1 Shuffle调优

1.1 调优概述

大多数Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。但是也必须提醒大家的是，影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占到一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解shuffle的原理，以及相关参数的说明，同时给出各个参数的调优建议。

1.2. ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。而随着Spark的版本的发展，ShuffleManager也在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

下面我们详细分析一下HashShuffleManager和SortShuffleManager的原理。

1.3. HashShuffleManager运行原理

未经优化的HashShuffleManager

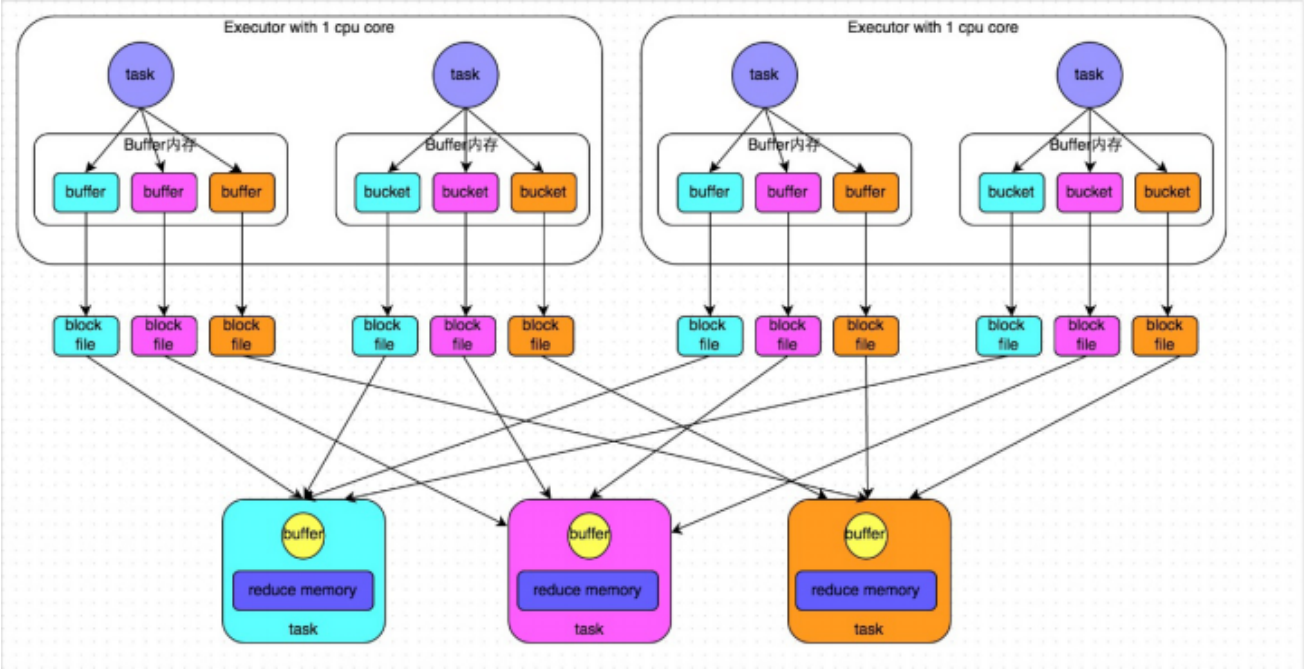
下图说明了未经优化的HashShuffleManager的原理。这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

我们先从shuffle write开始说起。shuffle write阶段，主要就是在一个stage结束计算之后，为了下一个stage可以执行shuffle类的算子（比如reduceByKey），而将每个task处理的数据按key进行“分类”。所谓“分类”，就是对相同的key执行hash算法，从而将相同key都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游stage的一个task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。由此可见，未经优化的shuffle write操作所产生的磁盘文件的数量是极其惊人的。

接着我们来说说shuffle read。shuffle read，通常就是一个stage刚开始时要做的事情。此时该stage的每一个task就需要将上一个stage的计算结果中的所有相同key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行key的聚合或连接等操作。由于shuffle write的过程中，task给下游stage的每个task都创建了一个磁盘文件，因此shuffle read的过程中，每个task只要从上游stage的所有task所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到buffer缓冲中进行聚合操作。以此类推，直到最后将所有数据到拉取完，并得到最终的结果。



