> 在一个孤立系统里，如果没有外力做功，其总混乱度（即熵）会不断增大

RDD函数传递和依赖关系

RDD中的函数传递

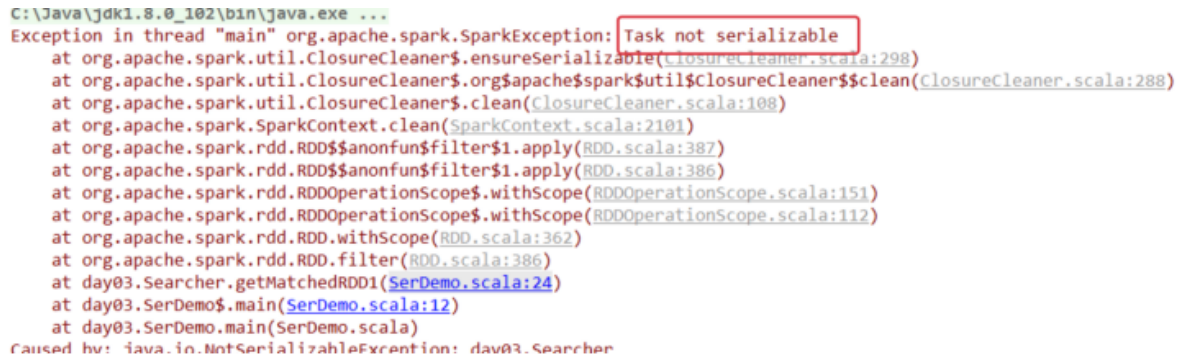
> 在实际开发中我们往往需要自己定义一些对于RDD的操作，那么此时需要主要的是，初始化工作是在Driver端进行的，而实际运行程序是在Executor端进行的，这就涉及到了跨进程通信，是需要序列化的。

跨进程通信，对象传递，序列化

传递一个方法

```
import org.apache.spark.{SparkConf， SparkContext}
import org.apache.spark.rdd.RDD

object SerDemo {
    def main(args: Array[String]): Unit = {
        valconf:SparkConf=new SparkConf()
.setAppName("SerDemo")
.setMaster("local[*]")
        val sc = new SparkContext(conf)
        val rdd: RDD[String] = sc.parallelize(Array("hello world"， "hello
zhutian"， "zhutian"， "hahah")， 2)
        val searcher = new Searcher("hello")
        val result: RDD[String] = searcher.getMatchedRDD1(rdd)
        result.collect.foreach(println)
    }
}
//需求: 在 RDD 中查找出来包含 query 子字符串的元素
// query 为需要查找的子字符串
class Searcher(val query: String){
    // 判断 s 中是否包括子字符串 query
    def isMatch(s : String) ={
        s.contains(query)
    }
    // 过滤出包含 query字符串的字符串组成的新的 RDD
    def getMatchedRDD1(rdd: RDD[String]) ={
        rdd.filter(isMatch)  //
    }
    // 过滤出包含 query字符串的字符串组成的新的 RDD
    def getMatchedRDD2(rdd: RDD[String]) ={
        rdd.filter(_.contains(query))
    }
}
```

直接运行程序会发现报错:没有初始化.因为rdd.filter(isMatch)用到了对象this的方法isMatch，所以对象this需要序列化，才能把对象从driver发送到executor。【其实就是对象在进程间传输需要序列化】



> 解决方案:让Searcher类实现序列化接口:Serializable【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

传递一个属性

```
object SerDemo {
    def main(args: Array[String]): Unit = {
        val conf:SparkConf = new SparkConf()
.setAppName("SerDemo")
.setMaster("local[*]")
        val sc = new SparkContext(conf)
        val rdd: RDD[String] = sc.parallelize(Array("hello world", "hello
zhutian", "zhutian", "hahah"), 2)
        val searcher = new Searcher("hello")
        // 这里调用上面定义的第二个方法，这个方法里面传递的是query这个属性，如果对象没序列化
还是报错
        val result: RDD[String] = searcher.getMatchedRDD2(rdd)
        result.collect.foreach(println)
    }
}
```

