大数据技术基础

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

第四章 大数据处理
——分布式内存计算框架Spark

鄂海红 计算机学院 教授
ehaihong@bupt.edu.cn 微信: 87837001 QQ: 3027581960

大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Spark简介

什么是Spark

- Apache Spark是开源的基于内存的迭代计算框架
- 已经成为大数据计算领域最热门的技术之一
- 基于RDD提供了比Hadoop更加丰富的计算模型
- 支持复杂的机器学习、图计算和准实时流处理等
- 效率更高，速度更快（相对Hadoop 的MapReduce ）

- Lightning-fast unified analytics engine
- （快如闪电的计算引擎）　　——Spark官方

Spark在Hadoop生态的位置

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 在Hadoop生态圈中Spark提供了一个更快、更通用的数据处理平台

Spark的发展

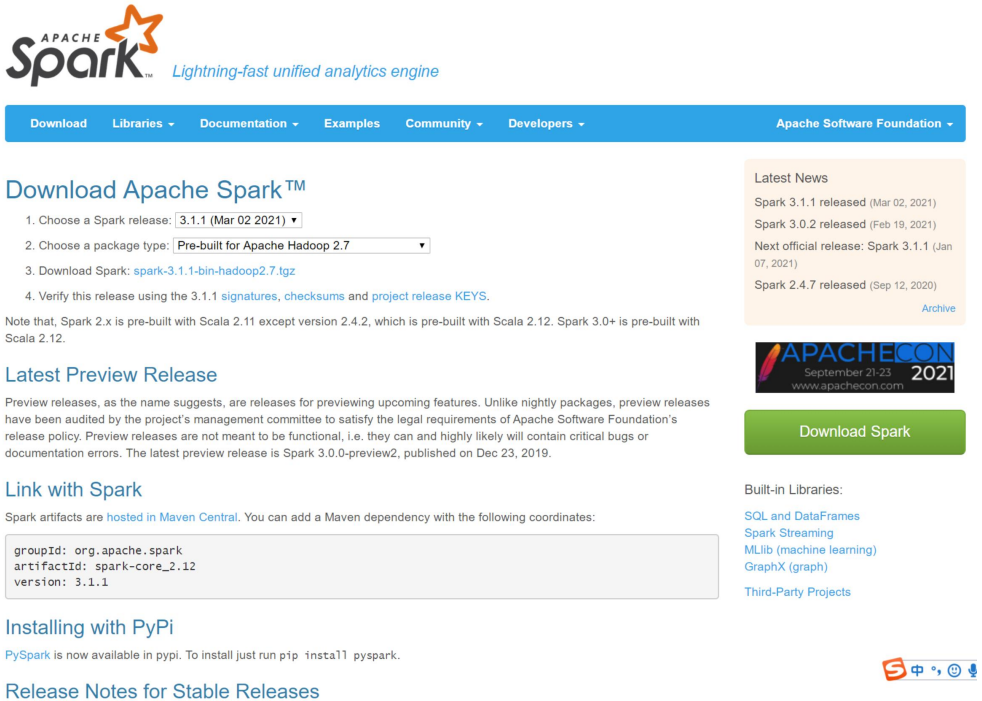
北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

年份	事件
2009年	由Matei Zaharia在加州大学伯克利分校的AMP实验室开发
2010年	通过BSD授权条款发布开放源码
2012年	Spark 0.6.0版本发布，大范围的性能改进，增加了一些新特性，并对Standalone部署模式进行了简化
2013年	该项目被捐赠给Apache软件基金会
2014年	Spark成为Apache的顶级项目
2016年	Spark 1.6.0发布，该版本含了超过1000个补丁，主要的改进：新的Dataset API，性能提升，以及大量新的机器学习和统计分析算法
2018年	Spark 2.3.0发布，此版本增加了对 Structured Streaming 中的 Continuous Processing 以及全新的 Kubernetes Scheduler 后端的支持

Spark目前现状

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 自从Spark将其代码部署到GitHub上之后，截止2018年11月份一共有23093次提交，19个分支，82次发布，1296位代码贡献者
- Spark开源社区的活跃度相当高，目前仍然是最受欢迎的集群运算框架之一
- Spark官方网站的网址是：http://spark.apache.org/
- 截止2021年3月Spark最新版本是：Spark 3.1.1



http://spark.apache.org/downloads.html

大数据处理的任务场景

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 复杂的批量处理（Batch Data Processing），偏重点在于处理海量数据的能力，通常的时间可能是在数十分钟到数小时
- 基于历史数据的交互式查询（Interactive Query），通常的时间在数十秒到数十分钟之间
- 基于实时数据流的数据处理（Streaming Data Processing），通常在数百毫秒到数秒之间

- 复杂的批量处理可以使用MapReduce
- 交互式查询可以使用Impala
- 实时数据流的数据处理可以使用Storm

- MapReduce 、Impala 、Storm这些框架相互独立，需要各自维护，成本较高
- Spark的出现能够一站式平台满足以上需求，大大的降低了维护成本

Hadoop MapReduce的慢

- **Hadoop MapReduce一个最大的问题是在很多应用场景中速度非常慢**
  - 只适合离线的计算任务
- **MapReduce需要将任务划分成map和reduce两个阶段**
  - map阶段产生的中间结果要写回磁盘
  - 在这两个阶段之间需要进行shuffle（洗牌）、sort（排序）操作
- **Shuffle操作需要从网络中的各个节点进行数据拷贝**
  - 使其往往成为最为耗时的步骤
  - 这也是Hadoop MapReduce慢的根本原因之一
  - 大量的时间耗费在网络磁盘IO中而不是用于计算

Hadoop MapReduce的慢

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

举例——WordCount：文档词频统计

- 使用四个map节点：map: (k1; v1) → [(k2; v2)]

map节点1:
输入(k1; v1) ： (text1, “the weather is good”)
输出 [(k2; v2)]： (the, 1), (weather, 1), (is, 1), (good, 1)

map节点2:
输入： (text2, “today is good”)
输出： (today, 1), (is, 1), (good, 1)

map节点3:
输入： (text3, “good weather is good”)
输出： (good, 1), (weather, 1), (is, 1), (good, 1)

map节点4:
输入： (text3, “today has good weather”)
输出： (today, 1), (has, 1), (good, 1), (weather, 1)

- map: (k1; v1) → [(k2; v2)]

输入： 键值对(k1; v1)表示的数据

处理： 文档数据记录(如文本文件中的行，或数据表格中的行)将以“键值对”形式传入map函数；map函数将处理这些键值对，并以另一种键值对形式输出处理的一组键值对中间结果[(k2; v2)]

输出： 键值对[(k2; v2)]表示的一组中间数据

Hadoop MapReduce的慢

举例——WordCount：文档词频统计

- 使用三个reduce节点： reduce: (k2; [v2]) → [(k3; v3)]

reduce节点1:

输入：(good, 1), (good, 1), (good, 1), (good, 1), (good, 1)

输出：(good, 5)

reduce节点2:

输入：(has, 1), (is,1), (is,1), (is, 1),

输出：(has, 1), (is, 3)

reduce节点3:

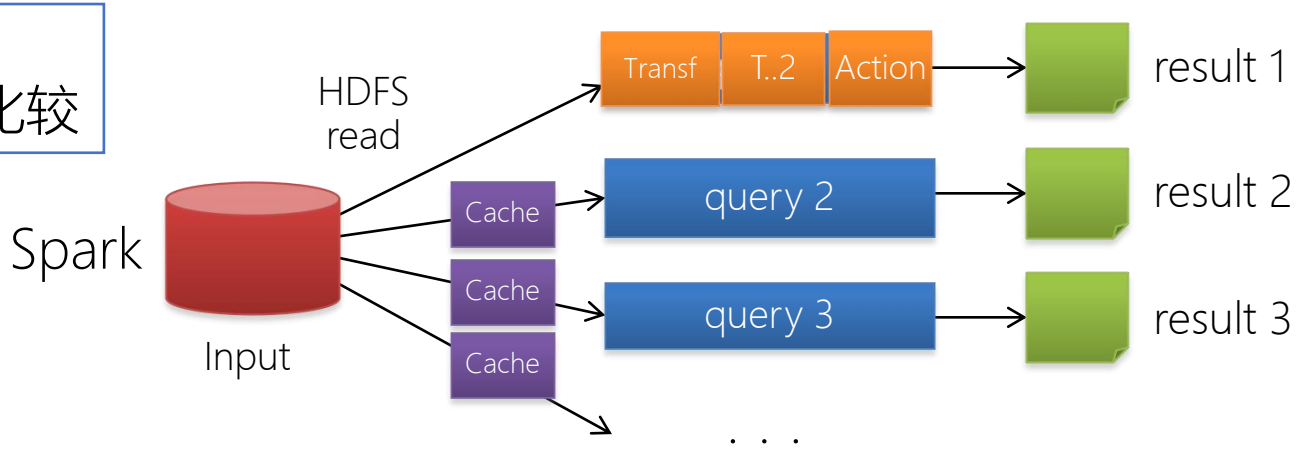
输入：(the, 1), (today, 1), (today, 1)
(weather, 1), (weather,1), (weather, 1)

输出：(the, 1), (today, 2), (weather, 3)

改为大部分处理在内存中进行

- MapReduce每次读写，都需要**序列化**到磁盘。一个复杂任务，需要多次处理，**几十次磁盘读写**
- Spark只需要一次磁盘读写，大部分处理在内存中进行

Spark和MapReduce比较

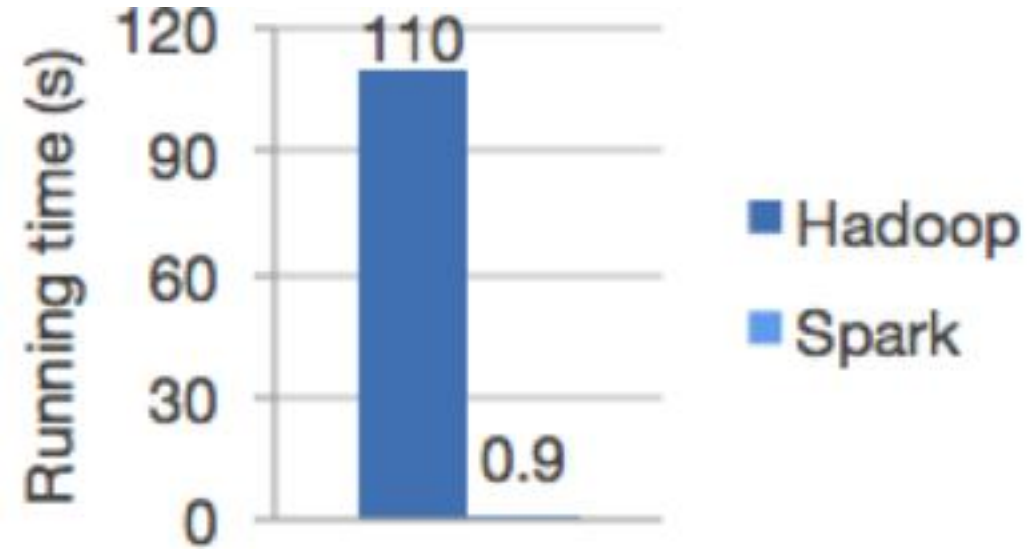


I/O and serialization can take 90% of the time

Spark的优点

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

速度快



- 左图是逻辑回归运算下，Hadoop与Spark的运行速度对比
- Spark在批处理和流式处理上都具有很好的性能

Spark的优点

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

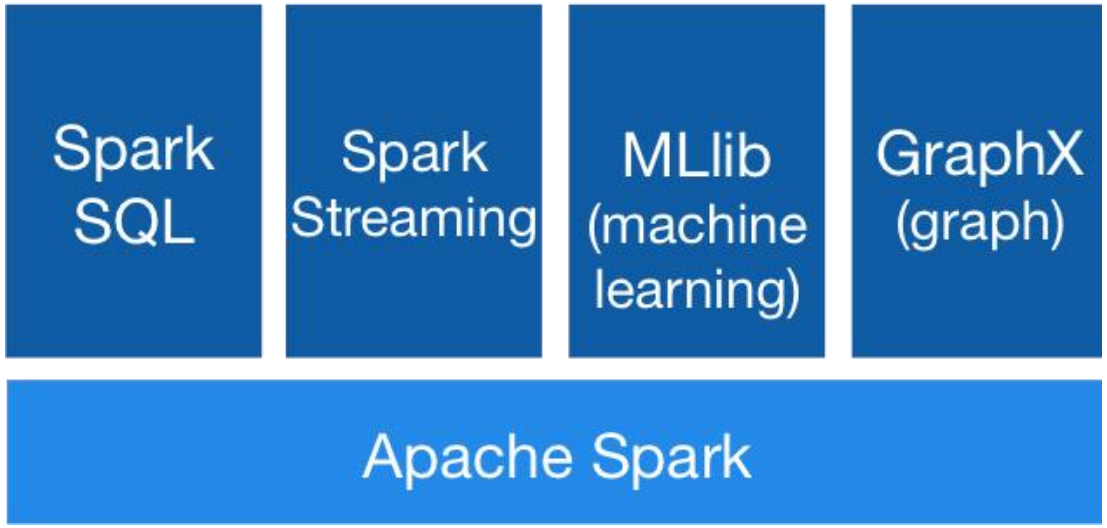
易使用

- 可以使用Java，Scala，Python，R，SQL快速写一个Spark应用
- Spark提供了超过80中操作（Transformation操作和Action操作）使它更容易生成平行化的应用
- 还可以使用Scala,Python,R,SQL shell 进行交互操作

