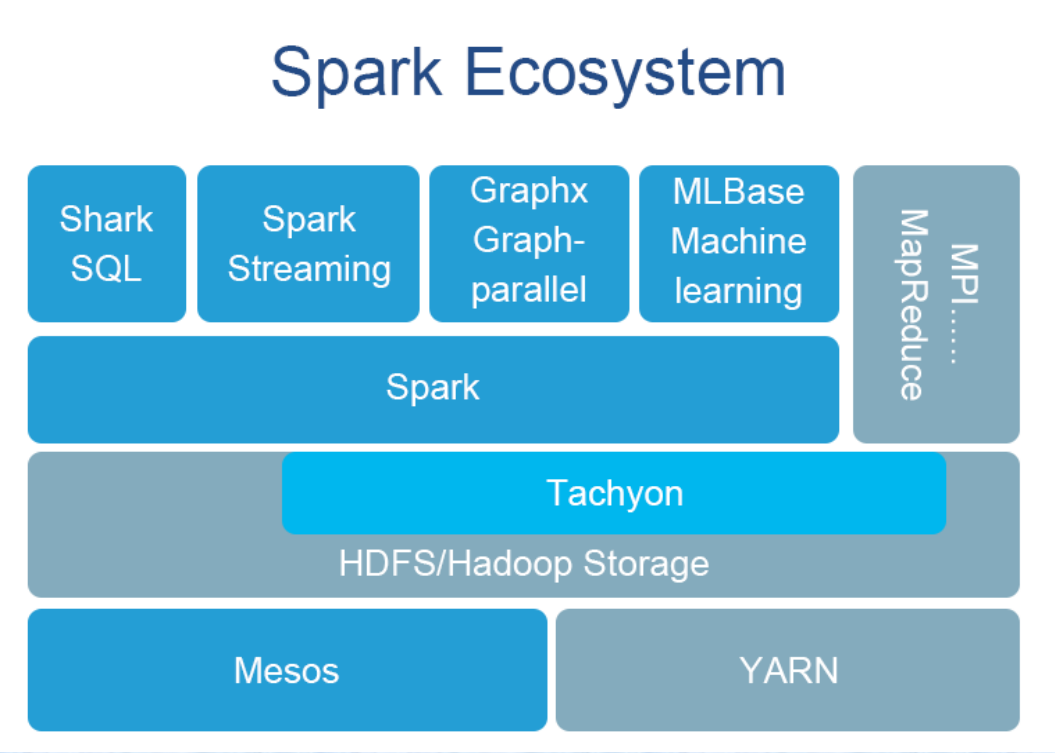
Lecture 19 Spark内存计算

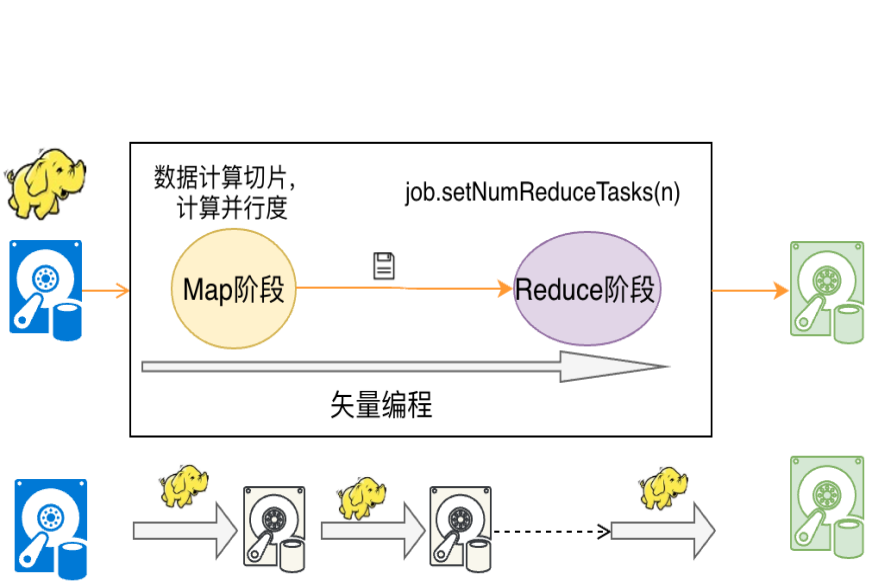
- 逻辑计算模型
- 物理计算架构

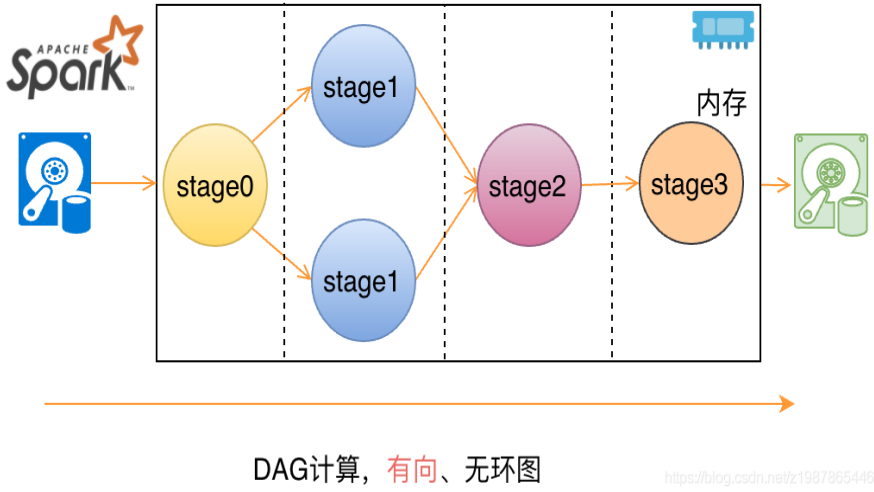
1. 逻辑计算模型

Spark 计算生态系统：构建在Hadoop平台上，利用HDFS存储系统架构，使用Mesos或YARN作为集群资源管理系统。

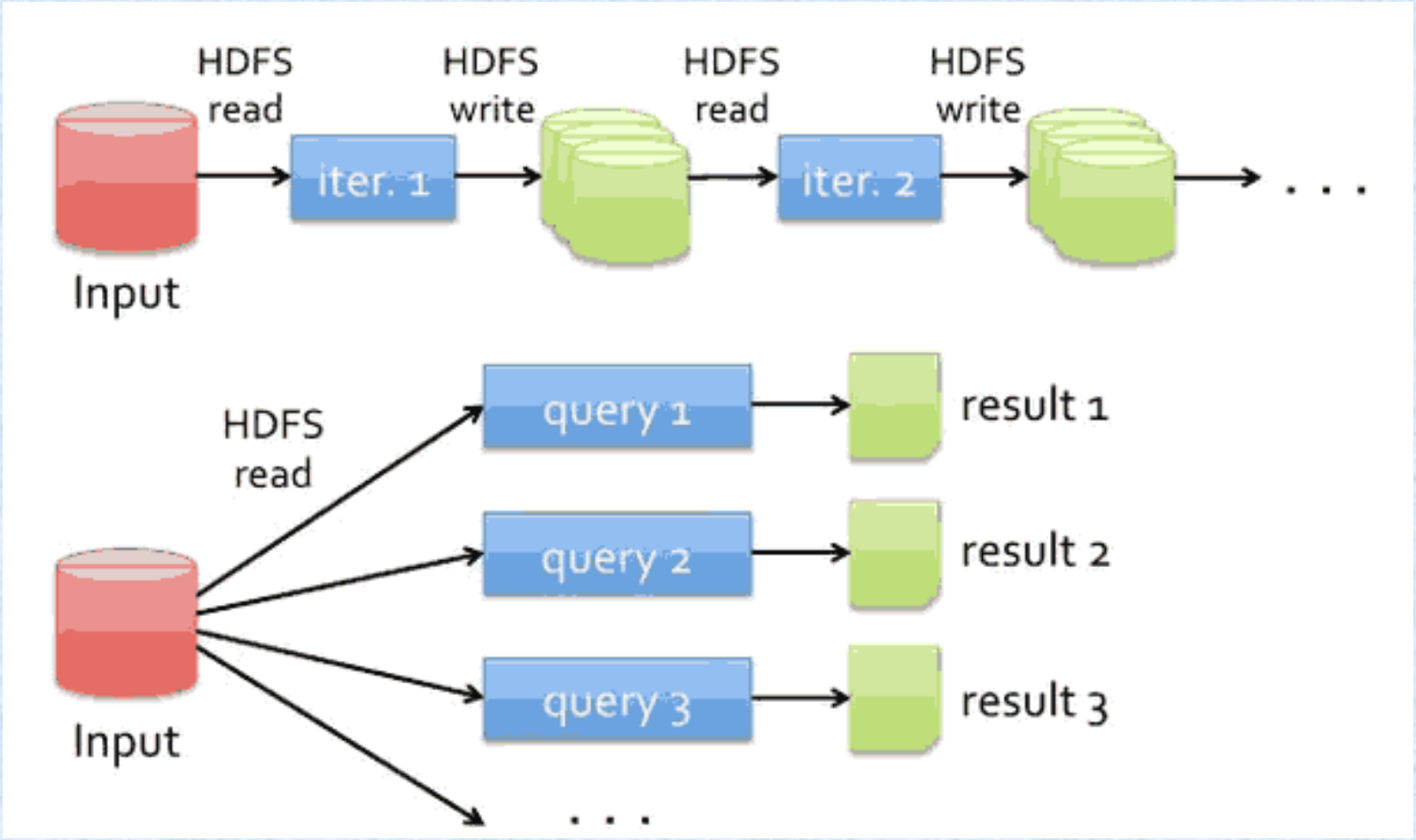


MapReduce vs. Spark In-mem

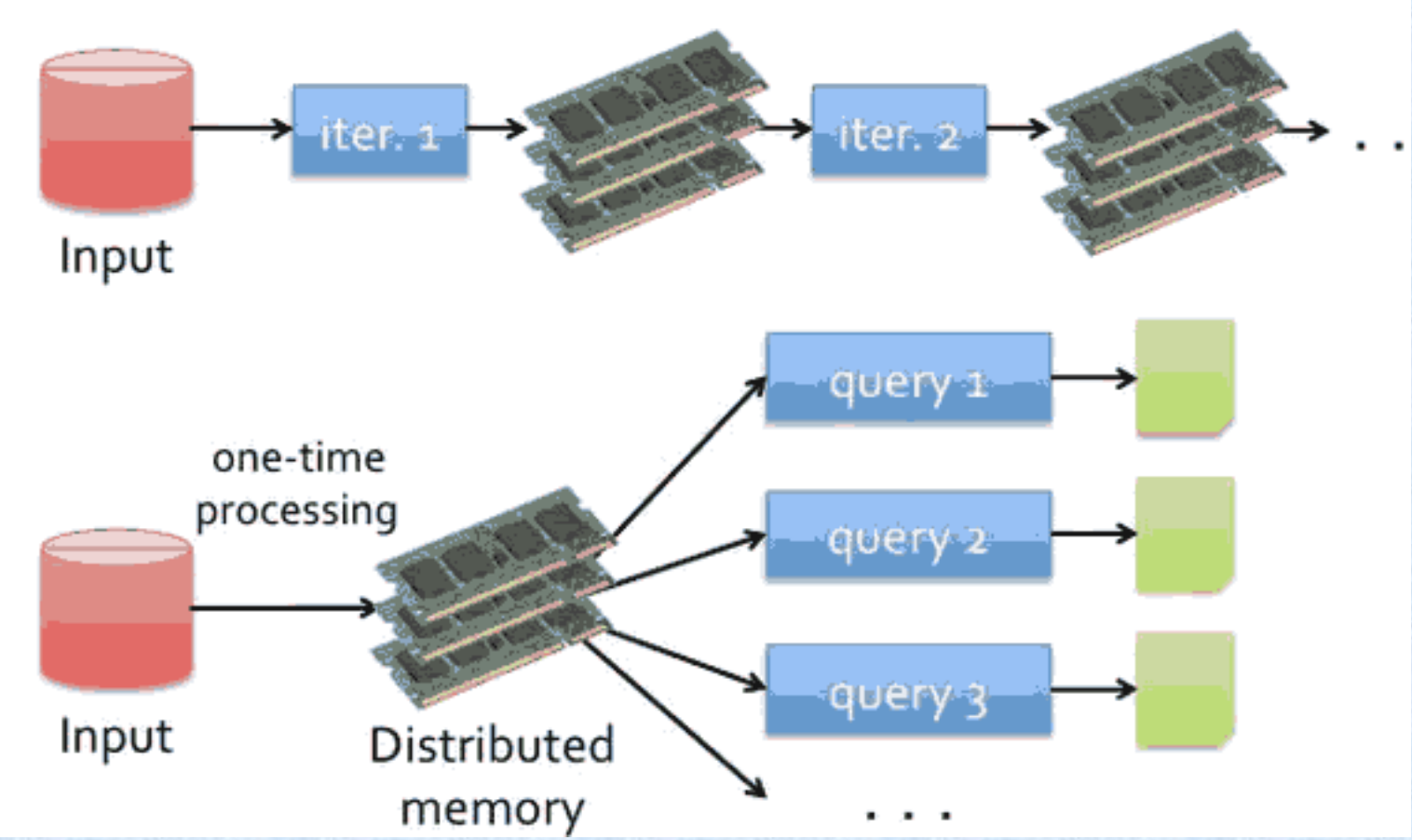




MapReduce Computing



Spark Computing

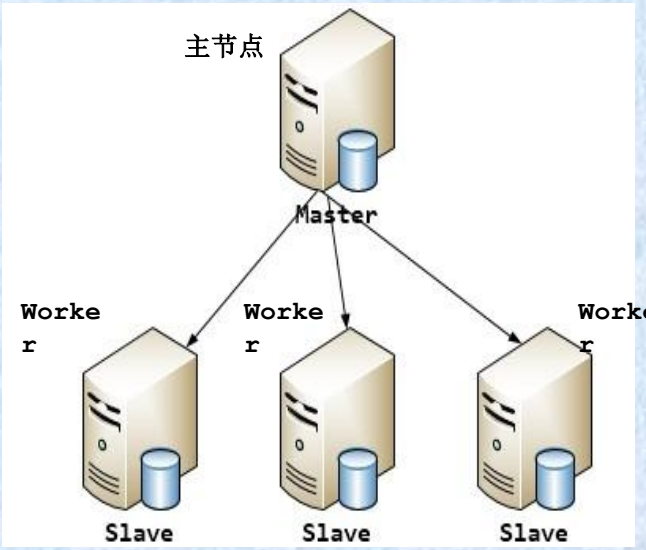


