

14 | 弹性分布式数据集：Spark大厦的地基（下）

2019-05-17 蔡元楠 来自北京

《大规模数据处理实战》



你好，我是蔡元楠。

上一讲我们介绍了弹性分布式数据集（RDD）的定义、特性以及结构，并且深入讨论了依赖关系（Dependencies）。

今天让我们一起来继续学习 RDD 的其他特性。

RDD 的结构

首先，我来介绍一下 RDD 结构中其他的几个知识点：检查点（Checkpoint）、存储级别（Storage Level）和迭代函数（Iterator）。

RDD

SparkContext

SparkConf

Partitioner

Dependencies

Checkpoint

Preferred location

Storage Level

Iterator

P
A
R
T
I
T
I
O
N
S

通过上一讲，你应该已经知道了，基于 RDD 的依赖关系，如果任意一个 RDD 在相应的节点丢失，你只需要从上一步的 RDD 出发再次计算，便可恢复该 RDD。

但是，如果一个 RDD 的依赖链比较长，而且中间又有多个 RDD 出现故障的话，进行恢复可能会非常耗费时间和计算资源。

而检查点（Checkpoint）的引入，就是为了优化这些情况下的数据恢复。

很多数据库系统都有检查点机制，在连续的 transaction 列表中记录某几个 transaction 后数据的内容，从而加快错误恢复。

RDD 中的检查点的思想与之类似。

在计算过程中，对于一些计算过程比较耗时的 RDD，我们可以将它缓存至硬盘或 HDFS 中，标记这个 RDD 有被检查点处理过，并且清空它的所有依赖关系。同时，给它新建一个依赖于 CheckpointRDD 的依赖关系，CheckpointRDD 可以用来从硬盘中读取 RDD 和生成新的分区信息。

这样，当某个子 RDD 需要错误恢复时，回溯至该 RDD，发现它被检查点记录过，就可以直接去硬盘中读取这个 RDD，而无需再向前回溯计算。

存储级别（Storage Level）是一个枚举类型，用来记录 RDD 持久化时的存储级别，常用的有以下几个：

MEMORY_ONLY：只缓存在内存中，如果内存空间不够则不缓存多出来的部分。这是 RDD 存储级别的默认值。

MEMORY_AND_DISK：缓存在内存中，如果空间不够则缓存在硬盘中。

DISK_ONLY：只缓存在硬盘中。

MEMORY_ONLY_2 和 MEMORY_AND_DISK_2 等：与上面的级别功能相同，只不过每个分区在集群中两个节点上建立副本。

这就是我们在前文提到过的，Spark 相比于 Hadoop 在性能上的提升。我们可以随时把计算好的 RDD 缓存在内存中，以便下次计算时使用，这大幅度减小了硬盘读写的开销。

迭代函数 (Iterator) 和计算函数 (Compute) 是用来表示 RDD 怎样通过父 RDD 计算得到的。

迭代函数会首先判断缓存中是否有想要计算的 RDD，如果有就直接读取，如果没有，就查找想要计算的 RDD 是否被检查点处理过。如果有，就直接读取，如果没有，就调用计算函数向上递归，查找父 RDD 进行计算。

到现在，相信你已经对弹性分布式数据集的基本结构有了初步了解。但是光理解 RDD 的结构是远远不够的，我们的终极目标是使用 RDD 进行数据处理。

要使用 RDD 进行数据处理，你需要先了解一些 RDD 的数据操作。

在🔗第 12 讲中，我曾经提过，相比起 MapReduce 只支持两种数据操作，Spark 支持大量的基本操作，从而减轻了程序员的负担。

接下来，让我们进一步了解基于 RDD 的各种数据操作。

RDD 的转换操作

RDD 的数据操作分为两种：转换 (Transformation) 和动作 (Action)。

顾名思义，转换是用来把一个 RDD 转换成另一个 RDD，而动作则是通过计算返回一个结果。


不难想到，之前举例的 map、filter、groupByKey 等都属于转换操作。

Map

map 是最基本的转换操作。

与 MapReduce 中的 map 一样，它把一个 RDD 中的所有数据通过一个函数，映射成一个新的 RDD，任何原 RDD 中的元素在新 RDD 中都有且只有一个元素与之对应。


