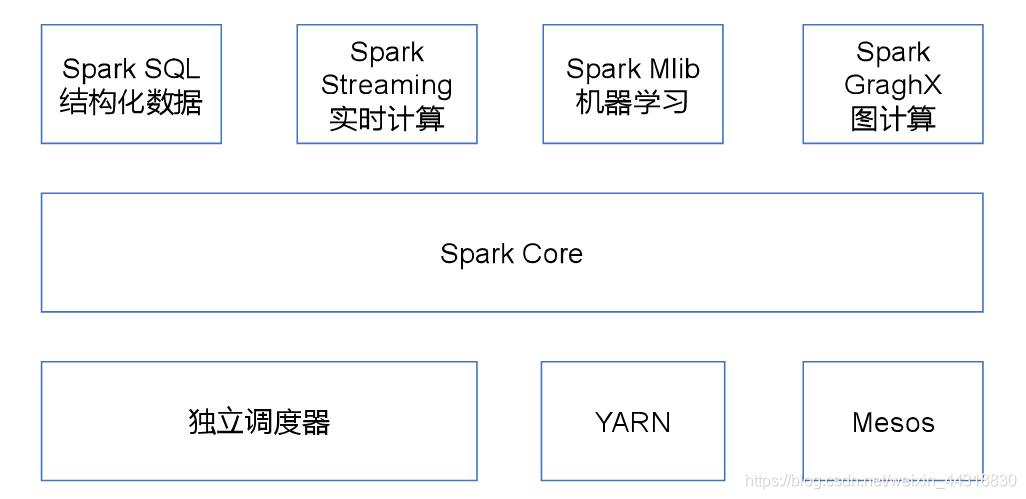
# Spark面试题

## 一、你是怎么理解Spark，它的特点是什么？

Spark是一个基于内存的，用于大规模数据处理（离线计算、实时计算、快速查询（交互式查询））的统一分析引擎。

它内部的组成模块，包含SparkCore，SparkSQL，SparkStreaming，SparkMLlib，SparkGraghx等



它的特点：

快

Spark计算速度是MapReduce计算速度的10-100倍

易用

MR支持1种计算模型，Spsark支持更多的计算模型(算法多)

通用

Spark 能够进行离线计算、交互式查询（快速查询）、实时计算、机器学习、图计算

兼容性

Spark支持大数据中的Yarn调度，支持mesos。可以处理hadoop计算的数据。

## 二、Spark有几种部署方式，请分别简要论述

1）Local:运行在一台机器上，通常是练手或者测试环境。

2）Standalone:构建一个基于Master+Slaves的资源调度集群，Spark任务提交给Master运行。是Spark自身的一个调度系统。

3）Yarn: Spark客户端直接连接Yarn，不需要额外构建Spark集群。有yarn-client和yarn-cluster两种模式，主要区别在于：Driver程序的运行节点。

4）Mesos：国内大环境比较少用。

## 三、Spark提交作业的参数

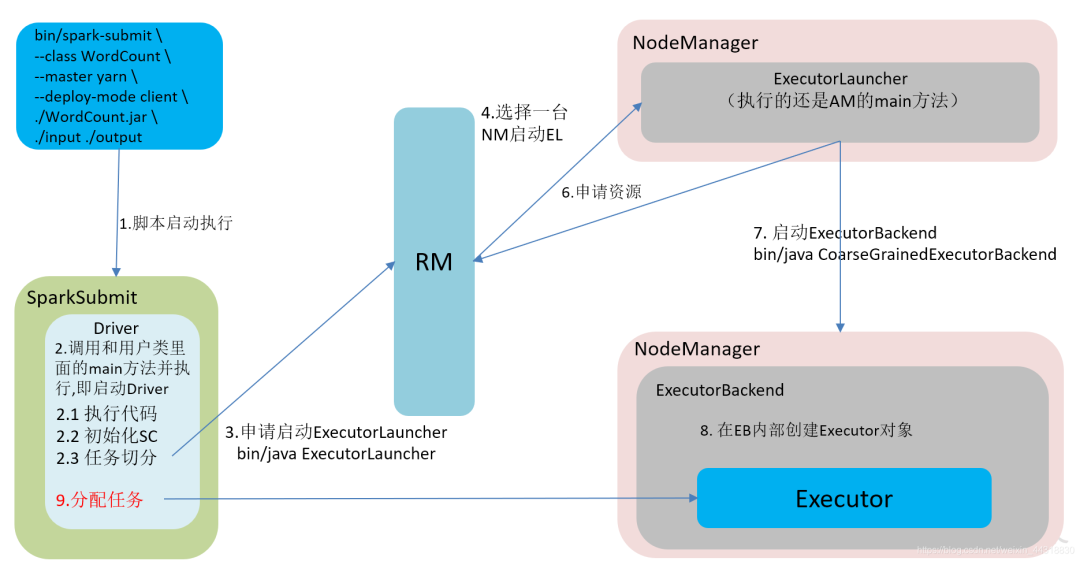
因为我们Spark任务是采用的Shell脚本进行提交，所以一定会涉及到几个重要的参数，而这个也是在面试的时候容易被考察到的“细节”。

|  |
| --- |
| executor-cores —— 每个executor使用的内核数，默认为1，官方建议2-5个，我们企业是4个  num-executors —— 启动executors的数量，默认为2  executor-memory —— executor内存大小，默认1G  driver-cores —— driver使用内核数，默认为1  driver-memory —— driver内存大小，默认512M |

## 四、简述Spark的作业提交流程

Spark的任务提交方式实际上有两种，分别是YarnClient模式和YarnCluster模式。大家在回答这个问题的时候，也需要分类去介绍。千万不要被冗长的步骤吓到，一定要学会总结差异，发现规律，通过图形去增强记忆。

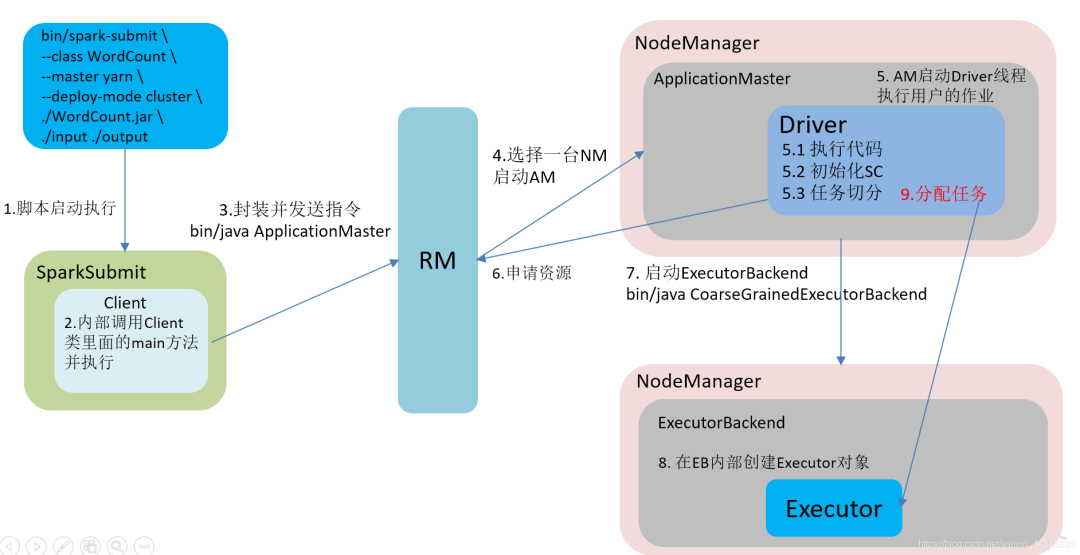
### YarnClient运行模式介绍



在YARN Client模式下，Driver在任务提交的本地机器上运行，Driver启动后会和ResourceManager通讯申请启动ApplicationMaster，随后ResourceManager分配container，在合适的NodeManager上启动ApplicationMaster，此时的ApplicationMaster的功能相当于一个ExecutorLaucher，只负责向ResourceManager申请Executor内存。

