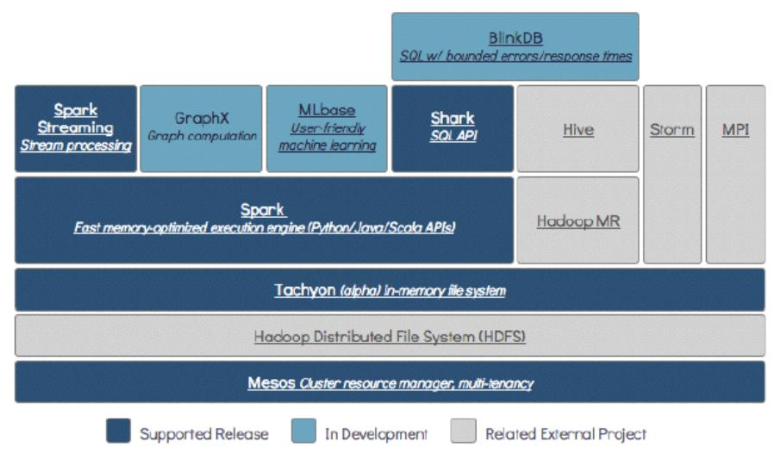
# Spark-day01

1. **Spark初始**
2. 什么是Spark

Apache Spark 是专为大规模数据处理而设计的快速通用的计算引擎。Spark是UC Berkeley AMP lab (加州大学伯克利分校的AMP实验室)所开源的类Hadoop MapReduce的通用并行计算框架，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。

Spark是Scala编写，方便快速编程。

1. 总体技术栈讲解



1. Spark演变历史
2. Spark与MapReduce的区别

* 都是分布式计算框架，Spark基于内存，MR基于HDFS。Spark处理数据的能力一般是MR的十倍以上，Spark中除了基于内存计算外，还有DAG有向无环图来切分任务的执行先后顺序。

1. Spark运行模式

* Local

多用于本地测试，如在eclipse，idea中写程序测试等。

* Standalone

Standalone是Spark自带的一个资源调度框架，它支持完全分布式。

* Yarn

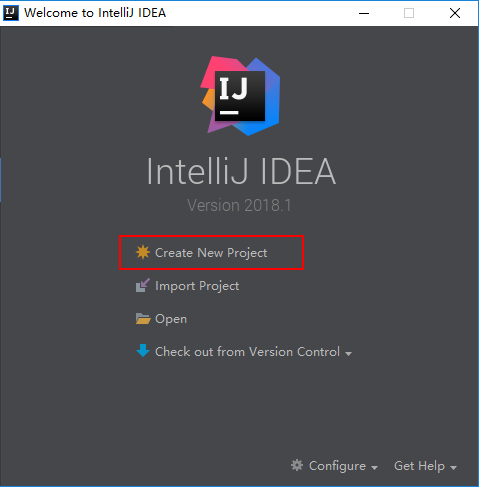
Hadoop生态圈里面的一个资源调度框架，Spark也是可以基于Yarn来计算的。

* Mesos

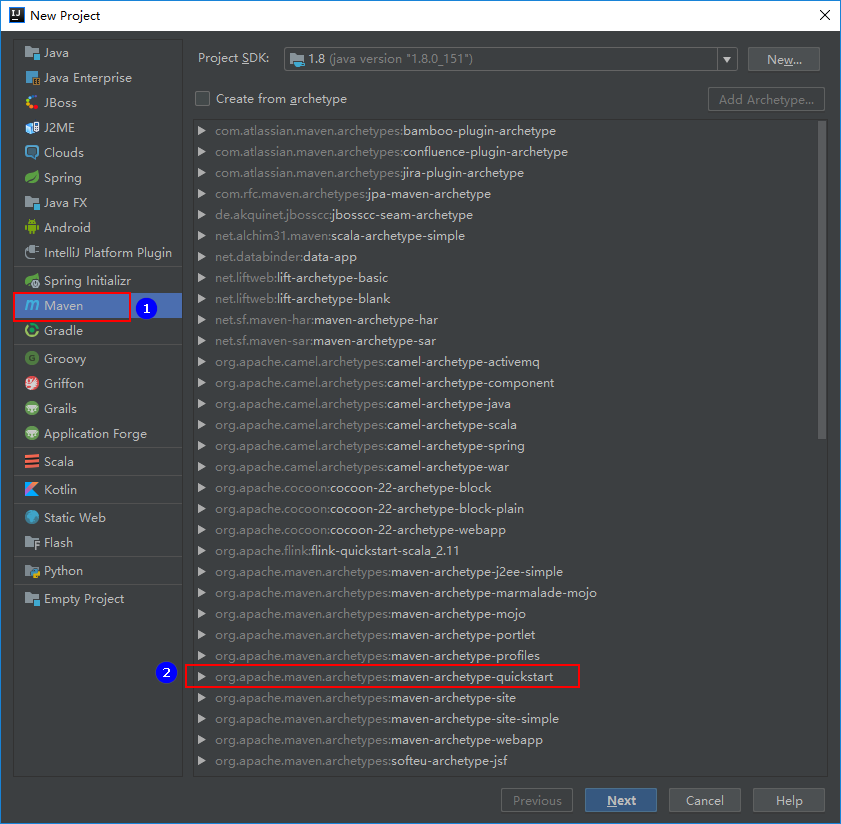
资源调度框架。

* 要基于Yarn来进行资源调度，必须实现AppalicationMaster接口，Spark实现了这个接口，所以可以基于Yarn。

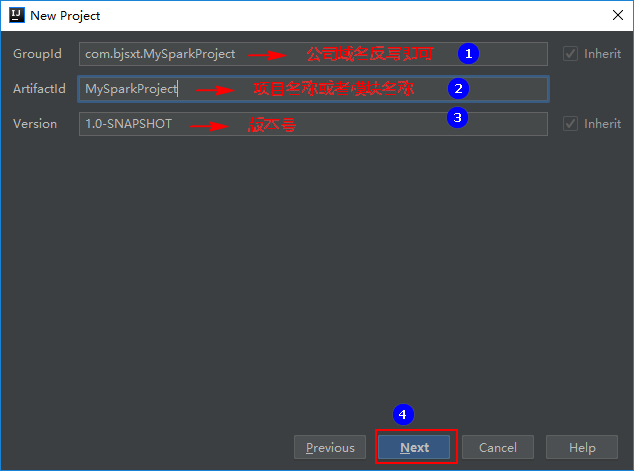
1. **Spark Java-Scala 混编Maven开发**
2. IDEA创建Maven 项目
3. 创建项目



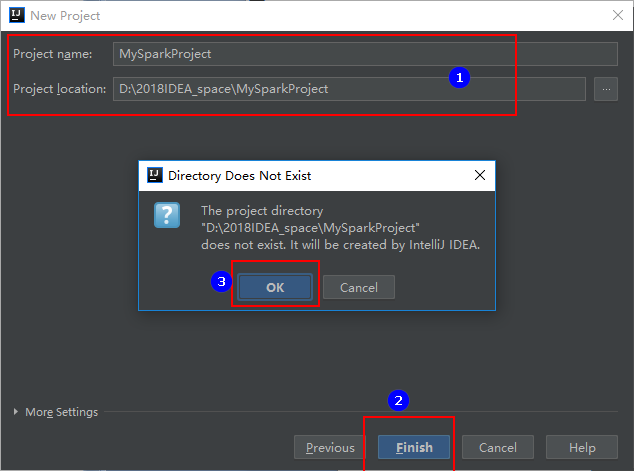
1. 创建选择 maven-archetype-quickstart



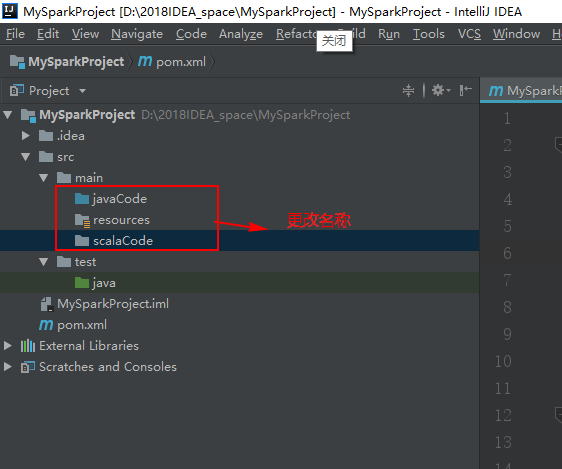
1. 配置名称，点击下一步配置Maven及本地Maven仓库地址。



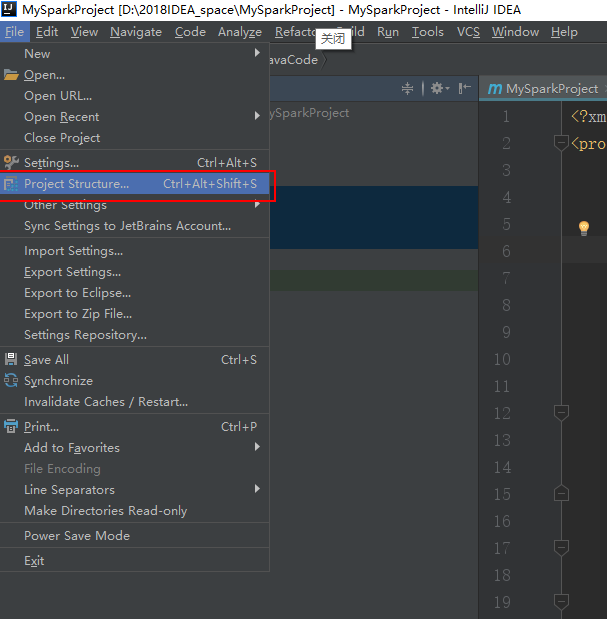
1. 配置项目名称和位置，并创建。

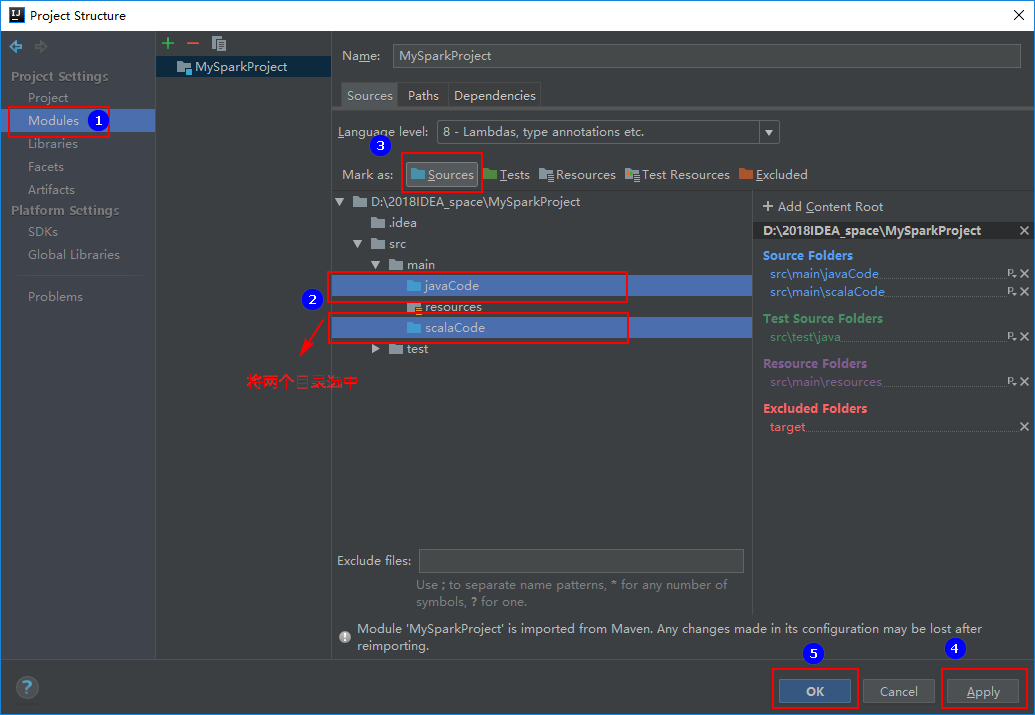


1. 更新替换Maven pom.xml文件,注意groupId，artifactId，version不要更新替换。
2. 在main 目录下创建javaCode和scalaCode 并指定为源目录。名称任意。



将main下的javaCode和scalaCode指定为源目录：





1. 配置的pom.xml文件

附件：

|  |
| --- |
|  |

1. **SparkCore**
2. RDD

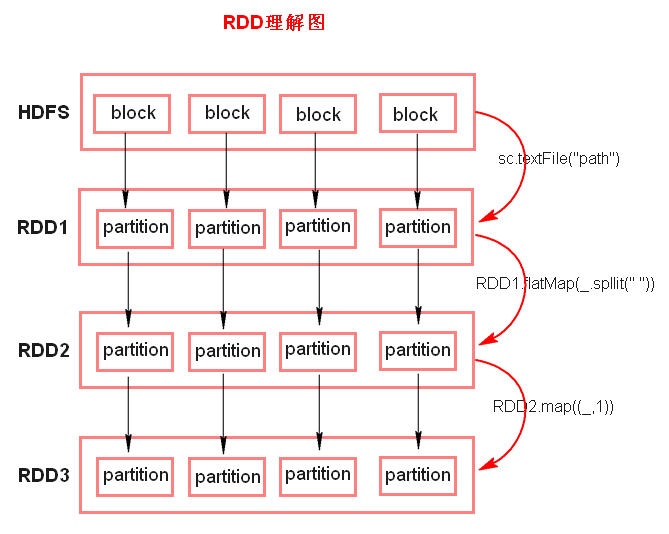
* 概念

RDD(Resilient Distributed Dateset)，弹性分布式数据集。

* RDD的五大特性：

1. RDD是由一系列的partition组成的。
2. 函数是作用在每一个partition（split）上的。
3. RDD之间有一系列的依赖关系。
4. 分区器是作用在K,V格式的RDD上。
5. RDD提供一系列最佳的计算位置。

* RDD理解图:



* 注意：
* textFile方法底层封装的是读取MR读取文件的方式，读取文件之前先split，默认split大小是一个block大小。
* **RDD实际上不存储数据，这里方便理解，暂时理解为存储数据。**
* 什么是K,V格式的RDD?
* 如果RDD里面存储的数据都是二元组对象，那么这个RDD我们就叫做K,V格式的RDD。
* 哪里体现RDD的弹性（容错）？
* partition数量，大小没有限制，体现了RDD的弹性。
* RDD之间依赖关系，可以基于上一个RDD重新计算出RDD。
* 哪里体现RDD的分布式？
* RDD是由Partition组成，partition是分布在不同节点上的。
* RDD提供计算最佳位置，体现了数据本地化。体现了大数据中“计算移动数据不移动”的理念。

1. Spark任务执行原理

|  |
| --- |
|  |

以上图中有四个机器节点，Driver和Worker是启动在节点上的进程，运行在JVM中的进程。

* Driver与集群节点之间有频繁的通信。
* Driver负责任务(tasks)的分发和结果的回收。任务的调度。如果task的计算结果非常大就不要回收了。会造成oom。
* Worker是Standalone资源调度框架里面资源管理的从节点。也是JVM进程。
* Master是Standalone资源调度框架里面资源管理的主节点。也是JVM进程。

1. Spark代码流程
2. 创建SparkConf对象

* 可以设置Application name。
* 可以设置运行模式及资源需求。

1. 创建SparkContext对象
2. 基于Spark的上下文创建一个RDD，对RDD进行处理。
3. 应用程序中要有Action类算子来触发Transformation类算子执行。
4. 关闭Spark上下文对象SparkContext。
5. Transformations转换算子

* 概念：

Transformations类算子是一类算子（函数）叫做转换算子，如map,flatMap,reduceByKey等。Transformations算子是延迟执行，也叫懒加载执行。

* Transformation类算子：

|  |
| --- |
| * filter   过滤符合条件的记录数，true保留，false过滤掉。   * map   将一个RDD中的每个数据项，通过map中的函数映射变为一个新的元素。  特点：输入一条，输出一条数据。   * flatMap   先map后flat。与map类似，每个输入项可以映射为0到多个输出项。   * sample   随机抽样算子，根据传进去的小数按比例进行又放回或者无放回的抽样。   * reduceByKey   将相同的Key根据相应的逻辑进行处理。   * sortByKey/sortBy   作用在K,V格式的RDD上，对key进行升序或者降序排序。 |

1. Action行动算子

* 概念：

Action类算子也是一类算子（函数）叫做行动算子，如foreach,collect，count等。Transformations类算子是延迟执行，Action类算子是触发执行。一个application应用程序中有几个Action类算子执行，就有几个job运行。

* Action类算子

|  |
| --- |
| * count   返回数据集中的元素数。会在结果计算完成后回收到Driver端。   * take(n)   返回一个包含数据集前n个元素的集合。   * first   first=take(1),返回数据集中的第一个元素。   * foreach   循环遍历数据集中的每个元素，运行相应的逻辑。   * collect   将计算结果回收到Driver端。 |

* *思考：一千万条数据量的文件，过滤掉出现次数多的记录，并且其余记录按照出现次数降序排序。*

文件：

|  |
| --- |
|  |

代码：

|  |
| --- |
|  |

1. 控制算子

* 概念：

控制算子有三种，cache,persist,checkpoint，以上算子都可以将RDD持久化，持久化的单位是partition。cache和persist都是懒执行的。必须有一个action类算子触发执行。checkpoint算子不仅能将RDD持久化到磁盘，还能切断RDD之间的依赖关系。

* cache

默认将RDD的数据持久化到内存中。cache是懒执行。

* 注意：chche () = persist()=persist(StorageLevel.Memory\_Only)
* 测试cache文件：

文件：见“NASA\_access\_log\_Aug95”文件。

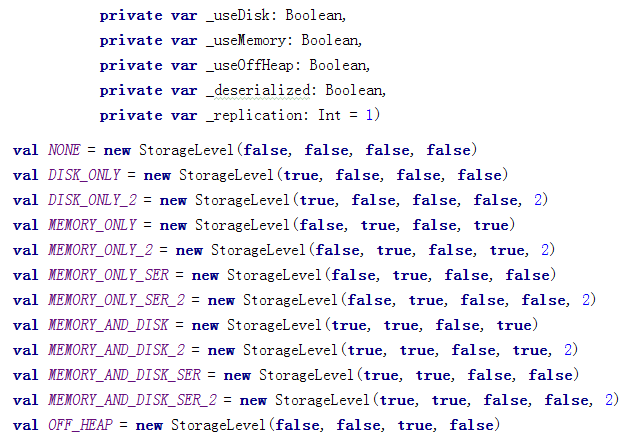
测试代码：

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("CacheTest");  JavaSparkContext jsc = **new** JavaSparkContext(conf);  JavaRDD<String> lines = jsc.textFile("./NASA\_access\_log\_Aug95");  lines = lines.cache();  **long** startTime = System.*currentTimeMillis*();  **long** count = lines.count();  **long** endTime = System.*currentTimeMillis*();  System.***out***.println("共"+count+ "条数据，"+"初始化时间+cache时间+计算时间="+  (endTime-startTime));    **long** countStartTime = System.*currentTimeMillis*();  **long** countrResult = lines.count();  **long** countEndTime = System.*currentTimeMillis*();  System.***out***.println("共"+countrResult+ "条数据，"+"计算时间="+ (countEndTime-  countStartTime));    jsc.stop(); |

* persist：

可以指定持久化的级别。最常用的是MEMORY\_ONLY和MEMORY\_AND\_DISK。”\_2”表示有副本数。

持久化级别如下：



* **cache和persist的注意事项：**

1. cache和persist都是懒执行，必须有一个action类算子触发执行。
2. cache和persist算子的返回值可以赋值给一个变量，在其他job中直接使用这个变量就是使用持久化的数据了。持久化的单位是partition。
3. cache和persist算子后不能立即紧跟action算子。
4. cache和persist算子持久化的数据当applilcation执行完成之后会被清除。

错误：rdd.cache().count() 返回的不是持久化的RDD，而是一个数值了。

* checkpoint

checkpoint将RDD持久化到磁盘，还可以切断RDD之间的依赖关系。checkpoint目录数据当application执行完之后不会被清除。

* checkpoint 的执行原理：

1. 当RDD的job执行完毕后，会从finalRDD从后往前回溯。
2. 当回溯到某一个RDD调用了checkpoint方法，会对当前的RDD做一个标记。
3. Spark框架会自动启动一个新的job，重新计算这个RDD的数据，将数据持久化到HDFS上。

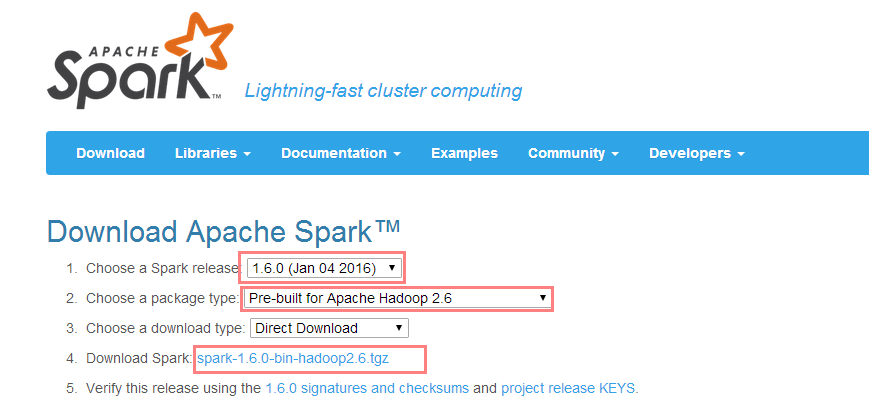
* 优化：对RDD执行checkpoint之前，最好对这个RDD先执行cache，这样新启动的job只需要将内存中的数据拷贝到HDFS上就可以，省去了重新计算这一步。
* 使用：

