# 《大数据基础》

## 2021~2022年秋期末复习

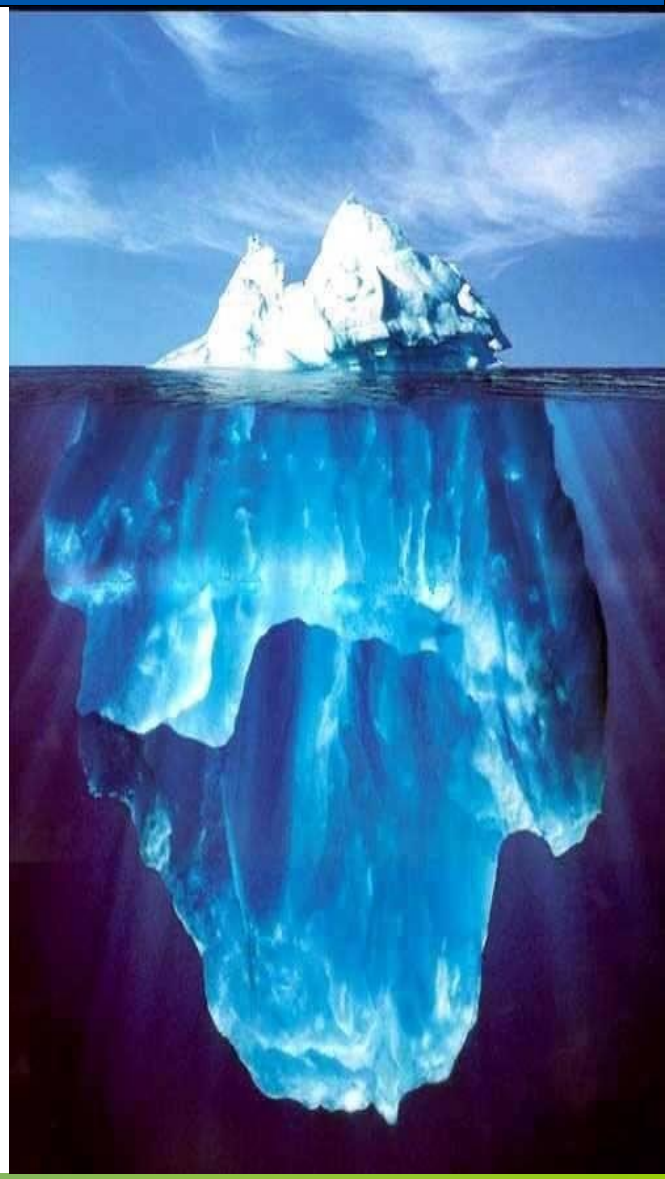
孙笑笑

杭州电子科技大学计算机学院

**E-mail: sunxiaoxiao@hdu.edu.cn**

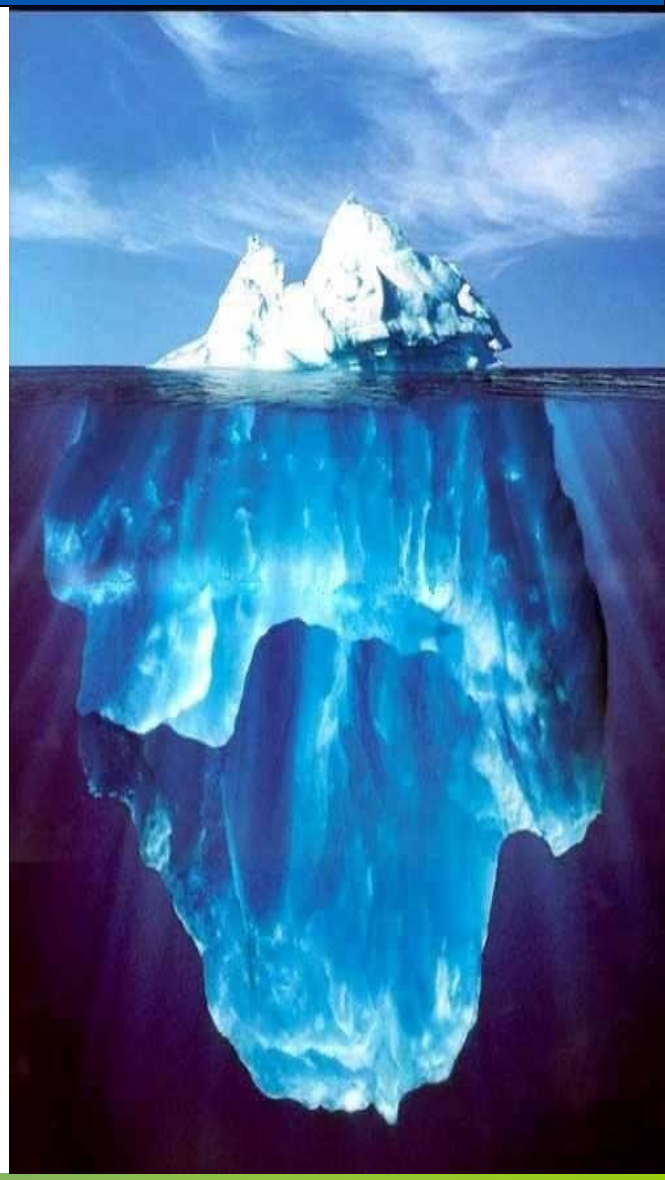
# 题型

1. 选择题（30分=15题*2分/题）
2. 简答题（35分=15分+20分）
3. 分析题（35分=17分+18分）

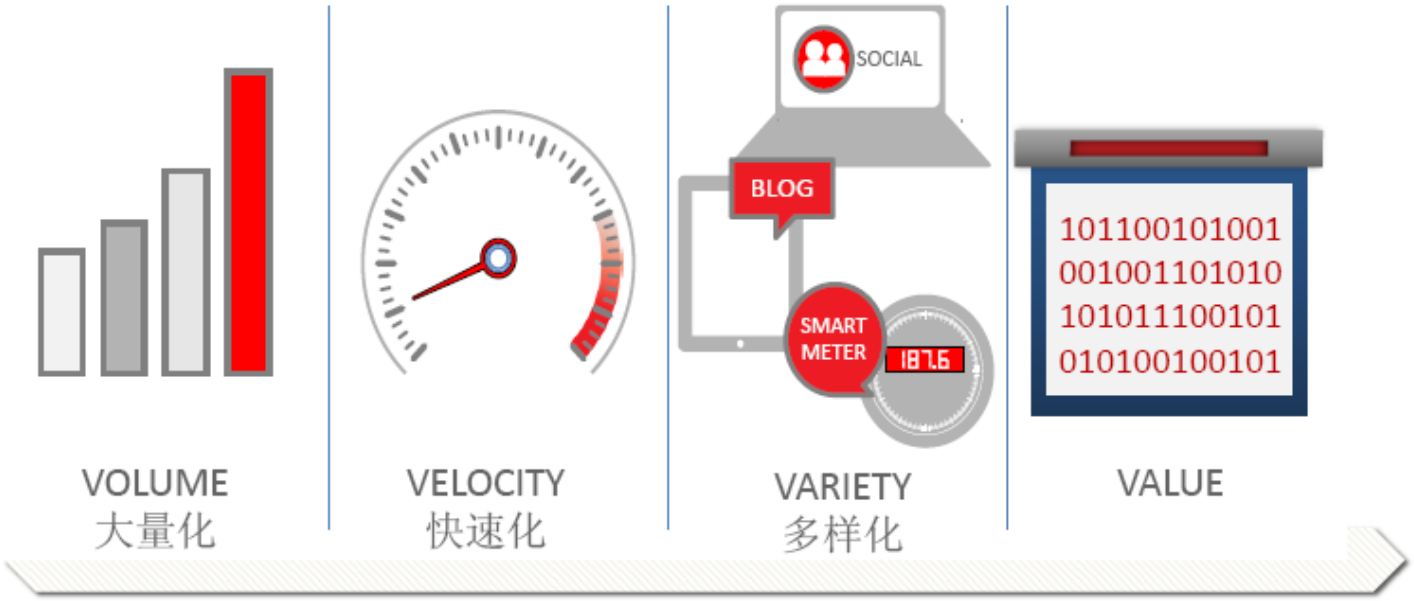


# 第一章 大数据概述

1.1 大数据时代
1.2 大数据概念
1.3 大数据的影响
1.4 大数据的应用
1.5 大数据关键技术
1.6 大数据计算模式
1.7 大数据产业
1.8 大数据与云计算、物联网的关系



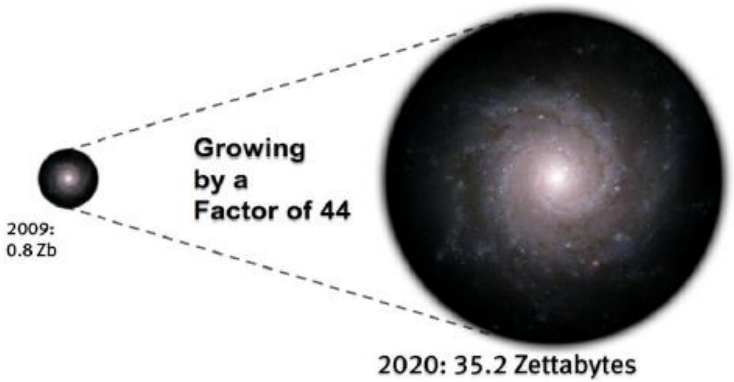
# 1.2大数据概念




大数据不仅仅是数据的“大量化”，而是包含“快速化”、“多样化”和“价值化”等多重属性。

# 1.2.1数据量大（Volume）

- 根据IDC作出的估测，数据一直都在以每年50%的速度增长，也就是说每两年就增长一倍（大数据摩尔定律）
- 人类在最近两年产生的数据量相当于之前产生的全部数据量
- 预计到2020年，全球将总共拥有35ZB的数据量，相较于2010年，数据量将增长近30倍




| | | | | |
|---|---|---|---|---|
| TERABYTE | 10 的 12 次方 | 一块 1TB 硬盘 |  | 200,000 照片或 mp3 歌曲 |
| PETABYTE | 10 的 15 次方 | 两个数据中心机柜 |  | 16 个 Blackblaze pod 存储单元 |
| EXABYTE | 10 的 18 次方 | 2,000 个机柜 |  | 占据一个街区的 4 层数据中心 |
| ZETTABYTE | 10 的 21 次方 | 1000 个数据中心 |  | 纽约曼哈顿的 1/5 区域 |
| YOTTABYTE | 10 的 24 次方 | 一百万个数据中心 |  | 特拉华州和罗德岛州 |

# 1.2.2 数据类型繁多（Variety）

- 大数据是由结构化和非结构化数据组成的
  - 10%的结构化数据，存储在数据库中
  - 90%的非结构化数据，它们与人类信息密切相关

# 1.2.2 数据类型繁多


京东
亚马逊
淘宝
购物
王者荣耀
游戏
绝地求生
支付宝
微信支付
银联
金融
优酷
娱乐
抖音
快手
SAP
IBM
业务
个人微博
微信
新浪微博
社交

# 1.2.3处理速度快（Velocity）

- 从数据的生成到消耗，时间窗口非常小，可用于生成决策的时间非常少
- 1秒定律：这一点也是和传统的数据挖掘技术有着本质的不同

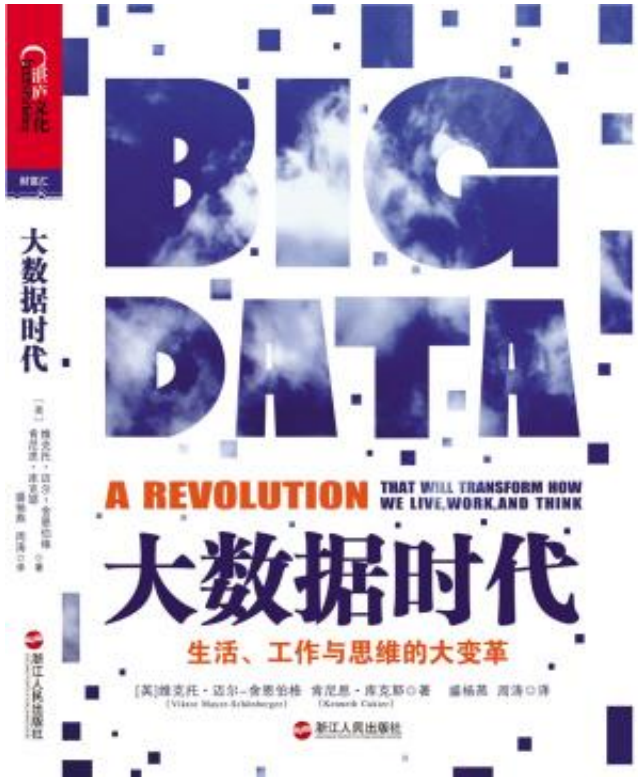
# 1.2.4价值密度低（Value）

**价值密度低，商业价值高**

以视频为例，连续不间断监控过程中，可能有用的数据仅仅有一两秒，但是具有很高的商业价值

# 1.3大数据的影响

- 在思维方式方面，大数据完全颠覆了传统的思维方式：
  - 全样而非抽样
  - 效率而非精确
  - 相关而非因果



# 1.8.3大数据与云计算、物联网的关系

- 云计算、大数据和物联网代表了IT领域最新的技术发展趋势，三者既有区别又有联系

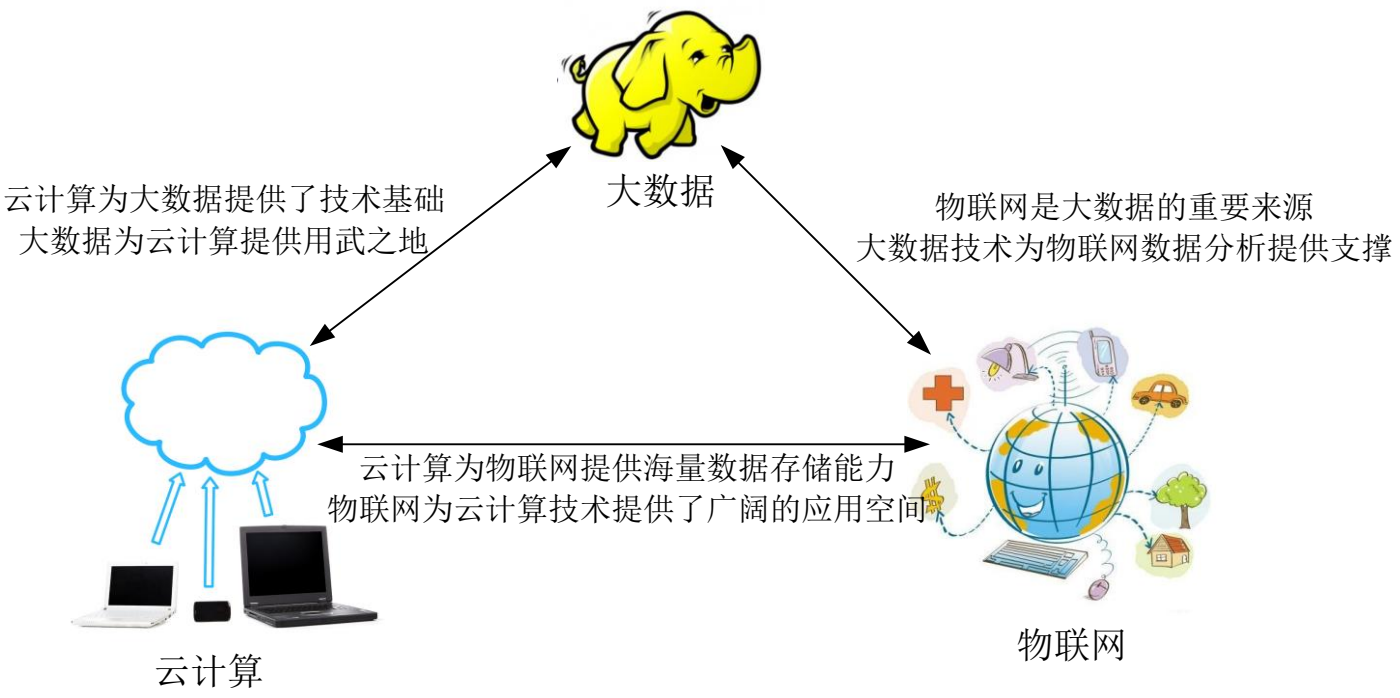
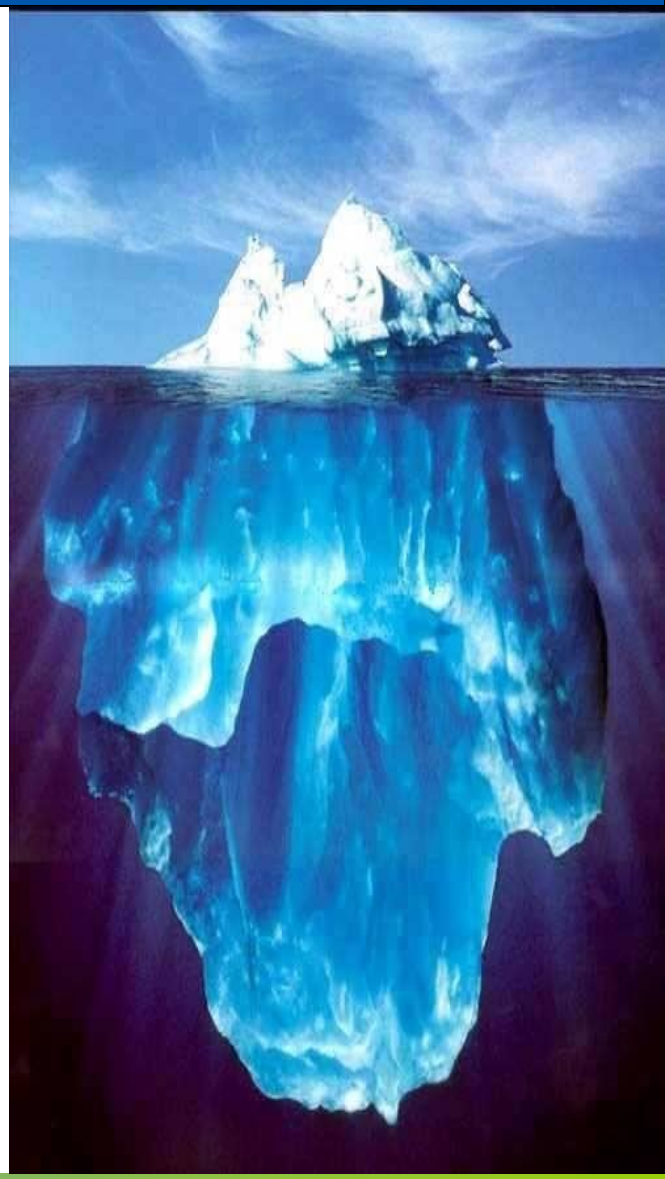



图1-9 大数据、云计算和物联网之间的关系

# 第二章 大数据处理架构Hadoop

- 2.1 概述
- 2.2 Hadoop项目结构
- 2.3 Hadoop的安装与使用
- 2.4 Hadoop集群的部署与使用



# 2.1.1 Hadoop简介

• Hadoop是Apache软件基金会旗下的一个**开源**分布式计算平台，为用户提供了系统底层细节透明的分布式基础架构

•Hadoop是基于Java语言开发的，具有很好的**跨平台**特性，并且可以部署在廉价的计算机集群中 (C,C++,Python均可开发)

