Apache Spark细节与优化

本文将深入Spark一些细节方面以及在使用Spark的过程中优化的方法。

1. SparkContext加载

每个Spark应用都由一个驱动器程序（driver program）来发起集群上的各种并行操作。驱动器程序包含应用的main函数，并且定义了集群上的分布式数据集，还对这些分布式数据集应用了相关操作。

驱动器程序通过一个SparkContext对象来访问Spark。这个对象代表对计算集群的一个连接。在当用scala或者python版本的Spark shell时（在Spark根目录下通过bin/spark-shell或者bin/pyspark启动），shell启动时会自动创建一个SparkContext对象，是一个叫做sc的对象。而在独立应用中，需要自己创建SparkConf对象配置应用，然后基于这个SparkConf创建SparkContext对象。

在Java中初始化SparkContext：

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf( ).setMaster("local").setAppName("My App");
JavaSparkContext sc = new JavaSparkContext(conf);
```

在python中初始化SparkContext：

```
from pyspark import SparkConf, SparkContext

conf = SparkConf( ).setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

在Scala中初始化SparkContext：

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

val conf = new SparkConf( ).setMaster("local").setAppName("My App")
val sc = new SparkContext(conf)
```

需要传递两个参数：

（1）集群URL： 告诉Spark如何连接到集群上。在上面代码中为local，这个特殊值可以让Spark运行在单机单线程上而无需连接到集群。

（2）应用名：在上面代码中使用的是My App。当连接到一个集群时，这个值用来在集群管理器的用户界面中找到对应的应用。

2. RDD分区

为了能进行并行计算，RDD的内部数据集合在逻辑上和物理上被划分成为多个子集，每一个子集都称为分区，分区的个数会决定并行计算的粒度，而每一个分区数值的计算都是在一个单独的任务中进行，因此并行任务的个数，也是由RDD分区的个数决定的。

分区的源码实现类为partition类：

```
/**
 * An identifier for a partition in an RDD.
 */
trait Partition extends Serializable {
  /**
   * Get the partition's index within its parent RDD
   */
  def index: Int

  // A better default implementation of HashCode
  override def hashCode(): Int = index
}
```

图 1 Partition类源码

RDD只是数据集的抽象，分区内部并不会存储具体的数据。Partition类内包含一个index成员，表示该分区在RDD内的编号，通过RDD编号和分区编号就可以确定该分区对应的数据块编号，利用底层数据存储层提供的接口，就能从存储介质（比如HDFS或内存）中提取出分区对应的数据。

关于分区个数方面，RDD分区的一个分配原则是尽可能使得分区的个数，等于集群核心数目。RDD可以通过创建操作或者转换操作得到。转换操作中，分区的个数会根据转换操作对应多个RDD之间的依赖关系确定，窄依赖子RDD由父RDD分区个数决定，宽依赖由子RDD分区器决定。

创建操作中，可以手动指定分区的个数，例如sc.parallelize(Array(1, 2, 3, 4, 5), 2)表示创建得到的RDD分区个数为2，在没有指定分区个数的情况下，Spark会根据集群部署模式，来确定一个分区个数默认值。

对于parallelize方法，默认情况下，分区的个数会受Apache Spark配置参数spark.default.parallelism的影响，官方对于该参数的解释是用于控制shuffle过程中默认使用的任务数量。

```
/** Distribute a local Scala collection to form an RDD.
 *
 * @note Parallelize acts lazily. If `seq` is a mutable collection and is altered after
 * to parallelize and before the first action on the RDD, the resultant RDD will reflect
 * modified collection. Pass a copy of the argument to avoid this.
 * @note avoid using `parallelize(Seq())` to create an empty `RDD`. Consider `emptyRDD`
 * RDD with no partitions, or `parallelize(Seq[T]())` for an RDD of `T` with empty parti
 */
def parallelize[T: ClassTag](
    seq: Seq[T],
    numSlices: Int = defaultParallelism): RDD[T] = withScope {
  assertNotStopped()
  new ParallelCollectionRDD[T](this, seq, numSlices, Map[Int, Seq[String]]())
}
```

图 2 parallelize方法源码

无论是以本地模式、Standalone模式、Yarn模式还是Mesos模式来运行Apache Spark，分区的默认个数等于对spark.default.parallelism的指定值，若该值未设置，则Apache Spark会根据不同集群模式的特征来确定这个值。

对于本地模式，默认的分区个数等于本地机器的CPU核心总数，把每个分区的计算任务交给单个核心执行能够保证最大的计算效率。若使用Apache Mesos作为集群的

资源管理系统，默认分区个数等于8。对于Standalone或者Yarn模式，默认分区个数等于集群中所有核心数目的总和或者2，取两者中的较大值。

对于textFile方法，默认分区个数是min（spark.default.parallelism，2）。

3. 累加器与广播变量

共享变量是一种可以在Spark任务中使用的特殊类型的变量，介绍两种类型的共享变量：累加器（accumulator）与广播变量(broadcast variable)。通常在向Spark传递函数时，比如使用map( )函数或者用filter( )传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个task都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中对应的变量。这两个共享变量分别为结果聚合与广播这两种常见的通信模式突破了这一限制。

3.1. 累加器

累加器提供了将工作节点中的值聚合到驱动器程序中的简单语法。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。假设一个例子，从文件中读取拨号列表对应的日志，同时也需要统计文件中空行的数量。

