**4、hadoop和spark的shuffle相同和差异？**

1）从 high-level 的角度来看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() （Spark 里可能是后续的一系列操作）。

2）从 low-level 的角度来看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。目前的 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是Spark 1.1的用户，可以将spark.shuffle.manager设置为sort，则会对数据进行排序。在Spark 1.2中，sort将作为默认的Shuffle实现。

3）从实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。如果我们将 map 端划分数据、持久化数据的过程称为 shuffle write，而将 reducer 读入数据、aggregate 数据的过程称为 shuffle read。那么在 Spark 中，问题就变为怎么在 job 的逻辑或者物理执行图中加入 shuffle write 和 shuffle read 的处理逻辑？以及两个处理逻辑应该怎么高效实现？ Shuffle write由于不要求数据有序，shuffle write 的任务很简单：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。

#### 8、RDD有哪些缺陷？

1）不支持细粒度的写和更新操作（如网络爬虫），spark写数据是粗粒度的所谓粗粒度，就是批量写入数据，为了提高效率。但是读数据是细粒度的也就是说可以一条条的读

2）不支持增量迭代计算，Flink支持

#### 9、Spark中数据的位置是被谁管理的？

每个数据分片都对应具体物理位置，数据的位置是被blockManager，无论数据是在磁盘，内存还是tacyan，都是由blockManager管理

#### 10、Spark的数据本地性有哪几种？

答：Spark中的数据本地性有三种：a.PROCESS\_LOCAL是指读取缓存在本地节点的数据b.NODE\_LOCAL是指读取本地节点硬盘数据c.ANY是指读取非本地节点数据通常读取数据PROCESS\_LOCAL>NODE\_LOCAL>ANY，尽量使数据以PROCESS\_LOCAL或NODE\_LOCAL方式读取。其中PROCESS\_LOCAL还和cache有关，如果RDD经常用的话将该RDD cache到内存中，注意，由于cache是lazy的，所以必须通过一个action的触发，才能真正的将该RDD cache到内存中

#### 11、rdd有几种操作类型？

1）transformation，rdd由一种转为另一种rdd

2）action，

3）cronroller，crontroller是控制算子,cache,persist，对性能和效率的有很好的支持三种类型，不要回答只有2中操作

#### 13、为什么Spark Application在没有获得足够的资源，job就开始执行了，可能会导致什么什么问题发生?

答：会导致执行该job时候集群资源不足，导致执行job结束也没有分配足够的资源，分配了部分Executor，该job就开始执行task，应该是task的调度线程和Executor资源申请是异步的；如果想等待申请完所有的资源再执行job的：需要将spark.scheduler.maxRegisteredResourcesWaitingTime设置的很大；spark.scheduler.minRegisteredResourcesRatio 设置为1，但是应该结合实际考虑否则很容易出现长时间分配不到资源，job一直不能运行的情况。

#### 14、join操作优化经验？

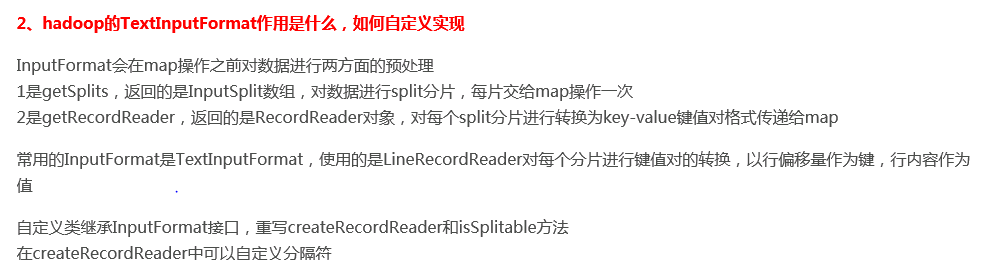
join其实常见的就分为两类： map-side join 和  reduce-side join。当大表和小表join时，用map-side join能显著提高效率。将多份数据进行关联是数据处理过程中非常普遍的用法，不过在分布式计算系统中，这个问题往往会变的非常麻烦，因为框架提供的 join 操作一般会将所有数据根据 key 发送到所有的 reduce 分区中去，也就是 shuffle 的过程。造成大量的网络以及磁盘IO消耗，运行效率极其低下，这个过程一般被称为 reduce-side-join。如果其中有张表较小的话，我们则可以自己实现在 map 端实现数据关联，跳过大量数据进行 shuffle 的过程，运行时间得到大量缩短，根据不同数据可能会有几倍到数十倍的性能提升。

#### 15、介绍一下cogroup rdd实现原理，你在什么场景下用过这个rdd？

答：cogroup的函数实现:这个实现根据两个要进行合并的两个RDD操作,生成一个CoGroupedRDD的实例,这个RDD的返回结果是把相同的key中两个RDD分别进行合并操作,最后返回的RDD的value是一个Pair的实例,这个实例包含两个Iterable的值,第一个值表示的是RDD1中相同KEY的值,第二个值表示的是RDD2中相同key的值.由于做cogroup的操作,需要通过partitioner进行重新分区的操作,因此,执行这个流程时,需要执行一次shuffle的操作(如果要进行合并的两个RDD的都已经是shuffle后的rdd,同时他们对应的partitioner相同时,就不需要执行shuffle

#### 2、hadoop的TextInputFormat作用是什么，如何自定义实现

InputFormat会在map操作之前对数据进行两方面的预处理1是getSplits，返回的是InputSplit数组，对数据进行split分片，每片交给map操作一次2是getRecordReader，返回的是RecordReader对象，对每个split分片进行转换为key-value键值对格式传递给map常用的InputFormat是TextInputFormat，使用的是LineRecordReader对每个分片进行键值对的转换，以行偏移量作为键，行内容作为值自定义类继承InputFormat接口，重写createRecordReader和isSplitable方法在createRecordReader中可以自定义分隔符\



**16、Hadoop性能调优？**

调优可以通过系统配置、程序编写和作业调度算法来进行。  
hdfs的block.size可以调到128/256（网络很好的情况下，默认为64）  
调优的大头：mapred.map.tasks、mapred.reduce.tasks设置mr任务数（默认都是1）  
mapred.tasktracker.map.tasks.maximum每台机器上的最大map任务数  
mapred.tasktracker.reduce.tasks.maximum每台机器上的最大reduce任务数  
mapred.reduce.slowstart.completed.maps配置reduce任务在map任务完成到百分之几的时候开始进入  
这个几个参数要看实际节点的情况进行配置，reduce任务是在33%的时候完成copy，要在这之前完成map任务，（map可以提前完成）  
mapred.compress.map.output,mapred.output.compress配置压缩项，消耗cpu提升网络和磁盘io  
合理利用combiner  
注意重用writable对象

**21、map-reduce程序运行的时候会有什么比较常见的问题？**

比如说作业中大部分都完成了，但是总有几个reduce一直在运行。  
这是因为这几个reduce中的处理的数据要远远大于其他的reduce，可能是因为对键值对任务划分的不均匀造成的数据倾斜。  
解决的方法可以在分区的时候重新定义分区规则对于value数据很多的key可以进行拆分、均匀打散等处理，或者是在map端的combiner中进行数据预处理的操作。

**24、spark有哪些组件？**

（1）master：管理集群和节点，不参与计算。  
（2）worker：计算节点，进程本身不参与计算，和master汇报。  
（3）Driver：运行程序的main方法，创建spark context对象。  
（4）spark context：控制整个application的生命周期，包括dagsheduler和task scheduler等组件。  
（5）client：用户提交程序的入口。

**spark 如何防止内存溢出**

driver端的内存溢出

可以增大driver的内存参数：spark.driver.memory (default 1g)

这个参数用来设置Driver的内存。在Spark程序中，SparkContext，DAGScheduler都是运行在Driver端的。对应rdd的Stage切分也是在Driver端运行，如果用户自己写的程序有过多的步骤，切分出过多的Stage，这部分信息消耗的是Driver的内存，这个时候就需要调大Driver的内存。

map过程产生大量对象导致内存溢出

这种溢出的原因是在单个map中产生了大量的对象导致的，例如：rdd.map(x=>for(i <- 1 to 10000) yield i.toString)，这个操作在rdd中，每个对象都产生了10000个对象，这肯定很容易产生内存溢出的问题。针对这种问题，在不增加内存的情况下，可以通过减少每个Task的大小，以便达到每个Task即使产生大量的对象Executor的内存也能够装得下。具体做法可以在会产生大量对象的map操作之前调用repartition方法，分区成更小的块传入map。例如：rdd.repartition(10000).map(x=>for(i <- 1 to 10000) yield i.toString)。

面对这种问题注意，不能使用rdd.coalesce方法，这个方法只能减少分区，不能增加分区，不会有shuffle的过程。

数据不平衡导致内存溢出

数据不平衡除了有可能导致内存溢出外，也有可能导致性能的问题，解决方法和上面说的类似，就是调用repartition重新分区。这里就不再累赘了。

shuffle后内存溢出

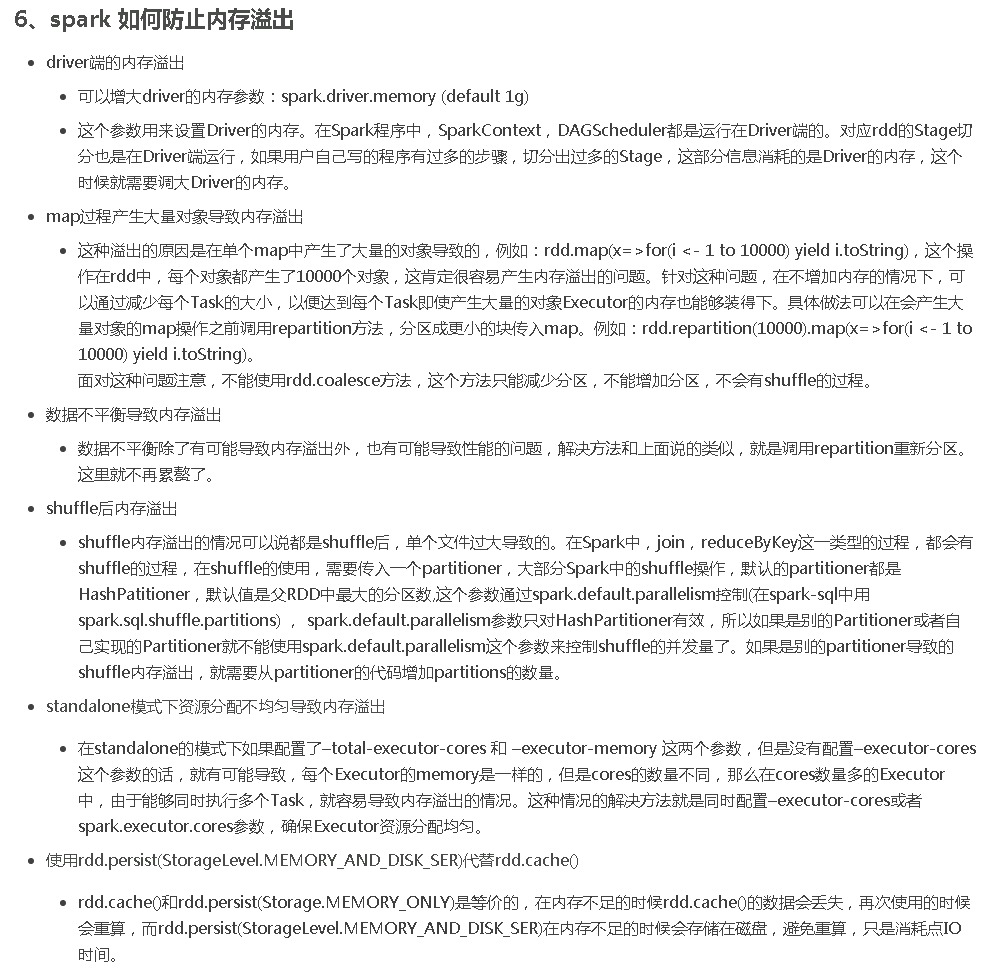
shuffle内存溢出的情况可以说都是shuffle后，单个文件过大导致的。在Spark中，join，reduceByKey这一类型的过程，都会有shuffle的过程，在shuffle的使用，需要传入一个partitioner，大部分Spark中的shuffle操作，默认的partitioner都是HashPatitioner，默认值是父RDD中最大的分区数,这个参数通过spark.default.parallelism控制(在spark-sql中用spark.sql.shuffle.partitions) ， spark.default.parallelism参数只对HashPartitioner有效，所以如果是别的Partitioner或者自己实现的Partitioner就不能使用spark.default.parallelism这个参数来控制shuffle的并发量了。如果是别的partitioner导致的shuffle内存溢出，就需要从partitioner的代码增加partitions的数量。

standalone模式下资源分配不均匀导致内存溢出

在standalone的模式下如果配置了–total-executor-cores 和 –executor-memory 这两个参数，但是没有配置–executor-cores这个参数的话，就有可能导致，每个Executor的memory是一样的，但是cores的数量不同，那么在cores数量多的Executor中，由于能够同时执行多个Task，就容易导致内存溢出的情况。这种情况的解决方法就是同时配置–executor-cores或者spark.executor.cores参数，确保Executor资源分配均匀。

使用rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)代替rdd.cache()

rdd.cache()和rdd.persist(Storage.MEMORY\_ONLY)是等价的，在内存不足的时候rdd.cache()的数据会丢失，再次使用的时候会重算，而rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)在内存不足的时候会存储在磁盘，避免重算，只是消耗点IO时间。



**spark中的数据倾斜的现象、原因、后果**

(1)、数据倾斜的现象

多数task执行速度较快,少数task执行时间非常长，或者等待很长时间后提示你内存不足，执行失败。

(2)、数据倾斜的原因

数据问题

1、key本身分布不均衡（包括大量的key为空）

2、key的设置不合理

spark使用问题

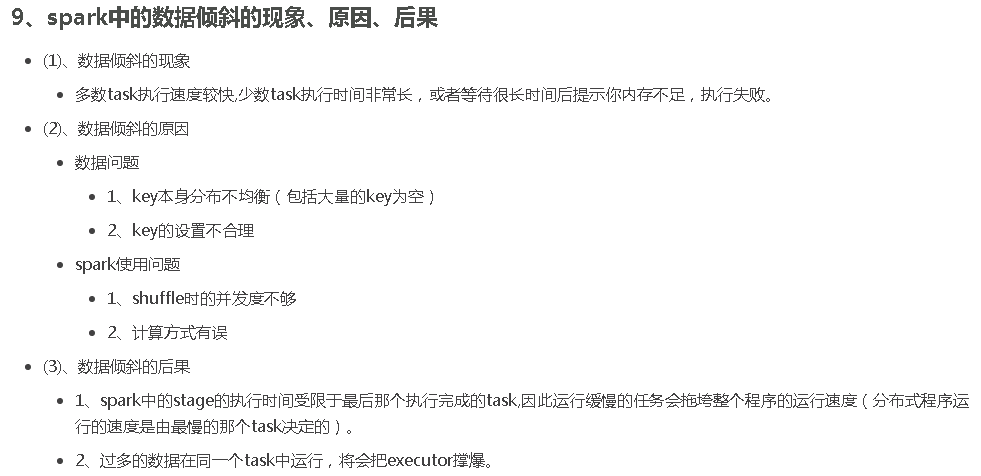
1、shuffle时的并发度不够

2、计算方式有误

(3)、数据倾斜的后果

1、spark中的stage的执行时间受限于最后那个执行完成的task,因此运行缓慢的任务会拖垮整个程序的运行速度（分布式程序运行的速度是由最慢的那个task决定的）。

2、过多的数据在同一个task中运行，将会把executor撑爆。



**10、如何解决spark中的数据倾斜问题**

发现数据倾斜的时候，不要急于提高executor的资源，修改参数或是修改程序，首先要检查数据本身，是否存在异常数据。

1、数据问题造成的数据倾斜

找出异常的key

如果任务长时间卡在最后最后1个(几个)任务，首先要对key进行抽样分析，判断是哪些key造成的。

选取key，对数据进行抽样，统计出现的次数，根据出现次数大小排序取出前几个。

比如: df.select(“key”).sample(false,0.1).(k=>(k,1)).reduceBykey(+).map(k=>(k.\_2,k.\_1)).sortByKey(false).take(10)

如果发现多数数据分布都较为平均，而个别数据比其他数据大上若干个数量级，则说明发生了数据倾斜。

经过分析，倾斜的数据主要有以下三种情况:

1、null（空值）或是一些无意义的信息()之类的,大多是这个原因引起。

2、无效数据，大量重复的测试数据或是对结果影响不大的有效数据。

3、有效数据，业务导致的正常数据分布。

解决办法

第1，2种情况，直接对数据进行过滤即可（因为该数据对当前业务不会产生影响）。

第3种情况则需要进行一些特殊操作，常见的有以下几种做法

(1) 隔离执行，将异常的key过滤出来单独处理，最后与正常数据的处理结果进行union操作。

(2) 对key先添加随机值，进行操作后，去掉随机值，再进行一次操作。

(3) 使用reduceByKey 代替 groupByKey(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.)

(4) 使用map join。

案例

如果使用reduceByKey因为数据倾斜造成运行失败的问题。具体操作流程如下:

(1) 将原始的 key 转化为 key + 随机值(例如Random.nextInt)

(2) 对数据进行 reduceByKey(func)

(3) 将 key + 随机值 转成 key

(4) 再对数据进行 reduceByKey(func)

案例操作流程分析：

假设说有倾斜的Key，我们给所有的Key加上一个随机数，然后进行reduceByKey操作；此时同一个Key会有不同的随机数前缀，在进行reduceByKey操作的时候原来的一个非常大的倾斜的Key就分而治之变成若干个更小的Key，不过此时结果和原来不一样，怎么破？进行map操作，目的是把随机数前缀去掉，然后再次进行reduceByKey操作。（当然，如果你很无聊，可以再次做随机数前缀），这样我们就可以把原本倾斜的Key通过分而治之方案分散开来，最后又进行了全局聚合

注意1: 如果此时依旧存在问题，建议筛选出倾斜的数据单独处理。最后将这份数据与正常的数据进行union即可。

注意2: 单独处理异常数据时，可以配合使用Map Join解决。

2、spark使用不当造成的数据倾斜

提高shuffle并行度

dataFrame和sparkSql可以设置spark.sql.shuffle.partitions参数控制shuffle的并发度，默认为200。

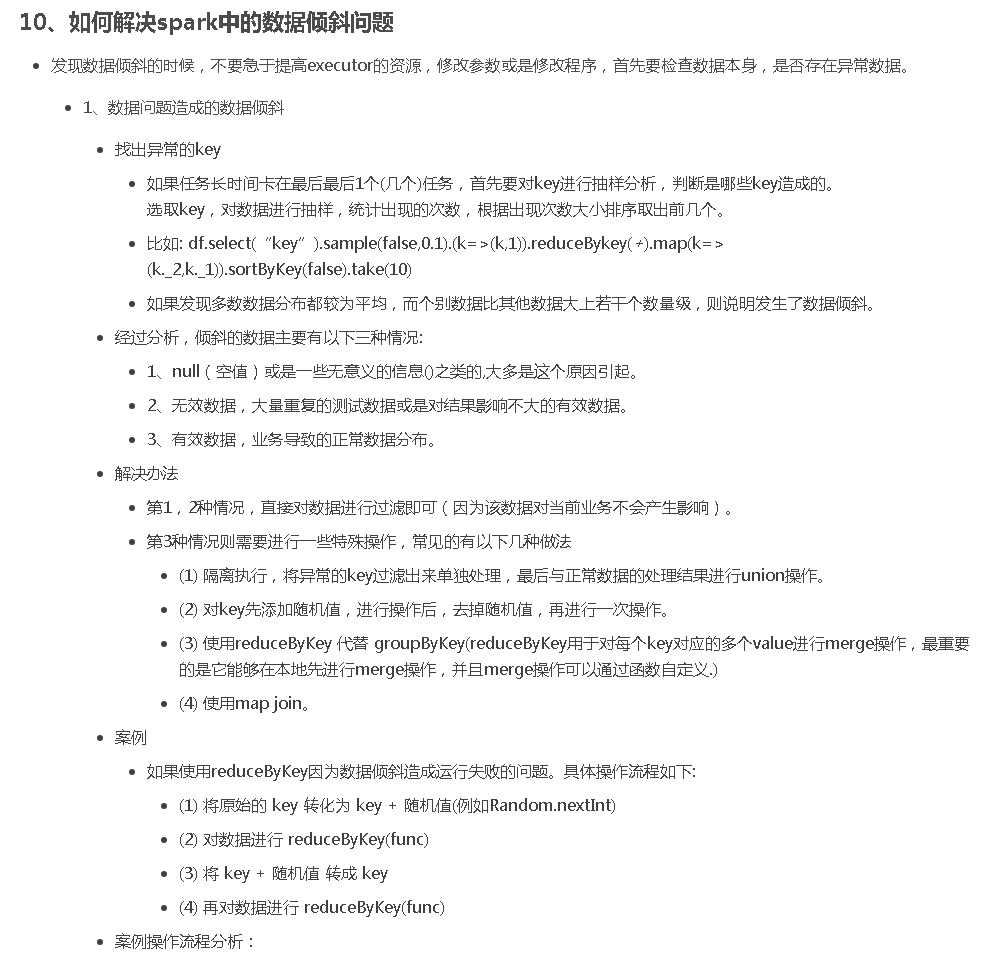
rdd操作可以设置spark.default.parallelism控制并发度，默认参数由不同的Cluster Manager控制。

局限性: 只是让每个task执行更少的不同的key。无法解决个别key特别大的情况造成的倾斜，如果某些key的大小非常大，即使一个task单独执行它，也会受到数据倾斜的困扰。

使用map join 代替reduce join

在小表不是特别大(取决于你的executor大小)的情况下使用，可以使程序避免shuffle的过程，自然也就没有数据倾斜的困扰了.（详细见http://blog.csdn.net/lsshlsw/article/details/50834858、http://blog.csdn.net/lsshlsw/article/details/48694893）

局限性: 因为是先将小数据发送到每个executor上，所以数据量不能太大





**12、kafka整合sparkStreaming问题**

(1)、如何实现sparkStreaming读取kafka中的数据

可以这样说：在kafka0.10版本之前有二种方式与sparkStreaming整合，一种是基于receiver，一种是direct,然后分别阐述这2种方式分别是什么

receiver：是采用了kafka高级api,利用receiver接收器来接受kafka topic中的数据，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，kafka中topic的偏移量是保存在zk中的。

基本使用： val kafkaStream = KafkaUtils.createStream(streamingContext,

[ZK quorum], [consumer group id], [per-topic number of Kafka partitions to consume])

还有几个需要注意的点：

在Receiver的方式中，Spark中的partition和kafka中的partition并不是相关的，所以如果我们加大每个topic的partition数量，仅仅是增加线程来处理由单一Receiver消费的主题。但是这并没有增加Spark在处理数据上的并行度.

