第3章

Spark内核架构

Spark是一个用于大规模数据处理分析的引擎，它为分布式数据处理提供了方便的API，使用户可以更加便捷地用统一的API操作分布式的数据集。本章将介绍Spark的内核架构，剖析Spark的重要组件，为后续性能调优章节提供理论支持。本章的主要内容有：

- Spark编程模型。
- Spark组件简介。
- Spark作业执行原理。
- Spark内存管理。
- Spark存储原理。

3.1 Spark编程模型


视频讲解

Spark能够做到分布式的迭代计算、容错恢复等，离不开其最基本、最重要的抽象——RDD。本节将对RDD的概念及RDD的重要特性进行讲解，并分析Spark内部对RDD的编码实现。


视频讲解

3.1.1 RDD概述

正如MapReduce的编程模型来源于Google的论文一样，RDD最初的概念也是来源于一篇论文——伯克利实验室的*Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*。这篇论文奠定了RDD实现功能的基本思想，感兴趣的读者可以查看这篇论文，网址是http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf。

从论文的名称即可看出，RDD实际为Resilient Distributed Datasets的简称，意为弹性分布式数据集，是一种基于集群内存计算的容错的抽象。因此，RDD定义的是一种编程抽

象，它规定了在这样的一个基于内存计算的容错的场景下，RDD 应该具备的功能。RDD 并不是 Spark 独有的编程模型，Spark 只是对 RDD 编程模型的一种实现，如果用户有一定的编程功底，完全可以根据这个思想开发出另一套类似于 Spark 的计算引擎。

3.1.2 RDD 的基本属性

对于计算系统而言，RDD 是一个数据集操作的接口，集群系统可以对 RDD 的数据进行存储、计算等操作。对于用户而言，RDD 是一组数据集，用户可以通过 RDD 的 API 对其进行一些操作，如进行转换、过滤等操作。基于以上 RDD 的基本功能，RDD 被抽象出几个最基本的属性，分别为分区、计算函数、依赖、分区器、首选运行位置。这几个基本属性对于 RDD 而言非常重要，Spark 的程序实现基本上都是围绕这些属性进行操作的，所以理解 RDD 的原理是理解 Spark 的关键。

1. 分区

RDD 的中文含义为弹性分布式数据集，其中分区的概念实现了分布式所需的功能。一个 RDD 从某种角度上讲就是一组数据集，在一些大规模计算中，一个数据集中的数据量会达到非常大的级别，而这些数据难以在一台机器中进行存储、计算。RDD 的解决思路就是将数据进行分区，一个大的数据集被分为很多小的分区，每个分区中包含有 RDD 中的一部分数据，大量的分区可以在不同的节点中同时运行，通过对每个分区的数据进行计算，然后对计算结果进行汇总，从而实现对整个数据集的计算。这些分区的数量按照数字从 0 开始依次进行编号，RDD 的分区如图 3.1 所示。

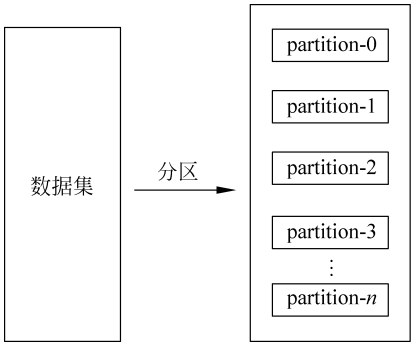

图 3.1 RDD 的逻辑分区

在真正计算时，RDD 的不同分区是分散到不同的计算节点进行计算的，每个计算节点计算 RDD 的一个或多个分区，从而形成数据的分布式计算。这里需要注意的是，RDD 的一个分区数据是不能分散到多个节点上的，一个分区的数据只能在某个具体的节点上。RDD 在节点上的分布如图 3.2 所示。

RDD 的计算是以分区为单位进行的，而且同一分区的所有数据都进行相同的计算。对于一个分区数据而言，必须执行相同的操作：要么都执行，要么都不执行。所以 RDD 的计算是一种粗粒度的计算操作，必须对一批数据全部执行相同的操作。分区的数量决定了同时执行的任务的数量，因为可以为每个分区启动一个计算任务用于单独计算这个分区的数据。理论上分区数越大，能够并行执行的任务数量就越多。但实际运行时，还会受到物理资源如 CPU 等的限制。