Spark 功能框架

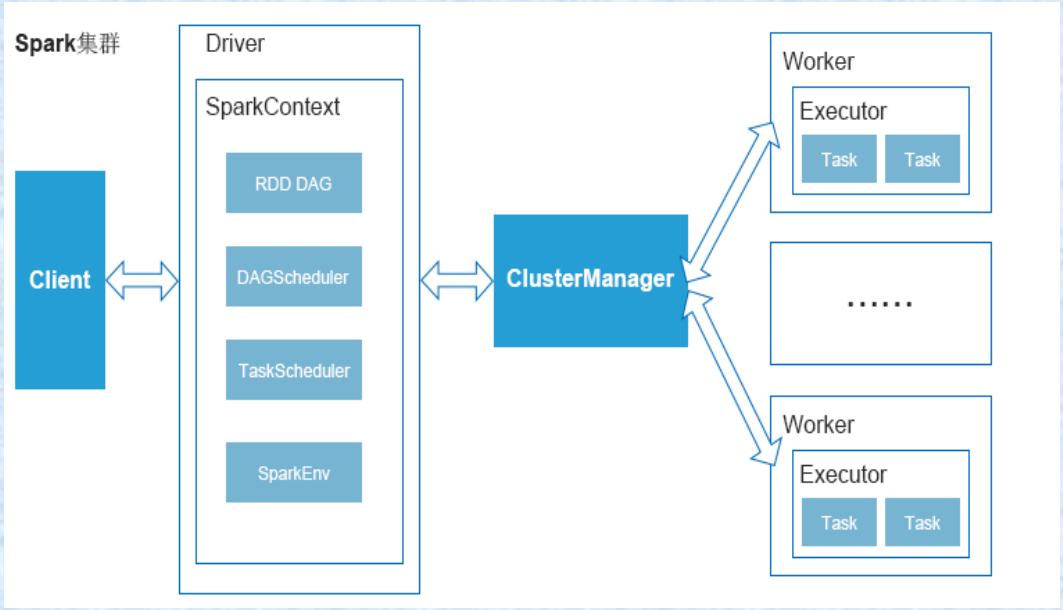
- Shark SQL：SQL查询引擎。通过Hive的HQL语句解析，把HQL翻译成Spark上的RDD操作。
- Spark Core：内存计算模型。提供有向无环图（DAG）的分布式并行计算框架，支持内存多次迭代计算和数据共享，大大减少了迭代之间I/O的开销，对于需要进行多次迭代的数据计算性能有很大提升
- Spark Streaming：支持流计算。将流数据按照batch size（如1秒）分成一段一段的数据段（Discretized Stream），每一段数据都转换成Spark的RDD（Resilient Distributed Dataset），对RDD经过操作变成中间结果保存在内存中，Spark流计算引擎也可根据需求将中间结果存储到外部设备
- GraphX：Spark的图并行计算框架。
- MLBase：支持机器学习的组件库。

