

Spark

学习笔记

spark 三种部署模式

- 1、使用自己集成的Standalone作为资源分配器
 - 2、使用mesos作为资源分配器
 - 3、使用Yarn作为统一的资源分配器
-
- 从对比上看， mesos似乎是Spark更好的选择， 也是被官方推荐的
 - 但如果你同时运行hadoop和Spark,从兼容性上考虑， Yarn是更好的选择。
 - 如果你不仅运行了hadoop, spark。还在资源管理上运行了docker， Mesos更加通用。
 - Standalone对于小规模计算集群更适合

为迭代而生—Spark

- 1、RDD：弹性分布式数据集(resilient distributed dataset)
- 2、Spark 支持很多种输入输出源（HDFS、本地、结构化数据源、数据库）
- 3、可以将大型工作数据集（如中间输出结果）保留在内存中（cache操作），避免多次读取数据。因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的MapReduce的算法。创建 Spark 就是为了支持分布式数据集上的迭代作业。Spark官网认为在逻辑回归上Spark比Hadoop快100倍。
- 4、Apache Spark使用最先进的DAG调度程序，查询优化器和物理执行引擎，实现批处理和流数据的高性能。

Spark为什么比HadoopMR快

首先，Spark Task的启动时间快。Spark采用fork线程的方式，而HadoopMR采用创建新的进程的方式。

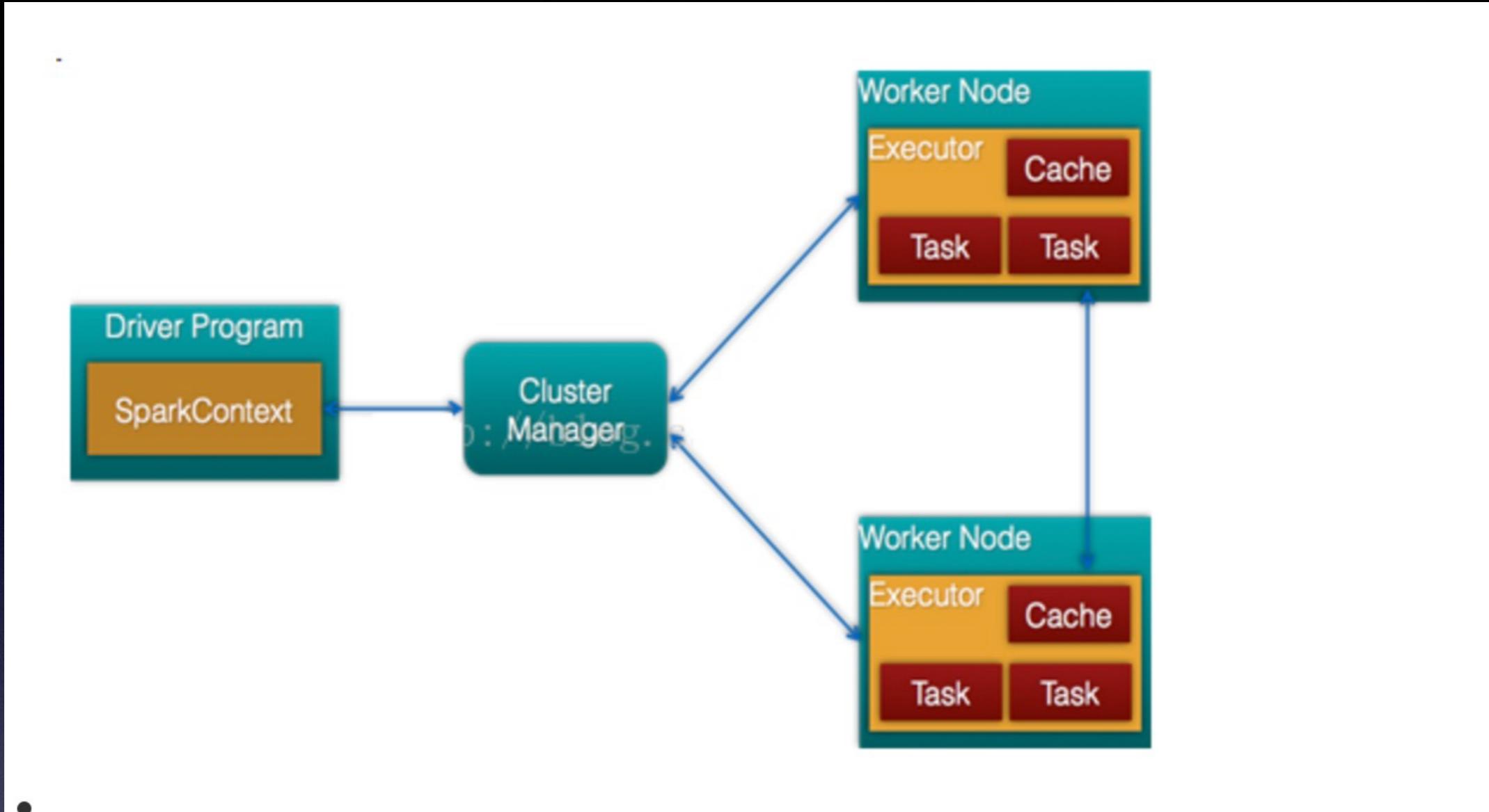
其次，Hadoop中多个MR作业之间的数据交互都要依赖于磁盘交互。MapReduce的一个主要限制是它在运行每个作业后将完整数据集保留到HDFS，这会造成很高的IO开销。

Spark只有在shuffle的时候将数据写入磁盘，当需要将操作的输出馈送到另一个操作时，Spark直接传递数据而不写入持久存储。

Spark的主要创新是引入内存缓存抽象。这使Spark成为多个操作访问相同输入数据的工作负载的理想选择。用户可以指示Spark将输入数据集缓存在内存中，因此不需要为每个操作从磁盘读取它们。

MR缺陷

- 1) 仅支持Map和Reduce两种操作
- 2) 处理效率低效。
 - a) Map中间结果写磁盘, Reduce写HDFS, 多个MR之间通过HDFS交换数据; 任务调度和启动开销大;
 - b) 无法充分利用内存
 - c) Map端和Reduce端均需要排序
- 3) 不适合迭代计算(如机器学习、图计算等), 交互式处理(数据挖掘) 和流式处理(点击日志分析)



Spark的运行模式

每个Spark应用都由一个驱动器程序(driver program)来发起集群上的各种并行操作。驱动器程序通过一个 SparkContext 对象来访问 Spark。这个对象代表对计算集群的一个连接。

RDD

- 一旦有了 SparkContext，你就可以用它来创建 RDD。弹性分布式数据集RDD是Spark 对数据的核心抽象。RDD 其实就是分布式的元素集合。在 Spark 中，对数据的所有操作不外乎创建 RDD、转化已有 RDD 以及调用 RDD 操作进行求值。而在这一切背后，Spark 会自动将 RDD 中的数据分发到集群上，并将操作并行化执行。
- Spark 中的 RDD 就是一个不可变的分布式对象集合。每个 RDD 都被分为多个分区，这些 分区运行在集群中的不同节点上。RDD 可以包含 Python、Java、Scala 中任意类型的对象，甚至可以包含用户自定义的对象。RDD只读，不可以改，只能转化为新的RDD，所以可以通过RDD父子依赖关系重新计算丢失的分区来实现高容错性。
- Resilient because RDDs are immutable(can't be modified once created) and fault tolerant, Distributed because it is distributed across cluster and Dataset because it holds data.
- 两个RDD的创建例子：
 - 从文件系统中加载：如`val lines = sc.textFile("a.txt")`
 - 进行并行集合：`val array = Array(1,2,3,4,5)`
 - `val rdd=sc.parallelize(array)`

RDD 操作：

- 转化操作：RDD 的转化操作就是返回新 RDD 的操作，如之前的读取文件。它有一个特点就是 RDD 的转化操作都是惰性求值的。这意味着在被调用行动操作之前 Spark 不会开始任何计算。因此，当我们调用 sc.textFile() 时，数据并没有读取进来，而是在必要时才会读取。因此我们不应该把 RDD 看作存放着特定数据的数据集，而最好把每个 RDD 当作我们通过转化操作构建出来的、记录如何计算数据的指令列表。
- 常见指令：filter(), map()

