Hadoop

极客学院出版

前言

主要记录了Hadoop各个组件的基本原理，处理过程和关键的知识点等，包括HDFS、YARN、MapReduce等。

本教程内容来源于 PennyWong (http://pennywong.gitbooks.io/hadoop-notebook/content/index.html)

更新日期	更新内容
2015-5-7	Hadoop文档

铺垫

- 人产生数据的速度越来越快，机器则更加快，more data usually beats better algorithms，所以需要另外的一种处理数据的方法。
- 硬盘的容量增加了，但性能没有跟上，解决办法是把数据分到多块硬盘，然后同时读取。但带来一些问题：

硬件问题：复制数据解决（RAID）

分析需要从不同的硬盘读取数据：MapReduce

而Hadoop提供了

1.可靠的共享存储（分布式存储）2.抽象的分析接口（分布式分析）

大数据

概念

不能使用一台机器进行处理的数据

大数据的核心是样本=总体

特性

- **大量性(volume):** 一般在大数据里，单个文件的级别至少为几十，几百GB以上
- **快速性(velocity):** 反映在数据的快速产生及数据变更的频率上
- **多样性(variety):** 泛指数据类型及其来源的多样化，进一步可以把数据结构归纳为结构化(structured)，半结构化(semi-structured)，和非结构化(unstructured)

- **易变性:** 伴随数据快速性的特征，数据流还呈现一种波动的特征。不稳定的数据流会随着日，季节，特定事件的触发出现周期性峰值
- **准确性:** 又称为数据保证(data assurance)。不同方式，渠道收集到的数据在质量上会有很大差异。数据分析和输出结果的错误程度和可信度在很大程度上取决于收集到的数据质量的高低
- **复杂性:** 体现在数据的管理和操作上。如何抽取，转换，加载，连接，关联以把握数据内蕴的有用信息已经变得越来越有挑战性

关键技术

1.数据分布在多台机器

可靠性：每个数据块都复制到多个节点

性能：多个节点同时处理数据

2.计算随数据走

网络IO速度 << 本地磁盘IO速度，大数据系统会尽量地将任务分配到离数据最近的机器上运行（程序运行时，将程序及其依赖包都复制到数据所在的机器运行）

代码向数据迁移，避免大规模数据时，造成大量数据迁移的情况，尽量让一段数据的计算发生在同一台机器上

3.串行IO取代随机IO

传输时间 << 寻道时间，一般数据写入后不再修改

目录

1

Hadoop – 简介

Hadoop可运行于一般的商用服务器上，具有高容错、高可靠性、高扩展性等特点

特别适合写一次，读多次的场景

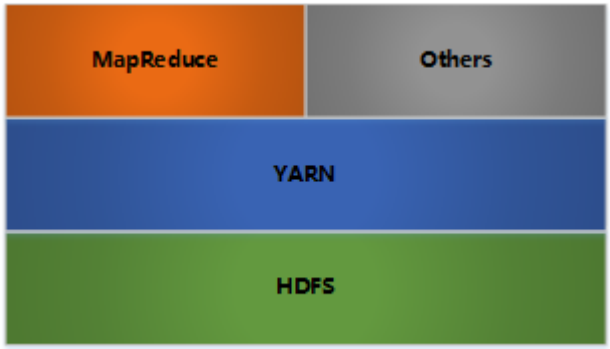
适合

- 大规模数据
- 流式数据（写一次，读多次）
- 商用硬件（一般硬件）

不适合

- 低延时的数据访问
- 大量的小文件
- 频繁修改文件（基本就是写1次）

Hadoop架构



- HDFS: 分布式文件存储
- YARN: 分布式资源管理
- MapReduce: 分布式计算
- Others: 利用YARN的资源管理功能实现其他的数据处理方式

内部各个节点基本都是采用Master-Woker架构

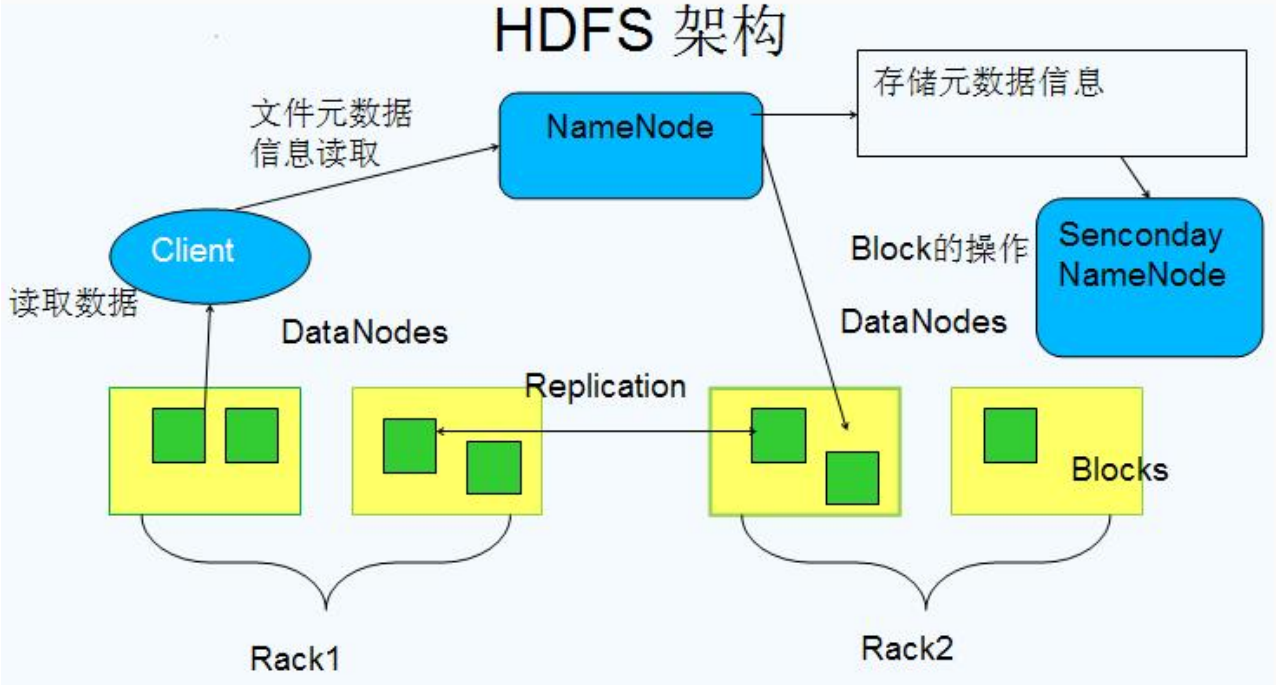
2

Hadoop – HDFS

简介

Hadoop Distributed File System，分布式文件系统

架构



- Block数据&##x5757;

  1. 基本存储单位，一般大小为64M（配置大的块主要是因为：1）减少搜寻时间，一般硬盘传输速率比寻道时间要快，大的块可以减少寻道时间；2）减少管理块的数据开销，每个块都需要在NameNode上有对应的记录；3）对数据块进行读写，减少建立网络的连接成本）
  2. 一个大文件会被拆分成一个个的块，然后存储于不同的机器。如果一个文件少于Block大小，那么实际占用的空间为其文件的大小
  3. 基本的读写S#x5355;位，类似于磁盘的页，每次都是读写一个块
  4. 每个块都会被复制到多台机器，默认复制3份

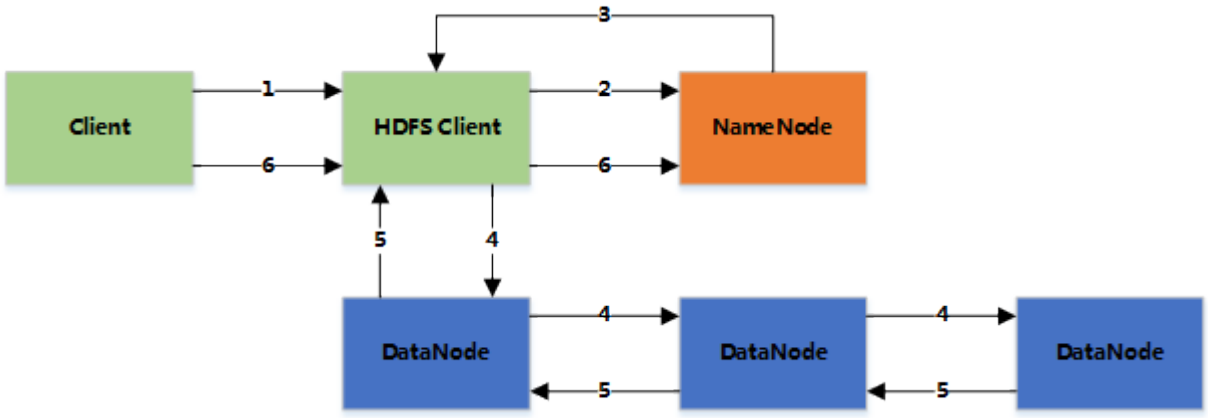
- NameNode

  1. 存储文件的metadata，运行时所有数据都保存到内存，整个HDFS可存储的文件数受限于NameNode的内存大小

2. 一个Block在NameNode中对应一条记录（一般一个block占用150字节），如果是大量的小文件，会消耗大量内存。同时map task的数量是由splits来决定的，所以用MapReduce处理大量的小文件时，就会产生过多的map task，线程管理开销将会增加作业时间。处理大量小文件的速度远远小于处理同等大小的大文件的速度。因此Hadoop建议存储大文件
3. 数据会定时保存到本地磁盘，但不保存block的位置信息，而是由DataNode注册时上报和运行时维护（NameNode中与DataNode相关的信息并不保存到NameNode的文件系统中，而是NameNode每次重启后，动态重建）
4. NameNode失效则整个HDFS都失效了，所以要保证NameNode的可用性

- Secondary NameNode
  1. 定时与NameNode进行同步（定期合并文件系统镜像和编辑日&#x#x5FD7;，然后把合并后的传给NameNode，替换其镜像，并清空编辑日志，类似于CheckPoint机制），但NameNode失效后仍需要手工将其设置成主机
- DataNode
  1. 保存具体的block数据
  2. 负责数据的读写操作和复制操作
  3. DataNode启动时会向NameNode报告当前存储的数据块信息，后续也会定时报告修改信息
  4. DataNode之间会进行通信，复制数据块，保证数据的冗余性

HDFS – 写文件



1.客户端将文件写入本地磁盘的N#x4E34;时文件中

2.当临时文件大小达到一个block大小时，HDFS client通知NameNode，申请写入文件

3.NameNode在HDFS的文件系统中创建一个文件，并把该block id和要写入的DataNode的列表返回给客户端

4.客户端收到这些信息后，将临时文件写入DataNodes

- 4.1 客户端将文件内容写入第一个DataNode（一般以4kb为单位进行传输）
- 4.2 第一个DataNode接收后，将数据写入本地磁盘，同时也传输给第二个DataNode
- 4.3 依此类推到最后一个DataNode，数据在DataNode之间是通过pipeline的方式进行复制的
- 4.4 后面的DataNode接收完数据后，都会发送一个确认给前一个DataNode，最终第一个DataNode返回确认给客户端
- 4.5 当客户端接收到整个block的确认后，会向NameNode发送一个最终的确认信息
- 4.6 如果写入某个DataNode失败，数据会继续写入其他的DataNode。然后NameNode会找另外一个好的DataNode继续复制，以保证冗余性
- 4.7 每个block都会有一个校验码，并存放到独立的文件中，以便读的时候来验证其完整性

5.文件写完后（客户端关闭），NameNode提交文件（这时文件才可见，?#x5982;果提交前，NameNode垮掉，那文件也就丢失了。fsync：只保证数据的信息写到NameNode上，但并不保证数据已经被写到DataNode中）