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("checkpoint");  JavaSparkContext sc = **new** JavaSparkContext(conf);  sc.setCheckpointDir("./checkpoint");  JavaRDD<Integer> parallelize = sc.parallelize(Arrays.*asList*(1,2,3));  parallelize.checkpoint();  parallelize.count();  sc.stop(); |

1. **集群搭建以及测试**
2. 搭建

* Standalone

1).下载安装包，解压



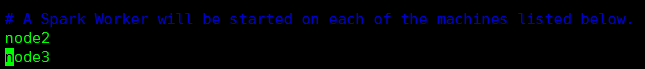


2).改名



3).进入安装包的conf目录下，修改slaves.template文件，添加从节点。保存。





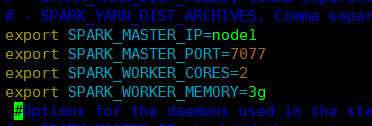
4).修改spark-env.sh

SPARK\_MASTER\_IP:master的ip

SPARK\_MASTER\_PORT:提交任务的端口，默认是7077

SPARK\_WORKER\_CORES：每个worker从节点能够支配的core的个数

SPARK\_WORKER\_MEMORY:每个worker从节点能够支配的内存数



5).同步到其他节点上





6).启动集群

进入sbin目录下，执行当前目录下的./start-all.sh

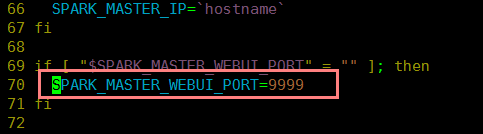


7).搭建客户端

将spark安装包原封不动的拷贝到一个新的节点上，然后，在新的节点上提交任务即可。

注意：

* 8080是Spark WEBUI界面的端口，7077是Spark任务提交的端口。
* 修改master的WEBUI端口：
  + - 修改start-master.sh即可。



* + - 也可以在Master节点上导入临时环境变量，只是作用于之后的程序，重启就无效了。



删除临时环境变量：



* yarn

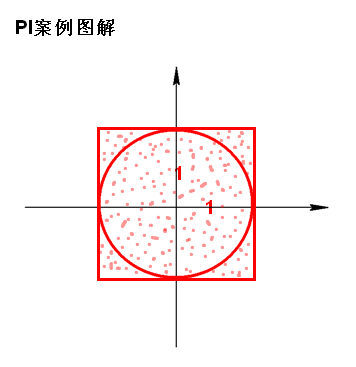
1). 1，2，3，4，5，7步同standalone。

2).在客户端中配置：



1. 测试

PI案例：



Standalone提交命令：

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar 10000 |

YARN提交命令：

|  |
| --- |
| ./spark-submit  --master yarn  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar 10000 |

Spark day02

1. **Standalone模式两种提交任务方式**
2. **Standalone-client提交任务方式**

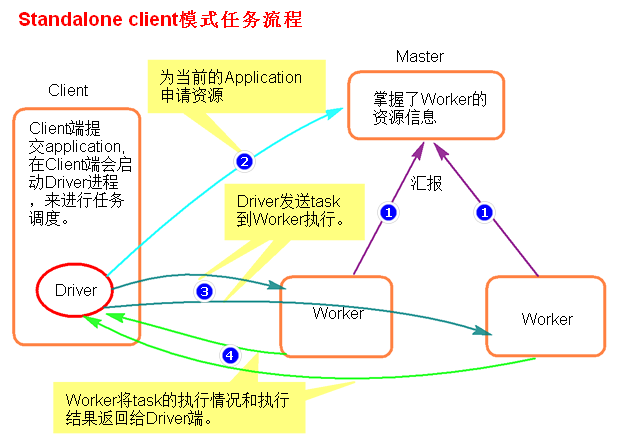
* 提交命令

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  1000 |

或者

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --deploy-mode client  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

* 执行原理图解



* 执行流程

1. client模式提交任务后，会在客户端启动Driver进程。
2. Driver会向Master申请启动Application启动的资源。
3. 资源申请成功，Driver端将task发送到worker端执行。
4. worker将task执行结果返回到Driver端。

* 总结

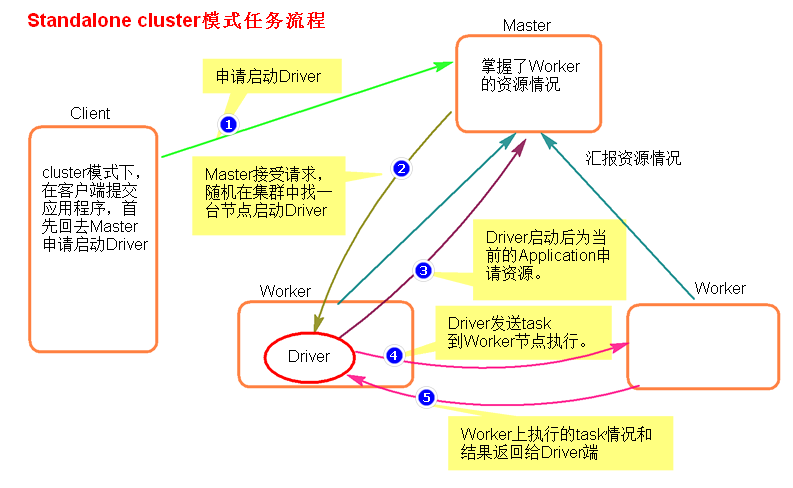
client模式适用于测试调试程序。Driver进程是在客户端启动的，这里的客户端就是指提交应用程序的当前节点。在Driver端可以看到task执行的情况。生产环境下不能使用client模式，是因为：假设要提交100个application到集群运行，Driver每次都会在client端启动，那么就会导致客户端100次网卡流量暴增的问题。

1. **Standalone-cluster提交任务方式**

* 提交命令

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --deploy-mode cluster  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

* 执行原理图解



* 执行流程

1. cluster模式提交应用程序后，会向Master请求启动Driver.
2. Master接受请求，随机在集群一台节点启动Driver进程。
3. Driver启动后为当前的应用程序申请资源。
4. Driver端发送task到worker节点上执行。
5. worker将执行情况和执行结果返回给Driver端。

* 总结

Driver进程是在集群某一台Worker上启动的，在客户端是无法查看task的执行情况的。假设要提交100个application到集群运行,每次Driver会随机在集群中某一台Worker上启动，那么这100次网卡流量暴增的问题就散布在集群上。

* **总结Standalone两种方式提交任务，Driver与集群的通信包括：**

1. Driver负责应用程序资源的申请

2. 任务的分发。

3. 结果的回收。

4. 监控task执行情况。

1. **Yarn模式两种提交任务方式**
2. **yarn-client提交任务方式**

* 提交命令

|  |
| --- |
| ./spark-submit  --master yarn  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

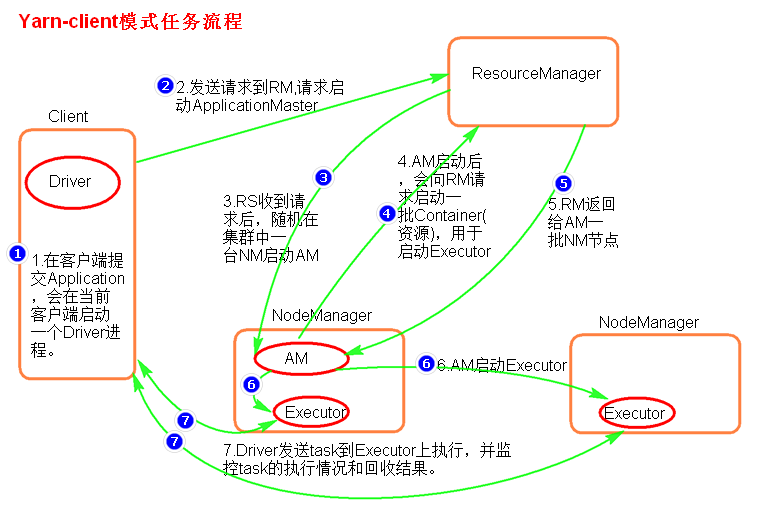
或者

|  |
| --- |
| ./spark-submit  --master yarn–client  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

或者

|  |
| --- |
| ./spark-submit  --master yarn  --deploy-mode client  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

* 执行原理图解



* 执行流程

1. 客户端提交一个Application，在客户端启动一个Driver进程。
2. 应用程序启动后会向RS(ResourceManager)发送请求，启动AM(ApplicationMaster)的资源。
3. RS收到请求，随机选择一台NM(NodeManager)启动AM。这里的NM相当于Standalone中的Worker节点。
4. AM启动后，会向RS请求一批container资源，用于启动Executor.
5. RS会找到一批NM返回给AM,用于启动Executor。
6. AM会向NM发送命令启动Executor。
7. Executor启动后，会反向注册给Driver，Driver发送task到Executor,执行情况和结果返回给Driver端。

* 总结

Yarn-client模式同样是适用于测试，因为Driver运行在本地，Driver会与yarn集群中的Executor进行大量的通信，会造成客户机网卡流量的大量增加.

* **ApplicationMaster的作用：**

1. 为当前的Application申请资源
2. 给NameNode发送消息启动Executor。

注意：ApplicationMaster有launchExecutor和申请资源的功能，并没有作业调度的功能。

1. **yarn-cluster提交任务方式**

* 提交命令

|  |
| --- |
| ./spark-submit  --master yarn  --deploy-mode cluster  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

或者

|  |
| --- |
| ./spark-submit  --master yarn-cluster  --class org.apache.spark.examples.SparkPi ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  100 |

* 执行原理图解



* 执行流程

1. 客户机提交Application应用程序，发送请求到RS(ResourceManager),请求启动AM(ApplicationMaster)。
2. RS收到请求后随机在一台NM(NodeManager)上启动AM（相当于Driver端）。
3. AM启动，AM发送请求到RS，请求一批container用于启动Executor。
4. RS返回一批NM节点给AM。
5. AM连接到NM,发送请求到NM启动Executor。
6. Executor反向注册到AM所在的节点的Driver。Driver发送task到Executor。

* 总结

Yarn-Cluster主要用于生产环境中，因为Driver运行在Yarn集群中某一台nodeManager中，每次提交任务的Driver所在的机器都是随机的，不会产生某一台机器网卡流量激增的现象，缺点是任务提交后不能看到日志。只能通过yarn查看日志。

* **ApplicationMaster的作用：**

1. 为当前的Application申请资源
2. 给NameNode发送消息启动Excutor。
3. 任务调度。

* **停止集群任务命令：yarn application -kill applicationID**

1. **补充部分算子**

transformation

* join,leftOuterJoin,rightOuterJoin,fullOuterJoin

作用在K,V格式的RDD上。根据K进行连接，对（K,V）join(K,W)返回（K,(V,W)）

* join后的分区数与父RDD分区数多的那一个相同。
* union

合并两个数据集。两个数据集的类型要一致。

* 返回新的RDD的分区数是合并RDD分区数的总和。
* intersection

取两个数据集的交集，返回新的RDD与父RDD分区多的一致

* subtract

取两个数据集的差集，结果RDD的分区数与subtract前面的RDD的分区数一致。

* mapPartitions

与map类似，遍历的单位是每个partition上的数据。

* distinct(map+reduceByKey+map)
* cogroup

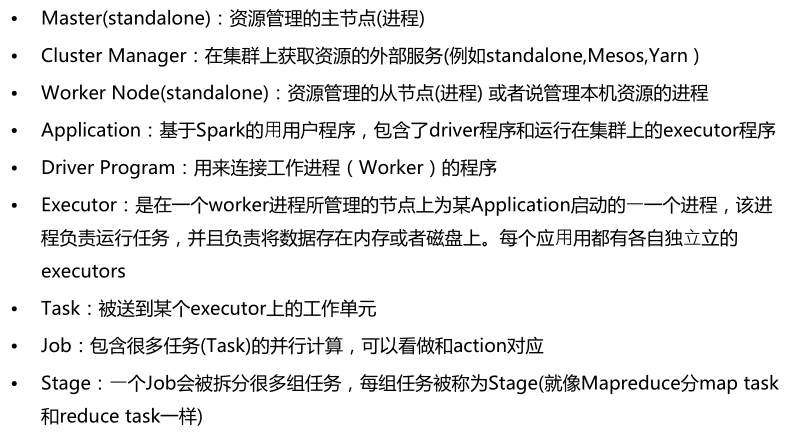
当调用类型（K,V）和（K，W）的数据上时，返回一个数据集（K，（Iterable<V>,Iterable<W>）），子RDD的分区与父RDD多的一致。

action

* foreachPartition

遍历的数据是每个partition的数据。

1. **术语解释**

****

1. **窄依赖和宽依赖**

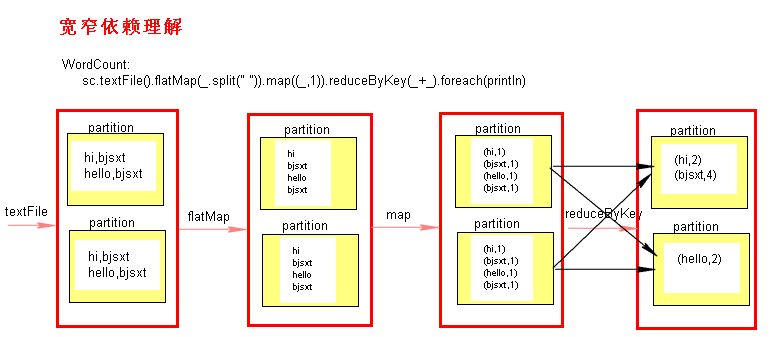
RDD之间有一系列的依赖关系，依赖关系又分为窄依赖和宽依赖。

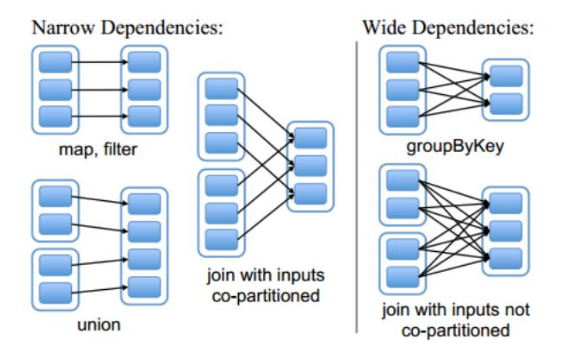
* 窄依赖

父RDD和子RDD partition之间的关系是一对一的。或者父RDD一个partition只对应一个子RDD的partition情况下的父RDD和子RDD partition关系是多对一的。不会有shuffle的产生。

* 宽依赖

父RDD与子RDD partition之间的关系是一对多。会有shuffle的产生。

* 宽窄依赖图理解



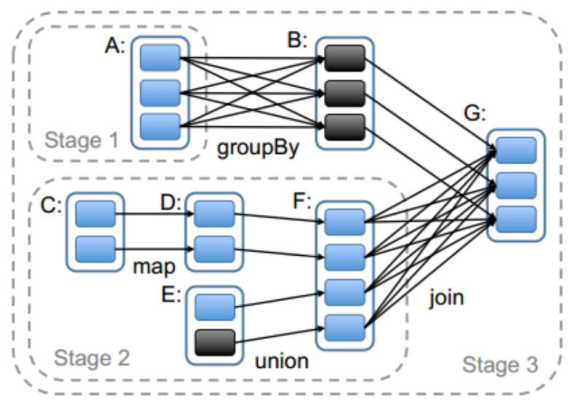
1. **Stage**

Spark任务会根据RDD之间的依赖关系，形成一个DAG有向无环图，DAG会提交给DAGScheduler，DAGScheduler会把DAG划分相互依赖的多个stage，划分stage的依据就是RDD之间的宽窄依赖。遇到宽依赖就划分stage,每个stage包含一个或多个task任务。然后将这些task以taskSet的形式提交给TaskScheduler运行。

stage是由一组并行的task组成。

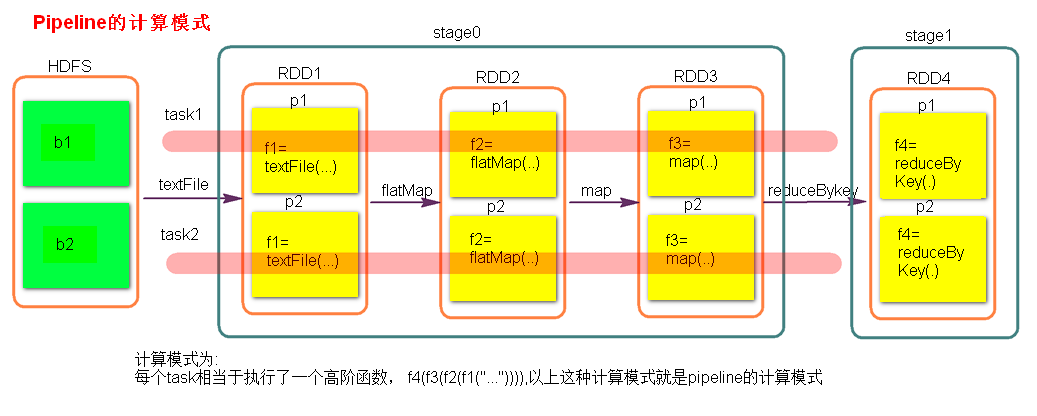
* stage切割规则

切割规则：从后往前，遇到宽依赖就切割stage。



* stage计算模式

pipeline管道计算模式,pipeline只是一种计算思想，模式。



* **数据一直在管道里面什么时候数据会落地？**

1. 对RDD进行持久化。
2. shuffle write的时候。

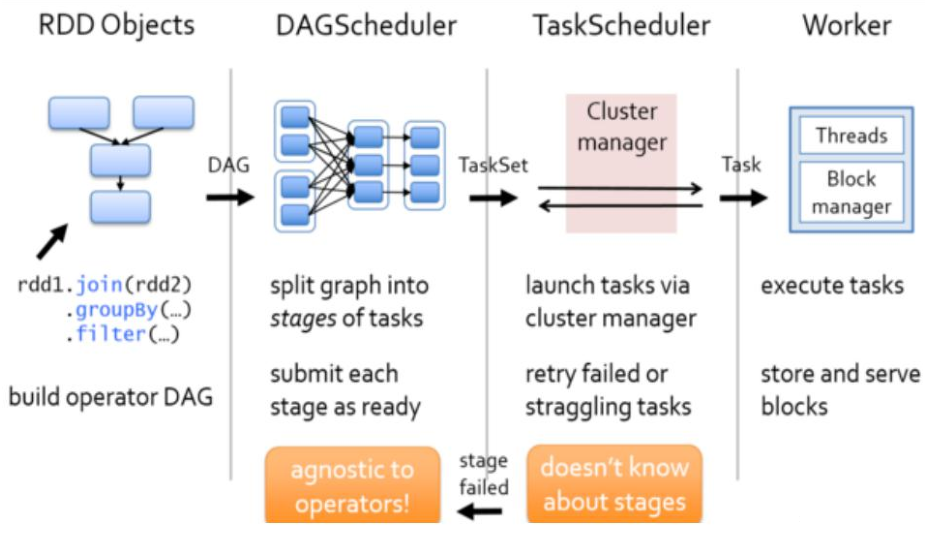
* **Stage的task并行度是由stage的最后一个RDD的分区数来决定的 。**
* **如何改变RDD的分区数？**

例如：reduceByKey(XXX,3),GroupByKey(4)

* 测试验证pipeline计算模式

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("pipeline");  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.parallelize(Array(1,2,3,4))  **val** rdd1 = rdd.map { x => {  println("map--------"+x)  x  }}  **val** rdd2 = rdd1.filter { x => {  println("fliter\*\*\*\*\*\*\*\*"+x)  **true**  } }  rdd2.collect()  sc.stop() |