表3-2：对一个数据为{1, 2, 3, 3}的RDD进行基本的RDD转化操作

| 函数名 | 目的 | 示例 | 结果 |
|---|---|---------------------------|-----------------------|
| map() | 将函数应用于 RDD 中的每个元素，将返回值构成新的 RDD | rdd.map(x => x + 1) | {2, 3, 4, 4} |
| flatMap() | 将函数应用于 RDD 中的每个元素，将返回的迭代器的所有内容构成新的 RDD。通常用来切分单词 | rdd.flatMap(x => x.to(3)) | {1, 2, 3, 2, 3, 3, 3} |
| filter() | 返回一个由通过传给 filter() 的函数的元素组成的 RDD | rdd.filter(x => x != 1) | {2, 3, 3} |
| distinct() | 去重 | rdd.distinct() | {1, 2, 3} |
| sample(withReplacement, fraction, [seed]) | 对 RDD 采样，以及是否替换 | rdd.sample(false, 0.5) | 非确定的 |

表3-3：对数据分别为{1, 2, 3}和{3, 4, 5}的RDD进行针对两个RDD的转化操作

| 函数名 | 目的 | 示例 | 结果 |
|----------------|-------------------------|-------------------------|------------------------------|
| union() | 生成一个包含两个 RDD 中所有元素的 RDD | rdd.union(other) | {1, 2, 3, 3, 4, 5} |
| intersection() | 求两个 RDD 共同的元素的 RDD | rdd.intersection(other) | {3} |
| subtract() | 移除一个 RDD 中的内容（例如移除训练数据） | rdd.subtract(other) | {1, 2} |
| cartesian() | 与另一个 RDD 的笛卡儿积 | rdd.cartesian(other) | {(1, 3), (1, 4), ... (3, 5)} |

RDD里存的是什么

- 最主要的：(1) 一组分区 (partition, 即数据集的原子组成部分)；(2) 对父RDD的一组依赖，这些依赖描述了RDD的Lineage；(3) 一个函数，即在父RDD上执行何种计算；(4) 元数据，描述分区模式和数据存放的位置。
- 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 操作：

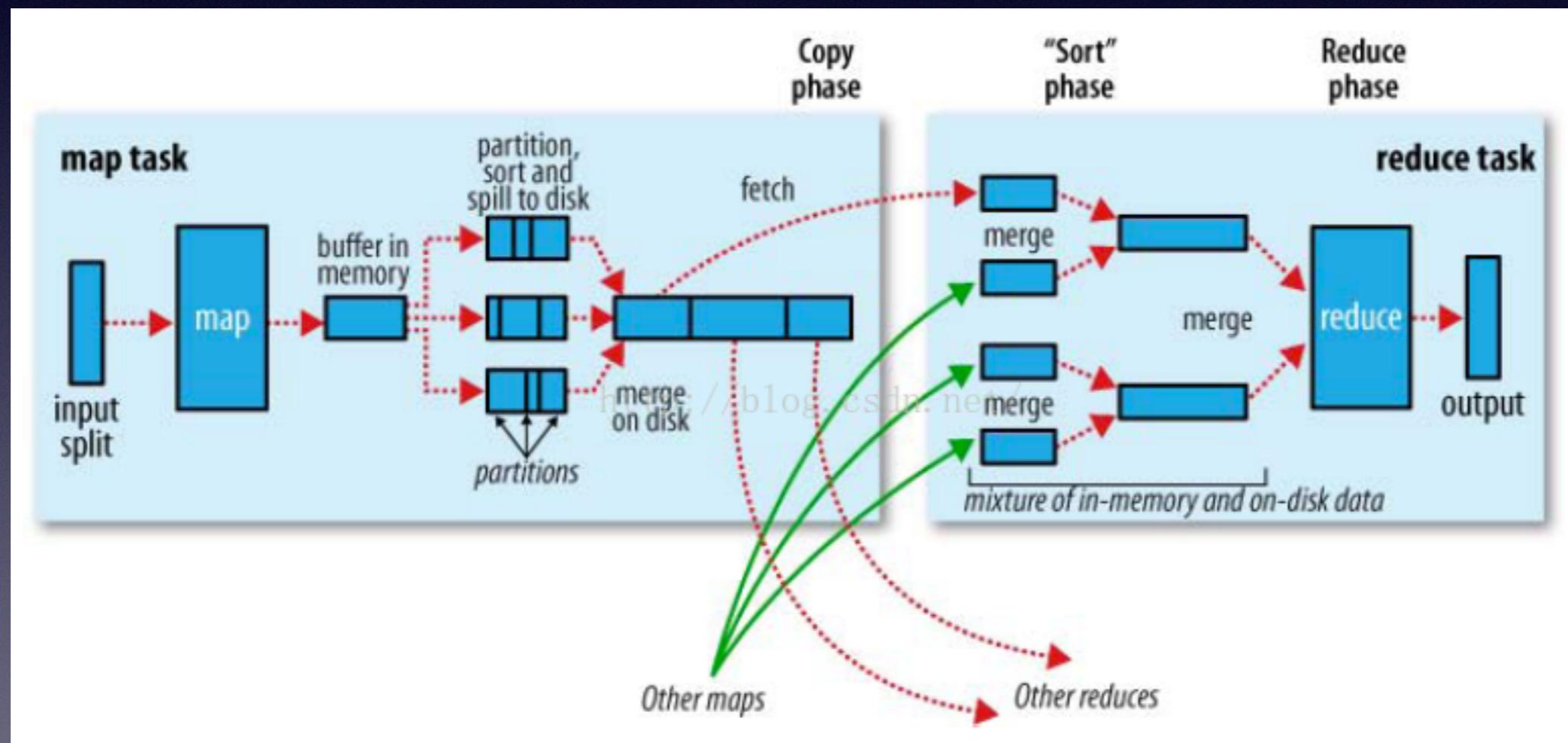
- **行动操作**: 会把最终求得的结果返回到驱动器程序，或者写入外部存储系统中。由于行动操作需要生成实际的输出，它们会强制执行那些求值必须用到的 RDD 的转化操作。
- 常见指令:count(),first(),数组返回函数collect(),返回前n元素take(n), reduce, foreach()
- **持久化**: 默认情况下，Spark 的 RDD 会在你每次对它们进行行动操作时重新计算。如果想在多个行动操作中重用同一个 RDD，可以使用 RDD.persist() 让 Spark 把这个 RDD 缓存下来。我们可以让 Spark 把数据持久化到许多不同的地方，可用的选项会在表 3-6 中列出。在第一次对持久化的 RDD 计算之后，Spark 会把 RDD 的内容保存到内存中(以分区方式存储到集群中的各机器上)，这样在之后的行动操作中，就可以重用这些数据了。
- 常见指令: persist() , cache() 与使用默认存储级别调用 persist() 是一样的。

| 级 别 | 使用的空间 | CPU 时间 | 是否在内存中 | 是否在磁盘上 | 备 注 |
|---------------------|-------|--------|--------|--------|-----------------------------------|
| MEMORY_ONLY | 高 | 低 | 是 | 否 | |
| MEMORY_ONLY_SER | 低 | 高 | 是 | 否 | |
| MEMORY_AND_DISK | 高 | 中等 | 部分 | 部分 | 如果数据在内存中放不下，则溢写到磁盘上 |
| MEMORY_AND_DISK_SER | 低 | 高 | 部分 | 部分 | 如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据 |
| DISK_ONLY | 低 | 高 | 否 | 是 | |

