# 第7章 调度系统

“诸君听吾计，使勃海引河内之众临孟津，酸枣诸将守成皋，据敖仓，塞轘辕、太谷，全制其险；使袁将军率南阳之军军丹、析，入武关，以震三辅：皆高垒深壁，勿与战，益为疑兵，示天下形势，以顺诛逆，可立定也。”

——《三国志·魏书·武帝纪》

本章导读

根据《三国志》对“联军伐卓”的记载，袁绍、袁术、韩馥、孔伷、刘岱、张邈、王匡、桥瑁、袁遗、鲍信、曹操等各方军事势力结成军事联盟，一同攻伐董卓。当时的曹操可以说是各方军事势力中最弱的，但曹操身担奋武将军一职并筹划了上面的军事调遣与分派。尽管联军并未采纳曹操的军事谋略，但是足以看出曹操是个军事天才。此后曹操从盟军中势力最弱的一方逐渐发展成为独霸北方，挟天子以令诸侯的一代枭雄。

在Spark集群上运行的任务多种多样，有的任务计算的数据量很大，有的任务执行的时间较长，有的任务执行的优先级较高，有的任务对公司的业务非常重要。整个Spark集群中各个机器的资源情况又有很多差异，有些机器CPU内核数较多，有些机器的内存很大，有些机器的磁盘空间非常充足，有些机器的网卡配置较高。有些机器在同一个机架上，有些机器可能在其他机房。任务应当分配在哪些机器上执行，任务应当分配多少资源，任务又应当有怎样的优先级？对集群资源及任务的合理分配与调度将决定整个集群的性能和效率。

本章将介绍Spark的调度系统，主要讲解的内容如下。

·调度系统概述。

·为什么需要RDD及RDD实现的初次分析。

·阶段（Stage）。

·DAG调度器（DAGScheduler）。

·调度池（Pool）。

·任务集合管理器（TaskSetManager）。

·运行器后端接口（LauncherBackend）。

·调度后端接口（SchedulerBackend）。

·任务结果获取器（TaskResultGetter）。

·任务调度器（TaskScheduler）。

## 7.1 调度系统概述

简单来讲，Spark调度系统用于将用户提交的“任务”调度到集群中的不同节点执行。但是Spark实现的调度系统，并非一句话所概括的这么简单。Spark资源调度分为两层：第一层是Cluster Manager（在YARN模式下为ResourceManager，在Mesos模式下为Mesos Master，在Standalone模式下为Master），将资源分配给Application；第二层是Application，进一步将资源分配给Application的各个Task。第一层调度中，Standalone模式下的Master对资源的调度将在9.6.4节详细介绍，至于其他模式中ResourceManager、Mesos Master对资源的调度，读者可以查阅YARN或Mesos的相关资料了解。本章所介绍的调度系统只限于第二层。

调度的对象是什么？作业或者任务。在业务场景中所说的任务与Spark集群中运行的任务在概念上存在很多偏差。人类往往把一件要做的事情认为是任务，因此“向Spark集群提交一个任务”这句话看似合情合理。工程师向Spark提交的一个任务，Spark却看作一个作业（Job）。Spark首先会对Job进行一系列RDD转换，并通过RDD之间的依赖关系构建有向无环图（Direct Acyclic Graph，DAG）。然后根据RDD依赖的不同将RDD划分到不同的阶段（Stage），每个阶段按照分区（Partition）的数量创建多个任务（Task）。最后将这些任务提交到集群的各个运行节点上运行。因此Spark中的Task和业务场景中所述的任务是不同的。

图7-1对调度系统的主要工作流程进行了展示，笔者将对此图的内容进行一些补充，以使读者更容易理解。

根据图7-1，我们知道Spark调度系统主要由DAGScheduler和TaskScheduler构成。调度系统的主要工作流程如下。

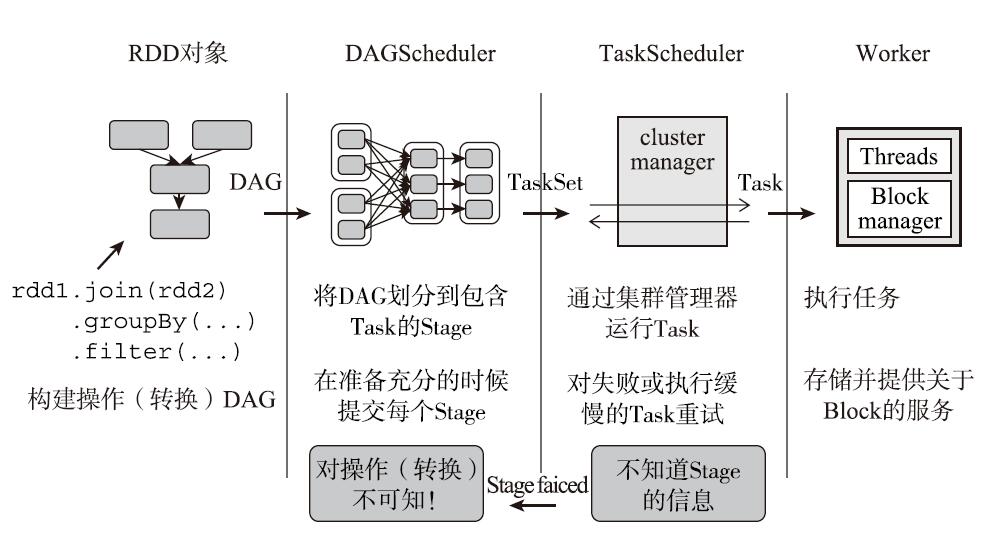


图7-1 调度系统的主要工作流程 [[1]](#_1_43)

1）build operator DAG：用户提交的Job将首先被转换为一系列RDD并通过RDD之间的依赖关系构建DAG，然后将RDD构成的DAG提交到调度系统。

2）split graph into stages of tasks：DAGScheduler负责接收由RDD构成的DAG，将一系列RDD划分到不同的Stage。根据Stage的不同类型（目前有ResultStage和Shuffle MapStage两种），给Stage中未完成的Partition创建不同类型的Task（目前有ResultTask和ShuffleMapTask两种）。每个Stage将因为未完成Partition的多少，创建零到多个Task。DAGScheduler最后将每个Stage中的Task以任务集合（TaskSet）的形式提交给Task Scheduler继续处理。

3）launch tasks via cluster manager：使用集群管理器（cluster manager）分配资源与任务调度，对于失败的任务还会有一定的重试与容错机制。TaskScheduler负责从DAGScheduler接收TaskSet，创建TaskSetManager对TaskSet进行管理，并将此TaskSetManager添加到调度池中，最后将对Task的调度交给调度后端接口（SchedulerBackend）处理。SchedulerBackend首先申请TaskScheduler，按照Task调度算法（目前有FIFO和FAIR两种）对调度池中的所有TaskSetManager进行排序，然后对TaskSet按照最大本地性原则分配资源，最后在各个分配的节点上运行TaskSet中的Task。

4）execute tasks：执行任务，并将任务中间结果和最终结果存入存储体系。

Image00042.jpg 注意 以上介绍的4个步骤中，严格来讲只有第2步和第3步属于调度系统的范畴，第1步是将作业提交给调度系统前的准备工作，第4步也和调度系统有很多关系，例如，map任务执行成功后将唤醒下游的reduce任务。

[[1]](#_1_42) 此图源自网络，出处不可考，笔者认为画得非常好，没有必要再画了，如果原图作者看到本书，可与出版社或笔者联系。

## 7.2 RDD详解

RDD（Resilient Distributed Datasets，弹性分布式数据集）代表可并行操作元素的不可变分区集合。严格来讲，RDD的转换及DAG的构成并不属于调度系统的内容，但是RDD却是调度系统操作的主要对象，因此有必要对RDD进行详细介绍。

### 7.2.1 为什么需要RDD

以下从数据处理模型、依赖划分原则、数据处理效率及容错处理4个方面解释Spark为什么需要RDD。

1.数据处理模型

RDD是一个容错的、并行的数据结构，可以控制将数据存储到磁盘或内存，能够获取数据的分区。RDD提供了一组类似于Scala的操作，比如map、flatMap、filter、reduceByKey、join、mapPartitions等，这些操作实际是对RDD进行转换（transformation）。此外，RDD还提供了collect、foreach、count、reduce、countByKey等操作完成数据计算的动作（action）。

当前的大数据应用场景非常丰富，如流式计算、图计算、机器学习等，它们既有相似之处，又各有不同。为了能够对所有场景下的数据处理使用统一的方式，抽象出RDD这一模型。

通常数据处理的模型包括迭代计算、关系查询、MapReduce、流式处理等。Hadoop采用MapReduce模型，Storm采用流式处理模型，而Spark则借助RDD实现了以上所有模型。

2.依赖划分原则

一个RDD包含一个或者多个分区，每个分区实际是一个数据集合的片段。在构建DAG的过程中，会将RDD用依赖关系串联起来。每个RDD都有其依赖（除了最顶级RDD的依赖是空列表），这些依赖分为窄依赖（即NarrowDependency）和Shuffle依赖（即ShuffleDependency，也称为宽依赖）两种。为什么要对依赖进行区分？从功能角度讲它们是不一样的。NarrowDependency会被划分到同一个Stage中，这样它们就能以管道的方式迭代执行。ShuffleDependency由于所依赖的分区Task不止一个，所以往往需要跨节点传输数据。从容灾角度讲，它们恢复计算结果的方式不同。NarrowDependency只需要重新执行父RDD的丢失分区的计算即可恢复，而ShuffleDependency则需要考虑恢复所有父RDD的丢失分区。

解释了依赖划分的原因，实际也解释了为什么要划分Stage这个问题。

3.数据处理效率

RDD的计算过程允许在多个节点并发执行。如果数据量很大，可以适当增加分区数量，这种根据硬件条件对并发任务数量的控制，能更好地利用各种资源，也能有效提高Spark的数据处理效率。

4.容错处理

传统关系型数据库往往采用日志记录的方式来容灾容错，数据恢复都依赖于重新执行日志。Hadoop为了避免单机故障概率较高的问题，通常将数据备份到其他机器容灾。由于所有备份机器同时出故障的概率比单机故障概率低很多，所以在发生宕机等问题时能够从备份机读取数据。RDD本身是一个不可变的（Scala中称为immutable）数据集，当某个Worker节点上的Task失败时，可以利用DAG重新调度计算这些失败的Task（执行已成功的Task可以从CheckPoint（检查点）中读取，而不用重新计算）。在流式计算的场景中，Spark需要记录日志和CheckPoint，以便利用CheckPoint和日志对数据恢复。

### 7.2.2 RDD实现的初次分析

有些读者可能对本节的标题感到困惑，这是因为RDD的API非常多，为了使本章对调度系统的表述更为方便，所以本节只对RDD中与调度系统息息相关的API方法进行分析，转换API、动作API及检查点API将在10.4节进行介绍。

抽象类RDD定义了所有RDD的规范，我们从RDD的属性开始，逐步了解RDD的实现。

·\_sc：指SparkContext。\_sc由@transient修饰，所以此属性不会被序列化。

·deps：构造器参数之一，是Dependency的序列，用于存储当前RDD的依赖。RDD的子类在实现时不一定会传递此参数。由于deps由@transient修饰，所以此属性不会被序列化。

·partitioner：当前RDD的分区计算器。partitioner由@transient修饰，所以此属性不会被序列化。

·id：当前RDD的唯一身份标识。此属性通过调用SparkContext的nextRddId属性生成。

·name：RDD的名称。name由@transient修饰，所以此属性不会被序列化。

·dependencies\_：与deps相同，但是可以被序列化。

·partitions\_：存储当前RDD的所有分区的数组。partitions\_由@transient修饰，所以此属性不会被序列化。

·storageLevel：当前RDD的存储级别。

·creationSite：创建当前RDD的用户代码。creationSite由@transient修饰，所以此属性不会被序列化。

·scope：当前RDD的操作作用域。scope由@transient修饰，所以此属性不会被序列化。

·checkpointData：当前RDD的检查点数据。

·checkpointAllMarkedAncestors：是否对所有标记了需要保存检查点的祖先保存检查点。

·doCheckpointCalled：是否已经调用了doCheckpoint方法设置检查点。此属性可以阻止对RDD多次设置检查点。

RDD采用了模板方法的模式设计，抽象类RDD中定义了模板方法及一些未实现的接口，这些接口将需要RDD的各个子类分别实现。下面先来介绍RDD中定义的接口。

·compute：对RDD的分区进行计算。此方法的定义如下：

　　@DeveloperApi  
　　def compute(split: Partition, context: TaskContext): Iterator[T]

·getPartitions：获取当前RDD的所有分区。此方法的定义如下：

　　protected def getPartitions: Array[Partition]

·getDependencies：获取当前RDD的所有依赖。此方法的定义如下：

　　protected def getDependencies: Seq[Dependency[\_]] = deps

·getPreferredLocations：获取某一分区的偏好位置。此方法的定义如下：

　　protected def getPreferredLocations(split: Partition): Seq[String] = Nil

RDD中除定义了以上接口外，还实现了一些模板方法。

1.partitions

partitions方法（见代码清单7-1）用于获取RDD的分区数组。

代码清单7-1 partitions的实现

final def partitions: Array[Partition] = {  
 checkpointRDD.map(\_.partitions).getOrElse {  
 if (partitions\_ == null) {  
 partitions\_ = getPartitions  
 // 省略次要代码  
 }  
 partitions\_  
 }  
}

根据代码清单7-1，partitions方法查找分区数组的优先级为：从CheckPoint查找→读取partitions\_属性→调用getPartitions方法获取。检查点的内容将在10.3节详细介绍。

2.preferredLocations

preferredLocations方法（见代码清单7-2）优先调用CheckPoint中保存的RDD的getPreferred-Locations方法获取指定分区的偏好位置，当没有保存CheckPoint时，调用自身的getPreferred-Locations方法获取指定分区的偏好位置。

代码清单7-2 preferredLocations的实现

final def preferredLocations(split: Partition): Seq[String] = {  
 checkpointRDD.map(\_.getPreferredLocations(split)).getOrElse {  
 getPreferredLocations(split)  
 }  
}

3.dependencies

dependencies方法（见代码清单7-3）用于获取当前RDD的所有依赖的序列。

代码清单7-3 dependencies的实现

final def dependencies: Seq[Dependency[\_]] = {  
 checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {  
 if (dependencies\_ == null) {  
 dependencies\_ = getDependencies  
 }  
 dependencies\_  
 }  
}

根据代码清单7-3，dependencies方法的执行步骤如下。

1）从CheckPoint中获取RDD，并将这些RDD封装为OneToOneDependency列表。如果从CheckPoint中获取到RDD的依赖，则返回RDD的依赖，否则进入下一步。

2）如果dependencies\_等于null，那么调用子类实现的getDependencies方法获取当前RDD的依赖后赋予dependencies，最后返回dependencies\_。

4.其他方法

除了以上的模板方法，RDD中还实现了以下一些方法。

（1）context

context方法（见代码清单7-4）实际返回了\_sc（即SparkContext）。

代码清单7-4 context的实现

def context: SparkContext = sc

（2）getStorageLevel

getStorageLevel方法（见代码清单7-5）实际返回了当前RDD的StorageLevel。

代码清单7-5 获取当前RDD的存储级别

def getStorageLevel: StorageLevel = storageLevel

（3）getNarrowAncestors

getNarrowAncestors方法（见代码清单7-6）用于获取当前RDD的祖先依赖中属于窄依赖的RDD序列。

代码清单7-6 getNarrowAncestors的实现

private[spark] def getNarrowAncestors: Seq[RDD[\_]] = {  
 val ancestors = new mutable.HashSet[RDD[\_]]  
 def visit(rdd: RDD[\_]) {  
 val narrowDependencies = rdd.dependencies.filter(\_.isInstanceOf[NarrowDependency[\_]])  
 val narrowParents = narrowDependencies.map(\_.rdd)  
 val narrowParentsNotVisited = narrowParents.filterNot(ancestors.contains)  
 narrowParentsNotVisited.foreach { parent =>  
 ancestors.add(parent)  
 visit(parent)  
 }  
 }  
  
 visit(this)  
  
 ancestors.filterNot(\_ == this).toSeq  
}

### 7.2.3 RDD依赖

DAG中的各个RDD之间存在着依赖关系。换言之，正是RDD之间的依赖关系构建了由RDD所组成的DAG。Spark使用Dependency来表示RDD之间的依赖关系，Dependency的定义如下。

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

抽象类Dependency只定义了一个名叫rdd的方法，此方法返回当前依赖的RDD。

Dependency分为NarrowDependency和ShuffleDependency两种依赖，下面对它们分别进行介绍。

1.窄依赖

如果RDD与上游RDD的分区是一对一的关系，那么RDD和其上游RDD之间的依赖关系属于窄依赖（NarrowDependency）。NarrowDependency继承了Dependency，以表示窄依赖。NarrowDependency的定义如下。

@DeveloperApi  
abstract class NarrowDependency[T](\_rdd: RDD[T]) extends Dependency[T] {  
 def getParents(partitionId: Int): Seq[Int]  
 override def rdd: RDD[T] = \_rdd  
}

NarrowDependency定义了一个类型为RDD的构造器参数\_rdd，NarrowDependency重写了Dependency的rdd方法，让其返回\_rdd。NarrowDependency还定义了一个获取某一分区的所有父级别分区序列的getParents方法。NarrowDependency一共有两个子类，它们的实现如代码清单7-7所示。

代码清单7-7 OneToOneDependency与RangeDependency

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

根据代码清单7-7，OneToOneDependency重写的getParents方法告诉我们，子RDD的分区与依赖的父RDD的分区相同。OneToOne Dependency可以用图7-2更形象地说明。

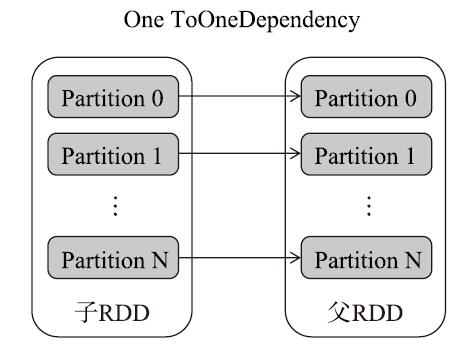


图7-2 OneToOneDependency的依赖示意图

根据代码清单7-7，RangeDependency重写了Dependency的getParents方法，其实现告诉我们Range Dependency的分区是一对一的，且索引为partitionId的子RDD分区与索引为partitionId-outStart+inStart的父RDD分区相对应（outStart代表子RDD的分区范围起始值，inStart代表父RDD的分区范围起始值）。RangeDependency可以用图7-3更形象地说明。

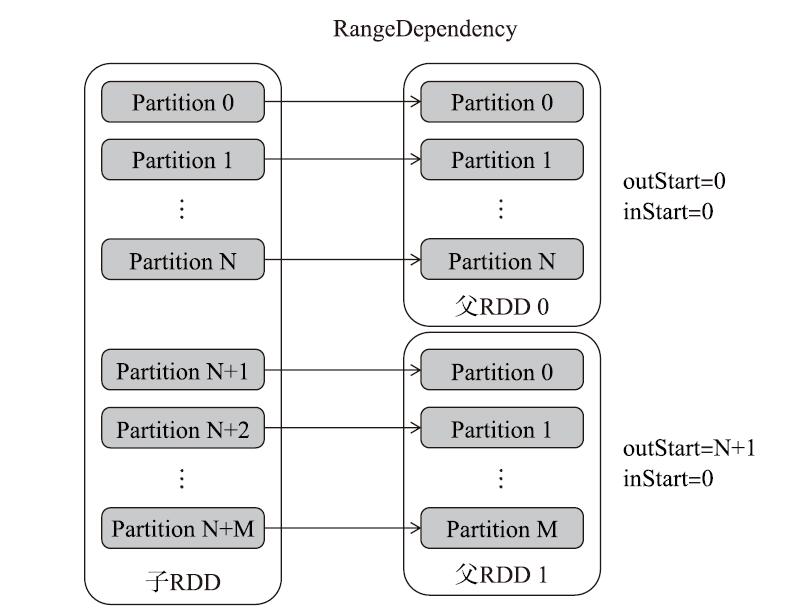


图7-3 RangeDependency的依赖示意图

2.Shuffle依赖

RDD与上游RDD的分区如果不是一对一的关系，或者RDD的分区依赖于上游RDD的多个分区，那么这种依赖关系就叫做Shuffle依赖（ShuffleDependency）。ShuffleDependency的实现如代码清单7-8所示。

代码清单7-8 ShuffleDependency的实现

@DeveloperApi  
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  
 private[spark] val combinerClassName: Option[String] =  
 Option(reflect.classTag[C]).map(\_.runtimeClass.getName)  
 val shuffleId: Int = \_rdd.context.newShuffleId()  
 val shuffleHandle: ShuffleHandle = \_rdd.context.env.shuffleManager.registerShuffle(  
 shuffleId, \_rdd.partitions.length, this)  
 \_rdd.sparkContext.cleaner.foreach(\_.registerShuffleForCleanup(this))  
}

根据代码清单7-8，ShuffleDependency定义了如下属性。

·\_rdd：泛型要求必须是Product2[K,V]及其子类的RDD。

·partitioner：分区计算器Partitioner。Partitioner将在下一节详细介绍。

·serializer：SparkEnv中创建的serializer，即org.apache.spark.serializer.JavaSerializer。

·keyOrdering：按照K进行排序的scala.math.Ordering的实现类。

·aggregator：对map任务的输出数据进行聚合的聚合器。

·mapSideCombine：是否在map端进行合并，默认为false。

·keyClassName：K的类名。

·valueClassName：V的类名。

·combinerClassName：结合器C的类名。

·shuffleId：当前ShuffleDependency的身份标识。

·shuffleHandle：当前ShuffleDependency的处理器。

此外，ShuffleDependency还重写了父类Dependency的rdd方法，其实现将\_rdd转换为RDD[Product2[K,V]]后返回。ShuffleDependency在构造的过程中还将自己注册到SparkContext的ContextCleaner中。

### 7.2.4 分区计算器Partitioner

RDD之间的依赖关系如果是Shuffle依赖，那么上游RDD该如何确定每个分区的输出将交由下游RDD的哪些分区呢？或者下游RDD的各个分区将具体依赖于上游RDD的哪些分区呢？Spark提供了分区计算器来解决这个问题。ShuffleDependency的partitioner属性的类型是Partitioner，抽象类Partitioner定义了分区计算器的接口规范，ShuffleDependency的分区取决于Partitioner的具体实现。Partitioner的定义如下。

abstract class Partitioner extends Serializable {  
 def numPartitions: Int  
 def getPartition(key: Any): Int  
}

Partitioner的numPartitions方法用于获取分区数量。Partitioner的getPartition方法用于将输入的key映射到下游RDD的从0到numPartitions-1这一范围中的某一个分区。

Partitioner有很多具体的实现类，它们的继承体系如图7-4所示。

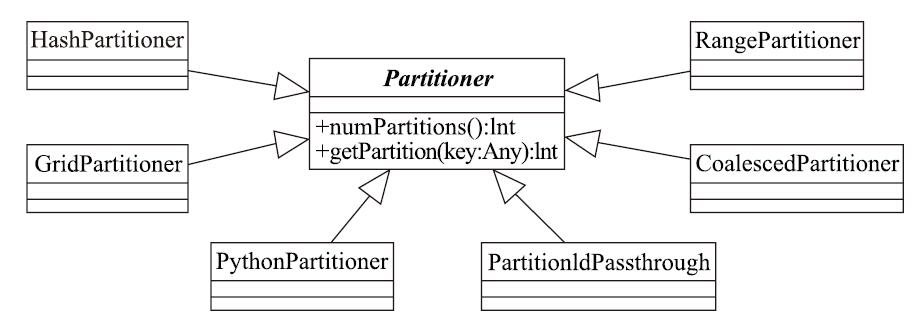


图7-4 Partitioner的继承体系

Spark中除图7-4列出的Partitioner子类，还有很多Partitioner的匿名实现类，这里就不一一介绍了。本书以HashPartitioner（哈希分区计算器）为例，详细介绍Partitioner的实现。之所以选择对HashPartitioner的实现进行分析，一方面是由于其实现简洁明了，读者更容易理解；另一方面通过介绍HashPartitioner，已经足够达到本书的目的。

HashPartitioner的实现如代码清单7-9所示。

代码清单7-9 哈希分区计算器的实现

class HashPartitioner(partitions: Int) extends Partitioner {  
 require(partitions >= 0, s"Number of partitions ($partitions) cannot be negative.")  
 def numPartitions: Int = partitions  
 def getPartition(key: Any): Int = key match { // 计算出下游RDD的各个分区将具体处理哪些key  
 case null => 0  
 case \_ => Utils.nonNegativeMod(key.hashCode, numPartitions)  
 }  
 override def equals(other: Any): Boolean = other match {  
 case h: HashPartitioner =>  
 h.numPartitions == numPartitions  
 case \_ =>  
 false  
 }  
 override def hashCode: Int = numPartitions  
}

根据代码清单7-9，HashPartitioner增加了一个名为partitions的构造器参数作为分区数，重写的numPartitions方法只是返回了partitions。重写的getPartition方法实际以key的hashCode和numPartitions作为参数调用了Utils工具类的nonNegativeMod方法（具体实现可以参阅附录A）。nonNegativeMod方法将对key的hashCode和numPartitions进行取模运算，得到key对应的分区索引。使用哈希和取模的方式，可以方便地计算出下游RDD的各个分区将具体处理哪些key。由于上游RDD所处理的key的哈希值在取模后很可能产生数据倾斜，所以HashPartitioner并不是一个均衡的分区计算器。

根据HashPartitioner的实现，我们知道Shuffle Dependency中的分区依赖关系不再是一对一的，而是取决于key，并且当前RDD的某个分区将可能依赖于ShuffleDependency的RDD的任何一个分区。经过以上分析，ShuffleDependency采用HashPartitioner后的分区依赖可以用图7-5来表示。

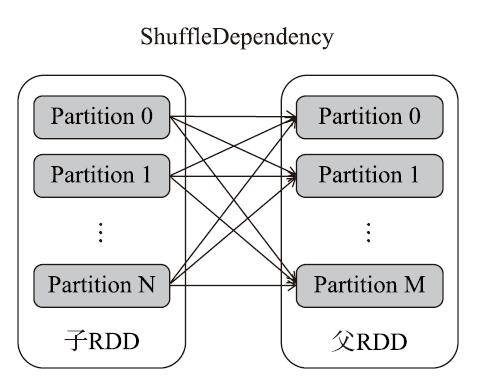


图7-5 ShuffleDependency的依赖示意图

### 7.2.5 RDDInfo

RDDInfo用于描述RDD的信息，RDDInfo提供的信息如下。

·id：RDD的id。

·name：RDD的名称。

·numPartitions：RDD的分区数量。

·storageLevel：RDD的存储级别（即StorageLevel）。

·parentIds：RDD的父RDD的id序列。这说明一个RDD会有零到多个父RDD。

·callSite：RDD的用户调用栈信息。

·scope：RDD的操作范围。scope的类型为RDDOperationScope，每一个RDD都有一个RDDOperationScope。RDDOperationScope与Stage或Job之间并无特殊关系，一个RDDOperationScope可以存在于一个Stage内，也可以跨越多个Job。

·numCachedPartitions：缓存的分区数量。

·memSize：使用的内存大小。

·diskSize：使用的磁盘大小。

·externalBlockStoreSize：Block存储在外部的大小。

RDDInfo还提供了以下方法。

·isCached：是否已经缓存。isCached的实现如代码清单7-10所示。

代码清单7-10 isCached的实现

def isCached: Boolean = (memSize + diskSize > 0) && numCachedPartitions > 0

·compare：由于RDDInfo继承了Ordered，所以重写了compare方法用于排序。compare的实现如代码清单7-11所示。

代码清单7-11 compare的实现

override def compare(that: RDDInfo): Int = {  
 this.id - that.id  
}

此外，RDDInfo的伴生对象中定义了fromRdd方法，用于从RDD构建出对应的RDDInfo,其实现如代码清单7-12所示。

代码清单7-12 RDDInfo伴生对象的fromRdd方法

private[spark] object RDDInfo {  
 def fromRdd(rdd: RDD[\_]): RDDInfo = {  
 val rddName = Option(rdd.name).getOrElse(Utils.getFormattedClassName(rdd))  
 val parentIds = rdd.dependencies.map(\_.rdd.id)  
 new RDDInfo(rdd.id, rddName, rdd.partitions.length,  
 rdd.getStorageLevel, parentIds, rdd.creationSite.shortForm, rdd.scope)  
 }  
}

根据代码清单7-12，fromRdd方法的执行步骤如下。

1）获取当前RDD的名称（即name属性）作为RDDInfo的name属性，如果RDD还没有名称，那么调用Utils工具类的getFormattedClassName方法（见附录A）生成RDDInfo的name属性。

2）获取当前RDD依赖的所有父RDD的身份标识作为RDDInfo的parentIds属性。

3）创建RDDInfo对象。

## 7.3 Stage详解

DAGScheduler会将Job的RDD划分到不同的Stage，并构建这些Stage的依赖关系。这样可以使得没有依赖关系的Stage并行执行，并保证有依赖关系的Stage顺序执行。并行执行能够有效利用集群资源，提升运行效率，而串行执行则适用于那些在时间和数据资源上存在强制依赖的场景。Stage分为需要处理Shuffle的ShuffleMapStage和最下游的ResultStage。上游Stage先于下游Stage执行，ResultStage是最后执行的Stage。要了解Stage，应该从Stage的属性开始。Stage的属性如下。

·id：Stage的身份标识。

·rdd：当前Stage包含的RDD。

·numTasks：当前Stage的Task数量。

·parents：当前Stage的父Stage列表。这说明一个Stage可以有一到多个父亲Stage。

·firstJobId：第一个提交当前Stage的Job的身份标识（即Job的id）。当使用FIFO调度时，通过firstJobId首先计算来自较早Job的Stage，或者在发生故障时更快的恢复。

·callSite：应用程序中与当前Stage相关联的调用栈信息。

·numPartitions：当前Stage的分区数量。numPartitions实际为rdd的分区的数量。

·jobIds：当前Stage所属的Job的身份标识集合。这说明一个Stage可以属于一到多个Job。

·pendingPartitions：存储待处理分区的索引的集合。

·nextAttemptId：用于生成Stage下一次尝试的身份标识。

·\_latestInfo：Stage最近一次尝试的信息，即StageInfo。

·fetchFailedAttemptIds：发生过FetchFailure的Stage尝试的身份标识的集合。此属性用于避免在发生FetchFailure后无止境的重试。