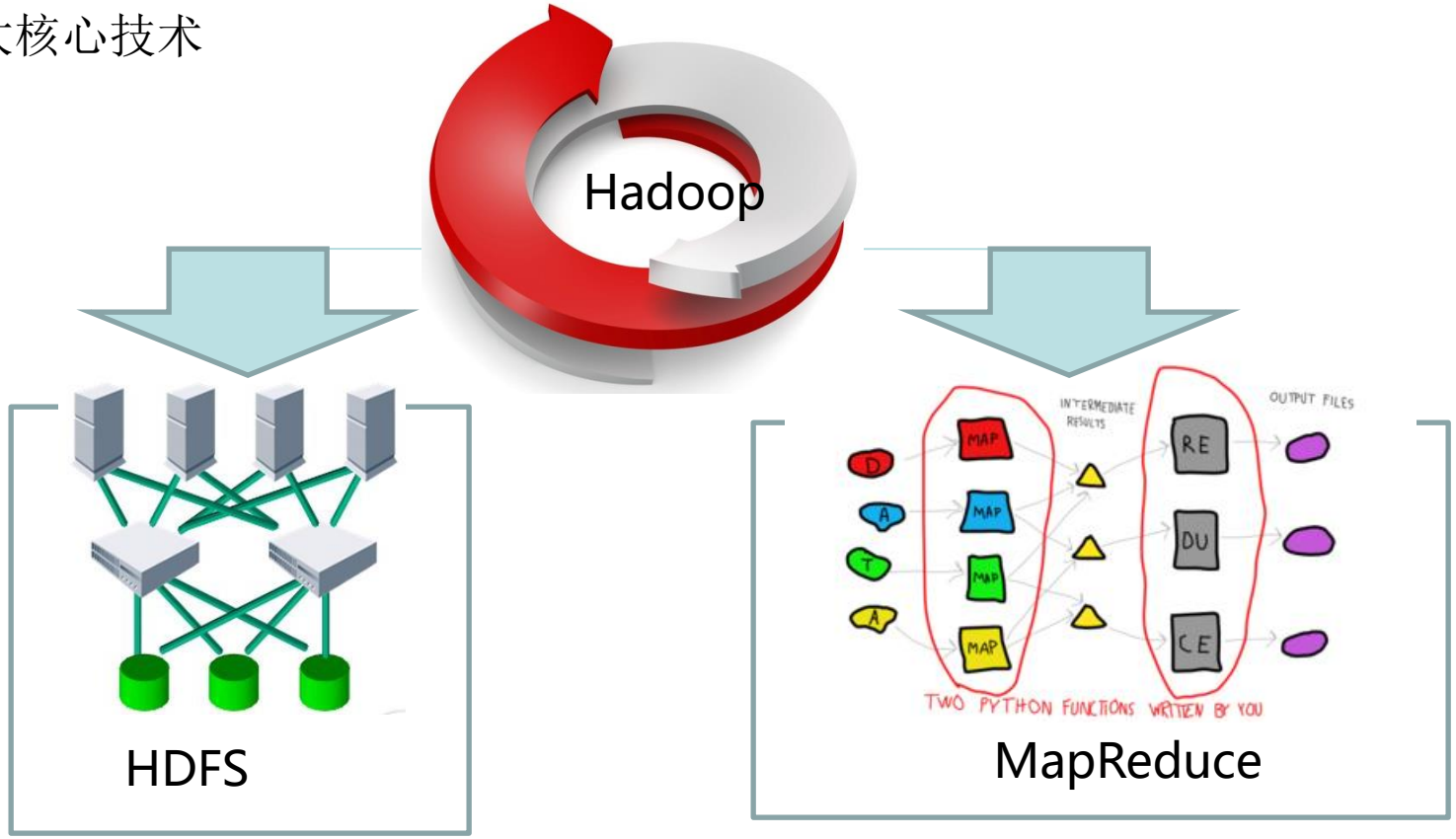
•Hadoop的核心是分布式文件系统**HDFS**（Hadoop Distributed File System）和**MapReduce**（解决了大数据两大核心问题：海量数据的分布式存储和分布式处理）

•Hadoop被公认为行业大数据标准开源软件，在分布式环境下提供了海量数据的处理能力

•几乎所有主流厂商都围绕Hadoop提供开发工具、开源软件、商业化工具和技术服务，如谷歌、雅虎、微软、思科、淘宝等，都支持Hadoop

# 2.1.1 Hadoop简介

两大核心技术




# 2.1.1 Hadoop简介

• Hadoop是Apache软件基金会旗下的一个**开源**分布式计算平台，为用户提供了系统底层细节透明的分布式基础架构

•Hadoop是基于Java语言开发的，具有很好的**跨平台**特性，并且可以部署在廉价的计算机集群中 (C,C++,Python均可开发)

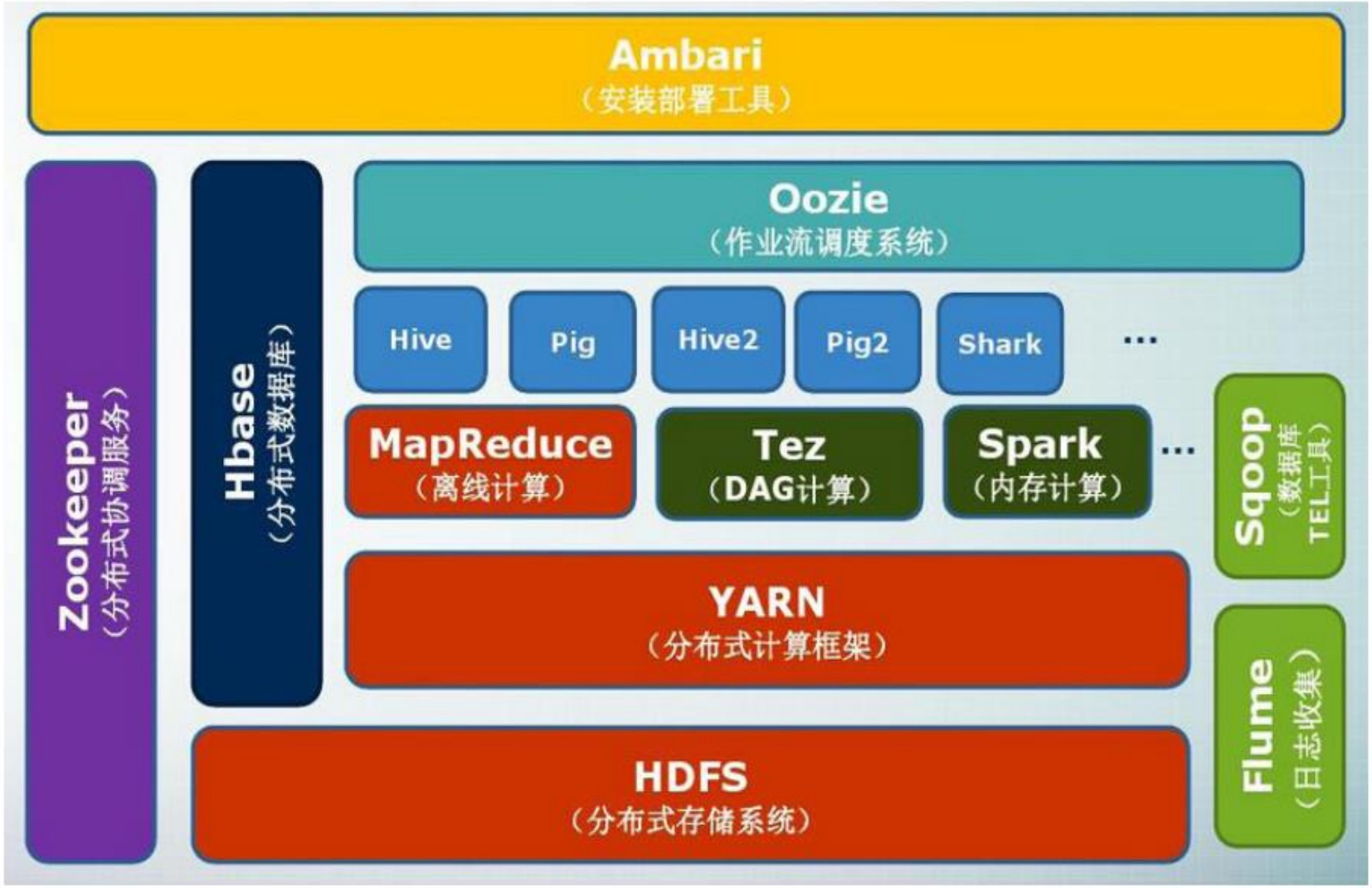
•Hadoop的核心是分布式文件系统**HDFS**（Hadoop Distributed File System）和**MapReduce**（解决了大数据两大核心问题：海量数据的分布式存储和分布式处理）

•Hadoop被公认为行业大数据标准开源软件，在分布式环境下提供了海量数据的处理能力

•几乎所有主流厂商都围绕Hadoop提供开发工具、开源软件、商业化工具和技术服务，如谷歌、雅虎、微软、思科、淘宝等，都支持Hadoop

# 2.2 Hadoop项目结构

Hadoop的项目结构不断丰富发展，已经形成一个丰富的Hadoop生态系统

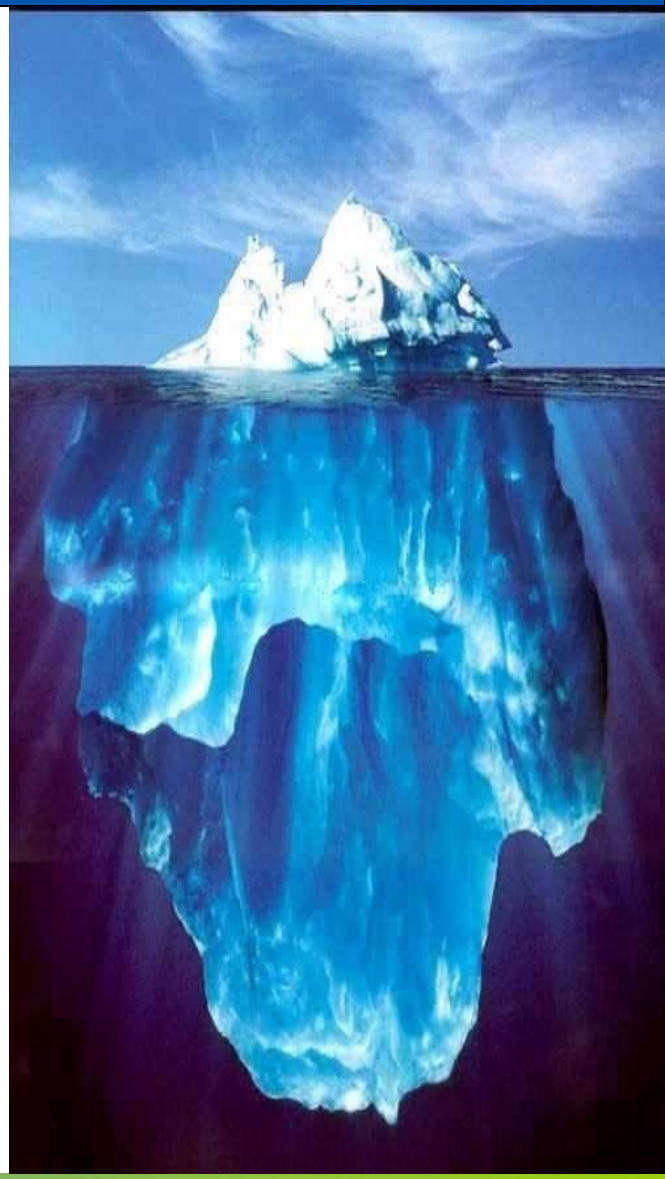


# 2.2 Hadoop项目结构

| 组件 | 功能 |
| --- | --- |
| HDFS | 分布式文件系统 |
| MapReduce | 分布式并行编程模型 |
| YARN | 资源管理和调度器 |
| Tez | 运行在YARN之上的下一代Hadoop查询处理框架，构建DAG有向无环图，子任务的先后执行顺序 |
| Hive | Hadoop上的数据仓库，保留大量历史数据，支持SQL语句进行数据OLAP分析，并转化为MapReduce作业 |
| HBase | Hadoop上的非关系型的分布式数据库 |
| Pig | 一个基于Hadoop的大规模数据分析平台，提供类似SQL的查询语言Pig Latin，轻量级简单编程 |
| Sqoop | 用于在Hadoop与传统数据库之间进行数据传递 |
| Oozie | Hadoop上的工作流管理系统 |
| Zookeeper | 提供分布式协调一致性服务 |
| Storm | 流计算框架 |
| Flume | 一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统 |
| Ambari | Hadoop快速部署工具，支持Apache Hadoop集群的供应、管理和监控 |
| Kafka | 一种高吞吐量的分布式发布订阅消息系统，可以处理消费者规模的网站中的所有动作流数据 |
| Spark | 类似于Hadoop MapReduce的通用并行框架 |

# 第三章 分布式文件系统HDFS

- 3.1 分布式文件系统
- 3.2 HDFS简介
- 3.3 HDFS相关概念
- 3.4 HDFS体系结构
- 3.5 HDFS存储原理
- 3.6 HDFS数据读写过程
- 3.7 HDFS编程实践



# 3.2 HDFS简介

总体而言，HDFS要实现以下目标：

- 兼容廉价的硬件设备
- 大数据集
- 简单的文件模型
- 强大的跨平台兼容性

HDFS特殊的设计，在实现上述优良特性的同时，也使得自身具有一些应用局限性，主要包括以下几个方面：

- 不适合低延迟数据访问
- 无法高效存储大量小文件
- 不支持多用户写入及任意修改文件

# 3.3.1块

HDFS默认一个块64MB，一个文件被分成多个块，以块作为存储单位

块的大小远远大于普通文件系统，可以最小化寻址开销

HDFS采用抽象的块概念可以带来以下几个明显的好处：

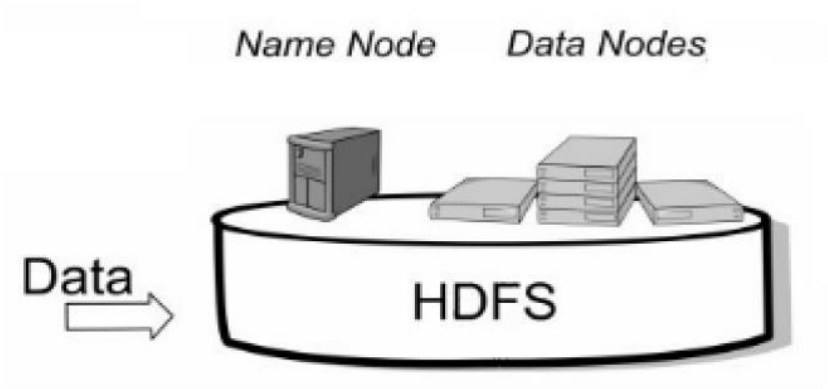
- **支持大规模文件存储**：文件以块为单位进行存储，一个大规模文件可以被分拆成若干个文件块，不同的文件块可以被分发到不同的节点上，因此，一个文件的大小不会受到单个节点的存储容量的限制，可以远远大于网络中任意节点的存储容量

- **简化系统设计**：首先，大大简化了存储管理，因为文件块大小是固定的，这样就可以很容易计算出一个节点可以存储多少文件块；其次，方便了元数据的管理，元数据不需要和文件块一起存储，可以由其他系统负责管理元数据

- **适合数据备份**：每个文件块都可以冗余存储到多个节点上，大大提高了系统的容错性和可用性

# 3.3.2名称节点和数据节点

## HDFS主要组件的功能




metadata

File.txt=
Blk A:
DN1, DN5, DN6

Blk B:
DN7, DN1, DN2

Blk C:
DN5, DN8,DN9

| NameNode | DataNode |
| --- | --- |
| • 存储元数据 | • 存储文件内容 |
| •元数据保存在内存中 | •文件内容保存在磁盘 |
| • 保存文件,block， datanode之间的映射关系 | •维护了block id到datanode本地文件的映射关系 |

# 3.3.2名称节点和数据节点

## 名称节点的数据结构

- 在HDFS中，名称节点（NameNode）负责管理分布式文件系统的命名空间（Namespace），保存了两个核心的数据结构，即FsImage和EditLog
  - FsImage用于维护文件系统树以及文件树中所有的文件和文件夹的元数据
  - 操作日志文件EditLog中记录了所有针对文件的创建、删除、重命名等操作
- 名称节点记录了每个文件中各个块所在的数据节点的位置信息

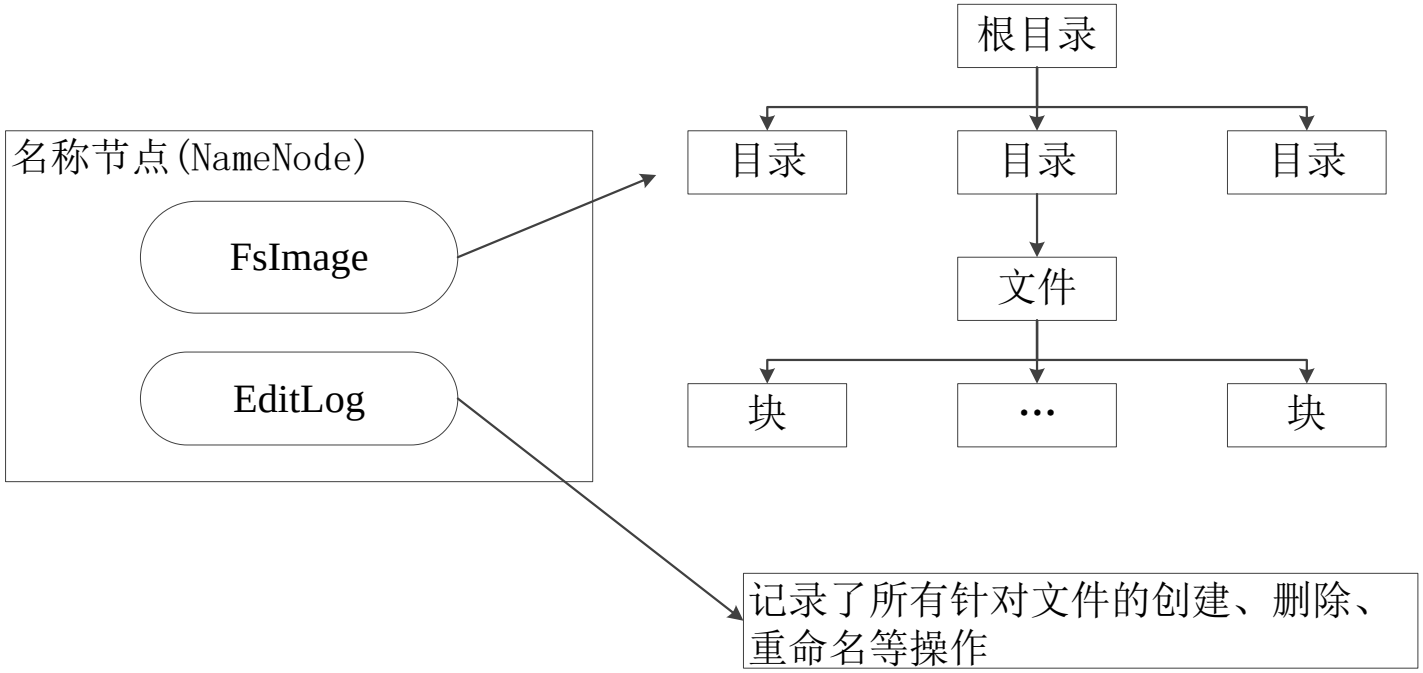



图3-3 名称节点的数据结构

# 3.3.2名称节点和数据节点

## FsImage文件

•FsImage文件包含文件系统中所有目录和文件inode的序列化形式。每个inode是一个文件或目录的元数据的内部表示，并包含此类信息：文件的复制等级、修改和访问时间、访问权限、块大小以及组成文件的块。对于目录，则存储修改时间、权限和配额元数据

•FsImage文件没有记录文件包含哪些块以及每个块存储在哪个数据节点。而是由名称节点把这些映射信息保留在内存中，当数据节点加入HDFS集群时，数据节点会把自己所包含的块列表告知给名称节点，此后会定期执行这种告知操作，以确保名称节点的块映射是最新的。

# 3.3.2名称节点和数据节点

## 名称节点的启动

•在名称节点启动的时候，它会将FsImage文件中的内容加载到内存中，之后再执行EditLog文件中的各项操作，使得内存中的元数据和实际的同步，存在内存中的元数据支持客户端的读操作。

•一旦在内存中成功建立文件系统元数据的映射，则创建一个新的FsImage文件和一个空的EditLog文件

•名称节点起来之后，HDFS中的更新操作会重新写到EditLog文件中，因为FsImage文件一般都很大（GB级别的很常见），如果所有的更新操作都往FsImage文件中添加，这样会导致系统运行的十分缓慢，但是，如果往EditLog文件里面写就不会这样，因为EditLog 要小很多。每次执行写操作之后，且在向客户端发送成功代码之前，edits文件都需要同步更新

# 3.3.2名称节点和数据节点

**名称节点运行期间EditLog不断变大的问题**

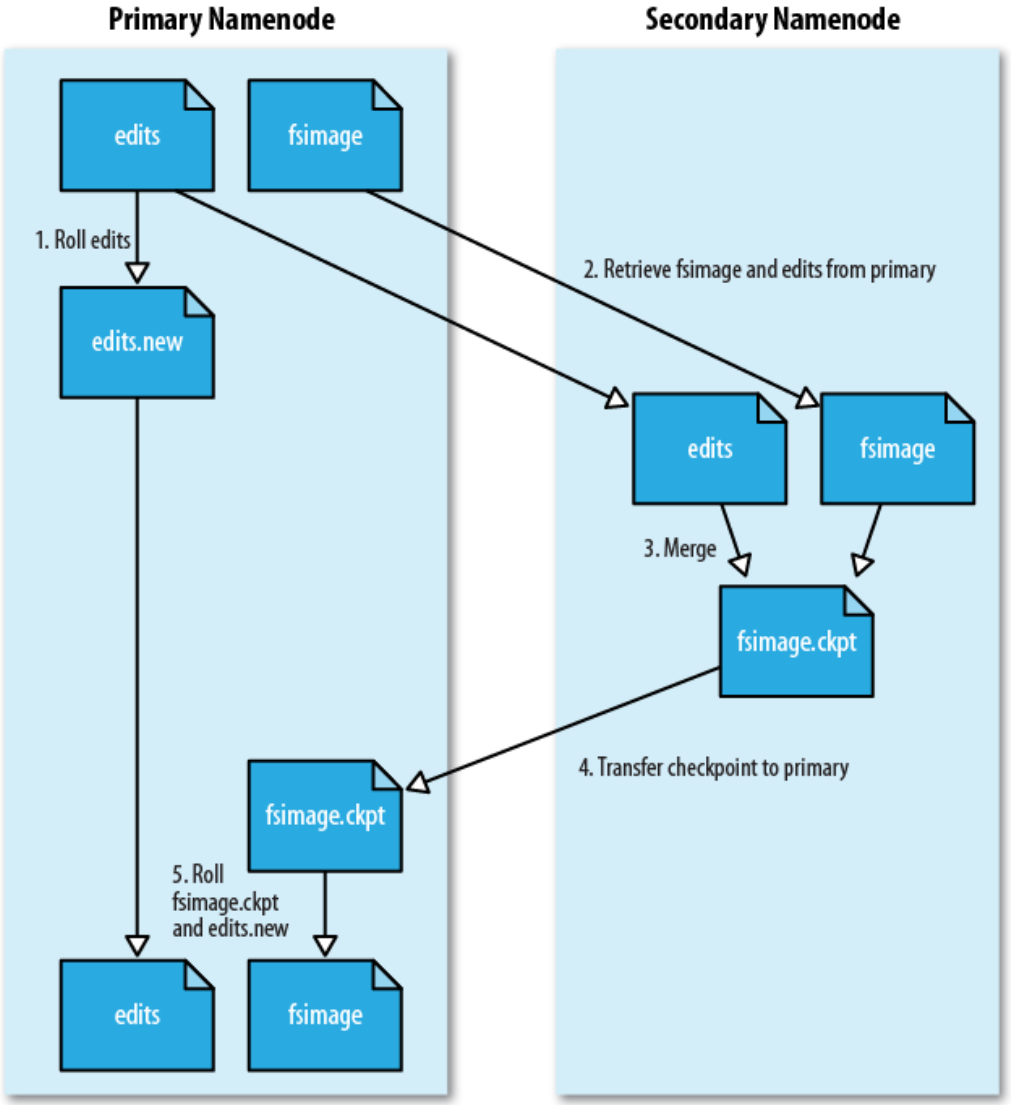
•在名称节点运行期间，HDFS的所有更新操作都是直接写到EditLog中，久而久之， EditLog文件将会变得很大

