第5章 – 大数据多机计算：Hadoop

Chapter 5 – Big Data Computing across Computers: Hadoop

大数据和分布式：基础概念

大数据 BIG DATA

- 伴随着信息技术的快速发展，接入互联网的用户和终端的体量愈发庞大，其能够产生的数据量已经是天文数字：
- 根据IDC预测，到2025年全球将能够生产175ZB的数据

- ZB：Zettabyte；1ZB=10^3EB；1EB=10^3PB；1PB=10^3TB；
- 1ZB=10^{21}Byte

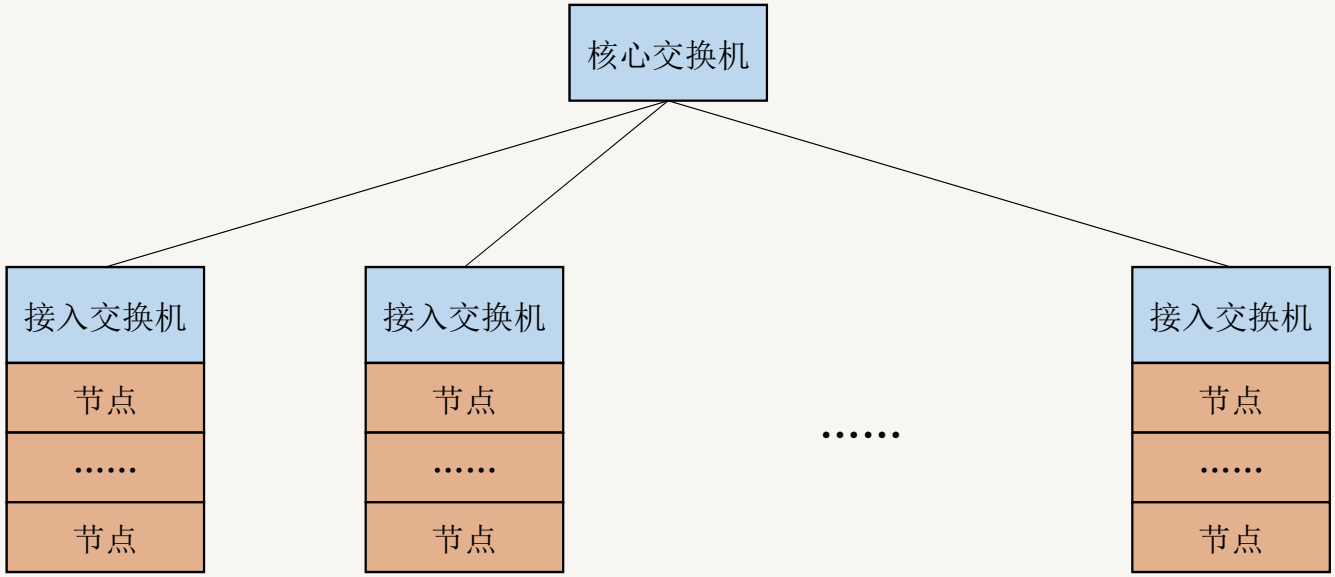
- 如此海量而复杂的数据固然是一片宝藏，但其对于计算机的数据传输、存储、计算等各方面都提出了挑战，传统的机器和软件已经远不足以对其进行处理。因此面向大数据的数据中心，以及其上运行的各种面向大数据的软件，已经成为计算机领域的热门话题

从以下角度来思考大数据：

- 硬件
- 软件

从**硬件角度**看，一台或是几台机器似乎难以胜任大数据的存储和计算工作，因此我们需要：

- 大量机器的集群构成数据中心
- 使用高速互联网络对大量机器进行连接以确保数据传递
- 综合考量数据中心的散热问题、能耗问题，以及各方面成本
- 集群中硬件发生故障的概率很高，如何确保可靠性
- 单一架构的机器难以胜任各种计算类型，考虑异构计算
- ……



一种常见的数据中心机房网络拓扑，一个机架内的节点连接到接入交换机，接入交换机再连接到核心交换机

可靠性问题：

- 由于数据中心的集群中往往包含数以万计的计算机，为顾及成本，集群往往使用较为廉价的普通商用硬件

- 试算：假设集群有50000台机器，每台机器有4块硬盘，每块硬盘的年度故障率为1%，则平均每天能遇到几次硬盘故障？
- 50000*4*1%/365≈5次

从**软件角度**看，由于大数据的计算与存储分布在未必可靠的大量计算机组成的集群上，因此我们需要：

- 分而治之，使用分片存储策略和分布式算法对大数据进行存储与处理
- 考虑存储与计算的容错性，以使得故障发生时造成的损失最小化　有硬件级别的容错，也有软件级别的容错，软件级别考虑的多
- 算法设计方面要尽可能减少节点间通信（因为这很耗时）
- ……

分布式 DISTRIBUTED

- 谈到大数据就不得不谈到分布式，在单机上存储和处理大数据是不可能的。分布式，就是将任务分配到许多节点（机器）上去，这是一种借助网络而产生的并行方法。在大数据方向，可以将它的研究分为两方面：
  - 分布式存储（如分布式文件系统、分布式数据库）
  - 分布式计算（本质上是并行计算模型与算法，如上一章的MPI可以用于分布式计算）

谷歌“三驾马车”

在分布式领域，谷歌公司提出了三项技术来分别解决文件存储、结构化数据存储和分布式计算模型这三个关键问题，并以此开启了大数据时代。这些技术有些已经停止使用并有了更新的替代品，但并不妨碍我们了解他们的思想。他们分别是：

- GFS（分布式文件系统，SOSP2003）
- BigTable（分布式存储系统，OSDI06）
- MapReduce（分布式运算编程框架，OSDI04）

GFS (SOSP2003)

- Google File System（GFS）是一个可扩展的**分布式文件系统**，适用于大型分布式数据密集型应用程序。它能够在廉价的商用硬件上提供容错能力，并能为大量客户端提供较高的总体性能。

  - ✓ 把一个较大的文件切分成不同的单元块；
  - ✓ 把每个单元块存储在一个ChunkServer服务器节点上，并且每一块都会复制在多个ChunkServer服务器；
  - ✓ 每一个文件包含多少块和哪些块，这些元数据存储在GFS Master服务器上；
  - ✓ 构成一个低成本的分布式存储系统，被用来处理数据量非常大的存储场景，为MapReduce的大数据处理模型提供输入和输出的存储系统；

BigTable (OSDI2006)

- Bigtable 是一个分布式存储系统，用于管理**结构化数据**，旨在扩展到非常大的规模：数千个商用服务器上的 PB 级数据。 Google 的许多项目（60）都将数据存储在 Bigtable 中，包括网络索引、Google 地球和 Google 财经。

MapReduce (OSDI2004)

- MapReduce 是一种用于处理和生成大型数据集的分布式运算程序**编程模型**和相关实现。用户指定 map 函数来处理键-值对（KV）以生成一组中间键-值对，以及 reduce 函数来合并与同一中间键关联的所有中间值（相当于分组合并）最后得到结果。

