

第7章

Chapter 7

调度系统

“诸君听吾计，使渤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、孔侑、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)。

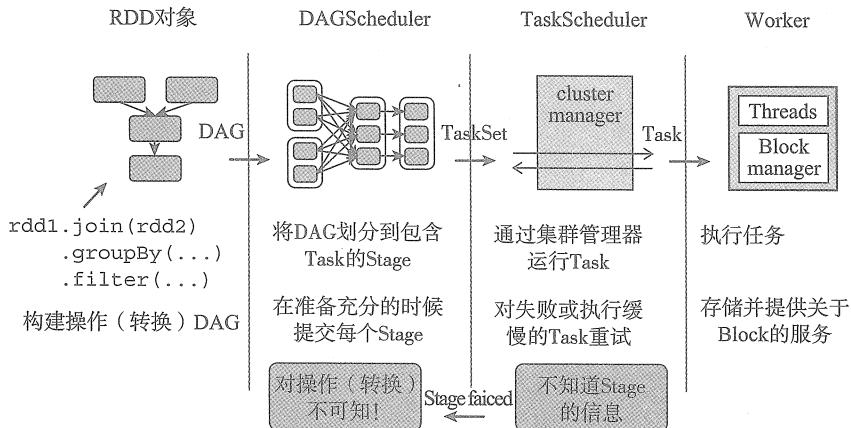
7.1 调度系统概述

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) 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。



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。

[⊖] 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 getPreferredLocations 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 getPreferredLocations 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。

