一、Hadoop 理论知识---Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构。

1）Hadoop 的三大核心组件

HADOOP 集群具体来说包含两个集群：HDFS 集群和 YARN 集群，两者逻辑上分离，但物理上常在一起

HDFS 集群：负责海量数据的存储，集群中的角色主要有 NameNode / DataNode

YARN 集群：负责海量数据运算时的资源调度，集群中的角色主要有 ResourceManager /NodeManager

那 mapreduce 是什么呢？它其实是一个应用程序开发包。

1、HDFS

HDFS 是一个高度容错性的系统，适合部署在廉价的机器上。HDFS 采用 master/slave 架构。一个 HDFS 集群是由一个 Namenode 和一定数目的 Datanodes 组成。Namenode 是一个中心服务器，负责管理文件系统的名字空间(namespace)以及客户端对文件的访问。集群中的 Datanode 一般是一个节点一个，负责管理它所在节点上的存储。

A、NameNode---用于保存元数据(MetaData)信息(NameNode 保存在内存中)---元数据(FileName、副本数、每一个副本所在的位置...)

a、fsimage--对元数据定期进行镜像

b、edits--存放一定时间内对 HDFS 的操作记录

c、checkpoint---检查点

Namenode 在内存中保存着整个文件系统的名字空间和文件数据块映射(Blockmap)的映像。这个关键的元数据结构设计得很紧凑，因而一个有 4G 内存的 Namenode 足够支撑大量的文件和目录。当 Namenode 启动时，它从硬盘中读取 Editlog 和 FsImage，将所有 Editlog 中的事务作用在内存中的 FsImage 上，并将这个新版本的 FsImage 从内存中保存到本地磁盘上,然后删除旧的 Editlog,因为这个旧的 Editlog 的事务都已经作用在 FsImage 上了。这个过程称为一个检查点(checkpoint)。在当前实现中，检查点只发生在 Namenode 启动时，在不久的将来将实现支持周期性的检查点。

B、DataNode---存储节点，真正存放数据的节点，用于保存数据，保存在磁盘上（在 HDFS 上保存的数据副本数默认是 3 个，这个副本数量是可以设置的）。基本单位是 block，默认 128M。

★名词扩展：心跳机制、宕机、安全模式

Datanode 负责处理文件系统客户端的读写请求。在 Namenode 的统一调度下进行数据块的创建、删除和复制。集群中单一 Namenode 的结构大大简化了系统的架构。Namenode 是所有 HDFS 元数据的仲裁者和管理者，这样，用户数据永远不会流过 Namenode。

★理解读过程和写过程！！！

C、ScendaryNameNode---辅助节点，用于同步元数据信息。辅助 NameNode 对 fsimage 和 edits 进行合并（冷备份）

NameNode 的元数据信息先往 edits 文件中写,当 edits 文件达到一定的阈值(3600 秒或大小到 64M) 的时候,会开启合并的流程。合并流程如下:

①当开始合并的时候,SecondaryNameNode 会把 edits 和 fsimage 拷贝到自己服务器所在内存中,开始合并,合并生成一个名为 fsimage.ckpt 的文件。

②将 fsimage.ckpt 文件拷贝到 NameNode 上,成功后,再删除原有的 fsimage,并将 fsimage.ckpt 文件重命名为 fsimage。

③当 SecondaryNameNode 将 edits 和 fsimage 拷贝走之后,NameNode 会立刻生成一个 edits.new 文件,用于记录新来的元数据,当合并完成之后,原有的 edits 文件才会被删除,并将 edits.new 文件重命名为 edits 文件,开启下一轮流程。

2、MapReduce 分布式计算框架--hadoop 的一个程序，不会产生进程

-A、自定义序列化类

有时候，默认的数据类型不能满足我们的需求时，需要我们自定义序列化类，实现 WritableComparable。在自定义的序列化类中，最重要的是重写 compareTo 方法以及序列化反序列化方法。序列化和反序列化的内容需要重点关注，容易犯低级错误。

★二次排序：compareTo 方法也可以实现二次排序的功能，但会产生大量的序列化反序列化实例，浪费资源；比较优化的方法是在自定义序列化类中的一个静态内部类--Comparator，继承 WritableComparator，在这个类中的 compare 方法中

写排序的逻辑。需要对这个内部类进行注册。

A、Mapper---Mapper 对来的每一条数据进行一次计算（这里的计算指的时代码逻辑，这句话的意思就是每来一条数据走一次 map 方法）

自定义 Mapper，需要继承 Mapper，并指定泛型<key-in,value-in,key-out,value-out>,然后重写 map 方法。泛型中的 key-in，value-in 是读取文件时读的内容，默认 k 按偏移量操作，v 读一行内容，可以通过自定义输入格式来改变 k-in 和 v-in;key-out 和 value-out 是写入环形缓存区的内容,如果有 Reducer 的话这里的 k-out 和 v-out 最终是 Reducer 的 key-in 和 value-in。

