1.java中的常用的集合有几种？（set,list,map）除了这几个常用的，还有一个queue（队列）这个 并且set,list,queue的同一个父类是collection，map和collectin是

同一级，但是没有什么联系，是相互独立的。

Queue用于模拟队列这种数据结构。队列通常是指“先进先出（FIFO）”的容器。队列的头部保存在队列中存放时间最长的元素，尾部保存存放时间最短的元素。新元素插入到队列的尾部，取出元素会返回队列头部的元素。通常，队列不允许随机访问队列中的元素。

Set集合：集合元素是**不能重复**的。元素是**没有顺序**的。所以它不能基于位置访问元素。TreeSet和HashSet是它的实现类。

List集合： 集合元素是**可以重复**的。元素是**有顺序**的。所以它可以基于位置访问元素。ArrayList和LinkedList是它的实现类。

Map：它包含键值对。Map的键是不能重复的。Map不能保证存储的顺序。HashMap和TreeMap是它的实现类。

ListedList,LinkedHashSet 数据结构：链表

ArrayList 数据结构：数组

HashSet，HashMap: 数据结构：HashSet是基于HashMap实现，都是基于数组和链表，java8之后是红黑树。

HashMap的get迭代了一个链表，那怎么保证HashMap的时间复杂度o(1)?链表的时间复杂度又是多少？

分四步：

1.判断key，根据key算出索引。

2.根据索引获得索引位置所对应的键值对链表。

3.遍历键值对链表，根据key找到对应的Entry键值对。

4.拿到value。

分析：

以上四步要保证HashMap的时间复杂度O(1)，需要保证每一步都是O(1)，现在看起来就第三步对链表的循环的时间复杂度影响最大，链表查找的时间复杂度为O(n)，与链表长度有关。我们要保证那个链表长度为1，才可以说时间复杂度能满足O(1)。但这么说来只有那个hash算法尽量减少冲突，才能使链表长度尽可能短，理想状态为1。因此可以得出结论：HashMap的查找时间复杂度只有在最理想的情况下才会为O(1)，而要保证这个理想状态不是我们开发者控制的。

数据结构

Set集合： HashSet hash表，存入的对象必须定义hashCode()方法

TreeSet 树结构

LinkedHashSet 链表+hash

List集合： ArrayList 数组

LinkedList 链表

Map集合：HashMap hashCode()进行快速查询,基于散列表实现

LinkedHashMap 和HashMap实现是一样的，用链表维护内部次序

TreeMap 红黑树

比较特殊不常用的集合：

Queue：PriorityQueuePriorityQueue保存队列元素的顺序不是按照元素添加的顺序来保存的，而是在添加元素的时候对元素的大小排序后再保存的。因此在PriorityQueue中使用

peek()或pool()取出队列中头部的元素，取出的不是最先添加的元素，而是最小的元素。

Deque接口是Queue接口的子接口

stack类：

（堆类）是一种后进先出的模式，只能在栈头进行插入与删除操作,继承了Conllecntion

Queue类：

（队列）是一种先进先出的模式，只能在队尾进行插入，在队头进行删除，继承了Conllecntion

Vector类：其实ArrayList非常相似，不同就是Vector是线程安全的，实现锁的机制。

以上集合的时间复杂度的总结：

从上面的时间复杂度可以得出以下总结：

hash的集合的时间复杂度就是：o(1)

Tree的集合的时间复杂度是：olog(N)

集合是否线程安全的总结：

线程安全：Vector，HashTable（jdk1.0引入），

线程不安全：ArrayList，LinkedList，HashMap,TreeMap,TreeSet,HashSet

ConcurrentHashMap应用了锁分段技术，HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程

2.String,StringBuilder和StringBuffer之间的区别

1.String类是不可变的类，每次修改都会生成一个新的类

2.StringBuffer支持并发操作是线程安全的,在java1.0中就已经生成

3.StringBuilder不支持并发操作是线程不安全的 但是在但是在单线程中效率比较高，在jdk1.5中

生成（你可以想成现在的生活都很快，所以后来的StringBuilder就追求了速度）

HashTable和hashMap之间的区别：

1. HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。

2.HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提

供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好

3. 由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。

HashSet和HashMap区别：

TreeSet和TreeMap区别：

1. 最主要的区别就是TreeSet和TreeMap分别实现Set和Map接口

2.TreeSet只存储一个对象，而TreeMap存储两个对象Key和Value（仅仅key对象有序）

3.TreeSet中不能有重复对象，而TreeMap中可以存在

4.TreeMap的底层采用红黑树的实现，完成数据有序的插入，排序。因此它要求一定要有Key比较的方法，要么传入Comparator实现，要么key对象实现Comparable接口。

3.spark中的action算子有哪些？

foreach:(无输出)

saveAsTextFile:(保存到hdfs中,将RDD的每个分区存储为HDFS中一个block)

saveAsObjectFile:(saveAsObjectFile将分区中的每10个元素组成一个Array，然后将这个Array

序列化)

collect：collect相当于toArray，toArray已经过时不推荐使用，collect将分布式的RDD返回为一

个单机的scala Array数组，将结果返回到Driver程序所在的节点，以数组形式存储。

collectAsMap:collectAsMap对（K，V）型的RDD数据返回一个单机HashMap。对于重复K的

RDD元素，后面的元素覆盖前面的元素。数据通过collectAsMap函数返回给

Driver程序计算结果，结果以HashMap形式存储。

reduceByKeyLocally:实现的是先reduce再collectAsMap的功能，先对RDD的整体进行reduce操

作，然后再收集所有结果返回为一个HashMap

lookUp: Lookup函数对（Key，Value）型的RDD操作，返回指定Key对应的元素形成的Seq。

这个函数处理优化的部分在于，如果这个RDD包含分区器，则只会对应处理K所在的分

区，然后返回由（K，V）形成的Seq。如果RDD不包含分区器，则需要对全RDD元素进

行暴力扫描处理，搜索指定K对应的元素。

count：返回整个RDD的元素个数。

top： 可返回最大的k个元素。

take: 可返回最小的K个元素

takeOrdered:返回最小的K个元素，并且在返回的数组中保持元素的顺序

first：相当于top(1)返回整个RDD中的前K个元素，可以定义排序方式Ordering[T].返回的是一个

含前K个元素的数组

reduce：函数相当于对RDD中的元素进行reduceLeft函数的操作。

reduceLeft先对两个元素

fold：和reduce的原理相同，但是与reduce不同，相当于每个reduce时，迭代器取的第一个元素

是zeroValue。

aggregata:aggregate先对每个分区的所有元素进行aggregate操作，再对分区的结果进行fold操

作，**aggreagate与fold和reduce的不同之处在于，aggregate相当于采用归并的方**

**式进行数据聚集，这种聚集是并行化的。 而在fold和reduce函数的运算过程中，每**

**个分区中需要进行串行处理，每个分区串行计算完结果，结果再按之前的方式进行聚**

**集，并返回最终聚集结果。**

4.NameNode和secondNameNode:

NameNode:主要用来保存HDFS信息，比如命名空间信息，块信息等等，当它运行的时候，这

些信息是存在内存中。但是这些信息也可以持久化到磁盘上

fsimage:它是namenode启动时对整个文件系统的快照

edits: 它是在NameNode启动后，对文件系统的改动序列

只有NameNode重启的时候，edits才会合并到fsimage文件中，从而得到一个文件系统的最新快照。

但是生产环境中NameNode是很少启动的，这意味着当NameNode运行很长时间后，edits文件会变得很大：(1.如何管理,2.重启花费很长时间去合并，3.如果NameNode挡掉了，那我们就丢失了很多改动):为了修改这个问题那么SencondaryNameNode

1. 首先，它定时到NameNode去获取edits，并更新到fsimage上。

2.一旦它有新的fsimage文件，它将其拷贝回NameNode上。

3.NameNode在下次重启时回使用这个新的fsimage文件，从而减少重启的时间。

Secondary NameNode的检查点进程启动，是由两个参数控制：

1.fs.checkpoint.period，指定连续两次检查点的最大时间间隔， 默认值是1小时

2.fs.checkpoint.size定义了edits日志文件的最大值，一旦超过这个值会导致强制执行检查点（即

使没到检查点 的最大时间间隔）。默认值是64MB。

注意：关于NameNode是什么时候将改动写到edit logs中的？这个操作实际上是由DataNode的

写操作触发的，当我们往DataNode写文件时，DataNode会跟NameNode通信，告诉

NameNode什么文件的第几个block放在它那里，NameNode这个时候会将这些元数据信息

写到edit logs文件中。

5.Hadoop HA的原理

active和standly模式

6.Spark中Driver的作用：

Driver作为应用程序的总控，负责发送任务以及监控任务的运行状态->executor会把执行任务

发送给

Driver线程主要是初始化SparkContext对象，准备所需要的上下文，一方面保持与ApplicationMaster的RPC连接，通过ApplicationMaster申请资源，另一方面根据用户业务逻辑开始 调度任务，将任务下发到已有的空闲的Executor上，当ResourceManager向ApplicationMaster返回Container资源时，ApplicationMaster就尝试在对应的Container上启动Executor进程，Executor进程起来之后，会向Driver注册，注册成功后保持与Driver的心跳，同时等待Driver分发任务，当分发的任务执行完毕后，将任务状态上报给Driver。*Driver把资源申请的逻辑抽象出来，以适配不同的资源管理[系统](https://www.2cto.com/os/)，所以才间接通过ApplicationMaster去和Yarn打交道。*

