面试题目总结

# Java篇

## Java的拷贝

Java中有浅拷贝和深拷贝之说，对于浅拷贝，当对象中的数据类型是基本类型时，浅拷贝直接进行值拷贝，也就是将该属性的值复制一份给新的对象，对其中一个对象的某个值进行修改不影响另一个对象的值，当对象中的数据类型包含引用类型时（比如成员变量为数组，类的对象），那么浅拷贝会进行引用拷贝，也就是将该成员的引用值（内存地址）复制一份给新的对象，实际上两个对象的该成员指向同一个实例，在一个对象中修改该成员变量会影响另一个对象的该成员的值。

## LinkdList 和 ArrayList 的区别

LinkdList与ArrayList都是List的实现类，但二者的底层存储不同，ArrayList使用数组作为存储，查找速度快。LinkdList使用链表作为存储，删除，插入时只需要移动指针，不像ArrayList需要移动元素，比ArrayList的速度快，二者都是线程不安全，可以动态扩容（ArrayList扩容为当前容量的1.5倍）。当对查找速度有要求时，建议采用ArrayList，对插入，删除有要求时，采用后者。

## ArrayList 与 Vector 的区别

Vector的方法是同步的(Synchronized),是线程安全的(thread-safe)，而ArrayList的方法不是，由于线程的同步必然要影响性能，因此,ArrayList的性能比Vector好。 当Vector或ArrayList中的元素超过它的初始大小时,Vector会将它的容量翻倍,而ArrayList只增加50%的大小，这样,ArrayList就有利于节约内存空间。当容量不够时进行扩容，使用Arrays.copyOf(),把原数组复制到新的数组中，这个操作代价很高，因此在创建ArrayList对象时就指定大概的容量大小，减少扩容操作的次数

## hashMap，为什么会使用红黑树而不是平衡二叉树

红黑树牺牲了一些查找性能 但其本身并不是完全平衡的二叉树。因此插入删除操作效率略高于AVL树。AVL树用于自平衡的计算牺牲了插入删除性能，但是因为最多只有一层的高度差，查询效率会高一些。

## http 传输的时候是明文的，怎么解决安全问题

采用https，访问采用加密方式传输，中间者无法直接查看原始内容

## STRING 与 STRINGBUFFER 的区别

String类一旦产生一个字符串，其对象就不可变。String类的内容和长度是固定的。如果程序需要获得字符串的信息需要调用[系统](https://www.2cto.com/os/)提供的各种字符串操作方法实现。虽然通过各种系统方法可以对字符串施加操作，但这并不改变对象实例本身，而是生成一个新的实例。Buffer是缓冲的意思，这个类有缓冲的功能。该类处理可变的字符串。如果要修改一个StringBuffer类的字符串，不需要再创建新的字符串对象，而是直接操作原来的串。

## String s = new String("xyz");创建了几个 String Object?并作说明

产生两个对象，首先是new时产生一个，然后赋给s时产生一个

## short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 += 1;有什么错?

第一个会报错，在s1+1时转换成int型，而s1是short类型，直接赋值会导致类型不匹配，可以通过类型转换解决 s1 = （short）（s1+1）。后一个不报错。因为使用了隐式转换

## 垃圾回收机制

Java的垃圾回收机制有引用计数法，标记清除算法，标记整理算法，复制回收算法，分代回收算法。

**引用计数算法：**在堆中为每个对象创建一个引用计数器，当对象被引用时计数器加一，当引用被置为空或者离开作用域，引用计数器减一，当引用计数器为零时，该对象被回收，该方法执行效率高，但不能解决互相引用问题

**标记清除算法：**分为标记和清除阶段，标记阶段：从应用程序的根对象开始遍历，每一个可以从根对象访问到的对象被标记，否者不标记。清除阶段：从根对象开始，清除那些没有被标记的对象，该算法会产生大量的不连续内存碎片。

**标记整理算法**：标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

**复制回收算法：**在内存中将堆分成两个大小相等块，任何时候只有一个堆被使用。当该堆快被使用完时，此时垃圾回收器会停止程序的运行，将所有还活动的对象复制到另一个堆中，前面那个堆被清空。

**分代回收算法：**根据堆中对象的年龄，划分老年代与新生代。新生代的存活率低，回收速度快，采用复制回收算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收

## Minor GC与Full GC的出发条件及区别

Minor GC在Eden区满时触发，

Full GC在：

调用System.gc，系统建议执行Full GC，但不必然执行

老年代空间不足

方法区空间不足

## 造成内存泄漏的几种情况

**静态集合类引起的内存泄漏：**像HashMap，Vector等的使用最容易出现内存泄漏，这些静态变量的生命周期和应用程序一致，他们所引用的对象object也不能被释放，因为他们被vector等集合引用着。

**各种连接：**比如数据库连接，网络连接，除非显示调用了close方法，否者不会被gc回收掉。

## 运行时的数据区域

1. **程序计数器：**记录正在执行的字节码指令地址
2. **Java虚拟机栈：**每个java方法在执行的同时会创建一个栈帧用于存储局部变量表，操作数栈，常量池引用等信息，从方法调用到完成的过程对应一个栈帧在java虚拟机栈中入栈和出栈的过程
3. **本地方法栈：**与java虚拟机栈类似，只是本地方法栈为本地方法服务
4. **堆：**所有的对象在这里分配内存，是垃圾收集的主要区域，可以通过-Xms 和 –Xmx指定堆的大小
5. **方法区：**用于存放已被加载的类信息，常量，静态变量，即时编译器编译后的代码等数据