Spark的优点

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

能通用



- Spark生态圈包含了Spark Core、Spark SQL、Spark Streaming、MLLib和GraphX等组件
- Spark Streaming的实时处理应用
- Spark SQL的即席查询
- MLlib的机器学习
- GraphX的图处理
- 它们能够无缝的集成并提供一站式解决平台

Spark的优点

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

随处用

- Spark能够很好的与Hadoop生态其他组件组合使用
- Spark能够读取HDFS、Cassandra、HBase、S3和Techyon为持久层读写原生数据
- 能够以Mesos、YARN和自身携带的Standalone作为资源管理器调度job，来完成Spark应用程序的计算

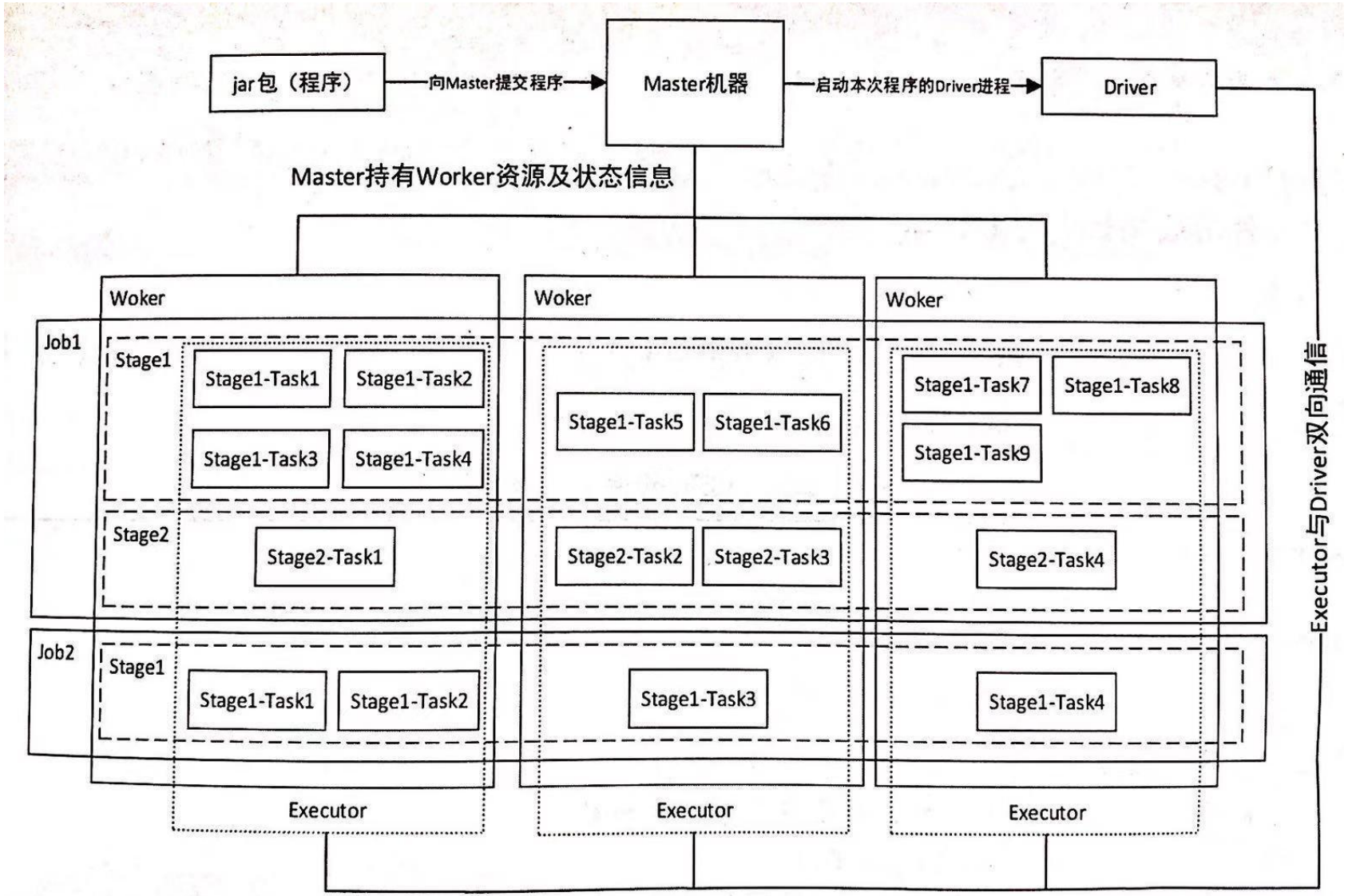
大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Spark框架

Spark专业术语

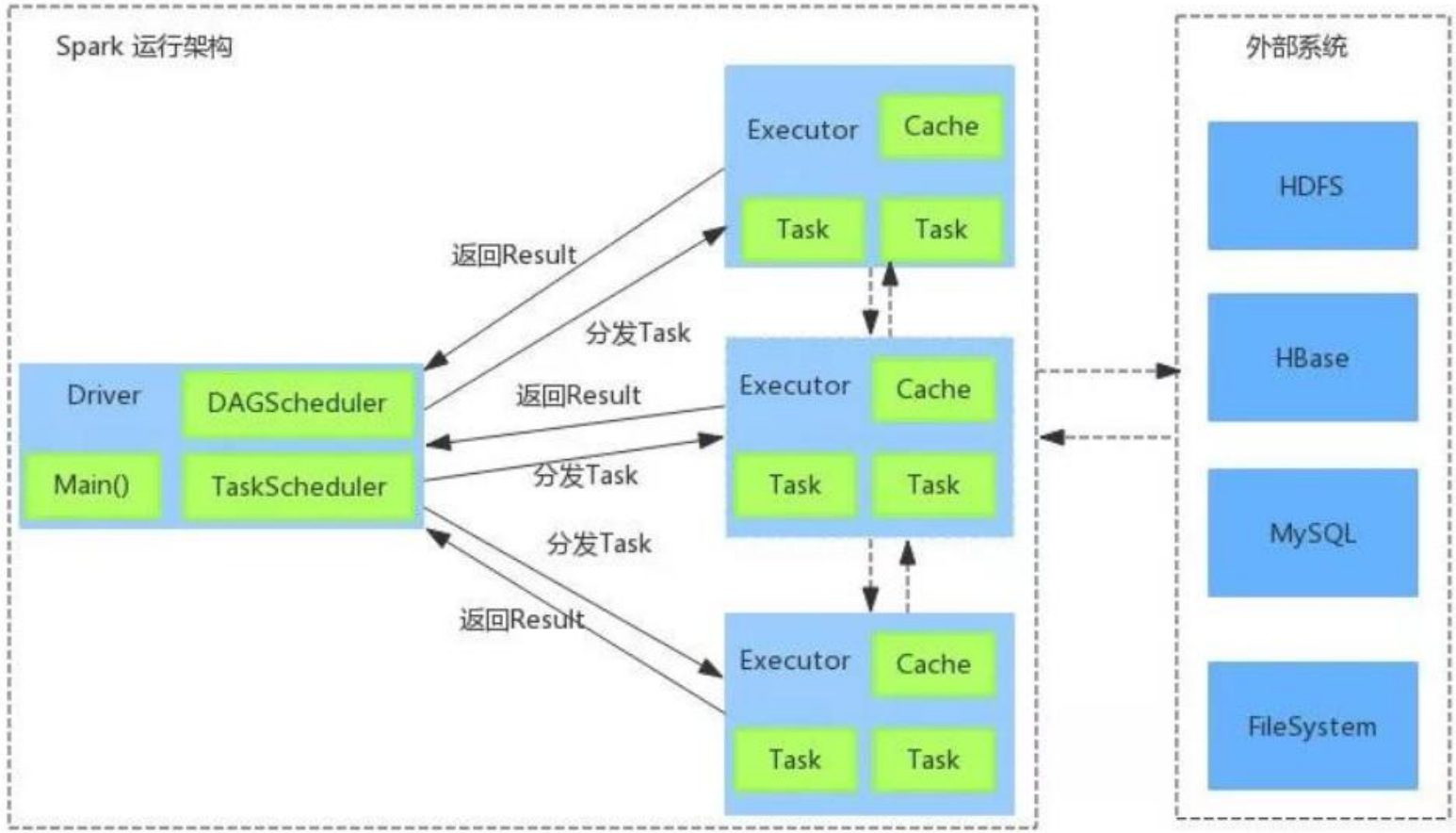
- 一个spark application提交后，陆续被分解为job、stage、Task
- Job：RDD执行一次Action操作就会生成一个Job
- Stage：DAGScheduler将Job划分为多个Stage，Stage的划分界限为Shuffle的产生，Shuffle标志着上一个Stage的结束和下一个Stage的开始
- TaskSet：每个Stage创建一个Taskset一组相关联的任务集
- Task：Spark运行的基本单位，负责处理RDD的计算逻辑
- Task是Spark最细的执行单元
- Task的数量反映了stage的并行度。



Spark运行架构

- Client：客户端进程，负责提交作业
- Driver（Driver program）：driver就是用户编写的spark应用程序，用来创建sparkcontext或者sparksession
- Driver主要负责Spark作业的解析，作业的main函数运行在Driver中。
- Driver还负责通过DAGScheduler划分Stage，将Stage转化成TaskSet提交给TaskScheduler任务调度器，进而调度Task到Executor上执行
- Executor：一个独立的JVM进程，负责执行Driver分发的Task任务。集群中一个节点可以启动多个Executor，每一个Executor可以执行多个Task任务

DAG（Directed Acyclic Graph）有向无环图：Spark实现了DAG的计算模型，DAG计算模型是指将一个计算任务按照计算规则分解为若干子任务，这些子任务之间根据逻辑关系构建成有向无环图。



Spark提交作业命令

spark-submit --class org.example.ScalaWordCount --master yarn --num-executors 3 --driver-memory 1g --executor-memory 1g --executor-cores 1 spark-test.jar

```
21/04/14 10:46:04 INFO scheduler.DAGScheduler: Job 1 finished: collect at ScalaWordCount.scala:19, took 0.080027
(hi,6),(hello,5),(spark,2),(sparkkgraphx,1),(sparkstreaming,1),(sparksql,1)21/04/14 10:46:04 INFO storage.BlockMa
t 3 piece0 on 192.168.0.138:44381 in memory (size: 2023.0 B, free: 366.3 MB)
```

```
./bin/spark-submit --class package.MainClass \    # 作业执行主类，需要完成的包路径
    --master spark://host:port, mesos://host:port, yarn, or local\Maste
                        # 运行方式
    ---deploy-mode client,cluster\ # 部署模式，如果Master采用YARN模式则可以选择使用clent模式或者cluster模式，默认client模式
    --driver-memory 1g \           # Driver运行内存，默认1G
    ---driver-cores 1 \          # Driver分配的CPU核个数
    --executor-memory 4g \       # Executor内存大小
    --executor-cores 1 \           # Executor分配的CPU核个数
    ---num-executors \           # 作业执行需要启动的Executor数
    ---jars \                 # 作业程序依赖的外部jar包，这些jar包会从本地上传到Driver然后分发到各Executor classpath中。
    --queue QUEUE_NAME \    #  提交应用程序给哪个YARN的队列，默认是default队列
    lib/spark-examples*.jar \      # 作业执行JAR包
[other application arguments ]       # 程序运行需要传入的参数
```