驱动节点（Driver）：

Driver在spark作业中执行时主要负责以下操作：

1）把用户程序转化任务：

Driver程序负责把用户程序转为多个物理单元，这些物理单元被称为任务。从上层来看，

spark程序的流程是这样的：读取或者转化数据创建一系列的RDD，然后使用转化操作生成新

的RDD，最后使用行动操作得到结果或者将数据存储到文件存储系统中。spark其中隐士创建

了一个由上述操作组成的逻辑上的有向无环图。当Driver运行时，它会把这个逻辑图转化为物

理执行计划。

Spark 会对逻辑执行计划作一些优化，比如将连续的映射转为流水线化执行，将多个操作合并

到一个步骤中等。这样 Spark 就把逻辑计划转为一系列步骤（stage）。而每个stage又由多

个task组成。这些task会被打包并送到集群中。task是 Spark 中最小的执行单元，用户程序

通常要启动成百上千的独立任务。

2）为执行器节点调度任务

Driver程序会根据当前的Executor节点集合，尝试把所有Task基于数据所在位置分配给合适的

Executor进程。当Task执行时，Executor进程会把缓存数据存储起来，而Driver进程同样会跟

踪这些缓存数据的位置，并且利用这些位置信息来调度以后的任务，以尽量减少数据的网络输。

3）跟踪Executor的运行状态：

有了物理执行计划之后，Driver程序必须在各个Executor进程间协调任务的调度。Executor进

程启动后，会向Driver进程注册自己。因此，Driver进程就可以跟踪应用中所有的Executor节

点的运行信息。

7.执行器节点（Executor）

Spark Executor节点是一个工作进程，负责在 Spark 作业中运行任务，任务间相互独立。Spark 应用启动时，Executor节点被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有Executor节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他Executor节点上继续运行。

执行器进程有两大作用：

1、它们负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程；

2、它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在Executor进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

执行器程序通常都运行在专用的进程中。

8.Job是以Action为界限，遇到一个Action方法则触发一个job

DAGScheduler负责Stage级的调度，stage的划分是根据款依赖和窄依赖划分的

TaskSheduler负责Task级的调度

SchedulerBackend负责提供可用资源

9.redis支持的数据类型有哪些？

1 、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存

储。

      2 、Redis支持数据的备份，即master-slave模式的数据备份。

     3 、Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行

使用。

**1) :String（字符串）**

* string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。
* string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。
* string类型是Redis最基本的数据类型，一个键最大能存储512MB。

**2) Hash（哈希）**

* Redis hash 是一个键值对集合。
* Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。类似于java里面的Map<String,Object>

**3) List（列表）**

* Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

**4) Set（集合）**

Redis的Set是string类型的无序集合,是通过HashTable实现的

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

**5) zset(sorted set：有序集合)**

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

10.hive和MySQL数据库有什么不同？

①数据库可以用在Online的应用中，Hive主要进行离线的大数据分析；

②数据库的查询语句为SQL，Hive的查询语句为HQL；

③数据库数据存储在LocalFS，Hive的数据存储在HDFS；

④Hive执行引擎是MapReduce，MySQL有自己的执行引擎；

⑤Hive没有索引；

⑥Hive延迟性高；

⑦Hive可扩展性高；

⑧Hive数据规模大；

11.spark中RDD,DateFrame和DataSet之间的区别？

<1> DataFrame是spark1.3.0版本提出来的，spark1.6.0版本又引入了DateSet的，但是在spark2.0版本中，DataFrame和DataSet合并为 DataSet。首先，DataFrame和DataSet是基于RDDs的，而且这三者之间可以通过简单的API调用进行无缝切换。

<2>RDD缺点：

对于结构化数据处理对于sql来说比较麻烦，默认采用的java的序列化方式，序列化结构比较大，而且数据存储在java内存中，导致 GC比较频繁

<3> DataFrame的优点：

A.结构化数据处理非常方便，支持Avro, CSV, elastic search, and Cassandra等kv数据，也支持HIVE tables, MySQL等传统数据表

B.有针对性的优化,如采用Kryo序列化，由于数据结构元信息spark已经保存，序列化时不需要带上元信息，大大的减少了序列化大小，而且数

据保存在堆 外内存中,减少了gc次数,所以运行更快。

C.hive兼容，支持hql，udf等

<4>DataSet:

A.DataSet整合了RDD和DataFrame的优点，支持结构化和非结构化的数据

B.和RDD一样，支持自定义对象的存储

C.和DataFrame一样，支持结构化数据的sql查询

D.采用了堆外内存，GC友好

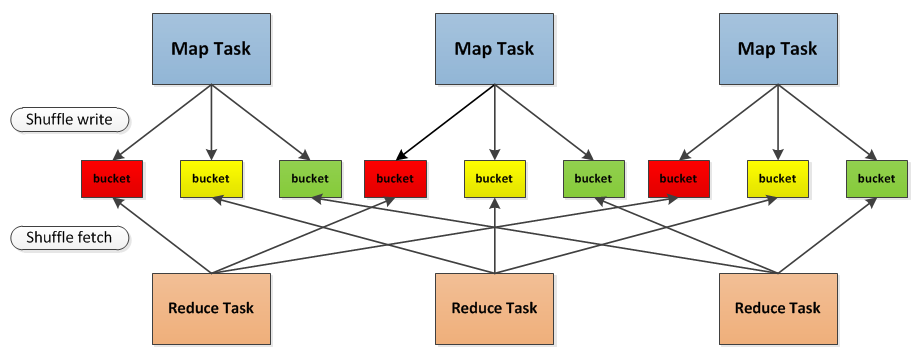
自Spark2.0之后，DataFrame和DataSet合并为更高级的DataSet，新的DataSet具有两个不同的API特性：

1.非强类型(untyped)，DataSet[Row]是泛型对象的集合，它的别名是DataFrame；

2.强类型(strongly-typed)，DataSet[T]是具体对象的集合，如scala和java中定义的类

12.spark中shuffle的整个过程详解？

1.spark中有shuffle writer有4种方式：



* 首先每一个Mapper会根据Reducer的数量创建出相应的bucket，bucket的数量是M×RM×R，其中MM是Map的个数，RR是Reduce的个数。
* 其次Mapper产生的结果会根据设置的partition算法填充到每个bucket中去。这里的partition算法是可以自定义的，当然默认的算法是根据key哈希到不同的bucket中去。
* 当Reducer启动时，它会根据自己task的id和所依赖的Mapper的id从远端或是本地的block manager中取得相应的bucket作为Reducer的输入进行处理。

这里的bucket是一个抽象概念，在实现中每个bucket可以对应一个文件，可以对应文件的一部分或是其他等。

shuffleMaptask会封装成一个叫mapStatus,这个mapstatus,里面包含了每一个resultTask拉取数据的大小

每一个resultTask拉取过来的数据，就会在内部形成一个rdd,这个rdd叫做shuffleRdd,这个rdd的数据优先存放到内存中，内存中不够然后存到磁盘里

我们把这一组的shuffle文件称为shuffleGroup,每个文件中都存储了很多shuffleMapTask对应的数据，这个文件叫做segment,这个时候因为不同的shuffleMapTask都是存在一个文件中

所以建立索引文件，来标记shuffleMapTask在shuffleBlockFile的位置+偏移量，这样就可以在一个文件里面把不同的shuffleMaptask数据分出来

当前次map还没执行完，ShuffleFileGroup还没有释放，这时如果有新的map在这个节点上执行，无法循环利用这个ShuffleFileGroup，而是只能创建新的bucket文件组成新的ShuffleFileGroup来写输出。

读取数据的过程：

有一个类blockManager，封装了临时文件的位置信息,resultTask先通过blockManager,就知道我从哪个节点拿数据

如果是远程，它就是发起一次socket请求，创建一个socket链接。然后发起一次远程调用，告诉远程的读取程序，读取哪些数据。读到的内容再通过socket传过来。

可以这样理解分区partition是RDD存储数据的地方，实际是个逻辑单位，真正要取数据时，它就调用BlockManage去读，它是以数据块的方式来读。

比如一次读取32k还是64k。它不是一条一条读，一条一条读肯定性能低。它读时首先是看本地还是远程，如果是本地就直接读这个文件了

spark有2个Task一个是shuffleMapTask和ResultTask，最终的action执行所产生的就是resultTask

Shuffle read指的是reducer对属于自己的FileSegment文件进行fetch操作，这里采用的netty框架，效率明显好于Mapreduce的http传输。fetch操作会等到所有的Shuffle Write过程结束后再进行，这也是因为ShuffleMapTask可能并不在一个stage里面，需要在父stage执行之后提交才会进行子stage的执行。reducer通过fetch得到的FileSegment先放在缓冲区softBuffer中，默认大小48MB。Spark不要求Shuffle后的数据是全局有序的，所以没有必要等到shuffle read全部结束后再进行reduce，是可以并行处理的

mapper运行后，通过Partitioner接口，根据key或value及reduce的数量来决定当前map的输出数据最终应该交由哪个reduce task处理

在Spark的中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。

ShuffleManager随着Spark的发展有两种实现的方式，分别为HashShuffleManager和SortShuffleManager，因此spark的Shuffle有Hash Shuffle和Sort Shuffle两种 按照是否优化又可以分为4种

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。

从Task 开始那边各自把自己进行 Hash 计算(分区器：hash/numreduce取模)，分类出3个不同的类别，每个 Task 都分成3种类别的数据，想把不同的数据汇聚然后计算出最终的结果，所以Reducer 会在每个 Task 中把属于自己类别的数据收集过来，汇聚成一个同类别的大集合，

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。

注意：

1).buffer起到的是缓存作用，缓存能够加速写磁盘，提高计算的效率,buffer的默认大小32k。

分区器：根据hash/numRedcue取模决定数据由几个Reduce处理，也决定了写入几个buffer中

block file：磁盘小文件，从图中我们可以知道磁盘小文件的个数计算公式：

block file=M\*R

在同一个进程中，无论是有多少过Task，都会把同样的Key放在同一个Buffer里，然后把Buffer中的数据写入以Core数量为单位的本地文件中，(一个Core只有一种类型的Key的数据)，每1个Task所在的进程中，分别写入共同进程中的3份本地文件，这里有4个Mapper Tasks，所以总共输出是 2个Cores x 3个分类文件 = 6个本地小文件。