Spark计算架构

系统架构仍采用了Master/Slave结构，即集群由一个主节点Master和多个从节点Worker组成，Master作为整个集群的控制节点负责整个集群的运行管理，Worker作为计算节点接受主节点命令并报告本节点状态。



系统架构



计算逻辑架构　　计算物理架构

系统软件（功能组件）到系统硬件（服务器）的部署关系如下：

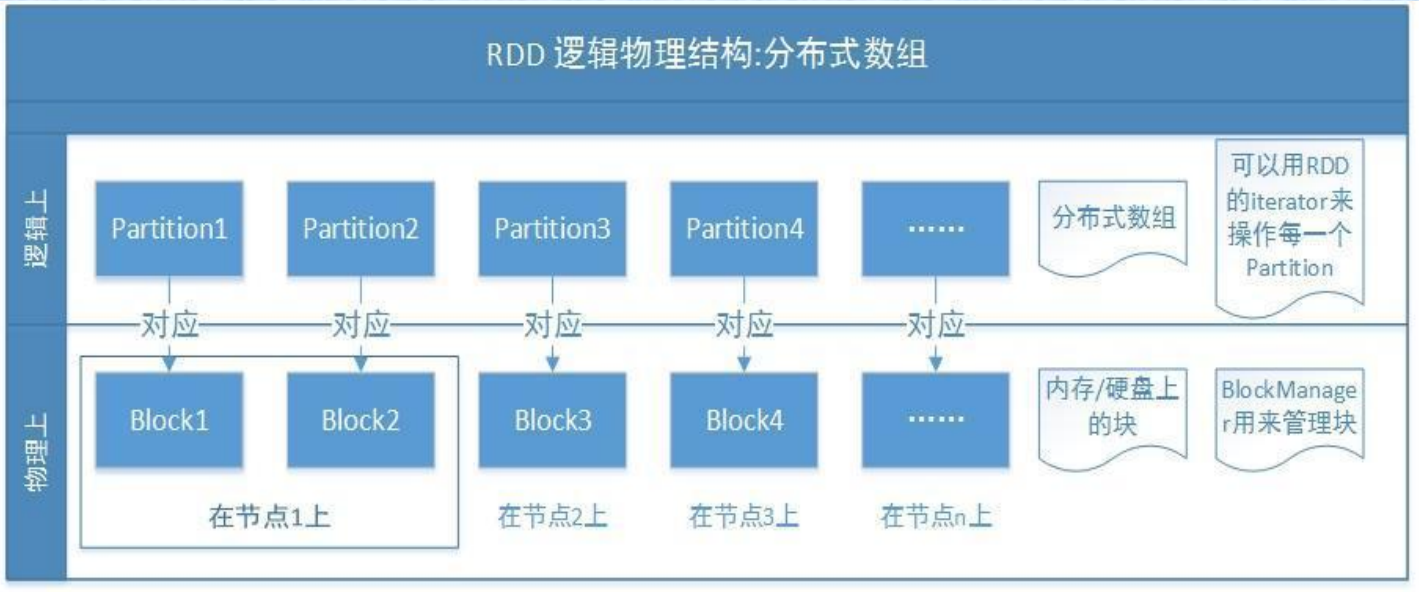
- **主节点（Master）：** 部署有ClusterManager（Standalone模式是Master程序，分布式模式是YARN 的ResourceManager）；
- **工作节点（Worker）：** 部署有YARN的 NodeManager，ApplicationMaster，Executor，以及由Executor启动的Task线程；
- **客户端节点（Client）：** 应用程序Application。

Spark集群有三种典型的运行模式：

- Standalone模式
- YARN-Client模式
- YARN-Cluster模式

RDD 数据模型

RDD（Resilient Distributed Datasets）定义为弹性分布式数据集，即一组不可改变、可并行计算、分区的（partitioned）数据集集合。RDD既是一个数据模型也是一个内存抽象模型。在逻辑结构上，RDD可以理解为一个数组，数组的元素即是分区Partition；在物理数据存储上，RDD的每一个Partition对应的就是一个数据块Block，Block可以有多个副本，分别存储在不同节点的内存中，当内存不够时还可以持久化存储到磁盘上。



RDD的五个属性

- A list of partitions
  一个rdd有多个分区
- A function for computing each split
  作用在每一个分区中函数
- A list of dependencies on other RDDs
  一个rdd会依赖于很多其他RDD，这里就涉及到rdd的依赖关系
- Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  （可选项） 针对于kv类型的rdd才会有分区函数（必须要产生shuffle），分区函数就决定了数据会流入到子rdd的那些分区中
- Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
  （可选项）一个列表，存储每个Partition的优先位置 ，数据位置最优(spark在进行任务分配的时候，会优先考虑存有数据的worker节点来进行任务计算)

RDD特点

- immutable：任何操作都不会改变RDD本身，只会创造新的RDD，但需记录RDD的转换过程，以支持无共享数据读写同步及可重算性
- partitioned：RDD是分布存储在集群节点上的、分区的（partitioned）数据集，以分区（partition）作为最小存储和处理单位，可通过分区方法（如采用Hash分区）来优化存储结构
- in parallel：一个Task对应一个partition，Task之间相互独立、并行计算
- fault-tolerant：基于Lineage的高容错性，对于丢失的部分partitions只需根据其Lineage就可重新计算出来，而不需做checkpoint操作
- persistence：必须是可序列化的，可通过控制存储级别（内存、磁盘等）来进行重用，当内存空间不足时可把RDD存储于磁盘上

RDD 算子

算子是RDD中定义的外部函数，可以对RDD中的数据进行转换和操作。RDD算子有转换（Transformation）和操作（Action）两种。其中，转换又分为数值型（value）Transformation和键值对型（key-value）Transformation两种。

✓ **Transformation** 按照一定的准则将一个RDD转换生成另一个新RDD，即返回值还是一个RDD。但Transformation属于延迟转换，即对一个RDD执行Transformation动作时并不是立即进行转换，而是记住其执行逻辑，等到有Action操作的时候才真正启动转换过程完成计算。Transformation算子有map, filter, join, cogroup, ...等多种类型。

✓ **Action** 是完成对RDD的计算后返回结果或把RDD写到存储系统中，它也是触发Spark计算流程的动因，Action的返回值不是一个RDD。Action算子有count, collect, reduce, lookup和save等操作。

基本算子列表

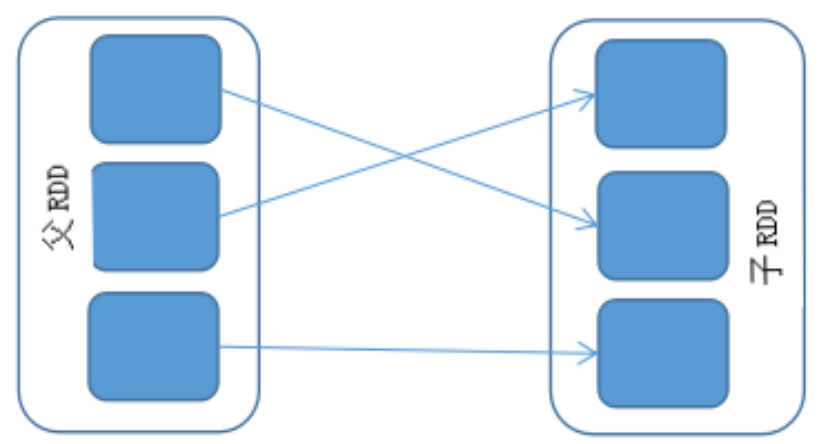
Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ 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

RDD 依赖(Dependency) 与血缘(Lineage)

