```
val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))
val dataRDD2 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val rdd: RDD[(String, (Int, Option[Int]))] = dataRDD1.leftOuterJoin(dataRDD2)
```

26) cogroup

➢ 函数签名

def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]

➢ 函数说明

在类型为(K,V)和(K,W)的 RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1),("a",2),("c",3)))
val dataRDD2 = sparkContext.makeRDD(List(("a",1),("c",2),("c",3)))

val value: RDD[(String, (Iterable[Int], Iterable[Int]))] =

dataRDD1.cogroup(dataRDD2)
```

5.1.4.4 案例实操

1) 数据准备

agent.log：时间戳，省份，城市，用户，广告，中间字段使用空格分隔。

2) 需求描述

统计出每一个省份每个广告被点击数量排行的 Top3

3) 需求分析

4) 功能实现

5.1.4.5 RDD 行动算子

1) reduce

➢ 函数签名

def reduce(f: (T, T) => T): T

➢ 函数说明

聚集 RDD 中的所有元素，先聚合分区内数据，再聚合分区间数据

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 聚合数据
val reduceResult: Int = rdd.reduce(_+_)
```

2) collect

➢ 函数签名

def collect(): Array[T]

➢ 函数说明

在驱动程序中，以数组 Array 的形式返回数据集的所有元素

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 收集数据到 Driver
rdd.collect().foreach(println)
```

3) count

➢ 函数签名

def count(): Long

➢ 函数说明

返回 RDD 中元素的个数

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回 RDD 中元素的个数
val countResult: Long = rdd.count()
```

4) first

➢ 函数签名

def first(): T

➢ 函数说明

返回 RDD 中的第一个元素

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回 RDD 中元素的个数
val firstResult: Int = rdd.first()
println(firstResult)
```

5) take

➢ 函数签名

def take(num: Int): Array[T]

➢ 函数说明

返回一个由 RDD 的前 n 个元素组成的数组

```
vval rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回 RDD 中元素的个数
val takeResult: Array[Int] = rdd.take(2)
println(takeResult.mkString(","))
```

6) takeOrdered

➢ 函数签名

def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]

➢ 函数说明

返回该 RDD 排序后的前 n 个元素组成的数组

```
val rdd: RDD[Int] = sc.makeRDD(List(1,3,2,4))

// 返回 RDD 中元素的个数
val result: Array[Int] = rdd.takeOrdered(2)
```

7) aggregate

➢ 函数签名

def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U

➢ 函数说明

分区的数据通过初始值和分区内的数据进行聚合，然后再和初始值进行分区间的数据聚合

```
val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4), 8)

// 将该 RDD 所有元素相加得到结果
//val result: Int = rdd.aggregate(0)(_ + _, _ + _)
val result: Int = rdd.aggregate(10)(_ + _, _ + _)
```

8) fold

➢ 函数签名

```
def fold(zeroValue: T)(op: (T, T) => T): T
```

➢ 函数说明

折叠操作，aggregate 的简化版操作

```
val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4))

val foldResult: Int = rdd.fold(0)(_+_)
```

9) countByKey

➢ 函数签名

```
def countByKey(): Map[K, Long]
```

➢ 函数说明

统计每种 key 的个数

```
val rdd: RDD[(Int, String)] = sc.makeRDD(List((1, "a"), (1, "a"), (1, "a"), (2,
"b"), (3, "c"), (3, "c")))

// 统计每种 key 的个数
val result: collection.Map[Int, Long] = rdd.countByKey()
```

10) save 相关算子

➢ 函数签名

```
def saveAsTextFile(path: String): Unit
def saveAsObjectFile(path: String): Unit
def saveAsSequenceFile(
  path: String,
  codec: Option[Class[_ <: CompressionCodec]] = None): Unit
```

➢ 函数说明

将数据保存到不同格式的文件中

```
// 保存成 Text 文件
rdd.saveAsTextFile("output")

// 序列化成对象保存到文件
rdd.saveAsObjectFile("output1")

// 保存成 Sequencefile 文件
rdd.map((_,1)).saveAsSequenceFile("output2")
```

11) foreach

- 函数签名

