

Spark介绍

2018年2月4日 17:21



Apache Spark™ is a fast and general engine for large-scale data processing.

Spark Introduce

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

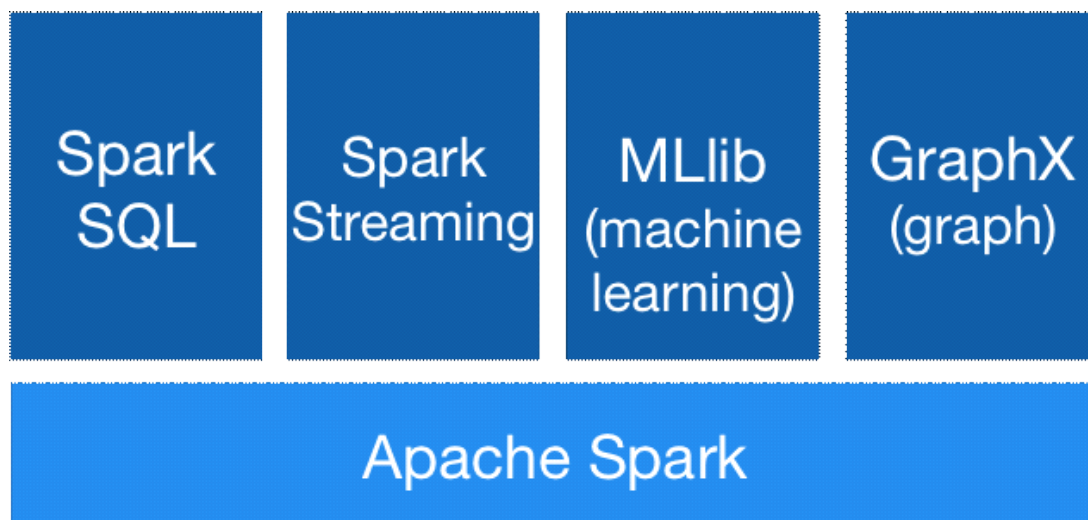
Apache Spark has an advanced **DAG execution** engine that supports acyclic data flow and in-memory computing.

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including [SQL and DataFrames](#), [MLlib](#) for machine learning, [GraphX](#), and [Spark Streaming](#). You can combine these libraries seamlessly in the same application.



Spark是UC Berkeley AMP lab (加州大学伯克利分校的AMP实验室)所开源的，后贡献给Apache。是一种快速、通用、可扩展的大数据分析引擎。它是不断壮大的大数据分析解决方案家族中备受关注的明星成员，为分布式数据集的处理提供了一个有效框架，并以高效的方式处理分布式数据集。**Spark集批处理、实时流处理、交互式查询、机器学习与图计算于一体**，避免了多种运算场景下需要部署不同集群带来的资源浪费。目前，Spark社区也成为大数据领域和Apache软件基金会最活跃的项目之一，其活跃度甚至远超曾经只能望其项背的Hadoop。

Spark的技术背景

无论是工业界还是学术界，都已经广泛使用高级集群编程模型来处理日益增长的数据，如MapReduce和Dryad。这些系统将**分布式编程简化**为自动提供位置感知性调度、容错以及负载均衡，使得大量用户能够在商用集群上分析超大数据集。

大多数现有的集群计算系统都是**基于非循环的数据流模型**。即从稳定的物理存储（如分布式文件系统）中加载记录，记录被传入由一组确定性操作

构成的DAG (Directed Acyclic Graph, 有向无环图), 然后写回稳定存储。DAG数据流图能够在运行时自动实现任务调度和故障恢复。

尽管非循环数据流是一种很强大的抽象方法, 但仍然有些应用无法使用这种方式描述。这类应用包括: ①机器学习和图应用中常用的迭代算法 (每一步对数据执行相似的函数)

②交互式数据挖掘工具 (用户反复查询一个数据子集)

基于数据流的框架并不明确支持工作集, **所以需要将数据输出到磁盘, 然后在每次查询时重新加载**, 这会带来较大的开销。针对上述问题, Spark实现了一种分布式的**内存抽象, 称为弹性分布式数据集**

(Resilient Distributed Dataset, **RDD**)。它支持基于工作集的应用, 同时具有数据流模型的特点: 自动容错、位置感知性调度和可伸缩性。**RDD允许用户在执行多个查询时显式地将工作集缓存在内存中, 后续的查询能够重用工作集, 这极大地提升了查询速度。**

Spark VS MapReduce

2018年2月4日 17:52

MapReduce存在的问题

一个 Hadoop job 通常都是这样的：

- 1) 从 HDFS 读取输入数据；
- 2) 在 Map 阶段使用用户定义的 mapper function, 然后把结果Spill到磁盘；
- 3) 在 Reduce 阶段，从各个处于 Map 阶段的机器中读取 Map 计算的中间结果，使用用户定义的 reduce function, 通常最后把结果写回 HDFS;

Hadoop的问题在于，一个 Hadoop job 会**进行多次磁盘读写**，比如写入机器本地磁盘，或是写入分布式文件系统中（这个过程包含磁盘的读写以及网络传输）。考虑到磁盘读取比内存读取慢了几个数量级，**所以像 Hadoop 这样高度依赖磁盘读写的架构就一定会有性能瓶颈。**

此外，在实际应用中我们通常需要设计复杂算法处理海量数据, 而且不是一个 Hadoop job 可以完成的。比如机器学习领域，需要大量使用迭代的方法训练机器学习模型。而像 Hadoop 的基本模型就只包括了一个 Map 和一个 Reduce 阶段，想要完成复杂运算就需要切分出无数单独的 Hadoop jobs, 而且每个 Hadoop job 都是磁盘读写大户，这就让 Hadoop 显得力不从心。

随着业界对大数据使用越来越深入，大家都呼唤一个更强大的处理框架，能够真正解决更多复杂的大数据问题。

Spark的优势

2009年，美国加州大学伯克利分校的 AMPLab 设计并开发了名叫 Spark 的大数据处理框架。真如其名，Spark 像燎原之火，迅猛占领大数据处理框架市场。

Spark 没有像 Hadoop 一样使用磁盘读写，而转用性能高得多的**内存**存储输入数据、处理中间结果、和存储最终结果。在大数据的场景中，很多计算都有循环往复的特点，像 Spark 这样**允许在内存中缓存输入输出**，上一个 job 的结果马上可以被下一个使用，性能自然要比 Hadoop MapReduce 好得多。

同样重要的是，Spark 提供了更多灵活可用的数据操作，比如 filter, join, 以及各种对 key value pair 的方便操作，甚至提供了一个通用接口，让用户根据需要开发定制的数据操作。

此外，Spark 本身作为平台也开发了 streaming 处理框架 spark streaming, SQL 处理框架 Dataframe, 机器学习库 MLlib, 和图处理库 GraphX. 如此强大，如此开放，基于 Spark 的操作，应有尽有。

Hadoop 的 MapReduce 为什么不使用内存存储？

是历史原因。当初 MapReduce 选择磁盘，除了要保证数据存储安全以外，更重要的是当时企业级数据中心购买大容量内存的成本非常高，选择基于内存的架构并不现实；现在 Spark 真的赶上了好时候，企业可以轻松部署多台大内存机器，内存大到可以装载所有要处理的数据。

Spark单机模式安装

2018年2月4日 17:33

实现步骤:

- 1) 安装和配置好JDK
- 2) 上传和解压Spark安装包
- 3) 进入Spark安装目录下的conf目录

复制conf spark-env.sh.template 文件为 spark-env.sh

在其中修改, 增加如下内容:

SPARK_LOCAL_IP=服务器IP地址

Spark单机模式启动

在bin目录下执行: sh spark-shell --master=local

启动后 发现打印消息

Spark context Web UI available at <http://192.168.242.101:4040//Spark>的浏览器界面

Spark Jobs (?)

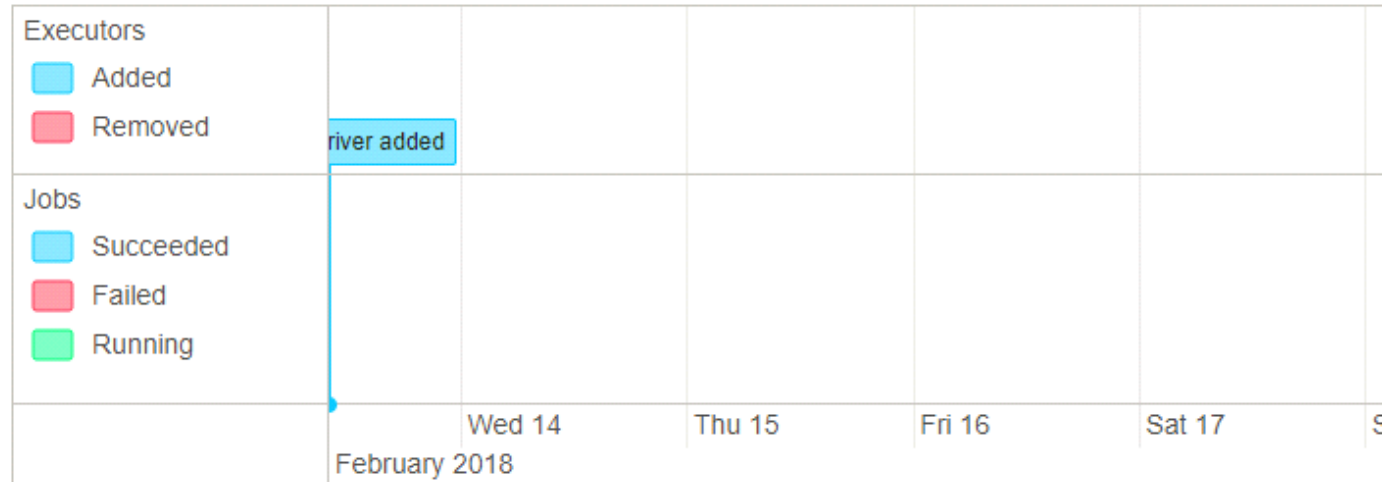
User: root

Total Uptime: 37 s

Scheduling Mode: FIFO

▼ Event Timeline

☐ Enable zooming



Spark context available as 'sc' (master = local, app id = local-1490336686508).//Spark

提供了环境对象 sc

Spark session available as 'spark'.//Spark提供了会话独享spark

RDD介绍

2018年2月4日 17:34

概述

Resilient Distributed Datasets (RDDs)

Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a **fault-tolerant collection** of elements that can be operated on **in parallel**. There are two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

RDD就是带有分区的集合类型

弹性分布式数据集 (RDD) , 特点是可以并行操作, 并且是容错的。有两种方法可以创建RDD:

- 1) 执行Transform操作 (变换操作) ,
- 2) 读取外部存储系统的数据集, 如HDFS, HBase, 或任何与Hadoop有关的数据源。

RDD入门示例

案例一:

Parallelized collections are created by calling `SparkContext`'s `parallelize` method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:


```
val data = Array(1, 2, 3, 4, 5)
val r1 = sc.parallelize(data)
val r2 = sc.parallelize(data,2)
```

你可以这样理解RDD：它是spark提供的一个特殊集合类。诸如普通的集合类型，如传统的Array：(1,2,3,4,5) 是一个整体，但转换成RDD后，我们可以对数据进行Partition（分区）处理，这样做的目的就是为了分布式。

你可以让这个RDD有两个分区，那么有可能是这个形式：RDD(1,2)(3,4)。

这样设计的目的在于：可以进行分布式运算。

注：创建RDD的方式有多种，比如案例一中是基于一个基本的集合类型（Array）转换而来，像parallelize这样的方法还有很多，之后就会学到。此外，我们也可以在读取数据集时就创建RDD。

案例二：

Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, [Amazon S3](#), etc. Spark supports text files, [SequenceFiles](#), and any other Hadoop [InputFormat](#).

Text file RDDs can be created using SparkContext's textFile method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines. Here is an example invocation:

```
val distFile = sc.textFile("data.txt")
```

[查看RDD](#)

```
scala>rdd.collect
```

收集rdd中的数据组成Array返回，此方法将会把分布式存储的rdd中的数据**集中**到一台机器中组建Array。

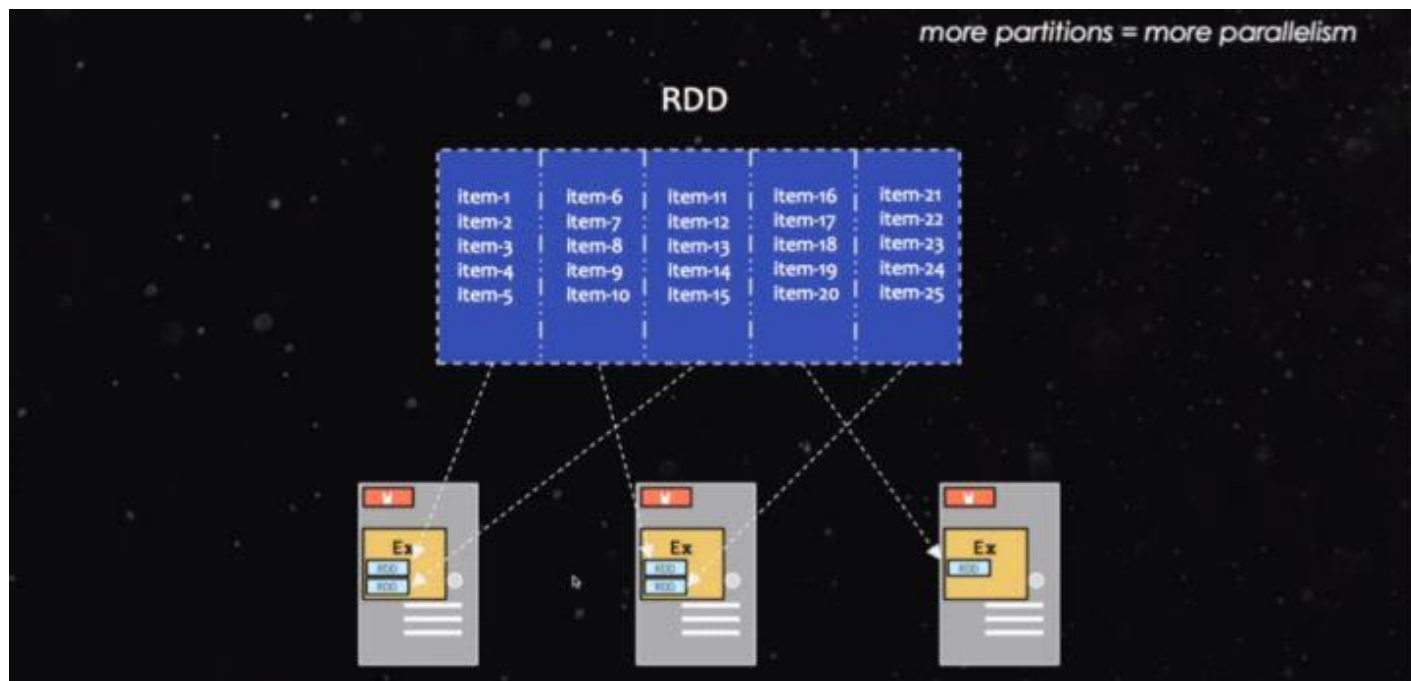
在生产环境下一定要**慎用这个方法**，容易内存溢出。

查看RDD的分区数量： `scala>rdd.partitions.size`

查看RDD每个分区的元素： `scala>rdd.glom.collect`

此方法会将每个分区的元素以Array形式返回

分区概念



在上图中，一个RDD有item1~item25个数据，共5个分区，分别在3台机器上进行处理。

此外，spark并没有原生的提供rdd的分区查看工具 我们可以自己来写一个**示例代码**：

```
import org.apache.spark.rdd.RDD
```

```

import scala.reflect.ClassTag

object su {

def debug[T: ClassTag](rdd: RDD[T]) = {

rdd.mapPartitionsWithIndex((i: Int, iter: Iterator[T]) => {

val m = scala.collection.mutable.Map[Int, List[T]]()

var list = List[T]()

while (iter.hasNext) {

list = list :+ iter.next

}

m(i) = list

m.iterator

}).collect().foreach((x: Tuple2[Int, List[T]]) => {

val i = x._1

println(s"partition:[$i]")

x._2.foreach { println }

}))

}

}

```

RDD操作

2018年2月4日 20:09

概述

针对RDD的操作，分两种，一种是Transformation（变换），一种是Actions（执行）。

Transformation（变换）操作属于懒操作（算子），不会真正触发RDD的处理计算。

每执行一次懒操作，都会创一个新的RDD，而且不会马上计算。

Actions（执行）操作才会真正触发。

Transformations

Transformation	Meaning
<code>map(func)</code>	<p>Return a new distributed dataset formed by passing each element of the source through a function <i>func</i>.</p> <p>参数是函数，函数应用于RDD每一个元素，返回值是新的RDD</p> <p>案例展示：</p> <p>map 将函数应用到rdd的每个元素中</p> <pre>val rdd = sc.makeRDD(List(1,3,5,7,9)) rdd.map(_*10)</pre>
<code>flatMap(func)</code>	<p>Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).</p> <p>扁平化map，对RDD每个元素转换,然后再扁平化处理</p> <p>案例展示：</p> <p>flatMap 扁平map处理</p> <pre>val rdd = sc.makeRDD(List("hello world","hello count","world spark"),2) rdd.map(_split{" "})//Array(Array(hello, world), Array(hello, count), Array(world, spark)) rdd.flatMap(_split{" "})//Array[String] = Array(hello, world, hello, count, world, spark) //Array[String] = Array(hello, world, hello, count, world, spark)</pre> <p>注：map和flatMap有何不同？</p> <p>map: 对RDD每个元素转换</p> <p>flatMap: 对RDD每个元素转换,然后再扁平化（即去除集合）</p> <p>所以，一般我们在读取数据源后，第一步执行的操作是flatMap</p>
<code>filter(func)</code>	<p>Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true. 参数是函数，函数会过滤掉不符合条件的元素，返回值是新的RDD</p> <p>案例展示：</p> <p>filter 用来从rdd中过滤掉不符合条件的数据</p> <pre>val rdd = sc.makeRDD(List(1,3,5,7,9));</pre>

	<pre>rdd.filter(_<5);</pre>
<pre>mapPartitions(func)</pre>	<p>Similar to map, but runs separately on each partition (block) of the RDD, so <i>func</i> must be of type <code>Iterator<T> => Iterator<U></code> when running on an RDD of type T.</p> <p>该函数和map函数类似，只不过映射函数的参数由RDD中的每一个元素变成了RDD中每一个分区的迭代器。</p> <p>案例展示：</p> <pre>val rdd3 = rdd1.mapPartitions{ x => { val result = List[Int]() var i = 0 while(x.hasNext){ i += x.next() } result:::(i).iterator }}</pre> <pre>scala> rdd3.collect</pre> <pre>scala> rdd3.collect res10: Array[Int] = Array(3, 12)</pre> <p>补充：此方法可以用于某些场景的调优，比如将数据存储数据库，如果用map方法来存，有一条数据就会建立和销毁一次连接，性能较低所以此时可以用mapPartitions代替map</p>
<pre>mapPartitionsWithIndex(func)</pre>	<p>Similar to mapPartitions, but also provides <i>func</i> with an integer value representing the index of the partition, so <i>func</i> must be of type <code>(Int, Iterator<T>) => Iterator<U></code> when running on an RDD of type T.</p> <p>函数作用同mapPartitions，不过提供了两个参数，第一个参数为分区的索引。</p> <p>案例展示：</p> <pre>var rdd1 = sc.makeRDD(1 to 5,2) var rdd2 = rdd1.mapPartitionsWithIndex{ (index,iter) => { var result = List[String]() var i = 0 while(iter.hasNext){ i += iter.next() } result:::(index + " " + i).iterator } }</pre> <pre>scala> rdd2.collect res11: Array[String] = Array(0 3, 1 12)</pre> <p>案例展示：</p> <pre>val rdd = sc.makeRDD(List(1,2,3,4,5),2); rdd.mapPartitionsWithIndex((index, iter)=>{ var list = List[String]() while(iter.hasNext){ if(index==0) list = list :+ (iter.next + "a") else { list = list :+ (iter.next + "b") } } })</pre>

	<pre> } list.iterator }); scala> res12.collect res13: Array[String] = Array(1a, 2a, 3b, 4b, 5b) </pre>
union (<i>otherDataset</i>)	<p>Return a new dataset that contains the union of the elements in the source dataset and the argument.</p> <p>案例展示:</p> <p>union 并集 -- 也可以用++实现</p> <pre> val rdd1 = sc.makeRDD(List(1,3,5)); val rdd2 = sc.makeRDD(List(2,4,6,8)); val rdd = rdd1.union(rdd2); val rdd = rdd1 ++ rdd2; </pre> <pre> scala> rdd.collect res17: Array[Int] = Array(1, 3, 5, 2, 4, 6, 8) </pre>
intersection (<i>otherDataset</i>)	<p>Return a new RDD that contains the intersection of elements in the source dataset and the argument.</p> <p>案例展示:</p> <p>intersection 交集</p> <pre> val rdd1 = sc.makeRDD(List(1,3,5,7)); val rdd2 = sc.makeRDD(List(5,7,9,11)); val rdd = rdd1.intersection(rdd2); </pre> <pre> res18: Array[Int] = Array(7, 5) </pre>
subtract	<p>案例展示:</p> <p>subtract 差集</p> <pre> val rdd1 = sc.makeRDD(List(1,3,5,7,9)); val rdd2 = sc.makeRDD(List(5,7,9,11,13)); val rdd = rdd1.subtract(rdd2); </pre>
distinct ([<i>numTasks</i>])	<p>Return a new dataset that contains the distinct elements of the source dataset.</p> <p>没有参数, 将RDD里的元素进行去重操作</p> <p>案例展示:</p> <pre> val rdd = sc.makeRDD(List(1,3,5,7,9,3,7,10,23,7)); rdd.distinct </pre>
groupByKey ([<i>numTasks</i>])	<p>When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.</p> <p>Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using <code>reduceByKey</code> or <code>aggregateByKey</code> will yield much better performance.</p> <p>Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.</p> <p>案例展示:</p>

	<pre>scala>val rdd = sc.parallelize(List(("cat",2), ("dog",5),("cat",4),("dog",3),("cat",6),("dog",3),("cat",9),("dog",1)),2); scala>rdd.groupByKey()</pre> <p>注：groupByKey对于数据格式是有要求的，即操作的元素必须是一个二元tuple，tuple._1是key，tuple._2是value</p> <p>比如下面这种数据格式：</p> <pre>sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)就不符合要求以及这种：</pre> <pre>sc.parallelize(List(("cat",2,1), ("dog",5,1),("cat",4,1),("dog",3,2),("cat",6,2),("dog",3,4),("cat",9,4),("dog",1,4)),2);</pre>
reduceByKey (<i>func</i> , <i>numTasks</i>)	<p>When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function <i>func</i>, which must be of type (V, V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.</p> <p>案例展示：</p> <pre>scala>var rdd = sc.makeRDD(List(("hello",1),("spark",1),("hello",1),("world",1))) rdd.reduceByKey(_+_);</pre> <p>注：reduceByKey操作的数据格式必须是一个二元tuple</p>
aggregateByKey (<i>zeroValue</i> (<i>seqOp</i> , <i>combOp</i> , <i>numTasks</i>))	<p>When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.</p> <p>使用方法及案例展示：</p> <pre>aggregateByKey(zeroValue) (func1, func2)</pre> <ul style="list-style-type: none"> • zeroValue表示初始值，初始值会参与func1的计算 • 在分区内，按key分组，把每组的值进行func1的计算 • 再将每个分区每组的计算结果按func2进行计算 <pre>scala> val rdd = sc.parallelize(List(("cat", 2), ("dog", 5), ("cat", 4), ("dog", 3), ("cat", 6), ("dog", 3), ("cat", 9), ("dog", 1)), 2);</pre> <p>查看分区结果：</p> <pre>partition:[0] (cat, 2) (dog, 5) (cat, 4) (dog, 3) partition:[1] (cat, 6) (dog, 3) (cat, 9)</pre>

	<pre>(dog, 1)</pre> <pre>scala> rdd.aggregateByKey(0)(_+_ , _*_);</pre> <pre>(dog,12), (cat,21)</pre> <pre>scala> rdd.aggregateByKey(0)(_+_ , _*_);</pre> <pre>(dog,32), (cat,90)</pre>
sortByKey ([ascending], [numTasks])	<p>When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.</p> <p>案例展示:</p> <pre>val d2 = sc.parallelize(Array(("cc", 32), ("bb", 32), ("cc", 22), ("aa", 18), ("bb", 6), ("dd", 16), ("ee", 104), ("cc", 1), ("ff", 13), ("gg", 68), ("bb", 44)))</pre> <pre>d2.sortByKey(true).collect</pre> <pre>res31: Array[(String, Int)] = Array((aa,18), (bb,6), (bb,44), (bb,32), (cc,32), (cc,22), (cc,1), (dd,16), (ee,104), (ff,13), (gg,68))</pre>
join (otherDataset, [numTasks])	<p>When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.</p> <p>案例展示:</p> <pre>val rdd1 = sc.makeRDD(List(("cat",1),("dog",2))) val rdd2 = sc.makeRDD(List(("cat",3),("dog",4),("tiger",9))) rdd1.join(rdd2);</pre>
cartesian (otherDataset)	<p>When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).</p> <p>参数是RDD，求两个RDD的笛卡尔积</p> <p>案例展示:</p> <p>cartesian 笛卡尔积</p> <pre>val rdd1 = sc.makeRDD(List(1,2,3)) val rdd2 = sc.makeRDD(List("a","b")) rdd1.cartesian(rdd2);</pre>
coalesce (numPartitions)	<p>Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.</p> <p>coalesce(n,true/false) 扩大或缩小分区</p> <p>案例展示:</p> <pre>val rdd = sc.makeRDD(List(1,2,3,4,5),2) rdd.coalesce(3,true);//如果是扩大分区 需要传入一个true 表示要重新shuffle rdd.coalesce(2);//如果是缩小分区 默认就是false 不需要明确的传入</pre>

repartition (<i>numPartitions</i>)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. repartition(n) 等价于上面的 coalesce

Actions

Action	Meaning
reduce (<i>func</i>)	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. 并行整合所有RDD数据，例如求和操作
collect ()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. 返回RDD所有元素，将rdd分布式存储在集群中不同分区的数据 获取到一起组成一个数组返回 要注意 这个方法将会把所有数据收集到一个机器内，容易造成内存的溢出 在生产环境下千万慎用
count ()	Return the number of elements in the dataset. 统计RDD里元素个数 案例展示： <pre>val rdd = sc.makeRDD(List(1,2,3,4,5),2) rdd.count</pre>
first ()	Return the first element of the dataset (similar to take(1)).
take (<i>n</i>)	Return an array with the first <i>n</i> elements of the dataset. 案例展示： take 获取前几个数据 <pre>val rdd = sc.makeRDD(List(52,31,22,43,14,35)) rdd.take(2)</pre>
takeOrdered (<i>n</i> , [<i>ordering</i>])	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator. 案例展示： takeOrdered(n) 先将rdd中的数据进行升序排序 然后取前n个 <pre>val rdd = sc.makeRDD(List(52,31,22,43,14,35)) rdd.takeOrdered(3)</pre>
top (<i>n</i>)	top(n) 先将rdd中的数据进行降序排序 然后取前n个 <pre>val rdd = sc.makeRDD(List(52,31,22,43,14,35)) rdd.top(3)</pre>
saveAsTextFile (<i>path</i>)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. 案例示例： saveAsTextFile 按照文本方式保存分区数据 <pre>val rdd = sc.makeRDD(List(1,2,3,4,5),2);</pre>

	<code>rdd.saveAsTextFile("/root/work/aaa")</code>
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(<i>func</i>)	<p>Run a function <i>func</i> on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.</p> <p>Note: modifying variables other than Accumulators outside of the <code>foreach()</code> may result in undefined behavior. See Understanding closures for more details.</p>

案例：通过rdd实现统计文件中的单词数量

```
sc.textFile("/root/work/words.txt").flatMap(_.split(" ")).map(_._1).reduceByKey(_+_).saveAsTextFile("/root/work/wcresult")
```

DAG概念

2018年2月22日 13:56

概述

Spark会根据用户提交的计算逻辑中的**RDD的转换和动作来生成RDD之间的依赖关系**，同时这个计算链也就生成了逻辑上的DAG。接下来以“Word Count”为例，详细描述这个DAG生成的实现过程。

Spark Scala版本的Word Count程序如下：

```
1: val file=sc.textFile("hdfs://hadoop01:9000/hello1.txt")
2: val counts = file.flatMap(line => line.split(" "))
3:      .map(word => (word, 1))
4:      .reduceByKey(_ + _)
5: counts.saveAsTextFile("hdfs://...")
```

file和counts都是RDD，其中file是从HDFS上读取文件并创建了RDD，而counts是在file的基础上通过flatMap、map和reduceByKey这三个RDD转换生成的。最后，counts调用了动作saveAsTextFile，用户的计算逻辑就从这里开始提交的集群进行计算。那么上面这5行代码的具体实现是什么呢？

1) 行1：sc是org.apache.spark.SparkContext的实例，它是用户程序和Spark的交互接口，会负责连接到集群管理者，并根据用户设置或者系统默认设置来申请计算资源，完成RDD的创建等。

sc.textFile ("hdfs: //...") 就完成了org.apache.spark.rdd.HadoopRDD的创建，并且完成了一次RDD的转换：通过map转换到一个org.apache.spark.rdd.MapPartitions-RDD。也就是说，file实际上是一个MapPartitionsRDD，它保存了文件的所有行的数据内容。

2) 行2: 将file中的所有行的内容, 以空格分隔为单词的列表, 然后将这个按照行构成的单词列表合并为一个列表。最后, 以每个单词为元素的列表被保存到MapPartitionsRDD。

3) 行3: 将第2步生成的MapPartitionsRDD再次经过map将每个单词word转为 (word, 1) 的元组。这些元组最终被放到一个MapPartitionsRDD中。

4) 行4: 首先会生成一个MapPartitionsRDD, 起到map端combiner的作用; 然后会生成一个ShuffledRDD, 它从上一个RDD的输出读取数据, 作为reducer的开始; 最后, 还会生成一个MapPartitionsRDD, 起到reducer端reduce的作用。

5) 行5: 向HDFS输出RDD的数据内容。最后, 调用 `org.apache.spark.SparkContext#runJob` 向集群提交这个计算任务。

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, 也就是RDD的 parent RDD (s) 是什么; 还有就是依赖于parent RDD (s) 的哪些Partition (s) 。这个关系, 就是RDD之间的依赖, `org.apache.spark.Dependency`。根据依赖于parent RDD (s) 的Partitions的不同情况, Spark将这种依赖分为两种, 一种是**宽依赖**, 一种是**窄依赖**。

