3.6 部署模式对比

模式	Spark 安装机器数	需启动的进程	所属者	应用场景
Local	1	无	Spark	测试
Standalone	3	Master 及 Worker	Spark	单独部署
Yarn	1	Yarn 及 HDFS	Hadoop	混合部署

3.7 端口号

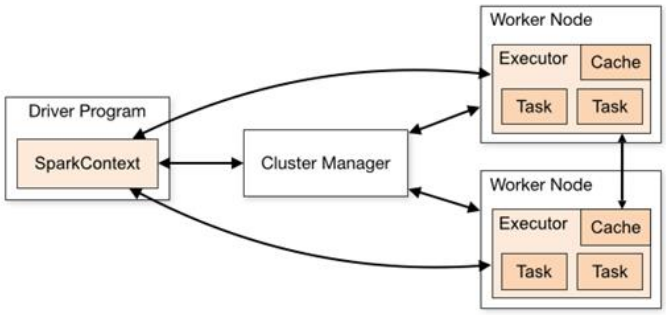
- Spark 查看当前 Spark-shell 运行任务情况端口号：4040（计算）
- Spark Master 内部通信服务端口号：7077
- Standalone 模式下，Spark Master Web 端口号：8080（资源）
- Spark 历史服务器端口号：18080
- Hadoop YARN 任务运行情况查看端口号：8088

第4章　Spark 运行架构

4.1 运行架构

Spark 框架的核心是一个计算引擎，整体来说，它采用了标准 master-slave 的结构。

如下图所示，它展示了一个 Spark 执行时的基本结构。图形中的 Driver 表示 master，负责管理整个集群中的作业任务调度。图形中的 Executor 则是 slave，负责实际执行任务。



4.2 核心组件

由上图可以看出，对于 Spark 框架有两个核心组件：

4.2.1 Driver

Spark 驱动器节点，用于执行 Spark 任务中的 main 方法，负责实际代码的执行工作。Driver 在 Spark 作业执行时主要负责：

- 将用户程序转化为作业（job）
- 在 Executor 之间调度任务(task)
- 跟踪 Executor 的执行情况
- 通过 UI 展示查询运行情况

实际上，我们无法准确地描述 Driver 的定义，因为在整个的编程过程中没有看到任何有关 Driver 的字眼。所以简单理解，所谓的 Driver 就是驱使整个应用运行起来的程序，也称之为 Driver 类。

4.2.2 Executor

Spark Executor 是集群中工作节点（Worker）中的一个 JVM 进程，负责在 Spark 作业中运行具体任务（Task），任务彼此之间相互独立。Spark 应用启动时，Executor 节点被同

时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有 Executor 节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他 Executor 节点上继续运行。

Executor 有两个核心功能：

- 负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程
- 它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在 Executor 进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

4.2.3 Master & Worker

Spark 集群的独立部署环境中，不需要依赖其他的资源调度框架，自身就实现了资源调度的功能，所以环境中还有其他两个核心组件：Master 和 Worker，这里的 Master 是一个进程，主要负责资源的调度和分配，并进行集群的监控等职责，类似于 Yarn 环境中的 RM, 而 Worker 呢，也是进程，一个 Worker 运行在集群中的一台服务器上，由 Master 分配资源对数据进行并行的处理和计算，类似于 Yarn 环境中 NM。

4.2.4 ApplicationMaster

Hadoop 用户向 YARN 集群提交应用程序时,提交程序中应该包含 ApplicationMaster，用于向资源调度器申请执行任务的资源容器 Container，运行用户自己的程序任务 job，监控整个任务的执行，跟踪整个任务的状态，处理任务失败等异常情况。

说的简单点就是，ResourceManager（资源）和 Driver（计算）之间的解耦合靠的就是 ApplicationMaster。

4.3 核心概念

4.3.1 Executor 与 Core

Spark Executor 是集群中运行在工作节点（Worker）中的一个 JVM 进程，是整个集群中的专门用于计算的节点。在提交应用中，可以提供参数指定计算节点的个数，以及对应的资源。这里的资源一般指的是工作节点 Executor 的内存大小和使用的虚拟 CPU 核（Core）数量。

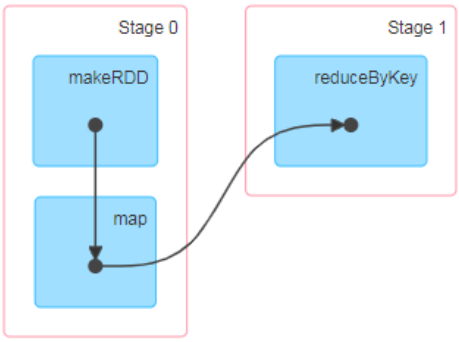
应用程序相关启动参数如下：

名称	说明
--num-executors	配置 Executor 的数量
--executor-memory	配置每个 Executor 的内存大小
--executor-cores	配置每个 Executor 的虚拟 CPU core 数量

