唯快不破——Spark

@八斗学院　郑老师

OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

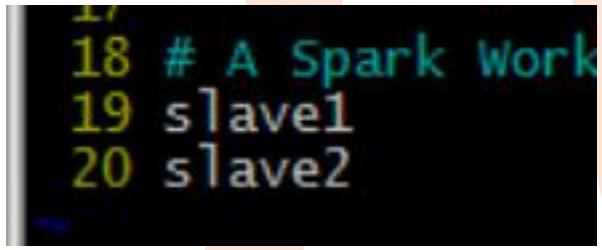
Spark的安装

- 在之前Yarn集群上继续搭建
-]# wget http://mirror.bit.edu.cn/apache/spark/spark-2.0.2/spark-2.0.2-bin-hadoop2.6.tgz
- 解压后，进入conf目录
-]# cp spark-env.sh.template spark-env.sh
-

```
71 export SCALA_HOME=/usr/local/src/scala-2.11.4
72 export JAVA_HOME=/usr/local/src/jdk1.6.0_45
73 export HADOOP_HOME=/usr/local/src/hadoop-2.6.1
74 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
75 SPARK_MASTER_IP=master
76 SPARK_LOCAL_DIRS=/usr/local/src/spark-1.6.0-bin-hadoop2.6
77 SPARK_DRIVER_MEMORY=1G
78
```

Spark的安装

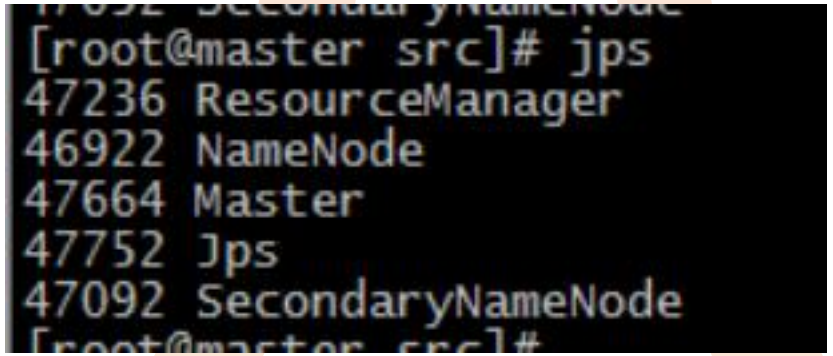
-]# cp slaves.template slaves
- 修改内容为：

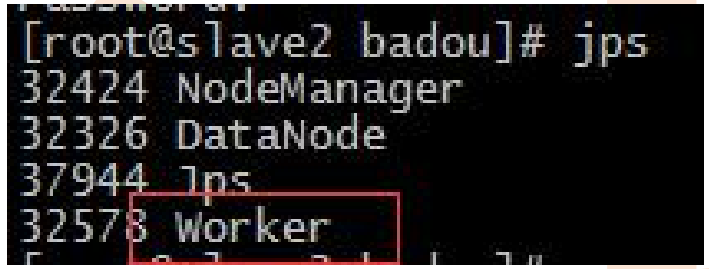


- 最后将配置好的spark安装目录，分发到slave1/2节点上

Spark的启动

- 启动Spark
-]# ./sbin/start-all.sh





Spark的验证

- 验证Spark
- 本地模式：
  -]# ./bin/run-example SparkPi 10 --master local[2]
- 集群模式 Spark Standalone：
  -]# ./bin/spark-submit --class org.apache.spark.examples.SparkPi --master spark://master:7077 examples/jars/spark-examples_2.11-2.0.2.jar 100
- 集群模式 Spark on Yarn集群上yarn-cluster模式：
  -]# ./bin/spark-submit --class org.apache.spark.examples.SparkPi --master yarn-cluster examples/jars/spark-examples_2.11-2.0.2.jar 10

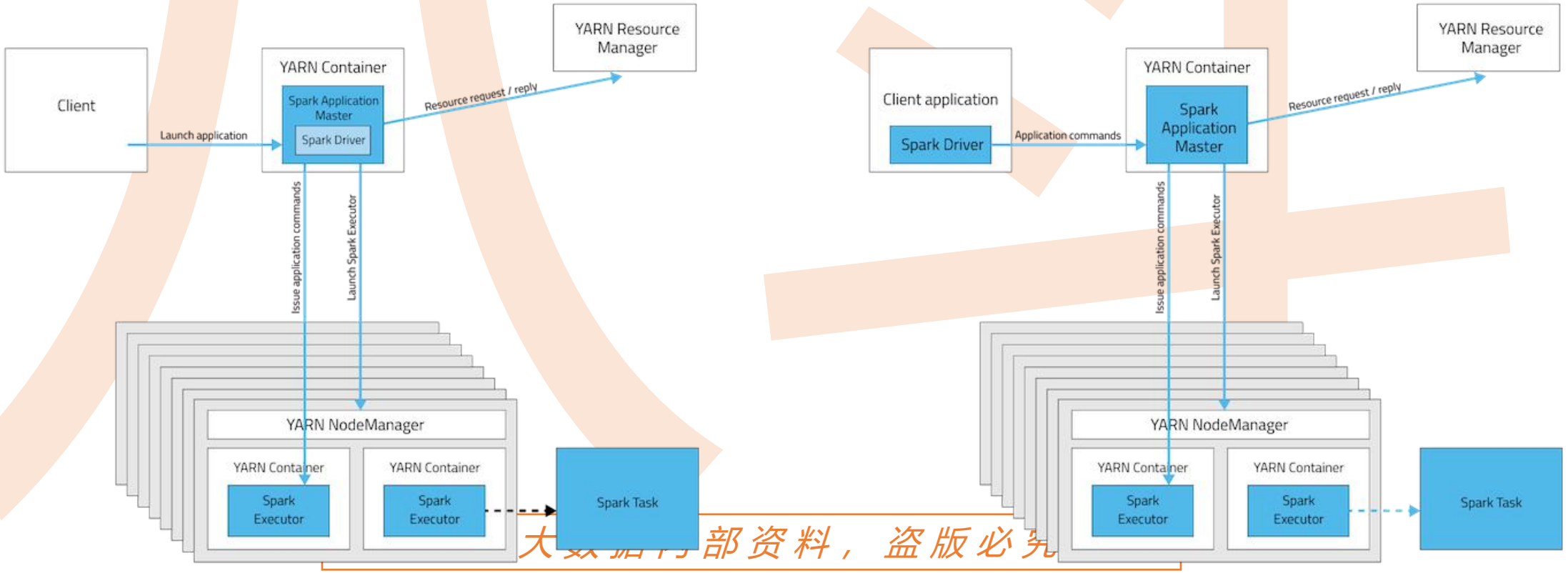
Spark的验证

- Spark standalone vs. Spark on Yarn
- Spark standalone：独立模式，类似MapReduce 1.0所采取的模式，完全由内部实现容错性和资源管理
- Spark on Yarn：让Spark运行在一个通用的资源管理系统之上，这样可以与其他计算框架共享资源

- Yarn Client vs. Spark Standlone vs. Yarn Cluster
- Yarn Client：适用于交互与调试
  - Driver在任务提交机上执行
  - ApplicationMaster只负责向ResourceManager申请executor需要的资源
  - 基于yarn时，spark-shell和pyspark必须要使用yarn-client模式
- Yarn Cluster：适用于生产环境

Spark的验证

- Yarn Cluster vs. Yarn Client区别：本质是AM进程的区别，cluster模式下，driver运行在AM中，负责向Yarn申请资源，并监督作业运行状况，当用户提交完作用后，就关掉Client，作业会继续在yarn上运行。然而cluster模式不适合交互类型的作业。而client模式，AM仅向yarn请求executor，client会和请求的container通信来调度任务，即client不能离开



Spark的资源管理组件

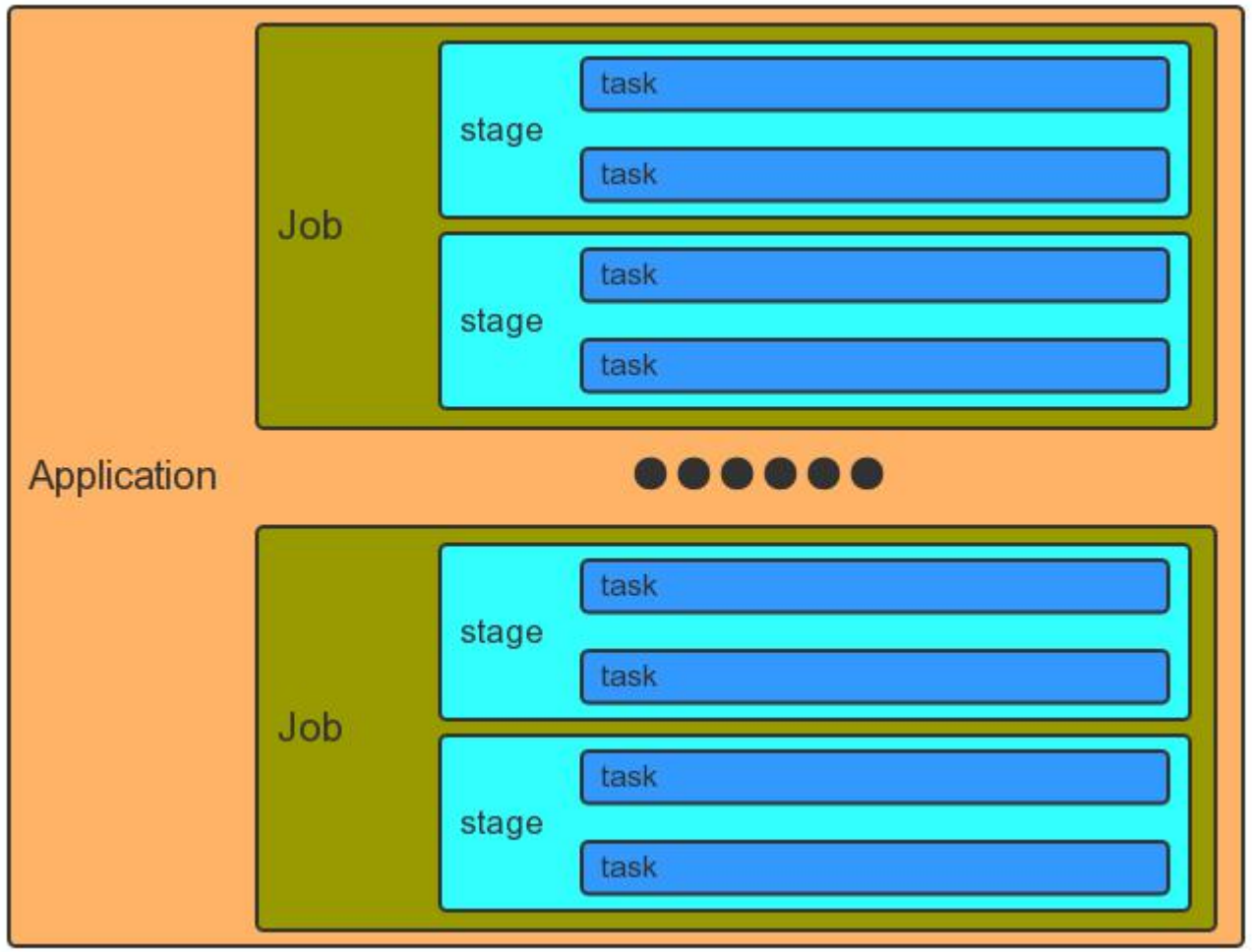
- Yarn（通用）
  - Master/Slave结构
    - RM：全局资源管理器，负责系统的资源管理和分配
    - NM：每个节点上的资源和任务管理器
    - AM：每个应用程序都有一个，负责任务调度和监视，并与RM调度器协商为任务获取资源
- Standalone（Spark自带）
  - Master/Slave结构
    - Master：类似Yarn中的RM
    - Worker：类似Yarn中的NM