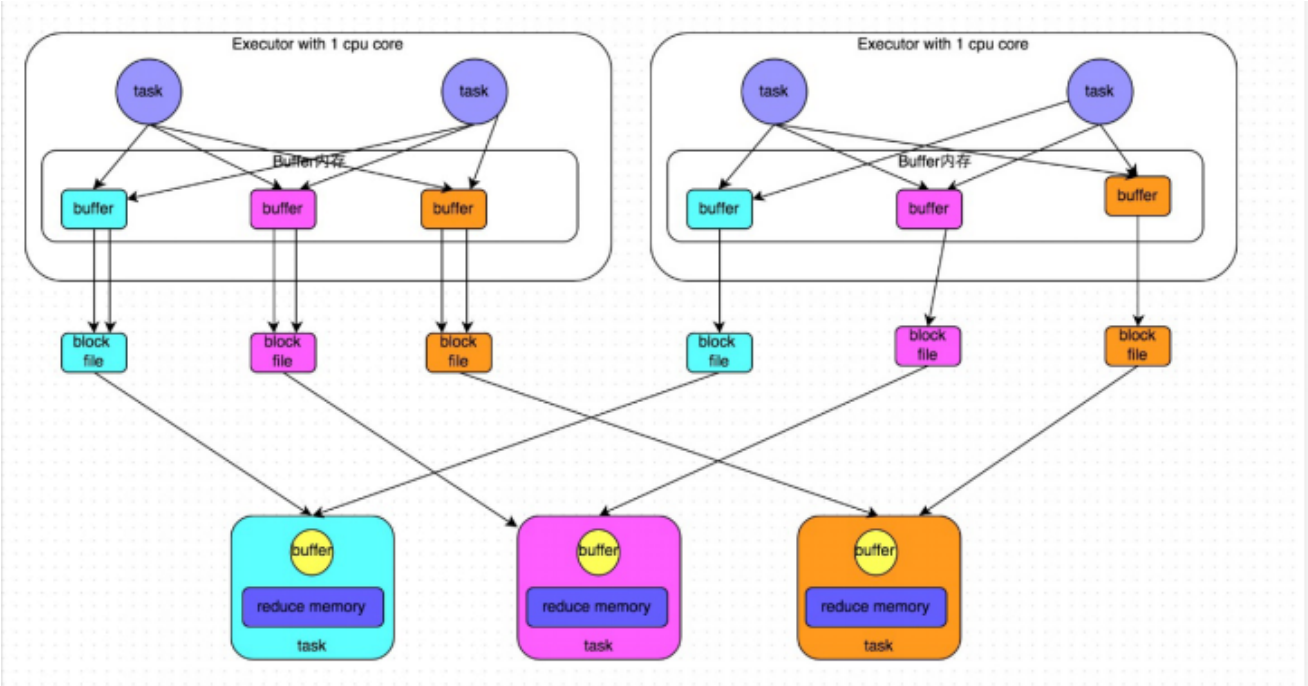
优化后的HashShuffleManager

下图说明了优化后的HashShuffleManager的原理。这里说的优化，是指我们可以设置一个参数，spark.shuffle.consolidateFiles。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

开启consolidate机制之后，在shuffle write过程中，task就不是为下游stage的每个task创建一个磁盘文件了。此时会出现shuffleFileGroup的概念，每个shuffleFileGroup会对应一批磁盘文件，磁盘文件的数量与下游stage的task数量是相同的。一个Executor上有多少个CPU core，就可以并行执行多少个task。而第一批并行执行的每个task都会创建一个shuffleFileGroup，并将数据写入对应的磁盘文件内。

当Executor的CPU core执行完一批task，接着执行下一批task时，下一批task就会复用之前已有的shuffleFileGroup，包括其中的磁盘文件。也就是说，此时task会将数据写入已有的磁盘文件中，而不会写入新的磁盘文件中。因此，consolidate机制允许不同的task复用同一批磁盘文件，这样就可以有效将多个task的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升shuffle write的性能。

假设第二个stage有100个task，第一个stage有50个task，总共还是有10个Executor，每个Executor执行5个task。那么原本使用未经优化的HashShuffleManager时，每个Executor会产生500个磁盘文件，所有Executor会产生5000个磁盘文件的。但是此时经过优化之后，每个Executor创建的磁盘文件的数量的计算公式为：CPU core的数量 * 下一个stage的task数量。也就是说，每个Executor此时只会创建100个磁盘文件，所有Executor只会创建1000个磁盘文件。



1.4. SortShuffleManager运行原理

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。当shuffle read task的数量小于等于spark.shuffle.sort.bypassMergeThreshold参数的值时（默认为200），就会启用bypass机制。

