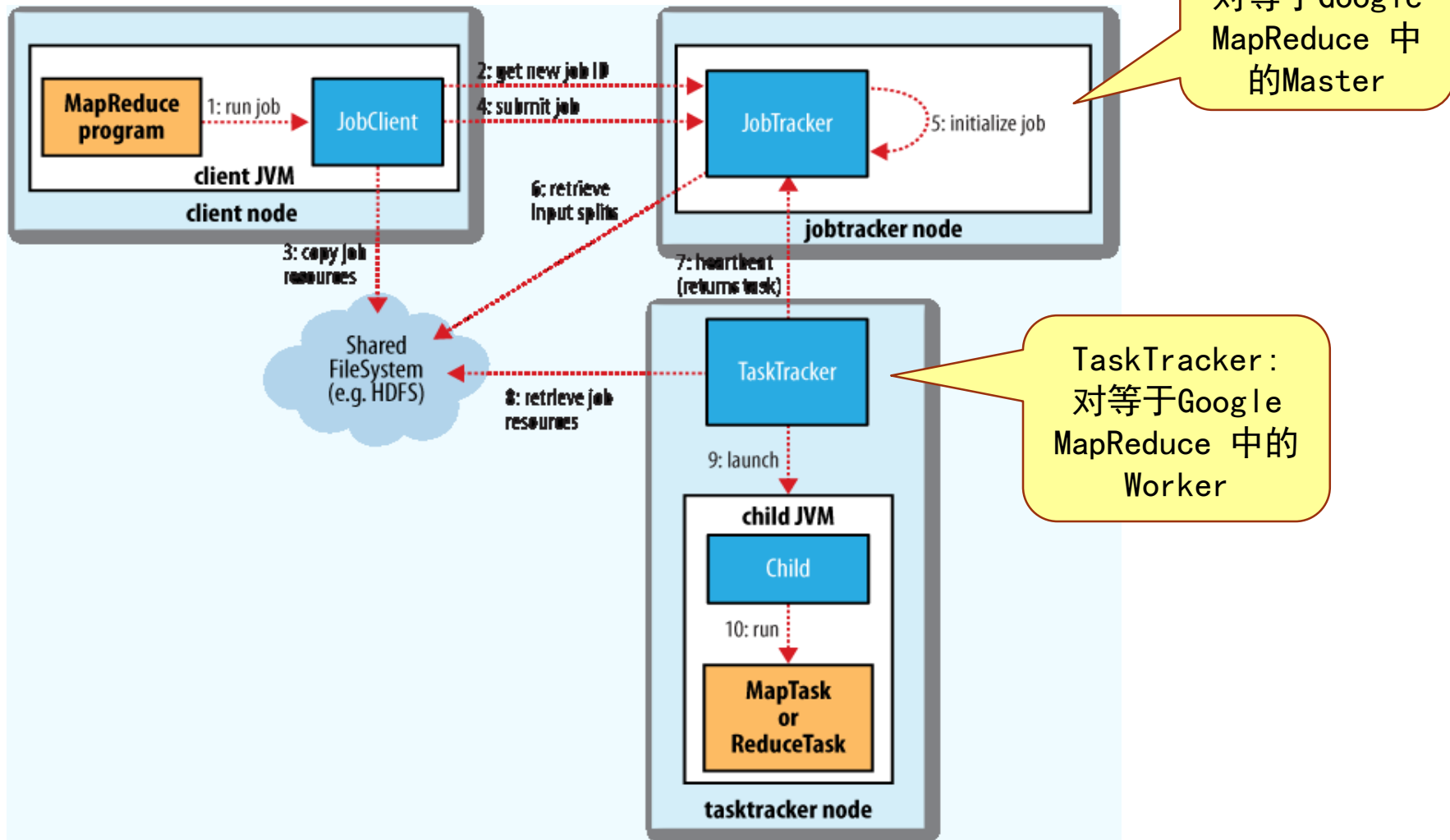
• 解决方案

在Map阶段进行了Combining以后，可以根据一定的策略对Map输出的中间结果进行分区(partitioning)，这样即可解决以上数据相关性问题避免Reduce计算过程中的数据通信。

例如：有一个巨大的数组，其最终结果需要排序，每个Map节点数据处理好后，为了避免在每个Reduce节点本地排序完成后还需要进行全局排序，我们可以使用一个分区策略如： $(d\%R)$ ， d 为数据大小， R 为Reduce节点的个数，则可根据数据的大小将其划分到指定数据范围的Reduce节点上，每个Reduce将本地数据拍好序后即为最终结果