Spark和Hadoop作业之间的区别

- Hadoop中：
  - 一个MapReduce程序就是一个job，而一个job里面可以有一个或多个Task，Task又可以区分为Map Task和Reduce Task
  - MapReduce中的每个Task分别在自己的进程中运行，当该Task运行完时，进程也就结束

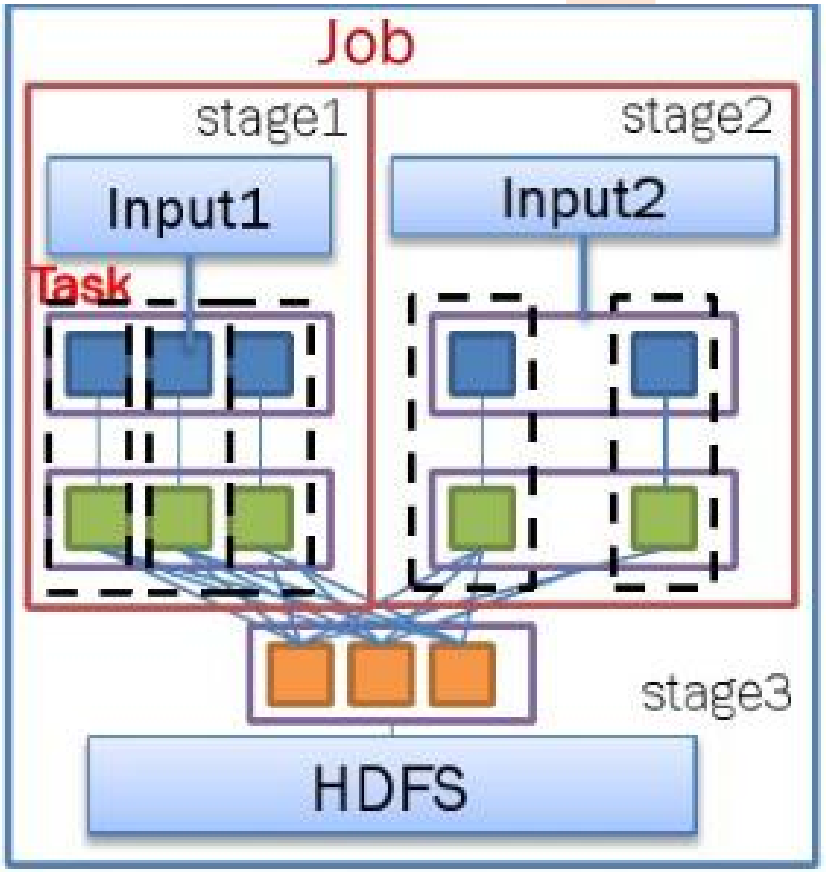
Spark和Hadoop作业之间的区别

- Spark中：
  - 同样有job的概念，但这里的job和MR中的job不一样
  - 一个Application和一个SparkContext相关联，每个Application中可以有一个或多个job，可以并行或串行运行job
  - Spark中一个Action可以触发一个job运行
  - 在job里面包含多个stage，stage以shuffle进行划分，stage中包含多个Task，多个Task构成了Task Set
  - 和MapReduce不一样，Spark中多个Task可以运行在一个进程里面，而且这个进程的生命周期和Application一样，即使没有job运行
  - **优点：**加快Spark运行速度，Task可以快速启动，并处理内存中数据
  - **缺点：**每个Application拥有固定数量的executor和固定数目的内存



Spark和Hadoop作业之间的区别

- Spark中：
  - *应用程序：*由一个driver program和多个job构成
  - *Job：*由多个stage组成
  - *Stage：*对应一个taskset
  - *Taskset：*对应一组关联的相互之间没有shuffle依赖关系的task组成
  - *Task：*任务最小的工作单元



Spark和Hadoop作业之间的区别

- Driver Program：
  - （驱动程序）是Spark的核心组件
  - 构建SparkContext（Spark应用的入口，创建需要的变量，还包含集群的配置信息等）
  - 将用户提交的job转换为DAG图（类似数据处理的流程图）
  - 根据策略将DAG图划分为多个stage，根据分区从而生成一系列tasks
  - 根据tasks要求向RM申请资源
  - 提交任务并检测任务状态
- Executor：
  - 真正执行task的单元，一个Work Node上可以有多个Executor

OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

Scala语言

- Scala语言诞生于2003年，Scalable Lanuage
- Spark源码由Scala语言开发
- 学习参考：http://www.runoob.com/scala/scala-tutorial.html

- 变量定义：有两种变量val和var
- val类似于Java里的final变量。一旦初始化后就不能再赋值。
- var如同Java里面的非final变量。var可以在它生命周期中被多次赋值。
- 与Java不同的是类型声明在变量后面，用一个”:”分隔，如果没有指定变量类型，编译器将会自动推断。

```
scala> val msg1 : String = "hello scala"
msg1: String = hello scala

scala> val msg1 : String = "hello scala"
msg1: String = hello scala
```

- **注意：**当val被声明为lazy时，它的初始化将被推迟，直到首次取用它的值。

Scala语言

- 支持的数据类型：

数据类型	描述
Byte	8位有符号补码整数。数值区间为 -128 到 127
Short	16位有符号补码整数。数值区间为 -32768 到 32767
Int	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807
Float	32位IEEE754单精度浮点数
Double	64位IEEE754单精度浮点数
Char	16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF
String	字符序列
Boolean	true或false
Unit	表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。Unit只有一个实例值，写成()。
Null	null 或空引用
Nothing	Nothing类型在Scala的类层级的最低端；它是任何其他类型的子类型。
Any	Any是所有其他类的超类
AnyRef	AnyRef类是Scala里所有引用类(reference class)的基类

- 复合类型：
  - 数组（Array）、列表（List）、元组（Tuple）、集合（Set）、映射（Map）

- 复合类型：数组（Array）
- 声明方法： val 数组名 = new Array[类型名](数组大小)
- 类似： val greetStrings= new Array[String](3)
- 其中Array[String]变量greetStrings的类型，3为实例初始化参数

- 注意：Scala里的数组是通过把索引放在圆括号里面访问的，而不是像Java那样放在方括号里。所以数组的第零个元素是greetStrings(0)，不是greetStrings[0]

Scala语言

- 复合类型：数组（Array）——多维数组
- 声明方式：Array.ofDim[类型](维度1, 维度2, 维度3,....)
- 例子：val muldimArr= Array.ofDim[Double](2,3)

```
scala> val muldimArr= Array.ofDim[Double](2,3)
muldimArr: Array[Array[Double]] = Array(Array(0.0, 0.0, 0.0), Array(0.0, 0.0, 0.0))
```

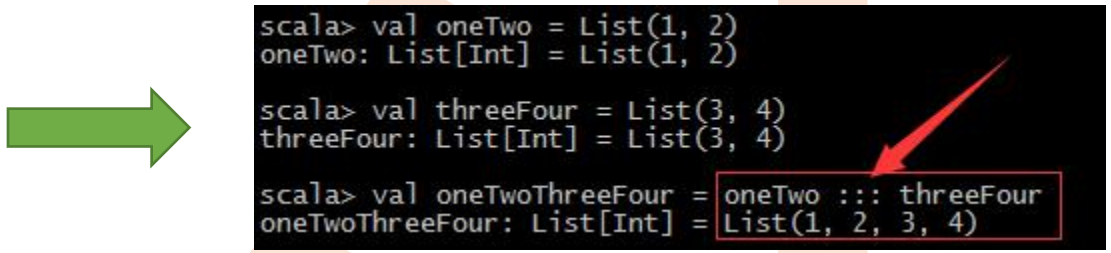
- 也可以通过Array[Array[Int]](维度) 来声明数组，可以声明不规则数组

```
val difLenMulArr= new Array[Array[Int]](3)
for(i <- 1 to difLenMulArr.length)
{
  difLenMulArr(i-1) = new Array[Int](i)
}
```

```
scala> val difLenMulArr= new Array[Array[Int]](3)
difLenMulArr: Array[Array[Int]] = Array(null, null, null)

scala> for(i <- 1 to difLenMulArr.length)
     | {
     |   difLenMulArr(i-1) = new Array[Int](i)
     | }

scala> difLenMulArr
res3: Array[Array[Int]] = Array(Array(0), Array(0, 0), Array(0, 0, 0))
```

