目录20180810

[1 基础 6](#_Toc524553922)

[1.1 113\_Scala编程进阶：Scaladoc的使用 6](#_Toc524553923)

[1.2 114\_Scala编程进阶：跳出循环语句的3种方法 6](#_Toc524553924)

[1.3 115\_Scala编程进阶：多维数组、Java数组与Scala数组的隐式转换 6](#_Toc524553925)

[1.4 116\_Scala编程进阶：Tuple拉链操作、JavaMap与ScalaMap的隐式转换 7](#_Toc524553926)

[1.5 117\_Scala编程进阶：扩大内部类作用域的2种方法、内部类获取外部类引用 7](#_Toc524553927)

[1.6 118\_Scala编程进阶：package与import实战详解 10](#_Toc524553928)

[1.7 119\_Scala编程进阶：重写field的提前定义、Scala继承层级、对象相等性 17](#_Toc524553929)

[1.8 120\_Scala编程进阶：文件操作实战详解 19](#_Toc524553930)

[1.9 121\_Scala编程进阶：偏函数实战详解 23](#_Toc524553931)

[1.10 122\_Scala编程进阶：执行外部命令 23](#_Toc524553932)

[1.11 123\_Scala编程进阶：正则表达式支持 24](#_Toc524553933)

[1.12 124\_Scala编程进阶：提取器实战详解 26](#_Toc524553934)

[1.13 125\_Scala编程进阶：样例类的提取器实战详解 27](#_Toc524553935)

[1.14 126\_Scala编程进阶：只有一个参数的提取器 28](#_Toc524553936)

[1.15 127\_Scala编程进阶：注解实战详解 29](#_Toc524553937)

[1.16 128\_Scala编程进阶：常用注解介绍 30](#_Toc524553938)

[1.17 129\_Scala编程进阶：XML基础操作实战详解 32](#_Toc524553939)

[1.18 130\_Scala编程进阶：XML中嵌入scala代码 33](#_Toc524553940)

[1.19 131\_Scala编程进阶：XML修改元素实战详解 34](#_Toc524553941)

[1.20 132\_Scala编程进阶：XML加载和写入外部文档 35](#_Toc524553942)

[1.21 133\_Scala编程进阶：集合元素操作 35](#_Toc524553943)

[1.22 134\_Scala编程进阶：集合的常用操作方法 36](#_Toc524553944)

[1.23 135\_Scala编程进阶：map、flatMap、collect、foreach实战详解 37](#_Toc524553945)

[1.24 136\_Scala编程进阶：reduce和fold实战详解1 37](#_Toc524553946)

[1.25 138\_环境搭建-Hadoop25伪分布式集群搭建 37](#_Toc524553947)

[1.26 139\_环境搭建-Spark15伪分布式集群搭建 39](#_Toc524553948)

[1.27 140\_第一次课程升级大纲介绍以及要点说明 40](#_Toc524553949)

[1.28 141\_Spark核心编程进阶-Spark集群架构概览 40](#_Toc524553950)

[1.29 142\_Spark核心编程进阶-Spark集群架构的几点特别说明 40](#_Toc524553951)

[1.30 143\_Spark核心编程进阶-Spark的核心术语讲解 41](#_Toc524553952)

[1.31 145\_Spark核心编程进阶-单独启动master和worker脚本详解 42](#_Toc524553953)

[1.32 146\_Spark核心编程进阶-实验：单独启动master和worker进程 45](#_Toc524553954)

[1.33 147\_Spark核心编程进阶-worker节点配置以及spark-evnsh参数详解 48](#_Toc524553955)

[1.34 148\_Spark核心编程进阶-实验：local模式提交spark作业 53](#_Toc524553956)

[1.35 149\_Spark核心编程进阶-实验：standalone client模式提交spark作业 55](#_Toc524553957)

[1.36 150\_Spark核心编程进阶-实验：standalonecluster模式提交spark作业 57](#_Toc524553958)

[1.37 151\_Spark核心编程进阶-standalone模式下的多作业资源调度 59](#_Toc524553959)

[1.38 152\_Spark核心编程进阶-standalone模式下的作业监控与日志记录 60](#_Toc524553960)

[1.39 153\_Spark核心编程进阶-实验：运行中作业监控以及手工打印日志 62](#_Toc524553961)

[1.40 154\_Spark核心编程进阶-yarn-client模式原理讲解 62](#_Toc524553962)

[1.41 155\_Spark核心编程进阶-yarn-cluster模式原理讲解 63](#_Toc524553963)

[1.42 156\_Spark核心编程进阶-实验：yarn-client模式提交spark作业 64](#_Toc524553964)

[1.43 157\_Spark核心编程进阶-yarn模式下日志查看详解 66](#_Toc524553965)

[1.44 158\_Spark核心编程进阶-yarn模式相关参数详解 68](#_Toc524553966)

[1.45 159\_Spark核心编程进阶-spark工程打包以及spark-submit详解 69](#_Toc524553967)

[1.46 160\_Spark核心编程进阶-spark-submit示例以及基础参数讲解 70](#_Toc524553968)

[1.47 163\_Spark核心编程进阶-spark-submit多个示例以及常用参数详解 72](#_Toc524553969)

[1.48 164\_Spark核心编程进阶-SparkConf、spark-submit以及spark-defaultsconf 74](#_Toc524553970)

[1.49 165\_Spark核心编程进阶-spark-submit配置第三方依赖 76](#_Toc524553971)

[1.50 166\_Spark核心编程进阶-spark算子的闭包原理详解 77](#_Toc524553972)

[1.51 167\_Spark核心编程进阶-实验：对闭包变量进行累加操作的无效现象 80](#_Toc524553973)

[1.52 168\_Spark核心编程进阶-实验：在算子内打印数据的无法看到现象 81](#_Toc524553974)

[1.53 169\_Spark核心编程进阶-mapPartitions以及学生成绩查询案例 81](#_Toc524553975)

[1.54 170\_Spark核心编程进阶-mapPartitionsWithIndex以开学分班案例 82](#_Toc524553976)

[1.55 171\_Spark核心编程进阶-sample以及公司年会抽奖案例 84](#_Toc524553977)

[1.56 172\_Spark核心编程进阶-union以及公司部门合并案例 85](#_Toc524553978)

[1.57 173\_Spark核心编程进阶-intersection以及公司跨多项目人员查询案例 85](#_Toc524553979)

[1.58 174\_Spark核心编程进阶-distinct以及网站uv统计案例 86](#_Toc524553980)

[1.59 175\_Spark核心编程进阶-aggregateByKey以及单词计数案例 88](#_Toc524553981)

[1.60 176\_Spark核心编程进阶-cartesian以及服装搭配案例 90](#_Toc524553982)

[1.61 177\_Spark核心编程进阶-coalesce以及公司部门整合案例 91](#_Toc524553983)

[1.62 178\_Spark核心编程进阶-repartition以及公司新增部门案例 93](#_Toc524553984)

[1.63 179\_Spark核心编程进阶-takeSampled以及公司年会抽奖案例 95](#_Toc524553985)

[1.64 180\_Spark核心编程进阶-shuffle操作原理详解 96](#_Toc524553986)

[1.65 181\_Spark核心编程进阶-shuffle操作过程中进行数据排序 97](#_Toc524553987)

[1.66 182\_Spark核心编程进阶-会触发shuffle操作的算子 98](#_Toc524553988)

[1.67 183\_Spark核心编程进阶-shuffle操作对性能消耗的原理详解 99](#_Toc524553989)

[1.68 184\_Spark核心编程进阶-shuffle操作所有相关参数详解以及性能调优 100](#_Toc524553990)

[1.69 185\_Spark核心编程进阶-综合案例1：移动端app访问流量日志分析 102](#_Toc524553991)

[1.70 186\_Spark核心编程进阶-综合案例1：日志文件格式分析 103](#_Toc524553992)

[1.71 187\_Spark核心编程进阶-综合案例1：读取日志文件并创建RDD 103](#_Toc524553993)

[1.72 188\_Spark核心编程进阶-综合案例1：创建自定义的可序列化类 106](#_Toc524553994)

[1.73 189\_Spark核心编程进阶-综合案例1：将RDD映射为key-value格式 107](#_Toc524553995)

[1.74 190\_Spark核心编程进阶-综合案例1：基于deviceID进行聚合操作 108](#_Toc524553996)

[1.75 191\_Spark核心编程进阶-综合案例1：自定义二次排序key类 111](#_Toc524553997)

[1.76 192\_Spark核心编程进阶-综合案例1：将二次排序key映射为RDD的key 114](#_Toc524553998)

[1.77 193\_Spark核心编程进阶-综合案例1：执行二次排序以及获取top10数据 118](#_Toc524553999)

[1.78 194\_Spark核心编程进阶-综合案例1：程序运行测试以及代码调试 121](#_Toc524554000)

[1.79 195\_Spark核心编程进阶-部署第二台CentOS机器 121](#_Toc524554001)

[1.80 196\_Spark核心编程进阶-部署第二个Hadoop节点 123](#_Toc524554002)

[1.81 197\_Spark核心编程进阶-将第二个Hadoop节点动态加入集群 125](#_Toc524554003)

[1.82 198\_Spark核心编程进阶-使用yarn-client和yarn-cluster提交spark作业 126](#_Toc524554004)

[1.83 199\_Spark内核原理进阶-union算子内部实现原理剖析 126](#_Toc524554005)

[1.84 200\_Spark内核原理进阶-groupByKey算子内部实现原理剖析 126](#_Toc524554006)

[1.85 201\_Spark内核原理进阶-reduceByKey算子内部实现原理剖析 127](#_Toc524554007)

[1.86 202\_Spark内核原理进阶-distinct算子内部实现原理剖析 128](#_Toc524554008)

[1.87 203\_Spark内核原理进阶-cogroup算子内部实现原理剖析 128](#_Toc524554009)

[1.88 204\_Spark内核原理进阶-intersection算子内部实现原理剖析 129](#_Toc524554010)

[1.89 205\_Spark内核原理进阶-join算子内部实现原理剖析 130](#_Toc524554011)

[1.90 206\_Spark内核原理进阶-sortByKey算子内部实现原理剖析 130](#_Toc524554012)

[1.91 207\_Spark内核原理进阶-cartesian算子内部实现原理剖析 131](#_Toc524554013)

[1.92 208\_Spark内核原理进阶-coalesce算子内部实现原理剖析 132](#_Toc524554014)

[1.93 209\_Spark内核原理进阶-repartition算子内部实现原理剖析 133](#_Toc524554015)

[1.94 210\_SparkSQL实战开发进阶-Hive013安装与测试 134](#_Toc524554016)

[1.95 211\_SparkSQL实战开发进阶-ThriftJDBC、ODBCServer 134](#_Toc524554017)

[1.96 212\_SparkSQL实战开发进阶-CLI命令行使用 137](#_Toc524554018)

[1.97 213\_SparkSQL实战开发进阶-综合案例2：新闻网站关键指标离线统计 138](#_Toc524554019)

[1.98 214\_SparkSQL实战开发进阶-综合案例2：页面pv统计以及排序 142](#_Toc524554020)

[1.99 215\_SparkSQL实战开发进阶-综合案例2：页面uv统计以及排序 142](#_Toc524554021)

[1.100 216\_SparkSQL实战开发进阶-综合案例2：新用户注册比例统计 142](#_Toc524554022)

[1.101 217\_Spark SQL实战开发进阶-综合案例2：用户跳出率统计 142](#_Toc524554023)

[1.102 218\_SparkSQL实战开发进阶-综合案例2：版块热度排行榜统计 142](#_Toc524554024)

[1.103 219\_SparkSQL实战开发进阶-综合案例2：测试与调试 142](#_Toc524554025)

[1.104 220\_SparkStreaming实战开发进阶-flume安装 148](#_Toc524554026)

[1.105 221\_SparkStreaming实战开发进阶-接收flume实时数据流 156](#_Toc524554027)

[1.106 222\_SparkStreaming实战开发进阶-接收flume实时数据流 160](#_Toc524554028)

[1.107 223\_SparkStreaming实战开发进阶-高阶技术之自定义Receiver(结束) 163](#_Toc524554029)

[1.108 223\_SparkStreaming实战开发进阶-高阶技术之自定义Receiver **错误!未定义书签。**](#_Toc524554030)

[1.109 224\_SparkStreaming实战开发进阶-kafka安装 165](#_Toc524554031)

[1.110 225\_SparkStreaming实战开发进阶-综合案例3 167](#_Toc524554032)

[1.111 226\_SparkStreaming实战开发进阶-综合案例3 168](#_Toc524554033)

[1.112 227\_SparkStreaming实战开发进阶-综合案例3 174](#_Toc524554034)

[1.113 228\_SparkStreaming实战开发进阶-综合案例3：注册用户数实时统计 179](#_Toc524554035)

[1.114 229\_SparkStreaming实战开发进阶-综合案例3 179](#_Toc524554036)

[1.115 230\_SparkStreaming实战开发进阶-综合案例3 179](#_Toc524554037)

[1.116 231\_Spark运维管理进阶-基于ZooKeeper实现HA高可用性以及自动主备切换 187](#_Toc524554038)

[1.117 232\_Spark运维管理进阶-实验：基于ZooKeeper实现HA高可用性以及自动主备切换 189](#_Toc524554039)

[1.118 233\_Spark运维管理进阶-基于文件系统实现HA高可用性以及手动主备切换 191](#_Toc524554040)

[1.119 234\_Spark运维管理进阶-实验：基于文件系统实现HA高可用性以及手动主备切换 193](#_Toc524554041)

[1.120 235\_Spark运维管理进阶-作业监控-SparkWebUI以及监控实验 193](#_Toc524554042)

[1.121 236\_Spark运维管理进阶-作业监控 194](#_Toc524554043)

[1.122 237\_Spark运维管理进阶-作业监控 **错误!未定义书签。**](#_Toc524554044)

[1.123 238\_Spark运维管理进阶-作业监控 **错误!未定义书签。**](#_Toc524554045)

[1.124 239\_Spark运维管理进阶-作业监控 **错误!未定义书签。**](#_Toc524554046)

[1.125 240\_Spark运维管理进阶-作业资源调度 **错误!未定义书签。**](#_Toc524554047)

[1.126 241\_Spark运维管理进阶-作业资源调度 **错误!未定义书签。**](#_Toc524554048)

[1.127 242\_Spark运维管理进阶-作业资源调度 **错误!未定义书签。**](#_Toc524554049)

[1.128 243\_Spark运维管理进阶-作业资源调度 **错误!未定义书签。**](#_Toc524554050)

[1.129 244\_Spark运维管理进阶-作业资源调度 **错误!未定义书签。**](#_Toc524554051)

[1.130 245\_Spark运维管理进阶-作业资源调度 **错误!未定义书签。**](#_Toc524554052)

# 基础

## 113\_Scala编程进阶：Scaladoc的使用

## 114\_Scala编程进阶：跳出循环语句的3种方法

|  |  |  |  |
| --- | --- | --- | --- |
| while循环：  var flag = true  var res = 0  var n = 0  while(flag) {  res += n  n += 1  if (n == 5) {  flag = false  }  } | for循环：（高级for循环，加上了if守卫）  var flag = true  var res = 0  for (i <- 0 until 10 if flag) {  res += i  if (i == 4) flag = false  } | def add\_outer() = {  var res = 0  def add\_inner() {  for (i <- 0 until 10) {  if (i == 5) {  return  }  res += i  }  }  add\_inner()  res  } | 跟java里面的break比较类似，相对来说，比较灵活好用；与breakable代码块配合使用  import scala.util.control.Breaks.\_  var res = 0  breakable {  for (i <- 0 until 10) {  if (i == 5) {  break;  }  res += i  }  } |

## 115\_Scala编程进阶：多维数组、Java数组与Scala数组的隐式转换

|  |
| --- |
| Scala代码中，直接调用JDK（Java）的API，比如调用一个Java类的方法，势必可能会传入Java类型的list；Scala中构造出来的list，其实是ArrayBuffer；你直接把Scala的ArrayBuffer传入Java接收ArrayList的方法，肯定不行。  import scala.collection.JavaConversions.bufferAsJavaList  import scala.collection.mutable.ArrayBuffer  val command = ArrayBuffer("javac", "C:\\Users\\Administrator\\Desktop\\HelloWorld.java")  val processBuilder = new ProcessBuilder(command) // 构造函数是java.util.List  val process = processBuilder.start()  val res = process.waitFor()  import scala.collection.JavaConversions.asScalaBuffer  import scala.collection.mutable.Buffer  val cmd: Buffer[String] = processBuilder.command() |

## 116\_Scala编程进阶：Tuple拉链操作、JavaMap与ScalaMap的隐式转换

Tuple拉链操作指的就是zip操作

zip操作，是Array类的方法，用于将两个Array，合并为一个Array

比如Array(v1)和Array(v2)，使用zip操作合并后的格式为Array((v1,v2))

合并后的Array的元素类型为Tuple

val students = Array("Leo", "Jack", "Jen")

val scores = Array(80, 100, 90)

val studentScores = students.zip(scores)

for ((student, score) <- studentScores)

println(student + " " + score)

如果Array的元素类型是个Tuple，调用Array的toMap方法，可以将Array转换为Map

studentScores.toMap

**Java Map与Scala Map的隐式转换**

import scala.collection.JavaConversions.mapAsScalaMap

val javaScores = new java.util.HashMap[String, Int]()

javaScores.put("Alice", 10)

javaScores.put("Bob", 3)

javaScores.put("Cindy", 8)

val scalaScores: scala.collection.mutable.Map[String, Int] = javaScores

import scala.collection.JavaConversions.mapAsJavaMap

import java.awt.font.TextAttribute.\_

val scalaAttrMap = Map(FAMILY -> "Serif", SIZE -> 12)

val font = new java.awt.Font(scalaAttrMap)

## 117\_Scala编程进阶：扩大内部类作用域的2种方法、内部类获取外部类引用

**内部类的作用域：外部类对象**

import scala.collection.mutable.ArrayBuffer

class Class {

class Student(val name: String)

val students = new ArrayBuffer[Student]

def register(name: String) = {

new Student(name)

}

}

val c1 = new Class

val leo = c1.register("leo")

c1.students += leo

val c2 = new Class

val jack = c2.register("jack")

c1.students += jack

**扩大内部类作用域：伴生对象**

object Class {

class Student(val name: String)

}

class Class {

val students = new ArrayBuffer[Class.Student]

def register(name: String) = {

new Class.Student(name)

}

}

val c1 = new Class

val leo = c1.register("leo")

c1.students += leo

val c2 = new Class

val jack = c2.register("jack")

c1.students += jack

**扩大内部类作用域：类型投影**

class Class {

class Student(val name: String)

val students = new ArrayBuffer[Class#Student]

def register(name: String) = {

new Student(name)

}

}

val c1 = new Class

val leo = c1.register("leo")

c1.students += leo

val c2 = new Class

val jack = c2.register("jack")

c1.students += jack

**内部类获取外部类的引用**

class Class(val name: String) { outer =>

class Student(val name: String) {

def introduceMyself = "Hello, I'm " + name + ", I'm very happy to join class " + outer.name

}

def register(name: String) = {

new Student(name)

}

}

val c1 = new Class("c1")

val leo = c1.register("leo")

leo.introduceMyself

## 118\_Scala编程进阶：package与import实战详解

为什么要有package的概念？

因为要对多个同名的类进行命名空间的管理，避免同名类发生冲突

比如说，scala.collection.mutable.Map和scala.collection.immutable.Map

package定义的第一种方式: 多层级package定义（比较差的做法，一般不这么干）

package com {

package ibeifeng {

package scala {

class Test {}

}

}

}

package定义的第二种方式: 串联式package定义（也不怎么样，一般也不这么干）

package com.ibeifeng.scala {

package service {

class Test {}

}

}

package定义的第三种方式: 文件顶部package定义

package com.ibeifeng.scala.service

class Test {

}

package定义的第四种方式: IDE自动生成包

package特性一: 同一个包定义，可以在不同的scala源文件中的; 一个scala源文件内，可以包含两个包

Test1.scala

package com {

package ibeifeng {

package scala {

class Test1

}

}

}

Test2.scala

package com {

package ibeifeng {

package scala {

class Test2

}

}

}

Test3.scala

package com {

package ibeifeng {

package scala1 {

class Test

}

}

}

package com {

package ibeifeng {

package scala2 {

class Test

}

}

}

package特性二: 子包中的类，可以访问父包中的类

Test.scala

package com {

package ibeifeng {

package scala {

object Utils {

def isNotEmpty(str: String): Boolean = str != null && str != ""

}

class Test

package service {

class MyService {

def sayHello(name: String) {

if(Utils.isNotEmpty(name)) {

println("Hello, " + name)

} else {

println("Who are you?")

}

}

}

}

}

}

}

object MainClass {

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

val service = new com.ibeifeng.scala.service.MyService

service.sayHello("leo")

service.sayHello("")

}

}

package特性三: 相对包名与绝对包名

package com {

package ibeifeng {

package scala {

object Utils {

def isNotEmpty(str: String): Boolean = str != null && str != ""

}

class Test

package collection {}

package service {

class MyService {

// 这会报错，默认使用相对报名，从com.ibeifeng.scala.collection包中，寻找mutable包下的ArrayBuffer类

// 但是找不到，所以会报错

// val names = new scala.collection.mutable.ArrayBuffer[String]

// 正确的做法是使用\_root\_，引用绝对包名

val names = new \_root\_.scala.collection.mutable.ArrayBuffer[String]

def sayHello(name: String) {

if(Utils.isNotEmpty(name)) {

println("Hello, " + name)

} else {

println("Who are you?")

}

}

}

}

}

}

}

package特性四: 定义package对象（比较少）

package内的成员，可以直接访问package对象内的成员

package com.ibeifeng.scala

package object service {

val defaultName = "Somebody"

}

package service {

class MyService {

def sayHello(name: String) {

if(name != null && name != "") {

println("Hello, " + name)

} else {

println("Hello, " + defaultName)

}

}

}

}

package特性五: package可见性

package com.ibeifeng.scala

class Person {

private[scala] val name = "leo"

private[ibeifeng] val age = 25

}

import

如果没有import，那么你每次创建某个包下的类的对象，都得用new com.ibeifeng.scala.service.MyService这种冗长的格式。。。

所以如果用了import，那么。。。你只要先import com.ibeifeng.scala.service.MyService，然后再new MyService，即可。。。

import com.ibeifeng.scala.service.MyService;

object MainClass {

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

val service = new MyService

service.sayHello("leo")

service.sayHello("")

}

}

import特性一: 用import com.ibeifeng.scala.service.\_这种格式，可以导入包下所有的成员

import特性二: scala与java不同之处在于，任何地方都可以使用import，比如类内、方法内，这种方式的好处在于，可以在一定作用域范围内使用导入

object MainClass {

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

import com.ibeifeng.scala.service.\_

val service = new MyService

service.sayHello("leo")

service.sayHello("")

}

}

import特性三: 选择器、重命名、隐藏

import com.ibeifeng.scala.service.{ MyService }，仅仅导入java.awt包下的Color和Font类

import com.ibeifeng.scala.service.{ MyService => MyServiceImpl }，将导入的类进行重命名

import com.ibeifeng.scala.service.{ MyService => \_, \_ }，导入java.util包下所有的类，但是隐藏掉HashMap类，对导入的包别名为下划线，那么相当于隐藏这个包

import特性四: 隐式导入

每个scala程序默认都会隐式导入以下几个包下所有的成员

import java.lang.\_

import scala.\_

import Predef.\_

## 119\_Scala编程进阶：重写field的提前定义、Scala继承层级、对象相等性

重写field的提前定义

默认情况下，如果父类中的构造函数代码，用到了会被子类重写的field; 那么出出现令人意想不到的一幕:

1、子类的构造函数（无参）调用父类的构造函数（无参）

2、父类的构造函数初始化field（结果正确）

3、父类的构造函数使用field执行其他构造代码，但是此时其他构造代码如果使用了该field，而且field要被子类重写，那么它的getter方法被重写，返回0（比如Int）

4、子类的构造函数再执行，重写field（结果也正确）

5、但是此时子类从父类继承的其他构造代码，已经出现了错误了

class Student {

val classNumber: Int = 10

val classScores: Array[Int] = new Array[Int](classNumber)

}

class PEStudent {

override val classNumber: Int = 3

}

本来我们期望的是，PEStudent，可以从Student继承来一个长度为3的classScores数组

结果。。。PEStudent对象，只有一个长度为0的classScores数组

此时只能使用Scala对象继承的一个高级特性: 提前定义，在父类构造函数执行之前，先执行子类的构造函数中的某些代码

class PEStudent extends student {

override val classNumber: Int = 3

} with Student

Scala的继承层级

这里我们大概知道一下Scala的继承层级，我们写的所有的Scala trait和class，都是默认继承自一些Scala根类的，有一些基础的方法

Scala中，最顶端的两个trait是Nothing和Null，Null trait唯一的对象就是null

其次是继承了Nothing trait的Any类

接着Anyval trait和AnyRef类，都继承自Any类

Any类是个比较重要的类，其中定义了isInstanceOf和asInstanceOf等方法，以及equals、hashCode等对象的基本方法

Any类，有点像Java中的Object基类

AnyRef类，增加了一些多线程的方法，比如wait、notify/notifyAll、synchronized等，也是属于Java Object类的一部分

**对象相等性**

这里，我们要知道，在scala中，你如何判断两个引用变量，是否指向同一个对象实例

AnyRef的eq方法用于检查两个变量是否指向同一个对象实例

AnyRef的equals方法默认调用eq方法实现，也就是说，默认情况下，判断两个变量相等，要求必须指向同一个对象实例

通常情况下，自己可以重写equals方法，根据类的fields来判定是否相等

此外，定义equals方法时，也最好使用同样的fields，重写hashCode方法

如果只是想要简单地通过是否指向同一个对象实例，判定变量是否相当，那么直接使用==操作符即可，默认判断null，然后调用equals方法

class Product(val name: String, val price: Double) {

final override def equals(other: Any) = {

val that = other.asInstanceOf[Product]

if(that == null) false

else name == that.name && price == that.price

}

final override def hashCode = 13 \* name.hashCode + 17 \* price.hashCode

}

## 120\_Scala编程进阶：文件操作实战详解

遍历一个文件中的每一行

必须导入scala.io.Source类: import scala.io.Source

方法一: 使用Source.getLines返回的迭代器

val source = Source.fromFile("C://Users//Administrator//Desktop//test.txt", "UTF-8")

val lineIterator = source.getLines

for (line <- lineIterator) println(line)

方法二: 将Source.getLines返回的迭代器，转换成数组

这里说明一点: 一个BufferedSource对象的getLines方法，只能调用一次，一次调用完之后，遍历了迭代器里所有的内容，就已经把文件里的内容读取完了

如果反复调用source.getLines，是获取不到内容的

此时，必须重新创建一个BufferedSource对象

val source = Source.fromFile("C://Users//Administrator//Desktop//test.txt", "UTF-8")

val lines = source.getLines.toArray

for(line <- lines) println(line)

方法三: 调用Source.mkString，返回文本中所有的内容

val source = Source.fromFile("C://Users//Administrator//Desktop//test.txt", "UTF-8")

val lines = source.mkString

使用完BufferedSource对象之后，调用BufferedSource.close方法，关闭IO流资源

遍历一个文件中的每一个字符

BufferedSource，也实现了一个Iterator[Char]的这么一个trait

val source = Source.fromFile("C://Users//Administrator//Desktop//test.txt", "UTF-8")

for(c <- source) print(c)

从URL以及字符串中读取字符

val source = Source.fromURL("http://www.baidu.com", "UTF-8")

val source = Source.fromString("Hello World")

结合Java IO流，读取任意文件

如果你真的深入读了spark的源代码

真的对scala掌握的很深入，你就会知道一点

spark的源码实际上是由scala和java共同编写而成的，Java的多线程

scala，本身的编程语言的功能，就不是特别的强大和完善，比如说，scala甚至不能很方便地写文件，必须依赖于java的io流才可以

所以说，scala，其实主要就是针对某些特定领域的一些复杂系统的，比较适用的一种编程语言而已

完全无法替代java的，scala和java是相辅相成，荣辱与共的这么一种，共生关系

可以这么跟大家说

scala还有一种作用，可以用scala，编写spark的作业

但是问题是，为什么，我们要用java开发hive udf、mapreduce、hbase client、zookeeper client，用Java开发storm的作业

然后作为一个大数据工程师，偏偏用到spark的时候，一定要用scala开发呢？

用spark开发作业，用java，个人认为，个人观点，是最合适的，最通用的，最可移植的，最方便维护的

scala，这套课程里，scala编程详解、scala编程进阶

1、有些公司的技术leader，要求用scala开发spark作业，我也没办法，我是极力反对的; 保证学员，学了这套课程以后，可以用scala去工作和面试

2、有些同学，可能压根儿不会java; 大多数是上学的时候，主要是搞算法的，或者只会c++，只会python; 这套课程学了，不用会java，那么也可以精通和使用spark

3、最重要的一点，深入掌握scala所有的初中高级语法，才能透彻和深入的理解和阅读spark的源码

4、也有，但是很少，就是有些公司，可能会用scala，开发复杂的大型分布式后端系统

案例: 结合java IO流，做一个文件拷贝的案例

import java.io.\_

val fis = new FileInputStream(new File("C://Users//Administrator//Desktop//test.txt"))

val fos = new FileOutputStream(new File("C://Users//Administrator//Desktop//test3.txt"))

val buf = new Array[Byte](1024)

fis.read(buf)

fos.write(buf, 0, 1024)

fis.close()

fos.close()

结合Java IO流，写文件

val pw = new PrintWriter("C://Users//Administrator//Desktop//test4.txt")

pw.println("Hello World")

pw.close()

-------------------------------------------------------------------------------------------------------

递归遍历子目录

def getSubdirIterator(dir: File): Iterator[File] = {

val childDirs = dir.listFiles.filter(\_.isDirectory)

childDirs.toIterator ++ childDirs.toIterator.flatMap(getSubdirIterator \_)

}

val iterator = getSubdirIterator(new File("C://Users//Administrator//Desktop"))

for(d <- iterator) println(d)

-------------------------------------------------------------------------------------------------------

序列化以及反序列化（Java序列化和反序列化机制）

如果要序列化，那么就必须让类，有一个@SerialVersionUID，定义一个版本号

要让类继承一个Serializable trait

@SerialVersionUID(42L) class Person(val name: String) extends Serializable

val leo = new Person("leo")

import java.io.\_

val oos =

new ObjectOutputStream(new FileOutputStream("C://Users//Administrator//Desktop//test.obj"))

oos.writeObject(leo)

oos.close()

val ois =

new ObjectInputStream(new FileInputStream("C://Users//Administrator//Desktop//test.obj"))

val restoredLeo = ois.readObject().asInstanceOf[Person]

restoredLeo.name

## 121\_Scala编程进阶：偏函数实战详解

偏函数，是一种高级的函数形式

简单来说，偏函数是什么，**其实就是没有定义好明确的输入参数的函数**，函数体就是一连串的case语句

一般的函数

def getStudentGrade(name: String) = {

...

}

偏函数是PartialFunction[A, B]类的一个实例

这个类有两个方法，一个是apply()方法，直接调用可以通过函数体内的case进行匹配，返回结果;

另一个是isDefinedAt()方法，可以返回一个输入，是否跟任何一个case语句匹配

学生成绩查询案例

val getStudentGrade: PartialFunction[String, Int] = {

case "Leo" => 90; case "Jack" => 85; case "Marry" => 95

}

getStudentGrade("Leo")

getStudentGrade.isDefinedAt("Tom")

## 122\_Scala编程进阶：执行外部命令

scala执行外部命令

咱们的scala程序，实际上，写好以后，跑起来，关键是，跑在哪里？

scala程序是运行在java虚拟机中的，也就是咱们平时常说的jvm，所以我们之前能够看到，scala可以直接调用jdk

jdk: java development kit，java基础的开发api

scala的程序，是运行在一个进程中的

运行在什么进程中？是运行在jvm虚拟机进程中的

比如说，如果说，我们的scala程序，希望去执行scala所在进程之外的，比如说，本地操作系统的一个命令

也许执行的本地操作系统的命令，会启动一个新的进程，也许也不会

但是，如果想要实现这样的效果和功能，scala能不能够做到？

这个是可以的

scala实际上，是提供了这样的支持的

也就是说，咱们的scala程序，运行在一个独立的进程中，但是可以随心所欲地执行外部操作系统的其他命令

甚至是说，启动其他的进程

案例: 使用scala编译和执行外部的java程序

import sys.process.\_

"javac HelloWorld.java" !

"java HelloWorld" !