ResourceManager接到ApplicationMaster的资源申请后会分配container，然后ApplicationMaster在资源分配指定的NodeManager上启动Executor进程，Executor进程启动后会向Driver反向注册，Executor全部注册完成后Driver开始执行main函数，之后执行到Action算子时，触发一个job，并根据宽依赖开始划分stage，每个stage生成对应的taskSet，之后将task分发到各个Executor上执行。

### YarnCluster 模式介绍



在YARN Cluster模式下，任务提交后会和ResourceManager通讯申请启动ApplicationMaster，随后ResourceManager分配container，在合适的NodeManager上启动ApplicationMaster，此时的ApplicationMaster就是Driver。

Driver启动后向ResourceManager申请Executor内存，ResourceManager接到ApplicationMaster的资源申请后会分配container，然后在合适的NodeManager上启动Executor进程，Executor进程启动后会向Driver反向注册，Executor全部注册完成后Driver开始执行main函数，之后执行到Action算子时，触发一个job，并根据宽依赖开始划分stage，每个stage生成对应的taskSet，之后将task分发到各个Executor上执行。

## 五、你是如何理解Spark中血统(RDD)的概念?它的作用是什么？

### 概念：

RDD是弹性分布式数据集，是Spark中最基本的数据抽象，代表一个不可变、可分区、里面的元素可并行计算 的集合。

### 作用：

提供了一个抽象的数据模型，将具体的应用逻辑表达为一系列转换操作(函数)。另外不同RDD之间的转换操作之间还可以形成依赖关系，进而实现管道化，从而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，并且还提供了更多的API(map/reduec/filter/groupBy...)

如果还想锦上添花，可以添上这一句：

RDD在Lineage依赖方面分为两种Narrow Dependencies与Wide Dependencies，用来解决数据容错时的高效性以及划分任务时候起到重要作用

## 六、简述Spark的宽窄依赖，以及Spark如何划分stage，每个stage又根据什么决定task个数?

Spark的宽窄依赖问题是SparkCore部分的重点考察内容，多数出现在笔试中，大家需要注意。

窄依赖：父RDD的一个分区只会被子RDD的一个分区依赖

宽依赖：父RDD的一个分区会被子RDD的多个分区依赖(涉及到shuffle)

### Stage是如何划分的呢？

根据RDD之间的依赖关系的不同将Job划分成不同的Stage，遇到一个宽依赖则划分一个Stage。

### 每个stage又根据什么决定task个数?

Stage是一个TaskSet，将Stage根据分区数划分成一个个的Task。

## 七、列举Spark常用的transformation和action算子，有哪些算子会导致Shuffle？

Spark开发过程中，避不开与各种算子打交道，其中Spark 算子分为transformation 和 action 算子，下面列出一些常用的算子，具体的功能还需要小伙伴们自行去了解。

### transformation算子

map

mapRartition

flatMap

filter

### action算子

reduce

collect

first

take

有哪些会引起Shuffle过程的Spark算子呢?

reduceByKey

groupByKey

ByKey

## 八、reduceByKey与groupByKey的区别,哪一种更具优势?

reduceByKey：按照key进行聚合，在shuffle之前有combine（预聚合）操作，返回结果是RDD[k,v]。

groupByKey：按照key进行分组，直接进行shuffle。

所以，在实际开发过程中，reduceByKey比groupByKey更建议使用。但是需要注意是否会影响业务逻辑。

## 九、Repartition和Coalesce的关系与区别，能简单说说吗？

这道题就已经开始掺和有“源码”的味道了，为什么呢？

1）关系：

两者都是用来改变RDD的partition数量的，repartition底层调用的就是coalesce方法：coalesce(numPartitions, shuffle = true)

2）区别：

repartition一定会发生shuffle，coalesce 根据传入的参数来判断是否发生shuffle。

一般情况下增大rdd的partition数量使用repartition，减少partition数量时使用coalesce。

## 十、简述下Spark中的缓存(cache和persist)与checkpoint机制，并指出两者的区别和联系

关于Spark缓存和检查点的区别，大致可以从这3个角度去回答：

**位置：**

Persist 和 Cache将数据保存在内存，Checkpoint将数据保存在HDFS

**生命周期：**

Persist 和 Cache 程序结束后会被清除或手动调用unpersist方法，Checkpoint永久存储不会被删除。

**RDD依赖关系：**

Persist 和 Cache不会丢掉RDD间的依赖链/依赖关系，CheckPoint会斩断依赖链。

## 十一、简述Spark中共享变量（广播变量和累加器）的基本原理与用途

关于Spark中的广播变量和累加器的基本原理和用途，答案较为固定，大家无需刻意去记忆。

累加器（accumulator）是Spark中提供的一种分布式的变量机制，其原理类似于mapreduce，即分布式的改变，然后聚合这些改变。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。

广播变量是在每个机器上缓存一份，不可变，只读的，相同的变量，该节点每个任务都能访问，起到节省资源和优化的作用。它通常用来高效分发较大的对象。

## 十二、当Spark涉及到数据库的操作时，如何减少Spark运行中的数据库连接数？

嗯，有点“调优”的味道，感觉真正的“风暴”即将到来，这道题还是很好回答的，我们只需要减少连接数据库的次数即可。

使用foreachPartition代替foreach，在foreachPartition内获取数据库的连接。

## 十三、能介绍下你所知道和使用过的Spark调优吗?

### 资源参数调优

num-executors：设置Spark作业总共要用多少个Executor进程来执行

executor-memory：设置每个Executor进程的内存

executor-cores：设置每个Executor进程的CPU core数量

driver-memory：设置Driver进程的内存

spark.default.parallelism：设置每个stage的默认task数量

### 开发调优

* 避免创建重复的RDD
* 尽可能复用同一个RDD
* 对多次使用的RDD进行持久化
* 尽量避免使用shuffle类算子
* 使用map-side预聚合的shuffle操作
* 使用高性能的算子

|  |
| --- |
| ①使用reduceByKey/aggregateByKey替代groupByKey  ②使用mapPartitions替代普通map  ③使用foreachPartitions替代foreach  ④使用filter之后进行coalesce操作  ⑤使用repartitionAndSortWithinPartitions替代repartition与sort类操作 |

* 广播大变量

在算子函数中使用到外部变量时，默认情况下，Spark会将该变量复制多个副本，通过网络传输到task中，此时每个task都有一个变量副本。如果变量本身比较大的话（比如100M，甚至1G），那么大量的变量副本在网络中传输的性能开销，以及在各个节点的Executor中占用过多内存导致的频繁GC(垃圾回收)，都会极大地影响性能。

* 使用Kryo优化序列化性能
* 优化数据结构

在可能以及合适的情况下，使用占用内存较少的数据结构，但是前提是要保证代码的可维护性。

## 十四、如何使用Spark实现TopN的获取（描述思路或使用伪代码）？

能让你使用伪代码来描述这已经非常“苛刻”了，但是不慌，这里提供3种思路供大家参考：

**方法1：**

（1）按照key对数据进行聚合（groupByKey）；

（2）将value转换为数组，利用scala的sortBy或者sortWith进行排序（mapValues）；

注意：当数据量太大时，会导致OOM；

**方法2：**

（1）取出所有的key；

（2）对key进行迭代，每次取出一个key利用spark的排序算子进行排序；

**方法3：**

（1）自定义分区器，按照key进行分区，使不同的key进到不同的分区；

（2）对每个分区运用spark的排序算子进行排序；

## 十五、你所理解的 Spark 的 shuffle 过程？

Spark shuffle 处于一个宽依赖，可以实现类似混洗的功能，将相同的 Key 分发至同一个 Reducer上进行处理。

## 十六、Spark有哪些聚合类的算子,我们应该尽量避免什么类型的算子？

在我们的开发过程中，能避免则尽可能避免使用 reduceByKey、join、distinct、repartition 等会进行 shuffle 的算子，尽量使用 map 类的非 shuffle 算子。这样的话，没有 shuffle 操作或者仅有较少 shuffle 操作的 Spark 作业，可以大大减少性能开销。

