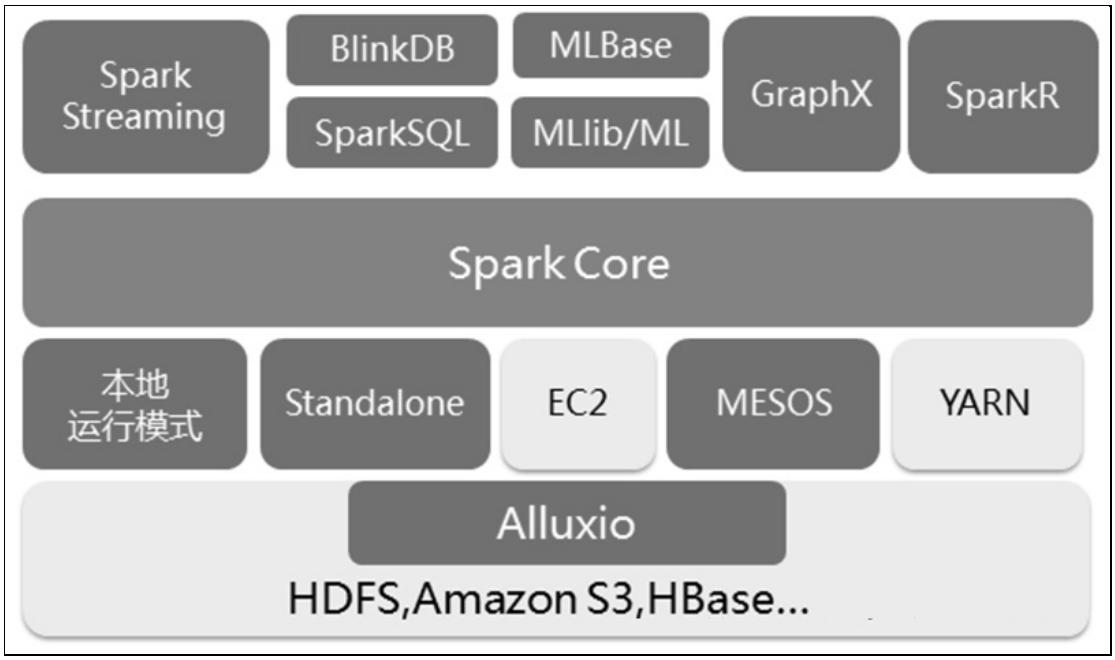
Spark Core 核心知识--RDD

目录

1、Spark 核心功能



Spark Core 提供 Spark 最基础的最核心的功能，主要包括：

SparkContext

通常而言，DriverApplication 的执行与输出都是通过 SparkContext 来完成的，在正式提交 Application 之前，首先需要初始化 SparkContext。SparkContext 隐藏了网络通信、分布式部署、消息通信、存储能力、计算能力、缓存、测量系统、文件服务、Web 服务等内容，应用程序开发者只需要使用 SparkContext 提供的 API 完成功能开发。

SparkContext 内置的 DAGScheduler 负责创建 Job，将 DAG 中的 RDD 划分到不同的 Stage，提交 Stage 等功能。

SparkContext 内置的 TaskScheduler 负责资源的申请、任务的提交及请求集群对任务的调度等工作。

存储体系

Spark 优先考虑使用各节点的内存作为存储，当内存不足时才会考虑使用磁盘，这极大地减少了磁盘 I/O，提升了任务执行的效率，使得 Spark 适用于实时计算、流式计算等场景。此外，Spark 还提供了以内存为中心的高容错的分布式文件系统 Tachyon 供用户进行选择。Tachyon 能够为 Spark 提供可靠的内存级的文件共享服务。

计算引擎

计算引擎由 SparkContext 中的 DAGScheduler、RDD 以及具体节点上的 Executor 负责执行的 Map 和 Reduce 任务组成。DAGScheduler 和 RDD 虽然位于 SparkContext 内部，但是在任务正式提交与执行之前将 Job 中的 RDD 组织成有向无关图（简称 DAG）、并对 Stage 进行划分决定了任务执行阶段任务的数量、迭代计算、shuffle 等过程。

部署模式

由于单节点不足以提供足够的存储及计算能力，所以作为大数据处理的 Spark 在

SparkContext 的 TaskScheduler 组件中提供了对 Standalone 部署模式的实现和 YARN、Mesos 等分布式资源管理系统的支持。通过使用 Standalone、YARN、Mesos、kubernetes、Cloud 等部署模式为 Task 分配计算资源，提高任务的并发执行效率。除了可用于实际生产环境的 Standalone、YARN、Mesos、kubernetes、Cloud 等部署模式外，Spark 还提供了 Local 模式和 local-cluster 模式便于开发和调试。

2、Spark 扩展功能

为了扩大应用范围，Spark 陆续增加了一些扩展功能，主要包括：

Spark SQL

由于 SQL 具有普及率高、学习成本低等特点，为了扩大 Spark 的应用面，因此增加了对 SQL 及 Hive 的支持。Spark SQL 的过程可以总结为：首先使用 SQL 语句解析器（SqlParser）将 SQL 转换为语法树（Tree），并且使用规则执行器（RuleExecutor）将一系列规则（Rule）应用到语法树，最终生成物理执行计划并执行的过程。其中，规则包括语法分析器（Analyzer）和优化器（Optimizer）。Hive 的执行过程与 SQL 类似。

Spark Streaming

Spark Streaming 与 Apache Storm 类似，也用于流式计算。Spark Streaming 支持 Kafka、Flume、Twitter、MQTT、ZeroMQ、Kinesis 和简单的 TCP 套接字等多种数据输入源。输入流接收器（Receiver）负责接入数据，是接入数据流的接口规范。Dstream 是 Spark Streaming 中所有数据流的抽象，Dstream 可以被组织为 DStreamGraph。**Dstream 本质上由一系列连续的 RDD 组成**。

Spark GraphX

Spark 提供的分布式图计算框架。GraphX 主要遵循整体同步并行计算模式（BulkSynchronous Parallell，简称 BSP）下的 Pregel 模型实现。GraphX 提供了对图的抽象 Graph，Graph 由顶点（Vertex）、边（Edge）及继承了 Edge 的 EdgeTriplet（添加了 srcAttr 和 dstAttr 用来保存源顶点和目的顶点的属性）三种结构组成。GraphX 目前已经封装了最短路径、网页排名、连接组件、三角关系统计等算法的实现，用户可以选择使用。

Spark MLlib

Spark 提供的机器学习框架。机器学习是一门涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多领域的交叉学科。MLlib 目前已经提供了基础统计、分类、回归、决策树、随机森林、朴素贝叶斯、保序回归、协同过滤、聚类、维数缩减、特征提取与转型、频繁模式挖掘、预言模型标记语言、管道等多种数理统计、概率论、数据挖掘方面的数学算法。

3、Spark 核心概念

Term	Meaning
Application	User program built on Spark. Consists of a *driver program* and *executors* on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. `save`, `collect`); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

