分布式-期末复习

⚠ **Warning** >

！！表示需要额外注意

⚠ 背 表示需要背诵

常用端口

以下是我们本学期的hadoop和spark的常用端口：

- HDFS NameNode 通信端口: 9820
- HDFS NameNode HTTP UI: 9870
- Yarn HTTP UI: 8088
- SPARK Master 通信端口：7077
- SPARK Master HTTP UI: 8080

配置相关文件作用整理

配置文件	作用	主要调整部分
`core-site.xml`	核心配置，适用于整个 Hadoop 环境	- **文件系统**（如hdfs，或者本地）
`hdfs-site.xml`	HDFS 相关设置	- 副本数（如 `dfs.replication`）- **NameNode 和 DataNode 目录**
`yarn-site.xml`	YARN 资源调度和管理相关设置	- **ResourceManager 地址- NodeManager 资源限制**
`mapred-site.xml`	MapReduce 框架设置	- **执行引擎**（**Yarn**）- **Map 和 Reduce 任务资源设置- 可修改默认队列**
`hadoop-env.sh`	Hadoop 环境变量配置	- **Java 路径**（如 `JAVA_HOME`）- **NameNode 和 DataNode 运行用户- ResourceManager 用户**
`capacity-scheduler.xml`	添加队列	- 队列的配置（如容量、优先级等）
`log4j.properties`	日志级别和存储路径	- **日志级别- 日志输出文件**

- hdfs-default.xml 如果需要修改块的参数 完全分布式只需在上面的基础上修改hosts/hostname

配置文件补充

hadoop

1. core-site.xml
2. hdfs-site.xml
3. hadoop-env.sh

9820
9870

yarn

1. mapred-site.xml
2. yarn-site.xml
3. hadoop-env.sh

7077

spark

1. spark-env.sh
2. workers
3. ~/. bashrc

8088
8080

HDFS

节点

- 名称节点，主节点
    - 存储元数据
    - 存储在内存
    - 保存block，datanode映射关系
- 数据节点，从节点
    - 存储文件
    - 存储在磁盘
    - 保存block id到datanode本地文件映射关系
- 第二名称节点
- 块block

    - 128mb，可调，
    - 最小化寻址开销
    - 最后一个块中的元素会被浪费
- 名称节点（NameNode）负责管理分布式文件系统的命名空间（Namespace）
    - FsImage
        - FsImage用于维护文件系统树以及文件树中所有的文件和文件夹的元数据
        - FsImage文件没有记录每个块存储在哪个数据节点，被报告机制
            - 一旦在内存中成功建立文件系统元数据的映射，则创建一个新的FsImage文件和一个空的EditLog文件
    - EditLog
        - 操作日志文件EditLog中记录了所有针对文件的创建、删除、重命名等操作
    - 名称节点记录了每个文件拆分成了哪几块，但是不知道他们在哪
- 数据节点是分布式文件系统HDFS的工作节点，负责数据的存储和读取，会根据客户端或者是名称节点的调度来进行数据的存储和检索，并且向名称节点定期发送自己所存储的块的列表
- SecondaryNameNode一般是单独运行在一台机器上
- HDFS只设置唯一一个名称节点：命名空间的限制：名称节点是保存在内存中的
- 多副本方式
    - •第一个副本：放置在上传文件的数据节点；如果是集群外提交，则随机挑选一台磁盘不太满、CPU不太忙的节点
    - •第二个副本：放置在与第一个副本不同的机架的节点上
    - •第三个副本：与第二个副本相同机架的其他节点上
    - •每个数据节点会定期向名称节点发送“心跳”信息，向名称节点报告自己的状态
    - •HDFS和其它分布式文件系统的最大区别就是可以调整冗余数据的位置

文件读取，创建

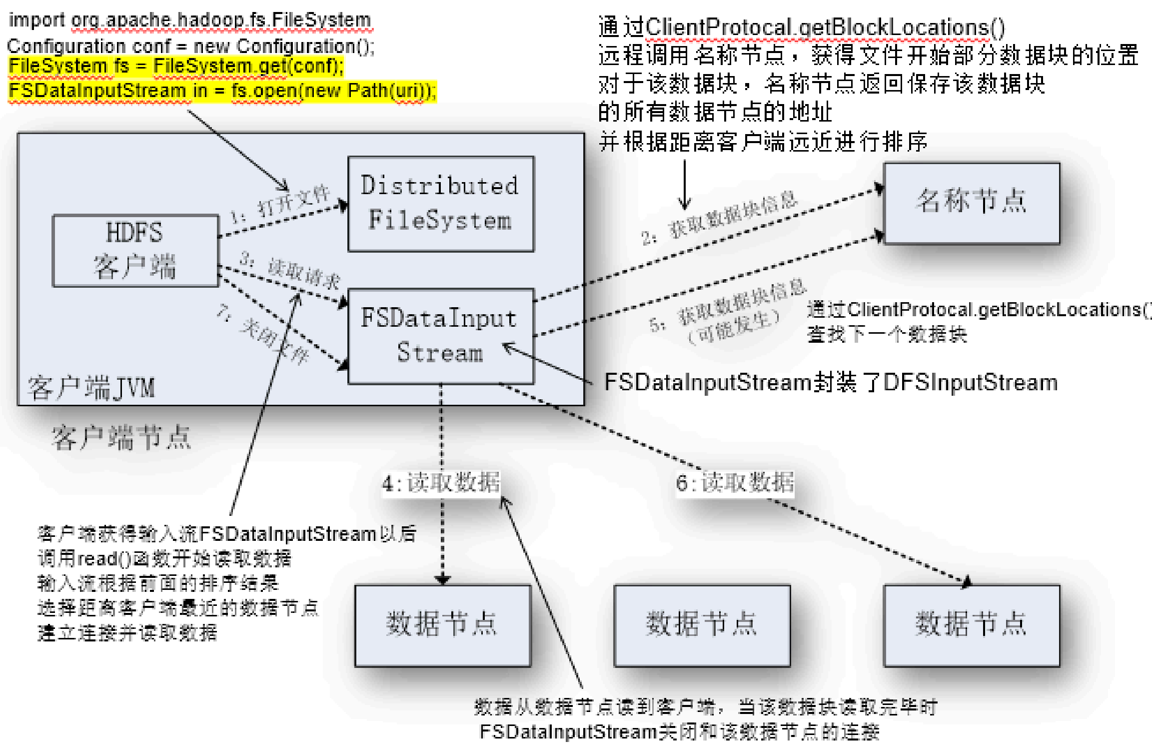
•FileSystem的open()方法返回的是一个输入流FSDataInputStream对象，在HDFS文件系统中，具体的输入流就是DFSInputStream；FileSystem中的create()方法返回的是一个输出流FSDataOutputStream对象，在HDFS文件系统中，具体的输出流就是DFSOutputStream。

```
FileSystem fs = FileSystem.get(conf);

FSDataInputStream in = fs.open(new Path(uri));

FSDataOutputStream out = fs.create(new Path(uri));
```