## 123\_Scala编程进阶：正则表达式支持

scala的正则表达式支持

正则表达式是什么？

一种语法，用一个表达式，来匹配一系列的字符串

[a-z]+: 一个或多个a~z范围的26个小写英文字母，比如hello，world

强调一点，咱们不会去给大家详细介绍正则表达式的语法

正则表达式这个东西，语法比较多

有兴趣，有需要的话，那么建议直接百度一下，自己看一下介绍，就知道是怎么回事了

大致学习一下正则表达式的语法即可

我们这里呢，讲解一下scala中，对这个正则表达式的支持

---------------------------------------------------------------------------------

定义一个正则表达式，使用String类的r方法

此时返回的类型是scala.util.matching.Regex类的对象

val pattern1 = "[a-z]+".r

拿到一个正则表达式以后，我们一般会用它来做什么事情？

比如，我们会用正则表达式来匹配一些字符串，比如来看看，某个字符串是否符合表达式规定的范围之内

比如，从一个长的字符串中，提取出来，匹配正则表达式的各个部分

val str = "hello 123 world 456"

获取一个字符串中，匹配正则表达式的部分，使用findAllIn，会获取到一个Iterator，迭代器

然后就可以去遍历各个匹配正则的部分，去进行处理

for (matchString <- pattern1.findAllIn(str)) println(matchString)

同理，使用findFirstIn，可以获取第一个匹配正则表达式的部分

pattern1.findFirstIn(str)

使用replaceAllIn，可以将匹配正则的部分，替换掉

pattern1.replaceFirstIn("hello world", "replacement")

使用replaceFirstIn，可以将第一个匹配正则的部分，替换掉

pattern1.replaceAllIn("hello world", "replacement")

## 124\_Scala编程进阶：提取器实战详解

提取器

apply方法

伴生类和伴生对象的概念，companion class和companion object

伴生对象里面，可以定义一个apply方法

直接调用类(参数)，方式，就相当于在调用apply方法

此时在apply方法中，通常来说（也不一定），会创建一个伴生类的对象，返回回去

这种方式，有一个好处，创建对象呢，非常的方便

不要每次都是new 类(参数)，类(参数)

提取器

unapply方法

和apply方法，顾名思义，那就是反过来

apply方法，可以理解为，接收一堆参数，然后返回一个对象

unapply方法，可以理解为，接收一个字符串，解析成一个对象的各个字段

提取器就是一个包含了unapply方法的对象，跟apply方法正好相反

apply方法，是接收一堆参数，然后构造出来一个对象

unapply方法，是接收一个字符串，然后解析出对象的属性值

class Person(val name: String, val age: Int)

object Person {

def apply(name: String, age: Int) = new Person(name, age)

def unapply(str: String) = {

val splitIndex = str.indexOf(" ")

if (splitIndex == -1) None

else Some((str.substring(0, splitIndex), str.substring(splitIndex + 1)))

}

}

val Person(name, age) = "leo 25"

name

age

## 125\_Scala编程进阶：样例类的提取器实战详解

样例类的提取器

scala中的样例类，说白了，也很简单

类似于java中的javabean，java中的JavaBean，是什么东东？

包含了一堆属性，field; 每个field都有一对getter和setter方法

public class Person {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

scala中的样例类，默认就是提供apply方法和unapply方法的

case class Person(name: String, age: Int)

val p = Person("leo", 25)

p match {

case Person(name, age) => println(name + ": " + age)

}

## 126\_Scala编程进阶：只有一个参数的提取器

只有一个参数的提取器

之前，已经跟大家讲解过普通的提取器

相当于是，比如说，接收一个字符串，作为参数

然后从字符串里面解析出来多个字段值，然后将多个字段值封装在一个tuple中

作为Some类型的对象，返回

现在我们来想一下，如果你的类只有一个字段

字符串里面只有一个字段

解析出来的一个字段，是没有办法放在tuple中的，因为scala中的tuple，规定了，必须要两个以及两个以上的值

这个时候，在提取器，unapply方法中，只能将一个字段值，封装在Some对象中，直接返回

class Person(val name: String)

object Person {

def unapply(input: String): Option[String] = Some(input)

}

val Person(name) = "leo"

## 127\_Scala编程进阶：注解实战详解

什么是注解？

注解其实说白了，就是在我们的代码中，加入一些特殊的标记

特殊的标记大概长什么样子呢？

我们之前学过一个很常用，和很经典的一个注解，其实就是@BeanProperty，让编译器自动生成属性的JavaBean风格的getter和setter方法

除此之外，还在《文件操作实战详解》那一讲，讲过一个序列化的这个东西，@SerialUID（可能是错误的），指定一个序列化的版本号

**注解是用来干嘛的？**

然后我们的scala编译器，就可以在编译的时候，在碰到注解的时候，做一些特殊的操作。一个非常经典的例子就是

@BeanProperty注解，我们之前讲解过，给某个field添加了这个注解之后，scala编译器就会给field编译出新的JavaBean风格的getter和setter方法

------------------------------------------------------------------------

scala中，在哪些地方可以添加注解呢？

scala中，可以给类、方法、field、local variable、constructor / method / function parameter添加注解

而且scala是支持给某个目标添加多个注解的

这里有一些特例：如果要给类的主构造函数添加注解，那么需要在构造函数前添加注解，并加上一对圆括号

比如说

class Person @Unchecked() (val name: String, val age: Int)

还可以给表达式添加注解，此时需要在表达式后面加上冒号以及注解，比如

val scores = Map("Leo" -> 90, "Jack" -> 60)

(scores.get("Leo"): @unchecked) match { case score => println(score) }

除此之外，还可以给类型参数和变量的类型定义添加注解

------------------------------------------------------------------------

Scala中开发注解

要自己动手开发一个注解，就必须扩展Annotation trait，比如

class Test extends annotation.Annotation

@Test

class myTest

------------------------------------------------------------------------

注解的参数

注解中，是可以有参数的，比如

class Test(var timeout: Int) extends annotation.Annotation

@Test(timeout = 100) class myTest

如果注解的参数是value的话，那么也可以不用指定注解的参数名，比如

class Test(var value: String) extends annotation.Annotation

## 128\_Scala编程进阶：常用注解介绍

dscala提供的常用注解

其实里面很多都是针对java中的概念和概念提供的

跟大家提示一下，再次证明了，你搞scala，真想搞好的话，先学和精通java

scala和java的关系是唇齿相依的

scala依赖java

java并不依赖scala

scala中，常用的一些注解，全部是针对java的一些概念

所以呢，在这里我们没办法给大家详细讲解java的概念，就直接介绍scala中针对java的一些注解

@volatile var name = "leo" 轻量级的java多线程并发安全控制

jvm，java虚拟机中，可以有多个线程

每个线程都有自己的工作区，还有一块儿所有线程共享的工作区

每次一个线程拿到一个公共的变量，都需要从共享区中拷贝一个副本到自己的工作区中使用，和修改

然后修改完以后，再在一个合适的时机，将副本的值，写回到共享区中

这里就会出现一个多线程并发访问安全的问题

多个线程如果同时拷贝了变量副本，都做了不同的修改

然后依次将副本修改的值，写回到共享区中，会依次覆盖掉之前的一些副本值

就会出现变量的值，是不符合预期的

咱们的系统，出现了错误和bug

volatile关键字修饰的变量

它可以保证，一个线程在从共享区获取一个变量的副本时，都会强制刷新一下这个变量的值

保证自己获取到的变量的副本值是最新的

所以这样子做呢，是一种轻量级的多线程并发访问控制办法

但是也不是百分之百保险的，还是有可能会出现错误的风险

@transient var name = "leo" 瞬态字段，不会序列化这个字段

之前讲序列化，默认会将一个对象中所有的字段的值，都序列化到磁盘文件中去

然后反序列化的时候，还可以获取这些字段的值

加了transient的字段，是瞬态的，序列化的时候，不会序列化这个字段

反序列化的时候，这个字段也就没有值了

@SerialVersionUID(value) 标记类的序列化版本号

序列化版本号，这个什么意思

如果我们将一个类的对象序列化到磁盘文件上了

结果过了一段时间以后，这个类在代码中改变了，此时如果你想将磁盘文件中的对象反序列化回来

就会报错，因为你的序列化的对象的结构与代码中的类结构已经不一样了

针对这种问题，就应该有一个序列化版本号

如果你的类改变了，就重新生成一个序列化版本号

反序列化的时候，就会发现序列化类型的版本号和代码中的类的版本号，不一样

@native 标注用c实现的本地方法

@throws(classOf[Exception]) def test() {} 给方法标记要抛出的checked异常

@varargs def test(args: String\*) {} 标记方法接收的是变长参数

@BeanProperty 标记生成JavaBean风格的getter和setter方法

@BooleanBeanProperty 标记生成is风格的getter方法，用于boolean类型的field

@deprecated(message = "") 让编译器提示警告

@unchecked 让编译器提示类型转换的警告

## 129\_Scala编程进阶：XML基础操作实战详解

scala中定义xml

scala对xml有很好的支持，可以直接在scala代码中定义一个xml文档元素

val books = <books><book>my first scala book</book></books>

此时doc的类型是scala.xml.Elem，也就是一个xml元素

scala还可以直接定义多个同级别的xml元素

val books = <book>my first scala book</book><book>my first spark book</book>

此时doc的类型是scala.xml.NodeBuffer，也就是一个xml节点序列

---------------------------------------------------------------------------------

XML节点类型

Node类是所有XML节点类型的父类型，两个重要的子类型是Text和Elem。

Elem表示一个XML元素，也就是一个XML节点。scala.xml.Elem类型的label属性，返回的是标签名，child属性，返回的是子元素。

scala.xml.NodeSeq类型，是一个元素序列，可以用for循环，直接遍历它。

可以通过scala.xml.NodeBuffer类型，来手动创建一个节点序列

val booksBuffer = new scala.xml.NodeBuffer

booksBuffer += <book>book1</book>

booksBuffer += <book>book2</book>

val books: scala.xml.NodeSeq = booksBuffer

---------------------------------------------------------------------------------

xml元素的属性

scala.xml.Elem.attributes属性，可以返回这儿xml元素的属性，是Seq[scala.xml.Node]类型的，继续调用text属性，可以拿到属性的值

val book = <book id=“1” price=“10.0”>book1</book>

val bookId = book.attributes(“id”).text

还可以遍历属性

for(attr <- book.attributes) println(attr)

还可以调用book.attributes.asAttrMap，获取一个属性Map

## 130\_Scala编程进阶：XML中嵌入scala代码

在xml中嵌入scala代码

val books = Array("book1", "book2")

<books><book>{ books(0) }</book><book>{ books(1) }</book></books>

<books>{ for (book <- books) yield <book>{book}</book> }</books>

还可以在xml属性中嵌入scala代码

<book id={ books(0) }>{ books(0) }</book>

## 131\_Scala编程进阶：XML修改元素实战详解

修改xml元素

默认情况下，scala中的xml表达式是不可改变的；如果要修改xml元素的话，必须拷贝一份再修改

val books = <books><book>book1</book></books>

添加一个子元素

val booksCopy = books.copy(child = books.child ++ <book>book2</book>)

val book = <book id="1">book1</book>

import scala.xml.\_

修改一个属性

val bookCopy = book % Attribute(null, "id", "2", Null)

添加一个属性

val bookCopy = book % Attribute(null, "id", "2", Attribute(null, "price", "10.0", Null))

说点闲话

如果大家真的对java比较精通的话

然后过来学习这个scala，就会发现有个特点

java的功能是非常强大的

但是，从各个方面来看，比如io、xml操作、第三方类库的支持、socket、gui界面编程、jdbc访问数据库等等，scala都比java差很多

之所以现在scala有点火，有些人推崇这个scala

其实主要是因为spark是用scala作为主要的语言开发的（但是spark底层的源码，其实都是java）

类加载器、线程、反射、线程池等等这些东西，全部都是java底层，外部命令的执行（ProcessBuilder）

## 132\_Scala编程进阶：XML加载和写入外部文档

加载和写入外部xml文件

import scala.xml.\_

import java.io.\_

使用scala的XML类加载

val books = XML.loadFile("C://Users//Administrator//Desktop//books.xml")

使用Java的FileInputStream类加载

val books = XML.load(new FileInputStream("C://Users//Administrator//Desktop//books.xml"))

使用Java的InputStreamReader类指定加载编码

val books = XML.load(new InputStreamReader(

new FileInputStream("C://Users//Administrator//Desktop//books.xml"), "UTF-8"))

将内存中的xml对象，写入外部xml文档

XML.save("C://Users//Administrator//Desktop//books2.xml", books)

## 133\_Scala编程进阶：集合元素操作

集合元素操作

col :+ ele 将元素添加到集合尾部 Seq

ele +: col 将元素添加到集合头部 Seq

col + ele 在集合尾部添加元素 Set、Map

col + (ele1, ele2) 将其他集合添加到集合的尾部 Set、Map

col - ele 将元素从集合中删除 Set、Map、ArrayBuffer

col - (ele1, ele2) 将子集合从集合中删除 Set、Map、ArrayBuffer

col1 ++ col2 将其他集合添加到集合尾部 Iterable

col2 ++: col1 将其他集合添加到集合头部 Iterable

ele :: list 将元素添加到list的头部 List

list2 ::: list1 将其他list添加到list的头部 List

list1 ::: list2 将其他list添加到list的尾部 List

set1 | set2 取两个set的并集 Set

set1 & set2 取两个set的交集 Set

set1 &~ set2 取两个set的diff Set

col += ele 给集合添加一个元素 可变集合

col += (ele1, ele2) 给集合添加一个集合 可变集合

col ++= col2 给集合添加一个集合 可变集合

col -= ele 从集合中删除一个元素 可变集合

col -= (ele1, ele2) 从集合中删除一个子集合 可变集合

col —= col2 从集合中删除一个子集合 可变集合

ele +=: col 向集合头部添加一个元素 ArrayBuffer

col2 ++=: col 向集合头部添加一个集合 ArrayBuffer

## 134\_Scala编程进阶：集合的常用操作方法

集合的常用方法

head、last、tail

length、isEmpty

sum、max、min

count、exists、filter、filterNot

takeWhile、dropWhile

take、drop、splitAt

takeRight、dropRight

sclie

contains、startsWith、endsWith

indexOf

intersect、diff

## 135\_Scala编程进阶：map、flatMap、collect、foreach实战详解

scala集合map操作实战详解

map操作，一对一映射

val scoreMap = Map("leo" -> 90, "jack" -> 60, "tom" -> 70)

val names = List("leo", "jack", "tom")

names.map(scoreMap(\_))

flatMap操作，一对多映射

val scoreMap = Map("leo" -> List(80, 90, 60), "jack" -> List(70, 90, 50), "tom" -> List(60,70,40))

names.map(scoreMap(\_))

names.flatMap(scoreMap(\_))

collect操作，结合偏函数使用

"abc".collect { case 'a' => 1; case 'b' => 2; case 'c' => 3 }

foreach操作，遍历

names.foreach(println \_)

## 136\_Scala编程进阶：reduce和fold实战详解1

reduce操作

List(1, 2, 3, 4).reduceLeft(\_ - \_)

List(1, 2, 3, 4).reduceRight(\_ - \_)

fold操作

List(1, 2, 3, 4).foldLeft(10)(\_ - \_)

List(1, 2, 3, 4).foldRight(10)(\_ - \_)

## 138\_环境搭建-Hadoop25伪分布式集群搭建

**修改core-site.xml**

|  |
| --- |
| <property>  <name>fs.default.name</name>  <value>hdfs://sparkupgrade1:9000</value>  </property> |

**修改hdfs-site.xml**

|  |
| --- |
| <property>  <name>dfs.name.dir</name>  <value>/usr/local/data/namenode</value>  </property>  <property>  <name>dfs.data.dir</name>  <value>/usr/local/data/datanode</value>  </property>  <property>  <name>dfs.tmp.dir</name>  <value>/usr/local/data/tmp</value>  </property>  <property>  <name>dfs.replication</name>  <value>1</value>  </property> |

**修改mapred-site.xml**

|  |
| --- |
| <property>  <name>mapreduce.framework.name</name>  <value>yarn</value>  </property> |

**修改yarn-site.xml**

|  |
| --- |
| **<property>**  **<name>yarn.resourcemanager.hostname</name>**  **<value>sparkupgrade1</value>**  **</property>**  **<property>**  **<name>yarn.nodemanager.aux-services</name>**  **<value>mapreduce\_shuffle</value>**  **</property>** |

**启动hdfs集群**

|  |
| --- |
| **1、格式化namenode：hdfs namenode -format**  **2、启动hdfs集群：start-dfs.sh 3、验证启动是否成功：**  **jps：namenode、datanode、secondarynamenode**  **50070端口**  **4、读写操作测试**  **hdfs dfs -mkdir /test**  **hdfs dfs -put hello.txt /test/hello.txt**  **hdfs dfs -text /test/hello.txt** |

**启动yarn集群**

|  |
| --- |
| **1、启动yarn集群：start-yarn.sh 2、验证启动是否成功：**  **jps：resourcemanager、nodemanager**  **8088端口** |

## 139\_环境搭建-Spark15伪分布式集群搭建

**安装scala 2.11.4**

|  |
| --- |
| 1、将课程提供的scala-2.11.4.tgz使用WinSCP拷贝到/usr/local/src目录下。  2、对scala-2.11.4.tgz进行解压缩：tar -zxvf scala-2.11.4.tgz  3、对scala目录进行重命名：mv scala-2.11.4 scala  4、配置scala相关的环境变量  vi ~/.bashrc  export SCALA\_HOME=/usr/local/scala  export PATH=$SCALA\_HOME/bin  source ~/.bashrc  5、查看scala是否安装成功：scala -version |

**安装spark客户端**

|  |
| --- |
| 1、将spark-1.5.1-bin-hadoop2.4.tgz使用WinSCP上传到/usr/local/src目录下。  2、解压缩spark包：tar -zxvf spark-1.5.1-bin-hadoop2.4.tgz。  3、重命名spark目录：mv spark-1.5.1-bin-hadoop2.4 spark  4、修改spark环境变量  vi ~/.bashrc  export SPARK\_HOME=/usr/local/spark  export PATH=$SPARK\_HOME/bin  export CLASSPATH=.:$CLASSPATH:$JAVA\_HOME/lib:$JAVA\_HOME/jre/lib  source ~/.bashrc |

**修改spark-env.sh文件**

|  |
| --- |
| 1、cd /usr/local/spark/conf  2、cp spark-env.sh.template spark-env.sh  3、vi spark-env.sh  export JAVA\_HOME=/usr/java/latest  export SCALA\_HOME=/usr/local/scala  export HADOOP\_HOME=/usr/local/hadoop  export HADOOP\_CONF\_DIR=/usr/local/hadoop/etc/hadoop  export SPARK\_MASTER\_IP=192.168.1.107  export SPARK\_WORKER\_MEMORY=1g |

**启动spark集群**

|  |
| --- |
| 1、在spark目录下的sbin目录执行./start-all.sh  2、使用jsp和8080端口可以检查集群是否启动成功  3、进入spark-shell查看是否正常  4、编写一个简单的wordcount程序，检验一下spark是否正常工作  5、通过4040端口，看一下spark web ui界面 |

## 140\_第一次课程升级大纲介绍以及要点说明

## 141\_Spark核心编程进阶-Spark集群架构概览

## 142\_Spark核心编程进阶-Spark集群架构的几点特别说明

关于spark集群架构的一些说明

1、每个spark application，都有属于自己的executor进程；绝对不可能出现多个spark application共享一个executor进程的

executor进程，在整个spark application运行的生命周期内，都会一直存在，不会自己消失的

executor进程，最主要的，就是使用多线程的方式，运行SparkContext分配过来的task，来一批task就执行一批，一批执行完了，再换下一批task执行

2、spark application，跟集群管理器之间，是透明的

不管你是哪个集群管理器，我就知道，我找你就可以申请到executor进程就好了

所以说，就算在一个能够容纳其他类型的计算作业的集群管理器中，也可以提交spark作业，比如说YARN、Mesos这种

大公司里，其实一般都是用YARN作为大数据计算作业管理器的，包括mapreduce、hive、storm和spark，统一在yarn上面运行，统一调度和管理公司的系统资源

3、driver（其实也就是咱们的main类运行的jvm进程），必须时刻监听着属于它这个spark application的executor进程发来的通信和连接

而且driver除了监听，自己也得负责调度整个spark作业（你自己写的spark代码）的调度和运行，也得大量跟executor进程通信，给他们分派计算任务

所以driver在网络环境中的位置，还是很重要的，driver尽量离spark集群得近一些

4、上面说了，driver要调度task给executor执行，所以driver最好和spark集群在一片网络内

## 143\_Spark核心编程进阶-Spark的核心术语讲解

一些重要的spark术语

Application spark应用程序，说白了，就是用户基于spark api开发的程序，一定是通过一个有main方法的类执行的，比如java开发spark，就是在eclipse中，建立的一个工程

Application Jar 这个就是把写好的spark工程，打包成一个jar包，其中包括了所有的第三方jar依赖包，比如java中，就用maven+assembly插件打包最方便

Driver Program 说白了，就是运行程序中main方法的进程，这就是driver，也叫driver进程

Cluster Manager 集群管理器，就是为每个spark application，在集群中调度和分配资源的组件，比如Spark Standalone、YARN、Mesos等

Deploy Mode 部署模式，无论是基于哪种集群管理器，spark作业部署或者运行模式，都分为两种，client和cluster，client模式下driver运行在提交spark作业的机器上；cluster模式下，运行在spark集群中

Worker Node 集群中的工作节点，能够运行executor进程，运行作业代码的节点

Executor 集群管理器为application分配的进程，运行在worker节点上，负责执行作业的任务，并将数据保存在内存或磁盘中，每个application都有自己的executor

Job 每个spark application，根据你执行了多少次action操作，就会有多少个job

Stage 每个job都会划分为多个stage（阶段），每个stage都会有对应的一批task，分配到executor上去执行

Task driver发送到executor上执行的计算单元，每个task负责在一个阶段（stage），处理一小片数据，计算出对应的结果

## 145\_Spark核心编程进阶-单独启动master和worker脚本详解

standalone模式启动集群命令详解

我们之前搭建那个伪分布式spark standalone集群的时候，以及我们最早搭建分布式集群的时候

在启动集群（master进程和worker进程）的时候，大家回忆一下，我们用的是哪个命令，用的是sbin/start-all.sh脚本

这个脚本一旦执行，就会直接在集群（节点，部署了spark安装包）中，启动master进程和所有worker进程

sbin/start-all.sh脚本，其实是用来便捷地快速启动整个spark standalone集群的

但是我们本讲呢，就是对spark standalone集群的启动脚本，作一个更加详细和全面的介绍和讲解

我们除了会用sbin/start-all.sh脚本，直接启动整个集群

还要会用另外两个脚本，单独启动master和worker进程

先讲理论，后面我们来做实验

直接启动master、worker集群，使用sbin/start-all.sh即可

如果你要单独，分别，启动master和worker进程的话

那么必须得先启动master进程，然后再启动worker进程，因为worker进程启动以后，需要向master进程去注册

反过来先启动worker进程，再启动这个master进程，可能会有问题

为什么我们有的时候也需要单独启动master和worker进程呢

因为我们后面会讲的，在单独启动两个进程的时候，是可以通过命令行参数，为进程配置一些独特的参数

比如说监听的端口号、web ui的端口号、使用的cpu和内存

比如你想单独给某个worker节点配置不同的cpu和内存资源的使用限制，那么就可以使用脚本单独启动这个worker进程的时候，通过命令行参数来设置

手动启动master进程

需要在某个部署了spark安装包的节点上，使用sbin/start-master.sh启动

master启动之后，启动日志就会打印一行spark://HOST:PORT URL出来，这就是master的URL地址

worker进程就会通过这个URL地址来连接到master进程，并且进行注册

另外，除了worker进程要使用这个URL意外，我们自己在编写spark代码时，也可以给SparkContext的setMaster()方法，传入这个URL地址

然后我们的spark作业，就会使用standalone模式连接master，并提交作业

此外，还可以通过http://MASTER\_HOST:8080 URL来访问master集群的监控web ui，那个web ui上，也会显示master的URL地址

剧透一下，后面会做单独启动master和worker进程的实验

启动master的时候，要观察什么呢？要观察一下日志中透露出来的这个master URL

在http://HOST:8080端口，观察一下master的URL地址

手动启动worker进程

需要在你希望作为worker node的节点上，在部署了spark安装包的前提下，使用sbin/start-slave.sh <master-spark-URL>在当前节点上启动

如上所述，使用sbin/start-slave.sh时，需要指定master URL

启动worker进程之后，再访问http://MASTER\_HOST:8080，在spark集群web ui上，就可以看见新启动的worker节点，包括该节点的cpu和内存资源等信息

一会儿实验，启动worker之后，要在master web ui上，观察一下新启动的worker节点

此外，以下参数是可以在手动启动master和worker的时候指定的

-h HOST, --host HOST 在哪台机器上启动，默认就是本机，这个很少配

-p PORT, --port PORT 在机器上启动后，使用哪个端口对外提供服务，master默认是7077，worker默认是随机的，也很少配

--webui-port PORT web ui的端口，master默认是8080，worker默认是8081，也很少配

-c CORES, --cores CORES 仅限于worker，总共能让spark作业使用多少个cpu core，默认是当前机器上所有的cpu core

-m MEM, --memory MEM 仅限于worker，总共能让spark作业使用多少内存，是100M或者1G这样的格式，默认是1g

-d DIR, --work-dir DIR 仅限于worker，工作目录，默认是SPARK\_HOME/work目录

--properties-file FILE master和worker加载默认配置文件的地址，默认是conf/spark-defaults.conf，很少配

咱们举个例子，比如说小公司里面，物理集群可能就一套，同一台机器上面，可能要部署Storm的supervisor进程，可能还要同时部署Spark的worker进程

机器，cpu和内存，既要供storm使用，还要供spark使用

这个时候，可能你就需要限制一下worker节点能够使用的cpu和内存的数量

小公司里面，搭建spark集群的机器可能还不太一样，有的机器比如说是有5个g内存，有的机器才1个g内存

那你对于1个g内存的机器，是不是得限制一下内存使用量，比如说500m

实验

1、启动master: 日志和web ui，观察master url

2、启动worker: 观察web ui，是否有新加入的worker节点，以及对应的信息

3、关闭master和worker

4、再次单独启动master和worker，给worker限定，就使用500m内存，跟之前看到的worker信息比对一下内存最大使用量

## 146\_Spark核心编程进阶-实验：单独启动master和worker进程

standalone模式启动集群命令详解

我们之前搭建那个伪分布式spark standalone集群的时候，以及我们最早搭建分布式集群的时候

在启动集群（master进程和worker进程）的时候，大家回忆一下，我们用的是哪个命令，用的是sbin/start-all.sh脚本

这个脚本一旦执行，就会直接在集群（节点，部署了spark安装包）中，启动master进程和所有worker进程

sbin/start-all.sh脚本，其实是用来便捷地快速启动整个spark standalone集群的

但是我们本讲呢，就是对spark standalone集群的启动脚本，作一个更加详细和全面的介绍和讲解

我们除了会用sbin/start-all.sh脚本，直接启动整个集群

还要会用另外两个脚本，单独启动master和worker进程

先讲理论，后面我们来做实验

直接启动master、worker集群，使用sbin/start-all.sh即可

如果你要单独，分别，启动master和worker进程的话

那么必须得先启动master进程，然后再启动worker进程，因为worker进程启动以后，需要向master进程去注册

反过来先启动worker进程，再启动这个master进程，可能会有问题

为什么我们有的时候也需要单独启动master和worker进程呢

因为我们后面会讲的，在单独启动两个进程的时候，是可以通过命令行参数，为进程配置一些独特的参数

比如说监听的端口号、web ui的端口号、使用的cpu和内存

比如你想单独给某个worker节点配置不同的cpu和内存资源的使用限制，那么就可以使用脚本单独启动这个worker进程的时候，通过命令行参数来设置

手动启动master进程

需要在某个部署了spark安装包的节点上，使用sbin/start-master.sh启动

master启动之后，启动日志就会打印一行spark://HOST:PORT URL出来，这就是master的URL地址

worker进程就会通过这个URL地址来连接到master进程，并且进行注册

另外，除了worker进程要使用这个URL意外，我们自己在编写spark代码时，也可以给SparkContext的setMaster()方法，传入这个URL地址

然后我们的spark作业，就会使用standalone模式连接master，并提交作业

此外，还可以通过http://MASTER\_HOST:8080 URL来访问master集群的监控web ui，那个web ui上，也会显示master的URL地址

单独启动master进程以后，你会看到打印出来一行日志文件的名称

通常就是放在logs/spark-root-org.apache.spark.deploy.master.Master-1-sparkupgrade1.out

通过这个日志，就可以查看master的启动日志

剧透一下，后面会做单独启动master和worker进程的实验

启动master的时候，要观察什么呢？要观察一下日志中透露出来的这个master URL

在http://HOST:8080端口，观察一下master的URL地址

手动启动worker进程

需要在你希望作为worker node的节点上，在部署了spark安装包的前提下，使用sbin/start-slave.sh <master-spark-URL>在当前节点上启动

如上所述，使用sbin/start-slave.sh时，需要指定master URL

启动worker进程之后，再访问http://MASTER\_HOST:8080，在spark集群web ui上，就可以看见新启动的worker节点，包括该节点的cpu和内存资源等信息

一会儿实验，启动worker之后，要在master web ui上，观察一下新启动的worker节点

此外，以下参数是可以在手动启动master和worker的时候指定的

-h HOST, --host HOST 在哪台机器上启动，默认就是本机，这个很少配

-p PORT, --port PORT 在机器上启动后，使用哪个端口对外提供服务，master默认是7077，worker默认是随机的，也很少配

--webui-port PORT web ui的端口，master默认是8080，worker默认是8081，也很少配

-c CORES, --cores CORES 仅限于worker，总共能让spark作业使用多少个cpu core，默认是当前机器上所有的cpu core

-m MEM, --memory MEM 仅限于worker，总共能让spark作业使用多少内存，是100M或者1G这样的格式，默认是1g

-d DIR, --work-dir DIR 仅限于worker，工作目录，默认是SPARK\_HOME/work目录

--properties-file FILE master和worker加载默认配置文件的地址，默认是conf/spark-defaults.conf，很少配

咱们举个例子，比如说小公司里面，物理集群可能就一套，同一台机器上面，可能要部署Storm的supervisor进程，可能还要同时部署Spark的worker进程

机器，cpu和内存，既要供storm使用，还要供spark使用

这个时候，可能你就需要限制一下worker节点能够使用的cpu和内存的数量

小公司里面，搭建spark集群的机器可能还不太一样，有的机器比如说是有5个g内存，有的机器才1个g内存

那你对于1个g内存的机器，是不是得限制一下内存使用量，比如说500m

实验

1、启动master: 日志和web ui，观察master url

2、启动worker: 观察web ui，是否有新加入的worker节点，以及对应的信息

3、单独关闭master和worker，顺序得反过来，先关worker，再关master

4、再次单独启动master和worker，给worker限定--memory参数，就使用500m内存，再到web ui，跟之前看到的worker信息比对一下内存最大使用量

5、集体关闭集群，再启动集群，保证后面可以正常使用

## 147\_Spark核心编程进阶-worker节点配置以及spark-evnsh参数详解

standalone部署细节以及相关参数详解

这里主要讲解spark部署过程中的一些细节

配置集群中的worker节点

首先第一点，如果你想将某台机器部署成standalone集群架构中的worker节点（会运行worker daemon进程）

那么你就必须在那台机器上部署咱们的spark安装包

配置conf/slaves文件

在conf/salves文件中，哪些机器是作为worker节点的，可以配置你要在哪些机器上启动worker进程

之前已经演示过分布式的spark standalone集群部署了，需要将master和worker进程所在的节点，都部署好spark安装包

默认情况下，没有conf/slaves这个文件，只有一个conf/slaves.template，而且还是空的

此时，就只是在当前主节点上启动一个master进程和一个worker进程，此时就是master进程和worker进程在一个节点上，也就是伪分布式部署

我们升级课程中就是用的伪分布式部署，但是之前第一个大的阶段，用的是分布式的部署方式，就是说，你得去手动将slaves.template拷贝为一份slaves文件

然后需要手动去编辑conf/slaves文件中的内容

此时，在conf/slaves文件中，你可以编辑要作为worker节点的机器，比如说hostname，或者ip地址，都可以，一个机器是一行

配置以后，所有的节点上，spark部署安装包中，都得去拷贝一份这个conf/slaves文件

这里master机器和worker机器之间的访问时通过linux ssh方式进行的，所以要配置多个机器之间的ssh免密码连接

conf/spark-env.sh

spark-env.sh文件，它的地位，就类似于hadoop中的core-site.xml、hdfs-site.xml等等

应该说是spark中最核心的一份配置文件

这份文件，可以对整个spark的集群部署，各个master和worker进程的相应的行为，进行充分和细节化的配置

SPARK\_MASTER\_IP 指定master进程所在的机器的ip地址

SPARK\_MASTER\_PORT 指定master监听的端口号（默认是7077）

SPARK\_MASTER\_WEBUI\_PORT 指定master web ui的端口号（默认是8080）

大家会发现什么，就是这个东东，怎么跟我们上一讲，讲解的单独启动master和worker进程的命令行参数，作用，是一样的

sbin/start-master.sh --port 7078，类似这种方式，貌似可以指定一样的配置属性

我明确告诉大家，这个作用的确是一模一样的

你可以在spark-evn.sh中就去配置好

但是有时呢，可能你会遇到需要临时更改配置，并启动master或worker进程的情况

此时就比较适合，用sbin/start-master.sh这种脚本的命令行参数，来设置这种配置属性

但是通常来说呢，还是推荐在部署的时候，通过spark-env.sh来设定

脚本命令行参数通常用于临时的情况

SPARK\_MASTER\_OPTS 设置master的额外参数，使用"-Dx=y"设置各个参数

比如说export SPARK\_MASTER\_OPTS="-Dspark.deploy.defaultCores=1"

参数名 默认值 含义

spark.deploy.retainedApplications 200 在spark web ui上最多显示多少个application的信息

spark.deploy.retainedDrivers 200 在spark web ui上最多显示多少个driver的信息

spark.deploy.spreadOut true 资源调度策略，spreadOut会尽量将application的executor进程分布在更多worker上，适合基于hdfs文件计算的情况，提升数据本地化概率；非spreadOut会尽量将executor分配到一个worker上，适合计算密集型的作业

spark.deploy.defaultCores 无限大 每个spark作业最多在standalone集群中使用多少个cpu core，默认是无限大，有多少用多少

spark.deploy.timeout 60 单位秒，一个worker多少时间没有响应之后，master认为worker挂掉了

SPARK\_LOCAL\_DIRS spark的工作目录，包括了shuffle map输出文件，以及持久化到磁盘的RDD等

SPARK\_WORKER\_PORT worker节点的端口号，默认是随机的

SPARK\_WORKER\_WEBUI\_PORT worker节点的web ui端口号，默认是8081

SPARK\_WORKER\_CORES worker节点上，允许spark作业使用的最大cpu数量，默认是机器上所有的cpu core

SPARK\_WORKER\_MEMORY worker节点上，允许spark作业使用的最大内存量，格式为1000m，2g等，默认最小是1g内存

就是说，有些master和worker的配置，可以在spark-env.sh中部署时即配置，但是也可以在start-slave.sh脚本启动进程时命令行参数设置

但是命令行参数的优先级比较高，会覆盖掉spark-env.sh中的配置

比如说，上一讲我们的实验，worker的内存默认是1g，但是我们通过--memory 500m，是可以覆盖掉这个属性的

SPARK\_WORKER\_INSTANCES 当前机器上的worker进程数量，默认是1，可以设置成多个，但是这时一定要设置SPARK\_WORKER\_CORES，限制每个worker的cpu数量

SPARK\_WORKER\_DIR spark作业的工作目录，包括了作业的日志等，默认是spark\_home/work

SPARK\_WORKER\_OPTS worker的额外参数，使用"-Dx=y"设置各个参数

参数名 默认值 含义

spark.worker.cleanup.enabled false 是否启动自动清理worker工作目录，默认是false

spark.worker.cleanup.interval 1800 单位秒，自动清理的时间间隔，默认是30分钟

spark.worker.cleanup.appDataTtl 7 \* 24 \* 3600 默认将一个spark作业的文件在worker工作目录保留多少时间，默认是7天