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode cluster \
--driver-memory 1g \
--executor-memory 1g \
--executor-cores 1 \
--queue thequeue \
examples/target/scala-2.11/jars/spark-examples*.jar 10
```

大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Spark安装

Spark安装

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 在master节点解压压缩包
- 配置spark的slaves、spark-env.sh文件
- 配置.bash_profile文件
- 并将spark目录及.bash_profile远程拷贝到其它从节点
- 在master节点启动spark集群
- 以spark standalone模式运行一个spark的计算pi的例子

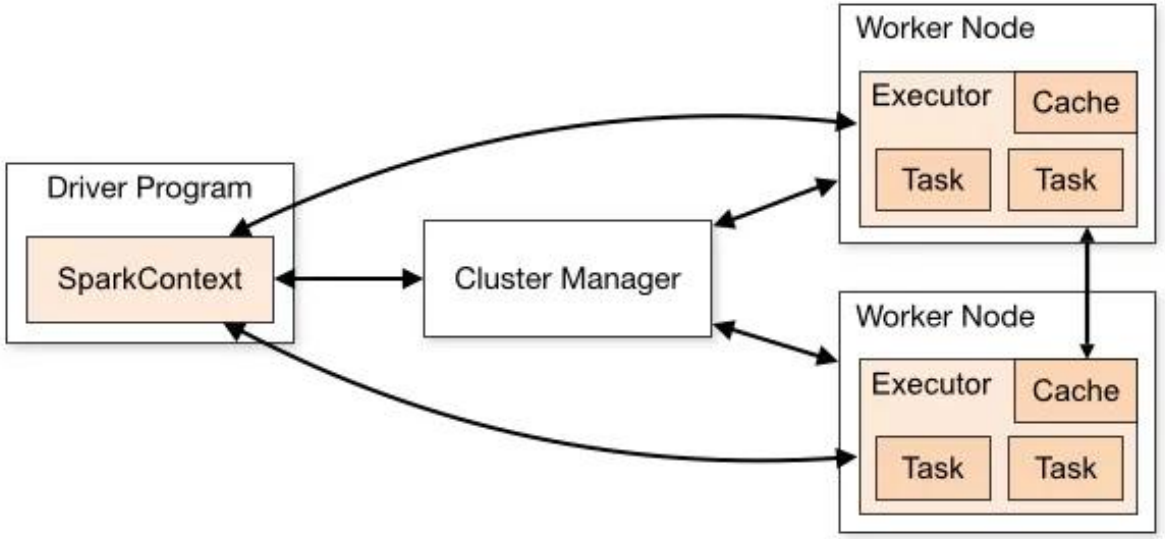
Spark启动模式

- Spark运行模式主要有以下几种：

1. local**单机模式**：单机的多个线程来模拟Spark分布式计算，主要用于开发测试。N个线程分别充当Driver和Executors。由于Driver和Executors处于同一个JVM，算子可以访问外部的变量。
2. local-cluster**单机伪分布式模式**：和Local很像，不同的是，它会在单机启动多个进程来模拟集群下的分布式场景，而不像Local这种多个线程只能在一个进程下委屈求全的共享资源。**通常也是用来验证开发出来的应用程序逻辑上有没有问题，或者想使用Spark的计算框架而没有太多资源。**用法是：提交应用程序时使用local-cluster[x,y,z]参数：x代表要生成的executor数，y和z分别代表每个executor所拥有的core和memory数。
3. Standalone独立运行**集群模式**：该模式是Spark自身实现的资源调度框架，由客户端、Master节点和多个Worker节点组成。其中SparkContext既可以运行在Master节点上，也可以运行在客户端。每个Worker节点上存在一个或多个Executor对象。 该对象持有一个线程池，每个线程可以执行一个task。
4. Spark On Yarn**集群模式**：Spark On Yarn有两种模式，分别为yarn-client和yarn-cluster模式。yarn-client模式中，Driver运行在客户端，其作业运行日志在客户端查看，适合返回小数据量结果集交互式场景使用。yarn-cluster模式中，Driver运行在集群中的某个节点，节点的选择由YARN调度，作业日志通过yarn管理名称查看：yarn logs -applicationId <app ID>，也可以在YARN的Web UI中查看，适合大数据量非交互式场景使用。

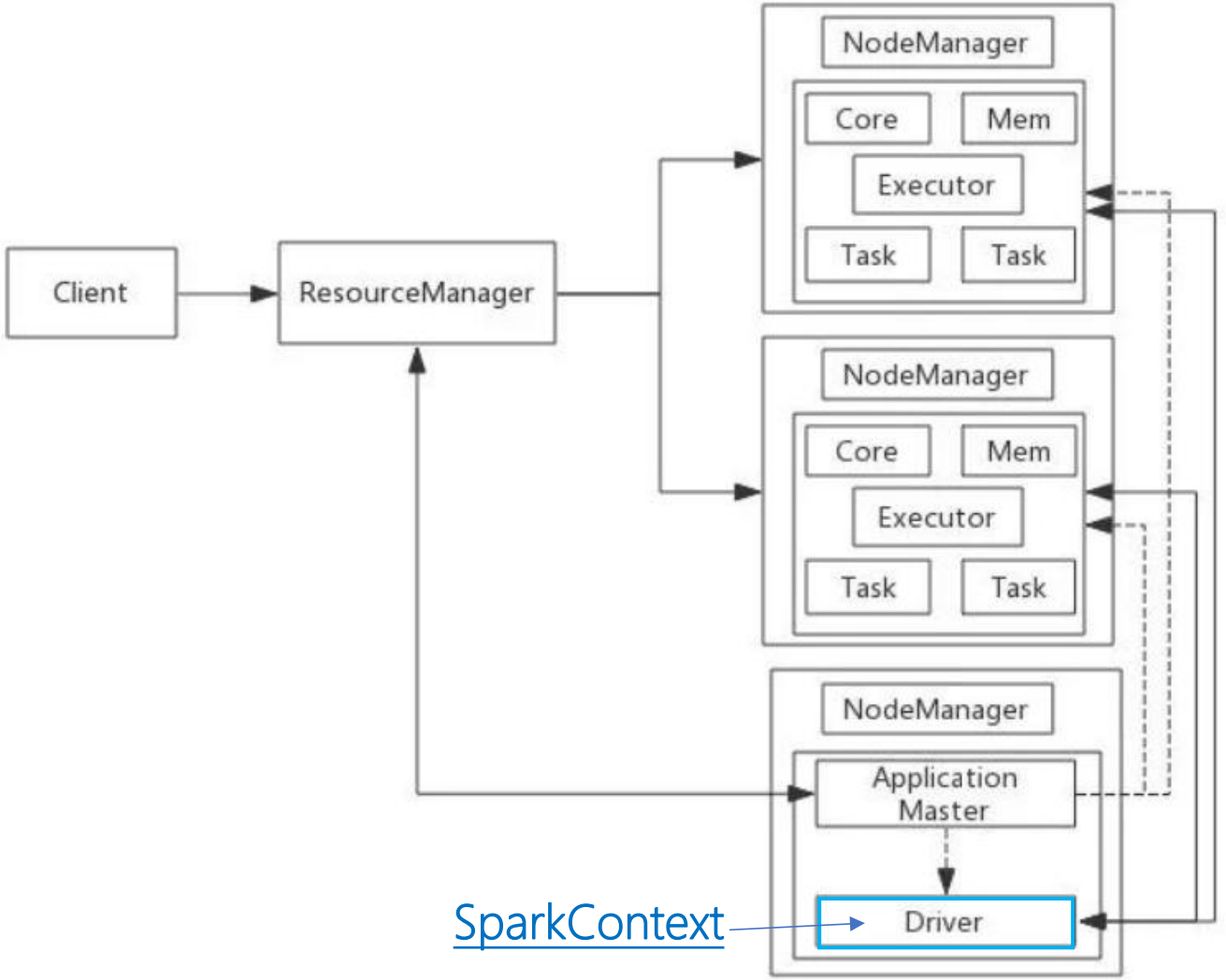
Spark运行架构本质

- Spark虽然有多种运行模式，但是其运行架构基本上由三部分组成
  - SparkContext
  - ClusterManager（集群资源管理器）
  - Executor（任务执行进程）
- SparkContext用于负责与ClusterManager通信，进行资源的申请、任务的分配和监控等，负责作业执行的全生命周期管理
- ClusterManager提供了资源的分配和管理，
- ClusterManager 可以选择不同调度器组件完成整个集群的资源调度：YARN、Spark Standalone、Mesos
- 不同方式下， ClusterManager由对应的角色完成：
  - Standalone模式下由Master提供
  - Yarn模式下由RM担任



Yarn-Cluster运行模式（简化）

1. Client向Yarn提交作业，提交的配置中可以设置申请的资源情况，如果没有配置则将采用默认配置
2. RM接收到Client的作业请求后，首先检查程序启动的AM需要的资源情况，然后申请选取一个能够满足资源要求的NM节点用于启动AM进程，AM启动成功之后立即在该节点启动Driver进程
3. AM根据提交作业时设置的Executor相关配置参数或者默认配置参数与RM通信领取Executor资源信息
4. AM与相关NM通信启动Executor进程
5. Executor启动成功之后与Driver通信领取Driver分发的任务
6. Task执行，运行成功输出结果



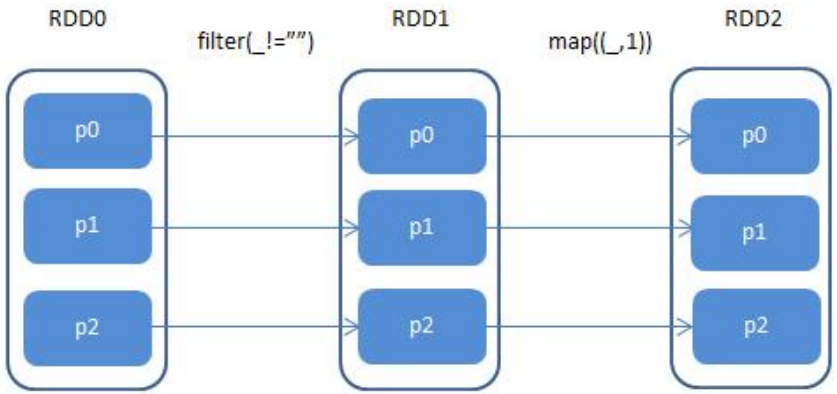
大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

RDD概念理解

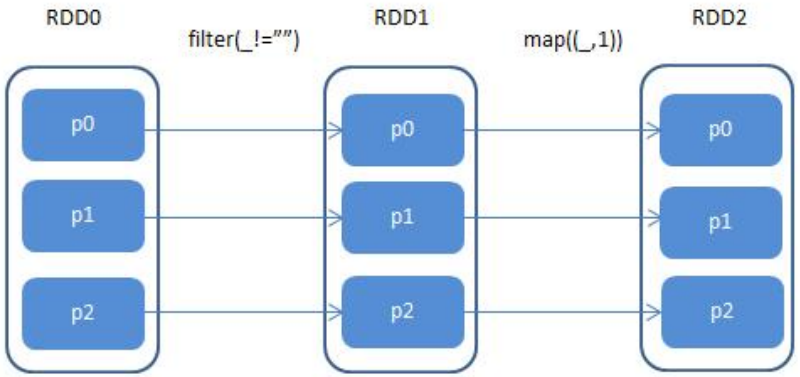
RDD基本概念

- 弹性分布数据集（Resilient Distributed Dataset）RDD
- RDD是Spark中对数据和计算的抽象，它表示已被分片（partition），不可变的，并能够被并行操作的数据集合
- 对RDD的操作分为两种transformation和action
- Transformation操作是通过转换从一个或多个RDD生成新的RDD
- Action操作是从RDD生成最后的计算结果
- 在Spark最新的版本中，提供丰富的transformation和action操作，比起MapReduce计算模型中仅有的两种操作，会大大简化程序开发的难度



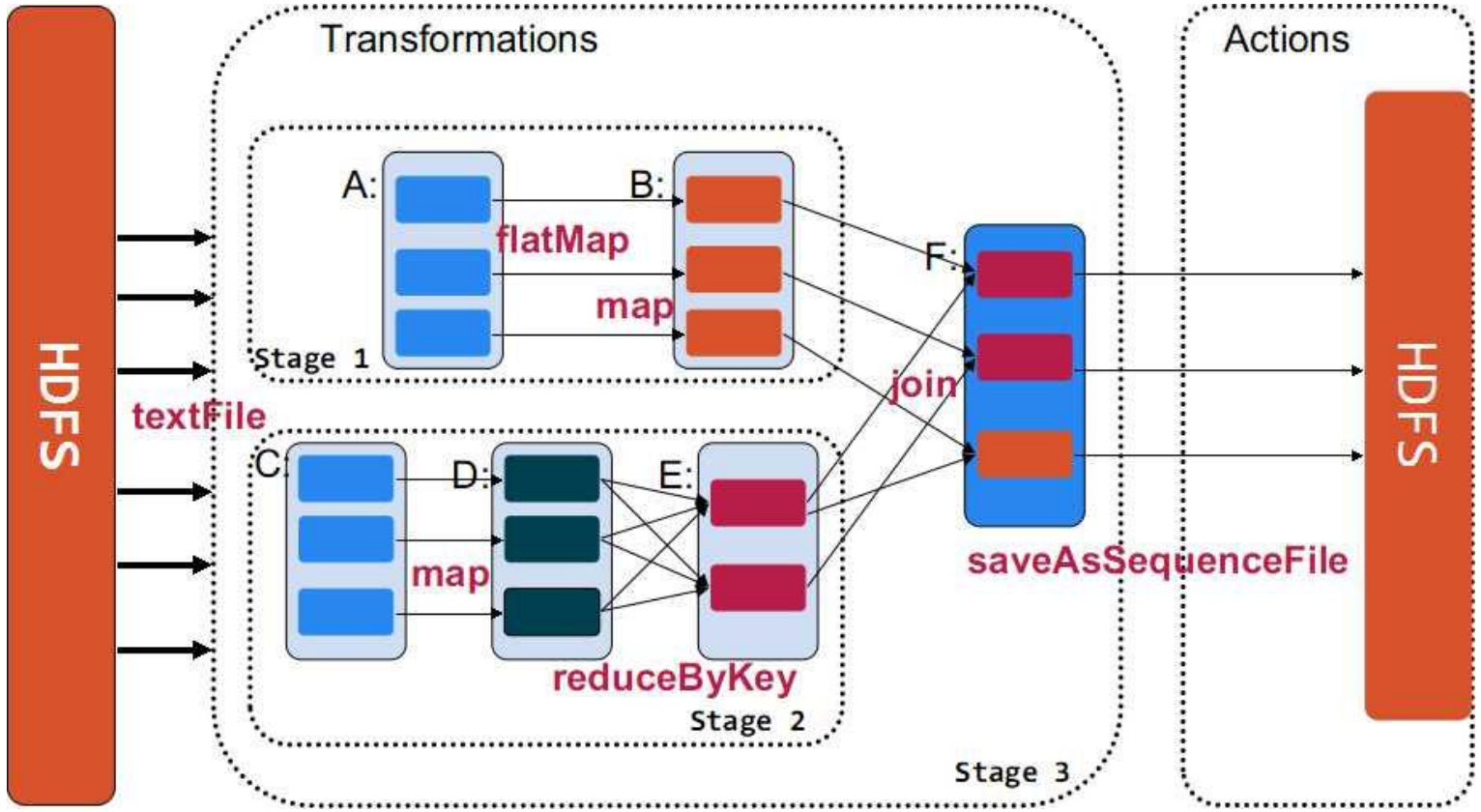
RDD的转换与操作

- RDD两种计算方式：转换Transformations （返回值还是一个RDD）与操作Actions （返回值不是一个RDD）
- 转换(Transformations)：如：map, filter, groupBy, join等
