

Spark

Spark 于 2009 年诞生于加州大学伯克利分校 AMPLab，2013 年被捐赠给 Apache 软件基金会，2014 年 2 月成为 Apache 的顶级项目。相对于 MapReduce 的批处理计算，Spark 可以带来上百倍的性能提升，因此它成为继 MapReduce 之后，最为广泛使用的分布式计算框架。

第一章 SparkCore

1 RDD简介

RDD full name is Resilient Distributed Datasets, is the most basic data processing model of Spark, based on memory calculation, it is a read-only, partitioned record collection, supports parallel operations, can be converted from external datasets or other RDDs, it has the following characteristics:

- 一个 RDD 由一个或者多个分区 (Partitions) 组成。对于 RDD 来说，每个分区会被一个计算任务所处理，用户可以在创建 RDD 时指定其分区个数，如果没有指定，则默认采用程序所分配到的 CPU 的核心数；
- RDD 拥有一个用于计算分区的函数 compute；
- RDD 会保存彼此间的依赖关系，RDD 的每次转换都会生成一个新的依赖关系，这种 RDD 之间的依赖关系就像流水线一样。在部分分区数据丢失后，可以通过这种依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算；
- Key-Value 型的 RDD 还拥有 Partitioner(分区器)，用于决定数据被存储在哪个分区中，目前 Spark 中支持 HashPartitioner(按照哈希分区) 和 RangePartitioner(按照范围进行分区)；
- 一个优先位置列表 (可选)，用于存储每个分区的优先位置 (preferred location)。对于一个 HDFS 文件来说，这个列表保存的就是每个分区所在的块的位置，按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能的将计算任务分配到其所要处理数据块的存储位置。

RDD[T] 抽象类的部分相关代码如下：

```
1 // 由子类实现以计算给定分区
2 def compute(split: Partition, context: TaskContext): Iterator[T]
3
4 // 获取所有分区
5 protected def getPartitions: Array[Partition]
6
7 // 获取所有依赖关系
8 protected def getDependencies: Seq[Dependency[_]] = deps
9
10 // 获取优先位置列表
11 protected def getPreferredLocations(split: Partition): Seq[String] = Nil
12
13 // 分区器 由子类重写以指定它们的分区方式
14 @transient val partitioner: Option[Partitioner] = None
```

2 创建RDD

```
1 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("RDD")
2 val sparkContext = new SparkContext(sparkConf)
3
4 // 从内存中创建RDD，将内存中集合的数据处理的数据源
5 val rdd1 = sparkContext.parallelize(
```

```

6     List(1, 2, 3, 4)
7   )
8
9   val rdd2 = sparkContext.makeRDD(
10     List(1, 2, 3, 4)
11   )
12
13   rdd1.collect().foreach(println)
14   rdd2.collect().foreach(println)
15
16   sparkContext.stop()

```

```

1   val sparkConf = new SparkConf().setMaster("local[*]").setAppName("RDD")
2   val sparkContext = new SparkContext(sparkConf)
3
4   // 从文件中创建RDD
5   val rdd = sparkContext.textFile("data/1.txt")
6
7   rdd.collect().foreach(println)
8
9   sparkContext.stop()

```

```

1   val sparkConf = new SparkConf().setMaster("local[*]").setAppName("RDD")
2   val sparkContext = new SparkContext(sparkConf)
3
4   // RDD的并行度与分区
5   // makeRDD第二个参数表示分区数量
6   val rdd = sparkContext.makeRDD(
7     List(1, 2, 3, 4), 2
8   )
9
10  // 将处理的数据保存成分区文件
11  rdd.saveAsTextFile("output")
12
13  sparkContext.stop()

```

3 SparkCore转换算子

转换算子：功能的补充和封装（把旧的RDD转换成新的RDD的操作）

coalesce

```

1   // coalesce是将RDD中的分区数量减少到numpartition
2   // 大数据集过滤后，可以将过滤的数据弄到一个分区，减少资源调度
3
4   // coalesce方法默认情况下不会将分区的数据打乱重新组合
5   // 这种情况下的缩减分区可能会导致数据不均衡，出现数据倾斜
6   // 如果想要让数据均衡，可以进行shuffle处理
7   // coalesce第二个参数shuffle: true和false
8
9   val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
10  val sc = new SparkContext(sparkConf)
11  val rdd = sc.makeRDD (
12    List(1, 2, 3, 4)
13  )
14  println("当前分区数: "+rdd.partitions.size)
15  println("=====")

```

```

16     val rdd2 = rdd.coalesce(1)
17     println("合并分区后, 现在的分区数: "+rdd2.partitions.size)
18
19     sc.stop()
20     // 返回结果:
21     // 当前分区数: 8
22     // =====
23     // 合并分区后, 现在的分区数: 1

```

distinct

```

1     // distinct 去重
2     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
3     val sc = new SparkContext(sparkConf)
4     val rdd = sc.makeRDD(
5         List(1, 2, 2, 4, 4, 5, 5, 3, 9, 10)
6     )
7     rdd.distinct().collect().foreach(println)
8     // 返回结果: 1 9 10 2 3 4 5
9     sc.stop()

```

filter

```

1     // filter 过滤掉不符合规则的元素
2     // 当数据进行筛选过滤后, 分区不变, 但是分区内的数据可能不均衡, 生产环境下可能会出现数据倾斜。
3     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
4     val sc = new SparkContext(sparkConf)
5     val rdd = sc.makeRDD(
6         List(1, 2, 3, 4)
7     )
8     rdd.filter(
9         _ != 3
10    ).collect().foreach(println)
11    // 返回结果: 1 2 4
12    sc.stop()

```

```

1     // 从服务器日志数据apache.log中获取2015年5月17日的请求路径
2     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
3     val sc = new SparkContext(sparkConf)
4     val rdd = sc.textFile("apache.log")
5
6     rdd.filter(
7         line => {
8             val datas = line.split(" ")
9             val time = datas(3)
10            time.startsWith("17/05/2015")
11        }
12    ).collect().foreach(println)
13
14    sc.stop()

```

flatMap

```
1 // 扁平映射
2 // flatMap会先执行map的操作，再将所有对象合并为一个对象，返回值是一个Sequence
3 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("flatMap")
4 val sc = new SparkContext(sparkConf)
5 val rdd = sc.makeRDD(
6     List(1, 2, 3, 4)
7 )
8 rdd.flatMap(data => data).collect().foreach(println)
9 // 返回结果: 1 2 3 4
10 sc.stop()
```

```
1 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("flatMap")
2 val sc = new SparkContext(sparkConf)
3 val rdd = sc.makeRDD(List(
4     List(1, 2), List(3, 4)
5 ))
6
7 val flatRDD = rdd.flatMap(
8     list => {
9         list
10    }
11 )
12
13 flatRDD.collect().foreach(println)
14
15 sc.stop()
```

```
1 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("flatMap")
2 val sc = new SparkContext(sparkConf)
3 val rdd = sc.makeRDD(List(
4     "Hello Scala", "Hello Spark"
5 ))
6
7 val flatRDD = rdd.flatMap(
8     s => {
9         s.split(" ")
10    }
11 )
12
13 flatRDD.collect().foreach(println)
14
15 sc.stop()
```

```
1 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("flatMap")
2 val sc = new SparkContext(sparkConf)
3 val rdd = sc.makeRDD(List(List(1, 2), 3, List(4, 5)))
4
5 val flatRDD = rdd.flatMap(
6     data => {
7         data match {
8             case list: List[_] => list
9             case dat => List(dat)
10        }
11    }
12 )
```

```
13
14 flatRDD.collect().foreach(println)
15
16 sc.stop()
```

map

```
1 // map是对数据逐条进行操作（映射转换），这里的转换可以是类型的转换，也可以是值的转换。
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("map")
3 val sc = new SparkContext(sparkConf)
4
5 val rdd = sc.makeRDD(
6     List(1, 2, 3, 4)
7 )
8
9 val mapRDD = rdd.map(
10     _ + 1
11 )
12
13 mapRDD.collect().foreach(println)
14 // 返回结果: 2 3 4 5
15
16 sc.stop()
```

```
1 // 小功能：从服务器日志数据apache.log中获取用户请求URL资源路径
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("map")
3 val sc = new SparkContext(sparkConf)
4
5 val rdd = sc.textFile("apache.log")
6
7 val mapRDD = rdd.map(
8     line => {
9         val datas = line.split(" ")
10         datas(6)
11     }
12 )
13
14 mapRDD.collect().foreach(println)
15
16 sc.stop()
```

```
1 // 并行计算演示
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("map")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD(List(1, 2, 3, 4), 1)
5 val mapRDD = rdd.map(
6     num => {
7         println(">>>>>>" + num)
8         num
9     }
10 )
11
12 val mapRDD1 = mapRDD.map(
13     num => {
14         println("#####" + num)
15         num
16     }
17 )
```

```

17     )
18
19     mapRDD1.collect()
20
21     sc.stop()

```

map与flatMap

```

1 // map操作后会返回到原来的集合中
2 // flatmap操作后会返回一个新的集合
3 // map与flatMap一般一起使用**
4 // 合并使用：
5 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("flatMap")
6 val sc = new SparkContext(sparkConf)
7 val rdd = sc.makeRDD(
8     List(List(1, 2), List(3, 4)), 1
9 )
10 rdd.flatMap(data => {
11     data.map(_*3)
12 }).collect().foreach(println)
13
14 sc.stop()
15 // 返回结果3 6 9 12
16 // 先进行map操作，将数据逐条*3，然后放回到data中
17 // flatmap再对data进行扁平映射，将list里面的数据逐条拿出来

```

glom

```

1 // glom将同一个分区中的所有元素合并成一个数组并返回
2 // 分区1:1,2 glom => List(1,2)
3 // 分区2:3,4 glom => List(3,4)
4 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
5 val sc = new SparkContext(sparkConf)
6 val rdd = sc.makeRDD (
7     List(1, 2, 3, 4), 2
8 )
9 rdd.glom().collect().foreach(println)
10 println("=====")
11
12 //分区求和
13 rdd.glom().map(_._sum).collect().foreach(println)
14
15 sc.stop()

```

```

1 // 分区内取最大值，分区间最大值求和
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(1, 2, 3, 4), 2
6 )
7
8 val glomRDD = rdd.glom()
9
10 val maxRDD = glomRDD.map(
11     array => {
12         array.max
13     }
14 )

```

```

14 )
15
16 println(maxRDD.collect().sum)
17
18 sc.stop()

