深入理解Spark 2.1 Core （一）：RDD的原理与源码分析

版权声明：本文为博主原创文章，转载请附上原文地址。 http://blog.csdn.net/u011239443/article/details/53894611

本文链接：http://blog.csdn.net/u011239443/article/details/53894611
该论文来自Berkeley实验室，英文标题为： Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing。下面的翻译，我是基于博文http://shiyanjun.cn/archives/744.html翻译基础上进行优化、修改、补充注释和源码分析。如果翻译措辞或逻辑有误，欢迎批评指正。

摘要

本文提出了分布式内存抽象的概念——弹性分布式数据集（RDD，Resilient Distributed Datasets），它具备像MapReduce等数据流模型的容错特性，并且允许开发人员在大型集群上执行基于内存的计算。现有的数据流系统对两种应用的处理并不高效：一是迭代式算法，这在图应用和机器学习领域很常见；二是交互式数据挖掘工具。这两种情况下，将数据保存在内存中能够极大地提高性能。为了有效地实现容错，RDD提供了一种高度受限的共享内存，即RDD是只读的，并且只能通过其他RDD上的批量操作来创建（***注：还可以由外部存储系数据集创建，如HDFS***）。尽管如此，RDD仍然足以表示很多类型的计算，包括MapReduce和专用的迭代编程模型（如Pregel）等。我们实现的RDD在迭代计算方面比Hadoop快20多倍，同时还可以在5-7秒内交互式地查询1TB数据集。

1.引言

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。从稳定的物理存储（如分布式文件系统）(***注：即磁盘***)中加载记录，记录被传入由一组确定性操作构成的DAG，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。我们就是针对这些不太适合非循环模型的应用，它们的特点是在多个并行操作之间重用工作数据集。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）(***注：有许多机器学习算法需要将这次迭代权值调优后的结果数据集作为下次迭代的输入，而使用MapReduce计算框架经过一次Reduce操作后输出数据结果写回磁盘，速度大大的降低了***)；（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，这带来较大的开销。

我们提出了一种分布式的内存抽象，称为弹性分布式数据集（RDD，Resilient Distributed Datasets）。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。

RDD提供了一种高度受限的共享内存模型，即RDD是只读的记录分区的集合，只能通过在其他RDD执行确定的转换操作（如map、join和group by）而创建，然而这些限制使得实现容错的开销很低。与分布式共享内存系统需要付出高昂代价的检查点和回滚机制不同，RDD通过Lineage来重建丢失的分区：一个RDD中包含了如何从其他RDD衍生所必需的相关信息，从而不需要检查点操作就可以重构丢失的数据分区。尽管RDD不是一个通用的共享内存抽象，但却具备了良好的描述能力、可伸缩性和可靠性，但却能够广泛适用于数据并行类应用。

第一个指出非循环数据流存在不足的并非是我们，例如，Google的Pregel[21]，是一种专门用于迭代式图算法的编程模型；Twister[13]和HaLoop[8]，是两种典型的迭代式MapReduce模型。但是，对于一些特定类型的应用，这些系统提供了一个受限的通信模型。相比之下，RDD则为基于工作集的应用提供了更为通用的抽象，用户可以对中间结果进行显式的命名和物化，控制其分区，还能执行用户选择的特定操作（而不是在运行时去循环执行一系列MapReduce步骤）。RDD可以用来描述Pregel、迭代式MapReduce，以及这两种模型无法描述的其他应用，如交互式数据挖掘工具（用户将数据集装入内存，然后执行ad-hoc查询）。

Spark是我们实现的RDD系统，在我们内部能够被用于开发多种并行应用。Spark采用Scala语言[5]实现，提供类似于DryadLINQ的集成语言编程接口[34]，使用户可以非常容易地编写并行任务。此外，随着Scala新版本解释器的完善，Spark还能够用于交互式查询大数据集。我们相信Spark会是第一个能够使用有效、通用编程语言，并在集群上对大数据集进行交互式分析的系统。

我们通过微基准和用户应用程序来评估RDD。实验表明，在处理迭代式应用上Spark比Hadoop快高达20多倍，计算数据分析类报表的性能提高了40多倍，同时能够在5-7秒的延时内交互式扫描1TB数据集。此外，我们还在Spark之上实现了Pregel和HaLoop编程模型（包括其位置优化策略），以库的形式实现（分别使用了100和200行Scala代码）。最后，利用RDD内在的确定性特性，我们还创建了一种Spark调试工具rddbg，允许用户在任务期间利用Lineage重建RDD，然后像传统调试器那样重新执行任务。