2. 计算函数

RDD 的数据被分区了，但是每个分区的数据是如何得来的呢？一个 RDD 的数据来源只有两种：一是从数据源或集合中进行加载得到 RDD 的数据；二是通过其他的 RDD 进行一定的转换得到的数据。无论哪一种方式，RDD 的数据其实都是通过 RDD 的计算函数得到的。

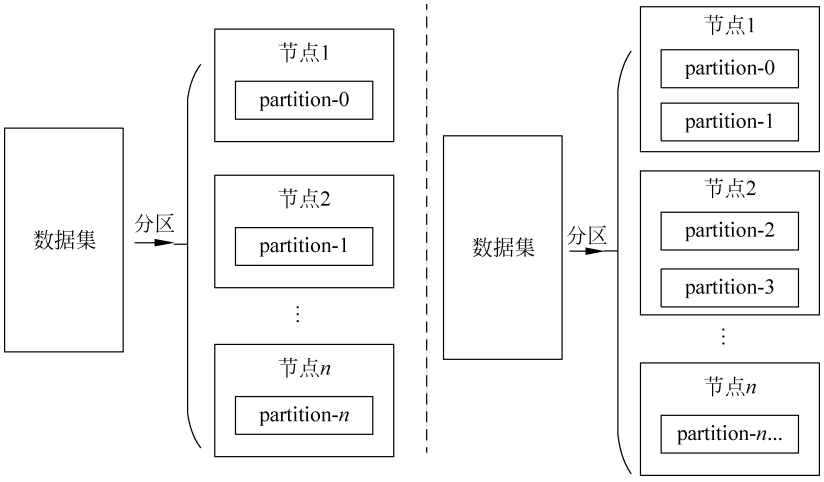

图 3.2　RDD 在节点上的分布

1）创建新的 RDD

RDD 的计算函数定义了如何根据一个分区计算出该分区的数据，并且这个函数的返回值为迭代器类型。这样通过给定某一个具体的分区，就可以通过计算函数计算出该 RDD 这个分区的数据。通过对 RDD 的所有分区进行计算，即可得到 RDD 的所有分区的数据。如 Spark 在加载 HDFS 中的数据时，每个分区的数据通过计算函数加载对应 block 块的数据，从而实现了数据的分布式加载的过程，如图 3.3 所示。

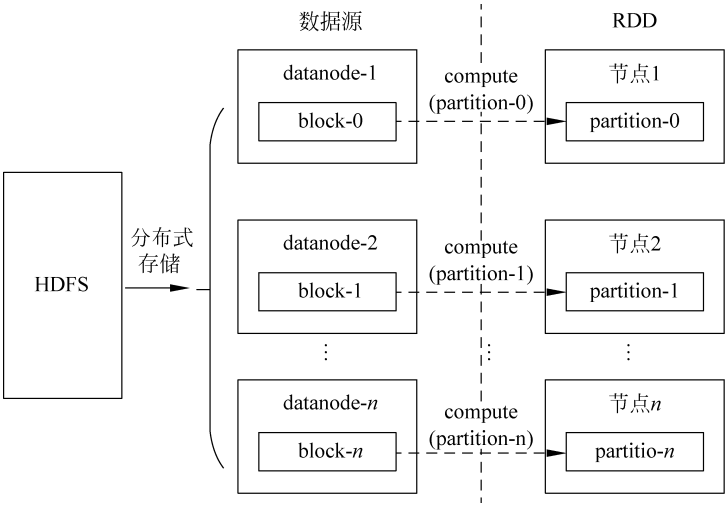

图 3.3　RDD 从 HDFS 加载示意图

RDD 通过定义分区和计算函数可以实现非常丰富的数据加载的类型，如上文中提到的从 HDFS 中加载数据，每个分区加载一个 block 块的数据，还可以实现从集合中进行创建，实现每个分区加载一个集合中的部分数据，如 SparkContext 中实现的 parallelize 的并行集合的方法。甚至用户可以自定义分区函数实现特定加载数据的方式，如将历史数据按照时间分区进行加载等，如图 3.4 所示。