注意下图的顺序！

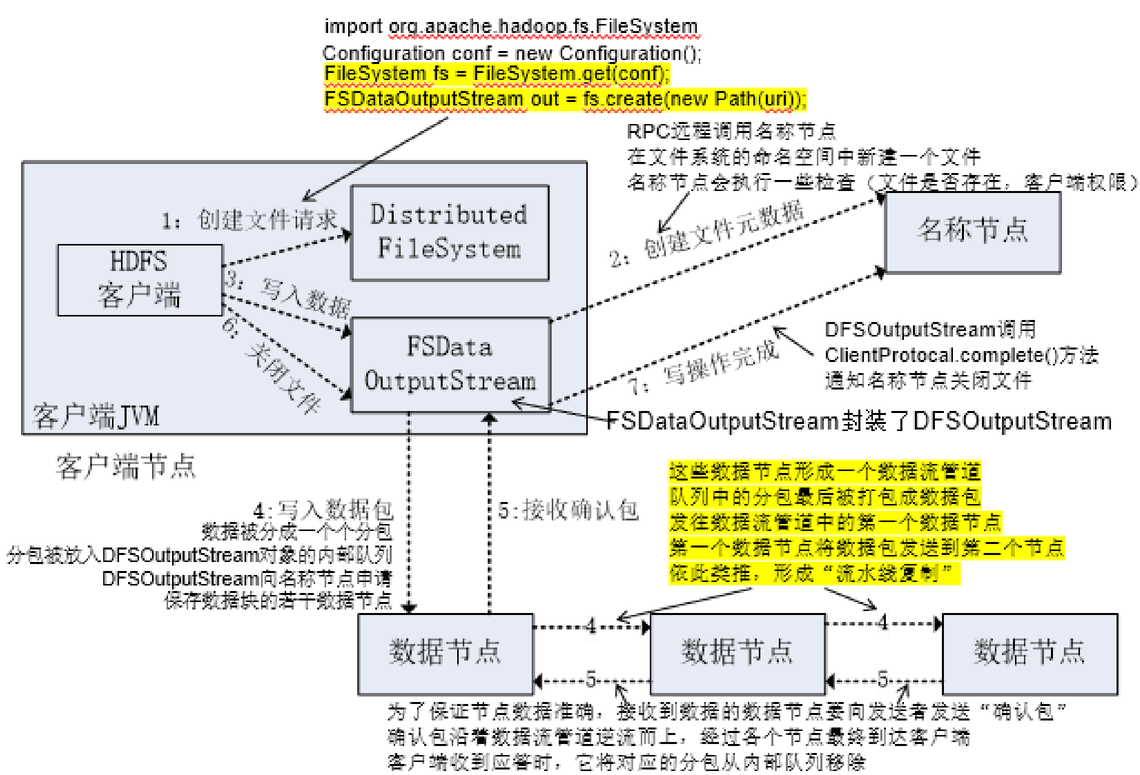
文件读取顺序，从最近的开始读

- 蹦出来的FSDataInputStream，掌管一切



文件写入顺序，流水线复制，

- 发出创建文件请求后，告知名称节点创建文件元数据。
- 最后一步需要告知名称节点关闭文件。

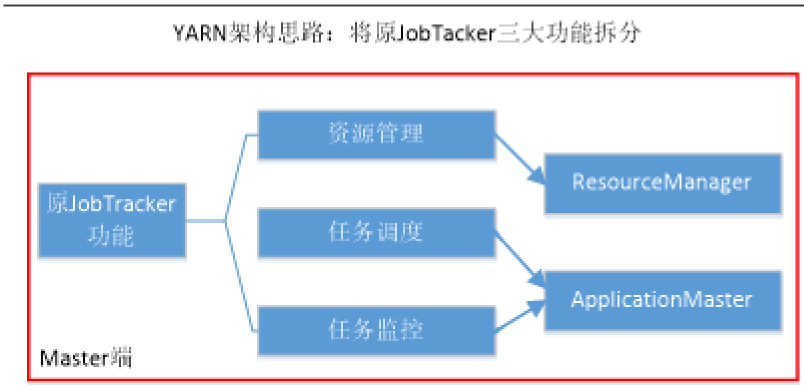


hdfs基本命令类似linux

```
# 启动/停止集群
1. start-dfs.sh
2. stop-dfs.sh
#创建目录
1. hdfs dfs -mkdir /test #单独创建
2. hdfs dfs -mkdir -p /test/a #级联创建
#查看
1. hdfs dfs -ls /hdfspath
2. hdfs dfs -cat
3. hdfs dfs -tail
#上传
1. hdfs dfs -put ~/input/test4.txt /input/ #local hdfs
# 2. hdfs dfs -moveFromLocal
# 3. hdfs dfs -copyFromLocal
#下载(从分布式下载到本地)
1. hdfs dfs -get hdfspath localpath
# 2. hdfs dfs -copyToLocal
# 3. hdfs dfs -getmerge #合并下载
#删除
1. hdfs dfs -rm (-r)#删除文件 r表递归,用于指定目录的情况
2. hdfs dfs -rmdir
3. hdfs dfs -cp path1 path2 #copy
4. hdfs dfs -mv path1 path2 #mov
```

YARN

- 纯粹的资源管理调度框架，而不是一个计算框架
- 拆分jobtracker（在namenode）
    - 任务相关的全交给application master（分布出去
    - 资源管理相关的全都交给resource manager
- 1.0中各个节点有tasktracker



ResourceManager

- 处理客户端请求
- 资源分配与调度
- 组成
    - Scheduler
        - 以“容器”（动态资源分配单位）的形式分配给提出申请的应用程序
        - 可插拔
    - Applications Manager
        - application master

ApplicationMaster

- 由Applications Manager启动，注销
- 为应用程序协商申请资源，并分配给内部任务
- 一个作业一个
- 与NodeManager保持交互通信
- 定时向ResourceManager发送“心跳”消息

NodeManager

- •单个节点上的资源管理
- NodeManager主要负责管理抽象的容器，只处理与容器相关的事情，而不具体负责每个任务（Map任务或Reduce任务）自身状态的管理

yarn工作流程

- 提交关于application master的请求
- resource manager启动容器，容器里有application master
- 注册
- 申请资源

- 给一个新容器
- 执行，监视，汇报

- application master自己申请注销

步骤1：用户编写客户端应用程序，向YARN提交应用程序，提交的内容包括ApplicationMaster程序、启动ApplicationMaster的命令、用户程序等

步骤2：YARN中的ResourceManager负责接收和处理来自客户端的请求，为应用程序分配一个容器，在该容器中启动一个ApplicationMaster

步骤3：ApplicationMaster被创建后会首先向ResourceManager注册

步骤4：ApplicationMaster采用轮询的方式向ResourceManager申请资源

步骤5：ResourceManager以“容器”的形式向提出申请的ApplicationMaster分配资源

步骤6：在容器中启动任务（运行环境、脚本）

步骤7：各个任务向ApplicationMaster汇报自己的状态和进度

步骤8：应用程序运行完成后，ApplicationMaster向ResourceManager的应用程序管理器注销并关闭自己

MapReduce

- •高度地抽象到了两个函数：Map和Reduce
- 计算向数据靠拢
- 输入输出，参考wordcount理解

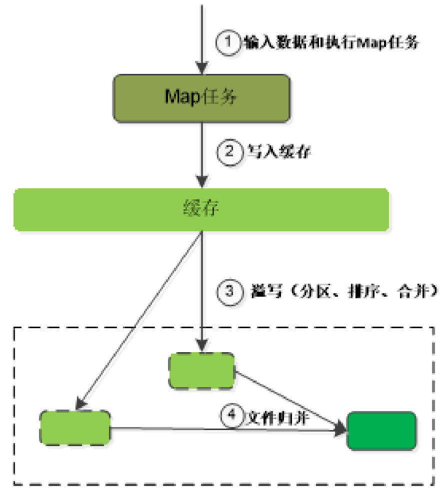
函数	输入	输出	说明
Map	$<k_1,v_1>$ 如： <行号,"a b c">	List($<k_2,v_2>$) 如： <"a",1> <"b",1> <"c",1>	1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理 2.每一个输入的$<k_1,v_1>$会输出一批$<k_2,v_2>$。$<k_2,v_2>$是计算的中间结果
Reduce	$<k_2,List(v_2)>$ 如： <"a",<1,1,1>>	$<k_3,v_3>$ <"a",3>	输入的中间结果$<k_2,List(v_2)>$中的List(v_2)表示是一批属于同一个k_2的value

- 不同的Map任务之间不会进行通信
- 不同的Reduce任务之间也不会发生任何信息交换
- HDFS 以固定大小的block 为基本单位存储数据，而对于MapReduce 而言，其处理单位是split。split 是一个逻辑概念
- •Hadoop为每个split创建一个Map任务，split 的多少决定了Map任务的数目。大多数情况下，理想的分片大小是一个HDFS块
- 一个map一个split
- 一个reduce一个分区
- shuffle
  - **分区**：将 Map 的输出按照键分配给不同的 Reduce 任务。
  - **排序**：对每个 Reduce 任务分区中的数据按照键排序。
  - **合并**（可选）：在数据传输和排序时合并小文件，减少网络和存储负担。

