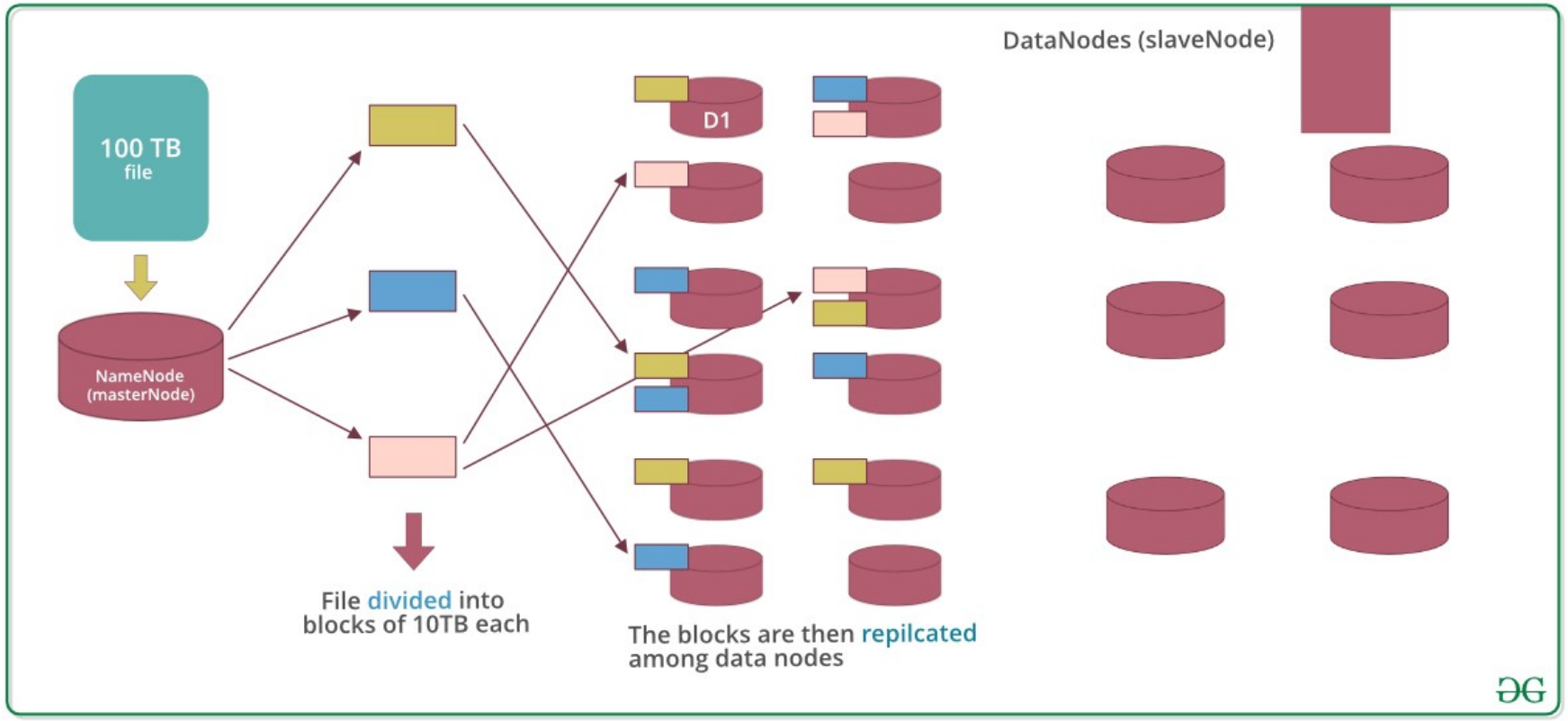
什么是HDFS

- HDFS 是一种分布式文件系统，用于处理在商业硬件上运行的大型数据集。
- 它用于将单个 Apache Hadoop 集群扩展到数百 甚至数千个节点。
- HDFS 是 Apache Hadoop 的主要组件之一，其他组件包括 MapReduce 和 YARN。
- HDFS是用Java编写的，任何支持Java的机器都可以部署HDFS。

HDFS示例



HDFS目标（1/3）

◆硬件故障

- 硬件故障是常态，而不是异常。整个HDFS系统将由数百或数千个存储着文件数据片段的服务器组成。实际上它里面有非常巨大的组成部分，每一个组成部分都很可能出现故障，这就意味着HDFS里的总是有一些部件是失效的，因此，故障的检测和自动快速恢复是HDFS一个很核心的设计目标。

◆流式数据访问

- 运行在HDFS之上的应用程序必须流式地访问数据集，这些不是运行在普通文件系统之上的普通应用。HDFS被设计成适合批量处理的，而不是用户交互式的。重点是在数据吞吐量，而不是数据访问的反应时间，POSIX（可移植操作系统接口）的很多硬性需求对于HDFS应用都是非必须的，去掉POSIX一小部分关键语义可以获得更好的数据吞吐率。

HDFS目标(2/3)

- 大数据集
  - 运行在HDFS之上的程序有很大量的数据集。典型的HDFS文件大小是GB到TB的级别。所以，HDFS被调整成支持大文件。它应该提供很高的聚合数据带宽，一个集群中支持数百个节点，一个集群中还应该支持千万级别的文件。
- 简单一致性模型
  - 大部分的HDFS程序对文件操作需要的是一次写多次读的操作模式。一个文件一旦创建、写入、关闭之后就不需要修改了。这个假定简单化了数据一致的问题，并使高吞吐量的数据访问变得可能。一个Map-Reduce程序或者网络爬虫程序都可以完美地适合这个模型。

HDFS目标(3/3)