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。

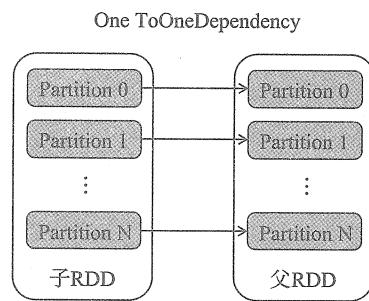


图 7-2 OneToOneDependency 的依赖示意图

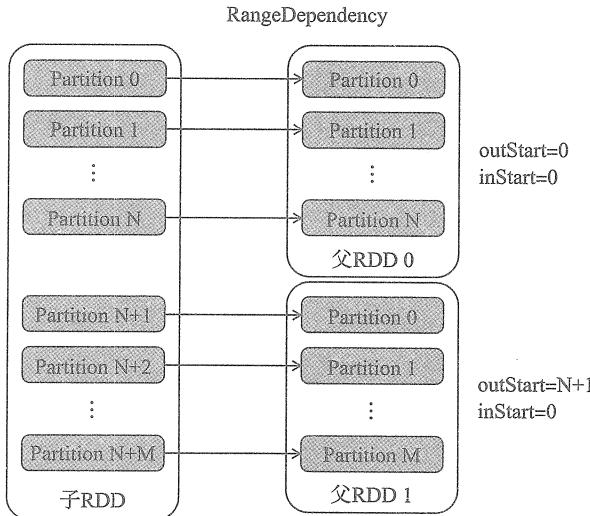


图 7-3 RangeDependency 的依赖示意图

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。

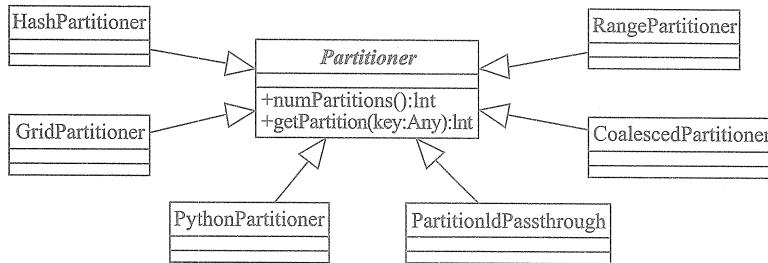


图 7-4 Partitioner 的继承体系

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，已经能够达到本书的目的。

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。

ShuffleDependency

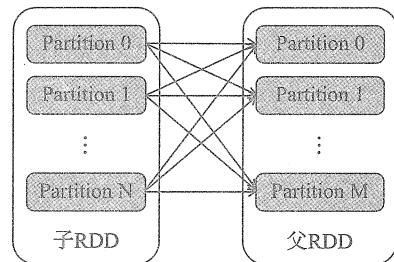


图 7-5 ShuffleDependency 的依赖示意图

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。

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 ShuffleMapTask，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。其实现如代码清单 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。DAGSchedulerEventProcessLoop 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 getTimeMillis 方法和使得当前线程睡眠直到指定时间为为止的 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 `DAGScheduler` 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]): Unit = {
    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
        }
      }
      cacheLocs += (rdd.id, locs)
    }
  }
}
```

```

        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 DAGScheduler-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 ExecutorAdded 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 DAGSchedulerEventProcessLoop 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.currentTimeMillis() // 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 =>
            }
        }
    }
    parents
}
```

```

        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 ShuffleMapStage 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。

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 submitStage 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 waitingStages、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
                            }
                    }
                }
            }
        }
    }
    waitingForVisit.push(stage)
    while (!waitingForVisit.isEmpty) {
        val rdd = waitingForVisit.pop()
        visit(rdd)
    }
    missing.toList
}
```

```

        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 getCacheLocs 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))
                }
        }
    } catch {
        case e: Exception =>
            log.error(s"Error while getting preferred locations for partitions $partitionsToCompute: $e")
            throw new SparkException("Error while getting preferred locations for partitions " +
                partitionsToCompute.mkString(", ") + " in stage " + stage.id + ". Error: " + e.getMessage)
    }
}

```

```

        } .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.currentTimeMillis())
} 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 partitionsToCompute 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 makeNewStageAttempt 方法开始一次新的 Stage 执行尝试，然后向 listenerBus 投递 SparkListenerStageSubmitted 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 ${errorMessage.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。

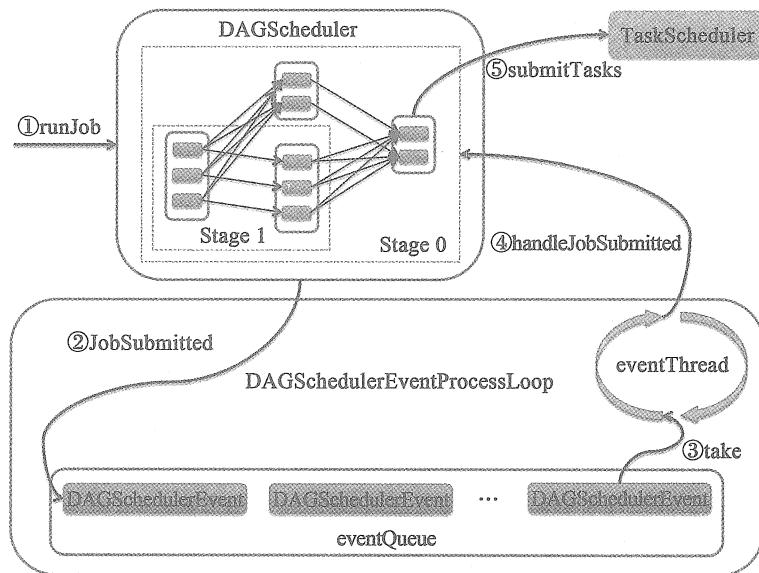


图 7-6 DAGScheduler 的调度流程

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 MapOutputTrackerMaster 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 CompletionEvent 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.currentTimeMillis(), JobSucceeded))
              }
            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 SparkListenerJobEnd 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 markStageAsFinished 方法（见代码清单 7-43），将 ShuffleMapStage 标记为完成，然后调用 MapOutputTrackerMaster 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 numPartitions 相等），那么说明所有任务执行成功了，这时调用 MapOutputTracker 的 getStatistics 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 ShuffleMapStage 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。

