

第4章 Chapter 4

SparkContext 的初始化

“合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。”

——《道德经》

本章导读

两千多年前，道家就已经总结出万事万物都是从最基本或最简单的工作开始的。参天大树最初是从一棵树芽生长而来的；摩天大厦是由一砖一瓦逐步建造而成的；如果你想游历祖国的大好河山，那么就得像徐霞客一样从迈出第一步开始。无论 Spark 提供的功能多么丰富，如果想要使用它，那么第一步就是对 SparkContext 进行初始化。对 SparkContext 进行初始化是你使用 Spark 所需要迈出的第一步，这一步看起来是多么的微不足道。

SparkContext 的初始化过程实际也是对 Driver 的初始化，这一准备工作是 Spark 应用程序提交与执行的前提。SparkContext 的初始化过程囊括了内部各个组件的初始化与准备，这些组件具有不同的功能，它们涉及网络通信、分布式、消息、存储、计算、缓存、度量、清理、文件服务、Web UI 的方方面面。这一切初始化工作都由 SparkContext 负责，所以我们不用过于担心它的内部复杂性。本章在介绍 SparkContext 初始化过程的同时，将向读者介绍各个组件的作用。有些组件只会作简单的介绍，后面的章节会安排详细的分析。

本章主要讲解的内容如下。

- SparkContext 概述。
- 创建 Spark 环境。
- Spark UI。

- 其他组件的创建和启动。
- 环境更新。

4.1 SparkContext 概述

Spark 应用程序的提交离不开 Spark Driver，后者是驱动应用程序在 Spark 集群上执行的原动力。了解 Spark Driver 的初始化，有助于读者理解 Spark 应用程序与 Spark Driver 的关系。

Spark Driver 的初始化始终围绕着 SparkContext 的初始化。SparkContext 可以算得上是 Spark 应用程序的发动机引擎，轿车要想跑起来，首先要启动发动机。SparkContext 初始化完毕，才能向 Spark 集群提交应用程序。发动机只需以较低的转速，就可以在平坦的公路上游刃有余；在山区，你可能需要一台能够提供大功率的发动机，才能满足你转山的体验。发动机的参数都是通过驾驶员操作油门、档位等传送给发动机的，而 SparkContext 的配置参数则由 SparkConf 负责，SparkConf 就是你的操作面板。

SparkContext 是 Spark 中的元老级 API，从 0.x.x 版本就已经存在。有过 Spark 使用经验的部分读者也许感觉 SparkContext 已经太老了，然而 SparkContext 始终跟随着 Spark 的迭代不断向前。SparkContext 的内部“血液”也发生了很多翻天覆地的变化，有些内部组件废弃了，有些内部组件有了一些优化，而且还会不断地输入一些新鲜的“血液”。希望刚才这些描述没有吓到 Spark 的老用户，因为 Spark 的灵魂——Spark 核心原理，依然是那么令人熟悉。

在讲解 SparkContext 的初始化过程之前，我们先来认识下 SparkContext 中的各个组成部分，如图 4-1 所示。

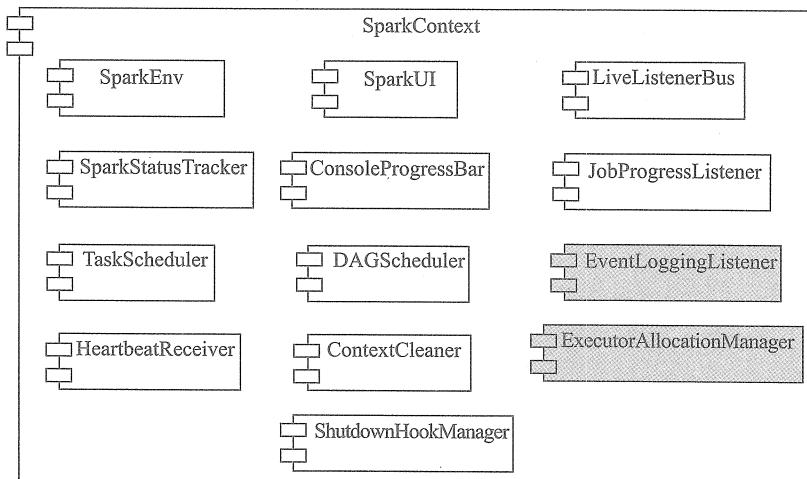


图 4-1 SparkContext 的组成[⊖]

[⊖] 图中深色的组件是 SparkContext 中的可选组件。

从图 4-1 可以知道，SparkContext 主要由以下部分组成。

- ❑ SparkEnv：Spark 运行时环境。Executor 是处理任务的执行器，它依赖于 SparkEnv 提供的运行时环境。此外，在 Driver 中也包含了 SparkEnv，这是为了保证 local 模式下任务的执行。SparkEnv 内部包含了很多组件，例如，serializerManager、RpcEnv、BlockManager、mapOutputTracker 等，在这里只需要认识它们即可，第 5 章将会对这些组件进行具体介绍。
- ❑ LiveListenerBus：SparkContext 中的事件总线，可以接收各个使用方的事件，并且通过异步方式对事件进行匹配后调用 SparkListener 的不同方法。LiveListenerBus 的具体内容已经在 3.3 节详细介绍，此处不再赘述。
- ❑ SparkUI：Spark 的用户界面。SparkUI 间接依赖于计算引擎、调度系统、存储体系，作业（Job）、阶段（Stage）、存储、执行器（Executor）等组件的监控数据都会以 SparkListenerEvent 的形式投递到 LiveListenerBus 中，SparkUI 将从各个 SparkListener 中读取数据并显示到 Web 界面。
- ❑ SparkStatusTracker：提供对作业、Stage（阶段）等的监控信息。SparkStatusTracker 是一个低级的 API，这意味着只能提供非常脆弱的一致性机制。
- ❑ ConsoleProgressBar：利用 SparkStatusTracker 的 API，在控制台展示 Stage 的进度。由于 SparkStatusTracker 存在的一致性问题，所以 ConsoleProgressBar 在控制台的显示往往有一定的时延。
- ❑ DAGScheduler：DAG 调度器，是调度系统中的重要组件之一，负责创建 Job，将 DAG 中的 RDD 划分到不同的 Stage、提交 Stage 等。SparkUI 中有关 Job 和 Stage 的监控数据都来自 DAGScheduler。
- ❑ TaskScheduler：任务调度器，是调度系统中的重要组件之一。TaskScheduler 按照调度算法对集群管理器已经分配给应用程序的资源进行二次调度后分配给任务。TaskScheduler 调度的 Task 是由 DAGScheduler 创建的，所以 DAGScheduler 是 TaskScheduler 的前置调度。
- ❑ HeartbeatReceiver：心跳接收器。所有 Executor 都会向 HeartbeatReceiver 发送心跳信息，HeartbeatReceiver 接收到 Executor 的心跳信息后，首先更新 Executor 的最后可见时间，然后将此信息交给 TaskScheduler 作进一步处理。
- ❑ ContextCleaner：上下文清理器。ContextCleaner 实际用异步方式清理那些超出应用作用域范围的 RDD、ShuffleDependency 和 Broadcast 等信息。
- ❑ JobProgressListener：作业进度监听器。JobProgressListener 在 3.3.3 节介绍 SparkListener 的继承体系时提到过，根据之前的介绍我们知道，JobProgressListener 将注册到 LiveListenerBus 中作为事件监听器之一使用。
- ❑ EventLoggingListener：将事件持久化到存储的监听器，是 SparkContext 中的可选组件。当 spark.eventLog.enabled 属性为 true 时启用。

- ExecutorAllocationManager：Executor 动态分配管理器。顾名思义，可以根据工作负载动态调整 Executor 的数量。在配置 spark.dynamicAllocation.enabled 属性为 true 的前提下，在非 local 模式下或者当 spark.dynamicAllocation.testing 属性为 true 时启用。
- ShutdownHookManager：用于设置关闭钩子的管理器。可以给应用设置关闭钩子，这样就可以在 JVM 进程退出时，执行一些清理工作。
- 除了以上介绍的这些 SparkContext 内部组件，SparkContext 内部还包括以下属性。
- creationSite [⊖]：类型为 CallSite，其中保存着线程栈中最靠近栈顶的用户定义的类及最靠近栈底的 Scala 或者 Spark 核心类信息，CallSite 的 shortForm 属性保存着以上信息的简短描述，CallSite 的 longForm 属性则保存着以上信息的完整描述。Spark 自带的 examples 项目中有对单词进行计数的应用例子 JavaWordCount，运行 JavaWordCount 得到的 CallSite 对象的属性值分别如下。
 - shortForm: getOrCreate at JavaWordCount.java:48。
 - longForm : org.apache.spark.sql.SparkSession\$Builder.getOrCreate(SparkSession.scala:860) org.apache.spark.examples.JavaWordCount.main(JavaWordCount.java:48)。
- allowMultipleContexts：是否允许多个 SparkContext 实例。默认为 false，可以通过属性 spark.driver.allowMultipleContexts 来控制。
- startTime：SparkContext 启动的时间戳。
- stopped：标记 SparkContext 是否已经停止的状态，采用原子类型 AtomicBoolean。
- addedFiles：用于每个本地文件的 URL 与添加此文件到 addedFiles 时的时间戳之间的映射缓存。
- addedJars：用于每个本地 Jar 文件的 URL 与添加此文件到 addedJars 时的时间戳之间的映射缓存。
- persistentRdds：用于对所有持久化的 RDD 保持跟踪。
- executorEnvs：用于存储环境变量。executorEnvs 中环境变量都将传递给执行任务的 Executor 使用。
- sparkUser：当前系统的登录用户，也可以通过系统环境变量 SPARK_USER 进行设置。这里使用的 Utils 的 getCurrentUserName 方法的更多介绍，请阅读附录 A。
- checkpointDir：RDD 计算过程中保存检查点时所需要的目录。
- localProperties：由 InheritableThreadLocal 保护的线程本地变量，其中的属性值可以沿着线程栈传递下去，供用户使用。
- _conf：SparkContext 的配置，通过调用 SparkConf 的 clone 方法的克隆体。在 SparkContext 初始化的过程中会对 conf 中的配置信息做校验，例如，用户必须给自己

[⊖] creationSite 通过调用 Utils.getCallSite 获得，Utils.getCallSite 的详细信息见附录 A。

的应用程序设置 spark.master (采用的部署模式) 和 spark.app.name (用户应用的名称); 用户设置的 spark.master 属性为 yarn 时, spark.submit.deployMode 属性必须为 cluster, 且必须设置 spark.yarn.app.id 属性。

- ❑ _jars: 用户设置的 Jar 文件。当用户选择的部署模式是 YARN 时, _jars 是由 spark.jars 属性指定的 Jar 文件和 spark.yarn.dist.jars 属性指定的 Jar 文件的并集。其他模式下只采用由 spark.jars 属性指定的 Jar 文件。这里使用了 Utils 的 getUserJars 方法, 其具体介绍请阅读附录 A。
- ❑ _files: 用户设置的文件。可以使用 spark.files 属性进行指定。
- ❑ _eventLogDir: 事件日志的路径。当 spark.eventLog.enabled 属性为 true 时启用。默认为 /tmp/spark-events, 也可以通过 spark.eventLog.dir 属性指定。
- ❑ _eventLogCodec: 事件日志的压缩算法。当 spark.eventLog.enabled 属性与 spark.eventLog.compress 属性皆为 true 时启用。压缩算法默认为 lz4, 也可以通过 spark.io.compression.codec 属性指定。Spark 目前支持的压缩算法包括 lzf、snappy 和 lz4 这 3 种。
- ❑ _hadoopConfiguration: Hadoop 的配置信息, 具体根据 Hadoop (Hadoop 2.0 之前的版本) 和 Hadoop YARN (Hadoop2.0+ 的版本) 的环境有所区别。如果系统属性 SPARK_YARN_MODE 为 true 或者环境变量 SPARK_YARN_MODE 为 true, 那么将会是 YARN 的配置, 否则为 Hadoop 配置。
- ❑ _executorMemory: Executor 的内存大小。默认值为 1024MB。可以通过设置环境变量 (SPARK_MEM 或者 SPARK_EXECUTOR_MEMORY) 或者 spark.executor.memory 属性指定。其中, spark.executor.memory 的优先级最高, SPARK_EXECUTOR_MEMORY 次之, SPARK_MEM 是老版本 Spark 遗留下来的配置方式, 未来将会废弃。
- ❑ _applicationId: 当前应用的标识。TaskScheduler 启动后会创建应用标识, SparkContext 中的 _applicationId 就是通过调用 TaskScheduler 的 applicationId 方法获得的。
- ❑ _applicationAttemptId: 当前应用尝试执行的标识。Spark Driver 在执行时会多次尝试执行, 每次尝试都将生成一个标识来代表应用尝试执行的身份。
- ❑ _listenerBusStarted: LiveListenerBus 是否已经启动的标记。
- ❑ nextShuffleId: 类型为 AtomicInteger, 用于生成下一个 Shuffle 的身份标识。
- ❑ nextRddId: 类型为 AtomicInteger, 用于生成下一个 RDD 的身份标识。

