

## Chapter 9 第9章

# 部署模式

“事在四方，要在中央。圣人执要，四方来效。”

——《韩非子·物权》

## 本章导读

韩非子认为：“事情发生在天下四方，而要害却在中央”，所以他主张：“中央集权，决策权在圣人或皇帝手中，这样就可以掌管好整个国家、整个天下”。这种主张顺应了大一统的历史潮流，也迎合了秦始皇及后来封建帝王的需要。

Spark 集群也是一个“天下”，在每个节点上时时刻刻都在发生着改变，为了对整个集群能够进行管理，也需要一个统领全局的“人”。集群管理器就是 Spark 集群中的“圣人”。针对不同的集群管理器，Spark 提供了不同的部署模式对整个集群进行协调和管理。

之前的章节为了突出 Spark 各个组件的原理，凡是涉及部署模式的地方都以本地部署模式为主。无论哪种部署模式，大多数代码逻辑是通用的。但是各种部署模式在原理上存在诸多差异，例如，高可用性、容错与故障恢复、可扩展性等，本章将对这些差异进行分析。

1) Spark 目前支持的部署模式如下。

- 本地部署模式：local、local[N]、local[\*]、local[N, maxRetries]。此类模式主要用于代码调试和跟踪，不具备高可用性、可扩展性，所以不适用于生产环境。
- 本地集群部署模式：local-cluster[N, cores, memory]。此模式也用于代码调试和测试，是源码学习常用的模式，不具备高可用性、可扩展性，不适用于生产环境。

- Standalone 部署模式：spark://host:port。此模式具备高可用性并且支持分布式部署，所以可用于实际的生产环境。
  - 第三方部署模式：YARN client、YARN cluster、mesos://、zk://、simr:// 等。
- 2) 部署模式涉及很多角色。
- Application：用户使用 Spark API 开发的应用程序。
  - Driver：应用驱动程序。有了 Driver，Application 才会被提交到 Spark 集群运行。Driver 可以选择在客户端运行，也可以选择向 Master 注册，然后由 Master 命令 Worker 启动 Driver。
  - Master：Spark 的主控节点。在实际的生产环境中会有多个 Master，只有一个 Master 处于 active 状态，其余的 Master 处于 slave 状态。
  - Worker：Spark 的工作节点，向 Master 汇报自身的资源、Executor 执行状态的改变，并接受 Master 的命令启动 Executor 或 Driver。
  - Executor：Spark 的工作进程，由 Worker 监管，负责具体任务的执行。

## 9.1 心跳接收器 HeartbeatReceiver

在 4.4 节曾经介绍过 HeartbeatReceiver 的创建和注册，并对 HeartbeatReceiver 的作用进行了简单介绍。由于 HeartbeatReceiver 与部署模式息息相关，因此本节将深入分析 HeartbeatReceiver 的实现。

HeartbeatReceiver 运行在 Driver 上，用以接收各个 Executor 的心跳（HeartBeat）消息，对各个 Executor 的“生死”进行监控。HeartbeatReceiver 包含以下属性。

- sc：即 SparkContext。
- clock：类型为 org.apache.spark.util.Clock，Clock 实际是对 System.currentTimeMillis() 的一层封装，以便于获取系统时间。
- rpcEnv：即 SparkEnv 的子组件 RpcEnv。
- scheduler：即 TaskSchedulerImpl。
- executorLastSeen：用于维护 Executor 的身份标识与 HeartbeatReceiver 最后一次收到 Executor 的心跳（HeartBeat）消息的时间戳之间的映射关系。
- slaveTimeoutMs：Executor 节点上的 BlockManager 的超时时间（单位为 ms）。可通过 spark.storage.blockManagerSlaveTimeoutMs 属性配置，默认为 120000。
- executorTimeoutMs：Executor 的超时时间（单位为 ms）。可通过 spark.network.timeout 属性配置，默认为 120000。
- timeoutIntervalMs：超时的间隔（单位为 ms）。可通过 spark.storage.blockManagerTimeoutIntervalMs 属性配置，默认为 60000。
- checkTimeoutIntervalMs：检查超时的间隔（单位为 ms）。可通过 spark.network.time-

outInterval 属性配置， 默认采用 timeoutIntervalMs 的值。

**小贴士：**spark.network.timeoutInterval 属性在配置时以 s 为单位，HeartbeatReceiver 会将秒转换为 ms。

- eventLoopThread：类型为 ScheduledThreadPoolExecutor，用于执行心跳接收器的超时检查任务，eventLoopThread 只包含一个线程，此线程以 heartbeat-receiver-event-loop-thread 作为名称。
- timeoutCheckingTask：向 eventLoopThread 提交执行超时检查的定时任务后返回的 ScheduledFuture。提交的定时任务的执行间隔为 checkTimeoutIntervalMs。
- killExecutorThread：以 Executors.newSingleThreadExecutor 方式创建的 ExecutorService，运行的单线程用于“杀死”（kill）Executor，此线程以 kill-executor-thread 作为名称。

了解了 HeartbeatReceiver 中的属性，现在来看看 HeartbeatReceiver 实现的方法。

### 1. 注册 Executor

HeartbeatReceiver 继承了 SparkListener，并实现了 onExecutorAdded 方法（见代码清单 9-1）。根据 3.3 节的内容，我们知道事件总线在接收到 SparkListenerExecutorAdded 消息后，将调用 HeartbeatReceiver 的 onExecutorAdded 方法，这样 HeartbeatReceiver 将监听到 Executor 的添加。

代码清单 9-1 HeartbeatReceiver 实现的 onExecutorAdded 方法

---

```
override def onExecutorAdded(executorAdded: SparkListenerExecutorAdded): Unit = {
    addExecutor(executorAdded.executorId)
}
```

---

根据代码清单 9-1，onExecutorAdded 方法从 SparkListenerExecutorAdded 事件中拿到 Executor 的身份标识后，将调用 addExecutor 方法。addExecutor 方法的实现如下。

```
def addExecutor(executorId: String): Option[Future[Boolean]] = {
    Option(self).map(_.ask[Boolean](ExecutorRegistered(executorId)))
}
```

根据上述代码，addExecutor 方法将向 HeartbeatReceiver 自己发送 ExecutorRegistered 消息。HeartbeatReceiver 继承了 ThreadSafeRpcEndpoint，并实现了 receiveAndReply 方法用以接收消息。HeartbeatReceiver 的 receiveAndReply 方法中处理 ExecutorRegistered 消息的代码如下。

```
case ExecutorRegistered(executorId) =>
    executorLastSeen(executorId) = clock.currentTimeMillis()
    context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 ExecutorRegistered 消息后，取出 Executor-

Registered 消息携带的要注册的 Executor 的身份标识，并把此 Executor 的身份标识与最后一次见面时间（即当前的时间戳）放入 executorLastSeen 中，最后向 HeartbeatReceiver 自己回复 true。

## 2. 移除 Executor

HeartbeatReceiver 继承了 SparkListener，并实现了 onExecutorRemoved 方法（见代码清单 9-2），这样 HeartbeatReceiver 将监听到 Executor 的移除。

代码清单9-2 HeartbeatReceiver实现的onExecutorRemoved方法

---

```
override def onExecutorRemoved(executorRemoved: SparkListenerExecutorRemoved): Unit = {
    removeExecutor(executorRemoved.executorId)
}
```

---

根据代码清单 9-2，onExecutorRemoved 方法从 SparkListenerExecutorRemoved 事件中拿到 Executor 的身份标识后，将调用 removeExecutor 方法。removeExecutor 方法的实现如下。

```
def removeExecutor(executorId: String): Option[Future[Boolean]] = {
    Option(self).map(_.ask[Boolean](ExecutorRemoved(executorId)))
}
```

根据上述代码，removeExecutor 方法将向 HeartbeatReceiver 自己发送 ExecutorRemoved 消息。HeartbeatReceiver 的 receiveAndReply 方法中处理 ExecutorRemoved 消息的代码如下。

```
case ExecutorRemoved(executorId) =>
    executorLastSeen.remove(executorId)
    context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 ExecutorRemoved 消息后，取出 ExecutorRemoved 消息携带的要移除的 Executor 的身份标识，并从 executorLastSeen 中移除此 Executor 的缓存，最后向 HeartbeatReceiver 自己回复 true。

## 3. TaskSchedulerIsSet 消息

TaskSchedulerIsSet 消息表示 SparkContext 的 \_taskScheduler 属性已经持有了 TaskScheduler 的引用。

在 4.5 节，我们曾经介绍过创建 TaskScheduler 和 DAGScheduler 的内容，根据代码清单 4-17，在创建了 TaskScheduler 后，SparkContext 的 \_taskScheduler 属性将持有了 TaskScheduler 的引用，然后 SparkContext 将向 HeartbeatReceiver 发送 TaskSchedulerIsSet 消息。

HeartbeatReceiver 的 receiveAndReply 方法中处理 TaskSchedulerIsSet 消息的代码如下。

```
case TaskSchedulerIsSet =>
    scheduler = sc.taskScheduler
    context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 TaskSchedulerIsSet 消息后，将获取 Spark-

Context 的 \_taskScheduler 属性持有的 TaskScheduler 的引用，并由自身的 scheduler 属性保存，最后向 SparkContext 回复 true。

#### 4. 检查超时的 Executor

HeartbeatReceiver 实现了 RpcEndpoint 的 onStart 方法，根据 5.3 节对 Dispatcher 的分析，我们知道当向 Dispatcher 注册 HeartbeatReceiver 时，与 HeartbeatReceiver 对应的 Inbox 将向自身的 messages 列表中放入 OnStart 消息，之后 Dispatcher 将处理 OnStart 消息，并调用 HeartbeatReceiver 的 onStart 方法。HeartbeatReceiver 的 onStart 方法的实现如代码清单 9-3 所示。

代码清单9-3 HeartbeatReceiver的onStart方法

---

```
override def onStart(): Unit = {
    timeoutCheckingTask = eventLoopThread.scheduleAtFixedRate(new Runnable {
        override def run(): Unit = Utils.tryLogNonFatalError {
            Option(self).foreach(_.ask[Boolean](ExpireDeadHosts))
        }
    }, 0, checkTimeoutIntervalMs, TimeUnit.MILLISECONDS)
}
```

---

根据代码清单 9-3，HeartbeatReceiver 的 onStart 方法将创建定时调度 timeoutCheckingTask。timeoutCheckingTask 按照 checkTimeoutIntervalMs 指定的间隔，向 HeartbeatReceiver 自身发送 ExpireDeadHosts 消息。HeartbeatReceiver 的 receiveAndReply 方法中处理 ExpireDeadHosts 消息的代码如下。

```
case ExpireDeadHosts =>
    expireDeadHosts()
    context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 ExpireDeadHosts 消息后，将调用 expireDeadHosts 方法，然后向 HeartbeatReceiver 自己回复 true。expireDeadHosts 方法（见代码清单 9-4）用于检查超时的 Executor。

代码清单9-4 查超时的Executor

---

```
private def expireDeadHosts(): Unit = {
    logTrace("Checking for hosts with no recent heartbeats in HeartbeatReceiver.")
    val now = clock.currentTimeMillis()
    for ((executorId, lastSeenMs) <- executorLastSeen) {
        if (now - lastSeenMs > executorTimeoutMs) {
            logWarning(s"Removing executor $executorId with no recent heartbeats: " +
                s"${now - lastSeenMs} ms exceeds timeout $executorTimeoutMs ms")
            scheduler.executorLost(executorId, SlaveLost("Executor heartbeat " +
                s"timed out after ${now - lastSeenMs} ms")) // 移除丢失的Executor
            killExecutorThread.submit(new Runnable {
                override def run(): Unit = Utils.tryLogNonFatalError {
                    sc.killAndReplaceExecutor(executorId) // “杀死” Executor
                }
            })
        }
    }
}
```

---

```
        executorLastSeen.remove(executorId) // 移除对Executor的超时检查
    }
}
}
```

---

根据代码清单 9-4, expireDeadHosts 方法将遍历 executorLastSeen 中的每个 Executor 的身份标识对应的最后一次见面时间 (lastSeenMs), 并对当前时间 (now) 与 lastSeenMs 的差值大于 executorTimeoutMs 的 Executor (这说明此 Executor 已经太长时间没有向 HeartbeatReceiver 发送 HeartBeat 了) 作以下处理。

- 1) 调用 TaskSchedulerImpl 的 executorLost 方法 (见代码清单 7-107) 移除丢失的 Executor, 并对 Executor 上正在运行的 Task 重新分配资源后进行调度。
- 2) 向单线程线程池 killExecutorThread 提交“杀死”Executor 的任务。“杀死”Executor 的任务通过调用 SparkContext 的 killAndReplaceExecutor 方法“杀死”Executor。
- 3) 从 executorLastSeen 中移除此 Executor。

**小贴士:** 通过异步线程“杀死”(kill) Executor, 是为了不阻塞执行 expireDeadHosts 方法的线程。

## 5. Executor 的心跳

每个活跃的 Executor 都会向 Driver 上的 HeartbeatReceiver 发送 HeartBeat 消息 (将在 9.2.1 节详细介绍), HeartbeatReceiver 在接收到 HeartBeat 消息后, 无论如何处理, 都将向 Executor 回复 HeartbeatResponse 消息。HeartbeatResponse 的实现如下。

```
private[spark] case class HeartbeatResponse(reregisterBlockManager: Boolean)
```

HeartbeatResponse 消息携带的 reregisterBlockManager 表示是否要求 Executor 重新向 BlockManagerMaster 注册 BlockManager。

HeartbeatReceiver 的 receiveAndReply 方法中处理 HeartBeat 消息的代码如下。

```
case heartbeat @ Heartbeat(executorId, accumUpdates, blockManagerId) =>
  if (scheduler != null) {
    if (executorLastSeen.contains(executorId)) {
      executorLastSeen(executorId) = clock.currentTimeMillis() // 更新此Executor的最
                                                               后一次见面时间
      eventLoopThread.submit(new Runnable {
        override def run(): Unit = Utils.tryLogNonFatalError {
          val unknownExecutor = !scheduler.executorHeartbeatReceived(
            executorId, accumUpdates, blockManagerId)
          val response = HeartbeatResponse(reregisterBlockManager = unknown-
            Executor)
          context.reply(response) // 回复HeartbeatResponse消息
        }
      })
    } else {
      logDebug(s"Received heartbeat from unknown executor $executorId")
      context.reply(HeartbeatResponse(reregisterBlockManager = true))
    }
  }
}
```

```

    }
} else {
  logWarning(s"Dropping $heartbeat because TaskScheduler is not ready yet")
  context.reply(HeartbeatResponse(reregisterBlockManager = true))
}

```

根据上述代码，HeartbeatReceiver 接收到 HeartBeat 消息后，如果 TaskSchedulerImpl 为空，则向 Executor 发送 HeartbeatResponse 消息，此消息将会要求 BlockManager 重新注册。如果 TaskSchedulerImpl 不为空，那么将执行以下操作。

1) 如果 executorLastSeen 中包含 HeartBeat 消息携带的 Executor 的身份标识，那么更新此 Executor 的最后一次见面时间，然后向 eventLoopThread 提交用于调用 TaskSchedulerImpl 的 executorHeartbeatReceived 方法的任务，最后向 Executor 回复 HeartbeatResponse 消息。TaskSchedulerImpl 的 executorHeartbeatReceived 方法用于更新正在处理的 Task 的度量信息，并且让 BlockManagerMaster 知道 BlockManager 仍然“活着”。

2) 如果 executorLastSeen 中不包含 HeartBeat 消息携带的 Executor 的身份标识，那么向 Executor 发送 HeartbeatResponse 消息，此消息将会要求 BlockManager 重新注册。这种情况发生在刚刚从 executorLastSeen 中移除 Executor 及其最后见面时间后，却收到了 Executor 在被移除之前发送的 HeartBeat 消息。

经过对 HeartbeatReceiver 的实现分析，其工作原理可以用图 9-1 表示。

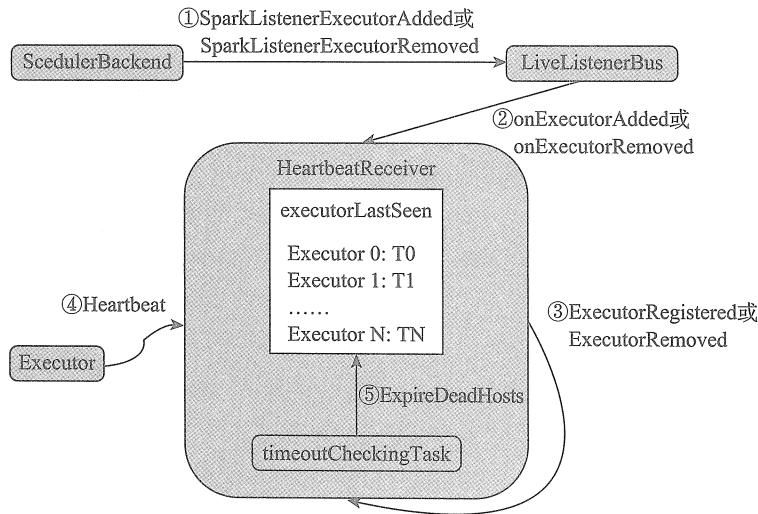


图 9-1 HeartbeatReceiver 的工作原理

这里对图 9-1 中的各个序号进行说明。

序号①：当 SchedulerBackend 已经创建或注册了 Executor 后，SchedulerBackend 将向 LiveListenerBus 投递 SparkListenerExecutorAdded 事件，当 Executor 与集群断开连接后，SchedulerBackend 将向 LiveListenerBus 投递 SparkListenerExecutorRemoved 事件。

序号②：由于 HeartbeatReceiver 也实现了 SparkListener，因此 LiveListenerBus 内部的异步线程在处理事件时，如果发现事件是 SparkListenerExecutorAdded 或 SparkListenerExecutorRemoved，将调用 HeartbeatReceiver 的 onExecutorAdded 或 onExecutorRemoved 方法。

序号③：当调用 HeartbeatReceiver 的 onExecutorAdded 或 onExecutorRemoved 方法时，HeartbeatReceiver 将向自己发送 ExecutorRegistered 消息或 ExecutorRemoved 消息。HeartbeatReceiver 接收到 ExecutorRegistered 消息，将把 Executor 的身份标识和当前系统时间作为对偶放入 executorLastSeen 缓存中。HeartbeatReceiver 接收到 ExecutorRemoved 消息，将把 Executor 的身份标识和对应的系统时间从 executorLastSeen 缓存中清除。

序号④：Executor 将不断向 HeartbeatReceiver 发送 Heartbeat 消息，HeartbeatReceiver 接收到 Heartbeat 消息后，将更新 executorLastSeen 缓存中 Executor 的身份标识对应的时间为系统当前时间。

序号⑤：HeartbeatReceiver 内部的检查 Executor 超时的定时任务，每隔一段时间将向 HeartbeatReceiver 自身发送 ExpireDeadHosts 消息，HeartbeatReceiver 接收到 ExpireDeadHosts 消息后，进行多项处理，移除丢失的 Executor、对 Executor 上正在运行的 Task 重新分配资源后进行调度、kill 丢失的 Executor、从 executorLastSeen 中移除此 Executor。

## 9.2 Executor 的实现分析

根据第 7 章对 LocalEndpoint 的 reviveOffers 方法（见代码清单 7-89）的介绍，我们知道最终会调用 Executor 的 launchTask 方法运行 Task。由于第 7 章主要介绍调度系统的内容，因此笔者没有分析 Executor 的实现，但现在已是继续深入的良好契机。

Executor 中包含以下属性。

- executorId：当前 Executor 的身份标识。
- executorHostname：当前 Executor 所在的主机名。
- env：即 SparkEnv。
- userClassPath：用户指定的类路径。可通过 spark.executor.extraClassPath 属性进行配置。如果有多个类路径，可以在配置时用英文逗号分隔。
- isLocal：是否是 local 部署模式。
- currentFiles：当前执行的 Task 所需要的文件。
- currentJars：当前执行的 Task 所需要的 Jar 包。
- EMPTY\_BYTE\_BUFFER：空的 java.nio.ByteBuffer，定义如下。

```
private val EMPTY_BYTE_BUFFER = ByteBuffer.wrap(new Array[Byte](0))
```

- conf：即 SparkConf。
- threadPool：使用 Executors.newCachedThreadPool 方式创建的 ThreadPoolExecutor，

用此线程池运行的线程将以 Executor task launch worker 为前缀。

- executorSource：类型为 ExecutorSource，是 Executor 的度量来源。
- taskReaperPool：使用 Executors.newCachedThreadPool 方式创建的 ThreadPoolExecutor，此线程池执行的线程用于监督 Task 的 kill 和取消。
- taskReaperForTask：用户缓存正在被 kill 的 Task 的身份标识与执行 kill 工作的任务收割者（TaskReaper）之间的映射关系。
- userClassPathFirst：是否首先从用户指定的类路径中加载类，然后再去 Spark 的 Jar 文件中加载。可通过 spark.executor.userClassPathFirst 属性配置，默认为 false。
- taskReaperEnabled：是否监控 kill 或中断 Task。可通过 spark.task.reaper.enabled 属性配置，默认为 false。
- urlClassLoader：Task 需要的类加载器。
- replClassLoader：当使用交互式解释器环境（Read-Evaluate-Print-Loop，简称 REPL）时，此类加载器用于加载 REPL 根据用户键入的代码定义的新类型。
- maxDirectResultSize：直接结果的最大大小。取 spark.task.maxDirectResultSize 属性（默认为  $1L << 20$ ，即 1 048 576）与 spark.rpc.message.maxSize 属性（默认为 128MB）之间的最小值。
- maxResultSize：结果的最大限制。此属性通过调用 Utils 工具类的 getMaxResultSize 方法获得，默认为 1GB。Task 运行的结果如果超过 maxResultSize，则会被删除。Task 运行的结果如果小于等于 maxResultSize 且大于 maxDirectResultSize，则会写入本地存储体系。Task 运行的结果如果小于等于 maxDirectResultSize，则会直接返回给 Driver。
- runningTasks：用于维护正在运行的 Task 的身份标识与 TaskRunner 之间的映射关系。
- heartbeater：只有一个线程的 ScheduledThreadPoolExecutor，此线程池运行的线程以 driver-heartbeater 作为名称。
- heartbeatReceiverRef：HeartbeatReceiver 的 RpcEndpointRef，通过调用 RpcEnv 的 setupEndpointRef 方法（见代码清单 5-42）获得。
- HEARTBEAT\_MAX\_FAILURES：心跳的最大失败次数。可通过 spark.executor.heartbeat.maxFailures 属性配置，默认为 60。
- heartbeatFailures：心跳失败数的计数器，初始值为 0。

在 Executor 的伴生对象中还定义了 ThreadLocal 属性 taskDeserializationProps。

```
val taskDeserializationProps: ThreadLocal[Properties] = new ThreadLocal[Properties]
```

### 9.2.1 Executor 的心跳报告

在初始化 Executor 的过程中，Executor 会调用自己的 startDriverHeartbeater 方法启动心跳报告的定时任务。startDriverHeartbeater 方法的实现如代码清单 9-5 所示。

## 代码清单9-5 启动心跳报告

---

```

private def startDriverHeartbeater(): Unit = {
    val intervalMs = conf.getTimeAsMs("spark.executor.heartbeatInterval", "10s")
    val initialDelay = intervalMs + (math.random * intervalMs).asInstanceOf[Int]

    val heartbeatTask = new Runnable() { // 向Driver报告心跳
        override def run(): Unit = Utils.logUncaughtExceptions(reportHeartBeat())
    }
    heartbeater.scheduleAtFixedRate(heartbeatTask, initialDelay, intervalMs, Time-
        Unit.MILLISECONDS)
}

```

---

根据代码清单 9-5，startDriverHeartbeater 方法的执行步骤如下。

- 1) 获取发送心跳报告的时间间隔 intervalMs。可以看到，intervalMs 可通过 spark.executor.heartbeatInterval 属性配置，默认为 10s。
- 2) 获取心跳定时器第一次执行的时间延迟 initialDelay。可以看到，initialDelay 是在 intervalMs 的基础上累加了随机数。
- 3) 创建心跳任务 heartbeatTask。heartbeatTask 任务将通过 Utils 工具类的 logUncaughtExceptions 方法调用 reportHeartBeat 方法（见代码清单 9-6）报告心跳。
- 4) 使 heartbeater 定时调度执行 heartbeatTask。

## 代码清单9-6 报告心跳

---

```

private def reportHeartBeat(): Unit = {
    val accumUpdates = new ArrayBuffer[(Long, Seq[AccumulatorV2[_, _]])]()
    val curGCTime = computeTotalGcTime()

    for (taskRunner <- runningTasks.values().asScala) {
        if (taskRunner.task != null) {
            taskRunner.task.metrics.mergeShuffleReadMetrics()
            taskRunner.task.metrics.setJvmGCTime(curGCTime - taskRunner.startGCTime)
            accumUpdates += ((taskRunner.taskId, taskRunner.task.metrics.accumulators()))
        }
    }

    val message = Heartbeat(executorId, accumUpdates.toArray, env.blockManager.
        blockManagerId)
    try { // 向Driver的HeartbeatReceiver发送Heartbeat消息
        val response = heartbeatReceiverRef.askWithRetry[HeartbeatResponse](
            message, RpcTimeout(conf, "spark.executor.heartbeatInterval", "10s"))
        if (response.reregisterBlockManager) {
            logInfo("Told to re-register on heartbeat")
            env.blockManager.reregister()
        }
        heartbeatFailures = 0
    } catch {
        case NonFatal(e) =>
            logWarning("Issue communicating with driver in heartbeater", e)
            heartbeatFailures += 1
            if (heartbeatFailures >= HEARTBEAT_MAX_FAILURES) {
                logError(s"Exit as unable to send heartbeats to driver " +

```

---

```

        s"more than $HEARTBEAT_MAX_FAILURES times")
        System.exit(ExecutorExitCode.HEARTBEAT_FAILURE)
    }
}
}
}
```

根据代码清单 9-6，reportHeartBeat 方法的执行步骤如下。

- 1) 遍历 runningTasks 中正在运行的 Task，将每个 Task 的度量信息更新到数组缓冲 accumUpdates 中。
- 2) 创建 Heartbeat 消息。
- 3) 向 HeartbeatReceiver 发送 Heartbeat 消息，并接收 HeartbeatReceiver 的响应消息 HeartbeatResponse。
- 4) 如果 HeartbeatResponse 的 reregisterBlockManager 属性为 true，此时调用 BlockManager 的 reregister 方法（见代码清单 6-60）向 BlockManagerMaster 重新注册 BlockManager。
- 5) 将 heartbeatFailures 置为 0。

## 9.2.2 运行 Task

Executor 的 launchTask 方法用于运行 Task，其实现如代码清单 9-7 所示。

代码清单 9-7 运行Task

---

```
def launchTask(
  context: ExecutorBackend,
  taskId: Long,
  attemptNumber: Int,
  taskName: String,
  serializedTask: ByteBuffer): Unit = {
  val tr = new TaskRunner(context, taskId = taskId, attemptNumber = attempt-
    Number, taskName,
    serializedTask)
  runningTasks.put(taskId, tr)
  threadPool.execute(tr)
}
```

---

根据代码清单 9-7，launchTask 方法的执行步骤如下。

- 1) 创建 TaskRunner。
- 2) 将 Task 的身份标识与 TaskRunner 的对应关系放入 runningTasks。
- 3) 执行 TaskRunner。

TaskRunner 实现了 Runnable 接口，这样就可以作为线程要执行的任务。

TaskRunner 的构造器属性如下。

- execBackend：类型为 ExecutorBackend。根据 7.8 节对 SchedulerBackend 的介绍，我们知道 SchedulerBackend 有 CoarseGrainedSchedulerBackend 和 LocalSchedulerBackend 两个子类。由于 LocalSchedulerBackend 也继承了特质 ExecutorBackend，并实现了

其唯一的方法 statusUpdate，所以 local 部署模式下将 LocalSchedulerBackend 作为 ExecutorBackend。其他部署模式使用特质 ExecutorBackend 的另一个实现类 Coarse-GrainedExecutorBackend，来跟 Coarse-GrainedSchedulerBackend 配合。

- ❑ taskId: Task 的身份标识。
- ❑ attemptNumber: 任务尝试号。
- ❑ taskName: Task 的名称。
- ❑ serializedTask: 序列化后的 Task。

TaskRunner 中的其他属性如下。

- ❑ threadName: 线程名称。以 Executor task launch worker for task 为前缀，后跟 taskId。
- ❑ killed: Task 是否被 kill。
- ❑ threadId: 线程的 ID。getThreadId 方法专门用于获取 threadId 的值。
- ❑ finished: 任务是否已经完成。isFinished 方法专门用于获取 finished 的布尔值。
- ❑ startGCTime: 任务尝试开始运行前，JVM 进程执行 GC 已花费的时间。
- ❑ task: 要运行的 Task。通过对 Driver 传递过来的序列化的 Task 进行反序列化后获得。

TaskRunner 实现的 run 方法非常长，所以笔者将其实现分为以下六部分来讲解。

## 1. 运行准备

TaskRunner 实现的 run 方法中有关 Task 的运行准备的部分如下。

```
override def run(): Unit = {
    // 省略获取threadId、设置threadName的代码
    val taskMemoryManager = new TaskMemoryManager(env.memoryManager, taskId)
    // 省略次要代码
    Thread.currentThread.setContextClassLoader(replClassLoader)
    val ser = env.closureSerializer.newInstance()
    execBackend.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER) //更新
        Task为RUNNING
    var taskStart: Long = 0
    var taskStartCpu: Long = 0
    startGCTime = computeTotalGcTime() // 计算JVM进程执行GC已经花费的时间

    try {
        val (taskFiles, taskJars, taskProps, taskBytes) =
            Task.deserializeWithDependencies(serializedTask) // 对序列化的Task进行反序列化
        //Task所需的属性信息 (taskProps) 放入ThreadLocal
        Executor.taskDeserializationProps.set(taskProps)
        // 从taskFiles和taskJars中获取所需的资源依赖
        updateDependencies(taskFiles, taskJars)
        // 反序列化得到Task实例
        task = ser.deserialize[Task[Any]](taskBytes, Thread.currentThread.getContext-
            ClassLoader)
        task.localProperties = taskProps
        task.setTaskMemoryManager(taskMemoryManager) // 将TaskMemoryManager设置为
            Task的内存管理器
        // 省略当前TaskRunner的killed属性为true时，抛出TaskKilledException的代码
    }
}
```

根据上述代码，Task 的运行准备工作如下。

- 1 ) 创建 Task 尝试所需要的 TaskMemoryManager。
- 2 ) 设置当前线程上下文的类加载器为 replClassLoader。
- 3 ) 生成新的对闭包序列化的实例 ser。
- 4 ) 调用 ExecutorBackend 的 statusUpdate 方法，将 Task 的状态更新为 RUNNING。可参考 LocalSchedulerBackend 实现的 statusUpdate 方法（见代码清单 7-93）或 CoarseGrained-ExecutorBackend 的 statusUpdate 方法（见代码清单 9-92）。
- 5 ) 调用 computeTotalGcTime 方法计算 JVM 进程执行 GC 已经花费的时间。
- 6 ) 调用 Task 的 deserializeWithDependencies 方法对序列化的 Task 进行反序列化，得到任务所需的文件（taskFiles）、任务所需的 Jar 包（taskJars）、Task 所需的属性信息（taskProps）、任务本身（taskBytes）。
- 7 ) 使用 Executor 的伴生对象的 taskDeserializationProps 将 taskProps 放入 ThreadLocal。
- 8 ) 调用 updateDependencies 方法（见代码清单 9-8）从 taskFiles 和 taskJars 中获取所需的依赖。
- 9 ) 将 Task 的 ByteBuffer（即 taskBytes）反序列化为 Task 实例。
- 10 ) 将 taskProps 保存到 Task 的 localProperties 属性中。
- 11 ) 将刚创建的 TaskMemoryManager 设置为 Task 的内存管理器。

代码清单9-8 更新依赖的文件和Jar包

---

```

private def updateDependencies(newFiles: HashMap[String, Long], newJars: HashMap[String, Long]) {
    lazy val hadoopConf = SparkHadoopUtil.get.newConfiguration(conf)
    synchronized {
        for ((name, timestamp) <- newFiles if currentFiles.getOrElse(name, -1L) <
            timestamp) {
            logInfo("Fetching " + name + " with timestamp " + timestamp)
            Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
                env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
            currentFiles(name) = timestamp
        }
        for ((name, timestamp) <- newJars) {
            val localName = name.split("/").last
            val currentTimeStamp = currentJars.get(name)
                .orElse(currentJars.get(localName))
                .getOrElse(-1L)
            if (currentTimeStamp < timestamp) {
                logInfo("Fetching " + name + " with timestamp " + timestamp)
                Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
                    env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
                currentJars(name) = timestamp
            }
            val url = new File(SparkFiles.getRootDirectory(), localName).toURI.toURL
            if (!urlClassLoader.getURLs().contains(url)) {
                logInfo("Adding " + url + " to class loader")
                urlClassLoader.addURL(url)
            }
        }
    }
}

```

```

        }
    }
}

```

根据代码清单 9-8, updateDependencies 方法首先下载 Task 所需的文件, 并将文件名与时间戳的关系放入 currentFiles 缓存, 然后下载 Task 所需的 Jar 包, 并将 Jar 文件名与时间戳的关系放入 currentJars 缓存。Jar 包还将添加到 urlClassLoader 的加载路径中。下载文件都是通过 Utils 工具类的 fetchFile 方法, 其具体实现请参阅附录 A。

## 2. 运行

TaskRunner 实现的 run 方法中有关运行 Task 的部分如代码清单 9-9 所示。

**代码清单9-9 TaskRunner实现的run方法中有关运行Task的部分**

---

```

val value = try {
    val res = task.run(
        taskId,
        attemptNumber,
        metricsSystem = env.metricsSystem)
    threwException = false
    res
} finally {
// 省略finally里的代码
}

```

---

根据代码清单 9-9, TaskRunner 调用了 Task 的 run 方法 (见代码清单 8-37), 根据 8.5.2 节对 Task 的 run 方法的分析, 我们知道 run 方法是一个模板方法, 真正运行 Task 的是子类实现的 runTask 方法。

## 3. 资源回收

在上面的代码中, 忽略了 finally 块中的代码, 这些代码的实现如下。

```

val releasedLocks = env.blockManager.releaseAllLocksForTask(taskId) // 释放所有的
    Block 锁
val freedMemory = taskMemoryManager.cleanUpAllAllocatedMemory() // 清空分配给任务尝
    尝的内存

if (freedMemory > 0 && !threwException) {
    val errMsg = s"Managed memory leak detected; size = $freedMemory bytes, TID =
        $taskId"
    if (conf.getBoolean("spark.unsafe.exceptionOnMemoryLeak", false)) {
        throw new SparkException(errMsg)
    } else {
        logWarning(errMsg)
    }
}

if (releasedLocks.nonEmpty && !threwException) {
    val errMsg =

```

```

    s"${releasedLocks.size} block locks were not released by TID = $task-Id:\n"
    n" +
    releasedLocks.mkString("[", " ", " ", "]")
  if (conf.getBoolean("spark.storage.exceptionOnPinLeak", false)) {
    throw new SparkException(errMsg)
  } else {
    logWarning(errMsg)
  }
}
}

```

根据上述代码，可以看到 finally 主要用于资源的回收，具体处理步骤如下。

- 1 ) 调用 BlockManager 的 releaseAllLocksForTask 方法释放被当前任务尝试占用的所有 Block 的锁，并通知所有等待获取锁的线程。
- 2 ) 调用 TaskMemoryManager 的 cleanUpAllAllocatedMemory 方法清空所有分配给当前任务尝试的 Page 和内存。
- 3 ) 如果当前任务尝试有内存溢出或者存在 Block 的锁无法释放，则抛出异常或打印警告信息。

#### 4. 执行结果序列化

下面的代码将对 Task 执行返回的结果 value 进行序列化处理。

```

val resultSer = env.serializer.newInstance()
val beforeSerialization = System.currentTimeMillis()
val valueBytes = resultSer.serialize(value)
val afterSerialization = System.currentTimeMillis()

// 省略度量更新相关代码
val accumUpdates = task.collectAccumulatorUpdates()
val directResult = new DirectTaskResult(valueBytes, accumUpdates)
val serializedDirectResult = ser.serialize(directResult)
val resultSize = serializedDirectResult.limit

```

#### 5. 更新度量

在介绍执行结果的序列化时，忽略了其中对度量进行更新的代码，这些代码如下。

```

task.metrics.setExecutorDeserializeTime(
  (taskStart - deserializeStartTime) + task.executorDeserializeTime)
task.metrics.setExecutorDeserializeCpuTime(
  (taskStartCpu - deserializeStartTimeCpuTime) + task.executorDeserializeCpuTime)
task.metrics.setExecutorRunTime((taskFinish - taskStart) - task.executorDeserializeTime)
task.metrics.setExecutorCpuTime(
  (taskFinishCpu - taskStartCpu) - task.executorDeserializeCpuTime)
task.metrics.setJvmGCTime(computeTotalGcTime() - startGCTime)
task.metrics.setResultSerializationTime(afterSerialization - beforeSerialization)

```

#### 6. 执行结果发送

Task 运行的最后一步是将执行结果发送给 Driver 应用程序，如代码清单 9-10 所示。

## 代码清单9-10 执行结果的发送

---

```

val serializedResult: ByteBuffer = {
    if (maxResultSize > 0 && resultSize > maxResultSize) {
        // 将结果的大小序列化为serializedResult，并不会保存执行结果
        ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId), result-
            Size))
    } else if (resultSize > maxDirectResultSize) {
        val blockId = TaskResultBlockId(taskId)
        env.blockManager.putBytes( // 将结果写入本地存储体系
            blockId,
            new ChunkedByteBuffer(serializedDirectResult.duplicate(),
                StorageLevel.MEMORY_AND_DISK_SER)
        // 将结果的大小序列化为serializedResult
        ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
    } else {
        logInfo(s"Finished $taskName (TID $taskId). $resultSize bytes result sent
            to driver")
        serializedDirectResult // 直接将结果序列化为serializedResult
    }
}
// 将Task的状态更新为FINISHED，并且将serializedResult发送给Driver
execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)

```

---

根据代码清单 9-10，执行结果发送的处理逻辑如下。

- 1) 如果 maxResultSize 大于 0 且任务尝试运行的结果超过 maxResultSize，那么只会将结果的大小序列化为 serializedResult，并且不会保存执行结果。
- 2) 如果任务尝试运行的结果小于等于 maxResultSize 且大于 maxDirectResultSize，则会将结果写入本地存储体系，并将结果的大小序列化为 serializedResult。
- 3) 如果任务尝试运行的结果小于等于 maxDirectResultSize，则会直接将结果序列化为 serializedResult。
- 4) 调用 ExecutorBackend 的 statusUpdate 方法将 Task 的状态更新为 FINISHED，并且将 serializedResult 发送给 Driver。可参考 LocalSchedulerBackend 实现的 statusUpdate 方法（见代码清单 7-93）或 CoarseGrainedExecutorBackend 的 statusUpdate 方法（见代码清单 9-92）。