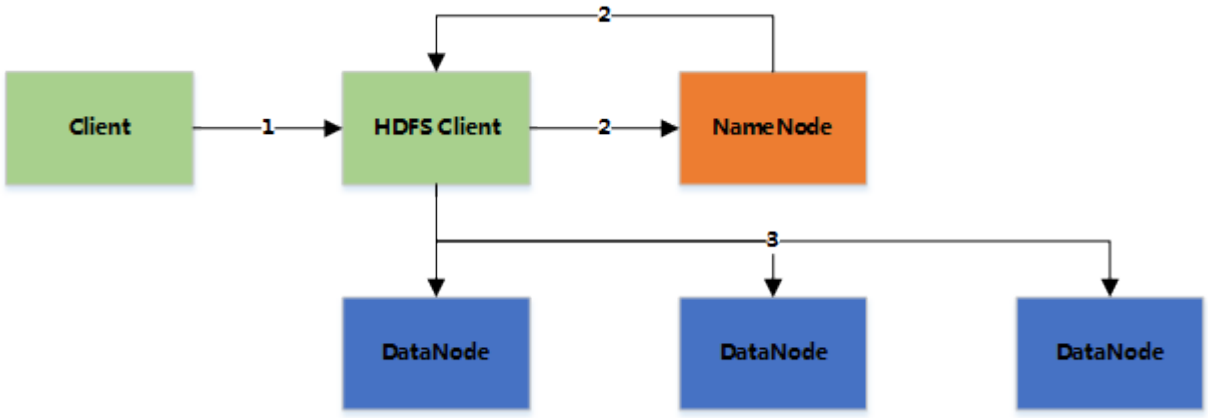
Rack aware（机架感知）

通过配置文件指定机架名和DNS的对应关系

假设复制参数是3，在写入文件时，会在本地的机架保存一份数据，然后在另外一个机架内保存两份数据（同机架内的传输速度快，从而提高性能）

整个HDFS的集群，最好是负载平衡的，这样才能尽量利用集群的优势

HDFS – 读文件



1. 客户端向NameNode发送读取请求
2. NameNode?#x8FD4;回文件的所有block和这些block所在的DataNodes（包括复制节点）
3. 客户端直接从DataNode中读取数据，如果该DataNode读取失败（DataNode失效或校验码不对），则从复制节点中读取（如果读取的数据就在本机，则直接读取，否则通过网络读取）

HDFS – 可靠性

1. DataNode可以失效

   DataNode会定时发送心跳到NameNode。如果η#x5728;一段时间内NameNode没有收到DataNode的心跳消息，则认为其失效。此时NameNode就会将该节点的数据（从该节点的复制节点中获取）复制到另外的DataNode中

2. 数据可以毁坏

   无论是写入时还是硬盘本身的问题，只要数据有问题（读取时通过校验码来检测），都可以通过其他的复制节点读取，同时还会再复制一份到健康的节点中

3. NameNode不可靠

HDFS – 命令工具

fsck: 检查文件的完整性

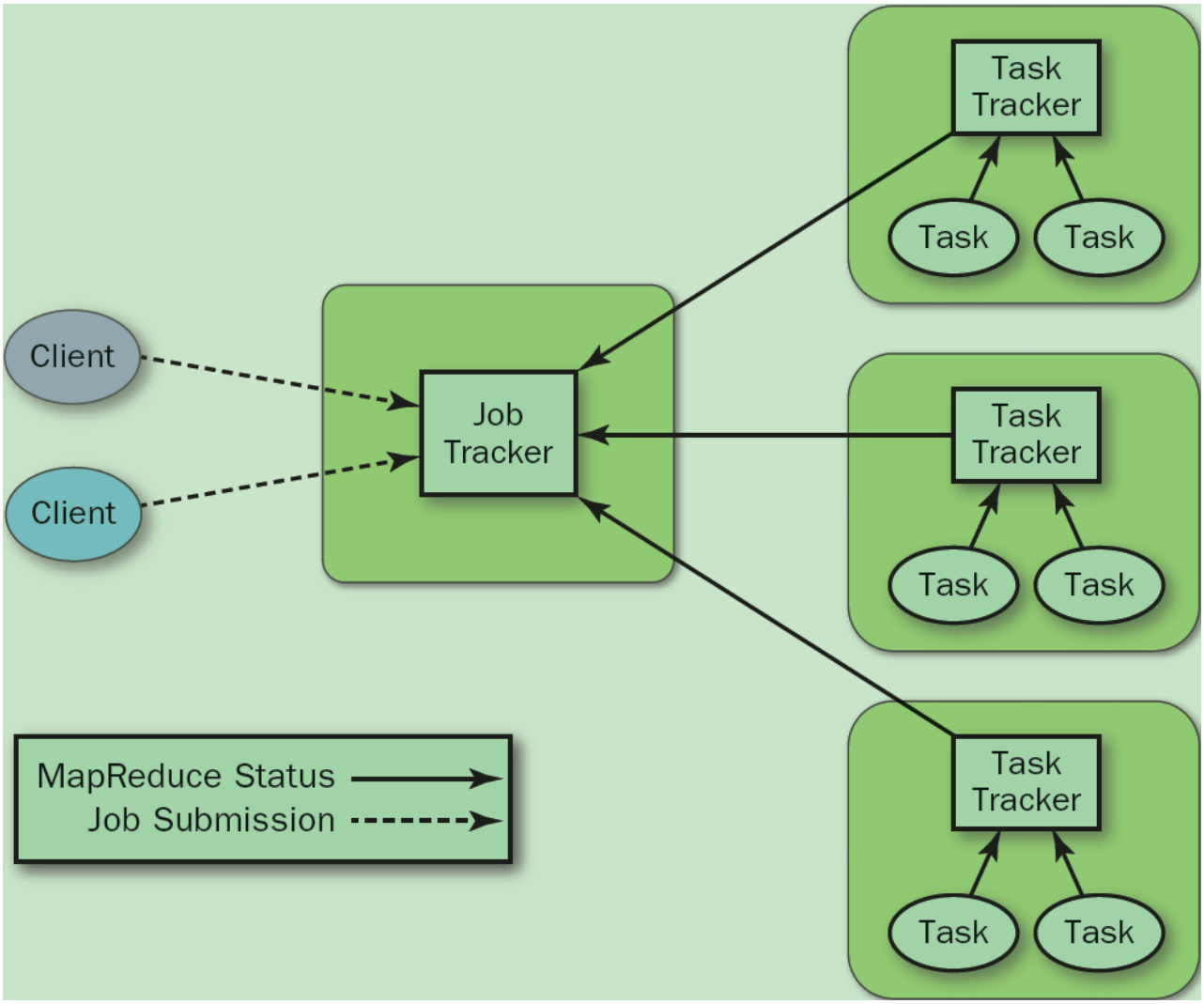
start-balancer.sh: 重新平衡HDFS

hdfs dfs -copyFromLocal 从本地磁盘复制文&#x#x4EF6;到HDFS

3

Hadoop – YARN

旧的MapReduce架构



- JobTracker: 负责资源管理，跟踪资源消耗和可用性，作业生命周期管理（调度作业任务，跟踪进度，为任务提供容错）
- TaskTracker: 加载或关闭任务，定时报告认为状态

