1. RDD

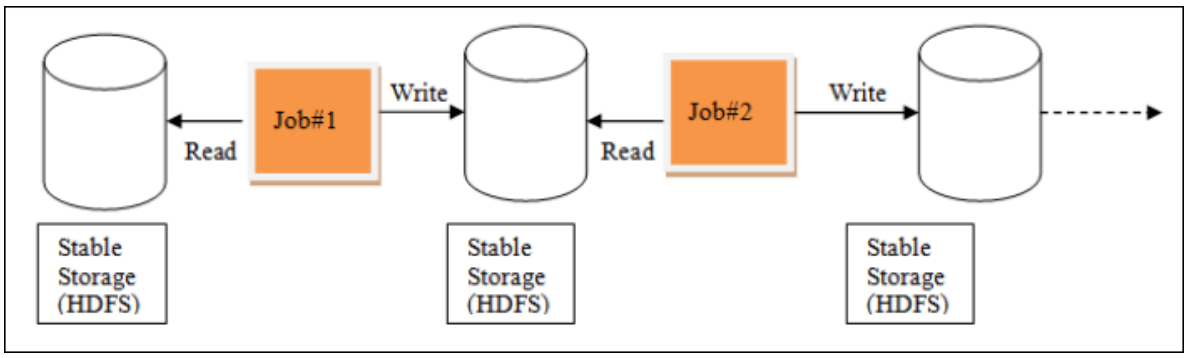
1.1. RDD概述

RDD 是 Spark 的基石，是实现 Spark 数据处理的核心抽象。那么 RDD 为什么会产生呢?

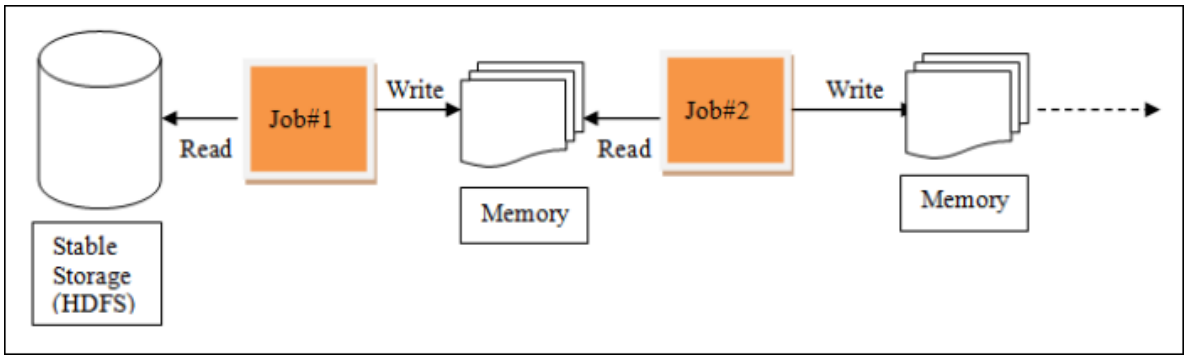
Hadoop 的 MapReduce是一种基于数据集的工作模式，面向数据，这种工作模式一般是从存储上加载数据集，然后操作数据集，最后写入物理存储设备。数据更多面临的是一次性处理。

MapReduce 的这种方式对数据领域两种常见的操作不是很高效。第一种是**迭代式的算法**。比如机器学习中 ALS、凸优化梯度下降等。这些都需要基于数据集或者数据集的衍生数据反复查询反复操作。MapReduce 这种模式不太合适，即使多 MapReduce 串行处理，性能和时间也是一个问题。数据的共享依赖于磁盘。另外一种是 **交互式数据挖掘**，MapReduce 显然不擅长。

MapReduce中的迭代:



Spark中的迭代：



我们需要一个效率非常快，且能够支持迭代计算和有效数据共享的模型，Spark 应运而生。RDD 是基于工作集的工作模式，更多的是面向工作流。

但是无论是 MapReduce 还是 RDD 都应该具有类似 **位置感知、容错和负载均衡** 等特性。

1.2. 什么是RDD

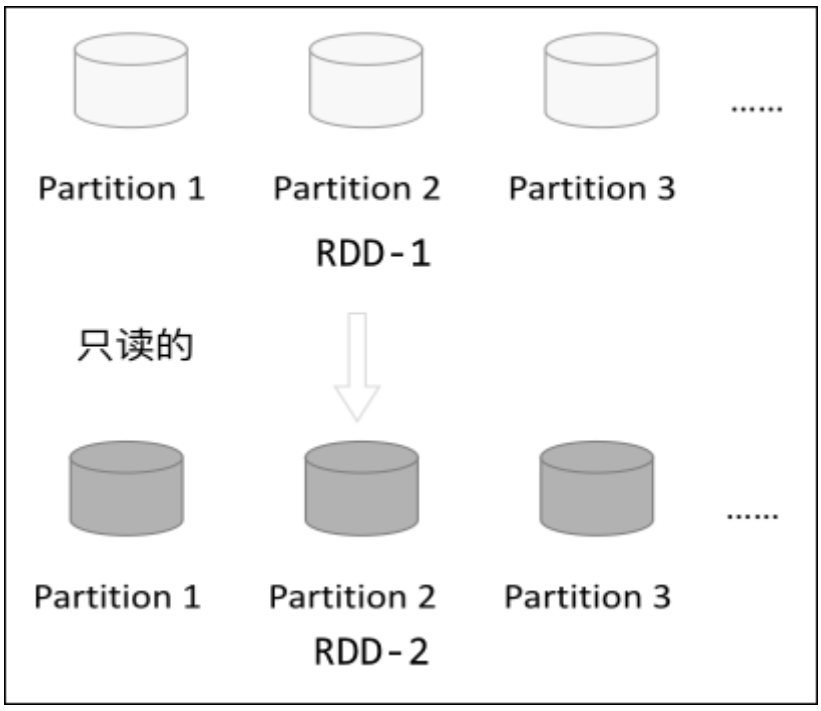
RDD（Resilient Distributed Dataset）叫做分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。在 Spark 中，对数据的所有操作不外乎创建 RDD、转化已有 RDD 以及调用RDD操作进行求值。每个 RDD 都被分为多个分区，这些分区运行在集群中的不同节点上。RDD可以包含Python、Java、Scala中任意类型的对象，甚至可以包含用户自定义的对象。**RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。**RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

RDD 支持两种操作：转化Transformation操作 和 行动Action操作。RDD 的Transformation操作是返回一个新的 RDD 的操作，比如 map() 和 filter()，而Action操作则是向驱动器程序返回结果或把结果写入外部系统的操作。比如 count() 和 first()。