1. **Spark资源调度和任务调度**

****

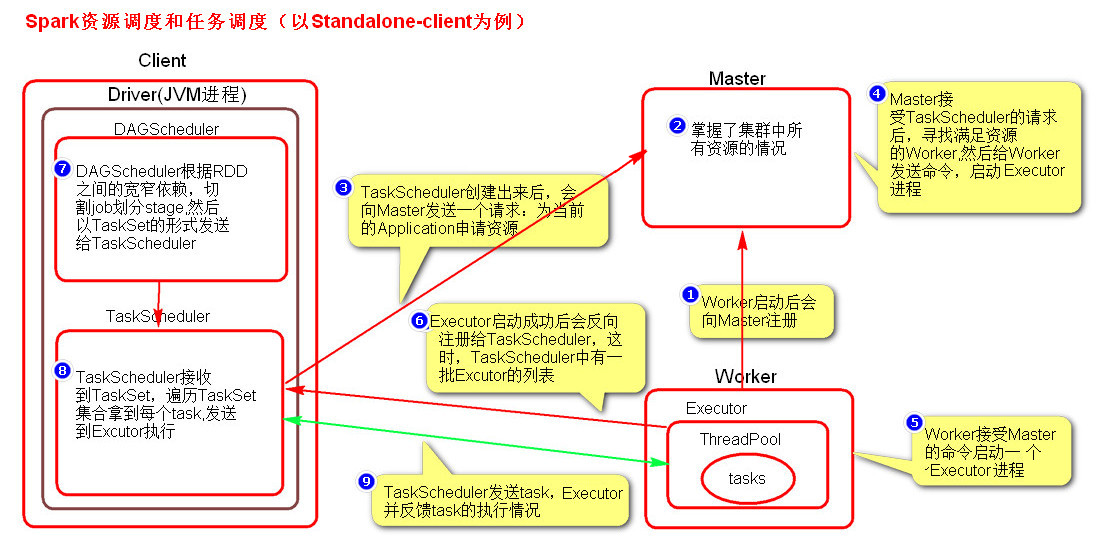
* Spark资源调度和任务调度的流程：

启动集群后，Worker节点会向Master节点汇报资源情况，Master掌握了集群资源情况。当Spark提交一个Application后，根据RDD之间的依赖关系将Application形成一个DAG有向无环图。任务提交后，Spark会在Driver端创建两个对象：DAGScheduler和TaskScheduler，DAGScheduler是任务调度的高层调度器，是一个对象。DAGScheduler的主要作用就是将DAG根据RDD之间的宽窄依赖关系划分为一个个的Stage，然后将这些Stage以TaskSet的形式提交给TaskScheduler（TaskScheduler是任务调度的低层调度器，这里TaskSet其实就是一个集合，里面封装的就是一个个的task任务,也就是stage中的并行度task任务），TaskSchedule会遍历TaskSet集合，拿到每个task后会将task发送到计算节点Executor中去执行（其实就是发送到Executor中的线程池ThreadPool去执行）。task在Executor线程池中的运行情况会向TaskScheduler反馈，当task执行失败时，则由TaskScheduler负责重试，将task重新发送给Executor去执行，默认重试3次。如果重试3次依然失败，那么这个task所在的stage就失败了。stage失败了则由DAGScheduler来负责重试，重新发送TaskSet到TaskSchdeuler，Stage默认重试4次。如果重试4次以后依然失败，那么这个job就失败了。job失败了，Application就失败了。

TaskScheduler不仅能重试失败的task,还会重试straggling（落后，缓慢）task（也就是执行速度比其他task慢太多的task）。如果有运行缓慢的task那么TaskScheduler会启动一个新的task来与这个运行缓慢的task执行相同的处理逻辑。两个task哪个先执行完，就以哪个task的执行结果为准。这就是Spark的推测执行机制。在Spark中推测执行默认是关闭的。推测执行可以通过spark.speculation属性来配置。

注意：

* 对于ETL类型要入数据库的业务要关闭推测执行机制，这样就不会有重复的数据入库。
* 如果遇到数据倾斜的情况，开启推测执行则有可能导致一直会有task重新启动处理相同的逻辑，任务可能一直处于处理不完的状态。
* 图解Spark资源调度和任务调度的流程



* 粗粒度资源申请和细粒度资源申请
* 粗粒度资源申请(Spark）

在Application执行之前，将所有的资源申请完毕，当资源申请成功后，才会进行任务的调度，当所有的task执行完成后，才会释放这部分资源。

优点：在Application执行之前，所有的资源都申请完毕，每一个task直接使用资源就可以了，不需要task在执行前自己去申请资源，task启动就快了，task执行快了，stage执行就快了，job就快了，application执行就快了。

缺点：直到最后一个task执行完成才会释放资源，集群的资源无法充分利用。

* 细粒度资源申请（MapReduce）

Application执行之前不需要先去申请资源，而是直接执行，让job中的每一个task在执行前自己去申请资源，task执行完成就释放资源。

优点：集群的资源可以充分利用。

缺点：task自己去申请资源，task启动变慢，Application的运行就相应的变慢了。

# Spark day03

1. 补充算子

**transformations**

* mapPartitionWithIndex

类似于mapPartitions,除此之外还会携带分区的索引值。

* repartition

增加或减少分区。会产生shuffle。（多个分区分到一个分区不会产生shuffle）

* coalesce

coalesce常用来减少分区，第二个参数是减少分区的过程中是否产生shuffle。

true为产生shuffle，false不产生shuffle。默认是false。

如果coalesce设置的分区数比原来的RDD的分区数还多的话，第二个参数设置为false不会起作用，如果设置成true，效果和repartition一样。即repartition(numPartitions) = coalesce(numPartitions,true)

* groupByKey

作用在K，V格式的RDD上。根据Key进行分组。作用在（K，V），返回（K，Iterable <V>）。

* zip

将两个RDD中的元素（KV格式/非KV格式）变成一个KV格式的RDD,两个RDD的每个分区元素个数必须相同。

* zipWithIndex

该函数将RDD中的元素和这个元素在RDD中的索引号（从0开始）组合成（K,V）对。

**Action**

* countByKey

作用到K,V格式的RDD上，根据Key计数相同Key的数据集元素。

* countByValue

根据数据集每个元素相同的内容来计数。返回相同内容的元素对应的条数。

* reduce

根据聚合逻辑聚合数据集中的每个元素。

1. PV&UV
2. Spark-Submit提交参数

**Options:**

* --master

 MASTER\_URL, 可以是spark://host:port, mesos://host:port, yarn,  yarn-cluster,yarn-client, local

* --deploy-mode

DEPLOY\_MODE, Driver程序运行的地方，client或者cluster,默认是client。

* --class

CLASS\_NAME, 主类名称，含包名

* --jars

逗号分隔的本地JARS, Driver和executor依赖的第三方jar包

* --files

用逗号隔开的文件列表,会放置在每个executor工作目录中

* --conf

spark的配置属性

* --driver-memory

Driver程序使用内存大小（例如：1000M，5G），默认1024M

* --executor-memory

每个executor内存大小（如：1000M，2G），默认1G

**Spark standalone with cluster deploy mode only:**

* --driver-cores

Driver程序的使用core个数（默认为1），仅限于Spark standalone模式

**Spark standalone or Mesos with cluster deploy mode only:**

* --supervise

失败后是否重启Driver，仅限于Spark  alone或者Mesos模式

**Spark standalone and Mesos only:**

* --total-executor-cores

executor使用的总核数，仅限于SparkStandalone、Spark on Mesos模式

**Spark standalone and YARN only:**

* --executor-cores

每个executor使用的core数，Spark on Yarn默认为1，standalone默认为worker上所有可用的core。

**YARN-only:**

* --driver-cores

driver使用的core,仅在cluster模式下，默认为1。

* --queue

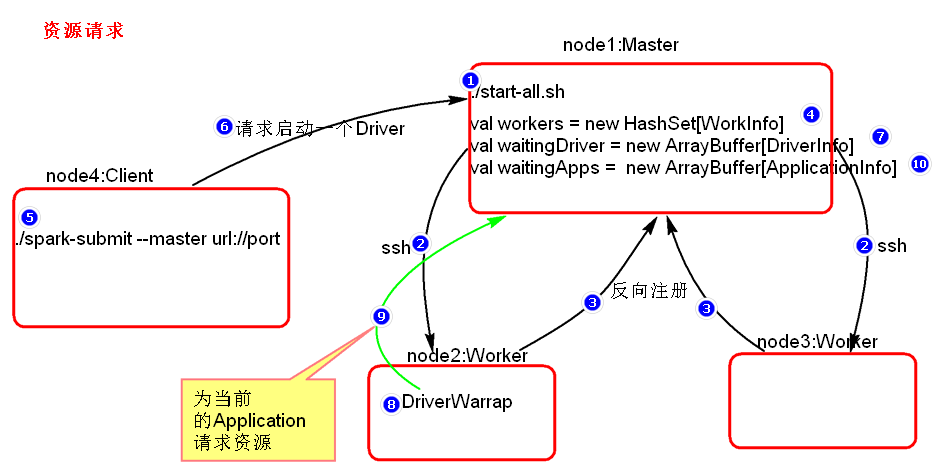
QUEUE\_NAME 指定资源队列的名称,默认：default

* --num-executors

一共启动的executor数量，默认是2个。

1. 资源调度源码分析

* 资源请求简单图



* 资源调度Master路径：



|  |
| --- |
| 路径：spark-1.6.0/core/src/main/scala/org.apache.spark/deploy/Master/Master.scala |

* 提交应用程序，submit的路径：



|  |
| --- |
| 路径：spark-1.6.0/core/src/main/scala/org.apache.spark/ deploy/SparkSubmit.scala |

* 总结：

1. Executor在集群中分散启动，有利于task计算的数据本地化。
2. 默认情况下（提交任务的时候没有设置--executor-cores选项），每一个Worker为当前的Application启动一个Executor,这个Executor会使用这个Worker的所有的cores和1G内存。
3. 如果想在Worker上启动多个Executor，提交Application的时候要加--executor-cores这个选项。
4. 默认情况下没有设置--total-executor-cores,一个Application会使用Spark集群中所有的cores。

* 结论演示

使用Spark-submit提交任务演示。也可以使用spark-shell

1. 默认情况每个worker为当前的Application启动一个Executor，这个Executor使用集群中所有的cores和1G内存。

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  10000 |

1. 在workr上启动多个Executor,设置--executor-cores参数指定每个executor使用的core数量。

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --executor-cores 1  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  10000 |

1. 内存不足的情况下启动core的情况。Spark启动是不仅看core配置参数，也要看配置的core的内存是否够用。

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --executor-cores 1  --executor-memory 3g  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  10000 |

1. --total-executor-cores集群中共使用多少cores

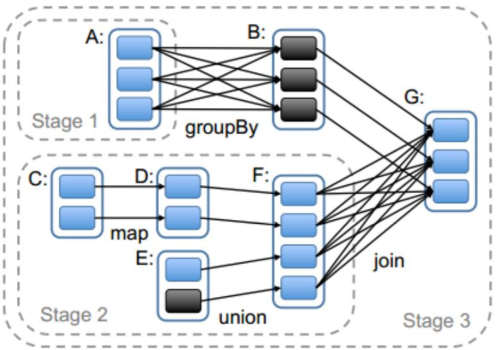
注意：一个进程不能让集群多个节点共同启动。

|  |
| --- |
| ./spark-submit  --master spark://node1:7077  --executor-cores 1  --executor-memory 2g  --total-executor-cores 3  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  10000 |

1. 任务调度源码分析

* Action算子开始分析

任务调度可以从一个Action类算子开始。因为Action类算子会触发一个job的执行。

* 划分stage,以taskSet形式提交任务DAGScheduler 类中getMessingParentStages()方法是切割job划分stage。可以结合以下这张图来分析：

1. 二次排序

|  |
| --- |
| SparkConf sparkConf = **new** SparkConf()  .setMaster("local")  .setAppName("SecondarySortTest");  **final** JavaSparkContext sc = **new** JavaSparkContext(sparkConf);  JavaRDD<String> secondRDD = sc.textFile("secondSort.txt");  JavaPairRDD<SecondSortKey, String> pairSecondRDD = secondRDD.mapToPair(**new** PairFunction<String, SecondSortKey, String>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Tuple2<SecondSortKey, String> call(String line) **throws** Exception {  String[] splited = line.split(" ");  **int** first = Integer.*valueOf*(splited[0]);  **int** second = Integer.*valueOf*(splited[1]);  SecondSortKey secondSortKey = **new** SecondSortKey(first,second);  **return** **new** Tuple2<SecondSortKey, String>(secondSortKey,line);  }  });  pairSecondRDD.sortByKey(**false**).foreach(**new**  VoidFunction<Tuple2<SecondSortKey,String>>() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** **void** call(Tuple2<SecondSortKey, String> tuple) **throws** Exception {  System.***out***.println(tuple.\_2);  }  }); |
| **public** **class** SecondSortKey **implements** Serializable,Comparable<SecondSortKey>{  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **int** first;  **private** **int** second;  **public** **int** getFirst() {  **return** first;  }  **public** **void** setFirst(**int** first) {  **this**.first = first;  }  **public** **int** getSecond() {  **return** second;  }  **public** **void** setSecond(**int** second) {  **this**.second = second;  }  **public** SecondSortKey(**int** first, **int** second) {  **super**();  **this**.first = first;  **this**.second = second;  }  @Override  **public** **int** compareTo(SecondSortKey o1) {  **if**(getFirst() - o1.getFirst() ==0 ){  **return** getSecond() - o1.getSecond();  }**else**{  **return** getFirst() - o1.getFirst();  }  }  } |

1. 分组取topN和topN

|  |
| --- |
| SparkConf conf = **new** SparkConf()  .setMaster("local")  .setAppName("TopOps");  JavaSparkContext sc = **new** JavaSparkContext(conf);  JavaRDD<String> linesRDD = sc.textFile("scores.txt");  JavaPairRDD<String, Integer> pairRDD = linesRDD.mapToPair(**new** PairFunction<String, String, Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Tuple2<String, Integer> call(String str) **throws** Exception {  String[] splited = str.split("\t");  String clazzName = splited[0];  Integer score = Integer.*valueOf*(splited[1]);  **return** **new** Tuple2<String, Integer> (clazzName,score);  }  });  pairRDD.groupByKey().foreach(**new**  VoidFunction<Tuple2<String,Iterable<Integer>>>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** **void** call(Tuple2<String, Iterable<Integer>> tuple) **throws** Exception {  String clazzName = tuple.\_1;  Iterator<Integer> iterator = tuple.\_2.iterator();    Integer[] top3 = **new** Integer[3];    **while** (iterator.hasNext()) {  Integer score = iterator.next();  **for** (**int** i = 0; i < top3.length; i++) {  **if**(top3[i] == **null**){  top3[i] = score;  **break**;  }**else** **if**(score > top3[i]){  **for** (**int** j = 2; j > i; j--) {  top3[j] = top3[j-1];  }  top3[i] = score;  **break**;  }  }  }  System.***out***.println("class Name:"+clazzName);  **for**(Integer sscore : top3){  System.***out***.println(sscore);  }  }  }); |

1. SparkShell的使用

* 概念：

SparkShell是Spark自带的一个快速原型开发工具，也可以说是Spark的scala REPL(Read-Eval-Print-Loop),即交互式shell。支持使用scala语言来进行Spark的交互式编程。

* 使用:

启动Standalone集群，./start-all.sh

在客户端上启动spark-shell:

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

启动hdfs，创建目录spark/test,上传文件wc.txt

|  |
| --- |
| 启动hdfs集群：  start-all.sh  创建目录：  hdfs dfs -mkdir -p /spark/test  上传wc.txt  hdfs dfs -put /root/test/wc.txt /spark/test/  wc附件： |

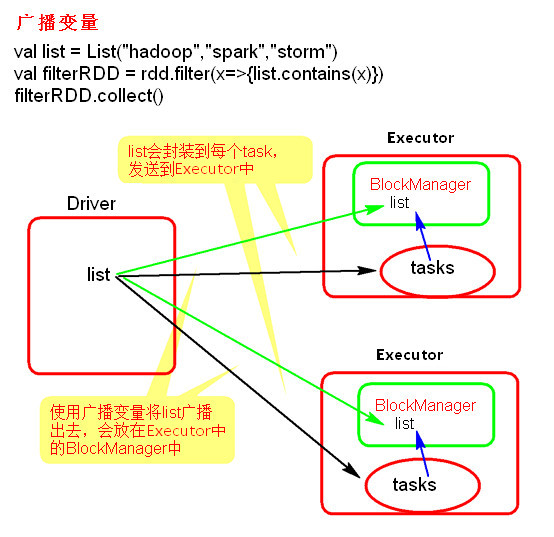
运行wordcount

|  |
| --- |
| sc.textFile("hdfs://node1:9000/spark/test/wc.txt")  .flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_).foreach(println) |

# Spark day04

1. **广播变量和累加器**
2. 广播变量

* 广播变量理解图



* 广播变量使用

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("brocast")  **val** sc = **new** SparkContext(conf)  **val** list = List("hello xasxt")  **val broadCast = sc.broadcast(list)**  **val** lineRDD = sc.textFile("./words.txt")  lineRDD.filter { x => **broadCast.value**.contains(x) }.foreach { println}  sc.stop() |

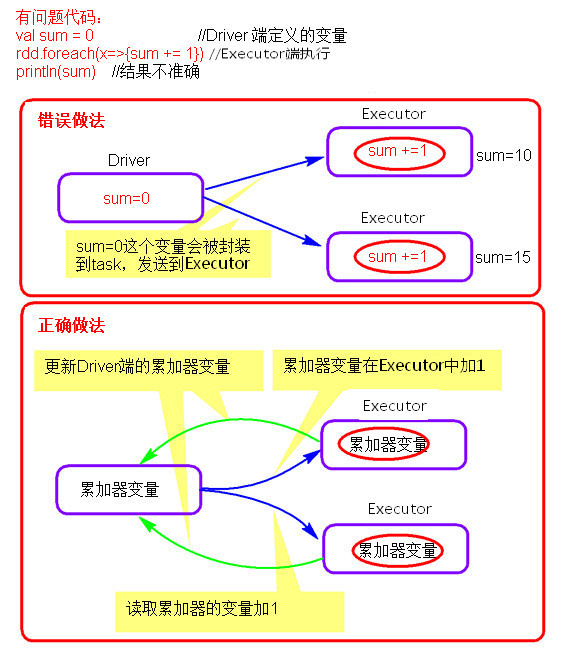
* 注意事项
* 能不能将一个RDD使用广播变量广播出去？

不能，因为RDD是不存储数据的。可以将RDD的结果广播出去。

* 广播变量只能在Driver端定义，不能在Executor端定义。
* 在Driver端可以修改广播变量的值，在Executor端无法修改广播变量的值。

1. 累加器

* 累加器理解图



* 累加器的使用

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("accumulator")  **val** sc = **new** SparkContext(conf)  **val** accumulator = sc.accumulator(0)  sc.textFile("./words.txt").foreach { x =>{accumulator.add(1)}}  println(accumulator.value)  sc.stop() |

* 注意事项
* 累加器在Driver端定义赋初始值，累加器只能在Driver端读取，在Excutor端更新。

1. **SparkUI**
2. SparkUI界面介绍

可以指定提交Application的名称

|  |
| --- |
| ./spark-shell --master spark://node1:7077 --name myapp |

1. 配置historyServer

* 临时配置，对本次提交的应用程序起作用

|  |
| --- |
| ./spark-shell --master spark://node1:7077  --name myapp1  **--conf spark.eventLog.enabled=true**  **--conf spark.eventLog.dir=hdfs://node1:9000/spark/test** |

停止程序，在Web Ui中Completed Applications对应的ApplicationID中能查看history。

* spark-default.conf配置文件中配置HistoryServer，对所有提交的Application都起作用

在客户端节点，进入../spark-1.6.0/conf/ spark-defaults.conf最后加入:

|  |
| --- |
| **//开启记录事件日志的功能**  **spark.eventLog.enabled true**  **//设置事件日志存储的目录**  **spark.eventLog.dir hdfs://node1:9000/spark/test**  **//设置HistoryServer加载事件日志的位置**  **spark.history.fs.logDirectory hdfs://node1:9000/spark/test**  **//日志优化选项,压缩日志**  **spark.eventLog.compress true** |

启动HistoryServer：

|  |
| --- |
| ./start-history-server.sh |

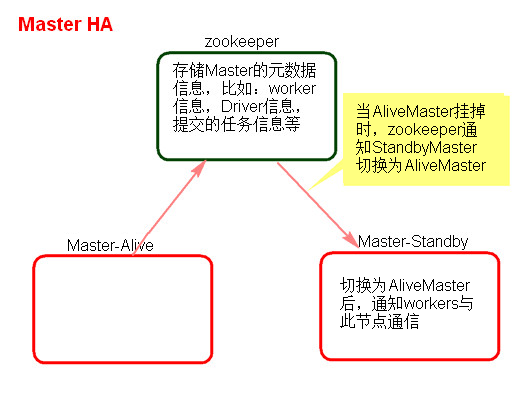
访问HistoryServer：node4:18080,之后所有提交的应用程序运行状况都会被记录。

1. **Master HA**
2. Master的高可用原理

Standalone集群只有一个Master，如果Master挂了就无法提交应用程序，需要给Master进行高可用配置，Master的高可用可以使用fileSystem(文件系统)和zookeeper（分布式协调服务）。

fileSystem只有存储功能，可以存储Master的元数据信息，用fileSystem搭建的Master高可用，在Master失败时，需要我们手动启动另外的备用Master，这种方式不推荐使用。

zookeeper有选举和存储功能，可以存储Master的元素据信息，使用zookeeper搭建的Master高可用，当Master挂掉时，备用的Master会自动切换，推荐使用这种方式搭建Master的HA。



1. Master高可用搭建
2. 在Spark Master节点上配置主Master，配置spark-env.sh

|  |
| --- |
| **export SPARK\_DAEMON\_JAVA\_OPTS="**  **-Dspark.deploy.recoveryMode=ZOOKEEPER**  **-Dspark.deploy.zookeeper.url=node3:2181,node4:2181,node5:2181**  **-Dspark.deploy.zookeeper.dir=/sparkmaster0821"** |



1. 发送到其他worker节点上





1. 找一台节点（非主Master节点）配置备用 Master,修改spark-env.sh配置节点上的MasterIP



1. 启动集群之前启动zookeeper集群：

|  |
| --- |
| ../zkServer.sh start |

1. 启动spark Standalone集群，启动备用Master
2. 打开主Master和备用Master WebUI页面，观察状态。
3. 注意点

* 主备切换过程中不能提交Application。
* 主备切换过程中不影响已经在集群中运行的Application。因为Spark是粗粒度资源调度。

1. 测试验证

提交SparkPi程序，kill主Master观察现象。

|  |
| --- |
| ./spark-submit  **--master spark://node1:7077,node2:7077**  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  10000 |