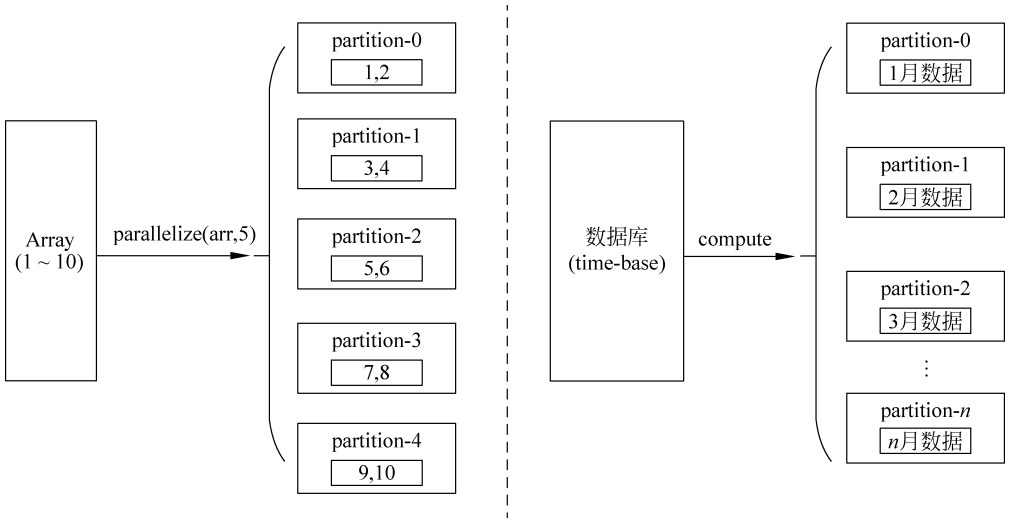

图 3.4　RDD 数据加载方式

2）RDD 的转换

一个 RDD 中的数据还可以通过其他 RDD 中的数据转换得到，通过一定的转换操作将一个 RDD 转换为一个新的 RDD，其中新 RDD 一般称为子 RDD，依赖的 RDD 成为父 RDD。如图 3.4 所示，将 Array 转换的 RDD 每个元素都加 1，即可形成一个新的 RDD，其转换过程如图 3.5 所示。

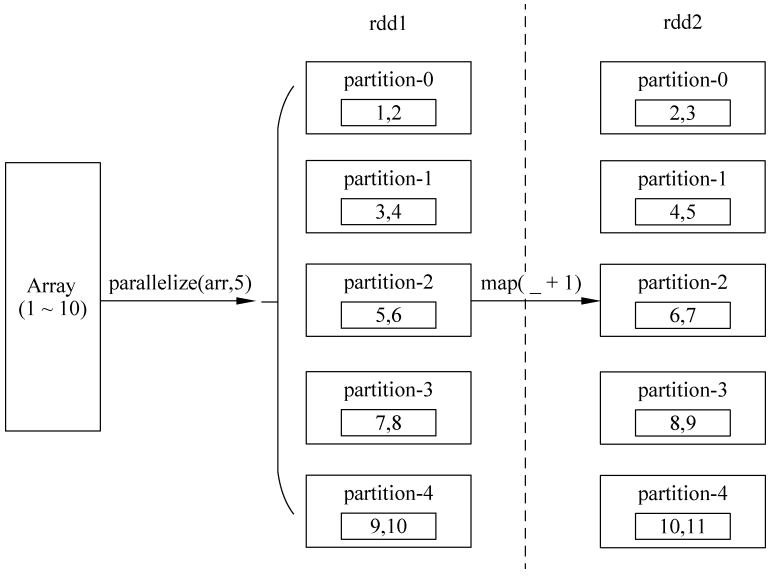

图 3.5　RDD 的转换操作

在图 3.5 中，rdd2 通过该 RDD 的计算函数在 rdd1 的数据基础之上计算出了该 RDD 的每个分区的数据。当需要计算 rdd2 的数据时，rdd2 就会调用其计算函数，使用 rdd1 的数据进行计算，如果 rdd1 的数据不存在，则首先调用 rdd1 的计算函数计算 rdd1 每个分区的数据。

这里需要注意的是，rdd2 的计算函数并不是图 3.5 中的 map()函数，计算函数是 rdd2 的一个属性，rdd1 通过 map()函数生成了 rdd2。在该过程中，确定了 rdd2 的计算函数应该如何执行。rdd2 的计算函数就是获取 rdd1 中的对应分区的数据，对该分区的数据中执行 map()函数传入的匿名函数(_+1)进行转换，从而得到 rdd2 对应分区的数据。rdd1 的转换操作是 API 级别的，可以控制整个 RDD 的转换，但真正数据的转换需要依靠每个分区的数据进行，这个过程就是由 RDD 的计算函数实现的，因此 RDD 的计算函数一定是和某个分区进行关联的，给定分区才能进行计算，具体计算过程取决于父 RDD 的转换操作。