本文首先在第2部分介绍了RDD的概念，然后第3部分描述Spark API，第4部分解释如何使用RDD表示几种并行应用（包括Pregel和HaLoop），第5部分讨论Spark中RDD的表示方法以及任务调度器，第6部分描述具体实现和rddbg，第7部分对RDD进行评估，第8部分给出了相关研究工作，最后第9部分总结。

2.弹性分布式数据集（RDD）

本部分描述RDD和编程模型。首先讨论设计目标（2.1），然后定义RDD（2.2），讨论Spark的编程模型（2.3），并给出一个示例（2.4），最后对比RDD与分布式共享内存（2.5）。

2.1 目标和概述

我们的目标是为基于工作集的应用（即多个并行操作重用中间结果的这类应用）提供抽象，同时保持MapReduce及其相关模型的优势特性：即自动容错、位置感知性调度和可伸缩性。RDD比数据流模型更易于编程，同时基于工作集的计算也具有良好的描述能力。

在这些特性中，最难实现的是容错性。一般来说，分布式数据集的容错性有两种方式：即***数据检查点***和***记录数据的更新***。我们面向的是大规模数据分析，数据检查点操作成本很高：需要通过数据中心的网络连接在机器之间复制庞大的数据集，而网络带宽往往比内存带宽低得多，同时还需要消耗更多的存储资源（在内存中复制数据可以减少需要缓存的数据量，而存储到磁盘则会拖慢应用程序）。所以，我们选择记录更新的方式。但是，如果更新太多，那么记录更新成本也不低。因此，***RDD只支持粗粒度转换，即在大量记录上执行的单个操作。将创建RDD的一系列转换记录下来（即Lineage），以便恢复丢失的分区。***

虽然只支持粗粒度转换限制了编程模型，但我们发现RDD仍然可以很好地适用于很多应用，特别是支持数据并行的批量分析应用，包括数据挖掘、机器学习、图算法等，因为这些程序通常都会在很多记录上执行相同的操作。RDD不太适合那些异步更新共享状态的应用，例如并行web爬虫。因此，我们的目标是为大多数分析型应用提供有效的编程模型，而其他类型的应用交给专门的系统。

2.2 RDD抽象

RDD是只读的、分区记录的集合。RDD只能基于在稳定物理存储中的数据集和其他已有的RDD上执行确定性操作来创建。这些确定性操作称之为转换，如map、filter、groupBy、join（转换不是程序开发人员在RDD上执行的操作（***注：这句话的意思可能是，转换操作并不会触发RDD真正的action。由于惰性执行，当进行action操作的时候，才会回溯去执行前面的转换操作***））。