Suffle过程

知识补充： shuffle过程

整个shuffle过程可以看做是从map输出到reduce输入的这个中间过程，在这个中间过程中，经过了一系列的步骤



MapReduce的shuffle过程

Suffle过程

Spark中RDD的suffle过程

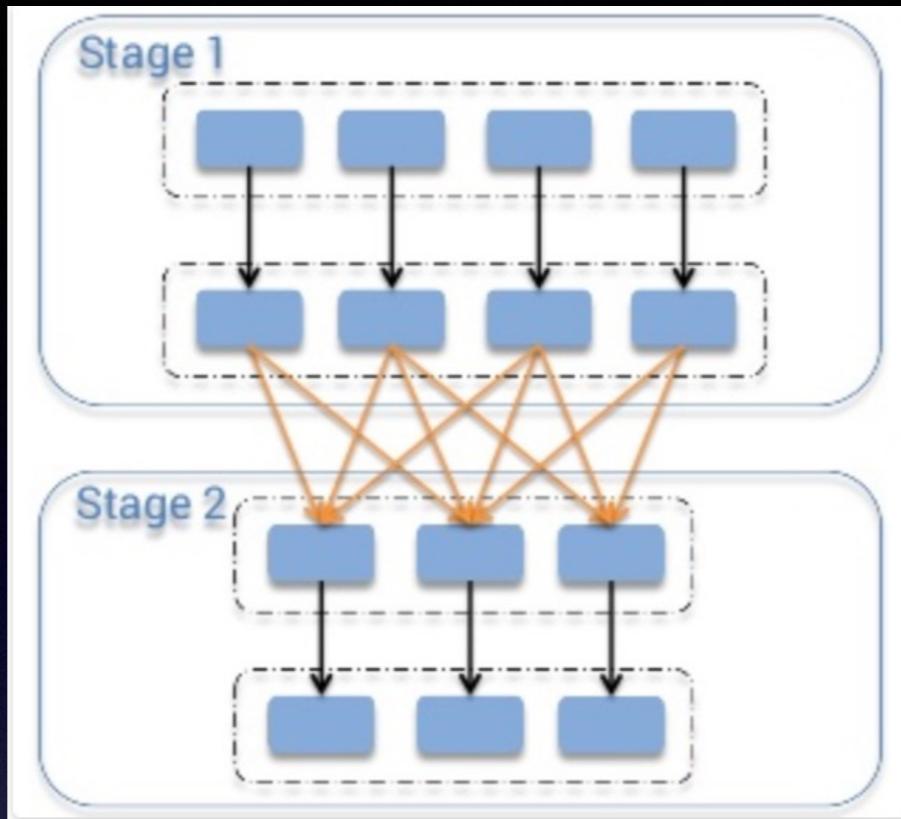
RDD 的 Transformation 函数中,又分为窄依赖(narrow dependency)和宽依赖(wide dependency)的操作. 窄依赖跟宽依赖的区别是是否发生 shuffle(洗牌) 操作. 宽依赖会发生 shuffle 操作. 窄依赖是子 RDD 的各个分片(partition)不依赖于其他分片,

```
val rdd1 = rdd.Map(x => (x.charAt(0), x))

val rdd2 = rdd1.groupBy(x => x._1).
    Map(x => (x._1, x._2.toList.length))
```

第一个 Map 操作将 RDD 里的各个元素进行映射, RDD 的各个数据元素之间不存在依赖,可以在集群的各个内存中独立计算,也就是并行化,第二个 groupby 之后的 Map 操作,为了计算相同 key 下的元素个数,需要把相同 key 的元素聚集到同一个 partition 下,所以造成了数据在内存中的重新分布,即 shuffle 操作.shuffle 操作是 spark 中最耗时的操作,应尽量避免不必要的 shuffle.

Shuffle过程



Spark的shuffle有三种方式：sort（默认），hash，tungsten-sort方式。Spark的shuffle过程也会将文件写入到本地文件中，其中hash最快，sort优化更好。所以比较MR，spark的shuffle过程更加灵活。而且MR的Merge on disk会将文件整合成大的文件，而Spark的MinHeap Merge方式是直接从小文件中获取数据的。

最常见的RDD：键值对RDD

表4-1：Pair RDD的转化操作（以键值对集合 $\{(1, 2), (3, 4), (3, 6)\}$ 为例）

| 函数名 | 目的 | 示例 | 结果 |
|--|--|----------------------------------|--|
| reduceByKey(func) | 合并具有相同键的值 | rdd.reduceByKey((x, y) => x + y) | $\{(1, 2), (3, 10)\}$ |
| groupByKey() | 对具有相同键的值进行分组 | rdd.groupByKey() | $\{(1, [2]), (3, [4, 6])\}$ |
| combineByKey Key(createCombiner, mergeValue, mergeCombiners, partitioner) | 使用不同的返回类型合并具有相同键的值 | 见例 4-12 到例 4-14。 | |
| mapValues(func) | 对 pair RDD 中的每个值应用一个函数而不改变键 | rdd.mapValues(x => x+1) | $\{(1, 3), (3, 5), (3, 7)\}$ |
| flatMapValues(func) | 对 pair RDD 中的每个值应用一个返回迭代器的函数，然后对返回的每个元素都生成一个对应原键的键值对记录。通常用于符号化 | rdd.flatMapValues(x => (x to 5)) | $\{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)\}$ |
| keys() | 返回一个仅包含键的 RDD | rdd.keys() | $\{1, 3, 3\}$ |
| values() | 返回一个仅包含值的 RDD | rdd.values() | $\{2, 4, 6\}$ |
| sortByKey() | 返回一个根据键排序的 RDD | rdd.sortByKey() | $\{(1, 2), (3, 4), (3, 6)\}$ |

表4-2：针对两个pair RDD的转化操作 (`rdd = {(1, 2), (3, 4), (3, 6)}``other = {(3, 9)}`)

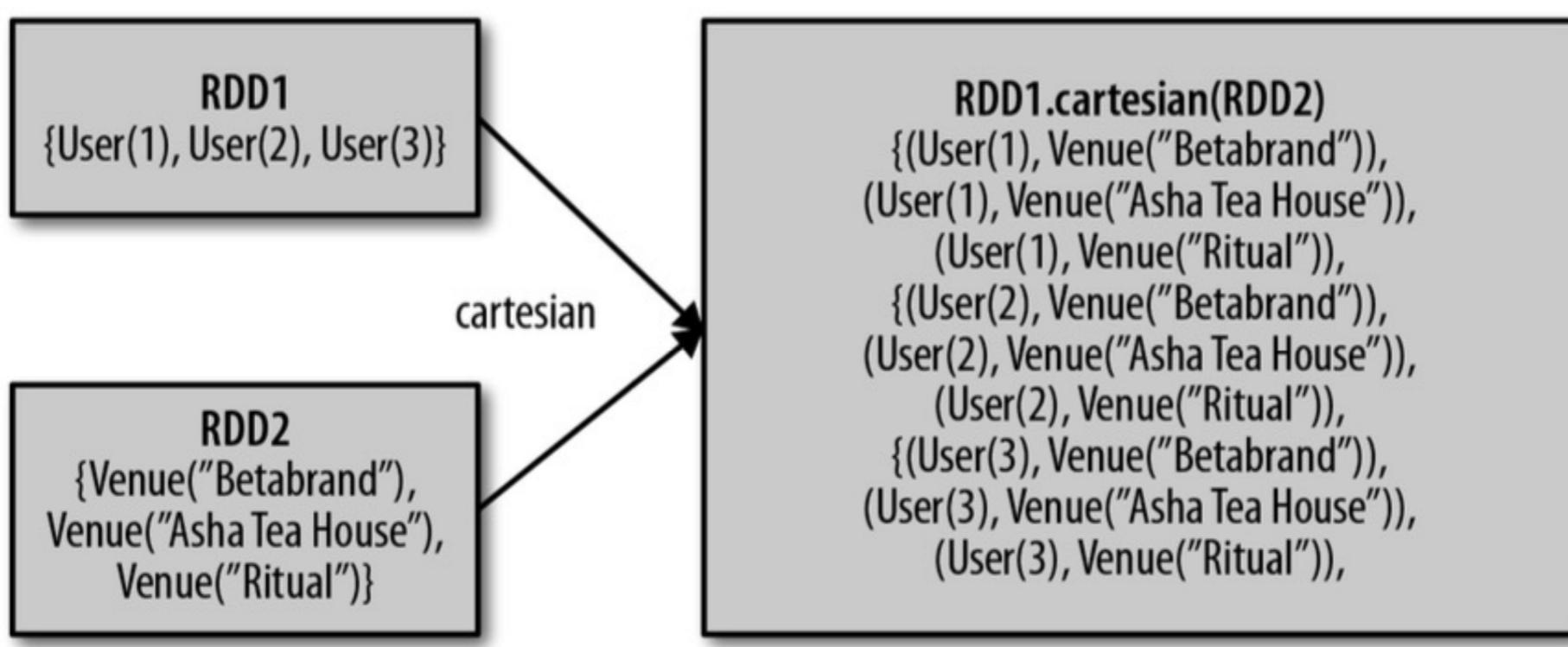
| 函数名 | 目的 | 示例 | 结果 |
|----------------------------|-------------------------------|---------------------------------------|---|
| <code>subtractByKey</code> | 删掉 RDD 中键与 other RDD 中的键相同的元素 | <code>rdd.subtractByKey(other)</code> | <code>{(1, 2)}</code> |
| <code>join</code> | 对两个 RDD 进行内连接 | <code>rdd.join(other)</code> | <code>{(3, (4, 9)), (3, (6, 9))}</code> |

| | | | |
|-----------------------------|---------------------------------------|--|---|
| <code>rightOuterJoin</code> | 对两个 RDD 进行连接操作，确保第一个 RDD 的键必须存在（右外连接） | <code>rdd.rightOuterJoin(other)</code> | <code>{(3,(Some(4),9)), (3,(Some(6),9))}</code> |
| <code>leftOuterJoin</code> | 对两个 RDD 进行连接操作，确保第二个 RDD 的键必须存在（左外连接） | <code>rdd.leftOuterJoin(other)</code> | <code>{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}</code> |
| <code>cogroup</code> | 将两个 RDD 中拥有相同键的数据分组到一起 | <code>rdd.cogroup(other)</code> | <code>{(1,([2],[])), (3, ([4, 6],[9])))}</code> |