  - ✓Transformations操作是Lazy的
  - ✓从一个RDD转换生成另一个RDD的操作不是马上执行，Spark在遇到Transformations操作时只会记录需要这样的操作，并不会去执行，需要等到有Actions操作的时候才会真正启动计算过程进行计算
- 操作(Actions)：如：count, collect, save等
  - ✓Actions返回结果或把RDD数据写到存储系统中。Actions是触发Spark启动计算的动因



Spark框架的RDD、Transformations、Actions

- 从输入中逻辑上生成A和C两个RDD，经过一系列transformation操作，逻辑上生成了F
- 逻辑上生成F，是因为此时计算没有发生
- Spark内核做的事情只是记录了RDD的生成和依赖关系
- 当F要进行输出时，也就是F进行了action操作，Spark会根据RDD的依赖生成DAG，并从起点开始真正的计算



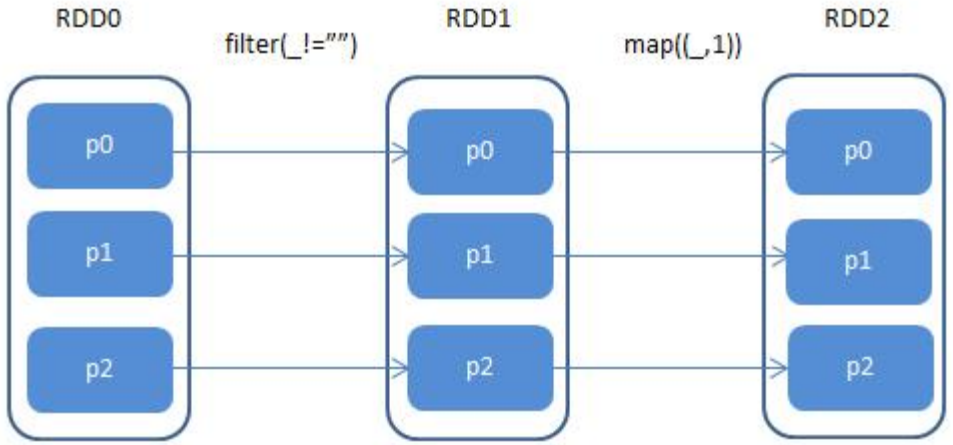
RDD的生成方式

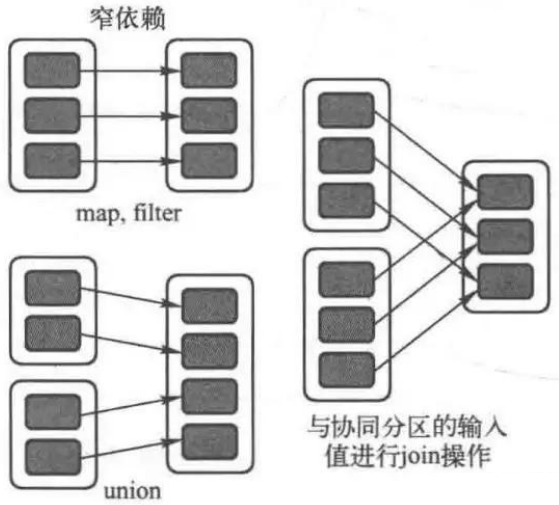
- RDD的生成方式只有两种：
  - ✓一是从数据源读入，另一种就是从其它RDD通过transformation操作转换
- 一个典型的Spark程序
  - ✓通过Spark上下文环境（SparkContext）生成一个或多个RDD
  - ✓在这些RDD上通过一系列的transformation操作生成最终的RDD
  - ✓最后通过调用最终RDD的action方法输出结果

RDD依赖关系-窄依赖

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- **窄依赖（Narrow Dependency）：**
- 表示每一个父RDD的1个Partition最多被子RDD的1个Partition所使用
- 如果子RDD只有部分分区数据损坏或者丢失，只需要从对应的父RDD重新计算恢复
- 窄依赖可以分为两类，一对一的依赖关系和范围依赖关系，如果子RDD最多依赖于1个父RDD，那么就是一对一的依赖关系。如果子RDD依赖于多个父RDD，那么就是范围依赖关系。

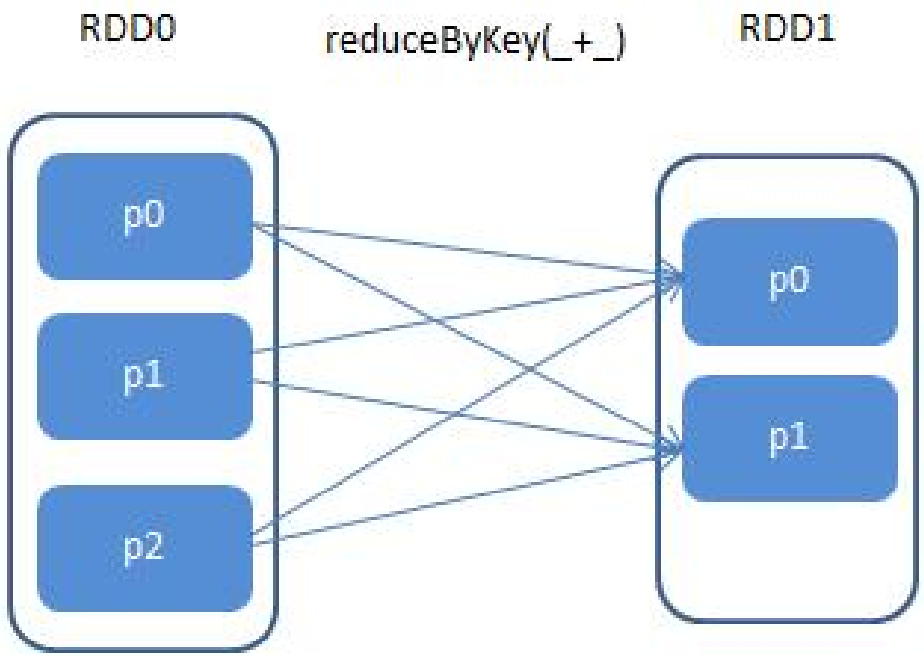




区分宽窄依赖主要就是看父RDD的一个Partition的流向，要是流向一个的话就是窄依赖，流向多个的话就是宽依赖

RDD依赖关系-宽依赖

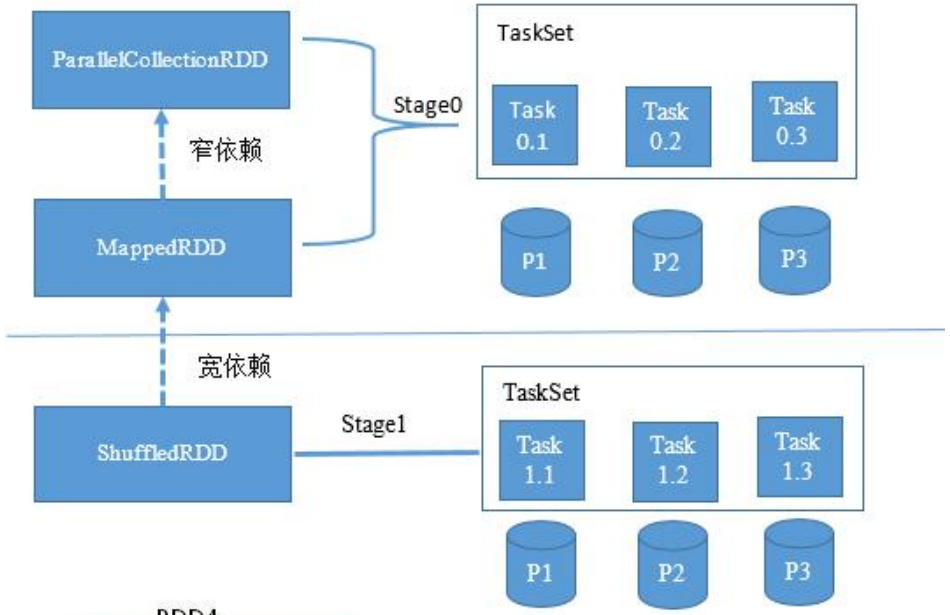
- **宽依赖（Shuffle Dependency）：**
- 一个父RDD的Partition会被多个子RDD的Partition使用
- 如果子RDD部分分区甚至全部分区数据损坏或丢失，需要从所有父RDD重新计算，相对窄依赖而言付出的代价更高

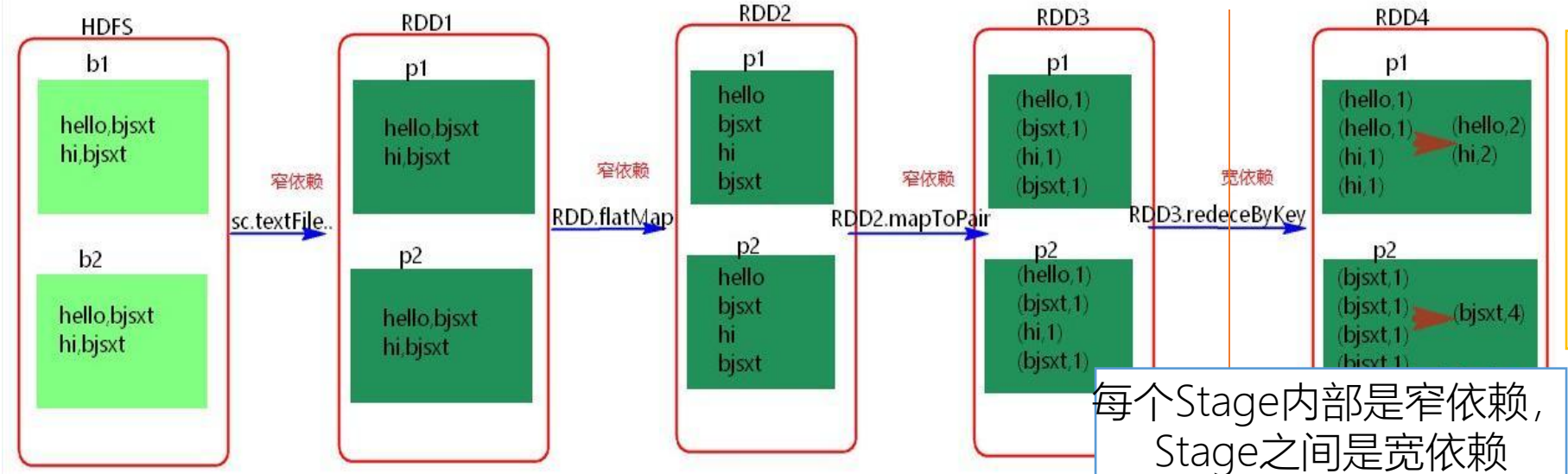


区分宽窄依赖主要就是看父RDD的一个Partition的流向，要是流向一个的话就是窄依赖，流向多个的话就是宽依赖

窄依赖 vs 宽依赖（Shuffle Dependency）

- Spark会将窄依赖的算子划分到一个Stage，同一个Stage里的上下游算子以Pipeline的方式连续的处理数据，减少数据的传输，提高数据的本地性、计算的连续性。
- 宽依赖的数据可能跨节点传输，而且需要父RDD数据可用并通过Shuffle动作后才能执行：即下游Task依赖上游Task计算结果，需上游Task执行完后，才能执行下游Task





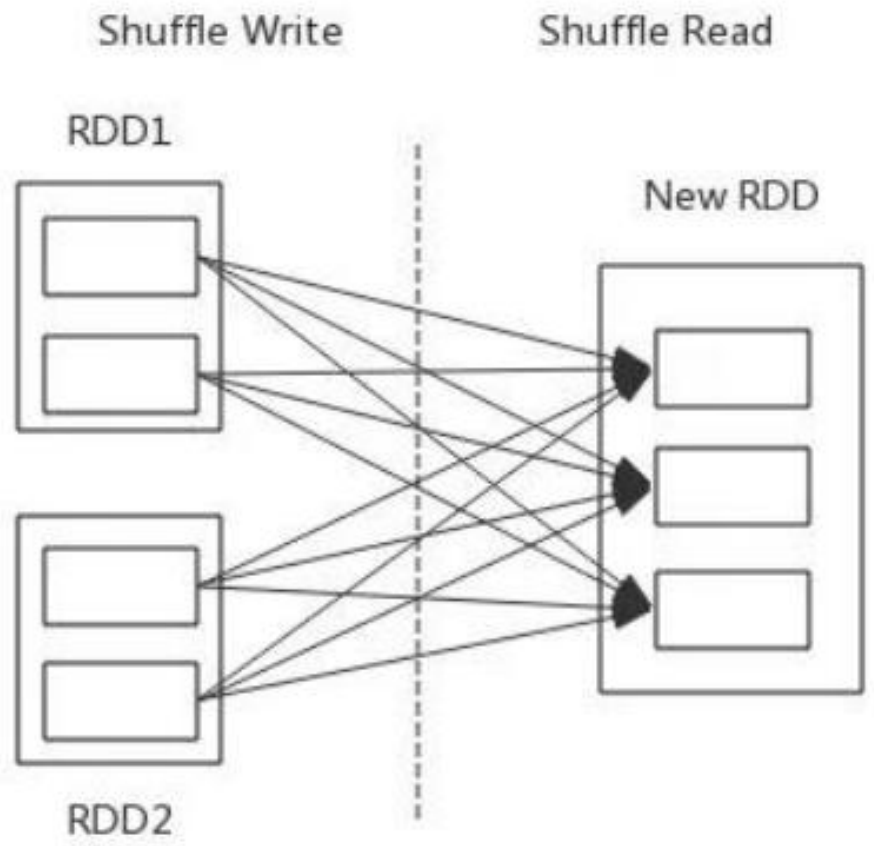
每个Stage内部是窄依赖，Stage之间是宽依赖

- ✓ 遇到宽依赖就划分stage
- ✓ 每个stage包含一个或多个task任务
- ✓ 将这些task以taskSet的形式提交给TaskScheduler运行
- ✓ stage是由一组并行task组成

Shuffle详解

- Shuffle最早出现于MapReduce框架中，负责连接Map阶段的输出与Reduce阶段的输入
- Shuffle就是将不同节点上相同的Key拉取到一个节点的过程
- **Shuffle阶段涉及磁盘IO、网络传输、内存使用等多种资源的调用，所以Shuffle阶段的执行效率影响整个作业的执行效率，大部分优化也都是针对Shuffle阶段进行的**
- Spark是实现了MapReduce原语的一种通用实时计算框架。Spark作业中Map阶段的Shuffle称为Shuffle Write，Reduce阶段的Shuffle称为Shuffle Read
- Shuffle Write阶段会将Map Task中间结果数据写入到本地磁盘，而在Shuffle Read阶段中，Reduce Task从Shuffle Write阶段拉取数据到内存中并行计算

Spark Shuffle阶段的划分方式

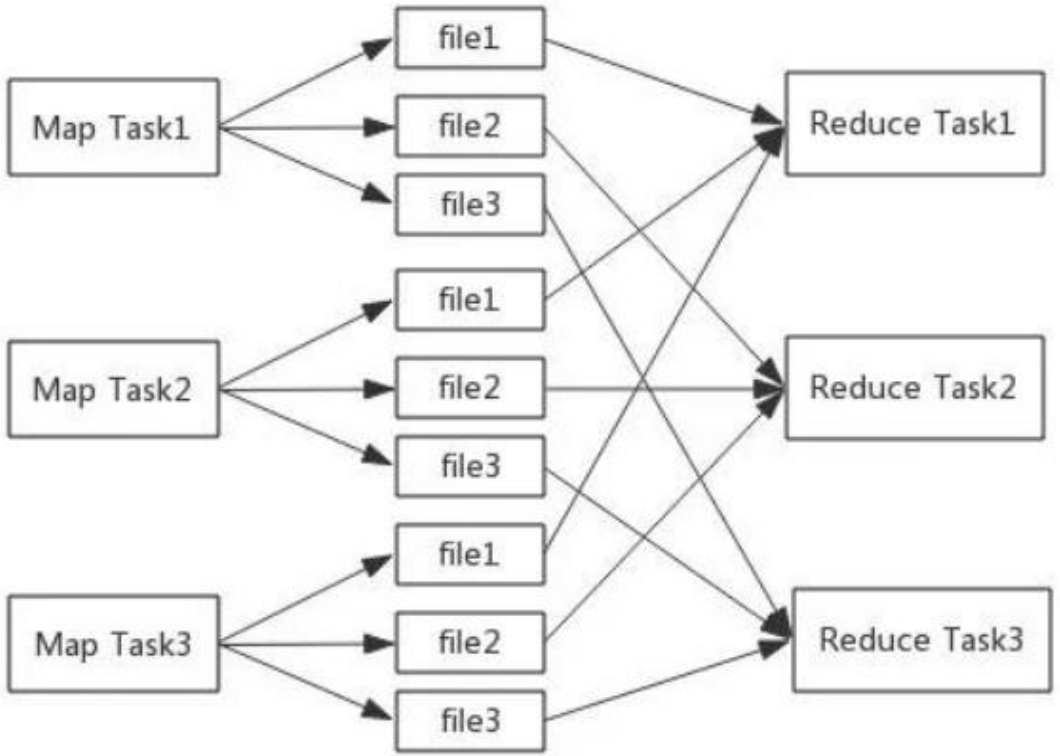


在Spark的中负责shuffle过程的执行、计算和处理的组件是ShuffleManager（shuffle管理器）

Shuffle Write实现-基于Hash的实现

- 每个Map Task都会生成与Reduce Task数据相同的文件数，对Key取Hash值分别写入对应的文件中
- 生成的文件数FileNum=MapTaskNum×ReduceTaskNum，
- 如果Map Task和Reduce Task数都比较多就会生成大量的小文件，写文件过程中，每个文件都要占用一部分缓冲区，总占用缓冲区大小TotalBufferSize=CoreNum×ReduceTaskNum×FileBufferSize，大量的小文件就会占用更多的缓冲区，造成不必要的内存开销，同时，大量的随机写操作会大大降低磁盘IO的性能

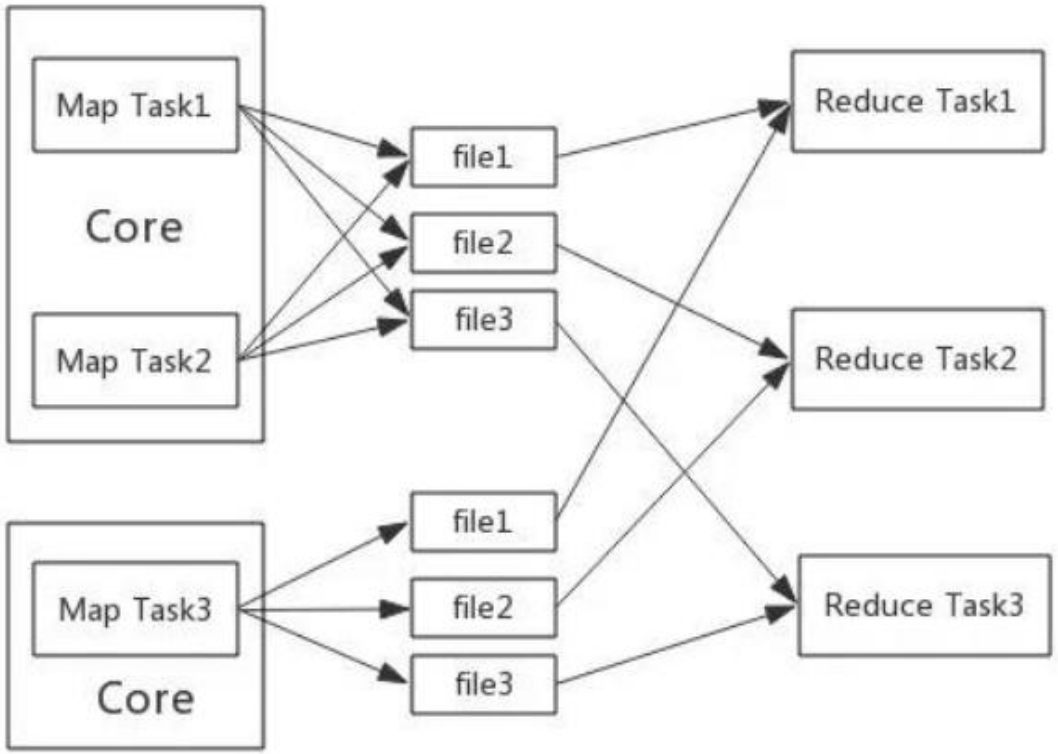
基于Hash的实现（hash-based）



Shuffle Write实现-基于Hash的实现

