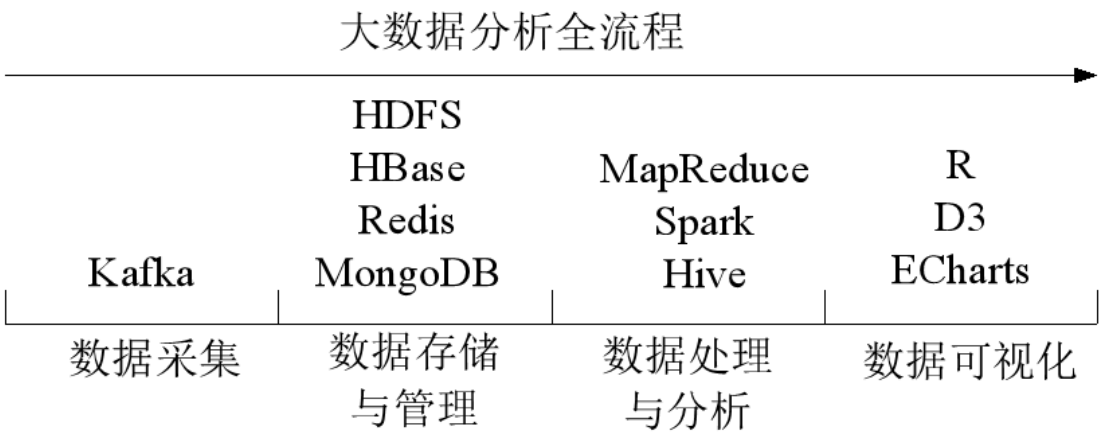
知识梳理

第一章　大数据分析平台概述

大数据4V特征：价值高（Value），速度快（Velocity），体量大（Volume），种类多（Variety）

大数据7个应用场景：环境，教育行业，医疗行业，农业，智慧城市，零售行业，金融行业



Lambda的数据通道分为两条分支：实时流和离线。Hadoop就是Lambda架构的典型代表。

Hadoop

分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。

Hadoop是一个以Hadoop分布式文件系统（HDFS）和MapReduce为核心的分布式基础架构。用户可以在不了解分布式底层细节的情况下，开发分布式程序。Hadoop是项目的总称，主要是由分布式存储（HDFS），分布式计算（MapReduce），集群资源管理YARN组成。Hadoop是一整套大数据存储和处理方案，包括数据收集，数据存储（离线存储，在线存储）和数据分析与挖掘。

第二章　分布式文件存储系统HDFS

（一）分布式文件系统

分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。

分布式文件系统的设计基于**客户机/服务器模式**。

文件级别分布式文件系统

缺点：1.难以负载均衡。2.难以并行处理。

块级别分布式文件系统

将文件分成等大的数据块（block）；Write one， read many；备份机制

Master：存储和管理元信息
Slave：存储实际的数据块，并与Master维持心跳信息，汇报自身健康状态以及负载情况
Client：用户通过客户端完成文件系统的管理和相关操作

（二）HDFS介绍

特点：易于扩展的块级别分布式文件系统；运行在大量普通廉价机器上，提供容错机制；为大量用户提供性能不错的文件存取服务

优点：高容错；适合批处理；适合大数据处理；流式文件访问；可以构建在廉价的机器上

缺点：延迟较高；不适合小文件存取；不支持并发写入和随机修改

（三）HDFS概念与架构

1. 概念

块：HDFS中的文件在物理上是分块存储（block），块的大小可以通过配置参数(dfs.blocksize)来规定，默认大小在hadoop2.x版本中是128M，老版本中是64M

Metadata：元数据，包括命名空间、文件到文件块的映射、文件块到DataNode的映射三部分（每个文件的块列表，以及每个块的存放节点列表）。
NameNode：HDFS系统中的管理者，负责管理和维护文件系统的元信息。
Secondary NameNode：NameNode的备份节点，当NameNode发生故障时进行数据恢复。
DataNode：HDFS文件系统中保存数据的节点。
Client：客户端，HDFS文件系统的使用者。

2. 架构

NameNode

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及元数据，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。

FsImage

FsImage文件没有记录每个块存储在哪个数据节点。而是由NameNode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NameNode，此后会定期执行这种告知操作，以确保NameNode的块映射是最新的。

Namenode的启动

名称节点启动时，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。
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。

SecondaryNameNode可以周期性地把 NameNode 中的 EditLog 日志文件合并到 FSImage 镜像文件中，从而减小 EditLog 日志文件的大小，缩短集群重启时间。

DataNode

数据节点负责数据的存储和读取，根据客户端或者是NameNode的调度来进行数据的存储和检索，并且向NameNode定期发送自己所存储的块的列表。

（四）HDFS关键技术

1. 副本放置策略

一个集群通常由多个机架构成，每个机架由16-64个datanode节点组成。默认情况下每个block有三个副本。

客户端与Datanode同节点

第一个副本放置在同节点的DataNode上，另外两个副本写到另外一个相同机架的不同DataNode上。

客户端与Datanode不同节点

随机选择一个DataNode作为第一个副本放置节点，另外两个副本写到另外一个相同机架的不同DataNode上。

2. 数据错误与恢复

名称节点出错

HDFS把FsImage和Editlog同步复制到备份服务器SecondaryNameNode上。当名称节点出错时根据SecondaryNameNode中的数据进行恢复。

数据节点出错

namenode就无法收到来自一些datanode的心跳信息时，这些datanode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。

namenode定期检查，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。

数据出错

会根据隐藏文件进行校验

第三章　分布式处理框架MapReduce

（一）MapReduce概述

MapReduce的出现，使得用户可以把主要精力放在设计数据处理算法上，其他的分布式问题如节点间通信、数据切分、任务并行化等全都由MapReduce运行时环境完成，用户无需关心这些细节。

优点：简化了分布式程序设计，易于编程；良好的扩展性；高容错性；高吞吐率

缺点：输入为静态数据，不适合流式计算；延迟高，不适合实时计算；不适合DAG计算

（二）MapReduce编程模型

编程思想

核心思想是分而治之，将一个分布式计算过程拆解成两个阶段：

Map阶段：由多个可并行执行的Map Task构成。将待处理数据集按照数据量大小切分成等大的数据分片，每个分片交由一个任务处理

Reduce阶段：由多个可并行执行的Reduce Task构成。对前一阶段中各种任务产生的结果进行规约，得到最终结果

可编程组件

Hadoop将输入数据切分成若个split，并将每个split交给一个Map Task处理；Map Task以迭代方式从对应的split中解析出一系列<key，value>，并调用map函数处理，将输入<key,value>对映射成另外一些<key,value>对。待数据处理完后，Reduce Task启动多线程远程拷贝其对应的数据partition，依据key对中间数据进行分组（grouping），以组为单位调用reduce函数对数据进行规约。迭代将最终产生的<key,value>保存到输出文件中。

1. InputFormat

按照某个策略将输入数据切分成若干个split，以便确定Map Task个数以及对应的split。然后为Mapper提供输入数据：给定某个split，将其解析成一系列<key,value>。

2. Mapper

map方法对输入的键值对进行处理，产生一系列的中间键值对，转换后的中间键值对可以有新的键值类型。该函数被多个Map Task并行调用。

public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

protected void map(KEYIN key, VALUEIN value, Context context)

3. Partitioner

对Mapper产生的中间结果进行分区，以便将同区的数据交给同一个Reducer处理

默认实现类为HashPartitioner

4. Combiner

可看作Map端的Local Reducer，通常跟Reducer的逻辑一样，主要作用是对Mapper输出结果做局部聚集，以减少本地磁盘写入量和网络数据传输量

5. Reducer

reduce方法对基于Mapper产生的结果进行规约操作，产生最终结果。

public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)

6. OutputFormat

用于描述输出数据的格式，将用户提供<key,value>对写入特定格式的文件中

（三）Shuffle

我们对Shuffle过程的期望：完整地从map task端拉取数据到reduce端；在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗；减少磁盘IO对task执行的影响。

