Apache Spark 源码阅读

ihainan

Published with GitBook

目錄

Apache Spark 源码阅读

Overview

本文档是我去年（2014）年末学习 Apache Spark 源码时所留下的阅读笔记，原为 Microsoft Word 文档。近期出于毕业求职需要，重温源码，顺带整理了下原文档，转换成 Markdown 文档，修正原文中出现的一些错误，对缺漏之处也做了相应补全。整理过后的文档会放在我的 Github 仓库 和 GitBook 上。

本文档对应的 **Apache Spark** 源码版本为 **1.4.1**。代码仓库中，不同分支表示不同版本的 Spark。

本人经验、能力以及实验条件实在是有限，在研究过程中难免会有诸多不足。若在阅读本文档时发现有错误与遗漏之处，还望能够提出指正。

How to Read

在学习过程中，Matei Zaharia 发表的论文 ***Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*** 和他的博士毕业论文 ***An Architecture for Fast and General Data Processing on Large Clusters*** ，Jerry Lead 关于 Apache Spark 内部机制的系列文章，ColZer 关于 Apache Spark 的 学习笔记，等等文章资料都给了我相当多的帮助。在阅读本系列文章时，我十分推荐配合上面几篇材料一起学习，不同文章解析 Apache Spark 的角度各不相同，相信都会对你有所启发。

搭配 IntelliJ IDEA + Apache Spark 1.4.1 源码阅读本文档味道更佳。具体配置步骤如下：

- 下载并解压缩 **Apache Spark 1.4.1** 源码

源码下载地址点此，在选择 Package Type 时候需要注意选择 **Source Code（can build several Hadoop Version）**。

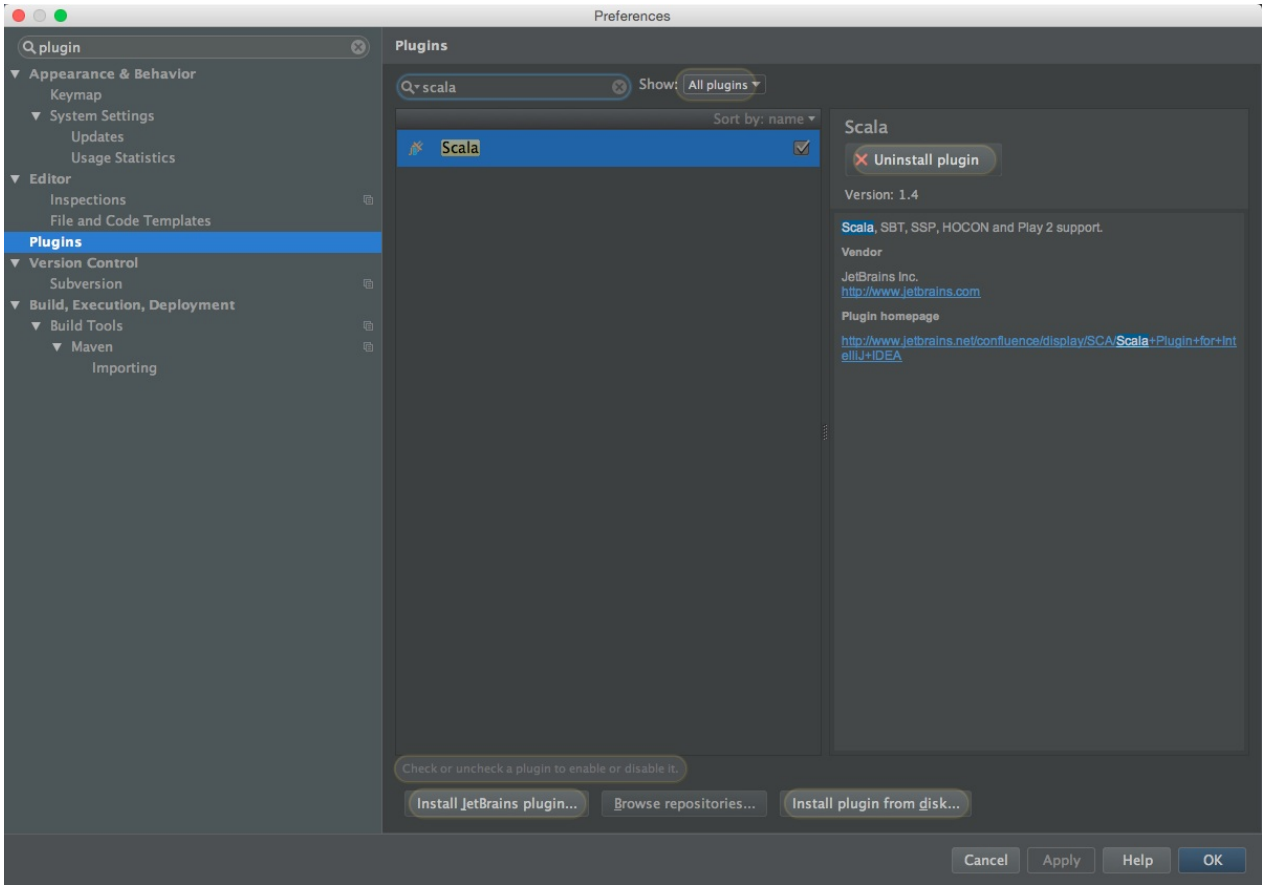
- 编译 **Apache Spark 1.4.1** 源码

依照官方文档 Build Spark - Spark 1.4.1 Documentation 所述步骤和参数，在命令行下编译 Apache Spark 1.4.1 的源码，也可以参考文档 Useful Developer Tools - Spark 中的方法直接在 IDE 中编译，本文采用前一种方法，编译和下载依赖的时间

会比较长，我在一个全新的系统中编译整套源码用了将近一小时半的时间。

- **IntelliJ IDEA** 安装 **Scala** 插件

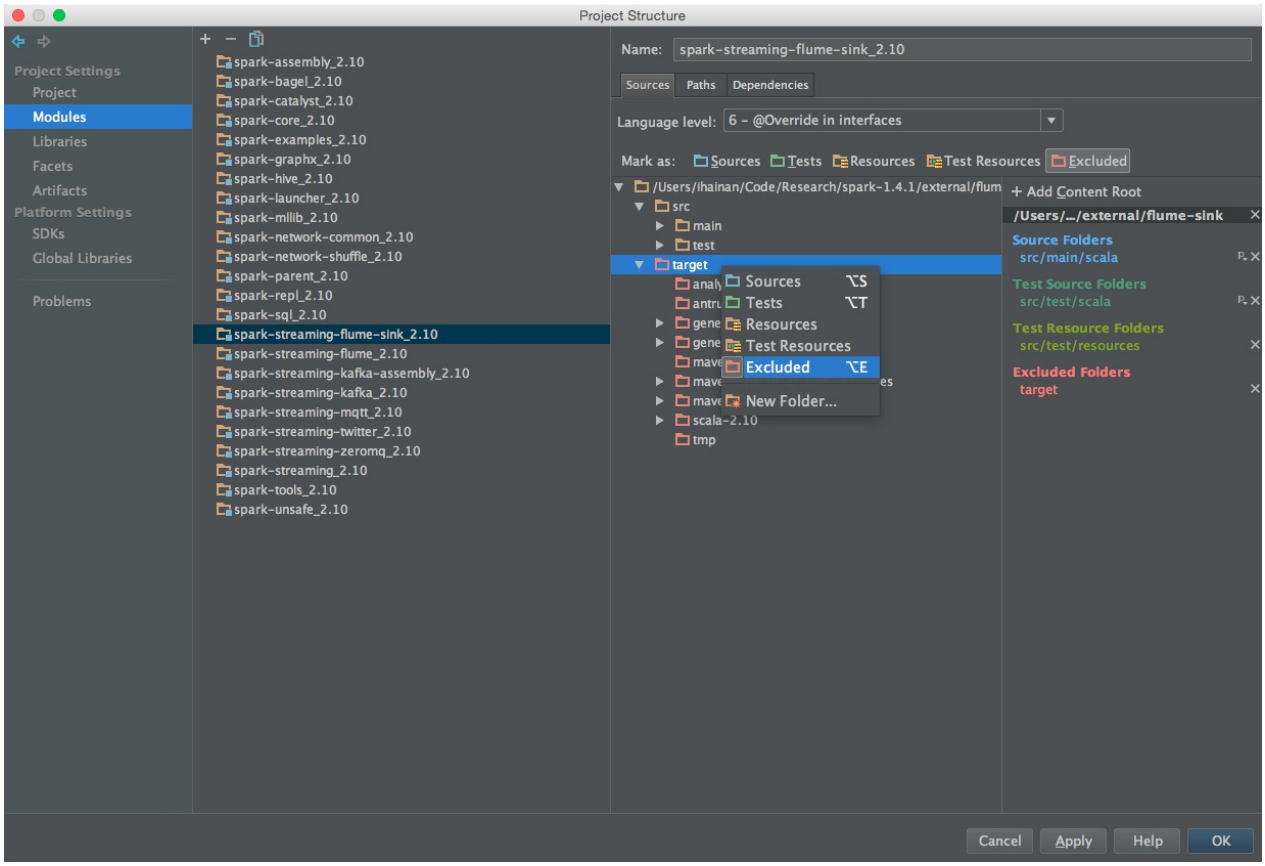
依次选择 Preferences -> Plugins -> Install JetBrains plugin。搜索框输入 scala，右侧点击 Install Plugin，安装成功后重启 IntelliJ IDEA。



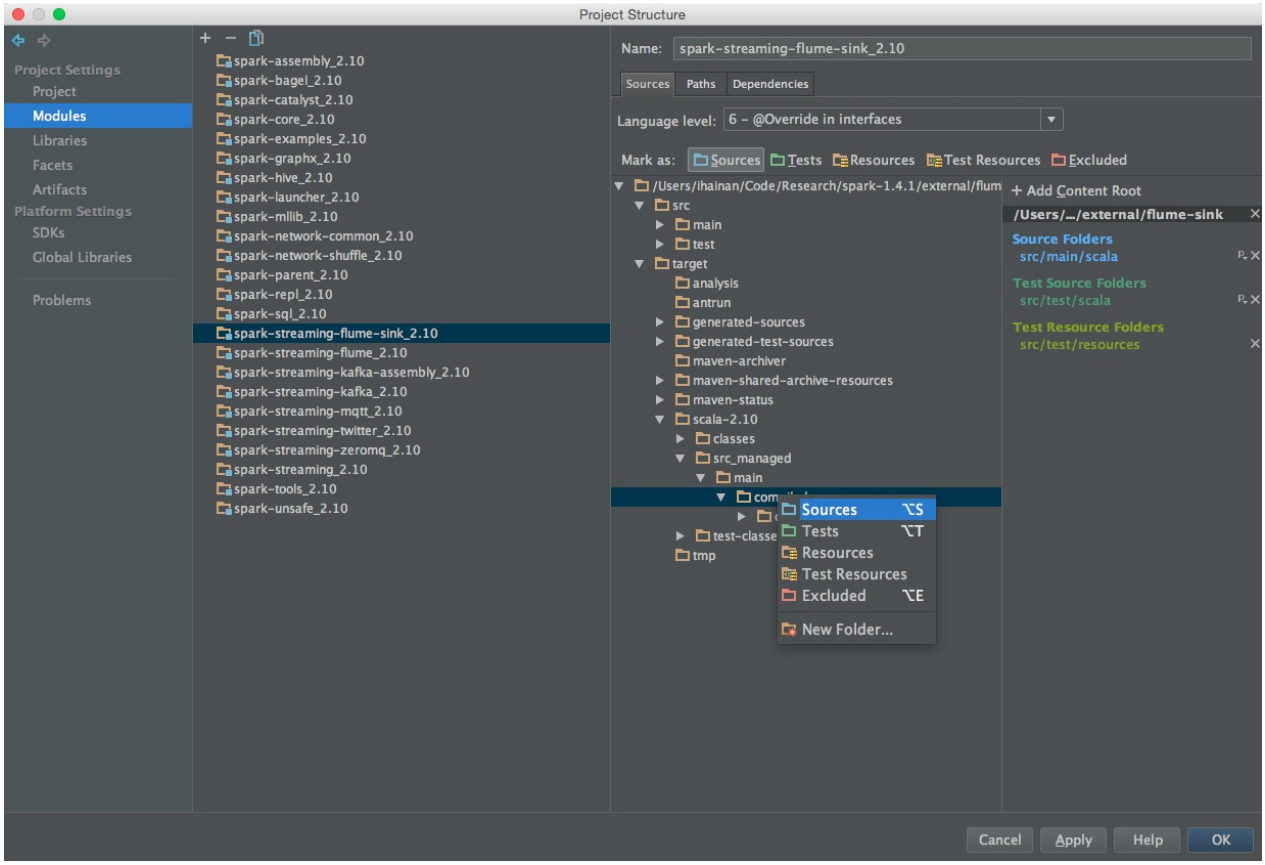
- **IntelliJ IDEA** 导入 **Apache Spark 1.4.1** 源码