•虽然这对名称节点运行时候是没有什么明显影响的，但是，当名称节点重启的时候，名称节点需要先将FsImage里面的所有内容映像到内存中，然后再一条一条地执行EditLog中的记录，当EditLog文件非常大的时候，会导致名称节点启动操作非常慢，而在这段时间内HDFS系统处于安全模式，一直无法对外提供写操作，影响了用户的使用

如何解决？答案是：SecondaryNameNode第二名称节点

**第二名称节点**是HDFS架构中的一个组成部分，它是用来保存名称节点中对HDFS元数据信息的备份，并减少名称节点重启的时间。SecondaryNameNode一般是单独运行在一台机器上

# 3.3.2名称节点和数据节点




SecondaryNameNode的工作情况：

（1）SecondaryNameNode会定期和NameNode通信，请求其停止使用EditLog文件，暂时将新的写操作写到一个新的文件edit.new上来，这个操作是瞬间完成，上层写日志的函数完全感觉不到差别；

（2）SecondaryNameNode通过HTTP GET方式从NameNode上获取到FsImage和EditLog文件，并下载到本地的相应目录下；

（3）SecondaryNameNode将下载下来的FsImage载入到内存，然后一条一条地执行EditLog文件中的各项更新操作，使得内存中的FsImage保持最新；这个过程就是EditLog和FsImage文件合并；

（4）SecondaryNameNode执行完（3）操作之后，会通过post方式将新的FsImage文件发送到NameNode节点上

（5）NameNode将从SecondaryNameNode接收到的新的FsImage替换旧的FsImage文件，同时将edit.new替换EditLog文件，通过这个过程EditLog就变小了

# 3.3.2名称节点和数据节点

数据节点（DataNode）

•数据节点是分布式文件系统HDFS的工作节点，负责数据的存储和读取，会根据客户端或者是名称节点的调度来进行数据的存储和检索，并且向名称节点定期发送自己所存储的块的列表

•每个数据节点中的数据会被保存在各自节点的本地Linux文件系统中

# 3.4.1HDFS体系结构概述

- HDFS采用了主从(Master/Slave)结构模型，一个HDFS集群包括一个名称节点和若干个数据节点。名称节点作为中心服务器，每个数据节点周期性地向名称节点发送“心跳”信息，报告自己的状态，没有按时发送心跳信息的数据节点会被标记为“宕机”，不会再给它分配任何I/O请求。

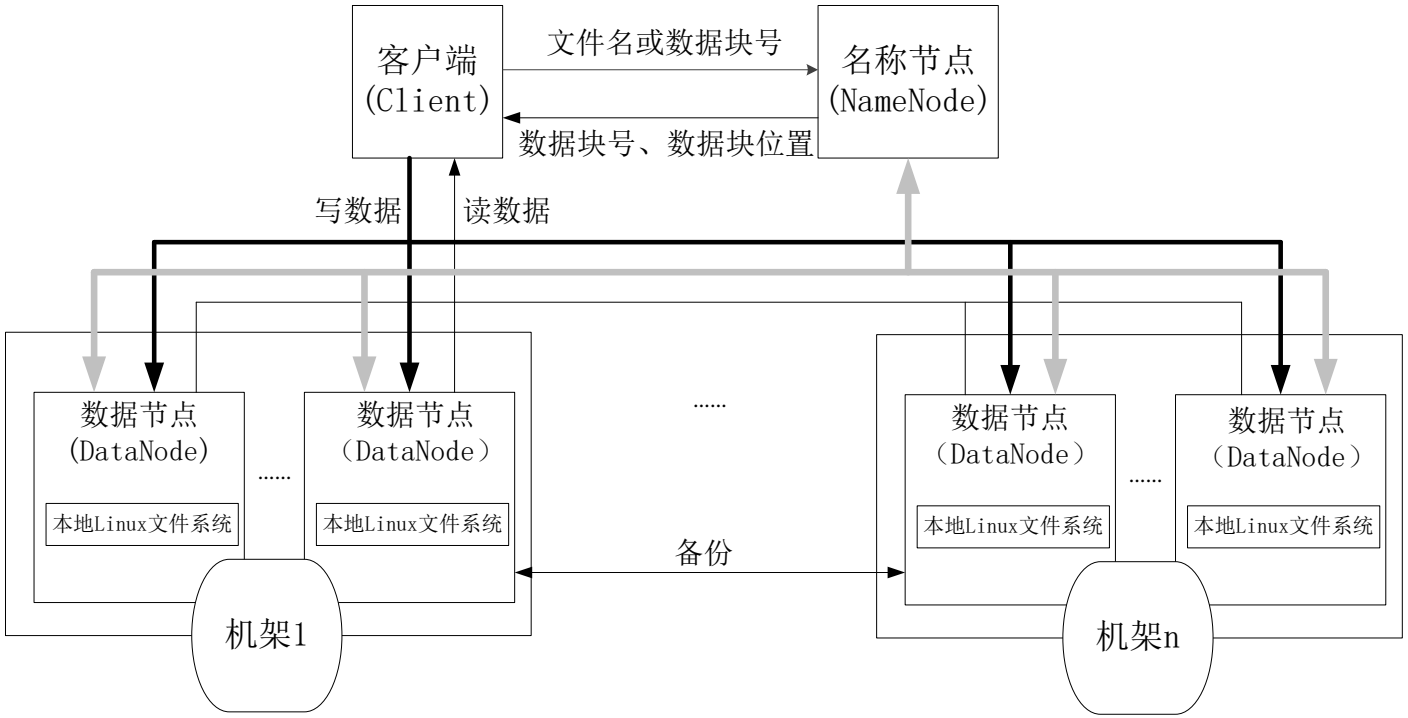



**图3-4 HDFS体系结构**

# 3.5 HDFS存储原理

- 3.5.1 冗余数据保存
- 3.5.2 数据存取策略
- 3.5.3 数据错误与恢复

# 3.5.1冗余数据保存

作为一个分布式文件系统，为了保证系统的容错性和可用性，HDFS采用了多副本方式对数据进行冗余存储，通常一个数据块的多个副本会被分布到不同的数据节点上，如图3-5所示，数据块1被分别存放到数据节点A和C上，数据块2被存放在数据节点A和B上。这种多副本方式具有以下几个优点：

**（1）加快数据传输速度（并行取数据）**

**（2）容易检查数据错误（对照错误）**

**（3）保证数据可靠性（出错后复制）**

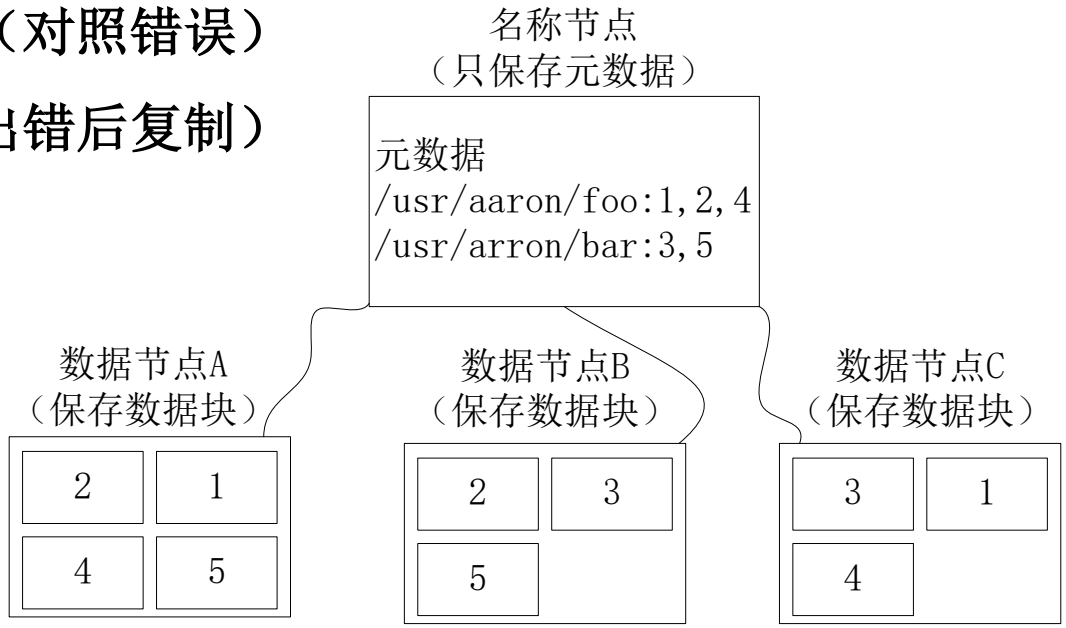


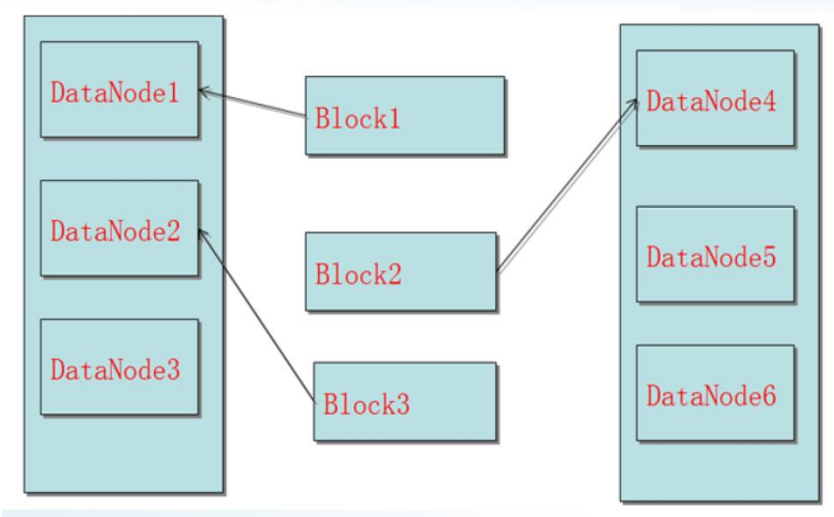

图3-5 HDFS数据块多副本存储

# 3.5.2数据存取策略

**1.数据存放**

•第一个副本：放置在上传文件的数据节点；如果是集群外提交，则随机挑选一台磁盘不太满、CPU不太忙的节点
•第二个副本：放置在与第一个副本不同的机架的节点上
•第三个副本：与第一个副本相同机架的其他节点上
•更多副本：随机节点

Block的副本放置策略



# 3.5.2数据存取策略

## 2. 数据读取

•HDFS提供了一个API可以确定一个数据节点所属的机架ID，客户端也可以调用API获取自己所属的机架ID

•当客户端读取数据时，从名称节点获得数据块不同副本的存放位置列表，列表中包含了副本所在的数据节点，可以调用API来确定客户端和这些数据节点所属的机架ID，当发现某个数据块副本对应的机架ID和客户端对应的机架ID相同时，就优先选择该副本读取数据，如果没有发现，就随机选择一个副本读取数据

# 3.5.3数据错误与恢复

HDFS具有较高的容错性，可以兼容廉价的硬件，它把硬件出错看作一种常态，而不是异常，并设计了相应的机制检测数据错误和进行自动恢复，主要包括以下几种情形：名称节点出错、数据节点出错和数据出错。

**1. 名称节点出错**

名称节点保存了所有的元数据信息，其中，最核心的两大数据结构是FsImage和Editlog，如果这两个文件发生损坏，那么整个HDFS实例将失效。因此，HDFS设置了备份机制，把这些核心文件同步复制到备份服务器SecondaryNameNode上。当名称节点出错时，就可以根据备份服务器SecondaryNameNode中的FsImage和Editlog数据进行恢复。

# 3.5.3数据错误与恢复

**2. 数据节点出错**

- 每个数据节点会定期向名称节点发送“心跳”信息，向名称节点报告自己的状态
- 当数据节点发生故障，或者网络发生断网时，名称节点就无法收到来自一些数据节点的心跳信息，这时，这些数据节点就会被标记为“宕机”，节点上面的所有数据都会被标记为“不可读”，名称节点不会再给它们发送任何I/O请求
- 这时，有可能出现一种情形，即由于一些数据节点的不可用，会导致一些数据块的副本数量小于冗余因子
- 名称节点会定期检查这种情况，一旦发现某个数据块的副本数量小于冗余因子，就会启动数据冗余复制，为它生成新的副本
- HDFS和其它分布式文件系统的最大区别就是可以调整冗余数据的位置

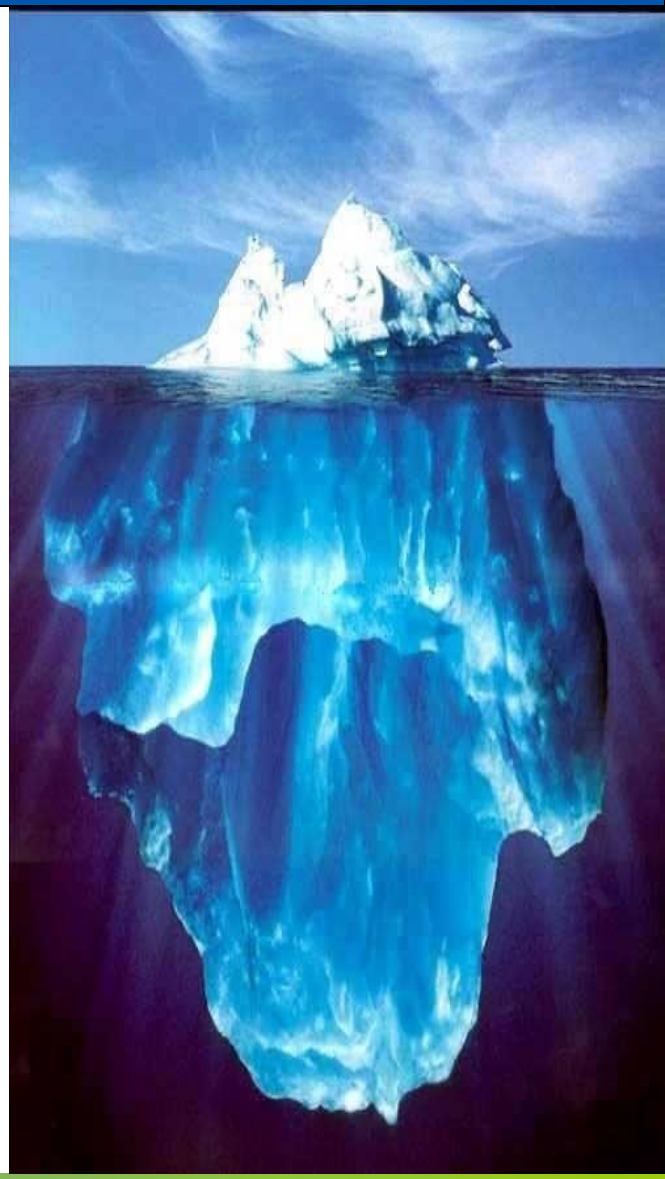
# 3.5.3数据错误与恢复

## 3. 数据出错

- 网络传输和磁盘错误等因素，都会造成数据错误
- 客户端在读取到数据后，会采用md5和sha1对数据块进行校验，以确定读取到正确的数据
- 在文件被创建时，客户端就会对每一个文件块进行信息摘录，并把这些信息写入到同一个路径的隐藏文件里面
- 当客户端读取文件的时候，会先读取该信息文件，然后，利用该信息文件对每个读取的数据块进行校验，如果校验出错，客户端就会请求到另外一个数据节点读取该文件块，并且向名称节点报告这个文件块有错误，名称节点会定期检查并且重新复制这个块

# 第四章 分布式数据库HBase

- 4.1 概述
- 4.2 HBase访问接口
- 4.3 HBase数据模型
- 4.4 HBase的实现原理
- 4.5 HBase运行机制
- 4.6 HBase应用方案
- 4.7 HBase编程实践（Shell)



# 4.1.2HBase简介

关系数据库已经流行很多年，并且Hadoop已经有了HDFS和MapReduce，为什么需要HBase?

- Hadoop可以很好地解决大规模数据的离线批量处理问题，但是，受限于Hadoop MapReduce编程框架的高延迟数据处理机制，使得Hadoop无法满足大规模数据实时处理应用的需求
- HDFS面向批量访问模式，不是随机访问模式
- 传统的通用关系型数据库无法应对在数据规模剧增时导致的系统扩展性和性能问题（分库分表也不能很好解决）
- 传统关系数据库在数据结构变化时一般需要停机维护；空列浪费存储空间
- 因此，业界出现了一类面向半结构化数据存储和处理的高可扩展、低写入/查询延迟的系统，例如，键值数据库、文档数据库和列族数据库（如BigTable和HBase等）
- HBase已经成功应用于互联网服务领域和传统行业的众多在线式数据分析处理系统中

# 4.1.3HBase与传统关系数据库的对比分析

- HBase与传统的关系数据库的区别主要体现在以下几个方面：
- （1）数据类型：关系数据库采用关系模型，具有丰富的数据类型和存储方式，HBase则采用了更加简单的数据模型，它把数据存储为未经解释的字符串
- （2）数据操作：关系数据库中包含了丰富的操作，其中会涉及复杂的多表连接。HBase操作则不存在复杂的表与表之间的关系，只有简单的插入、查询、删除、清空等，因为HBase在设计上就避免了复杂的表和表之间的关系
- （3）存储模式：关系数据库是基于行模式存储的。HBase是基于列存储的，每个列族都由几个文件保存，不同列族的文件是分离的

# 4.1.3HBase与传统关系数据库的对比分析

- HBase与传统的关系数据库的区别主要体现在以下几个方面：
- （4）数据索引：关系数据库通常可以针对不同列构建复杂的多个索引，以提高数据访问性能。HBase只有一个索引——行键，通过巧妙的设计，HBase中的所有访问方法，或者通过行键访问，或者通过行键扫描，从而使得整个系统不会慢下来
- （5）数据维护：在关系数据库中，更新操作会用最新的当前值去替换记录中原来的旧值，旧值被覆盖后就不会存在。而在HBase中执行更新操作时，并不会删除数据旧的版本，而是生成一个新的版本，旧有的版本仍然保留
- （6）可伸缩性：关系数据库很难实现横向扩展，纵向扩展的空间也比较有限。相反，HBase和BigTable这些分布式数据库就是为了实现灵活的水平扩展而开发的，能够轻易地通过在集群中增加或者减少硬件数量来实现性能的伸缩

# 4.1.3HBase与传统关系数据库的对比分析

| | HBase | RDBMS |
|---|---|---|
| 数据类型 | 只有字符串 | 丰富的数据类型 |
| 数据操作 | 简单的增删改查 | 各种各样的函数，表连接 |
| 存储模式 | 基于列存储 | 基于表格结构和行存储 |
| 数据索引 | 只有一个索引 | 可以有多个 |
| 数据保护 | 更新后旧版本仍然会保留 | 替换 |
| 可伸缩性 | 轻易的进行增加节点，可扩展性高 | 需要中间层 |

# 4.3.1数据模型概述

- HBase是一个稀疏、多维度、排序的映射表，这张表的索引是行键、列族、列限定符和时间戳
- 每个值是一个未经解释的字符串，没有数据类型
- 用户在表中存储数据，每一行都有一个可排序的行键和任意多的列
- 表在水平方向由一个或者多个列族组成，一个列族中可以包含任意多个列，同一个列族里面的数据存储在一起
- 列族支持动态扩展，可以很轻松地添加一个列族或列，无需预先定义列的数量以及类型，所有列均以字符串形式存储，用户需要自行进行数据类型转换
- HBase中执行更新操作时，并不会删除数据旧的版本，而是生成一个新的版本，旧有的版本仍然保留（这是和HDFS只允许追加不允许修改的特性相关的）

