# Spark学习笔记

## spark core

### Spark和MapReduce的对比：

spark速度更快，spark和MapReduce在进行逻辑回归机器学习的性能比较中，spark比MapReduce至少快了100倍；spark除了比MapReduce更快之外，还有更加简单易用的编程模型，RDD。

#### spark为什么会那么快？

在说明spark为什么那么快之前，我们得说说MapReduce为什么那么慢，其实刚开始的时候， 人们并没有觉得MapReduce很慢，毕竟人家针对的场景是大数据，海量数据，分布式计算么，就应该是这个速度；

但是自从马铁开源了spark之后，开发者们才知道原来分布式计算可以这么快，从此就开始嫌弃MapReduce的速度了。

为什么MapReduce较之于spark会很慢：首先MapReduce的应用一次只是运行一个map和reduce，而spark会根据应用的复杂程度将任务分割成多个计算阶段，这些计算阶段会组成一个有向无环图DAG，spark的任务调度器可以根据DAG的依赖关系执行计算阶段。

在进行逻辑回归的机器学习的算法的性能比较的过程中，spark比MapReduce快了100多倍。因为在某些机器学习的算法中有大量的迭代计算，产长数万个计算阶段，这些计算阶段可以在以个应用中完成。而MapReduce则需要启动数万个应用，因此极大的提高了效率。

在DAG中，spark会根据数RDD（RDD数据也就是我们要处理的数据，RDD存在有分区的）的血缘关系，判断宽窄依赖，从而生成stage（也就是有shuffle的阶段，就会产生stage），如此看来，spark可以进行连续的shuffle，而不是像MapReduce那样，一个应用只能进行一个shuffle。如此一来也就更快了。

虽然spark的DAG的本质也是MapReduce，但是这种多个计算阶段依赖执行的方案能够有效的减少对于HDFS的访问，减少作业的调度执行次数，因此速度也就更快。

MapReduce主要使用磁盘存储shuffle过程数据，而spark优先使用内存进行数据存储，，除非是内存不够用了，否则是尽可能的使用内存，所以spark更快。

回答的角度：

①：DAG角度，让多个MapReduce连接了起来，减少了对于HDFS的访问次数，减少了任务调度执行的次数，然后在举上面的机器学习的例子。

②：spark优先使用内存进行shuffle过程的数据的存储。但是后期的Spark再进行了shuffle之后，所有的数据都要写文件，由ShuffleMapTask去写，有ResultTask去读（如果shuffleMapStage是最后一个的话）。