RDD的依赖关系

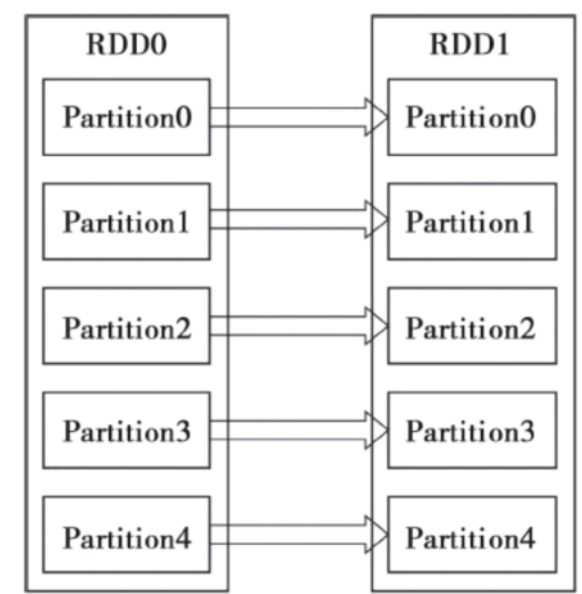
2018年2月22日 14:20

RDD的依赖关系

RDD和它依赖的parent RDD (s) 的关系有两种不同的类型，即**窄依赖**

(narrow dependency) 和**宽依赖** (wide dependency) 。

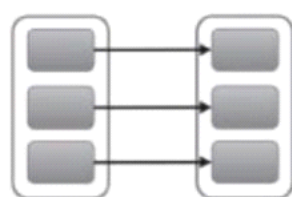
1) 窄依赖指的是每一个parent RDD的Partition最多被子RDD的一个Partition使用，如下图所示。



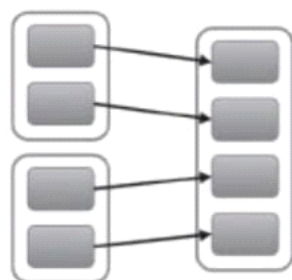
2) 宽依赖指的是多个子RDD的Partition会依赖同一个parent RDD的Partition。

我们可以从不同类型的转换来进一步理解RDD的窄依赖和宽依赖的区别，如下图所示。

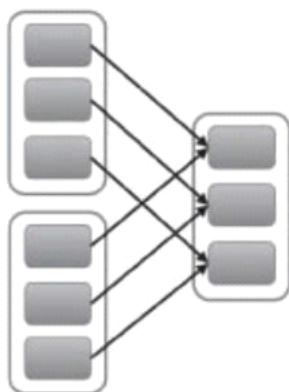
窄依赖：



map, filter

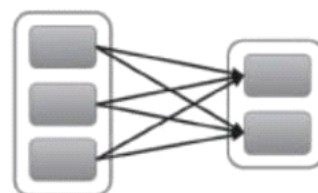


union

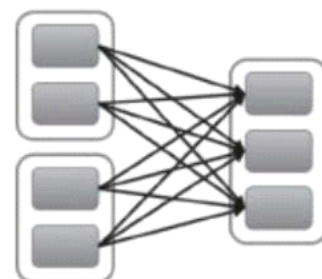


join with
inputs co-
partitioned

宽依赖：



groupByKey



join with inputs not
co-partitioned

■ = RDD's Partition

窄依赖

对于窄依赖操作，它们只是将Partition的数据根据转换的规则进行转化，并不涉及其他的处理，可以简单地认为只是将数据从一个形式转换到另一个形式。

窄依赖底层的源码：

```
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {  
  //返回子RDD的partitionId依赖的所有的parent RDD的Partition(s)  
  def getParents(partitionId: Int): Seq[Int]  
  override def rdd: RDD[T] = _rdd  
}  
  
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {  
  override def getParents(partitionId: Int) = List(partitionId)  
}
```

所以对于窄依赖，并不会引入昂贵的Shuffle。所以执行效率非常高。如果整个DAG中存在多个连续的窄依赖，则可以将这些连续的窄依赖整合到一起连续执行，中间不执行shuffle 从而

提高效率，这样的优化方式称之为流水线优化。

此外，针对窄依赖，如果子RDD某个分区数据丢失，只需要找到父RDD对应依赖的分区，恢复即可。

即窄依赖的数据恢复效率较高。

但如果是宽依赖，当分区丢失时，最糟糕的情况是要重算所有父RDD的所有分区。

宽依赖

对于groupByKey这样的操作，子RDD的所有Partition (s) 会依赖于parent RDD的所有Partition (s) ，**子RDD的Partition是parent RDD的所有Partition Shuffle的结果。**

宽依赖的源码：

```
class ShuffleDependency[K, V, C](
  @transient _rdd: RDD[_ <: Product2[K, V]],
  val partitioner: Partitioner,
  val serializer: Option[Serializer] = None,
  val keyOrdering: Option[Ordering[K]] = None,
  val aggregator: Option[Aggregator[K, V, C]] = None,
  val mapSideCombine: Boolean = false)
  extends Dependency[Product2[K, V]] {

  override def rdd = _rdd.asInstanceOf[RDD[Product2[K, V]]]
  //获取新的shuffleId
  val shuffleId: Int = _rdd.context.newShuffleId()
  //向ShuffleManager注册Shuffle的信息
  val shuffleHandle: ShuffleHandle =
    _rdd.context.env.shuffleManager.registerShuffle(
      shuffleId, _rdd.partitions.size, this)

  _rdd.sparkContext.cleaner.foreach(_registerShuffleForCleanup(this))
}
```

Shuffle概述

spark中一旦遇到宽依赖就需要进行shuffle的操作，所谓的shuffle的操作的本质就是将数据汇总后重新分发的过程。

这个过程数据要汇总到一起，数据量可能很大所以不可避免的需要**进行数据落磁盘的操作**，会降低程序的性能，所以spark并不是完全内存不读写磁盘，只能说它尽力避免这样的过程来提高效率。

spark中的shuffle，在早期的版本中，会产生多个临时文件，但是这种多临时文件的策略造成大量文件的的同时的读写，磁盘的性能被分摊给多个文件，每个文件读写效率都不高，影响spark的执行效率。所以在后续的spark中(1.2.0之后的版本)的shuffle中，只会产生一个文件，并且数据会经过排序再附加索引信息，减少了文件的数量并通过排序索引的方式提升了性能。

DAG的生成与Stage的划分

2018年2月22日 14:37

DAG的生成

原始的RDD (s) 通过一系列**转换**就形成了DAG。RDD之间的依赖关系，包含了RDD由哪些Parent RDD (s) 转换而来和它依赖parent RDD (s) 的哪些Partitions，是DAG的重要属性。

借助这些依赖关系，DAG可以认为这些RDD之间形成了Lineage（血统，血缘关系）。借助Lineage，能保证一个RDD被计算前，它所依赖的parent RDD都已经完成了计算；**同时也实现了RDD的容错性**，即如果一个RDD的部分或者全部的计算结果丢失了，那么就需要重新计算这部分丢失的数据。

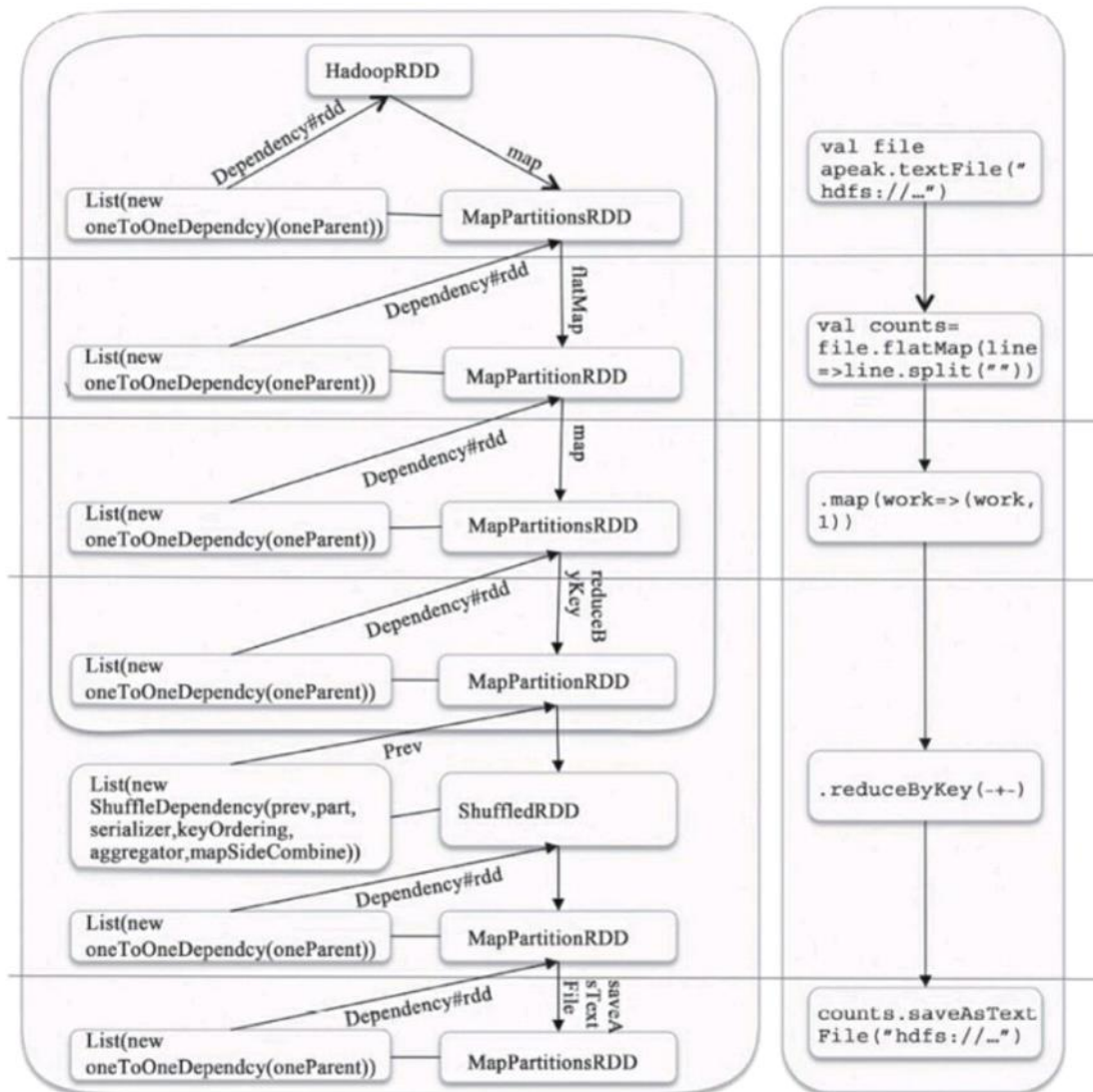
RDD是分布式的，弹性的，容错的数据结构

Spark的Stage（阶段）

Spark在执行任务（job）时，首先会根据依赖关系，将DAG划分为不同的阶段（Stage）。

处理流程是：

- 1) Spark在执行Transformation类型操作时都不会立即执行，而是懒执行（计算）
- 2) 执行若干步的Transformation类型的操作后，一旦遇到Action类型操作时，才会真正触发执行（计算）
- 3) 执行时，从当前Action方法向前回溯，如果遇到的是窄依赖则应用流水线优化，继续向前找，直到碰到某一个宽依赖
- 4) 因为宽依赖必须要进行shuffle，无法实现优化，所以将这一次段执行过程组装为一个stage
- 5) 再从当前宽依赖开始继续向前找。重复刚才的步骤，从而将这个DAG还分为若干的stage



在stage内部可以执行流水线优化，而在stage之间没办法执行流水线优化，因为有shuffle。但是这种机制已经尽力的去避免了shuffle。

Spark的Job和Task

原始的RDD经过一系列转换后（一个DAG），会在最后一个RDD上触发一个动作，这个动作会生成一个Job。

所以可以这样理解：一个DAG对应一个Spark的Job。

在Job被划分为一批计算任务（Task）后，这批Task会被提交到集群上的计算节点去计算

Spark的Task分为两种：

- 1) org.apache.spark.scheduler.ShuffleMapTask
- 2) org.apache.spark.scheduler.ResultTask

简单来说，DAG的最后一个阶段会为每个结果的Partition生成一个ResultTask，其余所有的阶段都会生成Shuff

fleMapTask。

可视化理解窄依赖和宽依赖

案例 单词统计

```
scala>val data=sc.textFile("/home/software/hello.txt",2)
scala> data.flatMap(_._split(" ")).map(_._1).reduceByKey(_+_).collect
```

1) 打开web页面控制台 (ip:4040端口地址) , 刷新, 会发现刚才的操作会出现在页面上

Spark Jobs (?)

User: root
Total Uptime: 17 min
Scheduling Mode: FIFO
Completed Jobs: 2

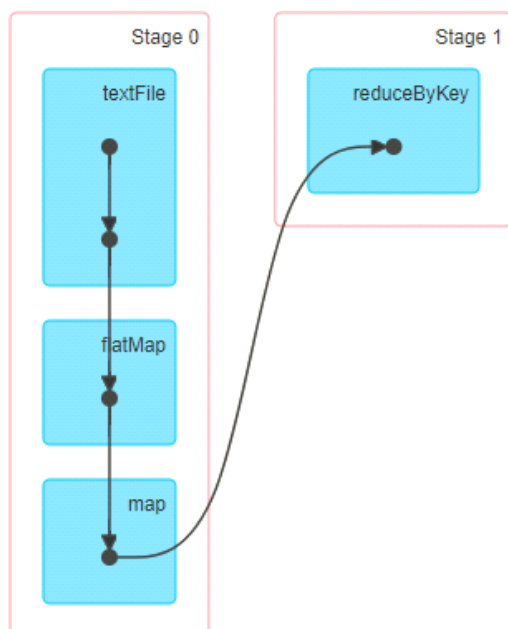
▶ Event Timeline

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <console>:35	2018/02/18 14:44:41	36 ms	1/1	2/2

2) 点击 Description下的 collect at..... 进入job的详细页面

3) 点击 DAG Visualization 会出现如下图形



Spark框架核心概念

2018年4月2日 16:11

1.RDD。弹性分布式数据集，是Spark最核心的数据结构。有分区机制，所以可以分布式进行处理。有容错机制，通过RDD之间的依赖关系来恢复数据。

2.依赖关系。RDD的依赖关系是通过各种**Transformation (变换)** 来得到的。父RDD和子RDD之间的依赖关系分两种：①窄依赖 ②宽依赖

①针对窄依赖：父RDD的分区和子RDD的分区关系是：一对一

窄依赖不会发生Shuffle，执行效率高，spark框架底层会针对多个连续的窄依赖执行流水线优化，从而提高性能。例如 map flatMap等方法都是窄依赖方法

②针对宽依赖：父RDD的分区和子RDD的分区关系是：一对多

宽依赖会产生shuffle，会产生磁盘读写，无法优化。

3.DAG。有向无环图，当一整条RDD的依赖关系形成之后，就形成了一个DAG。一般来说，一个DAG，最后都至少会触发一个Action操作，触发执行。一个Action对应一个Job任务。

4.Stage。一个DAG会根据RDD之间的依赖关系进行Stage划分，流程是：以Action为基准，向前回溯，遇到宽依赖，就形成一个Stage。遇到窄依赖，则执行流水线优化（将多个连续的窄依赖放到一起执行）

5.task。任务。一个分区对应一个task。可以这样理解：一个Stage是一组Task的集合

6.RDD的Transformation (变换) 操作：懒执行，并不会立即执行

7.RDD的Action(执行) 操作：触发真正的执行

Spark集群模式安装

2018年2月4日 22:10

实现步骤:

- 1) 上传解压spark安装包
- 2) 进入spark安装目录的conf目录
- 3) 配置spark-env.sh文件

配置示例:

#本机ip地址

```
SPARK_LOCAL_IP=hadoop01
```

#spark的shuffle中间过程会产生一些临时文件，此项指定的是其存放目录，不配置默认是在
/tmp目录下

```
SPARK_LOCAL_DIRS=/home/software/spark/tmp
```

```
export JAVA_HOME=/home/software/jdk1.8
```

- 4) 在conf目录下，编辑slaves文件

配置示例:

```
hadoop01
```

```
hadoop02
```

```
hadoop03
```

- 5) 配置完后，将spark目录发送至其他节点，并更改对应的 **SPARK_LOCAL_IP** 配置

启动集群

- 1) 如果你想让 01 虚拟机变为master节点，则进入01 的spark安装目录的sbin目录
执行： sh start-all.sh
- 2) 通过jps查看各机器进程，
01： Master +Worker

02: Worker

03: Worker

3) 通过浏览器访问管理界面

<http://192.168.234.11:8080>



Spark Master at spark://hadoop01:7077

URL: spark://hadoop01:7077

REST URL: spark://hadoop01:6066 (*cluster mode*)

Alive Workers: 3

Cores in use: 6 Total, 0 Used

Memory in use: 4.7 GB Total, 0.0 B Used

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers

Worker Id	Address	State	Cores
worker-20180218184550-192.168.234.11-44022	192.168.234.11:44022	ALIVE	2 (0 Used)
worker-20180218184617-192.168.234.210-50603	192.168.234.210:50603	ALIVE	2 (0 Used)
worker-20180218184639-192.168.234.211-48656	192.168.234.211:48656	ALIVE	2 (0 Used)

4) 通过spark shell 连接spark集群

进入spark的bin目录

执行: sh spark-shell.sh --master spark://192.168.234.11:7077

6) 在集群中读取文件:

```
sc.textFile("/root/work/words.txt")
```

默认读取本机数据 这种方式需要在集群的每台机器上的对应位置上都一份该文件 浪费磁盘

7) 所以应该通过hdfs存储数据

```
sc.textFile("hdfs://hadoop01:9000/mydata/words.txt");
```

注: 可以在spark-env.sh 中配置选项 HADOOP_CONF_DIR 配置为hadoop的etc/hadoop的地址 使默认访问的是hdfs的路径

注: 如果修改默认地址是hdfs地址 则如果想要访问文件系统中的文件 需要指明协议为file 例如

```
sc.text("file:///xxx/xx")
```

执行： `spark-submit --class cn.tedu.WordCountDriver /home/software/spark/conf/wc.jar`

扩展：Erasure code（纠删码）

概述

In coding theory, an erasure code is a forward error correction (FEC) code under the assumption of bit erasures (rather than bit errors), which transforms a message of k symbols into a longer message (code word) with n symbols such that the original message can be recovered from a subset of the n symbols.

在编码理论里，有一种前向纠错(FEC)编码方式，也称为纠删码。

这种技术可以将原始数据中丢失的 k 字节数据从 n 个含编码字节的信息中进行恢复。

在纠删码技术中，Reed-Solomon（里所码）码是一种常见的纠删码。

纠删码的应用

对于在分布式环境下数据存储的可靠性保证，有两种策略：

- 1) 引入副本冗余机制策略
- 2) 利用纠删码技术，相比于副本策略，纠删码技术可以节省更多磁盘的空间。即有更高的磁盘利用率

拿Hadoop的HDFS来说，策略一般是三副本，当某个副本丢失时，可以通过其他副本复制回来。所以在这种情况下，Hadoop集群的磁盘利用率为 $1/3$ 。而如果使用纠删码技术后，可以提高磁盘的利用率。

纠删码的思想

What is an Erasure Code?

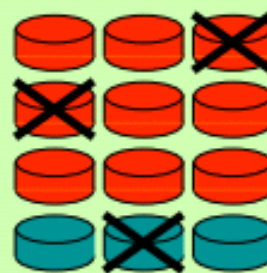
A technique that lets you take n storage devices:



Encode them onto m additional storage devices:



And have the entire system be resilient to up to m device failures:



上图的含义是：有 n 个原始数据块，再引入 m 个数据校验块，然后通过编码（encode）使得原始数据块和数据校验块产生关联。

纠删码技术是一种数据恢复技术，最早用于通信行业中数据传输中的数据恢复，是一种编码容错技术。它通过在原始数据中加入新的校验数据，使得各个部分的数据产生关联性。在一定范围内的数据出错情况下，通过纠删码技术都可以进行恢复。

比如：有原始数据块 n 个，然后加入 m 个校验数据块。

原始数据块和校验数据块在丢失时，都可以通过现有的数据块进行恢复

举例：

① $x=1$

$$\textcircled{2}y=2$$

$$\textcircled{3}z=3$$

$$\textcircled{4}x+y+z=6$$

$$\textcircled{5}2x+3y+z=11$$

$$\textcircled{6}x+2y+3z=14$$

1) 如果我们丢了3个原始数据块，可以恢复

$$\textcircled{4}x+y+z=6$$

$$\textcircled{5}2x+3y+z=11$$

$$\textcircled{6}x+2y+3z=14$$

2) 如果我们丢失了3个数据校验块，可以恢复

$$\textcircled{1}x=1$$

$$\textcircled{2}y=2$$

$$\textcircled{3}z=3$$

3) 可以

$$\textcircled{2}y=2$$

$$\textcircled{3}z=3$$

$$\textcircled{4}x+y+z=6$$

$$\textcircled{5}2x+3y+z=11$$

$$\textcircled{6}x+2y+3z=14$$

4) 可以

$$\textcircled{3}z=3$$

$$\textcircled{4}x+y+z=6$$

$$\textcircled{5} 2x + 3y + z = 11$$

$$\textcircled{6} x + 2y + 3z = 14$$

5) 不可以

$$\textcircled{3} z = 3$$

$$\textcircled{6} x + 2y + 3z = 14$$

扩展：Reed-solomon codes

概述

Reed-Solomon里所码（RS）码是存储系统较为常用的一种纠删码，也称为里所码。它有两个参数n和m，记为RS(n,m)。n代表原始数据块个数。m代表校验块个数。接下来介绍RS码的原理。

Reed-Solomon(RS)码的编码和解码过程

1) 编码过程 (encoding)

2) 解码过程 (decoding)

RS的优缺点：

优点：低冗余度，高磁盘利用率。

比如一个文件有5个文件块，假设用3副本冗余机制，最后总的文件块是15块，则磁盘利用率是： $5/15=1/3=33\%$

而用RS码来实现，5个文件块+3个数据校验块，则磁盘利用率是： $5/8=62.5\%$

缺点：

- 1) 计算代价高。丢失数据块或者编码块时，RS需要读取n个数据块和剩余的校验块才能恢复数据。
- 2) 数据更新代价高。数据更新相当于重新编码，代价很高，因此常常针对只读数据，或者冷数据。

工程实践中，一般对于热数据还是会使用多副本策略来冗余，冷数据使用纠删码。

两种冗余技术对比如下：

两种技术	磁盘利用率	计算开销	网络消耗	恢复效率
多副本(3副本)	1/3	几乎没有	较低	较高
纠删码(n+m)	$n/(n+m)$	高	较高	较低

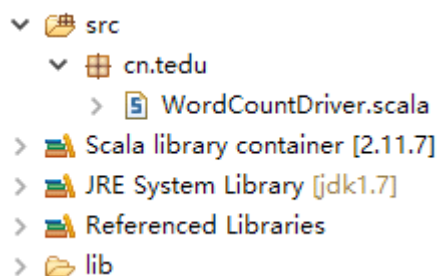
案例—WordCount

2018年3月1日 21:15

实现步骤

1) 创建spark的项目

在scala中创建项目 导入spark相关的jar包



2) 开发spark相关代码

代码示例:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object WordCountDriver {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("spark://hadoop01:7077").setAppName("wordcount")
    val sc=new SparkContext(conf)

    val data=sc.textFile("hdfs://hadoop01:9000/words.txt", 2)
    val result=data.flatMap { x => x.split(" ") }.map { x => (x,1) }.reduceByKey(_+_ )

    result.saveAsTextFile("hdfs://hadoop01:9000/wcrestult")
  }
}
```

3) 将写好的项目打成jar, 上传到服务器, 进入bin目录

执行: spark-submit --class cn.tedu.WordCountDriver /home/software/spark/conf/wc.jar

案例一求平均值

2018年3月2日 17:43

案例文件：

```
1 16
2 74
3 51
4 35
5 44
6 95
7 5
8 29
10 60
11 13
12 99
13 7
14 26
```

正确答案：42

代码示例一：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object AverageDriver {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("AverageDriver")

    val sc=new SparkContext(conf)

    val data=sc.textFile("d://average.txt")

    val ageData=data.map { line=>{line.split(" ")(1).toInt}}
```

```

val ageSum=ageData.reduce(_+_ )

val pepoleCount=data.count()

val average=ageSum/pepoleCount

println(average)
}
}

```

代码示例二:

```

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object AverageDriver {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("AverageDriver")

    val sc=new SparkContext(conf)

    val data=sc.textFile("d://average.txt",3)

    val ageData=data.map { line=>{line.split(" ")(1).toInt}}

    val ageSum=ageData.mapPartitions{it=>{
      val result=List[Int]()
      var i=0
      while(it.hasNext){
        i+=it.next()
      }
      result.+=(i).iterator
    }}.reduce(_+_ )

```



```
val pepopleCount=data.count()
```

```
val average=ageSum/pepopleCount
```

```
println(average)
```

```
}
```

```
}
```

案例一求最大值和最小值

2018年3月2日 17:43

案例文件:

```
1 M 174
2 F 165
3 M 172
4 M 180
5 F 160
6 F 162
7 M 172
8 M 191
9 F 175
10 F 167
```

代码示例一:

```
package cn.tedu

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object MaxMinDriver {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("local").setAppName("MaxMin")
    val sc=new SparkContext(conf)

    val data=sc.textFile("d://MaxMin.txt")

    val manData=data.filter { x => x.contains("M") }.map { x => x.split(" ")(2).toInt}
    val girlData=data.filter { x => x.contains("F") }.map { x => x.split(" ")(2).toInt}

    println("Man Max is:"+manData.max()+"Man min is:"+manData.min())
```

```
}  
}
```

代码示例二:

```
import org.apache.spark.SparkConf  
import org.apache.spark.SparkContext  
  
object MaxMinDriver {  
  
  def main(args: Array[String]): Unit = {  
  
    val conf=new SparkConf().setMaster("local").setAppName("MaxMin")  
    val sc=new SparkContext(conf)  
  
    val data=sc.textFile("d://MaxMin.txt")  
  
    val manMax=data.filter { line => line.contains("M") }.  
      sortBy({line=>line.split(" ")(2)},false).first().mkString  
  
    val manMin=data.filter { line => line.contains("M") }.  
      sortBy({line=>line.split(" ")(2)},true).first.mkString  
  
    println(manMax+"\n"+manMin)  
  
  }  
}
```

代码示例三:

```
import org.apache.spark.SparkConf  
import org.apache.spark.SparkContext  
  
object MaxMinDriver {  
  
  def main(args: Array[String]): Unit = {  
  
    val conf=new SparkConf().setMaster("spark://hadoop01:7077").setAppName("MaxMin")  
    val sc=new SparkContext(conf)
```

```
val data=sc.textFile("hdfs://hadoop01:9000/MaxMin.txt",3)

val manMax=data.filter { line => line.contains("M") }.
  sortBy({line=>line.split(" ")(2)},false).first.mkString

val manMin=data.filter { line => line.contains("M") }.
  sortBy({line=>line.split(" ")(2)},true).first.mkString

val result=sc.makeRDD(Array(manMax,manMin))

//--spark输出文件时，默认是有几个Task,就会生成几个结果文件，
//--所以如果想控制文件个数，控制分区数(task)即可
result.coalesce(1,true).saveAsTextFile("hdfs://hadoop01:9000/MaxMinResult")

}
}
```

案例—TopK

2018年3月1日 21:15

案例说明

Top K算法有两步，一是统计词频，二是找出词频最高的前K个词。

📖 文件数据 topk.txt:

```
hello world bye world
hello hadoop bye hadoop
hello world java web
hadoop scala java hive
hadoop hive redis hbase
hello hbase java redis
```

📖 代码示例:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object TopkDriver {

  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setMaster("local").setAppName("topk")
    val sc = new SparkContext(conf)

    val data = sc.textFile("e://topk.txt", 2)

    val count = data.flatMap { x => x.split(" ") }
      .map { x => (x, 1) }.reduceByKey(_+_ )

    val orderingDesc = Ordering.by [(String, Int), Int](_._2)

    val topk = count.top(3)(orderingDesc)

    //val topk = count.top(3)(Ordering.by{case (word, count) => count})

    topk.foreach{println}
```

```
}  
}
```

应用场景

Top K的示例模型可以应用在求过去一段时间消费次数最多的消费者、访问最频繁的IP地址和最近、更新、最频繁的微博等应用场景。

案例—二次排序

2018年3月2日 19:06

📖 文件数据:

```
aa 12
bb 32
aa 3
cc 43
dd 23
cc 5
cc 8
bb 33
bb 12
```

要求：先按第一列升序排序，再按第二列降序排序

📖 自定义排序类代码:

```
class SecondarySortKey(val first:String,val second:Int) extends Ordered[SecondarySortKey] with
Serializable {
```

```
    def compare(other:SecondarySortKey):Int={
        var comp=this.first.compareTo(other.first)
        if(comp==0){
            other.second.compareTo(this.second)
        }else{
            comp
        }
    }
}
```

📖 Driver代码:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
```

```
object SsortDriver {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("local").setAppName("ssort")
    val sc=new SparkContext(conf)
    val data=sc.textFile("d://ssort.txt",3)

    val ssortData=data.map { line =>{
      (new SecondarySortKey(line.split(" ")(0),line.split(" ")(1).toInt),line)
    }
    }

    val result=ssortData.sortByKey(true)

    result.foreach(println)

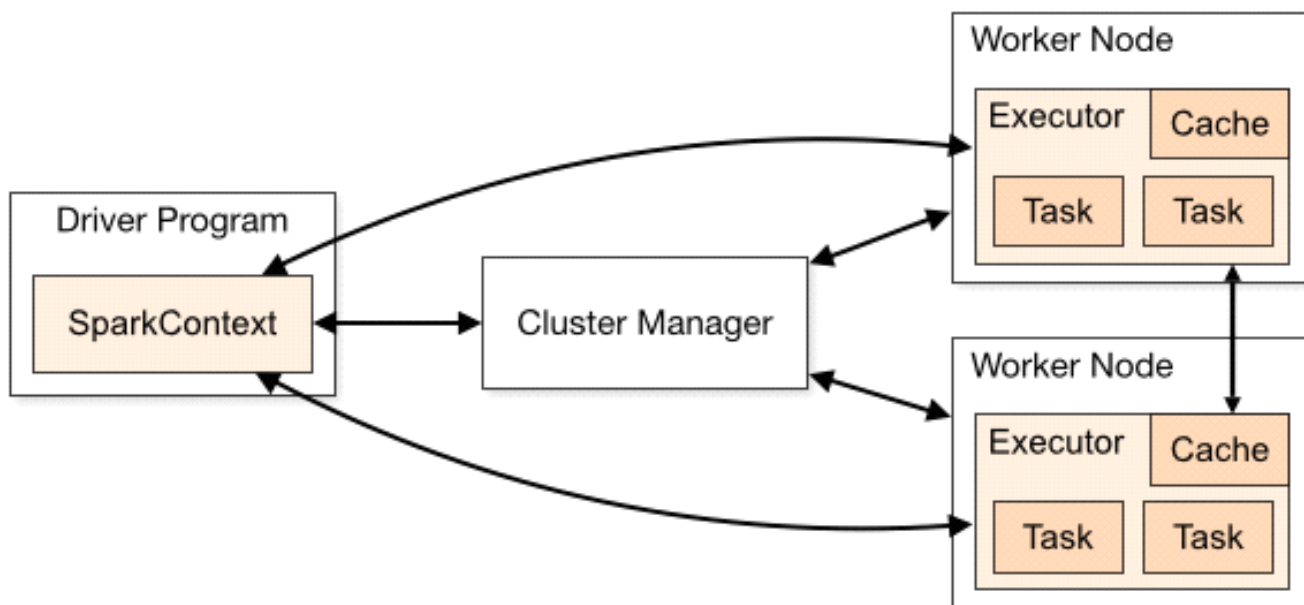
  }
}
```