```

groupby

```

1 // 进行分组，而分区不变,数据被重新打乱，这个过程叫做shuffle
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(1, 2, 3, 4),2
6 )
7 rdd.groupBy(_%2==0).collect().foreach(println)
8 // 返回结果：
9 // (false, CompactBuffer(1, 3))
10 // (true, CompactBuffer(2, 4))
11
12 sc.stop()

```

```

1 // 从服务器日志数据apache.log中获取每个时间段访问量
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
3 val sc = new SparkContext(sparkConf)
4
5 val rdd = sc.textFile("apache.log")
6
7 val timeRDD = rdd.map(
8     line => {
9         val datas = line.split(" ")
10        val time = datas(3)
11        val sdf = new SimpleDateFormat("dd/MM/yyyy:HH:mm:ss")
12        val date = sdf.parse(time)
13        val sdf1 = new SimpleDateFormat("HH")
14        val hour = sdf1.format(date)
15        (hour, 1)
16    }
17 ).groupBy(_._1)
18
19 timeRDD.map{
20     case (hour, iter) => {
21         (hour, iter.size)
22     }
23 }.collect.foreach(println)
24
25 sc.stop()

```

mapPartitions

```

1 // 以分区为单位进行操作
2 // 各个分区进行自我操作
3 // 但是会将整个分区的数据加载到内存进行引用
4 // 如果处理完的数据是不会被释放的，存在对象引用
5 // 在内存较小，数据量较大的场合下，容易出现内存溢出
6 // 传入迭代器，返回一个迭代器
7 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("mapPartitions")
8 val sc = new SparkContext(sparkConf)

```

```

9     val rdd = sc.makeRDD(
10         Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 2
11     )
12     rdd.mapPartitions(
13         iter=>Iterator(iter.toArray)
14     ).collect.foreach(item => println(item.toList))
15
16     sc.stop()
17     // 返回结果
18     // List(1, 2, 3, 4, 5)
19     // List(6, 7, 8, 9, 10)

```

```

1     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("mapPartitions")
2     val sc = new SparkContext(sparkConf)
3
4     val rdd = sc.makeRDD(List(1, 2, 3, 4), 2)
5
6     // 把一个分区的数据都拿到了以后做操作
7     val mpRDD = rdd.mapPartitions(
8         iter => {
9             println(">>>>>>>")
10            iter.map(_*2)
11        }
12    )
13
14    mpRDD.collect().foreach(println)
15
16    sc.stop()

```

```

1     // 获取每个分区数据的最大值
2     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("mapPartitions")
3     val sc = new SparkContext(sparkConf)
4
5     val rdd = sc.makeRDD(List(1, 2, 3, 4), 2)
6
7     val mapRDD = rdd.mapPartitions(
8         iter => {
9             List(iter.max).iterator
10        }
11    )
12
13    mapRDD.collect().foreach(println)
14
15    sc.stop()

```

mapPartitionsWithIndex

```

1     /**
2      * 通过对这个RDD的每个分区应用一个函数来返回一个新的RDD，
3      * 同时跟踪原始分区的索引。
4      * mapPartitionsWithIndex类似于mapPartitions()，
5      * 但它提供了第二个参数索引，用于跟踪分区。
6      */
7     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("mapPartitionsWithIndex")
8     val sc = new SparkContext(sparkConf)
9     val rdd = sc.makeRDD(
10         List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 2

```



```

11 )
12 def f(partitionIndex:Int, i:Iterator[Int])= {
13     (partitionIndex, i.sum).productIterator
14 }
15 rdd.mapPartitionsWithIndex(f).collect().foreach(println)
16
17 sc.stop()
18 // 返回结果:0 15 1 40

```

```

1 // 获取第二个分区的数据
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("mapPartitionsWithIndex")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD(List(1, 2, 3, 4), 2)
5
6 val mapiRDD = rdd.mapPartitionsWithIndex(
7     (index, iter) => {
8         if( index == 1) {
9             iter
10        }else{
11            Nil.iterator
12        }
13    }
14 )
15
16 mapiRDD.collect().foreach(println)
17
18 sc.stop()

```

```

1 // 查看数据所在分区
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("mapPartitionsWithIndex")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD(List(1, 2, 3, 4), 2)
5
6 val mapiRDD = rdd.mapPartitionsWithIndex(
7     (index, iter) => {
8         iter.map(
9             num => {
10                 (index, num)
11             }
12         )
13     }
14 )
15
16 mapiRDD.collect().foreach(println)
17
18 sc.stop()

```

repartition

```

1 // repartition()用于增加或减少RDD分区
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(1, 2, 3, 4)
6 )
7 println("当前分区数: "+rdd.partitions.size)
8 println("=====")

```

```

9    //得先repartition
10   val rdd2 = rdd.repartition(4)
11   println("重分区后, 现在的分区数: "+rdd2.partitions.size)
12
13   sc.stop()
14   // 返回结果
15   // 当前分区数: 8
16   // =====
17   // 重分区后, 现在的分区数: 4

```

sample

```

1    //从数据中抽取数据
2    /**
3     * 第一个参数: 抽取的数据是否放回, false: 不放回
4     * 第二个参数: 抽取的几率, 范围在[0,1]之间, 0: 全不取; 1: 全取;
5     * 第三个参数: 随机数种子
6     */
7    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("map")
8    val sc = new SparkContext(sparkConf)
9    val rdd = sc.makeRDD(
10       List(1, 2, 3, 4)
11    )
12    val rdd2 = rdd.sample(false, 0.5)
13    rdd2.collect().foreach(println)
14    sc.stop()
15    // 返回结果 1 3

```

sortBy

```

1    /**
2     * 排序数据。(排序大小)
3     * 在排序之前, 可以将数据通过 f 函数进行处理, 之后按照 f 函数处理
4     * 的结果进行排序
5     */
6    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("transformation")
7    val sc = new SparkContext(sparkConf)
8    val rdd = sc.makeRDD (
9       List(1, 2, 3, 4, 5, 9, 6, 46)
10    )
11    //False为降序
12    rdd.sortBy(x => x, false).collect().foreach(print)
13    println("=====")
14    //默认升序
15    rdd.sortBy(x => x).collect().foreach(print)
16    sc.stop()
17    // 返回结果
18    // 46 9 6 5 4 3 2 1
19    // =====
20    // 1 2 3 4 5 6 9 46

```

4 SparkCore键值算子

partitionBy

```
1 // partitionBy根据指定的分区规则对数据进行重分区
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("partitionBy")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD(List(1, 2, 3, 4))
5
6 // 隐式转换(二次编译)
7 val mapRDD = rdd.map(_ * 1)
8
9 mapRDD.partitionBy(new HashPartitioner(2))
10
11 sc.stop()
```

aggregateByKey

```
1 // 对数据的key按照不同的规则进行分区内计算和分区间计算
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("aggregateByKey")
3 val sc = new SparkContext(sparkConf)
4 val dataRDD1 = sc.makeRDD(List(("a", 1), ("a", 2), ("a", 3), ("a", 4)), 2)
5
6 // aggregateByKey存在函数柯里化，有两个参数列表
7
8 // 第一个参数：需要传递一个参数，表示为初始值
9 //           主要用于当碰见第一个key的时候，和value进行分区内计算
10
11 // 第二个参数：需要传递2个函数
12 //           第一个表示分区内计算规则
13 //           第二个表示分区间计算规则
14
15 val dataRDD2 = dataRDD1.aggregateByKey(0)(
16     (x, y) => math.max(x, y), // 分区内求最大值
17     (x, y) => x + y // 分区间求和
18 )
19 dataRDD2.collect().foreach(println)
20 sc.stop()
21 // 返回结果 (a, 6)
```

```
1 // aggregateByKey最终的返回数据结果应该和初始值的类型保持一致
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("aggregateByKey")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD(
5     List(
6         ("a", 1), ("a", 2), ("b", 3),
7         ("b", 4), ("b", 5), ("a", 6)
8     ), 2
9 )
10
11 // 获取相同key的数据的平均值 => (a, 3), (b, 4)
12 val newRDD = rdd.aggregateByKey((0, 0))(
13     (t, v) => {
14         (t._1 + v, t._2 + 1)
15     },
16     (t1, t2) => {
17         (t1._1 + t2._1, t1._2 + t2._2)
18     }
19 )
```

```

18     }
19 )
20
21 val resultRDD = newRDD.mapValues{
22     case(num, cnt) => {
23         num / cnt
24     }
25 }
26
27 resultRDD.collect().foreach(println)
28
29 sc.stop()

```

combineByKey

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("combineByKey")
2  val sc = new SparkContext(sparkConf)
3  val rdd = sc.makeRDD(
4      List(
5          ("a", 1), ("a", 2), ("b", 3),
6          ("b", 4), ("b", 5), ("a", 6)
7      ), 2
8  )
9
10 // combineByKey方法需要三个参数
11 // 第一个参数表示：将相同key的第一个数据进行结构的转换，实现操作
12 // 第二个参数表示：分区内的计算规则
13 // 第三个参数表示：分区间的计算规则
14
15 val newRDD = rdd.combineByKey(
16     v => (v, 1),
17     (t: (Int, Int), v) => {
18         (t._1 + v, t._2 + 1)
19     },
20     (t1: (Int, Int), t2: (Int, Int)) => {
21         (t1._1 + t2._1, t1._2 + t2._2)
22     }
23 )
24
25 val resultRDD = newRDD.mapValues{
26     case(num, cnt) => {
27         num / cnt
28     }
29 }
30
31 sc.stop()

```

cogroup

```

1  // 对两个RDD中的KV元素，每个RDD中相同key中的元素分别聚合成一个集合
2  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("cogroup")
3  val sc = new SparkContext(sparkConf)
4  val rdd1 = sc.makeRDD(
5      List(
6          ("a", 1), ("b", 2), ("c", 3)
7      )
8  )

```

```

9     val rdd2 = sc.makeRDD(
10         List(
11             ("a", 4), ("b", 5), ("c", 6)
12         )
13     )
14     rdd1.cogroup(rdd2).collect().foreach(print)
15     sc.stop()
16     // 返回结果
17     // (a, (CompactBuffer(1), CompactBuffer(4)))
18     // (b, (CompactBuffer(2), CompactBuffer(5)))
19     // (c, (CompactBuffer(3), CompactBuffer(6)))

```

foldByKey