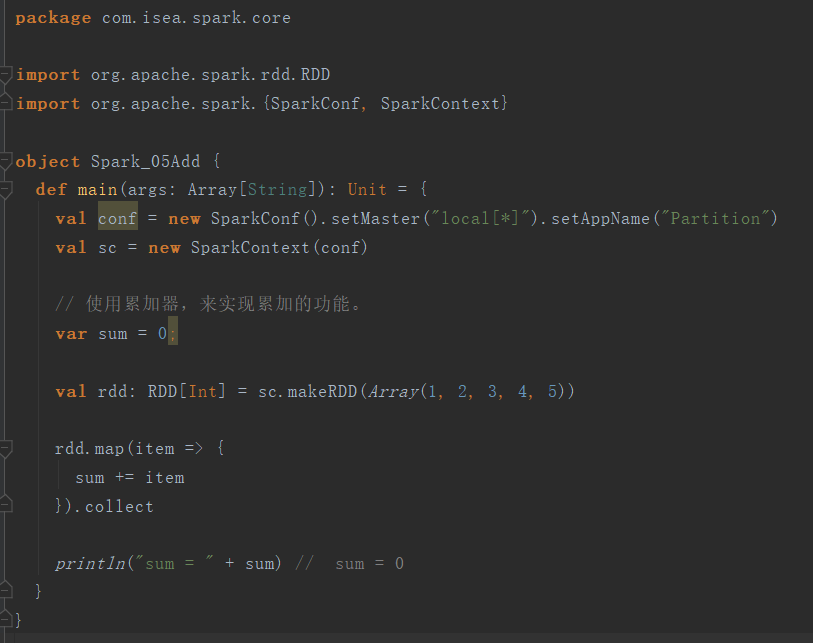
#### 累加器

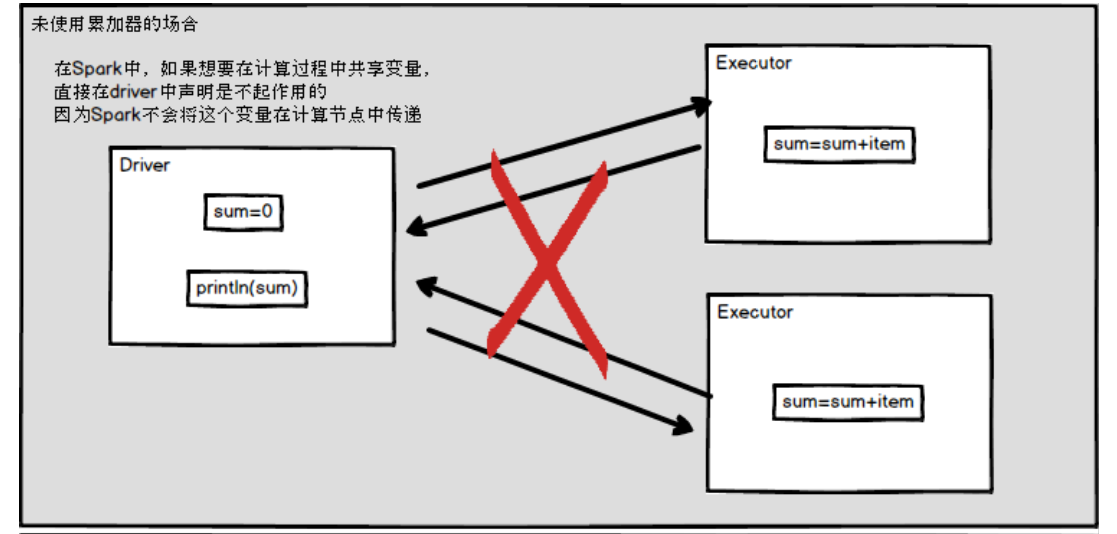
累加器是spark的三大核心数据结构之一，另外两个是RDD和广播变量

累加器是分布式的只写变量，在executor端是只写变量，返回driver端进行数据的merge，Driver端可以读取merge之后的值。

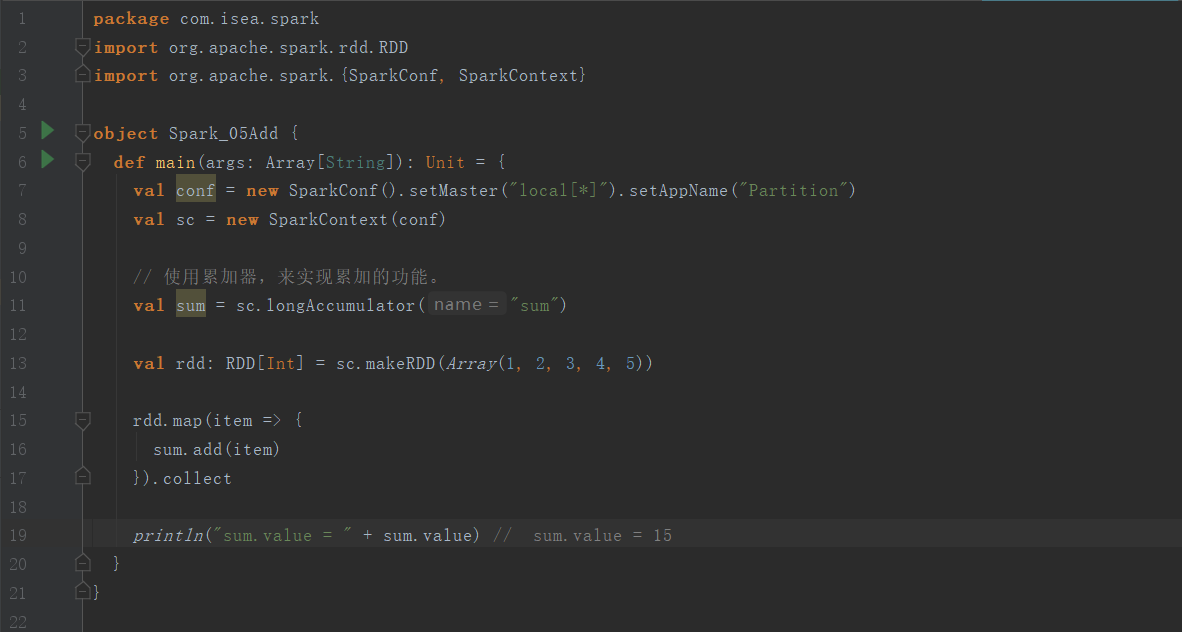
广播变量是分布式的只读变量。

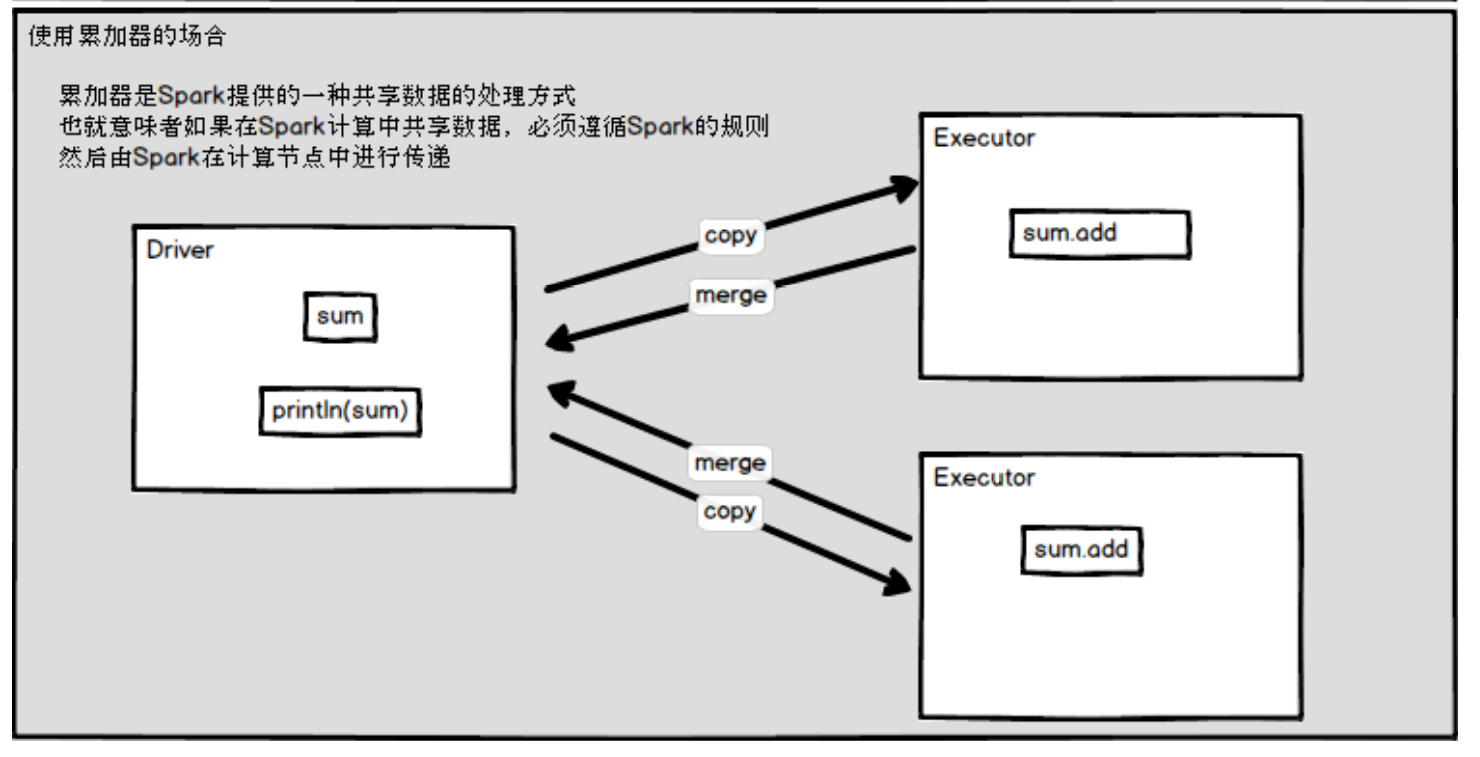
先看如下的程序：





如果使用累加器，就能够实现变量的共享；广播变量也是属于变量的共享的范畴。





自定义累加器的时候，需要继承Accumulatorv2类，并实现对应的方法。

#### 广播变量

广播变量用来高效的分发大的对象，如果在做RDD操作的时候，如果此时有shuffle的话，性能会比较慢，可以使用广播变量，将数据传入到Executor，然后在执行RDD的时候，不在需要做shuffle，Executor在计算的时候，直接从内存中读取数据，自己来做逻辑匹配，而且由于没有shuffle，还不需要落盘。