使用 IntelliJ IDEA 打开 Apache Spark 1.4.1 源码目录下的 pom.xml 文件。

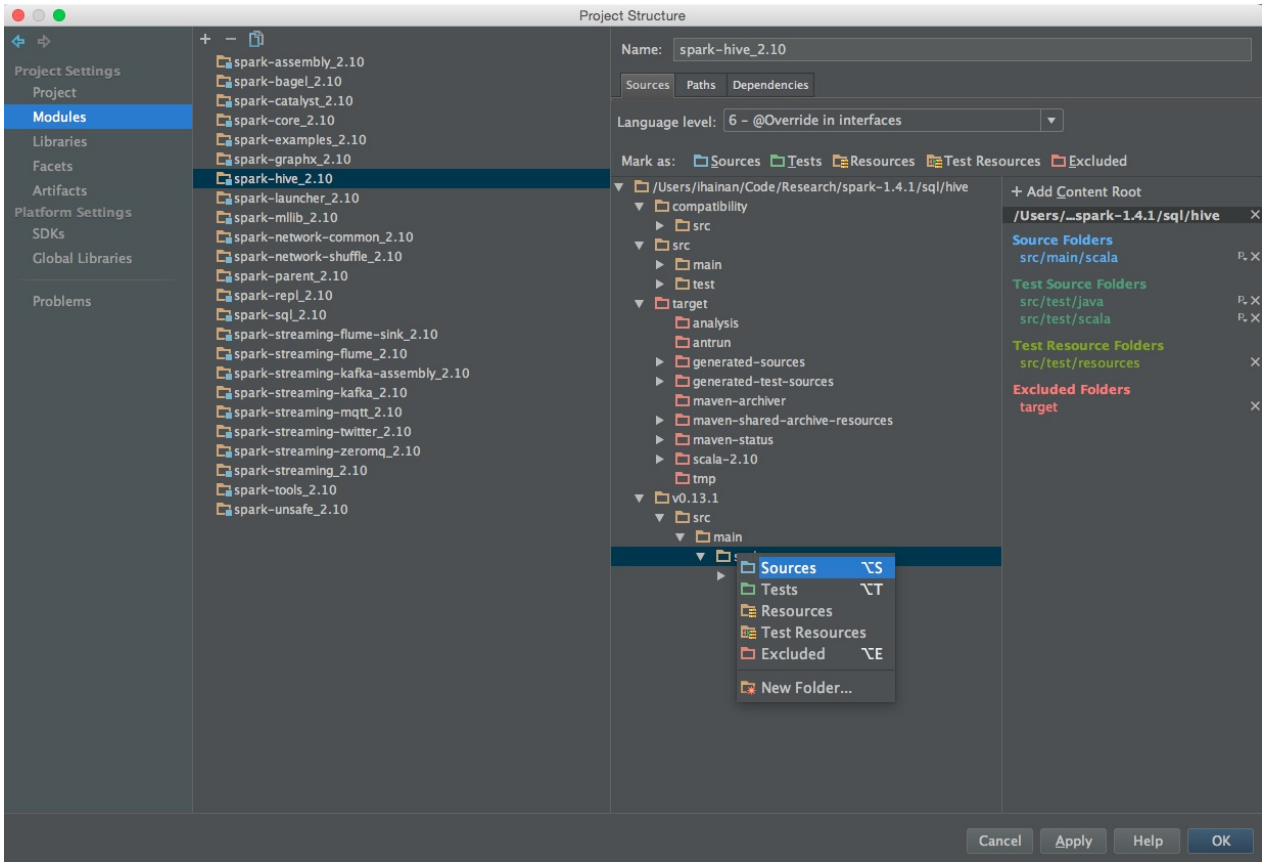
导入项目完成后，依次打开 Project Structure -> Modules -> spark-streaming-flume-sink.2.10，右键 target 目录，取消 Excluded 标签。



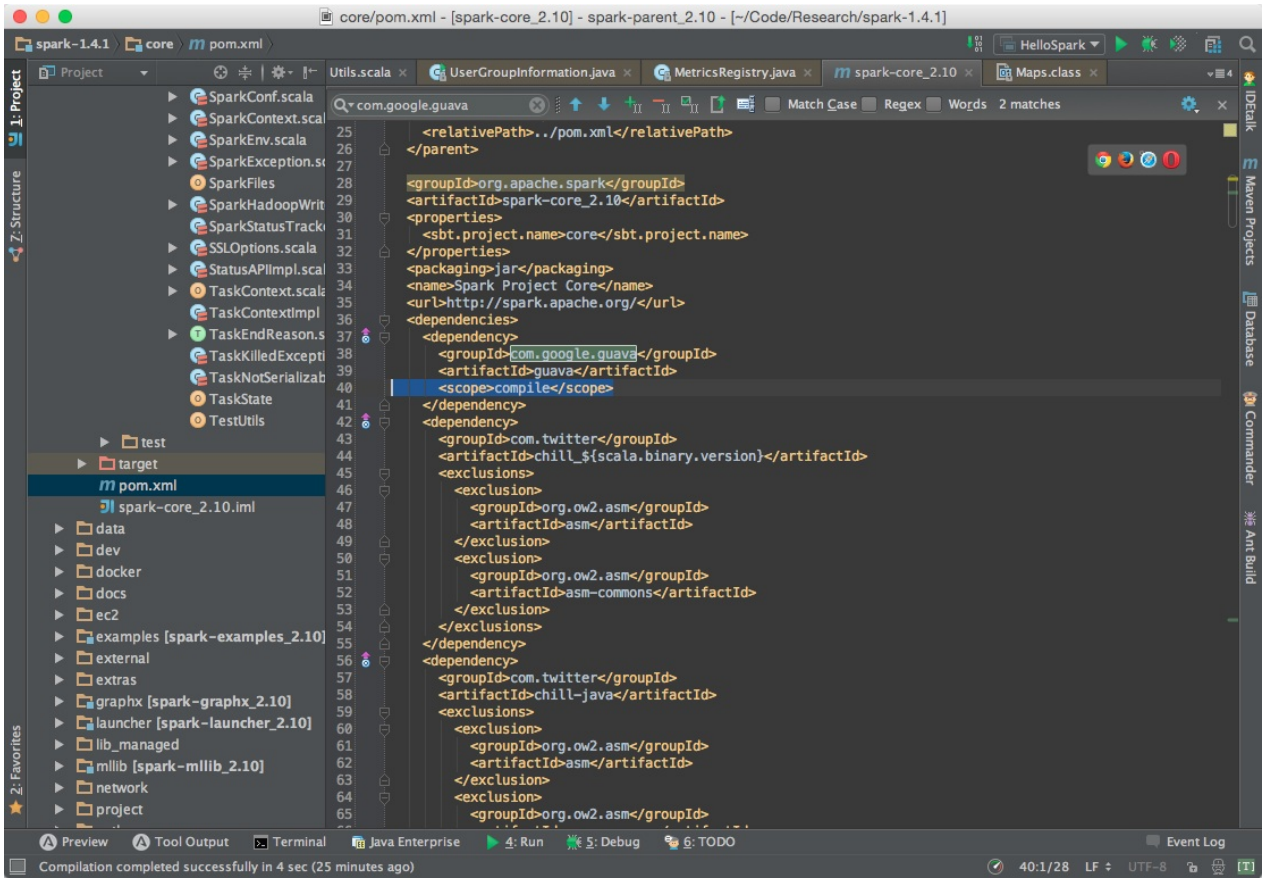
展开到 target -> scala-2.10 -> src_managed -> main -> compiled_avro 目录，右键，标记为 Source 目录。



同样，将 spark-hive_2.10 模块内的 v0.13.1 -> src -> main -> scala 目录标记为 Source 目录。



（**1.4** 之前版本无需此操作）编辑 pom.xml 文件，定位到如下位置，添加 `<scope>compile</scope>`，否则会出现能够顺利编译，运行时抛出异常 `NoClassDefFoundError` 的情况。

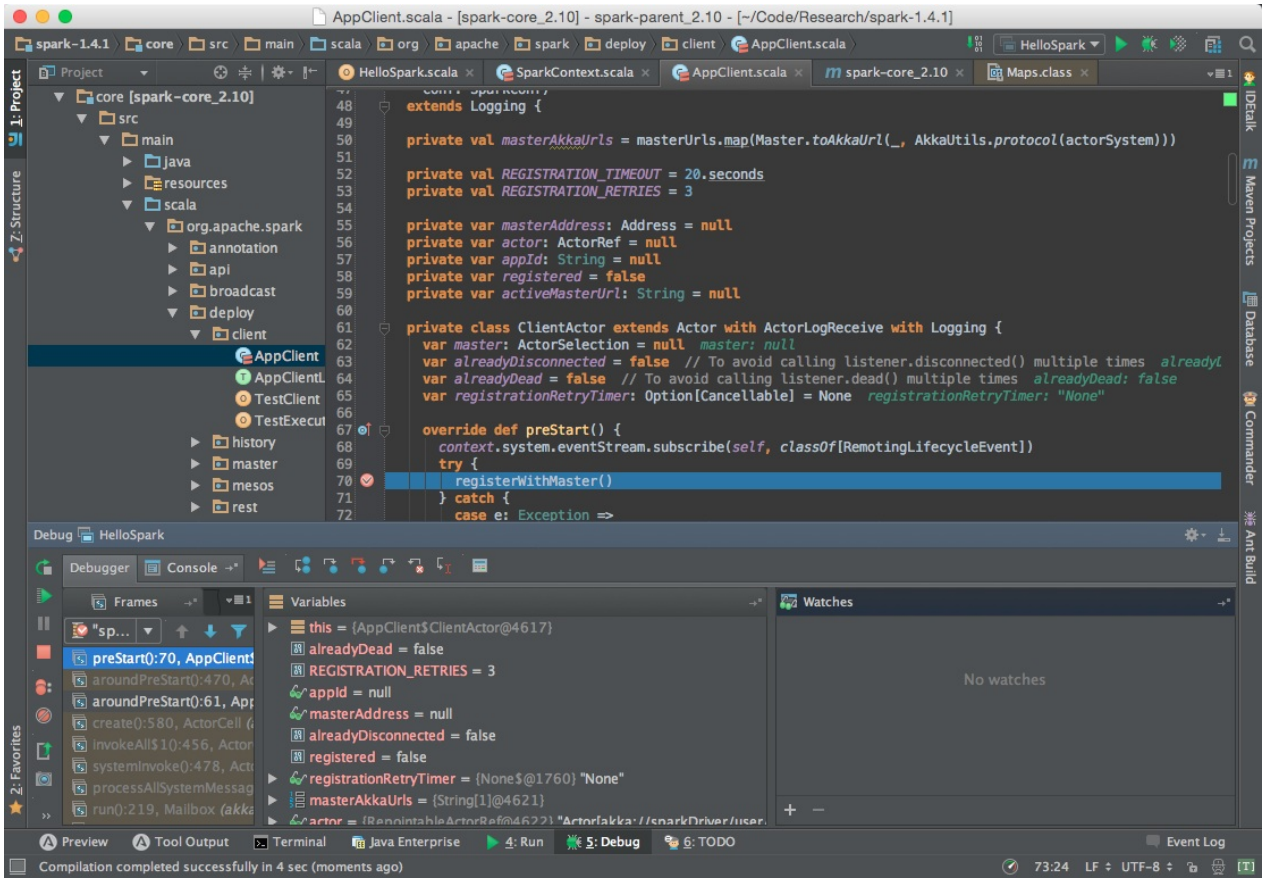


点击 Build -> Rebuild Project，等待项目构建完成即可。

- 编写测试程序，调试

我对源码的注释放在此处，里面还包含了一些测试用的程序。其中一些标记符号的含义如下：

```
I: Info，表信息，是我对代码的理解。
Q: Question，表问题，指我对代码的一些疑惑之处。
L: Link，表链接，附上一些有用的参考连接。
T: Test，表测试。
R: Result，表测试结果。
```



LICENSE

Author

@ihainan

RDD 内部结构

从 RDD 的名字说起

RDD 作为 Apache Spark 中最为重要的一类数据抽象，同时也是 Apache Spark 程序开发者接触最多的数据结构，自然而然地，也就成为我理解 Apache Spark 工作原理的最佳入口之一。

RDD 全称为 Resilient Distributed Datasets，即弹性分布式数据集，我对弹性分布式数据集的理解如下：

1. 数据集：顾名思义，说明 RDD 是数据集合的抽象，从外部来看，RDD 的确可被看待成带扩展特性（如容错性等）的数据集合。
2. 分布式：数据的计算并非只局限于单个节点，而是多个节点之间协同计算得到。
3. 弹性：RDD 内部数据是只读的，但 RDD 却具有弹性这一特性，实际上，RDD 可以在不改变内部存储数据记录的前提下，去调整并行计算计算单元的划分结构，弹性这一特性，也是为并行计算服务的。

我把 RDD 归纳为一句话：能够进行并行计算的数据集，其中最重要的是并行计算这一特征，基于它，可以进一步往下思考 RDD 在设计上的一些问题。

首先，我提及到了并行计算的计算单元，那么在 RDD 里面，这些单元应该如何表示更为合适；既然要进行并行计算，我们自然希望计算单元能够尽可能地均匀分配，从而保证集群资源能够被合理利用，那么，RDD 内部计算单元的划分依据又是什么；以及，这些计算单元又该如何被计算？

再者，分布式数据集往往需要具备一个重要特性，即容错性，分布式条件下数据的丢失可能会很常见，这时候就需要 Apache Spark 能够通过某种机制来恢复丢失的数据，从而保证数据的的可靠性和完整性。

传统方法的容错机制有两种，一是创建数据检查点，即将某个节点的数据保存在存储介质当中，二是记录更新，即记录下内部数据所遭遇过的所有的更新。对于前者，在网络中传输与复制数据集的带宽开销显然是非常庞大的，对于后者，如果要记录每一个数据记录的所有更新，成本自然也是不小。使用过 RDD 做过开发的话，自然知道 Apache Spark 最终采用的是第二种办法，而为了避免巨大的开销，

RDD 只支持粗粒度的转换操作，一个操作会应用到多个数据而非单个记录。那么，在 RDD 内部，应该如何去记录数据的更新，丢失的数据又是通过何种方式恢复的呢？

本章后面的小节都将是围着上面的这些问题进行展开，每一节都会回答一个或者多个问题，探索这些问题答案的过程中会配合解析相应的源码实现。我希望通过这种方式，能够从整体的角度去理解 Apache Spark 开发者们如此设计 RDD 的目的，而非单纯机械地一行一行去解释代码的含义。

RDD 内部接口

在 Apache Spark 源码级别，`RDD` 是一个抽象类，我们所使用的 `RDD` 实例，都是 `RDD` 的子类，例如执行 `map` 转换操作之后可以得到一个 `MapPartitionsRDD` 实例，执行 `groupByKey` 转换操作之后可以得到一个 `ShuffledRDD` 实例。不同的 `RDD` 子类会根据实际需求实现各自的功能，但无论如何，一个 `RDD` 内部都会包含如下几类接口的全部或者一部分。在后面的小节中，我们就能看到所有的这些接口究竟是如何为实现我们的并行计算目标服务的。

- 分区（Partition）相关接口
- 依赖（Dependency）相关接口
- 计算（Computing）相关接口
- 分区器（Partitioner）相关接口（可选）
- 首选位置（Prefered Location）相关接口（可选）
- 持久化（Persistence）与检查点（Checkpoint）相关接口

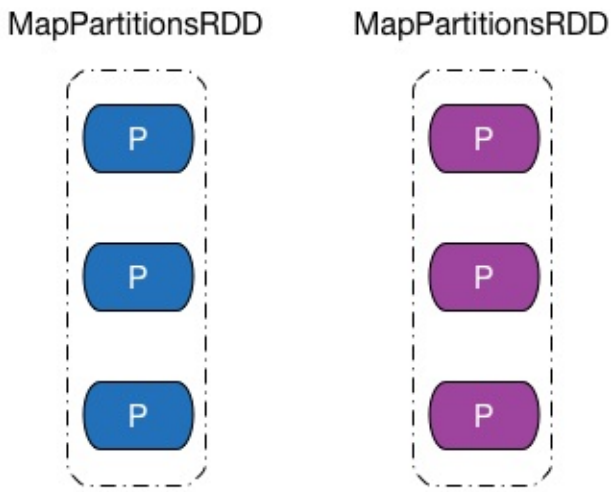
RDD 分区

分区

先回答第一个问题：RDD 内部，如何表示并行计算的一个计算单元。答案是使用分区（**Partition**）。

RDD 内部的数据集合在逻辑上和物理上被划分成多个小子集合，这样的每一个子集合我们将其称为分区，分区的个数会决定并行计算的粒度，而每一个分区数值的计算都是在一个单独的任务中进行，因此并行任务的个数，也是由 RDD（实际上是一个阶段的末 RDD，调度章节会介绍）分区的个数决定的，我会在 1.2 小节以及第二章中，具体说明分区与并行计算的关系。