对于不同的Group和topic我们可以使用多个Receiver创建不同的Dstream来并行接收数据，之后可以利用union来统一成一个Dstream。

在默认配置下，这种方式可能会因为底层的失败而丢失数据. 因为receiver一直在接收数据,在其已经通知zookeeper数据接收完成但是还没有处理的时候,executor突然挂掉(或是driver挂掉通知executor关闭),缓存在其中的数据就会丢失. 如果希望做到高可靠, 让数据零丢失,如果我们启用了Write Ahead Logs(spark.streaming.receiver.writeAheadLog.enable=true）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. 所以, 即使底层节点出现了失败, 也可以使用预写日志中的数据进行恢复. 复制到文件系统如HDFS，那么storage level需要设置成 StorageLevel.MEMORY\_AND\_DISK\_SER，也就是KafkaUtils.createStream(…, StorageLevel.MEMORY\_AND\_DISK\_SER)

direct:在spark1.3之后，引入了Direct方式。不同于Receiver的方式，Direct方式没有receiver这一层，其会周期性的获取Kafka中每个topic的每个partition中的最新offsets，之后根据设定的maxRatePerPartition来处理每个batch。（设置spark.streaming.kafka.maxRatePerPartition=10000。限制每秒钟从topic的每个partition最多消费的消息条数）。

(2) 对比这2中方式的优缺点：

采用receiver方式：这种方式可以保证数据不丢失，但是无法保证数据只被处理一次，WAL实现的是At-least-once语义（至少被处理一次），如果在写入到外部存储的数据还没有将offset更新到zookeeper就挂掉,这些数据将会被反复消费. 同时,降低了程序的吞吐量。

采用direct方式:相比Receiver模式而言能够确保机制更加健壮. 区别于使用Receiver来被动接收数据, Direct模式会周期性地主动查询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. 当处理数据的job启动时, 就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。

优点：

1、简化并行读取

如果要读取多个partition, 不需要创建多个输入DStream然后对它们进行union操作. Spark会创建跟Kafka partition一样多的RDD partition, 并且会并行从Kafka中读取数据. 所以在Kafka partition和RDD partition之间, 有一个一对一的映射关系.

2、高性能

如果要保证零数据丢失, 在基于receiver的方式中, 需要开启WAL机制. 这种方式其实效率低下, 因为数据实际上被复制了两份, Kafka自己本身就有高可靠的机制, 会对数据复制一份, 而这里又会复制一份到WAL中. 而基于direct的方式, 不依赖Receiver, 不需要开启WAL机制, 只要Kafka中作了数据的复制, 那么就可以通过Kafka的副本进行恢复.

3、一次且仅一次的事务机制

基于receiver的方式, 是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的. 这是消费Kafka数据的传统方式. 这种方式配合着WAL机制可以保证数据零丢失的高可靠性, 但是却无法保证数据被处理一次且仅一次, 可能会处理两次. 因为Spark和ZooKeeper之间可能是不同步的. 基于direct的方式, 使用kafka的简单api, Spark Streaming自己就负责追踪消费的offset, 并保存在checkpoint中. Spark自己一定是同步的, 因此可以保证数据是消费一次且仅消费一次。不过需要自己完成将offset写入zk的过程,在官方文档中都有相应介绍.

\*简单代码实例：

\* messages.foreachRDD(rdd=>{

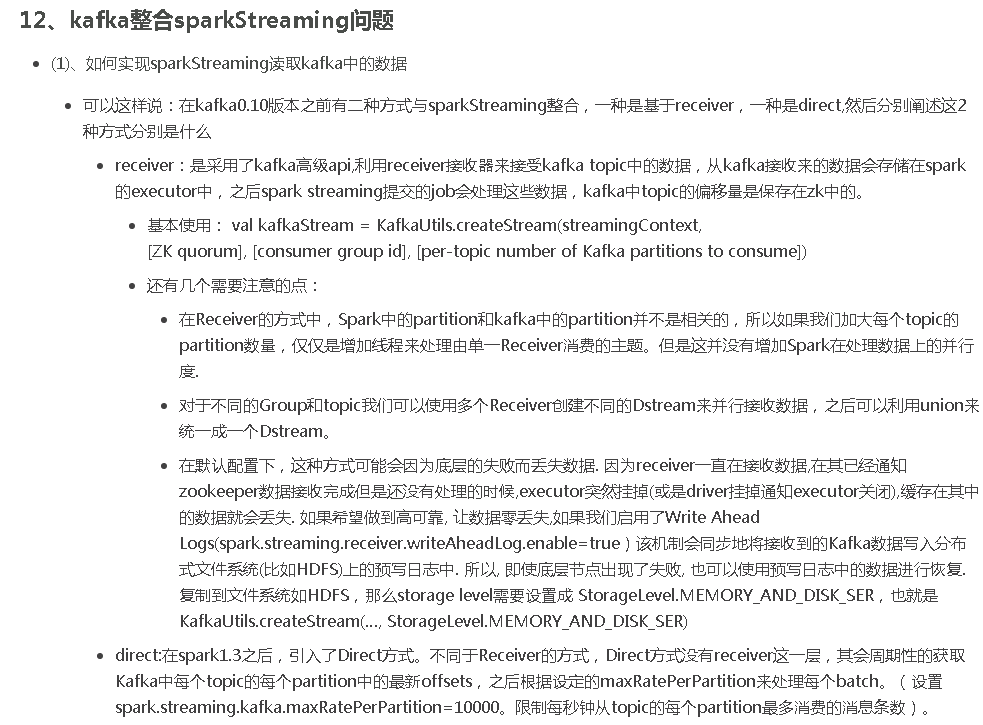
val message = rdd.map(\_.\_2)//对数据进行一些操作

message.map(method)//更新zk上的offset (自己实现)

updateZKOffsets(rdd)

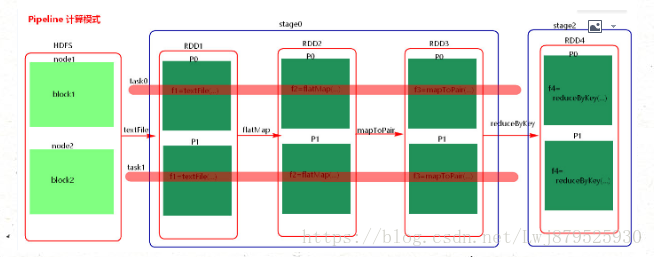
})

\* sparkStreaming程序自己消费完成后，自己主动去更新zk上面的偏移量。也可以将zk中的偏移量保存在mysql或者redis数据库中，下次重启的时候，直接读取mysql或者redis中的偏移量，获取到上次消费的偏移量，接着读取数据。





5.计算格式：pipeline管道计算模式，piepeline只是一种计算思想，一种模式



6.spark的pipeline管道计算模式相当于执行了一个高阶函数，也就是说来一条数据然后计算一条数据，会把所有的逻辑走完，然后落地，而MapReduce是1+1=2，2+1=3这样的计算模式，也就是计算完落地，然后再计算，然后再落地到磁盘或者内存，最后数据是落在计算节点上，按reduce的hash分区落地。管道计算模式完全基于内存计算，所以比MapReduce快的原因。

7.管道中的RDD何时落地：shuffle write的时候，对RDD进行持久化的时候。

8.stage的task的并行度是由stage的最后一个RDD的分区数来决定的，一般来说，一个partition对应一个task，但最后reduce的时候可以手动改变reduce的个数，也就是改变最后一个RDD的分区数，也就改变了并行度。例如：reduceByKey(\_+\_,3)

9.优化：提高stage的并行度：reduceByKey(\_+\_,patition的个数) ，join(\_+\_,patition的个数)

6.有向无环图：

答：DAG，有向无环图，简单的来说，就是一个由顶点和有方向性的边构成的图中，从任意一个顶点出发，没有任意一条路径会将其带回到出发点的顶点位置，为每个spark job计算具有依赖关系的多个stage任务阶段，通常根据shuffle来划分stage，如reduceByKey,groupByKey等涉及到shuffle的transformation就会产生新的stage ，然后将每个stage划分为具体的一组任务，以TaskSets的形式提交给底层的任务调度模块来执行，其中不同stage之前的RDD为宽依赖关系，TaskScheduler任务调度模块负责具体启动任务，监控和汇报任务运行情况。



**10.RDD共享变量：**

在应用开发中，一个函数被传递给Spark操作（例如map和reduce），在一个远程集群上运行，它实际上操作的是这个函数用到的所有变量的独立拷贝。这些变量会被拷贝到每一台机器。通常看来，在任务之间中，读写共享变量显然不够高效。然而，Spark还是为两种常见的使用模式，提供了两种有限的共享变量：广播变量和累加器。

(1). 广播变量（Broadcast Variables）

– 广播变量缓存到各个节点的内存中，而不是每个 Task

– 广播变量被创建后，能在集群中运行的任何函数调用

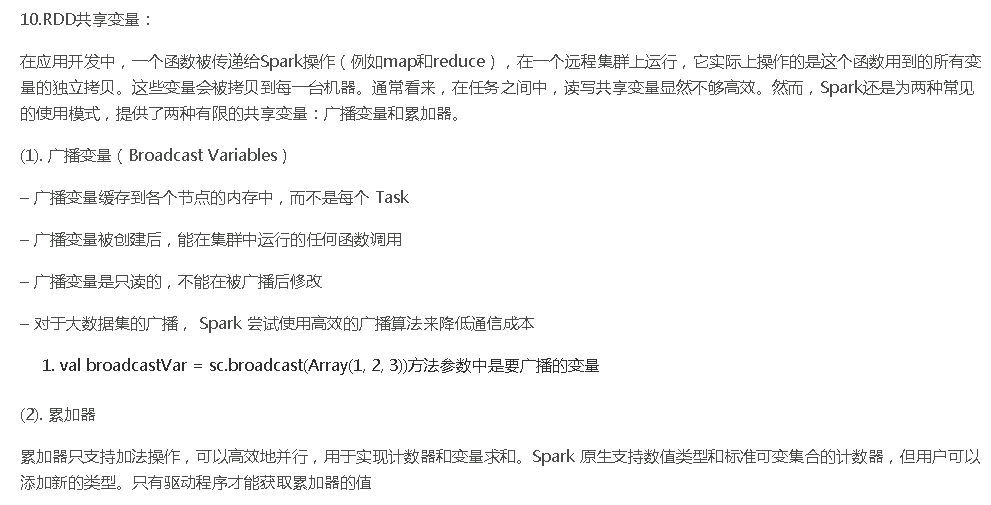
– 广播变量是只读的，不能在被广播后修改

– 对于大数据集的广播， Spark 尝试使用高效的广播算法来降低通信成本

val broadcastVar = sc.broadcast(Array(1, 2, 3))方法参数中是要广播的变量

(2). 累加器

累加器只支持加法操作，可以高效地并行，用于实现计数器和变量求和。Spark 原生支持数值类型和标准可变集合的计数器，但用户可以添加新的类型。只有驱动程序才能获取累加器的值



17.spark中map-side-join关联优化：

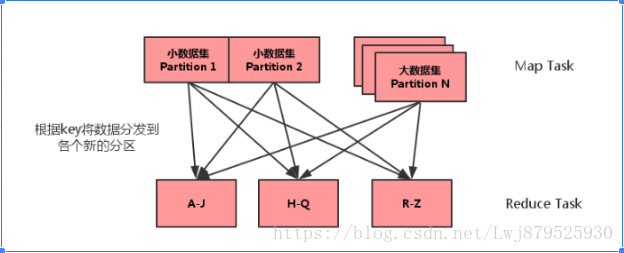
将多份数据进行关联是数据处理过程中非常普遍的用法，不过在分布式计算系统中，这个问题往往会变的非常麻烦，因为框架提供的 join 操作一般会将所有数据根据 key 发送到所有的 reduce 分区中去，也就是 shuffle 的过程。造成大量的网络以及磁盘IO消耗，运行效率极其低下，这个过程一般被称为 reduce-side-join。

如果其中有张表较小的话，我们则可以自己实现在 map 端实现数据关联，跳过大量数据进行 shuffle 的过程，运行时间得到大量缩短，根据不同数据可能会有几倍到数十倍的性能提升。

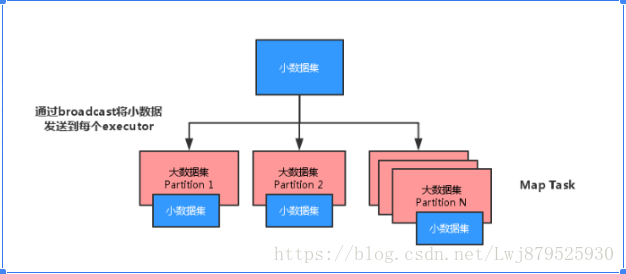
何时使用：在海量数据中匹配少量特定数据

原理：reduce-side-join 的缺陷在于会将key相同的数据发送到同一个partition中进行运算，大数据集的传输需要长时间的IO，同时任务并发度收到限制，还可能造成数据倾斜。

reduce-side-join 运行图如下



map-side-join 运行图如下：



将少量的数据转化为Map进行广播，广播会将此 Map 发送到每个节点中，如果不进行广播，每个task执行时都会去获取该Map数据，造成了性能浪费。对大数据进行遍历，使用mapPartition而不是map，因为mapPartition是在每个partition中进行操作，因此可以减少遍历时新建broadCastMap.value对象的空间消耗，同时匹配不到的数据也不会返回。

20.spark master在使用zookeeper进行HA时，有哪些元数据保存在zookeeper？

答：spark通过这个参数spark.deploy.zookeeper.dir指定master元数据在zookeeper中保存的位置，包括worker,master,application,executors.standby节点要从zk中获得元数据信息，恢复集群运行状态，才能对外继续提供服务，作业提交资源申请等，在恢复前是不能接受请求的，另外，master切换需要注意两点：

1.在master切换的过程中，所有的已经在运行的程序皆正常运行，因为spark application在运行前就已经通过cluster manager获得了计算资源，所以在运行时job本身的调度和处理master是没有任何关系的；

2.在master的切换过程中唯一的影响是不能提交新的job，一方面不能提交新的应用程序给集群，因为只有Active master才能接受新的程序的提交请求，另外一方面，已经运行的程序也不能action操作触发新的job提交请求。

21.spark master HA主从切换过程不会影响集群已有的作业运行，为什么？

答：因为程序在运行之前，已经向集群申请过资源，这些资源已经提交给driver了，也就是说已经分配好资源了，这是粗粒度分配，一次性分配好资源后不需要再关心资源分配，在运行时让driver和executor自动交互，弊端是如果资源分配太多，任务运行完不会很快释放，造成资源浪费，这里不适用细粒度分配的原因是因为任务提交太慢。

27.简单说一下hadoop和spark的shuffle相同和差异？

答：1）从 high-level 的角度来看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() （Spark 里可能是后续的一系列操作）。

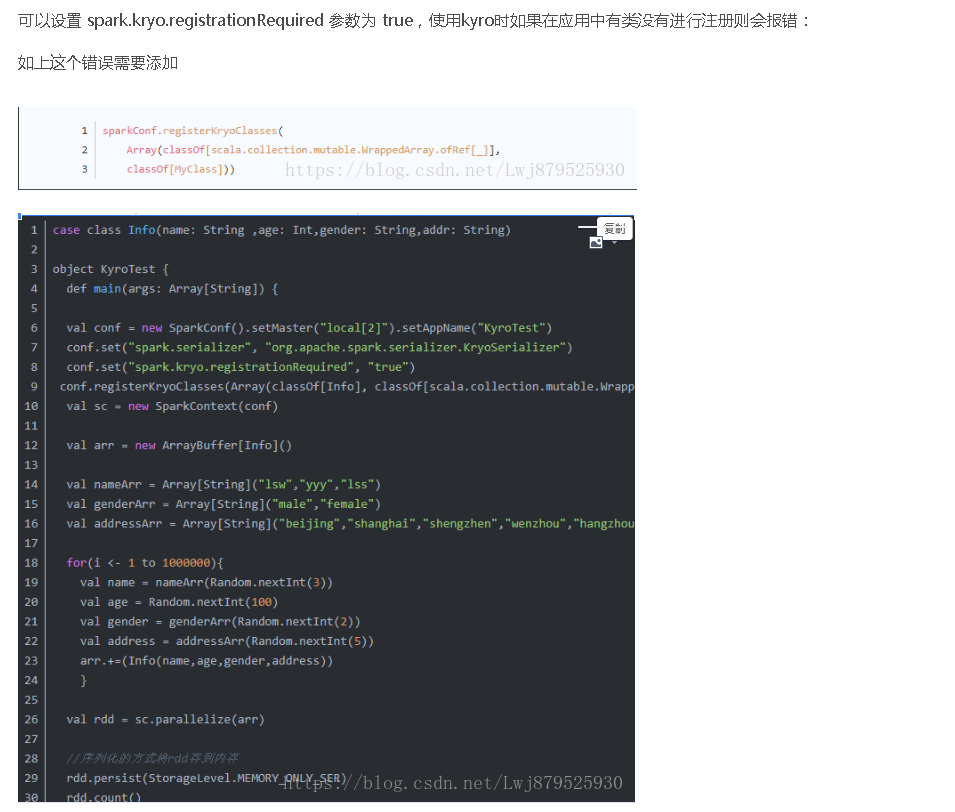
2）从 low-level 的角度来看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。目前的 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是Spark 1.1的用户，可以将spark.shuffle.manager设置为sort，则会对数据进行排序。在Spark 1.2中，sort将作为默认的Shuffle实现。

3）从实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。

如果我们将 map 端划分数据、持久化数据的过程称为 shuffle write，而将 reducer 读入数据、aggregate 数据的过程称为 shuffle read。那么在 Spark 中，问题就变为怎么在 job 的逻辑或者物理执行图中加入 shuffle write 和 shuffle read 的处理逻辑？以及两个处理逻辑应该怎么高效实现？

Shuffle write由于不要求数据有序，shuffle write 的任务很简单：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。