```
def foreach(f: T => Unit): Unit = withScope {
    val cleanF = sc.clean(f)
    sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))
}
```

- 函数说明

分布式遍历 RDD 中的每一个元素，调用指定函数

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 收集后打印
rdd.map(num=>num).collect().foreach(println)

println("****************")

// 分布式打印
rdd.foreach(println)
```

5.1.4.6 RDD 序列化

1) 闭包检查

从计算的角度，算子以外的代码都是在 Driver 端执行，算子里面的代码都是在 Executor 端执行。那么在 scala 的函数式编程中，就会导致算子内经常会用到算子外的数据，这样就形成了闭包的效果，如果使用的算子外的数据无法序列化，就意味着无法传值给 Executor 端执行，就会发生错误，所以需要在执行任务计算前，检测闭包内的对象是否可以进行序列化，这个操作我们称之为闭包检测。Scala2.12 版本后闭包编译方式发生了改变

2) 序列化方法和属性

从计算的角度，算子以外的代码都是在 Driver 端执行，算子里面的代码都是在 Executor 端执行，看如下代码：

```
object serializable02_function {

    def main(args: Array[String]): Unit = {
        //1.创建 SparkConf 并设置 App 名称
        val conf: SparkConf = new
SparkConf().setAppName("SparkCoreTest").setMaster("local[*]")

        //2.创建 SparkContext，该对象是提交 Spark App 的入口
        val sc: SparkContext = new SparkContext(conf)

        //3.创建一个 RDD
        val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello spark",
"hive", "atguigu"))

        //3.1 创建一个 Search 对象
        val search = new Search("hello")

        //3.2 函数传递，打印：ERROR Task not serializable
        search.getMatch1(rdd).collect().foreach(println)

        //3.3 属性传递，打印：ERROR Task not serializable
        search.getMatch2(rdd).collect().foreach(println)

        //4.关闭连接
        sc.stop()
    }
}

class Search(query:String) extends Serializable {

    def isMatch(s: String): Boolean = {
        s.contains(query)
    }

    // 函数序列化案例
    def getMatch1 (rdd: RDD[String]): RDD[String] = {
        //rdd.filter(this.isMatch)
        rdd.filter(isMatch)
    }
```

```
    // 属性序列化案例
    def getMatch2(rdd: RDD[String]): RDD[String] = {
        //rdd.filter(x => x.contains(this.query))
        rdd.filter(x => x.contains(query))
        //val q = query
        //rdd.filter(x => x.contains(q))
    }
}
```

3) Kryo 序列化框架

参考地址: https://github.com/EsotericSoftware/kryo

Java 的序列化能够序列化任何的类。但是比较重（字节多），序列化后，对象的提交也比较大。Spark 出于性能的考虑，Spark2.0 开始支持另外一种 Kryo 序列化机制。Kryo 速度是 Serializable 的 10 倍。当 RDD 在 Shuffle 数据的时候，简单数据类型、数组和字符串类型已经在 Spark 内部使用 Kryo 来序列化。

注意：即使使用 Kryo 序列化，也要继承 Serializable 接口。

```
object serializable_Kryo {

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

        val conf: SparkConf = new SparkConf()
                .setAppName("SerDemo")
                .setMaster("local[*]")
                // 替换默认的序列化机制
                .set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
                // 注册需要使用 kryo 序列化的自定义类
                .registerKryoClasses(Array(classOf[Searcher]))

        val sc = new SparkContext(conf)

        val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello atguigu",
"atguigu", "hahah"), 2)

        val searcher = new Searcher("hello")
        val result: RDD[String] = searcher.getMatchedRDD1(rdd)

        result.collect.foreach(println)
    }
}
case class Searcher(val query: String) {

    def isMatch(s: String) = {
        s.contains(query)
    }

    def getMatchedRDD1(rdd: RDD[String]) = {
        rdd.filter(isMatch)
    }

    def getMatchedRDD2(rdd: RDD[String]) = {
        val q = query
        rdd.filter(_.contains(q))
    }
}
```