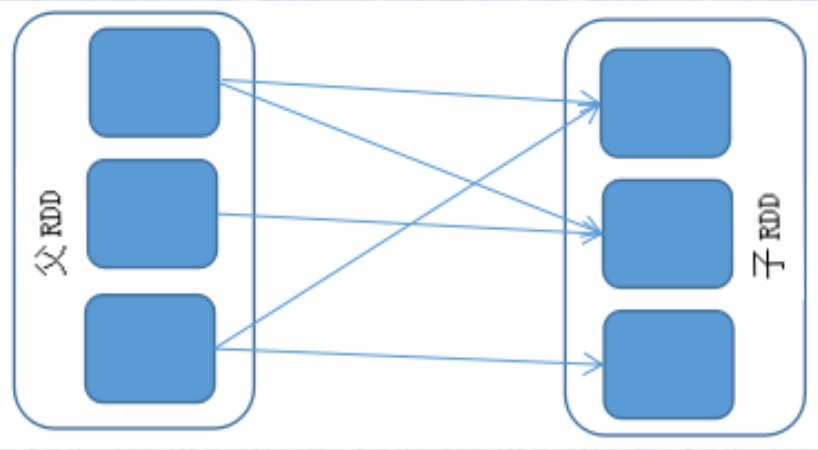
■ 依赖关系（Dependency）

对RDD的转换操作都是粗粒度的，一个旧RDD的转换操作会产生一个新的RDD，新旧RDD之间（又称父子RDD）会形成一个前后依赖关系，即所谓的dependency。Spark中存在两种依赖关系：

- ✓ **窄依赖（narrow dependencies）**：父RDD的每一个分区最多被子RDD的一个分区所用，表现为父RDD的一个分区对应于子RDD的一个分区或父RDD的多个分区对应于子RDD的一个分区，即转换前后父子的分区对应关系是一对一或多对一映射。
- ✓ **宽依赖（wide dependencies）**：子RDD的一个分区依赖于父RDD的所有分区或多个分区，父RDD的一个分区会被子RDD的多个分区使用，即转换前后父子的分区对应是一对多或多对多映射。

