第13章 Apache Spark基础及架构

• Objective (本课目标)

- [x] 了解Apache Spark 架构
- [x] Spark环境搭建
- [x] 掌握Spark核心数据结构 – Resilient Distributed Dataset (RDD)
- [x] 掌握RDD的数据变换及操作

• Apache Spark环境搭建

– 单节点模式

```
第一步，下载spark
下载spark https://mirrors.tuna.tsinghua.edu.cn/apache/spark/spark-2.4.0/

第二步，解压缩
tar -zxvf spark-2.4.0-bin-hadoop2.7.tgz -C ../install/

第三步骤，配置环境变量
SPARK_HOME=/root/install/spark-2.4.5-bin-hadoop2.7
PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$SCALA_HOME:$SPARK_H
OME:$FLINK_HOME:$HIVE_HOME/bin:$ZOOKEEPER_HOME/bin:$SPARK_HOME/bin

第四步，启动spark
sbin/start-all.sh

第五步，查看spark进程
120203 Worker
120030 Master

第六步，进入spark-shell进行测试
```

```
spark-shell
```

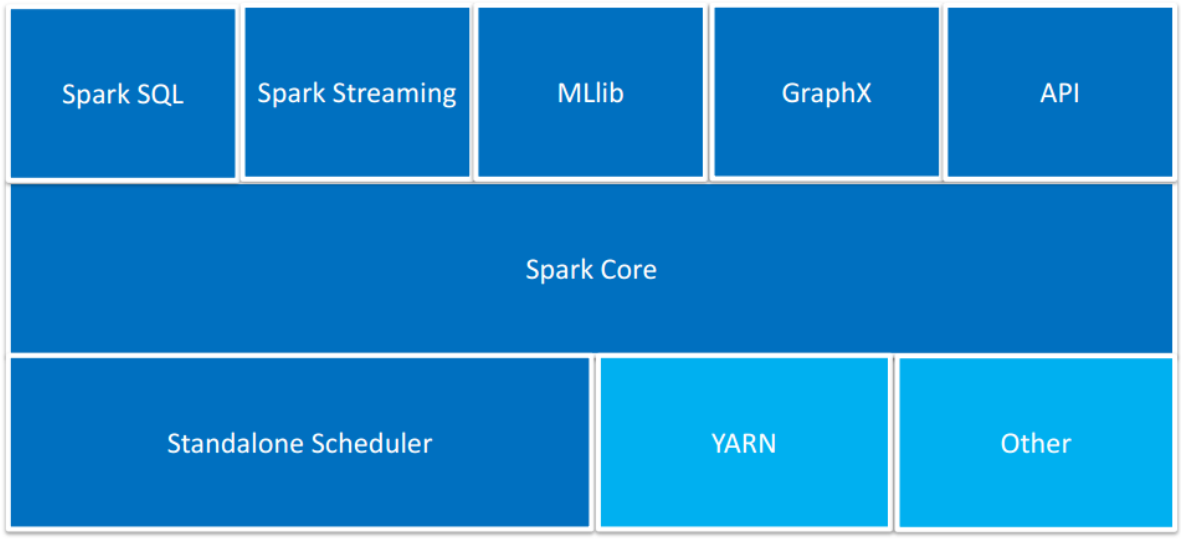
Spark分布式集群

Spark也是一个主从架构的分布式计算引擎。主节点是Master，从节点是Worker。所以集群规划：

Server	Master	Worker
hadoop1	√	√
hadoop2		√
hadoop3		√

```
第一步：上传安装包，然后解压缩安装到对应的目录
tar -zxvf spark-2.4.0-bin-hadoop2.7.tgz -C ../install/

第二步：修改配置文件spark-env.sh
export JAVA_HOME=/root/install/jdk1.8.0_65
export SCALA_HOME=/root/install/scala-2.12.11
export SPARK_HOME=/root/install/spark-2.4.5-bin-hadoop2.7
export SPARK_MASTER_HOST=hadoop1
export SPARK_EXECUTOR_MEMORY=1G

第三步：修改配置文件slave
hadoop1
hadoop2
hadoop3

第四步：分发安装到所有节点
scp -r spark-2.4.5-bin-hadoop2.7 root@hadoop2:/root/install/
scp -r spark-2.4.5-bin-hadoop2.7 root@hadoop3:/root/install/

第五步：配置环境变量
SPARK_HOME=/root/install/spark-2.4.5-bin-hadoop2.7/
PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME:$SCALA_HOME:$SPARK_HOME/bin:$FLINK_HOME:$H
ADOOP_HOME/bin:$HADOOP_HOME/sbin:$ZOOKEEPER_HOME/bin

第六步：启动集群
sbin/start-all.sh
```

Apache Spark Stack



Full stack and great ecosystem.
One stack to rule them all.

• Apache Spark Advantages(优势)

- Fast by leverage memory (基于内存速度快，尤其是迭代计算)
- General purpose framework – ecosystem (拥有自己的生态圈)
- Good and wide API support - native integration with Java, Python, Scala (良好而广泛的API支持 - 与Java，Python，Scala本地集成)
- Spark handles batch, interactive, and real-time within a single framework (单个框架内可以进行批处理，交互式处理，实时数据处理)
- Leverage Resilient Distributed Dataset – RDD to process data in memory (利用弹性分布式数据集 - RDD处理内存中的数据)
    - Resilient – Data can be recreated if it is lost in the memory (如果数据在内存中丢失，则可以重新创建数据)
    - Distributed – Stored in the memory across cluster (分布式 - 跨群集存储在内存中)
    - Dataset – Can be created from file or programmatically (数据集-可以从文件或编程方式创建)

• Spark Hall of Fame(名人堂)

How to Work with Spark(开发环境)

- Scala IDE
  - Add hadoop dependency or use mvn(添加hadoop依赖或者使用maven)
  - Compile to jar, build or download (http://spark.apache.org/docs/latest/building-spark.html)
  - Run with spark-submit
- spark-shell
  - Integrative command line application(集成命令行应用)
- spark-sql
  - where spark using hive metadata(读取hive的matedata)

Demo – Word Count(spark 词频统计)