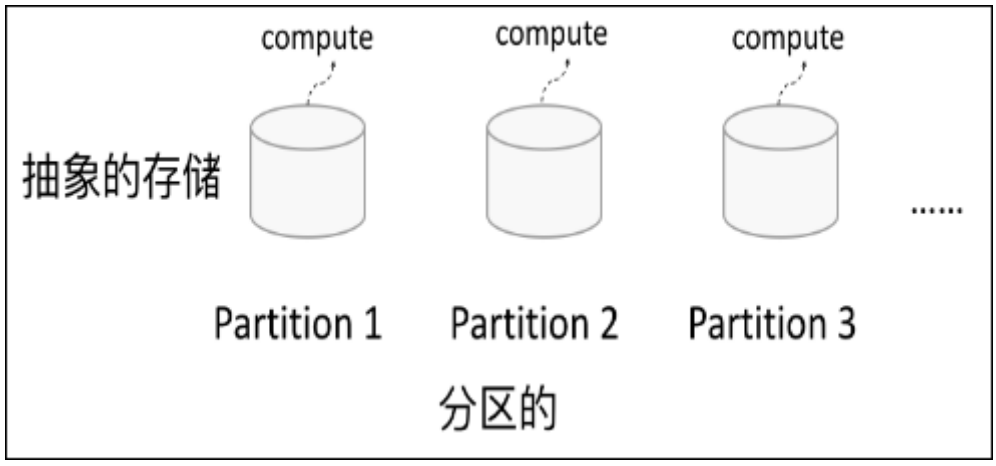
Spark 采用惰性计算模式，RDD只有第一次在一个行动操作中用到时，才会真正计算。Spark可以优化整个计算过程。默认情况下，Spark的RDD会在你每次对它们进行行动操作时重新计算。如果想在多个行动操作中重用同一个RDD，可以使用 RDD.persist() 让 Spark 把这个 RDD 缓存下来。

可以从三个方面来理解：

1、只读数据集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。RDD是只读的，要想改变RDD中的数据，只能在现有的RDD基础上创建新的 RDD。由一个RDD转换到另一个RDD，可以通过丰富的操作算子实现，不再像MapReduce那样只能写 map 和 reduce 了



2、分布式Distributed/分区：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。RDD 逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个 compute 函数得到每个分区的数据。如果 RDD 是通过已有的文件系统构建，则 compute 函数是读取指定文件系统中的数据，如果 RDD 是通过其他 RDD 转换而来，则 compute 函数是执行转换逻辑将其他 RDD 的数据进行转换。



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）并行计算单元的划分结构，也就是分区的数量。

Spark的RDD的弹性：

```
存储的弹性：内存和磁盘的自动切换
容错的弹性：数据丢失可以自动恢复
计算的弹性：计算出错重试机制
分片的弹性：根据需要重新分片
```

```
1、自动进行内存和磁盘数据存储的切换
Spark优先把数据放到内存中，如果内存放不下，就会放到磁盘里面，程序进行自动的存储切换。

2、基于血统的高效容错机制
```

在RDD进行转换和动作的时候，会形成RDD的Lineage依赖链，当某一个RDD失效的时候，可以通过重新计算上游的RDD来重新生成丢失的RDD数据。

3、Task如果失败会自动进行特定次数的重试
RDD的计算任务如果运行失败，会自动进行任务的重新计算，默认次数是4次。

4、Stage如果失败会自动进行特定次数的重试
如果Job某个Stage阶段计算失败，框架也会自动进行任务的重新计算，默认次数也是4次。

5、Checkpoint和Persist可主动或被动触发
RDD可以通过Persist持久化将RDD缓存到内存或者磁盘，当再次用到该RDD时直接读取就行。也可以将RDD进行检查点，检查点会将数据存储在HDFS中，该RDD的所有父RDD依赖都会被移除。

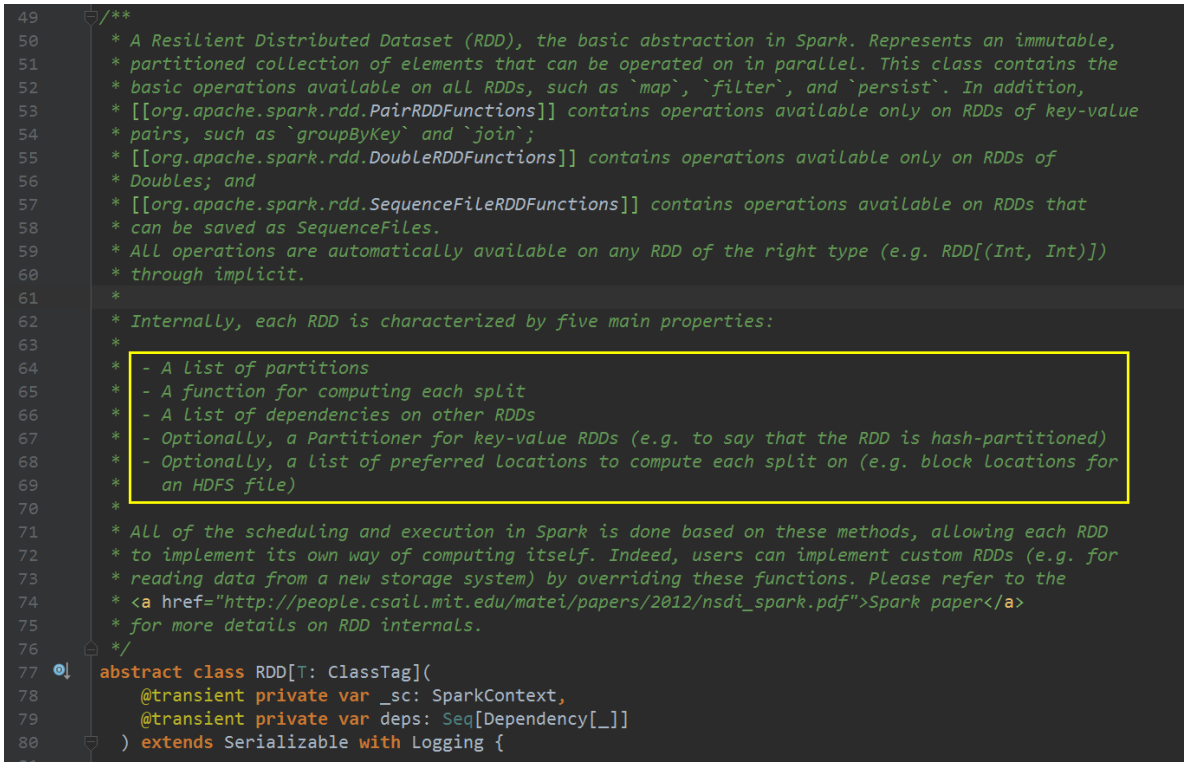
6、数据调度弹性
Spark把这个Job执行模型抽象为通用的有向无环图DAG，可以将多Stage的任务串联或并行执行，调度引擎自动处理Stage的失败以及Task的失败。