有了对Stage属性的了解，现在看看Stage提供的方法。

·clearFailures：清空fetchFailedAttemptIds。

·failedOnFetchAndShouldAbort：用于将发生FetchFailure的Stage尝试的身份标识添加到fetchFailedAttemptIds中，并返回发生FetchFailure的次数是否已经超过了允许发生FetchFailure的次数的状态。允许发生FetchFailure的次数固定为4。

·latestInfo：返回最近一次Stage尝试的StageInfo，即返回\_latestInfo。

·findMissingPartitions：找到还未执行完成的分区。此方法需要子类实现。

·makeNewStageAttempt：用于创建新的Stage尝试，实现如代码清单7-13所示。

代码清单7-13 创建新的Stage尝试

def makeNewStageAttempt(  
 numPartitionsToCompute: Int,  
 taskLocalityPreferences: Seq[Seq[TaskLocation]] = Seq.empty): Unit = {  
 val metrics = new TaskMetrics  
 metrics.register(rdd.sparkContext)  
 \_latestInfo = StageInfo.fromStage(  
 this, nextAttemptId, Some(numPartitionsToCompute), metrics, taskLocalityPreferences)  
 nextAttemptId += 1  
}

根据代码清单7-13，makeNewStageAttempt的执行步骤如下。

1）调用StageInfo的fromStage方法（见代码清单7-19）创建新的StageInfo。

2）增加nextAttemptId。

抽象类Stage有两个实现子类，分别为ShuffleMapStage和ResultStage，下面将逐个介绍。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 7.3.1 ResultStage的实现

ResultStage可以使用指定的函数对RDD中的分区进行计算并得出最终结果。ResultStage是最后执行的Stage，此阶段主要进行作业的收尾工作（例如，对各个分区的数据收拢、打印到控制台或写入到HDFS）。ResultStage除继承自父类Stage的属性外，还包括以下属性。

·func：即对RDD的分区进行计算的函数。func是ResultStage的构造器参数，指定了函数的形式必须满足：

(TaskContext, Iterator[\_]) => \_

·partitions：由RDD的各个分区的索引组成的数组。

·\_activeJob：ResultStage处理的ActiveJob。

ResultStage还提供了一些方法（见代码清单7-14）。由于这些方法的实现都很简单，所以把它们一次性列出。

代码清单7-14 ResultStage提供的方法

def activeJob: Option[ActiveJob] = \_activeJob  
def setActiveJob(job: ActiveJob): Unit = {  
 \_activeJob = Option(job)  
}  
def removeActiveJob(): Unit = {  
 \_activeJob = None  
}  
override def findMissingPartitions(): Seq[Int] = {  
 val job = activeJob.get  
 (0 until job.numPartitions).filter(id => !job.finished(id))  
}

这里特别对findMissingPartitions做一些解释：findMissingPartitions用于找出当前Job的所有分区中还没有完成的分区的索引。ResultStage判断一个分区是否完成，是通过ActiveJob的Boolean类型数组finished，因为finished记录了每个分区是否完成。

### 7.3.2 ShuffleMapStage的实现

ShuffleMapStage是DAG调度流程的中间Stage，它可以包括一到多个ShuffleMap-Task，这些ShuffleMapTask将生成用于Shuffle的数据。ShuffleMapStage一般是ResultStage或者其他ShuffleMapStage的前置Stage，ShuffleMapTask则通过Shuffle与下游Stage中的Task串联起来。从ShuffleMapStage的命名可以看出，它将对Shuffle的数据映射到下游Stage的各个分区中。ShuffleMapStage除继承自父类Stage的属性外，还包括以下属性。

·shuffleDep：与ShuffleMapStage相对应的ShuffleDependency。

·\_mapStageJobs：与ShuffleMapStage相关联的ActiveJob的列表。

·\_numAvailableOutputs：ShuffleMapStage可用的map任务的输出数量，这也代表了执行成功的map任务数。

·outputLocs：ShuffleMapStage的各个map任务与其对应的MapStatus列表的映射关系。由于map任务可能会运行多次，因而可能会有多个MapStatus。

ShuffleMapStage还提供了一些方法，分别如下。

·mapStageJobs：即读取\_mapStageJobs的方法。

·addActiveJob与removeActiveJob：向ShuffleMapStage相关联的ActiveJob的列表中添加或删除ActiveJob。实现如代码清单7-15所示。

代码清单7-15 addActiveJob与removeActiveJob

def addActiveJob(job: ActiveJob): Unit = {  
 \_mapStageJobs = job :: \_mapStageJobs  
}  
  
def removeActiveJob(job: ActiveJob): Unit = {  
 \_mapStageJobs = \_mapStageJobs.filter(\_ != job)  
}

·numAvailableOutputs：即读取\_numAvailableOutputs的方法。

·isAvailable：当\_numAvailableOutputs与numPartitions相等时为true。也就是说，ShuffleMapStage的所有分区的map任务都执行成功后，ShuffleMapStage才是可用的。

·findMissingPartitions：找到所有还未执行成功而需要计算的分区。其实现如代码清单7-16所示。

代码清单7-16 findMissingPartitions的实现

override def findMissingPartitions(): Seq[Int] = {  
 val missing = (0 until numPartitions).filter(id => outputLocs(id).isEmpty)  
 assert(missing.size == numPartitions - \_numAvailableOutputs,  
 s"${missing.size} missing, expected ${numPartitions - \_numAvailableOutputs}")  
 missing  
}

·addOutputLoc：当某一分区的任务执行完成后，首先将分区与MapStatus的对应关系添加到outputLocs中，然后将可用的输出数加一。其实现如代码清单7-17所示。

代码清单7-17 添加分区对应的MapStatus

def addOutputLoc(partition: Int, status: MapStatus): Unit = {  
 val prevList = outputLocs(partition)  
 outputLocs(partition) = status :: prevList  
 if (prevList == Nil) {  
 \_numAvailableOutputs += 1  
 }  
}

### 7.3.3 StageInfo

StageInfo用于描述Stage信息，并可以传递给SparkListener。StageInfo包括以下属性。

·stageId：Stage的id。

·attemptId：当前Stage尝试的id。

·name：当前Stage的名称。

·numTasks：当前Stage的Task数量。

·rddInfos：RDD信息（即RDDInfo）的序列。

·parentIds：当前Stage的父亲Stage的身份标识序列。

·details：详细的线程栈信息。

·taskMetrics：Task的度量信息。

·taskLocalityPreferences：类型为Seq[Seq[TaskLocation]]，用于存储任务的本地性偏好。

·submissionTime：DAGScheduler将当前Stage提交给TaskScheduler的时间。

·completionTime：当前Stage中的所有Task完成的时间（即Stage完成的时间）或者Stage被取消的时间。

·failureReason：如果Stage失败了，用于记录失败的原因。

·accumulables：存储了所有聚合器计算的最终值。

StageInfo提供了一个当Stage失败时要调用的方法stageFailed，其实现如代码清单7-18所示。

代码清单7-18 stageFailed的实现

def stageFailed(reason: String) {  
 failureReason = Some(reason)  
 completionTime = Some(System.currentTimeMillis)  
}

根据代码清单7-18，stageFailed方法将保存Stage失败的原因和Stage完成的时间。

在StageInfo的伴生对象中还提供了构建StageInfo的方法，如代码清单7-19所示。

代码清单7-19 StageInfo的fromStage方法

def fromStage(  
 stage: Stage,  
 attemptId: Int,  
 numTasks: Option[Int] = None,  
 taskMetrics: TaskMetrics = null,  
 taskLocalityPreferences: Seq[Seq[TaskLocation]] = Seq.empty  
 ): StageInfo = {  
 val ancestorRddInfos = stage.rdd.getNarrowAncestors.map(RDDInfo.fromRdd)  
 val rddInfos = Seq(RDDInfo.fromRdd(stage.rdd)) ++ ancestorRddInfos  
 new StageInfo(  
 stage.id,  
 attemptId,  
 stage.name,  
 numTasks.getOrElse(stage.numTasks),  
 rddInfos,  
 stage.parents.map(\_.id),  
 stage.details,  
 taskMetrics,  
 taskLocalityPreferences)  
}

根据代码清单7-19，fromStage方法的执行步骤如下。

1）调用当前Stage的RDD的getNarrowAncestors方法（见代码清单7-6），获取RDD的祖先依赖中属于窄依赖的RDD序列。

2）对上一步中获得的RDD序列中的每个RDD，调用RDDInfo伴生对象的fromRdd方法（见代码清单7-12）创建RDDInfo对象。

3）给当前Stage的RDD创建对应的RDDInfo对象，将上一步中创建的所有RDDInfo对象与此RDDInfo对象放入序列rddInfos中。

4）创建StageInfo。

## 7.4 面向DAG的调度器DAGScheduler

DAGScheduler实现了面向DAG的高层次调度，即将DAG中的各个RDD划分到不同的Stage。DAGScheduler可以通过计算将DAG中的一系列RDD划分到不同的Stage，然后构建这些Stage之间的父子关系，最后将每个Stage按照Partition切分为多个Task，并以Task集合（即TaskSet）的形式提交给底层的TaskScheduler。

所有的组件都通过向DAGScheduler投递DAGSchedulerEvent来使用DAGScheduler。DAGScheduler内部的DAGSchedulerEventProcessLoop将处理这些DAGScheduler-Event，并调用DAGScheduler的不同方法。JobListener用于对作业中每个Task执行成功或失败进行监听，JobWaiter实现了JobListener并最终确定作业的成功与失败。在正式介绍DAGScheduler之前，我们先来看看DAGScheduler所依赖的组件DAGSchedulerEventProcessLoop、Job Listener及ActiveJob的实现。

### 7.4.1 JobListener与JobWaiter

JobListener定义了所有Job的监听器的接口规范，其定义如下。

private[spark] trait JobListener {  
 def taskSucceeded(index: Int, result: Any): Unit  
 def jobFailed(exception: Exception): Unit  
}

Job执行成功后将调用JobListener定义的taskSucceeded方法，而在Job失败后调用Job Listener定义的jobFailed方法。

JobListener有JobWaiter和ApproximateActionListener两个实现类。JobWaiter用于等待整个Job执行完毕，然后调用给定的处理函数对返回结果进行处理。Approximate Action Listener只对有单一返回结果的Action（如count()和非并行的reduce()）进行监听。本节将重点介绍JobWaiter，ApproximateActionListener留给感兴趣的读者自行分析。

JobWaiter有以下成员属性。

·dagScheduler：即DAGScheduler，当前JobWaiter等待执行完成的Job的调度者。

·jobId：当前JobWaiter等待执行完成的Job的身份标识。

·totalTasks：等待完成的Job包括的Task数量。

·resultHandler：执行结果的处理器。resultHandler是JobWaiter构造器的一个函数参数，参数定义为：resultHandler:(Int,T)=>Unit)。

·finishedTasks：等待完成的Job中已经完成的Task数量。

·jobPromise：类型为scala.concurrent.Promise。jobPromise用来代表Job完成后的结果。如果totalTasks等于零，说明没有Task需要执行，此时jobPromise将被直接设置为Success。

此外，JobWaiter定义了以下方法。

·jobFinished：Job是否已经完成。jobFinished的实现如下。

　　def jobFinished: Boolean = jobPromise.isCompleted

·completionFuture：jobPromise的Future。completionFuture的实现如下。

　　def completionFuture: Future[Unit] = jobPromise.future

·cancel：取消对Job的执行。cancel的实现如下。

　　def cancel() {  
 dagScheduler.cancelJob(jobId)  
　　}

根据上述代码，cancel实际调用了DAGScheduler的cancelJob方法。DAGScheduler的cancelJob方法将使得Job的所有任务被取消，并向外抛出SparkException。

·taskSucceeded：JobWaiter重写的在特质JobListener中定义的方法，实现如代码清单7-20所示。

代码清单7-20 taskSucceeded的实现

override def taskSucceeded(index: Int, result: Any): Unit = {  
 synchronized {  
 resultHandler(index, result.asInstanceOf[T])  
 }  
 if (finishedTasks.incrementAndGet() == totalTasks) {  
 jobPromise.success(())  
 }  
}

根据代码清单7-20，taskSucceeded的执行步骤如下。

1）调用resultHandler函数来处理Job中每个Task的执行结果。

2）增加已完成的Task数量。

3）如果Job的所有Task都已经完成，那么将jobPromise设置为Success。

·jobFailed：JobWaiter重写的由JobListener定义的方法，实现如代码清单7-21所示。

代码清单7-21 jobFailed的实现

override def jobFailed(exception: Exception): Unit = {  
 if (!jobPromise.tryFailure(exception)) {  
 logWarning("Ignore failure", exception)  
 }  
}

根据代码清单7-21，jobFailed实际只是将jobPromise设置为Failure。

### 7.4.2 ActiveJob详解

ActiveJob用来表示已经激活的Job，即被DAGScheduler接收处理的Job。ActiveJob有以下属性。

·jobId：Job的身份标识。

·finalStage：Job的最下游Stage。

·callSite：应用程序调用栈。

·listener：监听当前Job的JobListener。

·properties：包含了当前Job的调度、Job group、描述等属性的Properties。

·numPartitions：当前Job的分区数量。如果finalStage为ResultStage，那么此属性等于ResultStage的partitions属性的长度。如果finalStage为ShuffleMapStage，那么此属性等于ShuffleMapStage的rdd的partitions属性的长度。

·finished：Boolean类型的数组，每个数组索引代表一个分区的任务是否执行完成。

·numFinished：当前Job的所有任务中已完成任务的数量。

### 7.4.3 DAGSchedulerEventProcessLoop的简要介绍

DAGSchedulerEventProcessLoop是DAGScheduler内部的事件循环处理器，用于处理DAGSchedulerEvent类型的事件。由于DAGSchedulerEventProcessLoop的实现与Live ListenerBus非常相似，而且其实现更为简单，所以我们将省略对DAGSchedulerEvent ProcessLoop的原理分析，只关注DAGSchedulerEventProcessLoop能够处理哪些事件。DAGS chedulerEventProcessLoop能够处理的事件类型如代码清单7-22所示。

代码清单7-22 DAGSchedulerEventProcessLoop处理的消息类型

private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {  
 case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>  
 dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)  
 case MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>  
 dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener, properties)  
 case StageCancelled(stageId) =>  
 dagScheduler.handleStageCancellation(stageId)  
 case JobCancelled(jobId) =>  
 dagScheduler.handleJobCancellation(jobId)  
 case JobGroupCancelled(groupId) =>  
 dagScheduler.handleJobGroupCancelled(groupId)  
 case AllJobsCancelled =>  
 dagScheduler.doCancelAllJobs()  
 case ExecutorAdded(execId, host) =>  
 dagScheduler.handleExecutorAdded(execId, host)  
 case ExecutorLost(execId, reason) =>  
 val filesLost = reason match {  
 case SlaveLost(\_, true) => true  
 case \_ => false  
 }  
 dagScheduler.handleExecutorLost(execId, filesLost)  
 case BeginEvent(task, taskInfo) =>  
 dagScheduler.handleBeginEvent(task, taskInfo)  
 case GettingResultEvent(taskInfo) =>  
 dagScheduler.handleGetTaskResult(taskInfo)  
 case completion: CompletionEvent =>  
 dagScheduler.handleTaskCompletion(completion)  
 case TaskSetFailed(taskSet, reason, exception) =>  
 dagScheduler.handleTaskSetFailed(taskSet, reason, exception)  
 case ResubmitFailedStages =>  
 dagScheduler.resubmitFailedStages()  
}

### 7.4.4 DAGScheduler的组成

DAGScheduler本身包含很多成员，了解这些成员是深入理解DAGScheduler的前提。DAGScheduler的成员包括以下这些。

·sc：SparkContext。

·taskScheduler：TaskScheduler的引用。

·listenerBus：LiveListenerBus。

·mapOutputTracker：MapOutputTrackerMaster。

·blockManagerMaster：BlockManagerMaster。

·env：SparkEnv。

·clock：时钟对象。类型为SystemClock，提供了获取系统当前毫秒时间的getTime Millis方法和使得当前线程睡眠直到指定时间为止的waitTillTime方法。SystemClock的实现很简单，所以这里不详细说明，感兴趣的读者可自行了解。

·metricsSource：有关DAGScheduler的度量源（即DAGSchedulerSource）。

·nextJobId：类型为AtomicInteger，用于生成下一个Job的身份标识（即JobId）。

·numTotalJobs：总共提交的作业数量。numTotalJobs实际读取了nextJobId的当前值。

·nextStageId：类型为AtomicInteger，用于生成下一个Stage的身份标识（即StageId）。

·jobIdToStageIds：用于缓存JobId与StageId之间的映射关系。由于jobIdToStageIds的类型为HashMap[Int,HashSet[Int]]，所以Job与Stage之间是一对多的关系。

·stageIdToStage：用于缓存StageId与Stage之间的映射关系。

·shuffleIdToMapStage：用于缓存Shuffle的身份标识（即shuffleId）与ShuffleMapStage之间的映射关系。

·jobIdToActiveJob：Job的身份标识与激活的Job（即ActiveJob）之间的映射关系。

·waitingStages：处于等待状态的Stage集合。

·runningStages：处于运行状态的Stage集合。

·failedStages：处于失败状态的Stage集合。

·activeJobs：所有激活的Job的集合。

·cacheLocs：缓存每个RDD的所有分区的位置信息。cacheLocs的数据类型是HashMap[Int,IndexedSeq[Seq[TaskLocation]]]，所以每个RDD的分区按照分区号作为索引存储到IndexedSeq。由于RDD的每个分区作为一个Block以及存储体系的复制因素，因此RDD的每个分区的Block可能存在于多个节点的BlockManager上，RDD每个分区的位置信息为TaskLocation的序列。

·failedEpoch：当检测到一个节点出现故障时，会将执行失败的Executor和MapOutputTracker当前的纪元添加到failedEpoch。此外，还使用failedEpoch忽略迷失的ShuffleMapTask的结果。

·outputCommitCoordinator：SparkEnv的子组件OutputCommitCoordinator。

·closureSerializer：SparkEnv中创建的closureSerializer。

·disallowStageRetryForTest：在测试过程中，当发生FetchFailed时，使用此属性则不会对Stage进行重试。可以通过spark.test.noStageRetry进行属性配置，默认为false。

·messageScheduler：只有一个线程的ScheduledThreadPoolExecutor，创建的线程以dag-scheduler-message开头。messageScheduler的职责是对失败的Stage进行重试。

·eventProcessLoop：DAGSchedulerEventProcessLoop。

### 7.4.5 DAGScheduler提供的常用方法

上一节我们对DAGScheduler中的各个成员有了基本的了解，这一节来介绍DAGS-cheduler提供的常用方法。

1.clearCacheLocs

clearCacheLocs（见代码清单7-23）用于清空cacheLocs中缓存的各个RDD的所有分区的位置信息。

代码清单7-23 清空cacheLocs

private def clearCacheLocs(): Unit = cacheLocs.synchronized {  
 cacheLocs.clear()  
}

2.updateJobIdStageIdMaps

updateJobIdStageIdMaps方法（见代码清单7-24）用来更新Job的身份标识与Stage及其所有祖先的映射关系。

代码清单7-24 updateJobIdStageIdMaps的实现

private def updateJobIdStageIdMaps(jobId: Int, stage: Stage): Unit = {  
 @tailrec  
 def updateJobIdStageIdMapsList(stages: List[Stage]) {  
 if (stages.nonEmpty) {  
 val s = stages.head  
 s.jobIds += jobId  
 jobIdToStageIds.getOrElseUpdate(jobId, new HashSet[Int]()) += s.id  
 val parentsWithoutThisJobId = s.parents.filter { ! \_.jobIds.contains(jobId) }  
 updateJobIdStageIdMapsList(parentsWithoutThisJobId ++ stages.tail)  
 }  
 }  
 updateJobIdStageIdMapsList(List(stage))  
}

根据代码清单7-24，updateJobIdStageIdMaps方法对Stage及Stage的所有祖先Stage进行如下处理。

1）将jobId添加到每个Stage的jobIds。

2）将jobId和与每个Stage的id之间的映射关系更新到jobIdToStageIds。

3.activeJobForStage

activeJobForStage方法（见代码清单7-25）用于找到Stage的所有已经激活的Job的身份标识。

代码清单7-25 activeJobForStage的实现

private def activeJobForStage(stage: Stage): Option[Int] = {  
 val jobsThatUseStage: Array[Int] = stage.jobIds.toArray.sorted  
 jobsThatUseStage.find(jobIdToActiveJob.contains)  
}

4.getCacheLocs

getCacheLocs方法（见代码清单7-26）用于获取RDD各个分区的TaskLocation序列。

代码清单7-26 获取RDD各个分区的TaskLocation序列

private[scheduler]  
def getCacheLocs(rdd: RDD[\_]): IndexedSeq[Seq[TaskLocation]] = cacheLocs.synchronized {  
 if (!cacheLocs.contains(rdd.id)) {  
 val locs: IndexedSeq[Seq[TaskLocation]] = if (rdd.getStorageLevel == StorageLevel.NONE) {  
 IndexedSeq.fill(rdd.partitions.length)(Nil)  
 } else {  
 val blockIds =  
 rdd.partitions.indices.map(index => RDDBlockId(rdd.id, index)).toArray[BlockId]  
 blockManagerMaster.getLocations(blockIds).map { bms => // 获取每个RDDBlockId所存储的位置信息  
 bms.map(bm => TaskLocation(bm.host, bm.executorId))  
 }  
 }  
 cacheLocs(rdd.id) = locs  
 }  
 cacheLocs(rdd.id)  
}

根据代码清单7-26，getCacheLocs方法的执行步骤如下。

1）如果cacheLocs中不包含RDD对应的IndexedSeq[Seq[TaskLocation]]，那么执行以下操作：

①如果RDD的存储级别为NONE，那么构造一个空的IndexedSeq[Seq[TaskLocation]]返回。

②如果RDD的存储级别不为NONE，则首先构造RDD各个分区的RDDBlockId数组，然后调用BlockManagerMaster的getLocations方法获取数组中每个RDDBlockId所存储的位置信息（即BlockManager的身份标识BlockManagerId）序列，并封装为TaskLocation。

③将RDD的id与第①或第②步得到的IndexedSeq[Seq[TaskLocation]]之间的映射关系放入cacheLocs。

2）返回cacheLocs中缓存的RDD对应的IndexedSeq[Seq[TaskLocation]]。

5.getPreferredLocsInternal

getPreferredLocsInternal方法（见代码清单7-27）用于获取RDD的指定分区的偏好位置。

代码清单7-27 getPreferredLocsInternal的实现

private def getPreferredLocsInternal(  
 rdd: RDD[\_],  
 partition: Int,  
 visited: HashSet[(RDD[\_], Int)]): Seq[TaskLocation] = {  
 if (!visited.add((rdd, partition))) { // 避免对RDD的指定分区的重复访问  
 return Nil  
 }  
 val cached = getCacheLocs(rdd)(partition) // 获取RDD的指定分区的位置信息  
 if (cached.nonEmpty) {  
 return cached  
 }  
 val rddPrefs = rdd.preferredLocations(rdd.partitions(partition)).toList//获取RDD指定分区的偏好位置  
 if (rddPrefs.nonEmpty) {  
 return rddPrefs.map(TaskLocation(\_))  
 }  
 rdd.dependencies.foreach { // 以窄依赖的RDD的同一分区的偏好位置作为当前RDD的此分区的偏好位置  
 case n: NarrowDependency[\_] =>  
 for (inPart <- n.getParents(partition)) {  
 val locs = getPreferredLocsInternal(n.rdd, inPart, visited)  
 if (locs != Nil) {  
 return locs  
 }  
 }  
 case \_ =>  
 }  
 Nil  
}

根据代码清单7-27，getPreferredLocsInternal的执行步骤如下。

1）避免对RDD的指定分区的重复访问。

2）调用getCacheLocs方法，从cacheLocs中获取RDD的指定分区的位置信息。如果能够获取到位置信息，则返回获取到的TaskLocation序列，否则进入下一步。

3）调用RDD的preferredLocations方法获取指定分区的偏好位置信息，并封装为TaskLocation后返回。如果未能获取到位置信息，则进入下一步。

4）遍历RDD的窄依赖，获取窄依赖中RDD的同一分区的偏好位置，以首先找到的偏好位置作为当前RDD的偏好位置返回。

5）如果以上步骤都未能获取到RDD的指定分区的偏好位置，则返回Nil。

6.getPreferredLocs

DAGScheduler的getPreferredLocs方法（见代码清单7-28）用于获取RDD的指定分区的偏好位置。

代码清单7-28 获取RDD的指定分区的偏好位置

private[spark]  
def getPreferredLocs(rdd: RDD[\_], partition: Int): Seq[TaskLocation] = {  
 getPreferredLocsInternal(rdd, partition, new HashSet)  
}

根据代码清单7-28，getPreferredLocs实际调用了getPreferredLocsInternal方法。

7.handleExecutorAdded

DAGScheduler的handleExecutorAdded方法（见代码清单7-29）用于将Executor的身份标识从failedEpoch中移除。

代码清单7-29 handleExecutorAdded的实现

private[scheduler] def handleExecutorAdded(execId: String, host: String) {  
 if (failedEpoch.contains(execId)) {  
 logInfo("Host added was in lost list earlier: " + host)  
 failedEpoch -= execId  
 }  
}

8.executorAdded

DAGScheduler的executorAdded方法（见代码清单7-30）用于向DAGScheduler的DAGSch-eduler-EventProcessLoop投递ExecutorAdded事件。

代码清单7-30 executorAdded的实现

def executorAdded(execId: String, host: String): Unit = {  
 eventProcessLoop.post(ExecutorAdded(execId, host))  
}

根据代码清单7-22，我们知道DAGSchedulerEventProcessLoop在接收到Executor-Added事件后将调用DAGScheduler的handleExecutorAdded方法。

有了对DAGScheduler中各个成员和常用方法的了解，现在来介绍DAGScheduler的调度流程。通过对DAGScheduler的调度流程的分析，我们能进一步认识DAGScheduler提供的其他方法的实现。

### 7.4.6 DAGScheduler与Job的提交

1.提交Job

用户提交的Job首先会被转换为一系列RDD，然后才交给DAGScheduler进行处理。DAGScheduler的runJob方法是这一过程的入口，其实现如代码清单7-31所示。

代码清单7-31 DAGScheduler的调度入口

def runJob[T, U](  
 rdd: RDD[T],  
 func: (TaskContext, Iterator[T]) => U,  
 partitions: Seq[Int],  
 callSite: CallSite,  
 resultHandler: (Int, U) => Unit,  
 properties: Properties): Unit = {  
 val start = System.nanoTime  
 val waiter = submitJob(rdd, func, partitions, callSite, resultHandler, properties) // 提交Job  
 val awaitPermission = null.asInstanceOf[scala.concurrent.CanAwait]  
 waiter.completionFuture.ready(Duration.Inf)(awaitPermission) // 利用JobWaiter等待Job处理完毕  
 waiter.completionFuture.value.get match { // JobWaiter监听到Job的处理结果，进行进一步处理  
 case scala.util.Success(\_) =>  
 logInfo("Job %d finished: %s, took %f s".format  
 (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))  
 case scala.util.Failure(exception) =>  
 logInfo("Job %d failed: %s, took %f s".format  
 (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))  
 val callerStackTrace = Thread.currentThread().getStackTrace.tail  
 exception.setStackTrace(exception.getStackTrace ++ callerStackTrace)  
 throw exception  
 }  
}

根据代码清单7-31，runJob的执行步骤如下。

1）生成运行Job的启动时间start。

2）调用submitJob方法提交Job。由于执行Job的过程是异步的，因此submitJob方法将立即返回JobWaiter对象。

3）利用JobWaiter等待Job处理完毕。如果Job执行成功，根据处理结果打印相应的日志。如果Job执行失败，除打印日志外，还将抛出引起Job失败的异常信息。

DAGScheduler的submitJob方法用于提交Job，其实现如代码清单7-32所示。

代码清单7-32 提交Job

def submitJob[T, U](  
 rdd: RDD[T],  
 func: (TaskContext, Iterator[T]) => U,  
 partitions: Seq[Int],  
 callSite: CallSite,  
 resultHandler: (Int, U) => Unit,  
 properties: Properties): JobWaiter[U] = {  
 val maxPartitions = rdd.partitions.length // 获取当前Job的最大分区数maxPartitions  
 partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>  
 throw new IllegalArgumentException(  
 "Attempting to access a non-existent partition: " + p + ". " +  
 "Total number of partitions: " + maxPartitions)  
 }  
 val jobId = nextJobId.getAndIncrement() // 生成下一个Job的jobId  
 if (partitions.size == 0) {  
 return new JobWaiter[U](this, jobId, 0, resultHandler)  
 }  
 assert(partitions.size > 0)  
 val func2 = func.asInstanceOf[(TaskContext, Iterator[\_]) => \_]  
 // 创建等待Job完成的JobWaiter  
 val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)  
 // 向DAGSchedulerEventProcessLoop发送JobSubmitted事件  
 eventProcessLoop.post(JobSubmitted(   
 jobId, rdd, func2, partitions.toArray, callSite, waiter,  
 SerializationUtils.clone(properties)))  
 waiter  
}

根据代码清单7-32，submitJob的处理步骤如下。

1）调用RDD的partitions方法来获取当前Job的最大分区数maxPartitions。如果检查到有不存在的分区，那么抛出IllegalArgumentException。

2）生成下一个Job的jobId。

3）如果Job的分区数量等于0，则创建一个totalTasks属性为0的JobWaiter并返回。根据JobWaiter的实现，JobWaiter的jobPromise将被设置为Success。

4）如果Job的分区数量大于0，则创建真正等待Job执行完成的JobWaiter。

5）向eventProcessActor（即DAGSchedulerEventProcessLoop）发送JobSubmitted事件。

6）返回JobWaiter。

2.处理Job的提交

根据DAGSchedulerEventProcessLoop的实现（见代码清单7-22），我们知道DAGSche-dulerEventProcessLoop接收到JobSubmitted事件后，将调用DAGScheduler的handleJob-Submitted方法。handleJobSubmitted的实现如代码清单7-33所示。