5.1.4.7 RDD 依赖关系

1) RDD 血缘关系

RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列 Lineage（血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

```
val fileRDD: RDD[String] = sc.textFile("input/1.txt")
println(fileRDD.toDebugString)
println("----------------------")

val wordRDD: RDD[String] = fileRDD.flatMap(_.split(" "))
println(wordRDD.toDebugString)
println("----------------------")

val mapRDD: RDD[(String, Int)] = wordRDD.map((_,1))
println(mapRDD.toDebugString)
println("----------------------")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(_+_)
println(resultRDD.toDebugString)

resultRDD.collect()
```

2) RDD 依赖关系

这里所谓的依赖关系，其实就是两个相邻 RDD 之间的关系

```
val sc: SparkContext = new SparkContext(conf)

val fileRDD: RDD[String] = sc.textFile("input/1.txt")
println(fileRDD.dependencies)
println("----------------------")

val wordRDD: RDD[String] = fileRDD.flatMap(_.split(" "))
println(wordRDD.dependencies)
println("----------------------")

val mapRDD: RDD[(String, Int)] = wordRDD.map((_,1))
println(mapRDD.dependencies)
println("----------------------")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(_+_)
println(resultRDD.dependencies)

resultRDD.collect()
```

3) RDD 窄依赖

窄依赖表示每一个父(上游)RDD 的 Partition 最多被子（下游）RDD 的一个 Partition 使用，窄依赖我们形象的比喻为独生子女。

```
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd)
```

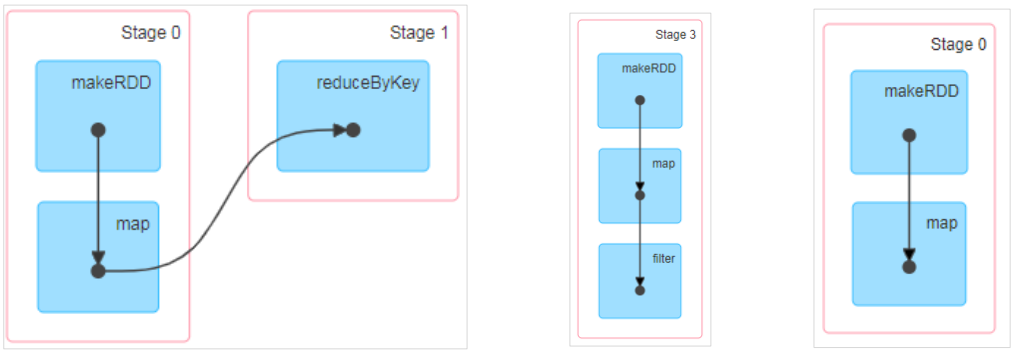
4) RDD 宽依赖

宽依赖表示同一个父（上游）RDD 的 Partition 被多个子（下游）RDD 的 Partition 依赖，会引起 Shuffle，总结：宽依赖我们形象的比喻为多生。

```
class ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag](
    @transient private val _rdd: RDD[_ <: Product2[K, V]],
    val partitioner: Partitioner,
    val serializer: Serializer = SparkEnv.get.serializer,
    val keyOrdering: Option[Ordering[K]] = None,
    val aggregator: Option[Aggregator[K, V, C]] = None,
    val mapSideCombine: Boolean = false)
  extends Dependency[Product2[K, V]]
```

5) RDD 阶段划分

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。例如，DAG 记录了 RDD 的转换过程和任务的阶段。



6) RDD 阶段划分源码

```
try {
  // New stage creation may throw an exception if, for example, jobs are run on
a
  // HadoopRDD whose underlying HDFS files have been deleted.
  finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)
} catch {
  case e: Exception =>
    logWarning("Creating new stage failed due to exception - job: " + jobId, e)
    listener.jobFailed(e)
    return
}

……

private def createResultStage(
  rdd: RDD[_],
  func: (TaskContext, Iterator[_]) => _,
  partitions: Array[Int],
  jobId: Int,
  callSite: CallSite): ResultStage = {
val parents = getOrCreateParentStages(rdd, jobId)
val id = nextStageId.getAndIncrement()
val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)
stageIdToStage(id) = stage
updateJobIdStageIdMaps(jobId, stage)
stage
}
```