将两组键值对RDD一起使用是对键值对数据执行的最有用的操作之一。例如外连接，内连接。

默认情况下，连接操作会将两个数据集中的所有键的哈希值都求出来，将该哈希值相同的记录通过网络传到同一台机器上，然后在那台机器上对所有键相同的记录进行连接操作

RDD

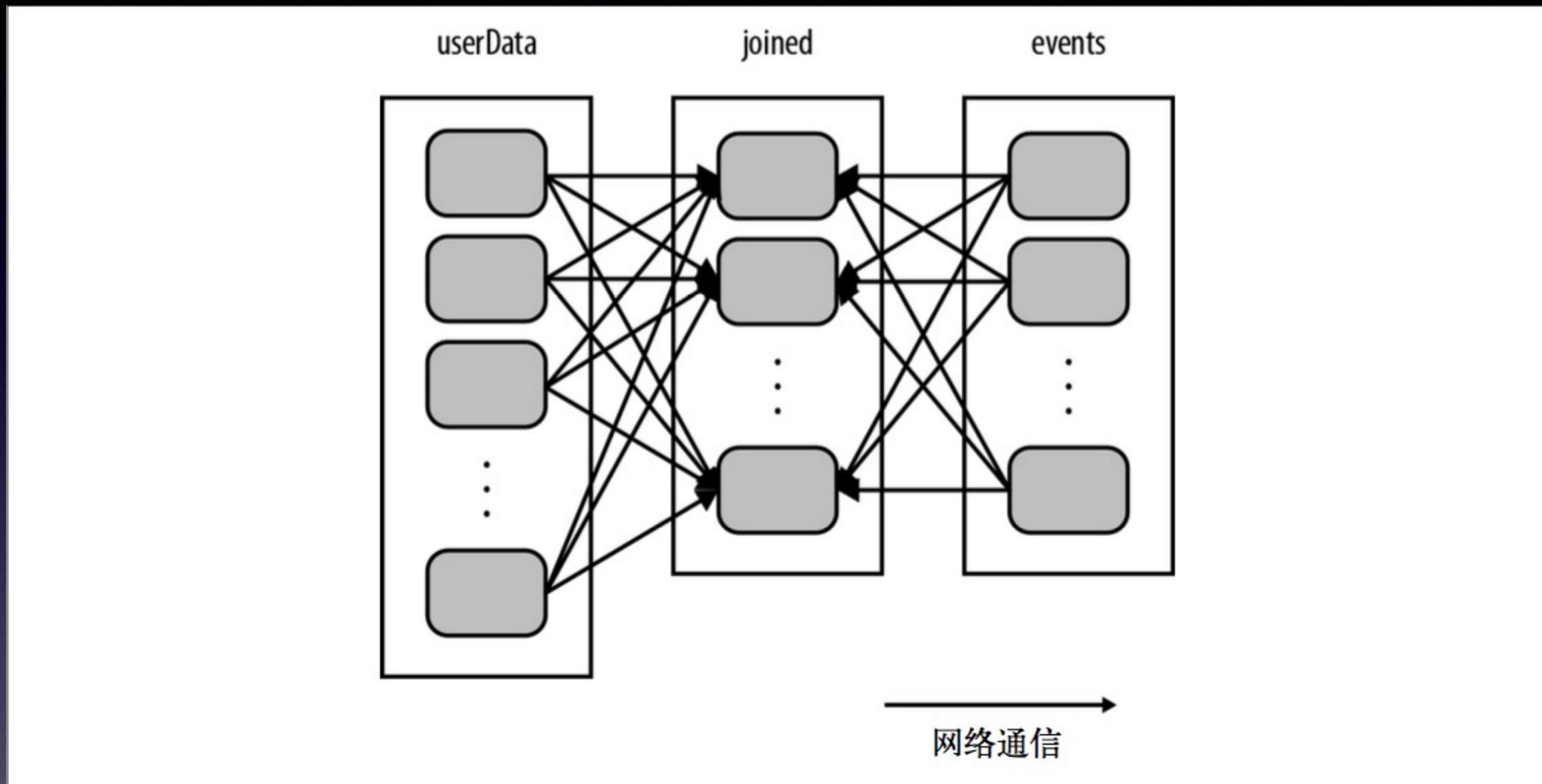


RDD就是一种伪集合，它支持并交等集合操作

RDD的连接 (不分区)

```
// 初始化代码;从HDFS商的一个Hadoop SequenceFile中读取用户信息
// userData中的元素会根据它们被读取时的来源，即HDFS块所在的节点来分布
// Spark此时无法获知某个特定的UserID对应的记录位于哪个节点上
val sc = new SparkContext...
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()
// 假设会周期性调用函数来处理过去五分钟产生的事件日志
// 假设这是一个包含(UserID, LinkInfo)对的SequenceFile
def processNewLogs(logFileName: String) {
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
    val joined = userData.join(events)// RDD of (UserID, (UserInfo, LinkInfo)) pairs
    val offTopicVisits = joined.filter {
        case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
            !userInfo.topics.contains(linkInfo.topic)
    }.count()
    println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
//userData很大， events很小
```

