spark之内存计算框架

课前准备

```
1、对具有分布式存储和分布式计算框架hadoop有一定的了解和认知

2、掌握scala语言的基础语法和面向函数式编程风格

3、掌握搭建一个spark集群

4、为什么要学习spark ？
    spark是基于内存的分布式计算框架，计算速度是非常之快，它的出现是解决了海量数据计算慢的问题，并且有很多不同的应用场景。它是非常受企业青睐！
```

1、通过IDEA工具开发spark的入门案例

- 1、添加依赖

```
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.11</artifactId>
            <version>2.2.0</version>
        </dependency>

```

- 2、代码开发

```
package com.kaikeba

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}
```

```scala


//todo: 利用scala语言开发spark的单词统计程序     wordcount
object WordCount {
  def main(args: Array[String]): Unit = {

    //1、构建SparkConf对象  设置application的名称和master的地址  local[2]:表示本地模拟2个线程去运行程序
        val sparkConf: SparkConf = new SparkConf().setAppName("WordCount").setMaster("local[2]")

    //2、构建SparkContext对象 ，该对象非常重要，它是所有spark程序的执行入口
         val sc = new SparkContext(sparkConf)

    //3、读取数据文件     这里的String表示文件中的每一行数据的类型
        val data: RDD[String] = sc.textFile("./data/test.txt")

    //4、切分每一行 获取所有的单词  类似于scala语言中的先进行map操作然后在进行flatten压扁的操作
          //这里的String表示文件中的每一个单词的类型
            // hadoop spark  hadoop hdfs
         val words: RDD[String] = data.flatMap( x => x.split(" "))

    //5、把每一个单词计为1  (String, Int)  -->String表示每一个单词，Int就是每一个单词计为1类型
              //(hadoop,1) (spark,1)  (hadoop,1) (hdfs,1)
        val wordAndOne: RDD[(String, Int)] = words.map(x => (x,1))



    //6、相同单词出现的1累加
            //reduceByKey：它是先按照key进行分组，然后获取得到了相同的key出现的所有的1
           //hadoop------->List(1,1)
```

```
                //(String, Int):String表示单词，Int：表示每一个单词出现的总次数
        val result: RDD[(String, Int)] = wordAndOne.reduceByKey((x,y)=>x+y)


    //7、收集数据 打印结果
      val finalResult: Array[(String, Int)] = result.collect()
       //finalResult.foreach( x => println(x))
       //finalResult.foreach(println(_))
       finalResult.foreach(println)


    //8、把最终的结果数据保存写入到文件中
    result.saveAsTextFile("./data/out")


    //一行代码实现spark的wordcount程序
     //sc.textFile("./data/test.txt").flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_)

    //9、关闭SparkContext对象
      sc.stop()


  }

}

```

2、提交任务到spark集群中去运行

- 1、代码开发

```
package com.kaikeba

```

```scala
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD

//todo:利用scala语言开发spark的入门程序（提交到spark集群中运行）
object WordCount_Online {
  def main(args: Array[String]): Unit = {

    //1、构建SparkConf对象  设置application的名称
    val sparkConf: SparkConf = new SparkConf().setAppName("WordCount_Online")

    //2、构建SparkContext对象 ，该对象非常重要，它是所有spark程序的执行入口
    val sc = new SparkContext(sparkConf)

    //3、读取HDFS上数据文件     这里的String表示文件中的每一行数据的类型
    val data: RDD[String] = sc.textFile(args(0))

    //4、切分每一行 获取所有的单词  类似于scala语言中的先进行map操作然后在进行flatten压扁的操作
    //这里的String表示文件中的每一个单词的类型
    // hadoop spark  hadoop hdfs
    val words: RDD[String] = data.flatMap( x => x.split(" "))

    //5、把每一个单词计为1  (String, Int)  -->String表示每一个单词，Int就是每一个单词计为1类型
    //(hadoop,1) (spark,1)  (hadoop,1) (hdfs,1)
    val wordAndOne: RDD[(String, Int)] = words.map(x => (x,1))



    //6、相同单词出现的1累加
    //reduceByKey：它是先按照key进行分组，然后获取得到了相同的key出现的所有的1
    //hadoop------->List(1,1)
```

```
    //(String, Int):String表示单词，Int：表示每一个单词出现的总次数
    val result: RDD[(String, Int)] = wordAndOne.reduceByKey((x,y)=>x+y)


    //7、把结果数据保存到hdfs上
      result.saveAsTextFile(args(1))

    //8、关闭SparkContext对象
     sc.stop()



  }
}