```
JavaRDD<String> rdd = sc.textFile(args[1]);
final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
    new FlatMapFunction<String, String>( ) {
    if (line.equals("")) {
        blankLines.add(1);
    }
    return Arrays.asList(line.split(" "));
  }});
callSigns.saveAsTextFile("output.txt");
System.out.println("Blank lines: " + blankLines.value( ));
```

在上面代码中，创建了一个叫blankLines的Accumulator[Int]对象，然后在输入文件中读到空行就对累加器加1.执行完转换操作之后，就打印出累加器中的值。

累加器的用法如下：

1. 通过在驱动器中调用SparkContext.accumulator(initialValue)方法，创建出存有初始值的累加器。返回值为org.apache.spark.Accumulator[T]对象，其中T是初始值initialValue的类型。
2. Spark闭包里的执行器代码可以使用累加器的+=方法（在Java中是add）增加累加器的值。
3. 驱动器程序可以调用累加器的value属性来访问累加器的值。

工作节点上的任务不能访问累加器的值。从这些任务的角度来看，累加器是一个只写变量。在这种模式下，累加器的实现可以更加高效，不需要对每次更新操作进行复杂的通信。累加器的值只有在驱动器程序中可以访问。

3.2. 广播变量

广播变量让程序高效地向所有工作节点发送一个较大的只读值，以供一个或多个操作使用。比如，需要向所有节点发送一个较大的只读查询表。在默认情况下，Spark会自动把闭包中的所有引用到的变量发送到工作节点上。虽然这很方便，同时也很低效，原因有二：首先，默认的任务发射机制是专门为小任务进行优化的；其次，事实上可能会在多个并行操作中使用同一变量，但Spark会为每个操作分别发送。如果这个引用到的变量很大，从主节点为每个任务发送数据的代价会很大。

可以把共享的变量设为广播变量来解决这一问题。广播变量其实就是类型为spark.broadcast.Broadcast[T]的一个对象，其中存放着类型为T的值。可以在任务中通过对Broadcast对象调用value来获取该对象的值。这个值只会被发送到各节点一次，使用的是一种类似高效的类似BitTorrent的通信机制。

举一个例子，有一组通话记录，要根据区号统计联系每个国家的次数：

```
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable( ));
JavaPairRDD<String, Integer> countryContactCounts = contactCounts.mapToPair(
   new PairFunction<Tuple2<String, Integer>, String, Integer> ( ) {
       public Tuple2<String, Integer> call (Tuple2<String, Integer> callSignCount) {
    String sign = callSignCount._1( );
    String country = lookupCountry(sign, signPrefixes.value( ));
    return new Tuple2(country, callSignCount._2( ));
        }}).reduceByKey(new SumInts( ));
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

使用广播变量的过程：

（1）通过对一个类型T的对象调用SparkContext.broadcast创建出一个Broadcast[T]对象。任何可序列化的类型都可以这么实现。

（2）通过value属性访问该对象的值。

（3）变量只会被发到各个节点一次，应作为只读值处理，修改这个值不会影响到别的节点。

4. Spark运行时架构

在分布式环境下，Spark集群采用的是主从结构。在一个Spark集群中，有一个节点负责中央协调，调度各个分布式工作节点。这个中央协调节点被称为驱动器（Driver）节点，与之对应的工作节点被称为执行器（Executor）节点。驱动器节点可以和大量的执行器节点进行通信，它们也都作为独立的Java进程运行。驱动器节点和所有的执行器节点被称为一个Spark应用（Application）。

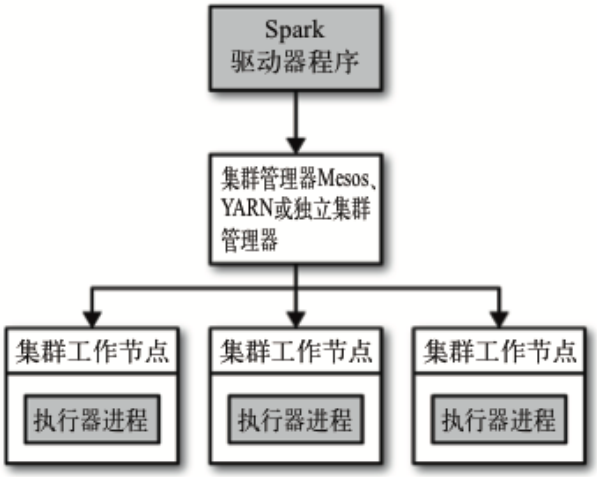

图 3 分布式Spark应用中的组件

Spark应用通过一个叫做集群管理器（Cluster Manager）的外部服务在集群中的机器上启动。Spark自带的集群管理器被称为独立集群管理器。Spark也能运行在Hadoop YARN和Apache Mesos这两大开源集群管理器上。

