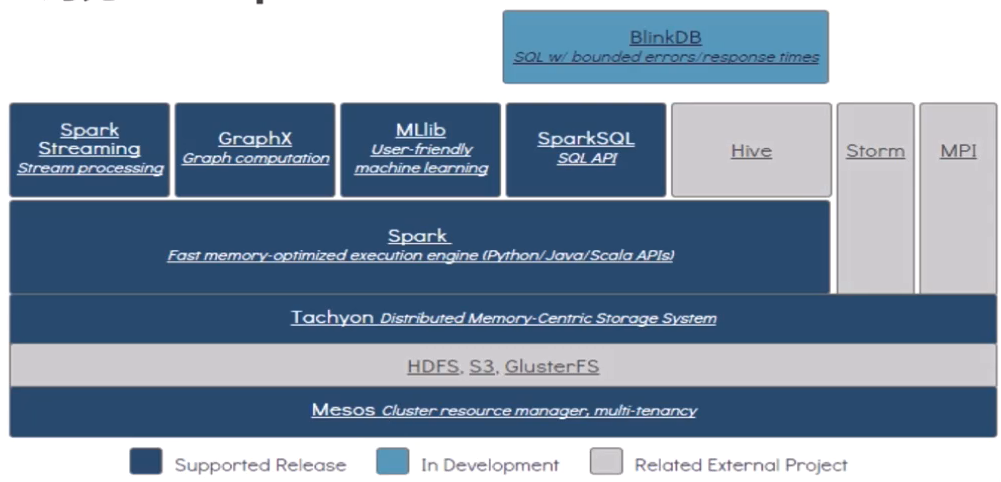
分布式计算框架Spark特点：计算快，通用，开发容易，可以访问任何数据库以及HDFS分布式文件系统

Spark、Kafka、Flink的源码语言是Scala，Hadoop和Hive的源码语言是Java，Storm的源码语言是Clojure但已经被阿里用Java重写成了JStorm贡献给Apache基金会

Spark生态系统BDAS（Berkeley Data Analysis Stacks）：



Mesos：集群资源调度管理框架类似YARN，YARN（Java语句开发）是粗粒度的资源调度管理框架，Mesos（c++语句开发）是可粗可细（可根据使用资源的真实情况进行分配和收回相应的资源）的资源调度管理框架，支持多租户