SPARK\_DAEMON\_MEMORY 分配给master和worker进程自己本身的内存，默认是1g

SPARK\_DAEMON\_JAVA\_OPTS 设置master和worker自己的jvm参数，使用"-Dx=y"设置各个参数

SPARK\_PUBLISC\_DNS master和worker的公共dns域名，默认是没有的

这里提示一下，大家可以观察一下，咱们的内存使用情况

在没有启动spark集群之前，我们的内存使用是1个多g，启动了spark集群之后，就一下子耗费到2个多g

每次又执行一个作业时，可能会耗费到3个多g左右

所以大家就明白了，为什么之前用分布式的集群，每个worker节点才1个g内存，根本是没有办法使用standalone模式和yarn模式运行作业的

上一节课讲解了单独启动spark standalone集群

这一节课，就给大家列出spark所有的启动和关闭shell脚本

sbin/start-all.sh 根据配置，在集群中各个节点上，启动一个master进程和多个worker进程

sbin/stop-all.sh 在集群中停止所有master和worker进程

sbin/start-master.sh 在本地启动一个master进程

sbin/stop-master.sh 关闭master进程

sbin/start-slaves.sh 根据conf/slaves文件中配置的worker节点，启动所有的worker进程

sbin/stop-slaves.sh 关闭所有worker进程

sbin/start-slave.sh 在本地启动一个worker进程

## 148\_Spark核心编程进阶-实验：local模式提交spark作业

之前讲解过，spark作业运行集群，有两种部署方式，一种是Spark Standalone集群，还有一种是YARN集群+Spark客户端

所以，我们认为，提交spark作业的两种主要方式，就是Spark Standalone和YARN，这两种方式，分别还分为两种模式，分别是client mode和cluster mode

在体验standalone提交模式之前，咱们呢，先得体验一种Spark中最基本的一种提交模式

也就是，所谓的local模式

local模式的基本原理

local模式下，没有所谓的master+worker这种概念

local模式，相当于，启动一个本地进程，然后在一个进程内，模拟spark集群中作业的运行

一个spark作业，就对应了进程中的一个或多个executor线程

就开始执行，包括作业的调度，task分配

在实际工作当中，local模式，主要用于测试

最常见的就是在我们的开发环境中，比如说eclipse，或者是InteliJ IDEA

直接运行spark程序，此时就可以采用local模式

本次实验，通过在eclipse中使用local来运行，以及在生产机器上（linux机器）上用local模式来运行

**wordcount\_local.sh**

|  |
| --- |
| /usr/local/spark/bin/spark-submit \  --class cn.spark.study.core.WordCount \  --num-executors 1 \  --driver-memory 100m \  --executor-memory 100m \  --executor-cores 1 \  /usr/local/test/spark-study-java-0.0.1-SNAPSHOT-jar-with-dependencies.jar \ |

WordCount.java

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.FlatMapFunction;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import scala.Tuple2;  public class WordCount {    public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("WordCount");  // .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);  // String file = "C://Users//Administrator//Desktop//hello.txt";  String file = "hdfs://192.168.0.103:9000/test/hello.txt";    JavaRDD<String> lines = sc.textFile(file);    JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {    private static final long serialVersionUID = 1L;  @Override  public Iterable<String> call(String line) throws Exception {  return Arrays.asList(line.split(" "));  }    });    JavaPairRDD<String, Integer> pairs = words.mapToPair(    new PairFunction<String, String, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, Integer> call(String word) throws Exception {  return new Tuple2<String, Integer>(word, 1);  }  });  JavaPairRDD<String, Integer> wordCounts = pairs.reduceByKey(  new Function2<Integer, Integer, Integer>() {    private static final long serialVersionUID = 1L;  @Override  public Integer call(Integer v1, Integer v2) throws Exception {  return v1 + v2;  }  });  List<Tuple2<String, Integer>> wordCountList = wordCounts.collect();  for(Tuple2<String, Integer> wordCount : wordCountList) {  System.out.println(wordCount);  }  sc.close();  }  } |

## 149\_Spark核心编程进阶-实验：standalone client模式提交spark作业

local模式下，我们都不会放到生产机器上面去提交

local模式，其实仅仅用于eclipse中运行spark作业，以及打断点，调试spark作业来用

通常，用local模式执行，我们都会手工生成一份数据，来使用

通常情况下来说，部署在测试机器上去，进行测试运行spark作业的时候

都是使用client模式（standalone client模式，或standalone cluster模式）

client模式下，提交作业以后，driver在本机启动，可以实时看到详细的日志信息，方便你追踪和排查错误

standalone模式下提交spark作业

standalone模式提交，唯一与local区别，就是一定要想办法将master设置成spark://master\_ip:port，比如spark://192.168.0.103:7077

三种设置master的方式

1、硬编码: SparkConf.setMaster("spark://IP:PORT")

2、spark-submit: --master spark://IP:PORT

3、spark-shell: --master spark://IP:PORT

通常来说，上面三种写法，使用第二种，是最合适

一般我们都是用spark-submit脚本来提交spark application的，很少用spark-shell，spark-shell仅仅用于实验和测试一些特别简单的spark作业

使用spark-submit脚本来提交时，还可以指定两种deploy mode，spark standalone cluster支持client mode和cluster mode

client mode下，你在哪台机器上用spark-submit脚本提交application，driver就会在那台机器上启动

cluster mode下，driver会通过master进程，随机被分配给某一个worker进程来启动

这里我们就知道了，standalone模式，是要在spark-submit中，用--master指定Master进程的URL

其次，使用standalone client模式或cluster模式，是要在spark-submit中，使用--deploy-mode client/cluster来设置

默认，如果你不设置--deploy-mode，那么就是client模式

使用spark-submit脚本来提交application时，application jar是会自动被分发到所有worker节点上去的

对于你的application依赖的额外jar包，可以通过spark-submit脚本中的--jars标识，来指定，可以使用逗号分隔多个jar

比如说，你写spark-sql的时候，有的时候，在作业中，要往mysql中写数据，此时可能会出现找不到mysql驱动jar包的情况

此时，就需要你手动在spark-submit脚本中，使用--jars，加入一些依赖的jar包

我们提交standalone client模式的作业

1、--master和--deploy-mode，来提交作业

2、观察作业执行日志，可以看到去连接spark://192.168.0.103:7077 URL的master

3、观察一些spark web ui，可以看到completed applications一栏中，有我们刚刚运行的作业，比对一下ui上的applicationId和日志中的applicationId

4、重新再执行一次作业，一执行，立即看一下jps查看进程，看看standalone client模式提交作业的时候，当前机器上会启动哪些进程

SparkSubmit: 也就是我们的driver进程，在本机上启动（spark-submit所在的机器）

CoarseGrainedExecutorBackend（内部持有一个Executor对象）: 在你的执行spark作业的worker机器上，肯定会给你的作业分配和启动一个executor进程，具体名字就是那个

## 150\_Spark核心编程进阶-实验：standalonecluster模式提交spark作业

standalone cluster模式

通常用于，spark作业部署到生产环境中去使用，是用standalone cluster模式

因为这种模式，会由master在集群中，某个节点上，来启动driver，然后driver会进行频繁的作业调度

此时driver跟集群在一起，那么是性能比较高的

standalone client模式，在spark-submit脚本执行的机器上，会启动driver进程，然后去进行整个作业的调度

通常来说，你的spark-submit脚本能够执行的机器，也就是，作为一个开发人员能够登录的机器，通常不会直接是spark集群部署的机器

因为，随便谁，都登录到spark集群中，某个机器上去，执行一个脚本，这个太扯淡了，没有安全性可言了

所有此时呢，是这样子，用client模式，你的机器，可能与spark集群部署的机器，都不在一个机房，或者距离很远，那么此时通常遥远的频繁的网络通信

会影响你的整个作业的执行性能

此外，standalone cluster模式，还支持监控你的driver进程，并且在driver挂掉的时候，自动重启该进程

要使用这个功能，在spark-submit脚本中，使用--supervise标识即可

这个跟我们的这个作业关系不是太大，主要是用于spark streaming中的HA高可用性，配置driver的高可用

如果想要杀掉反复挂掉的driver进程，使用以下命令即可: bin/spark-class org.apache.spark.deploy.Client kill <master url> <driver ID>

如果要查看driver id，通过http://<maser url>:8080即可查看到

实验观察

1、日志：命令行只会打印一点点内容，没有什么东西; 主要呢，是在web ui上，可以看到日志

2、web ui: running drivers出现，此时就可以看到在spark集群中执行的driver的信息; completed drivers

3、进程: SparkSubmit一闪而过，仅仅只是将driver注册到master上，由master来启动driver，马上就停止了; 在Worker上，会启动DriverWrapper进程

如果能够申请到足够的cpu资源，其实还会在其他worker上，启动CoarseGrainedExecutorBackend进程

4、可以通过点击running中的application，去查看作业中的每个job、每个stage、每个executor和task的执行信息，4040端口来看的

5、对于正在运行的driver或application，可以在web ui上，点击kill按钮，给杀掉

首先呢，cluster模式下，driver也是要通过worker来启动的，executor也是要通过worker来启动的

首先，我们可以看到，此时driver已经启动起来了，在web ui上是可以看到的，包括driver ID

然后呢，通过web ui就可以看到，driver在唯一的worker上启动了，已经获取到了一个cpu core了

此时，driver去跟master申请资源，启动一个executor进程，但是问题来了，此时我们的worker进程，就只有一个，而且只有一个cpu core

那么，master的资源调度算法中，始终无法找到还有空闲cpu资源的worker，所以作业一直处于等待，waiting的一个状态

所以，我们的作业在当前1个cpu core下，是无法通过cluster模式来运行的

但是至少我们可以看到，cluster模式的确奏效了，因为已经开始使用cluster模式来执行作业了

包括driver的启动，开始申请executor

## 151\_Spark核心编程进阶-standalone模式下的多作业资源调度

standalone多作业资源调度问题

目前来说，standalone集群对于同时提交上来的多个作业，仅仅支持FIFO调度策略，也就是先入先出

默认情况下，集群对多个作业同时执行的支持是不好的，没有办法同时执行多个作业

因为先提交上来的每一个作业都会尝试使用集群中所有可用的cpu资源，此时相当于就是只能支持作业串行起来，一个一个运行了

此时，如果我们希望能够支持多作业同时运行，那么就需要调整一下资源参数

我们可以设置spark.cores.max参数，来限制每个作业能够使用的最大的cpu core数量，这样先提交上来的作业不会使用所有的cpu资源

后面提交上来的作业就可以获取到资源，也可以同时并行运行了

比如说，咱们集群一共有20个节点，每个节点是8核，160 cpu core

ok，那么，如果你不限制每个作业获取的最大cpu资源大小，而且在你spark-submit的时候，或者说，你就设置了num-executors，total-cores，160

此时，你的作业是会使用所有的cpu core资源的

所以，如果我们可以通过设置全局的一个参数，让每个作业最多只能获取到一部分cpu core资源

那么，后面提交上来的作业，就也可以获取到一部分资源

standalone集群，才可以支持同时执行多个作业

使用SparkConf或spark-submit中的--conf标识，设置参数即可

比如说:

SparkConf conf = new SparkConf()

.set("spark.cores.max", "10")

通常不建议使用SparkConf，硬编码，来设置一些属性，不够灵活

建议使用spark-submit来设置属性

--conf spark.cores.max=10

此外，还可以直接通过spark-env.sh配置每个application默认能使用的最大cpu数量来进行限制，默认是无限大，此时就不需要每个application都自己手动设置了

在spark-env.sh中配置spark.deploy.defaultCores即可

比如说: export SPARK\_MASTER\_OPTS="-Dspark.deploy.defaultCores=10"

## 152\_Spark核心编程进阶-standalone模式下的作业监控与日志记录

standalone作业监控和日志记录

standalone模式下的作业的监控，很简单，就是通过我们之前给大家看的spark web ui

spark standalone模式，提供了一个web界面来让我们监控集群，并监控所有的作业的运行

web界面上，提供了master和worker的相关信息，默认的话，我们的web界面是运行在master机器上的8080端口

之前已经讲解过，可以通过配置spark-env.sh文件等方式，来配置web ui的端口

spark web ui

1、哪些作业在跑

2、哪些作业跑完了，花了多少时间，使用了多少资源

3、哪些作业失败了

application web ui

作业监控这一块儿，其实很重要的一点是，作业的web ui

我们之前给大家讲过，application detail ui，其实就是作业的driver所在的机器的4040端口

可以看到job、stage、task的详细运行信息，shuffle read、shuffle write、gc、运行时间、每个task分配的数据量

定位很多性能问题、troubleshooting等等，task数据分布不允许，那么就是数据倾斜

哪个stage运行的时间最慢，通过之前讲解的stage划分算法，去你的代码里定位到，那个stage对应的是哪一块儿代码，你的那段代码为什么会运行太慢

是不是可以用我们之前讲解的一些性能优化的策略，去优化一下性能

但是有个问题，作业运行完了以后，我们就看不到了

此时跟history server有关，我们后面会讲解，全局的去配置整个spark集群的监控和日志

history server

日志记录

1、系统级别的，spark自己的日志记录

2、我们在程序里面，用log4j，或者System.out.println打印出来的日志

spark web ui中可以看到

1、看每个application在每个executor上的日志

2、stdout，可以显示我们用System.out.println打印出来的日志，stderr，可以显示我们用System.err.println打印出来的日志

此外，我们自己在spark作业代码中，打出来的日志，比如用System.out.println()等，是打到每个作业在每个节点的工作目录中去的

默认是SPARK\_HOME/work目录下

这个目录下，每个作业都有两个文件，一个是stdout，一个是stderr，分别代表了标准输出流和异常输出流

## 153\_Spark核心编程进阶-实验：运行中作业监控以及手工打印日志

## 154\_Spark核心编程进阶-yarn-client模式原理讲解

1、driver在本机启动

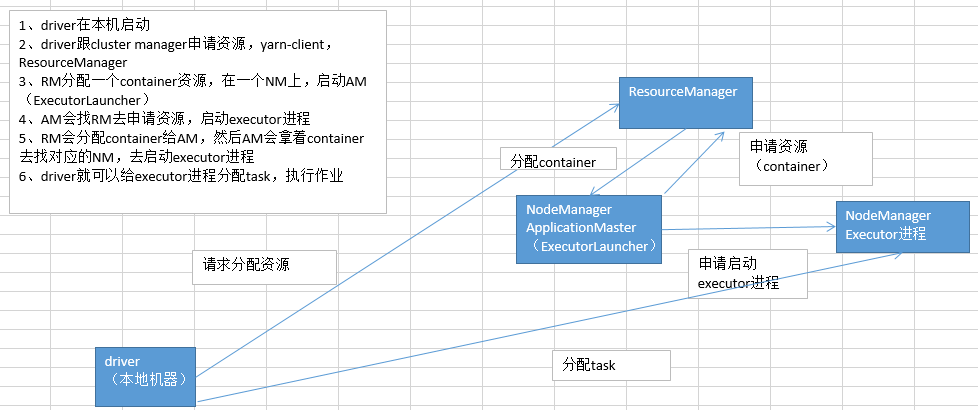
2、driver跟cluster manager申请资源，yarn-client，ResourceManager

3、RM分配一个container资源，在一个NM上，启动AM（ExecutorLauncher）

4、AM会找RM去申请资源，启动executor进程

5、RM会分配container给AM，然后AM会拿着container去找对应的NM，去启动executor进程

6、driver就可以给executor进程分配task，执行作业



## 155\_Spark核心编程进阶-yarn-cluster模式原理讲解

**yarn-cluster与yarn-client唯一的不同**

1**、yarn-client**下，driver运行在spark-submit提交的机器上，ApplicationMaster只是相当于一个ExecutorLauncher，仅仅负责申请启动executor；具体负责调度的，还是driver

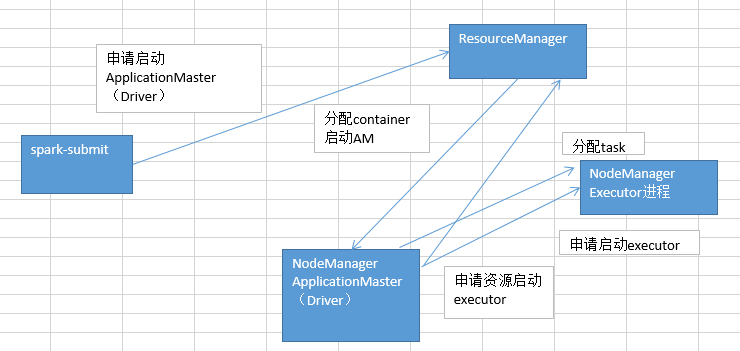
2、**yarn-cluster**下，ApplicationMaster是driver；具体负责调度的，也是ApplicationMaster

yarn-client下，driver所在机器可能跟yarn集群不在一起，所以性能可能比较差

yarn-cluster下，driver（AM）就在yarn集群中，进行复杂调度的时候，网络通信性能较好

**yarn-client用于在测试环境中，进行测试，方便你看日志**

**yarn-cluster推荐作业部署上线运行时使用，性能比较好**



## 156\_Spark核心编程进阶-实验：yarn-client模式提交spark作业

yarn模式运行spark作业

yarn运行spark作业的大前提

如果想要让spark作业可以运行在yarn上面，那么首先就必须在spark-env.sh文件中，配置HADOOP\_CONF\_DIR或者YARN\_CONF\_DIR属性，值为hadoop的配置文件目录

即HADOOP\_HOME/etc/hadoop，其中包含了hadoop和yarn所有的配置文件，比如hdfs-site、yarn-site等

spark需要这些配置来读写hdfs，以及连接到yarn resourcemanager上，这个目录中包含的配置文件都会被分发到yarn集群中去的

**在yarn模式下，也有两种运行模式**

**yarn-client**模式下，driver进程会运行在提交作业的机器上，ApplicationMaster仅仅只是负责为作业向yarn申请资源（executor）而已，driver还是会负责作业调度

**yarn-cluster**模式下，driver进程会运行在yarn集群的某个工作节点上，作为一个ApplicationMaster进程运行

跟spark standalone模式不同，通常不需要使用--master指定master URL

cluster manager，也就是yarn resourcemanager的地址，会自动从hadoop配置目录中的配置文件中后去

因此，设置--master时，指定为yarn-client或yarn-cluster即可，也就代表了上面说的两种deploy mode了

这里提示一下，与standalone模式类似，yarn-client模式通常建议在测试时使用，方便你直接在提交作业的机器上查看日志

但是作业实际部署到生产环境进行运行的时候，还是使用yarn-cluster模式

使用yarn-cluster模式提交时，使用以下语法即可:

|  |
| --- |
| ./bin/spark-submit \  --class path.to.your.Class \  --master yarn-cluster \  [options] \  <app jar> \  [app options] |

比如如下脚本示例:

|  |
| --- |
| $ ./bin/spark-submit --class org.leo.spark.study.WordCount \  --master yarn-cluster \  --num-executors 1 \  --driver-memory 100m \  --executor-memory 100m \  --executor-cores 1 \  --queue hadoop队列 \  /usr/local/spark-study/spark-study.jar \ |

--queue，在大公司里面，队列很重要

不同的数据部门，或者是不同的大数据项目，共用同一个yarn集群，运行spark作业

推荐一定要用--queue，指定不同的hadoop队列，做项目或部门之间的队列隔离

在yarn-cluster模式下运行时，首先在本地机器会启动一个YARN client进程

YARN client进程会连接到resourcemanager上，然后启动一个spark的ApplicationMaster进程

接着我们自己写的main类，会作为一个ApplicationMaster进程的子线程来运行

提交作业的本地机器上，YARN client进程会周期性地跟ApplicationMaster进程，拉取作业运行的进度，并打印在控制台上

一旦我们的作业完成之后，YARN client进程也就会退出了

使用yarn-client模式提交时，使用以下语法即可: ./bin/spark-shell --master yarn-client

添加其他的jar

在spark-submit脚本中，使用--jars命令即可

实验中要观察的几个点

1、日志

命令行日志: 会详细打印你的所有的日志

web ui看日志: stdout、stderr

2、web ui，spark://192.168.0.103:8080这种URL了，因为那是standalone集群的监控web ui

yarn的web ui上，才可以看到，stdout、stderr

http://192.168.0.103:8088/，URL，YARN web ui，来做作业的监控

http://driver:4040，通过yarn，进入spark application web ui

3、进程

driver是什么进程

ApplicationMaster进程

executor进程

yarn模式下

我们的工程jar，是要拷贝到hdfs上面去的

而且它的replication，副本数量，默认是跟hadoop中的副本数量一样的

hdfs，一个datanode，没有办法做replication，所以也没有退出safemode

## 157\_Spark核心编程进阶-yarn模式下日志查看详解

yarn模式下调试运行中的spark作业

在yarn模式下，spark作业运行相关的executor和ApplicationMaster都是运行在yarn的container中的

一个作业运行完了以后，yarn有两种方式来处理spark作业打印出的日志

**第一种是聚合日志方式（推荐，比较常用）**

这种方式的话，顾名思义，就是说，将散落在集群中各个机器上的日志，最后都给聚合起来，让我们可以统一查看

如果打开了日志聚合的选项，即yarn.log-aggregation-enable，container的日志会拷贝到hdfs上去，并从机器中删除

对于这种情况，可以使用yarn logs -applicationId <app ID>命令，来查看日志

yarn logs命令，会打印出application对应的所有container的日志出来，当然，因为日志是在hdfs上的，我们自然也可以通过hdfs的命令行来直接从hdfs中查看日志

日志在hdfs中的目录，可以通过查看yarn.nodemanager.remote-app-log-dir和yarn.nodemanager.remote-app-log-dir-suffix属性来获知

**第二种 web ui（如果你有精力的话，可以去配一下）**

日志也可以通过spark web ui来查看executor的输出日志

但是此时需要启动History Server，需要让spark history server和mapreduce history server运行着

并且在yarn-site.xml文件中，配置yarn.log.server.url属性

spark history server web ui中的log url，会将你重定向到mapreduce history server上，去查看日志

**第三种 分散查看（通常不推荐）**

如果没有打开聚合日志选项，那么日志默认就是散落在各个机器上的本次磁盘目录中的，在YARN\_APP\_LOGS\_DIR目录下

根据hadoop版本的不同，通常在/tmp/logs目录下，或者$HADOOP\_HOME/logs/userlogs目录下

如果你要查看某个container的日志，那么就得登录到那台机器上去，然后到指定的目录下去，找到那个日志文件，然后才能查看

## 158\_Spark核心编程进阶-yarn模式相关参数详解

|  |  |  |
| --- | --- | --- |
| **属性名称** | **默认值** | **含义** |
| spark.yarn.am.memory | 512m | client模式下，YARN Application Master使用的内存总量 |
| spark.yarn.am.cores | 1 | client模式下，Application Master使用的cpu数量 |
| spark.driver.cores | 1 | cluster模式下，driver使用的cpu core数量，driver与Application Master运行在一个进程中，所以也控制了Application Master的cpu数量 |
| spark.yarn.am.waitTime | 100s | cluster模式下，Application Master要等待SparkContext初始化的时长; client模式下，application master等待driver来连接它的时长 |
| spark.yarn.submit.file.replication | hdfs副本数 | 作业写到hdfs上的文件的副本数量，比如工程jar，依赖jar，配置文件等，最小一定是1 |
| spark.yarn.preserve.staging.files | FALSE | 如果设置为true，那么在作业运行完之后，会避免工程jar等文件被删除掉 |
| spark.yarn.scheduler.heartbeat.interval-ms | 3000 | application master向resourcemanager发送心跳的间隔，单位ms |
| spark.yarn.scheduler.initial-allocation.interval | 200ms | application master在有pending住的container分配需求时，立即向resourcemanager发送心跳的间隔 |
| spark.yarn.max.executor.failures | executor数量\*2，最小3 | 整个作业判定为失败之前，executor最大的失败次数 |
| spark.yarn.historyServer.address | 无 | spark history server的地址 |
| spark.yarn.dist.archives | 无 | 每个executor都要获取并放入工作目录的archive |
| spark.yarn.dist.files | 无 | 每个executor都要放入的工作目录的文件 |
| spark.executor.instances | 2 | 默认的executor数量 |
| spark.yarn.executor.memoryOverhead | executor内存10% | 每个executor的堆外内存大小，用来存放诸如常量字符串等东西 |
| spark.yarn.driver.memoryOverhead | driver内存7% | 同上 |
| spark.yarn.am.memoryOverhead | AM内存7% | 同上 |
| spark.yarn.am.port | 随机 | application master端口 |
| spark.yarn.jar | 无 | spark jar文件的位置 |
| spark.yarn.access.namenodes | 无 | spark作业能访问的hdfs namenode地址 |
| spark.yarn.containerLauncherMaxThreads | 25 | application master能用来启动executor container的最大线程数量 |
| spark.yarn.am.extraJavaOptions | 无 | application master的jvm参数 |
| spark.yarn.am.extraLibraryPath | 无 | application master的额外库路径 |
| spark.yarn.maxAppAttempts |  | 提交spark作业最大的尝试次数 |
| spark.yarn.submit.waitAppCompletion | TRUE | cluster模式下，client是否等到作业运行完再退出 |

## 159\_Spark核心编程进阶-spark工程打包以及spark-submit详解

spark工程打包与spark-submit的关系

我们在eclipse编写代码，基于spark api开发自己的大数据计算和处理程序

将我们写好的spark工程打包，比如说java开发环境中，就使用maven assembly插件来打包，将第三方依赖包都打进去

jar包，生产环境中，通常是本机ssh远程连接到部署了spark客户端的linux机器上，使用scp命令将本机的jar包拷贝到远程linux机器上

然后在那个linux机器上，用spark-submit脚本，去将我们的spark工程，作为一次作业/application，提交到集群上去执行

课程中，使用WinSCP工具，从本机windows系统，上传到虚拟机中的linux上去

打包Spark工程

要使用spark-submit提交spark应用程序，首先就必须将我们的spark工程（java/scala）打包成一个jar包

如果我们的spark工程，依赖了其他一些第三方的组件，那就必须把所有组件jar包都打包到工程中，这样才能将完整的工程代码和第三方依赖都分发到spark集群中去

所以必须创建一个assembly jar来包含你所有的代码和依赖，sbt和maven都有assembly插件的（咱们课程里的java工程，就使用了maven的assembly插件）

配置依赖的时候（比如maven工程的pom.xml），可以把Spark和Hadoop的依赖配置成provided类型的依赖，也就是说仅仅开发和编译时有效，打包时就不将这两种依赖打到jar包里去了，因为集群管理器都会提供这些依赖

打好一个assembly jar包之后（也就是你的spark应用程序工程），就可以使用spark-submit脚本提交jar包中的spark应用程序了

spark-submit是什么？

在spark安装目录的bin目录中，有一个spark-submit脚本，这个脚本主要就是用来提交我们自己开发的spark应用程序到集群上执行

spark-submit可以通过一个统一的接口，将spark应用程序提交到所有spark支持的集群管理器上（Standalone（Master）、Yarn（ResourceManager）等）

所以我们并不需要为每种集群管理器都做特殊的配置

--master

1、如果不设置，那么就是local模式

2、如果设置spark://打头的URL，那么就是standalone模式，会提交到指定的URL的Master进程上去

3、如果设置yarn-打头的，那么就是yarn模式，会读取hadoop配置文件，然后连接ResourceManager

## 160\_Spark核心编程进阶-spark-submit示例以及基础参数讲解

使用spark-submit提交spark应用

将我们的spark工程打包好之后，就可以使用spark-submit脚本提交工程中的spark应用了

spark-submit脚本会设置好spark的classpath环境变量（用于类加载）和相关的依赖，而且还可以支持多种不同的集群管理器和不同的部署模式

以下是一个spark应用提交脚本的示例，以及其基本语法

一般会将执行spark-submit脚本的命令，放置在一个自定义的shell脚本里面，所以说这是比较灵活的一种做法

建议大家还是要熟悉linux操作系统，不需要太熟悉，会基本的操作就可以了

wordcount.sh

/usr/local/spark/bin/spark-submit \

--class org.leo.spark.study.WordCount \

--master spark://192.168.0.101:7077 \

--deploy-mode client \

--conf <key>=<value> \

/usr/local/spark-study/spark-study.jar \

${1}

以下是上面的spark-submit

--class: spark应用程序对应的主类，也就是spark应用运行的主入口，通常是一个包含了main方法的java类或scala类，需要包含全限定包名，比如org.leo.spark.study.WordCount

--master: spark集群管理器的master URL，standalone模式下，就是ip地址+端口号，比如spark://192.168.0.101:7077，standalone默认端口号就是7077

--deploy-mode: 部署模式，决定了将driver进程在worker节点上启动，还是在当前本地机器上启动；默认是client模式，就是在当前本地机器上启动driver进程，如果是cluster，那么就会在worker上启动

--conf: 配置所有spark支持的配置属性，使用key=value的格式；如果value中包含了空格，那么需要将key=value包裹的双引号中

application-jar: 打包好的spark工程jar包，在当前机器上的全路径名

application-arguments: 传递给主类的main方法的参数; 在shell中用${1}这种格式获取传递给shell的参数；然后在比如java中，可以通过main方法的args[0]等参数获取

## 163\_Spark核心编程进阶-spark-submit多个示例以及常用参数详解

**# 使用local本地模式，以及8个线程运行**

# --class 指定要执行的main类

# --master 指定集群模式，local，本地模式，local[8]，进程中用几个线程来模拟集群的执行

./bin/spark-submit \

--class org.leo.spark.study.WordCount \

--master local[8] \

/usr/local/spark-study.jar \

**# 使用standalone client模式运行**

# executor-memory，指定每个executor的内存量，这里每个executor内存是2G

# total-executor-cores，指定所有executor的总cpu core数量，这里所有executor的总cpu core数量是100个

./bin/spark-submit \

--class org.leo.spark.study.WordCount \

--master spark://192.168.0.101:7077 \

--executor-memory 2G \

--total-executor-cores 100 \

/usr/local/spark-study.jar \

**# 使用standalone cluster模式运行**

# supervise参数，指定了spark监控driver节点，如果driver挂掉，自动重启driver

./bin/spark-submit \

--class org.leo.spark.study.WordCount \

--master spark://192.168.0.101:7077 \

--deploy-mode cluster \

--supervise \

--executor-memory 2G \

--total-executor-cores 100 \

/usr/local/spark-study.jar \

**# 使用yarn-cluster模式运行**

# num-executors，指定总共使用多少个executor运行spark应用

./bin/spark-submit \

--class org.leo.spark.study.WordCount \

--master yarn-cluster \

--executor-memory 20G \

--num-executors 50 \

/usr/local/spark-study.jar \

**# 使用standalone client模式，运行一个python应用**

./bin/spark-submit \

--master spark://192.168.0.101:7077 \

/usr/local/python-spark-wordcount.py \

**比较常用的配置：**

--class

application jar

--master

--num-executors

--executor-memory

--total-executor-cores

--supervise

--executor-cores

--driver-memory

**公司比较常用的资源：**

./bin/spark-submit \

--class org.leo.spark.study.WordCount \

--master yarn-cluster \

--num-executors 100 \

--executor-cores 2 \ //一般比较小的2-4

--executor-memory 6G \ //一般6-10

--driver-memory 1G \ //一般1g够了

/usr/local/spark-study.jar \

## 164\_Spark核心编程进阶-SparkConf、spark-submit以及spark-defaultsconf

默认的配置属性

spark-submit脚本会自动加载conf/spark-defaults.conf文件中的配置属性，并传递给我们的spark应用程序

加载默认的配置属性，一大好处就在于，我们不需要在spark-submit脚本中设置所有的属性

比如说，默认属性中有一个spark.master属性，所以我们的spark-submit脚本中，就不一定要显式地设置--master，默认就是local

SparkConf.getOrElse("spark.master", "local")

spark配置的优先级如下: SparkConf、spark-submit、spark-defaults.conf

spark.default.parallelism

SparkConf.set("spark.default.parallelism", "100")

spark-submit: --conf spark.default.parallelism=50

spark-defaults.conf: spark.default.parallelism 10

如果想要了解更多关于配置属性的信息，可以在spark-submit脚本中，使用--verbose，打印详细的调试信息

使用spark-submit设置属性

虽然说SparkConf设置属性的优先级是最高的，但是有的时候咱们可能不希望在代码中硬编码一些配置属性，否则每次修改了参数以后

还得去代码里修改，然后得重新打包应用程序，再部署到生产机器上去，非常得麻烦

对于上述的情况，我们可以在代码中仅仅创建一个空的SparkConf对象，比如: val sc = new SparkContext(new SparkConf())

然后可以在spark-submit脚本中，配置各种属性的值，比如

./bin/spark-submit \

--name "My app" \

--master local[4] \

--conf spark.shuffle.spill=false \

--conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" \

myApp.jar

这里的spark.shuffle.spill属性，我们本来如果是在代码中，SparkConf.set("spark.shuffle.spill", "false")来配置的

此时在spark-submit中配置了，不需要更改代码，就可以更改属性，非常得方便，

尤其是对于spark程序的调优，格外方便，因为调优说白了，就是不断地调整各种各样的参数，然后反复跑反复试的过程

spark的属性配置方式

spark-shell和spark-submit两个工具，都支持两种加载配置的方式

一种是基于命令行参数，比如上面的--master，spark-submit可以通过--conf参数，接收所有spark属性

另一种是从conf/spark-defaults.conf文件中加载，其中每一行都包括了一个key和value

比如spark.executor.memory 4g

所有在SparkConf、spark-submit和spark-defaults.conf中配置的属性，在运行的时候，都会被综合使用

直接通过SparkConf设置的属性，优先级是最高的，会覆盖其余两种方式设置的属性

其次是spark-submit脚本中通过--conf设置的属性

最后是spark-defaults.conf中设置的属性

通常来说，如果你要对所有的spark作业都生效的配置，放在spark-defaults.conf文件中，只要将spark-defaults.conf.template拷贝成那个文，然后在其中编辑即可

然后呢，对于某个spark作业比较特殊的配置，推荐放在spark-submit脚本中，用--conf配置，比较灵活

SparkConf配置属性，有什么用呢？也有用，在eclipse中用local模式执行运行的时候，那你就只能在SparkConf中设置属性了

这里还有一种特例，就是说，在新的spark版本中，可能会将一些属性的名称改变，那些旧的属性名称就变成过期的了

此时旧的属性名称还是会被接受的，但是新的属性名称会覆盖掉旧的属性名称，并且优先级是比旧属性名称更高的

举例来说

shuffle reduce read操作的内存缓冲块儿

spark 1.3.0: spark.reducer.maxMbInFlight

spark 1.5.0: spark.reducer.maxSizeInFlight

## 165\_Spark核心编程进阶-spark-submit配置第三方依赖

高级依赖管理

使用spark-submit脚本提交spark application时，application jar，还有我们使用--jars命令绑定的其他jar，都会自动被发送到集群上去

spark支持以下几种URL来指定关联的其他jar

file: 是由driver的http文件服务提供支持的，所有的executor都会通过driver的HTTP服务来拉取文件

hdfs:，http:，https:，ftp:，这种文件，就是直接根据URI，从指定的地方去拉取，比如hdfs、或者http链接、或者ftp服务器

local: 这种格式的文件必须在每个worker节点上都要存在，所以不需要通过网络io去拉取文件，这对于特别大的文件或者jar包特别适用，可以提升作业的执行性能

