ctrl+alt + B 看子类

ctrl+ H 查看类的层级结构

1.RDD 的依赖关系

1.1.综述

RDD 可以从本地集合并行化，从外部文件系统，也可以从其他的 RDD 转换得到。

能从其他 RDD 通过 transformation 创建新的 RDD 的原因是 rdd 之间有依赖关系（Dependency 代表了 RDD 之间的依赖关系，即血缘（Lineage））

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

1.2.划分依赖的背景

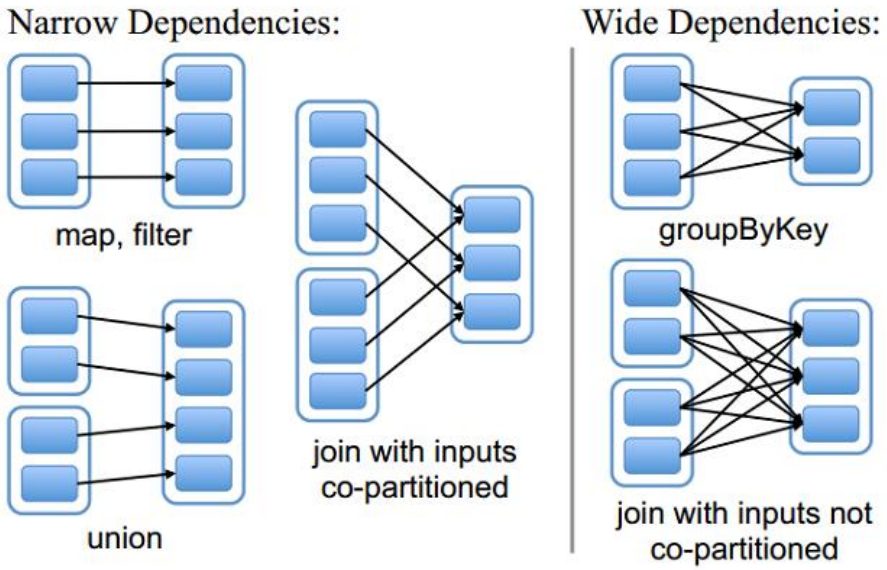
1，从计算过程来看，窄依赖是数据以管道方式经一系列计算操作可以运行在了一个集群节点上，如（map、filter 等），宽依赖则可能需要将数据通过跨节点传递后运行（如 groupByKey），有点类似于 MR 的 shuffle 过程。

2，从失败恢复来看，窄依赖的失败恢复起来更高效，因为它只需找到父 RDD 的一个对应分区即可，而且可以在不同节点上并行计算做恢复；宽依赖则牵涉到父 RDD 的多个分区，恢复起来相对复杂些。

3，综上，引入了一个新的概念 Stage。Stage 可以简单理解为是由一组 RDD 组成的可进行优化的执行计划。如果 RDD 的衍生关系都是窄依赖，则可放在同一个 Stage 中运行，若 RDD 的依赖关系为宽依赖，则要划分到不同的 Stage。这样 Spark 在执行作业时，会按照

Stage 的划分，生成一个完整的最优的执行计划。

1.3.划分依赖的依据



划分宽依赖和窄依赖的关键点在： 分区的依赖关系

也就是说父 rdd 的一个分区的数据，是给子 rdd 的一个分区，还是给子 RDD 的所有分区。

父 RDD 的每一个分区，是被一个子 RDD 的一个分区依赖（一对一），这种就是窄依赖

如果父 RDD 的每一个分区，被子 RDD 的多个分区所依赖（一对多），就是宽依赖。

一旦有宽依赖，就会发生数据的 shuffle

窄依赖不会发生数据的 shuffle，宽依赖才会进行数据的 shuffle。发生了数据 shuffle，就会进行阶段切分。（也就是切分 stage）

1.4. 窄依赖

窄依赖指的是父 RDD 的一个分区，被子 RDD 的一个分区所依赖（一对一）

输入输出一对一的算子，且结果 RDD 的分区结构不变。主要是 map/flatmap/filter

输入输出一对一的算子，但结果 RDD 的分区结构发生了变化，如 union/coalesce

总结：窄依赖我们形象的比喻为**独生子女**

1.5. 宽依赖

宽依赖指的是父 RDD 的一个分区，被子 RDD 的多个分区依赖。（一对多）

对单个 RDD 基于 key 进行重组和归并，如 groupByKey，reduceByKey 等

对两个 RDD 基于 key 进行 join 和重组，如 join(父 RDD 不是 hash-partitioned)