3. Hadoop MapReduce的基本工作原理

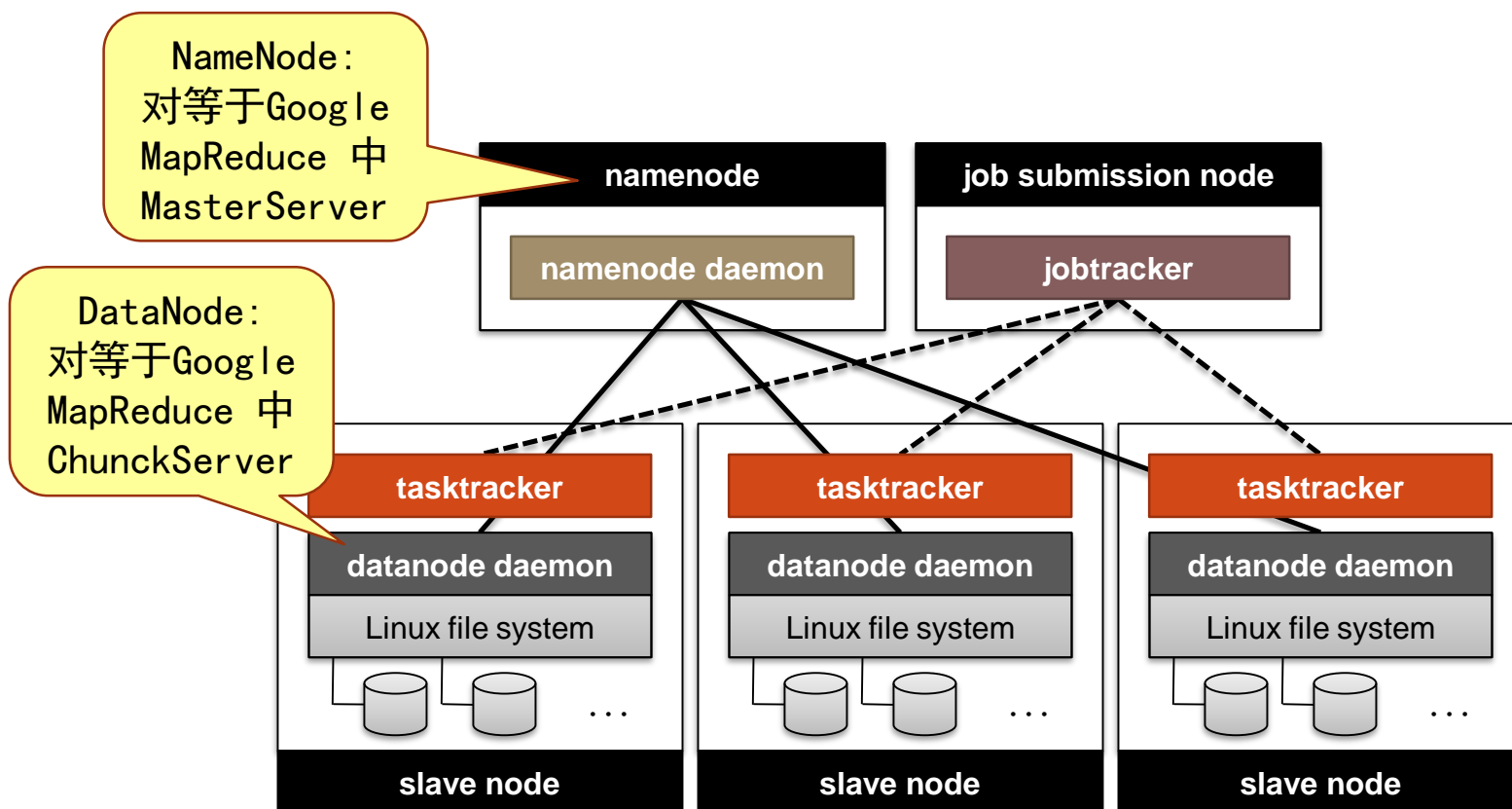
Hadoop MapReduce基本构架与工作过程



Hadoop MapReduce的基本工作原理

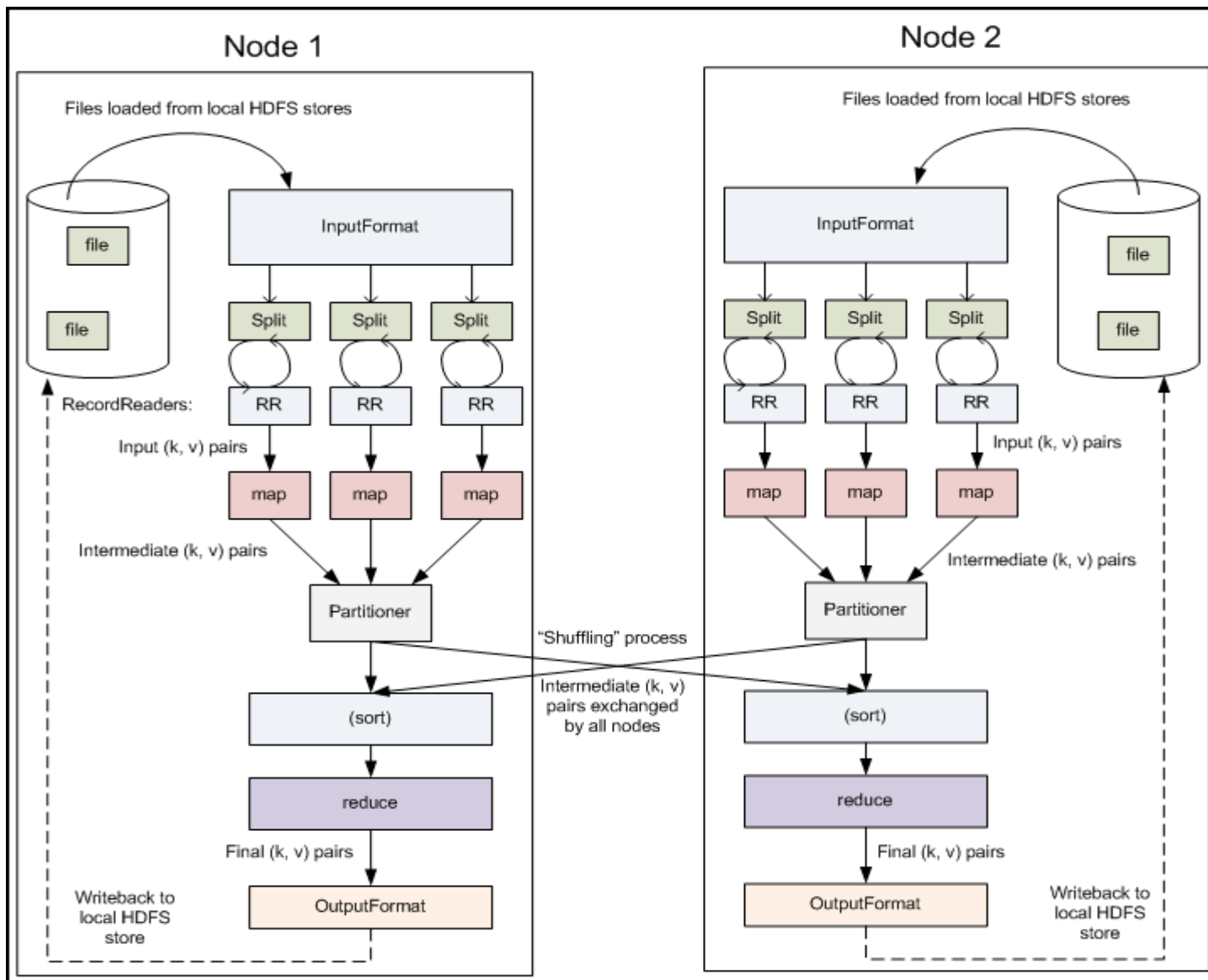
Hadoop MapReduce基本构架与工作过程

数据存储与计算节点构架



Hadoop MapReduce的基本工作原理

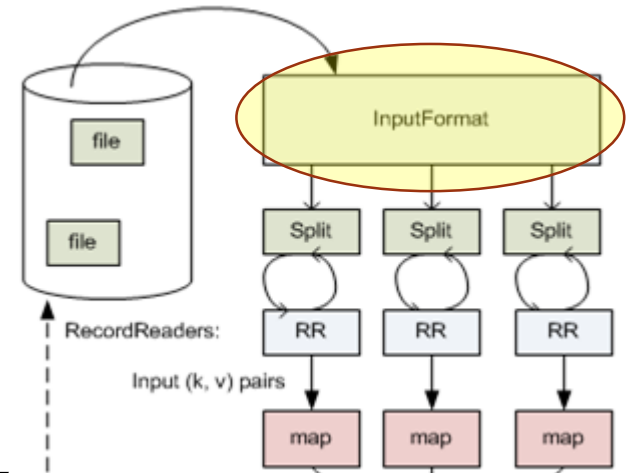
Hadoop MapReduce基本工作过程



Hadoop MapReduce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, 将一个文件分为不同任务
 - 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其功能以及特性。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将这些文件分割为多个InputSplits。
- 通过在JobConf对象上设置JobConf.setInputFormat设置文件输入的格式