4.2 创建 Spark 环境

在 Spark 中, 凡是需要执行任务的地方就需要 SparkEnv。在生产环境中, SparkEnv 往

往运行于不同节点的 Executor 中。但是由于 local 模式在本地执行的需要，因此在 Driver 本地的 Executor 也需要 SparkEnv。SparkContext 中创建 SparkEnv 的实现如代码清单 4-1 所示。

代码清单4-1 创建SparkEnv

```
private[spark] val listenerBus = new LiveListenerBus(this)
    _jobProgressListener = new JobProgressListener(_conf)
    listenerBus.addListener(jobProgressListener)
    _env = createSparkEnv(_conf, isLocal, listenerBus)
    SparkEnv.set(_env)
```

因为 SparkEnv 内的很多组件都将向 LiveListenerBus 的事件队列中投递事件，所以在代码清单 4-1 中首先创建 LiveListenerBus 和 JobProgressListener，然后将 JobProgressListener 添加到 LiveListenerBus 的监听器列表中，最后将 LiveListenerBus 通过 SparkEnv 的构造器传递给 SparkEnv 及 SparkEnv 内部的组件。JobProgressListener 继承自 SparkListener，LiveListenerBus 和 SparkListener 的详细内容已在 3.3 节介绍，此处不再赘述。

createDriverEnv 方法用于创建 SparkEnv，根据 createDriverEnv 这个方法名，我们知道此方法将为 Driver 实例创建 SparkEnv。调用 createSparkEnv 方法创建完 SparkEnv 后，SparkEnv 实例的引用将通过 SparkEnv 的 set 方法设置到 SparkEnv 伴生对象^Θ的 env 属性中，类似于设置为 Java 类的静态属性，这将便于在任何需要 SparkEnv 的地方，通过伴生对象的 get 方法获取 SparkEnv。

createSparkEnv 方法创建 SparkEnv 的代码如下。

```
private[spark] def createSparkEnv(
    conf: SparkConf,
    isLocal: Boolean,
    listenerBus: LiveListenerBus): SparkEnv = {
    SparkEnv.createDriverEnv(conf, isLocal, listenerBus, SparkContext.
        numDriverCores(master))
}
```

可以看到实际调用了 SparkEnv 的 createDriverEnv 方法来创建 SparkEnv。SparkEnv 的 createDriverEnv 方法的实现如下。

```
private[spark] def createDriverEnv(
    conf: SparkConf,
    isLocal: Boolean,
    listenerBus: LiveListenerBus,
    numCores: Int,
    mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None):
    SparkEnv = {
    assert(conf.contains(DRIVER_HOST_ADDRESS),
        s"${DRIVER_HOST_ADDRESS.key} is not set on the driver!")
```

^Θ 伴生对象是 Scala 语言的语法特性。Scala 中的伴生对象可以提供类似于 Java 中的静态方法、静态属性等的定义，也可以支持类似于 Java 中的 main 方法。更多内容请读者阅读 Scala 语言相关资料。

```

    assert(conf.contains("spark.driver.port"), "spark.driver.port is not set on the
driver!")
    val bindAddress = conf.get(DRIVER_BIND_ADDRESS)
    val advertiseAddress = conf.get(DRIVER_HOST_ADDRESS)
    val port = conf.get("spark.driver.port").toInt
    val ioEncryptionKey = if (conf.get(IO_ENCRYPTION_ENABLED)) {
      Some(CryptoStreamUtils.createKey(conf))
    } else {
      None
    }
    create(
      conf,
      SparkContext.DRIVER_IDENTIFIER,
      bindAddress,
      advertiseAddress,
      port,
      isLocal,
      numCores,
      ioEncryptionKey,
      listenerBus = listenerBus,
      mockOutputCommitCoordinator = mockOutputCommitCoordinator
    )
  }
}

```

createDriverEnv 方法首先从 SparkConf 中获取 4 个属性。

- bindAddress : Driver 实例的 host。此属性通过从 SparkConf 中获取 DRIVER_BIND_ADDRESS 指定的属性值。DRIVER_BIND_ADDRESS 的定义如下。

```

private[spark] val DRIVER_HOST_ADDRESS = ConfigBuilder("spark.driver.host")
  .doc("Address of driver endpoints.")
  .stringConf
  .createWithDefault(Utils.localHostName())

private[spark] val DRIVER_BIND_ADDRESS = ConfigBuilder("spark.driver.bind
  Address")
  .doc("Address where to bind network listen sockets on the driver.")
  .fallbackConf(DRIVER_HOST_ADDRESS)

```

根据 DRIVER_BIND_ADDRESS 的定义，说明按照优先级从高到低，可以通过 spark.driver.bindAddress 属性、spark.driver.host 属性及调用 Utils 的 localHostName 方法获得 bindAddress。Utils 的 localHostName 方法的实现请参阅附录 A。

- advertiseAddress : Driver 实例对外宣称的 host。可以通过 spark.driver.host 属性或者 Utils 的 localHostName 方法获得。
- port : Driver 实例的端口，可以通过 spark.driver.port 属性指定。
- ioEncryptionKey : I/O 加密的密钥。当 spark.io.encryption.enabled 属性为 true 时，调用 CryptoStreamUtils 的 createKey 方法创建密钥。

真正创建 SparkEnv 的实现都在 create 方法中，由于 SparkEnv 是 Driver 和 Executor 实例中都有的组件，本书将在第 5 章对 SparkEnv 作详细介绍。

4.3 SparkUI 的实现

任何系统都需要提供监控功能，否则在运行期间发生一些异常时，我们将会束手无策。也许有人说，可以增加日志来解决这个问题。日志只能解决你的程序逻辑在运行期的监控，进而发现 Bug，以及提供对业务有帮助的调试信息，当你的 JVM 进程崩溃或者程序响应速度很慢时，这些日志将毫无用处。好在 JVM 提供了 jstat、jstack、jinfo、jmap、jhat 等工具帮助我们分析，更有 VisualVM 的可视化界面以更加直观的方式对 JVM 运行期的状况进行监控。此外，像 Tomcat、Hadoop 等服务都提供了基于 Web 的监控页面，用浏览器能访问具有样式及布局，并提供丰富监控数据的页面无疑是一种简单、高效的方式。

Spark 自然也提供了 Web 页面来浏览监控数据，而且 Master、Worker、Driver 根据自身功能提供了不同内容的 Web 监控页面。无论是 Master、Worker，还是 Driver，它们都使用了统一的 Web 框架 WebUI。Master、Worker 及 Driver 分别使用 MasterWebUI、WorkerWebUI 及 SparkUI 提供的 Web 界面服务，后三者都继承自 WebUI，并增加了个性化功能。此外，在 YARN 或 Mesos 模式下还有 WebUI 的另一个扩展实现 HistoryServer。HistoryServer 将会展现已经运行完成的应用程序信息。本章以 SparkUI 为例，并深入分析 WebUI 的框架体系。

4.3.1 SparkUI 概述

在大型分布式系统中，采用事件监听机制是最常见的。为什么要使用事件监听机制？假如 SparkUI 采用 Scala 的函数调用方式，那么随着整个集群规模的增加，对函数的调用会越来越多，最终会受到 Driver 所在 JVM 的线程数量限制而影响监控数据的更新，甚至出现监控数据无法及时显示给用户的情况。由于函数调用多数情况下是同步调用，这就导致线程被阻塞，在分布式环境中，还可能因为网络问题，导致线程被长时间占用。将函数调用更换为发送事件，事件的处理是异步的，当前线程可以继续执行后续逻辑进而被快速释放。线程池中的线程还可以被重用，这样整个系统的并发度会大大增加。发送的事件会存入缓存，由定时调度器取出后，分配给监听此事件的监听器对监控数据进行更新。SparkUI 就是这样的服务，它的构成如图 4-2 所示。

图 4-2 展示了 SparkUI 中的各个组件，这里对这些组件作简单介绍。

1) SparkListenerEvent 事件的来源：包括 DAGScheduler、SparkContext、DriverEndpoint、BlockManagerMasterEndpoint 及 LocalSchedulerBackend 等，这些组件将会产生各种 SparkListenerEvent，并发送到 listenerBus 的事件队列中。DriverEndpoint 是 Driver 在 Standalone 或 local-cluster 模式下与其他组件进行通信的组件，本书将在 9.9.2 节详细介绍。BlockManagerMasterEndpoint 是 Driver 对分配给应用的所有 Executor 及其 BlockManager 进行统一管理的组件，本书将在 6.8 节详细介绍。LocalSchedulerBackend 是 local 模式下的调度后端接口，用于给任务分配资源或对任务的状态进行更新，本书将在 7.8.2 节详细