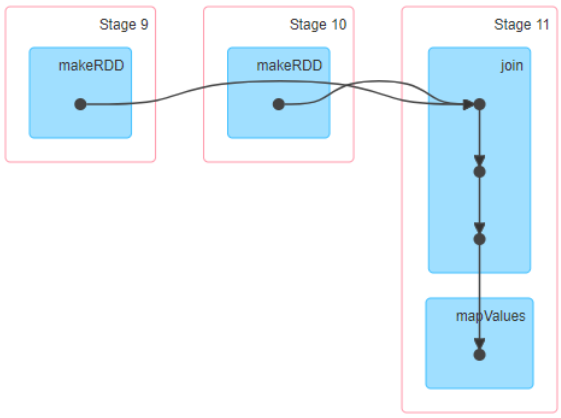
需要进行分区，如 partitionBy

总结：宽依赖我们形象的比喻为**超生**

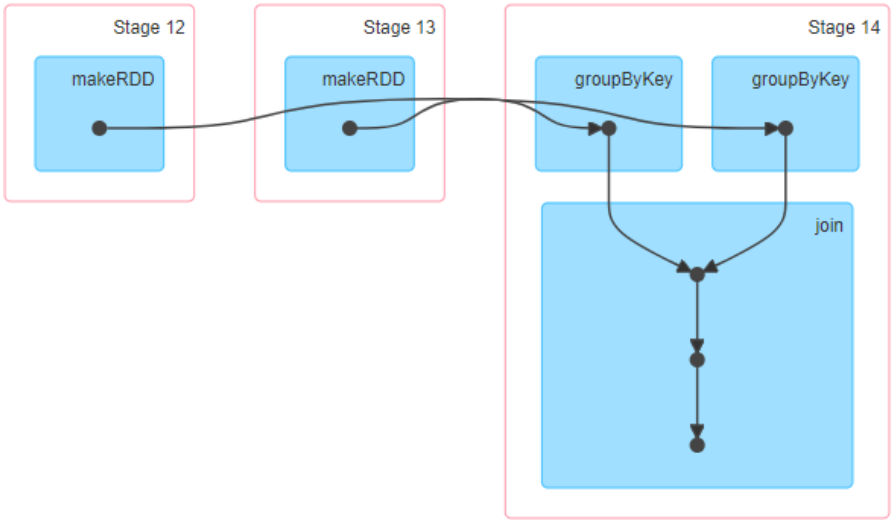
普通的 join：

```
val rdd1 = sc.makeRDD(List(("a",1),("b",2),("c",3))

val rdd2 = sc.makeRDD(List(("a",11),("b",12),("c",13))

val rdd3 = rdd1 join rdd2

rdd3.collect
```

宽依赖



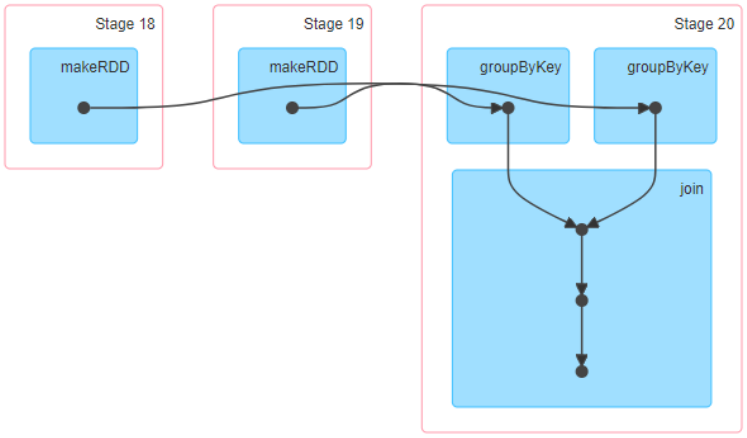
特殊的 join，窄依赖




```
val rdd1 = sc.makeRDD(List(("a",1),("b",2),("c",3)),3)

val rdd2 = sc.makeRDD(List(("a1",1),("b",2),("c",3)),3)

val rdd1group = rdd1.groupByKey()

val rdd2group = rdd2.groupByKey()

val rdd5 = rdd1group join rdd2group
```

```
rdd13.collect
```

条件：进行 join 的两个 rdd 必须经过了 shuffle（必须对 key 的分组类的算子操作），而且有相同的分区个数。



从数据角度来查看宽依赖：

```
val rdd2 = sc.makeRDD(List(11,12,13,14,15,16),2)

val rdd3 = rdd2.zipWithIndex

val f =(i:Int,it:Iterator[(Int,Long)]) =>

        it.map(t => s"part:$i,value:$t")

// 查看数据

rdd3.mapPartitionsWithIndex(f).collect



// 调用 groupByKey 方法，宽依赖触发 shuffle

val rdd4 = rdd3.groupByKey()
```

```
val f1 = (i:Int,it:Iterator[(Int,Iterable[Long])]) =>
        it.map(t=> s"$i,${t._1}")
// 查看数据
rdd4.mapPartitionsWithIndex(f1).collect
```