7、数据分片的高度弹性
可以根据业务的特征，动态调整数据分片的个数，提升整体的应用执行效率。

总结：

RDD全称叫做弹性分布式数据集(Resilient Distributed Datasets)，它是一种分布式的内存抽象，表示一个只读的记录分区的集合，它只能通过其他RDD转换而创建，为此，RDD支持丰富的转换操作(如map, join, filter, groupby等)，通过这种转换操作，新的RDD则包含了如何从其他RDDs衍生所必需的信息，所以说RDDs之间是有依赖关系的。基于RDDs之间的依赖，RDDs会形成一个有向无环图DAG，该DAG描述了整个流式计算的流程，实际执行的时候，RDD是通过血缘关系(Lineage)一气呵成的，即使出现数据分区丢失，也可以通过血缘关系重建分区，总结起来，基于RDD的流式计算任务可描述为：从稳定的物理存储(如分布式文件系统)中加载记录，记录被传入由一组确定性操作构成的DAG，然后写回稳定存储。另外RDD 还可以将数据集缓存到内存中，使得在多个操作之间可以重用数据集，基于这个特点可以很方便地构建迭代型应用(图计算、机器学习等)或者交互式数据分析应用。可以说Spark 最初也就是实现 RDD 的一个分布式系统，后面通过不断发展壮大成为现在较为完善的大数据生态系统，简单来讲，Spark-RDD 的关系类似于 Hadoop-MapReduce 关系。

1.3. 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。如果没有设置的话，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

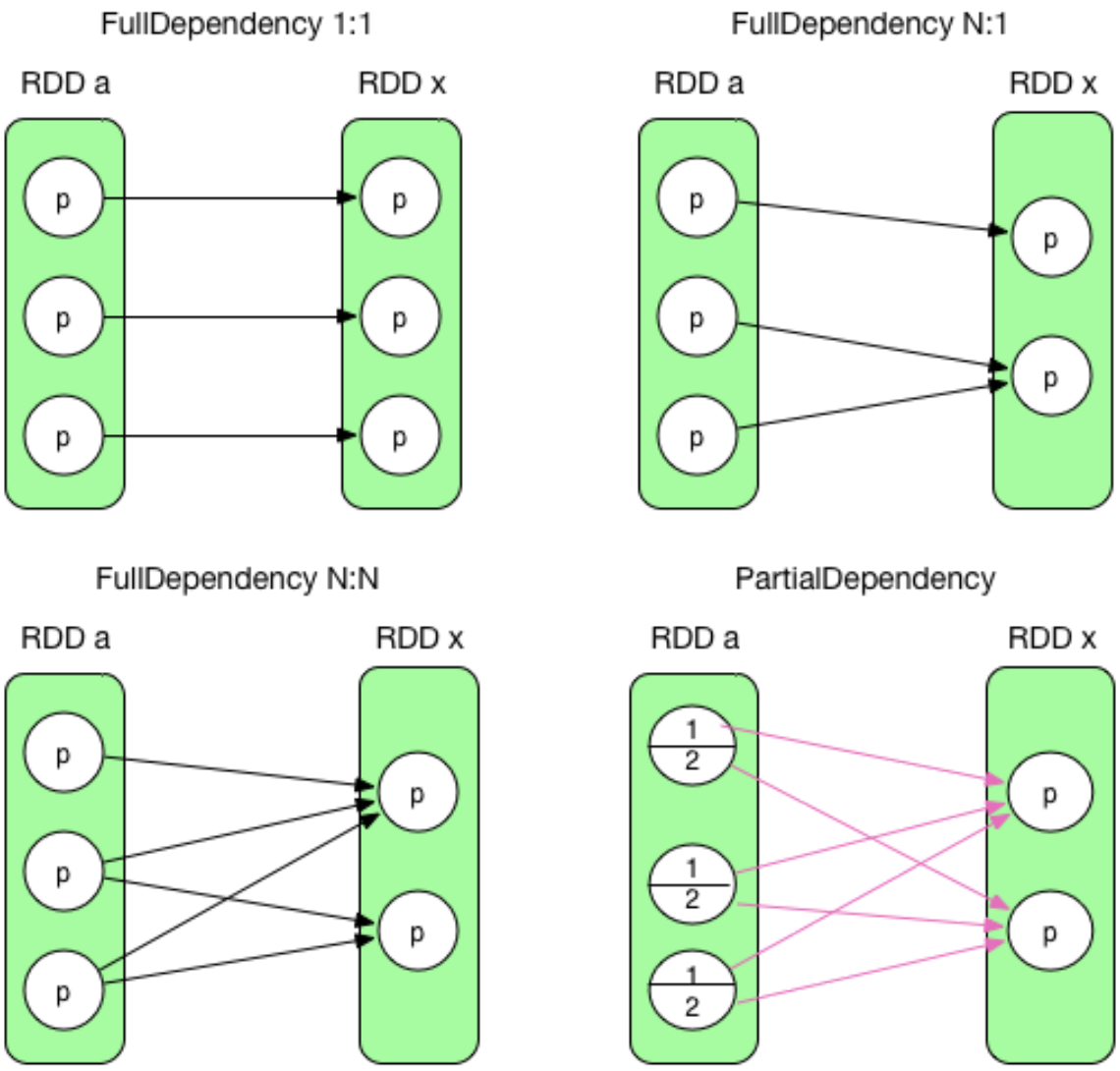
分区处理函数compute
1、每个RDD都会实现compute，用于对分区进行计算；
2、compute函数会对迭代器进行复合，不需要保存每次计算结果；
3、该方法负责接收parent RDDs或者data block流入的records并进行计算，然后输出加工后的records。

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。

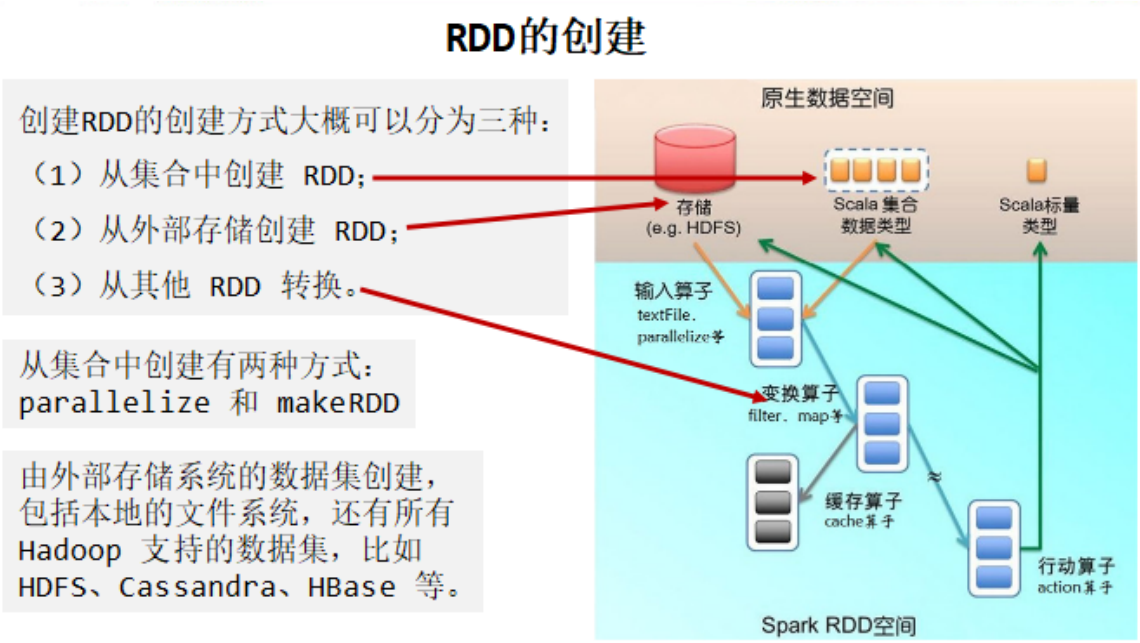
在代码中的表现：

```
// 由子类实现以计算给定分区
def compute(split: Partition, context: TaskContext): Iterator[T]

// 获取所有分区
protected def getPartitions: Array[Partition]

// 获取所有依赖关系
protected def getDependencies: Seq[Dependency[_]] = deps

// 获取优先位置列表
protected def getPreferredLocations(split: Partition): Seq[String] = Nil

// 分区器 由子类重写以指定它们的分区方式
@transient val partitioner: Option[Partition] = None
```

