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，顾荣

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.分布式文件系统GFS的基本工作原理

4.分布式结构化数据表BigTable

5. Hadoop MapReduce的基本工作原理

6. Hadoop 分布式文件系统HDFS

7. 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

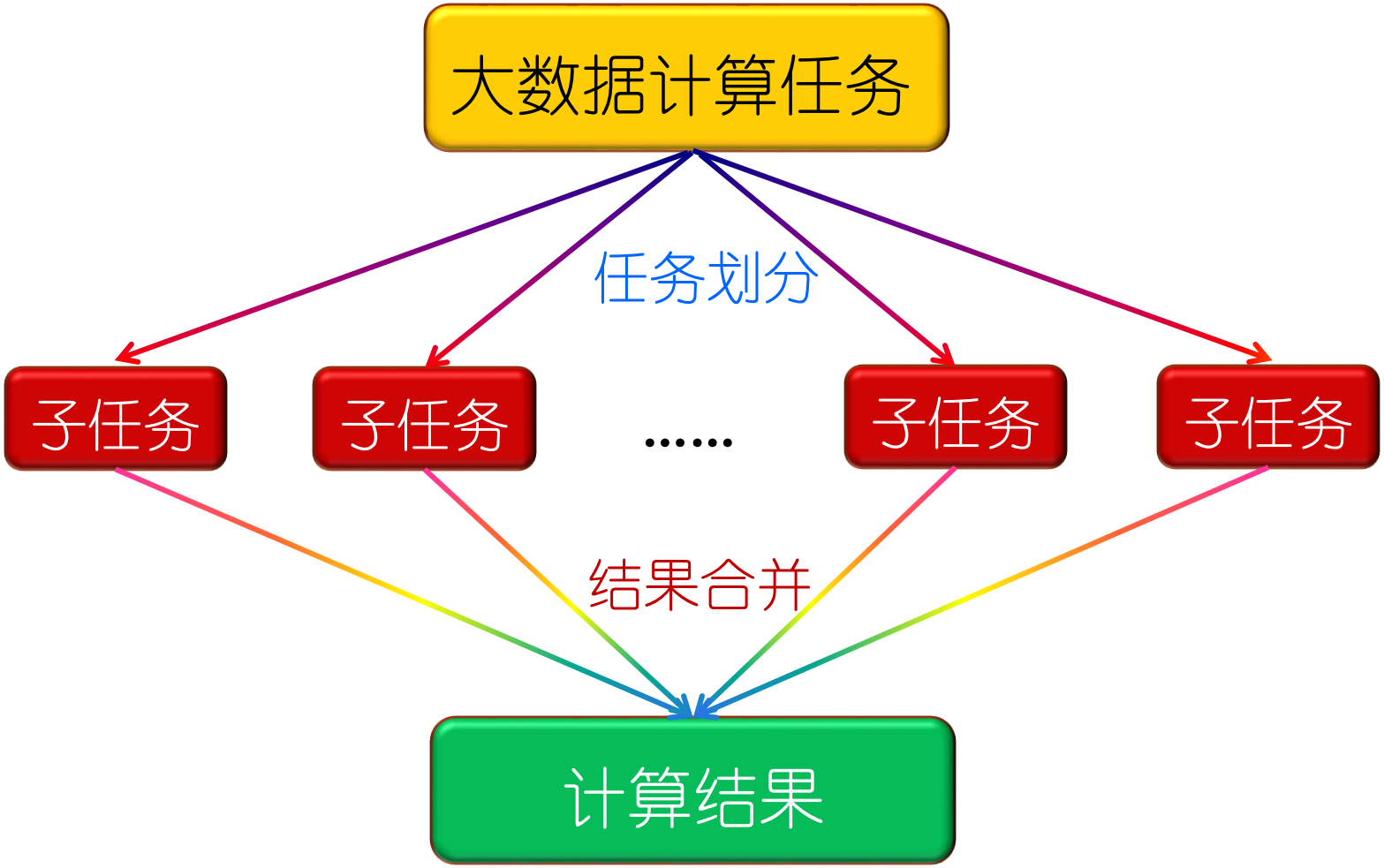
上升到抽象模型：Mapper与Reducer

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

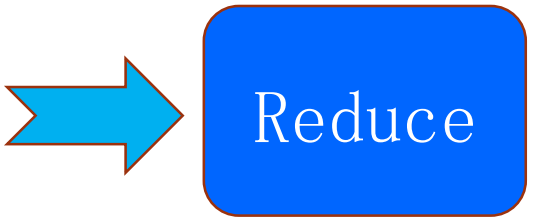


建立Map和Reduce抽象模型

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



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

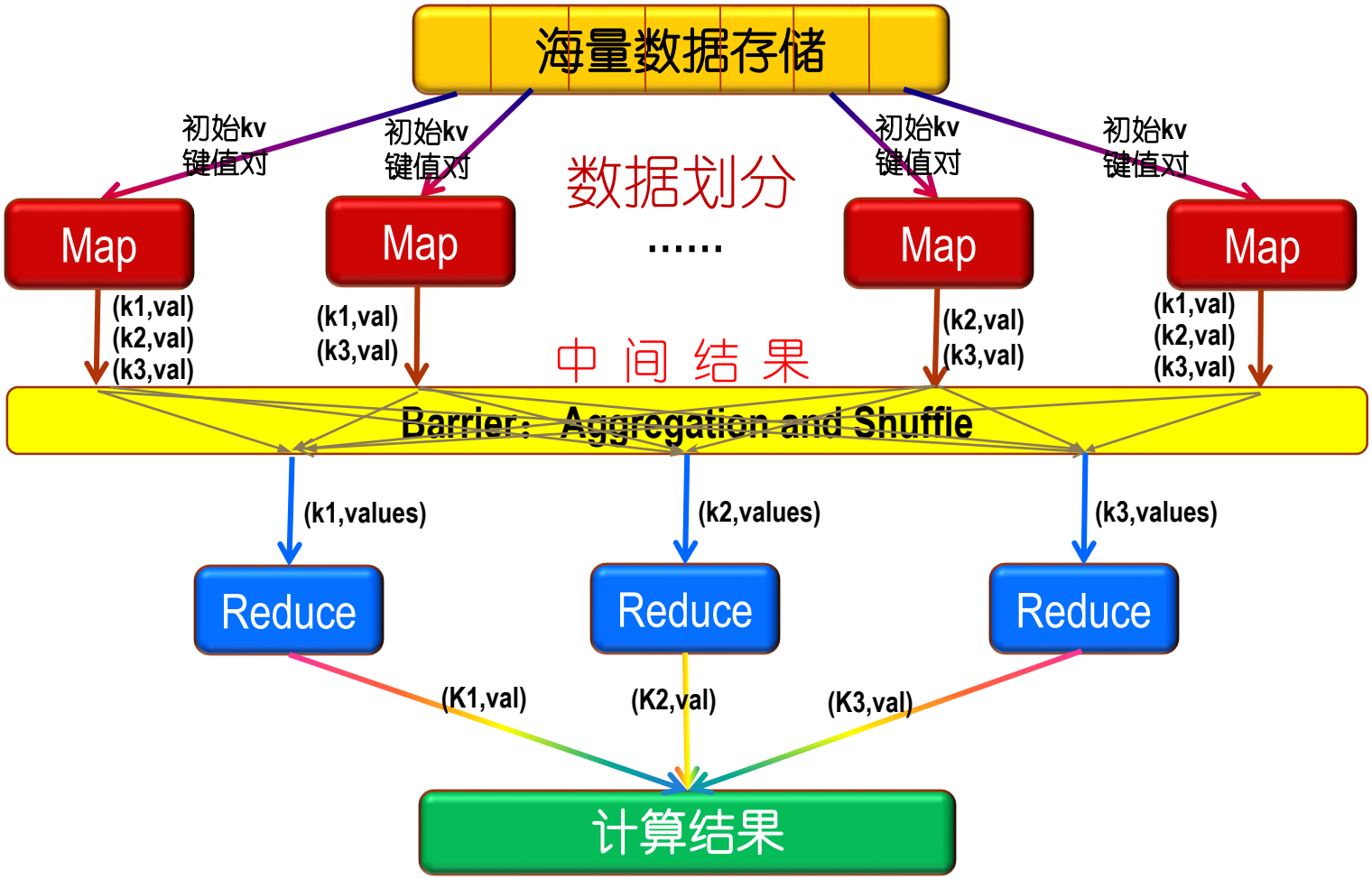
借鉴函数式程序设计语言Lisp中的思想，定义了Map和Reduce两个抽象的操作函数：

- map: (k1; v1) → [(k2; v2)]
- reduce: (k2; [v2]) → [(k3; v3)]

特点：

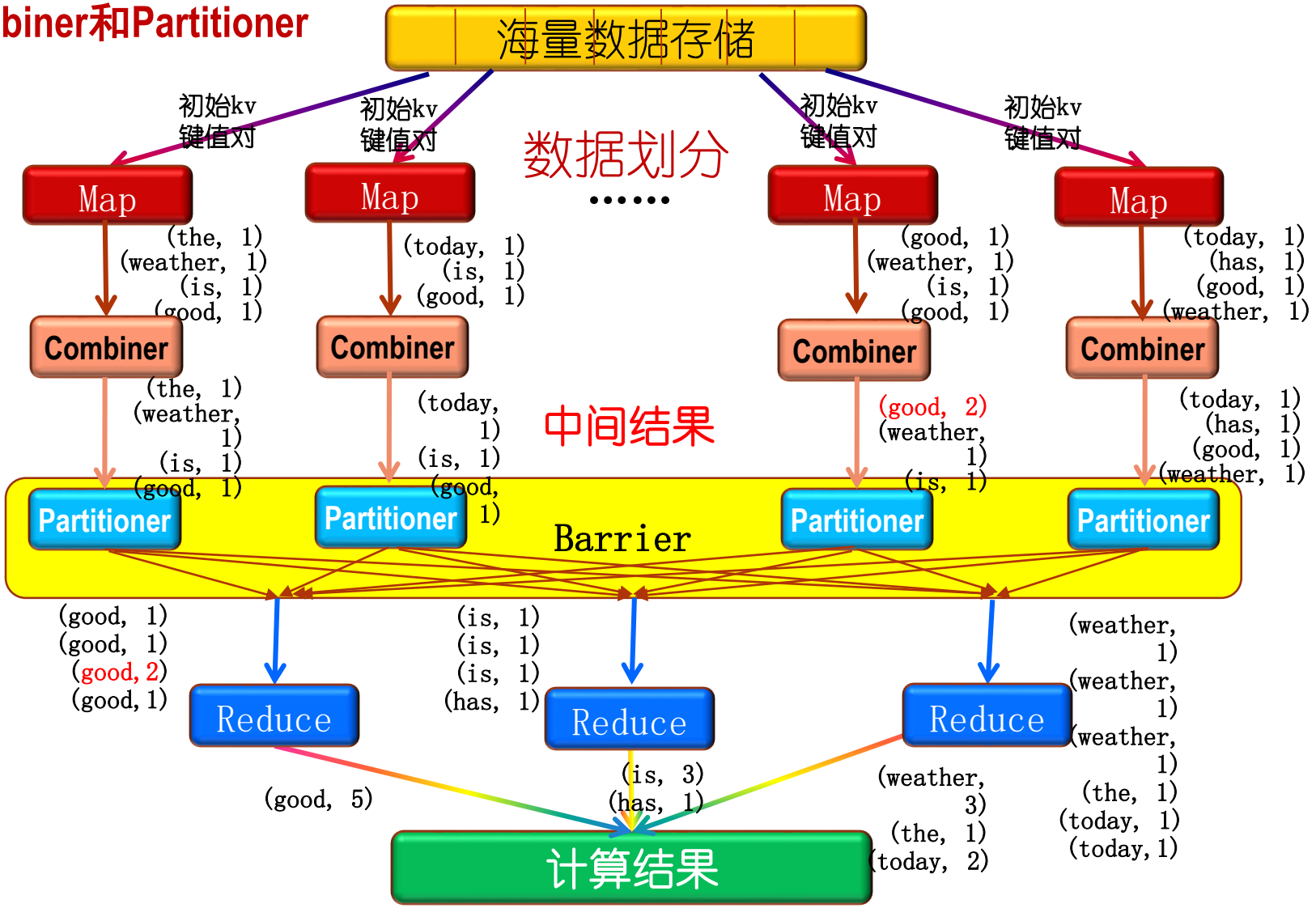
- **描述了对一组数据处理的两个阶段的抽象操作**
- **仅仅描述了需要做什么，不需要关注怎么做**

上升到构架--自动并行化并隐藏低层细节



上升到构架--自动并行化并隐藏低层细节

Combiner和Partitioner

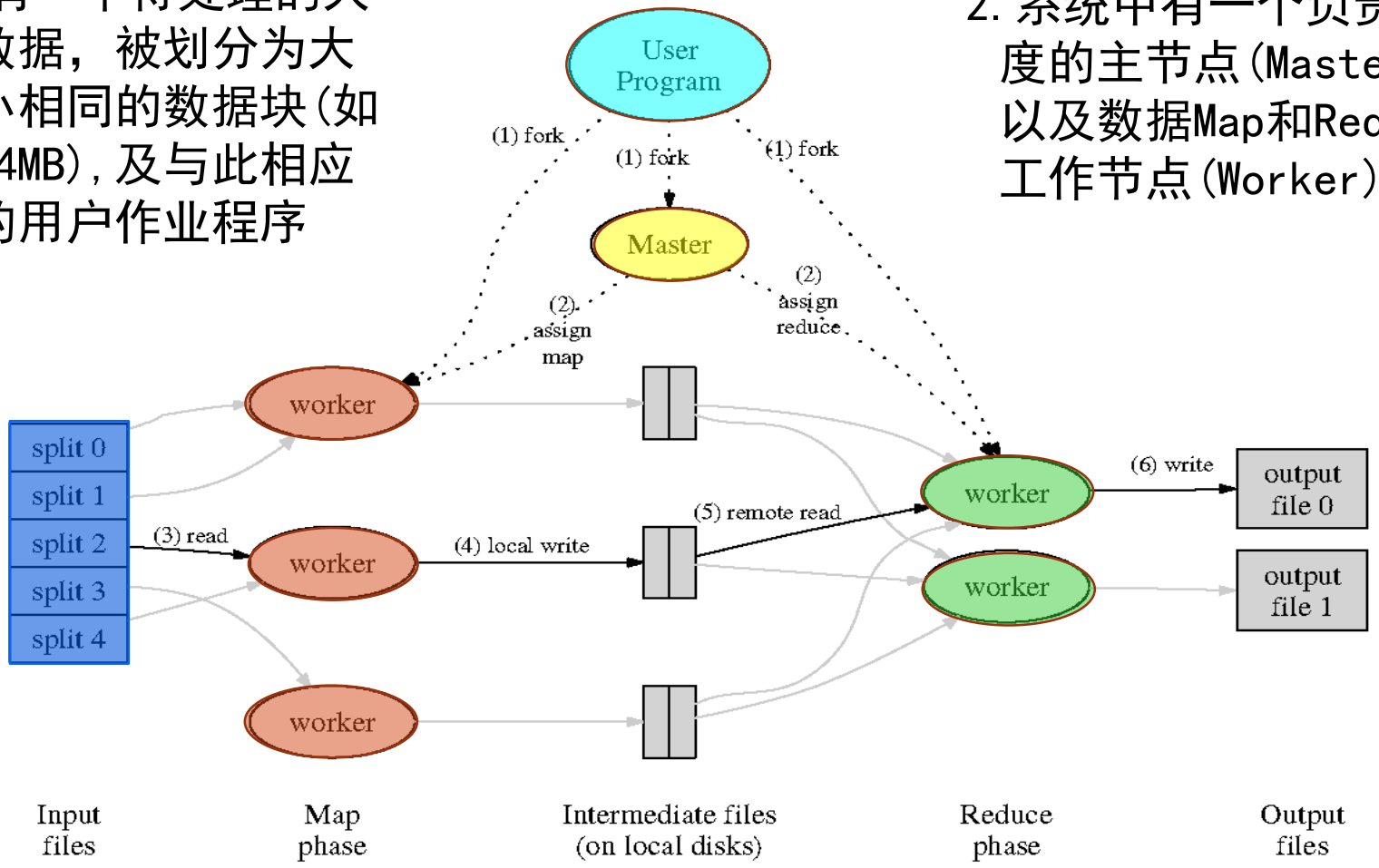


2. Google MapReduce的基本工作原理

Google MapReduce并行处理的基本过程

1. 有一个待处理的大数据，被划分为大小相同的数据块(如64MB)，及与此相应的用户作业程序

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)