MR code

```
package org.apache.hadoop.examples;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{

private final static IntWritable one = new IntWritable(1);
private Text word = new Text();

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);
  }
}
}

public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
}

public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
if (otherArgs.length != 2) {
  System.err.println("Usage: wordcount <in> <out>");
  System.exit(2);
}
Job job = new Job(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

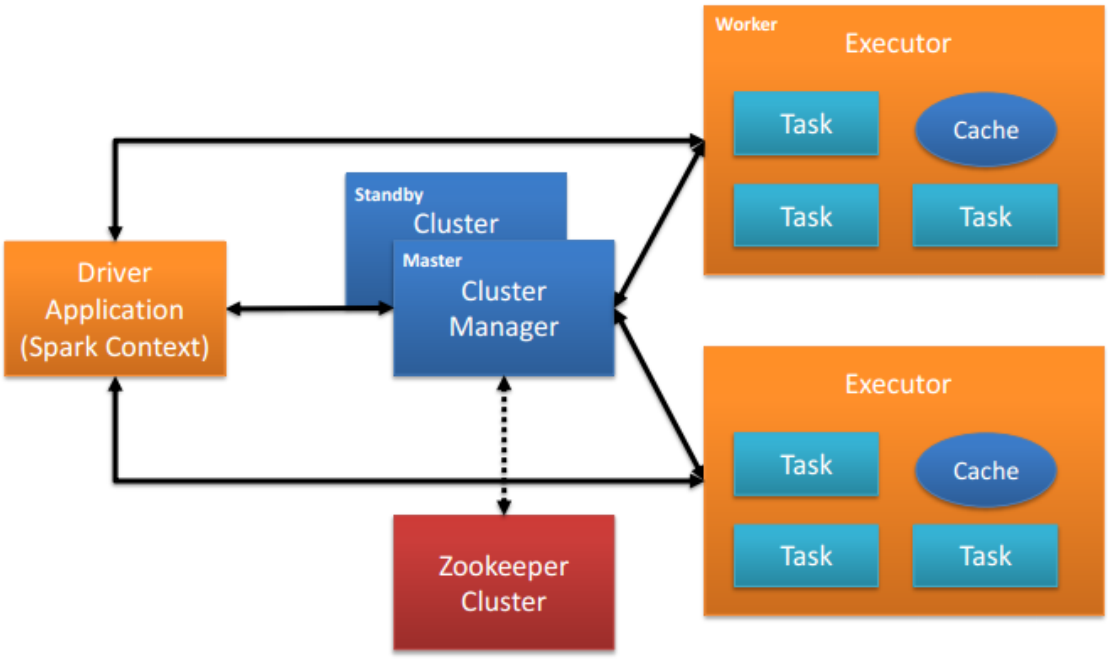
Hive code

```
select word, count(*) as word_cnt
from wordcount
lateral view explode(split(line,' ')) t1 as word
group by word
order by word_cnt desc
limit 10
```

How about spark code?

```
val wordCounts = sc.textFile("/data/data_words/wordcount.txt").
flatMap(_.split(" ")).filter(_.length>1).map(word=>(word,1)).
reduceByKey(_+_).map(item => item.swap).sortByKey(false);

val result = wordCounts.take(10).foreach(x=>println(x));
```

Spark Architecture(Spark架构)

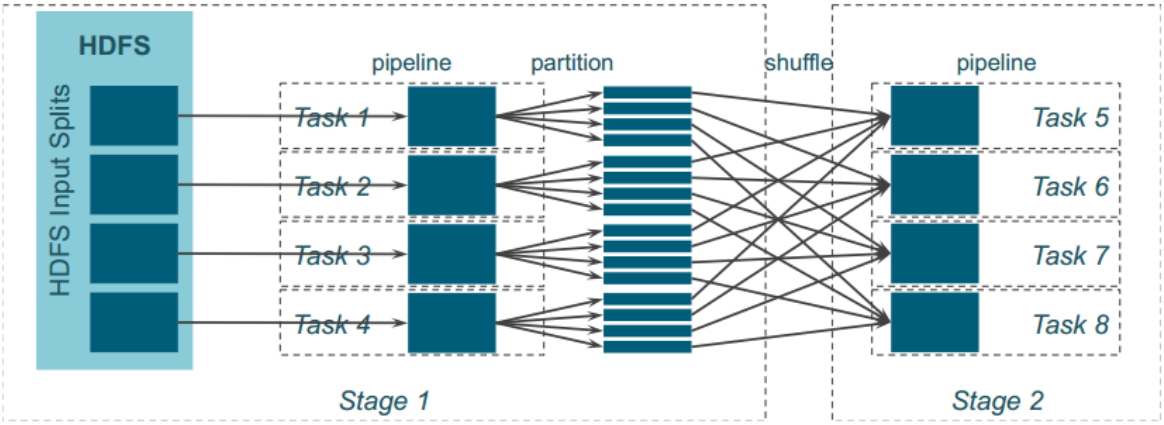


- SparkContxt：写Spark任务一定要创建SparkContxt，在Driver节点上创建并且停留在Driver节点上

Spark Core Components(Spark核心组件)

Application	**User program built on Spark. Consists of a driver program and executors on the cluster.(用户程序建立在Spark上。 由群集上的驱动程序和执行程序组成)**
Driver program	The process running the main() function of the application and creating the SparkContext (运行应用程序的main函数并创建SparkContext的进程)
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN) 集群资源管理器，standalone, Yarn，Mesos。
Worker node	Any node that can run application code in the cluster (可以在集群中运行应用程序代码的 任何节点)
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. (在工作节点上为应用程序启动的进程，它运行任务并将数据保存在内存或磁盘存储中。每个应用程序都有自己的执行器，可以理解它就是一个JVM)
Task	A unit of work that will be sent to one executor (具体工作的任务，一个partition一个task)
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.(由多个任务组成的并行计算，一个任务包含多个Task)
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.(每个作业被分成较小的任务组，称为阶段，彼此依赖)

Example of Partial Word Count(部分统计)



- One task per partition, one partition per split(一个任务对应一个分区，一个分区对应一个逻辑块)
- No. of stages is coming from shuffle/wide transformations (stages来自于shuffle/wide变换)

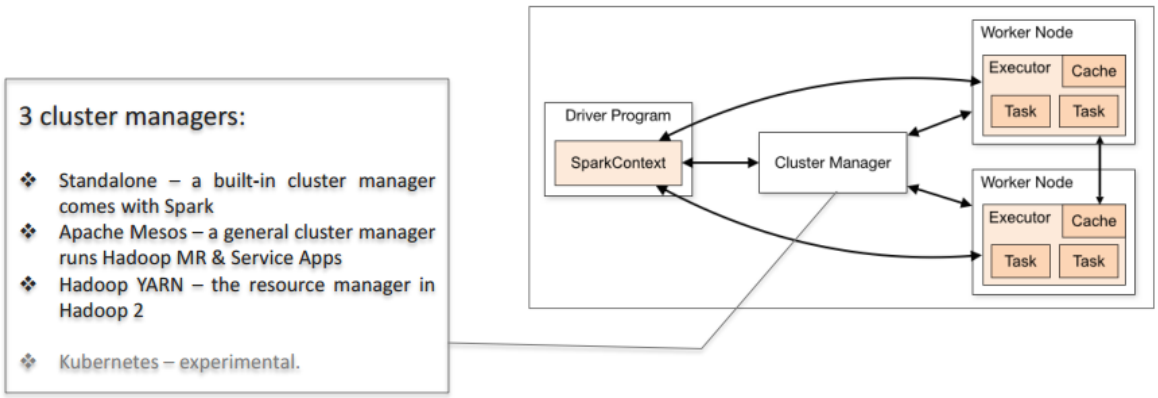
RDD默认分区数量

- 分区数量是根据在HDFS的物理块决定的