代码清单7-33 handleJobSubmitted的实现

private[scheduler] def handleJobSubmitted(jobId: Int,  
 finalRDD: RDD[\_],  
 func: (TaskContext, Iterator[\_]) => \_,  
 partitions: Array[Int],  
 callSite: CallSite,  
 listener: JobListener,  
 properties: Properties) {  
 var finalStage: ResultStage = null  
 try {  
 //创建ResultStage  
 finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)  
 } catch {  
 case e: Exception =>  
 logWarning("Creating new stage failed due to exception - job: " + jobId, e)  
 listener.jobFailed(e)  
 return  
 }  
  
 // 创建ActiveJob  
 val job = new ActiveJob(jobId, finalStage, callSite, listener, properties)   
 clearCacheLocs()  
  
 val jobSubmissionTime = clock.getTimeMillis() // 生成Job提交的时间  
 jobIdToActiveJob(jobId) = job  
 activeJobs += job  
 finalStage.setActiveJob(job)  
 val stageIds = jobIdToStageIds(jobId).toArray  
 val stageInfos = stageIds.flatMap(id => stageIdToStage.get(id).map(\_.latestInfo))  
 listenerBus.post(  
 SparkListenerJobStart(job.jobId, jobSubmissionTime, stageInfos, properties))  
 submitStage(finalStage) // 提交ResultStage  
}

根据代码清单7-33，handleJobSubmitted的执行过程如下。

1）调用createResultStage方法（见代码清单7-34）创建ResultStage。创建Stage的过程可能发生异常（比如运行在HadoopRDD上的任务所依赖的底层HDFS文件被删除了），当异常发生时需要主动调用JobWaiter的jobFailed方法。createResultStage方法在创建ResultStage的过程中会引起创建一系列Stage的连锁反应，Job与这些Stage的关系将被放入jobIdToStageIds中。

2）创建ActiveJob。

3）调用clearCacheLocs方法（见代码清单7-23）清空cacheLocs。

4）生成Job提交的时间。

5）将jobId与刚创建的ActiveJob之间的对应关系放入jobIdToActiveJob中。

6）将刚创建的ActiveJob放入activeJobs集合中。

7）使ResultStage的\_activeJob属性持有刚创建的ActiveJob。

8）获取当前Job的所有Stage对应的StageInfo（即数组stageInfos）。

9）向LiveListenerBus投递SparkListenerJobStart事件，进而引发所有关注此事件的监听器执行相应的操作。

10）调用submitStage方法（见代码清单7-40）提交ResultStage。

### 7.4.7 构建Stage

在Spark中，一个Job可能被划分为一到多个Stage。各个Stage之间存在着依赖关系，下游的Stage依赖于上游的Stage。Job中所有Stage的提交过程包括反向驱动与正向提交。

所谓Stage的反向驱动，就是从最下游的ResultStage开始，由ResultStage驱动所有父Stage的执行，父Stage又驱动祖父Stage的执行，这个驱动过程不断向祖先方向传递，直到最上游的Stage为止，整个驱动过程可以说是“长江后浪推前浪”。而正向提交，就是前代Stage先于后代Stage将Task提交给TaskScheduler，祖父Stage先于父Stage将Task提交给TaskScheduler，父Stage先于子Stage将Task提交给TaskScheduler，ResultStage最后一个将Task提交给TaskScheduler，整个提交过程可以说是“代代相传”。

本节将详细介绍Stage依赖关系的构建过程。

1.创建ResultStage

DAGScheduler的createResultStage方法用于创建ResultStage，其实现如代码清单7-34所示。

代码清单7-34 创建ResultStage

private def createResultStage(  
 rdd: RDD[\_],  
 func: (TaskContext, Iterator[\_]) => \_,  
 partitions: Array[Int],  
 jobId: Int,  
 callSite: CallSite): ResultStage = {  
 val parents = getOrCreateParentStages(rdd, jobId) // 获取所有父Stage的列表  
 val id = nextStageId.getAndIncrement() // 生成ResultStage的身份标识  
 // 创建ResultStage  
 val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)   
 stageIdToStage(id) = stage  
 updateJobIdStageIdMaps(jobId, stage)  
 stage  
}

根据代码清单7-34，createResultStage的处理步骤如下。

1）调用getOrCreateParentStages方法（见代码清单7-35）获取所有父Stage的列表，父Stage主要是宽依赖（ShuffleDependency）对应的Stage，此列表内的Stage包含以下几种：

①当前RDD的直接或间接的依赖是ShuffleDependency且已经注册过的Stage。

②当前RDD的直接或间接的依赖是ShuffleDependency且没有注册过Stage的。对于这种ShuffleDependency，则根据ShuffleDependency中的RDD，找到它的直接或间接的依赖是ShuffleDependency且没有注册过Stage的所有ShuffleDependency，为它们创建并注册Stage。

③当前RDD的直接或间接的依赖是ShuffleDependency且没有注册过Stage的。为此ShuffleDependency创建并注册Stage。

2）生成Stage的身份标识，并创建ResultStage。

3）将ResultStage注册到stageIdToStage中。

4）调用updateJobIdStageIdMaps方法（见代码清单7-24）更新Job的身份标识与ResultStage及其所有祖先的映射关系。

2.获取或创建父Stage列表

Spark中的Job可能包含一到多个Stage，这些Stage的划分是从ResultStage开始，从后往前边划分边创建的。DAGScheduler的getOrCreateParentStages方法（见代码清单7-35）用于获取或者创建给定RDD的所有父Stage，这些Stage将被分配给jobId对应的Job。

代码清单7-35 获取或创建给定RDD的所有父Stage

private def getOrCreateParentStages(rdd: RDD[\_], firstJobId: Int): List[Stage] = {  
 getShuffleDependencies(rdd).map { shuffleDep =>  
 getOrCreateShuffleMapStage(shuffleDep, firstJobId)  
 }.toList  
}

根据代码清单7-35，getOrCreateParentStages的处理步骤如下。

1）调用DAGScheduler的getShuffleDependencies方法（见代码清单7-36）获取RDD的所有ShuffleDependency的序列，逐个访问每个RDD及其依赖的非Shuffle的RDD，获取所有非Shuffle的RDD的ShuffleDependency依赖。

2）调用DAGScheduler的getOrCreateShuffleMapStage方法（见代码清单7-37）为每一个ShuffleDependency获取或者创建对应的ShuffleMapStage，并返回得到的Shuffle-MapStage列表。

代码清单7-36 获取RDD的所有ShuffleDependency的序列

private[scheduler] def getShuffleDependencies(  
 rdd: RDD[\_]): HashSet[ShuffleDependency[\_, \_, \_]] = {  
 val parents = new HashSet[ShuffleDependency[\_, \_, \_]]  
 val visited = new HashSet[RDD[\_]]  
 val waitingForVisit = new Stack[RDD[\_]]  
 waitingForVisit.push(rdd)  
 while (waitingForVisit.nonEmpty) {  
 val toVisit = waitingForVisit.pop()  
 if (!visited(toVisit)) {  
 visited += toVisit  
 toVisit.dependencies.foreach {  
 case shuffleDep: ShuffleDependency[\_, \_, \_] =>  
 parents += shuffleDep  
 case dependency =>  
 waitingForVisit.push(dependency.rdd)  
 }  
 }  
 }  
 parents  
}

getOrCreateShuffleMapStage方法（见代码清单7-37）用于获取或者创建ShuffleMapStage。

代码清单7-37 获取或创建ShuffleMapStage

private def getOrCreateShuffleMapStage(  
 shuffleDep: ShuffleDependency[\_, \_, \_],  
 firstJobId: Int): ShuffleMapStage = {  
 shuffleIdToMapStage.get(shuffleDep.shuffleId) match {  
 case Some(stage) =>   
 stage // 如果已经创建了ShuffleDependency对应的ShuffleMapStage，则直接返回此ShuffleMapStage  
  
 case None =>  
 getMissingAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>  
 if (!shuffleIdToMapStage.contains(dep.shuffleId)) {  
 createShuffleMapStage(dep, firstJobId) // 创建当前Stage的上游ShuffleMapStage  
 }  
 }  
 createShuffleMapStage(shuffleDep, firstJobId) // 为当前ShuffleDependency创建Shuffle MapStage  
 }  
}

根据代码清单7-37，getOrCreateShuffleMapStage的处理步骤如下。

1）如果已经创建了ShuffleDependency对应的ShuffleMapStage，则直接返回此ShuffleMapStage。

2）否则调用getMissingAncestorShuffleDependencies方法（见代码清单7-38）找到所有还未创建过ShuffleMapStage的祖先ShuffleDependency，并调用createShuffleMapStage方法（见代码清单7-39）创建ShuffleMapStage并注册。最后还会为当前ShuffleDependency调用方法createShuffleMapStage创建ShuffleMapStage并注册。

代码清单7-38 找到所有还未创建过ShuffleMapStage的祖先ShuffleDependency

private def getMissingAncestorShuffleDependencies(  
 rdd: RDD[\_]): Stack[ShuffleDependency[\_, \_, \_]] = {  
 val ancestors = new Stack[ShuffleDependency[\_, \_, \_]]  
 val visited = new HashSet[RDD[\_]]  
 val waitingForVisit = new Stack[RDD[\_]]  
 waitingForVisit.push(rdd)  
 while (waitingForVisit.nonEmpty) {  
 val toVisit = waitingForVisit.pop()  
 if (!visited(toVisit)) {  
 visited += toVisit  
 getShuffleDependencies(toVisit).foreach { shuffleDep =>  
 if (!shuffleIdToMapStage.contains(shuffleDep.shuffleId)) {  
 ancestors.push(shuffleDep)  
 waitingForVisit.push(shuffleDep.rdd)  
 } // Otherwise, the dependency and its ancestors have already been registered.  
 }  
 }  
 }  
 ancestors  
}

createShuffleMapStage用于创建ShuffleMapStage，其实现如代码清单7-39所示。

代码清单7-39 创建ShuffleMapStage

def createShuffleMapStage(shuffleDep: ShuffleDependency[\_, \_, \_], jobId: Int): ShuffleMapStage = {  
 val rdd = shuffleDep.rdd  
 val numTasks = rdd.partitions.length  
 val parents = getOrCreateParentStages(rdd, jobId)  
 val id = nextStageId.getAndIncrement()  
 val stage = new ShuffleMapStage(id, rdd, numTasks, parents, jobId, rdd.creationSite, shuffleDep)  
  
 stageIdToStage(id) = stage  
 shuffleIdToMapStage(shuffleDep.shuffleId) = stage  
 updateJobIdStageIdMaps(jobId, stage)  
  
 if (mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {  
 val serLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.shuffleId)  
 val locs = MapOutputTracker.deserializeMapStatuses(serLocs)  
 (0 until locs.length).foreach { i =>  
 if (locs(i) ne null) {  
 stage.addOutputLoc(i, locs(i)) // 添加每个分区的输出位置信息  
 }  
 }  
 } else {  
 logInfo("Registering RDD " + rdd.id + " (" + rdd.getCreationSite + ")")  
 mapOutputTracker.registerShuffle(shuffleDep.shuffleId, rdd.partitions.length)  
 }  
 stage  
}

根据代码清单7-39，createShuffleMapStage的执行步骤如下。

1）创建ShuffleMapStage，具体如下。

①获取ShuffleDependency的rdd属性，作为将要创建的ShuffleMapStage的rdd。

②调用rdd（即RDD）的partitions方法得到rdd的分区数组，此分区数组的长度即为要创建的ShuffleMapStage的numTasks（Task数量）。这说明了map任务数量与RDD的各个分区一一对应。

③调用getOrCreateParentStages方法（见代码清单7-35）获得要创建ShuffleMapStage的所有父Stage（即parents）。

④生成将要创建的ShuffleMapStage的身份标识。

⑤创建ShuffleMapStage。

2）更新刚创建的ShuffleMapStage的映射关系。具体如下：

①将新创建的ShuffleMapStage的身份标识与ShuffleMapStage的映射关系放入stageId-ToStage中。

②将shuffleId与ShuffleMapStage的映射关系放入shuffleIdToMapStage中。

③调用updateJobIdStageIdMaps方法（见代码清单7-24）更新Job的身份标识与Shuffle-MapStage及其所有祖先的映射关系。

3）调用mapOutputTracker（mapOutputTracker这个变量名容易引起误解，其实际类型为MapOutputTrackerMaster）的containsShuffle方法（见代码清单5-70）查看是否已经存在shuffleId对应的MapStatus。如果MapOutputTrackerMaster中未缓存shuffleId对应的MapStatus，那么调用MapOutputTrackerMaster的registerShuffle方法（见代码清单5-71）注册shuffleId与对应的MapStatus的映射关系。如果MapOutputTrackerMaster中缓存了shuffleId对应的MapStatus，将执行以下操作：

①调用MapOutputTrackerMaster的getSerializedMapOutputStatuses（见代码清单5-65）从MapOutputTrackerMaster获取对MapStatus序列化后的字节数组。

②调用MapOutputTrackerMaster的deserializeMapStatuses方法对上一步获得的字节数组进行反序列化，得到MapStatus数组。

③调用ShuffleMapStage的addOutputLoc方法更新ShuffleMapStage的outputLocs。

小贴示： 因为Stage可以重试，所以当前的Stage可能在之前已经执行过。在上一次执行过程中，部分map任务可能执行成功，MapOutputTrackerMaster中将缓存这些成功的map任务的MapStatus，因而当前Stage只需要从MapOutputTrackerMaster中复制这些MapStatus即可，从而避免重新计算生成这些数据。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 7.4.8 提交ResultStage

handleJobSubmitted方法（见代码清单7-33）中处理Job提交的最后一步是调用submit-Stage方法（见代码清单7-40）提交ResultStage。

代码清单7-40 提交Stage

private def submitStage(stage: Stage) {  
 val jobId = activeJobForStage(stage) // 获取当前Stage对应的Job的ID  
 if (jobId.isDefined) {  
 logDebug("submitStage(" + stage + ")")  
 if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) { //当前Stage未提交  
 val missing = getMissingParentStages(stage).sortBy(\_.id)  
 logDebug("missing: " + missing)  
 if (missing.isEmpty) { // 不存在未提交的父Stage，那么提交当前Stage所有未提交的Task  
 logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")  
 submitMissingTasks(stage, jobId.get)  
 } else {  
 for (parent <- missing) { // 存在未提交的父Stage，那么逐个提交它们  
 submitStage(parent)  
 }  
 waitingStages += stage  
 }  
 }  
 } else { // 终止依赖于当前Stage的所有Job  
 abortStage(stage, "No active job for stage " + stage.id, None)  
 }  
}

根据代码清单7-40，submitStage方法的执行步骤如下。

1）调用activeJobForStage方法（见代码清单7-25）找到使用当前Stage的所有ActiveJob的身份标识。

2）如果存在Stage对应的ActiveJob的身份标识并且当前Stage还未提交（即waiting-Stages、runningStages、failedStages中都不包含当前Stage），则执行以下操作：

①调用getMissingParentStages方法（见代码清单7-41）获取当前Stage的所有未提交的父Stage。

②如果不存在未提交的父Stage，那么调用submitMissingTasks方法（见代码清单7-42）提交当前Stage所有未提交的Task。

③如果存在未提交的父Stage，那么将多次调用submitStage方法提交所有未提交的父Stage，并且将当前Stage加入waitingStages集合中（这表示当前Stage必须等待所有父Stage执行完成）。

3）如果不存在Stage对应的ActiveJob的身份标识，则调用abortStage方法终止依赖于当前Stage的所有Job。由于abortStage方法的实现并不是本节最重要的内容，所以留给感兴趣的读者自行研究。

代码清单7-41 获取Stage的所有未提交的父Stage

private def getMissingParentStages(stage: Stage): List[Stage] = {  
 val missing = new HashSet[Stage]  
 val visited = new HashSet[RDD[\_]]  
 val waitingForVisit = new Stack[RDD[\_]]  
 def visit(rdd: RDD[\_]) {  
 if (!visited(rdd)) {  
 visited += rdd  
 val rddHasUncachedPartitions = getCacheLocs(rdd).contains(Nil)  
 if (rddHasUncachedPartitions) {  
 for (dep <- rdd.dependencies) {  
 dep match {  
 case shufDep: ShuffleDependency[\_, \_, \_] =>  
 val mapStage = getOrCreateShuffleMapStage(shufDep, stage.firstJobId)  
 if (!mapStage.isAvailable) {  
 missing += mapStage  
 }  
 case narrowDep: NarrowDependency[\_] =>  
 waitingForVisit.push(narrowDep.rdd)  
 }  
 }  
 }  
 }  
 }  
 waitingForVisit.push(stage.rdd)  
 while (waitingForVisit.nonEmpty) {  
 visit(waitingForVisit.pop())  
 }  
 missing.toList  
}

根据代码清单7-41，判断Stage的未提交父Stage的条件如下。

·Stage的RDD的分区中存在没有对应TaskLocation序列的分区，即调用getCache-Locs方法（见代码清单7-26）获取不到某个分区的TaskLocation序列，则说明当前Stage的某个上游ShuffleMapStage的某个分区任务未执行。

·Stage的上游ShuffleMapStage不可用（即调用ShuffleMapStage的isAvailable方法返回false）。

### 7.4.9 提交还未计算的Task

提交Task的入口是submitMissingTasks方法（见代码清单7-42），此方法在Stage没有不可用的父Stage时，提交当前Stage还未提交的任务。

代码清单7-42 提交还未计算的任务

private def submitMissingTasks(stage: Stage, jobId: Int) {  
 stage.pendingPartitions.clear()  
 // 找出当前Stage的所有分区中还没有完成计算的分区的索引  
 val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()  
 // 获取ActiveJob的properties。properties包含了当前Job的调度、group、描述等属性信息  
 val properties = jobIdToActiveJob(jobId).properties  
 runningStages += stage  
 stage match { // 启动对当前Stage的输出提交到HDFS的协调  
 case s: ShuffleMapStage =>  
 outputCommitCoordinator.stageStart(stage = s.id, maxPartitionId = s.num Partitions - 1)  
 case s: ResultStage =>  
 outputCommitCoordinator.stageStart(  
 stage = s.id, maxPartitionId = s.rdd.partitions.length - 1)  
 }  
 val taskIdToLocations: Map[Int, Seq[TaskLocation]] = try {  
 stage match { // 获取还没有完成计算的每一个分区的偏好位置  
 case s: ShuffleMapStage =>  
 partitionsToCompute.map { id => (id, getPreferredLocs(stage.rdd, id))}.toMap  
 case s: ResultStage =>  
 partitionsToCompute.map { id =>  
 val p = s.partitions(id)  
 (id, getPreferredLocs(stage.rdd, p))  
 }.toMap  
 }  
 } catch { // 如果发生任何异常，则调用Stage的makeNewStageAttempt方法开始一次新的Stage执行尝试  
 case NonFatal(e) =>  
 stage.makeNewStageAttempt(partitionsToCompute.size)  
 listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))  
 abortStage(stage, s"Task creation failed: $e\n${Utils.exceptionString(e)}", Some(e))  
 runningStages -= stage  
 return  
 }  
 // 开始Stage的执行尝试  
 stage.makeNewStageAttempt(partitionsToCompute.size, taskIdToLocations.values.toSeq)  
 listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))  
  
 var taskBinary: Broadcast[Array[Byte]] = null //  
 try {  
 val taskBinaryBytes: Array[Byte] = stage match {  
 case stage: ShuffleMapStage =>  
 JavaUtils.bufferToArray(  
 closureSerializer.serialize((stage.rdd, stage.shuffleDep): AnyRef))  
 case stage: ResultStage =>  
 JavaUtils.bufferToArray(closureSerializer.serialize((stage.rdd, stage.func): AnyRef))  
 }  
  
 taskBinary = sc.broadcast(taskBinaryBytes) // 广播任务的序列化对象  
 } catch {  
 case e: NotSerializableException =>  
 abortStage(stage, "Task not serializable: " + e.toString, Some(e))  
 runningStages -= stage  
  
 return  
 case NonFatal(e) =>  
 abortStage(stage, s"Task serialization failed: $e\n${Utils.exception-String(e)}", Some(e))  
 runningStages -= stage  
 return  
 }  
  
 val tasks: Seq[Task[\_]] = try {  
 stage match {  
 case stage: ShuffleMapStage => // 为ShuffleMapStage的每一个分区创建一个ShuffleMapTask  
 partitionsToCompute.map { id =>  
 val locs = taskIdToLocations(id)  
 val part = stage.rdd.partitions(id)  
 new ShuffleMapTask(stage.id, stage.latestInfo.attemptId,  
 taskBinary, part, locs, stage.latestInfo.taskMetrics, properties, Option(jobId),  
 Option(sc.applicationId), sc.applicationAttemptId)  
 }  
  
 case stage: ResultStage => // 为ResultStage的每一个分区创建一个ResultTask  
 partitionsToCompute.map { id =>  
 val p: Int = stage.partitions(id)  
 val part = stage.rdd.partitions(p)  
 val locs = taskIdToLocations(id)  
 new ResultTask(stage.id, stage.latestInfo.attemptId,  
 taskBinary, part, locs, id, properties, stage.latestInfo.taskMetrics,  
 Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)  
 }  
 }  
 } catch {  
 case NonFatal(e) =>  
 abortStage(stage, s"Task creation failed: $e\n${Utils.exceptionString(e)}", Some(e))  
 runningStages -= stage  
 return  
 }  
  
 if (tasks.size > 0) { // 调用TaskScheduler的submitTasks方法提交此批Task  
 logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " (" + stage.rdd + ")")  
 stage.pendingPartitions ++= tasks.map(\_.partitionId)  
 logDebug("New pending partitions: " + stage.pendingPartitions)  
 taskScheduler.submitTasks(new TaskSet(  
 tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))  
 stage.latestInfo.submissionTime = Some(clock.getTimeMillis())  
 } else { // 没有创建任何Task,将当前Stage标记为完成  
 markStageAsFinished(stage, None)  
  
 val debugString = stage match {  
 case stage: ShuffleMapStage =>  
 s"Stage ${stage} is actually done; " +  
 s"(available: ${stage.isAvailable}," +  
 s"available outputs: ${stage.numAvailableOutputs}," +  
 s"partitions: ${stage.numPartitions})"  
 case stage : ResultStage =>  
 s"Stage ${stage} is actually done; (partitions: ${stage.numPartitions})"  
 }  
 logDebug(debugString)  
  
 submitWaitingChildStages(stage) // 提交当前Stage的子Stage  
 }  
}

通过对代码清单7-42的分析，submitMissingTasks的执行过程总结如下。

1）清空当前Stage的pendingPartitions。由于当前Stage的任务刚开始提交，所以需要清空便于记录需要计算的分区任务。

2）调用Stage的findMissingPartitions方法（见代码清单7-14和代码清单7-16），找出当前Stage的所有分区中还没有完成计算的分区的索引。

3）获取ActiveJob的properties。properties包含了当前Job的调度、group、描述等属性信息。

4）将当前Stage加入runningStages集合中，即当前Stage已经处于运行状态。

5）调用OutputCommitCoordinator的stageStart方法（见代码清单5-93），启动对当前Stage的输出提交到HDFS的协调。

6）调用DAGScheduler的getPreferredLocs方法（见代码清单7-28），获取partitions-ToCompute中的每一个分区的偏好位置。如果发生任何异常，则调用Stage的makeNew-StageAttempt方法开始一次新的Stage执行尝试，然后向listenerBus投递SparkListener-StageSubmitted事件。

7）调用Stage的makeNewStageAttempt方法（见代码清单7-13）开始Stage的执行尝试，并向listenerBus投递SparkListenerStageSubmitted事件。

8）如果当前Stage是ShuffleMapStage，那么对Stage的rdd和ShuffleDependency进行序列化；如果当前Stage是ResultStage，那么对Stage的rdd和对RDD的分区进行计算的函数func进行序列化。

9）调用SparkContext的broadcast方法（见代码清单4-27）广播上一步生成的序列化对象。

10）如果当前Stage是ShuffleMapStage，则为ShuffleMapStage的每一个分区创建一个ShuffleMapTask。如果当前Stage是ResultStage，则为ResultStage的每一个分区创建一个ResultTask。

11）如果第10）步创建了至少1个Task，那么将此Task处理的分区索引添加到Stage的pendingPartitions中，然后为这批Task创建TaskSet（即任务集合），并调用TaskScheduler的submitTasks方法提交此TaskSet。

12）如果第10）步没有创建任何Task，这意味着当前Stage没有Task需要提交执行，因此调用DAGScheduler的markStageAsFinished方法（见代码清单7-43），将当前Stage标记为完成。然后调用submitWaitingChildStages方法（见代码清单7-44），提交当前Stage的子Stage。

代码清单7-43 将Stage标记为完成

private def markStageAsFinished(stage: Stage, errorMessage: Option[String] = None): Unit = {  
 val serviceTime = stage.latestInfo.submissionTime match { // 计算Stage的执行时间  
 case Some(t) => "%.03f".format((clock.getTimeMillis() - t) / 1000.0)  
 case \_ => "Unknown"  
 }  
 if (errorMessage.isEmpty) {  
 logInfo("%s (%s) finished in %s s".format(stage, stage.name, serviceTime))  
 stage.latestInfo.completionTime = Some(clock.getTimeMillis()) // 设置Stage的完成时间  
  
 stage.clearFailures()   
 } else {  
 stage.latestInfo.stageFailed(errorMessage.get) // 保存失败原因和Stage的完成时间  
 logInfo(s"$stage (${stage.name}) failed in $serviceTime s due to ${error Message.get}")  
 }  
  
 outputCommitCoordinator.stageEnd(stage.id) // 停止对当前Stage的输出提交到HDFS的协调  
 listenerBus.post(SparkListenerStageCompleted(stage.latestInfo))  
 runningStages -= stage // 将当前Stage从正在运行的Stage中移除  
}

根据代码清单7-43，markStageAsFinished的执行步骤如下。

1）计算Stage的执行时间。

2）如果执行Stage的过程中没有发生错误，那么设置Stage的latestInfo（即StageInfo）的完成时间为系统当前时间，并调用Stage的clearFailures方法清空fetchFailedAttemptIds。

3）如果执行Stage的过程中发生了错误，那么调用StageInfo的stageFailed方法（见代码清单7-18）保存失败原因和Stage的完成时间。

4）调用OutputCommitCoordinator的stageEnd方法（见代码清单5-94），停止对当前Stage的输出提交到HDFS的协调。

5）向LiveListenerBus投递SparkListenerStageCompleted事件。

6）将当前Stage从正在运行的Stage中移除。

代码清单7-44 提交Stage的所有处于等待中的子Stage

private def submitWaitingChildStages(parent: Stage) {  
 // 忽略日志  
 val childStages = waitingStages.filter(\_.parents.contains(parent)).toArray  
 waitingStages --= childStages  
 for (stage <- childStages.sortBy(\_.firstJobId)) {  
 submitStage(stage)  
 }  
}

### 7.4.10 DAGScheduler的调度流程

经过对DAGScheduler的详细介绍，现在我们可以用图7-6来表示DAGScheduler的主要调度流程。

这里对图7-6中的记号进行介绍。

记号①： 表示应用程序通过对Spark API的调用，进行一系列RDD转换构建出RDD之间的依赖关系后，调用DAGScheduler的runJob方法将RDD及其血缘关系中的所有RDD传递给DAGScheduler进行调度。

记号②： DAGScheduler的runJob方法实际通过调用DAGScheduler的submitJob方法向DAGSchedulerEventProcessLoop发送JobSubmitted事件。DAGSchedulerEventProcessLoop接收到JobSubmitted事件后，将JobSubmitted事件放入事件队列（eventQueue）。

记号③： DAGSchedulerEventProcessLoop内部的轮询线程eventThread不断从事件队列（eventQueue）中获取DAGSchedulerEvent事件，并调用DAGSchedulerEventProcessLoop的doOnReceive方法对事件进行处理。

记号④： DAGSchedulerEventProcessLoop的doOnReceive方法处理JobSubmitted事件时，将调用DAGScheduler的handleJobSubmitted方法。handleJobSubmitted方法将对RDD构建Stage及Stage之间的依赖关系。

记号⑤： DAGScheduler首先把最上游的Stage中的Task集合提交给TaskScheduler，然后逐步将下游的Stage中的Task集合提交给TaskScheduler。TaskScheduler将对Task集合进行调度。

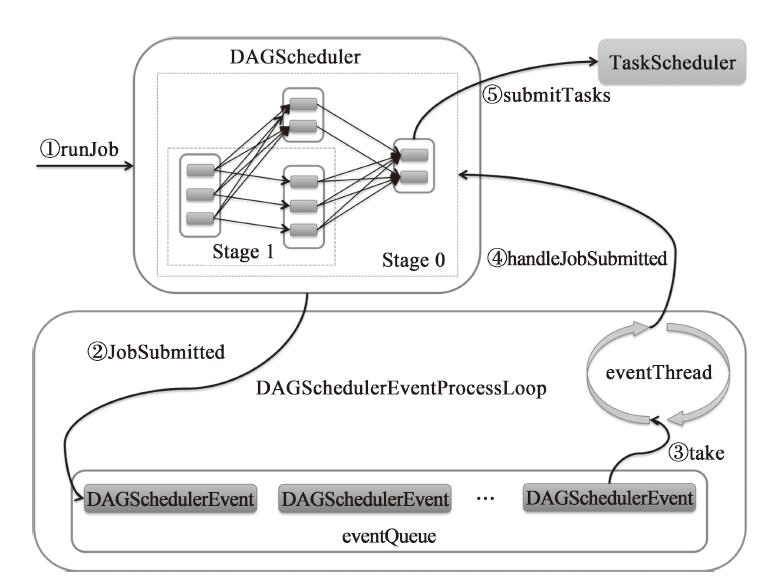


图7-6 DAGScheduler的调度流程

### 7.4.11 Task执行结果的处理

DAGScheduler的taskEnded方法（见代码清单7-45）用于对Task执行的结果进行处理。对于ShuffleMapTask而言，需要将它的状态信息MapStatus追加到ShuffleMapStage的outputLocs缓存中；如果ShuffleMapStage的所有分区的ShuffleMapTask都执行成功了，那么将需要把ShuffleMapStage的outputLocs缓存中的所有MapStatus注册到MapOutput-TrackerMaster的mapStatuses中，以便于下游Stage中的Task读取输入数据所在的位置信息；如果某个ShuffleMapTask执行失败了，则需要重新提交ShuffleMapStage；如果ShuffleMapStage的所有ShuffleMapTask都执行成功了，还需要唤醒下游Stage的执行。对于ResultTask而言，如果ResultStage中的所有ResultTask都执行成功了，则将ResultStage标记为成功，并通知JobWaiter对各个ResultTask的执行结果进行收集，然后根据应用程序的需要进行最终的处理（如打印到控制台、输出到HDFS）。