4、Spark 基本架构

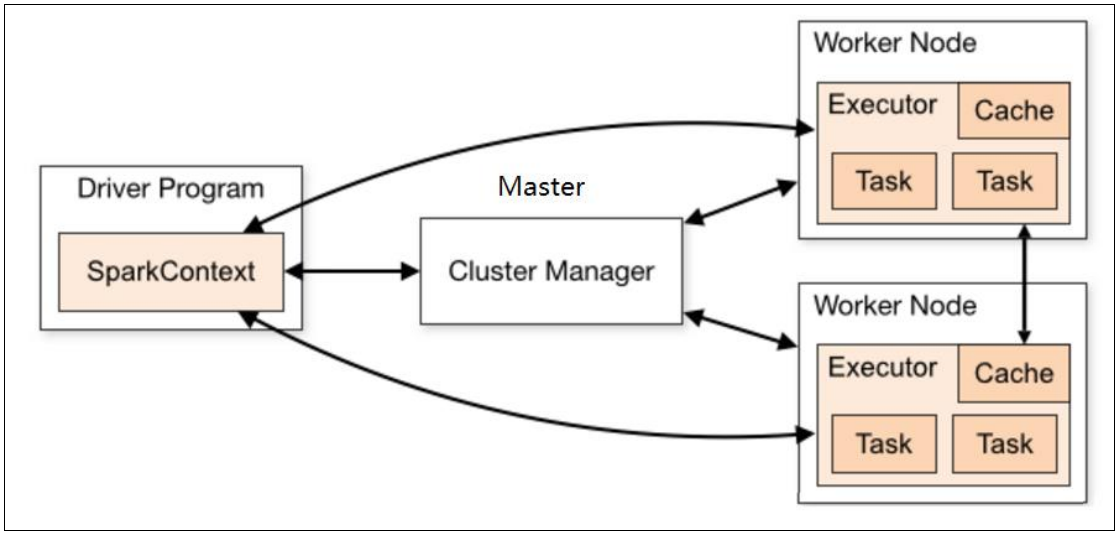
从集群部署的角度来看，Spark 集群由以下部分组成：

Cluster Manager：Spark 的集群管理器，主要负责资源的分配与管理。集群管理器分配的资源属于一级分配，它将各个 Worker 上的内存、CPU 等资源分配给应用程序，但是并不负责对 Executor 的资源分配。目前，Standalone、YARN、Mesos、K8S，EC2 等都可以作为 Spark 的集群管理器。

Master：Spark 集群的主节点。

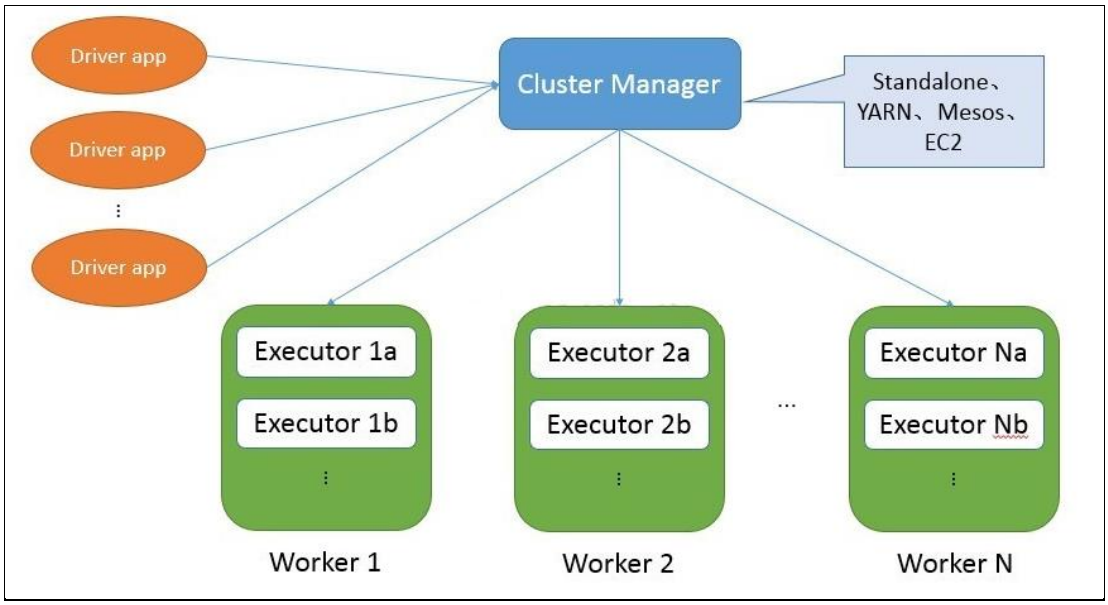
Worker：Spark 集群的工作节点。对 Spark 应用程序来说，由集群管理器分配得到资源的 Worker 节点主要负责以下工作：创建 Executor，将资源和任务进一步分配给 Executor，同步资源信息给 Cluster Manager。

Executor：执行计算任务的一些进程。主要负责任务的执行以及与 Worker、Driver Application 的信息同步。



Driver Appication：客户端驱动程序，也可以理解为客户端应用程序，用于将任务程序转换为 RDD 和 DAG，并与 Cluster Manager 进行通信与调度。

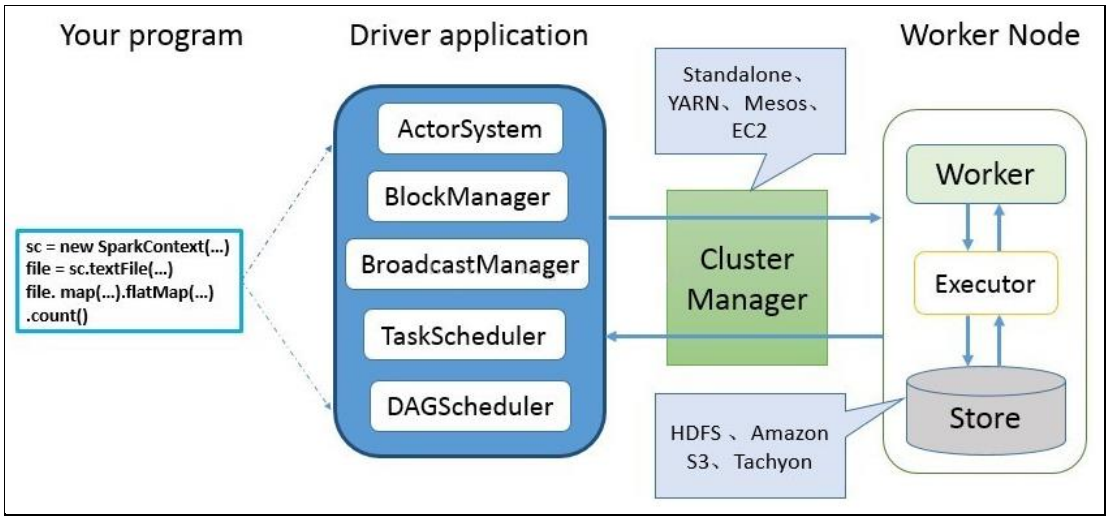
这些组成部分之间的整体关系如下图所示：



Spark 计算平台有两个重要角色，Driver 和 executor，不论是 StandAlone 模式还是 YARN 模式，都是 Driver 充当 Application 的 master 角色，负责任务执行计划生成和任务分发及调度；executor 充当 worker 角色，负责实际执行任务的 task，计算的结果返回 Driver。

5、Spark 编程模型

Spark 应用程序从编写到提交、执行、输出的整个过程如下图所示：



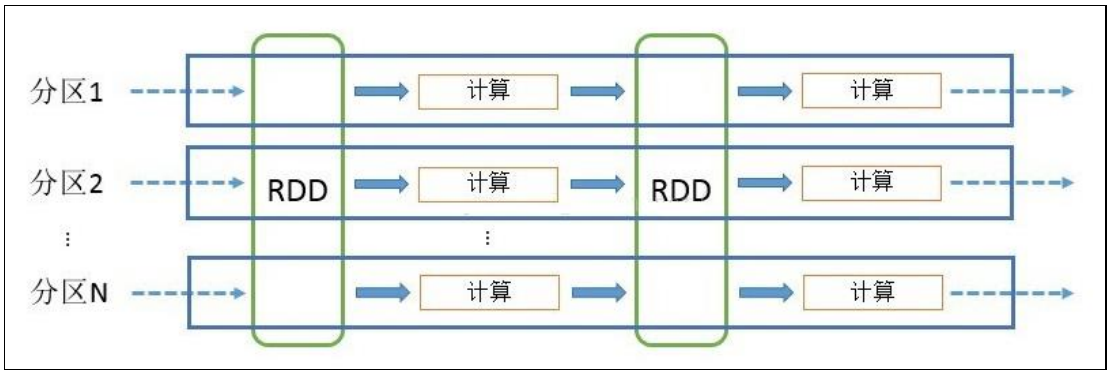
图中描述的步骤如下：

1、用户使用 SparkContext 提供的 API（常用的有 textFile、sequenceFile、runJob、stop 等）编写 Driver Application 程序。此外 SQLContext、HiveContext 及 StreamingContext 对 SparkContext 进行封装，并提供了 SQL、Hive 及流式计算相关的 API。