在这一讲中提到的所有的操作，我都会使用代码举例，帮助你更好地理解。

 复制代码

```
1 rdd = sc.parallelize(["b", "a", "c"])
2 rdd2 = rdd.map(lambda x: (x, 1)) // [('b', 1), ('a', 1), ('c', 1)]
```

Filter


filter 这个操作，是选择原 RDD 里所有数据中满足某个特定条件的数据，去返回一个新的 RDD。如下例所示，通过 filter，只选出了所有的偶数。

 复制代码

```
1 rdd = sc.parallelize([1, 2, 3, 4, 5])
2 rdd2 = rdd.filter(lambda x: x % 2 == 0) // [2, 4]
```

mapPartitions

mapPartitions 是 map 的变种。不同于 map 的输入函数是应用于 RDD 中每个元素，mapPartitions 的输入函数是应用于 RDD 的每个分区，也就是把每个分区中的内容作为整体来处理的，所以输入函数的类型是 `Iterator[T] => Iterator[U]`。


 复制代码

```
1 rdd = sc.parallelize([1, 2, 3, 4], 2)
2 def f(iterator): yield sum(iterator)
3 rdd2 = rdd.mapPartitions(f) // [3, 7]
```

在 mapPartitions 的例子中，我们首先创建了一个有两个分区的 RDD。mapPartitions 的输入函数是对每个分区内的元素求和，所以返回的 RDD 包含两个元素：1+2=3 和 3+4=7。

groupByKey

groupByKey 和 SQL 中的 groupBy 类似，是把对象的集合按某个 Key 来归类，返回的 RDD 中每个 Key 对应一个序列。

 复制代码

```
1 rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 2)])
2 rdd.groupByKey().collect()
3 //"a" [1, 2]
4 //"b" [1]
```


在此，我们只列举这几个常用的、有代表性的操作，对其他转换操作感兴趣的同学可以去自行查阅官方的 API 文档。

RDD 的动作操作

让我们再来看几个常用的动作操作。

Collect

RDD 中的动作操作 collect 与函数式编程中的 collect 类似，它会以数组的形式，返回 RDD 的所有元素。需要注意的是，collect 操作只有在输出数组所含的数据数量较小时使用，因为所有的数据都会载入到程序的内存中，如果数据量较大，会占用大量 JVM 内存，导致内存溢出。

 复制代码

```
1 rdd = sc.parallelize(["b", "a", "c"])
2 rdd.map(lambda x: (x, 1)).collect() // [('b', 1), ('a', 1), ('c', 1)]
```

实际上，上述转换操作中所有的例子，最后都需要将 RDD 的元素 collect 成数组才能得到标记好的输出。

Reduce

与 MapReduce 中的 reduce 类似，它会把 RDD 中的元素根据一个输入函数聚合起来。

```
1 from operator import add
2 sc.parallelize([1, 2, 3, 4, 5]).reduce(add) // 15
```

Count

Count 会返回 RDD 中元素的个数。

```
sc.parallelize([2, 3, 4]).count() // 3
```

CountByKey

仅适用于 Key-Value pair 类型的 RDD，返回具有每个 key 的计数的 <Key, Count> 的 map。

```
1 rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
2 sorted(rdd.countByKey().items()) // [('a', 2), ('b', 1)]
```