◆shuffle 阶段

MapReduce 的核心与基础是 Mapper 类、Reducer 类与 Driver。Driver 中主要是 main()方法，MR 的程序入口；Driver 中还要规定 job 的各种配置。自己的 Mapper 需要继承 Mapper 类,重写其中的 map()方法,自己的 Reducer 需要继承 Reducer 类，重写其中的 reduce()方法。map 的运行机制是来一条数据运行一次 map 方法，reduce 的运行机制是来一个 key 运行一次 reduce 方法。数据从 map 中出来到进入 reduce 之前称为 shuffle 阶段，Mapper 的数量不建议人为设定，一般一个 block 对应一个 Mapper，而 Reducer 的数量可以在 Driver 中人为控制，不设定默认是 1。

MR 过程中的数据流向：一个文件在 HDFS 中是分布存储在不同节点的 block 中，每一个 block 对应一个 Mapper，每一条数据以 K,V 的形式进入一个 map()方法，map()方法中对数据进行处理，再将处理结果以 K,V 的形式写入环形缓冲区，一个 Mapper 对应一个 context，context 对写入的数据按 key 进行聚合、排序、归约。context 的大小默认为 100M，当 context 容量达到 80%或 Mapper 处理结束时，context 会向外溢出，形成许多小文件，小文件为一个 K 和许多 V 的集合。处理完成后，这些文件会发送到 Reducer 所在节点，在该节点的 context 中，会对不同节点发送过来的数据按 key 进行再一次的聚合、排序和归约，最后进入 Reducer。

B、Reducer---Reducer 对相同的 Key 进行一次计算

自定义 Reducer，需要继承 Reducer，并指定泛型<key-in,value-in,key-out,value-out>,然后重写 reduce 方法。泛型中的 k-in 和 v-in 必须和 Mapper 的 k-out 和 v-out 的数据类型一致；可以通过自定义输出格式来改变 k-out 和 v-out。

C、Driver---main 方法，程序的入口

在 main 方法中需要获取一个配置实例，得到一个 job 实例，用这个 job 指定主类、Mapper 类、Reducer 类、Combiner 类（如果有）、Partitioner 类（如果有），指定 Mapper 和 Reducer 的输出格式类（如果 Mapper 和 Reducer 的输出类型相同，可以只设置 outputKey 和 outputValue；如果不同则需要设置 outputKey 和 outputValue、MapoutputKey 和 MapoutputValue），通过默认输入格式或自定义输入格式指定输入文件路径，指定输出目录，判断 job 结束并关闭程序。

D、Partitioner---分区类

自定义分区类需要继承 Partitioner，指定泛型<k,v>，重写 getPartition 方法，它可以实现将不同的 Key 写入不同的文件。如果自定义了分区类，那么需要在 Driver 中指定分区类并且设置 ReducTask 数量（通过 setNumReduceTasks 方法）。

注意：每来一条数据走一次 getPartition 方法；有几个 ReduceTask 就会生成几个文件；1 个 task 任务不要处理大于 10G 的内容；Partitioner 的泛型要和 Mapper 的 k-out、v-out 一致。

E、Combiner---本地的 reducer，只能起到过渡和优化的作用，它能做一些像归约类的对输出结果不造成影响的任务，比如求和

Combiner 是一个本地 reducer，所以它仍然继承 Reducer，指定泛型<key-in,value-in,key-out,value-out>，它的 key-in,value-in,key-out,value-out 必须要和 mapper 的 k-out，v-out 一致，也和 reducer 的 k-in，v-in 一样（因为它只做简单优化，不能影响输出结果）。因为 combiner 只对 reducer 进行优化，所以它的逻辑可以跟 reducer 完全相同，也可以不一样，但是不能影响输出结果。而且，在数据量小的时候使用 combiner 与否几乎没什么差别。至于使用 combiner 的原因，是 Reducer 是在运行在网络环境上的，当数据量太大时，网络 I/O 速度慢，会导致效率低下。用本地的 Reducer 过渡，预处理可以提高效率。

F、自定义输入格式--InputFormat

自定义输入格式，需要继承 FileInputFormat。里面主要是重写 createRecordReader 方法，返回一个自定义的 RecordReader。实际上还有一个重写方法是 isSplitable，但是我们一般不作重写；因为 hadoop 不适合管理小文件，所以我们需要这个方法的返回值一直时 true，而在 InputFormat 的源码中，方法是这样的：protected boolean

isSplitable(JobContext context, Path filename) {return true;}，所以我们不需要重写这个方法。

自定义 RecordReader，它是主要的处理输入数据格式的类，最终是写初始化方法 initialize 和 nextKeyValue

在自定义 RecordReader 中，需要重写的方法有：

①initialize()---初始化方法，完成自定义字段的初始化（以实现一次读取两行为例解释）

思路：因为 LineReader 可以完成一行一行读的目的，所以初始化时，要做的事情就是得到一个 LineReader 实例；通过查看 API 发现，想实例化一个 LineReader 最低要求是得到一个输入流，所以首先需要得到一个输入流；输入流可以通过文件对象 open（Path）方法获得，所以我们的目的变成了获取一个文件对象和一个 Path；文件对象可以通过 get（Configuration）获取，Path 可以通过 FileSplit 的 getPath（）获得；Configuration 可以通过 context 得到，FileSplit 可以通过 split 得到---context 是一直贯穿于整个过程中的，split 时 initialize 的参数。所以，将上面的思路倒序实现，就完成了初始化过程，得到一个 LineReader 的实例。