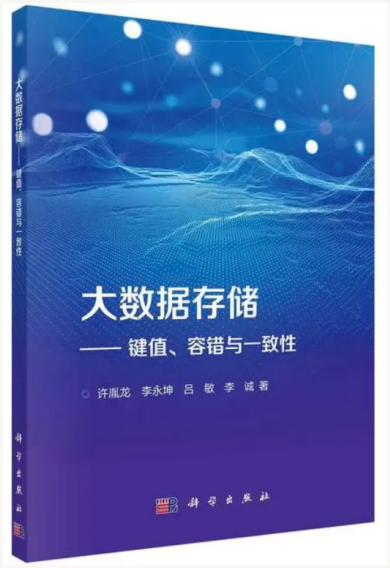
键值存储

优点：

- 简单：数据结构中只有键和值，并成对出现，值在理论上可以存放任一数据，并支持大数据存储。
- 快速：以内存运行模式为主，数据处理快是其最大优势。
- 高效计算：数据结构简单化，数据集之间的关系简单化，再加上基于内存的数据集计算，分布式计算等，形成了高效计算的前提条件。
- 分布式处理：分布式处理能力使键值数据库具备了处理大数据的能力。

缺点：

- 对值进行多值查找功能很弱。
- 缺少约束容易出错。
- 不容易建立复杂关系。
- 根据其缺点可以看出来，很多查询，排序，统计等功能需要程序员在业务代码进行编程约束。



Hadoop

Hadoop与云计算区别

- 范围与定位
- 资源利用率
- 可靠性

Hadoop与云计算联系

- 互补性
- 数据共享与交互

Hadoop 是什么?

- Hadoop 是一系列开源软件的集合，是为大数据的处理而设计的分布式框架。
- Hadoop 将单机扩展到数千台机器，每台机器都提供本地计算和存储。
- Hadoop 通过对应用层故障的检测和处理，再由不可靠硬件构成的集群上实现高可用。

Hadoop包含了谷歌“三驾马车”的开源实现：

- GFS -- HDFS
- MapReduce -- Hadoop MapReduce
- BigTable -- HBase（不展开）

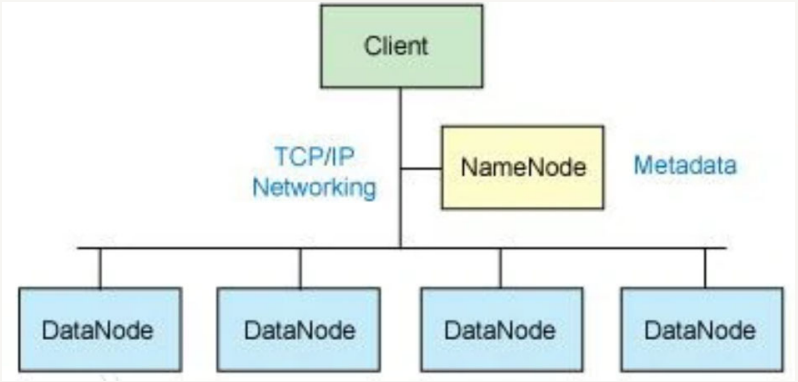
以及其他的相关组件：

- Hadoop Common：支持其他 Hadoop 模块的通用实用程序
- Hadoop YARN：作业调度和集群资源管理的框架
- ……

HDFS具有一些假设：

- 硬件故障经常发生，因此需要能检测故障并快速恢复
- 面向流式数据访问，为批处理而非用户交互使用，更注重高吞吐而非低延迟，因此并未兼容POSIX
- 针对大型数据集，典型文件大小为GB到TB级，不适合小文件读取，并应当在数百个节点上支持数千万的文件
- 简化的一致性模型，一个文件一旦创建、写入和关闭就不需要更改，除了追加和截断，这样简化了一致性问题且提高了吞吐
- 移动计算而非移动数据，尤其当数据集很大时，这将会较少网络拥塞并提升吞吐
- 跨软硬件平台的可移植性

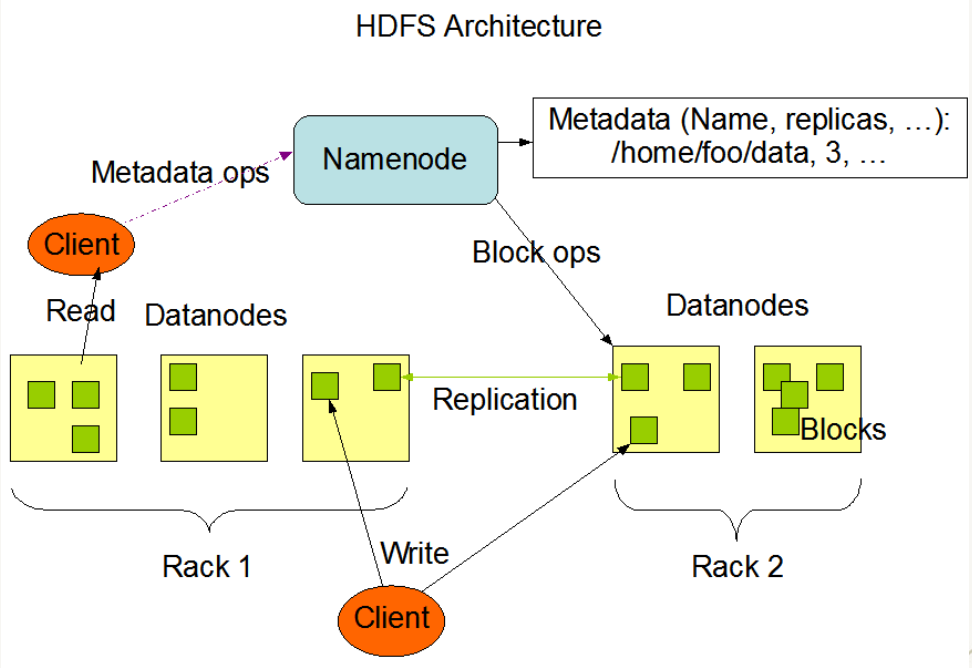
HDFS：分布式文件系统



- HDFS是一种依照GFS设计的分布式文件系统
- 运行在低成本商业硬件上，提供高容错性
- 提供高吞吐量访问，支持具有大量数据集的应用程序
- 运行在用户态，并非内核级文件系统