## 反射

在运行状态中，对于任意一个类，都能够知道这个类的属性和方法，对于任意一个类都能够调用它的属性和方法。这种动态的获取信息和动态调用对象的方法的功能叫做反射

优点：

1. 可扩展性：可以使用全限定名创建可扩展对象的实例，来使用来自外部的用户自定义类

缺点：

1. 性能开销：反射涉及了动态类型解析，所以jvm无法对这些代码进行优化。因此反射的操作效率比那些非反射的操作效率低
2. 安全限制：使用反射技术必须要在一个没有安全限值的环境中运行
3. 内部暴露：反射破环了对象的封装性

## Volatile

不保证原子性

保证此变量对所有线程可见

禁止指令重排序优化

## ReentrantLock

是一个可重入的互斥锁，又被称为独占锁。ReentrantLock在同一时间点只能被一个线程访问。但可以被单个线程多次获取。ReentrantLock包含公平锁和非公平锁，体现在获取锁的机制上。锁是为了保护竞争资源，防止多个线程访问而出错。ReentrantLock通过一个FIFO队列来管理获取该锁的所有线程。在公平锁机制下，线程依次排队获取锁，而非公平锁机制下，不管线程是否在队列的开头，都会竞争锁。**在finally中需要释放锁。**

## ReentrantLock与sychronized的区别

除了synchronized的功能，多了三个高级功能

1. 等待可中断，在持有锁的线程长时间不释放锁的时候，等待的线程可以放弃等待。tryLock(long timeout, TimeUnit unit)
2. 公平锁，按照申请锁顺序来依次获取锁称为公平锁，sychronized是非公平锁。ReentrantLock可以通过构造函数实现公平锁。new RenentrantLock(boolean fair)
3. 绑定多个Condition：通过多次newCondition可以获得多个Condition对象，可以简单的实现比较负责的线程同步的功能，通过await(),signal();

除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized。这是因为 synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持。并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放。

## Sleep与wait的区别

1. Sleep是Thread方法，wait是Object方法。
2. 调用sleep时，只会让当前线程暂停运行，不会释放获得的锁，而wait方法会释放获得的锁。
3. Wait方法需要在获得锁之后才能调用(通常在synchronized方法中调用)，否则会出现异常IllegalMonitorStateException

## 线程池

Java中有五种线程池，分别为CacheThreadPool，FixedThreadPool，SingleThreadExecutor，ScheduleThreadPool，ScheduledThreadPoolExecutor。他们分别通过Executors的静态方法创建出来的，底层是通过ThreadPollExecutor实现。

1. CacheThreadPool：工作线程的创建数量几乎没有限制，可以灵活的往线程池中添加线程如果长时间没有往线程池中添加任务，即如果工作线程空闲了指定的时间（默认1分钟）则该工作线程终止，终止后如果又提交了新的任务，则线程池重新创建一个工作线程。在使用CacheThreadPool时，一定要注意控制任务的数量，否者由于大量线程同时运行，很有可能造成系统瘫痪
2. FixedThreadPool：创建一个指定线程数量的线程池，每提交一个任务就创建一个工作线程，如果工作线程的数目达到线程初始的最大数，则新提交的任务存放到等待队列中。FixedThreadPool是一个优秀的线程池，它具有线程池提高效率和节省创建线程时所耗的开销。但是在线程池空闲时，也不会释放工作线程，还会占用一定的系统资源
3. SingleThreadExecutor：创建单线程的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有的任务按照指定的顺序执行。如果这个线程异常结束，则会有下一个取代他，保证顺序执行。
4. ScheduleThreadPool：创建一个定长的线程池，而且支持定时的以及周期性的任务执行

## Shutdown与shutdownNow的区别

Shutdown设置状态为Shutdown，而shutdownNow设置状态为stop

Shutdown只中断空闲线程，已提交的线程继续运行。而shutdownNow中断所有的线程

Shutdown无返回值，shutdownNow返回值中还有未执行的任务

## 什么是CAS

比较并交换，它的作用是将指定内存地址的内容与所给的某个值作比较，如果相等，则将其内容替换为指令中提供的新值，如果不等，则更新失败。从内存领域中来看就是乐观锁，因为它在对共享变量更新之前会比较当前值是否与更新前的值是否相等，如果相等，则更新否则无限循环，直到当前值与更新前的值相等，才执行更新。

# 数据库篇

## 索引的实现方式

B+树索引：用于范围查询和单值查询都可以，特别是范围查询

散列索引：Hash索引适用于单值查询，同时该索引存在一些弊端，无法对范围查询优化，无法利用前缀索引，无法堆排序进行优化，必须回行即通过索引拿到表的位置，必须回到表中取数据。

位图索引：适用于值得类型很少情况，比如男女

## InnoDB与Myisam引擎的区别

二者都是用BTree索引，但是Myisam使用非聚集索引，索引表与数据表相分离，其索引指向某行在磁盘上的位置，不同索引树之间并无关联。InnoDB使用聚集索引，数据和索引一块存放，其索引指向对主键的引用，非主键索引树指向主键索引

## 数据库优化

1. Scheme设计与数据类型优化：选择数据类型只要遵循小而简单的原则就好，越小的数据类型通常会更快，占用更少的磁盘，内存，处理时需要的cpu周期也更少。
2. **Mysql不会使用索引的情况：非独立列**，指的是索引列不能是表达式的一部分，也不能是函数的参数，比如select \* from A where id+1=5,很容易看出其等价于id=4，但是mysql无法自动解析这个表达式，使用函数也是同样的道理
3. 前缀索引：如果列很长，通常可以使用索引列开始的部分字符，这样可以节约索引空间提高索引效率，**只适用于最左前缀查找**
4. 在多数情况下，在多个列上建立独立索引并不能提高查询性能，mysql不知道使用哪个索引的查询效率更好。这种情况可以通过建立联合索引
5. 使用是索引扫描来排序，只有当索引列的顺序和order by的字句的顺序完全一致，并且所有列的排序方向也一样时，才能够使用索引来对结果排序