4.1 驱动器节点

Spark驱动器是执行程序中main()方法的进程。它执行用户编写的用来创建SparkContext、创建RDD，以及进行RDD的转化操作和行动操作的代码。驱动器程序一旦终止，Spark应用也就结束了。

驱动器程序在Spark中有两个职责：

（1）把用户程序转为任务

Spark驱动器程序负责把用户程序转为多个物理执行的单元，这些单元也被称为任务（task）。从上层来看，所有的Spark程序都遵循同样的结构：程序从输入数据创建一系列RDD，再使用转化操作派生出新的RDD，最后使用行动操作收集或存储结果RDD中的数据。Spark程序其实是隐式地创建出了一个由操作组成的逻辑上的有向无环图（Directed Acyclic Graph, 简称DAG）。当驱动器程序运行时，它会把这个逻辑图转为物理执行计划。

Spark会对逻辑执行计划作一些优化，比如将连续的映射转为流水线化执行，将多个操作合并到一个步骤中等等。这样Spark就把逻辑计划转为一系列步骤（stage）。而每个步骤又由多个任务组成。这些任务会被打包并送到集群中。任务是Spark中最小的工作单元，用户程序通常要启动成百上千个独立任务。

（2）为执行器节点调度任务

有了物理执行计划之后，Spark驱动器程序必须在各执行器进程间协调任务的调度。执行器进程启动后，会向驱动器进程注册自己。因此，驱动器进程始终对应用中所有的执行器节点有完整的记录。每个执行器节点代表一个能够处理任务和存储RDD数据的进程。

Spark驱动器程序会根据当前的执行器节点集合。尝试把所有任务基于数据所在位置分配给合适的执行器进程。当任务执行时，执行器进程会把缓存数据存储起来，而驱动器进程同样会跟踪这些缓存数据的位置，并且利用这些位置信息来调度以后的任务，以尽量减少数据的网络传输。

4.2. 执行器节点

Spark执行器是一种工作进程，负责在Spark作业中运行任务，任务间相互独立。Spark应用启动时，执行器节点就被同时启动，并且在整个Spark应用的生命周期中都同时存在。如果有执行器节点发生了异常或崩溃，Spark应用也可以继续执行。执行器有两大作用：第一，它们负责运行组成Spark应用的任务，并将结果返回给驱动器进程；第二，它们通过自身的块管理器为用户程序中要求缓存的RDD提供内存式存储。RDD是直接缓存在执行器进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

4.3. 集群上运行Spark应用的过程

（1）用户通过spark-submit脚本提交应用。

（2）spark-submit脚本启动驱动器程序，调用用户定义的main()方法。

（3）驱动器程序与集群管理器通信，申请资源以启动执行器节点。

（4）集群管理器为驱动器程序启动执行器节点。

（5）驱动器进程执行用户应用中的操作。根据程序中所定义的对RDD的转化操作和行动操作，驱动器节点把工作以任务的形式发送到执行器进程。

（6）任务在执行器程序中进行计算并保存结果。

（7）如果驱动器程序的main()方法退出，或者调用了SparkContext.stop()，驱动器程序会终止执行器进程，并且通过集群管理器释放资源。

5. Spark性能优化

5.1. 避免创建重复的RDD

在开发一个Spark作业时，通常是基于某个数据源（比如Hive表或HDFS文件）创建一个初始的RDD；接着对这个RDD执行某个转换操作，然后得到下一个RDD；经历多个转换操作后，通过行动操作得到最终需要的结果。在这个过程中，多个RDD会通过不同的转换操作（比如map、reduce等）串起来，这个RDD串就是RDD谱系图（lineage）。

在开发的过程中要注意：对于同一份数据，只应该创建一个RDD，避免创建多个RDD来代表同一份数据。在开发有很长的RDD谱系图的Spark作业时，可能会忘了之前对某一份数据已经创建过一个RDD了，从而导致对于同一份数据创建了多个RDD。这就意味着Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，这样增加了性能的开销。

```
// 需要对名为"hello.txt"的HDFS文件进行一次map操作，再进行一次reduce操作。
// 也就是说，需要对一份数据执行两次算子操作。
// 错误的做法：对于同一份数据执行多次算子操作时，创建多个RDD。
// 这里执行了两次textFile方法，针对同一个HDFS文件，创建了两个RDD出来，
// 然后分别对每个RDD都执行了一个算子操作。
// 这种情况下，Spark需要从HDFS上两次加载hello.txt文件的内容，并创建两个单独的RDD；
// 第二次加载HDFS文件以及创建RDD的性能开销，很明显是白白浪费掉的。
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd1.map(...)
val rdd2 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd2.reduce(...)

// 正确的用法：对于一份数据执行多次算子操作时，只使用一个RDD。
// 这种写法很明显比上一种写法要好多了，因为我们对于同一份数据只创建了一个RDD，
// 然后对这一个RDD执行了多次算子操作。
// 但是要注意到这里为止优化还没有结束，由于rdd1被执行了两次算子操作，
// 第二次执行reduce操作的时候，还会再次从源头处重新计算一次rdd1的数据，因此还是会有重复计算的性能开销。
// 要彻底解决这个问题，必须结合"原则三：对多次使用的RDD进行持久化"，才能保证一个RDD被多次使用时只被计算一次。
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd1.map(...)
rdd1.reduce(...)
```