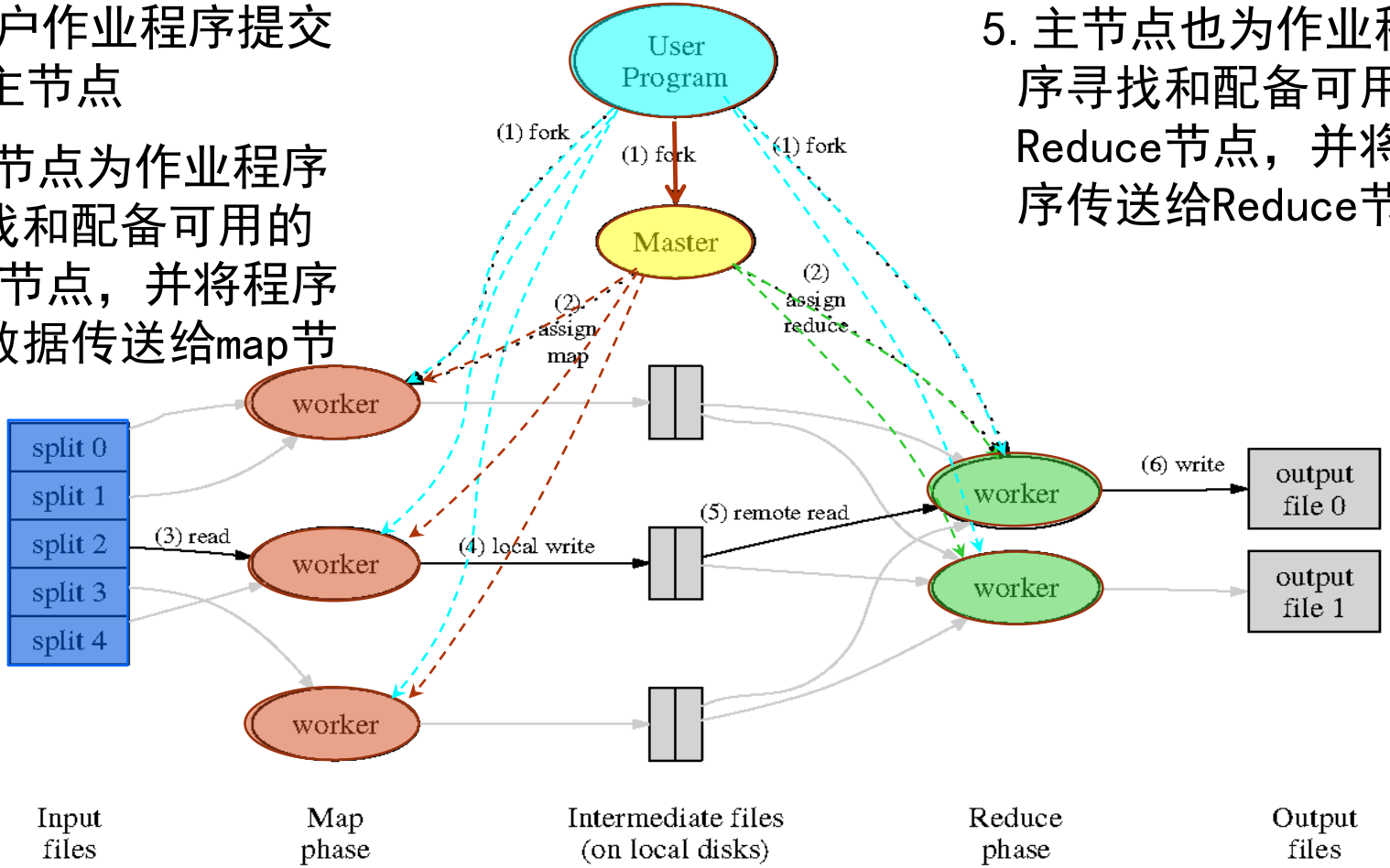
Cite from Dean and Ghemawat (OSDI 2004)

Google MapReduce并行处理的基本过程

3. 用户作业程序提交给主节点

4. 主节点为作业程序寻找和配备可用的Map节点，并将程序和数据传送给map节点

5. 主节点也为作业程序寻找和配备可用的Reduce节点，并将程序传送给Reduce节点

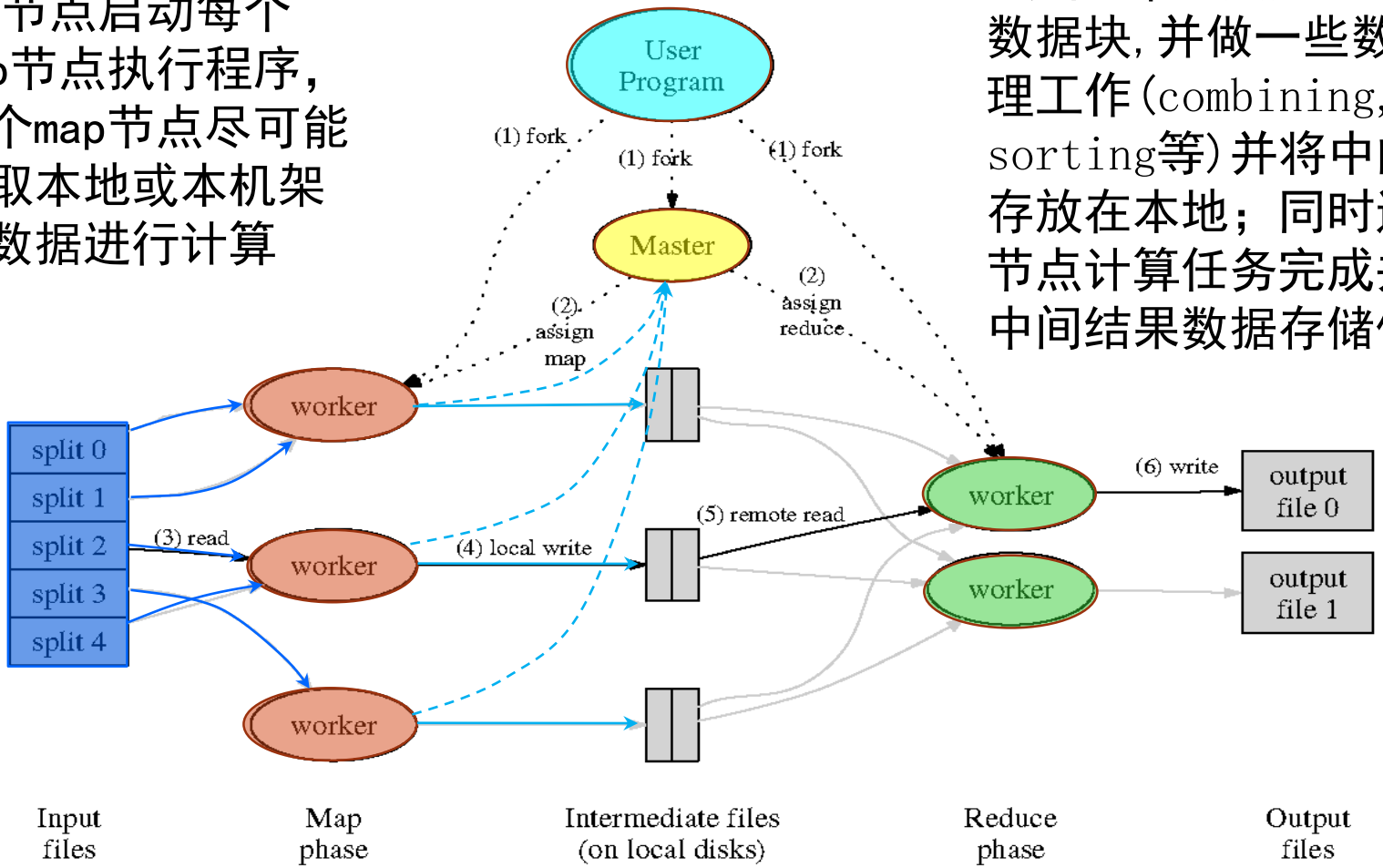


Cite from Dean and Ghemawat (OSDI 2004)

Google MapReduce并行处理的基本过程

6. 主节点启动每个Map节点执行程序，每个map节点尽可能读取本地或本机架的数据进行计算

7. 每个Map节点处理读取的数据块，并做一些数据整理工作(combining, sorting等)并将中间结果存放在本地；同时通知主节点计算任务完成并告知中间结果数据存储位置

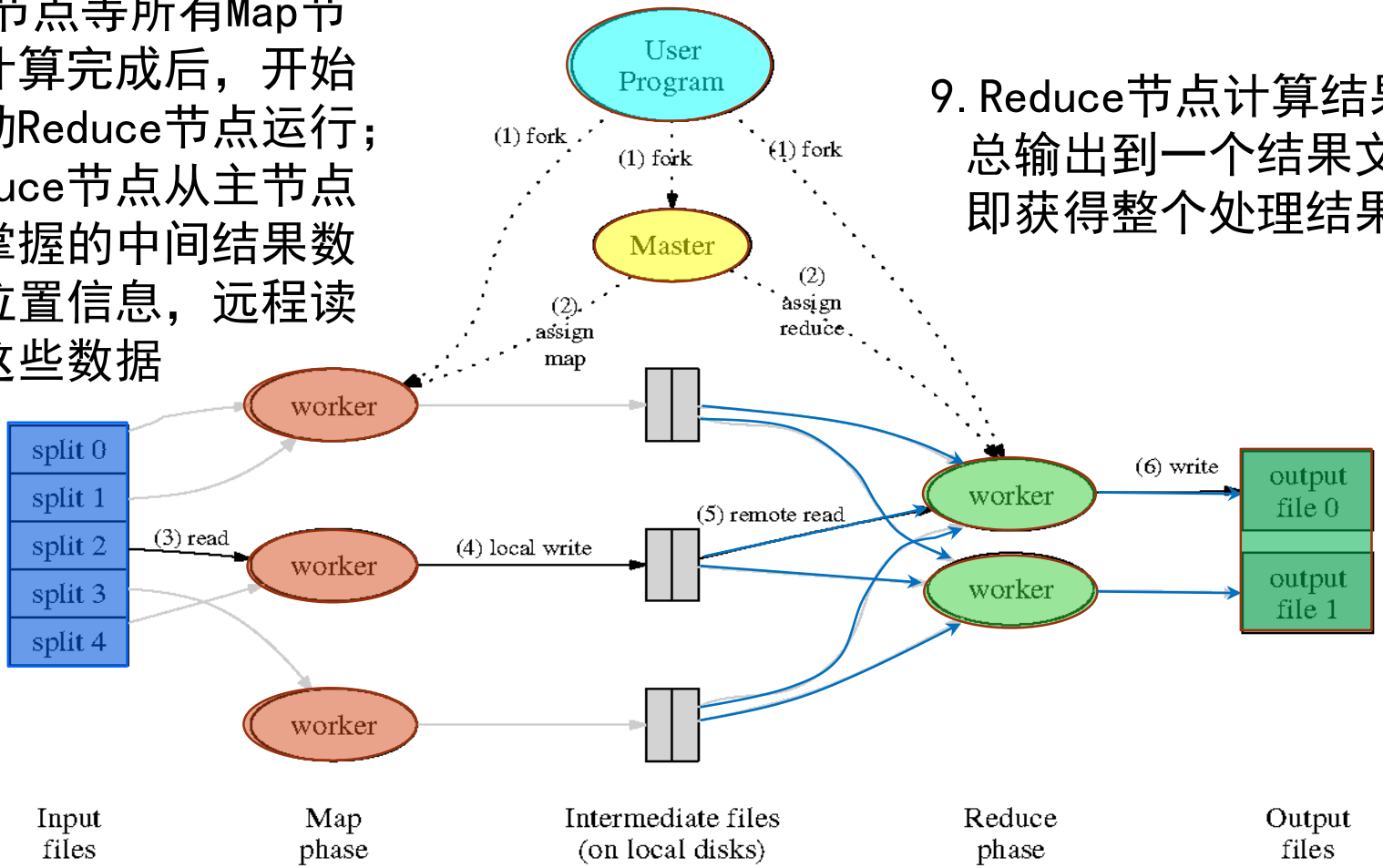


Cite from Dean and Ghemawat (OSDI 2004)

Google MapReduce并行处理的基本过程

8. 主节点等所有Map节点计算完成后，开始启动Reduce节点运行；Reduce节点从主节点所掌握的中间结果数据位置信息，远程读取这些数据

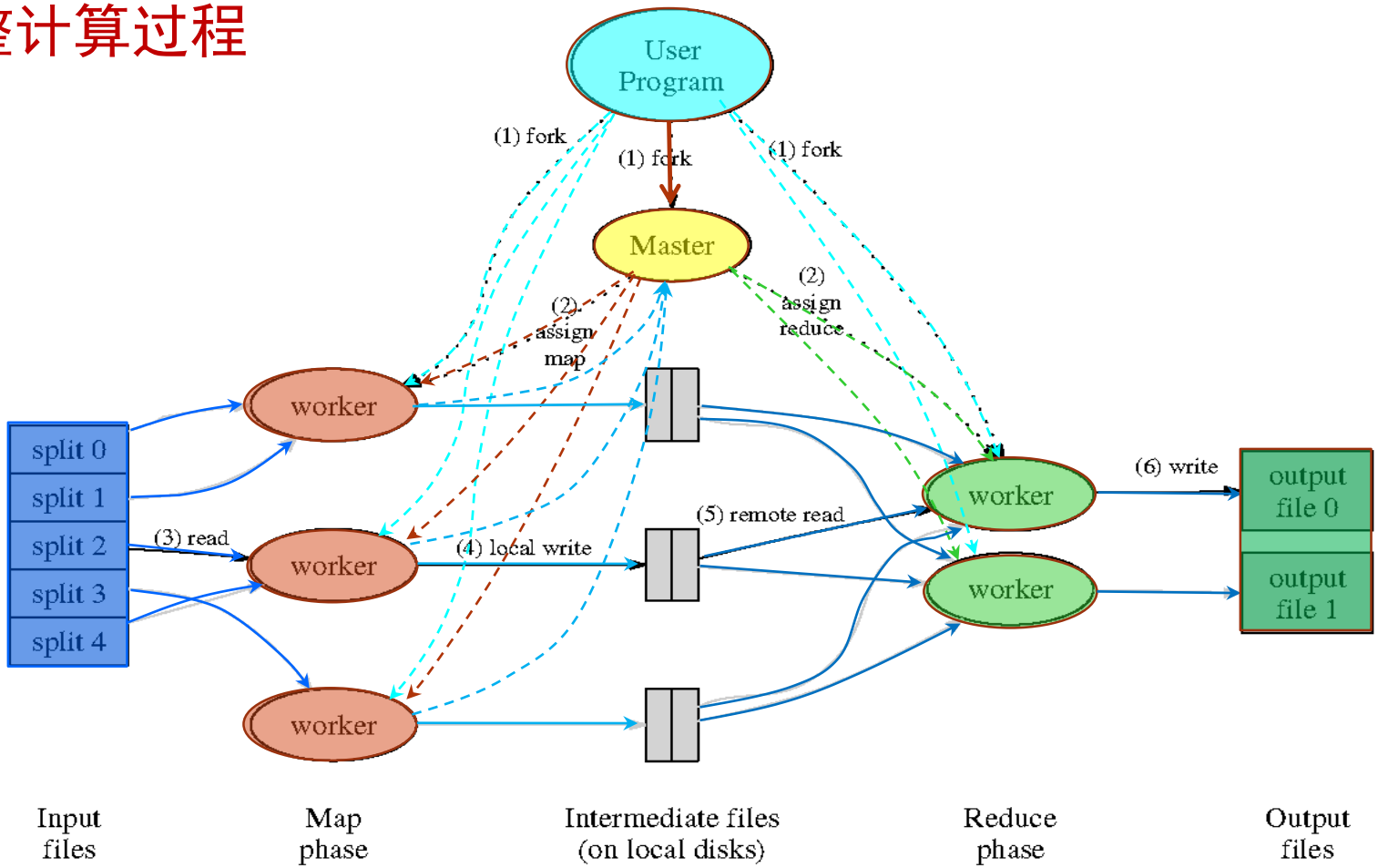
9. Reduce节点计算结果汇总输出到一个结果文件即获得整个处理结果



Cite from Dean and Ghemawat (OSDI 2004)