--jars，比如，mysql驱动包，或者是其他的一些包

文件和jar都会被拷贝到每个executor的工作目录中，这就会占用很大一片磁盘空间，因此需要在之后清理掉这些文件

在yarn上运行spark作业时，依赖文件的清理都是自动进行的

适用standalone模式，需要配置spark.worker.cleanup.appDataTtl属性，来开启自动清理依赖文件和jar包

用户还可以通过在spark-submit中，使用--packages，绑定一些maven的依赖包

此外，还可以通过--repositories来绑定过一些额外的仓库

但是说实话，这两种情况还的确不太常见

--files，比如，最典型的就是hive-site.xml配置文件

## 166\_Spark核心编程进阶-spark算子的闭包原理详解

Spark中一个非常难以理解的概念，就是在集群中分布式并行运行时操作的算子外部的变量的生命周期

通常来说，这个问题跟在RDD的算子中操作作用域外部的变量有关

所谓RDD算子中，操作作用域外部的变量，指的是，类似下面的语句:

val a = 0;

rdd.foreach(i -> a += i)

此时，对rdd执行的foreach算子的作用域，其实仅仅是它的内部代码，但是这里却操作了作用域外部的a变量

根据不同的编程语言的语法，这种功能是可以做到的，而这种现象就叫做闭包

闭包简单来说，就是**操作的不属于一个作用域范围的变量**

如果使用local模式运行spark作业，那么实际只有一个jvm进程在执行这个作业

此时，你所有的RDD算子的代码执行以及它们操作的外部变量，都是在一个进程的内存中，这个进程就是driver进程

此时是没有任何问题的

但是在作业提交到集群执行的模式下（无论是client或cluster模式，作业都是在集群中运行的）

为了分布式并行执行你的作业，spark会将你的RDD算子操作，分散成多个task，放到集群中的多个节点上的executor进程中去执行

每个task执行的是相同的代码，但是却是处理不同的数据

在提交作业的task到集群去执行之前，spark会先在driver端处理闭包

spark中的闭包，特指那些，不在算子的作用域内部，但是在作用域外部却被算子处理和操作了的变量

而算子代码的执行也需要这些变量才能顺利执行

此时，这些闭包变量会被序列化成多个副本，然后每个副本都发送到各个executor进程中，供那个executor进程运行的task执行代码时使用

对于上面说的闭包变量处理机制

对于local模式，没有任何特别的影响，毕竟都在一个jvm进程中，变量发送到executor，也不过就是进程中的一个线程而已

但是对于集群运行模式来说，每个executor进程，都会得到一个闭包变量的副本，这个时候，就会出问题

因此闭包变量发送到executor进程中之后，就变成了一个一个独立的变量副本了，这就是最关键的一点

此时在executor进程中，执行task和算子代码时，访问的闭包变量，也仅仅只是当前executor进程中的一个变量副本而已了

此时虽然在driver进程中，也有一个变量副本，但是却完全跟各个executor进程中的变量副本不是一个东西

此时，各个executor进程对于自己内存中的变量副本进行操作，即使改变了变量副本的值，但是对于driver端的程序，是完全感知不到的

driver端的变量没有被进行任何操作

因此综上所述，**在你使用集群模式运行作业的时候，切忌不要在算子内部，对作用域外面的闭包变量进行改变其值的操作**

因为那没有任何意义，算子仅仅会在executor进程中，改变变量副本的值

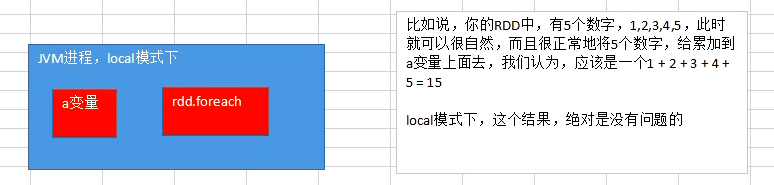
对于driver端的变量没有任何影响，我们也获取不到executor端的变量副本的值

如果希望在集群模式下，对某个driver端的变量，进行分布式并行地全局性的修改

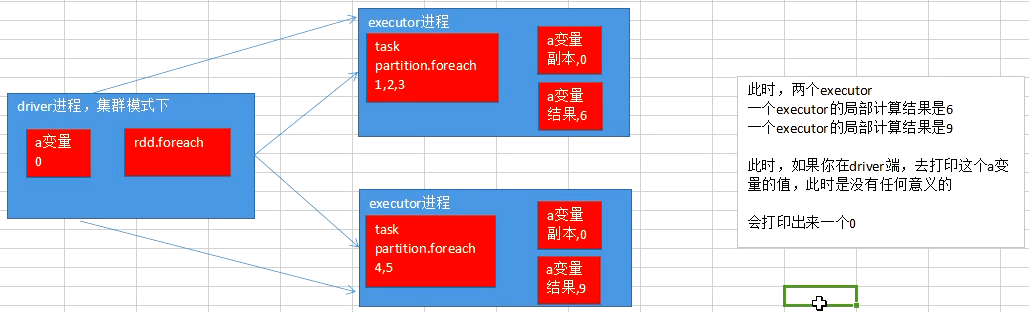
可以使用Spark提供的Accumulator，全局累加器

后面我们会讲解一个Accumulator的高级用法，自定义Accumulator，实现任意机制和算法的全局计算器

闭包在local模式下，没有问题，结果输出正确：



闭包在集群模式下，存在问题：



## 167\_Spark核心编程进阶-实验：对闭包变量进行累加操作的无效现象



## 168\_Spark核心编程进阶-实验：在算子内打印数据的无法看到现象

## 169\_Spark核心编程进阶-mapPartitions以及学生成绩查询案例

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.HashMap;  import java.util.Iterator;  import java.util.List;  import java.util.Map;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.FlatMapFunction;  public class MapPartitions {    public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("MapPartitions")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // 准备一下模拟数据  List<String> studentNames = Arrays.asList("张三", "李四", "王二", "麻子");  JavaRDD<String> studentNamesRDD = sc.parallelize(studentNames, 2);    final Map<String, Double> studentScoreMap = new HashMap<String, Double>();  studentScoreMap.put("张三", 278.5);  studentScoreMap.put("李四", 290.0);  studentScoreMap.put("王二", 301.0);  studentScoreMap.put("麻子", 205.0);    **// mapPartitions**  **// 类似map，不同之处在于，map算子，一次就处理一个partition中的一条数据**  **// mapPartitions算子，一次处理一个partition中所有的数据**    **// 推荐的使用场景**  **// 如果你的RDD的数据量不是特别大，那么建议采用mapPartitions算子替代map算子，可以加快处理速度**  **// 但是如果你的RDD的数据量特别大，比如说10亿，不建议用mapPartitions，可能会内存溢出**    JavaRDD<Double> studentScoresRDD = studentNamesRDD.mapPartitions(    new FlatMapFunction<Iterator<String>, Double>() {  private static final long serialVersionUID = 1L;    @Override  public Iterable<Double> call(Iterator<String> iterator)  throws Exception {  // 因为算子一次处理一个partition的所有数据  // call函数接收的参数，是iterator类型，代表了partition中所有数据的迭代器  // 返回的是一个iterable类型，代表了返回多条记录，通常使用List类型    List<Double> studentScoreList = new ArrayList<Double>();    while(iterator.hasNext()) {  String studentName = iterator.next();  Double studentScore = studentScoreMap.get(studentName);  studentScoreList.add(studentScore);  }    return studentScoreList;  }    });  for(Double studentScore: studentScoresRDD.collect()) {  System.out.println(studentScore);  }  sc.close();  }  } |

## 170\_Spark核心编程进阶-mapPartitionsWithIndex以开学分班案例

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.Iterator;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function2;  public class MapPartitionsWithIndex {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("MapPartitionsWithIndex")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    List<String> studentNames = Arrays.asList("张三", "李四", "王二", "麻子");  JavaRDD<String> studentNamesRDD = sc.parallelize(studentNames, 2);    // 这里，parallelize并行集合的时候，指定了numPartitions是2  // 也就是说，四个同学，会被分成2个班  // 但是spark自己判定怎么分班    // 如果你要分班的话，就必须拿到班级号  // mapPartitionsWithIndex这个算子来做，这个算子可以拿到每个partition的index  // 也就可以作为我们的班级号    JavaRDD<String> studentWithClassRDD = studentNamesRDD.mapPartitionsWithIndex(    new Function2<Integer, Iterator<String>, Iterator<String>>() {  private static final long serialVersionUID = 1L;  @Override  public Iterator<String> call(Integer index, Iterator<String> iterator)  throws Exception {  List<String> studentWithClassList = new ArrayList<String>();    while(iterator.hasNext()) {  String studentName = iterator.next();  String studentWithClass = studentName + "\_" + (index + 1);  studentWithClassList.add(studentWithClass);  }    return studentWithClassList.iterator();  }    }, true);    for(String studentWithClass : studentWithClassRDD.collect()) {  System.out.println(studentWithClass);  }    sc.close();  }    } |

## 171\_Spark核心编程进阶-sample以及公司年会抽奖案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  public class Sample {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Sample")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);  List<String> staffList = Arrays.asList("张三", "李四", "王二", "麻子",  "赵六", "王五", "李大个", "王大妞", "小明", "小倩");  JavaRDD<String> staffRDD = sc.parallelize(staffList);  // sample算子  // 可以使用指定的比例，比如说0.1或者0.9，从RDD中随机抽取10%或者90%的数据  // 从RDD中随机抽取数据的功能  // 推荐不要设置第三个参数，feed  JavaRDD<String> luckyStaffRDD = staffRDD.sample(false, 0.1);  for(String staff : luckyStaffRDD.collect()) {  System.out.println(staff);  }  sc.close();  }  } |

## 172\_Spark核心编程进阶-union以及公司部门合并案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  public class Union {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Union")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // union算子  // 将两个RDD的数据，合并为一个RDD    List<String> department1StaffList = Arrays.asList("张三", "李四", "王二", "麻子");  JavaRDD<String> department1StaffRDD = sc.parallelize(department1StaffList);    List<String> department2StaffList = Arrays.asList("赵六", "王五", "小明", "小倩");  JavaRDD<String> department2StaffRDD = sc.parallelize(department2StaffList);    JavaRDD<String> departmentStaffRDD = department1StaffRDD.union(department2StaffRDD);    for(String staff : departmentStaffRDD.collect()) {  System.out.println(staff);  }  sc.close();  }  } |

## 173\_Spark核心编程进阶-intersection以及公司跨多项目人员查询案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  public class Intersection {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Intersection")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // intersection算子  // 获取两个rdd中，相同的数据    // 有的公司内，有些人可能同时在做不同的项目，属于不同的项目组  // 所以要针对代表两个项目组同事的rdd，取出其交集    List<String> project1MemberList = Arrays.asList("张三", "李四", "王二", "麻子");  JavaRDD<String> project1MemberRDD = sc.parallelize(project1MemberList);    List<String> project2MemberList = Arrays.asList("张三", "王五", "小明", "小倩");  JavaRDD<String> project2MemberRDD = sc.parallelize(project2MemberList);    JavaRDD<String> projectIntersectionRDD = project1MemberRDD.intersection(project2MemberRDD);  for(String member : projectIntersectionRDD.collect()) {  System.out.println(member);  }  sc.close();  }  } |

## 174\_Spark核心编程进阶-distinct以及网站uv统计案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function;  public class Distinct {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Distinct")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // distinct算子  // 对rdd中的数据进行去重    // uv统计案例  // uv：user view，每天每个用户可能对网站会点击多次  // 此时，需要对用户进行去重，然后统计出每天有多少个用户访问了网站  // 而不是所有用户访问了网站多少次（pv）    List<String> accessLogs = Arrays.asList(  "user1 2016-01-01 23:58:42",  "user1 2016-01-01 23:58:43",  "user1 2016-01-01 23:58:44",  "user2 2016-01-01 12:58:42",  "user2 2016-01-01 12:58:46",  "user3 2016-01-01 12:58:42",  "user4 2016-01-01 12:58:42",  "user5 2016-01-01 12:58:42",  "user6 2016-01-01 12:58:42",  "user6 2016-01-01 12:58:45");  JavaRDD<String> accessLogsRDD = sc.parallelize(accessLogs);    JavaRDD<String> useridsRDD = accessLogsRDD.map(new Function<String, String>() {  private static final long serialVersionUID = 1L;  @Override  public String call(String accessLog) throws Exception {  String userid = accessLog.split(" ")[0];  return userid;  }    });    JavaRDD<String> distinctUseridsRDD = useridsRDD.distinct();  int uv = distinctUseridsRDD.collect().size();  System.out.println("uv: " + uv);  sc.close();  }  } |

## 175\_Spark核心编程进阶-aggregateByKey以及单词计数案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.FlatMapFunction;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import scala.Tuple2;  public class AggregateByKey {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("AggregateByKey")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    JavaRDD<String> lines = sc.textFile(  "C://Users//Administrator//Desktop//hello.txt",  3);    JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {    private static final long serialVersionUID = 1L;    @Override  public Iterable<String> call(String line) throws Exception {  return Arrays.asList(line.split(" "));  }  });    JavaPairRDD<String, Integer> pairs = words.mapToPair(    new PairFunction<String, String, Integer>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<String, Integer> call(String word) throws Exception {  return new Tuple2<String, Integer>(word, 1);  }  });    // aggregateByKey，分为三个参数  // reduceByKey认为是aggregateByKey的简化版  // aggregateByKey最重要的一点是，多提供了一个函数，Seq Function  // 就是说自己可以控制如何对每个partition中的数据进行先聚合，类似于mapreduce中的，map-side combine  // 然后才是对所有partition中的数据进行全局聚合    // 第一个参数是，每个key的初始值  // 第二个是个函数，Seq Function，如何进行shuffle map-side的本地聚合  // 第三个是个函数，Combiner Function，如何进行shuffle reduce-side的全局聚合    JavaPairRDD<String, Integer> wordCounts = pairs.aggregateByKey(  0,    new Function2<Integer, Integer, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Integer call(Integer v1, Integer v2)  throws Exception {  return v1 + v2;  }  },    new Function2<Integer, Integer, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Integer call(Integer v1, Integer v2)  throws Exception {  return v1 + v2;  }    });    List<Tuple2<String, Integer>> wordCountList = wordCounts.collect();  for(Tuple2<String, Integer> wordCount : wordCountList) {  System.out.println(wordCount);  }  sc.close();  }  } |

## 176\_Spark核心编程进阶-cartesian以及服装搭配案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import scala.Tuple2;  public class Cartesian {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Cartesian")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // cartesian，中文名，笛卡尔乘积  // 比如说两个RDD，分别有10条数据，用了cartesian算子以后  // 两个RDD的每一条数据都会和另外一个RDD的每一条数据执行一次join  // 最终组成了一个笛卡尔乘积    // 小案例  // 比如说，现在5件衣服，5条裤子，分别属于两个RDD  // 就是说，需要对每件衣服都和每天裤子做一次join，尝试进行服装搭配    List<String> clothes = Arrays.asList("夹克", "T恤", "皮衣", "风衣");  JavaRDD<String> clothesRDD = sc.parallelize(clothes);    List<String> trousers = Arrays.asList("皮裤", "运动裤", "牛仔裤", "休闲裤");  JavaRDD<String> trousersRDD = sc.parallelize(trousers);    JavaPairRDD<String, String> pairsRDD = clothesRDD.cartesian(trousersRDD);  for(Tuple2<String, String> pair : pairsRDD.collect()) {  System.out.println(pair);  }  sc.close();  }  } |

## 177\_Spark核心编程进阶-coalesce以及公司部门整合案例

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.Iterator;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function2;  public class Coalesce {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Coalesce")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // coalesce算子，功能是将RDD的partition缩减，减少  // 将一定量的数据，压缩到更少的partition中去    // 建议的使用场景，配合filter算子使用  // 使用filter算子过滤掉很多数据以后，比如30%的数据，出现了很多partition中的数据不均匀的情况  // 此时建议使用coalesce算子，压缩rdd的partition数量  // 从而让各个partition中的数据都更加的紧凑    // 公司原先有6个部门  // 但是呢，不巧，碰到了公司裁员，裁员以后呢，有的部门中的人员就没了  // 不同的部分人员不均匀  // 此时呢，做一个部门整合的操作，将不同的部门的员工进行压缩    List<String> staffList = Arrays.asList("张三", "李四", "王二", "麻子",  "赵六", "王五", "李大个", "王大妞", "小明", "小倩");  JavaRDD<String> staffRDD = sc.parallelize(staffList, 6);    JavaRDD<String> staffRDD2 = staffRDD.mapPartitionsWithIndex(    new Function2<Integer, Iterator<String>, Iterator<String>>() {  private static final long serialVersionUID = 1L;    @Override  public Iterator<String> call(Integer index, Iterator<String> iterator)  throws Exception {  List<String> list = new ArrayList<String>();    while(iterator.hasNext()) {  String staff = iterator.next();  list.add("部门[" + (index + 1) + "], " + staff);  }    return list.iterator();  }    }, true);    for(String staffInfo : staffRDD2.collect()) {  System.out.println(staffInfo);  }    JavaRDD<String> staffRDD3 = staffRDD2.coalesce(3);    JavaRDD<String> staffRDD4 = staffRDD3.mapPartitionsWithIndex(    new Function2<Integer, Iterator<String>, Iterator<String>>() {  private static final long serialVersionUID = 1L;    @Override  public Iterator<String> call(Integer index, Iterator<String> iterator)  throws Exception {  List<String> list = new ArrayList<String>();    while(iterator.hasNext()) {  String staff = iterator.next();  list.add("部门[" + (index + 1) + "], " + staff);  }    return list.iterator();  }    }, true);    for(String staffInfo : staffRDD4.collect()) {  System.out.println(staffInfo);  }    sc.close();  }  } |

## 178\_Spark核心编程进阶-repartition以及公司新增部门案例

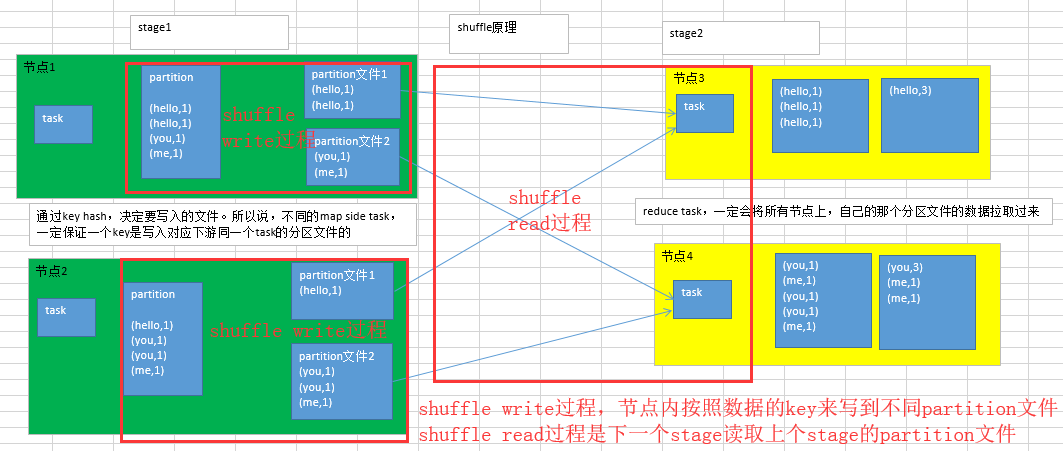
|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.Iterator;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function2;  public class Repartition {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Repartition")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // repartition算子，用于任意将rdd的partition增多，或者减少  // 与coalesce不同之处在于，coalesce仅仅能将rdd的partition变少  // 但是repartition可以将rdd的partiton变多    // 建议使用的场景  // 一个很经典的场景，使用Spark SQL从hive中查询数据时  // Spark SQL会根据hive对应的hdfs文件的block数量还决定加载出来的数据rdd有多少个partition  // 这里的partition数量，是我们根本无法设置的    // 有些时候，可能它自动设置的partition数量过于少了，导致我们后面的算子的运行特别慢  // 此时就可以在Spark SQL加载hive数据到rdd中以后  // 立即使用repartition算子，将rdd的partition数量变多    // 案例  // 公司要增加新部门  // 但是人员还是这么多，所以我们只能使用repartition操作，增加部门  // 将人员平均分配到更多的部门中去    List<String> staffList = Arrays.asList("张三", "李四", "王二", "麻子",  "赵六", "王五", "李大个", "王大妞", "小明", "小倩");  JavaRDD<String> staffRDD = sc.parallelize(staffList, 3);    JavaRDD<String> staffRDD2 = staffRDD.mapPartitionsWithIndex(    new Function2<Integer, Iterator<String>, Iterator<String>>() {  private static final long serialVersionUID = 1L;    @Override  public Iterator<String> call(Integer index, Iterator<String> iterator)  throws Exception {  List<String> list = new ArrayList<String>();    while(iterator.hasNext()) {  String staff = iterator.next();  list.add("部门[" + (index + 1) + "], " + staff);  }    return list.iterator();  }    }, true);    for(String staffInfo : staffRDD2.collect()) {  System.out.println(staffInfo);  }    JavaRDD<String> staffRDD3 = staffRDD2.repartition(6);    JavaRDD<String> staffRDD4 = staffRDD3.mapPartitionsWithIndex(    new Function2<Integer, Iterator<String>, Iterator<String>>() {  private static final long serialVersionUID = 1L;    @Override  public Iterator<String> call(Integer index, Iterator<String> iterator)  throws Exception {  List<String> list = new ArrayList<String>();    while(iterator.hasNext()) {  String staff = iterator.next();  list.add("部门[" + (index + 1) + "], " + staff);  }    return list.iterator();  }    }, true);  for(String staffInfo : staffRDD4.collect()) {  System.out.println(staffInfo);  }  sc.close();  }  } |

## 179\_Spark核心编程进阶-takeSample以及公司年会抽奖案例

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  public class TakeSampled {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setAppName("Sample")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    List<String> staffList = Arrays.asList("张三", "李四", "王二", "麻子",  "赵六", "王五", "李大个", "王大妞", "小明", "小倩");  JavaRDD<String> staffRDD = sc.parallelize(staffList);    // takeSample算子  // 与sample不同之处，两点  // 1、action操作，sample是transformation操作  // 2、不能指定抽取比例，只能是抽取几个    List<String> luckyStaffList = staffRDD.takeSample(false, 3);    for(String luckyStaff : luckyStaffList) {  System.out.println(luckyStaff);  }  sc.close();  }  } |

## 180\_Spark核心编程进阶-shuffle操作原理详解

|  |
| --- |
| shuffle操作，是spark中一些特殊的算子操作会触发的一种操作  shuffle操作，会导致大量的数据在不同的机器和节点之间进行传输，因此也是spark中最复杂、最消耗性能的一种操作  我们可以通过reduceByKey操作作为一个例子，来理解shuffle操作  reduceByKey算子会将上一个RDD中的每个key对应的所有value都聚合成一个value，然后生成一个新的RDD  新的RDD的元素类型就是<key,value>对的格式，每个key对应一个聚合起来的value  这里最大的问题就在于，对于上一个RDD来说，并不是一个key对应的所有value都是在一个partition中的，也更不太可能说key的所有value都在一台机器上  所以对于这种情况来说，就必须在整个集群中，将各个节点上，同一个key对应的values，统一传输到一个节点上来聚合处理  这个过程中就会发生大量的网络数据的传输  在进行一个key对应的values的聚合时  首先，上一个stage的每个map task就必须保证将自己处理的当前分区中的数据，相同的key写入一个分区文件中，可能会写多个不同的分区文件  接着下一个stage的reduce task就必须从上一个stage所有task所在的机器上，将各个task写入的多个分区文件中，找到属于自己的那个分区文件  接着将属于自己的分区数据，拉取过来，这样就可以保证每个key对应的所有values都汇聚到一个节点上去处理和聚合  这个过程就称之为shuffle  shuffle是分为shuffle write和shuffle read两个部分的  是在两个不同的stage中进行的 |



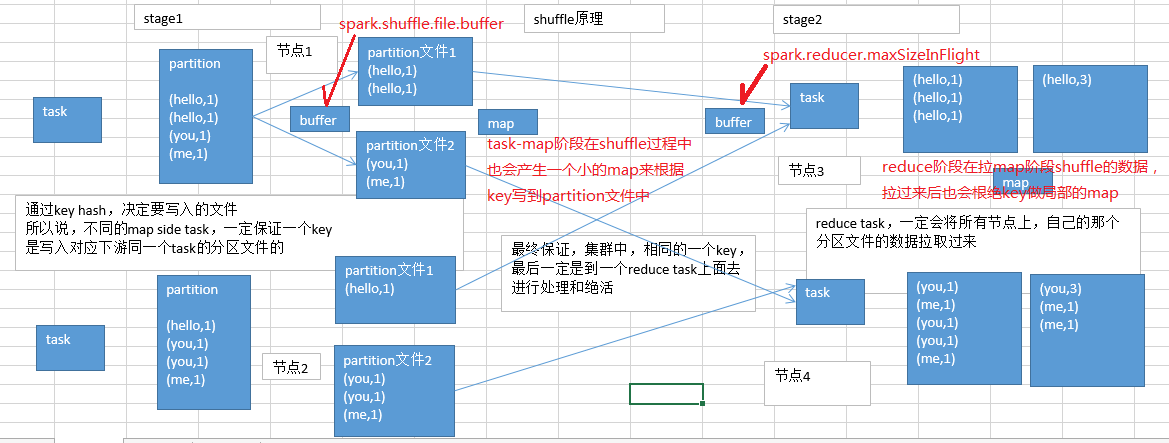
## 181\_Spark核心编程进阶-shuffle操作过程中进行数据排序

|  |
| --- |
| 默认情况下，shuffle操作是不会对每个分区中的数据进行排序的  如果想要对每个分区中的数据进行排序，那么可以使用以下三种方法：  1、使用mapPartitions算子处理每个partition，对每个partition中的数据进行排序  2、使用repartitionAndSortWithinPartitions，对RDD进行重分区，在重分区的过程中同时就进行分区内数据的排序  3、使用sortByKey对数据进行全局排序  上述三种方法中，相对来说，mapPartitions的代价比较小，因为不需要进行额外的shuffle操作  repartitionAndSortWithinPartitions和sortByKey可能会进行额外的shuffle操作的，因此性能并不是很高  val rdd2 = rdd1.reduceByKey(\_ + \_)  rdd2.mapPartitions(tuples.sort)  rdd2.repartitionAndSortWithinPartitions()，重分区，重分区的过程中，就进行分区内的key的排序，重分区的原理和repartition一样  rdd2.sortByKey，直接对rdd按照key进行全局性的排序 |

## 182\_Spark核心编程进阶-会触发shuffle操作的算子

|  |
| --- |
| spark中会导致shuffle操作的有以下几种算子  1、repartition类的操作：比如repartition、repartitionAndSortWithinPartitions、coalesce等  2、byKey类的操作：比如reduceByKey、groupByKey、sortByKey等  3、join类的操作：比如join、cogroup等  重分区: 一般会shuffle，因为需要在整个集群中，对之前所有的分区的数据进行随机，均匀的打乱，然后把数据放入下游新的指定数量的分区内  byKey类的操作：因为你要对一个key，进行聚合操作，那么肯定要保证集群中，所有节点上的，相同的key，一定是到同一个节点上进行处理  join类的操作：两个rdd进行join，就必须将相同join key的数据，shuffle到同一个节点上，然后进行相同key的两个rdd数据的笛卡尔乘积  提醒一下  所以对于上述的操作  首先第一原则，就是，能不用shuffle操作，就尽量不用shuffle操作，尽量使用不shuffle的操作  第二原则，就是，如果使用了shuffle操作，那么肯定要进行shuffle的调优，甚至是解决碰到的数据倾斜的问题 |

## 183\_Spark核心编程进阶-shuffle操作对性能消耗的原理详解



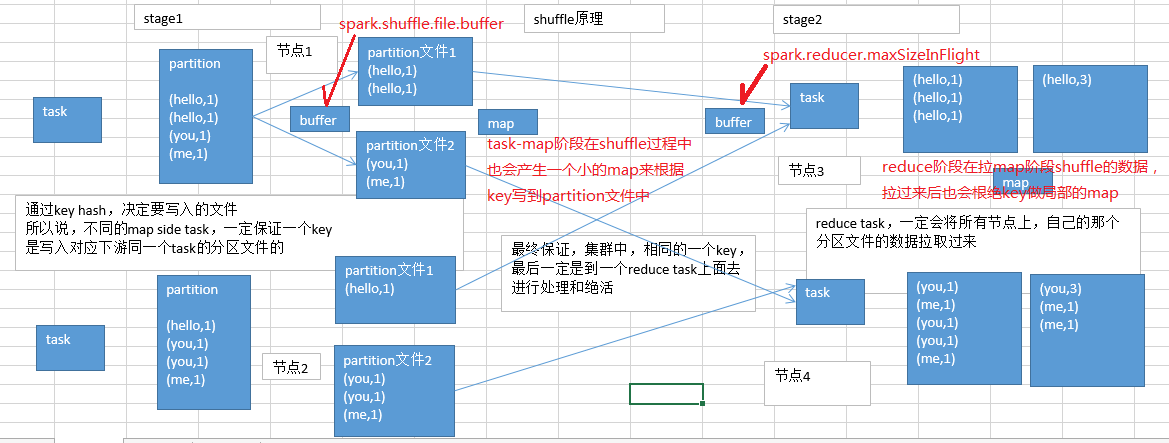
|  |
| --- |
| shuffle操作是spark中唯一最最消耗性能的地方  因此也就成了最需要进行性能调优的地方，最需要解决线上报错的地方，也是唯一可能出现数据倾斜的地方  因为shuffle过程中，会产生大量的磁盘IO、数据序列化和反序列化、网络IO  为了实施shuffle操作  spark中才有了stage的概念，在发生shuffle操作的算子中，进行stage的拆分  shuffle操作的前半部分，是上一个stage来进行，也称之为map task，shuffle操作的后半部分，是下一个stage来进行，也称之为reduce task  其中map task负责数据的组织，也就是将同一个key对应的value都写入同一个下游task对应的分区文件中  其中reduce task负责数据的聚合，也就是将上一个stage的task所在节点上，将属于自己的各个分区文件，都拉取过来聚合  这种模型，是参考和模拟了MapReduce的shuffle过程来的  map task会将数据先保存在内存中，如果内存不够时，就溢写到磁盘文件中去  reduce task会读取各个节点上属于自己的分区磁盘文件，到自己节点的内存中，并进行聚合  shuffle操作会消耗大量的内存，因为无论是网络传输数据之前，还是之后，都会使用大量的内存中数据结构来实施聚合操作  比如reduceByKey和aggregateByKey操作，会在map side使用内存中的数据结构进行预先聚合  其他的byKey类的操作，都是在reduce side，使用内存数据结构进行聚合  在聚合过程中，如果内存不够，只能溢写到磁盘文件中去，此时就会发生大量的磁盘IO，降低性能  此外，shuffle过程中，还会产生大量的中间文件，也就是map side写入的大量分区文件  比如Spark 1.3版本，这些中间文件会一致保留着，直到RDD不再被使用，而且被垃圾回收掉了，才会去清理中间文件  这主要是为了，如果要重新计算shuffle后的RDD，那么map side不需要重新做一次磁盘写操作  但是这种情况下，如果我们的应用程序中，一直保持着对RDD的引用，导致很长时间以后才会进行RDD垃圾回收操作  保存中间文件的目录，由spark.local.dir属性指定  内存的消耗、磁盘IO、网络数据传输（IO） |

## 184\_Spark核心编程进阶-shuffle操作所有相关参数详解以及性能调优

我们可以通过对一系列的参数进行调优，来优化shuffle的性能

spark 1.5.2版本

|  |  |  |
| --- | --- | --- |
| **属性名称** | **默认值** | **属性说明** |
| spark.reducer.maxSizeInFlight | 48m | reduce task的buffer缓冲，代表了每个reduce task每次能够拉取的map side数据最大大小，如果内存充足，可以考虑加大大小，从而减少网络传输次数，提升性能 |
| spark.shuffle.blockTransferService | netty | shuffle过程中，传输数据的方式，两种选项，netty或nio，spark 1.2开始，默认就是netty，比较简单而且性能较高，spark 1.5开始nio就是过期的了，而且spark 1.6中会去除掉 |
| spark.shuffle.compress | TRUE | 是否对map side输出的文件进行压缩，默认是启用压缩的，压缩器是由spark.io.compression.codec属性指定的，默认是snappy压缩器，该压缩器强调的是压缩速度，而不是压缩率 |
| spark.shuffle.consolidateFiles | FALSE | 默认为false，如果设置为true，那么就会合并map side输出文件，对于reduce task数量特别的情况下，可以极大减少磁盘IO开销，提升性能 |
| spark.shuffle.file.buffer | 32k | map side task的内存buffer大小，写数据到磁盘文件之前，会先保存在缓冲中，如果内存充足，可以适当加大大小，从而减少map side磁盘IO次数，提升性能 |
| spark.shuffle.io.maxRetries | 3 | 网络传输数据过程中，如果出现了网络IO异常，重试拉取数据的次数，默认是3次，对于耗时的shuffle操作，建议加大次数，以避免full gc或者网络不通常导致的数据拉取失败，进而导致task lost，增加shuffle操作的稳定性,在比较大的项目中作者一般调到60以及时间间隔也是60 |
| spark.shuffle.io.retryWait | 5s | 每次重试拉取数据的等待间隔，默认是5s，建议加大时长，理由同上，保证shuffle操作的稳定性，在比较大的项目中作者一般直接60s |
| spark.shuffle.io.numConnectionsPerPeer | 1 | 机器之间的可以重用的网络连接，主要用于在大型集群中减小网络连接的建立开销，如果一个集群的机器并不多，可以考虑增加这个值 |
| spark.shuffle.io.preferDirectBufs | TRUE | 启用堆外内存，可以避免shuffle过程的频繁gc，如果堆外内存非常紧张，则可以考虑关闭这个选项 |
| spark.shuffle.manager | sort | ShuffleManager，Spark 1.5以后，有三种可选的，hash、sort和tungsten-sort，sort-based ShuffleManager会更高效实用内存，并且避免产生大量的map side磁盘文件，从Spark 1.2开始就是默认的选项，tungsten-sort与sort类似，但是内存性能更高 |
| spark.shuffle.memoryFraction | 0.2 | 如果spark.shuffle.spill属性为true，那么该选项生效，代表了executor内存中，用于进行shuffle reduce side聚合的内存比例，默认是20%，如果内存充足，建议调高这个比例，给reduce聚合更多内存，避免内存不足频繁读写磁盘 |
| spark.shuffle.service.enabled | FALSE | 启用外部shuffle服务，这个服务会安全地保存shuffle过程中，executor写的磁盘文件，因此executor即使挂掉也不要紧，必须配合spark.dynamicAllocation.enabled属性设置为true，才能生效，而且外部shuffle服务必须进行安装和启动，才能启用这个属性 |
| spark.shuffle.service.port | 7337 | 外部shuffle服务的端口号，具体解释同上 |
| spark.shuffle.sort.bypassMergeThreshold | 200 | 对于sort-based ShuffleManager，如果没有进行map side聚合，而且reduce task数量少于这个值，那么就不会进行排序，如果你使用sort ShuffleManager，而且不需要排序，那么可以考虑将这个值加大，直到比你指定的所有task数量都打，以避免进行额外的sort，从而提升性能 |
| spark.shuffle.spill | TRUE | 当reduce side的聚合内存使用量超过了spark.shuffle.memoryFraction指定的比例时，就进行磁盘的溢写操作 |
| spark.shuffle.spill.compress | TRUE | 同上，进行磁盘溢写时，是否进行文件压缩，使用spark.io.compression.codec属性指定的压缩器，默认是snappy，速度优先 |



shuffle有三种：hash、sort和tungsten-sort，默认是sort，如果不需要排序就可以用hash，tungsten-sort与sort类似，但是内存性能更高