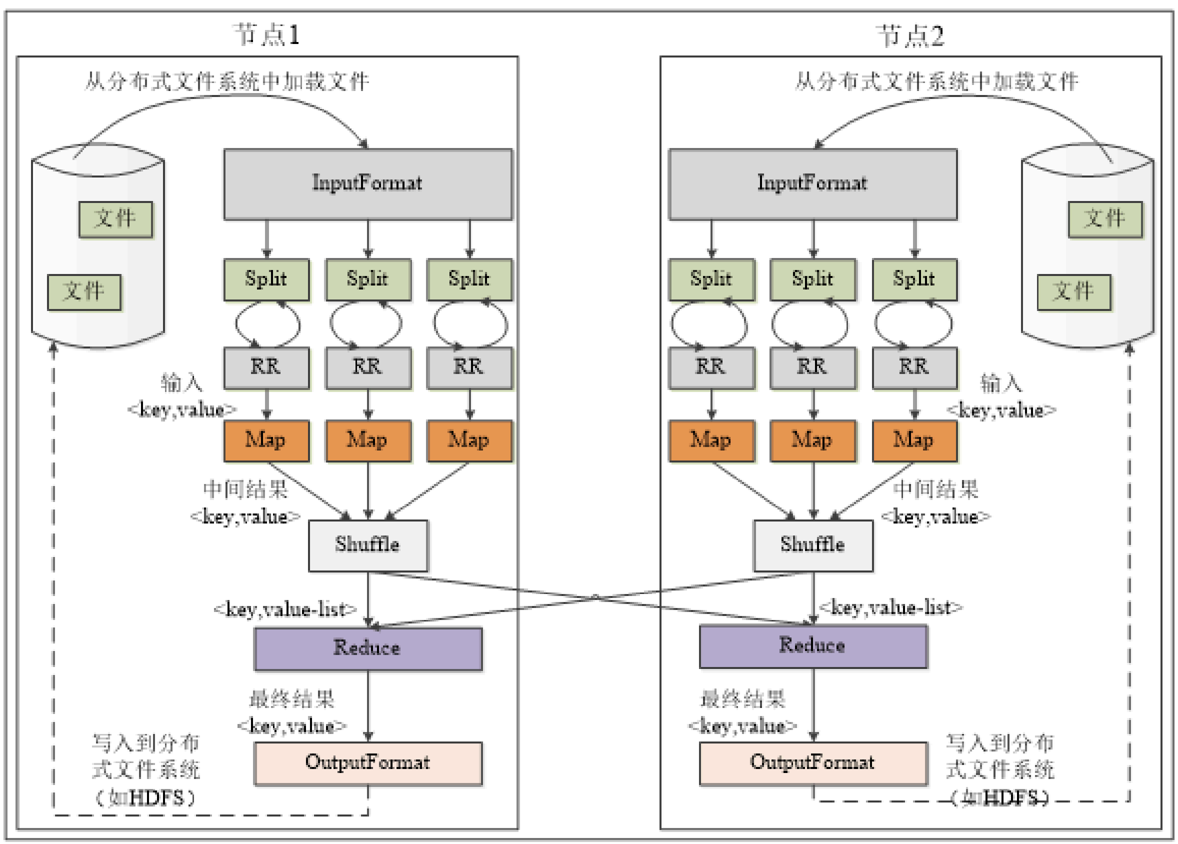
4.3.3 Shuffle过程详解，提前reduce

2. Map端的Shuffle过程

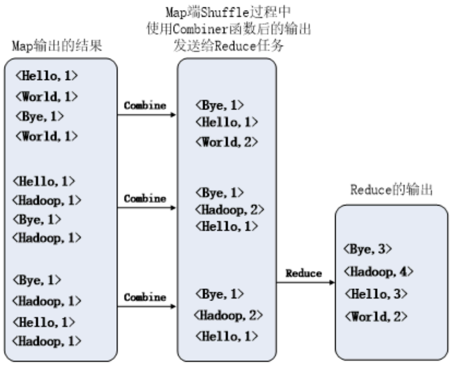
- 每个Map任务分配一个缓存;MapReduce默认100MB缓存;设置溢写比例0.8
- 分区默认采用哈希函数
- 排序是默认的操作
- 排序后可以合并（Combine）;合并不能改变最终结果
- 在Map任务全部结束之前进行归并
- 归并得到一个大的文件，放在本地磁盘
- 文件归并时，如果溢写文件数量大于预定值（默认是3）则可以再次启动Combiner，少于3不需要
- ApplicationMaster会一直监测Map任务的执行，并通知Reduce任务来领取数据



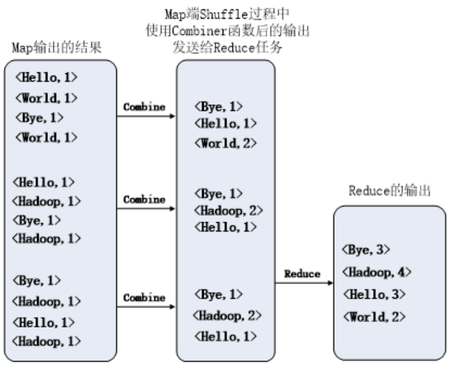
总体流程



wordcount，对比有无合并步骤的区别



有combiner



有combiner

4.4.4 WordCount源码

```
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> [<in>...] <out>");
    System.exit(2);
  }
  Job job = Job.getInstance(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);
```

关系代数 ⚠️背

- 选择
- 投影
- 并
- 交

- 差
- 自然连接

```
1. 选择
if ...
Map: t => <t, t>

2. 投影
Map: t => <t_1, t_1>  #键会出现多个t_1
Shuffle: <t_1, <t_1, t_1,...>>
Reduce: <t_1, t_1>

3. 并运算R \union S
Map: t => <t, t>
Shuffle: 1. <t, <t,t>> or 2. <t, t>
Reduce: if 1: <t, t>; if 2: <t, t>

4. 交运算 R \intersection S
Map: t => <t, t>
Shuffle: 1. <t, <t,t>> or 2. <t, t>
Reduce: if 1: <t, t>;

5. 差运算 R-S
Map: t => <t, <R, t>> or <t, <S, t>>
Shuffle: 1. <t, <<R, t>,<S,t>> or 2. <t, <R,t>> or 3. <t, <S,t>>
Reduce: if 2: <t, t>;

6. 自然连接 R<A, B> with S<B, C>
Map: (t_A, t_B) => <t_B, <R,t_A>> #R用于标记是属性A
(t_B, t_C) => <t_B, <S, t_C>>
Shuffle: 1.<t_B, <.<R, t_A>, <S, t_C>.> (R, S 同时出现) 2.others
Reduce: if 1: <t_A, t_B, t_C>
```

关系代数MapReduce的解释

1. 选择（Selection）

选择操作过滤出满足条件的元组。

伪代码解释：

```
if ...
Map: t => <t, t>
```

- **Map 阶段**：每个输入元组 `t` 被处理，如果满足选择条件（例如 `WHERE` 子句），则将该元组输出为 `(t, t)`。
- **Shuffle**：因为选择不涉及聚合，Shuffle 阶段只是将数据传递给 Reduce 阶段，通常会直接按键进行分发。
- **Reduce 阶段**：在 Reduce 阶段，所有满足选择条件的元组会被直接输出，不需要额外的聚合或处理。

---

2. 投影（Projection）

投影操作选择某些列。

伪代码解释：

```
Map: t => <t_1, t_1>
# 键会出现多个t_1
Shuffle: <t_1, <t_1, t_1,...>>
Reduce: <t_1, t_1>
```

- **Map 阶段**：每个输入元组 `t` 被处理，输出 `t_1`，其中 `t_1` 是选择的投影列（例如从 `t` 中选择某些特定列）。
- **Shuffle 阶段**：Shuffle 会将相同的投影列 `t_1` 聚集在一起，形成 `<t_1, <t_1, t_1,...>>` 的结构。
- **Reduce 阶段**：Reduce 阶段会根据键 `t_1` 将所有投影列聚合并输出最后的投影结果。

---

3. 并运算（Union）

并运算将两个关系 `R` 和 `S` 中的元组合并，去掉重复的元素。

伪代码解释：

```
Map: t => <t, t>
Shuffle: 1. <t, <t,t>> or 2. <t, t>
Reduce: if 1: <t, t>; if 2: <t, t>
```

- **Map 阶段**：每个输入元组 `t` 通过 Map 输出 `(t, t)`，即键值对的键和值都是 `t`。
- **Shuffle 阶段**：Shuffle 会将相同的 `t` 聚集在一起。可能会有两种情况：
  - 如果同一个键有多个相同的值（即 `t` 出现多次），会出现结构 `<t, <t,t>>`。