```
val rdd = sc.parallelize(text.split(" "),5)
rdd.partitions.size
rdd.mapPartitionsWithIndex((idx,iter) => if (idx ==0) iter else
Iterator()).collect
```

Apache Spark API - SparkContext

- SparkContext
  - The connection between Driver and Spark Cluster (Workers)
  - The main entry point for Spark functionality (Spark功能的主要入口点)
  - Only one active SparkContext per JVM. (一个JVM只有一个SparkContext)
  - SparkContext.getOrCreate



Apache Spark API - SparkSession

- SparkSession is the main-entry point for Spark 2.0+ applications: (SparkSession是Spark 2.0+应用程序的主要入口点)
    - The combination of SparkContext, SQLContext, HiveContext & StreamingContext (综合了以上所有功能)
    - SparkSession.getOrCreate
        - Get an existing SparkSession or create a new One.
        - Only one Global SparkSession per JVM. (一个JVM只有一个全局SparkSession)
    - SparkSession.newSession
        - Start a new session with isolated SQL configurations. Temporary tables, registered functions are isolated as well. (使用隔离的SQL配置启动一个新会话。临时表和已注册函数也是相互隔离的)
        - SparkContext and Cached Data are share d. (共享SparkContext和缓存的数据)
        - SparkSession.newSession，可以基于Global SparkSession创建更多的子session，用于资源隔离

Apache Spark API – RDD, DataSet

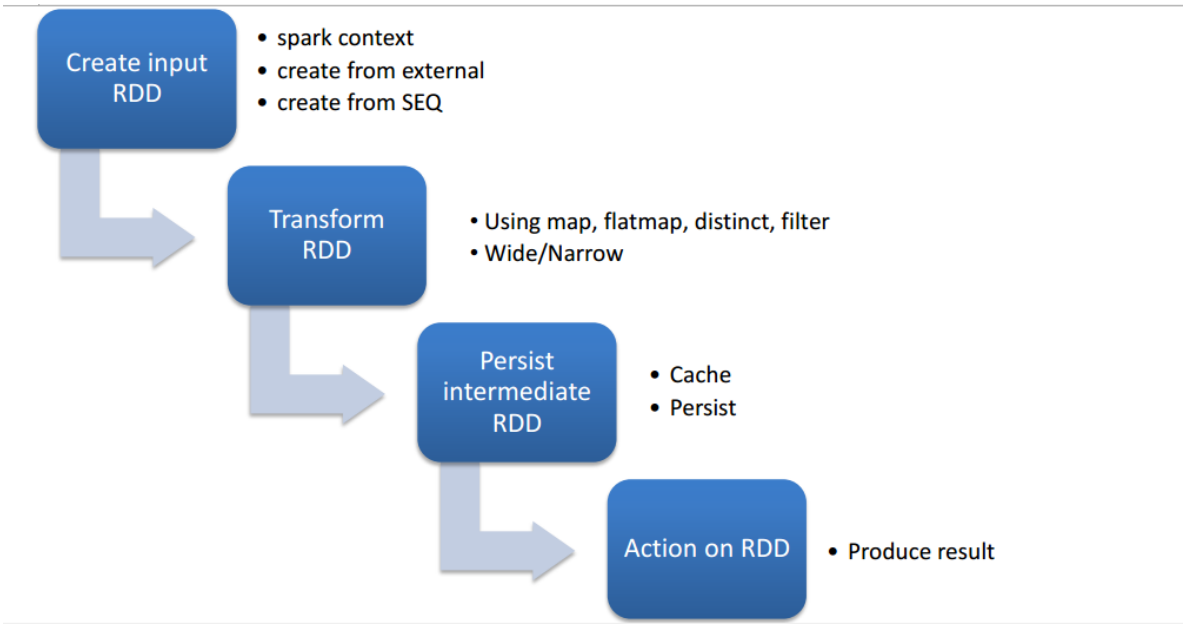
- RDD(Resilient Distributed Dataset)
    - The primary data abstraction and the core of Spark (主要是SparkCore的核心的抽象数据集)
        - RDD(Int,String) RDD(String) RDD(Array[])
- DataSet
    - Strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. (可使用函数或关系操作，并行转换的特定对象的强类型集合)
    - RDD(User) RDD(Employ)
    - String : id,name,age,birthyear -> caseCalss User -> UserObj
- DataFrame
    - An untyped view of one DataSet, which is a Dataset of Row. DataFrame = Dataset[Row] (一个数据集的无类型视图，它是一个行数据集。DataFrame =Dataset[Row])
    - String : id,name,age,birthyear -> DataFrame -> 基于SQL

RDD Concepts(概念)

- Simple explanation (简单的解答)
    - RDD is collection of data items split into partitions, stored in memory on worker nodes of the cluster, and perform proper operations. (RDD是分为多个分区的数据集，存储在集群中的worker节点的内存中，进行数据变换操作)
- Complex explanation (复杂的解答)

- RDD is an interface for data transformation (RDD是数据变换的接口)
  - rdd(String) -> rdd(int,String)
- RDD refers to the data stored either in persisted store (HDFS, Cassandra, HBase, etc.) or in cache (memory, memory+disks, disk only, etc.) or in another RDD (RDD是指存储在(HDFS、Cassandra、HBase)等或缓存(内存、内存+磁盘、仅磁盘等)或另一个RDD中的数据)
- Partitions are recomputed on failure or cache eviction (分区在失败或缓存清除时重新计算)

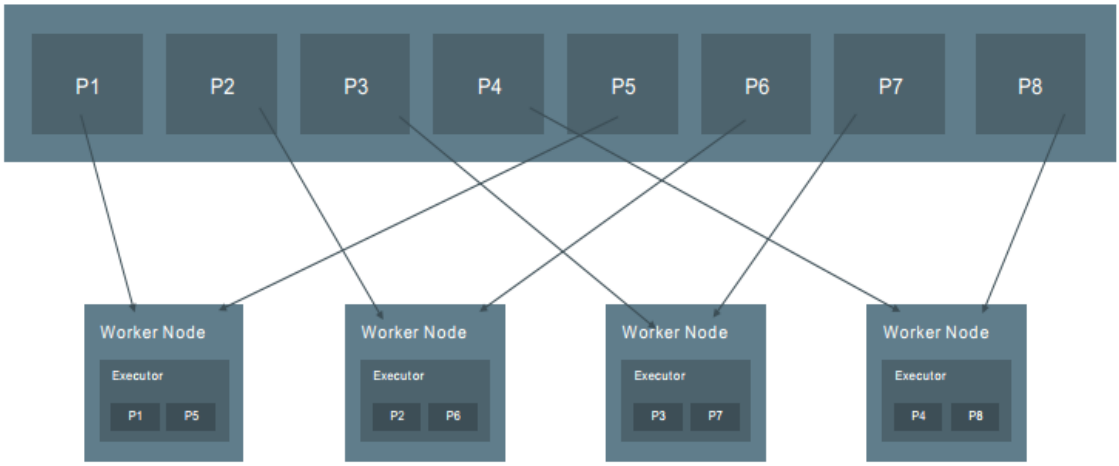
Spark Program Flow by RDD(RDD的Spark程序流程)