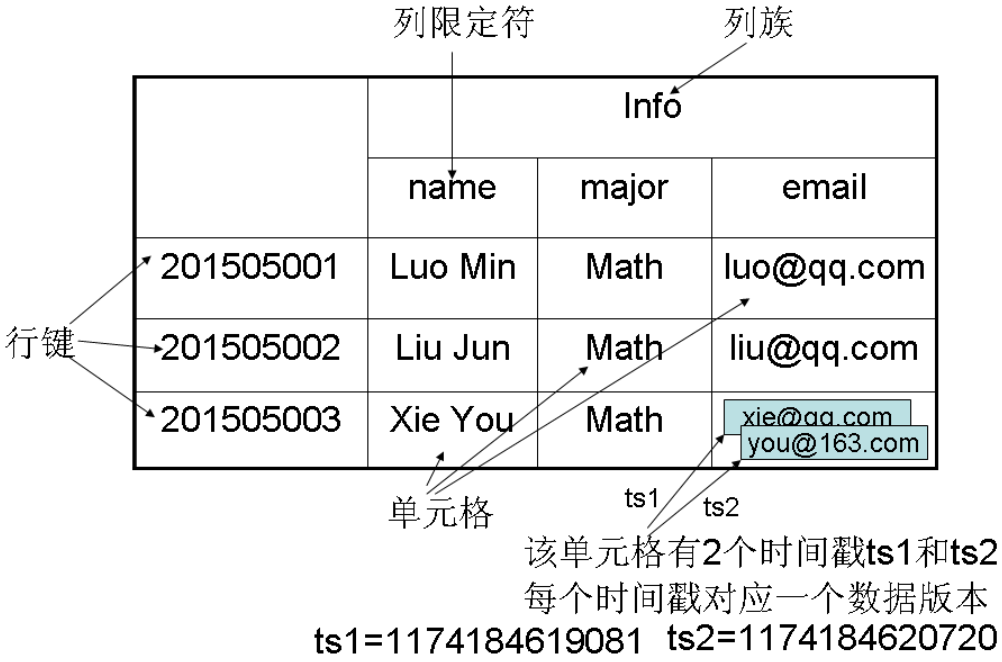
# 4.3.2数据模型相关概念

- 表：HBase采用表来组织数据，表由行和列组成，列划分为若干个列族
- 行：每个HBase表都由若干行组成，每个行由行键（row key）来标识。
- 列族：一个HBase表被分组成许多“列族”（Column Family）的集合，它是基本的访问控制单元
- 列限定符：列族里的数据通过列限定符（或列）来定位
- 单元格：在HBase表中，通过行、列族和列限定符确定一个“单元格”（cell），单元格中存储的数据没有数据类型，总被视为字节数组byte[]
- 时间戳：每个单元格都保存着同一份数据的多个版本，这些版本采用时间戳进行索引

# 4.3.3数据坐标

- HBase中需要根据行键、列族、列限定符和时间戳来确定一个单元格，因此，可以视为一个“四维坐标”，即[行键, 列族, 列限定符, 时间戳]

| 键 | 值 |
| --- | --- |
| [“201505003”, “Info”, “email”, 1174184619081] | “xie@qq.com” |
| [“201505003”, “Info”, “email”, 1174184620720] | “you@163.com” |



# 4.4 HBase的实现原理

- 4.4.1 HBase功能组件
- 4.4.2 表和Region
- 4.4.3 Region的定位

# 4.4.2表和Region

- 1、Table中的所有行都按照row key的字典序排列。
- 2、Table 在行的方向上分割为多个Region。

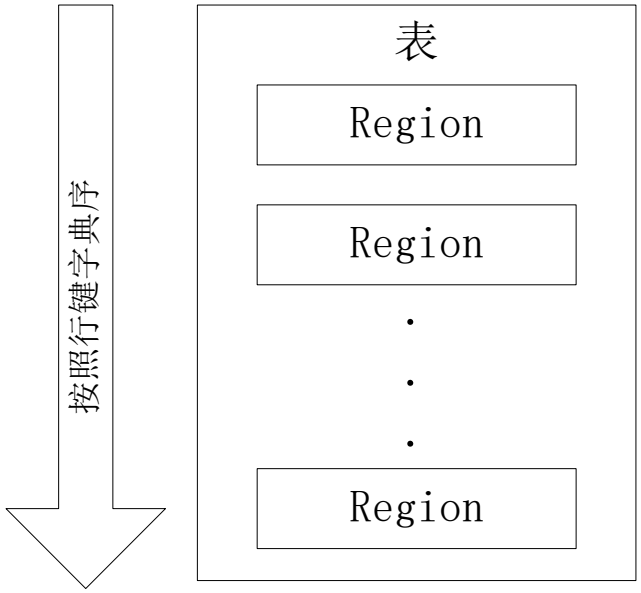



图4-5一个HBase表被划分成多个Region

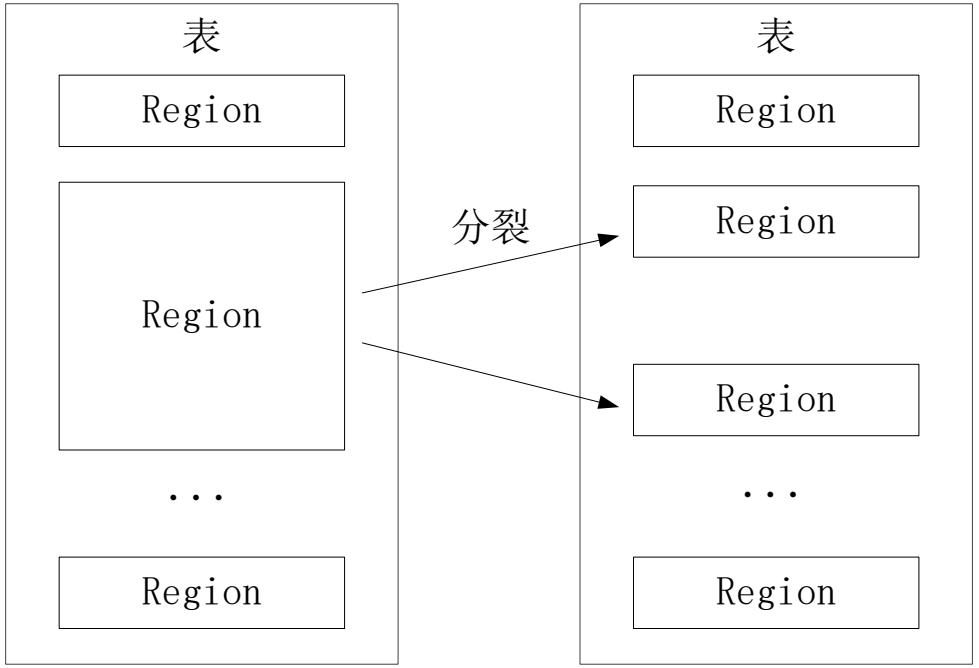



图4-6 一个Region会分裂成多个新的Region

# 4.4.2表和Region

- Region是HBase中分布式存储和负载均衡的最小单元
- 每个Region默认大小是100MB到200MB（2006年以前的硬件配置）
  - 每个Region的最佳大小取决于单台服务器的有效处理能力
  - 目前每个Region最佳大小建议1GB-2GB（2013年以后的硬件配置）
- 同一个Region不会被分拆到多个Region服务器
- 每个Region服务器存储10-1000个Region

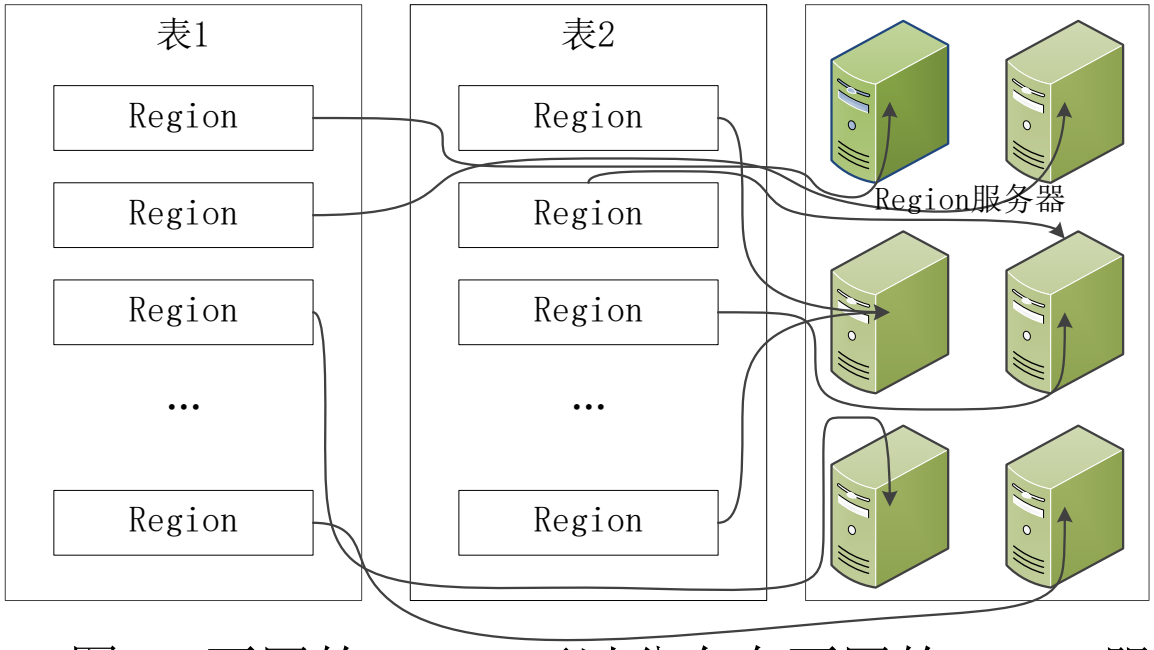



图4-7 不同的Region可以分布在不同的Region服务器上

# 4.4.3Region的定位

- 元数据表，又名.META.表，存储了Region和Region服务器的映射关系
- 当HBase表很大时， .META.表也会被分裂成多个Region
- 根数据表，又名-ROOT-表，记录所有元数据的具体位置
- -ROOT-表只有唯一一个Region，名字是在程序中被写死的
- Zookeeper文件记录了-ROOT-表的位置

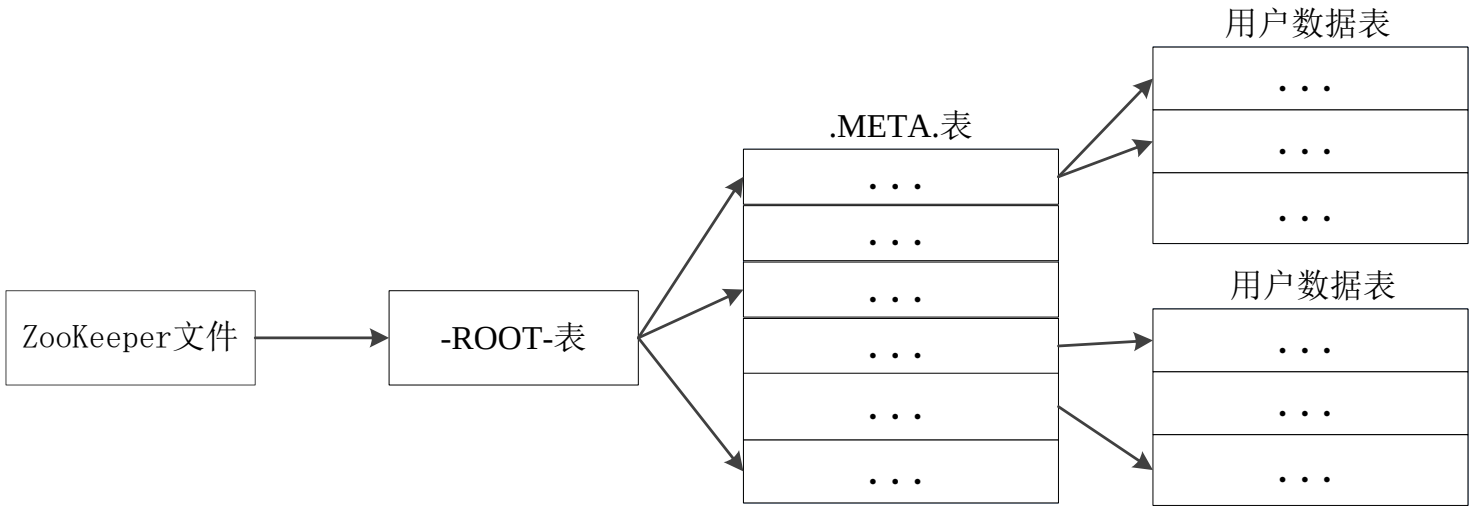



图4-8 HBase的三层结构

# 4.4.3Region的定位

表4-6 HBase的三层结构中各层次的名称和作用

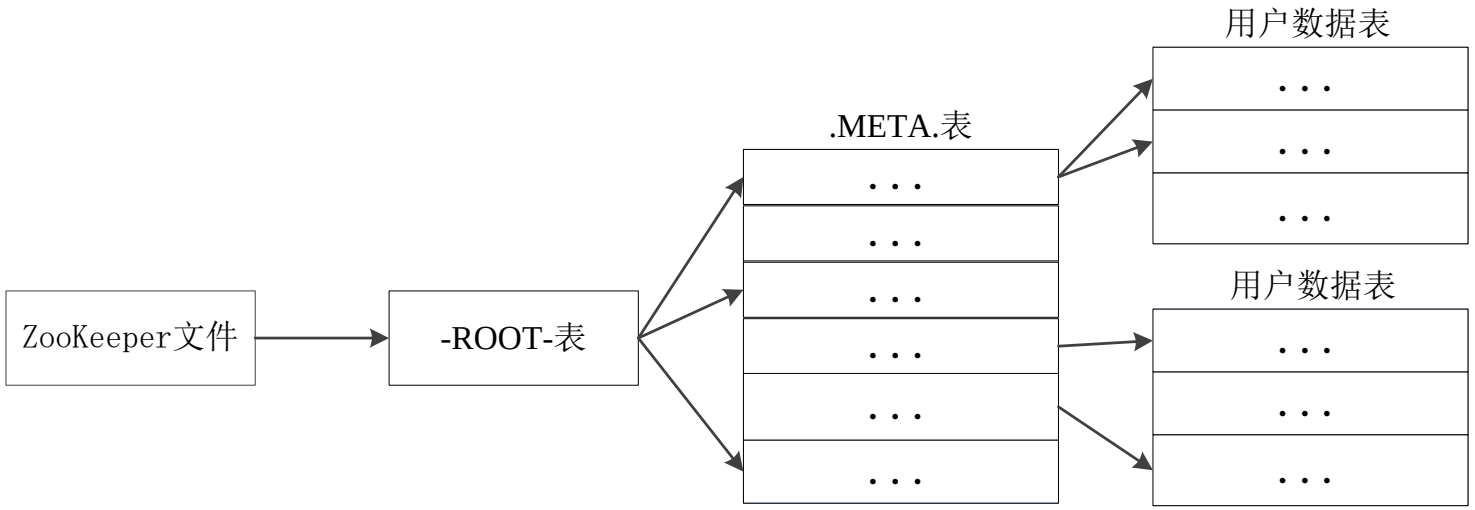
| 层次 | 名称 | 作用 |
| --- | --- | --- |
| 第一层 | Zookeeper文件 | 记录了-ROOT-表的位置信息 |
| 第二层 | -ROOT-表 | 记录了.META.表的Region位置信息<br>-ROOT-表只能有一个Region。通过-ROOT-表，就可以访问.META.表中的数据 |
| 第三层 | .META.表 | 记录了用户数据表的Region位置信息，<br>.META.表可以有多个Region，保存了HBase中所有用户数据表的Region位置信息 |

# 4.4.3Region的定位

客户端访问数据时的“三级寻址”

•为了加速寻址，客户端会缓存位置信息，同时，需要解决缓存失效问题

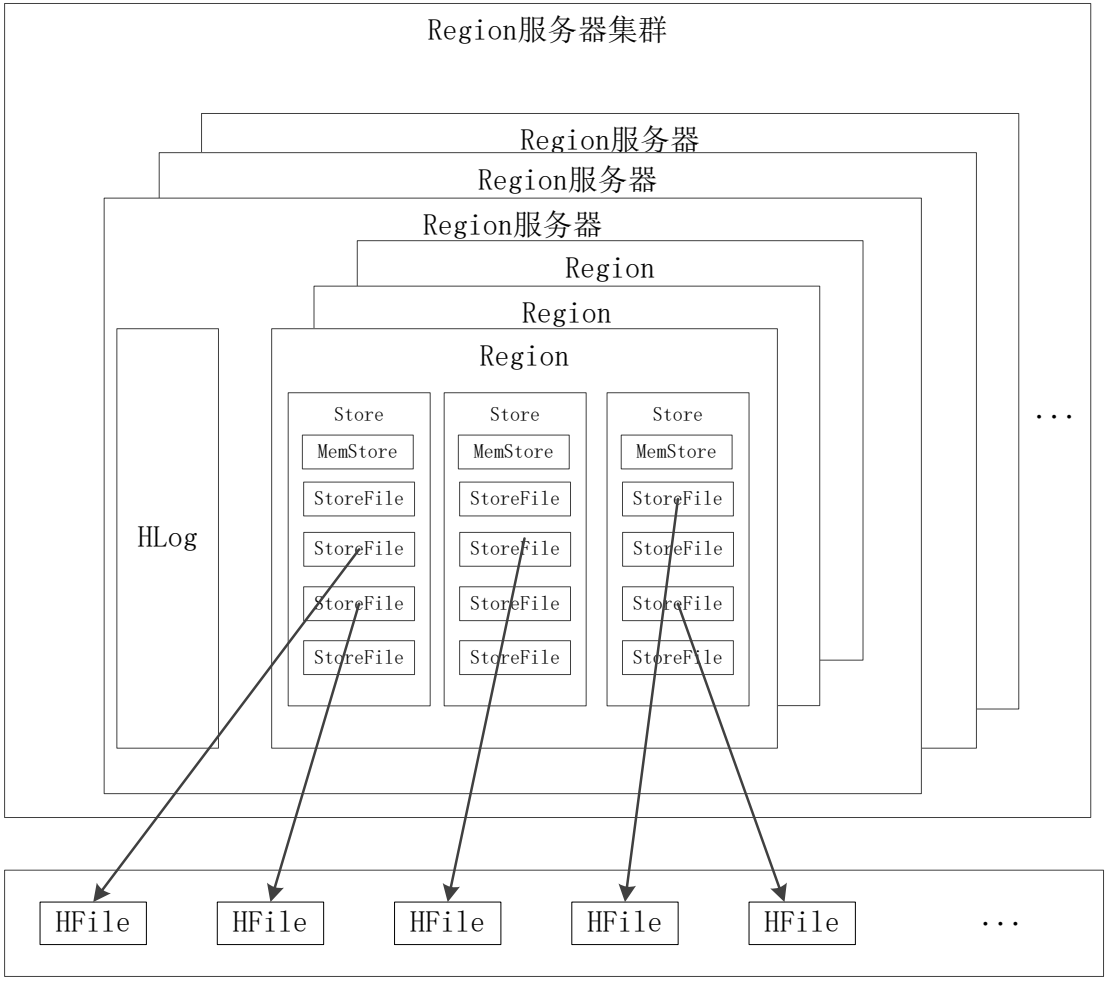
•寻址过程客户端只需要询问Zookeeper服务器，不需要连接Master服务器



# 4.5.1HBase系统架构

- 4. Region服务器
  - Region服务器是HBase中最核心的模块
  - Region server维护region，处理对这些region的IO请求
  - Region server负责切分在运行过程中变得过大的region

# 4.5.2Region服务器工作原理




1. 用户读写数据过程
2. 缓存的刷新
3. StoreFile的合并

图4-10 Region服务器向HDFS文件系统中读写数据

# 4.5.2Region服务器工作原理

**1. 用户读写数据过程**

- 用户写入数据时，被分配到相应Region服务器去执行
- 用户数据首先被写入到MemStore和Hlog中
- 当用户读取数据时，Region服务器会首先访问MemStore缓存，如果找不到，再去磁盘上面的StoreFile中寻找

# 4.5.2Region服务器工作原理

**2. 缓存的刷新**

•系统会周期性地把MemStore缓存里的内容刷写到磁盘的StoreFile文件中，清空缓存，并在Hlog里面写入一个标记
•每次刷写都生成一个新的StoreFile文件，因此，每个Store包含多个StoreFile文件

•每个Region服务器都有一个自己的HLog 文件，每次**启动**都检查该文件，确认最近一次执行缓存刷新操作之后是否发生新的写入操作；如果发现更新，则先写入MemStore，再刷写到StoreFile，最后删除旧的Hlog文件，开始为用户提供服务

# 4.5.2Region服务器工作原理

**3. StoreFile的合并**

- 每次刷写都生成一个新的StoreFile，数量太多，影响查找速度
- 调用Store.compact()把多个合并成一个
- 合并操作比较耗费资源，只有数量达到一个阈值才启动合并

# 4.5.3Store工作原理

- Store是Region服务器的核心
- 多个StoreFile合并成一个
- 单个StoreFile过大时，又触发分裂操作，1个父Region被分裂成两个子Region