2、使用 SparkContext 提交的用户应用程序，首先会使用 BlockManager 和 BroadcastManager 将任务的 Hadoop 配置进行广播。然后由 DAGScheduler 将任务转换为 RDD 并组织成 DAG，DAG 还将被划分为不同的 Stage。最后由 TaskScheduler 借助 ActorSystem 将任务提交给集群管理器（ClusterManager）。

3、集群管理器（ClusterManager）给任务分配资源，即将具体任务分配到 Worker 上，Worker 创建 Executor 来处理任务的运行。Standalone、YARN、Mesos、kubernetes、EC2 等都可以作为 Spark 的集群管理器。

计算模型：



RDD 可以看做是对各种数据计算模型的统一抽象，Spark 的计算过程主要是 RDD 的迭代计算过程，如上图。RDD 的迭代计算过程非常类似于管道。分区数量取决于 partition 数量的设定，每个分区的数据只会在一个 Task 中计算。所有分区可以在多个机器节点的 Executor 上并行执行。

6、RDD

6.1、RDD 概述

6.1.1、什么是 RDD

RDD（Resilient Distributed Dataset）叫做分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD 具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

可以从三个方面来理解：

1、数据集 DataSet：故名思议，RDD 是数据集合的抽象，是复杂物理介质上存在数据的一种逻辑视图。从外部来看，RDD 的确可以被看待成经过封装，带扩展特性（如容错性）的数据集合。

2、分布式 Distributed：RDD 的数据可能在物理上存储在多个节点的磁盘或内存中，也就是所谓的多级存储。

3、弹性 Resilient：虽然 RDD 内部存储的数据是只读的，但是，我们可以去修改（例如通过 repartition 转换操作）并行计算计算单元的划分结构，也就是分区的数量。

你将 RDD 理解为一个大的集合，将所有数据都加载到内存中，方便进行多次重用。第一，它是分布式的，可以分布在多台机器上，进行计算。第二，它是弹性的，我认为它的弹性体现在每个 RDD 都可以保存内存中，如果某个阶段的 RDD 丢失，不需要从头计算，只需要提取上一个 RDD，再做相应的计算就可以了

6.1.2、RDD 的属性

```scala
/**
 * A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable,
 * partitioned collection of elements that can be operated on in parallel. This class contains the
 * basic operations available on all RDDs, such as `map`, `filter`, and `persist`. In addition,
 * [[org.apache.spark.rdd.PairRDDFunctions]] contains operations available only on RDDs of key-value
 * pairs, such as `groupByKey` and `join`;
 * [[org.apache.spark.rdd.DoubleRDDFunctions]] contains operations available only on RDDs of
 * Doubles; and
 * [[org.apache.spark.rdd.SequenceFileRDDFunctions]] contains operations available on RDDs that
 * can be saved as SequenceFiles.
 * All operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)])
 * through implicit.
 *
 * 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)
 *
 * All of the scheduling and execution in Spark is done based on these methods, allowing each RDD
 * to implement its own way of computing itself. Indeed, users can implement custom RDDs (e.g. for
 * reading data from a new storage system) by overriding these functions. Please refer to the
 * <a href="http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf">Spark paper</a>
 * for more details on RDD internals.
 */
abstract class RDD[T: ClassTag](
    @transient private var _sc: SparkContext,
    @transient private var deps: Seq[Dependency[_]]
  ) extends Serializable with Logging {
```

1、A list of partitions： 一组分片（Partition），即数据集的基本组成单位

1、一个分区通常与一个计算任务关联，分区的个数决定了并行的粒度；
2、分区的个数可以在创建 RDD 的时候进行设置。如果没有设置，默认情况下由节点的 cores 个数决定；
3、每个 Partition 最终会被逻辑映射为 BlockManager 中的一个 Block，而这个 Block 会被下一个 Task（ShuffleMapTask/ResultTask）使用进行计算

2、A function for computing each split： 一个计算每个分区的函数，也就是算子

分区处理函数-compute
1、每个 RDD 都会实现 compute，用于对分区进行计算；
2、compute 函数会对迭代器进行复合，不需要保存每次计算结果；
3、该方法负责接收 parent RDDs 或者 data block 流入的 records 并进行计算，然后输出加工后的 records。

3、A list of dependencies on other RDDs： RDD 之间的依赖关系：宽依赖和窄依赖

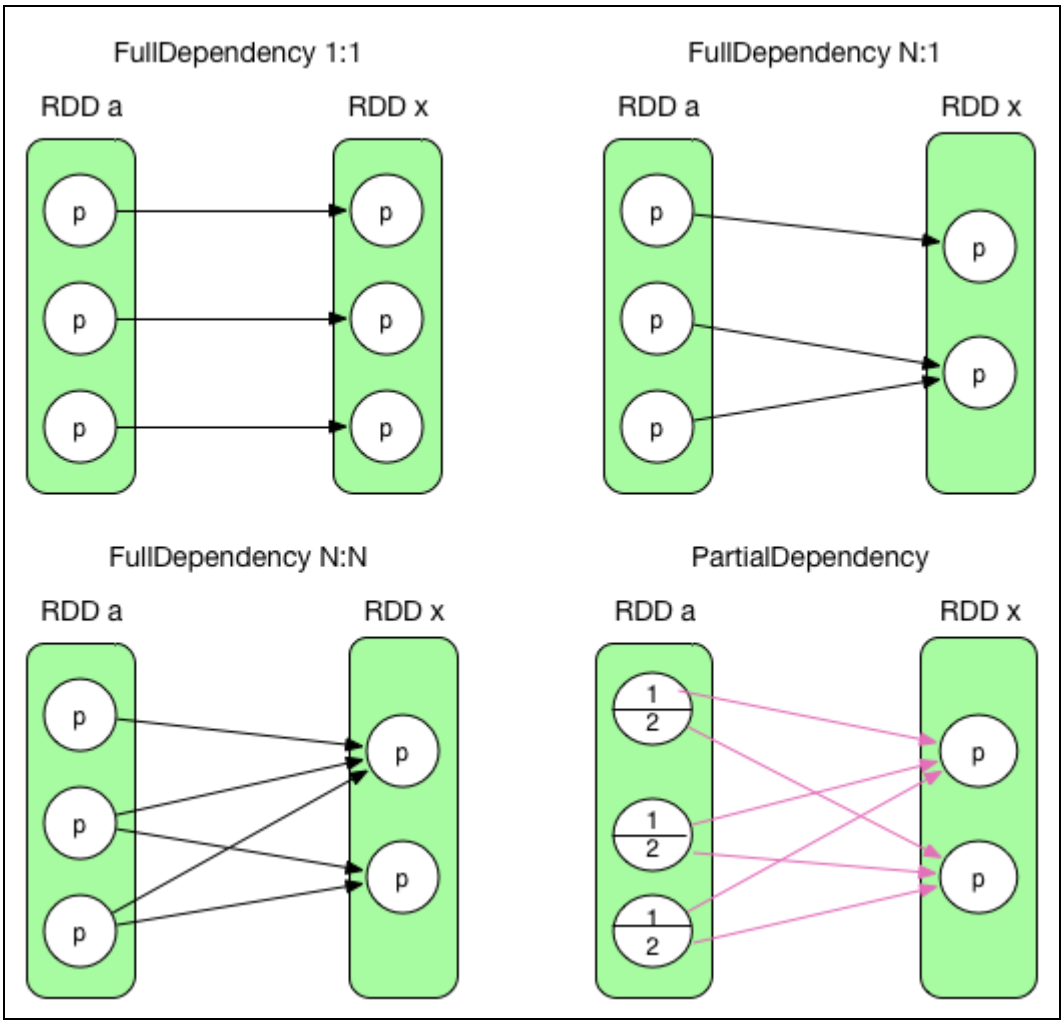
RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。