    - 如果每个键只出现一次，则结构为 `<t, t>`。
- **Reduce 阶段**：在 Reduce 阶段，处理相同 `t` 的记录。如果同一个键有多个值（`<t, <t,t>>`），我们输出 `t`，即去重；如果只有一个值，则直接输出 `t`。

---

4. 交运算（Intersection）

交运算将两个关系 `R` 和 `S` 中的公共元组保留下来。

伪代码解释：

```
Map: t => <t, t>
Shuffle: 1. <t, <t,t>> or 2. <t, t>
Reduce: if 1: <t, t>;
```

- **Map 阶段**：每个元组 `t` 会作为键值对 `(t, t)` 被输出。
- **Shuffle 阶段**：Shuffle 会将相同的 `t` 聚集在一起。如果 `t` 在多个输入中都出现，会形成 `<t, <t,t>>` 的结构，表示 `t` 出现在两个关系中。
- **Reduce 阶段**：在 Reduce 阶段，如果键 `t` 对应的值有多个（即出现在 `R` 和 `S` 中），则输出 `t`，表示 `t` 是 `R` 和 `S` 的交集。

---

5. 差运算（Difference）

差运算 `R - S` 用于从关系 `R` 中去除在关系 `S` 中出现的元组。

伪代码解释：

```
Map: t => <t, <R, t>> or <t, <S, t>>
Shuffle: 1. <t, <.<R, t>,<S,t>.> or 2. <t, <R,t>> or 3. <t, <S,t>>
Reduce: if 2: <t, t>;
```

- **Map 阶段**：每个元组 `t` 被处理，并在其值中标记 `R` 或 `S`。即输出键 `t`，值为 `<R, t>` 或 `<S, t>`，用来标记来自哪个关系。
- **Shuffle 阶段**：Shuffle 会将相同键 `t` 的记录聚集在一起，有三种情况：
    1. 如果 `t` 出现在 `R` 和 `S` 中，会形成结构 `<t, <R,t>, <S,t>>`。
    2. 如果 `t` 仅出现在 `R` 中，形成结构 `<t, <R,t>>`。
    3. 如果 `t` 仅出现在 `S` 中，形成结构 `<t, <S,t>>`。

- **Reduce 阶段**：如果 `t` 仅出现在 `R` 中（即只有 `<R,t>`），则将其输出，表示 `t` 属于 `R - S`。

---

6. 自然连接（Natural Join）

自然连接操作用于根据共同属性连接两个关系，通常基于相同的列。

伪代码解释：

```
Map: (t_A, t_B) => <t_B, <R,t_A>> # R 用于标记是属性 A
(t_B, t_C) => <t_B, <S, t_C>>
Shuffle: 1. <t_B, <.<R, t_A>, <S, t_C>.> (R, S 同时出现)  2. others
Reduce: if 1: <t_A, t_B, t_C>
```

- **Map 阶段**：对于两个关系 `R(A, B)` 和 `S(B, C)`：
  - 从关系 `R` 中提取 `t_A, t_B`，并输出 `<t_B, <R, t_A>>`，即以 `t_B` 为键。
  - 从关系 `S` 中提取 `t_B, t_C`，并输出 `<t_B, <S, t_C>>`，同样以 `t_B` 为键。
- **Shuffle 阶段**：Shuffle 会根据 `t_B` 聚集来自 `R` 和 `S` 的记录，可能有两种情况：
  1. 如果 `t_B` 在 `R` 和 `S` 中都出现，结构为 `<t_B, <R, t_A>, <S, t_C>>`，表示这是一个可以进行连接的键。
  2. 如果 `t_B` 仅出现在 `R` 或 `S` 中，结构为其他情况。
- **Reduce 阶段**：如果 `t_B` 出现在 `R` 和 `S` 中，都有记录（即结构为 `<t_B, <R, t_A>, <S, t_C>>`），则进行连接并输出连接结果 `t_A, t_B, t_C`。

MapReduce应用题1——自然连接

4.5 MapReduce的具体应用

用MapReduce实现关系的自然连接

雇员

Name	EmpId	DeptName
Harry	3415	财务
Sally	2241	销售
George	3401	财务
Harriet	2202	销售

部门

DeptName	Manager
财务	George
销售	Harriet
生产	Charles

雇员 ⋈ 部门

Name	EmpId	DeptName	Manage
Harry	3415	财务	George
Sally	2241	销售	Harriet
George	3401	财务	George
Harriet	2202	销售	Harriet

应用题1 MapReduce 伪代码

map阶段

映射逻辑：

- 从 `雇员` 表中提取键值对：`<DeptName, <雇员, Name, EmpId>>`。
- 从 `部门` 表中提取键值对：`<DeptName, <部门, Manager>>`。
- 每一行数据都以 `DeptName` 作为键，以行的其他部分作为值。

```
<财务, <雇员, Harry, 3415>>
<销售, <雇员, Sally, 2241>>
<财务, <雇员, George, 3401>>
<销售, <雇员, Harriet, 2202>>
<财务, <部门, George>>
<销售, <部门, Harriet>>
<生产, <部门, Charles>>
```

Shuffle 阶段

分组逻辑：

- 按照键 `DeptName` 对所有值进行分组，将来自两个表的数据组合到同一个键下。

示例分组结果：

```
财务: [<雇员, Harry, 3415>, <雇员, George, 3401>, <部门, George>]
销售: [<雇员, Sally, 2241>, <雇员, Harriet, 2202>, <部门, Harriet>]
生产: [<部门, Charles>]
```

Reduce 阶段

归约逻辑：

- 对每个键 `DeptName`，将来自 `雇员` 的记录与来自 `部门` 的记录进行匹配。
- 输出连接后的结果。

示例输出：

```
<Harry, 3415, 财务, George>
<George, 3401, 财务, George>
```

```
<Sally, 2241, 销售, Harriet>
<Harriet, 2202, 销售, Harriet>
```

MapReduce应用题2——分组聚合（投影+...）⚠️背

4.5.2 分组聚合运算

SUM

COUNT

AVG

MIN

MAX

设计相应的Reduce运算

```
1. 求和
Map: t => <k, v> #k为分组指标，v为作运算的值
Shuffle: <k, <...>>
Reduce: sum(list()) # list是k键对应的值的集合

2. 计数 同wordcount；或sum改为count
Map: t => <k, v> #k为分组指标，v为作运算的值
Shuffle: <k, <...>>
Reduce: count(list())

3. 平均:  sum改为averag
Map: t => <k, v> #k为分组指标，v为作运算的值
Shuffle: <k, <...>>
Reduce: averag(list())

4. min: sum改为min
Map: t => <k, v> #k为分组指标，v为作运算的值
Shuffle: <k, <...>>
Reduce: min(list())
```

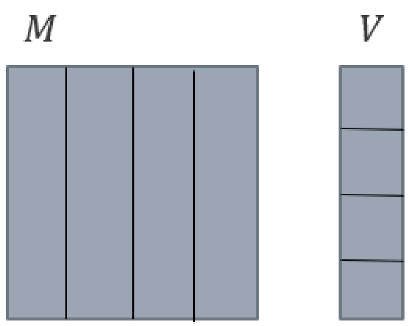
矩阵向量乘法，综合 ⚠️背

将M和V作为文件存放到HDFS中，M可以分块存放，V整体存放并读入内存

Map函数设计：输出$< i, m_{ij}v_j >$

Reduce函数设计:对输入的$< i, < m_{i1}v_1, m_{i2}v_2, \cdots, m_{in}v_{jn} \gg$,输出$< i, x_i >$