# Spark篇

## RDD特性

Spark采用RDD以后能够实现高效计算的主要原因如下

1. 高效的容错性：在RDD的设计中，数据只读，不可修改，如果要修改数据，必须从父RDD转换到子RDD，由此在不同的RDD之间建立了血缘关系。
2. 不同RDD之间的转换操作形成依赖关系，可以实现管道化，避免了中间结果的存储，大大降低了数据复制，磁盘IO和序列化开销。
3. 每个RDD可以分成多个分区，不同分区可以被保存到不同的节点上，从而可以在不同节点上进行并行计算

## Watermark

Watermark用于处理乱序事件的，通常结合窗口来实现。流处理从事件产生，到流经source，再到operator，中间是有一个过程和时间的。虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络、背压等原因，导致乱序的产生。但是对于late element，我们又不能无限期的等下去，必须要有个机制来保证一个特定的时间后，必须触发window去进行计算了。这个特别的机制，就是watermark。

## Spark任务调度过程

1. 客户端向资源调度平台的master提交任务
2. Master根据收到的请求与nodemanager通信，开启一个executor运行applicationmaster，然后AM向master注册
3. AM内部启动dagscheduleiml将DAG图划分为stage，生成taskset，交给taskschedule进行调度处理
4. AM向master申请任务运行需要的资源，然后与nodemanager通信，开启container
5. NM向AM请求任务
6. AM通过taskscheduleiml向NM分发任务

## Hadoop的调度器有哪些

有三种调度器，分别为默认调度器FIFO，计算能力调度器Capacity Scheduler，公平调度器Fair Scheduler

1. FIFO：先进先出
2. 计算能力调度器Capability Scheduler：选择占用资源小，优先级高的任务调度
3. 公平调度器Fair Scheduler：同一队列中的作业公平共享队列中的所有资源

## Spark Sql调优点

对一些SparkSQL任务，可以通过缓存数据、调优参数、增加并行度提升性能。

**设置缓存：**Spark SQL可以通过调用 spark.catalog.cacheTable("tableName")或者dataFrame.cache()方法来缓存表。然后，Spark将会仅仅浏览需要的列并且自动地压缩数据以减少内存的使用以及垃圾回收的压力。可以通过调用spark.catalog.uncacheTable("tableName")方法在内存中删除表。缓存时可以指定是否压缩setConf("spark.sql.inMemoryColumnarStorage.compressed","true") ，为每列自动选择压缩码，setConf("spark.sql.inMemoryColumnarStorage.batchSize","1000") ，列式缓存的批处理大小，大批量可以提升内存使用率和压缩了，但是缓存是会有溢出风险

参数调优：

**增加并行度：**Spark采用内存列式存储，实际执行查询效率很高，相对而言数据加载阶段耗时较长，合理设置并行度提升文件加载效率

## Spark数据倾斜解决办法

1. 如果partition的数据是从外部获取，应当保证数据是可以split的，并且保证split后的块是均衡的。
2. 如果是shuffle partition，可以通过调整shuffle partition的数目来避免某个shuffle partition的数据量过大导致的数据倾斜
3. 如果某个key的数据量非常大，调整partition的数据也不能解决数据倾斜问题，可以通过对key加一些前缀来分散数据
4. 从shuffle的角度出发，如果两个join表中一个是小表，可以将小表广播出去，变成BroadcastHashJoin来消除shuffle，从而消除shuffle引起的数据倾斜。

## MapReduce数据倾斜问题及解决办法

在mr模型中数据倾斜通常是因为key的分化严重不均，导致一部分数据很多，而另一部分数据很少的情况。

1. 调大reduce的内存（增加reduce数目适用于唯一值较多的情况）
2. 在map端进行聚合
3. 自定义分区，继承partition类
4. 重新设计key，在key的前面加一个随机数，待到reduce时再把随机数去掉

## Spark shuffle

Spark在DAG调度阶段会将一个Job划分为多个Stage，上游Stage做map工作，下游Stage做reduce工作，其本质上还是MapReduce计算框架。Shuffle是连接map和reduce之间的桥梁，它将map的输出对应到reduce输入中，这期间涉及到序列化反序列化、跨节点网络IO以及磁盘读写IO等，所以说Shuffle是整个应用程序运行过程中非常昂贵的一个阶段

## Spark调优

1. 数据序列化
2. 内存调优：主要包含使用多少内存，接收所有对象的开销，垃圾回收的开销。**Java对象**处理速度快，但是比原始数据大2到5倍，因为对象中有16字节的对象头。**Java字符串**在原始字符串数据上有大约40个字节的开销（因为它们将它存储在Chars数组中并保留额外的数据，如长度），并且由于String内部使用UTF-16编码而将每个字符存储为两个字节。 因此，10个字符的字符串可以轻松消耗60个字节。主要调节执行内存与存储内存
3. 使用简单的数据结构，避免集合类等，使用基本类型和数组
4. 调大并行度，可以将并行级别作为第二个参数传递（请参阅spark.PairRDDFunctions文档），或者设置config属性spark.default.parallelism以更改默认值。 通常，我们建议群集中每个CPU核心有2-3个任务。
5. 广播大变量，可以大大减少每个序列化任务的大小，以及在群集上启动作业的成本
6. 数据本地性
7. 持久化RDD
8. 使用高效的算子。比如使用reduceByKey，aggregateByKey取代groupbyKey，使用mapPartitions 替代普通 map。使用 foreachPartitions 替代 foreach。使用filter 之后进行 coalesce 操作：通常对一个 RDD 执行 filter 算子过滤掉RDD 中以后比较多的数据后，建议使用 coalesce 算子，手动减少 RDD 的partitioning 数量，将 RDD 中的数据压缩到更少的 partition 中去，只要使用更少的 task 即可处理完所有的 partition，在某些场景下对性能有提升
9. Shuffle调优