要知道在每一个executor中是有多个Task的，而每一个Task代表着一个任务，也就需要进行计算和数据的读取。广播变量的时候是每一个Executor保存一份数据。而不是每一个NodeManager，也不是每一个Task，但是累加器的时候，是每一个Task保存一份累加器数据。这主要是由累加器和广播变量的数据的大小决定的。

|  |
| --- |
| 补充：    执行的时间非常的长：如下图，为什么会这样呢？因为在做join操作的时候需要shuffle，底层就是rdd1的每一个key和rdd2中的每一个元素做key的比对，所以这样的效率是非常的低效的      针对这种情况，Spark提供给了一种解决方案：原理是，不在shuffle，而是将其中一个rdd中的数据，放在内存中，另外一个rdd的内容，直接做比对，放置在内存中的数据不能太大。 |

### Spark Shuffle

### Spark结构

#### Spark的架构和作业提交流程：

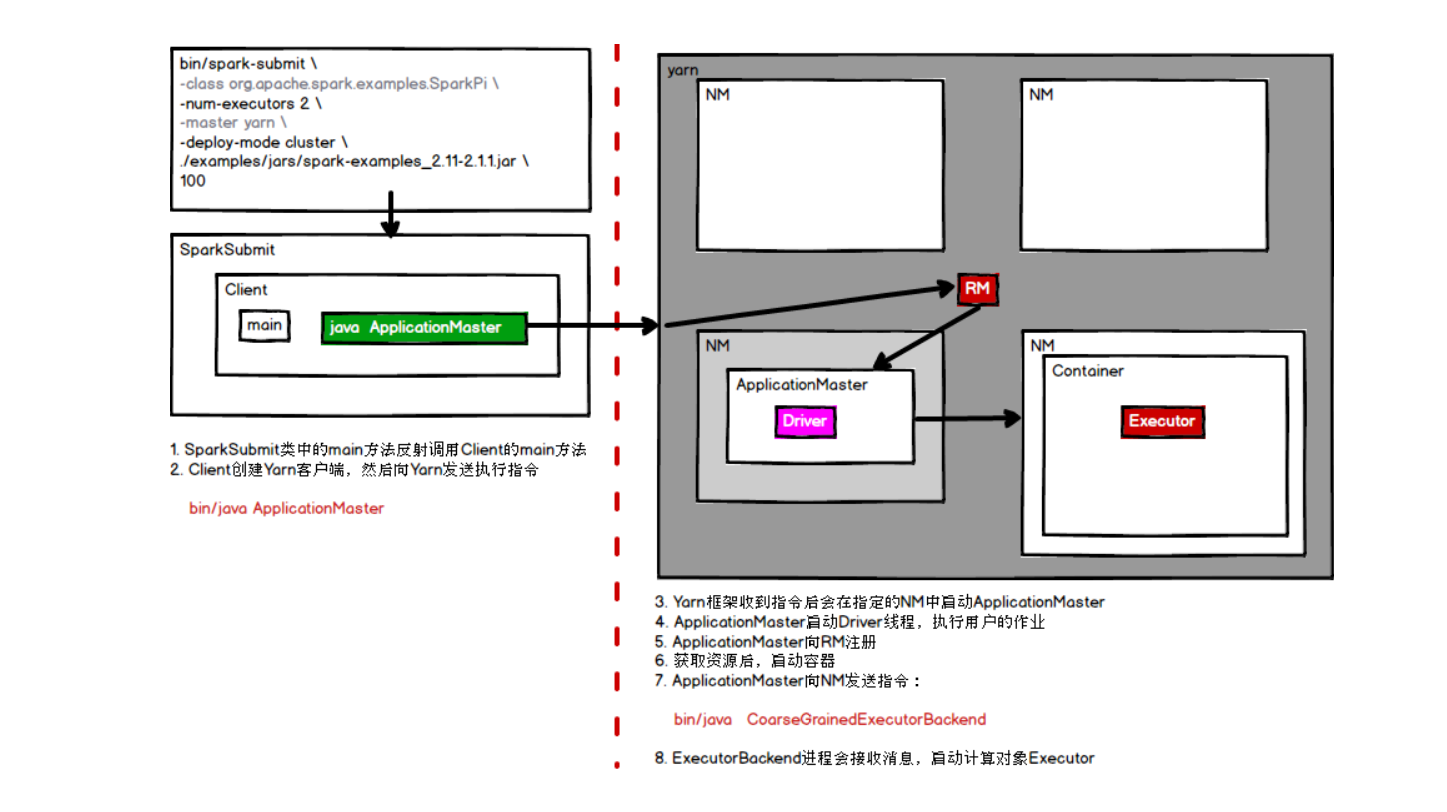
如果我们需要通过yarn来执行我们的spark程序，我们要提交：

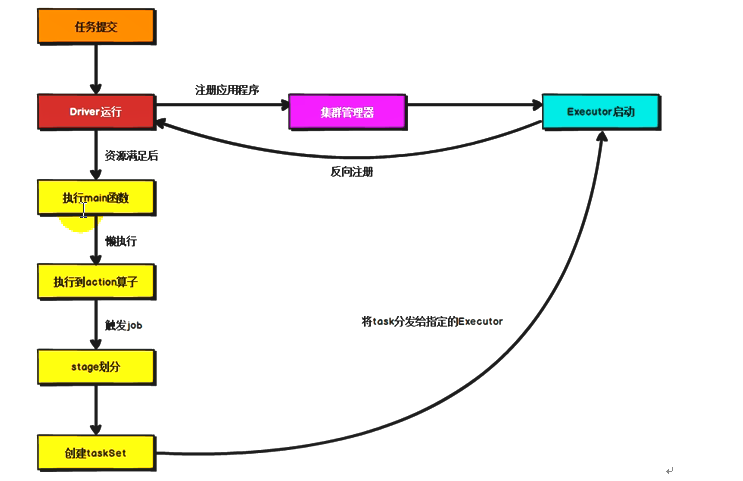
|  |
| --- |
| bin/spark-submit \ (下面是参数部分)  --class org.apache.spark.examples.SparkPi \  --num-executors 2 \  --master yarn \  --deploy-mode cluster \  ./examples/jars/spark-examples\_2.11-2.1.1.jar \  100 |