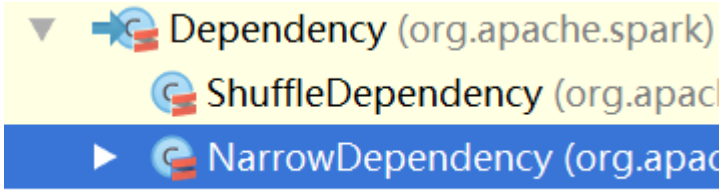
对比两次打印的结果数据

1.6. 依赖与 stage 划分

Spark 将窄依赖划分在同一个 stage 中，因为它们可以进行流水线计算。而宽依赖往往意味着 shuffle 操作，这也是 Spark 划分 stage 的主要边界。一个 Stage 的开始就是从外部存储或者 shuffle 结果中读取数据；一个 Stage 的结束就是发生 shuffle 或者生成结果时。

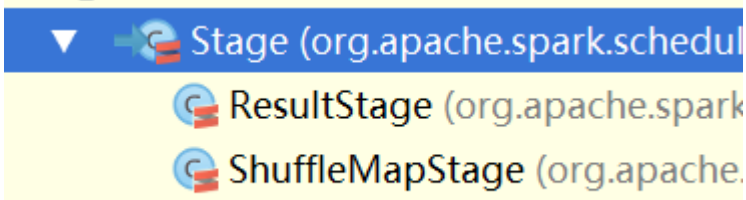
只要发生 shuffle，就会有 stage 的划分。

有两种类型的依赖关系：



Stage 的类别：

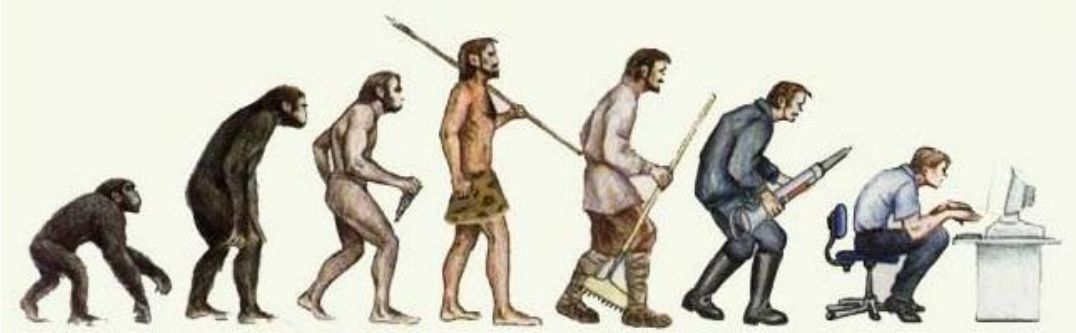
一个 job 有一个或多个 stage，一个 ResultStage + 多个 ShuffleMapStage



1.7. 依赖与 RDD 容错

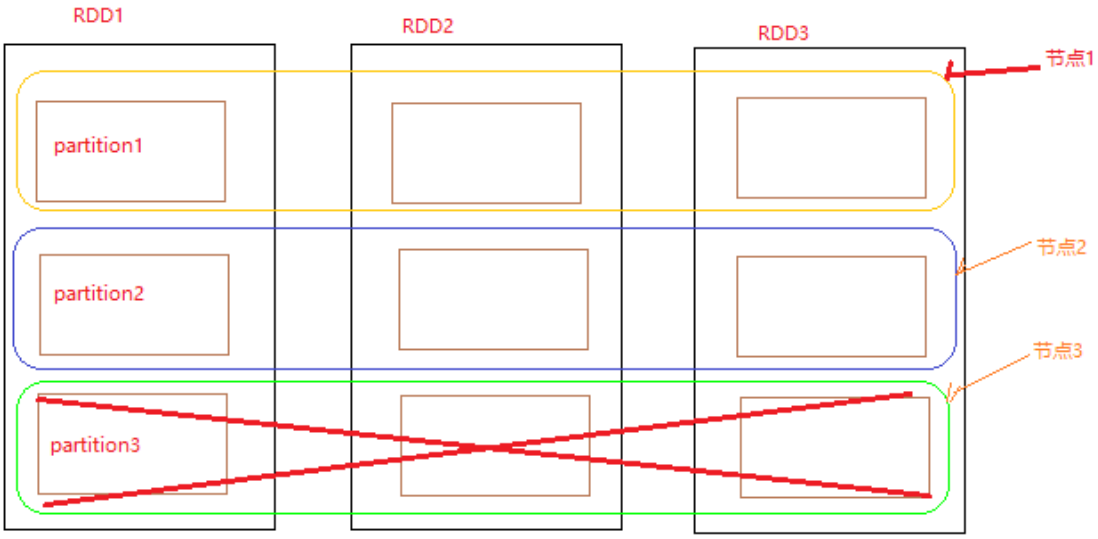
宽/窄依赖的概念不止用在 stage 划分中，对容错也很有用。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