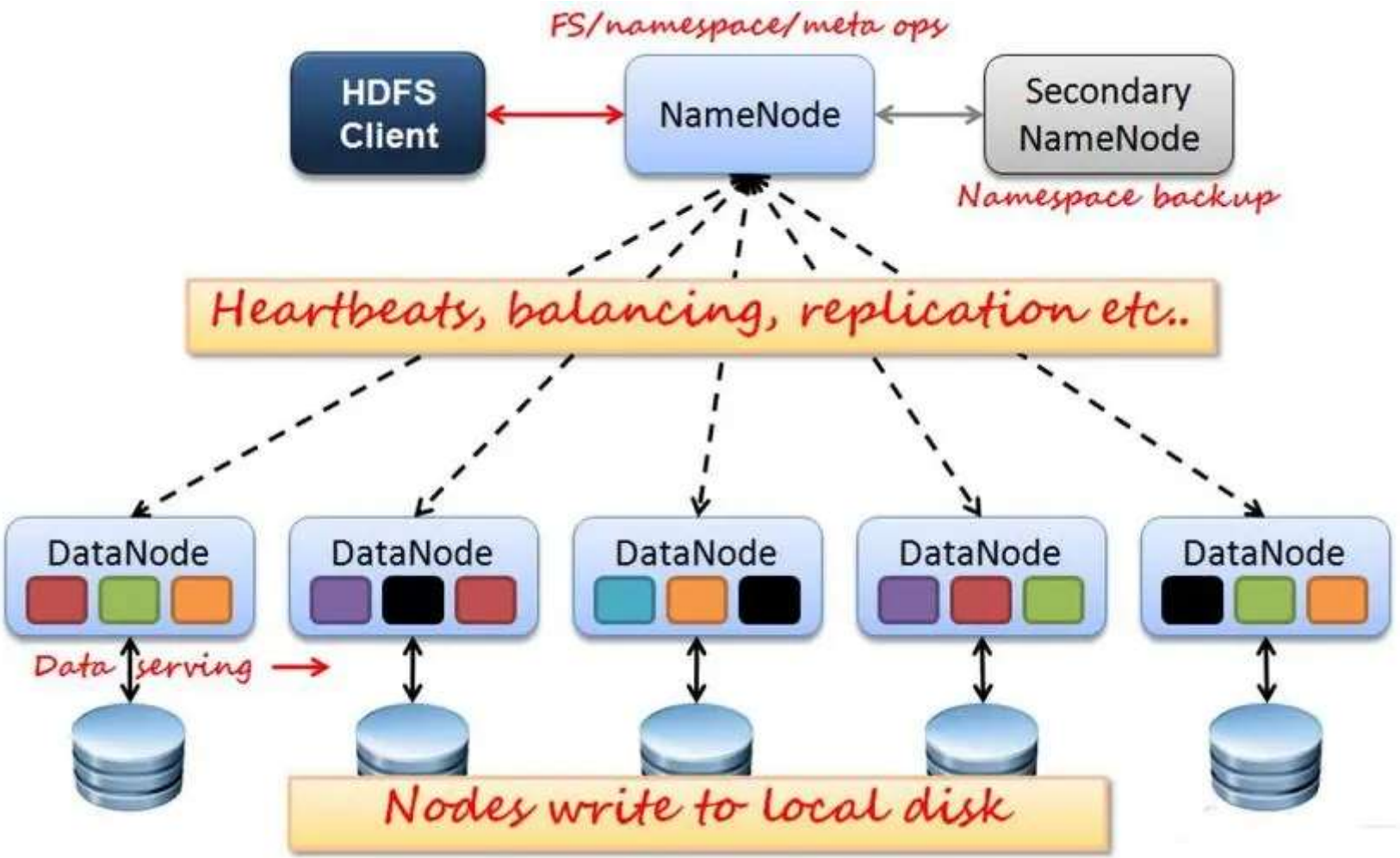
◆ 移动计算比移动数据更经济

- 在靠近计算数据所存储的位置进行计算是最理想的状态，尤其是在大数据集。它会消除网络拥堵，提高系统的整体吞吐量。
- HDFS提供了接口，来让程序将自己移动到离数据存储更近的位置。

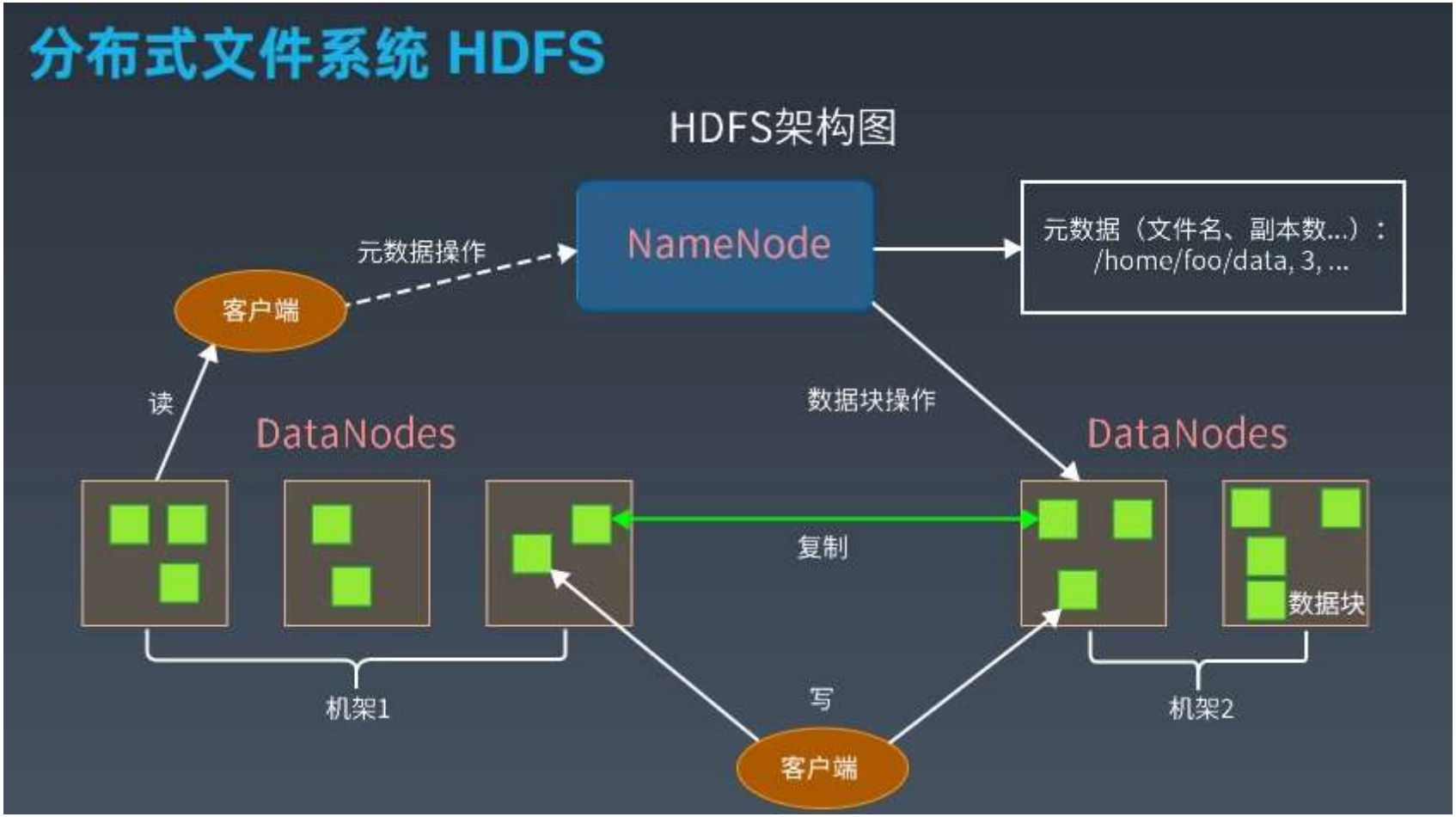
◆ 异构软硬件平台间的可移植性

- HDFS被设计成可以简便地实现平台间的迁移，这将推动需要大数据集的应用更广泛地采用HDFS作为平台。

HDFS体系结构


FS/namespace/meta ops
HDFS Client
NameNode
Secondary NameNode
Namespace backup
Heartbeats, balancing, replication etc..
DataNode
DataNode
DataNode
DataNode
DataNode
Data serving
Nodes write to local disk

名字节点与数据节点



北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

数据组织-数据块

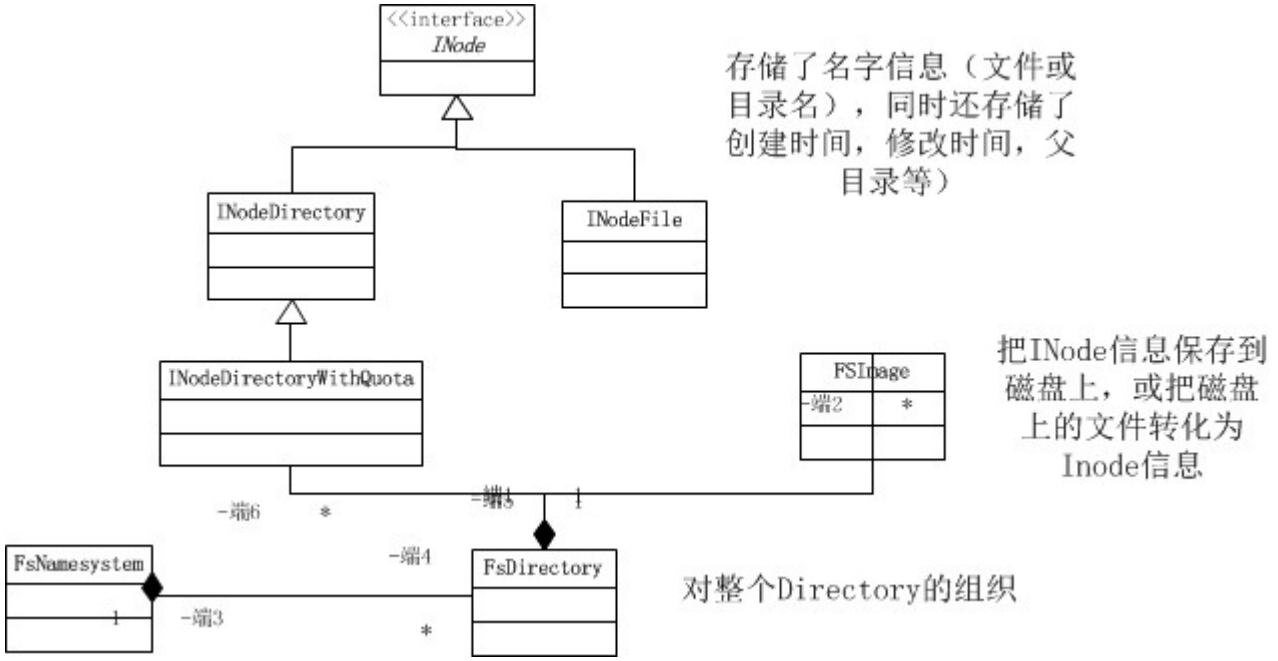
- HDFS的设计是用于支持大文件的。运行在HDFS上的程序也是用于处理大数据集的。
- 这些程序一次写多次读数据，并且这些读操作要求满足流式传输速度。
- HDFS中典型的块大小是128MB，一个HDFS文件可以被切分成多个128MB大小的块(chunk)，如果需要，每一个块可以分布在不同的数据节点上。