Map端Shuffle

①分区partition

map函数处理得到的（key,value）写入到缓冲区之前需要先进行分区操作，这样就能把map任务处理的结果发送给指定的reducer去执行

②写入环形内存缓冲区

每一个Map Task都有一个环形缓存，Mapper的输出会首先写进这个缓存里面。

③执行溢出写

当环形缓存里面的数据达到一个阈值，一个后台线程就会开始向磁盘spill这些内容。在向磁盘写入前，会先在缓存内部进行排序sort和合并combiner操作。此sort为local sort，把buffer内数据按partition值与key升序排列。

④归并merge

一个Map task可能产生多个spill文件，当Map写出最后一个输出时，会将所有的spill文件进行归并（merge）生成最终的一个已分区且已排序的大文件。在这个过程中仍然会进行sort和combiner。

溢出写文件归并完毕后，Map Task将删除所有的临时溢出写文件，并告知NodeManager任务已完成，只要其中一个MapTask完成，ReduceTask就开始复制它的输出。

Reduce端Shuffle

①复制copy

map任务完成后，会使用心跳机制通知它们的application master。对于指定作业，AM知道map输出和主机位置之间的映射关系。reducer中的一个线程定期询问AM以便获取map输出主机的位置。直到获得所有输出位置。

Reducer从不同的Map Task拉取数据。将Map端复制过来的数据先放入内存缓冲区中，如果map输出较大，则被输出到磁盘。

②归并merge

当属于该reducer的各map端输出全部拷贝完成，开始执行归并排序操作。最终Reduce shuffle过程会输出一个整体有序的数据块。

如果内存缓冲区中能放得下这次数据的话就直接把数据写到内存中，即内存到内存merge。

当内存缓存区中存储空间占用达到一定阈值，把内存中的数据merge输出到磁盘上一个文件中，即内存到磁盘merge。

③group

针对已根据键排好序的Key构造对应的Value迭代器。这个迭代器的Key使用属于同一个组的所有Key的第一个Key。

第五章　Hbase

（一）Hbase简介

HBase是一个高可靠、高性能、面向列、可伸缩的分布式数据库，主要用来存储非结构化和半结构化的松散数据。

为什么需要Hbase

受限于Hadoop的高延迟数据处理机制，使之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（块级别存储），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，且在数据结构变化时一般需要停机维护

HBase与传统关系数据库的对比分析

（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，统一把数据存储为字符串

（2）数据操作：关系数据库中有丰富的操作，如复杂的多表连接。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

（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）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。HBase只有一个索引——行键

（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

HBase访问接口

有Java API，HBase Shell（HBase管理），Hive（类SQL）等

（二）HBase模型

HBase是一个稀疏、多维度、排序的映射表，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。进而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。

（三）Hbase实现原理

Region

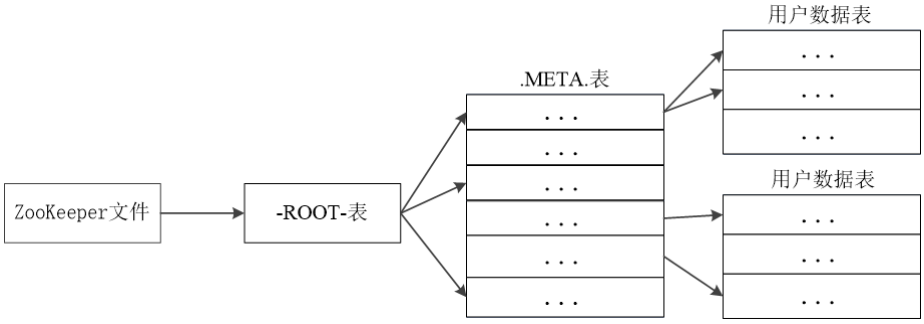
Region是HBase数据管理的基本单位，每个Region有三个主要要素：属于哪张表，包含的第一行，包含的最后一行。Region是一段Rowkey数据的集合（水平切分的表的一部分）。当表初写数据时，表中只有一个region，随着数据的增多，region开始变大，当它达到限定的阈值大小时，该region分裂为两个大小基本相同的region。同一个Region不会被分拆到多个Region服务器。

每个Region包含一定的行数范围，当查询一条数据时，会首先从元数据中查询该条数据的Rowkey属于哪个Region，然后到指定的Region中查找。当一个Region过大时，在这个Region中查找Rowkey的时间也就越长。但通常较少的Region数可以使集群运行更加稳定，当region和列族数量过多，内存占用也会提高。

Region的定位

元数据表（.META.表），存储了Region和Region服务器的映射关系，即用户数据表的Region位置信息。当HBase表很大时， .META.表也会分裂成多个Region。

根数据表（-ROOT-表），记录.META.表的Region位置信息。-ROOT-表只有唯一一个Region。Zookeeper文件记录了-ROOT-表的位置。



寻址时客户端只需要询问Zookeeper服务器，不需要连接Master服务器

功能组件

HBase的实现包括三个主要的功能组件：

（1）库函数：链接到每个客户端

（2）一个Master主服务器

（3）许多个Region服务器

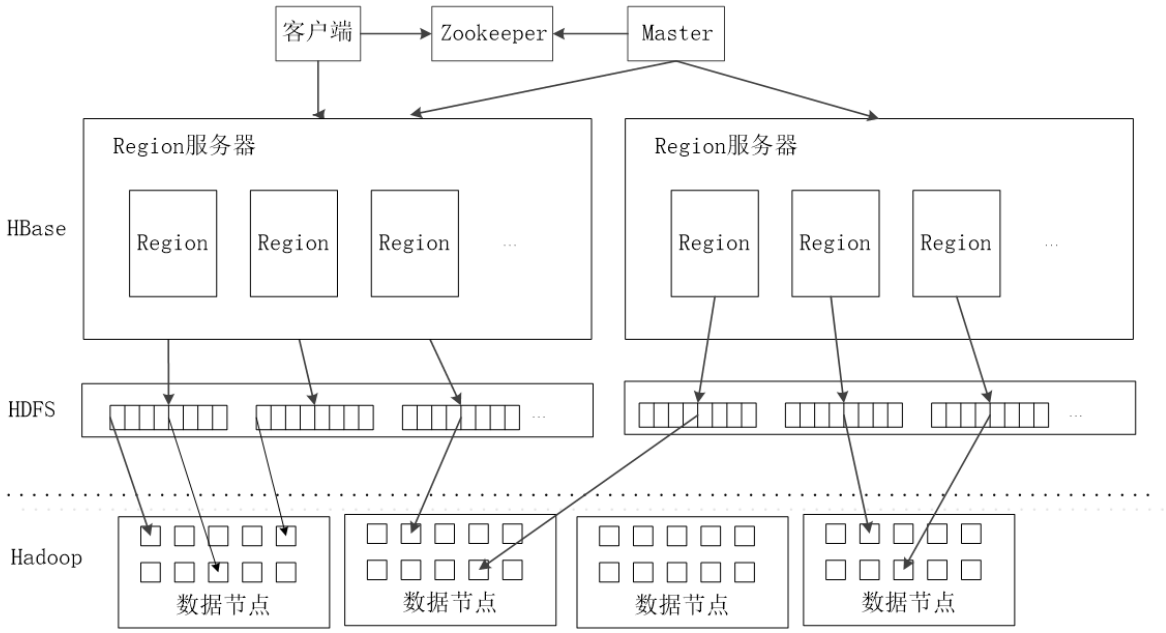
主服务器Master负责管理和维护HBase表的分区信息，维护Region服务器列表，分配Region，负载均衡。

Region服务器负责存储和维护分配给自己的Region，处理来自客户端的读写请求。客户端并不是直接从Master主服务器上读取数据，而是在获得Region的存储位置信息后，直接从Region服务器上读取数据。

客户端并不依赖Master，而是通过Zookeeper来获得Region位置信息，大多数客户端甚至从来不和Master通信，这种设计方式使得Master负载很小。