Tachyon：基于内存的分布式文件系统，HDFS是基于磁盘的分布式文件系统

Spark：基于内存优化的快速执行引擎

Spark Streaming：基于内存优化的快速执行流式处理的引擎框架

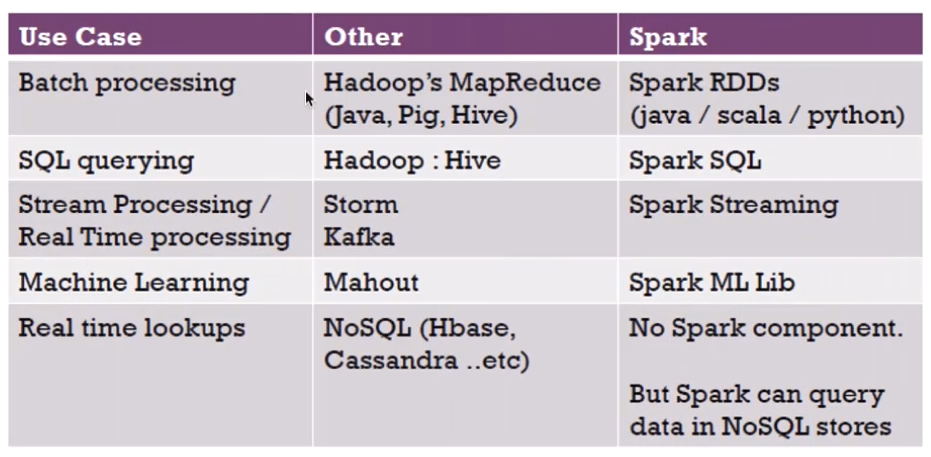
GraphX：图形计算框架

MLlib：机器学习框架

SparkSQL：SQL语言API

BlinkDB：在海量式数据上进行交互式快速查询引擎，以牺牲数据精准度提升查询响应时间的引擎

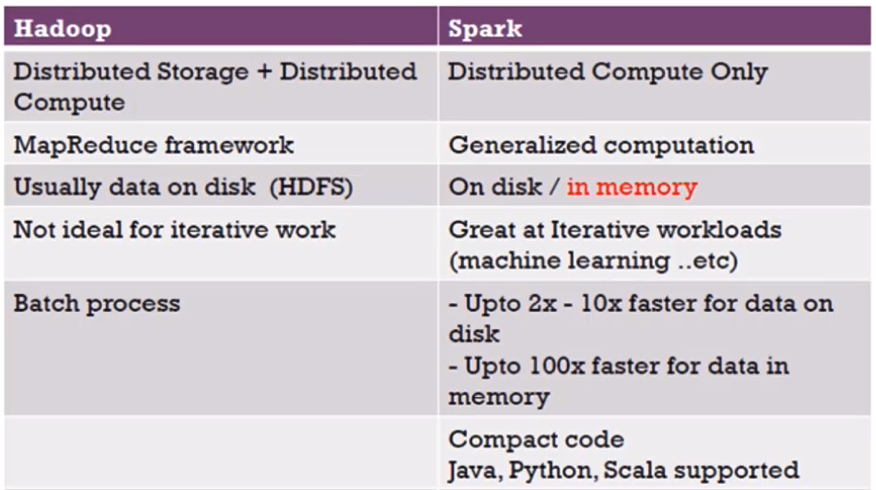
Hadoop生态圈与Spark BDAS对比



|  |  |  |
| --- | --- | --- |
| 场景 | Hadoop | Spark |
| 批处理 | Hadoop’s MapReduce(Java,Pig,Hive) | Spark RDDs(Java/Scala/Python) |
| SQL查询 | Hadoop:Hive | Spark SQL |
| 流处理/实时处理 | Storm Kafka | Spark Streaming |
| 机器学习 | Mahout | Spark ML Lib |
| 实时数据查找 | NoSQL(Hbase,Cassandra…etc) | Spark是计算引擎不关注数据存储在哪里，所有没有相应的Spark组件与实时数据查找对应，但Spark可以在非关系型数据库中查询数据 |

Hadoop与Spark对比

|  |  |
| --- | --- |
| Hadoop | Spark |
| 分布式存储系统，分布式计算框架，资源调度框架 | 只关注分布式系统的计算 |
| 使用MapReduce框架 | 通用计算框架 |
| 数据存储在磁盘（HDFS） | 磁盘/内存（内存中效率更高） |
| 不适合迭代次数多的工作，每次完成的数据需要落地即存到HDFS或者其他存储位置上 | 擅长迭代次数多的工作，因为基于内存计算所以数据不需要落地，例如机器学习 |
| 批处理框架 | 可以批处理，在磁盘上比Hadoop快2到10倍，在内存上快100倍以上 |
|  | 支持多种语言Java,Python,Scala |



Spark开发语言支持Python，Scala，Java，R语言，Spark大部分代码是由Scala语言开发

Spark运行模式：YARN，Local，Standalone是Spark内置的运行模式，Mesos是Spark最初开始支持的运行模式

Local模式Spark启动：解压Spark安装包

复制修改Spark安装目录/conf/spark-env.sh

cp spark-env.sh.template spark-env.sh

在配置文件中添加（默认使用系统环境变量的JDK）：

export JAVA\_HOME=JDK安装路径

不需要启动进程可以直接使用命令Spark安装目录/bin/spark-shell --master local[\*/数值] 以本地模式最大/指定数值线程数启动Spark