NAMENODE 和 DATANODE

- HDFS使用master/slave架构
- 集群包含一个NameNode和多个DataNode

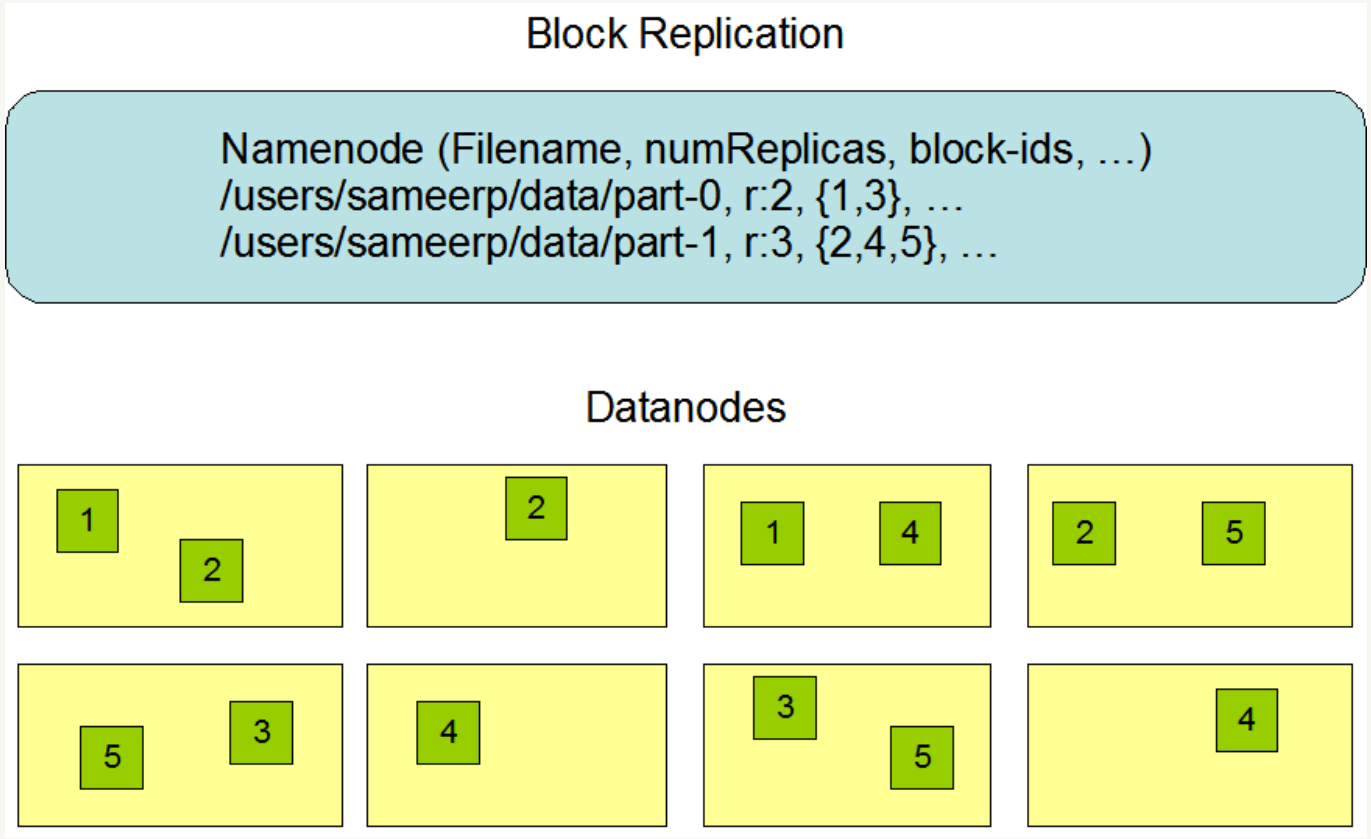


- DataNode：在HDFS中，文件被分为一到多个块（Block）。DataNode用于实际存储这种“块”，并处理块的创建、删除。同时，还负责处理来自文件系统客户端的读取和写入请求。

- NameNode：每个集群一个（也可以有备份），用于维护文件系统的元数据（命名空间），执行文件系统命名空间上的操作，如打开、关闭、重命名文件和目录，以及确定块（Block）和DataNode的映射。

数据的复制

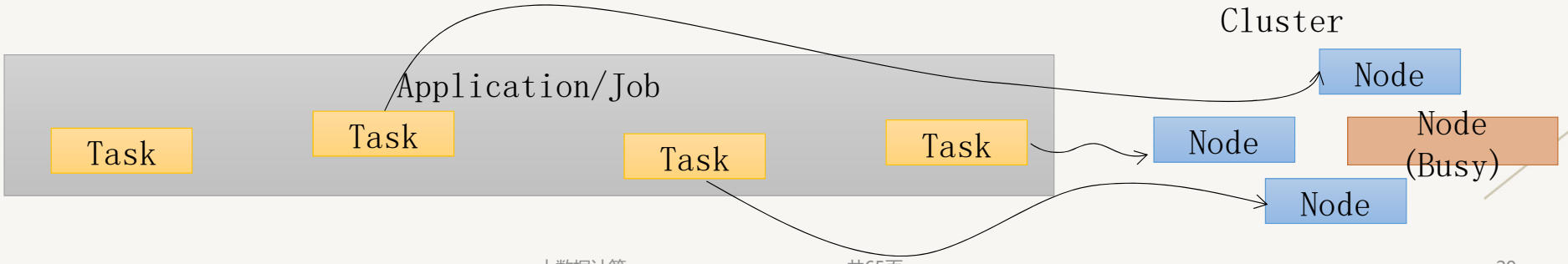
- HDFS要在大型集群中可靠地存储很大的文件。因此它将文件分块，并为每个块生成多个副本
- 每个文件的块大小和副本数量是可配置的
- NameNode 做出有关块复制的所有决定。它定期从集群中的每个 DataNode 接收 Heartbeat 和 Blockreport。收到心跳意味着 DataNode 运行正常。Blockreport 包含 DataNode 上所有块的列表


Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

- 对同一个机架内的访问速度要快过跨机架访问，但机架也会出现故障，有一定可能会整个机架一起挂掉。因此如何放置副本对集群性能与可靠性有很大影响：
  - 全放在一个机架上，机架挂掉数据就访问不到了
  - 全放在不同机架上，写操作成本就变得很高
- Hadoop拥有Rack Awareness（机架感知）功能，通过它可以制定不同的副本放置策略
- 例如副本数为3时，选择写操作所在机架放置一个副本，另选一个机架放置两个副本
- 类似地，在进行读取时也会优先选择相同机架上的副本

YARN：调度器

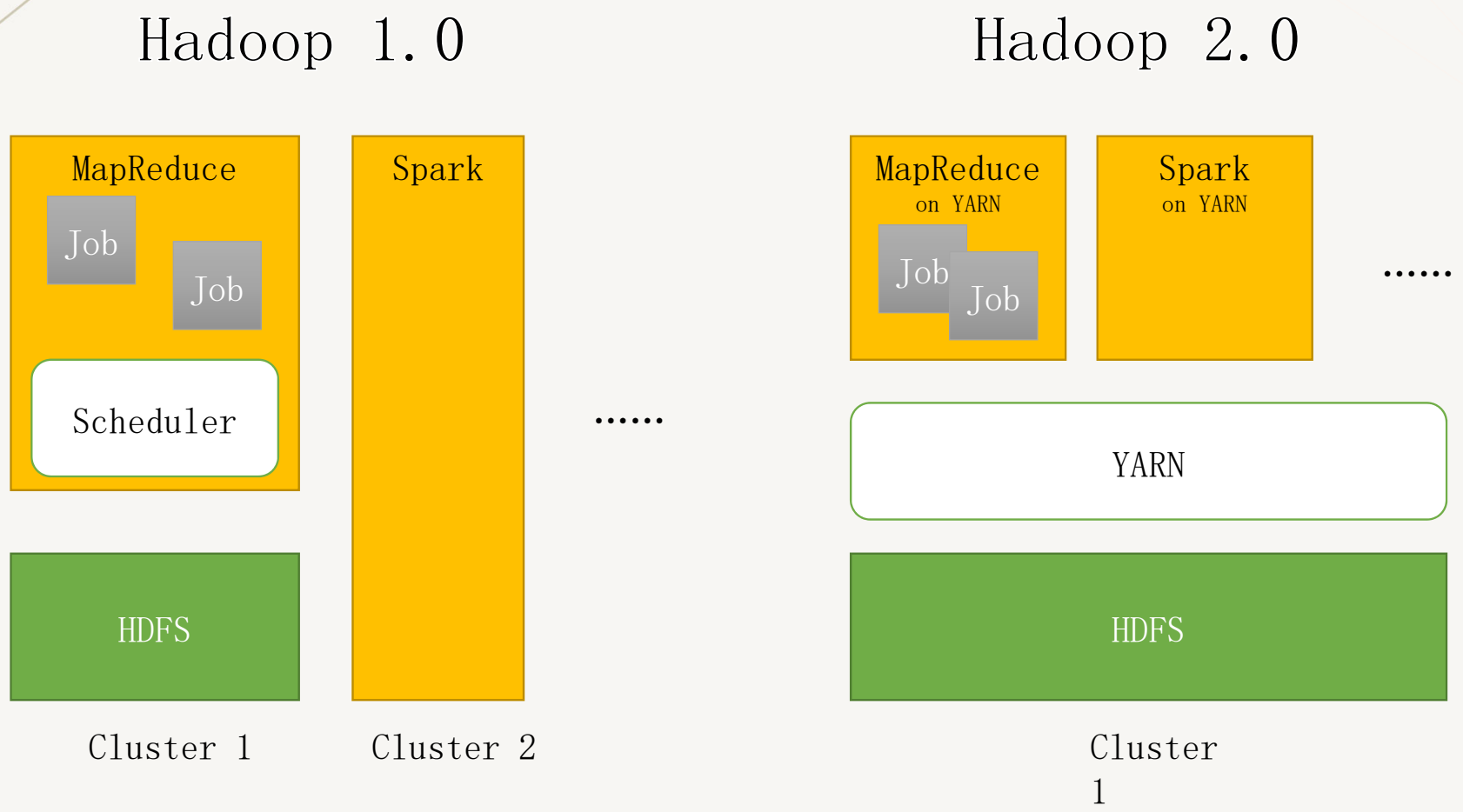
- 在Hadoop中，计算是以作业（job）的形式发布，并被划分为任务（task）的形式执行
- 计算任务的执行需要使用空闲计算资源（cpu等硬件资源）
- YARN就是用来调度管理计算任务和计算资源的框架