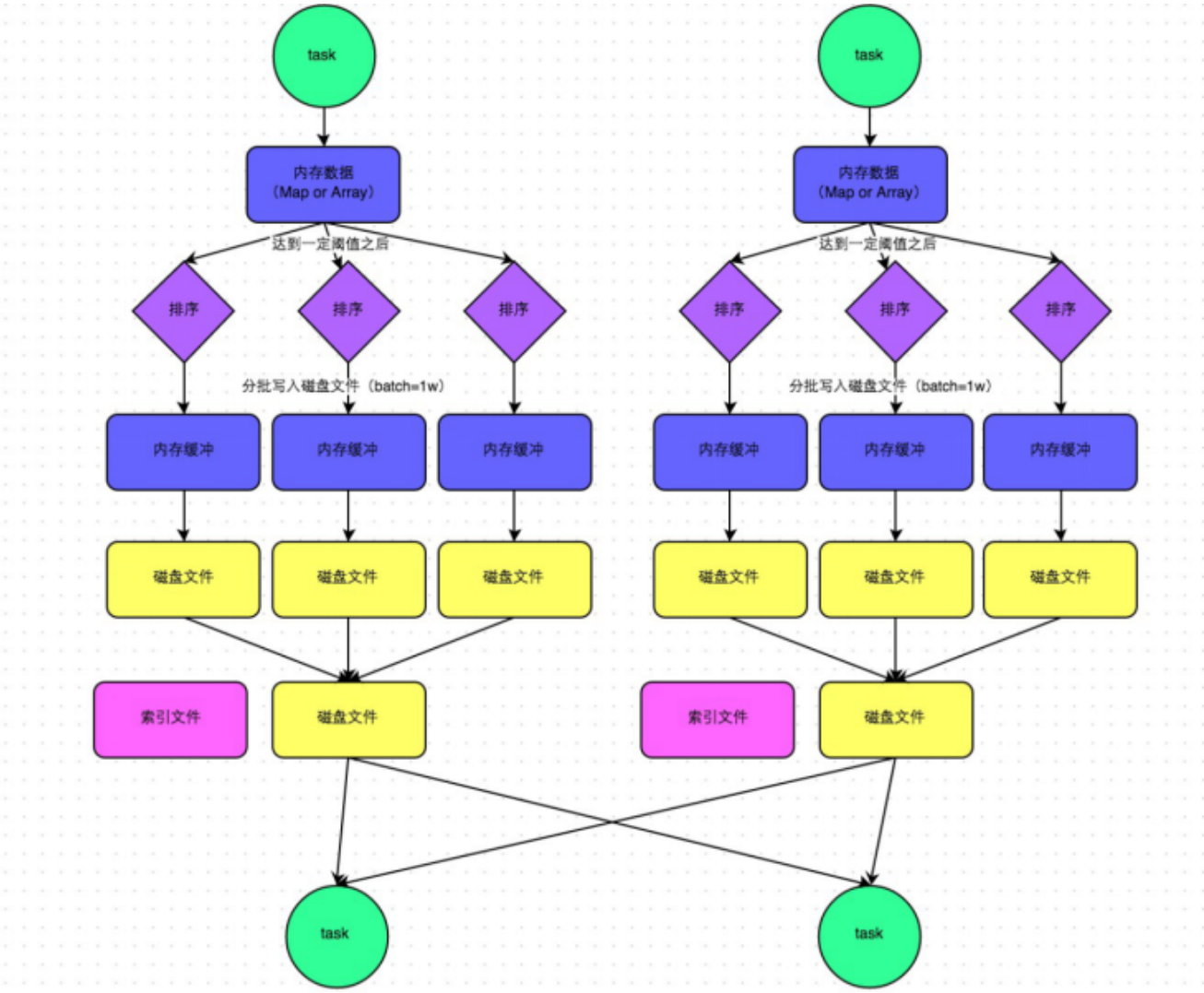
普通运行机制

下图说明了普通的SortShuffleManager的原理。在该模式下，数据会先写入一个内存数据结构中，此时根据不同的shuffle算子，可能选用不同的数据结构。如果是reduceByKey这种聚合类的shuffle算子，那么会选用Map数据结构，一边通过Map进行聚合，一边写入内存；如果是join这种普通的shuffle算子，那么会选用Array数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

在溢写到磁盘文件之前，会先根据key对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的batch数量是10000条，也就是说，排序好的数据，会以每批1万条数据的形式分批写入磁盘文件。写入磁盘文件是通过Java的BufferedOutputStream实现的。BufferedOutputStream是Java的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘IO次数，提升性能。

一个task将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是merge过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个task就只对应一个磁盘文件，也就意味着该task为下游stage的task准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个task的数据在文件中的start offset与end offset。

SortShuffleManager由于有一个磁盘文件merge的过程，因此大大减少了文件数量。比如第一个stage有50个task，总共有10个Executor，每个Executor执行5个task，而第二个stage有100个task。由于每个task最终只有一个磁盘文件，因此此时每个Executor上只有5个磁盘文件，所有Executor只有50个磁盘文件。