Spark集群安装：

Spark集群最小数量为2，安装JDK

上传解压安装包，解压安装包到指定位置

tar -zxvf spark-1. 6.1-bin-hadoop2 6 tgz -C /usr/local

复制修改Spark安装目录/conf/spark-env.sh

cp spark-env.sh.template spark-env.sh

在配置文件中添加：

export JAVA\_HOME=JDK安装路径

export SPARK\_MASTER\_IP=hadoop1 指定Spark的Master节点

export SPARK\_MASTER\_PORT=7077 指定Spark的RPC端口

export SPARK\_WORKER\_CORES=1 指定Spark的每个Worker进程运行的核心数

export SPARK\_WORKER\_INSTANCES=1 指定Spark集群每个节点运行Worker进程的实例数

export SPARK\_WORKER\_MEMORY=1g 指定Spark每个Worker进程运行的内存

复制修改Spark安装目录/conf/slaves文件：添加Spark集群的work节点主机名或IP地址

Spark安装目录/sbin/start-all.sh 启动Spark集群

启动后jps命令查看进程，主节点上有Master进程，其他节点上有Worker进程，Spark默认的webUI监控界面<http://主机名或IP地址:8080>

Spark安装目录/bin/spark-submit --deploy-mode client/cluster –supervise（监管，可省略） --class jar包的入口类全类名 --master Spark的master节点RPC地址 --executor-memory Worker启动的Executor可用内存大小 --total-executor-cores 执行任务的总CPU核心数 jar包的路径名 [jar包的参数] 以Client/Cluster模式（Client模式会在提交任务的节点控制台输出任务日志等信息，Cluster模式会在集群的随机Worker节点上启动任务Application对应的Driver管理任务的运行和结果，日志结果可以在Driver节点查看）提交任务到Spark集群

Spark高可用HA安装：

安装配置ZooKeeper集群，启动ZooKeeper集群

修改Spark安装目录/conf/spark-env.sh，在配置文件中添加：

export JAVA\_HOME=JDK安装路径

export SPARK\_MASTER\_PORT=7077 指定Spark的RPC端口

export SPARK\_WORKER\_CORES=1 指定Spark的每个Worker进程运行的核心数

export SPARK\_WORKER\_INSTANCES=1 指定Spark集群每个节点运行Worker进程的实例数

export SPARK\_WORKER\_MEMORY=1g 指定Spark每个Worker进程运行的内存

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=hadoop1:2181,hadoop2:2181,hadoop3:2181 -Dspark.deploy.zookeeper.dir=/home/kaku/tmp/spark"

修改s1aves配置文件指定Worker节点

在集群的节点上使用Spark安装目录/sbin/start-all.sh启动的节点默认为Spark高可用主节点Master，使用Spark安装目录/sbin/start-master.sh在指定节点上启动Spark的备用Master节点standby状态（ZooKeeper配置的Spark高可用热备自动切换Master会同步宕机Master节点的任务，但切换Master期间不能接受提交的任务）

Spark安装目录/bin/spark-shell --master spark://master节点主机名或IP地址:端口号（默认7077） --executor-memory Worker启动的执行任务的Executor进程可用内存大小带单位 --total-executor-cores 集群执行任务的总核心数（即Executor的数量）

Spark任务jar包依赖的jar包需要在Spark安装目录/conf/spark-defaults.conf文件中配置：

Spark.driver.extraClassPath=jar包路径名

Spark.executor.extraClassPath=jar包路径名

Spark Shell被Spark当做特殊任务运行，Spark Shell启动后Master节点会相应产生用于提交任务的SparkSubmit进程，Worker节点会相应产生用于计算的CoarseGrainedExecutorBackend进程

spark实现WordCount：

val file = sc.textFile("file:///home/文件路径")读取文件

val a = file.flatMap(line => line.split(" "))将行按照空格拆分

val b = a.map(word => (word,1))为每个单词添加数字1

val c = b.reduceByKey(\_ + \_)按照Key相同的统计

sc.textFile("file:///home/hadoop/data/hello.txt").flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(\_ + \_).collect Spark实现WordCount的代码合并