（四）Hbase运行机制

系统架构



客户端

客户端包含访问HBase的接口，同时在缓存中维护着已经访问过的Region位置信息，用来加快后续数据访问过程

Zookeeper服务器

Zookeeper选举出一个Master作为集群的总管，并保证在任何时刻总有唯一一个Master在运行

Master

主服务器Master主要负责表和Region的管理工作：管理用户对表的增加、删除、修改、查询等操作；实现不同Region服务器之间的负载均衡；在Region分裂或合并后，负责重新调整Region的分布；对发生故障失效的Region服务器上的Region进行迁移

Region服务器

Region服务器是HBase中最核心的模块，负责维护分配给自己的Region，并响应用户的读写请求

Region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。要被写入的数据首先被写入到MemStore和Hlog中。只有当操作写入Hlog之后，commit()调用才会返回。

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。

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。

3. StoreFile的合并

每次刷写都生成新的StoreFile，数量太多，影响查找速度。数量达到一个阈值后调用Store.compact()把多个合并成一个。

Store工作原理

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

HLog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，为每个Region服务器配置一个HLog文件，它是一种预写式日志。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

Zookeeper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。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。

（五）HBase编程

单机模式只需要start-hbase.sh后可hbase shell

分布式模式需要先start-all.sh然后start-hbase.sh（三个节点都要有hbase）

第六章　Hive

（一）Hive概述

为什么使用Hive?

直接使用hadoop所面临的问题：人员学习成本太高；MapReduce实现复杂查询逻辑开发难度太大。

使用Hbase问题：架构设计复杂，使用HDFS作为分布式存储在只是存储少量数据时也不占优势；Hbase不支持表的关联操作，因此数据分析是HBase的弱项。

Hive

Hive是一个构建于Hadoop顶层的数据仓库工具，某种程度上可以看作是用户编程接口，本身不存储和处理数据。

依赖HDFS存储数据，依赖MapReduce处理数据。定义了简单的类SQL查询语言——HiveQL(HQL)，用户可以通过编写的HiveQL语句运行MapReduce任务。

Hive与传统数据库的对比

（1）数据插入：在传统数据库中同时支持导入单条数据和批量数据，而Hive中仅支持批量导入数据。

（2）数据更新：Hive不支持数据更新。Hive是一个数据仓库工具，而数据仓库中存放的是静态数据。

（3）索引：Hive不像传统的关系型数据库那样有键的概念，它只提供有限的索引功能，使用户可以在某些列上创建索引来加速一些查询操作，Hive中给一个表创建的索引数据被保存在另外的表中。

（4）分区：传统的数据库提供分区功能来改善性能。Hive也支持分区功能。

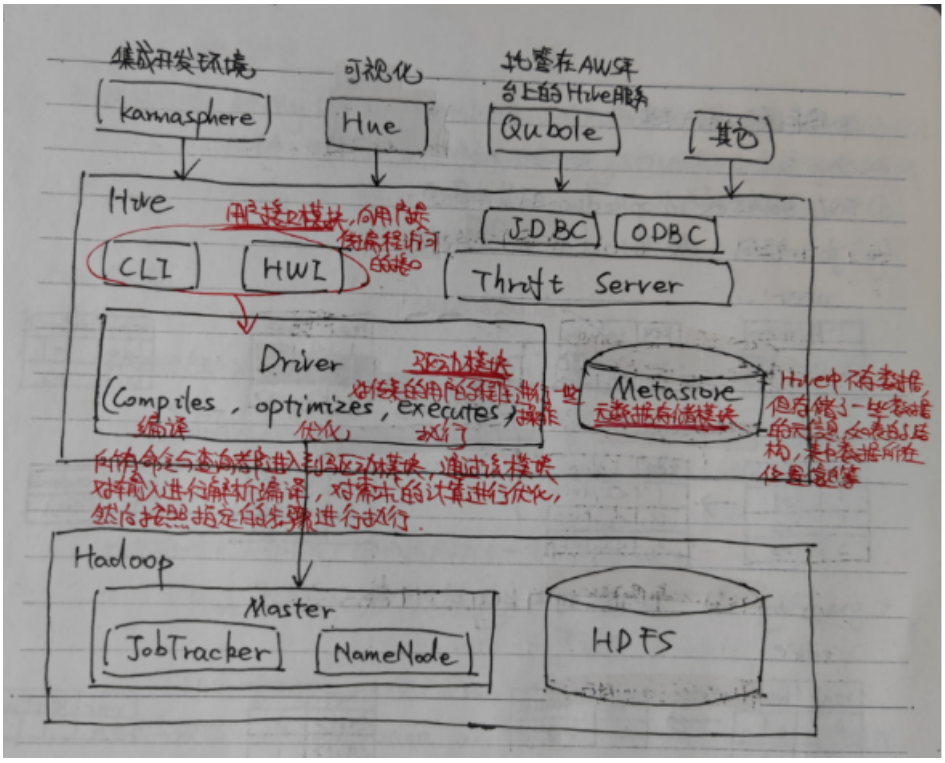
（5）执行延迟：因为Hive构建于HDFS与MapReduce上，所以对比传统数据库来说Hive的延迟比较高。

（6）扩展性：传统关系数据库很难横向扩展，纵向扩展的空间也很有限。相反Hive的开发环境是基于集群的，所以具有较好的可扩展性。

（二）Hive系统架构

1. Hive组成模块

包括用户接口模块，驱动模块，元数据存储模块（Metastore）。



2. SQL语句转换成MapReduce的基本原理

join转化为MapReduce任务的具体过程

user

uid	name
1	Lily
2	Tom

key	value
1	<1, Lily>
2	<1, Tom>

order

uid	orderid
1	101
1	102
2	103

key	value
1	<2, 101>
1	<2, 102>
2	<2, 103>

Map

Shuffle
Sort

key	value
1	<1, Lily>
1	<2, 101>
1	<2, 102>
2	<2, Tom>
2	<2, 103>

Reduce

name	orderid
Lily	101
Lily	102
Tom	103

group by转化为MapReduce任务的具体过程

score

rank	level
A	1
A	1
B	2

key	value
<A, 1>	2
<B, 2>	1

Map

Shuffle
Sort

key	value
<A, 1>	2
<A, 1>	1
<B, 2>	1

Reduce

rank	level	value
A	1	3
B	2	1

（三）Hive实践编程

Hive一共有3种运行模式：

内嵌模式：将元数据保存在本地内嵌的数据库中。但一个内嵌的 Derby 数据库每次只能访问一个数据文件，这也就意味着它不支持多会话连接。

本地模式：这种模式是将元数据保存在本地独立的数据库中（一般是 MySQL），这就可以支持多会话和多用户连接了。

远程模式：此模式应用于 Hive 客户端较多的情况。把 MySQL 数据库独立出来，将元数据保存在远端独立的 MySQL 服务中，避免了在每个客户端都安装 MySQL 服务从而造成冗余浪费的情况。远程模式下，需要单独启动metastore服务。在生产环境中，建议用远程模式。

（四）Hive与HBase集成

Hbase数据库没有类SQL的查询语言，因此在实际应用中使用起来不方便。而Hive支持Hive QL语言，若将Hbase与Hive集成，则可以使用HiveQL直接操作Hbase。