1.4. 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、从其他RDD转化来
```

1.5. RDD的编程API

官网：

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

RDD的操作算子包括两类，一类叫做transformations，它是用来将RDD进行转化，构建RDD 的血缘关系；另一类叫做actions，它是用来触发RDD的计算，得到RDD的相关计算结果或者将RDD保存的文件系统中。

下图是RDD所支持的操作算子列表。

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

1.5.1. 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
subtract(otherDataset**)**	返回前rdd元素不在后rdd的rdd
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
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

转换	含义
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)
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
pipe(command, [envVars])	调用外部命令
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
partitionBy(partitioner)	对RDD进行分区，partitioner是分区器例如new HashPartition(2)
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
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	
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,Iterable))类型的RDD
cartesian(otherDataset)	笛卡尔积

转换	含义
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

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。

1.5.2. Action

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

常用的 Action：

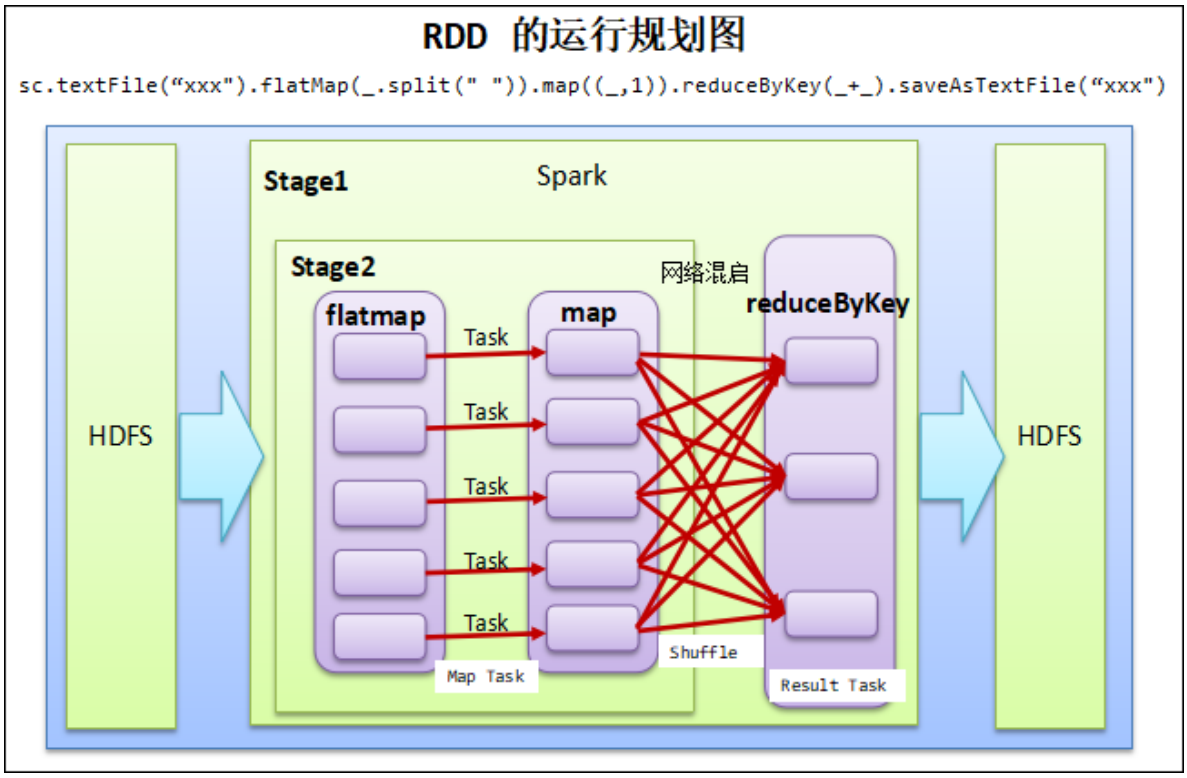
Action算子	算子含义
reduce(func)	通过func函数聚集RDD中的所有元素，这个功能必须是可交换且可并联的
reduceByKeyLocally	def reduceByKeyLocally(func: (V, V) => V): Map[K, V] 该函数将RDD[K,V]中每个K对应的V值根据映射函数来运算，运算结果映射到一个Map[K,V]中，而不是RDD[K,V]。
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
count()	返回RDD的元素个数
first()	返回RDD的第一个元素（类似于take(1)）
take(n)	返回一个由数据集的前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 aggregateU(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

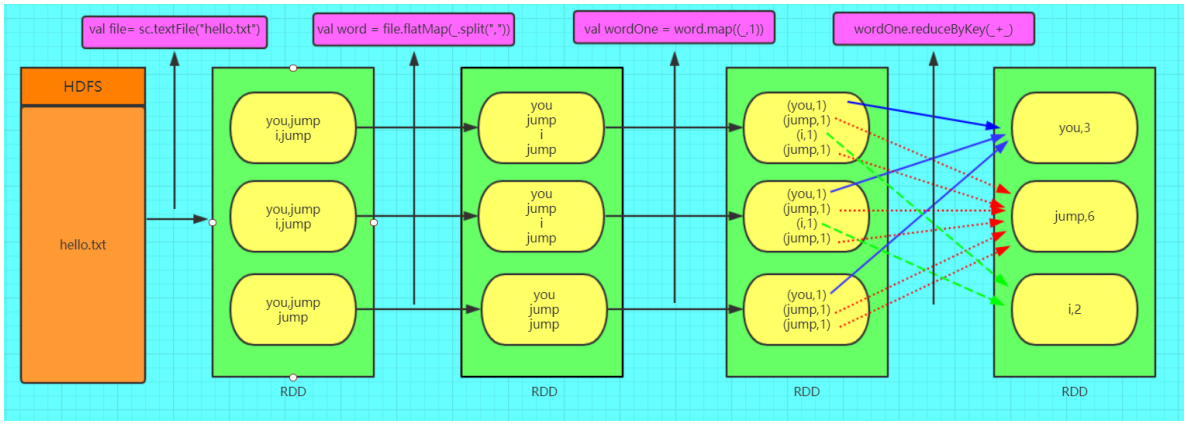
Action算子	算子含义
saveAsObjectFile(path)	saveAsObjectFile用于将RDD中的元素序列化成对象，存储到文件中。对于HDFS，默认采用SequenceFile保存

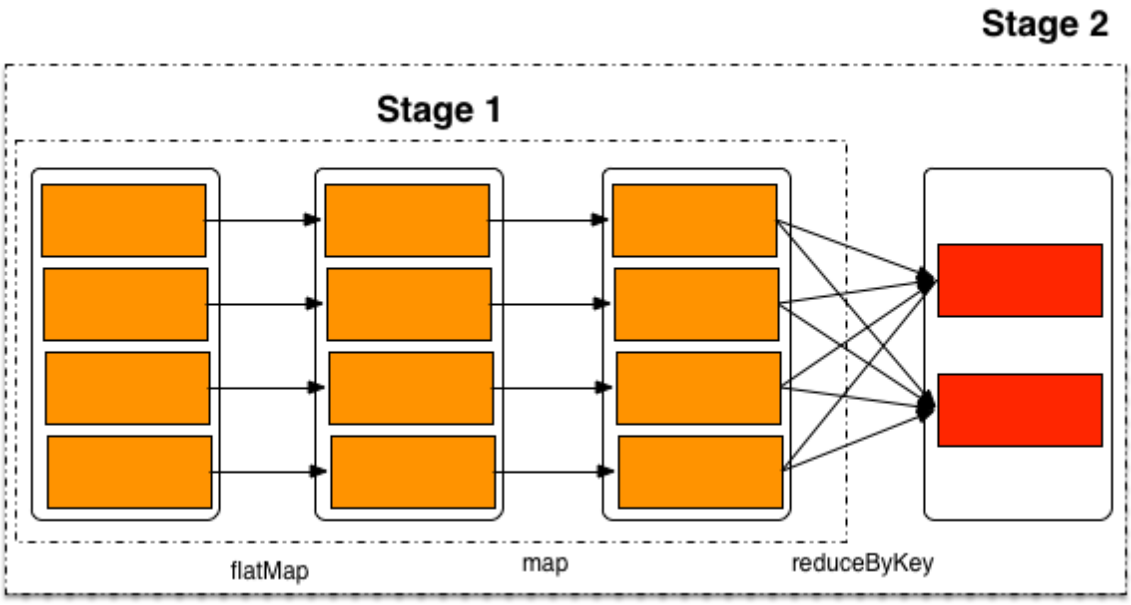
总结：

> 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

1.6. 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://hadoop277ha/wc/input/words.txt").flatMap(_.split("
")).map((_,1)).reduceByKey(_+_).collect
```

