# 一、安装

## 1、解压

## 2、重命名

|  |
| --- |
| 两个文件都在conf下  mv spark-env.sh.template spark-env.sh  mv slaves.template slaves |

## 3、配置jdk和master节点

|  |
| --- |
| vi spark-env.sh  添加如下  export JAVA\_HOME=/opt/module/jdk1.8  exoort SPARK\_MASTER\_IP=hadoop102  export SPARK\_MASTER\_PORT=7077 |

## 4、配置worker节点

|  |
| --- |
| vi slaves  hadoop102  hadoop103  hadoop104 |

## 5、启动

|  |
| --- |
| bin/start-all.sh  可访问浏览器http://192.168.111.102:8080/，访问管理页面 |

## 6、高可用的配置spark slaves

|  |
| --- |
| 1、通过1-5步配置 会存在master单点故障  可借助zokeeper进行配置高可用的master（即配置多个master）  修改配置文件spark-env.sh，在该配置文件中删掉SPARK\_MASTER\_IP和SPARK\_MASTER\_PORT并添加如下配置  export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=hadoop102:2181,hadoop103:2181,hadoop104:2181 -Dspark.deploy.zookeeper.dir=/spark"  启动  在102(第一个master所有主机)使用命令 sbin/start-all.sh 启动集群  然后在第二master所有主机，启动master。 命令： bin/start-master.sh |

# 二、启动

## 1、提交一个任务

|  |
| --- |
| bin/spark-submit \  --class org.apache.spark.examples.SparkPi \  --master spark://hadoop102:7077 \  examples/jars/spark-examples\_2.11-2.2.0.jar \  100  设置修改化参数  /usr/local/spark-2.1.0-bin-hadoop2.6/bin/spark-submit \  --class org.apache.spark.examples.SparkPi \  --master spark://node1.edu360.cn:7077 \  --executor-memory 1G \  --total-executor-cores 2 \  /usr/local/spark-2.1.0-bin-hadoop2.6/lib/spark-examples-2.1.0-hadoop2.6.0.jar \  100  参数说明：  --master spark://node1.edu360.cn:7077 指定Master的地址,多个master可用“”  --executor-memory 2g 指定每个worker可用内存为2G  --total-executor-cores 2 指定整个集群使用的cup核数为2个  提交一个事务会启动，会启动sparksubmit进程，和多个coreExcutor进程 |

## 2、spark shell

|  |
| --- |
| bin/spark-shell --master spark://hadoop102:7077  启动交互式窗口，用scala语言进行编写 ，多个master可用“,”号隔开 |

## 3、流程

|  |
| --- |
| 0、mater先启动，woker启动后，向master通信注册地址和一些计算资源。  1、sparkSubmit提交任务，同时提交计算资源信息给master  2、master负责资源调度  3、master需要和worker进行rpc通信，让worker启动excuter  4、excuter和sparkSubmit通信，在submit生成任务，通过网络发给excuter执行  5、excuter接收任务，执行真正的业务逻辑 |

# 三、编写park的wordcount

## 1、代码

|  |
| --- |
| 见代码SparkTest |

## 2、本地模式调试程序

|  |
| --- |
| 在conf对象调用 .setMaster(“local[\*]”)  并设置程序运行参数，即输入文件，和输出路径 |

# 四、RDD

## 1、简介

|  |
| --- |
| RDD，弹性分布式数据集，RDD里面并还不存储真正要计算 的数据 ，你对RDD进行操作，他会在Driver端转换为成Task，下发到Executor，计算分散在多台集群上的数据 |
| RDD 是一个代理，就像操作本地集合一样，。降低了编程的复杂度  RDD的算子分为两类，Transformation（lazy）和action（触发任务执行）  RDD的创建  1、通过外部的存储系统创建RDD  2、将Driver的Scala集合通过并行化的方式编程RDD（试验、测验） |
| 3、调用一个已经存在了的RDd的Transformation，会生成一个新RDD |

## 2、简单算子

|  |
| --- |
| //通过并行化scala集合创建RDD  val rdd1 = sc.parallelize(Array(1,2,3,4,5,6,7,8))  //查看该rdd的分区数量  rdd1.partitions.length  //map方法同scala中的一样，将List中的每个数据拿出来做函数运算。  //sortBy：将数据进行排序  val rdd2 = sc.parallelize(List(5,6,4,7,3,8,2,9,1,10)).map(\_\*2).sortBy(x=>x,true)  //filter：将List中的每个数据进行函数造作，挑选出大于10的值。  val rdd3 = rdd2.filter(\_>10)  //collect：将最终结果显示出来  //flatMap:对数据先进行map操作，再进行flat（碾压）操作。  rdd4.flatMap(\_.split(’ ‘)).collect |