bypass运行机制

下图说明了bypass SortShuffleManager的原理。bypass运行机制的触发条件如下：

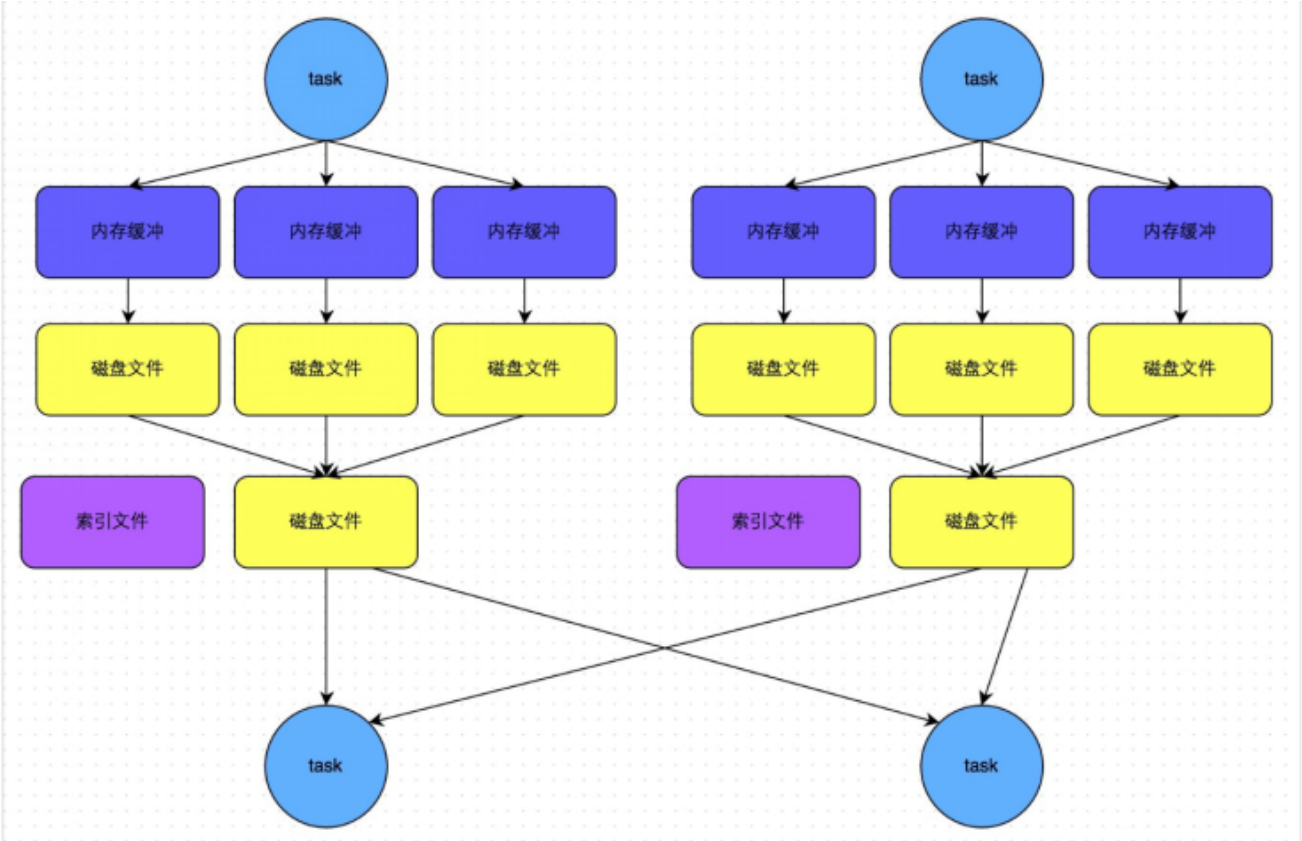
shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。

不是聚合类的shuffle算子（比如reduceByKey）。

此时task会为每个下游task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。



1.5. shuffle相关参数调优

以下是Shffule过程中的一些主要参数，这里详细讲解了各个参数的功能、默认值以及基于实践经验给出的调优建议。

spark.shuffle.file.buffer

默认值：32k

参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.reducer.maxSizeInFlight

默认值：48m

参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.shuffle.io.maxRetries

默认值：3

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

spark.shuffle.io.retryWait

默认值：5s

参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。

调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

spark.shuffle.memoryFraction

默认值：0.2

参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。

调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%左右。

spark.shuffle.manager

默认值：sort

参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。

调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

spark.shuffle.sort.bypassMergeThreshold

默认值：200

参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。

2 Accumulator累加器

累加器用来对信息进行聚合，通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱 动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本， 更新这些副本的值也不会影响驱动器中的对应变量。 如果我们想实现所有分片处理时更新共享变量的功能，那么累加器可以实现我们想要的效果。

1. Spark提供了一个默认的累加器,只能用于求和没啥用
2. 如何使用:

   2.1.通过SparkContext对象.accumulator(0) var sum = sc.accumulator(0)

通过accumulator声明一个累加器,0为初始化的值

2.2.通过转换或者行动操作,通过sum +=n 来使用

2.3.如何取值? 在Driver程序中,通过 sum .value来获取值

3.累加器是懒执行,需要行动触发

例子: 数据计算相加

```
 val numbers = sc .parallelize(List(1,2,3,4,5,6),2)
    println(numbers.partitions.length)
    //为什么sum值通过计算过后还是0
    //因为foreach是没有返回值,整个计算过程都是在executor端完后
    //foreach是在driver端运行所以打印的就是 0,foreach没有办法获取数据
    //var sum = 0
//    numbers.foreach(num =>{
//      sum += num
//    })
//    println(sum)
    //建议点击看原码 可以发现当前方法已经过时了,@deprecated("use AccumulatorV2", "2.0.0")
    //所以以后使用时候需要使用自定义累加器
    var sum = sc.accumulator(0)
        numbers.foreach(num =>{
          sum += num
        })
        println(sum.value)
  }
```

自定义累加器

自定义累加器类型的功能在1.X版本中就已经提供了，但是使用起来比较麻烦，在2.0版本后，累加器的易用性有了较大的改进，而且官方还提供了一个新的抽象类：AccumulatorV2来提供更加友好的自定义类型累加器的实现方式。官方同时给出了一个实现的示例：CollectionAccumulator类，这个类允许以集合的形式收集spark应用执行过程中的一些信息。例如，我们可以用这个类收集Spark处理数据时的一些细节，当然，由于累加器的值最终要汇聚到driver端，为了避免 driver端的outofmemory问题，需要对收集的信息的规模要加以控制，不宜过大。