从 RDD 的角度来看，map()函数是 rdd1 进行的转换，是 API 级别的转换；而计算函数是 rdd2 的属性，是 rdd2 执行的操作，是真正的数据转换过程，只不过 rdd2 执行计算函数时需要 rdd1 对应分区的数据。

在图 3.5 中，rdd1 也会有自己的计算函数，rdd1 的计算函数用于计算 rdd1 中的每个分区的数据。

当形成多个 RDD 的调用，如图 3.6 所示，使用最后一个 rdd4 的计算结果时，会首先调用 rdd4 的计算函数，计算 rdd4 中每个分区的数据。但是 rdd4d 分区的数据依赖于 rdd3 的数据，此时会继续调用 rdd3 的计算函数计算 rdd3 每个分区的数据。在这个过程中，各个 RDD 依次调用其依赖的 RDD 的计算函数，计算依赖的 RDD 分区的数据，并根据依赖的 RDD 分区的数据通过自身的计算函数计算出本 RDD 每个分区中的数据，从而实现 RDD 之间的流水线式的调用。在这多个 RDD 转换的过程中，因为它们之间的分区是一一对应的，也就是每个 RDD 只依赖父 RDD 的一个固定的分区，所以计算一个 RDD 分区的数据时，不必知道父 RDD 所有分区的数据，只需要知道依赖的这个分区的数据即可。每个分区中的数据可以通过一个流水线的任务(task)转换完成，各个任务之间相互独立，互不影响。而且通过流水线的调用避免了中间 RDD 结果的存储。在 rdd1 转换为 rdd4 的过程中，每个分区的数据由一个任务(task)即可完成。在一个具体的任务中，负责依次执行每个 RDD 的计算函数，从而以流水线的形式完成该数据的计算，避免了中间结果的存储。

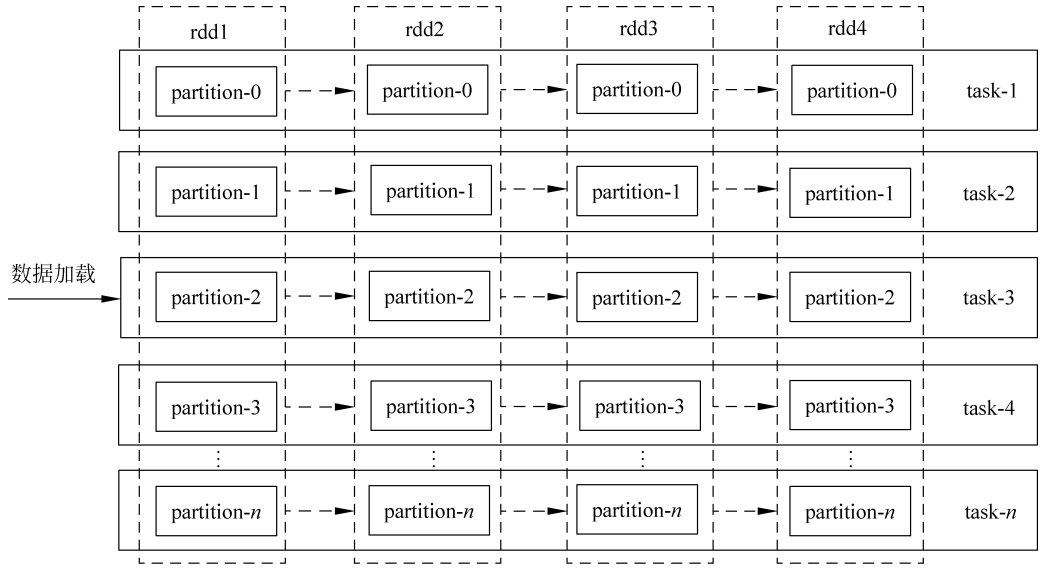

图 3.6　RDD 之间的流水线转换

RDD之间的转换不仅仅限于一个RDD转换为另一个RDD，还可以将多个RDD合并成一个RDD，新RDD的分区数为合并的RDD的分区数之和，这个合并的过程也是由新RDD的计算函数完成的。新RDD的分区需要与合并的分区通过计算函数实现关联，由计算函数计算每个分区中的数据，实现新RDD的分区和父RDD的分区数据的对应。其过程如图3.7所示。在这个计算过程中，每个分区也可以使用一个计算任务完成流水线的计算。