### 9.3 local 部署模式

本书在讲解各章内容时，主要以 local 模式为例。local 部署模式只有 Driver，没有 Master 和 Worker，执行任务的 Executor 与 Driver 在同一个 JVM 进程内。local 模式中使用的 ExecutorBackend 和 SchedulerBackend 的实现类都是 LocalSchedulerBackend。在第 7 章介绍调度系统时已经分析过 local 部署模式下的 LocalSchedulerBackend、LocalEndpoint 等组件的实现，本节将以图形的方式展现 local 部署模式的启动、local 部署模式下的任务提交与执行等流程。

local 部署模式的启动过程如图 9-2 所示。

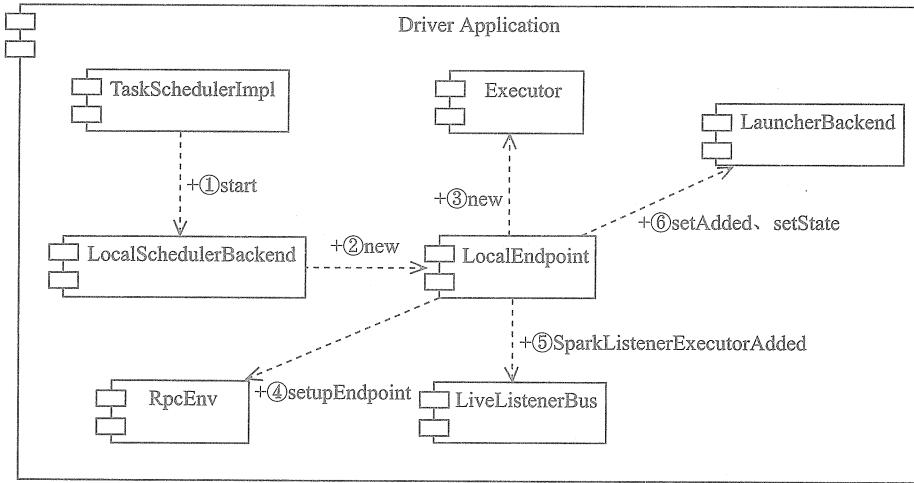


图 9-2 local 模式的启动过程

图 9-2 展示的执行过程已经在介绍 TaskSchedulerImpl 和 LocalSchedulerBackend 的启动时详细介绍过，所以不再赘述。

local 部署模式下的任务提交与执行过程如图 9-3 所示。

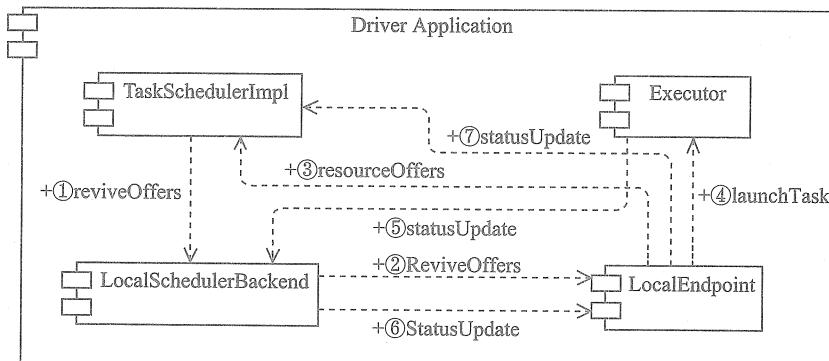


图 9-3 local 模式下的任务提交与执行

图 9-3 中 local 模式下的任务提交执行过程如下。

- 1 ) TaskSchedulerImpl 的 submitTasks 方法（见代码清单 7-99）在提交 Task 的最后会调用 LocalSchedulerBackend 的 reviveOffers 方法。
- 2 ) LocalSchedulerBackend 的 reviveOffers 方法（见代码清单 7-92）只是向 LocalEndpoint 发送 ReviveOffers 消息。
- 3 ) LocalEndpoint 收到 ReviveOffers 消息后（见代码清单 7-88），调用 TaskSchedulerImpl 的 resourceOffers 方法（见代码清单 7-101）申请资源，TaskSchedulerImpl 将根据任务申请的 CPU 核数、内存、本地化等条件为其分配资源。

- 4) 任务获得资源后，调用 Executor 的 launchTask 方法（见代码清单 9-7）运行任务。
- 5) 在任务运行过程（在 9.2.2 节已经详细介绍）中，Executor 中运行的 TaskRunner 通过调用 LocalSchedulerBackend 的 statusUpdate 方法更新 Task 的状态。
- 6) LocalSchedulerBackend 的 statusUpdate 方法将向 LocalEndpoint 发送 StatusUpdate 消息。
- 7) LocalEndpoint 接收到 StatusUpdate 消息，将调用 TaskSchedulerImpl 的 statusUpdate 方法（见代码清单 7-103）更新任务的状态。

## 9.4 持久化引擎 PersistenceEngine

PersistenceEngine 用于当 Master 发生故障后，通过领导选举选择其他 Master 接替整个集群的管理工作时，能够使得新激活的 Master 有能力从故障中恢复整个集群的状态信息，进而恢复对集群资源的管理和分配。抽象类 PersistenceEngine 定义了对 Master 必需的任何状态信息进行持久化的接口规范。实现 PersistenceEngine 必须满足以下机制。

- 在完成新的 Application 的注册之前，addApplication 方法必须被调用。
- 在完成新的 Worker 的注册之前，addWorker 方法必须被调用。
- removeApplication 方法和 removeWorker 方法可以在任何时候调用。

在以上机制的保证下，整个集群的 Worker、Driver 和 Application 的信息都被持久化，集群因此可以在领导选举和主 Master 的切换后，对集群状态进行恢复。

抽象类 PersistenceEngine 的定义如代码清单 9-11 所示。

代码清单9-11 PersistenceEngine的定义

---

```

@DeveloperApi
abstract class PersistenceEngine {
  def persist(name: String, obj: Object): Unit
  def unpersist(name: String): Unit
  def read[T: ClassTag](prefix: String): Seq[T]
  final def addApplication(app: ApplicationInfo): Unit = {
    persist("app_" + app.id, app)
  }
  final def removeApplication(app: ApplicationInfo): Unit = {
    unpersist("app_" + app.id)
  }
  final def addWorker(worker: WorkerInfo): Unit = {
    persist("worker_" + worker.id, worker)
  }
  final def removeWorker(worker: WorkerInfo): Unit = {
    unpersist("worker_" + worker.id)
  }
  final def addDriver(driver: DriverInfo): Unit = {
    persist("driver_" + driver.id, driver)
  }
}

```

```

final def removeDriver(driver: DriverInfo): Unit = {
    unpersist("driver_" + driver.id)
}
final def readPersistedData(
    rpcEnv: RpcEnv): (Seq[ApplicationInfo], Seq[DriverInfo], Seq[WorkerInfo]) = {
    rpcEnv.deserialize { () =>
        (read[ApplicationInfo]("app_"), read[DriverInfo]("driver_"), read[Worker-
            Info]("worker_"))
    }
}
def close() {}
}

```

---

根据代码清单 9-11，抽象类 PersistenceEngine 定义的方法分别如下。

- persist：对对象进行序列化和持久化的抽象方法，具体的实现依赖于底层使用的存储。
- unpersist：对对象进行非持久化（从存储中删除）的抽象方法，具体的实现依赖于底层使用的存储。
- read：读取匹配给定前缀的所有对象。
- addApplication：对应用的信息（ApplicationInfo）进行持久化的模板方法，其依赖于 persist 的具体实现。
- removeApplication：对应用的信息（ApplicationInfo）进行非持久化的模板方法，其依赖于 unpersist 的具体实现。
- addWorker：对 Worker 的信息（WorkerInfo）进行持久化的模板方法，其依赖于 persist 的具体实现。
- removeWorker：对 Worker 的信息（WorkerInfo）进行非持久化的模板方法，其依赖于 unpersist 的具体实现。
- addDriver：对 Driver 的信息（DriverInfo）进行持久化的模板方法，其依赖于 persist 的具体实现。
- removeDriver：对 Driver 的信息（DriverInfo）进行非持久化的模板方法，其依赖于 unpersist 的具体实现。
- readPersistedData：读取并反序列化所有持久化的数据。
- close：用于关闭 PersistenceEngine。

抽象类 PersistenceEngine 有四个实现子类，如图 9-4 所示。

从图 9-4 中可以看到，PersistenceEngine 一共有四种实现，下面对它们作简单介绍。BlackHolePersistenceEngine 正如“黑洞”一样，其实现都是空实现。

```

private[master] class BlackHolePersistenceEngine extends PersistenceEngine {
    override def persist(name: String, obj: Object): Unit = {}
    override def unpersist(name: String): Unit = {}
    override def read[T: ClassTag](name: String): Seq[T] = Nil
}

```

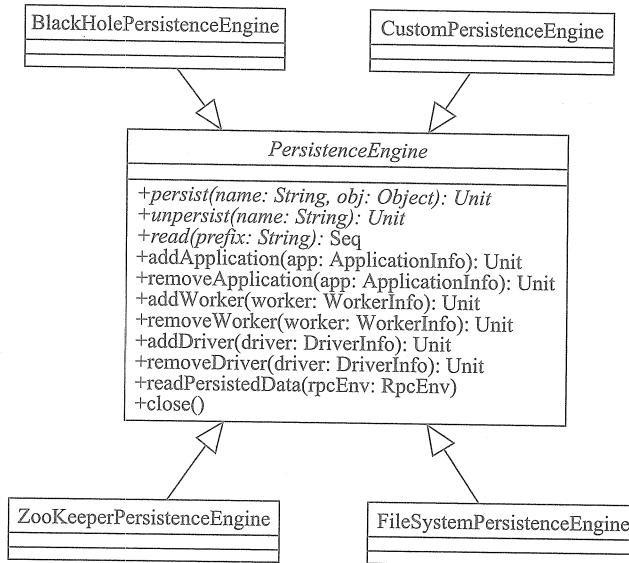


图 9-4 PersistenceEngine 的继承体系

CustomPersistenceEngine 是用于单元测试的实现。FileSystemPersistenceEngine 和 ZooKeeperPersistenceEngine 是真正可用于生产环境的实现类，因此本节将主要对后两者进行分析。

#### 9.4.1 基于文件系统的持久化引擎

FileSystemPersistenceEngine 是基于文件系统的持久化引擎。对于 ApplicationInfo、WorkerInfo 及 DriverInfo，FileSystemPersistenceEngine 会将它们的数据存储到磁盘上的单个文件夹中，当要移除它们时，这些磁盘文件将被删除。由于不同的 Master 往往不在同一个机器节点上，因此在使用 FileSystemPersistenceEngine 时，底层的文件系统应该是分布式的。

FileSystemPersistenceEngine 一共有两个属性，分别如下。

- dir：持久化的根目录。
- serializer：持久化时使用的序列化器。

了解了 FileSystemPersistenceEngine 的属性，下面将逐一介绍它实现的方法。

##### 1. persist

FileSystemPersistenceEngine 实现的 persist 方法如代码清单 9-12 所示。

代码清单9-12 FileSystemPersistenceEngine的persist方法

---

```

override def persist(name: String, obj: Object): Unit = {
  serializeToFile(new File(dir + File.separator + name), obj)
}
private def serializeToFile(file: File, value: AnyRef) {
  val created = file.createNewFile()
}
  
```

```

if (!created) { throw new IllegalStateException("Could not create file: " + file) }
val fileOut = new FileOutputStream(file)
var out: SerializationStream = null
Utils.tryWithSafeFinally {
    out = serializer.newInstance().serializeStream(fileOut)
    out.writeObject(value)
} {
    fileOut.close()
    if (out != null) {
        out.close()
    }
}
}

```

根据代码清单 9-12，persist 方法实际是通过创建文件，打开文件输出流，并对数据进行序列化后写入磁盘的。

## 2. unpersist

FileSystemPersistenceEngine 实现的 unpersist 方法实际是通过删除对应的文件实现的，如代码清单 9-13 所示。

代码清单9-13 FileSystemPersistenceEngine的unpersist方法

---

```

override def unpersist(name: String): Unit = {
    val f = new File(dir + File.separator + name)
    if (!f.delete()) {
        logWarning(s"Error deleting ${f.getPath()}")
    }
}

```

---

## 3. read

FileSystemPersistenceEngine 实现的 read 方法（见代码清单 9-14）实际是通过过滤出持久化根目录下，文件名匹配前缀的所有文件，并对每个文件的数据反序列化实现的。

代码清单9-14 FileSystemPersistenceEngine的read方法

---

```

override def read[T: ClassTag](prefix: String): Seq[T] = {
    val files = new File(dir).listFiles().filter(_.getName.startsWith(prefix))
    files.map(deserializeFromFile[T])
}

private def deserializeFromFile[T](file: File)(implicit m: ClassTag[T]): T = {
    val fileIn = new FileInputStream(file)
    var in: DeserializationStream = null
    try {
        in = serializer.newInstance().deserializeStream(fileIn)
        in.readObject[T]()
    } finally {
        fileIn.close()
        if (in != null) {
            in.close()
        }
    }
}

```

---

```

    }
}

```

## 9.4.2 基于 ZooKeeper 的持久化引擎

ZooKeeperPersistenceEngine 是基于 ZooKeeper 的持久化引擎。对于 ApplicationInfo、WorkerInfo 及 DriverInfo，ZooKeeperPersistenceEngine 会将它们的数据存储到 ZooKeeper 的不同节点（也称为 Znode）中，当要移除它们时，这些节点将被删除。

ZooKeeperPersistenceEngine 有以下属性。

- conf：即 SparkConf。
- serializer：持久化时使用的序列化器。
- WORKING\_DIR：ZooKeeperPersistenceEngine 在 ZooKeeper 上的工作目录，是 Spark 基于 ZooKeeper 进行热备的根节点（可通过 spark.deploy.ZooKeeper.dir 属性配置，默认为 spark）的子节点 master\_status。
- zk：连接 ZooKeeper 的客户端，类型为 CuratorFramework。

**小贴士：**Curator 是 Netflix 公司开源的一个 ZooKeeper 客户端，与 ZooKeeper 提供的原生客户端相比，Curator 的抽象层次更高，其核心目标是帮助工程师管理 ZooKeeper 的相关操作，简化 ZooKeeper 客户端的开发量。Curator 现已提升为 Apache 的顶级项目。Curator-Framework 就是 Curator 提供的 API。

了解了 ZooKeeperPersistenceEngine 的属性，下面将逐一介绍它实现的方法。

### 1. persist

ZooKeeperPersistenceEngine 实现的 persist 方法如代码清单 9-15 所示。

代码清单 9-15 ZooKeeperPersistenceEngine 的 persist 方法

---

```

override def persist(name: String, obj: Object): Unit = {
  serializeToFile(WORKING_DIR + "/" + name, obj)
}

private def serializeToFile(path: String, value: AnyRef) {
  val serialized = serializer.newInstance().serialize(value)
  val bytes = new Array[Byte](serialized.remaining())
  serialized.get(bytes)
  zk.create().withMode(CreateMode.PERSISTENT).forPath(path, bytes)
}

```

---

根据代码清单 9-15，persist 方法实际是通过对数据进行序列化，然后将序列化的数据保存到 ZooKeeper 的 master\_status 节点下创建的持久化子节点中实现的。

**小贴士：**ZooKeeper 的节点的创建模式（CreateMode）共有四种，分别是持久化（PERSISTENT）、持久化与顺序化（PERSISTENT\_SEQUENTIAL）、临时（EPHEMERAL）、

临时与顺序化 (EPHEMERAL\_SEQUENTIAL)。对于顺序化的节点，ZooKeeper 会为其创建自增长的序号。对于临时节点，ZooKeeper 会在创建它的客户端退出后自动删除。对于持久化的节点，ZooKeeper 不会在创建它的客户端退出后自动删除，需要客户端主动删除。

## 2. unpersist

ZooKeeperPersistenceEngine 实现的 unpersist 方法实际是通过删除对应的 ZooKeeper 节点实现的，如代码清单 9-16 所示。

代码清单9-16 ZooKeeperPersistenceEngine的unpersist方法

---

```
override def unpersist(name: String): Unit = {
    zk.delete().forPath(WORKING_DIR + "/" + name)
}
```

---

## 3. read

ZooKeeperPersistenceEngine 实现的 read 方法实际是通过过滤出 ZooKeeper 的 master\_status 节点下文件名匹配前缀的所有子节点，并对每个子节点的数据反序列化实现的，如代码清单 9-17 所示。

代码清单9-17 ZooKeeperPersistenceEngine的read方法

---

```
override def read[T: ClassTag](prefix: String): Seq[T] = {
    zk.getChildren().forPath(WORKING_DIR).asScala
        .filter(_.startsWith(prefix)).flatMap(deserializeFromFile[T])
}
private def deserializeFromFile[T](filename: String)(implicit m: ClassTag[T]): Option[T] = {
    val fileData = zk.getData().forPath(WORKING_DIR + "/" + filename)
    try {
        Some(serializer.newInstance().deserialize[T](ByteBuffer.wrap(fileData)))
    } catch {
        case e: Exception =>
            logWarning("Exception while reading persisted file, deleting", e)
            zk.delete().forPath(WORKING_DIR + "/" + filename)
            None
    }
}
```

---

## 4. close

ZooKeeperPersistenceEngine 实现的 close 方法实际只是关闭了连接 ZooKeeper 的客户端，代码如下。

```
override def close() {
    zk.close()
}
```

## 9.5 领导选举代理

领导选举机制 (Leader Election) 可以保证集群虽然存在多个 Master，但是只有一

个 Master 处于激活 (Active) 状态，其他的 Master 处于支持 (Standby) 状态。当 Active 状态的 Master 出现故障时，会选举出一个 Standby 状态的 Master 作为新的 Active 状态的 Master。由于整个集群的 Worker、Driver 和 Application 的信息都已经通过持久化引擎持久化，因此切换 Master 时只会影响新任务的提交，对于正在运行中的任务没有任何影响。

特质 LeaderElectionAgent 定义了对当前的 Master 进行跟踪和领导选举代理的通用接口，其定义如下。

```
@DeveloperApi
trait LeaderElectionAgent {
    val masterInstance: LeaderElectable
    def stop() {} // to avoid noops in implementations.
}
```

上述代码中的 masterInstance 属性的类型是 LeaderElectable，特质 LeaderElectable 的定义如下。

```
@DeveloperApi
trait LeaderElectable {
    def electedLeader(): Unit
    def revokedLeadership(): Unit
}
```

LeaderElectable 定义了两个方法。

- electedLeader：被选举为领导。
- revokedLeadership：撤销领导关系。

LeaderElectable 目前只有 Master 这一个实现类。

特质 LeaderElectionAgent 的继承体系如图 9-5 所示。

从图 9-5 中，可以看到 LeaderElectionAgent 一共有三种实现。由于 CustomLeaderElectionAgent 用于单元测试，所以我们不对其过多介绍。其余两种实现可用于生产环境，本节将逐一进行介绍。

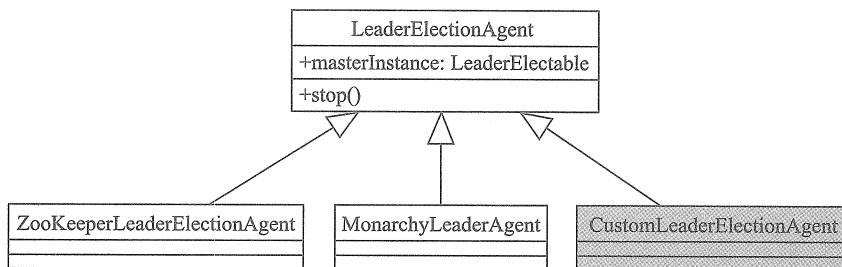


图 9-5 LeaderElectionAgent 的继承体系

## 1. MonarchyLeaderAgent 详解

MonarchyLeaderAgent 的实现非常简单，如代码清单 9-18 所示。

根据代码清单 9-18，可以看到 MonarchyLeaderAgent 在构造时会调用 masterInstance 的 electedLeader 方法选举领导。本章将在 9.6 节分析 Master 实现的 electedLeader 方法。

代码清单9-18 MonarchyLeaderAgent的实现

---

```
private[spark] class MonarchyLeaderAgent(val masterInstance: LeaderElectable)
  extends LeaderElectionAgent {
  masterInstance.electedLeader()
}
```

---

## 2. ZooKeeperLeaderElectionAgent 详解

ZooKeeperLeaderElectionAgent 是借助 ZooKeeper 实现的领导选举代理。ZooKeeperLeaderElectionAgent 中的属性如下。

- 1) WORKING\_DIR：ZooKeeperLeaderElectionAgent 在 ZooKeeper 上的工作目录，是 Spark 基于 ZooKeeper 进行热备的根节点（可通过 spark.deploy.ZooKeeper.dir 属性配置，默认为 spark）的子节点 leader\_election。
  - 2) zk：连接 ZooKeeper 的客户端，类型为 CuratorFramework。
  - 3) leaderLatch：使用 ZooKeeper 进行领导选举的客户端，类型为 LeaderLatch。
  - 4) status：领导选举的状态，包括有领导（LEADER）和无领导（NOT\_LEADER）。
- 了解了 ZooKeeperLeaderElectionAgent 的属性，下面将逐一介绍它实现的方法。
- 5) start：启动基于 ZooKeeper 的领导选举代理。start 的实现如代码清单 9-19 所示。

代码清单9-19 启动ZooKeeperLeaderElectionAgent

---

```
private def start() {
  logInfo("Starting ZooKeeper LeaderElection agent")
  zk = SparkCuratorUtil.newClient(conf)
  leaderLatch = new LeaderLatch(zk, WORKING_DIR)
  leaderLatch.addListener(this)
  leaderLatch.start()
}
```

---

由于 ZooKeeperLeaderElectionAgent 实现了 LeaderLatchListener，因此发生领导选举后，LeaderLatch 会回调 ZooKeeperLeaderElectionAgent 的 isLeader 或 notLeader 方法。

**小贴士：**LeaderLatch 是 Curator 提供的进行主节点选举的 API。为了在 Curator 客户端获得或者失去管理权时进行回调处理，需要注册一个 LeaderLatchListener 接口的实现类。LeaderLatchListener 接口中定义了 isLeader 和 notLeader 两个方法，需要实现类实现。

- 6) stop：停止基于 ZooKeeper 的领导选举代理。stop 方法的实现如代码清单 9-20 所示。

代码清单9-20 停止ZooKeeperLeaderElectionAgent

---

```
override def stop() {
  leaderLatch.close()
  zk.close()
```

---

 }

7) updateLeadershipStatus：更新领导关系状态。updateLeadershipStatus 方法的实现如代码清单 9-21 所示。

代码清单9-21 更新领导关系状态

---

```
private def updateLeadershipStatus(isLeader: Boolean) {
    if (isLeader && status == LeadershipStatus.NOT_LEADER) { // 选举为领导
        status = LeadershipStatus.LEADER
        masterInstance.electedLeader()
    } else if (!isLeader && status == LeadershipStatus.LEADER) { // 撤销领导职务
        status = LeadershipStatus.NOT_LEADER
        masterInstance.revokedLeadership()
    }
}
```

---

根据代码清单 9-21，updateLeadershipStatus 方法的执行步骤如下。

①如果 Master 节点之前不是领导（状态是 NOT\_LEADER），当被选为领导（Leader）时，将其状态设置为 LEADER 并调用 masterInstance 的 electedLeader 方法将 Master 节点设置为 Leader。

②如果 Master 节点之前是领导（状态是 LEADER），当没有被选为 Leader 时，将其状态设置为 NOT\_LEADER 并调用 masterInstance 的 revokedLeadership 方法撤销 Master 节点的领导关系。

8) isLeader：告知 ZooKeeperLeaderElectionAgent 所属的 Master 节点被选为 Leader，并更新领导关系状态。当 LeaderLatch 发现 Master 被选举为 Leader 后，会调用 ZooKeeperLeader-ElectionAgent 的 isLeader 方法。isLeader 的实现如代码清单 9-22 所示。

代码清单9-22 被选为Leader

---

```
override def isLeader() {
    synchronized {
        if (!leaderLatch.hasLeadership) {
            return
        }

        logInfo("We have gained leadership")
        updateLeadershipStatus(true)
    }
}
```

---

9) notLeader：告知 ZooKeeperLeaderElectionAgent 所属的 Master 节点没有被选为 Leader，并更新领导关系状态。当 LeaderLatch 发现 Master 没有被选举为 Leader，会调用 ZooKeeperLeaderElectionAgent 的 notLeader 方法。notLeader 的实现如代码清单 9-23 所示。

代码清单9-23 没有被选为Leader

---

```
override def notLeader() {
```

```

synchronized {
    if (leaderLatch.hasLeadership) {
        return
    }

    logInfo("We have lost leadership")
    updateLeadershipStatus(false)
}
}

```

**小贴士：**LeaderLatch 实际是通过在 Master 上创建领导选举的 Znode 节点，并对 Znode 节点添加监视点来发现 Master 是否成功“竞选”的。有关更多 ZooKeeper 节点、监视器的内容请访问 <http://zookeeper.apache.org>。

## 9.6 Master 详解

Master 是 local-cluster 部署模式和 Standalone 部署模式中，整个 Spark 集群最为重要的组件之一，它的设计将直接决定整个集群的可扩展性、可用性和容错性。Master 的职责包括 Worker 的管理、Application 的管理、Driver 的管理等。Master 负责对整个集群中所有资源的统一管理和分配，它接收各个 Worker 的注册、更新状态、心跳等消息，也接收 Driver 和 Application 的注册<sup>⊖</sup>。

Worker 向 Master 注册时会携带自身的身份和资源信息（如 ID、host、port、内核数、内存大小等），这些资源将按照一定的资源调度策略分配给 Driver 或 Application。Master 给 Driver 分配了资源后，将向 Worker 发送启动 Driver 的命令，后者在接收到启动 Driver 的命令后启动 Driver。Master 给 Application 分配了资源后，将向 Worker 发送启动 Executor 的命令，后者在接收到启动 Executor 的命令后启动 Executor。

Master 接收 Worker 的状态更新消息，用于“杀死”不匹配的 Driver 或 Application。

Worker 向 Master 发送的心跳消息有两个目的：一是告知 Master 自己还“活着”，另外则是某个 Master 出现故障后，通过领导选举选择了其他 Master 负责对整个集群的管理，此时被激活的 Master 可能并没有缓存 Worker 的相关信息，因此需要告知 Worker 重新向新的 Master 注册。

本节主要对 Master 进行详细分析，理解 local-cluster 部署模式和 Standalone 部署模式下 Master 如何对整个集群的资源进行管理和分配，但在此之前先需要按部就班地了解 Master 包含的属性。

- rpcEnv：即 RpcEnv。
- address：RpcEnv 的地址（即 RpcAddress）。RpcAddress 只包含 host 和 port 两个属

<sup>⊖</sup> 由于注册 Driver 只发生在需要将 Driver 委托给 Spark 集群运行的情况下，对于 Driver 运行在客户端的情形，不会有注册 Driver 的行为。由于注册 Driver 与注册 Application 都是类似的，限于篇幅，本书只挑选注册 Application 进行介绍。

性，用来记录 Master URL 的 host 和 port。

- ❑ webUiPort：参数要指定的 WebUI 的端口。
- ❑ securityMgr：即 SecurityManager。
- ❑ conf：即 SparkConf。
- ❑ checkForWorkerTimeOutTask：检查 Worker 超时的任务。
- ❑ forwardMessageThread：包含一个线程的 ScheduledThreadPoolExecutor，启动的线程以 master-forward-message-thread 作为名称。forwardMessageThread 主要用于运行 checkForWorkerTimeOutTask 和 recoveryCompletionTask。
- ❑ hadoopConf：Hadoop 的配置。
- ❑ WORKER\_TIMEOUT\_MS：Worker 的超时时间。可通过 spark.worker.timeout 属性配置，默认为 60s。
- ❑ workers：所有注册到 Master 的 Worker 信息（WorkerInfo）的集合。
- ❑ REAPER\_ITERATIONS：从 workers 中移除处于死亡（DEAD）状态的 Worker 所对应的 WorkerInfo 的权重。可通过 spark.dead.worker.persistence 属性配置，默认为 15。
- ❑ ECOVERY\_MODE：恢复模式。可通过 spark.deploy.recoveryMode 属性配置，默认为 NONE。
- ❑ MAX\_EXECUTOR\_RETRIES：Executor 的最大重试数。可通过 spark.deploy.max-ExecutorRetries 属性配置，默认为 10。
- ❑ idToApp：Application ID 与 ApplicationInfo 的映射关系。
- ❑ waitingApps：正等待调度的 Application 所对应的 ApplicationInfo 的集合。
- ❑ apps：所有 ApplicationInfo 的集合。
- ❑ idToWorker：Worker id 与 WorkerInfo 的映射关系。
- ❑ addressToWorker：Worker 的 RpcEnv 的地址（RpcAddress）与 WorkerInfo 的映射关系。
- ❑ endpointToApp：RpcEndpointRef 与 ApplicationInfo 的映射关系。
- ❑ addressToApp：Application 对应 Driver 的 RpcEnv 的地址（RpcAddress）与 ApplicationInfo 的映射关系。
- ❑ completedApps：已经完成的 ApplicationInfo 的集合。
- ❑ nextAppNumber：下一个 Application 的号码。nextAppNumber 将参与到 Application ID 的生成规则中。
- ❑ drivers：所有 Driver 信息（DriverInfo）的集合。
- ❑ completedDrivers：已经完成的 DriverInfo 的集合。
- ❑ RETAINED\_APPLICATIONS：completedApps 中最多可以保留的 ApplicationInfo 的数量的限制大小。当 completedApps 中的 ApplicationInfo 数量大于等于 RETAINED\_APPLICATIONS 时，需要对 completedApps 中的部分 ApplicationInfo 进行清

除。可通过 `spark.deploy.retainedApplications` 属性配置，默认为 200。

- ❑ RETAINED\_DRIVERS：`completedDrivers` 中最多可以保留的 `DriverInfo` 的数量的限制大小。当 `completedDrivers` 中的 `DriverInfo` 数量大于等于 `RETAINED_DRIVERS` 时，需要对 `completedDrivers` 中的部分 `DriverInfo` 进行清除。可通过 `spark.deploy.retainedDrivers` 属性配置，默认为 200。
- ❑ `waitingDrivers`：正等待调度的 Driver 所对应的 `DriverInfo` 的集合。
- ❑ `nextDriverNumber`：下一个 Driver 的号码。
- ❑ `masterMetricsSystem`：实例名为 `master` 的 `MetricsSystem`，即 Master 的度量系统。`masterMetricsSystem` 是通过 `MetricsSystem` 的 `createMetricsSystem` 方法（已在 5.8 节详细介绍）创建的。
- ❑ `applicationMetricsSystem`：实例名为 `applications` 的 `MetricsSystem`，即应用程序的度量系统。`applicationMetricsSystem` 也是通过 `MetricsSystem` 的 `createMetricsSystem` 方法创建的。
- ❑ `masterSource`：类型为 `MasterSource`，是有关 Master 的度量来源。
- ❑ `webUi`：Master 的 WebUI，类型为 `WebUI` 的子类 `MasterWebUI`。
- ❑ `masterPublicAddress`：Master 的公开地址。可通过 Java 系统环境变量 `SPARK_PUBLIC_DNS` 配置，默认为 Master 的 `RpcEnv` 的地址。
- ❑ `masterUrl`：Master 的 Spark URL（即 `spark://host:port` 格式的地址）。
- ❑ `masterWebUiUrl`：Master 的 WebUI 的 URL。
- ❑ `state`：Master 所处的状态。Master 的状态包括支持（`STANDBY`）、激活（`ALIVE`）、恢复中（`RECOVERING`）、完成恢复（`COMPLETING_RECOVERY`）等。
- ❑ `persistenceEngine`：持久化引擎（`PersistenceEngine`）。
- ❑ `leaderElectionAgent`：领导选举代理（`LeaderElectionAgent`）。
- ❑ `recoveryCompletionTask`：当 Master 被选举为领导后，用于集群状态恢复的任务。
- ❑ `spreadOutApps`：是否允许 Application 能够在所有节点间调度。在所有节点间执行循环调度是 Spark 在实现更好的配置内存方法之前的临时解决方案，通过此方案可以避免 Application 总是固定在一小群节点上执行。可通过 `spark.deploy.spreadOut` 属性配置，默认为 `true`。
- ❑ `defaultCores`：应用程序默认的最大内核数。可通过 `spark.deploy.defaultCores` 属性配置，默认为 `java.lang.Integer.MAX_VALUE`。
- ❑ `reverseProxy`：SparkUI 是否采用反向代理。可通过 `spark.ui.reverseProxy` 属性配置，默认为 `false`。
- ❑ `restServerEnabled`：是否提供 REST 服务以提交应用程序。可通过 `spark.master.rest.enabled` 属性配置，默认为 `true`。
- ❑ `restServer`：REST 服务的实例，类型为 `StandaloneRestServer`。

□ restServerBoundPort: REST 服务绑定的端口。

有了对 Master 属性的了解，现在可以来看看 Master 的主要执行流程了。

### 9.6.1 启动 Master

启动 Master 有作为 JVM 进程内的对象启动和作为单独的进程启动的两种方式。以对象启动的方式主要用于 local-cluster 模式，而作为进程启动则用于 Standalone 模式。

#### 1. 对象方式启动

Master 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-24）用于创建 Master 对象，并将 Master 对象注册到 RpcEnv 中完成对 Master 对象的启动。

代码清单9-24 创建、启动Master

---

```
def startRpcEnvAndEndpoint(
    host: String,
    port: Int,
    webUiPort: Int,
    conf: SparkConf): (RpcEnv, Int, Option[Int]) = {
    val securityMgr = new SecurityManager(conf)
    val rpcEnv = RpcEnv.create(SYSTEM_NAME, host, port, conf, securityMgr)
    val masterEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME,
        new Master(rpcEnv, rpcEnv.address, webUiPort, securityMgr, conf))
    val portsResponse = masterEndpoint.askWithRetry[BoundPortsResponse](Bound-
        PortsRequest)
    (rpcEnv, portsResponse.webUIPort, portsResponse.restPort)
}
```

---

根据代码清单 9-24，Master 的伴生对象的 startRpcEnvAndEndpoint 方法的执行步骤如下。

- 1) 创建 SecurityManager。
- 2) 创建 RpcEnv。5.3 节已对 RpcEnv 的 create 方法介绍过，此处不再赘述。
- 3) 创建 Master，并且将 Master (Master 也继承了 ThreadSafeRpcEndpoint) 注册到刚创建的 RpcEnv 中，并获得 Master 的 RpcEndpointRef。
- 4) 通过 Master 的 RpcEndpointRef，向 Master 发送 BoundPortsRequest 消息，并获得返回的 BoundPortsResponse 消息。
- 5) 返回创建的 RpcEnv、BoundPortsResponse 消息携带的 WebUIPort、REST 服务的端口 (restPort) 等信息。

在向 RpcEnv 中注册 Master 时，会触发对 Master 的 onStart 方法（见代码清单 9-25）的调用。

代码清单9-25 Master的onStart方法

---

```
override def onStart(): Unit = {
    logInfo("Starting Spark master at " + masterUrl)
```

```

    logInfo(s"Running Spark version ${org.apache.spark.SPARK_VERSION}")
    webUi = new MasterWebUI(this, webUiPort) // 创建Master的Web UI
    webUi.bind()
    masterWebUiUrl = "http://" + masterPublicAddress + ":" + webUi.boundPort
    if (reverseProxy) {
        masterWebUiUrl = conf.get("spark.ui.reverseProxyUrl", masterWebUiUrl)
        logInfo(s"Spark Master is acting as a reverse proxy. Master, Workers and " +
            s"Applications UIs are available at $masterWebUiUrl")
    }
    checkForWorkerTimeOutTask = forwardMessageThread.scheduleAtFixedRate(new
        Runnable {
        override def run(): Unit = Utils.tryLogNonFatalError {
            self.send(CheckForWorkerTimeOut) // 检查Worker超时
        }
    }, 0, WORKER_TIMEOUT_MS, TimeUnit.MILLISECONDS)

    if (restServerEnabled) { // 创建并启动REST服务
        val port = conf.getInt("spark.master.rest.port", 6066)
        restServer = Some(new StandaloneRestServer(address.host, port, conf, self,
            masterUrl))
    }
    restServerBoundPort = restServer.map(_.start())

    masterMetricsSystem.registerSource(masterSource)
    masterMetricsSystem.start() // 启动Master的度量系统
    applicationMetricsSystem.start() // 启动Application的度量系统
    masterMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)
    applicationMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)
    // 根据RECOVERY_MODE创建持久化引擎和领导选举代理
    val serializer = new JavaSerializer(conf)
    val (persistenceEngine_, leaderElectionAgent_) = RECOVERY_MODE match {
        case "ZOOKEEPER" =>
            logInfo("Persisting recovery state to ZooKeeper")
            val zkFactory =
                new ZooKeeperRecoveryModeFactory(conf, serializer)
                (zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent(
                    this))
        case "FILESYSTEM" =>
            val fsFactory =
                new FileSystemRecoveryModeFactory(conf, serializer)
                (fsFactory.createPersistenceEngine(), fsFactory.createLeaderElectionAgent(
                    this))
        case "CUSTOM" =>
            val clazz = Utils.classForName(conf.get("spark.deploy.recoveryMode.factory"))
            val factory = clazz.getConstructor(classOf[SparkConf], classOf[Serializer])
                .newInstance(conf, serializer)
                .asInstanceOf[StandaloneRecoveryModeFactory]
                (factory.createPersistenceEngine(), factory.createLeaderElectionAgent(this))
        case _ =>
            (new BlackHolePersistenceEngine(), new MonarchyLeaderAgent(this))
    }
    persistenceEngine = persistenceEngine_
    leaderElectionAgent = leaderElectionAgent_
}

```

根据代码清单 9-25，Master 的 onStart 方法的执行步骤如下。

1) 创建 MasterWebUI，并绑定端口。由于 MasterWebUI 的原理与 4.3 节介绍的 SparkUI 类似，所以不再赘述。

2) 拼接 MasterWebUI 的 URL，即 masterWebUiUrl。

3) 如果启用了 Spark UI 的反向代理，那么将 masterWebUiUrl 设置为从 spark.ui.reverseProxyUrl 属性获得的反向代理的 URL。

4) 启动对 Worker 超时进行检查的定时任务 checkForWorkerTimeOutTask，执行的时间间隔为 WORKER\_TIMEOUT\_MS。checkForWorkerTimeOutTask 实际上是向 Master 自身发送 CheckForWorkerTimeOut 消息完成检查的。

5) 如果需要提供 REST 服务，那么创建并启动 StandaloneRestServer。StandaloneRestServer 的端口可通过 spark.master.rest.port 属性配置，默认为 6066。对 REST 服务感兴趣的读者可自行研究，本书不会对其过多介绍。

6) 将 masterSource 注册到 masterMetricsSystem 后启动 masterMetricsSystem 和 applicationMetricsSystem，然后将 masterMetricsSystem 和 applicationMetricsSystem 的 Servlet-Context-Handler 添加到 MasterWebUI。

7) 根据 RECOVERY\_MODE 创建持久化引擎和领导选举代理。如果 RECOVERY\_MODE 为 ZOOKEEPER，那么分别为 ZooKeeperPersistenceEngine 和 ZooKeeper-LeaderElectionAgent；如果 RECOVERY\_MODE 为 FILESYSTEM，那么分别为 FileSystemPersistence-Engine 和 MonarchyLeaderAgent；如果 RECOVERY\_MODE 为 CUSTOM，那么可通过 spark.deploy.recoveryMode.factory 属性配置创建持久化引擎和领导选举代理的工厂类，并创建对应的实例；如果 RECOVERY\_MODE 为其他值时，则分别为 BlackHole-Persistence-Engine 和 MonarchyLeaderAgent。

## 2. 进程方式启动

Master 的伴生对象中实现了 main 方法，这样就可以作为单独的 JVM 进程启动了。main 方法的实现如代码清单 9-26 所示。

代码清单 9-26 进程方式启动 Master

---

```
def main(argStrings: Array[String]) {
    Utils.initDaemon(log)
    val conf = new SparkConf
    val args = new MasterArguments(argStrings, conf)
    val (rpcEnv, _, _) = startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, conf)
    rpcEnv.awaitTermination()
}
```

---

根据代码清单 9-26，main 方法的执行步骤如下。

1) 创建 SparkConf。

2) 创建 MasterArguments 以对执行 main 函数时传递的参数进行解析。在解析的过程中会将 Spark 属性配置文件中以 spark. 开头的属性保存到 SparkConf。

3) 调用 Master 伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-24）创建并启动 Master 对象。

MasterArguments 用于解析系统环境变量和启动 Master 时指定的命令行参数，如代码清单 9-27 所示。

代码清单9-27 MasterArguments解析Master参数

---

```
private[master] class MasterArguments(args: Array[String], conf: SparkConf)
  extends Logging {
  var host = Utils.localHostName()
  var port = 7077
  var webUiPort = 8080
  var propertiesFile: String = null
  if (System.getenv("SPARK_MASTER_IP") != null) {
    logWarning("SPARK_MASTER_IP is deprecated, please use SPARK_MASTER_HOST")
    host = System.getenv("SPARK_MASTER_IP")
  }
  if (System.getenv("SPARK_MASTER_HOST") != null) {
    host = System.getenv("SPARK_MASTER_HOST")
  }
  if (System.getenv("SPARK_MASTER_PORT") != null) {
    port = System.getenv("SPARK_MASTER_PORT").toInt
  }
  if (System.getenv("SPARK_MASTER_WEBUI_PORT") != null) {
    webUiPort = System.getenv("SPARK_MASTER_WEBUI_PORT").toInt
  }
  parse(args.toList)
  propertiesFile = Utils.loadDefaultSparkProperties(conf, propertiesFile)
  if (conf.contains("spark.master.ui.port")) {
    webUiPort = conf.get("spark.master.ui.port").toInt
  }
}
```