块副本（Replication）

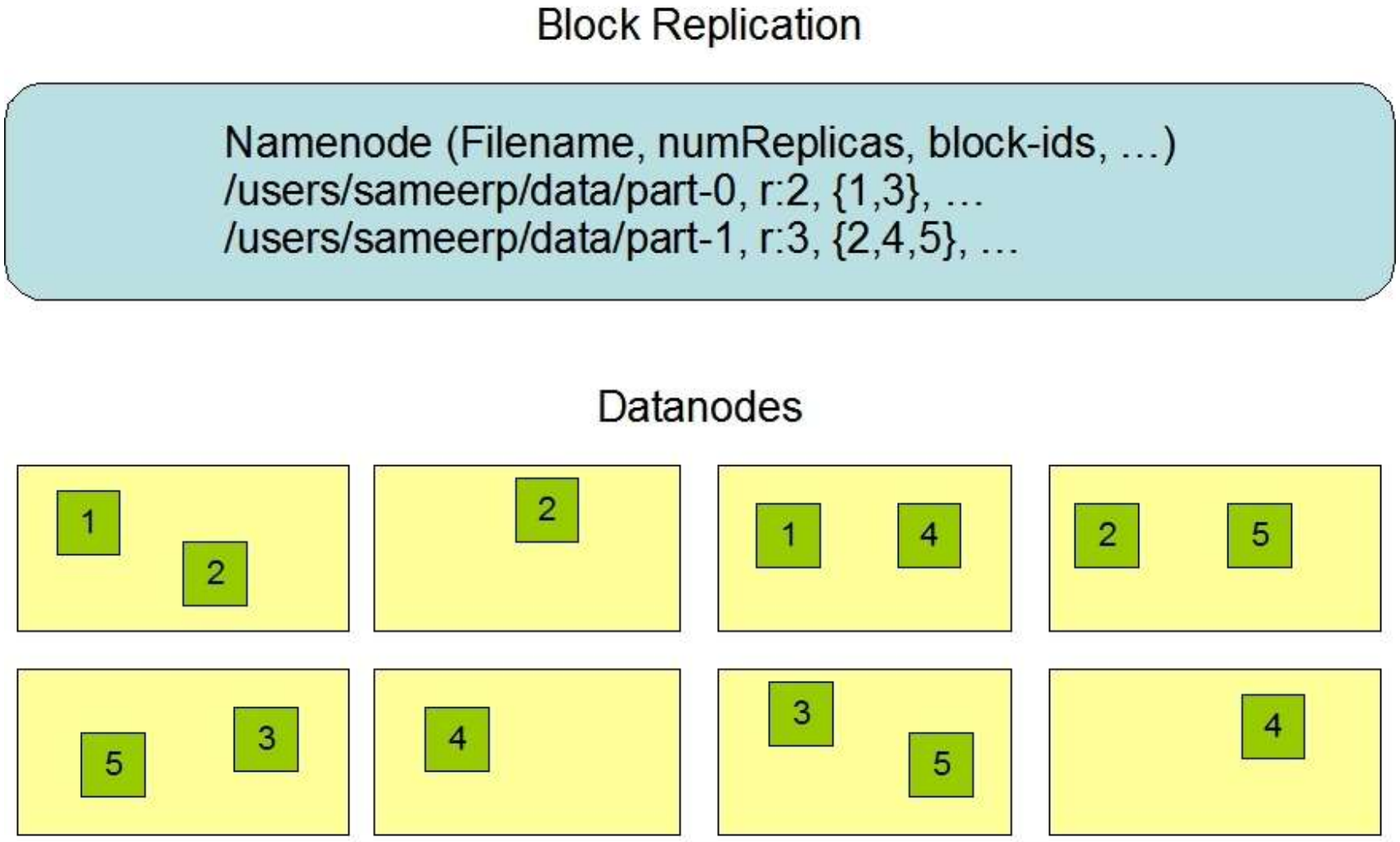
- 每一个文件可以配置块副本数量，默认是3。
- 副本的作用是防止因DataNode故障而导致数据丢失，除此之外块副本还可以提高块可读取的节点。
- 每个文件可以在写入时指定这个文件块的副本数量，也可以在未来修改某个文件的块副本数量，文件块的副本数量配置作为块元数据的一部分保存在NameNode中。

元数据

- 元数据（Metadata）：维护HDFS文件系统中文件和目录的信息，分为内存元数据和元数据文件两种。NameNode维护整个元数据。



块副本（Replication）示例



北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

数据复制流程（1/2）

◆副本位置：第一小步

- 块副本存放位置的选择严重影响HDFS的可靠性和性能。机架敏感的副本存放策略是为了提高数据的可靠性，可用性和网络带宽的利用率。
- 默认的HDFS block放置策略在最小化写开销和最大化数据可靠性、可用性以及总体读取带宽之间进行了一些折中。一般情况下复制因子为3，HDFS的副本放置策略是将第一个副本放在本地节点，将第二个副本放到本地机架上的另外一个节点而将第三个副本放到不同机架上的节点。

数据复制流程（2/2）

- 副本的选择
  - 为了尽量减小全局的带宽消耗读延迟，HDFS尝试返回一个读操作离它最近的副本。假如在读节点的同一个机架上就有这个副本，就直接读这个，如果HDFS集群是跨越多个数据中心，那么本地数据中心的副本优先于远程的副本。
- 安全模式
  - 在启动的时候，名字节点进入一个叫做安全模式的特殊状态。安全模式中不允许发生块复制。名字节点接受来自数据节点的心跳和块报告。一个块报告包含数据节点所拥有的数据块的列表。
  - 每个块有一个特定的最小复制数。当名字节点检查这个块已经大于最小的复制数就被认为是安全地复制了，当达到配置的块安全复制比例时（加上额外的30秒），名字节点就退出安全模式。它将检测数据块的列表，将小于特定复制数的块复制到其他的数据节点。

文件系统的元数据的持久化(1/3)

- HDFS的命名空间是由名字节点来存储的。名字节点使用叫做EditLog的事务日志来持久记录每一个对文件系统元数据的改变，如在HDFS中创建一个新的文件。名字节点在本地文件系统中用一个文件来存储这个EditLog。
- 整个文件系统命名空间，包括文件块的映射表和文件系统的配置都存在一个叫FsImage的文件中，FsImage也存放在名字节点的本地文件系统中。

文件系统的元数据的持久化(2/3)

- 名字节点在内存中保留一个完整的文件系统命名空间和文件块的映射表的镜像。
- 名字节点启动时，它将从磁盘中读取FsImage和EditLog，将EditLog中的所有事务应用到FsImage的仿内存空间，然后将新的FsImage刷新到本地磁盘中，因为事务已经被处理并已经持久化的FsImage中，然后就可以截去旧的EditLog。这个过程叫做**检查点**。当前实现中，检查点仅在名字节点启动的时候发生，正在支持周期性的检查点。

文件系统的元数据的持久化(3/3)

- 数据节点将HDFS数据存储到本地的文件系统中。数据节点并不知道HDFS文件的详细信息，不同的数据块会存储在本地文件系统多个文件中。
- 当数据节点启动的时候，它将扫描它的本地文件系统，根据本地的文件产生一个所有HDFS数据块的列表并报告给名字节点，这个报告称作**块报告**。