> 解决方案有2种:
>
> (1) 让类实现序列化接口:Serializable。【推荐，反正类都实现序列化接口准没错】
>
> (2) 传递局部变量而不是属性。

kryo序列化框架【推荐】

参考地址:https://github.com/EsotericSoftware/kryo
Java的序列化比较重，能够序列化任何的类。比较灵活，但是相当的慢，并且序列化后对象的提交也比较大。

Spark处于性能的考虑，支持另外一种序列化机制:kryo(2.0开始支持)。kryo比较快和简洁(速度是Serializable的10倍)。想获取更好的性能应该使用kryo来序列化。从2.0开始，Spark内部已经在使用kryo序列化机制：当RDD在Shuffle数据的时候，简单数据类型，简单数据类型的数组和字符串类型已经在使用kryo来序列化。

有一点需要注意的是:即使使用kryo序列化，也要继承Serializable接口

使用kryo框架

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD

object SerDemo {
```

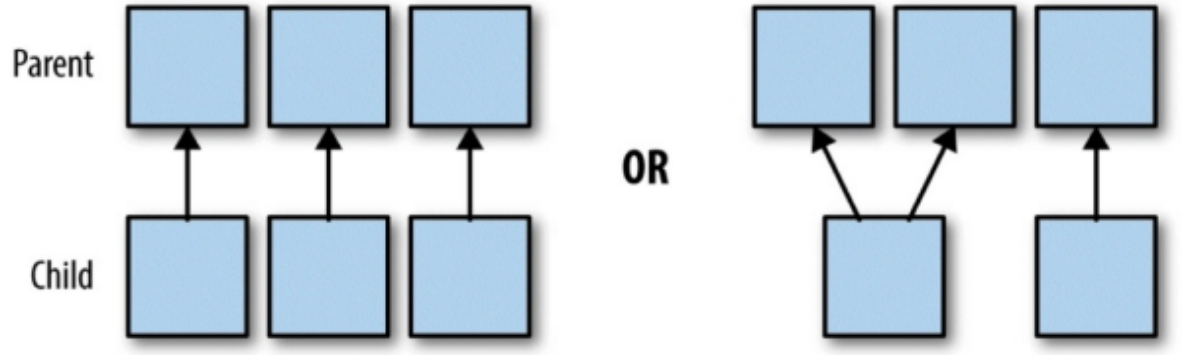
```
    def main(args: Array[String]): Unit = {
        val conf: SparkConf = new SparkConf()
            .setAppName("SerDemo")
            .setMaster("local[*]")
            // 替换默认的序列化机制
            .set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
            // 注册需要使用 kryo 序列化的自定义类
            .registerKryoClasses(Array(classOf[Searcher]))
        val sc = new SparkContext(conf)
        val rdd: RDD[String] = sc.parallelize(Array("hello world", "hello
zhutiansama", "zhutiansama", "hahah"), 2)
        val searcher = new Searcher("hello")
        val result: RDD[String] = searcher.getMatchedRDD1(rdd)
        result.collect.foreach(println)
    }
}
case class Searcher(val query: String) {
    // 判断 s 中是否包括子字符串 query
    def isMatch(s: String) = {
        s.contains(query)
    }

    // 过滤出包含 query字符串的字符串组成的新的 RDD
    def getMatchedRDD1(rdd: RDD[String]) = {
        rdd.filter(isMatch)
    }