## 十七、spark on yarn 作业执行流程，yarn-client 和 yarn cluster 有什么区别

### Spark On Yarn 的优势

1. Spark 支持资源动态共享，运行于 Yarn 的框架都共享一个集中配置好的资源池

2. 可以很方便的利用 Yarn 的资源调度特性来做分类·，隔离以及优先级控制负载，拥有更灵活的调度策略

3. Yarn 可以自由地选择 executor 数量

4. Yarn 是唯一支持 Spark 安全的集群管理器（Mesos???），使用 Yarn，Spark 可以运行于 Kerberized Hadoop 之上，在它们进程之间进行安全认证。

### yarn-client 和 yarn cluster 的异同

1. 从广义上讲，yarn-cluster 适用于生产环境。而 yarn-client 适用于交互和调试，也就是希望快速地看到 application 的输出。

2. 从深层次的含义讲，yarn-cluster 和 yarn-client 模式的区别其实就是 Application Master 进程的区别，yarn-cluster 模式下，driver 运行在 AM(Application Master)中，它负责向 YARN 申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉 Client，作业会继续在 YARN 上运行。然而 yarn-cluster 模式不适合运行交互类型的作业。而 yarn-client 模式下，Application Master 仅仅向 YARN 请求 executor，Client 会和请求的 container 通信来调度他们工作，也就是说 Client 不能离开。

## 十八、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。

结论：Spark 快不是绝对的，但是绝大多数，Spark 都比 Hadoop 计算要快。这主要得益于其对 mapreduce 操作的优化以及对 JVM 使用的优化。

## 十九、RDD, DAG, Stage怎么理解？

### DAG

Spark 中使用 DAG 对 RDD 的关系进行建模，描述了 RDD 的依赖关系，这种关系也被称之为 lineage（血缘），RDD 的依赖关系使用 Dependency 维护。DAG 在 Spark 中的对应的实现为 DAGScheduler。

### RDD

RDD 是 Spark 的灵魂，也称为弹性分布式数据集。一个 RDD 代表一个可以被分区的只读数据集。RDD 内部可以有许多分区(partitions)，每个分区又拥有大量的记录(records)。Rdd的五个特征：

1. dependencies: 建立 RDD 的依赖关系，主要 RDD 之间是宽窄依赖的关系，具有窄依赖关系的 RDD 可以在同一个 stage 中进行计算。

2. partition: 一个 RDD 会有若干个分区，分区的大小决定了对这个 RDD 计算的粒度，每个 RDD 的分区的计算都在一个单独的任务中进行。

3. preferedlocations: 按照“移动数据不如移动计算”原则，在 Spark 进行任务调度的时候，优先将任务分配到数据块存储的位置。

4. compute: Spark 中的计算都是以分区为基本单位的，compute 函数只是对迭代器进行复合，并不保存单次计算的结果。

5. partitioner: 只存在于（K,V）类型的 RDD 中，非（K,V）类型的 partitioner 的值就是 None。

RDD 的算子主要分成2类，action 和 transformation。这里的算子概念，可以理解成就是对数据集的变换。action 会触发真正的作业提交，而 transformation 算子是不会立即触发作业提交的。每一个 transformation 方法返回一个新的 RDD。只是某些 transformation 比较复杂，会包含多个子 transformation，因而会生成多个 RDD。这就是实际 RDD 个数比我们想象的多一些 的原因。通常是，当遇到 action 算子时会触发一个job的提交，然后反推回去看前面的 transformation 算子，进而形成一张有向无环图。

### Stage

在 DAG 中又进行 stage 的划分，划分的依据是依赖是否是 shuffle 的，每个 stage 又可以划分成若干 task。接下来的事情就是 driver 发送 task 到 executor，executor 自己的线程池去执行这些 task，完成之后将结果返回给 driver。action 算子是划分不同 job 的依据。

## 二十、RDD 如何通过记录更新的方式容错

RDD 的容错机制实现分布式数据集容错方法有两种:

1. 数据检查点

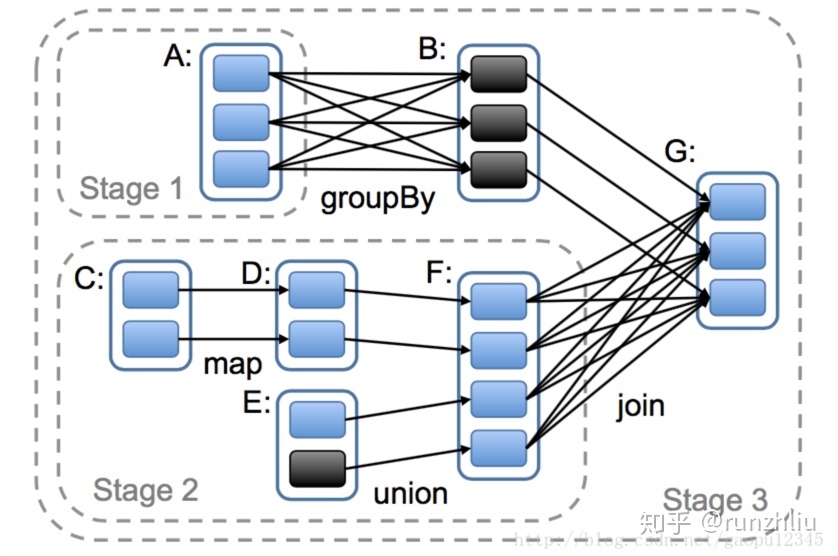
2. 记录更新。

RDD 采用记录更新的方式：记录所有更新点的成本很高。所以，RDD只支持粗颗粒变换，即只记录单个块（分区）上执行的单个操作，然后创建某个 RDD 的变换序列（血统 lineage）存储下来；变换序列指，每个 RDD 都包含了它是如何由其他 RDD 变换过来的以及如何重建某一块数据的信息。因此 RDD 的容错机制又称“血统”容错。

## 二十一、Job和Task怎么理解

**Job**：Spark 的 Job 来源于用户执行 action 操作（这是 Spark 中实际意义的 Job），就是从 RDD 中获取结果的操作，而不是将一个 RDD 转换成另一个 RDD 的 transformation 操作。

**Task**：一个 Stage 内，最终的 RDD 有多少个 partition，就会产生多少个 task。看一看图就明白了，可以数一数每个 Stage 有多少个Task。



## 二十二、Spark血统的概念

RDD 的 lineage 记录的是粗颗粒度的特定数据转换（transformation）操作（filter, map, join etc.)行为。当这个 RDD 的部分分区数据丢失时，它可以通过 lineage 获取足够的信息来重新运算和恢复丢失的数据分区。这种粗颗粒的数据模型，限制了 Spark 的运用场合，但同时相比细颗粒度的数据模型，也带来了性能的提升。

## 二十三、任务的概念

包含很多 task 的并行计算，可以认为是 Spark RDD 里面的 action，每个 action 的计算会生成一个 job。用户提交的 job 会提交给 DAGScheduler，job 会被分解成 Stage 和 Task。

## 二十四、容错方法

Spark 选择记录更新的方式。但是，如果更新粒度太细太多，那么记录更新成本也不低。因此，RDD只支持粗粒度转换，即只记录单个块上执行的单个操作，然后将创建 RDD 的一系列变换序列（每个 RDD 都包含了他是如何由其他 RDD 变换过来的以及如何重建某一块数据的信息。因此 RDD 的容错机制又称血统容错）记录下来，以便恢复丢失的分区。lineage本质上很类似于数据库中的重做日志（Redo Log），只不过这个重做日志粒度很大，是对全局数据做同样的重做进而恢复数据。