RDDx 依赖的 parent RDD 的个数由不同的转换操作决定，例如二元转换操作 x = a.join(b)，RDD x 就会同时依赖于 RDD a 和 RDD b。而具体的依赖关系可以细分为完全依赖和部分依赖，详细说明如下：

1、**完全依赖**：一个子 RDD 中的分区可以依赖于父 RDD 分区中一个或多个完整分区。
例如，map 操作产生的子 RDD 分区与父 RDD 分区之间是一对一的关系；对于 cartesian 操作

产生的子 RDD 分区与父 RDD 分区之间是多对多的关系。

2、**部分依赖**：父 RDD 的一个 partition 中的部分数据与 RDD x 的一个 partition 相关，而另一部分数据与 RDD x 中的另一个 partition 有关。
例如，groupByKey 操作产生的 ShuffledRDD 中的每个分区依赖于父 RDD 的所有分区中的部分元素。



在 Spark 中，**完全依赖是 NarrowDependency（黑色箭头），部分依赖是 ShuffleDependency（红色箭头）**，而 NarrowDependency 又可以细分为[1:1]OneToOneDependency、[N:1]NarrowDependency 和[N:N]NarrowDependency，还有特殊的 RangeDependency (只在 UnionRDD 中使用)。

需要注意的是，对于[N:N]NarrowDependency 很少见，最后生成的依赖图和 ShuffleDependency 没什么两样。只是对于父 RDD 来说，有一部分是完全依赖，有一部分是部分依赖。所以也只有[1:1]OneToOneDependency 和[N:1]NarrowDependency 两种情况。

4、Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned): 一个 Partitioner，即 RDD 的分片函数。
当前 Spark 中实现了两种类型的分片函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于于 key-value 的 RDD，才会有 Partitioner，非 key-value

的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分片数量，也决定了 parent RDD Shuffle 输出时的分片数量。

1、只有键值对 RDD，才会有 Partitioner。其他非键值对的 RDD 的 Partitioner 为 None；
2、它定义了键值对 RDD 中的元素如何被键分区，能够将每个键映射到对应的分区 ID，从 0 到”numPartitions- 1”上；
3、Partitioner 不但决定了 RDD 本身的分区个数，也决定了 parent RDD shuffle 输出的分区个数。
4、在分区器的选择上，默认情况下，如果有一组 RDDs（父 RDD）已经有了 Partitioner，则从中选择一个分区数较大的 Partitioner；否则，使用默认的 HashPartitioner。
5、对于 HashPartitioner 分区数的设置，如果配置了 spark.default.parallelism 属性，则将分区数设置为此值，否则，将分区数设置为上游 RDDs 中最大分区数。

5、Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)：一个列表，存储存取每个 Partition 的优先位置（preferred location）。

1、对于一个 HDFS 文件来说，这个列表保存的就是每个 Partition 所在的块的位置。
2、按照”移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。
3、每个子 RDDgetPreferredLocations 的实现中，都会优先选择父 RDD 中对应分区的 preferedLocation，其次才选择自己设置的优先位置。

6.2、创建 RDD

创建 RDD 主要有两种方式：官网解释
There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

1、由一个已经存在的 Scala 数据集合创建

```
val rdd = sc.parallelize(Array(1,2,3,4,5,6,7,8))
val rdd = sc.makeRDD(Array(1,2,3,4,5,6,7,8))
```

2、由外部存储系统的数据集创建，包括本地的文件系统，还有所有 Hadoop 支持的数据集，比如 HDFS、Cassandra、HBase 等

```
val rdd = sc.textFile("hdfs://myha01/spark/wc/input/words.txt")
```

3、扩展

从 HBase 当中读取
从 ElasticSearch 中读取

6.3、RDD 的编程 API

官网：

http://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds

6.3.1、Transformation

官网：http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

RDD 中的所有转换（Transformation）都是延迟加载的，也就是说，它们并不会直接计算结果。相反的，它们只是记住这些应用到基础数据集（例如一个文件）上的转换动作。**只有当发生一个要求返回结果给 Driver 的动作时，这些转换才会真正运行**。这种设计让 Spark 更加有效率地运行。

常用的 Transformation：

转换	含义
map(func)	返回一个新的 RDD，该 RDD 由每一个输入元素经过 func 函数转换后组成
filter(func)	返回一个新的 RDD，该 RDD 由经过 func 函数计算后返回值为 true 的输入元素组成
flatMap(func)	类似于 map，但是每一个输入元素可以被映射为 0 或多个输出元素（所以 func 应该返回一个序列，而不是单一元素）
mapPartitions(func)	类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]
mapPartitionsWithIndex(func)	类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是(Int, Interator[T]) => Iterator[U]
sample(withReplacement, fraction, seed)	根据 fraction 指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed 用于指定随机数生成器种子
union(otherDataset)	对源 RDD 和参数 RDD 求并集后返回一个新的 RDD
intersection(otherDataset)	对源 RDD 和参数 RDD 求交集后返回一个新的 RDD
distinct([numTasks]))	对源 RDD 进行去重后返回一个新的 RDD
groupByKey([numTasks])	在一个(K,V)的 RDD 上调用，返回一个(K, Iterator[V])的 RDD
reduceByKey(func, [numTasks])	在一个(K,V)对的数据集上使用，返回一个(K,V)对的数据集，key 相同的值，都被使用指定的 reduce 函数聚合到一起。和 groupByKey 类似，任务的个数是可以通过第二个可选参数来配置的。
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	先按分区聚合再总的聚合，每次要跟初始值交流 例如：aggregateByKey(0)(_+_,_+_) 对 K/V 的 RDD 进行操作
sortByKey([ascending], [numTasks])	在一个(K,V)的 RDD 上调用，K 必须实现 Ordered 接口，

	返回一个按照 key 进行排序的(K,V)的 RDD
sortBy(func,[ascending], [numTasks])	与 sortByKey 类似，但是更灵活 第一个参数是根据什么排序 第二个是怎么排序，true 正序，false 倒序 第三个排序后分区数，默认与原 RDD 一样
join(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素对在一起的(K,(V,W))的 RDD，相当于内连接（求交集）
coGroup(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD
cartesian(otherDataset)	笛卡尔积
pipe(command, [envVars])	调用外部排序
coalesce(numPartitions)	重新分区，第一个参数是分区数，第二个参数是否 shuffle 默认 false，少分区变多分区 true，多分区变少分区 false
repartition(numPartitions)	重新分区，必须 shuffle，参数是要分多少区，少变多
repartitionAndSortWithinPartitions(partitioner)	重新分区+排序，比先分区再排序效率高，对 K/V 的 RDD 进行操作
foldByKey(zeroValue)(seqOp)	该函数用于 K/V 做折叠，合并处理，与 aggregate 类似第一个括号的参数应用于每个 V 值，第二括号函数是聚合例如：_+_
combineByKey	合并相同的 key 的值 rdd1.combineByKey(x => x, (a: Int, b: Int) => a + b, (m: Int, n: Int) => m + n)
partitionBy(partitioner)	对 RDD 进行分区，partitioner 是分区器 例如 new HashPartition(2)
cache	RDD 缓存，可以避免重复计算从而减少时间，区别：cache 内部调用了 persist 算子，cache 默认就一个缓存级别 MEMORY-ONLY ，而 persist 则可以选择缓存级别
persist	
subtract(rdd)	返回前 rdd 元素不在后 rdd 的 rdd
leftOuterJoin	leftOuterJoin 类似于 SQL 中的左外关联 left outer join，返回结果以前面的 RDD 为主，关联不上的记录为空。只能用于两个 RDD 之间的关联，如果要多个 RDD 关联，多关联几次即可。
rightOuterJoin	rightOuterJoin 类似于 SQL 中的有外关联 right outer join，返回结果以参数中的 RDD 为主，关联不上的记录为空。只能用于两个 RDD 之间的关联，如果要多个 RDD 关联，多关联几次即可
subtractByKey	substractByKey 和基本转换操作中的 subtract 类似只不过这里是针对 K 的，返回在主 RDD 中出现，并且不在 otherRDD 中出现的元素