---

根据代码清单 9-27，其中调用了 parse 方法来解析命令行参数，parse 的实现如下。

```
def parse(args: List[String]): Unit = args match {
  case ("--ip" | "-i") :: value :: tail =>
    Utils.checkHost(value, "ip no longer supported, please use hostname " + value)
    host = value
    parse(tail)
  case ("--host" | "-h") :: value :: tail =>
    Utils.checkHost(value, "Please use hostname " + value)
    host = value
    parse(tail)
  case ("--port" | "-p") :: IntParam(value) :: tail =>
    port = value
    parse(tail)
  case "--webui-port" :: IntParam(value) :: tail =>
    webUiPort = value
    parse(tail)
  case "--properties-file" :: value :: tail =>
```

```

propertiesFile = value
parse(tail)
case ("--help") :: tail =>
  printUsageAndExit(0)
case Nil => {}
case _ =>
  printUsageAndExit(1)
}

```

根据上面的分析，MasterArguments 中解析的参数如表 9-1 所示。

表 9-1 MasterArguments 中解析的参数

属性	含义	默认值	系统环境变量	命令行参数
host	Master 的监听地址	机器名	SPARK_MASTER_IP (已弃用) SPARK_MASTER_HOST	--ip 或者 -i (已弃用) --host 或者 -h
port	Master 的监听端口	7077	SPARK_MASTER_PORT	--port 或者 -p
webUiPort	WebUI 的监听端口	8080	SPARK_MASTER_WEBUI_PORT	--webui-port
propertiesFile	Spark 属性文件	spark-defaults.conf	—	--properties-file

**小贴士：**命令行参数指定的值会覆盖系统环境变量指定的值。属性 spark.master.ui.port 指定的值会覆盖系统环境变量 SPARK\_MASTER\_WEBUI\_PORT 或命令行参数 --webui-port 的值。

parse 函数解析的命令行参数如表 9-2 所示。

表 9-2 parse 函数解析的命令行参数

参数名	参数含义
--ip 或者 -i	指定 host name，将来会停止使用，推荐使用 --host 或者 -h
--host 或者 -h	指定 host name
--port 或者 -p	指定端口
--webui-port	指定 WebUI 的端口
--properties-file	Spark 系统属性文件
--help	帮助

### 9.6.2 检查 Worker 超时

经过上一小节对启动 Master 的分析，我们知道定时任务 checkForWorkerTimeOutTask 是以 WORKER\_TIMEOUT\_MS 为时间间隔，通过不断向 Master 自身发送 CheckForWorkerTimeOut 消息来实现对 Worker 的超时检查的。Master 也继承自 RpcEndpoint，Master 实现的 receive 方法中处理 CheckForWorkerTimeOut 消息的代码如下。

```
case CheckForWorkerTimeOut =>
    timeOutDeadWorkers()
```

所以，Master 是通过调用 timeOutDeadWorkers 方法（见代码清单 9-28）处理 CheckForWorkerTimeOut 消息的。

代码清单9-28 检查Worker超时

---

```
private def timeOutDeadWorkers() {
    val currentTime = System.currentTimeMillis() // 下面过滤出所有超时的worker
    val toRemove = workers.filter(_.lastHeartbeat < currentTime - WORKER_TIMEOUT_MS).toArray
    for (worker <- toRemove) {
        if (worker.state != WorkerState.DEAD) {
            logWarning("Removing %s because we got no heartbeat in %d seconds".format(
                worker.id, WORKER_TIMEOUT_MS / 1000))
            removeWorker(worker) // 移除Worker的相关信息
        } else {
            if (worker.lastHeartbeat < currentTime - ((REAPER_ITERATIONS + 1) * WORKER_TIMEOUT_MS)) {
                workers -= worker // 等待足够长的时间后将它从workers列表中移除
            }
        }
    }
}
```

---

根据代码清单 9-28，timeOutDeadWorkers 方法的处理步骤如下。

1) 过滤出所有超时的 Worker，即当前时间与 WORKER\_TIMEOUT\_MS 之差仍然大于 WorkerInfo 的 lastHeartbeat 的 Worker 节点。

2) 对所有超时的 Worker 进行如下处理。

① 如果 WorkerInfo 的状态不是 DEAD，则调用 removeWorker 方法（见代码清单 9-48）移除 Worker 的相关信息。

② 如果 WorkerInfo 的状态是 DEAD，则等待足够长的时间后将它从 workers 列表中移除。足够长的时间的计算公式为：REAPER\_ITERATIONS 与 1 的和再乘以 WORKER\_TIMEOUT\_MS。

经过对检查 Worker 超时的分析，可以发现定时任务 checkForWorkerTimeOutTask 与 HeartbeatReceiver 的定时任务 timeoutCheckingTask 的原理十分类似，读者可以对比学习。

### 9.6.3 被选举为领导时的处理

Master 基于高可用性的考虑，可以同时启动多个 Master。这些 Master 中只有一个激活（Active）状态的，其余的都是支持（Standby）状态。根据 9.5 节的介绍，Master 为了具备故障迁移的能力，它实现了 LeaderElectable 接口，因此当 Master 被选举为领导时，领导选举代理（LeaderElectionAgent）将会调用 Master 的 electedLeader 方法。electedLeader 方法的实现如代码清单 9-29 所示。

---

代码清单9-29 Master的electedLeader方法

---

```
override def electedLeader() {
    self.send(ElectedLeader)
}
```

---

根据代码清单 9-29，electedLeader 方法只是向 Master 自身发送了 ElectedLeader 消息。Master 的 receive 方法中实现了对 ElectedLeader 消息的处理，如代码清单 9-30 所示。

---

代码清单9-30 Master对ElectedLeader消息的处理

---

```
case ElectedLeader =>
    val (storedApps, storedDrivers, storedWorkers) = persistenceEngine.readPersisted-
        Data(rpcEnv)
    state = if (storedApps.isEmpty && storedDrivers.isEmpty && storedWorkers.
        isEmpty) {
        RecoveryState.ALIVE // 如果没有任何持久化信息，那么将Master的当前状态设置为激活
        (ALIVE)
    } else {
        RecoveryState.RECOVERING // 如果有持久化信息，那么将Master的当前状态设置为恢复中
        (RECOVERING)
    }
    logInfo("I have been elected leader! New state: " + state)
    if (state == RecoveryState.RECOVERING) {
        // 对整个集群的状态进行恢复
        beginRecovery(storedApps, storedDrivers, storedWorkers)
        recoveryCompletionTask = forwardMessageThread.schedule(new Runnable {
            override def run(): Unit = Utils.tryLogNonFatalError {
                self.send(CompleteRecovery)
            }
        }, WORKER_TIMEOUT_MS, TimeUnit.MILLISECONDS)
    }
}
```

---

根据代码清单 9-30，Master 处理 ElectedLeader 消息的步骤如下。

- 1) 从持久化引擎中读取出持久化的 ApplicationInfo、DriverInfo、WorkerInfo 等信息。
- 2) 如果没有任何持久化信息，那么将 Master 的当前状态设置为激活 (ALIVE)，否则将 Master 的当前状态设置为恢复中 (RECOVERING)。
- 3) 如果 Master 的当前状态为 RECOVERING，则调用 beginRecovery 方法（见代码清单 9-31）对整个集群的状态进行恢复。在集群状态恢复完成后，创建延时任务 recoveryCompletionTask，在常量 WORKER\_TIMEOUT\_MS 指定的时间后向 Master 自身发送 CompleteRecovery 消息。

---

代码清单9-31 恢复集群状态

---

```
private def beginRecovery(storedApps: Seq[ApplicationInfo], storedDrivers:
    Seq[DriverInfo],
    storedWorkers: Seq[WorkerInfo]) {
    for (app <- storedApps) { // 遍历从持久化引擎中读取的ApplicationInfo
        logInfo("Trying to recover app: " + app.id)
        try {
            registerApplication(app) // 注册ApplicationInfo
        }
```

```
    app.state = ApplicationState.UNKNOWN
    app.driver.send(MasterChanged(self, masterWebUiUrl)) // 让Driver切换Master后重连
} catch {
    case e: Exception => logInfo("App " + app.id + " had exception on
        reconnect")
}
}
for (driver <- storedDrivers) { // 遍历从持久化引擎中读取的DriverInfo
    drivers += driver // 注册DriverInfo
}
for (worker <- storedWorkers) { // 遍历从持久化引擎中读取的WorkerInfo
    logInfo("Trying to recover worker: " + worker.id)
    try {
        registerWorker(worker) // 注册WorkerInfo
        worker.state = WorkerState.UNKNOWN
        // 让Worker切换Master后重连
        worker.endpoint.send(MasterChanged(self, masterWebUiUrl))
    } catch {
        case e: Exception => logInfo("Worker " + worker.id + " had exception on
            reconnect")
    }
}
}
}
```

根据代码清单 9-31，beginRecovery 方法的处理步骤如下。

1) 遍历从持久化引擎中读取的 ApplicationInfo，对每个 ApplicationInfo 执行如下操作。

① 调用 registerApplication 方法（见代码清单 9-44）将 ApplicationInfo 添加到 apps、idToApp、endpointToApp、addressToApp、waitingApps 等缓存中。

② 将 ApplicationInfo 的状态设置为 UNKNOWN。

③ 向提交应用程序的 Driver 发送 MasterChanged 消息（此消息将携带被选举为领导的 Master 和此 Master 的 masterWebUiUrl 属性）。Driver 接收到 MasterChanged 消息后，将自身的 master 属性修改为当前 Master 的 RpcEndpointRef，并将 alreadyDisconnected 设置为 false，最后向 Master 发送 MasterChangeAcknowledged 消息（9.8.1 节将会对此介绍）。Master 接收到 MasterChangeAcknowledged 消息后将 ApplicationInfo 的状态修改为 WAITING，然后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

2) 遍历从持久化引擎中读取的 DriverInfo，将每个 DriverInfo 添加到 drivers 缓存。

3) 遍历从持久化引擎中读取的 WorkerInfo，对每个 WorkerInfo 执行如下操作。

① 调用 registerWorker 方法将 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中。

② 将 WorkerInfo 的状态修改为 UNKNOWN。

③ 向 Worker 发送 MasterChanged 消息（此消息将携带被选举为领导的 Master 和此 Master 的 masterWebUiUrl 属性）。在 9.7.4 节中将介绍 Worker 接收到 MasterChanged 消息

后，将自身的 activeMasterUrl、activeMasterWebUiUrl、master 等属性修改为当前 Master 的对应信息，然后将 connected 设置为 true，最后向 Master 发送 WorkerSchedulerStateResponse 消息。Master 接收到 WorkerSchedulerStateResponse 消息后，首先将 Worker-Info 的状态修改为 ALIVE，然后对此 Worker 上的 Executor 和 Driver 也进行恢复，最后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

Master 对 ElectedLeader 消息进行处理时创建的 recoveryCompletionTask 将向 Master 自身发送 CompleteRecovery 消息，Master 的 receive 方法中对 CompleteRecovery 消息处理的代码如下。

```
case CompleteRecovery => completeRecovery()
```

从上述代码可以看到，Master 接收到 CompleteRecovery 消息后只调用了 completeRecovery 方法。completeRecovery 方法的实现如代码清单 9-32 所示。

代码清单9-32 完成集群恢复

---

```
private def completeRecovery() {
    if (state != RecoveryState.RECOVERING) { return }
    state = RecoveryState.COMPLETING_RECOVERY
    workers.filter(_.state == WorkerState.UNKNOWN).foreach(removeWorker)
    apps.filter(_.state == ApplicationState.UNKNOWN).foreach(finishApplication)
    drivers.filter(_.worker.isEmpty).foreach { d =>
        logWarning(s"Driver ${d.id} was not found after master recovery")
        if (d.desc.supervise) { // 由集群监管的Driver
            logWarning(s"Re-launching ${d.id}")
            relaunchDriver(d) // 重新调度运行指定的Driver
        } else {
            removeDriver(d.id, DriverState.ERROR, None)
            logWarning(s"Did not re-launch ${d.id} because it was not supervised")
        }
    }
    state = RecoveryState.ALIVE
    schedule()
    logInfo("Recovery complete - resuming operations!")
}
```

---

根据代码清单 9-32，completeRecovery 方法的执行步骤如下。

- 1) 如果 Master 的状态不是 RECOVERING，直接返回。
- 2) 将 workers 中状态为 UNKNOWN 的所有 WorkerInfo，通过调用 removeWorker 方法从 Master 中移除。
- 3) 将 apps 中状态为 UNKNOWN 的所有 ApplicationInfo，通过调用 finishApplication 方法从 Master 中移除。
- 4) 从 drivers 中过滤出还没有分配 Worker 的所有 DriverInfo，如果 Driver 是被监管的，则调用 relaunchDriver 方法重新调度运行指定的 Driver，否则调用 removeDriver 方法移除 Master 维护的关于指定 Driver 的相关信息和状态。

- 5) 将 Master 的状态设置为 ALIVE。
- 6) 调用 schedule 方法进行资源调度。

**小贴士：**为什么状态为 UNKNOWN 的 WorkerInfo 或 ApplicationInfo 需要被移除？根据对 beginRecovery 方法的分析，Master 会向从持久化引擎中恢复的 Worker 或 Application 发送 MasterChanged 消息。Worker 或 Application 接收到 MasterChanged 消息后将向新的 Master 发送重连请求，Master 接收到重连请求后就知道 Worker 或 Application 还“活着”，于是更改它们的状态。对于已经停止或发生故障的 Worker 或 Application，Master 接收不到它们的重连请求，因此它们的状态依然是 UNKNOWN。

#### 9.6.4 一级资源调度

Master 的资源调度是 Spark 的一级资源调度，分为对 Driver 的资源调度和对 Executor 的资源调度。schedule 方法是资源调度的入口方法，下面将通过它来深入分析 Master 的资源调度。

##### 1. Driver 的资源调度

Master 的 schedule 方法主要完成对 Driver 的资源调度，其实现如代码清单 9-33 所示。

代码清单9-33 Driver的资源调度

---

```

private def schedule(): Unit = {
    if (state != RecoveryState.ALIVE) {
        return
    }
    val shuffledAliveWorkers = Random.shuffle(workers.toSeq.filter(_.state == Worker-
        State.ALIVE))
    val numWorkersAlive = shuffledAliveWorkers.size
    var curPos = 0
    for (driver <- waitingDrivers.toList) { // 遍历waitingDrivers中的DriverInfo
        var launched = false
        var numWorkersVisited = 0
        while (numWorkersVisited < numWorkersAlive && !launched) {
            val worker = shuffledAliveWorkers(curPos)
            numWorkersVisited += 1
            if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.
                desc.cores) {
                launchDriver(worker, driver) // 运行Driver
                waitingDrivers -= driver
                launched = true
            }
            curPos = (curPos + 1) % numWorkersAlive
        }
    }
    startExecutorsOnWorkers() // 在Worker上启动Executor
}

```

---

根据代码清单 9-33，schedule 方法的执行步骤如下。

- 1) 如果 Master 的状态不是 ALIVE, 那么直接返回。
- 2) 过滤出 workers 中缓存的状态为 ALIVE 的 WorkerInfo, 并随机洗牌, 以避免 Driver 总是分配给小部分的 Worker。
- 3) 遍历 waitingDrivers 中的 DriverInfo, 对每个 DriverInfo 执行以下操作。
  - ① 从第 2) 步筛选的 WorkerInfo 中, 挑出内存大小和内核数都满足 Driver 需要的。
  - ② 调用 launchDriver 方法 (见代码清单 9-34) 运行 Driver。
  - ③ 将 DriverInfo 从 waitingDrivers 中移除。
- 4) 调用 startExecutorsOnWorkers 方法 (见代码清单 9-35) 在 Worker 上启动 Executor。

**注意** Driver 一般运行在客户端, 并不在 Spark 集群内。只有执行 Client.main 函数, 将 org.apache.spark.deploy.ClientEndpoint 实例注册到 RpcEnv 中时, 会构造 Driver 描述信息 (DriverDescription) 对象, 并向 Master 发送 RequestSubmitDriver 消息, 把 DriverDescription 对象传递给 Master, Master 将会根据 DriverDescription 的内容封装 DriverInfo 对象并注册到 waitingDrivers 中。

为使读者更容易理解 schedule 方法的实现, 这里假设 Driver 所需的内存是 8 (为简便起见, 笔者选用简单的数字表示), CPU 内核数是 2, workers 中一共缓存了四个 WorkerInfo, 那么对此 Driver 的资源调度如图 9-6 所示。

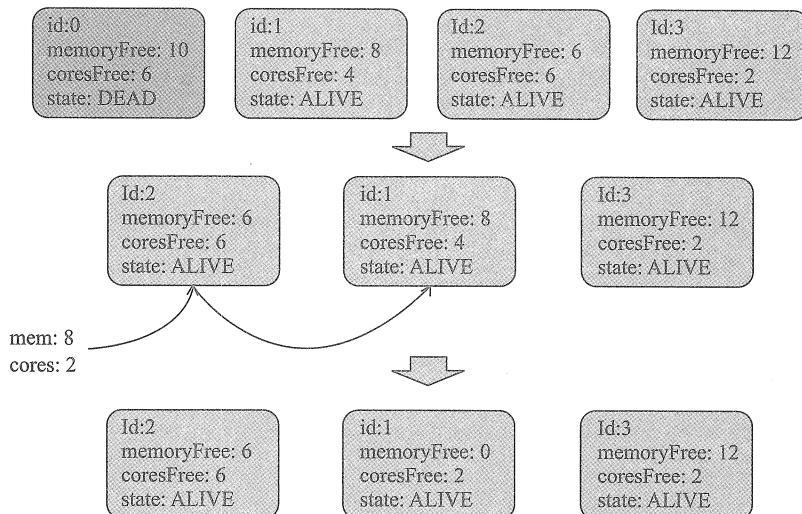


图 9-6 Driver 的资源调度过程示意图

这里对图 9-6 中的内容进行简单介绍: workers 中 id 为 0 的 WorkerInfo 的状态是 DEAD, 因此将被过滤。在随机洗牌之后, 假设 id 为 1 和 id 为 2 的 WorkerInfo 顺序上发生了互换, id 为 3 的还在最后。由于 id 为 2 的 WorkerInfo 的空闲内存不满足 Driver 的需求,

因此迭代查找下一个 WorkerInfo。id 为 1 的 WorkerInfo 的空闲内存和空闲 CPU 内核都满足 Driver 的需求，因此将在 id 为 1 的 Worker 上运行 Driver。由于 id 为 1 的 WorkerInfo 的内存和内核数分配给了 Driver，因此 id 为 1 的 WorkerInfo 剩余的空闲内存是 0，剩余的内核数是 2。

假设这时需要给另一个 Driver 进行资源调度，这个 Driver 所需的内存是 8，CPU 内核数是 4，那么资源调度过程可能如图 9-7 所示。

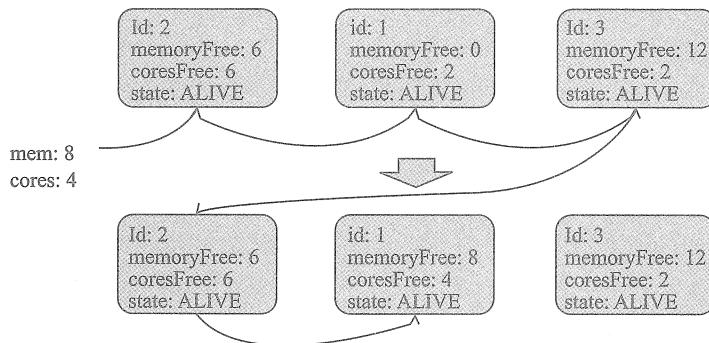


图 9-7 Driver 的资源调度过程示意图

图 9-7 中在第一次遍历过程中，发现所有可用的 WorkerInfo 都不满足 Driver 的资源需求，那么将进入下一次遍历。在第二次遍历之前，假设图 9-6 中的 Driver 已经释放了占用的资源给 id 为 1 的 WorkerInfo，那么 id 为 1 的 WorkerInfo 将满足资源的需要。

## 2. Driver 的运行

Master 的 launchDriver 方法用于运行 Driver，其实现如代码清单 9-34 所示。

代码清单9-34 运行Driver

---

```

private def launchDriver(worker: WorkerInfo, driver: DriverInfo) {
    logInfo("Launching driver " + driver.id + " on worker " + worker.id)
    worker.addDriver(driver)
    driver.worker = Some(worker)
    worker.endpoint.send(LaunchDriver(driver.id, driver.desc))
    driver.state = DriverState.RUNNING
}
  
```

---

根据代码清单 9-34，launchDriver 方法的执行步骤如下。

- 1) 在 WorkerInfo 和 DriverInfo 之间建立关系，表示 Driver 被调度到 Worker 上运行。
- 2) 向 Worker 发送 LaunchDriver 消息，Worker 接收到 LaunchDriver 消息后将运行 Driver（将在 9.7.5 节详细介绍）。
- 3) 将 DriverInfo 的状态修改为 RUNNING。

## 3. Executor 的资源调度

schedule 方法的最后调用了 startExecutorsOnWorkers 方法，进而引发对 Application 的

Executor 资源的调度。startExecutorsOnWorkers 方法的实现如代码清单 9-35 所示。

代码清单9-35 Executor的资源调度

---

```
private def startExecutorsOnWorkers(): Unit = {
    for (app <- waitingApps if app.coresLeft > 0) {
        val coresPerExecutor: Option[Int] = app.desc.coresPerExecutor //获取每个
            Executor使用的内核数
        val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
            .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&
                worker.coresFree >= coresPerExecutor.getOrElse(1))
            .sortBy(_.coresFree).reverse
        val assignedCores = scheduleExecutorsOnWorkers(app, usableWorkers, spread-
            OutApps)
        for (pos <- 0 until usableWorkers.length if assignedCores(pos) > 0) {
            allocateWorkerResourceToExecutors(
                app, assignedCores(pos), coresPerExecutor, usableWorkers(pos))
        }
    }
}
```

---

根据代码清单 9-35，startExecutorsOnWorkers 方法遍历在 waitingApps 中还需要内核的 ApplicationInfo，并对其进行如下操作。

- 1) 获取 Application 要求的每个 Executor 使用的内核数 coresPerExecutor。
- 2) 找出 workers 中状态为 ALIVE、空闲空间满足 Application 要求的每个 Executor 使用的内存大小、空闲内核数满足 coresPerExecutor 的所有 WorkerInfo，并对这些 WorkerInfo 按照空闲内核数倒序排列，这样可以优先将应用分配给内核资源充足的 Worker。
- 3) 调用 scheduleExecutorsOnWorkers 方法（见代码清单 9-36）在 Worker 上进行 Executor 的调度。scheduleExecutorsOnWorkers 方法返回在各个 Worker 上分配的内核数。
- 4) 调用 allocateWorkerResourceToExecutors 方法（见代码清单 9-37）将 Worker 上的资源分配给 Executor。

代码清单9-36 scheduleExecutorsOnWorkers的实现

---

```
private def scheduleExecutorsOnWorkers(
    app: ApplicationInfo,
    usableWorkers: Array[WorkerInfo],
    spreadOutApps: Boolean): Array[Int] = {
    val coresPerExecutor = app.desc.coresPerExecutor
    val minCoresPerExecutor = coresPerExecutor.getOrElse(1)
    val oneExecutorPerWorker = coresPerExecutor.isEmpty
    val memoryPerExecutor = app.desc.memoryPerExecutorMB
    val numUsable = usableWorkers.length
    val assignedCores = new Array[Int](numUsable)
    val assignedExecutors = new Array[Int](numUsable)
    var coresToAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)

    def canLaunchExecutor(pos: Int): Boolean = {
        val keepScheduling = coresToAssign >= minCoresPerExecutor
        val enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) >= min-
```

---

```

CoresPerExecutor

val launchingNewExecutor = !oneExecutorPerWorker || assignedExecutors(pos) == 0
if (launchingNewExecutor) {
    val assignedMemory = assignedExecutors(pos) * memoryPerExecutor
    val enoughMemory = usableWorkers(pos).memoryFree - assignedMemory >= memory-
        PerExecutor
    val underLimit = assignedExecutors.sum + app.executors.size < app.executor-
        Limit
    keepScheduling && enoughCores && enoughMemory && underLimit
} else {
    keepScheduling && enoughCores
}
}

// 获取所有可以运行Executor的Worker的索引
var freeWorkers = (0 until numUsable).filter(canLaunchExecutor)
while (freeWorkers.nonEmpty) {
    freeWorkers.foreach { pos =>
        var keepScheduling = true
        while (keepScheduling && canLaunchExecutor(pos)) { // 分配内核、Executor
            coresToAssign -= minCoresPerExecutor
            assignedCores(pos) += minCoresPerExecutor

            if (oneExecutorPerWorker) {
                assignedExecutors(pos) = 1
            } else {
                assignedExecutors(pos) += 1
            }

            if (spreadOutApps) {
                keepScheduling = false
            }
        }
    }
    freeWorkers = freeWorkers.filter(canLaunchExecutor)
}
assignedCores
}

```

---

在正式介绍 scheduleExecutorsOnWorkers 方法前，先对代码清单 9-36 中的一些变量进行说明。

- ❑ coresPerExecutor: Application 要求的每个 Executor 所需的内核数。
- ❑ minCoresPerExecutor: Application 要求的每个 Executor 所需的最小内核数。
- ❑ oneExecutorPerWorker: 是否在每个 Worker 上只分配一个 Executor。当 Application 没有配置 coresPerExecutor 时，oneExecutorPerWorker 为 true。
- ❑ memoryPerExecutor: Application 要求的每个 Executor 所需的内存大小（单位为字节）。
- ❑ usableWorkers: 缓存从 workers 中选出的状态为 ALIVE、空闲空间满足 ApplicationInfo 要求的每个 Executor 使用的内存大小、空闲内核数满足 coresPerExecutor 的所有 WorkerInfo。

- numUsable：可用的 Worker 的数量，即 usableWorkers 的大小。
- assignedCores：用于保存每个 Worker 给 Application 分配的内核数的数组。通过数组索引与 usableWorkers 中的 WorkerInfo 相对应。
- assignedExecutors：用于保存每个 Worker 给应用分配的 Executor 数的数组。通过数组索引与 usableWorkers 中的 WorkerInfo 相对应。
- coresToAssign：给 Application 要分配的内核数。coresToAssign 取 usableWorkers 中所有 WorkerInfo 的空闲内核数之和与 Application 还需的内核数中的最小值。

在 scheduleExecutorsOnWorkers 中还定义了一个名为 canLaunchExecutor 方法，用于判断 usableWorkers 中索引为参数 pos 指定的位置上的 WorkerInfo 是否能运行 Executor。判断的依据如下。

- coresToAssign  $\geq$  minCoresPerExecutor。
- usableWorkers(pos).coresFree-assignedCores(pos)  $\geq$  minCoresPerExecutor。

特别的，当 oneExecutorPerWorker 为 false 或 usableWorkers 中索引为参数 pos 指定的位置上的 WorkerInfo 还没有给 Application 分配任何 Executor，那么还需要以下依据。

- usableWorkers(pos).memoryFree  $\geq$  (assignedExecutors(pos) + 1) \* memoryPerExecutor。
- assignedExecutors.sum + app.executors.size < app.executorLimit。

有了以上的铺垫，现在来一起分析 scheduleExecutorsOnWorkers 方法的执行步骤。

- 1) 获取所有可以运行 Executor 的 Worker 的索引 freeWorkers。
- 2) 当 freeWorkers 不为空，那么遍历 freeWorkers 中的每一个索引位置，并进行如下操作。
  - ① 由于给 Application 分配 Executor 时，每个 Executor 都至少需要 minCoresPerExecutor 指定大小的内核，因此将 coresToAssign 减去 minCoresPerExecutor 的大小。
  - ② 将 usableWorkers 的 pos 位置的 WorkerInfo 已经分配的内核数（即 assignedCores 的 pos 位置的数字）增加 minCoresPerExecutor 的大小。
  - ③ 如果 oneExecutorPerWorker 为 true，则将 assignedExecutors 的索引为 pos 的值设置为 1，否则将 assignedExecutors 的索引为 pos 的值增 1。

④ 如果 spreadOutApps 为 true，则将 keepScheduling 设置为 false，这会导致对 pos 位置上的 WorkerInfo 的资源调度提前结束，那么应用需要的其他 Executor 资源将会在其他 WorkerInfo 上调度。如果 spreadOutApps 为 false，那么应用需要的 Executor 资源将会不断从 pos 位置的 WorkerInfo 上调度，直到 pos 位置的 WorkerInfo 上的的资源被使用完。

- 3) 返回 assignedCores。

为使读者更容易理解 startExecutorsOnWorkers 方法的实现，假设满足过滤条件并且已经按照内核数排序的 Worker 为 Worker0、Worker1、Worker2，Application 还需要的内核数 (coresLeft) 等于 7。Worker0 的内核数为 4，Worker1 的内核数为 4，Worker2 的内核数为 2。那么此时需要分配的内核数 coresToAssign 等于  $\min(7, (4 + 4 + 2))$ ，即 coresToAssign 为 7。在 spreadOutApps 为 true 的情况下，此时的内核分配可以用图 9-8 表示。

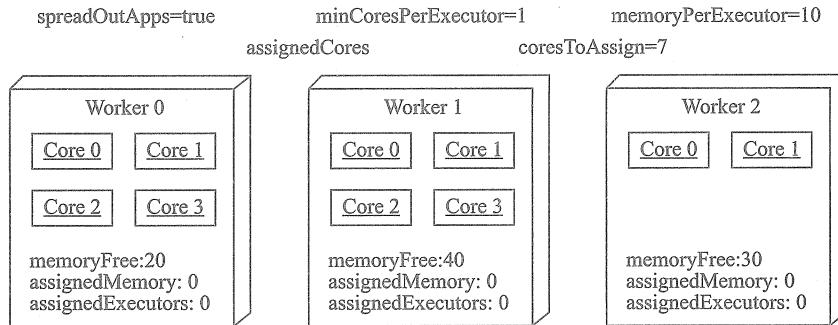


图 9-8 给应用分配 Executor 资源前

由于图 9-8 中的各个 Worker 都有足够的资源运行新的 Executor，所以在进行第一轮资源分配后的情况如图 9-9 所示。

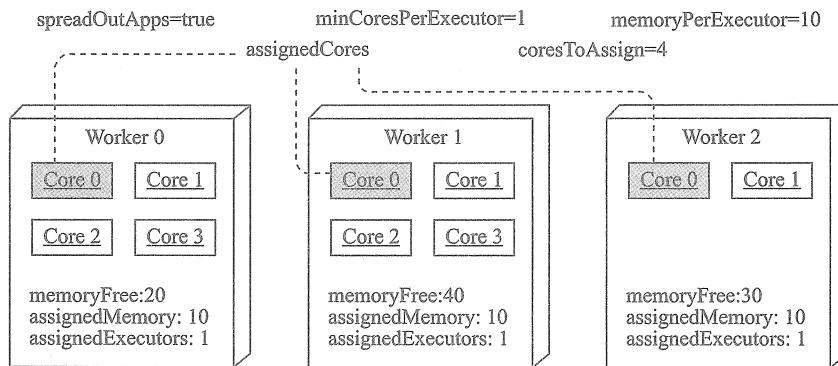


图 9-9 第一轮 Executor 资源分配后

由于图 9-9 中的各个 Worker 仍然都有足够的资源运行新的 Executor，所以在进行第二轮资源分配后的情况如图 9-10 所示。

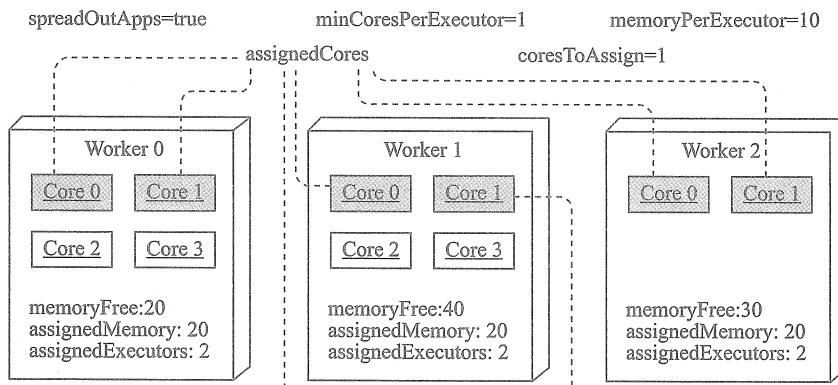


图 9-10 第二轮 Executor 资源分配后

此时，图 9-10 中的 Worker 0 没有了可用内存，Worker 2 则没有可用的内核，它们都没有充足的资源用来运行新的 Executor，所以在进行第三轮资源分配后的情况如图 9-11 所示。

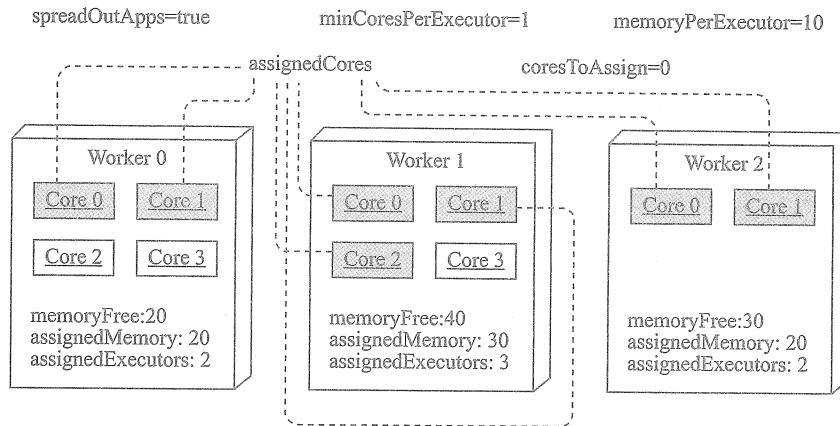


图 9-11 第三轮 Executor 资源分配后

图 9-11 显示，在第三轮 Executor 资源分配后，应用所需要的 7 个内核被分配完毕，不再需要从三个 Worker 上继续分配 Executor 了，应用的资源调度结束。

现在我们回到最开始的假设，在其他条件都不变的情况下，spreadOutApps 为 false，此时的内核分配可以用图 9-12 表示。

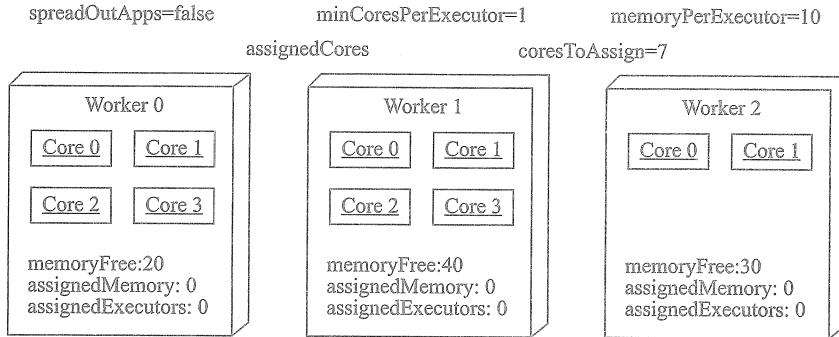


图 9-12 给应用分配 Executor 资源前

由于 spreadOutApps 为 false，所以将优先在 Worker 0 上分配 Executor 资源，分配之后如图 9-13 所示。

由于 spreadOutApps 为 false，将优先在 Worker 1 上分配 Executor 资源，分配之后如图 9-14 所示。

由于 spreadOutApps 为 false，将优先在 Worker 2 上分配 Executor 资源，分配之后如图 9-15 所示。

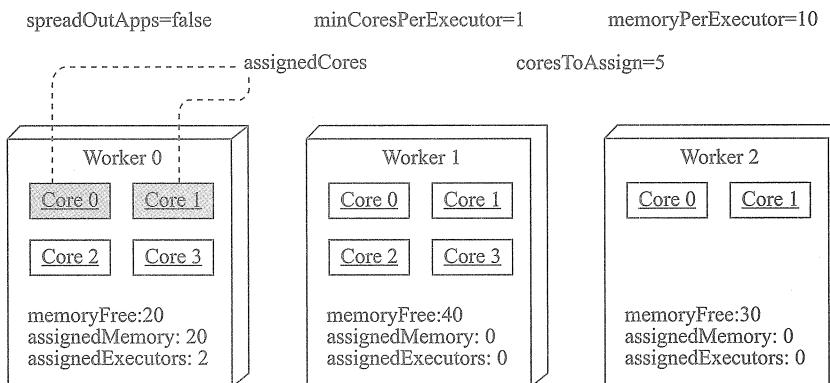


图 9-13 在 Worker 0 上分配 Executor 资源后

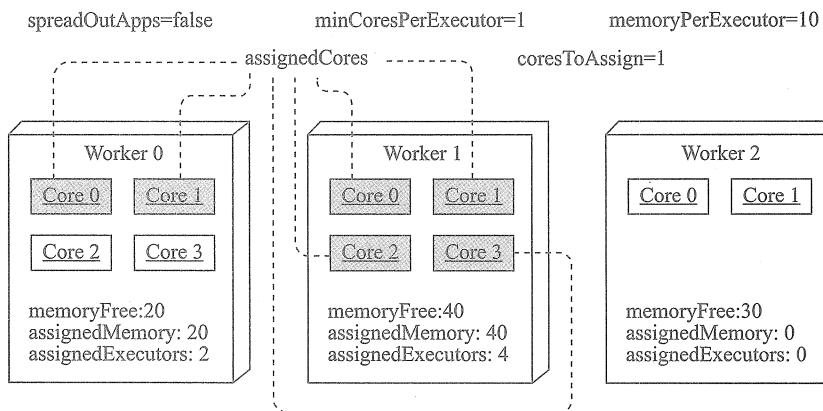


图 9-14 在 Worker 1 上分配 Executor 资源后

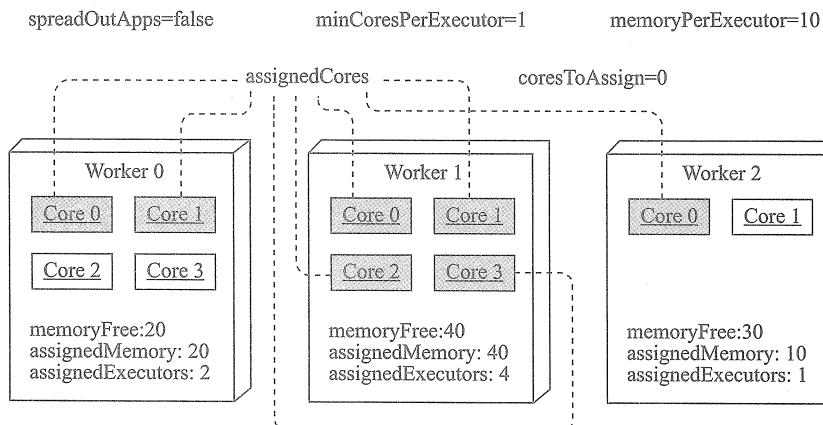


图 9-15 在 Worker 2 上分配 Executor 资源后

如图 9-15 所示，应用所需要的 7 个内核被分配完毕，不再需要从 3 个 Worker 上继续分配 Executor 了，应用的资源调度结束。

#### 4. Executor 的运行

Master 的 `allocateWorkerResourceToExecutors` 方法用于将 Worker 的资源分配给 Executor，并运行 Executor，其实现如代码清单 9-37 所示。

代码清单9-37 将Worker的资源分配给Executor并运行

---

```
private def allocateWorkerResourceToExecutors(
    app: ApplicationInfo,
    assignedCores: Int,
    coresPerExecutor: Option[Int],
    worker: WorkerInfo): Unit = {
    val numExecutors = coresPerExecutor.map { assignedCores / _ }.getOrElse(1)
    val coresToAssign = coresPerExecutor.getOrElse(assignedCores)
    for (i <- 1 to numExecutors) {
        val exec = app.addExecutor(worker, coresToAssign)
        launchExecutor(worker, exec)
        app.state = ApplicationState.RUNNING
    }
}
```