介绍。

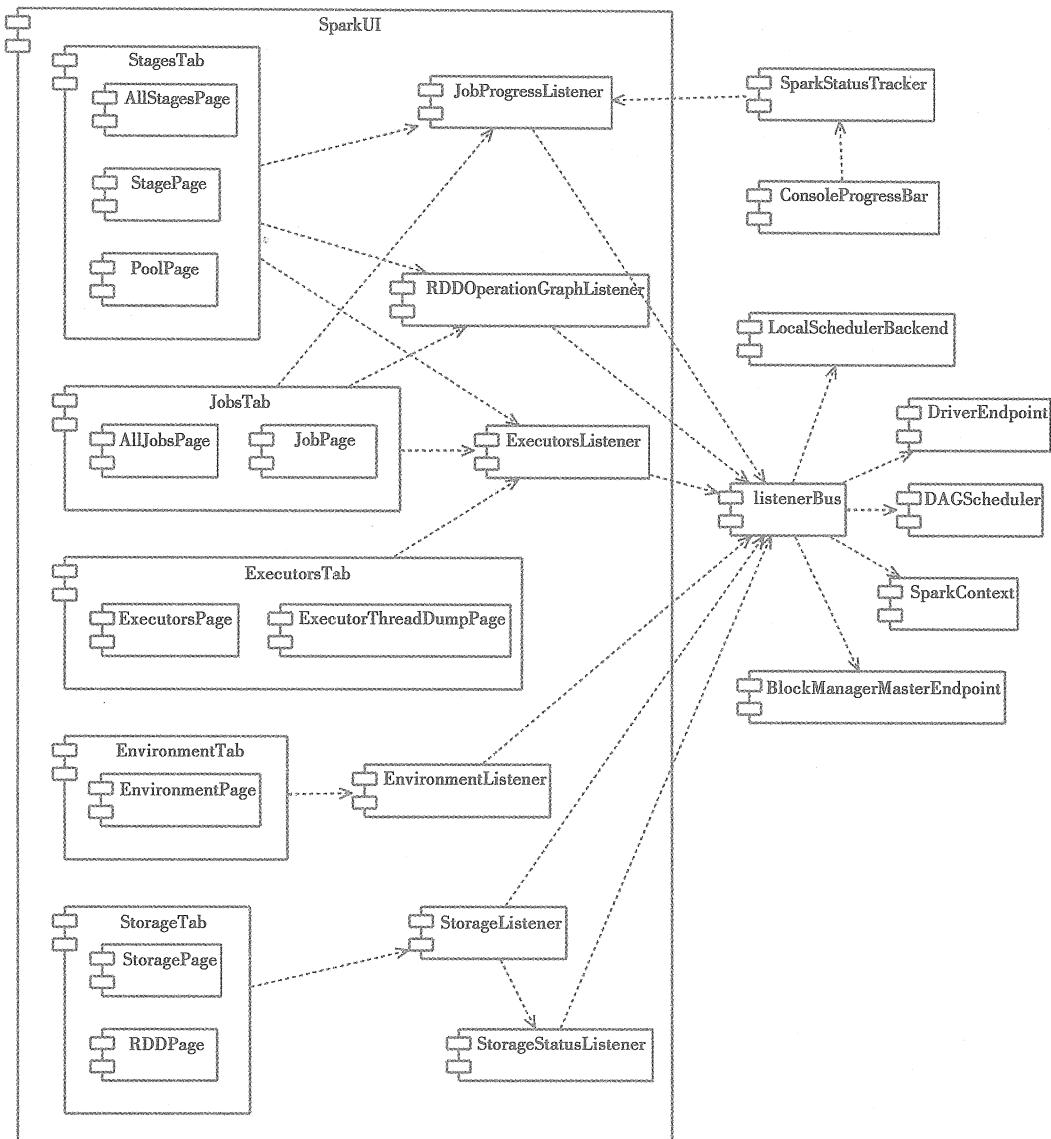


图 4-2 SparkUI 的组成

2) 事件总线 `listenerBus`。根据 3.3 节对事件总线的介绍，我们知道 `listenerBus` 通过定时器将 `SparkListenerEvent` 事件匹配到具体的 `SparkListener`，进而改变各个 `SparkListener` 中的统计监控数据。

3) SparkUI 的界面。各个 `SparkListener` 内的统计监控数据将会被各种标签页和具体页面展示到 Web 界面。标签页有 `StagesTab`、`JobsTab`、`ExecutorsTab`、`EnvironmentTab` 及 `Storage`

Tab。每个标签页中包含若干个页面，例如，StagesTab 标签页中包含了 AllStagesPage、StagePage 及 PoolPage 三个页面。

4) 控制台的展示。细心的读者会发现图 4-2 中还有 SparkStatusTracker (Spark 状态跟踪器) 和 ConsoleProgressBar (控制台进度条) 两个组件。SparkStatusTracker 负责对 Job 和 Stage 的监控，实际也是使用了 JobProgressListener 中的监控数据，并额外进行了一些加工。ConsoleProgressBar 负责将 SparkStatusTracker 提供的数据打印到控制台上。从最终展现的角度来看，SparkStatusTracker 和 ConsoleProgressBar 不应该属于 SparkUI 的组成部分，但是由于其实现与 JobProgressListener 密切相关，所以将它们也放在了 SparkUI 的内容中。

4.3.2 WebUI 框架体系

SparkUI 构建在 WebUI 的框架体系之上，因此应当首先了解 WebUI。WebUI 定义了一种 Web 界面展现的框架，并提供返回 Json 格式数据的 Web 服务。WebUI 用于展示一组标签页，WebUITab 定义了标签页的规范。每个标签页中包含着一组页面，WebUIPage 定义了页面的规范。我们将首先了解 WebUIPage 和 WebUITab，最后从整体来看 WebUI。

1. WebUIPage 的定义

任何 Web 界面往往由多个页面组成，每个页面都将提供不同的内容展示。WebUIPage 是 WebUI 框架体系的页节点，定义了所有页面应当遵循的规范。抽象类 WebUIPage 的定义如代码清单 4-2 所示。

代码清单4-2 WebUIPage的定义

```
private[spark] abstract class WebUIPage(var prefix: String) {
    def render(request: HttpServletRequest): Seq[Node]
    def renderJson(request: HttpServletRequest): JValue = JNothing
}
```

WebUIPage 定义了两个方法。

- render: 渲染页面。
- renderJson: 生成 JSON。

WebUIPage 在 WebUI 框架体系中的上一级节点（也可以称为父亲）可以是 WebUI 或者 WebUITab，其成员属性 prefix 将与上级节点的路径一起构成当前 WebUIPage 的访问路径。

2. WebUITab 的定义

有时候 Web 界面需要将多个页面作为一组内容放置在一起，这时候标签页是常见的展现形式。标签页 WebUITab 定义了所有标签页的规范，并用于展现一组 WebUIPage。抽象类 WebUITab 的定义如代码清单 4-3 所示。

代码清单4-3 WebUITab的定义

```
private[spark] abstract class WebUITab(parent: WebUI, val prefix: String) {
```

```

val pages = ArrayBuffer[WebUIPage]()
val name = prefix.capitalize
def attachPage(page: WebUIPage) {
    page.prefix = (prefix + "/" + page.prefix).stripSuffix("/")
    pages += page
}
def headerTabs: Seq[WebUITab] = parent.getTabs

def basePath: String = parent.getbasePath
}

```

根据代码清单 4-3，可以看到 WebUITab 有 4 个成员属性。

- parent：上一级节点，即父亲。WebUITab 的父亲只能是 WebUI。
- prefix：当前 WebUITab 的前缀。prefix 将与上级节点的路径一起构成当前 WebUITab 的访问路径。
- pages：当前 WebUITab 所包含的 WebUIPage 的缓冲数组。
- name：当前 WebUITab 的名称。name 实际是将 prefix 的首字母转换成大写字母后取得。

此外，WebUITab 还有 3 个成员方法，下面介绍它们的作用。

- attachPage：首先将当前 WebUITab 的前缀与 WebUIPage 的前缀拼接，作为 WebUIPage 的访问路径，然后向 pages 中添加 WebUIPage。
- headerTabs：获取父亲 WebUI 中的所有 WebUITab。此方法实际通过调用父亲 WebUI 的 getTabs 方法实现，getTabs 方法请参阅下面“WebUI 的定义”部分。
- basePath：获取父亲 WebUI 的基本路径。此方法实际通过调用父亲 WebUI 的 getbasePath 方法实现，getbasePath 方法请参阅下面“WebUI 的定义”部分。

3. WebUI 的定义

WebUI 是 Spark 实现的用于提供 Web 界面展现的框架，凡是需要页面展现的地方都可以继承它来完成。WebUI 定义了 WebUI 框架体系的规范。为便于理解，首先明确 WebUI 中各个成员属性的含义。

- securityManager：SparkEnv 中创建的安全管理器 SecurityManager，5.2 节对 Security Manager 有详细介绍。
- sslOptions：使用 SecurityManager 获取 spark.ssl.ui 属性指定的 WebUI 的 SSL(Secure Sockets Layer，安全套接层) 选项。
- port：WebUI 对外服务的端口，可以使用 spark.ui.port 属性进行配置。
- conf：即 SparkConf。
- basePath：WebUI 的基本路径，basePath 默认为空字符串。
- name：WebUI 的名称，Spark UI 的 name 为 SparkUI。
- tabs：WebUITab 的缓冲数组。

- handlers：ServletContextHandler 的缓冲数组。ServletContextHandler 是 Jetty 提供的 API，负责对 ServletContext 进行处理。ServletContextHandler 的使用及 Jetty 的更多内容可以参阅附录 C。
- pageToHandlers：WebUIPage 与 ServletContextHandler 缓冲数组之间的映射关系。由于 WebUIPage 的两个方法 render 和 renderJson 分别需要由一个对应的 ServletContextHandler 处理，所以一个 WebUIPage 对应两个 ServletContextHandler。
- serverInfo：用于缓存 ServerInfo，即 WebUI 的 Jetty 服务器信息。
- publicHostName：当前 WebUI 的 Jetty 服务的主机名。优先采用系统环境变量 SPARK_PUBLIC_DNS 指定的主机名，否则采用 spark.driver.host 属性指定的 host，在没有前两个配置的时候，将默认使用工具类 Utils 的 localHostName 方法（详见附录 A）返回的主机名。
- className：过滤了 \$ 符号的当前类的简单名称。className 是通过 Utils 的 getFormattedClassName 方法得到的。getFormattedClassName 方法的实现请参看附录 A。

了解了 WebUI 的成员属性，现在就可以理解其提供的各个方法了。WebUI 提供的方法有以下几种。

- getBasePath：获取 basePath。
- getTabs：获取 tabs 中的所有 WebUITab，并以 Scala 的序列返回。
- getHandlers：获取 handlers 中的所有 ServletContextHandler，并以 Scala 的序列返回。
- getSecurityManager：获取 securityManager。
- attachHandler：给 handlers 缓存数组中添加 ServletContextHandler，并且将此 ServletContextHandler 通过 ServerInfo 的 addHandler 方法添加到 Jetty 服务器中。attachHandler 的实现如代码清单 4-4 所示。ServerInfo 的 addHandler 方法请参阅附录 C。

代码清单4-4 attachHandler的实现

```
def attachHandler(handler: ServletContextHandler) {
    handlers += handler
    serverInfo.foreach(_.addHandler(handler))
}
```

- detachHandler：从 handlers 缓存数组中移除 ServletContextHandler，并且将此 ServletContextHandler 通过 ServerInfo 的 removeHandler 方法从 Jetty 服务器中移除。detachHandler 的实现如代码清单 4-5 所示。ServerInfo 的 removeHandler 方法请参阅附录 C。

代码清单4-5 detachHandler的实现

```
def detachHandler(handler: ServletContextHandler) {
```

```

    handlers -= handler
    serverInfo.foreach(_.removeHandler(handler))
}

```

- attachPage：首先调用工具类 JettyUtils 的 createServletHandler 方法给 WebUIPage 创建与 render 和 renderJson 两个方法分别关联的 ServletContextHandler，然后通过 attachHandler 方法添加到 handlers 缓存数组与 Jetty 服务器中，最后把 WebUIPage 与这两个 ServletContextHandler 的映射关系更新到 pageToHandlers 中。attachPage 的实现如代码清单 4-6 所示。

代码清单4-6 attachPage的实现

```

def attachPage(page: WebUIPage) {
    val pagePath = "/" + page.prefix
    val renderHandler = createServletHandler(pagePath,
        (request: HttpServletRequest) => page.render(request), securityManager, conf,
        basePath)
    val renderJsonHandler = createServletHandler(pagePath.stripSuffix("/") + "/json",
        (request: HttpServletRequest) => page.renderJson(request), securityManager,
        conf, basePath)
    attachHandler(renderHandler)
    attachHandler(renderJsonHandler)
    val handlers = pageToHandlers.getOrElseUpdate(page, ArrayBuffer[ServletContext
        Handler]())
    handlers += renderHandler
}

```

- detachPage：作用与 attachPage 相反。detachPage 的实现如代码清单 4-7 所示。

代码清单4-7 detachPage的实现

```

def detachPage(page: WebUIPage) {
    pageToHandlers.remove(page).foreach(_.foreach(detachHandler))
}

```

- attachTab：首先向 tabs 中添加 WebUITab，然后给 WebUITab 中的每个 WebUIPage 施加 attachPage 方法。attachTab 的实现如代码清单 4-8 所示。

代码清单4-8 attachTab的实现

```

def attachTab(tab: WebUITab) {
    tab.pages.foreach(attachPage)
    tabs += tab
}

```

- detachTab：作用与 attachTab 相反。detachTab 的实现如代码清单 4-9 所示。

代码清单4-9 detachTab的实现

```

def detachTab(tab: WebUITab) {
    tab.pages.foreach(detachPage)
}