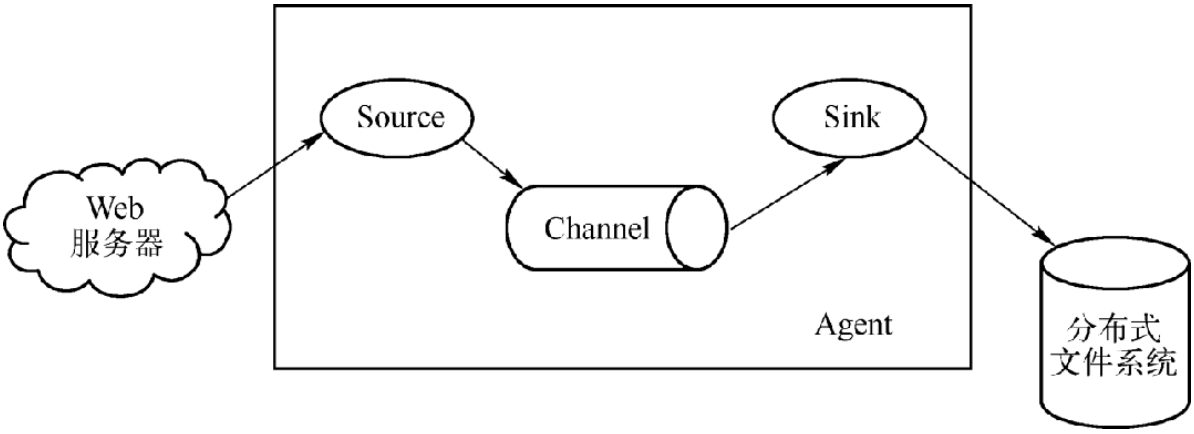
Hive与Hbase集成的关键就是如何将Hbase中的表与Hive中的表在列级别上建立映射关系。

在Hive中创建的Hbase映射表的数据都只存在于Hbase中，Hive的数据仓库中不存储任何数据，Hive只维护元数据。Hbase是Hive的数据源，Hive相当于Hbase的客户端，可以对Hbase数据进行查询与统计。若Hbase停止，则Hive中不会查询到任何数据。

第七章　Flume和Kafka

（一）Flume

Flume 是一个分布式的、可靠的、高可用的系统，能够将不同数据源的海量日志数据进行高效收集、聚合、移动，最后存储到一个中心化的数据存储系统中。



Flume架构设计

Web Server：数据产生的源头。

Agent：Flume的核心是Agent 。Agent是一个Java进程，包含组件Source、 Channel、 Sink，且运行在日志收集端，通过Agent接收日志，然后暂存起来，再发送到目的地。（Agent使用JVM 运行Flume。每台机器可以运行多个agent，但是在一个agent中只能包含一个source。）

Source（源）：用于从Web Server收集数据，然后发送到Channel。

Channel（通道）：用来从Source接收数据，然后发送到Sink，Channel存放临时数据，有点类似队列一样。

Sink（接收器）：用来把数据发送到目标地点，如上图放到HDFS中。

Flume编程

首先启动Hadoop start-all.sh
然后启动flume

（二）Kafka

Kafka是一个开源流处理平台，由Scala和Java编写。作为一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者在网站中的所有动作流数据。

具有良好特性：高吞吐低延迟，可扩展性，持久性可靠性，容错性，高并发，顺序保证（保证一个分区内的消息的有序性），异步通信（允许用户把一个消息放入队列，在需要的时候再去处理它们）

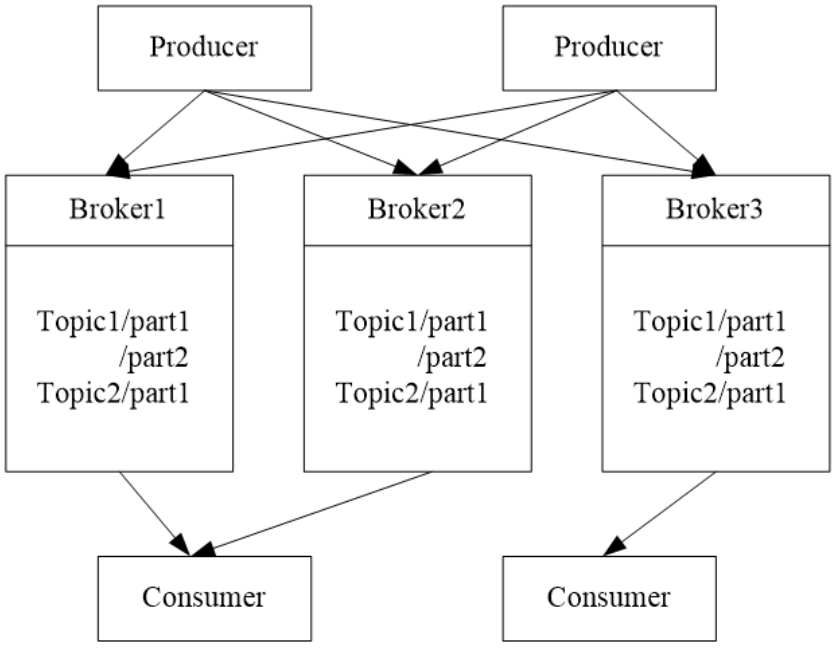
借助于Kafka作为交换枢纽，可以很好解决不同系统之间的数据生产/消费速率不同的问题。

发布订阅模式

在发布订阅消息系统中，消息被持久化到一个主题（topic）中。与点对点消息系统不同的是，消费者可以订阅一个或多个主题，消费者可以消费该主题中所有的数据，同一条数据可以被多个消费者消费，数据被消费后不会立马删除。在发布订阅消息系统中，消息的生产者称为“发布者”，消费者称为“订阅者”。

Kafka的总体架构

一个典型的Kafka集群中包含若干Producer、若干Broker、若干Consumer以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置。Producer在发布消息时，会发布到特定的Topic，Consumer从特定的Topic获取消息。每个Topic包含一个或多个Partition，以 Partition 为单位传输。



启动kafka只要先启zookeeper再启kafka即可。

ACK机制

Kafka的Ack机制指producer的消息发送确认机制，其影响kafka集群的吞吐量和消息可靠性。

Ack=0，producer不等待broker同步完成，消息发送完毕继续发送下一批信息。提供了最低延迟，但持久性最弱，当服务器发生故障时很可能发生数据丢失。如果leader死亡，producer继续发送消息，broker接收不到数据就会造成数据丢失。

Ack=1，producer要等待leader成功收到消息并确认，才发送下一条message。提供较低的延迟性以及较好的持久性。但是如果partition下的leader死亡，而follower尚未复制数据，数据就会丢失。

Ack=-1，leader收到所有消息，且follower同步完数据，才发送下一条数据。延迟性最差，持久性最好。

备份机制

备份机制就是把相同的数据拷贝到多台broker机器上，而这些相同的数据拷贝在 Kafka 中被称为副本。

副本被分类为领导者副本（Leader）和追随者副本（Follower）。前者对外提供服务，消费者总是从领导者副本读消息；后者只是被动地追随领导者副本而已，不能与外界进行交互，它只向领导者副本发送请求，请求领导者把最新生产的消息发给它。

kafka每个主题划分为多个分区，从broker层级上看，生产者向分区写入消息，消费者从分区读取消息。

（三）Flume和Kafka集成

如果Flume直接对接实时计算框架，当数据采集速度大于数据处理速度，很容易发生数据堆积或者数据丢失，而kafka可以当做一个消息缓存队列，当数据从数据源到flume再到Kafka时，数据一方面可以同步到HDFS做离线计算，另一方面可以做实时计算。

（四）Flume和Hbase集成

Flume使用hbase2将采集到的数据写入Hbase，但需要重写数据插入规则，自定义rowkey

第九章　Spark

（一）Spark产生背景

MR局限性

仅支持Map和Reduce两种操作；处理效率低；不适合迭代式和交互式计算；编程不够灵活

多个 MapReduce 任务之间没有基于内存的数据共享方式，只能通过磁盘来进行共享，这种方式明显比较低效。在 Spark 中，整个过程是共享内存的，而不需要将中间结果存放在分布式文件系统中。

Spark的特点

快速：任务秒级完成，支持交互式查询和流处理

通用：容纳独立的分布式系统功能如批处理，迭代计算

高度开放：支持四种语言，可以与Hadoop集成，也可以独立运行，可以运行在Hadoop数据源如HBase，HDFS，Hive上

（二）Spark生态系统

Spark Core

Spark Core是spark的核心与基础，实现了Spark的基本功能，包含任务调度，内存管理，错误恢复与存储系统交互等模块。核心技术为RDD。

1. 分布式数据集：Spark Core使用RDD来表示分布式数据集。
2. 内存计算：Spark Core使用内存计算来加速数据处理，它可以将数据集存储在内存中，并在多个任务之间共享这些数据，从而避免了频繁的磁盘IO操作。
3. 多语言支持：Spark Core支持多种编程语言，包括Scala、Java、Python和R等。
4. 高效的任务调度：Spark Core使用基于DAG（Directed Acyclic Graph）的任务调度机制，它可以将多个操作组合成一个DAG图，并根据依赖关系自动调度任务的执行顺序。