Google MapReduce并行处理的基本过程

完整计算过程



Cite from Dean and Ghemawat (OSDI 2004)

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队将由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. MapReduce分布式文件系统GFS的工作原理

基本问题

海量数据怎么存储？数据存储可靠性怎么解决？

当前主流的分布文件系统有：

- RedHat的GFS
- IBM的GPFS
- Sun的Lustre等

主要用于对硬件设施要求很高的高性能计算或大型数据中心；

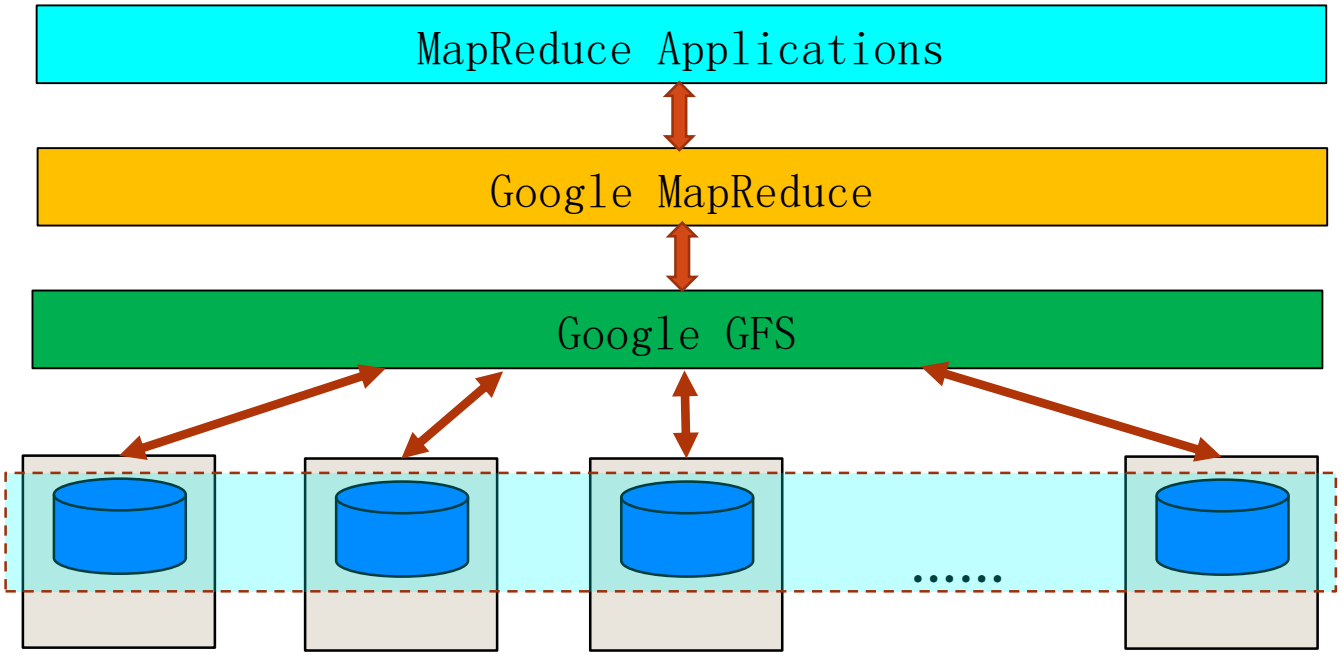
价格昂贵且缺少完整的数据存储容错解决方案

如Lustre只对元数据管理提供容错处理，但对于具体的分布存储节点，可靠性完全依赖于这些分布节点采用RAID或存储区域网(SAN)技术提供容错，一旦分布节点失效，数据就无法恢复。

Google GFS的基本设计原则

Google GFS是一个基于分布式集群的大型分布式文件系统，为MapReduce计算框架提供低层数据存储和数据可靠性支撑；

GFS是一个构建在分布节点本地文件系统之上的一个逻辑上文件系统，它将数据存储在物理上分布的每个节点上，但通过GFS将整个数据形成一个逻辑上整体的文件。



Google GFS的基本设计原则

- 廉价本地磁盘分布存储

  各节点本地分布式存储数据，优点是不需要采用价格较贵的集中式磁盘阵列，容量可随节点数增加自动增加

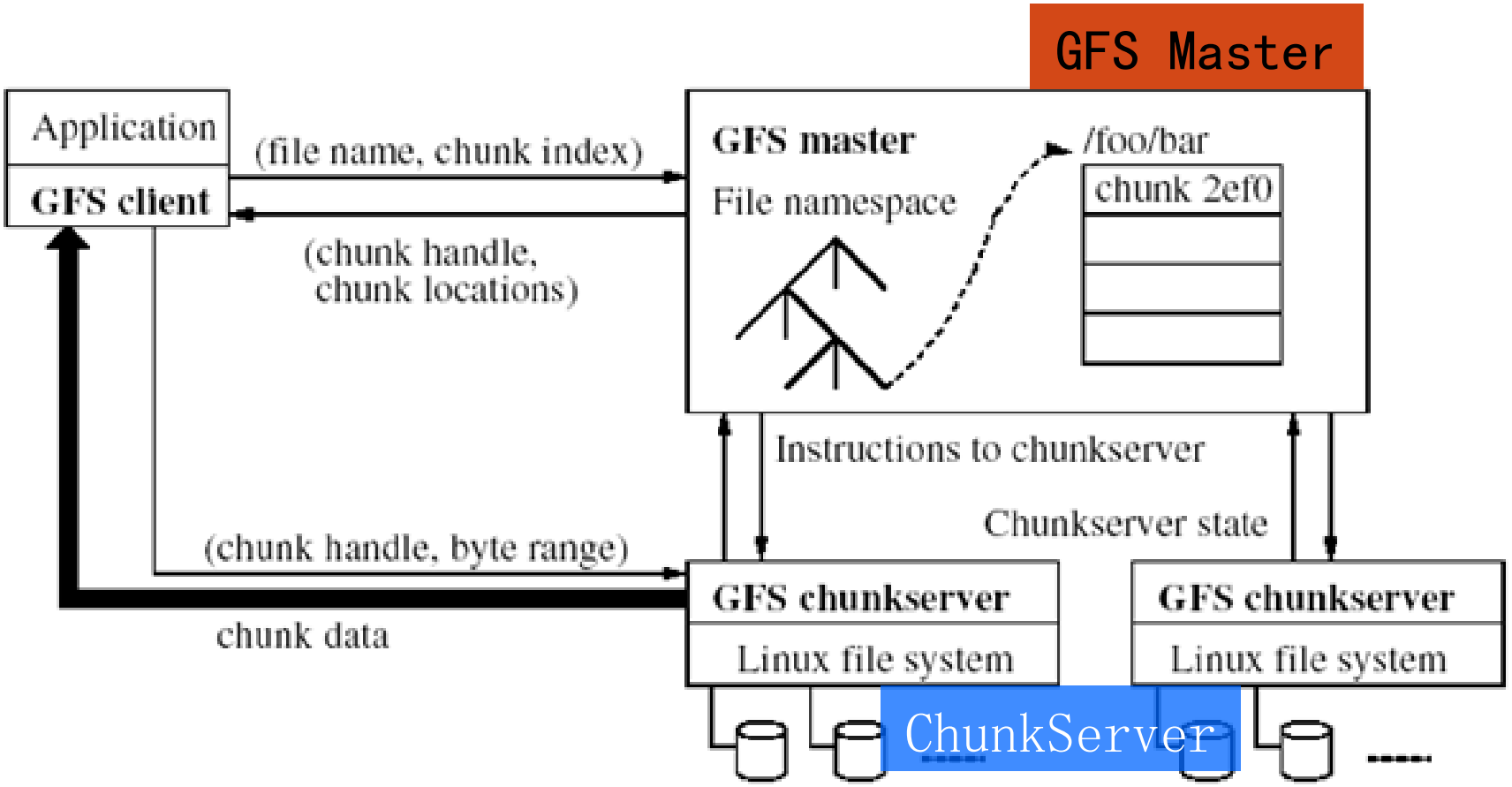
- 多数据自动备份解决可靠性

  采用廉价的普通磁盘，把磁盘数据出错视为常态，用自动多数据备份存储解决数据存储可靠性问题

- 为上层的MapReduce计算框架提供支撑

  GFS作为向上层MapReduce执行框架的底层数据存储支撑，负责处理所有的数据自动存储和容错处理，因而上层框架不需要考虑低层的数据存储和数据容错问题

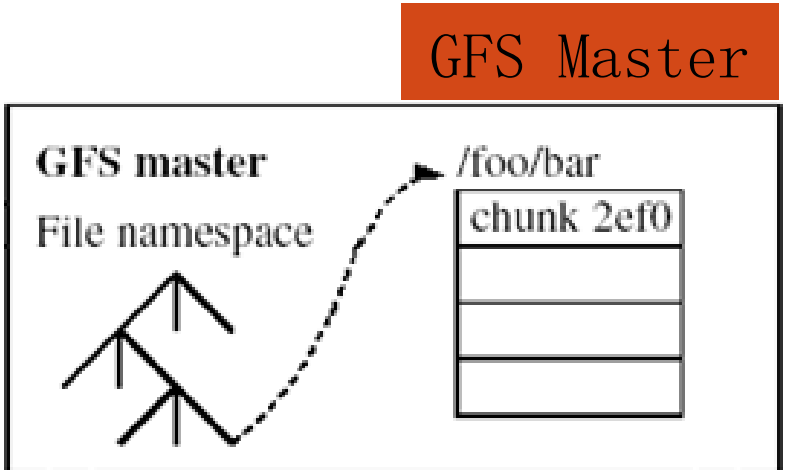
Google GFS的基本构架和工作原理



Cite from Ghemawat et al. (SOSP 2003)

Google GFS的基本构架和工作原理

GFS Master



Master上保存了GFS文件系统的三种元数据：

- 命名空间(Name Space),即整个分布式文件系统的目录结构
- Chunk与文件名的映射表
- Chunk副本的位置信息，每一个Chunk默认有3个副本

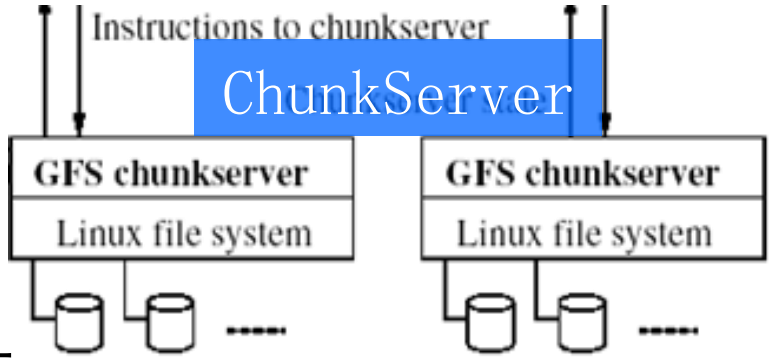
前两种元数据可通过操作日志提供容错处理能力；
第3个元数据直接保存在ChunkServer上， Master 启动或Chunk Server注册时自动完成在Chunk Server上元数据的生成；
因此，当Master失效时，只要ChunkServer数据保存完好，可迅速恢复Master上的元数据。

MapReduce分布式文件系统GFS的工作原理

Google GFS的基本构架和工作原理

GFS ChunkServer

即用来保存大量实际数据的数据服务器。

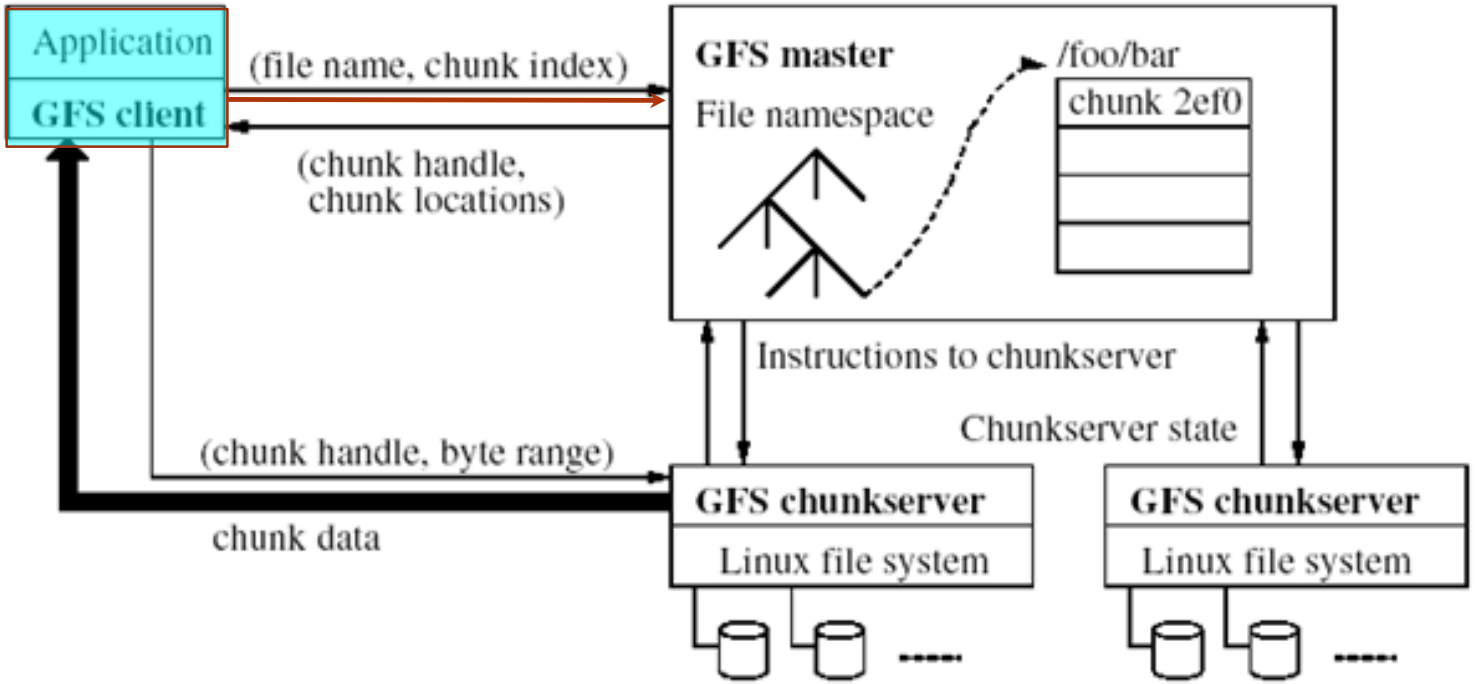


- GFS中每个数据块划分缺省为64MB
- 每个数据块会分别在3个(缺省情况下)不同的地方复制副本；
- 对每一个数据块，仅当3个副本都更新成功时，才认为数据保存成功。
- 当某个副本失效时，Master会自动将正确的副本数据进行复制以保证足够的副本数
- GFS上存储的数据块副本，在物理上以一个本地的Linux操作系统的文件形式存储，每一个数据块再划分为64KB的子块，每个子快有一个32位的校验和，读数据时会检查校验和以保证使用为失效的数据。