Spark架构

2018年2月13日 13:16

概述

为了更好地理解调度，我们先来鸟瞰一下集群模式下的Spark程序运行架构图。



1. Driver Program

用户编写的Spark程序称为Driver Program。每个Driver程序包含一个代表集群环境的

SparkContext对象，**程序的执行从Driver程序开始，所有操作执行结束后回到Driver程序**

中，在Driver程序中结束。如果你是用spark shell，那么当你启动 Spark shell的时候，系统

后台自启了一个 Spark 驱动器程序，就是在Spark shell 中预加载的一个叫作 sc 的

SparkContext 对象。如果驱动器程序终止，那么Spark 应用也就结束了。

2. SparkContext对象

每个Driver Program里都有一个SparkContext对象，职责如下：

1) SparkContext对象联系 cluster manager（集群管理器），让 cluster manager 为

Worker Node分配CPU、内存等资源。此外， cluster manager会在 Worker Node 上启动一个执行器（专属于本驱动程序）。

2) 和Executor进程交互，负责**任务**的调度分配。

3. cluster manager 集群管理器

它对应的是Master进程。集群管理器负责集群的**资源调度**，比如为Worker Node分配CPU、内存等资源。并实时监控Worker的资源使用情况。一个Worker Node默认情况下分配一个Executor（进程）。

从图中可以看到sc和Executor之间画了一根线条，这表明：程序运行时，sc是直接和Executor进行交互的。

所以，cluster manager **只是负责资源的管理调度，而任务的分配和结果处理它不管。**

4.Worker Node

Worker节点。集群上的计算节点，对应一台物理机器

5.Worker进程

它对应Worker进程，用于和Master进程交互，向Master注册和汇报自身节点的资源使用情况，并管理和启动Executor进程

6.Executor

负责运行Task计算任务，并将计算结果回传到Driver中。

7.Task

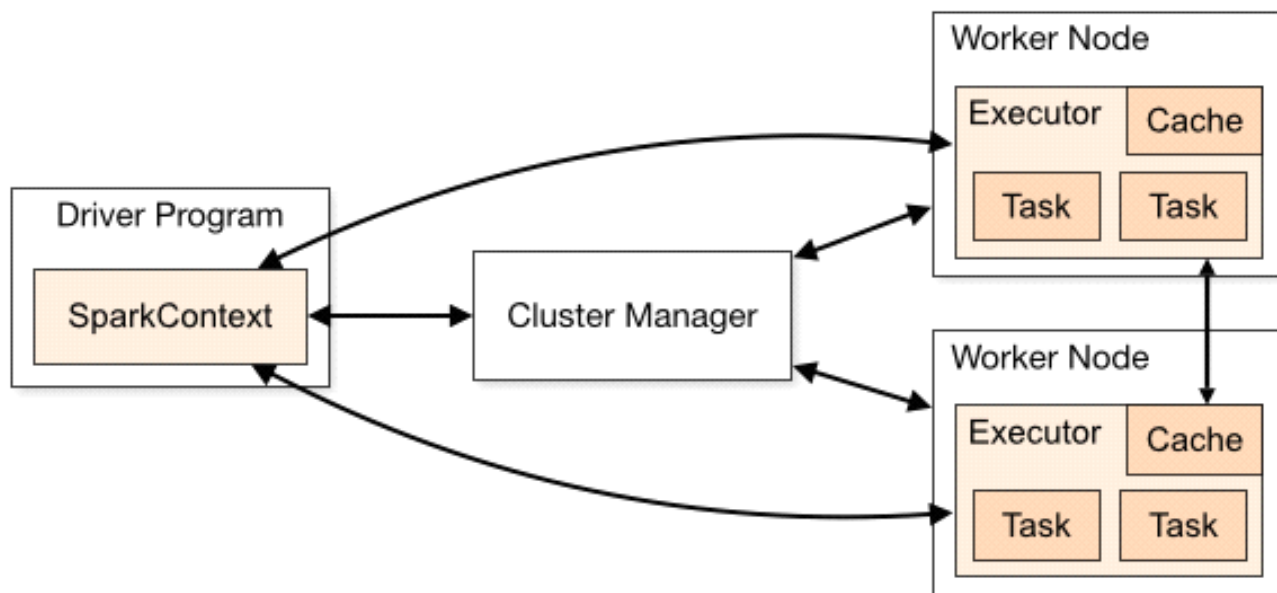
在执行器上执行的最小单元。比如RDD Transformation操作时对RDD内每个分区的计算都会

对应一个Task。

Spark调度模块

2018年3月1日 12:46

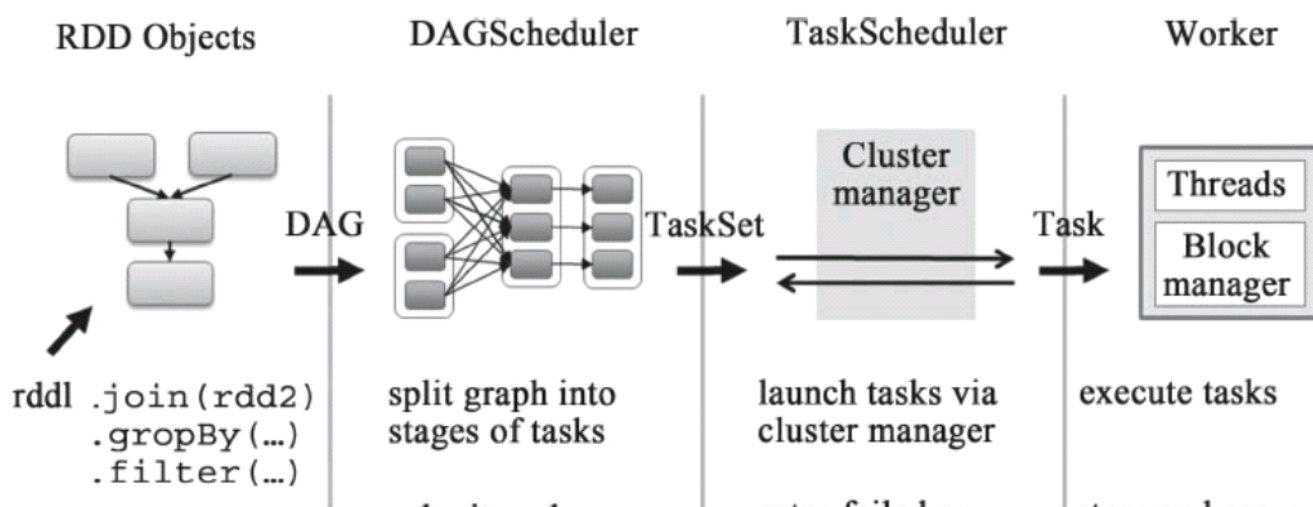
概述

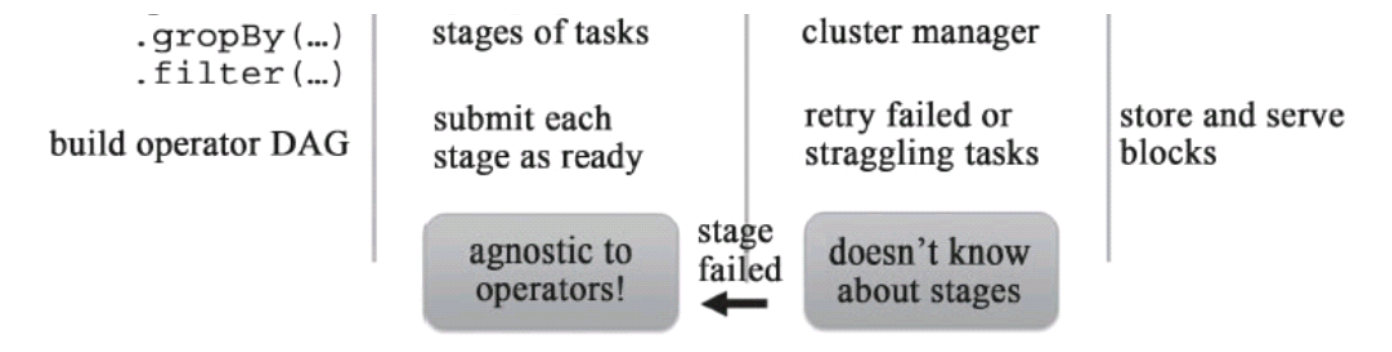


之前我们提到：Driver 的sc负责和Executor交互，完成任务的分配和调度，在底层，任务调度模块主要包含两大部分：

- 1) DAGScheduler
- 2) TaskScheduler

它们负责将用户提交的计算任务按照DAG划分为不同的阶段并且将不同阶段的计算任务提交到集群进行最终的计算。整个过程可以使用下图表示





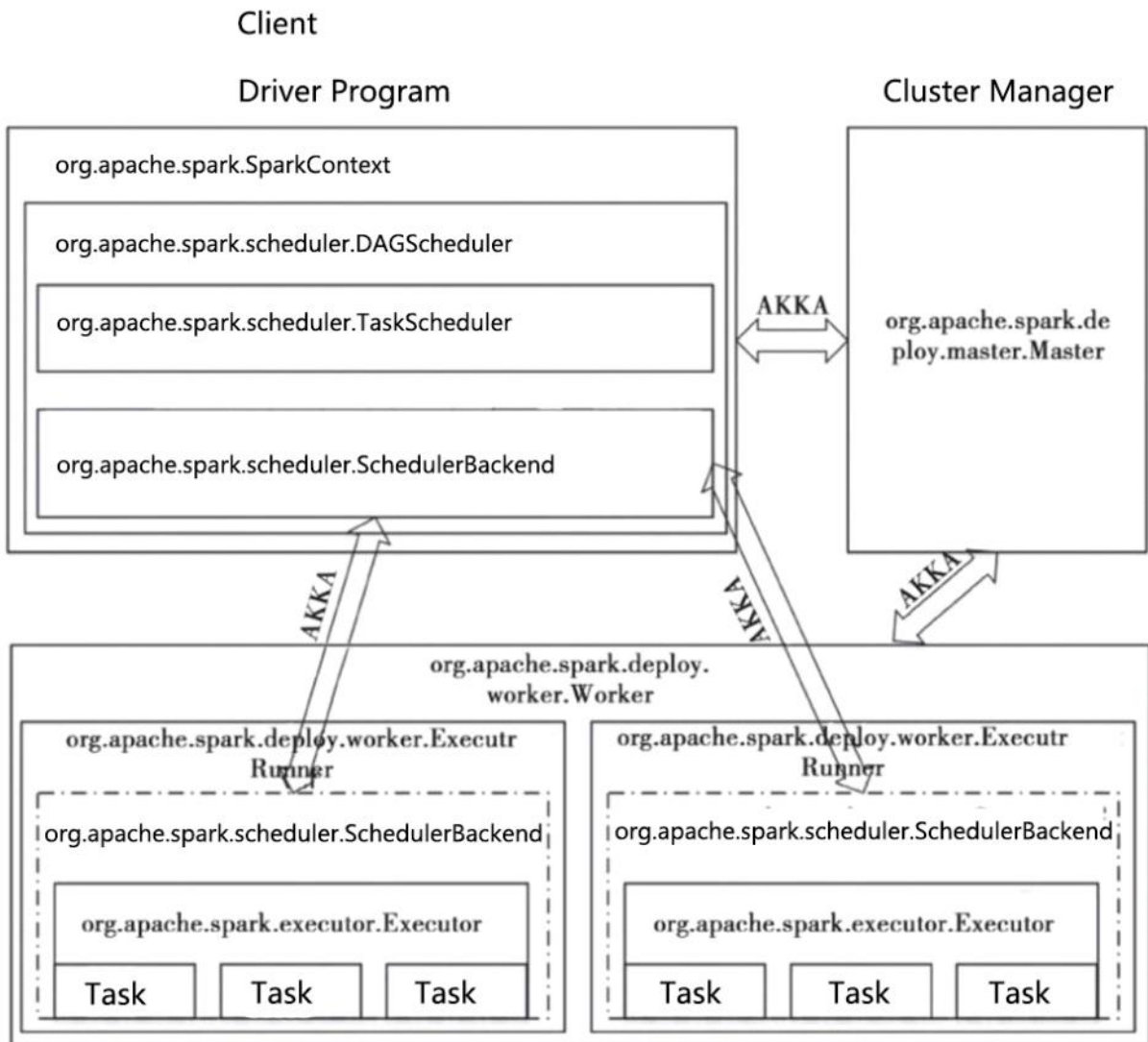
RDD Objects可以理解为用户实际代码中创建的RDD，这些代码逻辑上组成了一个DAG。

DAGScheduler主要负责分析依赖关系，然后将DAG划分为不同的Stage（阶段），其中每个Stage由可以并发执行的一组Task构成，这些Task的执行逻辑完全相同，只是作用于不同的数据。

在DAGScheduler将这组Task划分完成后，会将这组Task提交到

TaskScheduler。TaskScheduler通过Cluster Manager 申请计算资源，比如在集群中的某个Worker Node上启动专属的Executor，并分配CPU、内存等资源。接下来，就是在Executor中运行Task任务，如果缓存中没有计算结果，那么就需要开始计算，同时，计算的结果会回传到Driver或者保存在本地。

Scheduler的实现概述



任务调度模块涉及的最重要的三个类是：

1) `org.apache.spark.scheduler.DAGScheduler` 前面提到的DAGScheduler的实现。

将一个DAG划分为一个一个的Stage阶段（每个Stage是一组Task的集合）

然后把Task Set 交给TaskScheduler模块。

2) `org.apache.spark.scheduler.TaskScheduler`

它的作用是为创建它的SparkContext调度任务，即从DAGScheduler接收不同Stage的任

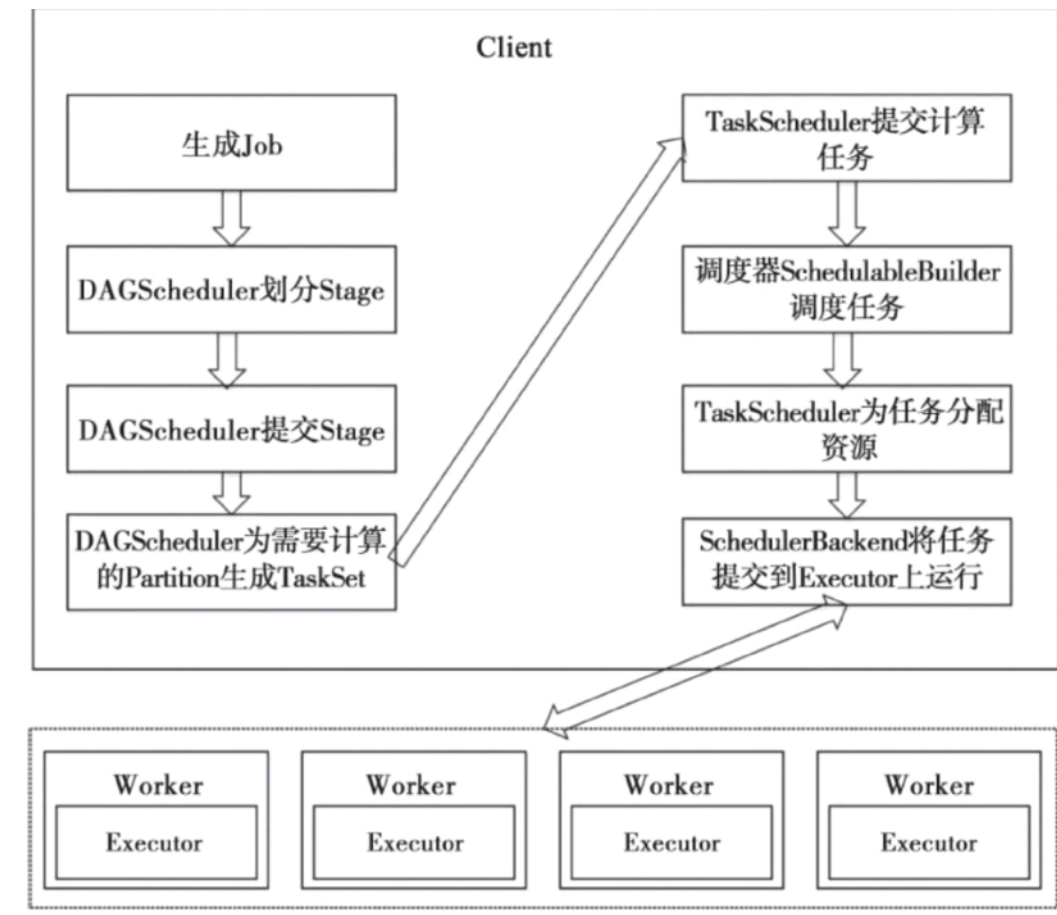
务。向Cluster Manager 申请资源。然后Cluster Manager收到资源请求之后，在Worker为其启动进程

3) `org.apache.spark.scheduler.SchedulerBackend`

是一个trait，作用是分配当前可用的资源，具体就是向当前等待分配计算资源的Task分配计算资源（即Executor），并且在分配的Executor上启动Task，完成计算的调度过程。

4) AKKA是一个网络通信框架，类似于Netty，此框架在Spark1.8之后已全部替换成Netty

任务调度流程图



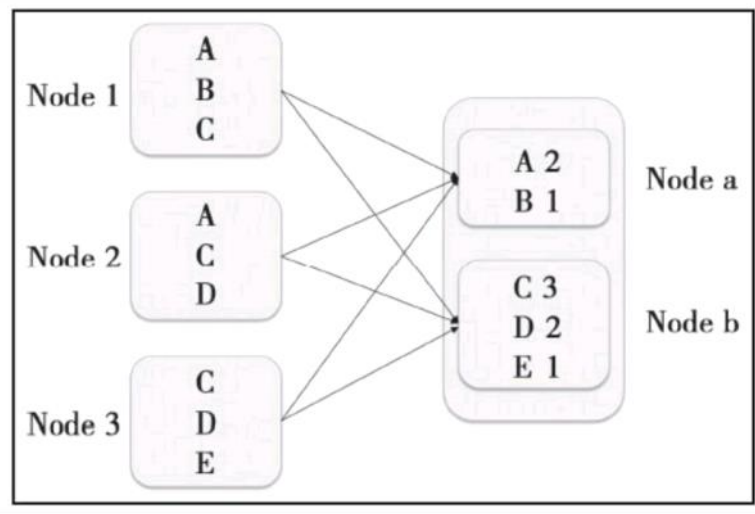
Spark Shuffle详解

2018年3月1日 14:36

概述

Shuffle，翻译成中文就是洗牌。之所以需要Shuffle，还是**因为具有某种共同特征的一类数据需要最终汇聚（aggregate）到一个计算节点上进行计算**。这些数据分布在各个存储节点上并且由不同节点的计算单元处理。以最简单的Word Count为例，其中数据保存在Node1、Node2和Node3；

经过处理后，这些数据最终会汇聚到Nodea、Nodeb处理，如下图所示。



这个数据重新打乱然后汇聚到不同节点的过程就是Shuffle。但是实际上，Shuffle过程可能会非常复杂：

- 1) 数据量会很大，比如单位为TB或PB的数据分散到几百甚至数千、数万台机器上。
- 2) 为了将这个数据汇聚到正确的节点，需要将这些数据放入正确的Partition，因为数据大小已经大于节点的内存，因此这个过程中可能会发生多次硬盘续写。
- 3) 为了节省带宽，这个数据可能需要压缩，如何在压缩率和压缩解压时间中间做一个比较好的选择？
- 4) 数据需要通过网络传输，因此数据的序列化和反序列化也变得相对复杂。

一般来说，每个Task处理的数据可以完全载入内存（如果不能，可以减小每个Partition的大小），因此Task可以做到在内存中计算。**但是对于Shuffle来说，如果不持久化这个中间结果，一旦数据丢失，就需要重新计算依赖的全部RDD**，因此有必要持久化这个中间结果。所以这就是为什么Shuffle过程会产生文件的原因。

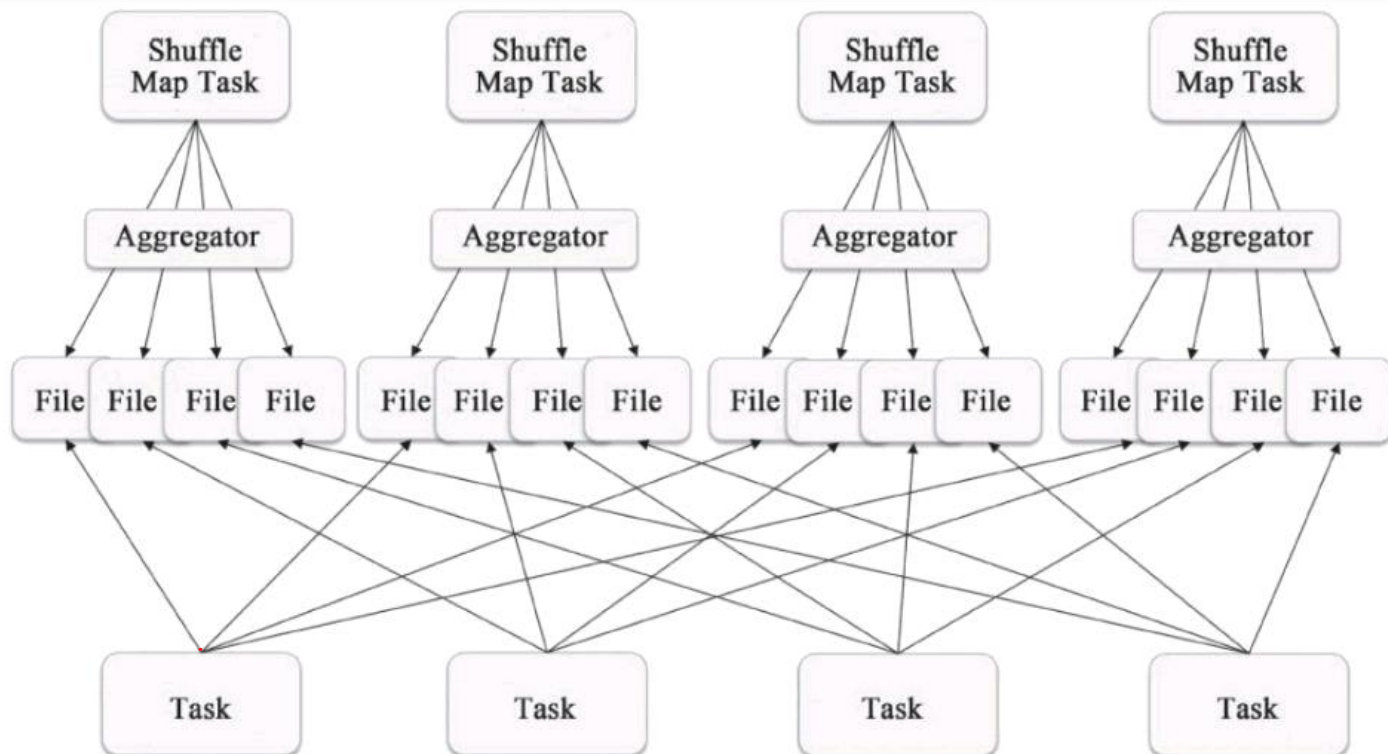
如果Shuffle过程不落地，①可能会造成内存溢出 ②当某分区丢失时，会重新计算所有父分区数据

Shuffle Write

Shuffle Write，即数据是如何持久化到文件中，以使得下游的Task可以获取到其需要处理的数据的（即 Shuffle Read）。在Spark 0.8之前，Shuffle Write是持久化到缓存的，但后来发现实际应用中，shuffle过程带来的数据通常是巨量的，所以经常会发生内存溢出的情况，所以在Spark 0.8以后，Shuffle Write会将数据持久化到硬盘，再之后Shuffle Write不断进行演进优化，但是数据落地到本地文件系统的实现并没有改变。

1) Hash Based Shuffle Write

在Spark 1.0以前，Spark只支持Hash Based Shuffle。因为在很多运算场景中并不需要**排序**，因此多余的排序只能使性能变差，比如Hadoop的Map Reduce就是这么实现的，也就是Reducer拿到的数据都是已经排好序的。实际上Spark的实现很简单：每个Shuffle Map Task根据key的哈希值，计算出每个key需要写入的Partition然后将数据单独写入一个文件，这个Partition实际上就对应了下游的一个Shuffle Map Task或者Result Task。因此下游的Task在计算时会通过网络（如果该Task与上游的Shuffle Map Task运行在同一个节点上，那么此时就是一个本地的硬盘读写）读取这个文件并进行计算。



Hash Based Shuffle Write存在的问题

由于每个Shuffle Map Task需要为每个下游的Task创建一个单独的文件，因此文件的数量就是：

$\text{number}(\text{shuffle_map_task}) * \text{number}(\text{result_task})$ 。

如果Shuffle Map Task是1000，下游的Task是500，那么理论上会产生500000个文件（对于size为0的文件Spark有特殊的处理）。生产环境中Task的数量实际上会更多，因此这个简单的实现会带来以下问题：

1) 每个节点可能会同时打开多个文件，每次打开文件都会占用一定内存。假设每个Write Handler的默认需要100KB的内存，那么同时打开这些文件需要50GB的内存，对于一个集群来说，还是有一定的压力的。尤其是如果Shuffle Map Task和下游的Task同时增大10倍，那么整体的内存就增长到5TB。

2) 从整体的角度来看，打开多个文件对于系统来说意味着随机读，尤其是每个文件比较小但是数量非常多的情况。而现在机械硬盘在随机读方面的性能特别差，非常容易成为性能的瓶颈。如果集群依赖的是固态硬盘，也许情况会改善很多，但是随机写的性能肯定不如顺序写的。

2) Sort Based Shuffle Write

在Spark 1.2.0中，Spark Core的一个重要的升级就是将默认的Hash Based Shuffle换成了Sort Based Shuffle，即spark.shuffle.manager从Hash换成了Sort，对应的实现类分别是

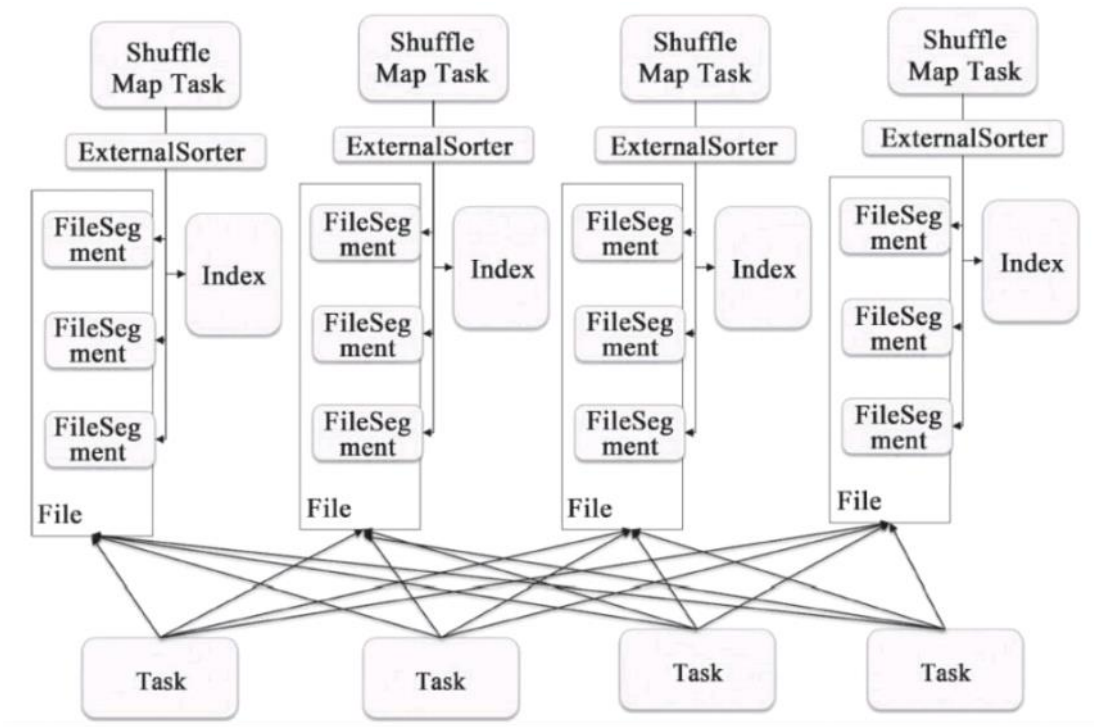
org.apache.spark.shuffle.hash.HashShuffleManager和

org.apache.spark.shuffle.sort.SortShuffleManager。

那么Sort Based Shuffle “取代” Hash Based Shuffle作为默认选项的原因是什么？

正如前面提到的，Hash Based Shuffle的每个Mapper都需要为每个Reducer写一个文件，供Reducer读取，即需要产生M*R个数量的文件，如果Mapper和Reducer的数量比较大，产生的文件数会非常多。

而Sort Based Shuffle的模式是：每个Shuffle Map Task不会为每个Reducer生成一个单独的文件；相反，它会将所有的结果写到一个文件里，同时会生成一个Index文件，



Reducer可以通过这个Index文件取得它需要处理的数据。避免产生大量文件的直接收益就是节省了内存的使用和顺序Disk IO带来的低延时。节省内存的使用可以减少GC的风险和频率。而减少文件的数量可以避免同时写多个文件给系统带来的压力。

Sort Based Write实现详解

Shuffle Map Task会按照key相对应的Partition ID进行Sort，其中属于同一个Partition的key不会Sort。因为对于不需要Sort的操作来说，这个Sort是负收益的；要知道之前Spark刚开始使用Hash Based的Shuffle而不是Sort Based就是为了避免Hadoop Map Reduce对于所有计算都会Sort的性能损耗。对于那些需要Sort的运算，比如sortByKey，这个Sort在Spark 1.2.0里还是由Reducer完成的。

- ①答出shuffle的定义
- ②spark shuffle的特点
- ③spark shuffle的目的
- ④spark shuffle的实现类，即对应优缺点

shuffle 相关参数配置

2018年3月1日 19:18

概述

Shuffle是Spark Core比较复杂的模块，它也是非常影响性能的操作之一。因此，在这里整理了会影响Shuffle性能的各项配置。

1) spark.shuffle.manager

Spark 1.2.0官方版本支持两种方式的Shuffle，即Hash Based Shuffle和Sort Based Shuffle。其中在Spark 1.0之前仅支持Hash Based Shuffle。Spark 1.1引入了Sort Based Shuffle。Spark 1.2的默认Shuffle机制从Hash变成了Sort。如果需要Hash Based Shuffle，只需将spark.shuffle.manager设置成“hash”即可。

配置方式：

- ①进入spark安装目录的conf目录
- ②cp spark-defaults.conf.template spark-defaults.conf
- ③spark.shuffle.ma

应用场景：当产生的临时文件不是很多时，性能可能会比sort shuffle要好。

如果对性能有比较苛刻的要求，那么就要理解这两种不同的Shuffle机制的原理，结合具体的应用场景进行选择。

对于不需要进行排序且Shuffle产生的文件数量不是特别多时，Hash Based Shuffle可能是更好的选择；因为Sort Based Shuffle会按照Reducer的Partition进行排序。

而Sort Based Shuffle的优势就在于可扩展性，它的出现实际上很大程度上是解决

Hash Based Shuffle的可扩展性的问题。由于Sort Based Shuffle还在不断地演进中，因此它的性能会得到不断改善。

对于选择哪种Shuffle，如果性能要求苛刻，最好还是通过实际测试后再做决定。不过选择默认的Sort，可以满足大部分的场景需要。

2) `spark.shuffle.spill`

这个参数的默认值是true，用于指定Shuffle过程中如果内存中的数据超过阈值（参考`spark.shuffle.memoryFraction`的设置）时是否需要将部分数据临时写入外部存储。如果设置为false，那么这个过程就会一直使用内存，会有内存溢出的风险。因此只有在确定内存足够使用时，才可以将这个选项设置为false。