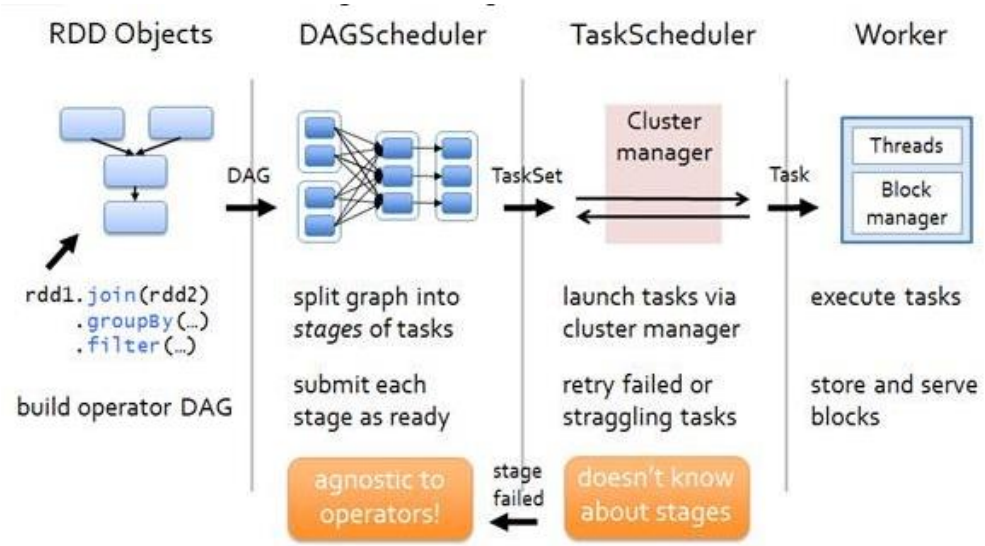
```
……

private def getOrCreateParentStages(rdd: RDD[_], firstJobId: Int): List[Stage]
= {
getShuffleDependencies(rdd).map { shuffleDep =>
  getOrCreateShuffleMapStage(shuffleDep, firstJobId)
}.toList
}

……

private[scheduler] def getShuffleDependencies(
  rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]] = {
val parents = new HashSet[ShuffleDependency[_, _, _]]
val visited = new HashSet[RDD[_]]
val waitingForVisit = new Stack[RDD[_]]
waitingForVisit.push(rdd)
while (waitingForVisit.nonEmpty) {
  val toVisit = waitingForVisit.pop()
  if (!visited(toVisit)) {
    visited += toVisit
    toVisit.dependencies.foreach {
      case shuffleDep: ShuffleDependency[_, _, _] =>
        parents += shuffleDep
      case dependency =>
        waitingForVisit.push(dependency.rdd)
    }
  }
}
parents
}
```

7) RDD 任务划分

RDD 任务切分中间分为：Application、Job、Stage 和 Task

- Application：初始化一个 SparkContext 即生成一个 Application；
- Job：一个 Action 算子就会生成一个 Job；
- Stage：Stage 等于宽依赖(ShuffleDependency)的个数加 1；
- Task：一个 Stage 阶段中，最后一个 RDD 的分区个数就是 Task 的个数。

注意：Application->Job->Stage->Task 每一层都是 1 对 n 的关系。



8) RDD 任务划分源码

```
val tasks: Seq[Task[_]] = try {
  stage match {
    case stage: ShuffleMapStage =>
      partitionsToCompute.map { id =>
        val locs = taskIdToLocations(id)
        val part = stage.rdd.partitions(id)
        new ShuffleMapTask(stage.id, stage.latestInfo.attemptId,
          taskBinary,  part,  locs,  stage.latestInfo.taskMetrics,  properties,
Option(jobId),
          Option(sc.applicationId), sc.applicationAttemptId)
      }

    case stage: ResultStage =>
      partitionsToCompute.map { id =>
        val p: Int = stage.partitions(id)
        val part = stage.rdd.partitions(p)
        val locs = taskIdToLocations(id)
        new ResultTask(stage.id, stage.latestInfo.attemptId,
          taskBinary, part, locs, id, properties, stage.latestInfo.taskMetrics,
          Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)
      }
  }

……

val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()

……

override def findMissingPartitions(): Seq[Int] = {
mapOutputTrackerMaster
  .findMissingPartitions(shuffleDep.shuffleId)
  .getOrElse(0 until numPartitions)
}
```