总结：

Transformation 返回值还是一个 RDD。它使用了链式调用的设计模式，对一个 RDD 进行计算后，变换成另外一个 RDD，然后这个 RDD 又可以进行另外一次转换。这个过程是分布式的

6.3.2、Action

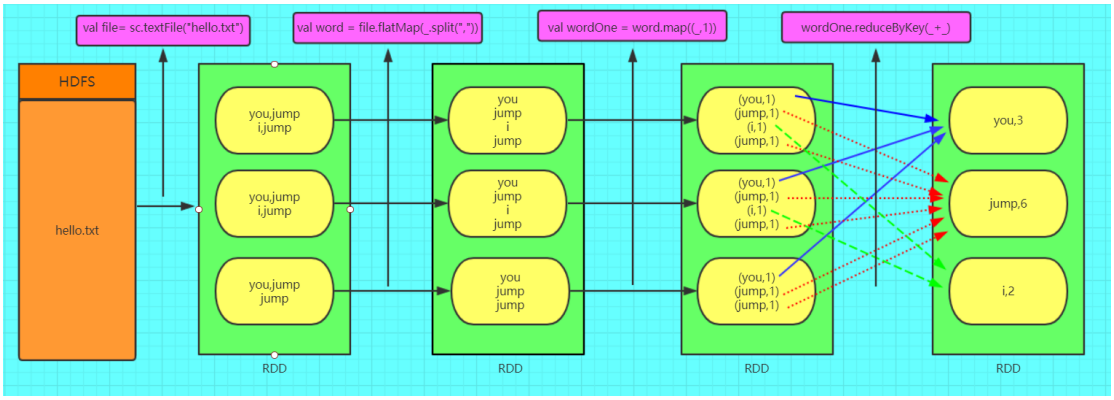
官网：http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

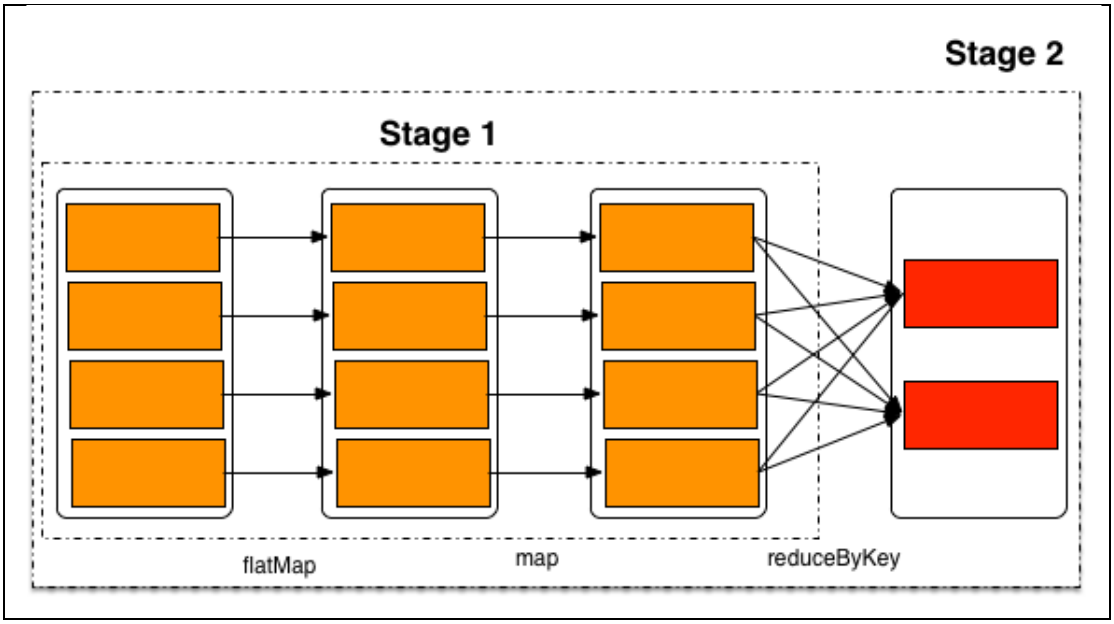
Action 算子	算子含义
reduce(func)	通过 func 函数聚集 RDD 中的所有元素，这个功能必须是可交换且可并联的
reduceByKeyLocally	def reduceByKeyLocally(func: (V, V) => V): Map[K, V] 该函数将 RDD[K,V]中每个 K 对应的 V 值根据映射函数来运算，运算结果映射到一个 Map[K,V]中，而不是 RDD[K,V]。
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
count()	返回 RDD 的元素个数
first()	返回 RDD 的第一个元素（类似于 take(1)）
take(n)	返回一个由数据集的前 n 个元素组成的数组
takeSample(withReplacement,num, [seed])	返回一个数组，该数组由从数据集中随机采样的 num 个元素组成，可以选择是否用随机数替换不足的部分，seed 用于指定随机数生成器种子
top	top 函数用于从 RDD 中，按照默认（降序）或者指定的排序规则，返回前 num 个元素
takeOrdered(n, [ordering])	takeOrdered 和 top 类似，只不过以和 top 相反的顺序返回元素
countByKey()	针对(K,V)类型的 RDD，返回一个(K,Int)的 map，表示每一个 key 对应的元素个数
foreach(func)	在数据集的每一个元素上，运行函数 func 进行更新。
foreachPartition	def foreachPartition(f: Iterator[T] => Unit): Unit 遍历每个 Partition
fold	def fold(zeroValue: T)(op: (T, T) => T): T fold 是 aggregate 的简化，将 aggregate 中的 seqOp 和 combOp 使用同一个函数 op
aggregate	def aggregate[U](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U)(implicit arg0: ClassTag[U]): U aggregate 用户聚合 RDD 中的元素，先使用 seqOp 将 RDD 中每个分区中的 T 类型元素聚合成 U 类型，再使用 combOp 将之前每个分区聚合后的 U 类型聚合成 U 类型，特别注意 seqOp 和 combOp 都会使用 zeroValue 的值，zeroValue 的类型为 U
lookup	针对 key-value 类型的 RDD 进行查找
saveAsTextFile(path)	将数据集的元素以 textfile 的形式保存到 HDFS 文件系统或者其他支持的文件系统，对于每个元素，Spark 将会调用 toString 方法，将它装换为文件中的文本
saveAsSequenceFile(path)	将数据集中的元素以 Hadoop sequencefile 的格式保存到指定的目录下，可以使 HDFS 或者其他 Hadoop 支持的文件系统
saveAsObjectFile(path)	saveAsObjectFile 用于将 RDD 中的元素序列化成对象，存储到文件中。对于 HDFS，默认采用 SequenceFile 保存

总结：

Action 返回值不是一个 RDD。它要么是一个 Scala 的普通集合，要么是一个值，要么是空，最终或返回到 Driver 程序，或把 RDD 写入到文件系统中

6.3.3、WordCount 中的 RDD





那么问题来了，请问在下面这一句标准的 wordcount 中到底产生了几个 RDD 呢？？？

sc.textFile("hdfs://myha01/wc/input/words.txt").flatMap(_.split("
")).map((_,1)).reduceByKey(_+_).collect