```

1     // 当分区内计算规则和分区间计算规则相同时, aggregateByKey 就可以简化为 foldByKey
2     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("foldByKey")
3     val sc = new SparkContext(sparkConf)
4     val dataRDD1 = sc.makeRDD(
5         List(
6             ("a", 1), ("a", 2), ("b", 3),
7             ("b", 4), ("b", 5), ("a", 6)
8         ), 2
9     )
10
11     dataRDD1.foldByKey(0)(_+_).collect().foreach(println)
12
13     sc.stop()
14     // ("a", 9)
15     // ("b", 12)

```

groupByKey

```

1     // 将数据源的数据根据 key 对 value 进行分组, 形成一个对偶元组
2     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("groupByKey")
3     val sc = new SparkContext(sparkConf)
4     val dataRDD1 = sc.makeRDD(List(("a", 1), ("a", 2), ("a", 3), ("b", 4)))
5     val dataRDD2 = dataRDD1.groupByKey()
6     dataRDD2.collect().foreach(println)
7     sc.stop()
8     // 返回结果
9     // (a, CompactBuffer(1, 2, 3))
10    // (b, CompactBuffer(4))

```

join

```

1     // 在类型为(K, V)和(K, W)的 RDD 上调用
2     // 返回一个相同 key对应的所有元素连接在一起的
3     // (K, (V, W))的 RDD
4     // 如果两个数据源中key没有匹配上, 那么数据不会出现在结果中
5     // 如果两个数据源中key有多个相同的, 会依次匹配, 可能会出现笛卡尔集
6     val sparkConf = new SparkConf().setMaster("local[*]").setAppName("join")
7     val sc = new SparkContext(sparkConf)
8     val rdd = sc.makeRDD(Array(("a", 1), ("b", 2), ("c", 3)))
9     val rdd1 = sc.makeRDD(Array(("a", 4), ("b", 5), ("c", 6)))
10    rdd.join(rdd1).collect().foreach(println)
11    sc.stop()
12    // 返回结果

```

```

13 // (a, (1, 4))
14 // (b, (2, 5))
15 // (c, (3, 6))

```

leftOuterJoin

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("leftOuterJoin")
2  val sc = new SparkContext(sparkConf)
3  val rdd1 = sc.makeRDD(
4      List(
5          ("a", 1), ("b", 2)
6      )
7  )
8  val rdd2 = sc.makeRDD(
9      List(
10         ("a", 4), ("b", 5), ("c", 6)
11     )
12 )
13 rdd1.leftOuterJoin(rdd2).collect().foreach(print)
14 sc.stop()
15 // 返回结果 (a, (Some(1), 4))
16 //           (b, (Some(2), 5))
17 //           (c, (None, 6))

```

reduceByKey

```

1  // 可以将数据按照相同的 Key 对 Value 进行聚合
2  // reduceByKey中如果key的数据只有一个，是不会参与计算的
3  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("reduceByKey")
4  val sc = new SparkContext(sparkConf)
5  val rdd = sc.makeRDD (
6      List(("a", 1), ("b", 2), ("c", 3), ("a", 4))
7  )
8  val result = rdd.reduceByKey(_ + _)
9  result.collect().foreach(println)
10 println("=====")
11 sc.stop()
12 // 返回结果
13 // ("a", 5)
14 // ("b", 2)
15 // ("c", 3)

```

sortByKey

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("partitionBy")
2  val sc = new SparkContext(sparkConf)
3  val dataRDD1 = sc.makeRDD(List(("a", 1), ("b", 2), ("c", 3)))
4  dataRDD1.sortByKey(true).collect().foreach(println)
5  println("=====")
6  dataRDD1.sortByKey(false).collect().foreach(println)
7  // 返回结果
8  // (a, 1)
9  // (b, 2)
10 // (c, 3)
11 // =====
12 // (c, 3)
13 // (b, 2)

```

```
14    // (a, 1)
```

5 SparkCore行动算子

行动算子：触发任务的调度和作业的执行

aggregate

```
1    //第一个是分区内操作
2    //第二个是分区间操作
3    //aggregate 方法是一个聚合函数，接受多个输入，并按照一定的规则运算以后输出一个结果值。
4    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
5    val sc = new SparkContext(sparkConf)
6    val rdd = sc.makeRDD (
7        List(1, 2, 3, 4), 2
8    )
9    val result = rdd.aggregate(0)(_ + _, _ + _)
10   // 各自分区内：0+1+2=3 0+3+4=7
11   // 分区间：0+3+7=10
12   val result2 = rdd.aggregate(10)(_ + _, _ + _)
13   // 各自分区间内：1+2+10=13 3+4+10=17
14   // 分区间：10+13+17=40
15   // *****初始值也会参与分区间的计算*****
16   // 如果分区数为8:13+17+10*(8-1)=100
17   println(result)
18   println(result2)
19
20   // 输出结果为10和40
```

collect

```
1    // collect在sparksql中特别重要
2    // 以一张静态网页举例，该网页全是table，里面都是td和tr
3    // 现在要爬取该网页tr里面的文字内容
4    // 用python的pandas.read_html()即可提取该网页并且变成DataFrame进行存储
5    // SparkSQL在show()后的结果就和该网页一样，有大量的tr格式
6    // 加上collect()后就和read_html()一样，可以取消格式并且将数据变成集合进行存储
7    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
8    val sc = new SparkContext(sparkConf)
9    val rdd = sc.makeRDD (
10        List(1, 2, 3, 4)
11    )
12    println(rdd.collect() (0))
13    // 输出结果为1
```

count

```
1    // count是计算rdd中的元素个数
2    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
3    val sc = new SparkContext(sparkConf)
4    val rdd = sc.makeRDD (
5        List(1, 2, 3, 4)
6    )
7    val countResult = rdd.count()
8    println(countResult)
9    // 输出结果为4
```

countByKey

```
1 // countByKey是统计每种key的个数
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(
6         (1, "a"), (1, "a"), (1, "a"), (1, "a"), (2, "b"), (3, "c"), (3, "c")
7     )
8 )
9 val result = rdd.countByKey()
10 println(result.mkString(","))
11 // 输出结果 1 -> 4,2 -> 1,3 -> 2
```

first

```
1 // first返回rdd中的第一个元素
2 // 等价于rdd.collect() (0)
3 // rdd.first() == rdd.collect() (0)
4 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
5 val sc = new SparkContext(sparkConf)
6 val rdd = sc.makeRDD (
7     List(1, 2, 3, 4)
8 )
9 println(rdd.first())
10 // 返回结果为1
```

fold

```
1 // aggregate的简化版
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(1, 2, 3, 4), 2
6 )
7 val result = rdd.fold(10) (_ + _)
8 println(result)
9 // 返回结果为40
```

reduce

```
1 // reduce聚合rdd中所有的元素
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(1, 2, 3, 4), 2
6 )
7 val rdd2 = rdd.reduce(_ + _)
8 println(rdd2)
9 // 返回结果为10
```


take

```
1 // 返回一个有RDD的前n个元素组成的素组
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
3 val sc = new SparkContext(sparkConf)
4 val rdd = sc.makeRDD (
5     List(1, 2, 3, 4)
6 )
7 val takeResult = rdd.take(2)
8 println(takeResult.mkString(", "))
9 // 返回结果为1,2
```

takeOrdered

```
1 // 返回RDD排序后的前n个元素组成的素组
2 // 先将RDD进行排序，然后取前n个元素
3 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("action")
4 val sc = new SparkContext(sparkConf)
5 val rdd = sc.makeRDD (
6     List(1, 2, 4, 3, 3, 5, 7, 0)
7 )
8 val result = rdd.takeOrdered(5)
9 println(result.mkString(", "))
10 // 返回结果为0,1,2,3,3
```

6 SparkCore双值算子

cartesian

```
1 // 笛卡尔集
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("DoubleValue")
3 val sc = new SparkContext(sparkConf)
4 val rdd1 = sc.makeRDD(
5     List(1, 2, 3, 4)
6 )
7 val rdd2 = sc.makeRDD(
8     List(3, 4, 5)
9 )
10 val rdd3 = rdd1.cartesian(rdd2)
11 rdd3.collect().foreach(println)
12
13 sc.stop()
14 // 返回结果 (1,3) (1,4) (1,5)
15 //           (2,3) (2,4) (2,5)
16 //           (3,3) (3,4) (3,5)
17 //           (4,3) (4,4) (4,5)
```

intersection

```
1 // 交集
2 val sparkConf = new SparkConf().setMaster("local[*]").setAppName("DoubleValue")
3 val sc = new SparkContext(sparkConf)
4 val rdd1 = sc.makeRDD(
5     List(1, 2, 3, 4)
6 )
7 val rdd2 = sc.makeRDD(
```

```

8         List(3, 4, 5)
9     )
10    val rdd3 = rdd1.intersection(rdd2)
11    rdd3.collect().foreach(println)
12
13    sc.stop()
14    // 返回结果 3 4

```

subtract

```

1    // 差集, 返回第一个rdd中有而第二个没的
2    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("DoubleValue")
3    val sc = new SparkContext(sparkConf)
4    val rdd1 = sc.makeRDD(
5        List(1, 2, 3, 4)
6    )
7    val rdd2 = sc.makeRDD(
8        List(3, 4, 5)
9    )
10    val rdd3 = rdd1.subtract(rdd2)
11    rdd3.collect().foreach(println)
12
13    sc.stop()
14    // 返回结果 1 2

```

union

```

1    // 并集两个RDD, 不会去重
2    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("DoubleValue")
3    val sc = new SparkContext(sparkConf)
4    val rdd1 = sc.makeRDD(
5        List(1, 2, 3, 4)
6    )
7    val rdd2 = sc.makeRDD(
8        List(3, 4, 5)
9    )
10    val rdd3 = rdd1.union(rdd2)
11    rdd3.collect().foreach(println)
12
13    sc.stop()
14    // 返回结果 1 2 3 4 3 4 5

```

zip

```

1    // 两个RDD拉链
2    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("DoubleValue")
3    val sc = new SparkContext(sparkConf)
4    val rdd1 = sc.makeRDD(
5        List(1, 2, 3, 4)
6    )
7    val rdd2 = sc.makeRDD(
8        List(3, 4, 5, 6)
9    )
10    val rdd3 = rdd1.zip(rdd2)
11    rdd3.collect().foreach(println)
12
13    sc.stop()

```

7 SparkCore练习

1.创建一个1-10数组的RDD，将所有元素*2形成新的RDD

```
1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam1")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(1 to 10)
4  val newRDD = inputRDD.map(_*2)
5  newRDD.collect().foreach(println)
```

2.创建一个10-20数组的RDD，使用mapPartitions将所有元素*2形成新的RDD

```
1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam2")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(10 to 20)
4  val newRDD = inputRDD.mapPartitions({
5      iter => {
6          iter.map(_*2)
7      }
8  })
9  newRDD.collect().foreach(println)
```

3.创建一个元素为 1-5 的RDD，运用 flatMap创建一个新的 RDD，新的 RDD 为原 RDD 每个元素的平方和三次方 来组成 1,1,4,8,9,27...