讲到这，你可能会问了，为什么要区分转换和动作呢？虽然转换是生成新的 RDD，动作是把 RDD 进行计算生成一个结果，它们本质上不都是计算吗？

这是因为，所有转换操作都很懒，它只是生成新的 RDD，并且记录依赖关系。

但是 Spark 并不会立刻计算出新 RDD 中各个分区的数值。直到遇到一个动作时，数据才会被计算，并且输出结果给 Driver。

比如，在之前的例子中，你先对 RDD 进行 map 转换，再进行 collect 动作，这时 map 后生成的 RDD 不会立即被计算。只有当执行到 collect 操作时，map 才会被计算。而且，map 之后得到的较大的数据量并不会传给 Driver，只有 collect 动作的结果才会传递给 Driver。

这种惰性求值的设计优势是什么呢？让我们来看这样一个例子。

假设，你要从一个很大的文本文件中筛选出包含某个词语的行，然后返回第一个这样的文本行。你需要先读取文件 `textFile()` 生成 `rdd1`，然后使用 `filter()` 方法生成 `rdd2`，最后是行动操作 `first()`，返回第一个元素。

读取文件的时候会把所有的行都存储起来，但我们马上就要筛选出只具有特定词组的行了，等筛选出来之后又要求只输出第一个。这样是不是太浪费存储空间了呢？确实。

所以实际上，Spark 是在行动操作 `first()` 的时候开始真正的运算：只扫描第一个匹配的行，不需要读取整个文件。所以，惰性求值的设计可以让 Spark 的运算更加高效和快速。

让我们总结一下 Spark 执行操作的流程吧。

Spark 在每次转换操作的时候，使用了新产生的 RDD 来记录计算逻辑，这样就把作用在 RDD 上的所有计算逻辑串起来，形成了一个链条。当对 RDD 进行动作时，Spark 会从计算链的最后一个 RDD 开始，依次从上一个 RDD 获取数据并执行计算逻辑，最后输出结果。

RDD 的持久化（缓存）

每当我们对 RDD 调用一个新的 action 操作时，整个 RDD 都会从头开始运算。因此，如果某个 RDD 会被反复重用的话，每次都从头计算非常低效，我们应该对多次使用的 RDD 进行一个持久化操作。

Spark 的 `persist()` 和 `cache()` 方法支持将 RDD 的数据缓存至内存或硬盘中，这样当下次对同一 RDD 进行 Action 操作时，可以直接读取 RDD 的结果，大幅提高了 Spark 的计算效率。

 复制代码

```
1 rdd = sc.parallelize([1, 2, 3, 4, 5])
2 rdd1 = rdd.map(lambda x: x+5)
3 rdd2 = rdd1.filter(lambda x: x % 2 == 0)
4 rdd2.persist()
5 count = rdd2.count() // 3
6 first = rdd2.first() // 6
7 rdd2.unpersist()
```


在文中的代码例子中你可以看到，我们对 RDD2 进行了多个不同的 action 操作。由于在第四行我把 RDD2 的结果缓存在内存中，所以 Spark 无需从一开始的 rdd 开始算起了（持久化处理过的 RDD 只有第一次有 action 操作时才会从源头计算，之后就把结果存储下来，所以在这个例子中，count 需要从源头开始计算，而 first 不需要）。

在缓存 RDD 的时候，它所有的依赖关系也会被一并存下来。所以持久化的 RDD 有自动的容错机制。如果 RDD 的任一分区丢失了，通过使用原先创建它的转换操作，它将会被自动重算。

持久化可以选择不同的存储级别。正如我们讲 RDD 的结构时提到的一样，有 MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY 等。cache() 方法会默认取 MEMORY_ONLY 这一级别。

小结

Spark 在每次转换操作的时候使用了新产生的 RDD 来记录计算逻辑，这样就把作用在 RDD 上的所有计算逻辑串起来形成了一个链条，但是并不会真的去计算结果。当对 RDD 进行动作 Action 时，Spark 会从计算链的最后一个 RDD 开始，利用迭代函数 (Iterator) 和计算函数 (Compute)，依次从上一个 RDD 获取数据并执行计算逻辑，最后输出结果。