案例1：使用系统的原生累加器

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

    val conf = new SparkConf().setAppName("accumulator").setMaster("local[*]")
    //2.创建SparkContext 提交SparkApp的入口
    val sc = new SparkContext(conf)
    val num1 = sc.parallelize(List(1, 2, 3, 4, 5, 6), 2)
    val num2 = sc.parallelize(List(1.1, 2.2, 3.3, 4.4, 5.5, 6.6), 2)
    //创建并注册一个long accumulator，从“0”开始，用“add”累加
    def longAccumulator(name: String): LongAccumulator = {
      val acc = new LongAccumulator
      sc.register(acc, name)
      acc
    }
    val acc1 =  longAccumulator("kk")
    num1.foreach(x => acc1.add(x))
    println(acc1.value)

    //创建并注册一个double accumulator，从“0”开始，用“add”累加
    def doubleAccumulator(name: String): DoubleAccumulator = {
      val acc = new DoubleAccumulator
      sc.register(acc, name)
      acc
    }
    val acc2 =  doubleAccumulator("kk")
    num1.foreach(x => acc2.add(x))
    println(acc2.value)
    创建并注册一个double accumulator，从“0”开始，用“add”累加
    def collectionAccumulator(name: String): CollectionAccumulator[Int] = {
      val acc = new CollectionAccumulator[Int]
      sc.register(acc, name)
      acc
    }
    val acc3 =  collectionAccumulator("kk")
    num1.foreach(x => acc3.add(x))
    println(acc3.value)
    sc.stop()
  }
```

案例2:

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.util.AccumulatorV2
//在继承的时候需要执行泛型 即 可以计算IN类型的输入值，产生Out类型的输出值
```

```scala
//继承后必须实现提供的方法
class MyAccumulator extends  AccumulatorV2[Int,Int]{
  //创建一个输出值的变量
  private var sum:Int = _

  //必须重写如下方法:
  //检测方法是否为空
  override def isZero: Boolean = sum == 0
  //拷贝一个新的累加器
  override def copy(): AccumulatorV2[Int, Int] = {
    //需要创建当前自定累加器对象
    val myaccumulator = new MyAccumulator()
    //需要将当前数据拷贝到新的累加器数据里面
   //也就是说将原有累加器中的数据拷贝到新的累加器数据中
    //ps:个人理解应该是为了数据的更新迭代
    myaccumulator.sum = this.sum
    myaccumulator
  }
  //重置一个累加器 将累加器中的数据清零
  override def reset(): Unit = sum = 0
  //每一个分区中用于添加数据的方法(分区中的数据计算)
  override def add(v: Int): Unit = {
    //v 即 分区中的数据
     //当累加器中有数据的时候需要计算累加器中的数据
     sum += v
  }
  //合并每一个分区的输出(将分区中的数进行汇总)
  override def merge(other: AccumulatorV2[Int, Int]): Unit = {
          //将每个分区中的数据进行汇总
            sum += other.value

  }
 //输出值(最终累加的值)
  override def value: Int = sum
}

object  MyAccumulator{
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("MyAccumulator").setMaster("local[*]")
    //2.创建SparkContext 提交SparkApp的入口
    val sc = new SparkContext(conf)
    val numbers = sc .parallelize(List(1,2,3,4,5,6),2)
    val accumulator = new MyAccumulator()
    //需要注册
    sc.register(accumulator,"acc")
    //切记不要使用Transformation算子 会出现无法更新数据的情况
    //应该使用Action算子
    //若使用了Map会得不到结果
    numbers.foreach(x => accumulator.add(x))
    println(accumulator.value)
  }
}
```

案例3累加器陷阱

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.{PairRDDFunctions, RDD}
import org.apache.spark.util.{CollectionAccumulator, DoubleAccumulator, LongAccumulator}

object SubjectCount2 {

  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("accumulator").setMaster("local[*]")
    //2.创建SparkContext 提交SparkApp的入口
    val sc = new SparkContext(conf)
    var accum=  new LongAccumulator
    sc.register(accum, "acc")
    val data = sc.parallelize(1 to 10)

    //用accumulator统计偶数出现的次数，同时偶数返回0，奇数返回1

    val newData = data.map{x => {
      if(x%2 == 0){
        accum.add(1)
      }else 1
    }}

    //使用action操作触发执行
    newData.count

    //此时accum的值为5，是我们要的结果
   println(accum.value)

    //继续操作，查看刚才变动的数据,foreach也是action操作
    newData.foreach(println)