（写的机制有以下三种：）

map-side aggregation Partition数(RDD) Serializer支持relocation

BypassMergeSortShuffleWriter 否 小于200(默认) -

UnsafeShuffleWriter 否 小于16777216 是

SortShuffleWriter - - -

1.没有map端聚合操作，且RDD的Partition数小于200，使用BypassMergeSortShuffleWriter。

2.没有map端聚合操作，且RDD的Partition数小于16777216，且Serializer支持relocation，使用UnsafeShuffleWriter。

3.上述条件都不满足，使用SortShuffleWriter。

上面提到UnsafeShuffleWriter需要Serializer支持relocation，Serializer支持relocation是指，Serializer可以对已经序列化的对象进行排序，这种排序起到的效果和先对数据排序再序列化一致

支持relocation的Serializer是KryoSerializer，Spark默认使用JavaSerializer，通过参数spark.serializer设置。

选择哪种ShuffleWriter实现是运行时动态选择的，因此，设置参数spark.shuffle.manager为sort或tungsten-sort没有区别。

接下来分析下为什么要满足这三个要求

map-side aggregation：从上面的实现也可以看出，UnsafeShuffleWriter不是类似HashMap的数据结构，无法聚合key对应的value，所以无法支持map端的aggregation。

Partition数小于16777216：参考第一幅图，存储PartitionId信息使用24bit，能表示的最大值为 (1 << 24) = 16777215，因此Partition数要小于16777216。

Serializer支持relocation：原始数据首先被序列化处理，并且再也不需要反序列，在其对应的元数据被排序后，需要Serializer支持relocation，在指定位置读取对应数据。

**<1>Sort shuffle的普通机制：**

图片的地址：

<https://img-blog.csdn.net/20171205202808068?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VpeGluXzM1NjAyNzQ4/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center>

a) map task 的计算结果会写入到一个内存数据结构里面，内存数据结构默认是5M

       b) 在shuffle的时候会有一个定时器，不定期的去估算这个内存结构的大小，当内存结构中的数据超过5M时，比如现在内存结构中的数据为5.01M，那么他会申请5.01\*2-5=5.02M内存给内存数据结构。

       c) 如果申请成功不会进行溢写，如果申请不成功，这时候会发生溢写磁盘。

       d) 在溢写之前内存结构中的数据会进行排序分区

       e) 然后开始溢写磁盘，写磁盘是以batch的形式去写，一个batch是1万条数据，

       f) map task执行完成后，会将这些磁盘小文件合并成一个大的磁盘文件（有序），同时生成一个索引文件。

      g) reduce task去map端拉取数据的时候，首先解析索引文件，根据索引文件再去拉取对应的数据。

普通的SortShuffleManager的原理。在该模式下，数据会先写入一个内存数据结构中(默认5M)，此时根据不同的shuffle算子，可能选用不同的数据结构。如果是reduceByKey这种聚合类的shuffle算子，那么会选用Map数据结构，一边通过Map进行聚合，一边写入内存；如果是join这种普通的shuffle算子，那么会选用Array数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

注意：

shuffle中的定时器：定时器会检查内存数据结构的大小，如果内存数据结构空间不够，那么会申请额外的内存，申请的大小满足如下公式：

applyMemory=nowMenory\*2-oldMemory

申请的内存=当前的内存情况\*2-上一次的内嵌情况

意思就是说内存数据结构的大小的动态变化，如果存储的数据超出内存数据结构的大小，将申请内存数据结构存储的数据\*2-内存数据结构的设定值的内存大小空间。申请到了，内存数据结构的大小变大，内存不够，申请不到，则发生溢写

**排序**

在溢写到磁盘文件之前，会先根据key对内存数据结构中已有的数据进行排序。

溢写

排序过后，会分批将数据写入磁盘文件。默认的batch数量是10000条，也就是说，排序好的数据，会以每批1万条数据的形式分批写入磁盘文件。写入磁盘文件是通过Java的BufferedOutputStream实现的。BufferedOutputStream是Java的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘IO次数，提升性能。

注意：

1）block file= 2M

一个map task会产生一个索引文件和一个数据大文件

2） m\*r>2m（r>2）：SortShuffle会使得磁盘小文件的个数再次的减少

<https://img-blog.csdn.net/20171205202822210?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VpeGluXzM1NjAyNzQ4/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center>

那么遇到Shuffle，spark是怎么进行文件定位和数据拉取的呢？这里我们要介绍mapOutputTracker和BlockManager两个组件，他们负责追踪磁盘小文件的地址和管理块文件

<https://img-blog.csdn.net/20171205204450785?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VpeGluXzM1NjAyNzQ4/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center>

spark是怎么进行文件定位？

mapOutputTracker组件：管理磁盘小文件的地址

主从架构：

主：MapOutputTrackerMaster---存在Driver进程中

从：MapOutputTrackerWorker---存在Executor进程中

map task执行完毕后，会将map task的执行的情况和小文件的地址封装到一个mapStatus对象中，调用MapoutputTrackerWorker向Driver中的MapOutputTrackerMaster发送消息，这样MapOutputTrackerMaster就掌握了磁盘小文件的位置信息redcue在执行之前会和MapOutputTrackerMaster发送信息获取磁盘小文件的位置，这样reduce就能到数据所在的节点拉取数据，拉取数据要用到BlockManager中的BlockTransforService，BlockTransforService，启动5个线程来拉取数据（拉取的数据不超过48M，如果失败重试3次），将其存放在聚合内存中。

注意：

reduce OOM错误排解：

1：拉取的数据过多，导致聚合的过程中的内存无法满足

解决方案：减少每次拉取的数据量

提高Shuffle聚合的内存比例

提高Executor的总内存

spark怎么拉取数据？

BlockManager组件：块管理者

主从架构：

主：BlockManagerMaster：存在Driver中

1：DiskStore：负责磁盘的管理

2：MemStore：负责内存的管理

3：ConnectionManager：负责连接其他的BlockManagerSlave

4：BlockTransforService ：负责数据的传输

BlockManagerMaster还管理着RDD的缓存数据，持久化和清除持久化

从：BlockManagerSlave：存在Executor中

1：DiskStore：负责磁盘的管理

2：MemStore：负责内存的管理

3：ConnectionManager：负责连接其他的BlockManagerSlave

4：BlockTransforService ：负责数据的传输

**Spark1.2之前是HashShuffle默认的分区器是HashPartitioner，Spark1.2引入SortShuffle默认的分区器是RangePartitioner。**

**<2>bypass机制（比如wordcount)不需要排序时使用**

* bypass机制示意图

<https://images2017.cnblogs.com/blog/1250469/201802/1250469-20180207014819841-1840545349.png>

总结

          a) bypass运行机制的触发条件如下：

                shuffle reduce task的数量小于spark.shuffle.sort.bypassMergeThreshold的参数值。这个值默认是200。

          b）产生的磁盘小文件为：2\*M（map task的个数）

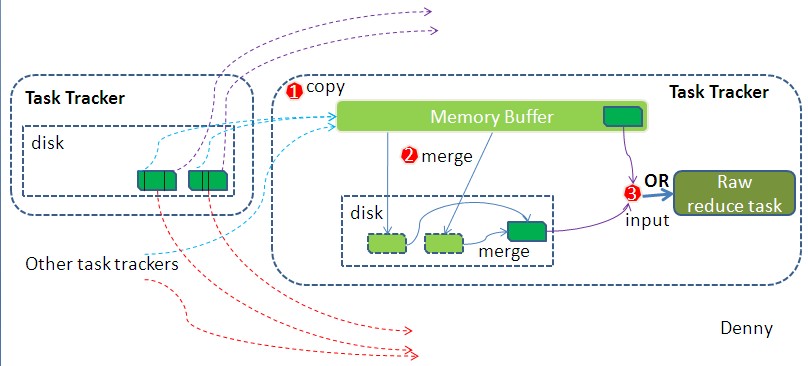
15.HashPartition(哈希分区)原理解析：

HashPartitioner采用哈希的方式对<Key，Value>键值对数据进行分区。其数据分区规则为 partitionId = Key.hashCode % numPartitions，其中partitionId代表该Key对应的键值对数据应当分配到的Partition标识，Key.hashCode表示该Key的哈希值，numPartitions表示包含的Partition个数。图3简单描述了HashPartitioner的数据分区过程。

一个Partition对应一个Task

13.Hadoop的shuffle过程详解：（必须写磁盘）





1.map的执行结果会被Outercollector组件收集，outercollector会将数据写入环形缓冲区内，

进行数据写入的时候根据map输出的key生成一个分区号，默认的是key.hashCode()&Integer\_MAX\_VALUE%reducetask获取分区号。（这个是由reduce的个数来确定partition的分区号）

如果用户自定义分区算法，则按照用户自定义的分区进行返回。

2.环形缓冲区默认大小为100M，阈值为80%，当达到阈值的时候会开启一个溢写线程，将环形缓冲区的数据向磁盘溢写，溢写之前需要做一些

工作溢写之前会会环形缓冲区内的数据进行排序，先根据分区号进行排序，这个排序内部是双层 for循环,作用是将相同分区的数据写在一

起，再根据map输出的key值进行排序，作用是将key相同的数据放在一起（写磁盘前，要进行partition、sort和combine等操作）

3.溢写到磁盘，形成一个个溢写文件，溢写的时候会根据元数据到对应的原始数据中获取原始数据进行溢写。一个mapTask会对应多个溢写

文件

4.对溢写文件进行归并，在归并的时候会再进行一次排序，排完序之后的文件就会按分区和map的key放在一起，并且分区之间会有标记

5.reduceTask会通过MRAppmaster获取mapTask的执行进度，当获取MapTask执行完成的时候就会开启多线程对map的输出文件进行拷 贝。拷贝的时候是按分区进行拷贝的。对应的ReduceTask会将对应的Maptask的输出文件的对应分区的文件进行拷贝