```

```
    tabs -= tab
}
```

- `addStaticHandler`：首先调用工具类 `JettyUtils` 的 `createStaticHandler` 方法创建静态文件服务的 `ServletContextHandler`，然后施加 `attachHandler` 方法。`addStaticHandler` 的实现如代码清单 4-10 所示。`JettyUtils` 的 `createStaticHandler` 方法的实现见附录 C。

代码清单4-10 `addStaticHandler`的实现

```
def addStaticHandler(resourceBase: String, path: String): Unit = {
  attachHandler(JettyUtils.createStaticHandler(resourceBase, path))
}
```

- `removeStaticHandler`：作用与 `addStaticHandler` 相反。`removeStaticHandler` 的实现如代码清单 4-11 所示。

代码清单4-11 `removeStaticHandler`的实现

```
def removeStaticHandler(path: String): Unit = {
  handlers.find(_.getContextPath() == path).foreach(detachHandler)
}
```

- `initialize`：用于初始化 WebUI 服务中的所有组件。WebUI 中此方法未实现，需要子类实现。
- `bind`：启动与 WebUI 绑定的 Jetty 服务。`bind` 方法的实现如代码清单 4-12 所示。

代码清单4-12 `bind`的实现

```
def bind() {
  assert(!serverInfo.isDefined, s"Attempted to bind $className more than once!")
  try {
    val host = Option(conf.getenv("SPARK_LOCAL_IP")).getOrElse("0.0.0.0")
    serverInfo = Some(startJettyServer(host, port, sslOptions, handlers, conf,
      name))
    logInfo(s"Bound $className to $host, and started at $webUrl")
  } catch {
    case e: Exception =>
      logError(s"Failed to bind $className", e)
      System.exit(1)
  }
}
```

- `webUrl`：获取 WebUI 的 Web 界面的 URL。`webUrl` 的实现如下。

```
def webUrl: String = shttp://$publicHostName:$boundPort
```

- `boundPort`：获取 WebUI 的 Jetty 服务的端口。`boundPort` 的实现如下。

```
def boundPort: Int = serverInfo.map(_.boundPort).getOrElse(-1)
```

- `stop`：停止 WebUI。实际是停止 WebUI 底层的 Jetty 服务。`stop` 方法的实现如代码清单 4-13 所示。

代码清单4-13 stop方法的实现

```
def stop() {
    assert(serverInfo.isDefined,
           s"Attempted to stop $className before binding to a server!")
    serverInfo.get.stop()
}
```

4.3.3 创建 SparkUI

在 SparkContext 的初始化过程中，会创建 SparkUI。有了对 WebUI 的总体认识，现在是时候了解 SparkContext 是如何构造 SparkUI 的了。SparkUI 是 WebUI 框架的使用范例，了解了 SparkUI 的创建过程，读者对 MasterWebUI、WorkerWebUI 及 HistoryServer 的创建过程也必然了然于心。创建 SparkUI 的代码如下。

```
_statusTracker = new SparkStatusTracker(this)

_progressBar =
  if (_conf.getBoolean("spark.ui.showConsoleProgress", true) && !log.isInfoEnabled) {
    Some(new ConsoleProgressBar(this))
  } else {
    None
  }

_ui =
  if (conf.getBoolean("spark.ui.enabled", true)) {
    Some(SparkUI.createLiveUI(this, _conf, listenerBus, _jobProgressListener,
      _env.securityManager, appName, startTime = startTime))
  } else {
    // For tests, do not enable the UI
    None
  }
_ui.foreach(_.bind())
```

这段代码的执行步骤如下。

- 1) 创建 Spark 状态跟踪器 SparkStatusTracker。
- 2) 创建 ConsoleProgressBar。可以配置 spark.ui.showConsoleProgress 属性为 false，取消对 ConsoleProgressBar 的创建，此属性默认为 true。
- 3) 调用 SparkUI 的 createLiveUI 方法创建 SparkUI。
- 4) 给 SparkUI 绑定端口。SparkUI 继承自 WebUI，因此调用了代码清单 4-12 中 WebUI 的 bind 方法启动 SparkUI 底层的 Jetty 服务。

上述步骤中，第 1)、2)、4) 步都很简单，所以着重来分析第 3) 步。SparkUI 的 createLiveUI 的实现如下。

```
def createLiveUI(
  sc: SparkContext,
  conf: SparkConf,
  listenerBus: SparkListenerBus,
```

```

    jobProgressListener: JobProgressListener,
    securityManager: SecurityManager,
    appName: String,
    startTime: Long): SparkUI = {
  create(Some(sc), conf, listenerBus, securityManager, appName,
    jobProgressListener = Some(jobProgressListener), startTime = startTime)
}

```

可以看到 SparkUI 的 createLiveUI 方法中调用了 create 方法。create 的实现如下。

```

private def create(
  sc: Option[SparkContext],
  conf: SparkConf,
  listenerBus: SparkListenerBus,
  securityManager: SecurityManager,
  appName: String,
  basePath: String = "",
  jobProgressListener: Option[JobProgressListener] = None,
  startTime: Long): SparkUI = {

  val _jobProgressListener: JobProgressListener = jobProgressListener.getOrElse {
    val listener = new JobProgressListener(conf)
    listenerBus.addListener(listener)
    listener
  }

  val environmentListener = new EnvironmentListener
  val storageStatusListener = new StorageStatusListener(conf)
  val executorsListener = new ExecutorsListener(storageStatusListener, conf)
  val storageListener = new StorageListener(storageStatusListener)
  val operationGraphListener = new RDDOperationGraphListener(conf)

  listenerBus.addListener(environmentListener)
  listenerBus.addListener(storageStatusListener)
  listenerBus.addListener(executorsListener)
  listenerBus.addListener(storageListener)
  listenerBus.addListener(operationGraphListener)

  new SparkUI(sc, conf, securityManager, environmentListener, storageStatusListener,
    executorsListener, _jobProgressListener, storageListener, operationGraphListener,
    appName, basePath, startTime)
}

```

可以看到，create 方法里除了 JobProgressListener 是外部传入的之外，又增加了一些 SparkListener，例如，用于对 JVM 参数、Spark 属性、Java 系统属性、classpath 等进行监控的 EnvironmentListener；用于维护 Executor 的存储状态的 StorageStatusListener；用于准备将 Executor 的信息展示在 ExecutorsTab 的 ExecutorsListener；用于准备将 Executor 相关存储信息展示在 BlockManagerUI 的 StorageListener；用于构建 RDD 的 DAG（有向无边图）的 RDDOperationGraphListener 等。根据代码清单 4-1，JobProgressListener 早已被添加到 listenerBus 的监听器列表中，所以此处只需要将另外 5 个 SparkListener 的实现添加到 listenerBus 的监听器列表中。最后使用 SparkUI 的构造器创建 SparkUI。

1. SparkUI 的初始化

调用 SparkUI 的构造器创建 SparkUI，实际也是对 SparkUI 的初始化过程。在介绍初始化之前，先来看看 SparkUI 中的两个成员属性。

❑ killEnabled：标记当前 SparkUI 能否提供“杀死”Stage 或者 Job 的链接。

❑ appId：当前应用的 ID。

SparkUI 的构造过程中会执行 initialize 方法，其实现如代码清单 4-14 所示。

代码清单4-14 SparkUI的初始化

```
def initialize() {
    val jobsTab = new JobsTab(this)
    attachTab(jobsTab)
    val stagesTab = new StagesTab(this)
    attachTab(stagesTab)
    attachTab(new StorageTab(this))
    attachTab(new EnvironmentTab(this))
    attachTab(new ExecutorsTab(this))
    attachHandler(createStaticHandler(SparkUI.STATIC_RESOURCE_DIR, "/static"))
    attachHandler(createRedirectHandler("/", "/jobs/", basePath = basePath))
    attachHandler(ApiRootResource.getServletHandler(this))
    // These should be POST only, but, the YARN AM proxy won't proxy POSTs
    attachHandler(createRedirectHandler(
        "/jobs/job/kill", "/jobs/", jobsTab.handleKillRequest, httpMethods = Set("GET",
        "POST")))
    attachHandler(createRedirectHandler(
        "/stages/stage/kill", "/stages/", stagesTab.handleKillRequest,
        httpMethods = Set("GET", "POST")))
}
initialize()
```

根据代码清单 4-14，SparkUI 的初始化步骤如下。

1) 构建页面布局并给每个 WebUITab 中的所有 WebUIPage 创建对应的 ServletContextHandler。这一步使用了代码清单 4-8 中展示的 attachTab 方法。

2) 调用 JettyUtils 的 createStaticHandler 方法创建对静态目录 org/apache/spark/ui/static 提供文件服务的 ServletContextHandler，并使用 attachHandler 方法追加到 SparkUI 的服务中。

3) 调用 JettyUtils 的 createRedirectHandler 方法创建几个将用户对源路径的请求重定向到目标路径的 ServletContextHandler。例如，将用户对根路径“/”的请求重定向到目标路径“/jobs/”的 ServletContextHandler。

2. SparkUI 的页面布局与展示

SparkUI 究竟是如何实现页面布局及展示的？由于所有标签页都继承了 SparkUITab，所以我们先来看看 SparkUITab 的实现。

```
private[spark] abstract class SparkUITab(parent: SparkUI, prefix: String)
  extends WebUITab(parent, prefix) {
```

```
    def appName: String = parent.getAppName
}
```

根据上述代码，我们知道 SparkUITab 继承了 WebUITab，并在实现中增加了一个用于获取当前应用名称的方法 appName。EnvironmentTab 是用于展示 JVM、Spark 属性、系统属性、类路径等相关信息的标签页，由于其实现简单且能说明问题，所以本节挑选 EnvironmentTab 作为示例解答本节一开始提出的问题。

EnvironmentTab 的实现如代码清单 4-15 所示。

代码清单4-15 EnvironmentTab的实现

```
private[ui] class EnvironmentTab(parent: SparkUI) extends SparkUITab(parent,
  "environment") {
  val listener = parent.environmentListener
  attachPage(new EnvironmentPage(this))
}
```

根据代码清单 4-15，我们知道 EnvironmentTab 引用了 SparkUI 的 environmentListener（类型为 EnvironmentListener），并且包含 EnvironmentPage 这个页面。EnvironmentTab 通过调用 attachPage 方法将 EnvironmentPage 与 Jetty 服务关联起来。根据代码清单 4-6 中 attachPage 的实现，创建的 renderHandler 将采用偏函数 (request: HttpServletRequest) => page.render(request) 处理请求，因而会调用 EnvironmentPage 的 render 方法。EnvironmentPage 的 render 方法将会渲染页面元素。EnvironmentPage 的实现如代码清单 4-16 所示。

代码清单4-16 EnvironmentPage的实现

```
private[ui] class EnvironmentPage(parent: EnvironmentTab) extends WebUIPage("") {
  private val listener = parent.listener

  private def removePass(kv: (String, String)): (String, String) = {
    if (kv._1.toLowerCase.contains("password") || kv._1.toLowerCase.contains("secret")) {
      (kv._1, "*****")
    } else kv
  }

  def render(request: HttpServletRequest): Seq[Node] = {
    // 调用UIUtils的listingTable方法生成JVM运行时信息、Spark属性信息、系统属性信息、类路径信息的表格
    val runtimeInformationTable = UIUtils.listingTable(
      propertyHeader, jvmRow, listener.jvmInformation, fixedWidth = true)
    val sparkPropertiesTable = UIUtils.listingTable(
      propertyHeader, propertyRow, listener.sparkProperties.map(removePass),
      fixedWidth = true)
    val systemPropertiesTable = UIUtils.listingTable(
      propertyHeader, propertyRow, listener.systemProperties, fixedWidth = true)
    val classpathEntriesTable = UIUtils.listingTable(
      classPathHeaders, classPathRow, listener.classpathEntries, fixedWidth = true)
    val content =
      <span>
        <h4>Runtime Information</h4> {runtimeInformationTable}
      </span>
  }
}
```

```

<h4>Spark Properties</h4> {sparkPropertiesTable}
<h4>System Properties</h4> {systemPropertiesTable}
<h4>Classpath Entries</h4> {classpathEntriesTable}
</span>
// 调用UIUtils的headerSparkPage方法封装好css、js、header及页面布局等
UIUtils.headerSparkPage("Environment", content, parent)
}
// 定义JVM运行时信息、Spark属性信息、系统属性信息的表格头部propertyHeader和类路径信息的表格头部
// classPathHeaders
private def propertyHeader = Seq("Name", "Value")
private def classPathHeaders = Seq("Resource", "Source")
// 定义JVM运行时信息的表格中每行数据的生成方法jvmRow
private def jvmRow(kv: (String, String)) = <tr><td>{kv._1}</td><td>{kv._2}</td></tr>
private def propertyRow(kv: (String, String)) = <tr><td>{kv._1}</td><td>{kv._2}</td></tr>
private def classPathRow(data: (String, String)) = <tr><td>{data._1}</td><td>{data._2}</td></tr>
}