## 3、其它算子

|  |
| --- |
| RDD的算子.txt  map（func） 对分区的每个数据进行操作  mapValues（func） 对分区的每个数据进行操作  take |

## 4、mapParitionWithIndex

|  |
| --- |
| 一次拿出一个分区  功能 ：取分区中对应的数据时，还可以将分区的编译器一取出来，这样就可以知道 数据 是属于哪个分的，（哪个分区对应的task的数据）  该函数的功能 是将对应分区中的数据取出来，并且带上分区编号  Val func= (index: Int, it: Iterator[Int]) =>{  It.map(e => s”part:$index, ele: $e”)  }  使用：  Val rdd = sc.parallelize(List(1,2,3,4,5,6,7,8,9),2) //使用两个分区  val rdd1=rdd. mapParitionWithIndex(func) |
| rdd1.collect //执行 |

## 5、aggregate

|  |
| --- |
| Aggregate(initVal)(func1,func2)  功能：action算子，对每个分区的每个数据进行聚合，然后对每个分区的结果进行聚合  参数：  initVal 是初始值，这个值会参与次数为：分区数+1次。相当与每个分区里有这个数据，会参与计算，然后对分区结果汇总时，也会参与计算  func1是个函数，对每个分区里面的原数据进行操作，  func2是个函数，对每个分区计算的结果进行操作  val rdd = sc.parallelize(List(“1”,”2”,”3”,”4”))  rdd.aggregate(“|”)(\_+\_,\_+\_)  结果是 ｜｜12｜34 或者｜｜34｜12 |

## 6、aggregateByKey

|  |
| --- |
| AggregateByKey (initVal)(func1,func2)  功能：transformation算子，对每个分区的每个数据按**key**进行聚合，然后对每个分区的结果按**key**进行聚合  参数：  initVal 是初始值，这个值只会在分区里 才会 参与计算。全局不会参与计算，比Aggregate少一次  func1是个函数，对每个分区里面的原数据进行操作，  func2是个函数，对每个分区计算的结果进行操作  如  如val rdd=sc. Parallelize(List((“a”,1),(“b”,2),(“c”,3), (“a”,3),(“b”,3),))  val rdd1=rdd.aggregateByKey(0)(\_+\_ , \_+\_) |

## 7、collectAsMap

|  |
| --- |
| collectAsMap  是action算子  如val rdd=sc. Parallelize(List((“a”,1),(“b”,2)))  Val rdd1=rdd. mapValues(\_\*100)  rdd1.collectAsMap  结果：Map(b->200,a->100) |

## 8、foreach

|  |
| --- |
|  |

## 9、combineByKey

|  |
| --- |
| Parallelize（func1,func2,func3）  func2 分区的合并  func3 分区结果的合并  一个底层的算子  val pairRDD=sc.parallelize(List(("hello",2),("hello1",3),("hello",4),("hello1",5)),2)  val rdd2=pairRDD.combineByKey(x => x,(m: Int, n: Int) => m+n, (a: Int, b: Int) => a+b)  Array[(String, Int)] = Array((hello,6), (hello1,8))  val rdd5 = sc.parallelize(List(1,1,2,2,2,1,2,2,2),3)  val rdd4 = sc.parallelize(List(“dog”, “cat”, “gnu”, “salmon”, “wolf”, “rabbit”, “turkey”, “bear”, “bee”),3)  val rdd6 = rdd5.zip(rdd4)  val rdd7 = rdd6.combineByKey(  x => ListBuffer(x),  (m: LIstBuffer[String], n: String) => m += n,  (a: ListBuffer[String], b: ListBuffer[String] ) => a++= b  )  Rdd7.collect |
| 可改变分区（最后输出的文件个数）  val rdd7 = rdd6.combineByKey(  x => ListBuffer(x),  (m: LIstBuffer[String], n: String) => m += n,  (a: ListBuffer[String], b: ListBuffer[String] ) => a++= b，  new HashPartitioner(2),  true,  null  )  倒数第二个参数，指定局部是否合并  最后个参数，序列化器  需要引入HashPartitioner  执行命令 import org.apache.spark.HashPartitioner |