调用上述的bin/spark-submit脚本，等同于我们执行了一个下面这个java类，（$@获取了上面脚本中的全部参数）

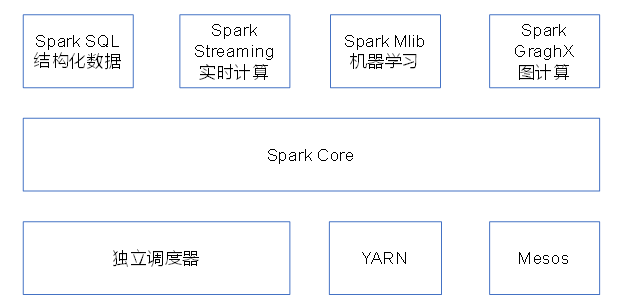


好了，现在我们知道了，实际上走的是这个SparkSubmit这个类





#### Spark的内置模块



Spark自带资源的调度器（集群管理器），但是spark的核心是计算，整个计算是由spark core和上面的应用组成的，资源的调度可以完全交由YARN（也是目前国内使用最多的）

#### Driver

Driver的作用如下：

1）把用户程序转为Job

2）跟踪Executor的运行状况

3）为执行器节点调度任务

4）UI展示应用运行状况

#### Executor

1）负责运行组成 Spark 应用的任务，**并将结果返回给Driver；**

2）通过自身的块管理器（Block Manager）为用户程序中要求缓存的RDD提供内存式存储。RDD是直接缓存在Executor进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

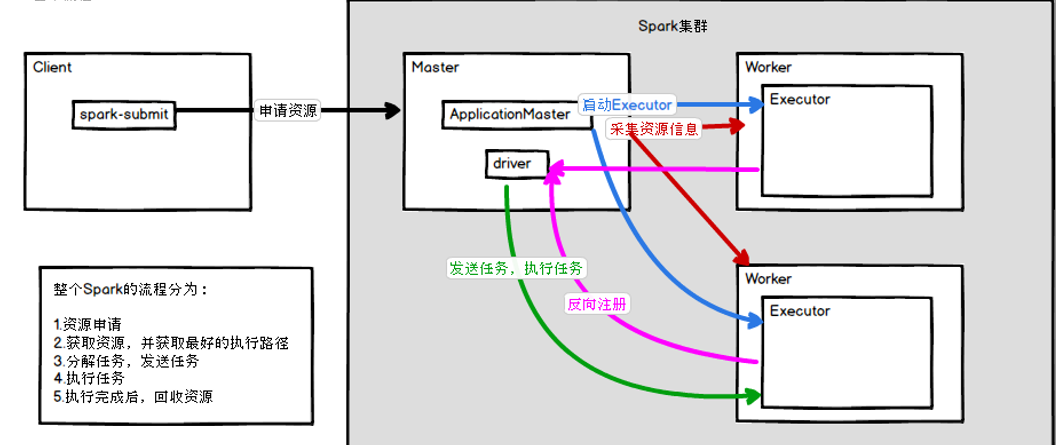
#### Spark的运行模式部署模式

1. Local模式，在一台计算机上运行spark，使用local [k]来设置线程数量，local[\*]表示最多

**打开一个spark-shell就能够运行spark程序，通常用来测试使用，开始了spark-shell之后，就会多一个叫做SparkSubmit的进程。**

1. Standalone模式，构建一个master+Slave的Spark集群（集群中的每一台机器都需要安装spark），Spark运行在集群中，使用spark自带的集群管理器，进行相关的资源调度。**此时需要启动sbin/start-all.sh，来启动spark自带的集群管理区，这里会产生master和worker进程，启动spark集群的环境。**

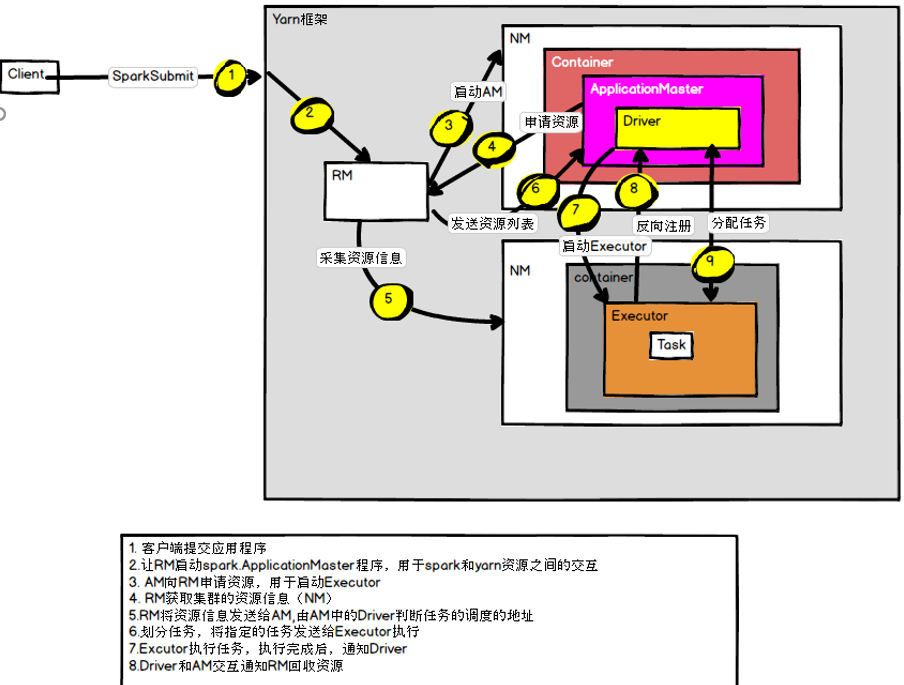
|  |
| --- |
| bin/spark-submit \  --master <master-url> spark : //linux1:7077 \    Spark-submit会提交一个作业，该作业执行完毕，关闭Spark-submit之后，粗粒度Executor的后台进程会关闭，但是worker和master不会关闭。 |



1. YARN模式，（只需要一台机器安装spark，用来提交作业）spark客户端直接连接YARN，不需要构建spark集群，有yarn-client（测试）和yarn-cluster（生产）两种模式（主要的区别是Driver程序的运行节点）

使用YARN作为集群管理器，进行相关的资源调度。**需要配置好yarn的环境，和hdfs的环境，并需要驱动，有yarn来提供资源调度，spark，只是单纯的作为计算的框架。不需要sbin/start-all.sh，来启动spark的资源调度框架**

|  |
| --- |
| bin/spark-submit \  --master yarn \  这里只是在ApplicationMaster位置有Spark的程序，使用YARN的资源调度框架。Spark的可插拔性就变的很高，只是做他擅长的计算。  **在启动之前，要启动HDFS和YARN的集群。**  **为什么要启动HDFS？因为可能程序需要连接Hadoop** |