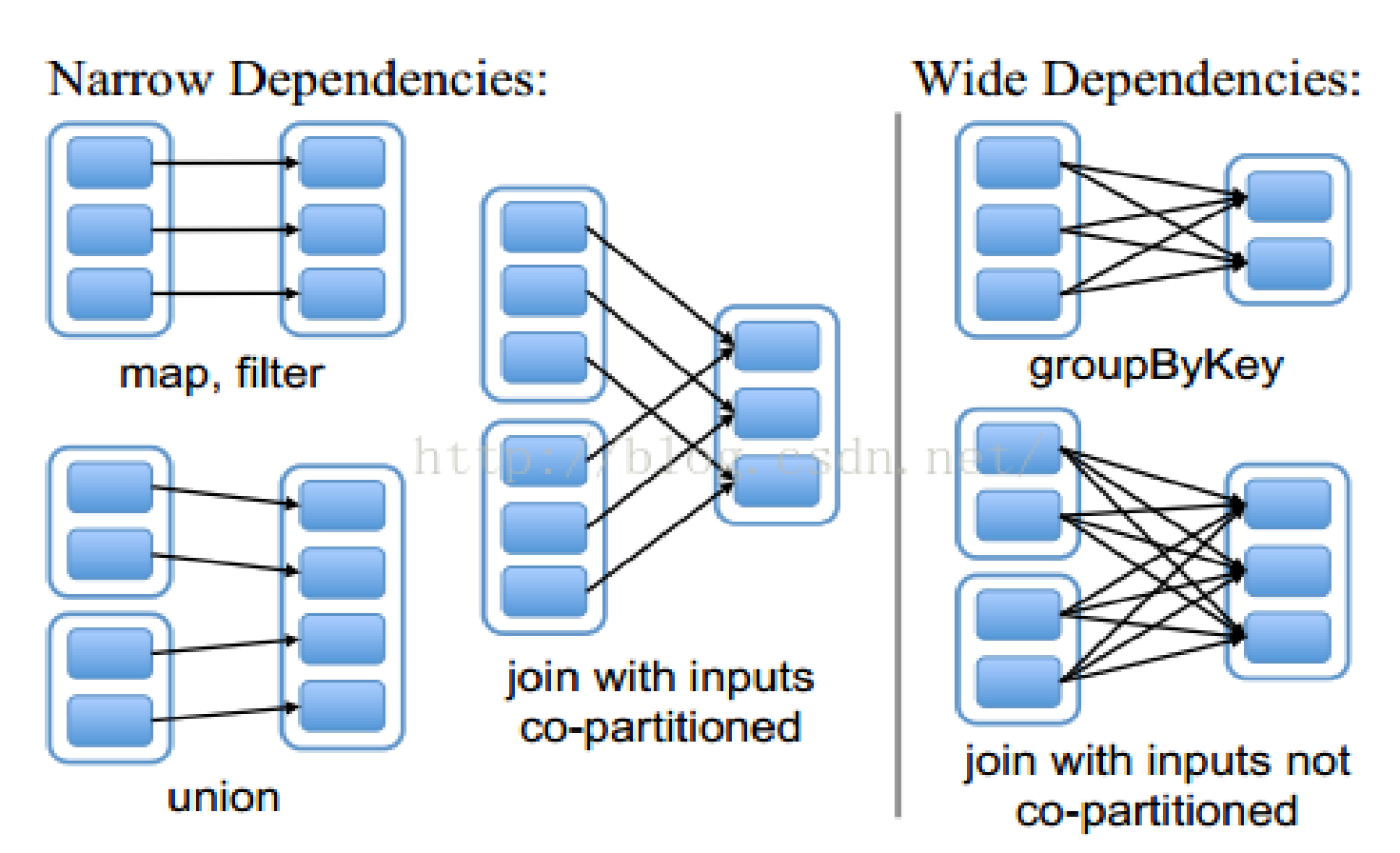


窄依赖
（narrow dependency）

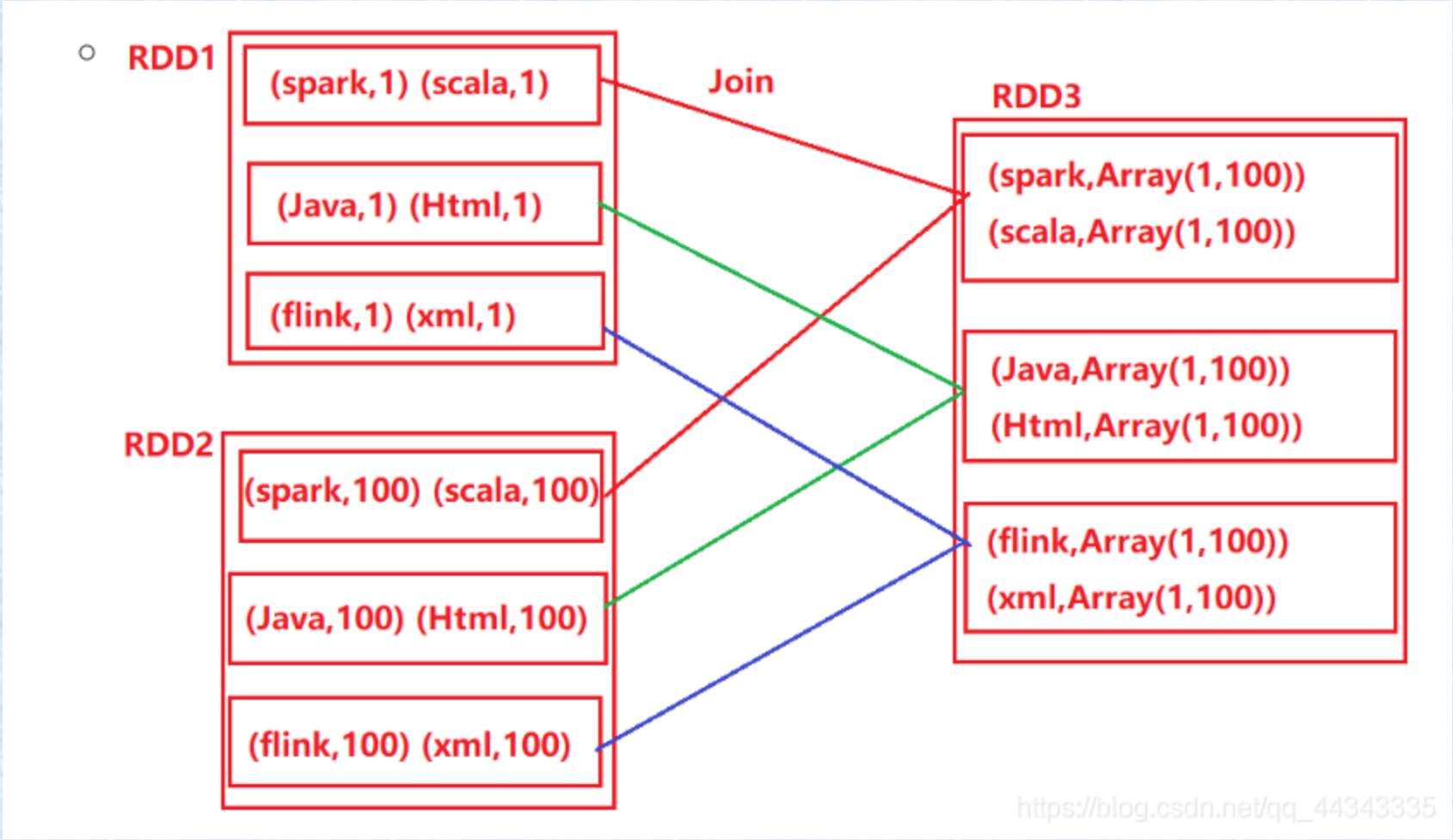


宽依赖
（wide dependency）

RDD算子的 依赖类型


Narrow Dependencies:
map, filter
union
join with inputs
co-partitioned
Wide Dependencies:
groupByKey
join with inputs not
co-partitioned

窄依赖算例：join



- 窄依赖的节点（RDD）关系如流水线一般，由于前后RDD的分区是一对一或多对一关系，所以当某个节点失败后只需重新计算父节点的分区即可；

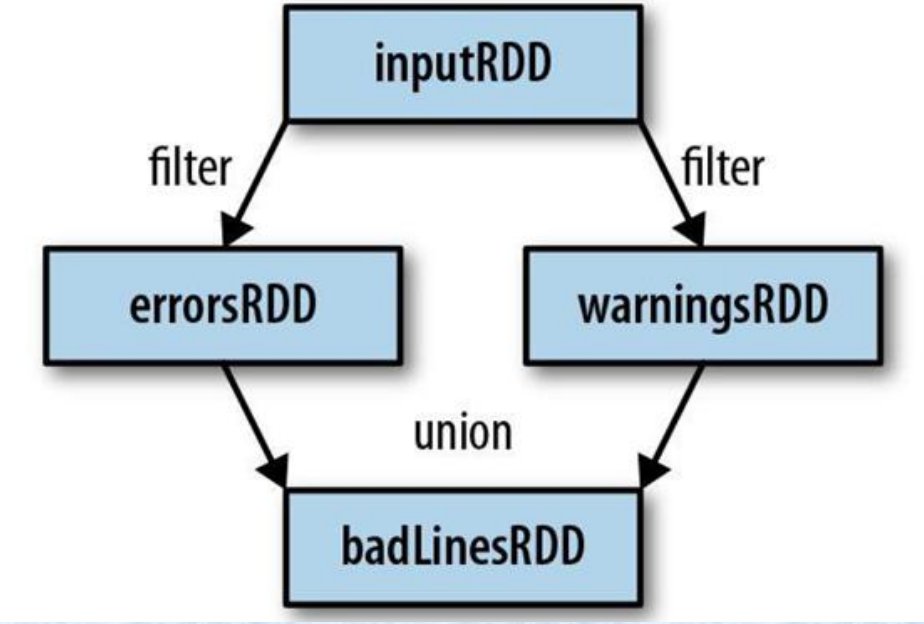
- 宽依赖是一对多映射，重算的父RDD分区对应多个子RDD分区，这样父RDD 中只有一部分数据是被用于恢复这个丢失的子RDD分区，而另一部分数据对应子RDD的其它未丢失分区，这就造成了多余计算。更一般的情况，宽依赖子RDD分区通常来自多个父RDD分区，极端情况下所有的父RDD分区都要进行重新计算；

- 窄依赖允许在一个集群节点上以流水线方式（pipeline）计算所有父分区，比如逐个分区地执行map，然后进行filter操作；

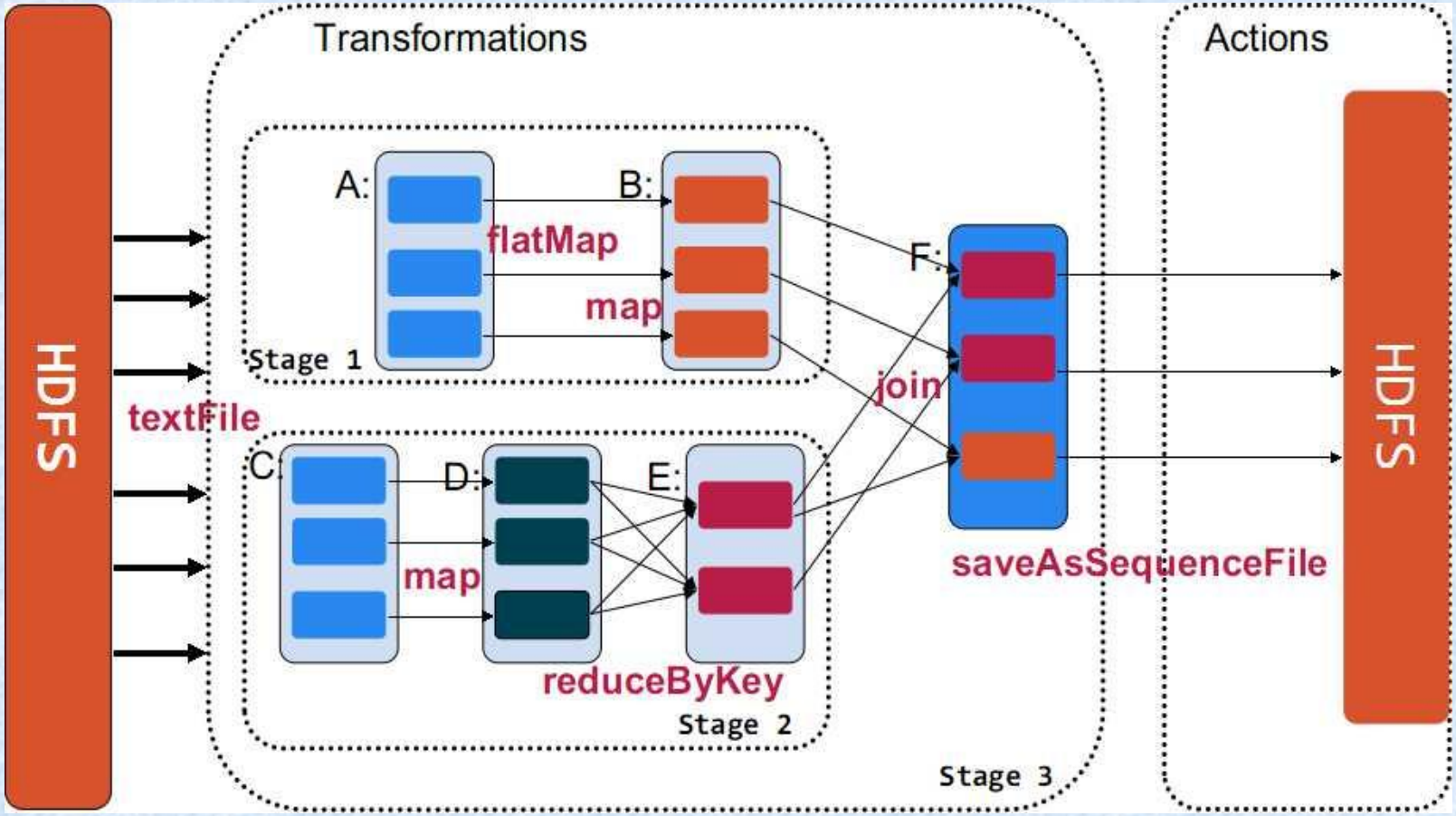
- 宽依赖则需要首先计算好父分区的所有数据，然后在节点之间进行Shuffle，这与MapReduce的中间步骤类似。