1.7. RDD的算子练习

启动spark-shell

```
$SPARK_HOME/bin/spark-shell --master spark://bigdata02:7077
```

练习1：map和filter

```
//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：flatMap

```
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)
//求差集
val rdd5 = rdd1.subtract(rdd2)
//去重
rdd3.distinct.collect
rdd4.collect
```

练习4:

```
val rdd1 = sc.parallelize(List(("huangbo", 1), ("xuzheng", 3), ("wangbaoqiang",
2)))
val rdd2 = sc.parallelize(List(("liuyifei", 2), ("liutao", 1), ("liushishi",
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(("huangbo", 1), ("xuzheng", 2), ("shenteng", 3),
("shenteng", 2)))
val rdd2 = sc.parallelize(List(("huangbo", 33), ("huangbo", 44), ("xuzheng",
11), ("shenteng", 22)))
//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
```

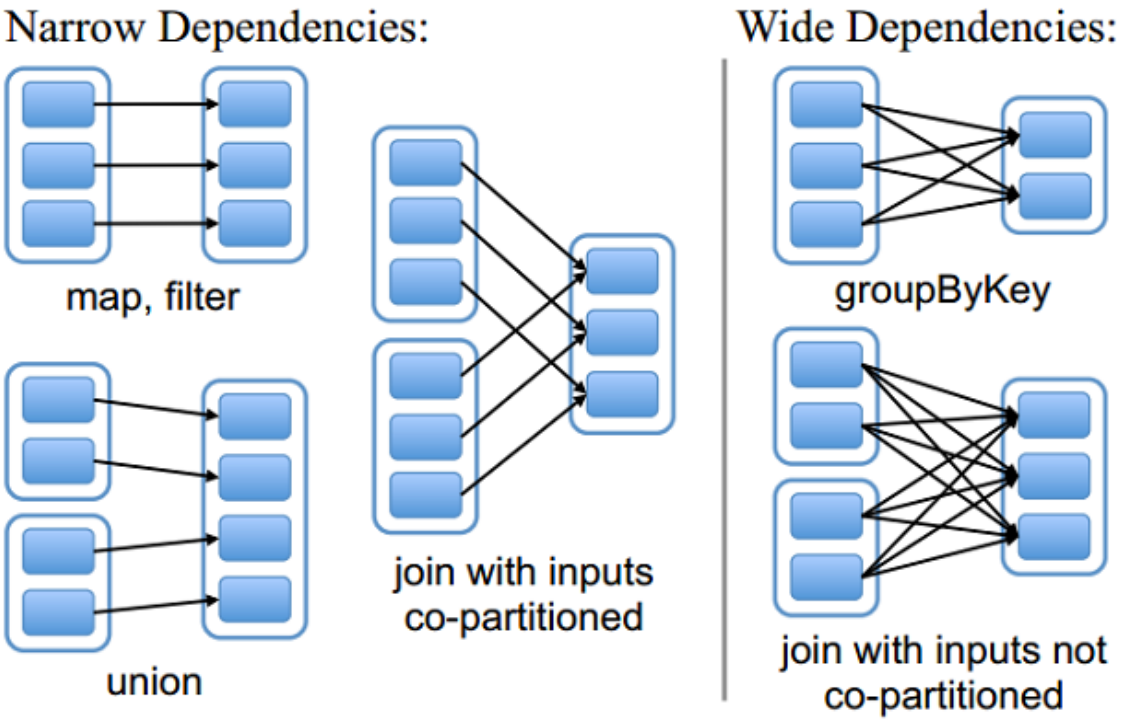
练习7:

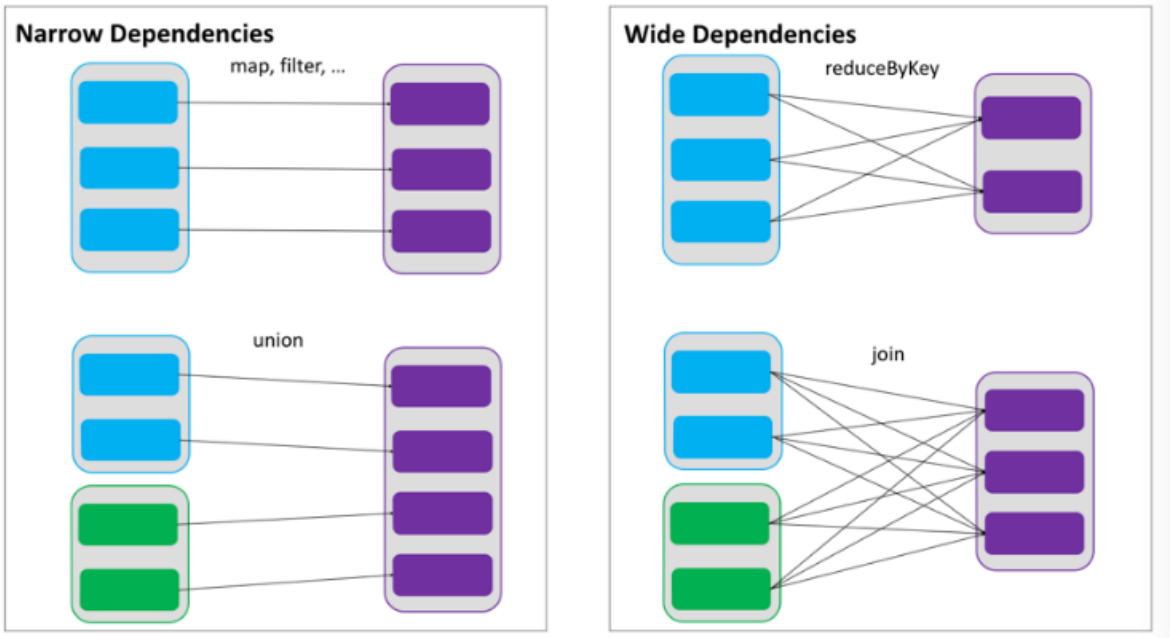
```
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2),
("shuke", 1)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 3), ("shuke", 2), ("kitty",
5)))
val rdd3 = rdd1.union(rdd2)
//按key进行聚合
val rdd4 = rdd3.reduceByKey(_ + _)
rdd4.collect
//按value的降序排序
val rdd5 = rdd4.map(t =>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