1. Mesos模式，

|  |
| --- |
| bin/spark-submit \  --class <main-class>  --master <master-url> \  --deploy-mode <deploy-mode> \ //发布class到client还是cluster执行  --conf <key>=<value> \  ... # other options  <application-jar> \  [application-arguments]  bin/spark-submit \  --class org.apache.spark.examples.SparkPi \  --executor-memory 1G \  --total-executor-cores 2 \  ./examples/jars/spark-examples\_2.11-2.1.1.jar \  100 |

词频统计的代码

|  |
| --- |
|  |

#### 并发并行区别：

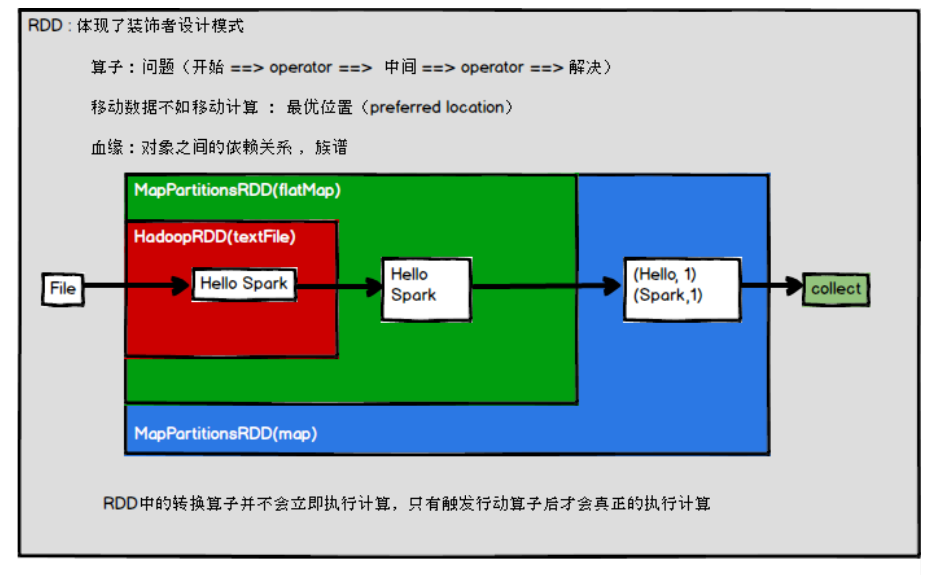
并行：是同时执行，三个core，三个任务同时执行

并发：是同时抢占，一个core，三个任务同时抢占

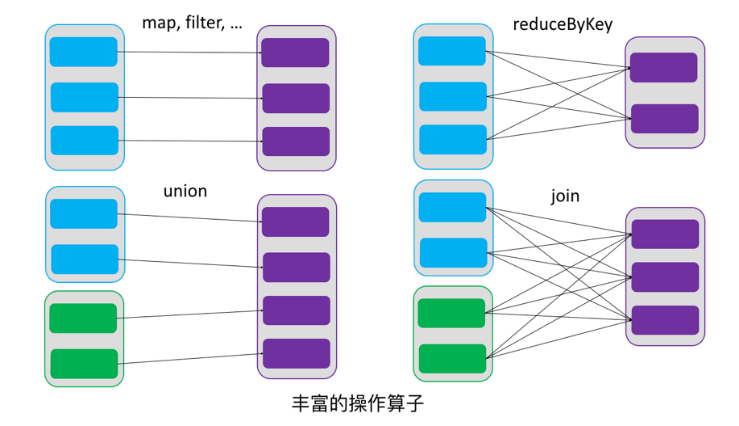
### RDD

RDD是分布式弹性数据集、是spark中最基本的数据抽象

1. RDD是只读的，如果想要改变RDD的值，会创建新的RDD，
2. **RDD在逻辑上是分区的**，具体在哪个分区，通过compute函数得到，如sc.makeRDD(Array(1,2,3,4))使用compute进行分区。
3. **血缘关系**



下面的这张图：一个大块就是一个RDD，里面的小块是RDD的某一个分区，可以发现一个分区中的数据去往了另外一个RDD的不同分区，就是属于款依赖。



#### 常见的转化算子和常见的执行算子，并解释

转化算子：

map，遍历，一个输入对应一个输出，可以返回不同的类型

flatMap，扁平化，可以说词频统计的例子，可以映射出多个元素构成序列，所以传入的func应该返回一个序列。

sc.texFile().flatMap(\_.split(\_,1)).reduceByKey(\_+\_).collect;