如果n很大，可以分割数据



矩阵和矩阵乘法 ⚠️背

矩阵X表示为$< i,j,x_{ij} >$,Y表示为$< j,k,y_{jk} >$，求$< i,k,z_{jk} >$

设计两个mapreduce任务

任务1，自然连接

1.Map函数：对X输出$< j,< X,i,x_{ij} >>$，对Y输出$< j,< Y,k,y_{ik} >>$

2.Reduce函数：输出$< j,< i,k,x_{ij}y_{jk} >>$

任务2，分组聚合

1.Map函数：输出$\ll i,k >,x_{ij}y_{jk}>$

2.Reduce函数：输入$\ll i,k >,< x_{i1}\ y_{1k}, x_{i2}y_{2k}, \ldots, x_{in}y_{2n}, \gg$,输出$\ll i,k >, z_{ik} >$

spark

基本知识

- •RDD：是Resillient Distributed Dataset（弹性分布式数据集）的简称，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型
- •DAG：是Directed Acyclic Graph（有向无环图）的简称，反映RDD之间的依赖关系
- •Executor：是运行在工作节点（WorkerNode）的一个进程，负责运行Task
    - •多线程来执行具体的任务，减少任务的启动开销

    - •二是Executor中有一个BlockManager存储模块，会将内存和磁盘共同作为存储设备，有效减少IO开销
- •Application：用户编写的Spark应用程序
- •Task：运行在Executor上的工作单元
- •Job：一个Job包含多个RDD及作用于相应RDD上的各种操作
- •Stage：是Job的基本调度单位，一个Job会分为多组Task，每组Task被称为Stage，或者也被称为TaskSet，代表了一组关联的、相互之间没有Shuffle依赖关系的任务组成的任务集

application

- 一个driver
- **很多 job**
    - **一个job 很多 stage，job调度单位，内部task关联**
        - **一个stage 很多 task**

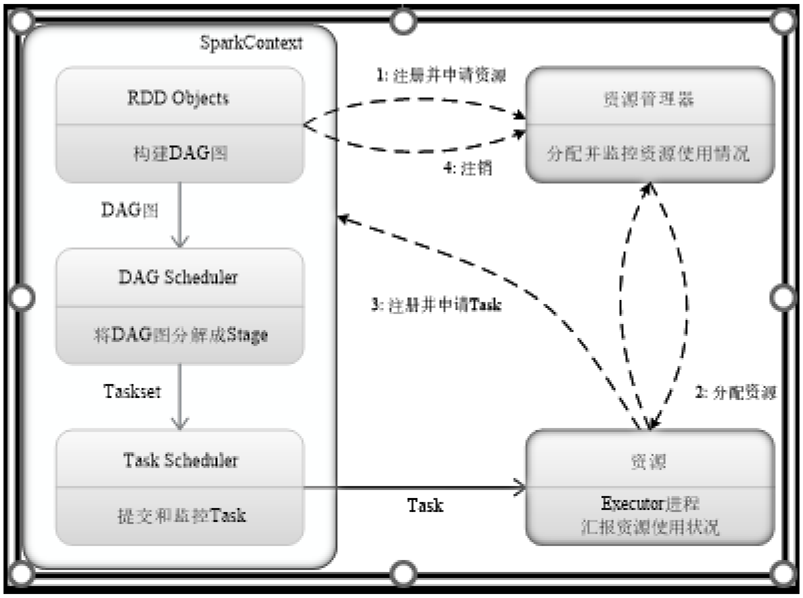
worknode

- executor
    - 执行一个task

架构设计

- •一个Application由一个Driver和若干个Job构成，一个Job由多个Stage构成，一个Stage由多个没有Shuffle关系的Task组成
- •当执行一个Application时，Driver会向集群管理器申请资源，启动Executor，并向Executor发送应用程序代码和文件，然后在Executor上执行Task，运行结束后，执行结果会返回给Driver，或者写到HDFS或者其他数据库中

运行流程



（1）首先为应用构建起基本的运行环境，即由Driver创建一个SparkContext，进行资源的申请、任务的分配和监控

（2）资源管理器为Executor分配资源，并启动Executor进程

（3）SparkContext根据RDD的依赖关系构建DAG图，DAG图提交给DAGScheduler解析成Stage，然后把一个个TaskSet提交给底层调度器TaskScheduler处理；Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行，并提供应用程序代码

（4）Task在Executor上运行，把执行结果反馈给TaskScheduler，然后反馈给DAGScheduler，运行完毕后写入数据并释放所有资源

RDD

- 一个RDD就是一个分布式对象集合，本质上是一个只读的分区记录集合
- RDD提供了一种高度受限的共享内存模型
  - RDD是只读的记录分区的集合，不能直接修改，只能基于稳定的物理存储中的数据集创建RDD
  - 或者通过在其他RDD上执行确定的转换操作（如map、join和group by）而创建得到新的RDD
- 操作：分为“动作”（Action）和“转换”（Transformation）两种类型
  - •RDD提供的转换接口都非常简单，都是类似map、filter、groupBy、join等粗粒度的数据转换操作，而不是针对某个数据项的细粒度修改
- 执行过程
  - •RDD读入外部数据源进行创建
  - •RDD经过一系列的转换（Transformation）操作，每一次都会产生不同的RDD，供给下一个转换操作使用
  - •最后一个RDD经过“动作”操作进行转换，并输出到外部数据源
- 这一系列处理称为一个Lineage（血缘关系），即DAG拓扑排序的结果

    - 重新计算丢失分区
    - RDD容错：无需重新计算整个RDD
- 优点：惰性调用、管道化、避免同步等待、不需要保存中间结果、每次操作变得简单
- RDD之间的宽窄依赖关系
    - •窄依赖表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区（类似函数关系）
    - •宽依赖则表现为存在一个父RDD的一个分区对应一个子RDD的多个分区，存在shuffle
- stage划分
    - Spark 根据DAG 图中的RDD 依赖关系，把一个作业分成多个阶段。阶段划分的依据是窄依赖和宽依赖。对于宽依赖和窄依赖而言，窄依赖对于作业的优化很有利，宽依赖无法优化
    - 窄依赖可以实现“流水线”优化
    - 优化：遇到宽依赖就断开

常见的动作和转换（见后面rdd部分）

也可以在同一条代码中同时使用多个API，连续进行运算，称为链式操作，实现wordcount!!

```
 scala > val wordCounts = textFile.flatMap(line => line.split(“ ”)).map(word
=> (word,1)).reduceByKey((a,b) => a+b)

 scala > wordCounts.collect()
```

sbt打包，上机用过了。

针对性复习启动！！！

scala

- –val：是不可变的，在声明时就必须被初始化，而且初始化以后就不能再赋值
- –var：是可变的，声明的时候需要进行初始化，初始化以后还可以再次对其赋值

val定义的推广即为函数式编程：

> **函数式编程**
>
> 函数式编程是一种以数学函数为基础，强调不可变数据、纯函数和无副作用的编程范式，侧重于声明性编程和函数的组合与重用。省流版，变量是函数

函数式编程实例wordcount ⚠️背

```
1   import java.io.File
2   import scala.io.Source
3   import collection.mutable.Map //注意此处导入的可变的map
4   object WordCount {
5       def main(args: Array[String]) {
6           val dirfile = new File("testfiles")
7           val files  = dirfile.listFiles

8           val results = Map.empty[String,Int]
9           for(file <- files) {
10              val data = Source.fromFile(file)
11              val strs = data.getLines.flatMap{s => s.split(" ")} //注意此处使用flatmap，允许一个映射到多个，并扁平化
12              strs foreach {word =>
13                  if (results.contains(word))
14                  results(word)+=1 else results(word)=1
15                  }
16              }
17          results foreach{case (k,v) => println(s"$k:$v")}



18      }
19  }
```

行1-3：导入需要的类；

行6：建立一个File对象，这里假设当前文件夹下有一个testfiles文件夹，且里面包含若干文本文件；

行7：调用File对象的listFiles方法，得到其下所有文件对象构成的数组，files的类型为Array[java.io.File]；

行8：建立一个可变的空的映射（Map）对象results，保存统计结果。映射中的条目都是一个(key,value)键值对，其中，key是单词，value是单词出现的次数；

行9：通过for循环对文件对象进行循环，分别处理各个文件；

行10：从File对象建立Source对象方便文件的读取；

行11：getLines方法返回文件各行构成的迭代器对象，类型为Iterator[String]，flatMap进一步将每一行字符串拆分成单词，再返回所有这些单词构成的新字符串迭代器；

行12-15：对上述的字符串迭代器进行遍历，在匿名函数中，对于当前遍历到的某个单词，如果这个单

词以前已经统计过，就把映射results中以该单词为key的映射条目的value增加1。如果以前没有被统计过，则为这个单词新创建一个映射条目，只需要直接对相应的key进行赋值，就实现了添加新的映射条目；