```
1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam3")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(1 to 5)
4  val newRDD = inputRDD.flatMap(n => {
5      List(Math.pow(n, 2).toInt, Math.pow(n, 3).toInt)
6  })
7  newRDD.collect().foreach(println)
```

4.创建一个 4 个分区的 RDD数据为Array(10,20,30,40,50,60)，使用glom将每个分区的数据放到一个数组

```
1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam4")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      Array(10, 20, 30, 40, 50, 60), 4
5  )
6  val newRDD = inputRDD.glom()
7  newRDD.foreach(x => println(x.mkString(", ")))
```

5.创建一个 RDD数据为Array(1, 3, 4, 20, 4, 5, 8)，按照元素的奇偶性进行分组

```
1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam5")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      Array(1, 3, 4, 20, 4, 5, 8)
5  )
6  val newRDD = inputRDD.groupBy(_*2==0)
7  newRDD.collect().foreach(println)
```

6.创建一个 RDD（由字符串组成）Array("xiaoli", "laoli", "laowang", "xiaocang", "xiaojing", "xiaokong")，过滤出一个新 RDD（包含"xiao"子串）

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam6")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      Array("xiaoli", "laoli", "laowang", "xiaocang", "xiaojing", "xiaokong")
5  )
6  val newRDD = inputRDD.filter(_.contains("xiao"))
7  newRDD.collect().foreach(println)

```

7.创建一个 RDD数据为1 to 10，请使用sample不放回抽样

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam7")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(1 to 10)
4  val newRDD = inputRDD.sample(false, 0.5)
5  newRDD.collect().foreach(println)

```

8.创建一个 RDD数据为Array(10,10,2,5,3,5,3,6,9,1),对 RDD 中元素执行去重操作

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam8")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      Array(10, 10, 2, 5, 3, 5, 3, 6, 9, 1)
5  )
6  val newRDD = inputRDD.distinct()
7  newRDD.collect().foreach(println)

```

9.创建一个分区数为5的 RDD，数据为0 to 100，之后使用coalesce再重新减少分区数量至 2

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam9")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(0 to 100, 5)
4  val newRDD = inputRDD.coalesce(2)
5  println(newRDD.partitions.length)

```

10.创建一个分区数为5的 RDD，数据为0 to 100，之后使用repartition再重新减少分区数量至 3

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam10")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(0 to 100, 5)
4  val newRDD = inputRDD.repartition(3)
5  println(newRDD.partitions.length)

```

11.创建一个 RDD数据为1,3,4,10,4,6,9,20,30,16,请给RDD进行分别进行升序和降序排列

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam11")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      Array(1, 3, 4, 10, 4, 6, 9, 20, 30, 16)
5  )
6  val newRDD1 = inputRDD.sortBy(x => x, true)
7  newRDD1.collect().foreach(print)
8  println()
9  println("=====")
10 val newRDD2 = inputRDD.sortBy(x => x, false)
11 newRDD2.collect().foreach(print)

```

12.创建两个RDD，分别为rdd1和rdd2数据分别为1 to 6和4 to 10，求并集

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam12")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD1 = sc.makeRDD(1 to 6)
4  val inputRDD2 = sc.makeRDD(4 to 10)
5  val newRDD = inputRDD1.union(inputRDD2)
6  newRDD.collect().foreach(println)

```

13.创建一个RDD数据为List(("female",1),("male",5),("female",5),("male",2)), 请计算出female和male的总数分别为多少

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam13")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      List(("female", 1), ("male", 5), ("female", 5), ("male", 2))
5  )
6  val newRDD = inputRDD.reduceByKey(_+_ )
7  newRDD.collect().foreach(println)

```

14.创建一个有两个分区的 RDD数据为List(("a",3),("a",2),("c",4),("b",3),("c",6),("c",8)), 取出每个分区相同key对应值的最大值, 然后相加

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam14")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      List(("a", 3), ("a", 2), ("c", 4), ("b", 3), ("c", 6), ("c", 8))
5  )
6  inputRDD.aggregateByKey(0) (
7      (tmp, item) => {
8          println(tmp, item, "===")
9          Math.max(tmp, item)
10     },
11     (tmp, result) => {
12         println(tmp, result, "===")
13         tmp + result
14     }
15 ).foreach(println(_))

```

15.创建一个有两个分区的 pairRDD数据为Array(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98)), 根据 key 计算每种 key 的value的平均值

```

1  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Exam15")
2  val sc = new SparkContext(sparkConf)
3  val inputRDD = sc.makeRDD(
4      Array(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98)), 2
5  )
6  inputRDD.groupByKey()
7      .map(x => x._2.sum / x._2.size)
8      .foreach(println)

```

8 SparkCore的shuffle操作

理解shuffle

shuffle介绍

在 Spark 中，一个任务对应一个分区，通常不会跨分区操作数据。但如果遇到 `reduceByKey` 等操作，Spark 必须从所有分区读取数据，并查找所有键的所有值，然后汇总在一起以计算每个键的最终结果，这称为 `Shuffle`。

Shuffle的影响

Shuffle 是一项昂贵的操作，因为它通常会跨节点操作数据，这会涉及磁盘 I/O、网络 I/O和数据序列化。某些Shuffle 操作还会消耗大量的堆内存，因为它们使用堆内存来临时存储需要网络传输的数据。Shuffle 还会在磁盘上生成大量中间文件，从 Spark 1.3 开始，这些文件将被保留，直到相应的 RDD 不再使用并进行垃圾回收，这样做是为了避免在计算时重复创建 Shuffle 文件。如果应用程序长期保留对这些 RDD 的引用，则垃圾回收可能在很长一段时间后才会发生，这意味着长时间运行的 Spark 作业可能会占用大量磁盘空间，通常可以使用 `spark.local.dir` 参数来指定这些临时文件的存储目录。

导致Shuffle的操作

由于 Shuffle 操作对性能的影响比较大，所以需要特别注意使用，以下操作都会导致 Shuffle：

- **涉及到重新分区操作**：如 `repartition` 和 `coalesce`；
- **所有涉及到 ByKey 的操作**：如 `groupByKey` 和 `reduceByKey`，但 `countByKey` 除外；
- **联结操作**：如 `cogroup` 和 `join`。

第二章 SparkSQL

1 DataFrame和DataSet

Spark SQL简介

Spark SQL 是 Spark 中的一个子模块，主要用于操作结构化数据。它具有以下特点：

- 能够将 SQL 查询与 Spark 程序无缝混合，允许您使用 SQL 或 DataFrame API 对结构化数据进行查询；
- 支持多种开发语言；
- 支持多达上百种的外部数据源，包括 Hive，Avro，Parquet，ORC，JSON 和 JDBC 等；
- 支持 HiveQL 语法以及 Hive SerDes 和 UDF，允许你访问现有的 Hive 仓库；
- 支持标准的 JDBC 和 ODBC 连接；
- 支持优化器，列式存储和代码生成等特性；
- 支持扩展并能保证容错。

DataFrame & DataSet

DataFrame

为了支持结构化数据的处理，Spark SQL 提供了新的数据结构 DataFrame。DataFrame 是一个由具名列组成的数据集。它在概念上等同于关系数据库中的表或 R/Python 语言中的 `data frame`。由于 Spark SQL 支持多种语言的开发，所以每种语言都定义了 `DataFrame` 的抽象，主要如下：

语言	主要抽象
Scala	Dataset[T] & DataFrame (Dataset[Row] 的别名)
Java	Dataset[T]
Python	DataFrame
R	DataFrame

DataFrame 对比 RDDs

DataFrame 和 RDDs 最主要的区别在于一个面向的是结构化数据，一个面向的是非结构化数据。

DataFrame 内部的有明确 Scheme 结构，即列名、列字段类型都是已知的，这带来的好处是可以减少数据读取以及更好地优化执行计划，从而保证查询效率。

DataFrame 和 RDDs 应该如何选择？

- 如果你想使用函数式编程而不是 DataFrame API，则使用 RDDs；
- 如果你的数据是非结构化的 (比如流媒体或者字符流)，则使用 RDDs，
- 如果你的数据是结构化的 (如 RDBMS 中的数据) 或者半结构化的 (如日志)，出于性能上的考虑，应优先使用 DataFrame。

DataSet

Dataset 也是分布式的数据集合，在 Spark 1.6 版本被引入，它集成了 RDD 和 DataFrame 的优点，具备强类型的特点，同时支持 Lambda 函数，但只能在 Scala 和 Java 语言中使用。在 Spark 2.0 后，为了方便开发者，Spark 将 DataFrame 和 Dataset 的 API 融合到一起，提供了结构化的 API (Structured API)，即用户可以通过一套标准的 API 就能完成对两者的操作。

静态类型与运行时类型安全

静态类型 (Static-typing) 与运行时类型安全 (runtime type-safety) 主要表现如下：

在实际使用中，如果你用的是 Spark SQL 的查询语句，则直到运行时你才会发现有语法错误，而如果你用的是 DataFrame 和 Dataset，则在编译时就可以发现错误 (这节省了开发时间和整体代价)。DataFrame 和 Dataset 主要区别在于：

在 DataFrame 中，当你调用了 API 之外的函数，编译器就会报错，但如果你使用了一个不存在的字段名字，编译器依然无法发现。而 Dataset 的 API 都是用 Lambda 函数和 JVM 类型对象表示的，所有不匹配的类型参数在编译时就会被发现。

以上这些最终都被解释成关于类型安全图谱，对应开发中的语法和分析错误。在图谱中，Dataset 最严格，但对于开发者来说效率最高。

上面的描述可能并没有那么直观，下面的给出一个 IDEA 中代码编译的示例：

这里一个可能的疑惑是 DataFrame 明明是有确定的 Scheme 结构 (即列名、列字段类型都是已知的)，但是为什么还是无法对列名进行推断和错误判断，这是因为 DataFrame 是 Untyped 的。