## Spark shuffle调优

1. **spark.shuffle.file.buffer:**设置shuffle writer缓冲区的buffer大小，在资源充足的情况下可以设置大一些（默认32K），减少shuffle writer过程中溢写磁盘的次数，提升性能
2. **spark.reducer.maxSizeInFlight:**设置shuffle read task的buffer缓冲区大小，决定每次能够拉取的数据大小
3. **spark.shuffle.io.maxRetries：**shuffle reader拉取数据失败时最大重试次数。对于那些特别耗时的shuffle操作的作业，建议增大重试次数，以避免由于jvm的full gc或者网络导致的数据拉取失败，可以提升稳定性
4. **spark.shuffle.memoryFraction:** 代表executor内存中分配给shuffle reader task执行聚合操作的内存比例。如果内存充足而且很少使用持久化操作，建议调高和这个比例，给 给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘

## spark资源参数设置

1. **num-executors:** 用于设置Spark作业总共要用多少个Executor进程来执行。每个spark作业建议设置50~100个较为合适
2. **executor-memory：**设置每个executor的内存，建议设置4-8G。num-executors乘以executor-memory，就代表Spark作业申请到的总内存量，不要逃超过总内存的1/3-1/2
3. **executor-cores:**决定了每个executor并行执行task线程的能力。建议设置2-4比较合适num-executors \* executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适
4. **driver-memory：**一般默认1G就足够了，如果需要使用collect等算子将数据全部拉取到driver上处理，则需要确保driver有足够的内存，以免oom
5. **spark.default.parallelism:**用于设置每个stage默认的task数。这个参数极为重要。Spark官网建议的设置原则是，设置该参数为num-executors \* executor-cores的2~3倍较为合适
6. **spark.storage.memoryFraction:**设置RDD持久化数据在executor内存中占有的比例。当shuffle操作较多时应当调小，如果有较多的持久化应当调大
7. **spark.shuffle.memoryFraction:**

**spark-submit \**

**--master yarn-cluster \**

**--num-executors 100 \**

**--executor-memory 6G \**

**--executor-cores 4 \**

**--driver-memory 1G \**

**--conf spark.default.parallelism=1000 \**

**--conf spark.storage.memoryFraction=0.5 \**

**--conf spark.shuffle.memoryFraction=0.3 \**

## Spark比hadoop快的原因

1. spark使用先进的DAG执行引擎，支持循环数据流与内存计算，基于内存的执行速度比hadoop快上百倍，基于磁盘的执行速度也能快上10倍
2. 提供众多的算子，操作更方便
3. Spark提供了基于内存的计算，中间结果直接放到内存中，带来了更高的迭代运算效率

## Spark为什么快，Spark SQL一定比Hive快吗

Spark SQL 比Hadoop Hive快，是有一定条件的，而且不是 Spark SQL的引擎比Hive的引擎快，相反，Hive 的 HQL 引擎还比 Spark SQL 的引擎更快。其实，关键还是在于 Spark 本身快。

**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，也是 Hive 更慢的主要原因了

**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。

**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

## Spark的依赖容错机制

Spark的依赖分为窄依赖和宽依赖，**窄依赖**是父rdd的每个分区最多被一个子rdd的分区引用，表现为一个父rdd的分区对应一个子rdd的分区或者多个父rdd的分区对应于一个子rdd的分区。**宽依赖**是子rdd的分区依赖于父rdd的多个分区或者全部分区，即存在一个父rdd的一个分区对应一个子rdd的多个分区。

在容错机制中，如果一个节点死机了，而且运算窄依赖，则只要把丢失的父RDD分区重算即可，不依赖于其他节点。而宽依赖需要父RDD的所有分区都存在，重算就很昂贵了。可以这样理解开销的经济与否：在窄依赖中，在子RDD的分区丢失、重算父RDD分区时，父RDD相应分区的所有数据都是子RDD分区的数据，并不存在冗余计算。在宽依赖情况下，丢失一个子RDD分区重算的每个父RDD的每个分区的所有数据并不是都给丢失的子RDD分区用的，会有一部分数据相当于对应的是未丢失的子RDD分区中需要的数据，这样就会产生冗余计算开销，这也是宽依赖开销更大的原因。因此如果使用Checkpoint算子来做检查点，不仅要考虑Lineage是否足够长，也要考虑是否有宽依赖，对宽依赖加Checkpoint是最物有所值的。

## Checkpoint机制与cache

当 spark 应用程序特别复杂，从初始的 RDD 开始到最后整个应用程序完成有很多的步骤，而且整个应用运行时间特别长，这种情况下就比较适合使用 checkpoint 功能。

最主要的区别在于持久化只是将数据保存在 BlockManager 中，但是 RDD 的 lineage(血

缘关系，依赖关系)是不变的。但是 checkpoint 执行完之后，rdd 已经没有之前所谓的依赖 rdd了，而只有一个强行为其设置的 checkpointRDD，checkpoint 之后 rdd 的 lineage 就改变了。持久化的数据丢失的可能性更大，因为节点的故障会导致磁盘、内存的数据丢失。但是checkpoint 的数据通常是保存在高可用的文件系统中，比如 HDFS 中，所以数据丢失可能性比较低。