---

根据代码清单 9-37，`allocateWorkerResourceToExecutors` 方法的执行步骤如下。

- 1) 根据 Worker 分配给 Application 的内核数（`assignedCores`）与每个 Executor 需要的内核数（`coresPerExecutor`），计算在 Worker 上要运行的 Executor 数量（`numExecutors`）。如果没有指定 `coresPerExecutor`，说明 `assignedCores` 指定的所有内核都由一个 Executor 使用。
- 2) 计算给 Executor 分配的内核数（`coresToAssign`）。如果指定了 `coresPerExecutor`，那么 `coresToAssign` 等于 `coresPerExecutor`，否则 `coresToAssign` 等于 `assignedCores`（即将所有内核分配给一个 Executor 使用）。
- 3) 按照 `numExecutors` 的值，多次调用 `launchExecutor` 方法（见代码清单 9-38），在 Worker 上创建、运行 Executor，并将 `ApplicationInfo` 的状态设置为 `RUNNING`。

代码清单9-38 launchExecutor的实现

---

```
private def launchExecutor(worker: WorkerInfo, exec: ExecutorDesc): Unit = {
    logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
    worker.addExecutor(exec)
    worker.endpoint.send(LaunchExecutor(masterUrl,
        exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory))
    exec.application.driver.send(
        ExecutorAdded(exec.id, worker.id, worker.hostPort, exec.cores, exec.memory))
}
```

---

根据代码清单 9-38，`launchExecutor` 方法的执行步骤如下。

- 1) 调用 `WorkerInfo` 的 `addExecutor` 方法添加新的 Executor 的信息。
- 2) 向 Worker 发送 `LaunchExecutor` 消息（此消息携带着 masterUrl、Application 的 ID、Executor 的 ID、Application 的描述信息 ApplicationDescription、Executor 分配获得的内核数、Executor 分配获得的内存大小等信息）。Worker 对 `LaunchExecutor` 消息的处理将在

### 9.7.6 节详细分析。

3) 向提交应用的 Driver 发送 ExecutorAdded 消息（此消息携带着 Executor 的 ID、Worker 的 ID、Worker 的 host 和 port、Executor 分配获得的内核数、Executor 分配获得的内存大小等信息）。ClientEndpoint 接收到 ExecutorAdded 消息后，将调用 StandaloneAppClientListener 的 executorAdded 方法。由于 StandaloneAppClientListener 只有 StandaloneSchedulerBackend 这一个实现类，因此实际上调用了 StandaloneSchedulerBackend 的 executorAdded 方法。StandaloneSchedulerBackend 的 executorAdded 方法只是打印 ExecutorAdded 消息携带的内容，所以不再展示其实现。

### 9.6.5 注册 Worker

在 Spark 集群中，Master 接收到 Driver 提交的应用程序后，需要根据应用的资源需求，将应用分配到 Worker 上去运行。一个集群刚开始的时候只有 Master，为了让后续启动的 Worker 加入到 Master 的集群中，每个 Worker 都需要在启动的时候向 Master 注册，Master 接收到 Worker 的注册信息后，将把 Worker 的各种重要信息（如 ID、host、port、内核数、内存大小等信息）缓存起来，以便进行资源的分配与调度。Master 为了容灾，还将 Worker 的信息通过持久化引擎进行持久化，以便经过领导选举出的新 Master 能够将集群的状态从错误或灾难中恢复。

Master 的 receiveAndReply 方法中实现了对 Worker 发送的 RegisterWorker 消息进行处理的实现，如代码清单 9-39 所示。

代码清单9-39 Master对RegisterWorker消息的处理

---

```

case RegisterWorker(
    id, workerHost, workerPort, workerRef, cores, memory, workerWebUiUrl) =>
  logInfo("Registering worker %s:%d with %d cores, %s RAM".format(
    workerHost, workerPort, cores, Utils.megabytesToString(memory)))
  if (state == RecoveryState.STANDBY) {
    context.reply(MasterInStandby)
  } else if (idToWorker.contains(id)) { // Worker重复注册
    context.reply(RegisterWorkerFailed("Duplicate worker ID"))
  } else {
    val worker = new WorkerInfo(id, workerHost, workerPort, cores, memory,
      workerRef, workerWebUiUrl) // 创建WorkerInfo
    if (registerWorker(worker)) { // 注册WorkerInfo
      persistenceEngine.addWorker(worker) // 持久化引擎对WorkerInfo持久化
      context.reply(RegisteredWorker(self, masterWebUiUrl))
      schedule()
    } else {
      val workerAddress = worker.endpoint.address
      logWarning("Worker registration failed. Attempted to re-register worker
        at same " +
        "address: " + workerAddress)
      context.reply(RegisterWorkerFailed("Attempted to re-register worker at
        same address: " +
        + workerAddress))
    }
  }
}

```

```

    }
}

```

根据代码清单 9-39，Master 处理 RegisterWorker 消息的步骤如下。

- 1) 如果 Master 的状态是 STANDBY，那么向 Worker 回复 MasterInStandby 消息。
- 2) 如果 idToWorker 中已经有了要注册的 Worker 的信息，那么说明 Worker 重复注册，Master 向 Worker 回复 RegisterWorkerFailed 消息。
- 3) 如果 Master 的状态不是 STANDBY，且 idToWorker 中不包含要注册的 Worker 的信息，则执行如下操作。

① 利用 RegisterWorker 消息携带的信息创建 WorkerInfo。

② 调用 registerWorker 方法（见代码清单 9-40）注册 WorkerInfo。如果注册成功，那么对 WorkerInfo 进行持久化并向 Worker 回复 RegisteredWorker 消息，最后调用 schedule 方法（见代码清单 9-33）进行资源的调度。如果注册失败，则向 Worker 回复 RegisterWorkerFailed 消息。

registerWorker 方法用于将 WorkerInfo 注册到 Master 中，其实现如代码清单 9-40 所示。

代码清单9-40 注册WorkerInfo

---

```

private def registerWorker(worker: WorkerInfo): Boolean = {
  workers.filter { w =>
    (w.host == worker.host && w.port == worker.port) && (w.state == WorkerState.DEAD)
  }.foreach { w =>
    workers -= w
  }
  val workerAddress = worker.endpoint.address
  if (addressToWorker.contains(workerAddress)) {
    val oldWorker = addressToWorker(workerAddress)
    if (oldWorker.state == WorkerState.UNKNOWN) {
      removeWorker(oldWorker)
    } else {
      logInfo("Attempted to re-register worker at same address: " + workerAddress)
      return false
    }
  }
  workers += worker
  idToWorker(worker.id) = worker
  addressToWorker(workerAddress) = worker
  if (reverseProxy) {
    webUi.addProxyTargets(worker.id, worker.webUiAddress)
  }
  true
}

```

---

根据代码清单 9-40，registerWorker 方法的执行步骤如下。

- 1) 从 workers 中移除 host 和 port 与要注册的 WorkerInfo 的 host 和 port 一样，且状态为 DEAD 的 WorkerInfo。

2) 如果 addressToWorker 中包含地址相同的 WorkerInfo，并且此 WorkerInfo 的状态为 UNKNOWN，那么调用 removeWorker 方法（见代码清单 9-48），移除此 WorkerInfo 的相关状态，否则返回 false。

3) 将要注册的 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中。

4) 对 WebUI 的反向代理进行设置。

### 9.6.6 更新 Worker 的最新状态

Worker 在向 Master 注册成功后，会向 Master 发送 WorkerLatestState 消息。WorkerLatestState 消息将携带 Worker 的身份标识、Worker 节点的所有 Executor 的描述信息、调度到当前 Worker 的所有 Driver 的身份标识。Master 接收到 WorkerLatestState 消息的处理如下。

```
case WorkerLatestState(workerId, executors, driverIds) =>
  idToWorker.get(workerId) match {
    case Some(worker) =>
      for (exec <- executors) {
        val executorMatches = worker.executors.exists {
          case (_, e) => e.application.id == exec.appId && e.id == exec.execId
        }
        if (!executorMatches) { // “杀死”不匹配的Executor
          worker.endpoint.send(KillExecutor(masterUrl, exec.appId, exec.execId))
        }
      }

      for (driverId <- driverIds) {
        val driverMatches = worker.drivers.exists { case (id, _) => id == driverId }
        if (!driverMatches) {
          worker.endpoint.send(KillDriver(driverId)) // “杀死”不匹配的Driver
        }
      }
    case None =>
      logWarning("Worker state from unknown worker: " + workerId)
  }
```

根据上述代码，Master 处理 WorkerLatestState 消息的步骤如下。

1) 根据 Worker 的身份标识，从 idToWorker 取出注册的 WorkerInfo。

2) 遍历 executors 中的每个 ExecutorDescription，与 WorkerInfo 的 executors 中保存的 ExecutorDesc 按照应用 ID 和 Executor ID 进行匹配。对于匹配不成功的，向 Worker 发送 KillExecutor 消息以“杀死” Executor。

3) 遍历 driverIds 中的每个 Driver ID，与 WorkerInfo 的 drivers 中保存的 Driver ID 进行匹配。对于匹配不成功的，向 Worker 发送 KillDriver 消息以“杀死” Driver。

### 9.6.7 处理 Worker 的心跳

向 Master 注册 Worker，可以让 Master 知道 Worker 的资源配置，进而通过资源调度使得

Driver 及 Executor 可以在 Worker 上执行。如果 Worker 的 JVM 进程发生了崩溃或者 Worker 所在的机器宕机或网络不通，那么 Master 所维护的关于 Worker 的注册信息将变得不可用。为了让 Master 及时得知 Worker 的最新状态，Worker 需要向 Master 发送心跳，Master 将根据 Worker 的心跳更新 Worker 的最后心跳时间，以便为整个集群的健康工作提供参考。Master 的 receive 方法中实现了对 Worker 的心跳的处理，如代码清单 9-41 所示。

代码清单9-41 处理Worker的心跳

---

```

case Heartbeat(workerId, worker) =>
  idToWorker.get(workerId) match {
    case Some(workerInfo) =>
      workerInfo.lastHeartbeat = System.currentTimeMillis()
    case None =>
      if (workers.map(_.id).contains(workerId)) {
        logWarning(s"Got heartbeat from unregistered worker $workerId." +
          " Asking it to re-register.")
        worker.send(ReconnectWorker(masterUrl))
      } else {
        logWarning(s"Got heartbeat from unregistered worker $workerId." +
          " This worker was never registered, so ignoring the heartbeat.")
      }
  }
}

```

---

根据代码清单 9-41，Master 接收到 Heartbeat 消息后，将从 idToWorker 中找出缓存的 WorkerInfo，并将 WorkerInfo 的最后心跳时间（lastHeartbeat）更新为系统当前时间的时间戳。如果 idToWorker 中没有缓存的 WorkerInfo，且 workers 中有对应的 WorkerInfo（这说明定时任务 checkForWorkerTimeOutTask 检查到 Worker 超时，但是 WorkerInfo 的状态不是 DEAD，那么在调用 removeWorker 方法时将 WorkerInfo 从 idToWorker 中清除，此时的 workers 中仍然持有 WorkerInfo），那么向 Worker 发送 ReconnectWorker 消息。如果 idToWorker 中没有缓存的 WorkerInfo，且 workers 中也没有对应的 WorkerInfo，那么说明 checkForWorkerTimeOutTask 已经发现 Worker 很长时间没有心跳，并且 WorkerInfo 的状态为 DEAD 后，将 WorkerInfo 从 workers 中也移除了。

## 9.6.8 注册 Application

Driver 在启动后需要向 Master 注册 Application 信息，这样 Master 就能将 Worker 上的资源及 Executor 分配给 Driver。Driver 通过向 Master 发送 RegisterApplication 消息来注册 Application 信息，Master 因而也实现了对 RegisterApplication 消息的处理，如代码清单 9-42 所示。

代码清单9-42 注册Application

---

```

case RegisterApplication(description, driver) =>
  if (state == RecoveryState.STANDBY) {
    // ignore, don't send response
  } else {

```

---

```

    logInfo("Registering app " + description.name)
    val app = createApplication(description, driver) // 创建ApplicationInfo
    registerApplication(app) // 注册ApplicationInfo
    logInfo("Registered app " + description.name + " with ID " + app.id)
    persistenceEngine.addApplication(app) // 对ApplicationInfo持久化
    driver.send(RegisteredApplication(app.id, self))
    schedule()
}

```

---

根据代码清单 9-42，Master 处理 RegisterApplication 消息的步骤如下。

- 1) 如果 Master 当前的状态是 STANDBY，那么不作任何处理。
- 2) 如果 Master 的状态不是 STANDBY，则执行如下操作。
  - ① 利用 RegisterApplication 消息携带的信息调用 createApplication 方法（见代码清单 9-43）创建 ApplicationInfo。createApplication 方法中会给 Application 分配 ID。此 ID 的生成规则为 app-\$\{yyyyMMddHHmmss\}-\$\{nextAppNumber\}。
  - ② 调用 registerApplication 方法（见代码清单 9-44）注册 ApplicationInfo。
  - ③ 调用 PersistenceEngine 的 addApplication 方法对 ApplicationInfo 持久化。
  - ④ 向 ClientEndpoint 发送 RegisteredApplication 消息（此消息将携带为 Application 生成的 ID 和 Master 自身的 RpcEndpointRef），以表示注册 Application 信息成功。
  - ⑤ 调用 schedule 方法进行资源调度。

代码清单9-43 创建ApplicationInfo

---

```

private def createApplication(desc: ApplicationDescription, driver: RpcEndpointRef):
    ApplicationInfo = {
        val now = System.currentTimeMillis()
        val date = new Date(now)
        val appId = newApplicationId(date)
        new ApplicationInfo(now, appId, desc, date, driver, defaultCores)
    }

```

---

代码清单9-44 注册ApplicationInfo

---

```

private def registerApplication(app: ApplicationInfo): Unit = {
    val appAddress = app.driver.address
    if (addressToApp.contains(appAddress)) {
        logInfo("Attempted to re-register application at same address: " + appAddress)
        return
    }

    applicationMetricsSystem.registerSource(app.appSource)
    apps += app
    idToApp(app.id) = app
    endpointToApp(app.driver) = app
    addressToApp(appAddress) = app
    waitingApps += app
    if (reverseProxy) {
        webUi.addProxyTargets(app.id, app.desc.appUiUrl)
    }
}

```

---

### 9.6.9 处理 Executor 的申请

如果 Driver 启用了 ExecutorAllocationManager，那么 ExecutorAllocationManager 将通过 StandaloneAppClient（将在 9.8 节介绍）中的 ClientEndpoint（将在 9.8.1 节介绍）向 Master 发送 RequestExecutors 消息请求 Executor。Master 接收并处理 RequestExecutors 消息的代码如下。

```
case RequestExecutors(appId, requestedTotal) =>
context.reply(handleRequestExecutors(appId, requestedTotal))
```

Master 通过 handleRequestExecutors 方法处理 RequestExecutors 消息，其实现如下。

```
private def handleRequestExecutors(appId: String, requestedTotal: Int): Boolean = {
  idToApp.get(appId) match {
    case Some(appInfo) =>
      logInfo(s"Application $appId requested to set total executors to $requested-
      Total.")
      appInfo.executorLimit = requestedTotal
      schedule()
      true
    case None =>
      logWarning(s"Unknown application $appId requested $requestedTotal total
      executors.")
      false
  }
}
```

handleRequestExecutors 方法首先更改 ApplicationInfo 的 Executor 总数，然后调用 schedule 方法进行资源调度。

### 9.6.10 处理 Executor 的状态变化

Executor 在运行的整个生命周期中，会向 Master 不断发送 ExecutorStateChanged 消息报告自身的状态改变。Master 中实现了对 ExecutorStateChanged 消息的处理，以根据 Executor 的状态变化进行调整，代码如下。

```
case ExecutorStateChanged(appId, execId, state, message, exitStatus) =>
  val execOption = idToApp.get(appId).flatMap(app => app.executors.get(execId))
  execOption match {
    case Some(exec) =>
      val appInfo = idToApp(appId)
      val oldState = exec.state
      exec.state = state

      if (state == ExecutorState.RUNNING) {
        assert(oldState == ExecutorState.LAUNCHING,
          s"executor $execId state transfer from $oldState to RUNNING is illegal")
        appInfo.resetRetryCount()
      }

      exec.application.driver.send(ExecutorUpdated(execId, state, message, exit-
      Status, false))
  }
```

```

    if (ExecutorState.isFinished(state)) {
      logInfo(s"Removing executor ${exec.fullId} because it is $state")
      if (!appInfo.isFinished) {
        appInfo.removeExecutor(exec)
      }
      exec.worker.removeExecutor(exec)

      val normalExit = exitStatus == Some(0)
      if (!normalExit
          && appInfo.incrementRetryCount() >= MAX_EXECUTOR_RETRIES
          && MAX_EXECUTOR_RETRIES >= 0) { // < 0 disables this application-
          killing path
      val execs = appInfo.executors.values
      if (!execs.exists(_.state == ExecutorState.RUNNING)) {
        logError(s"Application ${appInfo.desc.name} with ID ${appInfo.id}
                  failed " +
                  s"${appInfo.retryCount} times; removing it")
        removeApplication(appInfo, ApplicationState.FAILED)
      }
    }
    schedule()
  case None =>
    logWarning(s"Got status update for unknown executor $appId/$execId")
  }
}

```

根据上述代码，Master 收到 ExecutorStateChanged 消息后，将针对 Executor 的运行中 (RUNNING)、完成 (包括 KILLED、FAILED、LOST、EXITED 等状态)、正常退出、异常退出等状态进行相应的处理。如果 Executor 非正常退出且重试次数已经超过了 MAX\_EXECUTOR\_RETRIES 的限制，那么 Executor 所属的 Application 在没有任何 Executor 处于 RUNNING 状态时将被彻底移除。

### 9.6.11 Master 的常用方法

除了以上介绍的工作流程外，Master 还有一些值得注意的功能，包括回收 Master 的领导权、移除 Driver、重新运行 Driver、移除 Worker、移除 Application 等。下面将对这些功能的关键代码进行分析。

#### 1. revokedLeadership

根据 9.5 节对 ZooKeeperLeaderElectionAgent 的介绍，我们知道当 Master 没有被选举为领导时，ZooKeeperLeaderElectionAgent 将会调用 revokedLeadership 方法撤销 Master 的领导关系，其实现如代码清单 9-45 所示。

代码清单 9-45 撤销 Master 的领导关系

---

```

override def revokedLeadership() {
  self.send(RevokedLeadership)
}

```

---

根据代码清单 9-45, revokedLeadership 方法只是向 Master 自身发送了 RevokedLeadership 消息。

Master 实现的 receive 方法中实现了对 RevokedLeadership 消息的处理, 代码如下。

```
case RevokedLeadership =>
    logError("Leadership has been revoked -- master shutting down.")
    System.exit(0)
```

根据上述代码, Master 将退出。

## 2. removeDriver

removeDriver 方法用于移除 Master 维护的关于指定 Driver 的相关信息和状态, 其实现如代码清单 9-46 所示。

代码清单9-46 removeDriver方法的实现

---

```
private def removeDriver(
    driverId: String,
    finalState: DriverState,
    exception: Option[Exception]) {
    drivers.find(d => d.id == driverId) match { //从DriverInfo的集合drivers中找到指定
        的DriverInfo
        case Some(driver) =>
            logInfo(s"Removing driver: $driverId")
            drivers -= driver // 从drivers中移除找到的DriverInfo
            if (completedDrivers.size >= RETAINED_DRIVERS) {
                val toRemove = math.max(RETAINED_DRIVERS / 10, 1)
                completedDrivers.trimStart(toRemove) // 移除completedDrivers中开头的一些
                DriverInfo
            }
            completedDrivers += driver // 将找到的DriverInfo放入completedDrivers集合中
            persistenceEngine.removeDriver(driver) // 移除DriverInfo的持久化数据
            driver.state = finalState
            driver.exception = exception
            driver.worker.foreach(w => w.removeDriver(driver))
            schedule() // 由于腾出了Driver占用的资源, 所以对其他Application和Driver进行调度
        case None =>
            logWarning(s"Asked to remove unknown driver: $driverId")
    }
}
```

---

根据代码清单 9-46, removeDriver 方法的执行步骤如下。

- 1) 从 DriverInfo 的集合 drivers 中找到指定的 DriverInfo。
- 2) 从 drivers 中移除找到的 DriverInfo。
- 3) 如果已经完成的 DriverInfo 的集合 completedDrivers 中的元素数量大于等于 RETAINED\_DRIVERS, 那么移除 completedDrivers 中开头的一些 DriverInfo。
- 4) 将找到的 DriverInfo 放入 completedDrivers 集合中。
- 5) 调用持久化引擎的 removeDriver 方法移除 DriverInfo 的持久化数据。
- 6) 对 DriverInfo 的一些属性进行修改, 最后对处理此 Driver 提交的 Job 的所有 Worker,

调用 WorkerInfo 的 removeDriver 方法（实现很简单，感兴趣的读者请自行阅读），移除 Worker 处理的 Driver 信息。

7) 由于 Worker 上的资源已经腾出来了，所以需要调用 schedule 方法对其他 Application 和 Driver 进行调度。

### 3. relaunchDriver

relaunchDriver 方法用于重新调度运行指定的 Driver，其实现如代码清单 9-47 所示。

代码清单9-47 重新调度运行Driver

---

```
private def relaunchDriver(driver: DriverInfo) {
    driver.worker = None // 表示Driver提交的应用没有被任何Worker处理
    driver.state = DriverState.RELAUNCHING
    waitingDrivers += driver
    schedule()
}
```

---

根据代码清单 9-47，relaunchDriver 方法的执行步骤如下。

- 1) 将 DriverInfo 的 worker 属性置为 None，以表示 Driver 提交的应用没有被任何 Worker 处理。
- 2) 将 DriverInfo 的状态置为重新运行 (RELAUNCHING)。
- 3) 将 DriverInfo 放入所有等待调度的 Driver 的集合 waitingDrivers 中。
- 4) 调用 schedule 方法对 Driver 重新调度运行。

### 4. removeWorker

removeWorker 方法用于移除 Master 维护的关于指定 Worker 的相关信息和状态，其实现如代码清单 9-48 所示。

代码清单9-48 removeWorker的实现

---

```
private def removeWorker(worker: WorkerInfo) {
    logInfo("Removing worker " + worker.id + " on " + worker.host + ":" + worker.port)
    worker.setState(WorkerState.DEAD)
    idToWorker -= worker.id
    addressToWorker -= worker.endpoint.address
    if (reverseProxy) {
        webUi.removeProxyTargets(worker.id)
    }
    for (exec <- worker.executors.values) { // 将Worker上的Executor都作为丢失状态处理
        logInfo("Telling app of lost executor: " + exec.id)
        exec.application.driver.send(ExecutorUpdated(
            exec.id, ExecutorState.LOST, Some("worker lost"), None, workerLost = true))
        exec.state = ExecutorState.LOST
        exec.application.removeExecutor(exec)
    }
    for (driver <- worker.drivers.values) {
        if (driver.desc.supervise) {
            logInfo(s"Re-launching ${driver.id}")
            relaunchDriver(driver) // 重新调度运行Driver
        }
    }
}
```

---

```

    } else {
      logInfo(s"Not re-launching ${driver.id} because it was not supervised")
      removeDriver(driver.id, DriverState.ERROR, None)
    }
}
persistenceEngine.removeWorker(worker)
}

```

---

根据代码清单 9-48, removeWorker 方法的执行步骤如下。

- 1) 将 WorkerInfo 的状态置为 DEAD。
- 2) 从 idToWorker 和 addressToWorker 等缓存中移除 WorkerInfo。
- 3) 向此 Worker 的所有 Executor 所服务的 Driver Application 发送 ExecutorUpdated 消息, 更新 Executor 的状态为 LOST。最后调用 ApplicationInfo 的 removeExecutor 方法(此方法的实现很简单,感兴趣的读者可以自行阅读), 移除分配给 Application 的 Executor 的描述信息。
- 4) 对 Worker 所处理的所有 Driver 进行如下操作: 如果 Driver 是被监管的, 则调用 relaunchDriver 方法(见代码清单 9-47) 重新调度运行 Driver, 否则调用 removeDriver 方法(见代码清单 9-46) 移除 Driver 的相关信息和状态。
- 5) 调用持久化引擎的 removeWorker 方法移除 WorkerInfo 的持久化数据。

## 5. removeApplication

removeApplication 方法用于移除 Master 中缓存的 Application 及 Application 相关的 Driver 信息, 其实现如代码清单 9-49 所示。

代码清单 9-49 removeApplication 方法的实现

```

def removeApplication(app: ApplicationInfo, state: ApplicationState.Value) {
  if (apps.contains(app)) {
    logInfo("Removing app " + app.id)
    apps -= app
    idToApp -= app.id
    endpointToApp -= app.driver
    addressToApp -= app.driver.address
    if (reverseProxy) {
      webUi.removeProxyTargets(app.id)
    }
    if (completedApps.size >= RETAINED_APPLICATIONS) {
      val toRemove = math.max(RETAINED_APPLICATIONS / 10, 1)
      completedApps.take(toRemove).foreach { a =>
        applicationMetricsSystem.removeSource(a.appSource)
      }
      completedApps.trimStart(toRemove)
    }
    completedApps += app // 将ApplicationInfo作为已完成的历史应用
    waitingApps -= app

    for (exec <- app.executors.values) { // 遍历分配给Application的Executor
      killExecutor(exec) // “杀死” Executor
    }
  }
}

```

```

        }
        app.markFinished(state)
        if (state != ApplicationState.FINISHED) {
            app.driver.send(ApplicationRemoved(state.toString()))
        }
        persistenceEngine.removeApplication(app) // 移除ApplicationInfo的持久化信息
        schedule()

        workers.foreach { w =>
            w.endpoint.send(ApplicationFinished(app.id))
        }
    }
}

```

---

根据代码清单 9-49，removeApplication 方法的执行步骤如下。

- 1 ) 移除 Master 中缓存的 ApplicationInfo 及 ApplicationInfo 相关的 DriverInfo。
- 2 ) 将 ApplicationInfo 添加到数组缓冲 completedApps 中。
- 3 ) 遍历分配给 Application 的 Executor，调用 killExecutor 方法 “杀死”。
- 4 ) 向 DriverEndpoint 发送 ApplicationRemoved 消息，告诉 Driver 驱动的 Application 已经被移除。
- 5 ) 对 ApplicationInfo 去持久化。
- 6 ) 调用 schedule 方法进行资源调度。
- 7 ) 向每个 Worker 发送 ApplicationFinished 消息，以告知 Application 已完成。

## 9.7 Worker 详解

Worker 是 Spark 在 local-cluster 部署模式和 Standalone 部署模式中对工作节点的资源和 Executor 进行管理的服务。Worker 一方面向 Master 汇报自身所管理的资源信息，一方面接收 Master 的命令运行 Driver 或者为 Application 运行 Executor。同一个机器上可以同时部署多个 Worker 服务，一个 Worker 也可以启动多个 Executor。当 Executor 完成后，Worker 将回收 Executor 使用的资源。

本节将对 Worker 进行详细分析，但是我们还是从了解它所包含的属性开始。Worker 包含的属性如下。

- `rpcEnv`: 即 `RpcEnv`。
- `webUiPort`: 参数指定的 WebUI 的端口。
- `cores`: 内核数。
- `memory`: 内存大小。
- `masterRpcAddresses`: Master 的 `RpcEnv` 地址（即 `RpcAddress`）的数组。由于一个集群为了可靠性和容错，需要多个 Master 节点，因此用数组来存储它们的 `RpcEnv` 地址。
- `endpointName`: Worker 注册到 `RpcEnv` 的名称。

- ❑ workDirPath：字符串表示的 Worker 的工作目录。
- ❑ conf：即 SparkConf。
- ❑ securityMgr：即 SecurityManager。
- ❑ host：Worker 的 RpcEnv 的 host。
- ❑ port：Worker 的 RpcEnv 的端口。
- ❑ forwardMessageScheduler：用于发送消息的调度执行器（ScheduledThreadPoolExecutor）。forwardMessageScheduler 只能调度执行一个线程，执行的线程以 worker-forward-message-scheduler 作为名称。
- ❑ cleanupThreadExecutor：通过 Executors 的 newSingleThreadExecutor 方法创建的线程执行器，用于清理 Worker 的工作目录。由 cleanupThreadExecutor 执行的线程以 worker-cleanup-thread 作为名称。
- ❑ HEARTBEAT\_MILLIS：向 Master 发送心跳的时间间隔，是 spark.worker.timeout 属性值的 1/4，默认为 15 秒。
- ❑ INITIAL\_REGISTRATION\_ATTEMPTS：此常量固定为 6，代表连接 Master 的前六次尝试。
- ❑ TOTAL\_REGISTRATION\_ATTEMPTS：此常量固定为 16，代表连接 Master 最多有 16 次尝试。
- ❑ FUZZ\_MULTIPLIER\_INTERVAL\_LOWER\_BOUND：此常量固定为 0.500，是为了确保尝试的时间间隔不至于过小的下边界。
- ❑ REGISTRATION\_RETRY\_FUZZ\_MULTIPLIER：0~1 范围的随机数与 FUZZ\_MULTIPLIER\_INTERVAL\_LOWER\_BOUND 的和，所以 REGISTRATION\_RETRY\_FUZZ\_MULTIPLIER 的真实范围在 0.5~1.5 之间。加入随机数是为了避免各个 Worker 在同一时间发送心跳。
- ❑ INITIAL\_REGISTRATION\_RETRY\_INTERVAL\_SECONDS：代表前六次尝试的时间间隔。INITIAL\_REGISTRATION\_RETRY\_INTERVAL\_SECONDS 取 10 和 REGISTRATION\_RETRY\_FUZZ\_MULTIPLIER 乘积的最近整数，因此这六次尝试的时间间隔在 5~15s 之间。
- ❑ PROLONGED\_REGISTRATION\_RETRY\_INTERVAL\_SECONDS：代表最后十次尝试的时间间隔。PROLONGED\_REGISTRATION\_RETRY\_INTERVAL\_SECONDS 取 60 和 REGISTRATION\_RETRY\_FUZZ\_MULTIPLIER 乘积的最近整数，因此最后十次尝试的时间间隔在 30~90s 之间。
- ❑ CLEANUP\_ENABLED：是否对旧的 Application 产生的文件夹及文件进行清理。可通过 spark.worker.cleanup.enabled 属性配置，默认为 false。
- ❑ CLEANUP\_INTERVAL\_MILLIS：对旧的 Application 产生的文件夹及文件进行清理的时间间隔。可通过 spark.worker.cleanup.interval 属性配置，默认为 1800 秒。

- ❑ APP\_DATA\_RETENTION\_SECONDS：Application 产生的文件夹及文件保留的时间。可通过 spark.worker.cleanup.appDataTtl 属性配置，默认为 7 天。
- ❑ master：Master 的 RpcEndpointRef。
- ❑ activeMasterUrl：处于激活（ALIVE）状态的 Master 的 URL。
- ❑ activeMasterWebUiUrl：处于激活（ALIVE）状态的 Master 的 WebUI 的 URL。
- ❑ workerWebUiUrl：Worker 的 WebUI 的 URL。
- ❑ workerUri：Worker 的 URL，格式为 spark://\$name@\$host:\$port}。
- ❑ registered：标记 Worker 是否已经注册到 Master。
- ❑ connected：标记 Worker 是否已经连接到 Master。
- ❑ workerId：Worker 的身份标识。
- ❑ sparkHome：环境变量 SPARK\_HOME 的值。
- ❑ workDir：由 java.io.File 表示的 Worker 的工作目录。
- ❑ finishedExecutors：已经完成的 Executor 的身份标识与 ExecutorRunner 之间的映射关系。
- ❑ drivers：Driver 的身份标识与 DriverRunner 之间的映射关系。
- ❑ executors：Executor 的身份标识与 ExecutorRunner 之间的映射关系。
- ❑ finishedDrivers：已经完成的 Driver 的身份标识与 DriverRunner 之间的映射关系。
- ❑ appDirectories：Application 的 ID 与对应的目录集合之间的映射关系。
- ❑ finishedApps：已经完成的 Application 的 ID 的集合。
- ❑ retainedExecutors：保留的 Executor 的数量。可通过 spark.worker.ui.retainedExecutors 属性配置，默认为 1000。
- ❑ retainedDrivers：保留的 Driver 的数量。可通过 spark.worker.ui.retainedDrivers 属性配置，默认为 1000。
- ❑ shuffleService：外部的 Shuffle 服务。在 6.9 节曾经简单介绍过外部 Shuffle 服务的客户端 ExternalShuffleClient。这里则是外部 Shuffle 服务的服务端实现类 ExternalShuffleService。这里虽然创建了 ExternalShuffleService，但是只有配置了外部的 Shuffle 服务时，才会启动它。<sup>⊖</sup>
- ❑ publicAddress：Worker 的公共地址，如果设置了环境变量 SPARK\_PUBLIC\_DNS，则为环境变量 SPARK\_PUBLIC\_DNS 的值，否则为 host。
- ❑ webUi：Worker 的 WebUI，类型为 WebUI 的子类 WorkerWebUI。
- ❑ connectionAttemptCount：连接尝试次数的计数器。
- ❑ metricsSystem：实例名为 worker 的 MetricsSystem，即 Worker 的度量系统。
- ❑ workerSource：类型为 WorkerSource，是有关 Worker 的度量来源。

<sup>⊖</sup> 本书不会对 ExternalShuffleClient 和 ExternalShuffleService 的实现进行深入分析，留给感兴趣的读者自行研究。

- registerMasterFutures：由于 Worker 向 Master 进行注册的过程是异步的，此变量保存线程返回的 `java.util.concurrent.Future`。
- registrationRetryTimer：Worker 向 Master 进行注册重试的定时器。
- registerMasterThreadPool：Worker 向 Master 进行注册的线程池。此线程池的大小与 Master 节点的数量一样，启动的线程以 `worker-register-master-threadpool` 为前缀。
- coresUsed：当前 Worker 已经使用的内核数。Worker 提供了 `coresFree` 方法返回 `cores` 属性和 `coresUsed` 的差值，作为空闲的内核数。
- memoryUsed：当前 Worker 已经使用的内存大小。Worker 提供了 `memoryFree` 方法返回 `memory` 属性和 `memoryUsed` 的差值，作为空闲的内存大小。

有了对 Worker 中包含属性的了解，现在可以来看看 Worker 的主要执行流程了。

### 9.7.1 启动 Worker

启动 Worker 有作为 JVM 进程内的对象启动和作为单独的进程启动两种方式。以对象启动的方式主要用于 local-cluster 模式，而作为进程启动则用于 Standalone 模式。

#### 1. 对象方式启动

Worker 的伴生对象的 `startRpcEnvAndEndpoint` 方法（见代码清单 9-50）用于创建、启动 Worker。

代码清单9-50 创建、启动Worker

---

```
def startRpcEnvAndEndpoint(
    host: String, port: Int, webUiPort: Int, cores: Int, memory: Int,
    masterUrls: Array[String], workDir: String,
    workerNumber: Option[Int] = None, conf: SparkConf = new SparkConf): RpcEnv = {
  val systemName = SYSTEM_NAME + workerNumber.map(_.toString).getOrElse("")
  val securityMgr = new SecurityManager(conf)
  val rpcEnv = RpcEnv.create(systemName, host, port, conf, securityMgr)
  val masterAddresses = masterUrls.map(RpcAddress.fromSparkURL(_))
  rpcEnv.setupEndpoint(ENDPOINT_NAME, new Worker(rpcEnv, webUiPort, cores, memory,
    masterAddresses, ENDPOINT_NAME, workDir, conf, securityMgr))
  rpcEnv
}
```

---

根据代码清单 9-50，Worker 的伴生对象的 `startRpcEnvAndEndpoint` 方法的执行步骤如下。

- 1) 生成 Worker 的 `RpcEnv` 的名称。生成规则为 `SYSTEM_NAME` 后跟 Worker 号。由于常量 `SYSTEM_NAME` 的值为 `sparkWorker`，所以 Worker 的 `RpcEnv` 的名称将会是 `spark-Worker0`、`sparkWorker1` 等。
- 2) 创建 `SecurityManager`。
- 3) 创建 Worker 的 `RpcEnv`。
- 4) 将所有 Master 的 Spark URL（格式为 `spark://host:port`）转换为 `RpcAddress` 地址。`Rpc-`

Address 只包含 host 和 port 两个属性。

5) 创建 Worker，并且将 Worker（Worker 也继承了 ThreadSafeRpcEndpoint）注册到刚创建的 RpcEnv 中。

6) 返回 Worker 的 RpcEnv。

在向 RpcEnv 中注册 Worker 时，会触发对 Worker 的 onStart 方法（见代码清单 9-51）的调用。

代码清单9-51 Worker的onStart方法

---

```
override def onStart() {
    assert(!registered)
    // 省略日志输出代码
    createWorkDir()
    shuffleService.startIfEnabled()
    webUi = new WorkerWebUI(this, workDir, webUiPort)
    webUi.bind()
    workerWebUiUrl = s"http://$publicAddress:${webUi.boundPort}"
    registerWithMaster()

    metricsSystem.registerSource(workerSource)
    metricsSystem.start()
    metricsSystem.getServletHandlers.foreach(webUi.attachHandler)
}
```

---

根据代码清单 9-51，Worker 的 onStart 方法的执行步骤如下。

- 1) 调用 createWorkDir 方法（见代码清单 9-52）创建 Worker 的工作目录。
- 2) 如果配置了外部的 Shuffle 服务，那么启动 shuffleService。
- 3) 创建 WorkerWebUI，并为其绑定端口。
- 4) 拼接 Worker 的 WebUI 的 URL，并保存到 workerWebUiUrl 属性。
- 5) 调用 registerWithMaster 方法（见代码清单 9-54）向 Master 注册 Worker。
- 6) 将 workerSource 注册到 metricsSystem，然后启动 metricsSystem，最后将 metricsSystem 的 ServletContextHandler 添加到 WorkerWebUI。

代码清单9-52 创建Worker的工作目录

---

```
private def createWorkDir() {
    workDir = Option(workDirPath).map(new File(_)).getOrElse(new File(sparkHome,
        "work"))
    try {
        workDir.mkdirs()
        if (!workDir.exists() || !workDir.isDirectory) {
            logError("Failed to create work directory " + workDir)
            System.exit(1)
        }
        assert (workDir.isDirectory)
    } catch {
        // 省略捕获到异常后退出进程的代码
    }
```

---

## 2. 进程方式启动

Worker 的伴生对象中实现了 main 方法，这样就可以作为单独的 JVM 进程启动了。Main 函数的实现如代码清单 9-53 所示。

代码清单 9-53 Worker 伴生对象的 main 方法

---

```
def main(argStrings: Array[String]) {
    Utils.initDaemon(log)
    val conf = new SparkConf
    val args = new WorkerArguments(argStrings, conf)
    val rpcEnv = startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, args.cores,
        args.memory, args.masters, args.workDir, conf = conf)
    rpcEnv.awaitTermination()
}
```

---

根据代码清单 9-53，main 方法的执行步骤如下。

1 ) 创建 SparkConf。

2 ) 创建 WorkerArguments 以对执行 main 函数的参数进行解析，并将 Spark 属性配置文件中以 spark. 开头的属性保存到 SparkConf。

3 ) 调用 Worker 伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-50）创建并启动 Worker 对象。

WorkerArguments 方法用于解析系统环境变量和启动 Worker 时指定的命令行参数，由于其实现和 MasterArguments 非常相似，因此对其源码不再展示。这里只列出 WorkerArguments 解析的参数，如表 9-3 所示。

表 9-3 WorkerArguments 解析的参数

属性	含义	默认值	系统环境变量	命令行参数
host	Worker 的监听地址	机器名	—	--ip 或者 -i (已弃用) --host 或者 -h
port	Worker 的监听端口	0	SPARK_WORKER_PORT	--port 或者 -p
webUiPort	WebUI 的监听端口	8081	SPARK_WORKER_WEBUI_PORT	--webui-port
propertiesFile	Spark 属性文件	spark-defaults.conf	—	--properties-file
cores	Worker 的内核数	系统内核数	SPARK_WORKER_CORES	--cores 或者 -c
memory	Worker 的内存大小	系统最大内存减 1GB	SPARK_WORKER_MEMORY	--memory 或者 -m
masters	Master 地址列表	null	—	例如，spark://abc,def
workDir	Worker 的工作目录	null	SPARK_WORKER_DIR	--work-dir 或者 -d