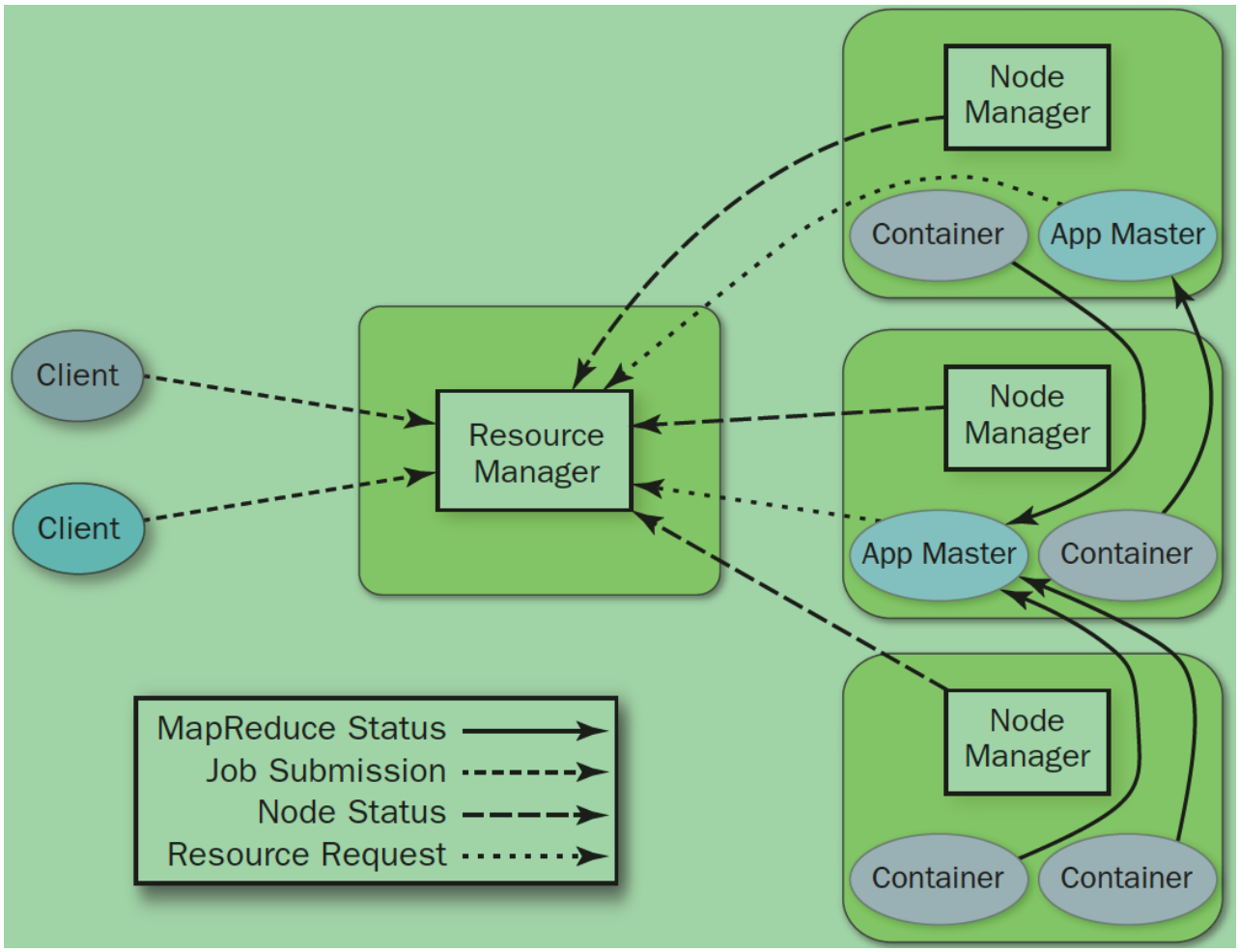
此架构会有以下问题：

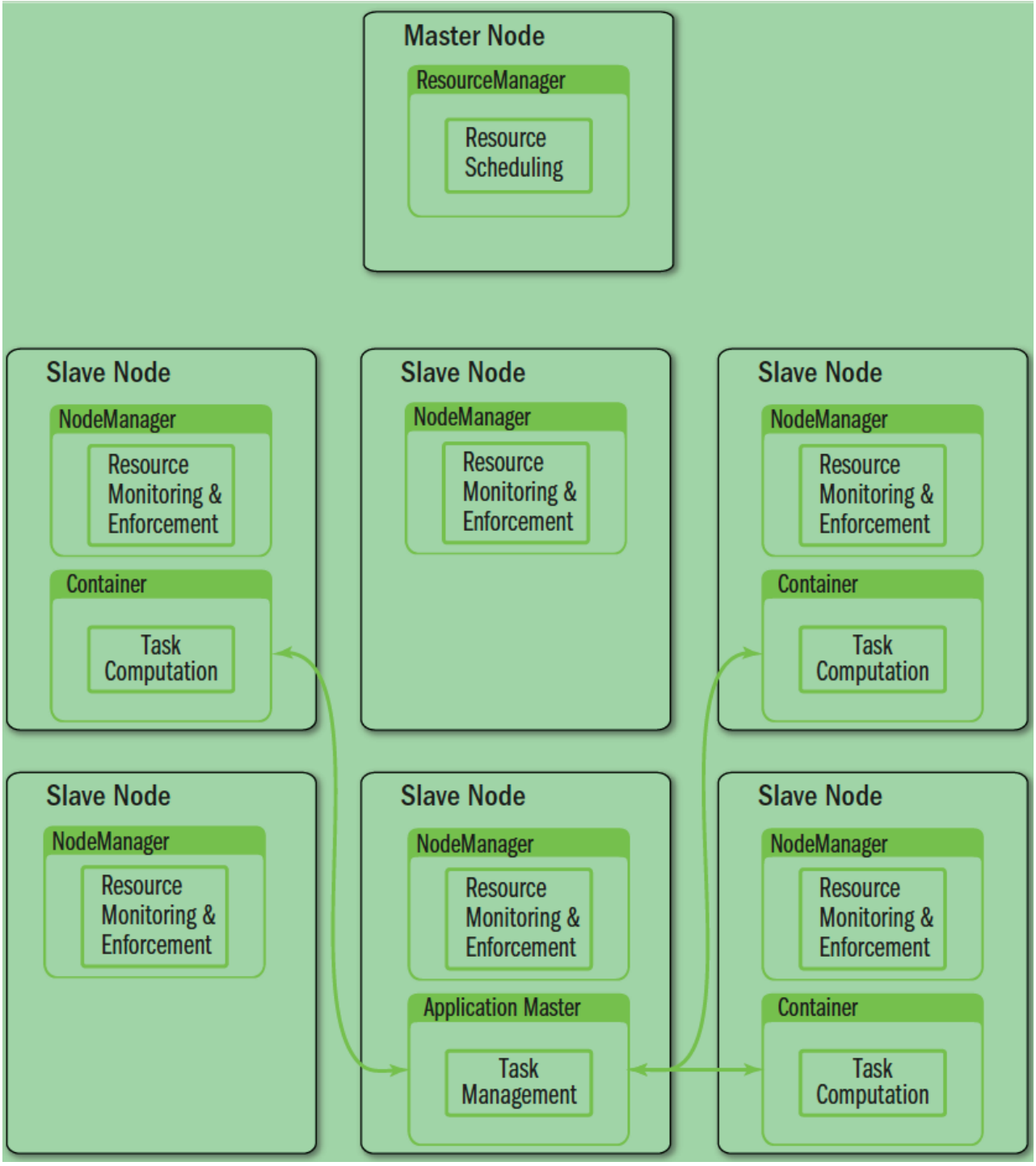
1. JobTracker是MapReduce的集中处理点，存在单点故障
2. JobTracker完成了太多的任务，造成了过多的资源消耗，当MapReduce job 非常多的时候，会造成很大的内存开销。这也是业界普遍总结出老Hadoop的MapReduce只能支持4000 节点主机的上限
3. 在TaskTracker端，以map/reduce task的数目作为资&##x6E90;的表示过于简单，没有考虑到cpu/ 内存的占用情况，如果两个大内存消耗的task被调度到了一块，很容易出现OOM

4. 在TaskTracker端，把资源强制划分为map task slot和reduce task slot, 如果当系统中只有map task或者只有reduce task的时候，会造成资源的浪费，也就集群资源利用的问题

总的来说就是**单点问题和资源利用率问题**

YARN架构



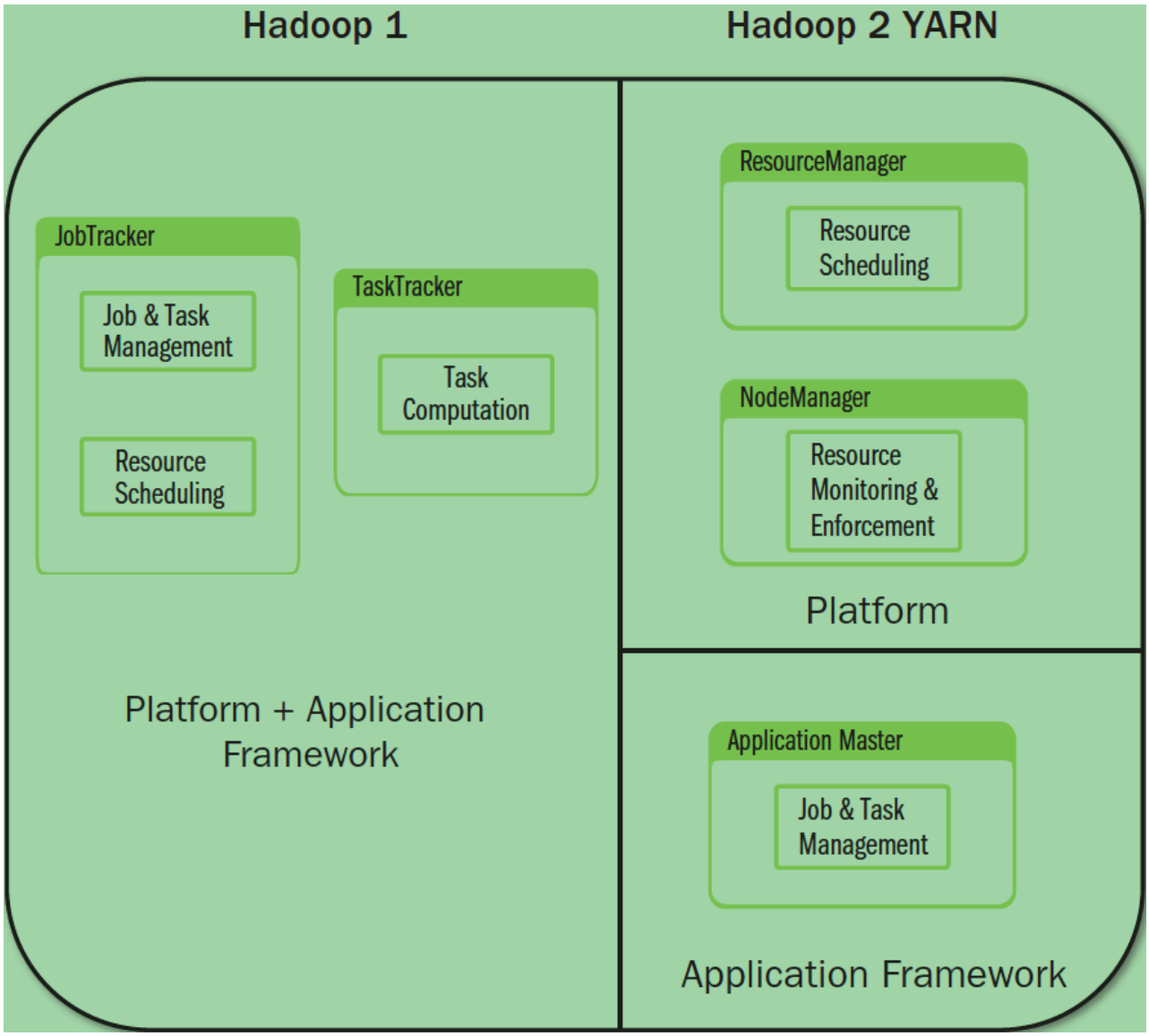


YARN就是将JobTracker的职责进行拆分，将资源管理和任务调度监控拆分成独立#x7ACB;的进程：一个全局的资源管理和一个每个作业的管理（ApplicationMaster）ResourceManager和NodeManager提供了计算资源的分配和管理，而ApplicationMaster则完成应用程序的运行

- ResourceManager: 全局资源管理和任务调度
- NodeManager: 单个节点的资源管理和监控
- ApplicationMaster: 单个作业的资源管理和任务监控