RDD不需要物化。RDD含有如何从其他RDD衍生（即计算）出本RDD的相关信息（即Lineage），据此可以从物理存储的数据计算出相应的RDD分区。

2.3 编程模型

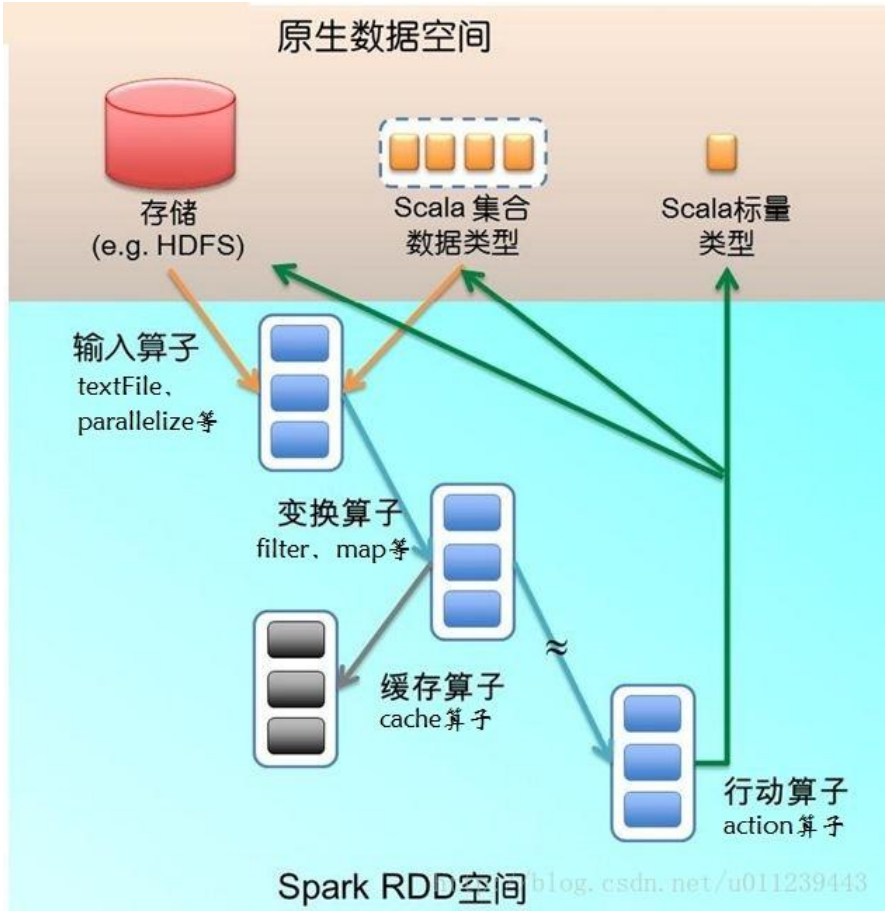
在Spark中，RDD被表示为对象，通过这些对象上的方法（或函数）调用转换。

定义RDD之后，程序员就可以在动作（***注：即action操作***）中使用RDD了。动作是向应用程序返回值，或向存储系统导出数据的那些操作，例如，count（返回RDD中的元素个数），collect（返回元素本身），save（将RDD输出到存储系统）。在Spark中，只有在动作第一次使用RDD时，才会计算RDD（即延迟计算）。这样在构建RDD的时候，运行时通过管道的方式传输多个转换。

程序员还可以从两个方面控制RDD，即缓存和分区。用户可以请求将RDD缓存，这样运行时将已经计算好的RDD分区存储起来，以加速后期的重用。缓存的RDD一般存储在内存中，但如果内存不够，可以写到磁盘上。

另一方面，RDD还允许用户根据关键字（key）指定分区顺序，这是一个可选的功能。目前支持哈希分区和范围分区。例如，应用程序请求将两个RDD按照同样的哈希分区方式进行分区（将同一机器上具有相同关键字的记录放在一个分区），以加速它们之间的join操作。在Pregel和HaLoop中，多次迭代之间采用一致性的分区置换策略进行优化，我们同样也允许用户指定这种优化。

（***注:***



）

2.4 示例：控制台日志挖掘

本部分我们通过一个具体示例来阐述RDD。假定有一个大型网站出错，操作员想要检查Hadoop文件系统（HDFS）中的日志文件（TB级大小）来找出原因。通过使用Spark，操作员只需将日志中的错误信息装载到一组节点的内存中，然后执行交互式查询。首先，需要在Spark解释器中输入如下Scala代码：

```
1     lines = spark.textFile("hdfs://...")
2     errors = lines.filter(_.startsWith("ERROR"))
3     errors.cache()
```

第1行从HDFS文件定义了一个RDD（即一个文本行集合），第2行获得一个过滤后的RDD，第3行请求将errors缓存起来。注意在Scala语法中filter的参数是一个闭包(***什么是闭包？ https://zhuanlan.zhihu.com/p/21346046***)。

这时集群还没有开始执行任何任务。但是，用户已经可以在这个RDD上执行对应的动作，例如统计错误消息的数目：

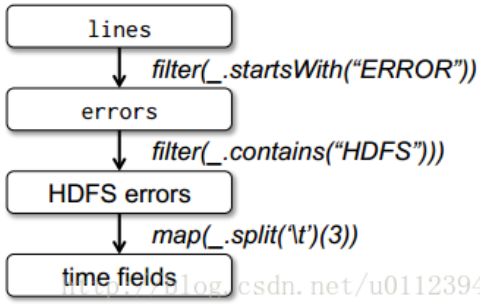
```
1 errors.count()
```

用户还可以在RDD上执行更多的转换操作，并使用转换结果，如：

```
1 // Count errors mentioning MySQL:
2 errors.filter(_.contains("MySQL")).count()
3 // Return the time fields of errors mentioning
4 // HDFS as an array (assuming time is field
5 // number 3 in a tab-separated format):
6 errors.filter(_.contains("HDFS"))
7     .map(_.split('\t')(3))
8     .collect()
```

使用errors的第一个action运行以后，Spark会把errors的分区缓存在内存中，极大地加快了后续计算速度。注意，最初的RDD lines不会被缓存。因为错误信息可能只占原数据集的很小一部分（小到足以放入内存）。

最后，为了说明模型的容错性，图1给出了第3个查询的Lineage图。在lines RDD上执行filter操作，得到errors，然后再filter、map后得到新的RDD，在这个RDD上执行collect操作。Spark调度器以流水线的方式执行后两个转换，向拥有errors分区缓存的节点发送一组任务。此外，如果某个errors分区丢失，Spark只在相应的lines分区上执行filter操作来重建该errors分区。



