RDD api 整理

RDD[T]

Transformations

rdd api	备注
persist/cache	
map(f: T => U)	
keyBy(f: T => K)	特殊的 map，提 key
flatMap(f: T => Iterable[U])	map 的一种，类似 UDTF
filter(f: T => Boolean)	map 的一种
distinct(numPartitions)	rdd 的实现为 map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1) reduceByKey 是特殊的 combineByKey，其 mergeValue 函数和 mergeCombiners 函数一致，都是(x, y) => x
repartition(numPartitions)/coalesce(numPartitions)	repartition 用于增减 rdd 分区。coalesce 特指减少分区，可以通过一次窄依赖的映射避免 shuffle
sample()/randomSplit()/takeSample()	采样
union(RDD[T])	不去重。使用 distinct()去重
sortBy[K](f: (T) => K)	传入的 f 是提 key 函数，rdd 的实现为 keyBy(f).sortByKey().values() 这次操作为 RDD 设置了一个 RangePartitioner
intersection(RDD[T])	两个集合取交集，并去重。RDD 的实现为 map(v

rdd api	备注
	=> (v, null)).cogroup(other.map(v => (v, null))).filter(两边都空).keys()　cogroup 是生成 K, List[V], List[V]的形态，这个过程可能内含一次 shuffle 操作，为了两边 RDD 的分区对齐。
glom():RDD[Array[T]]	把每个分区的数据合并成一个 Array。原本每个分区是 T 的迭代器。
cartesian(RDD[U]): RDD[(T, U)]	求两个集合的笛卡尔积。RDD 的做法是两个 RDD 内循环、外循环 yield 出每对(x, y)
groupBy[K](f: T => K): RDD[(K, Iterable[T])]	RDD 建议如果后续跟 agg 的话，直接使用 aggregateByKey 或 reduceByKey 更省时，这两个操作本质上就是 combineByKey
pipe(command: String)	把 RDD 数据通过 ProcessBuilder 创建额外的进程输出走
mapPartitions(f: Iterator[T] => Iterator[U])/mapPartitionsWithIndex(f: (Int, Iterator[T]) => Iterator[U])	RDD 的每个分区做 map 变换
zip(RDD[U]): RDD[(T, U)]	两个 RDD 分区数目一致，且每个分区数据条数一致

Actions

rdd api	备注
foreach(f: T => Unit)	rdd 实现为调用 sc.runJob()，把 f 作用于每个分区的每条记录
foreachPartition(f: Iterator[T] => Unit)	rdd 实现为调用 sc.runJob()，把 f 作用于每个分区
collect(): Array[T]	rdd 实现为调用 sc.runJob()，得到 results，把多个 result 的 array 合并成一个 array

rdd api	备注
toLocalIterator()	把所有数据以迭代器返回，rdd 实现是调用 sc.runJob()，每个分区迭代器转 array，收集到 driver 端再 flatMap 一次打散成大迭代器。理解为一种比较特殊的 driver 端 cache
collect[U](f: PartailFunction[T, U]): RDD[U]	rdd 实现为 filter(f.isDefinedAt).map(f) 先做一次 filter 找出满足的数据，然后一次 map 操作执行这个偏函数。
subtract(RDD[T])	rdd 实现为 map(x => (x, null)).subtractByKey(other.map((_, null)), p2).keys 与求交类似
reduce(f: (T, T) => T)	rdd 实现为调用 sc.runJob()，让 f 在 rdd 每个分区计算一次，最后汇总 merge 的时候再计算一次。
treeReduce(f: (T, T) => T, depth = 2)	见 treeAggregate
fold(zeroValue: T)(op: (T, T) => T)	特殊的 reduce，带初始值，函数式语义的 fold
aggregate(zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U)	带初始值、reduce 聚合、merge 聚合三个完整条件的聚合方法。rdd 的做法是把函数传入分区里去做计算，最后汇总各分区的结果再一次 combOp 计算。
treeAggregate(zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U)(depth = 2)	在分区处，做两次及以上的 merge 聚合，即每个分区的 merge 计算可能也会带 shuffle。其余部分同 aggregate。理解为更复杂的多阶 aggregate
count()	rdd 实现为调用 sc.runJob()，把每个分区的 size 汇总在 driver 端再 sum 一次
countApprox(timeout, confidence)	提交个体 DAGScheduler 特殊的任务，生成特殊的任务监听者，在 timeout 时间内返回，没计算完的话返回一个大致结果，返回值的计算逻辑可见 ApproximateEvaluator 的子类
countByValue(): Map[T, Long]	rdd 实现为 map(value => (value, null)).countByKey() 本质上是一次简单的 combineByKey，返回 Map，会全 load 进 driver 的内存里，需要数据集规模较小