- 由于简单的基于Hash的实现方式扩展性较差，内存资源利用率低，过多的小文件在文件拉取过程中增加了磁盘IO和网络开销，所以需要对基于Hash的实现方式进行进一步优化，为此引入了Consolidate（合并）机制
- 将同一个Core中执行的Task输出结果写入到相同的文件中，生成的文件数FileNum=CoreNum×ReduceTaskNum，
- 这种优化方式减少了生成的文件数目，提高了磁盘IO的吞吐量，但是文件缓存占用的空间并没有减少，性能没有得到明显有效的提高

Consolidate（合并）机制



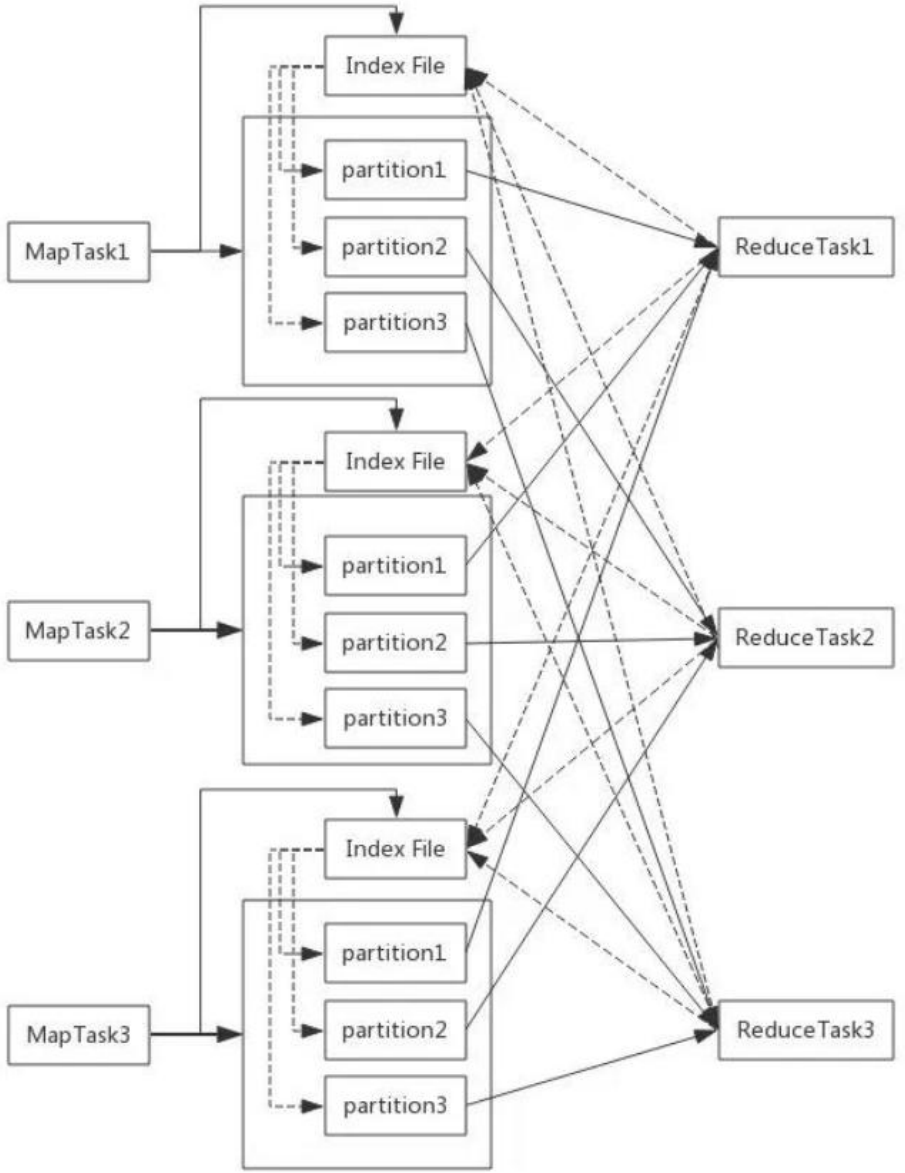
基于Hash的实现的设置方式及优缺点

- **设置方式：**
- 代码中设置：conf.get("spark.shuffle.manager", "hash")
- 配置文件中设置：在conf/spark-default.conf配置文件中添加spark.shuffle.managerhash

- **基于Hash的实现方式的优缺点：**
- 优点：实现简单，小数量级数据处理操作方便
- 缺点：产生小文件过多，内存利用率低，大量的随机读写造成磁盘IO性能下降

Shuffle Write实现-基于Sort的实现方式

- 为了解决基于Hash的实现方式的诸多问题，Spark Shuffle引入了基于Sort的实现方式
- 该方式中每个Map Task任务生成两个文件，一个是数据文件，一个是索引文件，生成的文件数**FileNum=MapTaskNum×2**
- 数据文件中的数据按照Key分区在不同分区之间排序，同一分区中的数据不排序，索引文件记录了文件中每个分区的偏移量和范围
- 当Reduce Task读取数据时，先读取索引文件找到对应的分区数据偏移量和范围，然后从数据文件读取指定的数据



基于Sort的实现方式的优缺点

- 设置方式：
- 代码中设置： conf.get("spark.shuffle.manager", "sort")
- 配置文件中设置： 在conf/spark-default.conf配置文件中添加 spark.shuffle.manager sort

- 优点：顺序读写能够大幅提高磁盘IO性能，不会产生过多小文件，降低文件缓存占用内存空间大小，提高内存使用率
- 缺点：多了一次粗粒度的排序

大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

RDD的转换和操作

RDD转换与操作介绍

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Transformation 算子	Meaning（含义）
map(*func*)	对原 RDD 中每个元素运用 *func* 函数，并生成新的 RDD
filter(*func*)	对原 RDD 中每个元素使用*func* 函数进行过滤，并生成新的 RDD
flatMap(*func*)	与 map 类似，但是每一个输入的 item 被映射成 0 个或多个输出的 items（ *func* 返回类型需要为 Seq ）。
mapPartitions(*func*)	与 map 类似，但函数单独在 RDD 的每个分区上运行， *func*函数的类型为 Iterator<T> => Iterator<U> ，其中 T 是 RDD 的类型，即 RDD[T]
mapPartitionsWithIndex(*func*)	与 mapPartitions 类似，但 *func* 类型为 (Int, Iterator<T>) => Iterator<U> ，其中第一个参数为分区索引
sample(*withReplacement*, *fraction*, *seed*)	数据采样，有三个可选参数：设置是否放回（withReplacement）、采样的百分比（*fraction*）、随机数生成器的种子（seed）；
union(*otherDataset*)	合并两个 RDD
intersection(*otherDataset*)	求两个 RDD 的交集
distinct([*numTasks*]))	去重

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Transformation 算子	Meaning（含义）
groupByKey([*numTasks*])	按照 key 值进行分区，即在一个 (K, V) 对的 dataset 上调用时，返回一个 (K, Iterable<V>) Note: 如果分组是为了在每一个 key 上执行聚合操作（例如，sum 或 average），此时使用 reduceByKey或 aggregateByKey 性能会更好 Note: 默认情况下，并行度取决于父 RDD 的分区数。可以传入 numTasks 参数进行修改。
reduceByKey(*func*, [*numTasks*])	按照 key 值进行分组，并对分组后的数据执行归约操作。
aggregateByKey(*zeroValue*,*numPartitions*)(*seqOp*, *combOp*, [*numTasks*])	当调用（K，V）对的数据集时，返回（K，U）对的数据集，其中使用给定的组合函数和 zeroValue 聚合每个键的值。与 groupByKey 类似，reduce 任务的数量可通过第二个参数进行配置。
sortByKey([*ascending*], [*numTasks*])	按照 key 进行排序，其中的 key 需要实现 Ordered 特质，即可比较
join(*otherDataset*, [*numTasks*])	在一个 (K, V) 和 (K, W) 类型的 dataset 上调用时，返回一个 (K, (V, W)) pairs 的 dataset，等价于内连接操作。如果想要执行外连接，可以使用 leftOuterJoin, rightOuterJoin 和 fullOuterJoin 等算子。

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

map()

- 对原 RDD 中每个元素运用 *func* 函数，并生成新的 RDD