5.1.4.8 RDD 持久化

1) RDD Cache 缓存

RDD 通过 Cache 或者 Persist 方法将前面的计算结果缓存，默认情况下会把数据以缓存在 JVM 的堆内存中。但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 算子时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。

```
// cache 操作会增加血缘关系，不改变原有的血缘关系
println(wordToOneRdd.toDebugString)

// 数据缓存。
wordToOneRdd.cache()

// 可以更改存储级别
//mapRdd.persist(StorageLevel.MEMORY_AND_DISK_2)
```

存储级别

```
object StorageLevel {
  val NONE = new StorageLevel(false, false, false, false)
  val DISK_ONLY = new StorageLevel(true, false, false, false)
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)
  val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
  val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
  val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
  val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
  val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
  val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
  val OFF_HEAP = new StorageLevel(true, true, true, false, 1)
```

级　别	使用的空间	CPU 时间	是否在内存中	是否在磁盘上	备　注
MEMORY_ONLY	高	低	是	否	
MEMORY_ONLY_SER	低	高	是	否	
MEMORY_AND_DISK	高	中等	部分	部分	如果数据在内存中放不下，则溢写到磁盘上
MEMORY_AND_DISK_SER	低	高	部分	部分	如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据
DISK_ONLY	低	高	否	是	

缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除，RDD 的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于 RDD 的一系列转换，丢失的数据会被重算，由于 RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

Spark 会自动对一些 Shuffle 操作的中间数据做持久化操作(比如：reduceByKey)。这样做的目的是为了当一个节点 Shuffle 失败了避免重新计算整个输入。但是，在实际使用的时候，如果想重用数据，仍然建议调用 persist 或 cache。

2) RDD CheckPoint 检查点

所谓的检查点其实就是通过将 RDD 中间结果写入磁盘

由于血缘依赖过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果检查点之后有节点出现问题，可以从检查点开始重做血缘，减少了开销。

对 RDD 进行 checkpoint 操作并不会马上被执行，必须执行 Action 操作才能触发。

```
// 设置检查点路径
sc.setCheckpointDir("./checkpoint1")

// 创建一个 RDD，读取指定位置文件:hello atguigu atguigu
val lineRdd: RDD[String] = sc.textFile("input/1.txt")

// 业务逻辑
val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))

val wordToOneRdd: RDD[(String, Long)] = wordRdd.map {
    word => {
        (word, System.currentTimeMillis())
    }
}

// 增加缓存,避免再重新跑一个 job 做 checkpoint
wordToOneRdd.cache()
// 数据检查点：针对 wordToOneRdd 做检查点计算
wordToOneRdd.checkpoint()

// 触发执行逻辑
wordToOneRdd.collect().foreach(println)
```

3) 缓存和检查点区别

1）Cache 缓存只是将数据保存起来，不切断血缘依赖。Checkpoint 检查点切断血缘依赖。

2）Cache 缓存的数据通常存储在磁盘、内存等地方，可靠性低。Checkpoint 的数据通常存储在 HDFS 等容错、高可用的文件系统，可靠性高。

3）建议对 checkpoint()的 RDD 使用 Cache 缓存，这样 checkpoint 的 job 只需从 Cache 缓存中读取数据即可，否则需要再从头计算一次 RDD。

5.1.4.9 RDD 分区器

Spark 目前支持 Hash 分区和 Range 分区，和用户自定义分区。Hash 分区为当前的默认分区。分区器直接决定了 RDD 中分区的个数、RDD 中每条数据经过 Shuffle 后进入哪个分区，进而决定了 Reduce 的个数。