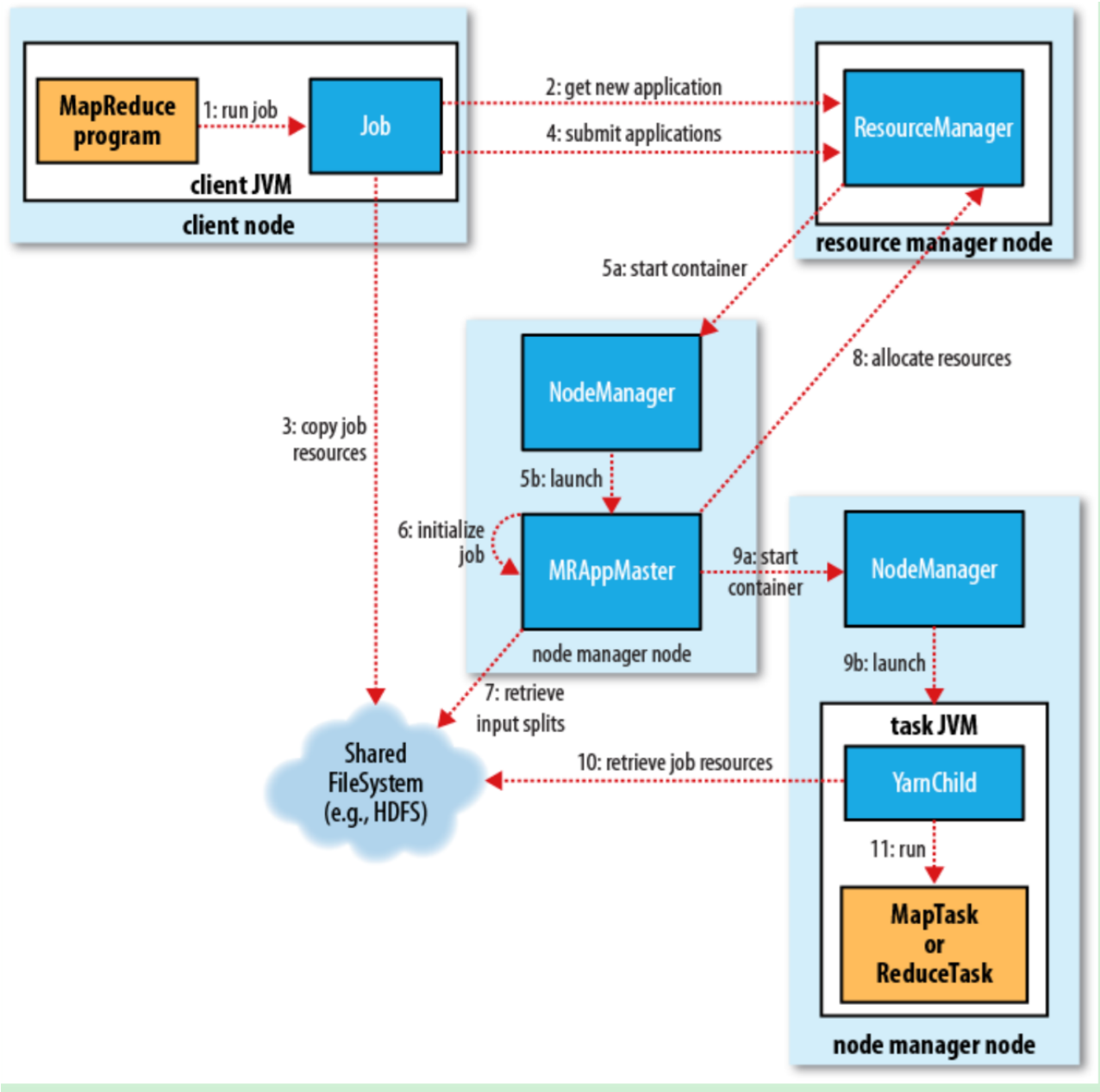
- Container: 资源申请的单位和任务运行的容器

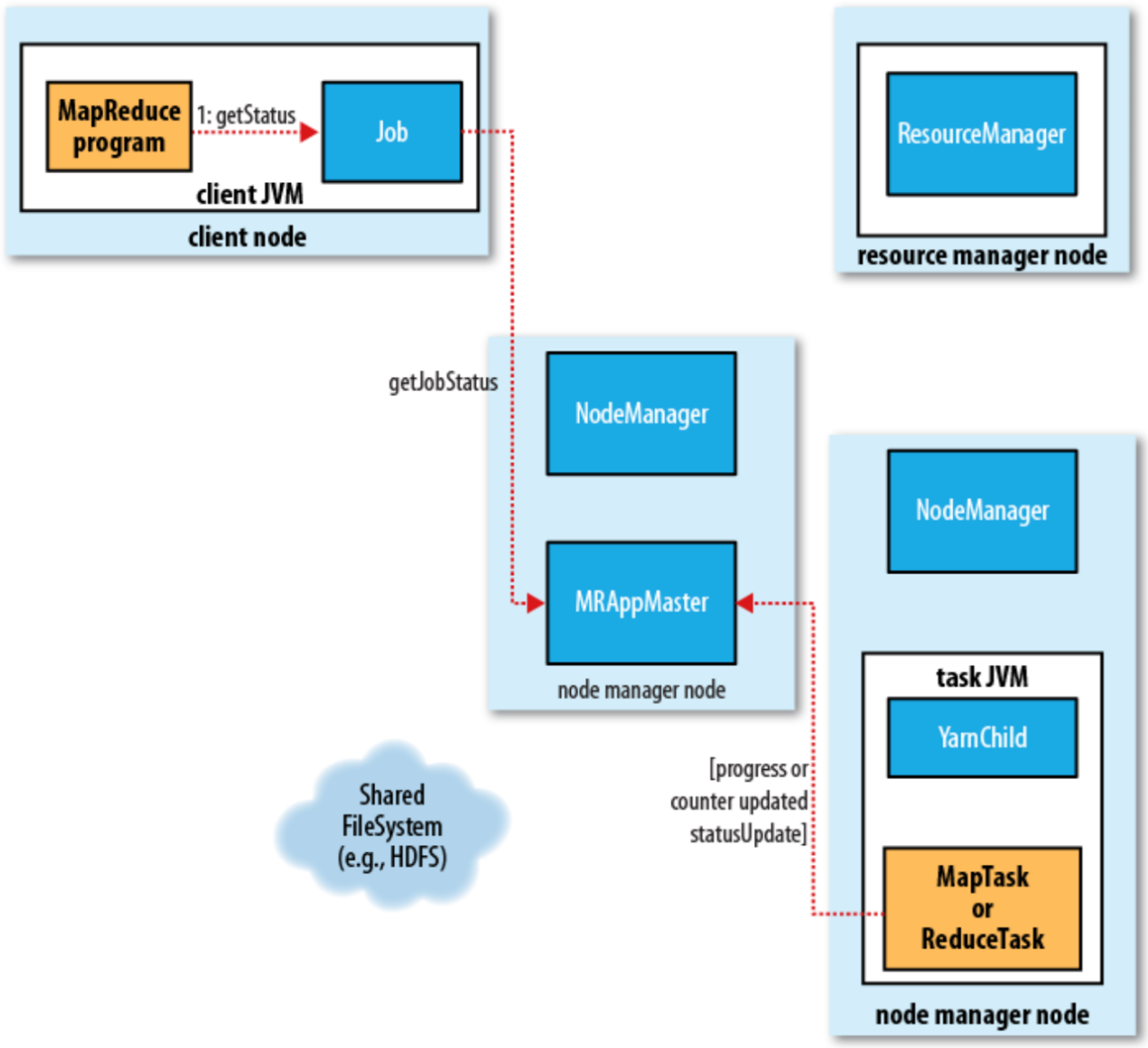
架构对比



YARN架构下形成了一个通用的资源管理平台和一个通用的应用计算^#x5E73;台，避免了旧架构的单点问题和资源利用率问题，同时也让在其上运行的应用不再局限于MapReduce形式

YARN基本流程


MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit applications
ResourceManager
resource manager node
5a: start container
NodeManager
5b: launch
6: initialize job
MRAppMaster
node manager node
3: copy job resources
8: allocate resources
9a: start container
NodeManager
9b: launch
task JVM
YarnChild
11: run
MapTask or ReduceTask
node manager node
7: retrieve input splits
10: retrieve job resources
Shared FileSystem (e.g., HDFS)



1. Job submission

从ResourceManager中获取一个Application ID 检查作业输出配置，计算输入分片 拷贝作业资源（job jar、配置文件、分片信息）到HDFS，以便后面任务的执行

2. Job initialization

ResourceManager将作业递交给Scheduler（有很多调度算法，一般是根据优先级）Scheduler为作业分配一个Container，ResourceManager就加载一个application master process并交给NodeManager管理ApplicationMaster主要是创建一系列的监控进程来跟踪作业的进度，同时获取输入分片，为每一个分片创建一个Map task和相应的reduce task Application Master还决定如何运行作业，如果作业很小（可配置），则直接在同一个JVM下运行

3. Task assignment

ApplicationMaster向Resource Manager申请资源（一个个的Container，指定任务分配的资源要求）一般是根据data locality来分配资源

4. Task execution

ApplicationMaster根据ResourceManager的分配情况，在对应的NodeManager中启动Container 从HDFS N#x4E2D;读取任务所需资源（job jar，配置文件等），然后执行该任务

5. Progress and status update

定时将任务的进度和状态报告给ApplicationMaster Client定时向ApplicationMaster获取整个任务的进度和状态

6. Job completion

Client定时检查整个作业是否完成 作业完成后，会清空临时文件、目录等

YARN - ResourceManager

负责全局的资源管理和任务调度，把整个集群当&##x6210;计算资源池，只关注分配，不管应用，且不负责容错

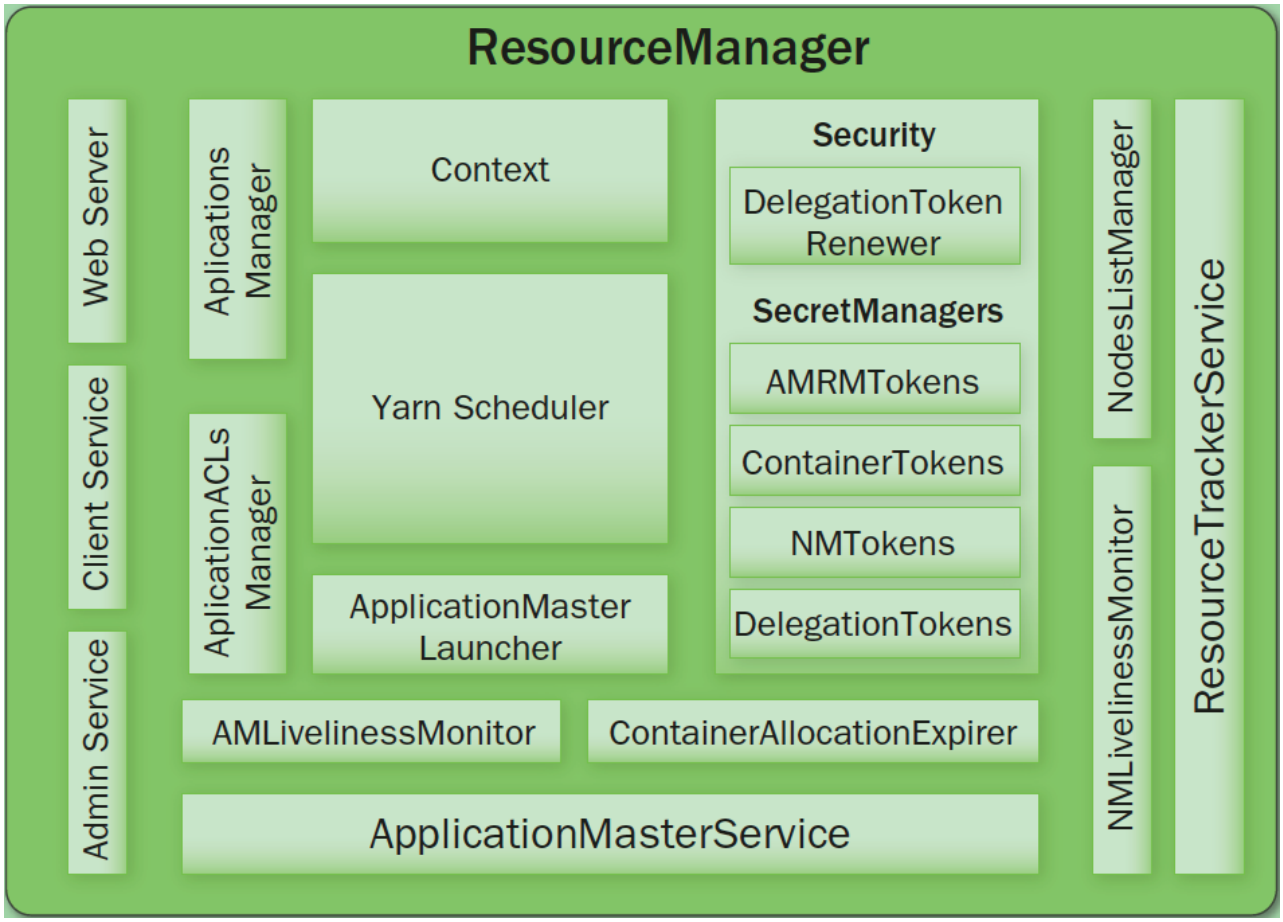
资源管理

1. 以前资源是每个节点分成一个个的Map slot和Reduce slot，现在是一个个Container，每个Container可以根据需要运行ApplicationMaster、Map、Reduce或者任意的程序
2. 以前的资源分配是静态的，目前是动态的，资源利用率更高
3. Container是资源申请的单位，一个资源申请格式：<resource-name, priority, resource-requirement, number-of-containers>, resource-name：主机名、机架名或*（代表任意机器）, resource-requirement：目前只支持CPU和内存
4. 用户提交作#x4F5C;业到ResourceManager，然后在某个NodeManager上分配一个Container来运行ApplicationMaster，ApplicationMaster再根据自身程序需要向ResourceManager申请资源
5. YARN有一套Container的生命周期管理机制，而ApplicationMaster和其Container之间的管理是应用程序自己定义的

任务调度

1. 只关注资源的使用情况，根据需求合理分配资源
2. Scheluer可以根据申请的需要，在特定的机器上申请特定的资源（ApplicationMaster负责申请资源时的数据本地化的考虑，ResourceManager将尽量满足其申请需求，在指定的机器上分配Container，从而减少数据移动）

内部结构



- Client Service: 应用提交、终止、输出信息（应用、队列、集群等的状态信息）
- Adaminstration Service: 队列、节点、Client权限管理
- ApplicationMasterService: 注册、终止ApplicationMaster, 获取ApplicationMaster的资源申请或取消的请求，并将其异步地传给Scheduler, 单线程处理
- ApplicationMaster Liveliness Monitor: 接收ApplicationMaster的心跳消息，如果某个ApplicationMaster在一定时间内没有发送心跳，则被任务失效，其资源将会被回收，然后ResourceManager会重新分配一个ApplicationMaster运行该应用（默认尝试2次）
- Resource Tracker Service: 注册节点, 接收各注册节点的心跳消息
- NodeManagers Liveliness Monitor: 监控每个节点的心跳消息，如果长时间没有收到心跳消息，则认为该节点无效, 同时所有在该节点上的Container都标记成无效，也不会调度任务到该节点运行
- ApplicationManager: 管理应用程序，记录和管理已完成的应用