Untyped & Typed

在上面我们介绍过 DataFrame API 被标记为 `Untyped API`，而 DataSet API 被标记为 `Typed API`。DataFrame 的 `Untyped` 是相对于语言或 API 层面而言，它确实有明确的 Scheme 结构，即列名，列类型都是确定的，但这些信息完全由 Spark 来维护，Spark 只会在运行时检查这些类型和指定类型是否一致。这也就是为什么在 Spark 2.0 之后，官方推荐把 DataFrame 看做是 `DatSet[Row]`，Row 是 Spark 中定义的一个 `trait`，其子类中封装了列字段的信息。

相对而言，DataSet 是 `Typed` 的，即强类型。如下面代码，DataSet 的类型由 Case Class(Scala) 或者 Java Bean(Java) 来明确指定的，在这里即每一行数据代表一个 `Person`，这些信息由 JVM 来保证正确性，所以字段名错误和类型错误在编译的时候就会被 IDE 所发现。

```
1 case class Person(name: String, age: Long)
2 val dataSet: Dataset[Person] = spark.read.json("people.json").as[Person]
```

DataFrame & DataSet & RDDs 总结

这里对三者做一下简单的总结：

- RDDs 适合非结构化数据的处理，而 DataFrame & DataSet 更适合结构化数据和半结构化的处理；
- DataFrame & DataSet 可以通过统一的 Structured API 进行访问，而 RDDs 则更适合函数式编程的场景；
- 相比于 DataFrame 而言，DataSet 是强类型的 (Typed)，有着更为严格的静态类型检查；
- DataSets、DataFrames、SQL 的底层都依赖了 RDDs API，并对外提供结构化的访问接口。

Spark SQL的运行原理

DataFrame、DataSet 和 Spark SQL 的实际执行流程都是相同的：

1. 进行 DataFrame/Dataset/SQL 编程；
2. 如果是有效的代码，即代码没有编译错误，Spark 会将其转换为一个逻辑计划；
3. Spark 将此逻辑计划转换为物理计划，同时进行代码优化；
4. Spark 然后在集群上执行这个物理计划 (基于 RDD 操作)。

逻辑计划(Logical Plan)

执行的第一个阶段是将用户代码转换成一个逻辑计划。它首先将用户代码转换成 `unresolved logical plan` (未解决的逻辑计划)，之所以这个计划是未解决的，是因为尽管您的代码在语法上是正确的，但是它引用的表或列可能不存在。Spark 使用 `analyzer` (分析器) 基于 `catalog` (存储的所有表和 DataFrames 的信息) 进行解析。解析失败则拒绝执行，解析成功则将结果传给 `Catalyst` 优化器 (`Catalyst Optimizer`)，优化器是一组规则的集合，用于优化逻辑计划，通过谓词下推等方式进行优化，最终输出优化后的逻辑执行计划。

物理计划(Physical Plan)

得到优化后的逻辑计划后，Spark 就开始了物理计划过程。它通过生成不同的物理执行策略，并通过成本模型来比较它们，从而选择一个最优的物理计划在集群上面执行的。物理规划的输出结果是一系列的 RDDs 和转换关系 (transformations)。

执行

在选择一个物理计划后，Spark 运行其 RDDs 代码，并在运行时执行进一步的优化，生成本地 Java 字节码，最后将运行结果返回给用户。

2 SparkSQL外部数据源

简介

多数据源支持

Spark 支持以下六个核心数据源，同时 Spark 社区还提供了多达上百种数据源的读取方式，能够满足绝大部分使用场景。

- CSV
- JSON
- Parquet
- ORC
- JDBC/ODBC connections
- Plain-text files

读数据格式

所有读取 API 遵循以下调用格式：

```
1 // 格式
2 DataFrameReader.format(...).option("key", "value").schema(...).load()
3
4 // 示例
5 spark.read.format("csv")
6 .option("mode", "FAILFAST")           // 读取模式
7 .option("inferSchema", "true")        // 是否自动推断 schema
8 .option("path", "path/to/file(s)")    // 文件路径
9 .schema(someSchema)                   // 使用预定义的 schema
10 .load()
```

读取模式有以下三种可选项：

读模式	描述
permissive	当遇到损坏的记录时，将其所有字段设置为 null，并将所有损坏的记录放在名为 _corruption_t_record 的字符串列中
dropMalformed	删除格式不正确的行
failFast	遇到格式不正确的数据时立即失败

写数据格式

```

1 // 格式
2 DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()
3
4 //示例
5 dataframe.write.format("csv")
6 .option("mode", "OVERWRITE") //写模式
7 .option("dateFormat", "yyyy-MM-dd") //日期格式
8 .option("path", "path/to/file(s)")
9 .save()

```

写数据模式有以下四种可选项：

Scala/Java	描述
<code>SaveMode.ErrorIfExists</code>	如果给定的路径已经存在文件，则抛出异常，这是写数据默认的模式
<code>SaveMode.Append</code>	数据以追加的方式写入
<code>SaveMode.Overwrite</code>	数据以覆盖的方式写入
<code>SaveMode.Ignore</code>	如果给定的路径已经存在文件，则不做任何操作

CSV

CSV 是一种常见的文本文件格式，其中每一行表示一条记录，记录中的每个字段用逗号分隔。

读取csv文件

自动推断类型读取读取示例：

```

1 spark.read.format("csv")
2 .option("header", "false") // 文件中的第一行是否为列的名称
3 .option("mode", "FAILFAST") // 是否快速失败
4 .option("inferSchema", "true") // 是否自动推断 schema
5 .load("/usr/file/csv/dept.csv")
6 .show()

```

使用预定义类型：

```

1 import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
2 //预定义数据格式
3 val myManualSchema = new StructType(Array(
4     StructField("deptno", LongType, nullable = false),
5     StructField("dname", StringType, nullable = true),
6     StructField("loc", StringType, nullable = true)
7 ))
8 spark.read.format("csv")
9 .option("mode", "FAILFAST")
10 .schema(myManualSchema)
11 .load("/usr/file/csv/dept.csv")
12 .show()

```

写入csv文件

```

1 df.write.format("csv").mode("overwrite").save("/tmp/csv/dept2")

```

也可以指定具体的分隔符：

```
1 df.write.format("csv").mode("overwrite").option("sep", "\t").save("/tmp/csv/dept2")
```

JSON

读取JSON文件

```
1 spark.read.format("json").option("mode", "FAILFAST").load("/usr/file/json/dept.json").show(5)
```

需要注意的是：默认不支持一条数据记录跨越多行 (如下)，可以通过配置 `multiLine` 为 `true` 来进行更改，其默认值为 `false`。

```
1 // 默认支持单行
2 {"DEPTNO": 10, "DNAME": "ACCOUNTING", "LOC": "NEW YORK"}
3
4 //默认不支持多行
5 {
6     "DEPTNO": 10,
7     "DNAME": "ACCOUNTING",
8     "LOC": "NEW YORK"
9 }
```

写入JSON文件

```
1 df.write.format("json").mode("overwrite").save("/tmp/spark/json/dept")
```

Parquet

Parquet 是一个开源的面向列的数据存储，它提供了多种存储优化，允许读取单独的列非整个文件，这不仅节省了存储空间而且提升了读取效率，它是 Spark 默认的文件格式。

读取Parquet文件

```
1 spark.read.format("parquet").load("/usr/file/parquet/dept.parquet").show(5)
```

写入Parquet文件

```
1 df.write.format("parquet").mode("overwrite").save("/tmp/spark/parquet/dept")
```

可选配置

Parquet 文件有着自己的存储规则，因此其可选配置项比较少，常用的有如下两个：

读写操作	配置项	可选值	默认值	描述
Write	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	None	压缩文件格式
Read	mergeSchema	true, false	取决于配置项 <code>spark.sql.parquet.mergeSchema</code>	当为真时，Parquet 数据源将所有数据文件收集的 Schema 合并在一起，否则将从摘要文件中选择 Schema，如果没有可用的摘要文件，则从随机数据文件中选择 Schema。

更多可选配置可以参阅官方文档：<https://spark.apache.org/docs/latest/sql-data-sources-parquet.html>

ORC

ORC 是一种自描述的、类型感知的列文件格式，它针对大型数据的读写进行了优化，也是大数据中常用的文件格式。

读取ORC文件

```
1 spark.read.format("orc").load("/usr/file/orc/dept.orc").show(5)
```

写入ORC文件

```
1 csvFile.write.format("orc").mode("overwrite").save("/tmp/spark/orc/dept")
```

SQL Databases

Spark 同样支持与传统的关系型数据库进行数据读写。但是 Spark 程序默认是没有提供数据库驱动的，所以在使用前需要将对应的数据库驱动上传到安装目录下的 `jars` 目录中。下面示例使用的是 Mysql 数据库，使用前需要将对应的 `mysql-connector-java-x.x.x.jar` 上传到 `jars` 目录下。

读取数据

读取全表数据示例如下，这里的 `help_keyword` 是 mysql 内置的字典表，只有 `help_keyword_id` 和 `name` 两个字段。

```

1 spark.read
2 .format("jdbc")
3 .option("driver", "com.mysql.jdbc.Driver")           //驱动
4 .option("url", "jdbc:mysql://127.0.0.1:3306/mysql")   //数据库地址
5 .option("dbtable", "help_keyword")                   //表名
6 .option("user", "root").option("password", "root").load().show(10)

```

从查询结果读取数据:

```

1 val pushDownQuery = """(SELECT * FROM help_keyword WHERE help_keyword_id <20) AS help_keywords"""
2 spark.read.format("jdbc")
3 .option("url", "jdbc:mysql://127.0.0.1:3306/mysql")
4 .option("driver", "com.mysql.jdbc.Driver")
5 .option("user", "root").option("password", "root")
6 .option("dbtable", pushDownQuery)
7 .load().show()
8
9 //输出
10 +-----+-----+
11 |help_keyword_id|      name|
12 +-----+-----+
13 |                0|      <>|
14 |                1|    ACTION|
15 |                2|      ADD|
16 |                3|AES_DECRYPT|
17 |                4|AES_ENCRYPT|
18 |                5|      AFTER|
19 |                6|    AGAINST|
20 |                7|  AGGREGATE|
21 |                8|  ALGORITHM|
22 |                9|      ALL|
23 |               10|    ALTER|
24 |               11|    ANALYSE|
25 |               12|    ANALYZE|
26 |               13|      AND|
27 |               14|    ARCHIVE|
28 |               15|      AREA|
29 |               16|      AS|
30 |               17|  ASBINARY|
31 |               18|      ASC|
32 |               19|    ASTEXT|
33 +-----+-----+