## Spark宕机的几种情况和解决办法

**Driver进程宕机：**1）如果是机器宕机，则重启机器及spark程序。2）如果是应用程序执行失败，则重启程序即可。在sparkStreaming中，重启程序可以通过checkpoint进行job数据恢复

**Executor进程宕机：**选择一个worker节点重新启动Executor进程，Driver重新分配任务

**Task执行失败：**1）spark程序会进行任务重试，如果某个task失败重试超过3次后，当前job失败。2）task数据恢复/重新运行的机制实际上是RDD的容错机制。3）如果血缘过长，可以通过设置检查点或缓存，将数据冗余的保存下来，当task出现异常时，可以在检查点处构建血缘关系，进行错误恢复，减少执行开销

## Yarn clent与yarn cluster的区别

从广义上讲，yarn cluster模式适合生产环境，而yarn clent模式适合于交互和调试，可以快速看到application的输出。从深层含义来讲，二者的主要区别是application master进程的区别。Yarn cluster模式下，driver运行在application master中，负责向yarn申请资源，监督作业的运行情况。当用户提交作业后，client就可以关闭了，作业会继续在yarn上运行。然而cluster模式不适合运行交互类型的作业。而在clent模式下，application master仅仅向yarn申请container，client跟container通信来调度他们的工作，也就是说client不能离开

## 减少内存的使用方式

1. 数据结构优先使用基本类型和对象数组，而不是标准的java或scala集合类（例如hashmap）
2. 尽量避免使用包含大量小对象和指针的嵌套结构
3. 序列化rdd存储，使用kryo方式，存储格式选用MEMORY\_ONLY\_SER

## Spark的序列化方式

以序列化形式存储RDD，以减少内存使用以及gc的开销

在spark中具有两种序列化方式，分别是java序列化，通过继承Serializable实现，另一种是kryo序列化。

Java序列化方式使用灵活，但是速度慢。在某些情况下序列化的结果也比较大

Kryo比Java序列化（通常高达10倍）明显更快，更紧凑，但不支持所有Seri​​alizable类型，并且要求提前注册您将在程序中使用的类，以获得最佳性能。

conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer"

val conf = new SparkConf().setMaster(...).setAppName(...)

要使用Kryo注册自己的自定义类，使用registerKryoClasses方法。 conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))

val sc = new SparkContext(conf)

如果没有注册自定义类，Kryo仍然会工作，但它必须存储每个对象的完整类名，这是浪费的。

## 防止OOM可能的方法

1. 如果Minor GC过多，说明Eden的内存过少，可以添加Eden的内存
2. 添加并行度，减少每个task的数据量
3. 广播大变量

## sparkStreaming调优

序列化，广播大变量，设置合理的批处理间隔，缓存反复使用的 Dstream（RDD）

合理的 Kafka 拉取量（maxRatePerPartition 参数设置）：对于数据源是 Kafka 的 Spark Streaming 应用，在 Kafka 数据频率过高的情况下，调整这个参数是非常必要的。我们可以改变 spark.streaming.kafka.maxRatePerPartition 参数的值来进行上限调整，默认是无上限的，即 Kafka 有多少数据，Spark Streaming 就会一次性全拉出，但是上节提到的批处理时间是一定的，不可能动态变化，如果持续数据频率过高，同样会造成数据堆积、阻塞的现象

## groupbykey与reducebykey的区别

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

## 执行一个sparkStreaming程序需要注意的事项

1. 将程序打包成一个jar包，通过spark-submit提交，对于一些额外的依赖包，比如kafka的，可以在提交时指定，通过—package
2. 为每个executor配置足够的内存，因为接收到的流数据存放在内存中，比如十分钟一个批次，则需要能够存放十分钟数据的内存
3. 配置检查点，用于故障恢复
4. 配置应用程序的驱动程序的自动重新启动，要自动从驱动程序故障中恢复，用于运行流应用程序的部署基础结构必须监视驱动程序进程并在驱动程序失败时重新启动驱动程序。 不同的集群管理器有不同的工具来实现这一点
5. 配置预写日志，自Spark 1.2起，我们引入了预写日志以实现强大的容错保证。如果启用，则从接收器接收的所有数据都将写入配置检查点目录中的预写日志。这可以防止驱动程序恢复时的数据丢失，从而确保零数据丢失。将

spark.streaming.receiver.writeAheadLog.enable设置为true来启用此功能。然而，这些更强的语义可能以单个接收器的接收吞吐量为代价。这可以通过并行运行更多接收器来更正聚合吞吐量来纠正。此外，建议在启用预写日志时禁用Spark中接收数据的复制，因为日志已存储在复制存储系统中。

1. 设置最大接收速率，如果群集资源不足以使流应用程序以接收数据的速度处理数据，则可以通过设置记录/秒的最大速率限制来限制接收器。

spark.streaming.receiver.maxRate和Direct Kafka方法的spark.streaming.kafka.maxRatePerPartition。 在Spark 1.5中，我们引入了一项称为背压的功能，无需设置此速率限制，因为Spark Streaming会自动计算出速率限制，并在处理条件发生变化时动态调整它们。 可以通过将配置参数spark.streaming.backpressure.enabled设置为true来启用此背压。

SparkStreaming程序更新需要注意的事项

1. 新旧程序同时运行，消费同一份数据。当新的程序消费到与旧程序数据一致时，旧的就可以停掉

## SparkStreaming与kafka两种集成方式与优缺点

sparkStreaming与kafka有两种集成方式，分别为**receiver**与**direct**模式。

**receiver模式：**采用push的方式，调用高层次的API，在实际的生产环境中，该方式使用的不多。主要是因为该方式将接收到的数据存放在executor中，然后由sparkStreaming启动的作业处理数据。但是，在默认配置下，这种方法可能会在失败时丢失数据。为了确保零数据丢失，必须在Spark Streaming中额外启用写入日志，同时保存所有接收到的Kafka数据写入分布式文件系统（例如HDFS）的预先写入日志，以便所有数据都可以在失败时恢复。