在后文中，我会用下图所示图形来表示 RDD 以及 RDD 内部的分区，RDD 上方文字表示该 RDD 的类型或者名字，分区颜色为紫红色表示该 RDD 内数据被缓存到存储介质中，蓝色表示该 RDD 为普通 RDD。—— 话说这配色真的很丑么……



分区实现

分区的源码实现为 `Partition` 类。

```scala
/**
 * An identifier for a partition in an RDD.
 */
trait Partition extends Serializable {
  /**
   * Get the partition's index within its parent RDD
   */
  def index: Int

  // A better default implementation of HashCode
  override def hashCode(): Int = index
}
```

RDD 只是数据集的抽象，分区内部并不会存储具体的数据。`Partition` 类内包含一个 `index` 成员，表示该分区在 RDD 内的编号，通过 RDD 编号 + 分区编号可以唯一确定该分区对应的块编号，利用底层数据存储层提供的接口，就能从存储介质（如：HDFS、Memory）中提取出分区对应的数据。

`RDD` 抽象类中定义了 `_partitions` 数组成员和 `partitions` 方法，`partitions` 方法提供给外部开发者调用，用于获取 RDD 的所有分区。`partitions` 方法会调用内部 `getPartitions` 接口，`RDD` 的子类需要自行实现 `getPartitions` 接口。

```
  @transient private var partitions_ : Array[Partition] = null

  /**
   * Implemented by subclasses to return the set of partitions in t
   * be called once, so it is safe to implement a time-consuming co
   */
  protected def getPartitions: Array[Partition]

  /**
   * Get the array of partitions of this RDD, taking into account w
   * RDD is checkpointed or not.
   */
  final def partitions: Array[Partition] = {
    checkpointRDD.map(_.partitions).getOrElse {
      if (partitions_ == null) {
        partitions_ = getPartitions
      }
      partitions_
    }
  }
```

以 `map` 转换操作生成 `MapPartitionsRDD` 类中的 `getPartitions` 方法为例。

```
  override def getPartitions: Array[Partition] = firstParent[T].par
```

可以看到，`MapPartitionsRDD` 的分区实际上与父 RDD 的分区完全一致，这也符合我们对 `map` 转换操作的认知。

分区个数

RDD 分区的一个分配原则是：尽可能使得分区的个数，等于集群核心数目。

RDD 可以通过创建操作或者转换操作得到。转换操作中，分区的个数会根据转换操作对应多个 RDD 之间的依赖关系确定，窄依赖子 RDD 由父 RDD 分区个数决定，Shuffle 依赖由子 RDD 分区器决定。

创建操作中，程序开发者可以手动指定分区的个数，例如 `sc.parallelize(Array(1, 2, 3, 4, 5), 2)` 表示创建得到的 RDD 分区个数为 2，在没有指定分区个数的情况下，Spark 会根据集群部署模式，来确定一个分区个数默认值。

分别讨论 `parallelize` 和 `textFile` 两种通过外部数据创建生成RDD的方法。

对于 `parallelize` 方法，默认情况下，分区的个数会受 Apache Spark 配置参数 `spark.default.parallelism` 的影响，官方对该参数的解释是用于控制 Shuffle 过程中默认使用的任务数量，这也符合我们之间对分区个数与任务个数之间关系的理解。

```
  /** Distribute a local Scala collection to form an RDD.
   *
   * @note Parallelize acts lazily. If `seq` is a mutable collectio
   * to parallelize and before the first action on the RDD, the res
   * modified collection. Pass a copy of the argument to avoid this
   * @note avoid using `parallelize(Seq())` to create an empty `RDD
   * RDD with no partitions, or `parallelize(Seq[T]())` for an RDD
   */
  def parallelize[T: ClassTag](
      seq: Seq[T],
      numSlices: Int = defaultParallelism): RDD[T] = withScope {
    assertNotStopped()
    new ParallelCollectionRDD[T](this, seq, numSlices, Map[Int, Seq
  }
```

无论是以本地模式、Standalone 模式、Yarn 模式或者是 Mesos 模式来运行 Apache Spark，分区的默认个数等于对 `spark.default.parallelism` 的指定值，若该值未设置，则 Apache Spark 会根据不同集群模式的特征，来确定这个值。

对于本地模式，默认分区个数等于本地机器的 CPU 核心总数（或者是用户通过 `local[N]` 参数指定分配给 Apache Spark 的核心数目，见 `LocalBackend` 类），显然这样设置是合理的，因为把每个分区的计算任务交付给单个核心执行，能够保证最大的计算效率。

```
override def defaultParallelism() =
  scheduler.conf.getInt("spark.default.parallelism", totalCores)
```

若使用 Apache Mesos 作为集群的资源管理系统，默认分区个数等于 8（对 Apache Mesos 不是很了解，根据这个 `TODO`，个人猜测 Apache Spark 暂时还无法获取 Mesos 集群的核心总数）（见 `MesosSchedulerBackend` 类）。

```
// TODO: query Mesos for number of cores
override def defaultParallelism(): Int = sc.conf.getInt("spark.de
```

其他集群模式（Standalone 或者 Yarn），默认分区个数等于集群中所有核心数目的总和，或者 2，取两者中的较大值（见 `CoarseGrainedSchedulerBackend` 类）。

```
override def defaultParallelism(): Int = {
  conf.getInt("spark.default.parallelism", math.max(totalCoreCour
}
```

对于 `textFile` 方法，默认分区个数等于 `min(defaultParallelism, 2)`（见 `SparkContext` 类），而 `defaultParallelism` 实际上就是 `parallelism` 方法的默认分区值。

```
/**
 * Read a text file from HDFS, a local file system (available on
 * Hadoop-supported file system URI, and return it as an RDD of S
 */
def textFile(
    path: String,
    minPartitions: Int = defaultMinPartitions): RDD[String] = wit
  assertNotStopped()
  hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable
    minPartitions).map(pair => pair._2.toString)
}
```

分区内部记录个数

分区分配的另一个分配原则是：尽可能使同一 RDD 不同分区内的记录的数量一致。

对于转换操作得到的 RDD，如果是窄依赖，则分区记录数量依赖于父 RDD 中相同编号分区是如何进行数据分配的，如果是 Shuffle 依赖，则分区记录数量依赖于选择的分区器，哈希分区器无法保证数据被平均分配到各个分区，而范围分区器则能做到这一点。这部分内容我会在 1.6 小节中讨论。

`parallelize` 方法通过把输入的数组做一次平均分配，尝试着让每个分区的记录个数尽可能大致相同（见 `ParallelCollectionRDD` 类）。

```
private object ParallelCollectionRDD {
  /**
   * Slice a collection into numSlices sub-collections. One extra t
   * collections specially, encoding the slices as other Ranges to
   * it efficient to run Spark over RDDs representing large sets of
   * is an inclusive Range, we use inclusive range for the last sli
   */
  def slice[T: ClassTag](seq: Seq[T], numSlices: Int): Seq[Seq[T]]
    if (numSlices < 1) {
      throw new IllegalArgumentException("Positive number of slices
    }
    // Sequences need to be sliced at the same set of index positio
    // like RDD.zip() to behave as expected
    def positions(length: Long, numSlices: Int): Iterator[(Int, Int
      (0 until numSlices).iterator.map(i => {
        val start = ((i * length) / numSlices).toInt
        val end = (((i + 1) * length) / numSlices).toInt
        (start, end)
      })
    }
    seq match {
      case r: Range => {
        positions(r.length, numSlices).zipWithIndex.map({ case ((st
          // If the range is inclusive, use inclusive range for the
          if (r.isInclusive && index == numSlices - 1) {
            new Range.Inclusive(r.start + start * r.step, r.end, r.
          }
```

```
            else {
              new Range(r.start + start * r.step, r.start + end * r.s
            }
          }).toSeq.asInstanceOf[Seq[Seq[T]]]
        }
        case nr: NumericRange[_] => {
          // For ranges of Long, Double, BigInteger, etc
          val slices = new ArrayBuffer[Seq[T]](numSlices)
          var r = nr
          for ((start, end) <- positions(nr.length, numSlices)) {
            val sliceSize = end - start
            slices += r.take(sliceSize).asInstanceOf[Seq[T]]
            r = r.drop(sliceSize)
          }
          slices
        }
        case _ => {
          val array = seq.toArray // To prevent O(n^2) operations for
          positions(array.length, numSlices).map({
            case (start, end) =>
              array.slice(start, end).toSeq
          }).toSeq
        }
      }
    }
  }
```

`textFile` 方法分区内数据的大小则是由 Hadoop API 接口 `FileInputFormat.getSplits` 方法决定（见 `HadoopRDD` 类），得到的每一个分片即为 RDD 的一个分区，分片内数据的大小会受文件大小、文件是否可分割、HDFS 中块大小等因素的影响，但总体而言会是比较均衡的分配。

```
override def getPartitions: Array[Partition] = {
  val jobConf = getJobConf()
  // add the credentials here as this can be called before SparkC
  SparkHadoopUtil.get.addCredentials(jobConf)
  val inputFormat = getInputFormat(jobConf)
  if (inputFormat.isInstanceOf[Configurable]) {
    inputFormat.asInstanceOf[Configurable].setConf(jobConf)
  }
  val inputSplits = inputFormat.getSplits(jobConf, minPartitions)
  val array = new Array[Partition](inputSplits.size)
  for (i <- 0 until inputSplits.size) {
    array(i) = new HadoopPartition(id, i, inputSplits(i))
  }
  array
}
```

参考资料

1. Spark Configuration - Spark 1.2.0 Documentation
2. FileInputFormat (Apache Hadoop Main 2.6.0 API)
3. Spark：RDD 理解

RDD 依赖

依赖与 RDD

RDD 的容错机制是通过记录更新来实现的，且记录的是粗粒度的转换操作。在外部，我们将记录的信息称为血统（**Lineage**）关系，而到了源码级别，Apache Spark 记录的则是 RDD 之间的依赖（**Dependency**）关系。在一次转换操作中，创建得到的新 RDD 称为子 RDD，提供数据的 RDD 称为父 RDD，父 RDD 可能会存在多个，我们把子 RDD 与父 RDD 之间的关系称为依赖关系，或者可以说是子 RDD 依赖于父 RDD。

依赖只保存父 RDD 信息，转换操作的其他信息，如数据处理函数，会在创建 RDD 时候，保存在新的 RDD 内。依赖在 Apache Spark 源码中的对应实现是 `Dependency` 抽象类。

```
/**
 * :: DeveloperApi ::
 * Base class for dependencies.
 */
@DeveloperApi
abstract class Dependency[T] extends Serializable {
  def rdd: RDD[T]
}
```

每个 `Dependency` 子类内部都会存储一个 `RDD` 对象，对应一个父 RDD，如果一次转换转换操作有多个父 RDD，就会对应产生多个 `Dependency` 对象，所有的 `Dependency` 对象存储在子 RDD 内部，通过遍历 RDD 内部的 `Dependency` 对象，就能获取该 RDD 所有依赖的父 RDD。

一些思考

RDD 被设计成内部数据不可改变和粗粒度转换，一个很主要的原因就是为了方便跟踪不同版本的数据集的依赖关系。但在我看来，即使没有这两个特性，RDD 应该也能记录依赖关系：

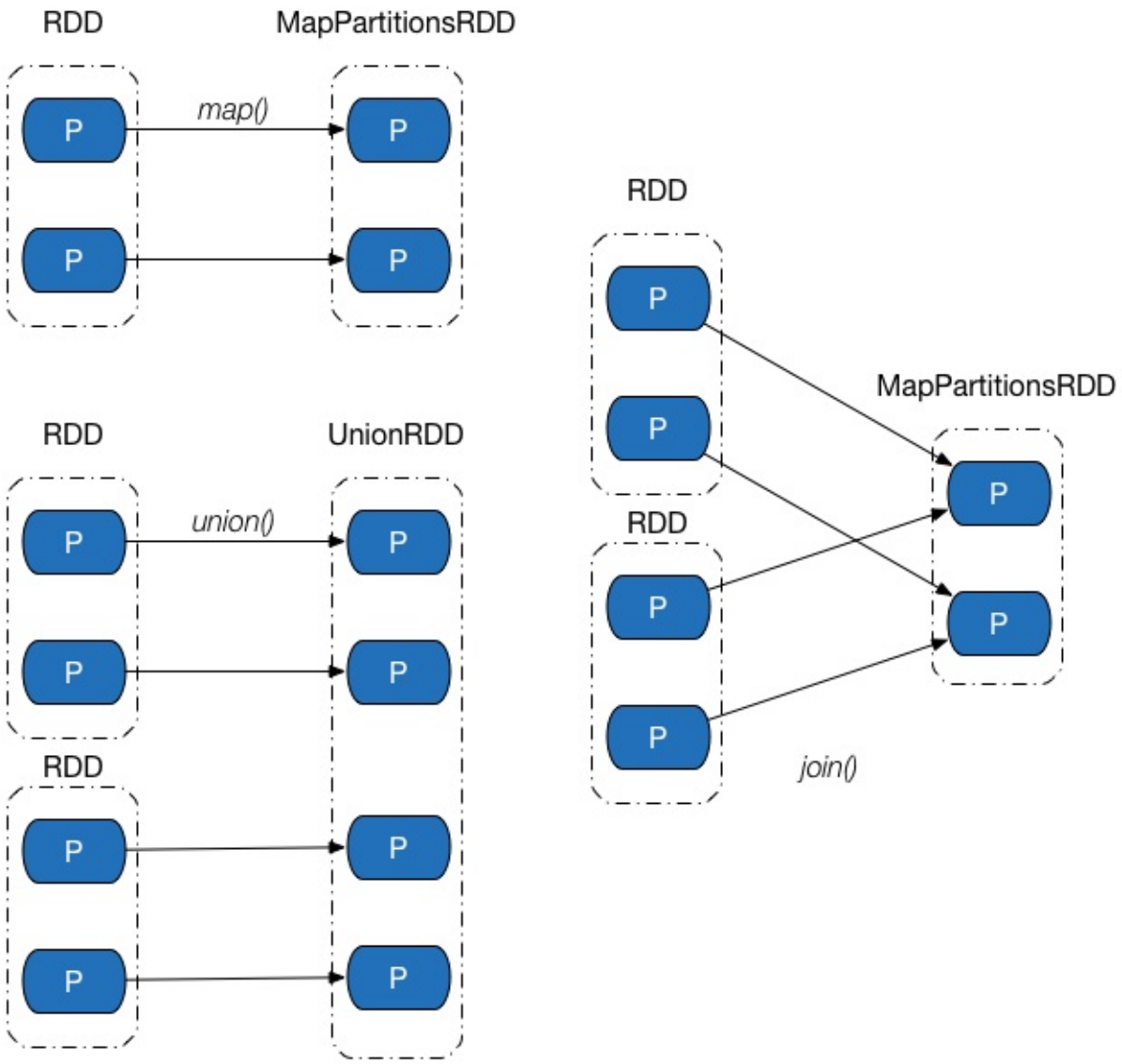
1. 若是数据可变，在单个 RDD 之内，保存所有的变动记录。
2. 若是细粒度，保存每一个数据变动时所使用的操作函数。