## 185\_Spark核心编程进阶-综合案例1：移动端app访问流量日志分析

|  |
| --- |
| 如果你是在一个互联网公司，然后你的公司现在也在做移动互联网，做了一个手机app  那么你的手机app的用户，每次进行点击，或者是一些搜索操作的时候，都会跟你的远程的后端服务器做一次交互  也就是说，你的手机app，首先会往后端服务器发送一个请求，然后你的后端服务器会给你的手机app返回一个响应，响应的内容可能是图片、或者文字、或者json  此时，就完成了一次你的移动端app和后端服务器之间的交互过程  通常来说，在你的移动端app访问你的后端服务器的时候，你的后端服务器会记录一条日志  这个日志，也就是你的移动端app访问流量的相关日志，但是也可以根据你自己的需要，移动端发送一条日志过来，服务器端的web系统保存日志  我们这里做的就是最基本的，记录你的移动端app和服务器之间的上行数据包和下行数据包，上行流量和下行流量  我们要来计算，就是说，你的每个移动端，唯一的一个标识是你的deviceID  然后呢，每条日志，都会有这一次请求和响应的上行流量和下行流量的记录，这里呢，上行流量指的是手机app向服务器发送的请求数据的流量  下行流量，认为是服务器端给手机app返回的数据（比如说图片、文字、json）的流量  每个设备（deviceID），总上行流量和总下行流量，计算之后，要根据上行流量和下行流量进行排序，需要进行倒序排序  获取流量最大的前10个设备  难点：根据上行流量和下行流量进行排序的时候，不是简单的排序，优先根据上行流量进行排序，如果上行流量相等，那么根据下行流量排序  二次排序 |

## 186\_Spark核心编程进阶-综合案例1：日志文件格式分析



|  |
| --- |
|  |

## 187\_Spark核心编程进阶-综合案例1：读取日志文件并创建RDD

AppLogSpark.java

|  |
| --- |
| import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  /\*\*  \* 移动端app访问流量日志分析案例  \* @author Administrator  \*  \*/  public class AppLogSpark {  public static void main(String[] args) throws Exception {  // 创建Spark配置和上下文对象  SparkConf conf = new SparkConf()  .setAppName("AppLogSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // 读取日志文件，并创建一个RDD  // 使用SparkContext的textFile()方法，即可读取本地磁盘文件，或者是HDFS上的文件  // 创建出来一个初始的RDD，其中包含了日志文件中的所有数据  JavaRDD<String> accessLogRDD = sc.textFile(  "C://Users//Administrator//Desktop//access.log");    // 关闭Spark上下文  sc.close();  }    } |

DataGenerator.java

|  |
| --- |
| import java.io.FileOutputStream;  import java.io.OutputStreamWriter;  import java.io.PrintWriter;  import java.util.ArrayList;  import java.util.Calendar;  import java.util.Date;  import java.util.List;  import java.util.Random;  import java.util.UUID;  public class DataGenerator {    public static void main(String[] args) throws Exception {  Random random = new Random();    // 生成100个deviceID  List<String> deviceIDs = new ArrayList<String>();  for(int i = 0; i < 100; i++) {  deviceIDs.add(getRandomUUID());  }    StringBuffer buffer = new StringBuffer("");    for(int i = 0; i < 1000; i++) {  // 生成随机时间戳  Calendar cal = Calendar.getInstance();  cal.setTime(new Date());  cal.add(Calendar.MINUTE, -random.nextInt(600));  long timestamp = cal.getTime().getTime();    // 生成随机deviceID  String deviceID = deviceIDs.get(random.nextInt(100));    // 生成随机的上行流量  long upTraffic = random.nextInt(100000);  // 生成随机的下行流量  long downTraffic = random.nextInt(100000);    buffer.append(timestamp).append("\t")  .append(deviceID).append("\t")  .append(upTraffic).append("\t")  .append(downTraffic).append("\n");  }    PrintWriter pw = null;  try {  pw = new PrintWriter(new OutputStreamWriter(  new FileOutputStream("C:\\Users\\Administrator\\Desktop\\access.log")));  pw.write(buffer.toString());  } catch (Exception e) {  e.printStackTrace();  } finally {  pw.close();  }  }  private static String getRandomUUID() {  return UUID.randomUUID().toString().replace("-", "");  }  } |

## 188\_Spark核心编程进阶-综合案例1：创建自定义的可序列化类

|  |
| --- |
| import java.io.Serializable;  /\*\*  \* 访问日志信息类（可序列化）  \*/  public class AccessLogInfo implements Serializable {  private static final long serialVersionUID = 5749943279909593929L;    private long timestamp; // 时间戳  private long upTraffic; // 上行流量  private long downTraffic; // 下行流量    public AccessLogInfo() {}    public AccessLogInfo(long timestamp, long upTraffic, long downTraffic) {  this.timestamp = timestamp;  this.upTraffic = upTraffic;  this.downTraffic = downTraffic;  }    public long getTimestamp() {  return timestamp;  }  public void setTimestamp(long timestamp) {  this.timestamp = timestamp;  }  public long getUpTraffic() {  return upTraffic;  }  public void setUpTraffic(long upTraffic) {  this.upTraffic = upTraffic;  }  public long getDownTraffic() {  return downTraffic;  }  public void setDownTraffic(long downTraffic) {  this.downTraffic = downTraffic;  }  } |

## 189\_Spark核心编程进阶-综合案例1：将RDD映射为key-value格式

|  |
| --- |
| import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.PairFunction;  import scala.Tuple2;  /\*\*  \* 移动端app访问流量日志分析案例  \* @author Administrator  \*  \*/  public class AppLogSpark {  public static void main(String[] args) throws Exception {  // 创建Spark配置和上下文对象  SparkConf conf = new SparkConf()  .setAppName("AppLogSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // 读取日志文件，并创建一个RDD  // 使用SparkContext的textFile()方法，即可读取本地磁盘文件，或者是HDFS上的文件  // 创建出来一个初始的RDD，其中包含了日志文件中的所有数据  JavaRDD<String> accessLogRDD = sc.textFile(  "C://Users//Administrator//Desktop//access.log");    // 将RDD映射为key-value格式，为后面的reduceByKey聚合做准备  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD =  mapAccessLogRDD2Pair(accessLogRDD);    // 关闭Spark上下文  sc.close();  }    /\*\*  \* 将日志RDD映射为key-value的格式  \* @param accessLogRDD 日志RDD  \* @return key-value格式RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> mapAccessLogRDD2Pair(  JavaRDD<String> accessLogRDD) {  return accessLogRDD.mapToPair(new PairFunction<String, String, AccessLogInfo>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, AccessLogInfo> call(String accessLog)  throws Exception {  // 根据\t对日志进行切分  String[] accessLogSplited = accessLog.split("\t");    // 获取四个字段  long timestamp = Long.valueOf(accessLogSplited[0]);  String deviceID = accessLogSplited[1];  long upTraffic = Long.valueOf(accessLogSplited[2]);  long downTraffic = Long.valueOf(accessLogSplited[3]);    // 将时间戳、上行流量、下行流量，封装为自定义的可序列化对象  AccessLogInfo accessLogInfo = new AccessLogInfo(timestamp,  upTraffic, downTraffic);    return new Tuple2<String, AccessLogInfo>(deviceID, accessLogInfo);  }    });  }  } |

## 190\_Spark核心编程进阶-综合案例1：基于deviceID进行聚合操作

|  |
| --- |
| import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import scala.Tuple2;  /\*\*  \* 移动端app访问流量日志分析案例  \*/  public class AppLogSpark {  public static void main(String[] args) throws Exception {  // 创建Spark配置和上下文对象  SparkConf conf = new SparkConf()  .setAppName("AppLogSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // 读取日志文件，并创建一个RDD  // 使用SparkContext的textFile()方法，即可读取本地磁盘文件，或者是HDFS上的文件  // 创建出来一个初始的RDD，其中包含了日志文件中的所有数据  JavaRDD<String> accessLogRDD = sc.textFile(  "C://Users//Administrator//Desktop//access.log");    // 将RDD映射为key-value格式，为后面的reduceByKey聚合做准备  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD =  mapAccessLogRDD2Pair(accessLogRDD);    // 根据deviceID进行聚合操作  // 获取每个deviceID的总上行流量、总下行流量、最早访问时间戳  JavaPairRDD<String, AccessLogInfo> aggrAccessLogPairRDD =  aggregateByDeviceID(accessLogPairRDD);    // 关闭Spark上下文  sc.close();  }    /\*\*  \* 将日志RDD映射为key-value的格式  \* @param accessLogRDD 日志RDD  \* @return key-value格式RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> mapAccessLogRDD2Pair(  JavaRDD<String> accessLogRDD) {  return accessLogRDD.mapToPair(new PairFunction<String, String, AccessLogInfo>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, AccessLogInfo> call(String accessLog)  throws Exception {  // 根据\t对日志进行切分  String[] accessLogSplited = accessLog.split("\t");    // 获取四个字段  long timestamp = Long.valueOf(accessLogSplited[0]);  String deviceID = accessLogSplited[1];  long upTraffic = Long.valueOf(accessLogSplited[2]);  long downTraffic = Long.valueOf(accessLogSplited[3]);    // 将时间戳、上行流量、下行流量，封装为自定义的可序列化对象  AccessLogInfo accessLogInfo = new AccessLogInfo(timestamp,  upTraffic, downTraffic);    return new Tuple2<String, AccessLogInfo>(deviceID, accessLogInfo);  }    });  }    /\*\*  \* 根据deviceID进行聚合操作  \* 计算出每个deviceID的总上行流量、总下行流量以及最早访问时间  \* @param accessLogPairRDD 日志key-value格式RDD  \* @return 按deviceID聚合RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> aggregateByDeviceID(  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD) {  return accessLogPairRDD.reduceByKey(new Function2<AccessLogInfo, AccessLogInfo, AccessLogInfo>() {    private static final long serialVersionUID = 1L;    @Override  public AccessLogInfo call(AccessLogInfo accessLogInfo1, AccessLogInfo accessLogInfo2)  throws Exception {  long timestamp = accessLogInfo1.getTimestamp() < accessLogInfo2.getTimestamp() ?  accessLogInfo1.getTimestamp() : accessLogInfo2.getTimestamp();  long upTraffic = accessLogInfo1.getUpTraffic() + accessLogInfo2.getUpTraffic();  long downTraffic = accessLogInfo1.getDownTraffic() + accessLogInfo2.getDownTraffic();    AccessLogInfo accessLogInfo = new AccessLogInfo();  accessLogInfo.setTimestamp(timestamp);  accessLogInfo.setUpTraffic(upTraffic);  accessLogInfo.setDownTraffic(downTraffic);    return accessLogInfo;  }    });  }  } |

## 191\_Spark核心编程进阶-综合案例1：自定义二次排序key类

|  |
| --- |
| import java.io.Serializable;  import scala.math.Ordered;  /\*\*  \* 日志的二次排序key  \*/  public class AccessLogSortKey implements Ordered<AccessLogSortKey>, Serializable {  private static final long serialVersionUID = 3702442700882342403L;    private long upTraffic;  private long downTraffic;  private long timestamp;    @Override  public boolean $greater(AccessLogSortKey other) {  if(upTraffic > other.upTraffic) {  return true;  } else if(upTraffic == other.upTraffic &&  downTraffic > other.downTraffic) {  return true;  } else if(upTraffic == other.upTraffic &&  downTraffic == other.downTraffic &&  timestamp > other.timestamp) {  return true;  }  return false;  }  @Override  public boolean $greater$eq(AccessLogSortKey other) {  if($greater(other)) {  return true;  } else if(upTraffic == other.upTraffic &&  downTraffic == other.downTraffic &&  timestamp == other.timestamp) {  return true;  }  return false;  }    @Override  public boolean $less(AccessLogSortKey other) {  if(upTraffic < other.upTraffic) {  return true;  } else if(upTraffic == other.upTraffic &&  downTraffic < other.downTraffic) {  return true;  } else if(upTraffic == other.upTraffic &&  downTraffic == other.downTraffic &&  timestamp < other.timestamp) {  return true;  }  return false;  }  @Override  public boolean $less$eq(AccessLogSortKey other) {  if($less(other)) {  return true;  } else if(upTraffic == other.upTraffic &&  downTraffic == other.downTraffic &&  timestamp == other.timestamp) {  return true;  }  return false;  }    @Override  public int compare(AccessLogSortKey other) {  if(upTraffic - other.upTraffic != 0) {  return (int) (upTraffic - other.upTraffic);  } else if(downTraffic - other.downTraffic != 0) {  return (int) (downTraffic - other.downTraffic);  } else if(timestamp - other.timestamp != 0) {  return (int) (timestamp - other.timestamp);  }  return 0;  }    @Override  public int compareTo(AccessLogSortKey other) {  if(upTraffic - other.upTraffic != 0) {  return (int) (upTraffic - other.upTraffic);  } else if(downTraffic - other.downTraffic != 0) {  return (int) (downTraffic - other.downTraffic);  } else if(timestamp - other.timestamp != 0) {  return (int) (timestamp - other.timestamp);  }  return 0;  }  public long getUpTraffic() {  return upTraffic;  }  public void setUpTraffic(long upTraffic) {  this.upTraffic = upTraffic;  }  public long getDownTraffic() {  return downTraffic;  }  public void setDownTraffic(long downTraffic) {  this.downTraffic = downTraffic;  }  public long getTimestamp() {  return timestamp;  }  public void setTimestamp(long timestamp) {  this.timestamp = timestamp;  }  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result + (int) (downTraffic ^ (downTraffic >>> 32));  result = prime \* result + (int) (timestamp ^ (timestamp >>> 32));  result = prime \* result + (int) (upTraffic ^ (upTraffic >>> 32));  return result;  }  @Override  public boolean equals(Object obj) {  if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  AccessLogSortKey other = (AccessLogSortKey) obj;  if (downTraffic != other.downTraffic)  return false;  if (timestamp != other.timestamp)  return false;  if (upTraffic != other.upTraffic)  return false;  return true;  }    } |

## 192\_Spark核心编程进阶-综合案例1：将二次排序key映射为RDD的key

|  |
| --- |
| import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import scala.Tuple2;  /\*\*  \* 移动端app访问流量日志分析案例  \*/  public class AppLogSpark {  public static void main(String[] args) throws Exception {  // 创建Spark配置和上下文对象  SparkConf conf = new SparkConf()  .setAppName("AppLogSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // 读取日志文件，并创建一个RDD  // 使用SparkContext的textFile()方法，即可读取本地磁盘文件，或者是HDFS上的文件  // 创建出来一个初始的RDD，其中包含了日志文件中的所有数据  JavaRDD<String> accessLogRDD = sc.textFile(  "C://Users//Administrator//Desktop//access.log");    // 将RDD映射为key-value格式，为后面的reduceByKey聚合做准备  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD =  mapAccessLogRDD2Pair(accessLogRDD);    // 根据deviceID进行聚合操作  // 获取每个deviceID的总上行流量、总下行流量、最早访问时间戳  JavaPairRDD<String, AccessLogInfo> aggrAccessLogPairRDD =  aggregateByDeviceID(accessLogPairRDD);    // 将按deviceID聚合RDD的key映射为二次排序key，value映射为deviceID  JavaPairRDD<AccessLogSortKey, String> accessLogSortRDD =  mapRDDKey2SortKey(aggrAccessLogPairRDD);    // 关闭Spark上下文  sc.close();  }    /\*\*  \* 将日志RDD映射为key-value的格式  \* @param accessLogRDD 日志RDD  \* @return key-value格式RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> mapAccessLogRDD2Pair(  JavaRDD<String> accessLogRDD) {  return accessLogRDD.mapToPair(new PairFunction<String, String, AccessLogInfo>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, AccessLogInfo> call(String accessLog)  throws Exception {  // 根据\t对日志进行切分  String[] accessLogSplited = accessLog.split("\t");    // 获取四个字段  long timestamp = Long.valueOf(accessLogSplited[0]);  String deviceID = accessLogSplited[1];  long upTraffic = Long.valueOf(accessLogSplited[2]);  long downTraffic = Long.valueOf(accessLogSplited[3]);    // 将时间戳、上行流量、下行流量，封装为自定义的可序列化对象  AccessLogInfo accessLogInfo = new AccessLogInfo(timestamp,  upTraffic, downTraffic);    return new Tuple2<String, AccessLogInfo>(deviceID, accessLogInfo);  }    });  }    /\*\*  \* 根据deviceID进行聚合操作  \* 计算出每个deviceID的总上行流量、总下行流量以及最早访问时间  \* @param accessLogPairRDD 日志key-value格式RDD  \* @return 按deviceID聚合RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> aggregateByDeviceID(  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD) {  return accessLogPairRDD.reduceByKey(new Function2<AccessLogInfo, AccessLogInfo, AccessLogInfo>() {    private static final long serialVersionUID = 1L;    @Override  public AccessLogInfo call(AccessLogInfo accessLogInfo1, AccessLogInfo accessLogInfo2)  throws Exception {  long timestamp = accessLogInfo1.getTimestamp() < accessLogInfo2.getTimestamp() ?  accessLogInfo1.getTimestamp() : accessLogInfo2.getTimestamp();  long upTraffic = accessLogInfo1.getUpTraffic() + accessLogInfo2.getUpTraffic();  long downTraffic = accessLogInfo1.getDownTraffic() + accessLogInfo2.getDownTraffic();    AccessLogInfo accessLogInfo = new AccessLogInfo();  accessLogInfo.setTimestamp(timestamp);  accessLogInfo.setUpTraffic(upTraffic);  accessLogInfo.setDownTraffic(downTraffic);    return accessLogInfo;  }    });  }    /\*\*  \* 将RDD的key映射为二次排序key  \* @param aggrAccessLogPairRDD 按deviceID聚合RDD  \* @return 二次排序key RDD  \*/  private static JavaPairRDD<AccessLogSortKey, String> mapRDDKey2SortKey(  JavaPairRDD<String, AccessLogInfo> aggrAccessLogPairRDD) {  return aggrAccessLogPairRDD.mapToPair(    new PairFunction<Tuple2<String,AccessLogInfo>, AccessLogSortKey, String>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<AccessLogSortKey, String> call(  Tuple2<String, AccessLogInfo> tuple) throws Exception {  // 获取tuple数据  String deviceID = tuple.\_1;  AccessLogInfo accessLogInfo = tuple.\_2;    // 将日志信息封装为二次排序key  AccessLogSortKey accessLogSortKey = new AccessLogSortKey(  accessLogInfo.getUpTraffic(),  accessLogInfo.getDownTraffic(),  accessLogInfo.getTimestamp());    // 返回新的Tuple  return new Tuple2<AccessLogSortKey, String>(accessLogSortKey, deviceID);  }    });  }  } |

## 193\_Spark核心编程进阶-综合案例1：执行二次排序以及获取top10数据

|  |
| --- |
| import java.util.List;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaPairRDD;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import scala.Tuple2;  /\*\*  \* 移动端app访问流量日志分析案例  \*/  public class AppLogSpark {  public static void main(String[] args) throws Exception {  // 创建Spark配置和上下文对象  SparkConf conf = new SparkConf()  .setAppName("AppLogSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);    // 读取日志文件，并创建一个RDD  // 使用SparkContext的textFile()方法，即可读取本地磁盘文件，或者是HDFS上的文件  // 创建出来一个初始的RDD，其中包含了日志文件中的所有数据  JavaRDD<String> accessLogRDD = sc.textFile(  "C://Users//Administrator//Desktop//access.log");    // 将RDD映射为key-value格式，为后面的reduceByKey聚合做准备  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD =  mapAccessLogRDD2Pair(accessLogRDD);    // 根据deviceID进行聚合操作  // 获取每个deviceID的总上行流量、总下行流量、最早访问时间戳  JavaPairRDD<String, AccessLogInfo> aggrAccessLogPairRDD =  aggregateByDeviceID(accessLogPairRDD);    // 将按deviceID聚合RDD的key映射为二次排序key，value映射为deviceID  JavaPairRDD<AccessLogSortKey, String> accessLogSortRDD =  mapRDDKey2SortKey(aggrAccessLogPairRDD);    // 执行二次排序操作，按照上行流量、下行流量以及时间戳进行倒序排序  JavaPairRDD<AccessLogSortKey ,String> sortedAccessLogRDD =  accessLogSortRDD.sortByKey(false);  // 获取top10数据  List<Tuple2<AccessLogSortKey, String>> top10DataList =  sortedAccessLogRDD.take(10);  for(Tuple2<AccessLogSortKey, String> data : top10DataList) {  System.out.println(data.\_2 + ": " + data.\_1);  }    // 关闭Spark上下文  sc.close();  }    /\*\*  \* 将日志RDD映射为key-value的格式  \* @param accessLogRDD 日志RDD  \* @return key-value格式RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> mapAccessLogRDD2Pair(  JavaRDD<String> accessLogRDD) {  return accessLogRDD.mapToPair(new PairFunction<String, String, AccessLogInfo>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, AccessLogInfo> call(String accessLog)  throws Exception {  // 根据\t对日志进行切分  String[] accessLogSplited = accessLog.split("\t");    // 获取四个字段  long timestamp = Long.valueOf(accessLogSplited[0]);  String deviceID = accessLogSplited[1];  long upTraffic = Long.valueOf(accessLogSplited[2]);  long downTraffic = Long.valueOf(accessLogSplited[3]);    // 将时间戳、上行流量、下行流量，封装为自定义的可序列化对象  AccessLogInfo accessLogInfo = new AccessLogInfo(timestamp,  upTraffic, downTraffic);    return new Tuple2<String, AccessLogInfo>(deviceID, accessLogInfo);  }    });  }    /\*\*  \* 根据deviceID进行聚合操作  \* 计算出每个deviceID的总上行流量、总下行流量以及最早访问时间  \* @param accessLogPairRDD 日志key-value格式RDD  \* @return 按deviceID聚合RDD  \*/  private static JavaPairRDD<String, AccessLogInfo> aggregateByDeviceID(  JavaPairRDD<String, AccessLogInfo> accessLogPairRDD) {  return accessLogPairRDD.reduceByKey(new Function2<AccessLogInfo, AccessLogInfo, AccessLogInfo>() {    private static final long serialVersionUID = 1L;    @Override  public AccessLogInfo call(AccessLogInfo accessLogInfo1, AccessLogInfo accessLogInfo2)  throws Exception {  long timestamp = accessLogInfo1.getTimestamp() < accessLogInfo2.getTimestamp() ?  accessLogInfo1.getTimestamp() : accessLogInfo2.getTimestamp();  long upTraffic = accessLogInfo1.getUpTraffic() + accessLogInfo2.getUpTraffic();  long downTraffic = accessLogInfo1.getDownTraffic() + accessLogInfo2.getDownTraffic();    AccessLogInfo accessLogInfo = new AccessLogInfo();  accessLogInfo.setTimestamp(timestamp);  accessLogInfo.setUpTraffic(upTraffic);  accessLogInfo.setDownTraffic(downTraffic);    return accessLogInfo;  }    });  }    /\*\*  \* 将RDD的key映射为二次排序key  \* @param aggrAccessLogPairRDD 按deviceID聚合RDD  \* @return 二次排序key RDD  \*/  private static JavaPairRDD<AccessLogSortKey, String> mapRDDKey2SortKey(  JavaPairRDD<String, AccessLogInfo> aggrAccessLogPairRDD) {  return aggrAccessLogPairRDD.mapToPair(    new PairFunction<Tuple2<String,AccessLogInfo>, AccessLogSortKey, String>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<AccessLogSortKey, String> call(  Tuple2<String, AccessLogInfo> tuple) throws Exception {  // 获取tuple数据  String deviceID = tuple.\_1;  AccessLogInfo accessLogInfo = tuple.\_2;    // 将日志信息封装为二次排序key  AccessLogSortKey accessLogSortKey = new AccessLogSortKey(  accessLogInfo.getUpTraffic(),  accessLogInfo.getDownTraffic(),  accessLogInfo.getTimestamp());    // 返回新的Tuple  return new Tuple2<AccessLogSortKey, String>(accessLogSortKey, deviceID);  }  });  }  } |

## 194\_Spark核心编程进阶-综合案例1：程序运行测试以及代码调试

## 195\_Spark核心编程进阶-部署第二台CentOS机器

|  |
| --- |
| 安装虚拟机和CentOS操作系统  1、使用课程提供的镜像：CentOS-6.4-i386-minimal.iso。  2、创建虚拟机：打开Virtual Box，点击“新建”按钮；输入虚拟机名称；选择操作系统为Linux，选择版本为Red Hat；分配1024MB内存；后面所有选项全部用默认的设置；注意，在Virtual disk file location and size中，一定要自己选择一个目录来存放虚拟机文件；最后点击“create”按钮，开始创建虚拟机。  3、设置网卡（桥接网卡）：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”，即桥接网卡。  4、安装CentOS 6.4操作系统：选择创建好的虚拟机，点击“开始”按钮；选择安装介质（即本地的CentOS 6.4镜像文件）；选择第一项开始安装-Skip-欢迎界面Next-选择默认语言-Baisc Storage Devices-Yes, discard any data-主机名:sparkproject1-选择时区-设置初始密码为hadoop-Replace Existing Linux System-Write changes to disk-开始安装。  5、安装完以后，会提醒你要重启一下，就是reboot，reboot就可以了。  配置CentOS网络  vi /etc/sysconfig/network-scripts/ifcfg-eth0  DEVICE=eth0  TYPE=Ethernet  ONBOOT=yes  BOOTPROTO=dhcp  service network restart  ifconfig  1、配置CentOS网络  vi /etc/sysconfig/network-scripts/ifcfg-eth0  DEVICE=eth0  TYPE=Ethernet  ONBOOT=yes  BOOTPROTO=static  IPADDR=IP地址  NETMASK=255.255.255.0  GATEWAY=Windows本机网关  2、重启网卡  service network restart  3、在/etc/hosts文件中配置本地ip到host的映射  4、使用SecureCRT从windows上连接虚拟机  关闭防火墙  service iptables stop  service ip6tables stop  service iptables status  service ip6tables status  chkconfig iptables off  chkconfig ip6tablesoff  vi /etc/selinux/config  SELINUX=disabled  配置yum  1、检查能否连通外网  ping www.baidu.com  2、配置yum  yum clean all  yum makecache  yum install telnet  安装JDk 1.7  1、将jdk-7u60-linux-i586.rpm上传到虚拟机中  2、安装JDK：rpm -ivh jdk-7u65-linux-i586.rpm  3、配置jdk相关的环境变量  vi ~/.bashrc  export JAVA\_HOME=/usr/java/latest  export PATH=$PATH:$JAVA\_HOME/bin  source ~/.bashrc  4、测试jdk安装是否成功：java -version  配置对自己的ssh免密码连接  生成本机的公钥，过程中不断敲回车即可，ssh-keygen命令默认会将公钥放在/root/.ssh目录下  ssh-keygen -t rsa  将公钥复制为authorized\_keys文件，此时使用ssh连接本机就不需要输入密码了  cd /root/.ssh  cp id\_rsa.pub authorized\_keys  配置互相的ip到hostname的映射  配置互相对对方的ssh免密码连接: ssh-copy-id -i hostname |

## 196\_Spark核心编程进阶-部署第二个Hadoop节点

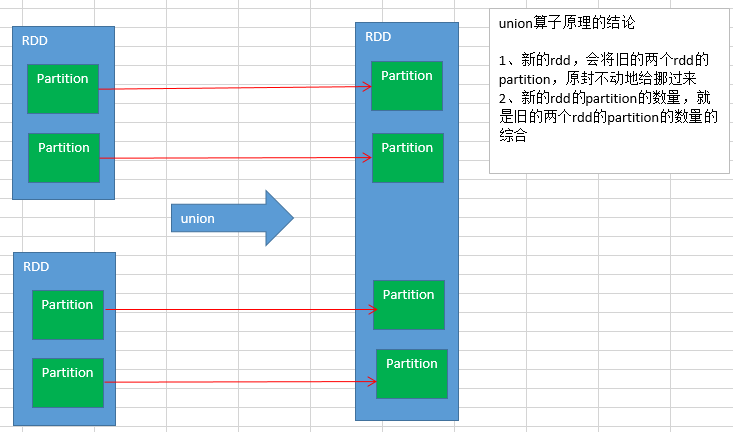
|  |
| --- |
| 部署安装包  1、使用课程提供的hadoop-2.5.0-cdh5.3.6.tar.gz，上传到虚拟机的/usr/local目录下。（http://archive.cloudera.com/cdh5/cdh/5/）  2、将hadoop包进行解压缩：tar -zxvf hadoop-2.5.0-cdh5.3.6.tar.gz  3、对hadoop目录进行重命名：mv hadoop-2.5.0-cdh5.3.6 hadoop  4、配置hadoop相关环境变量  vi ~/.bashrc  export HADOOP\_HOME=/usr/local/hadoop  export PATH=$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin  source ~/.bashrc  5、创建/usr/local/data目录  core-site.xml  <property>  <name>fs.default.name</name>  <value>hdfs://sparkupgrade1:9000</value>  </property>  hdfs-site.xml  <property>  <name>dfs.name.dir</name>  <value>/usr/local/data/namenode</value>  </property>  <property>  <name>dfs.data.dir</name>  <value>/usr/local/data/datanode</value>  </property>  <property>  <name>dfs.tmp.dir</name>  <value>/usr/local/data/tmp</value>  </property>  <property>  <name>dfs.replication</name>  <value>0</value>  </property>  mapred-site.xml  <property>  <name>mapreduce.framework.name</name>  <value>yarn</value>  </property>  yarn-site.xml  <property>  <name>yarn.resourcemanager.hostname</name>  <value>sparkupgrade11</value>  </property>  <property>  <name>yarn.nodemanager.aux-services</name>  <value>mapreduce\_shuffle</value>  </property>  slaves  sparkupgrade1  sparkupgrade2  在/usr/local目录下，创建data目录 |

## 197\_Spark核心编程进阶-将第二个Hadoop节点动态加入集群

|  |
| --- |
| 在master节点的slaves文件中增加新节点的主机名  在新节点手动启动  hadoop-daemon.sh start datanode  yarn-daemon.sh start nodemanager  在主节点  hdfs dfsadmin -refreshNodes  start-balancer.sh  通过web ui查看新节点是否增加  停止整个hadoop集群  然后将dfs.replication修改为1  再重新启动整个hadoop集群 |

## 198\_Spark核心编程进阶-使用yarn-client和yarn-cluster提交spark作业

## 199\_Spark内核原理进阶-union算子内部实现原理剖析

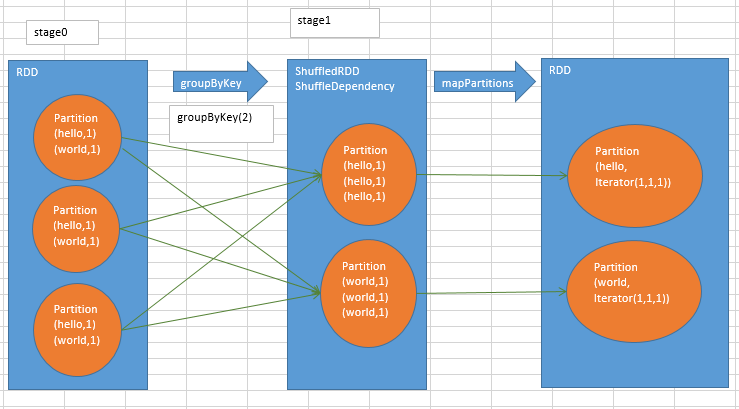


## 200\_Spark内核原理进阶-groupByKey算子内部实现原理剖析

一般来说，在执行shuffle类的算子的时候，比如groupByKey、reduceByKey、join等。

其实算子内部都会隐式地创建几个RDD出来。那些隐式创建的RDD，主要是作为这个操作的一些中间数据的表达，以及作为stage划分的边界。

因为有些隐式生成的RDD，可能是ShuffledRDD，dependency就是ShuffleDependency，DAGScheduler的源码，就会将这个RDD作为新的stage的第一个rdd，划分出来。

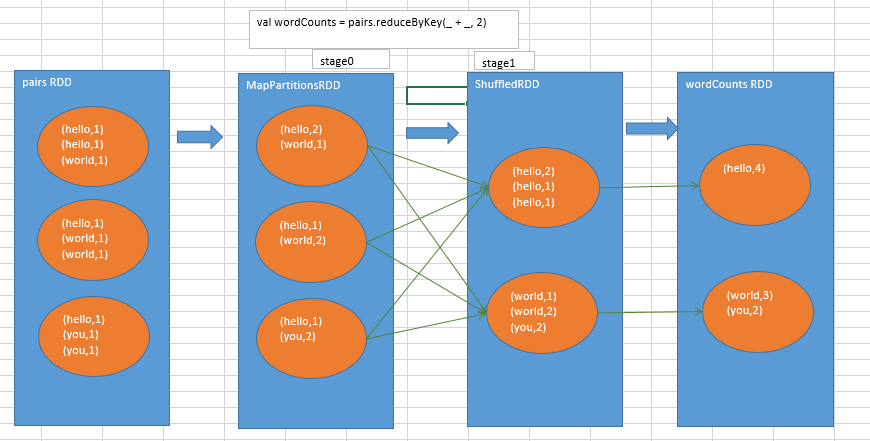


groupByKey等shuffle算子，都会创建一些隐式RDD。比如说这里，ShuffledRDD，作为一个shuffle过程中的中间数据的代表。

依赖这个ShuffledRDD创建出来一个新的stage（stage1）。ShuffledRDD会去触发shuffle read操作。从上游stage的task所在节点，拉取过来相同的key，做进一步的聚合。