Apache的档案库：<http://archive.apache.org/>

object SparkStudy {

def main(args: Array[String]): Unit = {

val conf:SparkConf=new SparkConf().setAppName("SparkWC").setMaster("local[2]")//配置信息类，设置应用程序名称以本地两个线程模拟集群

val conf:SparkConf=new SparkConf().setAppName("SparkWC")//通过打包上传至服务器集群启动

val sparkContext:SparkContext=new SparkContext(conf)//上下文对象

val lines=sparkContext.textFile(args(0))//读取数据

val words=lines.flatMap(\_.split(" "))//读取到的数据变平分隔

val paired=words.map((\_,1))

val reduced=paired.reduceByKey(\_+\_)

val res=reduced.sortBy(\_.\_2,false)

res.saveAsTextFile(args(1))//保存数据

println(res.collect().toBuffer)

sparkContext.stop()//结束任务释放资源

}

}

使用Maven进行编译Scala需要引入Scala插件

<build>

<!--scala待编译的文件目录-->

<sourceDirectory>src/main/scala</sourceDirectory>

<testSourceDirectory>src/test/java</testSourceDirectory>

<!--scala插件-->

<plugins>

<plugin>

<groupId>net.alchim31.maven</groupId>

<artifactId>scala-maven-plugin</artifactId>

<version>3.2.2</version>

<executions>

<execution>

<goals>

<goal>compile</goal>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

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

RDD的属性：

组分片( Partition)：即数据集的基本组成单位，是具体在集群某个节点上的一片数据，对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度，可以在创建RDD时指定RDD的分片个数，默认为程序所分配到的CPU Core的数目

计算每个分区的函数：Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的，compute函数会对迭代器进行复合，不需要保存每次计算的结果

RDD之间的依赖关系：

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

RDD的分片函数Partitioner：当前 Spark中实现了两种类型的分片函数，一个是基于哈希的 Hashpartitioner，另外一个是基于范围的 RangePartitioner，只有对于key-value的RDD才会有Partitioner，非key-value的RDD的partitioner的值是None，Partitioner函数决定RDD本身的分片数量，也决定parent RDD Shuffle输出时的分片数量

列表：存储存取每个Partition的优先位置（preferred location），对于一个HDFS文件来说，列表保存的就是每个Partition所在的块的位置，按照移动数据不如移动计算的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置

Spark生成RDD的两种方式SparkContext.textFile()和SparkContext.parallelize()

Spark的算子（与Scala的函数或者方法类似相对应）分为转换Transformation和动作Action类型，为了提高Spark的运行效率，transformation的算子只是计算的逻辑并不直接进行计算，而是在action的算子执行时将逻辑加入到action算子中一起计算，每一个action算子是一个任务

RDD相关算子

map() 按照函数参数映射处理

sortBy() 按照函数参数进行排序

filter() 过滤

flatMap() 变平映射处理

union() 并集

intersection() 交集

distinct() 去重

collect() 数据集的所有元素作为数组返回

toBuffer 转换为可变缓存，一般用于输出显示

join 连接

leftOuterJoin 左外连接

rightOuterJoin 右外连接

groupByKey() 按照键值分组

mapValues() 按照值value进行映射处理

reduceByKey() 按照键值聚合

reduce() 聚合

cogroup() 聚合

sortByKey() 按照键值排序

cartesian() 笛卡尔积

count() 计算数量

top() 从前获取

take() 获取前几个元素

first() 获取第一个元素

takeOrdered() 排序后获取前几个元素

class ProjectPartitioner(projects:Array[String])extends Partitioner{//使用自定义分区，需要继承Partitioner抽象类，并实现numPartitions和getPartition方法

private val projectAndPartNum=new mutable.HashMap[String,Int]()//用于存放数据String已经对应的区号

var n=0

for (pro<-projects){

projectAndPartNum+=(pro->n)

n+=1

}

override def numPartitions: Int = projects.length//获取分区数量

override def getPartition(key: Any): Int = {//获取分区号

projectAndPartNum.getOrElse(key.toString,0)