相比其他系统的细颗粒度的内存数据更新级别的备份或者 LOG 机制，RDD 的 lineage 记录的是粗颗粒度的特定数据 transformation 操作行为。当这个 RDD 的部分分区数据丢失时，它可以通过 lineage 获取足够的信息来重新运算和恢复丢失的数据分区。

## 二十五、Spark粗粒度和细粒度

如果问的是操作的粗细粒度，应该是，Spark在错误恢复的时候，只需要粗粒度的记住lineage，就可实现容错。

关于Mesos

1. 粗粒度模式（Coarse-grained Mode）: 每个应用程序的运行环境由一个 dirver 和若干个 executor 组成，其中，每个 executor 占用若干资源，内部可运行多个 task（对应多少个 slot）。应用程序的各个任务正式运行之前，需要将运行环境中的资源全部申请好，且运行过程中要一直占用这些资源，即使不用，最后程序运行结束后，回收这些资源。举个例子，比如你提交应用程序时，指定使用5个 executor 运行你的应用程序，每个 executor 占用5GB内存和5个 CPU，每个 executor 内部设置了5个 slot，则 Mesos 需要先为 executor 分配资源并启动它们，之后开始调度任务。另外，在程序运行过程中，Mesos 的 master 和 slave 并不知道 executor 内部各个 task 的运行情况，executor 直接将任务状态通过内部的通信机制汇报给 driver，从一定程度上可以认为，每个应用程序利用 Mesos 搭建了一个虚拟集群自己使用。

2. 细粒度模式（Fine-grained Mode）: 鉴于粗粒度模式会造成大量资源浪费，Spark On Mesos 还提供了另外一种调度模式：细粒度模式，这种模式类似于现在的云计算，思想是按需分配。与粗粒度模式一样，应用程序启动时，先会启动 executor，但每个 executor 占用资源仅仅是自己运行所需的资源，不需要考虑将来要运行的任务，之后，Mesos 会为每个 executor 动态分配资源，每分配一些，便可以运行一个新任务，单个 Task 运行完之后可以马上释放对应的资源。每个 Task 会汇报状态给 Mesos slave 和 Mesos Master，便于更加细粒度管理和容错，这种调度模式类似于 MapReduce 调度模式，每个 task 完全独立，优点是便于资源控制和隔离，但缺点也很明显，短作业运行延迟大。

Spark中，每个 application 对应一个 SparkContext。对于 SparkContext 之间的调度关系，取决于 Spark 的运行模式。对 Standalone 模式而言，Spark Master 节点先计算集群内的计算资源能否满足等待队列中的应用对内存和 CPU 资源的需求，如果可以，则 Master 创建 Spark Driver，启动应用的执行。宏观上来讲，这种对应用的调度类似于 FIFO 策略。在 Mesos 和 Yarn 模式下，底层的资源调度系统的调度策略都是由 Mesos 和 Yarn 决定的。具体分类描述如下：

Standalone 模式: 默认以用户提交 Applicaiton 的顺序来调度，即 FIFO 策略。每个应用执行时独占所有资源。如果有多个用户要共享集群资源，则可以使用参数 spark.cores.max 来配置应用在集群中可以使用的最大 CPU 核的数量。如果不配置，则采用默认参数 spark.deploy.defaultCore 的值来确定。

Mesos 模式: 如果在 Mesos 上运行 Spark，用户想要静态配置资源的话，可以设置 spark.mesos.coarse 为 true，这样 Mesos 变为粗粒度调度模式。然后可以设置 spark.cores.max 指定集群中可以使用的最大核数，与上面 Standalone 模式类似。同时，在 Mesos 模式下，用户还可以设置参数 spark.executor.memory 来配置每个 executor 的内存使用量。如果想使 Mesos 在细粒度模式下运行，可以通过 mesos://<url-info> 设置动态共享 CPU core 的执行模式。在这种模式下，应用不执行时的空闲 CPU 资源得以被其他用户使用，提升了 CPU 使用率。

## 二十六、Transformation和action是什么？区别？举几个常用方法

RDD 创建后就可以在 RDD 上进行数据处理。RDD 支持两种操作: 1. 转换（transformation）: 即从现有的数据集创建一个新的数据集 2. 动作（action）: 即在数据集上进行计算后，返回一个值给 Driver 程序

RDD 的转化操作是返回一个新的 RDD 的操作，比如 map() 和 filter() ，而行动操作则是向驱动器程序返回结果或把结果写入外部系统的操作，会触发实际的计算，比如 count() 和 first() 。Spark 对待转化操作和行动操作的方式很不一样，因此理解你正在进行的操作的类型是很重要的。如果对于一个特定的函数是属于转化操作还是行动操作感到困惑，你可以看看它的返回值类型：转化操作返回的是 RDD，而行动操作返回的是其他的数据类型。

RDD 中所有的 Transformation 都是惰性的，也就是说，它们并不会直接计算结果。相反的它们只是记住了这些应用到基础数据集（例如一个文件）上的转换动作。只有当发生一个要求返回结果给 Driver 的 Action 时，这些 Transformation 才会真正运行。

这个设计让 Spark 更加有效的运行。

## 二十七、Spark RDD是怎么容错的，基本原理是什么？

<https://www.cnblogs.com/duanxz/p/6329675.html>

## 二十八、为什么要用Yarn来部署Spark？

因为 Yarn 支持动态资源配置。Standalone 模式只支持简单的固定资源分配策略，每个任务固定数量的 core，各 Job 按顺序依次分配在资源，资源不够的时候就排队。这种模式比较适合单用户的情况，多用户的情境下，会有可能有些用户的任务得不到资源。

Yarn 作为通用的种子资源调度平台，除了 Spark 提供调度服务之外，还可以为其他系统提供调度，如 Hadoop MapReduce, Hive 等。

## 二十九、说说map和mapPartitions的区别

map 中的 func 作用的是 RDD 中每一个元素，而 mapPartitioons 中的 func 作用的对象是 RDD 的一整个分区。所以 func 的类型是 Iterator<T> => Iterator<T>，其中 T 是输入 RDD 的元素类型。

## 三十、groupByKey和reduceByKey是属于Transformation还是 Action？

前者，因为 Action 输出的不再是 RDD 了，也就意味着输出不是分布式的，而是回送到 Driver 程序。以上两种操作都是返回 RDD，所以应该属于 Transformation。

## 三十一、说说Worker和Excutor的异同

Worker 是指每个及节点上启动的一个进程，负责管理本节点，jps 可以看到 Worker 进程在运行。 Excutor 每个Spark 程序在每个节点上启动的一个进程，专属于一个 Spark 程序，与 Spark 程序有相同的生命周期，负责 Spark 在节点上启动的 Task，管理内存和磁盘。如果一个节点上有多个 Spark 程序，那么相应就会启动多个执行器。

## 三十二、说说Spark提供的两种共享变量

Spark 程序的大部分操作都是 RDD 操作，通过传入函数给 RDD 操作函数来计算，这些函数在不同的节点上并发执行，内部的变量有不同的作用域，不能相互访问，有些情况下不太方便。

广播变量，是一个只读对象，在所有节点上都有一份缓存，创建方法是 SparkContext.broadcast()。创建之后再更新它的值是没有意义的，一般用 val 来修改定义。

计数器，只能增加，可以用计数或求和，支持自定义类型。创建方法是 SparkContext.accumulator(V, name)。只有 Driver 程序可以读这个计算器的变量，RDD 操作中读取计数器变量是无意义的。但节点可以对该计算器进行增加。