## 10、reduceByKey

|  |
| --- |
| reduceByKey（func1）  底层调用combineByKey(x => x,func1,func1) |

## 11、sortBy 与自定义排序

|  |
| --- |
| SortBy(para)  排序 ，RDD的算子，内存+磁盘上进行排序 |

### 11.1自定义排序1

|  |
| --- |
| 方法1：定义一个排序规则类  class User(val name: String, val age: Int, val fv: Int) extends Ordered[User] with Serializable {  override def compare(that: User): Int = {  if(this.fv == that.fv) {  this.age - that.age  } else {  -(this.fv - that.fv)  }  }  override def toString: String = s"name: $name, age: $age, fv: $fv"  }  使用一：将数据封装成User，然后调用sortBy()  val userRDD: RDD[User] = lines.map(line => {  //其它操作  new User(name, age, fv)  })  userRDD.sortBy(u => u )  使用二：数据还是使用元组的方式，在sortBy时进行封装。此种方式不会改变，数据的数据结构.此时use对象只需要传入需要比较的字段即可。  val userRDD: RDD[User] = lines.map(line => {  //其它操作  (name, age, fv)  })  userRDD.sortBy(u=> new User(u.\_1.u.\_2,u\_3) ) //这里u.\_1是多余的，可以不要。  使用三：在使用二的情况下进行改进。  将User对象在声明时，对传入参数不声明成val。也不用实现序列化。  将  class User(val name: String, val age: Int, val fv: Int) extends Ordered[User] with Serializable  改成  class User(name: String, age: Int, fv: Int) extends Ordered[User]  使用时，也不用每次都new一个对象，像这样  userRDD.sortBy(u=> User(u.\_1.u.\_2,u\_3) ) //这里u.\_1是多余的，可以不要。 |

### 11.2自定义排序2

|  |
| --- |
| 1、声明一个Object,即排序规则 和一个calss  case class XianRou(age: Int, fv: Int)  object SortRules {  implicit object OrderingXiaoRou extends Ordering[XianRou] {  override def compare(x: XianRou, y: XianRou): Int = {  if(x.fv == y.fv) {  x.age - y.age  } else {  y.fv - x.fv  }  }  }  }  2、使用时，传入隐式参数  //排序(传入了一个排序规则，不会改变数据的格式，只会改变顺序)  import SortRules.OrderingXiaoRou  val sorted: RDD[(String, Int, Int)] = tpRDD.sortBy(tp => XianRou(tp.\_2, tp.\_3)) |

### 11.3自定义排序3

|  |
| --- |
| 1、充分利用元组的排序特性  //切分整理数据  val tpRDD: RDD[(String, Int, Int)] = lines.map(line => {  //其它操作  (name, age, fv)  })  使用一：  //充分利用元组的比较规则，元组的比较规则：先比第一，相等再比第二个  val sorted: RDD[(String, Int, Int)] = tpRDD.sortBy(tp => (-tp.\_3, tp.\_2))  使用二：  //Ordering[(Int, Int)]最终比较的规则格式  //on[(String, Int, Int)]未比较之前的数据格式  //(t =>(-t.\_3, t.\_2))怎样将规则转换成想要比较的格式  implicit val rules = Ordering[(Int, Int)].on[(String, Int, Int)](t =>(-t.\_3, t.\_2)) /./定义规则  val sorted: RDD[(String, Int, Int)] = tpRDD.sortBy(tp => tp) |

## 12、filter

|  |
| --- |
| Filter(condition)  过滤 |

## 13、partitionBy和自定义分区器

|  |
| --- |
| 作用： 决定了上游的数据到下游的哪个分区  //自定义分区器  class SubjectParitioner(sbs: Array[String]) extends Partitioner {  //相当于主构造器（new的时候回执行一次）  //用于存放规则的一个map  val rules = new mutable.HashMap[String, Int]()  var i = 0  for(sb <- sbs) {  //rules(sb) = i  rules.put(sb, i)  i += 1  }  //返回分区的数量（下一个RDD有多少分区）  override def numPartitions: Int = sbs.length  //根据传入的key计算分区标号  //key是一个元组（String， String）  override def getPartition(key: Any): Int = {  //获取学科名称  val subject = key.asInstanceOf[(String, String)].\_1  //根据规则计算分区编号  rules(subject)  }  }  使用：  val partitioned: RDD[((String, String), Int)] = reduced.partitionBy(sbPatitioner) |

## 14、mapPartitions

|  |
| --- |
| mapPartitions(func1(it : iterator =>{}))  作用：一次拿一个分区的数据  //如果一次拿出一个分区(可以操作一个分区中的数据了)  val sorted: RDD[((String, String), Int)] = partitioned.mapPartitions(it => {  //将迭代器转换成list，然后排序，在转换成迭代器返回  it.toList.sortBy(\_.\_2).reverse.take(topN).iterator  }) |