- 只有 Key-Value 类型的 RDD 才有分区器，非 Key-Value 类型的 RDD 分区的值是 None
- 每个 RDD 的分区 ID 范围：0 ~ (numPartitions - 1)，决定这个值是属于那个分区的。

1) **Hash 分区**：对于给定的 key，计算其 hashCode,并除以分区个数取余

```
class HashPartitioner(partitions: Int) extends Partitioner {
  require(partitions >= 0, s"Number of partitions ($partitions) cannot be
negative.")

  def numPartitions: Int = partitions

  def getPartition(key: Any): Int = key match {
    case null => 0
    case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
  }

  override def equals(other: Any): Boolean = other match {
    case h: HashPartitioner =>
      h.numPartitions == numPartitions
    case _ =>
      false
  }

  override def hashCode: Int = numPartitions
}
```

2) **Range 分区**：将一定范围内的数据映射到一个分区中，尽量保证每个分区数据均匀，而且分区间有序

```
class RangePartitioner[K : Ordering : ClassTag, V](
    partitions: Int,
    rdd: RDD[_ <: Product2[K, V]],
    private var ascending: Boolean = true)
  extends Partitioner {

  // We allow partitions = 0, which happens when sorting an empty RDD under the
default settings.
  require(partitions >= 0, s"Number of partitions cannot be negative but found
$partitions.")

  private var ordering = implicitly[Ordering[K]]

  // An array of upper bounds for the first (partitions - 1) partitions
  private var rangeBounds: Array[K] = {
  ...
  }

  def numPartitions: Int = rangeBounds.length + 1

  private var binarySearch: ((Array[K], K) => Int) =
CollectionsUtils.makeBinarySearch[K]
```

```
  def getPartition(key: Any): Int = {
    val k = key.asInstanceOf[K]
    var partition = 0
    if (rangeBounds.length <= 128) {
      // If we have less than 128 partitions naive search
      while (partition < rangeBounds.length && ordering.gt(k,
rangeBounds(partition))) {
        partition += 1
      }
    } else {
      // Determine which binary search method to use only once.
      partition = binarySearch(rangeBounds, k)
      // binarySearch either returns the match location or -[insertion point]-1
      if (partition < 0) {
        partition = -partition-1
      }
      if (partition > rangeBounds.length) {
        partition = rangeBounds.length
      }
    }
    if (ascending) {
      partition
    } else {
      rangeBounds.length - partition
    }
  }

  override def equals(other: Any): Boolean = other match {
  ...
  }

  override def hashCode(): Int = {
  ...
  }

  @throws(classOf[IOException])
  private def writeObject(out: ObjectOutputStream): Unit =
Utils.tryOrIOException {
  ...
  }

  @throws(classOf[IOException])
  private def readObject(in: ObjectInputStream): Unit = Utils.tryOrIOException
{
  ...
  }
}
```

5.1.4.10 RDD 文件读取与保存

Spark 的数据读取及数据保存可以从两个维度来作区分：文件格式以及文件系统。

文件格式分为：text 文件、csv 文件、sequence 文件以及 Object 文件；

文件系统分为：本地文件系统、HDFS、HBASE 以及数据库。

- **text 文件**

```
// 读取输入文件
val inputRDD: RDD[String] = sc.textFile("input/1.txt")

// 保存数据
inputRDD.saveAsTextFile("output")
```

- **sequence 文件**

SequenceFile 文件是 Hadoop 用来存储二进制形式的 key-value 对而设计的一种平面文件(Flat File)。在 SparkContext 中，可以调用 sequenceFile[keyClass, valueClass](path)。

```
// 保存数据为 SequenceFile
dataRDD.saveAsSequenceFile("output")

// 读取 SequenceFile 文件
sc.sequenceFile[Int,Int]("output").collect().foreach(println)
```

- **object 对象文件**

对象文件是将对象序列化后保存的文件，采用 Java 的序列化机制。可以通过 objectFile[T: ClassTag](path)函数接收一个路径，读取对象文件，返回对应的 RDD，也可以通过调用 saveAsObjectFile()实现对对象文件的输出。因为是序列化所以要指定类型。

```
// 保存数据
dataRDD.saveAsObjectFile("output")

// 读取数据
sc.objectFile[Int]("output").collect().foreach(println)
```