②nextKeyValue()---在这个方法中写逻辑，对 K 和 V 进行赋值

要实现一次读两行，而在初始化时得到的 LineReader 可以一次读一行，所以只需要读两次，然后赋值给 Value 就可以实现。

③getCurrentKey()---获取当前 key

④getCurrentValue()---获取当前 value

⑤getProgress()---获取进度，一般 return true?0.0f:1.0f;

⑥close()---如果开了流必须要关闭，如果没开流则不需要

G、自定义输出格式--OutputFormat

自定义输出格式,需要继承 FileOutputFormat,主要重写一个 getRecordWriter 方法,返回一个自定义的 RecordWriter。自定义的 RecordWriter 中重写 write 方法和 close。可以实现自定义输出文件名，也可以写逻辑改变输出内容。

3、YARN

A、ResourceManager

B、NodeManager

MapReduce 在 YARN 上的执行流程：

①client 提交 job,首先找 ResourceManager(ApplicationsManager)分配资源,同时将 jar 包默认拷贝 10 份到 hdfs。

②ResourceManager 指定一个 NodeManager 开启一个 container，在 Container 中运行一个 ApplicationMaster 来管理这个应用程序。

③ApplicationMaster 会计算此个应用所需资源,向 ResourceManager(ResourceScheduler)申请资源。

④ResourceManager 会分配资源,在 NodeManager 上开启不同的 container,在 container 中来运行 map 任务或者 reduce 任务

⑤当所有的 task 都执行完了,ApplicationMaster 会将结果反馈给客户端,所有工作执行完成之后,ApplicationMaster 就会自行关闭。

⑥如果某个 map 任务或者 reduce 任务失败,ApplicationMaster 会重新申请新的 container 来执行这个 task。

二、Hadoop 集群搭建

1）集群规划

A、　　分布式系统一般都是 master/slaver 结构，规划时需要遵循主从分离的原则。

2）Hadoop 安装及配置。关于 hadoop 单节点安装、伪分布式配置以及 hadoop 集群配置在另外的文档中详述。

三、Hadoop 优化

1、常见问题

A、单点故障

B、小文件问题

C、数据处理性能

2、优化思路

A、从应用程序角度进行优化。由于 mapreduce 是迭代逐行解析数据文件的，怎样在迭代的情况下，编写高效率的应用程序，是一种优化思路。

a、避免不必要的 reduce 任务

如果要处理的数据是排序且已经分区的，或者对于一份数据，需要多次处理，可以先排序分区；然后自定义 InputSplit，将单个分区作为单个 mapred 的输入；在 map 中处理数据，Reducer 设置为空。

这样，既重用了已有的 “排序”，也避免了多余的 reduce 任务。

b、外部文件引入

有些应用程序要使用外部文件，如字典，配置文件等，这些文件需要在所有 task 之间共享，可以放到分布式缓存 DistributedCache 中（或直接采用-files 选项，机制相同）。

更多的这方面的优化方法，还需要在实践中不断积累。

c、为 job 添加一个 Combiner

为 job 添加一个 combiner 可以大大减少 shuffle 阶段从 map task 拷贝给远程 reduce task 的数据量。一般而言，combiner 与 reducer 相同。

d、根据处理数据特征使用最适合和简洁的 Writable 类型

Text 对象使用起来很方便，但它在由数值转换到文本或是由 UTF8 字符串转换到文本时都是低效的，且会消耗大量的 CPU 时间。当处理那些非文本的数据时，可以使用二进制的 Writable 类型，如 IntWritable， FloatWritable 等。二进制 writable 好处：避免文件转换的消耗；使 map task 中间结果占用更少的空间。

e、重用 Writable 类型---对象抽取

f、使用 StringBuffer 而不是 String

当需要对字符串进行操作时，使用 StringBuffer 而不是 String，String 是 read-only 的，如果对它进行修改，会产生临时对象，而 StringBuffer 是可修改的，不会产生临时对象。

B、对 Hadoop 参数进行调优。当前 hadoop 系统有 190 多个配置参数，怎样调整这些参数，使 hadoop 作业运行尽可能的快，也是一种优化思路。

C、从系统实现角度进行优化。这种优化难度是最大的，它是从 hadoop 实现机制角度，发现当前 Hadoop 设计和实现上的缺点，然后进行源码级地修改。该方法虽难度大，但往往效果明显。

以上三种思路出发点均是提高 hadoop 应用程序的效率。实际上，随着社会的发展，绿色环保观念也越来越多地融入了企业，因而很多人开始研究 Green Hadoop，即怎样让 Hadoop 完成相应数据处理任务的同时，使用最少的能源。