血缘关系（Lineage）

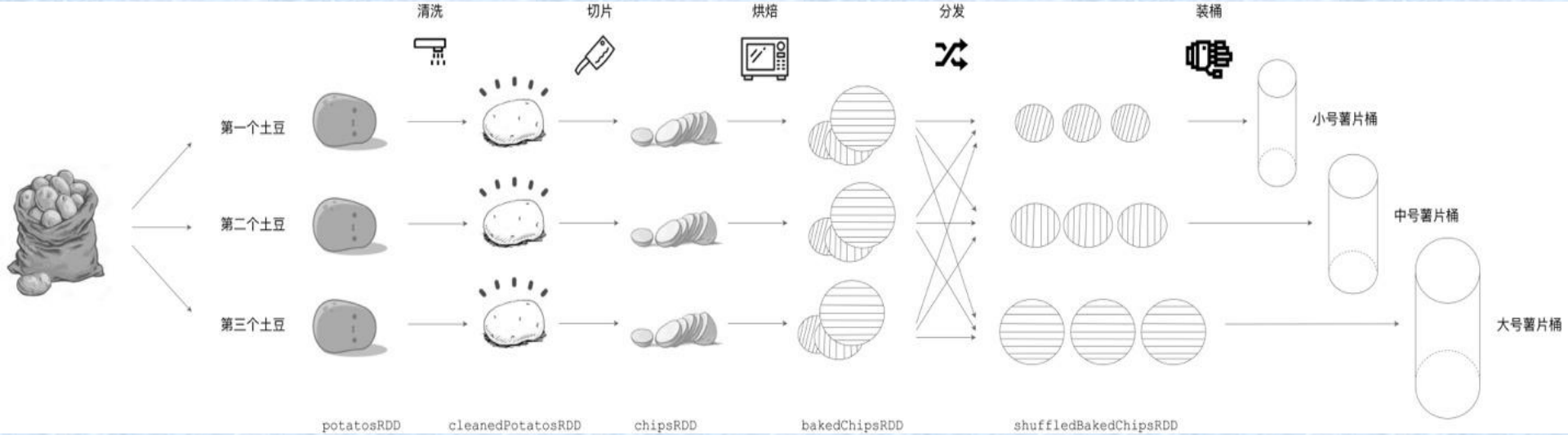
RDD的转换（Transformation）采用惰性调用机制，每个RDD记录父RDD转换的方法，但并不立即实施转换，直到一个操作（Action）触发了这一系列转换, 这种多个转换步骤调用构成了一个链表（如图），称之为血缘（Lineage），RDD的血缘关系图也就是计算模型的有向无环图（DAG）。



```
//创建SparkContext
val sc = new SparkContext(master, "Example",
System.getenv("SPARK_HOME"),
Seq(System.getenv("SPARK_TEST_JAR")))
//RDD A从HDFS文件创建
val rdd_A = sc.textFile(hdfs://.....)
//对A进行flatMap转换产生B
val rdd_B = rdd_A.flatMap((line =>
line.split("\\s+"))).map(word => (word, 1))
// RDD C从HDFS文件创建
val rdd_C = sc.textFile(hdfs://.....)
//对C进行Map转换产生 D
val rdd_D = rdd_C.map(line => (line.substring(10), 1))
//对D进行reduceByKey操作产生E
val rdd_E = rdd_D.reduceByKey((a, b) => a + b)
//对E进行join操作产生F
val rdd_F = rdd_B.jion(rdd_E)
//通过saveAsSequenceFile操作将RDD F写入存储系统
rdd_F.saveAsSequenceFile(hdfs://....)
```


HDFS
textFile
Transformations
A:
B:
flatMap
map
Stage 1
C:
D:
E:
map
reduceByKey
Stage 2
F:
join
Stage 3
Actions
saveAsSequenceFile
HDFS

Spark算例一：土豆加工坊



土豆工坊	流水线	不同阶段的食材形态	不同食材形态之间的关系	流水线上的加工方法	每种食材形态中的食材数量
Spark	节点内存	RDD	dependencies属性	compute属性	partitions属性

Spark算例二：WordCount

一个纯文本文件，内容非常简单，只有 3 行文字

```
1 I love potatos
2 I love chips
3 I want more chips
```

一，需要将文件内容读取到计算节点内存，同时对数据进行分片；

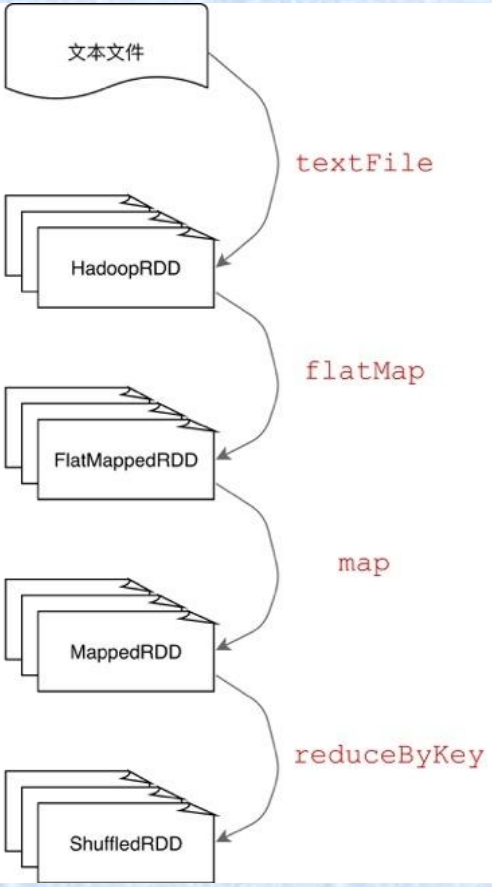
二，对于每个数据分片，我们要将句子分割为一个个的单词；

三，同样的单词可能存在于多个不同的分片中（如单词 I），因此需要对单词进行分发，从而使得同样的单词只存在于一个分片之中；

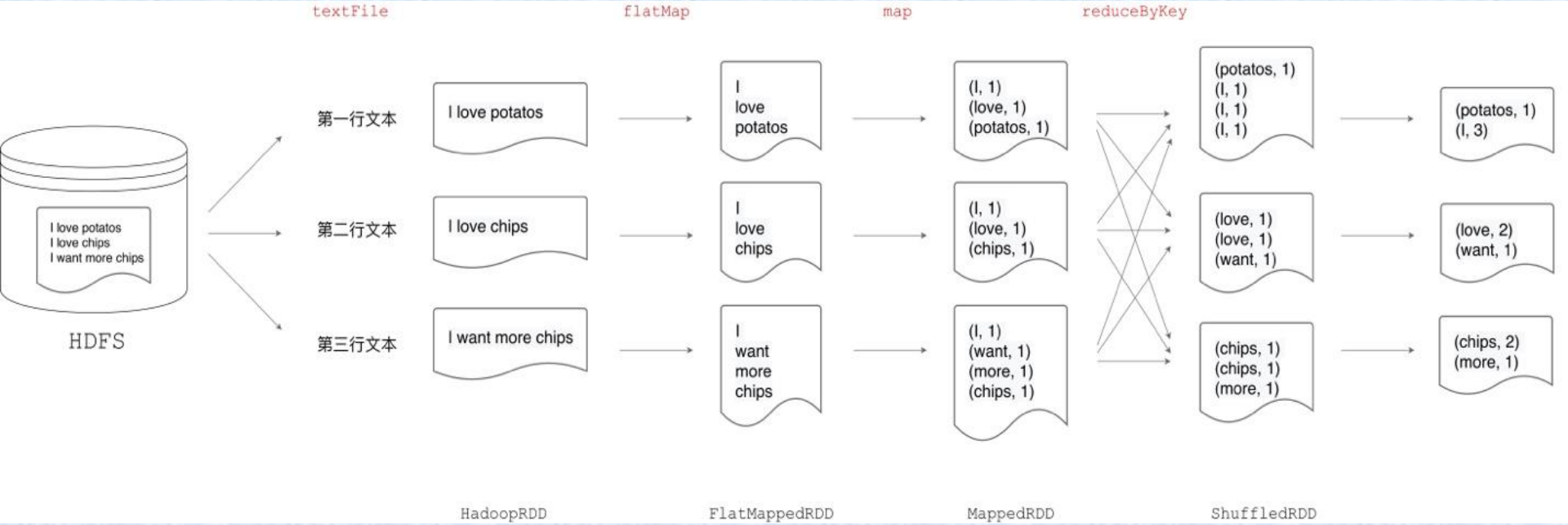
四，最后，在所有分片上计算每个单词的计数。

Spark算例二：WordCount（续）

```scala
val lines = spark.sparkContext.textFile("hdfs://bigdata:9000/users/wulei/text.txt")
// HadoopRDD[1] at textFile

val words = lines.flatMap(_.split(" "))
// FlatMappedRDD[2] at flatMap

val wordsKV = words.map((_, 1))
// MappedRDD[3] at map

val wordCounts = wordsKV.reduceByKey(_ + _)
// ShuffledRDD[4] at reduceByKey

wordCounts.collect()
// Array((love,2), (potatos,1), (want,1), (I,3), (more,1), (chips,2))
```