Google GFS的基本构架和工作原理

数据访问工作过程

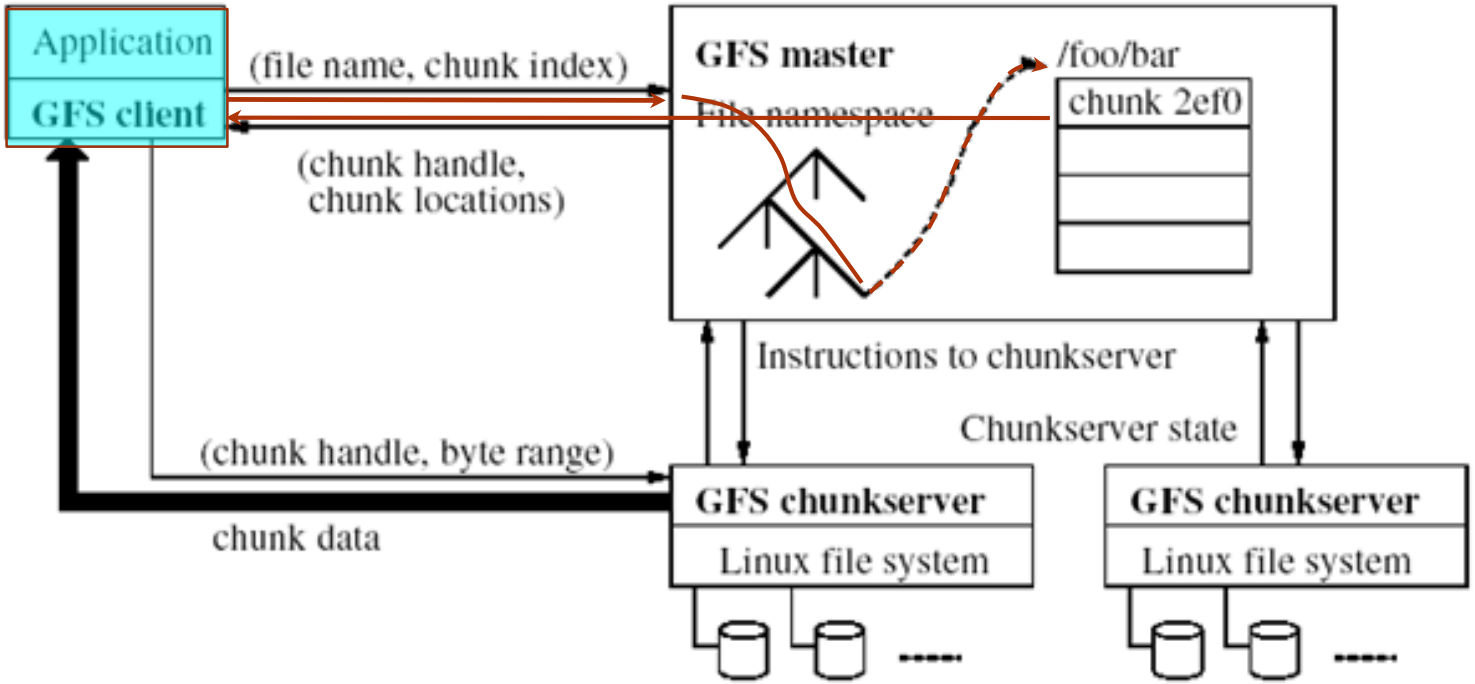
1. 在程序运行前，数据已经存储在GFS文件系统中；程序实行时应用程序会告诉GFS Server所要访问的文件名或者数据块索引是什么



Google GFS的基本构架和工作原理

数据访问工作过程

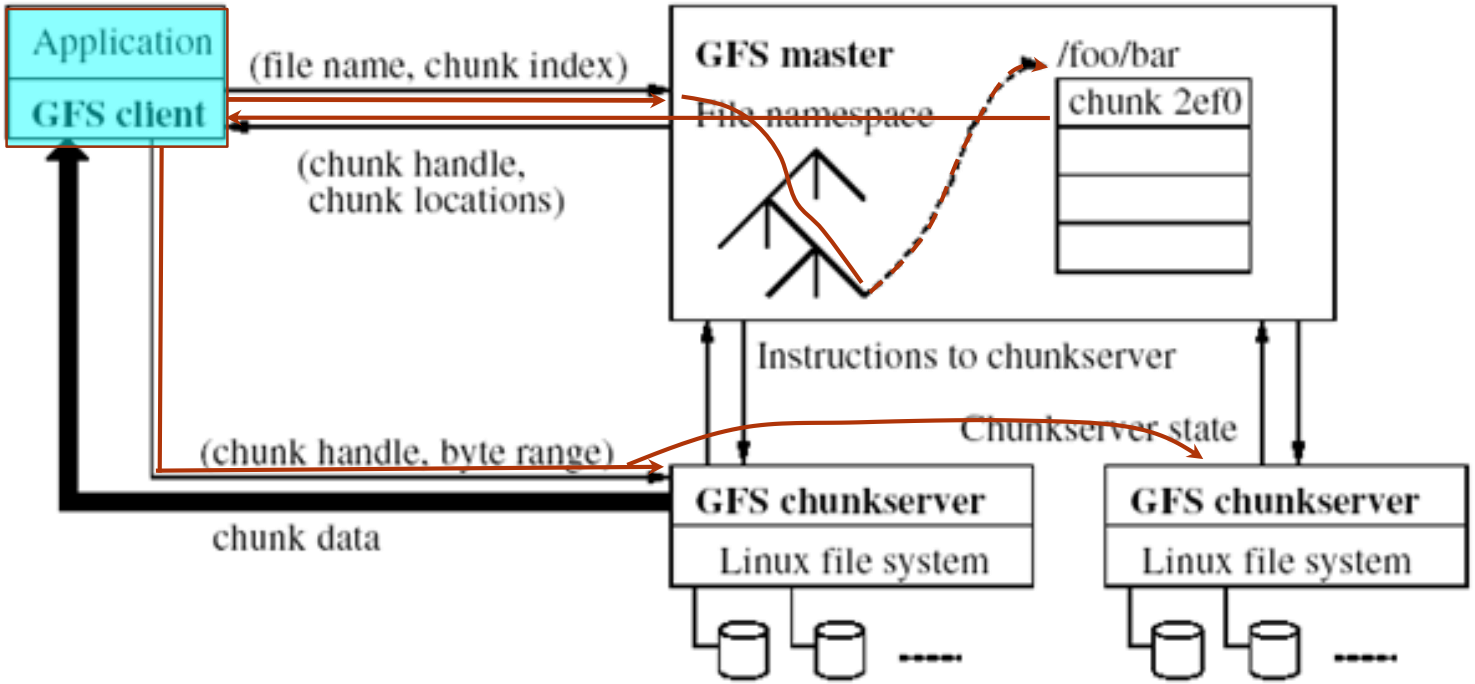
2. GFS Server根据文件名会数据块索引在其文件目录空间中查找和定位该文件或数据块，并找数据块在具体哪些ChunkServer上；将这些位置信息回送给应用程序



Google GFS的基本构架和工作原理

数据访问工作过程

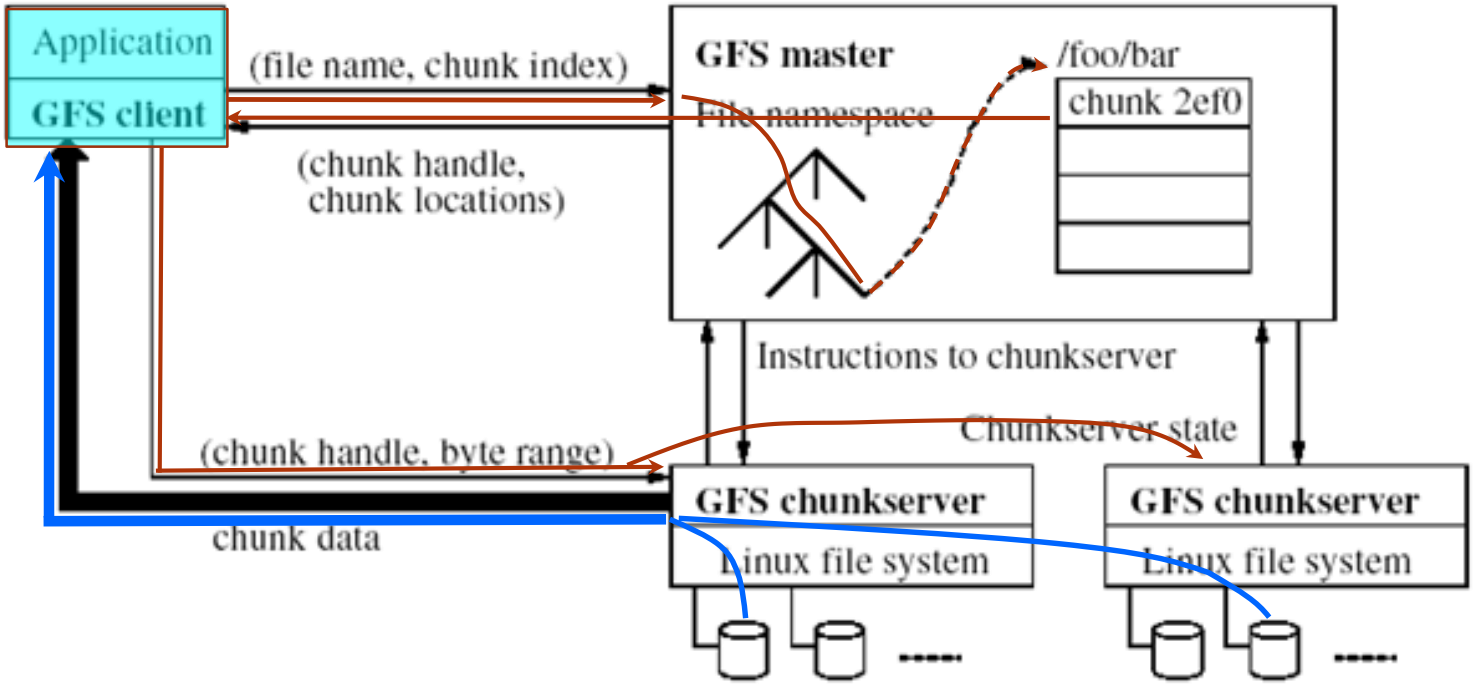
3. 应用程序根据GFSServer返回的具体Chunk数据块位置信息，直接访问相应的Chunk Server



Google GFS的基本构架和工作原理

数据访问工作过程

4. 应用程序根据GFSServer返回的具体Chunk数据块位置信息直接读取指定位置的数据进行计算处理

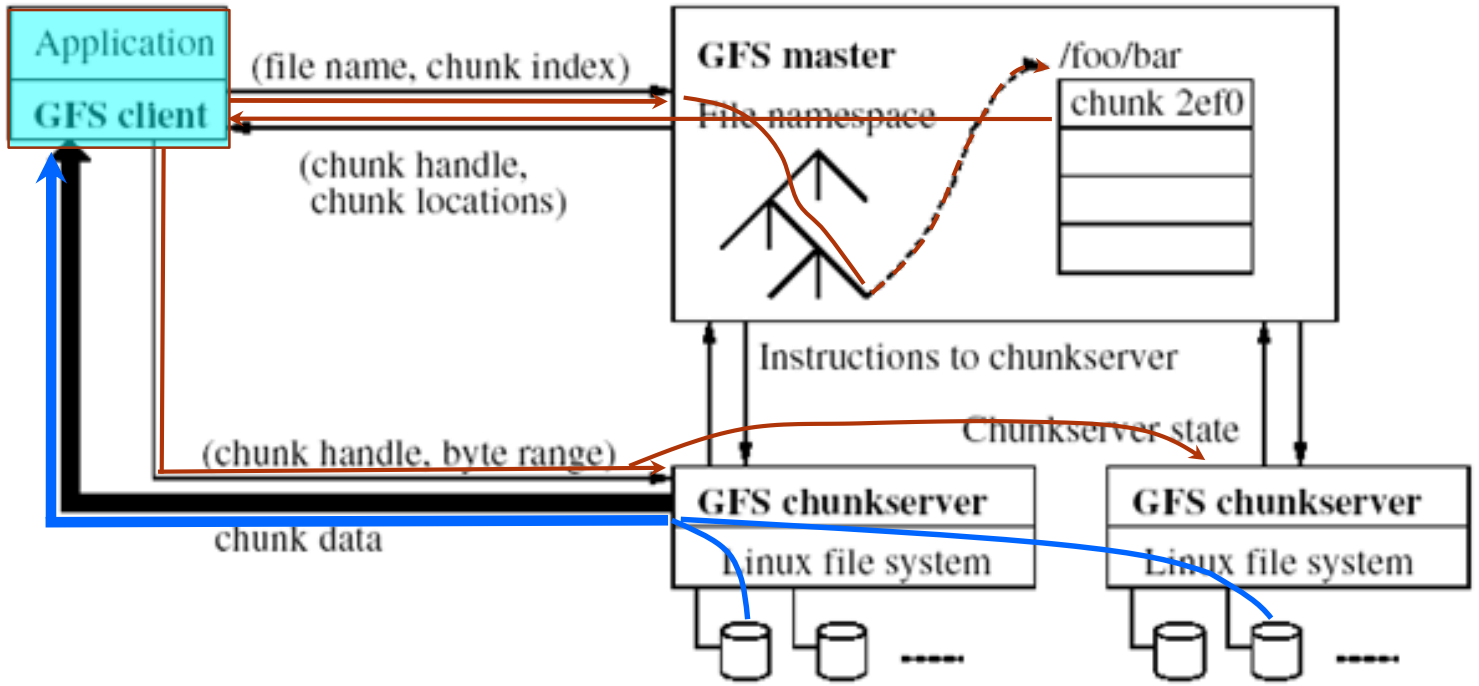


Google GFS的基本构架和工作原理

数据访问工作过程

特点：应用程序访问具体数据时不需要经过GFS Master，因此，避免了Master成为访问瓶颈

并发访问：由于一个大数据会存储在不同的ChunkServer中，应用程序可实现并发访问



Google GFS的基本构架和工作原理

GFS的系统管理技术

- 大规模集群安装技术：如何在一个成千上万个节点的集群上迅速部署GFS，升级管理和维护等
- 故障检测技术：GFS是构建在不可靠的廉价计算机之上的文件系统，节点数多，故障频繁，如何快速检测、定位、恢复或隔离故障节点
- 节点动态加入技术：当新的节点加入时，需要能自动安装和部署GFS
- 节能技术：服务器的耗电成本大于购买成本，Google为每个节点服务器配置了蓄电池替代UPS，大大节省了能耗。

4. 分布式结构化数据表BigTable

BigTable的基本作用和设计思想

- GFS是一个文件系统，难以提供对结构化数据的存储和访问管理。为此，Google在GFS之上又设计了一个结构化数据存储和访问管理系统—BigTable，为应用程序提供比单纯的文件系统更方便、更高层的数据操作能力
- Google的很多数据，包括Web索引、卫星图像数据、地图数据等都以结构化形式存放在BigTable中
- BigTable提供了一定粒度的结构化数据操作能力，主要解决一些大型媒体数据（Web文档、图片等）的结构化存储问题。但与传统的关系数据库相比，其结构化粒度没有那么高，也没有事务处理等能力，因此，它并不是真正意义上的数据库。