6.由于ReduceTask要针对所有的MapTask的输出文件，所以拷贝过来的文件是所有mapTask的输出的对应分区的文件，将从各个mapTask拷贝过来的对应分区文件进行文件的归并和排序，copy过来的数据会

先放入内存缓冲区中，这里的缓冲区大小要比map端的更灵活，它基于jvm的heap size设置，因为shuffle过程的中，reducer不运行，所以应该把绝大部分的内存都给Shuffle用.当内存中的数据量到达一定阈 值，就启动内存到磁盘的merge。与map 端类似，这也是spill的过程，这个过程中如果你设置有Combiner，也是会启用的，然后在磁盘中生成了众多的spill文件，第二种merge方式一直在运行，直到没有map端 的数据时才结束，然后启动第三种磁盘到磁盘的merge方式生成最终的那个文件。 Reducer的输入文件，不断地merge后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对我们来说，当然希望它存放于内存中，直接作为Reducer的输入，但默认情况下，这个文件是存放于磁盘中的，当Reducer的输入文件已定，整个Shuffle才最终结束，然后就是Reducer执行，把结果放到HDFS上。

7.归并完成后会调用GroupComoparator进行分组读取给自定义reduce函数，所以reduce函数拿到的就是一组组的数据（key相同的为一组）

8按照自己定义的reduce逻辑进行执行

9最终结果输出到hdfs

Map任务总要把输出的数据写到磁盘上，即使输出数据量很小在内存中全部能装得下，在最后也会把数据刷到磁盘上。

Reduce拖过来的数据会放在一个HashMap中，HashMap中存储的也是<key, value>对，key是Map输出的key，Map输出对应这个key的所有value组成HashMap的value

15.spark的shuffle和Hadoop的shuffle之间有什么区别？

MapReduce它必须将所有的数据都写入本地磁盘文件以后，才能启动reduce操作，拉取数据，因为mapreduce要实现默认的根据key的排序！所以要排序，肯定得写完所有数据，才能排序，然后reduce来拉取。但是Spark不需要，spark默认情况下，是不会对数据进行排序的。因此ShuffleMapTask每写入一点数据，ResultTask就可以拉取一点数据，然后在本地执行我们定义的聚合函数和算子，进行计算。  spark这种机制的好处在于，速度比mapreduce快多了。但是也有一个问题，mapreduce提供的reduce，是可以处理每个key对应的value上的，很方便。但是spark中，由于这种实时拉取的机制，因此提供不了，直接处理key对应的values的算子，只能通过groupByKey，先shuffle，有一个MapPartitionsRDD，然后用map算子，来处理每个key对应的values。就没有mapreduce的计算模型那么方便。

|  |  |  |
| --- | --- | --- |
| MapReduce | Spark |  |
| collect | 在内存中构造了一块数据结构用于map输出的缓冲 | 没有在内存中构造一块数据结构用于map输出的缓冲，而是直接把输出写到磁盘文件 |
| sort | map输出的数据有排序 | map输出的数据没有排序 |
| merge | 对磁盘上的多个spill文件最后进行合并成一个输出文件 | 在map端没有merge过程，在输出时直接是对应一个reduce的数据写到一个文件中，这些文件同时存在并发写，最后不需要合并成一个 |
| copy框架 | jetty | netty或者直接socket流 |
| 对于本节点上的文件 | 仍然是通过网络框架拖取数据 | 不通过网络框架，对于在本节点上的map输出文件，采用本地读取的方式 |
| copy过来的数据存放位置 | 先放在内存，内存放不下时写到磁盘 | 一种方式全部放在内存； 另一种方式先放在内存 |
| merge sort | 最后会对磁盘文件和内存中的数据进行合并排序 | 对于采用另一种方式时也会有合并排序的过程 |

14.scala中class和object之间的关系？

1.在Scala中,类名可以和对象名为同一个名字,该对象称为该类的伴生对象,类和伴生对象可以相互访问他们的私有属性,但是它们必须在同一个源 文件中

2.类只会被编译,不能直接执行,类的声明和主构造器在一起被声明,在一个类中,主构造器只有一个.

3.类和它的伴生对象可以相互访问其私有成员

4.class和object的一个差别是,单例对象不带参数,而类可以.因为你不能用new关键字实例化一个单例对象,你没有机会传递给它参数

5.可以理解为Scala把类中的static集中放到了object对象中,伴生对象和类文件必须是同一个源文件,可以用伴生对象做一些初始化操作.

calss scala编译器会字段帮我们生产一个私有字段和2个公有方法get和set

scala 中没有 static 关键字，所以 对于一个class来说，所有的方法和成员变量在实例被 new 出来之前都是无法访问的

因此在class中的main方法没什么用了

scala 的object 中所有成员变量和方法默认都是 static 的

所以可以直接访问main方法

15.Hadoop最好不能存储小文件为什么？

1.而这受到NameNode物理内存容量的限制

2.处理大量小文件速度远远小于处理同等大小的大文件的速度。每一个小文件要占用一个slot，而task启动将耗费大量时间甚至大部分时间都耗费在启动task和释放task上。

3.处理小文件并非Hadoop的设计目标，HDFS的设计目标是流式访问大数据集（TB级别）。因而，在HDFS中存储大量小文件是很低效的。访问大量小文件经常会导致大量的寻找，以及不断的从一个DatanNde跳到另一个DataNode去检索小文件，这都不是一个很有效的访问模式，严重影响性能。

4.在MapReduce中，一个map task就会处理一个block块，就会启动很多的map task

16.HIVE表中的内部表和外部表之间有什么区别？

1、在导入数据到外部表，数据并没有移动到自己的数据仓库目录下，也就是说外部表中的数据并不是由它自己来管理的！而表则不一样；

2、在删除表的时候，Hive将会把属于表的元数据和数据全部删掉；而删除外部表的时候，Hive仅仅删除外部表的元数据，数据是不会删除的！

那么，应该如何选择使用哪种表呢？在大多数情况没有太多的区别，因此选择只是个人喜好的问题。但是作为一个经验，如果所有处理都需要由Hive完成，那么你应该创建表，否则使用外部表！

**17.HIve的数据存储个格式有哪些？**

hive文件存储格式包括以下几类：

* TEXTFILE
* SEQUENCEFILE
* RCFILE
* Parquet
* 自定义格式

**1.** textfile:默认格式，数据不做压缩，磁盘开销大，数据解析开销大可结合Gzip、Bzip2使用（系统自动检查，执行查询时自动解压），但使用 这 种方式，hive不会对数据进行切分，从而无法对数据进行并行操作。

2.sequencefile:

SequenceFile是Hadoop API提供的一种二进制文件支持，其具有使用方便、可分割、可压缩的特点。

SequenceFile支持三种压缩选择：NONE, RECORD, BLOCK。 Record压缩率低，一般建议使用BLOCK压缩。

3.RCFILE

RCFILE是一种行列存储相结合的存储方式。首先，其将数据按行分块，保证同一个record在一个块上，避免读一个记录需要读取多个block。其次，块数据列式存储，有利于数据压缩和快速的列存取。RCFILE文件示例：

4.自定义格式

当用户的数据文件格式不能被当前 Hive 所识别的时候，可以自定义文件格式。

用户可以通过实现inputformat和outputformat来自定义输入输出格式，参考代码：

.\hive-0.8.1\src\contrib\src\java\org\apache\hadoop\hive\contrib\fileformat\base64

18.kafka中partition的分配？

《1》kafka中partition的设计优点是什么？

1.增强扩展能力，让其可以处理任意多的数据

2.作为并行单元，提高并行能力，吞吐量高

3.分区多的话打开的文件句柄也会多, partition最好的个数是consumer的倍数

4.在默认情况下，两个boker只有一个线程负责数据的复制。分区多的话，这个线程复制会是一个限制

5.新的producer可以允许用户设置一个缓冲区，缓冲一定量的数据，当缓冲区数据到达设定量或者到时间，数据会从缓存区删除发往broker。如果分区很多，每个分区都缓存一定

量的数据量在缓冲区，很可能会占用大量的内存，甚至超过[系统内](https://www.baidu.com/s?wd=%E7%B3%BB%E7%BB%9F%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)

6.Consumer也存在同样的问题，会从每个分区拉一批数据回来，分区越多，所需内存也就越大。

7.根据经验，应该给每个分区分配至少几十KB的内存

《2》kafka的分配：

1个partition只能被同组的一个consumer消费，同组的consumer则起到均衡效果。当就一个消费组中消费者数量多于parition的数量

的时候，多余的消费者空闲。当一个消费组中消费者数量少于partition的数量的时候，多个partition对应一个消费者。

当有多个消费组的时候：

消息被g3组的消费者均分，g4组的消费者在接收到了所有的消息。

g3组：

C1接收到了：2，5，8

C2接收到了：3，6，9

C3接收到了：1，4，7

g4组：

C1接收到了：1，2，3，4，5，6，7，8，9

**启动多个组，则会使同一个消息被消费多次**

partition在每个group中消费的offset 新版kafka把这个offsert保存到了一个\_\_consumer\_offsert的topic下 这个\_\_consumer\_offsert 有50个分区，通过将group的id哈希值%50的值来确定要保存到那一个分区.  这样也是为了考虑到zookeeper不擅长大量读写的原因。一个group用几个consumer来同时读取的话，需要多线程来读取，一个线程相当于一个consumer实例

Topic在逻辑上可以被认为是一个queue，每条消费都必须指定它的Topic，可以简单理解为必须指明把这条消息放进哪个queue里。为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，每个Partition在物理上对应一个文件夹，该文件夹下存储这个Partition的所有消息和索引文件

对于传统的message queue而言，一般会删除已经被消费的消息，而Kafka集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此Kafka提供两种策略删除旧数据。一是基于时间，二是基于Partition文件大小。