}

}

object PartionStudy {

def main(args: Array[String]): Unit = {

val sparkConf=new SparkConf().setAppName("").setMaster("")

val sparkContext=new SparkContext(sparkConf)

sparkContext.broadcast()//将需要的数据广播发送到集群中相应的Executor，可以提高Spark获取数据的效率，但要考虑硬件内存的配置

val file=sparkContext.textFile("D://in")

val urlAndOne=file.map(line=>{//将数据映射成元组

val fields=line.split("\t")

val url=fields(1)

(url,1)

})

val sumeUrl=urlAndOne.reduceByKey(\_+\_)

val cachedProject=sumeUrl.map(x=>{

val url=x.\_1

val project=new URL(url).getHost

val count=x.\_2

(project,(url,count))

}).cache()//缓存指定数据到内存，提高运行效率，RDD是转换状态，当产生RDD转换成新的RDD时，只会保留RDD的依赖关系，但旧的RDD数据就不存在了，一般可以通过RDD.persist()缓存RDD数据到内存进行复用减少硬盘的IO

val res=cachedProject.partitionBy(new HashPartitioner(3))//调用Spark的分区器，可能会产生数据倾斜，一般根据需要自定义需要的分区器进行分区，防止数据倾斜

val projects=cachedProject.keys.distinct().collect()//获取数据的所有类型

val partitioner=new ProjectPartitioner(projects)//调用自定义分区器获取分区号

val partitioned=cachedProject.partitionBy(partitioner)//数据进行分区

val res=partitioned.mapPartitions(it=>{

it.toList.sortBy(\_.\_2.\_2).reverse.take(3).iterator

})

res.saveAsTextFile("c://out")

sparkContext.stop()

}

}

Spark集群启动流程：启动 Master进程，Master进程启动完成后会解析slaves配置文件找到启动Worker的host，然后启动相应的Worker并发送注册相关的信息给Worker，Worker开始与Master进行注册，把注册信息发送给Master，Master收到注册信息后并保存到内存和磁盘里，Master给Worker发送注册成功的消息（masterURL），Worker收到Master的URL信息后，开始与 Master建立心跳

Spak任务提交流程：Client的SparkSubmit进程和Master进行通信，创建SparkContext对象，Master收到任务信息后开始资源凋度和所有的Worker进行通信，找到较空闲的Worker并通知Worker启动Executor（CoarseGrainedExecutorBackend）子进程，Executor进程启动后，开始与Client通信，Client开始把生成的任务提交到相应的Executor进行执行计算任务

RDD与子RDD的依赖关系：

窄依赖（Narrow Dependencies）：父RDD的分区最多被子RDD的一个分区使用

宽依赖（Wide Dependencies）：多个子RDD的分区会依赖同一个父RDD的分区

血统（Lineage）：记录RDD的元数据和转换行为，以便恢复丢失的分区提高Spark的运算效率

RDD的一系列转换可以看做是有向无环图（DAG），Spark任务会根据宽依赖（或Shuffler）划分为不同阶段Stage，源码是通过递归进行划分Stage，人工进行划分Stage可以从后向前推，Spark任务的子任务数一般为Stage与分区数的乘积，Spark的一个子任务Task对应一个管道pipeline或者血统lineage

//import MySort.girlOrdering

//case class Girl(faceValue:Int,age:Int)