```
// parallelize 基于现有数据形成 RDD
val s = sc.parallelize(Array(1,2,3,4,5,6,7))
myRDD = sc.textFile("hdfs://tmp/shakepeare.txt")
```

RDD Partitions (RDD分区)

- A Partition is one of the different chunks that a RDD is split on and that is sent to a node (分区是RDD被分割并发送到节点的不同块之一)
- The more partitions we have, the more parallelism we get (我们拥有的分区越多，获得的并行度就越高)
- Each partition is a candidate to be spread out to different worker nodes (每个分区都是分发到不同工作节点的候选对象)

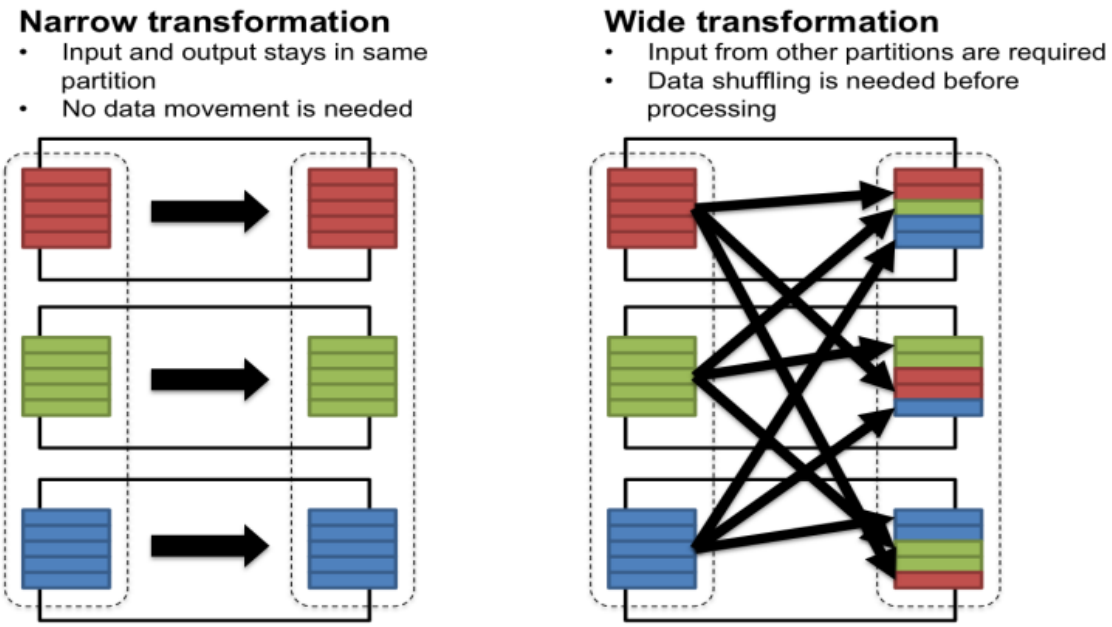


RDD Operations(RDD操作)

- RDD is the only data manipulation tool in Apache Spark (RDD是Apache Spark中唯一的数据集)
- Lazy and non-lazy operations (分为延迟和非延迟操作)
  - There are two types of RDDs (RDD有两种类型)
    - Transformation – lazy – returns Array[T] by collect action (Transformation是延迟执行的)
    - Actions – non-lazy – returns value (Action是非延迟的)
- Narrow and wide transformations (变换分为,Narrow和wide两种形式)
  - Narrow Dependency: operate on a single partition and map the data of the partition to the resulting single partition, such as map, filter, etc.(在本地节点上进行数据变换，比如map, filter)
  - Wide Dependency: operate across partitions, and map the data across partitions to the resulting partitions, such as groupByKey, distinct etc (跨集群节点进行数据变换，数据会在节点之间进行传递，比如groupByKey,distinc等等)

```
//延迟加载
val data = users.map(x => x.split(" "))