```

根据代码清单 4-16，EnvironmentPage 的 render 方法利用从父节点 EnvironmentTab 中得到的 EnvironmentListener 中的统计监控数据生成 JVM 运行时、Spark 属性、系统属性及类路径等状态的摘要信息。以 JVM 运行时为例，页面渲染的步骤如下。

- 1) 定义 JVM 运行时信息、Spark 属性信息、系统属性信息的表格头部 propertyHeader 和类路径信息的表格头部 classPathHeaders。
 - 2) 定义 JVM 运行时信息的表格中每行数据的生成方法 jvmRow。
 - 3) 调用 UIUtils 的 listingTable 方法生成 JVM 运行时信息、Spark 属性信息、系统属性信息、类路径信息的表格。
 - 4) 调用 UIUtils 的 headerSparkPage 方法封装好 CSS、JS、header 及页面布局等。
- UIUtils 工具类的实现细节留给感兴趣的读者自行查阅，本文不再赘述。

4.4 创建心跳接收器

根据前文所述，生产环境的 Executor 运行于不同节点上。在 local 模式下 Driver 与 Executor 属于同一个进程，所以 Driver 与 Executor 可以直接使用本地调用交互，当 Executor 运行出现问题时，Driver 可以很方便地知道，例如，通过捕获异常。但在生产环境下，Driver 与 Executor 很可能不在同一个进程中，它们也许运行在不同的机器上，甚至在不同的机房里，因此 Driver 对 Executor 失去了掌控。为了能够掌控 Executor，在 Driver 中创建了这个心跳接收器。为了弄清楚心跳接收器，我们需要从心跳接收器的创建开始。

SparkContext 中创建 HeartbeatReceiver 的代码如下。

```

_heartbeatReceiver = env.rpcEnv.setupEndpoint(
HeartbeatReceiver.ENDPOINT_NAME, new HeartbeatReceiver(this))

```

上面的代码中使用了 SparkEnv 的子组件 NettyRpcEnv 的 setupEndpoint 方法，此方法的作用是向 RpcEnv 的 Dispatcher 注册 HeartbeatReceiver，并返回 HeartbeatReceiver 的 NettyRpcEndpointRef 引用。有关 RpcEnv 及 NettyRpcEndpointRef 的内容将在 5.3 节详细介绍，而 HeartbeatReceiver 将在第 9 章详细介绍。

4.5 创建和启动调度系统

TaskScheduler 是 SparkContext 的重要组成部分，负责请求集群管理器给应用程序分配并运行 Executor（一级调度）和给任务分配 Executor 并运行任务（二级调度）。Task Scheduler 也可以看做任务调度的客户端。DAGScheduler 主要用于在任务正式交给 TaskSchedulerImpl 提交之前做一些准备工作，包括创建 Job、将 DAG 中的 RDD 划分到不同的 Stage、提交 Stage 等。代码清单 4-17 的代码用于在 SparkContext 中创建 Task Scheduler 和 DAGScheduler。

代码清单4-17 创建TaskScheduler和DAGScheduler

```
// Create and start the scheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler(this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)
```

根据代码清单 4-17，SparkContext 的 createTaskScheduler 方法用于创建任务调度器，此方法将返回 SchedulerBackend 和 TaskScheduler 的对偶，并由 _schedulerBackend 持有 SchedulerBackend 的引用，由 _taskScheduler 持有 TaskScheduler 的引用。代码清单 4-17 的最后还向 HeartbeatReceiver 发送 TaskSchedulerIsSet 消息，表示 SparkContext 的 _taskScheduler 属性已经持有了 TaskScheduler 的引用，HeartbeatReceiver 接收到 TaskSchedulerIsSet 消息后，将获取 SparkContext 的 _taskScheduler 属性设置到自身的 scheduler 属性中（将在第 9 章中详细介绍）。

createTaskScheduler 的实现如代码清单 4-18 所示。

代码清单4-18 创建任务调度器

```
private def createTaskScheduler(
  sc: SparkContext,
  master: String,
  deployMode: String): (SchedulerBackend, TaskScheduler) = {
  import SparkMasterRegex._

  // When running locally, don't try to re-execute tasks on failure.
  val MAX_LOCAL_TASK_FAILURES = 1

  master match {
```

```

case "local" =>
  val scheduler = new TaskSchedulerImpl(sc, MAX_LOCAL_TASK_FAILURES, isLocal = true)
  val backend = new LocalSchedulerBackend(sc.getConf, scheduler, 1)
  scheduler.initialize(backend)
  (backend, scheduler)

case LOCAL_N_REGEX(threads) =>
  def localCpuCount: Int = Runtime.getRuntime.availableProcessors()
  // local[*] estimates the number of cores on the machine; local[N] uses
  // exactly N threads.
  val threadCount = if (threads == "*") localCpuCount else threads.toInt
  if (threadCount <= 0) {
    throw new SparkException(s"Asked to run locally with $threadCount threads")
  }
  val scheduler = new TaskSchedulerImpl(sc, MAX_LOCAL_TASK_FAILURES, isLocal = true)
  val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)
  scheduler.initialize(backend)
  (backend, scheduler)

case LOCAL_N_FAILURES_REGEX(threads, maxFailures) =>
  def localCpuCount: Int = Runtime.getRuntime.availableProcessors()
  // local[*, M] means the number of cores on the computer with M failures
  // local[N, M] means exactly N threads with M failures
  val threadCount = if (threads == "*") localCpuCount else threads.toInt
  val scheduler = new TaskSchedulerImpl(sc, maxFailures.toInt, isLocal = true)
  val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)
  scheduler.initialize(backend)
  (backend, scheduler)

case SPARK_REGEX(sparkUrl) =>
  val scheduler = new TaskSchedulerImpl(sc)
  val masterUrls = sparkUrl.split(",").map("spark://" + _)
  val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)
  scheduler.initialize(backend)
  (backend, scheduler)

case LOCAL_CLUSTER_REGEX(numSlaves, coresPerSlave, memoryPerSlave) =>
  // Check to make sure memory requested <= memoryPerSlave. Otherwise Spark
  // will just hang.
  val memoryPerSlaveInt = memoryPerSlave.toInt
  if (sc.executorMemory > memoryPerSlaveInt) {
    throw new SparkException(
      "Asked to launch cluster with %d MB RAM / worker but requested %d MB/
      worker".format(
        memoryPerSlaveInt, sc.executorMemory))
  }

  val scheduler = new TaskSchedulerImpl(sc)
  val localCluster = new LocalSparkCluster(
    numSlaves.toInt, coresPerSlave.toInt, memoryPerSlaveInt, sc.conf)
  val masterUrls = localCluster.start()
  val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)
  scheduler.initialize(backend)
  backend.shutdownCallback = (backend: StandaloneSchedulerBackend) => {
    localCluster.stop()
  }
}

```

```

        (backend, scheduler)

    case masterUrl =>
        val cm = getClusterManager(masterUrl) match {
            case Some(clusterMgr) => clusterMgr
            case None => throw new SparkException("Could not parse Master URL: '" +
                master + "'")
        }
        try {
            val scheduler = cm.createTaskScheduler(sc, masterUrl)
            val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)
            cm.initialize(scheduler, backend)
            (backend, scheduler)
        } catch {
            case se: SparkException => throw se
            case NonFatal(e) =>
                throw new SparkException("External scheduler cannot be instantiated", e)
        }
    }
}

```

根据代码清单 4-18，我们看到针对不同的部署模式（即 `deployMode`），创建任务调度器的方式也各有不同。将这一大段代码放置此处显得非常拖沓，但由于本书的内容将多次涉及此段代码，所以还是决定在此处展示。有关任务调度的详细内容将放在第 7 章详细介绍。

`SparkContext` 创建完 `TaskScheduler` 后，将启动 `TaskScheduler`，如代码清单 4-19 所示。

代码清单4-19 启动调度系统

```
_taskScheduler.start()
```

`TaskScheduler` 在启动的时候还会启动 `DAGScheduler`，这些内容也将在第 7 章详细介绍。

4.6 初始化块管理器 BlockManager

`BlockManager` 是 `SparkEnv` 中的组件之一，从命名来看应该是 `Block` 的管理器，实际上囊括了存储体系的所有组件和功能，是存储体系中最重要的组件。`SparkContext` 初始化的过程中会对 `BlockManager` 进行初始化，代码如下。

```
_env.blockManager.initialize(_applicationId)
```

`_applicationId` 是应用程序向 `Master` 注册时，`Master` 为其创建的应用标识。`BlockManager` 的初始化将在 6.7 节详细介绍。

4.7 启动度量系统

3.4 节曾介绍过度量系统中的 `Source` 及 `Sink`，但是并未将整个度量系统贯穿起来。`Metrics`

System 对 Source 和 Sink 进行封装，将 Source 的数据输出到不同的 Sink。Metrics-System 是 SparkEnv 内部的组件之一，是整个 Spark 应用程序的度量系统。启动度量系统的代码如下。

```
_env.metricsSystem.start()
_env.metricsSystem.getServletHandlers.foreach(handler => ui.foreach(_.attachHandler(handler)))
```

上述代码还调用 WebUI 的 attachHandler 方法（见代码清单 4-4）将度量系统的 Servlet ContextHandler 添加到了 Spark UI 中。有关启动度量系统和 MetricsSystem 的 getServlet Handlers 方法将在 5.8 节详细介绍。

4.8 创建事件日志监听器

EventLoggingListener 是将事件持久化到存储的监听器，是 SparkContext 中的可选组件。当 spark.eventLog.enabled 属性为 true 时启用。这里来看看 SparkContext 是如何创建 EventLoggingListener 的。

```
_eventLogger =
if (isEventLogEnabled) {
    val logger =
        new EventLoggingListener(_applicationId, _applicationAttemptId, _
            eventLogDir.get,
            _conf, _hadoopConfiguration)
    logger.start()
    listenerBus.addListener(logger)
    Some(logger)
} else {
    None
}
```

EventLoggingListener 也将参与到对事件总线中事件的监听中，并把感兴趣的事件记录到日志。

以 EventLoggingListener 感兴趣的 SparkListenerBlockManagerRemoved 事件为例，EventLoggingListener 重写的 onBlockManagerRemoved 方法将对 SparkListenerBlockManagerRemoved 事件进行处理，代码如下。

```
override def onBlockManagerRemoved(event: SparkListenerBlockManagerRemoved): Unit
= {
    logEvent(event, flushLogger = true)
}
```

EventLoggingListener 最为核心的方法是 logEvent，其实现如下。

```
private def logEvent(event: SparkListenerEvent, flushLogger: Boolean = false) {
    val eventJson = JsonProtocol.sparkEventToJson(event)
    writer.foreach(_.println(compact(render(eventJson))))
    if (flushLogger) {
```

```

        writer.foreach(_.flush())
        hadoopDataStream.foreach(_.hflush())
    }
    if (testing) {
        loggedEvents += eventJson
    }
}
}