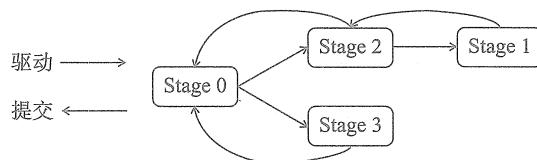


图 7-7 Stage 的正向驱动与反向提交

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 SchedulingAlgorithm 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 FIFO SchedulingAlgorithm 和公平调度算法的 Fair SchedulingAlgorithm，下面将对这两种算法实现进行介绍。

1. FIFO SchedulingAlgorithm 详解

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

代码清单 7-46 FIFO SchedulingAlgorithm 的实现

```
private[spark] class FIFO SchedulingAlgorithm 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，FIFO SchedulingAlgorithm 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. Fair SchedulingAlgorithm 详解

Fair SchedulingAlgorithm 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 taskToWeightRatio2 进行比较。如果 taskToWeightRatio1 小于 taskToWeightRatio2, 则优先调度

s_1 ；如果 $\text{taskToWeightRatio}_2$ 小于 $\text{taskToWeightRatio}_1$ ，则优先调度 s_2 。如果 $\text{taskToWeightRatio}_1$ 和 $\text{taskToWeightRatio}_2$ 相等，还需要对 s_1 和 s_2 的名字进行比较。如果 s_1 的名字小于 s_2 的名字，则优先调度 s_1 ，否则优先调度 s_2 。 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 schedulableQueue 是一个可以嵌套的层次结构，如图 7-8 所示。

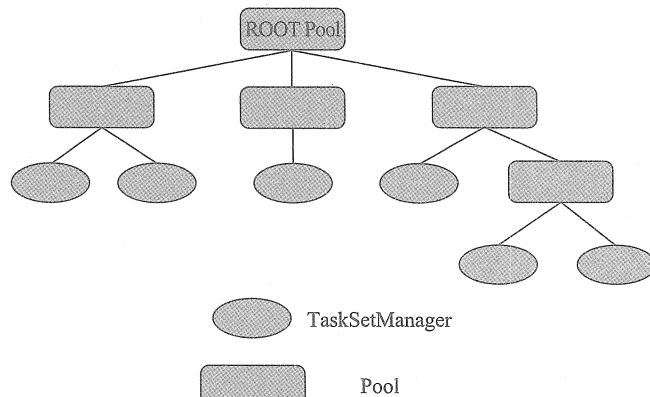


图 7-8 调度队列的层次关系

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。

- schedulableNameToScheduled：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 FIFO Scheduling Algorithm。了解了 Pool 的属性，现在来看看 Pool 中提供的方法。

1. addScheduledable

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

代码清单7-48 添加Schedulable

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

2. removeScheduledable

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

代码清单7-49 移除Schedulable

```
override def removeScheduledable(schedulable: Schedulable) {
    schedulableQueue.remove(schedulable)
    schedulableNameToScheduled.remove(schedulable.name)
}
```

3. getScheduledableByName

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