Hadoop MapReduce主要组件

文件输入格式InputFormat

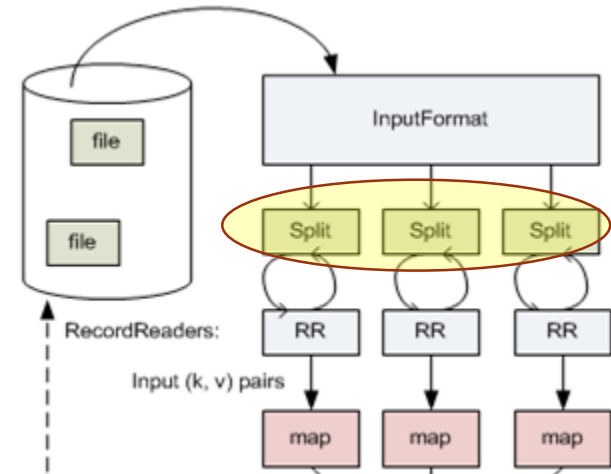
InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueTextInputFormat	Parses lines into key-val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific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

Hadoop MapReduce的基本工作原理

Hadoop MapReduce主要组件

输入数据分块InputSplits

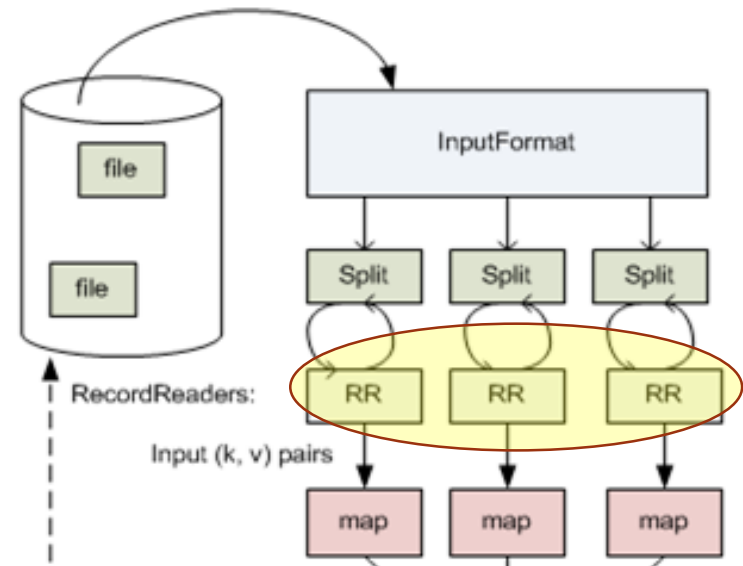
- InputSplit定义了输入到单个Map任务的输入数据
- 一个MapReduce程序被统称为一个Job，可能有上百个任务构成
- InputSplit将文件分为64MB的大小
 - 配置文件hadoop-site.xml中的mapred.min.split.size参数控制这个大小
- mapred.tasktracker.map.taks.maximum用来控制某一个节点上所有map任务的最大数目



Hadoop MapReduce主要组件

数据记录读入RecordReader

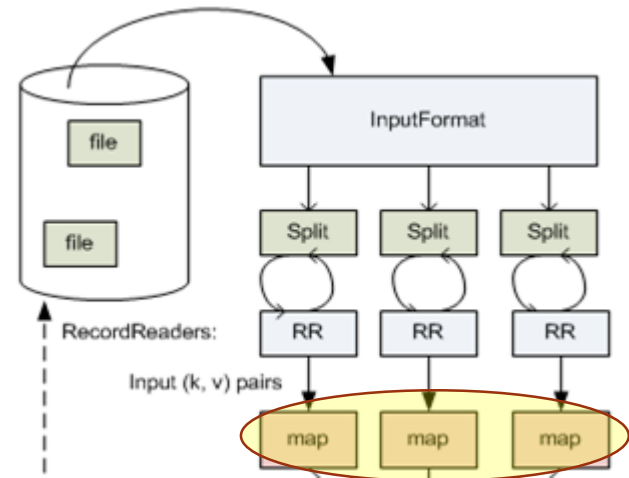
- InputSplit定义了一个数据分块，但是没有定义如何读取数据记录
- RecordReader实际上定义了如何将数据记录转化为一个(key,value)对的详细方法，并将数据记录传给Mapper类
- TextInputFormat提供了LineRecordReader，读入一个文本行数据记录



Hadoop MapReduce主要组件

Mapper

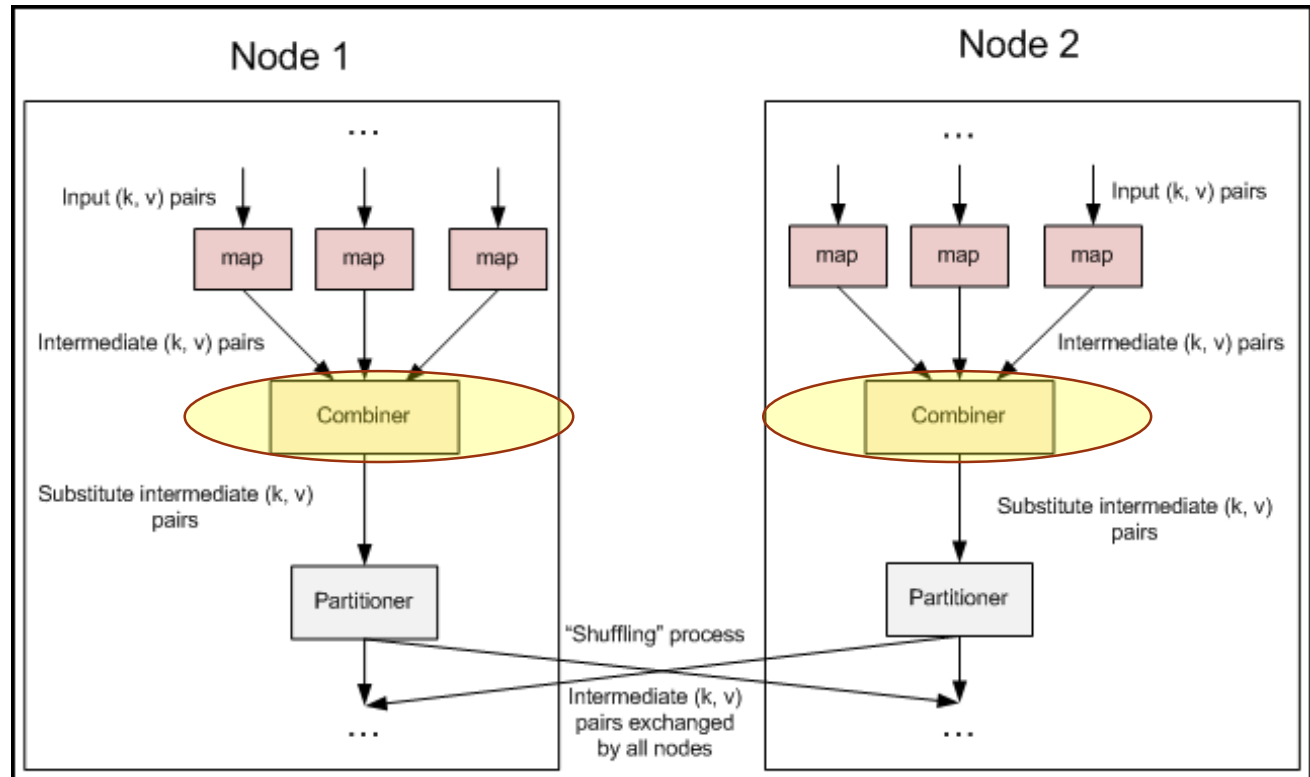
- 每一个Mapper类的实例生成了一个Java进程，负责处理某一个InputSplit上的数据
- 用Mapper.Context提供给每一个Mapper函数，用来提供上面两个对象的功能