对ShuffledRDD中的数据执行一个map类的操作，主要是对每个partition中的数据，都进行一个映射和聚合。这里主要是将每个key对应的数据都聚合到一个Iterator集合中。

## 201\_Spark内核原理进阶-reduceByKey算子内部实现原理剖析

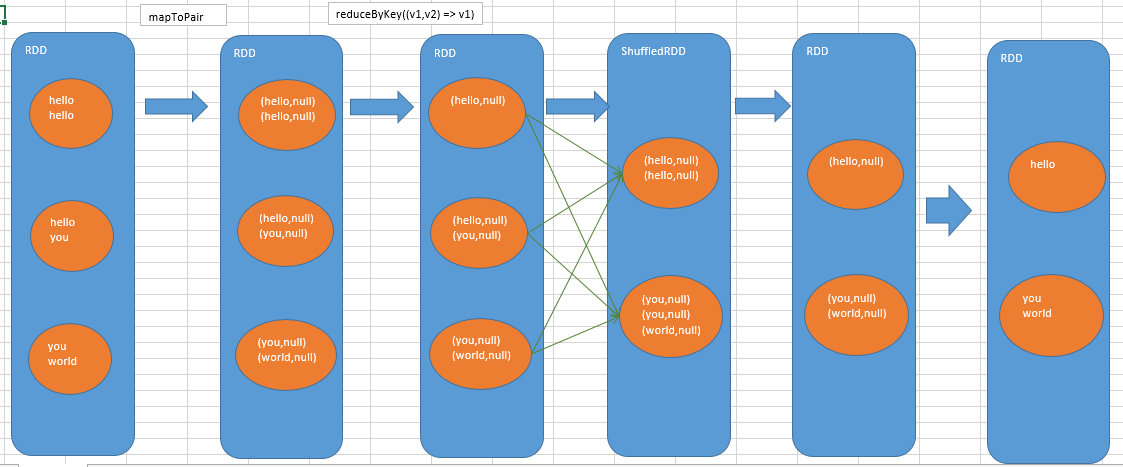


reduceByKey，看了内部原理之后，跟groupByKey的异同之处，在哪里？

1、不同之处：reduceByKey，多了一个rdd，MapPartitionsRDD，存在于stage0的，主要是代表了进行本地数据归约之后的rdd。所以，要网络传输的数据量，以及磁盘IO等，会减少，性能更高。

2、相同之处：后面进行shuffle read和聚合的过程基本和groupByKey类似。都是ShuffledRDD，去做shuffle read。然后聚合，聚合后的数据就是最终的rdd。

## 202\_Spark内核原理进阶-distinct算子内部实现原理剖析



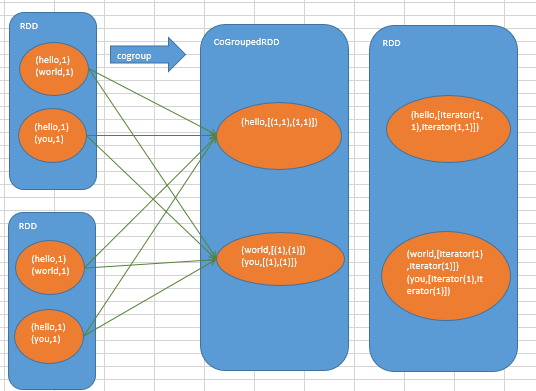
distinct算子的原理

1、首先，自己先给每个值打上一个v2，变成一个tuple

2、reduceByKey(...仅仅返回一个value)

3、将去重后的数据，从tuple还原为单值

## 203\_Spark内核原理进阶-cogroup算子内部实现原理剖析



cogroup算子

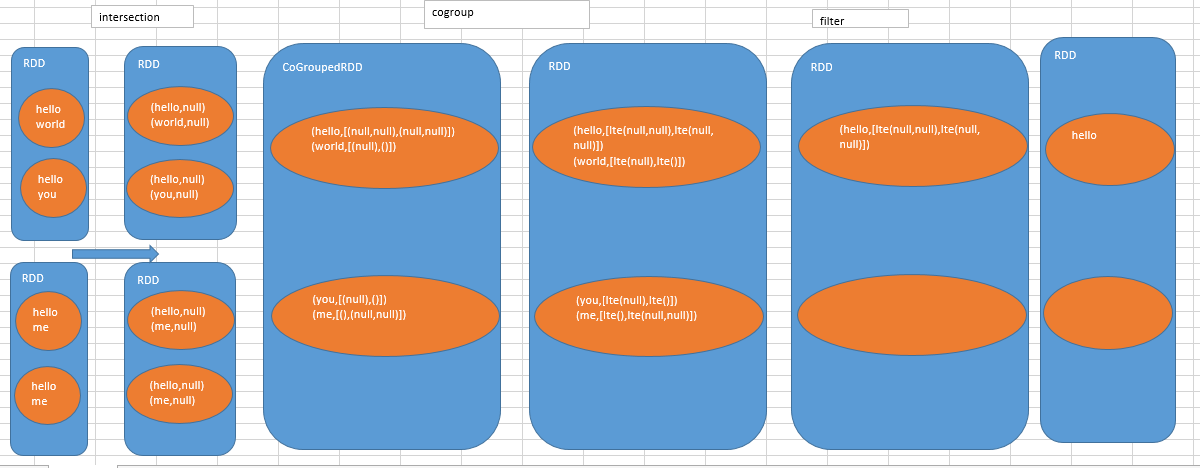
1、基础的算子

2、在我们大量的实践中，很少遇到说要用cogroup算子的情况

3、cogroup算子是其他很多算子的基础，比如join

每个RDD中的value会组合在一起，最终一个RDD形成一个单独的Iterator

## 204\_Spark内核原理进阶-intersection算子内部实现原理剖析



intersection算子原理

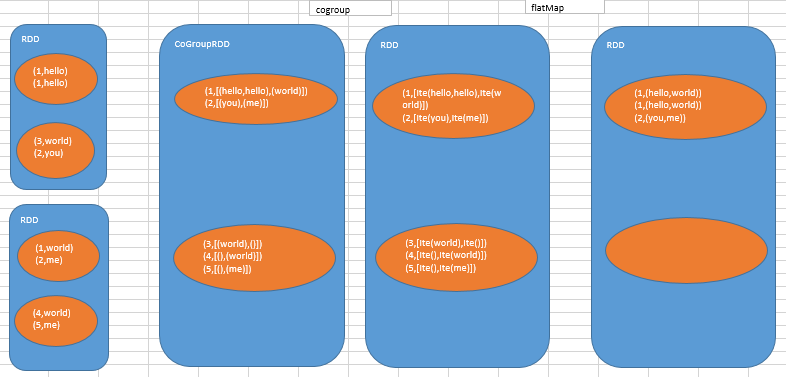
1、map，tuple

2、cogroup，聚合两个rdd的key

3、filter，过滤掉两个集合中任意一个集合为空的key

4、map，还原出单key

## 205\_Spark内核原理进阶-join算子内部实现原理剖析



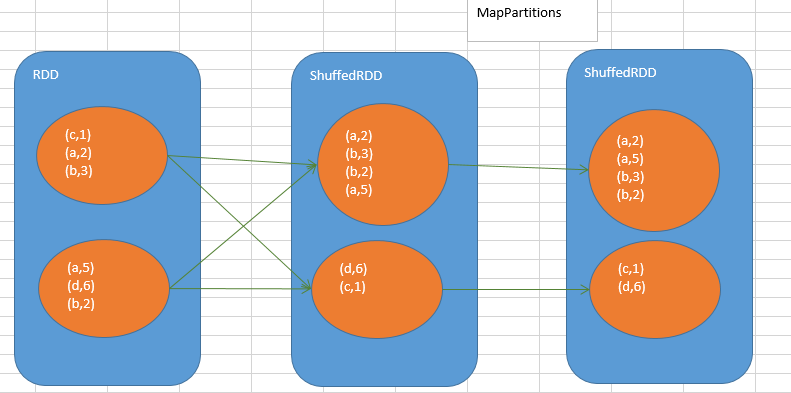
join算子的原理

1、cogroup，聚合两个rdd的key

2、flatMap，聚合后的每条数据，都可能返回多条数据

将每个key对应的两个集合的所有元素，做了一个笛卡尔积

## 206\_Spark内核原理进阶-sortByKey算子内部实现原理剖析

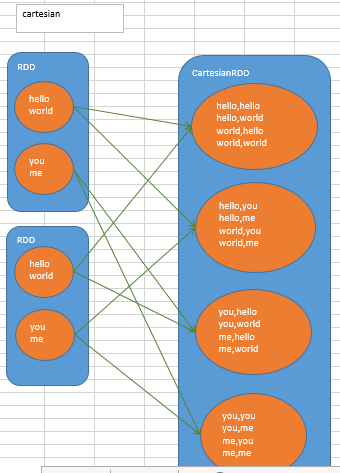


sortByKey

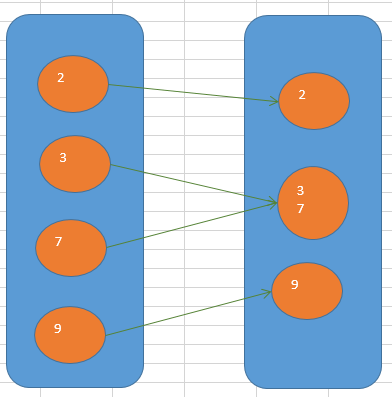
1、ShuffledRDD，做shuffle read，将相同的key拉到一个partition中来

2、mapPartitions，对每个partitions内的key进行全局的排序

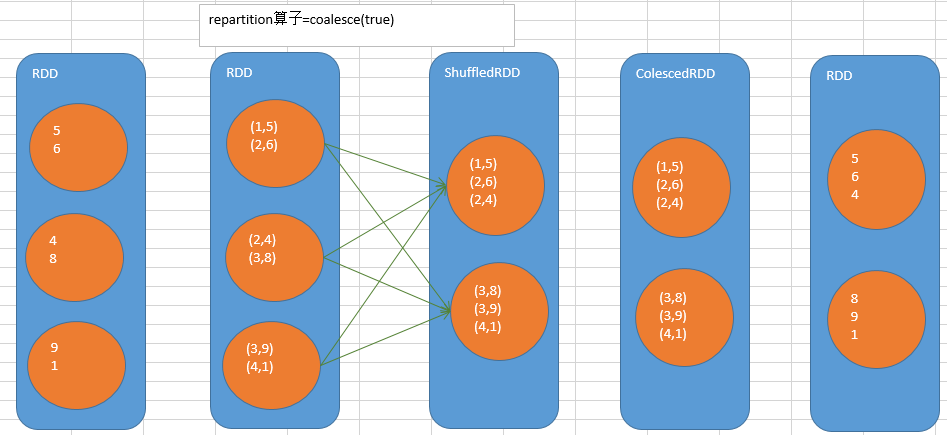
## 207\_Spark内核原理进阶-cartesian算子内部实现原理剖析



## 208\_Spark内核原理进阶-coalesce算子内部实现原理剖析



## 209\_Spark内核原理进阶-repartition算子内部实现原理剖析



repartition算子

1、map，附加了前缀，根据要重分区成几个分区，计算出前缀

2、shuffle->colesceRDD

3、去掉前缀，得到最终重分区好的RDD

## 210\_SparkSQL实战开发进阶-Hive013安装与测试

## 211\_SparkSQL实战开发进阶-ThriftJDBC、ODBCServer

Spark SQL的Thrift JDBC/ODBC server是基于Hive 0.13的HiveServer2实现的。这个服务启动之后，最主要的功能就是可以让我们通过Java JDBC来以编程的方式调用Spark SQL。此外，在启动该服务之后，可以通过Spark或Hive 0.13自带的beeline工具来进行测试。

要启动JDBC/ODBC server，主要执行Spark的sbin目录下的start-thriftserver.sh命令即可

start-thriftserver.sh命令可以接收所有spark-submit命令可以接收的参数，额外增加的一个参数是--hiveconf，可以用于指定一些

Hive的配置属性。可以通过执行./sbin/start-thriftserver.sh --help来查看所有可用参数的列表。默认情况下，启动的服务会在

localhost:10000地址上监听请求。

可以使用两种方式来改变服务监听的地址

第一种：指定环境变量

export HIVE\_SERVER2\_THRIFT\_PORT=<listening-port>

export HIVE\_SERVER2\_THRIFT\_BIND\_HOST=<listening-host>

./sbin/start-thriftserver.sh \

--master <master-uri> \

...

第二种：使用命令的参数

./sbin/start-thriftserver.sh \

--hiveconf hive.server2.thrift.port=<listening-port> \

--hiveconf hive.server2.thrift.bind.host=<listening-host> \

--master <master-uri>

...

hdfs dfs -chmod 777 /tmp/hive-root

./sbin/start-thriftserver.sh \

--jars /usr/local/hive/lib/mysql-connector-java-5.1.17.jar

这两种方式的区别就在于，第一种是针对整个机器上每次启动服务都生效的; 第二种仅仅针对本次启动生效

接着就可以通过Spark或Hive的beeline工具来测试Thrift JDBC/ODBC server

在Spark的bin目录中，执行beeline命令（当然，我们也可以使用Hive自带的beeline工具）：./bin/beeline

进入beeline命令行之后，连接到JDBC/ODBC server上去：beeline> !connect jdbc:hive2://localhost:10000

beeline通常会要求你输入一个用户名和密码。在非安全模式下，我们只要输入本机的用户名（比如root），以及一个空的密码即可。

对于安全模式，需要根据beeline的文档来进行认证。

除此之外，大家要注意的是，如果我们想要直接通过JDBC/ODBC服务访问Spark SQL，并直接对Hive执行SQL语句，那么就需要将Hive

的hive-site.xml配置文件放在Spark的conf目录下。

Thrift JDBC/ODBC server也支持通过HTTP传输协议发送thrift RPC消息。使用以下方式的配置可以启动HTTP模式：

命令参数

./sbin/start-thriftserver.sh \

--hive.server2.transport.mode=http \

--hive.server2.thrift.http.port=10001 \

--hive.server2.http.endpoint=cliservice \

--master <master-uri>

...

./sbin/start-thriftserver.sh \

--jars /usr/local/hive/lib/mysql-connector-java-5.1.17.jar \

--hiveconf hive.server2.transport.mode=http \

--hiveconf hive.server2.thrift.http.port=10001 \

--hiveconf hive.server2.http.endpoint=cliservice

beeline连接服务时指定参数

beeline> !connect jdbc:hive2://localhost:10001/default?hive.server2.transport.mode=http;hive.server2.thrift.http.path=cliservice

最重要的，当然是通过Java JDBC的方式，来访问Thrift JDBC/ODBC server，调用Spark SQL，并直接查询Hive中的数据

|  |
| --- |
| <dependency>  <groupId>org.apache.hive</groupId>  <artifactId>hive-jdbc</artifactId>  <version>0.13.0</version>  </dependency>  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  <version>4.4.1</version>  </dependency>  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpcore</artifactId>  <version>4.4.1</version>  </dependency> |

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  public class ThriftJDBCServerTest {    public static void main(String[] args) {  String sql = "select name from users where id=?";    Connection conn = null;  PreparedStatement pstmt = null;  ResultSet rs = null;    try {  Class.forName("org.apache.hive.jdbc.HiveDriver");    conn = DriverManager.getConnection("jdbc:hive2://192.168.0.103:10001/default?hive.server2.transport.mode=http;hive.server2.thrift.http.path=cliservice",  "root",  "");    pstmt = conn.prepareStatement(sql);  pstmt.setInt(1, 1);    rs = pstmt.executeQuery();  while(rs.next()) {  String name = rs.getString(1);  System.out.println(name);  }  } catch (Exception e) {  e.printStackTrace();  }  }    } |

## 212\_SparkSQL实战开发进阶-CLI命令行使用

Spark SQL CLI是一个很方便的工具，可以用来在本地模式下运行Hive的元数据服务，并且通过命令行执行针对Hive的SQL查询。但是

我们要注意的是，Spark SQL CLI是不能与Thrift JDBC server进行通信的。

如果要启动Spark SQL CLI，只要执行Spark的bin目录下的spark-sql命令即可

./bin/spark-sql --jars /usr/local/hive/lib/mysql-connector-java-5.1.17.jar

这里同样要注意的是，必须将我们的hive-site.xml文件放在Spark的conf目录下。

我们也可以通过执行./bin/spark-sql --help命令，来获取该命令的所有帮助选项。

## 213\_SparkSQL实战开发进阶-综合案例2：新闻网站关键指标离线统计

案例背景

新闻网站

1、版块

2、新闻页面

3、新用户注册

4、用户跳出

案例需求分析

每天每个页面的PV：PV是Page View，是指一个页面被所有用户访问次数的总和，页面被访问一次就被记录1次PV

每天每个页面的UV：UV是User View，是指一个页面被多少个用户访问了，一个用户访问一次是1次UV，一个用户访问多次还是1次UV

新用户注册比率：当天注册用户数 / 当天未注册用户数

用户跳出率：IP只浏览了一个页面就离开网站的次数/网站总访问数（PV）

版块热度排行榜：根据每个版块每天被访问的次数，做出一个排行榜

网站日志格式

date timestamp userid pageid section action

日志字段说明

date: 日期，yyyy-MM-dd格式

timestamp: 时间戳

userid: 用户id

pageid: 页面id

section: 新闻版块

action: 用户行为，两类，点击页面和注册

模拟数据生成程序

模式数据演示

在hive中创建访问日志表

create table news\_access (

date string,

timestamp bigint,

userid bigint,

pageid bigint,

section string,

action string)

将模拟数据导入hive表中

load data local inpath '/usr/local/test/news\_access.log' into table news\_access;

**OfflineDataGenerator.java**

|  |
| --- |
| import java.io.FileOutputStream;  import java.io.OutputStreamWriter;  import java.io.PrintWriter;  import java.text.SimpleDateFormat;  import java.util.Calendar;  import java.util.Date;  import java.util.Random;  /\*\*  \* 离线数据生成器  \* @author Administrator  \*  \*/  public class OfflineDataGenerator {  public static void main(String[] args) throws Exception {  StringBuffer buffer = new StringBuffer("");    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  Random random = new Random();  String[] sections = new String[] {"country", "international", "sport", "entertainment", "movie", "carton", "tv-show", "technology", "internet", "car"};    int[] newOldUserArr = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};    // 生成日期，默认就是昨天  Calendar cal = Calendar.getInstance();  cal.setTime(new Date());  cal.add(Calendar.DAY\_OF\_YEAR, -1);  Date yesterday = cal.getTime();    String date = sdf.format(yesterday);  // 生成1000条访问数据  for(int i = 0; i < 1000; i++) {  // 生成时间戳  long timestamp = new Date().getTime();    // 生成随机userid（默认1000注册用户，每天1/10的访客是未注册用户）  Long userid = 0L;  int newOldUser = newOldUserArr[random.nextInt(10)];  if(newOldUser == 1) {  userid = null;  } else {  userid = (long) random.nextInt(1000);  }  // 生成随机pageid（总共1k个页面）  Long pageid = (long) random.nextInt(1000);    // 生成随机版块（总共10个版块）  String section = sections[random.nextInt(10)];  // 生成固定的行为，view  String action = "view";  buffer.append(date).append("")  .append(timestamp).append("")  .append(userid).append("")  .append(pageid).append("")  .append(section).append("")  .append(action).append("\n");  }  // 生成10条注册数据  for(int i = 0; i < 10; i++) {  // 生成时间戳  long timestamp = new Date().getTime();  // 新用户都是userid为null  Long userid = null;  // 生成随机pageid，都是null  Long pageid = null;  // 生成随机版块，都是null  String section = null;  // 生成固定的行为，view  String action = "register";  buffer.append(date).append("")  .append(timestamp).append("")  .append(userid).append("")  .append(pageid).append("")  .append(section).append("")  .append(action).append("\n");  }  PrintWriter pw = null;  try {  pw = new PrintWriter(new OutputStreamWriter(  new FileOutputStream("C:\\Users\\Administrator\\Desktop\\access.log")));  pw.write(buffer.toString());  } catch (Exception e) {  e.printStackTrace();  } finally {  pw.close();  }  }  } |

## 214\_SparkSQL实战开发进阶-综合案例2：页面pv统计以及排序

## 215\_Spark SQL实战开发进阶-综合案例2：页面uv统计以及排序和count(distinct) bug说明

## 216\_SparkSQL实战开发进阶-综合案例2：新用户注册比例统计

## 217\_Spark SQL实战开发进阶-综合案例2：用户跳出率统计

## 218\_SparkSQL实战开发进阶-综合案例2：版块热度排行榜统计

## 219\_SparkSQL实战开发进阶-综合案例2：测试与调试

NewsOfflineStatSpark.java

|  |
| --- |
| import java.math.BigDecimal;  import java.text.SimpleDateFormat;  import java.util.Calendar;  import java.util.Date;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.sql.DataFrame;  import org.apache.spark.sql.hive.HiveContext;  /\*\*  \* 新闻网站关键指标离线统计Spark作业  \* @author Administrator  \*  \*/  public class NewsOfflineStatSpark {  public static void main(String[] args) {  // 一般来说，在小公司中，可能就是将我们的spark作业使用linux的crontab进行调度  // 将作业jar放在一台安装了spark客户端的机器上，并编写了对应的spark-submit shell脚本  // 在crontab中可以配置，比如说每天凌晨3点执行一次spark-submit shell脚本，提交一次spark作业  // 一般来说，离线的spark作业，每次运行，都是去计算昨天的数据  // 大公司总，可能是使用较为复杂的开源大数据作业调度平台，比如常用的有azkaban、oozie等  // 但是，最大的那几个互联网公司，比如说BAT、美团、京东，作业调度平台，都是自己开发的  // 我们就会将开发好的Spark作业，以及对应的spark-submit shell脚本，配置在调度平台上，几点运行  // 同理，每次运行，都是计算昨天的数据    // 一般来说，每次spark作业计算出来的结果，实际上，大部分情况下，都会写入mysql等存储  // 这样的话，我们可以基于mysql，用java web技术开发一套系统平台，来使用图表的方式展示每次spark计算  // 出来的关键指标  // 比如用折线图，可以反映最近一周的每天的用户跳出率的变化    // 也可以通过页面，给用户提供一个查询表单，可以查询指定的页面的最近一周的pv变化  // date pageid pv  // 插入mysql中，后面用户就可以查询指定日期段内的某个page对应的所有pv，然后用折线图来反映变化曲线    // 拿到昨天的日期，去hive表中，针对昨天的数据执行SQL语句  String yesterday = getYesterday();    // 创建SparkConf以及Spark上下文  SparkConf conf = new SparkConf()  .setAppName("NewsOfflineStatSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);  HiveContext hiveContext = new HiveContext(sc.sc());    // 开发第一个关键指标：页面pv统计以及排序  calculateDailyPagePv(hiveContext, yesterday);  // 开发第二个关键指标：页面uv统计以及排序  calculateDailyPageUv(hiveContext, yesterday);  // 开发第三个关键指标：新用户注册比率统计  calculateDailyNewUserRegisterRate(hiveContext, yesterday);  // 开发第四个关键指标：用户跳出率统计  calculateDailyUserJumpRate(hiveContext, yesterday);  // 开发第五个关键指标：版块热度排行榜  calculateDailySectionPvSort(hiveContext, yesterday);    // 关闭Spark上下文  sc.close();  }  /\*\*  \* 获取昨天的字符串类型的日期  \* @return 日期  \*/  private static String getYesterday() {  Calendar cal = Calendar.getInstance();  cal.setTime(new Date());  cal.add(Calendar.DAY\_OF\_YEAR, -1);    Date yesterday = cal.getTime();    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  return sdf.format(yesterday);  }    /\*\*  \* 计算每天每个页面的pv以及排序  \* 排序的好处：排序后，插入mysql，java web系统要查询每天pv top10的页面，直接查询mysql表limit 10就可以  \* 如果我们这里不排序，那么java web系统就要做排序，反而会影响java web系统的性能，以及用户响应时间  \*/  private static void calculateDailyPagePv(  HiveContext hiveContext, String date) {  String sql =  "SELECT "  + "date,"  + "pageid,"  + "pv "  + "FROM ( "  + "SELECT "  + "date,"  + "pageid,"  + "count(\*) pv "  + "FROM news\_access "  + "WHERE action='view' "  + "AND date='" + date + "' "  + "GROUP BY date,pageid "  + ") t "  + "ORDER BY pv DESC ";    DataFrame df = hiveContext.sql(sql);    // 在这里，我们也可以转换成一个RDD，然后对RDD执行一个foreach算子  // 在foreach算子中，将数据写入mysql中    df.show();  }    /\*\*  \* 计算每天每个页面的uv以及排序  \* Spark SQL的count(distinct)语句，有bug，默认会产生严重的数据倾斜  \* 只会用一个task，来做去重和汇总计数，性能很差  \* @param hiveContext  \* @param date  \*/  private static void calculateDailyPageUv(  HiveContext hiveContext, String date) {  String sql =  "SELECT "  + "date,"  + "pageid,"  + "uv "  + "FROM ( "  + "SELECT "  + "date,"  + "pageid,"  + "count(\*) uv "  + "FROM ( "  + "SELECT "  + "date,"  + "pageid,"  + "userid "  + "FROM news\_access "  + "WHERE action='view' "  + "AND date='" + date + "' "  + "GROUP BY date,pageid,userid "  + ") t2 "  + "GROUP BY date,pageid "  + ") t "  + "ORDER BY uv DESC ";    DataFrame df = hiveContext.sql(sql);    df.show();  }    /\*\*  \* 计算每天的新用户注册比例  \* @param hiveContext  \* @param date  \*/  private static void calculateDailyNewUserRegisterRate(  HiveContext hiveContext, String date) {  // 昨天所有访问行为中，userid为null，新用户的访问总数  String sql1 = "SELECT count(\*) FROM news\_access WHERE action='view' AND date='" + date + "' AND userid IS NULL";  // 昨天的总注册用户数  String sql2 = "SELECT count(\*) FROM news\_access WHERE action='register' AND date='" + date + "' ";    // 执行两条SQL，获取结果  Object result1 = hiveContext.sql(sql1).collect()[0].get(0);  long number1 = 0L;  if(result1 != null) {  number1 = Long.valueOf(String.valueOf(result1));  }    Object result2 = hiveContext.sql(sql2).collect()[0].get(0);  long number2 = 0L;  if(result2 != null) {  number2 = Long.valueOf(String.valueOf(result2));  }    // 计算结果  System.out.println("======================" + number1 + "======================");  System.out.println("======================" + number2 + "======================");  double rate = (double)number2 / (double)number1;  System.out.println("======================" + formatDouble(rate, 2) + "======================");  }    /\*\*  \* 计算每天的用户跳出率  \* @param hiveContext  \* @param date  \*/  private static void calculateDailyUserJumpRate(  HiveContext hiveContext, String date) {  // 计算已注册用户的昨天的总的访问pv  String sql1 = "SELECT count(\*) FROM news\_access WHERE action='view' AND date='" + date + "' AND userid IS NOT NULL ";    // 已注册用户的昨天跳出的总数  String sql2 = "SELECT count(\*) FROM ( SELECT count(\*) cnt FROM news\_access WHERE action='view' AND date='" + date + "' AND userid IS NOT NULL GROUP BY userid HAVING cnt=1 ) t ";    // 执行两条SQL，获取结果  Object result1 = hiveContext.sql(sql1).collect()[0].get(0);  long number1 = 0L;  if(result1 != null) {  number1 = Long.valueOf(String.valueOf(result1));  }    Object result2 = hiveContext.sql(sql2).collect()[0].get(0);  long number2 = 0L;  if(result2 != null) {  number2 = Long.valueOf(String.valueOf(result2));  }    // 计算结果  System.out.println("======================" + number1 + "======================");  System.out.println("======================" + number2 + "======================");  double rate = (double)number2 / (double)number1;  System.out.println("======================" + formatDouble(rate, 2) + "======================");  }    /\*\*  \* 计算每天的版块热度排行榜  \* @param hiveContext  \* @param date  \*/  private static void calculateDailySectionPvSort(  HiveContext hiveContext, String date) {  String sql =  "SELECT "  + "date,"  + "section,"  + "pv "  + "FROM ( "  + "SELECT "  + "date,"  + "section,"  + "count(\*) pv "  + "FROM news\_access "  + "WHERE action='view' "  + "AND date='" + date + "' "  + "GROUP BY date,section "  + ") t "  + "ORDER BY pv DESC ";    DataFrame df = hiveContext.sql(sql);    df.show();  }    /\*\*  \* 格式化小数  \* @param str 字符串  \* @param scale 四舍五入的位数  \* @return 格式化小数  \*/  private static double formatDouble(double num, int scale) {  BigDecimal bd = new BigDecimal(num);  return bd.setScale(scale, BigDecimal.ROUND\_HALF\_UP).doubleValue();  }    } |

/usr/local/spark/bin/spark-submit \

--class cn.spark.study.sql.upgrade.news.NewsOfflineStatSpark \

--num-executors 1 \

--driver-memory 100m \

--executor-memory 100m \

--executor-cores 1 \

--files /usr/local/hive/conf/hive-site.xml \

--driver-class-path /usr/local/hive/lib/mysql-connector-java-5.1.17.jar \

/usr/local/test/spark-study-java-0.0.1-SNAPSHOT-jar-with-dependencies.jar \



## 220\_SparkStreaming实战开发进阶-flume安装

|  |
| --- |
| /usr/local/spark/bin/spark-submit \  --class cn.spark.study.sql.upgrade.news.NewsOfflineStatSpark \  --num-executors 1 \  --driver-memory 100m \  --executor-memory 100m \  --executor-cores 1 \  --files /usr/local/hive/conf/hive-site.xml \  --driver-class-path /usr/local/hive/lib/mysql-connector-java-5.1.17.jar \  /usr/local/test/spark-study-java-0.0.1-SNAPSHOT-jar-with-dependencies.jar \ |

日志文件

2016-02-201456038028630459197tv-showview

2016-02-201456038028630475537internationalview

2016-02-201456038028630681885internationalview

2016-02-20145603802863059137countryview

2016-02-201456038028630null105technologyview

2016-02-201456038028630846776carview

2016-02-201456038028630396226internationalview

2016-02-201456038028630370407entertainmentview

flume安装：

|  |
| --- |
| 1、将课程提供的flume-ng-1.5.0-cdh5.3.6.tar.gz使用WinSCP拷贝到sparkproject1的/usr/local目录下。  2、对flume进行解压缩：tar -zxvf flume-ng-1.5.0-cdh5.3.6.tar.gz  3、对flume目录进行重命名：mv apache-flume-1.5.0-cdh5.3.6-bin flume  4、配置scala相关的环境变量  vi ~/.bashrc  export FLUME\_HOME=/usr/local/flume  export FLUME\_CONF\_DIR=$FLUME\_HOME/conf  export PATH=$FLUME\_HOME/bin  source ~/.bashrc |

flume配置文件：

|  |
| --- |
| vi conf/flume-conf.properties  #agent1表示代理名称  agent1.sources=source1  agent1.sinks=sink1  agent1.channels=channel1  #配置source1  agent1.sources.source1.type=spooldir  agent1.sources.source1.spoolDir=/usr/local/logs  agent1.sources.source1.channels=channel1  agent1.sources.source1.fileHeader = false  agent1.sources.source1.interceptors = i1  agent1.sources.source1.interceptors.i1.type = timestamp  #配置channel1  agent1.channels.channel1.type=file  agent1.channels.channel1.checkpointDir=/usr/local/logs\_tmp\_cp  agent1.channels.channel1.dataDirs=/usr/local/logs\_tmp  #配置sink1  agent1.sinks.sink1.type=hdfs  agent1.sinks.sink1.hdfs.path=hdfs://sparkproject1:9000/logs  agent1.sinks.sink1.hdfs.fileType=DataStream  agent1.sinks.sink1.hdfs.writeFormat=TEXT  agent1.sinks.sink1.hdfs.rollInterval=1  agent1.sinks.sink1.channel=channel1  agent1.sinks.sink1.hdfs.filePrefix=%Y-%m-%d |

创建目录：

本地文件夹：mkdir /usr/local/logs

HDFS文件夹：hdfs dfs -mkdir /logs

启动flume：

flume-ng agent -n agent1 -c conf -f /usr/local/flume/conf/flume-conf.properties -Dflume.root.logger=DEBUG,console