5. 可靠的容错机制：Spark Core具有强大的容错机制，它可以在节点故障或计算错误发生时自动恢复数据和任务状态。

四大组件

Spark Streaming，Spark SQL，Spark MLlib，Spark GraphX

（三）Spark的核心概念-RDD

1. 概述

RDD，弹性分布式数据集，是一个**只读**的分布式集合对象，内部由许多partitions(分片)组成，每个partition都包括一部分数据，这些partitions可以在集群的不同节点上计算，是Spark中的并行处理的单元。

RDD可以包含 Python, Java, 或者 Scala中的任何数据类型，包括用户自定义的类。

在Spark中，所有的计算都是通过RDD的创建、转换及动作完成的。

2. RDD的操作方式

RDD有两种操作方式：转换（返回RDD）和动作（返回值不是RDD）

3.转换和动作的区别

接口定义不同

转换：RDD[x]->RDD[y] （X和Y所代表的数据类型可能不同）

动作：RDD[x]->Z（Z不是RDD，可能是基本数据类型、数组等）

执行方式不同

转换只是记录RDD的转换关系，不会触发真正的分布式计算，动作才会触发程序的执行

4. 操作

转换操作	说明
map	通过自定义函数进行映射
filter	对元素过滤，保留符合条件的元素
flatMap	先映射（map），再把元素合并为一个集合
sample	对元素采样，返回一个子集
union	将两个RDD集合合并，返回并集
groupByKey	Key相同的值被分为一组
reduceByKey	返回KV对数据集，key相同的值被聚合
sortByKey	通过Key值对KV对数据集排序
join	对两个RDD进行cogroup、笛卡尔积、展平操作，(K,V)和(K,W)转换为(K,(V,W))
cogroup	组合两个key值相同的KV元素

Action操作	说明
reduce	通过自定义函数聚合数据集中的元素
collect	相当于toArray，将分布式的RDD返回为一个Array数组
count	返回RDD中元素的个数
first	返回数据集中第一个元素，与take(1)类似
take(n)	返回数据集中前n个元素形成的数组
saveAsTextFile	保存数据到文本文件
countByKey	只用于KV类型RDD，返回每个Key的个数
foreach(func)	对数据集中每个元素使用func函数

（四）Spark的基本工作流程

每个Spark程序的运行时环境是由一个Driver进程和多个Executor进程构成的，它们运行在不同机器上，并通过网络相互通信。Driver进程运行用户程序（main函数），并依次经历逻辑计划生成、物理计划生成、任务调度等阶段后，将任务分配到各个Executor上执行；Executor进程是拥有独立计算资源的JVM实例，其内部以线程方式运行Driver分配的任务。

1. 流程

初始化SparkContext（指定运行模式） -> 申请资源 -> 初始化Executor -> 解析RDD，划分Stage，调度任务 -> 发送任务到Executor -> 执行计算任务 -> 返回计算结果 -> 关闭SparkContext，回收资源

ClusterManager：在Standalone模式中为Master（Spark集群的管理器），控制整个集群，监控Worker。在YARN模式中为资源管理器

Worker：DataNode，负责控制计算节点，启动Executor或Driver。在YARN模式中为NodeManager，负责计算节点的控制。

Driver：运行Application的main()函数并创建SparkContext。

Executor：执行器，在worker node上执行任务的组件、用于启动线程池运行任务。每个Application拥有独立的一组Executors。

SparkContext：整个应用的上下文，控制应用的生命周期。

RDD：Spark的基本计算单元，一组RDD可形成执行的有向无环图RDD Graph。

DAG Scheduler：根据作业（Job）构建基于Stage的DAG，并提交Stage给TaskScheduler。

TaskScheduler：将任务（Task）分发给Executor执行。

2. 运行模式

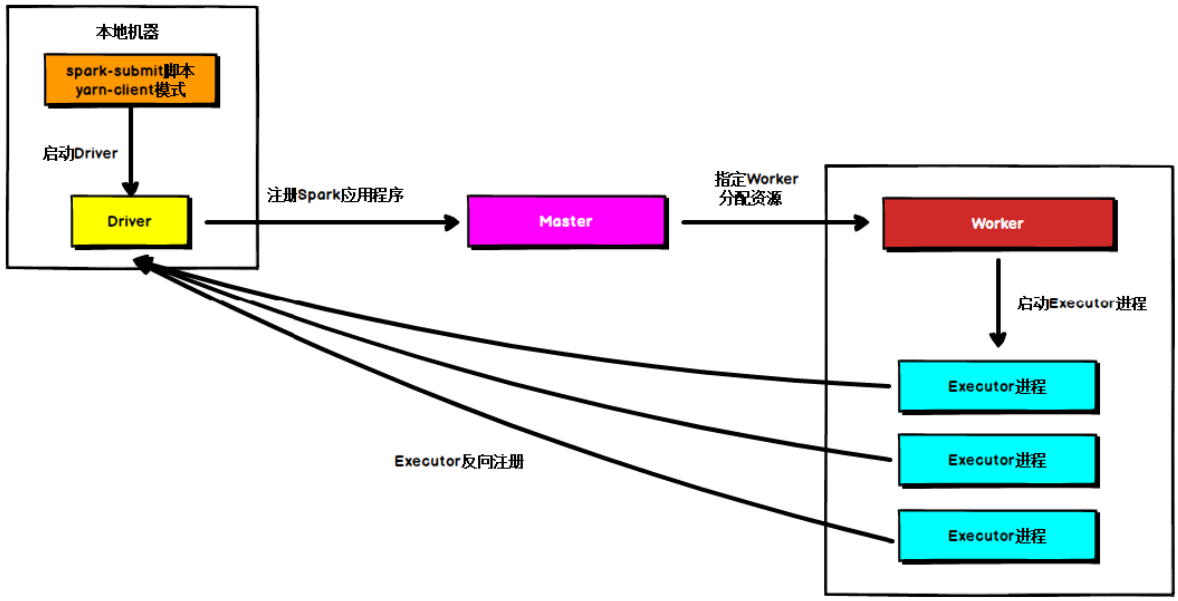
每个Spark程序由一个Driver和多个Executor组成，他们之间通过网络通信。用户可将程序的Driver和Executor两类进程运行在不同类型的系统中，进而产生多种运行模式。

