[第6章 spark 277](#_Toc515544358)

[6.1 什么是Spark 277](#_Toc515544359)

[6.1.1 Spark的特点及相对于MapReduce的优势 277](#_Toc515544360)

[6.1.2 Spark HA高可用部署 279](#_Toc515544361)

[6.1.3 Spark的角色 280](#_Toc515544362)

[6.1.4 Spark程序提交任务模式 281](#_Toc515544363)

[6.1.5 Spark-Shell 282](#_Toc515544364)

[6.1.6 在IDEA中编写WordCount程序 283](#_Toc515544365)

[6.2 Spark Streaming 283](#_Toc515544366)

[6.2.1 Spark Streming的特性 283](#_Toc515544367)

[6.2.2 Spark Streaming 对比 Storm 283](#_Toc515544368)

[6.3 算子操作 283](#_Toc515544369)

[6.3.1 Transformations 283](#_Toc515544370)

[6.3.2 OutputOperations 285](#_Toc515544371)

[6.3.3 Spark Streming编程实战 286](#_Toc515544372)

[6.3.4 SparkStreaming整合flume 286](#_Toc515544373)

[6.3.5 SparkStreaming整合Kafka 286](#_Toc515544374)

[6.3.6 Checkpoint 287](#_Toc515544375)

[6.4 Spark SQL 288](#_Toc515544376)

[6.4.1 几个知识点 288](#_Toc515544377)

[6.4.2 002 以编程方式执行Spark SQL查询 289](#_Toc515544378)

[6.4.3 JDBC数据源 291](#_Toc515544379)

[6.5 SparkRDD 291](#_Toc515544380)

[6.5.1 了解SparkRDD 291](#_Toc515544381)

[1）什么是RDD 291](#_Toc515544382)

[2）RDD的属性 292](#_Toc515544383)

[3.为什么会产生RDD？ 293](#_Toc515544384)

[4） RDD在Spark中的地位及作用 293](#_Toc515544385)

[6.5.2 创建RDD 294](#_Toc515544386)

[6.5.3 RDD编程API 295](#_Toc515544387)

[6.5.4 RDD常用的算子操作 297](#_Toc515544388)

[6.5.5 RDD的依赖关系 300](#_Toc515544389)

[6.5.6 RDD的缓存 301](#_Toc515544390)

[6.5.7 DAG的生成 302](#_Toc515544391)

[6.5.8 Spark任务调度 303](#_Toc515544392)

[6.6 RDD容错机制之checkpoint 305](#_Toc515544393)

[6.6.1 checkpoint是什么 305](#_Toc515544394)

[6.6.2 checkpoint原理机制 305](#_Toc515544395)

[6.6.3 Checkpoint常见面试问题 312](#_Toc515544396)

[6.7 Spark运行架构 316](#_Toc515544397)

[6.7.1 Spark运行基本流程 316](#_Toc515544398)

[6.7.2 Spark运行架构特点 317](#_Toc515544399)

[6.8 开发调优 323](#_Toc515544400)

[6.8.1 调优概述 323](#_Toc515544401)

[6.8.2 原则一：避免创建重复的RDD 323](#_Toc515544402)

[6.8.3 原则二：尽可能复用同一个RDD 324](#_Toc515544403)

[6.8.4 原则三：对多次使用的RDD进行持久化 325](#_Toc515544404)

[6.8.5 原则四：尽量避免使用shuffle类算子 327](#_Toc515544405)

[6.8.6 原则五：使用map-side预聚合的shuffle操作 328](#_Toc515544406)

[6.8.7 原则六：使用高性能的算子 330](#_Toc515544407)

[6.8.8 原则七：广播大变量 331](#_Toc515544408)

[6.8.9 原则八：使用Kryo优化序列化性能 332](#_Toc515544409)

[6.8.10 原则九：优化数据结构 332](#_Toc515544410)

[6.9 资源调优 333](#_Toc515544411)

[6.9.1 调优概述 333](#_Toc515544412)

[6.9.2 资源参数调优 335](#_Toc515544413)

[6.9.3 资源参数参考示例 339](#_Toc515544414)

[6.10 数据倾斜调优 340](#_Toc515544415)

[6.10.1 调优概述 340](#_Toc515544416)

[6.10.2 数据倾斜发生时的现象 340](#_Toc515544417)

[6.10.3 数据倾斜发生的原理 340](#_Toc515544418)

[6.10.4 如何定位导致数据倾斜的代码 341](#_Toc515544419)

[6.10.5 查看导致数据倾斜的key的数据分布情况 344](#_Toc515544420)

[6.10.6 数据倾斜的解决方案 345](#_Toc515544421)

[6.11 shuffle调优 358](#_Toc515544422)

[6.11.1 调优概述 358](#_Toc515544423)

[6.11.2 ShuffleManager发展概述 358](#_Toc515544424)

[6.11.3 HashShuffleManager运行原理 358](#_Toc515544425)

[6.11.4 SortShuffleManager运行原理 360](#_Toc515544426)

[6.11.5 shuffle相关参数调优 362](#_Toc515544427)

[6.12 Spark面试题汇总 365](#_Toc515544428)

[1. spark中的RDD是什么，有哪些特性 366](#_Toc515544429)

[2. 概述一下spark中的常用算子区别（map、mapPartitions、foreach、foreachPartition） 366](#_Toc515544430)

[3. 谈谈spark中的宽窄依赖 366](#_Toc515544431)

[4. spark中如何划分stage 367](#_Toc515544432)

[5. spark-submit的时候如何引入外部jar包 367](#_Toc515544433)

[6. spark 如何防止内存溢出 367](#_Toc515544434)

[7. spark中cache和persist的区别 368](#_Toc515544435)

[8. 简要描述Spark分布式集群搭建的步骤 368](#_Toc515544436)

[9. spark中的数据倾斜的现象、原因、后果 368](#_Toc515544437)

[10. 如何解决spark中的数据倾斜问题 369](#_Toc515544438)

[11. flume整合sparkStreaming问题 370](#_Toc515544439)

[12. kafka整合sparkStreaming问题 372](#_Toc515544440)

[6.12.1 ----简答题 ---- 网上资料 --- 374](#_Toc515544441)

[6.12.2 -------Spark on Yarn面试篇 379](#_Toc515544442)

[6.12.3 -------spark sql 面试篇 383](#_Toc515544443)

[6.12.4 ----选择题 --- 383](#_Toc515544444)

[6.12.5 补充资料: (spark 集群standalone + spark on yarn) 385](#_Toc515544445)

# spark

## 什么是Spark

Spark是基于内存计算大数据分析引擎，提高了在大数据环境下数据处理的实时性。Spark仅仅只涉及到数据的计算，没有涉及到数据的存储。

### Spark的特点及相对于MapReduce的优势

MapReduce存在的问题

1. MapReduce框架局限性

　　1）仅支持Map和Reduce两种操作

　　2）处理效率低效。

　　　　a）Map中间结果写磁盘，Reduce写HDFS，多个MR之间通过HDFS交换数据; 任务调度和启动开销大;

　　　　b）无法充分利用内存

　　　　c）Map端和Reduce端均需要排序

　　3）不适合迭代计算(如机器学习、图计算等)，交互式处理(数据挖掘) 和流式处理(点击日志分析)

2. MapReduce编程不够灵活

　　1）尝试scala函数式编程语言

Spark的特点及优势

1. 高效(比MapReduce快10~100倍)

　　1）内存计算引擎，提供Cache机制来支持需要反复迭代计算或者多次数据共享，减少数据读取的IO开销

　　2）DAG引擎，减少多次计算之间中间结果写到HDFS的开销

　　3）使用多线程池模型来减少task启动开稍，shuffle过程中避免 不必要的sort操作以及减少磁盘IO操作

2. 易用

　　1）提供了丰富的API，支持Java，Scala，Python和R四种语言

　　2）代码量比MapReduce少2~5倍

1. 兼容性

可与Hadoop集成 读写HDFS/Hbase/Cassandra 与YARN集成

1. 通用性

Spark可以用于批处理、交互式查询（Spark SQL）、实时流处理（Spark Streaming）、机器学习（Spark MLlib）和图计算（GraphX）

### Spark HA高可用部署

\*Spark HA解决Master单点故障的两种方案:

1.基于文件系统的单点恢复(主要用于开发或测试环境)

2.基于zookeeper的Standby Masters(用于生产模式)

\*基于zookeeper的Spark HA高可用集群部署

(1)vim spark-env.sh

注释掉export SPARK\_MASTER\_HOST=hdp-node-01

(2)在spark-env.sh添加SPARK\_DAEMON\_JAVA\_OPTS，内容如下：

|  |
| --- |
| export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=hdp-node-01:2181,hdp-node-02:2181,hdp-node-03:2181 -Dspark.deploy.zookeeper.dir=/spark" |

参数说明

spark.deploy.recoveryMode：

恢复模式（Master重新启动的模式）有三种：(1)ZooKeeper (2) FileSystem (3)NONE

spark.deploy.zookeeper.url：ZooKeeper的Server地址

spark.deploy.zookeeper.dir：保存集群元数据信息的文件、目录。

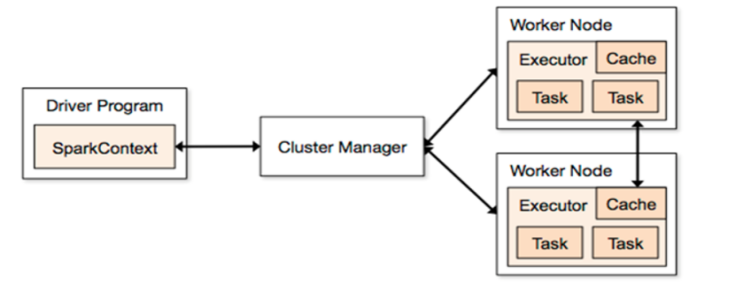
包括Worker，Driver和Application。

注意：

在普通模式下启动spark集群，只需要在主机上面执行start-all.sh 就可以了。

在高可用模式下启动spark集群，先需要在任意一台节点上启动start-all.sh命令。然后在另外一台节点上单独启动master。命令start-master.sh。

### Spark的角色



Driver Program ：运⾏main函数并且新建SparkContext的程序(初始化工作)。

Application：基于Spark的应用程序，包含了driver程序和集群上的executor(代码逻辑与运行资源)。

Cluster Manager：指的是在集群上获取资源的外部服务(提供外部运行资源,资源分配)。目前有三种类型

（1）Standalone: spark原生的资源管理，由Master负责资源的分配

（2）Apache Mesos:与hadoop MR兼容性良好的一种资源调度框架

（3）Hadoop Yarn: 主要是指Yarn中的ResourceManager

Master：集群中老大，负责资源的分配和任务的调度

Worker Node： 集群中任何可以运行Application代码的节点(小弟)，在Standalone模式中指的是通过slaves文件配置的Worker节点，在Spark on Yarn模式下就是NodeManager节点

Executor：是在一个worker node上为某应⽤启动的⼀个进程，该进程负责运⾏行任务，并且负责将数据存在内存或者磁盘上。每个应⽤都有各自独立的executor(运行当前任务所需要的资源)。

Task ：被送到某个executor上的工作单元(线程)。

Cache：Spark中设置的数据缓存

### Spark程序提交任务模式

普通模式提交任务(Master节点已知)：

|  |
| --- |
| bin/spark-submit \(提交任务脚本)  --class org.apache.spark.examples.SparkPi \(需要运行的主类)  --master spark://hdp-node-01:7077 \(指定任务提交到哪个Master)  --executor-memory 1G \(指定executor内存大小)  --total-executor-cores 2 \(总executor运行核数)  examples/jars/spark-examples\_2.11-2.0.2.jar \(程序jar包所在地址)  10(main方法所需参数) |

高可用模式提交任务：

|  |
| --- |
| bin/spark-submit \  --class org.apache.spark.examples.SparkPi \  --master spark://hdp-node-01:7077,hdp-node-02:7077,hdp-node-03:7077 \  (指定Master列表)  --executor-memory 1G \  --total-executor-cores 2 \  examples/jars/spark-examples\_2.11-2.0.2.jar \  10 |

### Spark-Shell

**读取本地文件**

1. 运行spark-shell --master local[N](N表线程数)
2. 编写scala代码

sc.textFile("file:///root///words.txt")

.flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_).collect

**读取HDFS上数据**

1.整合spark和HDFS，修改配置文件spark-env.sh

export HADOOP\_CONF\_DIR=/opt/bigdata/hadoop-2.6.4/etc/hadoop

1. 启动hdfs，然后重启spark集群
2. 向hdfs上传个文件
3. 在spark shell中用scala语言编写spark程序

**指定具体的master地址**

1. 执行启动命令:

spark-shell \

--master spark://hdp-node-01:7077 \

--executor-memory 1g \

--total-executor-cores 2

若没指定master地址则默认本地模式

1. 编写scala代码

### 在IDEA中编写WordCount程序

1.创建Maven项目并配置pom.xml

2.添加src/main/scala和src/test/scala，与pom.xml中的配置保持一致

3.新建一个scala class，类型为Object,并编写spark程序

4.使用Maven打包上传到Spark集群中的某个节点上

5.启动hdfs和Spark集群

6.使用spark-submit命令提交Spark应用

## Spark Streaming

### Spark Streming的特性

易用、容错、易整合

### Spark Streaming 对比 Storm

|  |  |  |
| --- | --- | --- |
|  | SparkStreaming | Storm |
| 开发语言 | Scala | Clojure |
| 编程模型 | DStream | Spout/Bolt |
| 实时性 | 准实时，批处理 | 实时流处理 |

## 算子操作

### Transformations

|  |  |
| --- | --- |
| **常用的转换算子** | |
| **Transformation** | **Meaning** |
| map(func) | 对DStream中的各个元素进行func函数操作，然后返回一个新的DStream |
| flatMap(func) | 与map方法类似，只不过各个输入项可以被输出为零个或多个输出项 |
| filter(func) | 过滤出所有函数func返回值为true的DStream元素并返回一个新的DStream |
| repartition(numPartitions) | 增加或减少DStream中的分区数，从而改变DStream的并行度 |
| union(otherStream) | 将源DStream和输入参数为otherDStream的元素合并，并返回一个新的DStream. |
| count() | 通过对DStream中的各个RDD中的元素进行计数，然后返回只有一个元素的RDD构成的DStream |
| reduce(func) | 对源DStream中的各个RDD中的元素利用func进行聚合操作，然后返回只有一个元素的RDD构成的新的DStream. |
| countByValue() | 对于元素类型为K的DStream，返回一个元素为（K,Long）键值对形式的新的DStream，Long对应的值为源DStream中各个RDD的key出现的次数 |
| reduceByKey(func, [numTasks]) | 利用func函数对源DStream中的key进行聚合操作，然后返回新的（K，V）对构成的DStream |
| join(otherStream, [numTasks]) | 输入为（K,V)、（K,W）类型的DStream，返回一个新的（K，（V，W））类型的DStream |
| cogroup(otherStream, [numTasks]) | 输入为（K,V)、（K,W）类型的DStream，返回一个新的 (K, Seq[V], Seq[W]) 元组类型的DStream |
| transform(func) | 通过RDD-to-RDD函数作用于DStream中的各个RDD，可以是任意的RDD操作，从而返回一个新的RDD |

|  |  |
| --- | --- |
| **特殊的转换算子（以下两个都需要设置checkpoint）** | |
| **Transformation** | **Meaning** |
| UpdateStateByKey（累加统计） | 根据key的之前状态值和key的新值，对key进行更新，返回一个新状态的DStream |
| **val** ds4: DStream[(String, Int)] = ds3.updateStateByKey((newValues, runningCount) => {  *Option*(newValues.sum + runningCount.getOrElse(0)) })  需要注意这个算子的参数是一个函数，函数的第一个参数newValues为ds3的值，runningCount为上一次的计算结果的值。 |
| Window Operations(开窗函数) | streaming-dstream-window  （1)红色的矩形就是一个窗口，窗口框住的是一段时间内的数据流。  （2）这里面每一个time都是时间单元，在官方的例子中，每隔window size是3 time unit, 而且每隔2个单位时间，窗口会slide一次。  所以需要两个参数：  a.窗口大小，一段时间内数据的容器。  b.滑动间隔，每隔多久计算一次。 |
| //构建StreamingContext对象，每个批处理的时间间隔 **val** scc = **new** StreamingContext(sc, *Seconds*(5))  …  **val** ds4: DStream[(String, Int)] = ds3.reduceByKeyAndWindow((a: Int, b: Int) => {  a + b }, *Seconds*(5), *Seconds*(5))  注意  窗口大小和滑动间隔 一定要是设定的数据批处理间隔的整数倍，否则会报错 |

### OutputOperations

Output Operations可以将DStream的数据输出到外部的数据库或文件系统，当某个Output Operations被调用时（与RDD的Action相同），spark streaming程序才会开始真正的计算过程

|  |  |
| --- | --- |
| Output Operation | Meaning |
| print() | 打印到控制台 |
| saveAsTextFiles(prefix, [suffix]) | 保存流的内容为文本文件，文件名为  "prefix-TIME\_IN\_MS[.suffix]". |
| saveAsObjectFiles(prefix, [suffix]) | 保存流的内容为SequenceFile，文件名为  "prefix-TIME\_IN\_MS[.suffix]". |
| saveAsHadoopFiles(prefix, [suffix]) | 保存流的内容为hadoop文件，文件名为  "prefix-TIME\_IN\_MS[.suffix]". |
| foreachRDD(func) | 对Dstream里面的每个RDD执行func |

### Spark Streming编程实战

|  |
| --- |
| 开发流程：  *\* 1、构建sparkContext对象 \* 2、构建StreamingContext对象 \* 3、创建输入流InputDStream \* 4、对DStream执行算子操作 \* 5、将执行结果保存或者输出* |

### SparkStreaming整合flume

Spark Streaming对接FlumeNG有两种方式：

**(1)、FlumeNG主动将消息Push推给Spark Streaming**

Spark程序需要启动一个端口接受数据，所以flume的配置文件中需要配置spark程序所运行的ip和端口

**(2)、Spark Streaming主动从flume 中Poll拉取数据。**

Flume需要启动一个端口来输出数据，所以flume配置文件中配置的是flume机器的主机名和端口，而在spark程序中需要绑定flume的ip和输出端口，这样spark程序才能主动拉取到数据

两种方式的启动顺序对比：

在push模式中,先启动spark application,进入等待状态,等待flume push数据,此时启动flume进行数据的传递.

    在pull模式中,spark application会从配置的端口pull数据,此时若flume还未启动,spark application会提示端口连接失败.所以需要先启动flume后启动spark application

### SparkStreaming整合Kafka

SparkStreaming整合Kafka有两种方式：

**（1）、Receiver方式**（KafkaUtils.createStream（调用kafka高级api）

这种方法使用Receiver来接收数据。Receiver是使用Kafka高级消费者API实现的。与所有的接收者一样，通过Receiver从Kafka接收的数据存储在Spark执行程序exector中，然后由Spark Streaming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，您必须在Spark Streaming（在Spark 1.2中引入）中额外启用wal预写日志（需要设置检查点checkpoint<*包含消息和offset*>**保证了消息不丢失**），同时保存所有接收到的Kafka数据写入hdfs的预写日志，以便所有数据都可以在失败时恢复。

每次消费完一条数据都需要在zk上更新offset的值，如果在更新offset过程中zk挂掉了，消费的消息偏移量并没有更新成功。下次程序在消费的时候就会**出现重复消费**的情况。

**（2）、Direct方式**（KafkaUtils.createDirectStream（调用kafka低级api）

定期地从kafka的topic下对应的partition中查询最新的偏移量，再根据偏移量范围在每个batch里面处理数据，Spark通过调用kafka简单的消费者Api（低级api）读取一定范围的数据。

Direct对比Receiver方式的优点：

1、简化并行

一个rdd分区对比topic上一个分区

2、高效

第一种实现数据的零丢失是将数据预先保存在WAL中，会复制一遍数据，会导致数据被拷贝两次，第一次是接受kafka中topic的数据，另一次是写到WAL中。而没有receiver的这种方式消除了这个问题

3、恰好一次

offset仅被保存在checkpoint目录里，消除了zk与ssc偏移量不一致问题，**解决重复消费问题**，缺点是无法使用基于zk的kafka监控工具

### Checkpoint

Spark Streaming的检查点具有容错机制，有足够的信息能够支持故障恢复。支持两种数据类型的检查点：元数据检查点和数据检查点。

**（1）元数据检查点**，在类似HDFS的容错存储上，保存Streaming计算信息。这种检查点用来恢复运行Streaming应用程序失败的Driver进程。

**（2）数据检查点**，在进行跨越多个批次合并数据的有状态操作时尤其重要。通过周期检查将转换RDD的中间状态进行可靠存储，借以切断无限增加的依赖。使用有状态的转换，如果updateStateByKey或者reduceByKeyAndWindow在应用程序中使用，那么需要提供检查点路径，对RDD进行周期性检查。

当程序因为异常重启时，如果检查点路径存在，则context将从检查点数据中重建。如果检查点目录不存在，将会新建context，并设置DStream。

## Spark SQL

### 几个知识点

#### Spark sql 的属性

① 易整合: 可以通过sql开发对应的应用程序, 也可以使用java/scala/phython/R编写的API来开发

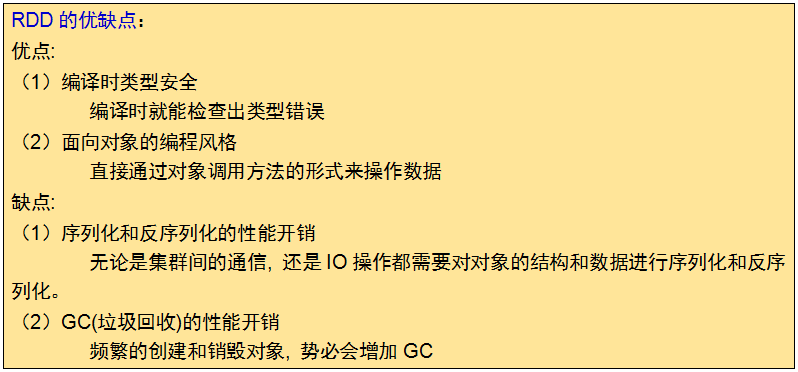
② 统一的数据源访问: 可以使用相同的方式来连接到不同的数据源

// 即: sparkSession.read.文件格式(文件路径)

③ 兼容hive: 可以使用spark sql来操作hive sql

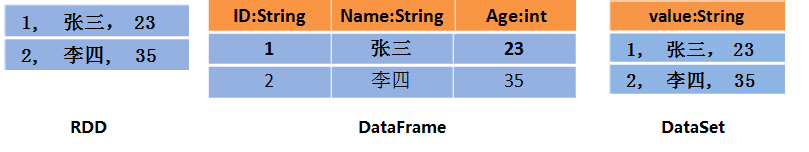
④ 标准的数据连接: spark sql可以使用标准的数据库连接(JDBC, ODBC)来操作关系型数据库

#### DataFrame与RDD的优缺点



DataFrame通过引入schema (即数据的结构信息)和off-heap（不在堆里面的内存，指的是除了不在堆的内存，使用操作系统上的内存），解决了RDD的缺点, Spark通过schame就能够读懂数据, 因此在通信和IO时就只需要序列化和反序列化数据, 而结构的部分就可以省略了；通过off-heap引入，可以快速的操作数据，避免大量的GC。但是却丢了RDD的优点，DataFrame不是类型安全的, API也不是面向对象风格的

#### DataFrame、DataSet、RDD的区别



DataSet包含了DataFrame的功能，Spark2.0中两者统一，DataFrame表示为DataSet[Row]，即DataSet的子集

① DataSet可以在编译时检查类型 (即类型安全)

② 并且是面向对象的编程接口

// 也就是说, DataSet弥补了DataFrame的缺点, 并且继承了它的优点

#### 读取数据源创建DataFrame

##### ① 读取文本文件创建DataFrame

sparkSession.read.text(“路径 + txt文件名”)

##### ② 读取json文件创建DataFrame

sparkSession.read.json(“路径 + json文件名”)