RDD的分区（重要）

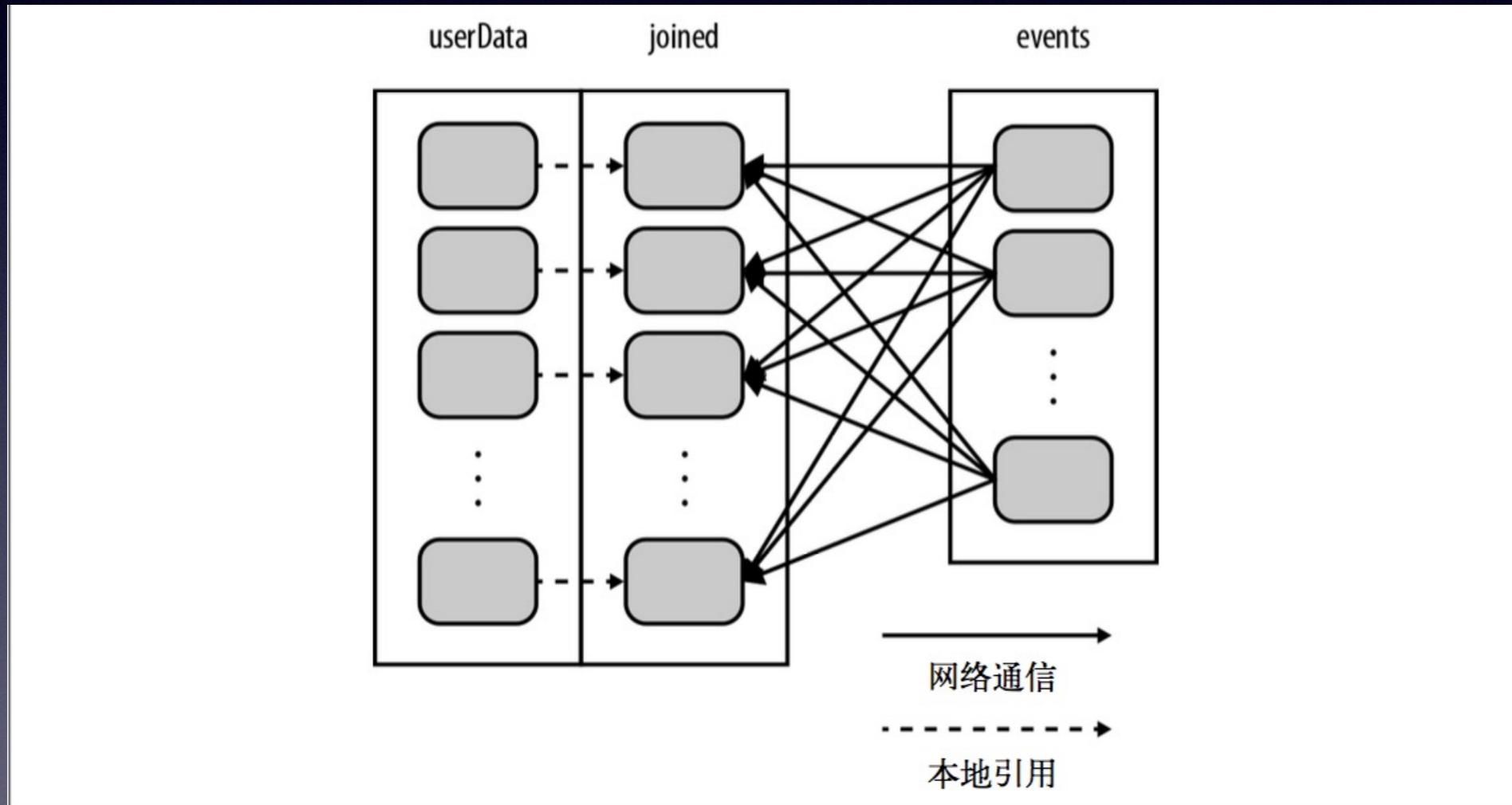


缺点，不够高效

`userData`远大于`events`，在每次调用时都对`userData`表进行哈希值计算和跨节点数据混洗，浪费资源与时间

RDD的分区（重要）

```
val sc = new SparkContext(...)  
    val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")  
.partitionBy(new HashPartitioner(100)) .persist() // 构造100个分区
```



在构建 `userData` 时调用了 `partitionBy()`, Spark 就知道了该 RDD 是根据键的哈希值来分区的, 这样在调用 `join()` 时, Spark 就会利用到这一点。具体来说, 当调用 `userData.join(events)` 时, Spark 只会对 `events` 进行数据混洗操作。

如果没有将 `partitionBy()` 转化操作的结果持久化, 那么后面每次用到这个 RDD 时都会重复地对数据进行分区操作。分区的好处就会被抵消。

除了join操作外还有哪些操作非常需要分区操作呢

就Spark 1.0而言，能够从数据分区中获益的操作有cogroup()、 groupWith()、 join()、 leftOuterJoin()、 rightOuterJoin()、 groupByKey()、 reduceByKey()、 combineByKey() 以及 lookup() 。

但是对于像 reduceByKey() 这样只作用于单个 RDD 的操作，运行在未分区的 RDD 上的时候会 导致每个键的所有对应值都在每台机器上进行本地计算，只需要把本地最终归约出的结果值从各工作节点传回主节点，所以原本的网络开销就不算大。而对于诸如 cogroup() 和 join() 这样的二元操作，预先进行数据分区会导致其中至少一个 RDD 不发生数据混淆。

获取RDD分区方式

输出数据的分区方式取决于父 RDD 的分区方式。但是像map, flatmap等函数不一定会按已知的分区方式分区，可以用partitioner获取分区方式。

```
scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <console>:14
scala> partitioned.partition
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

RDD分区典型应用：PageRank

PageRank 算法是以 Google 的拉里·佩吉(Larry Page)的名字命名的，用来根据外部文档指向一个 文档的链接，对集合中每个文档的重要程度赋一个度量值。该算法可以用于对网页进行排序，当然，也可以用于排序科技文章或社交网络中有影响的用户。

```
// 假设相邻页面列表以Spark objectFile的形式存储
val links = sc.objectFile[(String, Seq[String])]("links")
    .partitionBy(new HashPartitioner(100))
    .persist()

// 将每个页面的排序值初始化为1.0;由于使用mapValues，生成的RDD 的分区方式会和
// "links"的一样
var ranks = links.mapValues(v => 1.0)

// 运行10轮PageRank迭代
for(i <- 0 until 10) {
    val contributions = links.join(ranks).flatMap {
        case (pageId, (links, rank)) =>
            links.map(dest => (dest, rank / links.size))
    }
    ranks = contributions.reduceByKey((x, y) => x + y).mapValues(v => 0.15 + 0.85*v)
}

// 写出最终排名
ranks.saveAsTextFile("ranks")
```

PageRank 是执行多次连接的一个迭代算法，因此它是 RDD 分区操作的一个很好的用例。 算法会维护两个数据集:一个由(pageID, linkList)的元素组成，包含每个页面的相邻页面的列表;另一个由(pageID, rank)元素组成，包含每个页面的当前排序值。它按如下步骤进行计算：

- (1) 将每个页面的排序值初始化为 1.0。
- (2) 在每次迭代中，对页面 p，向其每个相邻页面(有直接链接的页面)发送一个值为 $\text{rank}(p)/\text{numNeighbors}(p)$ 的贡献值。
- (3) 将每个页面的排序值设为 $0.15 + 0.85 * \text{contributionsReceived}$ 。

linksRDD 在每次迭代中都会和 ranks 发生连接操作。由于 links 是一个静态数据集，所以我们在程序一开始的时候就对它进行了分区操作，这样就不需要把它通过网络进行数据混洗了。实际上，linksRDD 的字节数一般来说也会比 ranks 大很多，因此节约了相当可观的网络通信开销。

在循环体中，我们在 `reduceByKey()` 后使用 `mapValues()`;因为 `reduceByKey()` 的结果已经是哈希分区的了，调用 `map()` 时，结果不会有固定分区方式。这样一来，下一次循环中将映射操作的结果再次与 links 进行连接操作时就会更加高效。（因为两个数据集同时分区是不会发生数据混洗的）

共享变量

- Spark是用Scala写的，所以有很多的函数式编程的特性，函数式编程里进行迭代是会将所有变量拷贝为副本，对副本进行操作。
- 当一个传递给Spark操作(例如map和reduce)的函数在远程节点上面运行时，Spark操作实际上操作的是这个函数所用变量的一个独立副本。这些变量被复制到每台机器上，并且这些变量在远程机器上的所有更新都不会传递回驱动程序。

累加器Accumulator

```
val sc = new SparkContext(...)  
val file = sc.textFile("file.txt")  
val blankLines = sc.accumulator(0) // 创建Accumulator[Int]并初始化为0  
val callSigns = file.flatMap(line => {  
    if (line == "") {  
        blankLines += 1 // 累加器加1  
    }  
    line.split(" ")  
})  
callSigns.saveAsTextFile("output.txt")  
println("Blank lines: " + blankLines.value)
```

共享变量的一种：类似于静态全局变量，因为向 Spark 传递函数时，集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。累加器，提供了将工作节点中的值聚合到驱动器程序中的简单语法。

累加器Accumulator

使用注意：在行动操作中使用

对于要在行动操作中使用的累加器，Spark 只会把每个任务对各累加器的修改应用一次，但必须把它放在 `foreach()` 这样的行动操作中。对于在 RDD 转化操作中使用的累加器，转化操作中累加器可能会发生不止一次更新。在转化操作中，累加器通常只用于调试目的。例如，当一个被缓存下来但是没有经常使用的 RDD 在第一次从 LRU 缓存中被移除并又被重新用到时，这种非预期的多次更新就会发生。

广播变量broadcast

Spark 的第二种共享变量类型是广播变量，它可以让程序高效地向所有工作节点发送一个 较大的只读值，以供一个或多个 Spark 操作使用。

原因：Spark 会自动把闭包中所有引用到的变量发送到工作节点上，但是你可能会在多个并行操作中使用同一个变量，而 Spark 会为每个操作分别发送。

使用场景：例如，如果你的应用需要向所有节点发 送一个较大的只读查询表，甚至是机器学习算法中的一个很大的特征向量。

```
signPrefixes = loadCallSignTable()
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)
countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+y)))
```

```
# 查询RDD contactCounts中的呼号的对应位置。将呼号前缀 # 读取为国家代码来进行查询
signPrefixes = sc.broadcast(loadCallSignTable())
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)
countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+y)))
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

广播变量Python

```
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
    val country = lookupInArray(sign, signPrefixes.value)
    (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

广播变量Scala

```
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable()); JavaPairRDD<String, Integer>
countryContactCounts = contactCounts.mapToPair(
    new PairFunction<Tuple2<String, Integer>, String, Integer> (){
        public Tuple2<String, Integer> call(Tuple2<String, Integer> callSignCount) {
            String sign = callSignCount._1();
            String country = lookupCountry(sign, callSignInfo.value());
            return new Tuple2(country, callSignCount._2());
        }).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