Dependency 代表了 RDD 之间的依赖关系，即血缘（Lineage）



数据进行切片之后，每一个分区对应的数据，是固定的。（起始偏移量和结束的偏移量）

分区是在 driver 端生成的，即使某一个分区挂掉了，也没有关系，driver 端会根据 rdd 的依赖关系，重新起新的 task 进行任务执行。



如果 transformation 操作中间发生计算失败：

1、运算是窄依赖：只要把丢失的父 RDD 分区重新计算即可，跟其他节点没有依赖，这样可以大大减少恢复丢失的数据分区的开销

2、运算是宽依赖：需要父 RDD 的所有分区都存在，重算代价较高

3、如果整个节点挂掉，dirver 会把任务在其他的节点中的 executor 中重新启动

4、缓存或者增加检查点：当 Lineage 特别长或者有宽依赖时，主动调用 checkpoint 把当前数据写入稳定存储，作为检查点。但 Checkpoint 会产生磁盘开销，因为其就是将数据持久化到磁盘中，所以做检查点的 RDD 最好是已经在内存中缓存了。

为了提升运算的效率，更好的解决容错问题，spark 提供了一系列的解决方案，缓存，检查点（checkpoint）

缓存：把 rdd 的数据，写入到内存或者磁盘

checkpoint： 写入到 hdfs 中

使用缓存和持久化来提升运行效率。

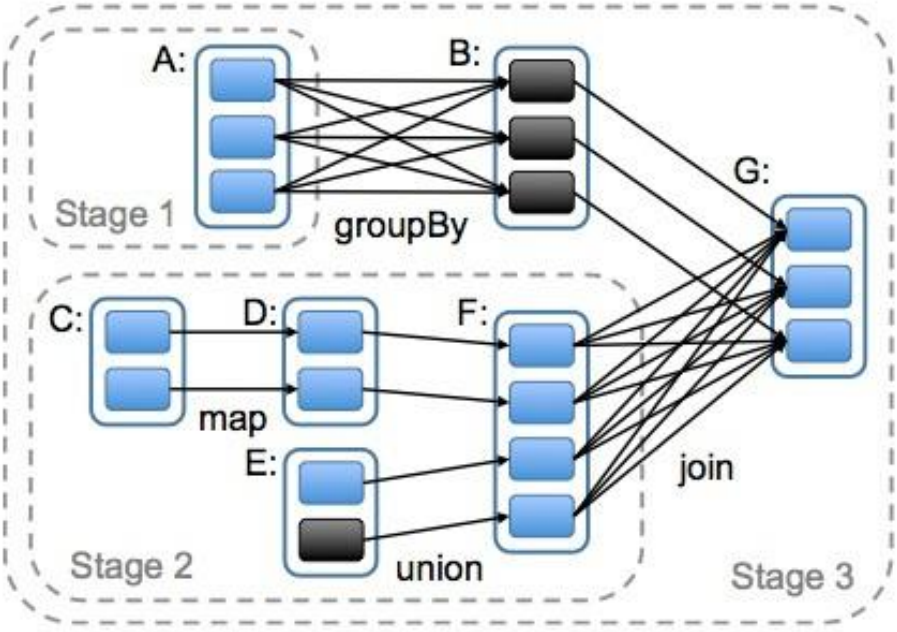
1.8. DAG 的生成

DAG(Directed Acyclic Graph)叫做**有向无环图**，指任意一条边有方向，且不存在环路的图。

点：RDD 边： RDD 之间的依赖关系

在 spark 里每一个操作生成一个 RDD，RDD 之间连一条边，最后这些 RDD 和他们之间的边组成一个有向无环图，这个就是 spark 中的 DAG。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。



宽依赖是划分 stage 的标识。

总结：

DAG 有向无环图，代表 RDD 的转换过程，其实就是代表着数据的流向。

DAG 是有边界的，有开始，有结束。通过 sparkContext 创建 RDD 就是开始，触发 action 就会生成一个完整的 DAG。DAG 会被切分成多个 Stage（阶段），切分的依据就是宽依赖（shuffle），先会提交前面的 stage，然后提交后面的 stage，一个 Stage 中有多个 Task，多个 Task 可以并行执行。