##### ③ 读取parquet列式存储格式文件创建DataFrame

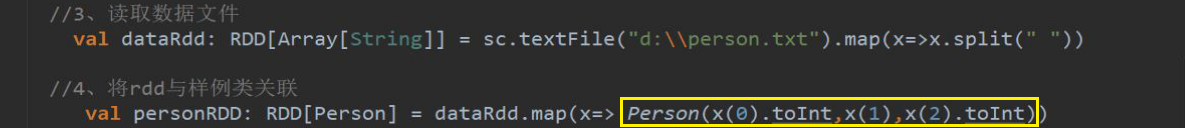
sparkSession.read.parquet(“路径 + parquet文件名”)

### 002 以编程方式执行Spark SQL查询

#### 编写Spark SQL程序实现RDD转换成DataFrame

##### ① 通过反射推断Schema

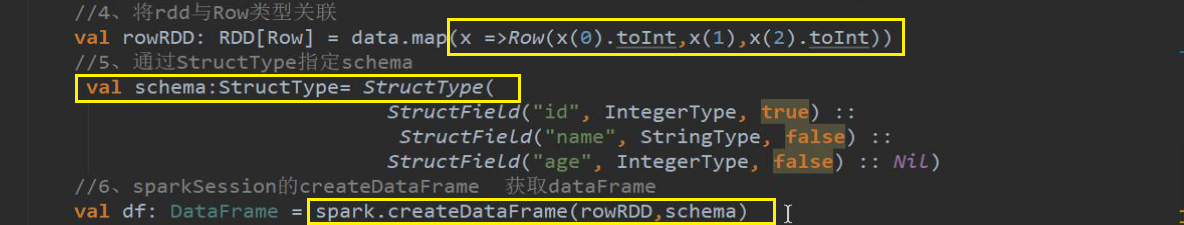
1





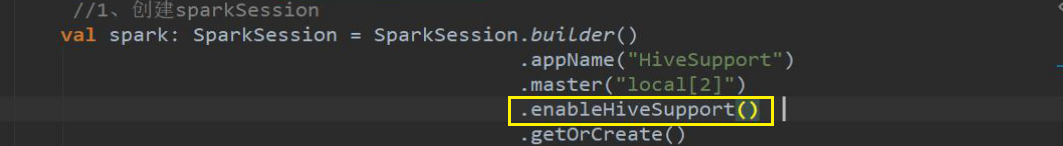
// 这种方式首先是定义样例类Person, 然后通过将rdd与该样例类关联(通过样例类创建schema，case class的参数名称会被利用反射机制作为列名) 生成最终的RDD, 最后该RDD调用toDF方法转换为DataFrame. 有了DataFrame之后就可以调用各种方法来执行spark sql查询了

##### ② 通过StructType直接指定Schema



// 这种方式首先是将rdd与row类型关联得到rowRDD, 然后定义structType类型数据用来指定schema. 最后通过sparkSession调用createDataFrame方法(含rowRDD以及schema两个参数)来得到DataFrame. 有了DataFrame之后就可以调用各种方法来执行spark sql查询了

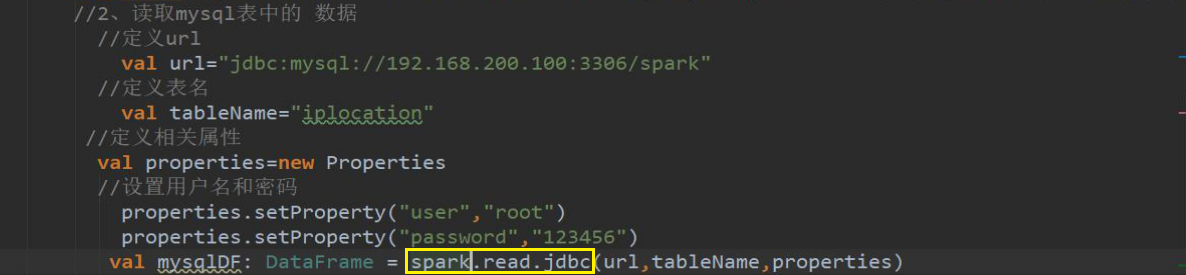
#### 编写Spark SQL程序操作HiveContext



// 设置为enableHiveSupport的sparkSession可以直接操作hql (即sparkSession调用sql方法, 里面可以直接写hql语句来操作)

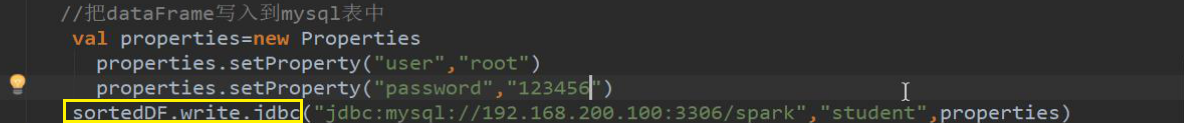
### JDBC数据源

#### SparkSql从MySQL中加载数据



// sparkSession通过调用read和jdbc方法(包含url, 表名, 用户名及密码这几个参数)来得到DataFrame. 有了DataFrame之后就可以调用各种方法来执行spark sql查询了

#### SparkSql将数据写入到MySQL中



// DataFrame通过调用write和jdbc方法(包含url, 表名, 用户名及密码这几个参数)就可以将数据直接写入到对应的mysql中

## SparkRDD

### **了解SparkRDD**

### 1）什么是RDD

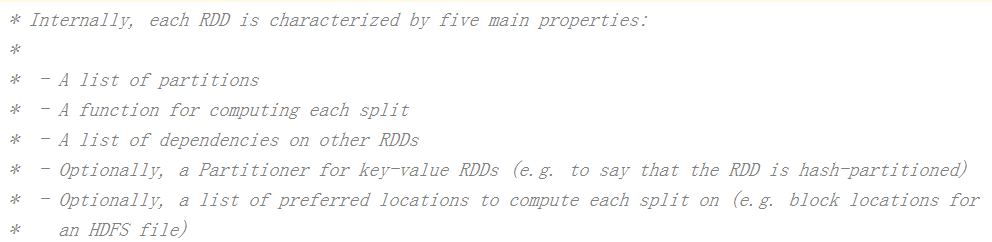
RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是Spark中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将数据缓存在内存中，后续的查询能够重用这些数据，这极大地提升了查询速度。

Dataset：一个数据集合，用于存放数据的。

Distributed：RDD中的数据是分布式存储的，可用于分布式计算。

Resilient：RDD中的数据可以存储在内存中或者磁盘中。

### 2）RDD的属性



1. A list of partitions ：一个分区（Partition）列表，数据集的基本组成单位。

对于RDD来说，每个分区都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分区个数，如果没有指定，那么就会采用默认值。（比如：读取HDFS上数据文件产生的RDD分区数跟block的个数相等）

1. A function for computing each split ：一个计算每个分区的函数。

Spark中RDD的计算是以分区为单位的，每个RDD都会实现compute函数以达到这个目的。

1. A list of dependencies on other RDDs：一个RDD会依赖于其他多个RDD，RDD之间的依赖关系。

RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

1. Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)：一个Partitioner，即RDD的分区函数（可选项）。

当前Spark中实现了两种类型的分区函数，一个是基于哈希的HashPartitioner，另外一 个是基于范围的RangePartitioner。只有对于key-value的RDD，才会有Partitioner，非key-value的RDD的Parititioner的值是None。Partitioner函数决定了parent RDD Shuffle输出时的分区数量。

1. Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)：一个列表，存储每个Partition的优先位置(可选项)。

对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置（spark进行任务分配的时候尽可能选择那些存有数据的worker节点来进行任务计算）

### 3.为什么会产生RDD？

1. 传统的MapReduce虽然具有自动容错、平衡负载和可拓展性的优点，但是其最大缺点是采用非循环式的数据流模型，使得在迭代计算中要进行大量的磁盘IO操作。RDD正是解决这一缺点的抽象方法。

（2） RDD是Spark提供的最重要的抽象的概念，它是一种具有容错机制的特殊集合，可以分布在集群的节点上，以函数式编程来操作集合，进行各种并行操作。可以把RDD的结果数据进行缓存，方便进行多次重用，避免重复计算。

### 4） RDD在Spark中的地位及作用

1. 为什么会有Spark？

因为传统的并行计算模型无法有效的解决迭代计算（iterative）和交互式计算（interactive）；而Spark的使命便是解决这两个问题，这也是他存在的价值和理由。

1. Spark如何解决迭代计算？

其主要实现思想就是RDD，把所有计算的数据保存在分布式的内存中。迭代计算通常情况下都是对同一个数据集做反复的迭代计算，数据在内存中将大大提升IO操作。这也是Spark涉及的核心：内存计算。

1. Spark如何实现交互式计算？

因为Spark是用scala语言实现的，Spark和scala能够紧密的集成，所以Spark可以完美的运用scala的解释器，使得其中的scala可以向操作本地集合对象一样轻松操作分布式数据集。

1. Spark和RDD的关系？

RDD是一种具有容错性、基于内存计算的抽象方法，RDD是Spark Core的底层核心，Spark则是这个抽象方法的实现。

### 创建RDD

1. 由一个已经存在的Scala集合创建。

val rdd1 = sc.parallelize(Array(1,2,3,4,5,6,7,8))

1. 由外部存储系统的文件创建。包括本地的文件系统，还有所有Hadoop支持的数据集，比如HDFS、Cassandra、HBase等。

val rdd2 = sc.textFile("/words.txt")

1. 已有的RDD经过算子转换生成新的RDD

val rdd3=rdd2.flatMap(\_.split(" "))

### RDD编程API

#### 1 RDD的算子分类

Transformation（转换）：根据数据集创建一个新的数据集，计算后返回一个新RDD；例如：一个rdd进行map操作后生了一个新的rdd。

Action（动作）：对rdd结果计算后返回一个数值value给驱动程序；

例如：collect算子将数据集的所有元素收集完成返回给驱动程序。

#### 2 Transformation

RDD中的所有转换都是延迟加载的，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的动作时，这些转换才会真正运行。这种设计让Spark更加有效率地运行。

常用的Transformation：

|  |  |
| --- | --- |
| **转换** | **含义** |
| **map(func)** | 返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成 |
| **filter(func)** | 返回一个新的RDD，该RDD由经过func函数计算后返回值为true的输入元素组成 |
| **flatMap(func)** | 类似于map，但是每一个输入元素可以被映射为0或多个输出元素（所以func应该返回一个序列，而不是单一元素） |
| **mapPartitions(func)** | 类似于map，但独立地在RDD的每一个分片上运行，因此在类型为T的RDD上运行时，func的函数类型必须是Iterator[T] => Iterator[U] |
| **mapPartitionsWithIndex(func)** | 类似于mapPartitions，但func带有一个整数参数表示分片的索引值，因此在类型为T的RDD上运行时，func的函数类型必须是  (Int, Interator[T]) => Iterator[U] |
| **union(otherDataset)** | 对源RDD和参数RDD求并集后返回一个新的RDD |
| **intersection(otherDataset)** | 对源RDD和参数RDD求交集后返回一个新的RDD |
| **distinct([numTasks]))** | 对源RDD进行去重后返回一个新的RDD |
| **groupByKey([numTasks])** | 在一个(K,V)的RDD上调用，返回一个(K, Iterator[V])的RDD |
| **reduceByKey(func, [numTasks])** | 在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用指定的reduce函数，将相同key的值聚合到一起，与groupByKey类似，reduce任务的个数可以通过第二个可选的参数来设置 |
| **sortByKey([ascending], [numTasks])** | 在一个(K,V)的RDD上调用，K必须实现Ordered接口，返回一个按照key进行排序的(K,V)的RDD |
| **sortBy(func,[ascending], [numTasks])** | 与sortByKey类似，但是更灵活 |
| **join(otherDataset, [numTasks])** | 在类型为(K,V)和(K,W)的RDD上调用，返回一个相同key对应的所有元素对在一起的(K,(V,W))的RDD |
| **cogroup(otherDataset, [numTasks])** | 在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的RDD |
| **coalesce(numPartitions)** | 减少 RDD 的分区数到指定值。 |
| **repartition(numPartitions)** | 重新给 RDD 分区 |
| **repartitionAndSortWithinPartitions(partitioner)** | 重新给 RDD 分区，并且每个分区内以记录的 key 排序 |

#### 3 Action

|  |  |
| --- | --- |
| **动作** | **含义** |
| **reduce(func)** | reduce将RDD中元素前两个传给输入函数，产生一个新的return值，新产生的return值与RDD中下一个元素（第三个元素）组成两个元素，再被传给输入函数，直到最后只有一个值为止。 |
| **collect()** | 在驱动程序中，以数组的形式返回数据集的所有元素 |
| **count()** | 返回RDD的元素个数 |
| **first()** | 返回RDD的第一个元素（类似于take(1)） |
| **take(n)** | 返回一个由数据集的前n个元素组成的数组 |
| **takeOrdered(n, [ordering])** | 返回自然顺序或者自定义顺序的前 n 个元素 |
| **saveAsTextFile(path)** | 将数据集的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统，对于每个元素，Spark将会调用to String方法，将它装换为文件中的文本 |
| **saveAsSequenceFile(path)** | 将数据集中的元素以Hadoop sequencefile的格式保存到指定的目录下，可以使HDFS或者其他Hadoop支持的文件系统。 |
| **saveAsObjectFile(path)** | 将数据集的元素，以 Java 序列化的方式保存到指定的目录下 |
| **countByKey()** | 针对(K,V)类型的RDD，返回一个(K,Int)的map，表示每一个key对应的元素个数。 |
| **foreach(func)** | 在数据集的每一个元素上，运行函数func |
| **foreachPartition(func)** | 在数据集的每一个分区上，运行函数func |

### RDD常用的算子操作

Spark Rdd的所有算子操作，请见《sparkRDD函数详解.docx》

启动spark-shell 进行测试：

spark-shell --master spark://node1:7077

#### 练习1：map、filter

//通过并行化生成rdd

val rdd1 = sc.parallelize(List(5, 6, 4, 7, 3, 8, 2, 9, 1, 10))

//对rdd1里的每一个元素乘2然后排序

val rdd2 = rdd1.map(\_ \* 2).sortBy(x => x, true)

//过滤出大于等于5的元素

val rdd3 = rdd2.filter(\_ >= 5)

//将元素以数组的方式在客户端显示

rdd3.collect

#### 练习2：flatMap

val rdd1 = sc.parallelize(Array("a b c", "d e f", "h i j"))

//将rdd1里面的每一个元素先切分在压平

val rdd2 = rdd1.flatMap(\_.split(" "))

rdd2.collect

#### 练习3：交集、并集

val rdd1 = sc.parallelize(List(5, 6, 4, 3))

val rdd2 = sc.parallelize(List(1, 2, 3, 4))

//求并集

val rdd3 = rdd1.union(rdd2)

//求交集

val rdd4 = rdd1.intersection(rdd2)

//去重

rdd3.distinct.collect

rdd4.collect

#### 练习4：join、groupByKey

val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2)))

val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)))

//求join

val rdd3 = rdd1.join(rdd2)

rdd3.collect

//求并集

val rdd4 = rdd1 union rdd2

rdd4.collect

//按key进行分组

val rdd5=rdd4.groupByKey

rdd5.collect

#### 练习5：cogroup

val rdd1 = sc.parallelize(List(("tom", 1), ("tom", 2), ("jerry", 3), ("kitty", 2)))

val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("jim", 2)))

//cogroup

val rdd3 = rdd1.cogroup(rdd2)

//注意cogroup与groupByKey的区别

rdd3.collect

#### 练习6：reduce

val rdd1 = sc.parallelize(List(1, 2, 3, 4, 5))

//reduce聚合

val rdd2 = rdd1.reduce(\_ + \_)

rdd2.collect

#### 练习7：reduceByKey、sortByKey

val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2), ("shuke", 1)))

val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 3), ("shuke", 2), ("kitty", 5)))

val rdd3 = rdd1.union(rdd2)

//按key进行聚合

val rdd4 = rdd3.reduceByKey(\_ + \_)

rdd4.collect

//按value的降序排序

val rdd5 = rdd4.map(t => (t.\_2, t.\_1)).sortByKey(false).map(t => (t.\_2, t.\_1))

rdd5.collect

#### 练习8：repartition、coalesce

val rdd1 = sc.parallelize(1 to 10,3)

//利用repartition改变rdd1分区数

//减少分区

rdd1.repartition(2).partitions.size

//增加分区

rdd1.repartition(4).partitions.size

//利用coalesce改变rdd1分区数

//减少分区

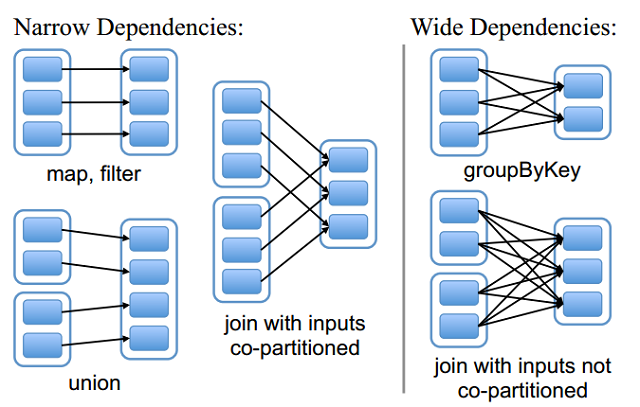
rdd1.coalesce(2).partitions.size

注意：repartition可以增加和减少rdd中的分区数，coalesce只能减少rdd分区数，增加rdd分区数不会生效。

### RDD的依赖关系

#### 1 RDD的依赖

RDD和它依赖的父RDD的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。



#### 2 窄依赖

窄依赖指的是每一个父RDD的Partition最多被子RDD的一个Partition使用

总结：窄依赖我们形象的比喻为独生子女

#### 3 宽依赖

宽依赖指的是多个子RDD的Partition会依赖同一个父RDD的Partition

总结：宽依赖我们形象的比喻为超生

#### 4 Lineage（血统）

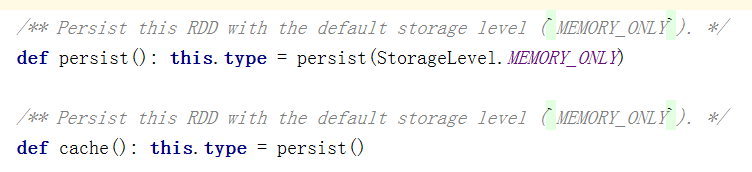
RDD只支持粗粒度转换，即只记录单个块上执行的单个操作。将创建RDD的一系列Lineage（即血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

### RDD的缓存

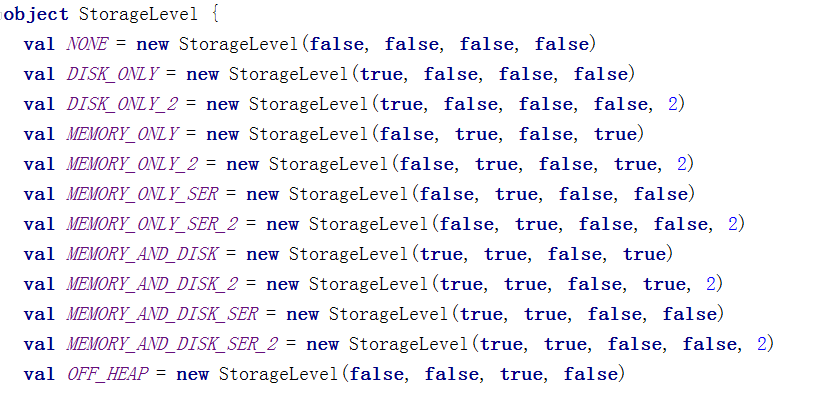
Spark速度非常快的原因之一，就是在不同操作中可以在内存中持久化或者缓存数据集。当持久化某个RDD后，每一个节点都将把计算分区结果保存在内存中，对此RDD或衍生出的RDD进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD相关的持久化和缓存，是Spark最重要的特征之一。可以说，缓存是Spark构建迭代式算法和快速交互式查询的关键。

#### RDD缓存方式

RDD通过persist方法或cache方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的action时，该RDD将会被缓存在计算节点的内存中，并供后面重用。



通过查看源码发现cache最终也是调用了persist方法，默认的存储级别都是仅在内存存储一份，Spark的存储级别还有好多种，存储级别在object StorageLevel中定义的。

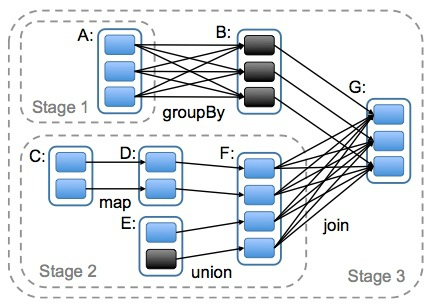


缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除，RDD的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于RDD的一系列转换，丢失的数据会被重算，由于RDD的各个Partition是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部Partition。

### DAG的生成

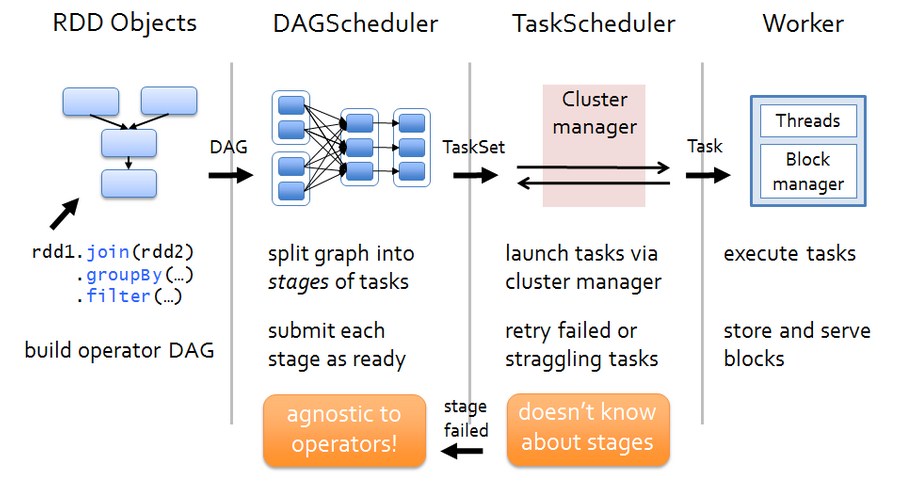
#### 什么是DAG

DAG(Directed Acyclic Graph)叫做有向无环图，原始的RDD通过一系列的转换就形成了DAG，根据RDD之间依赖关系的不同将DAG划分成不同的Stage(调度阶段)。对于窄依赖，partition的转换处理在一个Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，因此宽依赖是划分Stage的依据。



### Spark任务调度

#### 1任务调度流程图



各个RDD之间存在着依赖关系，这些依赖关系就形成有向无环图DAG，DAGScheduler对这些依赖关系形成的DAG进行Stage划分，划分的规则很简单，从后往前回溯，遇到窄依赖加入本stage，遇见宽依赖进行Stage切分。完成了Stage的划分。DAGScheduler基于每个Stage生成TaskSet,并将TaskSet提交给TaskScheduler。TaskScheduler 负责具体的task调度,最后在Worker节点上启动task。

#### 2 DAGScheduler

1. DAGScheduler对DAG有向无环图进行Stage划分。
2. 记录哪个RDD或者 Stage 输出被物化（缓存），通常在一个复杂的shuffle之后，通常物化一下(cache、persist)，方便之后的计算。
3. 重新提交shuffle输出丢失的stage（stage内部计算出错）给TaskScheduler
4. 将 Taskset 传给底层调度器
5. – spark-cluster TaskScheduler
6. – yarn-cluster YarnClusterScheduler
7. – yarn-client YarnClientClusterScheduler

#### 3TaskScheduler

（1）为每一个TaskSet构建一个TaskSetManager 实例管理这个TaskSet 的生命周期

（2）数据本地性决定每个Task最佳位置

（3）提交 taskset( 一组task) 到集群运行并监控

（4）推测执行，碰到计算缓慢任务需要放到别的节点上重试

（5）重新提交Shuffle输出丢失的Stage给DAGScheduler

## RDD容错机制之checkpoint

### checkpoint是什么

（1）、Spark 在生产环境下经常会面临transformation的RDD非常多（例如一个Job中包含1万个RDD）或者具体transformation的RDD本身计算特别复杂或者耗时（例如计算时长超过1个小时），这个时候就要考虑对计算结果数据持久化保存；