    //上个步骤没有进行累计器操作，可是累加器此时的结果已经是10了
    //这并不是我们想要的结果

   println(accum.value)
    sc.stop()
  }
}
```

总结:

1.累加器的创建:

1.1.创建一个累加器的实例

1.2.通过sc.register()注册一个累加器

1.3.通过累加器实名.add来添加数据

1.4.通过累加器实例名.value来获取累加器的值

2.最好不要在转换操作中访问累加器(因为血统的关系和转换操作可能执行多次),最好在行动操作中访问

作用:

1.能够精确的统计数据的各种数据例如:

可以统计出符合userID的记录数,在同一个时间段内产生了多少次购买,可以使用ETL进行数据清洗,并使用Accumulator来进行数据的统计

2.作为调试工具,能够观察每个task的信息,通过累加器可以在sparkIUI观察到每个task所处理的记录数

3 Broadcast广播变量

广播变量用来高效分发较大的对象。向所有工作节点发送一个 较大的只读值，以供一个或多个 Spark 操作使用。比如，如果你的应用需要向所有节点发 送一个较大的只读查询表，甚至是机器学习算法中的一个很大的特征向量，广播变量用起 来都很顺手。

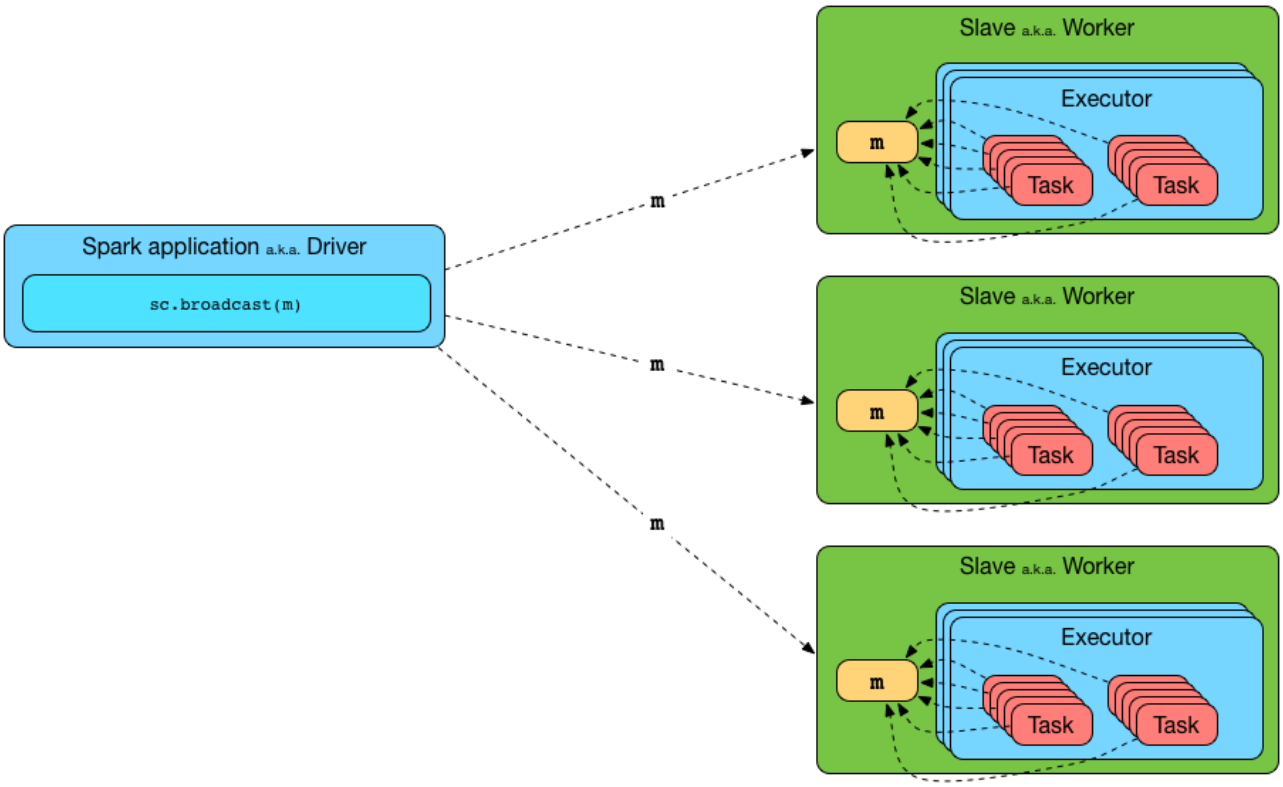
问题说明:

```
/**
以下代码就会出现一个问题:
list是在driver端创建的，但是因为需要在executor端使用，所以driver会把list以task的形式发送到excutor端,也就相当于在executor需要复制一份,如果有很多个task，就会有很多给excutor端携带很多个list，如果这个list非常大的时候，就可能会造成内存溢出
*/
val conf = new SparkConf().setAppName("BroadcastTest").setMaster("local")
      val sc = new SparkContext(conf)
       //list是在driver端创建也相当于是本地变量
      val list = List("hello java")
      //算子部分是在Excecutor端执行
      val lines = sc.textFile("dir/file")
      val filterStr = lines.filter(list.contains(_))
      filterStr.foreach(println)
```

广播变量的好处，不是每个task一份变量副本，而是变成每个节点的executor才一份副本。这样的话， 就可以让变量产生的副本大大减少。

task在运行的时候，想要使用广播变量中的数据，此时首先会在自己本地的Executor对应的BlockManager中， 尝试获取变量副本；如果本地没有，那么就从Driver远程拉取变量副本，并保存在本地的BlockManager中； 此后这个executor上的task，都会直接使用本地的BlockManager中的副本。 executor的BlockManager除了从driver上拉取，也可能从其他节点的BlockManager上拉取变量副本。 HttpBroadcast　TorrentBroadcast（默认）

BlockManager 负责管理某个Executor对应的内存和磁盘上的数据，尝试在本地BlockManager中找map