图 4 重复RDD例子

5.2. 尽可能重用同一个RDD

除了要避免在开发过程中对一份完全相同的数据创建多个RDD之外，在对不同的数据执行转换操作时还要尽可能地复用一个RDD。比如，有一个RDD的数据格式是key-value类型的，另一个是单value类型的，这两个RDD的value数据是完全一样的。那么此时可以只使用key-value类型的那个RDD，因为其中已经包含了另一个数据。对于类似这种多个RDD的数据有重叠或者包含的情况，应该尽量复用一个RDD，这样可以减少RDD的数量，从而尽可能减少算子执行的次数。

```
// 错误的做法。

// 有一个<Long, String>格式的RDD，即rdd1。
// 由于业务需要，对rdd1执行了一个map操作，创建了一个rdd2，而rdd2中的数据仅仅是rdd1中的value值而已，
// 也就是说，rdd2是rdd1的子集。
JavaPairRDD<Long, String> rdd1 = ...
JavaRDD<String> rdd2 = rdd1.map(...)

// 分别对rdd1和rdd2执行了不同的算子操作。
rdd1.reduceByKey(...)
rdd2.map(...)

// 正确的做法。

// 上面这个case中，其实rdd1和rdd2的区别无非就是数据格式不同而已，rdd2的数据完全就是rdd1的子集而已，
// 却创建了两个rdd，并对两个rdd都执行了一次算子操作。
// 此时会因为对rdd1执行map算子来创建rdd2，而多执行一次算子操作，进而增加性能开销。
// 其实在这种情况下完全可以复用同一个RDD。
// 可以使用rdd1，既做reduceByKey操作，也做map操作。
// 在进行第二个map操作时，只使用每个数据的tuple._2，也就是rdd1中的value值，即可。
JavaPairRDD<Long, String> rdd1 = ...
rdd1.reduceByKey(...)
rdd1.map(tuple._2...)

// 第二种方式相较于第一种方式而言，很明显减少了一次rdd2的计算开销。
// 但是到这里为止，优化还没有结束，对rdd1我们还是执行了两次算子操作，rdd1实际上还是会被计算两次。
// 因此还需要配合“原则三：对多次使用的RDD进行持久化”进行使用，才能保证一个RDD被多次使用时只被计算一次。
```

图 5 复用RDD例子

5.3. 对多次使用的RDD进行持久化

保证对一个RDD执行多次转换操作时，这个RDD本身被计算一次。Spark中对于一个RDD执行多个转换操作的默认原理：每次对一个RDD执行一个转换操作时，都会从源头计算一遍，计算出参数中需要用的RDD，然后再对这个RDD执行这次的转换操作。这种方式的性能是很差的。

这种情况下有必要对多次使用的RDD进行持久化。此时Spark会根据用户的持久化策略将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行转换操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不会从源头出重新计算一遍RDD。

```scala
// 如果要对一个RDD进行持久化，只要对这个RDD调用cache()和persist()即可。

// 正确的做法。
// cache()方法表示：使用非序列化的方式将RDD中的数据全部尝试持久化到内存中。
// 此时再对rdd1执行两次算子操作时，只有在第一次执行map算子时，才会将这个rdd1从源头处计算一次。
// 第二次执行reduce算子时，就会直接从内存中提取数据进行计算，不会重复计算一个rdd。
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").cache()
rdd1.map(...)
rdd1.reduce(...)

// persist()方法表示：手动选择持久化级别，并使用指定的方式进行持久化。
// 比如说，StorageLevel.MEMORY_AND_DISK_SER表示，内存充足时优先持久化到内存中，
// 内存不充足时持久化到磁盘文件中。
// 而且其中的_SER后缀表示，使用序列化的方式来保存RDD数据，此时RDD中的每个partition
// 都会序列化成一个大的字节数组，然后再持久化到内存或磁盘中。
// 序列化的方式可以减少持久化的数据对内存/磁盘的占用量，进而避免内存被持久化数据占用过多，从而发生频繁GC。
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").persist(StorageLevel.MEMORY_AND_DISK_SER)
rdd1.map(...)
rdd1.reduce(...)
```