mapPartition：一个分区作为一个输入，所以该分区的数据没有操作完的时候，不会释放，可以会OOM。

mapPartitionWithIndex：分区号和该分区内的数据构成一个元素作为输入。

glom：将RDD的每一个分区形成一个数组，

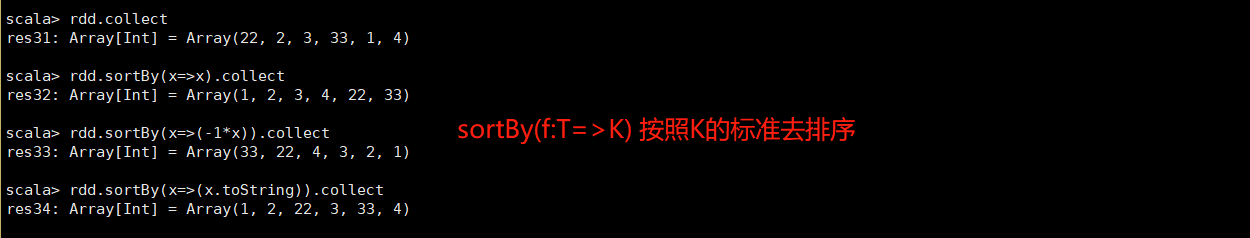
sample：主要用来取样，

filter：传入的func判断为true的，会被保留，返回回来新的RDD

distinct：去重，发生了shuffle

reparation：重新分区，需要进行shuffle，调用了coalease，colaease不需要进行shuffle。

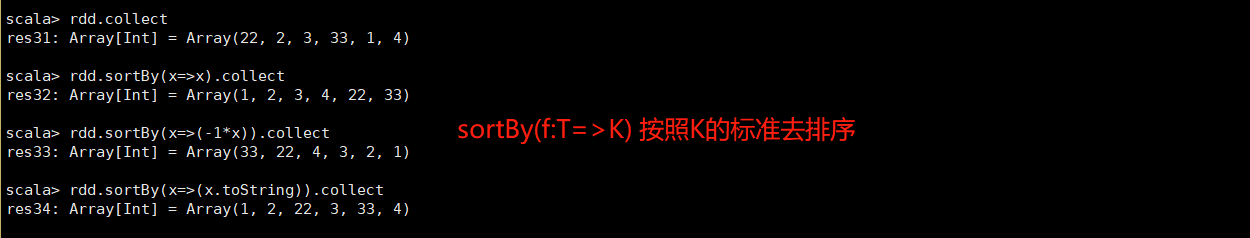
sortBykey：按照func的定义的规则排序。



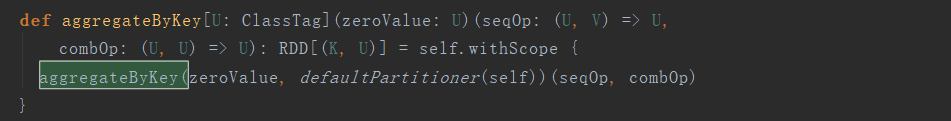
groupbykey：分组之后，直接shuffle，按照key进行聚合

reducebykey：在shuffle之前有一个预分区，按照key进行聚合

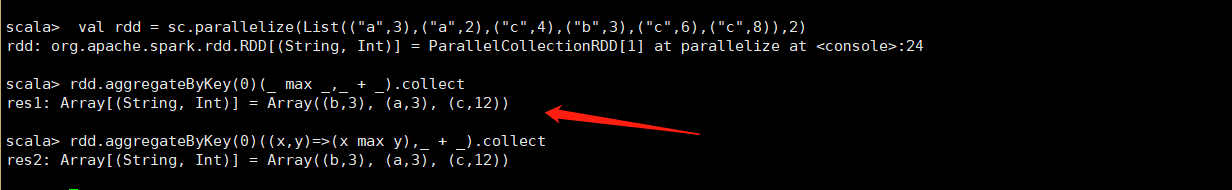
partitionByKey：



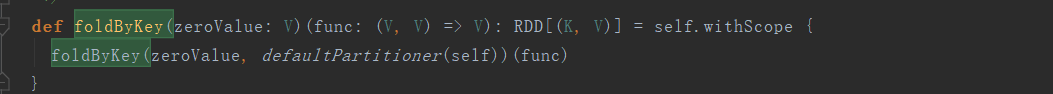
aggregateByKey：



取出每一个分区相同的key对应的最大值，然后相加。



foldByKey：分区内和分区间规则一样的，

combinerByKey：第一参数里面可以写逻辑：

|  |
| --- |
| 补充：    这里[C]表名该反省是不能被推断的，  createCombiner:V=>C 将当前的V类型，做了一个转化，创建聚合器；分区内第一会遇到一个key的时候，走这个方法，但是会将value传递过来  mergeValue，分区内第二次遇到key，就会执行第二个方法，这里需要指定类型  mergeCombiners：分区之间的数据的聚合，这里需要指定类型。 |

执行算子：  
collect

reduce

foreach

take

first

aggregate：combine函数将每个分区的结果和初始值(zeroValue)进行combine操作

|  |
| --- |
| 补充：aggregateByKey和aggregate的区别：  后者，第一个参数在分区之间计算的时候也参与运算。 |

#### RDD的创建

|  |
| --- |
| **def** textFile(  path: String,  minPartitions: Int = defaultMinPartitions): RDD[String] = withScope {  assertNotStopped()  hadoopFile(path, *classOf*[TextInputFormat], *classOf*[LongWritable], *classOf*[Text],  minPartitions).map(pair => pair.\_2.toString).setName(path) }  ClassTag 表示可推断类型  **def** parallelize[T: ClassTag](  seq: Seq[T],  numSlices: Int = defaultParallelism): RDD[T] = withScope {  assertNotStopped()  **new** ParallelCollectionRDD[T](**this**, seq, numSlices, Map[Int, Seq[String]]()) }  defaultParallelism 是默认并行度，这是一个方法。  Parallelize 方法调用了 withScope方法 |

创建RDD的方法中，第二个参数都和并行度相关，也和分区数有关。具体的关系如下：

|  |
| --- |
| 其中hadoop的分区规则是参见Hadoop源码。  在没有指定分区的数量的时候，local的分区如下： |

#### RDD的对比图

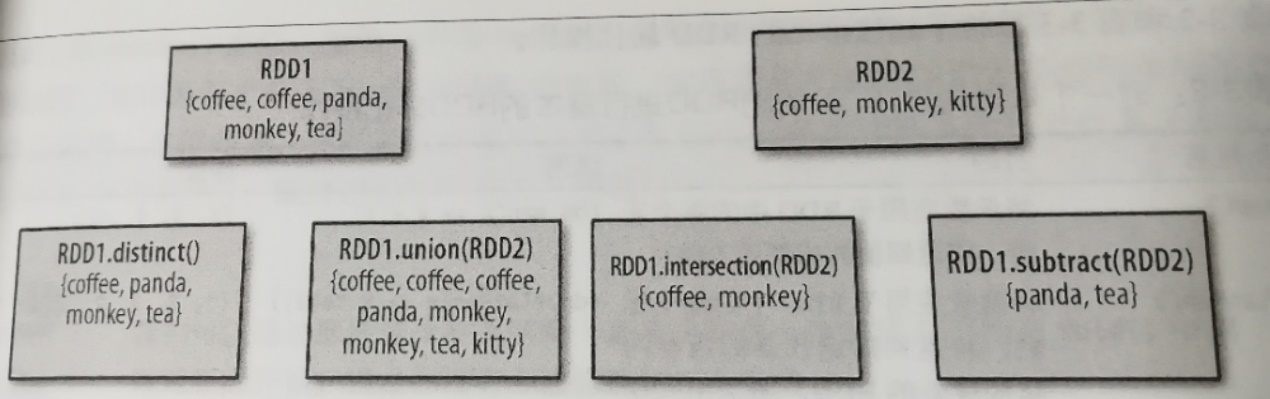
|  |
| --- |
| Map算子和 MapPartition的区别：MapPartition可能会出现OOM溢出，RDD，发送到Executor上去执行，如果发现Executor的内存不够了，就会触发GC，但是Partiton中还有其他的值要使用到，所以无法回收。 |