// 非延迟加载
data.take(10)
```

Narrow/Wide Transformation(宽窄变换)

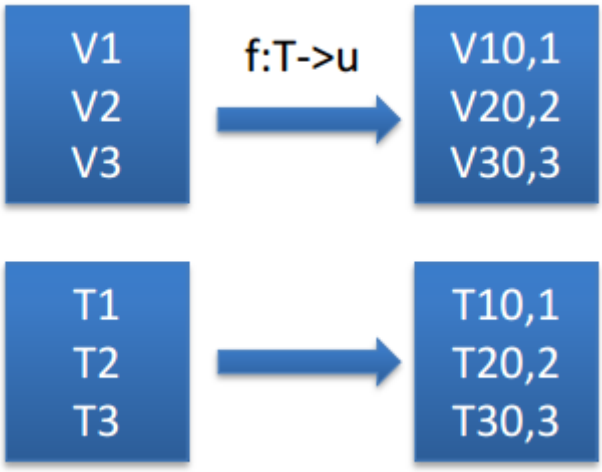


- Narrow：输入和输出都是同样的partition，不需要数据移动
- Wide：数据处理会产生shuffle然后形成新的partition

RDD Transformation - Map

- Applies a transformation function on each item of the RDD and returns the result as a new RDD(基于RDD进行数据变换，返回的结果是一个新的RDD)
  - A narrow transformation(窄依赖)

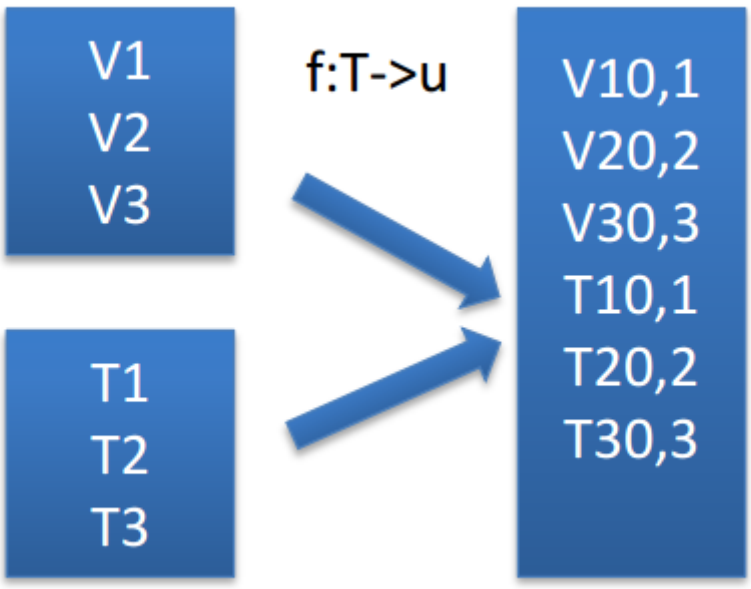
```
//案例1 有什么区别?
val data1 = rdd1.map(s => s.split(" ").map(s => (s,1)))
val data2 = rdd1.map(s => s.split("")).map(s => (s,1))

//案例2
val a = sc.parallelize(List("dog", "salmon", "pig"),3)
val b = a.map(x=>(x,x.length))
b.collect()
b.partitions.size
```



RDD Transformation - flatMap

- Similar to map, but each input item can be mapped to 0 or more output items(类似于map，每个item映射到0个或者多个输出)
    - A narrow transformation(窄依赖)

```
val a = sc.parallelize(List("dog", "salmon", "pig"), 3)
val c = a.flatMap(x=>List(x,x))
c.collect
val d = a.map(x=>List(x,x)).collect
```

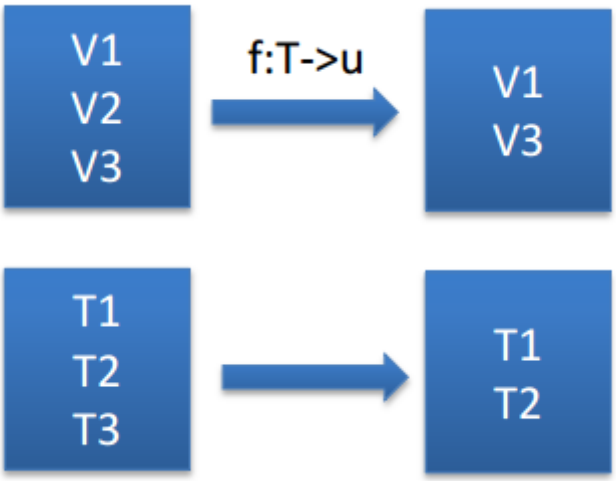


RDD Transformations - filter

- Create a new RDD by passing in a function used to filter the results. It is similar to where clause in SQL language(通过传入用于过滤的函数来创建新的RDD。 它类似于SQL语言中的where子句)

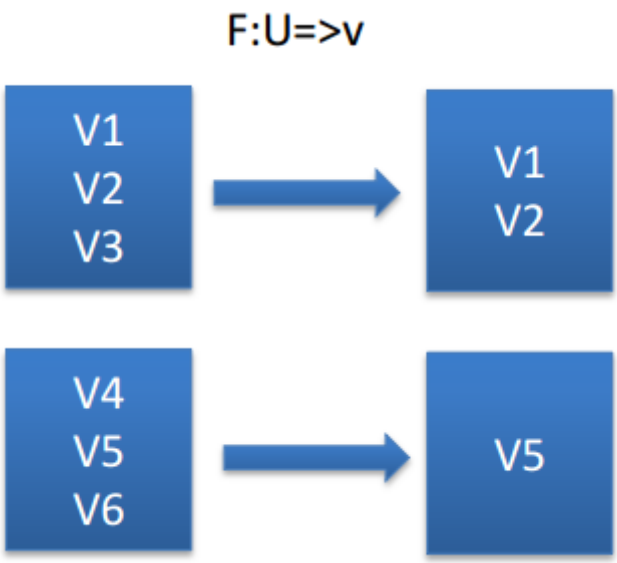
- A narrow transformation(窄依赖)

```
val a = sc.parallelize(List("dog", "salmon", "pig"), 3)
val e = a.filter(x=>x.length>3)
e.collect
```



- **RDD Transformations - sample**

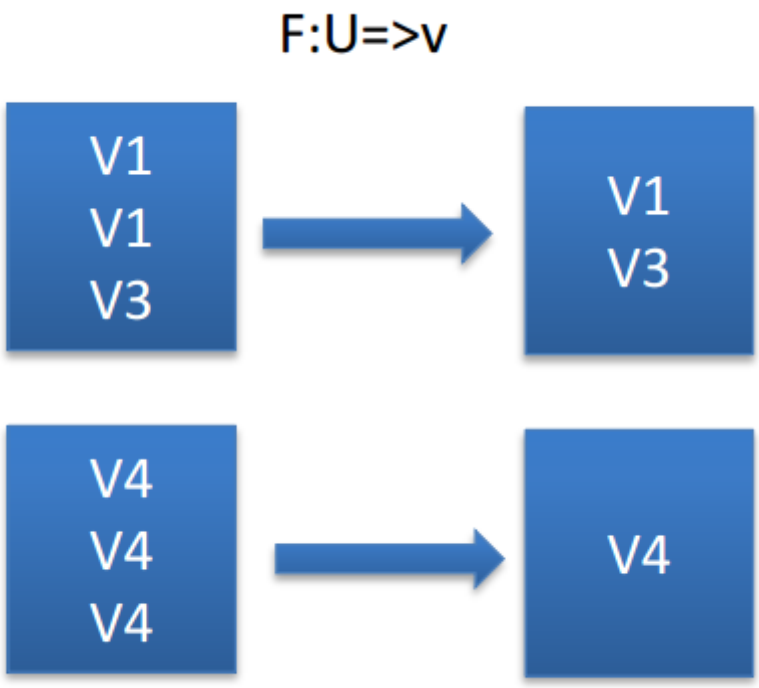
```
def sample(withReplacement: Boolean, fraction: Double, seed: Int): RDD[T]
//- withReplacement: whether or not to put the selected element back(是否放回所选元素)
//- fraction: the probability of every element in the source gets selected.(分数:选择源中每个元素的概率)
//- seed: random number generator seed. (种子: 随机数生成器种子)
val sam = sc.parallelize(1 to 10, 3)
sam.sample(false, 0.1, 0).collect
sam.sample(true, 0.3, 1).collect
```



RDD Transformations - distinct?

- Return a new RDD with distinct elements within a source RDD with specific partitions(去除重复)
    - A narrow/wide transformation