4.3.2 并行度（Parallelism）

在分布式计算框架中一般都是多个任务同时执行，由于任务分布在不同的计算节点进行计算，所以能够真正地实现多任务并行执行，记住，这里是并行，而不是并发。这里我们将整个集群并行执行任务的数量称之为并行度。那么一个作业到底并行度是多少呢？这个取决于框架的默认配置。应用程序也可以在运行过程中动态修改。

4.3.3 有向无环图（DAG）



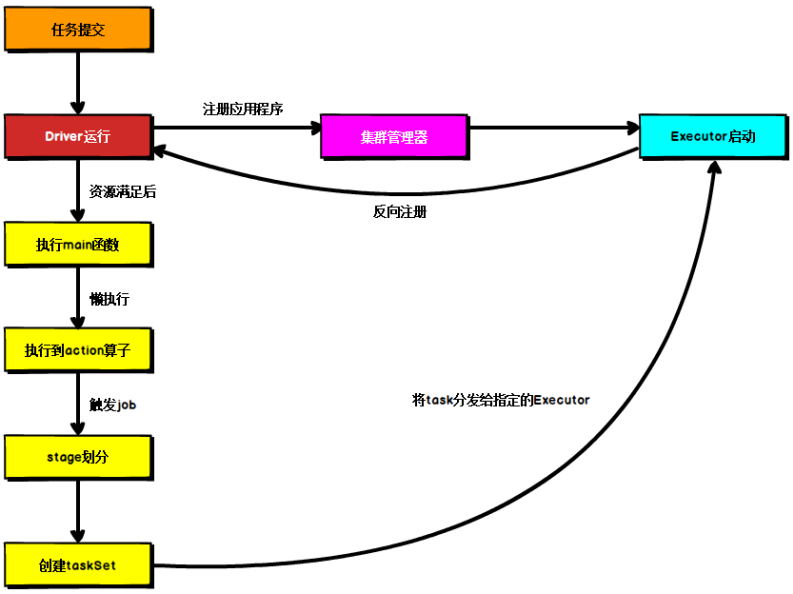
大数据计算引擎框架我们根据使用方式的不同一般会分为四类，其中第一类就是 Hadoop 所承载的 MapReduce,它将计算分为两个阶段，分别为 Map 阶段 和 Reduce 阶段。对于上层应用来说，就不得不想方设法去拆分算法，甚至于不得不在上层应用实现多个 Job 的串联，以完成一个完整的算法，例如迭代计算。 由于这样的弊端，催生了支持 DAG 框架的产生。因此，支持 DAG 的框架被划分为第二代计算引擎。如 Tez 以及更上层的 Oozie。这里我们不去细究各种 DAG 实现之间的区别，不过对于当时的 Tez 和 Oozie 来说，大多还是批处理的任务。接下来就是以 Spark 为代表的第三代的计算引擎。第三代计算引擎的特点主要是 Job 内部的 DAG 支持（不跨越 Job），以及实时计算。

这里所谓的有向无环图，并不是真正意义的图形，而是由 Spark 程序直接映射成的数据流的高级抽象模型。简单理解就是将整个程序计算的执行过程用图形表示出来,这样更直观，更便于理解，可以用于表示程序的拓扑结构。

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。

4.4 提交流程

所谓的提交流程，其实就是我们开发人员根据需求写的应用程序通过 Spark 客户端提交给 Spark 运行环境执行计算的流程。在不同的部署环境中，这个提交过程基本相同，但是又有细微的区别，我们这里不进行详细的比较，但是因为国内工作中，将 Spark 引用部署到 Yarn 环境中会更多一些，所以本课程中的提交流程是基于 Yarn 环境的。



Spark 应用程序提交到 Yarn 环境中执行的时候，一般会有两种部署执行的方式：Client 和 Cluster。两种模式主要区别在于：Driver 程序的运行节点位置。

4.2.1 Yarn Client 模式

Client 模式将用于监控和调度的 Driver 模块在客户端执行，而不是在 Yarn 中，所以一般用于测试。

- Driver 在任务提交的本地机器上运行
- Driver 启动后会和 ResourceManager 通讯申请启动 ApplicationMaster
- ResourceManager 分配 container，在合适的 NodeManager 上启动 ApplicationMaster，负责向 ResourceManager 申请 Executor 内存
- ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container，然后 ApplicationMaster 在资源分配指定的 NodeManager 上启动 Executor 进程

- Executor 进程启动后会向 Driver 反向注册，Executor 全部注册完成后 Driver 开始执行 main 函数
- 之后执行到 Action 算子时，触发一个 Job，并根据宽依赖开始划分 stage，每个 stage 生成对应的 TaskSet，之后将 task 分发到各个 Executor 上执行。