代码清单7-45 taskEnded方法的实现

def taskEnded(  
 task: Task[\_],  
 reason: TaskEndReason,  
 result: Any,  
 accumUpdates: Seq[AccumulatorV2[\_, \_]],  
 taskInfo: TaskInfo): Unit = {  
 eventProcessLoop.post(  
 CompletionEvent(task, reason, result, accumUpdates, taskInfo))  
}

根据代码清单7-45，taskEnded方法将向DAGSchedulerEventProcessLoop投递Completion-Event事件。根据代码清单7-22，DAGSchedulerEventProcessLoop接收到CompletionEvent事件后，将调用DAGScheduler的handleTaskCompletion方法。handleTaskCompletion方法中实现了针对不同执行状态的处理，这里只介绍对成功状态的处理。

1.ResultTask的结果处理

handleTaskCompletion方法对执行完成的ResultTask的处理如下。

val stage = stageIdToStage(task.stageId)  
event.reason match {  
 case Success =>  
 stage.pendingPartitions -= task.partitionId  
 task match {  
 case rt: ResultTask[\_, \_] =>  
 val resultStage = stage.asInstanceOf[ResultStage]  
 resultStage.activeJob match {  
 case Some(job) =>  
 if (!job.finished(rt.outputId)) {  
 updateAccumulators(event)  
 job.finished(rt.outputId) = true // 对应分区的任务设置为完成状态  
 job.numFinished += 1 // 将ActiveJob的已完成的任务数加一  
 if (job.numFinished == job.numPartitions) {  
 // ActiveJob 的所有分区的任务都完成了  
 markStageAsFinished(resultStage) //将当前Stage标记为已完成  
 cleanupStateForJobAndIndependentStages(job)  
 listenerBus.post(  
 SparkListenerJobEnd(job.jobId, clock.getTimeMillis(), Job Succeeded))  
 }  
  
 try {  
 // 由JobWaiter的resultHandler函数来处理Job中每个Task的执行结果  
 job.listener.taskSucceeded(rt.outputId, event.result)   
 } catch {  
 case e: Exception =>  
 job.listener.jobFailed(new SparkDriverExecutionException(e))//由JobWaiter处理失败  
 }  
 }  
 case None =>  
 logInfo("Ignoring result from " + rt + " because its job has finished")  
 }  
  
 case smt: ShuffleMapTask =>  
 // 忽略ShuffleMapTask的结果处理

根据上述代码，对ResultTask的执行结果进行处理的步骤如下。

1）将ResultStage的ActiveJob的finished里对应分区的任务设置为完成状态，并且将ActiveJob的已完成的任务数（numFinished）加1。

2）如果ActiveJob的所有任务都完成，则调用DAGScheduler的markStageAsFinished方法（见代码清单7-43）标记当前Stage完成，并向LiveListenerBus投递SparkListener-JobEnd事件。

3）调用JobWaiter的taskSucceeded方法（见代码清单7-20），进而调用JobWaiter的resultHandler函数来处理Job中每个Task的执行结果。

2.ShuffleMapTask的结果处理

handleTaskCompletion方法对执行完成的ShuffleMapTask的处理如下。

val stage = stageIdToStage(task.stageId)  
event.reason match {  
 case Success =>  
 stage.pendingPartitions -= task.partitionId  
 task match {  
 case rt: ResultTask[\_, \_] =>  
 // 忽略ResultTask的结果处理  
 case smt: ShuffleMapTask =>  
 val shuffleStage = stage.asInstanceOf[ShuffleMapStage]  
 updateAccumulators(event)  
 val status = event.result.asInstanceOf[MapStatus]  
 val execId = status.location.executorId  
 logDebug("ShuffleMapTask finished on " + execId)  
 if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {  
 logInfo(s"Ignoring possibly bogus $smt completion from executor $execId")  
 } else { //将分区任务的MapStatus追加到Stage的outputLocs中  
 shuffleStage.addOutputLoc(smt.partitionId, status)  
 }  
  
 if (runningStages.contains(shuffleStage) && shuffleStage.pendingPartitions.isEmpty) {  
 markStageAsFinished(shuffleStage) // 将ShuffleMapStage标记为完成  
 logInfo("looking for newly runnable stages")  
 logInfo("running: " + runningStages)  
 logInfo("waiting: " + waitingStages)  
 logInfo("failed: " + failedStages)  
 // 将当前Stage的shuffleId和outputLocs中的MapStatus注册到MapOutput-TrackerMaster的  
 // mapStatuses中  
 mapOutputTracker.registerMapOutputs(  
 shuffleStage.shuffleDep.shuffleId,  
 shuffleStage.outputLocInMapOutputTrackerFormat(),  
 changeEpoch = true)  
  
 clearCacheLocs()  
  
 if (!shuffleStage.isAvailable) {  
 logInfo("Resubmitting " + shuffleStage + " (" + shuffleStage.name +  
 ") because some of its tasks had failed: " +  
 shuffleStage.findMissingPartitions().mkString(", "))  
 submitStage(shuffleStage) // 有任务失败了，这时需要再次提交此ShuffleMapStage  
 } else {  
 if (shuffleStage.mapStageJobs.nonEmpty) {  
 val stats = mapOutputTracker.getStatistics(shuffleStage.shuffleDep)  
 for (job <- shuffleStage.mapStageJobs) {  
 markMapStageJobAsFinished(job, stats) // 将ActiveJob标记为执行成功  
 }  
 }  
 submitWaitingChildStages(shuffleStage) // 提交当前ShuffleMapStage的下游Stage  
 }  
 }  
 }

根据上述代码，对ShuffleMapTask的执行结果进行处理的步骤如下。

1）将Task的partitionId和MapStatus追加到Stage的outputLocs中。

2）如果ShuffleMapStage中没有待计算的分区，那么调用DAGScheduler的markStage-AsFinished方法（见代码清单7-43），将ShuffleMapStage标记为完成，然后调用MapOutput-TrackerMaster的registerMapOutputs方法（见代码清单5-72），将当前Stage的shuffleId和outputLocs中的MapStatus注册到MapOutputTrackerMaster的mapStatuses中。根据5.6节的内容，这里注册的MapStatus将最终被reduce任务所用。

3）如果ShuffleMapStage的isAvailable方法返回false（即\_numAvailableOutputs与numPartitions不相等），那么说明有任务失败了，这时需要再次提交此ShuffleMapStage。

4）如果ShuffleMapStage的isAvailable方法返回true（即\_numAvailableOutputs与num-Partitions相等），那么说明所有任务执行成功了，这时调用MapOutputTracker的get-Statistics方法获取Shuffle依赖的各个map任务输出Block大小的统计信息，并将Shuffle-MapStage的\_mapStageJobs属性中保存的各个ActiveJob标记为执行成功，最后调用submitWaitingChildStages方法（见代码清单7-44），提交当前ShuffleMapStage的子Stage。

通过本节对DAGScheduler的分析，可以看到DAGScheduler的主要工作包括创建Job，构建Stage的上下游关系，反向驱动Stage的提交及正向提交Stage等。反向驱动Stage和正向提交Stage的过程可以用图7-7来表示。

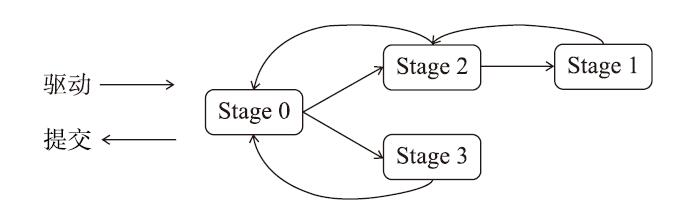


图7-7 Stage的正向驱动与反向提交

## 7.5 调度池Pool

TaskSchedulerImpl对Task的调度依赖于调度池Pool，所有需要被调度的TaskSet都被置于调度池中。调度池Pool通过调度算法对每个TaskSet进行调度，并将被调度的TaskSet交给TaskSchedulerImpl进行资源调度。

### 7.5.1 调度算法

调度池对TaskSet的调度取决于调度算法，所以本节将介绍调度算法。特质Schedu-lingAlgorithm定义了调度算法的规范，代码如下。

private[spark] trait SchedulingAlgorithm {  
 def comparator(s1: Schedulable, s2: Schedulable): Boolean  
}

根据上面的代码，我们知道SchedulingAlgorithm仅仅定义了一个comparator方法，用于对两个Schedulable进行比较。我们暂时不用去了解Schedulable，先来看看调度算法的具体实现。

SchedulingAlgorithm有两个实现类，分别为实现了先进先出（First In First Out，FIFO）算法的FIFOSchedulingAlgorithm和公平调度算法的FairSchedulingAlgorithm，下面将对这两种算法实现进行介绍。

1.FIFOSchedulingAlgorithm详解

FIFOSchedulingAlgorithm实现了FIFO调度算法，如代码清单7-46所示。

代码清单7-46 FIFOSchedulingAlgorithm的实现

private[spark] class FIFOSchedulingAlgorithm extends SchedulingAlgorithm {  
 override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {  
 val priority1 = s1.priority  
 val priority2 = s2.priority  
 var res = math.signum(priority1 - priority2) // 对s1和s2两个Schedulable的优先级进行比较  
 if (res == 0) {  
 val stageId1 = s1.stageId  
 val stageId2 = s2.stageId  
 res = math.signum(stageId1 - stageId2) // 对s1和s2所属的Stage的身份标识进行比较  
 }  
 res < 0  
 }  
}

根据代码清单7-46，FIFOSchedulingAlgorithm重写的comparator方法的执行步骤如下。

1）对s1和s2两个Schedulable的优先级（值越小，优先级越高）进行比较。

2）如果两个Schedulable的优先级相同，则对s1和s2所属的Stage的身份标识进行比较。

3）如果比较的结果小于0，则优先调度s1，否则优先调度s2。

2.FairSchedulingAlgorithm详解

FairSchedulingAlgorithm实现了公平调度算法，如代码清单7-47所示。

代码清单7-47 FairSchedulingAlgorithm的实现

private[spark] class FairSchedulingAlgorithm extends SchedulingAlgorithm {  
 override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {  
 val minShare1 = s1.minShare  
 val minShare2 = s2.minShare  
 val runningTasks1 = s1.runningTasks  
 val runningTasks2 = s2.runningTasks  
 val s1Needy = runningTasks1 < minShare1  
 val s2Needy = runningTasks2 < minShare2  
 val minShareRatio1 = runningTasks1.toDouble / math.max(minShare1, 1.0)  
 val minShareRatio2 = runningTasks2.toDouble / math.max(minShare2, 1.0)  
 val taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble  
 val taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble  
  
 var compare = 0  
 if (s1Needy && !s2Needy) {  
 return true  
 } else if (!s1Needy && s2Needy) {  
 return false  
 } else if (s1Needy && s2Needy) {  
 compare = minShareRatio1.compareTo(minShareRatio2)  
 } else {  
 compare = taskToWeightRatio1.compareTo(taskToWeightRatio2)  
 }  
 if (compare < 0) {  
 true  
 } else if (compare > 0) {  
 false  
 } else {  
 s1.name < s2.name  
 }  
 }

根据代码清单7-47，FairSchedulingAlgorithm重写的comparator方法的执行步骤如下。

1）如果s1中处于运行状态的Task的数量小于s1的minShare，并且s2中处于运行状态的Task的数量大于等于s2的minShare，那么优先调度s1。

2）如果s1中处于运行状态的Task的数量大于等于s1的minShare，并且s2中处于运行状态的Task的数量小于s2的minShare，那么优先调度s2。

3）如果s1中处于运行状态的Task的数量小于s1的minShare，并且s2中处于运行状态的Task的数量小于s2的minShare，那么再对minShareRatio1和minShareRatio2进行比较。如果minShareRatio1小于minShareRatio2，则优先调度s1；如果minShareRatio2小于minShareRatio1，则优先调度s2。如果minShareRatio1和minShareRatio2相等，还需要对s1和s2的名字进行比较。如果s1的名字小于s2的名字，则优先调度s1，否则优先调度s2。minShareRatio是正在运行的任务数量与minShare之间的比值。

4）如果s1中处于运行状态的Task的数量大于等于s1的minShare，并且s2中处于运行状态的Task的数量大于等于s2的minShare，那么再对taskToWeightRatio1和taskTo-WeightRatio2进行比较。如果taskToWeightRatio1小于taskToWeightRatio2，则优先调度s1；如果taskToWeightRatio2小于taskToWeightRatio1，则优先调度s2。如果taskToWeight-Ratio1和taskToWeightRatio2相等，还需要对s1和s2的名字进行比较。如果s1的名字小于s2的名字，则优先调度s1，否则优先调度s2。taskToWeightRatio是正在运行的任务数量与权重（weight）之间的比值。

### 7.5.2 Pool的实现

TaskScheduler对任务的调度是借助于调度池实现的，Pool是对Task集合进行调度的调度池。调度池内部有一个根调度队列，根调度队列中包含了多个子调度池。子调度池自身的调度队列中还可以包含其他的调度池或者TaskSetManager（TaskSetManager将在7.6节详细介绍，读者这里只需知道每个TaskSetManager包含着Task的集合），所以整个调度池是一个多层次的调度队列。Pool实现了Schedulable特质，其中包含如下属性。

·parent：当前Pool的父Pool。

·poolName：Pool的构造器属性之一，表示Pool的名称。

·schedulingMode：Pool的构造器属性之一，表示调度模式（SchedulingMode）。枚举类型SchedulingMode共有FAIR、FIFO、NONE三种枚举值。

·initMinShare：minShare的初始值。

·initWeight：weight的初始值。

·weight：用于公平调度算法的权重。

·minShare：用于公平调度算法的参考值。

·schedulableQueue：类型为ConcurrentLinkedQueue[Schedulable]，用于存储Schedulable。由于Schedulable只有Pool和TaskSetManager两个实现类，所以我们知道schedulable-Queue是一个可以嵌套的层次结构，如图7-8所示。

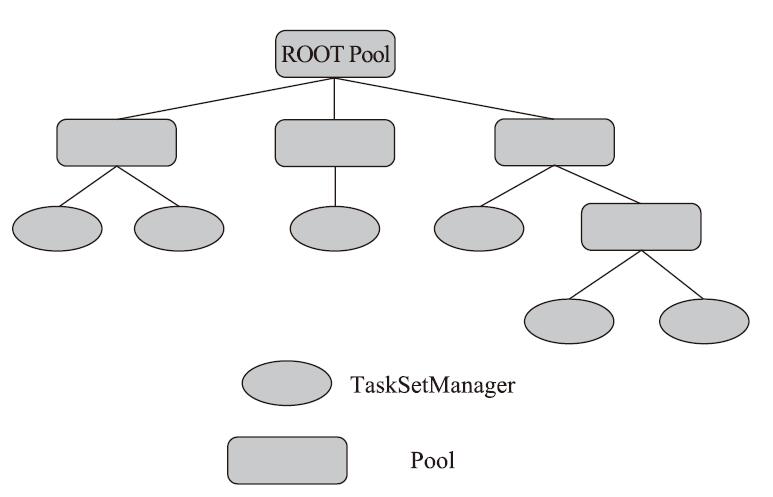


图7-8 调度队列的层次关系

小贴士： 熟悉Hadoop YARN的读者会发现，Spark的调度池中的调度队列与YARN中调度队列的设计非常相似，也采用了层级队列的管理方式。

·schedulableNameToSchedulable：调度名称与Schedulable的对应关系。

·runningTasks：当前正在运行的任务数量。

·priority：进行调度的优先级。

·stageId：调度池或TaskSetManager所属Stage的身份标识。

·name：与poolName相同。

·taskSetSchedulingAlgorithm：任务集合的调度算法，默认为FIFOSchedulingAlgorithm。

了解了Pool的属性，现在来看看Pool中提供的方法。

1.addSchedulable

addSchedulable方法（见代码清单7-48）用于将Schedulable添加到schedulableQueue和schedulableNameToSchedulable中，并将Schedulable的父亲设置为当前Pool。

代码清单7-48 添加Schedulable

override def addSchedulable(schedulable: Schedulable) {  
 require(schedulable != null)  
 schedulableQueue.add(schedulable)  
 schedulableNameToSchedulable.put(schedulable.name, schedulable)  
 schedulable.parent = this  
}

2.removeSchedulable

removeSchedulable方法（见代码清单7-49）用于将指定的Schedulable从schedulableQueue和schedulableNameToSchedulable中移除。

代码清单7-49 移除Schedulable

override def removeSchedulable(schedulable: Schedulable) {  
 schedulableQueue.remove(schedulable)  
 schedulableNameToSchedulable.remove(schedulable.name)  
}

3.getSchedulableByName

getSchedulableByName方法（见代码清单7-50）用于根据指定名称查找Schedulable。

代码清单7-50 根据指定名称查找Schedulable

override def getSchedulableByName(schedulableName: String): Schedulable = {  
 if (schedulableNameToSchedulable.containsKey(schedulableName)) {  
 return schedulableNameToSchedulable.get(schedulableName) // 从当前Pool找到的指定名称的Schedulable  
 }  
 for (schedulable <- schedulableQueue.asScala) {  
 val sched = schedulable.getSchedulableByName(schedulableName) //从子Schedulable中查找  
 if (sched != null) {  
 return sched  
 }  
 }  
 null  
}

根据代码清单7-50，getSchedulableByName方法的执行步骤如下。

1）从当前Pool的schedulableNameToSchedulable中查找指定名称的Schedulable。

2）如果上一步未找到，则从当前Pool的schedulableQueue中的各个子Schedulable中查找，直到找到指定名称的Schedulable或者返回null。

4.executorLost

executorLost方法（见代码清单7-51）用于当某个Executor丢失后，调用当前Pool的schedulableQueue中的各个Schedulable（可能为子调度池，也可能是TaskSetManager）的executorLost方法。TaskSetManager的executorLost方法（见代码清单7-74）进而将在此Executor上正在运行的Task作为失败任务处理，并重新提交这些任务。

代码清单7-51 executorLost的实现

override def executorLost(executorId: String, host: String, reason: Executor LossReason) {  
 schedulableQueue.asScala.foreach(\_.executorLost(executorId, host, reason))  
}

5.checkSpeculatableTasks

checkSpeculatableTasks方法（见代码清单7-52）用于检查当前Pool中是否有需要推断执行的任务。checkSpeculatableTasks实际通过迭代调用schedulableQueue中的各个子Schedulable的checkSpeculatableTasks方法来实现。Pool的checkSpeculatableTasks方法和TaskSetManager的checkSpeculatableTasks方法（见代码清单7-62），一起实现了按照深度遍历算法从调度池中查找可推断执行的任务。

代码清单7-52 检查当前Pool中是否有需要推断执行的任务

override def checkSpeculatableTasks(minTimeToSpeculation: Int): Boolean = {  
 var shouldRevive = false  
 for (schedulable <- schedulableQueue.asScala) {  
 shouldRevive |= schedulable.checkSpeculatableTasks(minTimeToSpeculation)  
 }  
 shouldRevive  
}

6.getSortedTaskSetQueue

getSortedTaskSetQueue方法（见代码清单7-53）用于对当前Pool中的所有TaskSetManager按照调度算法进行排序，并返回排序后的TaskSetManager。getSortedTaskSet-Queue实际是通过迭代调用schedulableQueue中的各个子Schedulable的getSortedTaskSetQueue方法来实现。

代码清单7-53 getSortedTaskSetQueue的实现

override def getSortedTaskSetQueue: ArrayBuffer[TaskSetManager] = {  
 var sortedTaskSetQueue = new ArrayBuffer[TaskSetManager]  
 val sortedSchedulableQueue =  
 schedulableQueue.asScala.toSeq.sortWith(taskSetSchedulingAlgorithm.comparator)  
 for (schedulable <- sortedSchedulableQueue) {  
 sortedTaskSetQueue ++= schedulable.getSortedTaskSetQueue  
 }  
 sortedTaskSetQueue  
}

7.increaseRunningTasks

increaseRunningTasks方法（见代码清单7-54）用于增加当前Pool及其父Pool中记录的当前正在运行的任务数量。

代码清单7-54 增加正在运行的Task数量

def increaseRunningTasks(taskNum: Int) {  
 runningTasks += taskNum  
 if (parent != null) {  
 parent.increaseRunningTasks(taskNum)  
 }  
}

8.decreaseRunningTasks

decreaseRunningTasks方法（见代码清单7-55）用于减少当前Pool及其父Pool中记录的当前正在运行的任务数量。

代码清单7-55 减少正在运行的Task数量

def decreaseRunningTasks(taskNum: Int) {  
 runningTasks -= taskNum  
 if (parent != null) {  
 parent.decreaseRunningTasks(taskNum)  
 }  
}

### 7.5.3 调度池构建器

在创建了调度池之后，池内还没有“水”，更没有“鱼”。需要园林建造师向池中注入水，有了水之后就可以养育多种多样的鱼儿了。SchedulableBuilder正是构建调度池的“建造师”。

特质SchedulableBuilder定义了调度池构建器的行为规范，请看如下代码。

private[spark] trait SchedulableBuilder {  
 def rootPool: Pool  
 def buildPools(): Unit  
 def addTaskSetManager(manager: Schedulable, properties: Properties): Unit  
}

上述代码中一共定义了三个方法。

·rootPool：返回根调度池。

·buildPools：对调度池进行构建。

·addTaskSetManager：向调度池内添加TaskSetManager。

针对FIFO和Fair两种调度算法，SchedulableBuilder共有两种实现，分别是FIFOSched-ulableBuilder和FairSchedulableBuilder。

1.FIFOSchedulableBuilder详解

FIFOSchedulableBuilder的实现如代码清单7-56所示。

代码清单7-56 FIFOSchedulableBuilder的实现

private[spark] class FIFOSchedulableBuilder(val rootPool: Pool)  
 extends SchedulableBuilder with Logging {  
 override def buildPools() {  
 // nothing  
 }  
  
 override def addTaskSetManager(manager: Schedulable, properties: Properties) {  
 rootPool.addSchedulable(manager)  
 }  
}

根据代码清单7-56，FIFOSchedulableBuilder实现的buildPools方法是个空方法，而实现的addTaskSetManager方法将向根调度池中添加TaskSetManager。

有了对FIFOSchedulableBuilder的实现分析，我们可以用图7-9来表示由FIFOSchedulable-Builder构建出的调度池的内存结构。

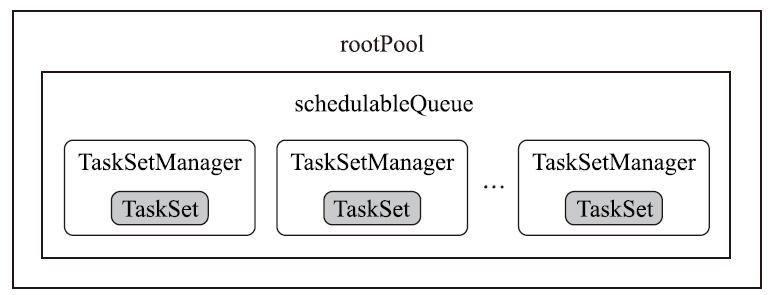


图7-9 FIFOSchedulableBuilder构建的调度池的内存结构

2.FairSchedulableBuilder详解

FairSchedulableBuilder的实现较为复杂，为便于分析，我们从了解它的属性开始。Fair-SchedulableBuilder的属性如下。

1）rootPool：根调度池。rootPool是FairSchedulableBuilder的构造器属性。

2）conf：即SparkConf。

3）schedulerAllocFile：用户指定的文件系统中的调度分配文件。此文件可以通过spark.scheduler.allocation.file属性配置，FairSchedulableBuilder将从文件系统中读取此文件提供的公平调度配置。

4）DEFAULT\_SCHEDULER\_FILE：默认的调度文件名。常量DEFAULT\_SCHEDULER\_FILE的值固定为"fairscheduler.xml"，FairSchedulableBuilder将从ClassPath中读取此文件提供的公平调度配置。下面展示了fairscheduler.xml的配置内容。

<allocations>  
 <pool name="production">  
 <schedulingMode>FAIR</schedulingMode>  
 <weight>1</weight>  
 <minShare>2</minShare>  
 </pool>  
 <pool name="test">  
 <schedulingMode>FIFO</schedulingMode>  
 <weight>2</weight>  
 <minShare>3</minShare>  
 </pool>  
</allocations>

从fairscheduler.xml的配置内容可以看到，FAIR调度算法下的每个调度池可以选择不同的调度算法。

①FAIR\_SCHEDULER\_PROPERTIES：即spark.scheduler.pool，此属性的值作为放置TaskSetManager的公平调度池的名称。

②DEFAULT\_POOL\_NAME：默认的调度池名。常量DEFAULT\_POOL\_NAME的值固定为"default"。

③MINIMUM\_SHARES\_PROPERTY：常量MINIMUM\_SHARES\_PROPERTY的值固定为"minShare"，即XML文件的<Pool>节点的子节点<mindshare>。节点<mindshare>的值将作为Pool的minShare属性。

④SCHEDULING\_MODE\_PROPERTY：常量SCHEDULING\_MODE\_PROPERTY的值固定为"schedulingMode"，即XML文件的<Pool>节点的子节点<schedulingMode>。节点<schedulingMode>的值将作为Pool的调度模式（schedulingMode）属性。

⑤WEIGHT\_PROPERTY：权重属性。常量WEIGHT\_PROPERTY的值固定为"weight"，即XML文件的<Pool>节点的子节点<weight>。节点<weight>的值将作为Pool的权重（weight）属性。

⑥POOL\_NAME\_PROPERTY：常量POOL\_NAME\_PROPERTY的值固定为"@name"，即XML文件的<Pool>节点的name属性。name属性的值将作为Pool的调度池名（poolName）属性。

⑦POOLS\_PROPERTY：调度池属性。常量POOLS\_PROPERTY的值固定为"pool"。

⑧DEFAULT\_SCHEDULING\_MODE：默认的调度模式FIFO。

⑨DEFAULT\_MINIMUM\_SHARE：公平调度算法中Schedulable的minShare属性的默认值。常量DEFAULT\_MINIMUM\_SHARE的值固定为0。

⑩DEFAULT\_WEIGHT：默认的权重。常量DEFAULT\_WEIGHT的值固定为1。

有了对FairSchedulableBuilder属性的了解，现在来看看FairSchedulableBuilder的实现。FairSchedulableBuilder实现了特质SchedulableBuilder的buildPools和addTaskSet-Manager，下面将逐一介绍。

5）buildPools：FairSchedulableBuilder实现的buildPools方法如代码清单7-57所示。

代码清单7-57 构建公平调度池

override def buildPools() {  
 var is: Option[InputStream] = None  
 try {  
 is = Option {  
 schedulerAllocFile.map { f =>  
 new FileInputStream(f) // 从文件系统中读取公平调度配置的文件输入流  
 }.getOrElse {  
 Utils.getSparkClassLoader.getResourceAsStream(DEFAULT\_SCHEDULER\_FILE)  
 }  
 }  
  
 is.foreach { i => buildFairSchedulerPool(i) } // 解析文件输入流并构建调度池  
 } finally {  
 is.foreach(\_.close())  
 }  
  
 buildDefaultPool() // 构建默认的调度池  
}

根据代码清单7-57，FairSchedulableBuilder的buildPools方法的执行步骤如下。

①优先从文件系统中读取schedulerAllocFile的文件输入流。如果用户未指定此scheduler-AllocFile，则默认从类路径下加载fairscheduler.xml文件的输入流。

②调用buildFairSchedulerPool方法（见代码清单7-58）对文件输入流进行解析并构建调度池。

③调用buildDefaultPool方法（见代码清单7-59）构建默认的调度池。

代码清单7-58 buildFairSchedulerPool的实现

private def buildFairSchedulerPool(is: InputStream) {  
 val xml = XML.load(is) // 将文件输入流转换为XML  
 for (poolNode <- (xml \\ POOLS\_PROPERTY)) { // 读取XML的每一个<Pool>节点  
  
 val poolName = (poolNode \ POOL\_NAME\_PROPERTY).text // 读取<Pool>的name属性作为调度池的名称  
 var schedulingMode = DEFAULT\_SCHEDULING\_MODE  
 var minShare = DEFAULT\_MINIMUM\_SHARE  
 var weight = DEFAULT\_WEIGHT  
  
 val xmlSchedulingMode = (poolNode \ SCHEDULING\_MODE\_PROPERTY).text  
 if (xmlSchedulingMode != "") {  
 try { // 读取<Pool>的子节点<schedulingMode>的值作为调度池的调度模式（schedulingMode）　　　　　　　　　 属性  
 schedulingMode = SchedulingMode.withName(xmlSchedulingMode)  
 } catch {  
 case e: NoSuchElementException =>  
 logWarning(s"Unsupported schedulingMode: $xmlSchedulingMode, " +  
 s"using the default schedulingMode: $schedulingMode")  
 }  
 }  
 // 读取<Pool>的子节点<minShare>的值作为调度池的minShare属性  
 val xmlMinShare = (poolNode \ MINIMUM\_SHARES\_PROPERTY).text  
 if (xmlMinShare != "") {  
 minShare = xmlMinShare.toInt  
 }  
 // 读取<Pool>的子节点<weight>的值作为调度池的权重（weight）属性  
 val xmlWeight = (poolNode \ WEIGHT\_PROPERTY).text  
 if (xmlWeight != "") {  
 weight = xmlWeight.toInt  
 }  
 val pool = new Pool(poolName, schedulingMode, minShare, weight) // 创建子调度池  
 rootPool.addSchedulable(pool) // 将创建的子调度池添加到根调度池的调度队列  
 logInfo("Created pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(  
 poolName, schedulingMode, minShare, weight))  
 }  
}

根据代码清单7-58，buildFairSchedulerPool方法的执行步骤如下。

①将文件输入流转换为XML。

②读取XML的每一个<Pool>节点，并对每个<Pool>节点的子元素执行以下读取操作。

·读取<Pool>的name属性作为调度池的名称。

·读取<Pool>的子节点schedulingMode的值作为调度池的调度模式（scheduling-Mode）属性。如果未对<Pool>节点指定子节点<schedulingMode>，则默认以常量DEFAULT\_SCHEDULING\_MODE的值作为调度池的调度模式。