此外，我们可以通过将一些需要复杂计算和经常调用的 RDD 进行持久化处理，从而提升计算效率。

思考题

对 RDD 进行持久化操作和记录 Checkpoint，有什么区别呢？

欢迎你将对弹性分布式数据集的疑问写在留言区，与我和其他同学一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。

精选留言 (32)



锦

2019-05-17

区别在于Checkpoint会清空该RDD的依赖关系，并新建一个CheckpointRDD依赖关系，让该RDD依赖，并保存在磁盘或HDFS文件系统中，当数据恢复时，可通过CheckpointRDD读取RDD进行数据计算；持久化RDD会保存依赖关系和计算结果至内存中，可用于后续计算。

作者回复: 👍

共 4 条评论 >

👍 62



RocWay

2019-05-17

主要区别应该是对依赖链的处理：

checkpoint在action之后执行，相当于事务完成后备份结果。既然结果有了，之前的计算过程，也就是RDD的依赖链，也就不需要了，所以不必保存。

但是cache和persist只是保存当前RDD，并不要求是在action之后调用。相当于事务的计算过程，还没有结果。既然没有结果，当需要恢复、重新计算时就要重放计算过程，自然之前的依赖链不能放弃，也需要保存下来。需要恢复时就要从最初的或最近的checkpoint开始重新计算。

作者回复: 这位同学的理解是很准确的



👍 27



涵

2019-05-17

从目的上来说，checkpoint用于数据恢复，RDD持久化用于RDD的多次计算操作的性能优化，避免重复计算。从存储位置上看checkpoint储存在外存中，RDD可以根据存储级别存储在内存或/和外存中。



👍 14



坤剑

2019-06-11

- 1.设置checkpoint时需要指定checkpoint的存储目录，而持久化不管是直接调用cache还是通过persist指定缓存级别都不需要指定存储目录，由系统自己指定
- 2.checkpoint是将RDD去除依赖关系后将数据直接存储到磁盘，且一般是HDFS，带有备份，因

此不容易丢失，恢复时直接获取checkpoint的数据；而持久化一般是直接cache到内存。数据容易丢失，即便是通过设置MEMORY_AND_DISK_2等缓存级别达到内存和磁盘都有备份，也会在每个备份中都缓存RDD的依赖关系，造成不必要的冗余



7



hua168

2019-05-17

老师，我想问下，如果是linux 命令分析单机300G log日志，内存只有16G，怎搞？

如果用spark思想，，从io读很卡，直接内存爆了。

如果先分割日志为100份，再用shell，一下10个并发执行，最后结果合并。感觉还是有点慢。

作者回复: 如果限定为单机处理，我觉得你的第二个思路是可行的，第一个行不通。

共 2 条评论 >



7



JohnT3e

2019-05-17

两者区别在于依赖关系是否保留吧。checkpoint的话，检查点之前的关系应该丢失了，但其数据已经持久化了；而persist或者cache保留了这个依赖关系，如果缓存结果有丢失，可以通过这个关系进行rebuild。

作者回复: 这位同学的理解很准确

共 2 条评论 >



5



挖矿的小戈

2019-05-17

1. 前者：persist或者cache除了持久化该RDD外，还会保留该RDD前面的依赖关系

2. 后者：将该RDD保存到磁盘上，并清除前面的依赖关系

感觉后者的开销会大很多

作者回复: 理解的很对



4



miwucc

2019-05-17

手动调用缓存函数和checkpoint本质上是一样的吧。就是一个手动控制落盘时间，一个自动控

制。

作者回复: 并不是, checkpoint会将一些RDD的结果存入硬盘, 但是不会保留依赖关系; 缓存函数或者持久化处理会保留依赖关系, 所以错误恢复会更方便。



👍 3



廖师虎

2019-05-17

记不太清楚了, checkpoint清除血缘关系, 一般保存在类hdfs文件系统, 目的是容错, 缓存是保留血缘关系, 并保存在本机, 的目的是提高效率, High performance Spark书讲得很详细。