2.5 RDD与分布式共享内存

为了进一步理解RDD是一种分布式的内存抽象，表1列出了RDD与分布式共享内存（DSM，Distributed Shared Memory）[24]的对比。在DSM系统中，应用可以向全局地址空间的任意位置进行读写操作。（注意这里的DSM，不仅指传统的共享内存系统，还包括那些通过分布式哈希表或分布式文件系统进行数据共享的系统，比如Piccolo[28]（***注:Spark生态系统中有一名为Alluxio的分布式内存文件系统，它通常可作为Spark和HDFS的中间层存在***））DSM是一种通用的抽象，但这种通用性同时也使得在商用集群上实现有效的容错性更加困难。

RDD与DSM主要区别在于，不仅可以通过批量转换创建（即"写"）RDD，还可以对任意内存位置读写。也就是说，RDD限制应用执行批量写操作，这样有利于实现有效的容错。特别地，RDD没有检查点开销，因为可以使用Lineage来恢复RDD。而且，失效时只需要重新计算丢失的那些RDD分区，可以在不同节点上并行执行，而不需要回滚整个程序。

表1 RDD与DSM对比

对比项目	RDD	分布式共享内存（DSM）
读	批量或细粒度操作	细粒度操作
写	批量转换操作	细粒度操作
一致性	不重要（RDD是不可更改的）	取决于应用程序或运行时
容错性	细粒度，低开销（使用Lineage）	需要检查点操作和程序回滚
落后任务的处理	任务备份	很难处理
任务安排	基于数据存放的位置自动实现	取决于应用程序（通过运行时实现透明性）
如果内存不够	与已有的数据流系统类似	性能较差
对比项目	RDD	分布式共享内存（DSM）

注意，通过备份任务的拷贝，RDD还可以处理落后任务（即运行很慢的节点），这点与MapReduce[12]类似。而DSM则难以实现备份任务，因为任务及其副本都需要读写同一个内存位置。

与DSM相比，RDD模型有两个好处。***第一，对于RDD中的批量操作，运行时将根据数据存放的位置来调度任务，从而提高性能。第二，对于基于扫描的操作，如果内存不足以缓存整个RDD，就进行部分缓存。把内存放不下的分区存储到磁盘上，此时性能与现有的数据流系统差不多。***

最后看一下读操作的粒度。RDD上的很多动作（如count和collect）都是批量读操作，即扫描整个数据集，可以将任务分配到距离数据最近的节点上。同时，RDD也支持细粒度操作，即在哈希或范围分区的RDD上执行关键字查找。

3. Spark编程接口

Spark用Scala[5]语言实现了RDD的API。Scala是一种基于JVM的静态类型、函数式、面向对象的语言。我们选择Scala是因为它简洁（特别适合交互式使用）、有效（因为是静态类型）。但是，RDD抽象并不局限于函数式语言，也可以使用其他语言来实现RDD，比如像Hadoop[2]那样用类表示用户函数。

要使用Spark，开发者需要编写一个driver程序，连接到集群以运行Worker，如图2所示。Driver定义了一个或多个RDD，并调用RDD上的动作。Worker是长时间运行的进程，将RDD分区以Java对象的形式缓存在内存中。

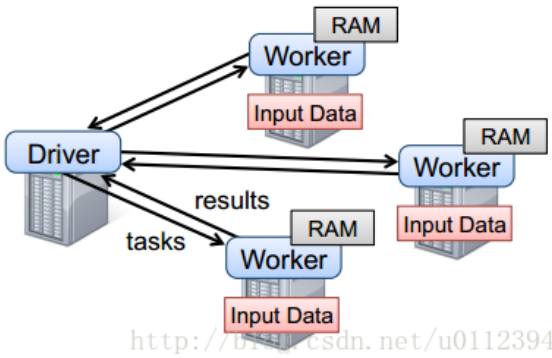


图2 Spark的运行时。用户的driver程序启动多个worker，worker从分布式文件系统中读取数据块，并将计算后的RDD分区缓存在内存中。

再看看2.4中的例子，用户执行RDD操作时会提供参数，比如map传递一个闭包（closure，函数式编程中的概念）。Scala将闭包表示为Java对象，如果传递的参数是闭包，则这些对象被序列化，通过网络传输到其他节点上进行装载。Scala将闭包内的变量保存为Java对象的字段。例如，var x = 5; rdd.map(_ + x) 这段代码将RDD中的每个元素加5。总的来说，Spark的语言集成类似于DryadLINQ。

RDD本身是静态类型对象，由参数指定其元素类型。例如，RDD[int]是一个整型RDD。不过，我们举的例子几乎都省略了这个类型参数，因为Scala支持类型推断。

虽然在概念上使用Scala实现RDD很简单，但还是要处理一些Scala闭包对象的反射问题。如何通过Scala解释器来使用Spark还需要更多工作，这点我们将在第6部分讨论。不管怎样，我们都不需要修改Scala编译器。