```

- 2、把程序打成jar包提交到集群中运行

```
spark-submit \
--master spark://node1:7077 \
--class com.kaikeba.WordCount_Online \
--executor-memory  1g  \
--total-executor-cores 4 \
/home/hadoop/spark_study-1.0.jar \
hdfs://node1:9000/hello.txt \
hdfs://node1:9000/out


--master: 指定master地址，后期程序的运行需要向对应的master申请计算资源
--class:  指定包含main方法的主类

--executor-memory：指定计算任务的时候需要的资源---每一个executor的内存大小
```

```
--total-executor-cores：指定计算任务的时候需要的资源--总的executor需要的cpu核数

/home/hadoop/spark_study-1.0.jar:就是需要打成的jar包
hdfs://node1:9000/hello.txt：  表示main方法中的参数
hdfs://node1:9000/out ：表示main方法中的参数
```

3、spark底层编程抽象之RDD是什么

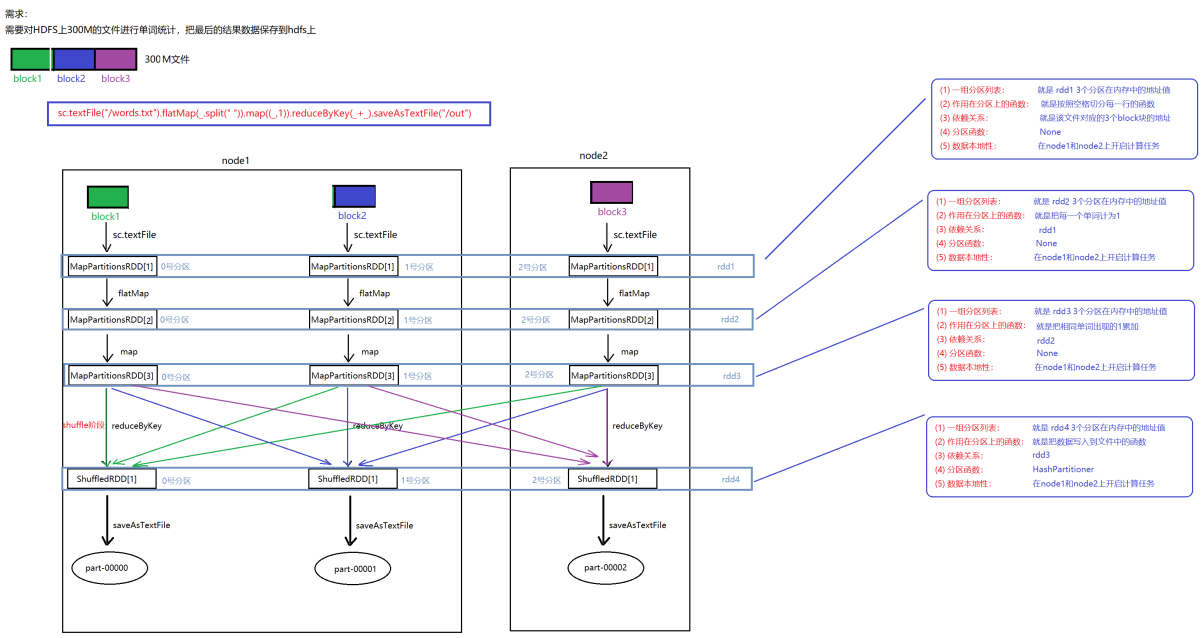
- A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable,partitioned collection of elements that can be operated on in parallel.
  - RDD是以下三个单词的首字母缩写（Resilient Distributed Dataset），它表示弹性分布式数据集，它是spark最基本的数据抽象，它代表了一个不可变、可分区、里面的元素可以被并行操作的集合。
    - Dataset
      - 数据集，在这里可以理解成它是一个集合，集合中存储了很多数据
    - Distributed
      - 它的数据是进行了分布式存储，为了方便于后期进行分布式计算
    - Resilient
      - 弹性，rdd的数据可以保存在内存中或者是磁盘中