2.下面哪个端口不是 spark 自带服务的端口 (C )

A.8080 B.4040 C.8090 D.18080

备注：8080：spark集群web ui端口，4040：sparkjob监控端口，18080：jobhistory端口

4. Spark Job 默认的调度模式 (A )

A FIFO   B FAIR

C 无   D 运行时指定

备注：Spark中的调度模式主要有两种：FIFO和FAIR。默认情况下Spark的调度模式是FIFO（先进先出），谁先提交谁先执行，后面的任务需要等待前面的任务执行。而FAIR（公平调度）模式支持在调度池中为任务进行分组，不同的调度池权重不同，任务可以按照权重来决定执行顺序。使用哪种调度器由参数spark.scheduler.mode来设置，可选的参数有FAIR和FIFO，默认是FIFO。

5.哪个不是本地模式运行的条件 ( D)

A spark.localExecution.enabled=true

B 显式指定本地运行

C finalStage 无父 Stage

D partition默认值

备注：【问题】Spark在windows能跑集群模式吗？

我认为是可以的，但是需要详细了解cmd命令行的写法。目前win下跑spark的单机模式是没有问题的。

7. 关于广播变量，下面哪个是错误的 (D )

A 任何函数调用    B 是只读的

C 存储在各个节点    D 存储在磁盘或 HDFS

8. 关于累加器，下面哪个是错误的 (D )

A 支持加法 B 支持数值类型

C 可并行 D 不支持自定义类型

10.Stage 的 Task 的数量由什么决定 (A )

A Partition B Job C Stage D TaskScheduler

13.spark 的 master 和 worker 通过什么方式进行通信的？ (D )

A http B nio C netty D Akka

备注：从spark1.3.1之后，netty完全代替 了akka

一直以来，基于Akka实现的RPC通信框架是Spark引以为豪的主要特性，也是与Hadoop等分布式计算框架对比过程中一大亮点，但是时代和技术都在演化，从Spark1.3.1版本开始，为了解决大数据块（如shuffle）的传输问题，Spark引入了Netty通信框架，到了1.6.0版本，Netty居然完全取代了Akka，承担Spark内部所有的RPC通信以及数据流传输。

那么Akka又是什么东西？从Akka出现背景来说，它是基于Actor的RPC通信系统，它的核心概念也是Message，它是基于协程的，性能不容置疑；基于scala的偏函数，易用性也没有话说，但是它毕竟只是RPC通信，无法适用大的package/stream的数据传输，这也是Spark早期引入Netty的原因。

那么Netty为什么可以取代Akka？首先不容置疑的是Akka可以做到的，Netty也可以做到，但是Netty可以做到，Akka却无法做到，原因是啥？在软件栈中，Akka相比Netty要Higher一点，它专门针对RPC做了很多事情，而Netty相比更加基础一点，可以为不同的应用层通信协议（RPC，FTP，HTTP等）提供支持，在早期的Akka版本，底层的NIO通信就是用的Netty；其次一个优雅的工程师是不会允许一个系统中容纳两套通信框架，恶心！最后，虽然Netty没有Akka协程级的性能优势，但是Netty内部高效的Reactor线程模型，无锁化的串行设计，高效的序列化，零拷贝，内存池等特性也保证了Netty不会存在性能问题。

那么Spark是怎么用Netty来取代Akka呢？一句话，利用偏函数的特性，基于Netty“仿造”出一个简约版本的Actor模型！！

15 spark.deploy.recoveryMode 不支持那种 (D )

A.ZooKeeper B. FileSystem

D NONE D Hadoop

18.hive 的元数据存储在 derby 和 MySQL 中有什么区别 (B )

A.没区别 B.多会话

C.支持网络环境 D数据库的区别

备注：  Hive 将元数据存储在 RDBMS 中，一般常用 MySQL 和 Derby。默认情况下，Hive 元数据保存在内嵌的 Derby 数据库中，只能允许一个会话连接，只适合简单的测试。实际生产环境中不适用， 为了支持多用户会话，则需要一个独立的元数据库，使用 MySQL 作为元数据库，Hive 内部对 MySQL 提供了很好的支持。

内置的derby主要问题是并发性能很差，可以理解为单线程操作。

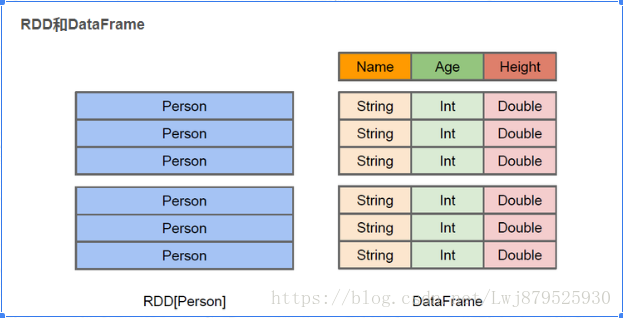
Derby还有一个特性。更换目录执行操作，会找不到相关表等

19.DataFrame 和 RDD 最大的区别 (B )

A.科学统计支持 B.多了 schema

C.存储方式不一样 D.外部数据源支持

备注：



上图直观体现了RDD与DataFrame的区别：左侧的RDD[Person]虽然以Person为类型参数，但Spark框架本身不了解Person类的内部结构。而右侧的DataFrame却提供了详细的结构信息，使得Spark SQL可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。DataFrame多了数据的结构信息，即schema。RDD是分布式的Java对象的集合。DataFrame是分布式的Row对象的集合。DataFrame除了提供了比RDD更丰富的算子以外，更重要的特点是提升执行效率、减少数据读取以及执行计划的优化，比如filter下推、裁剪等。

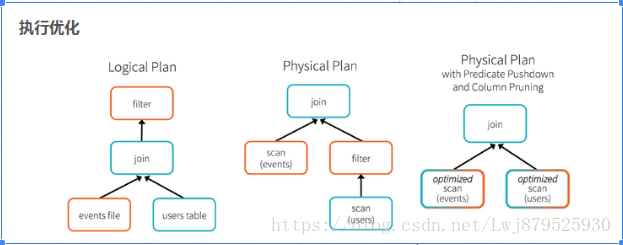
提升执行效率： RDD API是函数式的，强调不变性，在大部分场景下倾向于创建新对象而不是修改老对象。这一特点虽然带来了干净整洁的API，却也使得Spark应用程序在运行期倾向于创建大量临时对象，对GC造成压力。在现有RDD API的基础之上，我们固然可以利用mapPartitions方法来重载RDD单个分片内的数据创建方式，用复用可变对象的方式来减小对象分配和GC的开销，但这牺牲了代码的可读性，而且要求开发者对Spark运行时机制有一定的了解，门槛较高。另一方面，Spark SQL在框架内部已经在各种可能的情况下尽量重用对象，这样做虽然在内部会打破了不变性，但在将数据返回给用户时，还会重新转为不可变数据。利用 DataFrame API进行开发，可以免费地享受到这些优化效果。

减少数据读取：分析大数据，最快的方法就是 ——忽略它。这里的“忽略”并不是熟视无睹，而是根据查询条件进行恰当的剪枝。

     上文讨论分区表时提到的分区剪 枝便是其中一种——当查询的过滤条件中涉及到分区列时，我们可以根据查询条件剪掉肯定不包含目标数据的分区目录，从而减少IO。

      对于一些“智能”数据格 式，Spark SQL还可以根据数据文件中附带的统计信息来进行剪枝。简单来说，在这类数据格式中，数据是分段保存的，每段数据都带有最大值、最小值、null值数量等 一些基本的统计信息。当统计信息表名某一数据段肯定不包括符合查询条件的目标数据时，该数据段就可以直接跳过（例如某整数列a某段的最大值为100，而查询条件要求a > 200）。

      此外，Spark SQL也可以充分利用RCFile、ORC、Parquet等列式存储格式的优势，仅扫描查询真正涉及的列，忽略其余列的数据。



 为了说明查询优化，我们来看上图展示的人口数据分析的示例。图中构造了两个DataFrame，将它们join之后又做了一次filter操作。如果原封不动地执行这个执行计划，最终的执行效率是不高的。因为join是一个代价较大的操作，也可能会产生一个较大的数据集。如果我们能将filter下推到 join下方，先对DataFrame进行过滤，再join过滤后的较小的结果集，便可以有效缩短执行时间。而Spark SQL的查询优化器正是这样做的。简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程。

得到的优化执行计划在转换成物 理执行计划的过程中，还可以根据具体的数据源的特性将过滤条件下推至数据源内。最右侧的物理执行计划中Filter之所以消失不见，就是因为溶入了用于执行最终的读取操作的表扫描节点内。

对于普通开发者而言，查询优化 器的意义在于，即便是经验并不丰富的程序员写出的次优的查询，也可以被尽量转换为高效的形式予以执行。

RDD和Dataset

DataSet以Catalyst逻辑执行计划表示，并且数据以编码的二进制形式被存储，不需要反序列化就可以执行sorting、shuffle等操作。

DataSet创立需要一个显式的Encoder，把对象序列化为二进制，可以把对象的scheme映射为Spark

SQl类型，然而RDD依赖于运行时反射机制。

DataFrame和Dataset

      Dataset可以认为是DataFrame的一个特例，主要区别是Dataset每一个record存储的是一个强类型值而不是一个Row。因此具有如下三个特点：

DataSet可以在编译时检查类型

并且是面向对象的编程接口。

20.Master 的 ElectedLeader 事件后做了哪些操作 (D )

A. 通知 driver B.通知 worker

C.注册 application D.直接 ALIVE

34.cache后面能不能接其他算子,它是不是action操作？

答：cache可以接其他算子，但是接了算子之后，起不到缓存应有的效果，因为会重新触发cache。

cache不是action操作

35.reduceByKey是不是action？

答：不是，很多人都会以为是action，reduce rdd是action

36.数据本地性是在哪个环节确定的？

具体的task运行在那他机器上，dag划分stage的时候确定的

37.RDD的弹性表现在哪几点？

1）自动的进行内存和磁盘的存储切换；

2）基于Lingage的高效容错；

3）task如果失败会自动进行特定次数的重试；

4）stage如果失败会自动进行特定次数的重试，而且只会计算失败的分片；

5）checkpoint和persist，数据计算之后持久化缓存

6）数据调度弹性，DAG TASK调度和资源无关

7）数据分片的高度弹性，a.分片很多碎片可以合并成大的，b.par

39.RDD通过Linage（记录数据更新）的方式为何很高效？

1）lazy记录了数据的来源，RDD是不可变的，且是lazy级别的，且rDD

之间构成了链条，lazy是弹性的基石。由于RDD不可变，所以每次操作就

产生新的rdd，不存在全局修改的问题，控制难度下降，所有有计算链条

将复杂计算链条存储下来，计算的时候从后往前回溯

900步是上一个stage的结束，要么就checkpoint

2）记录原数据，是每次修改都记录，代价很大

如果修改一个集合，代价就很小，官方说rdd是

粗粒度的操作，是为了效率，为了简化，每次都是

操作数据集合，写或者修改操作，都是基于集合的

rdd的写操作是粗粒度的，rdd的读操作既可以是粗粒度的

也可以是细粒度，读可以读其中的一条条的记录。

3）简化复杂度，是高效率的一方面，写的粗粒度限制了使用场景

如网络爬虫，现实世界中，大多数写是粗粒度的场景

40.RDD有哪些缺陷？

1）不支持细粒度的写和更新操作（如网络爬虫），spark写数据是粗粒度的

所谓粗粒度，就是批量写入数据，为了提高效率。但是读数据是细粒度的也就是

说可以一条条的读

2）不支持增量迭代计算，Flink支持

45. 你所理解的Spark的shuffle过程？

答：从下面三点去展开

1）shuffle过程的划分

2）shuffle的中间结果如何存储

3）shuffle的数据如何拉取过来

可以参考这篇博文：<http://www.cnblogs.com/jxhd1/p/6528540.html>



Shuffle后续优化方向：通过上面的介绍，我们了解到，Shuffle过程的主要存储介质是磁盘，尽量的减少IO是Shuffle的主要优化方向。我们脑海中都有那个经典的存储金字塔体系，Shuffle过程为什么把结果都放在磁盘上，那是因为现在内存再大也大不过磁盘，内存就那么大，还这么多张嘴吃，当然是分配给最需要的了。如果具有“土豪”内存节点，减少Shuffle IO的最有效方式无疑是尽量把数据放在内存中。下面列举一些现在看可以优化的方面，期待经过我们不断的努力，TDW计算引擎运行地更好。

MapReduce Shuffle后续优化方向：压缩：对数据进行压缩，减少写读数据量；

减少不必要的排序：并不是所有类型的Reduce需要的数据都是需要排序的，排序这个nb的过程如果不需要最好还是不要的好；

内存化：Shuffle的数据不放在磁盘而是尽量放在内存中，除非逼不得已往磁盘上放；当然了如果有性能和内存相当的第三方存储系统，那放在第三方存储系统上也是很好的；这个是个大招；

网络框架：netty的性能据说要占优了；

本节点上的数据不走网络框架：对于本节点上的Map输出，Reduce直接去读吧，不需要绕道网络框架。

Spark Shuffle后续优化方向：Spark作为MapReduce的进阶架构，对于Shuffle过程已经是优化了的，特别是对于那些具有争议的步骤已经做了优化，但是Spark的Shuffle对于我们来说在一些方面还是需要优化的。

压缩：对数据进行压缩，减少写读数据量；

内存化：Spark历史版本中是有这样设计的：Map写数据先把数据全部写到内存中，写完之后再把数据刷到磁盘上；考虑内存是紧缺资源，后来修改成把数据直接写到磁盘了；对于具有较大内存的集群来讲，还是尽量地往内存上写吧，内存放不下了再放磁盘。

47. 对于Spark中的数据倾斜问题你有什么好的方案？

1）前提是定位数据倾斜，是OOM了，还是任务执行缓慢，看日志，看WebUI

2)解决方法，有多个方面

· 避免不必要的shuffle，如使用广播小表的方式，将reduce-side-join提升为map-side-join

·分拆发生数据倾斜的记录，分成几个部分进行，然后合并join后的结果

·改变并行度，可能并行度太少了，导致个别task数据压力大

·两阶段聚合，先局部聚合，再全局聚合

·自定义paritioner，分散key的分布，使其更加均匀

详细解决方案参考博文《Spark数据倾斜优化方法》<https://mp.weixin.qq.com/s?__biz=MzIzNzI1NzY3Nw==&mid=2247484221&idx=1&sn=7e20f08bfb490b91f0920aefb29ca271&chksm=e8ca159fdfbd9c89f610dd230e07f414521b4dd13018994ee9b873421d1e8efcdc535c810225&scene=21#wechat_redirect>

48.RDD创建有哪几种方式？

1).使用程序中的集合创建rdd

2).使用本地文件系统创建rdd

3).使用hdfs创建rdd，

4).基于数据库db创建rdd

5).基于Nosql创建rdd，如hbase

6).基于s3创建rdd，

7).基于数据流，如socket创建rdd

如果只回答了前面三种，是不够的，只能说明你的水平还是入门级的，实践过程中有很多种创建方式。

49.Spark并行度怎么设置比较合适

答：spark并行度，每个core承载2~4个partition,如，32个core，那么64~128之间的并行度，也就是

设置64~128个partion，并行读和数据规模无关，只和内存使用量和cpu使用

时间有关

50.Spark中数据的位置是被谁管理的？

答：每个数据分片都对应具体物理位置，数据的位置是被blockManager，无论

数据是在磁盘，内存还是tacyan，都是由blockManager管理

51.Spark的数据本地性有哪几种？

答：Spark中的数据本地性有三种：

a.PROCESS\_LOCAL是指读取缓存在本地节点的数据

b.NODE\_LOCAL是指读取本地节点硬盘数据

c.ANY是指读取非本地节点数据

通常读取数据PROCESS\_LOCAL>NODE\_LOCAL>ANY，尽量使数据以PROCESS\_LOCAL或NODE\_LOCAL方式读取。其中PROCESS\_LOCAL还和cache有关，如果RDD经常用的话将该RDD cache到内存中，注意，由于cache是lazy的，所以必须通过一个action的触发，才能真正的将该RDD cache到内存中。

52.rdd有几种操作类型？

1）transformation，rdd由一种转为另一种rdd

2）action，

3）cronroller，crontroller是控制算子,cache,persist，对性能和效率的有很好的支持

三种类型，不要回答只有2中操作

53.Spark如何处理不能被序列化的对象？

将不能序列化的内容封装成object

54.collect功能是什么，其底层是怎么实现的？

答：driver通过collect把集群中各个节点的内容收集过来汇总成结果，collect返回结果是Array类型的，collect把各个节点上的数据抓过来，抓过来数据是Array型，collect对Array抓过来的结果进行合并，合并后Array中只有一个元素，是tuple类型（KV类型的）的。

55.Spaek程序执行，有时候默认为什么会产生很多task，怎么修改默认task执行个数？

答：1）因为输入数据有很多task，尤其是有很多小文件的时候，有多少个输入

block就会有多少个task启动；2）spark中有partition的概念，每个partition都会对应一个task，task越多，在处理大规模数据的时候，就会越有效率。不过task并不是越多越好，如果平时测试，或者数据量没有那么大，则没有必要task数量太多。3）参数可以通过spark\_home/conf/spark-default.conf配置文件设置:

spark.sql.shuffle.partitions 50 spark.default.parallelism 10

第一个是针对spark sql的task数量

第二个是非spark sql程序设置生效

56.为什么Spark Application在没有获得足够的资源，job就开始执行了，可能会导致什么什么问题发生?