**NewsOfflineStatSpark.java**

|  |
| --- |
| import java.math.BigDecimal;  import java.text.SimpleDateFormat;  import java.util.Calendar;  import java.util.Date;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.sql.DataFrame;  import org.apache.spark.sql.hive.HiveContext;  /\*\*新闻网站关键指标离线统计Spark作业\*/  public class NewsOfflineStatSpark {  public static void main(String[] args) {  // 一般来说，在小公司中，可能就是将我们的spark作业使用linux的crontab进行调度  // 将作业jar放在一台安装了spark客户端的机器上，并编写了对应的spark-submit shell脚本  // 在crontab中可以配置，比如说每天凌晨3点执行一次spark-submit shell脚本，提交一次spark作业  // 一般来说，离线的spark作业，每次运行，都是去计算昨天的数据  // 大公司总，可能是使用较为复杂的开源大数据作业调度平台，比如常用的有azkaban、oozie等  // 但是，最大的那几个互联网公司，比如说BAT、美团、京东，作业调度平台，都是自己开发的  // 我们就会将开发好的Spark作业，以及对应的spark-submit shell脚本，配置在调度平台上，几点运行  // 同理，每次运行，都是计算昨天的数据    // 一般来说，每次spark作业计算出来的结果，实际上，大部分情况下，都会写入mysql等存储  // 这样的话，我们可以基于mysql，用java web技术开发一套系统平台，来使用图表的方式展示每次spark计算  // 出来的关键指标  // 比如用折线图，可以反映最近一周的每天的用户跳出率的变化    // 也可以通过页面，给用户提供一个查询表单，可以查询指定的页面的最近一周的pv变化  // date pageid pv  // 插入mysql中，后面用户就可以查询指定日期段内的某个page对应的所有pv，然后用折线图来反映变化曲线    // 拿到昨天的日期，去hive表中，针对昨天的数据执行SQL语句  String yesterday = getYesterday();    // 创建SparkConf以及Spark上下文  SparkConf conf = new SparkConf()  .setAppName("NewsOfflineStatSpark")  .setMaster("local");  JavaSparkContext sc = new JavaSparkContext(conf);  HiveContext hiveContext = new HiveContext(sc.sc());    // 开发第一个关键指标：页面pv统计以及排序  calculateDailyPagePv(hiveContext, yesterday);  // 开发第二个关键指标：页面uv统计以及排序  calculateDailyPageUv(hiveContext, yesterday);  // 开发第三个关键指标：新用户注册比率统计  calculateDailyNewUserRegisterRate(hiveContext, yesterday);  // 开发第四个关键指标：用户跳出率统计  calculateDailyUserJumpRate(hiveContext, yesterday);  // 开发第五个关键指标：版块热度排行榜  calculateDailySectionPvSort(hiveContext, yesterday);    // 关闭Spark上下文  sc.close();  }  /\*\*  \* 获取昨天的字符串类型的日期  \* @return 日期  \*/  private static String getYesterday() {  Calendar cal = Calendar.getInstance();  cal.setTime(new Date());  cal.add(Calendar.DAY\_OF\_YEAR, -1);    Date yesterday = cal.getTime();    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  return sdf.format(yesterday);  }    /\*\*  \* 计算每天每个页面的pv以及排序  \* 排序的好处：排序后，插入mysql，java web系统要查询每天pv top10的页面，直接查询mysql表limit 10就可以  \* 如果我们这里不排序，那么java web系统就要做排序，反而会影响java web系统的性能，以及用户响应时间  \*/  private static void calculateDailyPagePv(  HiveContext hiveContext, String date) {  String sql =  "SELECT "  + "date,"  + "pageid,"  + "pv "  + "FROM ( "  + "SELECT "  + "date,"  + "pageid,"  + "count(\*) pv "  + "FROM news\_access "  + "WHERE action='view' "  + "AND date='" + date + "' "  + "GROUP BY date,pageid "  + ") t "  + "ORDER BY pv DESC ";    DataFrame df = hiveContext.sql(sql);    // 在这里，我们也可以转换成一个RDD，然后对RDD执行一个foreach算子  // 在foreach算子中，将数据写入mysql中    df.show();  }    /\*\*  \* 计算每天每个页面的uv以及排序  \* Spark SQL的count(distinct)语句，有bug，默认会产生严重的数据倾斜  \* 只会用一个task，来做去重和汇总计数，性能很差  \* @param hiveContext  \* @param date  \*/  private static void calculateDailyPageUv(  HiveContext hiveContext, String date) {  String sql =  "SELECT "  + "date,"  + "pageid,"  + "uv "  + "FROM ( "  + "SELECT "  + "date,"  + "pageid,"  + "count(\*) uv "  + "FROM ( "  + "SELECT "  + "date,"  + "pageid,"  + "userid "  + "FROM news\_access "  + "WHERE action='view' "  + "AND date='" + date + "' "  + "GROUP BY date,pageid,userid "  + ") t2 "  + "GROUP BY date,pageid "  + ") t "  + "ORDER BY uv DESC ";    DataFrame df = hiveContext.sql(sql);    df.show();  }    /\*\*  \* 计算每天的新用户注册比例  \* @param hiveContext  \* @param date  \*/  private static void calculateDailyNewUserRegisterRate(  HiveContext hiveContext, String date) {  // 昨天所有访问行为中，userid为null，新用户的访问总数  String sql1 = "SELECT count(\*) FROM news\_access WHERE action='view' AND date='" + date + "' AND userid IS NULL";  // 昨天的总注册用户数  String sql2 = "SELECT count(\*) FROM news\_access WHERE action='register' AND date='" + date + "' ";    // 执行两条SQL，获取结果  Object result1 = hiveContext.sql(sql1).collect()[0].get(0);  long number1 = 0L;  if(result1 != null) {  number1 = Long.valueOf(String.valueOf(result1));  }    Object result2 = hiveContext.sql(sql2).collect()[0].get(0);  long number2 = 0L;  if(result2 != null) {  number2 = Long.valueOf(String.valueOf(result2));  }    // 计算结果  System.out.println("======================" + number1 + "======================");  System.out.println("======================" + number2 + "======================");  double rate = (double)number2 / (double)number1;  System.out.println("======================" + formatDouble(rate, 2) + "======================");  }    /\*\*  \* 计算每天的用户跳出率  \* @param hiveContext  \* @param date  \*/  private static void calculateDailyUserJumpRate(  HiveContext hiveContext, String date) {  // 计算已注册用户的昨天的总的访问pv  String sql1 = "SELECT count(\*) FROM news\_access WHERE action='view' AND date='" + date + "' AND userid IS NOT NULL ";    // 已注册用户的昨天跳出的总数  String sql2 = "SELECT count(\*) FROM ( SELECT count(\*) cnt FROM news\_access WHERE action='view' AND date='" + date + "' AND userid IS NOT NULL GROUP BY userid HAVING cnt=1 ) t ";    // 执行两条SQL，获取结果  Object result1 = hiveContext.sql(sql1).collect()[0].get(0);  long number1 = 0L;  if(result1 != null) {  number1 = Long.valueOf(String.valueOf(result1));  }    Object result2 = hiveContext.sql(sql2).collect()[0].get(0);  long number2 = 0L;  if(result2 != null) {  number2 = Long.valueOf(String.valueOf(result2));  }    // 计算结果  System.out.println("======================" + number1 + "======================");  System.out.println("======================" + number2 + "======================");  double rate = (double)number2 / (double)number1;  System.out.println("======================" + formatDouble(rate, 2) + "======================");  }    /\*\*  \* 计算每天的版块热度排行榜  \* @param hiveContext  \* @param date  \*/  private static void calculateDailySectionPvSort(  HiveContext hiveContext, String date) {  String sql =  "SELECT "  + "date,"  + "section,"  + "pv "  + "FROM ( "  + "SELECT "  + "date,"  + "section,"  + "count(\*) pv "  + "FROM news\_access "  + "WHERE action='view' "  + "AND date='" + date + "' "  + "GROUP BY date,section "  + ") t "  + "ORDER BY pv DESC ";    DataFrame df = hiveContext.sql(sql);    df.show();  }    /\*\*  \* 格式化小数  \* @param str 字符串  \* @param scale 四舍五入的位数  \* @return 格式化小数  \*/  private static double formatDouble(double num, int scale) {  BigDecimal bd = new BigDecimal(num);  return bd.setScale(scale, BigDecimal.ROUND\_HALF\_UP).doubleValue();  }    } |

## 221\_Spark Streaming实战开发进阶-接收flume实时数据流-flume风格的基于push的方式

Flume被设计为可以在agent之间推送数据，而不一定是从agent将数据传输到sink中。在这种方式下，Spark Streaming需要启动一个作为Avro Agent的Receiver，来让

flume可以推送数据过来。下面是我们的整合步骤：

前提需要

选择一台机器：

1、Spark Streaming与Flume都可以在这台机器上启动，Spark的其中一个Worker必须运行在这台机器上面

2、Flume可以将数据推送到这台机器上的某个端口

由于flume的push模型，Spark Streaming必须先启动起来，Receiver要被调度起来并且监听本地某个端口，来让flume推送数据。

配置flume

在flume-conf.properties文件中，配置flume的sink是将数据推送到其他的agent中

agent1.sinks.sink1.type = avro

agent1.sinks.sink1.channel = channel1

agent1.sinks.sink1.hostname = 192.168.0.103

agent1.sinks.sink1.port = 8888

配置spark streaming

在我们的spark工程的pom.xml中加入spark streaming整合flume的依赖

groupId = org.apache.spark

artifactId = spark-streaming-flume\_2.10

version = 1.5.0

在代码中使用整合flume的方式创建输入DStream

import org.apache.spark.streaming.flume.\*;

JavaReceiverInputDStream<SparkFlumeEvent> flumeStream =

FlumeUtils.createStream(streamingContext, [chosen machine's hostname], [chosen port]);

这里有一点需要注意的是，这里监听的hostname，必须与cluster manager（比如Standalone Master、YARN ResourceManager）是同一台机器，这样cluster manager

才能匹配到正确的机器，并将receiver调度在正确的机器上运行。

部署spark streaming应用

打包工程为一个jar包，使用spark-submit来提交作业

启动flume agent

flume-ng agent -n agent1 -c conf -f /usr/local/flume/conf/flume-conf.properties -Dflume.root.logger=DEBUG,console

什么时候我们应该用Spark Streaming整合Kafka去用，做实时计算？

什么使用应该整合flume？

看你的实时数据流的产出频率

1、如果你的实时数据流产出特别频繁，比如说一秒钟10w条，那就必须是kafka，分布式的消息缓存中间件，可以承受超高并发

2、如果你的实时数据流产出频率不固定，比如有的时候是1秒10w，有的时候是1个小时才10w，可以选择将数据用nginx日志来表示，每隔一段时间将日志文件

放到flume监控的目录中，然后呢，spark streaming来计算

**FlumePushWordCount.java**

|  |
| --- |
| import java.util.Arrays;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.function.FlatMapFunction;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import org.apache.spark.streaming.Durations;  import org.apache.spark.streaming.api.java.JavaDStream;  import org.apache.spark.streaming.api.java.JavaPairDStream;  import org.apache.spark.streaming.api.java.JavaReceiverInputDStream;  import org.apache.spark.streaming.api.java.JavaStreamingContext;  import org.apache.spark.streaming.flume.FlumeUtils;  import org.apache.spark.streaming.flume.SparkFlumeEvent;  import scala.Tuple2;  /\*\*基于Flume Push方式的实时wordcount程序\*/  public class FlumePushWordCount {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setMaster("local[2]")  .setAppName("FlumePushWordCount");  JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(5));    JavaReceiverInputDStream<SparkFlumeEvent> lines =  FlumeUtils.createStream(jssc, "192.168.0.103", 8888);    JavaDStream<String> words = lines.flatMap(  new FlatMapFunction<SparkFlumeEvent, String>() {  private static final long serialVersionUID = 1L;  @Override  public Iterable<String> call(SparkFlumeEvent event) throws Exception {  String line = new String(event.event().getBody().array());  return Arrays.asList(line.split(" "));  }  });    JavaPairDStream<String, Integer> pairs = words.mapToPair(  new PairFunction<String, String, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, Integer> call(String word) throws Exception {  return new Tuple2<String, Integer>(word, 1);  }  });    JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey(    new Function2<Integer, Integer, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Integer call(Integer v1, Integer v2) throws Exception {  return v1 + v2;  }  });    wordCounts.print();    jssc.start();  jssc.awaitTermination();  jssc.close();  }  } |

## 222\_Spark Streaming实战开发进阶-接收flume实时数据流-自定义sink的基于poll的方式

除了让flume将数据推送到spark streaming，还有一种方式，可以运行一个自定义的flume sink

1、Flume推送数据到sink中，然后数据缓存在sink中

2、spark streaming用一个可靠的flume receiver以及事务机制从sink中拉取数据

前提条件

1、选择一台可以在flume agent中运行自定义sink的机器

2、将flume的数据管道流配置为将数据传送到那个sink中

3、spark streaming所在的机器可以从那个sink中拉取数据

配置flume

1、加入sink jars，将以下jar加入flume的classpath中

groupId = org.apache.spark

artifactId = spark-streaming-flume-sink\_2.10

version = 1.5.1

groupId = org.scala-lang

artifactId = scala-library

version = 2.10.4

groupId = org.apache.commons

artifactId = commons-lang3

version = 3.3.2

2、修改配置文件

agent.sinks.sink1.type = org.apache.spark.streaming.flume.sink.SparkSink

agent.sinks.sink1.hostname = 192.168.0.103

agent.sinks.sink1.port = 8888

agent.sinks.sink1.channel = channel1

配置spark streaming

import org.apache.spark.streaming.flume.\*;

JavaReceiverInputDStream<SparkFlumeEvent>flumeStream =

FlumeUtils.createPollingStream(streamingContext, [sink machine hostname], [sink port]);

一定要先启动flume，再启动spark streaming

flume-ng agent -n agent1 -c conf -f /usr/local/flume/conf/flume-conf.properties -Dflume.root.logger=DEBUG,console

**FlumePollWordCount.java**

|  |
| --- |
| import org.apache.spark.SparkConf;  import org.apache.spark.api.java.function.FlatMapFunction;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import org.apache.spark.streaming.Durations;  import org.apache.spark.streaming.api.java.JavaDStream;  import org.apache.spark.streaming.api.java.JavaPairDStream;  import org.apache.spark.streaming.api.java.JavaReceiverInputDStream;  import org.apache.spark.streaming.api.java.JavaStreamingContext;  import org.apache.spark.streaming.flume.FlumeUtils;  import org.apache.spark.streaming.flume.SparkFlumeEvent;  import scala.Tuple2;  /\*\*  \* 基于Flume Poll方式的实时wordcount程序  \*/  public class FlumePollWordCount {  public static void main(String[] args) {  SparkConf conf = new SparkConf()  .setMaster("local[2]")  .setAppName("FlumePollWordCount");  JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(5));    JavaReceiverInputDStream<SparkFlumeEvent> lines =  FlumeUtils.createPollingStream(jssc, "192.168.0.103", 8888);    JavaDStream<String> words = lines.flatMap(    new FlatMapFunction<SparkFlumeEvent, String>() {  private static final long serialVersionUID = 1L;  @Override  public Iterable<String> call(SparkFlumeEvent event) throws Exception {  String line = new String(event.event().getBody().array());  return Arrays.asList(line.split(" "));  }    });    JavaPairDStream<String, Integer> pairs = words.mapToPair(    new PairFunction<String, String, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Tuple2<String, Integer> call(String word) throws Exception {  return new Tuple2<String, Integer>(word, 1);  }    });  JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey(    new Function2<Integer, Integer, Integer>() {  private static final long serialVersionUID = 1L;  @Override  public Integer call(Integer v1, Integer v2) throws Exception {  return v1 + v2;  }    });    wordCounts.print();  jssc.start();  jssc.awaitTermination();  jssc.close();  }  } |

## 223\_SparkStreaming实战开发进阶-高阶技术之自定义Receiver(结束)

|  |
| --- |
| spark streaming可以从任何数据源来接收数据，哪怕是除了它内置支持的数据源以外的其他数据源（比如flume、kafka、socket等）。  如果我们想要从spark streaming没有内置支持的数据源中接收实时数据，那么我们需要自己实现一个receiver。  实现一个自定义的receiver  一个自定义的receiver必须实现以下两个方法：onStart()、onStop()。onStart()和onStop()方法必须不能阻塞数据，一般来说，  onStart()方法会启动负责接收数据的线程，onStop()方法会确保之前启动的线程都已经停止了。负责接收数据的线程可以调用  isStopped()方法来检查它们是否应该停止接收数据。  一旦数据被接收了，就可以调用store(data)方法，数据就可以被存储在Spark内部。有一系列的store()重载方法供我们调用，来将数据  每次一条进行存储，或是每次存储一个集合或序列化的数据。  接收线程中的任何异常都应该被捕获以及正确处理，从而避免receiver的静默失败。restart()方法会通过异步地调用onStop()和  onStart()方法来重启receiver。stop()方法会调用onStop()方法来停止receiver。reportError()方法会汇报一个错误消息给driver  ，但是不停止或重启receiver。  public class JavaCustomReceiver extends Receiver<String> {  String host = null;  int port = -1;  public JavaCustomReceiver(String host\_ , int port\_) {  super(StorageLevel.MEMORY\_AND\_DISK\_2());  host = host\_;  port = port\_;  }  public void onStart() {  // Start the thread that receives data over a connection  new Thread() {  @Override public void run() {  receive();  }  }.start();  }  public void onStop() {  }  /\*\* Create a socket connection and receive data until receiver is stopped \*/  private void receive() {  Socket socket = null;  String userInput = null;  try {  // connect to the server  socket = new Socket(host, port);  BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));  // Until stopped or connection broken continue reading  while (!isStopped() && (userInput = reader.readLine()) != null) {  System.out.println("Received data '" + userInput + "'");  store(userInput);  }  reader.close();  socket.close();  // Restart in an attempt to connect again when server is active again  restart("Trying to connect again");  } catch(ConnectException ce) {  // restart if could not connect to server  restart("Could not connect", ce);  } catch(Throwable t) {  // restart if there is any other error  restart("Error receiving data", t);  }  }  }  在spark streaming中使用自定义的receiver  JavaDStream<String> lines = ssc.receiverStream(new JavaCustomReceiver(host, port));  安装网络服务工具  rpm -ihv netcat-0.7.1-1.i386.rpm  nc -lk 9999 |

## 224\_SparkStreaming实战开发进阶-kafka安装

**安装zookeeper：**

1、将课程提供的zookeeper-3.4.5.tar.gz使用WinSCP拷贝到spark1的/usr/local目录下。

2、对zookeeper-3.4.5.tar.gz进行解压缩：tar -zxvf zookeeper-3.4.5.tar.gz。

3、对zookeeper目录进行重命名：mv zookeeper-3.4.5 zk。

4、配置zookeeper相关的环境变量

vi .bashrc

export ZOOKEEPER\_HOME=/usr/local/zk

export PATH=$ZOOKEEPER\_HOME/bin

source .bashrc

**配置zoo.cfg：**

cd zk/conf

mv zoo\_sample.cfg zoo.cfg

vi zoo.cfg

修改：dataDir=/usr/local/zk/data

新增：

server.0=spark1:2888:3888

server.1=spark2:2888:3888

server.2=spark3:2888:3888

**设置zk节点标识**

cd zk

mkdir data

cd data

vi myid

0

**启动zookeeper集群：**

1、分别在三台机器上执行：zkServer.sh start。

2、检查ZooKeeper状态：zkServer.sh status。

**安装kafka**

1、将课程提供的kafka\_2.9.2-0.8.1.tgz使用WinSCP拷贝到spark1的/usr/local目录下。

2、对kafka\_2.9.2-0.8.1.tgz进行解压缩：tar -zxvf kafka\_2.9.2-0.8.1.tgz。

3、对kafka目录进行改名：mv kafka\_2.9.2-0.8.1 kafka

4、配置kafka

vi /usr/local/kafka/config/server.properties

broker.id：依次增长的整数，0、1、2、3、4，集群中Broker的唯一id

zookeeper.connect=192.168.1.107:2181,192.168.1.108:2181,192.168.1.109:2181

5、安装slf4j

将课程提供的slf4j-1.7.6.zip上传到/usr/local目录下

unzip slf4j-1.7.6.zip

把slf4j中的slf4j-nop-1.7.6.jar复制到kafka的libs目录下面

**搭建kafka集群：**

1、按照上述步骤在spark2和spark3分别安装kafka。用scp把kafka拷贝到spark2和spark3行即可。

2、唯一区别的，就是server.properties中的broker.id，要设置为1和2

**启动kafka集群**

1、在三台机器上分别执行以下命令：

nohup kafka-server-start.sh /usr/local/kafka/config/server.properties &

2、解决kafka Unrecognized VM option 'UseCompressedOops'问题

vi bin/kafka-run-class.sh

if [ -z "$KAFKA\_JVM\_PERFORMANCE\_OPTS" ]; then

KAFKA\_JVM\_PERFORMANCE\_OPTS="-server -XX:+UseCompressedOops -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -XX:+CMSScavengeBeforeRemark -XX:+DisableExplicitGC -Djava.awt.headless=true"

fi

去掉-XX:+UseCompressedOops即可

3、使用jps检查启动是否成功

**测试kafka集群**

使用基本命令检查kafka是否搭建成功

kafka-topics.sh --zookeeper 192.168.0.103:2181,192.168.0.104:2181 --topic TestTopic --replication-factor 1 --partitions 1 --create

kafka-console-producer.sh --broker-list 192.168.0.103:9092,192.168.0.104:9092 --topic TestTopic

kafka-console-consumer.sh --zookeeper 192.168.0.103:2181,192.168.0.104:2181 --topic TestTopic --from-beginning

## 225\_Spark Streaming实战开发进阶-综合案例3：新闻网站关键指标实时统计

kafka-topics.sh --zookeeper 192.168.0.103:2181,192.168.0.104:2181 --topic news-access --replication-factor 1 --partitions 1 --create

kafka-console-consumer.sh --zookeeper 192.168.0.103:2181,192.168.0.104:2181 --topic news-access --from-beginning

## 226\_Spark Streaming实战开发进阶-综合案例3：页面pv实时统计

**AccessProducer.java生成测试数据**

|  |
| --- |
| package cn.spark.study.streaming.upgrade.news;  import java.text.SimpleDateFormat;  import java.util.Date;  import java.util.Properties;  import java.util.Random;  import kafka.javaapi.producer.Producer;  import kafka.producer.KeyedMessage;  import kafka.producer.ProducerConfig;  /\*\*访问日志Kafka Producer\*/  public class AccessProducer extends Thread {    private static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  private static Random random = new Random();  private static String[] sections = new String[] {"country", "international", "sport", "entertainment", "movie", "carton", "tv-show", "technology", "internet", "car"};  private static int[] arr = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  private static String date;    private Producer<Integer, String> producer;  private String topic;    public AccessProducer(String topic) {  this.topic = topic;  producer = new Producer<Integer, String>(createProducerConfig());  date = sdf.format(new Date());  }    private ProducerConfig createProducerConfig() {  Properties props = new Properties();  props.put("serializer.class", "kafka.serializer.StringEncoder");  props.put("metadata.broker.list", "192.168.0.103:9092,192.168.0.104:9092");  return new ProducerConfig(props);  }    public void run() {  int counter = 0;  while(true) {  for(int i = 0; i < 100; i++) {  String log = null;    if(arr[random.nextInt(10)] == 1) {  log = getRegisterLog();  } else {  log = getAccessLog();  }    //实际测试这里参数是个seq需要java转scala seq：**producer**.send(JavaConverters.*asScalaIteratorConverter*(Arrays.*asList*(**new** KeyedMessage<Integer,String>(**topic**,log)).iterator()).asScala().toSeq());  producer.send(new KeyedMessage<Integer, String>(topic, log));    counter++;  if(counter == 100) {  counter = 0;  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }  }    private static String getAccessLog() {  StringBuffer buffer = new StringBuffer("");    // 生成时间戳  long timestamp = new Date().getTime();    // 生成随机userid（默认1000注册用户，每天1/10的访客是未注册用户）  Long userid = 0L;    int newOldUser = arr[random.nextInt(10)];  if(newOldUser == 1) {  userid = null;  } else {  userid = (long) random.nextInt(1000);  }    // 生成随机pageid（总共1k个页面）  Long pageid = (long) random.nextInt(1000);    // 生成随机版块（总共10个版块）  String section = sections[random.nextInt(10)];    // 生成固定的行为，view  String action = "view";    return buffer.append(date).append(" ")  .append(timestamp).append(" ")  .append(userid).append(" ")  .append(pageid).append(" ")  .append(section).append(" ")  .append(action).toString();  }    private static String getRegisterLog() {  StringBuffer buffer = new StringBuffer("");    // 生成时间戳  long timestamp = new Date().getTime();    // 新用户都是userid为null  Long userid = null;  // 生成随机pageid，都是null  Long pageid = null;    // 生成随机版块，都是null  String section = null;    // 生成固定的行为，view  String action = "register";    return buffer.append(date).append(" ")  .append(timestamp).append(" ")  .append(userid).append(" ")  .append(pageid).append(" ")  .append(section).append(" ")  .append(action).toString();  }    public static void main(String[] args) {  AccessProducer producer = new AccessProducer("news-access");  producer.start();  }    } |

**NewsRealtimeStatSpark.java**

|  |
| --- |
| package cn.spark.study.streaming.upgrade.news;  import java.util.HashMap;  import java.util.HashSet;  import java.util.Map;  import java.util.Set;  import kafka.serializer.StringDecoder;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.function.Function;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import org.apache.spark.streaming.Durations;  import org.apache.spark.streaming.api.java.JavaPairDStream;  import org.apache.spark.streaming.api.java.JavaPairInputDStream;  import org.apache.spark.streaming.api.java.JavaStreamingContext;  import org.apache.spark.streaming.kafka.KafkaUtils;  import scala.Tuple2;  /\*\*  \* 新闻网站关键指标实时统计Spark应用程序  \* @author Administrator  \*  \*/  public class NewsRealtimeStatSpark {  public static void main(String[] args) throws Exception {  // 创建Spark上下文  SparkConf conf = new SparkConf()  .setMaster("local[2]")  .setAppName("NewsRealtimeStatSpark");  JavaStreamingContext jssc = new JavaStreamingContext(  conf, Durations.seconds(5));    // 创建输入DStream  Map<String, String> kafkaParams = new HashMap<String, String>();  kafkaParams.put("metadata.broker.list",  "192.168.0.103:9092,192.168.0.104:9092");    Set<String> topics = new HashSet<String>();  topics.add("news-access");    JavaPairInputDStream<String, String> lines = KafkaUtils.createDirectStream(  jssc,  String.class,  String.class,  StringDecoder.class,  StringDecoder.class,  kafkaParams,  topics);    // 过滤出访问日志  JavaPairDStream<String, String> accessDStream = lines.filter(    new Function<Tuple2<String,String>, Boolean>() {    private static final long serialVersionUID = 1L;    @Override  public Boolean call(Tuple2<String, String> tuple) throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    String action = logSplited[5];  if("view".equals(action)) {  return true;  } else {  return false;  }  }    });    // 统计第一个指标：每10秒内的各个页面的pv  calculatePagePv(accessDStream);    jssc.start();  jssc.awaitTermination();  jssc.close();  }    /\*\*  \* 计算页面pv  \* @param accessDStream  \*/  private static void calculatePagePv(JavaPairDStream<String, String> accessDStream) {  JavaPairDStream<Long, Long> pageidDStream = accessDStream.mapToPair(    new PairFunction<Tuple2<String,String>, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<Long, Long> call(Tuple2<String, String> tuple)  throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    Long pageid = Long.valueOf(logSplited[3]);    return new Tuple2<Long, Long>(pageid, 1L);  }    });    JavaPairDStream<Long, Long> pagePvDStream = pageidDStream.reduceByKey(    new Function2<Long, Long, Long>() {    private static final long serialVersionUID = 1L;    @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    pagePvDStream.print();    // 在计算出每10秒钟的页面pv之后，其实在真实项目中，应该持久化  // 到mysql，或redis中，对每个页面的pv进行累加  // javaee系统，就可以从mysql或redis中，读取page pv实时变化的数据，以及曲线图  }  } |

## 227\_Spark Streaming实战开发进阶-综合案例3：页面uv实时统计

NewsRealtimeStatSpark.java后续代码统一在330章节

|  |
| --- |
| import java.util.HashMap;  import java.util.HashSet;  import java.util.Map;  import java.util.Set;  import kafka.serializer.StringDecoder;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.function.Function;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import org.apache.spark.streaming.Durations;  import org.apache.spark.streaming.api.java.JavaDStream;  import org.apache.spark.streaming.api.java.JavaPairDStream;  import org.apache.spark.streaming.api.java.JavaPairInputDStream;  import org.apache.spark.streaming.api.java.JavaStreamingContext;  import org.apache.spark.streaming.kafka.KafkaUtils;  import scala.Tuple2;  /\*\*  \* 新闻网站关键指标实时统计Spark应用程序  \* @author Administrator  \*  \*/  public class NewsRealtimeStatSpark {  public static void main(String[] args) throws Exception {  // 创建Spark上下文  SparkConf conf = new SparkConf()  .setMaster("local[2]")  .setAppName("NewsRealtimeStatSpark");  JavaStreamingContext jssc = new JavaStreamingContext(  conf, Durations.seconds(5));    // 创建输入DStream  Map<String, String> kafkaParams = new HashMap<String, String>();  kafkaParams.put("metadata.broker.list",  "192.168.0.103:9092,192.168.0.104:9092");    Set<String> topics = new HashSet<String>();  topics.add("news-access");    JavaPairInputDStream<String, String> lines = KafkaUtils.createDirectStream(  jssc,  String.class,  String.class,  StringDecoder.class,  StringDecoder.class,  kafkaParams,  topics);    // 过滤出访问日志  JavaPairDStream<String, String> accessDStream = lines.filter(    new Function<Tuple2<String,String>, Boolean>() {    private static final long serialVersionUID = 1L;    @Override  public Boolean call(Tuple2<String, String> tuple) throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    String action = logSplited[5];  if("view".equals(action)) {  return true;  } else {  return false;  }  }    });    // 统计第一个指标：每10秒内的各个页面的pv  // calculatePagePv(accessDStream);  // 统计第二个指标：每10秒内的各个页面的uv  calculatePageUv(accessDStream);    jssc.start();  jssc.awaitTermination();  jssc.close();  }    /\*\*  \* 计算页面pv  \* @param accessDStream  \*/  private static void calculatePagePv(JavaPairDStream<String, String> accessDStream) {  JavaPairDStream<Long, Long> pageidDStream = accessDStream.mapToPair(    new PairFunction<Tuple2<String,String>, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<Long, Long> call(Tuple2<String, String> tuple)  throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    Long pageid = Long.valueOf(logSplited[3]);    return new Tuple2<Long, Long>(pageid, 1L);  }    });    JavaPairDStream<Long, Long> pagePvDStream = pageidDStream.reduceByKey(    new Function2<Long, Long, Long>() {    private static final long serialVersionUID = 1L;    @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    pagePvDStream.print();    // 在计算出每10秒钟的页面pv之后，其实在真实项目中，应该持久化  // 到mysql，或redis中，对每个页面的pv进行累加  // javaee系统，就可以从mysql或redis中，读取page pv实时变化的数据，以及曲线图  }    /\*\*  \* 计算页面uv  \* @param <U>  \* @param accessDStream  \*/  private static <U> void calculatePageUv(JavaPairDStream<String, String> accessDStream) {  JavaDStream<String> pageidUseridDStream = accessDStream.map(    new Function<Tuple2<String,String>, String>() {  private static final long serialVersionUID = 1L;  @Override  public String call(Tuple2<String, String> tuple) throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    Long pageid = Long.valueOf(logSplited[3]);  Long userid = Long.valueOf("null".equalsIgnoreCase(logSplited[2]) ? "-1" : logSplited[2]);    return pageid + "\_" + userid;  }    });    JavaDStream<String> distinctPageidUseridDStream = pageidUseridDStream.transform(    new Function<JavaRDD<String>, JavaRDD<String>>() {  private static final long serialVersionUID = 1L;    @Override  public JavaRDD<String> call(JavaRDD<String> rdd) throws Exception {  return rdd.distinct();  }    });    JavaPairDStream<Long, Long> pageidDStream = distinctPageidUseridDStream.mapToPair(    new PairFunction<String, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<Long, Long> call(String str) throws Exception {  String[] splited = str.split("\_");  Long pageid = Long.valueOf(splited[0]);  return new Tuple2<Long, Long>(pageid, 1L);  }    });    JavaPairDStream<Long, Long> pageUvDStream = pageidDStream.reduceByKey(    new Function2<Long, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    pageUvDStream.print();  }    } |

## 228\_SparkStreaming实战开发进阶-综合案例3：注册用户数实时统计

## 229\_Spark Streaming实战开发进阶-综合案例3：用户跳出量实时统计

## 230\_Spark Streaming实战开发进阶-综合案例3：版块pv实时统计

|  |
| --- |
| package cn.spark.study.streaming.upgrade.news;  import java.util.HashMap;  import java.util.HashSet;  import java.util.Map;  import java.util.Set;  import kafka.serializer.StringDecoder;  import org.apache.spark.SparkConf;  import org.apache.spark.api.java.JavaRDD;  import org.apache.spark.api.java.function.Function;  import org.apache.spark.api.java.function.Function2;  import org.apache.spark.api.java.function.PairFunction;  import org.apache.spark.streaming.Durations;  import org.apache.spark.streaming.api.java.JavaDStream;  import org.apache.spark.streaming.api.java.JavaPairDStream;  import org.apache.spark.streaming.api.java.JavaPairInputDStream;  import org.apache.spark.streaming.api.java.JavaStreamingContext;  import org.apache.spark.streaming.kafka.KafkaUtils;  import scala.Tuple2;  /\*\*  \* 新闻网站关键指标实时统计Spark应用程序  \* @author Administrator  \*  \*/  public class NewsRealtimeStatSpark {  public static void main(String[] args) throws Exception {  // 创建Spark上下文  SparkConf conf = new SparkConf()  .setMaster("local[2]")  .setAppName("NewsRealtimeStatSpark");  JavaStreamingContext jssc = new JavaStreamingContext(  conf, Durations.seconds(5));    // 创建输入DStream  Map<String, String> kafkaParams = new HashMap<String, String>();  kafkaParams.put("metadata.broker.list",  "192.168.0.103:9092,192.168.0.104:9092");    Set<String> topics = new HashSet<String>();  topics.add("news-access");    JavaPairInputDStream<String, String> lines = KafkaUtils.createDirectStream(  jssc,  String.class,  String.class,  StringDecoder.class,  StringDecoder.class,  kafkaParams,  topics);    // 过滤出访问日志  JavaPairDStream<String, String> accessDStream = lines.filter(    new Function<Tuple2<String,String>, Boolean>() {    private static final long serialVersionUID = 1L;    @Override  public Boolean call(Tuple2<String, String> tuple) throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    String action = logSplited[5];  if("view".equals(action)) {  return true;  } else {  return false;  }  }    });    // 统计第一个指标：实时页面pv  calculatePagePv(accessDStream);  // 统计第二个指标：实时页面uv  calculatePageUv(accessDStream);  // 统计第三个指标：实时注册用户数  calculateRegisterCount(lines);  // 统计第四个指标：实时用户跳出数  calculateUserJumpCount(accessDStream);  // 统计第五个指标：实时版块pv  calcualteSectionPv(accessDStream);    jssc.start();  jssc.awaitTermination();  jssc.close();  }    /\*\*  \* 计算页面pv  \* @param accessDStream  \*/  private static void calculatePagePv(JavaPairDStream<String, String> accessDStream) {  JavaPairDStream<Long, Long> pageidDStream = accessDStream.mapToPair(    new PairFunction<Tuple2<String,String>, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<Long, Long> call(Tuple2<String, String> tuple)  throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    Long pageid = Long.valueOf(logSplited[3]);    return new Tuple2<Long, Long>(pageid, 1L);  }    });    JavaPairDStream<Long, Long> pagePvDStream = pageidDStream.reduceByKey(    new Function2<Long, Long, Long>() {    private static final long serialVersionUID = 1L;    @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    pagePvDStream.print();    // 在计算出每10秒钟的页面pv之后，其实在真实项目中，应该持久化  // 到mysql，或redis中，对每个页面的pv进行累加  // javaee系统，就可以从mysql或redis中，读取page pv实时变化的数据，以及曲线图  }    /\*\*  \* 计算页面uv  \* @param <U>  \* @param accessDStream  \*/  private static <U> void calculatePageUv(JavaPairDStream<String, String> accessDStream) {  JavaDStream<String> pageidUseridDStream = accessDStream.map(    new Function<Tuple2<String,String>, String>() {  private static final long serialVersionUID = 1L;  @Override  public String call(Tuple2<String, String> tuple) throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    Long pageid = Long.valueOf(logSplited[3]);  Long userid = Long.valueOf("null".equalsIgnoreCase(logSplited[2]) ? "-1" : logSplited[2]);    return pageid + "\_" + userid;  }    });    JavaDStream<String> distinctPageidUseridDStream = pageidUseridDStream.transform(    new Function<JavaRDD<String>, JavaRDD<String>>() {  private static final long serialVersionUID = 1L;    @Override  public JavaRDD<String> call(JavaRDD<String> rdd) throws Exception {  return rdd.distinct();  }    });    JavaPairDStream<Long, Long> pageidDStream = distinctPageidUseridDStream.mapToPair(    new PairFunction<String, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<Long, Long> call(String str) throws Exception {  String[] splited = str.split("\_");  Long pageid = Long.valueOf(splited[0]);  return new Tuple2<Long, Long>(pageid, 1L);  }    });    JavaPairDStream<Long, Long> pageUvDStream = pageidDStream.reduceByKey(    new Function2<Long, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    pageUvDStream.print();  }    /\*\*  \* 计算实时注册用户数  \* @param lines  \*/  private static void calculateRegisterCount(JavaPairInputDStream<String, String> lines) {  JavaPairDStream<String, String> registerDStream = lines.filter(    new Function<Tuple2<String,String>, Boolean>() {  private static final long serialVersionUID = 1L;    @Override  public Boolean call(Tuple2<String, String> tuple) throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    String action = logSplited[5];  if("register".equals(action)) {  return true;  } else {  return false;  }  }    });    JavaDStream<Long> registerCountDStream = registerDStream.count();    registerCountDStream.print();    // 每次统计完一个最近10秒的数据之后，不是打印出来  // 去存储（mysql、redis、hbase），选用哪一种主要看你的公司提供的环境，以及你的看实时报表的用户以及并发数量，包括你的数据量  // 如果是一般的展示效果，就选用mysql就可以  // 如果是需要超高并发的展示，比如QPS 1w来看实时报表，那么建议用redis、memcached  // 如果是数据量特别大，建议用hbase    // 每次从存储中，查询注册数量，最近一次插入的记录，比如上一次是10秒前  // 然后将当前记录与上一次的记录累加，然后往存储中插入一条新记录，就是最新的一条数据  // 然后javaee系统在展示的时候，可以比如查看最近半小时内的注册用户数量变化的曲线图  // 查看一周内，每天的注册用户数量的变化曲线图（每天就取最后一条数据，就是每天的最终数据）  }  /\*\*  \* 计算用户跳出数量  \* @param accessDStream  \*/  private static void calculateUserJumpCount(JavaPairDStream<String, String> accessDStream) {  JavaPairDStream<Long, Long> useridDStream = accessDStream.mapToPair(    new PairFunction<Tuple2<String,String>, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<Long, Long> call(Tuple2<String, String> tuple)  throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");  Long userid = Long.valueOf("null".equalsIgnoreCase(logSplited[2]) ? "-1" : logSplited[2]);  return new Tuple2<Long, Long>(userid, 1L);  }    });    JavaPairDStream<Long, Long> useridCountDStream = useridDStream.reduceByKey(    new Function2<Long, Long, Long>() {  private static final long serialVersionUID = 1L;  @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    JavaPairDStream<Long, Long> jumpUserDStream = useridCountDStream.filter(    new Function<Tuple2<Long,Long>, Boolean>() {  private static final long serialVersionUID = 1L;  @Override  public Boolean call(Tuple2<Long, Long> tuple) throws Exception {  if(tuple.\_2 == 1) {  return true;  } else {  return false;  }  }    });    JavaDStream<Long> jumpUserCountDStream = jumpUserDStream.count();    jumpUserCountDStream.print();  }    /\*\*  \* 版块实时pv  \* @param accessDStream  \*/  private static void calcualteSectionPv(JavaPairDStream<String, String> accessDStream) {  JavaPairDStream<String, Long> sectionDStream = accessDStream.mapToPair(    new PairFunction<Tuple2<String,String>, String, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Tuple2<String, Long> call(Tuple2<String, String> tuple)  throws Exception {  String log = tuple.\_2;  String[] logSplited = log.split(" ");    String section = logSplited[4];    return new Tuple2<String, Long>(section, 1L);  }    });    JavaPairDStream<String, Long> sectionPvDStream = sectionDStream.reduceByKey(    new Function2<Long, Long, Long>() {  private static final long serialVersionUID = 1L;    @Override  public Long call(Long v1, Long v2) throws Exception {  return v1 + v2;  }    });    sectionPvDStream.print();  }    } |