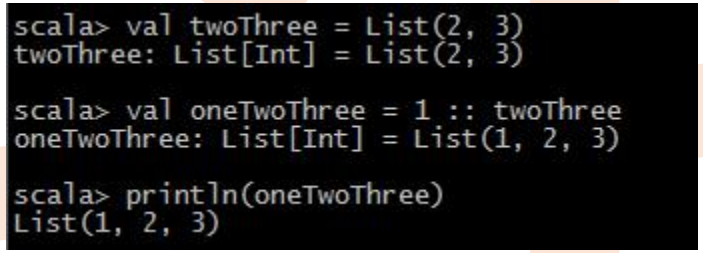
- 复合类型：列表（List）

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
```



- List提供了“:::”方法实现叠加功能

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```



- List提供了‘::’方法把一个新元素组合到已有List的最前端，然后返回结果List

- 复合类型：元组（Tuple）
- 元组可以包含不同类型的元素
- 将不同的值以逗号分隔，用小括号括起来表示，是不同类型值的聚集。
- 可以"_n"的形式访问元组元素，n代表元素在元组中的序号

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

```
scala> val pair = (99, "Luftballons")
pair: (Int, String) = (99,Luftballons)

scala> println(pair._1)
99

scala> println(pair._2)
Luftballons
```

- 复合类型：集合（Set）
- 元组可以包含不同类型的元素
- 将不同的值以逗号分隔，用小括号括起来表示，是不同类型值的聚集。
- 可以"_n"的形式访问元组元素，n代表元素在元组中的序号

```
scala> import scala.collection.immutable.Set
import scala.collection.immutable.Set

scala> var jetSet = Set("Boeing", "Airbus")//这里定义的是不可变集
jetSet: scala.collection.immutable.Set[String] = Set(Boeing, Airbus)

scala> jetSet += "Lear" //因为是不可变集，所以这里的+=其实是重新赋值jetSet，所以jetSet要声明成var而不是val

scala> println(jetSet.contains("Cessna"))
false

scala> println(jetSet.contains("Boeing"))
true

scala> println(jetSet.contains("Lear"))
true

scala> jetSet
res11: scala.collection.immutable.Set[String] = Set(Boeing, Airbus, Lear)

scala>
```

- 复合类型：集合（Set）

```
scala> jetSet
res28: scala.collection.immutable.Set[String] = Set(Boeing, Airbus, Lear)

scala> val site1 = Set("Runoob", "Google", "Baidu")
site1: scala.collection.immutable.Set[String] = Set(Runoob, Google, Baidu)

scala> val site2 = Set("Faceboook", "Taobao")
site2: scala.collection.immutable.Set[String] = Set(Faceboook, Taobao)

scala> var site = site1 ++ site2
site: scala.collection.immutable.Set[String] = Set(Faceboook, Taobao, Google, Baidu, Runoob)

scala> site = site1.++(site2)
site: scala.collection.immutable.Set[String] = Set(Faceboook, Taobao, Google, Baidu, Runoob)
```

```
scala> val num = Set(5,6,9,20,30,45)
num: scala.collection.immutable.Set[Int] = Set(5, 20, 6, 9, 45, 30)

scala> num.min
res29: Int = 5

scala> num.max
res30: Int = 45
```

```
scala> val num1 = Set(5,6,9,20,30,45)
num1: scala.collection.immutable.Set[Int] = Set(5, 20, 6, 9, 45, 30)

scala> val num2 = Set(50,60,9,20,35,55)
num2: scala.collection.immutable.Set[Int] = Set(20, 60, 9, 35, 50, 55)

scala> num1.&(num2)
res31: scala.collection.immutable.Set[Int] = Set(20, 9)

scala> num1.intersect(num2)
res32: scala.collection.immutable.Set[Int] = Set(20, 9)
```

- 复合类型：映射（Map）
- 对偶，即名值对。可以通过 -> 操作符来定义对偶， 名->值 运算的结果是(名, 值)

```
scala> import scala.collection.mutable.Map
import scala.collection.mutable.Map

scala> val treasureMap = Map[Int, String]()//定义一个可变的Map，因为是可变的，所以不需要对treasureMap 重新赋值，所以它是val
treasureMap: scala.collection.mutable.Map[Int,String] = Map()

scala> treasureMap += (1 -> "Go to island.")
res39: treasureMap.type = Map(1 -> Go to island.)

scala> treasureMap += (2 -> "Find big X on ground.")
res40: treasureMap.type = Map(2 -> Find big X on ground., 1 -> Go to island.)

scala> treasureMap += (3 -> "Dig.")
res41: treasureMap.type = Map(2 -> Find big X on ground., 1 -> Go to island., 3 -> Dig.)
```

```
scala> treasureMap.contains((1))
res43: Boolean = true
```

```
scala> treasureMap(1)
res44: String = Go to island.
```

- 函数
- 定义的格式为：def 函数名(参数列表) : 返回值类型 = { 函数体 }

```
def max(x: Int, y: Int): Int = { //定义函数
  if (x > y) x
  else y
}

max(3,7)  // 调用函数
```

```
scala> def max(x: Int, y: Int): Int = { //定义函数
     |   if (x > y) x
     |   else y
     | }
max: (x: Int, y: Int)Int

scala> max(3, 7)
res0: Int = 7
```

- 函数返回值可以用return指定，使用return时函数定义必须指定返回值类型。
- 如果没有使用return关键字，默认函数体代码块的最后计算的表达式的值作为返回值，无需指定返回值类型。
- 对于递归函数，必须指定返回值类型。
- 可以在函数的内部再定义函数，如同定义一个局部变量。

- 函数——匿名函数

```
var increase = (x :Int ) => x +1
increase(10) // 调用，输出11
```

```
scala> var increase = (x :Int ) => x +1
increase: Int => Int = <function1>

scala> increase(10)
res1: Int = 11
```

- Scala的库允许你使用函数作为参数，比如foreach方法，它使用一个函数参数，为集合中每个运算调用传入的函数

```
scala> val someNumbers = List ( -11, -10, - 5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x:Int) => println(x))
-11
-10
-5
0
5
10
```

- Scala的集合也支持一个filter方法用来过滤集合中的元素，filter的参数也是一个函数

```
scala> someNumbers.filter( x => x >0)
res4: List[Int] = List(5, 10)
```

Scala语言

- 函数——简化表达
- Scala提供了多种方法来简化函数字面量中多余的部分
- 可使用“_” 来代替单个的参数，如_ => _ > 0

```
scala> val someNumbers = List ( -11, -10, - 5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter(_ >0)
res5: List[Int] = List(5, 10)
```

Scala语言

- 类和对象
- Scala类定义和Java非常类似，也以关键字class 声明
- 和Java不同的，Scala的缺省修饰符为public ，也就是如果不带有访问范围的修饰符public,protected,private，Scala缺省定义为 public。

```
class ChecksumAccumulator{
  private var sum=0
  def add(b:Byte) :Unit = sum +=b
  def checksum() : Int = ~ (sum & 0xFF) +1
}
```

Scala语言

- 单例对象
- Scala不提供静态元素(静态变量或静态方法)
- 在Scala中提供类似功能的是称为"Singleton（单例对象）"的对象。
- 在Scala中定义Singleton对象的方法使用object关键字，与普通类定义形式非常类似

```
object ChecksumAccumulator {
  private val cache = Map [String, Int] ()
  def calculate(s:String) : Int =
  if(cache.contains(s))
    cache(s)
  else {
    val acc=new ChecksumAccumulator
    for( c <- s)
      acc.add(c.toByte)
    val cs=acc.checksum()
    cache += ( s -> cs)
    cs
  }
}
```

OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

Spark on yarn的结构

- 什么是spark?
  - 也是一个分布式的并行计算框架
  - spark是下一代的map-reduce，扩展了mr的数据处理流程。

MR有什么问题?

- 调度慢，启动map、reduce太耗时
- 计算慢，每一步都要保存中间结果落磁盘
- API抽象简单，只有map和reduce两个原语
- 缺乏作业流描述，一项任务需要多轮mr

Wordcount：map

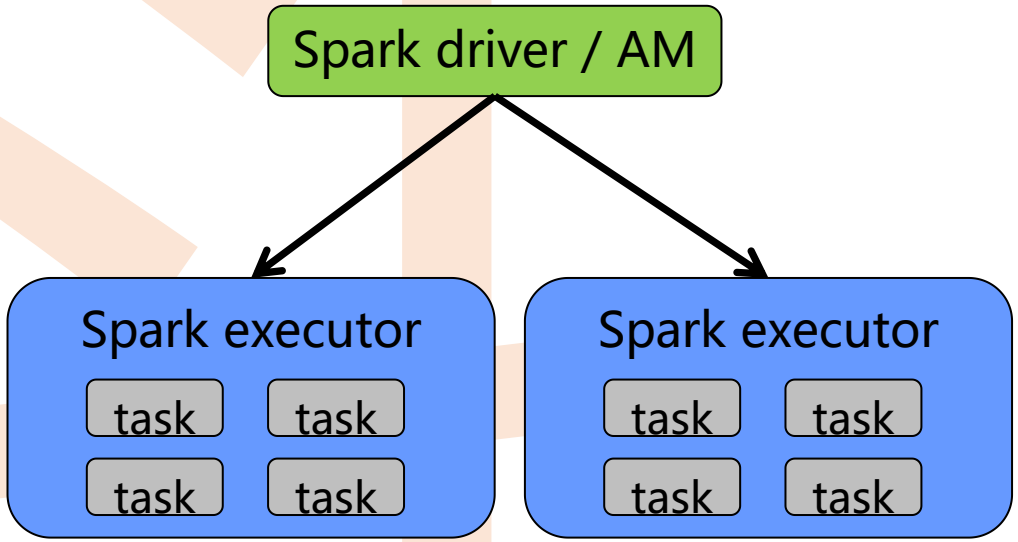
```
while read LINE; do
    for word in $LINE
    do
        echo "$word 1"
    done
done
```

Wordcount：reduce