缺点：

1. kafka中的主题分区与sparkStreaming中生成的RDD的分区不相关
2. 为防止数据丢失，需要开启wal机制，降低了数据的处理效率，另外由于数据备份机制，负载一高就会出现延迟的风险，导致应用崩溃
3. 单receiver内存，由于receiver也属于executor的一部分，那么为了提高吞吐量，需要提高receiver的内存。但是在每次的batch的计算中参与计算的batch并不需要那么的内存，导致资源的浪费
4. Receiver和计算的executor是异步的，那么遇到网络等的因素，导致计算出现延迟，计算队列一直在增加而receiver一直在接收数据，这非常容易导致程序崩溃

优点：

1. Kafka的high-level数据读取方式让用户可以专注于所读数据，而不用关注或维护consumer的offsets，这减少用户的工作量以及代码量而且相对比较简单。

**Direct模式：**采用pull方式，从kafka中拉取数据，使用kafka的低级API。此法提供了更强大的端到端保证。它定期查询Kafka在每个topic+分区(partition)中的最新偏移量，而不再使用接收器去接收数据。同时，它也定义了要在每个批次中处理的不同偏移范围。特别是在那些处理数据的作业被启动时，其简单消费者(consumer)API就会被用于读取Kafka中预定义的偏移范围。可见，此过程类似于从某个文件系统中读取各种文件

优点：

1. 简化并行，不需要创建以及union多输入源，kafka的topic分区数与RDD的partition数一一对应
2. 降低资源，direct不需要receiver，其申请的executors全部用于计算。而receiver需要专门的receivers来读取数据且不参与计算。因此相同的资源申请，direct能够支持更大的业务
3. 鲁棒性更好，Receiver-based方法需要Receivers来异步持续不断的读取数据，因此遇到网络、存储负载等因素，导致实时任务出现堆积，但Receivers却还在持续读取数据，此种情况很容易导致计算崩溃。Direct 则没有这种顾虑，其Driver在触发batch 计算任务时，才会读取数据并计算。队列出现堆积并不会引起程序的失败。

缺点：

1. 提高成本，Direct需要用户采用checkpoint或者第三方存储来维护offsets，而不像receiver那样通过zookeeper维护offsets，提高了开发成本
2. 监控可视化，receiver的消费情况可以通过zookeeper来监控，而direct模式则不可以

# Hive篇

## 数据倾斜问题

**解决办法：**

1. 参数调优：set hive.map.aggr=true 在map端做部分聚合操作，效率更高，但需要更多的内存；set hive.groupby.skewindata=true;数据倾斜时负责负载均衡，设为true时会生成两个MRJob，它使计算变成了两个mapreduce，第一个在shuffle时随机给 key 打标记，使每个key 随机均匀分布到各个reduce上计算，但是这样只能完成部分计算，因为相同key没有分配到相同reduce上，所以需要第二次的mapreduce,这次就回归正常 shuffle,但是数据分布不均匀的问题在第一次mapreduce已经有了很大的改善，因此基本解决数据倾斜。
2. 调节key:在map阶段将造成数据倾斜的key先分成多组，例如aaa这个key，在map时随机在后面添加1，2，3，4这四个数字，分成四组，先进行一次计算，之后恢复key正常运算
3. 能先进行group操作的时候进行group操作，把key先进行一次reduce操作，之后再进行count
4. Join操作中，对于小表与大表join，使用map join，免得reduce时卡住。
5. 当对三个或者更多个表进行join连接时，如果每个on子句都使用相同的连接键的话，那么只会产生一个mr job

## Hive优化

1. **数据存储与压缩**。针对hive中表的存储格式通常有orc和parquet，压缩格式一般使用snappy。相比与textfile格式表，orc 占有更少的存储。因为 hive 底层使用 MR 计算架构，数据流是 hdfs 到磁盘再到hdfs，而且会有很多次，所以使用 orc 数据格式和 snappy 压缩策略可以降低 IO 读写，还能降低网络传输量，这样在一定程度上可以节省存储，还能提升 hql 任务执行效率
2. **Sql优化**。大表对大表：尽量减少数据集，可以通过分区表，避免扫描全表或者全字段大表对小表：设置自动识别小表，将小表放入内存中去执行
3. **通过参数调优**。并行执行，调节 parallel 参数；调节 jvm 参数，重用 jvm；设置 map、reduce 的参数；开启 strict mode 模式；关闭推测执行设置。

## 仓库规范

分层：

1. 把复杂问题简单化，把一个复杂问题分成多个步骤完成，每一层只完成单一步骤，比较简单。而且方便定位问题
2. 减少重复开发，规范数据分层。通过中间层，能够减少较大的重复计算，增加一次结果的复用性
3. 隔离原始数据

仓库通常分为四层

1. ods层：原始数据层，保存原始数据，不做处理
2. dwd层：结构和粒度与原始层保持一致，对ods层数据进行清洗（去除空值，脏数据，超过极限范围的数据）
3. dws层：以dwd为基础，进行轻度汇总
4. ads层：为各种统计报表提供数据

## 关于不等值连接问题

在hive中是不支持不等连接的，比如select u1.\*,u2.\* from user\_info u1 join user\_info u2 on u1.age>u2.age;是会报错，提示不支持该操作，在mysql中，上述操作是可以支持的。

## Hive与hbase的联系

**共同点：**

1. hive和hbase都是架构在hadoop之上的，都是使用hdfs作为底层存储

**不同点：**

1. hive中的表是逻辑表，仅仅对表的元数据进行定义，没有物理存储的功能。完全依赖于hdfs与mapreduce。将结构化的文件映射成一张数据库表，并提供完整的sql查询，并将sql语句最终转换为mr任务进行运行。Hbase表则是物理表，适合存放非结构化的数据。
2. hive是在mapreduce基础之上对数据进行处理，而mapreduce的数据处理按照行模式。Hbase为列模式，使得对海量数据的随机查询变得可能
3. hive全面支持sql，一般用来进行基于历史数据的挖掘，分析。Hbase不适用于有join，多极索引，表关系复杂的应用场景。
4. hbase是物理表，不是逻辑表，提供一个超大的内存hash表，搜索引擎通过它来存储索引，方便查询操作