（2）、Spark是擅长多步骤迭代的，同时擅长基于Job的复用，这个时候如果能够对曾经计算的过程产生的数据进行复用，就可以极大的提升效率；

（3）、如果采用persist把数据放在内存中，虽然是快速的，但是也是最不可靠的；如果把数据放在磁盘上，也不是完全可靠的！例如磁盘会损坏，系统管理员可能清空磁盘。

（4）、Checkpoint的产生就是为了相对而言更加可靠的持久化数据，在Checkpoint的时候可以指定把数据放在本地，并且是多副本的方式，但是在生产环境下是放在HDFS上，这就天然的借助了HDFS高容错、高可靠的特征来完成了最大化的可靠的持久化数据的方式；

假如进行一个1万个算子操作，在9000个算子的时候persist，数据还是有可能丢失的，但是如果checkpoint，数据丢失的概率几乎为0。

### checkpoint原理机制

1. 当RDD使用cache机制从内存中读取数据，如果数据没有读到，会使用checkpoint机制读取数据。此时如果没有checkpoint机制，那么就需要找到父RDD重新计算数据了，因此checkpoint是个很重要的容错机制。checkpoint就是对于一个RDD chain（链）如果后面需要反复使用某些中间结果RDD，可能因为一些故障导致该中间数据丢失，那么就可以针对该RDD启动checkpoint机制，使用checkpoint首先需要调用sparkContext的setCheckpoint方法，设置一个容错文件系统目录，比如hdfs，然后对RDD调用checkpoint方法。之后在RDD所处的job运行结束后，会启动一个单独的job来将checkpoint过的数据写入之前设置的文件系统持久化，进行高可用。所以后面的计算在使用该RDD时，如果数据丢失了，但是还是可以从它的checkpoint中读取数据，不需要重新计算。
2. persist或者cache与checkpoint的区别在于,前者持久化只是将数据保存在BlockManager中但是其lineage是不变的，但是后者checkpoint执行完后，rdd已经没有依赖RDD，只有一个checkpointRDD，checkpoint之后，RDD的lineage就改变了。persist或者cache持久化的数据丢失的可能性更大，因为可能磁盘或内存被清理，但是checkpoint的数据通常保存到hdfs上，放在了高容错文件系统。

**问题：哪些 RDD 需要 cache？**

会被重复使用的（但不能太大）。

**问题：用户怎么设定哪些 RDD 要 cache？**

因为用户只与 driver program 打交道，因此只能用 rdd.cache() 去 cache 用户能看到的 RDD。所谓能看到指的是调用 transformation() 后生成的 RDD，而某些在 transformation() 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如 reduceByKey() 中会生成的 ShuffledRDD、MapPartitionsRDD 是不能被用户直接 cache 的。

**问题：哪些 RDD 需要 cache？**

会被重复使用的（但不能太大）。

**问题：用户怎么设定哪些 RDD 要 cache？**

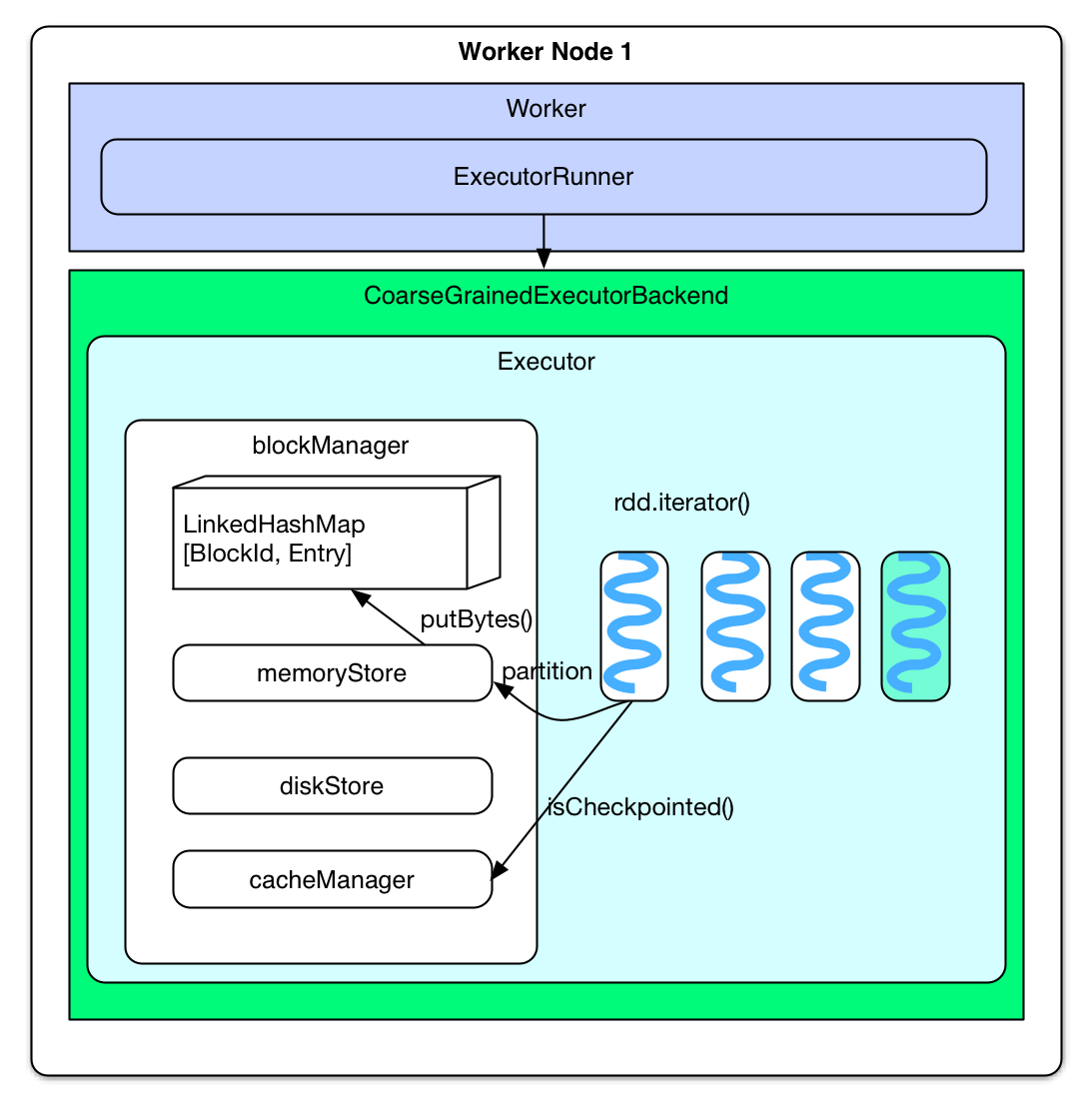
因为用户只与 driver program 打交道，因此只能用 rdd.cache() 去 cache 用户能看到的 RDD。所谓能看到指的是调用 transformation() 后生成的 RDD，而某些在 transformation() 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如 reduceByKey() 中会生成的 ShuffledRDD、MapPartitionsRDD 是不能被用户直接 cache 的。

**问题：driver program 设定 rdd.cache() 后，系统怎么对 RDD 进行 cache？**

先不看实现，自己来想象一下如何完成 cache：当 task 计算得到 RDD 的某个 partition 的第一个 record 后，就去判断该 RDD 是否要被 cache，如果要被 cache 的话，将这个 record 及后续计算的到的 records 直接丢给本地 blockManager 的 memoryStore，如果 memoryStore 存不下就交给 diskStore 存放到磁盘。

**实际实现与设想的基本类似，区别在于：**将要计算 RDD partition 的时候（而不是已经计算得到第一个 record 的时候）就去判断 partition 要不要被 cache。如果要被 cache 的话，先将 partition 计算出来，然后 cache 到内存。cache 只使用 memory，写磁盘的话那就叫 checkpoint 了。

调用 rdd.cache() 后， rdd 就变成 persistRDD 了，其 StorageLevel 为 MEMORY\_ONLY。persistRDD 会告知 driver 说自己是需要被 persist 的。



如果用代码表示：

rdd.iterator()

=> SparkEnv.get.cacheManager.getOrCompute(thisRDD, split, context, storageLevel)

=> key = RDDBlockId(rdd.id, split.index)

=> blockManager.get(key)

=> computedValues = rdd.computeOrReadCheckpoint(split, context)

if (isCheckpointed) firstParent[T].iterator(split, context)

else compute(split, context)

=> elements = new ArrayBuffer[Any]

=> elements ++= computedValues

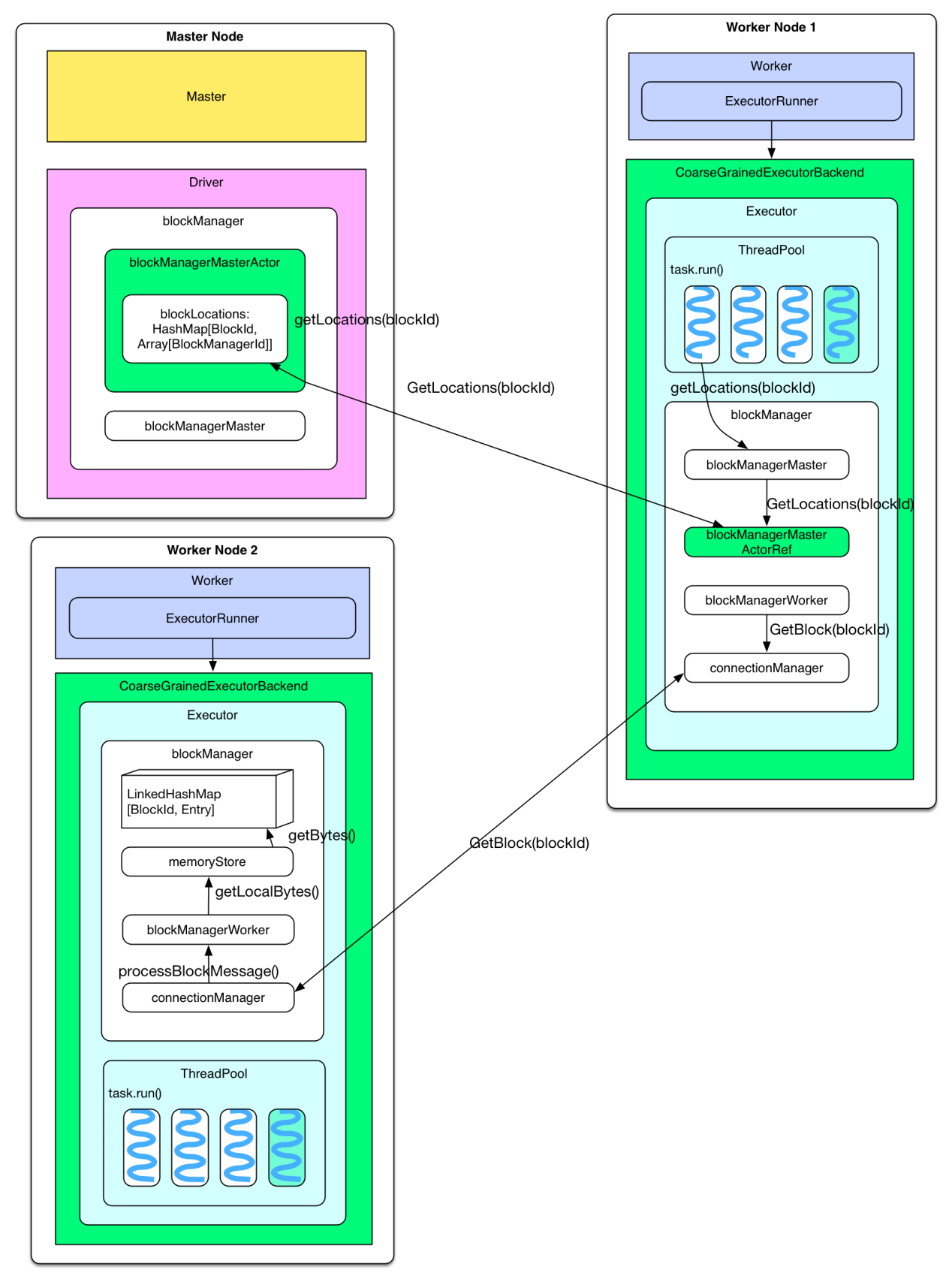
=> updatedBlocks = blockManager.put(key, elements, tellMaster = true)

当 rdd.iterator() 被调用的时候，也就是要计算该 rdd 中某个 partition 的时候，会先去 cacheManager 那里领取一个 blockId，表明是要存哪个 RDD 的哪个 partition，这个 blockId 类型是 RDDBlockId（memoryStore 里面可能还存放有 task 的 result 等数据，因此 blockId 的类型是用来区分不同的数据）。然后去 blockManager 里面查看该 partition 是不是已经被 checkpoint 了，如果是，表明以前运行过该 task，那就不用计算该 partition 了，直接从 checkpoint 中读取该 partition 的所有 records 放到叫做 elements 的 ArrayBuffer 里面。如果没有被 checkpoint 过，先将 partition 计算出来，然后将其所有 records 放到 elements 里面。最后将 elements 交给 blockManager 进行 cache。

blockManager 将 elements（也就是 partition） 存放到 memoryStore 管理的 LinkedHashMap[BlockId, Entry] 里面。如果 partition 大于 memoryStore 的存储极限（默认是 60% 的 heap），那么直接返回说存不下。如果剩余空间也许能放下，会先 drop 掉一些早先被 cached 的 RDD 的 partition，为新来的 partition 腾地方，如果腾出的地方够，就把新来的 partition 放到 LinkedHashMap 里面，腾不出就返回说存不下。注意 drop 的时候不会去 drop 与新来的 partition 同属于一个 RDD 的 partition。drop 的时候先 drop 最早被 cache 的 partition。（说好的 LRU 替换算法呢？）

**问题：cached RDD 怎么被读取？**

下次计算（一般是同一 application 的下一个 job 计算）时如果用到 cached RDD，task 会直接去 blockManager 的 memoryStore 中读取。具体地讲，当要计算某个 rdd 中的 partition 时候（通过调用 rdd.iterator()）会先去 blockManager 里面查找是否已经被 cache 了，如果 partition 被 cache 在本地，就直接使用 blockManager.getLocal() 去本地 memoryStore 里读取。如果该 partition 被其他节点上 blockManager cache 了，会通过 blockManager.getRemote() 去其他节点上读取，读取过程如下图。



**获取 cached partitions 的存储位置：**partition 被 cache 后所在节点上的 blockManager 会通知 driver 上的 blockMangerMasterActor 说某 rdd 的 partition 已经被我 cache 了，这个信息会存储在 blockMangerMasterActor 的 blockLocations: HashMap中。等到 task 执行需要 cached rdd 的时候，会调用 blockManagerMaster 的 getLocations(blockId) 去询问某 partition 的存储位置，这个询问信息会发到 driver 那里，driver 查询 blockLocations 获得位置信息并将信息送回。

**读取其他节点上的 cached partition：**task 得到 cached partition 的位置信息后，将 GetBlock(blockId) 的请求通过 connectionManager 发送到目标节点。目标节点收到请求后从本地 blockManager 那里的 memoryStore 读取 cached partition，最后发送回来。

### **Checkpoint常见面试问题**

**问题：哪些 RDD 需要 checkpoint？**

运算时间很长或运算量太大才能得到的 RDD，computing chain 过长或依赖其他 RDD 很多的 RDD。 实际上，将 ShuffleMapTask 的输出结果存放到本地磁盘也算是 checkpoint，只不过这个 checkpoint 的主要目的是去 partition 输出数据。

**问题：什么时候 checkpoint？**

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。**也就是说需要 checkpoint 的 RDD 会被计算两次。因此，在使用 rdd.checkpoint() 的时候，建议加上 rdd.cache()，**这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。其实 Spark 提供了 rdd.persist(StorageLevel.DISK\_ONLY) 这样的方法，相当于 cache 到磁盘上，这样可以做到 rdd 第一次被计算得到时就存储到磁盘上，但这个 persist 和 checkpoint 有很多不同，之后会讨论。

**问题：checkpoint 怎么实现？**

RDD 需要经过 [ Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed ] 这几个阶段才能被 checkpoint。

**Initialized：** 首先 driver program 需要使用 rdd.checkpoint() 去设定哪些 rdd 需要 checkpoint，设定后，该 rdd 就接受 RDDCheckpointData 管理。用户还要设定 checkpoint 的存储路径，一般在 HDFS 上。

**marked for checkpointing：**初始化后，RDDCheckpointData 会将 rdd 标记为 MarkedForCheckpoint。

**checkpointing in progress：**每个 job 运行结束后会调用 finalRdd.doCheckpoint()，finalRdd 会顺着 computing chain 回溯扫描，碰到要 checkpoint 的 RDD 就将其标记为 CheckpointingInProgress，然后将写磁盘（比如写 HDFS）需要的配置文件（如 core-site.xml 等）broadcast 到其他 worker 节点上的 blockManager。完成以后，启动一个 job 来完成 checkpoint（使用 rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString, broadcastedConf))）。

**checkpointed：**job 完成 checkpoint 后，将该 rdd 的 dependency 全部清掉，并设定该 rdd 状态为 checkpointed。然后，**为该 rdd 强加一个依赖，设置该 rdd 的 parent rdd 为 CheckpointRDD**，该 CheckpointRDD 负责以后读取在文件系统上的 checkpoint 文件，生成该 rdd 的 partition。

有意思的是我在 driver program 里 checkpoint 了两个 rdd，结果只有一个（下面的 result）被 checkpoint 成功，pairs2 没有被 checkpoint，也不知道是 bug 还是故意只 checkpoint 下游的 RDD：

val data1 = Array[(Int, Char)]((1, 'a'), (2, 'b'), (3, 'c'),

(4, 'd'), (5, 'e'), (3, 'f'), (2, 'g'), (1, 'h'))

val pairs1 = sc.parallelize(data1, 3)

val data2 = Array[(Int, Char)]((1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'))

val pairs2 = sc.parallelize(data2, 2)

pairs2.checkpoint

val result = pairs1.join(pairs2)

result.checkpoint

**问题：怎么读取 checkpoint 过的 RDD？**

在 runJob() 的时候会先调用 finalRDD 的 partitions() 来确定最后会有多个 task。rdd.partitions() 会去检查（通过 RDDCheckpointData 去检查，因为它负责管理被 checkpoint 过的 rdd）该 rdd 是会否被 checkpoint 过了，如果该 rdd 已经被 checkpoint 过了，直接返回该 rdd 的 partitions 也就是 Array[Partition]。

当调用 rdd.iterator() 去计算该 rdd 的 partition 的时候，会调用 computeOrReadCheckpoint(split: Partition) 去查看该 rdd 是否被 checkpoint 过了，如果是，就调用该 rdd 的 parent rdd 的 iterator() 也就是 CheckpointRDD.iterator()，CheckpointRDD 负责读取文件系统上的文件，生成该 rdd 的 partition。**这就解释了为什么那么 trickly 地为 checkpointed rdd 添加一个 parent CheckpointRDD。**

**问题：cache 与 checkpoint 的区别？**

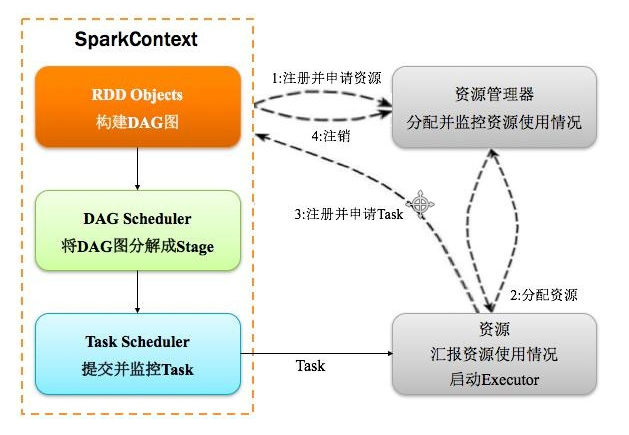
关于这个问题，Tathagata Das 有一段回答: There is a significant difference between cache and checkpoint. Cache materializes the RDD and keeps it in memory and/or disk（其实只有 memory）. But the lineage（也就是 computing chain） of RDD (that is, seq of operations that generated the RDD) will be remembered, so that if there are node failures and parts of the cached RDDs are lost, they can be regenerated. However, **checkpoint saves the RDD to an HDFS file and actually forgets the lineage completely.** This is allows long lineages to be truncated and the data to be saved reliably in HDFS (which is naturally fault tolerant by replication).