BigTable设计动机和目标

主要动机

- 需要存储多种数据

  Google提供的服务很多，需处理的数据类型也很多，如URL,网页,图片,地图数据,email,用户的个性化设置等

- 海量的服务请求

  Google是目前世界上最繁忙的系统之一，因此，需要有高性能的请求和数据处理能力

- 商用数据库无法适用

  在如此庞大的分布集群上难以有效部署商用数据库系统，且其难以承受如此巨量的数据存储和操作需求

BigTable设计动机和目标

主要设计目标

- 广泛的适用性:为一系列服务和应用而设计的数据存储系统,可满足对不同类型数据的存储和操作需求
- 很强的可扩展性:根据需要可随时自动加入或撤销服务器节点
- 高吞吐量数据访问:提供PB级数据存储能力，每秒数百万次的访问请求
- 高可用性和容错性:保证系统在各种情况下度能正常运转，服务不中断
- 自动管理能力：自动加入和撤销服务器，自动负载平衡
- 简单性：系统设计尽量简单以减少复杂性和出错率

BigTable数据模型

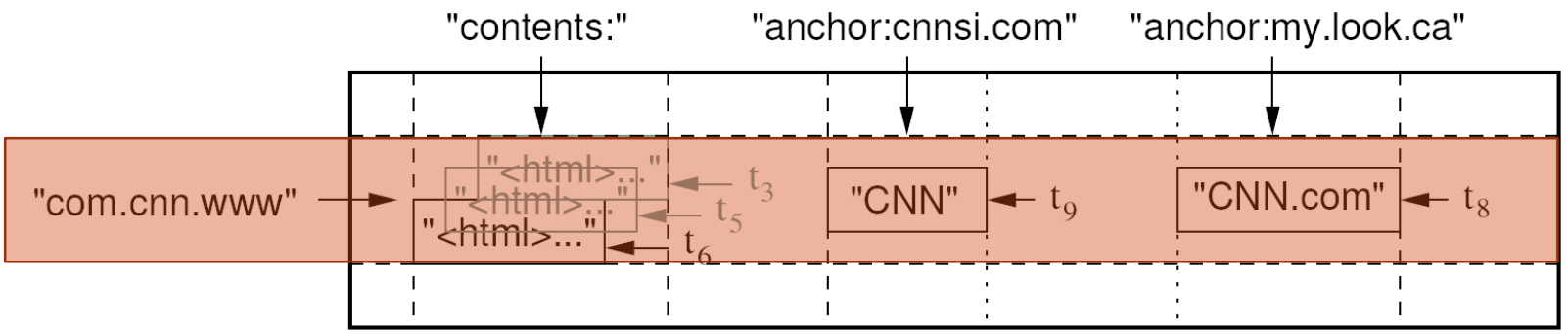
- BigTable主要是一个分布式多维表，表中的数据通过：
  - 一个行关键字(row key)
  - 一个列关键字(column key)
  - 一个时间戳(time stamp)

  进行索引和查询定位的。
- BigTable对存储在表中的数据不做任何解释，一律视为字节串，具体数据结构的实现有用户自行定义。
- BigTable查询模型

  (row:string, column:string,time:int64)→ 结果数据字节串
- 支持查询、插入和删除操作

BigTable数据模型

BigTable数据存储格式



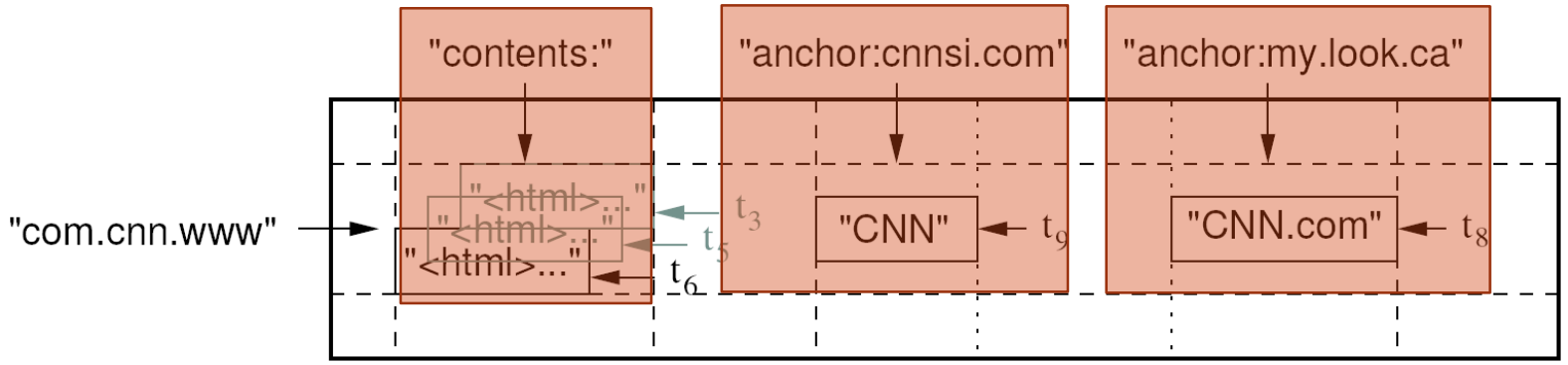
- 行(Row):大小不超过64KB的任意字节串。表中的数据都是根据行关键字进行排序的。

  com.cnn.www就是一个行关键字，指明一行存储数据。URL地址倒排好处是：1)同一地址的网页将被存储在表中连续的位置,便于查找；2)倒排便于数据压缩,可大幅提高数据压缩率

- 子表(Tablet)：一个大表可能太大，不利于存储管理，将在水平方向上被分为多个子表

BigTable数据模型

BigTable数据存储格式

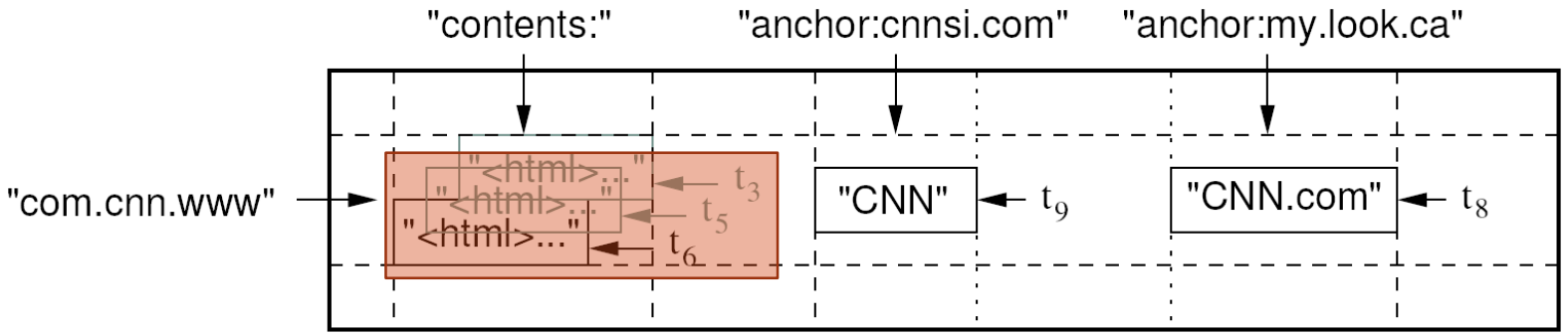


- 列(Column): BigTable将列关键字组织成为“列族”(column family),每个族中的数据属于同一类别,如anchor是一个列族，其下可有不同的表示一个个超链的列关键字。一个列族下的数据会被压缩在一起存放（按列存放）。因此,一个列关键字可表示为: 族名：列名(family:qualifier)

  content、anchor都是族名；而cnnsi.com和my.look.ca则是anchor族中的列名。

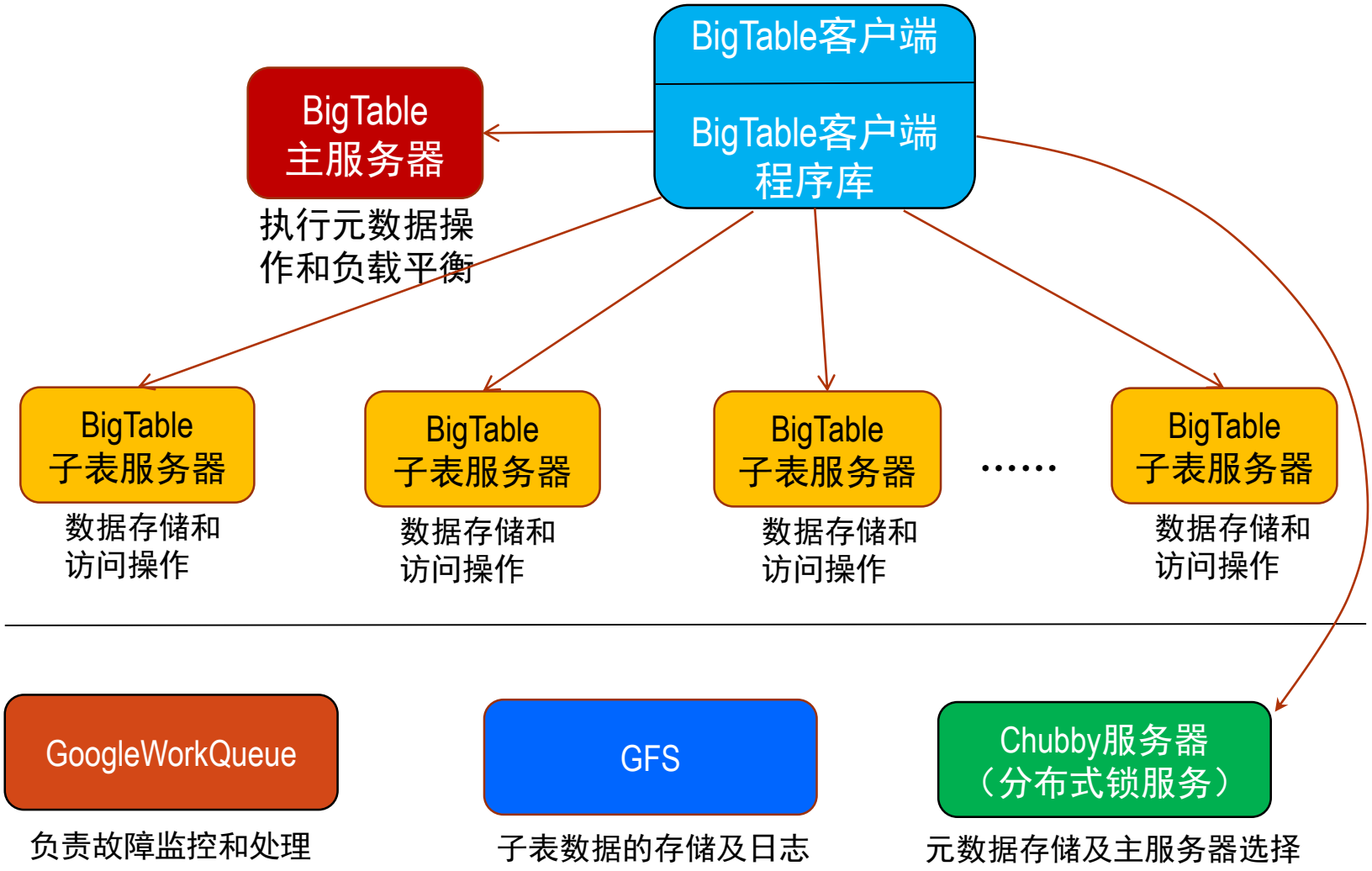
BigTable数据模型

BigTable数据存储格式



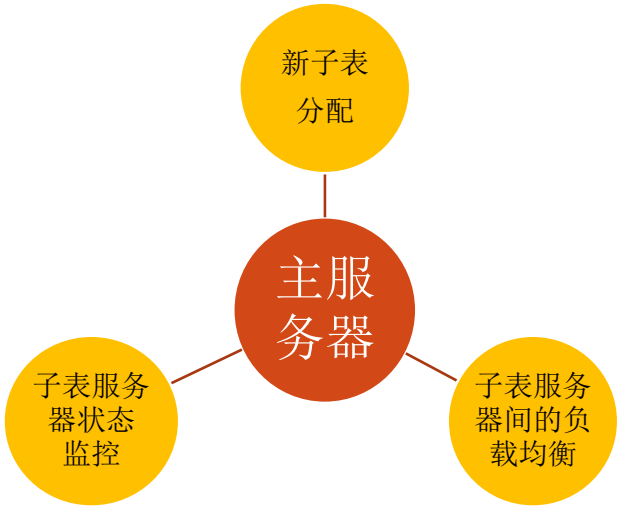
- 时间戳(time stamp): 很多时候同一个URL的网页会不断更新，而Google需要保存不同时间的网页数据，因此需要使用时间戳来加以区分。
- 为了简化不同版本的数据管理，BigTable提供给了两种设置：
  - 保留最近的n个版本数据
  - 保留限定时间内的所有不同版本数据

BigTable基本构架



BigTable基本构架

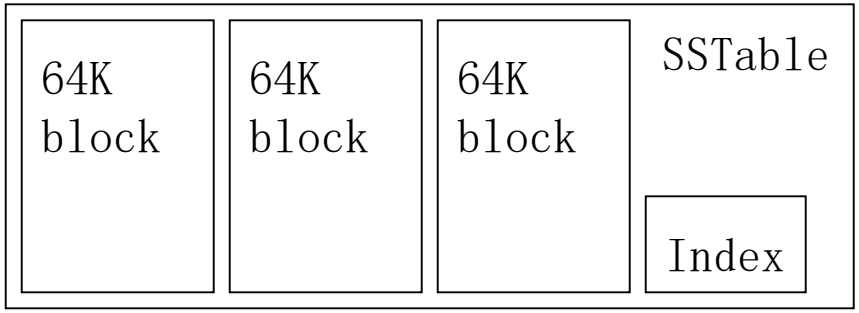
主服务器



- 新子表分配：当一个新子表产生时，主服务器通过加载命令将其分配给一个空间足够大的子表服务器；创建新表、表合并及较大子表的分裂都会产生新的子表。
- 子表监控：通过Chubby完成。所有子表服务器基本信息被保存在Chubby中的服务器目录中主服务器检测这个目录可获取最新子表服务器的状态信息。当子表服务器出现故障，主服务器将终止该子表服务器，并将其上的全部子表数据移动到其它子表服务器。
- 负载均衡：当主服务器发现某个子表服务器负载过重时，将对自动对其进行负载均衡操作。

BigTable基本构架

子表服务器



BigTable中的数据都以子表形式保存在子表服务器上，客户端程序也直接和子表服务器通信。

分配：当一个新子表产生时子表服务器的主要问题包括子表的定位、分配、及子表数据的最终存储。

- 子表的基本存储结构SSTable

  SSTable是BigTable内部的基本存储结构，以GFS文件形式存储在GFS文件系统中。一个SSTable实际上对应于GFS中的一个64MB的数据块(Chunk)

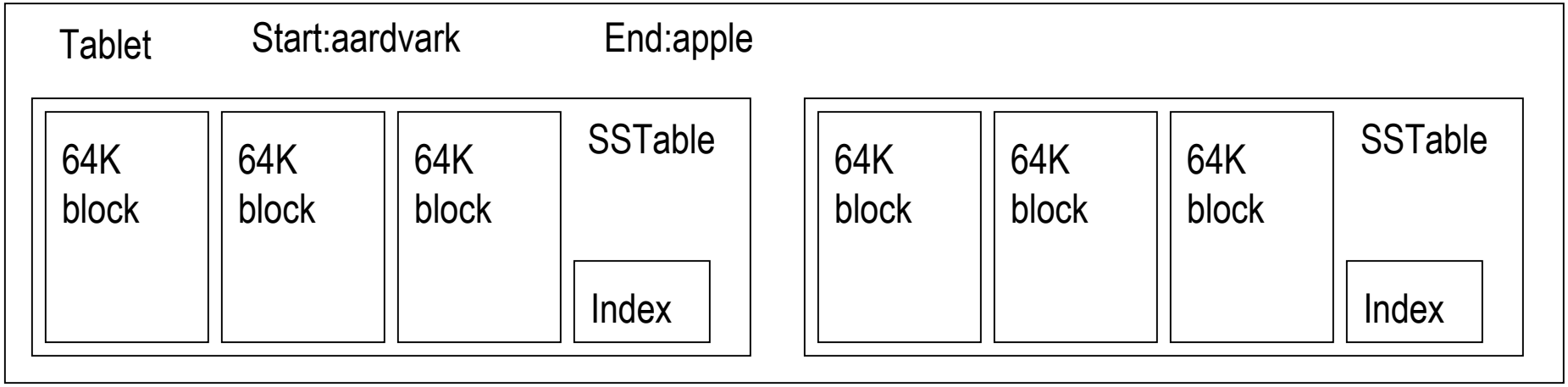
  SSTable中的数据进一步划分为64KB的子块，因此一个SSTable可以有多达1千个这样的子块。为了维护这些子块的位置信息，需要使用一个Index索引。

