大数据技术丛书

深入理解 Spark:
核心思想与源码分析

耿嘉安　著

HZ BOOKS
华章图书

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入理解 Spark：核心思想与源码分析 / 耿嘉安著 . —北京：机械工业出版社，2015.12
（大数据技术丛书）

ISBN 978-7-111-52234-8

I. 深…　II. 耿…　III. 数据处理软件　IV. TP274

中国版本图书馆 CIP 数据核字（2015）第 280808 号

HZ BOOKS
华章图书

深入理解 Spark：核心思想与源码分析

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：高婧雅　　责任校对：董纪丽
印　　刷：　　版　　次：2016 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：30.25
书　　号：ISBN 978-7-111-52234-8　　定　　价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

Preface 前　言

为什么写这本书

要回答这个问题，需要从我个人的经历说起。说来惭愧，我第一次接触计算机是在高三。当时跟大家一起去网吧玩 CS，跟身边的同学学怎么“玩”。正是通过这种“玩”的过程，让我了解到计算机并没有那么神秘，它也只是台机器，用起来似乎并不比打开电视机费劲多少。高考填志愿的时候，凭着直觉“糊里糊涂”就选择了计算机专业。等到真正学习计算机课程的时候却又发现，它其实很难！

早在 2004 年，还在学校的我跟很多同学一样，喜欢看 Flash，也喜欢谈论 Flash 甚至做 Flash。感觉 Flash 正如它的名字那样“闪光”。那些年，在学校里，知道 Flash 的人可要比知道 Java 的人多得多，这说明当时的 Flash 十分火热。此外，Oracle 也成为关系型数据库里的领军人物，很多人甚至觉得懂 Oracle 要比懂 Flash、Java 及其他数据库要厉害得多！

2007 年，我刚刚参加工作不久。那时 Struts1、Spring、Hibernate 几乎可以称为那些用 Java 作为开发语言的软件公司的三驾马车。很快，Struts2 替代了 Struts1 的地位，让我第一次意识到 IT 领域的技术更新竟然如此之快！随着很多传统软件公司向互联网公司转型，Hibernate 也难以确保其地位，iBATIS 诞生了！

2010 年，有关 Hadoop 的技术图书涌入中国，当时很多公司用它只是为了数据统计、数据挖掘或者搜索。一开始，人们对于 Hadoop 的认识和使用可能相对有限。大约 2011 年的时候，关于云计算的概念在网上炒得火热，当时依然在做互联网开发的我，对其只是“道听途说”。后来跟同事借了一本有关云计算的书，回家挑着看了一些内容，也没什么收获，怅然若失！ 20 世纪 60 年代，美国的军用网络作为互联网的雏形，很多内容已经与云计算中的某些说法类似。到 20 世纪 80 年代，互联网就已经启用了云计算，如今为什么又要重提这样的概念？这个问题我可能回答不了，还是交给历史吧。

2012 年，国内又呈现出大数据热的态势。从国家到媒体、教育、IT 等几乎所有领域，人人都在谈大数据。我的亲戚朋友中，无论老师、销售人员，还是工程师们都可以针对大数据谈谈自己的看法。我也找来一些 Hadoop 的书籍进行学习，希望能在其中探索到大数据的奥妙。

有幸在工作过程中接触到阿里的开放数据处理服务（open data processing service，ODPS），并且基于 ODPS 与其他小伙伴一起构建阿里的大数据商业解决方案——御膳房。去杭州出差的过程中，有幸认识和仲，跟他学习了阿里的实时多维分析平台——Garuda 和实时计算平台——Galaxy 的部分知识。和仲推荐我阅读 Spark 的源码，这样会对实时计算及流式计算有更深入的了解。2015 年春节期间，自己初次上网查阅 Spark 的相关资料学习，开始研究 Spark 源码。还记得那时只是出于对大数据的热爱，想使自己在这方面的技术能力有所提升。

从阅读 Hibernate 源码开始，到后来阅读 Tomcat、Spring 的源码，我也在从学习源码的过程中成长，我对源码阅读也越来越感兴趣。随着对 Spark 源码阅读的深入，发现很多内容从网上找不到答案，只能自己“硬啃”了。随着自己的积累越来越多，突然有一天发现，我所总结的这些内容好像可以写成一本书了！从闪光（Flash）到火花（Spark），足足有 11 个年头了。无论是 Flash、Java，还是 Spring、iBATIS，我一直扮演着一个追随者，我接受这些书籍的洗礼，从未给予。如今我也是 Spark 的追随者，不同的是，我不再只想简单攫取，还要给予。