1.8. RDD的依赖关系

RDDs通过操作算子进行转换，转换得到的新RDD包含了从其他RDDs衍生所必需的信息，RDDs之间维护着这种血缘关系，也称之为依赖。如下图所示，依赖包括两种，一种是窄依赖(narrow dependency)，RDDs之间分区是一一对应的，另一种是宽依赖(wide dependency)，下游RDD的每个分区与上游 RDD (也称之为父RDD)的每个分区都有关，是多对多的关系。





RDD之间依赖关系

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)

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。

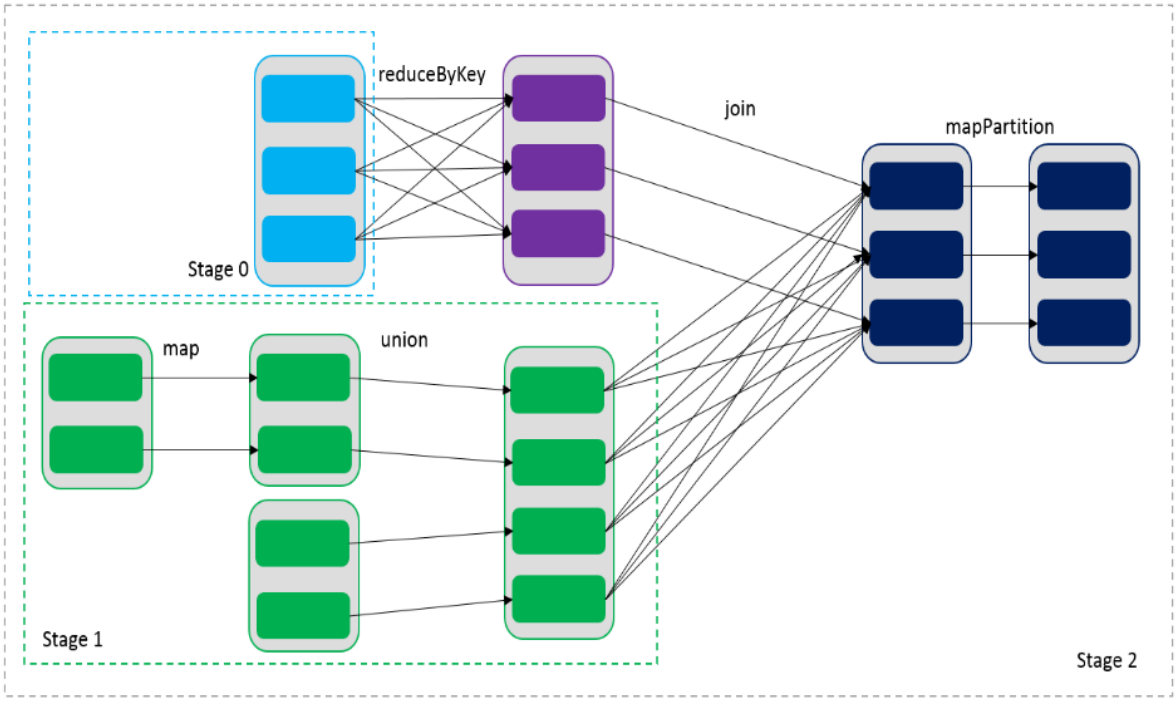
1.9. Lineage

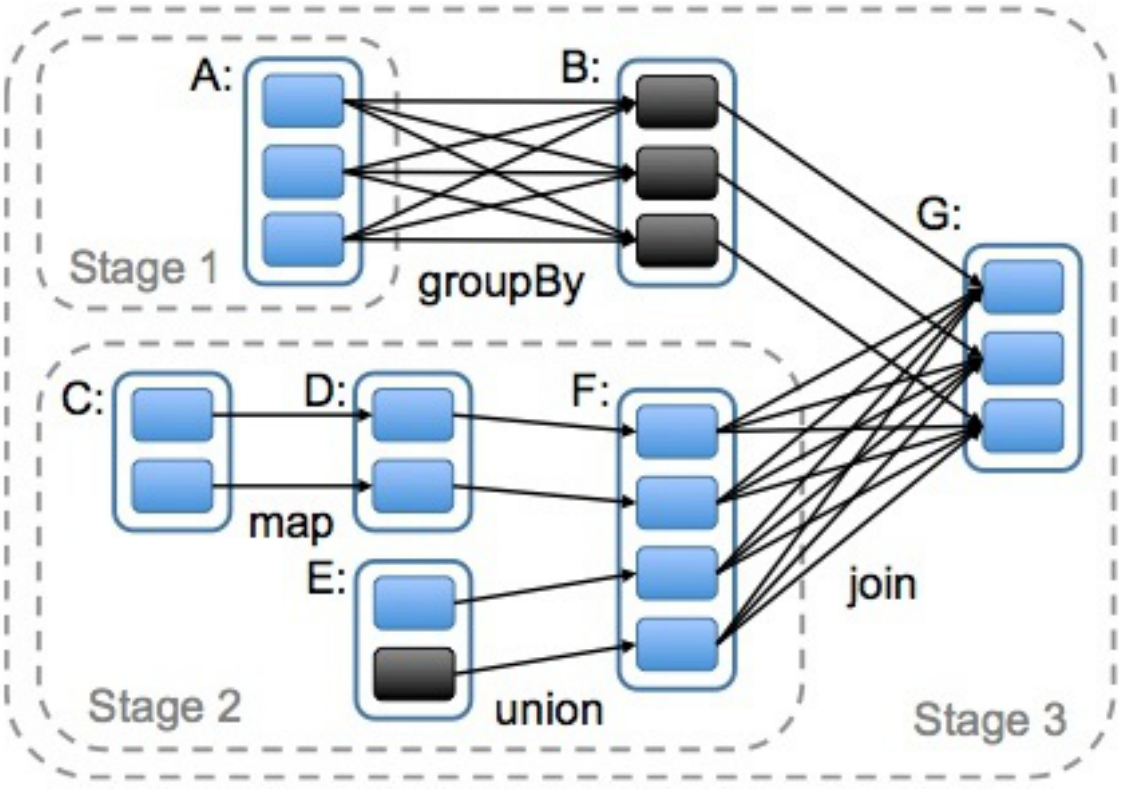
RDD只支持粗粒度转换，即在大量记录上执行的单个操作。将创建RDD的一系列Lineage（即血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