Spark算例二：WordCount（续）



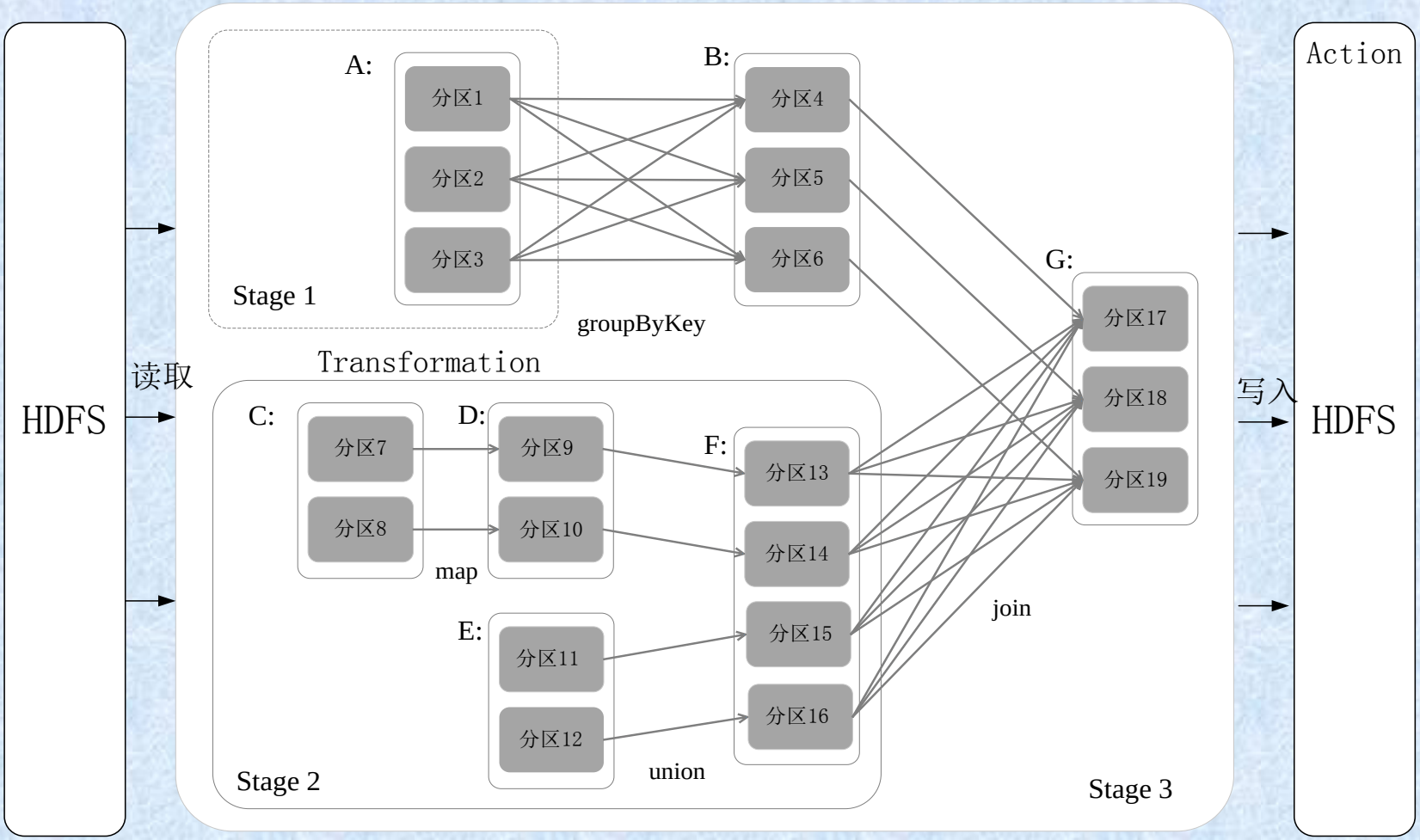
Spark 作业模型

作业模型包括Application（应用程序），Job（作业），Stage（阶段），Task（任务）四个等级。

- 一个Application由多个Jobs组成；
- 一个Job又分为多个Stages，不同的Stage之间需要进行shuffle（混编）；
- 每个Stage由一组执行相关任务但互相间没有Shuffle依赖的Tasks组成（组合成TaskSet）。

Spark在一个Job的DAG基础上通过分析各个RDD分区之间的依赖关系来决定如何划分Stage，划分方法是：

- ✓ 将宽依赖的两边归入不同的Stage，将窄依赖归入一个Stage中
- ✓ 在DAG中进行反向解析，遇到宽依赖就断开
- ✓ 遇到窄依赖就把当前的RDD加入到Stage中


HDFS
读取
Transformation
A:
分区1
分区2
分区3
Stage 1
B:
分区4
分区5
分区6
groupByKey
C:
分区7
分区8
D:
分区9
分区10
map
E:
分区11
分区12
union
F:
分区13
分区14
分区15
分区16
Stage 2
G:
分区17
分区18
分区19
join
Stage 3
写入
Action
HDFS

Spark双层多级调度模型

整个调度架构分为**计算需求调度**（Application/Job/Stage/Task）和**计算资源配置**（Worker/Executor/TaskThread）两层。

在需求调度层面又分为：

- ✓ Job调度（由DAGScheduler承担）
- ✓ Task调度（由TaskScheduler承担）

在资源配置层面则需决定：

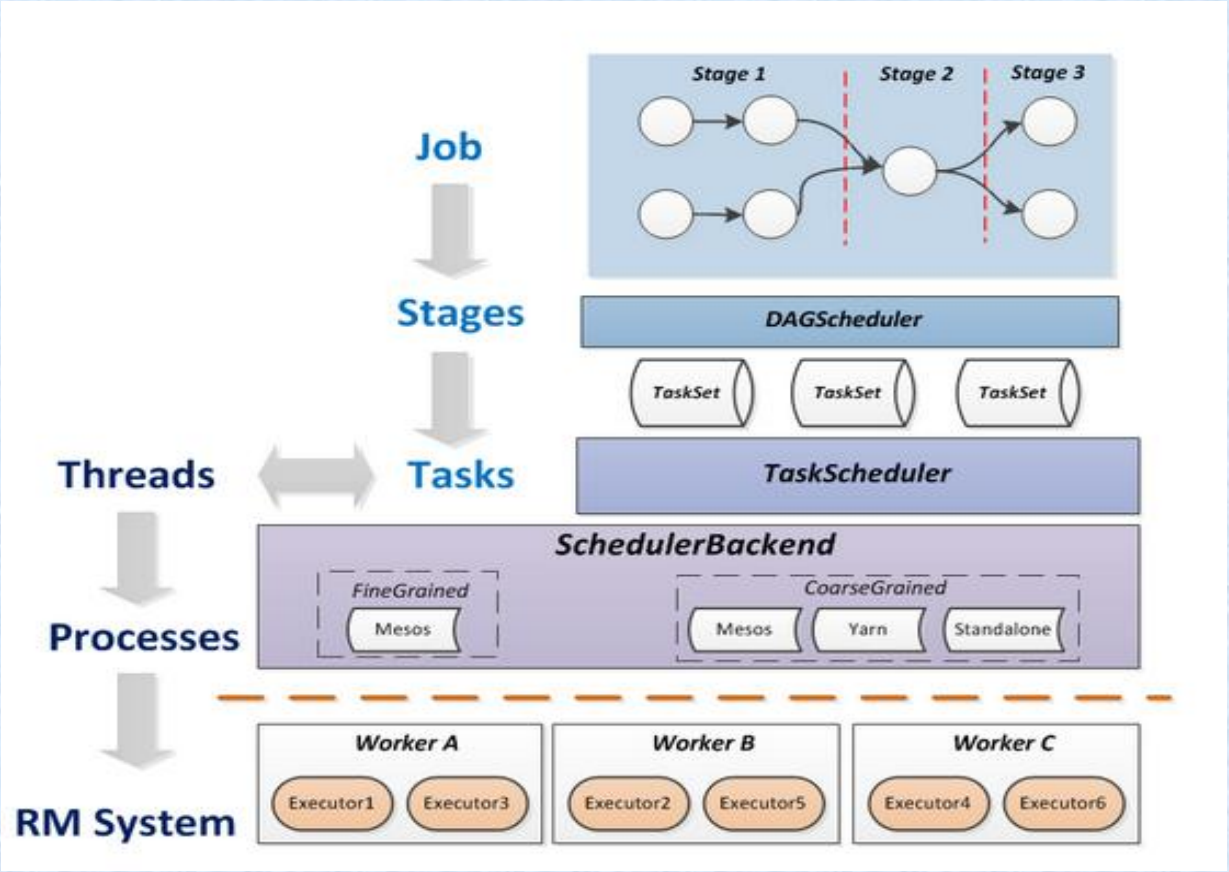
- ✓ 每个Worker上启动多少Executor进程，分配多少资源
- ✓ 每个Executor内运行多少个Task线程等