通信协议

- 所有的通信协议都是在TCP/IP协议之上构建的。
- 一个客户端和指定TCP配置端口的名字节点建立连接之后，它和名字节点之间通信的协议是Client Protocol。
- 数据节点和名字节点之间通过Datanode Protocol通信。
- RPC（Remote Procedure Call）抽象地封装了Client Protocol和DataNode Protocol协议。按照设计，名字节点不会主动发起一个RPC，它只是被动地对数据节点和客户端发起的RPC作出反馈。

可靠性

- HDFS的主要目标就是在存在故障的情况下也能可靠地存储数据。三个最常见的故障是名字节点故障，数据节点故障和网络断开。
- 一个数据节点周期性发送一个心跳包到名字节点。网络断开会造成一组数据节点子集和名字节点失去联系。名字节点根据缺失的心跳信息判断故障情况。名字节点将这些数据节点标记为死亡状态，不再将新的IO请求转发到这些数据节点上，这些数据节点上的数据将对HDFS不再可用，可能会导致一些块的复制因子降低到指定的值。
- 名字节点检查所有的需要复制的块，并开始复制他们到其他的数据节点上。重新复制在有些情况下是不可或缺的，例如：数据节点失效，副本损坏，数据节点磁盘损坏或者文件的复制因子增大。

负载重平衡(Cluster Rebalancing)

- 该方案会在数据节点的可用空间处于阈值以下时自动将数据转移到其它数据节点。
- 当一个文件遇到突发的高存取需求时，它可以动态创建数据复本以重新平衡集群中的负载。目前，数据负载的重平衡方案还没有实现。

数据完整性

- 从数据节点上取一个文件块有可能是坏块，坏块的出现可能是存储设备错误，网络错误或者软件的漏洞。
- HDFS客户端实现了HDFS文件内容的校验。当一个客户端创建一个HDFS文件时，它会为每一个文件块计算一个校验码并将校验码存储在同一个HDFS命名空间下一个单独的隐藏文件中。
- 当客户端访问这个文件时，它根据对应的校验文件来验证从数据节点接收到的数据。如果校验失败，客户端可以选择从其他拥有该块副本的数据节点获取这个块。

元数据失效

- FsImage和Editlog是HDFS的核心数据结构。这些文件的损坏会导致整个集群的失效。因此，名字节点可以配置成支持多个FsImage和EditLog的副本。任何FsImage和EditLog的更新都会同步到每一份副本中。
- 同步更新多个EditLog副本会降低名字节点的命名空间事务交易速率。但是这种降低是可以接受的，因为HDFS程序中产生大量的数据请求，而不是元数据请求。名字节点重新启动时，选择最新一致的FsImage和EditLog。
- 名字节点对于一个HDFS集群是单点失效的。假如名字节点失效，就需要人工的干预。还不支持自动重启和到其它名字节点的切换。

快照(snapshot)

- 快照支持在一个特定时间存储一个数据拷贝，快照可以将失效的集群回滚到之前一个正常的时间点上。HDFS已经支持元数据快照。

数据组织-阶段状态(staging)(1/2)

- 一个客户端创建一个文件的请求并不会立即转发到名字节点。
- 实际上，一开始HDFS客户端将文件数据缓存在本地的临时文件中。应用程序的写操作被透明地重定向到这个临时本地文件。当本地文件堆积到一个HDFS块大小的时候，客户端才会通知名字节点。
- 名字节点将文件名插入到文件系统层次中，然后为它分配一个数据块。名字节点构造包括数据节点ID（可能是多个，副本数据块存放的节点也有）和目标数据块标识的报文，用它回复客户端的请求。
- 客户端收到后将本地的临时文件刷新到指定的数据节点数据块中。

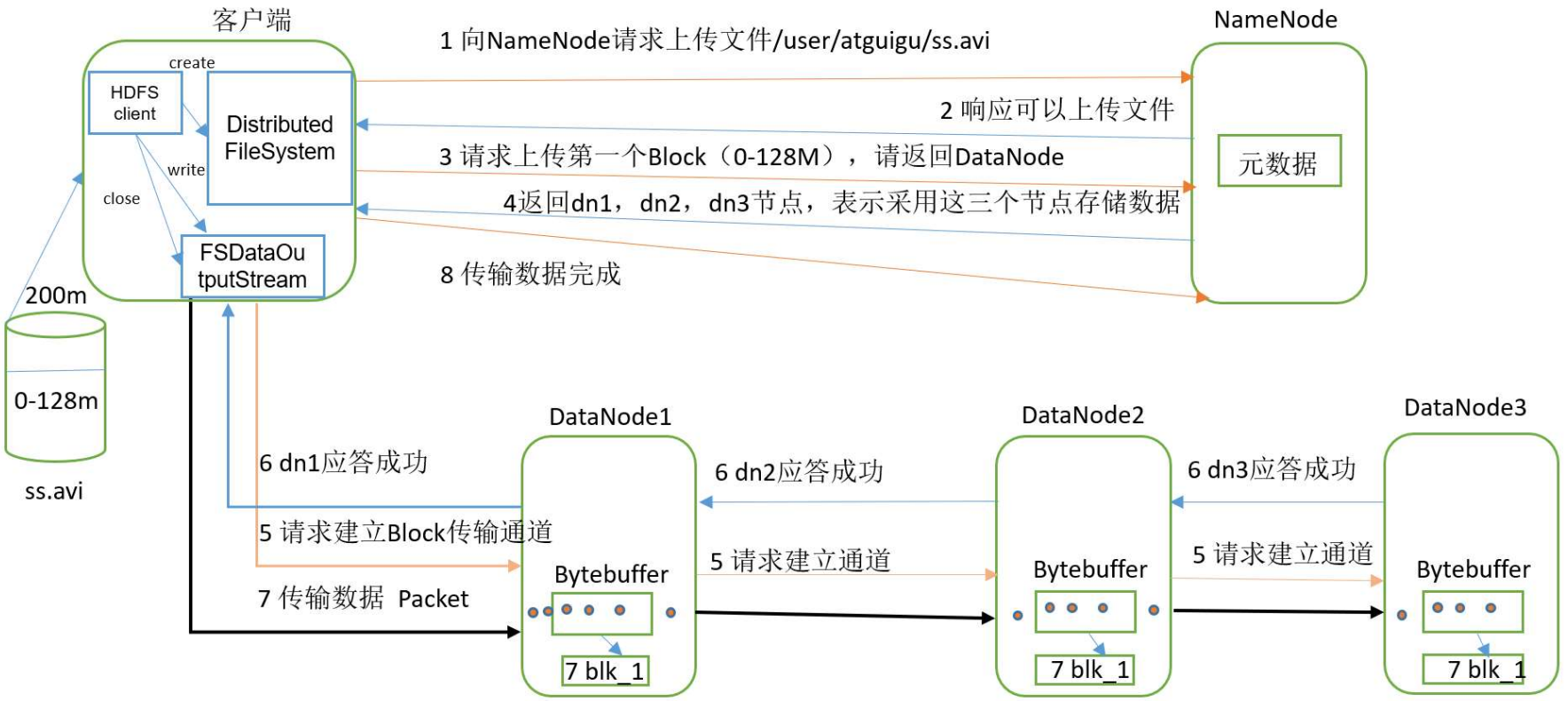
数据组织-阶段状态(2/2)

- 当文件关闭时，本地临时文件中未上传的残留数据就会被转送到数据节点。然后客户端就可以通知名字节点文件已经关闭。此时，名字节点将文件的创建操作添加到到持久化存储中。假如名字节点在文件关闭之前发生故障，文件就丢掉了。
- 上述流程是在认真考虑了运行在HDFS上的目标程序之后被采用。这些应用程序需要流式地写文件。如果客户端对远程文件系统进行直接写入而没有任何本地的缓存，这就会对网速和网络吞吐量产生很大的影响。这方面早有前车之鉴，早期的分布式文件系统如AFS，也用客户端缓冲来提高性能，POSIX接口的限制也被放宽以达到更高的数据上传速率。