4.2.2 Yarn Cluster 模式

Cluster 模式将用于监控和调度的 Driver 模块启动在 Yarn 集群资源中执行。一般应用于实际生产环境。

- 在 YARN Cluster 模式下，任务提交后会和 ResourceManager 通讯申请启动 ApplicationMaster，
- 随后 ResourceManager 分配 container，在合适的 NodeManager 上启动 ApplicationMaster，此时的 ApplicationMaster 就是 Driver。
- Driver 启动后向 ResourceManager 申请 Executor 内存，ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container，然后在合适的 NodeManager 上启动 Executor 进程
- Executor 进程启动后会向 Driver 反向注册，Executor 全部注册完成后 Driver 开始执行 main 函数，
- 之后执行到 Action 算子时，触发一个 Job，并根据宽依赖开始划分 stage，每个 stage 生成对应的 TaskSet，之后将 task 分发到各个 Executor 上执行。

第5章　Spark 核心编程

Spark 计算框架为了能够进行高并发和高吞吐的数据处理，封装了三大数据结构，用于处理不同的应用场景。三大数据结构分别是：

- RDD：弹性分布式数据集
- 累加器：分布式共享只写变量
- 广播变量：分布式共享只读变量

接下来我们一起看看这三大数据结构是如何在数据处理中使用的。

5.1 RDD

5.1.1 什么是 RDD

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是 Spark 中最基本的数据处理模型。代码中是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。

- 弹性
  - 存储的弹性：内存与磁盘的自动切换；
  - 容错的弹性：数据丢失可以自动恢复；
  - 计算的弹性：计算出错重试机制；
  - 分片的弹性：可根据需要重新分片。
- 分布式：数据存储在大数据集群不同节点上
- 数据集：RDD 封装了计算逻辑，并不保存数据
- 数据抽象：RDD 是一个抽象类，需要子类具体实现
- 不可变：RDD 封装了计算逻辑，是不可以改变的，想要改变，只能产生新的 RDD，在新的 RDD 里面封装计算逻辑
- 可分区、并行计算

5.1.2 核心属性

```
* Internally, each RDD is characterized by five main properties:
*
*  - A list of partitions
*  - A function for computing each split
*  - A list of dependencies on other RDDs
*  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
*  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
*    an HDFS file)
```

➢ 分区列表

RDD 数据结构中存在分区列表，用于执行任务时并行计算，是实现分布式计算的重要属性。

```
/**
 * Implemented by subclasses to return the set of partitions in this RDD. This method will only
 * be called once, so it is safe to implement a time-consuming computation in it.
 *
 * The partitions in this array must satisfy the following property:
 *   `rdd.partitions.zipWithIndex.forall { case (partition, index) => partition.index == index }`
 */
protected def getPartitions: Array[Partition]
```

➢ 分区计算函数

Spark 在计算时，是使用分区函数对每一个分区进行计算

```
/**
 * :: DeveloperApi ::
 * Implemented by subclasses to compute a given partition.
 */
@DeveloperApi
def compute(split: Partition, context: TaskContext): Iterator[T]
```

➢ RDD 之间的依赖关系

RDD 是计算模型的封装，当需求中需要将多个计算模型进行组合时，就需要将多个 RDD 建立依赖关系

```
/**
 * Implemented by subclasses to return how this RDD depends on parent RDDs. This method will only
 * be called once, so it is safe to implement a time-consuming computation in it.
 */
protected def getDependencies: Seq[Dependency[_]] = deps
```

➢ 分区器（可选）

当数据为 KV 类型数据时，可以通过设定分区器自定义数据的分区

```
/** Optionally overridden by subclasses to specify how they are partitioned. */
@transient val partitioner: Option[Partitioner] = None
```

➢ 首选位置（可选）

计算数据时，可以根据计算节点的状态选择不同的节点位置进行计算

```
/**
 * Optionally overridden by subclasses to specify placement preferences.
 */
protected def getPreferredLocations(split: Partition): Seq[String] = Nil
```