行17：对Map对象results进行遍历，输出统计结果

for推导式

•for推导式：for结构可以在每次执行的时候创造一个值，然后将包含了所有产生值的集合作为for循环表达式的结果返回，集合的类型由生成器中的集合类型确定

for(变量 <- 表达式) yield {语句块}

```
scala> r = for( i <- Array(12,32,11,17,25,64) if i%2==0 ) yield { i }
```

数组

```
val intValueArr = new Array[Int](3)  //声明一个长度为3的整型数组，每个数组元素初始化为0
intValueArr(0) = 12 //给第1个数组元素赋值为12
intValueArr(1) = 45  //给第2个数组元素赋值为45
intValueArr(2) = 33 //给第3个数组元素赋值为33

val myStrArr = new Array[String](3) //声明一个长度为3的字符串数组，每个数组元素初始化为null
 myStrArr(0) = "BigData"
 myStrArr(1) = "Hadoop"
 myStrArr(2) = "Spark"
 for (i <- 0 to 2) println(myStrArr(i))
```

容器

Scala 的集合类库主要分为三个层次结构：

- 序列（Seq）： 表示有序集合，可以通过索引访问元素，包括列表（List）、数组（Array）、向量（Vector）等。
- 集（Set）： 表示无序集合，不包含重复元素，包括集合（Set）、有序集（SortedSet）等。
- 映射（Map）： 表示键值对集合，每个键关联一个值，包括映射（Map）、有序映射（SortedMap）等。

```
  -  列表（List）
      - 一种共享相同类型的==不可变==的对象序列。
```

```
    - •列表有头部和尾部的概念，可以分别使用head和tail方法来获取
- ==数组array，就他默认可变==
-  Vetor可以实现所有访问操作都是常数时间。
- Range类：一种特殊的、带索引的不可变数字等差序列。其包含的值为从给定起点按一定步长增长(减小)到指定终点的所有数值
- 集合(Set)
    - •不重复元素的容器（collection）
    - •缺省情况下创建的是==不可变集==
- 映射(Map)
    - 键是唯一的，但值不一定是唯一的。
```

迭代器（Iterator）不是一个容器，而是提供了按顺序访问容器元素的数据结构

关于容器是否可变的补充！！

```
val valList = List(1,2,3)

valList(1) = 3 //不可以运行，值不是int

valList = 4::valList //不可以运行，val不可以重定向


var varList = List(1,2,3)

varList(1) = 3 //不可以运行，值不是int

varList = 4::varList //可以运行


val valArray = Array(1,2)

valArray(1) = 3 //可以运行

valArray = valArray::4 //不可以运行，val不可以重定向
```

创建类

```
//定义一个类
 class Counter {
   var value = 0
   def increment(step:Int):Unit = { value += step}
   def current():Int = {value}
 }
```

```
//使用new关键字创建一个类的实例
 val myCounter = new Counter
 myCounter.value = 5 //访问字段
 myCounter.increment(3) //调用方法
 println(myCounter.current) //调用无参数方法时，可以省略方法名后的括号

//def方法
class Counter {
      var value = 0
      def increment(step:Int):Unit = { value += step }
      def current:Int = value
      def getValue():Int = value
}

 val c= new Counter
 c increment 5 //中缀调用法，等价于 c.increment(5)
 c.getValue()   //定义时带括号
 c.getValue   //定义时带括号
 c.current     //定义时不带括号
 c.current()  //错误，因为定义的时候没有带括号
```

•当方法的返回结果类型可以从最后的表达式推断出时，可以省略结果类型
•如果方法返回类型为Unit，可以同时省略返回结果类型和等号，但不能省略大括号

特质

```
//一个特质
 trait Flyable {
          var maxFlyHeight:Int  //抽象字段
          def fly() //抽象方法
          def breathe(){ //具体的方法
                println("I can breathe")
         }
  }

//混入一个类中
//注意对特质的抽象字段与抽象方法的赋值
 class Bird(flyHeight:Int) extends Flyable{
      var maxFlyHeight:Int = flyHeight  //重载特质的抽象字段
      def fly(){
             printf("I can fly at the height of %d.",maxFlyHeight)
       } //重载特质的抽象方法
```

```
 }
```

特质多重混入重要例子 ⚠️背

```
trait Flyable {
       var maxFlyHeight:Int  //抽象字段
       def fly() //抽象方法
       def breathe(){ //具体的方法
             println("I can breathe")
       }
 }

trait HasLegs {
       val legs:Int   //抽象字段
       def move(){printf("I can walk with %d legs",legs)}
}

class Animal(val category:String){
       def info(){println("This is a "+category)}

class Bird(flyHeight:Int) extends Animal("Bird") with Flyable with HasLegs{
         var maxFlyHeight:Int = flyHeight //重载特质的抽象字段
         val legs=2 //重载特质的抽象字段
         def fly(){
               printf("I can fly at the height of %d",maxFlyHeight)
         }//重载特质的抽象方法！！不用重载具体方法
}
```

抽象类

如果一个类包含没有实现的成员，则必须使用abstract关键词进行修饰，定义为抽象类

```
abstract class Car(val name:String) {
    val carBrand:String //字段没有初始化值，就是一个抽象字段
    def info() //抽象方法
    def greeting() {
        println("Welcome to my car!")
    }
}
```

Note

关于上面的定义，说明几点：
（1）定义一个抽象类，需要使用关键字abstract
（2）定义一个抽象类的抽象方法，不需要关键字abstract，只要把方法体空着，不写方法体就可以
（3）抽象类中定义的字段，只要没有给出初始化值，就表示是一个抽象字段，但是，抽象字段必须要声明类型，否则编译会报错

apply函数

```
class TestApplyClass {
    def apply(param: String){
         println("apply method called: " + param)
    }
}

val myObject = new TestApplyClass
myObject(“Hello Apply”)//自动调用类中定义的apply方法！！
myObject.apply(“Hello Apply”) //手动调用apply方法
```

伴生对象的apply工厂方法 ⚠️背

```
//伴生对象中的apply方法：将所有类的构造方法以apply方法的形式定义在伴生对象中，这样伴生对象就像生成类实例的工厂，而这些apply方法也被称为工厂方法

class Car(name: String) {
    def info() {
        println("Car name is "+ name)
    }
}

object Car {
    def apply(name: String) = new Car(name) //调用伴生类Car的构造方法
}

object MyTestApply{
    def main (args: Array[String]) {
    val mycar = Car("BMW") //调用伴生对象中的apply方法！！new写到了apply里面了
    mycar.info() //输出结果为“Car name is BMW”
    }
}
```

针对容器的规约操作

reduce

```
scala> val list =List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala>  list.reduce(_ + _) //将列表元素累加，使用了占位符语法
 Int = 15
scala>  list.reduce(_ * _) //将列表元素连乘
Int = 120

scala> val list = List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala> list reduceLeft {_-_} //注意顺序1-2 -3  -4  -5
 Int = -13
scala> list reduceRight {_-_} //4-5,1 2 3 -1,3--1,1 2 4,2-4,1 -2，3
 Int = 3
```

fold

```
scala> val list =List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala> list.fold(10)(_*_)
 Int = 1200
scala> (list fold 10)(_*_) //fold的中缀调用写法
 Int = 1200
scala> (list foldLeft 10)(_-_)//计算顺序(((((10-1)-2)-3)-4)-5)，多给一个初值的区别
 Int = -5
scala> (list foldRight 10)(_-_) //计算顺序(1-(2-(3-(4-(5-10)))))
 Int = -7
```

RDD操作

操作	含义
filter(func)	筛选出满足函数func的元素，并返回一个新的数据集
map(func)	将每个元素传递到函数func中，并将结果返回为一个新的数据集
flatMap(func)	与map()相似，但每个输入元素都可以映射到0或多个输出结果
groupByKey()	应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)形式的数据集
reduceByKey(func)	应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中每个值是将每个key传递到函数func中进行聚合后的结果

操作	含义
count()	返回数据集中的元素个数
collect()	以数组的形式返回数据集中的所有元素
first()	返回数据集中的第一个元素
take(n)	以数组的形式返回数据集中的前n个元素
reduce(func)	通过函数func（输入两个参数并返回一个值）聚合数据集中的元素
foreach(func)	将数据集中的每个元素传递到函数func中运行