第一次遇到把driver翻译成驱动程序的, 个人感觉还是保留Driver, Action为佳。

作者回复: 翻译也是为了方便英文不好的同学理解, 但是每个名次第一次出现我都会标出英文。



👍 3



Peter

2019-05-18

在计算过程中, 对于一些计算过程比较耗时的 RDD, 我们可以将它缓存至硬盘或 HDFS 中, 标记这个 RDD 有被检查点处理过, 并且清空它的所有依赖关系。同时, 给它新建一个依赖于 CheckpointRDD 的依赖关系, CheckpointRDD 可以用来从硬盘中读取 RDD 和生成新的分区信息。



👍 2



Steven

2019-05-17

缓存了之后, 第一个action还是需要从头计算的吧? "所以无论是 count 还是 first, Spark 都无需从头计算", 这句话是不是有误?

作者回复: 你的观察很仔细, 这里确实是笔误。持久化处理过的RDD只有第一次有action操作时才会从源头计算, 之后就把结果存储下来。所以在这个例子中Count需要从源头开始计算, 而first不需要。

共 3 条评论 >

👍 2



cricket1981

2019-05-17

RDD的checkpoint会导致写入可靠存储的开销。这可能导致RDD被checkpoint的那些批次的处理时间增加。相反，checkpoint太过不频繁会导致血统链增长和任务大小增加。请问该如何设置合理的checkpoint时间间隔呢？



👍 2



jon

2019-05-17

checkpoint不会存储该rdd前面的依赖关系，它后面的rdd都依赖于它。
persist、cache操作会存储依赖关系，当一个分区丢失后可以根据依赖重新计算。

作者回复: 👍



👍 2



cricket1981

2019-05-17

终于明白spark惰性求值的原理了。我理解对 RDD 进行持久化操作和记录 Checkpoint的区别是：前者是开发人员为了避免重复计算、减少长链路计算时间而主动去缓存中间结果，而后者是spark框架为了容错而提供的保存中间结果机制，它对开发人员是透明的，无感知的。

作者回复: 这些机制对开发者并不是透明的，开发者可以手动调用checkpoint和cache方法来存储RDD。他们的主要区别是是否存储依赖关系。



👍 2



茂杨

2020-09-12

Spark的计算是按照Action为单位的，多次的转换在一起只为了组成一个公式，只有真正把数据赋值在公式中并写上等号了才去执行(Action)



👍 1



夜吾夜

2019-09-25

我是否可以这样理解，使用checkpoint，而不是用持久化的RDD来进行数据恢复，是因为当从某一个节点进行回放时，checkpoint的路径比持久化RDD短，更能节省时间，但spark的这种机制也决定了它不支持确定性计算。



👍 1



refactor

2019-07-22

cache 和 checkpoint 区别：1.产生过程，cache 是 partition 分区计算完后就执行，而后者是要整个 rdd 计算完再去起新的 job 完成，成本更大；2.执行完后 cache 无论存在内存还是硬盘都会被清理，而后者不会，除非手动清理；3.cache保存依赖关系，而后者删除所有依赖关系。4.读取一个同时被 cache 和 checkpoint 处理过的 rdd，会先读取前者。



1



Peter

2019-05-18

在缓存 RDD 的时候，它所有的依赖关系也会被一并存下来。所以持久化的 RDD 有自动的容错机制。如果 RDD 的任一分区丢失了，通过使用原先创建它的转换操作，它将会被自动重算

作者回复: 👍



1



明翼

2019-05-17

checkpoint用的不多，是不是可以对目前所有的rdd均缓存，rdd是针对特定rdd缓存



1



紫日

2022-11-03 来自北京

RD持久化是给使用这优化选择，对外优化；Checkpoint面对复杂计算罗，是系统自动行为，对内部优化。