答：会导致执行该job时候集群资源不足，导致执行job结束也没有分配足够的资源，分配了部分Executor，该job就开始执行task，应该是task的调度线程和Executor资源申请是异步的；如果想等待申请完所有的资源再执行job的：需要将spark.scheduler.maxRegisteredResourcesWaitingTime设置的很大；spark.scheduler.minRegisteredResourcesRatio 设置为1，但是应该结合实际考虑

否则很容易出现长时间分配不到资源，job一直不能运行的情况。

59.Spark为什么要持久化，一般什么场景下要进行persist操作？

为什么要进行持久化？

spark所有复杂一点的算法都会有persist身影,spark默认数据放在内存，spark很多内容都是放在内存的，非常适合高速迭代，1000个步骤

只有第一个输入数据，中间不产生临时数据，但分布式系统风险很高，所以容易出错，就要容错，rdd出错或者分片可以根据血统算出来，如果没有对父rdd进行persist 或者cache的化，就需要重头做。

以下场景会使用persist

1）某个步骤计算非常耗时，需要进行persist持久化

2）计算链条非常长，重新恢复要算很多步骤，很好使，persist

3）checkpoint所在的rdd要持久化persist，

lazy级别，框架发现有checnkpoint，checkpoint时单独触发一个job，需要重算一遍，checkpoint前

要持久化，写个rdd.cache或者rdd.persist，将结果保存起来，再写checkpoint操作，这样执行起来会非常快，不需要重新计算rdd链条了。checkpoint之前一定会进行persist。

4）shuffle之后为什么要persist，shuffle要进性网络传输，风险很大，数据丢失重来，恢复代价很大

5）shuffle之前进行persist，框架默认将数据持久化到磁盘，这个是框架自动做的。

60.为什么要进行序列化

序列化可以减少数据的体积，减少存储空间，高效存储和传输数据，不好的是使用的时候要反序列化，非常消耗CPU

61.介绍一下join操作优化经验？

答：join其实常见的就分为两类： map-side join 和  reduce-side join。当大表和小表join时，用map-side join能显著提高效率。将多份数据进行关联是数据处理过程中非常普遍的用法，不过在分布式计算系统中，这个问题往往会变的非常麻烦，因为框架提供的 join 操作一般会将所有数据根据 key 发送到所有的 reduce 分区中去，也就是 shuffle 的过程。造成大量的网络以及磁盘IO消耗，运行效率极其低下，这个过程一般被称为 reduce-side-join。如果其中有张表较小的话，我们则可以自己实现在 map 端实现数据关联，跳过大量数据进行 shuffle 的过程，运行时间得到大量缩短，根据不同数据可能会有几倍到数十倍的性能提升。

备注：这个题目面试中非常非常大概率见到，务必搜索相关资料掌握，这里抛砖引玉。

62.介绍一下cogroup rdd实现原理，你在什么场景下用过这个rdd？

答：cogroup的函数实现:这个实现根据两个要进行合并的两个RDD操作,生成一个CoGroupedRDD的实例,这个RDD的返回结果是把相同的key中两个RDD分别进行合并操作,最后返回的RDD的value是一个Pair的实例,这个实例包含两个Iterable的值,第一个值表示的是RDD1中相同KEY的值,第二个值表示的是RDD2中相同key的值.由于做cogroup的操作,需要通过partitioner进行重新分区的操作,因此,执行这个流程时,需要执行一次shuffle的操作(如果要进行合并的两个RDD的都已经是shuffle后的rdd,同时他们对应的partitioner相同时,就不需要执行shuffle,)，

场景：表关联查询

HBASE

#### 1 每天百亿数据存入HBase，如何保证数据的存储正确和在规定的时间里全部录入完毕，不残留数据

答：看到这个题目的时候我们要思考的是它在考查什么知识点？  
我们来看看要求：  
1）百亿数据：证明数据量非常大  
2）存入HBase：证明是跟HBase的写入数据有关  
3）保证数据的正确：要设计正确的数据结构保证正确性  
4）在规定时间内完成：对存入速度是有要求的

那么针对以上的四个问题我们来一一分析  
1）数据量百亿条，什么概念呢？假设一整天60x60x24 = 86400秒都在写入数据，那么每秒的写入条数高达100万条，HBase当然是支持不了每秒百万条数据的，所以这百亿条数据可能不是通过实时地写入，而是批量地导入。批量导入推荐使用BulkLoad方式（[推荐阅读：Spark之读写HBase](https://www.jianshu.com/p/8bd9862f8567)），性能是普通写入方式几倍以上  
2）存入HBase：普通写入是用JavaAPI put来实现，批量导入推荐使用BulkLoad  
3）保证数据的正确：这里需要考虑RowKey的设计、预建分区和列族设计等问题  
4）在规定时间内完成也就是存入速度不能过慢，并且当然是越快越好，使用BulkLoad

#### 3 HBase优化方法

答：优化手段主要有以下四个方面

##### 1）减少调整

减少调整这个如何理解呢？HBase中有几个内容会动态调整，如region（分区）、HFile，所以通过一些方法来减少这些会带来I/O开销的调整

* Region  
  如果没有预建分区的话，那么随着region中条数的增加，region会进行分裂，这将增加I/O开销，所以解决方法就是根据你的RowKey设计来进行预建分区，减少region的动态分裂
* HFile  
  HFile是数据底层存储文件，在每个memstore进行刷新时会生成一个HFile，当HFile增加到一定程度时，会将属于一个region的HFile进行合并，这个步骤会带来开销但不可避免，但是合并后HFile大小如果大于设定的值，那么HFile会重新分裂。为了减少这样的无谓的I/O开销，建议估计项目数据量大小，给HFile设定一个合适的值

##### 2）减少启停

数据库事务机制就是为了更好地实现批量写入，较少数据库的开启关闭带来的开销，那么HBase中也存在频繁开启关闭带来的问题。

* 关闭Compaction，在闲时进行手动Compaction  
  因为HBase中存在Minor Compaction和Major Compaction，也就是对HFile进行合并，所谓合并就是I/O读写，大量的HFile进行肯定会带来I/O开销，甚至是I/O风暴，所以为了避免这种不受控制的意外发生，建议关闭自动Compaction，在闲时进行compaction
* 批量数据写入时采用BulkLoad  
  如果通过HBase-Shell或者JavaAPI的put来实现大量数据的写入，那么性能差是肯定并且还可能带来一些意想不到的问题，所以当需要写入大量离线数据时建议使用BulkLoad

##### 3）减少数据量

虽然我们是在进行大数据开发，但是如果可以通过某些方式在保证数据准确性同时减少数据量，何乐而不为呢？

* 开启过滤，提高查询速度  
  开启BloomFilter，BloomFilter是列族级别的过滤，在生成一个StoreFile同时会生成一个MetaBlock，用于查询时过滤数据
* 使用压缩：一般推荐使用Snappy和LZO压缩

###### 4）合理设计

在一张HBase表格中RowKey和ColumnFamily的设计是非常重要，好的设计能够提高性能和保证数据的准确性

* RowKey设计：应该具备以下几个属性
  + 散列性：散列性能够保证相同相似的rowkey聚合，相异的rowkey分散，有利于查询
* 简短性：rowkey作为key的一部分存储在HFile中，如果为了可读性将rowKey设计得过长，那么将会增加存储压力
* 唯一性：rowKey必须具备明显的区别性
* 业务性：举些例子
  + 假如我的查询条件比较多，而且不是针对列的条件，那么rowKey的设计就应该支持多条件查询
  + 如果我的查询要求是最近插入的数据优先，那么rowKey则可以采用叫上Long.Max-时间戳的方式，这样rowKey就是递减排列
* 列族的设计  
  列族的设计需要看应用场景
  + 多列族设计的优劣
    - 优势：HBase中数据时按列进行存储的，那么查询某一列族的某一列时就不需要全盘扫描，只需要扫描某一列族，减少了读I/O；其实多列族设计对减少的作用不是很明显，适用于读多写少的场景
  + 劣势：降低了写的I/O性能。原因如下：数据写到store以后是先缓存在memstore中，同一个region中存在多个列族则存在多个store，每个store都一个memstore，当其实memstore进行flush时，属于同一个region

#### 4 HBase中RowFilter和BloomFilter原理

答：

##### 1）RowFilter原理简析

RowFilter顾名思义就是对rowkey进行过滤，那么rowkey的过滤无非就是相等（EQUAL）、大于(GREATER)、小于(LESS)，大于等于(GREATER\_OR\_EQUAL)，小于等于(LESS\_OR\_EQUAL)和不等于(NOT\_EQUAL)几种过滤方式。Hbase中的RowFilter采用比较符结合比较器的方式来进行过滤。

比较器的类型如下：

* BinaryComparator
* BinaryPrefixComparator
* NullComparator
* BitComparator
* RegexStringComparator
* SubStringComparator

###### 2）BloomFilter原理简析

* 主要功能：提供随机读的性能
* 存储开销：BloomFilter是列族级别的配置，一旦表格中开启BloomFilter，那么在生成StoreFile时同时会生成一份包含BloomFilter结构的文件MetaBlock，所以会增加一定的存储开销和内存开销
* 粒度控制：ROW和ROWCOL
* BloomFilter的原理  
  简单说一下BloomFilter原理
  + 内部是一个bit数组，初始值均为0
  + 插入元素时对元素进行hash并且映射到数组中的某一个index，将其置为1，再进行多次不同的hash算法，将映射到的index置为1，同一个index只需要置1次。
  + 查询时使用跟插入时相同的hash算法，如果在对应的index的值都为1，那么就可以认为该元素可能存在，注意，只是可能存在
  + 所以BlomFilter只能保证过滤掉不包含的元素，而不能保证误判包含
* 设置：在建表时对某一列设置BloomFilter即可

#### 5 HBase的导入导出方式

答：  
1）导入：bin/hbase org.apache.hadoop.hbase.mapreduce.Driver import 表名 路径

路径：来源

* 本地路径 file:///path
* HDFS [hdfs://cluster1/path](https://link.jianshu.com?t=hdfs:/cluster1/path)

2）导出：bin/hbase org.apache.hadoop.hbase.mapreduce.Driver export 表名 路径

路径：目的地

* 本地路径 file:///path
* HDFS [hdfs://cluster1/path](https://link.jianshu.com?t=hdfs:/cluster1/path)

#### 6 Region如何预建分区

##### 预建分区的方法很简单，有以下两种

* hbase shell
* create 't1', 'f1',SPLITS=>['10','20','30']
* create 't1','f1',SPLITS\_FILE =>'splits.txt'
* Java API
  + 创建一个byte[][] splitKeys = {{1,2,3},{4,5,6}}
  + admin.createTable(tableDesc,splitKeys)

##### 预建分区的难点在于key如何设计，分多少个和如何分的问题，那么下面我们就对这三个问题一一分析：

1）如何设计Key，我们设计Key的原则就是要让Key足够散列，但同时又要保持Key的长度适中，这里给出一个方法，样本取自[Spark读写HBase](https://www.jianshu.com/p/8bd9862f8567)中的数据

01055HAXMTXG10100001@KEY\_VOLTAGE\_TEC\_PWR@1.60@1.62@1.75@1.55

我想要的rowKey是：01055HAXMTXG10100001KEY\_VOLTAGE\_TEC\_PWR

但是很明显这样肯定是不会足够散列的，那么我们可以对上面那个Key进行MD5，然后取前面三个字符（为了更加散列，可以取1,3,5或者其他组合）再加上原来的Key

DigestUtils.md5Hex(x(0)+x(1)).substring(0,3)+x(0)+x(1)

这样的话我们就可以得到足够散列的数据，并且MD5取得的是十六进制字符串，那么Key的范围就是（0,0,0）至（f，f，f）

2）分多少个：这个需要我们就要根据我们集群规模来进行安排，假设我们有5regionServer，每个regionServer有20个region，那么总共就是100个region，最后的工作就是将000-fff分成100份。

3）如何分：就是生成一个byte[][]，用于创建表格，这个我将会些一篇另外的博文来介绍，敬请期待

**7 HRegionServer宕机如何处理？**

1）ZooKeeper会监控HRegionServer的上下线情况，当ZK发现某个HRegionServer宕机之后会通知HMaster进行失效备援；  
2）该HRegionServer会停止对外提供服务，就是它所负责的region暂时停止对外提供服务  
3）HMaster会将该HRegionServer所负责的region转移到其他HRegionServer上，并且会对HRegionServer上存在memstore中还未持久化到磁盘中的数据进行恢复  
4）这个恢复的工作是由WAL重播来完成，这个过程如下：

* wal实际上就是一个文件，存在/hbase/WAL/对应RegionServer路径下
* 宕机发生时，读取该RegionServer所对应的路径下的wal文件，然后根据不同的region切分成不同的临时文件recover.edits
* 当region被分配到新的RegionServer中，RegionServer读取region时会进行是否存在recover.edits，如果有则进行恢复

#### 8 HBase简单读写流程

##### 读：

找到要读取数据的region所在的RegionServer，然后按照以下顺序进行读取：先去BlockCache读取，若BlockCache没有，则到Memstore读取，若MemStore中没有，则到HFile中读取。

##### 写：

找到要写入数据的region所在的RegionServer，然后将数据先写到WAL中，然后再将数据写到MemStore等待刷新，回复客户端写入完成。

#### 9 HBase和Hive的对比

|  | **HBase** | **Hive** |
| --- | --- | --- |
| 类型 | 列式数据库 | 数据仓库 |
| 内部机制 | 数据库引擎 | MapReduce |
| 增删改查 | 都支持 | 只支持导入和查询 |
| Schema | 只需要预先定义列族，不需要具体到列列可以动态修改 | 需要预先定义表格 |
| 应用场景 | 实时 | 离线处理 |
| 特点 | 以K-V形式存储 | 类SQL |

#### 10 HBase首次读写流程

* Client从ZooKeeper中读取hbase:meta表
* Client从hbase:meta中获取想要操作的region的位置信息，并且将hbase:meta缓存在Client端，用于后续的操作
* 当一个RegionServer宕机而执行重定位之后，Client需要重新获取新的hase:meta信息进行缓存

#### 11 HBase搭建过程中需要注意什么？

##### hbase-env.sh的配置

* 是否使用外部ZooKeeper，这个一般使用Hadoop集群的ZooKeeper集群即可。
  + HBASE\_MANAGES\_ZK=false

##### hbase-site.sh的配置

* hbase.zookeeper.quorum="host1:2181,host2:2181"

**hbase 的特点是什么**

 (1) Hbase一个分布式的基于列式存储的数据库,基于Hadoop的hdfs存储，zookeeper进行管理。

(2) Hbase适合存储半结构化或非结构化数据，对于数据结构字段不够确定或者杂乱无章很难按一个概念去抽取的数据。

(3) Hbase为null的记录不会被存储.

(4)基于的表包含rowkey，时间戳，和列族。新写入数据时，时间戳更新，同时可以查询到以前的版本.

(5) hbase是主从架构。hmaster作为主节点，hregionserver作为从节点。

**描述Hbase的rowKey的设计原则**

Rowkey长度原则

Rowkey 是一个二进制码流，Rowkey 的长度被很多开发者建议说设计在10~100 个字节，

不过建议是越短越好，不要超过16 个字节。

原因如下：

（1）数据的持久化文件HFile 中是按照KeyValue 存储的，如果Rowkey 过长比如100 个

字节，1000 万列数据光Rowkey 就要占用100\*1000 万=10 亿个字节，将近1G 数据，这会极

大影响HFile 的存储效率；

（2）MemStore 将缓存部分数据到内存，如果Rowkey 字段过长内存的有效利用率会降

低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey 的字节长度越短越好。

（3）目前操作系统是都是64 位系统，内存8 字节对齐。控制在16 个字节，8 字节的

整数倍利用操作系统的最佳特性。

Rowkey散列原则

如果Rowkey 是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将Rowkey

的高位作为散列字段，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个

Regionserver 实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息将产生所有

新数据都在一个 RegionServer 上堆积的热点现象，这样在做数据检索的时候负载将会集中

在个别RegionServer，降低查询效率。

Rowkey唯一原则

必须在设计上保证其唯一性。

**描述Hbase中scan和get的功能以及实现的异同**

HBase的查询实现只提供两种方式：

1、按指定RowKey 获取唯一一条记录，get方法（org.apache.hadoop.hbase.client.Get）

Get 的方法处理分两种 : 设置了ClosestRowBefore 和没有设置的rowlock .主要是用来保证行的事务性，即每个get 是以一个row 来标记的.一个row中可以有很多family 和column.

2、按指定的条件获取一批记录，scan方法(org.apache.Hadoop.hbase.client.Scan）实现条件查询功能使用的就是scan 方式.

1)scan 可以通过setCaching 与setBatch 方法提高速度(以空间换时间)；

2)scan 可以通过setStartRow 与setEndRow 来限定范围([start，end)start 是闭区间，

end 是开区间)。范围越小，性能越高。

3)、scan 可以通过setFilter 方法添加过滤器，这也是分页、多条件查询的基础。

**请描述Hbase中scan对象的setCache和setBatch 方法的使用.**

为设置获取记录的列个数，默认无限制，也就是返回所有的列.每次从服务器端读取的行数，默认为配置文件中设置的值.

**23 请详细描述Hbase中一个Cell 的结构**

HBase 中通过row 和columns 确定的为一个存贮单元称为cell。

Cell：由{row key, column(=<family> + <label>), version}唯一确定的单元。cell 中的数

据是没有类型的，全部是字节码形式存贮。

**请描述如何解决Hbase中region太小和region太大带来的冲突.**

Region过大会发生多次compaction，将数据读一遍并重写一遍到hdfs 上，占用io，region过小会造成多次split，region 会下线，影响访问服务，调整hbase.hregion.max.filesize 为256m.

**以 start-hbase.sh 为起点，Hbase 启动的流程是什么？**

start-hbase.sh 的流程如下：

1.运行 hbase-config.sh

hbase-config.sh的作用：

1>.装载相关配置，如HBASE\_HOME目录，conf目录，regionserver机器列表，JAVA\_HOME 目录等，它会调用$HBASE\_HOME/conf/hbase-env.sh .

2>.解析参数（0.96 版本及以后才可以带唯一参数 autorestart，作用就是重启）

3>.调用 hbase-daemon.sh 来启动 master.

4>.调用 hbase-daemons.sh 来启动 regionserver zookeeper master-backup.

2.hbase-env.sh 的作用：

主要是配置 JVM 及其 GC 参数，还可以配置 log 目录及参数，配置是否需要 hbase 管

理 ZK，配置进程 id 目录等.

3.hbase-daemons.sh 的作用：根据需要启动的进程，

如 zookeeper,则调用 zookeepers.sh

如 regionserver，则调用 regionservers.sh

如 master-backup，则调用 master-backup.sh

4.zookeepers.sh 的作用：

如果 hbase-env.sh 中的 HBASE\_MANAGES\_ZK"="true"，那么通过ZKServerTool这个类解析xml配置文件，获取 ZK 节点列表，然后通过 SSH 向这些节点发送远程命令执行。

5.regionservers.sh 的作用：

与 zookeepers.sh 类似，通过配置文件，获取 regionserver 机器列表，然后 SSH 向这些机器发送远程命令：

6.master-backup.sh 的作用：

通过 backup-masters 这个配置文件，获取 backup-masters 机器列表,然后 SSH 向这些机器发送远程命令。

**简述 HBASE中compact用途是什么，什么时候触发，分为哪两种,有什么区别，有哪些相关配置参数？**

在hbase中每当有memstore数据flush到磁盘之后，就形成一个storefile，当storeFile的数量达到一定程度后，就需要将 storefile 文件来进行 compaction 操作。

Compact 的作用：

1>.合并文件

2>.清除过期，多余版本的数据

3>.提高读写数据的效率

HBase 中实现了两种 compaction 的方式：minor and major. 这两种 compaction 方式的区别是：

1、Minor 操作只用来做部分文件的合并操作以及包括 minVersion=0 并且设置 ttl 的过

期版本清理，不做任何删除数据、多版本数据的清理工作。

2、Major 操作是对 Region 下的HStore下的所有StoreFile执行合并操作，最终的结果是整理合并出一个文件。

## 3.hbase 的存储结构？

　　答： Hbase 中的每张表都通过行键(rowkey)按照一定的范围被分割成多个子表（HRegion），默认一个 HRegion 超过 256M 就要被分割成两个，由 HRegionServer 管理，管理哪些 HRegion由 Hmaster 分配。 HRegion 存取一个子表时，会创建一个 HRegion 对象，然后对表的每个列族（Column Family）创建一个 store 实例，每个 store 都会有 0 个或多个 StoreFile 与之对应，每个 StoreFile 都会对应一个 HFile， HFile 就是实际的存储文件，因此，一个 HRegion 还拥有一个 MemStore 实例。

## 5.解释下 hbase 实时查询的原理

　　答：实时查询，可以认为是从内存中查询，一般响应时间在 1 秒内。 HBase 的机制是数据先写入到内存中，当数据量达到一定的量（如 128M），再写入磁盘中， 在内存中，是不进行数据的更新或合并操作的，只增加数据，这使得用户的写操作只要进入内存中就可以立即返回，保证了 HBase I/O 的高性能。

## 6.列簇怎么创建比较好？(<=2)

答： rowKey 最好要创建有规则的 rowKey，即最好是有序的。 HBase 中一张表最好只创建一到两个列族比较好，因为 HBase 不能很好的处理多个列族。

## 7.描述 Hbase 中 scan 和 get 的功能以及实现的异同.

1.按指定RowKey 获取唯一一条记录， get方法（org.apache.hadoop.hbase.client.Get）Get 的方法处理分两种 : 设置了 ClosestRowBefore 和没有设置的 rowlock .主要是用来保证行的事务性，即每个 get 是以一个 row 来标记的.一个 row 中可以有很多 family 和 column.