以上两种类型都是 Spark 的共享变量。

## 三十三、说说Spark的预写日志功能

也叫 WriteAheadLogs，通常被用于数据库和文件系统中，保证数据操作的持久性。预写日志通常是先将操作写入到一个持久可靠的日志文件中，然后才对数据施加该操作，当加入施加该操作中出现异常，可以通过读取日志文件并重新施加该操作，从而恢复系统。

当 WAL 开启后，所有收到的数据同时保存到了容错文件系统的日志文件中，当 Spark Streaming 失败，这些接受到的数据也不会丢失。另外，接收数据的正确性只在数据被预写到日志以后接收器才会确认。已经缓存但还没有保存的数据可以在 Driver 重新启动之后由数据源再发送一次（经常问）。

这两个机制保证了数据的零丢失，即所有的数据要么从日志中恢复，要么由数据源重发。

## 三十四、Spark Streaming小文件问题

使用 Spark Streaming 时，如果实时计算结果要写入到 HDFS，那么不可避免的会遇到一个问题，那就是在默认情况下会产生非常多的小文件，这是由 Spark Streaming 的微批处理模式和 DStream(RDD) 的分布式(partition)特性导致的，Spark Streaming 为每个 Partition 启动一个独立的线程（一个 task/partition 一个线程）来处理数据，一旦文件输出到 HDFS，那么这个文件流就关闭了，再来一个 batch 的 parttition 任务，就再使用一个新的文件流，那么假设，一个 batch 为10s，每个输出的 DStream 有32（最大？？？）个 partition，那么一个小时产生的文件数将会达到(3600/10)\*32=11520个之多。众多小文件带来的结果是有大量的文件元信息，比如文件的 location、文件大小、block number 等需要 NameNode 来维护，NameNode 会因此鸭梨山大。不管是什么格式的文件，parquet、text、JSON 或者 Avro，都会遇到这种小文件问题，这里讨论几种处理 Spark Streaming 小文件的典型方法。

增加 batch 大小: 这种方法很容易理解，batch 越大，从外部接收的 event 就越多，内存积累的数据也就越多，那么输出的文件数也就会变少，比如上边的时间从10s增加为100s，那么一个小时的文件数量就会减少到1152个。但别高兴太早，实时业务能等那么久吗，本来人家10s看到结果更新一次，现在要等快两分钟，是人都会骂娘。所以这种方法适用的场景是消息实时到达，但不想挤压在一起处理，因为挤压在一起处理的话，批处理任务在干等，这时就可以采用这种方法。

Coalesce大法好: 文章开头讲了，小文件的基数是 batch\_number \* partition\_number，而第一种方法是减少 batch\_number，那么这种方法就是减少 partition\_number 了，这个 api 不细说，就是减少初始的分区个数。看过 spark 源码的童鞋都知道，对于窄依赖，一个子 RDD 的 partition 规则继承父 RDD，对于宽依赖(就是那些个叉叉叉ByKey操作)，如果没有特殊指定分区个数，也继承自父 rdd。那么初始的 SourceDstream 是几个 partiion，最终的输出就是几个 partition。所以 Coalesce 大法的好处就是，可以在最终要输出的时候，来减少一把 partition 个数。但是这个方法的缺点也很明显，本来是32个线程在写256M数据，现在可能变成了4个线程在写256M数据，而没有写完成这256M数据，这个 batch 是不算结束的。那么一个 batch 的处理时延必定增长，batch 挤压会逐渐增大。

Spark Streaming 外部来处理: 我们既然把数据输出到 hdfs，那么说明肯定是要用 Hive 或者 Spark Sql 这样的“sql on hadoop”系统类进一步进行数据分析，而这些表一般都是按照半小时或者一小时、一天，这样来分区的(注意不要和 Spark Streaming 的分区混淆，这里的分区，是用来做分区裁剪优化的)，那么我们可以考虑在 Spark Streaming 外再启动定时的批处理任务来合并 Spark Streaming 产生的小文件。这种方法不是很直接，但是却比较有用，“性价比”较高，唯一要注意的是，批处理的合并任务在时间切割上要把握好，搞不好就可能会去合并一个还在写入的 Spark Streaming 小文件。

自己调用 foreach 去 append: Spark Streaming 提供的 foreach 这个 outout 类 api （一种 Action 操作），可以让我们自定义输出计算结果的方法。那么我们其实也可以利用这个特性，那就是每个 batch 在要写文件时，并不是去生成一个新的文件流，而是把之前的文件打开。考虑这种方法的可行性，首先，HDFS 上的文件不支持修改，但是很多都支持追加，那么每个 batch 的每个 partition 就对应一个输出文件，每次都去追加这个 partition 对应的输出文件，这样也可以实现减少文件数量的目的。这种方法要注意的就是不能无限制的追加，当判断一个文件已经达到某一个阈值时，就要产生一个新的文件进行追加了。所以大概就是一直32个文件咯。

## 三十五、driver的功能是什么？

1）一个Spark作业运行时包括一个Driver进程，也是作业的主进程，具有main函数，并且有SparkContext的实例，是程序的入口点；

2）功能：负责向集群申请资源，向master注册信息，负责了作业的调度，负责作业的解析、生成Stage并调度Task到Executor上。包括DAGScheduler，TaskScheduler。

## 三十六、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

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。

## 三十七、cache和pesist的区别

1）cache和persist都是用于将一个RDD进行缓存的，这样在之后使用的过程中就不需要重新计算了，可以大大节省程序运行时间；

2） cache只有一个默认的缓存级别MEMORY\_ONLY ，cache调用了persist，而persist可以根据情况设置其它的缓存级别；

3）executor执行的时候，默认60%做cache，40%做task操作，persist最根本的函数，最底层的函数。

## 三十八、常规的容错方式有哪几种类型？RDD通过Linage（记录数据更新）的方式为何很高效？

　　1）数据检查点，会发生拷贝，浪费资源。

　　2）记录数据的更新，每次更新都会记录下来，比较复杂且比较消耗性能。

**为何高效：**

　　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。

## 三十九、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。

## 四十、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。

## 四十一、Spark的数据本地性有哪几种？

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。

## 四十二、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种操作；

## 四十三、Spark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？

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；

## 四十四、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?

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。

## 四十五、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。

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

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)。

## 四十七、Kafka 分布式的情况下，如何保证消息的顺序？

Kafka 分布式的单位是 Partition。如何保证消息有序，需要分几个情况讨论。

* 同一个 Partition 用一个 write ahead log 组织，所以可以保证 FIFO 的顺序。
* 不同 Partition 之间不能保证顺序。但是绝大多数用户都可以通过 message key 来定义，因为同一个 key 的 message 可以保证只发送到同一个 Partition。比如说 key 是 user id，table row id 等等，所以同一个 user 或者同一个 record 的消息永远只会发送到同一个 Partition上，保证了同一个 user 或 record 的顺序。
* 当然，如果你有 key skewness 就有些麻烦，需要特殊处理。

## 四十八、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？

因为程序在运行之前，已经申请过资源了，driver和Executors通讯，不需要和master进行通讯的。

## 四十九、Spark中Work的主要工作是什么？

主要功能：管理当前节点内存，CPU的使用状况，接收master分配过来的资源指令，通过ExecutorRunner启动程序分配任务，worker就类似于包工头，管理分配新进程，做计算的服务，相当于process服务。需要注意的是：1）worker会不会汇报当前信息给master，worker心跳给master主要只有workid，它不会发送资源信息以心跳的方式给mater，master分配的时候就知道work，只有出现故障的时候才会发送资源。2）worker不会运行代码，具体运行的是Executor是可以运行具体appliaction写的业务逻辑代码，操作代码的节点，它不会运行程序的代码的。

## 五十、Mapreduce和Spark的都是并行计算，那么他们有什么相同和区别

答：两者都是用mr模型来进行并行计算:

1）hadoop的一个作业称为job，job里面分为map task和reduce task，每个task都是在自己的进程中运行的，当task结束时，进程也会结束。

2）spark用户提交的任务成为application，一个application对应一个sparkcontext，app中存在多个job，每触发一次action操作就会产生一个job。这些job可以并行或串行执行，每个job中有多个stage，stage是shuffle过程中DAGSchaduler通过RDD之间的依赖关系划分job而来的，每个stage里面有多个task，组成taskset有TaskSchaduler分发到各个executor中执行，executor的生命周期是和app一样的，即使没有job运行也是存在的，所以task可以快速启动读取内存进行计算。

3）hadoop的job只有map和reduce操作，表达能力比较欠缺而且在mr过程中会重复的读写hdfs，造成大量的io操作，多个job需要自己管理关系。

spark的迭代计算都是在内存中进行的，API中提供了大量的RDD操作如join，groupby等，而且通过DAG图可以实现良好的容错。

## 五十一、Spark技术栈有哪些组件，每个组件都有什么功能，适合什么应用场景？

1）**Spark core**：是其它组件的基础，spark的内核，主要包含：有向循环图、RDD、Lingage、Cache、broadcast等，并封装了底层通讯框架，是Spark的基础。

2）**SparkStreaming**：是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如Kdfka、Flume、Twitter、Zero和TCP 套接字）进行类似Map、Reduce和Join等复杂操作，将流式计算分解成一系列短小的批处理作业。

3）**Spark sql**：Shark是SparkSQL的前身，Spark SQL的一个重要特点是其能够统一处理关系表和RDD，使得开发人员可以轻松地使用SQL命令进行外部查询，同时进行更复杂的数据分析

4）**BlinkDB** ：是一个用于在海量数据上运行交互式 SQL 查询的大规模并行查询引擎，它允许用户通过权衡数据精度来提升查询响应时间，其数据的精度被控制在允许的误差范围内。

5）**MLBase：**是Spark生态圈的一部分专注于机器学习，让机器学习的门槛更低，让一些可能并不了解机器学习的用户也能方便地使用MLbase。MLBase分为四部分：MLlib，MLI、ML Optimizer和MLRuntime。

6）**GraphX：**是Spark中用于图和图并行计算；

## 五十二、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。

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种方式的优缺点：

**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。

## 五十三、spark的master和worker通过什么方式进行通信的？

Akka方式

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！

## 五十四、.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；

## 五十五、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；

## 五十六、Spark并行度怎么设置比较合适

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。

## 五十七、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。

## 五十八、collect功能是什么，其底层是怎么实现的？

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。

## 五十九、map与flatMap的区别

map：对RDD每个元素转换，文件中的每一行数据返回一个数组对象；

flatMap：对RDD每个元素转换，然后再扁平化，将所有的对象合并为一个对象，文件中的所有行数据仅返回一个数组对象，会抛弃值为null的值；

## 六十、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的场景；

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。

## 六十一、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。

## 六十二、spark支持处理那些文件格式的读取？

文本、json、csv、hadoop sequenceFiles、Protocal buffers...

## 六十三、spark如何将一个目录下的所有文件读到一个pair rdd中？

sc.wholetextFiles("file://root/home/dir")

## 六十四、说说SparkContext和SparkSession有什么区别

Application：用户编写的 Spark 应用程序，Driver 即运行上述 Application 的 main() 函数并且创建 SparkContext。Application 也叫应用。

SparkContext：整个应用的上下文，控制应用的生命周期。

RDD：不可变的数据集合，可由 SparkContext 创建，是 Spark 的基本计算单元。

SparkSession：Application、SparkSession、SparkContext、RDD之间具有包含关系，并且前三者是1对1的关系。

## 六十五、说说RDD和DataFrame和DataSet的关系

### 共性：

1、RDD、DataFrame、Dataset全都是spark平台下的分布式弹性数据集，为处理超大型数据提供便利；

2、三者都有惰性机制，在进行创建、转换，如map方法时，不会立即执行，只有在遇到Action如foreach时，三者才会开始遍历运算，极端情况下，如果代码里面有创建、转换，但是后面没有在Action中使用对应的结果，在执行时会被直接跳过；

3、三者都会根据spark的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存溢出；

4、三者都有partition的概念，这样对每一个分区进行操作时，就跟在操作数组一样，不但数据量比较小，而且可以方便的将map中的运算结果拿出来，如果直接用map，map中对外面的操作是无效的；

5、三者有许多共同的函数，如filter，排序等；

6、在对DataFrame和Dataset进行操作许多操作都需要这个包（import spark.implicits.\_）进行支持；

7、DataFrame和Dataset均可使用模式匹配获取各个字段的值和类型；

### 区别：

* RDD：

1、RDD一般和spark mlib同时使用；

2、RDD不支持sparksql操作；

3、RDD编译时类型安全：编译时能检查出类型错误；

4、面向对象的编程风格：直接通过类名点的方式操作数据。

5、序列化和反序列化的性能开销很大，大量的网络传输；

6、构建对象占用了大量的heap堆内存，导致频繁的GC（程序进行GC时，所有任务都是暂停）

* DataFrame：

1、与RDD和Dataset不同，DataFrame每一行的类型固定为Row，只有通过解析才能获取各个字段的值，如

每一列的值没法直接访问；

2、DataFrame与Dataset一般与spark ml同时使用；

3、DataFrame与Dataset均支持sparksql的操作，比如select，groupby之类，还能注册临时表/视窗，进行sql语句操作；

4、DataFrame与Dataset支持一些特别方便的保存方式，比如保存成csv，可以带上表头，这样每一列的字段名一目了然，利用这样的保存方式，可以方便的获得字段名和列的对应，而且分隔符（delimiter）可以自由指定；

5、DataFrame带有元数据schema，每一列都带有名称和类型。

6、DataFrame引入了off-heap，构建对象直接使用操作系统的内存，不会导致频繁GC。

7、DataFrame可以从很多数据源构建；

8、DataFrame把内部元素看成Row对象，表示一行行的数据

9、DataFrame=RDD+schema

10、编译时类型不安全，不具有面向对象编程的风格

* Dataset：

这里主要对比Dataset和DataFrame，因为Dataset和DataFrame拥有完全相同的成员函数，区别只是每一行的数据类型不同。DataSet可以在编译时检查类型，并且是面向对象的编程接口。

DataFrame也可以叫Dataset[Row],每一行的类型是Row，不解析，每一行究竟有哪些字段，各个字段又是什么类型都无从得知，只能用上面提到的getAS方法或者共性中的第七条提到的模式匹配拿出特定字段。

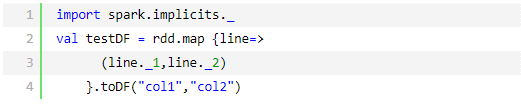
而Dataset中，每一行是什么类型是不一定的，在自定义了case class之后可以很自由的获得每一行的信息， Dataset在需要访问列中的某个字段时是非常方便的，然而，如果要写一些适配性很强的函数时，如果使用Dataset，行的类型又不确定，可能是各种case class，无法实现适配，这时候用DataFrame即Dataset[Row]就能比较好的解决问题。

### 三者之间的转化：

DataFrame/Dataset转RDD：

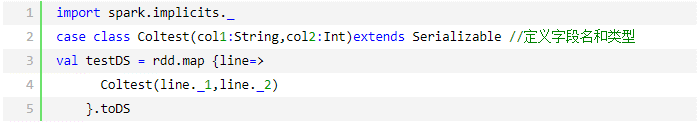


RDD转DataFrame：



一般用元组把一行的数据写在一起，然后在toDF中指定字段名

RDD转Dataset：



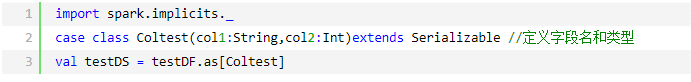
可以注意到，定义每一行的类型（case class）时，已经给出了字段名和类型，后面只要往case class里面添加值即可