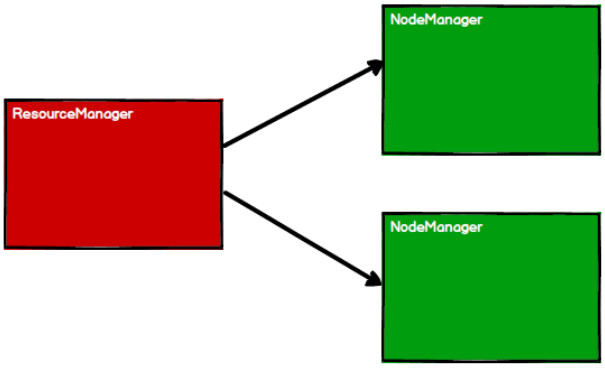
5.1.3 执行原理

从计算的角度来讲，数据处理过程中需要计算资源（内存 & CPU）和计算模型（逻辑）。执行时，需要将计算资源和计算模型进行协调和整合。

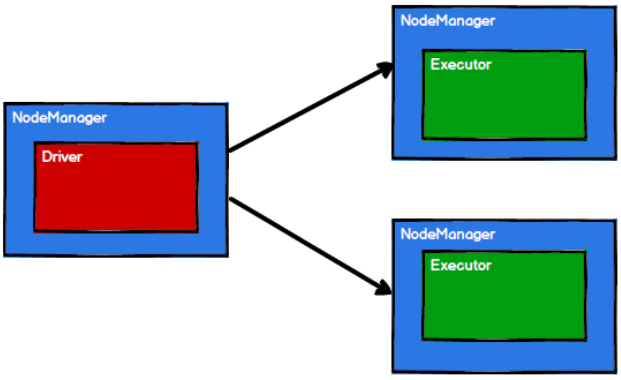
Spark 框架在执行时，先申请资源，然后将应用程序的数据处理逻辑分解成一个一个的计算任务。然后将任务发到已经分配资源的计算节点上，按照指定的计算模型进行数据计算。最后得到计算结果。

RDD 是 Spark 框架中用于数据处理的核心模型，接下来我们看看，在 Yarn 环境中，RDD 的工作原理:

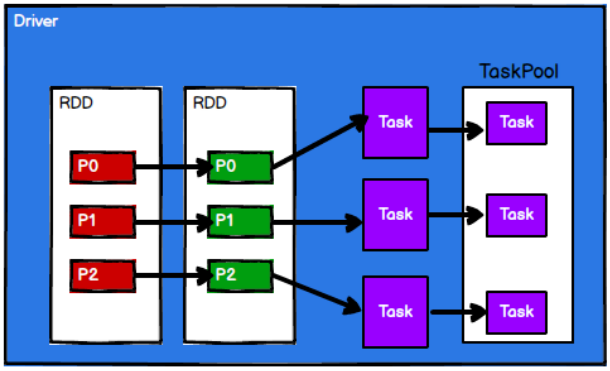
1) 启动 Yarn 集群环境



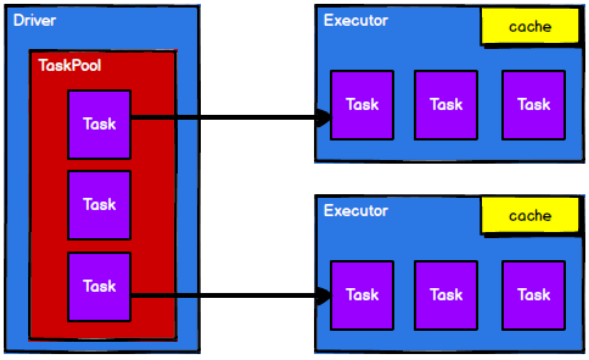
2) Spark 通过申请资源创建调度节点和计算节点



3) Spark 框架根据需求将计算逻辑根据分区划分成不同的任务



4) 调度节点将任务根据计算节点状态发送到对应的计算节点进行计算



从以上流程可以看出 RDD 在整个流程中主要用于将逻辑进行封装，并生成 Task 发送给 Executor 节点执行计算，接下来我们就一起看看 Spark 框架中 RDD 是具体是如何进行数据处理的。

5.1.4 基础编程

5.1.4.1 RDD 创建

在 Spark 中创建 RDD 的创建方式可以分为四种：

1) 从集合（内存）中创建 RDD

从集合中创建 RDD，Spark 主要提供了两个方法：parallelize 和 makeRDD

```
val sparkConf =
    new SparkConf().setMaster("local[*]").setAppName("spark")
val sparkContext = new SparkContext(sparkConf)
val rdd1 = sparkContext.parallelize(
    List(1,2,3,4)
)
val rdd2 = sparkContext.makeRDD(
    List(1,2,3,4)
)
rdd1.collect().foreach(println)
rdd2.collect().foreach(println)
sparkContext.stop()
```

从底层代码实现来讲，makeRDD 方法其实就是 parallelize 方法

```
def makeRDD[T: ClassTag](
    seq: Seq[T],
    numSlices: Int = defaultParallelism): RDD[T] = withScope {
  parallelize(seq, numSlices)
}
```

2) 从外部存储（文件）创建 RDD

由外部存储系统的数据集创建 RDD 包括：本地的文件系统，所有 Hadoop 支持的数据集，比如 HDFS、HBase 等。

```
val sparkConf =
    new SparkConf().setMaster("local[*]").setAppName("spark")
val sparkContext = new SparkContext(sparkConf)
val fileRDD: RDD[String] = sparkContext.textFile("input")
fileRDD.collect().foreach(println)
sparkContext.stop()
```