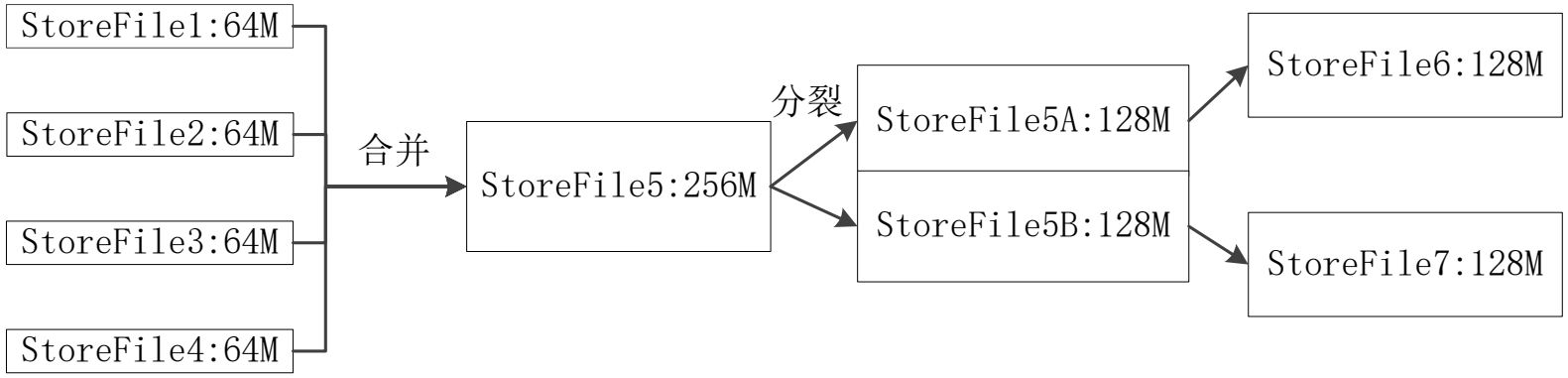



图4-11 StoreFile的合并和分裂过程

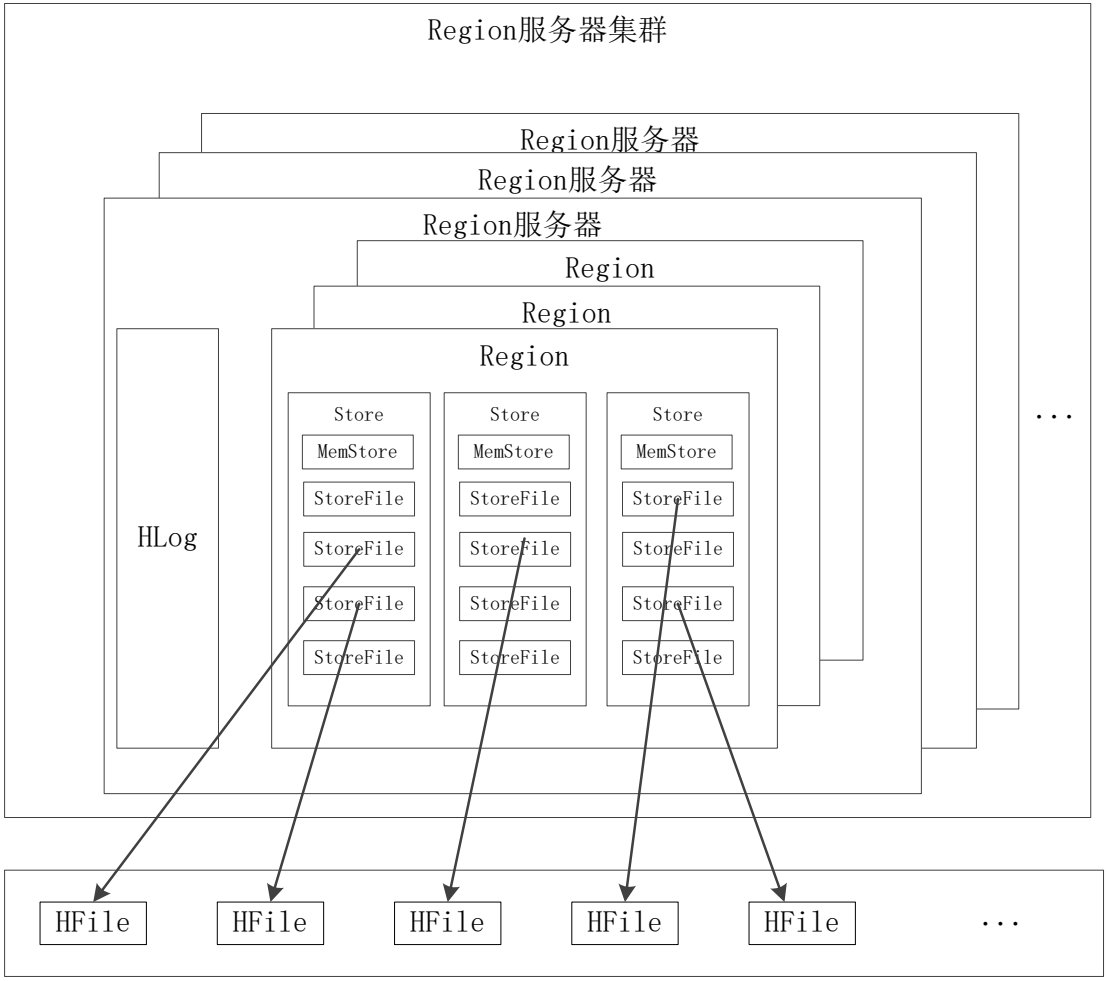
# 4.5.4HLog工作原理

- 分布式环境必须要考虑系统出错。HBase采用HLog保证系统恢复
- HBase系统为每个Region服务器配置了一个HLog文件，它是一种预写式日志（Write Ahead Log）
- 用户更新数据必须首先写入日志后，才能写入MemStore缓存，并且，直到MemStore缓存内容对应的日志已经写入磁盘，该缓存内容才能被刷写到磁盘

# 4.5.4HLog工作原理

- Zookeeper会实时监测每个Region服务器的状态，当某个Region服务器发生故障时，Zookeeper会通知Master
- Master首先会处理该故障Region服务器上面遗留的HLog文件，这个遗留的HLog文件中包含了来自多个Region对象的日志记录
- 系统会根据每条日志记录所属的Region对象对HLog数据进行拆分，分别放到相应Region对象的目录下，然后，再将失效的Region重新分配到可用的Region服务器中，并把与该Region对象相关的HLog日志记录也发送给相应的Region服务器
- Region服务器领取到分配给自己的Region对象以及与之相关的HLog日志记录以后，会重新做一遍日志记录中的各种操作，把日志记录中的数据写入到MemStore缓存中，然后，刷新到磁盘的StoreFile文件中，完成数据恢复
- 共用日志优点：提高对表的写操作性能；缺点：恢复时需要分拆日志
- 为何这么设计：发生故障的概率是小部分，不能牺牲平时正常的读写性能

# 4.5.2Region服务器工作原理




1. 用户读写数据过程
2. 缓存的刷新
3. StoreFile的合并

图4-10 Region服务器向HDFS文件系统中读写数据

# 4.7.2HBase常用Shell命令

- create：创建表
- list：列出HBase中所有的表信息

例子1：创建一个表，该表名称为tempTable，包含3个列族f1，f2和f3

```
hbase(main):002:0> create 'tempTable', 'f1', 'f2', 'f3'
0 row(s) in 1.3560 seconds

hbase(main):003:0> list
TABLE
tempTable
testTable
wordcount
3 row(s) in 0.0350 seconds
```

备注：后面的例子都在此基础上继续操作

# 4.7.2HBase常用Shell命令

put：向表、行、列指定的单元格添加数据

一次只能为一个表的一行数据的一个列添加一个数据

scan：浏览表的相关信息

例子2：继续向表tempTable中的第r1行、第“f1:c1”列，添加数据值为“hello,dblab”

```
hbase(main):005:0> put 'tempTable', 'r1', 'f1:c1', 'hello, dblab'
0 row(s) in 0.0240 seconds

hbase(main):006:0> scan 'tempTable'
ROW                           COLUMN+CELL
 r1                           column=f1:c1, timestamp=1430036599391, value=hello, dblab
1 row(s) in 0.0160 seconds
```

在添加数据时，HBase会自动为添加的数据添加一个时间戳，当然，也可以在添加数据时人工指定时间戳的值

# 4.7.2HBase常用Shell命令

get：通过表名、行、列、时间戳、时间范围和版本号来获得相应单元格的值

例子3：
（1）从tempTable中，获取第r1行、第“f1:c1”列的值
（2）从tempTable中，获取第r1行、第“f1:c3”列的值

备注：f1是列族，c1和c3都是列

```
hbase(main):012:0> get 'tempTable', 'r1', {COLUMN=>'f1:c1'}
COLUMN                          CELL
 f1:c1                          timestamp=1430036599391, value=hello, dblab
1 row(s) in 0.0090 seconds

hbase(main):013:0> get 'tempTable', 'r1', {COLUMN=>'f1:c3'}
COLUMN                          CELL
0 row(s) in 0.0030 seconds
```

从运行结果可以看出： tempTable中第r1行、第“f1:c3”列的值当前不存在

# 4.7.2HBase常用Shell命令

- enable/disable：使表有效或无效
- drop：删除表

例子4：使表tempTable无效、删除该表

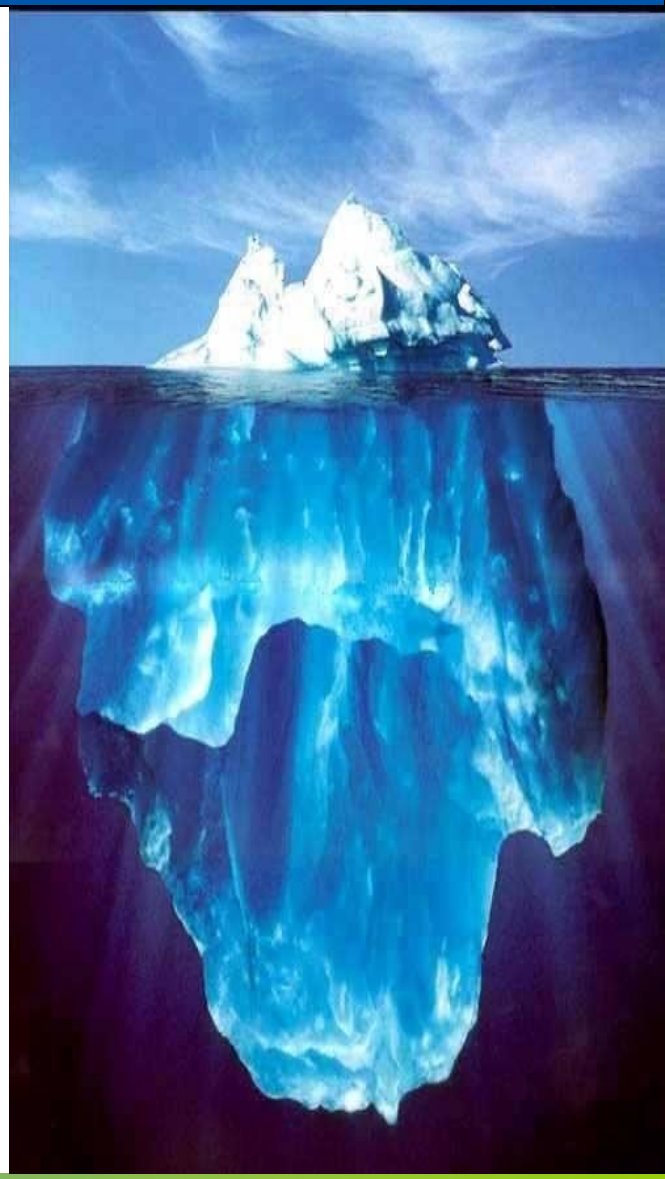
```
hbase(main):016:0> disable 'tempTable'
0 row(s) in 1.3720 seconds

hbase(main):017:0> drop 'tempTable'
0 row(s) in 1.1350 seconds

hbase(main):018:0> list
TABLE
testTable
wordcount
2 row(s) in 0.0370 seconds
```

# 第五章 NoSQL数据库

- 5.1 NoSQL简介
- 5.2 NoSQL兴起的原因
- 5.3 NoSQL与关系数据库的比较
- 5.4 NoSQL的四大类型
- 5.5 NoSQL的三大基石
- 5.6 从NoSQL到NewSQL数据库
- 5.7 文档数据库MongoDB



# 5.2 NoSQL兴起的原因

1、关系数据库已经无法满足Web2.0的需求。主要表现在以下几个方面：

**（1）无法满足海量数据的管理需求**

**（2）无法满足数据高并发的需求**

**（3）无法满足高可扩展性和高可用性的需求**

# 5.2 NoSQL兴起的原因

2、“One size fits all”模式很难适用于截然不同的业务场景

•关系模型作为统一的数据模型既被用于数据分析，也被用于在线业务。但这两者一个强调高吞吐，一个强调低延时，已经演化出完全不同的架构。用同一套模型来抽象显然是不合适的

•Hadoop就是针对数据分析，批处理

•MongoDB、Redis等是针对在线业务，两者都抛弃了关系模型

# 5.2 NoSQL兴起的原因

3、关系数据库的关键特性包括完善的事务机制和高效的查询机制。但是，关系数据库引以为傲的两个关键特性，到了Web2.0时代却成了鸡肋，主要表现在以下几个方面：

**（1）Web2.0网站系统通常不要求严格的数据库事务**

**（2）Web2.0并不要求严格的读写实时性**

**（3）Web2.0通常不包含大量复杂的SQL查询（去结构化，存储空间换取更好的查询性能）**

# 5.3 NoSQL与关系数据库的比较

表5-1 NoSQL和关系数据库的简单比较

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 数据库原理 | 完全支持 | 部分支持 | RDBMS有关系代数理论作为基础<br>NoSQL没有统一的理论基础 |
| 数据规模 | 大 | 超大 | RDBMS很难实现横向扩展，纵向扩展的空间也比较有限，性能会随着数据规模的增大而降低<br>NoSQL可以很容易通过添加更多设备来支持更大规模的数据 |
| 数据库模式 | 固定 | 灵活 | RDBMS需要定义数据库模式，严格遵守数据定义和相关约束条件<br>NoSQL不存在数据库模式，可以自由灵活定义并存储各种不同类型的数据 |
| 查询效率 | 快 | 可以实现高效的简单查询，但是不具备高度结构化查询等特性，复杂查询的性能不尽人意 | RDBMS借助于索引机制可以实现快速查询（包括记录查询和范围查询）<br>很多NoSQL数据库没有面向复杂查询的索引，虽然NoSQL可以使用MapReduce来加速查询，但是，在复杂查询方面的性能仍然不如RDBMS |

# 5.3 NoSQL与关系数据库的比较

表5-1 NoSQL和关系数据库的简单比较（续）

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 一致性 | 强一致性 | 弱一致性 | RDBMS严格遵守事务ACID模型，可以保证事务强一致性<br>很多NoSQL数据库放松了对事务ACID四性的要求，而是遵守BASE模型，只能保证最终一致性 |
| 数据完整性 | 容易实现 | 很难实现 | 任何一个RDBMS都可以很容易实现数据完整性，比如通过主键或者非空约束来实现实体完整性，通过主键、外键来实现参照完整性，通过约束或者触发器来实现用户自定义完整性<br>但是，在NoSQL数据库却无法实现 |
| 扩展性 | 一般 | 好 | RDBMS很难实现横向扩展，纵向扩展的空间也比较有限<br>NoSQL在设计之初就充分考虑了横向扩展的需求，可以很容易通过添加廉价设备实现扩展 |
| 可用性 | 好 | 很好 | RDBMS在任何时候都以保证数据一致性为优先目标，其次才是优化系统性能，随着数据规模的增大，RDBMS为了保证严格的一致性，只能提供相对较弱的可用性<br>大多数NoSQL都能提供较高的可用性 |

# 5.3 NoSQL与关系数据库的比较

表5-1 NoSQL和关系数据库的简单比较（续）

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 标准化 | 是 | 否 | RDBMS已经标准化（SQL）<br>NoSQL还没有行业标准，不同的NoSQL数据库都有自己的查询语言，很难规范应用程序接口<br>StoneBraker认为：NoSQL缺乏统一查询语言，将会拖慢NoSQL发展 |
| 技术支持 | 高 | 低 | RDBMS经过几十年的发展，已经非常成熟，Oracle等大型厂商都可以提供很好的技术支持<br>NoSQL在技术支持方面仍然处于起步阶段，还不成熟，缺乏有力的技术支持 |
| 可维护性 | 复杂 | 复杂 | RDBMS需要专门的数据库管理员(DBA)维护<br>NoSQL数据库虽然没有DBMS复杂，也难以维护 |

## 总结

**（1）关系数据库**

**优势：**以完善的关系代数理论作为基础，有严格的标准，支持事务ACID四性，借助索引机制可以实现高效的查询，技术成熟，有专业公司的技术支持

**劣势：**可扩展性较差，无法较好支持海量数据存储，数据模型过于死板、无法较好支持Web2.0应用，事务机制影响了系统的整体性能等

**（2）NoSQL数据库**

**优势：**可以支持超大规模数据存储，灵活的数据模型可以很好地支持Web2.0应用，具有强大的横向扩展能力等

**劣势：**缺乏数学理论基础，复杂查询性能不高，大都不能实现事务强一致性，很难实现数据完整性，技术尚不成熟，缺乏专业团队的技术支持，维护较困难等

# 5.4.1 键值数据库

| | |
|---|---|
| 相关产品 | Redis、Riak、SimpleDB、Chordless、Scalaris、Memcached |
| 数据模型 | 键/值对<br>键是一个字符串对象<br>值可以是任意类型的数据，比如整型、字符型、数组、列表、集合等 |
| 典型应用 | 涉及频繁读写、拥有简单数据模型的应用<br>内容缓存，比如会话、配置文件、参数、购物车等<br>存储配置和用户数据信息的移动应用 |
| 优点 | 扩展性好，灵活性好，大量写操作时性能高 |
| 缺点 | 无法存储结构化信息，条件查询效率较低 |
| 不适用情形 | 不是通过键而是通过值来查：键值数据库根本没有通过值查询的途径<br>需要存储数据之间的关系：在键值数据库中，不能通过两个或两个以上的键来关联数据<br>需要事务的支持：在一些键值数据库中，产生故障时，不可以回滚 |
| 使用者 | 百度云数据库（Redis）、GitHub（Riak）、BestBuy（Riak）、Twitter（Redis和Memcached）、StackOverFlow（Redis）、Instagram（Redis）、Youtube（Memcached）、Wikipedia（Memcached） |

# 5.4.2 列族数据库

| | |
|---|---|
| 相关产品 | BigTable、HBase、Cassandra、HadoopDB、GreenPlum、PNUTS |
| 数据模型 | 列族 |
| 典型应用 | 分布式数据存储与管理<br>数据在地理上分布于多个数据中心的应用程序<br>可以容忍副本中存在短期不一致情况的应用程序<br>拥有动态字段的应用程序<br>拥有潜在大量数据的应用程序，大到几百TB的数据 |
| 优点 | 查找速度快，可扩展性强，容易进行分布式扩展，复杂性低 |
| 缺点 | 功能较少，大都不支持强事务一致性 |
| 不适用情形 | 需要ACID事务支持的情形，Cassandra等产品就不适用 |
| 使用者 | Ebay（Cassandra）、Instagram（Cassandra）、NASA（Cassandra）、Twitter（Cassandra and HBase）、Facebook（HBase）、Yahoo!（HBase） |

# 5.4.3 文档数据库

| | |
|---|---|
| 相关产品 | MongoDB、CouchDB、Terrastore、ThruDB、RavenDB、SisoDB、RaptorDB、CloudKit、Perservere、Jackrabbit |
| 数据模型 | 键/值<br>值（value）是版本化的文档 |
| 典型应用 | 存储、索引并管理面向文档的数据或者类似的半结构化数据<br>比如，用于后台具有大量读写操作的网站、使用JSON数据结构的应用、使用嵌套结构等非规范化数据的应用程序 |
| 优点 | 性能好（高并发），灵活性高，复杂性低，数据结构灵活<br>提供嵌入式文档功能，将经常查询的数据存储在同一个文档中<br>既可以根据键来构建索引，也可以根据内容构建索引 |
| 缺点 | 缺乏统一的查询语法 |
| 不适用情形 | 在不同的文档上添加事务。文档数据库并不支持文档间的事务，如果对这方面有需求则不应该选用这个解决方案 |
| 使用者 | 百度云数据库（MongoDB）、SAP （MongoDB）、Codecademy（MongoDB）、Foursquare （MongoDB）、NBC News（RavenDB） |

# 5.4.4 图形数据库

| | |
|---|---|
| 相关产品 | Neo4J、OrientDB、InfoGrid、Infinite Graph、GraphDB |
| 数据模型 | 图结构 |
| 典型应用 | 专门用于处理具有高度相互关联关系的数据，比较适合于社交网络、模式识别、依赖分析、推荐系统以及路径寻找等问题 |
| 优点 | 灵活性高，支持复杂的图形算法，可用于构建复杂的关系图谱 |
| 缺点 | 复杂性高，只能支持一定的数据规模 |
| 使用者 | Adobe（Neo4J）、Cisco（Neo4J）、T-Mobile（Neo4J） |