4、spark底层编程抽象之RDD的五大特性

```
 (1) A list of partitions
一个RDD有很多个分区，一组分区列表
后期spark的任务是以rdd的分区为单位，一个分区对应一个task线程，spark任务最后是以task线程的方式运行在worker节点上的executor进程中


 (2) A function for computing each split
```

```
作用在每一个分区中的函数
val rdd2=rdd1.map(x =>(x,1))

(3)A list of dependencies on other RDDs
一个rdd会依赖于其他多个rdd，这里就涉及到rdd与rdd之间的依赖关系，spark任务的容错机制就是根据这个特性而来


 (4) Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
    (可选项)  对于kv类型的RDD才有分区函数（必须产生shuffle），如果不是kv类型的RDD它的分区函数是None-----就是表示没有
      在spark中，有2种分区函数：
         第一种：HashPartitioner分区函数（默认值）     对key取hashcode值 然后对分区数取余得到对应的分区号------->key.hashcode % 分区数= 分区号
         第二种：RangePartitioner分区函数 ，它是按照一定的范围进行分区，相同的范围的key会进入到同一个分区。

 (5) Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
   (可选项)  一组最优的数据库位置列表，数据的本地性、数据的位置最优
  spark后期任务的计算会优先考虑存有数据的的节点开启计算任务，数据在哪里，就在当前节点开启计算任务，大大减少数据的网络传输。提升性能。