3) 从其他 RDD 创建

主要是通过一个 RDD 运算完后，再产生新的 RDD。详情请参考后续章节

4) 直接创建 RDD（new）

使用 new 的方式直接构造 RDD，一般由 Spark 框架自身使用。

5.1.4.2 RDD 并行度与分区

默认情况下，Spark 可以将一个作业切分多个任务后，发送给 Executor 节点并行计算，而能够并行计算的任务数量我们称之为并行度。这个数量可以在构建 RDD 时指定。记住，这里的并行执行的任务数量，并不是指的切分任务的数量，不要混淆了。

```
val sparkConf =
    new SparkConf().setMaster("local[*]").setAppName("spark")
val sparkContext = new SparkContext(sparkConf)
val dataRDD: RDD[Int] =
    sparkContext.makeRDD(
        List(1,2,3,4),
        4)
val fileRDD: RDD[String] =
    sparkContext.textFile(
        "input",
        2)
fileRDD.collect().foreach(println)
sparkContext.stop()
```

- 读取内存数据时，数据可以按照并行度的设定进行数据的分区操作，数据分区规则的 Spark 核心源码如下：

```
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)
  }
```

```
}
```

- 读取文件数据时，数据是按照 Hadoop 文件读取的规则进行切片分区，而切片规则和数据读取的规则有些差异，具体 Spark 核心源码如下

```
public InputSplit[] getSplits(JobConf job, int numSplits)
    throws IOException {

    long totalSize = 0;                           // compute total size
    for (FileStatus file: files) {                // check we have valid files
      if (file.isDirectory()) {
        throw new IOException("Not a file: "+ file.getPath());
      }
      totalSize += file.getLen();
    }

    long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);
    long minSize = Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.
      FileInputFormat.SPLIT_MINSIZE, 1), minSplitSize);

    ...

    for (FileStatus file: files) {

        ...

    if (isSplitable(fs, path)) {
         long blockSize = file.getBlockSize();
         long splitSize = computeSplitSize(goalSize, minSize, blockSize);

         ...

  }
  protected long computeSplitSize(long goalSize, long minSize,
                                  long blockSize) {
    return Math.max(minSize, Math.min(goalSize, blockSize));
  }
```

5.1.4.3 RDD 转换算子

RDD 根据数据处理方式的不同将算子整体上分为 Value 类型、双 Value 类型和 Key-Value 类型

- **Value 类型**

1) map

➢ 函数签名

```
def map[U: ClassTag](f: T => U): RDD[U]
```

➢ 函数说明

将处理的数据逐条进行映射转换，这里的转换可以是类型的转换，也可以是值的转换。

```
val dataRDD: RDD[Int] = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD1: RDD[Int] = dataRDD.map(
    num => {
```

```
        num * 2
    }
)
val dataRDD2: RDD[String] = dataRDD1.map(
    num => {
        "" + num
    }
)
```

❖ 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) mapPartitions

➢ 函数签名

```
def mapPartitions[U: ClassTag](
    f: Iterator[T] => Iterator[U],
    preservesPartitioning: Boolean = false): RDD[U]
```

➢ 函数说明

将待处理的数据以分区为单位发送到计算节点进行处理，这里的处理是指可以进行任意的处理，哪怕是过滤数据。

```
val dataRDD1: RDD[Int] = dataRDD.mapPartitions(
    datas => {
        datas.filter(_==2)
    }
)
```

❖ 小功能：获取每个数据分区的最大值

思考一个问题：map 和 mapPartitions 的区别？

➢ 数据处理角度

Map 算子是分区内一个数据一个数据的执行，类似于串行操作。而 mapPartitions 算子是以分区为单位进行批处理操作。

➢ 功能的角度

Map 算子主要目的将数据源中的数据进行转换和改变。但是不会减少或增多数据。MapPartitions 算子需要传递一个迭代器，返回一个迭代器，没有要求的元素的个数保持不变，所以可以增加或减少数据

➢ 性能的角度

Map 算子因为类似于串行操作，所以性能比较低，而是 mapPartitions 算子类似于批处理，所以性能较高。但是 mapPartitions 算子会长时间占用内存，那么这样会导致内存可能不够用，出现内存溢出的错误。所以在内存有限的情况下，不推荐使用。使用 map 操作。

完成比完美更重要

3) mapPartitionsWithIndex

➢ 函数签名

```
def mapPartitionsWithIndex[U: ClassTag](
   f: (Int, Iterator[T]) => Iterator[U],
   preservesPartitioning: Boolean = false): RDD[U]
```