 **注意** memory 默认为操作系统最大内存减去 1GB，这 1GB 是留给操作系统使用的。

WorkerArguments 的 parse 函数与 MasterArguments 中的 parse 函数的实现类似，读者

可自行阅读其源码。WorkerArguments 的 parse 函数要求必须在启动 Worker 时指定 spark:// 的参数作为 master 的 url。命令行参数的值会覆盖系统环境变量的值，解析的命令行参数如表 9-4 所示。

表 9-4 WorkerArguments 的 parse 函数解析的命令行参数

参数名	参数含义
--ip 或者 -i	指定 host name，将来会停止使用，推荐使用 --host 或者 -h
--host 或者 -h	指定 host name
--port 或者 -p	指定端口
--webui-port	指定 WebUI 的端口
--properties-file	Spark 系统属性文件
--cores 或者 -c	指定内核数
--memory 或者 -m	指定内存大小
--work-dir 或者 -d	指定工作目录
--help	帮助



命令行参数指定的值会覆盖系统环境变量指定的值。属性 spark.worker.ui.port 的值，会覆盖环境变量 SPARK\_WORKER\_WEBUI\_PORT 和命令行参数 --webui-port 指定的值。

### 9.7.2 向 Master 注册 Worker

Worker 在启动后，需要加入到 Master 管理的整个集群中，以参与 Driver、Executor 的资源调度。Worker 要加入 Master 管理的集群，就必须将 Worker 注册到 Master。在启动 Worker 的过程中需要调用 registerWithMaster 方法向 Master 注册 Worker，其实现如代码清单 9-54 所示。

代码清单 9-54 向 Master 注册 Worker

```
private def registerWithMaster() {
    registrationRetryTimer match {
        case None =>
            registered = false
            registerMasterFutures = tryRegisterAllMasters() // 向所有的Master注册当前Worker
            connectionAttemptCount = 0
            registrationRetryTimer = Some(forwordMessageScheduler.scheduleAtFixedRate(
                new Runnable {
                    override def run(): Unit = Utils.tryLogNonFatalError {
                        Option(self).foreach(_.send(ReregisterWithMaster))
                    }
                },
                INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS,
                INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS,
                TimeUnit.SECONDS))
    }
}
```

```

case Some(_) =>
    logInfo("Not spawning another attempt to register with the master, since there
            is an" +
            " attempt scheduled already.")
}
}

```

---

根据代码清单 9-54，registerWithMaster 方法的执行步骤如下。

1) 调用 tryRegisterAllMasters 方法（见代码清单 9-55）向所有的 Master 注册当前 Worker。根据 9.6.5 节的内容，只有处于领导状态的 Master 来处理 Worker 的注册。

2) 创建定时任务 registrationRetryTimer，按照常量 INITIAL\_REGISTRATION\_RETRY\_INTERVAL\_SECONDS 指定的间隔向 Worker 自身发送 ReregisterWithMaster 消息。在 Worker 的 receive 方法中使用了重载的 reregisterWithMaster 方法来处理 ReregisterWithMaster 消息，代码如下。

```

case ReregisterWithMaster =>
    reregisterWithMaster()

```

reregisterWithMaster 方法用于在 Worker 的 registered 为 false 时，重新向 Master 注册，由于其实现与 registerWithMaster 方法很多地方都是相似的，因此本文不对其进行介绍，留给感兴趣的读者自行查阅。

#### 代码清单9-55 tryRegisterAllMasters的实现

```

private def tryRegisterAllMasters(): Array[JFuture[_]] = {
    masterRpcAddresses.map { masterAddress =>
        registerMasterThreadPool.submit(new Runnable {
            override def run(): Unit = {
                try {
                    logInfo("Connecting to master " + masterAddress + "...")
                    val masterEndpoint = rpcEnv.setupEndpointRef(masterAddress, Master.END-
                        POINT_NAME)
                    registerWithMaster(masterEndpoint)
                } catch {
                    // 忽略异常处理代码
                }
            }
        })
    }
}

```

---

根据代码清单 9-55，tryRegisterAllMasters 方法实际上是遍历 masterRpcAddresses 中的每个 Master 的 RPC 地址，然后向 registerMasterThreadPool 提交向 Master 注册 Worker 的任务。此任务实际是调用重载的 registerWithMaster 方法（见代码清单 9-56）用于完成最后的注册。

#### 代码清单9-56 重载的registerWithMaster方法

```

private def registerWithMaster(masterEndpoint: RpcEndpointRef): Unit = {

```

```

masterEndpoint.ask[RegisterWorkerResponse](RegisterWorker(
    workerId, host, port, self, cores, memory, workerWebUiUrl))
    .onComplete {
        case Success(msg) =>
            Utils.tryLogNonFatalError {
                handleRegisterResponse(msg)
            }
        case Failure(e) =>
            logError(s"Cannot register with master: ${masterEndpoint.address}", e)
            System.exit(1)
    }(ThreadUtils.sameThread)
}

```

根据代码清单 9-56，registerWithMaster 方法主要是向 Master 发送 RegisterWorker 消息（此消息将携带 Worker 的 ID、host、port、内核数、内存大小等信息），并对返回的结果使用 handleRegisterResponse 方法处理。Master 接收 RegisterWorker 消息的处理过程已在 9.6.5 节介绍过了，这里主要分析 handleRegisterResponse 方法的实现，如代码清单 9-57 所示。

代码清单9-57 handleRegisterResponse的实现

```

private def handleRegisterResponse(msg: RegisterWorkerResponse): Unit = synchronized {
    msg match {
        case RegisteredWorker(masterRef, masterWebUiUrl) =>
            logInfo("Successfully registered with master " + masterRef.address.toSparkURL)
            registered = true
            changeMaster(masterRef, masterWebUiUrl) // 切换Master
            forwardMessageScheduler.scheduleAtFixedRate(new Runnable {
                override def run(): Unit = Utils.tryLogNonFatalError {
                    self.send(SendHeartbeat) // 定时向Master发送心跳
                }
            }, 0, HEARTBEAT_MILLIS, TimeUnit.MILLISECONDS)
            if (CLEANUP_ENABLED) {
                logInfo(
                    s"Worker cleanup enabled; old application directories will be deleted in: "
                    + workDir)
                forwardMessageScheduler.scheduleAtFixedRate(new Runnable {
                    override def run(): Unit = Utils.tryLogNonFatalError {
                        self.send(WorkDirCleanup) // 定时清理工作目录
                    }
                }, CLEANUP_INTERVAL_MILLIS, CLEANUP_INTERVAL_MILLIS, TimeUnit.MILLISECONDS)
            }
        val execs = executors.values.map { e =>
            new ExecutorDescription(e.appId, e.execId, e.cores, e.state)
        }
        masterRef.send(WorkerLatestState(workerId, execs.toList, drivers.keys.toSeq)) //汇报状态
    }

    case RegisterWorkerFailed(message) =>
        if (!registered) {
            logError("Worker registration failed: " + message)
            System.exit(1)
        }
    }
}

```

```

        case MasterInStandby => // Master还未准备好或者Master处于支持(STANDBY)状态
    }
}

```

根据 9.6.5 节对 Master 注册 Worker 的分析，我们知道 Master 将给 Worker 回复三种可能的消息，而 handleRegisterResponse 方法也的确处理了这三种消息。

1) 处理 RegisteredWorker 消息。如果 Master 回复了此消息，说明 Worker 已经成功在 Master 上注册，具体的处理步骤如下。

- ① 将 registered 设置为 true。
- ② 调用 changeMaster 方法（见代码清单 9-58）修改激活的 Master 的信息。changeMaster 方法还调用 cancelLastRegistrationRetry 方法取消注册尝试（即取消了调用 tryRegisterAllMasters 方法产生的注册任务和 registrationRetryTimer）。
- ③ 向 forwardMessageScheduler 提交以 HEARTBEAT\_MILLIS 作为间隔向 Worker 自身发送 SendHeartbeat 消息的定时任务。9.7.3 节对 Worker 处理 SendHeartbeat 消息的过程进行了详细的分析。
- ④ 如果允许清理工作目录（即 CLEANUP\_ENABLED 为 true），向 forwardMessageScheduler 提交以 CLEANUP\_INTERVAL\_MILLIS 作为间隔向 Worker 自身发送 WorkDir-Cleanup 消息的定时任务。
- ⑤ 向 Master 发送 WorkerLatestState 消息（此消息携带 Worker 的身份标识、Worker 节点的所有 Executor 的描述信息、调度到当前 Worker 的所有 Driver 的身份标识）。

代码清单9-58 changeMaster方法的实现

```

private def changeMaster(masterRef: RpcEndpointRef, uiUrl: String) {
    activeMasterUrl = masterRef.address.toSparkURL
    activeMasterWebUiUrl = uiUrl
    master = Some(masterRef)
    connected = true
    if (conf.getBoolean("spark.ui.reverseProxy", false)) {
        logInfo(s"WorkerWebUI is available at $activeMasterWebUiUrl/proxy/$workerId")
    }
    cancelLastRegistrationRetry()
}

private def cancelLastRegistrationRetry(): Unit = {
    if (registerMasterFutures != null) {
        registerMasterFutures.foreach(_.cancel(true))
        registerMasterFutures = null
    }
    registrationRetryTimer.foreach(_.cancel(true))
    registrationRetryTimer = None
}

```

2) 处理 RegisterWorkerFailed 消息。如果 Master 回复了此消息，说明 Worker 在 Master 上注册失败了。如果 Worker 还未向任何 Master 节点注册成功，那么退出 Worker 进程。

3) 处理 MasterInStandby 消息。如果 Master 回复了此消息，说明 Master 处于 Standby 的状态，并不是领导身份，此时 Worker 不作任何处理。

有了对 Master 和 Worker 各自有关注册 Worker 的实现分析，我们用图 9-16 来表示注册 Worker 的完整流程。

这里对图 9-16 中的各个序号进行说明。

序号①：表示 Worker 刚刚启动时，由于不知道哪个 Master 是 ALIVE 的，所以向所有 Master 发送 RegisterWorker 消息（此消息将携带 Worker 的 ID、host、port、内核数、内存大小等信息）。向状态为 STANDBY 的 Master 发送消息将接收到 MasterInStandby 消息，Worker 由此知道这个 Master 不“管事”。如果 Worker 向 ALIVE 状态的 Master 重复注册或者注册失败，将会收到 RegisterWorkerFailed 消息，Worker 发现自己还未注册成功时将退出。

序号②：状态为 ALIVE 的 Master 接收到 RegisterWorker 消息后，将根据 RegisterWorker 消息携带的信息构建出 WorkerInfo，并将 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中，最后调用 PersistenceEngine 的 addWorker 方法将 WorkerInfo 持久化。

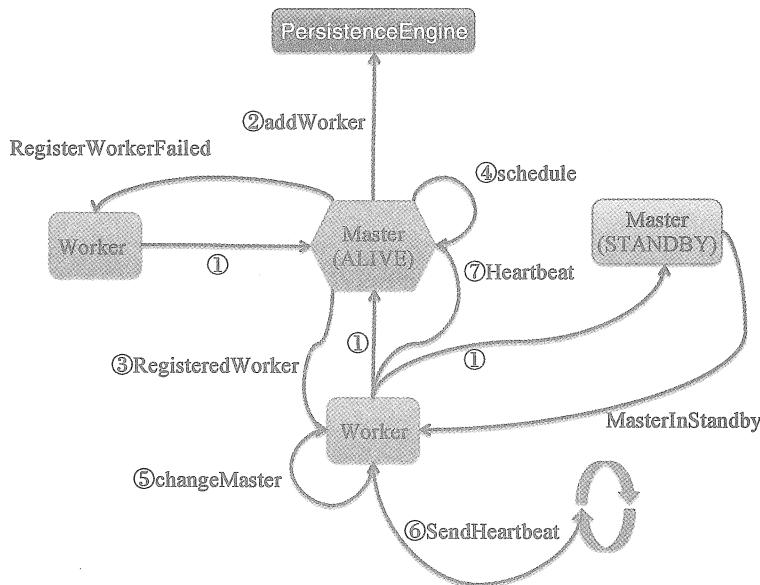


图 9-16 注册 Worker 的完整流程

序号③：状态为 ALIVE 的 Master 在 Worker 注册成功后，向 Worker 回复 RegisteredWorker 消息。

序号④：状态为 ALIVE 的 Master 在向 Worker 回复 RegisteredWorker 消息后，将调用

schedule 对 Driver 及 Executor 进行资源调度。

序号⑤：Worker 接收到 Master 回复的 RegisteredWorker 消息后，将调用 changeMaster 方法修改激活的 Master 的信息。

序号⑥：Worker 调用 changeMaster 方法后，将向 forwardMessageScheduler 提交向 Worker 自身发送 SendHeartbeat 消息的定时任务。

序号⑦：Worker 接收到 SendHeartbeat 消息后，将向 Master 发送心跳（Heartbeat）消息。Master 接收到 Heartbeat 消息后，将更新 WorkerInfo 的最后心跳时间（lastHeartbeat）。

### 9.7.3 向 Master 发送心跳

为了让 Master 得知 Worker 依然健康运行着，就需要不断地告诉 Master：“我活着”，这个过程是通过发送心跳实现的。

根据之前的内容我们知道，当 Worker 向 Master 注册成功后会接收到 Master 回复的 RegisteredWorker 消息，Worker 使用 handleRegisterResponse 方法处理 RegisteredWorker 消息时，将会向 forwardMessageScheduler 提交以 HEARTBEAT\_MILLIS 作为间隔向 Worker 自身发送 SendHeartbeat 消息的定时任务。Worker 的 receive 方法实现了对 SendHeartbeat 消息的处理，代码如下。

```
case SendHeartbeat =>
    if (connected) { sendToMaster(Heartbeat(workerId, self)) }
```

根据上述代码，我们知道如果 connected 为 true，则调用 sendToMaster 方法（见代码清单 9-59），向 Master 发送 Heartbeat 消息（此消息将携带 Worker 的 ID 和 Worker 自己的 RpcEndpointRef）。

代码清单9-59 sendToMaster方法的实现

---

```
private def sendToMaster(message: Any): Unit = {
    master match {
        case Some(masterRef) => masterRef.send(message)
        case None =>
            // 忽略Worker还未与Master建立连接时不作任何处理，只打印日志的代码
    }
}
```

---

根据 9.6.7 节的内容我们知道，Master 在处理 Heartbeat 消息时有可能向 Worker 发送 ReconnectWorker 消息，Worker 的 receive 方法接收到 ReconnectWorker 消息后只是再次调用了 registerWithMaster 方法（见代码清单 9-54）向 Master 注册 Worker，代码如下。

```
case ReconnectWorker(masterUrl) =>
    registerWithMaster()
```

经过对 Master 中检查 Worker 超时（9.6.2 节）、Master 处理 Worker 心跳（9.6.7 节），以及本节对 Worker 向 Master 发送心跳等环节的分析，我们可以用图 9-17 来表示 Worker 心跳

及超时检查的流程。

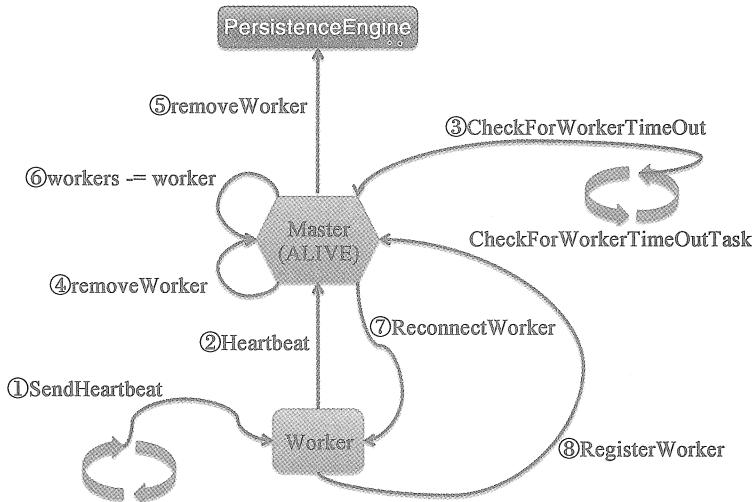


图 9-17 Worker 心跳及超时检查的流程

这里对图 9-17 中的各个序号进行说明。

序号①：Worker 在向 Master 注册成功后，将向 forwardMessageScheduler 提交以 HEARTBEAT\_MILLIS 作为间隔向 Worker 自身发送 SendHeartbeat 消息的定时任务。

序号②：Worker 接收到 SendHeartbeat 消息后，将向 Master 发送 Heartbeat 消息。在正常情况下，Worker 向 Master 发送的心跳足够及时，因此 Master 并未检查出 Worker 超时，此时 idToWorker 中将缓存着 WorkerInfo，Master 只需要将此 WorkerInfo 的最后心跳时间 (lastHeartbeat) 更新为系统当前时间戳。

序号③：Master 内部的定时任务 checkForWorkerTimeOutTask 将以 WORKER\_TIMEOUT\_MS 为时间间隔定时向 Master 自身发送 CheckForWorkerTimeOut 消息。Master 接收到 CheckForWorkerTimeOut 消息后，调用 timeOutDeadWorkers 方法检查超时的 Worker。

序号④：Master 在检查过程中发现 Worker 超时，但是对应的 WorkerInfo 的状态不是 DEAD，因此调用 removeWorker 方法移除 Master 维护的关于指定 Worker 的相关信息和状态。这里特别要关注的是，此时 Master 会将 WorkerInfo 从 idToWorker 中移除，但是 workers 中仍然保留 WorkerInfo。

序号⑤：removeWorker 方法的最后还将调用持久化引擎的 removeWorker 方法从持久化存储中移除 WorkerInfo。

序号⑥：Master 在检查过程中发现 Worker 超时且对应的 WorkerInfo 的状态是 DEAD，等待足够长的时间后将 WorkerInfo 从 workers 中移除。

序号⑦：Master 在检查 Worker 时，如果发生了序号④所示的情况，那么 idToWorker 中将没有对应的 WorkerInfo，但是 workers 中仍然保留 WorkerInfo，此时 Master 将向

Worker 发送 ReconnectWorker 消息。

序号⑧：Worker 接收到 ReconnectWorker 消息后，将重新调用 registerWithMaster 方法向 Master 注册，进而向 Master 发送 RegisterWorker 消息。

### 9.7.4 Worker 与领导选举

在 9.6.3 节介绍 Master 被选举为领导时，将会向 Worker 发送 MasterChanged 消息。Worker 的 receive 方法接收到 MasterChanged 消息后的处理如代码清单 9-60 所示。

代码清单9-60 Worker对MasterChanged消息的处理

---

```
case MasterChanged(masterRef, masterWebUiUrl) =>
    logInfo("Master has changed, new master is at " + masterRef.address.toSparkURL)
    changeMaster(masterRef, masterWebUiUrl)

    val execs = executors.values.
        map(e => new ExecutorDescription(e.appId, e.execId, e.cores, e.state))
    masterRef.send(WorkerSchedulerStateResponse(workerId, execs.toList, drivers.keys.toSeq))
```

---

根据代码清单 9-60，Worker 接收到 MasterChanged 消息后的处理步骤如下。

- 1) 调用 changeMaster 方法（见代码清单 9-58）改变激活的 Master 的相关信息。
- 2) 通过遍历 executors 中的每个 ExecutorRunner，利用 ExecutorRunner 的信息创建 ExecutorDescription。
- 3) 向 Master 发送 WorkerSchedulerStateResponse 消息。WorkerSchedulerStateResponse 消息携带着 Worker 的 ID、ExecutorDescription 列表、所有 Driver 的 ID 等信息。Master 接收到 WorkerSchedulerStateResponse 消息后将更新 Worker 的调度状态。由于 Master 处理 WorkerSchedulerStateResponse 消息的实现非常简单，因此留给感兴趣的读者自行阅读。

经过对持久化引擎、领导选举代理、Master 被选举为领导时的处理（9.6.3 节）、Master 如何注册 Worker、Master 如何注册 Application，以及本节对 Worker 处理 MasterChanged 消息等环节的分析，我们可以用图 9-18 来表示 Master 领导选举的整个流程。

图 9-18 中显示了三个 Master，其中 Master 0 是 ALIVE 状态的，Master 1 和 Master 2 是 STANDBY 状态的。图 9-18 中的各个序号的含义如下。

序号①：Worker 在启动后会向 Master 注册，Driver 在启动后也会向 Master 注册。在注册成功后，Master 会调用 PersistenceEngine 的 addWorker、addApplication 及 addDriver 三个方法，将 Worker、Driver、Application 的相关信息持久化。

序号②：Master 在运行的时候出现网络故障、宕机或者服务下线时，ZooKeeper 会感知到。

序号③：Master 1 和 Master 2 参加领导选举，最后 Master 1 胜出，于是 LeaderLatch 将调用 Master 1 的 ZooKeeperLeaderElectionAgent 的 isLeader 方法，isLeader 方法实际调用了 Master 1 的 electedLeader 方法。

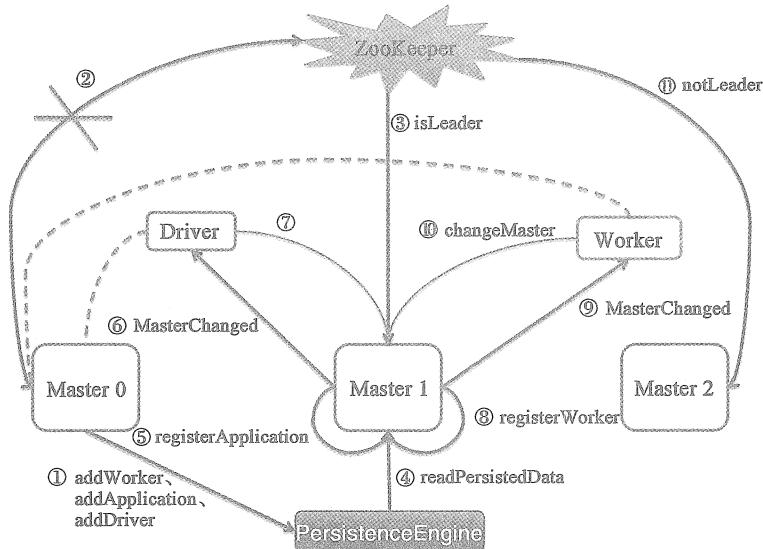


图 9-18 Master 领导选举的流程

序号④：electedLeader 方法调用 PersistenceEngine 的 readPersistedData 方法从持久化数据中读取 WorkerInfo、DriverInfo、ApplicationInfo，然后调用 Master 1 的 beginRecovery 方法开始恢复。

序号⑤：beginRecovery 方法对于从持久化引擎中读取的 ApplicationInfo，调用 registerApplication 方法注册到 apps、idToApp、endpointToApp、addressToApp、waitingApps 等缓存中，然后将 ApplicationInfo 的状态设置为 UNKNOWN。

序号⑥：beginRecovery 方法还将向提交 Application 的 Driver 发送 MasterChanged 消息。

序号⑦：Driver 接收到 MasterChanged 消息后，将自身的 master 属性修改为当前 Master 的 RpcEndpointRef，并将 alreadyDisconnected 设置为 false，最后向 Master 发送 MasterChangeAcknowledged 消息。Master 接收到 MasterChangeAcknowledged 消息后，将 ApplicationInfo 的状态修改为 WAITING，然后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

序号⑧：beginRecovery 方法对于从持久化引擎中读取的 WorkerInfo，调用 registerWorker 方法将 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中，然后将 WorkerInfo 的状态设置为 UNKNOWN。

序号⑨：beginRecovery 方法还将向 Worker 发送 MasterChanged 消息。

序号⑩：Worker 接收到 MasterChanged 消息后，将自身的 activeMasterUrl、activeMasterWebUiUrl、master 等属性修改为当前 Master 的对应信息，然后将 connected 设置为 true，最后向 Master 发送 WorkerSchedulerStateResponse 消息。Master 接收到 WorkerSchedulerStateResponse 消息后，首先将 WorkerInfo 的状态修改为 ALIVE，然后对此 Worker 上的

Executor 和 Driver 也进行恢复，最后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

序号⑪：Master 1 和 Master 2 参加领导选举，最后 Master 1 胜出，于是 LeaderLatch 将调用 Master 2 的 ZooKeeperLeaderElectionAgent 的 notLeader 方法，notLeader 方法实际调用了 Master 2 的 revokedLeadership 方法。

### 9.7.5 运行 Driver

在介绍 Master 对 Driver 的资源调度和运行时，我们知道 Master 将向 Worker 发送 LaunchDriver 消息以运行 Driver。下面一起来看 Worker 接收到 LaunchDriver 消息后，是如何运行 Driver 的。

Worker 的 receive 方法中实现了对 LaunchDriver 消息的处理，代码如下。

```
case LaunchDriver(driverId, driverDesc) =>
  val driver = new DriverRunner(conf, driverId, workDir, sparkHome,
    driverDesc.copy(command = Worker.maybeUpdateSSLSettings(driverDesc.command,
      conf)),
    self, workerUri, securityMgr)
  drivers(driverId) = driver
  driver.start()

  coresUsed += driverDesc.cores
  memoryUsed += driverDesc.mem
```

根据上述代码，我们知道 Worker 处理 LaunchDriver 消息的步骤如下。

- 1 ) 创建 DriverRunner。
- 2 ) 将 Driver 的身份标识与 DriverRunner 的关系放入 drivers 中缓存。
- 3 ) 调用 DriverRunner 的 start 方法（见代码清单 9-61）启动 DriverRunner。
- 4 ) 更新 Worker 已经使用的内核数和内存大小。

代码清单9-61 DriverRunner的start方法

---

```
private[worker] def start() = {
  new Thread("DriverRunner for " + driverId) {
    override def run() {
      var shutdownHook: AnyRef = null
      try {
        // 省略关闭钩子的代码，关闭钩子用于在JVM进程退出前“杀死”Driver进程
        val exitCode = prepareAndRunDriver()

        finalState = if (exitCode == 0) {
          Some(DriverState.FINISHED)
        } else if (killed) {
          Some(DriverState.KILLED)
        } else {
          Some(DriverState.FAILED)
        }
      } catch {
```

```

        // 省略异常处理的代码，发生异常时将“杀死”Driver进程
    } finally {
        // 省略移除钩子的代码
    }
    worker.send(DriverStateChanged(driverId, finalState.get, finalException))
}
}.start()
}

```

---

根据代码清单 9-61，DriverRunner 的 start 方法的核心在于 prepareAndRunDriver 方法。prepareAndRunDriver 方法的实现如代码清单 9-62 所示。

代码清单9-62 prepareAndRunDriver方法的实现

```

private[worker] def prepareAndRunDriver(): Int = {
    val driverDir = createWorkingDirectory()
    val localJarFilename = downloadUserJar(driverDir)

    def substituteVariables(argument: String): String = argument match {
        case "{{WORKER_URL}}" => workerUrl
        case "{{USER_JAR}}" => localJarFilename
        case other => other
    }
    val builder = CommandUtils.buildProcessBuilder(driverDesc.command, security-
        Manager,
        driverDesc.mem, sparkHome.getAbsolutePath, substituteVariables)

    runDriver(builder, driverDir, driverDesc.supervise)
}

```

---

根据代码清单 9-62，prepareAndRunDriver 方法的执行步骤如下。

- 1) 创建 Driver 的工作目录。
- 2) 下载用户指定的 Jar 文件。
- 3) 构造 ProcessBuilder，进程命令将由 DriverDescription 的 command 属性决定。ProcessBuilder 执行 java 命令的主类是 org.apache.spark.deploy.worker.DriverWrapper，有关 buildProcessBuilder 的实现，请参阅附录 F。在调用 CommandUtils 的 buildProcessBuilder 方法时将 substituteVariables 方法作为了参数，substituteVariables 方法的作用是将 DriverDescription 的 command 属性（即 Command）的 arguments 属性中每个参数中的变量替换为指定的值。例如，将 {{WORKER\_URL}} 替换为 DriverRunner 的 workerUrl 属性。
- 4) 调用 runDriver 方法运行 Driver。由于在 Standalone 集群中运行 Driver 的方式已经被抛弃，并会在将来从 Spark 中移除，所以对运行 Driver 的分析就讲到这里。对 runDriver 方法感兴趣的读者可以自行研究。

## 9.7.6 运行 Executor

在 9.6.4 节介绍 Master 的资源调度时，我们知道 Master 向 Worker 发送 LaunchExecutor 消息以运行 Executor。LaunchExecutor 消息携带着 masterUrl、Application 的 ID、Executor

的 ID、应用的描述信息、Executor 分配获得的内核数、Executor 分配获得的内存大小等信息。Worker 在接收到 LaunchExecutor 消息后的处理如下。

```

case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_, memory_) =>
  if (masterUrl != activeMasterUrl) {
    logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")
  } else {
    try {
      logInfo("Asked to launch executor %s/%d for %s".format(appId, execId, appDesc.name))

      val executorDir = new File(workDir, appId + "/" + execId)
      if (!executorDir.mkdirs()) {
        throw new IOException("Failed to create directory " + executorDir)
      }

      val appLocalDirs = appDirectories.getOrElse(appId,
        Utils.getOrCreateLocalRootDirs(conf).map { dir =>
          val appDir = Utils.createDirectory(dir, namePrefix = "executor")
          Utils.chmod700(appDir)
          appDir.getAbsolutePath()
        }.toSeq)
      appDirectories(appId) = appLocalDirs
      val manager = new ExecutorRunner(appId, execId,
        appDesc.copy(command = Worker.maybeUpdateSSLSettings(appDesc.command, conf)),
        cores_, memory_, self, workerId, host, webUi.boundPort, publicAddress,
        sparkHome,
        executorDir, workerUri, conf, appLocalDirs, ExecutorState.RUNNING)
      executors(appId + "/" + execId) = manager
      manager.start()
      coresUsed += cores_
      memoryUsed += memory_
      sendToMaster(ExecutorStateChanged(appId, execId, manager.state, None, None))
    } catch {
      case e: Exception =>
        logError(s"Failed to launch executor $appId/$execId for ${appDesc.name}.", e)
        if (executors.contains(appId + "/" + execId)) {
          executors(appId + "/" + execId).kill()
          executors -= appId + "/" + execId
        }
        sendToMaster(ExecutorStateChanged(appId, execId, ExecutorState.FAILED,
          Some(e.toString), None))
    }
  }
}

```

根据上述代码，Worker 接收到 LaunchExecutor 消息后，首先判断发送 LaunchExecutor 消息的 Master 是否是激活的 Master，如果不是，则什么都不做，否则进行以下处理。

- 1) 在 Worker 的工作目录下创建 Executor 的工作目录。
- 2) 给 Executor 创建本地目录，并将这些目录缓存到 Worker 的 appDirectories 属性中。这些目录将通过环境变量 SPARK\_EXECUTOR\_DIRS 传递给 Executor，并在 Application 完

成后由 Worker 删除。

3) 创建 ExecutorRunner，并将 ExecutorRunner 放入 executors。例如，假设 appId 为 app-20170528125247-0000，execId 为 0，那么将 app-20170528125247-0000/0 与 ExecutorRunner 的映射关系放入了 executors。在创建完 ExecutorRunner 时，ExecutorRunner 的状态为 RUNNING。

- 4) 调用 ExecutorRunner 的 start 方法（见代码清单 9-63）启动 ExecutorRunner。
- 5) 更新已经使用的内核数 coresUsed 和已经使用的内存大小 memoryUsed。
- 6) 向 Master 发送 ExecutorStateChanged 消息以更新 Executor 的状态。

代码清单9-63 ExecutorRunner的start方法

---

```
private[worker] def start() {
    workerThread = new Thread("ExecutorRunner for " + fullId) {
        override def run() { fetchAndRunExecutor() }
    }
    workerThread.start()
    // 省略关闭钩子的代码，关闭钩子用于在JVM进程退出前“杀死”Executor进程
}
```

---

根据代码清单 9-63，ExecutorRunner 的 start 方法的核心在于 workerThread，workerThread 在运行时将执行 fetchAndRunExecutor 方法。fetchAndRunExecutor 方法的实现如代码清单 9-64 所示。

代码清单9-64 fetchAndRunExecutor方法的实现

---

```
private def fetchAndRunExecutor() {
    try {
        val builder = CommandUtils.buildProcessBuilder(appDesc.command, new SecurityManager(conf),
            memory, sparkHome.getAbsolutePath, substituteVariables) // 构造ProcessBuilder
        val command = builder.command()
        val formattedCommand = command.asScala.mkString("\\", "\\", "\\\"", "\\\"")
        logInfo(s"Launch command: $formattedCommand")
        // 为ProcessBuilder设置执行目录和环境变量
        builder.directory(executorDir)
        builder.environment.put("SPARK_EXECUTOR_DIRS", appLocalDirs.mkString(File.separator))
        builder.environment.put("SPARK_LAUNCH_WITH_SCALA", "0")
        val baseUrl =
            if (conf.getBoolean("spark.ui.reverseProxy", false)) {
                s"/proxy/$workerId/logPage/?appId=$appId&executorId=$execId&logType="
            } else {
                s"http://$publicAddress:$webUiPort/logPage/?appId=$appId&executorId=$execId&logType="
            }
        builder.environment.put("SPARK_LOG_URL_STDERR", s"${baseUrl}stderr")
        builder.environment.put("SPARK_LOG_URL_STDOUT", s"${baseUrl}stdout")

        process = builder.start() // 启动ProcessBuilder，生成进程
        val header = "Spark Executor Command: %s\n%s\n%n".format(
```

```

        formattedCommand, " = " * 40)

    val stdout = new File(executorDir, "stdout") // 重定向进程的输出流与错误流
    stdoutAppender = FileAppender(process.getInputStream, stdout, conf)
    val stderr = new File(executorDir, "stderr")
    Files.write(header, stderr, StandardCharsets.UTF_8)
    stderrAppender = FileAppender(process.getErrorStream, stderr, conf)
    val exitCode = process.waitFor() // 等待获取进程的退出状态
    state = ExecutorState.EXITED
    val message = "Command exited with code " + exitCode
    worker.send(ExecutorStateChanged(appId, execId, state, Some(message), Some
        (exitCode)))
} catch {
    case interrupted: InterruptedException =>
        logInfo("Runner thread for executor " + fullId + " interrupted")
        state = ExecutorState.KILLED
        killProcess(None)
    case e: Exception =>
        logError("Error running executor", e)
        state = ExecutorState.FAILED
        killProcess(Some(e.toString))
}
}

```

---

根据代码清单 9-64，fetchAndRunExecutor 方法的执行步骤如下。

- 1) 构造 ProcessBuilder，进程命令将由 ApplicationDescription 的 command 属性决定。ApplicationDescription 的 command 属性是何时产生的呢？本书将在 9.9.3 节介绍。进程命令的主类是 org.apache.spark.executor.CoarseGrainedExecutorBackend（将在 9.10 节详细介绍其实现），有关 buildProcessBuilder 的实现，请参阅附录 F。在调用 CommandUtils 的 buildProcessBuilder 方法时将 substituteVariables 方法作为参数，substituteVariables 方法（见代码清单 9-65）的作用是将 ApplicationDescription 的 command 属性（即 Command）的 arguments 属性中每个参数中的变量替换为指定的值。例如，将 {WORKER\_URL} 替换为 ExecutorRunner 的 workerUrl 属性。
- 2) 将 executorDir 设置为 ProcessBuilder 执行目录。这里的 executorDir 就是 Worker 在接收到 LaunchExecutor 消息后创建的那个 executorDir。
- 3) 为 ProcessBuilder 设置环境变量。
- 4) 启动 ProcessBuilder，生成进程。一旦执行了 builder.start()，我们利用 jdk 自带的工具 VisualVM，就会发现此时已经产生了一个新的进程，笔者对本地的 CoarseGrained-ExecutorBackend 进程的截图如图 9-19 所示。CoarseGrainedExecutorBackend 进程的系统属性如图 9-20 所示。
- 5) 重定向进程的输出流与错误流为 executorDir 目录下的文件 stdout 与 stderr。以笔者本地为例，分别为 D:\git\spark\work\app-20150707144151-0000\0\stdout 和 D:\git\spark\work\app-20150707144151-0000\0\stderr。
- 6) 等待获取进程的退出状态，一旦收到退出状态，则向 Worker 发送 Executor-

StateChanged 消息。



图 9-19 CoarseGrainedExecutorBackend 进程的截图

#### 代码清单9-65 替换命令中的变量

```
private[worker] def substituteVariables(argument: String): String = argument match {
    case "{{WORKER_URL}}" => workerUrl
    case "{{EXECUTOR_ID}}" => execId.toString
    case "{{HOSTNAME}}" => host
    case "{{CORES}}" => cores.toString
    case "{{APP_ID}}" => appId
    case other => other
}
```

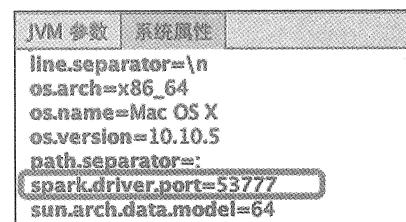


图 9-20 CoarseGrainedExecutorBackend 进程的系统属性

根据代码清单 9-64，如果在执行 fetchAndRunExecutor 方法的过程中捕获到异常，将调用 killProcess 方法。killProcess 方法的实现如代码清单 9-66 所示。

根据代码清单 9-66，killProcess 方法在“杀死”CoarseGrainedExecutorBackend 进程后将向 Worker 发送 ExecutorStateChanged 消息，有关 ExecutorStateChanged 消息的处理将在下一小节介绍。

#### 代码清单9-66 “杀死” CoarseGrainedExecutorBackend进程

```
private def killProcess(message: Option[String]) {
    var exitCode: Option[Int] = None
    if (process != null) {
        logInfo("Killing process!")
        if (stdoutAppender != null) {
            stdoutAppender.stop()
        }
        if (stderrAppender != null) {
            stderrAppender.stop()
        }
        exitCode = Utils.terminateProcess(process, EXECUTOR_TERMINATE_TIMEOUT_MS)
        if (exitCode.isEmpty) {
    }}
```

```

        logWarning("Failed to terminate process: " + process +
            ". This process will likely be orphaned.")
    }
}
try {
    worker.send(ExecutorStateChanged(appId, execId, state, message, exitCode))
} catch {
    case e: IllegalStateException => logWarning(e.getMessage(), e)
}
}

```

---

### 9.7.7 处理 Executor 的状态变化

Worker 接收 ExecutorStateChanged 消息，以对 Executor 的状态发生变化后进行处理。Worker 处理 ExecutorStateChanged 消息的代码如下。

```

case executorStateChanged @ ExecutorStateChanged(appId, execId, state, message,
    exitStatus) =>
handleExecutorStateChanged(executorStateChanged)

```

根据上述代码，Worker 将调用 handleExecutorStateChanged 方法处理 Executor 的状态变化。handleExecutorStateChanged 方法的实现如代码清单 9-67 所示。

代码清单9-67 处理ExecutorStateChanged消息

---

```

private[worker] def handleExecutorStateChanged(executorStateChanged: Executor-
    StateChanged):
    Unit = {
        sendToMaster(executorStateChanged)
        val state = executorStateChanged.state
        if (ExecutorState.isFinished(state)) {
            val appId = executorStateChanged.appId
            val fullId = appId + "/" + executorStateChanged.execId
            val message = executorStateChanged.message
            val exitStatus = executorStateChanged.exitStatus
            executors.get(fullId) match {
                case Some(executor) =>
                    logInfo("Executor " + fullId + " finished with state " + state +
                        message.map(" message " + _).getOrElse("") +
                        exitStatus.map(" exitStatus " + _).getOrElse(""))
                    executors -= fullId
                    finishedExecutors(fullId) = executor
                    trimFinishedExecutorsIfNecessary()
                    coresUsed -= executor.cores
                    memoryUsed -= executor.memory
                case None =>
                    logInfo("Unknown Executor " + fullId + " finished with state " + state +
                        message.map(" message " + _).getOrElse("") +
                        exitStatus.map(" exitStatus " + _).getOrElse(""))
            }
            maybeCleanupApplication(appId)
        }
    }
}

```

---

根据代码清单 9-67，handleExecutorStateChanged 方法除了更新 Worker 中与此 Executor 有关的缓存信息外，还向 Master 转发了 ExecutorStateChanged 消息。Master 对 ExecutorStateChanged 消息的处理已在 9.6.10 节详细介绍过。

## 9.8 StandaloneAppClient 实现

StandaloneAppClient 是在 Standalone 模式下，Application 与集群管理器进行对话的客户端。深入分析 StandaloneAppClient 之前，我们还是先来了解它的属性。