·读取<Pool>的子节点<minShare>的值作为调度池的minShare属性。如果未对Pool节点指定子节点<minShare>，则默认以常量DEFAULT\_MINIMUM\_SHARE的值作为调度池的minShare属性。

·读取<Pool>的子节点<weight>的值作为调度池的权重（weight）属性。如果未对Pool节点指定子节点<weight>，则默认以常量DEFAULT\_WEIGHT的值作为调度池的weight属性。

·使用调度池名称、schedulingMode、minShare及weight创建子调度池。

·将创建的子调度池添加到根调度池的调度队列中。

构建默认调度池的代码如代码清单7-59所示。

代码清单7-59 构建默认调度池

private def buildDefaultPool() {  
 if (rootPool.getSchedulableByName(DEFAULT\_POOL\_NAME) == null) {  
 val pool = new Pool(DEFAULT\_POOL\_NAME, DEFAULT\_SCHEDULING\_MODE,  
 DEFAULT\_MINIMUM\_SHARE, DEFAULT\_WEIGHT) // 创建默认调度池  
 rootPool.addSchedulable(pool) 　　　　　　 // 向根调度池的调度队列中添加默认的子调度池  
 logInfo("Created default pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(  
 DEFAULT\_POOL\_NAME, DEFAULT\_SCHEDULING\_MODE, DEFAULT\_MINIMUM\_SHARE, DEFAULT\_WEIGHT))  
 }  
}

根据代码清单7-59，buildDefaultPool方法用于当根调度池及其子调度池中不存在名为default的调度池时，执行如下操作。

①创建名为default，schedulingMode为FIFO，minShare为0，weight为1的默认子调度池。

②向根调度池的调度队列中添加默认的子调度池。

6）addTaskSetManager：FairSchedulableBuilder实现的addTaskSetManager方法如代码清单7-60所示。

代码清单7-60 添加TaskSetManager

override def addTaskSetManager(manager: Schedulable, properties: Properties) {  
 var poolName = DEFAULT\_POOL\_NAME  
 var parentPool = rootPool.getSchedulableByName(poolName) //以默认调度池作为TaskSetManager的父调度池  
 if (properties != null) { // 以spark.scheduler.pool属性指定的调度池作为TaskSetManager的父调度池  
 poolName = properties.getProperty(FAIR\_SCHEDULER\_PROPERTIES, DEFAULT\_POOL\_NAME)  
 parentPool = rootPool.getSchedulableByName(poolName)  
 if (parentPool == null) {  
 parentPool = new Pool(poolName, DEFAULT\_SCHEDULING\_MODE,  
 DEFAULT\_MINIMUM\_SHARE, DEFAULT\_WEIGHT)  
 rootPool.addSchedulable(parentPool)   
 logInfo("Created pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(  
 poolName, DEFAULT\_SCHEDULING\_MODE, DEFAULT\_MINIMUM\_SHARE, DEFAULT\_WEIGHT))  
 }  
 }  
 parentPool.addSchedulable(manager) // 将TaskSetManager放入父调度池  
 logInfo("Added task set " + manager.name + " tasks to pool " + poolName)  
}

根据代码清单7-60，FairSchedulableBuilder的addTaskSetManager方法的执行步骤如下。

①以名为default的调度池作为TaskSetManager的父调度池。

②如果在Properties中指定了spark.scheduler.pool属性，则从根调度池中查找此属性值对应的调度池作为TaskSetManager的父调度池。如果根调度池中不存在此属性值对应的调度池，则创建以此属性值为名称的调度池作为TaskSetManager的父调度池，并将此调度池作为根调度池的子调度池。

③将TaskSetManager放入前两步确定的父调度池。

有了对FairSchedulableBuilder的实现分析，我们可以用图7-10来表示由FairSchedulable-Builder构建出的调度池的内存结构。

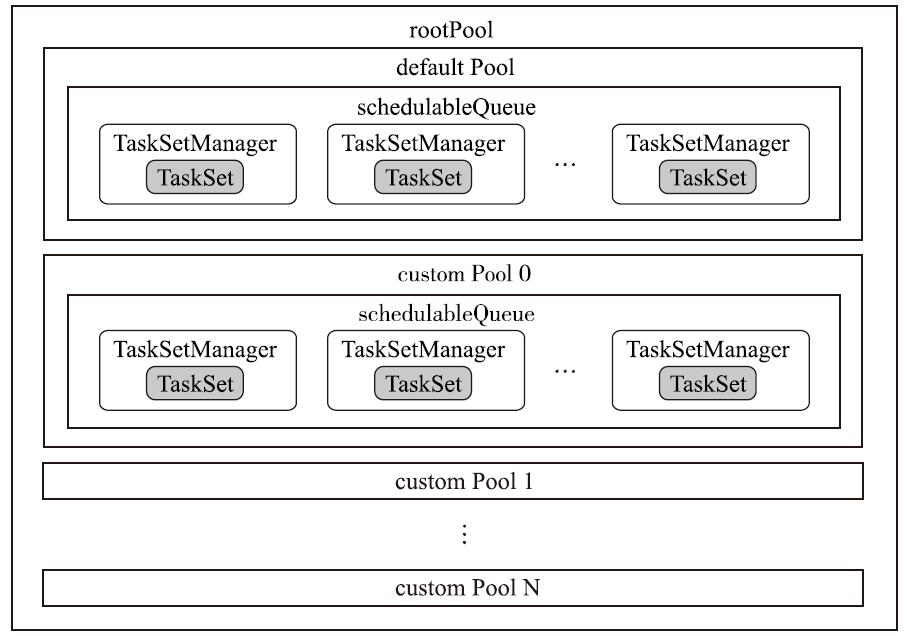


图7-10 FairSchedulableBuilder构建的调度池的内存结构

图7-10主要展示了根调度池中的多个子调度池，子调度池中包含默认的子调度池。每个调度池中都有若干TaskSetManager。

## 7.6 任务集合管理器TaskSetManager

TaskSetManager也实现了Schedulable特质，并参与到调度池的调度中。TaskSetManager对TaskSet进行管理，包括任务推断、Task本地性，并对Task进行资源分配。TaskScheduler-Impl依赖于TaskSetManager，本节将对TaskSetManager的实现进行分析。

### 7.6.1 Task集合

根据之前的介绍，DAGScheduler将Task提交给TaskScheduler时，需要将多个Task打包为TaskSet。TaskSet是整个调度池中对Task进行调度管理的基本单位，由调度池中的TaskSetManager来管理，它的定义如代码清单7-61所示。

代码清单7-61 TaskSet的定义

private[spark] class TaskSet(  
 val tasks: Array[Task[\_]],  
 val stageId: Int,  
 val stageAttemptId: Int,  
 val priority: Int,  
 val properties: Properties) {  
 val id: String = stageId + "." + stageAttemptId  
 override def toString: String = "TaskSet " + id  
}

这里对TaskSet中的属性进行介绍。

·tasks：TaskSet所包含的Task的数组。

·stageId：Task所属Stage的身份标识。

·stageAttemptId：Stage尝试的身份标识。

·priority：优先级。通常以JobId作为优先级。

·properties：包含了与Job有关的调度、Job group、描述等属性的Properties。

·id：TaskSet的身份标识。

### 7.6.2 TaskSetManager的成员属性

要了解TaskSetManager如何管理TaskSet，对它的属性成员进行了解，有助于读者的理解，TaskSetManager包含的属性如下。

·sched：即TaskSetManager所属的TaskSchedulerImpl。

·taskSet：当前TaskSetManager管理的TaskSet。

·maxTaskFailures：最大任务失败次数。

·clock：系统时钟。

·conf：即SparkConf。

·SPECULATION\_QUANTILE：开始推断执行的任务分数。可以通过spark.speculation.quantile属性进行配置，默认为0.75。

·SPECULATION\_MULTIPLIER：推断的乘数。可通过spark.speculation.multiplier属性进行配置，默认为1.5。SPECULATION\_MULTIPLIER将用于可推断执行Task的检查。

·maxResultSize：结果总大小的字节限制。可通过spark.driver.maxResultSize属性进行配置，默认为1GB。maxResultSize是通过调用Utils工具类的getMaxResultSize方法获得的，具体可参阅附录A。

·env：即SparkEnv。

·ser：即SparkEnv的closureSerializer的实例。

·tasks：TaskSet包含的Task数组，即TaskSet的tasks属性。

·numTasks：TaskSet包含的Task的数量，即tasks数组的长度。

·copiesRunning：对每个Task的复制运行数进行记录的数组。copiesRunning按照索引与tasks数组的同一索引位置的Task相对应，记录对应Task的复制运行数量。

·successful：对每个Task是否执行成功进行记录的数组。successful按照索引与tasks数组的同一索引位置的Task相对应，记录对应的Task是否执行成功。

·numFailures：对每个Task的执行失败次数进行记录的数组。numFailures按照索引与tasks数组的同一索引位置的Task相对应，记录对应Task的执行失败次数。

·taskAttempts：对每个Task的所有执行尝试信息进行记录的数组。taskAttempts按照索引与tasks数组的同一索引位置的Task相对应，记录对应Task的所有Task尝试信息（即TaskInfo，读者暂时只需要知道这个对象即可）。

·tasksSuccessful：执行成功的Task数量。

·weight：用于公平调度算法的权重。

·minShare：用于公平调度算法的参考值。

·priority：进行调度的优先级。

·stageId：调度池所属的Stage的身份标识。

·name：TaskSetManager的名称，可能会参与到公平调度算法中。

·parent：TaskSetManager的父Pool。

·totalResultSize：所有Task执行结果的总大小。

·calculatedTasks：计算过的Task数量。

·taskSetBlacklistHelperOpt：TaskSetManager所管理的TaskSet的Executor或节点的黑名单。

·runningTasksSet：正在运行的Task的集合。

·runningTasks：正在运行的Task的数量。runningTasks的值实际是通过调用running-TasksSet的size方法获取的。

·isZombie：当TaskSetManager所管理的TaskSet中的所有Task都执行成功了，不再有更多的Task尝试被启动时，就处于“僵尸”状态。例如，每个Task至少有一次尝试成功，或者TaskSet被舍弃了，TaskSetManager将会进入“僵尸”状态。直到所有的Task都运行成功为止，TaskSetManager将一直保持在“僵尸”状态。TaskSet-Manager的“僵尸”状态并不是无用的，在这种状态下TaskSetManager将继续跟踪、记录正在运行的Task。

·pendingTasksForExecutor：每个Executor上待处理的Task的集合，即Executor的身份标识与待处理Task的身份标识的集合之间的映射关系。

·pendingTasksForHost：每个Host上待处理的Task的集合，即Host与待处理Task的身份标识的集合之间的映射关系。

·pendingTasksForRack：每个机架上待处理的Task的集合，即机架与待处理Task的身份标识的集合之间的映射关系。

·pendingTasksWithNoPrefs：没有任何本地性偏好的待处理Task的身份标识的集合。

·allPendingTasks：所有待处理的Task的身份标识的集合。

·speculatableTasks：能够进行推断执行的Task的身份标识的集合。这类Task占所有Task的比例非常小。所谓推断执行，是指当Task的一次尝试运行非常缓慢，根据推断，如果此时可以另起一次尝试运行，后来的尝试运行也比原先的尝试运行要快。

·taskInfos：Task的身份标识与Task尝试的信息（如启动时间、完成时间等）之间的映射关系。

·EXCEPTION\_PRINT\_INTERVAL：异常打印到日志的时间间隔。可通过spark.logging.exceptionPrintInterval属性进行配置，默认为10000。

·recentExceptions：类型为HashMap[String,(Int,Long)]，用于缓存异常信息、异常次数及最后发生此异常的时间之间的映射关系。此属性与EXCEPTION\_PRINT\_INTERVAL配合使用。

·epoch：即MapOutputTracker的epoch，用于故障转移。

·myLocalityLevels：Task的本地性级别（7.6.4节将详细介绍）的数组。MyLocality-Levels是通过调用computeValidLocalityLevels方法（见代码清单7-64）获取的。

·localityWaits：与myLocalityLevels中的每个本地性级别相对应，表示对应本地性级别的等待时间。localityWaits实际是对myLocalityLevels中的每个本地性级别应用getLocalityWait方法（见代码清单7-65）获取的。

·currentLocalityIndex：当前的本地性级别在myLocalityLevels中的索引。

·lastLaunchTime：在当前的本地性级别上运行Task的时间。

·emittedTaskSizeWarning：当发现序列化后的Task的大小超过了100KB，此属性将被设置为true，并且打印警告级别的日志。

了解了TaskSetManager的属性，现在来看看TaskSetManager是如何实现推断执行、本地性计算、资源分配的。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 7.6.3 调度池与推断执行

在Hadoop 2.x.x版本中，当一个应用向YARN集群提交作业后，此作业的多个任务由于负载不均衡、资源分布不均等原因都会导致各个任务运行完成的时间不一致，甚至会出现一个Task尝试明显慢于同一作业的其他Task尝试的情况。如果对这种情况不加优化，最慢的Task尝试最终会拖慢整个作业的整体执行进度。好在mapreduce框架提供了任务推断执行机制，当有必要时就启动一个备份任务。最终会采用备份任务和原任务中率先执行完的结果作为最终结果。

与Hadoop类似，Spark应用向Spark集群提交作业后，也会因为相似的原因导致出现慢任务拖慢整个作业执行进度的问题。为了解决这些问题，Pool和TaskSetManager提供了Spark任务推断执行的实现。Pool和TaskSetManager中对推断执行的操作分为两类：一类是可推断任务的检测与缓存；另一类是从缓存中找到可推断任务进行推断执行。Pool的checkSpeculatableTasks方法（见代码清单7-52）和TaskSetManager的checkSpeculatable-Tasks方法（见代码清单7-62）实现了按照深度遍历算法对可推断任务的检测与缓存。TaskSet-Manager的dequeueSpeculativeTask方法（见代码清单7-71）则实现了从缓存中找到可推断任务进行推断执行。

1.checkSpeculatableTasks

checkSpeculatableTasks方法（见代码清单7-62）用于检查当前TaskSetManager中是否有需要推断的任务。

代码清单7-62 checkSpeculatableTasks的实现

override def checkSpeculatableTasks(minTimeToSpeculation: Int): Boolean = {  
 if (isZombie || numTasks == 1) {  
 return false // 没有可以推断的Task  
 }  
 var foundTasks = false  
 val minFinishedForSpeculation = (SPECULATION\_QUANTILE \* numTasks).floor.toInt  
 logDebug("Checking for speculative tasks: minFinished = " + minFinished ForSpeculation)  
 if (tasksSuccessful >= minFinishedForSpeculation && tasksSuccessful > 0) {  
 val time = clock.getTimeMillis()  
 val durations = taskInfos.values.filter(\_.successful).map(\_.duration).toArray  
 Arrays.sort(durations)  
 val medianDuration = durations(min((0.5 \* tasksSuccessful).round.toInt, durations.length - 1))  
 val threshold = max(SPECULATION\_MULTIPLIER \* medianDuration, minTime-ToSpeculation)  
 logDebug("Task length threshold for speculation: " + threshold)  
 for ((tid, info) <- taskInfos) { // 遍历taskInfos，寻找符合推断条件的Task  
 val index = info.index  
 if (!successful(index) && copiesRunning(index) == 1 && info.timeRunning(time) > threshold &&  
 !speculatableTasks.contains(index)) {  
 logInfo(  
 "Marking task %d in stage %s (on %s) as speculatable because it ran more than %.0f ms"  
 .format(index, taskSet.id, info.host, threshold))  
 speculatableTasks += index  
 foundTasks = true  
 }  
 }  
 }  
 foundTasks  
}

根据代码清单7-62，checkSpeculatableTasks方法的执行步骤如下。

1）如果TaskSetManager处于“僵尸”状态且TaskSet只包含一个Task，那么返回false（即没有可以推断的Task）。

2）计算进行推断的最小完成任务数量（minFinishedForSpeculation），即SPECULATION\_QUANTILE（开始推断的任务分数）与numTasks（TaskSet包含的Task的数量）的乘积向下取整。

3）如果执行成功的Task数量（tasksSuccessful）大于等于minFinishedForSpeculation并且tasksSuccessful大于0，就进入下一步，否则返回false。

4）找出taskInfos中执行成功的任务尝试信息（TaskInfo）中执行时间处于中间的时间medianDuration。

5）计算进行推断的最小时间（threshold），即SPECULATION\_MULTIPLIER与median-Duration的乘积和minTimeToSpeculation中的最大值。

6）遍历taskInfos，将符合推断条件的Task在TaskSet中的索引放入speculatableTasks中。推断条件包括还未执行成功、复制运行数为一、运行时间大于threshold。如果有Task的索引被放入了speculatableTasks，那么返回true。

2.dequeueSpeculativeTask

dequeueSpeculativeTask方法（见代码清单7-63）用于根据指定的Host、Executor和本地性级别，从可推断的Task中找出可推断的Task在TaskSet中的索引和相应的本地性级别。

代码清单7-63 找出可推断执行的Task

protected def dequeueSpeculativeTask(execId: String, host: String, locality: TaskLocality.Value)  
 : Option[(Int, TaskLocality.Value)] =  
{  
 speculatableTasks.retain(index => !successful(index)) // 移除已经完成的Task  
 def canRunOnHost(index: Int): Boolean = {  
 !hasAttemptOnHost(index, host) &&  
 !isTaskBlacklistedOnExecOrNode(index, execId, host)  
 }  
  
 if (!speculatableTasks.isEmpty) {  
 for (index <- speculatableTasks if canRunOnHost(index)) {  
 val prefs = tasks(index).preferredLocations  
 val executors = prefs.flatMap(\_ match {  
 case e: ExecutorCacheTaskLocation => Some(e.executorId)  
 case \_ => None  
 });  
 if (executors.contains(execId)) { // 找到了在指定的Executor上推断执行的Task  
 speculatableTasks -= index  
 return Some((index, TaskLocality.PROCESS\_LOCAL))  
 }  
 }  
  
 if (TaskLocality.isAllowed(locality, TaskLocality.NODE\_LOCAL)) {  
 for (index <- speculatableTasks if canRunOnHost(index)) {  
 val locations = tasks(index).preferredLocations.map(\_.host)  
 if (locations.contains(host)) { // 找到了在本地节点上推断执行的Task  
 speculatableTasks -= index  
 return Some((index, TaskLocality.NODE\_LOCAL))  
 }  
 }  
 }  
  
 if (TaskLocality.isAllowed(locality, TaskLocality.NO\_PREF)) {  
 for (index <- speculatableTasks if canRunOnHost(index)) {  
 val locations = tasks(index).preferredLocations  
 if (locations.size == 0) { // 对于没有本地性偏好的Task，让它在指定的Executor上推断执行  
 speculatableTasks -= index  
 return Some((index, TaskLocality.PROCESS\_LOCAL))  
 }  
 }  
 }  
  
 if (TaskLocality.isAllowed(locality, TaskLocality.RACK\_LOCAL)) {  
 for (rack <- sched.getRackForHost(host)) {  
 for (index <- speculatableTasks if canRunOnHost(index)) {  
 val racks = tasks(index).preferredLocations.map(\_.host).flatMap(sched.getRackForHost)  
 if (racks.contains(rack)) { // 找到了本地机架上推断执行的Task  
 speculatableTasks -= index  
 return Some((index, TaskLocality.RACK\_LOCAL))  
 }  
 }  
 }  
 }  
  
 if (TaskLocality.isAllowed(locality, TaskLocality.ANY)) {  
 for (index <- speculatableTasks if canRunOnHost(index)) {  
 speculatableTasks -= index // 找到可以在任何节点、机架上推断执行的Task  
 return Some((index, TaskLocality.ANY))  
 }  
 }  
 }  
 None  
}

根据代码清单7-63，dequeueSpeculativeTask方法的执行步骤如下。

1）从speculatableTasks中移除已经完成的Task，保留还未完成的Task。

2）对于speculatableTasks中的所有未在指定的Host上尝试运行，且指定的Host和Executor不在黑名单的所有Task进行以下处理：

①获取Task偏好的Executor；

②如果Task偏好的Executor中包含指定的Executor，那么将此Task的索引从speculatable-Tasks中移除，并返回此Task的索引与PROCESS\_LOCAL的对偶。

3）如果指定的本地性级别小于等于NODE\_LOCAL，对speculatableTasks中所有未在指定的Host上尝试运行，且指定的Host和Executor不在黑名单的所有Task进行以下处理：

①获取Task偏好的Host；

②如果Task偏好的Host中包含指定的Host，那么将此Task的索引从speculatable-Tasks中移除，并返回此Task的索引与NODE\_LOCAL的对偶。

4）如果指定的本地性级别小于等于NO\_PREF，对于speculatableTasks中的所有未在指定的Host上尝试运行，且指定的Host和Executor不在黑名单的所有Task进行以下处理：

①获取Task偏好的位置；

②如果Task没有偏好的位置信息，那么将此Task的索引从speculatableTasks中移除，并返回此Task的索引与PROCESS\_LOCAL的对偶。

5）如果指定的本地性级别小于等于RACK\_LOCAL，对于speculatableTasks中的所有未在指定的Host上尝试运行，且指定的Host和Executor不在黑名单的所有Task进行以下处理：

①获取Task偏好的机架；

②如果Task偏好的机架中包含指定的Host所在的机架，那么将此Task的索引从speculatableTasks中移除，并返回此Task的索引与RACK\_LOCAL的对偶。

6）如果指定的本地性级别小于等于ANY，对于speculatableTasks中的所有未在指定的Host上尝试运行，且指定的Host和Executor不在黑名单的所有Task进行以下处理：

①将此Task的索引从speculatableTasks中移除；

②返回此Task的索引与ANY的对偶。

### 7.6.4 Task本地性

与Hadoop类似，Spark对任务的处理也要考虑数据的本地性（Locality），好的数据本地性能够大幅减少节点间的数据传输，提升程序执行效率。Spark目前支持五种本地性级别，由高到低分别为：PROCESS\_LOCAL（本地进程），NODE\_LOCAL（本地节点），NO\_PREF（没有偏好），RACK\_LOCAL（本地机架），ANY（任何）。

Task本地性的分配优先考虑有较高的本地性的级别，否则分配较低的本地性级别，直到ANY。TaskSet可以有一到多个本地性级别，但在给Task分配本地性时只能是其中的一个。TaskSet中的所有Task都具有相同的允许使用的本地性级别，但在运行期可能因为资源不足、运行时间等因素，导致同一TaskSet中的各个Task的本地性级别可能不同。

TaskSet中实现的本地性操作包括对TaskSet的本地性级别进行计算、获取某个本地性级别的等待时间、给Task分配资源时获取允许的本地性级别等。

1.computeValidLocalityLevels

computeValidLocalityLevels方法（见代码清单7-64）用于计算有效的本地性级别，这样就可以将Task按照本地性级别，由高到低地分配给允许的Executor。

代码清单7-64 计算有效的本地性级别

private def computeValidLocalityLevels(): Array[TaskLocality.TaskLocality] = {  
 import TaskLocality.{PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY}  
 val levels = new ArrayBuffer[TaskLocality.TaskLocality]  
 if (!pendingTasksForExecutor.isEmpty && getLocalityWait(PROCESS\_LOCAL) != 0 &&  
 pendingTasksForExecutor.keySet.exists(sched.isExecutorAlive(\_))) {  
 levels += PROCESS\_LOCAL // 允许的本地性级别里包括PROCESS\_LOCAL  
 }  
 if (!pendingTasksForHost.isEmpty && getLocalityWait(NODE\_LOCAL) != 0 &&  
 pendingTasksForHost.keySet.exists(sched.hasExecutorsAliveOnHost(\_))) {  
 levels += NODE\_LOCAL // 允许的本地性级别里包括NODE\_LOCAL  
 }  
 if (!pendingTasksWithNoPrefs.isEmpty) {  
 levels += NO\_PREF // 允许的本地性级别里包括NO\_PREF  
 }  
 if (!pendingTasksForRack.isEmpty && getLocalityWait(RACK\_LOCAL) != 0 &&  
 pendingTasksForRack.keySet.exists(sched.hasHostAliveOnRack(\_))) {  
 levels += RACK\_LOCAL // 允许的本地性级别里包括RACK\_LOCAL  
 }  
 levels += ANY // 允许的本地性级别里增加ANY  
 logDebug("Valid locality levels for " + taskSet + ": " + levels.mkString(", "))  
 levels.toArray // 返回所有允许的本地性级别  
}

根据代码清单7-64，computeValidLocalityLevels方法的执行步骤如下。

1）如果存在Executor上待处理的Task的集合（即pendingTasksForExecutor不为空）且PROCESS\_LOCAL级别的等待时间不为0，还存在已被激活的Executor（即pendingTasks-ForExecutor中的ExecutorId有存在于TaskSchedulerImpl的executorIdToRunningTaskIds中的），那么允许的本地性级别里包括PROCESS\_LOCAL。

2）如果存在Host上待处理的Task的集合（即pendingTasksForHost不为空）且NODE\_LOCAL级别的等待时间不为0，除此以外，Host上存在已被激活的Executor（即pending-TasksForHost中的Host有存在于TaskSchedulerImpl的hostToExecutors中的），那么允许的本地性级别里包括NODE\_LOCAL。

3）如果存在没有任何本地性偏好的待处理Task，那么允许的本地性级别里包括NO\_PREF。

4）如果存在机架上待处理的Task的集合（即pendingTasksForRack不为空）且RACK\_LOCAL级别的等待时间不为0，除此以外，机架上存在已被激活的Executor（即pending TasksForRack中的机架有存在于TaskSchedulerImpl的hostsByRack中的），那么允许的本地性级别里包括RACK\_LOCAL。

5）允许的本地性级别里增加ANY。

6）返回所有允许的本地性级别。

2.getLocalityWait

getLocalityWait方法（见代码清单7-65）用于获取某个本地性级别的等待时间。

代码清单7-65 获取某个本地性级别的等待时间

private def getLocalityWait(level: TaskLocality.TaskLocality): Long = {  
 val defaultWait = conf.get("spark.locality.wait", "3s") // 获取默认的等待时间  
 val localityWaitKey = level match { // 根据本地性级别匹配到对应的配置属性  
 case TaskLocality.PROCESS\_LOCAL => "spark.locality.wait.process"  
 case TaskLocality.NODE\_LOCAL => "spark.locality.wait.node"  
 case TaskLocality.RACK\_LOCAL => "spark.locality.wait.rack"  
 case \_ => null  
 }  
 if (localityWaitKey != null) { // 获取本地性级别对应的等待时间  
 conf.getTimeAsMs(localityWaitKey, defaultWait)  
 } else {  
 0L  
 }  
}

根据代码清单7-65，getLocalityWait方法的执行步骤如下。

1）获取默认的等待时间。默认的等待时间可以通过spark.locality.wait属性配置，默认为3秒。

2）根据本地性级别匹配到对应的配置属性。PROCESS\_LOCAL级别的配置属性为spark.locality.wait.process,NODE\_LOCAL级别的配置属性为spark.locality.wait.node,RACK\_LOCAL级别的配置属性为spark.locality.wait.rack。其他级别没有配置属性。

3）使用第2）步得到的属性名称，获取本地性级别对应的等待时间。如果第2）步没有得到属性名称，那么将返回0。

3.getLocalityIndex

getLocalityIndex方法（见代码清单7-66）用于从myLocalityLevels中找出指定的本地性级别所对应的索引。

代码清单7-66 获取本地性级别所对应的索引

def getLocalityIndex(locality: TaskLocality.TaskLocality): Int = {  
 var index = 0  
 while (locality > myLocalityLevels(index)) {  
 index += 1  
 }  
 index  
}

4.getAllowedLocalityLevel

getAllowedLocalityLevel方法（见代码清单7-67）用于获取允许的本地性级别。

代码清单7-67 获取允许的本地性级别

private def getAllowedLocalityLevel(curTime: Long): TaskLocality.TaskLocality = {  
 while (currentLocalityIndex < myLocalityLevels.length - 1) {  
 val moreTasks = myLocalityLevels(currentLocalityIndex) match { // 查找本地性级别是否有Task要运行  
 case TaskLocality.PROCESS\_LOCAL => moreTasksToRunIn(pendingTasksForExecutor)  
 case TaskLocality.NODE\_LOCAL => moreTasksToRunIn(pendingTasksForHost)  
 case TaskLocality.NO\_PREF => pendingTasksWithNoPrefs.nonEmpty  
 case TaskLocality.RACK\_LOCAL => moreTasksToRunIn(pendingTasksForRack)  
 }  
 if (!moreTasks) {  
 lastLaunchTime = curTime // 没有Task需要处理，则将最后的运行时间设置为curTime  
 logDebug(s"No tasks for locality level ${myLocalityLevels(currentLocalityIndex)}, " +  
 s"so moving to locality level ${myLocalityLevels(currentLocalityIndex + 1)}")  
 currentLocalityIndex += 1  
 } else if (curTime - lastLaunchTime >= localityWaits(currentLocalityIndex)) {  
 lastLaunchTime += localityWaits(currentLocalityIndex) // 跳入更低的本地性级别  
 logDebug(s"Moving to ${myLocalityLevels(currentLocalityIndex + 1)} after waiting for " +  
 s"${localityWaits(currentLocalityIndex)}ms")  
 currentLocalityIndex += 1  
 } else {  
 return myLocalityLevels(currentLocalityIndex) // 返回当前本地性级别  
 }  
 }  
 myLocalityLevels(currentLocalityIndex) // 未能找到允许的本地性级别，那么返回最低的本地性级别  
}

根据代码清单7-67，getAllowedLocalityLevel方法的执行步骤如下。

1）按照索引由高到低从myLocalityLevels读取本地性级别，然后执行以下操作。

①调用moreTasksToRunIn方法（见代码清单7-76）判断本地性级别对应的待处理Task的缓存结构中是否有Task需要处理。

②如果没有Task需要处理，则将最后的运行时间设置为curTime。

③如果有Task需要处理且curTime与最后运行时间的差值大于当前本地性级别的等待时间，则将最后的运行时间增加当前本地性级别的等待时间（这样实际将直接跳入更低的本地性级别）。