深入一点讨论，rdd.persist(StorageLevel.DISK\_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉（**话说怎么 remove checkpoint 过的 RDD？**），是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。

## Spark运行架构

### Spark运行基本流程

Spark运行基本流程参见下面示意图：



1. 构建Spark Application的运行环境（启动SparkContext），SparkContext向资源管理器（可以是Standalone、Mesos或YARN）注册并申请运行Executor资源；
2. 资源管理器分配Executor资源并启动Executor，Executor运行情况将随着心跳发送到资源管理器上；
3. SparkContext构建成DAG图，将DAG图分解成Stage，并把Taskset发送给Task Scheduler。Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行同时SparkContext将应用程序代码发放给Executor。
4. Task在Executor上运行，运行完毕释放所有资源。

### Spark运行架构特点

Spark运行架构特点：

* 每个Application获取专属的executor进程，该进程在Application期间一直驻留，并以多线程方式运行tasks。
* Spark任务与资源管理器无关，只要能够获取executor进程，并能保持相互通信就可以了。
* 提交SparkContext的Client应该靠近Worker节点（运行Executor的节点)，最好是在同一个Rack里，因为Spark程序运行过程中SparkContext和Executor之间有大量的信息交换；如果想在远程集群中运行，最好使用RPC将SparkContext提交给集群，不要远离Worker运行SparkContext。
* Task采用了数据本地性和推测执行的优化机制。

应用内调度

在指定的 Spark 应用内部（对应同一 SparkContext 实例），多个线程可能并发地提交 Spark 作业（job）。在本节中，作业（job）是指，由 Spark action 算子（如 : collect）触发的一系列计算任务的集合。Spark 调度器是完全线程安全的，并且能够支持 Spark 应用同时处理多个请求（比如 : 来自不同用户的查询）。

默认，Spark 应用内部使用 FIFO 调度策略。每个作业被划分为多个阶段（stage）（例如 : map 阶段和 reduce 阶段），第一个作业在其启动后会优先获取所有的可用资源，然后是第二个作业再申请，再第三个……。如果前面的作业没有把集群资源占满，则后续的作业可以立即启动运行，否则，后提交的作业会有明显的延迟等待。

不过从 Spark 0.8 开始，Spark 也能支持各个作业间的公平（Fair）调度。公平调度时，Spark 以轮询的方式给每个作业分配资源，因此所有的作业获得的资源大体上是平均分配。这意味着，即使有大作业在运行，小的作业再提交也能立即获得计算资源而不是等待前面的作业结束，大大减少了延迟时间。这种模式特别适合于多用户配置。 要启用公平调度器，只需设置一下 SparkContext 中 spark.scheduler.mode 属性为 FAIR 即可 :

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

conf.set("spark.scheduler.mode", "FAIR")

**val** sc **=** **new** **SparkContext**(conf)

公平调度资源池

公平调度器还可以支持将作业分组放入资源池（pool），然后给每个资源池配置不同的选项（如 : 权重）。这样你就可以给一些比较重要的作业创建一个“高优先级”资源池，或者你也可以把每个用户的作业分到一组，这样一来就是各个用户平均分享集群资源，而不是各个作业平分集群资源。Spark 公平调度的实现方式基本都是模仿 [Hadoop Fair Scheduler](http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html). 来实现的。

默认情况下，新提交的作业都会进入到默认资源池中，不过作业对应于哪个资源池，可以在提交作业的线程中用 SparkContext.setLocalProperty 设定 spark.scheduler.pool 属性。示例代码如下 :

*// Assuming sc is your SparkContext variable*

sc.setLocalProperty("spark.scheduler.pool", "pool1")

一旦设好了局部属性，所有该线程所提交的作业（即 : 在该线程中调用action算子，如 : RDD.save/count/collect 等）都会使用这个资源池。这个设置是以线程为单位保存的，你很容易实现用同一线程来提交同一用户的所有作业到同一个资源池中。同样，如果需要清除资源池设置，只需在对应线程中调用如下代码 :

sc.setLocalProperty("spark.scheduler.pool", **null**)

资源池默认行为

默认地，各个资源池之间平分整个集群的资源（包括 default 资源池），但在资源池内部，默认情况下，作业是 FIFO 顺序执行的。举例来说，如果你为每个用户创建了一个资源池，那么久意味着各个用户之间共享整个集群的资源，但每个用户自己提交的作业是按顺序执行的，而不会出现后提交的作业抢占前面作业的资源。

配置资源池属性

资源池的属性需要通过配置文件来指定。每个资源池都支持以下3个属性 :

* schedulingMode: 可以是 FIFO 或 FAIR，控制资源池内部的作业是如何调度的。
* weight: 控制资源池相对其他资源池，可以分配到资源的比例。默认所有资源池的 weight 都是 1。如果你将某个资源池的 weight 设为 2，那么该资源池中的资源将是其他池子的2倍。如果将 weight 设得很高，如 1000，可以实现资源池之间的调度优先级 – 也就是说，weight=1000 的资源池总能立即启动其对应的作业。
* minShare: 除了整体 weight 之外，每个资源池还能指定一个最小资源分配值（CPU 个数），管理员可能会需要这个设置。公平调度器总是会尝试优先满足所有活跃（active）资源池的最小资源分配值，然后再根据各个池子的 weight 来分配剩下的资源。因此，minShare 属性能够确保每个资源池都能至少获得一定量的集群资源。minShare 的默认值是 0。

资源池属性是一个 XML 文件，可以基于 conf/fairscheduler.xml.template 修改，然后在 [SparkConf](http://spark.apachecn.org/docs/cn/2.2.0/configuration.html#spark-properties). 的 spark.scheduler.allocation.file 属性指定文件路径：

conf.set("spark.scheduler.allocation.file", "/path/to/file")

面试题

1.请列出spark的调度器，简述调度原理

分别是DAGScheduler、TaskScheduler。

DAGScheduler

1. DAGScheduler对DAG有向无环图进行Stage划分。
2. 记录哪个RDD或者 Stage 输出被物化（缓存），通常在一个复杂的shuffle之后，通常物化一下(cache、persist)，方便之后的计算。
3. 重新提交shuffle输出丢失的stage（stage内部计算出错）给TaskScheduler
4. 将 Taskset 传给底层调度器
5. – spark-cluster TaskScheduler
6. – yarn-cluster YarnClusterScheduler
7. – yarn-client YarnClientClusterScheduler

TaskScheduler

为每一个TaskSet构建一个TaskSetManager 实例管理这个TaskSet 的生命周期

（2）数据本地性决定每个Task最佳位置

（3）提交 taskset( 一组task) 到集群运行并监控

（4）推测执行，碰到计算缓慢任务需要放到别的节点上重试

（5）重新提交Shuffle输出丢失的Stage给DAGScheduler

2.rdd的特点

RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将数据缓存在内存中，后续的查询能够重用这些数据，这极大地提升了查询速度。

Spark优化指南完全版

---------spark 优化指南 --------

[Apache Spark 内存管理详解（转载）](http://www.cnblogs.com/shishanyuan/p/8457696.html)

[Spark性能优化指南——基础篇（转载）](http://www.cnblogs.com/shishanyuan/p/8454323.html)

[Spark性能优化指南——高级篇（转载）](http://www.cnblogs.com/shishanyuan/p/8454310.html)

[Spark官方调优文档翻译（转载）](http://www.cnblogs.com/shishanyuan/p/8481854.html)

整套调优方案主要分为开发调优、资源调优、数据倾斜调优、shuffle调优几个部分

## 开发调优

### 调优概述

Spark性能优化的第一步，就是要在开发Spark作业的过程中注意和应用一些性能优化的基本原则。开发调优，就是要让大家了解以下一些Spark基本开发原则，包括：RDD lineage设计、算子的合理使用、特殊操作的优化等。在开发过程中，时时刻刻都应该注意以上原则，并将这些原则根据具体的业务以及实际的应用场景，灵活地运用到自己的Spark作业中。

### 原则一：避免创建重复的RDD

通常来说，我们在开发一个Spark作业时，首先是基于某个数据源（比如Hive表或HDFS文件）创建一个初始的RDD；接着对这个RDD执行某个算子操作，然后得到下一个RDD；以此类推，循环往复，直到计算出最终我们需要的结果。在这个过程中，多个RDD会通过不同的算子操作（比如map、reduce等）串起来，这个“RDD串”，就是RDD lineage，也就是“RDD的血缘关系链”。

我们在开发过程中要注意：对于同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据。

一些Spark初学者在刚开始开发Spark作业时，或者是有经验的工程师在开发RDD lineage极其冗长的Spark作业时，可能会忘了自己之前对于某一份数据已经创建过一个RDD了，从而导致对于同一份数据，创建了多个RDD。这就意味着，我们的Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，进而增加了作业的性能开销。

一个简单的例子

[制代码](javascript:void(0);)

1 // 需要对名为“hello.txt”的HDFS文件进行一次map操作，再进行一次reduce操作。也就是说，需要对一份数据执行两次算子操作。 2 3 // 错误的做法：对于同一份数据执行多次算子操作时，创建多个RDD。 4 // 这里执行了两次textFile方法，针对同一个HDFS文件，创建了两个RDD出来，然后分别对每个RDD都执行了一个算子操作。 5 // 这种情况下，Spark需要从HDFS上两次加载hello.txt文件的内容，并创建两个单独的RDD；第二次加载HDFS文件以及创建RDD的性能开销，很明显是白白浪费掉的。 6 val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt") 7 rdd1.map(...) 8 val rdd2 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt") 9 rdd2.reduce(...) 10 11 // 正确的用法：对于一份数据执行多次算子操作时，只使用一个RDD。 12 // 这种写法很明显比上一种写法要好多了，因为我们对于同一份数据只创建了一个RDD，然后对这一个RDD执行了多次算子操作。 13 // 但是要注意到这里为止优化还没有结束，由于rdd1被执行了两次算子操作，第二次执行reduce操作的时候，还会再次从源头处重新计算一次rdd1的数据，因此还是会有重复计算的性能开销。 14 // 要彻底解决这个问题，必须结合“原则三：对多次使用的RDD进行持久化”，才能保证一个RDD被多次使用时只被计算一次。 15 val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt") 16 rdd1.map(...) 17 rdd1.reduce(...)

[制代码](javascript:void(0);)

### 原则二：尽可能复用同一个RDD

除了要避免在开发过程中对一份完全相同的数据创建多个RDD之外，在对不同的数据执行算子操作时还要尽可能地复用一个RDD。比如说，有一个RDD的数据格式是key-value类型的，另一个是单value类型的，这两个RDD的value数据是完全一样的。那么此时我们可以只使用key-value类型的那个RDD，因为其中已经包含了另一个的数据。对于类似这种多个RDD的数据有重叠或者包含的情况，我们应该尽量复用一个RDD，这样可以尽可能地减少RDD的数量，从而尽可能减少算子执行的次数。

一个简单的例子

[制代码](javascript:void(0);)

1 // 错误的做法。 2 3 // 有一个<Long, String>格式的RDD，即rdd1。 4 // 接着由于业务需要，对rdd1执行了一个map操作，创建了一个rdd2，而rdd2中的数据仅仅是rdd1中的value值而已，也就是说，rdd2是rdd1的子集。 5 JavaPairRDD<Long, String> rdd1 = ... 6 JavaRDD<String> rdd2 = rdd1.map(...) 7 8 // 分别对rdd1和rdd2执行了不同的算子操作。 9 rdd1.reduceByKey(...) 10 rdd2.map(...) 11 12 // 正确的做法。 13 14 // 上面这个case中，其实rdd1和rdd2的区别无非就是数据格式不同而已，rdd2的数据完全就是rdd1的子集而已，却创建了两个rdd，并对两个rdd都执行了一次算子操作。 15 // 此时会因为对rdd1执行map算子来创建rdd2，而多执行一次算子操作，进而增加性能开销。 16 17 // 其实在这种情况下完全可以复用同一个RDD。 18 // 我们可以使用rdd1，既做reduceByKey操作，也做map操作。 19 // 在进行第二个map操作时，只使用每个数据的tuple.\_2，也就是rdd1中的value值，即可。 20 JavaPairRDD<Long, String> rdd1 = ... 21 rdd1.reduceByKey(...) 22 rdd1.map(tuple.\_2...) 23 24 // 第二种方式相较于第一种方式而言，很明显减少了一次rdd2的计算开销。 25 // 但是到这里为止，优化还没有结束，对rdd1我们还是执行了两次算子操作，rdd1实际上还是会被计算两次。 26 // 因此还需要配合“原则三：对多次使用的RDD进行持久化”进行使用，才能保证一个RDD被多次使用时只被计算一次。

[制代码](javascript:void(0);)

### 原则三：对多次使用的RDD进行持久化

当你在Spark代码中多次对一个RDD做了算子操作后，恭喜，你已经实现Spark作业第一步的优化了，也就是尽可能复用RDD。此时就该在这个基础之上，进行第二步优化了，也就是要保证对一个RDD执行多次算子操作时，这个RDD本身仅仅被计算一次。

Spark中对于一个RDD执行多次算子的默认原理是这样的：每次你对一个RDD执行一个算子操作时，都会重新从源头处计算一遍，计算出那个RDD来，然后再对这个RDD执行你的算子操作。这种方式的性能是很差的。

因此对于这种情况，我们的建议是：对多次使用的RDD进行持久化。此时Spark就会根据你的持久化策略，将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不会从源头处重新计算一遍这个RDD，再执行算子操作。

#### 对多次使用的RDD进行持久化的代码示例

[制代码](javascript:void(0);)

1 // 如果要对一个RDD进行持久化，只要对这个RDD调用cache()和persist()即可。 2 3 // 正确的做法。 4 // cache()方法表示：使用非序列化的方式将RDD中的数据全部尝试持久化到内存中。 5 // 此时再对rdd1执行两次算子操作时，只有在第一次执行map算子时，才会将这个rdd1从源头处计算一次。 6 // 第二次执行reduce算子时，就会直接从内存中提取数据进行计算，不会重复计算一个rdd。 7 val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").cache() 8 rdd1.map(...) 9 rdd1.reduce(...) 10 11 // persist()方法表示：手动选择持久化级别，并使用指定的方式进行持久化。 12 // 比如说，StorageLevel.MEMORY\_AND\_DISK\_SER表示，内存充足时优先持久化到内存中，内存不充足时持久化到磁盘文件中。 13 // 而且其中的\_SER后缀表示，使用序列化的方式来保存RDD数据，此时RDD中的每个partition都会序列化成一个大的字节数组，然后再持久化到内存或磁盘中。 14 // 序列化的方式可以减少持久化的数据对内存/磁盘的占用量，进而避免内存被持久化数据占用过多，从而发生频繁GC。 15 val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").persist(StorageLevel.MEMORY\_AND\_DISK\_SER) 16 rdd1.map(...) 17 rdd1.reduce(...)

[制代码](javascript:void(0);)

对于persist()方法而言，我们可以根据不同的业务场景选择不同的持久化级别。

#### Spark的持久化级别

| **持久化级别** | **含义解释** |
| --- | --- |
| MEMORY\_ONLY | 使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。 |
| MEMORY\_AND\_DISK | 使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。 |
| MEMORY\_ONLY\_SER | 基本含义同MEMORY\_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。 |
| MEMORY\_AND\_DISK\_SER | 基本含义同MEMORY\_AND\_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。 |
| DISK\_ONLY | 使用未序列化的Java对象格式，将数据全部写入磁盘文件中。 |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, 等等. | 对于上述任意一种持久化策略，如果加上后缀\_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。 |

#### 如何选择一种最合适的持久化策略

* 默认情况下，性能最高的当然是MEMORY\_ONLY，但前提是你的内存必须足够足够大，可以绰绰有余地存放下整个RDD的所有数据。因为不进行序列化与反序列化操作，就避免了这部分的性能开销；对这个RDD的后续算子操作，都是基于纯内存中的数据的操作，不需要从磁盘文件中读取数据，性能也很高；而且不需要复制一份数据副本，并远程传送到其他节点上。但是这里必须要注意的是，在实际的生产环境中，恐怕能够直接用这种策略的场景还是有限的，如果RDD中数据比较多时（比如几十亿），直接用这种持久化级别，会导致JVM的OOM内存溢出异常。
* 如果使用MEMORY\_ONLY级别时发生了内存溢出，那么建议尝试使用MEMORY\_ONLY\_SER级别。该级别会将RDD数据序列化后再保存在内存中，此时每个partition仅仅是一个字节数组而已，大大减少了对象数量，并降低了内存占用。这种级别比MEMORY\_ONLY多出来的性能开销，主要就是序列化与反序列化的开销。但是后续算子可以基于纯内存进行操作，因此性能总体还是比较高的。此外，可能发生的问题同上，如果RDD中的数据量过多的话，还是可能会导致OOM内存溢出的异常。
* 如果纯内存的级别都无法使用，那么建议使用MEMORY\_AND\_DISK\_SER策略，而不是MEMORY\_AND\_DISK策略。因为既然到了这一步，就说明RDD的数据量很大，内存无法完全放下。序列化后的数据比较少，可以节省内存和磁盘的空间开销。同时该策略会优先尽量尝试将数据缓存在内存中，内存缓存不下才会写入磁盘。
* 通常不建议使用DISK\_ONLY和后缀为\_2的级别：因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新计算一次所有RDD。后缀为\_2的级别，必须将所有数据都复制一份副本，并发送到其他节点上，数据复制以及网络传输会导致较大的性能开销，除非是要求作业的高可用性，否则不建议使用。

### 原则四：尽量避免使用shuffle类算子

如果有可能的话，要尽量避免使用shuffle类算子。因为Spark作业运行过程中，最消耗性能的地方就是shuffle过程。shuffle过程，简单来说，就是将分布在集群中多个节点上的同一个key，拉取到同一个节点上，进行聚合或join等操作。比如reduceByKey、join等算子，都会触发shuffle操作。

shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的主要原因。

因此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。

#### Broadcast与map进行join代码示例

[制代码](javascript:void(0);)

1 // 传统的join操作会导致shuffle操作。 2 // 因为两个RDD中，相同的key都需要通过网络拉取到一个节点上，由一个task进行join操作。 3 val rdd3 = rdd1.join(rdd2) 4 5 // Broadcast+map的join操作，不会导致shuffle操作。 6 // 使用Broadcast将一个数据量较小的RDD作为广播变量。 7 val rdd2Data = rdd2.collect() 8 val rdd2DataBroadcast = sc.broadcast(rdd2Data) 9 10 // 在rdd1.map算子中，可以从rdd2DataBroadcast中，获取rdd2的所有数据。 11 // 然后进行遍历，如果发现rdd2中某条数据的key与rdd1的当前数据的key是相同的，那么就判定可以进行join。 12 // 此时就可以根据自己需要的方式，将rdd1当前数据与rdd2中可以连接的数据，拼接在一起（String或Tuple）。 13 val rdd3 = rdd1.map(rdd2DataBroadcast...) 14 15 // 注意，以上操作，建议仅仅在rdd2的数据量比较少（比如几百M，或者一两G）的情况下使用。 16 // 因为每个Executor的内存中，都会驻留一份rdd2的全量数据。

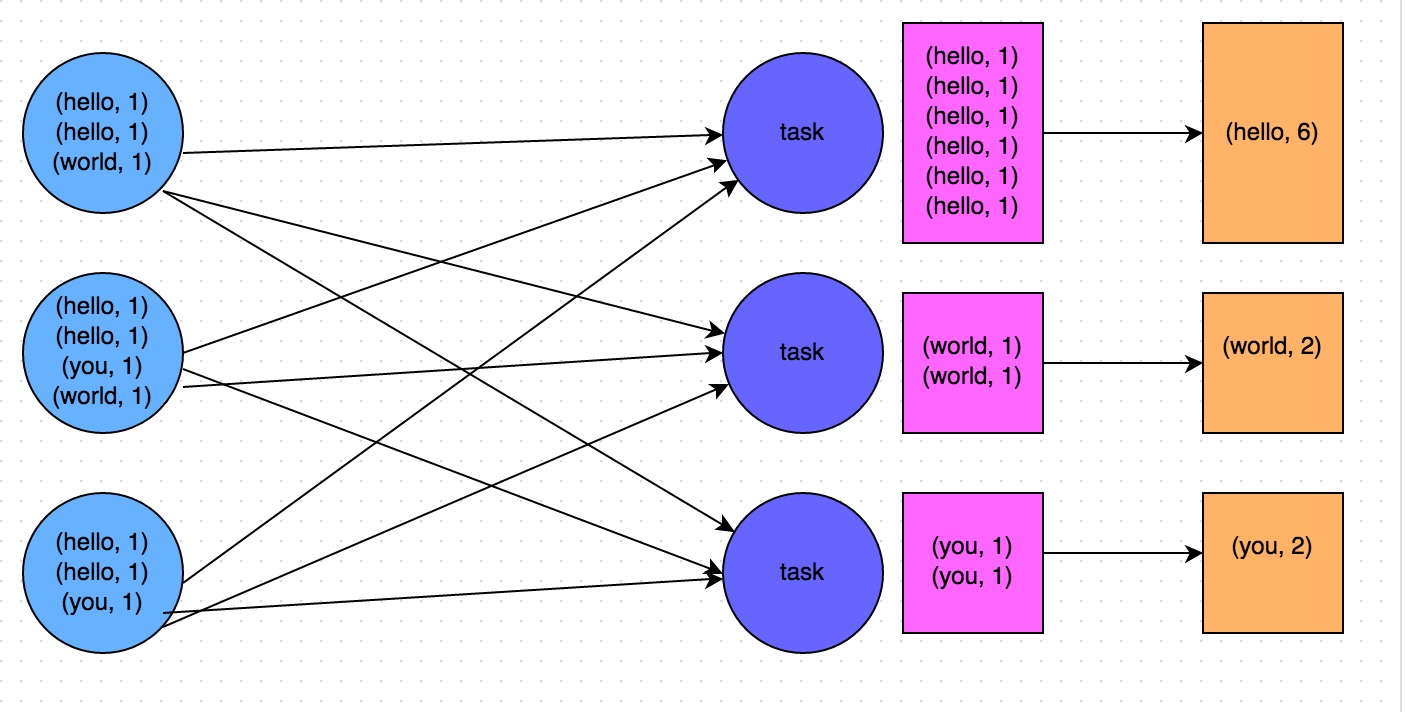
[制代码](javascript:void(0);)

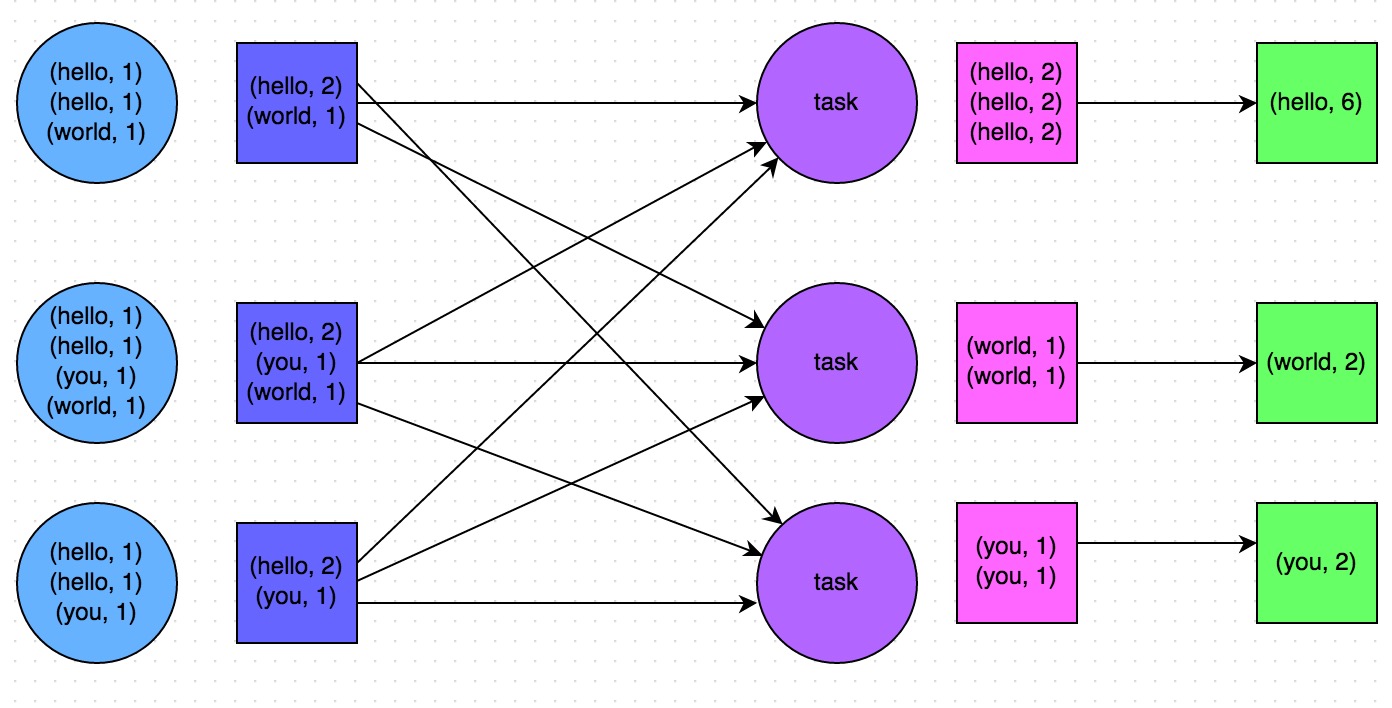
### 原则五：使用map-side预聚合的shuffle操作

如果因为业务需要，一定要使用shuffle操作，无法用map类的算子来替代，那么尽量使用可以map-side预聚合的算子。

所谓的map-side预聚合，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MapReduce中的本地combiner。map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘IO以及网络传输开销。通常来说，在可能的情况下，建议使用reduceByKey或者aggregateByKey算子来替代掉groupByKey算子。因为reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。而groupByKey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

比如如下两幅图，就是典型的例子，分别基于reduceByKey和groupByKey进行单词计数。其中第一张图是groupByKey的原理图，可以看到，没有进行任何本地聚合时，所有数据都会在集群节点之间传输；第二张图是reduceByKey的原理图，可以看到，每个节点本地的相同key数据，都进行了预聚合，然后才传输到其他节点上进行全局聚合。

。



### 原则六：使用高性能的算子

除了shuffle相关的算子有优化原则之外，其他的算子也都有着相应的优化原则。

#### 使用reduceByKey/aggregateByKey替代groupByKey

详情见“原则五：使用map-side预聚合的shuffle操作”。

#### 使用mapPartitions替代普通map

mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。但是有的时候，使用mapPartitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

#### 使用foreachPartitions替代foreach

原理类似于“使用mapPartitions替代map”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数据。在实践中发现，foreachPartitions类的算子，对性能的提升还是很有帮助的。比如在foreach函数中，将RDD中所有数据写MySQL，那么如果是普通的foreach算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用foreachPartitions算子一次性处理一个partition的数据，那么对于每个partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。实践中发现，对于1万条左右的数据量写MySQL，性能可以提升30%以上。

#### 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中较多数据后（比如30%以上的数据），建议使用coalesce算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为filter之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用coalesce减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。

#### 使用repartitionAndSortWithinPartitions替代repartition与sort类操作

repartitionAndSortWithinPartitions是Spark官网推荐的一个算子，官方建议，如果需要在repartition重分区之后，还要进行排序，建议直接使用repartitionAndSortWithinPartitions算子。因为该算子可以一边进行重分区的shuffle操作，一边进行排序。shuffle与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高的。

### 原则七：广播大变量

有时在开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如100M以上的大集合），那么此时就应该使用Spark的广播（Broadcast）功能来提升性能。

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，都会极大地影响性能。

因此对于上述情况，如果使用的外部变量比较大，建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对Executor内存的占用开销，降低GC的频率。

#### 广播大变量的代码示例

[制代码](javascript:void(0);)

1 // 以下代码在算子函数中，使用了外部的变量。 2 // 此时没有做任何特殊操作，每个task都会有一份list1的副本。 3 val list1 = ... 4 rdd1.map(list1...) 5 6 // 以下代码将list1封装成了Broadcast类型的广播变量。 7 // 在算子函数中，使用广播变量时，首先会判断当前task所在Executor内存中，是否有变量副本。 8 // 如果有则直接使用；如果没有则从Driver或者其他Executor节点上远程拉取一份放到本地Executor内存中。 9 // 每个Executor内存中，就只会驻留一份广播变量副本。 10 val list1 = ... 11 val list1Broadcast = sc.broadcast(list1) 12 rdd1.map(list1Broadcast...)