3.1 Spark中的RDD操作

表2列出了Spark中的RDD转换和动作。每个操作都给出了标识，其中方括号表示类型参数。前面说过转换是延迟操作，用于定义新的RDD；而动作启动计算操作，并向用户程序返回值或向外部存储写数据。

表3 Spark中支持的RDD转换和动作

转换	map(f : T) U) : RDD[T]) RDD[U] filter(f : T) Bool) : RDD[T]) RDD[T] flatMap(f : T) Seq[U]) : RDD[T]) RDD[U] sample(fraction : Float) : RDD[T]) RDD[T] (Deterministic sampling) groupByKey() : RDD[(K, V)]) RDD[(K, Seq[V])] reduceByKey(f : (V; V)) V) : RDD[(K, V)]) RDD[(K, V)] union() : (RDD[T]; RDD[T])) RDD[T] join() : (RDD[(K, V)]; RDD[(K, W)])) RDD[(K, (V, W))] cogroup() : (RDD[(K, V)]; RDD[(K, W)])) RDD[(K, (Seq[V], Seq[W]))] crossProduct() : (RDD[T]; RDD[U])) RDD[(T, U)] mapValues(f : V) W) : RDD[(K, V)]) RDD[(K, W)] (Preserves partitioning) sort(c : Comparator[K]) : RDD[(K, V)]) RDD[(K, V)] partitionBy(p : Partitioner[K]) : RDD[(K, V)]) RDD[(K, V)]
动作	count() : RDD[T]) Long collect() : RDD[T]) Seq[T] reduce(f : (T; T)) T) : RDD[T]) T lookup(k : K) : RDD[(K, V)]) Seq[V] (On hash/range partitioned RDDs) save(path : String) : Outputs RDD to a storage system, e.g., HDFS

注意，有些操作只对键值对可用，比如join。另外，函数名与Scala及其他函数式语言中的API匹配，例如map是一对一的映射，而flatMap是将每个输入映射为一个或多个输出（与MapReduce中的map类似）。

除了这些操作以外，用户还可以请求将RDD缓存起来。而且，用户还可以通过Partitioner类获取RDD的分区顺序，然后将另一个RDD按照同样的方式分区。有些操作会自动产生一个哈希或范围分区的RDD，像groupByKey，reduceByKey和sort等。

4. 应用程序示例

现在我们讲述如何使用RDD表示几种基于数据并行的应用。首先讨论一些迭代式机器学习应用（4.1），然后看看如何使用RDD描述几种已有的集群编程模型，即MapReduce（4.2），Pregel（4.3），和Hadoop（4.4）。最后讨论一下RDD不适合哪些应用（4.5）。

4.1 迭代式机器学习

很多机器学习算法都具有迭代特性，运行迭代优化方法来优化某个目标函数，例如梯度下降方法。如果这些算法的工作集能够放入内存，将极大地加速程序运行。而且，这些算法通常采用批量操作，例如映射和求和，这样更容易使用RDD来表示。

例如下面的程序是逻辑回归[15]的实现。逻辑回归是一种常见的分类算法，即寻找一个最佳分割两组点（即垃圾邮件和非垃圾邮件）的超平面w。算法采用梯度下降的方法：开始时w为随机值，在每一次迭代的过程中，对w的函数求和，然后朝着优化的方向移动w。

```
val points = spark.textFile(...)
     .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
     val gradient = points.map{ p =>
          p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
     }.reduce((a,b) => a+b)
     w -= gradient
}
```

首先定义一个名为points的缓存RDD，这是在文本文件上执行map转换之后得到的，即将每个文本行解析为一个Point对象。然后在points上反复执行map和reduce操作，每次迭代时通过对当前w的函数进行求和来计算梯度。7.1小节我们将看到这种在内存中缓存points的方式，比每次迭代都从磁盘文件装载数据并进行解析要快得多。

已经在Spark中实现的迭代式机器学习算法还有：kmeans（像逻辑回归一样每次迭代时执行一对map和reduce操作），期望最大化算法（EM，两个不同的map/reduce步骤交替执行），交替最小二乘矩阵分解和协同过滤算法。Chu等人提出迭代式MapReduce也可以用来实现常用的学习算法[11]。

4.2 使用RDD实现MapReduce

MapReduce模型[12]很容易使用RDD进行描述。假设有一个输入数据集（其元素类型为T），和两个函数myMap: T => List[(Ki, Vi)] 和myReduce: (Ki; List[Vi])) List[R]，代码如下：

```
data.flatMap(myMap)
    .groupByKey()
    .map((k, vs) => myReduce(k, vs))
```