④如果有Task需要处理且curTime与最后运行时间的差值小于等于当前本地性级别的等待时间，则返回当前本地性级别。

2）如果上一步未能找到允许的本地性级别，那么返回最低的本地性级别。

getAllowedLocalityLevel方法的执行步骤也可以用图7-11来表示。

经过对Spark任务本地性的分析后，读者可能觉得这样的代码实现有些过于复杂，并且在获取本地性级别的时候竟然每次都要等待一段本地性级别的等待时长，这种实现未免太过奇怪。正如刚开始说的，任何任务都希望被分配到可以从本地读取数据的节点上，以得到最大的性能提升。然而每个任务的运行时长都不是事先可以预料的，当一个任务在分配时，如果没有满足最佳本地性（PROCESS\_LOCAL）的资源，而一直固执地期盼得到最佳的资源，很有可能被已经占用最佳资源但是运行时间很长的任务耽搁，所以这些代码实现了当没有最佳本地性时，退而求其次，选择稍微差点的资源。

### 7.6.5 TaskSetManager的常用方法

TaskSetManager除提供了Task推断和本地性的方法外，还有很多方法。限于篇幅，本节将介绍TaskSetManager中其他最常用的方法。

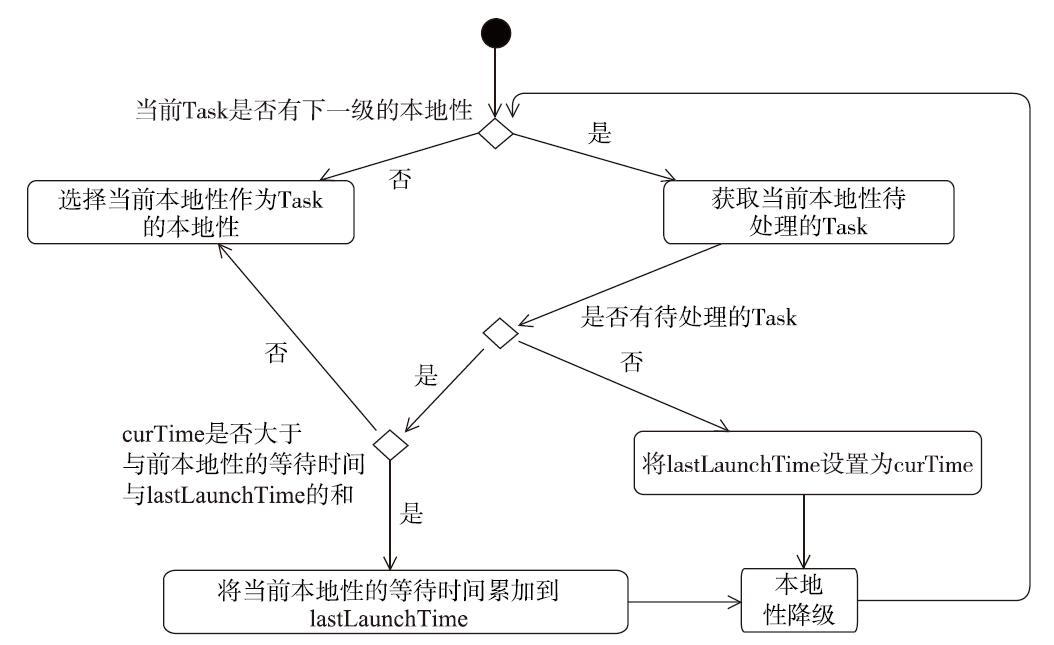


图7-11 获取允许的本地性级别

1.addPendingTask

addPendingTask方法（见代码清单7-68）用于将待处理Task的索引按照Task的偏好位置，添加到pendingTasksForExecutor、pendingTasksForHost、pendingTasksForRack、pending-TasksWithNoPrefs、allPendingTasks等缓存中。

代码清单7-68 添加待处理任务

private def addPendingTask(index: Int) {  
 for (loc <- tasks(index).preferredLocations) {  
 loc match {  
 case e: ExecutorCacheTaskLocation =>  
 pendingTasksForExecutor.getOrElseUpdate(e.executorId, new ArrayBuffer) += index  
 case e: HDFSCacheTaskLocation =>  
 val exe = sched.getExecutorsAliveOnHost(loc.host)  
 exe match {  
 case Some(set) =>  
 for (e <- set) {  
 pendingTasksForExecutor.getOrElseUpdate(e, new ArrayBuffer) += index  
 }  
 logInfo(s"Pending task $index has a cached location at ${e.host} " +  
 ", where there are executors " + set.mkString(","))  
 case None => logDebug(s"Pending task $index has a cached location at ${e.host} " +  
 ", but there are no executors alive there.")  
 }  
 case \_ =>  
 }  
 pendingTasksForHost.getOrElseUpdate(loc.host, new ArrayBuffer) += index  
 for (rack <- sched.getRackForHost(loc.host)) {  
 pendingTasksForRack.getOrElseUpdate(rack, new ArrayBuffer) += index  
 }  
 }  
  
 if (tasks(index).preferredLocations == Nil) {  
 pendingTasksWithNoPrefs += index  
 }  
  
 allPendingTasks += index // No point scanning this whole list to find the old task there  
}

2.dequeueTaskFromList

dequeueTaskFromList方法（见代码清单7-69）用于从给定的Task列表中按照索引，从高到低找出满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引。

代码清单7-69 dequeueTaskFromList的实现

private def dequeueTaskFromList(  
 execId: String,  
 host: String,  
 list: ArrayBuffer[Int]): Option[Int] = {  
 var indexOffset = list.size  
 while (indexOffset > 0) {  
 indexOffset -= 1  
 val index = list(indexOffset)  
 if (!isTaskBlacklistedOnExecOrNode(index, execId, host)) {  
 list.remove(indexOffset)  
 if (copiesRunning(index) == 0 && !successful(index)) {  
 return Some(index)  
 }  
 }  
 }  
 None  
}

3.dequeueTask

dequeueTask方法（见代码清单7-70）用于根据指定的Host、Executor和本地性级别，找出要执行的Task的索引、相应的本地性级别及是否进行推断执行。

代码清单7-70 dequeueTask的实现

private def dequeueTask(execId: String, host: String, maxLocality: TaskLocality.Value)  
 : Option[(Int, TaskLocality.Value, Boolean)] =  
{  
 for (index <- dequeueTaskFromList(execId, host, getPendingTasksForExecutor(execId))) {  
 return Some((index, TaskLocality.PROCESS\_LOCAL, false)) // 选择指定Executor的待处理Task  
 }  
  
 if (TaskLocality.isAllowed(maxLocality, TaskLocality.NODE\_LOCAL)) {  
 for (index <- dequeueTaskFromList(execId, host, getPendingTasksForHost(host))) {  
 return Some((index, TaskLocality.NODE\_LOCAL, false)) // 选择指定Host上的待处理Task  
 }  
 }  
 if (TaskLocality.isAllowed(maxLocality, TaskLocality.NO\_PREF)) {  
 for (index <- dequeueTaskFromList(execId, host, pendingTasksWithNoPrefs)) {  
 return Some((index, TaskLocality.PROCESS\_LOCAL, false)) // 选择没有本地性偏好的待处理Task  
 }  
 }  
 if (TaskLocality.isAllowed(maxLocality, TaskLocality.RACK\_LOCAL)) {  
 for {  
 rack <- sched.getRackForHost(host)  
 index <- dequeueTaskFromList(execId, host, getPendingTasksForRack(rack))  
 } {  
 return Some((index, TaskLocality.RACK\_LOCAL, false)) // 选择指定Host所在机架上的待处理Task  
 }  
 }  
 if (TaskLocality.isAllowed(maxLocality, TaskLocality.ANY)) {  
 for (index <- dequeueTaskFromList(execId, host, allPendingTasks)) {  
 return Some((index, TaskLocality.ANY, false)) // 选择在任意的本地性上的待处理Task  
 }  
 }  
  
 dequeueSpeculativeTask(execId, host, maxLocality).map { // 选择可推断的Task及其本地性  
 case (taskIndex, allowedLocality) => (taskIndex, allowedLocality, true)}  
}

根据代码清单7-70，dequeueTask方法的执行步骤如下。

1）从pendingTasksForExecutor中找出指定Executor的待处理Task中满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引，并返回此索引、PROCESS\_LOCAL、非推断执行标记（false）的三元组。

2）如果指定的本地性级别小于等于NODE\_LOCAL，调用dequeueTaskFromList方法（见代码清单7-69），从pendingTasksForHost中找出指定Host的待处理Task中满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引，并返回此索引、NODE\_LOCAL、非推断执行标记（false）的三元组。

3）如果指定的本地性级别小于等于NO\_PREF，调用dequeueTaskFromList方法（见代码清单7-69），从pendingTasksWithNoPrefs中找出满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引，并返回此索引、PROCESS\_LOCAL、非推断执行标记（false）的三元组。

4）如果指定的本地性级别小于等于RACK\_LOCAL，调用dequeueTaskFromList方法（见代码清单7-69），从pendingTasksForRack中找出指定Host所在机架的待处理Task中满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引，并返回此索引、RACK\_LOCAL、非推断执行标记（false）的三元组。

5）如果指定的本地性级别小于等于ANY，调用dequeueTaskFromList方法（见代码清单7-69），从allPendingTasks中找出满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引，并返回此索引、ANY、非推断执行标记（false）的三元组。

6）调用dequeueSpeculativeTask方法（见代码清单7-63），从可推断的Task中找出可推断的Task的索引和相应的本地性级别，并返回可推断的Task的索引、相应的本地性级别、推断执行标记（true）的三元组。

4.addRunningTask与removeRunningTask

addRunningTask方法（见代码清单7-71）用于向runningTasksSet中添加Task的身份标识，并调用TaskSetManager的父调度池的increaseRunningTasks方法（见代码清单7-54），增加父调度池及祖父调度池中记录的当前正在运行的任务数量。而removeRunning-Task方法则相反（见代码清单7-71）。

代码清单7-71 增加或减少正在运行的任务数量

def addRunningTask(tid: Long) {  
 if (runningTasksSet.add(tid) && parent != null) {  
 parent.increaseRunningTasks(1)  
 }  
}  
def removeRunningTask(tid: Long) {  
 if (runningTasksSet.remove(tid) && parent != null) {  
 parent.decreaseRunningTasks(1)  
 }  
}

5.maybeFinishTaskSet

maybeFinishTaskSet方法（见代码清单7-72）用于当TaskSet可能已经完成的时候进行一些清理工作。TaskSchedulerImpl的taskSetFinished方法的实现很简单，留给感兴趣的读者自行研究。

代码清单7-72 maybeFinishTaskSet的实现

private def maybeFinishTaskSet() {  
 if (isZombie && runningTasks == 0) {  
 sched.taskSetFinished(this)  
 }  
}

6.resourceOffer

resourceOffer方法（见代码清单7-73）用于给Task按照本地性分配资源。

代码清单7-73 给Task分配资源

def resourceOffer(  
 execId: String,  
 host: String,  
 maxLocality: TaskLocality.TaskLocality)  
 : Option[TaskDescription] =  
{  
 val offerBlacklisted = taskSetBlacklistHelperOpt.exists { blacklist =>  
 blacklist.isNodeBlacklistedForTaskSet(host) ||  
 blacklist.isExecutorBlacklistedForTaskSet(execId)  
 }  
 if (!isZombie && !offerBlacklisted) {  
 val curTime = clock.getTimeMillis()  
 var allowedLocality = maxLocality  
 if (maxLocality != TaskLocality.NO\_PREF) { // 计算允许的本地性级别  
 allowedLocality = getAllowedLocalityLevel(curTime)  
 if (allowedLocality > maxLocality) {  
 allowedLocality = maxLocality  
 }  
 }  
 // 根据指定的Host、Executor和本地性级别，找到合适的Task  
 dequeueTask(execId, host, allowedLocality).map { case ((index, taskLocality, speculative)) =>  
 val task = tasks(index) // 根据要执行的Task的索引找到要执行的Task  
 val taskId = sched.newTaskId() // 为Task生成新的身份标识  
 copiesRunning(index) += 1 // 增加复制运行数  
 val attemptNum = taskAttempts(index).size // 获取任务尝试号attemptNum  
 val info = new TaskInfo(taskId, index, attemptNum, curTime,  
 execId, host, taskLocality, speculative) // 创建Task尝试信息  
 taskInfos(taskId) = info // 将Task的身份标识与TaskInfo的对应关系放入taskInfos  
 // 将TaskInfo添加到taskAttempts中  
 taskAttempts(index) = info :: taskAttempts(index)  
 if (maxLocality != TaskLocality.NO\_PREF) { // 获取Task的本地性偏好级别  
 currentLocalityIndex = getLocalityIndex(taskLocality)  
 lastLaunchTime = curTime  
 }  
 val startTime = clock.getTimeMillis()  
 val serializedTask: ByteBuffer = try { // 序列化Task  
 Task.serializeWithDependencies(task, sched.sc.addedFiles, sched.sc.addedJars, ser)  
 } catch {  
 case NonFatal(e) =>  
 val msg = s"Failed to serialize task $taskId, not attempting to retry it."  
 logError(msg, e)  
 abort(s"$msg Exception during serialization: $e")  
 throw new TaskNotSerializableException(e)  
 }  
 if (serializedTask.limit > TaskSetManager.TASK\_SIZE\_TO\_WARN\_KB \* 1024 &&  
 !emittedTaskSizeWarning) {  
 emittedTaskSizeWarning = true  
 logWarning(s"Stage ${task.stageId} contains a task of very large size " +  
 s"(${serializedTask.limit / 1024} KB). The maximum recommended task size is " +  
 s"${TaskSetManager.TASK\_SIZE\_TO\_WARN\_KB} KB.")  
 }  
 addRunningTask(taskId) // 将Task放入正在运行的Task中  
  
 val taskName = s"task ${info.id} in stage ${taskSet.id}" // 生成Task的名称  
 logInfo(s"Starting $taskName (TID $taskId, $host, executor ${info.executorId}, " +  
 s"partition ${task.partitionId}, $taskLocality, ${serializedTask.limit} bytes)")  
 sched.dagScheduler.taskStarted(task, info)  
 new TaskDescription(taskId = taskId, attemptNumber = attemptNum, execId,  
 taskName, index, serializedTask) // 创建并返回TaskDescription对象  
 }  
 } else {  
 None  
 }  
}

根据代码清单7-73，resourceOffer方法在TaskSetManager处于“僵尸”状态并且分配Task的Host和Executor在黑名单中时直接返回None，否则执行如下步骤。

1）计算允许的本地性级别。如果最大本地性级别（即maxLocality）为NO\_PREF，则允许的本地性级别为NO\_PREF。如果maxLocality不是NO\_PREF，则允许的本地性级别为maxLocality和调用getAllowedLocalityLevel获取的本地性级别中较小的本地性级别。

2）调用dequeueTask方法（见代码清单7-70），根据指定的Host、Executor和本地性级别，找出三元组（包括要执行的Task的索引、相应的本地性级别、是否推断执行的标记），并对三元组进行如下操作。

①根据要执行的Task的索引找到要执行的Task。

②为Task生成新的身份标识。

③将Task对应的copiesRunning信息加一，即增加复制运行数。

④获取任务尝试号attemptNum。

⑤创建任务尝试信息（TaskInfo）。

⑥将Task的身份标识与TaskInfo的对应关系放入taskInfos。

⑦将TaskInfo添加到taskAttempts中Task对应的TaskInfo列表中。

⑧如果maxLocality不是NO\_PREF，那么调用getLocalityIndex方法（见代码清单7-66），获取任务的本地性偏好级别在myLocalityLevels中的索引，并将最后一次运行时间设置为当前系统时间。

⑨将Task、用户添加的Jar包及其他文件序列化，得到需要经过网络传输的序列化Task。

⑩如果序列化Task的大小超过了100KB，且emittedTaskSizeWarning仍然为false，则将emittedTaskSizeWarning设置为true并且打印警告日志。

Image00101.jpg 调用addRunningTask方法（见代码清单7-71），向runningTasksSet中添加Task的身份标识，并增加父调度池及祖父调度池中记录的当前正在运行的任务数量。

Image00102.jpg 生成Task的名称，格式为："task$index.$attemptNumber in stage$stageId.$stage AttemptId"。

Image00103.jpg 调用DAGScheduler的taskStarted方法向DAGSchedulerEventProcessLoop投递BeginEvent事件。

Image00104.jpg 创建并返回TaskDescription对象。

7.executorLost

executorLost方法在发生Executor丢失的情况下被调用，其实现如代码清单7-74所示。

代码清单7-74 executorLost的实现

override def executorLost(execId: String, host: String, reason: Executor LossReason) {  
 if (tasks(0).isInstanceOf[ShuffleMapTask] && !env.blockManager.externalShuffle ServiceEnabled) {  
 for ((tid, info) <- taskInfos if info.executorId == execId) {  
 val index = taskInfos(tid).index  
 if (successful(index)) {  
 successful(index) = false // 将此Task标记为未成功  
 copiesRunning(index) -= 1 // 将此Task的复制运行数量减一  
 tasksSuccessful -= 1 // 将当前TaskSetManager中成功执行的Task数量减一  
 addPendingTask(index) // 将此Task添加到待处理的Task中  
 sched.dagScheduler.taskEnded(  
 tasks(index), Resubmitted, null, Seq.empty, info) // 告知DAGScheduler重新提交Task  
 }  
 }  
 }  
 for ((tid, info) <- taskInfos if info.running && info.executorId == execId) {  
 val exitCausedByApp: Boolean = reason match { // 获取Executor丢失的具体原因是否是由应用程序引起的  
 case exited: ExecutorExited => exited.exitCausedByApp  
 case ExecutorKilled => false  
 case \_ => true  
 }  
 handleFailedTask(tid, TaskState.FAILED, ExecutorLostFailure(info.executorId, exitCausedByApp,  
 Some(reason.toString))) // 对失败的Task进行处理  
 }  
 recomputeLocality() // 重新计算本地性级别  
}

根据代码清单7-74，executorLost方法的执行步骤如下。

1）如果TaskSetManager管理的TaskSet中的Task为ShuffleMapTask，并且应用没有提供外部的Shuffle服务，那么对taskInfos中的所有在指定Executor上执行成功的Task执行以下操作。

①将此Task标记为未成功。

②将此Task的复制运行数量减一。

③将当前TaskSetManager中成功执行的Task数量减一。

④调用addPendingTask方法（见代码清单7-68），将此Task的索引添加到pendingTasksFor Executor、pendingTasksForHost、pendingTasksForRack、pendingTasksWithNoPrefs、allPending Tasks等缓存中。

⑤调用DAGScheduler的taskEnded方法向DAGSchedulerEventProcessLoop发送Completion Event。此时的CompletionEvent将携带Resubmitted，避免CompletionEvent的关心者在Stage的所有Task都完成后认为当前的Stage也执行完成，而是让它们知道此Task将被重新提交。

2）对taskInfos中的所有在身份标识为execId的Executor上正在运行的Task执行以下操作。

①从ExecutorLossReason中获取Executor丢失的具体原因是否是由应用程序引起的。

②调用handleFailedTask方法（由于此方法不属于主线逻辑，所以留给感兴趣的读者自行阅读）对失败的Task进行处理。

3）重新计算本地性级别。

8.tasksNeedToBeScheduledFrom

tasksNeedToBeScheduledFrom方法（见代码清单7-75）用于判断给定的待处理Task数组中是否有需要调度的Task。

代码清单7-75 tasksNeedToBeScheduledFrom的实现

def tasksNeedToBeScheduledFrom(pendingTaskIds: ArrayBuffer[Int]): Boolean = {  
 var indexOffset = pendingTaskIds.size  
 while (indexOffset > 0) {  
 indexOffset -= 1  
 val index = pendingTaskIds(indexOffset)  
 if (copiesRunning(index) == 0 && !successful(index)) {  
 return true  
 } else {  
 pendingTaskIds.remove(indexOffset)  
 }  
 }  
 false  
}

根据代码清单7-75，判断一个Task是否需要调度的条件是Task没有被复制运行且Task还未执行成功。

9.moreTasksToRunIn

moreTasksToRunIn方法（见代码清单7-76）用于判断待处理的Task集合（数据结构为HashMap[String,ArrayBuffer[Int]]）中是否有需要调度的Task。如果有，则返回true，否则将此key与Task集合的映射关系从待处理的Task集合中移除并返回false。

代码清单7-76 moreTasksToRunIn的实现

def moreTasksToRunIn(pendingTasks: HashMap[String, ArrayBuffer[Int]]): Boolean = {  
 val emptyKeys = new ArrayBuffer[String]  
 val hasTasks = pendingTasks.exists {  
 case (id: String, tasks: ArrayBuffer[Int]) =>  
 if (tasksNeedToBeScheduledFrom(tasks)) {  
 true  
 } else {  
 emptyKeys += id  
 false  
 }  
 }  
 emptyKeys.foreach(id => pendingTasks.remove(id))  
 hasTasks  
}

10.handleSuccessfulTask

handleSuccessfulTask方法（见代码清单7-77）用于对Task的执行结果（对于map任务而言，实际是任务状态）进行处理。

代码清单7-77 对Task的执行结果进行处理

def handleSuccessfulTask(tid: Long, result: DirectTaskResult[\_]): Unit = {  
 val info = taskInfos(tid)   
 val index = info.index  
 info.markFinished(TaskState.FINISHED) // 将taskInfos中缓存的TaskInfo标记为已经完成  
 removeRunningTask(tid) // 将Task从正在运行的Task集合中移除  
 // 将Task的执行结果交给DagScheduler处理  
 sched.dagScheduler.taskEnded(tasks(index), Success, result.value(), result.accumUpdates, info)  
 for (attemptInfo <- taskAttempts(index) if attemptInfo.running) { //"杀死"此Task正在运行的Task尝试  
 logInfo(s"Killing attempt ${attemptInfo.attemptNumber} for task ${attemptInfo.id} " +  
 s"in stage ${taskSet.id} (TID ${attemptInfo.taskId}) on ${attemptInfo.host} " +  
 s"as the attempt ${info.attemptNumber} succeeded on ${info.host}")  
 sched.backend.killTask(attemptInfo.taskId, attemptInfo.executorId, true)  
 }  
 if (!successful(index)) {  
 tasksSuccessful += 1 // 增加运行成功的Task数量  
 logInfo(s"Finished task ${info.id} in stage ${taskSet.id} (TID ${info.taskId}) in" +  
 s" ${info.duration} ms on ${info.host} (executor ${info.executorId})" +  
 s" ($tasksSuccessful/$numTasks)")  
 successful(index) = true // 将此任务的状态设置为true  
 if (tasksSuccessful == numTasks) {  
 isZombie = true // TaskSet中的所有Task都调度运行成功，于是将isZombie设置为true  
 }  
 } else {  
 logInfo("Ignoring task-finished event for " + info.id + " in stage " + taskSet.id +  
 " because task " + index + " has already completed successfully")  
 }  
 maybeFinishTaskSet() // 在TaskSet可能已经完成的时候进行一些清理工作  
}

根据代码清单7-77，handleSuccessfulTask方法的执行步骤如下。

1）将taskInfos中缓存的TaskInfo标记为已经完成。

2）将Task的ID从正在运行的Task集合中移除，并且减少正在运行的Task的数量。

3）调用DAGScheduler的taskEnded方法（见代码清单7-45），将Task的执行结果交给DagScheduler处理。

4）“杀死”此Task正在运行的Task尝试。

5）增加运行成功的Task数量，并且将successful中代表此任务的状态设置为true。

6）如果成功运行的Task数量与TaskSet中的Task数量相同，说明此TaskSet中的所有Task都调度运行成功，于是将isZombie设置为true。

7）调用maybeFinishTaskSet方法（见代码清单7-72），在TaskSet可能已经完成的时候进行一些清理工作。

## 7.7 运行器后端接口LauncherBackend

要介绍LauncherBackend，首先应该介绍下LauncherServer。当Spark应用程序没有在用户应用程序中运行，而是运行在单独的进程中时，用户可以在用户应用程序中使用LauncherServer与Spark应用程序通信。LauncherServer将提供Socket连接的服务端，与Spark应用程序中的Socket连接的客户端通信。LauncherServer的工作原理可以用图7-12来表示。

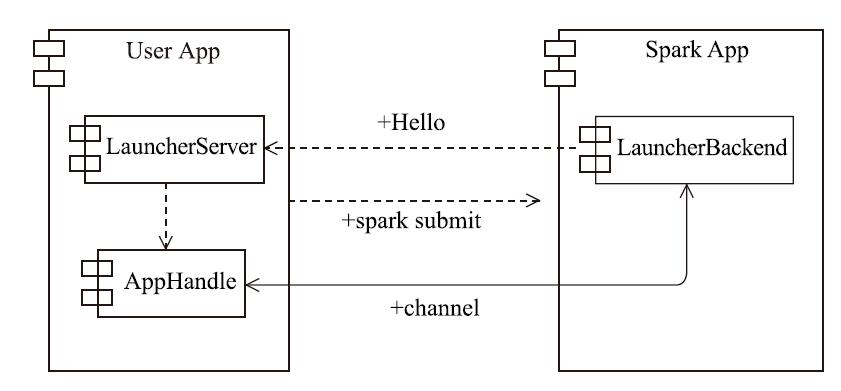


图7-12 LauncherServer的工作原理

由于TaskSchedulerImpl的底层依赖于LauncherBackend，所以本节需要对Launcher Backend进行介绍。LauncherBackend依赖于BackendConnection，跟Launcher Server进行通信，LauncherBackend的主要工作流程如图7-13所示。

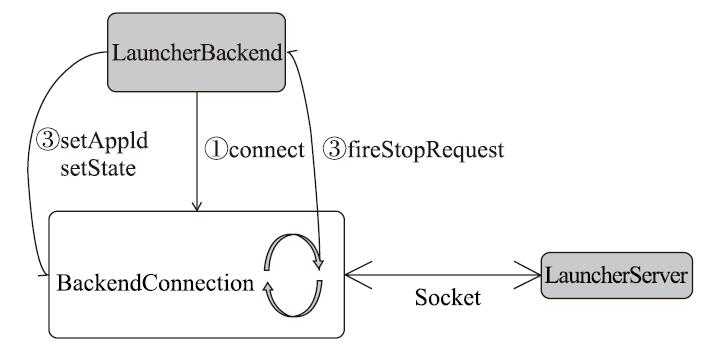


图7-13 LauncherBackend的主要工作流程

这里对图7-13中的各个记号进行说明。

记号①： 调用LauncherBackend的connect方法创建BackendConnection，并且创建线程执行BackendConnection。构造BackendConnection的过程中，BackendConnection会和LauncherServer之间建立起Socket连接。BackendConnection（实现了java.lang.Runnable接口）将不断从Socket连接中读取LauncherServer发送的数据。

记号②： 调用LauncherBackend的setAppId方法（见代码清单7-84）或SetState方法（见代码清单7-85），通过Socket连接向LauncherServer发送SetAppId消息或SetState消息。

记号③： BackendConnection从Socket连接中读取到LauncherServer发送的Stop消息，然后调用LauncherBackend的fireStopRequest方法停止请求。

下面将详细介绍LauncherBackend及其内部组件BackendConnection的实现。

### 7.7.1 BackendConnection的实现

BackendConnection是LauncherBackend的内部组件，用于保持与LauncherServer的Socket连接，并通过此Socket连接收发消息。BackendConnection继承了LauncherConnection，LauncherConnection提供了维护连接和收发消息的基本实现。

BackendConnection继承了抽象类LauncherConnection的属性，分别如下。

·socket：与LauncherServer的Socket服务端建立连接的Socket客户端。

·out：建立在Socket的输出流上的ObjectOutputStream，用于向服务端发送消息。

·closed：Socket客户端与LauncherServer的Socket服务端建立的连接是否已经关闭的状态。

有了对BackendConnection属性的了解，现在来看看BackendConnection提供的方法。

1.handle

handle是LauncherConnection提供的用于处理LauncherServer发送的消息的抽象方法，其定义如下。

protected abstract void handle(Message msg) throws IOException;

protected abstract void handle(Message msg)throws IOException;

BackendConnection实现了LauncherConnection的handle方法，如代码清单7-78所示。

代码清单7-78 handle的实现

override protected def handle(m: Message): Unit = m match {  
 case \_: Stop =>  
 fireStopRequest()  
  
 case \_ =>  
 throw new IllegalArgumentException(s"Unexpected message type: ${m.getClass().getName()}")  
}

根据代码清单7-78，BackendConnection实现的handle方法只处理Stop这一种消息。对于Stop消息，BackendConnection将调用外部类LauncherBackend的fireStopRequest方法（见代码清单7-86）停止Executor。

2.run

由于LauncherConnection实现了java.lang.Runnable接口，因此需要实现run方法。LauncherConnection的run方法（见代码清单7-79）用于从Socket客户端的输入流中读取LauncherServer发送的消息，并调用handle方法对消息进行处理。LauncherConnection的run方法同时也是一个模板方法。

代码清单7-79 LauncherConnection的run方法

@Override  
public void run() {  
 try {  
 ObjectInputStream in = new ObjectInputStream(socket.getInputStream());  
 while (!closed) {  
 Message msg = (Message) in.readObject();  
 handle(msg);  
 }  
 } catch (EOFException eof) {  
 try {  
 close();  
 } catch (Exception unused) {  
 // no-op.  
 }  
 } catch (Exception e) {  
 if (!closed) {  
 LOG.log(Level.WARNING, "Error in inbound message handling.", e);  
 try {  
 close();  
 } catch (Exception unused) {  
 // no-op.  
 }  
 }  
 }  
}

3.send

此方法通过Socket客户端与LauncherServer的Socket服务端建立的连接向LauncherServer发送消息，其实现如代码清单7-80所示。

代码清单7-80 向LauncherServer发送消息

protected synchronized void send(Message msg) throws IOException {  
 try {  
 CommandBuilderUtils.checkState(!closed, "Disconnected.");  
 out.writeObject(msg);  
 out.flush();  
 } catch (IOException ioe) {  
 if (!closed) {  
 LOG.log(Level.WARNING, "Error when sending message.", ioe);  
 try {  
 close();  
 } catch (Exception unused) {  
 // no-op.  
 }  
 }  
 throw ioe;  
 }  
}

4.close

此方法用于关闭Socket客户端与LauncherServer的Socket服务端建立的连接，其实现如代码清单7-81所示。

代码清单7-81 关闭连接

@Override  
public void close() throws IOException {  
 if (!closed) {  
 synchronized (this) {  
 if (!closed) {  
 closed = true;  
 socket.close();  
 }  
 }  
 }  
}