如此同样能够实现数据的并行计算和容错机制，但需要存储的数据量（单个 RDD 存储的数据量实在是太大了）、实现的复杂度都会大幅度增加，且很大程度上，没有这个必要，起码 Apache Spark 目前能够胜任大多数数据处理工作，提供的接口也比 Apache Hadoop 的 MR 要高层不少。

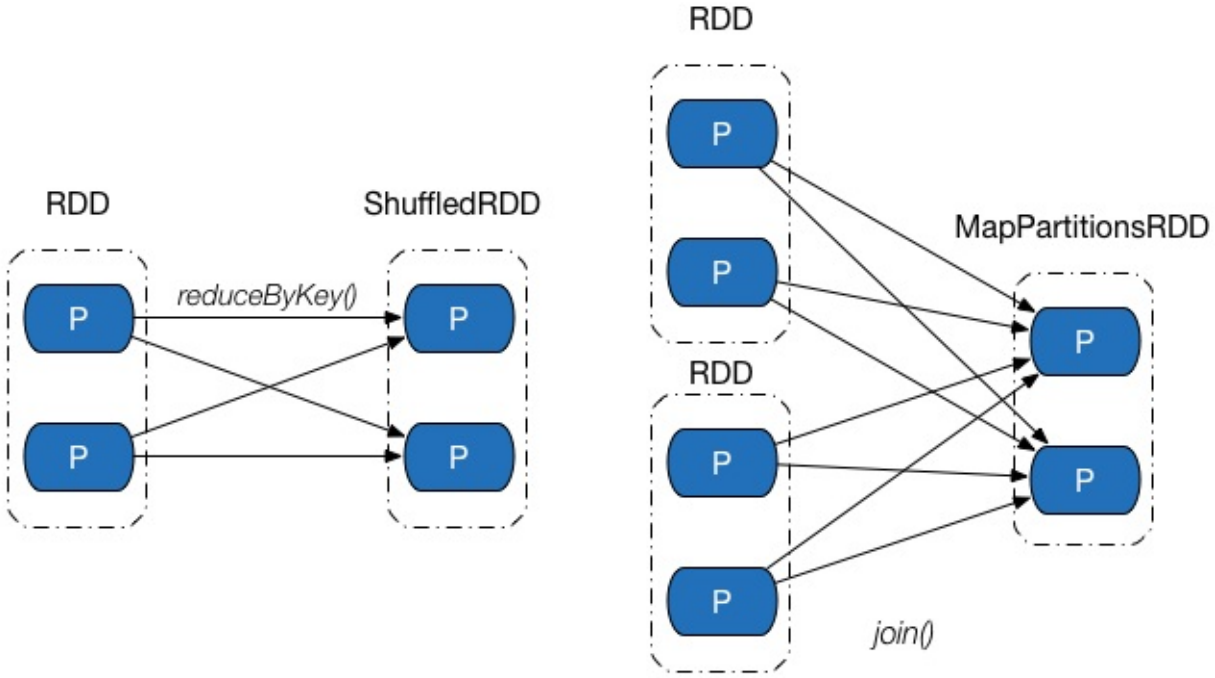
依赖分类

Apache Spark 将依赖进一步分为两类，分别是窄依赖（**Narrow Dependency**）和 **Shuffle** 依赖（**Shuffle Dependency**，在部分文献中也被称为 **Wide Dependency**，即宽依赖）。

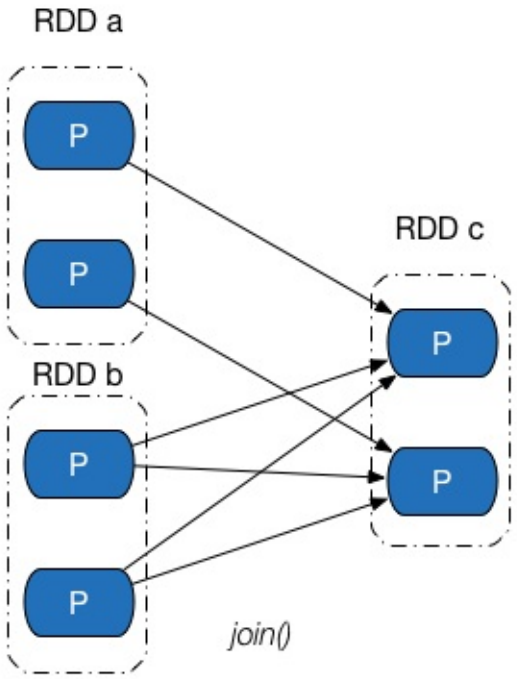
窄依赖中，父 RDD 中的一个分区最多只会被子 RDD 中的一个分区使用，换句话说，父 RDD 中，一个分区内的数据是不能被分割的，必须整个交付给子 RDD 中的一个分区。下图展示了几类常见的窄依赖及其对应的转换操作。



Shuffle 依赖中，父 RDD 中的分区可能会被多个子 RDD 分区使用。因为父 RDD 中一个分区内的数据会被分割，发送给子 RDD 的所有分区，因此 Shuffle 依赖也意味着父 RDD 与子 RDD 之间存在着 Shuffle 过程。下图展示了几类常见的Shuffle依赖及其对应的转换操作。



依赖关系是两个 RDD 之间的依赖，因此若一次转换操作中父 RDD 有多个，则可能会同时包含窄依赖和 Shuffle 依赖，下图所示的 `Join` 操作，RDD a 和 RDD c 采用了相同的分区器，两个 RDD 之间是窄依赖，Rdd b 的分区器与 RDD c 不同，因此它们之间是 Shuffle 依赖，具体实现可参见 `CoGroupedRDD` 类的 `getDependencies` 方法。这里能够再次发现：一个依赖对应的是两个 **RDD**，而不是一次转换操作。



```
  override def getDependencies: Seq[Dependency[_]] = {
    rdds.map { rdd: RDD[_ <: Product2[K, _]] =>
      /* I: Partitioner 相同，则是 OneToOneDepdencency */
      if (rdd.partitioner == Some(part)) {
        logDebug("Adding one-to-one dependency with " + rdd)
        new OneToOneDependency(rdd)
      } else {
        /* I: Partitioner 不同，则是 ShuffleDependency */
        logDebug("Adding shuffle dependency with " + rdd)
        new ShuffleDependency[K, Any, CoGroupCombiner](rdd, part, s
      }
    }
  }
```

窄依赖

窄依赖的实现在 `NarrowDependency` 抽象类中。

```
/**
 * :: DeveloperApi ::
 * Base class for dependencies where each partition of the child RD
 * of partitions of the parent RDD. Narrow dependencies allow for p
 */
@DeveloperApi
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency
  /**
   * Get the parent partitions for a child partition.
   * @param partitionId a partition of the child RDD
   * @return the partitions of the parent RDD that the child partit
   */
  def getParents(partitionId: Int): Seq[Int]

  override def rdd: RDD[T] = _rdd
}
```

`NarrowDependency` 要求子类实现 `getParent` 方法，用于获取一个分区数据来源于父 RDD 中的哪些分区（虽然要求返回 `Seq[Int]`，实际上却只有一个元素）。

窄依赖可进一步分类成一对一依赖和范围依赖，对应实现分别是 `OneToOneDependency` 类和 `RangeDependency` 类。

一对一依赖

一对一依赖表示子 RDD 分区的编号与父 RDD 分区的编号完全一致的情况，若两个 RDD 之间存在着一对一依赖，则子 RDD 的分区个数、分区内记录的个数都将继承自父 RDD。

一对一依赖的实现很简单，如下所示。

```
/**
 * :: DeveloperApi ::
 * Represents a one-to-one dependency between partitions of the par
 */
@DeveloperApi
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T
  override def getParents(partitionId: Int) = List(partitionId)
}
```

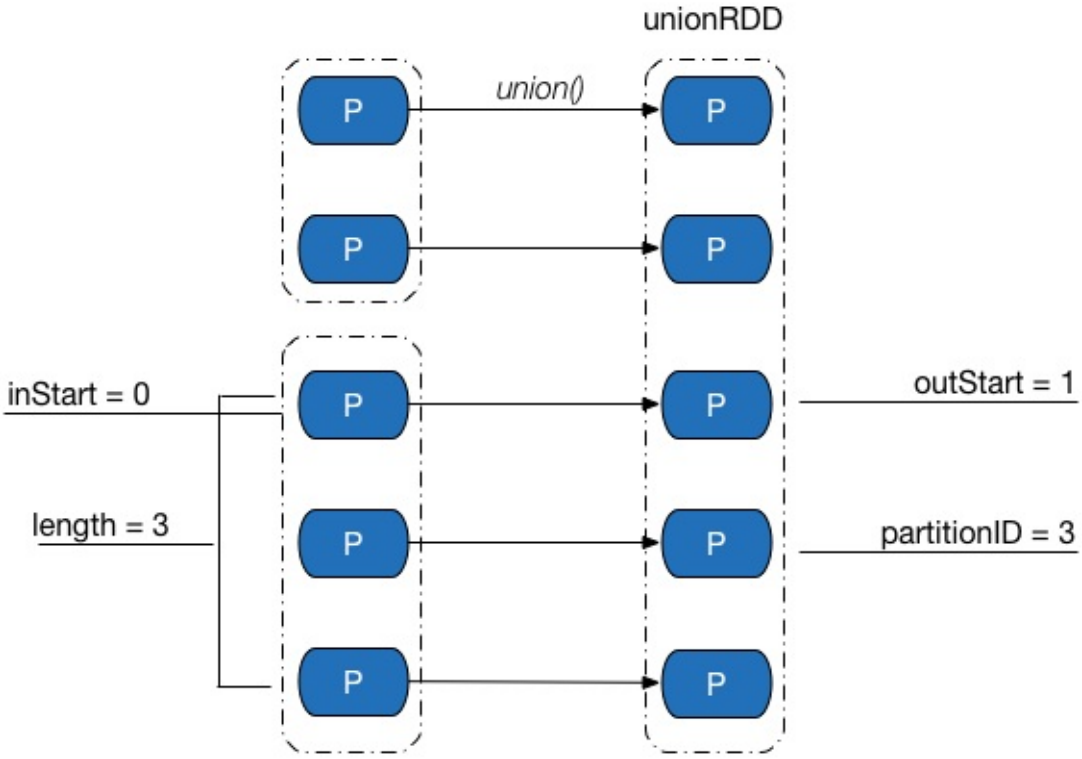
范围依赖

范围依赖是依赖关系中的一个特例，只被用于表示 `UnionRDD` 与父 RDD 之间的依赖关系。相比一对一依赖，除了第一个父 RDD，其他父 RDD 和子 RDD 的分区编号不再一致，Apache Spark 统一将 `unionRDD` 与父 RDD 之间（包含第一个 RDD）的关系都叫做范围依赖。范围依赖的实现如下。

```
/**
 * :: DeveloperApi ::
 * Represents a one-to-one dependency between ranges of partitions
 * @param rdd the parent RDD
 * @param inStart the start of the range in the parent RDD
 * @param outStart the start of the range in the child RDD
 * @param length the length of the range
 */
@DeveloperApi
  class RangeDependency[T](rdd: RDD[T], inStart: Int, outStart: Int
  extends NarrowDependency[T](rdd) {

  override def getParents(partitionId: Int) = {
    if (partitionId >= outStart && partitionId < outStart + length)
      List(partitionId - outStart + inStart)
    } else {
      Nil
    }
  }
}
```

`RangeDepdencency` 类中 `getParents` 的一个示例如下图所示，对于 `UnionRDD` 中编号为 3 的分区，可以计算得到其数据来源于父 RDD 中编号为 2 的分区。



Shuffle 依赖

Shuffle 依赖的对应实现为 `ShuffleDependency` 类，其源码如下。

```
/**
 * :: DeveloperApi ::
 * Represents a dependency on the output of a shuffle stage. Note t
 * the RDD is transient since we don't need it on the executor side
 *
 * @param _rdd the parent RDD
 * @param partitioner partitioner used to partition the shuffle out
 * @param serializer [[org.apache.spark.serializer.Serializer Seria
 *                   the default serializer, as specified by `spark
 *                   be used.
 */
@DeveloperApi
class ShuffleDependency[K, V, C](
    @transient _rdd: RDD[_ <: Product2[K, V]],
val partitioner: Partitioner,
val serializer: Option[Serializer] = None,
    val keyOrdering: Option[Ordering[K]] = None,
    val aggregator: Option[Aggregator[K, V, C]] = None,
    val mapSideCombine: Boolean = false)
  extends Dependency[Product2[K, V]] {

  override def rdd = _rdd.asInstanceOf[RDD[Product2[K, V]]]

  val shuffleId: Int = _rdd.context.newShuffleId()

  val shuffleHandle: ShuffleHandle = _rdd.context.env.shuffleManage
      shuffleId, _rdd.partitions.size, this)

  _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(thi
}
```