```
while read LINE;do
    newword=`echo $LINE | cut -d ' '  -f 1`
    if [ "$word" != "$newword" ];then
        [ $started -ne 0 ] && echo "$word $count"
        word=$newword
        count=1
        started=1
    else
        count=$(( $count + 1 ))
    fi
done
echo "$word $count"
```

Spark on yarn的结构

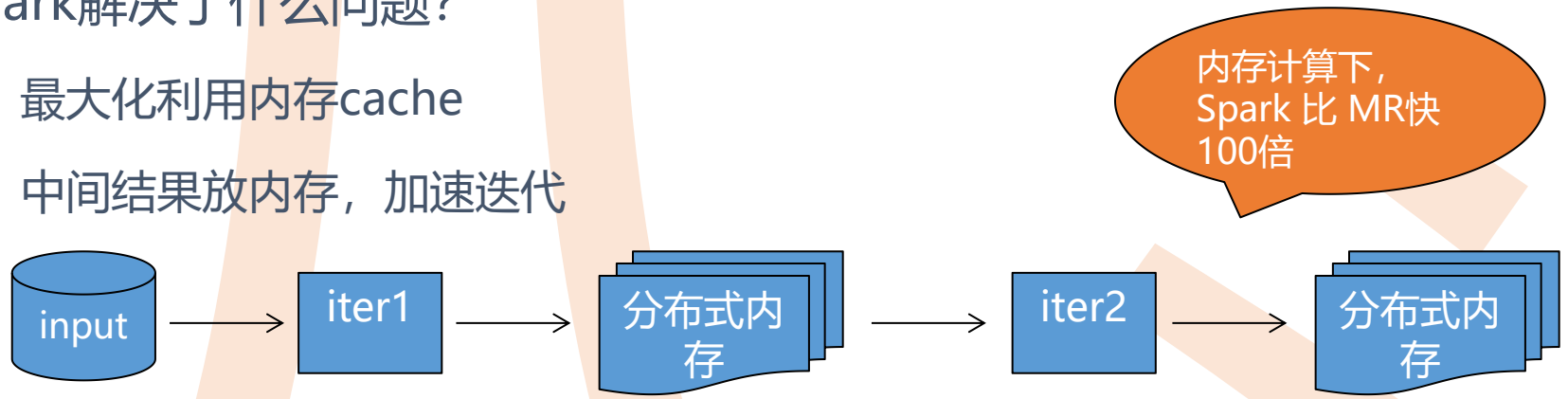
- 什么是spark?
  - 也是一个分布式的并行计算框架
  - spark是下一代的map-reduce，扩展了mr的数据处理流程。
  - executor都是装载在container里运行，container默认内存是1G（参数yarn.scheduler.minimum-allocation-mb定义）
  - executor分配的内存是executor-memory，向YARN申请的内存是（executor-memory + 1）* num-executors。
  - AM在Spark中叫driver，AM向RM申请的是executor资源，当分配完资源后，executor启动后，由spark的AM向executor分配task，分配多少task、分配到哪个executor由AM决定，可理解为spark也有个调度过程，这些task都运行在executor的坑里
  - Executor有线程池多线程管理这些坑内的task



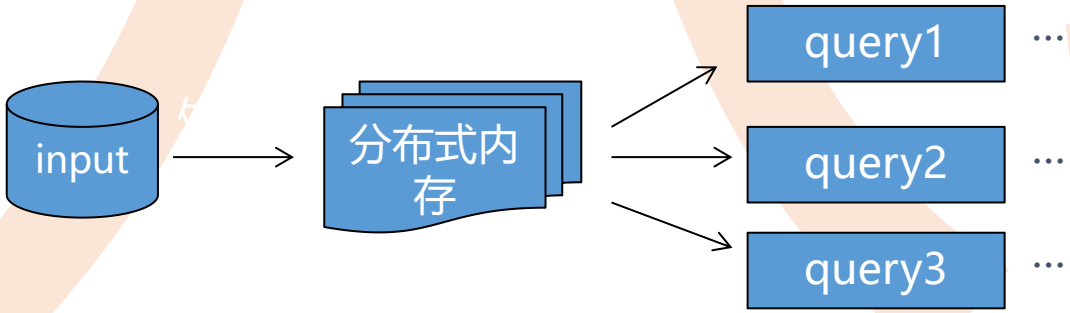
- Executor伴随整个app的生命周期
- 线程池模型，省去进程频繁启停的开销

Spark on yarn的结构

- Spark解决了什么问题?
  - 最大化利用内存cache
  - 中间结果放内存，加速迭代



  - 某结果集放内存，加速后续查询和处理，解决运行慢的问题



原始SQL：

select col2, max (col3) from table where col1 > 50 group by col2

select col3, max (col2) from table where col1 > 50 group by col3

Cachetable：

select * from table where col1 > 50

rdd.registerastable（cachetable）

改造SQL：

select col2, max (col3) from cachetable group by col2

select col3, max (col2) from cachetable group by col3

Spark on yarn的结构

- Spark解决了什么问题?
  - 更丰富的API

  (解决API单一问题)

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

  - Transfomation变换的api，比如map可对每一行做变换，filter过滤出符合条件的行等，这些API实现用户算法，灵活。
  - spark提供很多转换和动作，很多基本操作如Join，GroupBy已经在RDD转换和动作中实现。不需用户自己实现

Spark on yarn的结构

- Spark解决了什么问题?
  - 完整作业描述
  - 将用户的整个作业穿起来。关键是这3行。可以立即解释。不像mr那样，需要实现多个map和reduce脚本，解决MR缺乏作业流描述问题

```
val file = sc.textFile(hdfs://input)
val counts = file.flatMap(
                line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)