1. **SparkShuffle**
2. SparkShuffle概念

reduceByKey会将上一个RDD中的每一个key对应的所有value聚合成一个value，然后生成一个新的RDD，元素类型是<key,value>对的形式，这样每一个key对应一个聚合起来的value。

**问题：**聚合之前，每一个key对应的value不一定都是在一个partition中，也不太可能在同一个节点上，因为RDD是分布式的弹性的数据集，RDD的partition极有可能分布在各个节点上。

**如何聚合？**

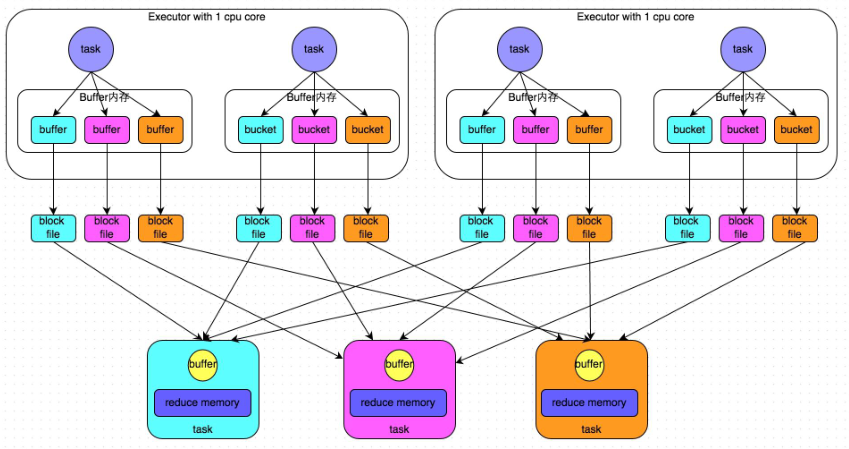
**– Shuffle Write：**上一个stage的每个map task就必须保证将自己处理的当前分区的数据相同的key写入一个分区文件中，可能会写入多个不同的分区文件中。

**– Shuffle Read：**reduce task就会从上一个stage的所有task所在的机器上寻找属于己的那些分区文件，这样就可以保证每一个key所对应的value都会汇聚到同一个节点上去处理和聚合。

Spark中有两种Shuffle管理类型，HashShufflManager和SortShuffleManager，Spark1.2之前是HashShuffleManager， Spark1.2引入SortShuffleManager,在Spark 2.0+版本中已经将HashShuffleManager丢弃。

1. HashShuffleManager
2. 普通机制

* 普通机制示意图



* 执行流程

1. 每一个map task将不同结果写到不同的buffer中，每个buffer的大小为32K。buffer起到数据缓存的作用。
2. 每个buffer文件最后对应一个磁盘小文件。
3. reduce task来拉取对应的磁盘小文件。

* 总结

1. .map task的计算结果会根据分区器（默认是hashPartitioner）来决定写入到哪一个磁盘小文件中去。ReduceTask会去Map端拉取相应的磁盘小文件。
2. .产生的磁盘小文件的个数：

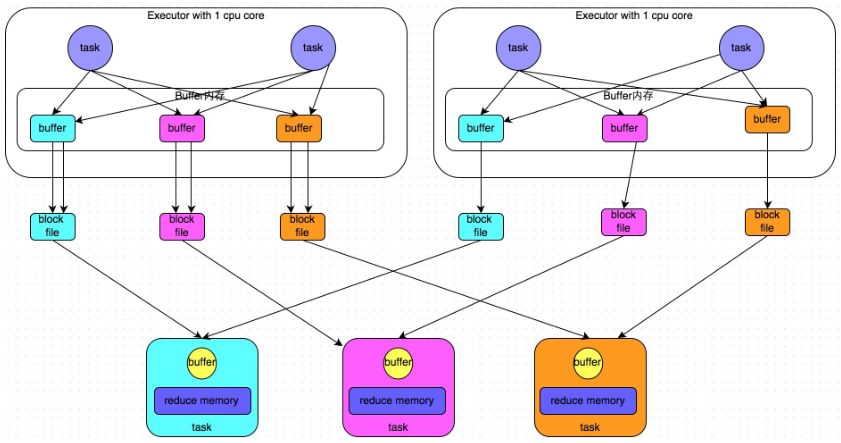
M（map task的个数）\*R（reduce task的个数）

* 存在的问题

产生的磁盘小文件过多，会导致以下问题：

1. 在Shuffle Write过程中会产生很多写磁盘小文件的对象。
2. 在Shuffle Read过程中会产生很多读取磁盘小文件的对象。
3. 在JVM堆内存中对象过多会造成频繁的gc,gc还无法解决运行所需要的内存 的话，就会OOM。
4. 在数据传输过程中会有频繁的网络通信，频繁的网络通信出现通信故障的可能性大大增加，一旦网络通信出现了故障会导致shuffle file cannot find 由于这个错误导致的task失败，TaskScheduler不负责重试，由DAGScheduler负责重试Stage。
5. 合并机制

* 合并机制示意图

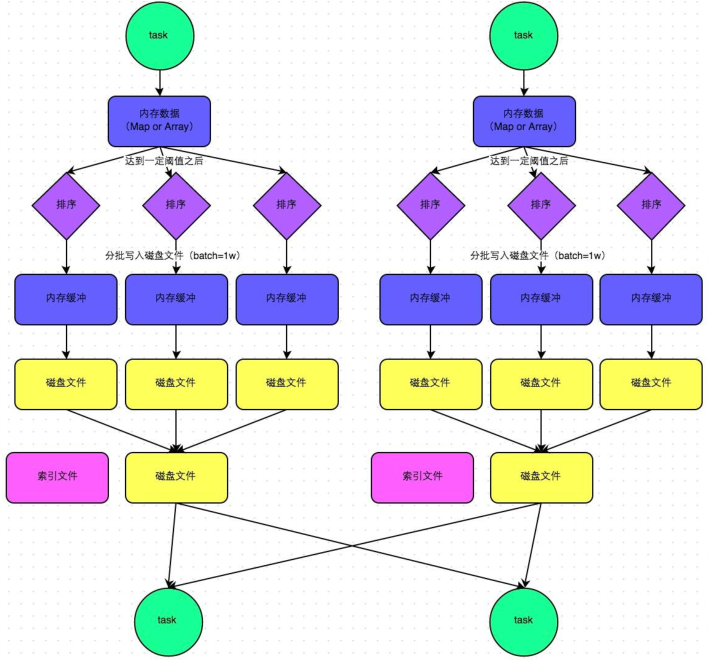


* 总结

产生磁盘小文件的个数：C(core的个数)\*R（reduce的个数）

1. SortShuffleManager
2. 普通机制

* 普通机制示意图



* 执行流程

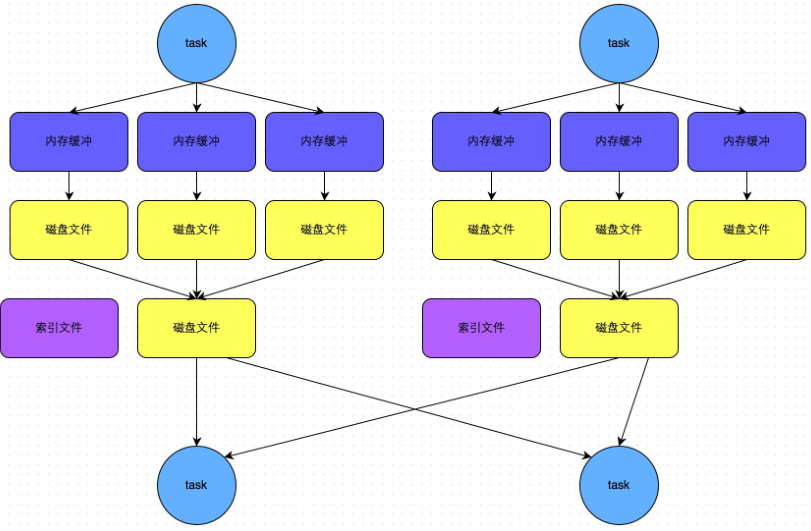
1. map task 的计算结果会写入到一个内存数据结构里面，内存数据结构默认是5M
2. 在shuffle的时候会有一个定时器，不定期的去估算这个内存结构的大小，当内存结构中的数据超过5M时，比如现在内存结构中的数据为5.01M，那么他会申请5.01\*2-5=5.02M内存给内存数据结构。
3. 如果申请成功不会进行溢写，如果申请不成功，这时候会发生溢写磁盘。
4. 在溢写之前内存结构中的数据会进行排序分区
5. 然后开始溢写磁盘，写磁盘是以batch的形式去写，一个batch是1万条数据，
6. map task执行完成后，会将这些磁盘小文件合并成一个大的磁盘文件，同时生成一个索引文件。
7. reduce task去map端拉取数据的时候，首先解析索引文件，根据索引文件再去拉取对应的数据。

* 总结

产生磁盘小文件的个数： 2\*M（map task的个数）

1. bypass机制

* bypass机制示意图



* 总结

1. .bypass运行机制的触发条件如下：

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

1. .产生的磁盘小文件为：2\*M（map task的个数）
2. Shuffle文件寻址
3. MapOutputTracker

MapOutputTracker是Spark架构中的一个模块，是一个主从架构。管理磁盘小文件的地址。

* MapOutputTrackerMaster是主对象，存在于Driver中。
* MapOutputTrackerWorker是从对象，存在于Excutor中。

1. BlockManager

BlockManager块管理者，是Spark架构中的一个模块，也是一个主从架构。

* BlockManagerMaster,主对象，存在于Driver中。

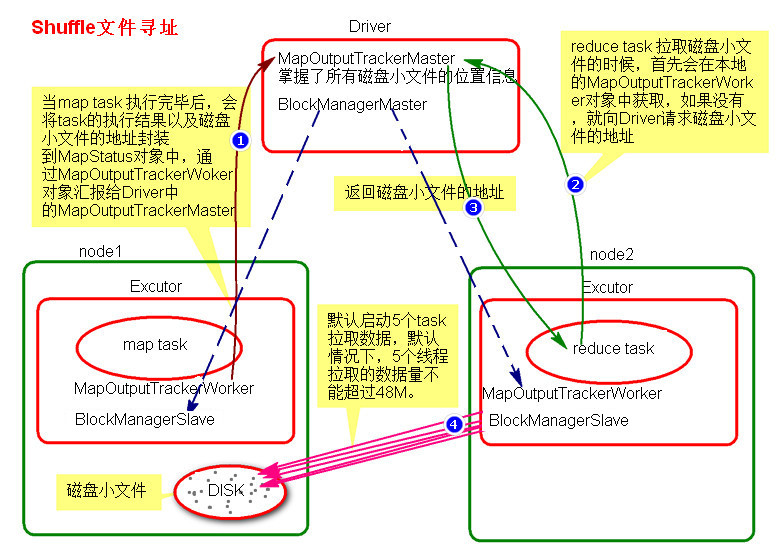
BlockManagerMaster会在集群中有用到广播变量和缓存数据或者删除缓存数据的时候，通知BlockManagerSlave传输或者删除数据。

* BlockManagerSlave，从对象，存在于Excutor中。

BlockManagerSlave会与BlockManagerSlave之间通信。

* 无论在Driver端的BlockManager还是在Excutor端的BlockManager都含有三个对象：

1. DiskStore:负责磁盘的管理。
2. MemoryStore：负责内存的管理。
3. BlockTransferService:负责数据的传输。
4. Shuffle文件寻址图



1. Shuffle文件寻址流程
2. 当map task执行完成后，会将task的执行情况和磁盘小文件的地址封装到MpStatus对象中，通过MapOutputTrackerWorker对象向Driver中的MapOutputTrackerMaster汇报。
3. 在所有的map task执行完毕后，Driver中就掌握了所有的磁盘小文件的地址。
4. 在reduce task执行之前，会通过Excutor中MapOutPutTrackerWorker向Driver端的MapOutputTrackerMaster获取磁盘小文件的地址。
5. 获取到磁盘小文件的地址后，会通过BlockManager连接数据所在节点，然后通过BlockTransferService进行数据的传输。
6. BlockTransferService默认启动5个task去节点拉取数据。默认情况下，5个task拉取数据量不能超过48M。
7. **Spark内存管理**

Spark执行应用程序时，Spark集群会启动Driver和Executor两种JVM进程，Driver负责创建SparkContext上下文，提交任务，task的分发等。Executor负责task的计算任务，并将结果返回给Driver。同时需要为需要持久化的RDD提供储存。Driver端的内存管理比较简单，这里所说的Spark内存管理针对Executor端的内存管理。

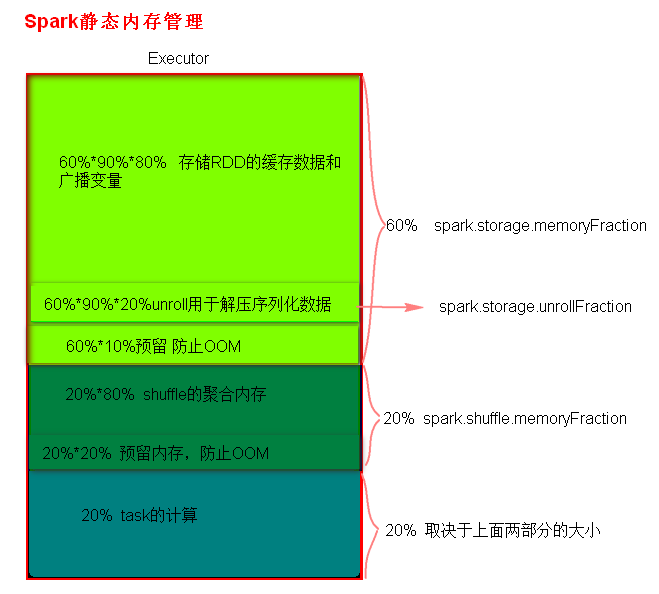
Spark内存管理分为静态内存管理和统一内存管理，Spark1.6之前使用的是静态内存管理，Spark1.6之后引入了统一内存管理。

**静态内存管理**中存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置。

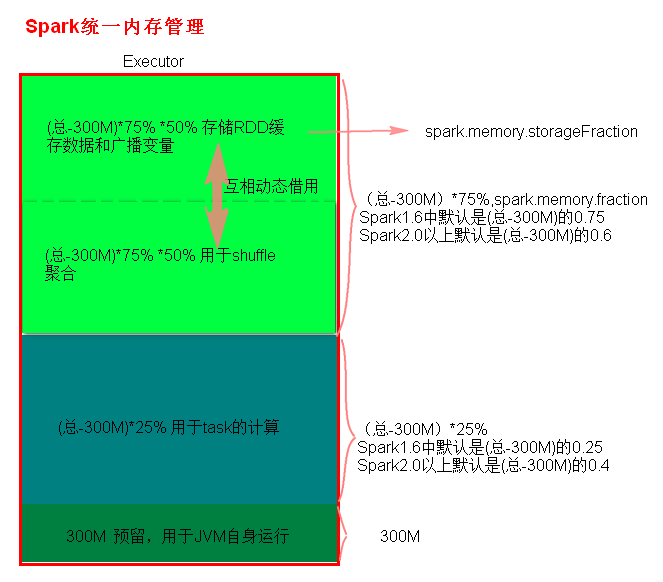
**统一内存管理**与静态内存管理的区别在于储存内存和执行内存共享同一块空间，可以互相借用对方的空间。

Spark1.6以上版本默认使用的是统一内存管理，可以通过参数spark.memory.useLegacyMode 设置为true(默认为false)使用静态内存管理。

1. 静态内存管理分布图

****

1. 统一内存管理分布图

****

1. reduce 中OOM如何处理？
2. 减少每次拉取的数据量
3. 提高shuffle聚合的内存比例
4. 提高Excutor的总内存
5. **Shuffle调优**
6. SparkShuffle调优配置项如何使用？
7. 在代码中,不推荐使用，硬编码。

new SparkConf().set(“spark.shuffle.file.buffer”,”64”)

1. 在提交spark任务的时候，推荐使用。

spark-submit --conf spark.shuffle.file.buffer=64 –conf ….

1. 在conf下的spark-default.conf配置文件中,不推荐，因为是写死后所有应用程序都要用。
2. Shuffle调优附件

|  |
| --- |
|  |

# Spark day05

1. Shark

Shark是基于Spark计算框架之上且兼容Hive语法的SQL执行引擎，由于底层的计算采用了Spark，性能比MapReduce的Hive普遍快2倍以上，当数据全部load在内存的话，将快10倍以上，因此Shark可以作为交互式查询应用服务来使用。除了基于Spark的特性外，Shark是完全兼容Hive的语法，表结构以及UDF函数等，已有的HiveSql可以直接进行迁移至Shark上Shark底层依赖于Hive的解析器，查询优化器，但正是由于SHark的整体设计架构对Hive的依赖性太强，难以支持其长远发展，比如不能和Spark的其他组件进行很好的集成，无法满足Spark的一栈式解决大数据处理的需求。

1. SparkSQL
2. SparkSQL介绍

Hive是Shark的前身，Shark是SparkSQL的前身,SparkSQL产生的根本原因是其完全脱离了Hive的限制。

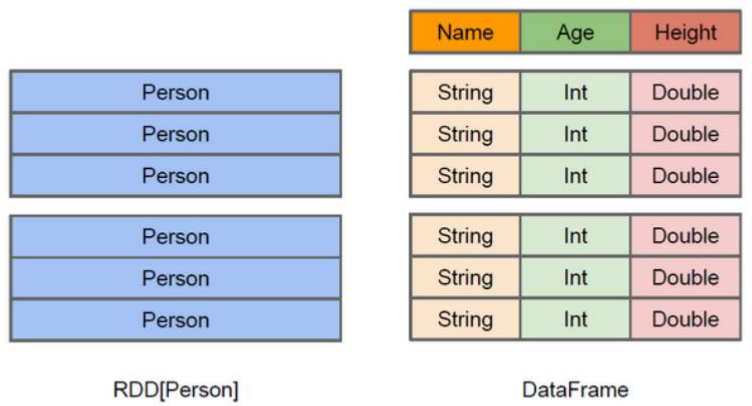
* SparkSQL支持查询原生的RDD。 RDD是Spark平台的核心概念，是Spark能够高效的处理大数据的各种场景的基础。
* 能够在Scala中写SQL语句。支持简单的SQL语法检查，能够在Scala中写Hive语句访问Hive数据，并将结果取回作为RDD使用。

1. Spark on Hive和Hive on Spark

Spark on Hive： Hive只作为储存角色，Spark负责sql解析优化，执行。

Hive on Spark：Hive即作为存储又负责sql的解析优化，Spark负责执行。

1. DataFrame

DataFrame也是一个分布式数据容器。与RDD类似，然而DataFrame更像传统数据库的二维表格，除了数据以外，还掌握数据的结构信息，即schema。同时，与Hive类似，DataFrame也支持嵌套数据类型（struct、array和map）。从API易用性的角度上 看， DataFrame API提供的是一套高层的关系操作，比函数式的RDD API要更加友好，门槛更低。

DataFrame的底层封装的是RDD，只不过RDD的泛型是Row类型。

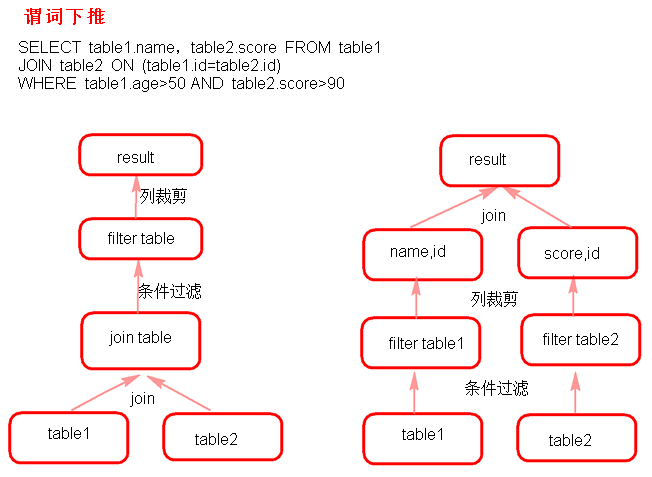
1. SparkSQL的数据源

SparkSQL的数据源可以是JSON类型的字符串，JDBC,Parquent,Hive，HDFS等。

1. SparkSQL底层架构

首先拿到sql后解析一批未被解决的逻辑计划，再经过分析得到分析后的逻辑计划，再经过一批优化规则转换成一批最佳优化的逻辑计划，再经过SparkPlanner的策略转化成一批物理计划，随后经过消费模型转换成一个个的Spark任务执行。