```

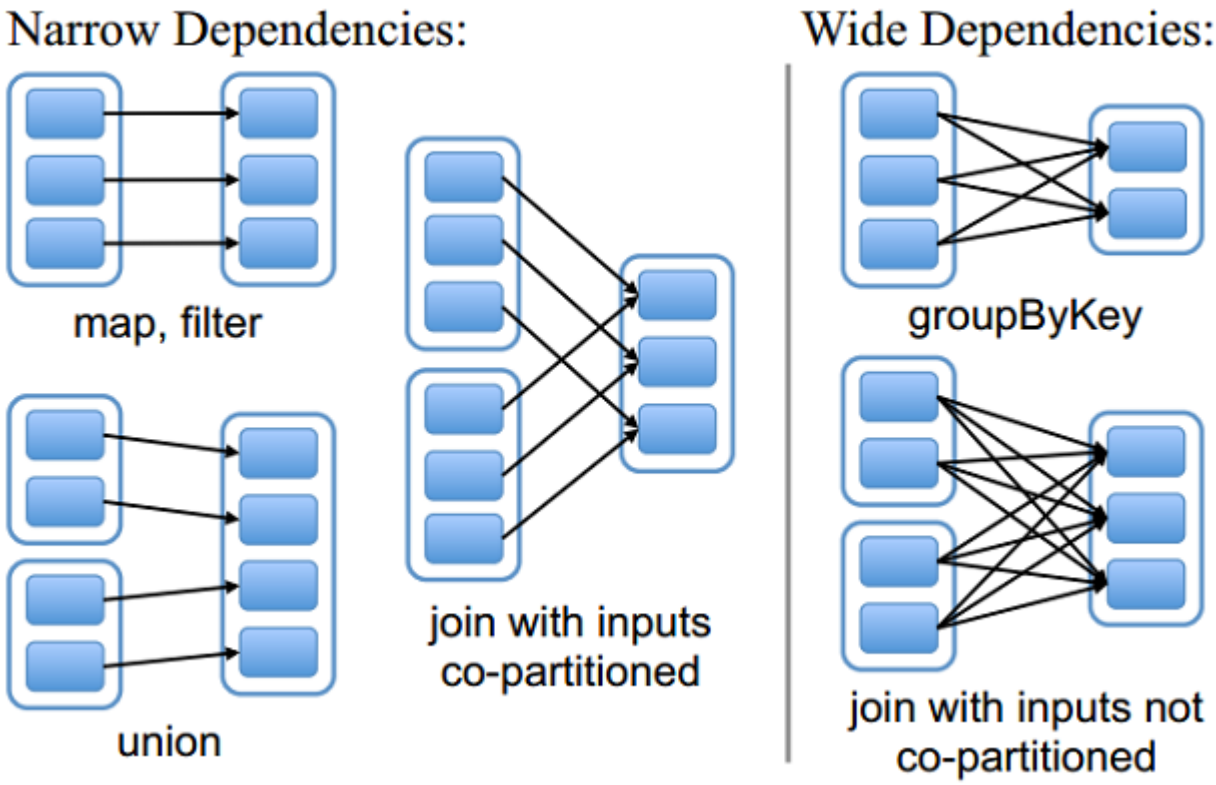
5、基于单词统计案例来深度剖析RDD的五大特性



6、RDD的算子操作分类

- 1、transformation（转换）
  - 它可以实现把一个rdd转换生成一个新的rdd，它是延迟加载，不会立即触发任务的真正运行
  - 比如 flatMap/map/reduceByKey
- 2、action(动作)
  - 它会触发任务的真正运行
  - 比如 collect/saveAsTextFile

7、RDD的依赖关系



- rdd与rdd之间的依赖关系有2种：
  - 窄依赖
    - 它指的是父RDD的partition数据最多只被子RDD的一个partition所使用
    - 比如 map/filter/flatMap等等
    - 它是不会产生shuffle
  - 宽依赖
    - 它指的是子RDD多个partition数据会依赖于父RDD的同一partition分区数据
    - 比如 reduceByKey / groupByKey /sortByKey /groupBy 等等
    - 它会产生shuffle

- lineage(血统)
  - 它就是记录下rdd上的一些转换操作
  - 后期如果某个rdd的部分分区数据丢失之后，可以通过血统这层关系进行重新计算来恢复得到丢失的分区数据。

8、RDD的缓存机制

8.1 rdd的缓存是什么

```
可以把rdd的数据缓存在内存或者是磁盘中，后续需要用到这份数据，就可以直接从缓存中获取得到，避免了数据的重复计算。
```

8.2 如何对rdd设置缓存

```
可以调用rdd中的cache和persist2个方法

cache和persist的区别：
cache:     其本质是调用了pesist方法，它是把数据缓存在内存中。
persist：可以把数据保存在内存或者是磁盘中，该方法中可以传入不同的缓存级别，这些缓存级别都被定义在这个Object中（StorageLevel）
```

8.3 具体使用cache和persist方法

- 1、需要对rdd调用cache和perisist方法
  - rdd1.cache/persist(缓存级别)
- 2、后续需要有触发任务执行的action操作
  - 比如
    - rdd1.collect

8.4 什么时候对rdd设置缓存

- 1、某个rdd后期被使用了多次

```
val rdd1=sc.textFile("/words.txt")
rdd1.cache
val rdd2=rdd1.flatMap(_.split(" "))
val rdd3=rdd1.map(x=>(x,1))

这里通过rdd1的数据来得到rdd2和rdd3,这里rdd1的数据就会被计算了2次。这里就可以对rdd1的数据设置缓存，后续可以直接从缓存中获取得到。
```