counts.saveAsTextFile(hdfs://output)
```

- 一个作业中描述完整的APP作业流
- 多语言SDK支持：Scala、java、python

OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

Spark核心

- Spark基于弹性分布式数据集（RDD）模型，具有良好的通用性、容错性与并行处理数据的能力
- RDD（Resilient Distributed Dataset）：弹性分布式数据集（相当于集合），它的本质是数据集的描述（只读的、可分区的分布式数据集），而不是数据集本身
- RDD的关键特征：
  - RDD使用户能够显式将计算结果保存在内存中，控制数据的划分，并使用更丰富的操作集合来处理
  - 使用更丰富的操作来处理，只读（由一个RDD变换得到另一个RDD，但是不能对本身的RDD修改）
  - 记录数据的变换而不是数据本身保证容错（lineage）
    - 通常在不同机器上备份数据或者记录数据更新的方式完成容错，但这种对任务密集型任务代价很高
    - RDD采用数据应用变换（map,filter,join），若部分数据丢失，RDD拥有足够的信息得知这部分数据是如何计算得到的，可通过重新计算来得到丢失的数据
    - 这种恢复数据方法很快，无需大量数据复制操作，可以认为Spark是基于RDD模型的系统
  - 懒操作，延迟计算，action的时候才操作
  - 瞬时性，用时才产生，用完就释放

Spark核心

- Spark允许从以下四个方面构建RDD
  - 从共享文件系统中获取，如从HDFS中读数据构建RDD
    - val a = sc.textFile(“/xxx/yyy/file”)
  - 通过现有RDD转换得到
    - val b = a.map(x => (x, 1))
  - 定义一个scala数组
    - val c = sc.parallelize(1 to 10, 1)
  - 有一个已经存在的RDD通过持久化操作生成
    - val d = a.persist(), a. saveAsHadoopFile(“/xxx/yyy/zzz”)

Sparkcontext是spark的入口，编写spark程序用到的第一个类，包含sparkconf sparkenv等类

- Spark针对RDD提供两类操作：transformations和action
  - transformations是RDD之间的变换，action会对数据执行一定的操作
  - transformations采用懒策略，仅在对相关RDD进行action提交时才触发计算

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Spark核心

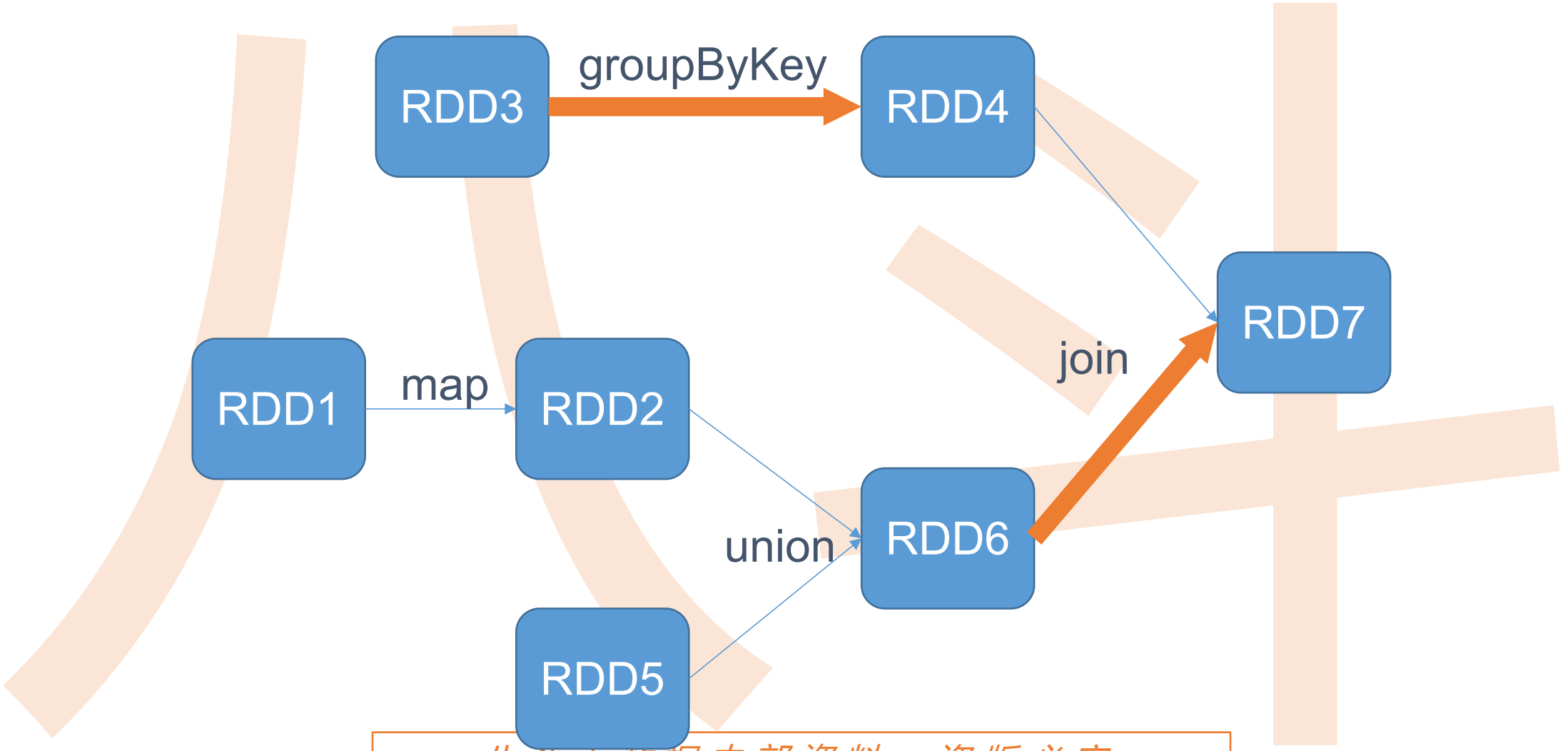
- 每个RDD包含了数据分块/分区（partition）的集合，每个partition是不可分割的
  - 实际数据块的描述（实际数据到底存在哪，或者不存在）
  - 其值依赖于哪些partition

lineage（家系）信息

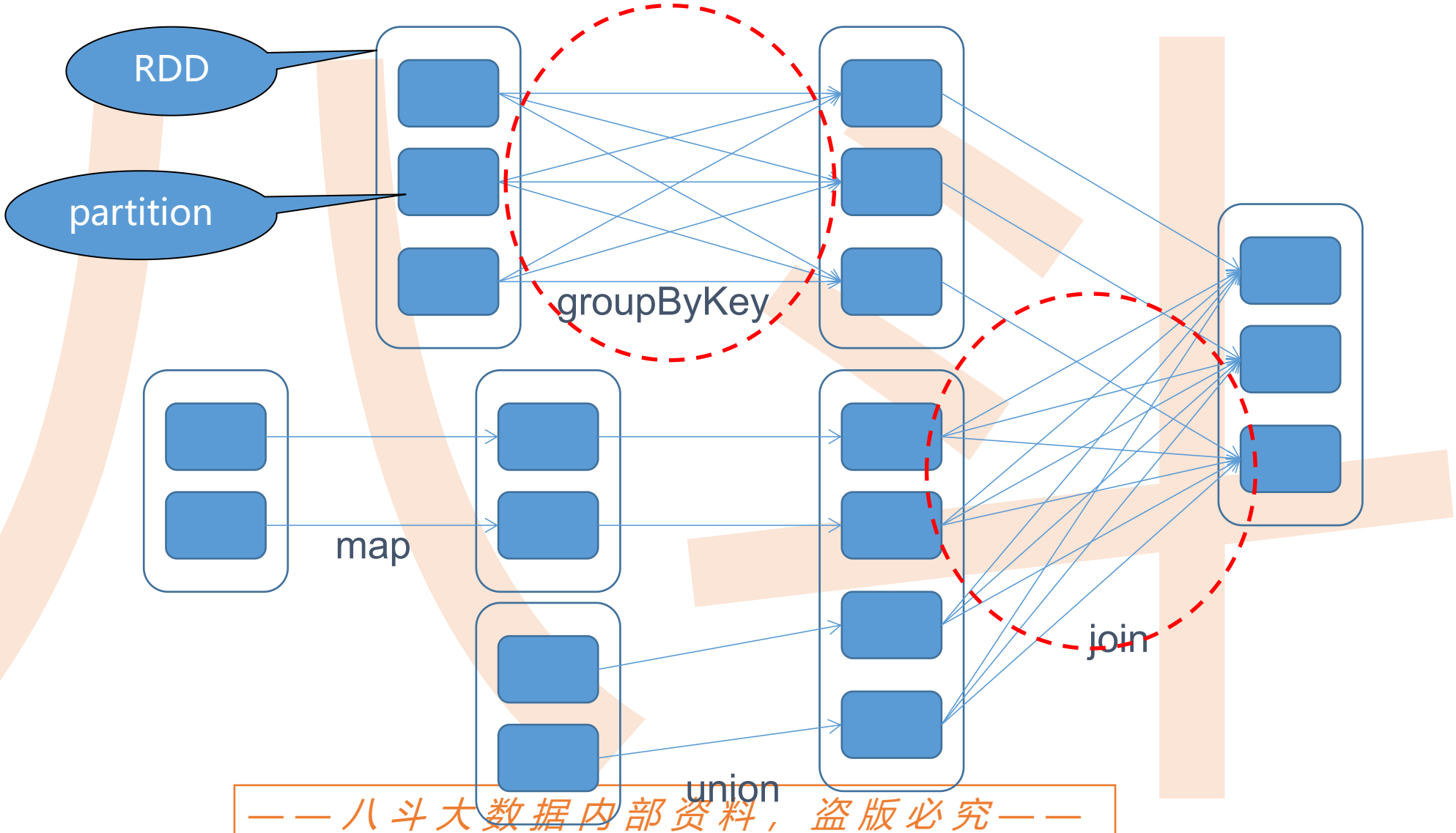
RDD的DAG关系

- 与父RDD的依赖关系（rddA=>rddB）
  - 宽依赖： B的每个partition依赖于A的所有partition
    - 比如groupByKey、reduceByKey、join……，由A产生B时会先对A做shuffle分桶
  - 窄依赖： B的每个partition依赖于A的常数个partition
    - 比如map、filter、union……

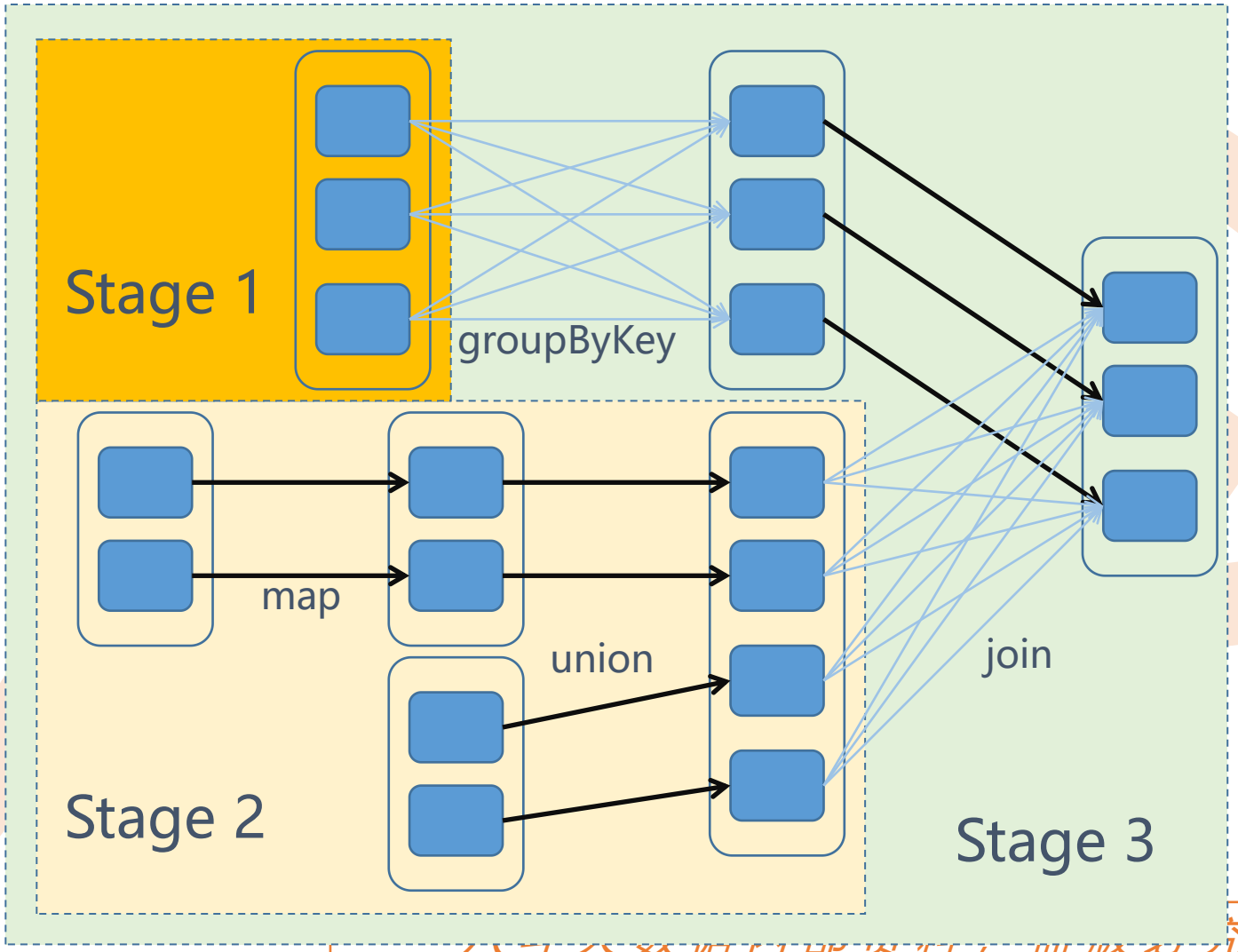
Spark核心——RDD依赖关系



Spark核心——RDD依赖关系



Spark核心——RDD依赖关系



- 从后往前，将宽依赖的边删掉，连通分量及其在原图中所有依赖的RDD，构成一个stage
- DAG是在计算过程中不断扩展，在action后才会启动计算
- 每个stage内部尽可能多地包含一组具有窄依赖关系的转换，并将它们流水线并行化（pipeline）

Spark核心——RDD依赖关系

- 每个partition的计算就是一个task，task是调度的基本单位
- 若一个stage包含的其他stage中的任务已经全部完成，这个stage中的任务才会被加入调度