- 在谷歌的MapReduce论文中给出了一些对调度器的要求，但并没有明确给出其实现
- Hadoop 1.0版本中使用JobTracker与TaskTracker对MR任务进行调度，这种任务调度是与MR框架深度耦合的
- Hadoop 2.0中将资源和作业管理部分提取为独立的YARN框架，与MR解耦，不仅优化了调度方式，而且能够在其上支持更多的计算模型

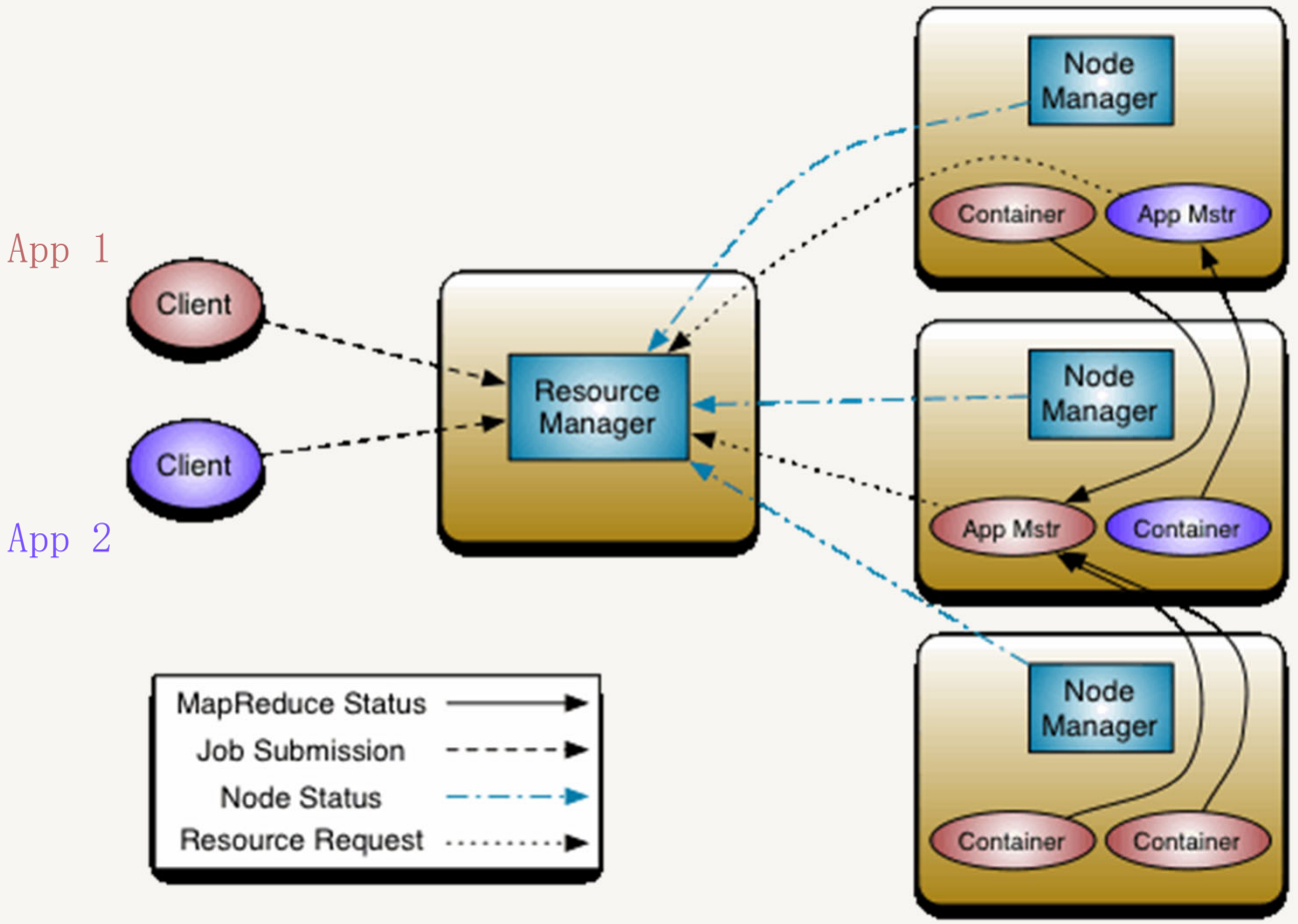
- 在之前没有YARN时，在Hadoop中只能使用MapReduce模型进行计算，这样一来整套体系耦合度很高。如果有其他的大数据计算框架如Spark等要运行，就需要另外的集群来支持。

- 使用YARN后，Hadoop的HDFS与调度部分与MR彻底解耦合，这样就可以让其他计算框架复用Hadoop的计算和存储资源，Hadoop变成了一个资源管理框架，而MR、Spark等变为该框架上的应用