```
Val dis = sc.parallelize(List(1,2,3,4,5,6,7,8,9,9,2,6))
dis.distinct.collect
dis.distinct(2).partitions.length
//内部肯定涉及到数据的排序和分区，把重复的去掉，如果完全没有重复的，那么就是Narrow
```

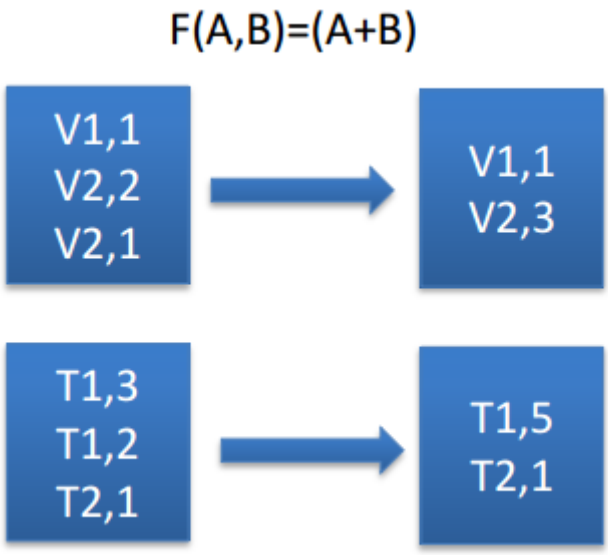


RDD Transformations - reduceByKey

- Operates on (K, V) pairs of course, but the function must be of type (V, V) => V (在（K，V）对上操作，但是该函数必须为（V，V）=> V类型)
    - A narrow/wide transformation

```
//把k-v相同的数据放到同一个分区
val a = sc.parallelize(List("dog", "salmon", "pig"), 3)
val f = a.map(x=>(x.length,x))
f.reduceByKey((a,b)=>(a+b)).collect
f.reduceByKey(_+_).collect
f.groupByKey.collect
```

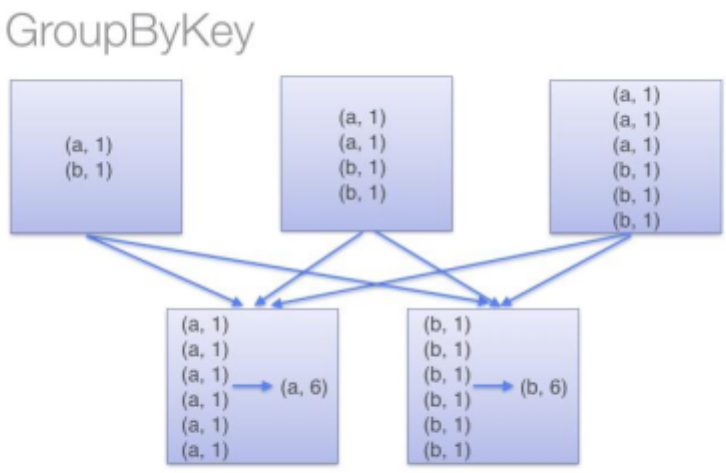
In terms of group, ReduceByKey is more efficient with combiner in the map phase. (在分组方面，ReduceByKey更高效，在map阶段使用了combiner进行本地聚合)



• RDD Transformation - groupByKey

- groupByKey groups elements by keys.(按key对元素进行分组)
  - Avoid groupByKey, and use reduceByKey or combineByKey(避免使用 groupByKey，推荐使用 reduceByKey,或 combineByKey)