3) `spark.shuffle.memoryFraction`

在启用`spark.shuffle.spill`的情况下，`spark.shuffle.memoryFraction`决定了当Shuffle过程中使用的内存达到总内存多少比例的时候开始spill。在Spark 1.2.0里，这个值是0.2。

此参数可以适当调大，可以控制在0.4~0.6。

通过这个参数可以设置Shuffle过程占用内存的大小，它直接影响了写入到外部存储的频率和垃圾回收的频率。**可以适当调大此值，可以减少磁盘I/O次数。**

4) `spark.shuffle.blockTransferService`

在Spark 1.2.0中这个配置的默认值是**netty**，而在之前的版本中是nio。它主要是用于在各个Executor之间传输Shuffle数据。netty的实现更加简洁，但实际上用户不用太关心这个选项。

除非有特殊需求，否则采用默认配置即可。

5) `spark.shuffle consolidateFiles`

这个配置的默认值是false。主要是为了解决在Hash Based Shuffle过程中产生过多文件的问题。如果配置选项为true，那么对于同一个Core上运行的Shuffle Map Task不会产生一个新的Shuffle文件而是重用原来的。补充：官方给出的建议是：不建议在生产环境下使用，不太稳定。

如果要使用的话：1.把shuffle 管理器变成Hash Shuffle 2.把此参数变为true

6) `spark.shuffle.compress`和`spark.shuffle.spill.compress`

这两个参数的默认配置都是true。`spark.shuffle.compress`和`spark.shuffle.spill.compress`都是用来设置Shuffle过程中是否对Shuffle数据进行压缩。其中，前者针对最终写入本地文件系统的输出文件；后者针对在处理过程需要写入到外部存储的中间数据，即针对最终的shuffle输出文件。

1. 设置`spark.shuffle.compress`

需要评估压缩解压时间带来的时间消耗和因为数据压缩带来的时间节省。如果网络成为瓶颈，比如集群普遍使用的是千兆网络，那么将这个选项设置为true可能更合理；如果计算是CPU密集型的，那么将这个选项设置为false可能更好。

2. 设置`spark.shuffle.spill.compress`

如果设置为true，代表处理的中间结果在spill到本地硬盘时都会进行压缩，在将中间结果取回进行merge的时候，要进行解压。因此要综合考虑CPU由于引入压缩、解压的消耗时间和Disk IO因为压缩带来的节省时间的比较。在Disk IO成为瓶颈的场景下，设置为true可能比较合适；如果本地硬盘是SSD，那么设置为false可能比较合适。

7) `spark.reducer.maxMblnFlight`

这个参数用于限制一个Result Task向其他的Executor请求Shuffle数据时所占用的最大内存数，默认是64MB。尤其是如果网卡是千兆和千兆以下的网卡时。默认值是 设置这个值需要综合考虑网卡带宽和内存。此参数，适当调大，根据经验：64MB~256MB

RDD容错机制

2018年2月13日 13:23

概述

分布式系统通常在一个机器集群上运行，同时运行的几百台机器中某些出问题的概率大大增加，所以容错设计是分布式系统的一个重要能力。

Spark以前的集群容错处理模型，像MapReduce，将计算转换为一个有向无环图（DAG）的任务集合，可以通过重复执行DAG里的一部分任务来完成容错恢复。但是由于主要的数据存储在分布式文件系统中，没有提供其他存储的概念，容错过程需要在网络上进行数据复制，从而增加了大量的消耗。所以，分布式编程中经常需要做检查点，即将某个时机的中间数据写到存储（通常是分布式文件系统）中。

RDD也是一个DAG，每一个RDD都会记住创建该数据集需要哪些操作，跟踪记录RDD的**继承关系**，这个关系在Spark里面叫**lineage（血缘关系）**。当一个RDD的某个分区丢失时，RDD是有足够的信息记录其如何通过其他RDD进行计算，且只需重新计算该分区，这是Spark的一个创新。

RDD的缓存

2018年2月22日 13:11

概述

相比Hadoop MapReduce来说，Spark计算具有巨大的性能优势，其中很大一部分原因是Spark对于内存的充分利用，以及提供的缓存机制。

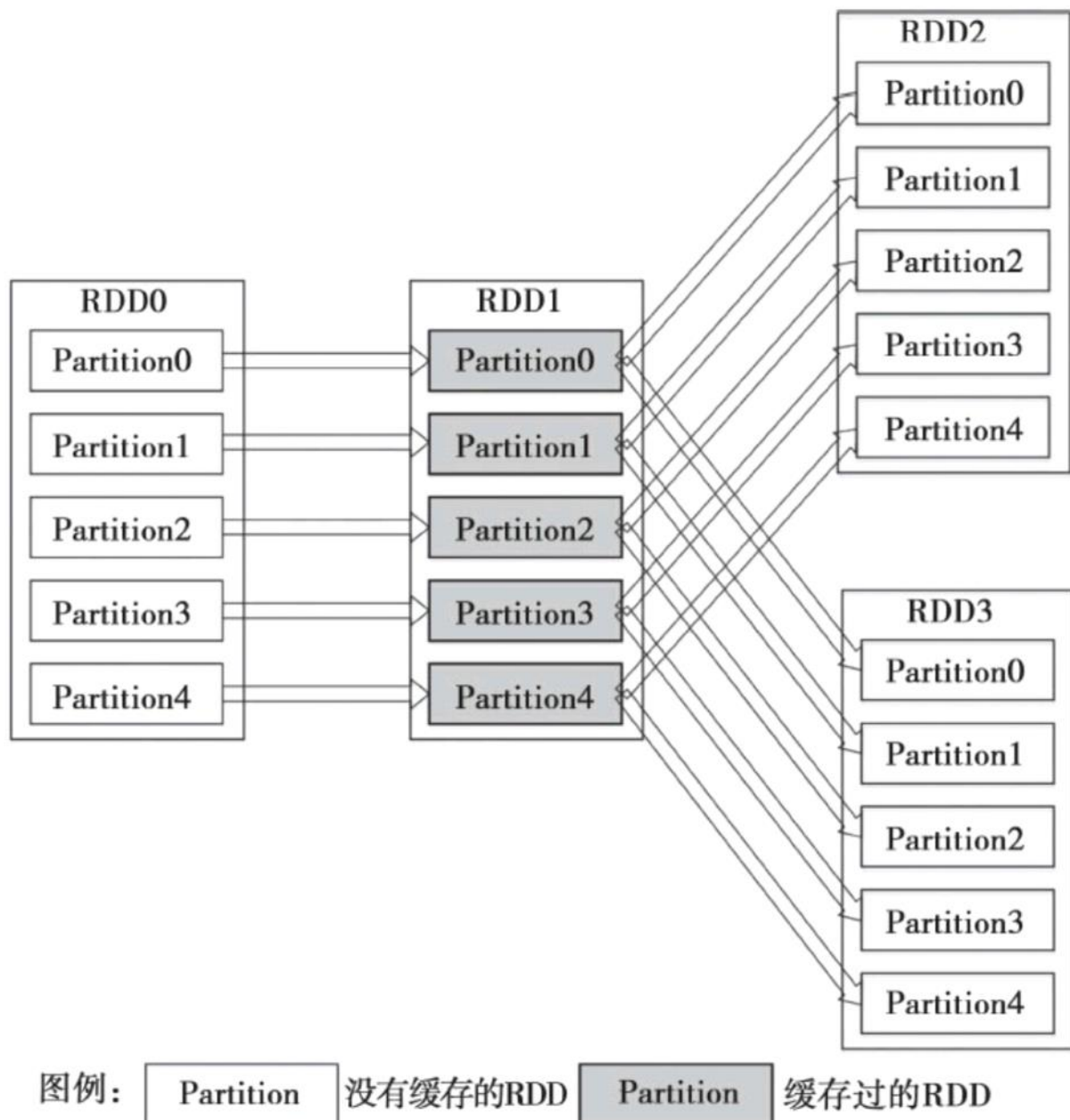
RDD持久化（缓存）

持久化在早期被称作缓存（cache），但缓存一般指将内容放在内存中。虽然持久化操作在绝大部分情况下都是将RDD缓存在内存中，但一般都会在内存不够时用磁盘顶上去（比操作系统默认的磁盘交换性能高很多）。当然，也可以选择不使用内存，而是仅仅保存到磁盘中。所以，现在Spark使用持久化（persistence）这一更广泛的名称。

如果一个RDD不止一次被用到，那么就可以持久化它，这样可以大幅提升程序的性能，甚至达10倍以上。

默认情况下，**RDD只使用一次，用完即扔**，再次使用时需要重新计算得到，而持久化操作**避免了这里的重复计算**，实际测试也显示持久化对性能提升明显，**这也是Spark刚出现时被人称为内存计算框架的原因**。

假设首先进行了RDD0→RDD1→RDD2的计算作业，那么计算结束时，RDD1就已经缓存在系统中了。在进行RDD0→RDD1→RDD3的计算作业时，由于RDD1已经缓存在系统中，因此RDD0→RDD1的转换不会重复进行，计算作业只须进行RDD1→RDD3的计算就可以了，因此计算速度可以得到很大提升。



持久化的方法是调用persist()函数，除了持久化至内存中，还可以在persist()中指定 storage level参数使用其他的类型，具体如下：

1) **MEMORY_ONLY**：将 RDD 以反序列化的 Java 对象的形式存储在 JVM 中. 如果内存空间不够，部分数据分区将不会被缓存，在每次需要用到这些数据时重新进行计算. 这是默认的级别。

cache()方法对应的级别就是MEMORY_ONLY级别

2) **MEMORY_AND_DISK**：将 RDD 以反序列化的 Java 对象的形式存储在 JVM 中。如果

内存空间不够，将未缓存的数据分区存储到磁盘，在需要使用这些分区时从磁盘读取。

3) **MEMORY_ONLY_SER**：将 RDD 以序列化的 Java 对象的形式进行存储（每个分区为一个 byte 数组）。这种方式会比反序列化对象的方式节省很多空间，尤其是在使用 fast serialize 时会节省更多的空间，但是在读取时会使得 CPU 的 read 变得更加密集。如果内存空间不够，部分数据分区将不会被缓存，在每次需要用到这些数据时重新进行计算。

4) **MEMORY_AND_DISK_SER**：类似于 MEMORY_ONLY_SER，但是溢出的分区会存储到磁盘，而不是在用到它们时重新计算。如果内存空间不够，将未缓存的数据分区存储到磁盘，在需要使用这些分区时从磁盘读取。

5) **DISK_ONLY**：只在磁盘上缓存 RDD。

6) **MEMORY_ONLY_2**, **MEMORY_AND_DISK_2**, etc.：与上面的级别功能相同，只不过每个分区在集群中两个节点上建立副本。

7) **OFF_HEAP** 将数据存储到 off-heap memory 中。使用堆外内存，这是 Java 虚拟机里面的概念，堆外内存意味着把内存对象分配在 Java 虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。使用堆外内存的好处：可能会利用到更大的内存存储空间。但是对于数据的垃圾回收会有影响，需要程序员来处理

注意，可能带来一些 GC 回收问题。

Spark 也会自动持久化一些在 shuffle 操作过程中产生的临时数据（比如 reduceByKey），即便是用户并没有调用持久化的方法。这样做可以避免当 shuffle 阶段时如果一个节点挂掉了就得重新计算整个数据的问题。如果用户打算多次重复使用这些数据，我们仍然建议用户自己调用持久化方法对数据进行持久化。

使用缓存

```
scala> import org.apache.spark.storage._
```

```
scala> val rdd1=sc.makeRDD(1 to 5)
```

```
scala> rdd1.cache //cache只有一种默认的缓存级别，即MEMORY_ONLY
```

```
scala> rdd1.persist(StorageLevel.MEMORY_ONLY)
```

缓存数据的清除

Spark 会自动监控每个节点上的缓存数据，然后使用 least-recently-used (LRU) 机制来处理旧的缓存数据。如果你想手动清理这些缓存的 RDD 数据而不是去等待它们被自动清理掉，可以使用 `RDD.unpersist()` 方法。

扩展：GC回收机制及算法

2018年2月19日 19:46

概述

说起垃圾收集（Garbage Collection, GC），大部分人都把这项技术当做Java语言的伴生产物。事实上，GC的历史比Java久远，1960年诞生于MIT的Lisp是第一门真正使用内存动态分配和垃圾收集技术的语言。当Lisp还在胚胎时期时，人们就在思考GC需要完成的3件事情：

- 1) 哪些内存数据需要回收？
- 2) 什么时候回收？
- 3) 如何回收？

经过半个多世纪的发展，目前内存的动态分配与内存回收技术已经相当成熟，一切看起来都进入了“自动化”时代，那为什么我们还要去了解GC和内存分配呢？答案很简单：当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

回到我们熟悉的Java语言。Java内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈3个区域随线程而生，随线程而灭，因此这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。而Java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，**所以垃圾收集器所关注的是java的堆内存。**

哪些内存数据需要被回收？

在堆里面存放着Java世界中几乎所有的对象实例，垃圾收集器在对堆进行回收前，第一件事情就是要确定这些对象之中哪些还“存活”着，哪些已经“死去”（即不可能再被任何途径使用的对象）。

1) 引用计数算法

很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

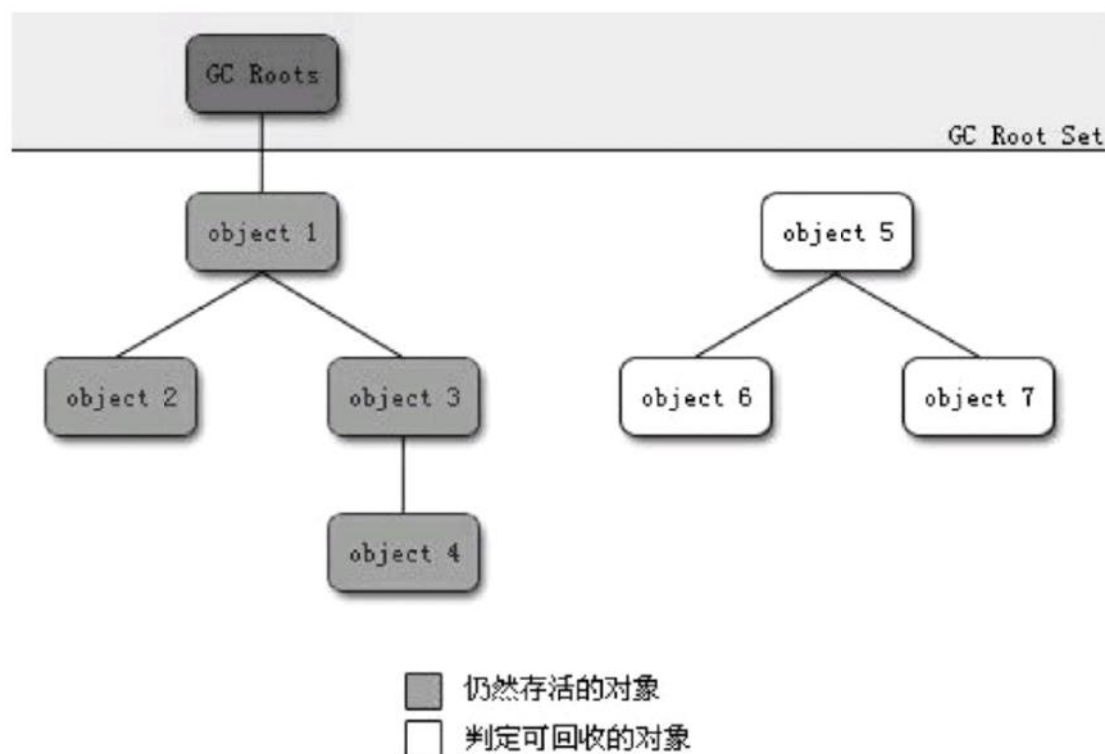
客观地说，引用计数算法（Reference Counting）的实现简单，判定效率也很高，在大部分情况下它都是一个不

错的算法，也有一些比较著名的应用案例，例如微软公司的COM（Component Object Model）技术、使用ActionScript 3的FlashPlayer、Python语言和在游戏脚本领域被广泛应用的Squirrel中都使用了引用计数算法进行内存管理。但是，至少主流的Java虚拟机里面没有选用引用计数算法来管理内存，**其中最主要的原因是它很难解决对象之间相互循环引用的问题。**

2) 可达性分析算法

在主流的商用程序语言（Java、C#，甚至包括前面提到的古老的Lisp）的主流实现中，都是称通过可达性分析（Reachability Analysis）来判定对象是否存活的。**这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。**

如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。



在Java语言中，可作为GC Roots的对象包括下面几种：

- 1) 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 2) 方法区中类静态属性引用的对象。

3) 方法区中常量引用的对象。

垃圾收集算法

1) 标记-清除算法

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其不足进行改进而得到的。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

标记—清除算法的执行过程如下图所示。



比如Linux，关闭Swap分区

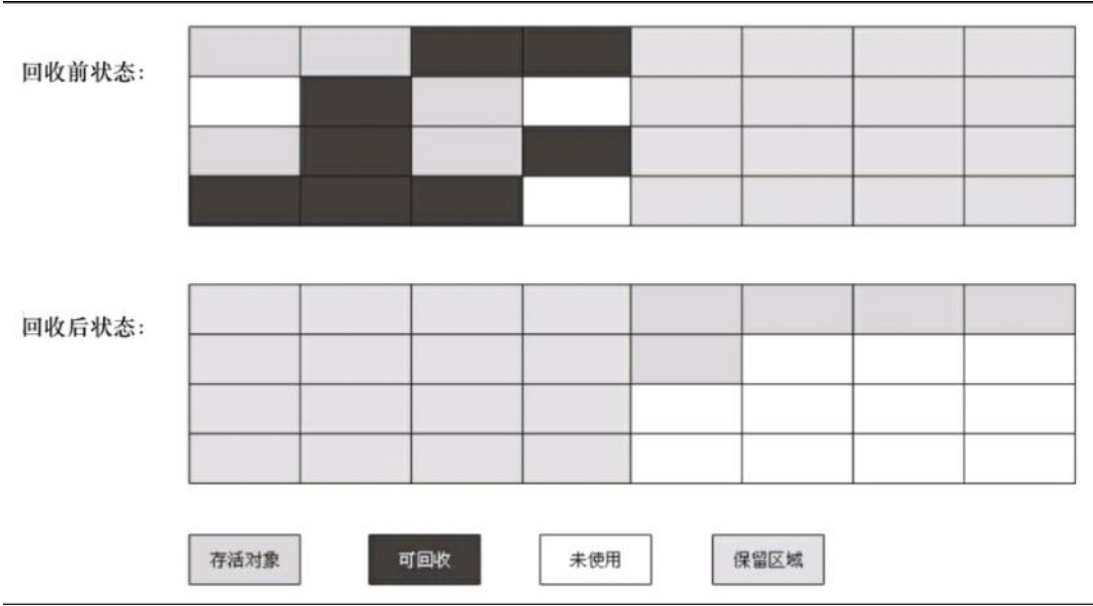
虚拟内存技术——用磁盘空间当做内存，可以将一部分不常用的内存数据交换到磁盘上，待用时，再从磁盘上进行恢复。但一般在使用技术框架时，会避免此情况产生，主要是为了避免磁盘I/O。比如使用Linux时，关闭Swap分区

2) 复制算法(copying)

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的

两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

复制算法的执行过程如下图所示。



现在的商业虚拟机都采用这种收集算法来回收**新生代**，IBM公司的专门研究表明，**新生代中的对象98%是“朝生夕死”的**，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。

当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

内存的分配担保指的是：如果另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。此外，对象进入老年代还有另外一个触发条件，就是一个对象在两个Survivor间进行过多次的移动。

3) 标记-整理算法

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，“标记-整理”算法的示意如下图所示。

所以很多教材说，老年代很少发生GC，因为都是常用的对象，但当老年代的空间满了之后，同样会发生GC回收，称为Major GC，也有的叫Full GC，此时底层一般用的算法就是这种标记-整理算法。

新生代的GC: Minor GC

老生代的GC: Major GC (full GC)



4) 分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

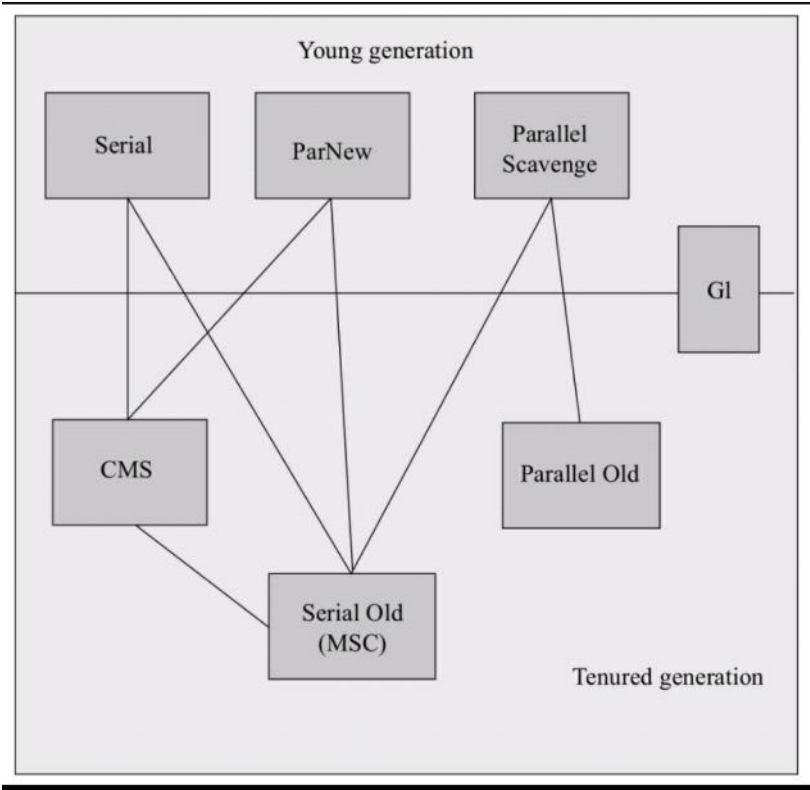
扩展：GC收集器

2018年3月4日 20:10

概述

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。Java虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。

这里讨论的收集器基于JDK 1.7 Update 14之后的HotSpot虚拟机（在这个版本中正式提供了商用的G1收集器，之前G1仍处于实验状态）



上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。虚拟机所处的区域，则表示它是属于新生代收集器还是老年代收集器。

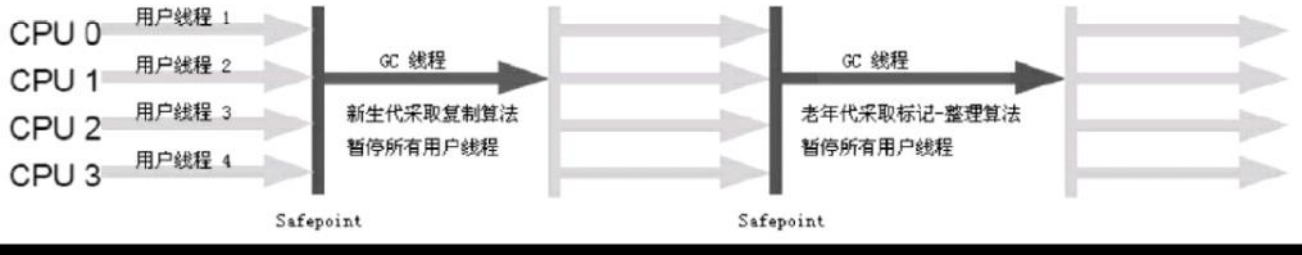
我们先来明确一个观点：虽然我们是在对各个收集器进行比较，但并非为了挑选出一个最好的收集器，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那HotSpot虚拟机就没必要实现那么多不同的收集器了。

Serial收集器

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。

这个收集器是一个**单线程**的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在**它进行垃圾收集时，必须暂停其他所有的工作线程**，直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项

工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说都是难以接受的。不妨试想一下，要是你的计算机每运行一个小时就会暂停响应5分钟，你会有什么样的心情？下图示意了Serial/Serial Old收集器的运行过程。



对于“Stop The World”带给用户的不良体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实地在椅子上或者房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比打扫房间复杂得多。

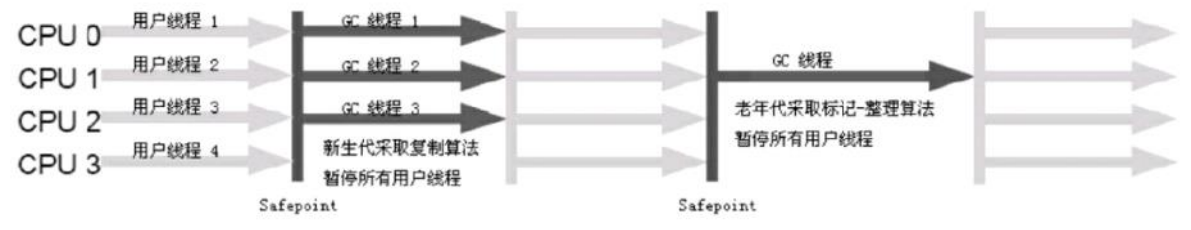
从JDK 1.3开始，一直到现在最新的JDK，HotSpot虚拟机开发团队为消除或者减少工作线程因内存回收而导致停顿的努力一直在进行着，从Serial收集器到Parallel收集器，再到Concurrent Mark Sweep（CMS）乃至GC收集器的最前沿成果Garbage First（G1）收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，**用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除**。寻找更优秀的垃圾收集器的工作仍在继续！

写到这里，似乎已经把Serial收集器描述成一个“老而无用、食之无味弃之可惜”的鸡肋了，但实际上它也有着优于其他收集器的地方：**简单而高效**（与其他收集器的单线程比），Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

比如在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。

ParNew收集器

ParNew收集器其实就是Serial收集器的**多线程**版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew收集器的工作过程如下图所示。



ParNew收集器除了多线程收集之外，其他与Serial收集器相比并没有太多创新之处，但其中有一个与性能无关但很重要的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作。在JDK 1.5时期，HotSpot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器（Concurrent Mark Sweep），这款收集器是HotSpot虚拟机中**第一款真正意义上的并发（Concurrent）收集器**，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作，用前面那个例子的话来说，就是做到了在你的妈妈打扫房间的时候你还能一边往地上扔纸屑。

注意：有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下。

并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

并发（Concurrent）：指用户线程与垃圾收集线程同时工作，可能会交替执行，用户程序不必一直等待。

Parallel Scavenge收集器

Parallel Scavenge收集器是一个**新生代**收集器，它也是使用**复制算法**的收集器，又是并行的多线程收集器，看上去和ParNew都一样，那它有什么特别之处呢？

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而**Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）**。所以，也称之为**吞吐量优先收集器**。

强交互系统底层不适合用Parallel Scavenge，因为要做到低延迟的响应

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，**主要适合在后台运算而不需要太多交互的任务**。

Spark默认用的是Parallel Scavenge收集器。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。

MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。

不过大家不要认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。

GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于吞吐量的倒数。如果把此参数设置为19，那允许的最大GC时间就占总时间的5%（即 $1/(1+19)$ ），默认值为99，就是允许最大1%（即 $1/(1+99)$ ）的垃圾收集时间。

GC的吞吐量定义： $\text{用户执行时间} / (\text{用户执行时间} + \text{GC回收时间})$

用户执行时间：99分钟 GC 1分钟

吞吐量：99%

由于与吞吐量关系密切，**Parallel Scavenge收集器也经常称为“吞吐量优先”收集器**。除上述两个参数之外，Parallel Scavenge收集器还有一个参数-XX:+UseAdaptiveSizePolicy值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，**动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）**。

如果你对于收集器运作原理不太了解，手工优化存在困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio（更关注吞吐量）参数给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。**自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别**。

Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6中才开始提供的。

Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场所，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取**最短回收停顿时间**为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的**响应速度，希望系统停顿时间最短**，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

从名字（包含“Mark Sweep”）上就可以看出，CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：

- 1) 初始标记（CMS initial mark）
- 2) 并发标记（CMS concurrent mark）
- 3) 重新标记（CMS remark）
- 4) 并发清除（CMS concurrent sweep）

其中，1) 初始标记、2) 重新标记这两个步骤仍然需要“Stop The World”。

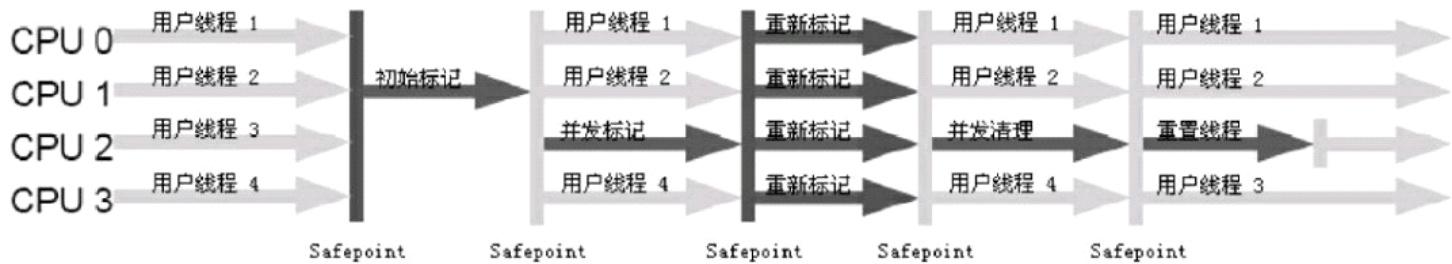
初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。

并发标记阶段就是进行GC RootsTracing的过程。CMS收集器允许在并发标记阶段，用户线

程是可以继续工作，所以这个过程可能会导致GC Roots的路径规则发生变化，所以需要下一个阶段重新标记去进行修正。

重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些。

由于整个过程中耗时最长的**并发标记**和**并发清除**过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。如下图：



CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿，Sun公司的一些官方文档中也称之为**并发低停顿收集器**（Concurrent Low Pause Collector）。

但是CMS还远达不到完美的程度，它有以下3个明显的缺点：

1) CMS收集器对CPU资源非常敏感。其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。CMS默认启动的回收线程数是（CPU数量+3）/4，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大，如果本来CPU负载就比较大，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了50%，其实也让人无法接受。

为了应付这种情况，虚拟机提供了一种称为“增量式并发收集器”（Incremental Concurrent Mark Sweep/i-CMS）的CMS收集器变种，所做的事情和单CPU年代PC机操作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记、清理的时候让GC线程、用户线程交替运行，尽量减少GC线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得少一些，也就是速度下降没有那么明显。实践证明，增量时的CMS收集器效果很一般，在目前版本中，i-CMS已经被声明为“deprecated”，即不再提倡用户使用。

2) 此外，CMS收集器无法处理**浮动垃圾**（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

正是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留**有足够的内存空间给用户线**

程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK 1.5的默认设置下，**CMS收集器当老年代使用了68%的空间后就会被激活**，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在JDK 1.6中，CMS收集器的启动阈值已经**提升至92%**。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置得太高很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

3) 还有最后一个缺点，CMS是一款基于“标记—清除”算法实现的收集器，这就意味着收集结束时可能会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。为了解决这个问题，CMS收集器提供了一个-XX:+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

案例一求中位数

2018年3月1日 23:07

📖 **文件数据 median.txt:**

1 20 8 2 5 11 29 10

7 4 45 6 23 17 19

一共是15个数，正确答案是10

案例—倒排索引

2018年3月2日 20:55

文件代码：



doc1.txt:

hello spark
hello hadoop



doc2.txt:

hello hive
hello hbase
hello spark



doc3.txt:

hadoop hbase
hive scala

最后的结果形式为：

```
(scala, doc3)  
(spark, doc1, doc2)  
(hive, doc2, doc3)  
(hadoop, doc1, doc3)  
(hello, doc1, doc2)  
(hbase, doc2, doc3)
```