Hadoop MapReduce的基本工作原理

Hadoop MapReduce主要组件

Combiner



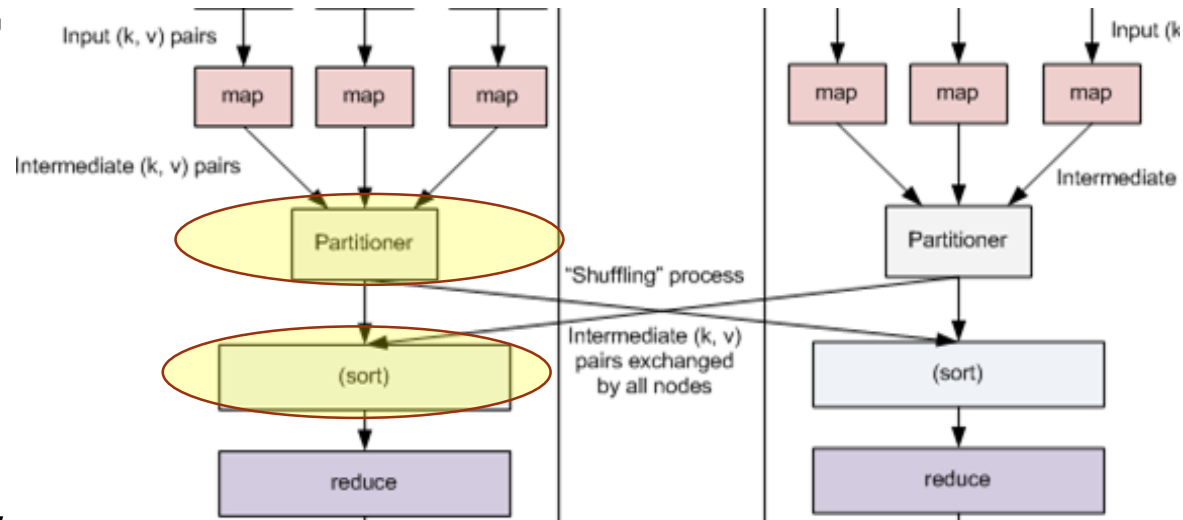
- 合并相同key的键值对，减少partitioning时候的数据通信开销
- `conf.setCombinerClass(Reduce.class);`
- 是在本地执行的一个Reducer，满足一定的条件才能够执行。

Hadoop MapReduce的基本工作原理

Hadoop MapReduce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)对传给哪个Reduce节点



Sort

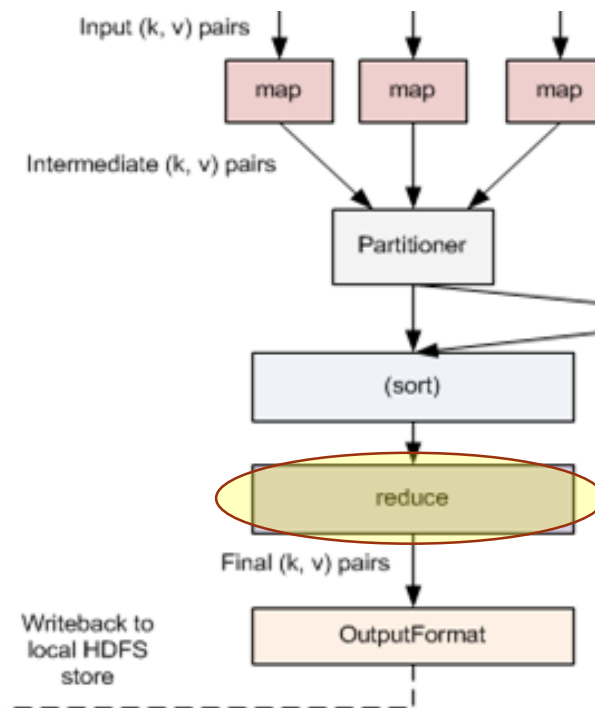
- 传输到每一个Reducer节点上的、将被所有的Reduce函数接收到的Key,value对会被Hadoop自动排序（即Map生成的结果传送到某一个节点的时候，会被自动排序）

Hadoop MapReduce的基本工作原理

Hadoop MapReduce主要组件

Reducer

- 做用户定义的Reduce操作
- 输出环境编程接口是Reducer.Context

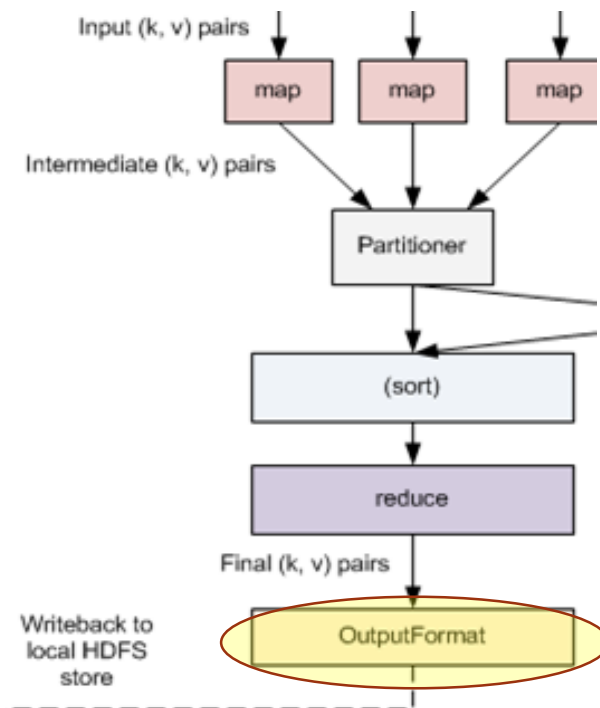


Hadoop MapReduce的基本工作原理

Hadoop MapReduce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()`



Hadoop MapReduce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。

程序执行时的容错处理与计算性能优化

- 由Hadoop系统自己解决
- 主要方法是将失败的任务进行再次执行
- TaskTracker会把状态信息汇报给JobTracker，最终由JobTracker决定重新执行哪一个任务
- 为了加快执行的速度，Hadoop也会自动重复执行同一个任务，以最先执行成功的为准（投机执行）
- `mapred.map.tasks.speculative.execution`
- `mapred.reduce.tasks.speculative.execution`