YARN模式：将Hadoop YARN作为资源管理器和调度系统，让spark程序运行在YARN之上。

Spark必须文件在各worker上都有。

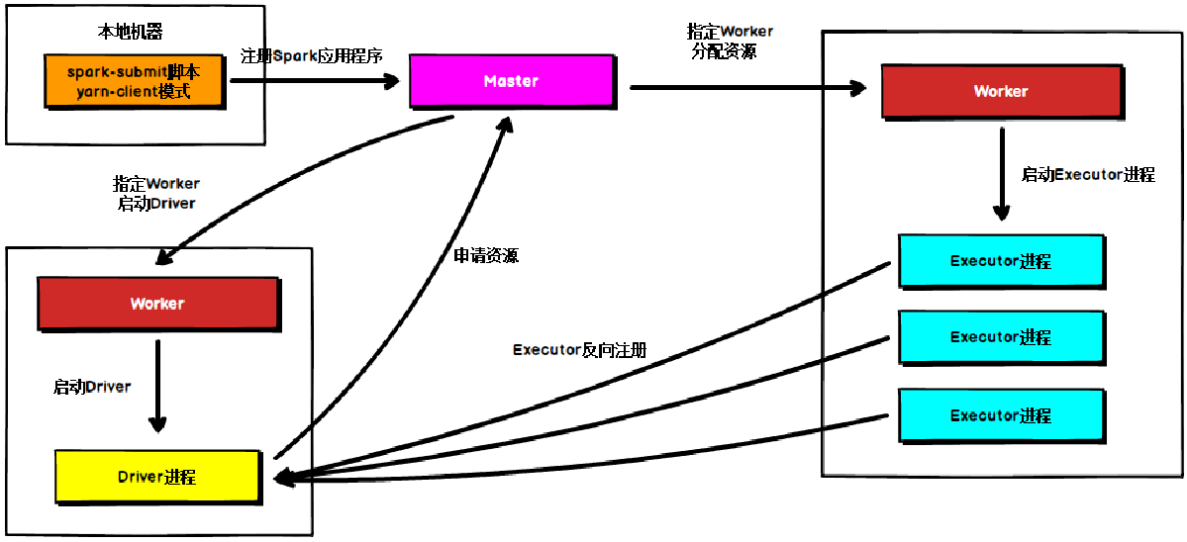
运行模式	说明
Local 模式	本地模式，Driver与worker均运行在本地，方便调试。用户可根据需要设置多个worker，worker本身以线程方式运行。
Standalone-client 模式	Spark独立集群运行环境。Driver运行在Client，不受Master管理和控制，Executor运行在Worker上，受Master管理和控制
Standalone-cluster模式	Spark独立集群运行环境，Driver和Executor运行在Worker节点（工作节点）上。
Yarn-Cluster 模式	运行在 Yarn上的模式，Driver和executor均运行在Yarn container中，受YARN管理和控制。
Yarn-Client 模式	运行在 Yarn上的模式，Driver运行在Client（客户端）上，不受YARN管理和控制

Standalone Client



在Standalone Client模式下，Driver在任务提交的本地机器上运行。Driver启动后向Master注册应用程序，Master根据submit脚本的资源需求找到内部资源至少可以启动一个Executor的所有Worker，然后在这些Worker之间分配Executor，Worker上的Executor启动后会向Driver反向注册。所有的Executor注册完成后， Driver开始执行main函数，之后执行到Action算子时，开始划分stage，每个stage生成对应的taskSet，之后（由Driver）将task分发到各个Executor上执行。

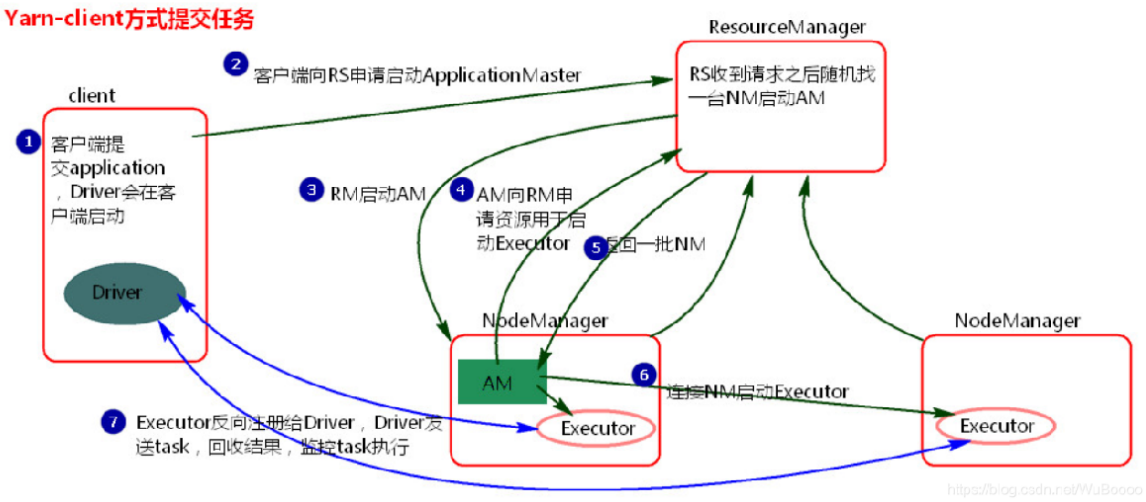
Standalone Cluster



在 Standalone Cluster 模式下，任务提交后，Master 会找到一个 Worker 启动 Driver进程，Driver 启动后向 Master 注册应用程序，Master 根据 submit 脚本的资源需求找到内部资源至少可以启动一个 Executor 的所有 Worker，然后在这些 Worker 之间分配 Executor。Worker 上的 Executor 启动后会向 Driver 反向注册，所有的 Executor 注册完成后，Driver 开始执行 main 函数，之后执行到 Action 算子时，开始划分 stage，每个 stage 生成对应的 taskSet，之后将 task 分发到各个 Executor 上执行。

Yarn Client

执行流程图：



1.客户端提交一个Application，在客户端启动一个Driver进程。

2.Driver进程会向RM(ResourceManager)发送请求，启动AM(ApplicationMaster)。

3.RS收到请求，随机选择一台NM(NodeManager)启动AM。这里的NM相当于Standalone中的Worker节点。

4.AM启动后，会向RM请求资源用于启动Executor。

5.RS会找到一批NM返回给AM，用于启动Executor。AM会向NM发送命令启动Executor。

6.Executor启动后，会反向注册给Driver，Driver发送task到Executor,执行情况和结果返回给Driver端。

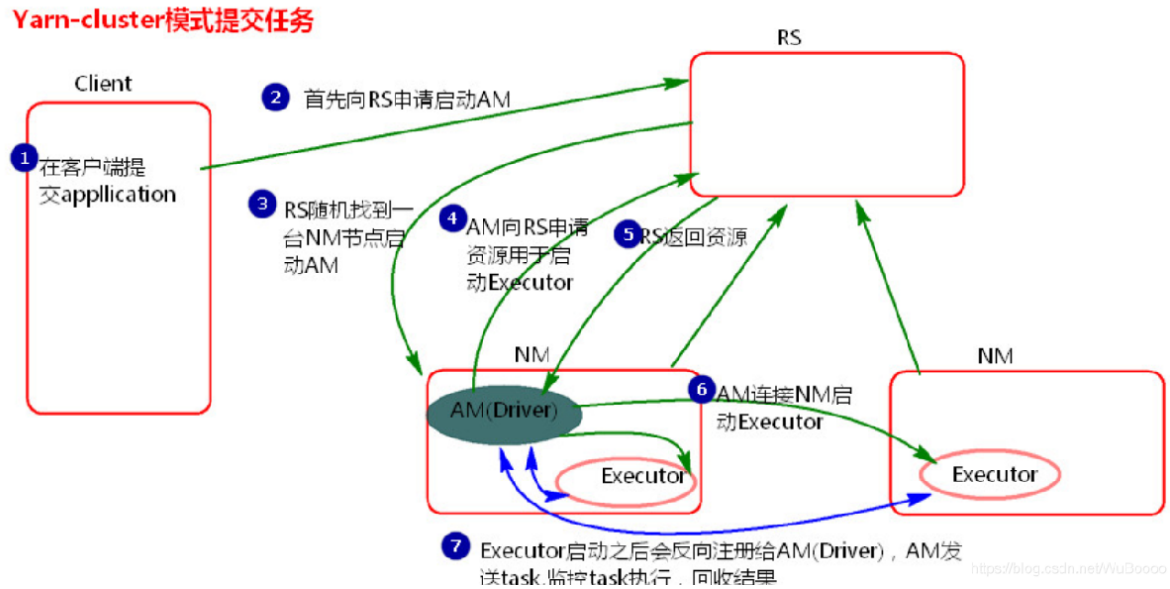
特点：

1、Yarn-client模式同样仅适用于测试，因为Driver运行在本地，Driver会与yarn集群中的Executor进行大量的通信，会造成客户机网卡流量的大量增加。