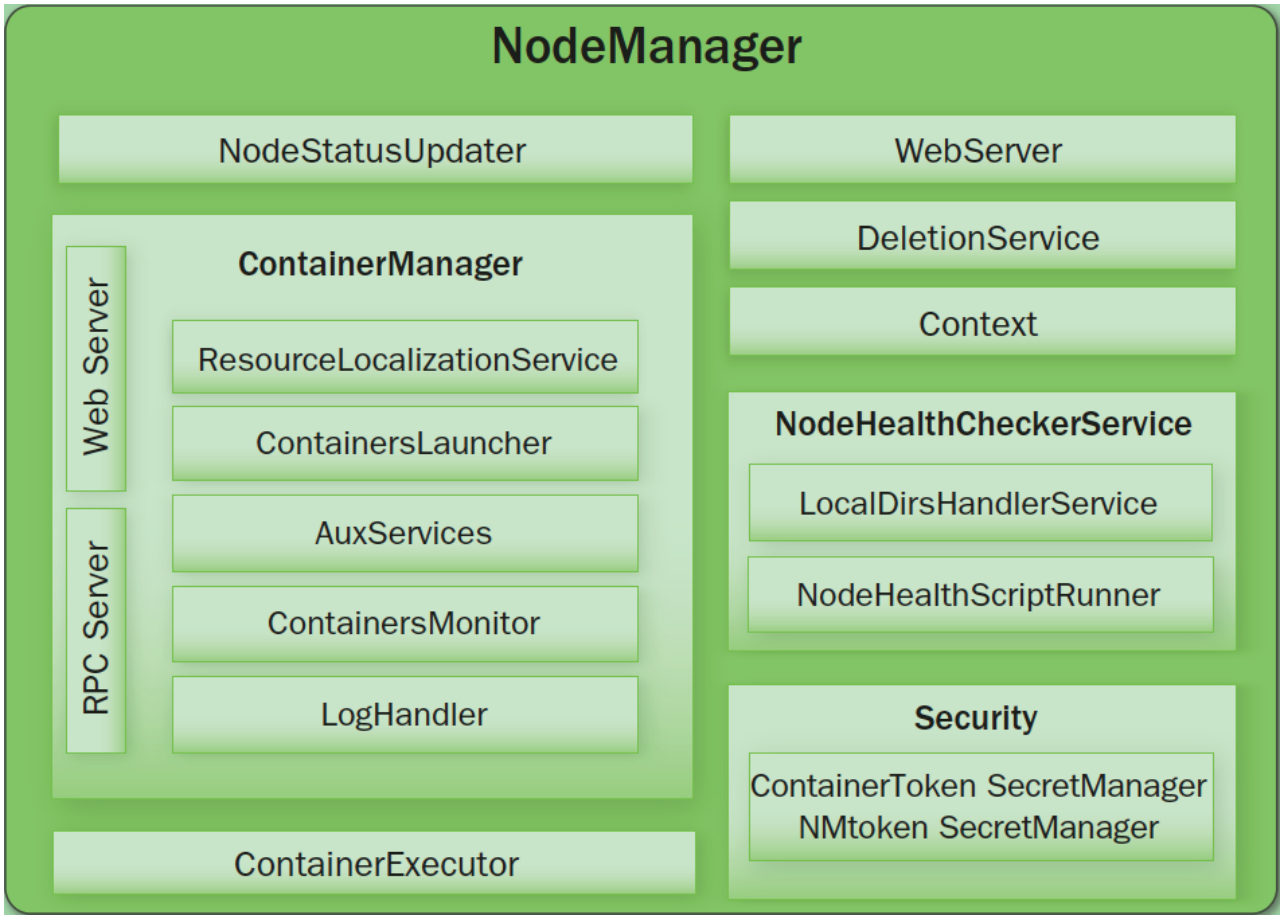
- ApplicationMaster Launcher: 一个应用提交后，负责与NodeManager交互，分配Container并加载ApplicationMaster，也负责终止或销毁
- YarnScheduler: 资源调度分配， 有FIFO(with Priority)，Fair，Capacity方式
- ContainerAllocationExpirer: 管理已分配但没有启用的Container，超过一定时间则将其回收

YARN – NodeManager

Node节点下的Container管理

1. 启动时向ResourceManager注册并定时发&##x9001;心跳消息，等待ResourceManager的指令
2. 监控Container的运行，维护Container的生命周期，监控Container的资源使用情况
3. 启动或停止Container，管理任务运行时的依赖包（根据ApplicationMaster的需要，启动Container之前将需要的程序及其依赖包、配置文件等拷贝到本地）

内部结构



- NodeStatusUpdater: 启动向ResourceManager注册，报告该节点的可用资源情况，通信的端口和后续状态的维护
- ContainerManager: 接收RPC请求（启动、停止），资源本地化（下载应用需要的资源到本地，根据需要共享这些资源）

PUBLIC: /filecache

PRIVATE: /usercache//filecache

APPLICATION: /usercache//appcache//（在程序完成后会被删除）

- ContainersLauncher: 加载或终止Container
- ContainerMonitor: 监控Container的运行和资源使用情况
- ContainerExecutor: 和底层操作系统交互，加载要运行的程序

YARN – ApplicationMaster

单个作业的资源管理和任务监控

具体功能描述#x8FF0;:

1. 计算应用的资源需求，资源可以是静态或动态计算的，静态的一般是Client申请时就指定了，动态则需要ApplicationMaster根据应用的运行状态来决定
2. 根据数据来申请对应位置的资源（Data Locality）
3. 向ResourceManager申请资源，与NodeManager交互进行程序的运行和监控，监控申请的资源的使用情况，监控作业进度
4. 跟踪任务状态和进度，定时向ResourceManager发送心跳消息，报告资源的使用情况和应用的进度信息
5. 负责本作业内的任务的容错

ApplicationMaster可以是用任何语言编写的程序，它和ResourceManager和NodeManager之间是通过ProtocolBuf交互，以前是一个全局的JobTracker负责的，现在每个作业都一个，可伸缩性更强，至少不会因为作业太多，造成JobTracker瓶颈。同时将作业的逻辑放到一个独立的ApplicationMaster中，使得灵活性更加高，每个作业都可以有自己的处理方式，不用绑定到MapReduce的处理模式上

如何计算资源需求

一般的MapReduce是根据block数量来定Map和Reduce的计算数量，然后一般的Map或Reduce就占用一个Container

如何发现数据的本地化

数据本地化是通过HDFS的block分片信息获取的

YARN – Container

1. 基本的资源单位（CPU、内存等）
2. Container可以加载任意程序，而且不限于Java
3. 一?#x4E2A;Node可以包含多个Container，也可以是一个大的Container
4. ApplicationMaster可以根据需要，动态申请和释放Container

YARN – Failover

失败类型

1. 程序问题
2. 进程崩溃
3. 硬&#x#x4EF6;问题

失败处理

任务失败

1. 运行时异常或者JVM退出都会报告给ApplicationMaster
2. 通过心跳来检查挂住的任务(timeout)，会检查多次（可配置）才判断该任务是否失效
3. 一个作业的任务失败率超过配置，则认为该作业失败
4. 失败的任务或作业都会有ApplicationMaster重新运行

ApplicationMaster失败

1. ApplicationMaster定时发送心跳信号到ResourceManager，通常一旦ApplicationMaster失败，则认为失败，但也可以通过配置多次后才失败
2. 一&##x65E6;ApplicationMaster失败，ResourceManager会启动一个新的ApplicationMaster
3. 新的ApplicationMaster负责恢复之前错误的ApplicationMaster的状态(yarn.app.mapreduce.am.job.recovery.enable=true)，这一步是通过将应用运行状态保存到共享的存储上来实现的，ResourceManager不会负责任务状态的保存和恢复
4. Client也会定时向ApplicationMaster查询进度和状态，一旦发现其失败，则向ResouceManager询问新的ApplicationMaster

NodeManager失败

1. NodeManager定时发送心跳到ResourceManager，如果超过一段时间没有收到心跳消息，ResourceManager就会将其移除

2. 任何运行在该NodeManager上的#x7684;任务和ApplicationMaster都会在其他NodeManager上进行恢复
3. 如果某个NodeManager失败的次数太多，ApplicationMaster会将其加入黑名单（ResourceManager没有），任务调度时不在其上运行任务

ResourceManager失败

1. 通过checkpoint机制，定时将其状态保存到磁盘，然后失败的时候，重新运行
2. 通过zookeeper同步状态和实现透明的HA

可以看出，一般的错误处理都是由当前模块的父模块进行监控（心跳）和恢复。而最顶端的模块则通过定时保存、同步状态和zookeeper来?#x5B9E;现HA

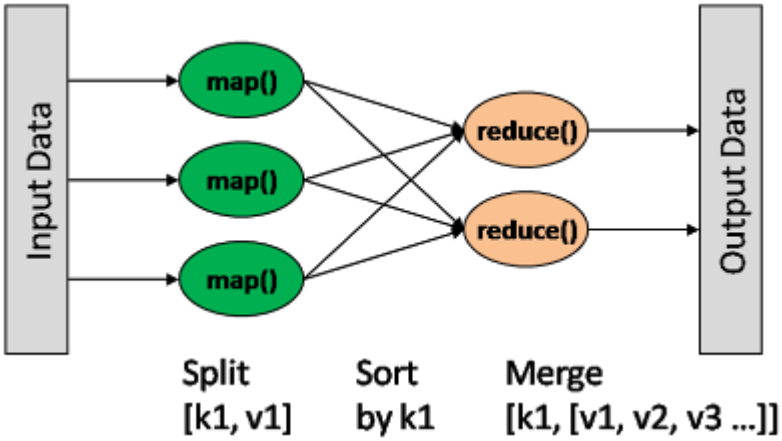
4

Hadoop – MapReduce

简介

一种分布式的计算方式指定一个Map（映?#x5C04;）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组

Pattern

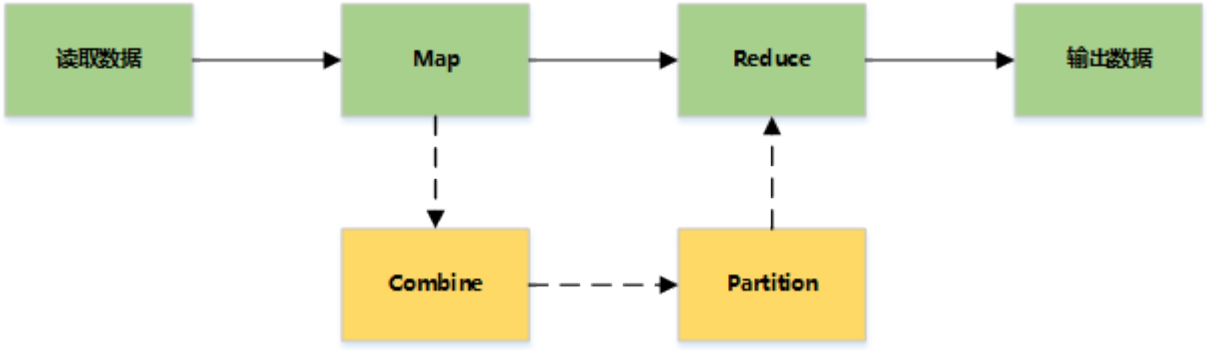


map: (K1, V1) → list(K2, V2) combine: (K2, list(V2)) → list(K2, V2) reduce: (K2, list(V2)) → list(K3, V3)