`ShuffleDependency` 类中几个成员的作用如下：

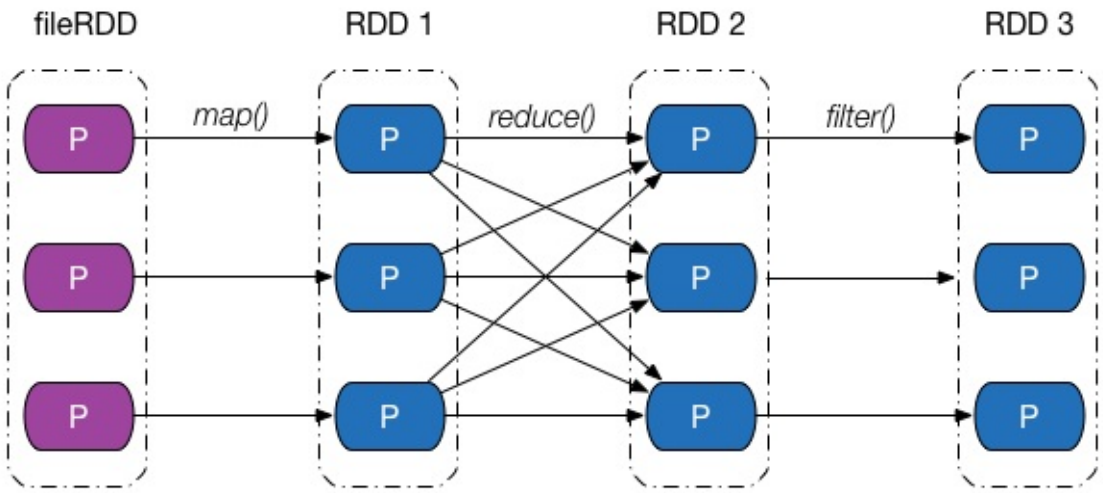
- `rdd`：用于表示 Shuffle 依赖中，子 RDD 所依赖的父 RDD。
- `shuffleId`：Shuffle 的 ID 编号，在一个 Spark 应用程序中，每个 Shuffle 的编号都是唯一的。
- `shuffleHandle`：Shuffle 句柄，`ShuffleHandle` 内部一般包含 Shuffle ID、Mapper 的个数以及对应的 Shuffle 依赖，在执行 `ShuffleMapTask` 时

候，任务可以通过 `ShuffleManager` 获取得到该句柄，并进一步得到 Shuffle 相关信息。

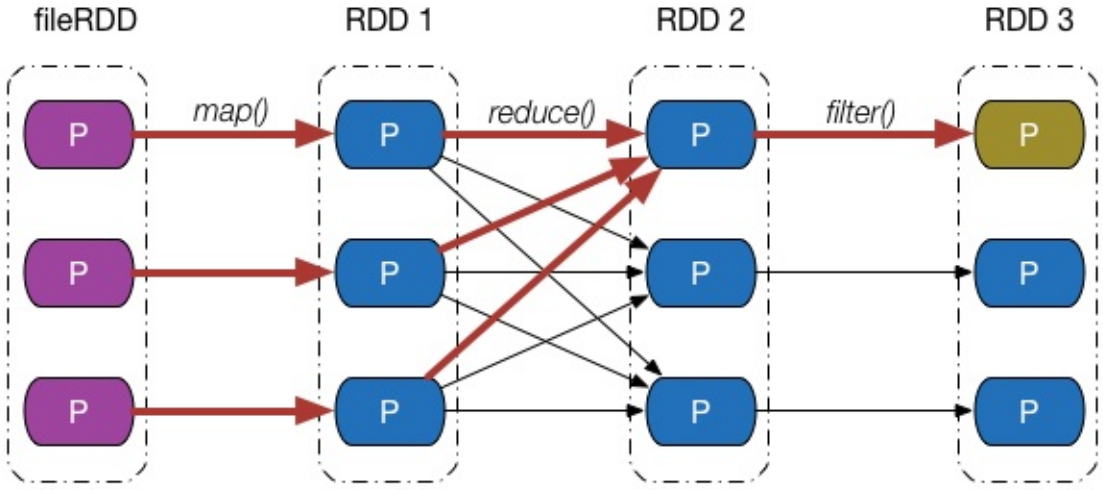
- `partitioner` ：分区器，用于决定 Shuffle 过程中 Reducer 的个数（实际上是子 RDD 的分区个数）以及 Map 端的一条数据记录应该分配给哪一个 Reducer，也可以被用在 `CoGroupedRDD` 中，确定父 RDD 与子 RDD 之间的依赖关系类型。
- `serializer` ：序列化器。用于 Shuffle 过程中 Map 端数据的序列化和 Reduce 端数据的反序列化。
- `KeyOrdering` ：键值排序策略，用于决定子 RDD 的一个分区内，如何根据键值对 类型数据记录进行排序。
- `Aggregator` ：聚合器，内部包含了多个聚合函数，比较重要的函数有 `createCombiner：V => C` ， `mergeValue: (C, V) => C` 以及 `mergeCombiners: (C, C) => C` 。例如，对于 `groupByKey` 操作， `createCombiner` 表示把第一个元素放入到集合中， `mergeValue` 表示一个元素添加到集合中， `mergeCombiners` 表示把两个集合进行合并。这些函数被用于 Shuffle 过程中数据的聚合。
- `mapSideCombine` ：用于指定 Shuffle 过程中是否需要在 map 端进行 combine 操作。如果指定该值为 `true` ，由于 combine 操作需要用到聚合器中的相关聚合函数，因此 `Aggregator` 不能为空，否则 Apache Spark 会抛出异常。例如： `groupByKey` 转换操作对应的 `ShuffleDependency` 中， `mapSideCombine = false` ，而 `reduceByKey` 转换操作中， `mapSideCombine = true` 。

依赖与容错机制

介绍完依赖的类别和实现之后，回过头来，从分区的角度继续探究 Apache Spark 是如何通过依赖关系来实现容错机制的。下图给出了一张依赖关系图， `fileRDD` 经历了 `map` 、 `reduce` 以及 `filter` 三次转换操作，得到了最终的 RDD，其中， `map` 、 `filter` 操作对应的依赖为窄依赖， `reduce` 操作对应的是 Shuffle 依赖。



假设最终 RDD 第一块分区内的数据因为某些原因丢失了，由于 RDD 内的每一个分区都会记录其对应的父 RDD 分区的信息，因此沿着下图所示的依赖关系往回走，我们就能找到该分区数据最终来源于 `fileRDD` 的所有分区，再沿着依赖关系往后计算路径中的每一个分区数据，即可得到丢失的分区数据。



这个例子并不是特别严谨，按照我们的思维，只有执行了持久化，存储在存储介质中的 RDD 分区才会出现数据丢失的情况，但是上例中最终的 RDD 并没有执行持久化操作。事实上，Apache Spark 将没有被持久化数据重新被计算，以及持久化的数据第一次被计算，也等价视为数据“丢失”，在 1.7 节中我们会看到这一点。

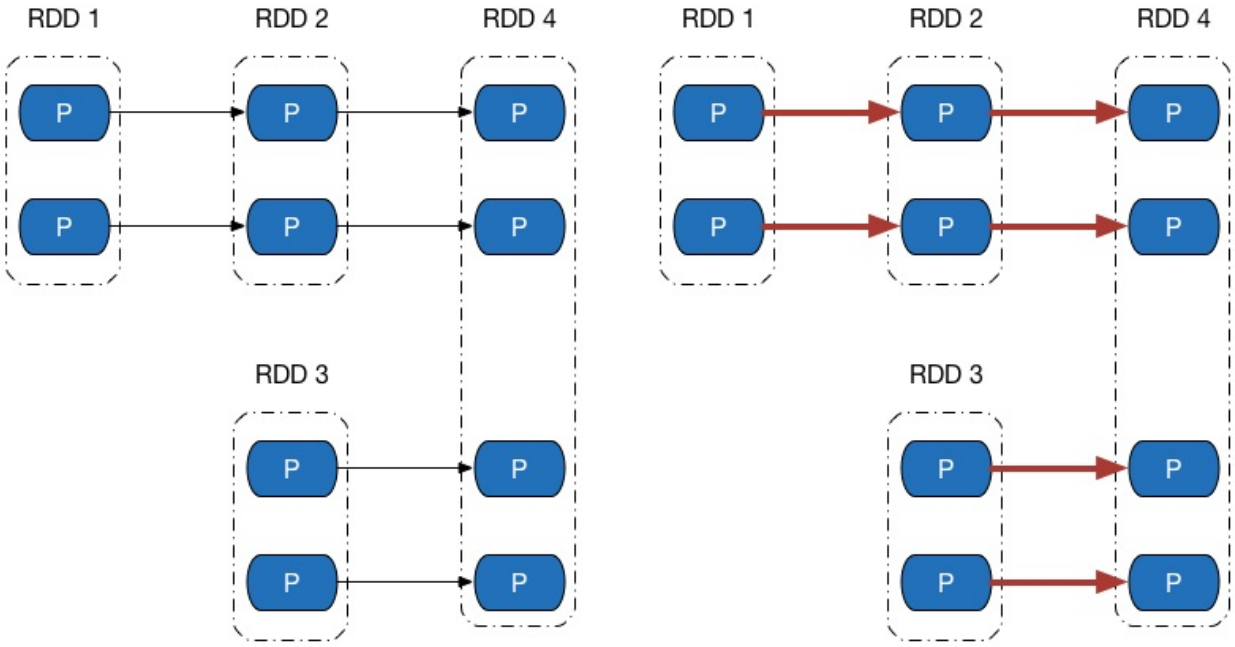
依赖与并行计算

在上一节中我们看到，在 RDD 中，可以通过计算链（**Computing Chain**）来计算某个 RDD 分区内的数据，我们也知道分区是并行计算的基本单位，这时候可能会有一种想法：能否把 RDD 每个分区内数据的计算当成一个并行任务，每个并行任

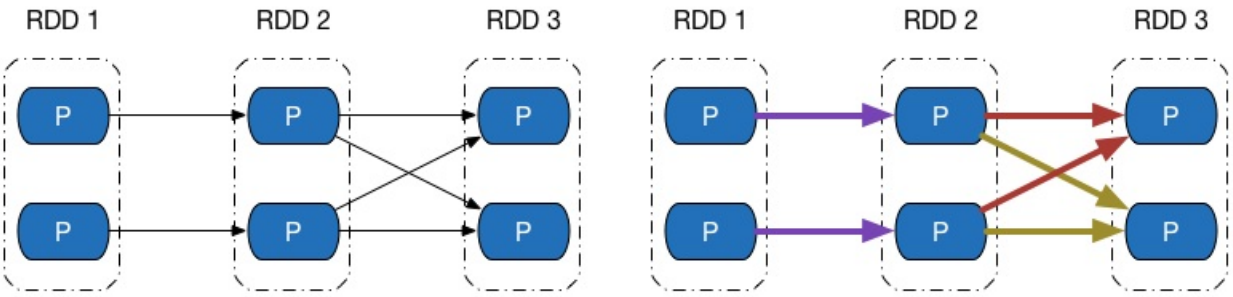
务包含一个计算链，将一个计算链交付给一个 CPU 核心去执行，集群中的 CPU 核心一起把 RDD 内的所有分区计算出来。

答案是可以，这得益于 RDD 内部分区的数据依赖相互之间并不会干扰，而 Apache Spark 也是这么做的，但在实现过程中，仍有很多实际问题需要去考虑。进一步观察窄依赖、Shuffle 依赖在做并行计算时候的异同点。

先来看下方左侧的依赖图，依赖图中所有的依赖关系都是窄依赖（包括一对一依赖和范围依赖），可以看到，不仅计算链是独立不干扰的（所以可以并行计算），所有计算链内的每个分区单元的计算工作也不会发生重复，如右侧的图所示。这意味着除非执行了持久化操作，否则计算过程中产生的中间数据我们没有必要保留 —— 因为当前分区的数据只会给计算链中的下一个分区使用，而不用专门保留给其他计算链使用。



再来观察 Shuffle 依赖的计算链，如图下方左侧的图中，既有窄依赖，又有 Shuffle 依赖，由于 Shuffle 依赖中，子 RDD 一个分区的数据依赖于父 RDD 内所有分区的数据，当我们想计算末 RDD 中一个分区的数据时，Shuffle 依赖处需要把父 RDD 所有分区的数据计算出来，如右侧的图所示（紫色表示最后两个分区计算链共同经过的地方） —— 而这些数据，在计算末 RDD 另外一个分区的数据时候，同样会被用到。



如果我们做到计算链的并行计算的话，这就意味着，要么 Shuffle 依赖处父 RDD 的数据在每次需要使用的时候都重复计算一遍，要么想办法把父 RDD 数据保存起来，提供给其余分区的数据计算使用。

Apache Spark 采用的是第二种办法，但保存数据的方法可能与想象中的会有所不同，**Spark** 把计算链从 **Shuffle** 依赖处断开，划分成不同的阶段（**Stage**），阶段之间存在依赖关系（其实就是 Shuffle 依赖），从而可以构建一张不同阶段之间的有向无环图（**DAG**）。这部分的内容我会在调度一章中细述。

参考资料

1. Job 逻辑执行图 | Apache Spark 的设计与实现

计算函数

分区数据计算过程

在依赖小节中，我提及了 RDD 通过计算链来实现容错机制和并行计算，本节进一步研究整个计算过程，建议配合依赖章节最后两小节来看，会有更深入的理解。

RDD 内部，数据的计算是惰性的，一系列转换操作只有在遇到动作操作时候，才会真的去计算 RDD 分区内的数据。在调度章节中，我们会发现，实际计算过程大概是这样的：

1. 根据动作操作来将一个应用程序划分成多个作业。
2. 一个作业经历 DAG 调度和任务调度之后，被划分成一个一个的任务，对应 `Task` 类。
3. 任务被分配到不同核心去执行，执行 `Task.run` 。
4. `Task.run` 会调用阶段末 RDD 的 `iterator` 方法，获取该 RDD 某个分区内的数据记录，而 `iterator` 方法有可能会调用 RDD 类的 `compute` 方法来负责父 RDD 与子 RDD 之间的计算逻辑。

整个过程会在调度章节中细述，在此不进行展开，我们只需要知道 Apache Spark 最终会调用 RDD 的 `iterator` 和 `compute` 方法来计算分区数据即可。下面我会分别介绍这两个方法的具体实现。

compute 方法

`RDD` 抽象类要求其所有子类都必须实现 `compute` 方法，该方法接受的参数之一是一个 `Partition` 对象，目的是计算该分区中的数据。

以 `map` 操作生成得到的 `MapPartitionsRDD` 类为例，观察其内部 `compute` 方法的实现。

```
override def compute(split: Partition, context: TaskContext): Itera
    f(context, split.index, firstParent[T].iterator(split, context)
```

其中， `firstParent` 在 `RDD` 抽象类中定义。

```
  /** Returns the first parent RDD */
  protected[spark] def firstParent[U: ClassTag] = {
    dependencies.head.rdd.asInstanceOf[RDD[U]]
  }
```

`MapPartitionsRDD` 类的 `compute` 方法调用当前 RDD 内的第一个父 RDD 的 `iterator` 方法，该方的目的是拉取父 `RDD` 对应分区内的数据。`iterator` 方法会返回一个迭代器对象，迭代器内部存储的每个元素即父 RDD 对应分区内已经计算完毕的数据记录。得到的迭代器作为 `f` 方法的一个参数。`f` 在 `RDD` 类的 `map` 方法中指定，如下所示。

```
  /**
   * Return a new RDD by applying a function to all elements of thi
   */
  def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.m
  }
```