#### RDD转换算子小记：

Map算子：**一个输入，得到一个输出**；输入的类型和输出的类型可以不一致。

flapMap算子：一个输入，希望得到多个输出，（比如一个文件输入了一行，得到了单词的序列）

RDD本身并不是严格意义的集合，但是它支持很多数学上的集合操作，相比较于集合，它最缺失的就是集合的唯一性：如果RDD出现了 重复的元素，**可以使用distinct去重，但是这会shuffle**，关于集合的相关操作，参加下面的这张图片：



最常用的执行算子：reduce，**该算子接收一个函数，该函数操作两个相同数据类型的元素，并返回一个相同数据类型的数据**。

fold算子需要传入两个参数，第一参数初始值，来作为每个分区第一次调用时的结果，该初始值应该是你所提供的操作的单位元素。

**RDD的执行算子，会将RDD的执行结果返回到Driver。**

foreach也是一个执行算子，没有返回值。

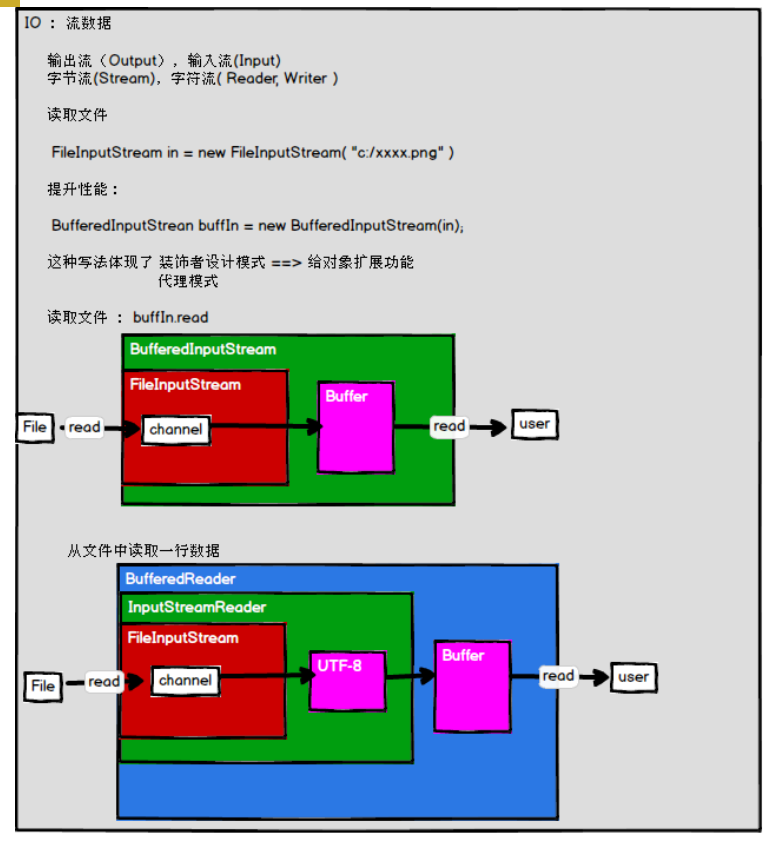
|  |
| --- |
| Glom算子：   **Glom算子求最大值**   Sample算子涉及到的两个概率分布： **if** (withReplacement) {  **new** PartitionwiseSampledRDD[T, T](**this**, **new** PoissonSampler[T](fraction), **true**, seed) } **else** {  **new** PartitionwiseSampledRDD[T, T](**this**, **new** BernoulliSampler[T](fraction), **true**, seed) }  **withReplacement**  **是第一个参数，表示是否放回，false表示不放回。**  **在传入的第一个参数为false的时候，使用伯努利分布，也即零一分布，传入的第二个参数参数是时间发生的概率，1表示一定发生，0 表示不发生。**  关于这两个分布的解释如下：  伯努利分布式？  伯努利分布也叫做0-1分布或者是两点分布，如果伯努利实验成功，则伯努利随机变量取值为1，如果伯努利实验失败，则伯努利随机变量为0；  成功的概率是p，失败的概率是q=1-p，数学期望是 1 \* p +0 \* q = p  泊松分布适合描述单位时间内随机事件发生的次数的概率分布。比如一台服务器在一定的时间内收到请求的次数，参数λ是单位时间内（或者单位面积）随机事件发生的概率 Distinct算子： **def** distinct(numPartitions: Int)(**implicit** ord: Ordering[T] = **null**): RDD[T] = withScope {  map(x => (x, **null**)).reduceByKey((x, y) => x, numPartitions).map(\_.\_1) }  该算子是有shuffle的过程的。 Reparation 和 coalesce算子 Reparation一定会进行shuffle，但是coalesce算子不一定会进行shuffle过程。  **def** repartition(numPartitions: Int)(**implicit** ord: Ordering[T] = **null**): RDD[T] = withScope { coalesce(numPartitions, shuffle = **true**) } sortBy算子：   SortBy 对传入的函数处理之后的结果，对原数据进行排序。  **关于Shuffle的特别说明：**  调用的shuffle的算子的时候，会涉及到数据的重组（聚合的时候会发生重新分区），数据的重组会涉及到跨Executor（跨机器）的操作，所以会有数据落盘的操作，体现出来就是分成不同的阶段。 reduceByKey和groupByKey的区别： reduceByKey在shuffle之前会有一个预聚合，groupByKey按照key分组，直接进行shuffle、 aggregateByKey的用法   这里有三个参数，初始值（第一次遇见这个key的时候，给定的初始值），分区内的聚合规则，分区间的聚合规则。上面的例子是，分区内求最大，分区间求和的操作。FoldByKey用于分区间和分区内的规则一样的时候。 CombineByKey 的初始值是可以写逻辑的，而不再是一个单纯的初始值，计算相同key’累加之后的和：       Join和cogroup算子 Join大约的等于内连接，cogroup大约等于外连接。 |

#### RDD行动算子

在没有执行行动算子之前，spark将RDD的数据结构准备好，执行行动算子即是将作业提交了，提交之后，由spark去执行。

|  |
| --- |
| 在执行了行动算子之后，要提交的真正的job就是  handleJobSubmitted方法中的：  **val** job = **new** ActiveJob(jobId, finalStage, callSite, listener, properties) aggregateByKey和aggregate算子的区别 + fold： 第一个是转换算子，剩余两个是执行算子。      第一个函数将RDD中得数据放入到累加器，每个节点都是在Executor中累加的，所以第二个函数用来将两个累加器相加。  Aggregate在分区之间，也会使用到初始值。 |