5.2 累加器

5.2.1 实现原理

累加器用来把 Executor 端变量信息聚合到 Driver 端。在 Driver 程序中定义的变量，在 Executor 端的每个 Task 都会得到这个变量的一份新的副本，每个 task 更新这些副本的值后，传回 Driver 端进行 merge。

5.2.2 基础编程

5.2.2.1 系统累加器

```
val rdd = sc.makeRDD(List(1,2,3,4,5))
// 声明累加器
var sum = sc.longAccumulator("sum");
rdd.foreach(
  num => {
    // 使用累加器
    sum.add(num)
  }
)
// 获取累加器的值
println("sum = " + sum.value)
```

5.2.2.2 自定义累加器

```
// 自定义累加器
// 1. 继承 AccumulatorV2，并设定泛型
// 2. 重写累加器的抽象方法
class WordCountAccumulator extends AccumulatorV2[String, mutable.Map[String,
Long]]{

var map : mutable.Map[String, Long] = mutable.Map()

// 累加器是否为初始状态
override def isZero: Boolean = {
  map.isEmpty
}

// 复制累加器
override def copy(): AccumulatorV2[String, mutable.Map[String, Long]] = {
  new WordCountAccumulator
}

// 重置累加器
override def reset(): Unit = {
  map.clear()
}

// 向累加器中增加数据 (In)
override def add(word: String): Unit = {
    // 查询 map 中是否存在相同的单词
    // 如果有相同的单词，那么单词的数量加 1
    // 如果没有相同的单词，那么在 map 中增加这个单词
    map(word) = map.getOrElse(word, 0L) + 1L
}
```

```
// 合并累加器
override def merge(other: AccumulatorV2[String, mutable.Map[String, Long]]):
Unit = {

  val map1 = map
  val map2 = other.value

  // 两个 Map 的合并
  map = map1.foldLeft(map2)(
    ( innerMap, kv ) => {
      innerMap(kv._1) = innerMap.getOrElse(kv._1, 0L) + kv._2
      innerMap
    }
  )
}

// 返回累加器的结果 （Out）
override def value: mutable.Map[String, Long] = map
}
```

5.3 广播变量

5.3.1 实现原理

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，广播变量用起来都很顺手。在多个并行操作中使用同一个变量，但是 Spark 会为每个任务分别发送。

5.3.2 基础编程

```
val rdd1 = sc.makeRDD(List( ("a",1), ("b", 2), ("c", 3), ("d", 4) ),4)
val list = List( ("a",4), ("b", 5), ("c", 6), ("d", 7) )
// 声明广播变量
val broadcast: Broadcast[List[(String, Int)]] = sc.broadcast(list)

val resultRDD: RDD[(String, (Int, Int))] = rdd1.map {
  case (key, num) => {
    var num2 = 0
    // 使用广播变量
    for ((k, v) <- broadcast.value) {
      if (k == key) {
        num2 = v
      }
    }
    (key, (num, num2))
  }
}
```

第6章　Spark 案例实操

在之前的学习中，我们已经学习了 Spark 的基础编程方式，接下来，我们看看在实际的工作中如何使用这些 API 实现具体的需求。这些需求是电商网站的真实需求，所以在实现功能前，咱们必须先将数据准备好。

上面的数据图是从数据文件中截取的一部分内容，表示为电商网站的用户行为数据，主要包含用户的 4 种行为：搜索，点击，下单，支付。数据规则如下：

- 数据文件中每行数据采用下划线分隔数据
- 每一行数据表示用户的一次行为，这个行为只能是 4 种行为的一种
- 如果搜索关键字为 null,表示数据不是搜索数据
- 如果点击的品类 ID 和产品 ID 为-1，表示数据不是点击数据
- 针对于下单行为，一次可以下单多个商品，所以品类 ID 和产品 ID 可以是多个，id 之间采用逗号分隔，如果本次不是下单行为，则数据采用 null 表示
- 支付行为和下单行为类似