`compute` 方法会将迭代器中的记录一一输入 `f` 方法，得到的新迭代器即为所求分区中的数据。

其他 `RDD` 子类的 `compute` 方法与之类似，在需要用到父 RDD 的分区数据时候，就会调用 `iterator` 方法，然后根据需求在得到的数据之上执行粗粒度的操作。换句话说，**`compute`** 函数负责的是父 **`RDD`** 分区数据到子 **`RDD`** 分区数据的变换逻辑。

iterator方法

`iterator` 方法的实现在 `RDD` 抽象类中。

```
/**
 * Internal method to this RDD; will read from cache if applicabl
 * This should ''not'' be called by users directly, but is availa
 * subclasses of RDD.
 */
final def iterator(split: Partition, context: TaskContext): Itera
  if (storageLevel != StorageLevel.NONE) {
    SparkEnv.get.cacheManager.getOrCompute(this, split, context,
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
```

`iterator` 方法首先检查当前 `RDD` 的存储级别，如果存储级别不为 `None`，说明分区的数据要么已经存储在文件系统当中，要么当前 RDD 曾经执行过 `cache`、`persise` 等持久化操作，因此需要想办法把数据从存储介质中提取出来。

`Iterator` 方法继续调用 `CacheManager` 的 `getOrCompute` 方法。

```
/** Gets or computes an RDD partition. Used by RDD.iterator() when
 def getOrCompute[T](
     rdd: RDD[T],
     partition: Partition,
     context: TaskContext,
     storageLevel: StorageLevel): Iterator[T] = {
   val key = RDDBlockId(rdd.id, partition.index)
   blockManager.get(key) match {
      case Some(blockResult) =>
        // Partition is already materialized, so just return its v
        context.taskMetrics.inputMetrics = Some(blockResult.inputM
        new InterruptibleIterator(context, blockResult.data.asInst
      case None =>
        // 省略部分源码
        val computedValues = rdd.computeOrReadCheckpoint(partitior
        val cachedValues = putInBlockManager(key, computedValues,
        new InterruptibleIterator(context, cachedValues)
   }
   // 省略部分源码
}
```

`getOrCompute` 方法会根据 RDD 编号与分区编号计算得到当前分区在存储层对应的块编号，通过存储层提供的数据读取接口提取出块的数据。这时候会有两种可能情况发生：

- 数据之前已经存储在存储介质当中，可能是数据本身就在存储介质（如读取 HDFS 中的文件创建得到的 RDD）当中，也可能是 RDD 经过持久化操作并经历了一次计算过程。这时候就能成功提取得到数据并将其返回。
- 数据不在存储介质当中，可能是数据已经丢失，或者 RDD 经过持久化操作，但是是当前分区数据是第一次被计算，因此会出现拉取得到数据为 `None` 的情况。这就意味着我们需要计算分区数据，继续调用 `RDD` 类 `computeOrReadCheckpoint` 方法来计算数据，并将计算得到的数据缓存到存储介质中，下次就无需再重复计算。

如果当前RDD的存储级别为 `None`，说明为未经持久化的 `RDD`，需要重新计算 RDD 内的数据，这时候调用 `RDD` 类的 `computeOrReadCheckpoint` 方法，该方法也在持久化 RDD 的分区获取数据失败时被调用。

```
/**
 * Compute an RDD partition or read it from a checkpoint if the R
 */
private[spark] def computeOrReadCheckpoint(split: Partition, cont
  if (isCheckpointed) firstParent[T].iterator(split, context) els
}
```

`computeOrReadCheckpoint` 方法会检查当前 RDD 是否已经被标记成检查点，如果未被标记成检查点，则执行自身的 `compute` 方法来计算分区数据，否则就直接拉取父 RDD 分区内的数据。

需要注意的是，对于标记成检查点的情况，当前 RDD 的父 RDD 不再是原先转换操作中提供数据的父 RDD，而是被 Apache Spark 替换成一个 `CheckpointRDD` 对象，该对象中的数据存放在文件系统中，因此最终该对象会从文件系统中读取数据并返回给 `computeOrReadCheckpoint` 方法，在 1.8 节我会解释这样做的原因。

参考资料

1. Cache 和 Checkpoint 功能 | Apache Spark 的设计与实现

分区器

RDD 分区器

分区器（**Partitioner**）在前面章节中或多或少有所提及。我总结了 RDD 分区器的三个作用，而这三个影响在本质上其实是相互关联的。

1. 决定 Shuffle 过程中 Reducer 的个数（实际上是子 RDD 的分区个数）以及 Map 端的一条数据记录应该分配给哪一个 Reducer。这个应该是最主要的作用。
2. 决定 RDD 的分区数量。例如执行操作 `groupByKey(new HashPartitioner(2))` 所生成的 `ShuffledRDD` 中，分区的数目等于 2。
3. 决定 `CoGroupedRDD` 与父 RDD 之间的依赖关系。这个在依赖小节说过。

由于分区器能够间接决定 RDD 中分区的数量和分区内部数据记录的个数，因此选择合适的分区器能够有效提高并行计算的性能（回忆下分区小节我们提及过的 `spark.default.parallelism` 配置参数）。Apache Spark 内置了两种分区器，分别是哈希分区器（**Hash Partitioner**）和范围分区器（**Range Partitioner**）。

开发者还可以根据实际需求，编写自己的分区器。分区器对应的源码实现是 `Partitioner` 抽象类，`Partitioner` 的子类（包括自定义分区器）需要实现自己的 `getPartition` 函数，用于确定对于某一特定键值的键值对记录，会被分配到子RDD中的哪一个分区。

```
/**
 * An object that defines how the elements in a key-value pair RDD
 * Maps each key to a partition ID, from 0 to `numPartitions - 1`.
 */
abstract class Partitioner extends Serializable {
  def numPartitions: Int
  def getPartition(key: Any): Int
}
```

哈希分区器

哈希分区器的实现在 `HashPartitioner` 中，其 `getPartition` 方法的实现很简单，取键值的 hashCode，除以子 RDD 的分区个数取余即可。

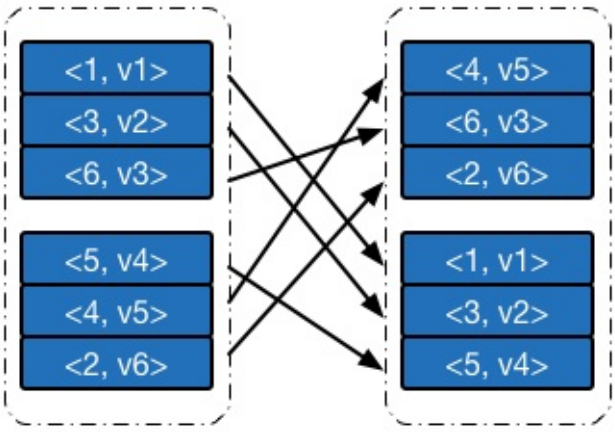
```
/**
 * A [[org.apache.spark.Partitioner]] that implements hash-based pa
 * Java's `Object.hashCode`.
 *
 * Java arrays have hashCodes that are based on the arrays' identit
 * so attempting to partition an RDD[Array[_]] or RDD[(Array[_], _)
 * produce an unexpected or incorrect result.
 */
class HashPartitioner(partitions: Int) extends Partitioner {
  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
}
```

使用哈希分区器进行分区的一个示例 如下图所示。此例中整数的 hashCode 即其本身。

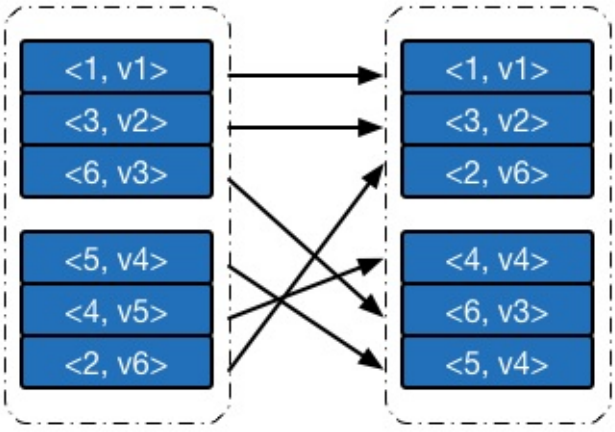


An example of HashPartitioner

范围分区器

哈希分析器的实现简单，运行速度快，但其本身有一明显的缺点：由于不关心键值的分布情况，其散列到不同分区的概率会因数据而异，个别情况下会导致一部分分区分配得到的数据多，一部分则比较少。范围分区器则在一定程度上避免这个问题，范围分区器争取将所有的分区尽可能分配得到相同多的数据，并且所有分区内数据的上界是有序的。使用范围分区器进行分区的一个示例 如下图所示。

如果你自己去测试下面这个例子的话，会发现键值 4 被分配到子 RDD 中的第一个分区，与下图并不一致，这是因为 Apache Spark 1.1 及之后的版本，划分分区边界时候用的是 `>` 而不是 `>=`，后文会细述相应的代码实现。我已经向 Apache Spark 提交了 JIRA 和 PR。



An example of RangePartitioner

范围分区器需要做的事情有两个：根据父 **RDD** 的数据特征，确定子 **RDD** 分区的边界，以及给定一个键值对数据，能够快速根据键值定位其所应该被分配的分区编号。

如果之前有接触过 Apache Hadoop 的 TeraSort 排序算法的话，应该会觉得范围分区器解决的事情与 TeraSort 算法在 Map 端所需要完成的其实是一回事。两者解决问题的思路也是十分类似：对父 RDD 的数据进行采样（**Sampling**），将采样得到的数据排序，并分成 M 个数据块，分割点的键值作为后面快速定位的依据。尽管思路基本一致，但由于 RDD 的一些独有特性，在具体的实现细节上，范围分区器又与 TeraSort 算法有许多不同之处。

1.1 版本之前的范围划分方法

Apache Spark 在 1.1 及之后版本对范围分区器的范围划分算法做了一次比较大的更新（参见 2e6efca）。在具体解读 1.4.1 版本范围分区器的实现之前，我们最好先研究下 1.0 版本是如何进行范围划分的，以便更好地理解新版本如此设计的意义所在。

样本总量

首先我们需要考虑的问题是：总共采样多少数据更合适。样本总量少了，划分得到的范围边界不够具有代表性，样本总数大了，采样、排序的时间会随之增加。Apache Spark 选择的采样总量 `maxSampleSize` 由子 RDD 的分区个数决定，平均每个分区需要 20 的样本。

```
val maxSampleSize = partitions * 20.0
```

单分区采样个数

总量确定后，我们接下来考虑父 RDD 中，每个分区需要采样多少数据。一种思路是每个分区抽取相同数量的样本，但这显然是不合适的，因为 父 **RDD** 分区中的数据不一定是均衡分布的，数据量大的分区应该抽取更多的样本，反之抽取少量的样本。Apache Spark 令每个分区抽取特定比例的数据，采样比例 `frac` 等于采样数据大小 `maxSampleSize` 除以父 RDD 中的数据总量 `rddSize` 。

```
val rddSize = rdd.count()
val frac = math.min(maxSampleSize / math.max(rddSize, 1), 1.0)
```

这里需要注意一点，由于计算 `frac` 需要 RDD 的数据总量 `rddSize`，这个值需要对整个 **RDD** 内部数据进行一次遍历（`count` 转换操作）才能得到。

采样算法

采样比例确定之后，就可以对数据进行采样。Apache Spark 使用的是 `sample` 转换操作进行数据的采样工作，采样完毕后对结果按照键值进行排序。需要注意，采样和排序操作会对 **RDD** 数据各进行一次遍历。

```
val rddSample = rdd.sample(false, frac, 1).map(_._1).collect().sort
if (rddSample.length == 0) {
  Array()
```

边界确定

采样和排序完毕之后，就可以分析样本来确定边界了，确定边界的算法很简单，最后可以得到 `partitions - 1` 个边界。

```
val bounds = new Array[K](partitions - 1)
for (i <- 0 until partitions - 1) {
  val index = (rddSample.length - 1) * (i + 1) / partitions
  bounds(i) = rddSample(index)
}
bounds
```

至此，我们已经完成了边界划分的所有工作。

1.1 版本之后的范围划分方法

在具体阅读 1.4.1 版本范围分区器的源码之前，我曾猜测过其内部实现，得出的方法与上文一致，阅读源码之后发现实际实现和我的猜测有很大的区别，最开始不理解作者的用意，直到后来看到了提交记录 2e6efca 以及对应的 JIRA Issue SPARK-

2568，才恍然大悟。2e6efca 中提到：

> As of Spark 1.0, RangePartitioner goes through data twice: once to compute the count and once to do sampling. As a result, to do sortByKey, Spark goes through data 3 times (once to count, once to sample, and once to sort).
>
> `RangePartitioner` should go through data only once, collecting samples from input partitions as well as counting. If the data is balanced, this should give us a good sketch. If we see big partitions, we re-sample from them in order to collect enough items.
>
> The downside is that we need to collect more from each partition in the first pass. An alternative solution is caching the intermediate result and decide whether to fetch the data after.