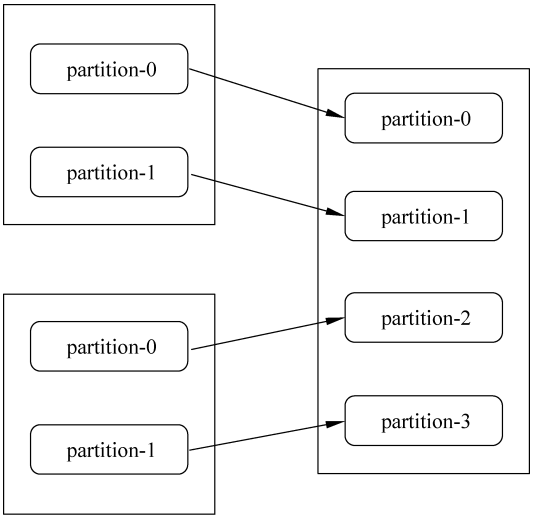

图3.7 RDD合并

在以上RDD的转换中，父RDD的每一个分区总会对应子RDD中的一个分区。但这并不是必然的，同样取决于子RDD的计算函数是如何工作的。图3.8是实现WordCount的经典过程，在rdd1中，每个分区中保存了文本中出现的每个单词，在rdd2中，将每个单词转换为(word,1)的形式，但最终希望得到每个单词出现的总次数，形成rdd3的结果。在这个过程中，rdd2到rdd3之间的转换便不再是上文中提到的父RDD的一个分区只被子RDD的一个分区使用，而是父RDD的每个分区的数据可能被子RDD的任何一个分区使用。在计算rdd3每个分区的数据时，需要知道rdd2中所有分区的数据，当rdd3计算一个分区的数据时，需要拉取rdd2上的所有分区对应的数据，完成这个分区数据的计算，这个过程称为Shuffle。由于计算rdd3的其中一个分区的数据，必须要知道rdd2中所有分区的数据，所以在rdd2转换为rdd3的过程中，就不能再像流水线一样进行计算，必须先计算rdd2，将rdd2所有分区的数据都计算完成以后，才能计算rdd3的数据。

在计算RDD的过程中，如果出现Shuffle，则其过程有如下特点。

- 必须首先计算出依赖的RDD的所有分区的数据，然后后续RDD才能够继续进行计算。
- Shuffle的过程必然分为两个阶段：第一个阶段为计算依赖RDD的数据的阶段(图3.8中的rdd2)，一般称为Map阶段；第二个阶段为计算汇聚结果的阶段(图3.8中的rdd3)，一般称为Reduce阶段。
- 两个阶段必须分到两组任务中进行计算，不能像流水线一样使用一组任务同时计算两个阶段的全部数据。因为后一个阶段必须在上一个阶段的数据全部计算完成以后才能够开始计算，所以必须拆分成两组不同的任务按照先后顺序执行。

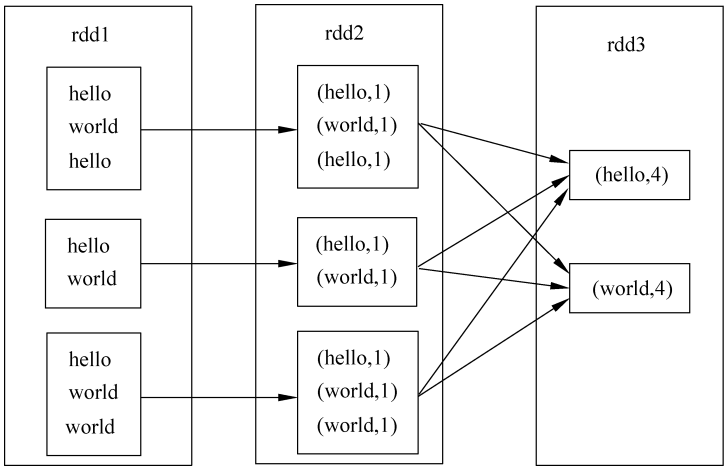

图 3.8 Shuffle 过程

涉及 Shuffle 操作的 RDD 的计算函数是比较复杂的，它必须向依赖的 RDD 的所有分区拉取需要的数据，完成某一个分区数据的计算。

无论是从数据源中创建 RDD，还是通过其他 RDD 进行转换，甚至通过 Shuffle 得到新的 RDD，这些操作都离不开 RDD 的两个基本属性：分区和计算函数。只有知道了 RDD 是如何被分区的、这个 RDD 一共有多少个分区，才能够通过 RDD 的计算函数计算每一分区的数据。