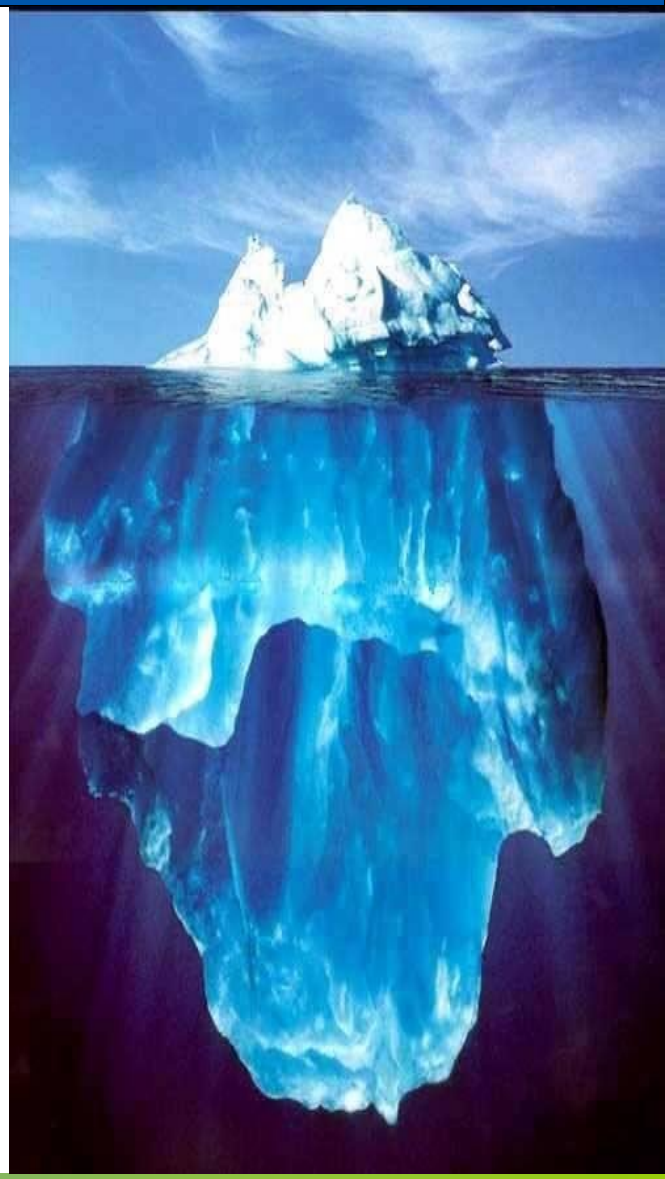
# 5.7.2 MongoDB概念解析

在mongodb中基本的概念是文档、集合、数据库

| SQL术语/概念 | MongoDB术语/概念 | 解释/说明 |
| --- | --- | --- |
| database | database | 数据库 |
| table | collection | 数据库表/集合 |
| row | document | 数据记录行/文档 |
| column | field | 数据字段/域 |
| index | index | 索引 |
| table joins | | 表连接,MongoDB不支持 |
| primary key | primary key | 主键,MongoDB自动将_id字段设置为主键 |

# 第六章 云数据库

- 6.1 云数据库概述
- 6.2 云数据库产品
- 6.3 云数据库系统架构
- 6.4 Amazon AWS和云数据库
- 6.5 微软云数据库SQL Azure
- 6.6 云数据库实践



# 6.2.1云数据库厂商概述

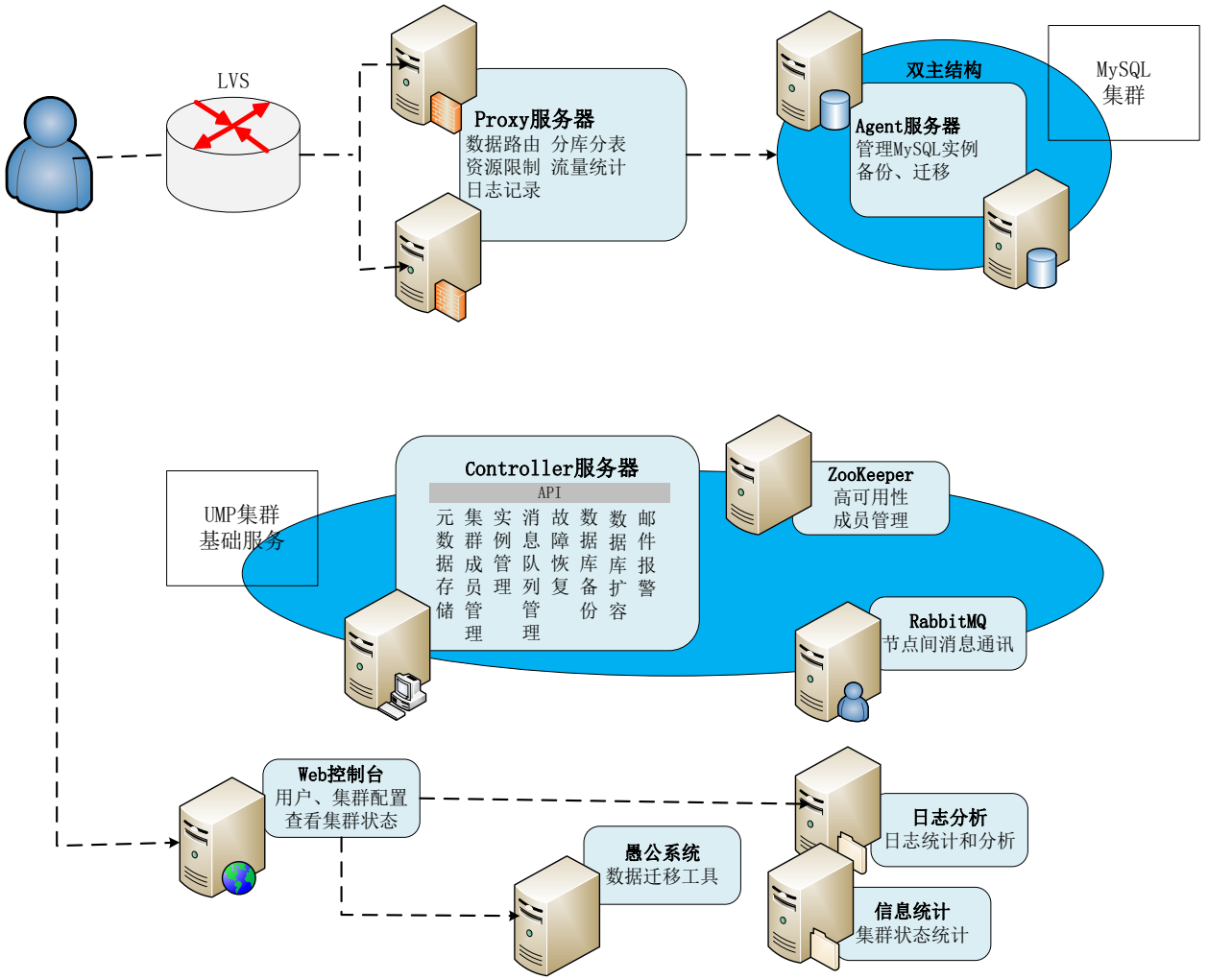
云数据库产品

| 企业 | 产品 |
| --- | --- |
| Amazon | Dynamo、SimpleDB、RDS |
| Google | Google Cloud SQL |
| Microsoft | Microsoft SQL Azure |
| Oracle | Oracle Cloud |
| Yahoo! | PNUTS |
| Vertica | Analytic Database v3.0 for the Cloud |
| EnerpriseDB | Postgres Plus in the Cloud |
| 阿里 | 阿里云RDS |
| 百度 | 百度云数据库 |
| 腾讯 | 腾讯云数据库 |

# 6.3.2UMP系统架构

UMP系统中的角色包括：
- Controller服务器
- Proxy服务器
- Agent服务器
- Web控制台
- 日志分析服务器
- 信息统计服务器
- 愚公系统

依赖的开源组件包括：
- Mnesia
- RabbitMQ
- ZooKeeper
- LVS



# 6.3.3UMP系统功能

UMP系统是构建在一个大的集群之上的，通过多个组件的协同作业，整个系统实现了对用户透明的各种功能：

- 容灾
- 读写分离
- 分库分表
- 资源管理
- 资源调度
- 资源隔离
- 数据安全

# 6.3.3UMP系统功能

**1. 容灾**

- 为了实现容灾，UMP系统会为每个用户创建两个MySQL实例，一个是主库，一个是从库
- 主库和从库的状态是由Zookeeper负责维护的
- 主从切换过程如下：
  - Zookeeper探测到主库故障，通知Controller服务器
  - Controller服务器启动主从切换时，会修改“路由表”，即用户名到后端MySQL实例地址的映射关系
  - 把主库标记为不可用
  - 借助于消息中间件RabbitMQ通知所有Proxy服务器修改用户名到后端MySQL实例地址的映射关系
  - 全部过程对用户透明

# BUMP系统功能

**1. 容灾**

- 宕机后的主库在进行恢复处理后需要再次上线，过程如下：
  - 在主库恢复时，会把从库的更新复制给自己
  - 当主库的数据库状态快要达到和从库一致的状态时，Controller服务器就会命令从库停止更新，进入不可写状态，禁止用户写入数据
  - 等到主库更新到和从库完全一致的状态时，Controller服务器就会发起主从切换操作，并在路由表中把主库标记为可用状态
  - 通知Proxy服务器把写操作切回主库上，用户写操作可以继续执行，之后再把从库修改为可写状态

# 6.3.3UMP系统功能

**2. 读写分离**

•充分利用主从库实现用户读写操作的分离，实现负载均衡

•UMP系统实现了对于用户透明的读写分离功能，当整个功能被开启时，负责向用户提供访问MySQL数据库服务的Proxy服务器，就会对用户发起的SQL语句进行解析，如果属于写操作，就直接发送到主库，如果是读操作，就会被均衡地发送到主库和从库上执行

# 6.3.3UMP系统功能

**3. 分库分表**

UMP支持对用户透明的分库分表（shard / horizontal partition）

当采用分库分表时，系统处理用户查询的过程如下：

- 首先，Proxy服务器解析用户SQL语句，提取出重写和分发SQL语句所需要的信息
- 其次，对SQL语句进行重写，得到多个针对相应MySQL实例的子语句，然后把子语句分发到对应的MySQL实例上执行
- 最后，接收来自各个MySQL实例的SQL语句执行结果，合并得到最终结果

# 6.3.3UMP系统功能

**4. 资源管理**

•UMP系统采用资源池机制来管理数据库服务器上的CPU、内存、磁盘等计算资源，所有的计算资源都放在资源池内进行统一分配，资源池是为MySQL实例分配资源的基本单位

•整个集群中的所有服务器会根据其机型、所在机房等因素被划分多个资源池，每台服务器会被加入到相应的资源池中

•对于每个具体MySQL实例，管理员会根据应用部署在哪些机房、需要哪些计算资源等因素，为该MySQL实例具体指定主库和从库所在的资源池，然后，系统的实例管理服务会本着负载均衡的原则，从资源池中选择负载较轻的服务器来创建MySQL实例

# 6.3.3UMP系统功能

**5. 资源调度**

•UMP系统中有三种规格的用户，分别是数据量和流量比较小的用户、中等规模用户以及需要分库分表的用户
•多个小规模用户可以共享同一个MySQL实例
•对于中等规模的用户，每个用户独占一个MySQL实例
•对于分库分表的用户，会占有多个独立的MySQL实例

# 6.3.3UMP系统功能

**6.资源隔离**

UMP采用的两种资源隔离方式

| 方法 | 应用场合 | 实现方式 |
| --- | --- | --- |
| 用Cgroup限制MySQL进程资源 | 适用于多个MySQL实例共享同一台物理机的情况 | 可以对用户的MySQL进程最大可以使用的CPU使用率、内存和IOPS等进行限制 |
| 在Proxy服务器端限制QPS | 适用于多个用户共享同一个MySQL实例的情况 | Controller服务器监测用户的MySQL实例的资源消耗情况，如果明显超出配额，就通知Proxy服务器通过增加延迟的方法去限制用户的QPS，以减少用户对系统资源的消耗 |

# 6.3.3UMP系统功能

**7. 数据安全**

UMP系统设计了多种机制来保证数据安全：

•SSL数据库连接：SSL(Secure Sockets Layer)是为网络通信提供安全及数据完整性的一种安全协议，它在传输层对网络连接进行加密。Proxy服务器实现了完整的MySQL客户端/服务器协议，可以与客户端之间建立SSL数据库连接

•数据访问IP白名单：可以把允许访问云数据库的IP地址放入“白名单”，只有白名单内的IP地址才能访问，其他IP地址的访问都会被拒绝，从而进一步保证账户安全

•记录用户操作日志：用户的所有操作记录都会被记录到日志分析服务器，通过检查用户操作记录，可以发现隐藏的安全漏洞

•SQL拦截：Proxy服务器可以根据要求拦截多种类型的SQL语句，比如全表扫描语句“select *”

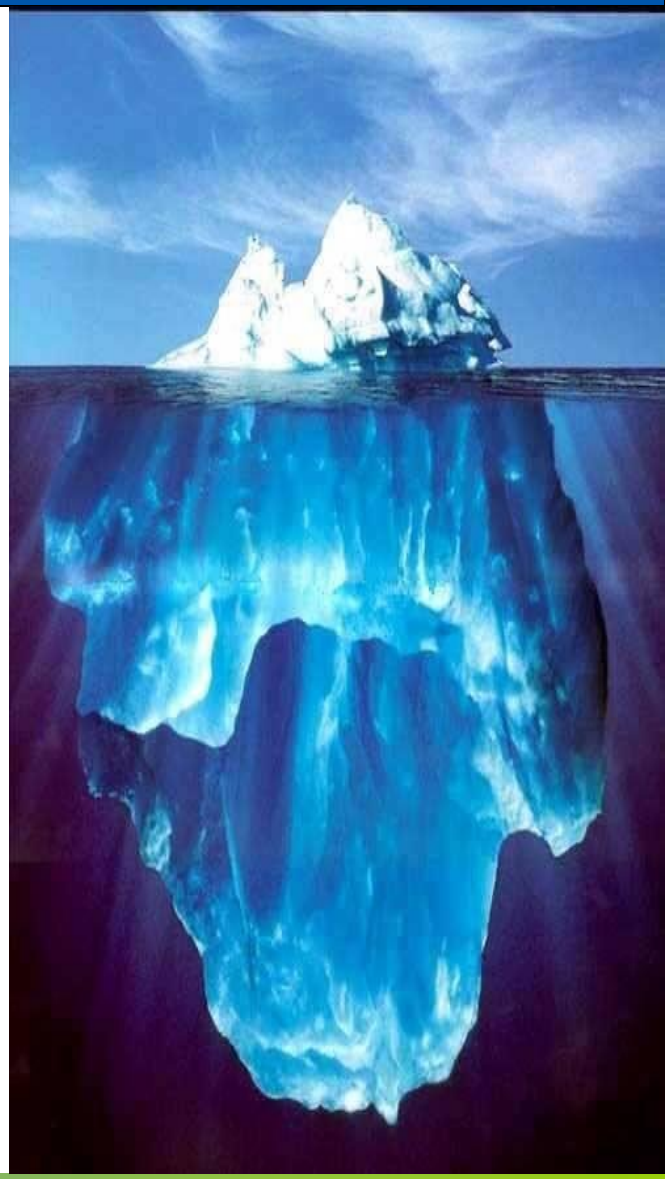
# 6.4.2 Amazon AWS

总体而言，Amazon AWS的产品分为几个部分：

- **计算类**
  - 弹性计算云EC2：EC2提供了云中的虚拟机
  - 弹性MapReduce：将Hadoop MapReduce搬到云环境中，大量EC2实例动态地成为执行大规模MapReduce计算任务的工作机
- **存储类**
  - 弹性块存储EBS
  - 简单消息存储SQS
  - Blob对象存储S3
  - NoSQL型数据库：SimpleDB和DynamoDB
  - 关系数据库RDS
- **工具支持**
  - AWS支持多种开发语言，提供Java、Rupy、Python、PHP、Windows &.NET 以及Android和iOS的工具集
  - 工具集中包含各种语言的SDK，程序自动部署以及各种管理工具
  - AWS通过CloudWatch系统提供丰富的监控功能

# 第7章 MapReduce

- 7.1 概述
- 7.2 MapReduce体系结构
- 7.3 MapReduce工作流程
- 7.4 实例分析：WordCount
- 7.5 MapReduce的具体应用
- 7.6 MapReduce编程实践



# 7.1.2MapReduce模型简介

•MapReduce将复杂的、运行于大规模集群上的并行计算过程高度地抽象到了两个函数：Map和Reduce
•编程容易，不需要掌握分布式并行编程细节，也可以很容易把自己的程序运行在分布式系统上，完成海量数据的计算
•MapReduce采用“**分而治之**”策略，一个存储在分布式文件系统中的大规模数据集，会被切分成许多独立的分片（split），这些分片可以被多个Map任务并行处理
•MapReduce设计的一个理念就是“**计算向数据靠拢**”，而不是“数据向计算靠拢”，因为，移动数据需要大量的网络传输开销
•MapReduce框架采用了Master/Slave架构，包括一个Master和若干个Slave。Master上运行JobTracker，Slave上运行TaskTracker
•Hadoop框架是用Java实现的，但是，MapReduce应用程序则不一定要用Java来写

# 7.1.3Map和Reduce函数

表7-1 Map和Reduce

| 函数 | 输入 | 输出 | 说明 |
|---|---|---|---|
| Map | <$k_1$,$v_1$><br>如：<br><行号,"a b c"> | List(<$k_2$,$v_2$>)<br>如：<br><"a",1><br><"b",1><br><"c",1> | 1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理<br>2.每一个输入的<$k_1$,$v_1$>会输出一批<$k_2$,$v_2$>。<$k_2$,$v_2$>是计算的中间结果 |
| Reduce | <$k_2$,List($v_2$)><br>如：<br><"a",<1,1,1>> | <$k_3$,$v_3$><br><"a",3> | 输入的中间结果<$k_2$,List($v_2$)>中的List($v_2$)表示是一批属于同一个$k_2$的value |