4. Hadoop的分布式文件系统HDFS

HDFS的基本特征

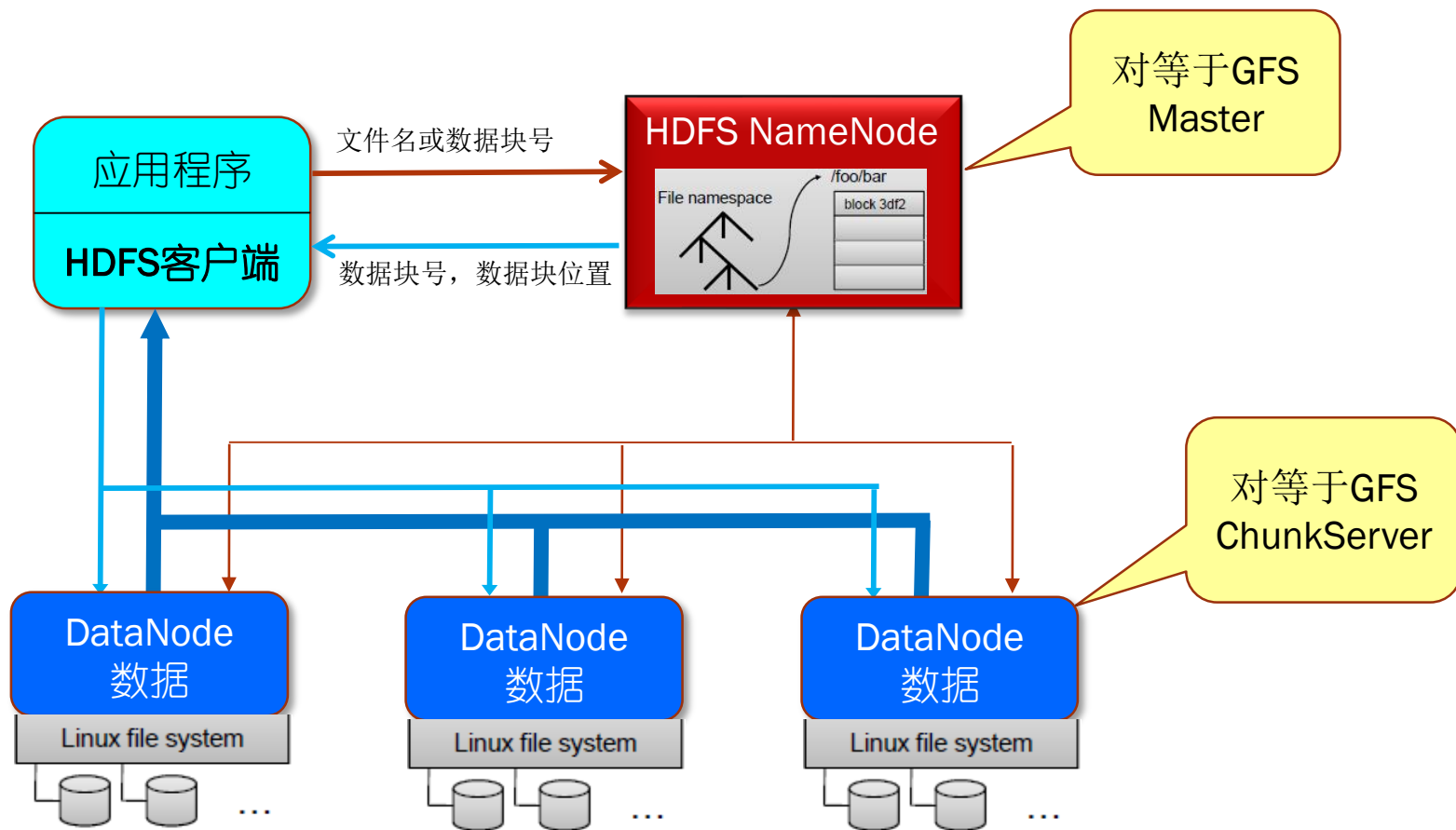
- 模仿Google GFS设计实现
- 存储极大数目的信息（terabytes or petabytes），将数据保存到大量的节点当中；支持很大的单个文件。
- 提供数据的高可靠性和容错能力，单个或者多个节点不工作，对系统不会造成任何影响，数据仍然可用。通过一定数量的数据复制保证数据存储的可靠性和出错恢复能力。
- 提供对数据的快速访问；并提供良好的可扩展性，通过简单加入更多服务器快速扩充系统容量，服务更多的客户端。
- 与GFS类似，HDFS是MapReduce的底层数据存储支撑，并使得数据尽可能根据其本地局部性进行访问与计算。

HDFS的基本特征

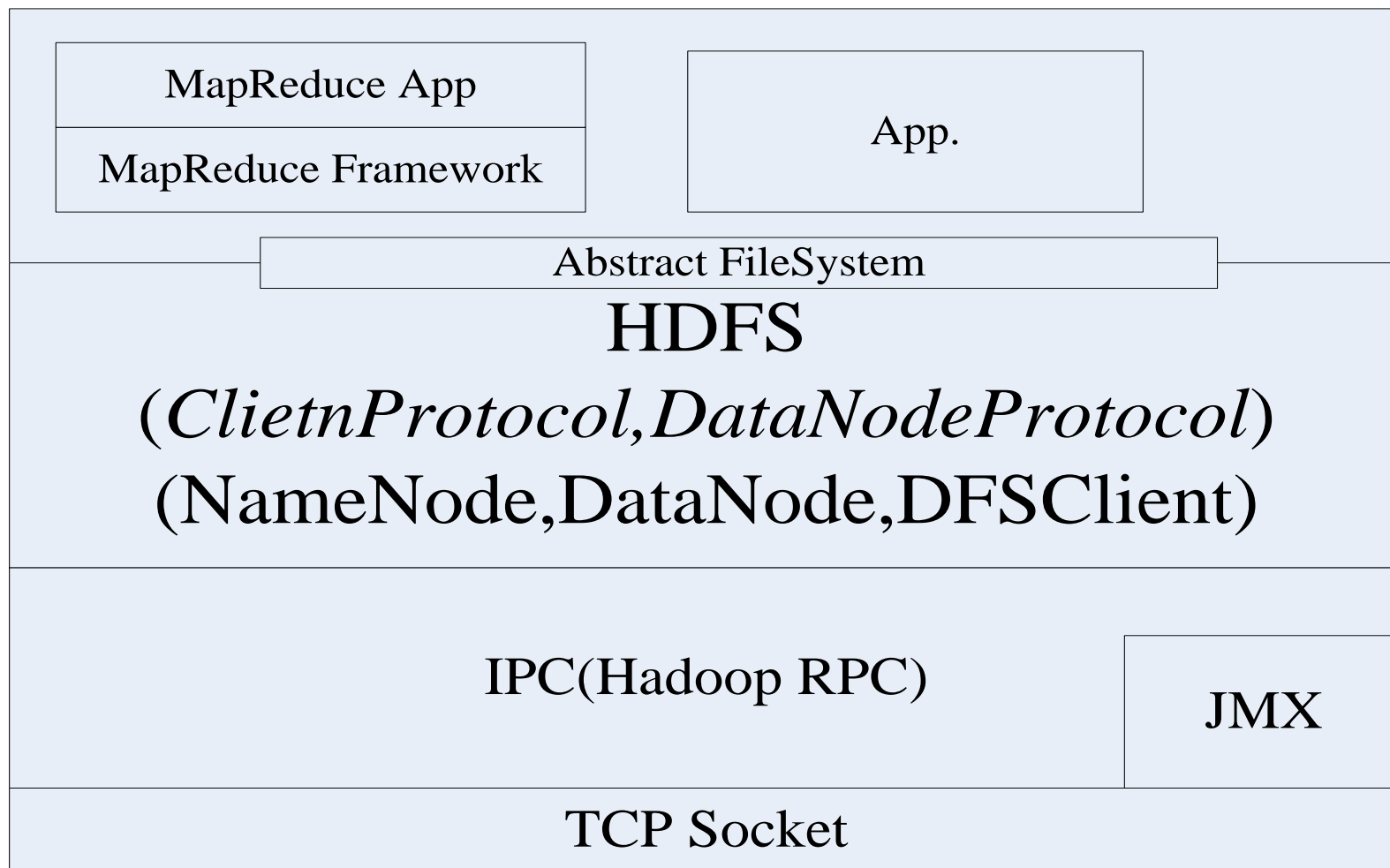
- HDFS对顺序读进行了优化，支持大量数据的快速顺序读出，代价是对于随机的访问负载较高。
- 数据支持一次写入，多次读取；不支持已写入数据的更新操作，但允许在文件尾部添加新的数据
- 数据不进行本地缓存（文件很大，且顺序读没有局部性）
- 基于块的文件存储，默认的块的大小是64MB
 - 减少元数据的量
 - 有利于顺序读写（在磁盘上数据顺序存放）
- 多副本数据块形式存储，按照块的方式随机选择存储节点，默认副本数目是3