- 2、经过一系列大量的算子操作之后得到了某个rdd，该RDD的数据是来之不易

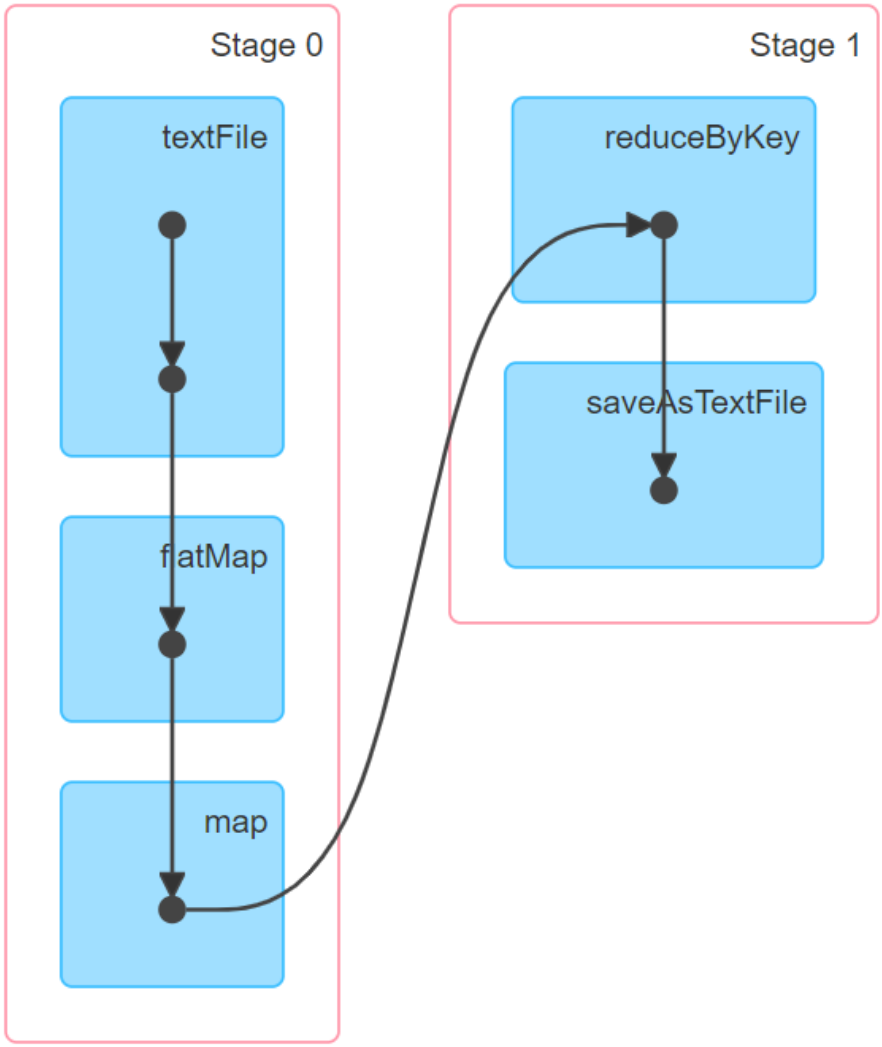
```
val rdd1=sc.textFile(xxx).flatMap.map.xxx.xxxx.xxxxx.xxxx.xxxx.....
```

8.5 清除缓存

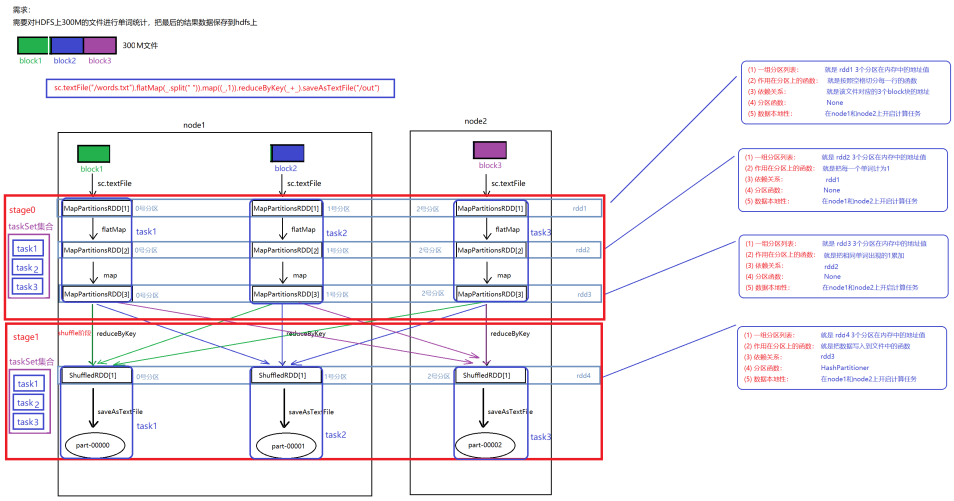
- 1、系统自动清除
  - 当应用程序执行完成之后，缓存数据也就消失了。
- 2、手动清除
  - rdd1.unpersist(true)