➢ 函数说明

将待处理的数据以分区为单位发送到计算节点进行处理，这里的处理是指可以进行任意的处理，哪怕是过滤数据，在处理时同时可以获取当前分区索引。

```
val dataRDD1 = dataRDD.mapPartitionsWithIndex(
    (index, datas) => {
        datas.map(index, _)
    }
)
```

❖ 小功能：获取第二个数据分区的数据

4) flatMap

➢ 函数签名

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]
```

➢ 函数说明

将处理的数据进行扁平化后再进行映射处理，所以算子也称之为扁平映射

```
val dataRDD = sparkContext.makeRDD(List(
    List(1,2),List(3,4)
),1)
val dataRDD1 = dataRDD.flatMap(
    list => list
)
```

❖ 小功能：将 List(List(1,2),3,List(4,5))进行扁平化操作

5) glom

➢ 函数签名

```
def glom(): RDD[Array[T]]
```

➢ 函数说明

将同一个分区的数据直接转换为相同类型的内存数组进行处理，分区不变

```
val dataRDD = sparkContext.makeRDD(List(
    1,2,3,4
),1)
val dataRDD1:RDD[Array[Int]] = dataRDD.glom()
```

❖ 小功能：计算所有分区最大值求和（分区内取最大值，分区间最大值求和）

6) groupBy

➢ 函数签名

def groupBy[K](f: T => K)(implicit kt: ClassTag[K]): RDD[(K, Iterable[T])]

➢ 函数说明

将数据根据指定的规则进行分组，分区默认不变，但是数据会被打乱重新组合，我们将这样的操作称之为 shuffle。极限情况下，数据可能被分在同一个分区中

一个组的数据在一个分区中，但是并不是说一个分区中只有一个组

```
val dataRDD = sparkContext.makeRDD(List(1,2,3,4),1)
val dataRDD1 = dataRDD.groupBy(
    _%2
)
```

❖ 小功能：将 List("Hello", "hive", "hbase", "Hadoop")根据单词首写字母进行分组。

❖ 小功能：从服务器日志数据 apache.log 中获取每个时间段访问量。

❖ 小功能：WordCount。

7) filter

➢ 函数签名

def filter(f: T => Boolean): RDD[T]

➢ 函数说明

将数据根据指定的规则进行筛选过滤，符合规则的数据保留，不符合规则的数据丢弃。

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。

```
val dataRDD = sparkContext.makeRDD(List(
    1,2,3,4
),1)
val dataRDD1 = dataRDD.filter(_%2 == 0)
```

❖ 小功能：从服务器日志数据 apache.log 中获取 2015 年 5 月 17 日的请求路径

8) sample

➢ 函数签名

```
def sample(
  withReplacement: Boolean,
  fraction: Double,
  seed: Long = Utils.random.nextLong): RDD[T]
```

➢ 函数说明

根据指定的规则从数据集中抽取数据

```
val dataRDD = sparkContext.makeRDD(List(
   1,2,3,4
),1)
// 抽取数据不放回（伯努利算法）
// 伯努利算法：又叫 0、1 分布。例如扔硬币，要么正面，要么反面。
// 具体实现：根据种子和随机算法算出一个数和第二个参数设置几率比较，小于第二个参数要，大于不要
// 第一个参数：抽取的数据是否放回，false：不放回
// 第二个参数：抽取的几率，范围在[0,1]之间,0：全不取；1：全取；
// 第三个参数：随机数种子
val dataRDD1 = dataRDD.sample(false, 0.5)
// 抽取数据放回（泊松算法）
// 第一个参数：抽取的数据是否放回，true：放回；false：不放回
// 第二个参数：重复数据的几率，范围大于等于 0.表示每一个元素被期望抽取到的次数
// 第三个参数：随机数种子
val dataRDD2 = dataRDD.sample(true, 2)
```


思考一个问题：有啥用，抽奖吗？

9) distinct

➢ 函数签名

```
def distinct()(implicit ord: Ordering[T] = null): RDD[T]

def distinct(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

➢ 函数说明

将数据集中重复的数据去重

```
val dataRDD = sparkContext.makeRDD(List(
   1,2,3,4,1,2
),1)
val dataRDD1 = dataRDD.distinct()

val dataRDD2 = dataRDD.distinct(2)
```


思考一个问题：如果不用该算子，你有什么办法实现数据去重？

10) coalesce

➢ 函数签名

```
def coalesce(numPartitions: Int, shuffle: Boolean = false,
             partitionCoalescer: Option[PartitionCoalescer] = Option.empty)
            (implicit ord: Ordering[T] = null)
  : RDD[T]
```