```

logEvent 用于将事件转换为 Json 字符串后写入日志文件。

4.9 创建和启动 ExecutorAllocationManager

ExecutorAllocationManager 的作用已在 4.1 节有过介绍，更为准确地说，ExecutorAllocationManager 是基于工作负载动态分配和删除 Executor 的代理。简单来讲，ExecutorAllocationManager 与集群管理器之间的关系可以用图 4-3 来表示。

ExecutorAllocationManager 内部会定时根据工作负载计算所需的 Executor 数量，如果对 Executor 需求数量大于之前向集群管理器申请的 Executor 数量，那么向集群管理器申请添加 Executor；如果对 Executor 需求数量小于之前向集群管理器申请的 Executor 数量，那么向集群管理器申请取消部分 Executor。此外，ExecutorAllocationManager 内部还会定时向集群管理器申请移除（“杀死”）过期的 Executor。

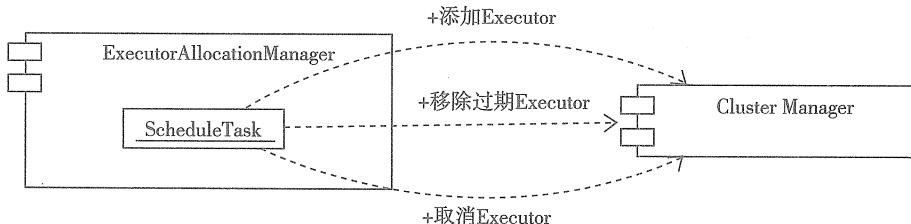


图 4-3 ExecutorAllocationManager 与集群管理器之间的关系

有了对 ExecutorAllocationManager 的了解，这里来看看 SparkContext 是如何创建和启动 ExecutorAllocationManager 的。创建 ExecutorAllocationManager 的代码如下。

```

val dynamicAllocationEnabled = Utils.isDynamicAllocationEnabled(_conf)
_executorAllocationManager =
  if (dynamicAllocationEnabled) {
    schedulerBackend match {
      case b: ExecutorAllocationClient =>
        Some(new ExecutorAllocationManager(
          schedulerBackend.asInstanceOf[ExecutorAllocationClient], listenerBus, _conf))
      case _ =>
        None
    }
  } else {
    None
  }
_executorAllocationManager.foreach(_.start())

```

根据上述代码，创建 ExecutorAllocationManager 的步骤如下。

1) 调用 Utils 工具类的 isDynamicAllocationEnabled 方法判断是否需要启用 ExecutorAllocationManager。根据附录 A 中对 isDynamicAllocationEnabled 方法的介绍，我们知道在配置 spark.dynamicAllocation.enabled 属性为 true 的前提下，在非 Local 模式下或者当 spark.dynamicAllocation.testing 属性为 true 时启用 ExecutorAllocationManager。

2) 在 SchedulerBackend 的实现类同时实现了特质 ExecutorAllocationClient 的情况下，才会创建 ExecutorAllocationManager。

3) 调用 ExecutorAllocationManager 的 start 方法启动 ExecutorAllocationManager。

ExecutorAllocationManager 的 start 方法的实现如下。

```
def start(): Unit = {
    listenerBus.addListener(listener)
    val scheduleTask = new Runnable() {
        override def run(): Unit = {
            try {
                schedule()
            } catch {
                case ct: ControlThrowable =>
                    throw ct
                case t: Throwable =>
                    logWarning(s"Uncaught exception in thread ${Thread.currentThread().getName}", t)
            }
        }
    }
    executor.scheduleWithFixedDelay(scheduleTask, 0, intervalMillis, TimeUnit.MILLISECONDS)
    client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks, hostToLocalTaskCount)
}
```

根据上述代码，ExecutorAllocationManager 的 start 方法的执行步骤如下。

- 1) 向事件总线添加 ExecutorAllocationListener。
- 2) 创建定时调度的任务 scheduleTask，此任务主要调用 schedule 方法。
- 3) 将 scheduleTask 提交给 executor (executor 是只有一个线程的 ScheduledThreadPoolExecutor)，以固定的间隔 intervalMillis (值为 100) 进行调度。
- 4) 调用 ExecutorAllocationClient 的 requestTotalExecutors 方法请求所有的 Executor。numExecutorsTarget 是动态分配 Executor 的总数，取 spark.dynamicAllocation.initialExecutors、spark.dynamicAllocation.minExecutors、spark.executor.instances 三个属性配置的最大值。localityAwareTasks 是由本地性偏好的 Task 数量。hostToLocalTaskCount 是 Host 与想要在此节点上运行的 Task 的数量之间的映射关系。

 **注意** 在 SchedulerBackend 的实现类中只有 CoarseGrainedSchedulerBackend 同时实现了特质 ExecutorAllocationClient，因此对 CoarseGrainedSchedulerBackend 实现的 request-

TotalExecutors 方法的具体内容将放在第 9 章详细介绍。

定时任务 scheduleTask 会按照固定的时间间隔调用 ExecutorAllocationManager 的 schedule 方法，以调整待执行 Executor 请求的数量和运行的 Executor 的数量。schedule 方法的实现如下。

```
private def schedule(): Unit = synchronized {
    val now = clock.currentTimeMillis
    updateAndSyncNumExecutorsTarget(now)
    val executorIdsToBeRemoved = ArrayBuffer[String]()
    removeTimesretain { case (executorId, expireTime) =>
        val expired = now >= expireTime
        if (expired) {
            initializing = false
            executorIdsToBeRemoved += executorId
        }
        !expired
    }
    if (executorIdsToBeRemoved.nonEmpty) {
        removeExecutors(executorIdsToBeRemoved)
    }
}
```

根据上述代码，schedule 方法的执行步骤如下。

- 1) 调用 updateAndSyncNumExecutorsTarget 方法重新计算所需的 Executor 数量，并更新请求的 Executor 数量。

- 2) 对过期的 Executor 进行删除。removeExecutors 方法将利用 ExecutorAllocationClient 的 killExecutors 方法通知集群管理器“杀死”Executor。killExecutors 方法需要 ExecutorAllocationClient 的实现类去实现。

updateAndSyncNumExecutorsTarget 方法的实现如下。

```
private def updateAndSyncNumExecutorsTarget(now: Long): Int = synchronized {
    val maxNeeded = maxNumExecutorsNeeded // 获得实际需要的Executor的最大数量maxNeeded

    if (initializing) { // ExecutorAllocationManager还在初始化，则返回0
        0
    } else if (maxNeeded < numExecutorsTarget) { // 减少需要的Executor的数量
        val oldNumExecutorsTarget = numExecutorsTarget
        numExecutorsTarget = math.max(maxNeeded, minNumExecutors)
        numExecutorsToAdd = 1

        if (numExecutorsTarget < oldNumExecutorsTarget) {
            client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks,
                hostToLocalTaskCount)
            logDebug(s"Lowering target number of executors to $numExecutorsTarget
            (previously " +
                s"$oldNumExecutorsTarget) because not all requested executors are
            actually needed")
        }
        numExecutorsTarget - oldNumExecutorsTarget
    } else if (addTime != NOT_SET && now >= addTime) { // 添加Executor
        numExecutorsTarget + 1
    } else {
        numExecutorsTarget
    }
}
```

```
    val delta = addExecutors(maxNeeded)
    logDebug(s"Starting timer to add more executors (to " +
      s"expire in $sustainedSchedulerBacklogTimeouts seconds)")
    addTime += sustainedSchedulerBacklogTimeouts * 1000
    delta
  } else {
    0
  }
}
```

根据上述代码，`updateAndSyncNumExecutorsTarget`方法的执行步骤如下。

- 1) 调用 `maxNumExecutorsNeeded` 方法获得实际需要的 Executor 的最大数量 `maxNeeded`。
 - 2) 如果 `ExecutorAllocationManager` 还在初始化，则返回 0。
 - 3) 如果 Executor 的目标数量 (`numExecutorsTarget`) 超过我们实际需要的数量 (`maxNeeded`)，那么首先将 `numExecutorsTarget` 设置为 `maxNeeded` 与最小 Executor 数量 (`minNumExecutors`) 之间的最大值，然后调用 `ExecutorAllocationClient` 的 `requestTotalExecutors` 方法重新请求 `numExecutorsTarget` 指定的目标 Executor 数量，以此停止添加新的执行程序，并通知集群管理器取消额外的待处理 Executor 的请求，最后返回减少的 Executor 数量。
 - 4) 如果 `maxNeeded` 大于等于 `numExecutorsTarget`，且当前时间大于上次添加 Executor 的时间，那么首先调用 `addExecutors` 方法[⊖]通知集群管理器添加额外的 Executor，然后更新添加 Executor 的时间，最后返回添加的 Executor 数量。

这里对 ExecutorAllocationManager 的主要工作原理进行了分析，maxNumExecutors-Needed 方法、addExecutors 方法及 removeExecutors 方法的实现留给感兴趣的读者自行阅读。local-cluster 和 Standalone 模式下的集群管理器 Master 如何给应用程序分配或取消 Executor 的分析，将在 9.9.5 节详细介绍。

根据对 ExecutorAllocationManager 的分析，可以用图 4-4 来表示 Executor 的动态分配过程。

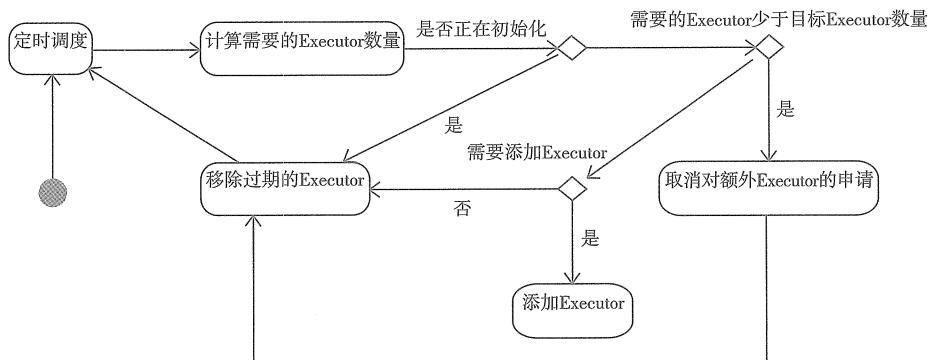


图 4-4 Executor 的动态分配过程

⊖ 此方法也是利用 ExecutorAllocationClient 的 requestTotalExecutors 方法。

4.10 ContextCleaner 的创建与启动

ContextCleaner 用于清理那些超出应用范围的 RDD、Shuffle 对应的 map 任务状态、Shuffle 元数据、Broadcast 对象及 RDD 的 Checkpoint 数据。

4.10.1 创建 ContextCleaner

创建 ContextCleaner 的代码如下。

```
_cleaner =
  if (_conf.getBoolean("spark.cleaner.referenceTracking", true)) {
    Some(new ContextCleaner(this))
  } else {
    None
  }
_cleaner.foreach(_.start())
```

根据上述代码，我们知道可以通过配置属性 spark.cleaner.referenceTracking（默认是 true）来决定是否启用 ContextCleaner。

ContextCleaner 的组成如下。

- referenceQueue：缓存顶级的 AnyRef 引用。
- referenceBuffer：缓存 AnyRef 的虚引用。
- listeners：缓存清理工作的监听器数组。
- cleaningThread：用于具体清理工作的线程。此线程为守护线程，名称为 Spark Context Cleaner。
- periodicGCService：类型为 ScheduledExecutorService，用于执行 GC（Garbage Collection，垃圾收集）的调度线程池，此线程池只包含一个线程，启动的线程名称以 context-cleaner-periodic-gc 开头。
- periodicGCIInterval：执行 GC 的时间间隔。可通过 spark.cleaner.periodicGC.interval 属性进行配置，默认是 30 分钟。
- blockOnCleanupTasks：清理非 Shuffle 的其他数据是否是阻塞式的。可通过 spark.cleaner.referenceTracking.blocking 属性进行配置，默认是 true。
- blockOnShuffleCleanupTasks：清理 Shuffle 数据是否是阻塞式的。可通过 spark.cleaner.referenceTracking.blocking.shuffle 属性进行配置，默认是 false。清理 Shuffle 数据包括清理 MapOutputTracker 中指定 ShuffleId 对应的 map 任务状态和 ShuffleManager 中注册的 ShuffleId 对应的 Shuffle 元数据。
- stopped：ContextCleaner 是否停止的状态标记。