广播变量Java

WinStar 陈光跃

基于分区进行操作

基于分区对数据进行操作可以让我们避免为每个数据元素进行重复的配置工作。诸如打开 数据库连接或创建随机数生成器等操作，都是我们应当尽量避免为每个元素都配置一次的工作。Spark 提供基于分区的 map 和 foreach，让你的部分代码只对 RDD 的每个分区运行一次，这样可以帮助降低这些操作的代价。

回到呼号的示例程序中来，我们有一个在线的业余电台呼号数据库，可以用这个数据库查询日志中记录过的联系人呼号列表。通过使用基于分区的操作，可以在每个分区内共享一个数据库连接池，来避免建立太多连接，同时还可以重用 JSON 解析器。使用 mapPartitions 函数获得输入 RDD 的每个分区中的元素迭代器，而需要返回的是执行结果的序列的迭代器。

```
def processCallSigns(signs):
    """使用连接池查询呼号"""
    # 创建一个连接池
    http = urllib3.PoolManager()
    # 与每条呼号记录相关联的URL
    urls = map(lambda x: "http://73s.com/qso/%s.json" % x, signs) # 创建请求(非阻塞)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # 获取结果
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests) # 删除空的结果并返回
    return filter(lambda x: x[1] is not None, result)
def fetchCallSigns(input): """获取呼号"""
    return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))
contactsContactList = fetchCallSigns(validSigns)
```

```

val contactsContactLists = validSigns.distinct().mapPartitions{
    signs =>
    val mapper = createMapper() val client = new HttpClient() client.start()
    // 创建http请求
    signs.map {sign =>
        createExchangeForSign(sign)
    }
    // 获取响应
}.map{ case (sign, exchange) =>
(sign, readExchangeCallLog(mapper, exchange)) }.filter(x => x._2 != null) // 删除空的呼叫日志
}

```

```

JavaPairRDD<String, CallLog[]> contactsContactLists =
validCallSigns.mapPartitionsToPair(
    new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input) { // 列出结果
ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>(); ArrayList<Tuple2<String, ContentExchange>>
requests = new ArrayList<>(); ObjectMapper mapper = createMapper();
HttpClient client = new HttpClient();
try {
    client.start();
    while (input.hasNext()) {
        requests.add(createRequestForSign(input.next(), client));
    }
    for (Tuple2<String, ContentExchange> signExchange : requests) {
        callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
    }
} catch (Exception e) {
}
return callsignLogs;
});
System.out.println(StringUtils.join(contactsContactLists.collect(), ","));

```

PartitionMap的理解

```
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])
def basicAvg(nums): """计算平均值"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

```
def partitionCtr(nums): """计算分区的sumCounter"""
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]
def fastAvg(nums): """计算平均值"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

Spark的运行流程与WordCount

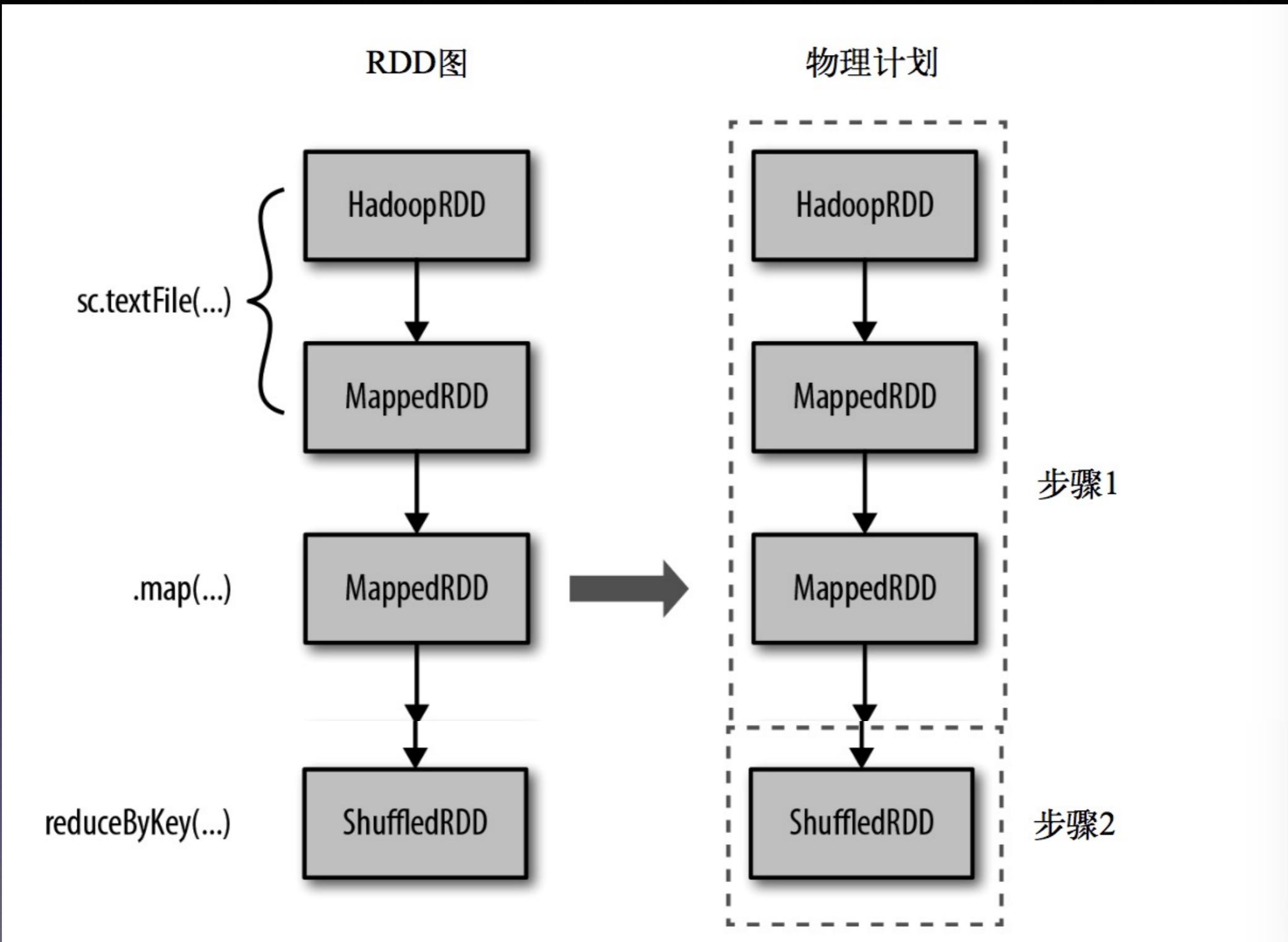
```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object WordCount {
    def main(args: Array[String]){
        val conf = new SparkConf().setAppName("wordCount")

        val sc = new SparkContext(conf)
        // 读取我们的输入数据
        val input = sc.textFile("/WordCount/input/words.txt")
        // 把它切分成一个个单词
        val words = input.flatMap(line => line.split(" "))
        // 转换为键值对并计数
        val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y}
        // 将统计出来的单词总数存入一个文本文件，引发求值
        counts.saveAsTextFile("/data/sparkOut/")

    }
}
```