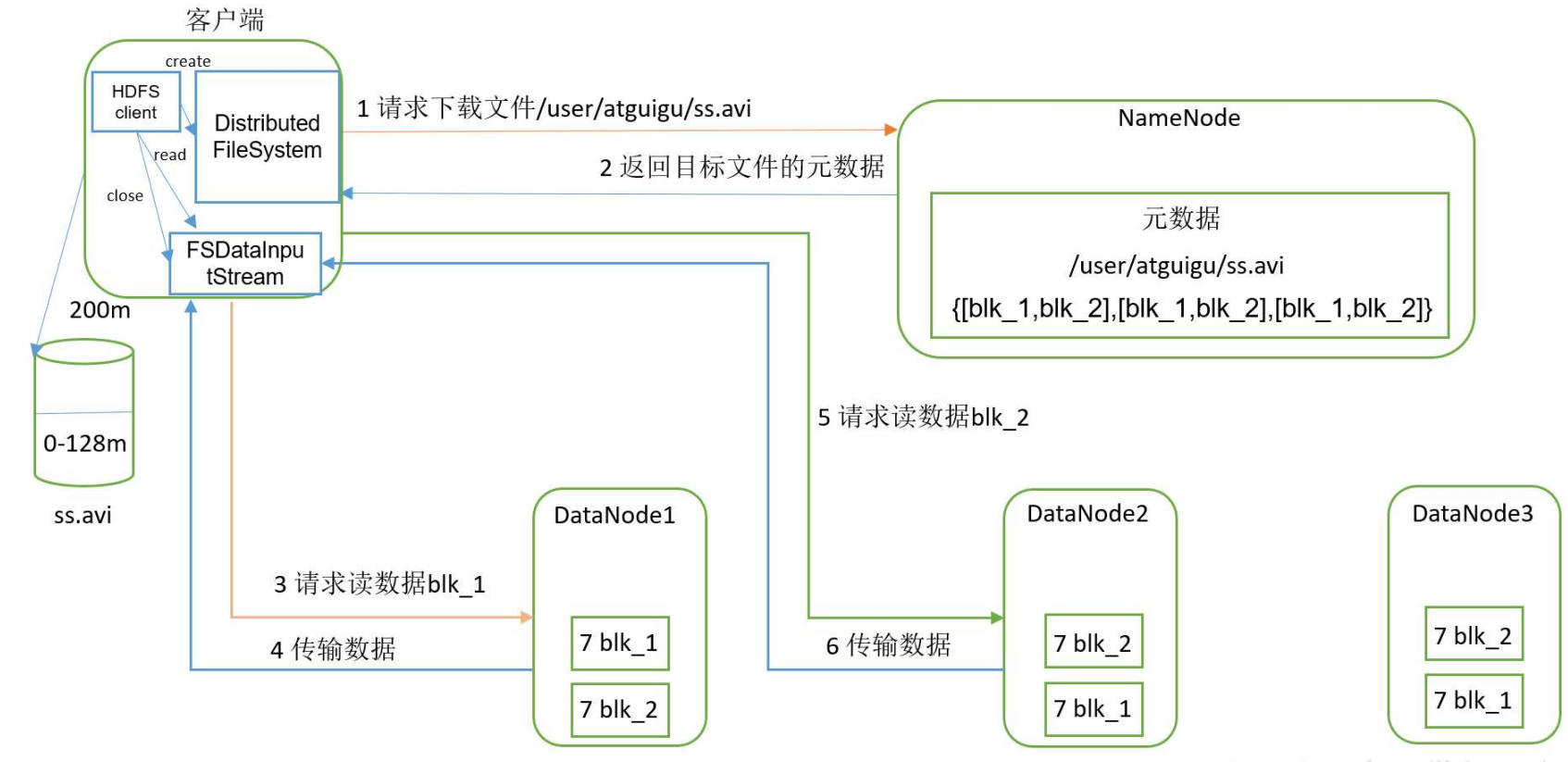
数据组织-流式复制(Replication Pipelining)

- 当客户端写数据到HDFS文件中时，如上所述，数据首先被写入本地文件中，假设HDFS文件的复制因子是3，当本地文件堆积到一块大小的数据，客户端从名字节点获得一个数据节点的列表。这个列表也包含存放数据块副本的数据节点。
- 当客户端刷新数据块到第一个数据节点。第一个数据节点开始以4kb为单元接收数据，将每一小块都写到本地库中，同时将每一小块都传送到列表中的第二个数据节点。同理，第二个数据节点将小块数据写入本地库中同时传给第三个数据节点，第三个数据节点直接写到本地库中。
- 一个数据节点在接前一个节点数据的同时，还可以将数据流水式传递给下一个节点，所以，数据是流水式地从一个数据节点传递到下一个。

HDFS写流程



读流程



访问方式(Accessibility)

- DFSShell：HDFS允许用户数据组织成文件和文件夹的方式，它提供一个叫DFSShell的接口，使用户可以和HDFS中的数据交互。命令集的语法跟bash,csh。
- DFSAdmin命令集是用于管理dfs集群的，这些命令只由HDFS管理员使用。
- 浏览器接口：典型的HDFS初始化配置了一个web 服务，通过一个可配的TCP端口可以访问HDFS的命名空间。这就使得用户可以通过web浏览器去查看HDFS命名空间的内容。
- Java api。

DFSShell示例

Action	Command
创建目录 /foodir	hdfs dfs -mkdir /foodir
查看文件 /foodir/myfile.txt	hadoop dfs -cat /foodir/myfile.txt
删除文件/foodir/myfile.txt	hadoop dfs -rm /foodir myfile.txt

DFSAdmin示例

Action	Command
将集群设置成安全模式	bin/hadoop dfsadmin -safemode enter
产生一个数据节点的列表	bin/hadoop dfsadmin -report
去掉一个数据节点	bin/hadoop dfsadmin -decommission datanodename

HDFS的文件读取解析

```java
public static void read(Path path) throws IOException{
    FileSystem hdfs = HdfsUtils.getFilesystem();  //步骤 1
    FSDataInputStream fsDataInputStream =  hdfs.open(path); //步骤 2
    IOUtils.copyBytes(fsDataInputStream, System.out, 4096,false);  //步骤 3
}
```

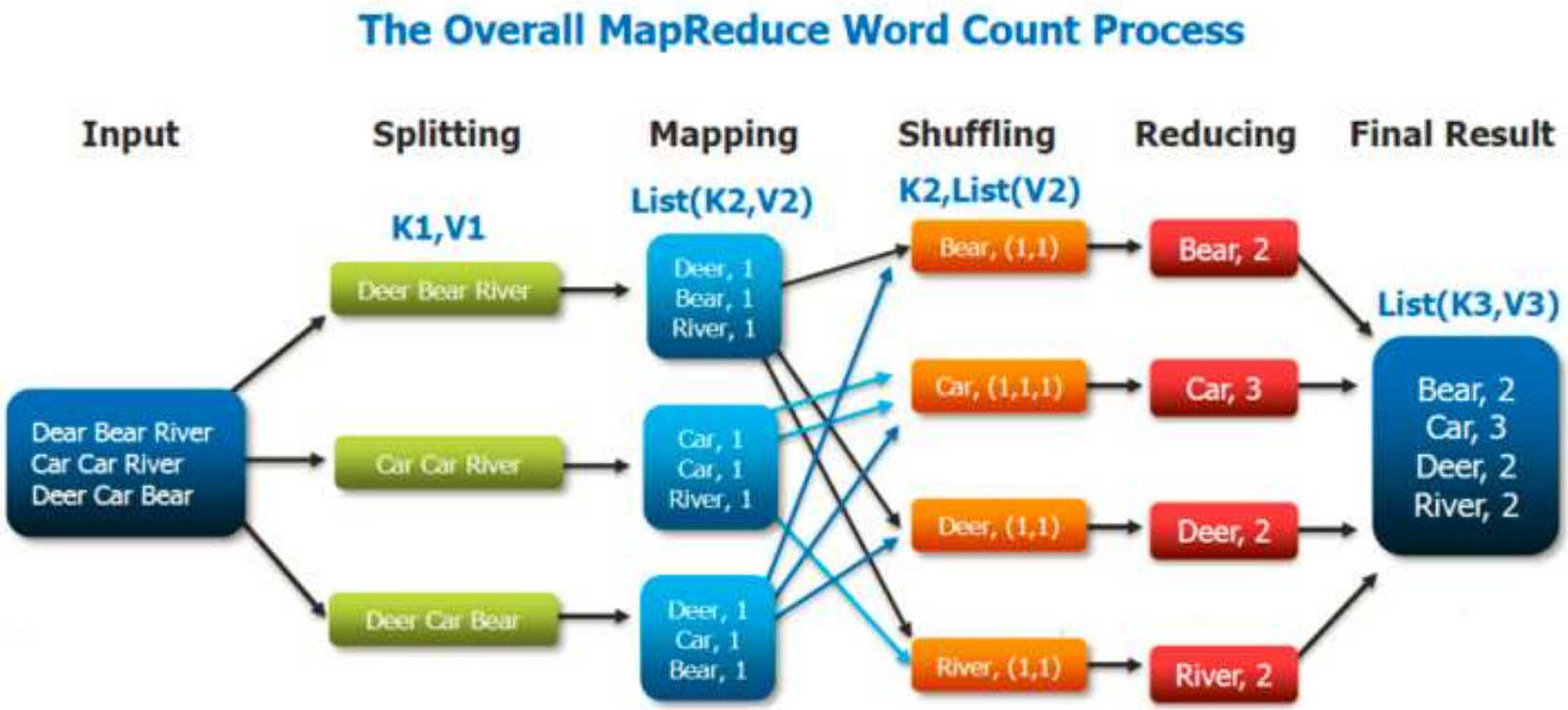
```java
package com.yq.common;

import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;

public class HdfsUtils {
    public static FileSystem getFilesystem(){
        FileSystem hdfs=null;
        Configuration conf=new Configuration();
        try{
            URI uri = new URI("hdfs://localhost:9000");
            hdfs = FileSystem.get(uri,conf);
        }
        catch(Exception ex){
        //
        }
    return hdfs;
    }
}
```