BackendConnection重写了LauncherConnection的close方法，其实现如代码清单7-82所示。

代码清单7-82 BackendConnection的close方法

override def close(): Unit = {  
 try {  
 super.close()  
 } finally {  
 onDisconnected()  
 \_isConnected = false  
 }  
}

根据代码清单7-82，BackendConnection重写的close方法首先调用了父类Launcher-Connection的close方法关闭Socket连接，然后调用外部类LauncherBackend的onDisconnected方法。由于LauncherBackend的onDisconnected是个空方法，因此onDisconnected并不会有额外的效果。

### 7.7.2 LauncherBackend的实现

LauncherBackend是SchedulerBackend与LauncherServer通信的组件，我们先来介绍其属性信息，然后分析LauncherBackend的具体实现。

LauncherBackend包含的属性如下。

·clientThread：读取与LauncherServer建立的Socket连接上的消息的线程。

·connection：即BackendConnection实例。

·lastState：LauncherBackend的最后一次状态。lastState的类型是枚举类型Spark-AppHandle.State。SparkAppHandle.State共有未知（UNKNOWN）、已连接（CONNE-CTED）、已提交（SUBMITTED）、运行中（RUNNING）、已完成（FINISHED）、已失败（FAILED）、已被杀（KILLED）、丢失（LOST）等状态。

·\_isConnected：clientThread是否与LauncherServer已经建立了Socket连接的状态。

LauncherBackend提供的方法如下。

1.connect

connect方法（见代码清单7-83）用于和LauncherServer建立连接。

代码清单7-83 建立连接

def connect(): Unit = {  
 val port = sys.env.get(LauncherProtocol.ENV\_LAUNCHER\_PORT).map(\_.toInt)  
 val secret = sys.env.get(LauncherProtocol.ENV\_LAUNCHER\_SECRET)  
 if (port != None && secret != None) { // 创建与LauncherServer的Socket服务端建立连接的Socket  
 val s = new Socket(InetAddress.getLoopbackAddress(), port.get)  
 connection = new BackendConnection(s)  
 connection.send(new Hello(secret.get, SPARK\_VERSION)) // 向LauncherServer发送Hello消息  
 clientThread = LauncherBackend.threadFactory.newThread(connection)  
 clientThread.start() // 创建并启动一个执行BackendConnection的run方法的线程  
 \_isConnected = true // 设置已连接的状态  
 }  
}

根据代码清单7-83，connect方法的执行步骤如下。

1）创建与LauncherServer的Socket服务端建立连接的Socket。LauncherServer的Socket端口和密钥可通过系统环境变量\_SPARK\_LAUNCHER\_PORT和\_SPARK\_LAUNCHER\_SECRET指定。

2）通过此连接向LauncherServer发送Hello消息。Hello消息携带着应用程序的Spark版本号和密钥信息。

3）创建并启动一个执行BackendConnection的run方法的线程。

4）将\_isConnected设置为true。

2.setAppId

setAppId方法（见代码清单7-84）用于向LauncherServer发送SetAppId消息。SetAppId消息携带着应用程序的身份标识。

代码清单7-84 向LauncherServer发送SetAppId消息

def setAppId(appId: String): Unit = {  
 if (connection != null) {  
 connection.send(new SetAppId(appId))  
 }  
}

3.setState

SetState方法（见代码清单7-85）用于向LauncherServer发送SetState消息。SetState消息携带着LauncherBackend的最后一次状态。

代码清单7-85 向LauncherServer发送SetState消息

def setState(state: SparkAppHandle.State): Unit = {  
 if (connection != null && lastState != state) {  
 connection.send(new SetState(state))  
 lastState = state  
 }  
}

4.isConnected

isConnected方法返回clientThread是否与LauncherServer已经建立了Socket连接的状态。

def isConnected(): Boolean = \_isConnected

5.onStopRequest

OnStopRequest是LauncherBackend定义的处理LauncherServer的停止消息的抽象方法，代码定义如下。

protected def onStopRequest(): Unit

6.onDisconnected

onDisconnected方法用于在关闭Socket客户端与LauncherServer的Socket服务端建立的连接时，进行一些额外的处理，但目前只是一个空方法。

protected def onDisconnected() : Unit = { }

7.fireStopRequest

fireStopRequest（见代码清单7-86）用于启动一个调用onStopRequest方法的线程。

代码清单7-86 fireStopRequest的实现

private def fireStopRequest(): Unit = {  
 val thread = LauncherBackend.threadFactory.newThread(new Runnable() {  
 override def run(): Unit = Utils.tryLogNonFatalError {  
 onStopRequest()  
 }  
 })  
 thread.start()  
}

## 7.8 调度后端接口SchedulerBackend

SchedulerBackend是TaskScheduler的调度后端接口。TaskScheduler给Task分配资源实际是通过SchedulerBackend来完成的，SchedulerBackend给Task分配完资源后将与分配给Task的Executor通信，并要求后者运行Task。

### 7.8.1 SchedulerBackend的定义

特质SchedulerBackend定义了所有调度后端接口的行为规范，其定义如代码清单7-87所示。

代码清单7-87 SchedulerBackend的定义

private[spark] trait SchedulerBackend {  
 private val appId = "spark-application-" + System.currentTimeMillis  
 def start(): Unit  
 def stop(): Unit  
 def reviveOffers(): Unit  
 def defaultParallelism(): Int  
  
 def killTask(taskId: Long, executorId: String, interruptThread: Boolean): Unit =  
 throw new UnsupportedOperationException  
 def isReady(): Boolean = true  
 def applicationId(): String = appId  
 def applicationAttemptId(): Option[String] = None  
 def getDriverLogUrls: Option[Map[String, String]] = None  
}

根据代码清单7-87，SchedulerBackend定义了以下成员。

·appId：与当前Job相关联的应用程序的身份标识。

·start：启动SchedulerBackend，需要子类实现。

·stop：停止SchedulerBackend，需要子类实现。

·reviveOffers：给调度池中的所有Task分配资源。

·defaultParallelism：获取Job的默认并行度。

·killTask：“杀死”指定的任务。可以通过设置interruptThread为true来中断任务执行线程。

·isReady：SchedulerBackend是否准备就绪。

·applicationId：获取appId。

·applicationAttemptId：当应用在cluster模式运行且集群管理器支持应用进行多次执行尝试时，此方法可以获取应用程序尝试的标识。当应用程序在client模式运行时，将不支持多次尝试，因此此方法不会获取到应用程序尝试的标识。

Image00035.jpg 注意 SparkContext的\_applicationId属性和\_applicationAttemptId属性是通过分别调用Task SchedulerImpl的applicationId方法和applicationAttemptId方法获得的，而TaskScheduler Impl的applicationId方法和applicationAttemptId方法实际又分别调用了Scheduler Backend的applicationId方法和applicationAttemptId方法。

·getDriverLogUrls：获取Driver日志的Url。这些Url将被用于在Spark UI的Executors标签页中展示。

了解了SchedulerBackend的接口定义，现在一起来看看在Spark中有哪些Scheduler-Backend的实现类，如图7-14所示。

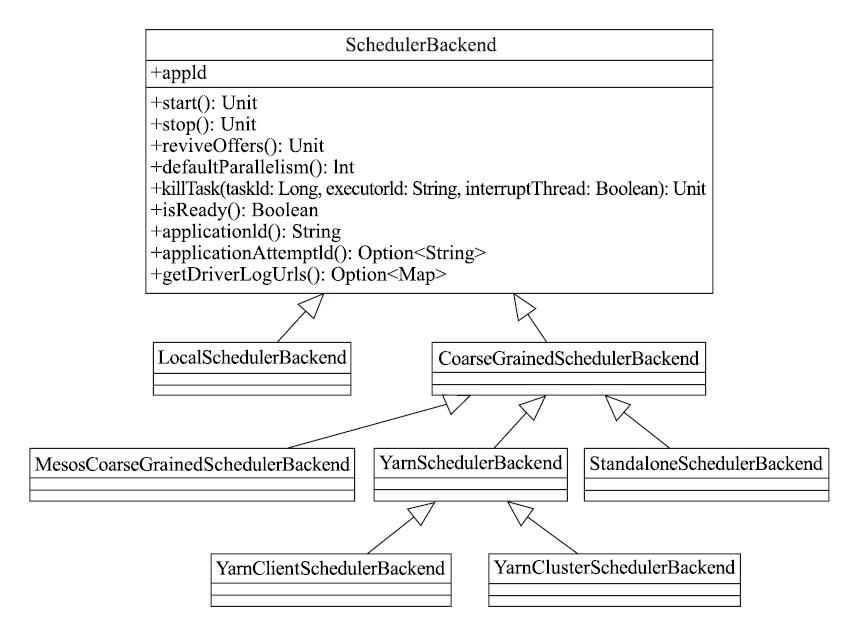


图7-14 SchedulerBackend的继承体系

从图7-14中可以看到SchedulerBackend有两个子类，分别如下。

1）CoarseGrainedSchedulerBackend：等待CoarseGrainedExecutorBackend进行连接的SchedulerBackend实现。由CoarseGrainedSchedulerBackend建立的CoarseGrainedExecutor-Backend进程将会一直存在，真正的Executor线程将在Coarse GrainedExecutor-Backend进程中执行。

2）LocalSchedulerBackend：local模式中的调度后端接口。在local模式下，Executor、LocalSchedulerBackend、Driver都运行在同一个JVM进程中。

CoarseGrainedSchedulerBackend的子类StandaloneSchedulerBackend是Standalone部署模式下的SchedulerBackend实现。对于Mesos或YAM，分别实现了继承自CoarseGrained Scheduler Backend的MesosCoarseGrainedSchedulerBackend和YarnScheduler Backend。YarnSchedulerBackend还有YarnClientSchedulerBackend和YarnCluster SchedulerBackend两个子类。

### 7.8.2 LocalSchedulerBackend的实现分析

我们虽然对特质SchedulerBackend的定义及继承体系有所了解，但是我们依然不是很明白SchedulerBackend的子类该如何实现。这里以SchedulerBackend在local部署模式下的实现类LocalSchedulerBackend为例，来详细分析其实现内容。

1.LocalEndpoint的实现分析

LocalSchedulerBackend与其他组件的通信都依赖于LocalEndpoint，所以我们需要先来认识它。LocalEndpoint包含以下属性。

·rpcEnv：即RpcEnv。

·userClassPath：用户指定的ClassPath。

·scheduler：即Driver中的TaskSchedulerImpl。

·executorBackend：与LocalEndpoint相关联的LocalSchedulerBackend。

·totalCores：用于执行任务的CPU内核总数。local模式下，totalCores固定为1。

·freeCores：空闲的CPU内核数。应用程序提交的Task正式运行之前，freeCores与totalCores相等。

·localExecutorId：local部署模式下，与Driver处于同一JVM进程的Executor的身份标识。由于LocalEndpoint只在local模式中使用，因此localExecutorId固定为driver。

·localExecutorHostname：与Driver处于同一JVM进程的Executor所在的Host。由于LocalEndpoint只在local模式中使用，因此localExecutorHostname固定为localhost。

·executor：与Driver处于同一JVM进程的Executor。由于LocalEndpoint的totalCores等于1，因此应用本地有且只有一个Executor，且此Executor在LocalEndpoint构造的过程中就已经实例化。

LocalEndpoint其实现如代码清单7-88所示。重写了RpcEndpoint的receive方法和receive AndReply方法。

代码清单7-88 LocalEndpoint接收的消息实现

override def receive: PartialFunction[Any, Unit] = {  
 case ReviveOffers =>  
 reviveOffers()  
 case StatusUpdate(taskId, state, serializedData) =>  
 scheduler.statusUpdate(taskId, state, serializedData)  
 if (TaskState.isFinished(state)) {  
 freeCores += scheduler.CPUS\_PER\_TASK  
 reviveOffers()  
 }  
 case KillTask(taskId, interruptThread) =>  
 executor.killTask(taskId, interruptThread)  
}  
  
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {  
 case StopExecutor =>  
 executor.stop()  
 context.reply(true)  
}

receive方法在处理ReviveOffers和StatusUpdate消息时，都会调用reviveOffers（见代码清单7-89）方法给Task分配资源。

代码清单7-89 LocalEndpoint的reviveOffers

def reviveOffers() {  
 val offers = IndexedSeq(new WorkerOffer(localExecutorId, localExecutorHostname, freeCores))  
 for (task <- scheduler.resourceOffers(offers).flatten) { // 给Task分配资源  
 freeCores -= scheduler.CPUS\_PER\_TASK // 将空闲CPU内核数freeCores减1  
 executor.launchTask(executorBackend, taskId = task.taskId, attemptNumber = task.attemptNumber,  
 task.name, task.serializedTask) // 运行Task  
 }  
}

根据代码清单7-89，LocalEndpoint的reviveOffers方法的执行步骤如下。

private[spark] case class WorkerOffer(executorId: String, host: String, cores: Int)

1）创建只包含一个WorkerOffer（理解为“就业机会”似乎更加生动，由于totalCores为1，因此只有一个职位）的序列。样例类WorkerOffer的实现如下。

根据对LocalEndpoint的分析，此处WorkerOffer的executorId为driver，host为localhost，cores为1。

2）调用TaskSchedulerImpl的resourceOffers方法（见代码清单7-101）给Task分配资源。

3）将空闲CPU内核数freeCores减1。

4）调用Executor的launchTask方法运行Task。Executor将在第9章详细介绍。

2.LocalSchedulerBackend的实现

我们从了解LocalSchedulerBackend的属性开始，逐步分析LocalSchedulerBackend的实现。LocalSchedulerBackend的属性如下。

·conf：即SparkConf。

·scheduler：即TaskSchedulerImpl。

·totalCores：LocalSchedulerBackend的CPU内核数，固定为1。

·appId：当前应用程序的身份标识。local模式下的appId以local-为前缀，以系统当前时间戳为后缀，local模式下应用程序的身份标识是在构造LocalSchedulerBackend的时候生成的。

·localEndpoint：即LocalEndpoint的NettyRpcEndpointRef。

·userClassPath：用户指定的类路径。可以通过spark.executor.extraClassPath属性进行配置，配置时可以用英文逗号分隔多个类路径。

·listenerBus：即SparkContext中创建的LiveListenerBus。

·launcherBackend：LauncherBackend的匿名实现类的实例。此匿名实现类实现了LauncherBackend的onStopRequest方法（见代码清单7-90），用于停止Executor、将launcherBackend的状态标记为KILLED、关闭launcherBackend与LauncherServer之间的Socket连接。

代码清单7-90 LauncherBackend的匿名实现类

private val launcherBackend = new LauncherBackend() {  
 override def onStopRequest(): Unit = stop(SparkAppHandle.State.KILLED)  
}

在构造LocalSchedulerBackend的最后，会调用launcherBackend的connect方法（见代码清单7-83）与LauncherServer进行连接。

launcherBackend.connect()

最后来看看LocalSchedulerBackend提供的常用方法。

·start：启动LocalSchedulerBackend。LocalSchedulerBackend的start方法的实现如代码清单7-91所示。

代码清单7-91 启动LocalSchedulerBackend

override def start() {  
 val rpcEnv = SparkEnv.get.rpcEnv  
 val executorEndpoint = new LocalEndpoint(rpcEnv, userClassPath, scheduler, this, totalCores)  
 localEndpoint = rpcEnv.setupEndpoint("LocalSchedulerBackendEndpoint", executor Endpoint)  
 listenerBus.post(SparkListenerExecutorAdded(  
 System.currentTimeMillis,  
 executorEndpoint.localExecutorId,  
 new ExecutorInfo(executorEndpoint.localExecutorHostname, totalCores, Map.empty)))  
 launcherBackend.setAppId(appId) // 向LauncherServer发送SetAppId消息  
 launcherBackend.setState(SparkAppHandle.State.RUNNING) // 向LauncherServer发送SetState消息  
}

根据代码清单7-91，LocalSchedulerBackend的start方法的执行步骤如下。

1）创建LocalEndpoint并注册到RpcEnv中，然后由localEndpoint属性持有LocalEndpoint的NettyRpcEndpointRef。

2）向LiveListenerBus投递SparkListenerExecutorAdded事件。local模式下SparkListener ExecutorAdded事件携带的time为系统当前时间，executorId为driver，executorInfo（即ExecutorInfo，ExecutorInfo的executorHost属性为localhost，totalCores为1，logUrlMap为空）。

3）调用LauncherBackend的setAppId方法向LauncherServer发送SetAppId消息。

4）调用LauncherBackend的setState方法向LauncherServer发送SetState消息。

·reviveOffers：对Task进行资源分配后运行Task。LocalSchedulerBackend的reviveOffers方法的实现如代码清单7-92所示。

代码清单7-92 LocalSchedulerBackend的reviveOffers方法

override def reviveOffers() {  
 localEndpoint.send(ReviveOffers)  
}

根据代码清单7-92，LocalSchedulerBackend的reviveOffers方法将向LocalEndpoint发送ReviveOffers消息。LocalEndpoint接收ReviveOffers消息后（见代码清单7-88），会调用reviveOffers方法给下一个要调度的Task分配资源并运行Task。

·statusUpdate：Task的状态更新。LocalSchedulerBackend实现了特质ExecutorBackend的唯一方法statusUpdate，其实现如代码清单7-93所示。

代码清单7-93 Task的状态更新

override def statusUpdate(taskId: Long, state: TaskState, serializedData: ByteBuffer) {  
 localEndpoint.send(StatusUpdate(taskId, state, serializedData))  
}

根据代码清单7-93，LocalSchedulerBackend的statusUpdate方法将向LocalEndpoint发送StatusUpdate消息。LocalEndpoint接收到StatusUpdate消息后（见代码清单7-88），会首先调用TaskSchedulerImpl的statusUpdate方法（见代码清单7-103）更新Task状态。

## 7.9 任务结果获取器TaskResultGetter

TaskResultGetter用于对序列化的Task执行结果进行反序列化，以得到Task执行结果。TaskResultGetter也可远程获取Task执行结果。

TaskResultGetter包含以下属性。

·sparkEnv：即SparkEnv。

·scheduler：即TaskSchedulerImpl。

·THREADS：获取Task执行结果的线程数。可通过spark.resultGetter.threads属性配置，默认为4。

·getTaskResultExecutor：使用Executors的newFixedThreadPool方法创建的ThreadPool Executor，用于提交获取Task执行结果的线程。线程池的大小由THREADS决定。

·serializer：类型为ThreadLocal[SerializerInstance]，通过使用本地线程缓存，保证在使用SerializerInstance时是线程安全的。

·taskResultSerializer：类型为ThreadLocal[SerializerInstance]，通过使用本地线程缓存，保证在使用SerializerInstance对Task的执行结果进行反序列化时是线程安全的。

### 7.9.1 处理成功的Task

TaskResultGetter的enqueueSuccessfulTask方法用于处理执行成功的Task的执行结果，其实现如代码清单7-94所示。

代码清单7-94 enqueueSuccessfulTask方法

def enqueueSuccessfulTask(  
 taskSetManager: TaskSetManager,  
 tid: Long,  
 serializedData: ByteBuffer): Unit = {  
 getTaskResultExecutor.execute(new Runnable {  
 override def run(): Unit = Utils.logUncaughtExceptions {  
 try { // 对Task的执行结果反序列化  
 val (result, size) = serializer.get().deserialize[TaskResult[\_]](serialized Data) match {  
 case directResult: DirectTaskResult[\_] =>  
 if (!taskSetManager.canFetchMoreResults(serializedData.limit())) {  
 return  
 }  
 directResult.value(taskResultSerializer.get()) // 获取Task执行结果  
 (directResult, serializedData.limit())  
 case IndirectTaskResult(blockId, size) =>  
 if (!taskSetManager.canFetchMoreResults(size)) {  
 sparkEnv.blockManager.master.removeBlock(blockId)  
 return  
 }  
 logDebug("Fetching indirect task result for TID %s".format(tid))  
 scheduler.handleTaskGettingResult(taskSetManager, tid)  
 val serializedTaskResult = sparkEnv.blockManager.getRemoteBytes (blockId) // 下载结果  
 if (!serializedTaskResult.isDefined) {  
 scheduler.handleFailedTask(  
 taskSetManager, tid, TaskState.FINISHED, TaskResultLost)  
 return  
 }  
 val deserializedResult = serializer.get().deserialize[DirectTask Result[\_]](  
 // 对下载到的数据反序列化得到Task的执行结果  
 serializedTaskResult.get.toByteBuffer)  
 deserializedResult.value(taskResultSerializer.get())  
 sparkEnv.blockManager.master.removeBlock(blockId)  
 (deserializedResult, size)  
 }  
  
 result.accumUpdates = result.accumUpdates.map { a =>  
 if (a.name == Some(InternalAccumulator.RESULT\_SIZE)) {  
 val acc = a.asInstanceOf[LongAccumulator]  
 assert(acc.sum == 0L, "task result size should not have been set on the executors")  
 acc.setValue(size.toLong)  
 acc  
 } else {  
 a  
 }  
 }  
 scheduler.handleSuccessfulTask(taskSetManager, tid, result) //  
 } catch {  
 // 忽略异常处理代码.  
 }  
 }  
 })  
}

根据代码清单7-94，enqueueSuccessfulTask方法实际向getTaskResultExecutor提交了一个获取Task执行结果的任务，这个任务的执行步骤如下。

1）对Task的执行结果反序列化，如果Task的结果类型为DirectTaskResult，说明Task的执行结果保存在DirectTaskResult中，此时只需要对DirectTaskResult保存的数据（即DirectTaskResult的valueBytes属性）进行反序列化就可以得到。

2）对Task的执行结果反序列化，如果Task的结果类型为IndirectTaskResult，说明Task的执行结果没有保存在IndirectTaskResult中，此时需要调用TaskSchedulerImpl的handleTaskGettingResult方法（此方法的实现非常简单，这里就不过多介绍了），向DAGSchedulerEventProcessLoop投递GettingResultEvent事件，然后调用BlockManager的getRemoteBytes方法（见代码清单6-75），从运行Task的节点上下载Block，最后对下载到的数据反序列化得到Task的执行结果。

3）更新Task的执行结果的累加器中的结果大小。

4）调用TaskSchedulerImpl的handleSuccessfulTask方法（见代码清单7-105）。

Image00042.jpg 注意 虽然enqueueSuccessfulTask方法最终可以得到DirectTaskResult，但是DirectTask Result并不像它的字面意思一样代表Task的执行结果，对于ResultTask来说，DirectTaskResult的value的确是Task的执行结果，但对于ShuffleMapTask而言，实际是任务的状态。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 7.9.2 处理失败的Task

TaskResultGetter的enqueueFailedTask方法用于处理执行失败的Task的执行结果，其实现如代码清单7-95所示。

代码清单7-95 enqueueFailedTask方法

def enqueueFailedTask(taskSetManager: TaskSetManager, tid: Long, taskState: TaskState,  
 serializedData: ByteBuffer) {  
 var reason : TaskFailedReason = UnknownReason  
 try {  
 getTaskResultExecutor.execute(new Runnable {  
 override def run(): Unit = Utils.logUncaughtExceptions {  
 val loader = Utils.getContextOrSparkClassLoader  
 try {  
 if (serializedData != null && serializedData.limit() > 0) {  
 reason = serializer.get().deserialize[TaskFailedReason](  
 serializedData, loader) // 对执行结果反序列化，得到失败原因  
 }  
 } catch {  
 case cnd: ClassNotFoundException =>  
 logError(  
 "Could not deserialize TaskEndReason: ClassNotFound with classloader " + loader)  
 case ex: Exception => // No-op  
 }  
 scheduler.handleFailedTask(taskSetManager, tid, taskState, reason) // 处理失败的Task   
 }  
 })  
 } catch {  
 case e: RejectedExecutionException if sparkEnv.isStopped =>  
 // ignore it  
 }  
}

根据代码清单7-95，enqueueFailedTask方法实际向getTaskResultExecutor提交了一个获取Task执行结果的任务，这个任务的执行步骤如下。

1）对执行结果反序列化，得到类型为TaskFailedReason的失败原因。

2）调用TaskSchedulerImpl的handleFailedTask方法将失败的Task重新放入待处理的Task列表，并通知DAGScheduler重新调度。TaskSchedulerImpl的handleFailedTask方法的分析方式与handleSuccessfulTask方法类似，留给读者去分析。

## 7.10 任务调度器TaskScheduler

任务调度器TaskScheduler定义了对任务进行调度的接口规范，允许向Spark调度系统插入不同的TaskScheduler实现，但目前只有TaskSchedulerImpl这一个具体实现。TaskScheduler只为单个Driver调度任务。TaskSchedulerImpl的功能包括接收DAGScheduler给每个Stage创建的Task集合，按照调度算法将资源分配给Task，将Task交给Spark集群不同节点上的Executor运行，在这些Task执行失败时进行重试，通过推断执行减轻落后的Task对整体作业进度的影响。

Spark的资源调度分为两层：

第一层是Cluster Manager（在YARN模式下为Resource Manager，在Mesos模式下为Mesos Master，在Standalone模式下为Master）将资源分配给Application；

第二层是Application进一步将资源分配给Application的各个Task。TaskSchedulerImpl中的资源调度就是第二层的资源调度。

### 7.10.1 TaskSchedulerImpl的属性

TaskSchedulerImpl中有很多成员属性，理解这些属性的含义是深入TaskSchedulerImpl的前提。TaskSchedulerImpl的成员属性如下。

·sc：此属性持有SparkContext的引用。

·maxTaskFailures：任务失败的最大次数。

·isLocal：是否是Local部署模式。Local部署模式下的LocalSchedulerBackend已在7.8.2节介绍，更多部署模式的内容将在第9章中详细介绍。

·conf：即SparkConf。

·SPECULATION\_INTERVAL\_MS：任务推断执行的时间间隔。可以通过spark.speculation.interval属性进行配置，默认为100ms。

·MIN\_TIME\_TO\_SPECULATION：用于保证原始任务至少需要运行的时间。原始任务只有超过此时间限制，才允许启动副本任务。这可以避免原始任务执行太短的时间就被推断执行副本任务。MIN\_TIME\_TO\_SPECULATION的大小固定为100。

·speculationScheduler：对任务调度进行推断执行的ScheduledThreadPoolExecutor。由speculationScheduler创建的线程以task-scheduler-speculation为前缀。

·STARVATION\_TIMEOUT\_MS：判断TaskSet饥饿的阈值。可通过spark.starvation.timeout属性配置，默认为15s。

·CPUS\_PER\_TASK：每个Task需要分配的CPU核数。可通过spark.task.cpus属性配置，默认为1。

·taskSetsByStageIdAndAttempt：数据类型为HashMap[Int,HashMap[Int,TaskSetManager]]，是用于StageId、Attempt、TaskSetManager的二级缓存。

·taskIdToTaskSetManager：Task与所属TaskSetManager的映射关系。

·taskIdToExecutorId：Task与执行此Task的Executor之间的映射关系。

·hasReceivedTask：标记TaskSchedulerImpl是否已经接收到Task。

·hasLaunchedTask：标记TaskSchedulerImpl接收的Task是否已经有运行过的。

·starvationTimer：处理饥饿的定时器。

·nextTaskId：类型为AtomicLong，用于生成新提交Task的标识。

·executorIdToRunningTaskIds：类型为HashMap[String,HashSet[Long]]，用于缓存Executor与运行在此Executor上的任务之间的映射关系，由此看出一个Executor上可以运行多个Task。

·hostToExecutors：类型为HashMap[String,HashSet[String]]，用于缓存机器的Host与运行在此机器上的Executor之间的映射关系，由此可以看出机器与Executor之间是一对多的关系。

·hostsByRack：类型为HashMap[String,HashSet[String]]，用于缓存机器所在的机架与机架上机器的Host之间的映射关系，由此可以看出机架与机器之间是一对多的关系。

·executorIdToHost：Executor与Executor运行所在机器的Host之间的映射关系。

·dagScheduler：即DAGScheduler。

·backend：即调度后端接口SchedulerBackend。

·mapOutputTracker：即SparkEnv的子组件MapOutputTrackerMaster。

·schedulableBuilder：调度池构建器，即SchedulableBuilder。

·rootPool：根调度池，类型为Pool。

·schedulingModeConf：调度模式配置。可以通过spark.scheduler.mode属性配置，默认为FIFO。

·schedulingMode：调度模式。此属性依据schedulingModeConf获取枚举类型SchedulingMode的具体值。SchedulingMode共有FAIR、FIFO、NONE三种枚举值。

·taskResultGetter：类型为TaskResultGetter，它的作用是通过线程池（此线程池由Executors.newFixedThreadPool创建，大小默认为4，生成的线程名以task-result-getter开头），对Slave发送的Task的执行结果进行处理。

### 7.10.2 TaskSchedulerImpl的初始化

在4.5节介绍创建任务调度器的时候，在代码清单4-18中展示了针对不同master配置，分别以不同方式创建TaskSchedulerImpl的内容。无论何种方式创建TaskSchedulerImpl，要想让TaskSchedulerImpl发挥作用，必须都调用TaskSchedulerImpl的initialize方法（见代码清单7-96）对TaskSchedulerImpl进行初始化。

代码清单7-96 TaskSchedulerImpl的初始化

def initialize(backend: SchedulerBackend) {  
 this.backend = backend  
 rootPool = new Pool("", schedulingMode, 0, 0) // 创建根调度池  
 schedulableBuilder = { // 根据调度模式，创建相应的调度池构建器  
 schedulingMode match {  
 case SchedulingMode.FIFO =>  
 new FIFOSchedulableBuilder(rootPool)  
 case SchedulingMode.FAIR =>  
 new FairSchedulableBuilder(rootPool, conf)  
 case \_ =>  
 throw new IllegalArgumentException(s"Unsupported spark.scheduler.mode: $schedulingMode")  
 }  
 }  
 schedulableBuilder.buildPools() // 构建调度池  
}

根据代码清单7-96，initialize方法的执行步骤如下。

1）使用参数传递的SchedulerBackend设置TaskSchedulerImpl的backend属性。

2）创建根调度池。

3）根据调度模式，创建相应的调度池构建器。由于SchedulingMode默认为FIFO，所以创建的调度构建器默认为FIFOSchedulableBuilder。