运行流程



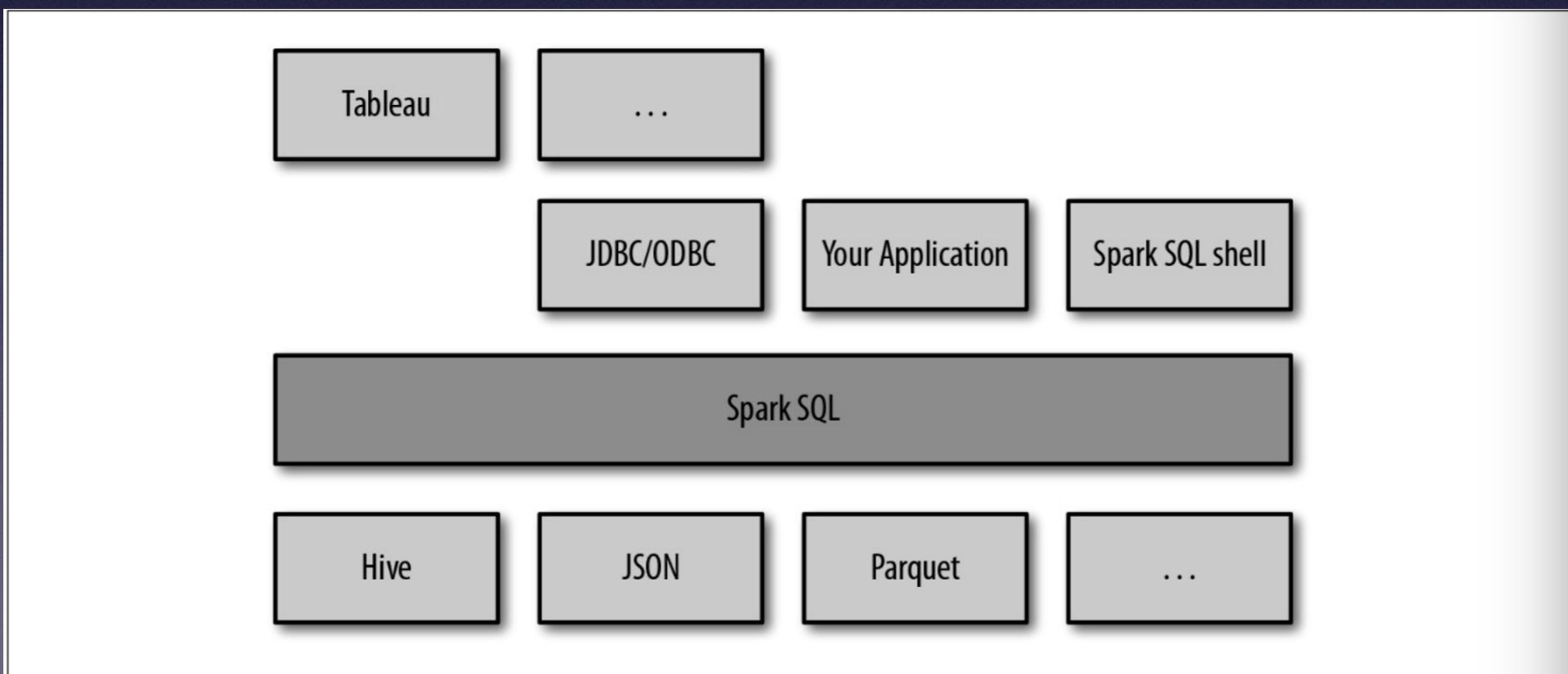
DAG

- 所有的 Spark 程序都遵循同样的结构:程序从输入数据创建一系列 RDD, 再使用转化操作派生出新的 RDD, 最后使用行动操作收集或存储结果 RDD 中的数据。Spark 程序其实是隐式地创建出了一个由操作组成的逻辑上的有向无环图 (Directed Acyclic Graph, 简称 DAG)。当驱动器程序运行时, 它会把这个逻辑图转为物理执行计划。
- Spark 会对逻辑执行计划作一些优化, 比如将连续的映射转为流水线化执行, 将多个操作合并到一个步骤中等。这样 Spark 就把逻辑计划转为一系列步骤(stage)。而每个步骤又由多个任务组成。这些任务会被打包并送到集群中。任务 (Task) 是 Spark 中最小的工作单元, 用户程序通常要启动成百上千的独立任务。

Spark SQL

Spark SQL 可以从各种结构化数据源(例如 JSON、Hive、Parquet 等)中读取数据。

Spark SQL提供了一种特殊的RDD，叫作SchemaRDD。¹ SchemaRDD 是存放 Row 对象的 RDD，每个 Row 对象代表一行记录。SchemaRDD 还包含记录的结构信息(即数据字段)。



Spark Streaming

许多应用需要实时处理收到的数据，例如追踪页面访问统计的应用、训练机器学习模型的应用，还有自动检测异常的应用。Spark Streaming是Spark为这些应用而设计的模型。它允许用户使用一套和批处理非常接近的 API 来编写流式计算应用，这样就可以大量重用批处理应用的技术甚至代码。

和Spark基于RDD的概念很相似，Spark Streaming使用离散化流(discretized stream)作为抽象表示，叫作 DStream。DStream 是随时间推移而收到的数据的序列。在内部，每个 时间区间收到的数据都作为 RDD 存在，而 DStream 是由这些 RDD 所组成的序列(因此 得名“离散化”)。DStream 可以从各种输入源创建，比如 Flume、Kafka 或者 HDFS。创建出来的 DStream 支持两种操作，一种是转化操作(transformation)，会生成一个新的 DStream，另一种是输出操作(output operation)，可以把数据写入外部系统中。DStream 提供了许多与 RDD 所支持的操作相类似的操作支持，还增加了与时间相关的新操作，比如滑动窗口。

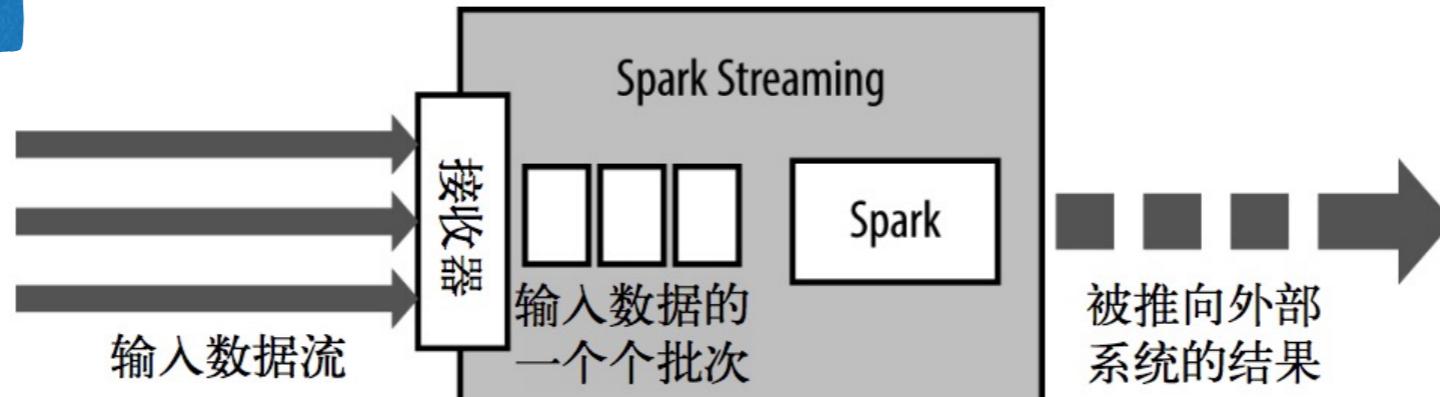
和批处理程序不同，Spark Streaming应用需要进行额外配置来保证24/7不间断工作。如检查点(checkpointing)机制，也就是把数据存储到可靠文件系统(比如 HDFS) 上的机制，这也是Spark Streaming用来实现不间断工作的主要方式。

流与实时处理

```
// 从SparkConf创建StreamingContext并指定1秒钟的批处理大小  
val ssc = new StreamingContext(conf, Seconds(1))  
// 连接到本地机器7777端口上后，使用收到的数据创建DStream  
val lines = ssc.socketTextStream("localhost", 7777)  
// 从DStream中筛选出包含字符串"error"的行  
val errorLines = lines.filter(_.contains("error"))  
// 打印出有"error"的行  
errorLines.print()
```

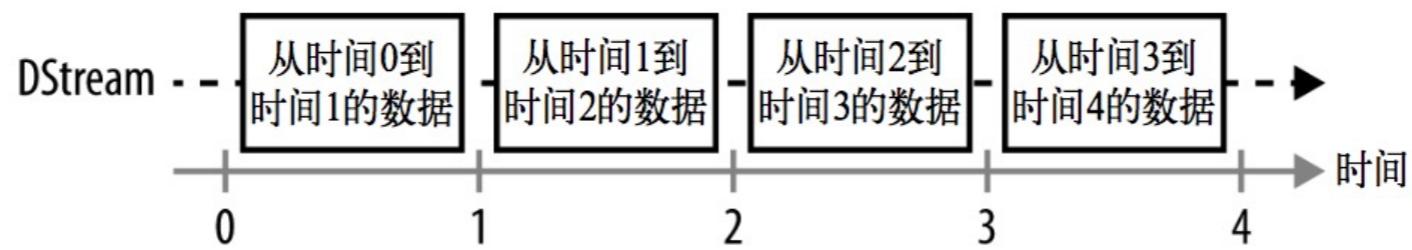
```
// 从SparkConf创建StreamingContext并指定1秒钟的批处理大小  
JavaStreamingContext jssc = new JavaStreamingContext(conf,  
Durations.seconds(1));  
// 以端口7777作为输入来源创建DStream  
JavaDStream<String> lines = jssc.socketTextStream("localhost", 7777);  
// 从DStream中筛选出包含字符串"error"的行  
JavaDStream<String> errorLines = lines.filter(new Function<String,  
Boolean>() {  
    public Boolean call(String line) {  
        return line.contains("error");  
    }  
});  
// 打印出有"error"的行  
errorLines.print();
```