2.按指定的条件获取一批记录， scan 方法(org.apache.Hadoop.hbase.client.Scan)实现条件查询功能使用的就是 scan 方式.1)scan 可以通过 setCaching 与 setBatch 方法提高速度(以空间换时间)； 2)scan 可以通过 setStartRow 与 setEndRow 来限定范围([start， end]start 是闭区间， end 是开区间)。范围越小，性能越高。3)scan 可以通过 setFilter 方法添加过滤器，这也是分页、多条件查询的基础。

3.全表扫描，即直接扫描整张表中所有行记录

## 11.　Hbase 内部是什么机制

　　在 HBase 中无论是增加新行还是修改已有行，其内部流程都是相同的。 HBase 接到命令后存下变化信息，或者写入失败抛出异常。默认情况下，执行写入时会写到两个地方：预写式日志（write-ahead log，也称 HLog）和 MemStore。 HBase 的默认方式是把写入动作记录在这两个地方，以保证数据持久化。只有当这两个地方的变化信息都写入并确认后，才认为写动作完成。MemStore 是内存里的写入缓冲区， HBase 中数据在永久写入硬盘之前在这里累积。当MemStore 填满后，其中的数据会刷写到硬盘，生成一个 HFile。 HFile 是 HBase 使用的底层存储格式。 HFile 对应于列族，一个列族可以有多个 HFile，但一个 HFile 不能存储多个列族的数据。在集群的每个节点上，每个列族有一个 MemStore。大型分布式系统中硬件故障很常见， HBase 也不例外。设想一下，如果 MemStore 还没有刷写，服务器就崩溃了，内存中没有写入硬盘的数据就会丢失。 HBase 的应对办法是在写动作完成之前先写入 WAL。 HBase 集群中每台服务器维护一个 WAL 来记录发生的变化。WAL 是底层文件系统上的一个文件。直到 WAL 新记录成功写入后，写动作才被认为成功完成。这可以保证 HBase 和支撑它的文件系统满足持久性。大多数情况下， HBase 使用Hadoop 分布式文件系统（HDFS）来作为底层文件系统。如果 HBase 服务器宕机，没有从 MemStore 里刷写到 HFile 的数据将可以通过回放WAL 来恢复。你不需要手工执行。 Hbase 的内部机制中有恢复流程部分来处理。每台HBase 服务器有一个 WAL，这台服务器上的所有表（和它们的列族）共享这个 WAL。你可能想到，写入时跳过 WAL 应该会提升写性能。但我们不建议禁用 WAL，除非你愿意在出问题时丢失数据。如果你想测试一下，如下代码可以禁用 WAL： 注意：不写入 WAL 会在 RegionServer 故障时增加丢失数据的风险。关闭 WAL，出现故障时 HBase 可能无法恢复数据，没有刷写到硬盘的所有写入数据都会丢失。

## 12.HBase 宕机如何处理

答：宕机分为 HMaster 宕机和 HRegisoner 宕机，如果是 HRegisoner 宕机， HMaster 会将其所管理的 region 重新分布到其他活动的 RegionServer 上，由于数据和日志都持久在 HDFS中，该操作不会导致数据丢失。所以数据的一致性和安全性是有保障的。如果是 HMaster 宕机， HMaster 没有单点问题， HBase 中可以启动多个 HMaster，通过Zookeeper 的 Master Election 机制保证总有一个 Master 运行。即 ZooKeeper 会保证总会有一个 HMaster 在对外提供服务。

## 13.导致Hbase挂掉的场景

导致Hbase挂掉的场景  
HMaster  
HMaster会出现异常(执行abort())停止的场景如下：  
1.zk异常导致的master停止服务是最常见的场景，涉及操作包含但不限于以下：  
  a)Zk链接超时，超时时间通过zookeeper.session.timeout配置，默认为3分钟, 如果fail.fast.expired.active.master配置的值为false（默认为false），则不会立即abort，而是会尝试恢复zk的过期session；  
  b)在打开region后，需要从zk中删除opened节点，如果zk有该节点，但是删除失败；  
  c)在split region过程中，从zk删除split节点时；  
  d)Master节点改变时；  
  e)从zk中创建unassigned节点时；  
  f)在下线disabled的regoin时，从zk中删除disabled的region如果发生zk异常；  
  g)还有很多操作zk的节点时如果出现异常。  
2.在assign时，如果设置region为offlined状态，但是region之前的状态不是closed或者offlined；  
3.在assign时，如果无法从.META.表中读取region信息；  
4.把新的hbase集群加入到正在运行的hbase集群时，如果zk的/hbase/unassigned节点没有数据；  
5.使用线程池批量分配region时，如果出现未被捕获的异常，实现方式如下：  
6.在启动master的服务线程时，出现了异常；  
7.在hdfs中检查hbase日志路径时，发现了dead的server时，需从hdfs中读出log，如果出现io异常需要检查hdfs文件系统，如果fsOk状态为true，但是通过FSUtils工具类进行检查时出现io异常；  
8.在校验并且分配-ROOT-的region时，如果zk异常，或者其它异常（其它异常会重试10次），比如：“-ROOT- is onlined on the dead server”。   
  
HRegionServer  
HRegionServer会出现异常停止(执行abort())服务的场景如下：  
1.在读写hdfs时如果出现IOException异常，此时会发起hdfs的文件系统检查（checkFileSystem）1.            
2.Regionserver的服务线程出现了未捕获异常；  
3.在启动HRegionServer时出现异常；  
4.在进行HLog回滚时，出现异常；  
5.在flush memstore时，如果持久化失败，会重启RS，在重启中把hlog的内容重新加载到memstore；  
6.出现zk异常，包括但不限于以下场景：  
  a)Zk链接超时，超时时间通过zookeeper.session.timeout配置，默认为3分钟，与master不同，如果zk操作不会重试；  
  b)启动HRegionServer时出现KeeperException异常；   
  c)在进行split操作时，如果出现异常会进行回滚操作，在回滚过程中需要从zk中删除region的spliting状态，如果删除时出现KeeperException或者回滚的其它操作出现异常；  
  d)在打开region时，出现了KeeperException异常；  
  e)在进行hbase集群复制时，很多与zk交互的操作出现KeeperException异常时均会导致abort；  
7.在close region时，如果出现异常，比如：不能成功的flush memstore；  
8.Flush memstore时，如果HLog发现该region已经在flush则会强制终止JVM，采用的是Runtime.getRuntime().halt(1)方法，该方法不会执行正常退出的关闭钩子，从而不会flush RS的所有region，也不会迁移region，只有等待ZK的session超时后master才会发现该RS不可用，做迁移工作。

## 9.请描述 Hbase 中 scan 对象的 setCache 和 setBatch 方法的使用.

**cache：**

 在默认情况下，如果你需要从hbase中查询数据，在获取结果ResultScanner时，hbase会在你每次调用ResultScanner.next（）操作时对返回的每个Row执行一次RPC操作。即使你使用ResultScanner.next(int nbRows)时也只是在客户端循环调用RsultScanner.next()操作，你可以理解为hbase将执行查询请求以迭代器的模式设计，在执行next（）操作时才会真正的执行查询操作，而对每个Row都会执行一次RPC操作。

     因此显而易见的就会想如果我对多个Row返回查询结果才执行一次RPC调用，那么就会减少实际的通讯开销。这个就是hbase配置属性“hbase.client.scanner.caching”的由来，设置cache可以在hbase配置文件中显示静态的配置，也可以在程序动态的设置。

     cache值得设置并不是越大越好，需要做一个平衡。cache的值越大，则查询的性能就越高，但是与此同时，每一次调用next（）操作都需要花费更长的时间，因为获取的数据更多并且数据量大了传输到客户端需要的时间就越长，一旦你超过了maximum heap the client process 拥有的值，就会报outofmemoryException异常。当传输rows数据到客户端的时候，如果花费时间过长，则会抛出ScannerTimeOutException异常。

**batch**：

     在cache的情况下，我们一般讨论的是相对比较小的row，那么如果一个Row特别大的时候应该怎么处理呢？要知道cache的值增加，那么在client process 占用的内存就会随着row的增大而增大。在hbase中同样为解决这种情况提供了类似的操作：Batch。可以这么理解，cache是面向行的优化处理，batch是面向列的优化处理。它用来控制每次调用next（）操作时会返回多少列，比如你设置setBatch（5），那么每一个Result实例就会返回5列，如果你的列数为17的话，那么就会获得四个Result实例，分别含有5,5,5,2个列。

下面会以表格的形式来帮助理解，假设我们拥有10Row，每个row拥有2个family，每个family拥有10个列。（也就是说每个Row含有20列）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| caching | batch | Results | RPCs | Notes |
| 1 | 1 | 200 | 201 | 额外的一个RPC是用来判断scan是否完成 |
| 200 | 1 | 200 | 2 |  |
| 2000 | 100 | 10 | 1 | 超过的部分没有用处，但是判断scan也在那一个RPC 中完成 |
| 2 | 100 | 10 | 6 | 10/2 +1 (额外的判断开销） |
| 2 | 10 | 20 | 11 |  |
| 5 | 100 | 10 | 3 |  |
| 5 | 20 | 10 | 3 |  |
| 10 | 10 | 20 | 3 |  |

RPCs=(Rows\* Cols per Row) / Min(Cols per Row, Batch size) / Scanner caching

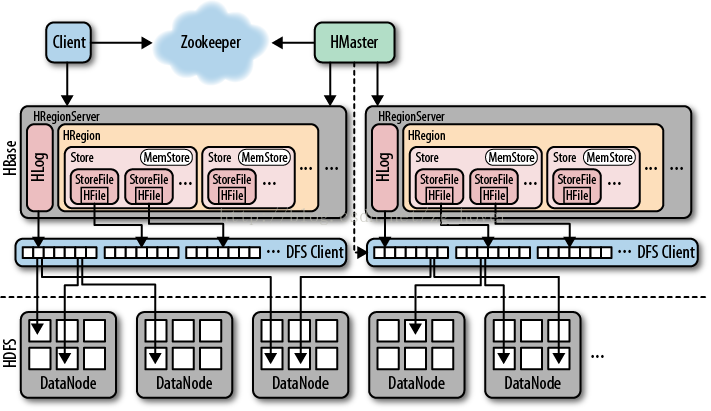
## 1. 总结性介绍

* HBase是一个构建在HDFS上的分布式列存储系统，是基于Google BigTable模型开发的，典型的key/value系统。
* 与传统mysql、Oracle数据库的主要区别就是列式存储和行式存储的区别。
* Hbase适合大量插入同时又有读的情况。输入一个Key获取一个value或输入一些key获得一些value。
* Hbase目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。
* Hbase表的特点
  + 大：一个表可以有数十亿行，上百万列；
  + 无模式：每行都有一个可排序的主键和任意多的列，列可以根据需要动态的增加，同一张表中不同的行可以有截然不同的列；
  + 面向列：面向列（族）的存储和权限控制，列（族）独立检索；
  + 稀疏：空（null）列并不占用存储空间，表可以设计的非常稀疏；
  + 数据多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳；
  + 数据类型单一：Hbase中的数据都是字符串，没有类型。

## 2. 存储形式-表结构

* table:
* rowkey
* column family
* column

## 3. 存储结构-HRegion, HStore, HFile

[](https://github.com/chen-kh/notebook/blob/master/learn_hadoop/hbase_structure.png)

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的状态，将Region server的上线和下线信息实时通知给Master
* 存储Hbase的schema,包括有哪些table，每个table有哪些column family

HMaster:

* 为Region server分配region，负责region server的负载均衡
* 管理用户对Table的增、删、改、查操作
* 发现失效的region server并重新分配其上的region
* GFS上的垃圾文件回收
* 在HRegionServer停机后，负责失效HRegionServer 上的Regions迁移

client访问Hbase上数据的过程并不需要master参与(寻址访问zookeeper和region server，数据读写访问regione server)，master仅仅维护着**table**和**region**的元数据信息，负载很低。

HRegion Server

* Region server维护Master分配给它的region，处理对这些region的IO请求
* Region server负责切分在运行过程中变得过大的region

## 4. region 定位问题

### 4.1. -ROOT- 和 .META. 怎么来的

如果要insert一个数据，需要找到相应的Region，如何找呢？就需要一个所有region的meta信息，但是region可能很多，那就分级来搞，先有一个-ROOT-索引.META.的内容，.META.再索引Region的内容。不需要再多级别了，因为这两个级别就可以存储2^34个region了，足够使用，再多了反而增加复杂度和网络开销。

### 4.2. -ROOT- 和 .META. 的存储结构

可以视为一般的region，存储形式都是相同的。只不过rowkey不太一样。

## 5. 一致性问题：HBase的读写一致性是怎么保证的？做到了什么程度？

参考：

* [HBase-强一致性详解](https://www.cnblogs.com/captainlucky/p/4720986.html)
* [LSM 算法的原理是什么？](http://www.open-open.com/lib/view/open1424916275249.html)

从一开始就知道hbase是CAP中的CP系统,即hbase是强一致性的.我原来一直以为hbase的强一致性是因为底层的HDFS写入时,必须所有副本都写入成功才能返回.最近才想明白,hbase之所以是CP系统,实际和底层HDFS无关,它是CP系统,是因为对每一个region同时只有一台region server为它服务,对一个region所有的操作请求,都由这一台region server来响应,自然是强一致性的.在这台region server fail的时候,它管理的region failover到其他region server时,需要根据WAL log来redo,这时候进行redo的region应该是unavailable的,所以hbase降低了可用性,提高了一致性.设想一下,如果redo的region能够响应请求,那么可用性提高了,则必然返回不一致的数据(因为redo可能还没完成),那么hbase就降低一致性来提高可用性了.

 1.HBase的架构和基本原理

Hbase以表的方式组织数据，

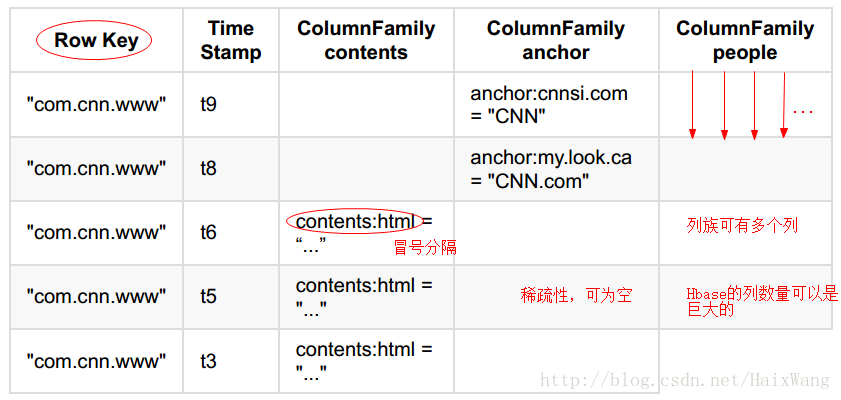
表由行（Row）以及列（Column）组成，行由row key和一个或多个列及其值组成（存储是按照row key的字典顺序排序，row key的设计非常重要！！），

列必须属于某一列族（Column family），一个列族可以有一各或多个列（一列由列簇和列修饰符组成，他们通常由冒号（：） 分隔），其在存储架构中就是一个Hfile。

Hbase中的列可以达到百万级，列中的数据可以是稀疏的，空值并不占用存储空间。

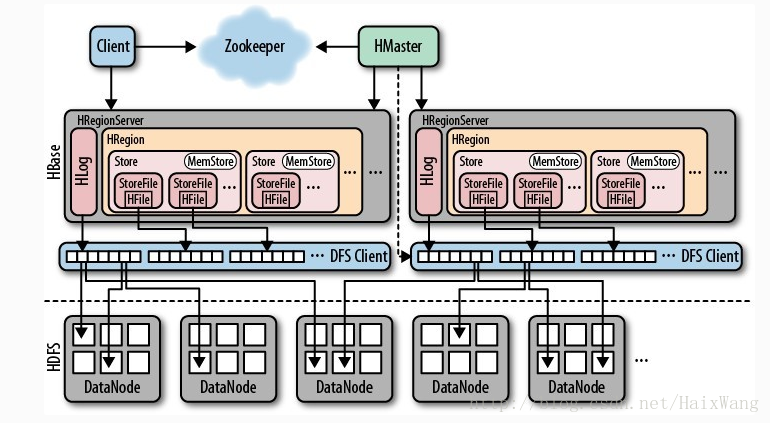
数据按主键排序，同时表按主键划分为多个Region。底层是LSM树（Long-Structed Merge Tree）。

对于以上叙述，表的简略结构：(逻辑模型)



注1：建议把Hbase理解为一个多维Map

Hbase简略架构图如下：



接下来对Zookeeper、HMaster、HRegionServer、HRegion、Store、MemStore、StoreFile、HFile、HLog等做个简单叙述：

Zookeeper

保证任何时候，集群中只有一个master（负责多HMaster的选举）

存贮所有Region的寻址入口

实时监控RegionServer的状态、将RegionServer的上线和下线信息实时通知给Master（服务器之间状态同步）

存储Hbase的schema（元数据信息）。包括有哪些table、每个table有哪些column family等

HMaster

(主要负责table和region的管理工作)

Region分裂后，为RegionServer分配新的Region

负责RegionServer的负载均衡，调整region的分配

发现失效的region server并重新分配其上的region

(Region自动切分是HBase能够拥有良好扩张性的最重要因素之一，见转载文章：http://developer.51cto.com/art/201708/549419.htm）

管理用户对table的增、删、改、查操作

监听zk,基于zookeeper感应rs的上下线

监听zk,基于zookeeper来保证HA

处理schema更新请求 （管理用户对表的增删修改）

不参与对表的读写访问

负载很低

无SPOF(单点故障)

在一个RegionServer死机后，负责失效节点的Region的迁移

HRegionServer

(主要负责响应用户对其上region的I/O请求，向HDFS读写数据，HbasHBase核心模块)

HRegionSserver维护Master分配给它的region

维护region的cache

处理region的flush、compact、split

内部管理一系列的HRegion对象

一个HRegionServer会有多个HRegion和一个HLog。

Region

（图中为Hregion，也就是指一个Table的分区）

每一个HRegion又由很多的Store组成，每一个Store存储的实际上是一个列簇（ColumnFamily）下所有的数据。此外，在每一个Store（又名HStore）中有包含一块MemStore。MemStore驻留在内存中，数据到来时首先更新到MemStore中，当到达阈值之后再flush（默认64M）到对应的StoreFile（又名HFile）中，所以每一个Store包含多个StoreFile，StoreFile负责的是实际数据存储，为HBase中最小的存储单元。

达到某个阈值时，分裂（默认256M）。所以一个HRegionServer管理多个表，一个表下有多个Region，一个HRegion有多少个列族就有多少个Store,Store下有多个StoreFile文件，是HBase中最小的存储单元

以Region为单位管理, region(startKey,endKey)；【默认情况下，刚创建一个表时，并不知道startkey和endkey】

每个Column Family单独存储：storeFile；（ storefile的数量一多（到达阀值），就合并（同时合并版本以及删除之前要删除的数据）；合并后大小到达阀值就split）

当某个Column Family累积的大小（具体的数据量） > 某阈值时，自动分裂成两个Region；合并之后，旧数据也不是立即删除，而是复制一份并同内存中的数据一起写到磁盘，在之后，LSM-Tree会提供一些机制来回收这些空间。[4]

如何找到某行属于哪个region呢？两张特殊的表：

-NAMESPACE- 和.META.