```
val list = List(1,2,3)
sc.parallelize(list).map(_ * 10).foreach(println)
// 输出结果： 10 20 30
```

filter()

- 对原 RDD 中每个元素使用*func* 函数进行过滤，并生成新的RDD

```
val list = List(3, 6, 9, 10, 12, 21)
sc.parallelize(list).filter(_ >= 10).foreach(println)
// 输出：10 12 21
```

转换与操作说明

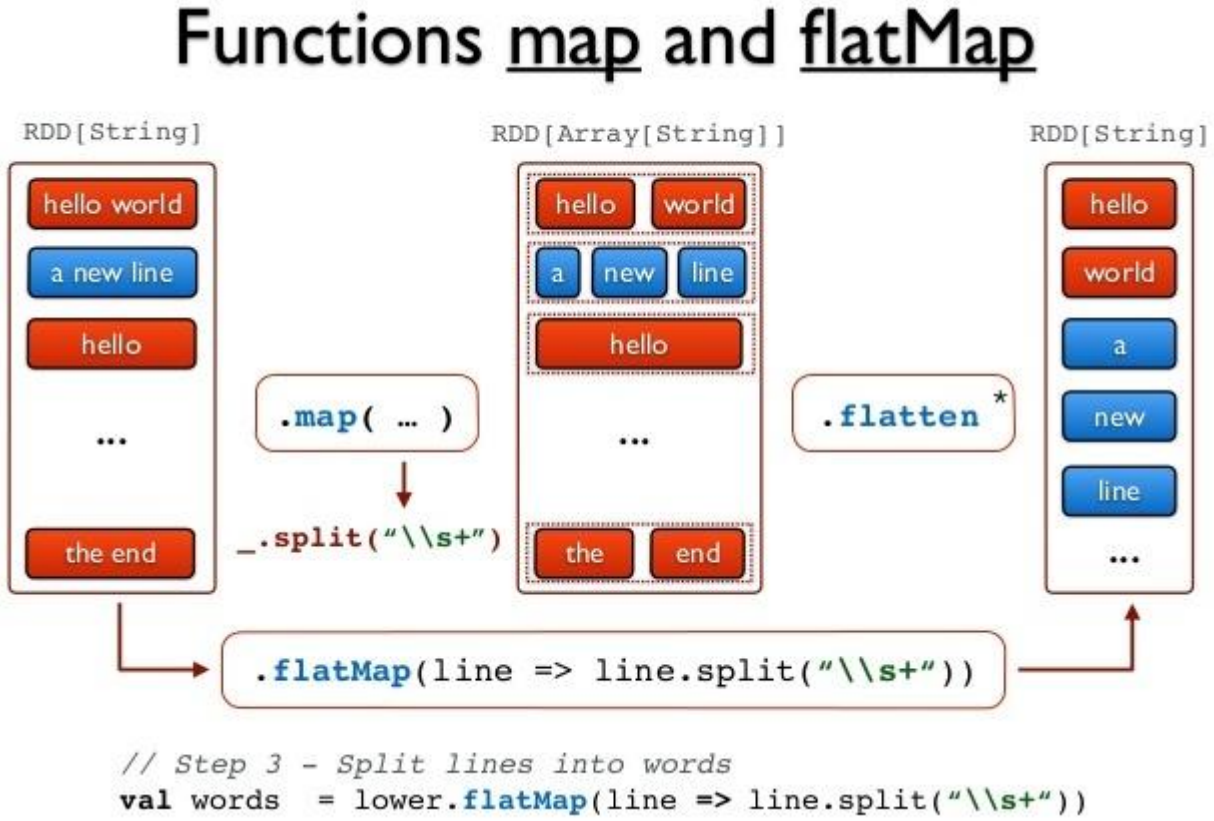
Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

flatMap()

- 与map类似，但是每一个输入的item被映射成0个或多个输出的items（*func*返回类型需要为Seq）