Hadoop的分布式文件系统HDFS

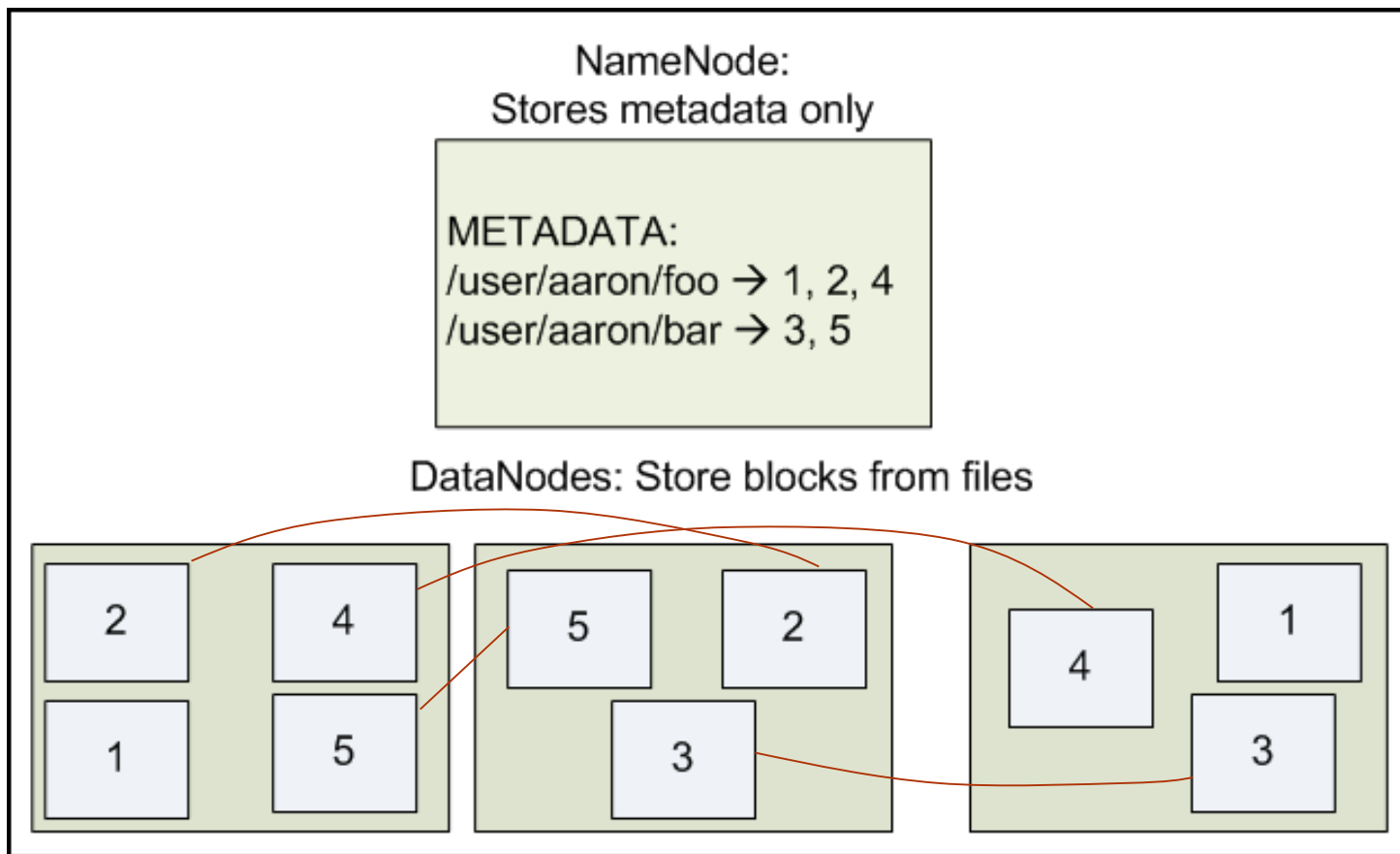
HDFS基本构架



HDFS基本实现构架



HDFS数据分布设计



多副本数据块形式存储，按照块的方式随机选择存储节点
默认副本数目是3

HDFS可靠性与出错恢复

- DataNode节点的检测
 - 心跳：NameNode 不断检测DataNode是否有效
 - 若失效，则寻找新的节点替代，将失效节点数据重新分布
- 集群负载均衡
- 数据一致性：校验和checksum
- 主节点元数据失效
 - Multiple FsImage and EditLog
 - Checkpoint

HDFS设计要点

- 命名空间
- 副本选择
 - Rack Awareness
- 安全模式
 - 刚启动的时候，等待每一个DataNode报告情况
 - 退出安全模式的时候才进行副本复制操作
- NameNode有自己的 FsImage和EditLog，前者有自己的文件系统状态，后者是还没有更新的记录

HDFS的安装和启动

- 下载hadoop-2.7.1.tar.gz
- tar zxvf hadoop-2.7.1.tar.gz，解压后Hadoop系统包括HDFS和所有配置文件都在指定的文件目录中
- 在Linux下进行必要的系统配置
- 设置与Hadoop相关的Java运行环境变量
- 启动Java虚拟机
- 启动Hadoop，则Hadoop和HDFS文件系统开始运行

Hadoop的分布式文件系统HDFS

HDFS文件系统操作命令

```
someone@anynode:hadoop$ bin/hadoop dfs -ls someone@anynode:hadoop$
```

```
someone@anynode:hadoop$ bin/hadoop dfs -ls / Found 2 items  
drwxr-xr-x - hadoop supergroup 0 2008-09-20 19:40 /hadoop  
drwxr-xr-x - hadoop supergroup 0 2008-09-20 20:08 /tmp
```