[制代码](javascript:void(0);)

### 原则八：使用Kryo优化序列化性能

在Spark中，主要有三个地方涉及到了序列化：

* 在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输（见“原则七：广播大变量”中的讲解）。
* 将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现Serializable接口。
* 使用可序列化的持久化策略时（比如MEMORY\_ONLY\_SER），Spark会将RDD中的每个partition都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。

以下是使用Kryo的代码示例，我们只要设置序列化类，再注册要序列化的自定义类型即可（比如算子函数中使用到的外部变量类型、作为RDD泛型类型的自定义类型等）：

1 // 创建SparkConf对象。 2 val conf = new SparkConf().setMaster(...).setAppName(...) 3 // 设置序列化器为KryoSerializer。 4 conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer") 5 // 注册要序列化的自定义类型。 6 conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))

### 原则九：优化数据结构

Java中，有三种类型比较耗费内存：

* 对象，每个Java对象都有对象头、引用等额外的信息，因此比较占用内存空间。
* 字符串，每个字符串内部都有一个字符数组以及长度等额外信息。
* 集合类型，比如HashMap、LinkedList等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如Map.Entry。

因此Spark官方建议，在Spark编码实现中，特别是对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型（比如Int、Long）替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低GC频率，提升性能。

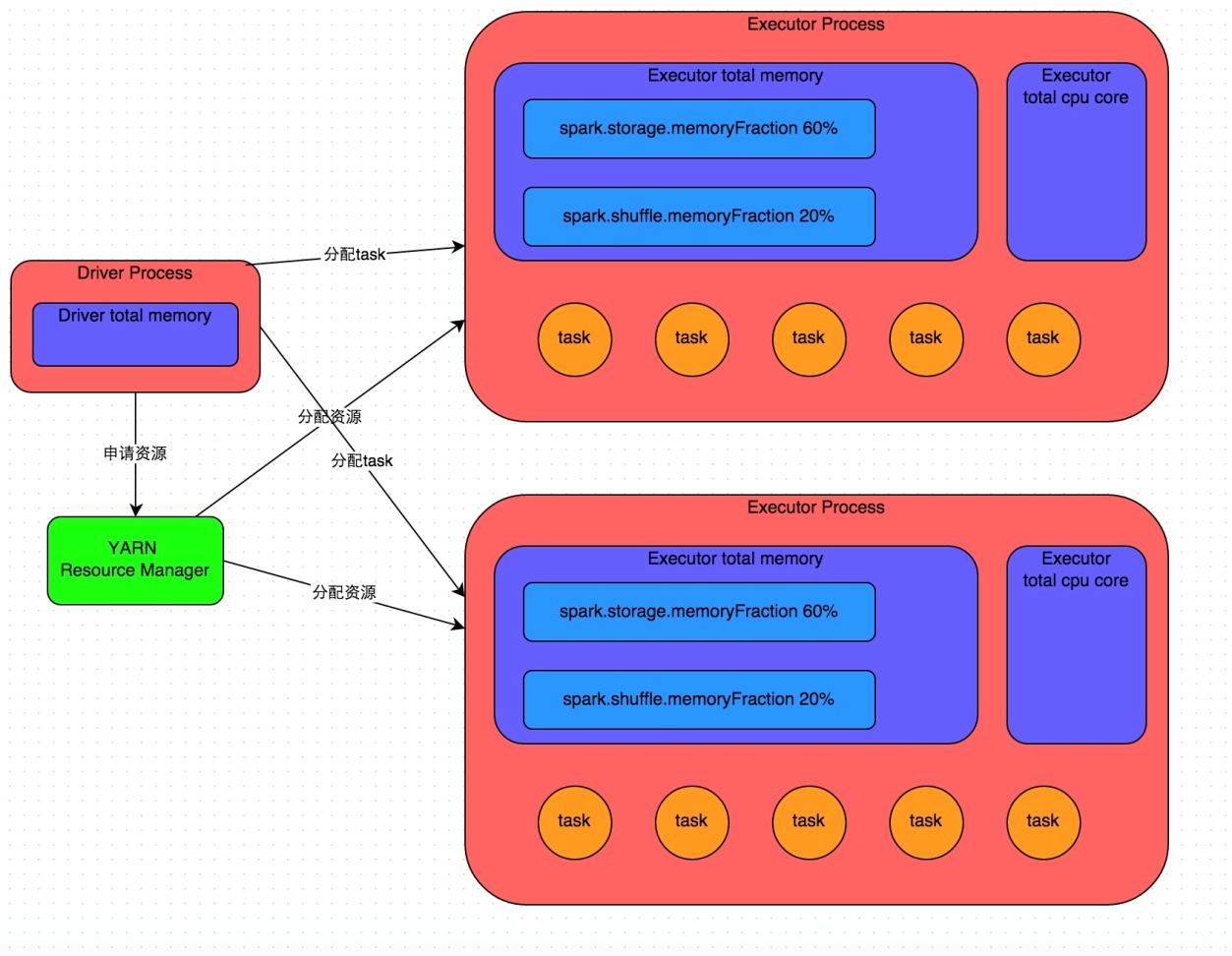
但是在笔者的编码实践中发现，要做到该原则其实并不容易。因为我们同时要考虑到代码的可维护性，如果一个代码中，完全没有任何对象抽象，全部是字符串拼接的方式，那么对于后续的代码维护和修改，无疑是一场巨大的灾难。同理，如果所有操作都基于数组实现，而不使用HashMap、LinkedList等集合类型，那么对于我们的编码难度以及代码可维护性，也是一个极大的挑战。因此笔者建议，在可能以及合适的情况下，使用占用内存较少的数据结构，但是前提是要保证代码的可维护性。

## 资源调优

### 调优概述

在开发完Spark作业之后，就该为作业配置合适的资源了。Spark的资源参数，基本都可以在spark-submit命令中作为参数设置。很多Spark初学者，通常不知道该设置哪些必要的参数，以及如何设置这些参数，最后就只能胡乱设置，甚至压根儿不设置。资源参数设置的不合理，可能会导致没有充分利用集群资源，作业运行会极其缓慢；或者设置的资源过大，队列没有足够的资源来提供，进而导致各种异常。总之，无论是哪种情况，都会导致Spark作业的运行效率低下，甚至根本无法运行。因此我们必须对Spark作业的资源使用原理有一个清晰的认识，并知道在Spark作业运行过程中，有哪些资源参数是可以设置的，以及如何设置合适的参数值。

Spark作业基本运行原理



详细原理见上图。我们使用spark-submit提交一个Spark作业之后，这个作业就会启动一个对应的Driver进程。根据你使用的部署模式（deploy-mode）不同，Driver进程可能在本地启动，也可能在集群中某个工作节点上启动。Driver进程本身会根据我们设置的参数，占有一定数量的内存和CPU core。而Driver进程要做的第一件事情，就是向集群管理器（可以是Spark Standalone集群，也可以是其他的资源管理集群，美团•大众点评使用的是YARN作为资源管理集群）申请运行Spark作业需要使用的资源，这里的资源指的就是Executor进程。YARN集群管理器会根据我们为Spark作业设置的资源参数，在各个工作节点上，启动一定数量的Executor进程，每个Executor进程都占有一定数量的内存和CPU core。

在申请到了作业执行所需的资源之后，Driver进程就会开始调度和执行我们编写的作业代码了。Driver进程会将我们编写的Spark作业代码分拆为多个stage，每个stage执行一部分代码片段，并为每个stage创建一批task，然后将这些task分配到各个Executor进程中执行。task是最小的计算单元，负责执行一模一样的计算逻辑（也就是我们自己编写的某个代码片段），只是每个task处理的数据不同而已。一个stage的所有task都执行完毕之后，会在各个节点本地的磁盘文件中写入计算中间结果，然后Driver就会调度运行下一个stage。下一个stage的task的输入数据就是上一个stage输出的中间结果。如此循环往复，直到将我们自己编写的代码逻辑全部执行完，并且计算完所有的数据，得到我们想要的结果为止。

Spark是根据shuffle类算子来进行stage的划分。如果我们的代码中执行了某个shuffle类算子（比如reduceByKey、join等），那么就会在该算子处，划分出一个stage界限来。可以大致理解为，shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作（比如reduceByKey()算子接收的函数）。这个过程就是shuffle。

当我们在代码中执行了cache/persist等持久化操作时，根据我们选择的持久化级别的不同，每个task计算出来的数据也会保存到Executor进程的内存或者所在节点的磁盘文件中。

因此Executor的内存主要分为三块：第一块是让task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；第二块是让task通过shuffle过程拉取了上一个stage的task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是让RDD持久化时使用，默认占Executor总内存的60%。

task的执行速度是跟每个Executor进程的CPU core数量有直接关系的。一个CPU core同一时间只能执行一个线程。而每个Executor进程上分配到的多个task，都是以每个task一条线程的方式，多线程并发运行的。如果CPU core数量比较充足，而且分配到的task数量比较合理，那么通常来说，可以比较快速和高效地执行完这些task线程。

以上就是Spark作业的基本运行原理的说明，大家可以结合上图来理解。理解作业基本原理，是我们进行资源参数调优的基本前提。

### 资源参数调优

了解完了Spark作业运行的基本原理之后，对资源相关的参数就容易理解了。所谓的Spark资源参数调优，其实主要就是对Spark运行过程中各个使用资源的地方，通过调节各种参数，来优化资源使用的效率，从而提升Spark作业的执行性能。以下参数就是Spark中主要的资源参数，每个参数都对应着作业运行原理中的某个部分，我们同时也给出了一个调优的参考值。

#### num-executors

* 参数说明：该参数用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。这个参数非常之重要，如果不设置的话，默认只会给你启动少量的Executor进程，此时你的Spark作业的运行速度是非常慢的。
* 参数调优建议：每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

#### executor-memory

* 参数说明：该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联。
* 参数调优建议：每个Executor进程的内存设置4G~8G较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，num-executors乘以executor-memory，是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，导致别的同学的作业无法运行。

#### executor-cores

* 参数说明：该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。
* 参数调优建议：Executor的CPU core数量设置为2~4个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大CPU core限制是多少，再依据设置的Executor数量，来决定每个Executor进程可以分配到几个CPU core。同样建议，如果是跟他人共享这个队列，那么num-executors \* executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适，也是避免影响其他同学的作业运行。

#### driver-memory

* 参数说明：该参数用于设置Driver进程的内存。
* 参数调优建议：Driver的内存通常来说不设置，或者设置1G左右应该就够了。唯一需要注意的一点是，如果需要使用collect算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存足够大，否则会出现OOM内存溢出的问题。

#### spark.default.parallelism

* 参数说明：该参数用于设置每个stage的默认task数量。这个参数极为重要，如果不设置可能会直接影响你的Spark作业性能。
* 参数调优建议：Spark作业的默认task数量为500~1000个较为合适。很多同学常犯的一个错误就是不去设置这个参数，那么此时就会导致Spark自己根据底层HDFS的block数量来设置task的数量，默认是一个HDFS block对应一个task。通常来说，Spark默认设置的数量是偏少的（比如就几十个task），如果task数量偏少的话，就会导致你前面设置好的Executor的参数都前功尽弃。试想一下，无论你的Executor进程有多少个，内存和CPU有多大，但是task只有1个或者10个，那么90%的Executor进程可能根本就没有task执行，也就是白白浪费了资源！因此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，比如Executor的总CPU core数量为300个，那么设置1000个task是可以的，此时可以充分地利用Spark集群的资源。

#### spark.storage.memoryFraction

* 参数说明：该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。
* 参数调优建议：如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark web ui可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

#### spark.shuffle.memoryFraction

* 参数说明：该参数用于设置shuffle过程中一个task拉取到上个stage的task的输出后，进行聚合操作时能够使用的Executor内存的比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超出了这个20%的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。
* 参数调优建议：如果Spark作业中的RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

资源参数的调优，没有一个固定的值，需要同学们根据自己的实际情况（包括Spark作业中的shuffle操作数量、RDD持久化操作数量以及spark web ui中显示的作业gc情况），同时参考本篇文章中给出的原理以及调优建议，合理地设置上述参数。

### 资源参数参考示例

以下是一份spark-submit命令的示例，大家可以参考一下，并根据自己的实际情况进行调节：

[制代码](javascript:void(0);)

1 ./bin/spark-submit \ 2 --master yarn-cluster \ 3 --num-executors 100 \ 4 --executor-memory 6G \ 5 --executor-cores 4 \ 6 --driver-memory 1G \ 7 --conf spark.default.parallelism=1000 \ 8 --conf spark.storage.memoryFraction=0.5 \ 9 --conf spark.shuffle.memoryFraction=0.3 \

[制代码](javascript:void(0);)

## 数据倾斜调优

### 调优概述

有的时候，我们可能会遇到大数据计算中一个最棘手的问题——数据倾斜，此时Spark作业的性能会比期望差很多。数据倾斜调优，就是使用各种技术方案解决不同类型的数据倾斜问题，以保证Spark作业的性能。

### 数据倾斜发生时的现象

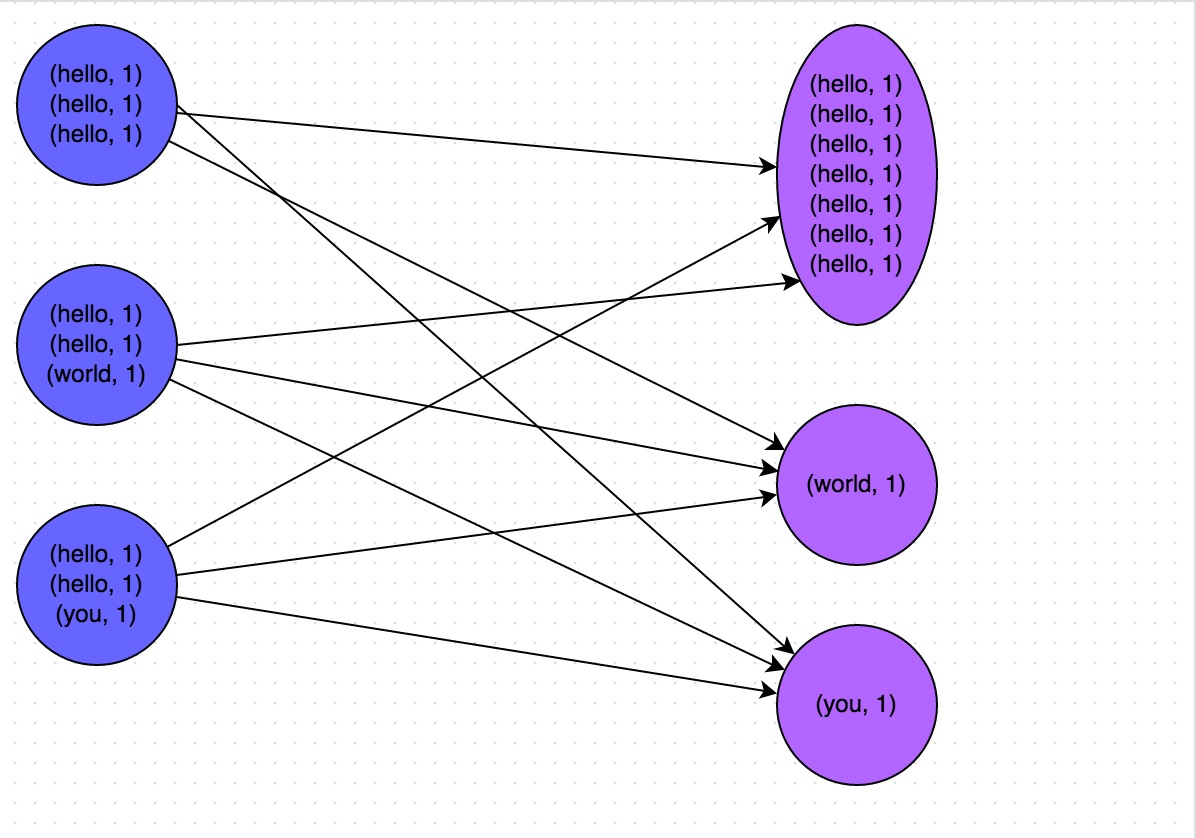
* 绝大多数task执行得都非常快，但个别task执行极慢。比如，总共有1000个task，997个task都在1分钟之内执行完了，但是剩余两三个task却要一两个小时。这种情况很常见。
* 原本能够正常执行的Spark作业，某天突然报出OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

### 数据倾斜发生的原理

数据倾斜的原理很简单：在进行shuffle的时候，必须将各个节点上相同的key拉取到某个节点上的一个task来进行处理，比如按照key进行聚合或join等操作。此时如果某个key对应的数据量特别大的话，就会发生数据倾斜。比如大部分key对应10条数据，但是个别key却对应了100万条数据，那么大部分task可能就只会分配到10条数据，然后1秒钟就运行完了；但是个别task可能分配到了100万数据，要运行一两个小时。因此，整个Spark作业的运行进度是由运行时间最长的那个task决定的。

因此出现数据倾斜的时候，Spark作业看起来会运行得非常缓慢，甚至可能因为某个task处理的数据量过大导致内存溢出。

下图就是一个很清晰的例子：hello这个key，在三个节点上对应了总共7条数据，这些数据都会被拉取到同一个task中进行处理；而world和you这两个key分别才对应1条数据，所以另外两个task只要分别处理1条数据即可。此时第一个task的运行时间可能是另外两个task的7倍，而整个stage的运行速度也由运行最慢的那个task所决定。



### 如何定位导致数据倾斜的代码

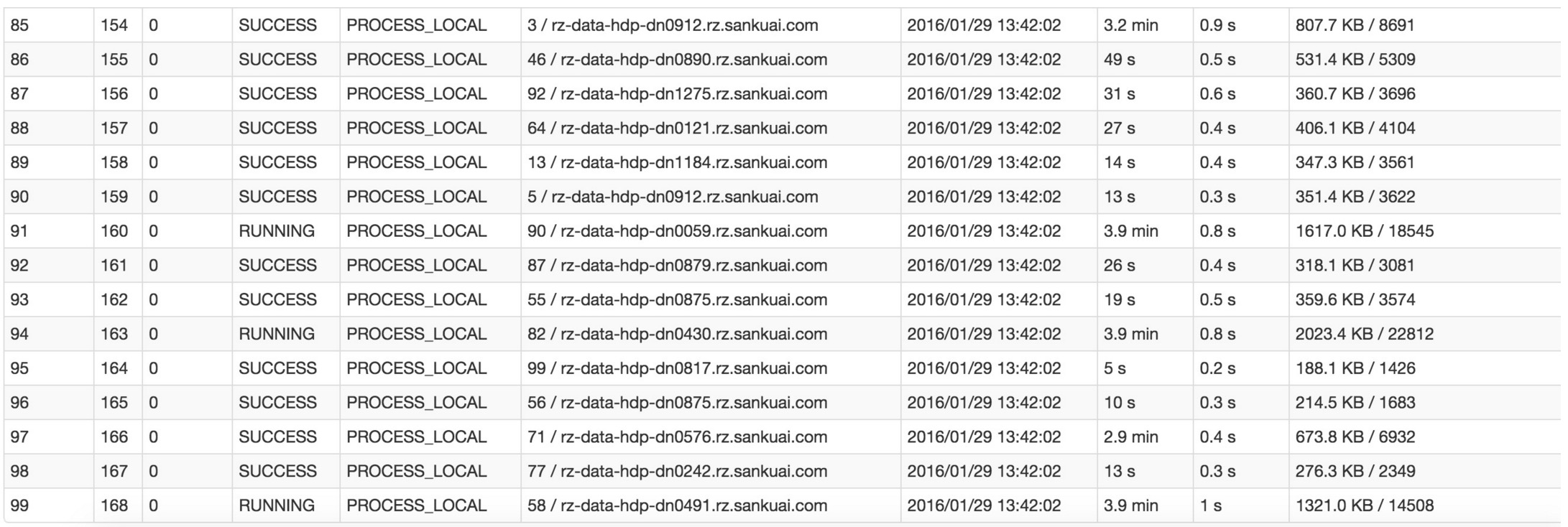
数据倾斜只会发生在shuffle过程中。这里给大家罗列一些常用的并且可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

#### 某个task执行特别慢的情况

首先要看的，就是数据倾斜发生在第几个stage中。

如果是用yarn-client模式提交，那么本地是直接可以看到log的，可以在log中找到当前运行到了第几个stage；如果是用yarn-cluster模式提交，则可以通过Spark Web UI来查看当前运行到了第几个stage。此外，无论是使用yarn-client模式还是yarn-cluster模式，我们都可以在Spark Web UI上深入看一下当前这个stage各个task分配的数据量，从而进一步确定是不是task分配的数据不均匀导致了数据倾斜。

比如下图中，倒数第三列显示了每个task的运行时间。明显可以看到，有的task运行特别快，只需要几秒钟就可以运行完；而有的task运行特别慢，需要几分钟才能运行完，此时单从运行时间上看就已经能够确定发生数据倾斜了。此外，倒数第一列显示了每个task处理的数据量，明显可以看到，运行时间特别短的task只需要处理几百KB的数据即可，而运行时间特别长的task需要处理几千KB的数据，处理的数据量差了10倍。此时更加能够确定是发生了数据倾斜。



知道数据倾斜发生在哪一个stage之后，接着我们就需要根据stage划分原理，推算出来发生倾斜的那个stage对应代码中的哪一部分，这部分代码中肯定会有一个shuffle类算子。精准推算stage与代码的对应关系，需要对Spark的源码有深入的理解，这里我们可以介绍一个相对简单实用的推算方法：只要看到Spark代码中出现了一个shuffle类算子或者是Spark SQL的SQL语句中出现了会导致shuffle的语句（比如group by语句），那么就可以判定，以那个地方为界限划分出了前后两个stage。

这里我们就以Spark最基础的入门程序——单词计数来举例，如何用最简单的方法大致推算出一个stage对应的代码。如下示例，在整个代码中，只有一个reduceByKey是会发生shuffle的算子，因此就可以认为，以这个算子为界限，会划分出前后两个stage。

* stage0，主要是执行从textFile到map操作，以及执行shuffle write操作。shuffle write操作，我们可以简单理解为对pairs RDD中的数据进行分区操作，每个task处理的数据中，相同的key会写入同一个磁盘文件内。
* stage1，主要是执行从reduceByKey到collect操作，stage1的各个task一开始运行，就会首先执行shuffle read操作。执行shuffle read操作的task，会从stage0的各个task所在节点拉取属于自己处理的那些key，然后对同一个key进行全局性的聚合或join等操作，在这里就是对key的value值进行累加。stage1在执行完reduceByKey算子之后，就计算出了最终的wordCounts RDD，然后会执行collect算子，将所有数据拉取到Driver上，供我们遍历和打印输出。

[制代码](javascript:void(0);)

1 val conf = new SparkConf() 2 val sc = new SparkContext(conf) 3 4 val lines = sc.textFile("hdfs://...") 5 val words = lines.flatMap(\_.split(" ")) 6 val pairs = words.map((\_, 1)) 7 val wordCounts = pairs.reduceByKey(\_ + \_) 8 9 wordCounts.collect().foreach(println(\_))

[制代码](javascript:void(0);)