## 15、cache

|  |
| --- |
| Cache到内存中的Rdd，必需是反复使用的rdd  val rdd =sc.textFile("hdfs://hadoop102:8020/spark/access")  val cached = rdd.cache() #cache并没有生成新的rdd，而是标记一个rdd，放入内存  cached .count  释放缓存  rdd.unuersist  若内存不够，它可以只缓存部分数据到内存中  什么时候使用缓存  1、要求的计算速度快  2、集群的资源要足够大  3、cache的数据要多次执行action的操作  4、cache的数据最好是进行过滤的  Cache（）的底层调用了，persist（）算子。  persist()算子的参数，存储层级对象的参数如下：  //第一个参数 放到磁盘  //第二个参数 放到内存  //第三个参数 磁盘中的数据 ，不是以java对象的方式保存  //第四个参数 内存中的数据 ，以java对象的方式保存  var MEMORY\_AND\_DISK = new StorageLevel(true, true, true, true,)  OFF\_HEAP: 不用jvm的内存管理系统，使用堆外内存。 |

## 16、checkpoint

|  |
| --- |
| 什么时候做checkpoint  1、迭代计算，要求保证数据安全  2、对速度要求不高（跟cache到内存进行对比）  3、将中间结果保存到hdfs  怎么做：  1、设置checkpoint的目录（分布式文件系统的目录）  2、经过复杂计算，得到中间结果  3、将中间结果checkpoint到指定的目录  4、后续的计算可以使用 直接使用中间结果  如：  sc.setCheckpointDir(“hdfs://hadoop102:8020/ch20190423 ”) #对行创建对应的目录  val rdd= ...  rdd.checkpoint  rdd.count  checkpoint后的rdd会取消掉之前该rrd对其它的依赖，因为可以直接从hdfs中读取 |

## 17、broadcast

|  |
| --- |
| //将Driver端的数据广播到Executor  //广播变量的引用（还在Driver端）  val rulesInDriver = 。。。 （driver端的数据）  val broadcastRef: Broadcast[Array[(Long, Long, String)]] = sc.broadcast(rulesInDriver) |
| //Task是在Driver端生成的，广播变量的引用是伴随着Task被发送到Executor中的  val rulesInExecutor: Array[(Long, Long, String)] = broadcastRef.value |

18、连接数据库

|  |
| --- |
| def data2MySQL(it: Iterator[(String, Int)]): Unit = {  //一个迭代器代表一个分区，分区中有多条数据  //先获得一个JDBC连接  val conn: Connection =  DriverManager.getConnection("jdbc:mysql://localhost:3306/bigdata?characterEncoding=UTF-8", "root", "123568")  //将数据通过Connection写入到数据库  val pstm: PreparedStatement = conn.prepareStatement("INSERT INTO access\_log VALUES (?, ?)")  //将分区中的数据一条一条写入到MySQL中  it.foreach(tp => {  pstm.setString(1, tp.\_1)  pstm.setInt(2, tp.\_2)  pstm.executeUpdate()  })  //将分区中的数据全部写完之后，在关闭连接  if(pstm != null) {  pstm.close()  }  if (conn != null) {  conn.close()  }  } |

## 18、foreach 与 foreachPartition

|  |
| --- |
| 两个都是action算子  Foreach (line : string => { } )遍历每条数据。  ForeachPartition （it : iterator<string> => {  //..其它操作  it.foreach(line : string =>{})  }）遍历每个分区。  作用：如在将分区的数据写入数据库。就只需要建立一个connection就行，过多浪费性能。用后者可以处理此种情况。 |

## 19、jdbcRDD

|  |
| --- |
| //new 了RDD，里面没有真正要计算的数据，而是告诉这个RDD，以后触发Action时到哪里读取数据  val jdbcRDD: RDD[(Int, String, Int)] = new JdbcRDD(  sc,  getConn,  "SELECT \* FROM logs WHERE id >= ? AND id < ?",  1,  5,  2, //分区数量  rs => {  val id = rs.getInt(1)  val name = rs.getString(2)  val age = rs.getInt(3)  (id, name, age)  }  )  //注意sql中这个<小于号。会导致数据丢失。换成<=即可 |

# 五、wordcount的执行流程

|  |
| --- |
| 加看视频spark-03-TopN与WordCount执行过程详解 |

# 六、需求

|  |
| --- |
| 根据 访问日志的ip地址计算出访问者的归属地，并且按照省份，计算访问次数，然后将计算好的结果写入到mysql  1、整理数据，切分出ip字段，然后将ip地址转换成十进制  2、加载规则，整理规则，取出有用的字段，然后将数据 缓存到内存中（executor的内存）  3、将访问log与ip规则进行匹配（二分法查找）  4、取出 对应 的省份名称，然后将其它和一组全在一起  5、近省份名进行聚合  6、钭聚合后的数据写入mysql中 |