- 建立用户自己的目录，用户目录在/user中，需要建立

```
someone@anynode:hadoop$ bin/hadoop dfs -mkdir /user
```

```
someone@anynode:hadoop$ bin/hadoop dfs -mkdir /user/someone
```

- 用-put命令在Linux文件系统与HDFS之间复制数据文件
- -put 等同于 -copyFromLocal

```
someone@anynode:hadoop$ bin/hadoop dfs -put  
/home/someone/interestingFile.txt /user/yourUserName/
```

Put上传整个目录

```
someone@anynode:hadoop$ bin/hadoop dfs -put source-directory destination
```

Hadoop的分布式文件系统HDFS

HDFS文件系统操作命令

Command:	Assuming:	Outcome:
<code>bin/hadoop dfs -put foo bar</code>	No file/directory named /user/\$USER/bar exists in HDFS	Uploads local file foo to a file named /user/\$USER/bar
<code>bin/hadoop dfs -put foo bar</code>	/user/\$USER/bar is a directory	Uploads local file foo to a file named /user/\$USER/bar/foo
<code>bin/hadoop dfs -put foo somedir/somefile</code>	/user/\$USER/somedir does not exist in HDFS	Uploads local file foo to a file named /user/\$USER/somedir/somefile, creating the missing directory
<code>bin/hadoop dfs -put foo bar</code>	/user/\$USER/bar is already a file in HDFS	No change in HDFS, and an error is returned to the user.

HDFS文件系统操作命令

<code>-ls path</code>	Lists the contents of the directory specified by path, showing the names, permissions, owner, size and modification date for each entry.
<code>-lsr path</code>	Behaves like <code>-ls</code> , but recursively displays entries in all subdirectories of path.
<code>-du path</code>	Shows disk usage, in bytes, for all files which match path; filenames are reported with the full HDFS protocol prefix.
<code>-dus path</code>	Like <code>-du</code> , but prints a summary of disk usage of all files/directories in the path.
<code>-mv src dest</code>	Moves the file or directory indicated by src to dest, within HDFS.
<code>-cp src dest</code>	Copies the file or directory identified by src to dest, within HDFS.

HDFS文件系统操作命令

<code>-rm path</code>	Removes the file or empty directory identified by path.
<code>-rmr path</code>	Removes the file or directory identified by path. Recursively deletes any child entries (i.e., files or subdirectories of path).
<code>-put localSrc dest</code>	Copies the file or directory from the local file system identified by localSrc to dest within the HDFS.
<code>-copyFromLocal localSrc dest</code>	Identical to -put
<code>-moveFromLocal localSrc dest</code>	Copies the file or directory from the local file system identified by localSrc to dest within HDFS, then deletes the local copy on success.

Hadoop的分布式文件系统HDFS

HDFS文件系统操作命令

<code>-get [-crc] src localDest</code>	Copies the file or directory in HDFS identified by src to the local file system path identified by localDest.
<code>-getmerge src localDest [addnl]</code>	Retrieves all files that match the path src in HDFS, and copies them to a single, merged file in the local file system identified by localDest.
<code>-cat filename</code>	Displays the contents of filename on stdout.
<code>-copyToLocal [-crc] src localDest</code>	Identical to -get
<code>-moveToLocal [-crc] src localDest</code>	Works like -get, but deletes the HDFS copy on success.
<code>-mkdir path</code>	Creates a directory named path in HDFS. Creates any parent directories in path that are missing (e.g., like mkdir -p in Linux).

HDFS文件系统操作命令

<code>-setrep [-R] [-w] rep path</code>	Sets the target replication factor for files identified by path to rep. (The actual replication factor will move toward the target over time)
<code>-touchz path</code>	Creates a file at path containing the current time as a timestamp. Fails if a file already exists at path, unless the file is already size 0.
<code>-test [-ezd] path</code>	Returns 1 if path exists; has zero length; or is a directory, or 0 otherwise.
<code>-stat [format] path</code>	Prints information about path. format is a string which accepts file size in blocks (%b), filename (%n), block size (%o), replication (%r), and modification date (%y, %Y).
<code>-tail [-f] file</code>	Shows the last 1KB of file on stdout.

HDFS文件系统操作命令

<code>-chmod [-R] mode,mode,... path...</code>	Changes the file permissions associated with one or more objects identified by path.... Performs changes recursively with -R. mode is a 3-digit octal mode, or {augo}+/{rwxX}. Assumes a if no scope is specified and does not apply a umask.
<code>-chown [-R] [owner][:group] path...</code>	Sets the owning user and/or group for files or directories identified by path.... Sets owner recursively if -R is specified.
<code>-chgrp [-R] group path...</code>	Sets the owning group for files or directories identified by path.... Sets group recursively if -R is specified.
<code>-help cmd</code>	Returns usage information for one of the commands listed above. You must omit the leading '-' character in cmd

HDFS Admin命令

获得HDFS总体的状态

- `bin/hadoop dfsadmin -report`
- `bin/hadoop dfsadmin -metasave filename`
 - what the state of the NameNode's metadata is
- Safemode
 - Safemode is an HDFS state in which the file system is mounted read-only; no replication is performed, nor can files be created or deleted.
 - `bin/hadoop dfsadmin -safemode enter/leave/get/wait`

HDFS Admin命令

- 更改HDFS成员
- 升级HDFS版本
 - bin/start-dfs.sh -upgrade(第一次运行新版本的时候使用)
 - bin/hadoop dfsadmin -upgradeProgress status
 - bin/hadoop dfsadmin -upgradeProgress details
 - bin/hadoop dfsadmin -upgradeProgress force (on your own risk!)
 - bin/start-dfs.sh -rollback(在旧版本重新安装后使用)(on your own risk!)
- 帮助 bin/admin dfsadmin -help

负载均衡

- 加入一个新节点的步骤

配置新节点上的hadoop程序

在Master的slaves文件中加入新的slave节点

启动slave节点上的DataNode，会自动去联系NameNode，加入到集群中

- Balancer类用来做负载均衡，默认的均衡参数是10%范围内
- `bin/start-balancer.sh -threshold 5`
- `bin/stop-balancer.sh` 随时可以停止负载均衡的工作

在MapReduce程序中使用HDFS

- 通过fs.default.name的配置选项，Hadoop MapReduce程序可以自动从NameNode中获得文件的情况
- HDFS接口包括：
 - 命令行接口
 - Hadoop MapReduce Job隐含的输入
 - Java程序直接操作
 - libhdfs从c/c++程序中操作

HDFS权限控制与安全特性

- 类似于POSIX的安全特性
- 不完全，主要预防操作失误
- 不是一个强的安全模型，不能保证操作的完全安全性
- `bin/hadoop dfs -chmod,-chown,-chgrp`
- 用户:当前登录的用户名, 即使用Linux自身设定的用户与组的概念
- 超级用户: The username which was used to start the Hadoop process (i.e., the username who actually ran `bin/start-all.sh` or `bin/start-dfs.sh`) is acknowledged to be the *superuser* for HDFS. If this user interacts with HDFS, he does so with a special username *superuser*. If Hadoop is shutdown and restarted under a different username, that username is then bound to the *superuser* account.
- 超级用户组

