LECUTURE 15 SPARK

1. RDD概述

- 对两类算法的处理很有效：
  - 迭代算法
  - 交互性数据挖掘
- 提供粗粒度的transformation
- RDD定义：一种数据集的抽象，具有如下特征：
  - 为程序猿提供集群上的基于内存的计算
  - 平行结构
  - 可容错
  - 只读
- 程序猿可以指定持久化策略、数据分区策略、使用丰富的运算符
- 实现容错：RDDs将transformation写入日志（也即建立了数据集的整个lineage）
- RDDs只适合于批处理应用，不适用于异步、细粒度更新的数据

2. RDD

- RDD起源：

  能够产生RDD的操作被称为**Transformation**，这些Transformation都是粗粒度的操作。注意读RDD仍然可以是细粒度的。一个RDD就是一个对象，Transformation是对象上的方法
- RDD无需物化，因为可以通过log来推出所有的RDD；但一个应用程序不能引用RDD，若该RDD不能从log中推出来
- （？？2.2.1）原始数据不会存入内存，在collect之后才会。
- RDD由**partition** 分区 组成，每个分区都是不可再分单元。当某个RDD丢失了，Spark会通过lineage来复原该RDD。
- **action：** 是满足如下特征的一些操作，可以用来指定哪一些RDD未来还要使用：

- 向应用程序返回数据的操作
- 将数据导出到存储系统中的操作

  包括：

  - count：统计数据集中数据数目
  - collect：返回数据集中的元素
  - save：将数据集输出到存储系统中
  - persist：

- 用户可自定义持久化策略，默认将数据存在内存中，内存不足时存入磁盘

3. Spark的接口

- 使用的接口与Scala类似
- RDD是静态类型对象，可以指定元素，如RDD[Int]
- 具体接口如图：

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

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

函数参数为闭包！右侧为函数的效果

- 当调用Transformation时，实际上是在创建一个lineage，而不是在实际计算。当调用action时，才会根据lineage进行实际的计算

    - 调用colloect则会通过lineage计算，并且输出实际的内容

4. RDDs表达 representation

- 表达一个RDD，需要以下信息：
    - RDD所包含的分区集合
    - 对父RDD的依赖集合，依赖又可以分为：
        - narrow dependency：父RDD中每个partition最多只被子RDD使用一次
        - wide dependency：父RDD中每个partition可被子RDD使用多次
    - 基于父RDD计算出当前RDD的函数
    - 元数据（分区模式和数据放置方式）
- narrow/wide dependency比较
    - Narrow dependency方便流水线执行，并且不需要shuffle之类的操作
    - Narrow dependency恢复方便，只需对失去的分区的子分区按照lineage重新执行即可

5. 实现

- Spark运行在Mesos集群管理器上，Mesos集群管理器可以共享资源。每一个Spark程序/program都作为独立Mesos 应用，具有driver和worker。不同的Spark program之间共享数据通过Mesos完成

5.1. 任务调度

- 调度器的设计需要考虑到哪些RDD已经在内存中。
- stage的概念：
    - stage由narrow dependency的Transformation组成，这些narrow-dependency-transformation可以流水线执行
    - stage的边界为：
        - wide dependency 的 transformation
        - 已经计算完成的partition（这些partition可以短路父RDDs的相关操作）

    - 调度器需要构建一个以stage为结点的DAG，而后根据拓扑序进行执行，得到目标RDD
- 调度器使用延迟调度，会将任务分配给含有partition的机器
- 当任务失败时，若stage仍然可用，则只需要在其他机器上重新运行即可；反之，需要重新提交任务，并行地计算丢失的分区。
- 调度器的故障是不可接受的！！
- 当使用lookup随机访问一个RDD中的一个元素时，若该元素所在的Partition丢失，则node需要要求调度器计算该partition

5.2. Scala解释器改装

- 每一行代码都是一个类
- 改装如下：
    - class shipping：worker会抓取类的代码，而后通过HTTP协议在各个worker之间传播
    - modified code generation：（？？？）若只通过类来传递数据，当遇到闭包时，则会丢失闭包所绑定的数据，因此需要改装解释器。

6. PageRank例子

```
val lines = spark.read.textFile("in").rdd
lines.collect()
val links1 = lines.map{s => val parts = s.split("\\s+");
(parts(0),parts(1))}
val links2 = links1.distinct()
val links3 = links2.groupByKey()
val links4 = links3.cache()
val ranks = links4.mapValues(v => 1.0)
val jj = links4.join(ranks)
val contribs = jj.values.flatMap{ case (urls, rank) => urls.map(url =>
(url, rank / ruls.size)}
val ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
```

- 第一行是一个Transformation，本质是在创建lineage，没有进行实际的执行；在第二行中，collect被调用（collect是一个action）。此时，Spark会选出一些worker来处理partition，而后编译lineage成为JAVA字节码，发送给各个worker进行执行
- 第三行在调用collect后的执行结果：

```
scala> links1.collect()
res2: Array[(String, String)] = Array((u1,u3), (u1,u1), (u2,u3), (u2,u2), (u3,u1
))
```

- 第五行本质上是对links1进行去重。这个过程需要将所有一样的元素找出来，用一个元素来代替。这里需要worker之间的协作，需要引入shuffle，调用collect后执行结果如下：

  ```
  scala> links2.collect()
  res4: Array[(String, String)] = Array((u1,u1), (u1,u3), (u3,u1), (u2,u2), (u2,u3
  ))
  ```

  注意，distinct实际的shuffle过程可能是按照key分区，而后，对相同的key进行distinct操作（这一过程为narrow dependency，不需要shuffle）