转换**记忆口诀：**

- “映射与过滤，分组与合并”

- 映射（`map`，`flatMap`），过滤（`filter`），分组（`groupBy`，`reduceByKey`），合并（`union`，`join`）。

行动**记忆口诀：**（离开rdd格式才行）

- **“收集与统计，保存与查看”**
  - 收集（`collect`，`take`），统计（`count`，`reduce`），保存（`saveAsTextFile`），查看（`foreach`）。

```
//转化+行动
val lines=sc.textFile(“file:///home/hadoop/module/spark/word.txt”)
val words = lines.flatMap(line => line.split (“ ”)) //转化
val wordPairs = words.map( word => (word,1)) //转化
val wordCount = wordPairs.reduceByKey((a,b)=>a+b) //行动


//惰性机制
```

```
scala> val list = List("Hadoop","Spark","Hive")
scala> val rdd = sc.parallelize(list)
scala> println(rdd.count())  //行动操作，触发一次真正从头到尾的计算
scala> rdd.map(s => s.length).collect.foreach(println) //重新！从头计算
```

上机作业1答案（伴生对象与模式匹配）⚠️背

练习 1：实现一个简单的类及伴生对象

1. 实现一个名字为Student的类，这个类包括属性：Name:String,Age:Int,ID:String

包括方法updateName更新姓名,updateAge更新年龄,getInfo打印学生全部信息

2. 定义类后，以下能够成功运行

```
val stu1=new Student("Alice",20,"S001")
stu1.updateName("Bob")
stu1.updateAge(31)
stu1.getInfo
```

3. 使用一个同名的单例对象，包含方法apply
4. 定义单例对象后，能够成功运行以下语句

```
val stu2 = Student("Tom",21,"S002")
```

解答

```
// 定义类 Student
class Student(var name: String, var age: Int, val id: String) {

  // 更新姓名的方法
  def updateName(newName: String): Unit = {
    name = newName
  }

  // 更新年龄的方法
  def updateAge(newAge: Int): Unit = {
    age = newAge
  }
```

```
  // 打印学生信息的方法
  def getInfo: Unit = {
    println(s"Name: $name, Age: $age, ID: $id")
  }
}

// 定义伴生对象 Student
object Student {
  // 使用 apply 方法实现便捷的对象创建
  def apply(name: String, age: Int, id: String): Student = {
    new Student(name, age, id)
  }
}

// 测试代码
val stu1 = new Student("Alice", 20, "S001")
stu1.updateName("Bob")
stu1.updateAge(31)
stu1.getInfo   // 输出: Name: Bob, Age: 31, ID: S001

val stu2 = Student("Tom", 21, "S002") // 通过伴生对象创建
stu2.getInfo   // 输出: Name: Tom, Age: 21, ID: S002
```

练习 2：实现一个简单的计算器

要求：

1. 创建一个 Calculator 对象，包含一个 calculate 方法，接收两个整数和一个表示操作的字符（+, -, *, /）。
2. 使用模式匹配来根据不同的操作符执行相应的运算，并返回计算结果。
3. 如果传入的操作符不合法，则返回 "Invalid operation"。
   定义完成后可以实现以下语句：

```
Calculator.caculate(1,2,'+')
Calculator.caculate(10,2,'-')
Calculator.caculate(3,5,'*')
Calculator.caculate(20,5,'/')
```

解答

```
object Calculator {
  def calculate(x: Int, y: Int, operation: Char): Any = {
```

```
    operation match {
      case '+' => x + y
      case '-' => x - y
      case '*' => x * y
      case '/' => if (y != 0) x / y else "Division by zero"
      case _   => "Invalid operation"
    }
  }
}

// 测试代码
println(Calculator.calculate(10, 5, '+'))  // 输出: 15
println(Calculator.calculate(10, 5, '-'))  // 输出: 5
println(Calculator.calculate(10, 5, '*'))  // 输出: 50
println(Calculator.calculate(10, 5, '/'))  // 输出: 2
println(Calculator.calculate(10, 0, '/'))  // 输出: Division by zero
println(Calculator.calculate(10, 5, '%'))  //
```

上机作业2 3答案 breeze包基本操作 ⚠️背

找不到“image-20241227110125739.png”。

```
// Q1. Create a matrix A by using a diagonal matrix of vector 'a' and a ones matrix
import breeze.linalg._ // lin + alg
val a = DenseVector(1.0, 3.0, 5.0, 7.0, 9.0, 7.0, 5.0, 3.0, 1.0)

// Create a 9x9 matrix of ones
val OneMat = DenseMatrix.ones[Double](9, 9)

// Calculate A = diag(a) * OneMat + OneMat * diag(a)
// a is a vector
val A = diag(a) * OneMat + OneMat * diag(a)
```

```
// Q2. Another version of matrix B calculation using diag(a) and OneMat
import breeze.linalg._
val a = DenseVector(1.0, 3.0, 5.0, 7.0, 9.0, 7.0, 5.0, 3.0, 1.0)

// Create a 9x9 matrix of ones
val OneMat = DenseMatrix.ones[Double](9, 9)

// Calculate B = diag(a) * OneMat * diag(a)
val B = diag(a) * OneMat * diag(a)
```

```scala
// Q3. Center a matrix by subtracting the column mean
import breeze.linalg._

// Define a matrix C
val C = DenseMatrix((-1.0, -2.0, -3.0),
                    (4.0, 5.0, 6.0),
                    (7.0, 8.0, 9.0),
                    (6.0, 5.0, 4.0),
                    (-3.0, -2.0, -1.0))

// Calculate the mean of each column
// Mean is a three element vector
// Understand by replace * with number 1,2,3,4....
//Trick// :: means all; and * means each
// （行，列）
val Mean = sum(C(::, *)) / C.rows.toDouble

// Subtract the mean from each column to center the matrix
val NewMat = C(*, ::) - Mean.t
```

`*` 操作

`sum(C(*, ::))` 是按行操作，结果是行向量；
`sum(C(::, *))` 是按列操作，结果是列向量；

```scala
// Q4. Generate random samples from a multivariate normal distribution and calculate PDF
import breeze.linalg._
import breeze.stats.distributions._

// Create a 3D Multivariate Normal distribution with mean vector and covariance matrix
val normal3D = new MultivariateGaussian(DenseVector(1.0, 2.0, 3.0),
                                        DenseMatrix((5.0, 2.0, 3.0),
                                                    (2.0, 5.0, 4.0),
                                                    (3.0, 4.0, 5.0)))

// Generate 100 random samples from the distribution
val RandomNumbers = normal3D.sample(100)

// Calculate the probability density function (PDF) for a specific point
val pdf_sample = normal3D.pdf(DenseVector(1.0, 2.0, 3.0))
```

```scala
// Q5. Generate random variables and transform them using a cubic root function
import java.util.concurrent.ThreadLocalRandom
import breeze.numerics._

// Initialize random number generator
val r = ThreadLocalRandom.current

// Generate 1000 random numbers, transform them, and store them in an array
val RVs = (1 to 1000).map(x => r.nextDouble) // Generate random numbers between 0 and 1
  .map(x => 2.0 * x - 1.0) // Rescale to range [-1, 1]
  .map(r => if (r > 0) pow(r, 1.0 / 3.0) else -pow(-1.0 * r, 1.0 / 3.0)) // Apply cubic root transformation
  .toArray[Double]
```

```scala
// Q6. Sample from a 2D multivariate normal distribution, compute expectations and matrix operations
import breeze.numerics._
import breeze.stats.distributions._
import breeze.linalg._

// 实例化分布
val normal2D = new MultivariateGaussian(DenseVector(10.0, -10.0),
                                        DenseMatrix((1.0, 0.5),
                                                    (0.5, 1.0)))
// 采样
val RandomNumbers = normal2D.sample(100000).toArray

// rdd化
val data = sc.parallelize(RandomNumbers)//**SparkContext**

// Compute the expected value of x^2 * y^2
val EX2Y2 = data.map(x => x(0) * x(0) * x(1) * x(1))
  .reduce((x, y) => x + y) / data.collect().size.toDouble
  //注意collect()

// Compute the empirical covariance matrix
val Q2bE = data.map(x => x * x.t)
  .reduce((x, y) => x + y) / data.collect().size.toDouble

// Compute the empirical covariance matrix and apply a transformation
val Q2cE = data.map(x => x * x.t)
  .map(x => DenseMatrix((sin(x(0, 0)), cos(x(0, 1))),
```

```
                         (cos(x(1, 0)), sin(x(1, 1)))))
   .reduce((x, y) => x + y) / data.collect().size.toDouble
```