# 七、SparkSQL

## 7.1 是什么

|  |
| --- |
| 功能：将sql语句转换成spark的计算程序。相当于hive将sql转换成mapreduce程序  相关代码见 sparkTest.day6 |

## 7.2 DataFrames

|  |
| --- |
| 相当于特殊换Rdd也是一个分布式的数据容器。  DataFrames很好的继承了传统单机数据分析的开发体验。 |

## 7.3 DataFrames的创建

|  |
| --- |
| 一、创建方式（一）  0、记取数据  Val lineRDD = sc.textFile(“hdfs://192.168.111.102:9000”);  1、定义case clacc (相当于schema，即表信息)  case class Person(id:Int, name:String, age:Int)  2、将RDD与case class并联。  Val personRdd = lineRdd.map( x => Person(x(0),toIne,x(2),x(2).toInt))  3、将RDD轮换成DataFrame  Val persionDF = persionRDD.toDF  4、对DataFrame进行处理  personDF.show  一、创建方式（二）  0、记取数据  Val lineRDD = sc.textFile(“hdfs://192.168.111.102:9000”);  1、将RDD与Row关联  Val rowRdd = lineRdd.map( x =>{ Row(id, name, age, fv)})  2、定义schema  //结果类型，其实就是表头，用于描述DataFrame  val schema: StructType = StructType(List(  StructField("id", LongType, true),  StructField("name", StringType, true),  StructField("age", IntegerType, true),  StructField("fv", DoubleType, true)  ))  3、创建DataFrame,此时创建需要使用spark2.x的编程api（本质上只是对之前的api进行再次的封装），2.x编程api见4  Val perssonDF=session.createDataFrame(rowRdd,schema)  4、2.x编程api  val session = SparkSession.builder()  .appName("SQLTest1")  .master("local[\*]")  .getOrCreate()  //创建RDD  val lines: RDD[String] = session.sparkContext.textFile("hdfs://node-4:9000/person") |
|  |

## 7.4 常用操作

|  |
| --- |
| 一、DSL风格语法  1、查看DataFrame中的内容  persionDF.show  2、查看DataFrame中部份列的内容  personDF.select(“name”,”age”,”fv”)  3、打印DataFrame中的Schema信息  persionDF.printSchema  4、查询所有的name和age，并将age+1  persionDF.select(col(“id”),col(“name”).col(“age”)+1).show  personDF.select(personDF(“id”), personDF(“name”), personDF(“age”)+1).show  5、过滤age大于18的  personDF.filter(col(“age”) >= 18).show  6、按年龄进行分组并统计相同年龄的人数  personDF.groupBy("age").count().show()  二、sql风格语法  如果想使用SQL风格的语法，需要将DataFrame注册成表  personDF.registerTempTable("t\_person")  personDF.registerTempView("v\_person")  //查询年龄最大的前两名  sqlContext.sql("select \* from t\_person order by age desc limit 2").show  //显示表的Schema信息  sqlContext.sql("desc t\_person").show |

## 7.5 join

|  |
| --- |
| 一、DSL风格  //import org.apache.spark.sql.functions.\_  val r = df1.join(df2, $"nation" === $"ename", "left\_outer")  二、sql风格  df1.createTempView("v\_users")  df2.createTempView("v\_nations")  val r: DataFrame = spark.sql("SELECT name, cname FROM v\_users JOIN v\_nations ON nation = ename")  三、spark sql 中的两种join  broadcastHashJoin 与 SortMergJoin  3.1 broadcastHashJoin  broadcastHashJoin实际上可细分为两种，broadcast hash join 与shuffle hash join  但原理都是hash join |
| 原理：  1、读取左表数据，对join key 进行hash分桶。  2、读取右表数据，使用相同的hash函数对join key进行hash分桶，如果匹配成功再查检hash key是否相同，相同的就可以join成功  broadcast hash join 是将数据进行广播，只适用于小数量数据。  shuffle hash join 是通过join key 进行分区，key相同的数据，必然在一个分区里。数据量比broadcast hash join稍大  3.2 SortMergJoin  1) shuffle阶段，将两张大表对join key进行重新分区。  2）sort阶段，对单个分区节点的两表数据进行排序  3）merge阶段，对排序好的两张分区表进行join。分别遍历两个有序序列，碰到相同的join key就输出  3.3 使用  // shuffle hash join  df1.join(df2,$”id” === $”aid”)  // broadcast hash join,  //可以设置广播的数据量大小, Spark.conf.set(“spark.sql.autoBroadcastJoinThreshold”,-1)  df1.join(broadcast(df2),$”id” === $”aid”)  //将广播数据量大小设置为-1，即为sort merg join.  Spark.conf.set(“spark.sql.autoBroadcastJoinThreshold”,-1)  df1.join(df2,$”id” === $”aid”) |