课后：案例—求中位数

2018年3月1日 23:07

📖 文件数据 median.txt:

1 20 8 2 5 11 29 10

7 4 45 6 23 17 19

一共是15个数，正确答案是10

📖 代码示例:

```
object Driver {  
  
  def main(args: Array[String]): Unit = {  
    val conf=new SparkConf().setMaster("local").setAppName("median")  
    val sc=new SparkContext(conf)  
    val data=sc.textFile("d://data/median.txt")  
  
    val count=data.flatMap{_.split(" ")}.count()  
    val medianIndex=(count+1)/2  
  
    val median=data.flatMap { _.split(" ") }  
      .map{x=>x.toInt}.sortBy(x=>x).take(medianIndex.toInt).last  
  
    println(median)  
  }  
}
```

📖 代码示例:

```
import org.apache.spark.SparkConf  
import org.apache.spark.SparkContext  
  
object MedianDriver {  
  
  def main(args: Array[String]): Unit = {  
    val conf=new SparkConf().setMaster("local").setAppName("Median")  
    val sc=new SparkContext(conf)  
    val data=sc.textFile("e://median.txt",2)  
  
    val nums=data.flatMap { x => x.split(" ") }  
  
    val sortResult=nums.map { x => (x.toInt,1) }.sortByKey()
```

```
val totalNum=sortResult.mapPartitions{x=>

    val list=List[Int]()
    list.::(x.size).iterator
}.sum().toInt

val medianIndex=(totalNum+1)/(2)

val medianResult=sortResult.takeOrdered(medianIndex).last._1
//val medianResult=sortResult.top(medianIndex)(Ordering.by [(Int, Int), Int](-_._1)).last._1

println(medianResult)

}
}
```

课后：案例—倒排索引

2018年3月2日 20:55

文件代码：



doc1.txt:

hello spark
hello hadoop



doc2.txt:

hello hive
hello hbase
hello spark



doc3.txt:

hadoop hbase
hive scala

最后的结果形式为：

```
(scala, doc3)
(spark, doc1, doc2)
(hive, doc2, doc3)
(hadoop, doc1, doc3)
(hello, doc1, doc2)
(hbase, doc2, doc3)
```



代码：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object Driver {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("inverted")
    val sc=new SparkContext(conf)

    //--读取指定目录下所有的文件，并返回到一个RDD中
    //--(filepath,filetext)
    val data=sc.wholeTextFiles("d://data/inverted/*")
```

```

val clearData=data.map{case(filepath,filetext)=>
  val filename=filepath.split("/").last.dropRight(4)
  (filename,filetext)
}

//--(hello,doc1) (hello,doc2) (hadoop,doc1).....
//--(hello,List(doc1,doc2))

val resultData=clearData.flatMap{case(filename,filetext)=>
  filetext.split("\r\n").flatMap { x => x.split(" ") }.map { x => (x,filename) }
}

val result=resultData.groupByKey.map{case(word,buffer)=>
(word,buffer.toList.distinct.mkString(","))}

result.foreach(println)
}
}

```

扩展：GC收集器

2018年3月4日 20:10

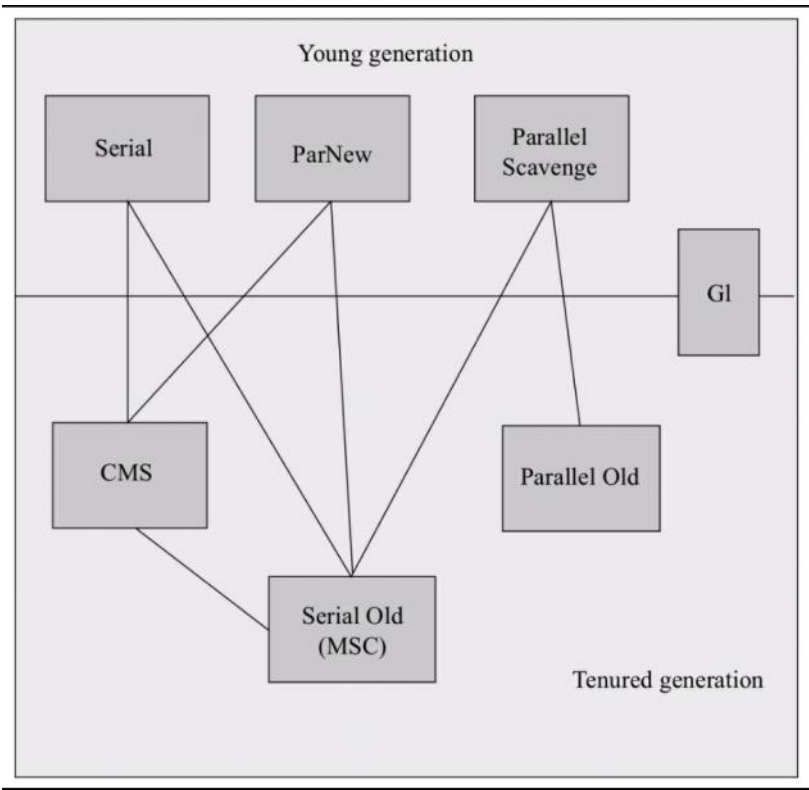
关于GC推荐的书籍

《深入理解Java虚拟机：JVM高级特性与最佳实践》，看第二张

概述

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。Java虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。

这里讨论的收集器基于JDK 1.7 Update 14之后的HotSpot虚拟机（在这个版本中正式提供了商用的G1收集器，之前G1仍处于实验状态）



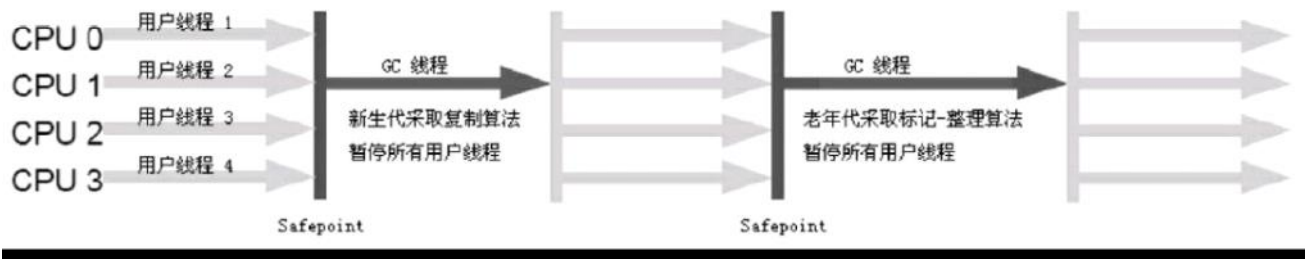
上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。虚拟机所处的区域，则表示它是属于新生代收集器还是老年代收集器。

我们先来明确一个观点：虽然我们是在对各个收集器进行比较，但并非为了挑选出一个最好的收集器，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那HotSpot虚拟机就没必要实现那么多不同的收集器了。

Serial收集器

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。

这个收集器是一个**单线程**的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在**它进行垃圾收集时，必须暂停其他所有的工作线程**，直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说都是难以接受的。不妨试想一下，要是你的计算机每运行一个小时就会暂停响应5分钟，你会有什么样的心情？下图示意了Serial/Serial Old收集器的运行过程。



对于“Stop The World”带给用户的不良体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实地在椅子上或者房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比打扫房间复杂得多。

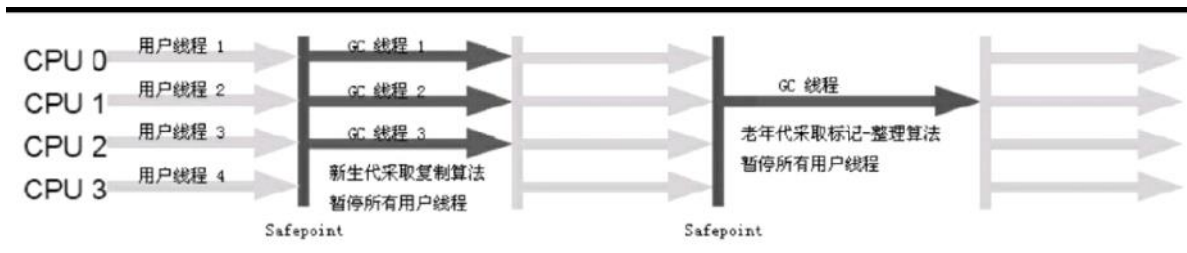
从JDK 1.3开始，一直到现在最新的JDK，HotSpot虚拟机开发团队为消除或者减少工作线程因内存回收而导致停顿的努力一直在进行着，从Serial收集器到Parallel收集器，再到Concurrent Mark Sweep（CMS）乃至GC收集器的最前沿成果Garbage First（G1）收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，**用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除**。寻找更优秀的垃圾收集器的工作仍在继续！

写到这里，似乎已经把Serial收集器描述成一个“老而无用、食之无味弃之可惜”的鸡肋了，但实际上它也有着优于其他收集器的地方：**简单而高效**（与其他收集器的单线程比），Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

比如在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。

ParNew收集器

ParNew收集器其实就是Serial收集器的**多线程**版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew收集器的工作过程如下图所示。



ParNew收集器除了多线程收集之外，其他与Serial收集器相比并没有太多创新之处，但其中有一个与性能无关但很重要的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作。在JDK 1.5时期，HotSpot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器（Concurrent Mark Sweep），这款收集器是HotSpot虚拟机中第一款真正意义上的并发（Concurrent）收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作，用前面那个例子的话来说，就是做到了在你的妈妈打扫房间的时候你还能一边往地上扔纸屑。

注意：有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下。

并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

并发（Concurrent）：指用户线程与垃圾收集线程同时工作，可能会交替执行，用户程序不必一直等待。

Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器，看上去和ParNew都一样，那它有什么特别之处呢？

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。所以，也称之为吞吐量优先收集器。

强交互系统底层不适合用Parallel Scavenge，因为要做到低延迟的响应

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Spark默认用的是Parallel Scavenge收集器。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。

MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。

不过大家不要认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下

降，但吞吐量也降下来了。

GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把此参数设置为19，那允许的最大GC时间就占总时间的5%（即 $1/(1+19)$ ），默认值为99，就是允许最大1%（即 $1/(1+99)$ ）的垃圾收集时间。

gc的吞吐量定义： $\text{用户执行时间} / (\text{用户执行时间} + \text{GC回收时间})$

用户执行时间：99分钟 GC 1分钟

吞吐量：99%

由于与吞吐量关系密切，**Parallel Scavenge收集器也经常称为“吞吐量优先”收集器**。除上述两个参数之外，Parallel Scavenge收集器还有一个参数-XX:+UseAdaptiveSizePolicy值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，**动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。**

如果你对于收集器运作原理不太了解，手工优化存在困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio（更关注吞吐量）参数给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。**自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。**

Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6中才开始提供的。

Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取**最短回收停顿时间**为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的**响应速度，希望系统停顿时间最短**，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

从名字（包含“Mark Sweep”）上就可以看出，CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：

- 1) 初始标记（CMS initial mark）
- 2) 并发标记（CMS concurrent mark）
- 3) 重新标记（CMS remark）

4) 并发清除 (CMS concurrent sweep)

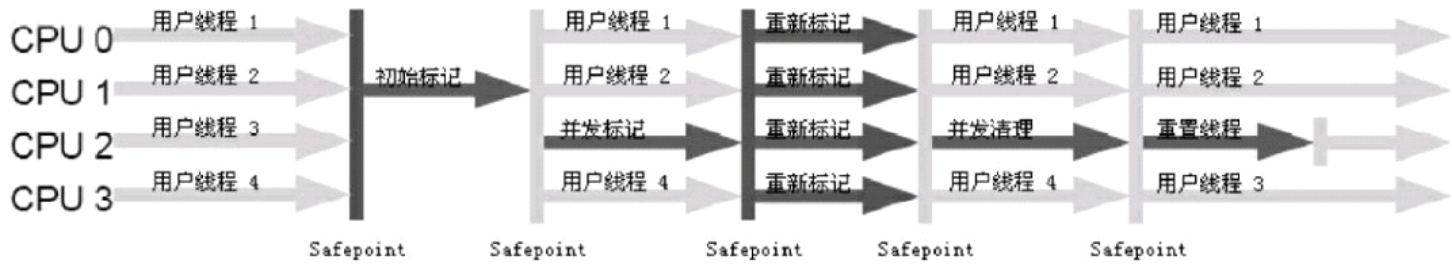
其中，1) 初始标记、2) 重新标记这两个步骤仍然需要 “Stop The World” 。

初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。

并发标记阶段就是进行GC RootsTracing的过程。CMS收集器允许在并发标记阶段，用户线程是可以继续工作，所以这个过程可能会导致GC Roots的路径规则发生变化，所以需要下一个阶段重新标记去进行修正。

重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些。

由于整个过程中耗时最长的**并发标记**和**并发清除**过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。如下图：



CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿，Sun公司的一些官方文档中也称之为**并发低停顿收集器** (Concurrent Low Pause Collector) 。

但是CMS还远达不到完美的程度，它有以下3个明显的缺点：

1) CMS收集器对CPU资源非常敏感。其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。CMS默认启动的回收线程数是（CPU数量+3）/4，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大，如果本来CPU负载就比较大，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了50%，其实也让人无法接受。

为了应付这种情况，虚拟机提供了一种称为“增量式并发收集器” (Incremental Concurrent Mark Sweep/i-CMS) 的CMS收集器变种，所做的事情和单CPU年代PC机操作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记、清理的时候让GC线程、用户线程交替运行，尽量减少GC线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得少一些，也就是速度下降没有那么明显。实践证明，增量时的CMS收集器效果很一般，在目前版本中，i-CMS已经被声明为“deprecated”，即不再提倡用户使用。

2) 此外，CMS收集器无法处理**浮动垃圾**（Floating Garbage），可能出现

“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

正是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留**有足够的内存空间给用户线程使用**，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK 1.5的默认设置下，**CMS收集器当老年代使用了68%的空间后就会被激活**，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在JDK 1.6中，CMS收集器的启动阈值已经**提升至92%**。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次

“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置得太高很容易导致大量

“Concurrent Mode Failure”失败，性能反而降低。

3) 还有最后一个缺点，CMS是一款基于“标记—清除”算法实现的收集器，这就意味着收集结束时可能会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。为了解决这个问题，CMS收集器提供了一个-XX:

+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

Spark调优—上篇

2018年2月13日 9:29

更好的序列化实现

Spark用到序列化的地方

- 1) Shuffle时需要将对象写入到外部的临时文件。
- 2) 每个Partition中的数据要发送到worker上，spark先把RDD包装成task对象，将task通过网络发给worker。
- 3) RDD如果支持内存+硬盘，只要往硬盘中写数据也会涉及序列化。

默认使用的是java的序列化。但java的序列化有两个问题，一个是性能相对较低，另外它序列化完二进制的内容长度也比较大，造成网络传输时间拉长。业界现在有很多更好的实现，如**kryo**，比java的序列化快10倍以上。而且生成内容长度也短。时间快，空间小，自然选择它了。

方法一：修改spark-defaults.conf配置文件

设置：

```
spark.serializer org.apache.spark.serializer.KryoSerializer
```

注意：用空格隔开

方法二：启动spark-shell或者spark-submit时配置

```
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer
```

方法三：在代码中

```
val conf = new SparkConf()
conf.set( "spark.serializer" , "org.apache.spark.serializer.KryoSerializer" )
```

三种方式实现效果相同，优先级越来越高。

通过代码使用Kryo

📖 文件数据:

rose 23

tom 25

📖 Person类代码:

```
class Person(var1:String,var2:Int) extends Serializable{

    var name=var1
    var age=var2

}
```

📖 MyKryoRegister类代码:

```
import org.apache.spark.serializer.KryoRegistrator
import com.esotericsoftware.kryo.Kryo

class MyKryoRegister extends KryoRegistrator {
    def registerClasses(kryo: Kryo): Unit = {
        kryo.register(classOf[Person])
    }
}
```

📖 KryoDriver代码:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.storage.StorageLevel

object KryoDriver {

    def main(args: Array[String]): Unit = {
        val conf = new SparkConf().setMaster("local").setAppName("kryoTest")
        .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
        .set("spark.kryo.registrator", "cn.tedu.MyKryoRegister")
        val sc = new SparkContext(conf)
```

```
val data=sc.textFile("d://people.txt")

val userData=data.map { x => new Person(x.split(" ")(0),x.split(" ")(1).toInt) }

userData.persist(StorageLevel.MEMORY_AND_DISK)

userData.foreach(x=>println(x.name))
}
}
```

Spark调优—中篇

2018年2月19日 15:54

配置多临时文件目录

spark.local.dir参数。当shuffle、归并排序（sort、merge）时都会产生临时文件。这些临时文件都在这个指定的目录下。那这个文件夹有很多临时文件，如果都发生读写操作，有的线程在读这个文件，有的线程在往这个文件里写，磁盘I/O性能就非常低。

怎么解决呢？**可以创建多个文件夹，每个文件夹都对应一个真实的硬盘。**假如原来是3个程序同时读写一个硬盘，效率肯定低，现在让三个程序分别读取3个磁盘，这样冲突减少，效率就提高了。这样就有效提高外部文件读和写的效率。怎么配置呢？只需要在这个配置时配置多个路径就可以。中间用逗号分隔。

```
spark.local.dir=/home/tmp,/home/tmp2
```

启用推测执行机制

可以设置spark.speculation true

开启后，spark会检测执行较慢的Task，并复制这个Task在其他节点运行，最后哪个节点先运行完，就用其结果，然后将慢Task 杀死

collect速度慢

我们在讲的时候一直强调，collect只适合在测试时，因为把结果都收集到Driver服务器上，数据要跨网络传输，同时要求Driver服务器内存大，所以收集过程慢。解决办法就是直接输出到分布式文件系统中。

有些情况下，RDD操作使用MapPartitions替代map

map方法对RDD的每一条记录逐一操作。mapPartitions是对RDD里的每个分区操作

```
rdd.map{ x=>conn=getDBConn.conn;write(x.toString);conn close;}
```

这样频繁的连接、断开数据库，效率差。

```
rdd.mapPartitions{(record:=>conn.getDBConn;for(item<-recorders;  
write(item.toString);conn close;}
```

这样就一次链接一次断开，中间批量操作，效率提升。

扩展：Spark调优—下篇

2018年2月19日 15:49

Spark的GC调优

由于Spark立足于内存计算，常常需要在内存中存放大量数据，因此也更依赖JVM的垃圾回收机制（GC）。并且同时，它支持兼容**批处理**和**流式处理**，对于程序吞吐量和延迟都有较高要求，因此GC参数的调优在Spark应用实践中显得尤为重要。

在运行Spark应用时，有些问题是由于GC所带来的，例如垃圾回收时间久、程序长时间无响应，甚至造成程序崩溃或者作业失败。

按照经验来说，当我们配置垃圾收集器时，主要有两种策略——Parallel GC（吞吐量优先）和CMS GC（低延迟响应）。

前者注重更高的吞吐量，而后者则注重更低的延迟。两者似乎是鱼和熊掌，不能兼得。在实际应用中，我们只能根据应用对性能瓶颈的侧重性，来选取合适的垃圾收集器。例如，当我们运行需要有实时响应的场景的应用时，我们一般选用CMS GC，而运行一些离线分析程序时，则选用Parallel GC。

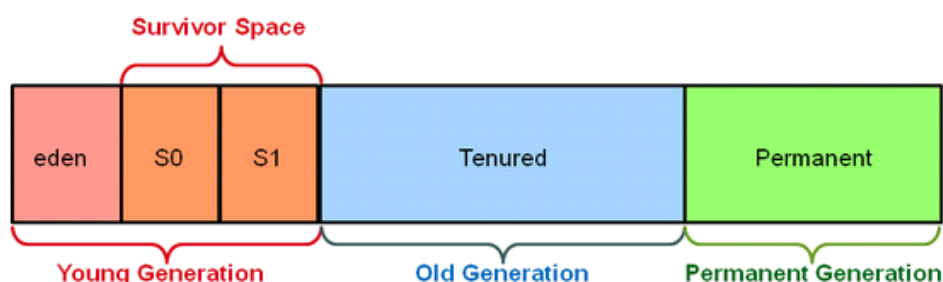
那么对于Spark这种既支持流式计算，又支持传统的批处理运算的计算框架来说，是否存在一组通用的配置选项呢？

通常CMS GC是企业比较常用的GC配置方案，并在长期实践中取得了比较好的效果。例如对于进程中若存在大量寿命较长的对象，Parallel GC经常带来较大的性能下降。因此，即使是批处理的程序也能从CMS GC中获益。不过，在从1.7开始的HOTSPOT JVM中，我们发现了一个新的GC设置项：**Garbage-First GC(G1 GC)**。Oracle将其定位为CMS GC的长期演进，这让我们重燃了鱼与熊掌兼得的希望！

GC算法原理

在传统JVM内存管理中，我们把Heap空间分为Young/Old两个分区，Young分区又包括一个Eden和两个Survivor分区，如下图所示。新产生的对象首先会被存放在Eden区，而每次minor GC发生时，JVM一方面将Eden分区内存活的对象拷贝到一个空的Survivor分区，另一方面将另一个正在被使用的Survivor分区中的存活对象也拷贝到空的Survivor分区内。

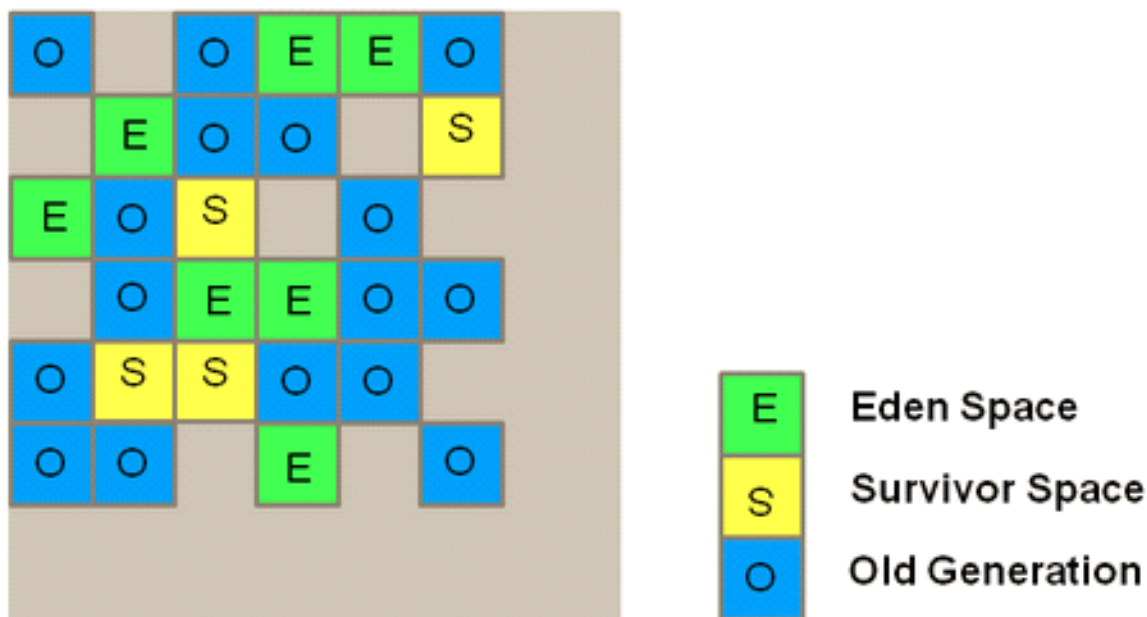
在此过程中，JVM始终保持一个Survivor分区处于全空的状态。一个对象在两个Survivor之间的拷贝到一定次数后，如果还是存活的，就将其拷入Old分区。当Old分区没有足够空间时，GC会停下所有程序线程，进行Full GC，即对Old区中的对象进行整理。注意：Full GC时，所有线程都暂停，所以这个阶段被称为Stop-The-World(STW)，也是大多数GC算法中对性能影响最大的部分。



而G1 GC则完全改变了这一传统思路。它将整个Heap分为若干个预先设定的小区域块，每个区域块内部不再进行新旧分区，而是将整个区域块标记为Eden/Survivor/Old。当创建新对象时，它首先被存放到某一个可用区块（Region）中。当该区块满了，JVM就会创建新的区块存放对象。当发生minor GC时，JVM将一个或几个区块中存活的对象拷贝到一个新的区块中，并在空余的空间中选择几个全新区块作为新的Eden分区。当所有区域中都有存活对象，找不到全空区块时，才发生Full GC。即G1 GC发生Full GC的频次要比其他GC更低，因为内存使用率很高。

而在标记存活对象时，G1使用RememberSet的概念，将每个分区外指向分区内的引用记录在该分区的RememberSet中，避免了对整个Heap的扫描，使得各个分区的GC更加独立。

在这样的背景下，我们可以看出G1 GC大大提高了触发Full GC时的Heap占用率，同时也使得Minor GC的暂停时间更加可控，对于内存较大的环境非常友好。因为G1 GC对于内存的使用率特别高，内存越大，此优势越明显。



关于Hotspot JVM所支持的完整的GC参数列表，可以参见Oracle官方的文档中对部分参数的解释。

Spark的内存管理

Spark的核心概念是RDD，实际运行中内存消耗都与RDD密切相关。Spark允许用户将应用中重复使用的RDD数据持久化缓存起来，从而避免反复计算的开销，而RDD的持久化形态之一就是将全部或者部分数据缓存在JVM的Heap中。当我们观察到GC延迟影响效率时，应当先检查Spark应用本身是否有效利用有限的内存空间。RDD占用的内存空间比较少的话，程序运行的heap空间也会比较宽松，GC效率也会相应提高；而RDD如果占用大量空间的话，则会带来巨大的性能损失。

下面从某个用户案例来说明：

这个应用其本质就是一个简单的迭代计算。而每次迭代计算依赖于上一次的迭代结果，因此每次迭代结果都会被主动持久化到内存空间中。当运行用户程序时，我们观察到随着迭代次数的增加，进程占用的内存空间不断快速增长，GC问题越来越突出。

造成这个问题的原因是没有及时释放掉不再使用的RDD，从而造成了内存空间不断增长，触发了更多GC执行。

迭代轮数	单次迭代缓存大小	总缓存大小(优化前)	总缓存大小(优化后)
初始化	4.3GB	4.3GB	4.3GB
1	8.2GB	12.5GB	8.2GB
2	98.8GB	111.3GB	98.8GB
3	90.8GB	202.1GB	90.8GB

小结：当观察到GC频繁或者延时长久的情况，也可能是Spark进程或者应用中内存空间没有有效利用。所以可以尝试检查是否存在RDD持久化后未得到及时释放等情况。

选择垃圾收集器

在解决了应用本身的问题之后，我们就要开始针对Spark应用的GC调优了。

Spark默认使用的是Parallel GC。经调研我们发现，Parallel GC常常受困于Full GC，而每次Full GC都给性能带来了较大的下降。而Parallel GC可以进行参数调优的空间也非常有限，我们只能通过调节一些基本参数来提高性能，如各年代分区大小比例、进入老年代前的拷贝次数等。而且这些调优策略只能推迟Full GC的到来，如果是长期运行的应用，Parallel GC调优的意义就非常有限了。

Configurati on Options	-XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+PrintFlagsFinal -XX: +PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX: +PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -Xms88g -Xmx88g
-----------------------------------	---

G1 GC的配置

Configurati on Options	-XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc - XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX: +PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX: +G1SummarizeConcMark -Xms88g -Xmx88g
-----------------------------------	---

大多数情况下，最大的性能下降是由Full GC导致的，G1 GC也不例外，所以当使用G1 GC时，需要根据一定的实际情况进行参数配置，这需要很丰富的工作经验和运维经验，以下仅提供一些处理思路。

比如G1 GC收集器在将某个需要垃圾回收的分区进行回收时，无法找到一个能将其中存活对象拷贝过去的空闲分区。这种情况被称为Evacuation Failure，常常会引发Full GC。对于这种情况，我们常见的处理办法有两种：

将InitiatingHeapOccupancyPercent参数调低（默认值是45），可以使G1 GC收集器更早开始Mixed GC（Minor GC）；但另一方面，会增加GC发生频率。（启动并发GC周期时的堆内存占用百分比. G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不是某一代内存的使用比. 值为 0 则表示"一直执行GC循环". 默认值为 45.）

降低此值，会提高Minor GC的频率，但是会推迟Full GC的到来。

提高ConcGCThreads的值，在Mixed GC阶段投入更多的并发线程，争取提高每次暂停的效率。但是此参数会占用一定的有效工作线程资源。

调试这两个参数可以有效降低Full GC出现的概率。Full GC被消除之后，最终的性能获得了大幅提升。

此外，可能还会遇到这样的情况：出现了一些比G1的一个分区的一半更大的对象。对于这些对象，G1会专门在Heap上开出一个个Humongous Area来存放，每个分区只放一个对象。但是申请这么大的空间是比较耗时的，而且这些区域也仅当Full GC时才进行处理，所以我们要尽量减少这样的对象产生。或者提高G1HeapRegionSize的值减少HumongousArea的创建。（G1HeapRegionSize=n 使用G1时Java堆会被分为大小统一的区(region)。此参数可以指定每个heap区的大小. 默认值将根据 heap size 算出最优解. 最小值为 1Mb, 最大值为 32Mb.）

不过在内存比较大的时，JVM默认把这个值设到了最大(32M)，此时我们只能通过分析程序本身找到这些对象并且尽量减少这样的对象产生。当然，相信随着G1 GC的发展，在后期的版本中相信这个最大值也会越来越大，毕竟G1号称是在1024 ~ 2048个Region时能够获得最佳性能。

Configuration Options	<code>-XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -Xms88g -Xmx88g -XX:InitiatingHeapOccupancyPercent=35 -XX:ConcGCThread=20</code>
------------------------------	--

总结

对于大量依赖于内存计算的Spark应用，GC调优显得尤为重要。在发现GC问题的时候，不要着急调试GC。而是先考虑是否存在Spark进程内存管理的效率问题，例如RDD缓存的持久化和释放。至于GC参数的调试，首先我们比较推荐使用G1 GC来运行Spark应用。相较于传统的垃圾收集器，随着G1的不断成熟，需要配置的选项会更少，能同时满足高吞吐量和低延迟的寻求。当然，GC的调优不是绝对的，不同的应用会有不同应用的特性，掌握根据GC日志进行调优的方法，才能以不变应万变。最后，也不能忘了先对程序本身的逻辑和代码编写进行考量，**例如减少中间变量的创建或者复制，控制大对象的创建，将长期存活对象放在Off-heap中等等。**

扩展：GC的一些配置

2018年2月19日 20:44

常见的4种GC

1. SerialGC

参数-XX:+UseSerialGC

就是Young区和old区都使用serial 垃圾回收算法

2. ParallelGC

参数-XX:+UseParallelGC

Young区：使用Parallel scavenge 回收算法

Old 区：可以使用单线程的或者Parallel 垃圾回收算法，由 -XX:+UseParallelOldGC 来控制

3. CMS

参数-XX:+UseConcMarkSweepGC

Young区：可以使用普通的或者parallel 垃圾回收算法，由参数 -XX:+UseParNewGC来控制

Old 区：只能使用Concurrent Mark Sweep

4. G1

参数：-XX:+UseG1GC

没有young/old区

一些配置解释

- -XX:+UseG1GC 使用 G1 (Garbage First) 垃圾收集器
- -XX:MaxGCPauseMillis=n 设置最大GC停顿时间(GC pause time)指标(target). 这是一个

软性指标(soft goal), JVM 会尽量去达成这个目标.