```

也可以使用如下的写法进行数据的过滤:

```

1 val props = new java.util.Properties
2 props.setProperty("driver", "com.mysql.jdbc.Driver")
3 props.setProperty("user", "root")
4 props.setProperty("password", "root")
5 val predicates = Array("help_keyword_id < 10 OR name = 'WHEN'") //指定数据过滤条件
6 spark.read.jdbc("jdbc:mysql://127.0.0.1:3306/mysql", "help_keyword", predicates, props).show()
7
8 //输出:
9 +-----+-----+
10 |help_keyword_id|      name|
11 +-----+-----+
12 |                0|      <>|
13 |                1|    ACTION|

```

```

14 |           2 |          ADD |
15 |           3 | AES_DECRYPT |
16 |           4 | AES_ENCRYPT |
17 |           5 |          AFTER |
18 |           6 |        AGAINST |
19 |           7 |      AGGREGATE |
20 |           8 |     ALGORITHM |
21 |           9 |          ALL |
22 |          604 |          WHEN |
23 | +-----+

```

可以使用 `numPartitions` 指定读取数据的并行度：

```
1 option("numPartitions", 10)
```

在这里，除了可以指定分区外，还可以设置上界和下界，任何小于下界的值都会被分配在第一个分区中，任何大于上界的值都会被分配在最后一个分区中。

```

1 val colName = "help_keyword_id" //用于判断上下界的列
2 val lowerBound = 300L //下界
3 val upperBound = 500L //上界
4 val numPartitions = 10 //分区综述
5 val jdbcDf = spark.read.jdbc("jdbc:mysql://127.0.0.1:3306/mysql", "help_keyword",
6                               colName, lowerBound, upperBound, numPartitions, props)

```

想要验证分区内容，可以使用 `mapPartitionsWithIndex` 这个算子，代码如下：

```

1 jdbcDf.rdd.mapPartitionsWithIndex((index, iterator) => {
2     val buffer = new ListBuffer[String]
3     while (iterator.hasNext) {
4         buffer.append(index + "分区:" + iterator.next())
5     }
6     buffer.toIterator
7 }).foreach(println)

```

执行结果如下：`help_keyword` 这张表只有 600 条左右的数据，本来数据应该均匀分布在 10 个分区，但是 0 分区里面却有 319 条数据，这是因为设置了下限，所有小于 300 的数据都会被限制在第一个分区，即 0 分区。同理所有大于 500 的数据被分配在 9 分区，即最后一个分区。

写入数据

```

1 val df = spark.read.format("json").load("/usr/file/json/emp.json")
2 df.write
3   .format("jdbc")
4   .option("url", "jdbc:mysql://127.0.0.1:3306/mysql")
5   .option("user", "root").option("password", "root")
6   .option("dbtable", "emp")
7   .save()

```

Text

Text 文件在读写性能方面并没有任何优势，且不能表达明确的数据结构，所以其使用的比较少，读写操作如下：

读取Text数据

```
1 spark.read.textFile("/usr/file/txt/dept.txt").show()
```

写入Text数据

```
1 df.write.text("/tmp/spark/txt/dept")
```

数据读写高级特性

并行读

多个 Executors 不能同时读取同一个文件，但它们可以同时读取不同的文件。这意味着当您从一个包含多个文件的文件夹中读取数据时，这些文件中的每一个都将成为 DataFrame 中的一个分区，并由可用的 Executors 并行读取。

并行写

写入的文件或数据的数量取决于写入数据时 DataFrame 拥有的分区数量。默认情况下，每个数据分区写一个文件。

分区写入

分区和分桶这两个概念和 Hive 中分区表和分桶表是一致的。都是将数据按照一定规则进行拆分存储。需要注意的是 `partitionBy` 指定的分区和 RDD 中分区不是一个概念：这里的**分区表现为输出目录的子目录**，数据分别存储在对应的子目录中。

```
1 val df = spark.read.format("json").load("/usr/file/json/emp.json")
2 df.write.mode("overwrite").partitionBy("deptno").save("/tmp/spark/partitions")
```

输出结果如下：可以看到输出被按照部门编号分为三个子目录，子目录中才是对应的输出文件。

分桶写入

分桶写入就是将数据按照指定的列和桶数进行散列，目前分桶写入只支持保存为表，实际上这就是 Hive 的分桶表。

```
1 val numberBuckets = 10
2 val columnToBucketBy = "empno"
3 df.write.format("parquet").mode("overwrite")
4   .bucketBy(numberBuckets, columnToBucketBy).saveAsTable("bucketedFiles")
```

文件大小管理


如果写入产生小文件数量过多，这时会产生大量的元数据开销。Spark 和 HDFS 一样，都不能很好的处理这个问题，这被称为“small file problem”。同时数据文件也不能过大，否则在查询时会有不必要的性能开销，因此要把文件大小控制在一个合理的范围内。

在上文我们已经介绍过可以通过分区数量来控制生成文件的数量，从而间接控制文件大小。Spark 2.2 引入了一种新的方法，以更自动化的方式控制文件大小，这就是 `maxRecordsPerFile` 参数，它允许你通过控制写入文件的记录数来控制文件大小。

```
1 // Spark 将确保文件最多包含 5000 条记录
2 df.write.option("maxRecordsPerFile", 5000)
```

可选配置附录

CSV读写可选配置

读 \写 操作	配置项	可选值	默认值	描述
Both	seq	任意字符	 (逗号)	分隔符
Both	header	true, false	false	文件中的第一行是否为列的名称。
Read	escape	任意字符	\	转义字符
Read	inferSchema	true, false	false	是否自动推断列类型
Read	ignoreLeadingWhiteSpace	true, false	false	是否跳过值前面的空格
Both	ignoreTrailingWhiteSpace	true, false	false	是否跳过值后面的空格
Both	nullValue	任意字符	“”	声明文件中哪个字符表示空值
Both	nanValue	任意字符	NaN	声明哪个值表示 NaN 或者缺省值
Both	positiveInf	任意字符	Inf	正无穷
Both	negativeInf	任意字符	-Inf	负无穷
Both	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	none	文件压缩格式
Both	dateFormat	任何能转换为 Java 的 SimpleDateFormat 的字符串	yyyy-MM-dd	日期格式
Both	timestampFormat	任何能转换为 Java 的 SimpleDateFormat 的字符串	yyyy-MMdd'T'HH:mm:ss.SSSZZ	时间戳格式
Read	maxColumns	任意整数	20480	声明文件中的最大列数
Read	maxCharsPerColumn	任意整数	1000000	声明一个列中的最大字符数。

读 \写 操作	配置项	可选值	默认值	描述
Read	escapeQuotes	true, false	true	是否应该转义行中的引号。
Read	maxMalformedLogPerPartition	任意整数	10	声明每个分区中最多允许多少条格式错误的数 据，超过这个值后格式错误的数 据将不会被读 取
Write	quoteAll	true, false	false	指定是否应 该将所有值 都括在引号 中，而不只 是转义具有 引号字符的 值。
Read	multiLine	true, false	false	是否允许每 条完整记录 跨域多行

JSON读写可选配置

读\写 操作	配置项	可选值	默认值
Both	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	none
Both	dateFormat	任何能转换为 Java 的 SimpleDateFormat 的字符串	yyyy-MM-dd
Both	timestampFormat	任何能转换为 Java 的 SimpleDateFormat 的字符串	yyyy-MMdd'T'HH:mm:ss.SSSZZ
Read	primitiveAsString	true, false	false
Read	allowComments	true, false	false
Read	allowUnquotedFieldNames	true, false	false
Read	allowSingleQuotes	true, false	true
Read	allowNumericLeadingZeros	true, false	false
Read	allowBackslashEscapingAnyCharacter	true, false	false
Read	columnNameOfCorruptRecord	true, false	Value of spark.sql.column&NameOf
Read	multiLine	true, false	false

数据库读写可选配置

属性名称	含义
url	数据库地址
dbtable	表名称
driver	数据库驱动
partitionColumn, lowerBound, upperBoun	分区总数，上界，下界
numPartitions	可用于表读写并行性的最大分区数。如果要写的分区数量超过这个限制，那么可以调用 <code>coalesce(numpartition)</code> 重置分区数。
fetchsize	每次往返要获取多少行数据。此选项仅适用于读取数据。
batchsize	每次往返插入多少行数据，这个选项只适用于写入数据。默认值是 1000。
isolationLevel	事务隔离级别：可以是 NONE，READ_COMMITTED，READ_UNCOMMITTED，REPEATABLE_READ 或 SERIALIZABLE，即标准事务隔离级别。 默认值是 READ_UNCOMMITTED。这个选项只适用于数据读取。
createTableOptions	写入数据时自定义创建表的相关配置
createTableColumnTypes	写入数据时自定义创建列的列类型

3 SparkSQL常用聚合函数

简单聚合

数据准备

```

1  // 需要导入 spark sql 内置的函数包
2  import org.apache.spark.sql.functions._
3
4  val spark = SparkSession.builder().appName("aggregations").master("local[2]").getOrCreate()
5  val empDF = spark.read.json("/usr/file/json/emp.json")
6  // 注册为临时视图，用于后面演示 SQL 查询
7  empDF.createOrReplaceTempView("emp")
8  empDF.show()

```

count

```

1  // 计算员工人数
2  empDF.select(count("ename")).show()

```

countDistinct

```

1  // 计算姓名不重复的员工人数
2  empDF.select(countDistinct("deptno")).show()

```

approx_count_distinct

通常在使用大型数据集时，你可能关注的只是近似值而不是准确值，这时可以使用 `approx_count_distinct` 函数，并可以使用第二个参数指定最大允许误差。

```
1 empDF.select(approx_count_distinct("ename", 0.1)).show()
```

first & last

获取 DataFrame 中指定列的第一个值或者最后一个值。

```
1 empDF.select(first("ename"), last("job")).show()
```

min & max

获取 DataFrame 中指定列的最小值或者最大值。

```
1 empDF.select(min("sal"), max("sal")).show()
```

sum & sumDistinct

求和以及求指定列所有不相同的值的和。

```
1 empDF.select(sum("sal")).show()
2 empDF.select(sumDistinct("sal")).show()
```

avg

内置的求平均数的函数。

```
1 empDF.select(avg("sal")).show()
```

数学函数

Spark SQL 中还支持多种数学聚合函数，用于通常的数学计算，以下是一些常用的例子：

```
1 // 1. 计算总体方差、均方差、总体标准差、样本标准差
2 empDF.select(var_pop("sal"), var_samp("sal"), stddev_pop("sal"), stddev_samp("sal")).show()
3
4 // 2. 计算偏度和峰度
5 empDF.select(skewness("sal"), kurtosis("sal")).show()
6
7 // 3. 计算两列的皮尔逊相关系数、样本协方差、总体协方差。（这里只是演示，员工编号和薪资两列实际上并没有什么关联关系）
8 empDF.select(corr("empno", "sal"), covar_samp("empno", "sal"), covar_pop("empno", "sal")).show()
```