9、DAG有向无环图的构建和划分stage

- Directed Acyclic Graph
  - 它是按照程序中的rdd之间的依赖关系，生成了一张有方向无闭环的图



- 后期会对DAG有向无环图划分成不同的stage（调度阶段）
  - 宽依赖是划分stage的依据



1、为什么要划分stage（调度阶段）？

由于一个job任务中可能有大量的宽窄依赖，由于窄依赖不会产生shuffle，宽依赖会产生shuffle。

后期划分完stage之后，在同一个stage中只有窄依赖，并没有宽依赖，这些窄依赖对应的task是可以互相独立的去运行

划分完stage之后，它内部是有很多可以并行运行的task。

2、如何划分stage？　　划分stage就是以宽依赖进行划分

(1)、生成DAG有向无环图之后，从最后一个rdd往前推，先创建一个stage，它是最后一个stage

(2)、如果遇到了窄依赖 就把该rdd加入到stage中，如果遇到了宽依赖，就从宽依赖切开，最后一个stage也就划分结束了

(3)、后面重新创建一个新的stage，还是按照第二步操作继续往前推，一直推到最开始的rdd，整个划分stage也就结束了。

3、stage内部的逻辑

（1）每一个stage中按照rdd的分区划分成了很多个可以并行运行的task

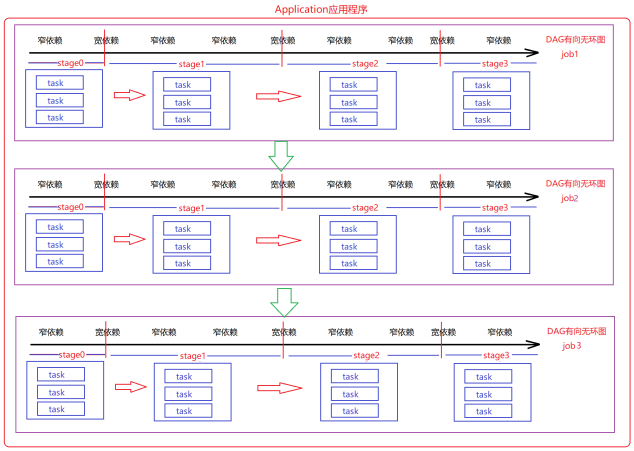
（2）把每一个stage中这些可以并行运行的task都封装在一个taskSet集合中

（3）rdd与rdd之间存在对应的依赖关系，stage与stage之间也存在对应的依赖关系，前面stage的task先运行，运行完成之后，后面stage中的task再运行，也就是说前面stage中task它的输出结果数据，是后面stage中task输入数据。