➢ 函数说明

根据数据量缩减分区，用于大数据集过滤后，提高小数据集的执行效率

当 spark 程序中，存在过多的小任务的时候，可以通过 coalesce 方法，收缩合并分区，减少分区的个数，减小任务调度成本

```
val dataRDD = sparkContext.makeRDD(List(
   1,2,3,4,1,2
),6)

val dataRDD1 = dataRDD.coalesce(2)
```


思考一个问题：我想要扩大分区，怎么办？

11) repartition

➢ 函数签名

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

➢ 函数说明

该操作内部其实执行的是 coalesce 操作，参数 shuffle 的默认值为 true。无论是将分区数多的 RDD 转换为分区数少的 RDD，还是将分区数少的 RDD 转换为分区数多的 RDD，repartition 操作都可以完成，因为无论如何都会经 shuffle 过程。

```
val dataRDD = sparkContext.makeRDD(List(
   1,2,3,4,1,2
),2)

val dataRDD1 = dataRDD.repartition(4)
```


思考一个问题：coalesce 和 repartition 区别？

12) sortBy

➢ 函数签名

```
def sortBy[K](
  f: (T) => K,
```

ascending: Boolean = true,

numPartitions: Int = this.partitions.length)

(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]

➢ 函数说明

该操作用于排序数据。在排序之前，可以将数据通过 f 函数进行处理，之后按照 f 函数处理的结果进行排序，默认为升序排列。排序后新产生的 RDD 的分区数与原 RDD 的分区数一致。中间存在 shuffle 的过程

```
val dataRDD = sparkContext.makeRDD(List(
   1,2,3,4,1,2
),2)

val dataRDD1 = dataRDD.sortBy(num=>num, false, 4)
```

- **双 Value 类型**

13) intersection

➢ 函数签名

def intersection(other: RDD[T]): RDD[T]

➢ 函数说明

对源 RDD 和参数 RDD 求交集后返回一个新的 RDD

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))
val dataRDD = dataRDD1.intersection(dataRDD2)
```


思考一个问题：如果两个 RDD 数据类型不一致怎么办？

14) union

➢ 函数签名

def union(other: RDD[T]): RDD[T]

➢ 函数说明

对源 RDD 和参数 RDD 求并集后返回一个新的 RDD

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))
val dataRDD = dataRDD1.union(dataRDD2)
```


思考一个问题：如果两个 RDD 数据类型不一致怎么办？

15) subtract

➢ 函数签名

def subtract(other: RDD[T]): RDD[T]

➢ 函数说明

以一个 RDD 元素为主，去除两个 RDD 中重复元素，将其他元素保留下来。求差集

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))
val dataRDD = dataRDD1.subtract(dataRDD2)
```

 思考一个问题：如果两个 RDD 数据类型不一致怎么办？

16) zip

➢ 函数签名

def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]

➢ 函数说明

将两个 RDD 中的元素，以键值对的形式进行合并。其中，键值对中的 Key 为第 1 个 RDD 中的元素，Value 为第 2 个 RDD 中的相同位置的元素。

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))
val dataRDD = dataRDD1.zip(dataRDD2)
```

 思考一个问题：如果两个 RDD 数据类型不一致怎么办？

 思考一个问题：如果两个 RDD 数据分区不一致怎么办？

 思考一个问题：如果两个 RDD 分区数据数量不一致怎么办？

● Key - Value 类型

17) partitionBy

➢ 函数签名

def partitionBy(partitioner: Partitioner): RDD[(K, V)]

➢ 函数说明

将数据按照指定 Partitioner 重新进行分区。Spark 默认的分区器是 HashPartitioner

```
val rdd: RDD[(Int, String)] =
    sc.makeRDD(Array((1,"aaa"),(2,"bbb"),(3,"ccc")),3)
import org.apache.spark.HashPartitioner
```

```
val rdd2: RDD[(Int, String)] =
    rdd.partitionBy(new HashPartitioner(2))
```

思考一个问题：如果重分区的分区器和当前 RDD 的分区器一样怎么办？

思考一个问题：Spark 还有其他分区器吗？

思考一个问题：如果想按照自己的方法进行数据分区怎么办？

思考一个问题：哪那么多问题？

18) reduceByKey

➢ 函数签名

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]

def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]
```

➢ 函数说明

可以将数据按照相同的 Key 对 Value 进行聚合

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))
val dataRDD2 = dataRDD1.reduceByKey(_+_)
val dataRDD3 = dataRDD1.reduceByKey(_+_, 2)
```

❖ 小功能：WordCount

19) groupByKey

➢ 函数签名

```
def groupByKey(): RDD[(K, Iterable[V])]

def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]

def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]
```

➢ 函数说明

将数据源的数据根据 key 对 value 进行分组

```
val dataRDD1 =
    sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))
val dataRDD2 = dataRDD1.groupByKey()
val dataRDD3 = dataRDD1.groupByKey(2)
val dataRDD4 = dataRDD1.groupByKey(new HashPartitioner(2))
```