- rpcEnv：即 SparkContext 的 SparkEnv 的 RpcEnv。
- masterUrls：用于缓存每个 Master 的 spark://host:port 格式的 URL 的数组。
- appDescription：Application 的描述信息（ApplicationDescription）。ApplicationDescription 中记录了 Application 的名称（name）、Application 需要的最大内核数（maxCores）、每个 Executor 所需要的内存大小（memoryPerExecutorMB）、运行 CoarseGrainedExecutorBackend 进程的命令（command）、Spark UI 的 URL（appUiUrl）、事件日志的路径（eventLogDir）、事件日志采用的压缩算法名（eventLogCodec）、每个 Executor 所需的内核数（coresPerExecutor）、提交 Application 的用户名（user）等信息。
- listener：类型为 StandaloneAppClientListener，是对集群事件的监听器。StandaloneAppClientListener 有两个实现类，分别是 StandaloneSchedulerBackend 和 AppClientCollector。AppClientCollector 只用于测试，StandaloneSchedulerBackend 可用在 local-cluster 或 Standalone 模式下。StandaloneSchedulerBackend 在构造 StandaloneAppClient 时会将自身作为 listener 参数。
- conf：即 SparkConf。
- masterRpcAddresses：Master 的 RPC 地址（RpcAddress）。
- REGISTRATION\_TIMEOUT\_SECONDS：注册超时的秒数，固定为 20。
- REGISTRATION\_RETRIES：注册的重试次数，固定为 3。
- endpoint：类型为 AtomicReference，用于持有 ClientEndpoint 的 RpcEndpointRef。本书在 9.8.1 节将详细分析 ClientEndpoint 的实现。



此处所介绍的 ClientEndpoint 是 org.apache.spark.deploy.client.ClientEndpoint，而不是 org.apache.spark.deploy.ClientEndpoint。

- appId：类型为 AtomicReference，用于持有 Application 的 ID。
- registered：类型为 AtomicReference，用于标识是否已经将 Application 注册到 Master。

有了对 StandaloneAppClient 中包含属性的了解，下面首先分析 ClientEndpoint，然后介绍 StandaloneAppClient 的功能。

### 9.8.1 ClientEndpoint 的实现分析

我们刚才说 StandaloneAppClient 是在 Standalone 模式下，Application 与集群进行对话的客户端，而 Spark 各个组件之间的通信离不开 RpcEnv 及 RpcEndpoint。ClientEndpoint 继承自 ThreadSafeRpcEndpoint，也是 StandaloneAppClient 的内部类，StandaloneAppClient 依赖于 ClientEndpoint 与集群管理器进行通信。

ClientEndpoint 的属性如下。

- ❑ rpcEnv：即 SparkContext 的 SparkEnv 的 RpcEnv。
- ❑ master：处于激活状态的 Master 的 RpcEndpointRef。
- ❑ alreadyDisconnected：是否已经与 Master 断开连接。此属性用于防止多次调用 StandaloneAppClientListener 的 disconnected 方法。
- ❑ alreadyDead：表示 ClientEndpoint 是否已经“死掉”。此属性用于防止多次调用 StandaloneAppClientListener 的 dead 方法。
- ❑ registerMasterThreadPool：用于向 Master 注册 Application 的 ThreadPoolExecutor。registerMasterThreadPool 的线程池大小为外部类 StandaloneAppClient 的 masterRpcAddresses 数组的大小，启动的线程以 appclient-register-master-threadpool 为前缀。
- ❑ registerMasterFutures：用于保存 registerMasterThreadPool 执行的向各个 Master 注册 Application 的任务返回的 Future。
- ❑ registrationRetryThread：类型为 ScheduledExecutorService，用于对 Application 向 Master 进行注册的重试。由 registrationRetryThread 启动的线程名称为 appclient-registration-retry-thread。
- ❑ registrationRetryTimer：用于持有向 registrationRetryThread 提交关于注册的定时调度返回的 ScheduledFuture。

了解了 ClientEndpoint 中包含的属性，现在来看看 ClientEndpoint 中的重要功能。

#### 1. ClientEndpoint 的启动

将 ClientEndpoint 注册到 RpcEnv 的 Dispatcher 时，会触发对 ClientEndpoint 的 onStart 方法的调用。ClientEndpoint 的 onStart 方法的实现如代码清单 9-68 所示。

代码清单9-68 ClientEndpoint的onStart方法

---

```
override def onStart(): Unit = {
    try {
        registerWithMaster(1)
    } catch {
        case e: Exception =>
            logWarning("Failed to connect to master", e)
            markDisconnected()
            stop()
    }
}
```

---

根据代码清单 9-68，ClientEndpoint 的 onStart 方法将尝试调用 registerWithMaster 方法（见代码清单 9-73）向 Master 注册 Application。如果执行 registerWithMaster 的过程中捕获到异常，那么会调用 markDisconnected 方法（见代码清单 9-69），将当前 ClientEndpoint 标记为和 Master 断开连接，并且调用 StandaloneAppClient 的 stop 方法（见代码清单 9-78）停止 StandaloneAppClient。

代码清单9-69 markDisconnected方法的实现

---

```
def markDisconnected() {
    if (!alreadyDisconnected) {
        listener.disconnected()
        alreadyDisconnected = true
    }
}
```

---

根据代码清单 9-69，markDisconnected 方法在判断 ClientEndpoint 的 alreadyDisconnected 状态不为 true 时，先调用外部类 StandaloneAppClient 的 listener 属性（即 StandaloneAppClientListener）的 disconnected 方法，然后将 alreadyDisconnected 设置为 true。

## 2. ClientEndpoint 的消息处理

ClientEndpoint 的 receive 方法能够接收 RegisteredApplication、ApplicationRemoved、ExecutorAdded、ExecutorUpdated、MasterChanged 等消息，如代码清单 9-70 所示。

代码清单9-70 ClientEndpoint的receive方法

---

```
override def receive: PartialFunction[Any, Unit] = {
    case RegisteredApplication(appId_, masterRef) =>
        appId.set(appId_)
        registered.set(true)
        master = Some(masterRef)
        listener.connected(appId.get)

    case ApplicationRemoved(message) =>
        markDead("Master removed our application: %s".format(message))
        stop()

    case ExecutorAdded(id: Int, workerId: String, hostPort: String, cores: Int,
                      memory: Int) =>
        val fullId = appId + "/" + id
        logInfo("Executor added: %s on %s (%s) with %d cores".format(fullId, workerId,
                                                                      hostPort,
                                                                      cores))
        listener.executorAdded(fullId, workerId, hostPort, cores, memory)

    case ExecutorUpdated(id, state, message, exitStatus, workerLost) =>
        val fullId = appId + "/" + id
        val messageText = message.map(s => " (" + s + ")").getOrElse("")
        logInfo("Executor updated: %s is now %s%s".format(fullId, state, messageText))
        if (ExecutorState.isFinished(state)) {
            listener.executorRemoved(fullId, message.getOrElse(""), exitStatus, workerLost)
        }
}
```

```

case MasterChanged(masterRef, masterWebUiUrl) =>
  logInfo("Master has changed, new master is at " + masterRef.address.toSparkURL)
  master = Some(masterRef)
  alreadyDisconnected = false
  masterRef.send(MasterChangeAcknowledged(appId.get))
}

```

根据代码清单 9-70 可以看到，ClientEndpoint 接收各种消息的处理都非常简单，这里就不赘述了。

除了 receive 方法，ClientEndpoint 的 receiveAndReply 方法还能够处理 StopAppClient、RequestExecutors、KillExecutors 三种消息，如代码清单 9-71 所示。

代码清单9-71 ClientEndpoint的receiveAndReply方法

```

override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case StopAppClient =>
    markDead("Application has been stopped.")
    sendToMaster(UnregisterApplication(appId.get))
    context.reply(true)
    stop()

  case r: RequestExecutors =>
    master match {
      case Some(m) => askAndReplyAsync(m, context, r)
      case None =>
        logWarning("Attempted to request executors before registering with Master.")
        context.reply(false)
    }

  case k: KillExecutors =>
    master match {
      case Some(m) => askAndReplyAsync(m, context, k)
      case None =>
        logWarning("Attempted to kill executors before registering with Master.")
        context.reply(false)
    }
}

```

根据代码清单 9-71，接收到 StopAppClient 消息后，ClientEndpoint 最重要的一步是向 Master 发送 UnregisterApplication 消息。对于请求 Executor (RequestExecutors) 和“杀死” Executor (KillExecutors) 两种消息，ClientEndpoint 将通过调用 askAndReplyAsync 方法（见代码清单 9-72）向 Master 转发这两种消息。

代码清单9-72 askAndReplyAsync方法的实现

```

private def askAndReplyAsync[T](
  endpointRef: RpcEndpointRef,
  context: RpcCallContext,
  msg: T): Unit = {
  endpointRef.ask[Boolean](msg).andThen {
    case Success(b) => context.reply(b)
    case Failure(ie: InterruptedException) => // Cancelled
  }
}

```

```

        case Failure(NonFatal(t)) => context.sendFailure(t)
    }(ThreadUtils.sameThread)
}

```

### 3. 向 Master 注册应用信息

registerWithMaster 方法用于向 Master 注册 Application，其实现如代码清单 9-73 所示。

代码清单 9-73 向 Master 注册应用信息

```

private def registerWithMaster(nthRetry: Int) {
    registerMasterFutures.set(tryRegisterAllMasters()) // 向所有的Master尝试注册
    Application
    registrationRetryTimer.set(registrationRetryThread.schedule(new Runnable {
        override def run(): Unit = {
            if (registered.get) { // 如果已经注册成功，那么取消向Master注册Application
                registerMasterFutures.get.foreach(_.cancel(true))
                registerMasterThreadPool.shutdownNow()
            } else if (nthRetry >= REGISTRATION_RETRIES) { // 重试次数超过限制，标记
                ClientEndpoint死亡
                markDead("All masters are unresponsive! Giving up.")
            } else {
                registerMasterFutures.get.foreach(_.cancel(true))
                registerWithMaster(nthRetry + 1) // 向Master注册Application的重试
            }
        }
    }, REGISTRATION_TIMEOUT_SECONDS, TimeUnit.SECONDS))
}

```

根据代码清单 9-73，registerWithMaster 方法的执行步骤如下。

- 1) 调用 tryRegisterAllMasters 方法（见代码清单 9-74）向所有的 Master 尝试注册 Application，并将返回的 Future 保存到 registerMasterFutures。
- 2) 向 registrationRetryThread 提交定时调度，调度间隔时长为 REGISTRATION\_TIMEOUT\_SECONDS。调度任务的执行逻辑是：如果已经注册成功，那么调用 registerMasterFutures 中保存的每一个 Future 的 cancel 方法取消向 Master 注册 Application；如果重试次数超过了 REGISTRATION\_RETRIES 的限制，那么调用 markDead 方法（见代码清单 9-75）标记当前 ClientEndpoint 进入死亡状态；其他情况下首先调用 registerMasterFutures 中保存的每一个 Future 的 cancel 方法取消向 Master 注册 Application，然后再次调用 registerWithMaster（这次调用将把重试次数加 1）。

代码清单 9-74 tryRegisterAllMasters 方法的实现

```

private def tryRegisterAllMasters(): Array[JFuture[_]] = {
    for (masterAddress <- masterRpcAddresses) yield {
        registerMasterThreadPool.submit(new Runnable {
            override def run(): Unit = try {
                if (registered.get) {
                    return
                }
                logInfo("Connecting to master " + masterAddress.toSparkURL + "...")
            }
        })
    }
}

```

```
    val masterRef = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT_NAME)
    masterRef.send(RegisterApplication(appDescription, self))
} catch {
    case ie: InterruptedException => // Cancelled
    case NonFatal(e) => logWarning(s"Failed to connect to master $master-
        Address", e)
}
})
}
}
```

根据代码清单 9-74, tryRegisterAllMasters 方法通过 masterRpcAddresses 中每个 Master 的 RpcAddress, 得到每个 Master 的 RpcEndpointRef, 然后通过这些 RpcEndpointRef 向每个 Master 发送 RegisterApplication 消息 (此消息携带外部类 StandaloneAppClient 的 appDescription 属性和 ClientEndpoint 自身的 RpcEndpointRef)。

代码清单9-75 markDead的实现

```
def markDead(reason: String) {
    if (!alreadyDead.get) {
        listener.dead(reason)
        alreadyDead.set(true)
    }
}
```

根据 9.6.8 节的内容我们知道, 当注册 Application 成功后, Master 将向 ClientEndpoint 回复 RegisteredApplication 消息, 那么 ClientEndpoint 是如何处理 RegisteredApplication 消息的呢? 代码清单 9-70 展示了 ClientEndpoint 对 RegisteredApplication 消息的处理实现, 具体处理包括: 将 Master 分配的 Application ID 设置到 appId 中, 将 registered 设置为 true, 将 Master 的 RpcEndpointRef 更新到 master、调用 StandaloneAppClientListener (实际为 StandaloneSchedulerBackend) 的 connected 方法等。

StandaloneSchedulerBackend 的 connected 方法的实现如代码清单 9-76 所示。

代码清单9-76 StandaloneSchedulerBackend的connected方法

```
override def connected(appId: String) {
    logInfo("Connected to Spark cluster with app ID " + appId)
    this.appId = appId
    notifyContext()
    launcherBackend.setappId(appId)
}
```

根据代码清单 9-76, StandaloneSchedulerBackend 的 connected 方法首先将设置 appId 为 Master 分配的 Application ID, 然后调用 notifyContext 方法 (见代码清单 9-77) 释放信号量, 最后调用 LauncherBackend 的 setappId 方法向 LaunchServer 发送 appId。

代码清单9-77 notifyContext方法的实现

```
private def notifyContext() = {
```

---

```
    registrationBarrier.release()
}
```

---

## 9.8.2 StandaloneAppClient 的实现分析

StandaloneAppClient 是 Application 与集群管理器进行对话的客户端。StandaloneAppClient 启动后才能正常工作，当不再需要与集群管理器通信时，需要停止它。StandaloneAppClient 最为核心的功能是向集群管理器请求或“杀死”Executor。下面将对这些内容一一进行介绍。

### 1. StandaloneAppClient 的启动

StandaloneAppClient 的 start 方法用于启动 StandaloneAppClient，代码如下。

```
def start() {
    endpoint.set(rpcEnv.setupEndpoint("AppClient", new ClientEndpoint(rpcEnv)))
}
```

根据上述代码，在启动 StandaloneAppClient 时只做了一件事——向 SparkContext 的 SparkEnv 的 RpcEnv 注册 ClientEndpoint，进而引起对 ClientEndpoint 的启动和向 Master 注册 Application。

### 2. StandaloneAppClient 的停止

StandaloneAppClient 的 stop 方法用于停止 StandaloneAppClient，如代码清单 9-78 所示。

代码清单9-78 StandaloneAppClient的stop方法

---

```
def stop() {
    if (endpoint.get != null) {
        try {
            val timeout = RpcUtils.askRpcTimeout(conf)
            timeout.awaitResult(endpoint.get.ask[Boolean](StopAppClient))
        } catch { // 忽略捕获到TimeoutException，打印日志的代码 }
        endpoint.set(null)
    }
}
```

---

根据 stop 方法的实现，我们知道停止 StandaloneAppClient 实际不过是向 ClientEndpoint 发送了 StopAppClient 消息，并将 ClientEndpoint 的 RpcEndpointRef 从 endpoint 中清除。

### 3. 请求 Executor 资源

StandaloneAppClient 的 requestTotalExecutors 方法（见代码清单 9-79）用于向 Master 请求所需的所有 Executor 资源。

代码清单9-79 requestTotalExecutors的实现

---

```
def requestTotalExecutors(requestedTotal: Int): Future[Boolean] = {
    if (endpoint.get != null && appId.get != null) {
        endpoint.get.ask[Boolean](RequestExecutors(appId.get, requestedTotal))
    } else {
```

```

        logWarning("Attempted to request executors before driver fully initialized.")
        Future.successful(false)
    }
}

```

根据代码清单 9-79，requestTotalExecutors 方法只能在 Application 注册成功之后调用。requestTotalExecutors 方法将向 ClientEndpoint 发送 RequestExecutors 消息（此消息携带 Application ID 和所需的 Executor 总数）。根据对 ClientEndpoint 处理 RequestExecutors 消息的分析，我们知道 ClientEndpoint 将向 Master 转发 RequestExecutors 消息。Master 处理 RequestExecutors 消息的内容请回顾 9.6.9 节。

#### 4. “杀死” Executor

StandaloneAppClient 的 killExecutors 方法（见代码清单 9-80）用于向 Master 请求“杀死” Executor。

代码清单9-80 killExecutors的实现

```

def killExecutors(executorIds: Seq[String]): Future[Boolean] = {
    if (endpoint.get != null && appId.get != null) {
        endpoint.get.ask[Boolean](KillExecutors(appId.get, executorIds))
    } else {
        logWarning("Attempted to kill executors before driver fully initialized.")
        Future.successful(false)
    }
}

```

根据代码清单 9-80，killExecutors 方法只能在 Application 注册成功之后调用。killExecutors 方法将向 ClientEndpoint 发送 KillExecutors 消息（此消息携带 Application ID 和要“杀死”的 Executor 的 ID）。

## 9.9 StandaloneSchedulerBackend 的实现分析

在 7.8.2 节我们曾经介绍了 local 部署模式下，SchedulerBackend 的实现类 Local-Scheduler-Backend，本节将介绍在 local-cluster 模式和 Standalone 模式下，SchedulerBackend 的另一个实现类 StandaloneSchedulerBackend。由于 StandaloneSchedulerBackend 继承自 Coarse-GrainedSchedulerBackend，本节还需要介绍 CoarseGrainedSchedulerBackend 及其与其他组件通信的内部类 DriverEndpoint。

### 9.9.1 StandaloneSchedulerBackend 的属性

为便于分析，本节首先来熟悉 StandaloneSchedulerBackend 包含的属性。Standalone-SchedulerBackend 的属性包括从父类 CoarseGrainedSchedulerBackend 继承的属性，也包括自身的属性。从 CoarseGrainedSchedulerBackend 继承的属性如下。

- ❑ scheduler：即 TaskSchedulerImpl。
- ❑ rpcEnv：即 RpcEnv。
- ❑ totalCoreCount：用于统计分配给 Driver 的内核总数。
- ❑ totalRegisteredExecutors：当前注册到 CoarseGrainedSchedulerBackend 的 Executor 的总数。
- ❑ conf：即 SparkConf。
- ❑ maxRpcMessageSize：RPC 消息的最大大小。可通过 spark.rpc.message.maxSize 属性配置，默认为 128MB。
- ❑ defaultAskTimeout：类型为 RpcTimeout，表示 RPC 请求的超时时间。可通过 spark.rpc.askTimeout 属性或 spark.network.timeout 属性配置，默认为 120s。
- ❑ \_minRegisteredRatio：已经注册的资源与期望得到的资源之间的最小比值，当比值大于等于 \_minRegisteredRatio 时，才提交 Task。可通过 spark.scheduler.minRegisteredResourcesRatio 属性配置，默认为 0。
- ❑ maxRegisteredWaitingTimeMs：在还未达到 \_minRegisteredRatio 时，如果已经等待了超过 maxRegisteredWaitingTimeMs 指定的时间，那么提交 Task。可通过 spark.scheduler.maxRegisteredResourcesWaitingTime 属性配置，默认为 30s。
- ❑ createTime：CoarseGrainedSchedulerBackend 的创建时间。
- ❑ executorDataMap：Executor 的 ID 与 ExecutorData 之间的映射关系缓存。ExecutorData 保存了 Executor 的 RpcEndpointRef、RpcAddress、Host、Executor 的空闲内核数（freeCores）、Executor 的总内核数（totalCores）等信息。
- ❑ numPendingExecutors：从集群管理器请求的还未注册到 CoarseGrainedSchedulerBackend 的 Executor 的数量。
- ❑ listenerBus：即 LiveListenerBus。
- ❑ executorsPendingToRemove：请求集群管理器 kill 一个 Executor 时，Executor 并不会立即被“杀死”，所以 executorsPendingToRemove 缓存那些请求 kill 的 Executor 的 ID 与是否被“杀死”之间的映射关系。
- ❑ hostToLocalTaskCount：缓存机器的 Host 和在机器本地运行的 Task 数量之间的映射关系。
- ❑ localityAwareTasks：用于统计有本地性需求的 Task 的数量。
- ❑ currentExecutorIdCounter：用于保存注册到 executorDataMap 的最大的 Executor 的 ID。再来看看 StandaloneSchedulerBackend 自身的属性。
- ❑ client：即 StandaloneAppClient。
- ❑ stopping：标记 StandaloneSchedulerBackend 是否正在停止。
- ❑ launcherBackend：即运行器后端接口 LauncherBackend。具体实现与 LocalSchedulerBackend 的 launcherBackend 属性的实现类似。

- ❑ appId：Application 的 ID。
- ❑ registrationBarrier：使用 Java 的信号量（Semaphore）实现的栅栏，用于等待 Application 向 Master 注册完成后，将 Application 的当前状态（此时为正在运行，即 RUNNING）告知 LauncherServer。
- ❑ maxCores：Application 可以申请获得的最大内核数。可通过 spark.cores.max 属性配置。
- ❑ totalExpectedCores：Application 期望获得的内核数，如果设置了 maxCores，则为 maxCores，否则为 0。

### 9.9.2 DriverEndpoint 的实现分析

DriverEndpoint 是 CoarseGrainedSchedulerBackend 的内部类，无论 CoarseGrainedSchedulerBackend 还是 StandaloneSchedulerBackend，都借助于 DriverEndpoint 和其他组件通信，所以我们先来分析它。

DriverEndpoint 包含以下属性。

- ❑ rpcEnv：即 SparkContext 的 SparkEnv 中的 RpcEnv。
- ❑ sparkProperties：从 SparkConf 中获取的所有以 spark. 开头的属性信息。
- ❑ executorsPendingLossReason：已经丢失的（但是还不知道真实的退出原因）的 Executor 的 ID。
- ❑ ser：SparkEnv 的子组件 closureSerializer 的实例，用于对闭包序列化。
- ❑ addressToExecutorId：每个 Executor 的 RpcEnv 的地址与 Executor 的 ID 之间的映射关系。
- ❑ reviveThread：只有一个线程的 ScheduledThreadPoolExecutor，用于唤起对延迟调度的 Task 进行资源分配与调度。

DriverEndpoint 中有很多功能实现，笔者挑选其中最为重要的启动、Task 状态更新、注册 Executor 进行分析，其他的功能（如停止 Driver、停止 Executor、移除 Executor 等）留给感兴趣的读者自行研究，笔者相信有了对启动、Task 状态更新、注册 Executor 的了解，其他功能的分析都将水到渠成。

#### 1. DriverEndpoint 的启动

有了对 DriverEndpoint 中属性的了解，现在来看看 DriverEndpoint 的启动过程。由于 DriverEndpoint 继承自 ThreadSafeRpcEndpoint，所以将 DriverEndpoint 注册到 RpcEnv 的 Dispatcher 时，会触发对 DriverEndpoint 的 onStart 方法的调用。DriverEndpoint 的 onStart 方法的实现如代码清单 9-81 所示。

代码清单9-81 DriverEndpoint的onStart方法

---

```
override def onStart() {
```

```

val reviveIntervalMs = conf.getTimeAsMs("spark.scheduler.revive.interval", "1s")
reviveThread.scheduleAtFixedRate(new Runnable {
    override def run(): Unit = Utils.tryLogNonFatalError {
        Option(self).foreach(_.send(ReviveOffers))
    }
}, 0, reviveIntervalMs, TimeUnit.MILLISECONDS)
}

```

---

根据代码清单 9-81，DriverEndpoint 的 onStart 方法中主要向 reviveThread 提交了一个向 DriverEndpoint 自己发送 ReviveOffers 消息的定时任务。此定时任务的执行间隔可通过 spark.scheduler.revive.interval 属性配置，默认为 1s。DriverEndpoint 接收到 ReviveOffers 消息后的处理如下。

```

case ReviveOffers =>
    makeOffers()

```

DriverEndpoint 的 makeOffers 方法的实现如代码清单 9-82 所示。

**代码清单9-82 DriverEndpoint的makeOffers方法**

```

private def makeOffers() {
    val activeExecutors = executorDataMap.filterKeys(executorIsAlive)
    val workOffers = activeExecutors.map { case (id, executorData) =>
        new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
    }.toIndexedSeq
    launchTasks(scheduler.resourceOffers(workOffers))
}

```

---

根据代码清单 9-82，DriverEndpoint 的 makeOffers 方法的执行步骤如下。

- 1) 过滤出激活的 Executor。
- 2) 根据每个激活的 Executor 的配置，创建 WorkerOffer。
- 3) 调用 TaskSchedulerImpl 的 resourceOffers 方法（见代码清单 7-101）给 Task 分配资源。
- 4) 调用 launchTasks 方法（见代码清单 9-83）运行 Task。

**代码清单9-83 DriverEndpoint的launchTasks方法**

```

private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
    for (task <- tasks.flatten) {
        val serializedTask = ser.serialize(task) // 对TaskDescription进行序列化
        if (serializedTask.limit >= maxRpcMessageSize) { // 序列化后的大小超出了Rpc消息的限制
            scheduler.taskIdToTaskSetManager.get(task.taskId).foreach { taskSetMgr =>
                try {
                    var msg = "Serialized task %s:%d was %d bytes, which exceeds max allowed: " +
                        "spark.rpc.message.maxSize (%d bytes). Consider increasing " +
                        "spark.rpc.message.maxSize or using broadcast variables for large values."
                    msg = msg.format(task.taskId, task.index, serializedTask.limit, maxRpc-
                        MessageSize)
                    taskSetMgr.abort(msg) // 放弃对TaskSetManager的调度
                } catch {
                    case e: Exception => logError("Exception in error callback", e)
                }
            }
        }
    }
}

```

```

        }
    }
    else {
        val executorData = executorDataMap(task.executorId)
        executorData.freeCores -= scheduler.CPUS_PER_TASK
        executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask)))
    }
}
}

```

根据代码清单 9-83，launchTasks 方法遍历每个 TaskDescription，并对每个 TaskDescription 执行以下操作。

- 1) 对 TaskDescription 进行序列化。
- 2) 如果序列化后的 TaskDescription 的大小大于等于 RPC 消息大小的最大值 maxRpcMessageSize，则执行如下操作。

- ① 从 TaskSchedulerImpl 的 taskIdToTaskSetManager 中找出 Task 对应的 TaskSetManager。
- ② 调用 TaskSetManager 的 abort 方法放弃对 TaskSetManager 的调度。
- 3) 如果序列化后的 TaskDescription 的大小小于 RPC 消息大小的最大值 maxRpcMessageSize，则执行如下操作。
  - ① 减少 Executor 的空闲内核数 freeCores。
  - ② 向 CoarseGrainedExecutorBackend 发送 LaunchTask 消息。CoarseGrainedExecutorBackend 将在收到 LaunchTask 消息后运行 Task。

## 2. DriverEndpoint 与 Task 状态更新

Task 在运行的过程中，会向 DriverEndpoint 发送 StatusUpdate 消息，这可以让 Driver 知道 Task 的当前状态，从而执行更新度量、将 Task 释放的资源分配给其他 Task 等操作。DriverEndpoint 的 receive 方法中实现了对 StatusUpdate 消息的接收处理，如代码清单 9-84 所示。

代码清单 9-84 DriverEndpoint 对 StatusUpdate 消息的处理

```

case StatusUpdate(executorId, taskId, state, data) =>
    scheduler.statusUpdate(taskId, state, data.value)
    if (TaskState.isFinished(state)) {
        executorDataMap.get(executorId) match {
            case Some(executorInfo) =>
                executorInfo.freeCores += scheduler.CPUS_PER_TASK
                makeOffers(executorId) // 给下一个要调度的Task分配资源并运行Task
            case None => // 对于未知的Executor, DriverEndpoint选择忽略
        }
    }
}

```

根据代码清单 9-84，DriverEndpoint 对 StatusUpdate 消息的处理步骤如下。

- 1) 调用 TaskSchedulerImpl 的 statusUpdate 方法（见代码清单 7-103）更新 Task 的状态。

2) 如果 Task 的状态是已经完成，则将 Task 释放的内核数增加到对应 Executor 的空闲内核数 (freeCores)，然后调用 makeOffers 方法给下一个要调度的 Task 分配资源并运行 Task。

### 3. 注册 Executor

给 Driver 分配了 Executor 之后，需要向 Driver 注册 Executor，以便于 Driver 与 Executor 之间直接通信，而不用通过 Worker 作为媒介，进而提升执行效率。DriverEndpoint 的 receiveAndReply 方法中实现了对 RegisterExecutor 消息的处理，代码如下。

```

case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>
  if (executorDataMap.contains(executorId)) { // 重复注册Executor
    executorRef.send(RegisterExecutorFailed("Duplicate executor ID: " + executorId))
    context.reply(true)
  } else {
    val executorAddress = if (executorRef.address != null) {
      executorRef.address
    } else {
      context.senderAddress
    }
    logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executor-
    Id")
    addressToExecutorId(executorAddress) = executorId
    totalCoreCount.addAndGet(cores) // 增加Driver已经获得的内核总数
    totalRegisteredExecutors.addAndGet(1) // 向Driver注册的所有Executor的总数
    val data = new ExecutorData(executorRef, executorRef.address, hostname,
      cores, cores, logUrls)
    CoarseGrainedSchedulerBackend.this.synchronized {
      executorDataMap.put(executorId, data)
      if (currentExecutorIdCounter < executorId.toInt) {
        currentExecutorIdCounter = executorId.toInt
      }
      if (numPendingExecutors > 0) {
        numPendingExecutors -= 1
        logDebug(s"Decrement number of pending executors ($numPendingExecutors
          left)")
      }
    }
    executorRef.send(RegisteredExecutor)
    context.reply(true)
    listenerBus.post(
      SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))
    makeOffers() // 给Task分配资源并运行Task
  }
}

```

根据上述代码，DriverEndpoint 处理 RegisterExecutor 消息的步骤如下。

1) 如果外部类 CoarseGrainedSchedulerBackend 的 executorDataMap 中已经注册了 RegisterExecutor 消息携带的 executorId，那么通过向 CoarseGrainedExecutorBackend 回复 RegisterExecutorFailed 消息，告诉后者重复注册了。

2) 如果外部类 CoarseGrainedSchedulerBackend 的 executorDataMap 中还未注册 RegisterExecutor 消息携带的 executorId，则执行如下操作。

- ① 获取 CoarseGrainedExecutorBackend 的 RpcAddress。

- ② 将 CoarseGrainedExecutorBackend 的 RpcAddress 和 executorId 的对应关系放入外部类 CoarseGrainedSchedulerBackend 的 addressToExecutorId 缓存中。
- ③ 增加外部类 CoarseGrainedSchedulerBackend 的 totalCoreCount，以表示 Driver 已经获得的内核总数。增加外部类 CoarseGrainedSchedulerBackend 的 totalRegisteredExecutors，以表示向 Driver 注册的所有 Executor 的总数。
- ④ 创建 ExecutorData 对象，并将 executorId 与 ExecutorData 的对应关系放入外部类 CoarseGrainedSchedulerBackend 的 executorDataMap 缓存中。此外，还将更新外部类 CoarseGrainedSchedulerBackend 的 currentExecutorIdCounter 和 numPendingExecutors 等属性。
- ⑤ 向 CoarseGrainedExecutorBackend 发送 RegisteredExecutor 消息。CoarseGrainedExecutorBackend 接收到 RegisteredExecutor 消息后将创建 Executor 实例（见代码清单 9-91）。
- ⑥ 向 LiveListenerBus 投递 SparkListenerExecutorAdded 事件。
- ⑦ 最后调用 makeOffers 方法（见代码清单 9-82）给 Task 分配资源并运行 Task。

#### 4. 对 RetrieveSparkAppConfig 消息的处理

CoarseGrainedExecutorBackend 进程在启动过程中会向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息，从 Driver 获取 Executor 所需的 Spark 属性信息和密钥信息。DriverEndpoint 的 receiveAndReply 方法中实现了对 RetrieveSparkAppConfig 消息的处理，代码如下。

```
case RetrieveSparkAppConfig =>
  val reply = SparkAppConfig(sparkProperties,
    SparkEnv.get.securityManager.getIOEncryptionKey())
  context.reply(reply)
```

#### 5. 停止 Driver

DriverEndpoint 的 receiveAndReply 方法中实现了对 StopDriver 消息的处理，代码如下。

```
case StopDriver =>
  context.reply(true)
  stop()
```

可以看到，DriverEndpoint 向发送端回复 true 之后调用父类 RpcEndpoint 的 stop 方法（见代码清单 5-2）。

#### 6. 停止 Executor

DriverEndpoint 的 receiveAndReply 方法中实现了对 StopExecutors 消息的处理，代码如下。

```
case StopExecutors =>
  logInfo("Asking each executor to shut down")
  for ((_, executorData) <- executorDataMap) {
    executorData.executorEndpoint.send(StopExecutor)
  }
  context.reply(true)
```

上面的代码遍历 executorDataMap 中的 ExecutorData，向每个 CoarseGrained-Executor-Backend 进程发送 StopExecutor 消息。CoarseGrainedExecutorBackend 进程收到 StopExecutor 消息后的资源回收处理如代码清单 9-91 所示。

### 9.9.3 StandaloneSchedulerBackend 的启动

由于 StandaloneSchedulerBackend 的 start 方法会率先调用父类 CoarseGrainedSchedulerBackend 的 start 方法，所以我们先来看看 CoarseGrainedSchedulerBackend 的 start 方法的实现，如代码清单 9-85 所示。

代码清单9-85 CoarseGrainedSchedulerBackend的start方法

---

```

override def start() {
    val properties = new ArrayBuffer[(String, String)]
    for ((key, value) <- scheduler.sc.conf.getAll) {
        if (key.startsWith("spark."))
            properties += ((key, value)) // 将以spark.开头的属性添加到数组缓冲properties中
    }
}

driverEndpoint = createDriverEndpointRef(properties) // 创建并注册DriverEndpoint
}

protected def createDriverEndpointRef(
    properties: ArrayBuffer[(String, String)]): RpcEndpointRef = {
    rpcEnv.setupEndpoint(ENDPOINT_NAME, createDriverEndpoint(properties))
}

protected def createDriverEndpoint(properties: Seq[(String, String)]): DriverEndpoint
    = {
    new DriverEndpoint(rpcEnv, properties)
}

```

---

根据代码清单 9-85，CoarseGrainedSchedulerBackend 的 start 方法的执行步骤如下。

- 1) 从 SparkContext 的 SparkConf 中将以 spark. 开头的属性添加到数组缓冲 properties 中。
- 2) 调用 createDriverEndpointRef 方法创建 DriverEndpoint，并将 DriverEndpoint 注册到 SparkContext 的 SparkEnv 的 RpcEnv 中，注册时以常量 ENDPOINT\_NAME（值为 Coarse-GrainedScheduler）作为注册名。

了解了父类的 start 方法，现在来看看 StandaloneSchedulerBackend 的 start 方法的实现，如代码清单 9-86 所示。

代码清单9-86 StandaloneSchedulerBackend的start方法

---

```

override def start() {
    super.start() // 创建并注册DriverEndpoint
    launcherBackend.connect() // 与LauncherServer建立连接

    val driverUrl = RpcEndpointAddress( // 生成Driver URL，Executor将通过此URL与Driver
       通信

```

---

```

sc.conf.get("spark.driver.host"),
sc.conf.get("spark.driver.port").toInt,
CoarseGrainedSchedulerBackend.ENDPOINT_NAME).toString
val args = Seq(
    "--driver-url", driverUrl,
    "--executor-id", "{{EXECUTOR_ID}}",
    "--hostname", "{{HOSTNAME}}",
    "--cores", "{{CORES}}",
    "--app-id", "{{APP_ID}}",
    "--worker-url", "{{WORKER_URL}}")
val extraJavaOpts = sc.conf.getOption("spark.executor.extraJavaOptions")
    .map(Utils.splitCommandString).getOrElse(Seq.empty)
val classPathEntries = sc.conf.getOption("spark.executor.extraClassPath")
    .map(_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)
val libraryPathEntries = sc.conf.getOption("spark.executor.extraLibraryPath")
    .map(_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)

val testingClassPath =
    if (sys.props.contains("spark.testing")) {
        sys.props("java.class.path").split(java.io.File.pathSeparator).toSeq
    } else {
        Nil
    }

val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)
val javaOpts = sparkJavaOpts ++ extraJavaOpts
val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
//创建Command
    args, sc.executorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries,
    javaOpts)
val appUIAddress = sc.ui.map(_.appUIAddress).getOrElse("")
val coresPerExecutor = conf.getOption("spark.executor.cores").map(_.toInt)
val initialExecutorLimit =
    if (Utils.isDynamicAllocationEnabled(conf)) {
        Some(0)
    } else {
        None
    }
val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory,
    command,
    appUIAddress, sc.eventLogDir, sc.eventLogCodec, coresPerExecutor, initial-
    ExecutorLimit)
client = new StandaloneAppClient(sc.env.rpcEnv, masters, appDesc, this, conf)
client.start() // 启动StandaloneAppClient
launcherBackend.setState(SparkAppHandle.State.SUBMITTED)
waitForRegistration() // 等待注册Application完成
launcherBackend.setState(SparkAppHandle.State.RUNNING)
}

```

根据代码清单 9-86，StandaloneSchedulerBackend 的 start 方法的执行步骤如下。

- 1) 调用父类 CoarseGrainedSchedulerBackend 的 start 方法（见代码清单 9-85）创建并注册 DriverEndpoint。
- 2) 调用 LauncherBackend 的 connect 方法（见代码清单 7-83）与 LauncherServer 建立

连接。

3 ) 生成 Driver URL，格式为 spark://CoarseGrainedScheduler@ \${driverHost}:\${driverPort}。Executor 将通过此 URL 与 Driver 通信。

4 ) 拼接参数列表 args、获取额外的 Java 参数 extraJavaOpts（通过 spark.executor.extraJavaOptions 属性配置）、额外的类路径 classPathEntries（通过 spark.executor.extraClassPath 属性配置）、额外的库路径 libraryPathEntries（通过 spark.executor.extraLibraryPath 属性配置）等。

5 ) 从 SparkConf 中获取需要传递给 Executor 用于启动的配置 sparkJavaOpts。这些配置包括：以 spark.auth 开头的配置（但不包括 spark.authenticate.secret）、以 spark.ssl 开头的配置、以 spark.rpc 开头的配置、以 spark. 开头且以 .port 结尾的配置及以 spark.port. 开头的配置。

6 ) 将 extraJavaOpts 与 sparkJavaOpts 合并到 javaOpts。

7 ) 创建 Command 对象。样例类 Command 定义了执行 Executor 的命令。这里以 org.apache.spark.executor.CoarseGrainedExecutorBackend 作为 Command 的 mainClass 属性、以 args 作为 Command 的 arguments 属性、以 SparkContext 的 executorEnvs 属性作为 Command 的 environment 属性、以 classPathEntries 作为 Command 的 classPathEntries 属性、以 libraryPathEntries 作为 Command 的 libraryPathEntries 属性、以 javaOpts 作为 Command 的 javaOpts 属性。

8 ) 获取 Spark UI 的 http 地址 appUIAddress、获取每个 Executor 分配的内核数 coresPerExecutor（可通过 spark.executor.cores 属性配置）、Executor 的初始限制 initialExecutorLimit（如果启用了动态分配 Executor，那么 initialExecutorLimit 被设置为 0，ExecutorAllocationManager 之后会将真实的初始限制值传递给 Master）。

9 ) 创建 Application 描述信息 (ApplicationDescription)。

10 ) 创建并启动 StandaloneAppClient。根据 9.8 节的内容我们知道，StandaloneAppClient 的 start 方法将创建 ClientEndpoint，并向 SparkContext 的 SparkEnv 的 RpcEnv 注册 ClientEndpoint，进而引起对 ClientEndpoint 的启动和向 Master 注册 Application。

11 ) 调用 LauncherBackend 的 setState 方法（见代码清单 7-85）向 LauncherServer 传递应用已经提交 (SUBMITTED) 的状态。

12 ) 调用 waitForRegistration 方法（见代码清单 9-87）等待注册 Application 完成。根据 9.6.8 节的内容，我们知道当注册 Application 成功后，Master 将向 ClientEndpoint 发送 RegisteredApplication 消息，进而调用 StandaloneSchedulerBackend 的 connected 方法释放信号量，这样 waitForRegistration 方法将可以获得信号量。