1. 谓词下推（predicate Pushdown）



1. 创建DataFrame的几种方式
2. 读取json格式的文件创建DataFrame

注意：

* json文件中的json数据不能嵌套json格式数据。
* DataFrame是一个一个Row类型的RDD，df.rdd()/df.javaRdd()。
* 可以两种方式读取json格式的文件。
* df.show()默认显示前20行数据。
* DataFrame原生API可以操作DataFrame（不方便）。
* 注册成临时表时，表中的列默认按ascii顺序显示列。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("jsonfile");  SparkContext sc = **new** SparkContext(conf);    //创建sqlContext  SQLContext sqlContext = **new** SQLContext(sc);    /\*\*  \* DataFrame的底层是一个一个的RDD RDD的泛型是Row类型。  \* 以下两种方式都可以读取json格式的文件  \*/  **DataFrame df = sqlContext.read().format("json").load("sparksql/json");**  **// DataFrame df2 = sqlContext.read().json("sparksql/json.txt");**  // df2.show();  /\*\*  \* DataFrame转换成RDD  \*/  **RDD<Row> rdd = df.rdd();**  /\*\*  \* 显示 DataFrame中的内容，默认显示前20行。如果现实多行要指定多少行show(行数)  \* 注意：当有多个列时，显示的列先后顺序是按列的ascii码先后显示。  \*/  // df.show();  /\*\*  \* 树形的形式显示schema信息  \*/  **df.printSchema();**    /\*\*  \* dataFram自带的API 操作DataFrame  \*/  //select name from table  // df.select("name").show();  //select name age+10 as addage from table  df.select(df.col("name"),df.col("age").plus(10).alias("addage")).show();  //select name ,age from table where age>19  df.select(df.col("name"),df.col("age")).where(df.col("age").gt(19)).show();  //select count(\*) from table group by age  df.groupBy(df.col("age")).count().show();    /\*\*  \* 将DataFrame注册成临时的一张表，这张表临时注册到内存中，是逻辑上的表，不会雾化到磁盘  \*/  **df.registerTempTable("jtable");**    DataFrame sql = sqlContext.sql("select age,count(1) from jtable group by age");  DataFrame sql2 = sqlContext.sql("select \* from jtable");    sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("jsonfile")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val df = sqlContext.read.json("sparksql/json")**  **//val df1 = sqlContext.read.format("json").load("sparksql/json")**  **df.show()**  **df.printSchema()**  //select \* from table  df.select(df.col("name")).show()  //select name from table where age>19  df.select(df.col("name"),df.col("age")).where(df.col("age").gt(19)).show()  //select count(\*) from table group by age  df.groupBy(df.col("age")).count().show();    /\*\*  \* 注册临时表  \*/  **df.registerTempTable("jtable")**  **val** result = sqlContext.sql("select \* from jtable")  result.show()  sc.stop() |

1. 通过json格式的RDD创建DataFrame

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("jsonRDD");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> nameRDD = sc.parallelize(Arrays.*asList*(  "{\"name\":\"zhangsan\",\"age\":\"18\"}",  "{\"name\":\"lisi\",\"age\":\"19\"}",  "{\"name\":\"wangwu\",\"age\":\"20\"}"  ));  JavaRDD<String> scoreRDD = sc.parallelize(Arrays.*asList*(  "{\"name\":\"zhangsan\",\"score\":\"100\"}",  "{\"name\":\"lisi\",\"score\":\"200\"}",  "{\"name\":\"wangwu\",\"score\":\"300\"}"  ));  **DataFrame namedf = sqlContext.read().json(nameRDD);**  **DataFrame scoredf = sqlContext.read().json(scoreRDD);**  namedf.registerTempTable("name");  scoredf.registerTempTable("score");  DataFrame result = sqlContext.sql("select name.name,name.age,score.score from name,score where name.name = score.name");  result.show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("jsonrdd")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** nameRDD = sc.makeRDD(Array(  "{\"name\":\"zhangsan\",\"age\":18}",  "{\"name\":\"lisi\",\"age\":19}",  "{\"name\":\"wangwu\",\"age\":20}"  ))  **val** scoreRDD = sc.makeRDD(Array(  "{\"name\":\"zhangsan\",\"score\":100}",  "{\"name\":\"lisi\",\"score\":200}",  "{\"name\":\"wangwu\",\"score\":300}"  ))  **val nameDF = sqlContext.read.json(nameRDD)**  **val scoreDF = sqlContext.read.json(scoreRDD)**  nameDF.registerTempTable("name")  scoreDF.registerTempTable("score")  **val** result = sqlContext.sql("select name.name,name.age,score.score from name,score where name.name = score.name")  result.show()  sc.stop() |

1. 非json格式的RDD创建DataFrame
2. 通过反射的方式将非json格式的RDD转换成DataFrame（不建议使用）

* 自定义类要可序列化
* 自定义类的访问级别是Public
* RDD转成DataFrame后会根据映射将字段按Assci码排序
* 将DataFrame转换成RDD时获取字段两种方式,一种是df.getInt(0)下标获取（不推荐使用），另一种是df.getAs(“列名”)获取（推荐使用）

|  |
| --- |
| /\*\*  \* 注意：  \* 1.自定义类必须是可序列化的  \* 2.自定义类访问级别必须是Public  \* 3.RDD转成DataFrame会把自定义类中字段的名称按assci码排序  \*/  SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("RDD");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> lineRDD = sc.textFile("sparksql/person.txt");  JavaRDD<Person> personRDD = lineRDD.map(**new** Function<String, Person>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Person call(String s) **throws** Exception {  Person p = **new** Person();  p.setId(s.split(",")[0]);  p.setName(s.split(",")[1]);  p.setAge(Integer.*valueOf*(s.split(",")[2]));  **return** p;  }  });  /\*\*  \* 传入进去Person.class的时候，sqlContext是通过反射的方式创建DataFrame  \* 在底层通过反射的方式获得Person的所有field，结合RDD本身，就生成了DataFrame  \*/  DataFrame df = sqlContext.createDataFrame(personRDD, Person.**class**);  df.show();  df.registerTempTable("person");  sqlContext.sql("select name from person where id = 2").show();  /\*\*  \* 将DataFrame转成JavaRDD  \* 注意：  \* 1.可以使用row.getInt(0),row.getString(1)...通过下标获取返回Row类型的数据，但是要注意列顺序问题---不常用  \* 2.可以使用row.getAs("列名")来获取对应的列值。  \*  \*/  JavaRDD<Row> javaRDD = df.javaRDD();  JavaRDD<Person> map = javaRDD.map(**new** Function<Row, Person>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Person call(Row row) **throws** Exception {  Person p = **new** Person();  //p.setId(row.getString(1));  //p.setName(row.getString(2));  //p.setAge(row.getInt(0));  p.setId((String)row.getAs("id"));  p.setName((String)row.getAs("name"));  p.setAge((Integer)row.getAs("age"));  **return** p;  }  });  map.foreach(**new** VoidFunction<Person>() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** **void** call(Person t) **throws** Exception {  System.***out***.println(t);  }  });  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("rddreflect")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** lineRDD = sc.textFile("./sparksql/person.txt")  /\*\*  \* 将RDD隐式转换成DataFrame  \*/  **import sqlContext.implicits.\_**  **val** personRDD = lineRDD.map { x => {  **val** person = **Person**(x.split(",")(0),x.split(",")(1),Integer.valueOf(x.split(",")(2)))  person  } }  **val df = personRDD.toDF();**  df.show()  /\*\*  \* 将DataFrame转换成PersonRDD  \*/  **val** rdd = df.rdd  **val** result = rdd.map { x => {  **Person**(x.getAs("id"),x.getAs("name"),x.getAs("age"))  } }  result.foreach { println}  sc.stop() |

1. 动态创建Schema将非json格式的RDD转换成DataFrame

java：

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("rddStruct");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> lineRDD = sc.textFile("./sparksql/person.txt");  /\*\*  \* 转换成Row类型的RDD  \*/  JavaRDD<Row> rowRDD = lineRDD.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(  String.*valueOf*(s.split(",")[0]),  String.*valueOf*(s.split(",")[1]),  Integer.*valueOf*(s.split(",")[2])  );  }  });  /\*\*  \* 动态构建DataFrame中的元数据，一般来说这里的字段可以来源自字符串，也可以来源于外部数据库  \*/  List<StructField> asList =Arrays.*asList*(  DataTypes.*createStructField*("id", DataTypes.***StringType***, **true**),  DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**),  DataTypes.*createStructField*("age", DataTypes.***IntegerType***, **true**)  );  StructType schema = DataTypes.*createStructType*(asList);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.show();  sc.stop(); |

scala：

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("rddStruct")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** lineRDD = sc.textFile("./sparksql/person.txt")  **val** rowRDD = lineRDD.map { x => {  **val** split = x.split(",")  RowFactory.create(split(0),split(1),Integer.valueOf(split(2)))  } }  **val** schema = **StructType**(List(  **StructField**("id",**StringType**,**true**),  **StructField**("name",**StringType**,**true**),  **StructField**("age",**IntegerType**,**true**)  ))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.printSchema()  sc.stop() |

1. 读取parquet文件创建DataFrame

注意：

* 可以将DataFrame存储成parquet文件。保存成parquet文件的方式有两种

|  |
| --- |
| **df.write().mode(SaveMode.*Overwrite*)format("parquet")**  **.save("./sparksql/parquet");**  **df.write().mode(SaveMode.*Overwrite*).parquet("./sparksql/parquet");** |

* SaveMode指定文件保存时的模式。

Overwrite：覆盖

Append：追加

ErrorIfExists：如果存在就报错

Ignore：如果存在就忽略

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("parquet");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> jsonRDD = sc.textFile("sparksql/json");  DataFrame df = sqlContext.read().json(jsonRDD);  /\*\*  \* 将DataFrame保存成parquet文件，SaveMode指定存储文件时的保存模式  \* 保存成parquet文件有以下两种方式：  \*/  df.write().mode(SaveMode.***Overwrite***).format("parquet").save("./sparksql/parquet");  df.write().mode(SaveMode.***Overwrite***).parquet("./sparksql/parquet");  df.show();  /\*\*  \* 加载parquet文件成DataFrame  \* 加载parquet文件有以下两种方式：  \*/  DataFrame load = sqlContext.read().format("parquet").load("./sparksql/parquet");  load = sqlContext.read().parquet("./sparksql/parquet");  load.show();  sc.stop(); |

scala：

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("parquet")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** jsonRDD = sc.textFile("sparksql/json")  **val** df = sqlContext.read.json(jsonRDD)  df.show()  /\*\*  \* 将DF保存为parquet文件  \*/  df.write.mode(SaveMode.Overwrite).format("parquet").save("./sparksql/parquet")  df.write.mode(SaveMode.Overwrite).parquet("./sparksql/parquet")  /\*\*  \* 读取parquet文件  \*/  **var** result = sqlContext.read.parquet("./sparksql/parquet")  result = sqlContext.read.format("parquet").load("./sparksql/parquet")  result.show()  sc.stop() |

1. 读取JDBC中的数据创建DataFrame(MySql为例)

两种方式创建DataFrame

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("mysql");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  /\*\*  \* 第一种方式读取MySql数据库表，加载为DataFrame  \*/  **Map<String, String> options = new HashMap<String,String>();**  **options.put("url", "jdbc:mysql://192.168.179.4:3306/spark");**  **options.put("driver", "com.mysql.jdbc.Driver");**  **options.put("user", "root");**  **options.put("password", "123456");**  **options.put("dbtable", "person");**  **DataFrame person = sqlContext.read().format("jdbc").options(options).load();**  person.show();  person.registerTempTable("person");  /\*\*  \* 第二种方式读取MySql数据表加载为DataFrame  \*/  **DataFrameReader reader = sqlContext.read().format("jdbc");**  **reader.option("url", "jdbc:mysql://192.168.179.4:3306/spark");**  **reader.option("driver", "com.mysql.jdbc.Driver");**  **reader.option("user", "root");**  **reader.option("password", "123456");**  **reader.option("dbtable", "score");**  **DataFrame score = reader.load();**  score.show();  score.registerTempTable("score");  DataFrame result =  sqlContext.sql("select person.id,person.name,score.score from person,score where person.name = score.name");  result.show();  /\*\*  \* 将DataFrame结果保存到Mysql中  \*/  Properties properties = **new** Properties();  properties.setProperty("user", "root");  properties.setProperty("password", "123456");  result.write().mode(SaveMode.***Overwrite***).jdbc("jdbc:mysql://192.168.179.4:3306/spark", "result", properties);  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("mysql")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  /\*\*  \* 第一种方式读取Mysql数据库表创建DF  \*/  **val options = new HashMap[*String*,*String*]();**  **options.put("url", "jdbc:mysql://192.168.179.4:3306/spark")**  **options.put("driver","com.mysql.jdbc.Driver")**  **options.put("user","root")**  **options.put("password", "123456")**  **options.put("dbtable","person")**  **val person = sqlContext.read.format("jdbc").options(options).load()**  person.show()  person.registerTempTable("person")  /\*\*  \* 第二种方式读取Mysql数据库表创建DF  \*/  **val reader = sqlContext.read.format("jdbc")**  **reader.option("url", "jdbc:mysql://192.168.179.4:3306/spark")**  **reader.option("driver","com.mysql.jdbc.Driver")**  **reader.option("user","root")**  **reader.option("password","123456")**  **reader.option("dbtable", "score")**  **val score = reader.load()**  score.show()  score.registerTempTable("score")  **val** result = sqlContext.sql("select person.id,person.name,score.score from person,score where person.name = score.name")  result.show()  /\*\*  \* 将数据写入到Mysql表中  \*/  **val** properties = **new** Properties()  properties.setProperty("user", "root")  properties.setProperty("password", "123456")  result.write.mode(SaveMode.Append).jdbc("jdbc:mysql://192.168.179.4:3306/spark", "result", properties)  sc.stop() |

1. 读取Hive中的数据加载成DataFrame

* HiveContext是SQLContext的子类，连接Hive建议使用HiveContext。
* 由于本地没有Hive环境，要提交到集群运行，提交命令：

|  |
| --- |
| ./spark-submit  --master spark://node1:7077,node2:7077  --executor-cores 1  --executor-memory 2G  --total-executor-cores 1  --class com.bjsxt.sparksql.dataframe.CreateDFFromHive  /root/test/HiveTest.jar |

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("hive");  JavaSparkContext sc = **new** JavaSparkContext(conf);  //HiveContext是SQLContext的子类。  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("USE spark");  hiveContext.sql("DROP TABLE IF EXISTS student\_infos");  //在hive中创建student\_infos表  hiveContext.sql("CREATE TABLE IF NOT EXISTS student\_infos (name STRING,age INT) row format delimited fields terminated by '\t' ");  hiveContext.sql("load data local inpath '/root/test/student\_infos' into table student\_infos");  hiveContext.sql("DROP TABLE IF EXISTS student\_scores");  hiveContext.sql("CREATE TABLE IF NOT EXISTS student\_scores (name STRING, score INT) row format delimited fields terminated by '\t'");  hiveContext.sql("LOAD DATA "  + "LOCAL INPATH '/root/test/student\_scores'"  + "INTO TABLE student\_scores");  /\*\*  \* 查询表生成DataFrame  \*/  DataFrame goodStudentsDF = hiveContext.sql("SELECT si.name, si.age, ss.score "  + "FROM student\_infos si "  + "JOIN student\_scores ss "  + "ON si.name=ss.name "  + "WHERE ss.score>=80");  hiveContext.sql("DROP TABLE IF EXISTS good\_student\_infos");  goodStudentsDF.registerTempTable("goodstudent");  DataFrame result = hiveContext.sql("select \* from goodstudent");  result.show();  /\*\*  \* 将结果保存到hive表 good\_student\_infos  \*/  **goodStudentsDF.write().mode(SaveMode.*Overwrite*).saveAsTable("good\_student\_infos");**  Row[] goodStudentRows = hiveContext.table("good\_student\_infos").collect();  **for**(Row goodStudentRow : goodStudentRows) {  System.***out***.println(goodStudentRow);  }  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("HiveSource")  **val** sc = **new** SparkContext(conf)  /\*\*  \* HiveContext是SQLContext的子类。  \*/  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark")  hiveContext.sql("drop table if exists student\_infos")  hiveContext.sql("create table if not exists student\_infos (name string,age int) row format delimited fields terminated by '\t'")  hiveContext.sql("load data local inpath '/root/test/student\_infos' into table student\_infos")    hiveContext.sql("drop table if exists student\_scores")  hiveContext.sql("create table if not exists student\_scores (name string,score int) row format delimited fields terminated by '\t'")  hiveContext.sql("load data local inpath '/root/test/student\_scores' into table student\_scores")    **val** df = hiveContext.sql("select si.name,si.age,ss.score from student\_infos si,student\_scores ss where si.name = ss.name")  hiveContext.sql("drop table if exists good\_student\_infos")  /\*\*  \* 将结果写入到hive表中  \*/  df.write.mode(SaveMode.Overwrite).saveAsTable("good\_student\_infos")    sc.stop() |

1. Spark On Hive的配置
2. 在Spark客户端配置Hive On Spark

在Spark客户端安装包下spark-1.6.0/conf中创建文件hive-site.xml：

配置hive的metastore路径

|  |
| --- |
| **<configuration>**  **<property>**  **<name>hive.metastore.uris</name>**  **<value>thrift://node1:9083</value>**  **</property>**  **</configuration>** |

1. 启动Hive的metastore服务

|  |
| --- |
| hive --service metastore |

1. 启动zookeeper集群，启动HDFS集群。
2. 启动SparkShell 读取Hive中的表总数，对比hive中查询同一表查询总数测试时间。

|  |
| --- |
| ./spark-shell  --master spark://node1:7077,node2:7077  --executor-cores 1  --executor-memory 1g  --total-executor-cores 1  import org.apache.spark.sql.hive.HiveContext  val hc = new HiveContext(sc)  hc.sql("show databases").show  hc.sql("user default").show  hc.sql("select count(\*) from jizhan").show |

* 注意：

如果使用Spark on Hive 查询数据时，出现错误：



找不到HDFS集群路径，要在客户端机器conf/spark-env.sh中设置HDFS的路径：

1. 序列化问题。
2. 储存DataFrame
3. 将DataFrame存储为parquet文件。
4. 将DataFrame存储到JDBC数据库。
5. 将DataFrame存储到Hive表。
6. 自定义函数UDF和UDAF
7. UDF:用户自定义函数。

可以自定义类实现UDFX接口。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local");  conf.setAppName("udf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***,**true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD,schema);  df.registerTempTable("user");  /\*\*  \* 根据UDF函数参数的个数来决定是实现哪一个UDF UDF1，UDF2。。。。UDF1xxx  \*/  sqlContext.udf().register("StrLen", **new** UDF1<String,Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Integer call(String t1) **throws** Exception {  **return** t1.length();  }  }, DataTypes.***IntegerType***);  sqlContext.sql("select name ,StrLen(name) as length from user").show();  //sqlContext.udf().register("StrLen",new UDF2<String, Integer, Integer>() {  //  // /\*\*  // \*  // \*/  // private static final long serialVersionUID = 1L;  //  // @Override  // public Integer call(String t1, Integer t2) throws Exception {  //return t1.length()+t2;  // }  //} ,DataTypes.IntegerType );  //sqlContext.sql("select name ,StrLen(name,10) as length from user").show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc);  **val** rdd = sc.makeRDD(Array("zhansan","lisi","wangwu"))  **val** rowRDD = rdd.map { x => {  RowFactory.create(x)  } }  **val** schema = DataTypes.createStructType(Array(**StructField**("name",**StringType**,**true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.registerTempTable("user")  //sqlContext.udf.register("StrLen",(s : String)=>{s.length()})  //sqlContext.sql("select name ,StrLen(name) as length from user").show  sqlContext.udf.register("StrLen",(s : *String*,i:Int)=>{s.length()+i})  sqlContext.sql("select name ,StrLen(name,10) as length from user").show  sc.stop() |

1. UDAF:用户自定义聚合函数。

* 实现UDAF函数如果要自定义类要继承UserDefinedAggregateFunction类

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("udaf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu","zhangsan","zhangsan","lisi"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.registerTempTable("user");  /\*\*  \* 注册一个UDAF函数,实现统计相同值得个数  \* 注意：这里可以自定义一个类继承UserDefinedAggregateFunction类也是可以的  \*/  sqlContext.udf().register("StringCount", **new** UserDefinedAggregateFunction() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  /\*\*  \* 更新 可以认为一个一个地将组内的字段值传递进来 实现拼接的逻辑  \* buffer.getInt(0)获取的是上一次聚合后的值  \* 相当于map端的combiner，combiner就是对每一个map task的处理结果进行一次小聚合  \* 大聚和发生在reduce端.  \* 这里即是:在进行聚合的时候，每当有新的值进来，对分组后的聚合如何进行计算  \*/  @Override  **public** **void** update(MutableAggregationBuffer buffer, Row arg1) {  buffer.update(0, buffer.getInt(0)+1);  }  /\*\*  \* 合并 update操作，可能是针对一个分组内的部分数据，在某个节点上发生的 但是可能一个分组内的数据，会分布在多个节点上处理  \* 此时就要用merge操作，将各个节点上分布式拼接好的串，合并起来  \* buffer1.getInt(0) : 大聚和的时候 上一次聚合后的值  \* buffer2.getInt(0) : 这次计算传入进来的update的结果  \* 这里即是：最后在分布式节点完成后需要进行全局级别的Merge操作  \*/  @Override  **public** **void** merge(MutableAggregationBuffer buffer1, Row buffer2) {  buffer1.update(0, buffer1.getInt(0) + buffer2.getInt(0));  }  /\*\*  \* 指定输入字段的字段及类型  \*/  @Override  **public** StructType inputSchema() {  **return** DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("name",  DataTypes.***StringType***, **true**)));  }  /\*\*  \* 初始化一个内部的自己定义的值,在Aggregate之前每组数据的初始化结果  \*/  @Override  **public** **void** initialize(MutableAggregationBuffer buffer) {  buffer.update(0, 0);  }  /\*\*  \* 最后返回一个和DataType的类型要一致的类型，返回UDAF最后的计算结果  \*/  @Override  **public** Object evaluate(Row row) {  **return** row.getInt(0);  }    @Override  **public** **boolean** deterministic() {  //设置为true  **return** **true**;  }  /\*\*  \* 指定UDAF函数计算后返回的结果类型  \*/  @Override  **public** DataType dataType() {  **return** DataTypes.***IntegerType***;  }  /\*\*  \* 在进行聚合操作的时候所要处理的数据的结果的类型  \*/  @Override  **public** StructType bufferSchema() {  **return**  DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("bf", DataTypes.***IntegerType***,  **true**)));  }    });  sqlContext.sql("select name ,StringCount(name) from user group by name").show();  sc.stop(); |

scala:

|  |
| --- |
| **class** MyUDAF **extends** UserDefinedAggregateFunction {  // 聚合操作时，所处理的数据的类型  **def** bufferSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("aaa", **IntegerType**, **true**)))  }  // 最终函数返回值的类型  **def** dataType: DataType = {  DataTypes.IntegerType  }  **def** deterministic: Boolean = {  **true**  }  // 最后返回一个最终的聚合值 要和dataType的类型一一对应  **def** evaluate(buffer: Row): Any = {  buffer.getAs[Int](0)  }  // 为每个分组的数据执行初始化值  **def** initialize(buffer: MutableAggregationBuffer): Unit = {  buffer(0) = 0  }  //输入数据的类型  **def** inputSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("input", **StringType**, **true**)))  }  // 最后merger的时候，在各个节点上的聚合值，要进行merge，也就是合并  **def** merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  buffer1(0) = buffer1.getAs[Int](0)+buffer2.getAs[Int](0)  }  // 每个组，有新的值进来的时候，进行分组对应的聚合值的计算  **def** update(buffer: MutableAggregationBuffer, input: Row): Unit = {  buffer(0) = buffer.getAs[Int](0)+1  }  }  **object** UDAF {  **def** main(args: Array[*String*]): Unit = {  **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udaf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** rdd = sc.makeRDD(Array("zhangsan","lisi","wangwu","zhangsan","lisi"))  **val** rowRDD = rdd.map { x => {RowFactory.create(x)} }    **val** schema = DataTypes.createStructType(Array(DataTypes.createStructField("name", **StringType**, **true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.registerTempTable("user")  /\*\*  \* 注册一个udaf函数  \*/  sqlContext.udf.register("StringCount", **new** MyUDAF())  sqlContext.sql("select name ,StringCount(name) from user group by name").show()  sc.stop()  }  } |