1.10. 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。

对于窄依赖：由于分区的依赖关系是确定的，其转换操作可以在同一个线程执行，所以可以划分到同一个执行阶段；
对于宽依赖：由于Shuffle的存在，只能在父RDD(s)被Shuffle处理完成后，才能开始接下来的计算，因此遇到宽依赖就需要重新划分阶段。





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。**

首先，窄依赖允许在一个集群节点上以流水线的方式（pipeline）对父分区数据进行计算，例如先执行map操作，然后执行filter操作。而宽依赖则需要计算好所有父分区的数据，然后再在节点之间进行Shuffle，这与MapReduce类似。窄依赖能够更有效地进行数据恢复，因为只需重新对丢失分区的父分区进行计算，且不同节点之间可以并行计算；而对于宽依赖而言，如果数据丢失，则需要对所有父分区数据进行计算并再次Shuffle。

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。

1.11. RDD缓存

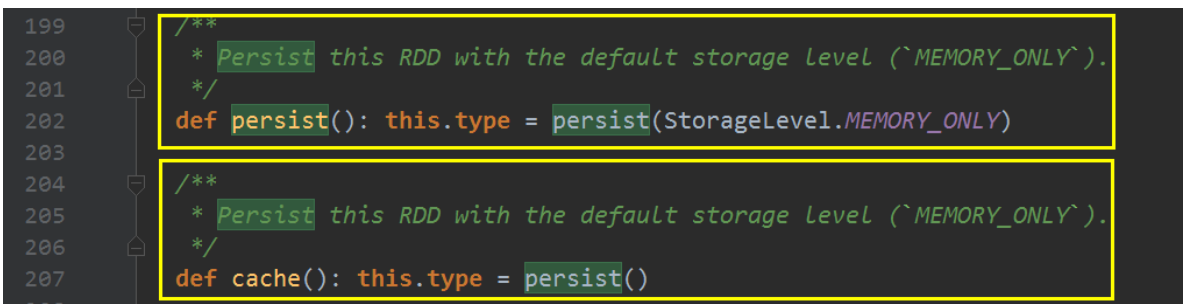
Spark速度非常快的原因之一，就是在不同操作中可以在内存中持久化或缓存个数据集。当持久化某个RDD后，每一个节点都将把计算的分片结果保存在内存中，并在对此RDD或衍生出的RDD进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD相关的持久化和缓存，是Spark最重要的特征之一。可以说，缓存是Spark构建迭代式算法和快速交互式查询的关键。

如果一个有持久化数据的节点发生故障，Spark会在需要用到缓存的数据时重算丢失的数据分区。如果希望节点故障的情况不会拖累我们的执行速度，也可以把数据备份到多个节点上。

持久化也是懒执行的，持久化有两个操作：persist(StorageLevel)，另外一个是 cache，cache 就相当于MEMORY_ONLY 的 persist。

1.11.1. 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。



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。

```
/**
 * Various [[org.apache.spark.storage.StorageLevel]] defined and utility functions for creating
 * new storage levels.
 */
object StorageLevel {
  val NONE = new StorageLevel(false, false, false, false)
  val DISK_ONLY = new StorageLevel(true, false, false, false)
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)
  val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
  val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
  val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
  val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
  val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
  val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
  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。

另外关注：unpersist 方法，用来取消已经 cache 或者 persist 的 RDD

1.12. Checkpoint

Spark 中对于数据的保存除了持久化操作之外，还提供了一种检查点机制，检查点（本质是通过将RDD写入Disk做检查点）是为了通过 lineage 做容错的辅助，lineage 过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有节点出现问题而丢失分区，从做检查点的 RDD 开始重做 Lineage，就会减少开销。检查点通过将数据写入到 HDFS 文件系统实现了 RDD 的检查点功能。

cache 和 checkpoint 是有显著区别的，缓存把 RDD 计算出来然后放在内存中，但是 RDD 的依赖链（相当于数据库中的 redo 日志），也不能丢掉，当某个worker上某个 executor 宕了，上面 cache 的 RDD 就会丢掉，需要通过依赖链重放计算出来，不同的是，checkpoint是把 RDD 保存在 HDFS 中，是多副本可靠存储，所以依赖链就可以丢掉了，就斩断了依赖链，是通过复制实现的高容错。

如果存在以下场景，则比较适合使用检查点机制：