图 6 RDD持久化

对于persist( )方法而言，可以根据不同的应用场景选择不同的持久化级别。

持久化级别	含义
MEMORY_ONLY	使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。
MEMORY_AND_DISK	使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。
MEMORY_ONLY_SER	基本含义同MEMORY_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
MEMORY_AND_DISK_SER	基本含义同MEMORY_AND_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
DISK_ONLY	使用未序列化的Java对象格式，将数据全部写入磁盘文件中。
MEMORY_ONLY_2, MEMORY_AND_DISK_2, 等等	对于上述任意一种持久化策略，如果加上后缀_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。

表 1 Spark的持久化级别

持久化策略选择：

在默认情况下，性能最高的是MEMORY_ONLY，但是前提是内存必须足够大，可以放得下需要缓存的整个RDD的数据。不进行序列化与反序列化操作，就避免了这部分的性能开销。对这个RDD的后续操作，都是基于纯内存中的数据的操作，不需要从磁盘中读取，性能高。并且不需要复制一份数据副本，并远程传送到其他节点上。但是在现实中，使用这种策略的场景还是有限的。如果RDD中数据比较多，直接用这种持久化级别可能会导致JVM的内存溢出。

如果使用MEMORY_ONLY级别时发生了内存溢出，可以尝试使用MEMORY_ONLY_SER级别。该级别会将RDD数据序列化后再保存在内存中，此时每个RDD的partition仅仅是一个字节数组，减少了对象的数量，降低了内存占用。这种级别MEMORY_ONLY多出来的性能开销主要是序列化与反序列化的开销。但是后续的算子可以基于纯内存操作，性能总体还是可以的。但是扔可能发生内存溢出的情况。如果纯内存级别都无法使用，那么建议使用MEMORY_AND_DISK_SER策略，而不是MEMORY_AND_DISK策略。因为既然到了这一步，就说明RDD的数据量十分庞大，内存无法完全放下。序列化后的数据占用空间比较少，可以节省内存和磁盘的空间开销。同时该策略会优先将数据缓存在内存中，缓存不下的部分才会写入磁盘。

通常不建议使用DISK_ONLY和后缀为_2的级别：因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新从源头计算一次RDD。后缀为_2的级别，必须将所有数据都复制一份副本，并发送到其他节点上，数据复制以及网络传输会产生较大的开销。

5.4. 尽量避免使用shuffle类算子

在Spark作业运行过程中，最消耗性能的地方就是shuffle过程。shuffle过程会将分布在集群中多个节点上的同一个key，拉取到同一节点上，进行聚合或join等操作。比如reduceByKey、join算子，都会触发shuffle操作。

shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中相同的key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的原因之一。

因此在开发过程中，尽可能避免使用reduceByKey、join、repartition等会进行shuffle的算子，尽量使用map类的非shuffle算子。这样的话，没有shuffle操作或者仅有较少shuffle操作的spark作业在性能上会有较好的表现。

```
// 传统的join操作会导致shuffle操作。
// 因为两个RDD中，相同的key都需要通过网络拉取到一个节点上，由一个task进行join操作。
val rdd3 = rdd1.join(rdd2)

// Broadcast+map的join操作，不会导致shuffle操作。
// 使用Broadcast将一个数据量较小的RDD作为广播变量。
val rdd2Data = rdd2.collect()
val rdd2DataBroadcast = sc.broadcast(rdd2Data)

// 在rdd1.map算子中，可以从rdd2DataBroadcast中，获取rdd2的所有数据。
// 然后进行遍历，如果发现rdd2中某条数据的key与rdd1的当前数据的key是相同的，那么就判定可以进行join。
// 此时就可以根据自己需要的方式，将rdd1当前数据与rdd2中可以连接的数据，拼接在一起（String或Tuple）。
val rdd3 = rdd1.map(rdd2DataBroadcast...)

// 注意，以上操作，建议仅仅在rdd2的数据量比较少（比如几百M，或者一两G）的情况下使用。
// 因为每个Executor的内存中，都会驻留一份rdd2的全量数据。
```

图 7 用broadcast与map进行join操作

5.5. 使用map-side预聚合的shuffle操作

如果因为业务需要，一定要使用shuffle操作，无法使用map类的算子来替代，那么尽量使用可以map-side预聚合的算子。

所谓的map-side预聚合，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MapReduce中的本地combiner。map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据数量，从而减少了磁盘IO以及网络传输的开销。通常来说，尽可能使用reduceByKey或者aggregateByKey算子来替代掉groupByKey

算子。因为reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。而groupByKey算子是不会进行预聚合的，全部的数据会在集群的各个节点之间分发和传输，性能相对前面两个较差。