- 遵循数据局部性原则，使得数据传输代价最小
  - 如果一个任务需要的数据在某个节点的内存中，这个任务就会被分配至那个节点
  - 需要的数据在某个节点的文件系统中，就分配至那个节点

此时的调度指的是：由spark的AM来决定计算partition的task，分配到哪个executor上

Spark核心——容错

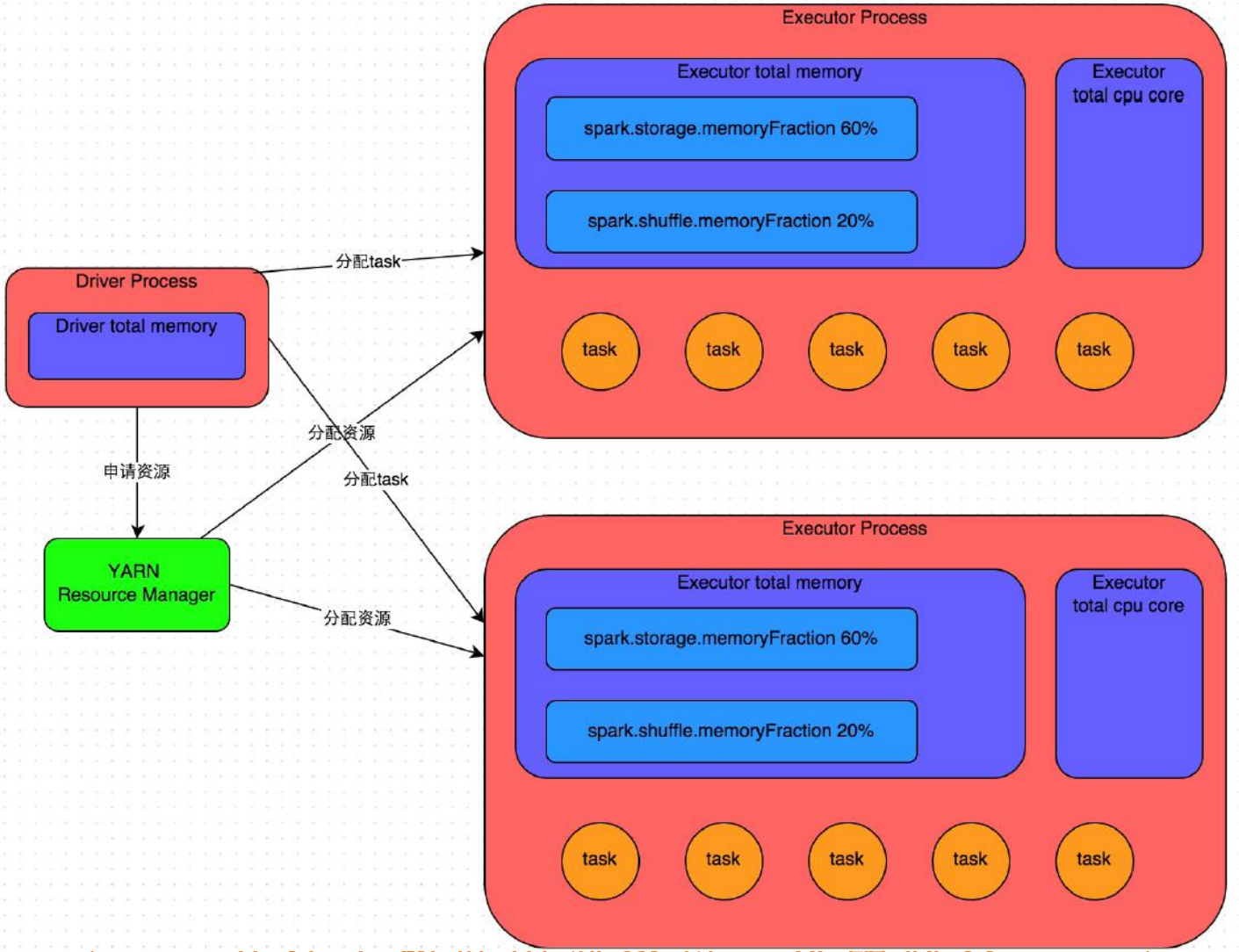
- 如果此task失败，AM会重新分配task
- 如果task依赖的上层partition数据已经失效了，会先将其依赖的partition计算任务再重算一遍
- 宽依赖中被依赖partition，可以将数据保存HDFS，以便快速重构（checkpoint）
  - 窄依赖只依赖上层一个partition，恢复代价较少；宽依赖依赖上层所有partition，如果数据丢失，上层所有partiton要重算
- 可以指定保存一个RDD的数据至节点的cache中，如果内存不够，会LRU释放一部分，仍有重构的可能

这是一个递归过程，会一直追本溯源，甚至直到最初的输入数据

OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

Spark作业运行原理



Spark资源参数调优

- Executor的内存分为3块
- 第一块：让task执行代码时，默认占executor总内存的20%
- 第二块：task通过shuffle过程拉取上一个stage的task的输出后，进行聚合等操作时使用，默认也是占20%
- 第三块：让RDD持久化时使用，默认占executor总内存的60%

- Task的执行速度和每个executor进程的CPU Core数量有直接关系，一个CPU Core同一时间只能执行一个线程，每个executor进程上分配到的多个task，都是以task一条线程的方式，多线程并发运行的。如果CPU Core数量比较充足，而且分配到的task数量比较合理，那么可以比较快速和高效地执行完这些task线程

Spark资源参数调优

- ***num-executors:*** 该作业总共需要多少executor进程执行
  - 建议：每个作业运行一般设置50~100个左右较合适
- ***executor-memory:*** 设置每个executor进程的内存， num-executors* num-executors代表作业申请的总内存量（尽量不要超过最大总内存的1/3~1/2）
  - 建议：设置4G~8G较合适
- ***executor-cores:*** 每个executor进程的CPU Core数量，该参数决定每个executor进程并行执行task线程的能力， num-executors* executor-cores代表作业申请总CPU core数（不要超过总CPU Core的1/3~1/2 ）
  - 建议：设置2~4个较合适

Spark资源参数调优

- ***driver-memory：*** 设置Driver进程的内存
  - 建议：通常不用设置，一般1G就够了，若出现使用collect算子将RDD数据全部拉取到Driver上处理，就必须确保该值足够大，否则OOM内存溢出
- ***spark.default.parallelism：*** 每个stage的默认task数量
  - 建议：设置500~1000较合适，默认一个HDFS的block对应一个task，Spark默认值偏少，这样导致不能充分利用资源
- ***spark.storage.memoryFraction：*** 设置RDD持久化数据在executor内存中能占的比例，默认0.6，即默认executor 60%的内存可以保存持久化RDD数据
  - 建议：若有较多的持久化操作，可以设置高些，超出内存的会频繁gc导致运行缓慢
- ***spark.shuffle.memoryFraction：*** 聚合操作占executor内存的比例，默认0.2
  - 建议：若持久化操作较少，但shuffle较多时，可以降低持久化内存占比，提高shuffle操作内存占比

Spark资源参数调优

- spark-submit命令示例：

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

Spark开发调优

- 原则一：避免创建重复的RDD
  - 对同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据
  - 极大浪费内存