---

breeze进阶

线性模型 ⚠️背

```
import java.util.concurrent.ThreadLocalRandom
//设置产生数据的参数并允许各个计算节点可以读取其取值
val RowSize = sc.broadcast(200)
val ColumnSize = sc.broadcast(5)
val RowLength = sc.broadcast(4)
val ColumnLength = sc.broadcast(1000)
val NonZeroLength = 10
val p = ColumnSize.value * ColumnLength.value

val beta = (1 to p).map(_.toDouble).toArray[Double].map(i =>
{if(i<NonZeroLength+1) 2.0 else 0.0})

//在整个Spark系统中广播beta系数，从而使得每个计算节点都可以读取变量MyBeta中的值
val MyBeta = sc.broadcast(beta)

val sigma = 1.0
//在整个Spark系统中广播Sigma系数，从而使得每个计算节点都可以读取变量Sigma中的值
val Sigma = sc.broadcast(sigma)

var indices = 0 until RowLength.value
var ParallelIndices = sc.parallelize(indices, indices.length)

//产生数据
var lines = ParallelIndices.map(s => { //一个巨大的map
    val r = ThreadLocalRandom.current
    //nextGaussian是新产生的类对象r中的函数，可以产生服从标准正态分布的随机数
    def rn(n: Int) = (0 until n).map(x => r.nextGaussian).toArray[Double]
    //读取MyBeta和Sigma中的值
    val beta = MyBeta.value
    val sigma = Sigma.value
    val rowsize = RowSize.value
    val columnsize = ColumnSize.value
    val columnlength = ColumnLength.value
    val lines = new Array[String](rowsize)
    val p = columnsize * columnlength
```

```
    //rowsize是要产生的y的行数
    for(i <- 0 until rowsize)
    {
         var line = "";
         var y = 0.0;


         for(j <- 0 until columnlength) //每一列的长度
         {
             var x = rn(columnsize)

             for(k <- 0 until columnsize) y+=beta(j*columnsize + k)*x(k)

             var segment = x.map("%.4f" format _).reduce(_+" "+_)
             line = line+","+segment
         }
        //加上一个误差项
         y+= sigma*r.nextGaussian

         lines(i) = "%.4f".format(y) + line + "\n"
    } //巨大的map结束
    lines.reduce(_+_)
})

import scala.sys.process._

var cmd = "hdfs dfs -ls /"

cmd.!!.foreach{print}

lines.saveAsTextFile("/SimData")

cmd.!!.foreach{print}

val lines_read = sc.textFile("/SimData/part-00000")

import breeze.linalg._

val transLines = lines_read.take(200).map( s => {
    val rowsize = RowSize.value
    val columnsize = ColumnSize.value
    val columnlength = ColumnLength.value
    val p = columnsize * columnlength
    val Y = DenseVector(s.map(_.split(",")(0).toDouble))
    val X = s.map(_.split(",").drop(1).map(_.split(" ").map(_.toDouble)))
    (Y,X)
```

```
  }
  )

  val Y = lines_read.map(_.split(",")(0).toDouble)

  val X = lines_read.map(_.split(",").drop(1).flatMap(_.split("
  ").map(_.toDouble)))
```

slice切片 ⚠️背

```
import java.util.concurrent.ThreadLocalRandom

def targetDensity(x: Double): Double = {
  if (x >= 0 && x <= 2){
    math.pow(2 - x, 2) * math.pow(x * x * x + 1, 3)
    } else
   {
    1e-8
   }
}
  // 定义采样函数
def sliceSampling(iterations: Int, width: Double): List[Double] = {
   var samples: List[Double] = List.empty
   var current = 1.0  // 初始样本x的位置
   val rand = ThreadLocalRandom.current

   for (_ <- 0 until iterations) { //产生多少个样本点
      val height = rand.nextDouble*targetDensity(current) //高度由目标pdf*随机
数
      var interval = (current - width * rand.nextDouble(), current + width *
rand.nextDouble()) //区间初始化是一个随机二元组

      var newX = interval._1 + (interval._2 - interval._1) *
rand.nextDouble() //在区间内随机选择新点
      var newY = targetDensity(newX) //计算新点的密度值

    //当新点的密度值大于h值直接采用，当新点的密度值小于h值时候缩小采样范围，重新采样，直至
采样到合适样本
      while (newY < height) {
        if (newX > current) interval = (interval._1, newX)
        else interval = (newX, interval._2)

        newX = interval._1 + (interval._2 - interval._1) * rand.nextDouble()
        newY = targetDensity(newX)
      }
```

```
      current = newX
      samples = current :: samples //将新样本加入到sample中
    }
    samples.reverse
}


val iterations = 10000   // 迭代次数
val width = 0.1          // 切片的宽度

val samples = sliceSampling(iterations, width)

println(samples.take(10).mkString(", "))
```

MCMC ⚠️背

```
import breeze.linalg._
import breeze.numerics._
import breeze.stats.distributions._
import java.util.concurrent.ThreadLocalRandom


def targetDensity(x: Double): Double = {
  if (x >= 0 && x<=2){
    math.pow(2 - x, 2) * math.pow(x * x * x + 1, 3)
    } else
    {
    1e-8
    }
}


def metropolisHastings(targetDensity: Double => Double, initial: Double,
iterations: Int, proposalStd: Double): DenseVector[Double] = {
  val samples = DenseVector.zeros[Double](iterations)
  var current = initial
  val rand = ThreadLocalRandom.current

  for (i <- 0 until iterations) {
   //产生一个提议样本
    val proposal = current + rand.nextGaussian() * proposalStd
   // 计算提议样本的接受率
    val acceptanceRatio = targetDensity(proposal) / targetDensity(current)
```

```
    //如果提议样本接受率高于1或者“比一个随机数大”，新样本赋值为提议样本
    if (acceptanceRatio >= 1 || rand.nextDouble() < acceptanceRatio) {
      current = proposal
    }

    samples(i) = current
  }

  samples
}


val initialSample = 0.0
val iterations = 10000
val proposalStd = 0.5

val samples = metropolisHastings(targetDensity, initialSample, iterations,
proposalStd)

println(samples(0 to 10))
```

接受拒绝??非官方

```
import java.util.concurrent.ThreadLocalRandom
import org.apache.commons.math3.special.Gamma._

val result = Gamma.gamma(5.0)
println(s"Gamma(5.0) = $result")  // 输出: 24.0

def targetDistribution(x: Double): Double = {
  val alpha = 2 // Beta分布的alpha参数
  val beta = 4 // Beta分布的beta参数
  val normalization = math.pow(math.gamma(alpha) * gamma(beta) / gamma(alpha
+ beta), -1) // Beta分布的归一化常数
  normalization * math.pow(x, alpha - 1) * math.pow(1 - x, beta - 1) // Beta
分布的概率密度函数
}

def samplingDistribution(x: Double): Double = {
  20 * x * math.pow(1 - x, 3) // 容易采样的分布的概率密度函数
}
```

```scala
def rejectionSampling(iterations: Int): Double = {
  val K = 20 // 常数K，使得目标分布在容易采样分布范围内的上界
  var acceptedSamples = 0 // 记录被接受的样本数量
  val r = ThreadLocalRandom.current
  for (_ <- 0 until iterations) {
    val x = r.nextDouble // 从容易采样的分布中生成样本，在 [0, 1] 区间上均匀分布
    val y = r.nextDouble * K // 从均匀分布中生成 [0, K] 区间上的随机数
    if (y <= targetDistribution(x) / samplingDistribution(x)) {
      acceptedSamples += 1 // 根据接受概率决定是否接受样本
    }
  }

 // 返回被接受的样本数量的比例，这个比例乘以总样本数量就是目标分布的近似随机数
  acceptedSamples.toDouble / iterations
}
val iterations = 1000 // 迭代次数
val result = rejectionSampling(iterations)
println("Generated random number from Beta(2, 4) distribution using
rejection sampling: " + result)
```