StoreFile

（底层存储格式是HFile，HBase中最小的存储单元）

memStore内存中的数据写到文件后就是StoreFile，StoreFile底层是以HFile的格式保存。

HFile[1]

（HFile基于Hadoop的TFile类实现，模仿Google的BIgTable论文中的SSTable格式。）

HBase中KeyValue数据的存储格式，是hadoop的二进制格式文件。

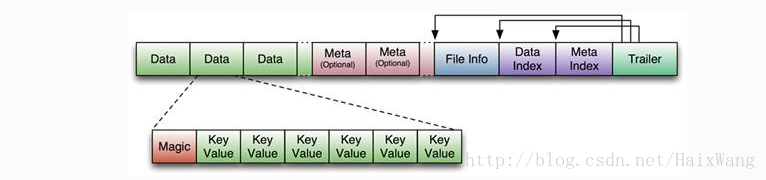
HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。

Trailer中有指针指向其他数据块的起始点，FileInfo记录了文件的一些meta信息。 - Data Block是hbase io的基本单元，为了提高效率，HRegionServer中有基于LRU的block cache机制。

每个Data块的大小可以在创建一个Table的时候通过参数指定（默认块大小64KB），大号的Block有利于顺序Scan，小号的Block利于随机查询。

每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成

Magic内容就是一些随机数字，目的是防止数据损坏



HLog(WAL log)

（HLog是WAL的核心实现类）

WAL意为write ahead log，HBase中的预写日志，用来做灾难恢复使用，底层实现是HLog，HLog记录数据的所有变更。使用WAL的原因：因为MemStore存储的数据是驻留在内存中的，是不稳定的（比如宕机时），所以采用了WAL预写日志来解决这个问题。（运行MApReduce作业时，可以通过关闭WAL功能来获得性能的提升——setWriteToWAL(boolean)）

其实HLog文件就是一个普通的Hadoop Sequence File， Sequence File的value是key时HLogKey对象，其中记录了写入数据的归属信息，除了table和region名字外，还同时包括sequence number和timestamp，timestamp是写入时间，sequence number的起始值为0，或者是最近一次存入文件系统中的sequence number。 Sequence File的value是HBase的KeyValue对象，即对应HFile中的KeyValue。[1]

2.HBase与传统关系型数据库(如MySQL)的区别

数据类型：没有数据类型，都是字节数组（有一个工具类Bytes，将java对象序列化为字节数组）。

数据操作：HBase只有很简单的插入、查询、删除、清空等操作，表和表之间是分离的，没有复杂的表和表之间的关系，而传统数据库通常有各式各样的函数和连接操作。

存储模式：Hbase适合于非结构化数据存储，基于列存储而不是行。

数据维护：HBase的更新操作不应该叫更新，它实际上是插入了新的数据，而传统数据库是替换修改

时间版本：Hbase数据写入cell时，还会附带时间戳，默认为数据写入时RegionServer的时间，但是也可以指定一个不同的时间。数据可以有多个版本。

可伸缩性，Hbase这类分布式数据库就是为了这个目的而开发出来的，所以它能够轻松增加或减少硬件的数量，并且对错误的兼容性比较高。而传统数据库通常需要增加中间层才能实现类似的功能

### 3.HBase的缺点？（优点已经在4中体现）

* 单一RowKey固有的局限性决定了它不可能有效地支持多条件查询[2]
* 不适合于大范围扫描查询
* 不直接支持 SQL 的语句查询

4.什么时候适合使用HBase（应用场景）

半结构化或非结构化数据:

对于数据结构字段不够确定或杂乱无章非常难按一个概念去进行抽取的数据适合用HBase，因为HBase支持动态添加列。

记录很稀疏：

RDBMS的行有多少列是固定的。为null的列浪费了存储空间。而如上文提到的，HBase为null的Column不会被存储，这样既节省了空间又提高了读性能。

多版本号数据：

依据Row key和Column key定位到的Value能够有随意数量的版本号值，因此对于须要存储变动历史记录的数据，用HBase是很方便的。比方某个用户的Address变更，用户的Address变更记录也许也是具有研究意义的。

仅要求最终一致性：

对于数据存储事务的要求不像金融行业和财务系统这么高，只要保证最终一致性就行。（比如HBase+elasticsearch时，可能出现数据不一致）

高可用和海量数据以及很大的瞬间写入量：

WAL解决高可用，支持PB级数据，put性能高

索引插入比查询操作更频繁的情况。比如，对于历史记录表和日志文件。（HBase的写操作更加高效）

业务场景简单：

不需要太多的关系型数据库特性，列入交叉列，交叉表，事务，连接等。

1. HBase来源于哪篇博文？ C

A The Google File System

B MapReduce

C BigTable

D Chubby

2. 下面对HBase的描述哪些是正确的？ B、C、D

A 不是开源的

B 是面向列的

C 是分布式的

D 是一种NoSQL数据库

3. HBase依靠（）存储底层数据 A

A HDFS

B Hadoop

C Memory

D MapReduce

4. HBase依赖（）提供消息通信机制 A

A Zookeeper

B Chubby

C RPC

D Socket

5. HBase依赖（）提供强大的计算能力 D

A Zookeeper

B Chubby

C RPC

D MapReduce

6. MapReduce与HBase的关系，哪些描述是正确的？ B、C

A 两者不可或缺，MapReduce是HBase可以正常运行的保证

B 两者不是强关联关系，没有MapReduce，HBase可以正常运行

C MapReduce可以直接访问HBase

D 它们之间没有任何关系

7. 下面哪些选项正确描述了HBase的特性？ A、B、C、D

A 高可靠性

B 高性能

C 面向列

D可伸缩

8. 下面与Zookeeper类似的框架是？D

A Protobuf

B Java

C Kafka

（Kafka是一个高吞吐量分布式消息系统。linkedin开源的kafka。 Kafka就跟这个名字一样，设计非常独特。首先，kafka的开发者们认为不需要在内存里缓存什么数据，操作系统的文件缓存已经足够完善和强大，只要你不搞随机写，顺序读写的性能是非常高效的。kafka的数据只会顺序append，数据的删除策略是累积到一定程度或者超过一定时间再删除。Kafka另一个独特的地方是将消费者信息保存在客户端而不是MQ服务器，这样服务器就不用记录消息的投递过程，每个客户端都自己知道自己下一次应该从什么地方什么位置读取消息，消息的投递过程也是采用客户端主动pull的模型，这样大大减轻了服务器的负担。Kafka还强调减少数据的序列化和拷贝开销，它会将一些消息组织成Message Set做批量存储和发送，并且客户端在pull数据的时候，尽量以zero-copy的方式传输，利用sendfile（对应java里的 FileChannel.transferTo/transferFrom）这样的高级IO函数来减少拷贝开销。可见，kafka是一个精心设计，特定于某些应用的MQ系统，这种偏向特定领域的MQ系统我估计会越来越多，垂直化的产品策略值的考虑）

D Chubby

（MapReduce 很多人已经知道了，但关于Chubyy似乎熟悉它的就非常有限，这倒是不奇怪，因为MapReduce是一个针对开发人员的 ProgrammingModel，自然会有很多人去学习它，而Chubby更多的是一种为了实现MapReduce或者Bigtable而构建的内部的 工具，对于开发人员来说基本上是透明的。

Chubby首先是一个分布式的文件系统。Chubby能够提供机制使得client可以在Chubby service上创建文件和执行一些文件的基本操作。说它是分布式的文件系统，是因为一个Chubby cell是一个分布式的系统，一般包含了5台机器，整个文件系统是部署在这5台机器上的。

但是，从更高一点的语义层面上，Chubby是一个 lock service，一个针对松耦合的分布式系统的lock service。所谓lock service，就是这个service能够提供开发人员经常用的“锁”，“解锁”功能。通过Chubby，一个分布式系统中的上千个client都能够 对于某项资源进行“加锁”，“解锁”。

那么，Chubby是怎样实现这样的“锁”功能的？就是通过文件。Chubby中的“锁”就是文件，在上例 中，创建文件其实就是进行“加锁”操作，创建文件成功的那个server其实就是抢占到了“锁”。用户通过打开、关闭和读取文件，获取共享锁或者独占锁； 并且通过通信机制，向用户发送更新信息。

综上所述，Chubby是一个lock service，通过这个lock service可以解决分布式中的一致性问题，而这个lock service的实现是一个分布式的文件系统。）

9. 下面与HDFS类似的框架是？C

A NTFS

B FAT32

C GFS(也是分布式文件系统，谷歌自己的分布式文件系统)

D EXT3

10. 下面哪些概念是HBase框架中使用的？A、C

A HDFS

B GridFS

C Zookeeper

D EXT3

第二部分：HBase核心知识点(对于Hbase实现核心LSM，会在下篇文章介绍）

11. LSM含义是？A

A 日志结构合并树（Log-Structured Merge Tree）

B 二叉树

C 平衡二叉树

D 长平衡二叉树

12. 下面对LSM结构描述正确的是？ A、C

A 顺序存储

B 直接写硬盘

C 需要将数据Flush到磁盘

D 是一种搜索平衡树

13. LSM更能保证哪种操作的性能？B

A 读

B 写

C 随机读

D 合并

14. LSM的读操作和写操作是独立的？A

A 是。

B 否。

C LSM并不区分读和写

D LSM中读写是同一种操作

15. LSM结构的数据首先存储在（）。 B

A 硬盘上

B 内存中

C 磁盘阵列中

D 闪存中

16 HFile数据格式中的Data字段用于（）。A

A 存储实际的KeyValue数据

B 存储数据的起点

C 指定字段的长度

D 存储数据块的起点

17 HFile数据格式中的MetaIndex字段用于（）。D

A Meta块的长度

B Meta块的结束点

C Meta块数据内容

D Meta块的起始点

18 HFile数据格式中的Magic字段用于（）。A

A 存储随机数，防止数据损坏

B 存储数据的起点

C 存储数据块的起点

D 指定字段的长度

19 HFile数据格式中的KeyValue数据格式，下列选项描述正确的是（）。A、D

A 是byte[]数组

B 没有固定的结构

C 数据的大小是定长的

D 有固定的结构

20 HFile数据格式中的KeyValue数据格式中Value部分是（）。C

A 拥有复杂结构的字符串

B 字符串

C 二进制数据

D 压缩数据

第三部分：HBase高级应用介绍

31 HBase中的批量加载底层使用（）实现。A

A MapReduce

B Hive

C Coprocessor

D Bloom Filter

32. HBase性能优化包含下面的哪些选项？A、B、C、D

A 读优化

B 写优化

C 配置优化

D JVM优化

33. Rowkey设计的原则，下列哪些选项的描述是正确的？A、B、C

A 尽量保证越短越好

B 可以使用汉字

C 可以使用字符串

D 本身是无序的

34. HBase构建二级索引的实现方式有哪些？ A、B

A MapReduce

B Coprocessor

(HBase在0.92之后引入了协处理器(coprocessors)，实现一些激动人心的新特性：能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等)

C Bloom Filter

D Filter

35. 关于HBase二级索引的描述，哪些是正确的？A、B

A 核心是倒排表

B 二级索引概念是对应Rowkey这个“一级”索引

C 二级索引使用平衡二叉树

D 二级索引使用LSM结构

36. 下列关于Bloom Filter的描述正确的是？A、C

A 是一个很长的二进制向量和一系列随机映射函数

B 没有误算率

C 有一定的误算率

D 可以在Bloom Filter中删除元素

第四部分：HBase安装、部署、启动

37. HBase官方版本可以安装在什么操作系统上？A、B、C

A CentOS

B Ubuntu

C RedHat

D Windows

38. HBase虚拟分布式模式需要（）个节点？A

A 1

B 2

C 3

D 最少3个

39. HBase分布式模式最好需要（）个节点？C

A 1

B 2

C 3

D 最少

40. 下列哪些选项是安装HBase前所必须安装的？A、B

A 操作系统

B JDK

C Shell Script

D Java Code

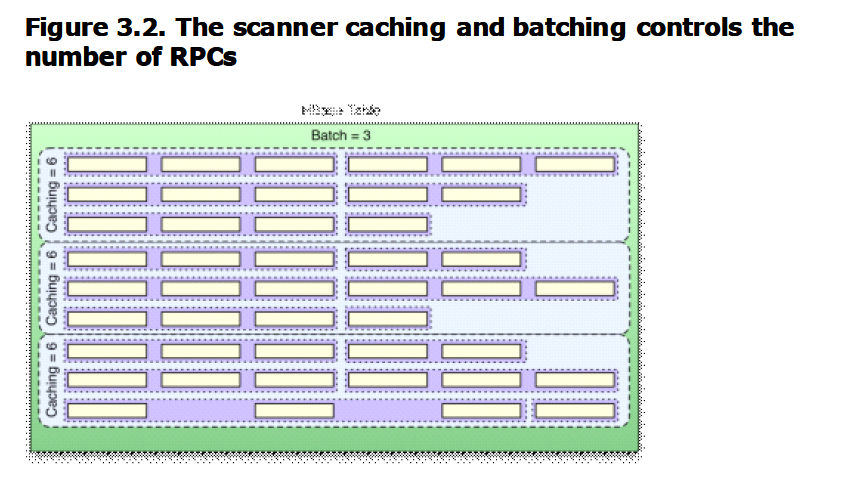
41. 解压.tar.gz结尾的HBase压缩包使用的Linux命令是？A

A tar -zxvf

B tar -zx

C tar -s

D tar -nf



1.Kafka的设计是什么样的呢？

Kafka将消息以topic为单位进行归纳

将向Kafka topic发布消息的程序成为producers.

将预订topics并消费消息的程序成为consumer.

Kafka以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个broker.

producers通过网络将消息发送到Kafka集群，集群向消费者提供消息

2.数据传输的事物定义有哪三种？

数据传输的事务定义通常有以下三种级别：

（1）最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输

（2）最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

（3）精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

3.Kafka判断一个节点是否还活着有那两个条件？

（1）节点必须可以维护和ZooKeeper的连接，Zookeeper通过心跳机制检查每个节点的连接

（2）如果节点是个follower,他必须能及时的同步leader的写操作，延时不能太久

4.producer是否直接将数据发送到broker的leader(主节点)？

producer直接将数据发送到broker的leader(主节点)，不需要在多个节点进行分发，为了帮助producer做到这点，所有的Kafka节点都可以及时的告知:哪些节点是活动的，目标topic目标分区的leader在哪。这样producer就可以直接将消息发送到目的地了

5、Kafa consumer是否可以消费指定分区消息？

Kafa consumer消费消息时，向broker发出"fetch"请求去消费特定分区的消息，consumer指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息，customer拥有了offset的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的

6、Kafka消息是采用Pull模式，还是Push模式？

Kafka最初考虑的问题是，customer应该从brokes拉取消息还是brokers将消息推送到consumer，也就是pull还push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer从broker拉取消息

一些消息系统比如Scribe和Apache Flume采用了push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式

Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略

Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到t达。为了避免这点，Kafka有个参数可以让consumer阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发

7.Kafka存储在硬盘上的消息格式是什么？

消息由一个固定长度的头部和可变长度的字节数组组成。头部包含了一个版本号和CRC32校验码。

消息长度: 4 bytes (value: 1+4+n)

版本号: 1 byte

CRC校验码: 4 bytes

具体的消息: n bytes

8.Kafka高效文件存储设计特点：

(1).Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

(2).通过索引信息可以快速定位message和确定response的最大大小。

(3).通过index元数据全部映射到memory，可以避免segment file的IO磁盘操作。

(4).通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。

9.Kafka 与传统消息系统之间有三个关键区别

(1).Kafka 持久化日志，这些日志可以被重复读取和无限期保留

(2).Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

(3).Kafka 支持实时的流式处理

10.Kafka创建Topic时如何将分区放置到不同的Broker中

副本因子不能大于 Broker 的个数；

第一个分区（编号为0）的第一个副本放置位置是随机从 brokerList 选择的；

其他分区的第一个副本放置位置相对于第0个分区依次往后移。也就是如果我们有5个 Broker，5个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；

剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

11.Kafka新建的分区会在哪个目录下创建

在启动 Kafka 集群之前，我们需要配置好 log.dirs 参数，其值是 Kafka 数据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。

当然我们也可以配置 log.dir 参数，含义一样。只需要设置其中一个即可。

如果 log.dirs 参数只配置了一个目录，那么分配到各个 Broker 上的分区肯定只能在这个目录下创建文件夹用于存放数据。

但是如果 log.dirs 参数配置了多个目录，那么 Kafka 会在哪个文件夹中创建分区目录呢？答案是：Kafka 会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic名+分区ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录！也就是说，如果你给 log.dirs 参数新增了一个新的磁盘，新的分区目录肯定是先在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

12.partition的数据如何保存到硬盘

topic中的多个partition以文件夹的形式保存到broker，每个分区序号从0递增，

且消息有序

Partition文件下有多个segment（xxx.index，xxx.log）

segment 文件里的 大小和配置文件大小一致可以根据要求修改 默认为1g

如果大小大于1g时，会滚动一个新的segment并且以上一个segment最后一条消息的偏移量命名

13.kafka的ack机制

request.required.acks有三个值 0 1 -1

0:生产者不会等待broker的ack，这个延迟最低但是存储的保证最弱当server挂掉的时候就会丢数据

1：服务端会等待ack值 leader副本确认接收到消息后发送ack但是如果leader挂掉后他不确保是否复制完成新leader也会导致数据丢失

-1：同样在1的基础上 服务端会等所有的follower的副本受到数据后才会受到leader发出的ack，这样数据不会丢失

14.Kafka的消费者如何消费数据

消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（offset）的位置

等到下次消费时，他会接着上次位置继续消费

15.消费者负载均衡策略

一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如果组中成员太多会有空闲的成员

16.数据有序

一个消费者组里它的内部是有序的

消费者组与消费者组之间是无序的

17.kafaka生产数据时数据的分组策略

生产者决定数据产生到集群的哪个partition中

每一条消息都是以（key，value）格式

Key是由生产者发送数据传入

所以生产者（key）决定了数据产生到集群的哪个partition

Kafka的用途有哪些？使用场景如何？

Kafka中的ISR、AR又代表什么？ISR的伸缩又指什么

Kafka中的HW、LEO、LSO、LW等分别代表什么？

Kafka中是怎么体现消息顺序性的？

Kafka中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

Kafka生产者客户端的整体结构是什么样子的？

Kafka生产者客户端中使用了几个线程来处理？分别是什么？

Kafka的旧版Scala的消费者客户端的设计有什么缺陷？

“消费组中的消费者个数如果超过topic的分区，那么就会有消费者消费不到数据”这句话是否正确？如果正确，那么有没有什么hack的手段？

消费者提交消费位移时提交的是当前消费到的最新消息的offset还是offset+1?

有哪些情形会造成重复消费？

那些情景下会造成消息漏消费？

KafkaConsumer是非线程安全的，那么怎么样实现多线程消费？

简述消费者与消费组之间的关系

当你使用kafka-topics.sh创建（删除）了一个topic之后，Kafka背后会执行什么逻辑？

topic的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

topic的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

创建topic时如何选择合适的分区数？

Kafka目前有那些内部topic，它们都有什么特征？各自的作用又是什么？

优先副本是什么？它有什么特殊的作用？

Kafka有哪几处地方有分区分配的概念？简述大致的过程及原理

简述Kafka的日志目录结构

Kafka中有那些索引文件？

如果我指定了一个offset，Kafka怎么查找到对应的消息？

如果我指定了一个timestamp，Kafka怎么查找到对应的消息？

聊一聊你对Kafka的Log Retention的理解

聊一聊你对Kafka的Log Compaction的理解

聊一聊你对Kafka底层存储的理解（页缓存、内核层、块层、设备层）

聊一聊Kafka的延时操作的原理

聊一聊Kafka控制器的作用

消费再均衡的原理是什么？（提示：消费者协调器和消费组协调器）

Kafka中的幂等是怎么实现的

Kafka中的事务是怎么实现的（这题我去面试6加被问4次，照着答案念也要念十几分钟，面试官简直凑不要脸）

Kafka中有那些地方需要选举？这些地方的选举策略又有哪些？

失效副本是指什么？有那些应对措施？

多副本下，各个副本中的HW和LEO的演变过程

为什么Kafka不支持读写分离？

Kafka在可靠性方面做了哪些改进？（HW, LeaderEpoch）

Kafka中怎么实现死信队列和重试队列？

Kafka中的延迟队列怎么实现（这题被问的比事务那题还要多！！！听说你会Kafka，那你说说延迟队列怎么实现？）

Kafka中怎么做消息审计？

Kafka中怎么做消息轨迹？

Kafka中有那些配置参数比较有意思？聊一聊你的看法

Kafka中有那些命名比较有意思？聊一聊你的看法

Kafka有哪些指标需要着重关注？

怎么计算Lag？(注意read\_uncommitted和read\_committed状态下的不同)

Kafka的那些设计让它有如此高的性能？

Kafka有什么优缺点？

还用过什么同质类的其它产品，与Kafka相比有什么优缺点？

为什么选择Kafka?