这个offset由Consumer控制。正常情况下Consumer会在消费完一条消息后递增该offset。当然，Consumer也可将offset设成一个较小的值，重新消费一些消息。因为offet由Consumer控制，所以Kafka broker是无状态的，它不需要标记哪些消息被哪些消费过，也不需要通过broker去保证同一个Consumer Group只有一个Consumer能消费某一条消息，因此也就不需要锁机制，这也为Kafka的高吞吐率提供了有力保障。

你可以把 topic 理解为文件夹，partition 为 topic 下面的子文件夹，log 在 partition 下，而消息保存在 log 中。

每一个分区都有一个broker为leader，它负责该分区内的所有读写操作，其他broker 被动的复制leader broker。如果leader broker 挂了，那么其他broker中的一个将自动成为该分区的新leader。

Topic:topic1 PartitionCount:3 ReplicationFactor:2 Configs:

Topic: topic1 Partition: 0 Leader: 0 Replicas: 0,2 Isr: 0,2

Topic: topic1 Partition: 1 Leader: 1 Replicas: 1,0 Isr: 1,0

Topic: topic1 Partition: 2 Leader: 2 Replicas: 2,1 Isr: 2,1

每一个分区都有一个broker为leader，它负责该分区内的所有读写操作，其他broker 被动的复制leader broker。如果leader broker 挂了，那么其他broker中的一个将自动成为该分区的新leader。

Replicas: 0,1 表示该分区保存在 broker 0 和 broker 1下。

Isr: 0,1 表示当前我们可以在 broker 0 和 broker 1 下访问该分区，如果 broker 0挂了，那就像这样子了 Isr: 1

19.既然offset是以partiton作为单位存储的，那么当一个consumer监听多个partiton的时候，consumer如何知道自己该去哪个partiton拉数据呢？

这里 kafka 提供两种分配策略 range和roundrobin，由参数partition.assignment.strategy指定，默认是range策略。本文只讨论range策 略。所谓的range其实就是按照阶段平均分配。

20.如何启动消费者组：

1.bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test2 --from-beginning --consumer.config config/consumer\_g4.properties --delete-consumer-offsets

1.bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test2 --from-beginning --consumer.config config/consumer\_g3.properties

2.bin/kafka-console-consumer.sh -- bootstrap-server localhost:9092 --topic test2 --from-beginning --consumer.config config/consumer\_g3.properties

3.bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test2 --from-beginning --consumer.config config/consumer\_g3.properties

颜色之间的查别就是他们一个属于g3组，一个属于g4组，g4组中只有一个消费者，g3组中有三个消费者

21.一个kafka broker可以同时启动多个topic吗？

每条发布到 Kafka 集群的消息都有一个类别，这个类别被称为 Topic。（物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 broker 上，但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存于何处。支持的数据格式包括 String、JSON、Avro

topic创建的过程：（命令行部分）

《1》当我们发起命令的时候主要有两件事情，确定分区副本的分配方案（就是每个分区的副本都分配到哪些broker上）

《2》创建zookpeer节点，把这个方案写入/broker/topic/<topic>节点下

后天逻辑部分：1.创建分区，2。创建副本 3。为每个分区选举leader，ISR 4.更新各种缓存

值得一提的是，0.10版本支持指定broker的机架信息，类似于Hadoop那样，可以更好地利用局部性原理减少集群中网络开销。如果指定了机架信息(broker.rack), Kafka在为分区做 副本分配时就会考虑这部分信息，尽可能地为副本挑选不同机架的broker

默认的副本数是1

partition最好的个数是consumer的倍数

20.java中多线程的实现方式有哪些？

1、继承Thread类创建线程 实现其中的run方法，并用.start()开始一个线程

2、实现Runnable接口创建线程 如果自己的类已经extends另一个类，就无法直接extends Thread，此时，可以实现一个Runnable接口，

3.实现Callable接口通过FutureTask包装器来创建Thread线程

4.使用ExecutorService、Callable、Future实现有返回结果的线程

21.chche（）和persist（）之间的区别？

<1>cache 底层调用的是 persist 方法，存储等级为: memory only，persist 的默认存储级别也是 memory only，persist 与 cache 的主要区别是persist 可以自定义存储级别

<2> cache 和 checkpoint 之间有一个重大的区别，cache 将 RDD 以及 RDD 的血统(记录了这个RDD如何产生)缓存到内存中，当缓存的 RDD失效的时候(如内存损坏)，它们可以通过血统重新计算来进行恢复。但是 checkpoint 将 RDD 缓存到了 HDFS 中，同时忽略了它的血统(也就是RDD之前的那些依赖)。为什么要丢掉依赖？因为可以利用 HDFS 多副本特性保证容错！

在进行checkpoint之前要设置 sc.setCheckpointDir("...")。除此之外，在进行 checkpoint 前，要先对 RDD 进行 cache。Why???

checkpoint 会等到 job 结束后另外启动专门的 job 去完成 checkpoint，也就是说需要 checkpoint 的 RDD 会被计算两次。

<3> persist与checkpoint的区别

rdd.persist(StorageLevel.DISK\_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉，是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。

22.yarn资源调度机制？

yarn调度的主要分为8个步骤：

1.yarnClient提交program信息并打拼ResourceManager,包括(应用代码和应用需要的一切参数和环境信息)

2.ResourceManager收到请求之后，调用ApplicationMasterManager向NodeManager发送请求，申请一个资源（Container），并且要求Container启动ApplicationMaster.

3.ApplicationMaster启动之后，首先注册自己到ResourceManager，然后为自己的Task申请Container，这个过程是轮训

的，循环申请资源，ResourceManager收到请求之后，会要求NodeManager分配资源

4.资源分配完毕之后，Application Master发送请求到NodeManager，启动任务。

5.NodeManager设置Container的运行时环境（jar包，环境变量，任务启动脚本），NodeManager会通过脚本启动任务

6.启动的过程是由NodeManager的ContainerLauncher负责的，ContainerLauncher完成启动任务的工作

7.第一，task和Application Master的交互，task会向AM汇报任务状态和进度信息，比如任务启动，停止，状态更新。Applicaiton Master利

用这些信息监控task整个执行过程。第二，是NodeManager和ResourceManager的交互，这个过程和任务没有关系，主要是两者之间保持

的心跳信息（状态的变化等等）

8.Application Master在检测到作业运行完毕之后，Application Master想Resource Manager 删除自己，并且停止自己执行。

23. yarn的资源调度策略：

（1）yarn的资源抽象比较简单，只有两种资源：内存和CPU。其中内存资源比较关键，直接决定了任务是否能够成功。很可能会发生内存溢出的现象

(2) container之间实现了资源隔离，理论上来说不同的container之间不能互相影像，可惜现阶段YARN的隔离做的还不是太好

(3)container是有最大资源限制的，在我们的设置中，每个Container最多只能有8G内存，8个CPU

(4)在RM端，根据不同的调度器，所有的资源被分成一个或多个队列，每个队列包含一定量的资源。用户的每个application，会被唯一的分配到一个队列去执行，队列

1.FIFO Scheduler把应用按提交的顺序排成一个队列，这是一个先进先出队列，在进行资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个 分配，以此类推。 资源分配的过程非常简单，很容易出现一个用户沾满集群所有资源的情况，优点是简单好理解，缺点是无法控制每个用户的资源使用。

2.Capacity Scheduler：在FIFO的基础上，增加多用户支持，最大化集群吞吐率和利用率.

朴素的思想是：每个用户都可以使用特定量的资源，但是集群空闲时，也可以使用整个集群的资源

3.FairScheduler

24.zookpeer提供了什么？

1.文件系统：

每个子目录项如 NameService 都被称作为znode，和文件系统一样，我们能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一

的不同在于znode是可以存储数据的。

有四种类型的znode：

1、PERSISTENT-持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

2、PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3、EPHEMERAL-临时目录节点

客户端与zookeeper断开连接后，该节点被删除

4、EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

2.zookpeer的通知机制：

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

25.zookpeer做了什么？

1.命名服务 ：

在zookeeper的文件系统里创建一个目录，即有唯一的path。在我们使用tborg无法确定上游程序的部署机器时即可与下游程序约定好path，通过

path即能互相探索发现。

2.配置管理

程序总是需要配置的，如果程序分散部署在多台机器上，要逐个改变配置就变得困难。现在把这些配置全部放到zookeeper上去，保存在

Zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 Zookeeper 的通

知，然后从 Zookeeper 获取新的配置信息应用到系统中就好

3.集群管理

所谓集群管理无在乎两点：是否有机器退出和加入、选举master。 对于第一点，所有机器约定在父目录GroupMembers下创建临时目录节点，然后

监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通

知：某个兄弟目录被删除，于是，所有人都知道：它上船了。新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount又有了，对于

第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为master就好。

4.分布式锁

有了zookeeper的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最

终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的distribute\_lock 节点就释放出锁。 对于第二类， /distribute\_lock 已经预先

存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，编号最小的获得锁，用完删除，依次方便。

5.队列管理

1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。

6.分布式和数据复制：

对zookeeper来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降 （这也是它建立observer的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

26.zookeeper工作原理：

Zookeeper 的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模 式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和 leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

27.spark中map和flapmap之间的区别？

api已经讲解较为清楚，map是将每个元素对应执行f函数，而flatMap对应的是将每个元素执行f函数后将其扁平化

word in text hello spark the third line

var mapResult = textFile.map(line => line.split("\\s+"))

res0:Array[Array[String]] = Array(Array(word, in, text), Array(hello, spark), Array(the, third, line))

var flatMapResult = textFile.flatMap(line => line.split("\\s+"))

res1: Array[String] = Array(word, in, text, hello, spark, the, third, line)