BigTable基本构架

子表服务器

- 子表数据格式

概念上子表是整个大表的多行数据划分后构成。而一个子表服务器上的子表将进一步由很多个SSTable构成，每个SSTable构成最终的在底层GFS中的存储单位。

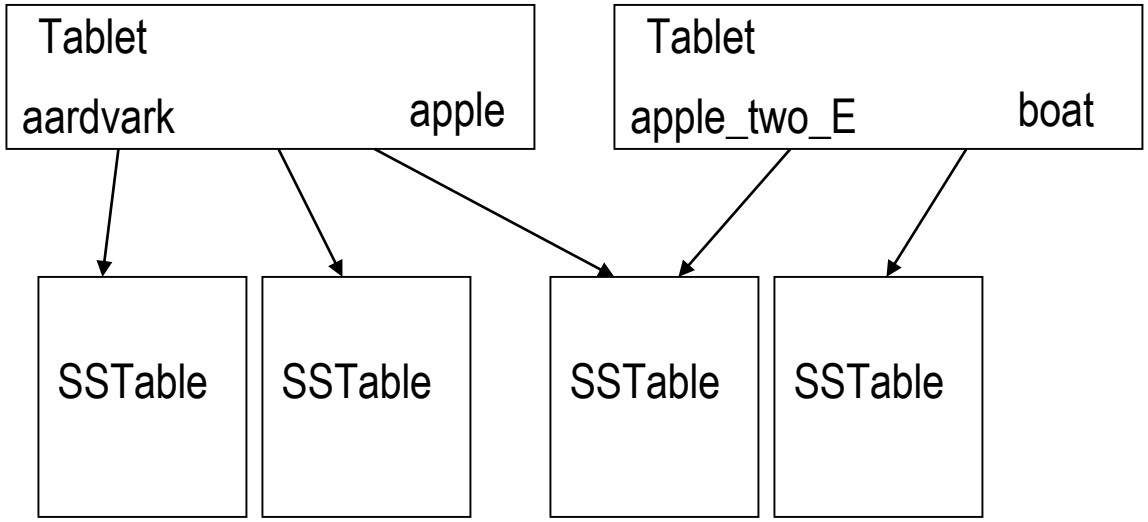


BigTable基本构架

子表服务器

- 子表数据格式

一个SSTable还可以为不同的子表所共享，以避免同样数据的重复存储。

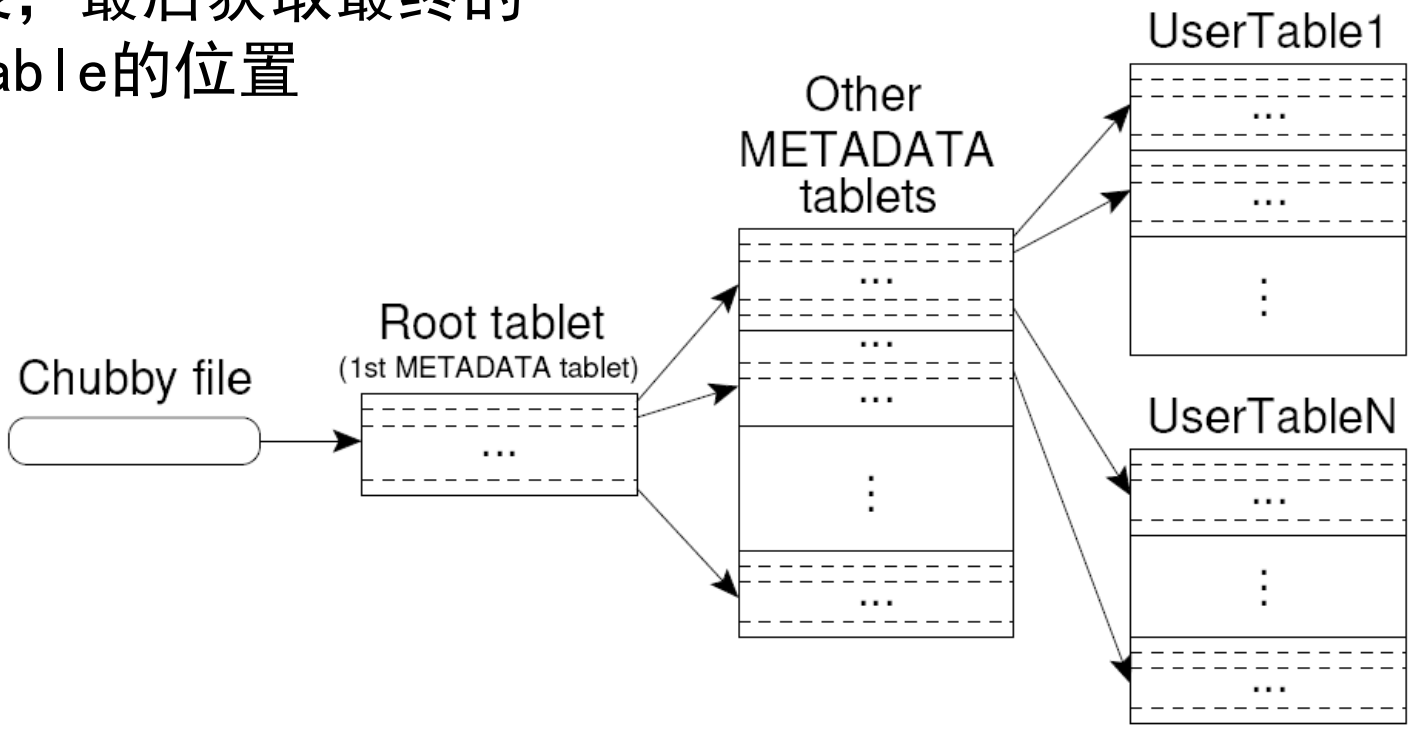


BigTable基本构架

子表服务器

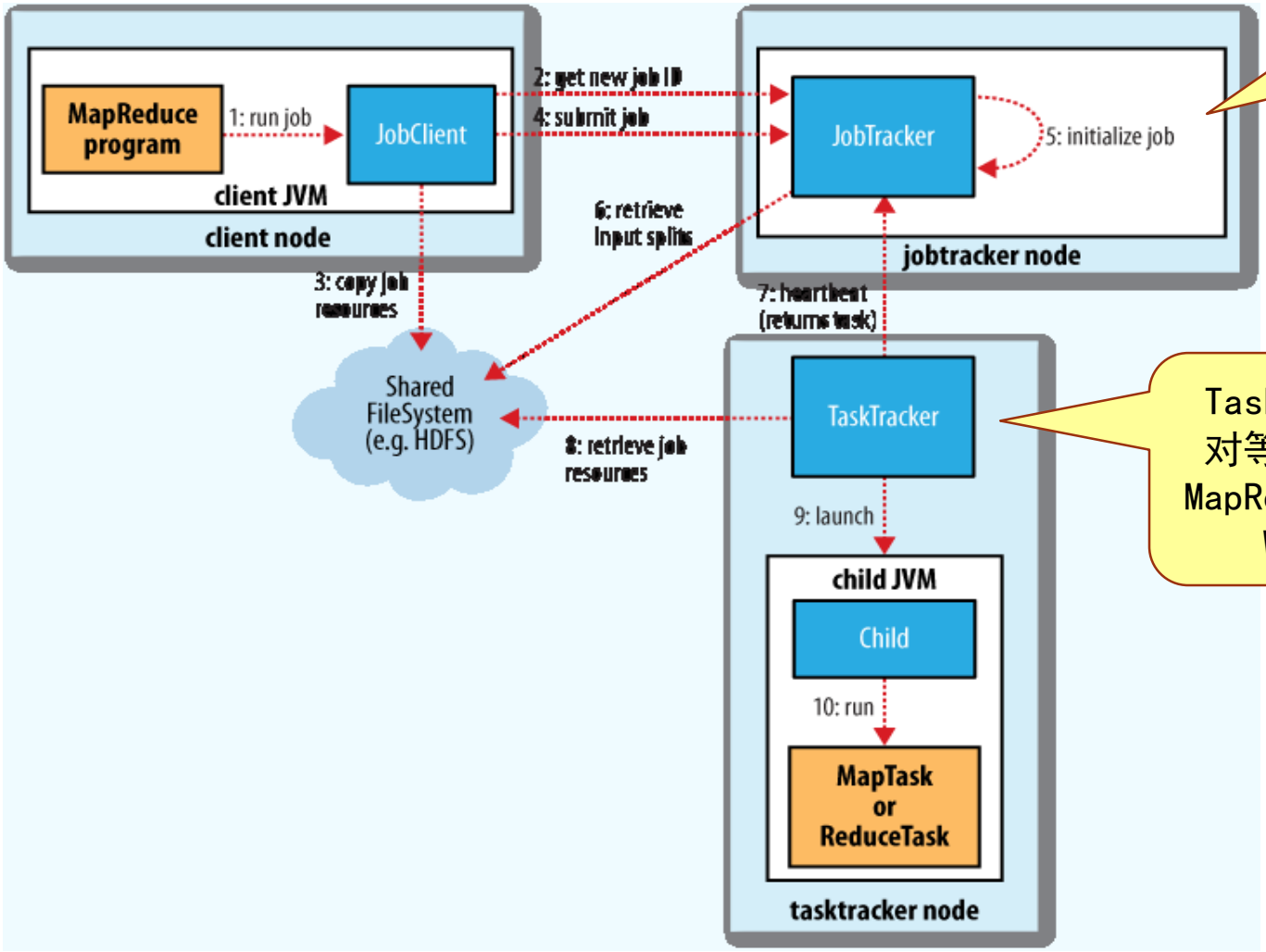
- 子表寻址

  子表地址以3级B+树形式进行索引；首先从Chubby服务器中取得根子表，由根子表找到二级索引子表，最后获取最终的SSTable的位置



5. Hadoop MapReduce的基本工作原理

Hadoop MapReduce基本构架与工作过程

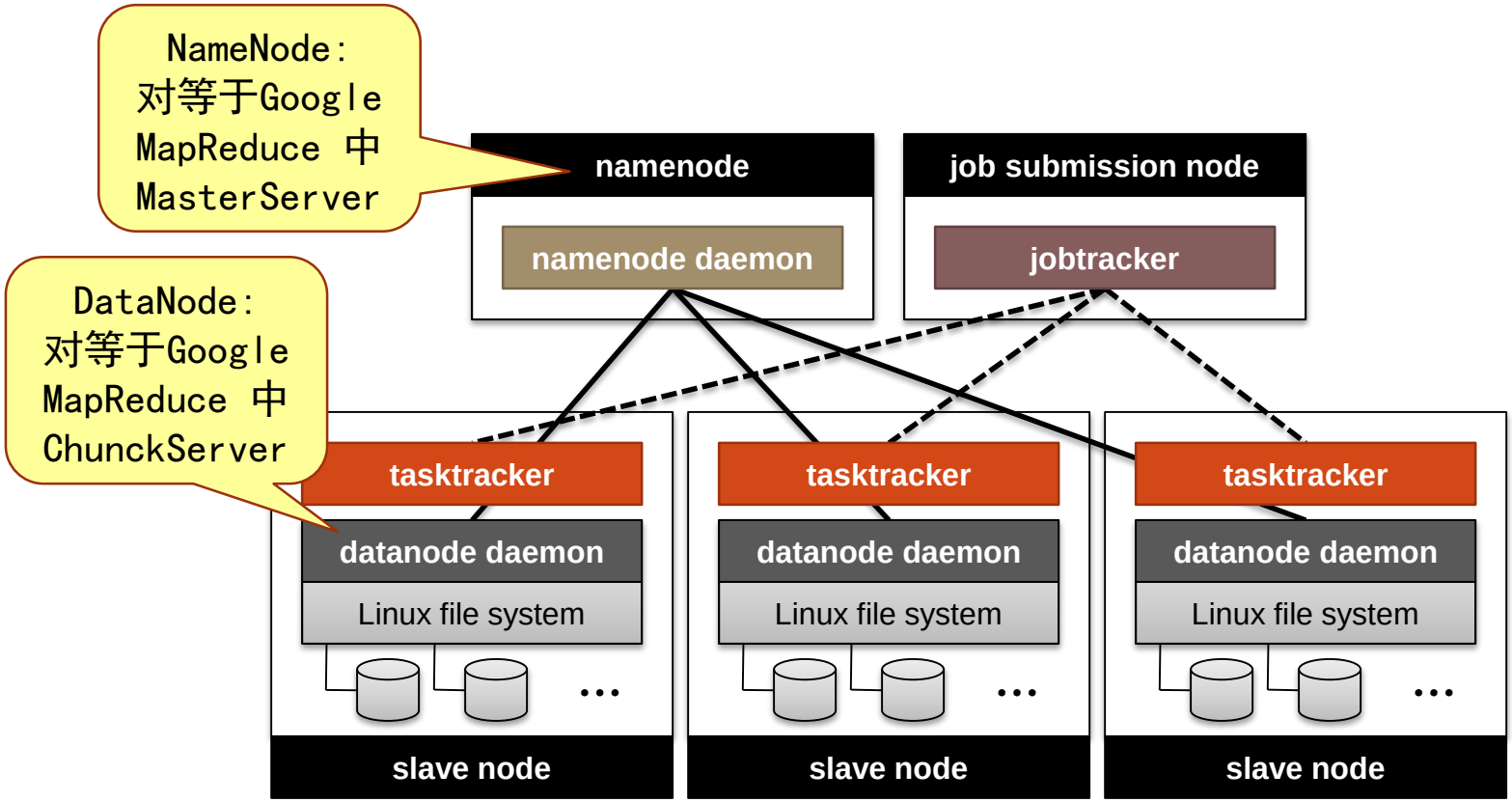


JobTracker：对等于Google MapReduce 中的Master

TaskTracker：对等于Google MapReduce 中的Worker

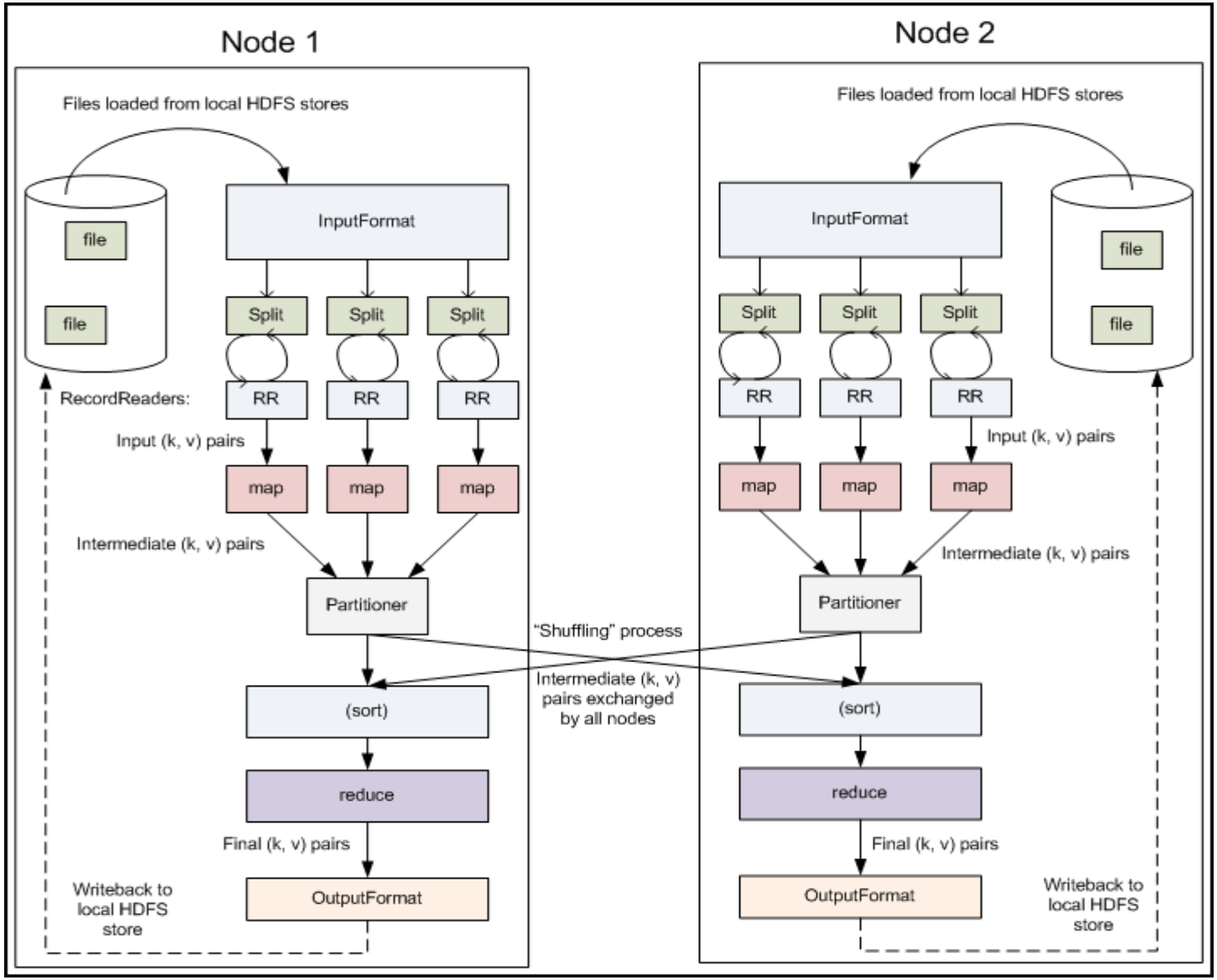
Hadoop MapReduce基本构架与工作过程

数据存储与计算节点构架



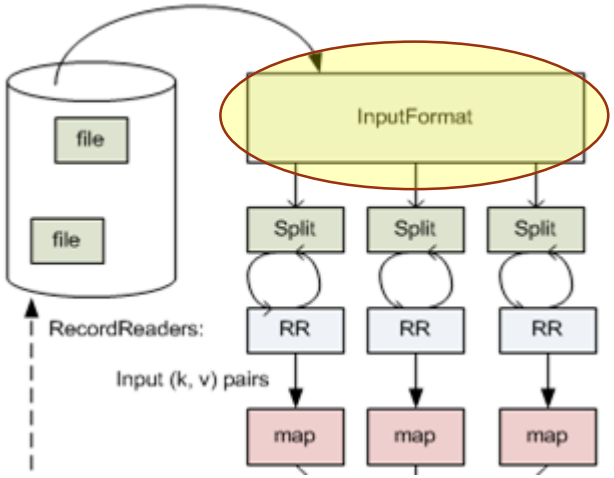
Hadoop MapReduce的基本工作原理

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

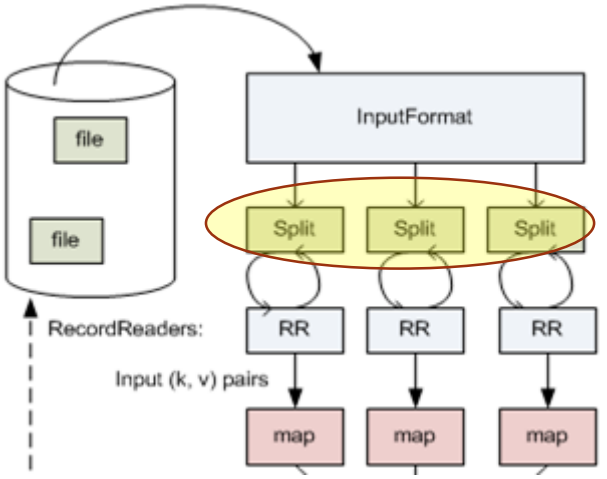
Hadoop MapReduce主要组件

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
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

Hadoop MapReduce主要组件

输入数据分块InputSplits

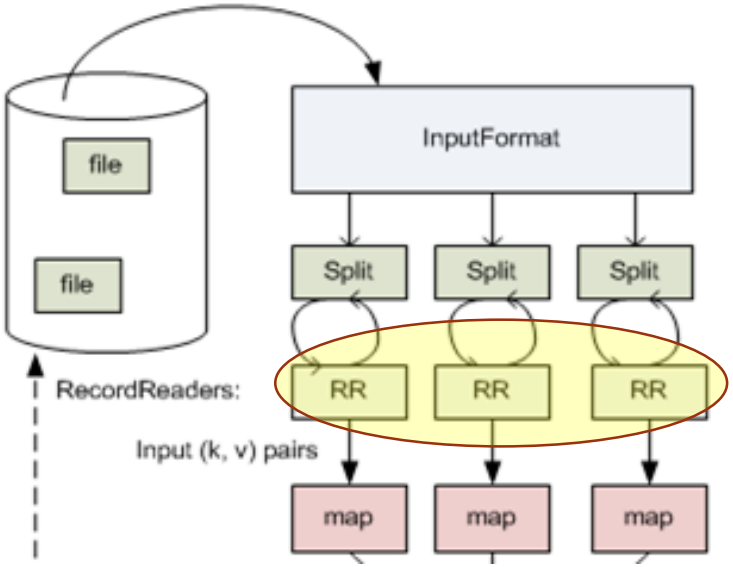