- 第六行执行结果：

  ```
  scala> links3.collect()
  res6: Array[(String, Iterable[String])] = Array((u2,CompactBuffer(u2, u3)), (u3,
  CompactBuffer(u1)), (u1,CompactBuffer(u1, u3)))
  ```

  这里groupByKey需要进行shuffle，但由于上一步distinct已经按照key进行分区，因此 这里理论上不需要shuffle（但是单一的groupByKey是肯定要shuffle的），但是这一点需要Spark的优化（如果Spark没有发现这里不需要shuffle，可能仍然会进行shuffle的步骤）

- 第7行中，对数据进行了持久化处理（存在内存中）。在未进行持久化处理之前，所有的数据都需要从头算起（Spark不保存中间结果），因此为了避免多次重复计算（尤其是在数据集非常大的情况下），可以对数据进行持久化处理

- 第8行创建了一个新的rdd，注意mapValues的定义。此处使用了原来links4数据集中数据的key值，将value值全部映射为1.0（排名）

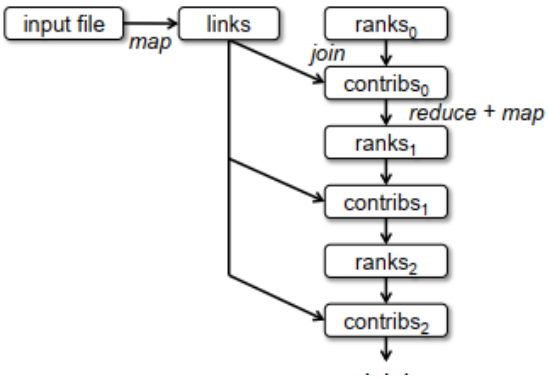
- 第9行join将ranks与links4进行连接操作，因此目前rdd中一条数据为一个key，和该key所指向的网页的列表 + 暂时排名

  ```
  scala> jj.collect()
  res8: Array[(String, (Iterable[String], Double))] = Array((u2,(CompactBuffer(u2,
   u3),1.0)), (u3,(CompactBuffer(u1),1.0)), (u1,(CompactBuffer(u1, u3),1.0)))
  ```

- 第10行在计算 网站对它指向的结点的贡献。具体公式为$\frac{\text{本网站的排名}}{\text{本网站能指向的网站数目}}$

- 第12行首先进行一个reduce工作，将所有key一样的数据聚集起来，将其value相加，最终返回：（key，value sum）；而后map工作，将value sum（记为V）进行$0.85V+0.15$的处理

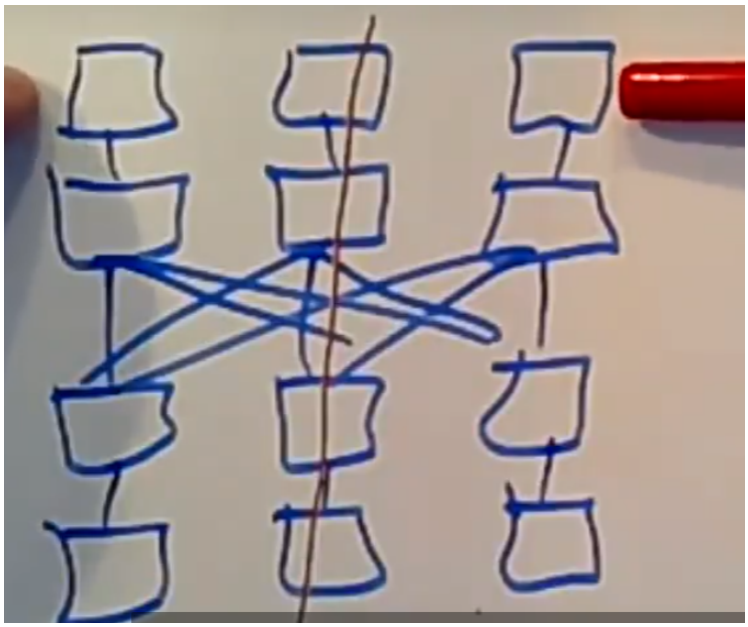
- 最后多次迭代，lineage如图示：



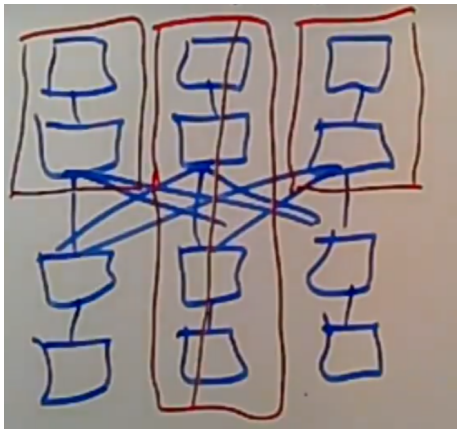
此处的links即为links4，也就是cache之后的结果

7. 容错

- Spark假定HDFS会使用复制状态机
- 一个worker会负责多个partition，当一个worker故障时，只需要将其已经丢失的partition分配给其他worker重算 即可
- **一个tricky的情况：**



此时，中间的worker故障，为了获取数据必须从头开始算，但是第三个结点遇到了困难，其数据需要第二层左右两边结点的数据。由于过程中没有显式调用cache，**Spark不会保存中间数据**，因此实际需要重新计算的数据包括下图红框部分：



在非常极端的情况下，一个worker的故障，所需要重新计算的RDD数目非常巨大，因此Spark引入了检查点机制。通过检查点，可以将数据写入HDFS，这样就可以避免数据丢失。（注意，cache只会将数据持久化到内存中，并不能保证数据是真的可用的）