Dataset转DataFrame：

这个也很简单，因为只是把case class封装成Row



DataFrame转Dataset：



## 六十六、spark on yarn Cluster 模式下，ApplicationMaster和driver是在同一个进程么？

是，driver 位于ApplicationMaster进程中。该进程负责申请资源，还负责监控程序、资源的动态情况。

## 六十七、spark中的RDD是什么，有哪些特性？

RDD（Resilient Distributed Dataset）叫做分布式数据集，是spark中最基本的数据抽象，它代表一个不可变，可分区，里面的元素可以并行计算的集合。

**五大特性：**

A list of partitions：一个分区列表，RDD中的数据都存储在一个分区列表中

A function for computing each split：作用在每一个分区中的函数

A list of dependencies on other RDDs：一个RDD依赖于其他多个RDD，这个点很重要，RDD的容错机制就是依据这个特性而来的

Optionally,a Partitioner for key-value RDDs(eg:to say that the RDD is hash-partitioned)：可选的，针对于kv类型的RDD才有这个特性，作用是决定了数据的来源以及数据处理后的去向

可选项，数据本地性，数据位置最优

## 六十八、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)

map过程产生大量对象导致内存溢出：

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。

### 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。

### 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：

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。

## 六十九、任务划分的几个重要角色

RDD任务切分中间分为：Application、Job、Stage和Task

1）Application：初始化一个SparkContext即生成一个Application；

2）Job：一个Action算子就会生成一个Job；

3）Stage：根据RDD之间的依赖关系的不同将Job划分成不同的Stage，遇到一个宽依赖则划分一个Stage；

4）Task：Stage是一个TaskSet，将Stage划分的结果发送到不同的Executor执行即为一个Task；

## 七十、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。

## 七十一、Spark如何处理非结构化数据？

通过Scala的函数式编程进行基于RDD的非结构化数据处理。

## 七十二、什么是Spark的推测执行？

推测执行

在Spark中任务会以DAG图的方式并行执行，每个节点都会并行的运行在不同的executor中，但是有的任务可能执行很快，有的任务执行很慢，比如网络抖动、性能不同、数据倾斜等等。有的Task很慢就会成为整个任务的瓶颈，此时可以触发 推测执行 (speculative) 功能，为长时间的task重新启动一个task，哪个先完成就使用哪个的结果，并Kill掉另一个task。

## 七十三、使用Spark时，如何最大限度地减少数据传输？

在Spark中，可以通过避免导致数据混洗的操作来减少数据传输。

避免重新分区和合并，ByBey操作（如groupByKey和reduceByKey）以及联合操作（如cogroup和join）之类的操作。

Spark Shared Variables有助于减少数据传输。共享变量有两种类型 - 广播变量和累加器。

**广播变量**：

如果我们有一个大型数据集，而不是为每个任务传输数据集的副本，我们可以使用一个广播变量，该变量可以一次复制到每个节点，

并为该节点中的每个任务共享相同的数据。广播变量有助于为每个节点提供大型数据集。

首先，我们需要使用SparkContext.broadcast创建一个广播变量，然后将其广播到驱动程序的所有节点。值方法

可用于访问共享值。仅当多个阶段的任务使用相同数据时，才会使用广播变量。

**累加器**：

Spark函数使用驱动程序中定义的变量，并生成变量的本地复制。累加器是共享变量，有助于

在执行期间并行更新变量，并将工作者的结果共享给驱动程序。

## 七十四、如何在Apache Spark中实现容错？

Apache Spark中容错的基本语义是，所有Spark RDD都是不可变的。它通过在DAG中创建的沿袭图表记住操作中涉及的每个RDD之间的依赖关系，并且在任何失败的情况下，Spark指的是应用相同操作来执行任务的沿袭图。有两种类型的故障 - 工作者或驱动程序故障。如果工作程序失败，该工作节点中的执行程序将与其内存中的数据一起被终止。使用沿袭图，这些任务将在任何其他工作节点中完成。数据也会复制到其他工作节点以实现容错。有两种情况：

接收和复制的数据 - 从源接收数据，并在工作节点之间复制数据。在任何故障的情况下，数据复制将有助于实现容错。

2.已接收但尚未复制的数据 - 从源接收数据但缓冲以进行复制。如果发生任何故障，则需要从源检索数据。

对于基于接收器的流输入，容错基于接收器的类型：

可靠的接收器 - 一旦接收和复制数据，就会向源发送确认。如果接收器发生故障，源将不会收到对接收数据的确认。当接收器重新启动时，源将重新发送数据以实现容错。

不可靠的接收器 - 接收的数据将不会被确认。在这种情况下，如果发生任何故障，源将不知道数据是否已被接收，并且它将不会重新发送数据，因此会丢失数据。

为了克服这种数据丢失情况，Apache Spark 1.2中引入了写入前向记录（WAL）。启用WAL后，首先在日志文件中记下操作的意图，这样如果驱动程序失败并重新启动，则该日志文件中的注释操作可以应用于数据。对于读取流数据的源，如Kafka或Flume，接收器将接收数据，这些数据将存储在执行程序的内存中。启用WAL后，这些接收的数据也将存储在日志文件中。

可以通过执行以下操作启用WAL：

使用streamingContext.checkpoint（path）设置检查点目录

通过将spark.stream.receiver.WriteAheadLog.enable设置为True来启用WAL日志记录

## 七十五、Spark的Driver运行在worker上和运行在客户端上的区别？

### Driver运行在Worker上

通过org.apache.spark.deploy.Client类执行作业，作业运行命令如下：

作业执行流程描述：

1、客户端提交作业给Master

2、Master让一个Worker启动Driver，即SchedulerBackend。Worker创建一个DriverRunner线程，DriverRunner启动SchedulerBackend进程。

3、另外Master还会让其余Worker启动Exeuctor，即ExecutorBackend。Worker创建一个ExecutorRunner线程，ExecutorRunner会启动ExecutorBackend进程。

4、ExecutorBackend启动后会向Driver的SchedulerBackend注册。SchedulerBackend进程中包含DAGScheduler，它会根据用户程序，生成执行计划，并调度执行。对于每个stage的task，都会被存放到TaskScheduler中，ExecutorBackend向SchedulerBackend汇报的时候把TaskScheduler中的task调度到ExecutorBackend执行。

5、所有stage都完成后作业结束。

### Driver运行在客户端

作业执行流程描述：

1、客户端启动后直接运行用户程序，启动Driver相关的工作：DAGScheduler和BlockManagerMaster等。

2、客户端的Driver向Master注册。

3、Master还会让Worker启动Exeuctor。Worker创建一个ExecutorRunner线程，ExecutorRunner会启动ExecutorBackend进程。

4、ExecutorBackend启动后会向Driver的SchedulerBackend注册。Driver的DAGScheduler解析作业并生成相应的Stage，每个Stage包含的Task通过TaskScheduler分配给Executor执行。

5、所有stage都完成后作业结束。

## 七十六、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

## 七十七、reduceByKey是不是action算子？

答：不是，很多人都会以为是action，reduce rdd是action

## 七十八、对于Spark中的数据倾斜问题你有什么好的方案？

简单一句: Spark 数据倾斜的几种场景以及对应的解决方案，包括避免数据源倾斜，调整并行度，使用自定义 Partitioner，使用 Map 侧 Join 代替 Reduce 侧 Join（内存表合并），给倾斜 Key 加上随机前缀等。