27.reduceBykey和groupBykey之间的区别？

(1) 采用reduceByKeyt时，Spark可以在每个分区移动数据之前将待输出数据与一个共用的key结合。借助下图可以理解在reduceByKey里究竟发生了什么。 注意在数据对被搬移前同一机器上同样的key是怎样被组合的

(2)当采用groupByKey时，由于它不接收函数，spark只能先将所有的键值对(key-value pair)都移动，这样的后果是集群节点之间的开销很大，导致传输延时。整个过程如下：

因此，在对大数据进行复杂计算时，reduceByKey优于groupByKey。

如果是groupByKey算子就结束了,下次执行ReduceByKey的时候，再进行相同key的聚合操作，这个时候会把shufflerdd进行聚合操作生成mapPartitionRdd,就是我们执行reduceByKey之后得到的那个rdd

28.oozie的任务调度？

OOZIE中定义了控制流节点和动作节点，都是基于XML定义的，其中控制流节点用来定义流程的开始，结束和执行路径，

三大功能：

workflow：顺序执行流程节点，支持fork，join

coordinator：定时触发一个workflow

bundle job：用来绑定多个coordinator或者多个workflow

总结wrokflow在oozie中的运行流程

1. 提交任务后，oozie中首先从指定的路径中读取job.properties
2. 查找需要运行的workflow的路径(在hdfs上)，将相应的变量传递给workflow
3. 读取workflow.xml文件中的节点定义，然后执行控制流节点和动作节点
4. 在hdfs读取运行任务所依赖的jar包

29.java的反射机制？

1.反射机制：

对于我们定义的每一个类，在任何的时刻，我们都能够知道这个类里面的属性和方法...对于任何一个对象，都能够调用这个类中的方法...这就

是反射机制的基本概念..

2.反射机制的功能：

在运行时判断任意一个对象所属的类，在运行时构造任意一个类的对象，在运行时判断任意类所具有的方法和属性，在运行时调用任意一个对象 的方法

生成动态代理...

3.如何使用反射机制：

我们如何去使用反射机制呢？

反射机制里一个特点就是实例化class对象，因为任意一个类对象都是class的实例...那么如何实例化class对象呢？三种方法：

i.通过forname()方法...

i i.对象.getclass()；

iii.类.class；

30.spark为啥比hadoop快？为什么？

1.统一的RDD抽象和操作，基于该抽象一个栈就可以轻松的处理实时流计算，SQL交互式查询，机

器学习和图计算等。

2.基于内存的迭代式计算

3.DAG 它可以把整个执行过程做一个图，然后进行优化。

4、消除了冗余的HDFS读写

Hadoop每次shuffle操作后，必须写到磁盘，而Spark在shuffle后不一定落盘，可以cache到内存中，以便迭代时使用。如果操作复杂，很多的shufle操作，那么Hadoop的读写IO时间会大大增加。

5、消除了冗余的MapReduce阶段

Hadoop的shuffle操作一定连着完整的MapReduce操作，冗余繁琐。而Spark基于RDD提供了丰富的算子操作，且reduce操作产生shuffle数据，可以缓存在内存中。

6、JVM的优化

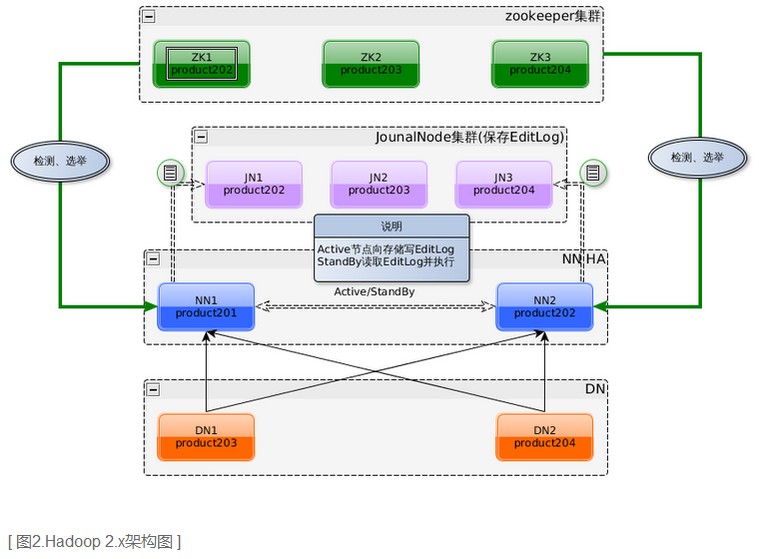
Hadoop每次MapReduce操作，启动一个Task便会启动一次JVM，基于进程的操作。而Spark 每次MapReduce操作是基于线程的，只在启动Executor是启动一次JVM，内存的Task操作是在线程复用的。每次启动JVM的时间可能就需要几秒甚至十几秒，那么当Task多了，这个时间Hadoop不知道比Spark慢了多少。

7.Spark Task的启动时间快。Spark采用fork线程的方式，而Hadoop采用创建新的进程的方式。

31.什么是内存计算技术？

我们说的内存计算技术是指将数据持久化至内存RAM中进行加工处理的技术。Spark并不支持将 数据持久化至内存中，我们通常所说的是spark的数据缓存技术，如将RDD数据缓存至内存，但并没有实现持久化。缓存数据是可以擦除的，擦除后同样是需要我们重新计算的。Spark的数据处理是在内存中进行的，这话并没有错，其实当前的所有操作系统的数据处理都是在内存中进行的。所以，这内存计算并不是Spark的特性。Spark是允许我们利用缓存技术和LRU算法缓存数据的。Spark的所有运算并不是全部都在内存中，当shuffle发生的时候，数据同样是需要写入磁盘的。所以，Spark并不是基于内存的技术，而是使用了缓存机制的技术。

32.hadoop中HA的原理：



为了确保快速切换，standby状态的NameNode有必要知道集群中所有数据块的位置。为了做到这点，所有的datanodes必须配置两个NameNode的地址，发送数据块位置信息和心跳给他们两个。

在HA集群中，standby状态的NameNode可以完成checkpoint操作，因此没必要配置Secondary NameNode、CheckpointNode、BackupNode。如果真的配置了，还会报错。

更新内存中的文件系统镜像。内存中的文件系统镜像用于 NameNode 向客户端提供读服务

内存中的文件系统镜像用于 NameNode 向客户端提供读服务，而 EditLog 仅仅只是在数据恢复的时候起作用

NameNode 会定期对内存中的文件系统镜像进行 checkpoint 操作，在磁盘上生成 FSImage 文件，

基于 QJM 的共享存储系统主要用于保存 EditLog，并不保存 FSImage 文件。FSImage 文件还是在 NameNode 的本地磁盘上。每个 JournalNode 保存同样的 EditLog 副本。

Standby NameNode 从 JournalNode 集群同步 EditLog

当 NameNode 进入 Standby 状态之后，会启动一个 EditLogTailer 线程。这个线程会定期调用 EditLogTailer 类的 doTailEdits 方法从 JournalNode 集群上同步 EditLog，然后把同步的 EditLog 回放到内存 之中的文件系统镜像上 (并不会同时把 EditLog 写入到本地磁盘上，这里为什么不写入磁盘呢)。

Standby NameNode 采用的是定时从 JournalNode 集群上同步 EditLog 的方式，那么 Standby NameNode 内存中文件系统镜像有很大的可能是落后于 Active NameNode 的，所以 Standby NameNode 在转换为 Active NameNode 的时候需要把落后的 EditLog 补上来。

33.mysql和postgresql之间的差别：

(1)mysql的各种text字段有不同的限制，需要手动区分small text,middle text, large text ，pg没有该限制，可以支持text的各种大小

(2)几乎任何数据库的子查询 (subquery) 性能都比 MySQL 好.

（3）pg它可以存储 array 和 json。空间函数比mysql好

（4）pg性能优于mysql（有人说三条街，哈哈哈）相同的数据量计算，pg时间一般只有mysql的四分之一

（5）般关系型数据库的字符串有限定长度8k左右，无限长 TEXT 类型的功能受限，只能作为外部大数据访问。而 PG 的 TEXT 类型可以直接访问，SQL语法内置正则表达式，可以索 引，还可以全文检索，或使用xml xpath。用PG的话，文档数据库都可以省了。

34.zookeeper介绍：

（1）是一个针对大型分布式系统的可靠协调系统；

（2）提供的功能包括：配置维护、名字服务、分布式同步、组服务等；

35.hbase:

<https://www.cnblogs.com/raphael5200/p/5229164.html>

36.redis:

[Redis](http://lib.csdn.net/base/redis) 是一个基于内存的高性能key-value[数据库](http://lib.csdn.net/base/mysql)

Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

<https://blog.csdn.net/waeceo/article/details/78701397>

37.redis和mysql之间的区别：

MySQL是关系型数据库，是持久化存储的，查询检索的话，会涉及到磁盘IO操作，为了提高性能，可以使用缓存技术

redis是缓存数据库，内存数据库，属于No sql系，可以大大提升高数据量的web访问速度

38.flume:

A.flume的一些基本的概念：

<1> flume是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。支持在日志系统中定制各类数据发送方，用于收集数据;同时，Flume提供对数据进行简单处理，

　　 并写到各种数据接受方(比如文本、HDFS、Hbase等)的能力 。

<2> flume中有source channel sink （三个部分的综合被称为agent）

source 数据源，兼容不同的协议，收集各种格式数据，包括：avro http thrift 自定义等，并处理产生event

sink 数据目的地，把event数据以各种格式和协议发往目的地组件，支持hdfs logger avro thrift file null hbase 自定义等

channel 对source采集到的数据进行缓存，存放方式可以支持memory、jdbc、file

agent flume的运行单位，是一个java进程，每台机器运行一个agent，一个agent对应一个配置文件，可以包含多个source和sink，channel

event 事件，flume的数据单元，传输数据的基本单位，包括event headers，event body。

<3>一行文本被序列化为一个event， event的最大定义字节是2048字节(默认，在avro方式中)，如果超过，则被切割，放到下一个event 默认编码utf-8