思考一个问题：reduceByKey 和 groupByKey 的区别？

从 shuffle 的角度：reduceByKey 和 groupByKey 都存在 shuffle 的操作，但是 reduceByKey 可以在 shuffle 前对分区内相同 key 的数据进行预聚合（combine）功能，这样会减少落盘的数据量，而 groupByKey 只是进行分组，不存在数据量减少的问题，reduceByKey 性能比较高。

从功能的角度：reduceByKey 其实包含分组和聚合的功能。GroupByKey 只能分组，不能聚合，所以在分组聚合的场合下，推荐使用 reduceByKey，如果仅仅是分组而不需要聚合。那么还是只能使用 groupByKey

❖ 小功能：WordCount

20) aggregateByKey

➢ 函数签名

```
def aggregateByKey[U: ClassTag](zeroValue: U)(seqOp: (U, V) => U,
  combOp: (U, U) => U): RDD[(K, U)]
```

➢ 函数说明

将数据根据不同的规则进行分区内计算和分区间计算

```
val dataRDD1 =
    sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))
val dataRDD2 =
    dataRDD1.aggregateByKey(0)(_+_,_+_)
```

❖ 取出每个分区内相同 key 的最大值然后分区间相加

```
// TODO : 取出每个分区内相同 key 的最大值然后分区间相加
// aggregateByKey 算子是函数柯里化，存在两个参数列表
// 1. 第一个参数列表中的参数表示初始值
// 2. 第二个参数列表中含有两个参数
//    2.1 第一个参数表示分区内的计算规则
//    2.2 第二个参数表示分区间的计算规则
val rdd =
    sc.makeRDD(List(
        ("a",1),("a",2),("c",3),
        ("b",4),("c",5),("c",6)
    ),2)
// 0:("a",1),("a",2),("c",3) => (a,10)(c,10)
//                                  => (a,10)(b,10)(c,20)
// 1:("b",4),("c",5),("c",6) => (b,10)(c,10)

val resultRDD =
    rdd.aggregateByKey(10)(
        (x, y) => math.max(x,y),
        (x, y) => x + y
    )

resultRDD.collect().foreach(println)
```

思考一个问题：分区内计算规则和分区间计算规则相同怎么办？（WordCount）

21) foldByKey

➢ 函数签名

```
def foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)]
```

➢ 函数说明

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

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))
val dataRDD2 = dataRDD1.foldByKey(0)(_+_)
```

22) combineByKey

➢ 函数签名

```
def combineByKey[C](
  createCombiner: V => C,
  mergeValue: (C, V) => C,
  mergeCombiners: (C, C) => C): RDD[(K, C)]
```

➢ 函数说明

最通用的对 key-value 型 rdd 进行聚集操作的聚集函数（aggregation function）。类似于 aggregate()，combineByKey()允许用户返回值的类型与输入不一致。

小练习：将数据 List(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98))求每个 key 的平均值

```
val list: List[(String, Int)] = List(("a", 88), ("b", 95), ("a", 91), ("b", 93),
("a", 95), ("b", 98))
val input: RDD[(String, Int)] = sc.makeRDD(list, 2)

val combineRdd: RDD[(String, (Int, Int))] = input.combineByKey(
    (_, 1),
    (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
    (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
)
```

思考一个问题：reduceByKey、foldByKey、aggregateByKey、combineByKey 的区别？

reduceByKey: 相同 key 的第一个数据不进行任何计算，分区内和分区间计算规则相同

FoldByKey: 相同 key 的第一个数据和初始值进行分区内计算，分区内和分区间计算规则相同

AggregateByKey：相同 key 的第一个数据和初始值进行分区内计算，分区内和分区间计算规则可以不相同

CombineByKey:当计算时，发现数据结构不满足要求时，可以让第一个数据转换结构。分区内和分区间计算规则不相同。

23) sortByKey

➢ 函数签名

```
def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length)
  : RDD[(K, V)]
```

➢ 函数说明

在一个(K,V)的 RDD 上调用，K 必须实现 Ordered 接口(特质)，返回一个按照 key 进行排序的

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))
val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(true)
val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(false)
```

❖ 小功能：设置 key 为自定义类 User

24) join

➢ 函数签名

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

➢ 函数说明

在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素连接在一起的(K,(V,W))的 RDD

```
val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "a"), (2, "b"), (3, "c")))
val rdd1: RDD[(Int, Int)] = sc.makeRDD(Array((1, 4), (2, 5), (3, 6)))
rdd.join(rdd1).collect().foreach(println)
```


思考一个问题：如果 key 存在不相等呢？

25) leftOuterJoin

➢ 函数签名

```
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]
```

➢ 函数说明

类似于 SQL 语句的左外连接