4）调用调度池构建器的buildPools方法构建调度池。

### 7.10.3 TaskSchedulerImpl的启动

根据4.5节的内容，我们知道启动任务调度器是通过调用其start方法实现的。Task SchedulerImpl的start方法的实现如代码清单7-97所示。

代码清单7-97 TaskSchedulerImpl的启动

override def start() {  
 backend.start() // 启动SchedulerBackend  
 if (!isLocal && conf.getBoolean("spark.speculation", false)) { // 设置检查可推断任  
　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　// 务的定时器  
 logInfo("Starting speculative execution thread")  
 speculationScheduler.scheduleAtFixedRate(new Runnable {  
 override def run(): Unit = Utils.tryOrStopSparkContext(sc) {  
 checkSpeculatableTasks()  
 }  
 }, SPECULATION\_INTERVAL\_MS, SPECULATION\_INTERVAL\_MS, TimeUnit.MILLISECONDS)  
 }  
}

根据代码清单7-97，start方法的执行步骤如下。

1）调用SchedulerBackend的start方法启动SchedulerBackend。

2）当应用不是在Local模式下，并且设置了推断执行（即spark.speculation属性为true），那么设置一个执行间隔为SPECULATION\_INTERVAL\_MS（默认为100ms）的检查可推断任务的定时器。此定时器通过调用checkSpeculatableTasks方法（见代码清单7-98）来检查可推断任务。

代码清单7-98 检测可推断执行任务

def checkSpeculatableTasks() {  
 var shouldRevive = false  
 synchronized {  
 shouldRevive = rootPool.checkSpeculatableTasks(MIN\_TIME\_TO\_SPECULATION)  
 }  
 if (shouldRevive) {  
 backend.reviveOffers()  
 }  
}

根据代码清单7-98，checkSpeculatableTasks方法实际依赖于rootPool的checkSpeculatable Tasks方法（见代码清单7-52）。如果检查到有可以推断执行的任务，则调用Scheduler Backend的reviveOffers方法。以Local模式下的LocalSchedulerBackend为例，LocalScheduler Backend的reviveOffers方法（见代码清单7-92）将向LocalEndpoint发送ReviveOffers消息，LocalEndpoint接收到ReviveOffers消息后（见代码清单7-88），将调用LocalEndpoint的reviveOffers方法（见代码清单7-89）分配资源并运行Task。

### 7.10.4 TaskSchedulerImpl与Task的提交

DAGScheduler将Stage中各个分区的Task封装为TaskSet后，会将TaskSet交给Task SchedulerImpl处理。TaskSchedulerImpl的submitTasks方法是这一过程的入口，其实现如代码清单7-99所示。

代码清单7-99 提交Task

override def submitTasks(taskSet: TaskSet) {  
 val tasks = taskSet.tasks // 获取TaskSet中的所有Task  
 logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")  
 this.synchronized {  
 val manager = createTaskSetManager(taskSet, maxTaskFailures) // 创建TaskSetManager  
 val stage = taskSet.stageId  
 val stageTaskSets =  
 taskSetsByStageIdAndAttempt.getOrElseUpdate(stage, new HashMap[Int, TaskSet Manager])  
 stageTaskSets(taskSet.stageAttemptId) = manager  
 val conflictingTaskSet = stageTaskSets.exists { case (\_, ts) =>  
 ts.taskSet != taskSet && !ts.isZombie  
 }  
 if (conflictingTaskSet) {  
 throw new IllegalStateException(s"more than one active taskSet for stage $stage:" +  
 s" ${stageTaskSets.toSeq.map{\_.\_2.taskSet.id}.mkString(",")}")  
 }  
 schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)  
  
 if (!isLocal && !hasReceivedTask) { // 设置检查TaskSchedulerImpl的饥饿状况的定时器  
 starvationTimer.scheduleAtFixedRate(new TimerTask() {  
 override def run() {  
 if (!hasLaunchedTask) {  
 logWarning("Initial job has not accepted any resources; " +  
 "check your cluster UI to ensure that workers are registered " +  
 "and have sufficient resources")  
 } else {  
 this.cancel()  
 }  
 }  
 }, STARVATION\_TIMEOUT\_MS, STARVATION\_TIMEOUT\_MS)  
 }  
 hasReceivedTask = true // 表示TaskSchedulerImpl已经接收到Task  
 }  
 backend.reviveOffers() // 给Task分配资源并运行Task  
}

根据代码清单7-99，submitTasks方法的执行步骤如下。

1）获取TaskSet中的所有Task。

2）调用createTaskSetManager方法（见代码清单7-100）创建TaskSetManager。

3）在taskSetsByStageIdAndAttempt中设置TaskSet关联的Stage、Stage尝试及刚创建的TaskSetManager之间的三级映射关系。

4）对当前TaskSet进行冲突检测，即taskSetsByStageIdAndAttempt中不应该存在同属于当前Stage，但是TaskSet却不相同的情况。

5）调用调度池构建器的addTaskSetManager方法（见代码清单7-56或代码清单7-60），将刚创建的TaskSetManager添加到调度池构建器的调度池中。

6）如果当前应用程序不是Local模式并且TaskSchedulerImpl还没有接收到Task，那么设置一个定时器按照STARVATION\_TIMEOUT\_MS指定的时间间隔检查TaskScheduler Impl的饥饿状况，当TaskSchedulerImpl已经运行Task后，取消此定时器。

7）将hasReceivedTask设置为true，以表示TaskSchedulerImpl已经接收到Task。

8）调用SchedulerBackend的reviveOffers方法给Task分配资源并运行Task。

代码清单7-100 创建TaskSetManager

private[scheduler] def createTaskSetManager(  
 taskSet: TaskSet,  
 maxTaskFailures: Int): TaskSetManager = {  
 new TaskSetManager(this, taskSet, maxTaskFailures)  
}

### 7.10.5 TaskSchedulerImpl与资源分配

以Local模式下SchedulerBackend的实现LocalSchedulerBackend为例，LocalScheduler Backend的reviveOffers方法（见代码清单7-92）实际向LocalEndpoint发送了ReviveOffers消息。LocalEndpoint接收到ReviveOffers消息后，将调用LocalEndpoint自己的reviveOffers方法（见代码清单7-89）。根据之前对LocalEndpoint的reviveOffers方法的介绍，最终将调用TaskSchedulerImpl的resourceOffers方法（见代码清单7-101）给Task分配资源。

代码清单7-101 给Task分配资源

def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] = synchronized {  
 var newExecAvail = false  
 for (o <- offers) {  
 if (!hostToExecutors.contains(o.host)) {  
 hostToExecutors(o.host) = new HashSet[String]()  
 }  
 if (!executorIdToRunningTaskIds.contains(o.executorId)) { // 更新Host与Executor  
 // 的各种映射关系  
 hostToExecutors(o.host) += o.executorId  
 executorAdded(o.executorId, o.host)  
 executorIdToHost(o.executorId) = o.host  
 executorIdToRunningTaskIds(o.executorId) = HashSet[Long]()  
 newExecAvail = true // 标记添加了新的Executor  
 }  
 for (rack <- getRackForHost(o.host)) { // 更新Host与机架之间的关系  
 hostsByRack.getOrElseUpdate(rack, new HashSet[String]()) += o.host  
 }  
 }  
  
 val shuffledOffers = Random.shuffle(offers) // 随机洗牌，避免将任务总是分配给同样一组Worker  
 val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))  
 // 统计每个Worker的可用的CPU核数  
 val availableCpus = shuffledOffers.map(o => o.cores).toArray  
 // 所有TaskSetManager按照调度算法排序  
 val sortedTaskSets = rootPool.getSortedTaskSetQueue  
 for (taskSet <- sortedTaskSets) {  
 logDebug("parentName: %s, name: %s, runningTasks: %s".format(  
 taskSet.parent.name, taskSet.name, taskSet.runningTasks))  
 if (newExecAvail) {  
 taskSet.executorAdded() // 重新计算TaskSet的本地性  
 }  
 }  
  
 for (taskSet <- sortedTaskSets) {  
 var launchedAnyTask = false  
 var launchedTaskAtCurrentMaxLocality = false  
 for (currentMaxLocality <- taskSet.myLocalityLevels) { //按照最大本地性的原则,   
　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　 //给Task提供资源  
 do {  
 launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(  
 taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)  
 launchedAnyTask |= launchedTaskAtCurrentMaxLocality  
 } while (launchedTaskAtCurrentMaxLocality)  
 }  
 if (!launchedAnyTask) {  
 taskSet.abortIfCompletelyBlacklisted(hostToExecutors)  
 }  
 }  
  
 if (tasks.size > 0) {  
 hasLaunchedTask = true  
 }  
 return tasks // 返回已经获得了资源的任务列表  
}

根据代码清单7-101，resourceOffers方法的执行步骤如下。

1）遍历WorkerOffer序列，对每一个WorkerOffer执行以下操作。

①更新Host与Executor的各种映射关系。

②调用TaskSchedulerImpl的executorAdded方法（此方法实际仅仅调用了代码清单7-30所示的DAGScheduler的executorAdded方法）向DAGScheduler的DAGScheduler Event-ProcessLoop投递ExecutorAdded事件。

③标记添加了新的Executor（即将newExecAvail设置为true）。

④更新Host与机架之间的关系。

小贴士： 这里的hostToExecutors及hostsByRack是为了在资源分配时计算Task本地性时使用。

2）对所有WorkerOffer随机洗牌，避免将任务总是分配给同样一组Worker。

3）根据每个WorkerOffer的可用的CPU核数创建同等尺寸的任务描述（TaskDescription）数组。

4）将每个WorkerOffer的可用的CPU核数统计到可用CPU（availableCpus）数组中。

5）调用rootPool的getSortedTaskSetQueue方法（见代码清单7-53），对rootPool中的所有TaskSetManager按照调度算法排序。

6）如果newExecAvail为true，那么调用每个TaskSetManager的executorAdded方法。此executorAdded方法实际调用了computeValidLocalityLevels方法（见代码清单7-64）重新计算TaskSet的本地性。

7）遍历TaskSetManager，按照最大本地性的原则（即从高本地性级别到低本地性级别）调用resourceOfferSingleTaskSet方法（见代码清单7-102），给单个TaskSet中的Task提供资源。如果在任何TaskSet所允许的本地性级别下，TaskSet中没有任何一个任务获得了资源，那么将调用TaskSetManager的abortIfCompletelyBlacklisted方法（由于abortIfCompletely Blacklisted方法与调度流程的主逻辑相离较远，所以留给感兴趣的读者自行研究），放弃在黑名单中的Task。

8）返回生成的TaskDescription列表，即已经获得了资源的任务列表。

代码清单7-102 给单个TaskSet中的Task提供资源

private def resourceOfferSingleTaskSet(  
 taskSet: TaskSetManager,  
 maxLocality: TaskLocality,  
 shuffledOffers: Seq[WorkerOffer],  
 availableCpus: Array[Int],  
 tasks: IndexedSeq[ArrayBuffer[TaskDescription]]) : Boolean = {  
 var launchedTask = false  
 for (i <- 0 until shuffledOffers.size) {  
 val execId = shuffledOffers(i).executorId  
 val host = shuffledOffers(i).host  
 if (availableCpus(i) >= CPUS\_PER\_TASK) {  
 try { // 给符合条件的待处理Task创建TaskDescription  
 for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {  
 tasks(i) += task  
 val tid = task.taskId  
 taskIdToTaskSetManager(tid) = taskSet  
 taskIdToExecutorId(tid) = execId  
 executorIdToRunningTaskIds(execId).add(tid)  
 availableCpus(i) -= CPUS\_PER\_TASK  
 assert(availableCpus(i) >= 0)  
 launchedTask = true  
 }  
 } catch {  
 case e: TaskNotSerializableException =>  
 logError(s"Resource offer failed, task set ${taskSet.name} was not serializable")  
 return launchedTask  
 }  
 }  
 }  
 return launchedTask  
}

根据代码清单7-102，resourceOfferSingleTaskSet方法将遍历WorkerOffer序列，对每一个WorkerOffer执行以下操作。

1）获取WorkerOffer的Executor的身份标识。

2）获取WorkerOffer的Host。

3）如果WorkerOffer的可用的CPU核数大于等于CPUS\_PER\_TASK，则执行以下操作。

①调用TaskSetManager的resourceOffer方法（见代码清单7-73），给符合条件的待处理Task创建TaskDescription。

②将TaskDescription添加到tasks数组。

③更新Task的身份标识与TaskSet、Executor的身份标识相关的缓存映射。

④由于给Task分配了CPUS\_PER\_TASK指定数量的CPU内核数，因此WorkerOffer的可用的CPU核数减去CPUS\_PER\_TASK。

⑤返回launchedTask，即是否已经给TaskSet中的某个Task分配到了资源。

### 7.10.6 TaskSchedulerImpl的调度流程

本节通过对TaskSchedulerImpl的属性、初始化、启动、提交Task、给Task分配资源等内容的分析，想必读者对TaskSchedulerImpl的调度流程有了深入的理解，现在可以用图7-15来表示TaskSchedulerImpl的调度流程。

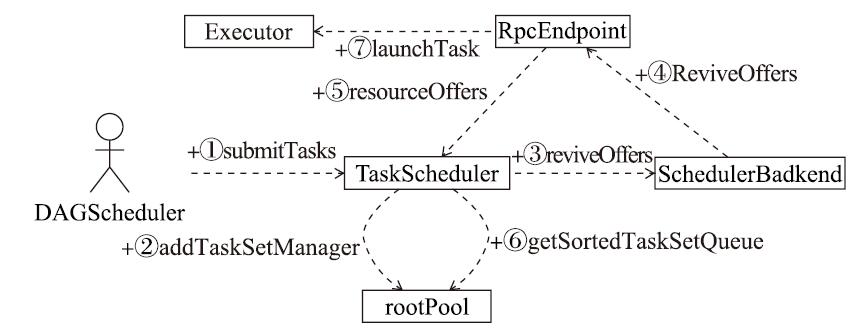


图7-15 TaskSchedulerImpl的调度流程

图7-15从抽象的角度描绘了TaskSchedulerImpl的调度流程，使用了SchedulerBackend，而不是SchedulerBackend的具体实现（如LocalSchedulerBackend），RpcEndpoint代表SchedulerBackend的具体实现中与其他组件进行通信的实例（如LocalSchedulerBackend中的LocalEndpoint）。这里对图7-15中的记号进行说明。

记号①： 代表DAGScheduler调用TaskScheduler的submitTasks方法向TaskScheduler提交TaskSet。

记号②： 代表TaskScheduler接收到TaskSet后，创建对此TaskSet进行管理的TaskSet Manager，并将此TaskSetManager通过调度池构建器添加到根调度池中。

记号③： 代表TaskScheduler调用SchedulerBackend的reviveOffers方法给Task提供资源。

记号④： SchedulerBackend向RpcEndpoint发送ReviveOffers消息。

记号⑤： RpcEndpoint将调用TaskScheduler的resourceOffers方法给Task提供资源。

记号⑥： TaskScheduler调用根调度池的getSortedTaskSetQueue方法对所有TaskSet Manager按照调度算法进行排序后，对TaskSetManager管理的TaskSet按照“最大本地性”的原则选择其中的Task，最后为Task创建尝试执行信息、对Task进行序列化、生成TaskDescription等。

记号⑦： 调用Executor的launchTask方法运行Task尝试。

### 7.10.7 TaskSchedulerImpl对执行结果的处理

Task在执行的时候会不断发送StatusUpdate消息，在Local模式下，LocalEndpoint接收到StatusUpdate消息后（见代码清单7-88）会先匹配执行TaskSchedulerImpl的statusUpdate方法，然后调用reviveOffers方法（见代码清单7-89）给其他Task分配资源。

TaskSchedulerImpl的statusUpdate方法（代码清单7-103）用于更新Task的状态。Task的状态包括：运行中（RUNNING）、已完成（FINISHED）、失败（FAILED）、被杀死（KILLED）、丢失（LOST）五种。会从taskIdToTaskSetId、taskIdToExecutorId中移除此任务，并且调用taskResultGetter的enqueueSuccessfulTask方法。

代码清单7-103 更新Task的状态

def statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer) {  
 var failedExecutor: Option[String] = None  
 var reason: Option[ExecutorLossReason] = None  
 synchronized {  
 try {  
 taskIdToTaskSetManager.get(tid) match {  
 case Some(taskSet) =>  
 if (state == TaskState.LOST) { // 从taskIdToExecutorId中获取Task对应的Executor的身份标识  
 val execId = taskIdToExecutorId.getOrElse(tid, throw new IllegalState Exception(  
 "taskIdToTaskSetManager.contains(tid) <=> taskIdToExecutorId.contains(tid)"))  
 if (executorIdToRunningTaskIds.contains(execId)) {  
 reason = Some(  
 SlaveLost(s"Task $tid was lost, so marking the executor as lost as well."))  
 removeExecutor(execId, reason.get) // 移除Executor，移除的原因是SlaveLost  
 failedExecutor = Some(execId)  
 }  
 }  
 if (TaskState.isFinished(state)) {  
 cleanupTaskState(tid) // 清除Task在taskIdToTaskSetManager、taskIdTo ExecutorId中的数据  
 taskSet.removeRunningTask(tid) // 减少正在运行的任务数量  
 if (state == TaskState.FINISHED) { // 对执行成功的任务的结果进行处理  
 taskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)  
 } else if (Set(TaskState.FAILED, TaskState.KILLED, TaskState.LOST).contains(state)) {  
 // 对执行失败的Task的结果进行处理  
 taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)  
 }  
 }  
 case None =>  
 logError(  
 ("Ignoring update with state %s for TID %s because its task set is gone (this is " +  
 "likely the result of receiving duplicate task finished status updates) or its " +  
 "executor has been marked as failed.")  
 .format(state, tid))  
 }  
 } catch {  
 case e: Exception => logError("Exception in statusUpdate", e)  
 }  
 }  
 // 妥善安置丢失的Executor上正在运行的Task  
 if (failedExecutor.isDefined) {  
 assert(reason.isDefined)  
 dagScheduler.executorLost(failedExecutor.get, reason.get)  
 backend.reviveOffers()  
 }  
}

根据代码清单7-103，statusUpdate方法的执行步骤如下。

1）从taskIdToTaskSetManager中获取Task对应的TaskSetManager。

2）如果要更新的任务状态是LOST，那么从taskIdToExecutorId中获取Task对应的Executor的身份标识。如果此Executor上正在运行Task，那么调用removeExecutor方法移除Executor，移除的原因是SlaveLost。

3）如果要更新的任务状态是完成状态（包括FINISHED、FAILED、KILLED、LOST），那么首先调用cleanupTaskState方法（见代码清单7-104），清除此Task在taskIdToTaskSet Manager、taskIdToExecutorId等缓存中的数据，然后调用TaskSetManager的remove RunningTask方法（见代码清单7-71），减少正在运行的任务数量。如果任务状态是FINISHED，则调用TaskResultGetter的enqueueSuccessfulTask方法（见代码清单7-94），对执行成功的任务的结果进行处理。如果任务状态是FAILED、KILLED、LOST三者中的一种，那么调用TaskResultGetter的enqueueFailedTask方法（见代码清单7-95），对执行失败的Task的结果进行处理。

4）如果failedExecutor设置了Executor的身份标识，说明此Executor已经被移除，那么此Executor上正在运行的Task需要得到妥善的安置。安置的办法是：首先调用DagScheduler的executorLost方法（此方法非常简单，只是向DAGSchedulerEvent ProcessLoop投递了ExecutorLost消息，根据代码清单7-22的内容，DAGSchedulerEvent ProcessLoop处理ExecutorLost消息时，将调用DagScheduler的handleExecutorLost方法对丢失的Executor作进一步处理，限于篇幅，handleExecutorLost方法的实现留给感兴趣的读者自行研究），然后调用SchedulerBackend的reviveOffers方法给Task分配资源并运行Task。

代码清单7-104 清除Task的状态

private def cleanupTaskState(tid: Long): Unit = {  
 taskIdToTaskSetManager.remove(tid)  
 taskIdToExecutorId.remove(tid).foreach { executorId =>  
 executorIdToRunningTaskIds.get(executorId).foreach { \_.remove(tid) }  
 }  
}

上面专门分析了TaskSchedulerImpl的statusUpdate方法的实现。现在我们对status Update方法的第3）步进行聚焦，即专门来分析任务执行成功后的结果处理。根据7.9.1节对TaskResultGetter的enqueueSuccessfulTask方法的介绍，我们知道在获取到Task的执行结果后，将调用TaskSchedulerImpl的handleSuccessfulTask方法。TaskSchedulerImpl的handleSuccessfulTask方法的实现如代码清单7-105所示。

代码清单7-105 TaskSchedulerImpl的handleSuccessfulTask方法

def handleSuccessfulTask(  
 taskSetManager: TaskSetManager,  
 tid: Long,  
 taskResult: DirectTaskResult[\_]): Unit = synchronized {  
 taskSetManager.handleSuccessfulTask(tid, taskResult)  
}

根据代码清单7-105，handleSuccessfulTask方法实际调用了TaskSetManager的handle SuccessfulTask方法（见代码清单7-77），这样就把Task执行结果的处理整个串联了起来，如图7-16所示。

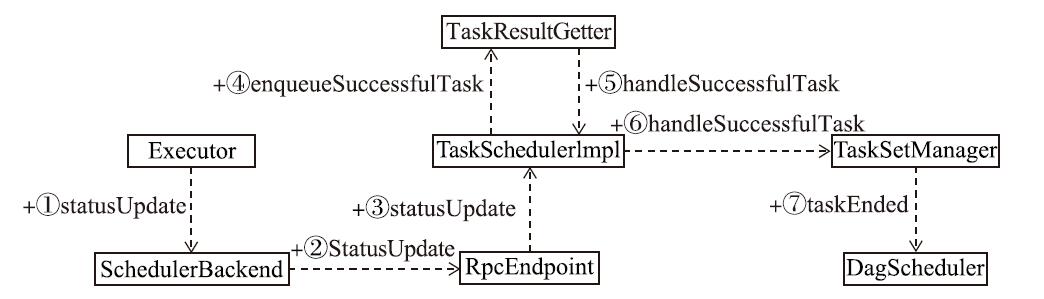


图7-16 Task执行结果的处理流程

这里对图7-16中的记号进行说明。

记号①： Executor中的TaskRunner在执行Task的过程中，不断将Task的状态通过调用SchedulerBackend的实现类（比如Local模式下的LocalSchedulerBackend或Standalone模式下的StandaloneSchedulerBackend）的statusUpdate方法告诉SchedulerBackend的实现类。当Task执行成功后，TaskRunner也会将Task的完成状态告诉SchedulerBackend的实现类。

记号②： 代表SchedulerBackend的实现类将Task的完成状态封装为StatusUpdate消息发送给RpcEndpoint的实现类（比如Local模式下的LocalEndpoint或Standalone模式下的DriverEndpoint）。

记号③： RpcEndpoint的实现类接收到StatusUpdate消息后，将调用TaskSchedulerImpl的statusUpdate方法。

记号④： TaskSchedulerImpl的statusUpdate方法发现Task是执行成功的状态，那么调用TaskResultGetter的enqueueSuccessfulTask方法。

记号⑤： TaskResultGetter的enqueueSuccessfulTask方法对DirectTaskResult类型的结果进行反序列化得到Task执行结果，对于IndirectTaskResult类型的结果需要先从远端下载Block数据，然后再进行反序列化得到Task执行结果。TaskResultGetter获取到Task执行结果后，调用TaskSchedulerImpl的handleSuccessfulTask方法交给TaskSchedulerImpl处理。

记号⑥： TaskSchedulerImpl的handleSuccessfulTask方法将直接调用TaskSetManager的handleSuccessfulTask方法。

记号⑦： TaskSetManager的handleSuccessfulTask方法最重要的一步是调用DAGScheduler的taskEnded方法。对于ResultTask的结果，DAGScheduler的taskEnded方法会将它交给JobWaiter的resultHandler函数来处理。对于ShuffleMapTask的结果，DAGScheduler的taskEnded方法则将Task的partitionId和MapStatus追加到Stage的outputLocs中。如果没有待计算的分区，则需要将Stage的shuffleId和outputLocs中的MapStatus注册到Map OutputTrackerMaster的mapStatuses中。如果有某些分区的Task执行失败，则重新提交Shuffle MapStage，否则调用submitWaitingChildStages方法提交当前ShuffleMapStage的子Stage。

### 7.10.8 TaskSchedulerImpl的常用方法

本节还将介绍一些TaskSchedulerImpl提供的常用方法。

1.removeExecutor

removeExecutor方法（见代码清单7-106）用于移除Executor及其对应的Host和机架等缓存信息。

代码清单7-106 移除Executor

private def removeExecutor(executorId: String, reason: ExecutorLossReason) {  
 executorIdToRunningTaskIds.remove(executorId).foreach { taskIds =>  
 logDebug("Cleaning up TaskScheduler state for tasks " +  
 s"${taskIds.mkString("[", ",", "]")} on failed executor $executorId")  
 taskIds.foreach(cleanupTaskState)  
 }  
  
 val host = executorIdToHost(executorId)  
 val execs = hostToExecutors.getOrElse(host, new HashSet)  
 execs -= executorId  
 if (execs.isEmpty) {  
 hostToExecutors -= host  
 for (rack <- getRackForHost(host); hosts <- hostsByRack.get(rack)) {  
 hosts -= host  
 if (hosts.isEmpty) {  
 hostsByRack -= rack  
 }  
 }  
 }  
  
 if (reason != LossReasonPending) {  
 executorIdToHost -= executorId  
 rootPool.executorLost(executorId, host, reason)  
 }  
}

根据代码清单7-106，removeExecutor方法接收两个参数：Executor的身份标识和移除Executor的原因（即ExecutorLossReason）。ExecutorLossReason有四个子类，分别代表四种不同的原因。

·SlaveLost：Worker丢失。

·LossReasonPending：未知的原因导致的Executor退出。

·ExecutorKilled：Executor被“杀死”了。

·ExecutorExited：Executor退出了。

有了对ExecutorLossReason的了解，现在来看看removeExecutor方法的执行步骤。

1）从executorIdToRunningTaskIds移除Executor的缓存，并调用cleanupTaskState方法（见代码清单7-104），清除在此Executor上正在运行的Task在taskIdToTaskSetManager、taskIdToExecutorId等缓存中的数据。

2）从hostToExecutors中移除此Executor的信息。

3）如果hostToExecutors中此Executor所在的Host主机已经没有任何Executor了，那么从hostsByRack中移除此Host的信息。

4）如果hostsByRack中此Executor所在的机架已经没有任何Host了，那么从hosts ByRack中移除此机架的信息。

5）如果移除Executor的原因不是LossReasonPending，那么首先从executorIdToHost中移除此Executor的缓存，然后调用根调度池rootPool的executorLost方法（见代码清单7-51），将在此Executor上正在运行的Task作为失败任务处理，最后重新提交这些任务。

2.executorLost

executorLost方法（见代码清单7-107）用于处理Executor丢失。

代码清单7-107 处理Executor丢失

override def executorLost(executorId: String, reason: ExecutorLossReason): Unit = {  
 var failedExecutor: Option[String] = None  
  
 synchronized {  
 if (executorIdToRunningTaskIds.contains(executorId)) {  
 val hostPort = executorIdToHost(executorId)  
 logExecutorLoss(executorId, hostPort, reason)  
 removeExecutor(executorId, reason) // 移除Executor  
 failedExecutor = Some(executorId) // 将failedExecutor设置为此Executor  
 } else {  
 executorIdToHost.get(executorId) match {  
 case Some(hostPort) =>  
 logExecutorLoss(executorId, hostPort, reason)  
 removeExecutor(executorId, reason) // 移除Executor  
  
 case None =>  
 logError(s"Lost an executor $executorId (already removed): $reason")  
 }  
 }  
 }  
 if (failedExecutor.isDefined) {  
 dagScheduler.executorLost(failedExecutor.get, reason)  
 backend.reviveOffers() // 给Task分配资源并运行Task  
 }  
}

根据代码清单7-107，executorLost方法的执行步骤如下。

1）如果executorIdToRunningTaskIds中包含指定的Executor的身份标识，这说明此时在此Executor上已经有Task正在运行，那么调用removeExecutor方法移除Executor，并将failedExecutor设置为此Executor。

2）如果executorIdToRunningTaskIds中不包含指定的Executor的身份标识，这说明此时在此Executor上没有Task正在运行，那么从executorIdToHost中获取此Executor对应的Host，并调用removeExecutor方法移除Executor。

3）如果failedExecutor设置了Executor的身份标识，这说明此Executor已经被移除，那么此Executor上正在运行的Task需要得到妥善的安置。安置的办法是：首先调用DagScheduler的executorLost方法（此方法非常简单，只是向DAGSchedulerEvent ProcessLoop投递了ExecutorLost消息，根据代码清单7-22的内容，DAGSchedulerEvent ProcessLoop处理ExecutorLost消息时，将调用DagScheduler的handleExecutorLost方法对丢失的Executor作进一步处理，限于篇幅，handleExecutorLost方法的实现留给感兴趣的读者自行研究），然后调用SchedulerBackend的reviveOffers方法给Task分配资源并运行Task。

3.postStartHook

由于TaskSchedulerImpl对任务资源的运行依赖于SchedulerBackend，所以为了避免Task SchedulerImpl在SchedulerBackend准备就绪之前，就将Task交给SchedulerBackend处理，因此实现了postStartHook方法用于等待SchedulerBackend准备就绪，其实现如代码清单7-108所示。

代码清单7-108 postStartHook方法的实现

override def postStartHook() {  
 waitBackendReady()  
}  
private def waitBackendReady(): Unit = {  
 if (backend.isReady) {  
 return  
 }  
 while (!backend.isReady) {  
 if (sc.stopped.get) {  
 throw new IllegalStateException("Spark context stopped while waiting for backend")  
 }  
 synchronized {  
 this.wait(100)  
 }  
 }  
}

## 7.11 小结

由于RDD是Spark调度系统的原材料，因此本章首先对RDD与DAG调度有关的功能进行了详细介绍。

通过对DAGScheduler处理RDD构成的DAG、将DAG中的RDD划分为多个Stage、提交ResultStage、提交未计算的Task等功能的分析，DAGScheduler的整个工作流程浮现在读者面前。OutputCommitCoordinator是DAGScheduler中的重要组件，其实现方式简洁明了，有很高的借鉴意义。

TaskSchedulerImpl依赖于LauncherBackend和SchedulerBackend。通过对TaskScheduler Impl的初始化、启动、提交Task、资源分配等功能的分析，读者可以更加深入地了解Task SchedulerImpl的调度流程。此外，通过理解调度算法、调度池、Task本地性、推断执行等内容，将对Spark性能优化、二次开发带来收益。