如果任务包含combiner，则相应的代码为：

```
data.flatMap(myMap)
    .reduceByKey(myCombiner)
    .map((k, v) => myReduce(k, v))
```

ReduceByKey操作在mapper节点上执行部分聚集，与MapReduce的combiner类似。

4.3 使用RDD实现Pregel

略

4.4 使用RDD实现HaLoop

略

4.5 不适合使用RDD的应用

在2.1节我们讨论过，RDD适用于具有批量转换需求的应用，并且相同的操作作用于数据集的每一个元素上。在这种情况下，RDD能够记住每个转换操作，对应于Lineage图中的一个步骤，恢复丢失分区数据时不需要写日志记录大量数据。RDD不适合那些通过异步

细粒度地更新来共享状态的应用，例如Web应用中的存储系统，或者增量抓取和索引Web数据的系统，这样的应用更适合使用一些传统的方法，例如数据库、RAMCloud[26]、Percolator[27]和Piccolo[28]。我们的目标是，面向批量分析应用的这类特定系统，提供一种高效的编程模型，而不是一些异步应用程序。

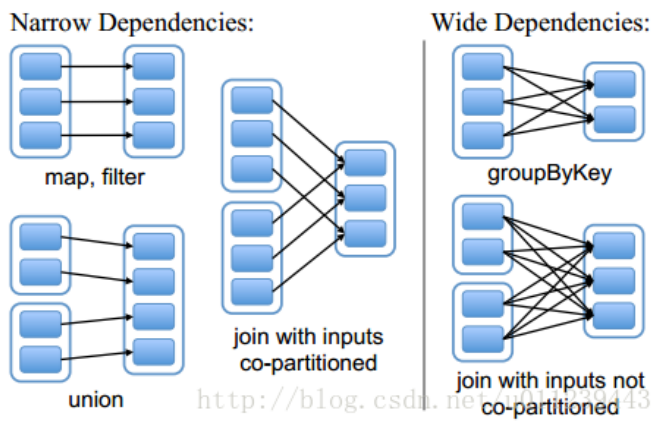
5. RDD的描述及作业调度

我们希望在不修改调度器的前提下，支持RDD上的各种转换操作，同时能够从这些转换获取Lineage信息。为此，我们为RDD设计了一组小型通用的内部接口。

简单地说，每个RDD都包含：（1）一组RDD分区（partition，即数据集的原子组成部分）；（2）对父RDD的一组依赖，这些依赖描述了RDD的Lineage；（3）一个函数，即在父RDD上执行何种计算；（4）元数据，描述分区模式和数据存放的位置。例如，一个表示HDFS文件的RDD包含：各个数据块的一个分区，并知道各个数据块放在哪些节点上。而且这个RDD上的map操作结果也具有同样的分区，map函数是在父数据上执行的。表3总结了RDD的内部接口。
表3 Spark中RDD的内部接口

操作	含义
partitions()	返回一组Partition对象
preferredLocations(p)	根据数据存放的位置，返回分区p在哪些节点访问更快
dependencies()	返回一组依赖
iterator(p, parentIters)	按照父分区的迭代器，逐个计算分区p的元素
partitioner()	返回RDD是否hash/range分区的元数据信息

设计接口的一个关键问题就是，如何表示RDD之间的依赖。我们发现RDD之间的依赖关系可以分为两类，即：（1）窄依赖（narrow dependencies）：子RDD的每个分区依赖于常数个父分区（即与数据规模无关）；（2）宽依赖（wide dependencies）：子RDD的每个分区依赖于所有父RDD分区。例如，map产生窄依赖，而join则是宽依赖（除非父RDD被哈希分区）。另一个例子见图5。



（注：我们可以这样认为：

> 窄依赖指的是：每个parent RDD 的 partition 最多被 child RDD的一个partition使用
> 宽依赖指的是：每个parent RDD 的 partition 被多个 child RDD的partition使用

窄依赖每个child RDD 的partition的生成操作都是可以并行的，而宽依赖则需要所有的parent partition shuffle结果得到后再进行。

下面我们来看下，我们来看下org.apache.spark.Dependency.scala的源码

抽象类Dependency：

```
abstract class Dependency[T] extends Serializable {
  def rdd: RDD[T]
}
```

Dependency有两个子类，一个子类为窄依赖：NarrowDependency；一个为宽依赖ShuffleDependency

NarrowDependency也是一个抽象类，它实现了getParents 重写了 rdd 函数，它有两个子类，一个是 OneToOneDependency，一个是 RangeDependency

```
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
  /**
   * Get the parent partitions for a child partition.
   * @param partitionId a partition of the child RDD
   * @return the partitions of the parent RDD that the child partition depends upon
   */
  def getParents(partitionId: Int): Seq[Int]

  override def rdd: RDD[T] = _rdd
}
```