- -XX:InitiatingHeapOccupancyPercent=n 启动并发GC周期时的堆内存占用百分比. G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示"一直执行GC循环". 默认值为 45.

此参数, 控制的是触发Minor GC时的内存占用百分比

调优思想: 适当降低此参数, 可以增多MinorGC的频率, 则Full GC的频率会降低

建议: 35%~45%

- -XX:NewRatio=n 新生代与老生代(old/new generation)的大小比例(Ratio). 默认值为 2.
- -XX:SurvivorRatio=n eden/survivor 空间大小的比例(Ratio). 默认值为 8.
- -XX:MaxTenuringThreshold=n 提升老年代的最大临界值(tenuring threshold). 默认值为 15.
- -XX:ParallelGCThreads=n 设置垃圾收集器在并行阶段使用的线程数,默认值随JVM运行的平台不同而不同.一般设置的数量=服务器的核数
- -XX:ConcGCThreads=n 并发垃圾收集器使用的线程数量. 默认值随JVM运行的平台不同而不同(调节CMS)
- -XX:G1ReservePercent=n 设置堆内存保留为假天花板的总量,以降低提升失败的可能性. 默认值是 10.

比如堆内存100GB,如果假天花板是10%,则G1能够用的是90GB, 预留出10GB

- -XX:G1HeapRegionSize=n 使用G1时Java堆会被分为大小统一的区(region)。此参数可以指定每个heap区的大小. 默认值将根据 heap size 算出最优解. 最小值为 1Mb, 最大值为 32Mb.

Checkpoint机制

2018年7月12日 16:53

概述

checkpoint的意思就是建立检查点,类似于快照,例如在spark计算里面 计算流程DAG特别长,服务器需要将整个DAG计算完成得出结果,但是如果在这很长的计算流程中突然中间算出的数据丢失了,spark又会根据RDD的依赖关系从头到尾计算一遍,这样子就很费性能,当然我们可以将中间的计算结果通过cache或者persist放到内存或者磁盘中,但是这样也不能保证数据完全不会丢失,存储的这个内存出问题了或者磁盘坏了,也会导致spark从头再根据RDD计算一遍,所以就有了checkpoint,其中checkpoint的作用就是将DAG中比较重要的中间数据做一个检查点将结果存储到一个高可用的地方

代码示例:

```
object Driver2 {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("local").setAppName("wordcount")

    val sc=new SparkContext(conf)
    sc.setCheckpointDir("hdfs://hadoop01:9000/check01")

    val data=sc.textFile("d://data/word.txt")
    data.cache()
    data.checkpoint()

    val wordcount=data.flatMap(_._1.split(" ")).map(_._1).reduceByKey(_+_ )

    wordcount.cache()
    wordcount.checkpoint()

    wordcount.foreach{println}
  }
}
```

总结: Spark的CheckPoint机制很重要, 也很常用, 尤其在机器学习的一些迭代算法中很常见。比如一个算法迭代10000次, 如果不适用缓冲机制, 如果某分区数据丢失, 会导致整个计算

链重新计算，所以引入缓存机制。但是光引入缓存，也不完全可靠，比如缓存丢失或缓存存储不下，也会导致重新计算，所以使用CheckPoint机制再做一层保证。

补充：检查目录的路径，一般都是设置到HDFS上

Spark懒执行的意义

Spark中，Transformation方法都是懒操作方法，比如map,flatMap,reduceByKey等。当触发某个Action操作时才真正执行。

懒操作的意义：①不运行job就触发计算，避免了大量的无意义的计算，即避免了大量的无意义的中间结果的产生，即避免产生无意义的磁盘I/O及网络传输

②更深层次的意义在于，执行运算时，看到之前的计算操作越多，执行优化的可能性就越高

Spark共享变量

2018年2月13日 13:22

概述

Spark程序的大部分操作都是RDD操作，通过传入函数给RDD操作函数来计算。这些函数在不同的节点上并发执行，但每个内部的变量有不同的作用域，不能相互访问，所以有时会不太方便，Spark提供了两类共享变量供编程使用——广播变量和计数器。

1. 广播变量

这是一个只读对象，在所有节点上都有一份缓存，创建方法是SparkContext.broadcast()，比如：

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))  
  
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)  
  
scala> broadcastVar.value  
  
res0: Array[Int] = Array(1, 2, 3)
```

注意，广播变量是只读的，所以创建之后再更新它的值是没有意义的，一般用val修饰符来定义广播变量。

2. 计数器

计数器只能增加，是共享变量，用于计数或求和。

计数器变量的创建方法是SparkContext.accumulator(v, name)，其中v是初始值，name是名称。

示例如下：

```
scala> val accum = sc.accumulator(0, "My Accumulator")  
  
accum: org.apache.spark.Accumulator[Int] = 0
```

```
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

```
scala> accum.value
```

```
res1: Int = 10
```

spark解决数据倾斜问题

2018年10月21日 19:06

文件1

id	name
1	tom
2	rose

文件2

id	school	sno
1	s1	211
2	s2	222
3	s3	233
4	s2	244

期望得到的数据：

```
1 tom s1
2 rose s2
```

将少量的数据转化为Map进行广播，广播会将此 Map 发送到每个节点中，如果不进行广播，每个task执行时都会去获取该Map数据，造成了性能浪费。

完整代码

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

import scala.collection.mutable.ArrayBuffer

object joinTest extends App{
```

```

val conf = new SparkConf().setMaster("local[2]").setAppName("test")

val sc = new SparkContext(conf)

/**
 * map-side-join
 * 取出小表中出现的用户与大表关联后取出所需要的信息
 * */

//--如果RDD的类型是:RDD[(key,value)]->collectAsMap->Map(key->value)

//--用一个Map存储了小表数据（不一定非要用Map来存储小表数据）

val people_info = sc.parallelize(Array(("1","tom"),("2","rose"))).collectAsMap()

val student_all = sc.parallelize(Array(("1","s1","211"),

                                         ("1","s2","222"),

                                         ("1","s3","233"),

                                         ("1","s2","244")))

//将需要关联的小表进行广播

val people_bc = sc.broadcast(people_info)

/**
 * 使用mapPartition而不是用map，减少创建broadcastMap.value的空间消耗
 * 同时匹配不到的数据也不需要返回（）
 * */

val res = student_all.mapPartitions(iter =>{

    //获取小表的数据

    val stuMap = people_bc.value

```

```
val arrayBuffer = ArrayBuffer[(String,String,String)]()

//做两表的操作

iter.foreach{case (idCard,school,sno) =>{

  if(stuMap.contains(idCard)){

    //--在实现两个表的join操作

    arrayBuffer.+= ((idCard, stuMap.getOrElse(idCard,""),school))

  }

}}

arrayBuffer.iterator

})
```


扩展：重要源码解读

2018年7月12日 16:54

SparkConf类

```
/**
 * Configuration for a Spark application. Used to set various Spark parameters as key-value pairs.
 *
 * Most of the time, you would create a SparkConf object with `new SparkConf()`, which will load
 * values from any `spark.*` Java system properties set in your application as well. In this case,
 * parameters you set directly on the `SparkConf` object take priority over system properties.
 *
 * For unit tests, you can also call `new SparkConf(false)` to skip loading external settings and
 * get the same configuration no matter what the system properties are.
 *
 * All setter methods in this class support chaining. For example, you can write
 * `new SparkConf().setMaster("local").setAppName("My app")`.
 *
 * @param loadDefaults whether to also load values from Java system properties
 *
 * @note Once a SparkConf object is passed to Spark, it is cloned and can no longer be modified
 * by the user. Spark does not support modifying the configuration at runtime.
 */
```

SparkContext实例化的时候需要传进一个SparkConf作为参数，SparkConf描述整个Spark应用程序的配置信息，

SparkConf可以进行链式的调用，即：

```
new SparkConf().setMaster("local").setAppName("TestApp")
```

□ SparkConf的部分源码如下：

// 用来存储key-value的配置信息

```
private val settings = new ConcurrentHashMap[String, String]()
```

// 默认会加载 "spark.*" 格式的配置信息

```
if (loadDefaults) {
```

// Load any spark.* system properties

```
for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark.")) {
```

```
    set(key, value)
```

```
}
```

```
}
```

/** Set a configuration variable. */

```
def set(key: String, value: String): SparkConf = {
```

```
    if (key == null) {
```

```
        throw new NullPointerException("null key")
```

```
    }
```

```
    if (value == null) {
```

```

        throw new NullPointerException("null value for " + key)
    }
    logDeprecationWarning(key)
    settings.put(key, value)
    // 每次进行设置后都会返回SparkConf自身，所以可以进行链式的调用
    this
}

```

SparkContext类

```

/**
 * Main entry point for Spark functionality. A SparkContext represents the connection to a Spark
 * cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
 *
 * Only one SparkContext may be active per JVM. You must `stop()` the active SparkContext before
 * creating a new one. This limitation may eventually be removed; see SPARK-2243 for more details.
 *
 * @param config a Spark Config object describing the application configuration. Any settings in
 * this config overrides the default configs as well as system properties.
 */

```

SparkContext是整个Spark功能的入口，代表了应用程序与整个集群的连接点，

Spark应用程序是通过SparkContext发布到Spark集群的，并且Spark程序的运行都是在SparkContext为核心的调度指挥下进行的，SparkContext崩溃或者结束就代表Spark应用程序执行结束，所以SparkContext在Spark中是非常重要的一个类。

□ SparkContext部分源码(只选取重要部分):

SparkContext最主要的作用:①初始化SparkEnv对象 ②初始化并启动三个调度模块DAG, Task,Backend, 此外，会建立各个工作节点的心跳机制，用于检测和监控

```

// 是否允许存在多个SparkContext，默认是false
// If true, log warnings instead of throwing exceptions when multiple SparkContexts are active
private val allowMultipleContexts: Boolean =
    config.getBoolean("spark.driver.allowMultipleContexts", false)

// An asynchronous listener bus for Spark events
private[spark] val listenerBus = new LiveListenerBus

// 追踪所有执行持久化的RDD
// Keeps track of all persisted RDDs
private[spark] val persistentRdds = new TimeStampedWeakValueHashMap[Int, RDD[_]]

```

```

// System property spark.yarn.app.id must be set if user code ran by AM on a YARN cluster
if (master == "yarn" && deployMode == "cluster" && !_conf.contains("spark.yarn.app.id")) {
    throw new SparkException("Detected yarn cluster mode, but isn't running on a cluster. " +
        "Deployment to YARN is not supported directly by SparkContext. Please use spark-submit.")
}

// Create the Spark execution environment (cache, map output tracker, etc)
_env = createSparkEnv(_conf, isLocal, listenerBus)
SparkEnv.set(_env)
//设置executor进程的大小，默认1GB
_executorMemory = _conf.getOption("spark.executor.memory")
    .orElse(Option(System.getenv("SPARK_EXECUTOR_MEMORY")))
    .orElse(Option(System.getenv("SPARK_MEM")))
    .map(warnSparkMem))
    .map(Utils.memoryStringToMb)
    .getOrElse(1024)

// We need to register "HeartbeatReceiver" before "createTaskScheduler" because Executor will
// retrieve "HeartbeatReceiver" in the constructor.
_heartbeatReceiver = env.rpcEnv.setupEndpoint(
    HeartbeatReceiver.ENDPOINT_NAME, new HeartbeatReceiver(this))

// Create and start the scheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler(this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
_taskScheduler.start()

_env.blockManager.initialize(_applicationId)
_env.metricsSystem.start()
}

```

其实SparkContext中最主要的三大核心对象就是DAGScheduler、TaskScheduler、SchedulerBackend

- 1) DAGScheduler主要负责分析依赖关系，然后将DAG划分为不同的Stage（阶段），其中每个Stage由可以并发执行的一组Task构成，这些Task的执行逻辑完全相同，只是作用于不同的数据。
- 2) TaskScheduler作用是为创建它的SparkContext调度任务，即从DAGScheduler接收不同Stage的任务，并且向集群提交这些任务，并为执行特别慢的任务启动备份任务
- 3) SchedulerBackend作用是依据当前任务申请到的可用资源，将Task在Executor进程中启动并执行，完成计算的调度过程。

SparkEnv

```
/**
 * :: DeveloperApi ::
 * Holds all the runtime environment objects for a running Spark instance (either master or worker),
 * including the serializer, RpcEnv, block manager, map output tracker, etc. Currently
 * Spark code finds the SparkEnv through a global variable, so all the threads can access the same
 * SparkEnv. It can be accessed by SparkEnv.get (e.g. after creating a SparkContext).
 *
 */
```

SparkEnv是Spark的执行环境对象，其中包括但不限于：

- 1) serializer
- 2) RpcEnv
- 3) BlockManager
- 4) MapOutputTracker(Shuffle过程中非常重要)等

在local模式下Driver会创建Executor，在Standalone部署模式下，Worker上创建Executor。所以SparkEnv存在于Spark任务调度时的每个Executor中，SparkEnv中的环境信息是对一个job中所有的Task都是可见且一致的。确保运行时的环境一致。

SparkEnv的构造步骤如下：

1. 创建安全管理器SecurityManager;

Spark currently supports authentication via a shared secret. Authentication can be configured to be on via the `spark.authenticate` configuration parameter. This parameter controls whether the Spark communication protocols do authentication using the shared secret. This authentication is a basic handshake to make sure both sides have the same shared secret and are allowed to communicate. If the shared secret is not identical they will not be allowed to communicate. The shared secret is created as follows:

For Spark on YARN deployments, configuring `spark.authenticate` to true will automatically handle generating and distributing the shared secret. Each application will use a unique shared secret.

For other types of Spark deployments, the Spark parameter `spark.authenticate.secret` should be configured on each of the nodes. This secret will be used by all the Master/Workers and applications.

SecurityManager是Spark的安全认证模块，通过共享密钥进行认证。启用认证功能可以通过参数 `spark.authenticate` 来配置。此参数控制spark通信协议是否使用共享密钥进行认证。这种认证方式基于握手机制，以确保通信双方都有相同的共享密钥时才能通信。如果共享密钥不一致，则双方将无法通信。可以通过以下过程来创建共享密钥：

①在spark on YARN部署模式下，配置`spark.authenticate`为true，就可以自动产生并分发共享密钥。每个应用程序都使用唯一的共享密钥。

②其他部署方式下，应当在每个节点上都配置参数`spark.authenticate.secret`。此密钥将由所有Master、worker及应用程序来使用。

2. 创建RpcEnv;

Spark1.6推出的RpcEnv、RpcEndPoint、RpcEndpointRef为核心的新型架构下的RPC通信方式，在底层封装了Akka和Netty，也为未来扩充更多的通信系统提供了可能。

①如果底层用的是Akka的RPC通信

RpcEnv=ActorSystem

RpcEndPoint=Actor

RpcEndpointRef=Actor通信的对象

②如果底层用的是Netty的RPC通信

RpcEnv=NettyServer

RpcEndPoint=NettyClient

RpcEndpointRef=NettyClient的通信对象

Spark1.6之间用的是Akka，1.6之后用的是Netty

3. 创建ShuffleManager

ShuffleManager负责管理本地及远程的Block数据的shuffle操作。ShuffleManager默认通过反射方式生成的SortShuffleManager的实例。默认使用的是sort模式的SortShuffleManager，当然也可以通过修改属性spark.shuffle.manager为hash来显式控制使用HashShuffleManager。

4. 创建Shuffle Map Task任务输出跟踪器MapOutputTracker

MapOutputTracker用于跟踪Shuffle Map Task任务的输出状态，此状态便于Result Task任务获取地址及中间结果。Result Task 会到各个Map Task 任务的所在节点上拉取Block，这一过程叫做Shuffle。

MapOutputTracker 有两个子类：

①MapOutputTrackerMaster (for driver)

②MapOutputTrackerWorker (for executors)

shuffleReader读取shuffle文件之前就是去请求MapOutputTrackerMaster 要自己处理的数据在哪里？

MapOutputTrackerMaster给它返回一批 MapOutputTrackerWorker的列表（地址，port等信息）

然后进行shuffleReader

5. 内存管理器MemoryManager

spark的内存管理有两套方案，新旧方案分别对应的类是UnifiedMemoryManager和StaticMemoryManager。

旧方案是静态的，storageMemory（存储内存）和executionMemory（执行内存）拥有的内存是独享的不可相互借用，故在其中一方内存充足，另一方内存不足但又不能借用的情况下会造成资源的浪费。新方案是统一管理的，初始状态是内存各占一半，但其中一方内存不足时可以向对方借用，对内存资源进行合理有效的利用，提高了整体资源的利用率。

Spark的内存管理，是把内存分为两大块，包括storageMemory和executionMemory。其中storageMemory用来缓存rdd，unroll partition，direct task result、广播变量等。executionMemory用于

shuffle、join、sort、aggregation 计算中的缓存。除了这两者以外的内存都是预留给系统的。每个Executor进程都有一个MemoryManager。

MemoryManager 的选择是由spark.memory.useLegacyMode来控制的，默认是使用UnifiedMemoryManager来管理内存。用的是动态管理机制。即存储缓存和执行缓存可以相互借用，动态管理的优势在于可以充分里用缓存，不会出现一块缓存紧张，而另外一块缓存空闲的情况。

6. 创建块传输服务NettyBlockTransferService

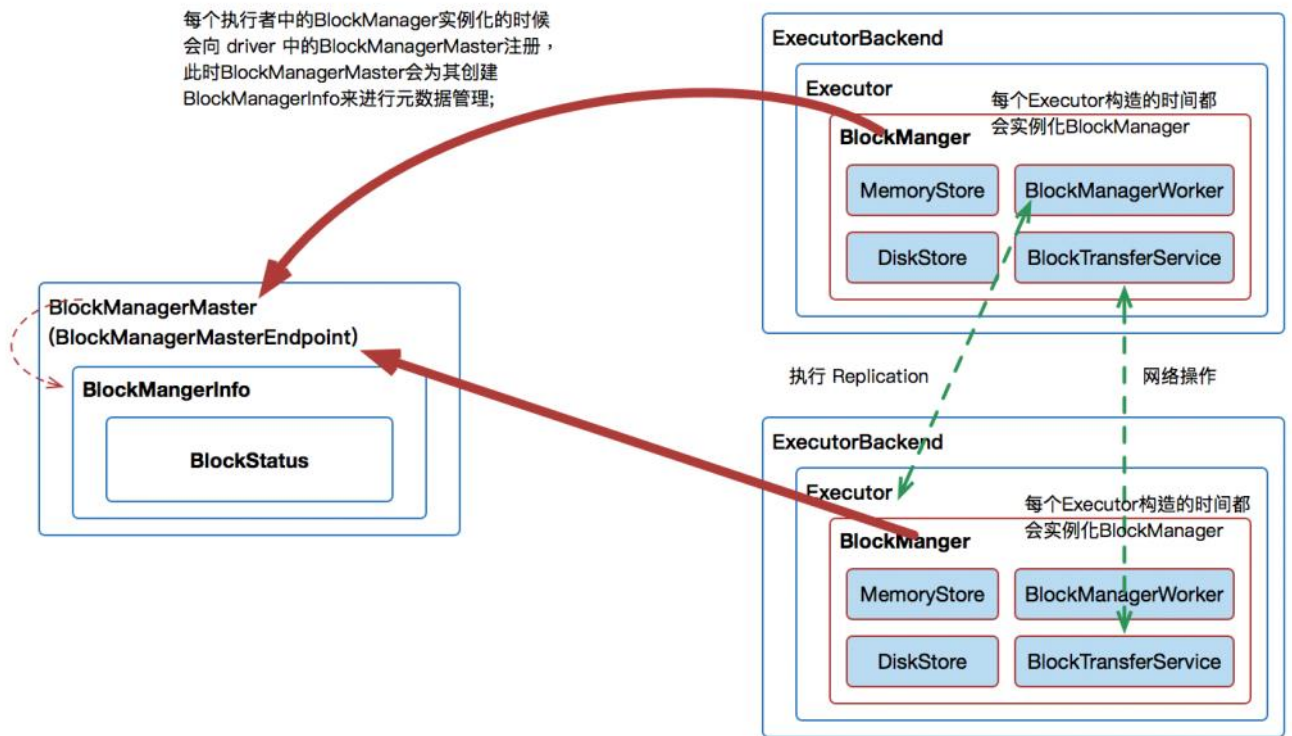
NettyBlockTransferService使用Netty提供的网络应用框架，提供web服务及客户端，获取远程节点上Block的集合。底层的fetchBlocks方法用于获取远程shuffle文件中的数据。

7. 创建BlockManagerMaster

BlockManagerMaster负责对BlockManager的管理和协调

8. 创建块管理器BlockManager