在使用Kafka的过程中遇到过什么困难？怎么解决的？

怎么样才能确保Kafka极大程度上的可靠性？

聊一聊你对Kafka生态的理解

**请说明什么是Apache Kafka?**

Apache Kafka是由Apache开发的一种发布订阅消息系统，它是一个分布式的、分区的和重复的日志服务。

**请说明什么是传统的消息传递方法?**

传统的消息传递方法包括两种：

排队：在队列中，一组用户可以从服务器中读取消息，每条消息都发送给其中一个人。

发布-订阅：在这个模型中，消息被广播给所有的用户。

**请说明Kafka相对传统技术有什么优势?**

快速：单一的Kafka代理可以处理成千上万的客户端，每秒处理数兆字节的读写操作。

可伸缩：在一组机器上对数据进行分区和简化，以支持更大的数据

持久：消息是持久性的，并在集群中进行复制，以防止数据丢失。

设计：它提供了容错保证和持久性

**在Kafka中broker的意义是什么?**

接收Producer发过来的数据，并且将它持久化，同时提供给Consumer去订阅

组成Kafka集群节点，之间没有主从关系，依赖ZooKeeper来协调，broker负责消息的读取和存储，一个broker可以管理多个partition

**什么是broker？作用是什么?**

 一个单独的kafka server就是一个broker，broker主要工作就是接收生产者发过来的消息，分配offset，之后保存到磁盘中。同时，接收消费者、其他broker的请求，根据请求类型进行相应的处理并返回响应，在一般的生产环境中，一个broker独占一台物理服务器

**Kafka服务器能接收到的最大信息是多少?**

Kafka服务器可以接收到的消息的最大大小是1000000字节。

**解释Kafka的Zookeeper是什么?我们可以在没有Zookeeper的情况下使用Kafka吗?**

Zookeeper是一个开放源码的、高性能的协调服务，它用于Kafka的分布式应用。

作用：协调Kafka Broker，存储原数据：consumer的offset+broker信息+topic信息+partition个信息。

不，不可能越过Zookeeper，直接联系Kafka broker。一旦Zookeeper停止工作，它就不能服务客户端请求。

Zookeeper主要用于在集群中不同节点之间进行通信

在Kafka中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从之前提交的偏移量中获取

除此之外，它还执行其他活动，如: leader检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

**解释Kafka的用户如何消费信息?**

在Kafka中传递消息是通过使用sendfile【零拷贝】 API完成的。它支持将字节从套接口转移到磁盘，通过内核空间保存副本，并在内核用户之间调用内核。

零拷贝：用户向内核去发送一个命令，我要操作那些数据，然后直接从磁盘转成Socket Buffer，再从Socket Buffer到网卡Buffer，再传出去【少了两次的copy】

**解释如何提高远程用户的吞吐量?**

如果用户位于与broker不同的数据中心，则可能需要调优套接口缓冲区大小，以对长网络延迟进行摊销。

**解释一下，在数据制作过程中，你如何能从Kafka得到准确的信息?**

在数据中，为了精确地获得Kafka的消息，你必须遵循两件事: 在数据消耗期间避免重复，在数据生产过程中避免重复。

这里有两种方法，可以在数据生成时准确地获得一个语义

每个分区使用一个单独的写入器，每当你发现一个网络错误，检查该分区中的最后一条消息，以查看您的最后一次写入是否成功

在消息中包含一个主键(UUID或其他)，并在用户中进行反复制

**解释如何减少ISR中的扰动?broker什么时候离开ISR?**

ISR是一组与leaders完全同步的消息副本，也就是说ISR中包含了所有提交的消息。ISR应该总是包含所有的副本，直到出现真正的故障。如果一个副本从leader中脱离出来，将会从ISR中删除。

**Kafka为什么需要复制?**

Kafka的信息复制确保了任何已发布的消息不会丢失，并且可以在机器错误、程序错误或更常见些的软件升级中使用。

**如果副本在ISR中停留了很长时间表明什么?**

如果一个副本在ISR中保留了很长一段时间，那么它就表明，跟踪器无法像在leader收集数据那样快速地获取数据。

**请说明如果首选的副本不在ISR中会发生什么?**

如果首选的副本不在ISR中，控制器将无法将leadership转移到首选的副本。

**有可能在生产后发生消息偏移吗?**

在大多数队列系统中，作为生产者的类无法做到这一点，它的作用是触发并忘记消息。broker将完成剩下的工作，比如使用id进行适当的元数据处理、偏移量等。

作为消息的用户，你可以从Kafka broker中获得补偿。如果你注视SimpleConsumer类，你会注意到它会获取包括偏移量作为列表的MultiFetchResponse对象。此外，当你对Kafka消息进行迭代时，你会拥有包括偏移量和消息发送的MessageAndOffset对象。

**kafka主要特征**

kafka具有近乎实时性的消息处理能力，面对海量数据，高效的存储消息和查询消息。kafka将消息保存在磁盘中，以顺序读写的方式访问磁盘，从而避免了随机读写磁盘导致的性能瓶颈

kafka支持批量读写消息，并且对消息批量压缩，提高了网络利用率和压缩效率

kafka支持消息分区，每个分区中的消息保证顺序传输，而分区之间可以并发操作，提高了kafka的并发能力

kafka支持在线增加分区，支持在线水平扩展

kafka支持为每个分区创建多个副本，其中只会有一个leader副本负责读写，其他副本只负责与leader副本同步，这种方式提高了数据的容灾能力，kafka会将leader副本均匀的分布在集群中的服务器上，实现性能最大化

**列举kafka的应用场景**

日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer，例如Hadoop、Hbase、Solr等

消息系统：解耦和生产者和消费者、缓存消息等

用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到Hadoop、数据仓库中做离线分析和挖掘

运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告

流式处理：比如spark streaming和storm

事件源

**kafka主题分区的作用**

kafka的每个topic都可以分为多个partition，每个partition都有多个replica（副本），每个分区中的消息是不同的，提高了并发读写的能力，而同一分区的不同副本中保存的是相同的消息，副本之间是一主多从关系，其中leader副本处理读写请求，follower副本只与leader副本进行消息同步，当leader副本出现故障时，则从follower副本中重新选举leader副本对外提供服务。这样，通过提高分区的数量，就可以实现水平扩展，通过提高副本数量，就可以提高容灾能力

**consumer水平扩展如何实现**

kafka支持consumer水平扩展，可以让多个consumer加入一个consumer group，在一个consumer group中，每个分区只能分配给一个consumer，当kafka服务端增加分区数量进行水平扩展后，可以向consumer group中增加新的consumer来提高整个consumer group的消费能力，当consumer group 中的一个consumer出现故障下线时，会通过rebalance操作下线consumer，它负责处理的分区将分配给其他consumer

**消息的顺序**

kafka保证一个partition内消息是有序的，但是并不保证多个partition之间的数据有顺序，每个topic可以划分成多个分区，同一个topic下的不同分区包含的消息是不同的，每个消息在被添加到分区时，都会被分配一个offset，它是此消息在分区中的唯一编号，kafka通过offset保证消息在分区内的顺序，offset顺序不跨分区，即kafka只保证在同一个分区内的消息是有序的

**为了避免磁盘被占满，kafka会周期性的删除陈旧的消息，删除策略是什么?**

一种是根据消息保留的时间

一种是根据topic存储的数据大小

**什么是日志压缩**

在很多场景中，消息的key与value之间的对应关系是不断变化的，消费者只关心key对应的最新value，此时，可以开启kafka的日志压缩功能，kafka会在后台启动一个线程，定期将相同key的消息进行合并，只保留最新的value值

**同一分区的多个副本包括的消息是否一致？**

每个副本中包含的消息是一样的，但是再同一时刻，副本之间并不是完全一样的

**ISR集合是什么？谁维护着？如何维护？**

ISR（In-Sync Replica）集合表示的是目前可用并且消息量与leader相差不多的副本集合，这是整个副本集合的一个子集

ISR集合的副本必须满足：副本所在节点必须维持着与zookeeper的连接；副本最后一条消息的offset与leader副本最后一条消息的offset之间的差值不能超出指定的阈值

每个分区的leader副本都会维护此分区的ISR集合，写请求首先由leader副本处理，之后follower副本会从leader副本上拉取写入的消息，这个过程会有一定的延迟，导致follower副本中保存的消息略少于leader副本，只要未超出阈值都是可以容忍的

**Kafka的设计时什么样的呢？**

Kafka将消息以topic为单位进行归纳

将向Kafka topic发布消息的程序成为producer

将预订topics并消费消息的程序成为consumer

Kafka以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个broker

producers通过网络将消息发送到Kafka集群，集群向消费者提供消息

**数据传输的事物定义有哪三种？**

数据传输的事务定义通常有以下三种级别：

（1）最多一次：消息不会被重复发送，最多被传输一次，但也有可能一次不传输

（2）最少一次：消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

（3）精确的一次（Exactly once）：不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

**Kafka判断一个节点是否还活着有那两个条件？**

节点必须可以维护和ZooKeeper的连接，Zookeeper通过心跳机制检查每个节点的连接

如果节点是个follower,他必须能及时的同步leader的写操作，延时不能太久

**producer是否直接将数据发送到broker的leader(主节点)？**

producer直接将数据发送到broker的leader(主节点)，不需要在多个节点进行分发，为了帮助producer做到这点，所有的Kafka节点都可以及时的告知：哪些节点是活动的，目标topic目标分区的leader在哪。这样producer就可以直接将消息发送到目的地了

**Kafa consumer是否可以消费指定分区消息？**

Kafa consumer消费消息时，向broker发出"fetch"请求去消费特定分区的消息，consumer指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息，customer拥有了offset的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的

**Kafka消息是采用Pull模式，还是Push模式？**

Kafka最初考虑的问题是，customer应该从brokes拉取消息还是brokers将消息推送到consumer，也就是pull还push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer从broker拉取消息

一些消息系统比如Scribe和Apache Flume采用了push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式

Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略

Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到t达。为了避免这点，Kafka有个参数可以让consumer阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发

**Kafka存储在硬盘上的消息格式是什么？**

消息由一个固定长度的头部和可变长度的字节数组组成。头部包含了一个版本号和CRC32校验码。

消息长度: 4 bytes (value: 1+4+n)

版本号: 1 byte

CRC校验码: 4 bytes

具体的消息: n bytes

**Kafka高效文件存储设计特点：**

Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

通过索引信息可以快速定位message和确定response的最大大小。

通过index元数据全部映射到memory，可以避免segment file的IO磁盘操作。

通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。

**Kafka 与传统消息系统之间有三个关键区别**

Kafka 持久化日志，这些日志可以被重复读取和无限期保留

Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

Kafka 支持实时的流式处理

**Kafka创建Topic时如何将分区放置到不同的Broker中**

副本因子不能大于 Broker 的个数；

第一个分区（编号为0）的第一个副本放置位置是随机从 brokerList 选择的；

其他分区的第一个副本放置位置相对于第0个分区依次往后移。也就是如果我们有5个 Broker，5个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；

剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

**Kafka新建的分区会在哪个目录下创建**

在启动 Kafka 集群之前，我们需要配置好 log.dirs 参数，其值是 Kafka 数据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。

当然我们也可以配置 log.dir 参数，含义一样。只需要设置其中一个即可。

如果 log.dirs 参数只配置了一个目录，那么分配到各个 Broker 上的分区肯定只能在这个目录下创建文件夹用于存放数据。

但是如果 log.dirs 参数配置了多个目录，那么 Kafka 会在哪个文件夹中创建分区目录呢？答案是：Kafka 会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic名+分区ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录！也就是说，如果你给 log.dirs 参数新增了一个新的磁盘，新的分区目录肯定是先在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

**partition的数据如何保存到硬盘**

topic中的多个partition以文件夹的形式保存到broker，每个分区序号从0递增，且消息有序

Partition文件下有多个segment（xxx.index，xxx.log）

segment 文件里的 大小和配置文件大小一致可以根据要求修改 默认为1g

如果大小大于1g时，会滚动一个新的segment并且以上一个segment最后一条消息的偏移量命名

**kafka的ack机制**

request.required.acks有三个值 0、1、-1：

0：生产者不会等待broker的ack，这个延迟最低但是存储的保证最弱当server挂掉的时候就会丢数据

1：服务端会等待ack值 leader副本确认接收到消息后发送ack但是如果leader挂掉后他不确保是否复制完成新leader也会导致数据丢失

-1：同样在1的基础上 服务端会等所有的follower的副本受到数据后才会受到leader发出的ack，这样数据不会丢失

**Kafka的消费者如何消费数据**

消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（offset）的位置

等到下次消费时，他会接着上次位置继续消费

**消费者负载均衡策略**

一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如果组中成员太多会有空闲的成员

**数据有序**

一个消费者组里它的内部是有序的

消费者组与消费者组之间是无序的

**kafaka生产数据时数据的分组策略**

生产者决定数据产生到集群的哪个partition中

每一条消息都是以（key，value）格式

Key是由生产者发送数据传入

所以生产者（key）决定了数据产生到集群的哪个partition

1 什么是kafka  
Kafka是分布式发布-订阅消息系统，它最初是由LinkedIn公司开发的，之后成为Apache项目的一部分，Kafka是一个分布式，可划分的，冗余备份的持久性的日志服务，它主要用于处理流式数据。  
2 为什么要使用 kafka，为什么要使用消息队列  
缓冲和削峰：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理。  
解耦和扩展性：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。  
冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。  
健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。  
异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。  
3 kafka中的broker 是干什么的  
broker 是消息的代理，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉取指定Topic的消息，然后进行业务处理，broker在中间起到一个代理保存消息的中转站。  
4 kafka中的 zookeeper 起到什么作用，可以不用zookeeper么  
zookeeper 是一个分布式的协调组件，早期版本的kafka用zk做meta信息存储，consumer的消费状态，group的管理以及 offset的值。考虑到zk本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了zookeeper的作用。新的consumer使用了kafka内部的group coordination协议，也减少了对zookeeper的依赖，  
但是broker依然依赖于ZK，zookeeper 在kafka中还用来选举controller 和 检测broker是否存活等等。  
5 kafka follower如何与leader同步数据  
Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制。完全同步复制要求All Alive Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下，如果leader挂掉，会丢失数据，kafka使用ISR的方式很好的均衡了确保数据不丢失以及吞吐率。Follower可以批量的从Leader复制数据，而且Leader充分利用磁盘顺序读以及send file(zero copy)机制，这样极大的提高复制性能，内部批量写磁盘，大幅减少了Follower与Leader的消息量差。  
6什么情况下一个 broker 会从 isr中踢出去  
leader会维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，每个Partition都会有一个ISR，而且是由leader动态维护 ，如果一个follower比一个leader落后太多，或者超过一定时间未发起数据复制请求，则leader将其重ISR中移除 ，详细参考kafka的高可用机制  
7 kafka 为什么那么快  
CacheFilesystem Cache PageCache缓存  
顺序写由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。  
Zero-copy零拷⻉技术减少拷贝次数  
Batching of Messages批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。  
Pull 拉模式使用拉模式进行消息的获取消费，与消费端处理能力相符。  
8kafka producer如何优化打入速度  
增加线程  
提高 batch.size  
增加更多 producer 实例  
增加 partition 数  
设置 acks=-1 时，如果延迟增大：可以增大 num.replica.fetchers（follower 同步数据的线程数）来调解；  
跨数据中心的传输：增加 socket 缓冲区设置以及 OS tcp 缓冲区设置。  
优化方面的参考kafka最佳实践  
9 kafka producer 打数据，ack 为 0， 1， -1 的时候代表啥， 设置 -1 的时候，什么情况下，leader 会认为一条消息 commit了  
1（默认）数据发送到Kafka后，经过leader成功接收消息的的确认，就算是发送成功了。在这种情况下，如果leader宕机了，则会丢失数据。  
0生产者将数据发送出去就不管了，不去等待任何返回。这种情况下数据传输效率最高，但是数据可靠性确是最低的。  
-1producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。  
当ISR中所有Replica都向Leader发送ACK时，leader才commit，这时候producer才能认为一个请求中的消息都commit了。  
10 kafka unclean 配置代表啥，会对 spark streaming 消费有什么影响  
unclean.leader.election.enable 为true的话，意味着非ISR集合的broker 也可以参与选举，这样有可能就会丢数据，spark streaming在消费过程中拿到的 end offset 会突然变小，导致 spark streaming job挂掉。如果unclean.leader.election.enable参数设置为true，就有可能发生数据丢失和数据不一致的情况，Kafka的可靠性就会降低；而如果unclean.leader.election.enable参数设置为false，Kafka的可用性就会降低。  
11 如果leader crash时，ISR为空怎么办  
kafka在Broker端提供了一个配置参数：unclean.leader.election,这个参数有两个值：true（默认）：允许不同步副本成为leader，由于不同步副本的消息较为滞后，此时成为leader，可能会出现消息不一致的情况。false：不允许不同步副本成为leader，此时如果发生ISR列表为空，会一直等待旧leader恢复，降低了可用性。  
12 kafka的message格式是什么样的  
一个Kafka的Message由一个固定长度的header和一个变长的消息体body组成  
header部分由一个字节的magic(文件格式)和四个字节的CRC32(用于判断body消息体是否正常)构成。  
当magic的值为1的时候，会在magic和crc32之间多一个字节的数据：attributes(保存一些相关属性，  
比如是否压缩、压缩格式等等);如果magic的值为0，那么不存在attributes属性  
body是由N个字节构成的一个消息体，包含了具体的key/value消息  
13 kafka中consumer group 是什么概念  
同样是逻辑上的概念，是Kafka实现单播和广播两种消息模型的手段。同一个topic的数据，会广播给不同的group；同一个group中的worker，只有一个worker能拿到这个数据。换句话说，对于同一个topic，每个group都可以拿到同样的所有数据，但是数据进入group后只能被其中的一个worker消费。group内的worker可以使用多线程或多进程来实现，也可以将进程分散在多台机器上，worker的数量通常不超过partition的数量，且二者最好保持整数倍关系，因为Kafka在设计时假定了一个partition只能被一个worker消费（同一group内）。

## 1. ZooKeeper是什么？

ZooKeeper是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于Zookeeper实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master选举、分布式锁和分布式队列等功能。

Zookeeper保证了如下分布式一致性特性：

* 顺序一致性
* 原子性
* 单一视图
* 可靠性
* 实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，**如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理**。对于写请求，这些请求会同时发给其他zookeeper机器并且达成一致后，请求才会返回成功。因此，随着zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是zookeeper中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper最新的zxid。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 2. ZooKeeper提供了什么？

1、文件系统  
2、通知机制

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 3. Zookeeper文件系统

Zookeeper提供一个多层级的节点命名空间（节点称为znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。  
Zookeeper为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，**这种特性使得Zookeeper不能用于存放大量的数据，每个节点的存放数据上限为1M**。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 4. ZAB协议？

ZAB协议是为分布式协调服务Zookeeper专门设计的一种支持崩溃恢复的原子广播协议。

ZAB协议包括两种基本的模式：崩溃恢复和消息广播。

当整个zookeeper集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与Leader服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的Leader服务器，然后集群中Follower服务器开始与新的Leader服务器进行数据同步，当集群中超过半数机器与该Leader服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 5. 四种类型的数据节点 Znode

* PERSISTENT-持久节点  
  除非手动删除，否则节点一直存在于Zookeeper上
* EPHEMERAL-临时节点  
  临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与zookeeper连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。
* PERSISTENT\_SEQUENTIAL-持久顺序节点  
  基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。
* EPHEMERAL\_SEQUENTIAL-临时顺序节点  
  基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 6. Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper允许客户端向服务端的某个Znode注册一个Watcher监听，当服务端的一些指定事件触发了这个Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

工作机制：

* 客户端注册watcher
* 服务端处理watcher
* 客户端回调watcher

Watcher特性总结：

1. 一次性  
   无论是服务端还是客户端，一旦一个Watcher被触发，Zookeeper都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。
2. 客户端串行执行  
   客户端Watcher回调的过程是一个串行同步的过程。
3. 轻量
   * Watcher通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。
   * 客户端向服务端注册Watcher的时候，并不会把客户端真实的Watcher对象实体传递到服务端，仅仅是在客户端请求中使用boolean类型属性进行了标记。
4. watcher event异步发送watcher的通知事件从server发送到client是异步的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于Zookeeper本身提供了ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper只能保证最终的一致性，而无法保证强一致性。
5. 注册watcher getData、exists、getChildren
6. 触发watcher create、delete、setData
7. 当一个客户端连接到一个新的服务器上时，watch将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 7. 客户端注册Watcher实现

1. 调用getData()/getChildren()/exist()三个API，传入Watcher对象
2. 标记请求request，封装Watcher到WatchRegistration
3. 封装成Packet对象，发服务端发送request
4. 收到服务端响应后，将Watcher注册到ZKWatcherManager中进行管理
5. 请求返回，完成注册。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 8. 服务端处理Watcher实现

1. 服务端接收Watcher并存储  
   接收到客户端请求，处理请求判断是否需要注册Watcher，需要的话将数据节点的节点路径和ServerCnxn（ServerCnxn代表一个客户端和服务端的连接，实现了Watcher的process接口，此时可以看成一个Watcher对象）存储在WatcherManager的WatchTable和watch2Paths中去。
2. Watcher触发  
   以服务端接收到 setData() 事务请求触发NodeDataChanged事件为例：
   * 封装WatchedEvent  
     将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个WatchedEvent对象
   * 查询Watcher  
     从WatchTable中根据节点路径查找Watcher
   * 没找到；说明没有客户端在该数据节点上注册过Watcher
   * 找到；提取并从WatchTable和Watch2Paths中删除对应Watcher（**从这里可以看出Watcher在服务端是一次性的，触发一次就失效了**）
3. 调用process方法来触发Watcher  
   这里process主要就是通过ServerCnxn对应的TCP连接发送Watcher事件通知。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 9. 客户端回调Watcher

客户端SendThread线程接收事件通知，交由EventThread线程回调Watcher。客户端的Watcher机制同样是一次性的，一旦被触发后，该Watcher就失效了。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 10. ACL权限控制机制