聚合数据到集合

```
1 scala> empDF.agg(collect_set("job"), collect_list("ename")).show()
2
3 输出：
4 +-----+-----+
5 | collect_set(job) | collect_list(ename) |
6 +-----+-----+
7 | [MANAGER, SALESMA... | [SMITH, ALLEN, WA... |
8 +-----+-----+
```

分组聚合

简单分组

```
1 empDF.groupBy("deptno", "job").count().show()
2 //等价 SQL
3 spark.sql("SELECT deptno, job, count(*) FROM emp GROUP BY deptno, job").show()
4
5 输出:
6 +-----+-----+
7 |deptno|      job|count|
8 +-----+-----+
9 |    10|PRESIDENT|    1|
10 |    30|   CLERK|    1|
11 |    10|  MANAGER|    1|
12 |    30|  MANAGER|    1|
13 |    20|   CLERK|    2|
14 |    30|SALESMAN|    4|
15 |    20|  ANALYST|    2|
16 |    10|   CLERK|    1|
17 |    20|  MANAGER|    1|
18 +-----+-----+
```

分组聚合

```
1 empDF.groupBy("deptno").agg(count("ename").alias("人数"), sum("sal").alias("总工资")).show()
2 // 等价语法
3 empDF.groupBy("deptno").agg("ename"->"count", "sal"->"sum").show()
4 // 等价 SQL
5 spark.sql("SELECT deptno, count(ename) ,sum(sal) FROM emp GROUP BY deptno").show()
6
7 输出:
8 +-----+-----+
9 |deptno|人数|总工资|
10 +-----+-----+
11 |    10|   3|8750.0|
12 |    30|   6|9400.0|
13 |    20|   5|9375.0|
14 +-----+-----+
```

自定义聚合函数

Scala 提供了两种自定义聚合函数的方法，分别如下：

- 有类型的自定义聚合函数，主要适用于 DataSet；
- 无类型的自定义聚合函数，主要适用于 DataFrame。

以下分别使用两种方式来自定义一个求平均值的聚合函数，这里以计算员工平均工资为例。两种自定义方式分别如下：

有类型的自定义函数

```
1 import org.apache.spark.sql.expressions.Aggregator
2 import org.apache.spark.sql.{Encoder, Encoders, SparkSession, functions}
3
4 // 1. 定义员工类, 对于可能存在 null 值的字段需要使用 Option 进行包装
5 case class Emp(ename: String, comm: scala.Option[Double], deptno: Long, empno: Long,
```

```

6             hiredate: String, job: String, mgr: scala.Option[Long], sal: Double)
7
8 // 2. 定义聚合操作的中间输出类型
9 case class SumAndCount(var sum: Double, var count: Long)
10
11 /* 3. 自定义聚合函数
12  * @IN 聚合操作的输入类型
13  * @BUF reduction 操作输出值的类型
14  * @OUT 聚合操作的输出类型
15  */
16 object MyAverage extends Aggregator[Emp, SumAndCount, Double] {
17
18     // 4. 用于聚合操作的初始零值
19     override def zero: SumAndCount = SumAndCount(0, 0)
20
21     // 5. 同一分区中的 reduce 操作
22     override def reduce(avg: SumAndCount, emp: Emp): SumAndCount = {
23         avg.sum += emp.sal
24         avg.count += 1
25         avg
26     }
27
28     // 6. 不同分区中的 merge 操作
29     override def merge(avg1: SumAndCount, avg2: SumAndCount): SumAndCount = {
30         avg1.sum += avg2.sum
31         avg1.count += avg2.count
32         avg1
33     }
34
35     // 7. 定义最终的输出类型
36     override def finish(reduction: SumAndCount): Double = reduction.sum / reduction.count
37
38     // 8. 中间类型的编码转换
39     override def bufferEncoder: Encoder[SumAndCount] = Encoders.product
40
41     // 9. 输出类型的编码转换
42     override def outputEncoder: Encoder[Double] = Encoders.scalaDouble
43 }
44
45 object SparkSqlApp {
46
47     // 测试方法
48     def main(args: Array[String]): Unit = {
49
50         val spark = SparkSession.builder().appName("Spark-
SQL").master("local[2]").getOrCreate()
51         import spark.implicits._
52         val ds = spark.read.json("file/emp.json").as[Emp]
53
54         // 10. 使用内置 avg() 函数和自定义函数分别进行计算，验证自定义函数是否正确
55         val myAvg = ds.select(MyAverage.toColumn.name("average_sal")).first()
56         val avg = ds.select(functions.avg(ds.col("sal"))).first().get(0)
57
58         println("自定义 average 函数 : " + myAvg)
59         println("内置的 average 函数 : " + avg)
60     }
61 }

```

自定义聚合函数需要实现的方法比较多，这里以绘图的方式来演示其执行流程，以及每个方法的作用：

关于 `zero` , `reduce` , `merge` , `finish` 方法的作用在上图都有说明，这里解释一下中间类型和输出类型的编码转换，这个写法比较固定，基本上就是两种情况：

- 自定义类型 Case Class 或者元组就使用 `Encoders.product` 方法；
- 基本类型就使用其对应名称的方法，如 `scalaByte` , `scalaFloat` , `scalaShort` 等，示例如下：

```
1  override def bufferEncoder: Encoder[SumAndCount] = Encoders.product
2  override def outputEncoder: Encoder[Double] = Encoders.scalaDouble
```

无类型的自定义聚合函数

理解了有类型的自定义聚合函数后，无类型的定义方式也基本相同，代码如下：

```
1  import org.apache.spark.sql.expressions.{MutableAggregationBuffer, UserDefinedAggregateFunction}
2  import org.apache.spark.sql.types._
3  import org.apache.spark.sql.{Row, SparkSession}
4
5  object MyAverage extends UserDefinedAggregateFunction {
6      // 1. 聚合操作输入参数的类型, 字段名称可以自定义
7      def inputSchema: StructType = StructType(StructField("MyInputColumn", LongType) :: Nil)
8
9      // 2. 聚合操作中间值的类型, 字段名称可以自定义
10     def bufferSchema: StructType = {
11         StructType(StructField("sum", LongType) :: StructField("MyCount", LongType) :: Nil)
12     }
13
14     // 3. 聚合操作输出参数的类型
15     def dataType: DataType = DoubleType
16
17     // 4. 此函数是否始终在相同输入上返回相同的输出, 通常为 true
18     def deterministic: Boolean = true
19
20     // 5. 定义零值
21     def initialize(buffer: MutableAggregationBuffer): Unit = {
22         buffer(0) = 0L
23         buffer(1) = 0L
24     }
25
26     // 6. 同一分区中的 reduce 操作
27     def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
28         if (!input.isNullAt(0)) {
29             buffer(0) = buffer.getLong(0) + input.getLong(0)
30             buffer(1) = buffer.getLong(1) + 1
31         }
32     }
33
34     // 7. 不同分区中的 merge 操作
35     def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
36         buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)
37         buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
38     }
```



```

39
40     // 8. 计算最终的输出值
41     def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)
42 }
43
44 object SparkSqlApp {
45
46     // 测试方法
47     def main(args: Array[String]): Unit = {
48
49         val spark = SparkSession.builder().appName("Spark-SQL").master("local[2]").getOrCreate()
50         // 9. 注册自定义的聚合函数
51         spark.udf.register("myAverage", MyAverage)
52
53         val df = spark.read.json("file/emp.json")
54         df.createOrReplaceTempView("emp")
55
56         // 10. 使用自定义函数和内置函数分别进行计算
57         val myAvg = spark.sql("SELECT myAverage(sal) as avg_sal FROM emp").first()
58         val avg = spark.sql("SELECT avg(sal) as avg_sal FROM emp").first()
59
60         println("自定义 average 函数 : " + myAvg)
61         println("内置的 average 函数 : " + avg)
62     }
63 }

```

4 SparkSQL联结操作

数据准备

本文主要介绍 Spark SQL 的多表连接，需要预先准备测试数据。分别创建员工和部门的 Dataframe，并注册为临时视图，代码如下：

```

1     val spark = SparkSession.builder().appName("aggregations").master("local[2]").getOrCreate()
2
3     val empDF = spark.read.json("/usr/file/json/emp.json")
4     empDF.createOrReplaceTempView("emp")
5
6     val deptDF = spark.read.json("/usr/file/json/dept.json")
7     deptDF.createOrReplaceTempView("dept")

```

两表的主要字段如下：

```

1     emp 员工表
2     |-- ENAME: 员工姓名
3     |-- DEPTNO: 部门编号
4     |-- EMPNO: 员工编号
5     |-- HIREDATE: 入职时间
6     |-- JOB: 职务
7     |-- MGR: 上级编号
8     |-- SAL: 薪资
9     |-- COMM: 奖金

```

```
1 dept 部门表
2 |-- DEPTNO: 部门编号
3 |-- DNAME: 部门名称
4 |-- LOC: 部门所在城市
```

连接类型

Spark 中支持多种连接类型：

- **Inner Join** : 内连接;
- **Full Outer Join** : 全外连接;
- **Left Outer Join** : 左外连接;
- **Right Outer Join** : 右外连接;
- **Left Semi Join** : 左半连接;
- **Left Anti Join** : 左反连接;
- **Natural Join** : 自然连接;
- **Cross (or Cartesian) Join** : 交叉 (或笛卡尔) 连接。

其中内, 外连接, 笛卡尔积均与普通关系型数据库中的相同, 如下图所示:

这里解释一下左半连接和左反连接, 这两个连接等价于关系型数据库中的 **IN** 和 **NOT IN** 字句:

```
1 -- LEFT SEMI JOIN
2 SELECT * FROM emp LEFT SEMI JOIN dept ON emp.deptno = dept.deptno
3 -- 等价于如下的 IN 语句
4 SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept)
5
6 -- LEFT ANTI JOIN
7 SELECT * FROM emp LEFT ANTI JOIN dept ON emp.deptno = dept.deptno
8 -- 等价于如下的 IN 语句
9 SELECT * FROM emp WHERE deptno NOT IN (SELECT deptno FROM dept)
```

所有连接类型的示例代码如下:

INNER JOIN

```
1 // 1. 定义连接表达式
2 val joinExpression = empDF.col("deptno") === deptDF.col("deptno")
3 // 2. 连接查询
4 empDF.join(deptDF, joinExpression).select("ename", "dname").show()
5
6 // 等价 SQL 如下:
7 spark.sql("SELECT ename, dname FROM emp JOIN dept ON emp.deptno = dept.deptno").show()
```

FULL OUTER JOIN

```
1 empDF.join(deptDF, joinExpression, "outer").show()
2 spark.sql("SELECT * FROM emp FULL OUTER JOIN dept ON emp.deptno = dept.deptno").show()
```

LEFT OUTER JOIN

```
1 empDF.join(deptDF, joinExpression, "left_outer").show()
2 spark.sql("SELECT * FROM emp LEFT OUTER JOIN dept ON emp.deptno = dept.deptno").show()
```

RIGHT OUTER JOIN

```
1 empDF.join(deptDF, joinExpression, "right_outer").show()
2 spark.sql("SELECT * FROM emp RIGHT OUTER JOIN dept ON emp.deptno = dept.deptno").show()
```

LEFT SEMI JOIN

```
1 empDF.join(deptDF, joinExpression, "left_semi").show()
2 spark.sql("SELECT * FROM emp LEFT SEMI JOIN dept ON emp.deptno = dept.deptno").show()
```

LEFT ANTI JOIN

```
1 empDF.join(deptDF, joinExpression, "left_anti").show()
2 spark.sql("SELECT * FROM emp LEFT ANTI JOIN dept ON emp.deptno = dept.deptno").show()
```

CROSS JOIN

```
1 empDF.join(deptDF, joinExpression, "cross").show()
2 spark.sql("SELECT * FROM emp CROSS JOIN dept ON emp.deptno = dept.deptno").show()
```

NATURAL JOIN

自然连接是在两张表中寻找那些数据类型和列名都相同的字段，然后自动地将他们连接起来，并返回所有符合条件的结果。

```
1 spark.sql("SELECT * FROM emp NATURAL JOIN dept").show()
```

以下是一个自然连接的查询结果，程序自动推断出使用两张表都存在的 dept 列进行连接，其实际等价于：

```
1 spark.sql("SELECT * FROM emp JOIN dept ON emp.deptno = dept.deptno").show()
```

由于自然连接常常会产生不可预期的结果，所以并不推荐使用。

连接的执行

在对大表与大表之间进行连接操作时，通常都会触发 **Shuffle Join**，两表的所有分区节点会进行 **All-to-All** 的通讯，这种查询通常比较昂贵，会对网络 IO 会造成比较大的负担。

而对于大表和小表的连接操作，Spark 会在一定程度上进行优化，如果小表的数据量小于 Worker Node 的内存空间，Spark 会考虑将小表的数据广播到每一个 Worker Node，在每个工作节点内部执行连接计算，这可以降低网络的 IO，但会加大每个 Worker Node 的 CPU 负担。

是否采用广播方式进行 **Join** 取决于程序内部对小表的判断，如果想明确使用广播方式进行 **Join**，则可以在 DataFrame API 中使用 **broadcast** 方法指定需要广播的小表：

```
1 empDF.join(broadcast(deptDF), joinExpression).show()
```

5 Structured API

创建DataFrame和Dataset

创建DataFrame

Spark 中所有功能的入口点是 `SparkSession`，可以使用 `SparkSession.builder()` 创建。创建后应用程序就可以从现有 RDD，Hive 表或 Spark 数据源创建 DataFrame。示例如下：

```
1  val spark = SparkSession.builder().appName("Spark-SQL").master("local[2]").getOrCreate()
2  val df = spark.read.json("/usr/file/json/emp.json")
3  df.show()
4
5  // 建议在进行 spark SQL 编程前导入下面的隐式转换，因为 DataFrames 和 datasets 中很多操作都依赖了隐
   式转换
6  import spark.implicits._
```

创建Dataset

Spark 支持由内部数据集和外部数据集来创建 DataSet，其创建方式分别如下：

由外部数据集创建

```
1  // 1. 需要导入隐式转换
2  import spark.implicits._
3
4  // 2. 创建 case class, 等价于 Java Bean
5  case class Emp(ename: String, comm: Double, deptno: Long, empno: Long,
6                hiredate: String, job: String, mgr: Long, sal: Double)
7
8  // 3. 由外部数据集创建 Datasets
9  val ds = spark.read.json("/usr/file/emp.json").as[Emp]
10 ds.show()
```

由内部数据集创建

```
1  // 1. 需要导入隐式转换
2  import spark.implicits._
3
4  // 2. 创建 case class, 等价于 Java Bean
5  case class Emp(ename: String, comm: Double, deptno: Long, empno: Long,
6                hiredate: String, job: String, mgr: Long, sal: Double)
7
8  // 3. 由内部数据集创建 Datasets
9  val caseClassDS = Seq(Emp("ALLEN", 300.0, 30, 7499, "1981-02-20 00:00:00", "SALESMAN", 7698,
10                        1600.0),
11                        Emp("JONES", 300.0, 30, 7499, "1981-02-20 00:00:00", "SALESMAN",
12                        7698, 1600.0))
13
14                        .toDS()
15 caseClassDS.show()
```

由RDD创建DataFrame

Spark 支持两种方式把 RDD 转换为 DataFrame，分别是使用反射推断和指定 Schema 转换：

使用反射推断

```
1 // 1. 导入隐式转换
2 import spark.implicits._
3
4 // 2. 创建部门类
5 case class Dept(deptno: Long, dname: String, loc: String)
6
7 // 3. 创建 RDD 并转换为 dataSet
8 val rddToDS = spark.sparkContext
9   .textFile("/usr/file/dept.txt")
10  .map(_ .split("\t"))
11  .map(line => Dept(line(0).trim.toLong, line(1), line(2)))
12  .toDS() // 如果调用 toDF() 则转换为 dataframe
```

以编程方式指定Schema

```
1 import org.apache.spark.sql.Row
2 import org.apache.spark.sql.types._
3
4
5 // 1. 定义每个列的列类型
6 val fields = Array(StructField("deptno", LongType, nullable = true),
7                      StructField("dname", StringType, nullable = true),
8                      StructField("loc", StringType, nullable = true))
9
10 // 2. 创建 schema
11 val schema = StructType(fields)
12
13 // 3. 创建 RDD
14 val deptRDD = spark.sparkContext.textFile("/usr/file/dept.txt")
15 val rowRDD = deptRDD.map(_ .split("\t")).map(line => Row(line(0).toLong, line(1), line(2)))
16
17
18 // 4. 将 RDD 转换为 dataframe
19 val deptDF = spark.createDataFrame(rowRDD, schema)
20 deptDF.show()
```

DataFrames与Datasets互相转换

Spark 提供了非常简单的转换方法用于 DataFrame 与 Dataset 间的互相转换，示例如下：

```
1 # DataFrames转Datasets
2 scala> df.as[Emp]
3 res1: org.apache.spark.sql.Dataset[Emp] = [COMM: double, DEPTNO: bigint ... 6 more fields]
4
5 # Datasets转DataFrames
6 scala> ds.toDF()
7 res2: org.apache.spark.sql.DataFrame = [COMM: double, DEPTNO: bigint ... 6 more fields]
```

Columns列操作

引用列

Spark 支持多种方法来构造和引用列，最简单的是使用 `col()` 或 `column()` 函数。

```
1 col("colName")
2 column("colName")
3
4 // 对于 Scala 语言而言，还可以使用"$myColumn"和`myColumn`这两种语法糖进行引用。
5 df.select($"ename", $"job").show()
6 df.select(`ename`, `job`).show()
```

新增列

```
1 // 基于已有列值新增列
2 df.withColumn("upSal", $"sal"+1000)
3 // 基于固定值新增列
4 df.withColumn("intCol", lit(1000))
```

删除列

```
1 // 支持删除多个列
2 df.drop("comm", "job").show()
```

重命名列

```
1 df.withColumnRenamed("comm", "common").show()
```

需要说明的是新增，删除，重命名列都会产生新的 DataFrame，原来的 DataFrame 不会被改变。

使用Structured API进行基本查询

```
1 // 1. 查询员工姓名及工作
2 df.select($"ename", $"job").show()
3
4 // 2. filter 查询工资大于 2000 的员工信息
5 df.filter($"sal" > 2000).show()
6
7 // 3. orderBy 按照部门编号降序，工资升序进行查询
8 df.orderBy(desc("deptno"), asc("sal")).show()
9
10 // 4. limit 查询工资最高的 3 名员工的信息
11 df.orderBy(desc("sal")).limit(3).show()
12
13 // 5. distinct 查询所有部门编号
14 df.select("deptno").distinct().show()
15
16 // 6. groupBy 分组统计部门人数
17 df.groupBy("deptno").count().show()
```

使用Spark SQL进行基本查询

Spark SQL基本使用

```
1 // 1. 首先需要将 DataFrame 注册为临时视图
2 df.createOrReplaceTempView("emp")
3
4 // 2. 查询员工姓名及工作
5 spark.sql("SELECT ename, job FROM emp").show()
6
7 // 3. 查询工资大于 2000 的员工信息
8 spark.sql("SELECT * FROM emp where sal > 2000").show()
9
10 // 4. orderBy 按照部门编号降序, 工资升序进行查询
11 spark.sql("SELECT * FROM emp ORDER BY deptno DESC, sal ASC").show()
12
13 // 5. limit 查询工资最高的 3 名员工的信息
14 spark.sql("SELECT * FROM emp ORDER BY sal DESC LIMIT 3").show()
15
16 // 6. distinct 查询所有部门编号
17 spark.sql("SELECT DISTINCT(deptno) FROM emp").show()
18
19 // 7. 分组统计部门人数
20 spark.sql("SELECT deptno, count(ename) FROM emp group by deptno").show()
```

全局临时视图

上面使用 `createOrReplaceTempView` 创建的是会话临时视图, 它的生命周期仅限于会话范围, 会随会话的结束而结束。

你也可以使用 `createGlobalTempView` 创建全局临时视图, 全局临时视图可以在所有会话之间共享, 并直到整个 Spark 应用程序终止后才会消失。全局临时视图被定义在内置的 `global_temp` 数据库下, 需要使用限定名称进行引用, 如 `SELECT * FROM global_temp.view1`。

```
1 // 注册为全局临时视图
2 df.createGlobalTempView("gemp")
3
4 // 使用限定名称进行引用
5 spark.sql("SELECT ename, job FROM global_temp.gemp").show()
```