详细字段说明：

编号	字段名称	字段类型	字段含义
1	date	String	用户点击行为的日期
2	user_id	Long	用户的 ID
3	session_id	String	Session 的 ID
4	page_id	Long	某个页面的 ID
5	action_time	String	动作的时间点
6	search_keyword	String	用户搜索的关键词

7	click_category_id	Long	某一个商品品类的 ID
8	click_product_id	Long	某一个商品的 ID
9	order_category_ids	String	一次订单中所有品类的 ID 集合
10	order_product_ids	String	一次订单中所有商品的 ID 集合
11	pay_category_ids	String	一次支付中所有品类的 ID 集合
12	pay_product_ids	String	一次支付中所有商品的 ID 集合
13	city_id	Long	城市 id

样例类：

```
//用户访问动作表
case class UserVisitAction(
    date: String,//用户点击行为的日期
    user_id: Long,//用户的 ID
    session_id: String,//Session 的 ID
    page_id: Long,//某个页面的 ID
    action_time: String,//动作的时间点
    search_keyword: String,//用户搜索的关键词
    click_category_id: Long,//某一个商品品类的 ID
    click_product_id: Long,//某一个商品的 ID
    order_category_ids: String,//一次订单中所有品类的 ID 集合
    order_product_ids: String,//一次订单中所有商品的 ID 集合
    pay_category_ids: String,//一次支付中所有品类的 ID 集合
    pay_product_ids: String,//一次支付中所有商品的 ID 集合
    city_id: Long
)//城市 id
```

6.1 需求 1：Top10 热门品类

6.1.1 需求说明

品类是指产品的分类，大型电商网站品类分多级，咱们的项目中品类只有一级，不同的公司可能对热门的定义不一样。我们按照每个品类的点击、下单、支付的量来统计热门品类。

鞋　　点击数 下单数　支付数

衣服　　点击数 下单数　支付数

电脑　　点击数 下单数　支付数

例如，综合排名 = 点击数*20%+下单数*30%+支付数*50%

本项目需求优化为：先按照点击数排名，靠前的就排名高；如果点击数相同，再比较下单数；下单数再相同，就比较支付数。

6.1.2 实现方案一

6.1.2.1 需求分析

分别统计每个品类点击的次数，下单的次数和支付的次数：

（品类，点击总数）（品类，下单总数）（品类，支付总数）

6.1.2.2 需求实现

6.1.3 实现方案二

6.1.3.1 需求分析

一次性统计每个品类点击的次数，下单的次数和支付的次数：

（品类，（点击总数，下单总数，支付总数））

6.1.3.2 需求实现

6.1.4 实现方案三

6.1.4.1 需求分析

使用累加器的方式聚合数据

6.1.4.2 需求实现

6.2 需求 2：Top10 热门品类中每个品类的 Top10 活跃 Session 统计

6.2.1 需求说明

在需求一的基础上，增加每个品类用户 session 的点击统计

6.2.2 需求分析

6.2.3 功能实现

6.3 需求 3：页面单跳转换率统计

6.3.1 需求说明

1）页面单跳转化率

计算页面单跳转化率，什么是页面单跳转换率，比如一个用户在一次 Session 过程中访问的页面路径 3,5,7,9,10,21，那么页面 3 跳到页面 5 叫一次单跳，7-9 也叫一次单跳，那么单跳转化率就是要统计页面点击的概率。

比如：计算 3-5 的单跳转化率，先获取符合条件的 Session 对于页面 3 的访问次数（PV）为 A，然后获取符合条件的 Session 中访问了页面 3 又紧接着访问了页面 5 的次数为 B，那么 B/A 就是 3-5 的页面单跳转化率。

2）统计页面单跳转化率意义

产品经理和运营总监，可以根据这个指标，去尝试分析，整个网站，产品，各个页面的表现怎么样，是不是需要去优化产品的布局；吸引用户最终可以进入最后的支付页面。

数据分析师，可以此数据做更深一步的计算和分析。

企业管理层，可以看到整个公司的网站，各个页面的之间的跳转的表现如何，可以适当调整公司的经营战略或策略。

6.3.2 需求分析

6.3.3 功能实现