6.3.4、RDD 作业

启动 spark-shell

$SPARK_HOME/bin/spark-shell --master spark://hadoop02:7077

练习 1： //通过并行化生成 rdd

```
val rdd1 = sc.parallelize(List(5, 6, 4, 7, 3, 8, 2, 9, 1, 10))
//对 rdd1 里的每一个元素乘 2 然后排序
val rdd2 = rdd1.map(_ * 2).sortBy(x => x, true)
//过滤出大于等于十的元素
val rdd3 = rdd2.filter(_ >= 10)
//将元素以数组的方式在客户端显示
rdd3.collect
```

```
练习 2：
val rdd1 = sc.parallelize(Array("a b c", "d e f", "h i j"))
//将 rdd1 里面的每一个元素先切分在压平
val rdd2 = rdd1.flatMap(_.split(' '))
rdd2.collect
```

```
练习 3：
val rdd1 = sc.parallelize(List(5, 6, 4, 3))
val rdd2 = sc.parallelize(List(1, 2, 3, 4))
//求并集
val rdd3 = rdd1.union(rdd2)
//求交集
val rdd4 = rdd1.intersection(rdd2)
//去重
rdd3.distinct.collect
rdd4.collect
```

```
练习 4：
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)))
//求 jion
val rdd3 = rdd1.join(rdd2)
rdd3.collect
//求并集
val rdd4 = rdd1 union rdd2
//按 key 进行分组
rdd4.groupByKey
rdd4.collect
```

```
练习 5：
val rdd1 = sc.parallelize(List(("tom", 1), ("tom", 2), ("jerry", 3), ("kitty", 2)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)))
//cogroup
val rdd3 = rdd1.cogroup(rdd2)
//注意 cogroup 与 groupByKey 的区别
rdd3.collect
```

练习 6：

```
val rdd1 = sc.parallelize(List(1, 2, 3, 4, 5))
//reduce 聚合
val rdd2 = rdd1.reduce(_ + _)
rdd2.collect
```

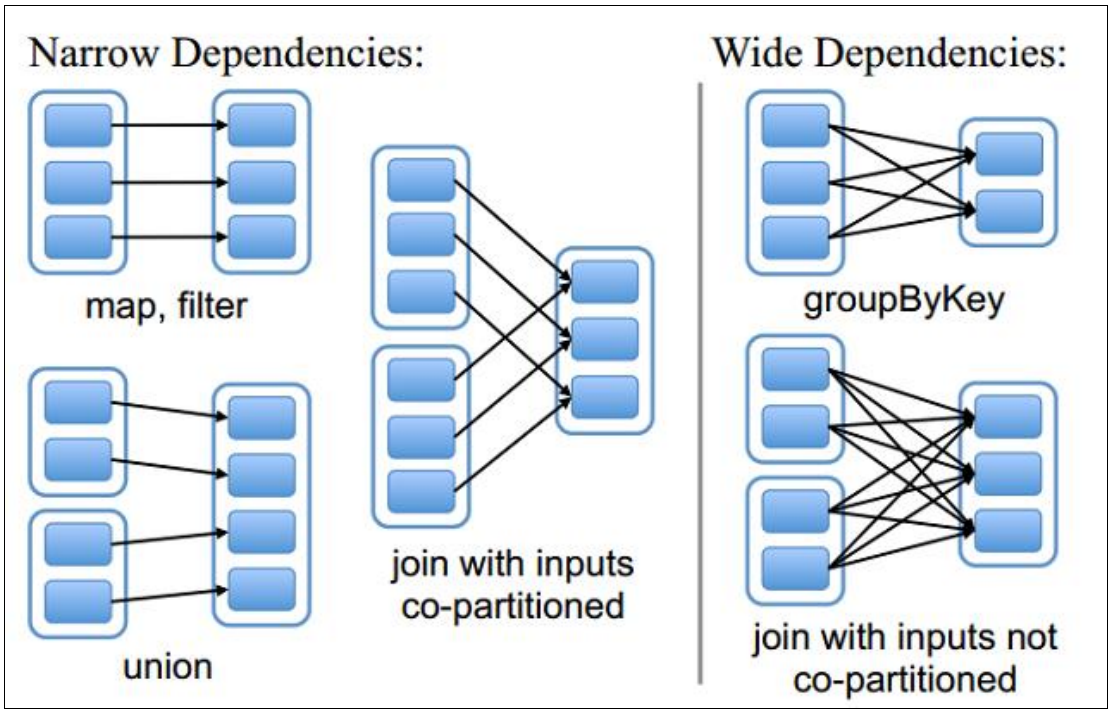
练习 7：

```
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2),    ("shuke", 1)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 3), ("shuke", 2), ("kitty", 5)))
val rdd3 = rdd1.union(rdd2)
//按 key 进行聚合
val rdd4 = rdd3.reduceByKey(_ + _)
rdd4.collect
//按 value 的降序排序
val rdd5 = rdd4.map(t => (t._2, t._1)).sortByKey(false).map(t => (t._2, t._1))
rdd5.collect
```

//想要了解更多，访问下面的地址
http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html

6.4、RDD 的依赖关系

RDD 和它依赖的父 RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）



6.4.1、窄依赖和宽依赖对比

窄依赖指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用
总结：窄依赖我们形象的比喻为独生子女，窄依赖的函数有：map, filter, union, join(父 RDD 是 hash-partitioned), mapPartitions, mapValues

宽依赖指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition
总结：窄依赖我们形象的比喻为超生，宽依赖的函数有：groupByKey、partitionBy、reduceByKey、sortByKey、join(父 RDD 不是 hash-partitioned)

6.4.2、窄依赖和宽依赖总结

在这里我们是从父 RDD 的 partition 被使用的个数来定义窄依赖和宽依赖，因此可以用一句话概括下：如果父 RDD 的一个 Partition 被子 RDD 的一个 Partition 所使用就是窄依赖，否则的话就是宽依赖。因为是确定的 partition 数量的依赖关系，所以 RDD 之间的依赖关系就是窄依赖；由此我们可以得出一个推论：即窄依赖不仅包含一对一的窄依赖，还包含一对固定个数的窄依赖。