```

        if (sched != null) {
            return sched
        }
    }
null
}

```

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

- 1) 从当前 Pool 的 schedulableNameToScheduled 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 FIFOschedulableBuilder 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 FIFOschedulableBuilder 构建出的调度池的内存结构。

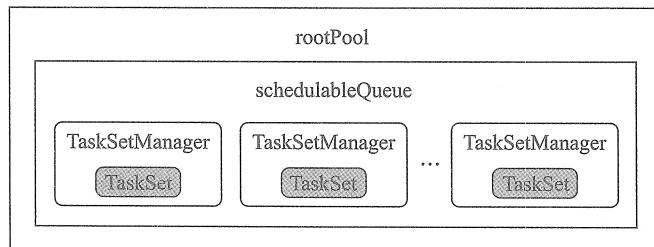


图 7-9 FIFOschedulableBuilder 构建的调度池的内存结构

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。FairSchedulableBuilder 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。

① FAIR_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。

② DEFAULT_POOL_NAME：默认的调度池名。常量 DEFAULT_POOL_NAME 的值固定为 "default"。

③ MINIMUM_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。

④ SCHEDULING_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。

⑤ WEIGHT_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。

⑥ POOL_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。

⑦ POOLS_PROPERTY：调度池属性。常量 POOLS_PROPERTY 的值固定为 "pool"。

⑧ DEFAULT_SCHEDULING_MODE：默认的调度模式 FIFO。

⑨ DEFAULT_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。

⑩ DEFAULT_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 schedulerAllocFile，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 (schedulingMode) 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.getSchedulerByName(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.getSchedulerByName(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.getSchedulerByName(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 FairSchedulableBuilder 构建出的调度池的内存结构。

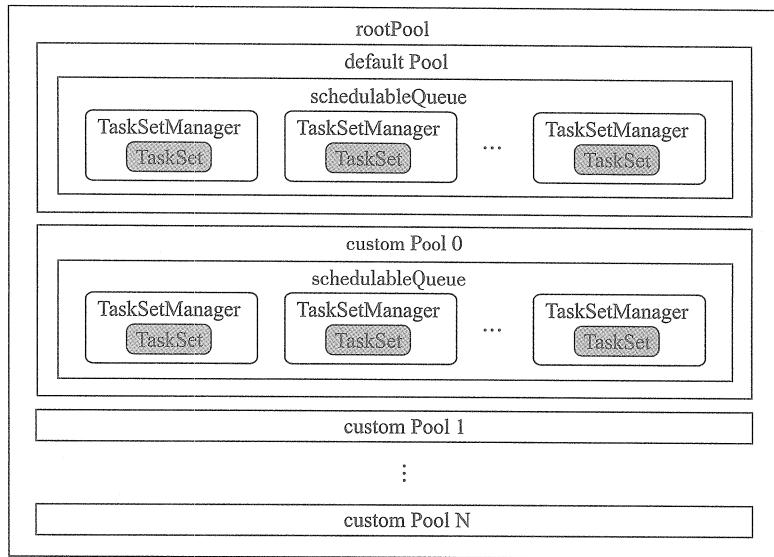


图 7-10 FairSchedulableBuilder 构建的调度池的内存结构

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。TaskSchedulerImpl 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 runningTasksSet 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。TaskSetManager 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。MyLocalityLevels 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。

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 checkSpeculatableTasks 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) 计算进行推断的最长时间（threshold），即 SPECULATION_MULTIPLIER 与 medianDuration 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 speculatableTasks 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 speculatableTasks 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 pendingTasksForExecutor 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 pendingTasksForHost 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 pendingTasksForRack 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 {
        OL
    }
}