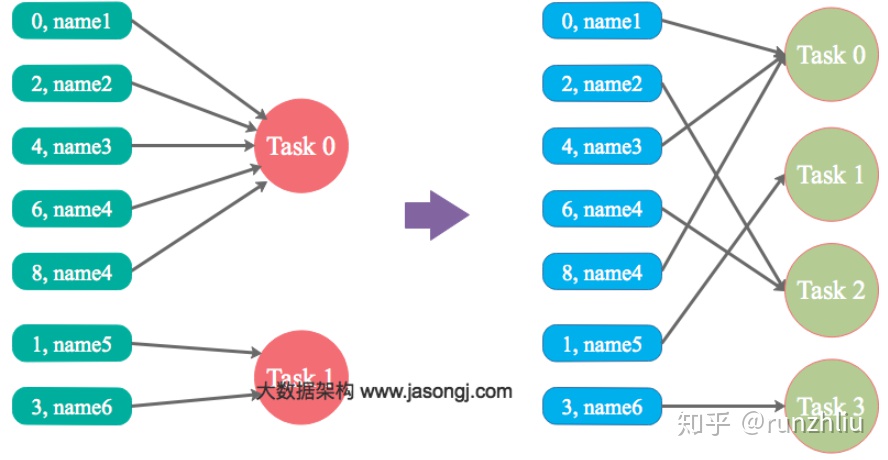
什么是数据倾斜 对 Spark/Hadoop 这样的大数据系统来讲，数据量大并不可怕，可怕的是数据倾斜。数据倾斜指的是，并行处理的数据集中，某一部分（如 Spark 或 Kafka 的一个 Partition）的数据显著多于其它部分，从而使得该部分的处理速度成为整个数据集处理的瓶颈（木桶效应）。

数据倾斜是如何造成的 在 Spark 中，同一个 Stage 的不同 Partition 可以并行处理，而具有依赖关系的不同 Stage 之间是串行处理的。假设某个 Spark Job 分为 Stage 0和 Stage 1两个 Stage，且 Stage 1依赖于 Stage 0，那 Stage 0完全处理结束之前不会处理Stage 1。而 Stage 0可能包含 N 个 Task，这 N 个 Task 可以并行进行。如果其中 N-1个 Task 都在10秒内完成，而另外一个 Task 却耗时1分钟，那该 Stage 的总时间至少为1分钟。换句话说，一个 Stage 所耗费的时间，主要由最慢的那个 Task 决定。由于同一个 Stage 内的所有 Task 执行相同的计算，在排除不同计算节点计算能力差异的前提下，不同 Task 之间耗时的差异主要由该 Task 所处理的数据量决定。

具体解决方案：

**1. 调整并行度分散同一个Task的不同Key：**

Spark在做Shuffle时，默认使用HashPartitioner（非 Hash Shuffle ???）对数据进行分区。如果并行度设置的不合适，可能造成大量不相同的 Key 对应的数据被分配到了同一个Task上，造成该Task所处理的数据远大于其它Task，从而造成数据倾斜。如果调整Shuffle时的并行度，使得原本被分配到同一Task的不同Key发配到不同Task上处理，则可降低原Task所需处理的数据量，从而缓解数据倾斜问题造成的短板效应。图中左边绿色框表示 kv样式的数据，key可以理解成name。可以看到Task0分配了许多的key，调整并行度，多了几个Task，那么每个Task处理的数据量就分散了。

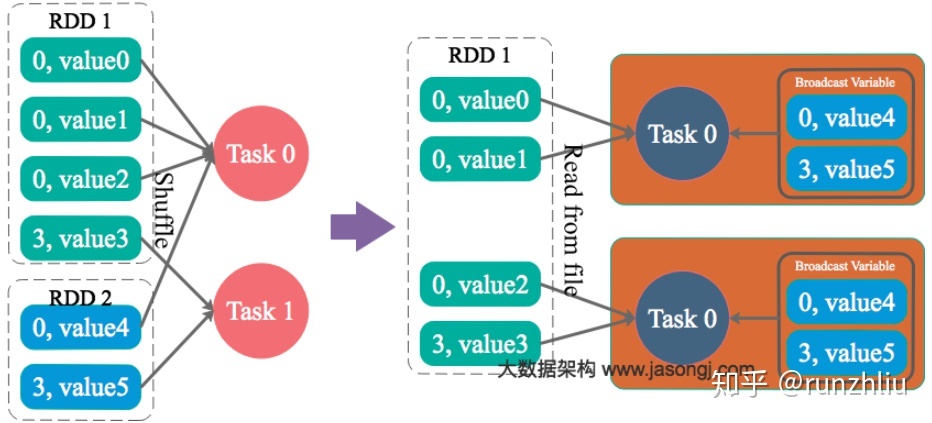


**2. 自定义Partitioner：**

使用自定义的Partitioner（默认为 HashPartitioner），将原本被分配到同一个Task的不同Key分配到不同Task，可以拿上图继续想象一下，通过自定义Partitioner可以把原本分到Task0的Key分到Task1，那么Task0的要处理的数据量就少了。

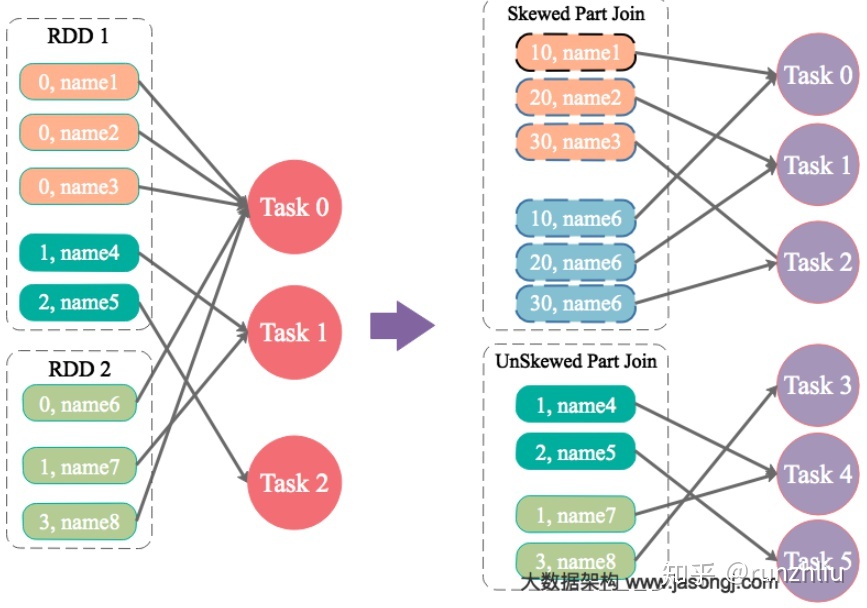
**3. 将Reduce side（侧）Join转变为Map side（侧）Join：**

通过Spark的Broadcast机制，将Reduce侧Join转化为Map侧Join，避免Shuffle从而完全消除Shuffle带来的数据倾斜。可以看到RDD2被加载到内存中了。



**4. 为skew的key增加随机前/后缀：**

为数据量特别大的Key增加随机前/后缀，使得原来Key相同的数据变为Key不相同的数据，从而使倾斜的数据集分散到不同的Task中，彻底解决数据倾斜问题。Join另一则的数据中，与倾斜Key对应的部分数据，与随机前缀集作笛卡尔乘积，从而保证无论数据倾斜侧倾斜Key如何加前缀，都能与之正常Join。



**5. 大表随机添加N种随机前缀，小表扩大N倍：**

如果出现数据倾斜的Key比较多，上一种方法将这些大量的倾斜Key分拆出来，意义不大（很难一个Key一个Key 都加上后缀）。此时更适合直接对存在数据倾斜的数据集全部加上随机前缀，然后对另外一个不存在严重数据倾斜的数据集整体与随机前缀集作笛卡尔乘积（即将数据量扩大 N 倍），可以看到RDD2扩大了N倍了，再和加完前缀的大数据做笛卡尔积。