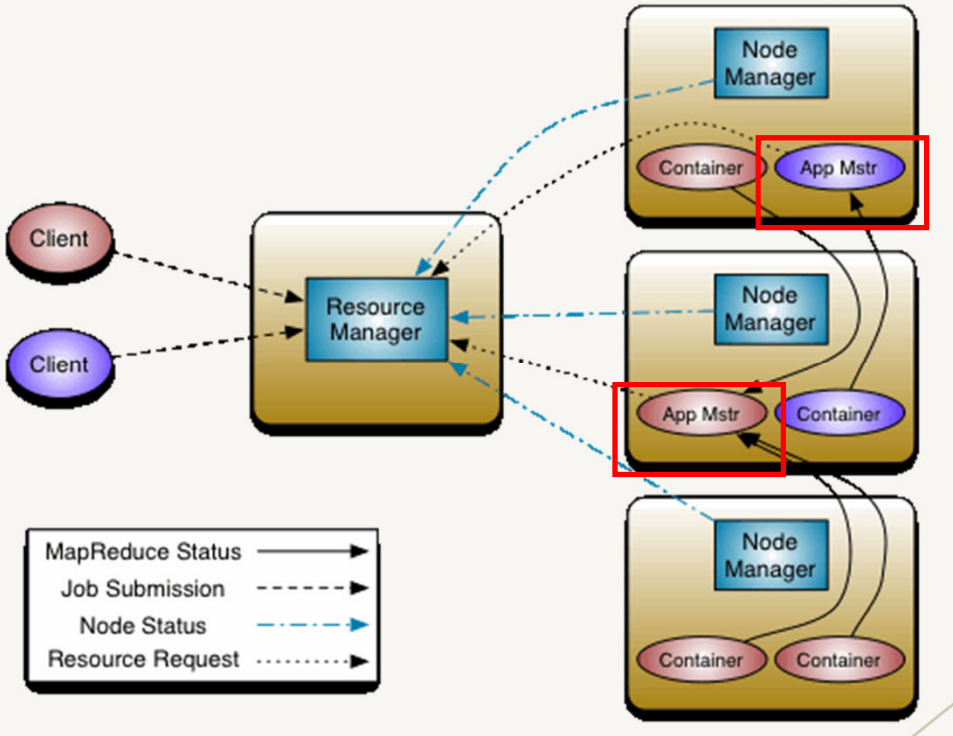

Hadoop 1.0
MapReduce
Job
Job
Scheduler
HDFS
Spark
.....
Cluster 1
Cluster 2
Hadoop 2.0
MapReduce
on YARN
Job
Job
Spark
on YARN
.....
YARN
HDFS
Cluster
1

YARN的主要工作包括：

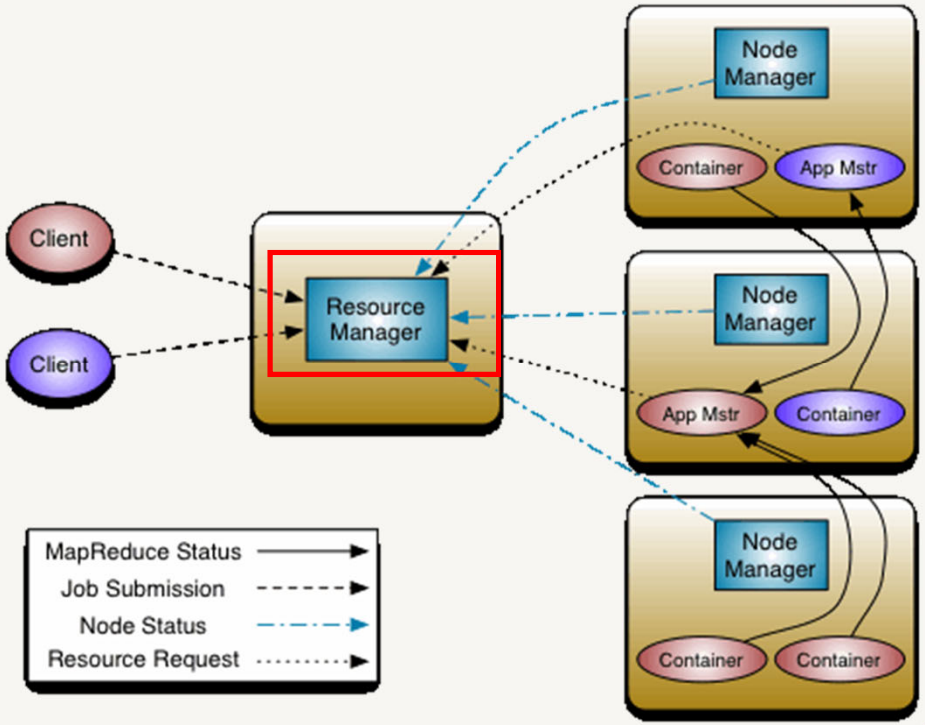
- 接受作业（应用程序），启动作业，失败时重启作业
- 管理集群资源（基于容器，囊括了内存、cpu、磁盘、网络等）
- 创建、管理、监控各节点上的容器（任务执行的微环境）
- 为应用程序按需分配资源（将任务发放到适当的节点、适当的容器）
- 跟踪任务的执行，监控任务健康状态，处理任务的失败


App 1
Client
Client
App 2
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

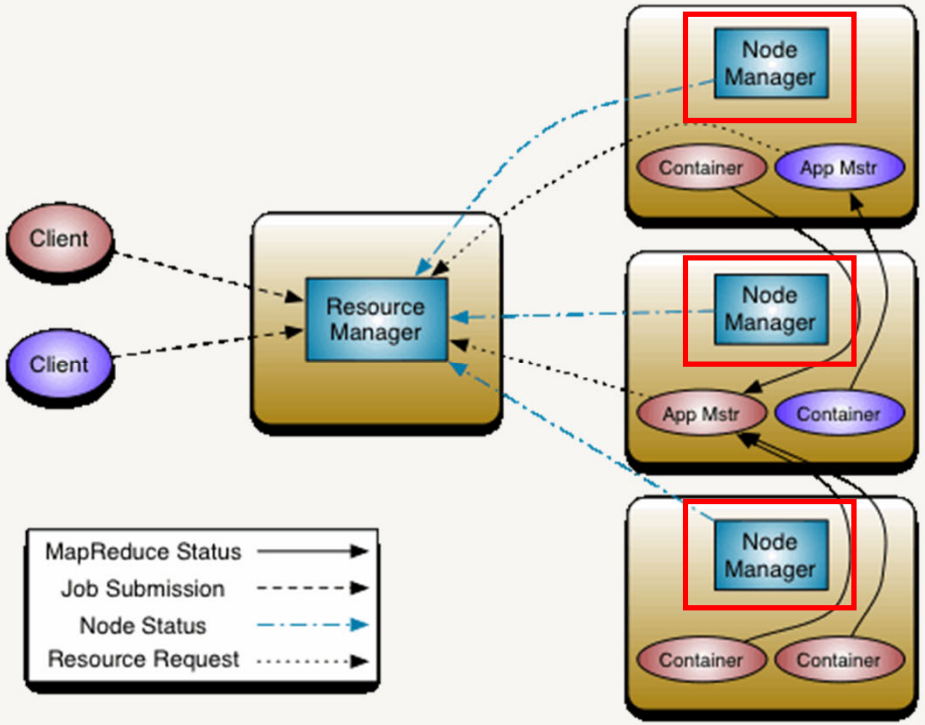
- ApplicationMaster：是一个框架特定的库（MR有MR的，Spark有Spark的）。每个应用程序一个，是该应用的“管家”。负责与ResourceManager协商资源，与NodeManager一起执行和监视任务的执行进度和状态



- ResourceManager：在系统中所有应用程序之间仲裁资源的最终权威，包含Scheduler和ApplicationsManager两个组件
- Scheduler负责为各种应用程序按需分配资源，资源基于容器。其分配策略是可插拔的，可以按需选择
- ApplicationsManager负责接收作业提交，协商用于启动ApplicationMaster的容器，并在失败时重启ApplicationMaster容器



- NodeManager：每台机器上的代理，负责管理容器，监控其资源使用情况（cpu、内存、磁盘、网络）以及Contaioner的运行状态，并将这些信息汇报给ResourceManager中的Scheduler。
- Contaioner：是一个动态资源分配单位，将内存、CPU、磁盘、网络等资源封装在一起，从而限定每个人物使用的资源量。可以理解为Yarn的资源抽象，相当于一台小型机器，真正运行任务的地方。