- InputSplit定义了输入到单个Map任务的输入数据
- 一个MapReduce程序被统称为一个Job，可能有上百个任务构成
- InputSplit将文件分为64MB的大小
  - 配置文件hadoop-site.xml中的mapred.min.split.size参数控制这个大小
- mapred.tasktracker.map.taks.maximum用来控制某一个节点上所有map任务的最大数目

Hadoop MapReduce主要组件

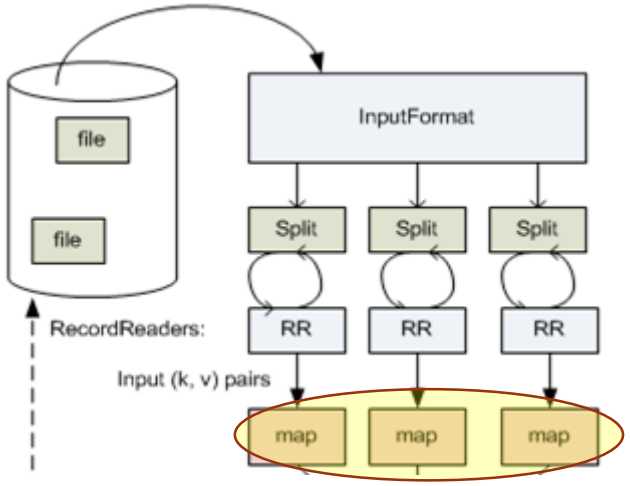
数据记录读入RecordReader

- 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

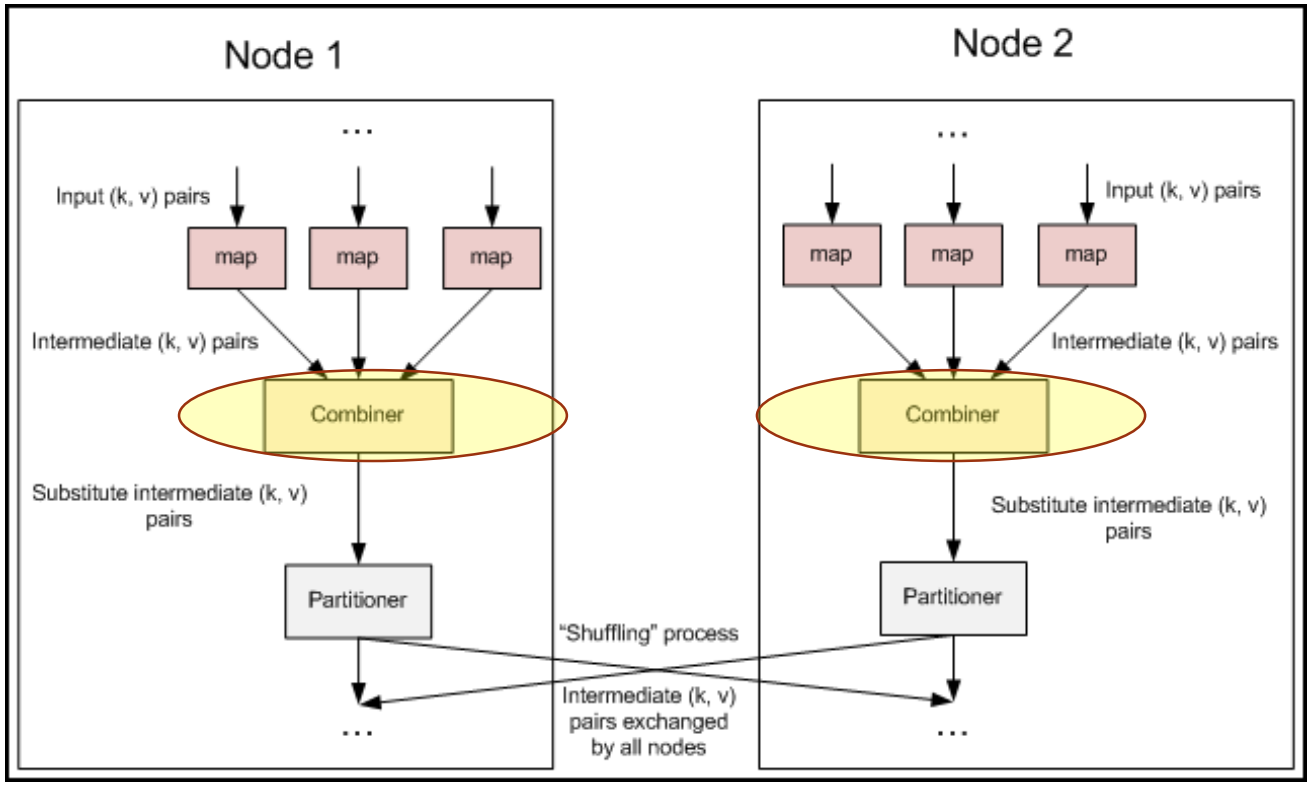
Mapper



- 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
- 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。
- 现在的版本用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

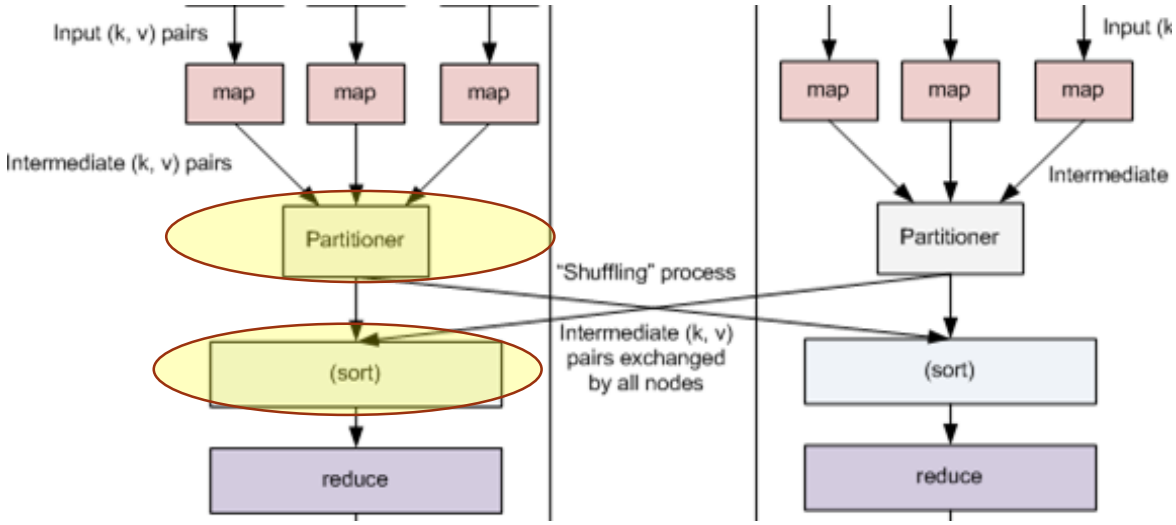


- 合并相同key的键值对，减少partitioning时候的数据通信开销
- conf.setCombinerClass(Reduce.class);
- 是在本地执行的一个Reducer，满足一定的条件才能够执行。

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



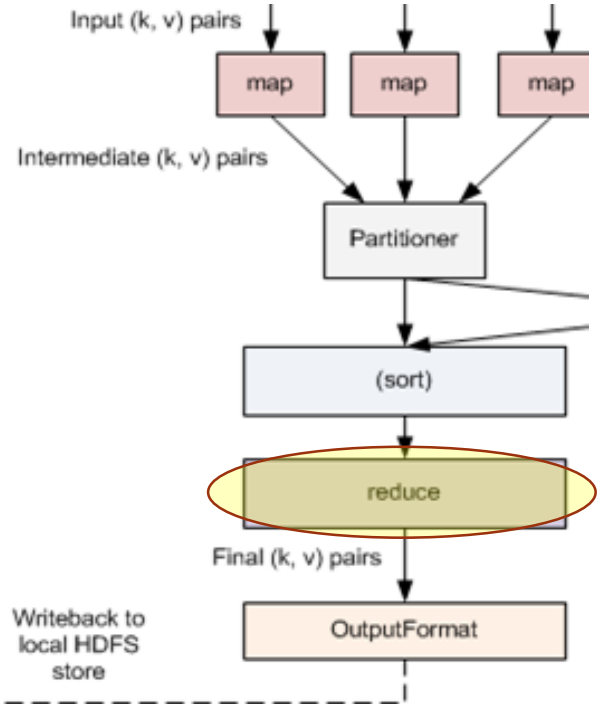
Sort

- 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主要组件

Reducer

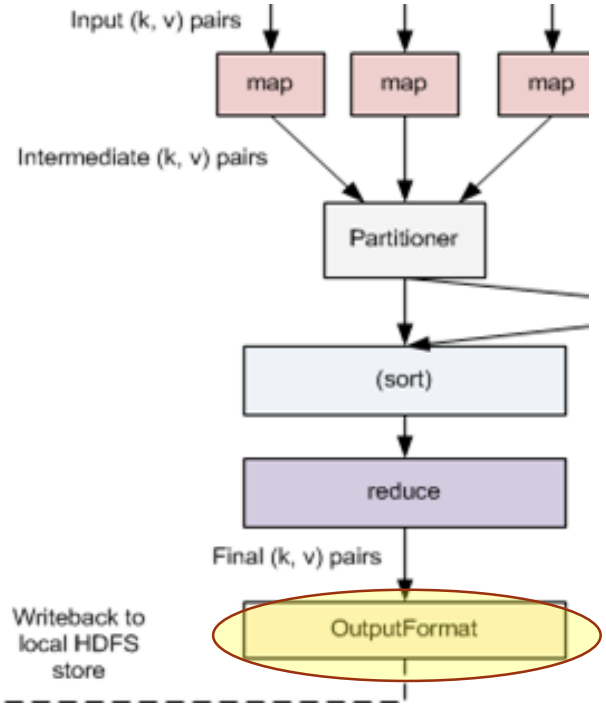
- 做用户定义的Reduce操作
- 接收到一个OutputCollector的类作为输出
- 新版本的编程接口是Reducer.Context