## 7.6 自定义函数

|  |
| --- |
| 一、UDF（User Defined Function）  定义和使用  1、1使用  //该函数的功能是（输入一个IP地址对应的十进制，返回一个省份名称）  spark.udf.register("ip2Province", (ipNum: Long) => {  //查找ip规则（事先已经广播了，已经在Executor中了）  //函数的逻辑是在Executor中执行的，怎样获取ip规则的对应的数据呢？  //使用广播变量的引用，就可以获得  val ipRulesInExecutor: Array[(Long, Long, String)] = broadcastRef.value  //根据IP地址对应的十进制查找省份名称  val index = MyUtils.binarySearch(ipRulesInExecutor, ipNum)  var province = "未知"  if(index != -1) {  province = ipRulesInExecutor(index).\_3  }  province  })  1、2 使用  在sql里直接使用  val r = spark.sql("SELECT ip2Province(ip\_num) province, COUNT(\*) counts FROM v\_log GROUP BY province ORDER BY counts DESC")  二、UDAF  2、1定义  定义一个class，extents，UserDefinedAggregateFunction，重写里面的方法  class GeoMean extends UserDefinedAggregateFunction {  //输入数据的类型  override def inputSchema: StructType = StructType(List(  StructField("value", DoubleType)  ))  //产生中间结果的数据类型  override def bufferSchema: StructType = StructType(List(  //相乘之后返回的积  StructField("product", DoubleType),  //参与运算数字的个数  StructField("counts", LongType)  ))  //最终返回的结果类型  override def dataType: DataType = DoubleType  //确保一致性 一般用true  override def deterministic: Boolean = true  //指定初始值  override def initialize(buffer: MutableAggregationBuffer): Unit = {  //相乘的初始值  buffer(0) = 1.0  //参与运算数字的个数的初始值  buffer(1) = 0L  }  //每有一条数据参与运算就更新一下中间结果(update相当于在每一个分区中的运算)  override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {  //每有一个数字参与运算就进行相乘（包含中间结果）  buffer(0) = buffer.getDouble(0) \* input.getDouble(0)  //参与运算数据的个数也有更新  buffer(1) = buffer.getLong(1) + 1L  }  //全局聚合  override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  //每个分区计算的结果进行相乘  buffer1(0) = buffer1.getDouble(0) \* buffer2.getDouble(0)  //每个分区参与预算的中间结果进行相加  buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)  }  //计算最终的结果  override def evaluate(buffer: Row): Double = {  math.pow(buffer.getDouble(0), 1.toDouble / buffer.getLong(1))  }  }  2、2使用  Sql方式使用  New 一个这个对象，然后注册函数，即可在sql中使用  val geomean = new GeoMean  spark.udf.register("gm", geomean)  val result = spark.sql("SELECT gm(id) result FROM v\_range")  DSL方式使用  New 一个这个对象，调用Dataset.的rgg算子。  val result = range.agg(geomean($"id").as("geomean"))  三、UDTF  无 |

## 7.7 多数据源支持

### 7.7.1 JDBC

|  |
| --- |
| import spark.implicits.\_  val logs: DataFrame = spark.read.format("jdbc").options(  Map("url" -> "jdbc:mysql://localhost:3306/bigdata",  "driver" -> "com.mysql.jdbc.Driver",  "dbtable" -> "logs",  "user" -> "root",  "password" -> "123568")  ).load()  val reslut: DataFrame = r.select($"id", $"name", $"age" \* 10 as "age")  reslut.show() |

### 7.7.2 JSON/CSV

|  |
| --- |
| import spark.implicits.\_  val jsons: DataFrame = spark.read.json("/Users/zx/Desktop/json")  val filtered: DataFrame = jsons.where($"age" <=500)  filtered.printSchema()  filtered.show()  val csv:DataFrame =spark.read.csv(“/...s”)  val pdf: DataFrame = csv.toDF("id", "name", "age") |

### 7.7.3 PAQUET

|  |
| --- |
| val parquetLine: DataFrame = spark.read.parquet("/Users/zx/Desktop/parquet")  parquetLine.printSchema()  //show是Action  parquetLine.show() |