可以看到，无论父 RDD 中分区数据平衡与否，不考虑排序的话，划分一次范围就要遍历两次 RDD 数据。我们在之前提及过，让每个分区抽取相同数量的样本是不合适的，但这需要视情况而定，若数据均匀分布，实际上每个分区抽取的样本量都是将近的，平均分配是合适的，所以我们可以采取这种策略：先假设所有数据分配到各个分区的，每个分区抽取相同的数据，同时统计每个分区的实际数据量，判断是否出现数据偏移的情况，若有，则对个别分区（而不是全部）分区重新采样。这种情况下，若数据均匀分布，遍历次数为 1，若不均衡，则只对部分分区重新采样，遍历次数大于 1 小于等于 2。

我们按照之前的思路，配合最开始的分区例子，再来看看新版的范围划分方法与之前有何区别。

样本总量

新版本的范围分区器的预计采样总数依旧是平均每个分区需要 20 的样本，但增加了最大限制为一百万个数据。在我们的例子中，总共需要采集 `20 * 2 = 40` 个样本。

```
// This is the sample size we need to have roughly balanced output
val sampleSize = math.min(20.0 * partitions, 1e6)
```

单分区采样个数

理论上，父 RDD 平均每个分区需要采样的个数是 `sampleSize / rdd.partitions.size`，然而在新版本中采样个数却是理论值的 3 倍，3 这个值是有实际的意义存在，范围分区器的作者认为 若一个分区的数据个数大于平均值的 **3** 倍，那么就认为出现了严重的数据偏差。

在后面我们会看到，若检测到一个分区的数据太多了，就对对其进行重新采样，而数据个数在平均值 **3** 倍以内的分区，不再进行调整，为了保证数据个数在平均值的 **1** 倍和 **3** 倍之间的数据能够采集到足够多的数据，就在默认情况下，对这些分区多采样 **3** 倍于期望值的数据，来保证总体样本量是足够的。在我们的例子中，平均每个分区采样的数量等于 `math.ceil(3.0 * 40 / 2) = 60` 个。

```
// Assume the input partitions are roughly balanced and over-sample
val sampleSizePerPartition = math.ceil(3.0 * sampleSize / rdd.parti
```

采样算法

在确定单个分区的采样个数之后，就可以对数据进行采样了，但在这时候，我们会发现 —— 单个分区内数据的个数我们是不知道的，换句话说，我们还无法获知抽样比例，因此无法调用 `sample` 转换操作来进行抽样，这时候如果先对分区进行遍历，获取分区内数据个数之后再进行采样的话，效率会变得同老版本一样低下，因此新版本 Apache Spark 使用的是水塘采样法来进行数据抽样。

水塘采样法是一种在线抽样法，能在不知道样本总量或者样本数量太大导致无法载入内存的情况下，实现等概率抽样。具体算法可以参看维基百科上的相关词条，我就不在这边详细说明了。

```
val (numItems, sketched) = RangePartitioner.sketch(rdd.map(_._1), s
```

`sketch` 方法用于数据的采样，返回结果中，`numItems` 表示 RDD 所有数据的个数（等价于之前的 `rddSize`），`sketched` 是一个迭代器，每个元素是一个三元组 `(idx, n, sample)`，其中 `idx` 是分区编号，`n` 是分区的数据个数（而不是采样个数），`sample` 是一个数组，存储该分区采样得到的样本数据。

```
  /**
   * Sketches the input RDD via reservoir sampling on each partitio
   *
   * @param rdd the input RDD to sketch
   * @param sampleSizePerPartition max sample size per partition
   * @return (total number of items, an array of (partitionId, numb
   */
  def sketch[K : ClassTag](
      rdd: RDD[K],
      sampleSizePerPartition: Int): (Long, Array[(Int, Int, Array[K
    val shift = rdd.id
    // val classTagK = classTag[K] // to avoid serializing the enti
    val sketched = rdd.mapPartitionsWithIndex { (idx, iter) =>
      val seed = byteswap32(idx ^ (shift << 16))
      val (sample, n) = SamplingUtils.reservoirSampleAndCount(
        iter, sampleSizePerPartition, seed)
      Iterator((idx, n, sample))
    }.collect()
    val numItems = sketched.map(_._2.toLong).sum
    (numItems, sketched)
  }
```

水塘采样法的实现在 `SamplingUtils.reservoirSampleAndCount` 方法中。

```
  /**
   * Reservoir sampling implementation that also returns the input
   *
   * @param input input size
   * @param k reservoir size
   * @param seed random seed
   * @return (samples, input size)
   */
  def reservoirSampleAndCount[T: ClassTag](
      input: Iterator[T],
      k: Int,
      seed: Long = Random.nextLong())
    : (Array[T], Int) = {
    val reservoir = new Array[T](k) /* 鱼塘，k 项 */
```

```
      // Put the first k elements in the reservoir.
      var i = 0
      while (i < k && input.hasNext) {
        val item = input.next()
        reservoir(i) = item
        i += 1
      }

      // If we have consumed all the elements, return them. Otherwise
      if (i < k) {
        // If input size < k, trim the array to return only an array
        val trimReservoir = new Array[T](i)
        System.arraycopy(reservoir, 0, trimReservoir, 0, i)
        (trimReservoir, i)
      } else {
        // If input size > k, continue the sampling process.
        val rand = new XORShiftRandom(seed)
        while (input.hasNext) {
          val item = input.next()
          val replacementIndex = rand.nextInt(i)
          if (replacementIndex < k) {
            reservoir(replacementIndex) = item
          }
          i += 1
        }
        (reservoir, i)
      }
    }
```

到此步，采样的过程并没有完全结束，我们说过范围分区器会假设数据是均衡分布的，从而进行第一次采样，但实际上可能并不是，对于分区内数据大于平均值 3 倍的，范围分区器会对其进行一次重采样，采样率等于 `math.min(sampleSize / math.max(numItems, 1L), 1.0)`，实际上就是老版本范围分区器中的采样率，由于这时候 `numItems` 已知，所以可以直接计算得到采样率，并执行 `sample` 转换操作来进行采样。采样后的每一个样本以及该样本采样时候的采样间隔（**1 / 采样率**，记为 **`weight`**）都会放到 `candidates` 数组中。采样过程到此结束。

```
// If a partition contains much more than the average number of ite
// to ensure that enough items are collected from that partition.
val fraction = math.min(sampleSize / math.max(numItems, 1L), 1.0)
val candidates = ArrayBuffer.empty[(K, Float)]
val imbalancedPartitions = mutable.Set.empty[Int]

sketched.foreach { case (idx, n, sample) =>
  /* I: 应该采样的数据比实际采样的数据要大 */
  if (fraction * n > sampleSizePerPartition) {
    imbalancedPartitions += idx
  } else {
    // The weight is 1 over the sampling probability.
    val weight = (n.toDouble / sample.size).toFloat
    for (key <- sample) {
      candidates += ((key, weight))
    }
  }
}
if (imbalancedPartitions.nonEmpty) {
  // Re-sample imbalanced partitions with the desired sampling prob
  val imbalanced = new PartitionPruningRDD(rdd.map(_._1), imbalance
  val seed = byteswap32(-rdd.id - 1)
    val reSampled = imbalanced.sample(withReplacement = false, frac
  val weight = (1.0 / fraction).toFloat
  candidates ++= reSampled.map(x => (x, weight))
}
```

在我们的例子中，由于父 RDD 中分区内数据的个数都小于 60，因此整个分区数据都被当做样本，`numItems` 等于 6，`sketched` 中有两个三元组，分别是 `(0, 3, {1, 3, 6})` 和 `{1, 3, {5, 4, 2}}`，没有出现需要重采样的情况，所以最后的 `candidates` 等于 `{(1, 1.0), (3, 1.0), (6, 1.0), (5, 1.0), (4, 1.0), (2, 1.0)}` 。

边界确定

采样拿到之后，就可以确定边界了。由于每个分区的采样率（或者说是采样间隔）是不一样的，所以不能再使用原来的平均划分法来确定边界，也就是说，每个样本之间不再是等距的了，需要再乘以一个权值，而这个权值正是采样间隔。接下来的

过程就很简单了，唯一需要注意的是，1.4.1 版本中，边界的判断条件是 `cumWeight > target`，可能会导致划分后的分区不大均匀，例如在我们的例子中，划分得到的边界是 `{4}`，这就意味着，小于等于 4 的数据在一个分区内，也就是 `{1, 2, 3, 4}`，其他数据在另一个分区内，也就是 `{5, 6}`，这样显然是不均匀的，而把判断条件改成 `cumWeight >= target` 就可以避免这个问题。

```
  /**
   * Determines the bounds for range partitioning from candidates w
   * items each represents. Usually this is 1 over the probability
   *
   * @param candidates unordered candidates with weights
   * @param partitions number of partitions
   * @return selected bounds
   */
  def determineBounds[K : Ordering : ClassTag](
      candidates: ArrayBuffer[(K, Float)],
      partitions: Int): Array[K] = {
    val ordering = implicitly[Ordering[K]]
    val ordered = candidates.sortBy(_._1)
    val numCandidates = ordered.size
    val sumWeights = ordered.map(_._2.toDouble).sum
    val step = sumWeights / partitions
    var cumWeight = 0.0
    var target = step
    val bounds = ArrayBuffer.empty[K]
    var i = 0
    var j = 0
    var previousBound = Option.empty[K]
    while ((i < numCandidates) && (j < partitions - 1)) {
      val (key, weight) = ordered(i)
      cumWeight += weight
      if (cumWeight > target) {
        // Skip duplicate values.
        if (previousBound.isEmpty || ordering.gt(key, previousBound
          bounds += key
          target += step
          j += 1
          previousBound = Some(key)
        }
      }
      i += 1
    }
    bounds.toArray
  }
```

快速定位

无论是新版本还是老版本的范围分区器，使用的定位方法都是一样的。范围分区器的定位实现在 `getPartition` 方法内，若边界的数量小于 128，则直接遍历，否则使用二叉查找法来查找合适的分区编号。

```
  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, range
        partition += 1
      }
    } else {
      // Determine which binary search method to use only once.
      partition = binarySearch(rangeBounds, k)
      // binarySearch either returns the match location or -[insert
      if (partition < 0) {
        partition = -partition-1
      }
      if (partition > rangeBounds.length) {
        partition = rangeBounds.length
      }
    }
    if (ascending) {
      partition
    } else {
      rangeBounds.length - partition
    }
  }
```

参考资料

1. Reservoir sampling - Wikipedia, the free encyclopedia
2. 董的博客 » Hadoop 中 TeraSort 算法分析
3. Reservoir Sampling 学习笔记 | Sigmainfy 烟客旅人

4. Apache Spark 中的 RangePartitioner 是如何实现数据采样的？ - 知乎

Shuffle 过程

在前面的章节中，我们已经讨论过了在同一个阶段内部数据的流动过程，包括任务的切分和运行，调用 `iterator` 和 `compute` 方法自后向前获取和计算分区数据，等等。

而对于相邻阶段之间的数据传输，我曾在 Shuffle 依赖 小节中有过简单提及：如果希望实现带 Shuffle 依赖的计算链中分区数据（或者说是任务）的并行计算，由于 Shuffle 依赖父 RDD 的同一个分区数据会被子 RDD 的所有分区重复使用，因此为了提高计算效率，当计算父 RDD 的一个分区数据完毕之后，Apache Spark 会把分区数据临时存储在文件系统中，提供给子 RDD 的分区去使用。

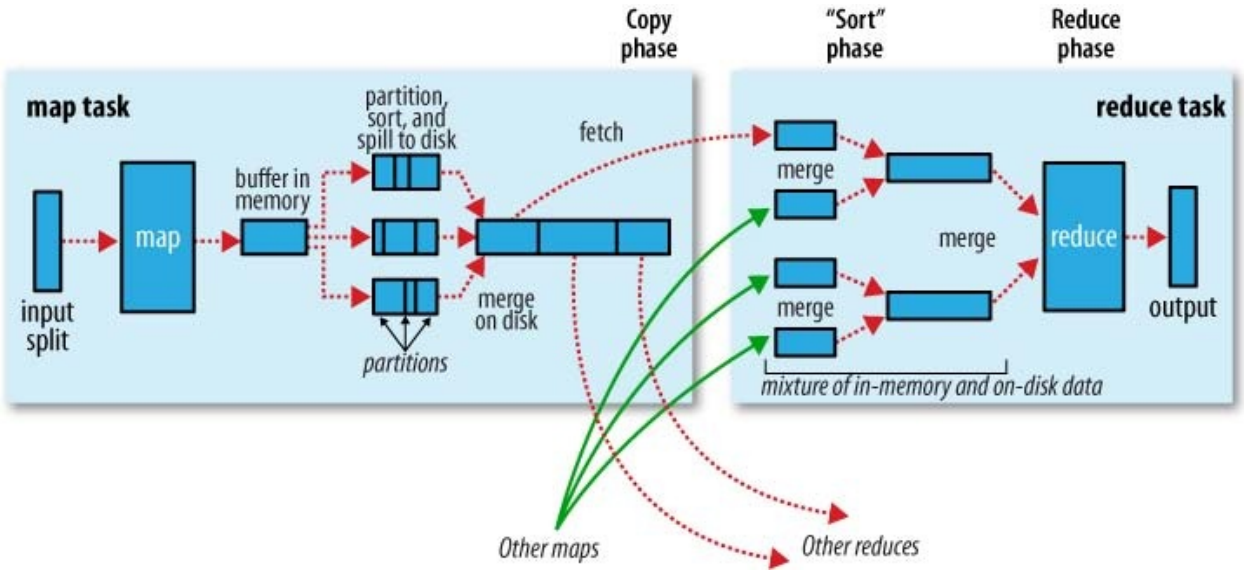
为了符合子阶段需要等待父阶段执行完毕后子阶段才开始执行这一特性，只有当 Shuffle 依赖中父 RDD 所有分区的数据被计算和存储完毕之后，子 RDD 才会开始去拉取需要的分区数据。我们将整个数据传输的过程称为 Apache Spark 的 **Shuffle** 过程。Shuffle 过程中，我们把一个分区数据计算完毕到数据被写入到磁盘的过程，称为 **Shuffle** 写过程。对应的，在子 RDD 某个分区计算的过程中，把所需的数据从父 RDD 拉取过来的过程，称为 **Shuffle** 读过程。

Apache Spark 的 Shuffle 过程与 Apache Hadoop 的 Shuffle 过程有着诸多类似，一些概念可直接套用，例如，Shuffle 过程中，提供数据的一端，被称作 Map 端，Map 端每个生成数据的任务称为 Mapper，对应的，接收数据的一端，被称作 Reduce 端，Reduce 端每个拉取数据的任务称为 Reducer，Shuffle 过程本质上都是将 **Map** 端获得的数据使用分区器进行划分，并将数据发送给对应的 **Reducer** 的过程。

与 Apache Hadoop 的区别

当然，Apache Spark 和 Apache Hadoop 毕竟不是同一套计算框架，其在 Shuffle 过程实现的细节上有着诸多不同。咱们先来温习下经典 MR 模型内的 Shuffle 过程。

MR 模型的 Shuffle 过程



（图像来源：***Hadoop Definitive Guide***）

MR 模型中，每个 Mapper 维护一个环形内存缓冲区，用于存储任务输出，当内存缓冲区数据达到一定阈值时候，将缓冲区中的数据进行分区（**Partition**），对于同一个分区内部的数据按照键值进行排序（**Sort**），如果开发者指定了 `Combiner`，那么对于排序后的结果，还会执行一次合并**(Combine**）操作，最后的结果会被溢存（**Spill**）到磁盘文件中，在任务完成之前，Apache Hadoop 会采用多路归并算法（**K-way Merge Algorithm**）来归并（**Merge**）这几个内部有序的溢存文件，新文件中数据同样是有序的。

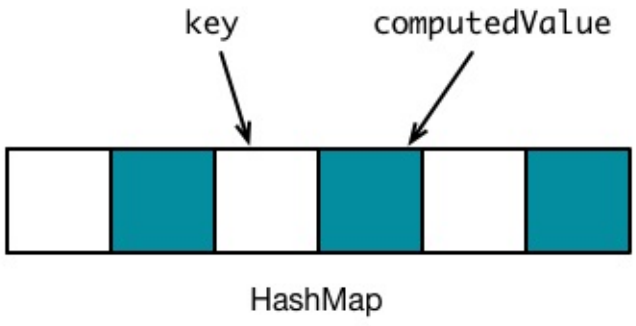
Reducer 需要从 Map 端拉取数据，当一个 Mapper 运行结束之后，会通知 JobTracker，Reducer 定期从 JobTracker 获取相关信息，然后从 Mapper 端获取数据，所有需要的数据复制完毕后，Reducer 会合并来自不同 Map 端拉取过来的数据，并将最后排序好的数据送往 `Reduce` 方法处理。