## 231\_Spark运维管理进阶-基于ZooKeeper实现HA高可用性以及自动主备切换

默认情况下，standalone cluster manager对于worker节点的失败是具有容错性的（迄今为止，Spark自身而言对于丢失部分计算工作

是有容错性的，它会将丢失的计算工作迁移到其他worker节点上执行）。然而，调度器是依托于master进程来做出调度决策的，这就会

造成单点故障：如果master挂掉了，就没法提交新的应用程序了。为了解决这个问题，spark提供了两种高可用性方案，分别是基于

zookeeper的HA方案以及基于文件系统的HA方案。

基于zookeeper的HA方案

概述

使用zookeeper来提供leader选举以及一些状态存储，你可以在集群中启动多个master进程，让它们连接到zookeeper实例。其中

一个master进程会被选举为leader，其他的master会被指定为standby模式。如果当前的leader master进程挂掉了，其他的

standby master会被选举，从而恢复旧master的状态。并且恢复作业调度。整个恢复过程（从leader master挂掉开始计算）大概

会花费1~2分钟。要注意的是，这只会推迟调度新的应用程序，master挂掉之前就运行的应用程序是不被影响的。

配置

如果要启用这个恢复模式，需要在spark-env.sh文件中，设置SPARK\_DAEMON\_JAVA\_OPTS选项：

spark.deploy.recoveryMode 设置为ZOOKEEPER来启用standby master恢复模式（默认为NONE）

spark.deploy.zookeeper.url zookeeper集群url（举例来说，192.168.0.103:2181,192.168.0.104:2181）

spark.deploy.zookeeper.dir zookeeper中用来存储恢复状态的目录（默认是/spark）

备注：如果在集群中启动了多个master节点，但是没有正确配置master去使用zookeeper，master在挂掉进行恢复时是会失败的，

因为没法发现其他master，并且都会认为自己是leader。这会导致集群的状态不是健康的，因为所有master都会自顾自地去调度。

细节

在启动一个zookeeper集群之后，启用高可用性是很直接的。简单地在多个节点上启动多个master进程，并且给它们相同的zookeeper

配置（zookeeper url和目录）。master就可以被动态加入master集群，并可以在任何时间被移除掉。

为了调度新的应用程序或者向集群中添加worker节点，它们需要知道当前leader master的ip地址。这可以通过传递一个master列表

来完成。举例来说，我们可以将我们的SparkContext连接的地址指向spark://host1:port1,host2:port2。这就会导致你的

SparkContext尝试去注册所有的master，如果host1挂掉了，那么配置还是正确的，因为会找到新的leader master，也就是host2。

对于注册一个master和普通的操作，这是一个重要的区别。当一个应用程序启动的时候，或者worker需要被找到并且注册到当前

的leader master的时候。一旦它成功注册了，就被保存在zookeeper中了。如果故障发生了，new leader master会去联系所有的

之前注册过的应用程序和worker，并且通知它们master的改变。这样的话，它们甚至在启动的时候都不需要知道new master的存在。

正是由于这个属性，new master可以在任何时间被创建，并且我们唯一需要担心的一件事情就是新的应用程序和worker可以找到

并且注册到master。一旦注册上去之后，我们就不用担心它了。

## 232\_Spark运维管理进阶-实验：基于ZooKeeper实现HA高可用性以及自动主备切换

实验

1、将192.168.0.103机器上的spark集群先停止

./sbin/stop-all.sh

2、修改机器上的spark-env.sh文件，在其中加入上述三个属性

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=192.168.0.103:2181,192.168.0.104:2181 -Dspark.deploy.zookeeper.dir=/spark"

3、启动集群

在192.168.0.103上直接用启动集群：./sbin/start-all.sh

4、在192.168.0.104上部署spark安装包，并启动一个master进程

安装scala 2.11.4

1、将课程提供的scala-2.11.4.tgz使用WinSCP拷贝到/usr/local/src目录下。

2、对scala-2.11.4.tgz进行解压缩：tar -zxvf scala-2.11.4.tgz

3、对scala目录进行重命名：mv scala-2.11.4 scala

4、配置scala相关的环境变量

vi ~/.bashrc

export SCALA\_HOME=/usr/local/scala

export PATH=$SCALA\_HOME/bin

source ~/.bashrc

5、查看scala是否安装成功：scala -version

安装spark客户端

1、将spark-1.5.1-bin-hadoop2.4.tgz使用WinSCP上传到/usr/local/src目录下。

2、解压缩spark包：tar -zxvf spark-1.5.1-bin-hadoop2.4.tgz。

3、重命名spark目录：mv spark-1.5.1-bin-hadoop2.4 spark

4、修改spark环境变量

vi ~/.bashrc

export SPARK\_HOME=/usr/local/spark

export PATH=$SPARK\_HOME/bin

export CLASSPATH=.:$CLASSPATH:$JAVA\_HOME/lib:$JAVA\_HOME/jre/lib

source ~/.bashrc

修改spark-env.sh文件

1、cd /usr/local/spark/conf

2、cp spark-env.sh.template spark-env.sh

3、vi spark-env.sh

export JAVA\_HOME=/usr/java/latest

export SCALA\_HOME=/usr/local/scala

export HADOOP\_HOME=/usr/local/hadoop

export HADOOP\_CONF\_DIR=/usr/local/hadoop/etc/hadoop

export SPARK\_MASTER\_IP=192.168.0.104

export SPARK\_DAEMON\_MEMORY=100m

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=192.168.0.103:2181,192.168.0.104:2181 -Dspark.deploy.zookeeper.dir=/spark"

在192.168.0.104上单独启动一个standby master进程：./sbin/start-master.sh

4、提交应用程序

将master地址修改为192.168.0.103:7077,192.168.0.103:7078

5、杀掉原先的leader master，等到standby master接管集群

再次提交应用程序

6、再次手动启动原来的leader master（死掉）

## 233\_Spark运维管理进阶-基于文件系统实现HA高可用性以及手动主备切换

概述

zookeeper是实现生产级别的高可用性的最佳方式，但是如果你就是想要在master进程挂掉的时候，手动去重启它，而不是依靠

zookeeper实现自动主备切换，那么可以使用FILESYSTEM模式。当应用程序和worker都注册到master之后，master就会将它们的信息

写入指定的文件系统目录中，以便于当master重启的时候可以从文件系统中恢复注册的应用程序和worker状态。

配置

要启用这种恢复模式，需要在spark-env.sh中设置SPARK\_DAEMON\_JAVA\_OPTS

spark.deploy.recoveryMode 设置为FILESYSTEM来启用单点恢复（默认值为NONE）

spark.deploy.recoveryDirectory spark在哪个文件系统目录内存储状态信息，必须是master可以访问的目录

细节

1、这个解决方案可以与进程监控或管理器（比如monit）结合使用，或者就仅仅是启用手动重启恢复机制即可。

2、文件系统恢复比不做任何恢复机制肯定是要好的，这个模式更加适合于开发和测试环境，而不是生产环境。此外，通过

stop-master.sh脚本杀掉一个master进程是不会清理它的恢复状态的，所以当你重启一个新的master进程时，它会进入恢复模式。

这会增加你的恢复时间至少1分钟，因为它需要等待之前所有已经注册的worker等节点先timeout。

3、这种方式没有得到官方的支持，也可以使用一个NFS目录作为恢复目录。如果原先的master节点完全死掉了，你可以在其他节点

上启动一个master进程，它会正确地恢复之前所有注册的worker和应用程序。之后的应用程序可以找到新的master，然后注册。

## 234\_Spark运维管理进阶-实验：基于文件系统实现HA高可用性以及手动主备切换

1、关闭两台机器上的master和worker

2、修改192.168.0.103机器上的spark-env.sh

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=FILESYSTEM -Dspark.deploy.recoveryDirectory=/usr/local/spark\_recovery"

3、在192.168.0.103上启动spark集群

4、在spark-shell中进行wordcount计数，到一半，有一个running application

5、杀掉master进程

6、重启master进程

7、观察web ui上，是否恢复了worker以及原先正在运行的application

## 235\_Spark运维管理进阶-作业监控-SparkWebUI以及监控实验

对于Spark作业的监控，Spark给我们提供了很多种方式：Spark Web UI，Spark History Web UI，RESTFUL API以及Metrics。

这里我们就先来看看如何通过Spark Web UI监控作业的运行。

Spark Web UI

每提交一个Spark作业，并且启动SparkContext之后，都会启动一个对应的Spark Web UI服务。默认情况下Spark Web UI的访问地址是

driver进程所在节点的4040端口。在Spark Web UI上会展示作业相关的详细信息，非常有用，是Spark作业监控的最主要的手段。

Spark Web UI包括了以下信息：

1、stage和task列表

2、RDD大小以及内存使用的概览

3、环境信息

4、作业对应的executor的信息

可以通过在浏览器中访问http://<driver-node>:4040地址，来进入Spark Web UI界面。如果多个driver在一个机器上运行，它们

会自动绑定到不同的端口上。默认从4040端口开始，如果发现已经被绑定了，那么会选择4041、4042等端口，以此类推。

要注意的是，这些信息默认情况下仅仅在作业运行期间有效并且可以看到。一旦作业完毕，那么driver进程以及对应的web ui服务

也会停止，我们就无法看到已经完成的作业的信息了。如果要在作业完成之后，也可以看到其Spark Web UI以及详细信息，那么就

需要启用Spark的History Server。

监控实验

1、通过spark-shell以standalone模式执行一个wordcount作业，通过直接访问4040端口以及从8080端口两种方式进入web ui。

2、在作业运行完毕之后，再尝试看看作业的Web UI。

3、通过spark-shell以yarn模式执行一个wordcount作业，并重复上述过程。

## 236\_Spark运维管理进阶-作业监控-实验：standalone模式下查看历史作业的Web UI

默认情况下，一个作业运行完成之后，就再也无法看到其web ui以及执行信息了，在生产环境中，这对调试以及故障定位有影响。

如果要在作业执行完之后，还能看到其web ui，那么必须将作业的spark.eventLog.enabled属性设置为true，这个属性会告诉spark

去记录该作业的所有要在web ui上展示的事件以及信息。

如果spark记录下了一个作业生命周期内的所有事件，那么就会在该作业执行完成之后，我们进入其web ui时，自动用记录的数据

重新绘制作业的web ui。

有3个属性我们可以设置

1、spark.eventLog.enabled，必须设置为true

2、spark.eventLog.dir，默认是/tmp/spark-events，建议自己手动调整为其他目录，比如/usr/local/spark-event或是hdfs目录，必须手动创建

3、spark.eventLog.compress ，是否压缩数据，默认为false，建议可以开启压缩以减少磁盘空间占用

这些属性可以在提交一个作业的时候设置

如果想要对所有作业都启用该机制，那么可以在spark-defaults.conf文件中配置这三个属性

实验

1、先看看之前的已经执行完成的作业，是否可以进入spark web ui界面

2、关闭现有的master和worker进程

3、修改spark-defaults.conf文件，配置上述三个属性，启用standalone模式下的作业历史信息记录，手动创建hdfs目录

4、重新启动spark集群

5、使用spark-shell提交一个作业，然后再次尝试进入spark web ui界面

注意：如果要让spark完成作业的事件记录，那么必须最后以sc.stop()结尾。

## 237\_Spark运维管理进阶-作业监控-实验：启动HistoryServer查看历史作业的Web UI

spark-defaults.conf

spark.eventLog.enabled true

spark.eventLog.dir hdfs://192.168.0.103:9000/spark-events

spark.eventLog.compress true

spark-env.sh

export SPARK\_HISTORY\_OPTS="-Dspark.history.ui.port=18080 -Dspark.history.retainedApplications=50 -Dspark.history.fs.logDirectory=hdfs://192.168.0.103:9000/spark-events"

务必预先创建好hdfs://192.168.0.103:9000/spark-events目录

而且要注意，spark.eventLog.dir与spark.history.fs.logDirectory指向的必须是同一个目录

因为spark.eventLog.dir会指定作业事件记录在哪里，spark.history.fs.logDirectory会指定从哪个目录中去读取作业数据

启动HistoryServer: ./sbin/start-history-server.sh

访问地址: 192.168.0.103:18080

实验

1、停止集群

2、配置spark-env.sh

3、重启集群

4、启动history server

5、运行spark-shell，在standalone模式下和yarn模式下，分别执行一个作业

6、通过192.168.80.103:18080的HistoryServer UI可以看到所有运行后的作业信息

## 238\_Spark运维管理进阶-作业监控-实验：使用curl+REST API进行作业监控

除了查看ui上的统计来监控作业，还可以通过Spark提供的REST API来获取作业信息，并进行作业监控。REST API就给我们自己开发

Spark的一些监控系统或平台提供了可能。REST API是通过http协议发送的，并给我们返回JSON格式的数据。因此无论你是用java，

还是python，亦或是php，都可以获取Spark的监控信息。

运行中的作业以及history server中的历史作业，都可以获取到信息

1、如果是要获取运行中的作业的信息，可以通过http://host:4040/api/v1/...的方式来获取

2、如果是要获取历史作业的信息，可以通过http://host:18080/api/v1/...的方式来获取

比如说，http://192.168.0.103:18080/api/v1/applications，就可以获取到所有历史作业的基本信息

以下是所有API的说明

/applications 获取作业列表

/applications/[app-id]/jobs 指定作业的job列表

/applications/[app-id]/jobs/[job-id] 指定job的信息

/applications/[app-id]/stages 指定作业的stage列表

/applications/[app-id]/stages/[stage-id] 指定stage的所有attempt列表

/applications/[app-id]/stages/[stage-id]/[stage-attempt-id] 指定stage attempt的信息

/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskSummary 指定stage attempt所有task的metrics统计信息

/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskList 指定stage attempt的task列表

/applications/[app-id]/executors 指定作业的executor列表

/applications/[app-id]/storage/rdd 指定作业的持久化rdd列表

/applications/[app-id]/storage/rdd/[rdd-id] 指定持久化rdd的信息

/applications/[app-id]/logs 下载指定作业的所有日志的压缩包

/applications/[app-id]/[attempt-id]/logs 下载指定作业的某次attempt的所有日志的压缩包

当作业运行在yarn中时，每个作业都可能会尝试多次运行，所以上述的所有[app-id]都必须替换为[app-id]/[attempt-id]

这些API都非常便于让我们去基于它们开发各种监控系统或应用。特别是，spark保证以下几点:

1、API永远不会因为版本的变更而更改

2、JSON中的字段用于不会被移除

3、新的API接口可能会被增加

4、已有API接口中可能会增加新的字段

5、API的新版本可能会作为新接口被添加进来。新版本的接口不要求向后兼容。

6、API版本可能会被删除掉，但是肯定是在一个相关的新API版本发布之后。

要注意的是，当查看运行中作业的UI时，applications/[app-id]还是需要提供的，尽管此时在那个4040端口上可能只有一个

作业在运行。比如说，要查看正在运行的作业的job列表，可能需要使用以下API: http://host:4040/api/v1/applications/[app-id]/jobs

这主要是为了尽可能地复用API接口

实验

1、安装curl工具，来发送http请求: yum install -y curl

2、试一试以上的几个API，去获取standalone模式和yarn模式运行中的作业，以及历史作业的信息

## 239\_Spark运维管理进阶-作业监控-实验：Spark Metrics系统以及自定义Metrics Sink

Spark有一套可配置的metrics系统，是基于Coda Hale Metrics类库实现的。该metrics系统允许用户将Spark的metrics统计指标

上报到多种目标源（sink）中，包括http，jmx和csv文件。这个metrics系统是通过一个配置文件进行配置的，在$SPARK\_HOME目录

的conf目录下，用一个metrics.properties文件来配置。可以通过在spark-defaults.conf中配置spark.metrics.conf属性来配置

自定义的文件路径。spark metrics依据不同的spark组件划分为了不同的实例。在每一个实例中，你都可以配置一系列的sink来

指定该实例的metrics要上报到哪里去。

以下实例是目前被支持的

master: spark standalone master进程

applications: master中的组件，可以上报所有application的metrics

worker: spark standalone worker进程

executor: spark executor进程

driver: spark driver进程

每个实例都可以上报metrics到0个或多个sink中。sink被包含在了org.apache.spark.metrics.sink包下。

ConsoleSink: 日志metrics，打印到控制台

CSVSink: 以固定的频率将metrics数据导出到CSV文件中

JmxSink: 注册metrics到JMX console中

MetricsServlet: 在Spark UI中添加一个servlet来通过JSON数据提供metrics数据（之前的REST API就是通过该方式进行的）

Slf4jSink: 以日志的形式发送metrics到slf4j

GraphiteSink: 发送metrics到Graphite节点

Spark也支持Ganglia sink，但是没有包含在默认的打包内，因为有版权的问题。

GangliaSink: 发送metrics到Ganglia节点。

要安装GangliaSink，就需要自己编译一个spark。要注意，必须要提供必要的授权信息。

metrics系统的意义

1、metrics只能在spark web ui上看到，或者是history server上看历史作业的web ui。

2、如果你希望将metrics数据，结构化处理以后导入到，比如mysql里面，然后进行一个存储，开发一个系统对外开放

3、spark集群运行分析系统

4、自定义metrics sink，将所有的metrics全部写入外部的你指定的存储文件中，然后定时导入到你的mysql中

实验: 自定义metrics sink

1、停止集群

2、配置spark.metrics.conf文件，启用CSVSink

3、重启集群

4、运行一个作业，查看指定目录下的csv文件

## 240\_Spark运维管理进阶-作业资源调度-静态资源分配原理

spark提供了许多功能用来在集群中同时调度多个作业。首先，回想一下，每个spark作业都会运行自己独立的一批executor进程，此时集群管理器会为我们提供同时调度多个作业的功能。第二，在每个spark作业内部，多个job也可以并行执行，比如说spark-shell就是一个spark application，但是随着我们输入scala rdd action类代码，就会触发多个job，多个job是可以并行执行的。为这种情况，spark也提供了不同的调度器来在一个application内部调度多个job。

我们先来看一下多个作业的同时调度

**静态资源分配**

当一个spark application运行在集群中时，会获取一批独立的executor进程专门为自己服务，比如运行task和存储数据。如果多个用户同时在使用一个集群，并且同时提交多个作业，那么根据cluster manager的不同，有几种不同的方式来管理作业间的资源分配。

最简单的一种方式，是所有cluster manager都提供的，也就是静态资源分配。在这种方式下，每个作业都会被给予一个它能使用的最大资源量的限额，并且可以在运行期间持有这些资源。这是spark standalone集群和YARN集群使用的默认方式。

Standalone集群: 默认情况下，提交到standalone集群上的多个作业，会通过FIFO的方式来运行，每个作业都会尝试获取所有的资源。可以限制每个作业能够使用的cpu core的最大数量（spark.cores.max），或者设置每个作业的默认cpu core使用量（spark.deploy.defaultCores）。最后，除了控制cpu core之外，每个作业的spark.executor.memory也用来控制它的最大内存的使用。

YARN: --num-executors属性用来配置作业可以在集群中分配到多少个executor，--executor-memory和--executor-cores可以控制每个executor能够使用的资源。

要注意的是，没有一种cluster manager可以提供多个作业间的内存共享功能。如果你想要通过这种方式来在多个作业间共享数据，我们建议就运行一个spark作业，但是可以接收网络请求，并对相同RDD的进行计算操作。在未来的版本中，内存存储系统，比如Tachyon会提供其他的方式来共享RDD数据。

## 241\_Spark运维管理进阶-作业资源调度-动态资源分配原理

**动态资源分配**

<http://spark.apache.org/docs/latest/job-scheduling.html>

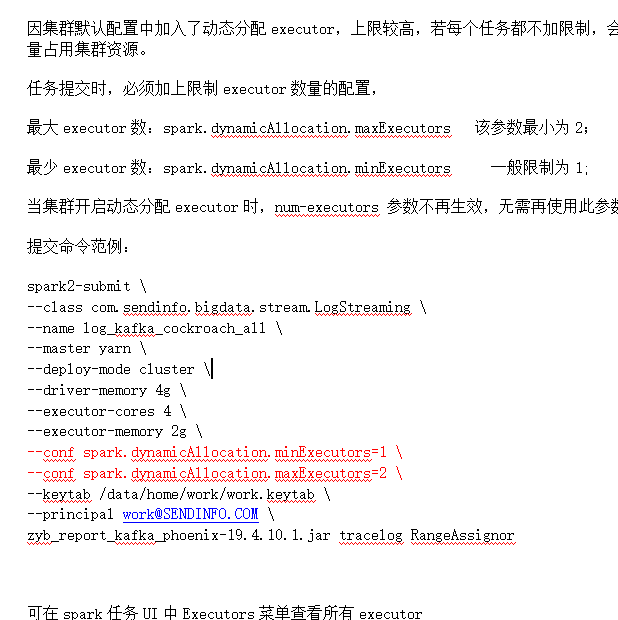
spark 1.2开始，引入了一种根据作业负载动态分配集群资源给你的多个作业的功能。这意味着你的作业在申请到了资源之后，可以在使用完之后将资源还给cluster manager，而且可以在之后有需要的时候再次申请这些资源。这个功能对于多个作业在集群中共享资源是非常有用的。如果部分资源被分配给了一个作业，然后出现了空闲，那么可以还给cluster manager的资源池中，并且被其他作业使用。在spark中，动态资源分配在executor粒度上被实现，可以通过spark.dynamicAllocation.enabled来启用。

**资源分配策略:**

以一个较高的角度来说，当executor不再被使用的时候，spark就应该释放这些executor，并且在需要的时候再次获取这些executor。因为没有一个绝对的方法去预测一个未来可能会运行一个task的executor应该被移除掉，或者一个新的executor应该别加入，我们需要一系列的探索式算法来决定什么应该移除和申请executor。

**申请策略**

**(深大智能要求提交spark作业)**

****

一个启用了动态资源分配的spark作业会在它有pending住的task等待被调度时，申请额外的executor。这个条件必要地暗示了，已经存在的executor是不足以同时运行所有的task的，这些task已经提交了，但是没有完成。driver会轮询式地申请executor。

当在一定时间内（spark.dynamicAllocation.schedulerBacklogTimeout）有pending的task时，

就会触发真正的executor申请，然后每隔一定时间后

（spark.dynamicAllocation.sustainedSchedulerBacklogTimeout），如果又有pending的task了，则再次触发申请操作。此外，每一轮申请到的executor数量都会比上一轮要增加。举例来说，一个作业需要增加一个executor在第一轮申请时，那么在后续的一轮中会申请2个、4个、8个executor。

每轮增加executor数量的原因主要有两方面。

1. 一个作业应该在开始谨慎地申请以防它只需要一点点executor就足够了。

2.作业应该会随着时间的推移逐渐增加它的资源使用量，以防突然大量executor被增加进来。

**移除策略**

移除一个executor的策略比较简单。一个spark作业会在它的executor出现了空闲超过一定时间后（spark.dynamicAllocation.executorIdleTimeout），被移除掉。要注意，在大多数环境下，这个条件都是跟申请条件互斥的，因为如果有task被pending住的话，executor是不该是空闲的。

**executor如何优雅地被释放掉**

在使用动态分配之前，executor无论是发生了故障失败，还是关联的application退出了，都还是存在的。在所有场景中，executor关联的所有状态都不再被需要，并且可以被安全地抛弃。使用动态分配之后，executor移除之后，作业还是存在的。如果作业尝试获取executor写的中间状态数据，就需要去重新计算哪些数据。因此，spark需要一种机制来优雅地卸载executor，在移除它之前要保护它的状态。

解决方案就是使用一个外部的shuffle服务来保存每个executor的中间写状态，这也是spark 1.2引入的特性。这个服务是一个长时间运行的进程，集群的每个节点上都会运行一个，为你的spark作业和executor服务。如果服务被启用了，那么spark executor会在shuffle write和read时，将数据写入该服务，并从该服务获取数据。这意味着所有executor写的shuffle数据都可以在executor生命周期之外继续使用。

除了写shuffle文件，executor也会在内存或磁盘中持久化数据。当一个executor被移除掉时，所有缓存的数据都会消失。目前还没有有效的方案。在未来的版本中，缓存的数据可能会通过堆外存储来进行保存，就像external shuffle service保存shuffle write文件一样。

## 242\_Spark运维管理进阶-作业资源调度-实验：standalone模式下使用动态资源分配

要先启动external shuffle service，后面才能找到7337这个端口的服务：

./sbin/.start-shuffle-service.sh

以下参数跟着spark-submit后面：

--conf spark.shuffle.service.enabled=true \

--conf spark.dynamicAllocation.enabled=true \

--conf spark.shuffle.service.port=7337 \

整体试验步骤：

1、启动external shuffle service

2、启动spark-shell，启用动态资源分配

3、过60s，发现打印日志，说executor被removed，executor进程也没了

4、然后动手写一个wordcount程序，最后提交job的时候，会动态申请一个新的executor，出来一个新的executor进程

5、然后整个作业执行完毕，证明external shuffle service+动态资源分配，流程可以走通

6、再等60s，executor又被释放掉

## 243\_Spark运维管理进阶-作业资源调度-实验：yarn模式下使用动态资源分配

先停止之前为standalone集群启动的shuffle service，./sbin/stop-shuffle-service.sh

**配置**

动态资源分配功能使用的所有配置，都是以spark.dynamicAllocation作为前缀的。要启用这个功能，你的作业必须将spark.dynamicAllocation.enabled设置为true。其他相关的配置之后会详细说明。

此外，你的作业必须有一个外部shuffle服务（external shuffle service）。这个服务的目的是去保存executor的shuffle write文件，从而让executor可以被安全地移除。要启用这个服务，可以将spark.shuffle.service.enabled设置为true。在YARN中，这个外部shuffle service是由org.apache.spark.yarn.network.YarnShuffleService实现的，在每个NodeManager中都会运行。要启用这个服务，需要使用以下步骤：

1、使用预编译好的spark版本。

2、定位到spark-<version>-yarn-shuffle.jar。这个在$SPARK\_HOME/yarn目录下。

3、将上面的jar加入到所有NodeManager的classpath中。

4、在yarn-site.xml中，将yarn.nodemanager.aux-services设置为spark\_shuffle，将

yarn.nodemanager.aux-services.spark\_shuffle.class

设置为org.apache.spark.network.yarn.YarnShuffleService

5、重启所有NodeManager

--conf spark.shuffle.service.enabled=true \

--conf spark.dynamicAllocation.enabled=true \

--conf spark.shuffle.service.port=7337 \

1、首先配置好yarn的shuffle service，然后重启集群

2、接着呢，启动spark shell，并启用动态资源分配，但是这里跟standalone不一样，上来不会立刻申请executor

3、接着执行wordcount，会尝试动态申请executor，并且申请到后，执行job，在spark web ui上，有两个executor

4、过了一会儿，60s过后，executor由于空闲，所以自动被释放掉了，在看spark web ui，没有executor了

## 244\_Spark运维管理进阶-作业资源调度-多个job资源调度原理

在一个spark作业内部，多个并行的job是可以同时运行的。对于job，就是一个spark action操作触发的计算单元。spark的调度器是完全线程安全的，而且支持一个spark application来服务多个网络请求，以及并发执行多个job。

默认情况下，spark的调度会使用FIFO的方式来调度多个job。每个job都会被划分为多个stage，而且第一个job会对所有可用的资源获取优先使用权，并且让它的stage的task去运行，然后第二个job再获取资源的使用权，以此类推。如果队列头部的job不需要使用

整个集群资源，之后的job可以立即运行，但是如果队列头部的job使用了集群几乎所有的资源，那么之后的job的运行会被推迟。

从spark 0.8开始，我们是可以在多个job之间配置公平的调度器的。在公平的资源共享策略下，spark会将多个job的task使用一种轮询的方式来分配资源和执行，所以所有的job都有一个基本公平的机会去使用集群的资源。这就意味着，即使运行时间很长的job

先提交并在运行了，之后提交的运行时间较短的job，也同样可以立即获取到资源并且运行，而不会等待运行时间很长的job结束之后才能获取到资源。这种模式对于多个并发的job是最好的一种调度方式。

## 245\_Spark运维管理进阶-作业资源调度-Fair Scheduler使用详解

要启用Fair Scheduler，只要简单地将spark.scheduler.mode属性设置为FAIR即可

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

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

val sc = new SparkContext(conf)

或者

--conf spark.scheduler.mode=FAIR

fair scheduler也支持将job分成多个组并放入多个池中，以及为每个池设置不同的调度优先级。这个feature对于将重要的和不重要的job隔离运行的情况非常有用，可以为重要的job分配一个池，并给予更高的优先级; 为不重要的job分配另一个池，并给予较低的优先级。

默认情况下，新提交的job会进入一个默认池，但是job的池是可以通过spark.scheduler.pool属性来设置的。

如果你的spark application是作为一个服务启动的，SparkContext 7\*24小时长时间存在，然后服务每次接收到一个请求，就用一个子线程去服务它

在子线程内部，去执行一系列的RDD算子以及代码来触发job的执行

在子线程内部，可以调用SparkContext.setLocalProperty("spark.scheduler.pool", "pool1")

在设置这个属性之后，所有在这个线程中提交的job都会进入这个池中。同样也可以通过将该属性设置为null来清空池子。

**池的默认行为**

默认情况下，每个池子都会对集群资源有相同的优先使用权，但是在每个池内，job会使用FIFO的模式来执行。举例来说，如果要为每个用户创建一个池，这就意味着每个用户都会获得集群的公平使用权，但是每个用户自己的job会按照顺序来执行。

**配置池的属性**

可以通过配置文件来修改池的属性。每个池都支持以下三个属性:

1、schedulingMode: 可以是FIFO或FAIR，来控制池中的jobs是否要排队，或者是共享池中的资源

2、weight: 控制每个池子对集群资源使用的权重。默认情况下，所有池子的权重都是1.如果指定了一个池子的权重为2。举例来说，

它就会获取其他池子两倍的资源使用权。设置一个很高的权重值，比如1000，也会很有影响，基本上该池子的task会在其他所有池子的task之前运行。

3、minShare: 除了权重之外，每个池子还能被给予一个最小的资源使用量。

池子的配置是通过xml文件来配置的，在spark/conf的fairscheduler.xml中配置

我们自己去设置这个文件的路径，conf.set("spark.scheduler.allocation.file", "/path/to/file")

文件内容大致如下所示

|  |
| --- |
| <?xml version="1.0"?>  <allocations>  <pool name="production">  <schedulingMode>FAIR</schedulingMode>  <weight>1</weight>  <minShare>2</minShare>  </pool>  <pool name="test">  <schedulingMode>FIFO</schedulingMode>  <weight>2</weight>  <minShare>3</minShare>  </pool>  </allocations> |