如果数据源是文本文件，body则是文本中的单行记录。

在flume源码中，headers的类型是Map集合

我们对headers的一些翻译：

fileHeader及fileHeaderKey：

fileHeader是个布尔值，可配置为true或者false，表示在flume读取数据之后，是否在封装出来的event中将文件名添加到event的header中。

fileHeaderKey表示，如果event中有了header（当fileHeader配置为true）时，header中存储文件名的<key,value>中，key命名是什么。比如，如果将fileHeaderKey定义为file，当读取文件名为/root/a.txt的内容时，header中就会有file=/root/a.txt这样的记录，如果fileHeaderKey定义为name，这个header就会是name=/root/a.txt这样的形式。

basenameHeader及basenameHeaderKey：

这两个配置的关系类似于fileHeader和fileHeaderKey的关系，只不过，basename返回的是文件的名字（不含目录部分）。比如上面的例子，如果定义basenameHeader为true，basenameHeaderKey为file，flume的event的header中，就会有file=a.txt这样的信息。

B.flume的可靠性和可恢复性：

可靠性主要依靠运行机制保证，当sink传输成功后，再删除channel的缓存数据。在0.9版本以前，可靠性是可以配置的，即三种级别的可靠性保证：end-to-end级别 收到数据的agent首先将event写到磁盘 上，数据传送成功后，再删除，如果数据发送失败，则重新发送 Store on failure级别 这时scribe采用的策略，当接受方crash时，将数据写到本地，待恢复后，继续发送 besteffort级别 数据发送到接收方后，不再确认，直接删除本地数据。但1.0版本以后，是不用配置级别的，依靠source和sink选取的协议组合搭配而产生的效果，比如sink使用tcp或者udt，对可靠性是有影响的。

可恢复性主要依靠file channel，可以把event持久化到本地文件系统。建议FileChannel设置的目录和程序日志文件保存的目录设成不同的磁盘，以便提高效率

C。flume的用法：

前后相继，多级串联

扇入，即接受多个source 输入

扇出，即数据可以输出到多个sink

D.拦截器：

拦截器的类型：

host主机名的拦截器（type=host），时间戳的拦截器(type=timestamp)，自定义类型拦截器（type=static），正则表达式来过滤是（type=regex\_filter）

39.sqoop

Sqoop是一个用于在Hadoop和关系数据库服务器之间传输数据的工具。它用于从关系数据库（如MySQL，Oracle）导入数据到Hadoop HDFS，并从Hadoop文件系统导出到关系数据库。它由Apache软件基金会提供。

40.kafka的安装和配置：

<https://www.cnblogs.com/alan319/p/8651434.html>

配制文件：broker.id 每个broker在集群中的唯一标示

log.dirs = /tmp/kafka-logs 数据存放的地址

提供给客户端响应的端口：port =6667

数据存储的最大时间 超过这个时间 会根据log.cleanup.policy设置的策略处理数据，也就是消费端能够多久去消费数据

## log.retention.bytes和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖

log.retention.minutes=7days

41.Linux下 环境变量/etc/profile、/etc/bashrc、~/.bashrc的区别

分系统环境变量和用户环境变量，系统环境变量对所有用户有效，而用户环境变量只对当前用户有效。

以上为系统环境变量及用户环境变量的配置，如果既配置了系统环境变量，又配置了用户环境变量，最终会以用户环境变量的配置为准。

1、etc目录下存放系统管理和配置文件

1)etc/profile:

profile为所有的用户设置***系统范围***的环境变量和启动顺序。当用户登录时读取该文件。这个文件对每个shell都有效。

2)/etc/bashrc：

为每一个运行bash shell的用户执行此文件，当bash shell被打开时,该文件被读取。也就是说，当用户shell执行了bash时，运行这个文件。

2、~/.bashrc

该文件存储的是***专属于个人***bash shell的信息，当登录时以及每次打开一个新的shell时,执行这个文件。在这个文件里可以自定义用户专属的个人信息。

42.各种组件的版本的判断：

spark版本的判断：spark-shell

hadoop版本的判断：hadoop version

kafka 版本的判断：find ./libs/ -name \\*kafka\_\\* | head -1 | grep -o '\kafka[^\n]\*'或者查看lib下的jar包文件

java的版本：java -version

flume的版本：flume-ng version

43.hadoop中wordcount编程计算的过程中，map和reduce函数：

map函数：接受一个键值对（key-value pair），产生一组中间键值对。MapReduce框架会将map函数产生的中间键值对里键相同的值传递给一个reduce函数。

　　reduce函数：接受一个键，以及相关的一组值，将这组值进行合并产生一组规模更小的值（通常只有一个或零个 值）。

44.hadoop 的wordcount程序：

package cn.edu.nuist.wordcount;

import java.io.IOException;

import java.util.Iterator;

import java.util.StringTokenizer;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

import org.apache.hadoop.mapred.TextInputFormat;

import org.apache.hadoop.mapred.TextOutputFormat;

/\*\*

\*

\* 描述：WordCount explains

\* @author Hadoop Dev Group

\*/

public class WordCount

{

private static String inPath = "/countWords/number.txt";

private static String outPath = "/countWords/output";

private static String hdfs = "hdfs://10.255.248.61:9000"; //这里配置集群中master主机IP和端口号

/\*\*

\* MapReduceBase类:实现了Mapper和Reducer接口的基类（其中的方法只是实现接口，而未作任何事情）

\* Mapper接口：

\* WritableComparable接口：实现WritableComparable的类可以相互比较。所有被用作key的类应该实现此接口。

\* Reporter 则可用于报告整个应用的运行进度，本例中未使用。

\*

\*/

public static class Map extends MapReduceBase implements

Mapper<LongWritable, Text, Text, IntWritable>

{

/\*\*

\* LongWritable, IntWritable, Text 均是 Hadoop 中实现的用于封装 Java 数据类型的类，这些类实现了WritableComparable接口，

\* 都能够被串行化从而便于在分布式环境中进行数据交换，你可以将它们分别视为long,int,String 的替代品。

\*/

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

private Text word = new Text();

/\*\*

\* Mapper接口中的map方法：

\* void map(K1 key, V1 value, OutputCollector<K2,V2> output, Reporter reporter)

\* 映射一个单个的输入k/v对到一个中间的k/v对

\* 输出对不需要和输入对是相同的类型，输入对可以映射到0个或多个输出对。

\* OutputCollector接口：收集Mapper和Reducer输出的<k,v>对。

\* OutputCollector接口的collect(k, v)方法:增加一个(k,v)对到output

\*/

public void map(LongWritable key, Text value,

OutputCollector<Text, IntWritable> output, Reporter reporter)

throws IOException

{

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line);

while (tokenizer.hasMoreTokens())

{

word.set(tokenizer.nextToken());

output.collect(word, one);

}

}

}

public static class Reduce extends MapReduceBase implements

Reducer<Text, IntWritable, Text, IntWritable>

{

public void reduce(Text key, Iterator<IntWritable> values,

OutputCollector<Text, IntWritable> output, Reporter reporter)

throws IOException

{

int sum = 0;

while (values.hasNext())

{

sum += values.next().get();

}

output.collect(key, new IntWritable(sum));

}

}

public static void main(String[] args) throws Exception

{

/\*\*

\* JobConf：map/reduce的job配置类，向hadoop框架描述map-reduce执行的工作

\* 构造方法：JobConf()、JobConf(Class exampleClass)、JobConf(Configuration conf)等

\*/

JobConf conf = new JobConf(WordCount.class);

conf.setJobName("wordcount"); //设置一个用户定义的job名称

conf.set("fs.default.name", hdfs);

conf.setOutputKeyClass(Text.class); //为job的输出数据设置Key类

conf.setOutputValueClass(IntWritable.class); //为job输出设置value类

conf.setMapperClass(Map.class); //为job设置Mapper类

conf.setCombinerClass(Reduce.class); //为job设置Combiner类

conf.setReducerClass(Reduce.class); //为job设置Reduce类

conf.setInputFormat(TextInputFormat.class); //为map-reduce任务设置InputFormat实现类

conf.setOutputFormat(TextOutputFormat.class); //为map-reduce任务设置OutputFormat实现类

/\*\*

\* InputFormat描述map-reduce中对job的输入定义

\* setInputPaths():为map-reduce job设置路径数组作为输入列表

\* setInputPath()：为map-reduce job设置路径数组作为输出列表

\*/

//设置输入输出文件路径

FileSystem fs = FileSystem.get(conf);

Path inputPath = new Path(inPath);

if(fs.exists(inputPath)) {

FileInputFormat.setInputPaths(conf, inputPath);

}

Path outputPath = new Path(outPath);

fs.delete(outputPath, true);

FileOutputFormat.setOutputPath(conf, outputPath);

JobClient.runJob(conf); //运行一个job

}

}

 45.spark的wordcount的程序：

package cn.spark.study.core

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

/\*\*

 \* @author Administrator

 \*/

object WordCount {

  def main(args: Array[String]) {

    val conf = new SparkConf()

        .setAppName("WordCount");

    val sc = new SparkContext(conf)

    val lines = sc.textFile("hdfs://spark1:9000/spark.txt", 1);

    val words = lines.flatMap { line => line.split(" ") }

    val pairs = words.map { word => (word, 1) }

    val wordCounts = pairs.reduceByKey { \_ + \_ }

    wordCounts.foreach(wordCount => println(wordCount.\_1 + " appeared " + wordCount.\_2 + " times."))