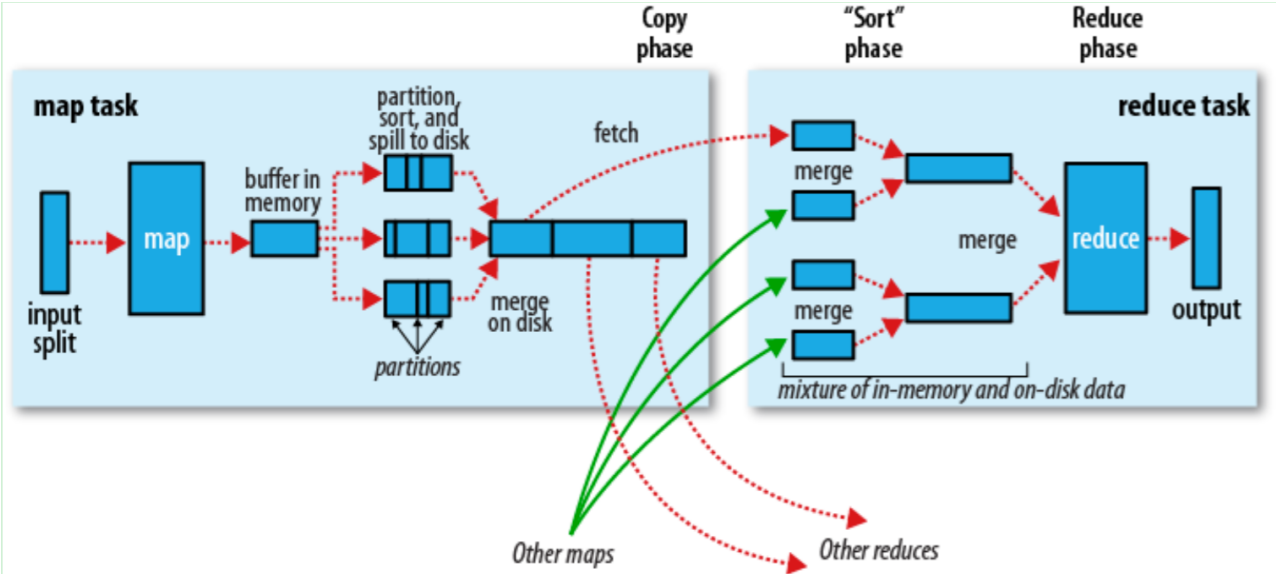
Map输出格式和Reduce输入格式一定是相同的

基本流程

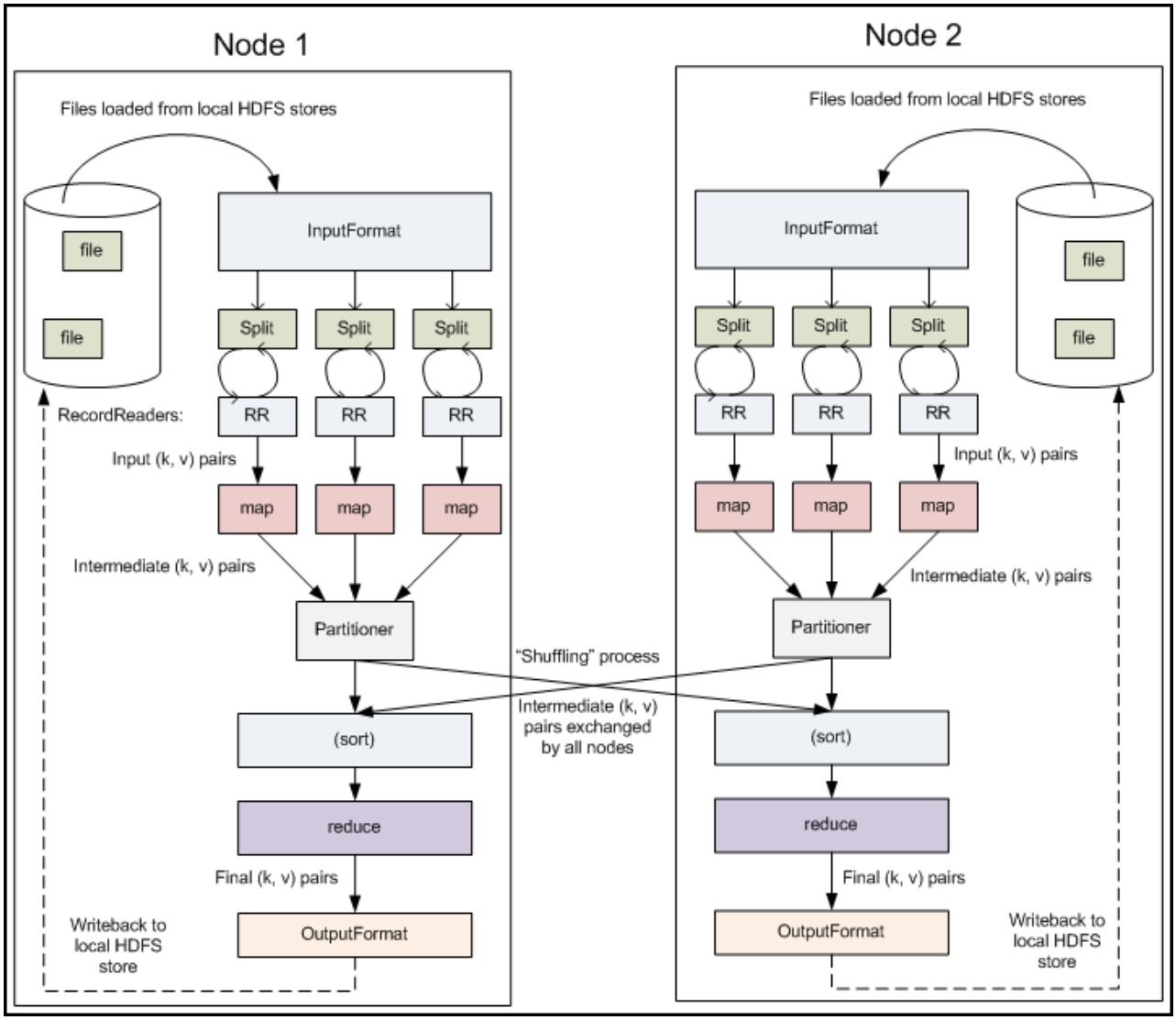
MapReduce主要是先读取文件数据，然后进行Map处理，接着Reduce处理，最后把处理结果写到文件中



详细流程


Copy phase
"Sort" phase
Reduce phase
map task
reduce task
partition, sort, and spill to disk
fetch
buffer in memory
map
input split
merge on disk
partitions
merge
merge
merge
reduce
output
mixture of in-memory and on-disk data
Other maps
Other reduces

多节点下的流程



主要过程

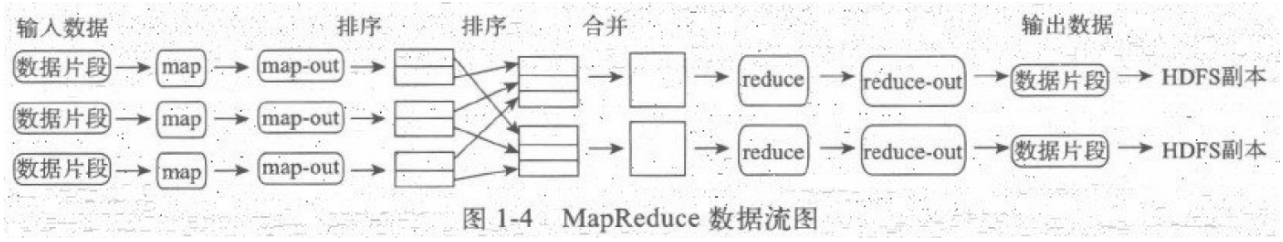


图 1-4 MapReduce 数据流图

Map Side

Record reader

记录阅读器会翻译由输入格式生成的记录，记录阅读器用于将数据解析给记录，并不分析记录自身。记录读取器的目的是将数据解析成记录，但不分析记录本身。它将数据以键值对的形式传输给mapper。通常键是位置信息，值是构成记录的数据存储块.自定义记录不在本文讨论范围之内.

Map

在映射器中用户提供的代码称为中间对。对于键值的具体定义是慎重的，因为定义对于分布式任务的完成具有重要意义.键决定了数据分类的依据，而值决定了处理器中的分析信息.本书的设计模式将会展示大量细节来解释特定键值如何选择.

Shuffle and Sort

ruduce任务以随机和排序步骤开始。此步骤写入输出文件并下载到本地计算机。这些数据采用键进行排序以把等价密钥组合到一起。

Reduce

reducer采用分组数据作为输入。该功能传递键和此键相关值的迭代器。可以采用多种方式来汇总、过滤或者合并数据。当ruduce功能完成，就会发送0个或多个键值对。

输出格式

输出格式会转换最终的键值对并写入文件。默认情况下键和值以tab分割，各记录以换行符分割。因此可以自定义更多输出格式，最终数据会写入HDFS。类似记录读取，自定义输出格式不在本书范围。

MapReduce – 读取数据

通过InputFormat决定读取的数据的类型，然后拆分成一个个InputSplit，每个InputSplit对应一个Map处理，RecordReader读取InputSplit的内容给Map

InputFormat

决定读取数据的格式，可以是文件或数据库等

功能

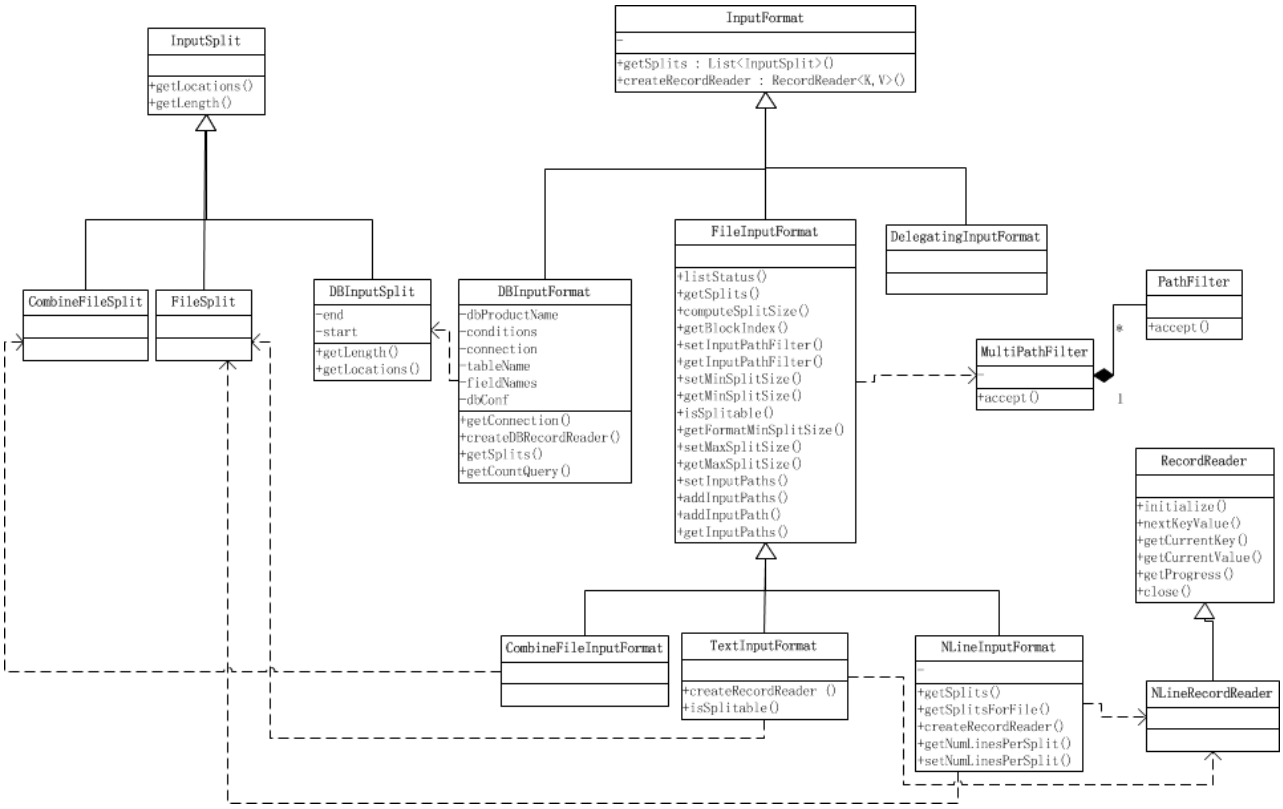
1. 验证作业输入的正确性，如格式等
2. 将输入文件切割成逻辑分片(InputSplit)，一个InputSplit将会被分配给一个独立的Map任务
3. 提供RecordReader实现，读取InputSplit中的"K-V对"供Mapper使用