### UGO（User/Group/Others）

目前在Linux/Unix文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

### ACL（Access Control List）访问控制列表

包括三个方面：

* 权限模式（Scheme）
  + IP：从IP地址粒度进行权限控制
  + Digest：最常用，用类似于 username:password 的权限标识来进行权限配置，便于区分不同应用来进行权限控制
  + World：最开放的权限控制方式，是一种特殊的digest模式，只有一个权限标识“world:anyone”
  + Super：超级用户
* 授权对象  
  授权对象指的是权限赋予的用户或一个指定实体，例如IP地址或是机器灯。
* 权限 Permission
  + CREATE：数据节点创建权限，允许授权对象在该Znode下创建子节点
  + DELETE：子节点删除权限，允许授权对象删除该数据节点的子节点
  + READ：数据节点的读取权限，允许授权对象访问该数据节点并读取其数据内容或子节点列表等
  + WRITE：数据节点更新权限，允许授权对象对该数据节点进行更新操作
  + ADMIN：数据节点管理权限，允许授权对象对该数据节点进行ACL相关设置操作

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 11. Chroot特性

3.2.0版本后，添加了 Chroot特性，该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了Chroot，那么该客户端对服务器的任何操作，都将会被限制在其自己的命名空间下。

通过设置Chroot，能够将一个客户端应用于Zookeeper服务端的一颗子树相对应，在那些多个应用公用一个Zookeeper进群的场景下，对实现不同应用间的相互隔离非常有帮助。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 12. 会话管理

分桶策略：将类似的会话放在同一区块中进行管理，以便于Zookeeper对会话进行不同区块的隔离处理以及同一区块的统一处理。

分配原则：每个会话的“下次超时时间点”（ExpirationTime）

计算公式：

ExpirationTime\_ = currentTime + sessionTimeout

ExpirationTime = (ExpirationTime\_ / ExpirationInrerval + 1) \* ExpirationInterval , ExpirationInterval 是指 Zookeeper 会话超时检查时间间隔，默认 tickTime

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 13. 服务器角色

### Leader

* 事务请求的唯一调度和处理者，保证集群事务处理的顺序性
* 集群内部各服务的调度者

### Follower

* 处理客户端的非事务请求，转发事务请求给Leader服务器
* 参与事务请求Proposal的投票
* 参与Leader选举投票

### Observer

3.3.0版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力

* 处理客户端的非事务请求，转发事务请求给Leader服务器
* 不参与任何形式的投票

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 14. Zookeeper 下 Server工作状态

服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

* LOOKING：寻找Leader状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入Leader选举状态。
* FOLLOWING：跟随者状态。表明当前服务器角色是Follower。
* LEADING：领导者状态。表明当前服务器角色是Leader。
* OBSERVING：观察者状态。表明当前服务器角色是Observer。

## 17. zookeeper是如何保证事务的顺序一致性的？

zookeeper采用了全局递增的事务Id来标识，所有的proposal（提议）都在被提出的时候加上了zxid，zxid实际上是一个64位的数字，高32位是epoch（时期; 纪元; 世; 新时代）用来标识leader周期，如果有新的leader产生出来，epoch会自增，低32位用来递增计数。当新产生proposal的时候，会依据数据库的两阶段过程，首先会向其他的server发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 18. 分布式集群中为什么会有Master？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行leader选举。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 19. zk节点宕机如何处理？

Zookeeper本身也是集群，推荐配置不少于3个服务器。Zookeeper自身也要保证当一个节点宕机时，其他节点会继续提供服务。  
如果是一个Follower宕机，还有2台服务器提供访问，因为Zookeeper上的数据是有多个副本的，数据并不会丢失；  
如果是一个Leader宕机，Zookeeper会选举出新的Leader。  
ZK集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。  
所以  
3个节点的cluster可以挂掉1个节点(leader可以得到2票>1.5)  
2个节点的cluster就不能挂掉任何1个节点了(leader可以得到1票<=1)

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 20. zookeeper负载均衡和nginx负载均衡区别

zk的负载均衡是可以调控，nginx只是能调权重，其他需要可控的都需要自己写插件；但是nginx的吞吐量比zk大很多，应该说按业务选择用哪种方式。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 21. Zookeeper有哪几种几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 22. 集群最少要几台机器，集群规则是怎样的?

集群规则为2N+1台，N>0，即3台。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 23. 集群支持动态添加机器吗？

其实就是水平扩容了，Zookeeper在这方面不太好。两种方式：

* 全部重启：关闭所有Zookeeper服务，修改配置之后启动。不影响之前客户端的会话。
* 逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5版本开始支持动态扩容。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 24. Zookeeper对节点的watch监听通知是永久的吗？为什么不是永久的?

不是。官方声明：一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。  
一般是客户端执行getData(“/节点A”,true)，如果节点A发生了变更或删除，客户端会得到它的watch事件，但是在之后节点A又发生了变更，而客户端又没有设置watch事件，就不再给客户端发送。  
在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 25. Zookeeper的java客户端都有哪些？

java客户端：zk自带的zkclient及Apache开源的Curator。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 26. chubby是什么，和zookeeper比你怎么看？

chubby是google的，完全实现paxos算法，不开源。zookeeper是chubby的开源实现，使用zab协议，paxos算法的变种。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 27. 说几个zookeeper常用的命令。

常用命令：ls get set create delete等。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 28. ZAB和Paxos算法的联系与区别？

* 相同点：
  + 两者都存在一个类似于Leader进程的角色，由其负责协调多个Follower进程的运行
  + Leader进程都会等待超过半数的Follower做出正确的反馈后，才会将一个提案进行提交
  + ZAB协议中，每个Proposal中都包含一个 epoch 值来代表当前的Leader周期，Paxos中名字为Ballot
* 不同点：  
  ZAB用来构建高可用的分布式数据主备系统（Zookeeper），Paxos是用来构建分布式一致性状态机系统。

[回到顶部](https://www.cnblogs.com/lanqiu5ge/p/9405601.html#_labelTop)

## 29. Zookeeper的典型应用场景

Zookeeper是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对Zookeeper中丰富的数据节点进行交叉使用，配合Watcher事件通知机制，可以非常方便的构建一系列分布式应用中年都会涉及的核心功能，如：

* 数据发布/订阅
* 负载均衡
* 命名服务
* 分布式协调/通知
* 集群管理
* Master选举
* 分布式锁
* 分布式队列

### 1. 数据发布/订阅

#### 介绍

数据发布/订阅系统，即所谓的配置中心，顾名思义就是发布者发布数据供订阅者进行数据订阅。

#### 目的

* 动态获取数据（配置信息）
* 实现数据（配置信息）的集中式管理和数据的动态更新

#### 设计模式

* Push 模式
* Pull 模式

#### 数据（配置信息）特性：

* 数据量通常比较小
* 数据内容在运行时会发生动态更新
* 集群中各机器共享，配置一致

如：机器列表信息、运行时开关配置、数据库配置信息等

#### 基于Zookeeper的实现方式

1. 数据存储：将数据（配置信息）存储到Zookeeper上的一个数据节点
2. 数据获取：应用在启动初始化节点从Zookeeper数据节点读取数据，并在该节点上注册一个数据变更Watcher
3. 数据变更：当变更数据时，更新Zookeeper对应节点数据，Zookeeper会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

### 2. 负载均衡

zk的命名服务  
命名服务是指通过指定的名字来获取资源或者服务的地址，利用zk创建一个全局的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

分布式通知和协调  
对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后zk将这些变化发送给注册了这个节点的watcher的所有客户端。  
对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

7.zk的命名服务（文件系统）  
命名服务是指通过指定的名字来获取资源或者服务的地址，利用zk创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

8.zk的配置管理（文件系统、通知机制）  
程序分布式的部署在不同的机器上，将程序的配置信息放在zk的znode下，当有配置发生改变时，也就是znode发生变化时，可以通过改变zk中某个目录节点的内容，利用watcher通知给各个客户端，从而更改配置。

9.Zookeeper集群管理（文件系统、通知机制）  
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。

10.Zookeeper分布式锁（文件系统、通知机制）  
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。

11.获取分布式锁的流程  
clipboard.png

在获取分布式锁的时候在locker节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用createNode方法在locker下创建临时顺序节点，  
然后调用getChildren(“locker”)来获取locker下面的所有子节点，注意此时不用设置任何Watcher。客户端获取到所有的子节点path之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非locker所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，然后对其调用exist()方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的Watcher会收到相应通知，此时再次判断自己创建的节点是否是locker子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。  
clipboard.png

代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于BaseDistributedLock，实现了基于Zookeeper实现分布式锁的细节。

12.Zookeeper队列管理（文件系统、通知机制）  
两种类型的队列：  
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。在特定的目录下创建PERSISTENT\_SEQUENTIAL节点，创建成功时Watcher通知等待的队列，队列删除序列号最小的节点用以消费。此场景下Zookeeper的znode用于消息存储，znode存储的数据就是消息队列中的消息内容，SEQUENTIAL序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

## 1.ZooKeeper是什么？

ZooKeeper是一个**分布式**的，开放源码的分布式**应用程序协调服务**，是Google的Chubby一个开源的实现，它是**集群的管理者**，**监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作**。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。  
客户端的**读请求**可以被集群中的**任意一台机器处理**，如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理。对于**写请求**，这些请求会同**时发给其他zookeeper机器并且达成一致后，请求才会返回成功**。因此，随着**zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降**。  
有序性是zookeeper中非常重要的一个特性，所有的**更新都是全局有序的**，每个更新都有一个**唯一的时间戳**，这个时间戳称为**zxid（Zookeeper Transaction Id）**。而**读请求只会相对于更新有序**，也就是读请求的返回结果中会带有这个**zookeeper最新的zxid**。

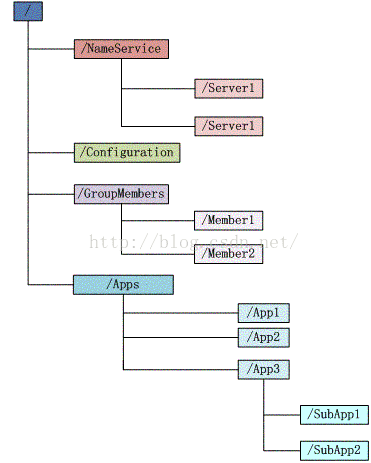
## 2.ZooKeeper提供了什么？

1、**文件系统**  
2、**通知机制**

## 3.Zookeeper文件系统

Zookeeper提供一个多层级的节点命名空间（节点称为znode）。与文件系统不同的是，这些节点**都可以设置关联的数据**，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得Zookeeper**不能用于存放大量的数据**，每个节点的存放数据上限为**1M**。

## 4.四种类型的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  


## 5.Zookeeper通知机制

client端会对某个znode建立一个**watcher事件**，当该znode发生变化时，这些client会收到zk的通知，然后client可以根据znode变化来做出业务上的改变等。

## 6.Zookeeper做了什么？

1、命名服务  
2、配置管理  
3、集群管理  
4、分布式锁  
5、队列管理

## 7.zk的命名服务（文件系统）

命名服务是指通过指定的名字来**获取资源**或者**服务的地址**，利用zk创建一个全局的路径，即是**唯一**的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

## 8.zk的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在zk的**znode**下，当有配置发生改变时，也就是znode发生变化时，可以通过改变zk中某个目录节点的内容，利用**watcher**通知给各个客户端，从而更改配置。

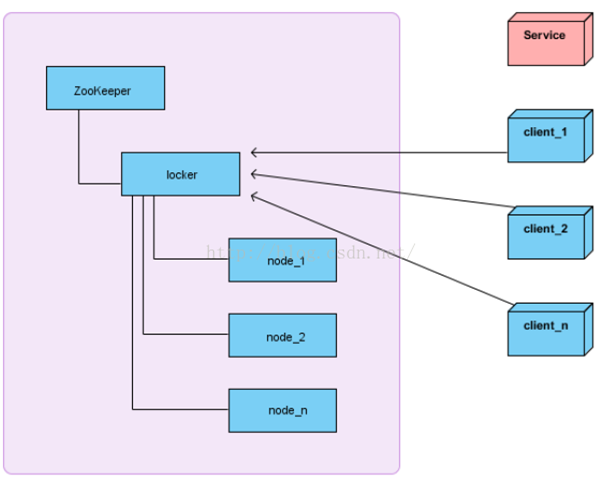
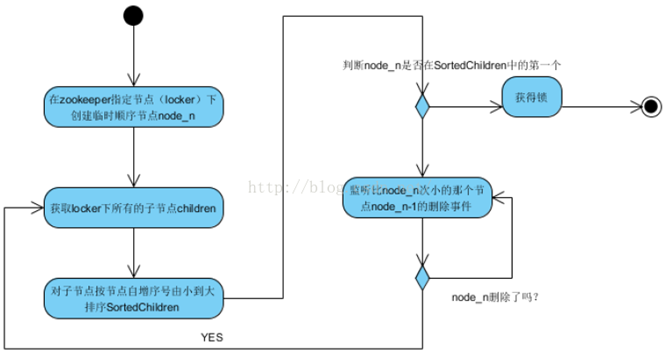
## 9.Zookeeper集群管理（文件系统、通知机制）

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**。

## 10.Zookeeper分布式锁（文件系统、通知机制）

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。

## 11.获取分布式锁的流程

  
在获取分布式锁的时候在locker节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用createNode方法在locker下创建临时顺序节点，  
然后调用getChildren(“locker”)来获取locker下面的所有子节点，注意此时不用设置任何Watcher。客户端获取到所有的子节点path之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非locker所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到**比自己小的那个节点**，然后对其调用**exist()**方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的Watcher会收到相应通知，此时再次判断自己创建的节点是否是locker子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。  


代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于**BaseDistributedLock**，实现了基于Zookeeper实现分布式锁的细节。

## 12.Zookeeper队列管理（文件系统、通知机制）

两种类型的队列：  
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。在特定的目录下创建**PERSISTENT\_SEQUENTIAL**节点，创建成功时**Watcher**通知等待的队列，队列删除**序列号最小的节点**用以消费。此场景下Zookeeper的znode用于消息存储，znode存储的数据就是消息队列中的消息内容，SEQUENTIAL序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以**不必担心队列消息的丢失问题**。

## 13.Zookeeper数据复制

Zookeeper作为一个集群提供一致的数据服务，自然，它要在**所有机器间**做数据复制。数据复制的好处：   
1、容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；   
2、提高系统的扩展能力 ：把负载分布到多个节点上，或者增加节点来提高系统的负载能力；   
3、提高性能：让**客户端本地访问就近的节点，提高用户访问速度**。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：   
1、**写主**(WriteMaster) ：对数据的**修改提交给指定的节点**。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称**读写分离**；   
2、**写任意**(Write Any)：对数据的**修改可提交给任意的节点**，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

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**。

## 14.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。

## 15.zookeeper是如何保证事务的顺序一致性的？

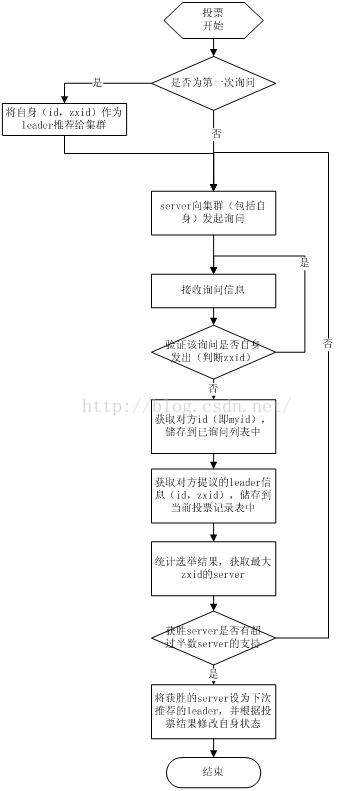
zookeeper采用了**递增的事务Id**来标识，所有的proposal（提议）都在被提出的时候加上了zxid，zxid实际上是一个64位的数字，高32位是epoch（时期; 纪元; 世; 新时代）用来标识leader是否发生改变，如果有新的leader产生出来，epoch会自增，**低32位用来递增计数**。当新产生proposal的时候，会依据数据库的两阶段过程，首先会向其他的server发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

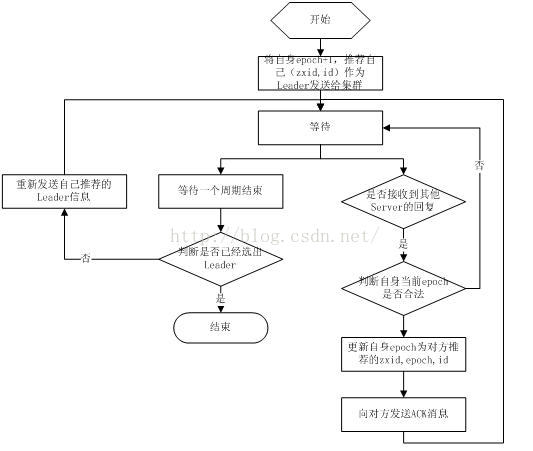
## 16.Zookeeper 下 Server工作状态

每个Server在工作过程中有三种状态：   
LOOKING：当前Server**不知道leader是谁**，正在搜寻  
LEADING：当前Server即为选举出来的leader  
FOLLOWING：leader已经选举出来，当前Server与之同步

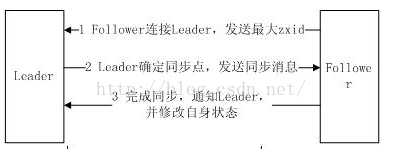
## 17.zookeeper是如何选取主leader的？

当leader崩溃或者leader失去大多数的follower，这时zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为**fast paxos**。

1、Zookeeper选主流程(basic paxos)  
（1）选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；   
（2）选举线程首先向所有Server发起一次询问(包括自己)；   
（3）选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；   
（4）收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；   
（5）线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数，设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。 通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1. 每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。  


2、Zookeeper选主流程(basic paxos)  
fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和 zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。  


## 18.Zookeeper同步流程

选完Leader以后，zk就进入状态同步过程。   
1、Leader等待server连接；   
2、Follower连接leader，将最大的zxid发送给leader；   
3、Leader根据follower的zxid确定同步点；   
4、完成同步后通知follower 已经成为uptodate状态；   
5、Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。  


## 19.分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台**改变某个节点的状态**，**然后zk将这些变化发送给注册了这个节点的watcher的所有客户端**。  
对于执行情况汇报：每个工作进程都在某个目录下**创建一个临时节点**。**并携带工作的进度数据**，这样**汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况**。

## 20.机器中为什么会有leader？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，**其他的机器可以共享这个结果**，这样可以大大**减少重复计算**，**提高性能**，于是就需要进行leader选举。

## 21.zk节点宕机如何处理？

Zookeeper本身也是集群，推荐配置不少于3个服务器。Zookeeper自身也要保证当一个节点宕机时，其他节点会继续提供服务。  
如果是一个Follower宕机，还有2台服务器提供访问，因为Zookeeper上的数据是有多个副本的，数据并不会丢失；  
如果是一个Leader宕机，Zookeeper会选举出新的Leader。  
ZK集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。  
所以  
3个节点的cluster可以挂掉1个节点(leader可以得到2票>1.5)  
2个节点的cluster就不能挂掉任何1个节点了(leader可以得到1票<=1)

## 22.zookeeper负载均衡和nginx负载均衡区别

zk的负载均衡是可以调控，nginx只是能调权重，其他需要可控的都需要自己写插件；但是nginx的吞吐量比zk大很多，应该说按业务选择用哪种方式。

## 23.zookeeper watch机制

Watch机制官方声明：一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。  
Zookeeper机制的特点：  
1、一次性触发数据发生改变时，一个watcher event会被发送到client，但是client**只会收到一次这样的信息**。  
2、watcher event异步发送watcher的通知事件从server发送到client是**异步**的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于**网络延迟或其他因素导致客户端在不通的时刻监听到事件**，由于Zookeeper本身提供了**ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化**。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper**只能保证最终的一致性，而无法保证强一致性**。  
3、数据监视Zookeeper有数据监视和子数据监视getdata() and exists()设置数据监视，getchildren()设置了子节点监视。  
4、注册watcher **getData、exists、getChildren**  
5、触发watcher **create、delete、setData**  
6、**setData()**会触发znode上设置的data watch（如果set成功的话）。一个成功的**create()** 操作会触发被创建的znode上的数据watch，以及其父节点上的child watch。而一个成功的**delete()**操作将会同时触发一个znode的data watch和child watch（因为这样就没有子节点了），同时也会触发其父节点的child watch。  
7、当一个客户端**连接到一个新的服务器上**时，watch将会被以任意会话事件触发。当**与一个服务器失去连接**的时候，是无法接收到watch的。而当client**重新连接**时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，**watch可能会丢失**：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。  
8、Watch是轻量级的，其实就是本地JVM的**Callback**，服务器端只是存了是否有设置了Watcher的布尔类型