RDD 的计算函数返回值都是迭代器类型，该迭代器中能够返回 RDD 某分区的所有的数据。RDD 的计算函数通过迭代器的形式，避免了分区的数据同时加载至内存中，从而避免了大量内存被占用。

3. 依赖

在 RDD 进行转换的过程中，子 RDD 都是通过父 RDD 转换而来的。但在具体的实现过程中，所有 RDD 的数据都是通过它自身的计算函数而得到的。所以，子 RDD 在计算的过程中需要得到其父 RDD，根据父 RDD 的数据计算出子 RDD 的每个分区的数据。子 RDD 在计算时，就是通过依赖的属性，找到其依赖的父 RDD 的。

在 RDD 进行计算时，有些子 RDD 的一个分区只依赖父 RDD 的一个分区，即每个父 RDD 的分区最多被子 RDD 的一个分区使用，这种依赖关系称为窄依赖。在窄依赖中，多个 RDD 的每一组分区都能够被一个任务以流水线的形式执行。RDD 的窄依赖如图 3.9 所示。

在 RDD 进行计算时，如果一个分区的数据依赖了父 RDD 的多个分区的数据，即多个子 RDD 的分区数据依赖父 RDD 的同一个分区的数据，则这种依赖方式称为宽依赖。在发生宽依赖的位置，必定要发生 Shuffle 操作，将这个过程分为两个阶段进行操作。第一个阶段计算父 RDD 的所有分区的数据，并将结果进行保存。在这个过程中每个分区保存的数据已经做出了分类，区分哪一部分的数据为第二阶段的哪个分区使用。第一阶段完成之后，才能进行第二阶段。第二个阶段在计算每一个分区数据时，通过拉取父 RDD 的对应分区的数据完成数据的聚合。RDD 的宽依赖如图 3.10 所示。