方法

List getSplits(): 获取由输入文件计算出输入分片(InputSplit)，解决数据或文件分割成片问题

RecordReader <k,v>createRecordReader():</k,v> 创建#x5EFA;RecordReader，从InputSplit中读取数据，解决读取分片中数据问题

类结构



TextInputFormat: 输入文件中的每一行就是一个记录，Key是这一行的byte offset，而value是这一行的内容

KeyValueTextInputFormat: 输入文件中每一行就是一个记录，第一个分隔符字符切分每行。在分隔符字符之前的内容为Key，在之后的为Value。分隔符变量通过key.value.separator.in.input.line变量设置，默认为(\t)字符。

NLineInputFormat: 与TextInputFormat一样，但每个数据块必须保证有且只有N行，mapred.line.input.format.linespermap属性，默认为 1

SequenceFileInputFormat: 一个用来读取字符流数据的InputFormat，<key,value>为用户自定义的。字符流数据是Hadoop自定义的压缩的二进制数据格式。它用来优化从一个MapReduce任务的输出到另一个MapReduce任务的输入之间的数据传输过程。</key,value>

InputSplit

代表一个个逻辑分片，并没有真正存储数据，只是提供了一个如何将数据分片的方法

Split内有Location信息，利于数据局部化

一个InputSplit给一个单独的Map处理

```
public abstract class InputSplit {
    /**
     * 获取Split的大小，支持根据size对InputSplit排序.
     */
    public abstract long getLength() throws IOException, InterruptedException;

    /**
     * 获取存储该分片的数据所在的节点位置.
     */
    public abstract String[] getLocations() throws IOException, InterruptedException;
}
```

RecordReader

将InputSplit拆分成一个个<key,value>对给Map处理，也是实际的文件读取分隔对象</key,value>

问题

大量小文件如何处理

CombineFileInputFormat可以将若干个Split打包成一个，目的是避免过多的Map任务（因为Split的数目决定了Map的数目，大量的Mapper Task创建销毁开销将是巨大的）

怎么计算split的

通常一个split就是一个block（FileInputFormat仅仅拆分比block大的文件），这样做的好处是使得Map可以在存储有当前数据的节点上运行本地的任务，而不需要通过网络进行跨节点的任务调度

通过mapred.min.split.size， mapred.max.split.size, block.size来控制拆分的大小

如果mapred.min.split.size大于block size，则会将两个block合成到一个split，这样有部分block数据需要通过网络读取

如果mapred.max.split.size小于block size，则会将一个block拆成多个split，增加了Map任务数（Map对split进行计算?#x5E76;且上报结果，关闭当前计算打开新的split均需要耗费资源）

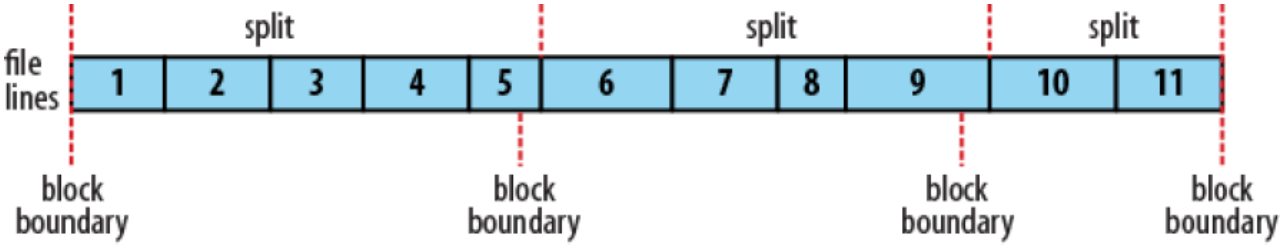
先获取文件在HDFS上的路径和Block信息，然后根据splitSize对文件进行切分（ splitSize = computeSplitSize(blockSize, minSize, maxSize)），默认splitSize 就等于blockSize的默认值（64m）

```
public List<InputSplit> getSplits(JobContext job) throws IOException {
  // 首先计算分片的最大和最小值。这两个值将会用来计算分片的大小
  long minSize = Math.max(getFormatMinSplitSize(), getMinSplitSize(job));
  long maxSize = getMaxSplitSize(job);

  // generate splits
  List<InputSplit> splits = new ArrayList<InputSplit>();
  List<FileStatus> files = listStatus(job);
  for (FileStatus file: files) {
    Path path = file.getPath();
    long length = file.getLen();
    if (length != 0) {
      FileSystem fs = path.getFileSystem(job.getConfiguration());
      // 获取该文件所有的block信息列表[hostname, offset, length]
      BlockLocation[] blkLocations = fs.getFileBlockLocations(file, 0, length);
      // 判断文件是否可分割，通常是可分割的，但如果文件是压缩的，将不可分割
      if (isSplitable(job, path)) {
        long blockSize = file.getBlockSize();
        // 计算分片大小
        // 即 Math.max(minSize, Math.min(maxSize, blockSize));
        long splitSize = computeSplitSize(blockSize, minSize, maxSize);

        long bytesRemaining = length;
        // 循环分片。
        // 当剩余数据与分片大小比值大于Split_Slop时，继续分片， 小于等于时，停止分片
        while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
          int blkIndex = getBlockIndex(blkLocations, length-bytesRemaining);
          splits.add(makeSplit(path, length-bytesRemaining, splitSize, blkLocations[blkIndex].getHosts()));
          bytesRemaining -= splitSize;
        }
        // 处理余下的数据
        if (bytesRemaining != 0) {
          splits.add(makeSplit(path, length-bytesRemaining, bytesRemaining, blkLocations[blkLocations.length-1].getH
        }
      } else {
        // 不可split，整块返回
        splits.add(makeSplit(path, 0, length, blkLocations[0].getHosts()));
      }
    } else {
      // 对于长度为0的文件，创建空Hosts列表，返回
      splits.add(makeSplit(path, 0, length, new String[0]));
    }
```

```
    }

    // 设置输入文件数量
    job.getConfiguration().setLong(NUM_INPUT_FILES, files.size());
    LOG.debug("Total # of splits: " + splits.size());
    return splits;
}
```

分片间的数据如何处理

split是根据文件大小分割的，而一般处理是根据分隔符进行分割的，这样势必存在一条记录横跨两个split



解决办法是只要不是第一个split，都会远程读取一条记录。不是第一个split的都忽略到第一条记录

```
public class LineRecordReader extends RecordReader<LongWritable, Text> {
    private CompressionCodecFactory compressionCodecs = null;
    private long start;
    private long pos;
    private long end;
    private LineReader in;
    private int maxLineLength;
    private LongWritable key = null;
    private Text value = null;

    // initialize函数即对LineRecordReader的一个初始化
    // 主要是计算分片的始末位置，打开输入流以供读取K-V对，处理分片经过压缩的情况等
    public void initialize(InputSplit genericSplit, TaskAttemptContext context) throws IOException {
        FileSplit split = (FileSplit) genericSplit;
        Configuration job = context.getConfiguration();
        this.maxLineLength = job.getInt("mapred.linerecordreader.maxlength", Integer.MAX_VALUE);
        start = split.getStart();
        end = start + split.getLength();
        final Path file = split.getPath();
        compressionCodecs = new CompressionCodecFactory(job);
        final CompressionCodec codec = compressionCodecs.getCodec(file);

        // 打开文件，并定位到分片读取的起始位置
        FileSystem fs = file.getFileSystem(job);
```

```
    FSDataInputStream fileIn = fs.open(split.getPath());

    boolean skipFirstLine = false;
    if (codec != null) {
        // 文件是压缩文件的话，直接打开文件
        in = new LineReader(codec.createInputStream(fileIn), job);
        end = Long.MAX_VALUE;
    } else {
        // 只要不是第一个split，则忽略本split的第一行数据
        if (start != 0) {
            skipFirstLine = true;
            --start;
            // 定位到偏移位置，下&#x#x6B21;读取就会从偏移位置开始
            fileIn.seek(start);
        }
        in = new LineReader(fileIn, job);
    }

    if (skipFirstLine) {
        // 忽略第一行数据，重新定位start
        start += in.readLine(new Text(), 0, (int) Math.min((long) Integer.MAX_VALUE, end - start));
    }
    this.pos = start;
}

public boolean nextKeyValue() throws IOException {
    if (key == null) {
        key = new LongWritable();
    }
    key.set(pos);// key即为偏移量
    if (value == null) {
        value = new Text();
    }
    int newSize = 0;
    while (pos < end) {
        newSize = in.readLine(value, maxLineLength,  Math.max((int) Math.min(Integer.MAX_VALUE, end - pos), maxLi
        // 读取的数据长度为0，则说明已读完
        if (newSize == 0) {
            break;
        }
        pos += newSize;
        // 读取的数据长度小于最大行长度，也说明已读取完毕
        if (newSize < maxLineLength) {
            break;
        }
```

```
            // 执行到此处，说明该行数据没读完，继续读入
        }
        if (newSize == 0) {
            key = null;
            value = null;
            return false;
        } else {
            return true;
        }
    }
}
```

MapReduce – Mapper

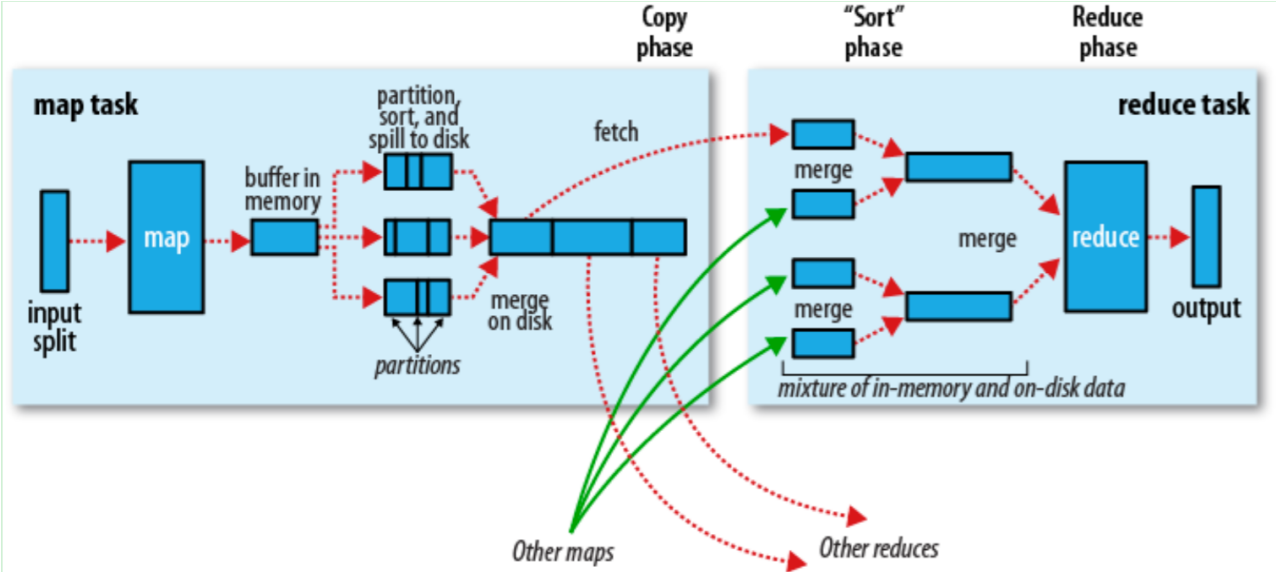
主要是读取InputSplit的每一个Key,Value对并进行处理

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    /**
     * 预处理，仅在map task启动时运行一次
     */
    protected void setup(Context context) throws  IOException, InterruptedException {
    }