OneToOneDependency，可以看到getParents实现很简单，就是传进一个partitionId: Int，再把partitionId放在List里面传出去，即去parent RDD 中取与该RDD 相同 partitionID的数据

```
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
  override def getParents(partitionId: Int): List[Int] = List(partitionId)
}
```

RangeDependency，用于union。与上面不同的是，这里我们要算出该位置，设某个parent RDD 从 inStart 开始的partition，逐个生成了 child RDD 从outStart 开始的partition，则计算方式为： partitionId - outStart + inStart

```
class RangeDependency[T](rdd: RDD[T], inStart: Int, outStart: Int, length: Int)
  extends NarrowDependency[T](rdd) {

  override def getParents(partitionId: Int): List[Int] = {
    if (partitionId >= outStart && partitionId < outStart + length) {
      List(partitionId - outStart + inStart)
    } else {
      Nil
    }
  }
}
```

ShuffleDependency，需要进行shuffle

```
class ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag](
    @transient private val _rdd: RDD[_ <: Product2[K, V]],
    val partitioner: Partitioner,
    val serializer: Serializer = SparkEnv.get.serializer,
    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[Product2[K, V]] = _rdd.asInstanceOf[RDD[Product2[K, V]]]

  private[spark] val keyClassName: String = reflect.classTag[K].runtimeClass.getName
  private[spark] val valueClassName: String = reflect.classTag[V].runtimeClass.getName
  // Note: It's possible that the combiner class tag is null, if the combineByKey
  // methods in PairRDDFunctions are used instead of combineByKeyWithClassTag.
  private[spark] val combinerClassName: Option[String] =
    Option(reflect.classTag[C]).map(_.runtimeClass.getName)
//获取shuffleID
  val shuffleId: Int = _rdd.context.newShuffleId()
//向注册shuffleManager注册Shuffle信息
  val shuffleHandle: ShuffleHandle = _rdd.context.env.shuffleManager.registerShuffle(
    shuffleId, _rdd.partitions.length, this)
```

```
23
24    _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
25  }
```

)
区分这两种依赖很有用。首先，窄依赖允许在一个集群节点上以流水线的方式（pipeline）计算所有父分区。例如，逐个元素地执行map、然后filter操作；而宽依赖则需要首先计算好所有父分区数据，然后在节点之间进行Shuffle，这与MapReduce类似。第二，窄依赖能够更有效地进行失效节点的恢复，即只需重新计算丢失RDD分区的父分区，而且不同节点之间可以并行计算；而对于一个宽依赖关系的Lineage图，单个节点失效可能导致这个RDD的所有祖先丢失部分分区，因而需要整体重新计算。

通过RDD接口，Spark只需要不超过20行代码实现便可以实现大多数转换。5.1小节给出了例子，然后我们讨论了怎样使用RDD接口进行调度（5.2），最后讨论一下基于RDD的程序何时需要数据检查点操作（5.3）。

5.2 Spark任务调度器

可见：http://blog.csdn.net/u011239443/article/details/53911902

5.3 检查点

尽管RDD中的Lineage信息可以用来故障恢复，但对于那些Lineage链较长的RDD来说，这种恢复可能很耗时。例如4.3小节中的Pregel任务，每次迭代的顶点状态和消息都跟前一次迭代有关，所以Lineage链很长。如果将Lineage链存到物理存储中，再定期对RDD执行检查点操作就很有效。

一般来说，Lineage链较长、宽依赖的RDD需要采用检查点机制。这种情况下，集群的节点故障可能导致每个父RDD的数据块丢失，因此需要全部重新计算[20]。将窄依赖的RDD数据存到物理存储中可以实现优化，例如前面4.1小节逻辑回归的例子，将数据点和不变的顶点状态存储起来，就不再需要检查点操作。

当前Spark版本提供检查点API，但由用户决定是否需要执行检查点操作。今后我们将实现自动检查点，根据成本效益分析确定RDD Lineage图中的最佳检查点位置。