配置参数: `dfs.permissions.supergroup`

5. Hadoop HDFS的编程

FileSystem基类

- FileSystem是一个用来与文件系统交互的抽象类，可以通过实现FileSystem的子类来处理具体的文件系统，比如HDFS或者其它文件系统
- 通过factory方法FileSystem.get(Configuration conf)获得所需的文件系统实例

```
Configuration conf = new Configuration();
```

```
FileSystemhdfs = FileSystem.get(conf);
```

- Hadoop中，使用Path类的对象来编码目录或者文件的路径，使用FileStatus类来存放目录和文件的信息。

HDFS基本文件操作

- 创建文件

create方法有很多种定义形式，但一般仅需使用简单的几种

```
public FSDDataOutputStream create(Path f);
```

```
public FSDDataOutputStream create(Path f,boolean overwrite);
```

```
public FSDDataOutputStream create  
    (Path f,boolean overwrite,int bufferSize);
```

HDFS基本文件操作

- 打开文件

FileSystem.open方法有2个，参数最多的一个定义如下：

```
public abstract FSDataInputStream  
    open(Path f,int bufferSize)  
    throws IOException
```

f: 文件名

bufferSize: 文件缓存大小。默认值： Configuration中
io.file.buffer.size的值，如果Configuration中未显式设置该值则是
4096。

HDFS基本文件操作

- 获取文件信息

FileSystem.getFileStatus方法格式如下：

```
public abstract FileStatus getFileStatus(Path f)  
    throws IOException;
```

返回一个FileStatus对象。FileStatus保存文件的很多信息，包括：

- path: 文件路径
- length: 文件长度
- isDir: 是否为目录
- block_replication: 数据块副本因子
- blockSize: 文件长度（数据块数）
- modification_time: 最近一次修改时间
- access_time: 最近一次访问时间
- owner: 文件所属用户
- group: 文件所属组

如果想了解文件的这些信息，可以在获得文件的FileStatus实例之后，调用相应的getXXX方法（比如，FileStatus.getModificationTime()获得最近修改时间）

HDFS基本文件操作

- 获取目录信息

获取目录信息，不仅是目录本身，还有目录之下的文件和子目录信息：

```
public FileStatus[] listStatus(Path f) throws IOException;
```

如果f是目录，那么将目录之下的每个目录或文件信息保存在FileStatus数组中返回。

如果f是文件，和getFileStatus功能一致。另外，listStatus还有参数为Path[]的版本的接口定义以及参数带路径过滤器PathFilter的接口定义，参数为Path[]的listStatus就是对这个数组中的每个path都调用上面的参数为Path的listStatus。参数中的PathFilter则是一个接口，实现接口的accept方法可以自定义文件过滤规则。

HDFS基本文件操作

- 文件读取

调用open打开文件之后，使用了一个FSDataInputStream对象来负责数据的读取。通过FSDataInputStream进行文件读取时，提供的API就是FSDataInputStream.read方法：

```
public int read(long position, byte[] buffer, int offset, int length)
throws IOException
```

从文件的指定位置position开始，读取最多length字节的数据，保存到buffer中从offset个元素开始的空间中；返回值为实际读取的字节数。此函数不改变文件当前offset值。但使用更多的还有一种简化版本：

```
public final int read(byte[] b) throws IOException
```

从文件当前位置读取最多长度为b.len的数据保存到b中，返回值为实际读取的字节数。

HDFS基本文件操作

- 文件写入

从接口定义可以看出，调用create创建文件以后，使用了一个FSDataOutputStream对象来负责数据的写入。通过FSDataOutputStream进行文件写入时，最常用的API就是write方法：

```
public void write(byte[] b,int off,int len)  
    throws IOException
```

函数的意义是：将b中从off开始的最多len个字节的数据写入文件当前位置。返回值为实际写入的字节数。

HDFS基本文件操作

- 关闭

关闭为打开的逆过程，`FileSystem.close`定义如下：

```
public void close()throws IOException
```

不需要其它操作而关闭文件。释放所有持有的锁。

- 删除

```
public abstract boolean delete(Path f,boolean recursive) throws IOException
```

f: 待删除文件名

recursive: 如果recursive为true，并且f是目录，那么会递归删除f下所有文件。

f是文件的话，recursive为true还是false无影响。

另外，类似Java中File的接口DeleteOnExit，如果某些文件需要删除，但是当前不能被删；或者说当时删除代价太大，想留到退出时再删除的话，FileSystem中也提供了一个deleteOnExit接口：

```
public boolean deleteOnExit(Path f) throws IOException
```

标记文件f，当文件系统关闭时才真正删除此文件，但是这个文件f必须存在。

Hadoop HDFS的编程

HDFS编程实例

获取一个指定HDFS目录下所有文件的信息，对每一个文件，打开文件、循环读取每行数据、检查每行是否包含指定的字符串，如包含则写入本地目标文件。

```
import java.util.Scanner;
import java.io.IOException;
import java.io.File;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class resultFilter
{
    public static void main(String[] args) throws IOException
    {
        Configuration conf = new Configuration(); //以下两句中，hdfs和local分别对应HDFS实例和本地文件系统实例
        FileSystem hdfs = FileSystem.get(conf);
        FileSystem local = FileSystem.getLocal(conf);
        Path inputDir, localFile;
        FileStatus[] inputFiles;
        FSDataOutputStream out = null;
        FSDataInputStream in = null;
        Scanner scan; String str;    byte[] buf;        int singleFileLines;                int numLines, numFiles, i;
        inputDir = new Path(args[0]);
        singleFileLines = Integer.parseInt(args[3]);
    }
}
```

Hadoop HDFS的编程

HDFS编程实例

```
try { inputFiles = hdfs.listStatus(inputDir); //获得目录信息
    numLines = 0;          numFiles = 1; //输出文件从1开始编号
    localFile = new Path(args[1]);
    if(local.exists(localFile)) local.delete(localFile, true); //若目标路径存在，则删除之
    for (i = 0; i<inputFiles.length; i++) {
        if(inputFiles[i].isDir() == true) //忽略子目录
            continue;
        System.out.println(inputFiles[i].getPath().getName());
        in = hdfs.open(inputFiles[i].getPath()); scan = new Scanner(in);
        while (scan.hasNext()) {
            str = scan.nextLine();
            if(str.indexOf(args[2])==-1) continue; //如果该行没有match字符串，则忽略
            numLines++;
            if(numLines == 1) //如果是1，说明需要新建文件了
            { localFile = new Path(args[1] + File.separator + numFiles);
              out = local.create(localFile); //创建文件
              numFiles++;
            }
            buf = (str+"\n").getBytes();
            out.write(buf, 0, buf.length); //将字符串写入输出流
            if(numLines == singleFileLines) //如果已满足相应行数，关闭文件
            { out.close(); numLines = 0; //行数变为0，重新统计      }
        } //end of while
        scan.close(); in.close();
    } //end of for
    if(out != null)      out.close();
} //end of try
catch (IOException e) { e.printStackTrace();}
} //end of main
} //end of resultFilter
```

Thanks!