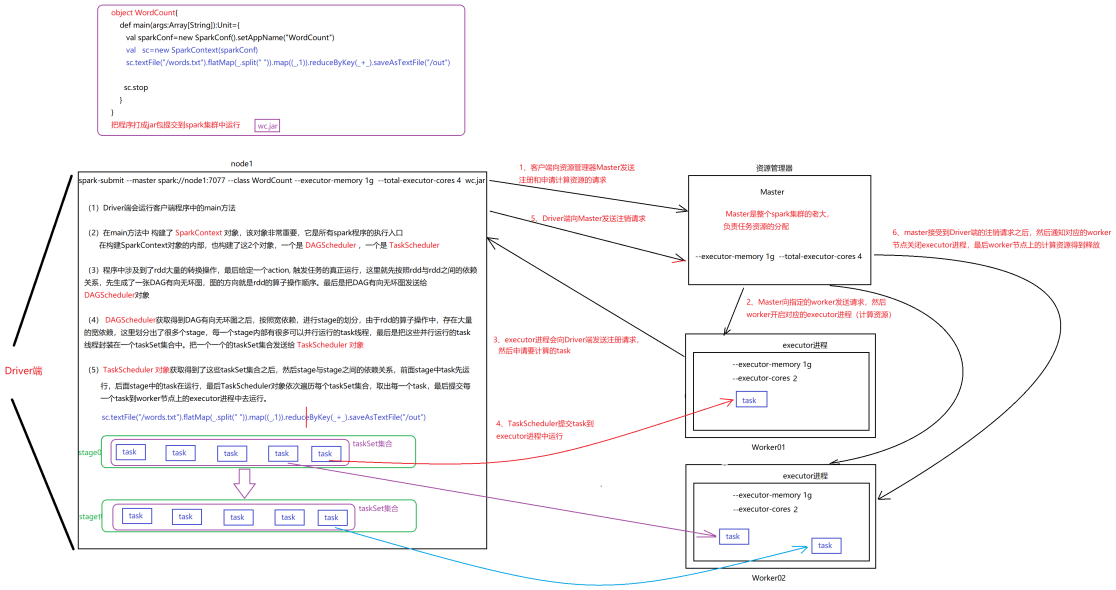
4、Application 、Job、Stage、Task之间的关系

application是spark的一个应用程序，它是包含了客户端写好的代码以及任务运行的时候需要的资源信息。

后期再一个application中有很多个action操作，一个action操作就是一个job，一个job会存在大量的宽依赖，后期按照宽依赖进行stage的划分，一个job又产生了很多个stage，每一个stage内部有很多并行运行的task.



10、基于wordcount程序剖析spark任务的提交、划分、调度流程



11、通过spark开发实现点击流日志分析案例

11.1 PV

```
package com.kaikeba

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

//TODO:通过spark实现点击流日志分析-------PV
object PV {

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

    //1、构建SparkConf对象
      val sparkConf: SparkConf = new SparkConf().setAppName("PV").setMaster("local[2]")

    //2、构建SparkContext对象
```

```
        val sc = new SparkContext(sparkConf)
        sc.setLogLevel("warn")

    //3、读取数据文件
        val data: RDD[String] =
sc.textFile("./data/access.log")

    //4、统计pv
         val pv: Long = data.count()

        println("pv:"+pv)

    sc.stop()

  }
}

```

11.2 UV

```
package com.kaikeba

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

//TODO:通过spark实现点击流日志分析-------UV
object UV {

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

     //1、构建SparkConf对象
        val sparkConf: SparkConf = new
SparkConf().setAppName("UV").setMaster("local[2]")

    //2、构建SparkContext对象
        val sc = new SparkContext(sparkConf)
        sc.setLogLevel("warn")
```

```
17
18      //3、读取数据文件
19          val data: RDD[String] =
    sc.textFile("./data/access.log")
20
21      //4、获取用户的唯一标识  ip
22          val ipRDD: RDD[String] = data.map( x=> x.split(" ")
    (0))
23
24      //5、ip去重
25          val distinctRDD: RDD[String] = ipRDD.distinct()
26
27      //6、统计uv
28          val uv: Long = distinctRDD.count()
29          println("uv:"+uv)
30
31
32      sc.stop()
33
34    }
35  }
36
```

11.3 TopN

```
1  package com.kaikeba
2
3  import org.apache.spark.rdd.RDD
4  import org.apache.spark.{SparkConf, SparkContext}
5
6  //TODO:通过spark实现点击流日志分析-------TopN(求访问url地址最多的
    前N位)
7  object TopN {
8
9    def main(args: Array[String]): Unit = {
10
11     //1、构建SparkConf对象
```

```
12          val sparkConf: SparkConf = new
    SparkConf().setAppName("TopN").setMaster("local[2]")
13
14      //2、构建SparkContext对象
15          val sc = new SparkContext(sparkConf)
16          sc.setLogLevel("warn")
17
18      //3、读取数据文件
19          val data: RDD[String] =
    sc.textFile("./data/access.log")
20
21
22      //4、过滤出丢失的数据记录，直接舍弃掉
23        val rightRDD: RDD[String] = data.filter(x =>x.split("
    ").length >10)
24
25      //5、获取每条数据的url地址 (url,1)
26          val urlAndOne: RDD[(String, Int)] =
    rightRDD.map(x=>(x.split(" ")(10),1))
27
28      //6、相同的url出现的1累加
29          val result: RDD[(String, Int)] =
    urlAndOne.reduceByKey(_+_)
30
31      //7、求出访问url次数最多的前5位    第一个参数默认是true表示升序，
    可以改为false表示降序
32      val sortedRDD: RDD[(String, Int)] =
    result.sortBy(x=>x._2,false)
33      val top5: Array[(String, Int)] = sortedRDD.take(5)
34      top5.foreach(println)
35
36
37
38      sc.stop()
39
40    }
41  }
42
```