# 7.3 MapReduce工作流程

- 7.3.1 工作流程概述
- 7.3.2 MapReduce各个执行阶段
- 7.3.3 Shuffle过程详解

# 7.3.1工作流程概述

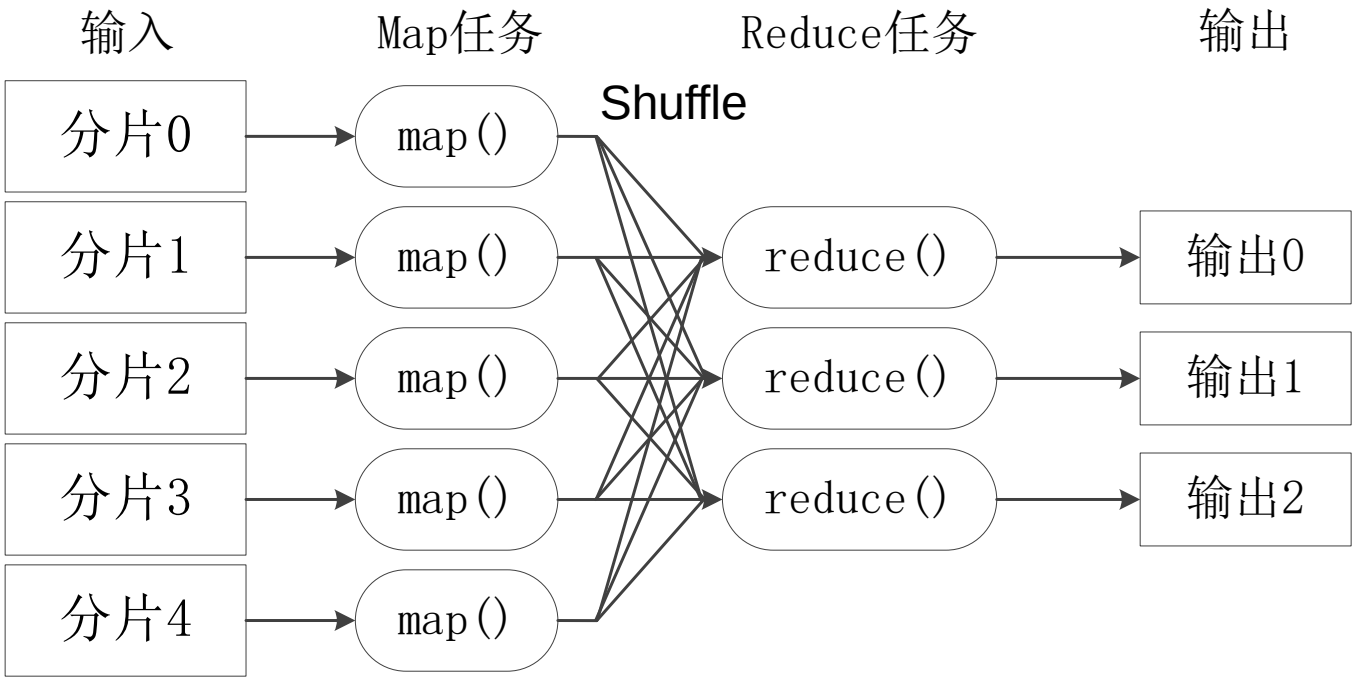
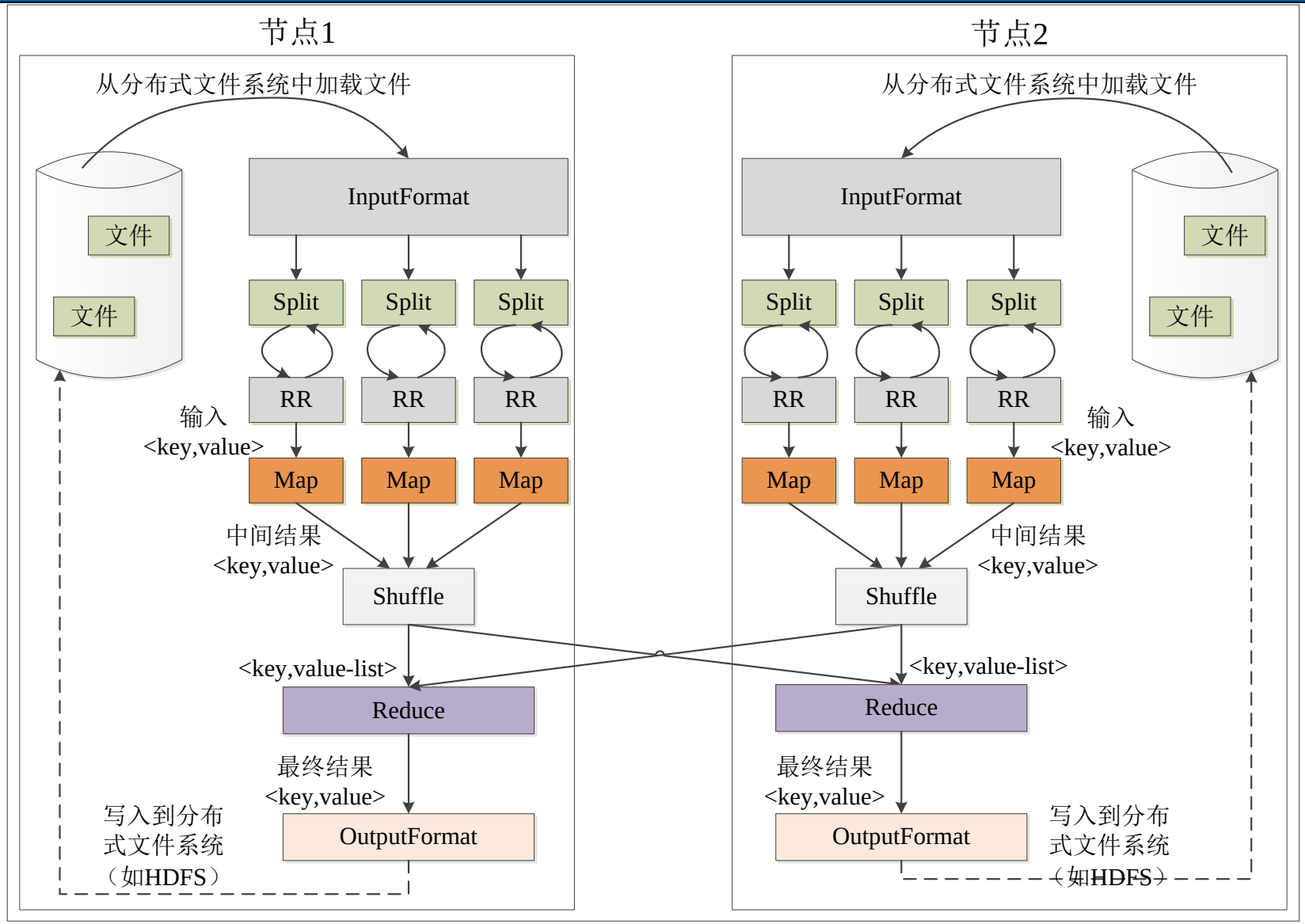



图7-1 MapReduce工作流程

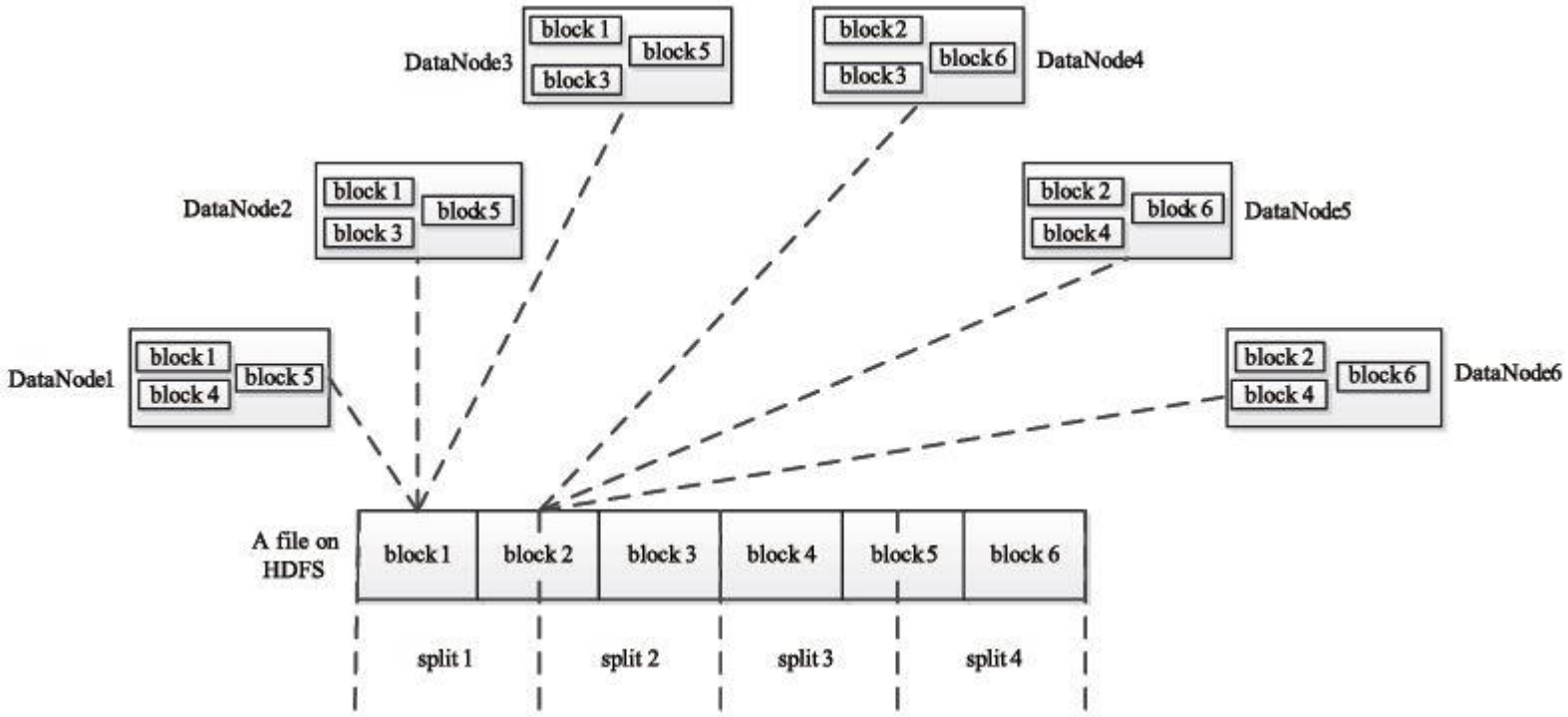
- 不同的Map任务之间不会进行通信
- 不同的Reduce任务之间也不会发生任何信息交换
- 用户不能显式地从一台机器向另一台机器发送消息
- 所有的数据交换都是通过MapReduce框架自身去实现的

# 7.3.2MapReduce各个执行阶段



# 7.3.2MapReduce各个执行阶段

关于**Split**（分片）



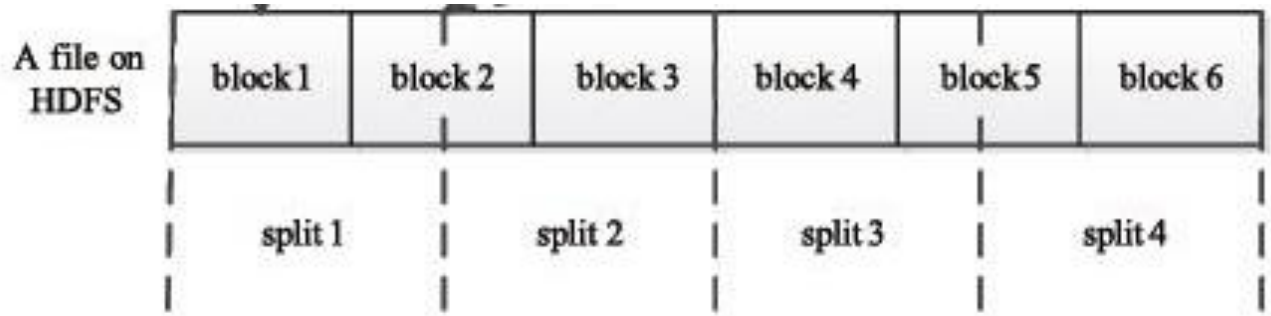

HDFS 以固定大小的block 为基本单位存储数据，而对于MapReduce 而言，其处理单位是split。

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。

# 7.3.2MapReduce各个执行阶段

**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）

# 7.3.3Shuffle过程详解

## 1. Shuffle过程简介

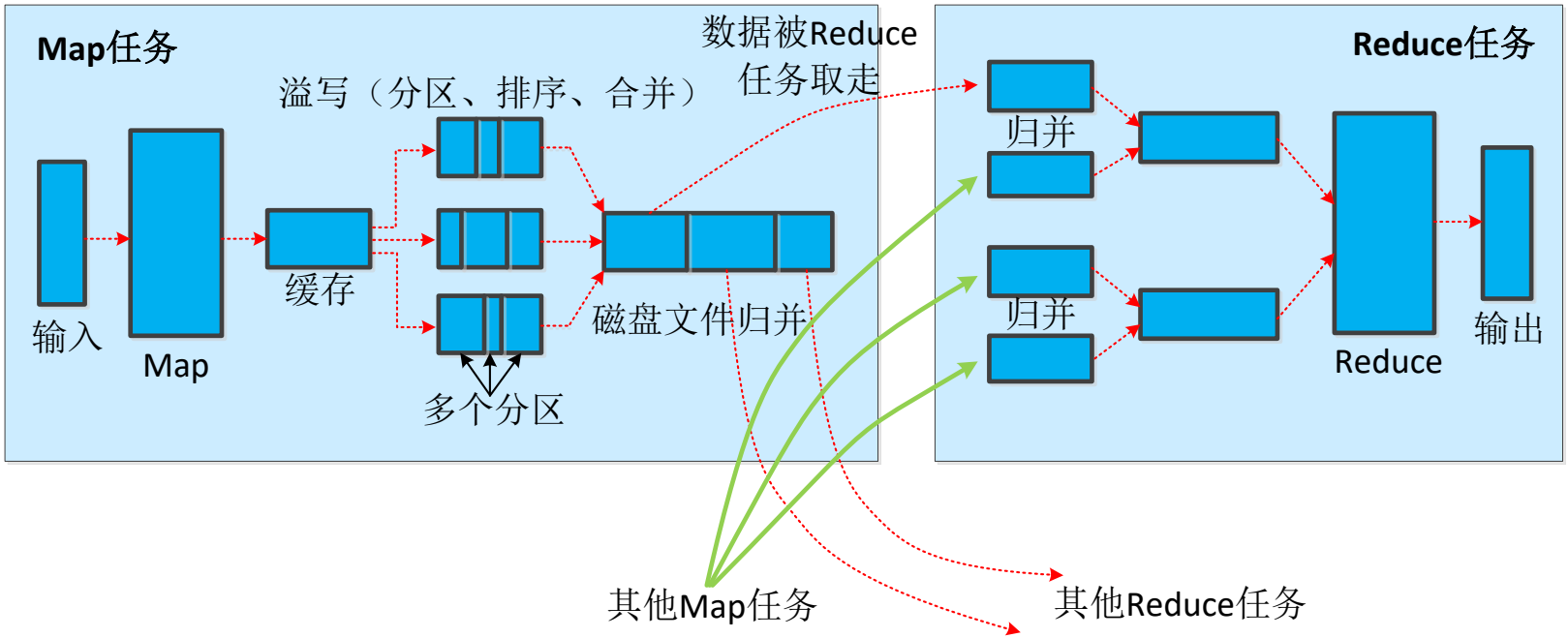
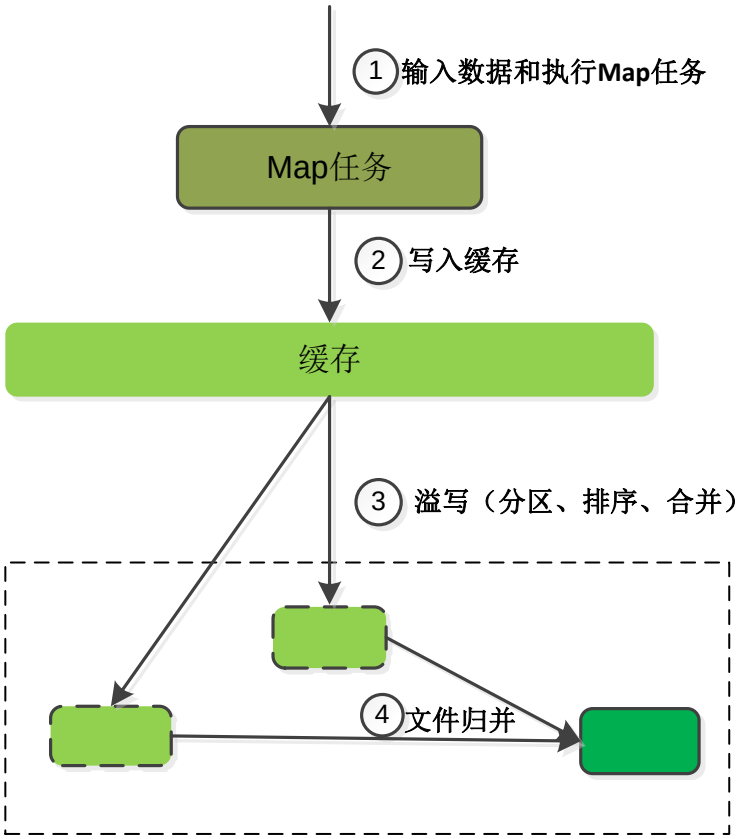



图7-3 Shuffle过程

# 7.3.3Shuffle过程详解

## 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>>

# 7.3.3Shuffle过程详解

### 3. Reduce端的Shuffle过程

- Reduce任务通过RPC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

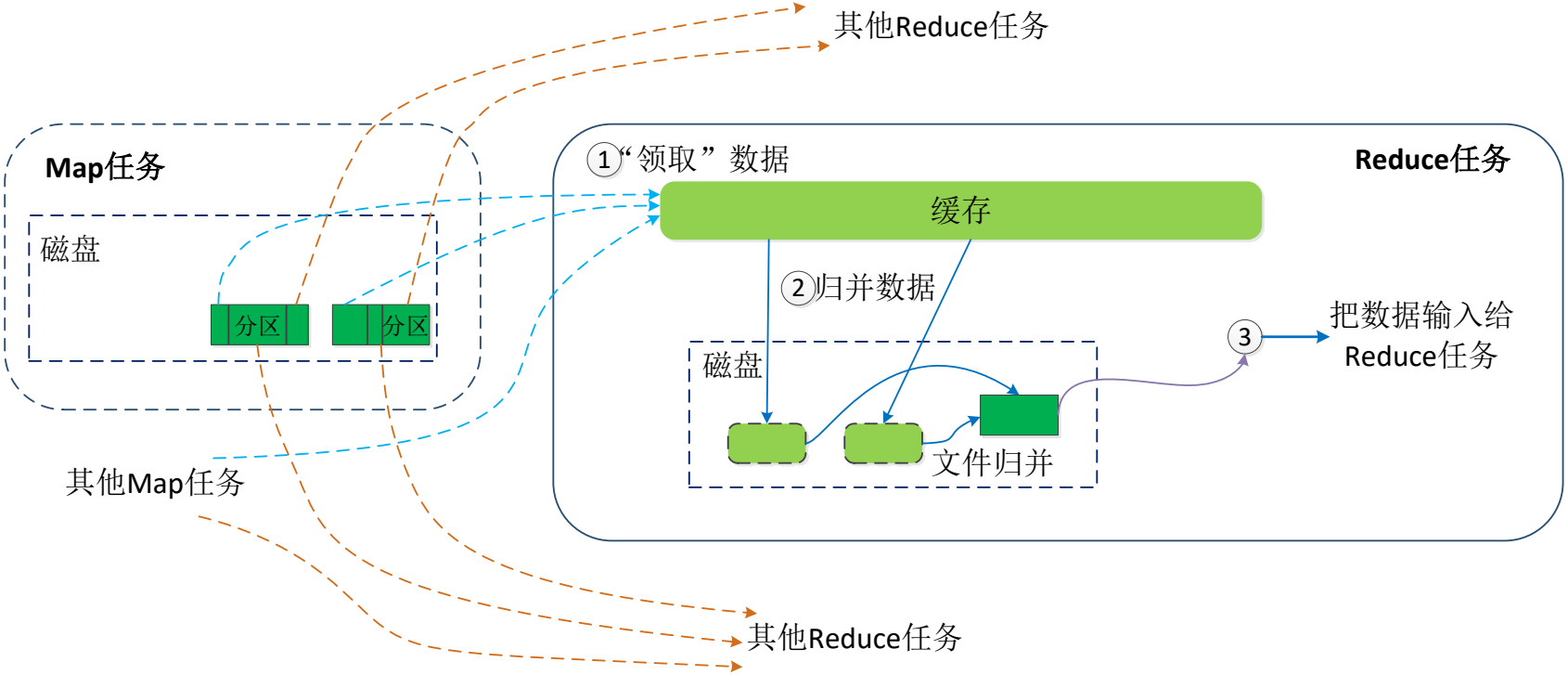



图7-5 Reduce端的Shuffle过程

# 7.4.1WordCount程序任务

表7-2 WordCount程序任务

| 程序 | WordCount |
| --- | --- |
| 输入 | 一个包含大量单词的文本文件 |
| 输出 | 文件中每个单词及其出现次数（频数），并按照单词字母顺序排序，每个单词和其频数占一行，单词和频数之间有间隔 |

表7-3 一个WordCount的输入和输出实例

| 输入 | 输出 |
| --- | --- |
| Hello World<br>Hello Hadoop<br>Hello MapReduce | Hadoop 1<br>Hello 3<br>MapReduce 1<br>World 1 |

# 7.4.2WordCount设计思路

- 首先，需要检查WordCount程序任务是否可以采用MapReduce来实现
- 其次，确定MapReduce程序的设计思路
- 最后，确定MapReduce程序的执行过程

# 7.1.3Map和Reduce函数

表7-1 Map和Reduce

| 函数 | 输入 | 输出 | 说明 |
|---|---|---|---|
| Map | <$k_1$,$v_1$><br>如：<br><行号,"a b c"> | List(<$k_2$,$v_2$>)<br>如：<br><"a",1><br><"b",1><br><"c",1> | 1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理<br>2.每一个输入的<$k_1$,$v_1$>会输出一批<$k_2$,$v_2$>。<$k_2$,$v_2$>是计算的中间结果 |
| Reduce | <$k_2$,List($v_2$)><br>如：<br><"a",<1,1,1>> | <$k_3$,$v_3$><br><"a",3> | 输入的中间结果<$k_2$,List($v_2$)>中的List($v_2$)表示是一批属于同一个$k_2$的value |