1. 开窗函数

注意：

row\_number() 开窗函数是按照某个字段分组，然后取另一字段的前几个的值，相当于 分组取topN

如果SQL语句里面使用到了开窗函数，那么这个SQL语句必须使用HiveContext来执行，HiveContext默认情况下在本地无法创建。在MySql8之后也增加了开窗函数。

开窗函数格式：

**row\_number() over (partitin by XXX order by XXX)**

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("windowfun");  JavaSparkContext sc = **new** JavaSparkContext(conf);  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  DataFrame result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("windowfun")  **val** sc = **new** SparkContext(conf)  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  **val** result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop() |

# Spark day06

1. Shark

Shark是基于Spark计算框架之上且兼容Hive语法的SQL执行引擎，由于底层的计算采用了Spark，性能比MapReduce的Hive普遍快2倍以上，当数据全部load在内存的话，将快10倍以上，因此Shark可以作为交互式查询应用服务来使用。除了基于Spark的特性外，Shark是完全兼容Hive的语法，表结构以及UDF函数等，已有的HiveSql可以直接进行迁移至Shark上Shark底层依赖于Hive的解析器，查询优化器，但正是由于SHark的整体设计架构对Hive的依赖性太强，难以支持其长远发展，比如不能和Spark的其他组件进行很好的集成，无法满足Spark的一栈式解决大数据处理的需求。

1. SparkSQL
2. SparkSQL介绍

Hive是Shark的前身，Shark是SparkSQL的前身,SparkSQL产生的根本原因是其完全脱离了Hive的限制。

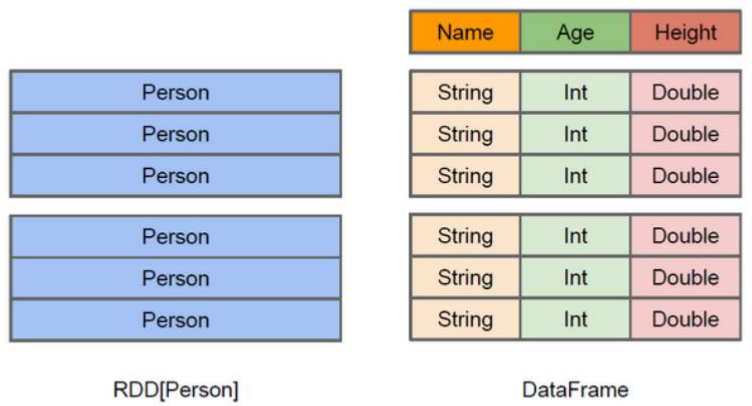
* SparkSQL支持查询原生的RDD。 RDD是Spark平台的核心概念，是Spark能够高效的处理大数据的各种场景的基础。
* 能够在Scala中写SQL语句。支持简单的SQL语法检查，能够在Scala中写Hive语句访问Hive数据，并将结果取回作为RDD使用。

1. Spark on Hive和Hive on Spark

Spark on Hive： Hive只作为储存角色，Spark负责sql解析优化，执行。

Hive on Spark：Hive即作为存储又负责sql的解析优化，Spark负责执行。

1. DataFrame

DataFrame也是一个分布式数据容器。与RDD类似，然而DataFrame更像传统数据库的二维表格，除了数据以外，还掌握数据的结构信息，即schema。同时，与Hive类似，DataFrame也支持嵌套数据类型（struct、array和map）。从API易用性的角度上 看， DataFrame API提供的是一套高层的关系操作，比函数式的RDD API要更加友好，门槛更低。

DataFrame就Row类型的DataSet。

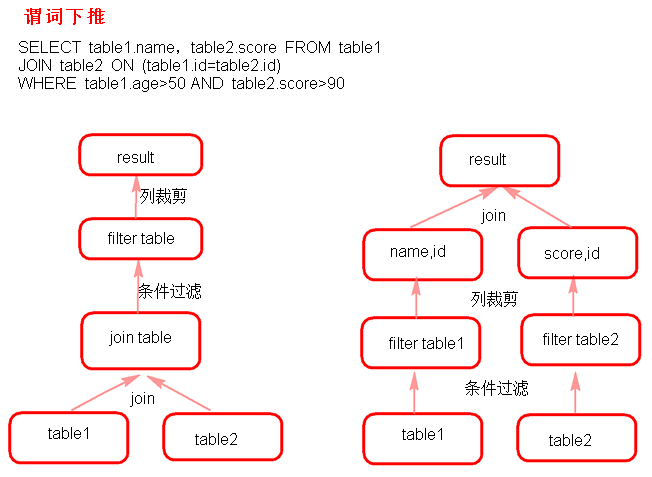
1. SparkSQL的数据源

SparkSQL的数据源可以是JSON类型的字符串，JDBC,Parquent,Hive，HDFS等。

1. SparkSQL底层架构

首先拿到sql后解析一批未被解决的逻辑计划，再经过分析得到分析后的逻辑计划，再经过一批优化规则转换成一批最佳优化的逻辑计划，再经过SparkPlanner的策略转化成一批物理计划，随后经过消费模型转换成一个个的Spark任务执行。

1. 谓词下推（predicate Pushdown）



1. 创建DataFrame的几种方式
2. 读取json格式的文件创建DataFrame

注意：

* json文件中的json数据不能嵌套json格式数据。
* DataFrame是一个一个Row类型的RDD，df.rdd()/df.javaRdd()。
* 可以两种方式读取json格式的文件。
* df.show()默认显示前20行数据。
* DataFrame原生API可以操作DataFrame。
* 注册成临时表时，表中的列默认按ascii顺序显示列。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("jsonfile");  SparkContext sc = **new** SparkContext(conf);    //创建sqlContext  SQLContext sqlContext = **new** SQLContext(sc);    /\*\*  \* DataFrame的底层是一个一个的RDD RDD的泛型是Row类型。  \* 以下两种方式都可以读取json格式的文件  \*/  **DataFrame df = sqlContext.read().format("json").load("sparksql/json");**  **// DataFrame df2 = sqlContext.read().json("sparksql/json.txt");**  // df2.show();  /\*\*  \* DataFrame转换成RDD  \*/  **RDD<Row> rdd = df.rdd();**  /\*\*  \* 显示 DataFrame中的内容，默认显示前20行。如果现实多行要指定多少行show(行数)  \* 注意：当有多个列时，显示的列先后顺序是按列的ascii码先后显示。  \*/  // df.show();  /\*\*  \* 树形的形式显示schema信息  \*/  **df.printSchema();**    /\*\*  \* dataFram自带的API 操作DataFrame  \*/  //select name from table  // df.select("name").show();  //select name age+10 as addage from table  df.select(df.col("name"),df.col("age").plus(10).alias("addage")).show();  //select name ,age from table where age>19  df.select(df.col("name"),df.col("age")).where(df.col("age").gt(19)).show();  //select count(\*) from table group by age  df.groupBy(df.col("age")).count().show();    /\*\*  \* 将DataFrame注册成临时的一张表，这张表临时注册到内存中，是逻辑上的表，不会雾化到磁盘  \*/  **df.registerTempTable("jtable");**    DataFrame sql = sqlContext.sql("select age,count(1) from jtable group by age");  DataFrame sql2 = sqlContext.sql("select \* from jtable");    sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("jsonfile")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val df = sqlContext.read.json("sparksql/json")**  **//val df1 = sqlContext.read.format("json").load("sparksql/json")**  **df.show()**  **df.printSchema()**  //select \* from table  df.select(df.col("name")).show()  //select name from table where age>19  df.select(df.col("name"),df.col("age")).where(df.col("age").gt(19)).show()  //select count(\*) from table group by age  df.groupBy(df.col("age")).count().show();    /\*\*  \* 注册临时表  \*/  **df.registerTempTable("jtable")**  **val** result = sqlContext.sql("select \* from jtable")  result.show()  sc.stop() |

1. 通过json格式的RDD创建DataFrame

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("jsonRDD");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> nameRDD = sc.parallelize(Arrays.*asList*(  "{\"name\":\"zhangsan\",\"age\":\"18\"}",  "{\"name\":\"lisi\",\"age\":\"19\"}",  "{\"name\":\"wangwu\",\"age\":\"20\"}"  ));  JavaRDD<String> scoreRDD = sc.parallelize(Arrays.*asList*(  "{\"name\":\"zhangsan\",\"score\":\"100\"}",  "{\"name\":\"lisi\",\"score\":\"200\"}",  "{\"name\":\"wangwu\",\"score\":\"300\"}"  ));  **DataFrame namedf = sqlContext.read().json(nameRDD);**  **DataFrame scoredf = sqlContext.read().json(scoreRDD);**  namedf.registerTempTable("name");  scoredf.registerTempTable("score");  DataFrame result = sqlContext.sql("select name.name,name.age,score.score from name,score where name.name = score.name");  result.show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("jsonrdd")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** nameRDD = sc.makeRDD(Array(  "{\"name\":\"zhangsan\",\"age\":18}",  "{\"name\":\"lisi\",\"age\":19}",  "{\"name\":\"wangwu\",\"age\":20}"  ))  **val** scoreRDD = sc.makeRDD(Array(  "{\"name\":\"zhangsan\",\"score\":100}",  "{\"name\":\"lisi\",\"score\":200}",  "{\"name\":\"wangwu\",\"score\":300}"  ))  **val nameDF = sqlContext.read.json(nameRDD)**  **val scoreDF = sqlContext.read.json(scoreRDD)**  nameDF.registerTempTable("name")  scoreDF.registerTempTable("score")  **val** result = sqlContext.sql("select name.name,name.age,score.score from name,score where name.name = score.name")  result.show()  sc.stop() |

1. 非json格式的RDD创建DataFrame
2. 通过反射的方式将非json格式的RDD转换成DataFrame（不建议使用）

* 自定义类要可序列化
* 自定义类的访问级别是Public
* RDD转成DataFrame后会根据映射将字段按Assci码排序
* 将DataFrame转换成RDD时获取字段两种方式,一种是df.getInt(0)下标获取（不推荐使用），另一种是df.getAs(“列名”)获取（推荐使用）

|  |
| --- |
| /\*\*  \* 注意：  \* 1.自定义类必须是可序列化的  \* 2.自定义类访问级别必须是Public  \* 3.RDD转成DataFrame会把自定义类中字段的名称按assci码排序  \*/  SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("RDD");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> lineRDD = sc.textFile("sparksql/person.txt");  JavaRDD<Person> personRDD = lineRDD.map(**new** Function<String, Person>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Person call(String s) **throws** Exception {  Person p = **new** Person();  p.setId(s.split(",")[0]);  p.setName(s.split(",")[1]);  p.setAge(Integer.*valueOf*(s.split(",")[2]));  **return** p;  }  });  /\*\*  \* 传入进去Person.class的时候，sqlContext是通过反射的方式创建DataFrame  \* 在底层通过反射的方式获得Person的所有field，结合RDD本身，就生成了DataFrame  \*/  DataFrame df = sqlContext.createDataFrame(personRDD, Person.**class**);  df.show();  df.registerTempTable("person");  sqlContext.sql("select name from person where id = 2").show();  /\*\*  \* 将DataFrame转成JavaRDD  \* 注意：  \* 1.可以使用row.getInt(0),row.getString(1)...通过下标获取返回Row类型的数据，但是要注意列顺序问题---不常用  \* 2.可以使用row.getAs("列名")来获取对应的列值。  \*  \*/  JavaRDD<Row> javaRDD = df.javaRDD();  JavaRDD<Person> map = javaRDD.map(**new** Function<Row, Person>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Person call(Row row) **throws** Exception {  Person p = **new** Person();  //p.setId(row.getString(1));  //p.setName(row.getString(2));  //p.setAge(row.getInt(0));  p.setId((String)row.getAs("id"));  p.setName((String)row.getAs("name"));  p.setAge((Integer)row.getAs("age"));  **return** p;  }  });  map.foreach(**new** VoidFunction<Person>() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** **void** call(Person t) **throws** Exception {  System.***out***.println(t);  }  });  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("rddreflect")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** lineRDD = sc.textFile("./sparksql/person.txt")  /\*\*  \* 将RDD隐式转换成DataFrame  \*/  **import sqlContext.implicits.\_**  **val** personRDD = lineRDD.map { x => {  **val** person = **Person**(x.split(",")(0),x.split(",")(1),Integer.valueOf(x.split(",")(2)))  person  } }  **val df = personRDD.toDF();**  df.show()  /\*\*  \* 将DataFrame转换成PersonRDD  \*/  **val** rdd = df.rdd  **val** result = rdd.map { x => {  **Person**(x.getAs("id"),x.getAs("name"),x.getAs("age"))  } }  result.foreach { println}  sc.stop() |

1. 动态创建Schema将非json格式的RDD转换成DataFrame

java：

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("rddStruct");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> lineRDD = sc.textFile("./sparksql/person.txt");  /\*\*  \* 转换成Row类型的RDD  \*/  JavaRDD<Row> rowRDD = lineRDD.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(  String.*valueOf*(s.split(",")[0]),  String.*valueOf*(s.split(",")[1]),  Integer.*valueOf*(s.split(",")[2])  );  }  });  /\*\*  \* 动态构建DataFrame中的元数据，一般来说这里的字段可以来源自字符串，也可以来源于外部数据库  \*/  List<StructField> asList =Arrays.*asList*(  DataTypes.*createStructField*("id", DataTypes.***StringType***, **true**),  DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**),  DataTypes.*createStructField*("age", DataTypes.***IntegerType***, **true**)  );  StructType schema = DataTypes.*createStructType*(asList);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.show();  sc.stop(); |

scala：

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("rddStruct")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** lineRDD = sc.textFile("./sparksql/person.txt")  **val** rowRDD = lineRDD.map { x => {  **val** split = x.split(",")  RowFactory.create(split(0),split(1),Integer.valueOf(split(2)))  } }  **val** schema = **StructType**(List(  **StructField**("id",**StringType**,**true**),  **StructField**("name",**StringType**,**true**),  **StructField**("age",**IntegerType**,**true**)  ))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.printSchema()  sc.stop() |

1. 读取parquet文件创建DataFrame

注意：

* 可以将DataFrame存储成parquet文件。保存成parquet文件的方式有两种

|  |
| --- |
| **df.write().mode(SaveMode.*Overwrite*)format("parquet")**  **.save("./sparksql/parquet");**  **df.write().mode(SaveMode.*Overwrite*).parquet("./sparksql/parquet");** |

* SaveMode指定文件保存时的模式。

Overwrite：覆盖

Append：追加

ErrorIfExists：如果存在就报错

Ignore：如果存在就忽略

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("parquet");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> jsonRDD = sc.textFile("sparksql/json");  DataFrame df = sqlContext.read().json(jsonRDD);  /\*\*  \* 将DataFrame保存成parquet文件，SaveMode指定存储文件时的保存模式  \* 保存成parquet文件有以下两种方式：  \*/  df.write().mode(SaveMode.***Overwrite***).format("parquet").save("./sparksql/parquet");  df.write().mode(SaveMode.***Overwrite***).parquet("./sparksql/parquet");  df.show();  /\*\*  \* 加载parquet文件成DataFrame  \* 加载parquet文件有以下两种方式：  \*/  DataFrame load = sqlContext.read().format("parquet").load("./sparksql/parquet");  load = sqlContext.read().parquet("./sparksql/parquet");  load.show();  sc.stop(); |

scala：

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("parquet")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** jsonRDD = sc.textFile("sparksql/json")  **val** df = sqlContext.read.json(jsonRDD)  df.show()  /\*\*  \* 将DF保存为parquet文件  \*/  df.write.mode(SaveMode.Overwrite).format("parquet").save("./sparksql/parquet")  df.write.mode(SaveMode.Overwrite).parquet("./sparksql/parquet")  /\*\*  \* 读取parquet文件  \*/  **var** result = sqlContext.read.parquet("./sparksql/parquet")  result = sqlContext.read.format("parquet").load("./sparksql/parquet")  result.show()  sc.stop() |

1. 读取JDBC中的数据创建DataFrame(MySql为例)

两种方式创建DataFrame

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("mysql");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  /\*\*  \* 第一种方式读取MySql数据库表，加载为DataFrame  \*/  **Map<String, String> options = new HashMap<String,String>();**  **options.put("url", "jdbc:mysql://192.168.179.4:3306/spark");**  **options.put("driver", "com.mysql.jdbc.Driver");**  **options.put("user", "root");**  **options.put("password", "123456");**  **options.put("dbtable", "person");**  **DataFrame person = sqlContext.read().format("jdbc").options(options).load();**  person.show();  person.registerTempTable("person");  /\*\*  \* 第二种方式读取MySql数据表加载为DataFrame  \*/  **DataFrameReader reader = sqlContext.read().format("jdbc");**  **reader.option("url", "jdbc:mysql://192.168.179.4:3306/spark");**  **reader.option("driver", "com.mysql.jdbc.Driver");**  **reader.option("user", "root");**  **reader.option("password", "123456");**  **reader.option("dbtable", "score");**  **DataFrame score = reader.load();**  score.show();  score.registerTempTable("score");  DataFrame result =  sqlContext.sql("select person.id,person.name,score.score from person,score where person.name = score.name");  result.show();  /\*\*  \* 将DataFrame结果保存到Mysql中  \*/  Properties properties = **new** Properties();  properties.setProperty("user", "root");  properties.setProperty("password", "123456");  result.write().mode(SaveMode.***Overwrite***).jdbc("jdbc:mysql://192.168.179.4:3306/spark", "result", properties);  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("mysql")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  /\*\*  \* 第一种方式读取Mysql数据库表创建DF  \*/  **val options = new HashMap[*String*,*String*]();**  **options.put("url", "jdbc:mysql://192.168.179.4:3306/spark")**  **options.put("driver","com.mysql.jdbc.Driver")**  **options.put("user","root")**  **options.put("password", "123456")**  **options.put("dbtable","person")**  **val person = sqlContext.read.format("jdbc").options(options).load()**  person.show()  person.registerTempTable("person")  /\*\*  \* 第二种方式读取Mysql数据库表创建DF  \*/  **val reader = sqlContext.read.format("jdbc")**  **reader.option("url", "jdbc:mysql://192.168.179.4:3306/spark")**  **reader.option("driver","com.mysql.jdbc.Driver")**  **reader.option("user","root")**  **reader.option("password","123456")**  **reader.option("dbtable", "score")**  **val score = reader.load()**  score.show()  score.registerTempTable("score")  **val** result = sqlContext.sql("select person.id,person.name,score.score from person,score where person.name = score.name")  result.show()  /\*\*  \* 将数据写入到Mysql表中  \*/  **val** properties = **new** Properties()  properties.setProperty("user", "root")  properties.setProperty("password", "123456")  result.write.mode(SaveMode.Append).jdbc("jdbc:mysql://192.168.179.4:3306/spark", "result", properties)  sc.stop() |

1. 读取Hive中的数据加载成DataFrame

* HiveContext是SQLContext的子类，连接Hive建议使用HiveContext。
* 由于本地没有Hive环境，要提交到集群运行，提交命令：

|  |
| --- |
| ./spark-submit  --master spark://node1:7077,node2:7077  --executor-cores 1  --executor-memory 2G  --total-executor-cores 1  --class com.lw.sparksql.dataframe.CreateDFFromHive  /root/test/HiveTest.jar |