聚合器

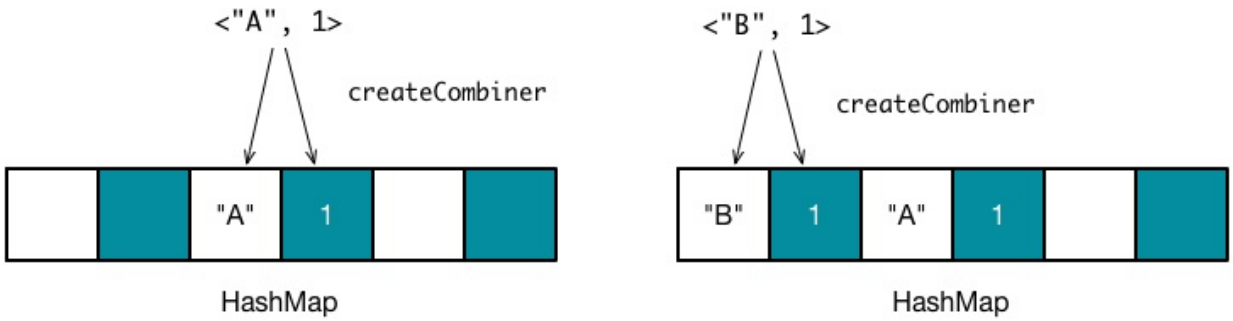
Apache Hadoop 和 Apache Spark 各自的 Shuffle 机制里，比较重要的一个区别在于数据的聚合（**Aggregation**）和聚合数据的计算。对于 Apache Hadoop，聚合是通过对同一分区内的数据按照键值排序，键值相同的数据会彼此相邻，从而达到聚合的目的，而聚合后的数据会被交给 `combine`（Map 端）和 `reduce`（Reduce 端）函数去处理。对于 Apache Spark，聚合以及数据计算的过程，则是交付给聚合器（**Aggregator**）去处理。

聚合器我在 Shuffle 依赖 小节介绍 `ShuffleDependency` 类时候曾经简单提及过，实例化一个聚合器的时候，需要提供三个函数，分别是：`createCombiner: V => C` ， `mergeValue: (C, V) => C` 以及 `mergeCombiners: (C, C) => C` 。我们以单词统计程序中的 `reduceByKey(_ + _)` 转换操作为例，介绍这三个函数究竟是如何实现数据的聚合和计算的。

Apache Spark 会使用哈希表来存储所有聚合数据的处理结果，表中的浅色空槽用于存储键值，右侧相邻深色空槽表示该键值对应的计算值。聚合器开始处理聚合数据之前，哈希表是空的，如下图所示。

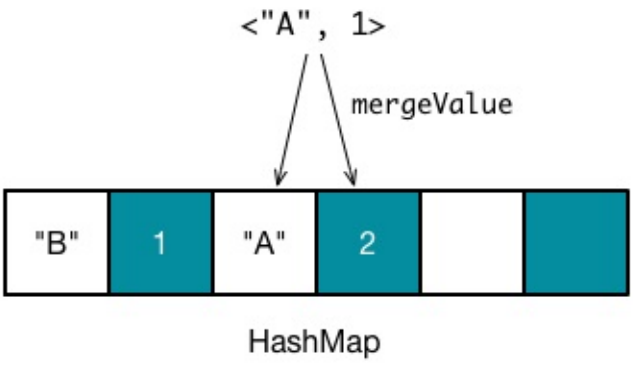


假设需要聚合的数据是 <"A", 1>，<"B", 1>，<"A", 1>，需要注意，这时候数据是无序的。对于第一个数据，<"A", 1>，Apache Spark 会通过散列函数计算键值 "A" 对应的哈希表地址，假设此时得到的哈希值为 1，因此哈希表中，地址为 2 的空槽用于存放键值 "A"，地址 为 3 的空槽用于存放计算后的值。由于地址为 **2** 和地址为 **3** 的槽均为空槽，这时候会调用 **`createCombiner(kv._2)`** 函数来计算初始值。对于 `reduceByKey` 转换操作， `createCombiner` 实际为 `(v: V) => v` ，因此得到计算值为 1，将 "A" 放入到地址为 2 的空槽中，将 1 放入到地址为 3 的插槽中。同理，对于数据 <"B", 1>，可以放入到另外两个空槽中。

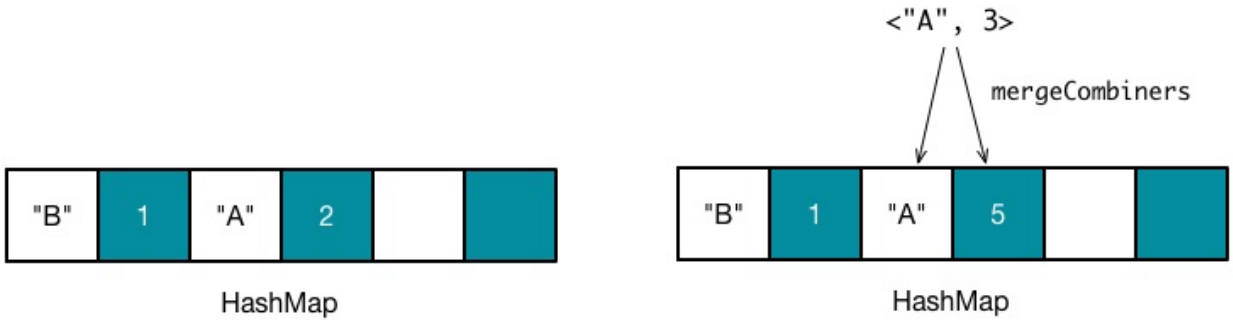


第三个数据是 <"A", 1>，计算得到地址为 1，此时因为地址为 **2** 和 地址为 **3** 的插槽已经有值 **`oldValue`** ，这时候调用 **`mergeValue(oldValue, kv._2)`** 来计算新的值。 对于 `reduceByKey` 转换操作， `mergeValue` 实际上为用户在调用

`reduceBykey` 时候指定的函数，在本例中，该函数为 `_ + _`，因此得到新的值 2，更新地址为 3 的槽。



`reduceByKey` 指定了 `Combiner`，因此会在 Map 端执行结合操作，Reducer 接收到的键值对数据，值的类型会是 `C` 而非 `V`（尽管在本例中，`C` 和 `V` 是相同类型），这时候如果键值对应的槽为空槽，直接插入 **`kc._2`**，否则调用 **`mergeCombiners(oldValue, kc._2)`** 函数来计算新的值。对于 `reduceByKey` 转换操作，`mergeCombiners` 实际为用户在调用 `reduceBykey` 时候指定的函数。



到此为止，数据已经被成功地聚合和计算了，当然在实际的过程中需要考虑的问题还很多，例如哈希表冲突解决、大小分配、内存限制等等。Apache Spark 实现了两类哈希表，分别是 `AppendOnlyMap` 和 `ExternalAppendOnlyMap`，我会在后面专门用一节内容，来介绍这两类哈希表的特性和实现。

接下来，我们思考下其与 MR 机制中聚合 - 计算过程的区别。首先，最明显的区别是，Apache Spark 的聚合 - 计算过程不需要进行任何排序！！！这意味着 Apache Spark 节省了排序所消耗的大量时间，代价是最后得到的分区内部数据是无序的；再者，Apache Spark 的聚合 / 计算过程是同步进行的，聚合完毕，结果也计算出来，而 Apache Hadoop 需要等聚合完成之后，才能开始数据的计算过程；最后，

Apache Spark 将所有的计算操作都限制在了 `createCombiner` 、 `mergeValue` 以及 `mergeCombiners` 之内，在灵活性之上显然要弱于 Apache Hadoop，例如，Apache Spark 很难通过一次聚合 - 计算过程求得平均数。

哈希 Shuffle 与排序 Shuffle

注：本节内容存在些许较为严重的理解错误，因近期求职较忙，一直未来得及修正，希望没有误导到各位读者，实在是抱歉。

在 1.1 之前的版本，Apache Spark 仅提供了哈希 **Shuffle**（**Hash-Based Shuffle**）机制，其实现同我们前面所述的聚合 / 计算过程基本一致，然而如我们上一小节所说的，聚合 / 计算过程后，分区内部的数据是无序的，如果开发者希望有序，就需要调用排序相关的转换操作，例如 `sortBy` 、 `sortByKey` 等等；再者，哈希 Shuffle 强制要求在 Map 端进行 Combine 操作，对于某些键值重复率不高的数据，Combine 操作反倒会影响效率；另外，哈希 Shuffle 每个 Mapper 会针对每个 Reducer 生成一个数据文件，当 Mapper 和 Reducer 数量比较多的时候，会导致磁盘上生成大量的文件（为什么不将所有数据放到一个文件里面，并额外生成一个索引文件用于分区的索引呢？）。

从 1.1 开始，Apache Spark 提供了另一套 Shuffle 机制 —— 排序 **Shuffle**（**Sort-Based Shuffle**），并且从 1.2 版本开始，把排序 Shuffle 作为默认的 Shuffle 机制，用户可以将配置项 `spark.shuffle.manager` 设置为 `hash` 或者 `sort` ，来使用对应的 Shuffle 机制。排序 Shuffle 相比哈希 Shuffle，两者的 Shuffle 读过程是完全一致的，唯一区别在 Shuffle 写过程。

排序 Shuffle 允许 Map 端不进行 `Combine` 操作，这意味着在不指定 `Combiner` 的情况下，排序 Shuffle 机制 Map 端不能使用一张哈希表来存储数据，而是改为用数据缓存区（**Buffer**）存储所有的数据。对于指定 `Combiner` 的情况下，排序 Shuffle 仍然使用哈希表存储数据，Combine 过程与哈希 Shuffle 基本一致。无论是 Buffer 还是 HashMap，每更新一次，都会检查是否需要将现有的数据溢存到磁盘当中，需要的话，就对数据进行排序，存储到一个文件中，当所有的数据都更新完毕之后，执行结合操作，把多个文件合并成一个文件，并且保证每个分区内部数据是有序的。

两类 Shuffle 机制的 Shuffle 读、Shuffle 写过程的实现我会在后面小节中具体讲解。

Shuffle 过程

我们继续从源码的角度，了解 Apache Spark 是如何触发 Shuffle 写和 Shuffle 读过程的。

我们知道，Mapper 本质上就是一个任务。调度章节曾提及过 DAG 调度器会在一个阶段内部划分任务，根据阶段的不同，得到 `ResultTask` 和 `ShuffleMapTask` 两类任务。`ResultTask` 会将计算结果返回给 Driver，`ShuffleMapTask` 则将结果传递给 Shuffle 依赖中的子 RDD。因此我们可以从 `ShuffleMapTask` 入手，观察 Mapper 的大致工作流程。

```
private[spark] class ShuffleMapTask(
    stageId: Int,
    taskBinary: Broadcast[Array[Byte]],
    partition: Partition,
    @transient private var locs: Seq[TaskLocation])
  extends Task[MapStatus](stageId, partition.index) with Logging {
    // 省略部分源码
    override def runTask(context: TaskContext): MapStatus = {
        // Deserialize the RDD using the broadcast variable.
        val deserializeStartTime = System.currentTimeMillis()
        val ser = SparkEnv.get.closureSerializer.newInstance()
        val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency
          ByteBuffer.wrap(taskBinary.value), Thread.currentThread.g
        _executorDeserializeTime = System.currentTimeMillis() - des
        try {
          val manager = SparkEnv.get.shuffleManager
          writer = manager.getWriter[Any, Any](dep.shuffleHandle, p
          writer.write(rdd.iterator(partition, context).asInstanceO
          return writer.stop(success = true).get
        } catch {
          case e: Exception =>
            // 省略部分源码
        }
  }
}
```

由于一个任务对应当前阶段末 RDD 内的一个分区，因此通过 `rdd.iterator(partition, context)` 可以计算得到该分区的数据，这个过程我在 RDD 计算函数 小节中已经介绍过。接下来便是执行 Shuffle 写操作，该操作由一个 `ShuffleWriter` 实例通过调用 `write` 接口完成，Apache Spark 从 `ShuffleManager` 实例中获取该 `ShuffleWriter` 对象。

上文提及过，Apache Spark 提供了两类 Shuffle 机制，对应的，`ShuffleManager` 也有两类子类，分别是 `HashShuffleManager` 和 `SortShuffleManager`，**`ShuffleManager`** 的主要作用是提供 **`ShuffleWriter`** 和 **`ShuffleReader`** 用于 **Shuffle** 写和 **Shuffle** 读过程。`HashShuffleManager` 提供 `HashShuffleWriter` 和 `HashShuffleReader`，而 `SortShffleManager` 提供的是 `SortShuffleWriter` 和 `HashShuffleReader`，可以看到，哈希 **Shuffle** 和排序 **Shuffle** 的唯一区别在于 **Shuffle** 写过程，读过程完全一致。

继续来观察 Shuffle 读的触发。Apache Spark 中，聚合器中三个函数是在 `PairRDDFunctions.combineByKey` 方法中指定。可以看到，若新 RDD 与旧 RDD 的分区器不同时，会生成一个 `ShuffledRDD`。

```
def combineByKey[C](createCombiner: V => C,
    mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C,
    partitioner: Partitioner,
    mapSideCombine: Boolean = true,
    serializer: Serializer = null): RDD[(K, C)] = self.withScope
  // 省略部分代码
  val aggregator = new Aggregator[K, V, C](
    self.context.clean(createCombiner),
    self.context.clean(mergeValue),
    self.context.clean(mergeCombiners))
  if (self.partitioner == Some(partitioner)) {
    self.mapPartitions(iter => {
      val context = TaskContext.get()
      new InterruptibleIterator(context, aggregator.combineValues
    }, preservesPartitioning = true)
  } else {
    new ShuffledRDD[K, V, C](self, partitioner)
      .setSerializer(serializer)
      .setAggregator(aggregator)
      .setMapSideCombine(mapSideCombine)
  }
}
```

观察 `ShuffledRDD` 是如何获取分区数据的。与 Shuffle 写过程类似，先从 `ShuffleManager` 中获取 `ShuffleReader`，通过 `ShuffleReader` 的 `read` 接口拉取和计算特定分区中的数据。

```
override def compute(split: Partition, context: TaskContext): Ite
  val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V
  SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.
    .read()
    .asInstanceOf[Iterator[(K, C)]]
}
```

在后面小节中，我们会进一步分析 `ShuffleWriter.write` 和 `ShuffleReader.read` 的具体实现。

参考资料

1. Shuffle 过程 | Apache Spark 的设计与实现
2. ***Hadoop Definitive Guide***