    // 过滤出包含 query字符串的字符串组成的新的 RDD
    def getMatchedRDD2(rdd: RDD[String]) = {
        val q = query
        rdd.filter(_.contains(q))
    }
}
```

RDD依赖关系

> 如果要真正的用好RDD的算子，那么就必须理清RDD的依赖关系。RDD之间的关系可以从两个维度来理解:一个是RDD是从哪些RDD转换而来，另一个就是RDD依赖于父RDD的哪些分区。这种关系就是RDD之间的依赖。依赖有2种策略:窄依赖和宽依赖。

窄依赖

如果依赖关系在设计的时候就可以确定，而不需要考虑父RDD分区中的记录，并且如果父RDD中的每个分区最多只有一个子分区，这样的依赖就叫窄依赖。核心就是父RDD的每个分区最多被一个RDD的分区使用。



具体来说，窄依赖的时候，子RDD中的分区要么只依赖一个父RDD中的一个分区(比如map，filter操作)，要么在设计时候就能确定子RDD是父RDD的一个子集(比如:coalesce)。所以，窄依赖的转换可以在任何的的一个分区上单独执行，而不需要其他分区的任何信息。

宽依赖

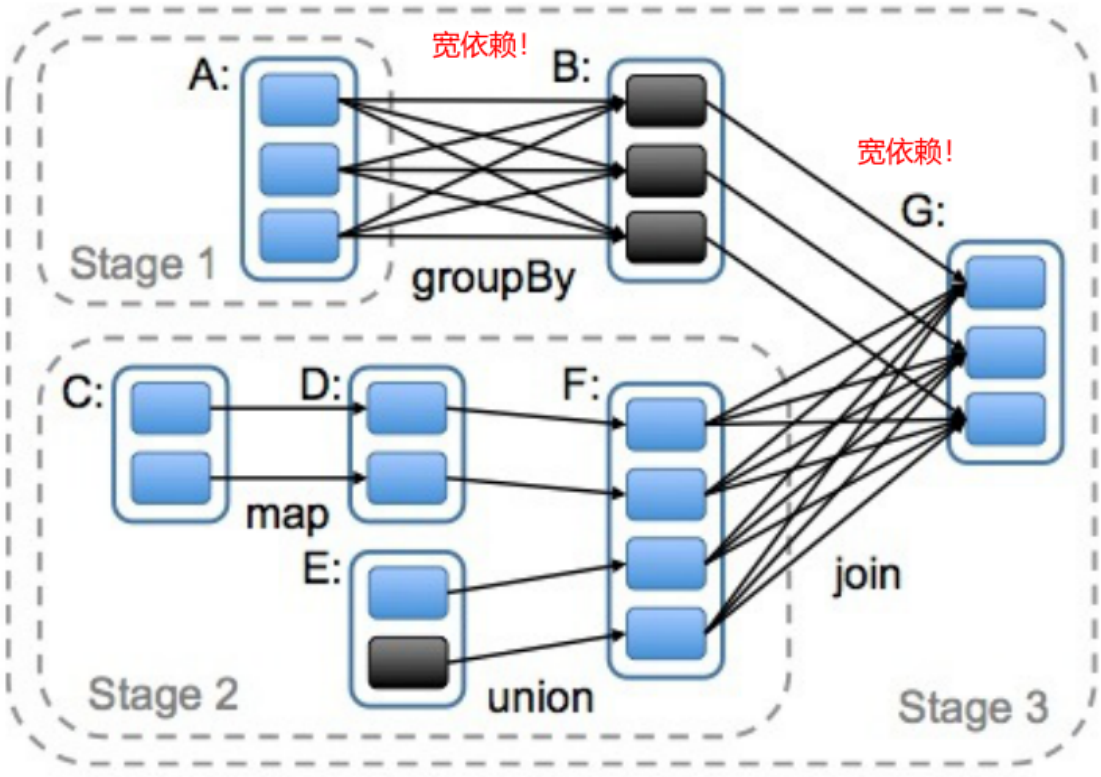
如果父RDD的分区被不止一个子RDD的分区依赖，就是宽依赖。

宽依赖工作的时候，不能随意在某些记录上运行，而是需要使用特殊的方式(比如按照key)来获取分区中的所有数据。例如:在排序的时候，数据必须被分区，同样范围的key必须在同一个分区内.具有宽依赖的transformations包括:sort，reduceByKey，groupByKey，join和调用rePartition函数的任何操作。

DAG有向无环图

> 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。

根据宽依赖划分Stage！！！



任务划分

RDD任务切分中间分为：Application、Job、Stage和Task

1）Application：初始化一个SparkContext即生成一个Application；

2）Job：一个Action算子就会生成一个Job；

3）Stage：根据RDD之间的依赖关系的不同将Job划分成不同的Stage，遇到一个宽依赖则划分一个Stage；