//object MySort{//自定义排序需要实现特质Ordering的compare隐式方法并需要import引用

// implicit val girlOrdering=new Ordering[Girl]{

// override def compare(x: Girl, y: Girl): Int = {

// if (x.faceValue!=y.faceValue){

// x.faceValue-y.faceValue

// }else{

// -(x.age-y.age)

// }

// }

// }

//}

case class Girl(faceValue:Int,age:Int)extends Ordered[Girl]{//自定义排序需要继承Ordered类实现compare不需要import引用

override def compare(that: Girl): Int ={

if (this.faceValue!=that.faceValue){

this.faceValue-that.faceValue

}else{

that.age-this.age

}

}

}

object CustomSort {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setAppName("CustomSort").setMaster("local[2]")

val sc=new SparkContext(conf)

val girlInfo=sc.parallelize(Array(("ting",80,25),("ning",90,26),("mi",90,27)))

// val res=girlInfo.sortBy(\_.\_2,false)

val res=girlInfo.sortBy(x=>Girl(x.\_2,x.\_3),false)

println(res.collect().toBuffer)

sc.stop()

}

}

checkpoint：将数据以二进制文件的形式保存为检查点，运行的中间结果（一般Shuffler之后的数据）很重要，为了保证数据的安全性，一般将数据checkpoint到HDFS，便于集群所有节点访问，一般先将数据放到缓存cache然后checkpoint，便于运行任务的调用，也便于checkpoint的时候可以从cache中获取数据

SparkContext.setCheckpointDir("") 设置检查点数据存放目录一般为HDFS

RDD.cache() 先将运行结果数据放入缓存

RDD.checkpoint() 将数据以二进制文件存放到检查点，一般为Shuffler之后的RDD

RDD.coalesce()一般在RDD.filter()之后使用减少或增加Partition的数量

DataFrames是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格

DataFrames比RDD多带有schema元信息即DataFrames所表示的二维表数据集的每一列都带有名称和类型，这使得Spark SQL得以洞察更多的结构信息，从而对藏于DataFrames背后的数据源以及作用于DataFrames之上的变换进行了针对性的优化，大幅提升运行时效率，RDD由于无从得知所存数据元素的具体内部结构，Spark Core只能在Stage层面进行简单、通用的流水线优化

Spark SQL Core

Spark SQL的核心是把已有的RDD，带上Schema信息注册成类似SQL里的Table表，对其进行SQL查询，主要可以分两部分生成SchemaRDD和执行查询，由于RDD的各种变换实际上只是在构造 RDD DAG，DataFrames的各种变换同样也是lazy的，Spark SQL不直接求出计算结果而是将各种变换组装成与RDD DAG类似的逻辑查询计划，由于DataFrames带有Schema元信息，Spark SQL的查询优化器得以洞察数据和计算的精细结构施行具有很强针对性的优化，经过优化的逻辑执行计划被翻译为物理执行计划最终落实为RDD DAG

case class Person(id:Int,name: String,age:Int,faceValue:Int) 定义DataFrames表对应的类

val line=sparkContext.textFile("").map(\_.split(",")) 获取数据

val personRDD=line.map(x=>Person(x(0).toInt,x(1),x(2).toInt,x(3).toInt)) 将数据转换成类对象

import sqlContext.implicits.\_ toDF方法需要引入隐式转换

val personDF=personRDD.toDF 数据类对象转换成DataFrames

personDF.show 显示DataFrames

DataFrames可以使用领域特定语言DSL（domain-specific language）和SQL进行操作

领域特定语言DSL风格：

personDF.select(personDF.col("字段名")).show 查询表

personDF.select(col("字段名")).show 查询表，可以在字段上进行操作

personDF.select(personDF("字段名")).show 查询表，可以在字段上进行操作

personDF.select("字段名").show 查询表

personDF.filter(col("字段名")>=数值).show 过滤表

personDF.groupBy("字段名").count().show 分组统计表

SQL语言风格：

personDF.registerTempTable("表名") 将DataFrames注册成指定表名的表

sqlContext.sql("SQL语句") 执行指定SQL语句

通过反射推断schema：

val sqlContext=new SQLContext(sparkContext) 程序实例化SQLContext对象

val df=sqlContext.sql("") 获取SQL的执行结果DataFrames

df.write.json(args(1)) 将数据写入指定地址

通过StructType直接指定schema：

val schema=StructType( 定义StructType

List(

[StructField("字段名",IntegerType,true)]