```

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。

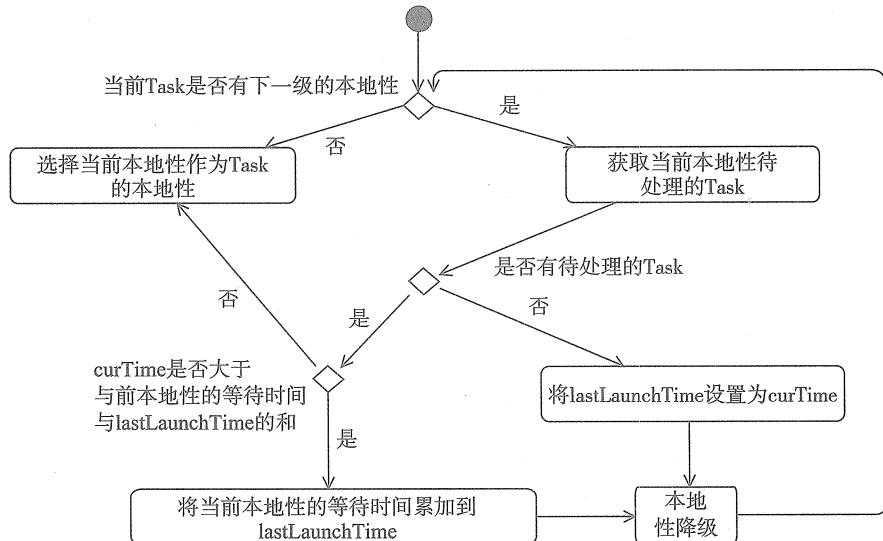


图 7-11 获取允许的本地性级别

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`、`pendingTasksWithNoPrefs`、`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(exe
        cId))) {
        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 removeRunningTask 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。
- ⑪ 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。
- ⑫ 生成 Task 的名称, 格式为: "task \$index.\$attemptNumber in stage \$stageId. \$stage AttemptId"。
- ⑬ 调用 DAGScheduler 的 taskStarted 方法向 DAGSchedulerEventProcessLoop 投递 BeginEvent 事件。
- ⑭ 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: ExecutorLossReason) {
    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 pendingTasksForExecutor、pendingTasksForHost、pendingTasksForRack、pendingTasksWithNoPrefs、allPendingTasks 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.isEmpty
}
```