```
val conf = new SparkConf().setAppName("BroadcastTest").setMaster("local")
     val sc = new SparkContext(conf)
      //list是在driver端创建也相当于是本地变量
     val list = List("hello java")
     //封装广播变量
     val broadcast = sc.broadcast(list)
     //算子部分是在Excecutor端执行
     val lines = sc.textFile("dir/file")
     //使用广播变量进行数据处理 value可以获取广播变量的值
     val filterStr = lines.filter(broadcast.value.contains(_))
     filterStr.foreach(println)
```

总结:

广播变量的过程如下：

(1) 通过对一个类型 T 的对象调用 SparkContext.broadcast 创建出一个 Broadcast[T] 对象。 任何可序列化的类型都可以这么实现。

(2) 通过 value 属性访问该对象的值(在 Java 中为 value() 方法)。

(3) 变量只会被发到各个节点一次，应作为只读值处理(修改这个值不会影响到别的节点)。

能不能将一个RDD使用广播变量广播出去？

不能，因为RDD是不存储数据的。可以将RDD的结果广播出去。

广播变量只能在Driver端定义，不能在Executor端定义。

广播变量的好处:

举例来说 50个executor，1000个task。一个map，10M。 默认情况下，1000个task，1000份副本。10G的数据，网络传输，在集群中，耗费10G的内存资源。 如果使用了广播变量。50个execurtor，50个副本。500M的数据，网络传输， 而且不一定都是从Driver传输到每个节点，还可能是就近从最近的节点的executor的bockmanager 上拉取变量副本，网络传输速度大大增加；500M的内存消耗。 10000M，500M，20倍。20倍~以上的网络传输性能消耗的降低；20倍的内存消耗的减少。 对性能的提升和影响，还是很客观的。 虽然说，不一定会对性能产生决定性的作用。比如运行30分钟的spark作业，可能做了广播变量以后，速度快了2分钟，或者5分钟。但是一点一滴的调优，积少成多。最后还是会有效果的。

4 jdbcRDD

```
package com.qf.gp1705.day10

import java.sql.{Date, DriverManager}

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

/**
  * Spark提供了JdbcRDD，用于获取关系型数据库的数据，需要指定配置信息
  * 注意：只能获取数据
  */
object JdbcRDDDemo {
  def main(args: Array[String]): Unit = {

    val conf = new SparkConf()
    conf.setAppName("JdbcRDDDemo")
    conf.setMaster("local[2]")
    val sc = new SparkContext(conf)

    val jdbcUrl = "jdbc:mysql://192.168.88.83:3306/bigdata?
useUnicode=true&characterEncoding=utf8"
    val user = "root"
    val password = "root"

    val sql = "select id,location,counts,access_date from location_info where id >= ? and id <=
?"

    val conn = () => {
      Class.forName("com.mysql.jdbc.Driver").newInstance()
      DriverManager.getConnection(jdbcUrl,user, password)
    }

    val jdbcRDD: JdbcRDD[(Int, String, Int, Date)] = new JdbcRDD(
      sc, conn, sql, 0, 1000, 1,
      res => {
        val id = res.getInt("id")
        val location = res.getString("location")
        val counts = res.getInt("counts")
        val access_date = res.getDate("access_date")


        (id, location, counts, access_date)
```

```
      }
    )

    jdbcRDD.foreach(println)


    sc.stop()
  }
}
```

5 排序

6 案例练习参考代码

分析CDN日志统计出访问的PV、UV、IP地址

日志格式为:

IP 命中率(Hit/Miss) 响应时间 请求时间 请求方法 请求URL 请求协议 状态吗 响应大小 referer 用户代理

PV(page view)，即页面浏览量，或点击量;通常是衡量一个网络新闻频道或网站甚至一条网络新闻的主要指标

uv(unique visitor)，指访问某个站点或点击某条新闻的不同IP地址的人数。

统计需求:

1.计算独立IP数

```
1.计算思路:
1.1. 从每行日志中筛选出IP地址
1.2. 去除重复的IP得到独立IP数
2.计算过程
flatMap(x=>IPPattern findFirstIn(x)) 通过正则取出每行日志中的IP地址
map(x=>(x,1)) 将每行中的IP映射成 (IP,1)，形成一个Pair RDD
reduceByKey((x,y)=>x+y) 将相同的IP合并，得到 (IP，数量)
sortBy(_._2,false) 按IP大小排序
3.统计结果样例:
(114.55.227.102,9348)
(220.191.255.197,2640)
```

2.统计每个视频独立IP数

