# 第4章 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。

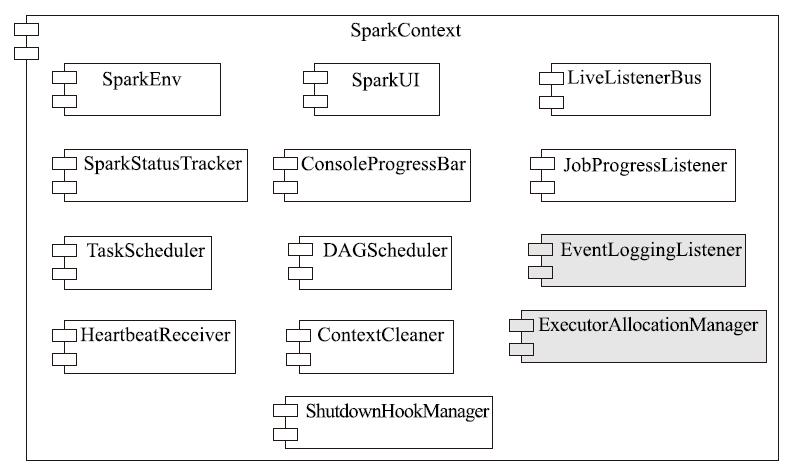


图4-1 SparkContext的组成 [[1]](#_1_29)

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Spark-Listener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 [[2]](#_2_5) ：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Spark-Context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。

[[1]](#_1_28) 图中深色的组件是SparkContext中的可选组件。

[[2]](#_2_4) creationSite通过调用Utils.getCallSite获得，Utils.getCallSite的详细信息见附录A。

## 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JobProgress-Listener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 [[1]](#_1_31) 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!")  
 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bind-Address。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。

[[1]](#_1_30) 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。

## 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，并增加了个性化的功能。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。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

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

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 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。

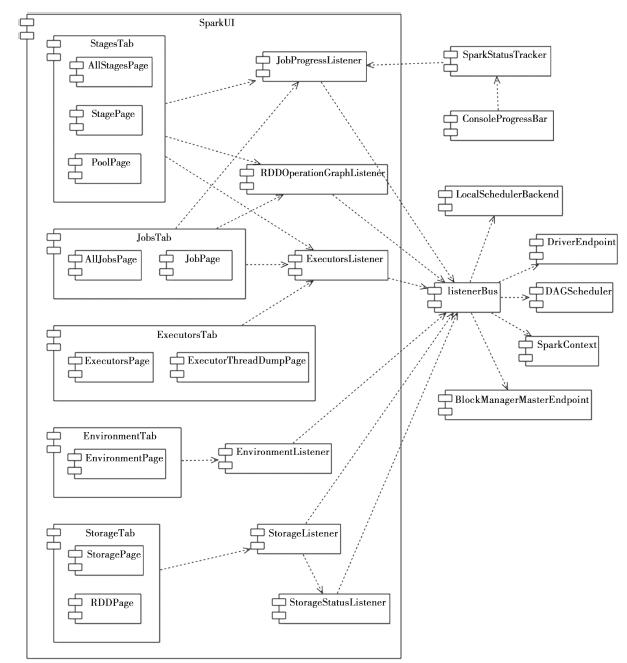


图4-2 SparkUI的组成

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Servlet-ContextHandler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getFormatted-ClassName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[ServletContextHandler]())  
 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create LiveUI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（有向无关图）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ServletContext-Handler。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}  
 <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内，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SparkListenerBlockManager-Removed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，Executor-AllocationManager是基于工作负载动态分配和删除Executor的代理。简单来讲，Executor-AllocationManager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。

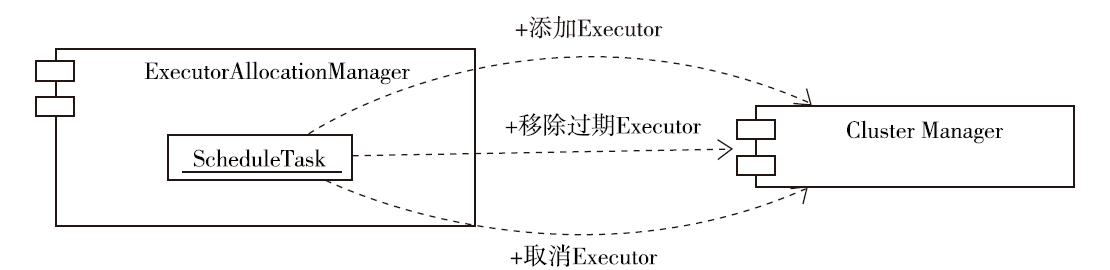


图4-3 ExecutorAllocationManager与集群管理器之间的关系

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Executor-AllocationManager。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, hostTo LocalTaskCount)  
}

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ScheduledThread-PoolExecutor），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。

Image00062.jpg 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.getTimeMillis  
 updateAndSyncNumExecutorsTarget(now)  
 val executorIdsToBeRemoved = ArrayBuffer[String]()  
 removeTimes.retain { 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Executor-AllocationClient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  
 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request-TotalExecutors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 [[1]](#_1_33) 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。

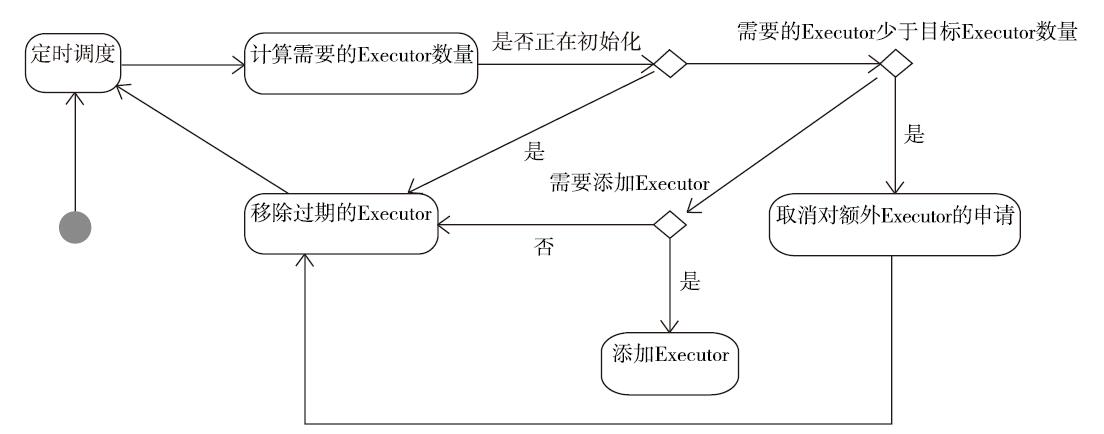


图4-4 Executor的动态分配过程

[[1]](#_1_32) 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。

·periodicGCInterval：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。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

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

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 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()  
 }, periodicGCInterval, periodicGCInterval, 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periodicGCInterval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Spark-Context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SparkListener-ApplicationStart（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ExecutorAllocationManagerSource。

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): 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DAG-Scheduler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 " +  
 s"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Spark-Context。

·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Spark-Context，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Spark-Context，并调用setActiveContext方法保存到activeContext中，最后返回激活的Spark-Context。

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。