rdd api	备注
countByValueApprox()	同 countApprox()
countApproxDistinct()	实验性方法，用 streamlib 库实现的 HyperLogLog 做
zipWithIndex(): RDD[(T, Long)]/zipWithUniqueId(): RDD[(T, Long)]	与生成的 index 做 zip 操作
take(num): Array[T]	扫某个分区
first()	即 take(1)
top(n)(ordering)	每个分区内传入 top 的处理函数，得到分区的堆，使用 rdd.reduce()，把每个分区的堆合起来，排序，取前 n 个
max()/min()	特殊的 reduce，传入 max/min 比较函数
saveAsXXXXX	输出存储介质
checkpoint	显示 cp 声明

特殊 RDD

PairRDDFunctions

rdd api	备注
combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C): RDD[(K, C)]	传统 MR 定义拆分，重要基础 api
aggregateByKey[U](zeroValue: U, seqOp: (U, V) => U, combOp: (U, U) => U): RDD[(K, U)]	rdd 里，把 zeroValue 转成了一个 createCombiner 方法，然后调用了 combineByKey()。本质上两者是一样的。
foldByKey(zeroValue: V, func: (V, V) => V): RDD[(K, V)]	func 即被当作 mergeValue，又被当作 mergeCombiners，调用了 combineByKey()
sampleByKey()	生成一个与 key 相关的 sampleFunc，调用 rdd.mapPartitionsWithIndex(sampleFunc)

rdd api	备注
reduceByKey()	调用 combineByKey
reduceByKeyLocally(func: (V, V) => V): Map[K, V]	rdd 实现为 self.mapPartitions(reducePartition).reduce(mergeMaps) reducePartition 是在每个分区生成一个 HashMap，mergeMaps 是合并多个 HashMap
countByKey()	rdd 实现为 mapValues(_ => 1L).reduceByKey(_ + _).collect().toMap
countByKeyApprox()	rdd 实现为 map(_._1).countByValueApprox
countApproxDistinctByKey()	类似 rdd 的 countApproxDistinct 方法，区别是把方法作用在了 combineByKey 里面
groupByKey()	简单的 combineByKey 实现
partitionBy(partitioner)	为 rdd 设置新的分区结构
join(RDD[(K, W)]): RDD[(K, (V, W))]	rdd 实现为 cogroup(other, partitioner).flatMapValues(…)
leftOuterJoin(…)	实现同上，只是 flatMapValues 里面遍历两个 rdd，yield 出结果的判断逻辑变了下
rightOuterJoin(…)	同上
fullOuterJoin(…)	同上
collectAsMap()	rdd 实现为 collect().foreach(pairToMap)
mapValues(f: V => U)	一种简单的 map()操作
flatMapValues(f: V => Iterable[U])	一种简单的 map()操作
cogroup(RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]	做集合性操作的基础 api，包括各种 join、求交等

rdd api	备注
subtractByKey(RDD[(K, W)]): RDD[(K, V)]	从原来的 rdd 里排除右侧有的 keys
lookup(key: K): Seq[V]	rdd 实现的时候，然后分区是基于 key 的，那比较高效可以直接遍历对应分区，否则全部遍历。全部遍历的实现为 *filter(_._1 == key).map(_._2).collect()*
saveAsXXX	写外部存储
keys()	一种简单的 map()操作
values()	一种简单的 map()操作

AsyncRDDActions

countAsync, collectAsync, takeAsync, foreachAsync, foreachPartitionAsync

OrderedRDDFunctions

针对 RDD[K: Ordering, V]

rdd api	备注
sortByKey()	见 rdd.sortBy()里的解释
filterByRange(lower: K, upper: K)	当 rdd 分区是 RangePartition 的时候可以做这样的 filter

DoubleRDDFunctions

针对 RDD[Double]

rdd api	备注
sum()	rdd 实现是 *reduce(_ + _)*

rdd api	备注
stats()	rdd 实现是 mapPartitions(nums => Iterator(StatCounter(nums))).reduce((a, b) => a.merge(b)) StatCounter 在一次遍历里统计出中位数、方差、count 三个值，merge()是他内部的方法
mean()	rdd 实现是 stats().mean
variance()/sampleVariance()	rdd 实现是 stats().variance
stdev()/sampleStdev()	rdd 实现是 stats().stdev 求标准差
meanApprox()/sumApprox()	调用 runApproximateJob
histogram()	比较复杂的计算，rdd 实现是先 mapPartitions 再 reduce，包含几次递归