通过对单词计数程序的分析，希望能够让大家了解最基本的stage划分的原理，以及stage划分后shuffle操作是如何在两个stage的边界处执行的。然后我们就知道如何快速定位出发生数据倾斜的stage对应代码的哪一个部分了。比如我们在Spark Web UI或者本地log中发现，stage1的某几个task执行得特别慢，判定stage1出现了数据倾斜，那么就可以回到代码中定位出stage1主要包括了reduceByKey这个shuffle类算子，此时基本就可以确定是由educeByKey算子导致的数据倾斜问题。比如某个单词出现了100万次，其他单词才出现10次，那么stage1的某个task就要处理100万数据，整个stage的速度就会被这个task拖慢。

#### 某个task莫名其妙内存溢出的情况

这种情况下去定位出问题的代码就比较容易了。我们建议直接看yarn-client模式下本地log的异常栈，或者是通过YARN查看yarn-cluster模式下的log中的异常栈。一般来说，通过异常栈信息就可以定位到你的代码中哪一行发生了内存溢出。然后在那行代码附近找找，一般也会有shuffle类算子，此时很可能就是这个算子导致了数据倾斜。

但是大家要注意的是，不能单纯靠偶然的内存溢出就判定发生了数据倾斜。因为自己编写的代码的bug，以及偶然出现的数据异常，也可能会导致内存溢出。因此还是要按照上面所讲的方法，通过Spark Web UI查看报错的那个stage的各个task的运行时间以及分配的数据量，才能确定是否是由于数据倾斜才导致了这次内存溢出。

### 查看导致数据倾斜的key的数据分布情况

知道了数据倾斜发生在哪里之后，通常需要分析一下那个执行了shuffle操作并且导致了数据倾斜的RDD/Hive表，查看一下其中key的分布情况。这主要是为之后选择哪一种技术方案提供依据。针对不同的key分布与不同的shuffle算子组合起来的各种情况，可能需要选择不同的技术方案来解决。

此时根据你执行操作的情况不同，可以有很多种查看key分布的方式：

1. 如果是Spark SQL中的group by、join语句导致的数据倾斜，那么就查询一下SQL中使用的表的key分布情况。
2. 如果是对Spark RDD执行shuffle算子导致的数据倾斜，那么可以在Spark作业中加入查看key分布的代码，比如RDD.countByKey()。然后对统计出来的各个key出现的次数，collect/take到客户端打印一下，就可以看到key的分布情况。

举例来说，对于上面所说的单词计数程序，如果确定了是stage1的reduceByKey算子导致了数据倾斜，那么就应该看看进行reduceByKey操作的RDD中的key分布情况，在这个例子中指的就是pairs RDD。如下示例，我们可以先对pairs采样10%的样本数据，然后使用countByKey算子统计出每个key出现的次数，最后在客户端遍历和打印样本数据中各个key的出现次数。

1 val sampledPairs = pairs.sample(false, 0.1) 2 val sampledWordCounts = sampledPairs.countByKey() 3 sampledWordCounts.foreach(println(\_))

### 数据倾斜的解决方案

#### 解决方案一：使用Hive ETL预处理数据

**方案适用场景：**导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且业务场景需要频繁使用Spark对Hive表执行某个分析操作，那么比较适合使用这种技术方案。

**方案实现思路：**此时可以评估一下，是否可以通过Hive来进行数据预处理（即通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join），然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。

**方案实现原理：**这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

**方案优点：**实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

**方案缺点：**治标不治本，Hive ETL中还是会发生数据倾斜。

**方案实践经验：**在一些Java系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验。

**项目实践经验：**在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过Java Web系统提交数据分析统计任务，后端通过Java提交Spark作业进行数据分析统计。要求Spark作业速度必须要快，尽量在10分钟以内，否则速度太慢，用户体验会很差。所以我们将有些Spark作业的shuffle操作提前到了Hive ETL中，从而让Spark直接使用预处理的Hive中间表，尽可能地减少Spark的shuffle操作，大幅度提升了性能，将部分作业的性能提升了6倍以上。

#### 解决方案二：过滤少数导致倾斜的key

**方案适用场景：**如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜。

**方案实现思路：**如果我们判断那少数几个数据量特别多的key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个key。比如，在Spark SQL中可以使用where子句过滤掉这些key或者在Spark Core中对RDD执行filter算子过滤掉这些key。如果需要每次作业执行时，动态判定哪些key的数据量最多然后再进行过滤，那么可以使用sample算子对RDD进行采样，然后计算出每个key的数量，取数据量最多的key过滤掉即可。

**方案实现原理：**将导致数据倾斜的key给过滤掉之后，这些key就不会参与计算了，自然不可能产生数据倾斜。

**方案优点：**实现简单，而且效果也很好，可以完全规避掉数据倾斜。

**方案缺点：**适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。

**方案实践经验：**在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉。

#### 解决方案三：提高shuffle操作的并行度

**方案适用场景：**如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。

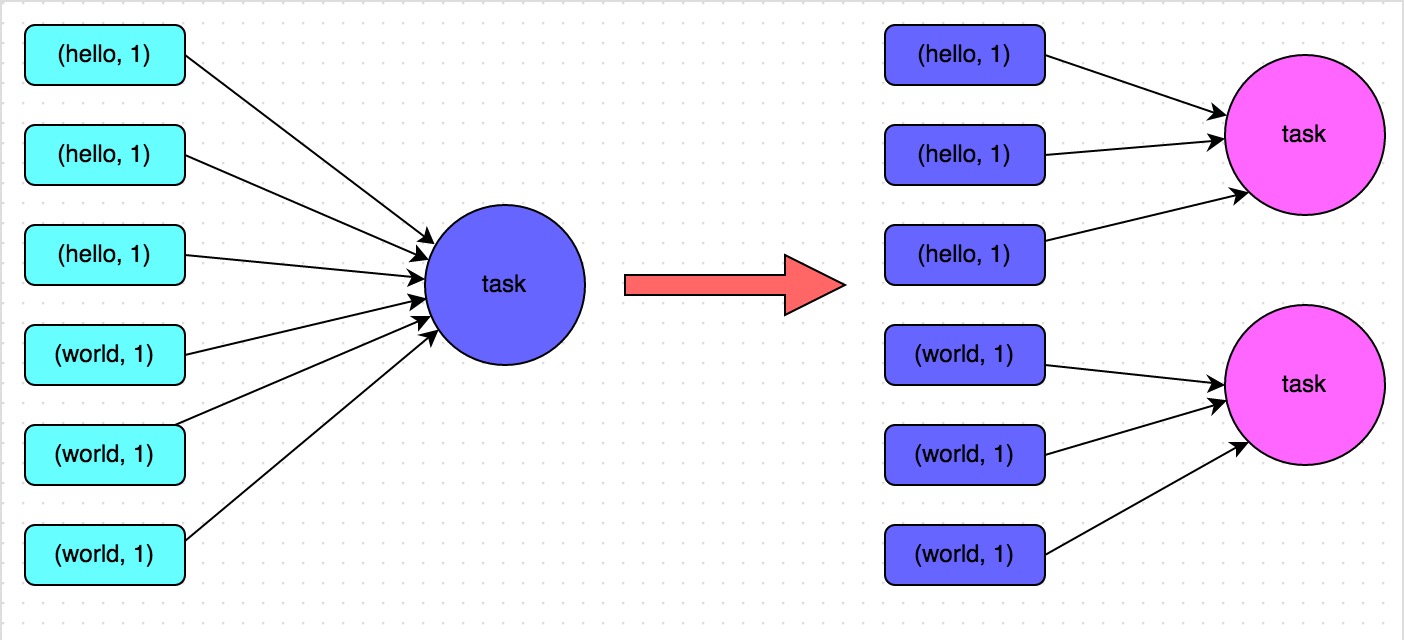
**方案实现思路：**在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小。

**方案实现原理：**增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。具体原理如下图所示。

**方案优点：**实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

**方案缺点：**只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

**方案实践经验：**该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用嘴简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。



#### 解决方案四：两阶段聚合（局部聚合+全局聚合）

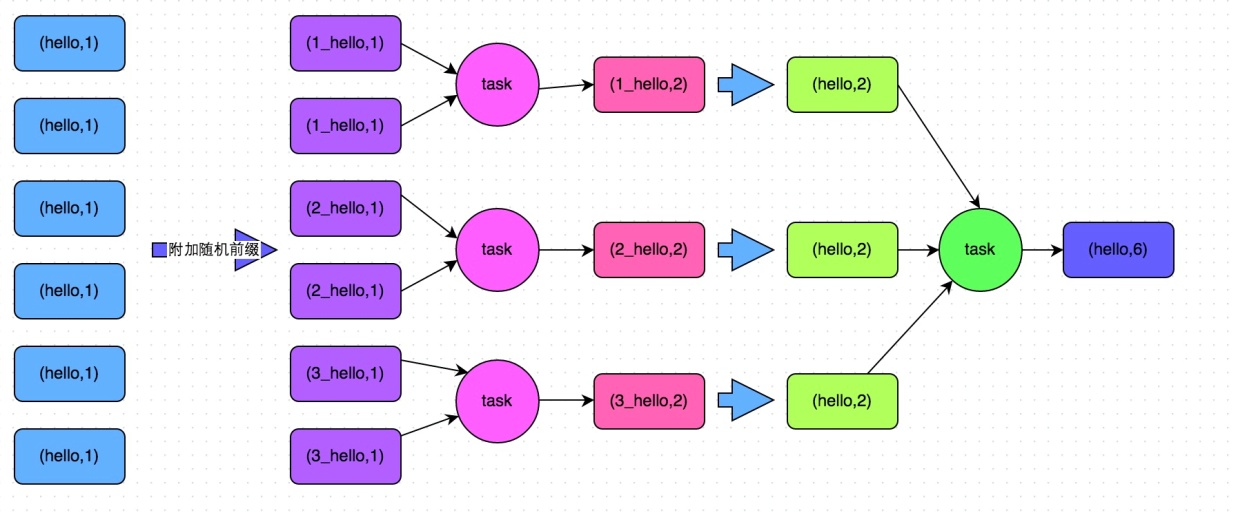
**方案适用场景：**对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案。

**方案实现思路：**这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1\_hello, 1) (1\_hello, 1) (2\_hello, 1) (2\_hello, 1)。接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1\_hello, 2) (2\_hello, 2)。然后将各个key的前缀给去掉，就会变成(hello,2)(hello,2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

**方案实现原理：**将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。

**方案优点：**对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上。

**方案缺点：**仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案。



[制代码](javascript:void(0);)

1 // 第一步，给RDD中的每个key都打上一个随机前缀。 2 JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair( 3 new PairFunction<Tuple2<Long,Long>, String, Long>() { 4 private static final long serialVersionUID = 1L; 5 @Override 6 public Tuple2<String, Long> call(Tuple2<Long, Long> tuple) 7 throws Exception { 8 Random random = new Random(); 9 int prefix = random.nextInt(10); 10 return new Tuple2<String, Long>(prefix + "\_" + tuple.\_1, tuple.\_2); 11 } 12 }); 13 14 // 第二步，对打上随机前缀的key进行局部聚合。 15 JavaPairRDD<String, Long> localAggrRdd = randomPrefixRdd.reduceByKey( 16 new Function2<Long, Long, Long>() { 17 private static final long serialVersionUID = 1L; 18 @Override 19 public Long call(Long v1, Long v2) throws Exception { 20 return v1 + v2; 21 } 22 }); 23 24 // 第三步，去除RDD中每个key的随机前缀。 25 JavaPairRDD<Long, Long> removedRandomPrefixRdd = localAggrRdd.mapToPair( 26 new PairFunction<Tuple2<String,Long>, Long, Long>() { 27 private static final long serialVersionUID = 1L; 28 @Override 29 public Tuple2<Long, Long> call(Tuple2<String, Long> tuple) 30 throws Exception { 31 long originalKey = Long.valueOf(tuple.\_1.split("\_")[1]); 32 return new Tuple2<Long, Long>(originalKey, tuple.\_2); 33 } 34 }); 35 36 // 第四步，对去除了随机前缀的RDD进行全局聚合。 37 JavaPairRDD<Long, Long> globalAggrRdd = removedRandomPrefixRdd.reduceByKey( 38 new Function2<Long, Long, Long>() { 39 private static final long serialVersionUID = 1L; 40 @Override 41 public Long call(Long v1, Long v2) throws Exception { 42 return v1 + v2; 43 } 44 });

[制代码](javascript:void(0);)

#### 解决方案五：将reduce join转为map join

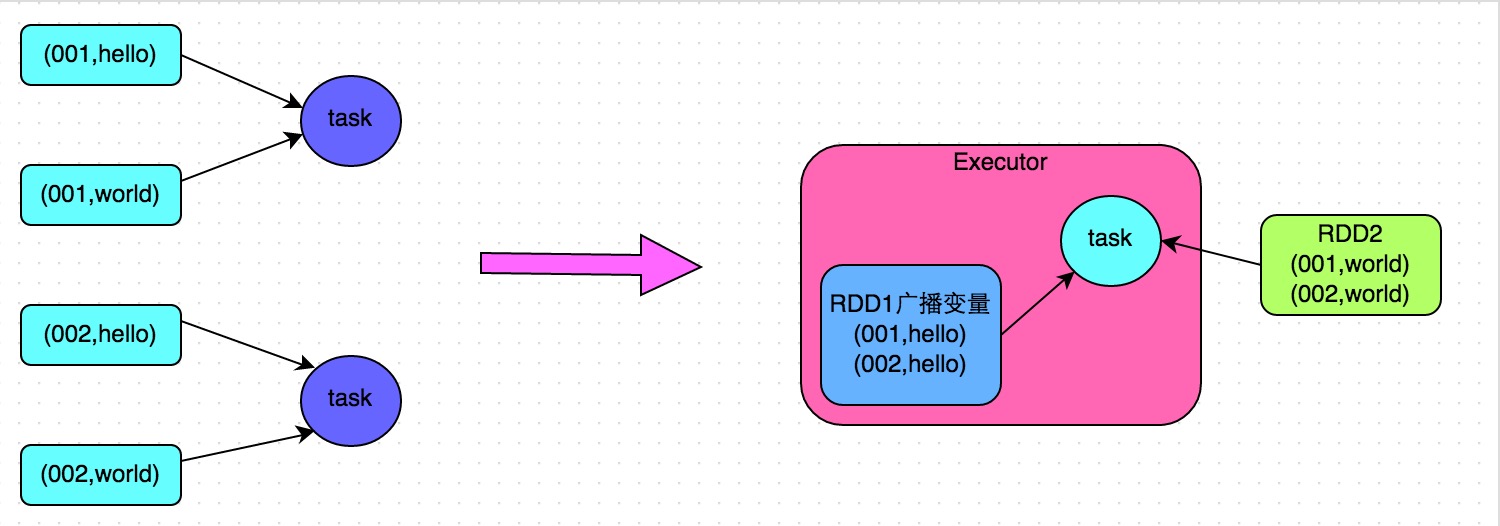
**方案适用场景：**在对RDD使用join类操作，或者是在Spark SQL中使用join语句时，而且join操作中的一个RDD或表的数据量比较小（比如几百M或者一两G），比较适用此方案。

**方案实现思路：**不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

**方案实现原理：**普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜。具体原理如下图所示。

**方案优点：**对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜。

**方案缺点：**适用场景较少，因为这个方案只适用于一个大表和一个小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver和每个Executor内存中都会驻留一份小RDD的全量数据。如果我们广播出去的RDD数据比较大，比如10G以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。



[制代码](javascript:void(0);)

1 // 首先将数据量比较小的RDD的数据，collect到Driver中来。 2 List<Tuple2<Long, Row>> rdd1Data = rdd1.collect() 3 // 然后使用Spark的广播功能，将小RDD的数据转换成广播变量，这样每个Executor就只有一份RDD的数据。 4 // 可以尽可能节省内存空间，并且减少网络传输性能开销。 5 final Broadcast<List<Tuple2<Long, Row>>> rdd1DataBroadcast = sc.broadcast(rdd1Data); 6 7 // 对另外一个RDD执行map类操作，而不再是join类操作。 8 JavaPairRDD<String, Tuple2<String, Row>> joinedRdd = rdd2.mapToPair( 9 new PairFunction<Tuple2<Long,String>, String, Tuple2<String, Row>>() { 10 private static final long serialVersionUID = 1L; 11 @Override 12 public Tuple2<String, Tuple2<String, Row>> call(Tuple2<Long, String> tuple) 13 throws Exception { 14 // 在算子函数中，通过广播变量，获取到本地Executor中的rdd1数据。 15 List<Tuple2<Long, Row>> rdd1Data = rdd1DataBroadcast.value(); 16 // 可以将rdd1的数据转换为一个Map，便于后面进行join操作。 17 Map<Long, Row> rdd1DataMap = new HashMap<Long, Row>(); 18 for(Tuple2<Long, Row> data : rdd1Data) { 19 rdd1DataMap.put(data.\_1, data.\_2); 20 } 21 // 获取当前RDD数据的key以及value。 22 String key = tuple.\_1; 23 String value = tuple.\_2; 24 // 从rdd1数据Map中，根据key获取到可以join到的数据。 25 Row rdd1Value = rdd1DataMap.get(key); 26 return new Tuple2<String, String>(key, new Tuple2<String, Row>(value, rdd1Value)); 27 } 28 }); 29 30 // 这里得提示一下。 31 // 上面的做法，仅仅适用于rdd1中的key没有重复，全部是唯一的场景。 32 // 如果rdd1中有多个相同的key，那么就得用flatMap类的操作，在进行join的时候不能用map，而是得遍历rdd1所有数据进行join。 33 // rdd2中每条数据都可能会返回多条join后的数据。

[制代码](javascript:void(0);)

#### 解决方案六：采样倾斜key并分拆join操作

**方案适用场景：**两个RDD/Hive表进行join的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个RDD/Hive表中的key分布情况。如果出现数据倾斜，是因为其中某一个RDD/Hive表中的少数几个key的数据量过大，而另一个RDD/Hive表中的所有key都分布比较均匀，那么采用这个解决方案是比较合适的。

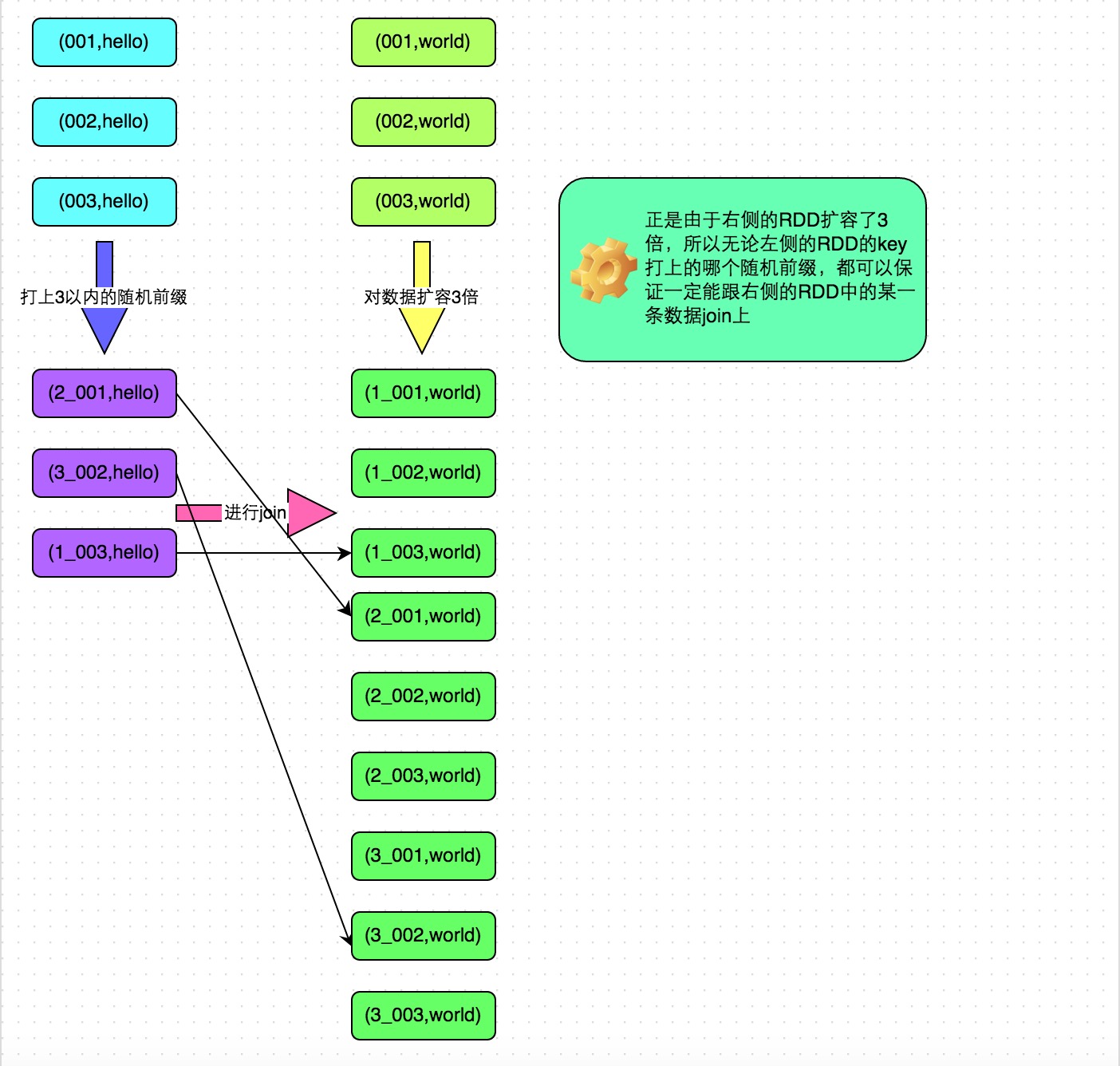
**方案实现思路：**

* 对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key。
* 然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀，而不会导致倾斜的大部分key形成另外一个RDD。
* 接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个单独的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀，不会导致倾斜的大部分key也形成另外一个RDD。
* 再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了。
* 而另外两个普通的RDD就照常join即可。
* 最后将两次join的结果使用union算子合并起来即可，就是最终的join结果。

**方案实现原理：**对于join导致的数据倾斜，如果只是某几个key导致了倾斜，可以将少数几个key分拆成独立RDD，并附加随机前缀打散成n份去进行join，此时这几个key对应的数据就不会集中在少数几个task上，而是分散到多个task进行join了。具体原理见下图。

**方案优点：**对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

**方案缺点：**如果导致倾斜的key特别多的话，比如成千上万个key都导致数据倾斜，那么这种方式也不适合。



[制代码](javascript:void(0);)