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）
- mapreduce.map.speculative
- mapreduce.reduce.speculative

6. Hadoop的分布式文件系统HDFS

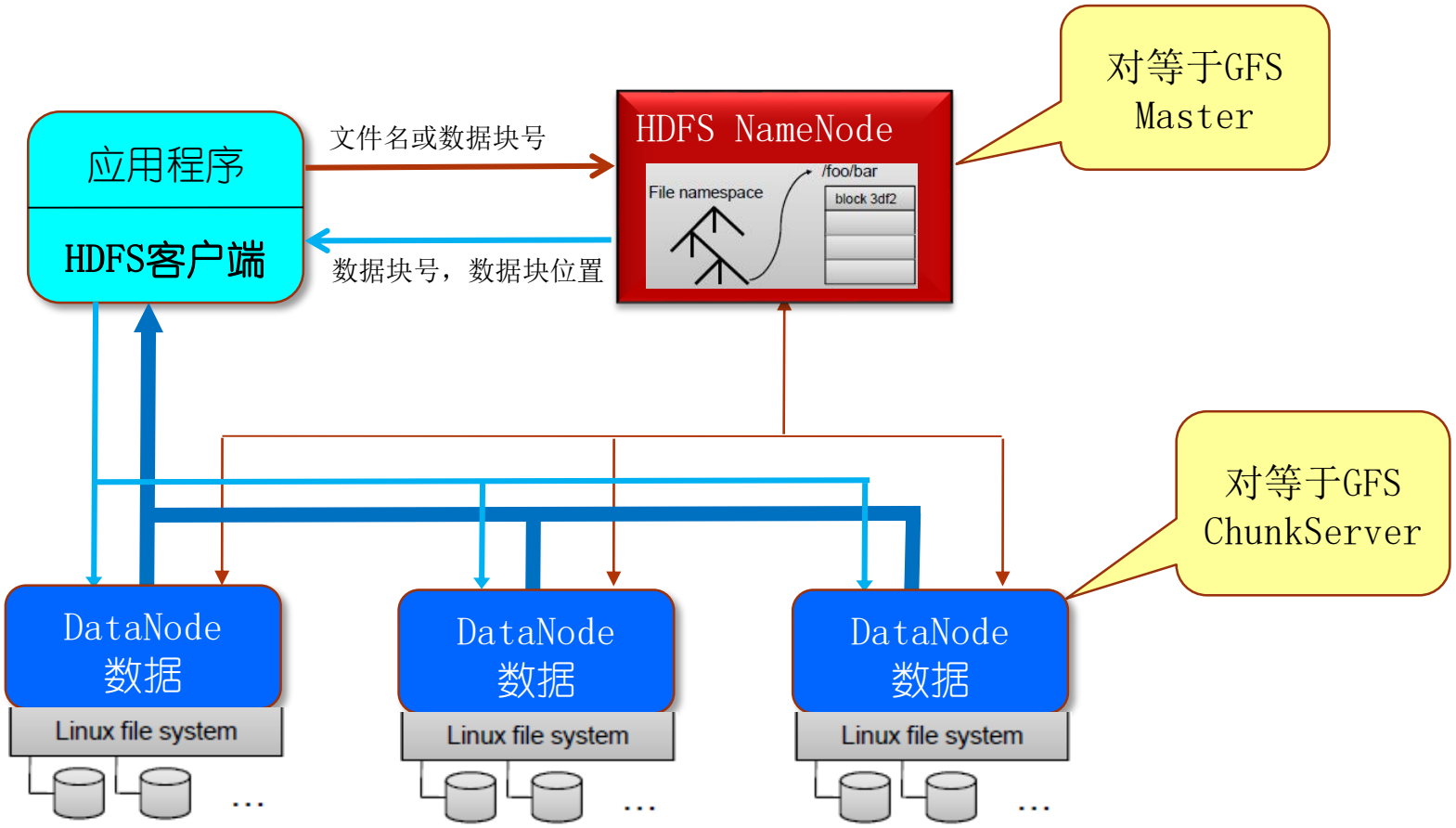
HDFS的基本特征

- 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。

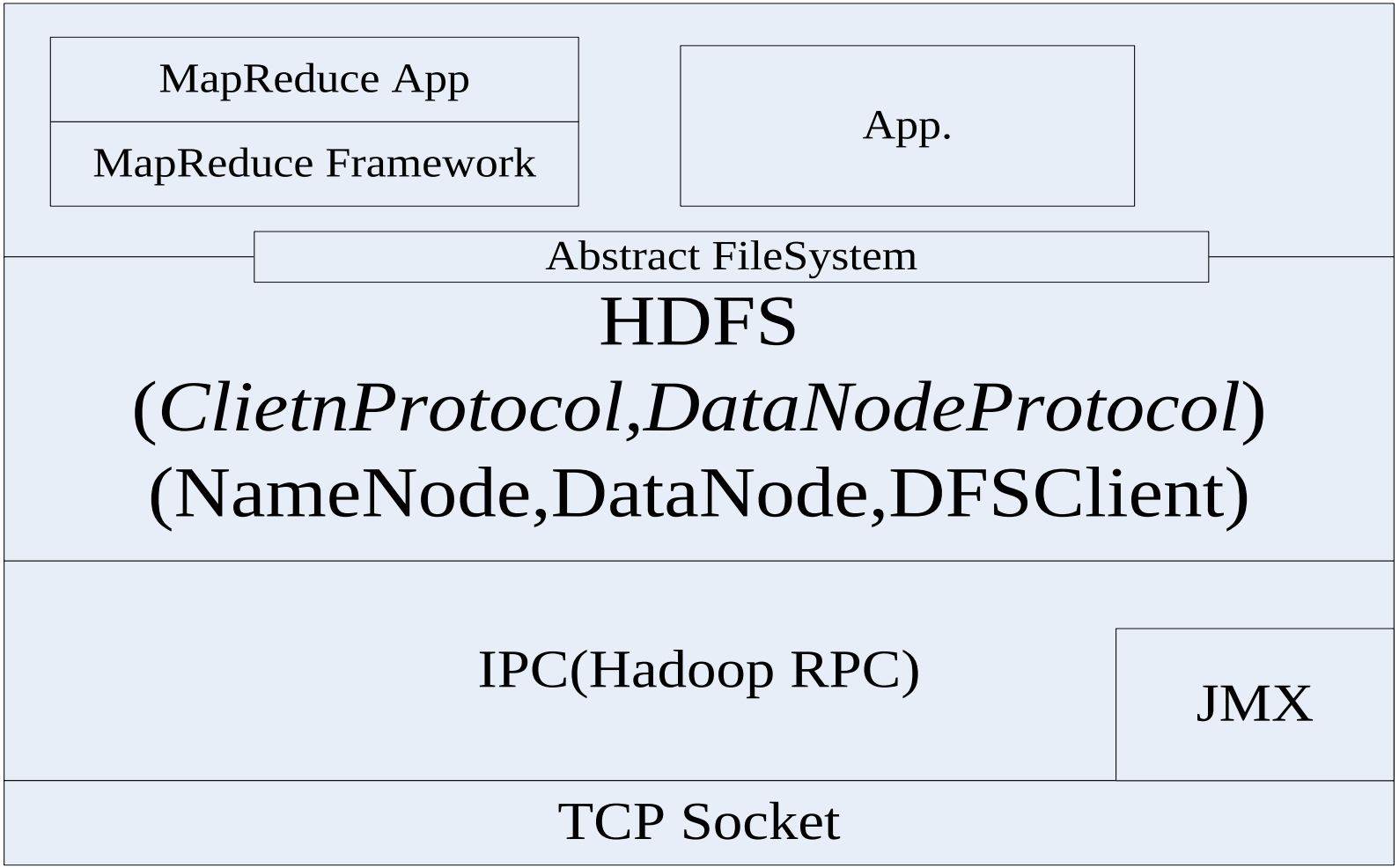
HDFS的基本特征

- 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

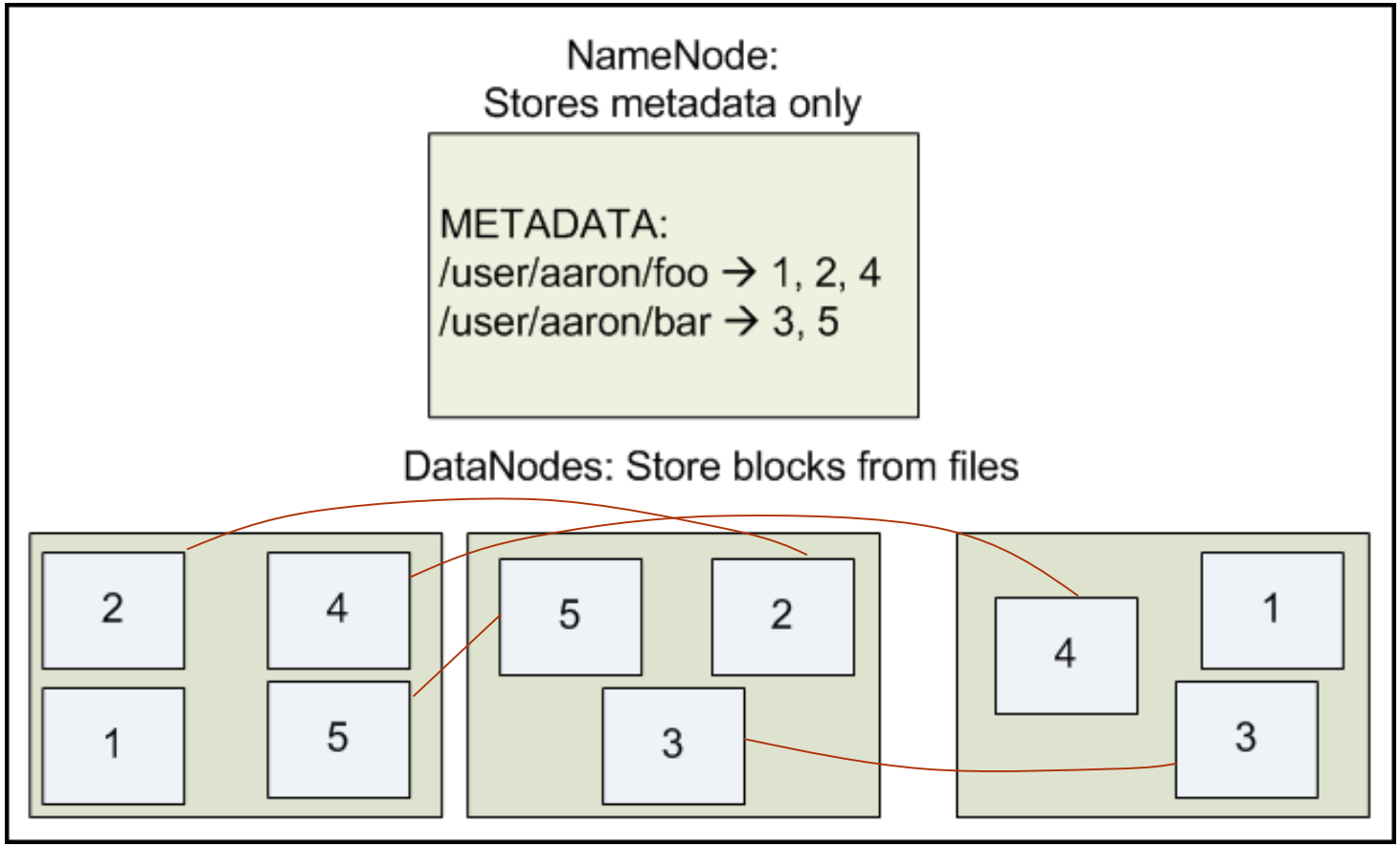
HDFS基本构架


应用程序
HDFS客户端
文件名或数据块号
数据块号，数据块位置
HDFS NameNode
File namespace
/foo/bar
block 3df2
对等于GFS Master
DataNode 数据
Linux file system
DataNode 数据
Linux file system
DataNode 数据
Linux file system
对等于GFS ChunkServer

HDFS基本实现构架


MapReduce App
MapReduce Framework
App.
Abstract FileSystem
HDFS
(ClietnProtocol,DataNodeProtocol)
(NameNode,DataNode,DFSClient)
IPC(Hadoop RPC)
JMX
TCP Socket

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

HDFS文件系统操作命令

- 建立用户自己的目录，用户目录在/user中，需要建立

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

Put上传整个目录

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

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

```
someone@anynode:hadoop$ bin/hadoop dfs -ls / Found 2 items
drwxr-xr-x - hadoop supergroup 0 2016-09-20 19:40 /hadoop
drwxr-xr-x - hadoop supergroup 0 2016-09-20 20:08 /tmp
```

HDFS文件系统操作命令

Command:	Assuming:	Outcome:
bin/hdfs dfs -put foo bar	No file/directory named /user/$USER/bar exists in HDFS	Uploads local file foo to a file named /user/$USER/bar
bin/hdfs dfs -put foo bar	/user/$USER/bar is a directory	Uploads local file foo to a file named /user/$USER/bar/foo
bin/hdfs dfs -put foo1 foo2 somedir	/user/$USER/somedir is a directory in HDFS	Uploads local file foo1, foo2 to a directory named
bin/hdfs dfs -put foo bar	/user/$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

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

HDFS文件系统操作命令

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

HDFS文件系统操作命令

-get [-crc] src localDest	Copies the file or directory in HDFS identified by src to the local file system path identified by localDest.
-getmerge src localDest [addnl]	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.
-cat filename	Displays the contents of filename on stdout.
-copyToLocal [-crc] src localDest	Identical to -get
-moveToLocal [-crc] src localDest	Works like -get, but deletes the HDFS copy on success.
-mkdir path	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

-setrep [-R] [-w] rep path	Sets the target replication factor for files identified by path to rep. (The actual replication factor will move toward the target over time)
-touchz path	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.
-test -[defsz] path	Returns 0 if path is a directory; exists; is a file; is not empty; or the file is zero length.
-stat [format] path	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).
-tail [-f] file	Shows the last 1KB of file on stdout.

HDFS文件系统操作命令

-chmod [-R] mode,mode,... path...	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.
-chown [-R] [owner][:[group]] path...	Sets the owning user and/or group for files or directories identified by path.... Sets owner recursively if -R is specified.
-chgrp [-R] group path...	Sets the owning group for files or directories identified by path.... Sets group recursively if -R is specified.
-help cmd	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/hdfs dfsadmin –report
- bin/hdfs 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/hdfs dfsadmin –safemode enter/leave/get/wait

HDFS Admin命令

- 更改HDFS成员
- 升级HDFS版本

  sbin/start-dfs.sh –upgrade(第一次运行新版本的时候使用)
  bin/hdfs dfsadmin –rollingUpgrade query
  bin/hdfs dfsadmin –rollingUpgrade prepare
  bin/hdfs dfsadmin –rollingUpgrade finalize (on your own risk!)
  sbin/start-dfs.sh –rollback(在旧版本重新安装后使用)(on your own risk!)
- 帮助 bin/hdfs 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
- sbin/start-balancer.sh –threshold 5
- sbin/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.defaultFS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/hdfs 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 sbin/start-all.sh or sbin/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.superusergroup

7.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();

FileSystem hdfs = 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。

7. Hadoop HDFS的编程

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

7. Hadoop HDFS的编程

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。

7. Hadoop HDFS的编程

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）

7. Hadoop HDFS的编程

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。

7. Hadoop HDFS的编程

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。

7. Hadoop HDFS的编程

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

7. Hadoop HDFS的编程

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。

7. 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、循环读取数据、写入目标位置，然后关闭文件，最后关闭输出文件。

```
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]);
```

7. 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].isDirectory() == 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 !