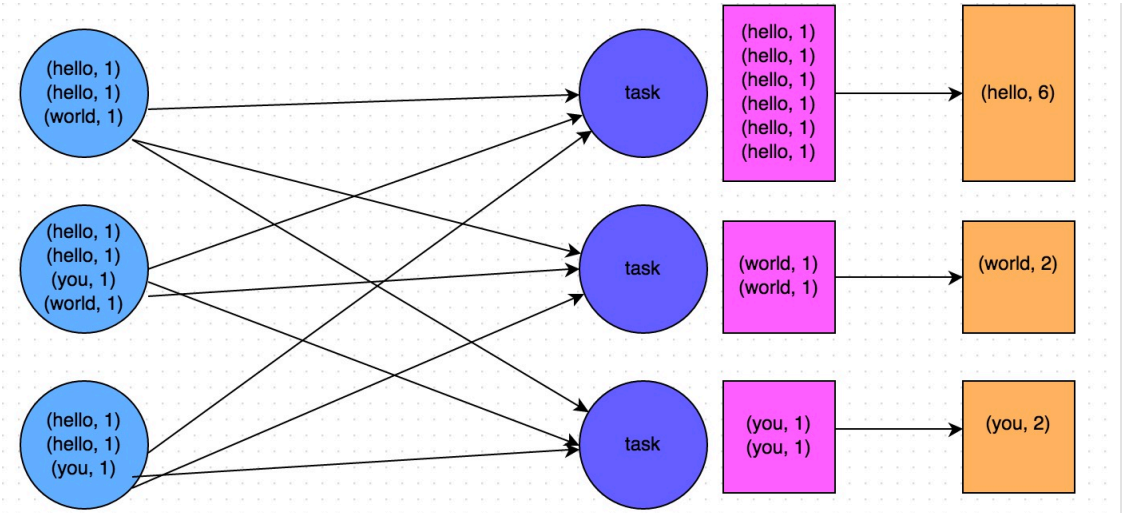

图 8 groupByKey原理

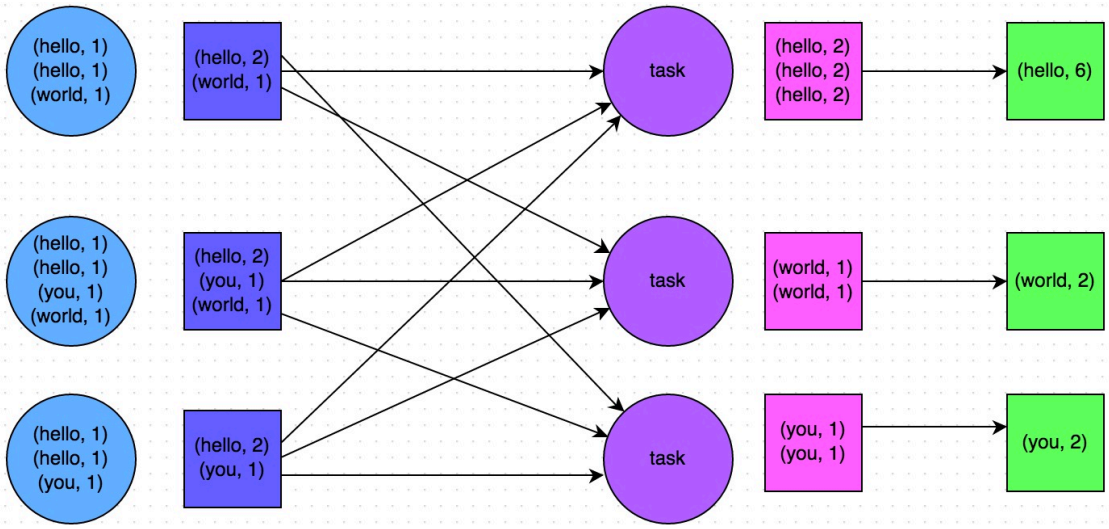

图 9 reduceByKey原理

5.6. 使用广播变量

有时在开发过程中，会遇到需要在算子中使用外部变量的场景（尤其是大的变量，比如100M以上的集合），那么这种情况下就应该使用Spark的广播变量功能来提升性能。

在算子函数中使用到外部变量时，默认情况下，Spark会将该变量复制多个副本，通过网络传输到各个task中，这样的话每个task都会有一个变量副本。如果变量本身比较大，那么大量的变量副本在网络中传输的性能开销以及在各个节点的Executor中占用过多内存导致的频繁GC，都会极大地影响性能。如果使用的外部变量比较大，建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本。这样的话可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对Executor内存的占用开销，降低GC的频率。

```
// 以下代码在算子函数中，使用了外部的变量。
// 此时没有做任何特殊操作，每个task都会有一份list1的副本。
val list1 = ...
rdd1.map(list1...)

// 以下代码将list1封装成了Broadcast类型的广播变量。
// 在算子函数中，使用广播变量时，首先会判断当前task所在Executor内存中，是否有变量副本。
// 如果有则直接使用；如果没有则从Driver或者其他Executor节点上远程拉取一份放到本地Executor内存中。
// 每个Executor内存中，就只会驻留一份广播变量副本。
val list1 = ...
val list1Broadcast = sc.broadcast(list1)
rdd1.map(list1Broadcast...)
```