北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

Hadoop MapReduce是什么

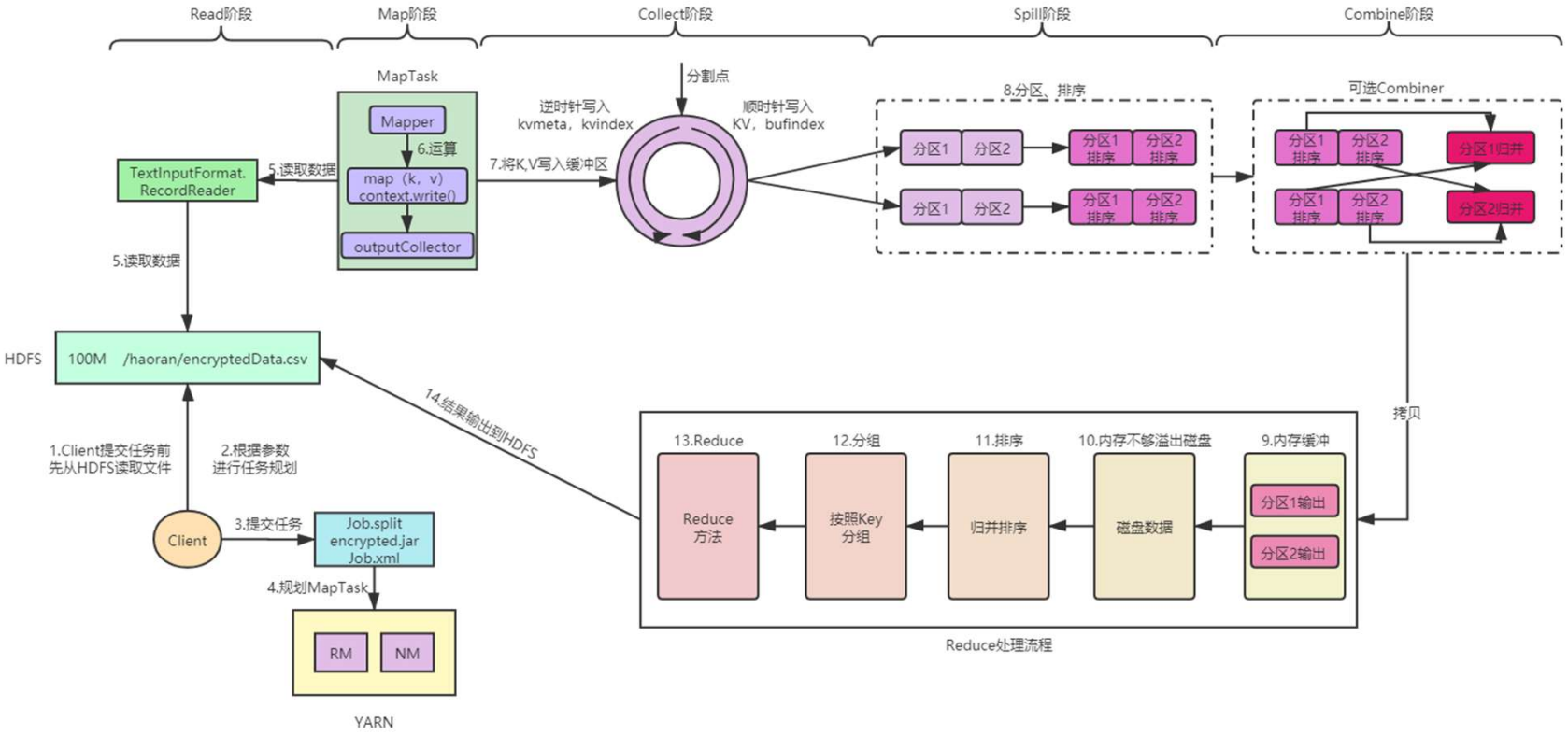
- Hadoop MapReduce是一个**软件框架**，基于该框架能够容易地编写应用程序，这些应用程序能够运行在由上千个商用机器组成的大集群上，并以一种可靠的，具有**容错**能力的方式**并行地处理**上TB级别的**海量数据集**。

MapReduce之词频统计分析



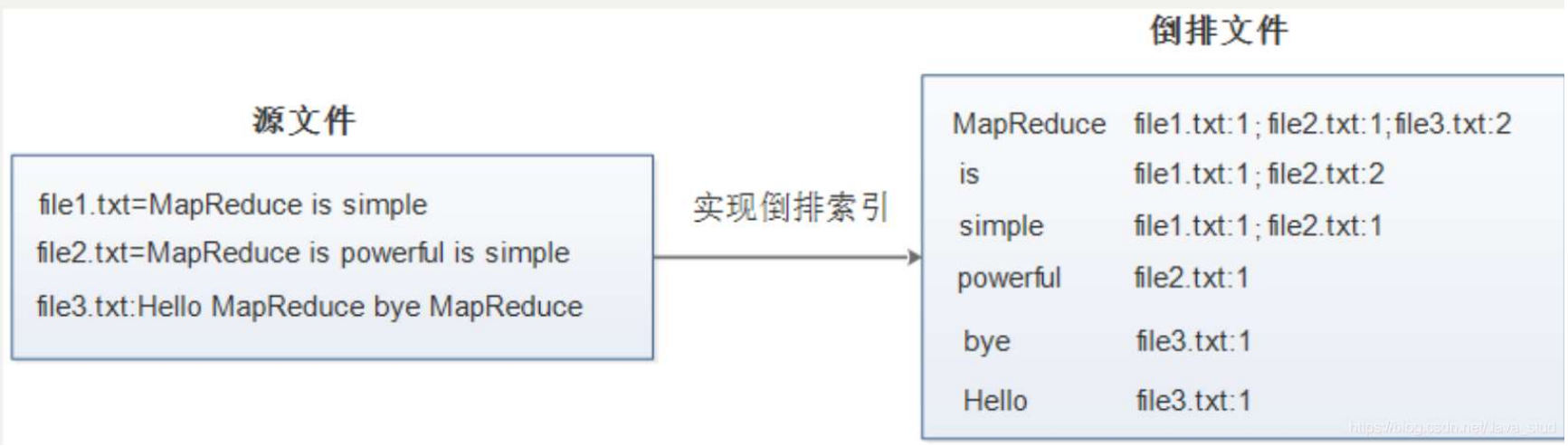
北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