### 装饰者模式与IO流



图中

### Spark中的

### Spark的调优策略

## SparkSQL

Hive是Hadoop上的SQL引擎，Spark SQL编译的时候也可以包含Hive支持，也可以不不包含Hive

## SparkStreaming

用于对流式数据的处理，也即实时处理。**数据一条一条的过来，SparkStraming会将一个周期中的数据封装成RDD，RDD（也即多个RDD）的序列在组成一个DStream（离散化流）**然后在交给Executor去执行。也叫做为微批次处理，Strom可以作为数来一条处理一条。

sparkStreaming的数据常来自于Kafka，kafka的数据来自flume，kafak可以存数据，flume不能存放数据。，Kafka保证数据不丢失的方法是：Ack应答机制为-1或者all（leader和follower都有数据之后才发下一条）+ 副本。

### SparkStreaming的架构图



接收器的作用是：接收实时的数据，并将一段时间内的数据转化为RDD。

## Spark的调优策略

## Scala常用的语法特性：

在spark的源码中，有一些难以理解的scala语法，这里给出解析

### 调用函数时使用{}来传递参数

|  |
| --- |
| ClassTag 表示可推断类型 map[U: ClassTag]  (f: T => U) 表示类型是可以变化的。  **def** parallelize[T: ClassTag](  seq: Seq[T],  numSlices: Int = defaultParallelism): RDD[T] = withScope {  assertNotStopped()  **new** ParallelCollectionRDD[T](**this**, seq, numSlices, Map[Int, Seq[String]]()) }  defaultParallelism 是默认并行度，这是一个方法。  Parallelize 方法调用了 withScope方法  其中withScope{ } 也是已给方法，{ }中放置的是该函数的的参数，Scala中的  = test（x,y,z）可以直接变为  = test{  X,  Y,  Z  }  如果test方法没有参数，直接 = test  ----------------------------------------------------------------------  TraversableOnce 表示的是可迭代的  **def** flatMap[U: ClassTag](f: T => TraversableOnce[U])  表示的是转换为一个可迭代的类型。也就是说T可以是任意的类型，但是转换之后，必须是可迭代的类型。  **package** com.isea.spark.scala\_way **object** TestT {  **def** main(args: Array[String]): Unit = {  **def** mul(x :Int)(y:Int) = x \* y // 参数以多个括号的方式罗列在函数的后面  *println*(mul(13)(3))  } }  在Scala中，ConsumerRecord[String,String]就表示了一个Map，[K,V]指代的是K，和V 的泛型。  Map集合中的元素，实际上就是Truple2类型，也就是Map中的元素即为二元组。  val map3 = mutable.Map(("Alice" , 10), ("Bob" , 20), ("Kotlin" , 30))  Scala中map.get(key) 如果key存在，则返回Some，如果key不存在则返回None，两者都是Option对象。  在算子中，算子（=>）就表示该算子需要传入一个函数，能够接收一个函数作为参数的函数叫做高阶函数，比如  Test(f:Double=>Double){  f(3.2)  }这里面，f就是形式参数的名字，Double表示传入的函数的形参是Double,并且该函数的返回类型也是Double，=> 仅仅就是一个规定。  另外=>的用法：在模式匹配中，相当于java 中switch的 ：  体会一下scala的能省则省：  **object** TTT {  **def** main(args: Array[String]): Unit = {  **val** result1: Array[Int] = *Array*(1,2,3).map(*plus2*)  *println*(result1.mkString(",")) // 3,4,5   **val** result2: Array[Int] = *Array*(1,2,3).map(*plus3*(\_))  *println*(result2.mkString("--")) // 4--5--6  }  **def** plus3(x : Int) = {  x + 3  }   **def** plus2(x : Int) = x + 2   */\*\*  \* scala这个语言，是能省则省，如果函数只有一行，干脆就写一行  \* map传入的参数确定是Int类型的，plus2传入的参数也是Int，推断出来了，直接省掉  \*/* }  体会一下匿名函数：  **object** TTT {  **def** main(args: Array[String]): Unit = {  **val** mul3 = (x : Int) => x \* 3  *println*(mul3(4)) // 12  }   */\*\*  \* 匿名函数：就是没有名字的函数， (x:Int)是函数的形参部分，x \* 3 是函数体，  \* 返回类型有类型推导完成  \* mul3是代表该该匿名函数的变量，通过他可以实现函数的调用  \*/* }  体会高阶函数：能够接收一个函数作为参数的函数，即为高阶函数  **object** TTT {  **def** main(args: Array[String]): Unit = {  **val** f1 = *minusxy*(10)  *println*("f1=" + f1) // function1  *println*(f1(1)) // 9  *println*(f1(2)) // 8  *println*(*minusxy*(20)(9)) // 20 - 9 = 11   //f1 就是 (y :int ) = { x - y}  }   //高阶函数，可以返回一个函数类型  //1. minusxy 返回的类型是函数 (y: Int) => x - y  //2. 返回的匿名函数，使用到本身函数的外部变量 x, 这时匿名函数和x 构成一个闭包  一个函数和其相关的引用环境（变量）组合成一个整体。  //3. 匿名函数  **def** minusxy(x: Int) = (y: Int) => {x - y}   **def** minusxyM(x: Int) = {  (y: Int) => {x - y}  }    */\*\*  \*  说明: def minusxy(x: Int) = (y: Int) => x - y  \* 1) 函数名为 minusxy  \* 2) 该函数返回一个匿名函数  \* (y: Int) = > x -y  \*  \*  说明val result3 = minusxy(3)(5)  \*  \* 1) minusxy(3)执行minusxy(x: Int)得到 (y: Int) => x - y 这个匿名函  \* 2) minusxy(3)(5)执行 (y: Int) => x - y 这个匿名函数  \* 3) 也可以分步执行: val f1 = minusxy(3); val res = f1(90)  \*/* }  类型推断时候的省略：  当传入的函数，只有单个参数的时候，可以省略括号  如果变量只在=>右边只出现一次，可以使用\_来替代。  **object** TTT {  **def** main(args: Array[String]): Unit = {  **val** list = *List*(1,2,3)   list.map((x:Int) => x + 1) // 正常的写法，传如一个匿名函数   list.map((x) => x + 1) // 编译器可以推断出来类型，因此类型省略   list.map(x => x + 1) // 传入的函数，只有单个参数，因此类型括号   *println*(list.map(\_ + 1))// 变量只在=>的右边出现一次，使用\_替代，并将参数去掉   *println*(list.map{  \_ + 1  }  ) // 使用{}可以放置多行代码   }   }  的  的  的  的 |