4.10.2 启动 ContextCleaner

SparkContext 在初始化的过程中会启动 ContextCleaner，只有这样 ContextCleaner

才能够清理那些超出应用范围的 RDD、Shuffle 对应的 map 任务状态、Shuffle 元数据、Broadcast 对象及 RDD 的 Checkpoint 数据。启动 ContextCleaner 的代码如下。

```
def start(): Unit = {
    cleaningThread.setDaemon(true)
    cleaningThread.setName("Spark Context Cleaner")
    cleaningThread.start()
    periodicGCService.scheduleAtFixedRate(new Runnable {
        override def run(): Unit = System.gc()
    }, periodicGCIInterval, periodicGCIInterval, TimeUnit.SECONDS)
}
```

根据上述代码，启动 ContextCleaner 的步骤如下。

- 1) 将 cleaningThread 设置为守护线程，并指定名称为 Spark Context Cleaner。
- 2) 启动 cleaningThread。
- 3) 给 periodicGCService 设置以 periodicGCIInterval 作为时间间隔定时进行 GC 操作的任务。

除了 GC 的定时器，ContextCleaner 其余部分的工作原理和 listenerBus 一样（也采用监听器模式，由异步线程来处理），因此不再赘述。异步线程实际只是调用 keepCleaning 方法，其实现如代码清单 4-20 所示。

代码清单4-20 keepCleaning的实现

```
private def keepCleaning(): Unit = Utils.tryOrStopSparkContext(sc) {
    while (!stopped) {
        try {
            val reference = Option(referenceQueue.remove(ContextCleaner.REF_QUEUE_POLL_TIMEOUT))
                .map(_.asInstanceOf[CleanupTaskWeakReference])
            synchronized {
                reference.foreach { ref =>
                    logDebug("Got cleaning task " + ref.task)
                    referenceBuffer.remove(ref)
                    ref.task match {
                        case CleanRDD(rddId) =>
                            doCleanupRDD(rddId, blocking = blockOnCleanupTasks)
                        case CleanShuffle(shuffleId) =>
                            doCleanupShuffle(shuffleId, blocking = blockOnShuffleCleanupTasks)
                        case CleanBroadcast(broadcastId) =>
                            doCleanupBroadcast(broadcastId, blocking = blockOnCleanupTasks)
                        case CleanAccum(accId) =>
                            doCleanupAccum(accId, blocking = blockOnCleanupTasks)
                        case CleanCheckpoint(rddId) =>
                            doCleanCheckpoint(rddId)
                    }
                }
            }
        } catch {
            case ie: InterruptedException if stopped => // ignore
            case e: Exception => logError("Error in cleaning thread", e)
        }
    }
}
```

```

    }
}
```

从代码清单 4-20 可以看出，异步线程将匹配各种引用，并执行相应的方法进行清理。以 doCleanupRDD 为例，其实现如代码清单 4-21 所示。

代码清单4-21 清理RDD数据

```

def doCleanupRDD(rddId: Int, blocking: Boolean): Unit = {
  try {
    logDebug("Cleaning RDD " + rddId)
    sc.unpersistRDD(rddId, blocking)
    listeners.asScala.foreach(_.rddCleaned(rddId))
    logInfo("Cleaned RDD " + rddId)
  } catch {
    case e: Exception => logError("Error cleaning RDD " + rddId, e)
  }
}
```

根据代码清单 4-21，doCleanupRDD 的执行步骤如下。

- 1) 调用 SparkContext 的 unpersistRDD 方法从内存或磁盘中移除 RDD。
- 2) 从 persistentRdds 中移除对 RDD 的跟踪。
- 3) 调用所有监听器的 rddCleaned 方法。

4.11 额外的 SparkListener 与启动事件总线

SparkContext 中提供了添加用于自定义 SparkListener 的地方，代码如下。

```
setupAndStartListenerBus()
```

setupAndStartListenerBus 方法的实现如下。

```

private def setupAndStartListenerBus(): Unit = {
  try {
    // 获取用户自定义的SparkListener的类名
    val listenerClassNames: Seq[String] =
      conf.get("spark.extraListeners", "").split(',').map(_.trim).filter(_ != "")
    // 通过反射生成每一个自定义SparkListener的实例，并添加到事件总线的监听器列表中
    for (className <- listenerClassNames) {
      // Use reflection to find the right constructor
      val constructors = {
        val listenerClass = Utils.classForName(className)
        listenerClass
          .getConstructors
          .asInstanceOf[Array[Constructor[_ <: SparkListenerInterface]]]
      }
      val constructorTakingSparkConf = constructors.find { c =>
        c.getParameterTypes.sameElements(Array(classOf[SparkConf]))
      }
      lazy val zeroArgumentConstructor = constructors.find { c =>
```

```

    c.getParameterTypes.isEmpty
  }
  val listener: SparkListenerInterface = {
    if (constructorTakingSparkConf.isDefined) {
      constructorTakingSparkConf.get.newInstance(conf)
    } else if (zeroArgumentConstructor.isDefined) {
      zeroArgumentConstructor.get.newInstance()
    } else {
      throw new SparkException(
        s"$className did not have a zero-argument constructor or a" +
        " single-argument constructor that accepts SparkConf. Note: if the
        class is" +
        " defined inside of another Scala class, then its constructors may
        accept an" +
        " implicit parameter that references the enclosing class; in this
        case, you must" +
        " define the listener as a top-level class in order to prevent this
        extra" +
        " parameter from breaking Spark's ability to find a valid const-
        ructor.")
    }
  }
  listenerBus.addListener(listener)
  logInfo(s"Registered listener $className")
}
} catch {
  case e: Exception =>
    try {
      stop()
    } finally {
      throw new SparkException(s"Exception when registering SparkListener", e)
    }
}
// 启动事件总线，并将_listenerBusStarted置为true
listenerBus.start()
_listenerBusStarted = true
}

```

根据上述代码，setupAndStartListenerBus 的执行步骤如下。

- 1) 从 spark.extraListeners 属性中获取用户自定义的 SparkListener 的类名。用户可以通过逗号分隔多个自定义 SparkListener。
- 2) 通过反射生成每一个自定义 SparkListener 的实例，并添加到事件总线的监听器列表中。
- 3) 启动事件总线，并将 _listenerBusStarted 置为 true。

4.12 Spark 环境更新

用户提交任务时往往需要添加额外的 Jar 包或其他文件，用户任务的执行将依赖这些文件。这些文件该如何指定？任务在各个节点上运行时又是如何获取到这些文件的呢？我们

首先回答第一个问题。

在 SparkContext 的初始化过程中会读取用户指定的 Jar 文件或其他文件，代码如下。

```
_jars = Utils.getUserJars(_conf)
_files = _conf.getOption("spark.files").map(_.split(",")).map(_.filter(_.
    nonEmpty))
.toSeq.flatten
```

上述代码首先读取用户设置的 Jar 文件，然后读取用户设置的其他文件。当用户选择的部署模式是 YARN 时，_jars 是由 spark.jars 属性指定的 Jar 文件和 spark.yarn.dist.jars 属性指定的 Jar 文件的并集。其他模式下只采用由 spark.jars 属性指定的 Jar 文件。这里使用了 Utils 的 getUserJars 方法，其具体介绍请阅读附录 A。通过 spark.files 属性可以指定其他文件。

回答了第一个问题，第二个问题该如何解决？

在 SparkContext 的初始化过程中有以下代码。

```
def jars: Seq[String] = _jars
def files: Seq[String] = _files
// Add each JAR given through the constructor
if (jars != null) {
    jars.foreach(addJar)
}
if (files != null) {
    files.foreach(addFile)
}
```

上述代码中，jars 和 files 是两个简单的方法，分别用来获取 Jar 包的序列集合和其他文件的序列集合。上述代码还遍历每一个 Jar 文件并调用 addJar 方法，遍历每一个其他文件并调用 addFile 方法。

addJar 方法是做什么的呢？它用于将 Jar 文件添加到 Driver 的 RPC 环境中。addJar 的实现如代码清单 4-22 所示。

代码清单 4-22 addJar 的实现

```
def addJar(path: String) {
    if (path == null) {
        logWarning("null specified as parameter to addJar")
    } else {
        var key = ""
        if (path.contains("\\\\")) {
            key = env.rpcEnv.fileServer.addJar(new File(path))
        } else {
            val uri = new URI(path)
            Utils.validateURL(uri)
            key = uri.getScheme match {
                case null | "file" =>
                    try {
                        env.rpcEnv.fileServer.addJar(new File(uri.getPath))
                    } catch {
```

```

        case exc: FileNotFoundException =>
          logError(s"Jar not found at $path")
          null
      }
      case "local" =>
        "file:" + uri.getPath
      case _ =>
        path
    }
}
if (key != null) {
  val timestamp = System.currentTimeMillis
  if (addedJars.putIfAbsent(key, timestamp).isEmpty) {
    logInfo(s"Added JAR $path at $key with timestamp $timestamp")
    postEnvironmentUpdate()
  }
}
}

```

根据代码清单 4-22，将调用 SparkEnv 的 RpcEnv 的 fileServer（fileServer 实际是 5.3.5 节要介绍的 NettyStreamManager）的 addJar 方法把 Jar 文件添加到 Driver 本地 RpcEnv 的 NettyStreamManager 中，并将 Jar 文件添加的时间戳信息缓存到 addedJars 中。SparkEnv 及 fileServer 的内容将在第 5 章详细介绍。

`addFile` 与 `addJar` 类似，其实现如代码清单 4-23 所示。

代码清单4-23 addFile的实现

```
def addFile(path: String): Unit = {
  addFile(path, false)
}
def addFile(path: String, recursive: Boolean): Unit = {
  val uri = new Path(path).toUri
  val schemeCorrectedPath = uri.getScheme match {
    case null | "local" => new File(path).getCanonicalFile.toURI.toString
    case _ => path
  }
  val hadoopPath = new Path(schemeCorrectedPath)
  val scheme = new URI(schemeCorrectedPath).getScheme
  if (!Array("http", "https", "ftp").contains(scheme)) {
    val fs = hadoopPath.getFileSystem(hadoopConfiguration)
    val isDir = fs.getFileStatus(hadoopPath).isDirectory
    if (!isLocal && scheme == "file" && isDir) {
      throw new SparkException(s"addFile does not support local directories when
        not running " +
        "local mode.")
    }
    if (!recursive && isDir) {
      throw new SparkException(s"Added file $hadoopPath is a directory and
recursive is not " +
        "turned on.")
    }
  } else {
}
```

```

        Utils.validateURL(uri)
    }

    val key = if (!isLocal && scheme == "file") {
        env.rpcEnv.fileServer.addFile(new File(uri.getPath))
    } else {
        schemeCorrectedPath
    }
    val timestamp = System.currentTimeMillis
    if (addedFiles.putIfAbsent(key, timestamp).isEmpty) {
        logInfo(s"Added file $path at $key with timestamp $timestamp")
        Utils.fetchFile(uri.toString, new File(SparkFiles.getRootDirectory()), conf,
            env.securityManager, hadoopConfiguration, timestamp, useCache = false)
        postEnvironmentUpdate()
    }
}

```

根据代码清单 4-23，将调用 SparkEnv 的 RpcEnv 的 fileServer 的 addFile 方法将文件添加到 Driver 本地 RpcEnv 的 NettyStreamManager 中，并将文件添加的时间戳信息缓存到 addedFiles 中。