)

)

val rowRDD=line.map(x=>Row(x(0).toInt)) 将数据映射成行对象，不需要定义样例类

val personDF=sqlContext.createDataFrame(rowRDD,schema) 将行对象与StructType映射并创建相应DataFrames

Hive On Spark：使用Hive的元数据表结合Spark的计算引擎进行任务的分析

Hive On Spark的配置：

将Hadoop的配置文件core-site.xml和Hive的配置文件hive-site.xml复制到Spark安装目录/conf目录下，由于是使用Hive的元数据表信息，所以可以不启动Hive但需要启动Hive的元数据库MySQL

Spark安装目录/bin/spark-sql --master Spark的RPC地址 –executor-memory 每个Worker可用内存大小带单位 --total-executor-cores 任务使用的总核心数 --driver-class-path MySQL数据库的驱动包jar路径名 启动Hive On Spark

Spark Streaming是Spark API的扩展，可实现实时数据流的可扩展，高吞吐量，容错流处理，Spark Streaming接收实时输入数据流（TCP套接字）并将数据分成批处理，然后由Spark引擎处理以批量生成最终结果流，Storm是真正实时处理，Spark Streaming是以短期分批处理近似实时处理但吞吐量比Storm高

Spark Streaming提供称为离散流或DStream的高级抽象，表示连续的数据流，Spark Streaming获取服务器的数据到Spark的Executor进程中，Executor进程的Receiver线程用于接收获取数据，Executor进程的Executor线程用于执行处理数据

object StreamingWC {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setAppName("SteamingWC").setMaster("local[2]") 设置模拟Spark Streaming的配置，由于Executor进程要启动Receiver和Executor线程，模拟线程数量最小为2,

val ssc=new StreamingContext(conf,Seconds(5)) 创建Spark Streaming上下文对象，批次处理的时间间隔设置为指定数值单位秒

ssc.checkpoint("hdfs://hadoop1:8020/test") 设置检查点

val dStream=ssc.socketTextStream("IP地址",8888) 获取Netcat的实时数据

val tuples=dStream.flatMap(\_.split(" ")).map((\_,1)) 将数据转换成元组

val res=tuples.updateStateByKey(func,new HashPartitioner(ssc.sparkContext.defaultParallelism),false) 将数据执行自定义处理函数

val res1=tuples.reduceByKeyAndWindow((a:Int,b:Int)=>(a+b),Seconds(10),Seconds(10)) 使用窗口函数按照指定自定义处理函数获取数据的变化，窗口长度和滑动间隔设置为指定时间间隔单位秒

res.print() 显示

ssc.start() 启动Spark Streaming

ssc.awaitTermination() Spark Streaming等待数据

}

val func=(it:Iterator[(String,Seq[Int],Option[Int])])=>{ 自定义处理函数，String为元组的数据，Seq[]当前批次执行的结果，Option[]以往批次执行的结果

it.map(t=>{

(t.\_1,t.\_2.sum+t.\_3.getOrElse(0))

})

}

}

窗口函数：Spark Streaming一段时间内数据发生的变化即数据依据按照Spark Streaming的时间间隔进行批次处理，窗口函数对批次处理的指定时间长度和间隔的数据进行获取，两个参数必须是DStream批次间隔的倍数