MapReduce的处理流程

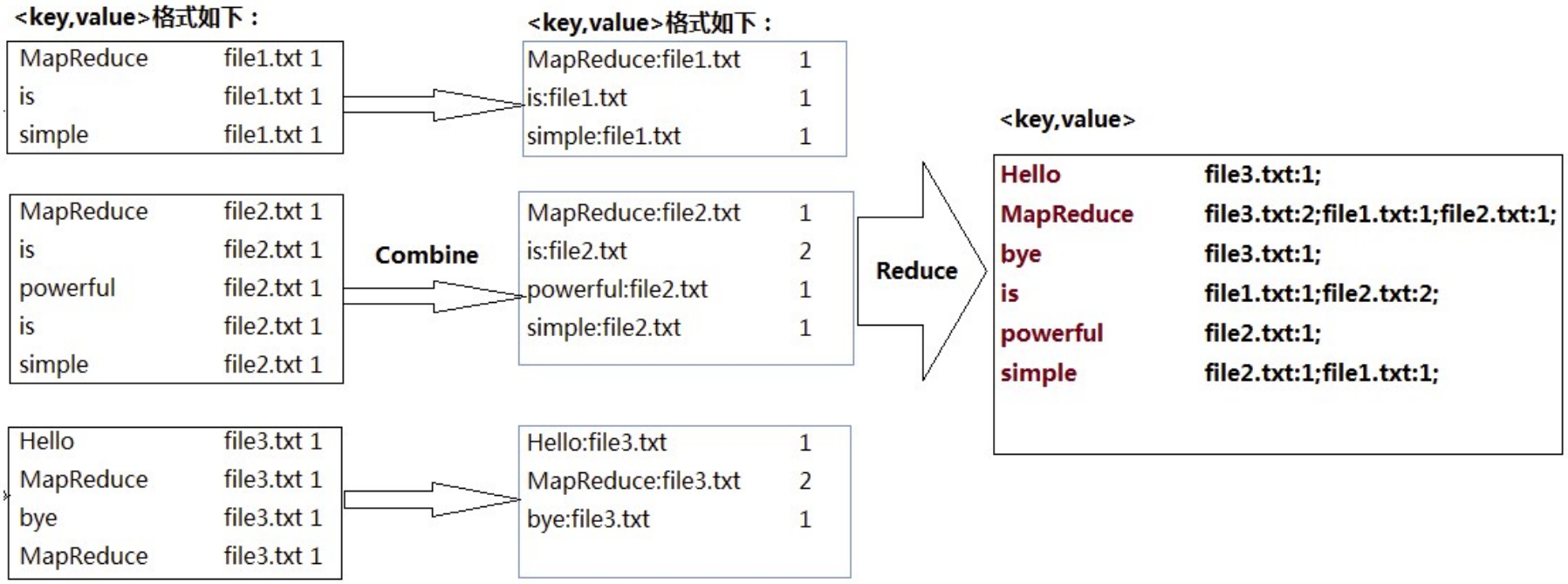


北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

倒排索引


倒排文件
源文件
file1.txt=MapReduce is simple
file2.txt=MapReduce is powerful is simple
file3.txt:Hello MapReduce bye MapReduce
实现倒排索引
MapReduce file1.txt:1;file2.txt:1;file3.txt:2
is file1.txt:1;file2.txt:2
simple file1.txt:1;file2.txt:1
powerful file2.txt:1
bye file3.txt:1
Hello file3.txt:1