    /**
     * 对于InputSplit中的每一对<key, value>都会运行一次
     */
    @SuppressWarnings("unchecked")
    protected void map(KEYIN key, VALUEIN value, Context context) throws IOException, InterruptedException {
        context.write((KEYOUT) key, (VALUEOUT) value);
    }

    /**
     * 扫尾工作，比如关闭流等
     */
    protected void cleanup(Context context) throws IOException, InterruptedException {
    }

    /**
     * map task的驱动器
     */
    public void run(Context context) throws IOException, InterruptedException {
        setup(context);
        while (context.nextKeyValue()) {
            map(context.getCurrentKey(), context.getCurrentValue(), context);
        }
        cleanup(context);
    }
}

public class MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> extends TaskInputOutputContext<KEYIN, VALUEIN
    private RecordReader<KEYIN, VALUEIN> reader;
    private InputSplit split;

    /**
     * Get the input split for this map.
```

```
     */
    public InputSplit getInputSplit() {
        return split;
    }

    @Override
    public KEYIN getCurrentKey() throws IOException, InterruptedException {
        return reader.getCurrentKey();
    }

    @Override
    public VALUEIN getCurrentValue() throws IOException, InterruptedException {
        return reader.getCurrentValue();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        return reader.nextKeyValue();
    }
}
```

MapReduce – Shuffle

对Map的结果进行排序并传输到Reduce进行处理 Map的结果并不?#x662F;直接存放到硬盘,而是利用缓存做一些预排序处理 Map会调用Combiner，压缩，按key进行分区、排序等，尽量减少结果的大小 每个Map完成后都会通知Task，然后Reduce就可以进行处理



Map端

当Map程序开始产生结果的时候，并不是直接写到文件的，而是利用缓存做一些排序方面的预处理操作

每个Map任务都有一个循环内存缓冲区（默认100MB），当缓存的内容达到80%时，后台线程开始将内容写到文件，此时Map任务可以&#x#x7EE7;续输出结果，但如果缓冲区满了，Map任务则需要等待

写文件使用round-robin方式。在写入文件之前，先将数据按照Reduce进行分区。对于每一个分区，都会在内存中根据key进行排序，如果配置了Combiner，则排序后执行Combiner（Combine之后可以减少写入文件和传输的数据）

每次结果达到缓冲区的阀值时，都会创建一个文件，在Map结束时，可能会产生大量的文件。在Map完成前，会将这些文件进行合并和排序。如果文件的数量超过3个，则&##x5408;并后会再次运行Combiner（1、2个文件就没有必要了）

如果配置了压缩，则最终写入的文件会先进行压缩，这样可以减少写入和传输的数据

一旦Map完成，则通知任务管理器，此时Reduce就可以开始复制结果数据

Reduce端

Map的结果文件都存放到运行Map任务的机器的本地硬盘中

如果Map的结果很少，则直接放到内存，否则写入文件中

同时后台线程将这些文件进行合并和排序到一个更大的文件中（如果文件是压缩的ÿ#xFF0C;则需要先解压）

当所有的Map结果都被复制和合并后，就会调用Reduce方法

Reduce结果会写入到HDFS中

调优

一般的原则是给shuffle分配尽可能多的内存，但前提是要保证Map、Reduce任务有足够的内存

对于Map，主要就是避免把文件写入磁盘，例如使用Combiner，增大io.sort.mb的值

对于Reduce，主要是把Map的结果尽可能地保存到内存中，同样也是要避免把中间结果写入磁盘。默认情况下，所有的内存都是分配给Reduce方法的，如果Reduce方法不怎&##x4E48;消耗内存，可以mapred.inmem.merge.threshold设成0，mapred.job.reduce.input.buffer.percent设成1.0

在任务监控中可通过Spilled records counter来监控写入磁盘的数，但这个值是包括map和reduce的

对于IO方面，可以Map的结果可以使用压缩，同时增大buffer size（io.file.buffer.size，默认4kb）

配置

<

table>

属性

默认值

描述

io.sort.mb

100

映射输出分类时所使用缓冲区的大小.

io.sort.record.percent

0.05

剩余空间用于映射输出自身记录.在1.X发布后去除此属性.随机代码用于使用映射所有内存并记录信息.

io.sort.spill.percent

0.80

针对映射输出内存缓冲和记录索引的阈值使用比例.

io.sort.factor

10

文件分类时合并流的最大数量。此属性也用于reduce。通常把数字设为100.

min.num.spills.for.combine

3

组合运行所需最小溢出文件数目.

mapred.compress.map.output

false

压缩映射输出.

mapred.map.output.compression.codec

DefaultCodec

映射输出所需的压缩解编码器.

mapred.reduce.parallel.copies

5

用于向reducer传送映射输出的线程数目.

mapred.reduce.copy.backoff

300

时间的最大数量，以秒为单位，这段时间内若reducer失败则会反复尝试传输

io.sort.factor

10

组合运行所需最大溢出文件数目.

mapred.job.shuffle.input.buffer.percent

0.70

随机复制阶段映射输出缓冲器的堆栈大小比例

mapred.job.shuffle.merge.percent

0.66

用于启动合并输出进程和磁盘传输的映射输出缓冲器的阀值使用比例

mapred.inmem.merge.threshold

1000

用于启动合并输出和磁盘传输进程的映射输出的阀值数目。小于等于0意味着没有门槛，而溢出行为由 mapred.job.shuffle.merge.percent单独管理.

mapred.job.reduce.input.buffer.percent

0.0

用于减少内存映射输出的堆栈大小比例，内存中映射大小不得超出此值。若reducer需要较少内存则可以提高该值.

MapReduce – 编程

在线练习

http://cloudcomputing.ruc.edu.cn

处理

1. select：直接分析输入数据，取出需要的字段数据即可
2. where: 也是对输入数据处理的过程中进行处理，判断是否需要该数据
3. aggregation:min, max, sum
4. group by: 通过Reducer实现
5. sort
6. join: map join, reduce join

Third-Party Libraries

export LIBJARS=$MYLIB/commons-lang-2.3.jar, hadoop jar prohadoop-0.0.1-SNAPSHOT.jar org.aspress.prohadoop.c3. WordCountUsingToolRunner -libjars $LIBJARS

hadoop jar prohadoop-0.0.1-SNAPSHOT-jar-with-dependencies.jar org.aspress.prohadoop.c3. WordCountUsingToolRunner The dependent libraries are now included inside the application JAR file

一般还是上面的好，指定依赖可以利用Public Cache，如果是包含依赖，则每次都需要拷贝

参考书籍

MapReduce Design Patterns (http://book.douban.com/subject/11229683/)

5

Hadoop – IO

1. 输入文件从HDFS进行读取.
2. 输出文件会存入本地磁盘.
3. Reducer和Mapper间的网络I/O,从Mapper节点得到Reducer的检索文件.
4. 使用Reducer实例从本地磁盘回读数据.
5. Reducer输出– 回传到HDFS.

串行化

传输、存储都需要

Writable接口

Avro框架：IDL，版本支持，跨语言，JSON-linke

压缩

能够减少磁盘的占用空间和网络传输的量

Compressed Size, Speed, Splittable

gzip, bzip2, LZO, LZ4, Snappy

要比较各种压缩算法的压缩比和性能

重点：压缩和拆分一般是冲突的（压缩后的文件的block是不能很好地拆分独立运行，很多时候某个文件的拆分点是被&#x#x62C6;分到两个压缩文件中，这时Map任务就无法处理，所以对于这些压缩，Hadoop往往是直接使用一个Map任务处理整个文件的分析）

Map的输出结果也可以进行压缩，这样可以减少Map结果到Reduce的传输的数据量，加快传输速率

完整性

磁盘和网络很容易出错，保证数据传输的完整性一般是通过CRC32这种校验法

每次写数据到磁盘前都验证一下，同时保存校验码

每次读取数据时，也验证校验码，避免磁盘问题

同时#x65F6;每个datanode都会定时检查每一个block的完整性

当发现某个block数据有问题时，也不是立刻报错，而是先去Namenode找一块该数据的完整备份进行恢复，不能恢复才报错

6 Hadoop 测试

MRUnit单元测试Mapper和Reducer类在内存上独立运行, PipelineMapReduceDriver单线程运行.

LocalJobRunner单线程运行, 且仅有一个 Reducer能够启动conf.set("mapred.job.tracker", "local"); conf.set("fs.default.name", "file:////&ququot;); FileSystem fs = FileSystem.getLocal(conf);

MiniMRCluster, MiniYarnCluster, MiniDFSCluster 多线程

7

Hadoop安装

- **单节点安装**

  所有服务运行在一个JVM中，适合调试、单元测试

- **伪集群**

  所有服务运行在一台机器中，每个服务都在独立的JVM中，适合做简单、抽样测试

- **多节点集群**

  服务运行在不同的机器中，适合生产环境

  配置公共帐号

方便主与从进行无密钥通信，主要是使用公钥/私钥机制 所有节点的帐号都一样 在主节点上执行 ssh-keygen -t rsa生成密钥对 复制公钥到每台目标节点中

8

Hadoop配置

有两种配置文件：

一种是__-default.xml（只读，默认的配置）

一种是#x662F;__-site.xml（替换default中的配置）

- core-site.xml 配置公共属性
- hdfs-site.xml 配置HDFS
- yarn-site.xml 配置YARN
- mapred-site.xml 配置MapReduce

配置文件应用的顺序：

1. 在JobConf中指定的
2. 客户端机器上的__-site.xml配置
3. slave节点上的__-site.xml配置
4. __-default.xml中的配置

如果某个属性不想被覆盖，可以将其设置成final

```
<property>
  <name>{PROPERTY_NAME}</name>
  <value>{PROPERTY_VALUE}</value>
  <final>true</final>
</property>
```

9

Hadoop 监控

Log yarn.log-aggregation-enable=true如果显示错误，则日志存储在节点管理器运行节点上。当聚集启用时所有日志进行汇总，任务完成后转移到HDFS。

Hadoop集群性能监控Ganglia, Nagios

使用Hadoop工具 Ambari管理集群

10

Hadoop – 参考

- Yahoo教程 (https://developer.yahoo.com/hadoop/tutorial/index.html)
- 细细品味Hadoop (http://www.cnblogs.com/xia520pi/category/346943.html)
- HDFS 原理、架构与特性介绍 (http://www.open-open.com/lib/view/open1376228205209.html)
- Hadoop MapReduce开发最佳实践 (http://www.infoq.com/cn/articles/MapReduce-Best-Practice-1)
- MapReduce的Shuffle阶段 (http://blog.sina.com.cn/s/blog_605f5b4f010188lp.html)
- Hadoop的最佳实践和反模式 (http://www.oschina.net/translate/apache-hadoop-best-practices-and-anti-patterns)
- Hadoop读split (http://stackoverflow.com/questions/14291170/how-does-hadoop-process-records-records-split-across-block-boundaries)