## hive中存放的是什么

hive中存放的是和hdfs的映射关系，hive是逻辑上的数据仓库，实际上操作的都是hdfs文件。HQL就是用sql语法来写的mr程序

## UDAF与UDTF

UDAF：用户自定义聚合函数，用于输入多行，输出一行，比如sun（），avg()等。

UDTF：用户自定义表生成函数，用于输入一行，输出多行，比如explode（）

# Hbase篇

## Rowkey设计规范

Hbase表的数据是按照rowkey来分散到不同的region，不合理的rowkey设计会导致热点问题。热点问题指的是直接访问集群一个或极少数节点，而集群中其他节点处于空闲状态。

**rowkey设计加盐：**在rowkey头部随机添加一个字符，将数据分散到不同的region上，有利于写操作，但增加读开销。可以很好的支持去全表扫描

**hashing方式：**计算rowkey的hash值，然后取部分hash值与进行拼接原rowkey。可以一定程度打散整个数据集，但是不利于scan操作。对hash操作可以采用md5算法。

**rowkey反转：**如果先导字体本身会带来热点问题，但该字段尾部信息却具有良好的随机性，此时可以考虑将先导字体做反转处理，将尾部几位直接提到前面，或直接将整个字段反转

## hbase的应用场景

**对象存储：**我们知道不少的头条类、新闻类的的新闻、网页、图片存储在HBase之中，一些病毒公司的病毒库也是存储在HBase之中

**时序数据：**HBase之上有OpenTSDB模块，可以满足时序类场景的需求

**推荐画像：**特别是用户的画像，是一个比较大的稀疏矩阵，蚂蚁的风控就是构建在HBase之上

**时空数据：**主要是轨迹、气象网格之类，滴滴打车的轨迹数据主要存在HBase之中，另外在技术所有大一点的数据量的车联网企业，数据都是存在HBase之中

CubeDB OLAP：Kylin一个cube分析工具，底层的数据就是存储在HBase之中，不少客户自己基于离线计算构建cube存储在hbase之中，满足在线报表查询的需求

**消息/订单：**在电信领域、银行领域，不少的订单查询底层的存储，另外不少通信、消息同步的应用构建在HBase之上

**Feeds流：**典型的应用就是xx朋友圈类似的应用

**NewSQL：**之上有Phoenix的插件，可以满足二级索引、SQL的需求，对接传统数据需要SQL非事务的需求

## Hmaster的作用

1. 为regionServer分配region
2. 负责regionserver的负载均衡
3. Regionserver失效时，重新分配其上的region
4. Gfs文件上的垃圾文件回收
5. 处理schema更新请求

# Kafka篇

## 常见参数配置

auto.create.topics.enable：是否允许自动创建topic

unclean.leader.election.enable：是否允许 Unclean Leader 选举

auto.leader.rebalance.enable：是否允许定期进行 Leader 选举

log.retention.{hour|minutes|ms}：控制一条消息被保留的时间

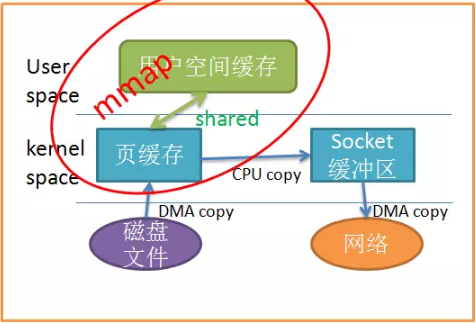
## 分区的作用

分区的作用就是提供负载均衡的能力，或者说对数据分区就是实现系统的高伸缩性。不同的分区被放置到不同的机器节点上，这样每个机器可以执行各自分区的读写请求，还可以通过增加机器数来提升系统整体的吞吐性。

## Kafka高吞吐量的原因

1. 顺序写入
2. 零拷贝：就是跳过“用户缓冲区”的拷贝，建立一个磁盘空间和内存的直接映射，数据不再复制到“用户态缓冲区”
3. 分区：kafka中的topic中的内容可以被分为多分partition存在,每个partition又分为多个段segment,所以每次操作都是针对一小部分做操作，很轻便，并且增加并行操作的能力

## 什么是零拷贝



从上图中可以看出，共产生了四次数据拷贝，即使使用了DMA来处理了与硬件的通讯，CPU仍然需要处理两次数据拷贝，与此同时，在用户态与内核态也发生了多次上下文切换，无疑也加重了CPU负担。在此过程中，我们没有对文件内容做任何修改，那么在内核空间和用户空间来回拷贝数据无疑就是一种浪费，而零拷贝主要就是为了解决这种低效性。

## 消费者组

消费者组是kafka提供的可扩展，且具有容错性的消费者机制。组下具有多个消费者实例，组内的所有消费者协调在一起共同消费订阅主题的所有分区，每个分区只能由同一个消费者组的一个实例消费。**理想情况下，消费者实例的个数应当等于组内所有主题的分区数之和，过大则会浪费资源，过小则会导致每个实例多个分区，消费速度缓慢**。在sparkstreamign消费kafka时，direct方式的消费实例数就等于所有topic的分区总数