13 ) 调用 LauncherBackend 的 setState 方法（见代码清单 7-85）向 LauncherServer 传递应用正在运行 (RUNNING) 的状态。

代码清单9-87 waitForRegistration方法的实现

---

```
private def waitForRegistration() = {
    registrationBarrier.acquire()
}
```

---

### 9.9.4 StandaloneSchedulerBackend 的停止

StandaloneSchedulerBackend 的 stop 方法用于停止 StandaloneSchedulerBackend，其实现如下。

```
override def stop(): Unit = synchronized {
    stop(SparkAppHandle.State.FINISHED)
}
```

上面的 stop 方法调用了重载的 stop 方法，重载的 stop 方法如代码清单 9-88 所示。

代码清单9-88 StandaloneSchedulerBackend中重载的stop方法

---

```
private def stop(finalState: SparkAppHandle.State): Unit = synchronized {
    try {
        stopping = true

        super.stop()
        client.stop()

        val callback = shutdownCallback
        if (callback != null) {
            callback(this)
        }
    } finally {
        launcherBackend.setState(finalState)
        launcherBackend.close()
    }
}
```

---

根据代码清单 9-88，stop 方法首先调用 StandaloneAppClient 的 stop 方法（见代码清单 9-78）停止 StandaloneAppClient，然后调用父类 CoarseGrainedSchedulerBackend 的 stop 方法，最后回调关闭函数。CoarseGrainedSchedulerBackend 的 stop 方法的实现如代码清单 9-89 所示。

代码清单9-89 CoarseGrainedSchedulerBackend的stop方法

---

```
def stopExecutors() {
    try {
        if (driverEndpoint != null) {
            logInfo("Shutting down all executors")
            driverEndpoint.askWithRetry[Boolean](StopExecutors)
        }
    } catch {
        // 忽略对异常的处理
    }
}
```

---

```

override def stop() {
    stopExecutors()
    try {
        if (driverEndpoint != null) {
            driverEndpoint.askWithRetry[Boolean](StopDriver)
        }
    } catch {
        // 忽略对异常的处理
    }
}

```

---

根据代码清单 9-89，CoarseGrainedSchedulerBackend 的 stop 方法首先调用 stopExecutors 方法（实际向 DriverEndpoint 发送 StopExecutors 消息）停止 Executor，然后向 DriverEndpoint 发送 StopDriver 消息停止 Driver。

### 9.9.5 StandaloneSchedulerBackend 与资源分配

SchedulerBackend 存在的最大价值是代理 TaskSchedulerImpl，将集群分配给 Application 的资源进一步分配给 Task。本节将介绍在 Standalone 模式下进行资源的二级分配的组件，即 SchedulerBackend 的实现类 StandaloneSchedulerBackend。

#### 1. StandaloneSchedulerBackend 的二级资源分配

根据 7.10.4 节对 TaskSchedulerImpl 的提交 Task 的实现分析，我们知道 TaskSchedulerImpl 的 submitTasks 方法最后会调用 SchedulerBackend 的 reviveOffers 方法给 Task 分配资源并运行 Task。StandaloneSchedulerBackend 的父类 CoarseGrainedSchedulerBackend 实现了这一方法，代码如下。

```

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

```

根据上述代码，CoarseGrainedSchedulerBackend 的实现非常简单——只是向 DriverEndpoint 发送了 ReviveOffers 消息。根据之前对 DriverEndpoint 的启动的分析，我们知道 DriverEndpoint 收到 ReviveOffers 消息后，将调用 makeOffers 方法给 Task 分配资源并运行 Task。

#### 2. StandaloneSchedulerBackend 与 ExecutorAllocationManager

根据 4.9 节的内容，只有在 SchedulerBackend 的实现类同时实现了特质 ExecutorAllocationClient 的情况下，才会创建 ExecutorAllocationManager。SchedulerBackend 的实现类中只有 CoarseGrainedSchedulerBackend 同时实现了特质 ExecutorAllocationClient。在启动 ExecutorAllocationManager 的最后，会调用 CoarseGrainedSchedulerBackend 的 requestTotalExecutors 方法，其实现如下。

```

final override def requestTotalExecutors(
    numExecutors: Int, localityAwareTasks: Int, hostToLocalTaskCount: Map[String,

```

```

    Int]
): Boolean = {
if (numExecutors < 0) { // 忽略抛出IllegalArgumentException的代码}

val response = synchronized {
  this.localityAwareTasks = localityAwareTasks
  this.hostToLocalTaskCount = hostToLocalTaskCount

  numPendingExecutors =
    math.max(numExecutors - numExistingExecutors + executorsPendingToRemove.size, 0)

  doRequestTotalExecutors(numExecutors)
}

defaultAskTimeout.awaitResult(response)
}

```

`requestTotalExecutors`方法一共有三个参数，分别是请求的 Executor 数量 (`numExecutors`)、有本地性偏好的 Task 总数 (`localityAwareTasks`)、Host 与想要在 Host 本地运行的 Task 数量之间的映射关系 (`hostToLocalTaskCount`)。`requestTotalExecutors`方法计算了待申请的 Executor 数量 (`numPendingExecutors`) 后调用 `doRequestTotalExecutors` 方法。`StandaloneSchedulerBackend` 实现了父类 `CoarseGrainedSchedulerBackend` 的 `doRequestTotalExecutors` 方法，代码如下。

```

protected override def doRequestTotalExecutors(requestedTotal: Int): Future[Boolean] =
{
  Option(client) match {
    case Some(c) => c.requestTotalExecutors(requestedTotal)
    case None =>
      logWarning("Attempted to request executors before driver fully initialized.")
      Future.successful(false)
  }
}

```

上面的 `doRequestTotalExecutors` 方法主要调用了 `StandaloneAppClient` 的 `requestTotalExecutors` 方法（见代码清单 9-79）向 Master 请求所需的所有 Executor 资源。

## 9.10 CoarseGrainedExecutorBackend 详解

在 9.7.6 节曾经提到过 `CoarseGrainedExecutorBackend`，而且还给出了 `CoarseGrainedExecutorBackend` 进程的截图，本节将对 `CoarseGrainedExecutorBackend` 的实现进行详细分析。`CoarseGrainedExecutorBackend` 是 Driver 与 Executor 通信的后端接口，但只在 local-cluster 和 Standalone 模式下才会使用。由于 `CoarseGrainedExecutorBackend` 作为单独的进程存在，所以 `CoarseGrainedExecutorBackend` 的伴生对象中实现了 `main` 方法。在 `main` 方法中将创建 `CoarseGrainedExecutorBackend` 实例，因此本节还需要介绍 `CoarseGrained-`

ExecutorBackend。对于 CoarseGrainedExecutorBackend，我们首先介绍它的属性，然后挑选其最核心的功能进行分析。

### 9.10.1 CoarseGrainedExecutorBackend 进程

CoarseGrainedExecutorBackend 的伴生对象中实现了 main 方法，所以可以作为单独的 Java 进程启动。main 方法的实现如下。

```
def main(args: Array[String]) {
    var driverUrl: String = null
    var executorId: String = null
    var hostname: String = null
    var cores: Int = 0
    var appId: String = null
    var workerUrl: Option[String] = None
    val userClassPath = new mutable.ListBuffer[URL]()
    var argv = args.toList
    while (!argv.isEmpty) {
        argv match {
            case ("--driver-url") :: value :: tail =>
                driverUrl = value
                argv = tail
            case ("--executor-id") :: value :: tail =>
                executorId = value
                argv = tail
            case ("--hostname") :: value :: tail =>
                hostname = value
                argv = tail
            case ("--cores") :: value :: tail =>
                cores = value.toInt
                argv = tail
            case ("--app-id") :: value :: tail =>
                appId = value
                argv = tail
            case ("--worker-url") :: value :: tail =>
                workerUrl = Some(value)
                argv = tail
            case ("--user-class-path") :: value :: tail =>
                userClassPath += new URL(value)
                argv = tail
            case Nil =>
            case tail =>
                System.err.println(s"Unrecognized options: ${tail.mkString(" ")}")
                printUsageAndExit()
        }
    }
    if (driverUrl == null || executorId == null || hostname == null || cores <= 0 ||
        appId == null) {
        printUsageAndExit()
    }
    run(driverUrl, executorId, hostname, cores, appId, workerUrl, userClassPath)
}
```

```

        System.exit(0)
    }
}

```

根据 main 方法的代码实现，其执行步骤如下。

- 1) 对执行参数进行解析，如将参数 --driver-url 的值解析后保存到变量 driverUrl 中。
- 2) 将解析得到的变量作为参数，调用 run 方法。run 方法的实现如代码清单 9-90 所示。

**代码清单9-90 CoarseGrainedExecutorBackend伴生对象的run方法**

---

```

private def run(
    driverUrl: String,
    executorId: String,
    hostname: String,
    cores: Int,
    appId: String,
    workerUrl: Option[String],
    userClassPath: Seq[URL]) {

    Utils.initDaemon(log)

    SparkHadoopUtil.get.runAsSparkUser { () =>
        Utils.checkHost(hostname)
        // 从CoarseGrainedSchedulerBackend获取所需的Spark属性信息和密钥
        val executorConf = new SparkConf
        val port = executorConf.getInt("spark.executor.port", 0)
        val fetcher = RpcEnv.create("driverPropsFetcher", hostname, port, executor-
            Conf,
            new SecurityManager(executorConf), clientMode = true)
        val driver = fetcher.setupEndpointRefByURI(driverUrl)
        val cfg = driver.askWithRetry[SparkAppConfig](RetrieveSparkAppConfig)
        val props = cfg.sparkProperties ++ Seq[(String, String)](("spark.app.id",
            appId))
        fetcher.shutdown()

        val driverConf = new SparkConf() // 创建Executor自己的SparkConf
        for ((key, value) <- props) {
            if (SparkConf.isExecutorStartupConf(key)) {
                driverConf.setIfMissing(key, value)
            } else {
                driverConf.set(key, value)
            }
        }
        if (driverConf.contains("spark.yarn.credentials.file")) {
            logInfo("Will periodically update credentials from: " +
                driverConf.get("spark.yarn.credentials.file"))
            SparkHadoopUtil.get.startCredentialUpdater(driverConf)
        }
    }

    val env = SparkEnv.createExecutorEnv( // 创建Executor自身的SparkEnv
        driverConf, executorId, hostname, port, cores, cfg.ioEncryptionKey, isLocal
        = false)
    // 创建并注册CoarseGrainedExecutorBackend实例
    env.rpcEnv.setupEndpoint("Executor", new CoarseGrainedExecutorBackend(
        env.rpcEnv, driverUrl, executorId, hostname, cores, userClassPath, env))
    workerUrl.foreach { url => // 创建并注册WorkerWatcher实例
}
}

```

```

        env.rpcEnv.setupEndpoint("WorkerWatcher", new WorkerWatcher(env.rpcEnv, url))
    }
    env.rpcEnv.awaitTermination()
    SparkHadoopUtil.get.stopCredentialUpdater()
}
}

```

---

根据代码清单 9-90，run 方法的执行步骤如下。

1) 创建名为 driverPropsFetcher 的 RpcEnv。此 RpcEnv 主要用于从 Driver 拉取属性信息，并非 Executor 的 SparkEnv 中的 RpcEnv。

2) 向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息，从 CoarseGrainedSchedulerBackend 获取所需的 Spark 属性信息和密钥。DriverEndpoint 对 RetrieveSparkAppConfig 消息的处理已在 9.9.2 节介绍。

3) 将 CoarseGrainedSchedulerBackend 回复的 SparkAppConfig 消息中携带的 Spark 属性和 Application 的 ID 属性保存到 props 中。

4) 由于名为 driverPropsFetcher 的 RpcEnv 已经完成了它的使命，因此关闭它。

5) 创建 Executor 自己的 SparkConf。变量名 driverConf 说明 SparkConf 都是从 Driver 获取的。

6) 将 props 中的属性全部保存到 SparkConf 中。

7) 调用 SparkEnv 伴生对象的 createExecutorEnv 方法创建 Executor 自身的 SparkEnv。createExecutorEnv 方法与 4.2 节介绍的 SparkEnv 伴生对象的 createDriverEnv 方法一样，实际也是调用了 SparkEnv 伴生对象的 create 方法。

8) 创建 CoarseGrainedExecutorBackend 实例，并注册到 Executor 自身 SparkEnv 的子组件 RpcEnv 中。

9) 创建 WorkerWatcher 实例，并注册到 Executor 自身 SparkEnv 的子组件 RpcEnv 中。顾名思义，WorkerWatcher 是 Worker 的守望者，当发生 Worker 进程退出、连接断开、网络出错等情况时，终止 CoarseGrainedExecutorBackend 进程。Worker 与 CoarseGrainedExecutorBackend 进程相辅相成。由于 WorkerWatcher 的实现非常简单，因此感兴趣的读者可自行阅读。

### 9.10.2 CoarseGrainedExecutorBackend 的功能分析

按照本书一往的安排，笔者依然先来介绍 CoarseGrainedExecutorBackend 的属性，这些属性如下。

- ❑ rpcEnv: Executor 所需的 RpcEnv。
- ❑ driverUrl: Driver 的 Spark 格式的 URL。例如，spark://CoarseGrainedScheduler@192.168.0.111:53777。
- ❑ executorId: Master 分配给 Executor 的身份标识。

- hostname：主机名。
- cores：分配给 Executor 的内核数。
- userClassPath：用户指定的类路径。
- env：Executor 所需的 SparkEnv。
- stopping：标记 CoarseGrainedExecutorBackend 是否正在停止。
- executor：Executor 实例。
- driver：DriverEndpoint 的 RpcEndpointRef。
- ser：Executor 所需的 SparkEnv 中的 closureSerializer 的实例。

有了对以上属性的了解，现在可以展开对 CoarseGrainedExecutorBackend 的核心功能的分析了。

### 1. CoarseGrainedExecutorBackend 的 onStart 方法

根据之前对 CoarseGrainedExecutorBackend 的伴生对象的 run 方法的分析，我们知道，run 方法中创建了 CoarseGrainedExecutorBackend，并将它注册到 Executor 自身的 RpcEnv 中。在注册到 RpcEnv 的过程中会触发对 CoarseGrainedExecutorBackend 的 onStart 方法的调用，其实现如下。

```
override def onStart() {
    logInfo("Connecting to driver: " + driverUrl)
    rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
        driver = Some(ref)
        ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))
    } (ThreadUtils.sameThread).onComplete {
        case Success(msg) =>
        case Failure(e) =>
            exitExecutor(1, s"Cannot register with driver: $driverUrl", e, notifyDriver = false)
    } (ThreadUtils.sameThread)
}
```

根据上述代码，onStart 方法中向 DriverEndpoint 发送 RegisterExecutor 消息注册 Executor。DriverEndpoint 对 RegisterExecutor 消息的处理已经在 9.9.2 节介绍过。

### 2. CoarseGrainedExecutorBackend 的 receive 方法

CoarseGrainedExecutorBackend 的 receive 方法中实现了对 RegisteredExecutor、LaunchTask、KillTask 等消息的处理，如代码清单 9-91 所示。

代码清单9-91 CoarseGrainedExecutorBackend的receive方法

---

```
override def receive: PartialFunction[Any, Unit] = {
    case RegisteredExecutor =>
        logInfo("Successfully registered with driver")
        try {
            executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)
```

```

    } catch {
      case NonFatal(e) =>
        exitExecutor(1, "Unable to create executor due to " + e.getMessage, e)
    }

    case RegisterExecutorFailed(message) =>
      exitExecutor(1, "Slave registration failed: " + message)

    case LaunchTask(data) =>
      if (executor == null) {
        exitExecutor(1, "Received LaunchTask command but executor was null")
      } else {
        val taskDesc = ser.deserialize[TaskDescription](data.value)
        logInfo("Got assigned task " + taskDesc.taskId)
        executor.launchTask(this, taskId = taskDesc.taskId, attemptNumber = task-
          Desc.attemptNumber,
          taskDesc.name, taskDesc.serializedTask)
      }

    case KillTask(taskId, _, interruptThread) =>
      if (executor == null) {
        exitExecutor(1, "Received KillTask command but executor was null")
      } else {
        executor.killTask(taskId, interruptThread)
      }

    case StopExecutor =>
      stopping.set(true)
      logInfo("Driver commanded a shutdown")
      self.send(Shutdown)

    case Shutdown =>
      stopping.set(true)
      new Thread("CoarseGrainedExecutorBackend-stop-executor") {
        override def run(): Unit = {
          executor.stop()
        }
      }.start()
  }
}

```

### 3. CoarseGrainedExecutorBackend 的 statusUpdate 方法

在 Standalone 模式下，由于 CoarseGrainedExecutorBackend 与 Executor 在同一个进程中，所以它对 Task 的状态能够准确实时地捕获。利用这一天生优势，CoarseGrainedExecutorBackend 可以将 Task 的状态发送给 DriverEndpoint，以便对 Task 的状态进行更新。CoarseGrainedExecutorBackend 的 statusUpdate 方法实现了这一功能，如代码清单 9-92 所示。

代码清单9-92 CoarseGrainedExecutorBackend的statusUpdate方法

---

```

override def statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer) {
  val msg = StatusUpdate(executorId, taskId, state, data)
  driver match {
    case Some(driverRef) => driverRef.send(msg)
    case None => logWarning(s"Drop $msg because has not yet connected to driver")
  }
}

```

```

    }
}

```

## 9.11 local-cluster 部署模式

local-cluster 是一种伪集群部署模式，Driver、Master 和 Worker 在同一个 JVM 进程内，可以存在多个 Worker，每个 Worker 会有多个 Executor，但这些 Executor 都独自存在于一个 JVM 进程内。除了这些，它与 local 部署模式的区别如下。

- 使用 LocalSparkCluster 启动集群。
- StandaloneSchedulerBackend 的启动过程不同。
- AppClient 的启动与调度。
- local-cluster 模式的任务执行。

在第 7 章，笔者曾讲解了 TaskSchedulerImpl 和 local 模式下 SchedulerBackend 的实现 LocalSchedulerBackend。本节我们设置 master 为 local-cluster[2,1,1024]，那么在创建 TaskSchedulerImpl 时（见代码清单 4-18）就会匹配 local-cluster 模式。local-cluster [2,1,1024] 中的 2 指定了 Worker 的数量（numSlaves），1 指定了每个 Worker 占用的 CPU 核数（coresPerSlave），1024 指定的是每个 Worker 占用的内存大小（memoryPerSlave）。memoryPerSlave 必须大于等于 executorMemory（即 SparkContext 的 \_executorMemory 属性），因为 Worker 的内存大小包括 Executor 占用的内存。

与 local 模式不同的是，local-cluster 除 TaskSchedulerImpl 外，还创建了 LocalSparkCluster，LocalSparkCluster 的 start 方法用于启动集群。local-cluster 模式中，ExecutorBackend 的实现类是 CoarseGrainedExecutorBackend，SchedulerBackend 的实现类是 StandaloneSchedulerBackend。

 **注意** 给 StandaloneSchedulerBackend 的 shutdownCallback 绑定 LocalSparkCluster 的 stop 方法，用于当 Driver 关闭时关闭集群。这仅限于 local-cluster 模式。

### 9.11.1 启动本地集群

LocalSparkCluster 实现了本地集群。LocalSparkCluster 包含以下属性。

- numWorkers：Worker 的数量。
- coresPerWorker：每个 Worker 的内核数。
- memoryPerWorker：每个 Worker 的内存大小。
- conf：即 SparkConf。
- localHostname：本地主机名。通过调用 Utils 工具类的 localHostName 方法获得。
- masterRpcEnvs：用于缓存所有 Master 的 RpcEnv。
- workerRpcEnvs：维护所有 Worker 的 RpcEnv。

- ❑ masterWebUIPort: Master 的 WebUI 的端口。

有了对 LocalSparkCluster 中定义的成员属性的了解，现在来看看如何启动本地集群。LocalSparkCluster 的 start 方法（见代码清单 9-93）用于创建、启动 Master 的 RpcEnv 与多个 Worker 的 RpcEnv。

代码清单9-93 LocalSparkCluster的start方法

---

```

def start(): Array[String] = {
    logInfo("Starting a local Spark cluster with " + numWorkers + " workers.")

    val _conf = conf.clone()
        .setIfMissing("spark.master.rest.enabled", "false")
        .set("spark.shuffle.service.enabled", "false")

    /* 启动Master */
    val (rpcEnv, webUiPort, _) = Master.startRpcEnvAndEndpoint(localHostname, 0, 0, _
        conf)
    masterWebUIPort = webUiPort
    masterRpcEnvs += rpcEnv
    val masterUrl = "spark://" + Utils.localHostNameForURI() + ":" + rpcEnv.-
        address.port
    val masters = Array(masterUrl)

    for (workerNum <- 1 to numWorkers) { /* 启动多个Worker */
        val workerEnv = Worker.startRpcEnvAndEndpoint(localHostname, 0, 0, cores-
            PerWorker,
            memoryPerWorker, masters, null, Some(workerNum), _conf)
        workerRpcEnvs += workerEnv
    }
}

masters
}

```

---

根据代码清单 9-93，启动本地 Spark 集群的步骤如下。

- 1) 在本地模式下通过将 spark.master.rest.enabled 和 spark.shuffle.service.enabled 设置为 false，使得 REST 服务不可用。
- 2) 调用 Master 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-24）启动 Master，然后将 Master 的 WebUI 的端口保存到 masterWebUIPort 属性，并将 Master 的 RpcEnv 放入 masterRpcEnvs 缓存，最后创建 Master 的 URL。
- 3) 按照 numWorkers 指定的 Worker 数量，调用 Worker 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-50）启动多个 Worker，并将每个 Worker 的 RpcEnv 放入 workerRpcEnvs 缓存。
- 4) 返回 Master 的 URL。

此外，LocalSparkCluster 还提供了 stop 方法用于关闭、清理 Master 的 RpcEnv 与多个 Worker 的 RpcEnv。由于 stop 方法的实现非常简单，因此留给感兴趣的读者自行阅读。

### 9.11.2 local-cluster 部署模式的启动过程

有了本章对 Master、Worker、StandaloneAppClient、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 local-cluster 部署模式的启动过程，如图 9-21 所示。

这里对图 9-21 中的各个序号进行说明。

序号①：根据 SparkContext 的 createTaskScheduler 方法（见代码清单 4-18）匹配 local-cluster 模式的代码，首先调用 LocalSparkCluster 的 start 方法启动本地集群。

序号②：根据 9.11.1 节的介绍，LocalSparkCluster 的 start 方法首先创建并启动 Master。

序号③：启动 Master 的过程（详见 9.6.1 节）中将创建领导选举代理（LeaderElectionAgent）。

序号④：由于 local-cluster 模式下只有一个 Master，所以无论采用哪种领导选举代理，都将选择 Master 作为激活的 Master。

序号⑤：LocalSparkCluster 的 start 方法之后会根据 local-cluster[numSlaves, coresPerSlave, memoryPerSlave] 指定的 numSlaves 的大小，创建并启动一到多个 Worker。

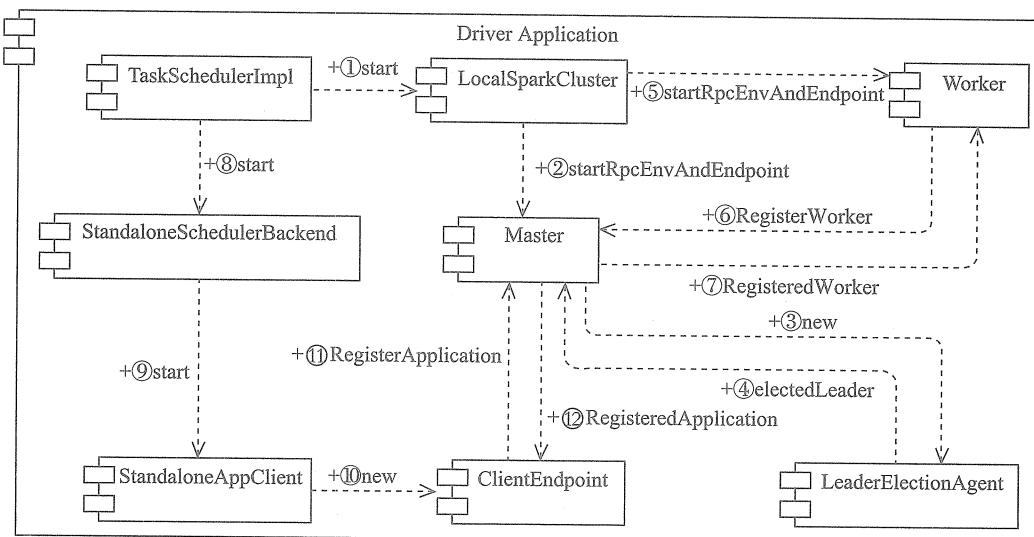


图 9-21 local-cluster 部署模式的启动过程

序号⑥：在启动 Worker 的过程中将向 Master 发送 RegisterWorker 消息（见 9.7.2 节）。

序号⑦：Master 在接收到 RegisterWorker 消息后对 WorkerInfo 进行注册，WorkerInfo 注册成功后，Master 向 Worker 回复 RegisteredWorker 消息（见 9.6.5 节）。Worker 收到 RegisteredWorker 消息后，将向 Master 发送 WorkerLatestState 消息，以便把 Worker 的最新状态汇报给 Master。根据 9.6.6 节的内容，我们知道 Master 接收到 WorkerLatestState 消息后，将对 Executor 和 Driver 进行匹配，对于不匹配的 Executor 和 Driver，将通知 Worker “杀死”。



**注意** 在通过 LocalSparkCluster 启动本地集群后，会对 TaskSchedulerImpl 进行初始化。根据 7.10.2 节的内容，我们知道 TaskSchedulerImpl 的 initialize 方法会将参数传递的 StandaloneSchedulerBackend 保存到 TaskSchedulerImpl 的 backend 属性。

**序号⑧：**SparkContext 创建完 TaskSchedulerImpl 后，将启动 TaskSchedulerImpl（见代码清单 4-19）。根据 7.10.3 节对启动 TaskSchedulerImpl 的分析，TaskSchedulerImpl 的 start 方法一开始就会调用 StandaloneSchedulerBackend 的 start 方法。

**序号⑨：**通过 9.9.3 节对 StandaloneSchedulerBackend 的启动分析，StandaloneSchedulerBackend 的 start 方法中会创建并启动 StandaloneAppClient。

**序号⑩：**StandaloneAppClient 的 start 方法会创建 ClientEndpoint 并将 ClientEndpoint 注册到 SparkContext 的 SparkEnv 的 RpcEnv 中，进而引起 ClientEndpoint 的启动。

**序号⑪：**9.8.1 节详细介绍了 ClientEndpoint 的启动，ClientEndpoint 在启动时会向每个 Master 发送 RegisterApplication 消息。

**序号⑫：**根据 9.6.8 节的介绍，Master 收到 RegisterApplication 消息并将 Application 注册成功后，会向 DriverEndpoint 发送 RegisteredApplication 消息。ClientEndpoint 的 receive 方法接收到 RegisteredApplication 消息后更新 ClientEndpoint 的 appId、registered、master 属性及 StandaloneSchedulerBackend 的 appId 属性。

### 9.11.3 local-cluster 部署模式下 Executor 的分配过程

有了本章对 Master、Worker、StandaloneAppClient、DriverEndpoint、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 local-cluster 部署模式下的 Executor 资源分配过程，如图 9-22 所示。

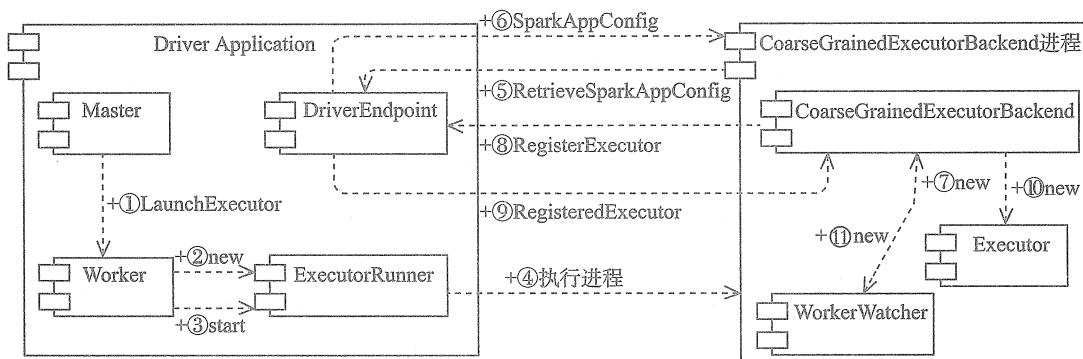


图 9-22 local-cluster 部署模式下的 Executor 资源分配过程

这里对图 9-22 中的各个序号进行说明。

**序号①：**Master 的 schedule 方法在对资源调度时将调用 startExecutorsOnWorkers 方

法对 Executor 的资源进行分配和调度，然后通过向 Worker 发送 LaunchExecutor 消息运行 Executor。

序号②：Worker 接收到 LaunchExecutor 消息后将创建 ExecutorRunner。

序号③：Worker 创建完 ExecutorRunner 后，将调用 ExecutorRunner 的 start 方法。

序号④：ExecutorRunner 的 start 方法主要创建了线程 workerThread，线程 workerThread 将构建并启动 ProcessBuilder。由于 ProcessBuilder 的命令中指定了 Java 进程的主方法是 CoarseGrainedExecutorBackend 的伴生对象的 main 函数，因此将启动执行 CoarseGrained-ExecutorBackend 的 main 函数的进程。

序号⑤：执行 CoarseGrainedExecutorBackend 的 main 函数的进程首先创建名为 driverPropsFetcher 的 RpcEnv，然后向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息。

序号⑥：DriverEndpoint 接收到 RetrieveSparkAppConfig 消息后，将向执行 CoarseGrained-ExecutorBackend 的 main 函数的进程回复 SparkAppConfig 消息。SparkAppConfig 消息携带着 Spark 属性信息和分配给 Executor 的密钥。

序号⑦：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，将首先创建 SparkConf 和 SparkEnv，然后创建 CoarseGrainedExecutorBackend 实例并注册到 Executor 自身的 SparkEnv 的 RpcEnv。

序号⑧：将 CoarseGrainedExecutorBackend 实例注册到 Executor 自身的 SparkEnv 的 RpcEnv 时，会触发对 CoarseGrainedExecutorBackend 的 onStart 方法的调用，进而向 DriverEndpoint 发送 RegisterExecutor 消息以注册 Executor。

序号⑨：DriverEndpoint 接收到 RegisterExecutor 消息对 Executor 注册成功后，将向 CoarseGrainedExecutorBackend 实例回复 RegisteredExecutor 消息。

序号⑩：CoarseGrainedExecutorBackend 实例接收到 RegisteredExecutor 消息后将创建 Executor。

序号⑪：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，除了创建 CoarseGrainedExecutorBackend 实例，还将创建 WorkerWatcher，以保证执行 CoarseGrainedExecutorBackend 的 main 函数的进程在与 Worker 断开连接或发生网络错误的情况下退出。

#### 9.11.4 local-cluster 部署模式下的任务提交执行过程

有了本章对 Master、Worker、StandaloneAppClient、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 local-cluster 部署模式下的任务提交执行过程，如图 9-23 所示。

这里对图 9-23 中的各个序号进行说明。

序号①：TaskSchedulerImpl 的 submitTasks 方法（见代码清单 7-99）提交 Task 的最后会调用 StandaloneSchedulerBackend 的父类 CoarseGrainedSchedulerBackend 的 reviveOffers

方法。

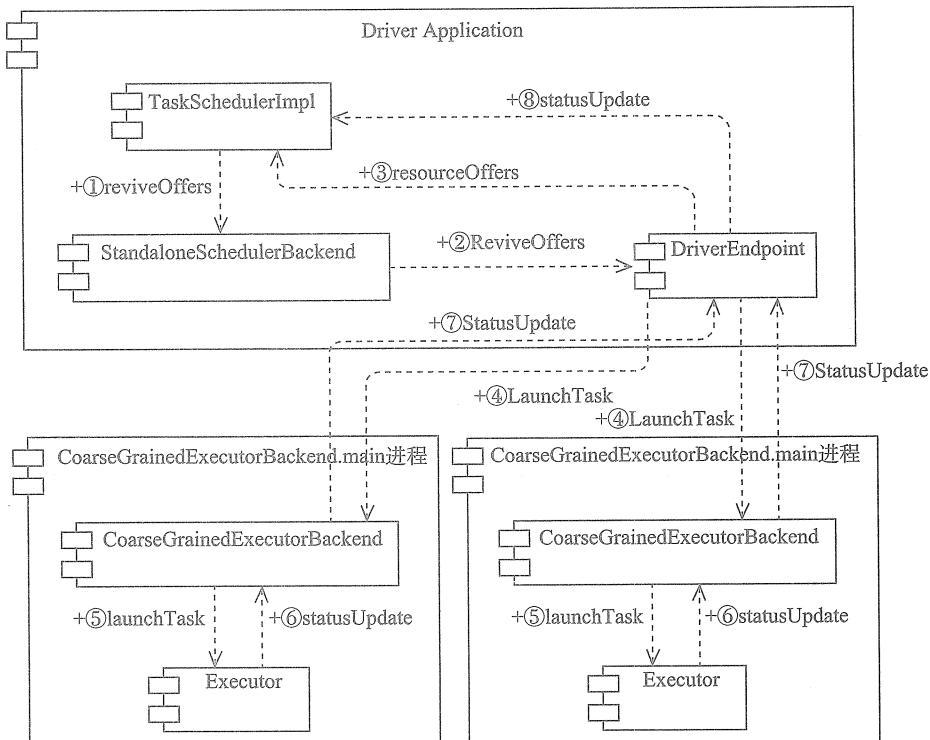


图 9-23 local-cluster 部署模式下的任务提交执行过程

序号②：`CoarseGrainedSchedulerBackend` 的 `reviveOffers` 方法只是向 `DriverEndpoint` 发送 `ReviveOffers` 消息。

序号③：`DriverEndpoint` 接收到 `ReviveOffers` 消息后，首先调用 `TaskSchedulerImpl` 的 `resourceOffers` 方法给 Task 分配资源。

序号④：给 Task 分配好资源后，向 `CoarseGrainedExecutorBackend` 实例发送 `LaunchTask` 消息。

序号⑤：`CoarseGrainedExecutorBackend` 实例接收到 `LaunchTask` 消息后，调用 `Executor` 的 `launchTask` 方法运行 Task。

序号⑥：`Executor` 在运行 Task 的过程中将不断调用 `CoarseGrainedExecutorBackend` 的 `statusUpdate` 方法更新 Task 状态。

序号⑦：`CoarseGrainedExecutorBackend` 的 `statusUpdate` 方法将向 `DriverEndpoint` 发送 `StatusUpdate` 消息。

序号⑧：`DriverEndpoint` 接收到 `StatusUpdate` 消息后调用 `TaskSchedulerImpl` 的 `statusUpdate` 方法更新 Task 状态。

## 9.12 Standalone 部署模式

local 模式只有 Driver 和 Executor，且在同一个 JVM 进程内。local-cluster 模式的 Driver、Master、Worker 也都位于同一个 JVM 进程内部。所以 local 模式和 local-cluster 模式便于开发、测试，也便于源码阅读与调试，但是却不适合在生产环境使用。Standalone 部署模式有哪些特点呢？

- Driver 是单独的进程，可以存在于集群之内，也可以存在于集群之外，对 Spark Application 的执行进行驱动。
- Master 是单独的进程，甚至应该在单独的机器节点上。Master 有多个，但同时最多只有一个处于激活状态。
- Worker 是单独的进程，也推荐在单独的机器节点上部署。

我们以 Linux 环境下启动 Standalone 模式为例。启动 Standalone 模式需要保证一定的顺序，即先启动 Master，再逐个启动 Worker。 `${SPARK_HOME}/sbin/start-master.sh` 脚本将以 `org.apache.spark.deploy.master.Master` 作为主类（已在 9.6.1 节详细介绍），启动 JVM 进程。启动 Master 的命令如图 9-24 所示。

Master 进程的默认端口是 7077，WEBUI 的端口是 8080。可以用 `jps` 命令查看 Master 进程的执行参数，输出如图 9-25 所示。

```
bin@11269:~$ start master.sh
Starting org.apache.spark.deploy.master.Master, logging to /Users/bin/install/spark-2.0.0-bin-hadoop
master.master-1-EB11269-local.out
```

图 9-24 启动 Master 进程

```
bin@11269:~$ jps -lvm
1154 org.apache.spark.deploy.Master Master -host EB11269-local -port 7077 -webui-port 8080 -exe
```

图 9-25 Master 进程的执行参数

启动完 Master，就可以启动 Worker 了。 `${SPARK_HOME}/sbin/start-slaves.sh` 或  `${SPARK_HOME}/sbin/start-slave.sh` 都能启动 Worker，区别在于 `start-slaves.sh` 可以批量启动多个 Worker。为简单起见，笔者使用 `start-slave.sh`。`start-slave.sh` 脚本以 `org.apache.spark.deploy.worker.Worker` 为主类（已在 9.7.1 节详细介绍），启动 JVM 进程。我们指定 Master 的连接地址，然后启动 Worker 的命令如图 9-26 所示。

```
bin@11269:~$ start slave.sh spark://127.0.0.1:7077
Starting org.apache.spark.deploy.worker.Worker, logging to /Users/bin/install/spark-2.0.0-bin-hadoop2.3/logs/
worker.Worker-1-EB11269-local.out
```

图 9-26 启动 Worker 进程

我们同样用 `jps -lmv` 命令查看 Worker 进程的执行参数，如图 9-27 所示。

```
1003 org.apache.spark.deploy.worker.Worker -port 7078 -host 127.0.0.1 -sparkHome
```

图 9-27 Worker 进程的执行参数

有关 Master 和 Worker 的实现分析，前文都已经详细介绍，本节我们设置 master 为 spark://127.0.0.1:7077，那么在创建 TaskSchedulerImpl 时（见代码清单 4-18）就会匹配 Standalone 模式。

Standalone 模式中，ExecutorBackend 的实现类是 CoarseGrainedExecutorBackend，SchedulerBackend 的实现类是 StandaloneSchedulerBackend。由于 Standalone 模式中的各个 Master 和 Worker 都是单独的进程，因此不再需要 local-cluster 模式中的 LocalSparkCluster 来启动位于 JVM 进程内的 Master 和 Worker。本章已经详细分析了 Master 和 Worker 的实现，下面将展示 Standalone 模式下最主要的工作原理。由于 Standalone 部署模式下的任务提交执行过程与 local-cluster 模式下的相同，所以不再展示。

### 9.12.1 Standalone 部署模式的启动过程

有了本章对 Master、Worker、StandaloneAppClient、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 Standalone 部署模式的启动过程。Standalone 部署模式能够选用的领导选举代理只能是 ZooKeeperLeaderElectionAgent，本节以 2 个 Worker、2 个 Master、1 个 Driver application 的情况，展示整个 Standalone 部署模式的启动过程，如图 9-28 所示。

这里对图 9-28 中的各个序号进行说明。

**序号①：**Standalone 部署模式中首先会启动一到多个 Master.main 函数的进程。每个 Master.main 进程中都会创建 Master 实例并注册到 Master 自己的 RpcEnv 中，在启动 Master 实例的过程中会创建 ZooKeeperLeaderElectionAgent 实例。

**序号②：**每个 Master 实例对应的 ZooKeeperLeaderElectionAgent 实例都会参与 ZooKeeper 的领导选举，最终确定正式的领导者并调用 ZooKeeperLeaderElectionAgent 的 isLeader 方法。isLeader 方法将会调用 Master 的 electedLeader 方法将 Master 选举为领导，Master 的状态被设置为激活。对于没有被选举为领导的 Master，会调用 ZooKeeperLeaderElectionAgent 的 notLeader 方法。notLeader 方法会将领导者的身份告知支持者。

**序号③：**Standalone 部署模式中会启动一到多个 Worker.main 函数的进程。每个 Worker.main 进程中都会创建 Worker 实例并注册到 Worker 自己的 RpcEnv 中，在启动 Worker 实例的过程中会向 Master 实例发送 RegisterWorker 消息。

**序号④：**Master 接收到 RegisterWorker 消息后对 WorkerInfo 进行注册。WorkerInfo 注册成功后，Master 向 Worker 回复 RegisteredWorker 消息（见 9.6.5 节）。Worker 收到 RegisteredWorker 消息后，将向 Master 发送 WorkerLatestState 消息，以便把 Worker 的最新状态汇报给 Master。根据 9.6.6 节的内容，我们知道 Master 接收到 WorkerLatestState 消息后，将对 Executor 和 Driver 进行匹配，对于不匹配的 Executor 和 Driver，将通知 Worker “杀死”。