```

    }
    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。

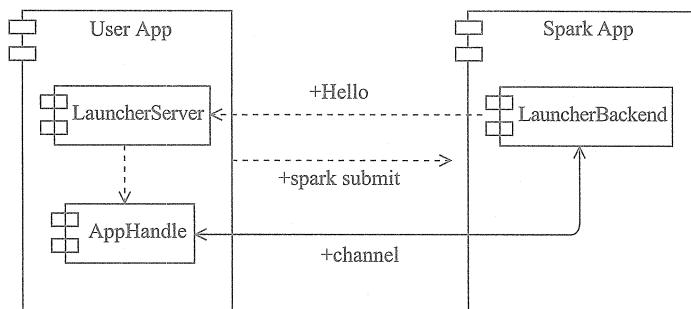


图 7-12 LauncherServer 的工作原理

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。

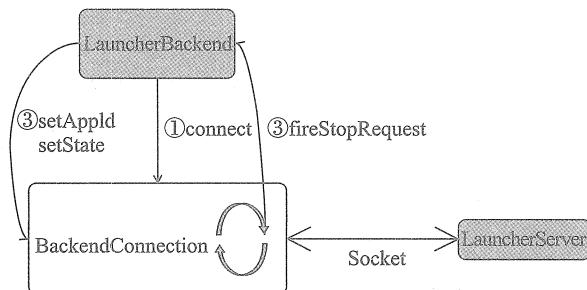


图 7-13 LauncherBackend 的主要工作流程

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;
```

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 LauncherConnection 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 SparkAppHandle.State。SparkAppHandle.State 共有未知(UNKNOWN)、已连接(CONNECTED)、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。



注意 SparkContext 的 _applicationId 属性和 _applicationAttemptId 属性是通过分别调用 TaskSchedulerImpl 的 applicationId 方法和 applicationAttemptId 方法获得的，而 TaskSchedulerImpl 的 applicationId 方法和 applicationAttemptId 方法实际又分别调用了 SchedulerBackend 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 SchedulerBackend 的实现类，如图 7-14 所示。

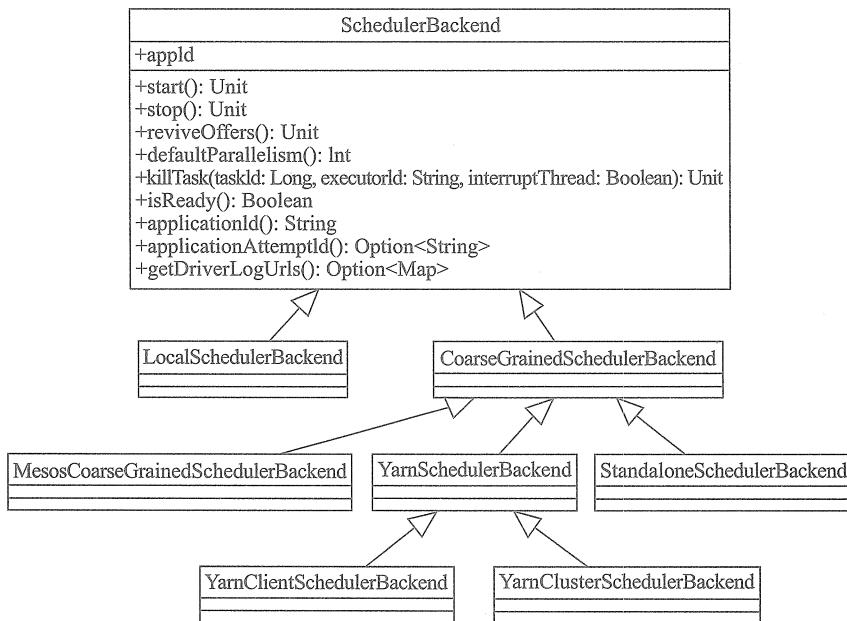


图 7-14 SchedulerBackend 的继承体系

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 `CoarseGrainedExecutorBackend` 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 receiveAndReply 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。

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。

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

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[_]](serializedData)
        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) //忽略异常处理代码.
    } 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)。

注意 虽然 enqueueSuccessfulTask 方法最终可以得到 DirectTaskResult，但是 DirectTaskResult 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。

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] (br/>                            serializedData, loader) // 对执行结果反序列化，得到失败原因  
                    }  
                } catch {  
                    ...  
                }  
            }  
        })  
    } 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 checkSpeculatableTasks 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 TaskSchedulerImpl 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，LocalSchedulerBackend 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 abortIfCompletelyBlacklisted 方法与调度流程的主要逻辑相离较远, 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。

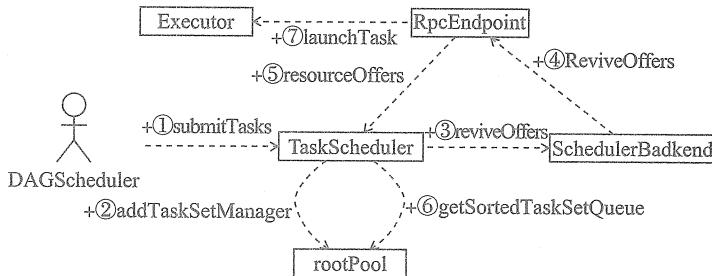


图 7-15 TaskSchedulerImpl 的调度流程

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 IllegalStateException(
                            "taskToTaskSetManager.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 statusUpdate 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 handleSuccessfulTask 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。

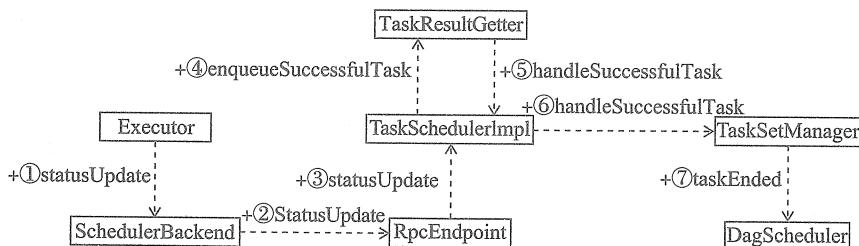


图 7-16 Task 执行结果的处理流程

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 MapOutputTrackerMaster 的 mapStatuses 中。如果有某些分区的 Task 执行失败，则重新提交 ShuffleMapStage，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 hostsByRack 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 TaskSchedulerImpl 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 TaskSchedulerImpl 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。