## 谈谈kafka的重平衡问题

Rebalance本质上一种协议，规定了一个group下的所有consumer如何达成一致，来分配订阅topic的每个分区。比如某个group下有二十个消费者，订阅了一个具有100个分区的主题。正常情况下，kafka为每个消费者分配五个分区，这个过程就叫做重平衡。**当消费者个数变化，主题个数变化，主题的分区数目变化时，就会导致Rebalance触发。重新分配分区给消费者，期间所有消费者停止消费，等待Rebalance完成，代价非常高**

**解决不必要的重平衡方法（由consumer引起的）：**

1. consumer未能及时发送心跳到Coordinator，导致consumer被剔除group，可以通过调节session.timeout.ms(超过这个时间没有收到消息则认为consumer死亡，从而将consumer剔除，启动rebalance)和heartbeat.interval.ms(consumer每隔多长时间发送心跳)
2. 由于consumer消费时间过长导致rebalance，可通过调大max.poll.internal.ms(如果consumer在规定时间内没有将一次poll到的数据消费完，则将该consumer剔除)可以适当的调大

## Kafka的位移管理

在老版本的kafka中，将位移信息存放在zookeeper中，每次启动consumer时，自动从zookeeper中读取位移信息，从而在上次截止的地方继续消费。由于zookeeper更新不及时问题会导致重复消费，同时zookeeper也不适合这种高频的写操作。因此在新版本中将位移信息提交到作为一条条kafka消息\_\_consumer\_offsets（位移主题）中。消息格式为kv对，key为<groupID->主题名->分区号>，value为位移等相关信息。

## Kafka的CommitFailedExecption原因及处理方案

该异常的发生原因是在两次调用poll方法的时间间隔到之后数据还没有处理完。在该情况下提交位移时就会抛出该异常

Properties props = new Properties();

props.put("max.poll.interval.ms", 5000);

consumer.subscribe(Arrays.asList("test-topic"));

while (true) {

ConsumerRecords<String, String> records =

consumer.poll(Duration.ofSeconds(1));

// 使用 Thread.sleep 模拟真实的消息处理逻辑

Thread.sleep(6000L);

consumer.commitSync();

}

解决方法：

1. 增加期望的时间间隔，max.poll.interval.ms参数值
2. 减少poll方法一次性返回的消息数量，即减少，max.poll.records参数值
3. 缩短单条消息的处理时长

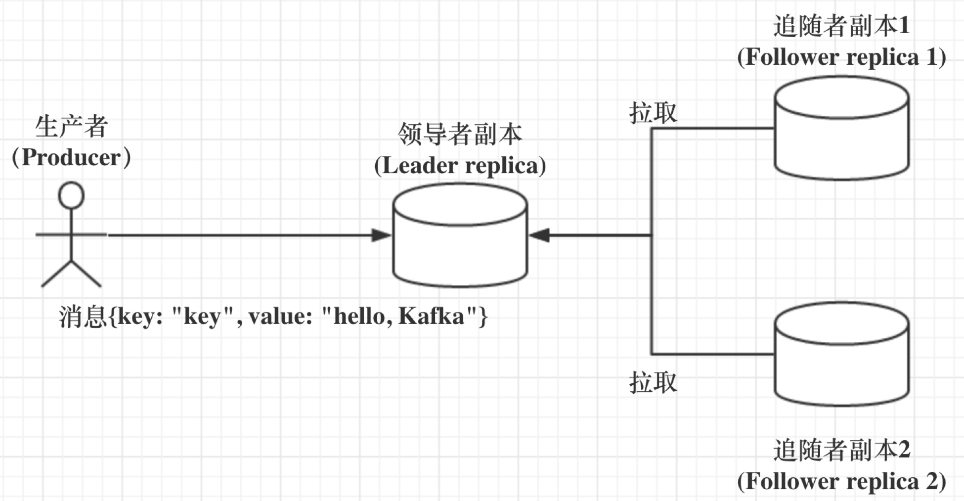
## Kafka的副本机制了解下

副本机制就是备份机制，在多台网络互联的机器上保存相同的数据拷贝。通常来说，副本机制拥有以下好处。

1. **提供数据冗余：**即使系统部分组件失效，系统依然可以继续运转
2. **提供高伸缩性：**支持横向扩展，通过增加机器的数目来提升读性能
3. **改善数据局部性：**允许将数据放入与用户地理位置相近的地方，从而降低数据延时

对于kafka来说，只能享有分区的第一个特性（副本机制）。只有领导者副本提供服务，从副本定期从主副本拉取数据进行数据同步。该设计的目的有以下两点：

1. **方便实现Read-your-writes：**往kafka写入一条消息后，马上就能通过消费者API读取到数据。如果允许允许追谁者副本对外提供服务，由于kafka采用异步同步数据，有可能追随者副本还没有拉取到最新消息，从而在客户端看不到最新写入的消息。
2. **方便实现单调读：**对于一个消费者而言，在多次消费消息时，他不会看到某条消息一会存在一会不存在。如果允许允许追谁者副本对外提供服务，比如两个副本F1，F2，先读取F1时看到最新的消息，在读取F2时，由于更新不及时，看到的数据不全，导致两次读取到的数据不一致问题



## ISR副本集合

ISR中的副本都是与leader副本同步的副本，leader副本天然在ISR副本中。

Broker端的参数replica.lag.time.max.ms控制着其它副本是否与leader副本是同步副本，若从副本落后的时间超过前面参数规定的最大时间，则表明不是同步副本，从ISR中剔除，否则就是同步副本。如果被踢除的副本后面又追上了leader的进度，那么它是能够被重新加回ISR的。当leader挂掉后，从ISR集合中选举一个副本作为leader继续对外提供服务。如果ISR集合为空，如果将ISR外的副本作为leader就是unclean选举，此时由于该副本落后leader太多，会导致数据丢失，一般不建议采用，通过参数**unclean.leader.election.enable**控制，建议设为false