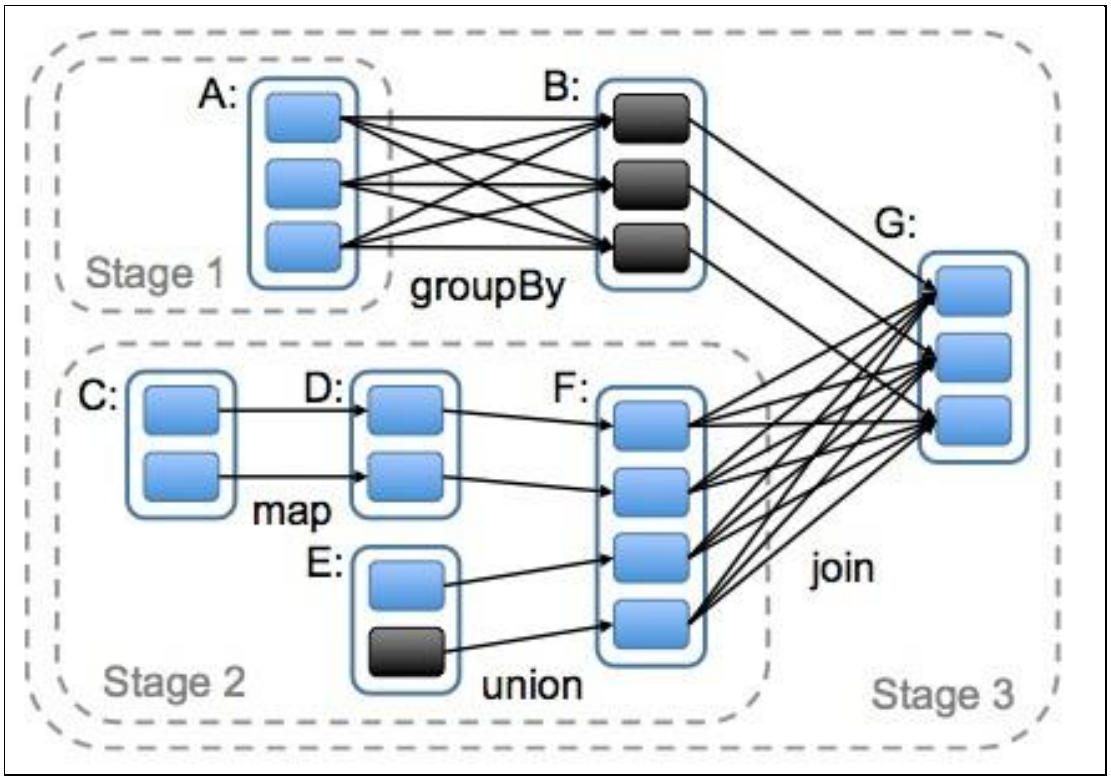
一对固定个数的窄依赖的理解：即子 RDD 的 partition 对父 RDD 依赖的 Partition 的数量不会随着 RDD 数据规模的改变而改变；换句话说，无论是有 100T 的数据量还是 1P 的数据量，在窄依赖中，子 RDD 所依赖的父 RDD 的 partition 的个数是确定的，而宽依赖是 shuffle 级别的，数据量越大，那么子 RDD 所依赖的父 RDD 的个数就越多，从而子 RDD 所依赖的父 RDD 的 partition 的个数也会变得越来越多。

6.4.3、Lineage

RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列 Lineage（即血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

6.5、DAG 生成

DAG(Directed Acyclic Graph)叫做有向无环图，原始的 RDD 通过一系列的转换就就形成了 DAG，根据 RDD 之间的依赖关系的不同将 DAG 划分成不同的 Stage，对于窄依赖，partition 的转换处理在 Stage 中完成计算。对于宽依赖，由于有 Shuffle 的存在，只能在 parent RDD 处理完成后，才能开始接下来的计算，因此宽依赖是划分 Stage 的依据。



在 spark 中，会根据 RDD 之间的依赖关系将 DAG 图（有向无环图）划分为不同的阶段，对于窄依赖，由于 partition 依赖关系的确定性，partition 的转换处理就可以在同一个线程里完成，窄依赖就被 spark 划分到同一个 stage 中，而对于宽依赖，只能等父 RDD shuffle 处理完成后，下一个 stage 才能开始接下来的计算。

因此 **spark 划分 stage 的整体思路是：从后往前推，遇到宽依赖就断开，划分为一个 stage；遇到窄依赖就将这个 RDD 加入该 stage 中**。因此在上图中 RDD C，RDD D，RDD E，RDD F 被构建在一个 stage 中，RDD A 被构建在一个单独的 Stage 中，而 RDD B 和 RDD G 又被构建在同一个 stage 中。

在 spark 中，Task 的类型分为 2 种：**ShuffleMapTask** 和 **ResultTask**

简单来说，**DAG 的最后一个阶段会为每个结果的 partition 生成一个 ResultTask，即每个 Stage 里面的 Task 的数量是由该 Stage 中最后一个 RDD 的 Partition 的数量所决定的！而其余所有阶段都会生成 ShuffleMapTask**；之所以称之为 ShuffleMapTask 是因为它需要将自己的计算结果通过 shuffle 到下一个 stage 中；也就是说上图中的 stage1 和 stage2 相当于 MapReduce 中的 Mapper，而 ResultTask 所代表的 stage3 就相当于 MapReduce 中的 reducer。

在之前动手操作了一个 WordCount 程序，因此可知，Hadoop 中 MapReduce 操作中的 Mapper 和 Reducer 在 spark 中的基本等量算子是 map 和 reduceByKey；不过区别在于：Hadoop 中的 MapReduce 天生就是排序的；而 reduceByKey 只是根据 Key 进行 reduce，但 spark 除了这两个算子还有其他的算子；因此从这个意义上来说，Spark 比 Hadoop 的计算算子更为丰富。

6.6、RDD 缓存

Spark 速度非常快的原因之一，就是在不同操作中可以在内存中持久化或缓存个数据集。当持久化某个 RDD 后，每一个节点都将把计算的分片结果保存在内存中，并在对此 RDD 或衍生出的 RDD 进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD 相关的持久化和缓存，是 Spark 最重要的特征之一。可以说，缓存是 Spark 构建迭代式算法和快速交互式查询的关键。

6.6.1、RDD 的缓存方式

RDD 通过 persist 方法或 cache 方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。

```
199   /**
200    * Persist this RDD with the default storage level (`MEMORY_ONLY`).
201    */
202   def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
203
204   /**
205    * Persist this RDD with the default storage level (`MEMORY_ONLY`).
206    */
207   def cache(): this.type = persist()
```

通过查看源码发现 cache 最终也是调用了 persist 方法，默认的存储级别都是仅在内存存储一份，Spark 的存储级别还有好多种，存储级别在 object StorageLevel 中定义的。

```
148 /**
149  * Various [[org.apache.spark.storage.StorageLevel]] defined and utility functions for creating
150  * new storage levels.
151  */
152 object StorageLevel {
153   val NONE = new StorageLevel(false, false, false, false)
154   val DISK_ONLY = new StorageLevel(true, false, false, false)
155   val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
156   val MEMORY_ONLY = new StorageLevel(false, true, false, true)
157   val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
158   val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
159   val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
160   val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
161   val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
162   val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
163   val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
164   val OFF_HEAP = new StorageLevel(true, true, true, false, 1)
```

缓存有可能丢失，或者存储存储于内存的数据由于内存不足而被删除，RDD 的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于 RDD 的一系列转换，丢失的数据会被重算，由于 RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

7、Shared Variables（共享变量）

在 Spark 程序中，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。这些变量会被复制到每

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。通常跨任务的读写变量是低效的，但是，Spark 还是为两种常见的使用模式提供了两种有限的共享变量：**广播变（Broadcast Variable）**和**累加器（Accumulator）**

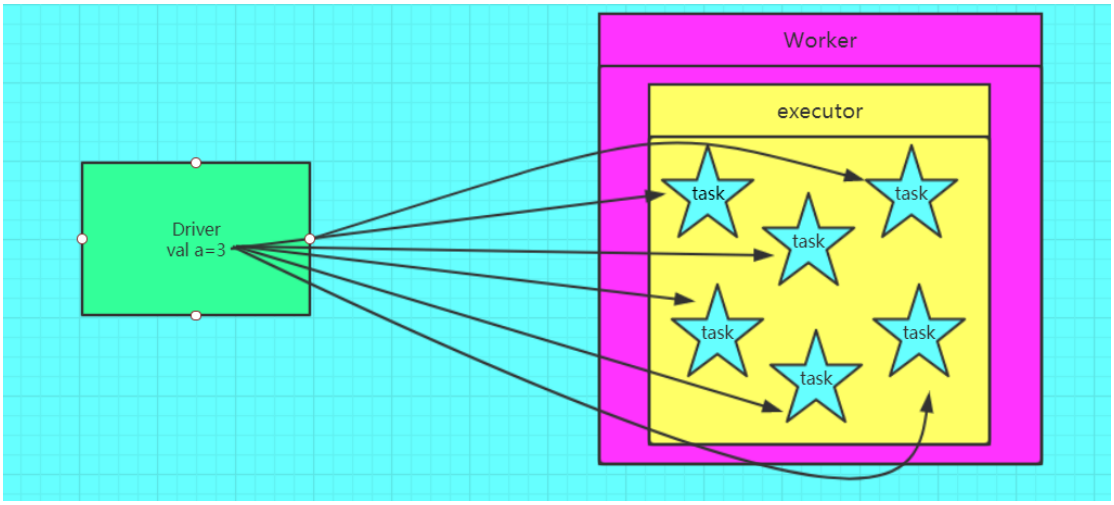
官网：http://spark.apache.org/docs/latest/rdd-programming-guide.html#shared-variables