```
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd1.map(...)
val rdd2 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd2.reduce(...)
```

```
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd1.map(...)
rdd1.reduce(...)
```

Spark开发调优

- 原则二：尽可能复用同一个RDD
  - 比如：一个RDD数据格式是key-value，另一个是单独value类型，这两个RDD的value部分完全一样，这样可以复用达到减少算子执行次数

```
JavaPairRDD</long><long , String> rdd1 = ...
JavaRDD<string> rdd2 = rdd1.map(...)
```

```
JavaPairRDD<long , String> rdd1 = ...
rdd1.reduceByKey(...)
rdd1.map(tuple._2...)
```

Spark开发调优

- 原则三：对多次使用的RDD进行持久化处理
  - 每次对一个RDD执行一个算子操作时，都会重新从源头处理计算一遍，计算出那个RDD出来，然后进一步操作，这种方式性能很差
  - 对多次使用的RDD进行持久化，将RDD的数据保存在内存或磁盘中，避免重复劳动
  - 借助cache()和persist()方法

```
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").cache()
rdd1.map(...)
rdd1.reduce(...)
```

```
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
.persist(StorageLevel.MEMORY_AND_DISK_SER)
rdd1.map(...)
rdd1.reduce(...)
```

内存充足以内存持久化优先，_SER表示序列化

Spark开发调优

- 原则三：对多次使用的RDD进行持久化处理
  - persist持久化级别

持久化级别	含义解释
MEMORY_ONLY	使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。
MEMORY_AND_DISK	使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。
MEMORY_ONLY_SER	基本含义同MEMORY_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
MEMORY_AND_DISK_SER	基本含义同MEMORY_AND_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
DISK_ONLY	使用未序列化的Java对象格式，将数据全部写入磁盘文件中。
MEMORY_ONLY_2, MEMORY_AND_DISK_2, 等等.	对于上述任意一种持久化策略，如果加上后缀_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。

Spark开发调优

- 原则四：避免使用shuffle类算子
  - 在spark作业运行过程中，最消耗性能的地方就是shuffle过程
  - 将分布在集群中多个节点上的同一个key，拉取到同一个节点上，进行聚合和join处理，比如groupByKey、reduceByKey、join等算子，都会触发shuffle

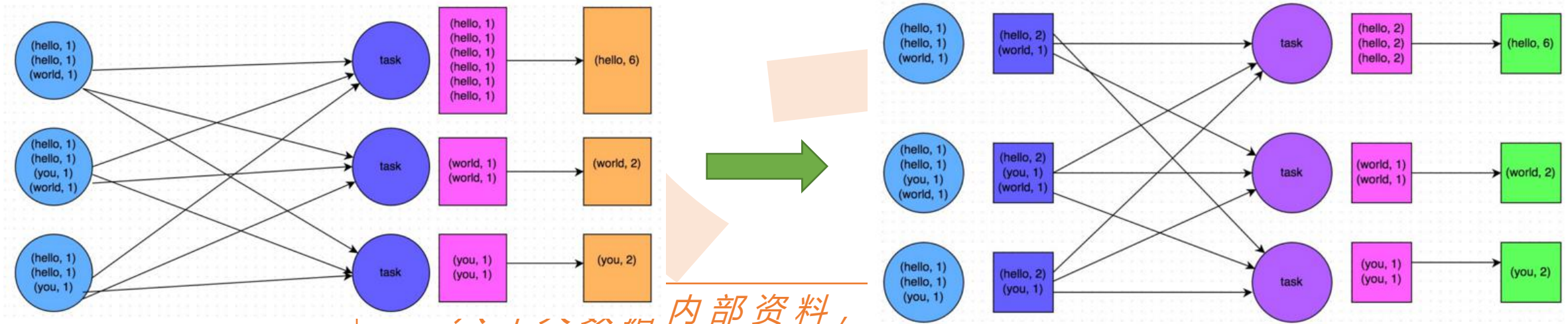
```
val rdd3 = rdd1.join(rdd2)
```

```
val rdd2Data = rdd2.collect()
val rdd2DataBroadcast = sc.broadcast(rdd2Data)

val rdd3 = rdd1.map(rdd2DataBroadcast...)
```

Broadcast+map的join操作，不会导致shuffle操作，但前提适合RDD数据量较少时使用

Spark开发调优

- 原则五：使用map-side预聚合的shuffle操作
  - 一定要使用shuffle的，无法用map类算子替代的，那么尽量使用map-site预聚合的算子
  - 思想类似MapReduce中的Combiner
  - 可能的情况下使用reduceByKey或aggregateByKey算子替代groupByKey算子，因为reduceByKey或aggregateByKey算子会使用用户自定义的函数对每个节点本地相同的key进行预聚合，而groupByKey算子不会预聚合



Spark开发调优

- 原则六：使用Kryo优化序列化性能
  - Kryo是一个序列化类库，来优化序列化和反序列化性能
  - Spark默认使用Java序列化机制（ObjectOutputStream/ ObjectInputStream API）进行序列化和反序列化
  - Spark支持使用Kryo序列化库，性能比Java序列化库高很多，10倍左右

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

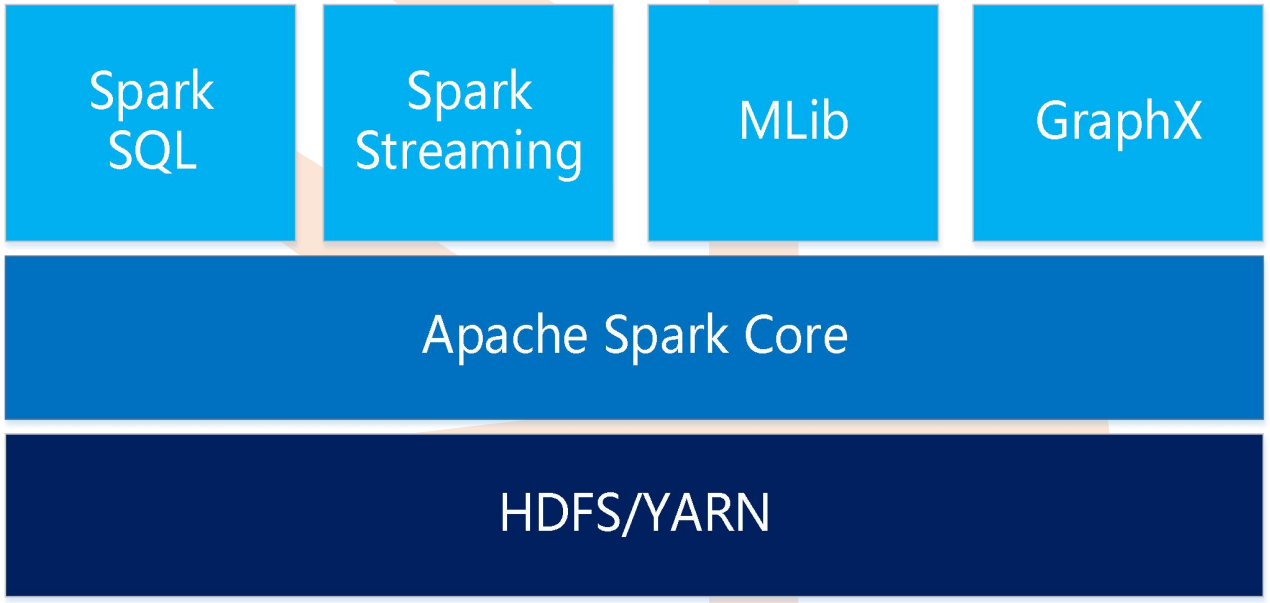
OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

Spark技术栈

- Spark和Hadoop关系： Spark依赖于HDFS文件系统，如果是Spark on YARN部署模式，又依赖于YARN计算框架

- ***Spark Core：*** 基于RDD提供操作接口，利用DAG进行统一的任务规划
- ***Spark SQL：*** Hive的表 + Spark的里。通过把Hive的HQL转化为Spark DAG计算来实现
- ***Spark Streaming：*** Spark的流式计算框架
- ***MLIB：*** Spark的机器学习库，包含常用的机器学习算法
- ***GraphX：*** Spark图并行操作库



由于这些组件满足了很多大数据需求，也满足了很多数据科学任务的算法和计算上的需要，Spark快速流行起来。

OutLine

- Spark的安装
- Scala语言
- Spark基础
- Spark核心
- Spark调优
- Spark常用组件
- Spark实践

环 境 准 备

- SBT编译器安装
- 安装包： sbt-0.13.15.tgz

```
export SBT_HOME=/usr/local/src/sbt

export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$HADOOP_HOME/bin:$SBT_HOME/bin
[root@master bin]#
```

- [root@master spark_test]# mkdir -p spark_wordcount/lib
- [root@master spark_test]# mkdir -p spark_wordcount/project
- [root@master spark_test]# mkdir -p spark_wordcount/src
- [root@master spark_test]# mkdir -p spark_wordcount/target
- [root@master spark_test]# mkdir -p spark_wordcount/src/main/scala

环 境 准 备

- 拷贝spark-assembly-1.6.0-hadoop2.6.0.jar到spark_wordcount/lib目录下
- 写完code后，执行编译：
-]# sbt compile
- 执行打包命令：
-]# sbt package

```
[root@master spark_wordcount]# sbt compile
Getting org.scala-sbt sbt 0.13.15  (this may take some time)...
downloading file:///root/.sbt/preloaded/org.scala-sbt/sbt/0.13.15/jars/sbt.jar ...
        [SUCCESSFUL ] org.scala-sbt#sbt;0.13.15!sbt.jar (200ms)
downloading file:///root/.sbt/preloaded/org.scala-lang/scala-library/2.10.6/jars/scala-library.jar ...
        [SUCCESSFUL ] org.scala-lang#scala-library;2.10.6!scala-library.jar (1112ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/main/0.13.15/jars/main.jar ...
        [SUCCESSFUL ] org.scala-sbt#main;0.13.15!main.jar (1242ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/compiler-interface/0.13.15/jars/compiler-interface.jar ...
        [SUCCESSFUL ] org.scala-sbt#compiler-interface;0.13.15!compiler-interface.jar (546ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/actions/0.13.15/jars/actions.jar ...
        [SUCCESSFUL ] org.scala-sbt#actions;0.13.15!actions.jar (110ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/main-settings/0.13.15/jars/main-settings.jar ...
        [SUCCESSFUL ] org.scala-sbt#main-settings;0.13.15!main-settings.jar (110ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/interface/0.13.15/jars/interface.jar ...
        [SUCCESSFUL ] org.scala-sbt#interface;0.13.15!interface.jar (81ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/io/0.13.15/jars/io.jar ...
        [SUCCESSFUL ] org.scala-sbt#io;0.13.15!io.jar (87ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/ivy/0.13.15/jars/ivy.jar ...
        [SUCCESSFUL ] org.scala-sbt#ivy;0.13.15!ivy.jar (117ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/logging/0.13.15/jars/logging.jar ...
        [SUCCESSFUL ] org.scala-sbt#logging;0.13.15!logging.jar (60ms)
downloading file:///root/.sbt/preloaded/org.scala-sbt/logic/0.13.15/jars/logic.jar ...
        [SUCCESSFUL ] org.scala-sbt#logic;0.13.15!logic.jar (39ms)
                                                          .15/jars/process.jar ...
                                                          ms)
```