```
val list = List(List(1, 2),
List(3), List(), List(4, 5))
sc.parallelize(list).flatMap(_.to
List).map(_ * 10).foreach(println)
// 输出结果 ： 10 20 30 40 50
```



转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

mapPartitions()

- 与map类似,但函数单独在RDD 的每个分区上运行，*func*函数的类型为Iterator<T> => Iterator<U> ，其中T是RDD 的类型，即RDD[T]

mapPartitions对一个rdd里所有分区遍历

效率优于map算子，减少了发送到执行器执行的交互次数，mapPartitions是批量将分区数据一次发送

假设有N个元素，有M个分区，那么map的函数的将被调用N次,而mapPartitions被调用M次,一个函数一次处理所有分区

func的函数类型必须是Iterator[T] => Iterator[U]

map每次拿到的数据是集合中一个元素，mapPartitions每次拿到的是一个分区里的所有元素

```
val list = List(1, 2, 3, 4, 5, 6)
sc.parallelize(list, 3).mapPartitions(iterator => { val buffer
= new ListBuffer[Int] while (iterator.hasNext)
{ buffer.append(iterator.next() * 100) }
buffer.toIterator }).foreach(println)
//输出结果 100 200 300 400 500 600
```

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

mapPartitionsWithIndex(*func*)

- 与 mapPartitions 类似，但 *func* 类型为 (Int, Iterator<T>) => Iterator<U> ，其中第一个参数为分区索引

```
val list = List(1, 2, 3, 4, 5, 6) sc.parallelize(list,
3).mapPartitionsWithIndex((index, iterator) => { val buffer = new
ListBuffer[String] while (iterator.hasNext) { buffer.append(index + "分区:"
+ iterator.next() * 100) } buffer.toIterator }).foreach(println)
//输出 0 分区:100 0 分区:200 1 分区:300 1 分区:400 2 分区:500 2 分区:600
```

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

sample()

- 数据采样，有三个可选参数：设置是否放回（withReplacement）、采样的百分比（*fraction*）、随机数生成器的种子（seed）；

```
val list = List(1, 2, 3, 4, 5, 6)
sc.parallelize(list).sample(withReplacement = false, fraction =
0.5).foreach(println)
```

union()

- 合并两个RDD

```
val list1 = List(1, 2, 3)
val list2 = List(4, 5, 6)
sc.parallelize(list1).union(sc.parallelize(list2)).foreach(println)
// 输出: 1 2 3 4 5 6
```

转换与操作说明

Transformation

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

intersection()

- 求两个 RDD 的交集

```
val list1 = List(1, 2, 3, 4, 5)
val list2 = List(4, 5, 6)
sc.parallelize(list1).intersection(sc.parallelize(list2)).foreach(println)
// 输出: 4 5
```

distinct()

- 去重

```
val list = List(1, 2, 2, 4, 4)
sc.parallelize(list).distinct().foreach(println)
//输出:4 1 2
```

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

groupByKey()

- 按照 key 值进行分区，即在一个 (K, V) 对的dataset上调用时，返回一个 (K, Iterable<V>)
- Note: 如果分组是为了在每一个 key 上执行聚合操作（例如，sum 或 average)，此时使用reduceByKey或aggregateByKey性能会更好
- Note: 默认情况下，并行度取决于父RDD的分区数。可以传入 numTasks 参数进行修改

按照键进行分组：

```
val list = List(("hadoop", 2), ("spark", 3), ("spark", 5), ("storm", 6), ("hadoop", 2))
sc.parallelize(list).groupByKey().map(x => (x._1, x._2.toList)).foreach(println)
//输出：(spark,List(3, 5)) (hadoop,List(2, 2)) (storm,List(6))
```

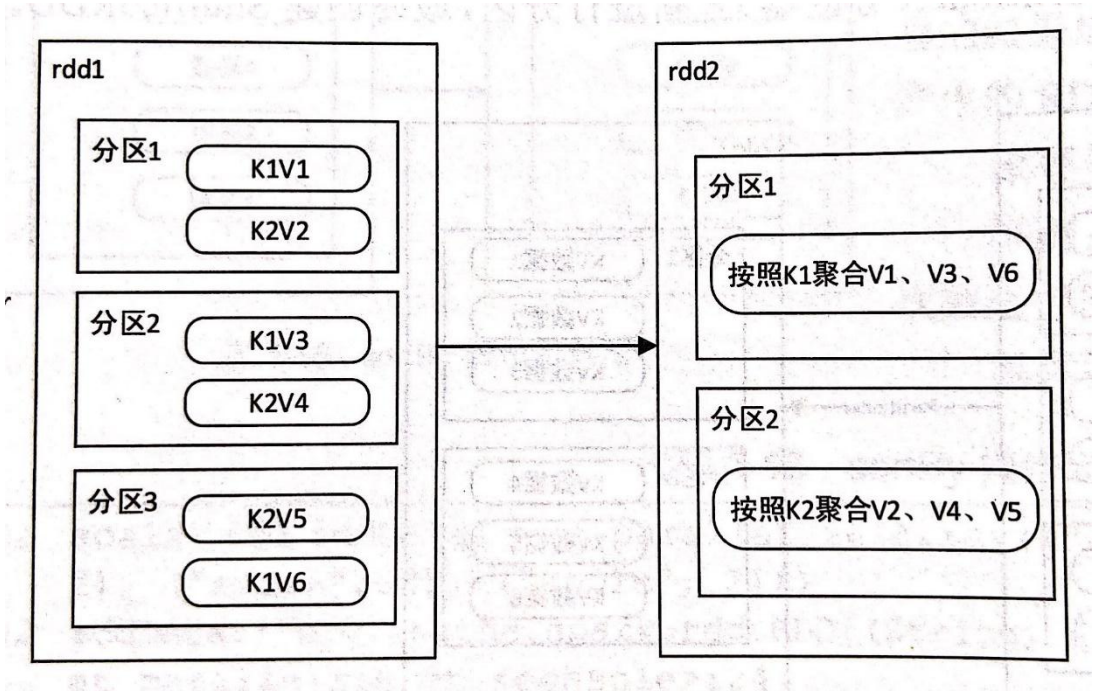
转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

reduceByKey()

- 按照 key 值进行分组，并对分组后的数据执行归约操作。



按照键进行归约操作：

```
val list = List(("hadoop", 2), ("spark",
3), ("spark", 5), ("storm", 6), ("hadoop",
2)) sc.parallelize(list).reduceByKey(_ +
_).foreach(println)
//输出 (spark,8) (hadoop,4) (storm,6)
```

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

sortByKey()

- 按照 key 进行排序，其中的 key 需要实现 Ordered 特质，即可比较。

按照键进行排序：

```
sc.parallelize(Array(("a", 1), ("c", 3), ("b", 2), ("d",
4))).sortBy(_._2)
```

输出

```
Array[(String, Int)] = Array((a,1), (b,2), (c,3), (d,4)
```

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

join()

- 在一个 (K, V) 和 (K, W) 类型的 dataset 上调用时，返回一个 (K, (V, W)) pairs 的 dataset，等价于内连接操作；如果想要执行外连接可以使用 leftOuterJoin, rightOuterJoin 和 fullOuterJoin 等算子。

```
val list01 = List((1, "student01"), (2, "student02"), (3, "student03")) val list02 = List((1, "teacher01"), (2, "teacher02"), (3, "teacher03"))
sc.parallelize(list01).join(sc.parallelize(list02)).foreach(println)
// 输出
(1,(student01,teacher01))
(3,(student03,teacher03))
(2,(student02,teacher02))
```

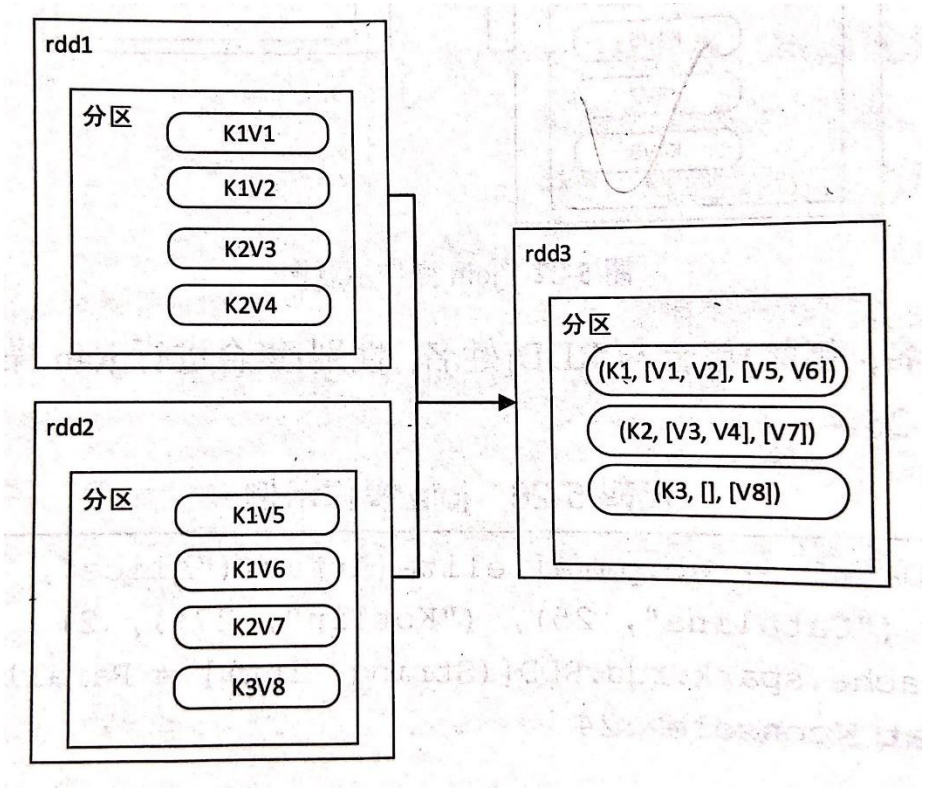
转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

cogroup()

- 在一个 (K, V) 对的 dataset 上调用时，返回一个 (K, (Iterable<V>, Iterable<W>)) tuples 的 dataset。




```
val list01 = List((1, "a"),(1, "a"), (2, "b"),
(3, "e")) val list02 = List((1, "A"), (2, "B"),
(3, "E")) val list03 = List((1, "[ab]"), (2,
"[bB]"), (3, "eE"),(3, "eE"))
sc.parallelize(list01).cogroup(sc.parallelize(l
ist02),sc.parallelize(list03)).foreach(println)
// 输出：同一个RDD中的元素先按照key进行分组，然后
再对不同RDD中的元素按照key进行分组
(1,(CompactBuffer(a,
a),CompactBuffer(A),CompactBuffer([ab])))
(3,(CompactBuffer(e),CompactBuffer(E),CompactBu
ffer(eE, eE)))
(2,(CompactBuffer(b),CompactBuffer(B),CompactBu
ffer([bB])))
```

转换与操作说明

Transformation

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

cartesian()

- 在一个*T*和*U*类型的*dataset*上调用时，返回一个(*T*, *U*)类型的*dataset*（即笛卡尔积）。

```
val list1 = List("A", "B", "C") val list2 = List(1, 2, 3)
sc.parallelize(list1).cartesian(sc.parallelize(list2)).foreach(println)
//输出笛卡尔积
(A,1)
(A,2)
(A,3)
(B,1)
(B,2)
(B,3)
(C,1)
(C,2)
(C,3)
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Action（动作）	Meaning（含义）
reduce(*func*)	使用函数*func*执行归约操作
collect()	以一个 array 数组的形式返回 dataset 的所有元素，适用于小结果集。
count()	返回 dataset 中元素的个数。
first()	返回 dataset 中的第一个元素，等价于 take(1)。
take(*n*)	将数据集中的前 *n* 个元素作为一个 array 数组返回。
takeSample(*withReplacement*, *num*, [*seed*])	对一个 dataset 进行随机抽样
takeOrdered(*n*, *[ordering]*)	按自然顺序（natural order）或自定义比较器（custom comparator）排序后返回前 *n* 个元素。只适用于小结果集，因为所有数据都会被加载到驱动程序的内存中进行排序。
saveAsTextFile(*path*)	将 dataset 中的元素以文本文件的形式写入本地文件系统、HDFS 或其它 Hadoop 支持的文件系统中。Spark 将对每个元素调用 toString 方法，将元素转换为文本文件中的一行记录。
saveAsSequenceFile(*path*)	将 dataset 中的元素以 Hadoop SequenceFile 的形式写入到本地文件系统、HDFS 或其它 Hadoop 支持的文件系统中。该操作要求 RDD 中的元素需要实现 Hadoop 的 Writable 接口。对于 Scala 语言而言，它可以将 Spark 中的基本数据类型自动隐式转换为对应 Writable 类型。(目前仅支持 Java and Scala)
saveAsObjectFile(*path*)	使用 Java 序列化后存储，可以使用 SparkContext.objectFile() 进行加载。(目前仅支持 Java and Scala)
countByKey()	计算每个键出现的次数。
foreach(*func*)	遍历 RDD 中每个元素，并对其执行*fun*函数