MapReduce：编程模型框架

- MapReduce（后文称MR）是一个用于编写并行大数据处理程序，并使其在集群上可靠运行的编程框架
- MR操作的数据存储在分布式文件系统上，也就是在磁盘中（在Hadoop上就是HDFS）
- MR的输入输出数据形式均为键值对〈k, v〉
- 一个MR作业通常将数据分为多个部分，每个部分分别由map操作生成中间值，然后由reduce操作对具有相同key的所有value进行汇总

- 用户编写的Map函数，接受输入对，产出一系列中间k-v对
- MR库将同一个中间key对应的所有中间value收集起来，并发送给Reduce
- 用户编写的Reduce函数，接受中间key，以及该key对应的所有value的集合，并将其整合为一个更小的集合（一般只有0或1个元素）

API（Application Programming Interface）VS ABI（Application Binary Interface）

举个例子，比如POSIX的pirntf()函数，在API层面，他能保证所有支持POSIX标准的系统之间都一样，但是它不能保证printf在实际系统中执行时，是否都遵循从右到左将参数压入堆栈，参数如何在堆栈中分布等这些实际运行时的二进制级别的问题。比如，两台计算机，一个是Intel X86，一个台是MIPS，他们都安装了Linux操作系统，因为Linux操作系统支持POSIX标准，所以两台电脑应该都是支持printf函数的，但是在实际运行时，两台电脑printf函数被调用过程中，关于参数和堆栈分布的细节在不同的机器上肯定是不一样的，甚至调用printf函数的指令都不一样，也就是说，API相同，ABI完全可能不同。