```
val words = Array("one", "two", "two", "three", "three", "three")
val rddWordPairs = sc.parallelize(words).map(word => (word, 1))
rddWordPairs.groupByKey().map(t => (t._1, t._2.sum)).collect()
rddWordPairs.reduceByKey(_ + _).collect()
//reduceByKey和groupByKey有什么区别，reduceByKey会先在本地进行局部的汇总，然后再做全局汇总，groupByKey是直接全集做汇总。reduceByKey的效率更高
```

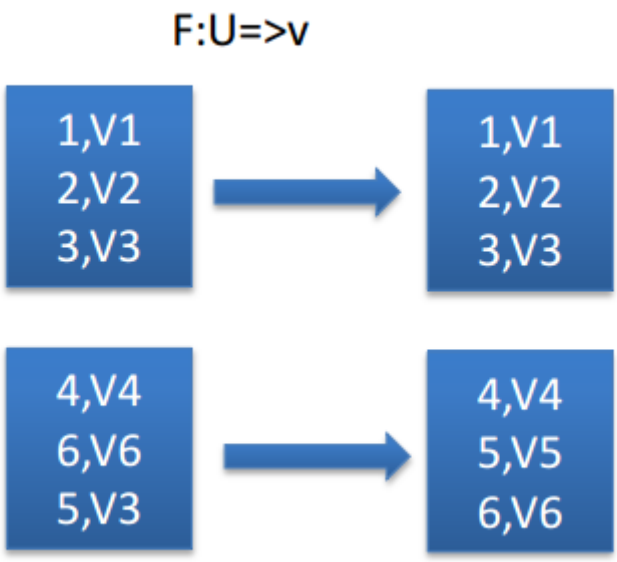


- groupByKey shuffles all data;
- reduceByKey suffles only the results of sub-aggregations in each partition of the data(reduceByKey会先对结果进行本地聚合)

RDD Transformations - sortByKey

- This simply sorts the (K,V) pair by K. Default is asc. False as desc.(这仅按K对（K，V）对进行排序。默认值为asc。 false是desc)
    - A narrow/wide transformation

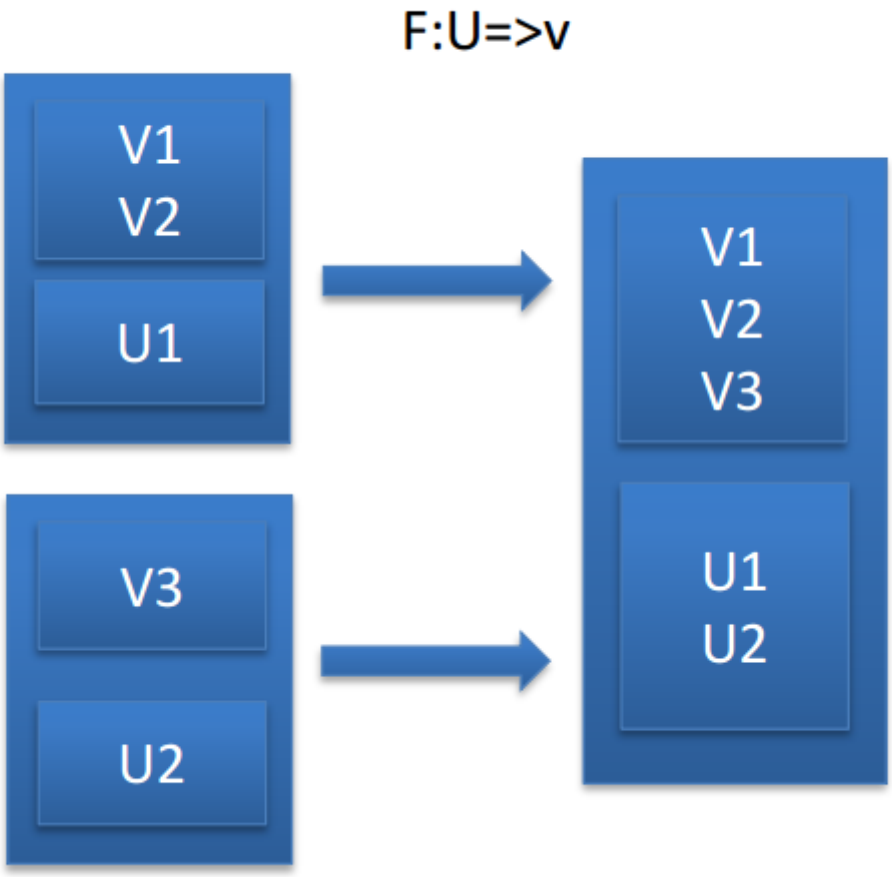
```
val a = sc.parallelize(List("dog", "salmon", "pig"), 3)
val f = a.map(x=>(x.length,x))
f.sortByKey().collect
f.sortByKey(false).collect
```



RDD Transformations - union

- Returns the union of two RDDs to a new RDD(合并两个RDD到一个RDD)
    - A narrow transformation

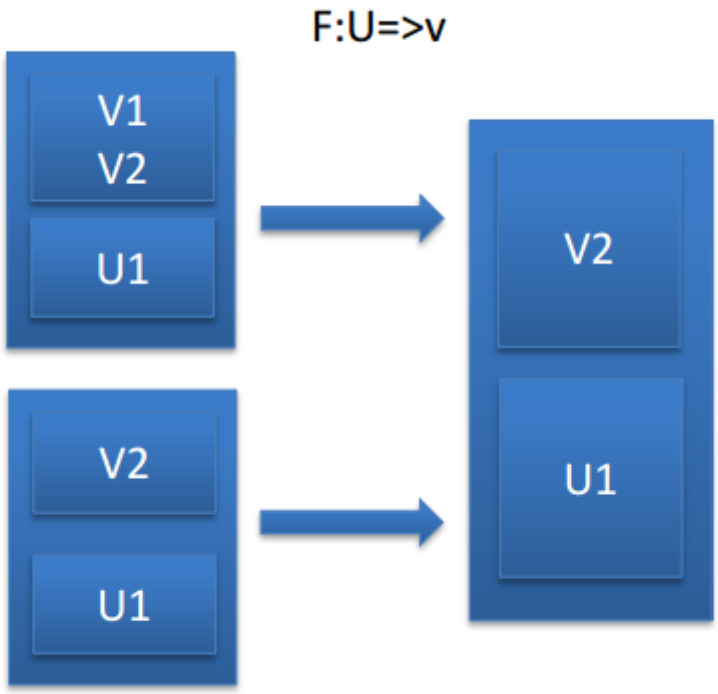
```
val u1 = sc.parallelize(1 to 10)
val u2 = sc.parallelize(11 to 20)
u1.union(u2).collect
(u1 ++ u2).collect
```



- **RDD Transformations - intersection**

  - Returns a new RDD containing only common elements from source RDD and argument (取交集)
    - A narrow/wide transformation

```
val u1 = sc.parallelize(1 to 3)
val u2 = sc.parallelize(3 to 4)
V2 u1.intersection(u2).collect
//wide 操作，需要进行比较是否有重复的，然后形成交集。
```

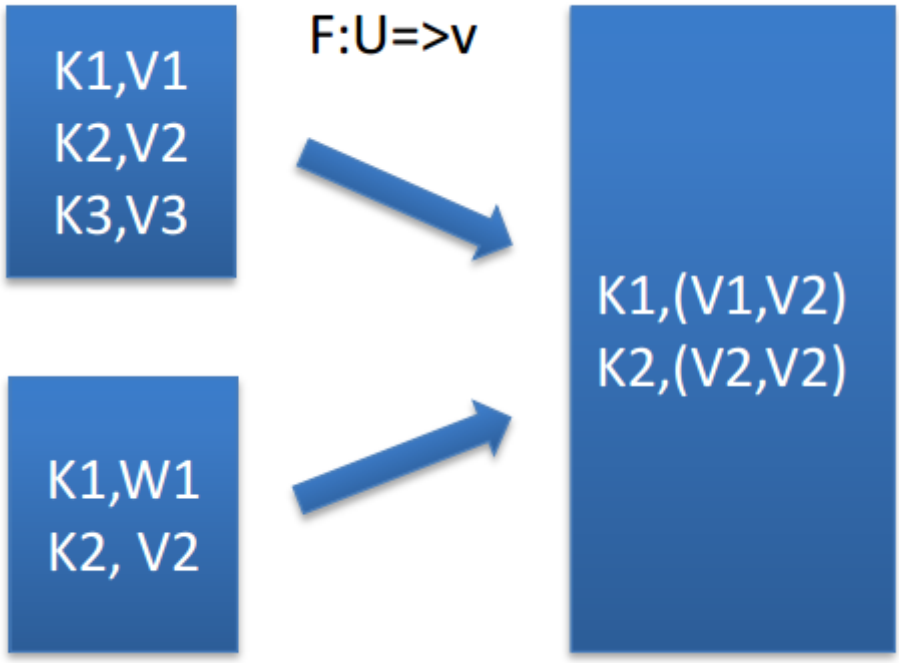


• RDD Transformations - join

- When invoked on (K, V) and (K, W), this operation creates a new RDD of (K, (V, W)). Must apply on k-v pairs. (相同key的value进行合并)
    - A wide transformation