```
/**
 * Manager running on every node (driver and executors) which provides interfaces for putting and
 * retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap).
 *
 */
```



如上图所示，每一个Executor进程创建时，都会创建一个BlockManager,而所有的BlockManager都由BlockManagerMaster来管理。

BlockManager主要提供了读取和写数据的接口，可以从本地或者是远程读取和写数据，读写数据可以基于内存、磁盘或者是堆外空间 (OffHeap)。

9. 创建广播管理器BroadcastManager

BroadcastManager用于将配置信息、序列化后的RDD以及ShuffleDependency等信息在本地存储。此外，BroadcastManager会将数据从一个节点广播到其他的节点上。例如Driver上有一张表，而Executor中的每个并行执行的Task（100万个）都要查询这张表，那我们通过广播的方式就只需要往每个Executor把这张表发送一次就行了。Executor中的每个运行的Task查询这张唯一的表，而不是每次执行的时候都从Driver获得这张表。避免Driver节点称为性能瓶颈。

Spark的广播机制

当声明一个广播变量时，最终的结果是所有的节点都会收到这个广播变量，Spark底层的实现细节如下：

①驱动程序driver将序列化的对象分为小块并存储在驱动器的blockmanager中。

②根据spark.broadcast.compress配置属性确认是否对广播消息进行压缩，根据spark.broadcast.blockSize配置

属性确认块的大小，默认为4MB。

③为每个block生成BroadcastBlockId。即driver端会把广播数据分块，每个块做为一个block存进driver端的BlockManager

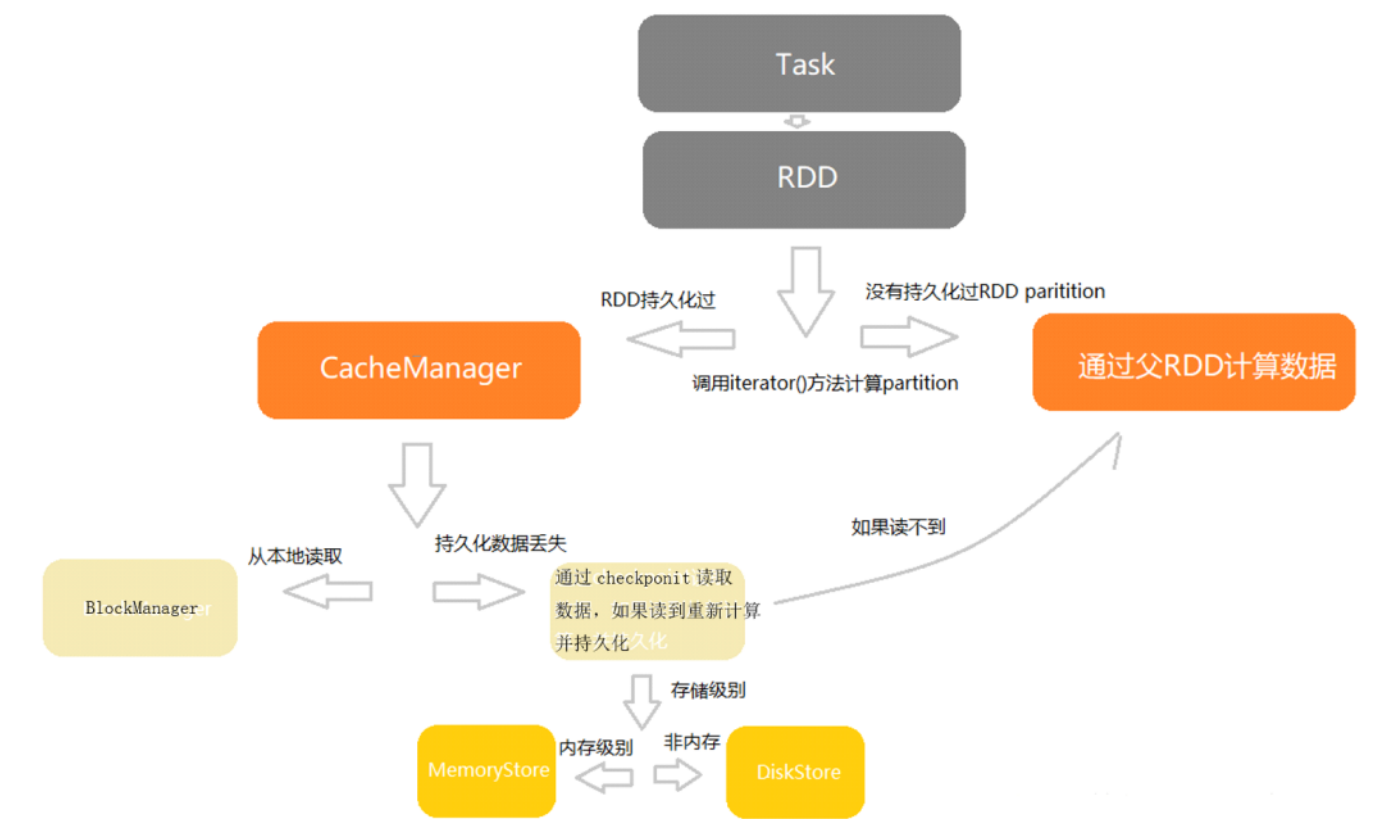
④每个executor会试图获取所有的块，来组装成一个完整的broadcast的变量。“获取块”的方法是首先从executor自身的BlockManager中获取，如果自己的BlockManager中没有这个块，就从别的BlockManager中获取。这样最初的时候，driver是获取这些块的唯一的源。

⑤但是随着各个BlockManager从driver端获取了不同的块(TorrentBroadcast会有意避免各个executor以同样的顺序获取这些块)，这样做的好处是可以使“块”的源变多。

⑥每个executor就可能从多个源中的一个,包括driver和其它executor的BlockManager中获取块，**这要就使得流量在整个集群中更均匀，而不是由driver作为唯一的源。**

10. 创建缓存管理器CacheManager

CacheManager用于管理和持久化RDD



11. 创建监听总线ListenerBus和检测系统MetricsSystem

Spark整个系统运行情况的监控是由ListenerBus以及MetricsSystem 来完成的。spark监听总线

(LiveListenerBus) 负责监听spark中的各种事件，比如job启动、各Worker的内存使用率、BlockManager的添加等等，并通过MetricsSystem展示给UI

12. 创建SparkEnv

当所有的组件准备好之后，最终可以创建执行环境SparkEnv

扩展：贝叶斯定理

贝叶斯介绍

英国数学家，贝叶斯在数学方面主要研究概率论.对于统计决策函数、统计推断、统计的估算等做出了贡献。

他对统计推理的主要贡献是使用了"逆概率"这个概念，并把它作为一种普遍的推理方法提出来。贝叶斯定理原本是概率论中的一个定理，这一定理可用一个数学公式来表达，这个公式就是著名的贝叶斯公式。

贝叶斯定理

公式：

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}$$

求得是后验概率
等式右侧为先验概率

案例说明一：

假设：H代表胃癌事件，X代表胃疼事件。则P（H | X）表示的是：当一个人胃疼时，是胃癌的概率有多少？
P（H | X）称为后验概率，我们利用贝叶斯公式的目的就是求得这个后验概率是多少。

而求后验的前提是要知道 P（X | H），P（H），以及P（X）的概率。

针对本例，这三个是可以根据临床试验数据得到的，

P（X | H）表示的是：胃癌发生时，胃疼的概率，假设是：85%。P（X | H）称为先验概率，先验概率一般是由大量过去的经验总结得到，或者也可以通过抽样得到。

比如说：电商的28定律（20%的热门商品集中了80%的访问流量）就是一个总结得到的先验概率，当然我们也可以通过抽样，通过实验数据来得到这个结论，根据大数定律，当实验样本越大，越接近于正确结论。

P（H）表示的是：总人群患胃癌的概率：0.1%

P（X）表示的是：总人群患胃疼的概率：40%

有了以上数据后，问：当一个人胃疼时，他患胃癌的概率P（X | H）是多少？

结果是：0.85*0.001%0.4=0.002125=0.021

即当一个人胃疼时，是胃癌的概率是2.1%。这个概率是很小的。

案例说明二：

比如我们要判断某一封邮件是否是垃圾邮件，假设：H代表此邮件是垃圾邮件
X代表此邮件里出现了"美女"词汇。则P（H | X）表示的是：当一封邮件里出现"美女"词汇时，它是垃圾邮件的概率。

为了求得这个后验概率，我们需要知道P（X | H）、P（H）、P（X）的概率

- $P(X|H)$ 表示一封垃圾邮件里，出现"美女"词汇的概率。关键这个先验概率怎么求得，因为这个问题并不像胃癌案例那样受到广泛关注，所以并没有现成的先验概率供使用，所以这个概率需要通过实验样本来获取。

实现步骤：

1.从已有的垃圾邮件箱里随机收取100封垃圾邮件，然后统计每封垃圾邮件里，出现"美女"的次数，假设最后的结果：100封垃圾邮件里，有20封出现了"美女"。

则： $P(X|H) = 20\%$

当然，这个先验概率如果为了更准确，可以扩大样本数据或增加实验次数。

2.接下来求 $P(H)$ 和 $P(X)$

$P(H)$ 表示的是一封邮件是垃圾邮件的概率，

$P(X)$ 表示的是一封邮件里出现"美女"的概率，

这两个也没有现成的先验概率，所以需要通过实验获取。

我们可以从邮件箱里（包含正常邮件和垃圾邮件），随机抽取500封邮件，然后统计有多少封是垃圾邮件，以及统计每封邮件里出现"美女"的次数。

假设：500封邮件里，出现了60封垃圾邮件。

500封邮件里，出现了200次"美女"

则： $P(H) = 60/500\%$ $P(X) = 200/500\%$

所以综上，我们可以利用贝叶斯来对邮件过滤，当收到一封邮件时，这封邮件包含了"美女"词汇，请问它是正常邮件还是垃圾邮件？

经计算可得： $P(H|X) = 0.2 * (60/500) / (200/500) = 0.06 = 6\%$

总结：本例中，根据概率的阈值，来判定一封邮件是否是垃圾邮件。比如算得的概率是80%，则可以认定此邮件是垃圾邮件

即这封邮件是垃圾邮件的概率是6%，一般地，垃圾邮件设定的阈值在60%~100%。而6%<60%，所以这封邮件是一封正常邮件。

针对本例，如果换个条件，比如： $P(X)$ 表示的是一封邮件里出现"发票"的概率，

$P(X|H) = 0.9$ 一封垃圾邮件里出现"发票"的概率是90%

$P(H) = 0.2$ 一封邮件是垃圾邮件的概率是20%

$P(X) = 0.25$ "发票"在邮件中出现的概率25%

最后算得：

当一封邮件含有"发票"时，它是垃圾邮件的概率是 $0.9 * 0.2 / 0.25 = 72\%$

它是垃圾邮件

贝叶斯公式实际可以做如下变形：

$$P(X|H) \cdot P(H)/P(X) = P(H|X)$$

先验概率 · 似然比 = 后验概率

贝叶斯决策理论方法是统计模型决策中的一个基本方法，其基本思想是：

- 1、已知类条件概率密度参数表达式和先验概率。
- 2、利用贝叶斯公式转换成后验概率。
- 3、根据后验概率大小进行决策分类。

□ 案例说明三：

根据上两个案例，可以将贝叶斯公式进行变换得到下面的公式：

【公式】

$$\frac{P(H_1)}{P(H_2)} \cdot \frac{P(E|H_1)}{P(E|H_2)} = \frac{P(H_1|E)}{P(H_2|E)}$$

H1和H2是两个事件，一般是对立事件，即 $P(H_1) + P(H_2) = 100\%$

$P(E|H_1)$ 表示当满足H1条件时，发生E事件的概率。 $P(E|H_2)$ 同理

$P(H_1|E)$ 或 $P(H_2|E)$ 表示的是当E事件发生时，H1或H2发生的概率，即我们要求的后验概率。

场景说明：

一机器在良好状态生产合格产品几率是90%，在故障状态生产合格产品几率是30%，机器良好的概率是75%，若一日第一件产品是合格品，那么此日机器良好的概率是多少。

$$P(H|X)$$

$$P(X|H) = 0.9$$

$$P(H) = 0.75$$

$$P(X) =$$

这道题心算即可——

- (1) 先验比率是 $75\% : (1-75\%) = 3 : 1$;
- (2) 似然比率 (Likelihood ratio) = $90\% : 30\% = 3$;
- (3) 两者相乘，得后验比率 = $9 : 1$ ；然后
- (4) 标准化 (normalize)，得后验概率 = $9 / (9+1) = 90\%$ 。

解释：

假设美国和日本要打架，美国军事战斗力是75%，日本军事战斗力是25%。

问：此时，美国和日本的战力比是多少？

答： $75\% : 25\% = 3 : 1$ 。

问：现在，德国发明了一种新型战斗机，能增强军事战斗力，卖给了美国和日本。但是美国和日本的军事体量不同，美国引进这种战斗机后，战斗力增加了 90 倍；而日本引入之后战斗力只增加 30 倍。

请问，两国引入这种战斗机后，目前的战力比是多少？

答：美国战斗力： $3 \times 90 = 270$ ，日本战斗力： $1 \times 30 = 30$ 。

美国：日本 = $270 : 30 = 9 : 1$

即美国军事力量目前占90%，日本军事力量10%。如果美国打日本，有9成把握取胜。

【解释】

现在让我把上面的故事翻译一下，套到题目上。

(1) 你一开始有两个假说，良好 (H1) 和故障 (H2) ，

H1和H2的先验概率比 = $P(H1) : P(H2) = 3 : 1$ 。【用 $P(\text{良好}) = 75\%$ 即可推出。】

(2)现在你拿到了一个证据E：第一天产品是合格的。这个证据的作用，在于改变两个假说的概率之比。

根据贝叶斯定理，这个「1 个零件合格」的证据会产生两个效果：

a. 会让「机器良好」(H1) 的相对概率增加 90 倍【因为 $P(E|H1) = 90\%$ 】，及

b. 会让「机器故障」(H2) 的相对概率增加 30 倍【因为 $P(E|H2) = 30\%$ 】。

【公式上看，应该是分别缩减到 0.9 倍和 0.3 倍，但是化为整数比较容易思考。】

(3) 根据(2)，我们有了证据 E 之后，良好和故障的概率之比变为 $3 \times 90 : 1 \times 30 = 270 : 30 = 9 : 1$ 。

(4) 根据现有条件，其实还算不出 $P(\text{良好}|\text{1个合格})$ 。要算出 $P(\text{良好}|\text{1个合格})$ 的具体数值，还须明确给出一个条件，即「良好」和「故障」已经包括所有的假设了：

$P(\text{良好}|\text{1个合格}) + P(\text{故障}|\text{1个合格}) = 1$ 。

然后联立刚才得到的

$P(\text{良好}|\text{1个合格}) : P(\text{故障}|\text{1个合格}) = 9 : 1$ ，可解出

$P(\text{良好}|\text{1个合格}) = 90\%$ 。

【公式】

$$\frac{P(H_1)}{P(H_2)} \cdot \frac{P(E|H_1)}{P(E|H_2)} = \frac{P(H_1|E)}{P(H_2|E)}$$

对应刚才的推理的，是贝叶斯定理常见形式的一个变体——分别对H1、H2列式，两式相除即可得。

最左边一项是先验比率（美国:日本=75:25），

中间一项是似然比率（新型战机对美国和日本战力影响，90:30），

最右边一项是后验比率（最后的结果，9:1）。

扩展：朴素贝叶斯分类器

概念介绍

朴素贝叶斯分类器是基于贝叶斯条件概率论为基础的，总体思想是：给定一个样本点，判断它属于第一类还是第二类的概率高。

朴素贝叶斯分类器原理

利用贝叶斯原理实现的分类器，过滤垃圾邮件，当某一封邮件里同时出现"贷款","发票","理财"词汇时，请问它是垃圾邮件吗？

事件属性	学习集样本数量	贷款（X1）	发票（X2）	理财（X3）
总邮件数量	100			
正常邮件	80	5	0	5
垃圾邮件（H）	20	20	18	15

根据贝叶斯公式：

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}$$

H 表示：此邮件是垃圾邮件事件

X 表示：同时出现 "贷款"," 发票" ,"理财"词汇 事件

上式中：

P（H）是最易求得的：20/100=20%

P（X | H）和 P（X）不易求得，因为X 是由三个属性 X1,X2,X3组成的，

这里可以根据朴素贝叶斯公式求得：

$$P(X|C_i) = \prod_{k=1}^n p(X_k|C_i)$$

朴素贝叶斯的理论是：认为每个变量 X1，X2.....之间是相互独立的，即"发票"词汇的出现和"贷款"词汇的出现没有必然联系。朴素贝叶斯就是基于这种场景下来求解的，因为这种求解方式简单，不考虑相关复杂性（如果考虑相关性，P（X | H）的计算开销将会非常大，需要计算 $2^n - 1$ 次，n为变量个数），所以叫朴素贝叶斯。

所以， $P（X | H）= P（X1 | H） \cdot P（X2 | H） \cdot P（X3 | H）$

$P（X）= P（X1） \cdot P（X2） \cdot P（X3）$

最后，求得 $P(H|X) = 0.6$

朴素贝叶斯分类器如果推广到一般形式，是判断：给定一个待分类的样本X，判断这个样本是属于哪个类别 C_i ，判断它属于第C1类的概率、C2类的概率..... C_i 类的概率，哪个概率最高，即属于哪类

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}$$

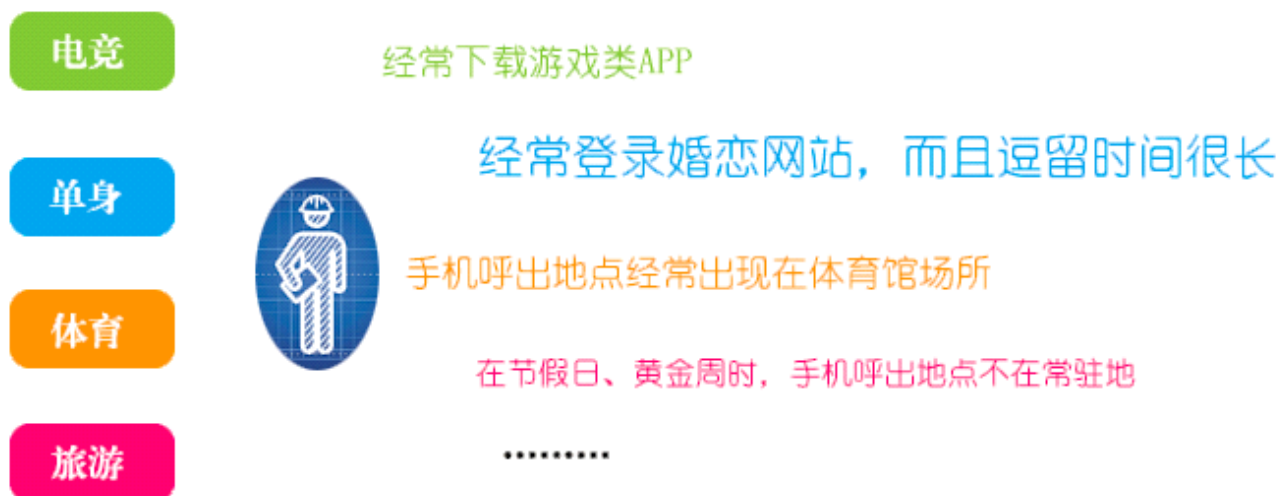
贝叶斯分类器的应用

1.流失用户的预警

贝叶斯分类广泛应用于现实生活中，比如流失用户的预警。

我们可以先根据以往的数据建立学习集，得出流失用户的特征，比如上线频率低，充值次数、充值金额低等。我们可以利用贝叶斯分类器判断出当前哪些用户是流失用户，然后可以推送一些优惠或是提高抽卡、装备爆率等措施挽留。

2.用户画像



比如为用户建立其用户画像，分析其具备哪些特点爱好，然后做定向推送。也可以用贝叶斯来实现，比如系统里一共有10个标签（分类），然后结合用户数据，根据贝叶斯公式算出此用户属于每个标签（分类）的概率，这里我们可以设定一个阈值，比如35%。当用户属于此标签的概率 $\geq 35\%$ 时，就把此标签贴给这个用户。以后，可以定期向用户推荐符合其爱好的信息。

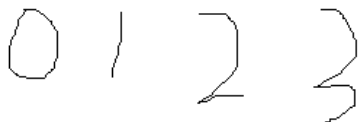
3.人脑中的贝叶斯

此外，每个人的人脑也是一个贝叶斯分类器。

比如针对下图，请问它代表什么？



如果提供下图，我们的答案是：数字0



如果提供下图，我们的答案是：字母o



如果提供下图，我们的答案是：鸡蛋



所以，我们人脑在判断一种事物时，是根据某个特定的学习集来判断的，而从人出生开始，我们的大脑在接收各种的学习集，数字学习集、文字学习集、图像学习集等等，所以当一事物在某个学习集中出现时，能够立马辨识事物的含义。

但是如果没有给定一个特定的学习集，让人来判断



的含义，我们人会给出所有的可能，

“它可能代表是数0”

“它可能代表字母o”

“它可能代表鸡蛋”

此外，我们还会更倾向性的给出每种答案的概率，

“它是0的可能性更大，因为它比字母o要长些，而鸡蛋的概率更小，因为它不够椭圆”

所以说，在很多时候，我们的人脑的工作原理实际上就是贝叶斯原理。

最后，再来看这句话：

早上好，各位学同们！欢迎学来习数据挖掘。

——研究表明，字符的顺序不一定能影响阅读

Prof. Daniel Kahneman的研究

这是因为我们已经有丰富的阅读学习集，当我们看到“早”，“好”，“上”时，组成的含义“早上好”并不是由实际顺序决定，而是由贝叶斯原理从大脑中已有的学习中得到的。所以说，贝叶斯无处不在。

案例

下表是一组学习集数据

Tom是一名网球爱好者，我们对Tom每天是否打网球进行了观测，一共观测了两周（14天），观测了四个变量：

天气状况（X1）——晴天、阴天、雨天

气候（x2）——干燥、温和、凉爽

气温（X3）——高温、正常

风力（X4）——弱风、强风

在四个变量作用下，Tom是否打网球。现要求根据已有的学习集建立贝叶斯分类器，并预测：

当天气状况=overcast，气候=mild，气温=normal，风力=weak时，Tom是否会打网球？

$P(\text{打网球}|\text{overcast, mild, normal, weak}) = ?$

$P(\text{不打网球}|\text{overcast, mild, normal, weak}) = ?$

观测天数	天气状况（X1）	气候（X2）	气温（X3）	风力（X4）	结果
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

□ 代码示例：

```
>data <- matrix(c("sunny","hot","high","weak","no",  
"sunny","hot","high","strong","no",  
"overcast","hot","high","weak","yes",
```

```

"rain","mild","high","weak","yes",
"rain","cool","normal","weak","yes",
"rain","cool","normal","strong","no",
"overcast","cool","normal","strong","yes",
"sunny","mild","high","weak","no",
"sunny","cool","normal","weak","yes",
"rain","mild","normal","weak","yes",
"sunny","mild","normal","strong","yes",
"overcast","mild","high","strong","yes",
"overcast","hot","normal","weak","yes",
"rain","mild","high","strong","no"),
byrow = TRUE,
dimnames = list(day = c(),condition = c("outlook","temperature","humidity","wind","playtennis")),
nrow=14, ncol=5);

```

#计算先验概率

```

> prior.yes = sum(data[,5] == "yes") / length(data[,5]);
> prior.no = sum(data[,5] == "no") / length(data[,5]);

> naive.bayes.prediction <- function(condition.vec) {
# Calculate unnormalized posterior probability for playtennis = yes.
playtennis.yes <-
sum((data[,1] == condition.vec[1]) & (data[,5] == "yes")) / sum(data[,5] == "yes") * # P(outlook = f_1 | playtennis =
yes)
sum((data[,2] == condition.vec[2]) & (data[,5] == "yes")) / sum(data[,5] == "yes") * # P(temperature = f_2 |
playtennis = yes)
sum((data[,3] == condition.vec[3]) & (data[,5] == "yes")) / sum(data[,5] == "yes") * # P(humidity = f_3 | playtennis
= yes)
sum((data[,4] == condition.vec[4]) & (data[,5] == "yes")) / sum(data[,5] == "yes") * # P(wind = f_4 | playtennis =
yes)
prior.yes; # P(playtennis = yes)
# Calculate unnormalized posterior probability for playtennis = no.
playtennis.no <-
sum((data[,1] == condition.vec[1]) & (data[,5] == "no")) / sum(data[,5] == "no") * # P(outlook = f_1 | playtennis =
no)
sum((data[,2] == condition.vec[2]) & (data[,5] == "no")) / sum(data[,5] == "no") * # P(temperature = f_2 |
playtennis = no)
sum((data[,3] == condition.vec[3]) & (data[,5] == "no")) / sum(data[,5] == "no") * # P(humidity = f_3 | playtennis

```

```

= no)
sum((data[,4] == condition.vec[4]) & (data[,5] == "no")) / sum(data[,5] == "no") * # P(wind = f_4 | playtennis =
no)
prior.no;# P(playtennis = no)
return(list(post.pr.yes = playtennis.yes,
post.pr.no = playtennis.no,
prediction = ifelse(playtennis.yes >= playtennis.no, "yes", "no"))); }

> naive.bayes.prediction(c("overcast", "mild", "normal", "weak"));
$post.pr.yes
[1] 0.05643739

$post.pr.no
[1] 0

$prediction
[1] "yes"

```

结果:

Tom会打网球

Spark Sql

2018年2月10日 15:59

概述

Spark为**结构化数据**处理引入了一个称为Spark SQL的编程模块。它提供了一个称为**DataFrame（数据框）**的编程抽象，DF的底层仍然是RDD，并且可以充当分布式SQL查询引擎。

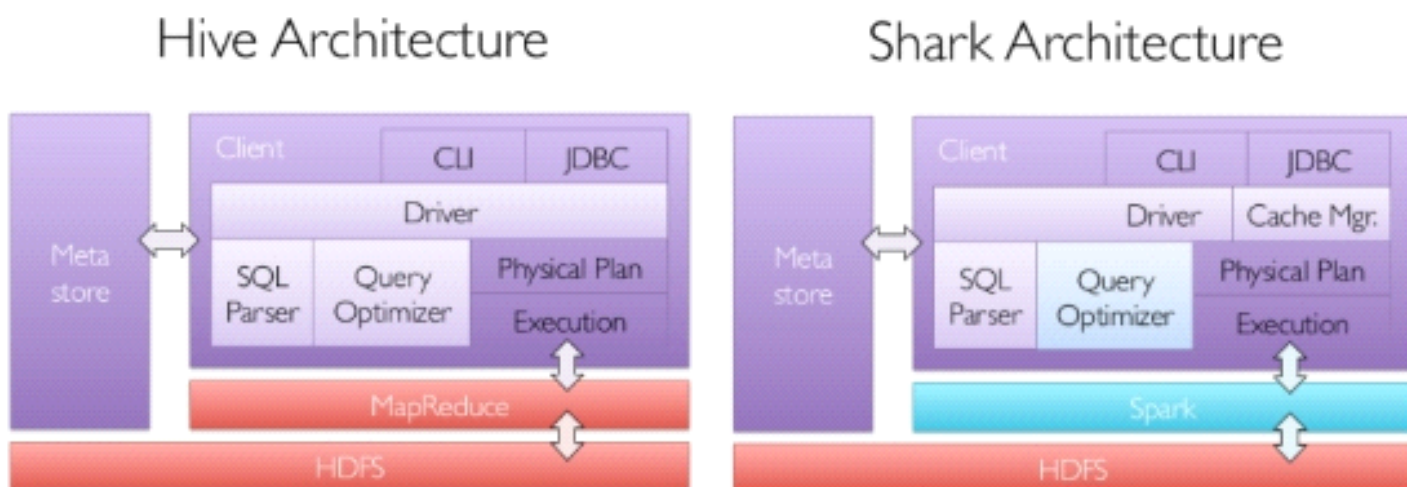
SparkSQL的由来

SparkSQL的前身是Shark。在Hadoop发展过程中，为了给熟悉RDBMS但又不理解MapReduce的技术人员提供快速上手的工具，Hive应运而生，是当时唯一运行在hadoop上的SQL-on-Hadoop工具。但是，MapReduce计算过程中大量的中间磁盘落地过程消耗了大量的I/O，运行效率较低。

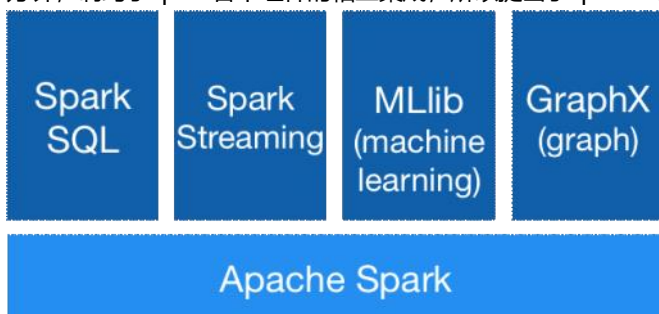
后来，为了提高SQL-on-Hadoop的效率，大量的SQL-on-Hadoop工具开始产生，其中表现较为突出的是：

- 1) MapR的Drill
- 2) Cloudera的Impala
- 3) Shark

其中Shark是伯克利实验室Spark生态环境的组件之一，它基于Hive实施了一些改进，比如引入缓存管理，改进和优化执行器等，并使之能运行在Spark引擎上，从而使得SQL查询的速度得到10-100倍的提升。



但是，随着Spark的发展，对于野心勃勃的Spark团队来说，Shark对于hive的太多依赖（如采用hive的语法解析器、查询优化器等等），制约了Spark的One Stack rule them all的既定方针，制约了spark各个组件的相互集成，所以提出了sparkSQL项目。



SparkSQL抛弃原有Shark的代码，汲取了Shark的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive兼容性等，重新开发了SparkSQL代码。

由于摆脱了对hive的依赖性，SparkSQL无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。

2014年6月1日，Shark项目和SparkSQL项目的主持人Reynold Xin宣布：停止对Shark的开发，团队将所有资源放SparkSQL项目上，至此，Shark的发展画上了句号。

SparkSql特点

- 1) 引入了新的RDD类型SchemaRDD，可以像传统数据库定义表一样来定义SchemaRDD
- 2) 在应用程序中可以混合使用不同来源的数据，如可以将来自HiveQL的数据和来自SQL的数据进行Join操作。
- 3) 内嵌了查询优化框架，在把SQL解析成逻辑执行计划之后，**最后变成RDD的计算**

为什么sparkSQL的性能会得到怎么大的提升呢？

主要sparkSQL在下面几点做了优化：

- 1) 内存列存储（In-Memory Columnar Storage）

列存储的优势：

①海量数据查询时，不存在冗余列问题。如果是基于行存储，查询时会产生冗余列，消除冗余列一般在内存中进行的。或者基于行存储的查询，实现物化索引（建立B-tree B+tree），但是物化索引也是需要耗费cpu的

②基于列存储，每一列数据类型都是同质的，好处一可以避免数据在内存中类型的频繁转换。好处二可以采用更高效的压缩算法，比如增量压缩算法，二进制压缩算法。性别：男 女 男 女 0101

SparkSQL的表数据在内存中存储不是采用原生态的JVM对象存储方式，而是采用内存列存储，如下图所示。



该存储方式无论在**空间占用量**和**读取吞吐率**上都占有很大优势。

对于原生态的JVM对象存储方式，每个对象通常要增加12-16字节的额外开销

（toString、hashCode等方法），如对于一个270MB的电商的商品表数据，使用这种方式读入内存，要使用970MB左右的内存空间（通常是2~5倍于原生数据空间）。

另外，使用这种方式，每个数据记录产生一个JVM对象，如果是大小为200GB的数据记录，

堆栈将产生1.6亿个对象，这么多的对象，对于GC来说，可能要消耗几分钟的时间来处理

（JVM的垃圾收集时间与堆栈中的对象数量呈线性相关。显然这种内存存储方式对于基于内存计算的spark来说，很昂贵也负担不起）

SparkSql的存储方式：对于**内存列存储**来说，将所有原生数据类型的**列采用原生数组来存储**，将Hive支持的复杂数据类型（如array、map等）先序化后并接成一个字节数组来存储。

此外，基于列存储，每列数据都是同质的，所以可以降低数据类型转换的CPU消耗。此外，

可以采用高效的压缩算法来压缩，是的数据更少。比如针对二元数据列，可以用字节编码压缩来实现（010101）

这样，每个列创建一个JVM对象，**从而可以快速的GC和紧凑的数据存储**；额外的，还可以使用低廉CPU开销的高效压缩方法（如字典编码、行长度编码等压缩方法）降低内存开销；更有趣的是，对于分析查询中频繁使用的聚合特定列，性能会得到很大的提高，原因就是这些列的数据放在一起，更容易读入内存进行计算。

SparkSQL入门

2018年2月10日 16:14

概述

SparkSql将RDD封装成一个DataFrame对象，这个对象类似于关系型数据库中的表。

创建DataFrame对象

DataFrame就相当于数据库的一张表。它是个只读的表，不能在运算过程再往里加元素。

RDD.toDF("列名")

```
scala> val rdd = sc.parallelize(List(1,2,3,4,5,6))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:21
```

```
scala> rdd.toDF("id")
res0: org.apache.spark.sql.DataFrame = [id: int]
```

```
scala> res0.show#默认只显示20条数据
```

```
+---+
```

```
| id|
```

```
+---+
```

```
| 1|
```

```
| 2|
```

```
| 3|
```

```
| 4|
```

```
| 5|
```

```
| 6|
```

```
+---+
```

```
scala> res0.printSchema #查看列的类型等属性
```

```
root
```

```
-- id: integer (nullable = true)
```

创建多列DataFrame对象

DataFrame就相当于数据库的一张表。

```
scala> sc.parallelize(List( (1,"beijing"),(2,"shanghai") ) )
```

```
res3: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[5] at parallelize at
<console>:22
```

```
scala> res3.toDF("id","name")
```

```
res4: org.apache.spark.sql.DataFrame = [id: int, name: string]
```

```
scala> res4.show
```

```
+---+-----+
| id|  name|
+---+-----+
|  1|beijing|
|  2|shanghai|
+---+-----+
```

例如3列的

```
scala> sc.parallelize(List( (1,"beijing",100780),(2,"shanghai",560090),(3,"xi'an",600329)))
```

```
res6: org.apache.spark.rdd.RDD[(Int, String, Int)] = ParallelCollectionRDD[10] at
```

```
parallelize at <console>:22
```

```
scala> res6.toDF("id","name","postcode")
```

```
res7: org.apache.spark.sql.DataFrame = [id: int, name: string, postcode: int]
```

```
scala> res7.show
```

```
+---+-----+-----+
| id|  name|postcode|
+---+-----+-----+
|  1|beijing| 100780|
|  2|shanghai| 560090|
|  3| xi'an| 600329|
+---+-----+-----+
```

可以看出，需要构建几列，tuple就有几个内容。

由外部文件构造DataFrame对象

1) txt文件

txt文件不能直接转换成，先利用RDD转换为tuple。然后toDF()转换为DataFrame。

```
scala> val rdd = sc.textFile("/root/words.txt")
```

```
.map( x => (x,1) )
.reduceByKey( (x,y) => x+y )
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[18] at reduceByKey at
<console>:21
```

```
scala> rdd.toDF("word","count")
res9: org.apache.spark.sql.DataFrame = [word: string, count: int]
```

```
scala> res9.show
```

```
+-----+-----+
| word|count|
+-----+-----+
| spark|   3|
| hive|   1|
|hadoop|  2|
| big|   2|
| scla|   1|
| data|   1|
+-----+-----+
```

2) json文件

📄 文件代码:

```
{"id":1, "name":"leo", "age":18}
{"id":2, "name":"jack", "age":19}
{"id":3, "name":"marry", "age":17}
```

📄 代码:

```
import org.apache.spark.sql.SQLContext
scala> val sqc=new SQLContext(sc)
scala> val tb4=sqc.read.json("/home/software/people.json")
scala> tb4.show
```

```

+---+---+---+
|age| id| name|
+---+---+---+
| 18|  1|  leo|
| 19|  2| jack|
| 17|  3|marry|
+---+---+---+

```

3) jdbc读取

实现步骤：

- 1) 将mysql 的驱动jar上传到spark的jars目录下
- 2) 重启spark服务
- 3) 进入spark客户端
- 4) 执行代码，比如在Mysql数据库下，有一个test库，在test库下有一张表为tabx

执行代码：

```

import org.apache.spark.sql.SQLContext
scala> val sqc = new SQLContext(sc);
scala> val prop = new java.util.Properties
scala> prop.put("user","root")
scala> prop.put("password","root")
scala> val
tabx=sqc.read.jdbc("jdbc:mysql://hadoop01:3306/zebra","D_H_HTTP_APPTYPE",prop)
scala> tabx.show
+---+---+
|id|name|
+---+---+
| 1|aaa|
| 2|bbb|
| 3|ccc|
| 1|ddd|
| 2|eee|
| 3|fff|
+---+---+

```

注：如果报权限不足，则进入mysql，执行：

```
grant all privileges on *.* to 'root'@'hadoop01' identified by 'root' with grant option;
```

然后执行：

```
flush privileges;
```

SparkSql基础语法一上

2018年2月11日 17:59

通过方法来使用

(1)查询

```
df.select("id","name").show();
```

(2)带条件的查询

```
df.select($"id",$"name").where($"name" === "bbb").show()
```

(3)排序查询

orderBy/sort(\$"列名") 升序排列

orderBy/sort(\$"列名".desc) 降序排列

orderBy/sort(\$"列1" , \$"列2".desc) 按两列排序

```
df.select($"id",$"name").orderBy($"name".desc).show
```

```
df.select($"id",$"name").sort($"name".desc).show
```

```
tabx.select($"id",$"name").sort($"id",$"name".desc).show
```

(4)分组查询

groupBy("列名", ...).max(列名) 求最大值

groupBy("列名", ...).min(列名) 求最小值

groupBy("列名", ...).avg(列名) 求平均值

groupBy("列名", ...).sum(列名) 求和

groupBy("列名", ...).count() 求个数

groupBy("列名", ...).agg 可以将多个方法进行聚合

```
scala>val rdd = sc.makeRDD(List((1,"a","bj",100),(2,"b","sh",80),(3,"c","gz",50),(4,"d","bj",45)));
```

```
scala>val df = rdd.toDF("id","name","addr","score");
```

```
scala>df.groupBy("addr").count().show()
```

```
scala>df.groupBy("addr").agg(max($"score"), min($"score"), count($"*")).show
```

(5)连接查询

```
scala>val dept=sc.parallelize(List((100,"caiwubu"),(200,"yanfabu"))).toDF("deptid","deptname")
```

```
scala>val emp=sc.parallelize(List((1,100,"zhang"),(2,200,"li"),(3,300,"wang"))).toDF("id","did","name")
```

```
scala>dept.join(emp,$"deptid" === $"did").show
```

```
scala>dept.join(emp,$"deptid" === $"did","left").show
```

左向外联接的结果集包括 LEFT OUTER子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。

```
scala>dept.join(emp,$"deptid" === $"did","right").show
```

```
scala>dept. join(emp, $"deptid" === $"did", "full"). show
```

```
scala>dept. join(emp, $"deptid" === $"did", "inner"). show
```

(6)执行运算

```
val df = sc.makeRDD(List(1,2,3,4,5)).toDF("num");
```

```
df.select($"num" * 100).show
```

(7)使用列表

```
val df = sc.makeRDD(List(("zhang",Array("bj","sh")),("li",Array("sz","gz")))).toDF("name","addrs")
```

```
df.selectExpr("name","addrs[0]").show
```

(8)使用结构体

```
{"name":"陈晨","address":{"city":"西安","street":"南二环甲字1号"}}
```

```
{"name":"娜娜","address":{"city":"西安","street":"南二环甲字2号"}}
```

```
val df = sqlContext.read.json("file:///root/work/users.json")
```

```
dfs.select("name","address.street").show
```

(9)其他

```
df.count//获取记录总数
```

```
val row = df.first()//获取第一条记录
```

```
val take=df.take(2) //获取前n条记录
```

```
val value = row.getString(1) //获取该行指定列的值
```

```
df.collect //获取当前df对象中的所有数据为一个Array 其实就是调用了df对象对应的底层的rdd的collect方法
```

SparkSql基础语法一下

2018年2月11日 18:03

通过sql语句来调用

(0)创建表

```
df.registerTempTable("tabName")
```

(1)查询

```
val sqc = new org.apache.spark.sql.SQLContext(sc);
```

```
val df =
```

```
sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");
```

```
df.registerTempTable("stu");
```

```
sqc.sql("select * from stu").show()
```

(2)带条件的查询

```
val df =
```

```
sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");
```

```
df.registerTempTable("stu");
```

```
sqc.sql("select * from stu where addr = 'bj']").show()
```

(3)排序查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
```

```
val df =
```

```
sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");
```

```
df.registerTempTable("stu");
```

```
sqlContext.sql("select * from stu order by addr").show()
```



```
sqlContext.sql("select * from stu order by addr desc").show()
```

(4)分组查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
```

```
val df =
```

```
sc.makeRDD(List((1,"a","bj"),(2,"b","sh"),(3,"c","gz"),(4,"d","bj"),(5,"e","gz"))).toDF("id","name","addr");
```

```
df.registerTempTable("stu");
```

```
sqlContext.sql("select addr,count(*) from stu group by addr").show()
```

(5)连接查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
```

```
val dept=sc.parallelize(List((100,"财务部"),(200,"研发部"))).toDF("deptid","deptname")
```

```
val emp=sc.parallelize(List((1,100,"张财务"),(2,100,"李会计"),(3,300,"王艳发"))).toDF("id","did","name")
```

```
dept.registerTempTable("deptTab");
```

```
emp.registerTempTable("empTab");
```

```
sqlContext.sql("select deptname,name from dept inner join emp on dept.deptid = emp.did").show()
```

(6)执行运算

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
```

```
val df = sc.makeRDD(List(1,2,3,4,5)).toDF("num");
```

```
df.registerTempTable("tabx")
```

```
sqlContext.sql("select num * 100 from tabx").show();
```

(7)分页查询

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
```

```
val df = sc.makeRDD(List(1,2,3,4,5)).toDF("num");

df.registerTempTable("tabx")

sqlContext.sql("select * from tabx limit 3").show();
```