示例：字数统计

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

（伪代码）

输入：〈文档名，文档内容〉

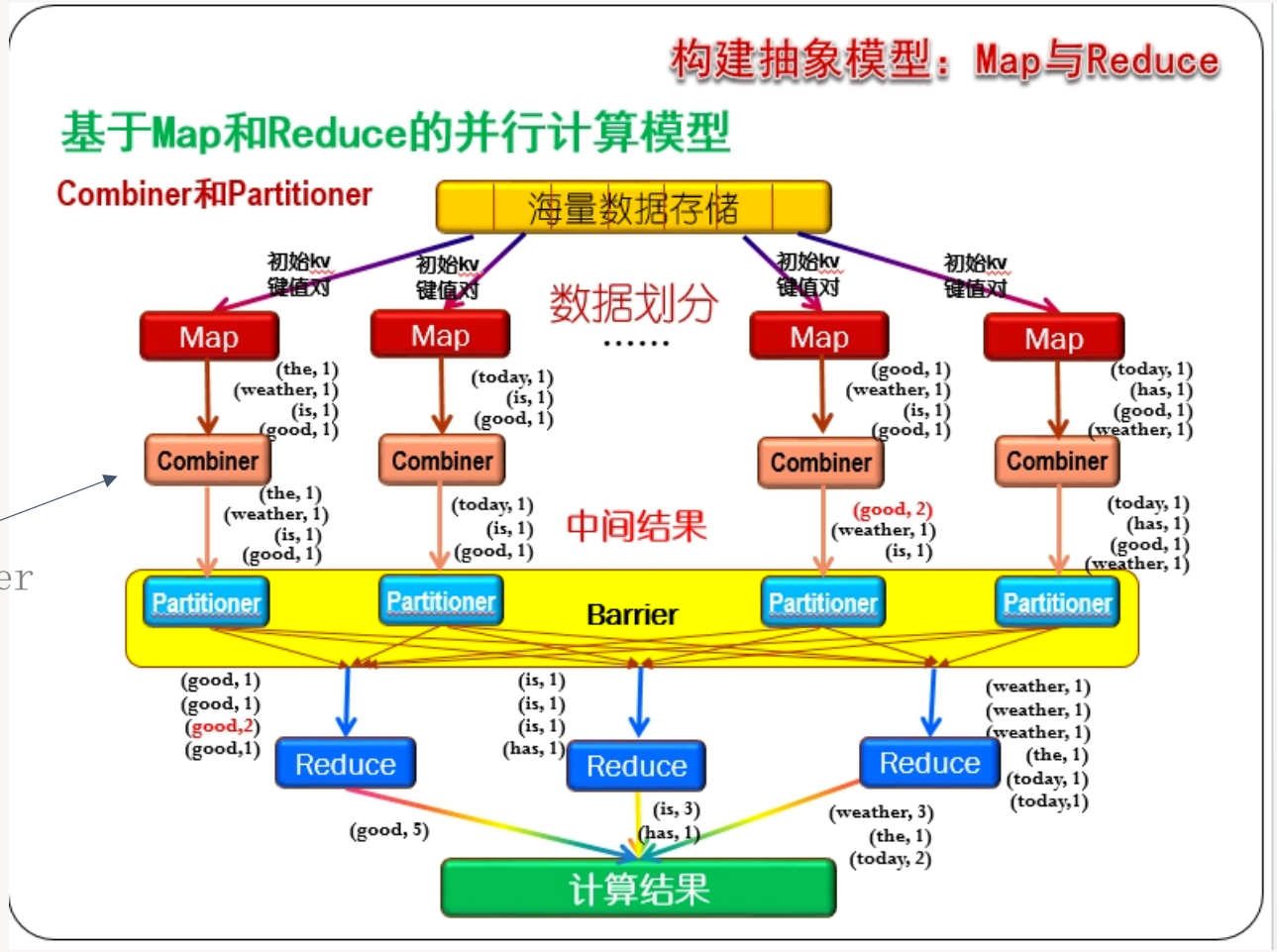
遍历文档中每个单词

产生中间对：〈单词，“1”〉

输入：〈单词，数量的列表〉

遍历数量列表，累加所有数量

产生结果



示例中不包含Combiner

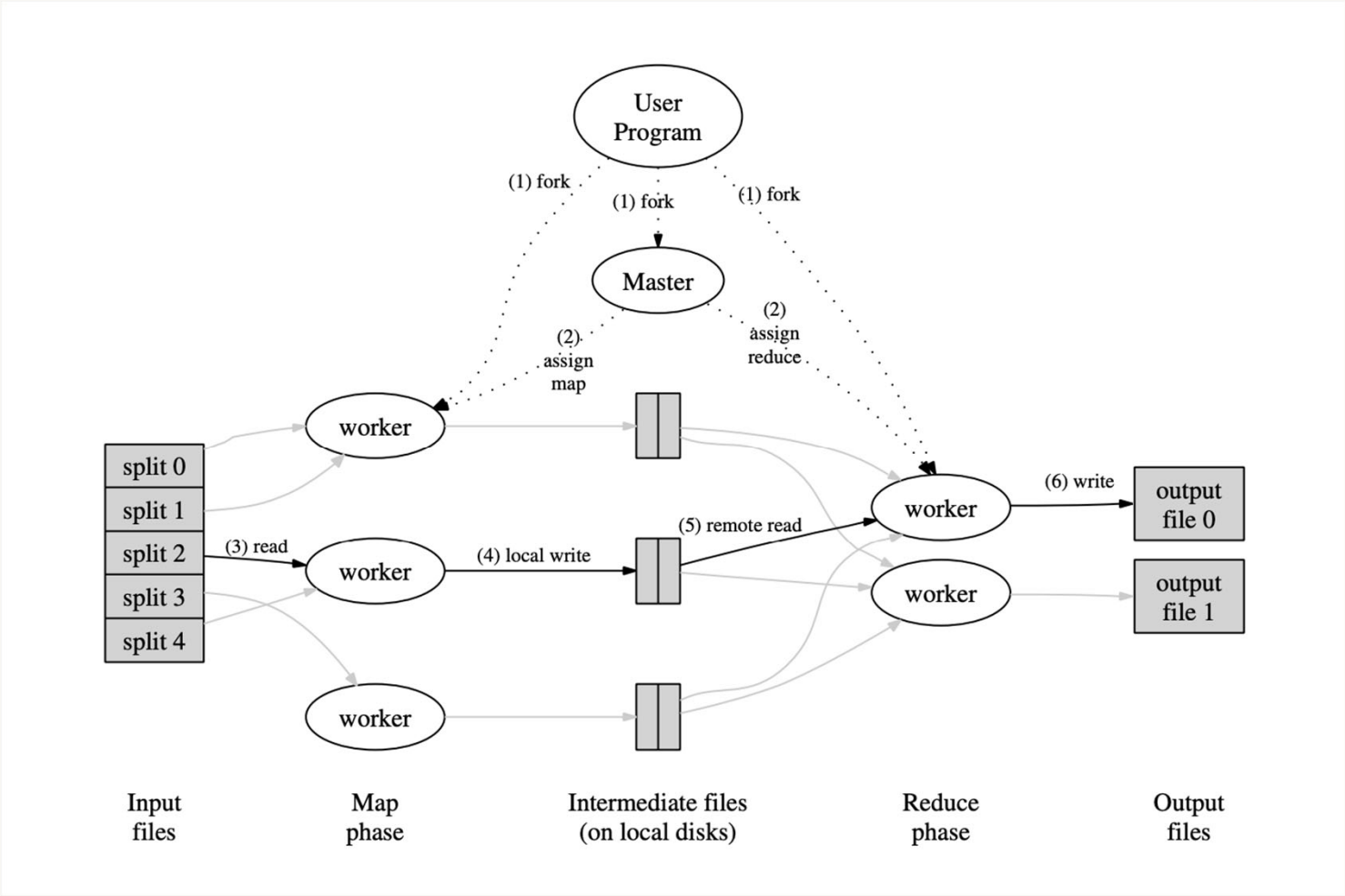
Map与Reduce操作的是数据分片而非所有数据，因此在各机器、各分片上的操作是并行的

- 数据被切分成块
- 在各机器(map worker)上启动代码副本，执行map操作，读入分块并输出中间值

并行

- （对中间值按key进行排序）
- 在各机器(reduce worker)上启动代码副本，执行reduce操作，读入各自key对应的中间值并生成结果

并行


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
split 0
split 1
split 2
split 3
split 4
(3) read
worker
(4) local write
worker
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Hadoop MapReduce 编程——以WordCount为例

- Hadoop MapReduce编程语言为Java
- 必要的包：

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

- 首先在main函数中建立一个工作（Job）：

```
public class WordCount { ...
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    ...
}}
```

Mapper

- 应用程序通过实现Mapper接口来提供map方法
- Mapper将输入键/值对映射到一组中间键/值对
- 转换后的中间记录不需要与输入记录的类型相同。给定的输入对可以映射到零个或多个输出对

Java

```
public class WordCount { ...

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
...
}
```

重写map方法

- 使用 Job.setMapperClass(Class) 把Mapper传递给Job。框架会为输入中的每个分块（键值对）调用map函数

```
job.setMapperClass(TokenizerMapper.class);
```

- 可以选择使用Combiner在Mapper本地对输出进行聚合，从而减少传输到Reducer的数据量

```
job.setCombinerClass(IntSumReducer.class);
```

- 注： IntSumReducer是稍后我们会编写的Reducer，这里相当于先在Map本地进行了一次Reduce

需要多少个Map?

- Map的数量一般由输入的大小（也就是有多少个输入块）决定
- 比如，如果有10TB的输入，块大小（blocksize）是128MB，就有约82000个map要运行

Reducer

- 应用程序通过实现Reducer接口来提供reduce方法
- Reducer将同一个key所对应的大量中间值进行约简

```
public class WordCount { ...

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

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

- 使用 Job.setReducerClass(Class) 把Reducer传递给Job。框架会为输入中的每个分块（键值对）调用reduce函数

```
job.setReducerClass(IntSumReducer.class);
```

reduce分为三个阶段：

- 框架首先将所有相关的map输出分片取回
- 框架将取回的中间值按key进行分组、排序
  - 以上两步是同时进行的，取回数据时就会进行合并操作
  - 可以使用Job.setGroupingComparatorClass(Class)、Job.setSortComparatorClass(Class)控制分组和排序
- 以上操作结束后，开始进行reduce

需要多少Reduce?

- Hadoop建议Reduce数量为
  $$0.95\ or\ 1.75\ *(<no.of\ nodes>*<no.of\ max.\ containers\ per\ node>)$$
- 取0.95时，所有reduce都能立即启动
- 取1.75时，执行快的节点可以在第一轮结束后开启第二轮reduce
- 随着reduce数量增加，框架开销会增加，但也会增加负载均衡，降低发生故障后的成本
- 因数略小于整数是为了为推测任务和失败任务留出一些空余
- 可以用 Job.setNumReduceTasks(int) 设定reduce数量

推测执行

- 在集群上分布式执行任务时，总会有一些节点跑得比其他节点慢很多
- Hadoop默认情况下不会傻傻等着慢节点运行完，如果发现有的任务执行比平均速度慢，它会尝试开启一个与该任务相同的“推测任务”
- 原任务和推测任务谁先跑完就用谁，另一个会被终止
- 推测执行会占用更多集群资源，可以通过配置将其关闭

完整代码 1/3

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

完整代码 2/3

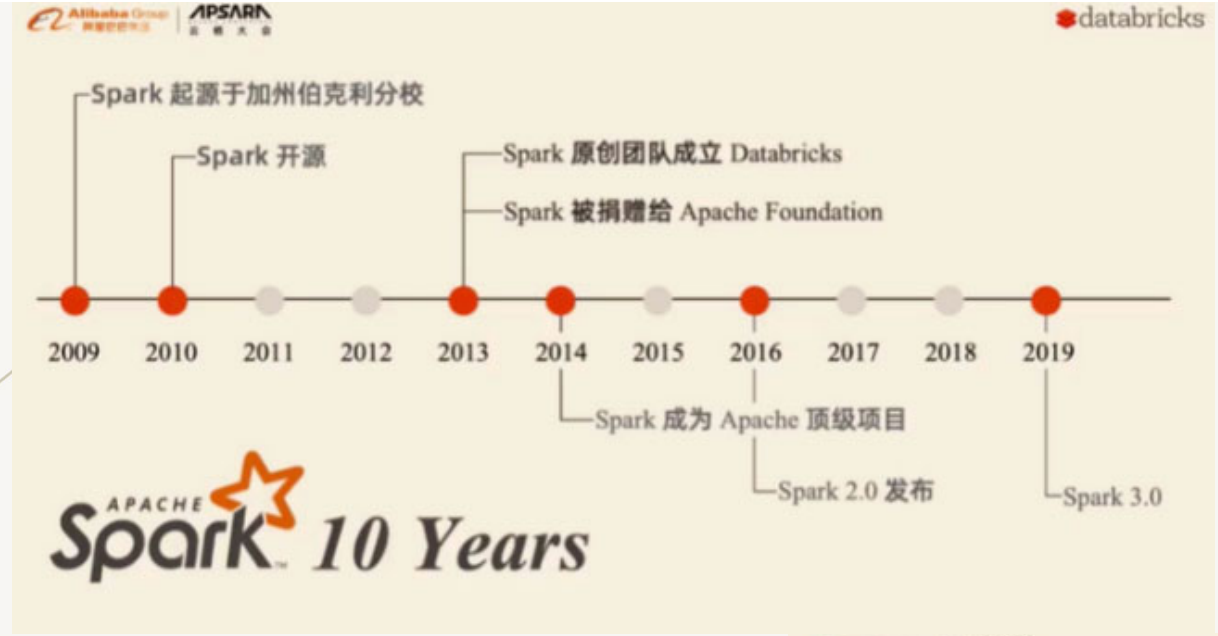
```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