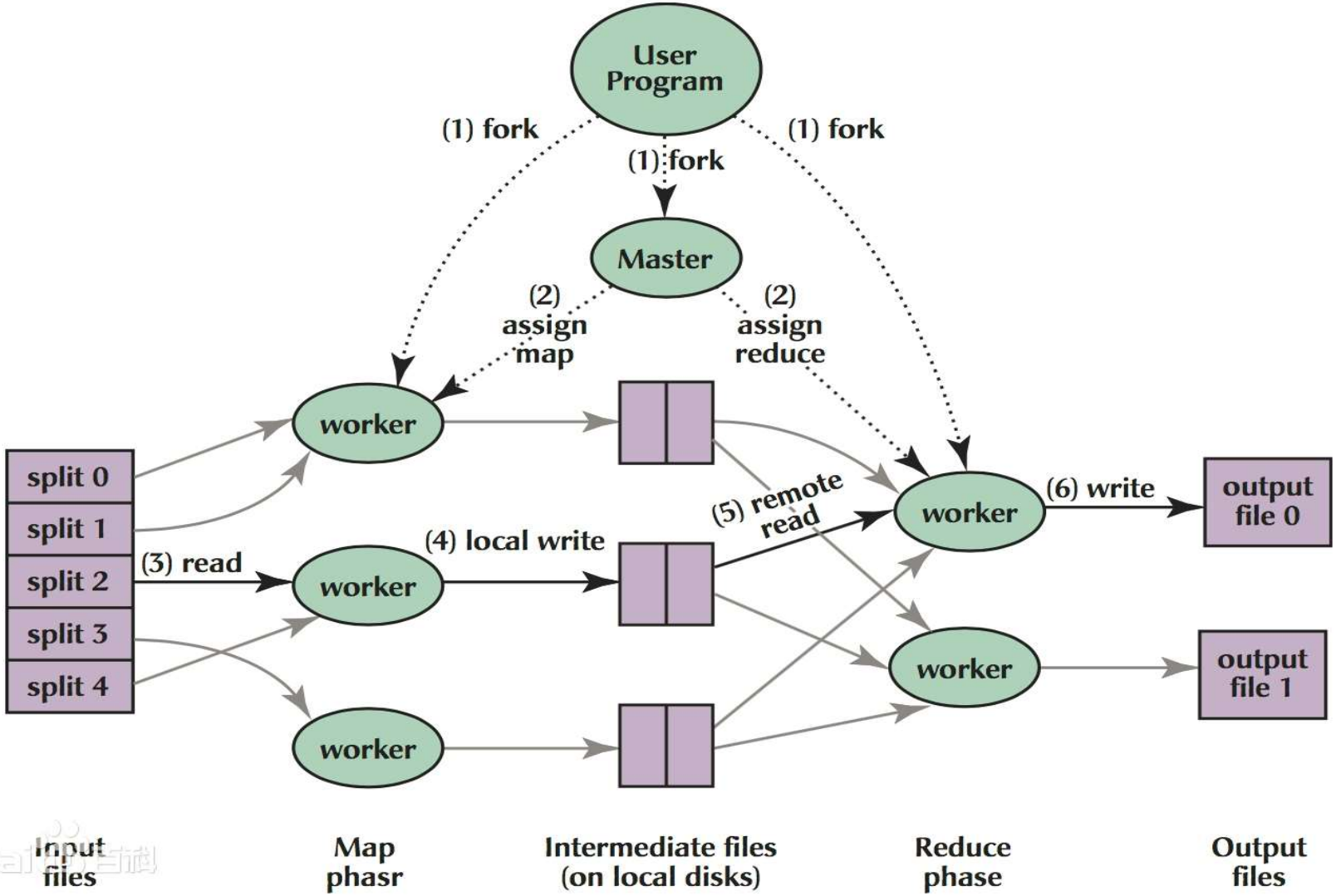
MapReduce之倒排索引



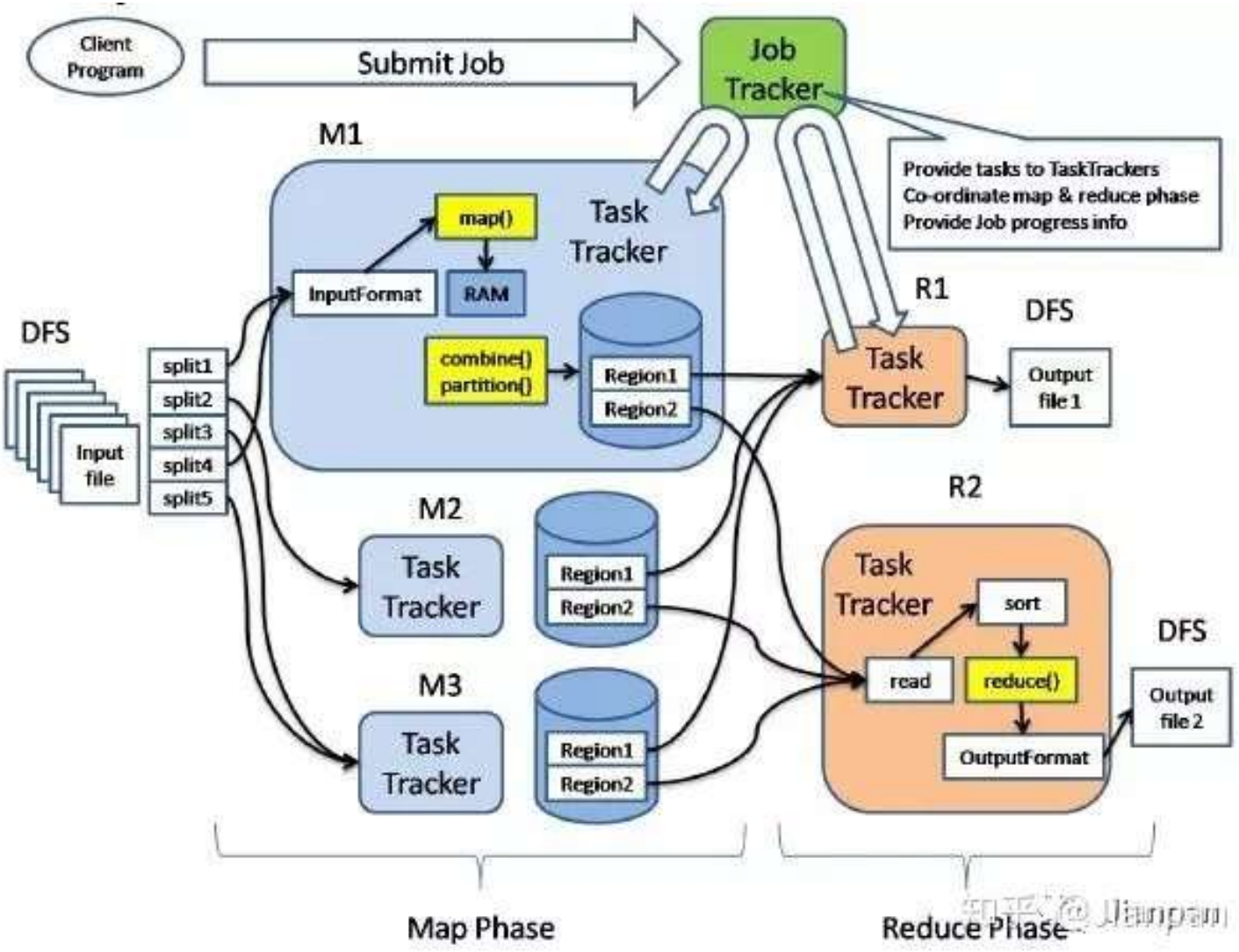
MapReduce处理过程

- 什么是MapReduce统合来看，MapReduce就是你有很多各种各样的蔬菜水果面包（Input），有很多厨师，不同的厨师分到了不同的蔬菜水果面包，自己主动去拿过来（Split），拿到手上以后切碎（Map），切碎以后给到不同的烤箱里，冷藏机里（Shuffle），冷藏机往往需要主动去拿，拿到这些东西存放好以后会根据不同的顾客需求拿不同的素材拼装成最终的结果，这就是Reduce，产生结果以后会放到顾客那边等待付费（Ticket），这个过程是Finalize。
- 所以MapReduce是六大过程：**Input, Split, Map, Shuffle, Reduce, Finalize**。

MapReduce架构



MapReduce工作原理



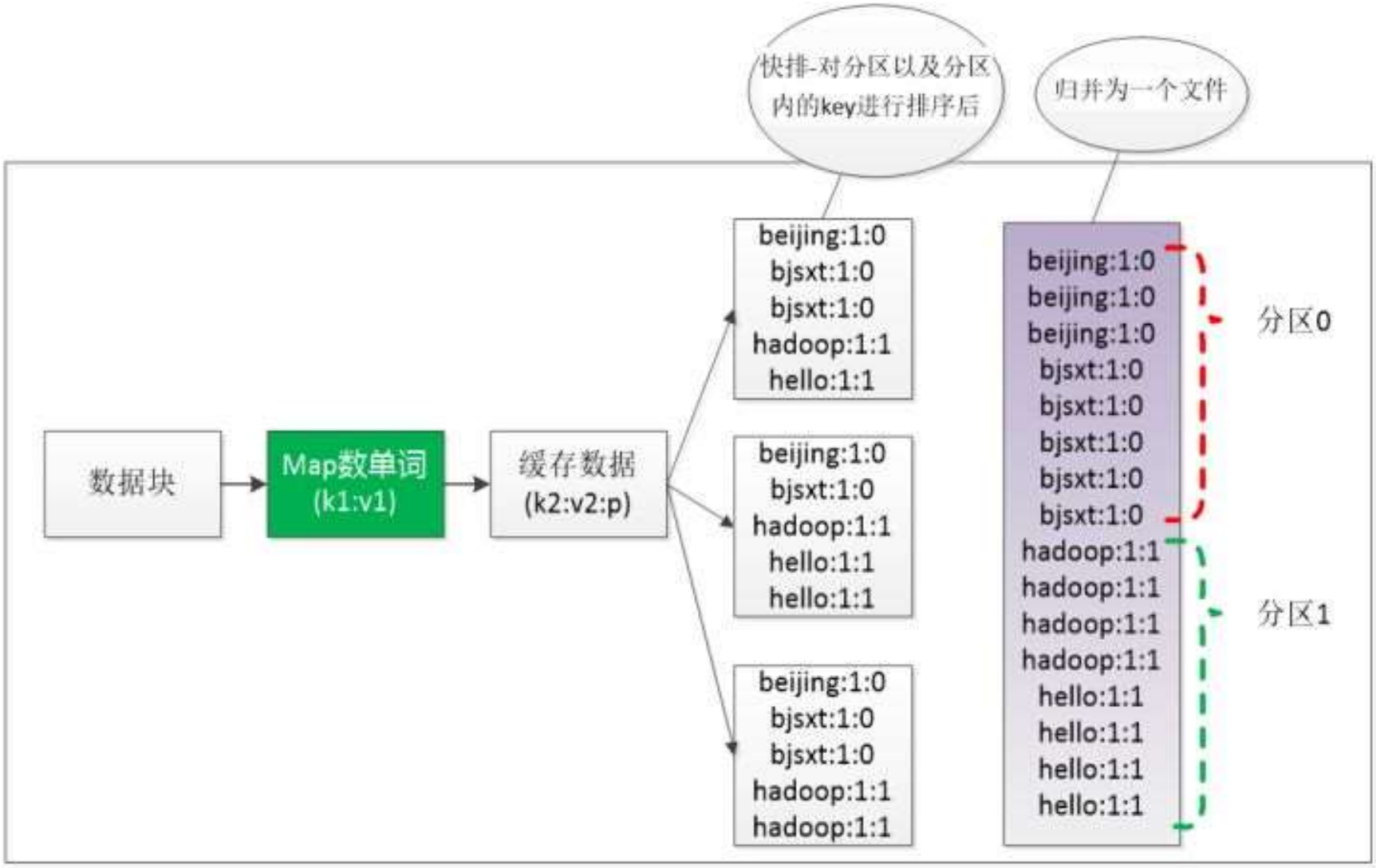
MapReduce示例程序

- 参见MapReduce官方网站
  https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Purpose

MapReduce的输入输出

- MapReduce框架运转在<key,value>键值对上，也就是说，框架把作业的输入看成是一组<key,value>键值对，同样也产生一组<key,value>键值对作为作业的输出，这两组键值对有可能是不同的。
- 一个MapReduce作业的输入和输出类型如下图所示：可以看出在整个流程中，会有三组<key,value>键值对类型的存在。

MapReduce分区



分区实现及自定义分区

- Partitioner决定Map节点的输出将被分区到哪个Reduce节点。默认的Partitioner是HashPartitioner。
- 用户可以定制并使用自己的Partitioner。

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

  /** Use {@link Object#hashCode()} to partition. */
  public int getPartition(K key, V value,
                          int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }

}
```

```
job.setPartitionerClass(NewPartitionerName.class);
```