```
//两个RDD如果需要join的话，那么一定要形成K-V，相同的key的value聚合到一起，形成新的RDD
val j1 = sc.parallelize(List("abe", "abby","apple")).map(a => (a, 1))
val j2 = sc.parallelize(List("apple", "beatty","beatrice")).map(a => (a, 1))
j1.join(j2).collect
j1.leftOuterJoin(j2).collect
j1.rightOuterJoin(j2).collect

// 复杂数据结构案例
case class User (id:Int,name: String, address: String)
case class Order(id:Int,cusotmer_id:Int,amount:Double)

val userRDD =
sc.parallelize(Seq(User(1,"zhangsan","beijing"),User(2,"lisi","shanghai"),User(3,
"wangwu","wuhan")))

val orderRDD =
sc.parallelize(Seq(Order(1,1,122.00d),Order(2,2,55d),Order(3,1,51d)))
userRDD.map(x => (x.id,x)).join(orderRDD.map(o => (o.customer_id,o))).collect


//案例2 join and cogroup diff
val rdd1 = sc.makeRDD(Array(("A",1),("B",2),("C",3)),2)
```

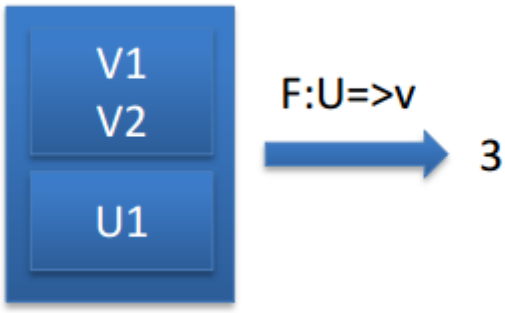
```
val rdd2 = sc.makeRDD(Array(("A",11),("B",22),("C",33)),2)
rdd1.join(rdd2)
rdd1.cogroup(rdd2)

val rdd2 = sc.makeRDD(Array(("A",11),("B",22),("C",33),("A",44)),2)
rdd1.join(rdd2)
rdd1.cogroup(rdd2)
```



- **RDD Actions - count**

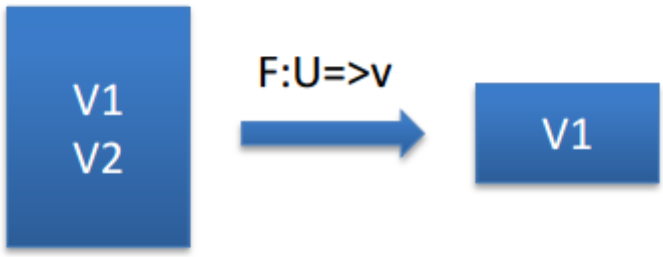
  - Get the number of data elements in the RDD (获取RDD中的数据元素数量)

```
val u1 = sc.parallelize(1 to 3)
val u2 = sc.parallelize(3 to 10)
u1.count
u2.count
```



• RDD Actions - take

- Fetch number of data elements from the RDD (从RDD中获取数据元素的数量)

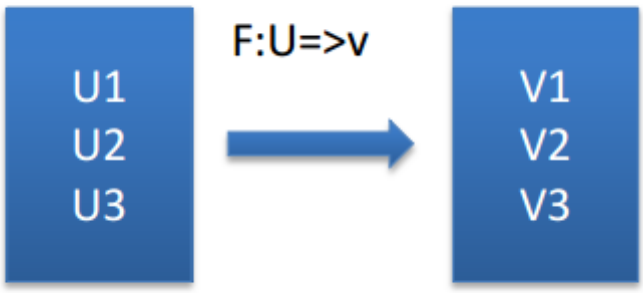



```
val t1 = sc.parallelize(1 to 100)
V1 t1.take(10)
//take: 从第一个分区开始往下找10个元素到DriverNode的内存，如果数据量过大会内存溢出
(DriverNode内存溢出)
// 数据集过大会造成内存溢出。take(count) 慎用
```

• RDD Actions - foreach

- Execute function for each data element in RDD(对RDD中的每个数据元素执行函数)

```
val c = sc.parallelize(List("cat", "dog", "tiger", "lion"))
u1.foreach(x => println(x + "is a number"))
//1. 把数据写到database可以通过foreach
//2. 如果多个节点同时工作，由于Spark的写数据速度很快，会对database造成压力
//3. 可以使用rePartitions，重新分区，减少partition，减轻database的压力（不推荐，
rePartition操作会有data shuffle）
//4. 最佳实践，rdd.foreachPartition(p => {...}) 每个partition逐个的写数据，不会造成数据
压力过大，也不会造成溢出
```



• RDD Actions - Collect

```
// 基本不用
//collect == rdd.take(rdd.count)
```

• RDD Actions - saveAsTextFile

- Writes the content of RDD to a text file or a set of text files to local file system/HDFS. The default path is HDFS or hdfs:// (将RDD的内容写到一个文本文件或一组文本文件到本地文件系统/ HDFS。 默认路径是HDFS或hdfs)

```
val a = sc.parallelize(1 to 10000, 3)
a.saveAsTextFile("file:///Users/will/Downloads/mydata")
-rw-r--r-- 1 root root 141 May 13 15:53 part-00000
-rw-r--r-- 1 root root 151 May 13 15:53 part-00001
-rw-r--r-- 1 root root 0 May 13 15:53 _SUCCESS
//保存为1个文件
a.repartition(1).saveAsTextFile(...) //default hdfs://
```

• 总结 (Summary)

- 了解了Spark的基本架构和功能组件

- 什么是RDD
  - Wide vs Narrow? 下面哪些是Wide或Narrow变换
  - flatMap, filter, groupByKey, distinct, union
- Transformation vs Action
  - foreach
  - filter, count

作业

- 使用spark完成词频统计
- 请将retail_db中的orders和order_items表输出成 csv. 请找
  - 谁是最大买“货”？(谁购买的最多，以￥￥￥算)
  - 哪个产品是最大卖货？(哪个产品销售的最多)