图 10 广播变量

5.7. 使用Kryo优化序列化性能

在Spark中，主要有三个地方涉及到了序列化：

（1）在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输。

（2）将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化，因此这种情况下，也要求自定义的类必须实现Serializable接口。

（3）使用可序列化的持久化策略时，比如MEMORY_ONLY_SER，Spark会将RDD中的每个partition都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库来优化序列化和反序列化的性能。Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream来进行序列化和反序列化。但是Spark同时支持

使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介绍Kryo的序列化机制比Java序列化机制性能要高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说这种方式比较繁琐。

```
// 创建SparkConf对象。
val conf = new SparkConf().setMaster(...).setAppName(...)
// 设置序列化器为KryoSerializer。
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
// 注册要序列化的自定义类型。
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
```

图 11 Kryo示例

5.8. 资源调优

Spark的资源参数基本都可以在spark-submit命令中作为参数设置。参数设置的不合理可能会导致集群资源的利用低效，作业运行缓慢。或者设置的资源过大，队列没有足够的资源来提供，进而导致各种异常。

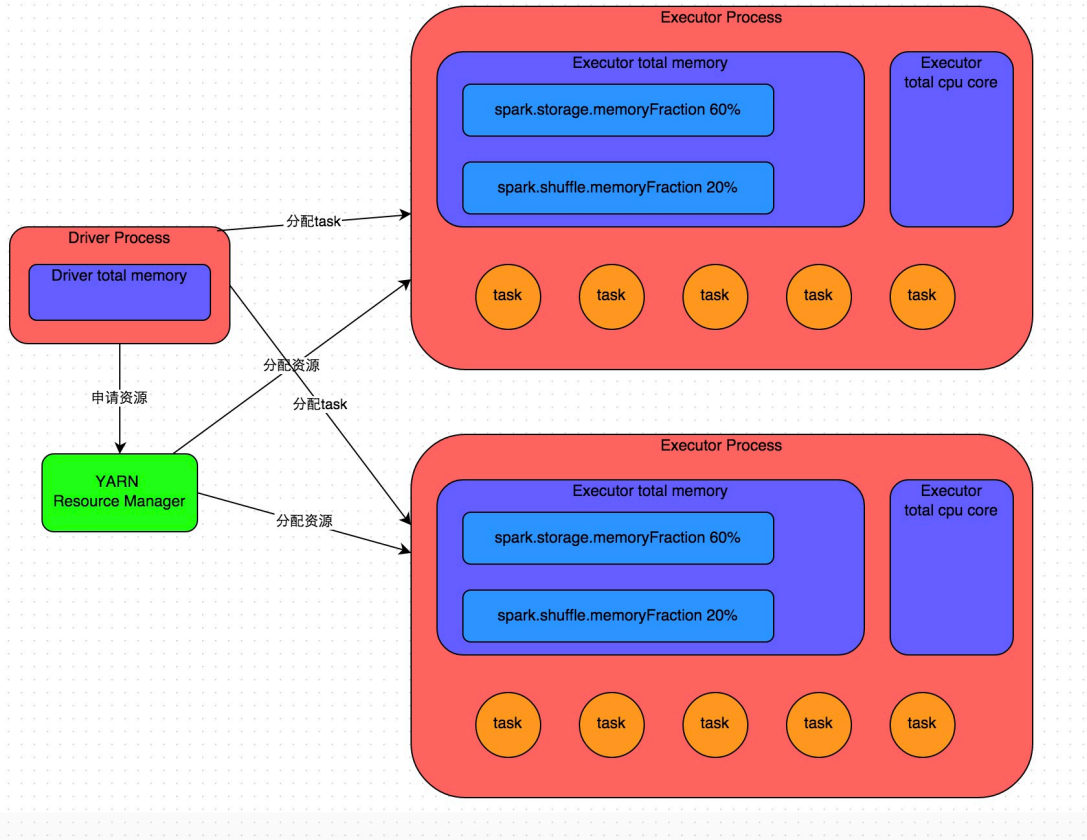

图 12 Spark作业运行流程

5.8.1. Spark作业基本运行原理

使用spark-submit提交一个作业之后，这个作业就会启动一个对应的Driver进程。根据使用的部署模式不同，Driver进程可能在本地启动，也可能在集群中某个工作节点上启动。Driver进程本身会根据设置的参数，占有一定数量的内存和CPU core。而Driver进程要做的第一件事就是向集群管理器申请运行Spark作业需要使用的资源，这里的资源就是Executor进程。集群管理器会根据Spark作业设置的资源参数，在各个工作节点上，启动一定数量的Executor进程，每个Executor进程都占有一定数量的内存和CPU core。