12、通过spark开发实现ip归属地查询案例

- 1、代码开发

```
package com.kaikeba

import org.apache.spark.broadcast.Broadcast
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

//todo: 通过spark实现ip归属地查询
object Iplocation {

  //实现把ip地址转换成Long类型数字     192.168.200.100
  def ip2Long(ip: String): Long = {
      val ips: Array[String] = ip.split("\\.")
      var ipNum:Long=0L

      //遍历
       for( i <- ips){
         ipNum= i.toLong | ipNum << 8L
       }

    ipNum
  }

   //利用二分查询，查询到数字在数组中的下标
  def binarySearch(ipNum: Long, city_ip_array:
Array[(String, String, String, String)]): Int = {
    //定义开始下标
    var start=0

    //定义结束下标
    var end =city_ip_array.length-1

    while (start <= end){
```

```
      val middle=(start+end)/2

      if(ipNum >= city_ip_array(middle)._1.toLong && ipNum <= city_ip_array(middle)._2.toLong){
          //return 可以直接退出while
               return  middle
      }

      if(ipNum < city_ip_array(middle)._1.toLong){
             end= middle-1
      }

      if(ipNum > city_ip_array(middle)._2.toLong){
            start=middle+1
      }

    }
    -1

  }

  def main(args: Array[String]): Unit = {
      //1、创建SparkConf对象
      val sparkConf: SparkConf = new SparkConf().setAppName("Iplocation").setMaster("local[2]")


     //2、构建SparkContext对象
      val sc = new SparkContext(sparkConf)
      sc.setLogLevel("warn")


     //3、加载城市ip信息数据，获取  (ip开始数字、ip结束数字、经度、纬度)
      val city_ip_rdd: RDD[(String, String, String, String)] = sc.textFile("./data/ip.txt").map(x=>x.split("\\|")).map(x=>(x(2),x(3),x(x.length-2),x(x.length-1)))
```

```
64
65
66          //使用spark的广播变量把共同的数据广播到参与计算的worker节点
67              val cityIpBroadcast: Broadcast[Array[(String,
String, String, String)]] =
sc.broadcast(city_ip_rdd.collect())
68
69          //4、读取运营商日志数据
70              val userIpsRDD: RDD[String] =
sc.textFile("./data/20090121000132.394251.http.format").ma
p(x=>x.split("\\|")(1))
71
72
73          //5、遍历userIpsRDD  获取每一个ip地址，然后转换Long类型数字，
去广播变量值中去比较
74          val result: RDD[((String, String), Int)] =
userIpsRDD.mapPartitions(iter => {
75              //获取广播变量的值
76              val city_ip_array: Array[(String, String, String,
String)] = cityIpBroadcast.value
77
78              //获取每一个ip地址
79              iter.map(ip => {
80
81                  // 把ip地址转换成Long类型数字
82                  val ipNum: Long = ip2Long(ip)
83
84                  //需要拿到ipNum数字去广播变量值中进行匹配，获取该数字在广播
变量数组中的下标
85                  val index: Int = binarySearch(ipNum,
city_ip_array)
86
87                  //获取对应的信息
88                  val result: (String, String, String, String) =
city_ip_array(index)
89
90                  //封装结果数据  进行返回  ((经度，纬度),1)
91                  ((result._3, result._4), 1)
```

```
92
93          })
94
95      })
96
97      //6、相同经纬度出现的1累加
98      val finalResult: RDD[((String, String), Int)] =
    result.reduceByKey(_+_)
99      finalResult.foreach(println)
100
101
102     sc.stop()
103
104
105   }
106 }
107
```