```
1、DAG中的Lineage过长，如果重算，则开销太大（如在PageRank中）。
2、在宽依赖上做checkpoint获得的收益更大。
```

为当前 RDD 设置检查点。该函数将会创建一个二进制的文件，并存储到 checkpoint 目录中，该目录是用 `sparkContext.setCheckpointDir()` 设置的。在 checkpoint 的过程中，该 RDD 的所有依赖于父 RDD 中的信息将全部被移出。对 RDD 进行 checkpoint 操作并不会马上被执行，必须执行 Action 操作才能触发。

2. 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）**

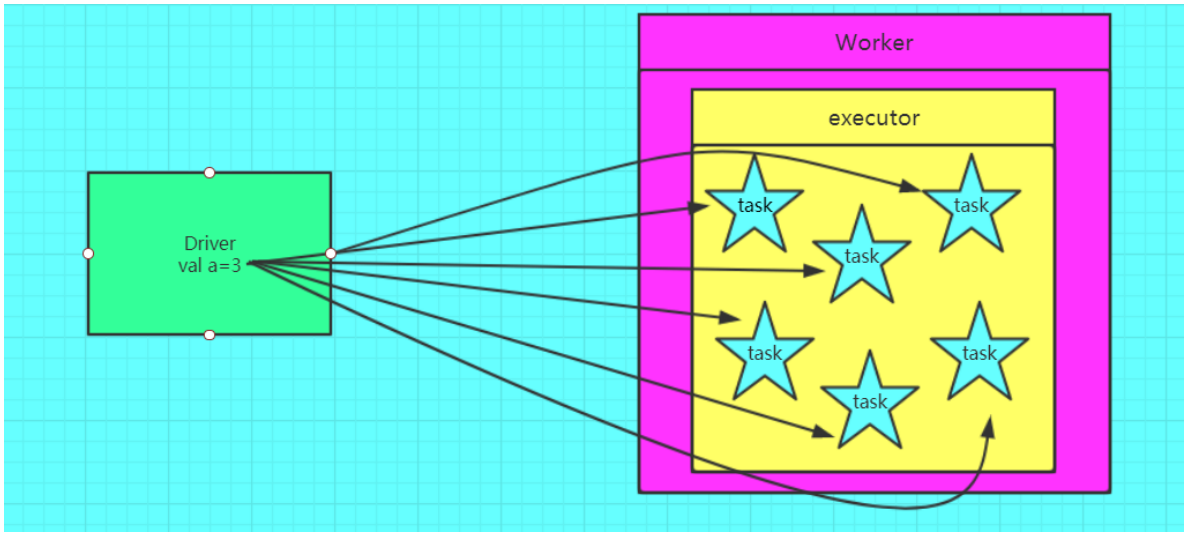
官网：http://spark.apache.org/docs/latest/rdd-programming-guide.html#shared-variables

2.1. Broadcast Variables（广播变量）

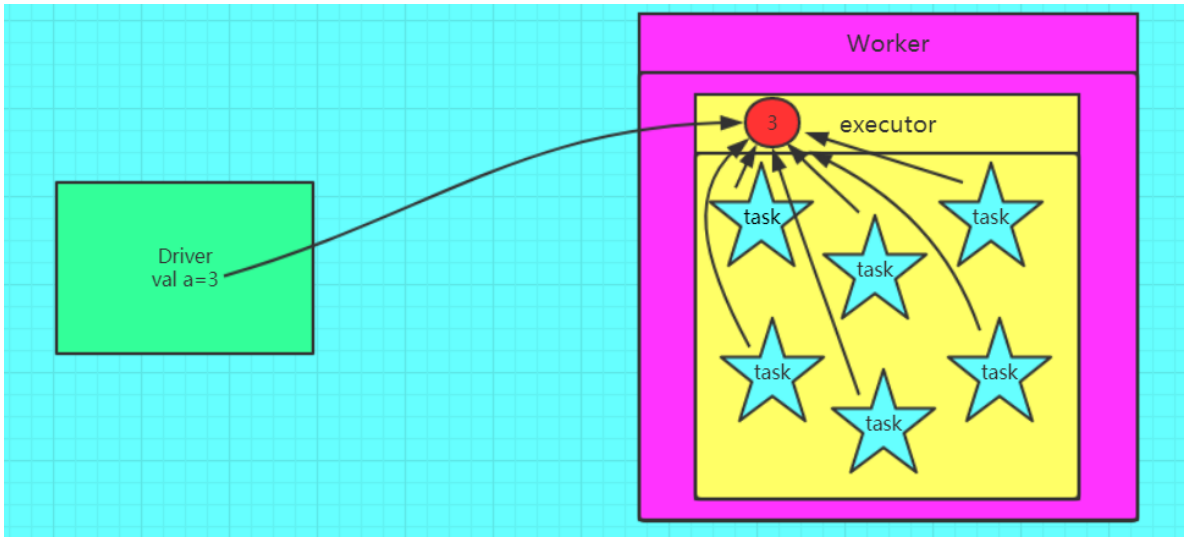
2.1.1. 为什么要定义广播变量

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：



2.1.2. 如何定义和还原一个广播变量

定义：

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

访问：

```
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

2.1.3. 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。

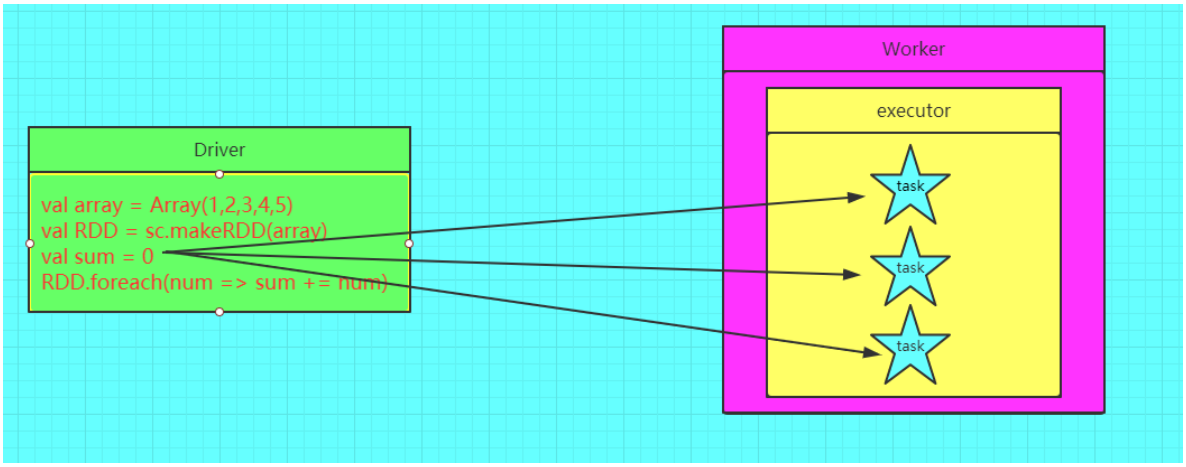
5、如果 Executor 端用到了 Driver 的变量，如果使用广播变量在每个 Executor 中都只有一份 Driver 端的变量副本。

2.2. Accumulators（累加器）

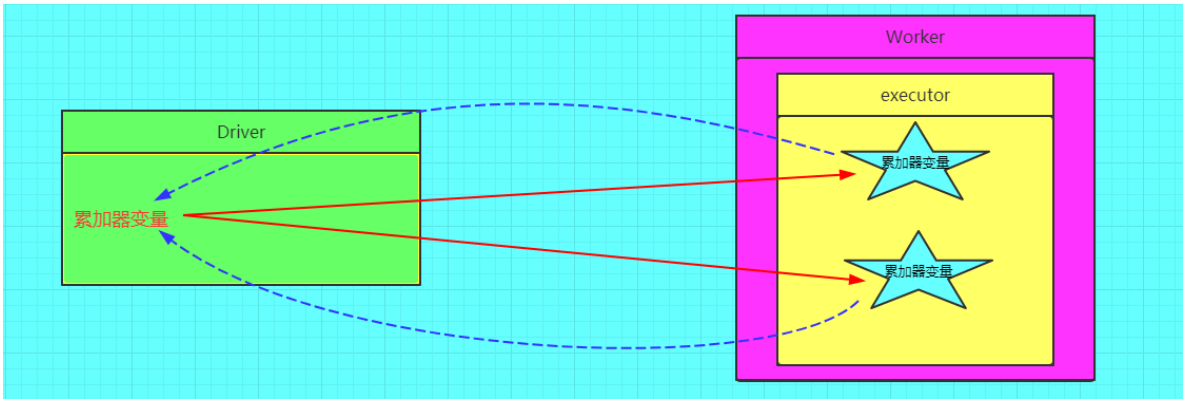
2.2.1. 为什么要定义累加器

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。

错误的图解：



正确的图解：



2.2.2. 如果定义和还原一个累加器

定义累加器：

```
val acc = sc.longAccumulator("myacc")
```

还原累加器：

```
val value = acc.value
```

2.2.3. 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