2、ApplicationMaster的作用：为当前的Application申请资源以及给NodeManager发送消息启动Executor。

Yarn Cluster

执行流程图：



1.客户机提交Application应用程序，发送请求到RS(ResourceManager),请求启动AM(ApplicationMaster)。

2.RS收到请求后随机在一台NM(NodeManager)上启动AM（相当于Driver端）。

3.AM启动，AM发送请求到RS，请求一批container用于启动Executor。

4.RS返回一批NM节点给AM。

5.AM连接到NM,发送请求到NM启动Executor。

6.Executor反向注册到AM所在的节点的Driver。Driver发送task到Executor。

特点：

1.Yarn-Cluster主要用于生产环境中，因为Driver运行在Yarn集群中某一台nodeManager中，每次提交任务的Driver所在的机器都是随机的，不会产生某一台机器网卡流量激增的现象，缺点是任务提交后不能看到日志。只能通过yarn查看日志。

2.ApplicationMaster的作用：为当前的Application申请资源以及给nodemanager发送消息启动Executor。任务调度。(这里和client模式的区别是AM具有调度能力，因为其就是Driver端，包含Driver进程)

3.Spark提交应用工具 Spark-Submit

第十章　RDD编程

（一）RDD操作

1.转换操作

对于RDD而言，每一次转换操作都会产生不同的RDD，供给下一个“转换”使用。

转换得到的RDD是惰性求值的，也就是说，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会从血缘关系源头开始，进行物理的转换操作。

2.行动操作

行动操作是真正触发计算的地方。Spark程序执行到行动操作时，才会执行真正的计算，从文件中加载数据，完成一次又一次转换操作，最终，完成行动操作得到结果。

3.持久化

持久化后的RDD将会被保留在计算节点的内存中被后面的行动操作重复使用。可以通过持久化（缓存）机制避免重复计算的开销

可以使用persist()方法对一个RDD标记为持久化。之所以说“标记为持久化”，是因为出现persist()语句的地方，并不会马上计算生成RDD并把它持久化，而是要等到遇到第一个行动操作触发真正计算以后，才会把计算结果进行持久化

4.分区

RDD是弹性分布式数据集，通常RDD很大，会被分成很多个分区，分别保存在不同的节点上。

分区与分块区别

分块：只是把数据按照一定的大小进行打包，不考虑数据的分类，只是分为一定数据大小的block块，存储在不同机器上。

分区：把数据按照一定的规则，进行分类存储在不同的机器上。

分区的作用

1.增加并行度

2.减小通信开销

RDD分区原则

RDD分区的一个原则是使得分区的个数尽量等于集群中的CPU核心（core）数目

（二）代码

reduceByKey和groupByKey的区别

reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义

groupByKey也是对每个key进行操作，但只生成一个sequence，groupByKey本身不能自定义函数，需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作

sortByKey()和sortBy()

sortBy是对标准的RDD进行排序，而sortByKey函数是对PairRDD进行排序，也就是有Key和Value的RDD。sortBy可以指定对键还是value进行排序。

对于给定的两个输入数据集(K,V1)和(K,V2)，只有在两个数据集中都存在的key才会被输出，最终得到一个(K,(V1,V2))类型的数据集。

第十一章　Spark SQL

（一）Spark SQL概述

Spark SQL 的版本历史

Shark：1.0之后被废弃

Spark 1.0：Spark SQL诞生，开始成为顶级项目

Spark 1.3 ： 推出了DataFrame，取代了原有的SchemaRDD

Spark SQL 1.6：DataSet 诞生，它添加更强约束，属于测试阶段

Spark 2.0： DataSet属于正式阶段

SparkSession是Spark-2.0引入的新概念。SparkSession为用户提供了统一的切入点。在Spark的早期版本中，SparkContext是Spark的主要切入点，通过sparkContext来创建和操作RDD。SparkSession实质上是SQLContext和HiveContext的组合，所以在SQLContext和HiveContext上可用的API在SparkSession上同样是可以使用的。SparkSession内部封装了SparkContext，所以计算实际上是由SparkContext完成的。

（二）RDD、DataFrames和DataSet

三者关联

DataFrame 和 DataSet 是 Spark SQL 提供的基于 RDD 的结构化数据抽象。它既有 RDD **不可变**、分区、存储依赖关系等特性，又拥有类似于关系型数据库的结构化信息。基于 DataFrame 和 DataSet API 开发出的程序会被自动优化，使得开发人员不需要操作底层的 RDD API 来进行手动优化，大大提升开发效率。

DataFrame与RDD的区别

RDD是整个Spark平台的存储、计算以及任务调度的逻辑基础，更具有**通用性**，适用于**各类数据源**，是分布式的Java对象的集合。

而DataFrame是针对**结构化数据源**的高层数据抽象，在DataFrame对象创建过程中必须**指定数据集的结构信息**（Schema），所以DataFrame是具有专业性的数据抽象，只能读取具有鲜明结构的数据集。

DataFrame与RDD最大的不同在于，RDD仅是**一条条数据的集合**，由于不了解RDD数据集内部的结构，Spark作业执行只能进行简单通用的优化，而DataFrame所表示的**二维表数据集**带有Schema结构信息，可以根据结构信息进行针对性的优化，最终优化运行效率。

DataSet

Dataset是DataFrame的扩展，它提供了类型安全，面向对象的编程接口。 Dataset可以让用户通过类似于sql的表达式对数据进行查询。

1 ）DataSet是在Spark1.6中添加的新的接口。

2）与RDD相比，保存了更多的描述信息，概念上等同于关系型数据库中的二维表。

3）与DataFrame相比，保存了类型信息，是强类型的，提供了编译时类型检查。

4）调用Dataset的方法先会生成逻辑计划，然后Spark的优化器进行优化，最终生成物理计划，然后提交到集群中运行。

5）DataSet包含了DataFrame的功能，在Spark2.0中两者得到了统一：DataFrame表示为DataSet[Row]，即DataSet的子集。

DataSet与DataFrame区别

DataFrame = RDD + schema

1.DataFrame的每一行的固定类型为Row，只有通过解析才能获得各个字段的值。可以通过 as 方法将 DataFrame 转换为 DataSet。

2.DataFrame与DataSet均支持sparkSql操作，比如select，groupby等，也可以注册成临时表，进行sql语句操作

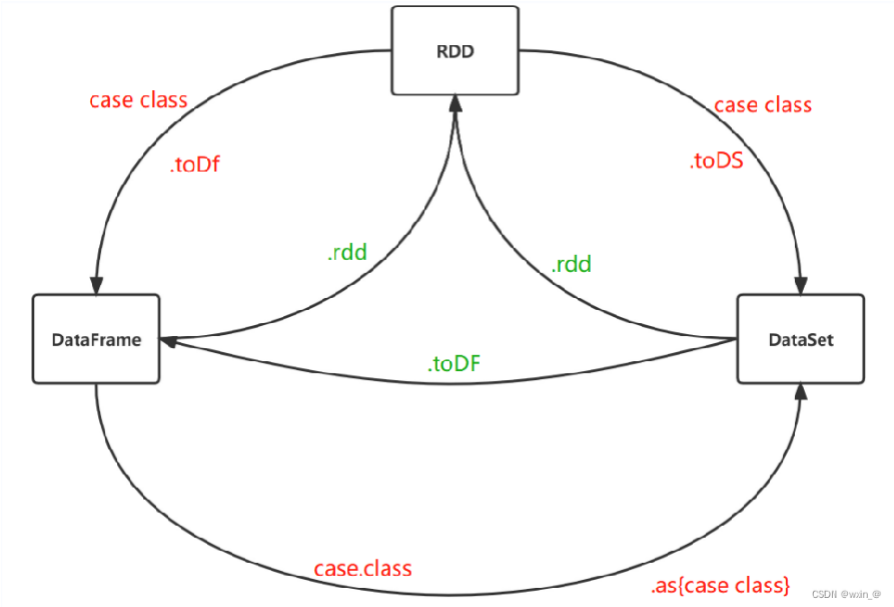
3.DataFrame与DateSet支持一些方便的保存方式，比如csv，可以带上表头，这样每一列的字段名就可以一目了然