# 7.4.3一个WordCount执行过程的实例


图7-7 Map过程示意图

# 7.4.3一个WordCount执行过程的实例

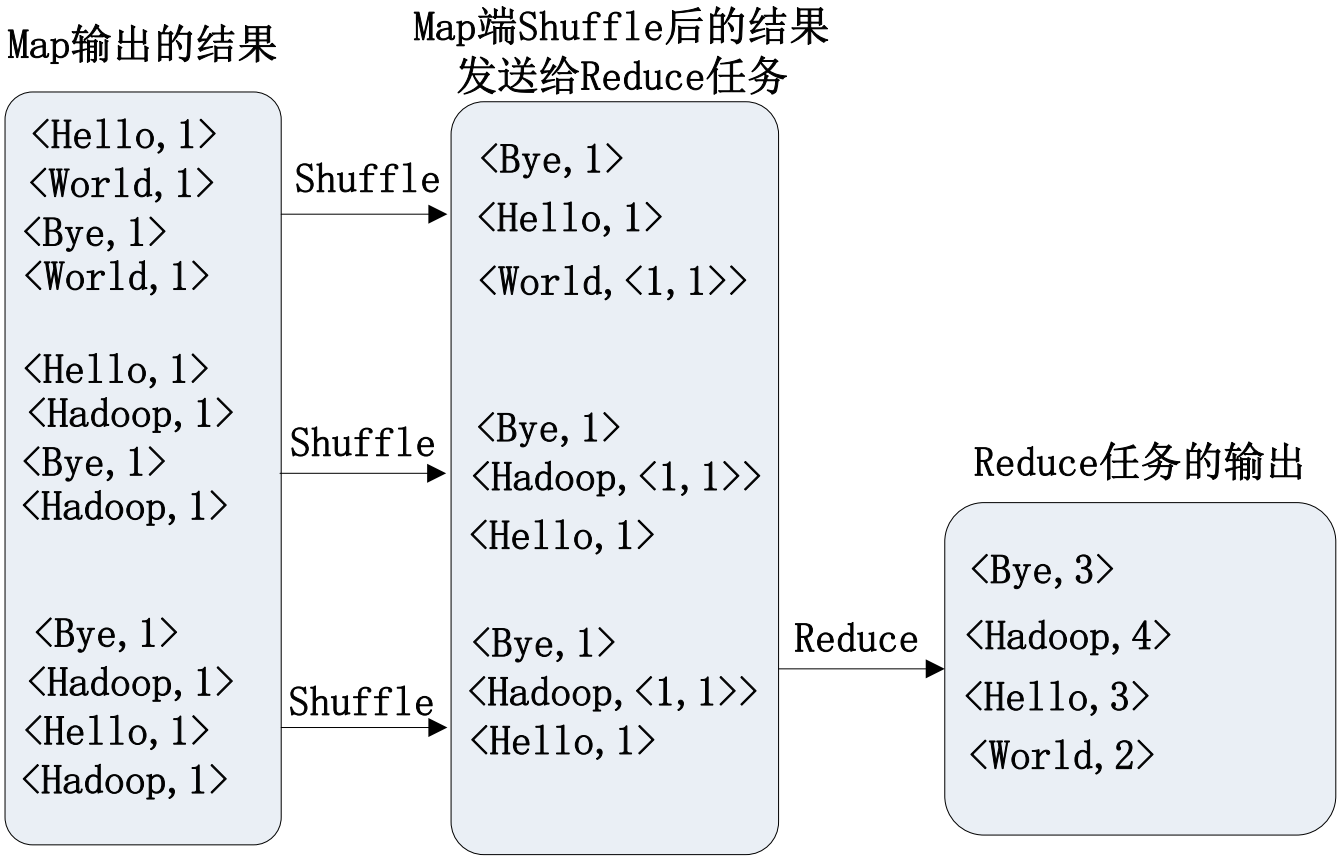



图7-8 用户没有定义Combiner时的Reduce过程示意图

# 7.4.3一个WordCount执行过程的实例

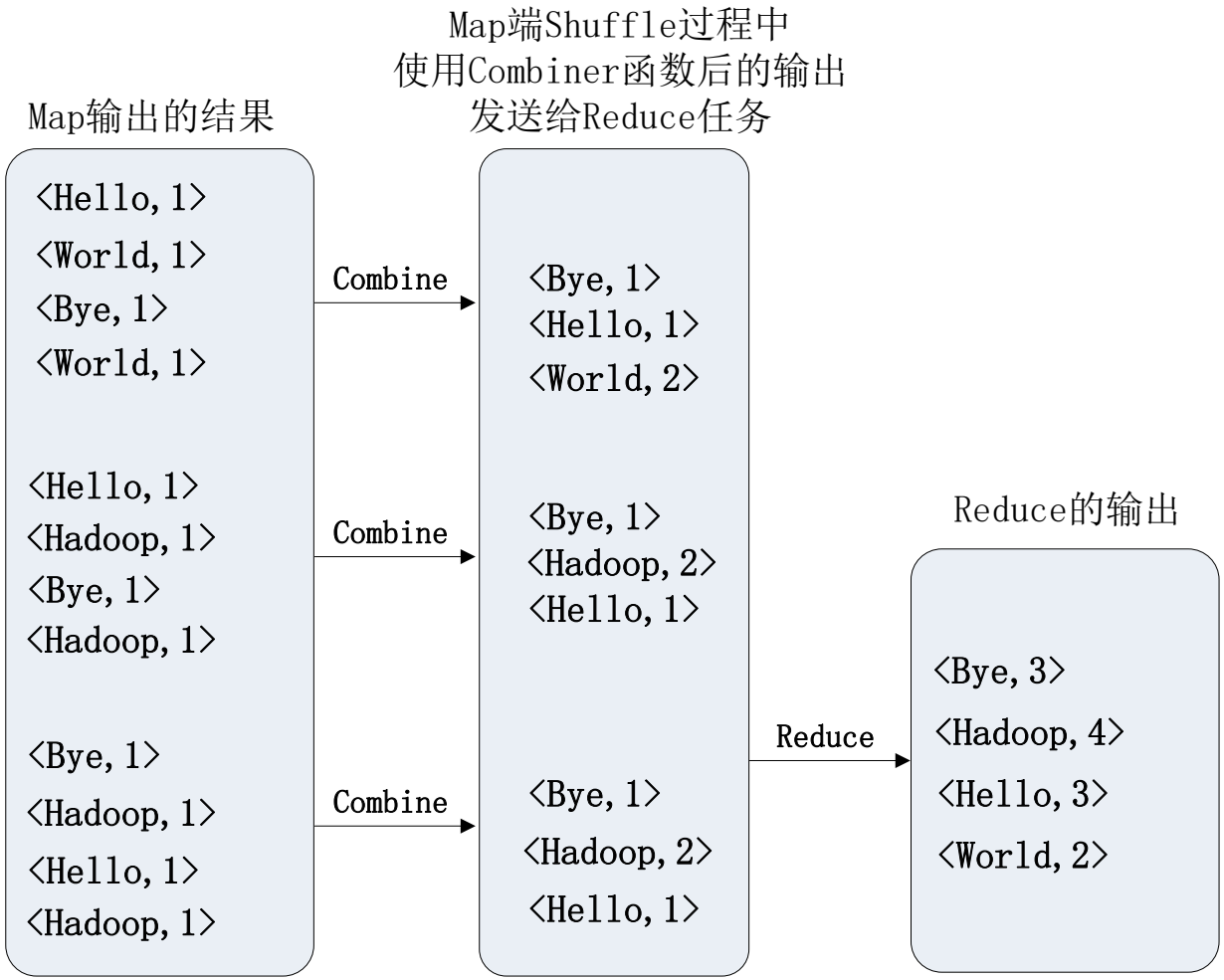
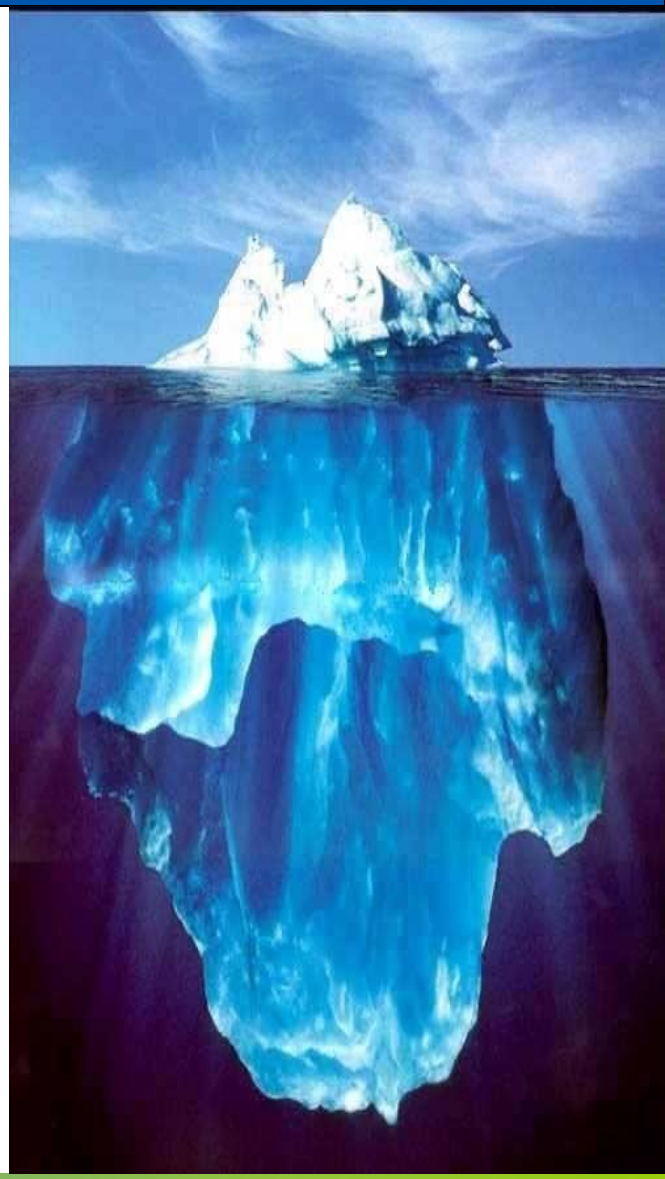



图7-9 用户有定义Combiner时的Reduce过程示意图

# 第8章 Hadoop架构再探讨

- 8.1 Hadoop的优化与发展
- 8.2 HDFS2.0的新特性
- 8.3 新一代资源管理调度框架YARN
- 8.4 Hadoop生态系统中具有代表性的功能组件



# 8.1.2针对Hadoop的改进与提升

表 Hadoop框架自身的改进：从1.0到2.0

| 组件 | **Hadoop1.0的问题** | **Hadoop2.0的改进** |
| --- | --- | --- |
| HDFS | 单一名称节点，存在单点失效问题 | 设计了HDFS HA，提供名称节点热备机制 |
| HDFS | 单一命名空间，无法实现资源隔离 | 设计了HDFS Federation，管理多个命名空间 |
| MapReduce | 资源管理效率低 | 设计了新的资源管理框架YARN |

# 8.2.1HDFS HA

- HDFS HA（High Availability）是为了解决单点故障问题
- HA集群设置两个名称节点，“活跃（Active）”和“待命（Standby）”
- 两种名称节点的状态同步，可以借助于一个共享存储系统来实现
- 一旦活跃名称节点出现故障，就可以立即切换到待命名称节点
- Zookeeper确保一个名称节点在对外服务
- 名称节点维护映射信息，数据节点同时向两个名称节点汇报信息

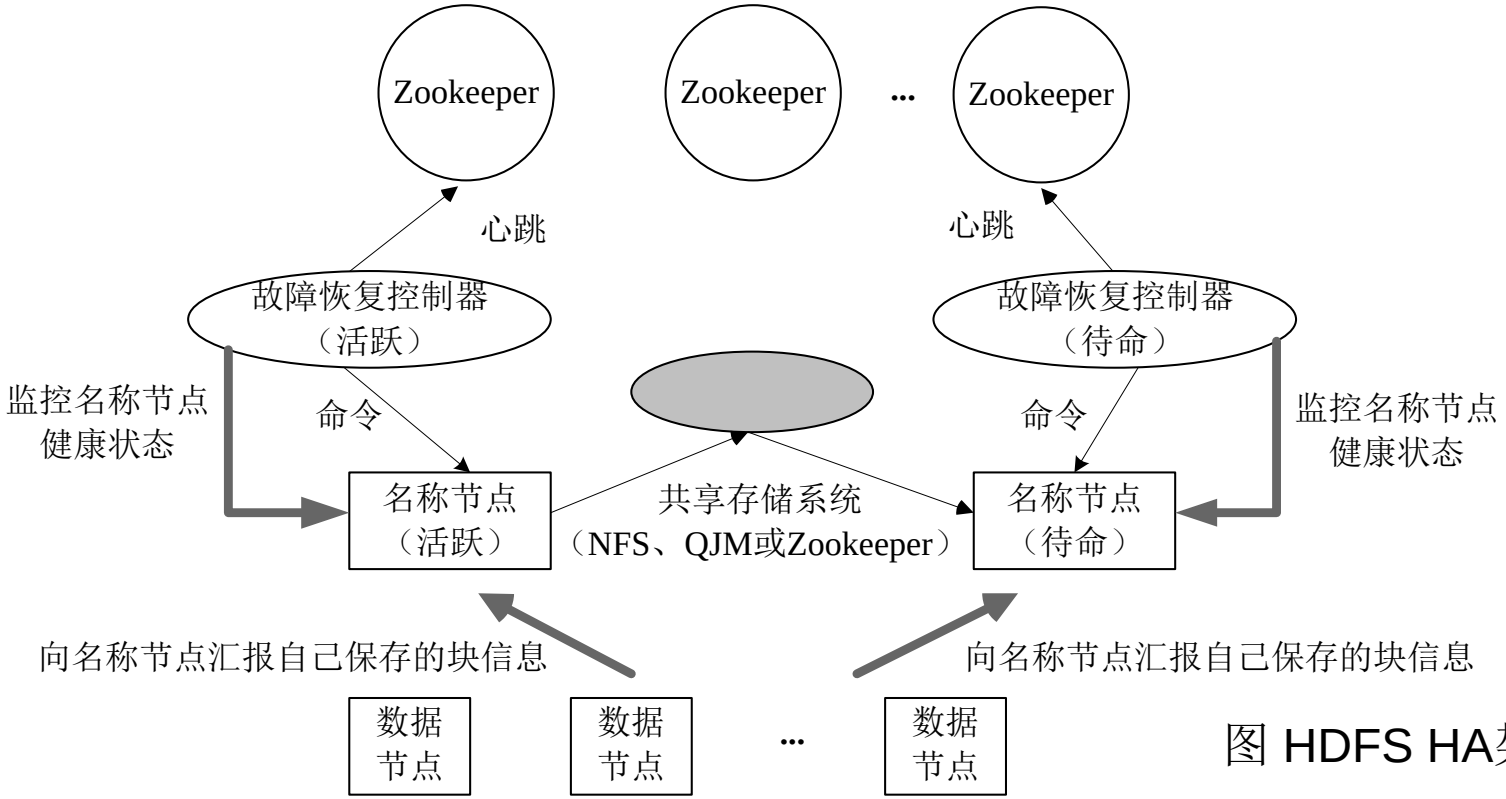



图 HDFS HA架构

# 8.2.2HDFS Federation

## 1.HDFS1.0中存在的问题

- 单点故障问题
- 不可以水平扩展（是否可以通过纵向扩展来解决？）
- 系统整体性能受限于单个名称节点的吞吐量
- 单个名称节点难以提供不同程序之间的隔离性
- HDFS HA是热备份，提供高可用性，但是无法解决可扩展性、系统性能和隔离性

## 2.HDFS Federation的设计

- 在HDFS Federation中，设计了多个相互独立的名称节点，使得HDFS的命名服务能够水平扩展，这些名称节点分别进行各自命名空间和块的管理，相互之间是联盟（Federation）关系，不需要彼此协调。并且向后兼容
- HDFS Federation中，所有名称节点会共享底层的数据节点存储资源，数据节点向所有名称节点汇报
- 属于同一个命名空间的块构成一个“块池”

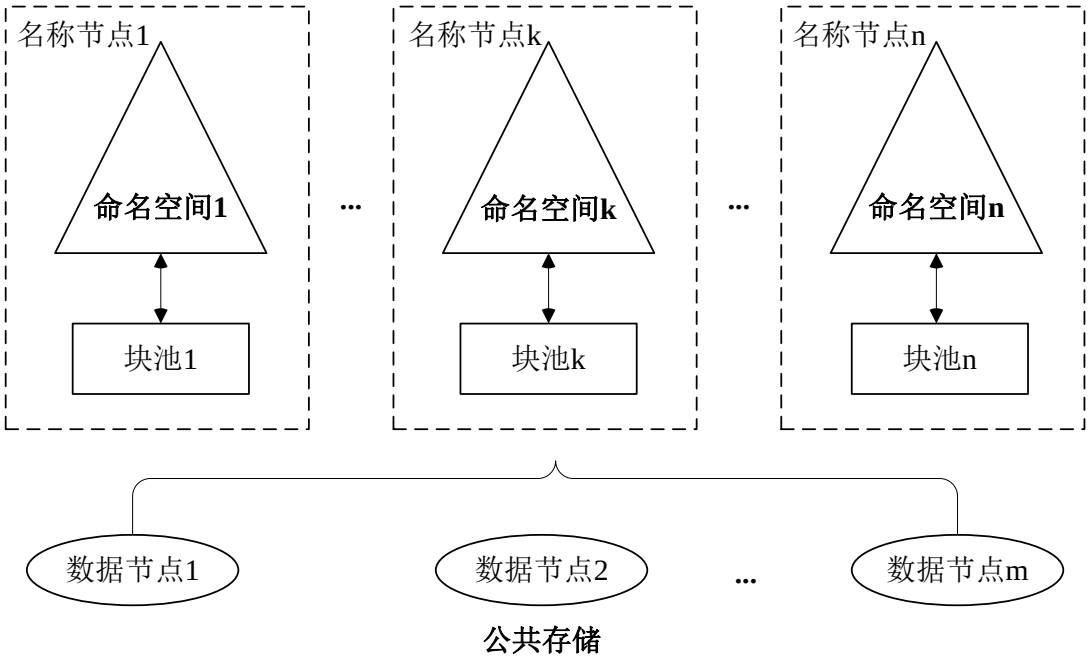



图 HDFS Federation架构

# 8.3.6 YARN的发展目标

- YARN的目标就是实现“一个集群多个框架”，即在一个集群上部署一个统一的资源调度管理框架YARN，在YARN之上可以部署其他各种计算框架
- 由YARN为这些计算框架提供统一的资源调度管理服务，并且能够根据各种计算框架的负载需求，调整各自占用的资源，实现集群资源共享和资源弹性收缩
- 可以实现一个集群上的不同应用负载混搭，有效提高了集群的利用率
- 不同计算框架可以共享底层存储，避免了数据集跨集群移动

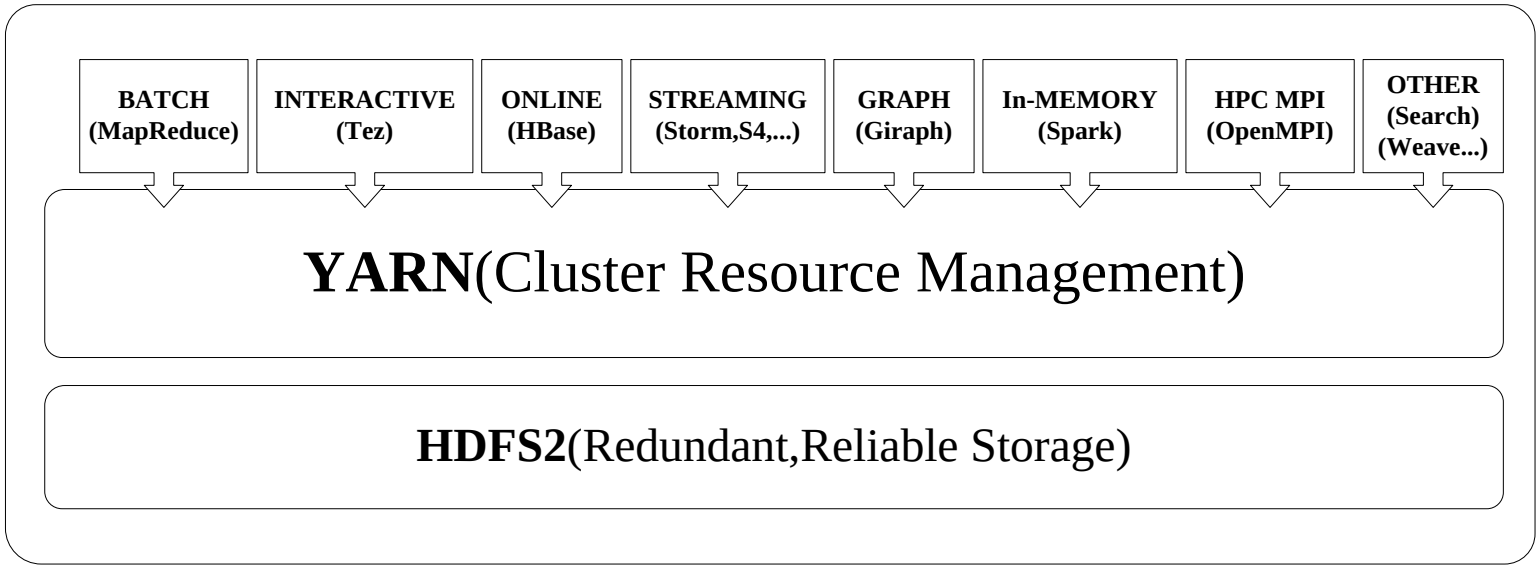



图 在YARN上部署各种计算框架

# 8.1.2针对Hadoop的改进与提升

表 不断完善的Hadoop生态系统

| 组件 | 功能 | 解决Hadoop中存在的问题 |
| --- | --- | --- |
| Pig | 处理大规模数据的脚本语言，用户只需要编写几条简单的语句，系统会自动转换为MapReduce作业 | 抽象层次低，需要手工编写大量代码 |
| Spark | 基于内存的分布式并行编程框架，具有较高的实时性，并且较好支持迭代计算 | 延迟高，而且不适合执行迭代计算 |
| Oozie | 工作流和协作服务引擎，协调Hadoop上运行的不同任务 | 没有提供作业（Job）之间依赖关系管理机制，需要用户自己处理作业之间依赖关系 |
| Tez | 支持DAG作业的计算框架，对作业的操作进行重新分解和组合，形成一个大的DAG作业，减少不必要操作 | 不同的MapReduce任务之间存在重复操作，降低了效率 |
| Kafka | 分布式发布订阅消息系统，一般作为企业大数据分析平台的数据交换枢纽，不同类型的分布式系统可以统一接入到Kafka，实现和Hadoop各个组件之间的不同类型数据的实时高效交换 | Hadoop生态系统中各个组件和其他产品之间缺乏统一的、高效的数据交换中介 |