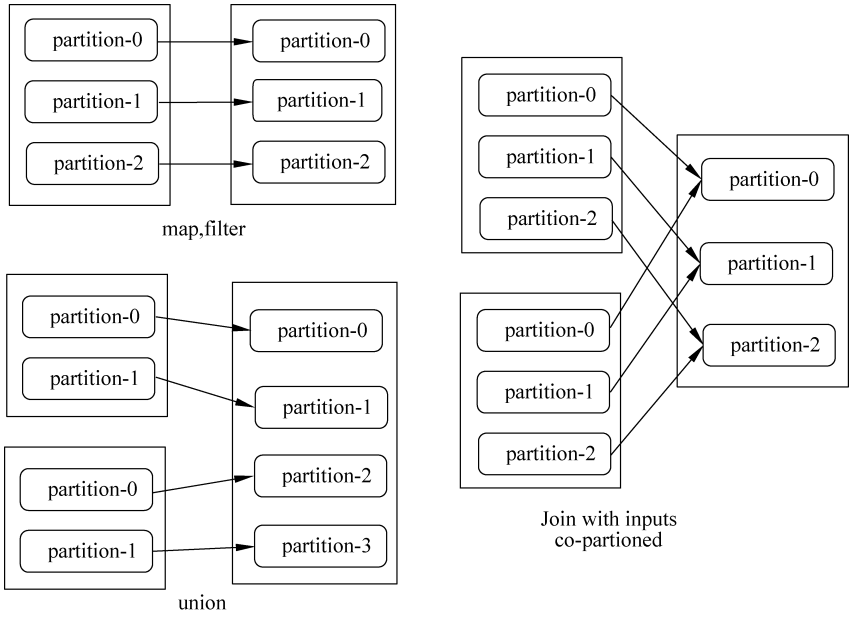

图 3.9　RDD 的窄依赖

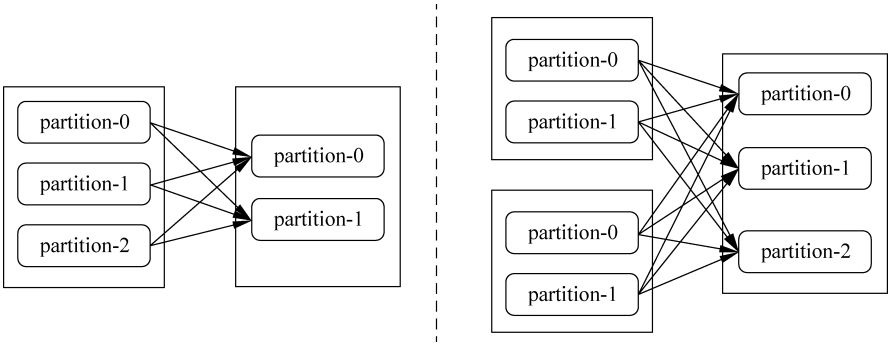

图 3.10　RDD 的宽依赖

RDD 根据其依赖关系和计算函数便可以根据父 RDD 的数据计算出每个分区的数据。RDD 这样的转换和依赖关系称为血统(lineage),即每一个 RDD 只要根据其 lineage,就能够把 RDD 的数据计算出来。甚至根据 lineage 只计算其中某一个分区的数据,而不用计算 RDD 中所有分区的数据。

4. 分区器

并不是所有的 RDD 都有分区器(partitioner),一般只有(key,value)形式的 RDD 才有分区器。分区器在 Shuffle 的 Map 阶段使用,当 RDD 的计算发生 Shuffle 时,Map 阶段虽然将计算结果进行了保存,供 Reduce 阶段的任务来拉取数据,但是 Map 阶段的每个分区的结果可能会被 Reduce 阶段的多个分区使用。如何把 Map 阶段的结果进行分组,区分出结果是给 Reduce 阶段的 RDD 哪个分区呢?这就是分区器的作用。分区器根据每条记录的 key,判断这个 key 属于 Reduce 哪个分区的数据,同时分区器中也计算了按照 key 进行分区将会产生分区的总数量。这个数量决定了在 Map 阶段每个任务结果分组的大小,也决定了

下游的 Reduce 任务的分区数量的大小。在一个分区数为 2 的分区器中，分区器的工作原理如图 3.11 所示。

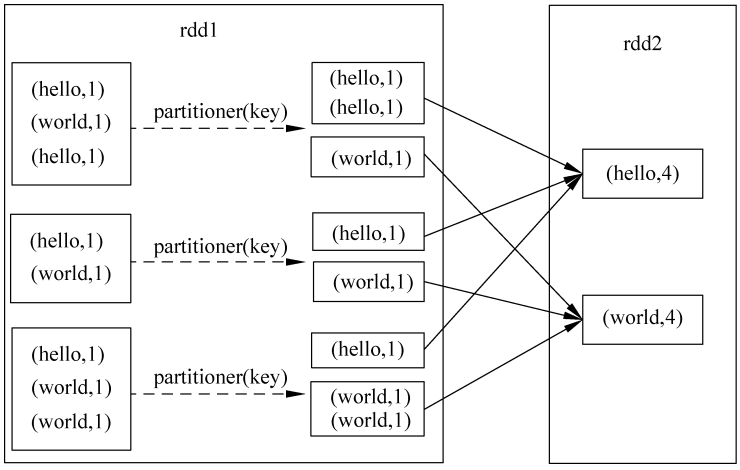

图 3.11 分区器的工作原理

5. 首选运行位置

每个 RDD 对于每个分区来说有一组首选运行位置，用于标识 RDD 的这个分区数据最好能够在哪台主机上运行。通过 RDD 的首选运行位置，可以让 RDD 的某个分区的计算任务直接在指定的主机上运行，从而实现了移动计算而不是移动数据的目的，减小了网络传输的开销，如 Spark 中 HadoopRDD 能够实现加载数据的任务在相应的数据节点上执行。

3.1.3 RDD 的缓存

RDD 是进行迭代式计算的，默认并不会保存中间结果的数据，在计算完成后，中间迭代的结果数据都将会丢失。如果一个 RDD 在计算完成后，不是通过流水线的方式被一个 RDD 调用，而是被多个 RDD 分别调用，则在计算过程中就需要对 RDD 进行保存，避免 RDD 的第二次执行相同的计算。当一个 RDD 被缓存后，后边调用的时候需要 RDD 的数据时，直接从缓存中读取，而不是对 RDD 再次进行计算。尤其是一个 RDD 经过了特别复杂的计算过程、经过多次 Shuffle 生成的数据，如果多次使用其结果，对其进行缓存，可以极大地提高程序的执行效率。其原理如图 3.12 所示。