值得注意的是，因为RDD是只读的，所以不需要任何一致性维护（例如写复制策略，分布式快照或者程序暂停等）带来的开销，后台执行检查点操作。
（***注：***
我们来阅读下org.apache.spark.rdd.ReliableCheckpointRDD中的def writePartitionToCheckpointFile和 def writeRDDToCheckpointDirectory：

writePartitionToCheckpointFile:把RDD一个Partition文件里面的数据写到一个Checkpoint文件里面

```
1   def writePartitionToCheckpointFile[T: ClassTag](
2       path: String,
3       broadcastedConf: Broadcast[SerializableConfiguration],
4       blockSize: Int = -1)(ctx: TaskContext, iterator: Iterator[T]) {
5     val env = SparkEnv.get
6     //获取Checkpoint文件输出路径
7     val outputDir = new Path(path)
8     val fs = outputDir.getFileSystem(broadcastedConf.value.value)
9
10    //根据partitionId 生成 checkpointFileName
11    val finalOutputName = ReliableCheckpointRDD.checkpointFileName(ctx.partitionId())
12    //拼接路径与文件名
13    val finalOutputPath = new Path(outputDir, finalOutputName)
14    //生成临时输出路径
15    val tempOutputPath =
16      new Path(outputDir, s".$finalOutputName-attempt-${ctx.attemptNumber()}")
17
18    if (fs.exists(tempOutputPath)) {
19      throw new IOException(s"Checkpoint failed: temporary path $tempOutputPath already exists")
20    }
21    //得到块大小，默认为64MB
22    val bufferSize = env.conf.getInt("spark.buffer.size", 65536)
23    //得到文件输出流
24    val fileOutputStream = if (blockSize < 0) {
```

```
      fs.create(tempOutputPath, false, bufferSize)
    } else {
      // This is mainly for testing purpose
      fs.create(tempOutputPath, false, bufferSize,
        fs.getDefaultReplication(fs.getWorkingDirectory), blockSize)
    }
    //序列化文件输出流
    val serializer = env.serializer.newInstance()
    val serializeStream = serializer.serializeStream(fileOutputStream)
    Utils.tryWithSafeFinally {
    //写数据
      serializeStream.writeAll(iterator)
    } {
      serializeStream.close()
    }

    if (!fs.rename(tempOutputPath, finalOutputPath)) {
      if (!fs.exists(finalOutputPath)) {
        logInfo(s"Deleting tempOutputPath $tempOutputPath")
        fs.delete(tempOutputPath, false)
        throw new IOException("Checkpoint failed: failed to save output of task: " +
          s"${ctx.attemptNumber()} and final output path does not exist: $finalOutputPath")
      } else {
        // Some other copy of this task must've finished before us and renamed it
        logInfo(s"Final output path $finalOutputPath already exists; not overwriting it")
        if (!fs.delete(tempOutputPath, false)) {
          logWarning(s"Error deleting ${tempOutputPath}")
        }
      }
    }
  }
```

writeRDDToCheckpointDirectoryWrite，将一个RDD写入到多个checkpoint文件，并返回一个ReliableCheckpointRDD来代表这个RDD

```
def writeRDDToCheckpointDirectory[T: ClassTag](
      originalRDD: RDD[T],
      checkpointDir: String,
      blockSize: Int = -1): ReliableCheckpointRDD[T] = {

    val sc = originalRDD.sparkContext

    // 生成 checkpoint文件 的输出路径
    val checkpointDirPath = new Path(checkpointDir)
    val fs = checkpointDirPath.getFileSystem(sc.hadoopConfiguration)
    if (!fs.mkdirs(checkpointDirPath)) {
      throw new SparkException(s"Failed to create checkpoint path $checkpointDirPath")
    }

    // 保存文件，并重新加载它作为一个RDD
    val broadcastedConf = sc.broadcast(
      new SerializableConfiguration(sc.hadoopConfiguration))
    sc.runJob(originalRDD,
      writePartitionToCheckpointFile[T](checkpointDirPath.toString, broadcastedConf) _)

    if (originalRDD.partitioner.nonEmpty) {
      writePartitionerToCheckpointDir(sc, originalRDD.partitioner.get, checkpointDirPath)
    }

    val newRDD = new ReliableCheckpointRDD[T](
      sc, checkpointDirPath.toString, originalRDD.partitioner)
    if (newRDD.partitions.length != originalRDD.partitions.length) {
      throw new SparkException(
        s"Checkpoint RDD $newRDD(${newRDD.partitions.length}) has different " +
```

```
30            s"number of partitions from original RDD $originalRDD(${originalRDD.partitions.length})")
31      }
32      newRDD
33    }
```

以上源码有可以改进的地方，因为重新计算RDD其实是没有必要的。

RDD checkpoint之后得到了一个新的RDD，那么child RDD 如何知道 parent RDD 有没有被checkpoint过呢？ 看 RDD的源码，我们可以发现：

```
1  private var dependencies_ : Seq[Dependency[_]] = null
```

dependencies_ 用来存放checkpoint后的结果的，如为null，则就判断没checkpoint：

```
1   final def dependencies: Seq[Dependency[_]] = {
2      checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {
3        if (dependencies_ == null) {
4          dependencies_ = getDependencies
5        }
6        dependencies_
7      }
8    }
```