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("hive");  JavaSparkContext sc = **new** JavaSparkContext(conf);  //HiveContext是SQLContext的子类。  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("USE spark");  hiveContext.sql("DROP TABLE IF EXISTS student\_infos");  //在hive中创建student\_infos表  hiveContext.sql("CREATE TABLE IF NOT EXISTS student\_infos (name STRING,age INT) row format delimited fields terminated by '\t' ");  hiveContext.sql("load data local inpath '/root/test/student\_infos' into table student\_infos");  hiveContext.sql("DROP TABLE IF EXISTS student\_scores");  hiveContext.sql("CREATE TABLE IF NOT EXISTS student\_scores (name STRING, score INT) row format delimited fields terminated by '\t'");  hiveContext.sql("LOAD DATA "  + "LOCAL INPATH '/root/test/student\_scores'"  + "INTO TABLE student\_scores");  /\*\*  \* 查询表生成DataFrame  \*/  DataFrame goodStudentsDF = hiveContext.sql("SELECT si.name, si.age, ss.score "  + "FROM student\_infos si "  + "JOIN student\_scores ss "  + "ON si.name=ss.name "  + "WHERE ss.score>=80");  hiveContext.sql("DROP TABLE IF EXISTS good\_student\_infos");  goodStudentsDF.registerTempTable("goodstudent");  DataFrame result = hiveContext.sql("select \* from goodstudent");  result.show();  /\*\*  \* 将结果保存到hive表 good\_student\_infos  \*/  **goodStudentsDF.write().mode(SaveMode.*Overwrite*).saveAsTable("good\_student\_infos");**  Row[] goodStudentRows = hiveContext.table("good\_student\_infos").collect();  **for**(Row goodStudentRow : goodStudentRows) {  System.***out***.println(goodStudentRow);  }  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("HiveSource")  **val** sc = **new** SparkContext(conf)  /\*\*  \* HiveContext是SQLContext的子类。  \*/  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark")  hiveContext.sql("drop table if exists student\_infos")  hiveContext.sql("create table if not exists student\_infos (name string,age int) row format delimited fields terminated by '\t'")  hiveContext.sql("load data local inpath '/root/test/student\_infos' into table student\_infos")    hiveContext.sql("drop table if exists student\_scores")  hiveContext.sql("create table if not exists student\_scores (name string,score int) row format delimited fields terminated by '\t'")  hiveContext.sql("load data local inpath '/root/test/student\_scores' into table student\_scores")    **val** df = hiveContext.sql("select si.name,si.age,ss.score from student\_infos si,student\_scores ss where si.name = ss.name")  hiveContext.sql("drop table if exists good\_student\_infos")  /\*\*  \* 将结果写入到hive表中  \*/  df.write.mode(SaveMode.Overwrite).saveAsTable("good\_student\_infos")    sc.stop() |

1. Spark On Hive的配置
2. 在Spark客户端配置Hive On Spark

在Spark客户端安装包下spark-1.6.0/conf中创建文件hive-site.xml：

配置hive的metastore路径

|  |
| --- |
| **<configuration>**  **<property>**  **<name>hive.metastore.uris</name>**  **<value>thrift://node1:9083</value>**  **</property>**  **</configuration>** |

1. 启动Hive的metastore服务

|  |
| --- |
| hive --service metastore |

1. 启动zookeeper集群，启动HDFS集群。
2. 启动SparkShell 读取Hive中的表总数，对比hive中查询同一表查询总数测试时间。

|  |
| --- |
| ./spark-shell  --master spark://node1:7077,node2:7077  --executor-cores 1  --executor-memory 1g  --total-executor-cores 1  import org.apache.spark.sql.hive.HiveContext  val hc = new HiveContext(sc)  hc.sql("show databases").show  hc.sql("user default").show  hc.sql("select count(\*) from jizhan").show |

* 注意：

如果使用Spark on Hive 查询数据时，出现错误：



找不到HDFS集群路径，要在客户端机器conf/spark-env.sh中设置HDFS的路径：

1. 序列化问题。
2. 储存DataFrame
3. 将DataFrame存储为parquet文件。
4. 将DataFrame存储到JDBC数据库。
5. 将DataFrame存储到Hive表。
6. 自定义函数UDF和UDAF
7. UDF:用户自定义函数。

可以自定义类实现UDFX接口。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local");  conf.setAppName("udf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***,**true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD,schema);  df.registerTempTable("user");  /\*\*  \* 根据UDF函数参数的个数来决定是实现哪一个UDF UDF1，UDF2。。。。UDF1xxx  \*/  sqlContext.udf().register("StrLen", **new** UDF1<String,Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Integer call(String t1) **throws** Exception {  **return** t1.length();  }  }, DataTypes.***IntegerType***);  sqlContext.sql("select name ,StrLen(name) as length from user").show();  //sqlContext.udf().register("StrLen",new UDF2<String, Integer, Integer>() {  //  // /\*\*  // \*  // \*/  // private static final long serialVersionUID = 1L;  //  // @Override  // public Integer call(String t1, Integer t2) throws Exception {  //return t1.length()+t2;  // }  //} ,DataTypes.IntegerType );  //sqlContext.sql("select name ,StrLen(name,10) as length from user").show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc);  **val** rdd = sc.makeRDD(Array("zhansan","lisi","wangwu"))  **val** rowRDD = rdd.map { x => {  RowFactory.create(x)  } }  **val** schema = DataTypes.createStructType(Array(**StructField**("name",**StringType**,**true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.registerTempTable("user")  //sqlContext.udf.register("StrLen",(s : String)=>{s.length()})  //sqlContext.sql("select name ,StrLen(name) as length from user").show  sqlContext.udf.register("StrLen",(s : *String*,i:Int)=>{s.length()+i})  sqlContext.sql("select name ,StrLen(name,10) as length from user").show  sc.stop() |

1. UDAF:用户自定义聚合函数。

* 实现UDAF函数如果要自定义类要继承UserDefinedAggregateFunction类

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("udaf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu","zhangsan","zhangsan","lisi"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.registerTempTable("user");  /\*\*  \* 注册一个UDAF函数,实现统计相同值得个数  \* 注意：这里可以自定义一个类继承UserDefinedAggregateFunction类也是可以的  \*/  sqlContext.udf().register("StringCount", **new** UserDefinedAggregateFunction() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  /\*\*  \* 更新 可以认为一个一个地将组内的字段值传递进来 实现拼接的逻辑  \* buffer.getInt(0)获取的是上一次聚合后的值  \* 相当于map端的combiner，combiner就是对每一个map task的处理结果进行一次小聚合  \* 大聚和发生在reduce端.  \* 这里即是:在进行聚合的时候，每当有新的值进来，对分组后的聚合如何进行计算  \*/  @Override  **public** **void** update(MutableAggregationBuffer buffer, Row arg1) {  buffer.update(0, buffer.getInt(0)+1);  }  /\*\*  \* 合并 update操作，可能是针对一个分组内的部分数据，在某个节点上发生的 但是可能一个分组内的数据，会分布在多个节点上处理  \* 此时就要用merge操作，将各个节点上分布式拼接好的串，合并起来  \* buffer1.getInt(0) : 大聚和的时候 上一次聚合后的值  \* buffer2.getInt(0) : 这次计算传入进来的update的结果  \* 这里即是：最后在分布式节点完成后需要进行全局级别的Merge操作  \*/  @Override  **public** **void** merge(MutableAggregationBuffer buffer1, Row buffer2) {  buffer1.update(0, buffer1.getInt(0) + buffer2.getInt(0));  }  /\*\*  \* 指定输入字段的字段及类型  \*/  @Override  **public** StructType inputSchema() {  **return** DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("name",  DataTypes.***StringType***, **true**)));  }  /\*\*  \* 初始化一个内部的自己定义的值,在Aggregate之前每组数据的初始化结果  \*/  @Override  **public** **void** initialize(MutableAggregationBuffer buffer) {  buffer.update(0, 0);  }  /\*\*  \* 最后返回一个和DataType的类型要一致的类型，返回UDAF最后的计算结果  \*/  @Override  **public** Object evaluate(Row row) {  **return** row.getInt(0);  }    @Override  **public** **boolean** deterministic() {  //设置为true  **return** **true**;  }  /\*\*  \* 指定UDAF函数计算后返回的结果类型  \*/  @Override  **public** DataType dataType() {  **return** DataTypes.***IntegerType***;  }  /\*\*  \* 在进行聚合操作的时候所要处理的数据的结果的类型  \*/  @Override  **public** StructType bufferSchema() {  **return**  DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("bf", DataTypes.***IntegerType***,  **true**)));  }    });  sqlContext.sql("select name ,StringCount(name) from user group by name").show();  sc.stop(); |

scala:

|  |
| --- |
| **class** MyUDAF **extends** UserDefinedAggregateFunction {  // 聚合操作时，所处理的数据的类型  **def** bufferSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("aaa", **IntegerType**, **true**)))  }  // 最终函数返回值的类型  **def** dataType: DataType = {  DataTypes.IntegerType  }  **def** deterministic: Boolean = {  **true**  }  // 最后返回一个最终的聚合值 要和dataType的类型一一对应  **def** evaluate(buffer: Row): Any = {  buffer.getAs[Int](0)  }  // 为每个分组的数据执行初始化值  **def** initialize(buffer: MutableAggregationBuffer): Unit = {  buffer(0) = 0  }  //输入数据的类型  **def** inputSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("input", **StringType**, **true**)))  }  // 最后merger的时候，在各个节点上的聚合值，要进行merge，也就是合并  **def** merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  buffer1(0) = buffer1.getAs[Int](0)+buffer2.getAs[Int](0)  }  // 每个组，有新的值进来的时候，进行分组对应的聚合值的计算  **def** update(buffer: MutableAggregationBuffer, input: Row): Unit = {  buffer(0) = buffer.getAs[Int](0)+1  }  }  **object** UDAF {  **def** main(args: Array[*String*]): Unit = {  **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udaf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** rdd = sc.makeRDD(Array("zhangsan","lisi","wangwu","zhangsan","lisi"))  **val** rowRDD = rdd.map { x => {RowFactory.create(x)} }    **val** schema = DataTypes.createStructType(Array(DataTypes.createStructField("name", **StringType**, **true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.registerTempTable("user")  /\*\*  \* 注册一个udaf函数  \*/  sqlContext.udf.register("StringCount", **new** MyUDAF())  sqlContext.sql("select name ,StringCount(name) from user group by name").show()  sc.stop()  }  } |

1. 开窗函数

注意：

row\_number() 开窗函数是按照某个字段分组，然后取另一字段的前几个的值，相当于 分组取topN

如果SQL语句里面使用到了开窗函数，那么这个SQL语句必须使用HiveContext来执行，HiveContext默认情况下在本地无法创建。在MySql8之后也增加了开窗函数。

开窗函数格式：

**row\_number() over (partitin by XXX order by XXX)**

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("windowfun");  JavaSparkContext sc = **new** JavaSparkContext(conf);  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  DataFrame result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("windowfun")  **val** sc = **new** SparkContext(conf)  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  **val** result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop() |

# SparkDay7 - Streaming

1. 自定义函数UDF和UDAF
2. UDF:用户自定义函数。

可以自定义类实现UDFX接口。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local");  conf.setAppName("udf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***,**true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD,schema);  df.registerTempTable("user");  /\*\*  \* 根据UDF函数参数的个数来决定是实现哪一个UDF UDF1，UDF2。。。。UDF1xxx  \*/  sqlContext.udf().register("StrLen", **new** UDF1<String,Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Integer call(String t1) **throws** Exception {  **return** t1.length();  }  }, DataTypes.***IntegerType***);  sqlContext.sql("select name ,StrLen(name) as length from user").show();  //sqlContext.udf().register("StrLen",new UDF2<String, Integer, Integer>() {  //  // /\*\*  // \*  // \*/  // private static final long serialVersionUID = 1L;  //  // @Override  // public Integer call(String t1, Integer t2) throws Exception {  //return t1.length()+t2;  // }  //} ,DataTypes.IntegerType );  //sqlContext.sql("select name ,StrLen(name,10) as length from user").show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc);  **val** rdd = sc.makeRDD(Array("zhansan","lisi","wangwu"))  **val** rowRDD = rdd.map { x => {  RowFactory.create(x)  } }  **val** schema = DataTypes.createStructType(Array(**StructField**("name",**StringType**,**true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.registerTempTable("user")  //sqlContext.udf.register("StrLen",(s : String)=>{s.length()})  //sqlContext.sql("select name ,StrLen(name) as length from user").show  sqlContext.udf.register("StrLen",(s : *String*,i:Int)=>{s.length()+i})  sqlContext.sql("select name ,StrLen(name,10) as length from user").show  sc.stop() |

1. UDAF:用户自定义聚合函数。

* 实现UDAF函数如果要自定义类要继承UserDefinedAggregateFunction类

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("udaf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu","zhangsan","zhangsan","lisi"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.registerTempTable("user");  /\*\*  \* 注册一个UDAF函数,实现统计相同值得个数  \* 注意：这里可以自定义一个类继承UserDefinedAggregateFunction类也是可以的  \*/  sqlContext.udf().register("StringCount", **new** UserDefinedAggregateFunction() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  /\*\*  \* 更新 可以认为一个一个地将组内的字段值传递进来 实现拼接的逻辑  \* buffer.getInt(0)获取的是上一次聚合后的值  \* 相当于map端的combiner，combiner就是对每一个map task的处理结果进行一次小聚合  \* 大聚和发生在reduce端.  \* 这里即是:在进行聚合的时候，每当有新的值进来，对分组后的聚合如何进行计算  \*/  @Override  **public** **void** update(MutableAggregationBuffer buffer, Row arg1) {  buffer.update(0, buffer.getInt(0)+1);  }  /\*\*  \* 合并 update操作，可能是针对一个分组内的部分数据，在某个节点上发生的 但是可能一个分组内的数据，会分布在多个节点上处理  \* 此时就要用merge操作，将各个节点上分布式拼接好的串，合并起来  \* buffer1.getInt(0) : 大聚和的时候 上一次聚合后的值  \* buffer2.getInt(0) : 这次计算传入进来的update的结果  \* 这里即是：最后在分布式节点完成后需要进行全局级别的Merge操作  \*/  @Override  **public** **void** merge(MutableAggregationBuffer buffer1, Row buffer2) {  buffer1.update(0, buffer1.getInt(0) + buffer2.getInt(0));  }  /\*\*  \* 指定输入字段的字段及类型  \*/  @Override  **public** StructType inputSchema() {  **return** DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("name",  DataTypes.***StringType***, **true**)));  }  /\*\*  \* 初始化一个内部的自己定义的值,在Aggregate之前每组数据的初始化结果  \*/  @Override  **public** **void** initialize(MutableAggregationBuffer buffer) {  buffer.update(0, 0);  }  /\*\*  \* 最后返回一个和DataType的类型要一致的类型，返回UDAF最后的计算结果  \*/  @Override  **public** Object evaluate(Row row) {  **return** row.getInt(0);  }    @Override  **public** **boolean** deterministic() {  //设置为true  **return** **true**;  }  /\*\*  \* 指定UDAF函数计算后返回的结果类型  \*/  @Override  **public** DataType dataType() {  **return** DataTypes.***IntegerType***;  }  /\*\*  \* 在进行聚合操作的时候所要处理的数据的结果的类型  \*/  @Override  **public** StructType bufferSchema() {  **return**  DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("bf", DataTypes.***IntegerType***,  **true**)));  }    });  sqlContext.sql("select name ,StringCount(name) from user group by name").show();  sc.stop(); |

scala:

|  |
| --- |
| **class** MyUDAF **extends** UserDefinedAggregateFunction {  // 聚合操作时，所处理的数据的类型  **def** bufferSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("aaa", **IntegerType**, **true**)))  }  // 最终函数返回值的类型  **def** dataType: DataType = {  DataTypes.IntegerType  }  **def** deterministic: Boolean = {  **true**  }  // 最后返回一个最终的聚合值 要和dataType的类型一一对应  **def** evaluate(buffer: Row): Any = {  buffer.getAs[Int](0)  }  // 为每个分组的数据执行初始化值  **def** initialize(buffer: MutableAggregationBuffer): Unit = {  buffer(0) = 0  }  //输入数据的类型  **def** inputSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("input", **StringType**, **true**)))  }  // 最后merger的时候，在各个节点上的聚合值，要进行merge，也就是合并  **def** merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  buffer1(0) = buffer1.getAs[Int](0)+buffer2.getAs[Int](0)  }  // 每个组，有新的值进来的时候，进行分组对应的聚合值的计算  **def** update(buffer: MutableAggregationBuffer, input: Row): Unit = {  buffer(0) = buffer.getAs[Int](0)+1  }  }  **object** UDAF {  **def** main(args: Array[*String*]): Unit = {  **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udaf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** rdd = sc.makeRDD(Array("zhangsan","lisi","wangwu","zhangsan","lisi"))  **val** rowRDD = rdd.map { x => {RowFactory.create(x)} }    **val** schema = DataTypes.createStructType(Array(DataTypes.createStructField("name", **StringType**, **true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.registerTempTable("user")  /\*\*  \* 注册一个udaf函数  \*/  sqlContext.udf.register("StringCount", **new** MyUDAF())  sqlContext.sql("select name ,StringCount(name) from user group by name").show()  sc.stop()  }  } |

1. 开窗函数

注意：

row\_number() 开窗函数是按照某个字段分组，然后取另一字段的前几个的值，相当于 分组取topN

如果SQL语句里面使用到了开窗函数，那么这个SQL语句必须使用HiveContext来执行，HiveContext默认情况下在本地无法创建。在MySql8之后也增加了开窗函数。

开窗函数格式：

**row\_number() over (partitin by XXX order by XXX)**

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("windowfun");  JavaSparkContext sc = **new** JavaSparkContext(conf);  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  DataFrame result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop(); |

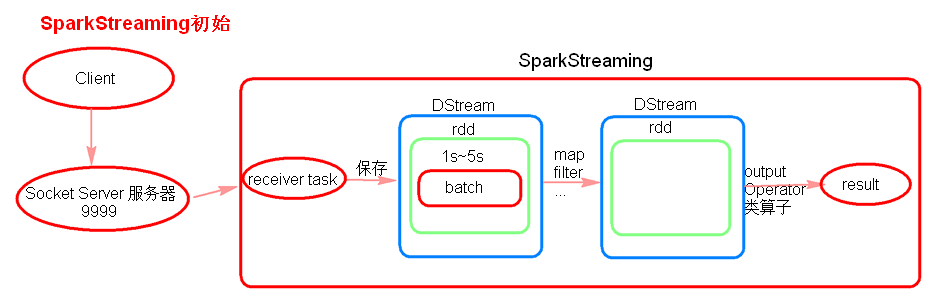
scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("windowfun")  **val** sc = **new** SparkContext(conf)  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  **val** result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop() |

1. SparkStreaming简介

SparkStreaming是流式处理框架，是Spark API的扩展，支持可扩展、高吞吐量、容错的实时数据流处理，实时数据的来源可以是：Kafka, Flume, Twitter, ZeroMQ或者TCP sockets，并且可以使用高级功能的复杂算子来处理流数据。例如：map,reduce,join,window 。最终，处理后的数据可以存放在文件系统，数据库等，方便实时展现。

1. SparkStreaming与Storm的区别
2. Storm是纯实时的流式处理框架，SparkStreaming是准实时的处理框架（微批处理）。因为微批处理，SparkStreaming的吞吐量比Storm要高。
3. Storm 的事务机制要比SparkStreaming的要完善。
4. Storm支持动态资源调度。(spark1.2开始和之后也支持)
5. SparkStreaming擅长复杂的业务处理，Storm不擅长复杂的业务处理，擅长简单的汇总型计算。
6. SparkStreaming初始
7. SparkStreaming初始理解



注意:

* receiver task是7\*24小时一直在执行，一直接受数据，将一段时间内接收来的数据保存到batch中。假设batchInterval为5s,那么会将接收来的数据每隔5秒封装到一个batch中，batch没有分布式计算特性，这一个batch的数据又被封装到一个RDD中，RDD最终封装到一个DStream中。

例如：假设batchInterval为5秒，每隔5秒通过SparkStreaming将得到一个DStream,在第6秒的时候计算这5秒的数据，假设执行任务的时间是3秒,那么第6~9秒一边在接收数据，一边在计算任务，9~10秒只是在接收数据。然后在第11秒的时候重复上面的操作。

* 如果job执行的时间大于batchInterval会有什么样的问题？

如果接受过来的数据设置的级别是仅内存，接收来的数据会越堆积越多，最后可能会导致OOM（如果设置StorageLevel包含disk, 则内存存放不下的数据会溢写至disk, 加大延迟 ）。