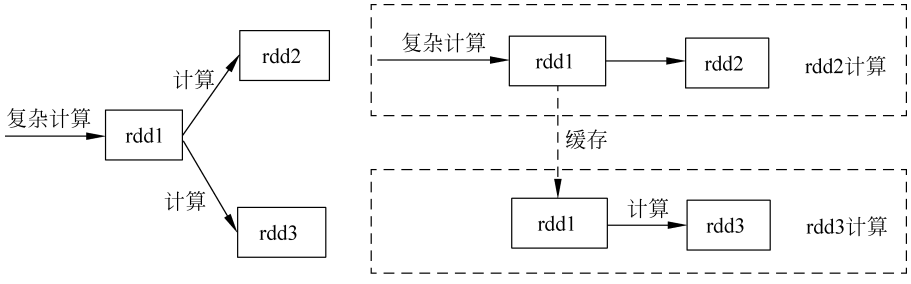

图 3.12 RDD 缓存计算原理

因为 RDD 的数据是分布式的，不同的分区散落在不同的节点上，所以 RDD 的缓存也是分布式的。当对一个 RDD 进行缓存时，可以直接将每个分区的数据缓存在当前分区的计算节点中，每个节点缓存 RDD 的一部分，完成整个 RDD 的缓存。RDD 在节点中的缓存如图 3.13 所示。

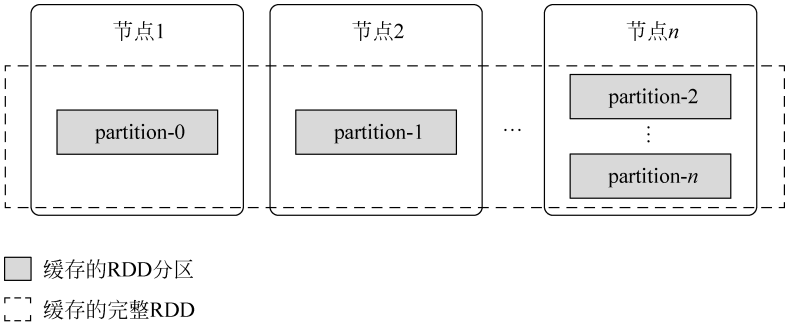

图 3.13　RDD 在节点中的缓存

根据程序实现的不同，缓存的数据可以选择在内存、磁盘或其他的外部存储器中。也可以实现多副本机制，将本节点的数据复制到其他的节点进行保存，实现数据的冗余。

3.1.4　RDD 容错机制

RDD 的容错是通过其 lineage 机制实现的。因为一个 RDD 的数据都可以通过其父 RDD 的转换而来。如果在运行的过程中，某一分区的缓存数据丢失，则重新计算该分区的数据，当此 RDD 的依赖是窄依赖时，只需要计算依赖的父 RDD 的一个分区的数据即可，避免了一个节点出错则所有数据都需重新计算的缺点。但是如果丢失数据的 RDD 的依赖是宽依赖，那么这个分区的数据可能依赖父 RDD 的所有分区的数据，在这种情况下就必须重新计算父 RDD 的所有分区的数据，从而完成数据恢复。RDD 容错机制如图 3.14 所示。

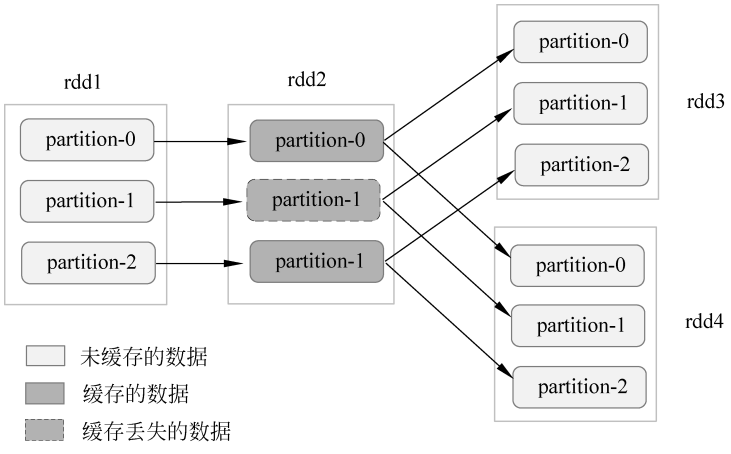

图 3.14　RDD 的容错