(8)查看表

```
sqlContext.sql("show tables").show
```

(9)类似hive方式的操作

```
scala>val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

scala>hiveContext.sql("create table if not exists zzz (key int, value string) row format delimited fields
terminated by '|'|")

scala>hiveContext.sql("load data local inpath 'file:///home/software/hdata.txt' into table zzz")

scala>hiveContext.sql("select key,value from zzz").show
```

(10)案例

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);

val df = sc.textFile("file:///root/work/words.txt").flatMap{_.split(" ")} .toDF("word")

df.registerTempTable("wordTab")

sqlContext.sql("select word,count(*) from wordTab group by word").show
```

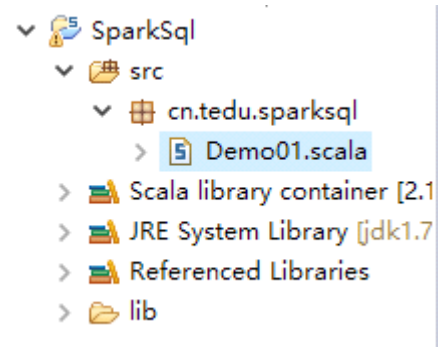
SparkSql API

2018年2月11日 18:05

通过api使用sparksql

实现步骤:

- 1) 打开scala IDE开发环境, 创建一个scala工程
- 2) 导入spark相关依赖jar包



- 3) 创建包路径以object类
- 4) 写代码

📄 代码示意:

```
package cn.tedu.sparksql
```

```
import org.apache.spark.SparkConf
```

```
import org.apache.spark.SparkContext
```

```
import org.apache.spark.sql.SQLContext
```

```
object Demo01 {
```

```
  def main(args: Array[String]): Unit = {
```

```
    val conf=new SparkConf().setMaster("spark://hadoop01:7077").setAppName("sqlDemo01");
```

```

val sc=new SparkContext(conf)

val sqlContext=new SQLContext(sc)


val rdd=sc.makeRDD(List((1,"zhang"),(2,"li"),(3,"wang")))


import sqlContext.implicits._

val df=rdd.toDF("id","name")

df.registerTempTable("tabx")


val df2=sqlContext.sql("select * from tabx order by name");

val rdd2=df2.toJavaRDD;

//将结果输出到linux的本地目录下，当然，也可以输出到HDFS上

rdd2.saveAsTextFile("file:///home/software/result");

}

}

```

5) 打jar包，并上传到linux虚拟机上

6) 在spark的bin目录下

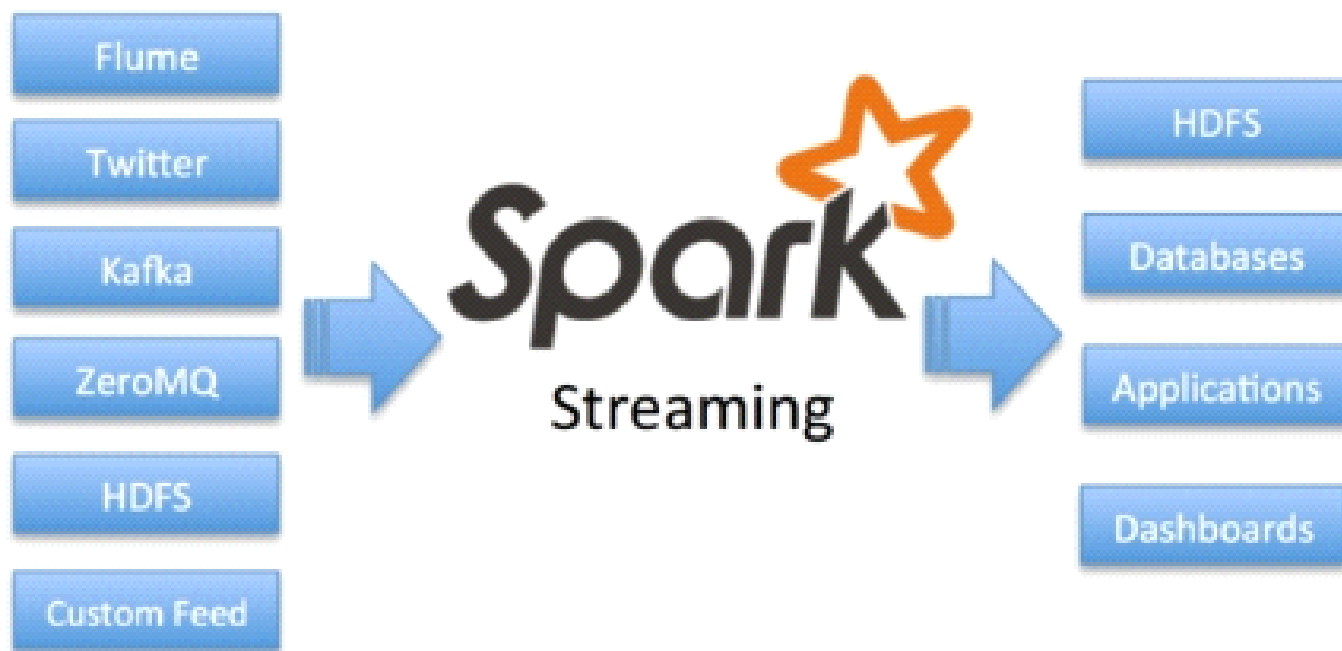
执行：sh spark-submit --class cn.tedu.sparksql.Demo01 ./sqlDemo01.jar

7) 最后检验

SparkStreaming介绍

2018年2月11日 18:08

概述

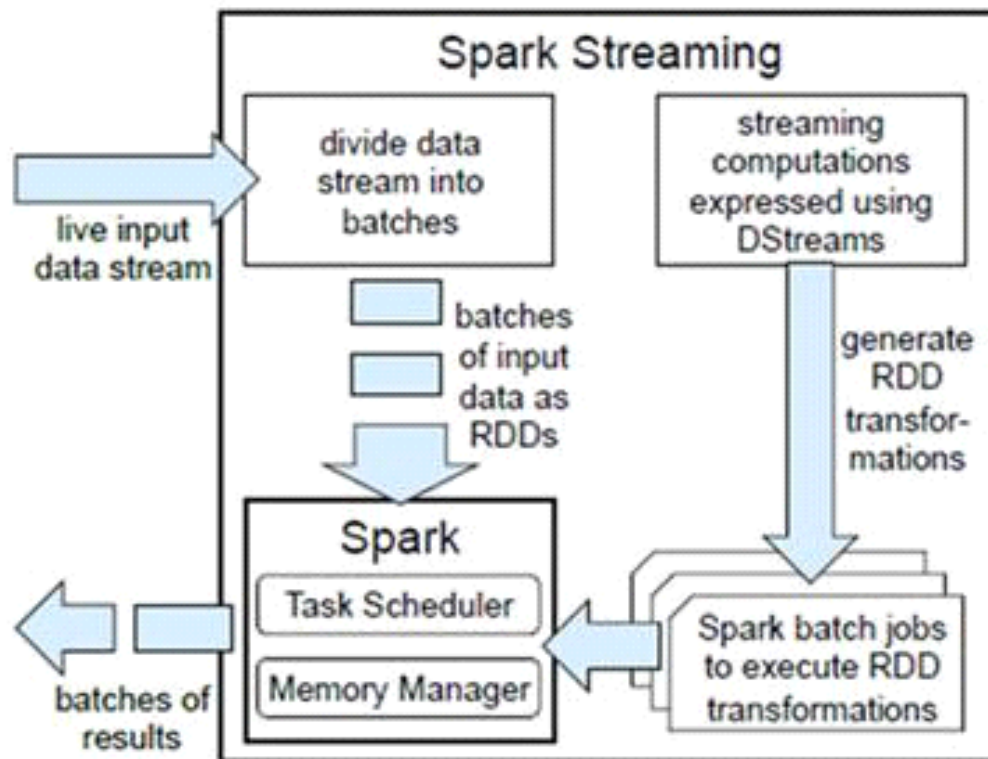


Spark Streaming是一种构建在Spark上的**实时计算**框架，它扩展了Spark处理大规模**流式数据**的能力，以**吞吐量高**和**容错能力强**著称。



架构设计

SparkStreaming是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如Kafka、Flume、Twitter、ZeroMQ和TCP 套接字）进行类似Map、Reduce和Join等复杂操作，并将结果保存到外部文件系统、数据库或应用到实时仪表盘。



Spark Streaming是将流式计算**分解成一系列短小的批处理作业**，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据**DStream**（Discretized-离散化Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformations操作变为针对Spark中对RDD的Transformations操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。

对DStream的处理，每个DStream都要按照数据流到达的**先后顺序依次**进行处理。即

SparkStreaming天然确保了数据处理的顺序性。

这样使所有的批处理具有了一个顺序的特性，其本质是**转换成RDD的血缘关系**。所以，

SparkStreaming对数据天然具有容错性保证。

为了提高SparkStreaming的工作效率，你应该合理的配置批的时间间隔，最好能够实现上一个批处理完某个算子，下一个批子刚好到来。

入门基础案例

2018年2月11日 22:41

WordCount案例

□ 案例一：

```
import org.apache.spark.streaming._

val ssc = new StreamingContext(sc,Seconds(5));

val lines = ssc.textFileStream("file:///home/software/stream");

//val lines = ssc.textFileStream("hdfs://hadoop01:9000/wordcount");

val words = lines.flatMap(_.split(" "));

val wordCounts = words.map(_._1).reduceByKey(_+_);

wordCounts.print();

ssc.start();
```

基本概念

1. StreamingContext

StreamingContext是Spark Streaming编程的最基本环境对象，就像Spark编程中的SparkContext一样。StreamingContext提供最基本功能入口，包括从各途径创建最基本的对象DStream（就像Spark编程中的RDD）。

创建StreamingContext的方法很简单，生成一个SparkConf实例，设置程序名，指定运行周期（示例中是5秒），这样就可以了：

```
val conf = new SparkConf().setAppName("SparkStreamingWordCount")

val sc=new SparkContext(conf)

val ssc = new StreamingContext(sc, Seconds(5))
```

运行周期为5秒，表示流式计算每间隔5秒执行一次。这个时间的设置需要综合考虑程序的延时需求和集群的工作负载，应该大于每次的运行时间。

StreamingContext还可以从一个现存的org.apache.spark.SparkContext创建而来，并保持关联，比如上面示例

中的创建方法：

```
val ssc = new StreamingContext(sc, Seconds(5))
```

StreamingContext创建好之后，还需要下面这几步来实现一个完整的Spark流式计算：

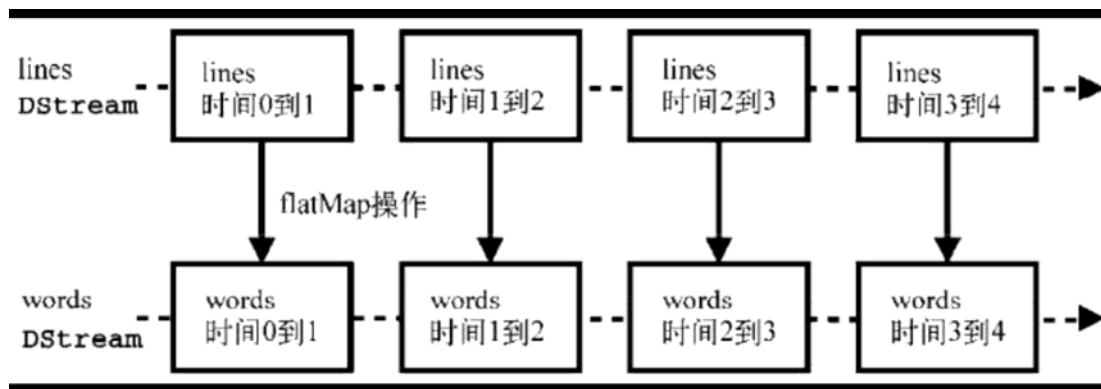
- (1) 创建一个输入DStream，用于接收数据；
- (2) 使用作用于DStream上的**Transformation**和**Output操作**来定义流式计算（Spark程序是使用Transformation和Action操作）；
- (3) 启动计算，使用streamingContext.start();
- (4) 等待计算结束（人为或错误），使用streamingContext.awaitTermination();
- (5) 也可以手工结束计算，使用streamingContext.stop()。

2. DStream抽象

DStream（discretized stream）是Spark Streaming的核心抽象，类似于RDD在Spark编程中的地位。DStream表示连续的数据流，要么是从数据源接收到的输入数据流，要求是经过计算产生的新数据流。DStream的内部是一个RDD序列，每个RDD对应一个计算周期。比如，在上面的WordCount示例中，**每5秒一个周期，那么每5秒的数据都分别对应一个RDD**，如图所示，图中的时间点1、2、3、4代表连续的时间周期。



所有应用在DStream上的操作，都会被映射为对DStream内部的RDD上的操作，比如上面的WordCount示例中对lines DStream的flatMap操作，如下图



RDD操作将由Spark核心来调度执行，但DStream屏蔽了这些细节，给开发者更简洁的编程体验。当然，我们也可以直接对DStream内部的RDD进行操作（后面会讲到）。

□ 案例二：

经过测试，案例一代码确实可以监控指定的文件夹处理其中产生的新的文件

但数据在每个新的周期到来后，都会重新进行计算

而如果需要对历史数据进行累计处理 该怎么做呢？

SparkStreaming提供了**checkPoint**机制，首先需要设置一个检查点目录，在这个目录，存储了历史周期数据。通过在临时文件中存储中间数据 为历史数据累计处理提供了可能性

```
import org.apache.spark.streaming._

val ssc = new StreamingContext(sc,Seconds(5));

ssc.checkpoint("file:///home/software/chk");

val lines = ssc.textFileStream("file:///home/software/stream");

val result= lines.flatMap(_.split(" ")).map((_,1)).updateStateByKey{(seq, op:Option[Int]) => { Some(seq.sum
+op.getOrElse(0)) }}

result.print();

ssc.start();
```

updateStateByKey 方法说明：

1.seq:是一个序列，存的是某个key的历史数据

2.op:是一个值，是某个key当前的值

比如: (hello,1)

①seq里是空的, Some(1)=>Some (返回的是历史值的和+当前值)

②(hello,2), seq(1) op=2 Some(1+2)

③(hello,1), seq(1,2) op=1 Some(3+1)

□ 案例三:

但是这上面的例子里所有的数据不停的累计 一直累计下去

很多的时候我们要的也不是这样的效果 **我们希望能够每隔一段时间重新统计下一段时间的数据**, 并且能够对设置的批时间进行更细粒度的控制, 这样的功能可以通过滑动窗口的方式来实现。

在DStream中提供了如下的和滑动窗口相关的方法:

window(windowLength, slideInterval)

windowLength: 窗口长度

slideInterval: 滑动区间



reduceByWindow(func, windowLength, slideInterval)

可以通过以上机制改造案例

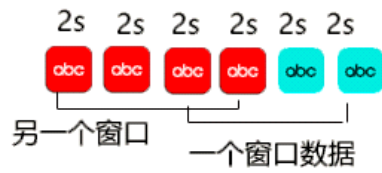
```
import org.apache.spark.streaming._

val ssc = new StreamingContext(sc, Seconds(1));
ssc.checkpoint("file:///home/software/chk");

val lines = ssc.textFileStream("file:///home/software/stream");
val result = lines.flatMap(_._split(" ")).map(_._1).reduceByKeyAndWindow((x:Int,y:Int)=>x+y, Seconds(5),
Seconds(5) ).print()

ssc.start();
```

SparkStreaming的窗口机制



窗口长度：可以设置窗口时间8s

滑动时间：8s

注意：窗口长度和滑动长度必须是batch size的整数倍

此外，使用窗口机制，必须要设定检查点目录

与Kafka整合

2018年10月25日 16:14

📖 代码示例:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.kafka.KafkaUtils

object Driver {

  def main(args: Array[String]): Unit = {

    //--启动线程数, 至少是两个。一个线程用于监听数据源, 其他线程用于消费或打印。至少是2个

    val conf=new SparkConf().setMaster("local[5]").setAppName("kafkainput")

    val sc=new SparkContext(conf)

    val ssc=new StreamingContext(sc,Seconds(5))
    ssc.checkpoint("d://check1801")

    //--连接kafka,并消费数据

    val zkHosts="192.168.150.137:2181,192.168.150.138:2181,192.168.150.139:2181"
    val groupName="gp1"
    //--Map的key是消费的主题名, value是消费的线程数。也可以消费多个主题, 比如:
    Map("parkx"->1,"enbook"->2)
    val topic=Map("parkx"->1)

    //--获取kafka的数据源
    //--SparkStreaming作为Kafka消费的数据源, 即从kafka中消费的偏移量(offset)存到
    zookeeper上

    val kafkaStream=KafkaUtils.createStream(ssc, zkHosts, groupName, topic).map{data=>data._2}

    val wordcount=kafkaStream.flatMap { line => line.split(" ") }.map { word=>(word,1) }
```

```
.updateStateByKey{(seq,op:Option[Int])=>Some(seq.sum+op.getOrElse(0))}  
  
wordcount.print()  
  
ssc.start()  
  
/--保持SparkStreaming线程一直开启  
ssc.awaitTermination()  
}  
}
```

与HBase整合

2018年10月25日 19:52

📖 写入HBase表代码示例:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.hadoop.hbase.mapreduce.TableOutputFormat
import org.apache.hadoop.mapreduce.Job
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.fs.shell.find.Result
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util.Bytes

object WriteDriver {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("local").setAppName("writeHbase")

    val sc=new SparkContext(conf)

    sc.hadoopConfiguration.set("hbase.zookeeper.quorum","hadoop01,hadoop02,hadoop03")
    sc.hadoopConfiguration.set("hbase.zookeeper.property.clientPort","2181")
    sc.hadoopConfiguration.set(TableOutputFormat.OUTPUT_TABLE,"tabx")

    val job=new Job(sc.hadoopConfiguration)

    job.setOutputKeyClass(classOf[ImmutableBytesWritable])
    job.setOutputValueClass(classOf[Result])
    job.setOutputFormatClass(classOf[TableOutputFormat[ImmutableBytesWritable]])

    val data=sc.makeRDD(Array("rk1,tom,23","rk2,rose,25","rk3,jary,30"))

    val hbaseRDD=data.map { line =>{
      val infos=line.split(",")
      val rowKey=infos(0)
      val name=infos(1)
```

```

val age=infos(2)

val put=new Put(Bytes.toBytes(rowKey))
put.add(Bytes.toBytes("cf1"),Bytes.toBytes("name"),Bytes.toBytes(name))
put.add(Bytes.toBytes("cf1"),Bytes.toBytes("age"),Bytes.toBytes(age))

(new ImmutableBytesWritable,put)
}}

//--将RDD数据存储进Hbase
hbaseRDD.saveAsNewAPIHadoopDataset(job.getConfiguration)

}
}

```

📖 读取HBase表代码：

```

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.util.Bytes

object ReadDriver {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setMaster("local").setAppName("readHbase")
val sc=new SparkContext(conf)

//--创建Hbase的环境变量参数
val hbaseConf=HBaseConfiguration.create()

hbaseConf.set("hbase.zookeeper.quorum","hadoop01,hadoop02,hadoop03")
hbaseConf.set("hbase.zookeeper.property.clientPort","2181")
hbaseConf.set(TableInputFormat.INPUT_TABLE,"tabx")

```

```

val resultRDD=sc.newAPIHadoopRDD(hbaseConf,classOf[TableInputFormat],
                                classOf[ImmutableBytesWritable],classOf[Result])

resultRDD.foreach{x=>{
  //--查询出来的结果集存在 (ImmutableBytesWritable, Result)第二个元素
  val result=x._2
  //--获取行键
  val rowKey=Bytes.toString(result.getRow)

  val name=Bytes.toString(result.getValue(Bytes.toBytes("cf1"),Bytes.toBytes("name")))
  val age=Bytes.toString(result.getValue(Bytes.toBytes("cf1"),Bytes.toBytes("age")))

  println(rowKey+"."+name+"."+age)
}}
}
}

```

过滤器代码:

```

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.hadoop.hbase.HBaseConfiguration
import org.datanucleus.store.types.backed.Set
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.hadoop.hbase.client.Scan
import org.apache.hadoop.hbase.filter.RandomRowFilter
import org.apache.hadoop.hbase.util.Base64
import org.apache.hadoop.hbase.protobuf.ProtobufUtil
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.util.Bytes

object ReadDriver2 {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("readHbaseFilter")
    val sc=new SparkContext(conf)

    val hbaseConf=HBaseConfiguration.create()

```



```

hbaseConf.set("hbase.zookeeper.quorum","hadoop01,hadoop02,hadoop03")
hbaseConf.set("hbase.zookeeper.property.clientPort","2181")
hbaseConf.set(TableInputFormat.INPUT_TABLE,"tabx")

val scan=new Scan
scan.setFilter(new RandomRowFilter(0.5f))
//--设置scan对象, 让filter生效
hbaseConf.set(TableInputFormat.SCAN,
    Base64.encodeBytes(ProtobufUtil.toScan(scan).toByteArray))

val resultRDD=sc.newAPIHadoopRDD(hbaseConf,classOf[TableInputFormat],
    classOf[ImmutableBytesWritable],classOf[Result])

resultRDD.foreach{x=>{
    //--查询出来的结果集存在 (ImmutableBytesWritable, Result)第二个元素
    val result=x._2
    //--获取行键
    val rowKey=Bytes.toString(result.getRow)

    val name=Bytes.toString(result.getValue(Bytes.toBytes("cf1"),Bytes.toBytes("name")))
    val age=Bytes.toString(result.getValue(Bytes.toBytes("cf1"),Bytes.toBytes("age")))

    println(rowKey+":"+name+":"+age)

}}
}
}

```

Spark On Yarn搭建

2018年4月6日 13:24

实现步骤:

1) 搭建好Hadoop (版本, 2.7) 集群

2) 安装和配置scala (版本, 2.11)

上传解压scala-2.11.0.tgz—>配置 /etc/profile文件

配置示例:

```
#java env
JAVA_HOME=/home/software/jdk1.8
HADOOP_HOME=/home/software/hadoop-2.7.1
SCALA_HOME=/home/software/scala-2.11.0
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
PATH=$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$SCALA_HOME/bin:$PATH
export JAVA_HOME PATH CLASSPATH HADOOP_HOME SCALA_HOME
```

3) 在NodeManager节点 (01,02,03节点) 上安装和配置Spark

4) 进入Spark安装目录的Conf目录, 配置: spark-env.sh 文件

配置示例:

```
export JAVA_HOME=/home/software/jdk1.8
export SCALA_HOME=/home/software/scala-2.11.0
export HADOOP_HOME=/home/software/hadoop-2.7.1
export HADOOP_CONF_DIR=/home/software/hadoop-2.7.1/etc/hadoop
```

5) 配置: slaves文件

配置示例:

```
hadoop01
hadoop02
hadoop03
```

6) 在HDFS上, 创建一个目录, 用来存放 spark的依赖jar包

执行: `hadoop fs -mkdir /spark_jars`

7) 进入spark 安装目录的jars目录,

执行: `hadoop fs -put ./ * /spark_jars`

8) 进入spark安装目录的 conf目录, 配置: spark-defaults.conf 文件

配置示例:

```
spark.yarn.jars=hdfs://hadoop02:9000/spark_jars/*
```

9) 至此, 完成Spark-Yarn的配置。注意: 如果是用虚拟机搭建, 可能会

由于虚拟机内存过小而导致启动失败，比如内存资源过小，yarn会直接kill掉进程导致rpc连接失败。所以，我们还需要配置Hadoop的yarn-site.xml文件，加入如下两项配置：

yarn-site.xml配置示例:

```
<property>
<name>yarn.nodemanager.vmem-check-enabled</name>
<value>>false</value>
</property>

<property>
<name>yarn.nodemanager.pmem-check-enabled</name>
<value>>false</value>
</property>
```

10) 启动Hadoop的yarn，进入Hadoop安装目录的sbin目录

执行: sh start-yarn.sh

11) 启动spark shell, 进入Spark安装目录的bin目录

执行: sh spark-shell --master yarn-client

```
Warning: Master yarn-client is deprecated since 2.0. Please use master "yarn" with specified deploy mode instead.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/04/06 15:30:30 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using built-in
re applicable
18/04/06 15:36:26 WARN metastore.ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Spark context Web UI available at http://192.168.234.24:4040
Spark context available as 'sc' (master = yarn, app id = application_1523053812019_0001).
Spark session available as 'spark'.
Welcome to

  ____  __
 / ___/  / /
/ /   /  / /
/ /___/  / /
/_____/  / /
         / /
        / /
       / /
      / /
     / /
    / /
   / /
  / /
 / /
/ /


version 2.1.1

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_65)
Type in expressions to have them evaluated.
Type :help for more information
```

然后可以通过yarn控制台来验证

192.168.234.21:8088/cluster

搜索



All Applications

Logged in as: dr.who

Cluster

About

Nodes

Node Labels

Applications

NEW

NEW SAVING

SUBMITTED

ACCEPTED

RUNNING

FINISHED

FAILED

KILLED

Scheduler

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	1	0	5	9 GB	24 GB	0 B	5	24	0	3	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:8>

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1523053812019_0001	root	Spark chall	SPARK	default	Fri Apr 6 15:21:55	N/A	RUNNING	UNDEFINED		ApplicationMaster

至于spark的使用，和之前都是一样的。只不过资源的分配和管理是交给 yarn来控制了。

分区 spark01 的第 164 页

Spark各运行模式详解

2018年4月6日 16:18

一、测试或实验性质的本地运行模式（单机）

该模式被称为Local[N]模式，是用单机的多个线程来模拟Spark分布式计算，通常用来验证开发出来的应用程序逻辑上有没有问题。

其中N代表可以使用N个线程，每个线程拥有一个core。如果不指定N，则默认是1个线程（该线程有1个core）。

指令示例：

- 1) spark-shell --master local 效果是一样的
- 2) spark-shell --master local[4] 代表会有4个线程（每个线程一个core）来并发执行应用程序。

运行该模式非常简单，只需要把Spark的安装包解压后，改一些常用的配置即可使用，而不用启动Spark的Master、Worker守护进程（只有集群的Standalone方式时，才需要这两个角色），也不用启动Hadoop的各服务（除非你要用到HDFS），这是和其他模式的区别，要记住才能理解。

二、测试或实验性质的本地伪集群运行模式（单机模拟集群）

这种运行模式，和Local[N]很像，不同的是，它会在单机启动多个进程来模拟集群下的分布式场景，而不像Local[N]这种多个线程只能在一个进程下委屈求全的共享资源。通常也是用来验证开发出来的应用程序逻辑上有没有问题，或者想使用Spark的计算框架而没有太多资源。

📖 指令示例:

1) `spark-shell --master local-cluster[2, 3, 1024]`

用法是：提交应用程序时使用`local-cluster[x,y,z]`参数：x代表要生成的executor数，y和z分别代表每个executor所拥有的core和memory数。

上面这条命令代表会使用2个executor进程，每个进程分配3个core和1G的内存，来运行应用程序。

该模式依然非常简单，只需要把Spark的安装包解压后，改一些常用的配置即可使用。而不用启动Spark的Master、Worker守护进程(只有集群的standalone方式时，才需要这两个角色)，也不用启动Hadoop的各服务（除非你要用到HDFS），这是和其他模式的区别哦，要记住才能理解。

三、Spark自带Cluster Manager的Standalone Client模式（集群）

和单机运行的模式不同，这里必须在执行应用程序前，先启动Spark的Master和Worker守护进程。不用启动Hadoop服务，除非你用到了HDFS的内容。

可以在想要做为Master的节点上用`start-all.sh`一条命令即可

这种运行模式，可以使用Spark的8080 web ui来观察资源和应用程序的执行情况了。

用如下命令提交应用程序

📖 指令示例：

1) `spark-shell --master spark://wl1:7077`

或者

2) `spark-shell --master spark://wl1:7077 --deploy-mode client`

四、spark自带cluster manager的standalone cluster模式（集群）

这种运行模式和上面第3个还是有很大的区别的。使用如下命令执行应用程序

📖 指令示例：

1) `spark-submit --master spark://wl1:6066 --deploy-mode cluster`

五、基于YARN的Resource Manager的Client模式（集群）

现在越来越多的场景，都是Spark跑在Hadoop集群中，所以为了做到资源能够均衡调度，会使用YARN来做为Spark的Cluster Manager，来为Spark的应用程序分配资源。

在执行Spark应用程序前，要启动Hadoop的各种服务。由于已经有了资源管理器，所以不需要启动Spark的Master、Worker守护进程。

使用如下命令执行应用程序

☐☐ **指令示例：**

1) spark-shell --master yarn

或者

2) spark-shell --master yarn --deploy-mode client

六、基于YARN的Resource Manager的Cluster模式（集群）

使用如下命令执行应用程序

☐☐ **指令示例：**

1) spark-shell --master yarn --deploy-mode cluster

此外，还有Spark On Mesos模式，对这部分感兴趣的同学自行查询了解。

可以参阅：

<http://ifeve.com/spark-mesos-spark/>

Scala练习

2017年12月11日 22:31

1.对"Hello"和"World"进行拉链操作，会产生什么结果？

```
Vector((H,W), ( e,o), (l,r), (l,l), (o,d))
```

2.编写函数values(fun:(Int)=>Int,low:Int,high:Int),该函数输出一个集合，对应给定区间内给定函数的输入和输出。比如，values(x=>x*x,-5,5)应该产生一个对偶的集合

```
def f1(f:(Int)=>Int,start:Int,end:Int)={  
  
  var arr = ListBuffer[(Int,Int)]()  
  
  for(num<-start to end){  
  
    arr.append((num,f(num)))  
  
  }  
  
  arr  
  
}  
  
f1(x=>x*x,-5,5)
```

3.如何用reduceLeft得到数组中的最大元素？

```
val arr = Array(3,2,6,8,4,6,9,3,6,7,1,2)  
  
print(arr.reduceLeft((a,b)=>if (a>b) a else b))
```

4.用to和reduceLeft实现阶乘函数

```
println(1 to 4 reduceLeft(_ * _))
```

5.编写函数largest(fun:(Int)=>Int,inputs:Seq[Int]),输出在给定输入序列中给定函数的最大值。举

例来说, `largest(x=>10*x-x*x,1 to 10)`应该返回25

1 2 3 4 10

```
def largest(fun:(Int)=>Int,inputs:Seq[Int])={  
    val s = inputs.reduceLeft((a,b)=>if (fun(a) > fun(b)) a else b)  
    fun(s)  
}
```

```
println(largest(x=>10*x-x*x,1 to 10))
```

6.修改前一个函数, 返回最大的输出对应的输入。举例来说,`largestAt(fun:(Int)=>Int,inputs:Seq[Int])`应该返回5

```
def largest(fun:(Int)=>Int,inputs:Seq[Int])={  
    inputs.reduceLeft((a,b)=>if (fun(a) > fun(b)) a else b)  
}
```

```
println(largest(x=>10*x-x*x,1 to 10))
```

7.给定一个整数数组, 产出一个新的数组, 包含原数组中的所有正值, 以原有顺序排列, 之后的元素是所有零或负值, 以原有顺序排列。

比如: `Array(1, -2, 0, -3, 0, 4, 5)`, 处理后的结果是: `1 4 5 0 0 -2 -3`

```
def main(args: Array[String]) {  
    val a = Array(1, -2, 0, -3, 0, 4, 5);
```

```
val b = sigNumArr(a);

b.foreach(println);

}

def sigNumArr(arr : Array[Int]) = {

    val buf = new ArrayBuffer[Int]();

    buf += (for (i <- arr if i > 0) yield i)

    buf += (for (i <- arr if i == 0) yield i)

    buf += (for (i <- arr if i < 0) yield i)


    buf.toArray

}
```