1 // 首先从包含了少数几个导致数据倾斜key的rdd1中，采样10%的样本数据。 2 JavaPairRDD<Long, String> sampledRDD = rdd1.sample(false, 0.1); 3 4 // 对样本数据RDD统计出每个key的出现次数，并按出现次数降序排序。 5 // 对降序排序后的数据，取出top 1或者top 100的数据，也就是key最多的前n个数据。 6 // 具体取出多少个数据量最多的key，由大家自己决定，我们这里就取1个作为示范。 7 JavaPairRDD<Long, Long> mappedSampledRDD = sampledRDD.mapToPair( 8 new PairFunction<Tuple2<Long,String>, Long, Long>() { 9 private static final long serialVersionUID = 1L; 10 @Override 11 public Tuple2<Long, Long> call(Tuple2<Long, String> tuple) 12 throws Exception { 13 return new Tuple2<Long, Long>(tuple.\_1, 1L); 14 } 15 }); 16 JavaPairRDD<Long, Long> countedSampledRDD = mappedSampledRDD.reduceByKey( 17 new Function2<Long, Long, Long>() { 18 private static final long serialVersionUID = 1L; 19 @Override 20 public Long call(Long v1, Long v2) throws Exception { 21 return v1 + v2; 22 } 23 }); 24 JavaPairRDD<Long, Long> reversedSampledRDD = countedSampledRDD.mapToPair( 25 new PairFunction<Tuple2<Long,Long>, Long, Long>() { 26 private static final long serialVersionUID = 1L; 27 @Override 28 public Tuple2<Long, Long> call(Tuple2<Long, Long> tuple) 29 throws Exception { 30 return new Tuple2<Long, Long>(tuple.\_2, tuple.\_1); 31 } 32 }); 33 final Long skewedUserid = reversedSampledRDD.sortByKey(false).take(1).get(0).\_2; 34 35 // 从rdd1中分拆出导致数据倾斜的key，形成独立的RDD。 36 JavaPairRDD<Long, String> skewedRDD = rdd1.filter( 37 new Function<Tuple2<Long,String>, Boolean>() { 38 private static final long serialVersionUID = 1L; 39 @Override 40 public Boolean call(Tuple2<Long, String> tuple) throws Exception { 41 return tuple.\_1.equals(skewedUserid); 42 } 43 }); 44 // 从rdd1中分拆出不导致数据倾斜的普通key，形成独立的RDD。 45 JavaPairRDD<Long, String> commonRDD = rdd1.filter( 46 new Function<Tuple2<Long,String>, Boolean>() { 47 private static final long serialVersionUID = 1L; 48 @Override 49 public Boolean call(Tuple2<Long, String> tuple) throws Exception { 50 return !tuple.\_1.equals(skewedUserid); 51 } 52 }); 53 54 // rdd2，就是那个所有key的分布相对较为均匀的rdd。 55 // 这里将rdd2中，前面获取到的key对应的数据，过滤出来，分拆成单独的rdd，并对rdd中的数据使用flatMap算子都扩容100倍。 56 // 对扩容的每条数据，都打上0～100的前缀。 57 JavaPairRDD<String, Row> skewedRdd2 = rdd2.filter( 58 new Function<Tuple2<Long,Row>, Boolean>() { 59 private static final long serialVersionUID = 1L; 60 @Override 61 public Boolean call(Tuple2<Long, Row> tuple) throws Exception { 62 return tuple.\_1.equals(skewedUserid); 63 } 64 }).flatMapToPair(new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() { 65 private static final long serialVersionUID = 1L; 66 @Override 67 public Iterable<Tuple2<String, Row>> call( 68 Tuple2<Long, Row> tuple) throws Exception { 69 Random random = new Random(); 70 List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>(); 71 for(int i = 0; i < 100; i++) { 72 list.add(new Tuple2<String, Row>(i + "\_" + tuple.\_1, tuple.\_2)); 73 } 74 return list; 75 } 76 77 }); 78 79 // 将rdd1中分拆出来的导致倾斜的key的独立rdd，每条数据都打上100以内的随机前缀。 80 // 然后将这个rdd1中分拆出来的独立rdd，与上面rdd2中分拆出来的独立rdd，进行join。 81 JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 = skewedRDD.mapToPair( 82 new PairFunction<Tuple2<Long,String>, String, String>() { 83 private static final long serialVersionUID = 1L; 84 @Override 85 public Tuple2<String, String> call(Tuple2<Long, String> tuple) 86 throws Exception { 87 Random random = new Random(); 88 int prefix = random.nextInt(100); 89 return new Tuple2<String, String>(prefix + "\_" + tuple.\_1, tuple.\_2); 90 } 91 }) 92 .join(skewedUserid2infoRDD) 93 .mapToPair(new PairFunction<Tuple2<String,Tuple2<String,Row>>, Long, Tuple2<String, Row>>() { 94 private static final long serialVersionUID = 1L; 95 @Override 96 public Tuple2<Long, Tuple2<String, Row>> call( 97 Tuple2<String, Tuple2<String, Row>> tuple) 98 throws Exception { 99 long key = Long.valueOf(tuple.\_1.split("\_")[1]); 100 return new Tuple2<Long, Tuple2<String, Row>>(key, tuple.\_2); 101 } 102 }); 103 104 // 将rdd1中分拆出来的包含普通key的独立rdd，直接与rdd2进行join。 105 JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(rdd2); 106 107 // 将倾斜key join后的结果与普通key join后的结果，uinon起来。 108 // 就是最终的join结果。 109 JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);

[制代码](javascript:void(0);)

#### 解决方案七：使用随机前缀和扩容RDD进行join

**方案适用场景：**如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义，此时就只能使用最后一种方案来解决问题了。

**方案实现思路：**

* 该方案的实现思路基本和“解决方案六”类似，首先查看RDD/Hive表中的数据分布情况，找到那个造成数据倾斜的RDD/Hive表，比如有多个key都对应了超过1万条数据。
* 然后将该RDD的每条数据都打上一个n以内的随机前缀。
* 同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。
* 最后将两个处理后的RDD进行join即可。

**方案实现原理：**将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。

**方案优点：**对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

**方案缺点：**该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。

**方案实践经验：**曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。

[制代码](javascript:void(0);)

1 // 首先将其中一个key分布相对较为均匀的RDD膨胀100倍。 2 JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair( 3 new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() { 4 private static final long serialVersionUID = 1L; 5 @Override 6 public Iterable<Tuple2<String, Row>> call(Tuple2<Long, Row> tuple) 7 throws Exception { 8 List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>(); 9 for(int i = 0; i < 100; i++) { 10 list.add(new Tuple2<String, Row>(0 + "\_" + tuple.\_1, tuple.\_2)); 11 } 12 return list; 13 } 14 }); 15 16 // 其次，将另一个有数据倾斜key的RDD，每条数据都打上100以内的随机前缀。 17 JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair( 18 new PairFunction<Tuple2<Long,String>, String, String>() { 19 private static final long serialVersionUID = 1L; 20 @Override 21 public Tuple2<String, String> call(Tuple2<Long, String> tuple) 22 throws Exception { 23 Random random = new Random(); 24 int prefix = random.nextInt(100); 25 return new Tuple2<String, String>(prefix + "\_" + tuple.\_1, tuple.\_2); 26 } 27 }); 28 29 // 将两个处理后的RDD进行join即可。 30 JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD);

[制代码](javascript:void(0);)

#### 解决方案八：多种方案组合使用

在实践中发现，很多情况下，如果只是处理较为简单的数据倾斜场景，那么使用上述方案中的某一种基本就可以解决。但是如果要处理一个较为复杂的数据倾斜场景，那么可能需要将多种方案组合起来使用。比如说，我们针对出现了多个数据倾斜环节的Spark作业，可以先运用解决方案一和二，预处理一部分数据，并过滤一部分数据来缓解；其次可以对某些shuffle操作提升并行度，优化其性能；最后还可以针对不同的聚合或join操作，选择一种方案来优化其性能。大家需要对这些方案的思路和原理都透彻理解之后，在实践中根据各种不同的情况，灵活运用多种方案，来解决自己的数据倾斜问题。

## shuffle调优

### 调优概述

大多数Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。但是也必须提醒大家的是，影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占到一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解shuffle的原理，以及相关参数的说明，同时给出各个参数的调优建议。

### ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。而随着Spark的版本的发展，ShuffleManager也在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager而HashShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

下面我们详细分析一下HashShuffleManager和SortShuffleManager的原理。

### HashShuffleManager运行原理

#### 未经优化的HashShuffleManager

下图说明了未经优化的HashShuffleManager的原理。这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

我们先从shuffle write开始说起。shuffle write阶段，主要就是在一个stage结束计算之后，为了下一个stage可以执行shuffle类的算子（比如reduceByKey），而将每个task处理的数据按key进行“分类”。所谓“分类”，就是对相同的key执行hash算法，从而将相同key都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游stage的一个task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。由此可见，未经优化的shuffle write操作所产生的磁盘文件的数量是极其惊人的。

接着我们来说说shuffle read。shuffle read，通常就是一个stage刚开始时要做的事情。此时该stage的每一个task就需要将上一个stage的计算结果中的所有相同key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行key的聚合或连接等操作。由于shuffle write的过程中，task给下游stage的每个task都创建了一个磁盘文件，因此shuffle read的过程中，每个task只要从上游stage的所有task所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到buffer缓冲中进行聚合操作。以此类推，直到最后将所有数据到拉取完，并得到最终的结果。



### SortShuffleManager运行原理

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。当shuffle read task的数量小于等于spark.shuffle.sort.bypassMergeThreshold参数的值时（默认为200），就会启用bypass机制。

#### 普通运行机制

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

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

一个task将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是merge过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个task就只对应一个磁盘文件，也就意味着该task为下游stage的task准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个task的数据在文件中的start offset与end offset。

SortShuffleManager由于有一个磁盘文件merge的过程，因此大大减少了文件数量。比如第一个stage有50个task，总共有10个Executor，每个Executor执行5个task，而第二个stage有100个task。由于每个task最终只有一个磁盘文件，因此此时每个Executor上只有5个磁盘文件，所有Executor只有50个磁盘文件。

#### bypass运行机制

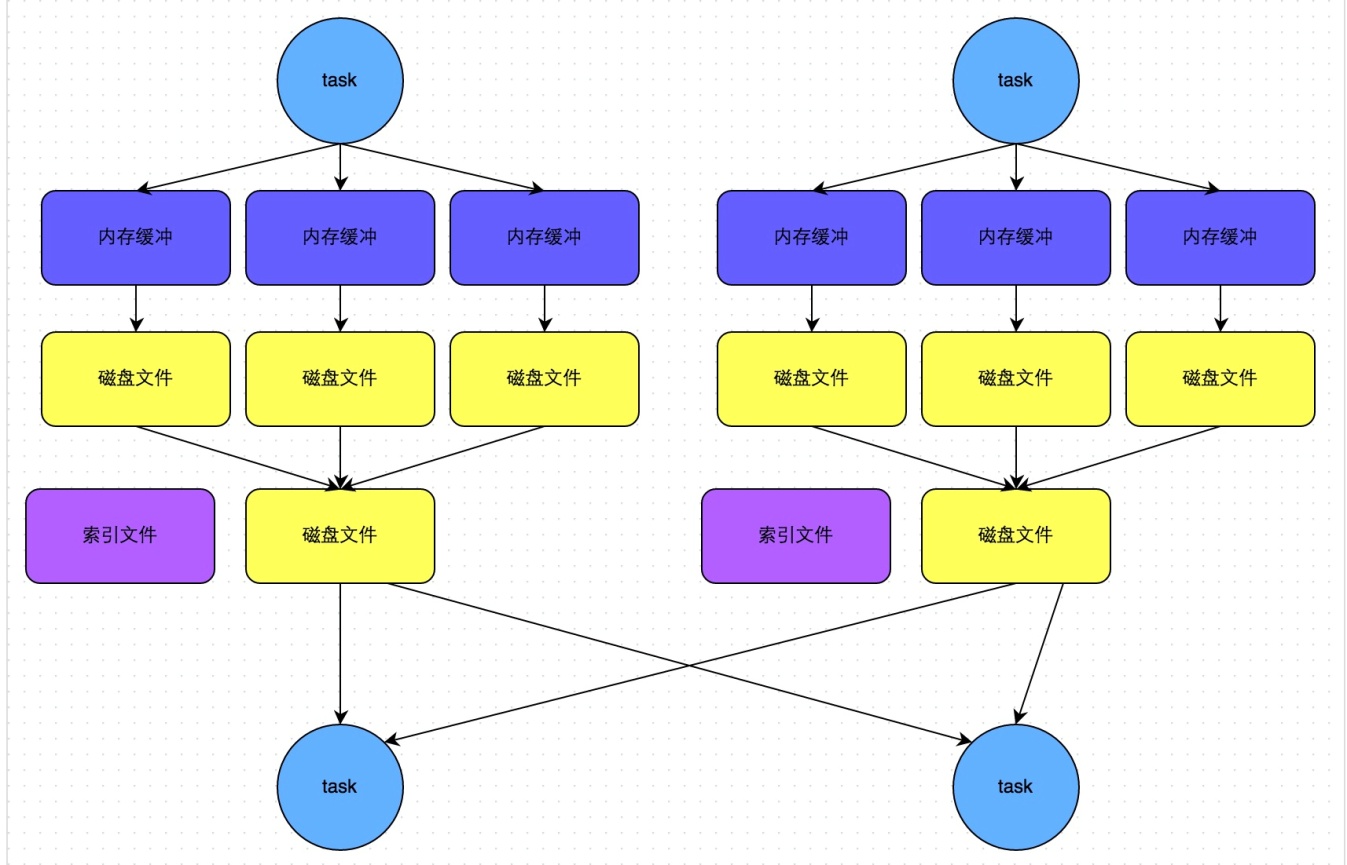
下图说明了bypass SortShuffleManager的原理。bypass运行机制的触发条件如下：

* shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。
* 不是聚合类的shuffle算子（比如reduceByKey）。

此时task会为每个下游task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。



### shuffle相关参数调优

以下是Shffule过程中的一些主要参数，这里详细讲解了各个参数的功能、默认值以及基于实践经验给出的调优建议。

#### spark.shuffle.file.buffer

* 默认值：32k
* 参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。
* 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

#### spark.reducer.maxSizeInFlight

* 默认值：48m
* 参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。
* 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

#### spark.shuffle.io.maxRetries

* 默认值：3
* 参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。
* 调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

#### spark.shuffle.io.retryWait

* 默认值：5s
* 参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。
* 调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

#### spark.shuffle.memoryFraction

* 默认值：0.2
* 参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。
* 调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，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。在实践中发现，合理调节该参数可以将性能提升10%左右。

#### spark.shuffle.manager

* 默认值：sort
* 参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。
* 调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

#### spark.shuffle.sort.bypassMergeThreshold

* 默认值：200
* 参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。
* 调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

#### spark.shuffle.consolidateFiles

* 默认值：false
* 参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。
* 调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。

## Spark面试题汇总

!!!!Spark面试题汇总大全!!!!:

<http://www.aboutyun.com/thread-24246-1-1.html>

### 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、里面的元素可并行计算的集合。

\* 弹性表示

\* 1、RDD中的数据可以存储在内存或者是磁盘

\* 2、RDD中的分区是可以改变的

\*五大特性：

\* A list of partitions

一个分区列表，RDD中的数据都存在一个分区列表里面

\* 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 (e.g. to say that the RDD is hash-partitioned)

可选的，针对于kv类型的RDD才具有这个特性，作用是决定了数据的来源以及数据处理后的去向

\* Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

可选项，数据本地性，数据位置最优

### 概述一下spark中的常用算子区别（map、mapPartitions、foreach、foreachPartition）

\* map：用于遍历RDD,将函数f应用于每一个元素，返回新的RDD(transformation算子)。

\* foreach:用于遍历RDD,将函数f应用于每一个元素，无返回值(action算子)。

\* mapPartitions:用于遍历操作RDD中的每一个分区，返回生成一个新的RDD（transformation算子）。

\* foreachPartition: 用于遍历操作RDD中的每一个分区。无返回值(action算子)。

\* 总结：一般使用mapPartitions或者foreachPartition算子比map和foreach更加高效，推荐使用。

### 谈谈spark中的宽窄依赖

\* RDD和它依赖的父RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。

\* 宽依赖：指的是多个子RDD的Partition会依赖同一个父RDD的Partition

\* 窄依赖：指的是每一个父RDD的Partition最多被子RDD的一个Partition使用。

### spark中如何划分stage

\* 1.Spark Application中可以因为不同的Action触发众多的job，一个Application中可以有很多的job，每个job是由一个或者多个Stage构成的，后面的Stage依赖于前面的Stage，也就是说只有前面依赖的Stage计算完毕后，后面的Stage才会运行。

\* 2.Stage划分的依据就是宽依赖，何时产生宽依赖，例如reduceByKey,groupByKey的算子，会导致宽依赖的产生。

\* 3.由Action（例如collect）导致了SparkContext.runJob的执行，最终导致了DAGScheduler中的submitJob的执行，其核心是通过发送一个case class JobSubmitted对象给eventProcessLoop。

eventProcessLoop是DAGSchedulerEventProcessLoop的具体实例，而DAGSchedulerEventProcessLoop是eventLoop的子类，具体实现EventLoop的onReceive方法，onReceive方法转过来回调doOnReceive

\* 4.在doOnReceive中通过模式匹配的方法把执行路由到

\* 5.在handleJobSubmitted中首先创建finalStage，创建finalStage时候会建立父Stage的依赖链条

\* 总结：以来是从代码的逻辑层面上来展开说的，可以简单点说：写介绍什么是RDD中的宽窄依赖，然后在根据DAG有向无环图进行划分，从当前job的最后一个算子往前推，遇到宽依赖，那么当前在这个批次中的所有算子操作都划分成一个stage,然后继续按照这种方式在继续往前推，如在遇到宽依赖，又划分成一个stage,一直到最前面的一个算子。最后整个job会被划分成多个stage,而stage之间又存在依赖关系，后面的stage依赖于前面的stage。

### spark-submit的时候如何引入外部jar包

\* 在通过spark-submit提交任务时，可以通过添加配置参数来指定

\* --driver-class-path 外部jar包

\* --jars 外部jar包

### 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，需要调大Driver的内存。

map过程产生大量对象导致内存溢出

\* 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。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，

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，同时配置--executor-cores或者spark.executor.cores参数，确保Executor资源分配均匀。

### spark中cache和persist的区别

\* cache：缓存数据，默认是缓存在内存中，其本质还是调用persist

\* persist:缓存数据，有丰富的数据缓存策略。数据可以保存在内存也可以保存在磁盘中，使用的时候指定对应的缓存级别就可以了。

### 简要描述Spark分布式集群搭建的步骤

主要是引入了zookeeper

### 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。

### 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。

### flume整合sparkStreaming问题

\* (1)、如何实现sparkStreaming读取flume中的数据

\* 可以这样说：

\* 前期经过技术调研，查看官网相关资料，发现sparkStreaming整合flume有2种模式，一种是拉模式，一种是推模式，然后在简单的聊聊这2种模式的特点，以及如何部署实现，需要做哪些事情，最后对比两种模式的特点，选择那种模式更好。

\* 推模式：Flume将数据Push推给Spark Streaming

\* 拉模式：Spark Streaming从flume 中Poll拉取数据

\* (2)、在实际开发的时候是如何保证数据不丢失的

\* 可以这样说：

\* flume那边采用的channel是将数据落地到磁盘中，保证数据源端安全性（可以在补充一下，flume在这里的channel可以设置为memory内存中，提高数据接收处理的效率，但是由于数据在内存中，安全机制保证不了，故选择channel为磁盘存储。整个流程运行有一点的延迟性）

\* sparkStreaming通过拉模式整合的时候，使用了FlumeUtils这样一个类，该类是需要依赖一个额外的jar包（spark-streaming-flume\_2.10）

\* 要想保证数据不丢失，数据的准确性，可以在构建StreamingConext的时候，利用StreamingContext.getOrCreate（checkpoint, creatingFunc: () => StreamingContext）来创建一个StreamingContext,使用StreamingContext.getOrCreate来创建StreamingContext对象，传入的第一个参数是checkpoint的存放目录，第二参数是生成StreamingContext对象的用户自定义函数。如果checkpoint的存放目录存在，则从这个目录中生成StreamingContext对象；如果不存在，才会调用第二个函数来生成新的StreamingContext对象。在creatingFunc函数中，除了生成一个新的StreamingContext操作，还需要完成各种操作，然后调用ssc.checkpoint(checkpointDirectory)来初始化checkpoint功能，最后再返回StreamingContext对象。

这样，在StreamingContext.getOrCreate之后，就可以直接调用start()函数来启动（或者是从中断点继续运行）流式应用了。如果有其他在启动或继续运行都要做的工作，可以在start()调用前执行。

---- 流式计算中使用checkpoint的作用----：

\* 保存元数据，包括流式应用的配置、流式没崩溃之前定义的各种操作、未完成所有操作的batch。元数据被存储到容忍失败的存储系统上，如HDFS。这种ckeckpoint主要针对driver失败后的修复。

\* 保存流式数据，也是存储到容忍失败的存储系统上，如HDFS。这种ckeckpoint主要针对window operation、有状态的操作。无论是driver失败了，还是worker失败了，这种checkpoint都够快速恢复，而不需要将很长的历史数据都重新计算一遍（以便得到当前的状态）。

\* 设置流式数据checkpoint的周期

\* 对于一个需要做checkpoint的DStream结构，可以通过调用DStream.checkpoint(checkpointInterval)来设置ckeckpoint的周期，经验上一般将这个checkpoint周期设置成batch周期的5至10倍。

\* 使用write ahead logs功能

\* 这是一个可选功能，建议加上。这个功能将使得输入数据写入之前配置的checkpoint目录。这样有状态的数据可以从上一个checkpoint开始计算。开启的方法是把spark.streaming.receiver.writeAheadLogs.enable这个property设置为true。另外，由于输入RDD的默认StorageLevel是MEMORY\_AND\_DISK\_2，即数据会在两台worker上做replication。实际上，Spark Streaming模式下，任何从网络输入数据的Receiver（如kafka、flume、socket）都会在两台机器上做数据备份。如果开启了write ahead logs的功能，建议把StorageLevel改成MEMORY\_AND\_DISK\_SER。修改的方法是，在创建RDD时由参数传入。

\* 使用以上的checkpoint机制，确实可以保证数据0丢失。但是一个前提条件是，数据发送端必须要有缓存功能，这样才能保证在spark应用重启期间，数据发送端不会因为spark streaming服务不可用而把数据丢弃。而flume具备这种特性，同样kafka也具备。

\* (3)Spark Streaming的数据可靠性

\* 有了checkpoint机制、write ahead log机制、Receiver缓存机器、可靠的Receiver（即数据接收并备份成功后会发送ack），可以保证无论是worker失效还是driver失效，都是数据0丢失。原因是：如果没有Receiver服务的worker失效了，RDD数据可以依赖血统来重新计算；如果Receiver所在worker失败了，由于Reciever是可靠的，并有write ahead log机制，则收到的数据可以保证不丢；如果driver失败了，可以从checkpoint中恢复数据重新构建。

### 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。

Spark：拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是Job中间输出结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的MapReduce的算法。

数据过于繁杂，并且需要让计算通过迭代，并在内存中，极大地提高效率的场景

Strom：一个分布式实时计算系统，Storm是一个任务并行连续计算引擎。 Storm本身并不典型在Hadoop集群上运行，它使用Apache ZooKeeper的和自己的主/从工作进程，协调拓扑，主机和工作者状态，保证信息的语义。无论如何， Storm必定还是可以从HDFS文件消费或者从文件写入到HDFS。

Hive：基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的sql查询功能，可以将sql语句转换为MapReduce任务进行运行。

应用场景：十分适合数据仓库的统计分析。

Hbase:

应用场景： 数据量太大，以至于传统RDBMS无法胜任、

联机业务功能开发、

离线数据分析（数据仓库），

### ----简答题 ---- 网上资料 ---

#### 1. Spark master使用zookeeper进行HA的，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，Driver和Application以及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2点: 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本身的调度和处理和Master是没有任何关系的！在Master的切换过程中唯一的影响是不能提交新的Job：一方面不能够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，已经运行的程序中也不能够因为Action操作触发新的Job的提交请求；

#### 2.Spark master HA 主从切换过程不会影响集群已有的作业运行？

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。

#### Apache Spark有哪些常见的稳定版本

答：常见的大的稳定版本有Spark 1.3,Spark1.6, Spark

#### driver的功能是什么？

答：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，是程序的人口点；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。

#### spark的有几种部署模式，每种模式特点？

1）本地模式  
Spark不一定非要跑在hadoop集群，可以在本地，起多个线程的方式来指定。将Spark应用以多线程的方式直接运行在本地，一般都是为了方便调试，本地模式分三类  
·  local：只启动一个executor  
·  local[k]:启动k个executor  
·  local：启动跟cpu数目相同的 executor  
2)standalone模式  
分布式部署集群， 自带完整的服务，资源管理和任务监控是Spark自己监控，这个模式也是其他模式的基础，  
3)Spark on yarn模式  
分布式部署集群，资源和任务监控交给yarn管理，但是目前仅支持粗粒度资源分配方式，包含cluster和client运行模式，cluster适合生产，driver运行在集群子节点，具有容错功能，client适合调试，dirver运行在客户端  
4）Spark On Mesos模式。官方推荐这种模式（当然，原因之一是血缘关系）。正是由于Spark开发之初就考虑到支持Mesos，因此，目前而言，Spark运行在Mesos上会比运行在YARN上更加灵活，更加自然。用户可选择两种调度模式之一运行自己的应用程序：