```
有时我们不但需要知道全网访问的独立IP数，更想知道每个视频访问的独立IP数
1.计算思路:
1.1筛选视频文件将每行日志拆分成 (文件名，IP地址)形式
1.2按文件名分组，相当于数据库的Group by 这时RDD的结构为(文件名,[IP1,IP1,IP2,…])，这时IP有重复
1.3将每个文件名中的IP地址去重，这时RDD的结果为(文件名,[IP1,IP2,…])，这时IP没有重复
2.计算过程:
filter(x=>x.matches(“.([0-9]+).mp4.“)) 筛选日志中的视频请求
map(x=>getFileNameAndIp(x)) 将每行日志格式化成 (文件名，IP)这种格式
groupByKey() 按文件名分组，这时RDD 结构为 (文件名，[IP1,IP1,IP2….])，IP有重复
```

map(x=>(x.1,x.2.toList.distinct)) 去除value中重复的IP地址
sortBy(.2.size,false) 按IP数排序
3.计算结果样例:
视频：141081.mp4 独立IP数:2393
视频：140995.mp4 独立IP数:2050

3.统计一天中每个小时的流量

有时我想知道网站每小时视频的观看流量，看看用户都喜欢在什么时间段过来看视频
1.计算思路:
1.1.将日志中的访问时间及请求大小两个数据提取出来形成 RDD（访问时间,访问大小），这里要去除404之类的非法请求
1.2.按访问时间分组形成 RDD （访问时间，[大小1，大小2，….]）
1.3.将访问时间对应的大小相加形成（访问时间，总大小）
2.计算过程:
filter(x=>isMatch(httpSizePattern,x)).filter(x=>isMatch(timePattern,x)) 过滤非法请求
map(x=>getTimeAndSize(x)) 将日志格式化成 RDD(请求小时,请求大小)
groupByKey() 按请求时间分组形成 RDD(请求小时，[大小1，大小2，….])
map(x=>(x._1,x._2.sum)) 将每小时的请求大小相加，形成 RDD(请求小时,总大小)
3.计算结果样例:
00时 CDN流量=14G
01时 CDN流量=3G
02时 CDN流量=5G

案例代码

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}
import org.slf4j.LoggerFactory

import scala.util.matching.Regex

/**
  * Created by wuyufei on 31/07/2017.
  */
object CdnStatics {

  val logger = LoggerFactory.getLogger(CdnStatics.getClass)

  //匹配IP地址
  val IPPattern = "((?:(?:25[0-5]|2[0-4]\\d|((1\\d{2})|([1-9]?\\d)))\\.){3}(?:25[0-5]|2[0-4]\\d|((1\\d{2})|([1-9]?\\d))))".r

  //匹配视频文件名
  val videoPattern = "([0-9]+).mp4".r

  //[15/Feb/2017:11:17:13 +0800]  匹配 2017:11 按每小时播放量统计
  val timePattern = ".*(2017):([0-9]{2}):[0-9]{2}:[0-9]{2}.*".r

  //匹配 http 响应码和请求数据大小

  val httpSizePattern = ".*\\s(200|206|304)\\s([0-9]+)\\s.*".r
```

```
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setMaster("local[*]").setAppName("CdnStatics")

    val sc = new SparkContext(conf)

    val input = sc.textFile("dir/cdn.txt").cache()

    //统计独立IP访问量前10位
    ipStatics(input)

    //统计每个视频独立IP数
    videoIpStatics(input)

    //统计一天中每个小时间的流量
    flowOfHour(input)

    sc.stop()
  }

  //统计一天中每个小时间的流量
  def flowOfHour(data: RDD[String]): Unit = {

    def isMatch(pattern: Regex, str: String) = {
      str match {
        case pattern(_*) => true
        case _ => false
      }
    }

    /**
      * 获取日志中小时和http 请求体大小
      *
      * @param line
      * @return
      */
    def getTimeAndSize(line: String) = {
      var res = ("", 0L)
      try {
        val httpSizePattern(code, size) = line
        val timePattern(year, hour) = line
        res = (hour, size.toLong)
      } catch {
        case ex: Exception => ex.printStackTrace()
      }
      res
    }

    //3.统计一天中每个小时间的流量

 data.filter(x=>isMatch(httpSizePattern,x)).filter(x=>isMatch(timePattern,x)).map(x=>getTimeAndS
ize(x)).groupByKey()

      .map(x=>(x._1,x._2.sum)).sortByKey().foreach(x=>println(x._1+"时 CDN流量
```

```
="+x._2/(1024*1024*1024)+"G"))
  }

  // 统计每个视频独立IP数
  def videoIpStatics(data: RDD[String]): Unit = {
    def getFileNameAndIp(line: String) = {
      (videoPattern.findFirstIn(line).mkString, IPPattern.findFirstIn(line).mkString)
    }
    //2.统计每个视频独立IP数
    data.filter(x => x.matches(".*([0-9]+)\\.mp4.*")).map(x =>
getFileNameAndIp(x)).groupByKey().map(x => (x._1, x._2.toList.distinct)).
      sortBy(_._2.size, false).take(10).foreach(x => println("视频：" + x._1 + " 独立IP数:" +
x._2.size))
  }


  // 统计独立IP访问量前10位
  def ipStatics(data: RDD[String]): Unit = {

    //1.统计独立IP数
    val ipNums = data.map(x => (IPPattern.findFirstIn(x).get, 1)).reduceByKey(_ +
_).sortBy(_._2, false)

    //输出IP访问数前量前10位
    ipNums.take(10).foreach(println)

    println("独立IP数：" + ipNums.count())
  }

}
```