1. SparkStreaming代码

代码注意事项：

* 启动socket server 服务器：nc –lk 9999
* receiver模式下接受数据，local的模拟线程必须大于等于2，一个线程用来receiver用来接受数据，另一个线程用来执行job。
* Durations时间设置就是我们能接收的延迟度。这个需要根据集群的资源情况以及任务的执行情况来调节。
* 创建JavaStreamingContext有两种方式（SparkConf,SparkContext）
* 所有的代码逻辑完成后要有一个output operation类算子。
* JavaStreamingContext.start() Streaming框架启动后不能再次添加业务逻辑。
* JavaStreamingContext.stop() 无参的stop方法将SparkContext一同关闭，stop(false)，不会关闭SparkContext。
* JavaStreamingContext.stop()停止之后不能再调用start。

|  |
| --- |
| SparkConf conf = **new** SparkConf().setMaster("local[2]").setAppName("WordCountOnline");  /\*\*  \* 在创建streaminContext的时候 设置batch Interval  \*/  JavaStreamingContext jsc = **new** JavaStreamingContext(conf, Durations.*seconds*(5));  JavaReceiverInputDStream<String> lines = jsc.socketTextStream("node5", 9999);  JavaDStream<String> words = lines.flatMap(**new** FlatMapFunction<String, String>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Iterable<String> call(String s) {  **return** Arrays.*asList*(s.split(" "));  }  });  JavaPairDStream<String, Integer> ones = words.mapToPair(**new** PairFunction<String, String, Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Tuple2<String, Integer> call(String s) {  **return** **new** Tuple2<String, Integer>(s, 1);  }  });  JavaPairDStream<String, Integer> counts = ones.reduceByKey(**new** Function2<Integer, Integer, Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Integer call(Integer i1, Integer i2) {  **return** i1 + i2;  }  });    //outputoperator类的算子  counts.print();    jsc.start();  //等待spark程序被终止  jsc.awaitTermination();  jsc.stop(**false**); |

1. SparkStreaming算子操作
2. foreachRDD

* output operation算子,必须对抽取出来的RDD执行action类算子，代码才能执行。

1. transform

* transformation类算子
* 可以通过transform算子，对Dstream做RDD到RDD的任意操作。

1. updateStateByKey

* transformation算子
* updateStateByKey作用：

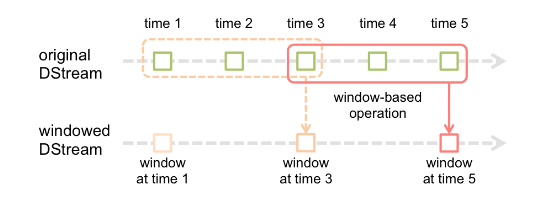
1. 为SparkStreaming中每一个Key维护一份state状态，state类型可以是任意类型的，可以是一个自定义的对象，更新函数也可以是自定义的。
2. 通过更新函数对该key的状态不断更新，对于每个新的batch而言，SparkStreaming会在使用updateStateByKey的时候为已经存在的key进行state的状态更新。

* 使用到updateStateByKey要开启checkpoint机制和功能。
* 多久会将内存中的数据写入到磁盘一份？

如果batchInterval设置的时间小于10秒，那么10秒写入磁盘一份。如果batchInterval设置的时间大于10秒，那么就会batchInterval时间间隔写入磁盘一份。

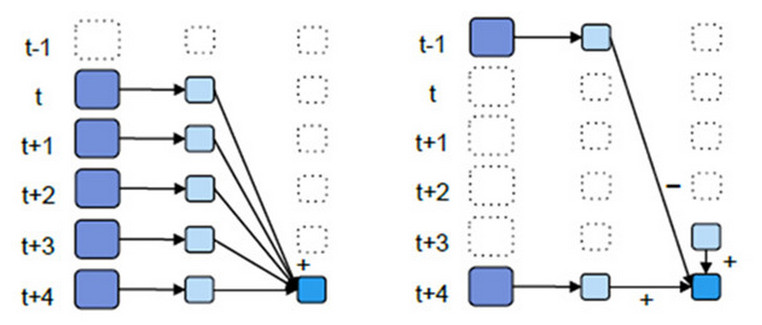
1. 窗口操作

* 窗口操作理解图：



假设每隔5s 1个batch,上图中窗口长度为15s，窗口滑动间隔10s。

* 窗口长度和滑动间隔必须是batchInterval的整数倍。如果不是整数倍会检测报错。
* 优化后的window窗口操作示意图：



* 优化后的window操作要保存状态所以要设置checkpoint路径，没有优化的window操作可以不设置checkpoint路径。

1. Driver HA（Standalone或者Mesos）

因为SparkStreaming是7\*24小时运行，Driver只是一个简单的进程，有可能挂掉，所以实现Driver的HA就有必要（如果使用的Client模式就无法实现Driver HA ，这里针对的是cluster模式）。Yarn平台cluster模式提交任务，AM(AplicationMaster)相当于Driver，如果挂掉会自动启动AM。这里所说的DriverHA针对的是Spark standalone和Mesos资源调度的情况下。实现Driver的高可用有两个步骤:

第一：提交任务层面，在提交任务的时候加上选项 --supervise,当Driver挂掉的时候会自动重启Driver。

第二：代码层面，使用JavaStreamingContext.getOrCreate（checkpoint路径，JavaStreamingContextFactory）

* Driver中元数据包括：

1. 创建应用程序的配置信息。
2. DStream的操作逻辑。
3. job中没有完成的批次数据，也就是job的执行进度。

# Spark day08

1. Driver HA（Standalone或者Mesos）

因为SparkStreaming是7\*24小时运行，Driver只是一个简单的进程，有可能挂掉，所以实现Driver的HA就有必要（如果使用的Client模式就无法实现Driver HA ，这里针对的是cluster模式）。Yarn平台cluster模式提交任务，AM(AplicationMaster)相当于Driver，如果挂掉会自动启动AM。这里所说的DriverHA针对的是Spark standalone和Mesos资源调度的情况下。实现Driver的高可用有两个步骤:

第一：提交任务层面，在提交任务的时候加上选项 --supervise,当Driver挂掉的时候会自动重启Driver。

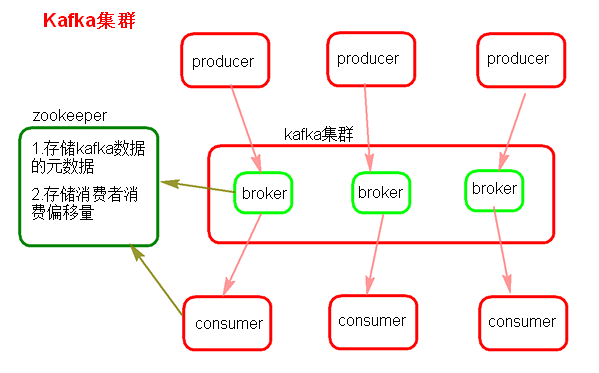
第二：代码层面，使用JavaStreamingContext.getOrCreate（checkpoint路径，JavaStreamingContextFactory）

* Driver中元数据包括：

1. 创建应用程序的配置信息。
2. DStream的操作逻辑。
3. job中没有完成的批次数据，也就是job的执行进度。
4. Kafka
5. kafka是什么？使用场景？

kafka是一个高吞吐的分布式消息队列系统。特点是生产者消费者模式，先进先出（FIFO）保证顺序，自己不丢数据，默认每隔7天清理数据。消息列队常见场景：系统之间解耦合、峰值压力缓冲、异步通信。

1. kafka生产消息、存储消息、消费消息



Kafka架构是由producer（消息生产者）、consumer（消息消费者）、borker(kafka集群的server，负责处理消息读、写请求，存储消息，在kafka cluster这一层这里，其实里面是有很多个broker)、topic（消息队列/分类相当于队列，里面有生产者和消费者模型）、zookeeper(元数据信息存在zookeeper中，包括：存储消费偏移量，topic话题信息，partition信息) 这些部分组成。

kafka里面的消息是有topic来组织的，简单的我们可以想象为一个队列，一个队列就是一个topic，然后它把每个topic又分为很多个partition，这个是为了做并行的，在每个partition内部消息强有序，相当于有序的队列，其中每个消息都有个序号offset，比如0到12，从前面读往后面写。一个partition对应一个broker，一个broker可以管多个partition，比如说，topic有6个partition，有两个broker，那每个broker就管3个partition。这个partition可以很简单想象为一个文件，当数据发过来的时候它就往这个partition上面append，追加就行，消息不经过内存缓冲，直接写入文件，kafka和很多消息系统不一样，很多消息系统是消费完了我就把它删掉，而kafka是根据时间策略删除，而不是消费完就删除，在kafka里面没有一个消费完这么个概念，只有过期这样一个概念。

producer自己决定往哪个partition里面去写，这里有一些的策略，譬如hash。consumer自己维护消费到哪个offset，每个consumer都有对应的group，group内是queue消费模型（各个consumer消费不同的partition，因此一个消息在group内只消费一次），group间是publish-subscribe消费模型，各个group各自独立消费，互不影响，因此一个消息在被每个group消费一次。

1. kafka的特点

* 系统的特点：生产者消费者模型，FIFO

Partition内部是FIFO的，partition之间呢不是FIFO的，当然我们可以把topic设为一个partition，这样就是严格的FIFO。

* 高性能：单节点支持上千个客户端，百MB/s吞吐，接近网卡的极限
* 持久性：消息直接持久化在普通磁盘上且性能好

直接写到磁盘中去，就是直接append到磁盘里去，这样的好处是直接持久化，数据不会丢失，第二个好处是顺序写，然后消费数据也是顺序的读，所以持久化的同时还能保证顺序，比较好，因为磁盘顺序读比较好。

* 分布式：数据副本冗余、流量负载均衡、可扩展

分布式，数据副本，也就是同一份数据可以到不同的broker上面去，也就是当一份数据，磁盘坏掉的时候，数据不会丢失，比如3个副本，就是在3个机器磁盘都坏掉的情况下数据才会丢，在大量使用情况下看这样是非常好的，负载均衡，可扩展，在线扩展，不需要停服务。

* 很灵活：消息长时间持久化+Client维护消费状态

消费方式非常灵活，第一原因是消息持久化时间跨度比较长，一天或者一星期等，第二消费状态自己维护消费到哪个地方了可以自定义消费偏移量。

1. kafka集群搭建
2. 上传kafka\_2.10-0.8.2.2.tgz包到三个不同节点上，解压。
3. 配置../ kafka\_2.10-0.8.2.2/config/server.properties文件

节点编号：（不同节点按0,1,2,3整数来配置）



真实数据存储位置：



zookeeper的节点：



1. 启动zookeeper集群。
2. 三个节点上，启动kafka:

|  |
| --- |
| bin/kafka-server-start.sh config/server.properties |

最好使用自己写的脚本启动，将启动命令写入到一个文件：

|  |
| --- |
| nohup bin/kafka-server-start.sh config/server.properties > kafka.log 2>&1 &  脚本附件：  **（放在与bin同一级别下，注意创建后要修改权限：chmod 755 startkafka.sh）** |

1. 相关命令：

创建topic：

|  |
| --- |
| **./kafka-topics.sh** **--zookeeper** node3:2181,node4:2181,node5:2181 **--create --topic** topic2017 **--partitions** 3 **--replication-factor** 3 |

用一台节点控制台来当kafka的生产者：

|  |
| --- |
| **./kafka-console-producer.sh**  **--topic** topic2017  **--broker-list** node1:9092,node2:9092,node3:9092 |

用另一台节点控制台来当kafka的消费者：

|  |
| --- |
| **./kafka-console-consumer.sh** **--zookeeper** node3:2181,node4:2181,node5:2181 **--topic** topic2017 |

查看kafka中topic列表:

|  |
| --- |
| **./kafka-topics.sh --list --zookeeper** node3:2181,node4:2181,node5:2181 |

查看kafka中topic的描述：

|  |
| --- |
| **./kafka-topics.sh --describe --zookeeper** node3:2181,node4:2181,node5:2181 --topic topic2017    注意：ISR是检查数据的完整性有哪些个节点。 |

查看zookeeper中topic相关信息：

|  |
| --- |
| 启动zookeeper客户端：  ./zkCli.sh  查看topic相关信息：  **ls** /brokers/topics/  查看消费者相关信息：  **ls** /consumers |

1. 删除kafka中的数据。
   1. ：在kafka集群中删除topic，当前topic被标记成删除。

|  |
| --- |
| ./kafka-topics.sh --zookeeper node3:2181,node4:2181,node5:2181 --delete --topic t1205 |

在每台broker节点上删除当前这个topic对应的真实数据。

* 1. ：进入zookeeper客户端，删除topic信息

|  |
| --- |
| rmr /brokers/topics/t1205 |

* 1. ：删除zookeeper中被标记为删除的topic信息

|  |
| --- |
| rmr /admin/delete\_topics/t1205 |

以上方式比较笨重，可以直接在每台broker节点中的../config/server.properties中配置属性：delete.topic.enable=true,重启kafka集群即可实现删除topic时，自动清除topic信息。

1. kafka的leader的均衡机制

当一个broker停止或者crashes时，所有本来将它作为leader的分区将会把leader转移到其他broker上去，极端情况下，会导致同一个leader管理多个分区，导致负载不均衡，同时当这个broker重启时，如果这个broker不再是任何分区的leader,kafka的client也不会从这个broker来读取消息，从而导致资源的浪费。

kafka中有一个被称为优先副本（preferred replicas）的概念。如果一个分区有3个副本，且这3个副本的优先级别分别为0,1,2，根据优先副本的概念，0会作为leader 。当0节点的broker挂掉时，会启动1这个节点broker当做leader。当0节点的broker再次启动后，会自动恢复为此partition的leader。不会导致负载不均衡和资源浪费，这就是leader的均衡机制。

在配置文件conf/ server.properties中配置开启（默认就是开启）：

|  |
| --- |
| auto.leader.rebalance.enable true |

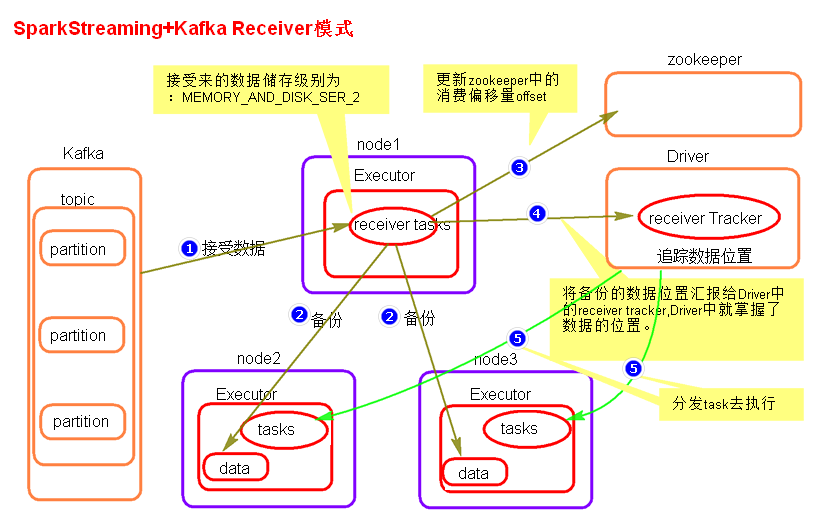
1. kafka 0.11版本改变

kafka 0.8.2版本消费者offset存储在zookeeper中，对于zookeeper而言每次写操作代价是很昂贵的，而且zookeeper集群是不能扩展写能力。kafka 0.11版本默认使用新的消费者api ,消费者offset会更新到一个kafka自带的topic【\_\_consumer\_offsets】中。以消费者组groupid 为单位，可以查询每个组的消费topic情况：

|  |
| --- |
| #查看所有消费者组  ./kafka-consumer-groups.sh --bootstrap-server c7node1:9092,c7node2:9092,c7node3:9092 --list  #查看消费者消费的offset位置信息  ./kafka-consumer-groups.sh --bootstrap-server c7node1:9092,c7node2:9092,c7node3:9092 --describe --group MyGroupId  #重置消费者组的消费offset信息 ，--reset-offsets –all-topics 所有offset。--to-earliest 最小位置。  # --execute 执行  ./kafka-consumer-groups.sh --bootstrap-server c7node1:9092,c7node2:9092,c7node3:9092 --group MyGroupId --reset-offsets --all-topics --to-earliest --execute |

1. SparkStreaming+Kafka
2. receiver模式

* receiver模式原理图



* receiver模式理解：

在SparkStreaming程序运行起来后，Executor中会有receiver tasks接收kafka推送过来的数据。数据会被持久化，默认级别为MEMORY\_AND\_DISK\_SER\_2,这个级别也可以修改。receiver task对接收过来的数据进行存储和备份，这个过程会有节点之间的数据传输。备份完成后去zookeeper中更新消费偏移量，然后向Driver中的receiver tracker汇报数据的位置。最后Driver根据数据本地化将task分发到不同节点上执行。

* receiver模式中存在的问题

当Driver进程挂掉后，Driver下的Executor都会被杀掉，当更新完zookeeper消费偏移量的时候，Driver如果挂掉了，就会存在找不到数据的问题，相当于丢失数据。

如何解决这个问题？

开启WAL(write ahead log)预写日志机制,在接受过来数据备份到其他节点的时候，同时备份到HDFS上一份（我们需要将接收来的数据的持久化级别降级到MEMORY\_AND\_DISK），这样就能保证数据的安全性。不过，因为写HDFS比较消耗性能，要在备份完数据之后才能进行更新zookeeper以及汇报位置等，这样会增加job的执行时间，这样对于任务的执行提高了延迟度。

* receiver模式代码（见代码）
* receiver的并行度设置

receiver的并行度是由spark.streaming.blockInterval来决定的，默认为200ms,假设batchInterval为5s,那么每隔blockInterval就会产生一个block,这里就对应每批次产生RDD的partition,这样5秒产生的这个Dstream中的这个RDD的partition为25个，并行度就是25。如果想提高并行度可以减少blockInterval的数值，但是最好不要低于50ms。

1. Driect模式

* Direct模式理解

SparkStreaming+kafka 的Driect模式就是将kafka看成存数据的一方，不是被动接收数据，而是主动去取数据。消费者偏移量也不是用zookeeper来管理，而是SparkStreaming内部对消费者偏移量自动来维护，默认消费偏移量是在内存中，当然如果设置了checkpoint目录，那么消费偏移量也会保存在checkpoint中。当然也可以实现用zookeeper来管理。

* Direct模式并行度设置

Direct模式的并行度是由读取的kafka中topic的partition数决定的。

* Direct模式代码（见代码）

1. 相关配置

预写日志:

|  |
| --- |
| spark.streaming.receiver.writeAheadLog.enable 默认false没有开启 |

blockInterval:

|  |
| --- |
| spark.streaming.blockInterval 默认200ms |

反压机制:

|  |
| --- |
| spark.streaming.backpressure.enabled 默认false |

接收数据速率:

|  |
| --- |
| spark.streaming.receiver.maxRate 默认没有设置 |

1. SparkStreaming2.3+kafka 改变
2. 丢弃了SparkStreaming+kafka 的receiver模式。
3. 采用了新的消费者api实现，类似于1.6中SparkStreaming 读取 kafka Direct模式。并行度一样。
4. 因为采用了新的消费者api实现，所有相对于1.6的Direct模式【simple api实现】 ，api使用上有很大差别。未来这种api有可能继续变化
5. kafka中有两个参数：

heartbeat.interval.ms：这个值代表 kafka集群与消费者之间的心跳间隔时间，kafka 集群确保消费者保持连接的心跳通信时间间隔。这个时间默认是3s.这个值必须设置的比session.timeout.ms appropriately 小，一般设置不大于 session.timeout.ms的1/3。

session.timeout.ms：这个值代表消费者与kafka之间的session 会话超时时间，如果在这个时间内，kafka 没有接收到消费者的心跳【heartbeat.interval.ms 控制】，那么kafka将移除当前的消费者。这个时间默认是10s。这个时间是位于 group.min.session.timeout.ms【6s】 和 group.max.session.timeout.ms【300s】之间的一个参数,如果SparkSteaming 批次间隔时间大于5分钟，也就是大于300s,那么就要相应的调大group.max.session.timeout.ms 这个值。

1. 大多数情况下，SparkStreaming读取数据使用 LocationStrategies.PreferConsistent 这种策略，这种策略会将分区均匀的分布在集群的Executor之间。

如果Executor在kafka 集群中的某些节点上，可以使用 LocationStrategies.PreferBrokers 这种策略，那么当前这个Executor 中的数据会来自当前broker节点。

如果节点之间的分区有明显的分布不均，可以使用 LocationStrategies.PreferFixed 这种策略,可以通过一个map 指定将topic分区分布在哪些节点中。

1. 新的消费者api 可以将kafka 中的消息预读取到缓存区中，默认大小为64k。默认缓存区在 Executor 中，加快处理数据速度。可以通过参数 spark.streaming.kafka.consumer.cache.maxCapacity 来增大，也可以通过spark.streaming.kafka.consumer.cache.enabled 设置成false 关闭缓存机制。
2. 关于消费者offset

1).如果设置了checkpoint ,那么offset 将会存储在checkpoint中。这种有缺点: 第一，当从checkpoint中恢复数据时，有可能造成重复的消费，需要我们写代码来保证数据的输出幂等。第二，当代码逻辑改变时，无法从checkpoint中来恢复offset.

2).依靠kafka 来存储消费者offset,kafka 中有一个特殊的topic 来存储消费者offset。新的消费者api中，会定期自动提交offset。这种情况有可能也不是我们想要的，因为有可能消费者自动提交了offset,但是后期SparkStreaming 没有将接收来的数据及时处理保存。这里也就是为什么会在配置中将enable.auto.commit 设置成false的原因。这种消费模式也称最多消费一次，默认sparkStreaming 拉取到数据之后就可以更新offset,无论是否消费成功。自动提交offset的频率由参数auto.commit.interval.ms 决定，默认5s。如果我们能保证完全处理完业务之后，可以后期异步的手动提交消费者offset。但是这种将offset存储在kafka中由参数offsets.retention.minutes=1440控制是否过期删除，默认是保存一天，如果停机没有消费达到时长，存储在kafka中的消费者组会被清空，offset也就被清除了。

3).自己存储offset,这样在处理逻辑时，保证数据处理的事务，如果处理数据失败，就不保存offset，处理数据成功则保存offset.这样可以做到精准的处理一次处理数据。