通过 addJar 和 addFile 可以将各种任务执行所依赖的文件添加到 Driver 的 RPC 环境中，这样各个 Executor 节点就可以使用 RPC 从 Driver 将文件下载到本地，以供任务执行。

在 addJar 和 addFile 方法的最后都调用了 postEnvironmentUpdate 方法，而且在 SparkContext 初始化过程的最后也会调用 postEnvironmentUpdate，代码如下。

```
postEnvironmentUpdate()
```

由于 addJar 和 addFile 可能会对应用的环境产生影响，所以在 SparkContext 初始化过程的最后需要调用 postEnvironmentUpdate 方法更新环境。postEnvironmentUpdate 的实现如代码清单 4-24 所示。

代码清单4-24 postEnvironmentUpdate的实现

```

private def postEnvironmentUpdate() {
    if (taskScheduler != null) {
        val schedulingMode = getSchedulingMode.toString
        val addedJarPaths = addedJars.keys.toSeq
        val addedFilePaths = addedFiles.keys.toSeq
        // 将 JVM 参数、Spark 属性、系统属性、classPath 等信息设置为环境明细信息
        val environmentDetails = SparkEnv.environmentDetails(conf, schedulingMode,
            addedJarPaths, addedFilePaths)
        // 生成 SparkListenerEnvironmentUpdate 事件，并投递到事件总线
        val environmentUpdate = SparkListenerEnvironmentUpdate(environmentDetails)
        listenerBus.post(environmentUpdate)
    }
}

```

根据代码清单 4-24，postEnvironmentUpdate 的处理步骤如下。

- 1) 通过调用 SparkEnv 的方法 environmentDetails（见代码清单 4-25），将环境的 JVM

参数、Spark属性、系统属性、classPath等信息设置为环境明细信息。

2) 生成事件 SparkListenerEnvironmentUpdate (此事件携带环境明细信息), 并投递到事件总线 listenerBus, 此事件最终将被 EnvironmentListener 监听, 并影响 EnvironmentPage 页面中的输出内容。

代码清单4-25 投递环境更新事件

```

private[spark]
def environmentDetails(
    conf: SparkConf,
    schedulingMode: String,
    addedJars: Seq[String],
    addedFiles: Seq[String]): Map[String, Seq[(String, String)]] = {
  import Properties._
  val jvmInformation = Seq(
    ("Java Version", s"$javaVersion ($javaVendor)"),
    ("Java Home", javaHome),
    ("Scala Version", versionString)
  ).sorted
  val schedulerMode =
    if (!conf.contains("spark.scheduler.mode")) {
      Seq(("spark.scheduler.mode", schedulingMode))
    } else {
      Seq[(String, String)]()
    }
  val sparkProperties = (conf.getAll ++ schedulerMode).sorted

  val systemProperties = Utils.getSystemProperties.toSeq
  val otherProperties = systemProperties.filter { case (k, _) =>
    k != "java.class.path" && !k.startsWith("spark.")
  }.sorted

  val classPathEntries = javaClassPath
    .split(File.pathSeparator)
    .filterNot(_.isEmpty)
    .map((_, "System Classpath"))

  val addedJarsAndFiles = (addedJars ++ addedFiles).map((_, "Added By User"))
  val classPaths = (addedJarsAndFiles ++ classPathEntries).sorted

  Map[String, Seq[(String, String)]](
    "JVM Information" -> jvmInformation,
    "Spark Properties" -> sparkProperties,
    "System Properties" -> otherProperties,
    "Classpath Entries" -> classPaths)
}

```

4.13 SparkContext 初始化的收尾

在 SparkContext 初始化过程的最后, 有一些收尾工作要做, 代码如下。

```
postApplicationStart() // 向事件总线投递SparkListenerApplicationStart事件
```

```

_taskScheduler.postStartHook() // 等待SchedulerBackend准备完成
// 向度量系统注册Source
_env.metricsSystem.registerSource(_dagScheduler.metricsSource)
_env.metricsSystem.registerSource(new BlockManagerSource(_env.blockManager))
_executorAllocationManager.foreach { e =>
  _env.metricsSystem.registerSource(e.executorAllocationManagerSource)
}
// 添加SparkContext的关闭钩子
logDebug("Adding shutdown hook") // force eager creation of logger
_shutdownHookRef = ShutdownHookManager.addShutdownHook(
  ShutdownHookManager.SPARK_CONTEXT_SHUTDOWN_PRIORITY) { () =>
  logInfo("Invoking stop() from shutdown hook")
  stop()
}
// 将SparkContext标记为激活
SparkContext.setActiveContext(this, allowMultipleContexts)

```

根据上述代码，收尾的工作包括以下几项。

- 1) 调用 postApplicationStart 方法（见代码清单 4-26），向事件总线投递 SparkListenerApplicationStart（即应用启动事件）。

代码清单4-26 postApplicationStart的实现

```

private def postApplicationStart() {
  listenerBus.post(SparkListenerApplicationStart(appName, Some(applicationId),
    startTime, sparkUser, applicationAttemptId, schedulerBackend.
    getDriverLogUrls))
}

```

- 2) 调用 TaskScheduler 的 postStartHook 方法（此方法的实现见代码清单 7-108），等待 SchedulerBackend 准备完成。

- 3) 向度量系统注册 DAGSchedulerSource、BlockManagerSource 及 ExecutorAllocation ManagerSource。

- 4) 添加 SparkContext 的关闭钩子，使得 JVM 退出之前调用 SparkContext 的 stop 方法进行一些关闭工作。

- 5) 将 SparkContext 标记为激活。setActiveContext 方法的实现如代码清单 4-31 所示。

4.14 SparkContext 提供的常用方法

1. broadcast

broadcast 方法用于广播给定的对象，其实现如代码清单 4-27 所示。

代码清单4-27 广播给定对象

```

def broadcast[T: ClassTag](value: T): Broadcast[T] = {
  assertNotStopped()
  require(!classOf[RDD[_]].isAssignableFrom(classTag[T].runtimeClass),
    "Can not directly broadcast RDDs; instead, call collect() and broadcast the
    
```

```

        result.")
    val bc = env.broadcastManager.newBroadcast[T](value, isLocal)
    val callSite = getCallSite
    logInfo("Created broadcast " + bc.id + " from " + callSite.shortForm)
    cleaner.foreach(_.registerBroadcastForCleanup(bc))
    bc
}

```

根据代码清单 4-27, broadcast 的实质是调用了 SparkEnv 的子组件 BroadcastManager 的 newBroadcast 方法生成广播对象。BroadcastManager 及其 newBroadcast 方法将在第 5 章详细介绍。

2. addSparkListener

addSparkListener 方法（见代码清单 4-28）用于向 LiveListenerBus 中添加实现了特质 SparkListenerInterface 的监听器。

代码清单4-28 添加SparkListener

```

@DeveloperApi
def addSparkListener(listener: SparkListenerInterface) {
    listenerBus.addListener(listener)
}

```

3. 多种多样的 runJob

SparkContext 提供了多个重载的 runJob 方法，但是这些 runJob 方法最终都将调用代码清单 4-29 所示的 runJob 方法。

代码清单4-29 runJob方法

```

def runJob[T, U: ClassTag] (
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    resultHandler: (Int, U) => Unit = {
    if (stopped.get()) {
        throw new IllegalStateException("SparkContext has been shutdown")
    }
    val callSite = getCallSite
    val cleanedFunc = clean(func)
    logInfo("Starting job: " + callSite.shortForm)
    if (conf.getBoolean("spark.logLineage", false)) {
        logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
    }
    // 将DAG及RDD提交给DAGScheduler进行调度
    dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler,
        localProperties.get)
    progressBar.foreach(_.finishAll())
    rdd.doCheckpoint() // 保存检查点
}

```

根据代码清单 4-29, runJob 方法的执行步骤如下。

1) 调用 DAGScheduler 的 runJob 方法将已经构建好 DAG 的 RDD 提交给 DAGScheduler 进行调度。DAGScheduler 将在 7.4 节进行详细介绍。

2) 调用 RDD 的 doCheckpoint 方法（见代码清单 10-25）保存检查点。

4. setCheckpointDir

SparkContext 的 setCheckpointDir 方法（见代码清单 4-30）用于给作业中的 RDD 指定检查点保存的目录。指定检查点目录是启用检查点机制的前提。本书将在 10.3.3 节详细介绍检查点的实现。

代码清单4-30 指定检查点保存的目录

```
def setCheckpointDir(directory: String) {
    if (!isLocal && Utils.nonLocalPaths(directory).isEmpty) {
        logWarning("Spark is not running in local mode, therefore the checkpoint
directory " +
        "must not be on the local filesystem. Directory '$directory' " +
        "appears to be on the local filesystem.")
    }
    checkpointDir = Option(directory).map { dir =>
        val path = new Path(dir, UUID.randomUUID().toString)
        val fs = path.getFileSystem(hadoopConfiguration)
        fs.mkdirs(path)
        fs.getFileStatus(path).getPath.toString
    }
}
```

4.15 SparkContext 的伴生对象

SparkContext 的伴生对象中提供了很多常用的功能，所以有必要对它们进行分析。SparkContext 的伴生对象中包含以下属性。

- VALID_LOG_LEVELS：有效的日志级别。VALID_LOG_LEVELS 包括 ALL、DEBUG、ERROR、FATAL、INFO、OFF、TRACE、WARN 等。
- SPARK_CONTEXT_CONSTRUCTOR_LOCK：对 SparkContext 进行构造时使用的锁，以此保证构造 SparkContext 的过程是线程安全的。
- activeContext：类型为 AtomicReference[SparkContext]，用于保存激活的 SparkContext。
- contextBeingConstructed：标记当前正在构造 SparkContext。

除了这些属性，SparkContext 的伴生对象还提供了一些常用的方法，下面一一进行介绍。

1. setActiveContext

setActiveContext 方法（见代码清单 4-31）用于将 SparkContext 作为激活的 SparkContext，保存到 activeContext 中。

代码清单4-31 设置激活的SparkContext

```

private[spark] def setActiveContext(
    sc: SparkContext,
    allowMultipleContexts: Boolean): Unit = {
  SPARK_CONTEXT_CONSTRUCTOR_LOCK.synchronized {
    assertNoOtherContextIsRunning(sc, allowMultipleContexts)
    contextBeingConstructed = None
    activeContext.set(sc)
  }
}

```

2. getOrCreate(config: SparkConf)

此方法（见代码清单 4-32）的功能为：如果没有激活的 SparkContext，则构造 SparkContext，并调用 setActiveContext 方法保存到 activeContext 中，最后返回激活的 SparkContext。

代码清单4-32 获取激活的SparkContext

```

def getOrCreate(config: SparkConf): SparkContext = {
  SPARK_CONTEXT_CONSTRUCTOR_LOCK.synchronized {
    if (activeContext.get() == null) {
      setActiveContext(new SparkContext(config), allowMultipleContexts = false)
    } else {
      if (config.getAll.nonEmpty) {
        logWarning("Using an existing SparkContext; some configuration may not
                   take effect.")
      }
    }
    activeContext.get()
  }
}

```

SparkContext 伴生对象中还提供了无参的 getOrCreate 方法，其实现与 getOrCreate(config: SparkConf) 方法类似。

4.16 小结

本章首先介绍了 SparkContext 的作用及其组成，然后按照执行顺序介绍 SparkContext 的初始化过程。初始化过程中需要创建并启动很多组件，例如，SparkEnv、SparkUI、HeartbeatReceiver、TaskScheduler、DAGScheduler、MetricsSystem、EventLoggingListener、ExecutorAllocationManager、ContextCleaner 等。其中重点介绍了基于 Jetty 构建的嵌入式 Web 服务 SparkUI 和 ContextCleaner，其他组件将放在后续章节中介绍。

最后还介绍了如何指定用户自定义的 SparkListener、如何指定用户自定义的 Jar 包及文件、投递环境更新事件及其他收尾工作。