## 7.8 写多数据源

|  |
| --- |
| //reslut.write.text("/Users/zx/Desktop/text")  //reslut.write.json("/Users/zx/Desktop/json")  //reslut.write.csv("/Users/zx/Desktop/csv")  //reslut.write.parquet("hdfs://node-4:9000/parquet") |

## 7.8 spark中支持hive

### 7.8.1 spark启动hive的终端

|  |
| --- |
| 1、在spark/conf下配置hive-site.xml  指明mysql的数据库 连接、驱动、登录用户和密码  2、上传一个mysql连接驱动（sparkSubmit也要连接MySQL，获取元数据信息）  ./spark-sql --master spark://node-4:7077,node-5:7077 --driver-class-path /home/xiaoniu/mysql-connector-java-5.1.7-bin.jar  或者将jar包放到sprark的jar(lib?)目录 |
| 3. sparkSQL会在mysql上创建一个database，需要手动改一下DBS表中的DB\_LOCATION\_UIR改成hdfs的地址  4.要在/etc/profile中配置一个环节变量(让sparkSQL知道hdfs在哪里，其实就是namenode在哪里)  exprot HADOOP\_CONF\_DIR  5.重新启动SparkSQL的命令行 |

### 7.8.1 spark程序中启用hive

|  |
| --- |
| <!-- spark如果想整合Hive，必须加入hive的支持 -->  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-hive\_2.11</artifactId>  <version>2.2.0</version>  </dependency>  将hadoop和hive的配置文件拷贝到项目的resources目录，其目的是为了让spark程序找到hdfs和数据库。  有三个文件 core-site.xml hdfs-site.xml hive-site.xml  val spark = SparkSession.builder()  .appName("HiveOnSpark")  .master("local[\*]")  .enableHiveSupport()//启用spark对hive的支持(可以兼容hive的语法了)  .getOrCreate() |

# 八、spark streaming

## 8.1 实时计算相关技术

|  |
| --- |
| Strom / JStrom Spark Streming Flink  实时性高 有延迟 实时性高  吞吐量较低 吞吐量高 吞吐量高  只能实时计算 离线+实时 离线+实时  算子比较少 算子丰富 算子丰富  没有 机器学习 没有  没有 图计算 没有  使用比较少 非常火 一般  一个完整的生态是非常重要的，spark生态特别完善Flink |
|  |

## 8.2 Kafka的一些概念

|  |
| --- |
| Broker ： 安装Kafka服务的那台集群就是一个broker（broker的id要全局唯一）  Producer ：消息的生产者，负责将数据写入到broker中（push）  Consumer：消息的消费者，负责从kafka中读取数据（pull），老版本的消费者需要依赖zk，新版本的不需要  Topic: 主题，相当于是数据的一个分类，不同topic存放不同的数据  Consumer Group： 消费者组，一个topic可以有多个消费者同时消费，多个消费者如果在一个消费者组中，那么他们不能重复消费数据  Spark Streaming 2.2.0兼容kafka 0.8.2.1以上的版本，主要支持0.8和0.10这两个版本 |

## 8.3 Kafka的安装

|  |
| --- |
| 1、解压  2、修改配置文件 server.property  Broker,id=0 #全局唯一  log.dir= #数据存储的位置  host.name= #当前机器的ip  zookeeper.connect = #zk的地址，多个zk，用逗号分割  3、集群配置，将配置好的kafka拷贝到其它机器  注意修改 broker.id和host.name  4、启动 (依次启动集群中的kafka，可在用jps查看是否有kafka进程)  bin/Kafka-server-start.sh –daemon confif/server.properties  5、操作  #查看topic信息  ./kafka-topics.sh –list  --zookeeper hadoop102:2181,hadoop103:2181,hadoop104:2181  #创建topic  /bigdata/kafka\_2.11-0.8.2.2/bin/kafka-topics.sh --create  --zookeeper hadoop102:2181,hadoop103:2181,hadoop104:2181  --replication-factor 3 --partitions 3 --topic xiaoniu  #往Kafka的topic中写入数据(命令行的生成者)  ./kafka-console-producer.sh --broker-list  hadoop102:9092,hadoop103:9092,hadoop104:9092 --topic xiaoniu  #启动消费者  /bigdata/kafka\_2.11-0.8.2.2/bin/kafka-console-consumer.sh  --zookeeper hadoop102:2181,hadoop103:2181,hadoop104:2181  --topic xiaoniu --from-beginning |