窗口长度：窗口的持续时间

滑动间隔：窗口操作的间隔

Spark资源调度模式：

Standalone模式：属于Spark独有的资源调度框架，不依赖其他的调度框架，对于Spark效率最高的资源调度框架

Local模式：单机模式

On YARN模式：Spark运行在YARN资源调度框架上，由于历史遗留问题以前使用YARN资源调度框架比较成型，部分离线计算任务没有迁移至Spark中，真正实时计算框架Storm需要YARN，以及节约运维成本，提供的Spark的一种运行模式

Spark On YARN配置：

安装Hadoop，启动HDFS与YARN，Spark运行时会将程序jar包放到HDFS上

安装Spark修改Spark安装目录/conf/spark-env.sh文件：

export JAVA\_HOME=JDK安装路径

export HADOOP\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop 指定Hadoop的配置文件的目录

export YARN\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop 指定Hadoop的配置文件的目录

export SPARK\_MASTER\_PORT=7077 指定Spark的RPC端口

export SPARK\_WORKER\_CORES=1 指定Spark的每个Worker进程运行的核心数

export SPARK\_WORKER\_INSTANCES=1 指定Spark集群每个节点运行Worker进程的实例数

export SPARK\_WORKER\_MEMORY=1g 指定Spark每个Worker进程运行的内存

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=hadoop1:2181,hadoop2:2181,hadoop3:2181 -Dspark.deploy.zookeeper.dir=/home/kaku/tmp/spark"

Spark可以不需要安装集群只安装单机以及不启动Spark，Spark仅作为YARN的客户端提交任务，启动HDFS和YARN

Spark提交任务的两种模式：

Cluster模式：程序在YARN中运行，应用的运行结果不能在客户端显示，最好是保存在外部存储介质，而非stdout输出的应用程序，客户端显示的是作为YARN的job的简单运行状况

Client模式：程序运行在Client上，应用程序运行结果会在客户端显示，一般用于测试环境

Spark程序Cluster模式处理过程：

Spark提交任务到ResourceManager，ResourceManager在对任务job在集群的所有NodeManager节点上启动ApplicationMaster进程（管理Spark任务的整个生命周期）并向ResourceManager进行注册，ApplicationMaster从HDFS上获取程序jar文件并启动SparkContext、DAGscheduler和YARN Cluster Scheduler，ResourceManager通知NodeManager分配Container（一个Container对应一个Executor），Spark ApplicationMaster直接与Container进行任务交互

Spark安装目录/bin/spark-submit --deploy-mode yarn-client/yarn-cluster --driver-memory ApplicationMaster进程可用内存大小 –supervise（监管，可省略，Driver挂掉会自动启动新Driver） --class jar包的入口类全类名 --master Spark的master节点RPC地址 --executor-memory Worker启动的Executor可用内存大小 --total-executor-cores 执行任务的总CPU核心数 jar包的路径名 [jar包的参数] 以YARN的Client/Cluster模式提交任务到Spark集群

Spark On YARN的任务的应用状态查看http://主机名（或者IP地址）:8088/，可以根据yarn-client提交的Spark任务YARN的Web界面信息的名称为任务类名，yarn-cluster提交的Spark任务YARN的Web界面信息的名称为任务全类名

Spark On YARN提交任务时有可能会因为内存分配问题导致任务失败，可以对yarn-site.xml进行配置：

<configuration>

<property>

<name>yarn.scheduler.maximum-allocation-mb</name>

<value>9000</value>

<discription>每个任务最多可用内存,默认8182MB</discription>

</property>

<property>

<name>yarn.scheduler.minimum-allocation-mb</name>

<value>3072</value>

<discription>每个任务最小可用内存</discription>

</property>

<property>

<name>yarn.nodemanager.vmem-pmem-ratio</name>

<value>3</value> ###物理内存和虚拟内存比率

</property>

</configuration>

Spark内存使用模式参考地址：<https://0x0fff.com/spark-architecture/>和<https://0x0fff.com/spark-memory-management/>