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

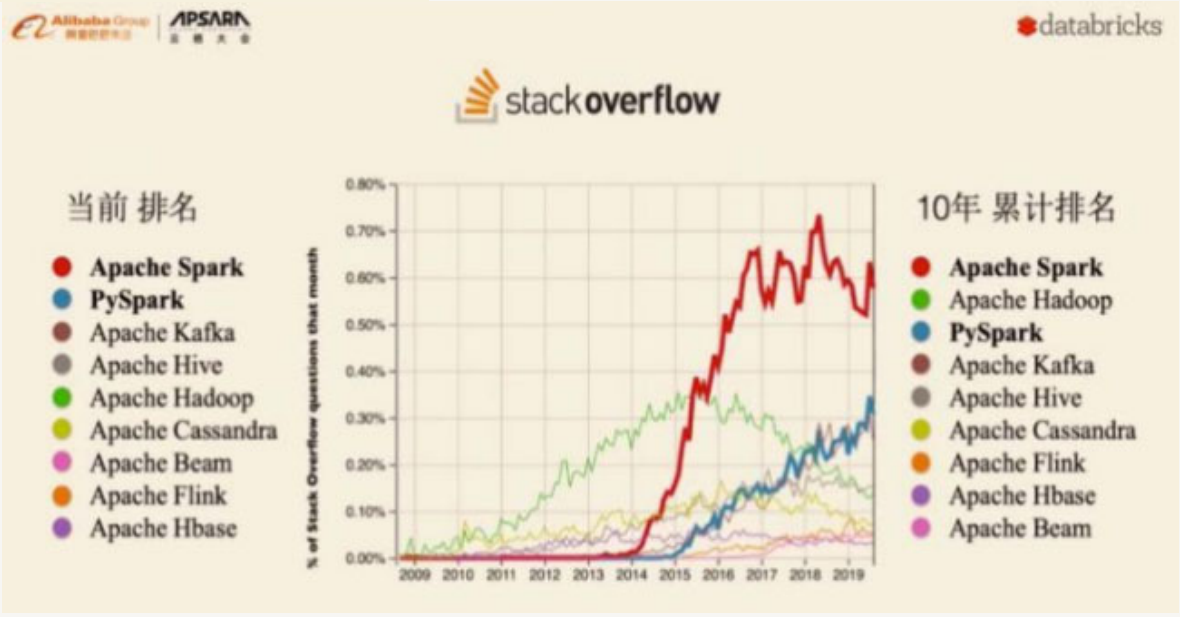
完整代码 3/3

```
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

新兴的大数据处理框架

- MapReduce框架解决了在集群上处理大数据的问题，在当时看来是突破性的创造。但随着技术的发展，人们对于大数据的处理也提出了更多的要求
- 许多新的大数据框架应运而生，其中最成功的一个便是Spark（ Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing NSDI12）


Alibaba Group
APSARA
databricks
Spark 起源于加州伯克利分校
Spark 开源
Spark 原创团队成立 Databricks
Spark 被捐赠给 Apache Foundation
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
Spark 成为 Apache 顶级项目
Spark 2.0 发布
Spark 3.0
APACHE Spark 10 Years


Alibaba Group
APSARA
databricks
stackoverflow
当前 排名
Apache Spark
PySpark
Apache Kafka
Apache Hive
Apache Hadoop
Apache Cassandra
Apache Beam
Apache Flink
Apache Hbase
% of Stack Overflow questions that month
0.80%
0.70%
0.60%
0.50%
0.40%
0.30%
0.20%
0.10%
0.00%
2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019
10年 累计排名
Apache Spark
Apache Hadoop
PySpark
Apache Kafka
Apache Hive
Apache Cassandra
Apache Flink
Apache Hbase
Apache Beam

MapReduce怎么不够用了？

- 编程范式较为局限，有些复杂任务（如机器学习）可能需要非常多次MR任务连接起来才能完成
- 每次MR任务都要进行大量磁盘IO，没有缓存，执行效率不高
- 不适用于低延迟要求的流式数据处理场合
- 语言支持有限
- ……

	Hadoop	Spark
类型	基础平台, 包含计算, 存储, 调度	纯计算工具(分布式)
场景	海量数据批处理(磁盘迭代计算)	海量数据的批处理(内存迭代计算、交互式计算)、海量数据流计算
价格	对机器要求低, 便宜	对内存有要求, 相对较贵
编程范式	Map+Reduce, API 较为底层, 算法适应性差	RDD组成DAG有向无环图, API 较为顶层, 方便使用
数据存储结构	MapReduce中间计算结果在HDFS磁盘上, 延迟大	RDD中间运算结果在内存中，延迟小
运行方式	Task以进程方式维护, 任务启动慢	Task以线程方式维护, 任务启动快，可批量创建提高并行能力

Spark 简介

- 一种专用于大规模数据分析的框架
- 可置于内存的弹性分布式数据集 RDD（以及新版本的 Dataset）
- 资源调度方面支持批处理或实时流式处理
- 在大规模数据和集群上进行数据分析与机器学习
- 多语言支持

Spark 示例

- Spark支持多种编程语言
- 在这里用Scala语言（基于JVM）展示一个简单的交互式分析案例
- 首先运行Spark Shell

```
./bin/spark-shell
```

- Dataset是Spark中的一个主要抽象，是一个分布式的数据项目集合。它可以从HDFS或以其他方式输入
- 创建一个基于文本文件README.md的Dataset：

```
scala> val textFile = spark.read.textFile("README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

- 可以从Dataset使用操作直接获取数据，或对Dataset进行转换等
- 这里我们进行操作（action）：

```
scala> textFile.count() // Number of items in this Dataset
res0: Long = 126      // May be different from yours as README.md will
                      // change over time, similar to other outputs

scala> textFile.first() // First item in this Dataset
res1: String = # Apache Spark
```

- 对Dataset进行转换（transformation），这里我们使用过滤器（filter），从子集产生一个新Dataset：

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string]
```

- Transformation 和 action 可以链接起来：

```
scala> textFile.filter(line => line.contains("Spark")).count()
                                        // How many lines contain "Spark"?
res3: Long = 15
```

- 通过map和reduce操作，找到单词数量最多的行

```
scala> textFile.map(line => line.split(" ").size).
             reduce((a, b) => if (a > b) a else b)

res4: Long = 15
```

- 以MapReduce的方式进行WordCount
- 用flatMap将行集合转为单词集合，然后用groupByKey进行聚合，使用count计算每个单词的数量

```
scala> val wordCounts = textFile.flatMap(
             line => line.split(" ")).
             groupByKey(identity).count()
wordCounts: org.apache.spark.sql.Dataset[(String, Long)] = [value:
string, count(1): bigint]

scala> wordCounts.collect()
res6: Array[(String, Int)] = Array((means,1), (under,2), (this,3),
(Because,1), (Python,2), (agree,1), (cluster.,1), ...)
```