**序号⑤：**SparkContext 创建完 TaskSchedulerImpl 后，将启动 TaskSchedulerImpl（见代码清单 4-19）。根据 7.10.3 节对启动 TaskSchedulerImpl 的分析，TaskSchedulerImpl 的 start

方法一开始就会调用 StandaloneSchedulerBackend 的 start 方法。

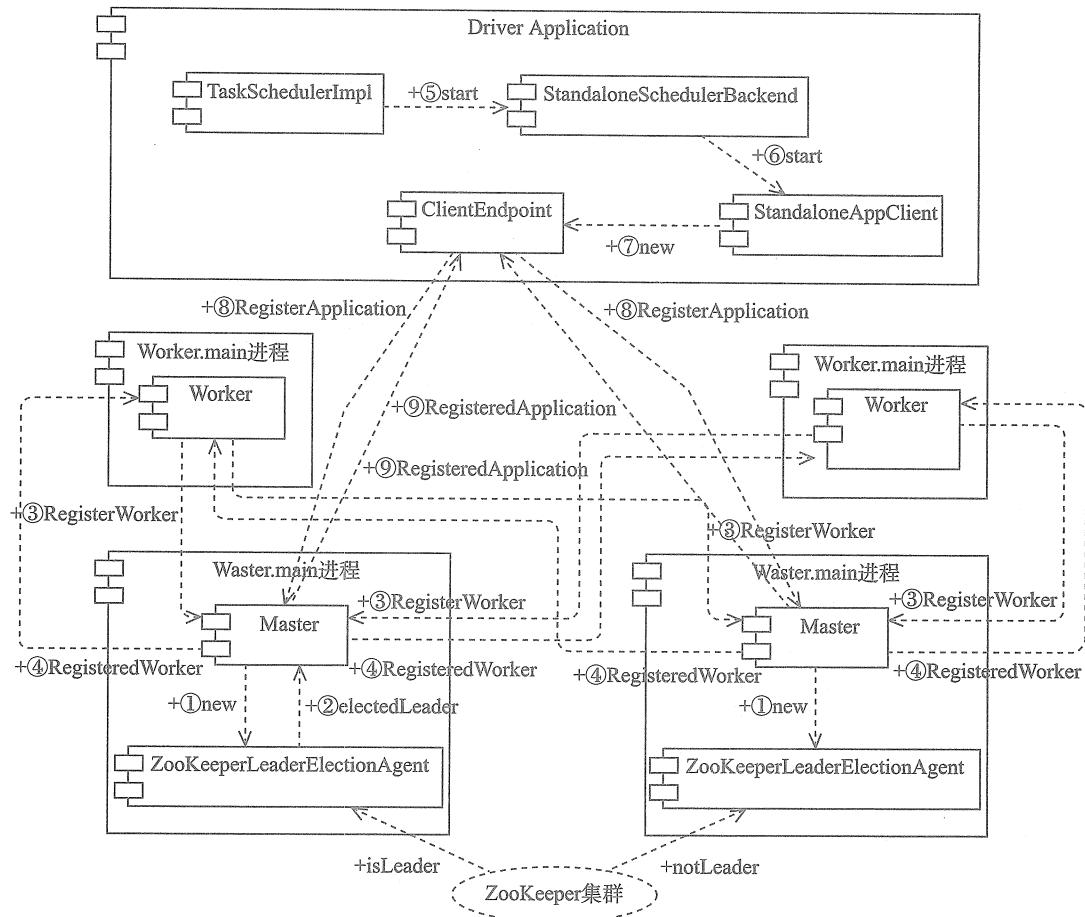


图 9-28 Standalone 部署模式的启动过程

序号⑥：通过 9.9.3 节对 StandaloneSchedulerBackend 的启动分析，StandaloneSchedulerBackend 的 start 方法中会创建并启动 StandaloneAppClient。

序号⑦：StandaloneAppClient 的 start 方法会创建 ClientEndpoint 并将 ClientEndpoint 注册到 SparkContext 的 SparkEnv 的 RpcEnv 中，进而引起 ClientEndpoint 的启动。

序号⑧：9.8.1 节详细介绍了 ClientEndpoint 的启动，ClientEndpoint 在启动时会向每个 Master 发送 RegisterApplication 消息。

序号⑨：根据 9.6.8 节的介绍，Master 收到 RegisterApplication 消息并注册成功后，会向 DriverEndpoint 发送 RegisteredApplication 消息。ClientEndpoint 的 receive 方法接收到 RegisteredApplication 消息（见代码清单 9-70）后，更新 ClientEndpoint 的 appId、registered、master 属性及 StandaloneSchedulerBackend 的 appId 属性。

### 9.12.2 Standalone 部署模式下 Executor 的分配过程

有了本章对 Master、Worker、StandaloneAppClient、DriverEndpoint、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 Standalone 部署模式下的 Executor 资源分配过程，如图 9-29 所示。

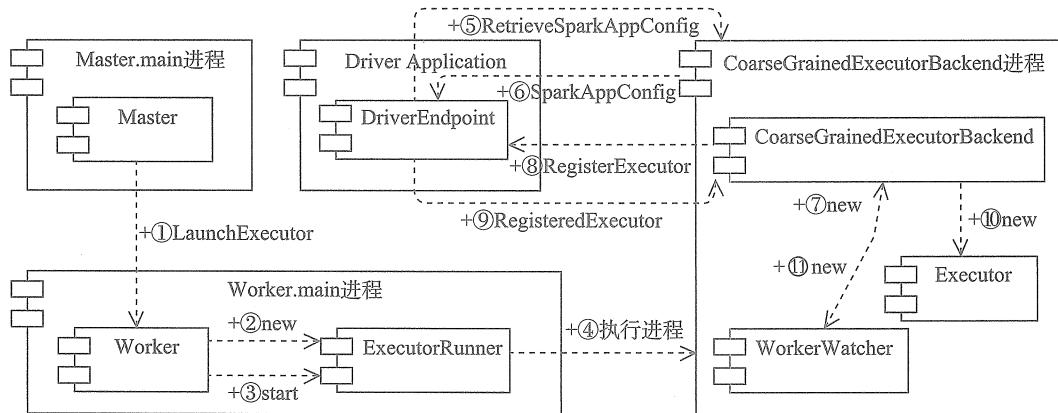


图 9-29 Standalone 部署模式下的 Executor 资源分配过程

这里对图 9-29 中的各个序号进行说明。

**序号①：**Master 的 schedule 方法在对资源调度时将调用 startExecutorsOnWorkers 方法对 Executor 的资源进行分配和调度，然后通过向 Worker 发送 LaunchExecutor 消息运行 Executor。

**序号②：**Worker 接收到 LaunchExecutor 消息后将创建 ExecutorRunner。

**序号③：**Worker 创建完 ExecutorRunner 后，将调用 ExecutorRunner 的 start 方法。

**序号④：**ExecutorRunner 的 start 方法主要创建了线程 workerThread，线程 workerThread 将构建并启动 ProcessBuilder。由于 ProcessBuilder 的命令中指定了 Java 进程的主方法是 CoarseGrainedExecutorBackend 的伴生对象的 main 函数，因此将启动执行 CoarseGrainedExecutorBackend 的 main 函数的进程。

**序号⑤：**执行 CoarseGrainedExecutorBackend 的 main 函数的进程首先创建名为 driverPropsFetcher 的 RpcEnv，然后向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息。

**序号⑥：**DriverEndpoint 接收到 RetrieveSparkAppConfig 消息后，将向执行 CoarseGrainedExecutorBackend 的 main 函数的进程回复 SparkAppConfig 消息。SparkAppConfig 消息携带着 Spark 属性信息和分配给 Executor 的密钥。

**序号⑦：**CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，将首先创建 SparkConf 和 SparkEnv，然后创建 CoarseGrainedExecutorBackend 实例并注册到 Executor 自身的 SparkEnv 的 RpcEnv。

**序号⑧：**将 CoarseGrainedExecutorBackend 实例注册到 Executor 自身的 SparkEnv 的

RpcEnv 时，会触发对 CoarseGrainedExecutorBackend 的 onStart 方法的调用，进而向 DriverEndpoint 发送 RegisterExecutor 消息以注册 Executor。

序号⑨：DriverEndpoint 接收到 RegisterExecutor 消息对 Executor 注册成功后，将向 CoarseGrainedExecutorBackend 实例回复 RegisteredExecutor 消息。

序号⑩：CoarseGrainedExecutorBackend 实例接收到 RegisteredExecutor 消息后将创建 Executor。

序号⑪：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，除了创建 CoarseGrainedExecutorBackend 实例，还将创建 WorkerWatcher，以保证执行 CoarseGrainedExecutorBackend 的 main 函数的进程在与 Worker 断开连接或发生网络错误的情况下退出。

### 9.12.3 Standalone 部署模式的资源回收

除了流式计算，大多数情况下的任务在运行一段时间之后都会完成，那么 Application 占用的资源应该被回收。Master 和 Executor 是如何感知到 Application 的退出呢？

Spark 中有两种处理方式：一种是离别之前先打声招呼，一种是不辞而别。

#### 1. 离别之前先打声招呼

这种方式很好理解，SparkContext 提供了 stop 方法用于告别，即停止各种服务和回收资源。SparkContext 的 stop 方法中有一段调用 DagScheduler 的 stop 方法的代码。

```
if (_dagScheduler != null) {
    Utils.tryLogNonFatalError {
        _dagScheduler.stop()
    }
    _dagScheduler = null
}
```

DagScheduler 的 stop 方法中会调用 TaskSchedulerImpl 的 stop 方法。

```
taskScheduler.stop()
```

TaskSchedulerImpl 的 stop 方法中会调用 StandaloneSchedulerBackend 的 stop 方法。

```
if (backend != null) {
    backend.stop()
}
```

根据 9.9.4 节的介绍，StandaloneSchedulerBackend 的 stop 方法将引起向 DriverEndpoint 发送 StopExecutors 消息。DriverEndpoint 接收到 StopExecutors 消息后，向每个 CoarseGrainedExecutorBackend 实例发送 StopExecutor 消息。CoarseGrainedExecutorBackend 的 receive 方法（见代码清单 9-91）接收到 StopExecutor 消息后，将向 CoarseGrainedExecutorBackend 自己发送 Shutdown 消息。CoarseGrainedExecutorBackend 接收到 Shutdown 消息，将启动一个线程用于调用 Executor 的 stop 方法（此方法的实现非常简单，留给感兴趣的读者）。

者自行阅读）停止 Executor。

## 2. 不辞而别

上面的分析发现 Application 只记得跟 Executor 打声招呼，却忘记了 Master。这该怎么办？下面将分析具体的解决方案。

在 5.3.5 节介绍 NettyRpcHandler 时，提到过 NettyRpcHandler 除实现了 RpcHandler 的两个 receive 方法，还实现了 exceptionCaught、channelActive 与 channelInactive 等。exceptionCaught 方法将会向 Inbox 中投递 RemoteProcessConnectionError 消息。channelActive 将会向 Inbox 中投递 RemoteProcessConnected 消息。channelInactive 将会向 Inbox 中投递 RemoteProcessDisconnected 消息。Inbox 的 process 方法（见代码清单 5-7）在处理 RemoteProcessConnected、RemoteProcessDisconnected、RemoteProcessConnectionError 三个消息时，将分别调用 RpcEndpoint 的 onConnected、onDisconnected、onNetworkError 三个方法。

Application 一旦退出，那么 Master 的 onDisconnected 方法将被调用。Master 的 onDisconnected 方法的实现如代码清单 9-94 所示。

代码清单 9-94 Master 的 onDisconnected 方法

---

```
override def onDisconnected(address: RpcAddress): Unit = {
    logInfo(s"$address got disassociated, removing it.")
    addressToWorker.get(address).foreach(removeWorker)
    addressToApp.get(address).foreach(finishApplication)
    if (state == RecoveryState.RECOVERING && canCompleteRecovery) { completeRecovery() }
}
```

---

根据代码清单 9-94，Master 将调用 finishApplication 方法处理 Application。finishApplication 方法的实现如下。

```
private def finishApplication(app: ApplicationInfo) {
    removeApplication(app, ApplicationState.FINISHED)
}
```

可以看到 finishApplication 方法中调用了 removeApplication 方法（见代码清单 9-49）移除 Application 及分配给 Application 的 Executor。

### 9.12.4 Standalone 部署模式的容错机制

在分布式系统中，由于机器数量众多，导致机器发生故障的概率很高，所以在设计任何分布式系统时都应考虑容错。本节主要针对 Standalone 部署模式的容错能力进行分析。

#### 1. Executor 异常退出

如果 Executor 异常退出，那么势必导致在此 Executor 上执行的任务无法运行。曾经在 9.7.6 节介绍 ExecutorRunner 的 fetchAndRunExecutor 方法时，介绍过 CoarseGrained-ExecutorBackend 进程退出后，ExecutorRunner 会向 Worker 发送 ExecutorStateChanged 消

息。根据 9.7.7 节的介绍，Worker 收到 ExecutorStateChanged 消息后将向 Master 转发 ExecutorStateChanged 消息。根据 9.6.10 节的介绍，Master 收到 ExecutorStateChanged 消息后对退出状态的处理步骤如下。

- 1) 找到使用此 Executor 的 Application 所对应的 ApplicationInfo，以及 Executor 对应的 ExecutorInfo。
- 2) 将 ExecutorInfo 的状态改为 EXITED。
- 3) EXITED 也属于 Executor 完成状态，所以会将 ExecutorInfo 从 ApplicationInfo 和 WorkerInfo 中移除。
- 4) 由于 Executor 是非正常退出，所以重新调用 schedule 方法给 Application 进行资源调度。

Executor 异常退出的容错处理可以用图 9-30 表示。

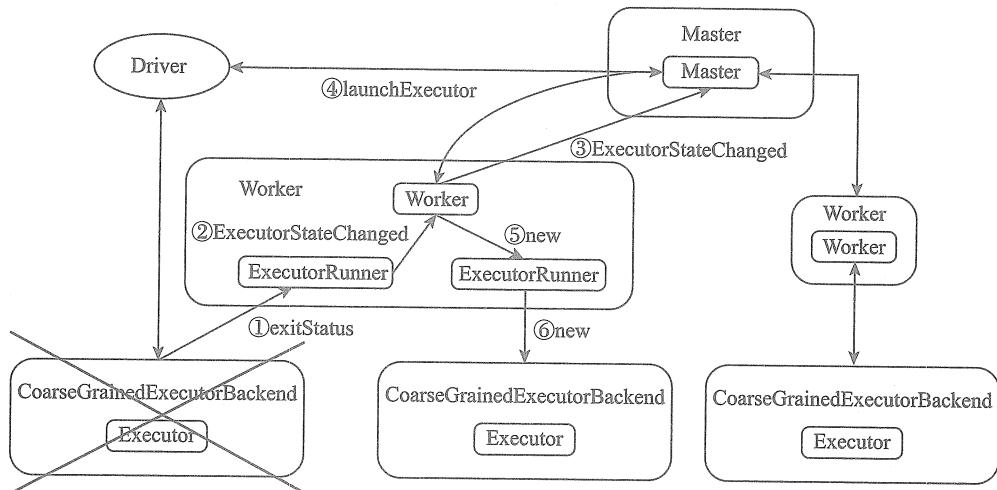


图 9-30 Executor 异常退出的容错处理

第⑥步之后 CoarseGrainedExecutorBackend 将向 Driver 注册 Executor，注册过程已在前面讲过。

## 2. Worker 异常退出

当 Worker 进程退出时，会发生什么呢？还记得 9.7.6 节中 fetchAndRunExecutor 方法中捕获到异常的处理吗？它将在 Worker 进程退出前调用 killProcess 方法（见代码清单 9-66），主动“杀死”CoarseGrainedExecutorBackend 进程，然后收到进程返回的退出状态（KILLED）后向 Worker 发送 ExecutorStateChanged 消息。由于 Worker 退出了，所以不会有 Heartbeat 消息发送给 Master，所以无法更新 WorkerInfo 的最后心跳时间（lastHeartbeat）。根据 timeOutDeadWorkers 方法（见代码清单 9-28）的实现，Master 会调用 removeWorker 方法（见代码清单 9-48）删除长期失联的 Worker 的 WorkerInfo，并将此 Worker 的所有 Executor

以 LOST 状态同步更新到它们服务的 Application。最后 Master 还会为 WorkerInfo 所服务的 Application 重新调度，分配到其他 Worker 上。整个过程可以用图 9-31 表示。

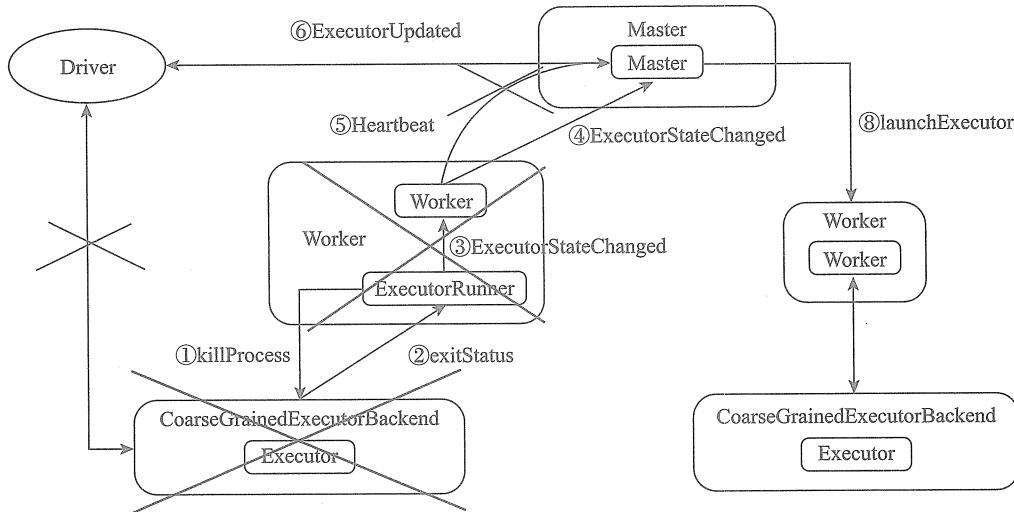


图 9-31 Worker 异常退出的容错处理

### 3. Master 异常退出

假如我们的 Standalone 模式下的集群只有一个 Master，当 Master 退出时会发生什么呢？

1) 当 Executor 上的任务执行完毕，需要对资源回收，按照 9.12.3 节的内容，如果主动调用 SparkContext 的 stop 方法，Application 最终会向给自己服务的 Executor 发送 StopExecutor 消息对资源回收，Master 虽然跑路了，但是 Driver Application 依然持有这些 Executor，所以没有影响。如果不辞而别，那么 CoarseGrainedExecutorBackend 将可以收到 RemoteProcessDisconnected 消息后退出进程。看来 Master 退出，如果 Worker、Executor 正常运行，则对于资源回收没有影响。

2) 此时如果再发生 Executor 异常退出问题，Worker 无法通过 ExecutorStateChanged 消息促使 Master 重新将 Application 调度运行 Executor，Application 只能眼巴巴看着自己提交的任务无法执行。而 Executor 占用的资源也无法回收。

3) 此时如果又发生 Worker 异常退出问题，那么 Worker 和 Executor 都将停止服务，由于无法通知 Master 重新将 Application 调度到其他 Worker 上，Application 提交的任务也将无法执行。Worker 虽然杀掉了 Executor，但是 Worker 的资源将无法被其他 Application 使用。

4) 有新的 Driver 需要提交任务则无法成功。

基于以上分析，发现 Standalone 部署集群只有一个 Master 是万万不可以的，这也称为单点故障问题。解决此问题的常用方式是采用多个 Master，但同时只有一个 Master 负责整个集群的调度、资源管理工作，其他 Master 作为热备。

在 9.4 节和 9.5 节中我们曾经详细分析了故障恢复的持久化引擎（persistenceEngine）

和领导选举代理 (leaderElectionAgent)。在分布式环境下，我们必然要选择 ZooKeeperPersistenceEngine 和 ZooKeeperLeaderElectionAgent。

如果选择 ZooKeeper 作为热备 (HA) 的方案，在集群启动时需要在 spark-env.sh 增加如下所示的配置。

```
spark.deploy.recoveryMode= ZOOKEEPER
spark.deploy.ZooKeeper.url=cms_zk_server_1:2181, cms_zk_server_2:2181, cms_zk_
server_3:2181
spark.deploy.ZooKeeper.dir=/usr/zk/persist
```

上面配置中的 spark.deploy.ZooKeeper.url 用于配置使用的 ZooKeeper 集群，理论上需要奇数个 ZooKeeper 节点。spark.deploy.ZooKeeper.dir 是对 Spark 集群状态进行持久化的根目录。

结合 9.6.3 节的内容，使用 ZooKeeper 作为热备的选举方案可以用图 9-32 表示。

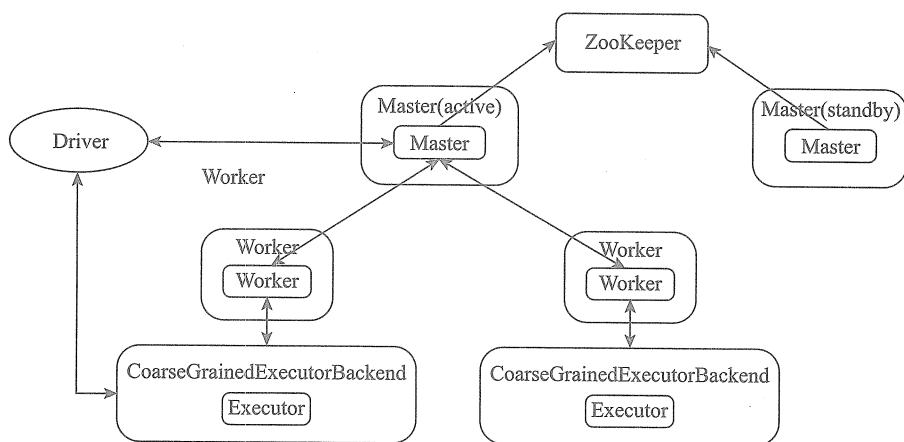


图 9-32 ZooKeeper 热备和选举示意图

ZooKeeper 热备的实现已在 9.6.3 节和 9.7.4 节详细分析过。我们也可以回顾图 9-18。

## 9.13 其他部署方案

本节介绍 Spark 运行在第三方资源管理集群上的部署方案。Spark 目前支持运行在 YARN 和 Mesos 甚至运行在 MRv1 框架上。对于 MRv1 框架运行 Spark 的内容，有兴趣的读者可以自行研究，本文不做过多介绍。

### 9.13.1 YARN

在本书第 2 章，笔者曾经简单介绍了 MRv1 的缺陷及 MRv2 的改进。MRv1 的运行环境由 JobTracker 和 TaskTracker 两类服务组成，JobTracker 负责资源和任务的管理与调度，TaskTracker 负责单个节点的资源管理和任务执行。在 YARN 中，JobTracker 被分为两部分：

ResourceManager (RM) 和 ApplicationMaster (AM)。RM 负责整个集群的资源管理和调度，AM 负责具体应用程序的任务划分、调度等工作，如图 9-33 所示。

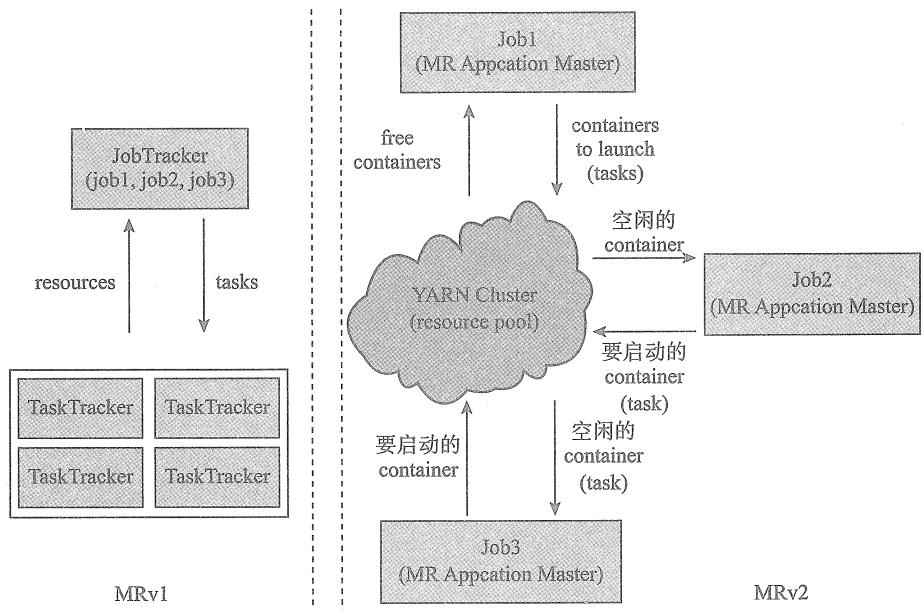


图 9-33 MRv1 与 Yarn 的运行时环境的对比

Yarn 的架构如图 9-34 所示。

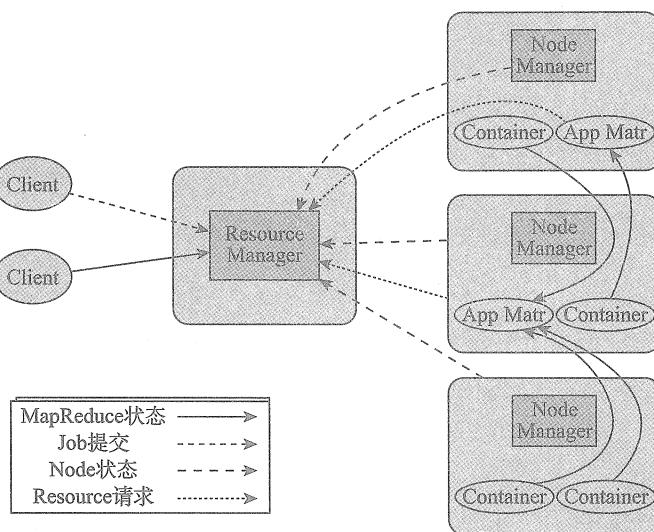


图 9-34 YARN 的架构

YARN 的基本结构如下。

- ResourceManager (RM)：全局资源管理器，负责整个集群的资源管理与分配。RM 由资源调度器和应用程序管理器组成。资源调度器将集群资源分配给各个应用程序，资源的分配单位不再是 MRv1 中的 slot，而是 Container。Container 是对 CPU、内存等资源的封装。应用程序管理器负责管理整个系统的应用程序，如处理程序提交，与资源调度器沟通后为应用程序启动 ApplicationMaster 等。
- ApplicationMaster (AM)：用户提交的每个应用程序都有一个 AM，它会与 RM 通信获取资源，将作业划分为更细粒度的任务，与 NodeManager (NM) 通信启动或停止任务，监控失败任务为其重新申请资源后重新启动等。
- NodeManager (NM)：单个节点上的资源与任务管理器，它负责向 RM 定时汇报本节点的资源使用情况及各个 Container 的状态，也接受处理 AM 的启动或停止任务请求。

YARN 对于支持的 MapReduce 框架是可插拔的，Spark 对于集群管理器也支持可插拔，两者不谋而合。有关 YARN 的安装和使用，读者可自行翻阅相关书籍。假设我们已经部署好了 YARN 集群，根据图 9-34 所有应用程序需要一个 Application Master，这个 Application Master 位于 YARN 集群的一个节点上，负责管理资源申请与资源的二级分配调度。Spark 也提供了 Application Master 的实现 ApplicationMaster。与 YARN 集成时整个 Spark 集群的启动顺序如下。

- 1 ) 将 Spark 提供的 ApplicationMaster 在 YARN 集群中启动（见代码清单 9-95）。
- 2 ) ApplicationMaster 向 ResourceManager 申请 Container。
- 3 ) 申请 Container 成功后，向具体的 NodeManager 发送指令启动 Container。
- 4 ) ApplicationMaster 启动对各个运行的 ContainerExecutor 进行监控。

代码清单9-95 ApplicationMaster的main方法

---

```
def main(args: Array[String]): Unit = {
    SignalUtils.registerLogger(log)
    val amArgs = new ApplicationMasterArguments(args)

    if (amArgs.propertiesFile != null) {
        Utils.getPropertiesFromFile(amArgs.propertiesFile).foreach { case (k, v) =>
            sys.props(k) = v
        }
    }
    SparkHadoopUtil.get.runAsSparkUser { () =>
        master = new ApplicationMaster(amArgs, new YarnRMClient)
        System.exit(master.run())
    }
}
```

---

ApplicationMaster 的 registerAM 方法（见代码清单 9-96）用于向 YARN 注册 AM。

代码清单9-96 ApplicationMaster的registerAM方法

---

```
private def registerAM(
    _sparkConf: SparkConf,
```

```

    _rpcEnv: RpcEnv,
    driverRef: RpcEndpointRef,
    uiAddress: String,
    securityMgr: SecurityManager) = {
  val appId = client.getAttemptId().getApplicationId().toString()
  val attemptId = client.getAttemptId().getAttemptId().toString()
  val historyAddress =
    _sparkConf.get(HISTORY_SERVER_ADDRESS)
      .map { text => SparkHadoopUtil.get.substituteHadoopVariables(text, yarn-
        Conf) }
      .map { address => s"${address}${HistoryServer.UI_PATH_PREFIX}/${appId}/
        ${attemptId}" }
      .getOrElse("")
  val driverUrl = RpcEndpointAddress(
    _sparkConf.get("spark.driver.host"),
    _sparkConf.get("spark.driver.port").toInt,
    CoarseGrainedSchedulerBackend.ENDPOINT_NAME).toString

  logInfo {
    val executorMemory = sparkConf.get(EXECUTOR_MEMORY).toInt
    val executorCores = sparkConf.get(EXECUTOR_CORES)
    val dummyRunner = new ExecutorRunnable(None, yarnConf, sparkConf, driverUrl,
      "<executorId>",
      "<hostname>", executorMemory, executorCores, appId, securityMgr, local-
      Resources)
    dummyRunner.launchContextDebugInfo()
  }

  allocator = client.register(driverUrl, driverRef, yarnConf, _sparkConf, uiAdd-
    ress,
    historyAddress, securityMgr, localResources)

  allocator.allocateResources()
  reporterThread = launchReporterThread()
}

```

下面我们设置 master 为 yarn，那么根据代码清单 4-18，在创建 TaskSchedulerImpl 时将无法匹配 local、local-cluster、Standalone 等模式，因而只能匹配最后的 masterUrl。根据代码清单 4-18，首先调用 getClusterManager 方法（见代码清单 9-97）获取集群管理器。

#### 代码清单9-97 SparkContext的getClusterManager方法

---

```

private def getClusterManager(url: String): Option[ExternalClusterManager] = {
  val loader = Utils.getContextOrSparkClassLoader
  val serviceLoaders =
    ServiceLoader.load(classOf[ExternalClusterManager], loader).asScala.filter(_.
      canCreate(url))
  if (serviceLoaders.size > 1) {
    throw new SparkException(
      s"Multiple external cluster managers registered for the url $url: $service-
      Loaders")
  }
  serviceLoaders.headOption
}

```

---

 }

根据上述代码，`getClusterManager`方法将通过类加载器加载所有实现了特质 `ExternalClusterManager` 的类，并调用实现的 `canCreate` 方法进行过滤。此时由于 `YarnClusterManager` 的 `canCreate` 方法将返回 `true`，因此获得的外部集群管理器就是 `YarnClusterManager`。`YarnClusterManager` 的 `canCreate` 方法的实现如下。

```
override def canCreate(masterURL: String): Boolean = {
  masterURL == "yarn"
}
```

根据代码清单 4-18，在获得 `ExternalClusterManager` 的具体实现类后，将调用 `ExternalClusterManager` 实现类的 `createTaskScheduler` 方法创建 `TaskScheduler`。`YarnClusterManager` 的 `createTaskScheduler` 方法的实现如下。

```
override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler =
  {
    sc.deployMode match {
      case "cluster" => new YarnClusterScheduler(sc)
      case "client"  => new YarnScheduler(sc)
      case _          => throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")
    }
}
```

根据上述代码，如果 `SparkContext` 的 `deployMode` 属性是 `cluster`，那么创建 `YarnClusterScheduler`；如果 `SparkContext` 的 `deployMode` 属性是 `client`，那么创建 `YarnScheduler`。

根据代码清单 4-18，在创建了 `TaskScheduler` 后，将调用 `ExternalClusterManager` 实现类的 `createSchedulerBackend` 方法创建 `SchedulerBackend`。`YarnClusterManager` 的 `createSchedulerBackend` 方法的实现如下。

```
override def createSchedulerBackend(sc: SparkContext,
                                    masterURL: String,
                                    scheduler: TaskScheduler): SchedulerBackend = {
  sc.deployMode match {
    case "cluster" =>
      new YarnClusterSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)
    case "client"  =>
      new YarnClientSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)
    case _          =>
      throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")
  }
}
```

根据上述代码，如果 `SparkContext` 的 `deployMode` 属性是 `cluster`，那么创建 `YarnClusterSchedulerBackend`；如果 `SparkContext` 的 `deployMode` 属性是 `client`，那么创建 `YarnClientSchedulerBackend`。

根据代码清单 4-18，在创建完 `SchedulerBackend` 后，将调用 `ExternalClusterManager`

实现类的 initialize 方法进行初始化。YarnClusterManager 的 initialize 方法的实现如下。

```
override def initialize(scheduler: TaskScheduler, backend: SchedulerBackend): Unit
  = {
    scheduler.asInstanceOf[TaskSchedulerImpl].initialize(backend)
}
```

根据上述代码，实际上调用了 TaskSchedulerImpl 的 initialize 方法。

根据 7.10.3 节的内容，我们知道在启动调度系统时会调用 YarnClusterSchedulerBackend 的 start 方法，其中主要确定了期望获得的 Executor 的数量，实现如下。

```
override def start() {
  val attemptId = ApplicationMaster.getAttemptId
  bindToYarn(attemptId.getApplicationId(), Some(attemptId))
  super.start()
  totalExpectedExecutors = YarnSparkHadoopUtil.getInitialTargetExecutorNumber(sc.
    conf)
}
```

Driver 初始化完毕会向 ApplicationMaster 进行注册，在 Yarn 部署模式中 Worker 已被 NodeManager 替代，ApplicationMaster 给 Application 分配资源主要借助于代码清单 9-96 中向 YARN 注册时返回的 allocator（类型为 YarnAllocator）。

关于 YARN 部署模式就简单介绍这些，更多内容可以访问 <http://hadoop.apache.org/index.html> 了解。

### 9.13.2 Mesos

Mesos 是诞生于 UC Berkeley 的另一个研究项目，现已成为 Apache Incubator 中的项目。Mesos 是一个集群管理器，提供了有效的、跨分布式应用或框架的资源隔离和共享，可以灵活支持 Hadoop、MPI、Hypertable、Spark 等。使用 ZooKeeper 实现容错复制，采用 Linux Container 对内存和 CPU 进行隔离。

Twitter 已经在使用 Mesos 管理集群资源。

Mesos 为了简化设计，采用 Master/Slave 架构，为了解决 Master 的单点故障，Master 非常轻量，仅保存了各种计算框架（如 Hadoop、MPI、Hypertable、Spark）和 Mesos slave 的一些状态，而这些状态很容易通过 framework 和 slave 重新注册而重构，因此很容易使用 ZooKeeper 解决 Mesos master 的单点故障问题。Mesos 的整体架构如图 9-35 所示。<sup>⊖</sup>

Mesos master 实际上是一个全局资源调度器，采用某种策略将某个 slave 上的空闲资源分配给各个计算框架，各种计算框架通过自己的调度器向 Mesos master 注册，以接入到 Mesos 中；而 Mesos slave 的主要功能是汇报任务的状态和启动各个计算框架的 executor（比如 Spark 的 Executor）。整个 Mesos 系统采用了双层调度框架：第一层，由 Mesos 将资源分配给框架；第二层，框架自己的调度器将资源分配给自己内部的任务。

---

<sup>⊖</sup> 部分内容参考自博客 <http://jiezhu2007.iteye.com/blog/2049073>。

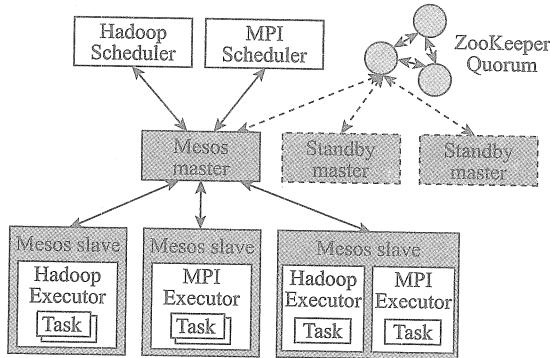


图 9-35 Mesos 的整体架构

在 Mesos 中，各种计算框架是完全融入 Mesos 中的，也就是说，如果你想在 Mesos 中添加一个新的计算框架，首先需要在 Mesos 中部署一套该框架。

如果我们设置 master 为 mesos://，那么在创建 TaskSchedulerImpl 时也只会匹配 masterUrl。由于 MesosClusterManager 的 canCreate 方法的实现如下。

```

override def canCreate(masterURL: String): Boolean = {
  masterURL.startsWith("mesos")
}
  
```

所以在调用 getClusterManager 方法后获得 MesosClusterManager。MesosClusterManager 的 createTaskScheduler 方法将创建 TaskSchedulerImpl，代码如下。

```

override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler =
  new TaskSchedulerImpl(sc)
  
```

MesosClusterManager 的 createSchedulerBackend 方法将根据 spark.mesos.coarse 属性的值（默认为 true）决定是创建 MesosCoarseGrainedSchedulerBackend 还是 MesosFineGrained SchedulerBackend。MesosClusterManager 的 createSchedulerBackend 方法的实现如下。

```

override def createSchedulerBackend(sc: SparkContext,
  masterURL: String,
  scheduler: TaskScheduler): SchedulerBackend = {
  require(!sc.conf.get(IO_ENCRYPTION_ENABLED),
    "I/O encryption is currently not supported in Mesos.")

  val mesosUrl = MESOS_REGEX.findFirstMatchIn(masterURL).get.group(1)
  val coarse = sc.conf.getBoolean("spark.mesos.coarse", defaultValue = true)
  if (coarse) {
    new MesosCoarseGrainedSchedulerBackend(
      scheduler.asInstanceOf[TaskSchedulerImpl], sc, mesosUrl, sc.env.securityManager)
  } else {
    new MesosFineGrainedSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc, mesosUrl)
  }
  
```

```

    }
}

```

MesosClusterManager 的 initialize 方法与 YarnClusterManager 的 initialize 方法的实现完全一样，这里就不再列出了。

根据 7.10.3 节的内容，我们知道在启动调度系统时会调用 SchedulerBackend 的 start 方法。此处以 MesosCoarseGrainedSchedulerBackend 为例来看看它的 start 方法的实现，代码如下。

```

override def start() {
  super.start()
  val driver = createSchedulerDriver( master, MesosCoarseGrainedSchedulerBackend.
    this,
    sc.sparkUser, sc.appName, sc.conf,
    sc.conf.getOption("spark.mesos.driver.webui.url").orElse(sc.ui.map(_.appUI-
      Address)),
    None, None, sc.conf.getOption("spark.mesos.driver.frameworkId")
  )

  unsetFrameworkID(sc)
  startScheduler(driver)
}

```

根据上述代码，MesosCoarseGrainedSchedulerBackend 的 start 方法通过调用 Mesos 的 API 完成 Driver 的注册。有关 Mesos 的 API 细节，请访问官网 <http://mesos.apache.org/>。

## 9.14 小结

我们最先了解 local 部署模式，然后精读 local-cluster 部署模式，最后分析 Standalone 部署模式。这样能够由简入难，由浅入深，尽可能帮助读者降低阅读 Spark 源码与理解部署模式架构原理的门槛。local 部署模式和 local-cluster 部署模式不能用于生成，为了简单，生成中可以选择使用 Standalone 部署模式。如果对 YARN 和 Mesos 也有一定的了解，那么使用 YARN 和 Mesos 也非常不错。

资源调度、资源回收及容错机制的源码剖析，除了能让我们理解 Spark 为什么能拥有很高的稳定性、可用性及容错能力外，其中各种架构设计的思想值得做架构设计的开发人员学习。