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

reduce()

- 最常用的是reduce()，接收一个函数，作用在RDD的两个类型相同的元素上，返回一个类型相同的新元素
- 使用reduce()可以很最常用的加法函数，RDD中元素的累加，计数，和其它类型的聚集操作
- reduce()例子-加法：

val sum = rdd.reduce((x, y) => x + y)

```
val list = List(1, 2, 3, 4, 5)
sc.parallelize(list).reduce((x, y) => x + y)
sc.parallelize(list).reduce(_ + _) // 输出 15
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

count

- 统计RDD中元素个数的算子

```
valrdd = sc.parallelize(

  List("hello","world!","hi","beijing"))

println(rdd.count())

输出:

4
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

collect

- 把RDD中的元素以数组形式提取到Driver内存。即将RDD转化为数组。

```
val rdd = sc.parallelize(
    List("hello","world!","hi","beijing"),2)

val arr: Array[String] = rdd.collect()
arr.foreach(println)

输出:
hello
world!
hi
beijing
```

foreach

- 遍历RDD中的每一个元素，并依次应用f函数，无返回值。即计算RDD中每个元素，但不返回到本地
- 可以配合println() 友好的打印出数据；如：foreach(println)　作用：把函数println当作参数传递给函数foreach

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

collectAsMap

- 作用于K-V类型的RDD上，作用与collect不同的是collectAsMap函数不包含重复的key，对于重复的key，后面的元素覆盖前面的元素

```
val rdd = sc.parallelize(List(("a",1),("b",2),("c",3),("c",4)),2)
val res: Map[String, Int] = rdd.collectAsMap()
res.foreach(println)

输出:
(b,2)
(a,1)
(c,4)
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

saveAsTextFile

- 把RDD中的数据以文本的形式保存。

```
valrdd = sc.parallelize(List(5,4,7,1,9),3)

rdd.saveAsTextFile("/home/myname/test")
```

saveAsSequenceFile

- 是个k-v算子，把RDD中的数据以序列化的形式保存。使用此算子的前提是RDD中元素是键值对格式。

```
val rdd = sc.parallelize(
    List(("a",1),("b",2),("c",3),("c",4)),2)

rdd.saveAsSequenceFile("/home/myname/test")
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

take

- 从RDD中取下标前n个元素，不排序。返回数组。

```
valrdd = sc.parallelize(List(5,4,7,1,9),3)
val take: Array[Int] = rdd.take(2)
take.foreach(println)

输出:
5
4
```

takeSample

- 从指定RDD中抽取样本。第一个参数为false表示取过的元素不再取，为true表示取过的元素可以再次被抽样；第二个参数表示取样数量。

```
val rdd = sc.makeRDD(Array("aaa","bbb","ccc","ddd","eee"))
val sample: Array[String] = rdd.takeSample(false,2)
sample.foreach(println)
输出:
eee
bbb
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

first

- 返回RDD中第一个元素。

```
valrdd = sc.parallelize(List(5,4,7,1,9),3)
val first: Int = rdd.first()
println(first)
输出:
5
```

top

- 从RDD中按默认顺序(降序)或指定顺序取n个元素。

```
valrdd = sc.parallelize(List(5,4,7,1,9),3)
val take: Array[Int] = rdd.top(2)
take.foreach(println)
输出:
9
7
```

转换与操作说明

Actions

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

takeOrdered

- 从RDD中取n个元素，与top算子不同的是它是以和top相反的顺序返回元素。

```
valrdd = sc.parallelize(List(5,4,7,1,9),3)
val take: Array[Int] = rdd.takeOrdered(2)
take.foreach(println)
输出:
1
4
```

lookup

- 是个k-v算子，指定key值，返回此key对应的所有v值。

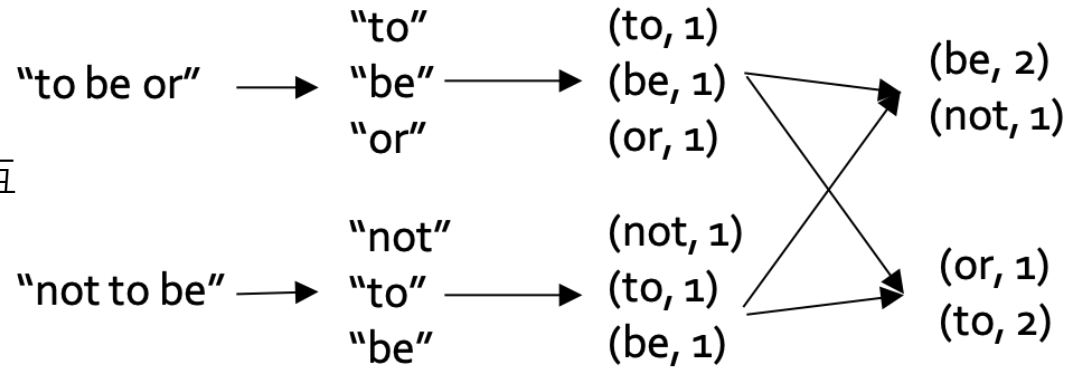
```
valrdd1 = sc.makeRDD(Array(("A",0),("A",2), ("B",1),("B",2),("C",1)))
val rdd2: Seq[Int] = rdd1.lookup("A")
rdd2.foreach(println)
输出:
0
2
```

转换与操作说明

Word Count示例

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

```
import org.apache.spark.{SparkConf, SparkContext}
object Chapter3_3_2 {
   def main (args:Array[String]) : Unit = {
       //初始化SparkConf对象，设置基本任务参数
      val conf = new SparkConf ()
     //设置目标Master通信地址
     .setMaster("Spark://linux01:7077")
     //设置任务参数
     .setAppName("WC")
    //实例化SparkContext, spakr的对外接口负责用户与Spark内部的交互
通信
   val sc = new SparkContext(conf)
   //读取文件并进行单词统计
   sc.textfile("hdfs://linuex01:8020/words.txt")
    .flatMap(._split(" "))
    .map((_,1))
    .reduceByKey(_ + _, 1 )
    .sortBy(_,_2, false)
    .saveAsTextFile("hdfs://linux01:8020/words/output")
    //停止sc,结束该任务
    sc.stop()   }
}
```



转换与操作说明

Word Count示例

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

```
val sc = new SparkContext(conf)
  //读取文件并进行单词统计
  sc.textfile("hdfs://linuex01:8020/words.txt")
   .flatMap(._split(" "))
   .map((_,1))
   .reduceByKey(_ + _, 1 )
   .sortBy(_,_2, false)
}
```

该行将HDFS中的文本内容读取到RDD中

对于开发者而言，RDD屏蔽了很多细节，让人感觉在使用本地数据集合一样

本行代码执行完毕后会得到一个RDD[String]实例，泛型为String，即RDD中每一个元素都对应着words.txt文本中的一行数据。比如在该示例中，RDD[String]实例的具体形态为：

Words.txt内容：

```
dog cat hadoop spark
dog hadoop kafka kylin
cat kafka spark
```

Res0: RDD[String] = RDD("dog cat hadoop spark"," dog hadoop kafka kylin", "cat kafka spark"

转换与操作说明

Word Count示例

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

```
val sc = new SparkContext(conf)
   //读取文件并进行单词统计
    sc.textfile("hdfs://linuex01:8020/words.txt")
     .flatMap(._split(" "))
     .map((_,1))
     .reduceByKey(_ + _, 1 )
     .sortBy(_,_2, false)
}
```

该行中，flatMap为RDD转换操作，之后将会对RDD的各类操作进行详细的说明，本行代码执行完毕后，原本RDD[String]类型的数据将会被转换为：

Res0: RDD[String] = RDD(“dog”, ”cat”, “Hadoop”, “spark”……

Words.txt内容：

dog cat hadoop spark
dog hadoop kafka kylin
cat kafka spark

转换与操作代码示例

Word Count示例

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

```
val sc = new SparkContext(conf)
   //读取文件并进行单词统计
    sc.textfile("hdfs://linuex01:8020/words.txt")
     .flatMap(._split(" "))
     .map((_,1))
     .reduceByKey(_ + _, 1 )
     .sortBy(_,_2, false)
}
```

该行进行map操作。将每个单词映射为”（word,1）”这样的二元组结构，所以得到的结果应该是：

Words.txt内容：

```
dog cat hadoop spark
dog hadoop kafka kylin
cat kafka spark
```

Res0: RDD[(String,Int)] =
RDD(Tuple2(“dog”,1), Tuple2(”cat”,1),
Tuple2(“hadoop”, 1), Tuple2(“spark”,1)....

转换与操作代码示例

Word Count示例

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

```
val sc = new SparkContext(conf)
  //读取文件并进行单词统计
   sc.textfile("hdfs://linuex01:8020/words.txt")
    .flatMap(._split(" "))
    .map((_,1))
    .reduceByKey(_ + _, 1 )
    .sortBy(_,_2, false)
    .saveAsTextFile("hdfs://linux01:8020/words/output")
}
```

首先进行reduceByKey操作，reduceByKey会把所有相同但此对应的数目"1"累加在一起，在执行操作前，可以指定分区个数，这里指定为1个分区，即spark会将结果输出到一个文件中

Words.txt内容：

```
dog cat hadoop spark
dog hadoop kafka kylin
cat kafka spark
```

接着次行将词频结果按照降序排列
前两行执行完毕后，得到的结果为:
Res5:RDD[(String,Int)] =
RDD(("Spark",2),("hadoop",2),("dog",2)...))

最后一行 saveAsTextFile为RDD的Action操作。该操作会触发执行整个任务计划，并且将最后的执行结果输出到HDFS的对应目录中。

大数据处理—Spark

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Scala语言

Scala介绍

代码

```
 public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
```

这段代码为Map端的核心，定义了Map Task 所需要执行的任务的具体逻辑实现。
map() 方法的参数为 Object key, Text value, Context context，其中：

Scala介绍

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

变量的声明

Java	Scala
变量：	变量：使用var关键字
private String name;	var name:String="jack"
private int age;	var age：Int=22
常量：	常量：使用val关键字
private final static long Id=00000000008L;	val name:String="jack"
	val age:Int=22

另外,scala可以在没有变量类型的情况下，会根据值自动生成相关类型，
比如：var name=“jack”,则name就为String类型　　var age =22, age 位Int类型
小结：Java变量声明中类型置前，如：int age,Scala是类型置后，如：var age:Int。

Scala介绍

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

函数的声明

Java

```
//Java的实现
public class TestJava{
public static void main(String [] args){
        addInt(6,8);
}

public static int addInt(int a,int b){
        return a+b;
}

public void printInfo(String info){
        System.out.println("无返回值的函数，
打印信息："+info);
}
}
```

Scala

```
object Test{
def main(args:Array[String]){
    addInt(6,8);
}

def addInt(a:Int,b:Int):Int={
     return a+b;
}

//Scala函数没有返回值用Unit
def printInfo(info:String):Unit={

     println("无返回值的函数，打印信息："+info)
}

}
```

Scala介绍

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

数组的输出

Java

```
public class ArrayDemo {
        public static void main(String args[]) {
                int data[] = new int[3];
                data[0] = 10; // 第一个元素
                data[1] = 20; // 第二个元素
                data[2] = 30; // 第三个元素
                for(int x = 0; x < data.length; x++) {
                        System.out.println(data[x]);
                }
        }
}
```

Scala

```
Import Array._
Object Test
Def main(args:Array[String]{
Var myList1 = Array(1.9,2.9,3.4,3.5)
Var myList2 = Array(8.9,7.9,0.4,15)
Var myList3 = concat(myList1,myList2)
For(x<- myList3){
  println(x)
}
```

Scala学习资料

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- ✓ Scala 与Java的关系
- ✓ Scala 代码是编译成Java字节码，在任何标准JVM上运行
- ✓ Scala编译器是Java编译器的作者写的
- ✓ 与Java的互操作性：可重用Java库、可重用Java工具

- ✓ Scala 的基本语法学习

  https://www.w3cschool.cn/scala/scala-index.html

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

谢谢聆听
批评指正

ehaihong@bupt.edu.cn