下图清楚地描绘了这种双层调度模型：

✓ 上层包括Job，Stage，Task等计算项，由DAGScheduler完成划分调派

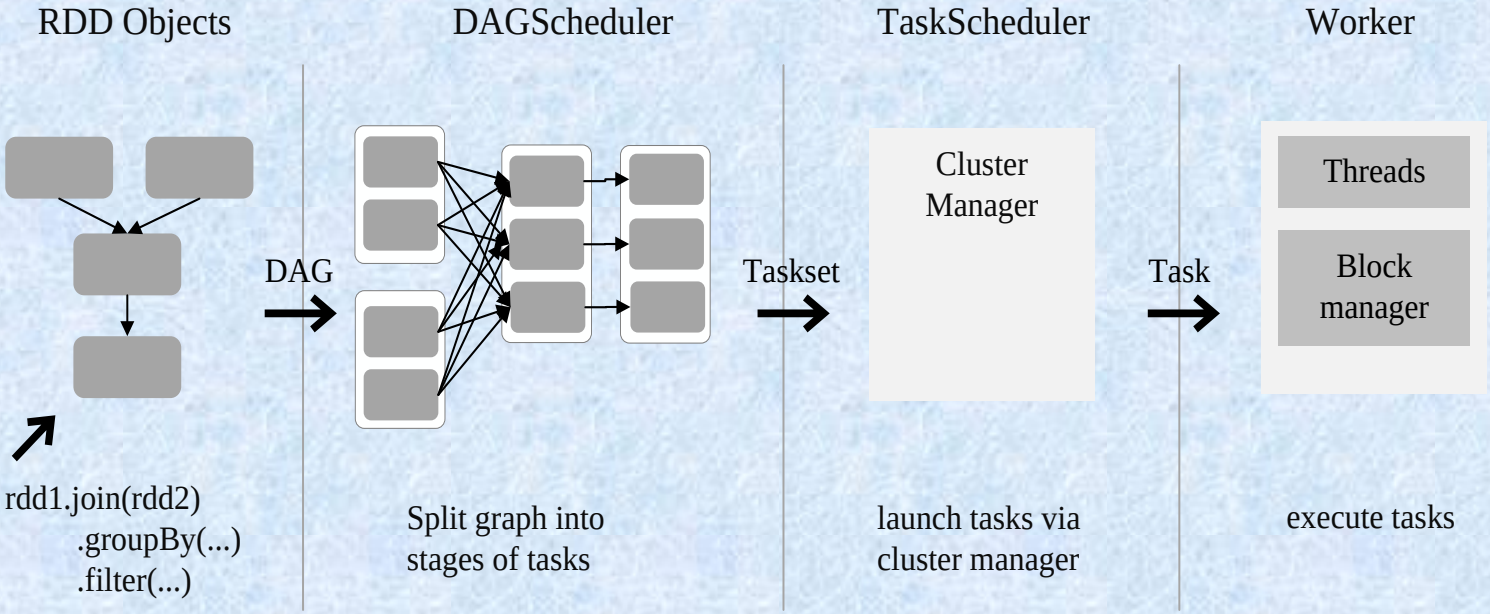
✓ 下层包括Worker，Executor，Thread，由SchedulerBackend负责分派

✓ 上层计算任务的调度（即如何将具体的RDD分区映射到Worker上的Task线程，或者说如何将Task分发到集群的Worker节点上去执行）则是由TaskSetManager通过TaskScheduler与下层的计算资源管理器（SchedulerBackend）的协调来实现



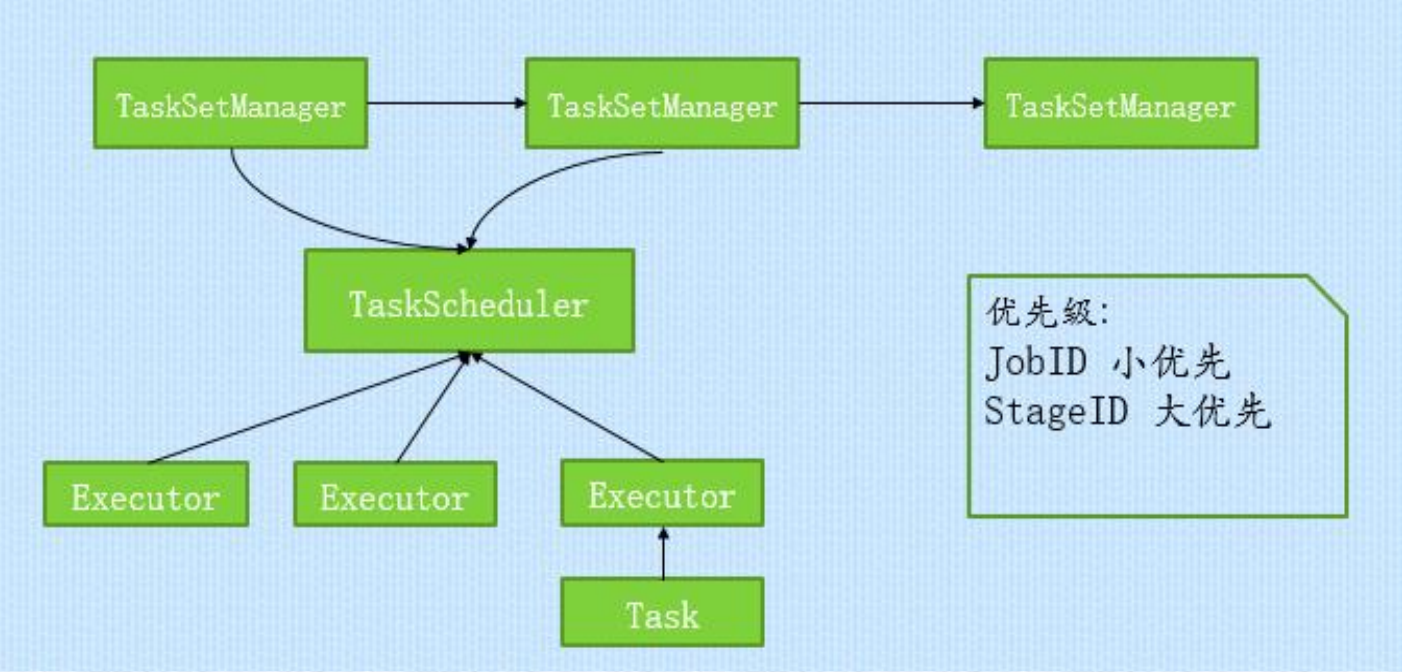
Job调度算法

在Application内部，调度流程如图所示，RDD处理流程构成一个DAG，然后由DAGScheduler按照shuffle dependency将DAG划分成多个Stage，每个Stage包含的分区组成一个TaskSet，DAGScheduler通过TaskScheduler接口提交TaskSet，这个TaskSet最终会触发TaskScheduler构建一个TaskSetManager的实例。



- **FIFO调度策略**

先进先出(First-In-First-Out)策略，Pool直接管理TaskSetManager。每个Job都有JobID，每个TaskSetManager都带有了其对应的Stage的StageID，Pool最终根据JobID小优先、StageID大优先的原则来调度TaskSetManager，如图所示。



- **Fair调度策略**

公平调度策略。目前采用的是两级结构，即rootPool管理一组子调度池（Pool），子调度池进一步管理属于该调度池的TaskSetManager，如图所示。在Pool之间，TaskScheduler采用轮询（Round Robin）方式分配资源。