7.1、Broadcast Variables（广播变量）

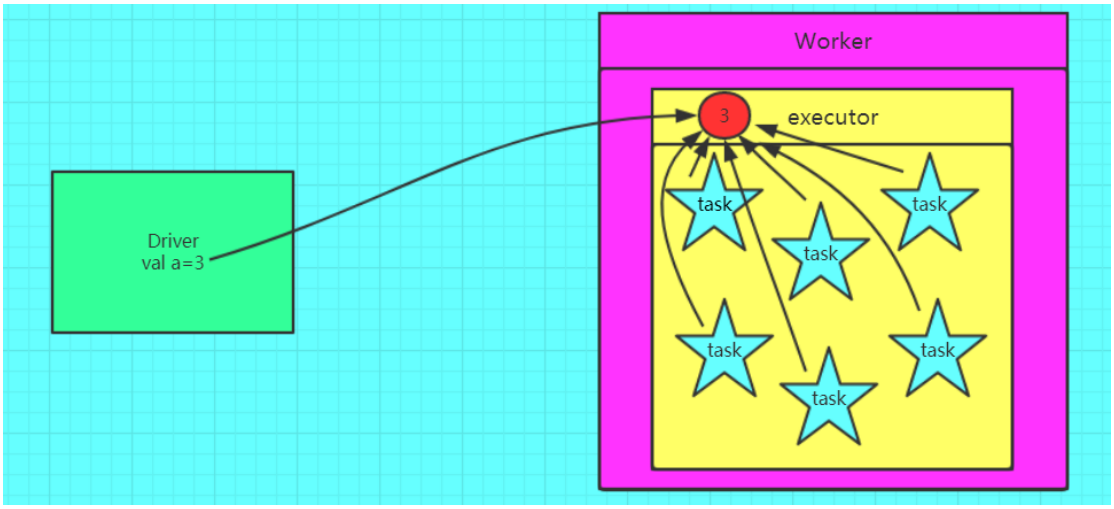
7.1.1、为什么要定义广播变量

如果我们要在分布式计算里面分发大对象，例如：字典，集合，黑白名单等，这个都会由 Driver 端进行分发，一般来讲，如果这个变量不是广播变量，那么每个 task 就会分发一份，这在 task 数目十分多的情况下 Driver 的带宽会成为系统的瓶颈，而且会大量消耗 task 服务器上的资源，如果将这个变量声明为广播变量，那么知识每个 executor 拥有一份，这个 executor 启动的 task 会共享这个变量，节省了通信的成本和服务器的资源。

没有使用广播变量：



使用了广播变量之后：



7.1.2、如何定义和还原一个广播变量

定义：

```
val a = 3
val broadcast = sc.broadcast(a)
```

还原：

```
val c = broadcast.value
```

注意：变量一旦被定义为一个广播变量，那么这个变量只能读，不能修改

7.1.3、注意事项

1、能不能将一个 RDD 使用广播变量广播出去？
不能，因为 RDD 是不存储数据的。可以将 RDD 的结果广播出去。

2、广播变量只能在 Driver 端定义，不能在 Executor 端定义。

3、在 Driver 端可以修改广播变量的值，在 Executor 端无法修改广播变量的值。

4、如果 executor 端用到了 Driver 的变量，如果不使用广播变量在 Executor 有多少 task 就有多少 Driver 端的变量副本。

5、如果 Executor 端用到了 Driver 的变量，如果使用广播变量在每个 Executor 中都只有一份 Driver 端的变量副本。

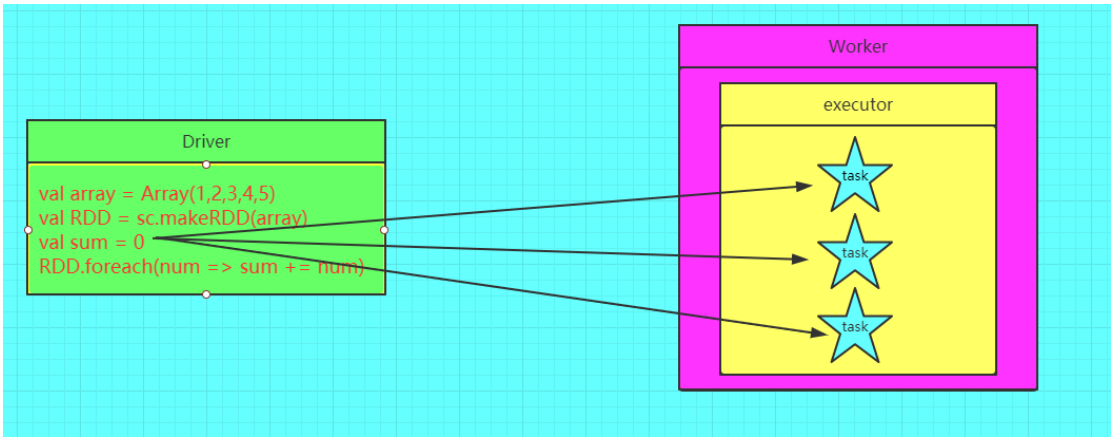
7.2、Accumulators（累加器）

7.2.1、为什么要定义累加器

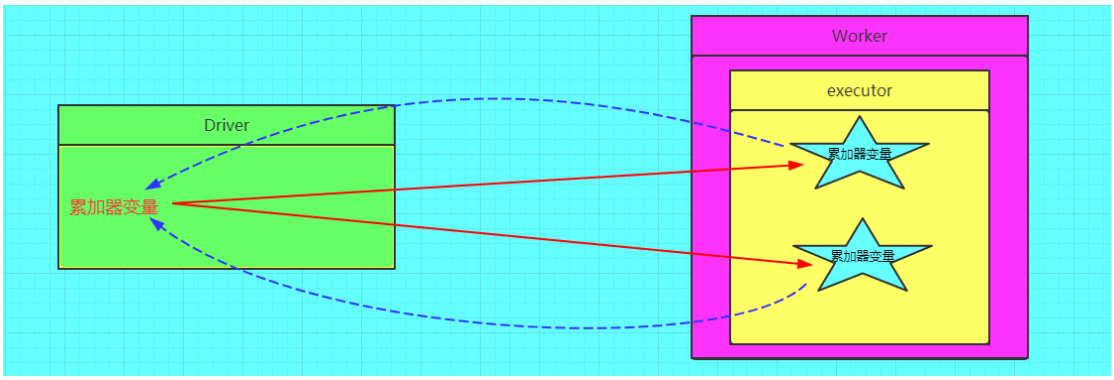
在 Spark 应用程序中，我们经常会有这样的需求，如异常监控，调试，记录符合某特性的数据的数目，这种需求都需要用到计数器，如果一个变量不被声明为一个累加器，那么它将在被改变时不会在 driver 端进行全局汇总，即在分布式运行时每个 task 运行的只是原始变量的一个副本，并不能改变原始变量的值，但是当这个变量被声明为累加器后，该变量就会有分布式计数的功能。

7.2.2、图解累加器

错误的图解：



正确的图解：



7.2.3、如果定义和还原一个累加器

定义累加器：

```
val a = sc.longAccumulator(0)
```

还原累加器：

```
val b = a.value
```

7.2.4、注意事项

1、累加器在 Driver 端定义赋初始值，累加器只能在 Driver 端读取最后的值，在 Excutor 端更新。

2、累加器不是一个调优的操作，因为如果不这样做，结果是错的