在申请到了作业执行所需的资源之后，Driver进程就会开始调度和执行编写的作业代码了。Driver进程会将编写的Spark作业代码分拆为多个stage，每个stage执行一部分代码片段，并为每个stage创建一批task，然后将这些task分配到各个Executor进程中执行。task是最小的计算单元，负责执行计算逻辑。一个stage的所有task都执行完毕后，会在各个节点本地的磁盘文件中写入中间结果，然后Driver就会调度运行下一个stage。下一个stage的task的输入数据就是上一个stage输出的结果。如此循环下去，直至代码逻辑全部执行完，得出结果。

Spark是根据shuffle类算子来进行stage的划分。如果代码中执行了某个shuffle类算子比如reduceByKey、join等，那么就会在该算子处划分出一个stage界限来。可以大致理解为，shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行已经之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点去通过网络传输拉取自己需要处理的key，然后对拉取到的所有相同的key使用自己编写的算子函数进行聚合操作。这个过程就是shuffle。

当在代码中执行cache或persist等持久化操作时，根据持久化级别不同，每个task计算出来的数据也会保存到Executor进程的内存或者所在节点的磁盘文件中。因此Executor的内存主要分为三块：第一块是让task执行编写的代码时使用，默认是占Executor总内存的20%；第二块是让task通过shuffle过程拉取了上一个stage的task的输出

后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是让RDD持久化时使用，默认占Executor总内存的60%；

task的执行速度是跟每个Executor进程的CPU core数量有直接关系的。一个CPU core同一时间只能执行一个线程。而每个Executor进程上分配到的多个task都是以每个task一条线程的方式，多线程并发运行的。如果CPU core数量比较充足，而且分配到的task数量比较合理，可以比较高效的完成这些task。

5.8.2. 资源参数调优

资源参数调优就是对Spark运行过程中各个使用资源的地方，通过调节各种参数，来优化资源使用的效率，从而提升Spark作业的执行性能。

（1）num-executors

该参数用于设置Spark作业总共要多少个Executor进程来执行。Driver在向集群管理器申请资源时，集群管理器会尽可能按照设置来在集群的各个工作节点上来启动相应数量的Executor进程。如果不设置的话，默认只会启动少量的Executor进程，这时候作业的运行速度是非常慢的。

每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适。

（2）executor-memory

该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能。每个Executor的进程的内存设置4G~8G比较合适。但是这只是一个参考值，具体还是根据资源队列的最大内存限制，num-executors * executor-memory的值不能超过资源队列的内存限制。

（3）executor-cores

该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core越多，越能快速的执行完分配给自己的所有task线程。

Executor的CPU core数量设置为2~4个比较合适，同样的具体还是根据资源队列的最大CPU core限制来看，num-executors * executor-cores的乘积不要大于这个限制。

（4）driver-memory

该参数用于设置Driver进程的内存

Driver的内存在大部分情况下不用设置或者设置为1G就足够，但需要注意的一点是，如果需要使用collect算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存够大，否则会出现内存溢出的问题。

（5）spark.default.parallelism

该参数用于设置每个stage的默认task数量。这个参数很重要，如果不设置可能会直接影响Spark的作业性能。

Spark作业的默认task数量为500~1000个较为合适，如果不设置这个参数，那么此时就会导致Spark自己根据底层HDFS的block数量来设置task的数量，默认是一个HDFS block对应一个task。通常来说，Spark默认设置的数量是偏少的，如果task数量偏少的话，无论Executor有多少个，内存和CPU有多大，但是task只有几十个，那么90%的Executor进程可能都没有执行task，这些资源都被浪费了。Spark官网建议的设置原则是该参数设置为num-executors * executor-cores的2~3倍比较合适。

（6）spark.storage.memoryFraction

该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor的60%的内存是用来保存持久化的RDD数据的。

如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致只能写入磁盘降低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数可以适当降低一些。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark的web界面可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么也可以调低这个参数的值来改善。

（7）spark.shuffle.memoryFraction

该参数用于设置shuffle过程中一个task拉取到上个stage的task输出后，进行聚合操作时能够使用的Executor内存比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超过了这个限制，多余的数据会溢写到磁盘中去，此时就会降低性能。

如果Spark作业中的RDD的shuffle操作较多，持久化操作较少时，建议提高这个参数的值，避免shuffle过程中数据过多是内存不够用的情况。此外，同样的，如果发现作业由于频繁的gc导致运行缓慢（通过spark的web界面可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么也可以调低这个参数的值来改善。

```
./bin/spark-submit \
  --master yarn-cluster \
  --num-executors 100 \
  --executor-memory 6G \
  --executor-cores 4 \
  --driver-memory 1G \
  --conf spark.default.parallelism=1000 \
  --conf spark.storage.memoryFraction=0.5 \
  --conf spark.shuffle.memoryFraction=0.3 \
```

图 13 spark-submit示例命令