```
[warn]   For better performance, hit [ENTER] to switch to interactive mode, or
[warn]   consider launching sbt without any commands, or explicitly passing 'shell'
[info] Loading project definition from /home/badou/spark_test/spark_wordcount/project
[info] Set current project to WordCount (in build file:/home/badou/spark_test/spark_wordcount/)
[info] Compiling 1 Scala source to /home/badou/spark_test/spark_wordcount/target/scala-2.11/classes...
[success] Total time: 51 s, completed May 8, 2017 4:46:31 AM
[root@master spark_wordcount]#
```

```
[root@master spark_wordcount]# sbt package
[warn] Executing in batch mode.
[warn]   For better performance, hit [ENTER] to switch to interactive mode, or
[warn]   consider launching sbt without any commands, or explicitly passing 'shell'
[info] Loading project definition from /home/badou/spark_test/spark_wordcount/project
[info] Set current project to WordCount (in build file:/home/badou/spark_test/spark_wordcount/)
[info] Packaging /home/badou/spark_test/spark_wordcount/target/scala-2.11/wordcount_2.11-1.6.0.jar ...
[info] Done packaging.
[success] Total time: 6 s, completed May 8, 2017 4:50:37 AM
[root@master spark_wordcount]#
```

任务一：WordCount

- 完成基于scala的spark任务，完成wordcount任务

```scala
package spark.example

import org.apache.spark._
import SparkContext._

object WordCount {

    def main(args: Array[String]) {
        if (args.length == 0) {
            System.err.println("Usage: spark.example.WordCount <input> <output>")
                System.exit(1)
        }

        val input_path = args(0).toString
        val output_path = args(1).toString

        val conf = new SparkConf().setAppName("WordCount")
        conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

        val sc = new SparkContext(conf)

        val inputFile = sc.textFile(input_path)

        val countResult =
        .map(word => (word
        .reduceByKey(_ + _
        .map(x => x._1 + "
        .saveAsTextFile(ou
    }
}
```

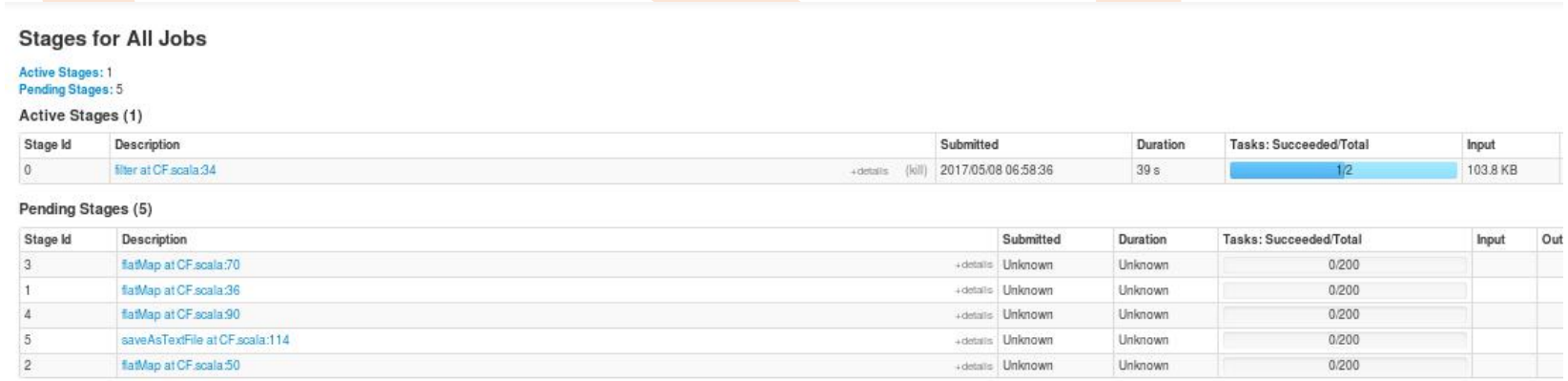
```
[root@master spark_wordcount]# cat build.sbt
name := "WordCount"
version := "1.6.0"
scalaVersion := "2.11.4"
[root@master spark_wordcount]# cat run.sh
#!/usr/bin/env bash

/usr/local/src/spark-1.6.0-bin-hadoop2.6/bin/spark-submit --master yarn-cluster \
    --num-executors 2 \
    --executor-memory '512m' \
    --executor-cores 1 \
    --class spark.example.WordCount ./target/scala-2.11/wordcount_2.11-1.6.0.jar \
    hdfs://master:9000//The_Man_of_Property.txt \
    hdfs://master:9000//spark_word_count_output
```

Show 20 entries

ID	User	Name	Application Type
application_1491925179093_0007	root	spark.example.WordCount	SPARK
application_1491925179093_0006	root	org.apache.spark.examples.SparkPi	SPARK

任务二：重构协同过滤推荐算法

- 完成基于Spark的协同过滤算法
  - 另外：杀死一个任务的方法：
  -]# yarn application -kill application_1491925179093_0012



Stages for All Jobs

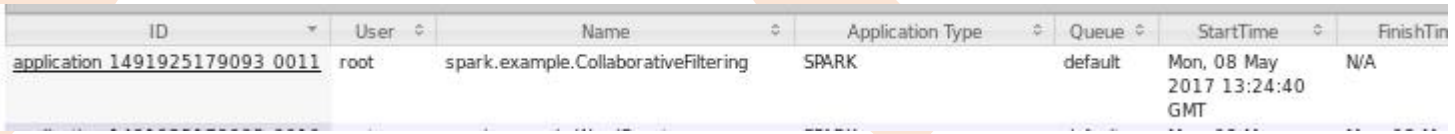
Active Stages: 1
Pending Stages: 5

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
0	filter at CF.scala:34 +details (kill)	2017/05/08 06:58:36	39 s	1/2	103.8 KB

Pending Stages (5)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Out
3	flatMap at CF.scala:70 +details	Unknown	Unknown	0/200		
1	flatMap at CF.scala:36 +details	Unknown	Unknown	0/200		
4	flatMap at CF.scala:90 +details	Unknown	Unknown	0/200		
5	saveAsTextFile at CF.scala:114 +details	Unknown	Unknown	0/200		
2	flatMap at CF.scala:50 +details	Unknown	Unknown	0/200		

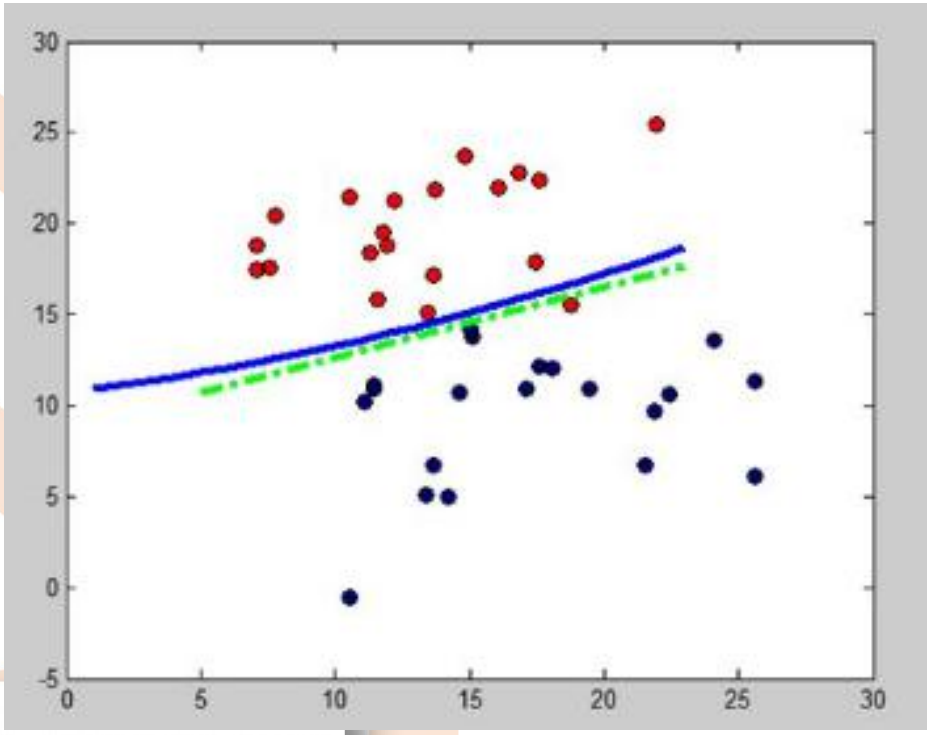


ID	User	Name	Application Type	Queue	StartTime	FinishTin
application_1491925179093_0011	root	spark.example.CollaborativeFiltering	SPARK	default	Mon, 08 May 2017 13:24:40 GMT	N/A

任务三：Mllib的应用

- 完成基于Mllib的朴素贝叶斯机器学习分类算法

```scala
package spark.example

import org.apache.spark.mllib.classification.NaiveBayes
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.{SparkContext,SparkConf}

object naiveBayes {
  def main(args: Array[String]) {


    val conf = new SparkConf().setAppName("naiveBayes")
    val sc = new SparkContext(conf)

    val data = sc.textFile(args(0))
    val parsedData =data.map { line =>
    val parts =line.split(',')
      LabeledPoint(parts(0).toDouble,Vectors.dense(parts(1).split(' ').map(_.toDouble)))
    }

    val splits = parsedData.randomSplit(Array(0.6,0.4),seed = 11L)
    val training =splits(0)
    val test = splits(1)

    val model = NaiveBayes.train(training,lambda = 1.0)

    val predictionAndLabel= test.map(p => (model.predict(p.features),p.label))
    val accuracy =1.0 *predictionAndLabel.filter(x => x._1 == x._2).count() / test.count()

    print
    print
    print
  }
}
```



Q&A

@八斗数据