流与准实时处理



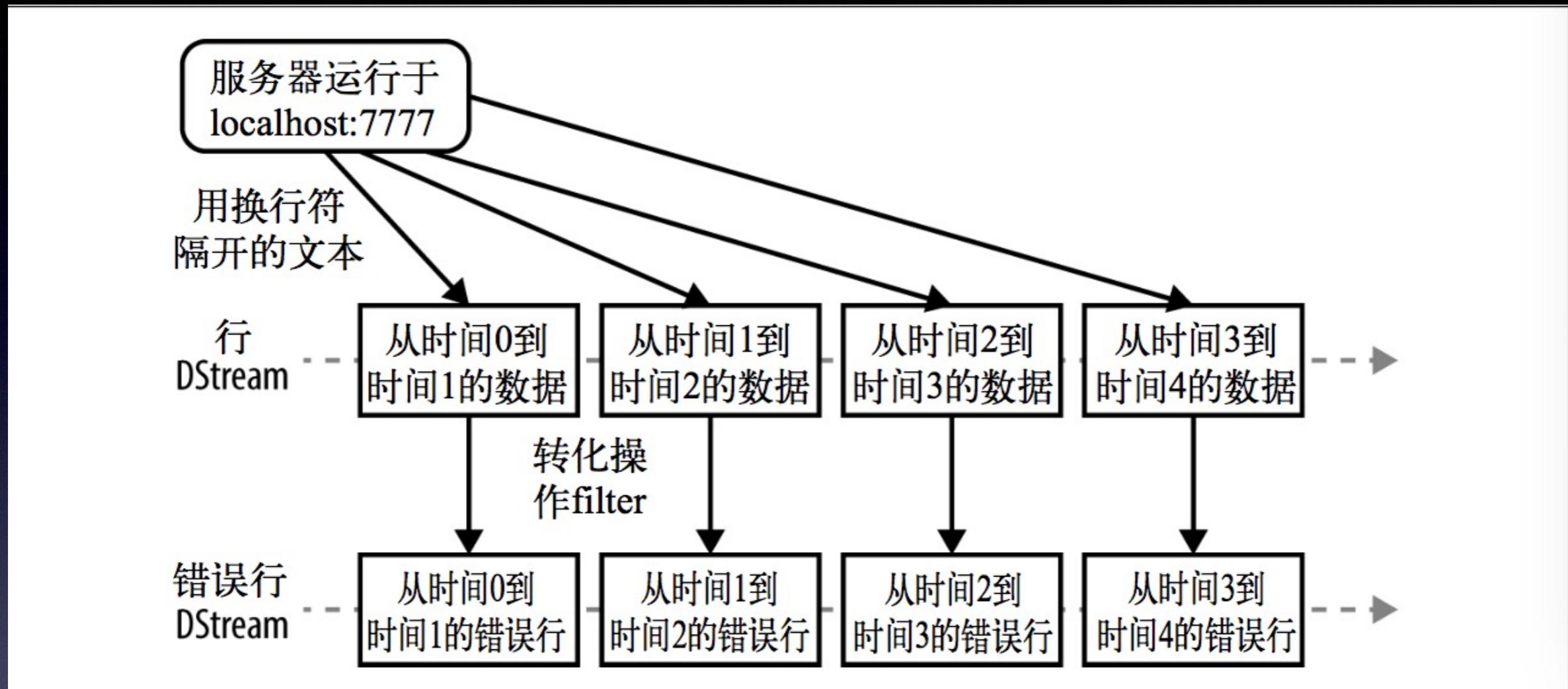
Spark Streaming从各种输入源中读取数据，并把数据分组为小的批次。新的批次按均匀的时间间隔创建出来。

批次间隔一般设在 500 毫秒到几秒之间，由应用开发者配置。每个输入批次都形成一个 RDD，以 Spark 作业的方式处理并生成其他的 RDD。处理的结果可以以批处理的方式传给外部系统。高层次的架构如图



Spark Streaming的编程抽象是离散化流，也就是DStream(如图10-2所示)。它是一个 RDD 序列，每个 RDD 代表数据流中一个时间片内的数据。

流与准实时处理



示例程序的微批次流程

数据源

Spark Streaming原生支持一些不同的数据源。一些“核心”数据源（如文件流）已经被打包到Spark Streaming 的 Maven 工件中，而其他的一些则可以通过 spark-streaming-kafka 等附加工件获取。

附加数据源

Apache Kafka因其速度与弹性成为了一个流行的输入源。使用 Kafka 原生的支持，可以轻松处理许多主题的消息。在工程中需要引入 Maven 工件 spark-streaming-kafka_2.10 来使用它。包内提供的 KafkaUtils 对象可以在 StreamingContext 和 JavaStreamingContext 中以你的 Kafka 消息创建出 DStream。由于 KafkaUtils 可以订阅多个主题，因此它创建出的 DStream 由成对的主题和消息组成。要创建出一个流数据，需要使用 StreamingContext 实例、一个由逗号隔开的 ZooKeeper 主机列表字符串、消费者组的名字(唯一名字)，以及一个从主题到针对这个主题的接收器线程数的映射表来调用 createStream() 方法。另外还有其他的附加数据源，如Apache Flume的推式接收器，拉式接收器。

示例：用 Apache Kafka 订阅 Panda 主题

```
import org.apache.spark.streaming.kafka.*;  
...  
// 创建一个从主题到接收器线程数的映射表  
Map<String, Integer> topics = new HashMap<String, Integer>();  
topics.put("pandas", 1);  
topics.put("logs", 1);  
JavaPairDStream<String, String> input =  
    KafkaUtils.createStream(jssc, zkQuorum, group, topics);  
input.print();
```

```
import org.apache.spark.streaming.kafka._  
...  
// 创建一个从主题到接收器线程数的映射表  
val topics = List(("pandas", 1), ("logs", 1)).toMap  
val topicLines = KafkaUtils.createStream(ssc, zkQuorum, group, topics)  
StreamingLogInput.processLines(topicLines.map(_._2))
```

检查点机制

检查点机制是我们在Spark Streaming中用来保障容错性的主要机制。它可以使Spark Streaming阶段性地把应用数据存储到诸如HDFS或Amazon S3这样的可靠存储系统中，以供恢复时使用。具体来说，检查点机制主要为以下两个目的服务。

- 控制发生失败时需要重算的状态数。我们在10.2节中讨论过，SparkStreaming可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。
 - 提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样Spark Streaming就可以读取之前运行的程序处理数据的进度，并从那里继续。
- 出于这些原因，检查点机制对于任何生产环境中的流计算应用都至关重要。你可以通过向 `ssc.checkpoint()` 方法传递一个路径参数(HDFS、S3 或者本地路径均可)来配置检查点机制

驱动器容错

驱动器程序的容错要求我们以特殊的方式创建 StreamingContext。我们需要把检查点目录提供给 StreamingContext。需要使用 StreamingContext.getOrCreate() 函数。以下是配置一个可以从错误中恢复的驱动器程序

```
def createStreamingContext() = {
    ...
    val sc = new SparkContext(conf)
    // 以1秒作为批次大小创建StreamingContext
    val ssc = new StreamingContext(sc, Seconds(1)) ssc.checkpoint(checkpointDir)
}
...
val ssc = StreamingContext.getOrCreate(checkpointDir, createStreamingContext _)
```

```
JavaStreamingContextFactory fact = new JavaStreamingContextFactory() {
    public JavaStreamingContext call() {
        ...
        JavaSparkContext sc = new JavaSparkContext(conf);
        // 以1秒作为批次大小创建StreamingContext
        JavaStreamingContext jssc = new JavaStreamingContext(sc, Durations.seconds(1));
        jssc.checkpoint(checkpointDir);
        return jssc;
    };
    JavaStreamingContext jssc = JavaStreamingContext.getOrCreate(checkpointDir, fact);
```