## 8.4 Kafka的java api

### 8.4.1 producer

|  |
| --- |
| public class ProducerDemo {   public static void main(String[] args) {  Properties props = new Properties();  props.put("metadata.broker.list", "node-4:9092,node-5:9092,node-6:9092");  props.put("serializer.class", "kafka.serializer.StringEncoder");  ProducerConfig config = new ProducerConfig(props);  Producer<String, String> producer = new Producer<String, String>(config);  for (int i = 1001; i <= 1100; i++)  producer.send(new KeyedMessage<String, String>("xiaoniu", "xiaoniu-msg" + i));  } } |

### 8.4.2 consumer

|  |
| --- |
| public class ConsumerDemo {  private static final String *topic* = "xiaoniu";  private static final Integer *threads* = 2;   public static void main(String[] args) {    Properties props = new Properties();  props.put("zookeeper.connect", "node-1:2181,node-2:2181,node-3:2181");  props.put("group.id", "vvvvv");  //smallest重最开始消费,largest代表重消费者启动后产生的数据才消费  //--from-beginning  props.put("auto.offset.reset", "smallest");   ConsumerConfig config = new ConsumerConfig(props);  ConsumerConnector consumer =Consumer.*createJavaConsumerConnector*(config);  Map<String, Integer> topicCountMap = new HashMap<String, Integer>();  topicCountMap.put(*topic*, *threads*);  Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCountMap);  List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(*topic*);    for(final KafkaStream<byte[], byte[]> kafkaStream : streams){  new Thread(new Runnable() {  public void run() {  for(MessageAndMetadata<byte[], byte[]> mm : kafkaStream){  String msg = new String(mm.message());  System.*out*.println(msg);  }  }  }).start();  }  } } |

## 8.5 kafka的相关概念

|  |
| --- |
| Broker ： 安装Kafka服务的那台集群就是一个broker（broker的id要全局唯一）  Producer ：消息的生产者，负责将数据写入到broker中（push）  Consumer：消息的消费者，负责从kafka中读取数据（pull），老版本的消费者需要依赖zk，新版本的不需要  Topic: 主题，相当于是数据的一个分类，不同topic存放不同的数据  Consumer Group： 消费者组，一个topic可以有多个消费者同时消费，多个消费者如果在一个消费者组中，那么他们不能重复消费数据 |

## 8.6 recvice方式

从kafka实时获取数据，并且计算wordcount

以下这种方式要达到固定时间才运行，效率低，并且容易丢失数据。使用高级api。

|  |
| --- |
| **object** StatefulKafkaWordCount {   */\*\*  \* 第一个参数：聚合的key，就是单词  \* 第二个参数：当前批次产生批次该单词在每一个分区出现的次数  \* 第三个参数：初始值或累加的中间结果  \*/* **val** *updateFunc* = (iter: Iterator[(String, Seq[Int], Option[Int])]) => {  //iter.map(t => (t.\_1, t.\_2.sum + t.\_3.getOrElse(0)))  iter.map{ **case**(x, y, z) => (x, y.sum + z.getOrElse(0))}  }   **def** main(args: Array[String]): Unit = {   **val** conf = **new** SparkConf().setAppName("StatefulKafkaWordCount").setMaster("local[\*]")   **val** ssc = **new** StreamingContext(conf, *Seconds*(5))  //如果要使用课更新历史数据（累加），那么就要把终结结果保存起来  ssc.checkpoint("./ck")  **val** zkQuorum = "node-1:2181,node-2:2181,node-3:2181"  **val** groupId = "g100"  **val** topic = *Map*[String, Int]("xiaoniuabc" -> 1)  //创建DStream，需要KafkaDStream  **val** data: ReceiverInputDStream[(String, String)] = KafkaUtils.*createStream*(ssc, zkQuorum, groupId, topic)  //对数据进行处理  //Kafak的ReceiverInputDStream[(String, String)]里面装的是一个元组（key是写入的key，value是实际写入的内容）  **val** lines: DStream[String] = data.map(\_.\_2)  //对DSteam进行操作，你操作这个抽象（代理，描述），就像操作一个本地的集合一样  //切分压平  **val** words: DStream[String] = lines.flatMap(\_.split(" "))  //单词和一组合在一起  **val** wordAndOne: DStream[(String, Int)] = words.map((\_, 1))  //聚合  **val** reduced: DStream[(String, Int)] = wordAndOne.updateStateByKey(*updateFunc*, **new** HashPartitioner(ssc.sparkContext.defaultParallelism), **true**)  //打印结果(Action)  reduced.print()  //启动sparksteaming程序  ssc.start()  //等待优雅的退出  ssc.awaitTermination()   } |

## 8.6 direct直连方式

从kafka实时获取数据，并且计算wordcount

|  |
| --- |
| Day09.KafkaDirectWordCountV2 |

# 九、spark on yarn