#### 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？

答： 1）Sparkcore：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

#### Spark中Work的主要工作是什么？

答：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。

#### Spark为什么比mapreduce快？

答：1）基于内存计算，减少低效的磁盘交互；2）高效的调度算法，基于DAG；3)容错机制Linage，精华部分就是DAG和Lingae

#### 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。

#### 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。

#### RDD机制？

答：rdd分布式弹性数据集，简单的理解成一种数据结构，是spark框架上的通用货币。   
所有算子都是基于rdd来执行的，不同的场景会有不同的rdd实现类，但是都可以进行互相转换。   
rdd执行过程中会形成dag图，然后形成lineage保证容错性等。 从物理的角度来看rdd存储的是block和node之间的映射。

#### spark工作机制？

答：用户在client端提交作业后，会由Driver运行main方法并创建sparkcontext上下文。 执行add算子形成dag图输入dagscheduler，按照add之间的依赖关系划分stage输入task scheduler。 task scheduler会将stage划分为taskset分发到各个节点executor中执行。

#### spark的优化怎么做？

答： spark调优比较复杂，但是大体可以分为三个方面来进行，1）平台层面的调优：防止不必要的jar包分发，提高数据的本地性，选择高效的存储格式如parquet，2）应用程序层面的调优：过滤操作符的优化降低过多小任务，降低单条记录的资源开销，处理数据倾斜，复用RDD进行缓存，作业并行化执行等等，3）JVM层面的调优：设置合适的资源量，设置合理的JVM，启用高效的序列化方法如kyro，增大off head内存等等

#### spark-submit的时候如何引入外部jar包

方法一：spark-submit –jars  
根据spark官网，在提交任务的时候指定–jars，用逗号分开。这样做的缺点是每次都要指定jar包，如果jar包少的话可以这么做，但是如果多的话会很麻烦。  
命令：spark-submit --master yarn-client --jars \*\*\*.jar,\*\*\*.jar  
方法二：extraClassPath  
提交时在spark-default中设定参数，将所有需要的jar包考到一个文件里，然后在参数中指定该目录就可以了，较上一个方便很多：  
spark.executor.extraClassPath=/home/hadoop/wzq\_workspace/lib/\* spark.driver.extraClassPath=/home/hadoop/wzq\_workspace/lib/\*  
需要注意的是,你要在所有可能运行spark任务的机器上保证该目录存在，并且将jar包考到所有机器上。这样做的好处是提交代码的时候不用再写一长串jar了，缺点是要把所有的jar包都拷一遍。

#### cache和pesist的区别

答：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

#### Spark使用parquet文件存储格式能带来哪些好处？

1) 如果说HDFS 是大数据时代分布式文件系统首选标准，那么parquet则是整个大数据时代文件存储格式实时首选标准  
2) 速度更快：从使用spark sql操作普通文件CSV和parquet文件速度对比上看，绝大多数情况会比使用csv等普通文件速度提升10倍左右，在一些普通文件系统无法在spark上成功运行的情况下，使用parquet很多时候可以成功运行  
3) parquet的压缩技术非常稳定出色，在spark sql中对压缩技术的处理可能无法正常的完成工作例如会导致losttask，lost executor但是此时如果使用parquet就可以正常的完成  
4) 极大的减少磁盘I/o,通常情况下能够减少75%的存储空间，由此可以极大的减少spark sql处理数据的时候的数据输入内容，尤其是在spark1.6x中有个下推过滤器在一些情况下可以极大的减少磁盘的IO和内存的占用，（下推过滤器）  
5) spark 1.6x parquet方式极大的提升了扫描的吞吐量，极大提高了数据的查找速度spark1.6和spark1.5x相比而言，提升了大约1倍的速度，在spark1.6X中，操作parquet时候cpu也进行了极大的优化，有效的降低了cpu  
6) 采用parquet可以极大的优化spark的调度和执行。我们测试spark如果用parquet可以有效的减少stage的执行消耗，同时可以优化执行路径

#### Spark如何自定义partitioner分区器？

答：1）spark默认实现了HashPartitioner和RangePartitioner两种分区策略，我们也可以自己扩展分区策略，自定义分区器的时候继承org.apache.spark.Partitioner类，实现类中的三个方法  
def numPartitions: Int：这个方法需要返回你想要创建分区的个数；  
def getPartition(key: Any): Int：这个函数需要对输入的key做计算，然后返回该key的分区ID，范围一定是0到numPartitions-1；  
equals()：这个是Java标准的判断相等的函数，之所以要求用户实现这个函数是因为Spark内部会比较两个RDD的分区是否一样。

### -------Spark on Yarn面试篇

#### 1.描述Yarn执行一个任务的过程？

1）客户端client向ResouceManager提交Application，ResouceManager接受Application  
并根据集群资源状况选取一个node来启动Application的任务调度器driver（ApplicationMaster）  
2）ResouceManager找到那个node，命令其该node上的nodeManager来启动一个新的  
JVM进程运行程序的driver（ApplicationMaster）部分，driver（ApplicationMaster）启动时会首先向ResourceManager注册，说明由自己来负责当前程序的运行  
3）driver（ApplicationMaster）开始下载相关jar包等各种资源，基于下载的jar等信息决定向ResourceManager申请具体的资源内容。  
4）ResouceManager接受到driver（ApplicationMaster）提出的申请后，会最大化的满足  
资源分配请求，并发送资源的元数据信息给driver（ApplicationMaster）；  
5）driver（ApplicationMaster）收到发过来的资源元数据信息后会根据元数据信息发指令给具体  
机器上的NodeManager，让其启动具体的container。  
6）NodeManager收到driver发来的指令，启动container，container启动后必须向driver（ApplicationMaster）注册。  
7）driver（ApplicationMaster）收到container的注册，开始进行任务的调度和计算，直到  
任务完成。  
补充：如果ResourceManager第一次没有能够满足driver（ApplicationMaster）的资源请求  
，后续发现有空闲的资源，会主动向driver（ApplicationMaster）发送可用资源的元数据信息  
以提供更多的资源用于当前程序的运行。

#### 2.Yarn中的container是由谁负责销毁的，在Hadoop Mapreduce中container可以复用么？

答：ApplicationMaster负责销毁，在Hadoop Mapreduce不可以复用，在spark on yarn程序container可以复用

#### 提交任务时，如何指定Spark Application的运行模式？

1）cluster模式：./spark-submit --class xx.xx.xx --master yarn --deploy-mode cluster xx.jar  
2) client模式:./spark-submit --class xx.xx.xx --master yarn --deploy-mode client xx.jar

#### 不启动Spark集群Master和work服务，可不可以运行Spark程序？

答：可以，只要资源管理器第三方管理就可以，如由yarn管理，spark集群不启动也可以使用spark；spark集群启动的是work和master，这个其实就是资源管理框架，yarn中的resourceManager相当于master，NodeManager相当于worker，做计算是Executor，和spark集群的work和manager可以没关系，归根接底还是JVM的运行，只要所在的JVM上安装了spark就可以。

#### Spark中的4040端口由什么功能?

答：收集Spark作业运行的信息

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

答：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。

#### 如何使用命令查看application运行的日志信息

答：yarn logs -applicationId <app ID>

#### Spark on Yarn 模式有哪些优点？

1)与其他计算框架共享集群资源（eg.Spark框架与MapReduce框架同时运行，如果不用Yarn进行资源分配，MapReduce分到的内存资源会很少，效率低下）；资源按需分配，进而提高集群资源利用等。  
2)相较于Spark自带的Standalone模式，Yarn的资源分配更加细致  
3)Application部署简化，例如Spark，Storm等多种框架的应用由客户端提交后，由Yarn负责资源的管理和调度，利用Container作为资源隔离的单位，以它为单位去使用内存,cpu等。  
4)Yarn通过队列的方式，管理同时运行在Yarn集群中的多个服务，可根据不同类型的应用程序负载情况，调整对应的资源使用量，实现资源弹性管理。

#### 谈谈你对container的理解？

1）Container作为资源分配和调度的基本单位，其中封装了的资源如内存，CPU，磁盘，网络带宽等。 目前yarn仅仅封装内存和CPU  
2)Container由ApplicationMaster向ResourceManager申请的，由ResouceManager中的资源调度器异步分配给ApplicationMaster  
3) Container的运行是由ApplicationMaster向资源所在的NodeManager发起的，Container运行时需提供内部执行的任务命令.

#### 运行在yarn中Application有几种类型的container？

1） 运行ApplicationMaster的Container：这是由ResourceManager（向内部的资源调度器）申请和启动的，用户提交应用程序时，可指定唯一的ApplicationMaster所需的资源；  
2） 运行各类任务的Container：这是由ApplicationMaster向ResourceManager申请的，并由ApplicationMaster与NodeManager通信以启动之。

#### Spark on Yarn架构是怎么样的？（要会画哦，这个图）

#### Executor启动时，资源通过哪几个参数指定？

1)num-executors是executor的数量  
2)executor-memory 是每个executor使用的内存  
3)executor-cores 是每个executor分配的CPU

#### 一个task的map数量由谁来决定？

一般情况下，在输入源是文件的时候，一个task的map数量由splitSize来决定的，那么splitSize是由以下几个来决定的  
goalSize = totalSize / mapred.map.tasks  
inSize = max {mapred.min.split.size, minSplitSize}  
splitSize = max (minSize, min(goalSize, dfs.block.size))  
一个task的reduce数量，由partition决定。

#### reduce后输出的数据量有多大？

并不是想知道确切的数据量有多大这个，而是想问你，MR的执行机制，开发完程序，有没有认真评估程序运行效率  
1）用于处理redcue任务的资源情况，如果是MRV1的话，分了多少资源给map，多少个reduce  
   如果是MRV2的话，可以提一下，集群有分了多少内存、CPU给yarn做计算 。  
2）结合实际应用场景回答，输入数据有多大，大约多少条记录，做了哪些逻辑操作，输出的时候有多少条记录，执行了多久，reduce执行时候的数据有没有倾斜等  
3）再提一下，针对mapReduce做了哪几点优化，速度提升了多久，列举1,2个优化点就可以

#### 你的项目提交到job的时候数据量有多大？

答：1）回答出数据是什么格式，有没有采用什么压缩，采用了压缩的话，压缩比大概是多少；2）文件大概多大：大概起了多少个map，起了多少个reduce，map阶段读取了多少数据，reduce阶段读取了多少数据，程序大约执行了多久，3）集群什么规模，集群有多少节点，多少内存，多少CPU核数等。把这些点回答进去，而不是给个数字了事。

#### 你们提交的job任务大概有多少个？这些job执行完大概用多少时间？

还是考察你开发完程序有没有认真观察过程序的运行，有没有评估程序运行的效率

#### 你们业务数据量多大？有多少行数据？

这个也是看你们有没有实际的经验,对于没有实战的同学，请把回答的侧重点放在MR的运行机制上面，  
MR运行效率方面，以及如何优化MR程序（看别人的优化demo，然后在虚拟机上拿demo做一下测试）。

#### 如何杀死一个正在运行的job

杀死一个job  
MRV1：Hadoop job kill jobid  
YARN: yarn application -kill applicationId

#### 列出你所知道的调度器，说明其工作原理

1. Fifo schedular 默认的调度器  先进先出  
   b) Capacity schedular  计算能力调度器  选择占用内存小  优先级高的  
   c) Fair schedular  公平调度器  所有job 占用相同资源

### -------spark sql 面试篇

Join常见分类以及基本实现机制

当前SparkSQL支持三种Join算法－shuffle hash join、broadcast hash join以及sort merge join。其中前两者归根到底都属于hash join，只不过在hash join之前需要先shuffle还是先broadcast

<http://hbasefly.com/2017/03/19/sparksql-basic-join/>

### ----选择题 ---

1. Spark 的四大组件下面哪个不是 (D )  
A.Spark Streaming    B. Mlib   
C Graphx    D.Spark R  
  
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  
  
3.spark 1.4 版本的最大变化 (B )  
A spark sql Release 版本  B .引入 Spark R   
C DataFrame D.支持动态资源分配  
  
4. Spark Job 默认的调度模式 (A )  
A FIFO   B FAIR     
C 无   D 运行时指定  
  
5.哪个不是本地模式运行的个条件 ( 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  
  
6.下面哪个不是 RDD 的特点 (C )  
A. 可分区   B 可序列化   C 可修改   D 可持久化  
  
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  
  
9.Spark 支持的分布式部署方式中哪个是错误的 (D )  
A standalone B spark on mesos    
C spark on YARN D Spark on local  
  
10.Stage 的 Task 的数量由什么决定 (A )  
A Partition B Job C Stage D TaskScheduler  
  
11.下面哪个操作是窄依赖 (B )  
A join B filter   
C group D sort  
  
12.下面哪个操作肯定是宽依赖 (C )  
A map B flatMap   
C reduceByKey D sample  
  
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  
  
14 默认的存储级别 (A )  
A MEMORY\_ONLY B MEMORY\_ONLY\_SER  
C MEMORY\_AND\_DISK D MEMORY\_AND\_DISK\_SER  
  
15 spark.deploy.recoveryMode 不支持那种 (D )  
A.ZooKeeper B. FileSystem   
D NONE D Hadoop  
  
16.下列哪个不是 RDD 的缓存方法 (C )  
A persist() B Cache()   
C Memory()  
  
17.Task 运行在下来哪里个选项中 Executor 上的工作单元 (C )  
A Driver program B. spark master   
C.worker node D Cluster manager  
  
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  
  
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  
  
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

### 补充资料: (spark 集群standalone + spark on yarn)

!!推荐!! [Spark入门实战系列--4.Spark运行架构](https://www.cnblogs.com/shishanyuan/p/4721326.html)

#### Spark在不同集群中的运行架构

Spark注重建立良好的生态系统，它不仅支持多种外部文件存储系统，提供了多种多样的集群运行模式。部署在单台机器上时，既可以用本地（Local）模式运行，也可以使用伪分布式模式来运行；当以分布式集群部署的时候，可以根据自己集群的实际情况选择Standalone模式（Spark自带的模式）、YARN-Client模式或者YARN-Cluster模式。Spark的各种运行模式虽然在启动方式、运行位置、调度策略上各有不同，但它们的目的基本都是一致的，就是在合适的位置安全可靠的根据用户的配置和Job的需要运行和管理Task。

##### Spark on Standalone运行过程

Standalone模式是Spark实现的资源调度框架，其主要的节点有Client节点、Master节点和Worker节点。其中Driver既可以运行在Master节点上中，也可以运行在本地Client端。当用spark-shell交互式工具提交Spark的Job时，Driver在Master节点上运行；当使用spark-submit工具提交Job或者在Eclips、IDEA等开发平台上使用”new SparkConf.setManager(“spark://master:7077”)”方式运行Spark任务时，Driver是运行在本地Client端上的。

其运行过程如下：

1.SparkContext连接到Master，向Master注册并申请资源（CPU Core 和Memory）；

2.Master根据SparkContext的资源申请要求和Worker心跳周期内报告的信息决定在哪个Worker上分配资源，然后在该Worker上获取资源，然后启动StandaloneExecutorBackend；

3.StandaloneExecutorBackend向SparkContext注册；

4.SparkContext将Applicaiton代码发送给StandaloneExecutorBackend；并且SparkContext解析Applicaiton代码，构建DAG图，并提交给DAG Scheduler分解成Stage（当碰到Action操作时，就会催生Job；每个Job中含有1个或多个Stage，Stage一般在获取外部数据和shuffle之前产生），然后以Stage（或者称为TaskSet）提交给Task Scheduler，Task Scheduler负责将Task分配到相应的Worker，最后提交给StandaloneExecutorBackend执行；

5.StandaloneExecutorBackend会建立Executor线程池，开始执行Task，并向SparkContext报告，直至Task完成。

6.所有Task完成后，SparkContext向Master注销，释放资源。

[](http://images0.cnblogs.com/blog/107289/201508/111610287072160.jpg)

##### Spark on YARN运行过程

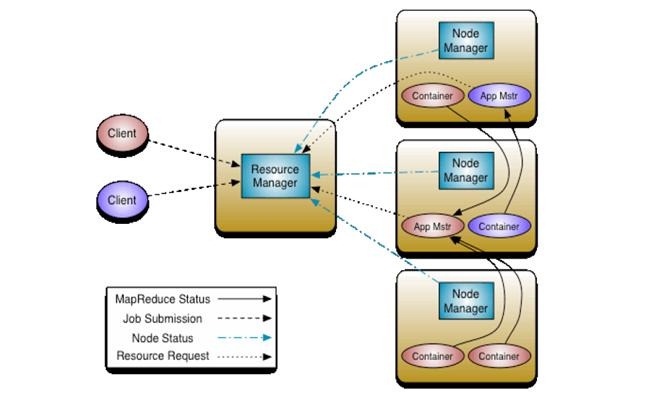
YARN是一种统一资源管理机制，在其上面可以运行多套计算框架。目前的大数据技术世界，大多数公司除了使用Spark来进行数据计算，由于历史原因或者单方面业务处理的性能考虑而使用着其他的计算框架，比如MapReduce、Storm等计算框架。Spark基于此种情况开发了Spark on YARN的运行模式，由于借助了YARN良好的弹性资源管理机制，不仅部署Application更加方便，而且用户在YARN集群中运行的服务和Application的资源也完全隔离，更具实践应用价值的是YARN可以通过队列的方式，管理同时运行在集群中的多个服务。

Spark on YARN模式根据Driver在集群中的位置分为两种模式：一种是YARN-Client模式，另一种是YARN-Cluster（或称为YARN-Standalone模式）。

2.2.1 YARN框架流程

任何框架与YARN的结合，都必须遵循YARN的开发模式。在分析Spark on YARN的实现细节之前，有必要先分析一下YARN框架的一些基本原理。

Yarn框架的基本运行流程图为：

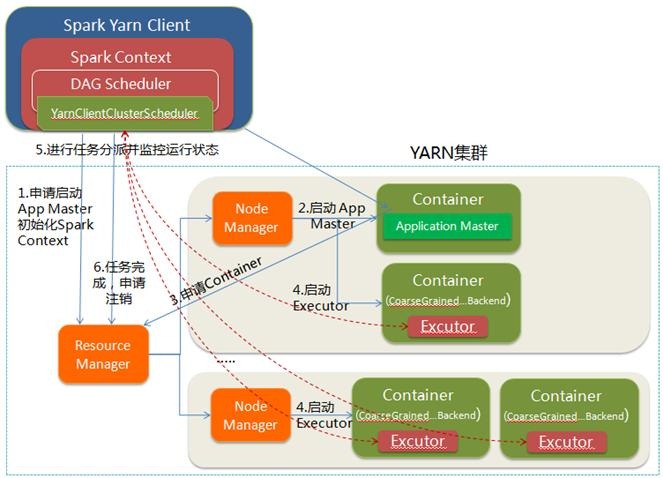
[](http://images0.cnblogs.com/blog/107289/201508/111610367546066.jpg)

其中，ResourceManager负责将集群的资源分配给各个应用使用，而资源分配和调度的基本单位是Container，其中封装了机器资源，如内存、CPU、磁盘和网络等，每个任务会被分配一个Container，该任务只能在该Container中执行，并使用该Container封装的资源。NodeManager是一个个的计算节点，主要负责启动Application所需的Container，监控资源（内存、CPU、磁盘和网络等）的使用情况并将之汇报给ResourceManager。ResourceManager与NodeManagers共同组成整个数据计算框架，ApplicationMaster与具体的Application相关，主要负责同ResourceManager协商以获取合适的Container，并跟踪这些Container的状态和监控其进度。

2.2.2 YARN-Client

Yarn-Client模式中，Driver在客户端本地运行，这种模式可以使得Spark Application和客户端进行交互，因为Driver在客户端，所以可以通过webUI访问Driver的状态，默认是http://hadoop1:4040访问，而YARN通过http:// hadoop1:8088访问。

YARN-client的工作流程分为以下几个步骤：

[](http://images0.cnblogs.com/blog/107289/201508/111610428794762.jpg)

1.Spark Yarn Client向YARN的ResourceManager申请启动Application Master。同时在SparkContent初始化中将创建DAGScheduler和TASKScheduler等，由于我们选择的是Yarn-Client模式，程序会选择YarnClientClusterScheduler和YarnClientSchedulerBackend；

2.ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，与YARN-Cluster区别的是在该ApplicationMaster不运行SparkContext，只与SparkContext进行联系进行资源的分派；

3.Client中的SparkContext初始化完毕后，与ApplicationMaster建立通讯，向ResourceManager注册，根据任务信息向ResourceManager申请资源（Container）；

4.一旦ApplicationMaster申请到资源（也就是Container）后，便与对应的NodeManager通信，要求它在获得的Container中启动启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启动后会向Client中的SparkContext注册并申请Task；

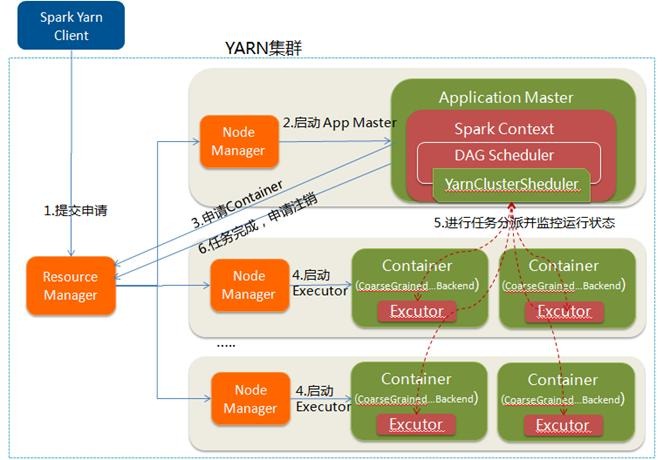
5.Client中的SparkContext分配Task给CoarseGrainedExecutorBackend执行，CoarseGrainedExecutorBackend运行Task并向Driver汇报运行的状态和进度，以让Client随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6.应用程序运行完成后，Client的SparkContext向ResourceManager申请注销并关闭自己。

2.2.3 YARN-Cluster

在YARN-Cluster模式中，当用户向YARN中提交一个应用程序后，YARN将分两个阶段运行该应用程序：第一个阶段是把Spark的Driver作为一个ApplicationMaster在YARN集群中先启动；第二个阶段是由ApplicationMaster创建应用程序，然后为它向ResourceManager申请资源，并启动Executor来运行Task，同时监控它的整个运行过程，直到运行完成。

YARN-cluster的工作流程分为以下几个步骤：

[](http://images0.cnblogs.com/blog/107289/201508/111610511924057.jpg)

1.   Spark Yarn Client向YARN中提交应用程序，包括ApplicationMaster程序、启动ApplicationMaster的命令、需要在Executor中运行的程序等；

2.   ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，其中ApplicationMaster进行SparkContext等的初始化；

3.   ApplicationMaster向ResourceManager注册，这样用户可以直接通过ResourceManage查看应用程序的运行状态，然后它将采用轮询的方式通过RPC协议为各个任务申请资源，并监控它们的运行状态直到运行结束；

4.   一旦ApplicationMaster申请到资源（也就是Container）后，便与对应的NodeManager通信，要求它在获得的Container中启动启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启动后会向ApplicationMaster中的SparkContext注册并申请Task。这一点和Standalone模式一样，只不过SparkContext在Spark Application中初始化时，使用CoarseGrainedSchedulerBackend配合YarnClusterScheduler进行任务的调度，其中YarnClusterScheduler只是对TaskSchedulerImpl的一个简单包装，增加了对Executor的等待逻辑等；

5.   ApplicationMaster中的SparkContext分配Task给CoarseGrainedExecutorBackend执行，CoarseGrainedExecutorBackend运行Task并向ApplicationMaster汇报运行的状态和进度，以让ApplicationMaster随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6.   应用程序运行完成后，ApplicationMaster向ResourceManager申请注销并关闭自己。

2.2.4 YARN-Client 与 YARN-Cluster 区别

理解YARN-Client和YARN-Cluster深层次的区别之前先清楚一个概念：Application Master。在YARN中，每个Application实例都有一个ApplicationMaster进程，它是Application启动的第一个容器。它负责和ResourceManager打交道并请求资源，获取资源之后告诉NodeManager为其启动Container。从深层次的含义讲YARN-Cluster和YARN-Client模式的区别其实就是ApplicationMaster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，因而YARN-Cluster模式不适合运行交互类型的作业；

  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。