  }

}

scala的wordcount的java版本：

package com.mengyao.spark.java.core; import java.util.Arrays; import org.apache.spark.SparkConf; import org.apache.spark.api.java.JavaPairRDD; import org.apache.spark.api.java.JavaRDD; import org.apache.spark.api.java.JavaSparkContext; import org.apache.spark.api.java.function.FlatMapFunction; import org.apache.spark.api.java.function.Function2; import org.apache.spark.api.java.function.PairFunction; import org.apache.spark.api.java.function.VoidFunction; import scala.Tuple2; public class WordCountApp { public static void main(String[] args) { SparkConf conf = new SparkConf() JavaSparkContext sc = new JavaSparkContext(conf) JavaRDD<String> lines = sc.textFile("hdfs://soy1:9000/mapreduces/word.txt") JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>(){ private static final long serialVersionUID = -3243665984299496473L; @Override public Iterable<String> call(String line) throws Exception { return Arrays.asList(line.split("\t")); } }); JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() { private static final long serialVersionUID = -7879847028195817507L; @Override public Tuple2<String, Integer> call(String word) throws Exception { return new Tuple2<String, Integer>(word, 1); } }); JavaPairRDD<String, Integer> wordCount = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() { private static final long serialVersionUID = -4171349401750495688L; @Override public Integer call(Integer v1, Integer v2) throws Exception { return v1+v2; } }); wordCount.foreach(new VoidFunction<Tuple2<String,Integer>>() { private static final long serialVersionUID = -5926812153234798612L; @Override public void call(Tuple2<String, Integer> wordCount) throws Exception { System.out.println(wordCount.\_1+":"+wordCount.\_2); } }); wordCount.saveAsTextFile("hdfs://soy1:9000/spark/wordCount"); sc.close(); } }

46。Zookpeer在hbase中扮演的角色：

**Zookeeper**

　　　　　• 保证任何时候，集群中只有一个master

　　　　　• 存贮所有Region的寻址入口。

　　　　　• 实时监控Region server的上线和下线信息。并实时通知Master

　　　　　• 存储HBase的schema和table元数据

用hbase自带的zookpeer的话 会出现hbase和zookpeer之间耦合的现象，所以最好独立部署

47.UDP和TCP之间的差别：

<https://www.cnblogs.com/longiang7510/p/5441502.html>

48. lamada 和stream APIjavaAPAid的使用

不知道转变为stream之后是不是可以进行并行化处理

<https://www.baidu.com/link?url=g7_LoAT8SWqgvFwVoEJi9SnwLwu5lSS_u9IARheBcQktRLPHTBxmCg8uQYNjsNhj0Qrcqe2K191CQu8MHbzphK&wd=&eqid=c0b8b88c00984cf4000000025c09f3d5>

49.面向对象编程和面向函数编程之间有什么差别？

是什么

在叙述区别和优劣之前，简单介绍一下两者。

函数式编程，顾名思义，这种编程是以函数思维做为核心，在这种思维的角度去思考问题。这种编程最重要的基础是λ演算，接受函数当作输入和输出。

面向对象编程，这种编程是把问题看作由对象的属性与对象所进行的行为组成。基于对象的概念，以类作为对象的模板，把类和继承作为构造机制，以对象为中心，来思考并解决问题。

优点

函数式编程：支持闭包和高阶函数，闭包是一种可以起函数的作用并可以如对象般操作的对象；而高阶函数是可以以另一个函数作为输入值来进行编程。支持惰性计算，这就可以在求值需要表达式的值得时候进行计算，而不是固定在变量时计算。还有就是可以用递归作为控制流程。函数式编程所编程出来的代码相对而言少很多，而且更加简洁明了。

面向对象编程：面向对象有三个主要特征，分别是封装性、继承性和多态性。类的说明展现了封装性，类作为对象的模板，含有私有数据和公有数据，封装性能使数据更加安全依赖的就是类的特性，使得用户只能看到对象的外在特性，不能看到对象的内在属性，用户只能访问公有数据不能直接访问到私有数据。类的派生功能展现了继承性，继承性是子类共享父类的机制，但是由于封装性，继承性也只限于公有数据的继承（还有保护数据的继承），子类在继承的同时还可以进行派生。而多态性是指对象根据接收的信息作出的行为的多态，不同对象接收同一信息会形成多种行为。

缺点

函数式编程：所有的数据都是不可以改变的，严重占据运行资源，导致运行速度也不够快。

面向对象编程：为了编写可以重用的代码导致许多无用代码的产生，并且许多人为了面向对象而面向对象导致代码给后期维护带来很多麻烦。

50.Driver在哪里启动的？

yarn-cluster模式下，Dirver运行在ApplicationMaster中，负责申请资源并监控task运行状态和重试失败的task，

当用户提交了作业之后就可以关掉client，作业会继续在yarn中运行；

yarn-client模式下，Dirver运行在本地客户端，client不能离开。

51. 如何查看远程查看一个电脑的端口是不是打开？

首先端口是有tcp和udp之分的，所以同一个端口号还有tcp协议和udp协议之分

tcp：

telnet ip +端口号 eg:telnet 192.168.75.128 22

nc(netcat) -z 192.168.75.128 22

udp:（其中u代表的是udp）

nc(netcat) -uz 192.168.75.128 22

并且我们的nc还可以同时查询多个端口：

nc -vz 192.168.75.128 22

那么不是远程的话，我们可以选择 netstat -an | grep 3306 测试的是所有的3306端口的使用状况

52.open命令的简单的使用：

open +文件的名字 可以使用默认的打开方式打开文件进行编辑

du命令: 显示每个文件和目录的磁盘使用空间

du log2012.log 显示某一个特定的文件占有的空间

du scf 显示某个目录占用的空间

du log30.tar.gz log31.tar.gz 显示多个目录占用的空间

du -s 显示当前目录下总的占用的空间

du -h test 以方便阅读的方式来显示

53.java的锁的机制：

死锁就是线程两房拿着对方的锁，使得线程阻塞

54.有10G大小的文件，但是目前只有50M的内存，该如何处理这批数据，求的是单词的频率的问题

处理的方式就是使用流的方式进行处理：

使用流将文件中的单词按照hash分别读到不同的小文件中，然后我们就可以在这50M内存中处理这些小文件

55.Hbase中的blockCache实现机制：

HBase上RegionServer的cache主要分为两个部分，分别是memstore&blockcache，其中memstore主要用于写缓存，而blockcache用于读缓存。

当数据写入hbase时，会先写入memstore，RegionServer会给每个region提供一个memstore，memstore中的数据达到系统设置的水位值后，会触发flush将memstore中的数据刷写到磁

盘。客户的读请求会先到memstore中查数据，若查不到就到blockcache中查，再查不到就会从磁盘上读，并把读入的数据同时放入blockcahce。我们知道缓存有三种不同的更新策略，

分别是先入先出（FIFO）、LRU（最近最少使用）和LFU（最近最不常使用），hbase的block使用的是LRU策略，当BlockCache的大小达到上限后，会触发缓存淘汰机制，将最老的一

批 数据淘汰掉。（那么写到磁盘的就是storeFile）一个RegionServer上有一个BlockCache和N个Memstore。

56.kafka和spark Streaming的集成：

在实际开发中Spark Streaming经常会结合Flume以及Kafka来实时计算数据。这篇文章主要讲述如何集成Kafka，对于Kafka的集成有两种方式：

1. Receiver-based Approach

2.Direct Approach (No Receivers)

这种方式使用一个Receiver 来接收数据，它可以使用kafka API来消费数据并且进行统计，接收到的数据存储在Spark Executor 中的Receiver 上，然后使用Spark Streaming对数据进

行分析，不过这种方式当节点发生错误的时候数据容易丢失。

57.spark算子中map和flatMap之间的区别：

map（）对RDD的每一个item都执行同一个操作

flatmap（）对RDD的item执行同一个操作以后得到一个list，然后以平铺的方式把这些list里所有的结果组成新的list

58.spark中collect()算子如果数据量太大怎么办？collect是将数据转成array数据发送到Driver端

这时候可以修改collect的源码，将数据放在分布式文件系统中

59.为什么对一些输入输出的文件流要进行关闭（对IO流必须要关闭的思考）

比如打开的物理资源，比如数据库连接，网络连接，磁盘文件等，打开这些文件之后必须显示关闭，否则会引发资源泄露

因为java的垃圾回收机制属于java内存管理的一部分，它只负责堆内存中分配出来的内存，至于程序中打开的物理资源，垃圾回收机制是无能为力的

你不关闭的话就会占用你的大量的内存空间资源，不释放，就会内存溢出。

60.桶表的简单使用：（以及桶表和分区表的区别）：

第一，分桶之前要执行命令hive.enforce.bucketiong=true;

A。基本的概念：

（1）桶表是对某一列数据进行hash取值将数据打撒，然后放到不同的文件中存储。

（2）在hive分区表中，分区中的数量过于庞大时，建议使用桶

（3）在分桶的，对执行的字段进行hash得到hash值，并使用hash值除以桶的根数做余数运算得到的值进行分桶，保证每个不同的桶中有数据但是每个桶的数据不一定相等

在做hash运算的时候，hash函数的选择取决于分桶字段的数据类型

（4）分桶的查询效率比分区后的查询效率低

B。桶表的创建：  create table btable1 (id int) clustered by(id) into 4 buckets;

C。桶表不能通过load的方式直接加载数据，只能从另一张表中插入数据

61.RDD和DataSet,DataFrame之间的区别：

Rdd可以存储对象非结构化的数据，不适用于SQLcontext，HIveContext，适用SparkContext

DataFram存储的是Row的结构化数据，支持SQLcontext，HIveContext，

DataSet支持sparkContext，hiveContext，SQLContext，DataFrame[Row]是Dataset的一种数据格式，

Dataset和DataFrame之间的区别就是DataSet不是一条数据，可以根据字段键直接返回值

但是DataFrame存储的是一条完整的数据，只有通过解析才能获得各个字段的值，每一列的值没办法直接访问（模式匹配）

DataSet的每行的数据都是非常确定。

62.解释性语言和编译型语言的区别：

这个非常的详细

<https://blog.csdn.net/u014647208/article/details/78329187>