# 1.rdd:

RDD是Spark中的抽象数据结构类型，任何数据在Spark中都被表示为RDD。

从编程的角度来看，RDD可以简单看成是一个数组。

和普通数组的区别是，RDD中的数据是分区存储的，这样不同分区的数据就可以分布在不同的机器上，

同时可以被并行处理。因此，Spark应用程序所做的无非是把需要处理的数据转换为RDD，然后对RDD进行一系列的变换和操作从而得到结果。

# 2.创建rdd

## 2.1举例：从普通数组创建RDD，里面包含了1到9这9个数字，它们分别在3个分区中。

val a = sc.parallelize(1 to 9, 3)

['pærəlelaɪz]

## 2.2举例：读取文件README.md来创建RDD，文件中的每一行就是RDD中的一个元素

val b = sc.textFile("README.md")

# 3.map

map是对RDD中的每个元素都执行一个指定的函数来产生一个新的RDD。 任何原RDD中的元素在新RDD中都有且只有一个元素与之对应。

scala> val a = sc.parallelize(1 to 9, 3)

scala> val b = a.map(x => x\*2)

scala> a.collect

res10: **Array**[Int] = **Array**(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> b.collect

res11: **Array**[Int] = **Array**(2, 4, 6, 8, 10, 12, 14, 16, 18)

把原RDD中每个元素都乘以2来产生一个新的RDD。

# 4.mapPartitions

mapPartitions是map的一个变种。map的输入函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区，也就是把每个分区中的内容作为整体来处理的。它的函数定义为：

def **mapPartitions**[**U**: ClassTag](f: Iterator[T] => Iterator[U], preservesPartitioning: Boolean = false): RDD[U]

f即为输入函数，它处理每个分区里面的内容。每个分区中的内容将以Iterator[T]传递给输入函数f，f的输出结果是Iterator[U]。最终的RDD由所有分区经过输入函数处理后的结果合并起来的。

scala> val a = sc.parallelize(1 to 9, 3)

scala> **def** **myfunc**[**T**](iter: Iterator[T]) : Iterator[(T, T)] = {

var res = List[(T, T)]()

var pre = iter.next **while** (iter.hasNext) {

val cur = iter.next;

res .::= (pre, cur) pre = cur;

}

res.iterator

}

scala> a.mapPartitions(myfunc).collect

res0: Array[(Int, Int)] = Array((2,3), (1,2), (5,6), (4,5), (8,9), (7,8))

函数myfunc是把分区中一个元素和它的下一个元素组成一个Tuple。因为分区中最后一个元素没有下一个元素了，所以(3,4)和(6,7)不在结果中。 mapPartitions还有些变种，比如mapPartitionsWithContext，它能把处理过程中的一些状态信息传递给用户指定的输入函数。还有mapPartitionsWithIndex，它能把分区的index传递给用户指定的输入函数。

# 5.mapValues

mapValues顾名思义就是输入函数应用于RDD中Kev-Value的Value，原RDD中的Key保持不变，与新的Value一起组成新的RDD中的元素。因此，该函数只适用于元素为KV对的RDD。

scala> val a = sc.parallelize(**List**("dog", "tiger", "lion", "cat", "panther", " eagle"), 2)

scala> val b = a.map(x => (x.length, x))

scala> b.mapValues("x" + \_ + "x").collect

res5: **Array**[(Int, String)] = **Array**((3,xdogx), (5,xtigerx), (4,xlionx),(3,xcatx), (7,xpantherx), (5,xeaglex))

# 6.mapWith

mapWith是map的另外一个变种，map只需要一个输入函数，而mapWith有两个输入函数。它的定义如下：

**def** **mapWith**[**A**: ClassTag, U: ](constructA: Int => A, preservesPartitioning: Boolean = false)(f: (T, A) => U): RDD[U]

第一个函数constructA是把RDD的partition index（index从0开始）作为输入，输出为新类型A；

第二个函数f是把二元组(T, A)作为输入（其中T为原RDD中的元素，A为第一个函数的输出），输出类型为U。

val x = sc.parallelize(**List**(1,2,3,4,5,6,7,8,9,10), 3)

x.mapWith(a => a \* 10)((a, b) => (b + 2)).collect

res4: **Array**[Int] = **Array**(2, 2, 2, 12, 12, 12, 22, 22, 22, 22)

把partition index 乘以10，然后加上2作为新的RDD的元素。

# 7.flatMap

与map类似，区别是原RDD中的元素经map处理后只能生成一个元素，而原RDD中的元素经flatmap处理后可生成多个元素来构建新RDD。 举例：对原RDD中的每个元素x产生y个元素（从1到y，y为元素x的值）

scala> val a = sc.parallelize(1 to 4, 2)

scala> val b = a.flatMap(x => 1 to x)

scala> b.collect

res12: **Array**[Int] = **Array**(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)

# 8.flatMapWith

flatMapWith与mapWith很类似，都是接收两个函数，一个函数把partitionIndex作为输入，输出是一个新类型A；另外一个函数是以二元组（T,A）作为输入，输出为一个序列，这些序列里面的元素组成了新的RDD。它的定义如下：

**def** **flatMapWith**[**A**: ClassTag, U: ClassTag](constructA: Int => A, preservesPartitioning: Boolean = false)(f: (T, A) => Seq[U]): RDD[U]

scala> val a = sc.parallelize(**List**(1,2,3,4,5,6,7,8,9), 3)

scala> a.flatMapWith(x => x, **true**)((x, y) => **List**(y, x)).collect

res58: **Array**[Int] = **Array**(0, 1, 0, 2, 0, 3, 1, 4, 1, 5, 1, 6, 2, 7, 2, 8, 2, 9)

# 9.flatMapValues

flatMapValues类似于mapValues，不同的在于flatMapValues应用于元素为KV对的RDD中Value。每个一元素的Value被输入函数映射为一系列的值，然后这些值再与原RDD中的Key组成一系列新的KV对。

scala> val a = sc.parallelize(**List**((1,2),(3,4),(3,6)))

scala> val b = a.flatMapValues(x=>x.to(5))

scala> b.collect res3: **Array**[(Int, Int)] = **Array**((1,2), (1,3), (1,4), (1,5), (3,4), (3,5))

上述例子中原RDD中每个元素的值被转换为一个序列（从其当前值到5），比如第一个KV对(1,2), 其值2被转换为2，3，4，5。然后其再与原KV对中Key组成一系列新的KV对(1,2),(1,3),(1,4),(1,5)。

# 10.reduce

reduce将RDD中元素两两传递给输入函数，同时产生一个新的值，新产生的值与RDD中下一个元素再被传递给输入函数直到最后只有一个值为止。

scala> val c = sc.parallelize(1 to 10)

scala> c.reduce((x, y) => x + y)

res4: Int = 55

# 11.reduceByKey

顾名思义，reduceByKey就是对元素为KV对的RDD中Key相同的元素的Value进行reduce，因此，Key相同的多个元素的值被reduce为一个值，然后与原RDD中的Key组成一个新的KV对。

scala> val a = sc.parallelize(**List**((1,2),(3,4),(3,6)))

scala> a.reduceByKey((x,y) => x + y).collect

res7: **Array**[(Int, Int)] = **Array**((1,2), (3,10))

对Key相同的元素的值求和，因此Key为3的两个元素被转为了(3,10)。