DataSet = RDD + case class

1.DataSet与DataFrame拥有相同的成员函数，区别只是每一行的数据类型不同。

2.DataSet的每一行都是case class，在自定义case class之后可以很方便的获取每一行的信息

RDD、DataFrame、DataSet间的相互转换



（三）SparkSQL查询方式

DataFrame和DataSet支持两种查询方式：一种是DSL风格，另外一种是SQL风格

(1)DSL风格：

需要引入import spark.implicit. _ 这个隐式转换，可以将RDD隐式转换成DataFrame或DataSet

(2)SQL风格：

需要将DataFrame注册成一张表格，如果通过CreateTempView这种方式来创建，那么该表格在Session内有效，如果通过CreateGlobalTempView来创建，那么该表格跨Session有效，但是SQL语句访问该表格的时候需要加上前缀global_temp。

需要通过sparkSession.sql方法来运行你的SQL语句。

（四）SparkSQL与Hive、Mysql、Hbase集成

Spark SQL与Hive、Mysql、Hbase集成，其核心就是Spark SQL通过hive外部表来获取HBase的表数据。

Hive是一个基于Hadoop的数据仓库架构，使用SQL语句读写和管理大型分布式数据集。Hive可以将SQL语句转换为MapReduce任务执行，大大降低Hadoop的使用门槛，减少了开发MapReduce程序的时间成本。Hive不仅可以分析HDFS文件系统中的数据，也可以分析其他存储系统，如Hbase。

Spark SQL与Hive整合后，可以在Spark SQL中使用HiveQL轻松操作数据仓库。与Hive不同的是，Hive的执行引擎为MapReduce，而Spark SQL的执行引擎为Spark RDD。

第十二章　Spark Streaming

（一）流计算

流计算：实时获取来自不同数据源的海量数据，经过实时分析处理，获得有价值的信息

流计算秉承一个基本理念，即数据的价值随着时间的流逝而降低，如用户点击流。因此，当事件出现时就应该立即进行处理，而不是缓存起来进行批量处理。为了及时处理流数据，就需要一个低延迟、可扩展、高可靠的处理引擎。

传统的数据处理流程，需要先采集数据并存储在关系数据库等数据管理系统中，之后由用户通过查询操作和数据管理系统进行交互。这隐含了两个前提：存储的数据是旧的；需要用户主动发出查询来获取结果。

流计算的处理流程一般包含三个阶段：数据实时采集、数据实时计算、实时查询服务

流处理系统与传统的数据处理系统有如下不同：

流处理系统处理的是实时的数据，而传统的数据处理系统处理的是预先存储好的静态数据

用户通过流处理系统获取的是实时结果，而通过传统的数据处理系统，获取的是过去某一时刻的结果

流处理系统无需用户主动发出查询，实时查询服务可以主动将实时结果推送给用户

（二）Spark-Streaming概述

1.简介

Spark Streaming是SparkCore API的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。

2.Spark Streaming处理数据流程

接收实时流的数据，把流数据根据一定的时间间隔拆分成一批批的数据，通过一个先进先出的队列，然后 Spark Engine从该队列中依次取出一个个批数据，把批数据封装成RDD，然后进行处理。

3.DStream

Spark Streaming最主要的抽象是DStream（离散化数据流），表示连续不断的数据流。

对应流数据的DStream可以看成是一组RDDs，即RDD的一个序列。

在内部实现上，Spark Streaming的输入数据按照时间片（如1秒）分成一段一段，每一段数据转换为Spark中的RDD，这些分段就是Dstream，并且对DStream的操作都最终转变为对相应的RDD的操作

4.Spark Streaming工作机制

在Spark Streaming中，会有一个组件Receiver，作为一个长期运行的task跑在一个Executor上。每个Receiver都会负责一个input DStream（比如从文件中读取数据的文件流，比如套接字流，或者从Kafka中读取的一个输入流等等），Spark Streaming通过input DStream与外部数据源进行连接，读取相关数据。

5.编写Spark Streaming程序的基本步骤

1.通过创建输入DStream来定义输入源

2.通过对DStream应用转换操作和输出操作来定义流计算

3.用streamingContext.start()来开始接收数据和处理流程

4.通过streamingContext.awaitTermination()方法来等待处理结束（手动结束或因为错误而结束）

5.可以通过streamingContext.stop()来手动结束流计算进程

（三）Spark Streaming整合Kafka

用spark streaming流式处理kafka中的数据，先把数据接收过来，转换为spark streaming中的数据结构Dstream。

接收数据的方式有两种：1.利用Receiver接收数据，2.直接从kafka读取数据。

1.基于Receiver的方法

这种方式利用接收器（Receiver）来接收kafka中的数据。对于所有的接收器，从kafka接收来的数据会存储在spark的executor中，之后spark streaming提交的job会处理这些数据。不适合在开发中使用。

在默认配置下, 这种情况可能会在故障下丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用Spark Streaming的预写日志机制。该机制会同步地将接收到的Kafka数据写入分布式文件系统（比如HDFS）上的预写日志中。效率会有所降低。

earliest：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费

latest：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据

none：topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的offset，则抛出异常

2.直接读取方法

Direct方式就是将kafka看成存数据的一方，不是被动接收数据，而是主动去取数据。周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。

简化并行

不需要创建多个kafka输入流，然后union它们，sparkStreaming将会创建和kafka分区数一样的rdd的分区数，而且会从kafka中并行读取数据，spark中RDD的分区数和kafka中的分区数据是一一对应的关系。

高效

Receiver实现数据的零丢失是将数据预先保存在WAL中，会复制一遍数据，会导致数据被拷贝两次，第一次是被kafka复制，另一次是写到WAL中。而Direct不使用WAL消除了这个问题。

恰好一次语义(Exactly-once-semantics)

Receiver读取kafka数据是通过kafka高层次api把偏移量写入zookeeper中，虽然这种方法可以通过数据保存在WAL中保证数据不丢失，但是可能会因为sparkStreaming和ZK中保存的偏移量不一致而导致数据被消费了多次。Direct的Exactly-once-semantics(EOS)通过实现kafka低层次api，偏移量仅仅被ssc保存在checkpoint中，消除了zk和ssc偏移量不一致的问题。

Direct方式API 使用方法

自动提交偏移量到默认主题

手动提交偏移量到默认主题

使用rdd.foreachPartition即对于每一个rdd的partition建立唯一的连接

第十三章　用户行为数据可视化

API

@responsebody

作用：将方法的返回值，以特定的格式写入到response的body区域，进而将数据返回给客户端。如果返回值是字符串，那么直接将字符串写到客户端；如果是一个对象，会将对象转化为json串，然后写到客户端。

@RequestMapping

是一个用来处理请求地址映射的注解，可用于映射一个请求或一个方法，可以用在类或方法上。

$.get(url, data, callback)

参数	描述
url	必需，规定您需要请求的URL
data	可选，规定连同请求发送到服务器的数据，格式是json
callback	可选，回掉函数，当请求成功时运行的函数

WebSocket

WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。

它的实现过程是这样的：客户端首先发起一个ws请求，这个请求最好能包含一个id（服务器将根据这个id将数据推送至客户端），当请求成功后，客户端将和服务端建立起一个通道，客户端可通过此通道向服务端发送数据，服务端同样可以根据id向该客户端下发数据。

WebSocket 一次握手就可以使客户端和服务端建立长连接，并进行双向数据传输。由于其双向传输特性，服务端可主动向客户端发送信息，实时性很高。

实时可视化思想：使用 spring boot 构建一个应用，从 kafka 队列中获取要推送的内容，通过`websocket`将内容推送到Echarts端。

API

@ServerEndpoint

将目前的类定义成一个websocket服务器端, 注解的值将被用于监听用户连接的终端访问URL地址,客户端可以通过这个URL来连接到WebSocket服务器端

集成

如果你想在Web应用程序中使用实时数据可视化，可以使用ECharts来生成图表，然后使用WebSocket实现实时数据传输和双向通信。而Spring Boot可以帮助你快速搭建和部署Web应用程序，并与ECharts和WebSocket进行集成。

mysql作用：

1. 在hive中，用mysql存储元信息
   2.存储sparkstreaming的结果，echarts要呈现的东西都是从mysql里取的