最后还想说一下，2016 年是我从事 IT 工作的第 10 个年头，此书特别作为送给自己的 10 周年礼物。

本书特色

- 按照源码分析的习惯设计，从脚本分析到初始化再到核心内容，最后介绍 Spark 的扩展内容。整个过程遵循由浅入深、由深到广的基本思路。
- 本书涉及的所有内容都有相应的例子，以便于读者对源码的深入研究。
- 本书尽可能用图来展示原理，加速读者对内容的掌握。
- 本书讲解的很多实现及原理都值得借鉴，能帮助读者提升架构设计、程序设计等方面的能力。
- 本书尽可能保留较多的源码，以便于初学者能够在像地铁、公交这样的地方，也能轻松阅读。

读者对象

源码阅读是一项苦差事，人力和时间成本都很高，尤其是对于 Spark 陌生或者刚刚开始学习的人来说，难度可想而知。本书尽可能保留源码，使得分析过程不至于产生跳跃感，目的是降低大多数人的学习门槛。如果你是从事 IT 工作 1 ～ 3 年的新人或者是希望学习 Spark 核心知识的人，本书非常适合你。如果你已经对 Spark 有所了解或者已经在使用它，还想进一步提高自己，那么本书更适合你。

如果你是一个开发新手，对 Java、Linux 等基础知识不是很了解，那么本书可能不太适合你。如果你已经对 Spark 有深入的研究，本书也许可以作为你的参考资料。

总体说来，本书适合以下人群：

- 想要使用 Spark，但对 Spark 实现原理不了解，不知道怎么学习的人；
- 大数据技术爱好者，以及想深入了解 Spark 技术内部实现细节的人；
- 有一定 Spark 使用基础，但是不了解 Spark 技术内部实现细节的人；
- 对性能优化和部署方案感兴趣的大型互联网工程师和架构师；
- 开源代码爱好者。喜欢研究源码的同学可以从本书学到一些阅读源码的方式与方法。

本书不会教你如何开发 Spark 应用程序，只是用一些经典例子演示。本书简单介绍 Hadoop MapReduce、Hadoop YARN、Mesos、Tachyon、ZooKeeper、HDFS、Amazon S3， 但不会过多介绍这些框架的使用，因为市场上已经有丰富的这类书籍供读者挑选。本书也不会过多介绍 Scala、Java、Shell 的语法，读者可以在市场上选择适合自己的书籍阅读。

如何阅读本书

本书分为三大部分（不包括附录）：

准备篇（第 1 ～ 2 章），简单介绍了 Spark 的环境搭建和基本原理，帮助读者了解一些背景知识。

核心设计篇（第 3 ～ 7 章），着重讲解 SparkContext 的初始化、存储体系、任务提交与执行、计算引擎及部署模式的原理和源码分析。

扩展篇（第 8 ～ 11 章），主要讲解基于 Spark 核心的各种扩展及应用，包括：SQL 处理引擎、Hive 处理、流式计算框架 Spark Streaming、图计算框架 GraphX、机器学习库 MLlib 等内容。

本书最后还添加了几个附录，包括：附录 A 介绍的 Spark 中最常用的工具类 Utils ；附录 B 是 Akka 的简介与工具类 AkkaUtils 的介绍；附录 C 为 Jetty 的简介和工具类 JettyUtils 的介绍；附录 D 为 Metrics 库的简介和测量容器 MetricRegistry 的介绍；附录 E 演示了 Hadoop1.0 版本中的 word count 例子；附录 F 介绍了工具类 CommandUtils 的常用方法；附录 G 是关于 Netty 的简介和工具类 NettyUtils 的介绍；附录 H 列举了笔者编译 Spark 源码时遇到的问题及解决办法。

为了降低读者阅读理解 Spark 源码的门槛，本书尽可能保留源码实现，希望读者能够怀着一颗好奇的心，Spark 当前很火热，其版本更新也很快，本书以 Spark 1.2.3 版本为主，有兴趣的读者也可按照本书的方式，阅读 Spark 的最新源码。

勘误和支持

本书内容很多，限于笔者水平有限，书中内容难免有错误之处。在本书出版后的任何时间，如果你对本书有任何问题或者意见，都可以通过邮箱 beliefer@163.com 或博客 http://www.cnblogs.com/jiaan-geng/ 联系我，说出你的建议或者想法，希望与大家共同进步。

致谢

感谢苍天，让我生活在这样一个时代，能接触互联网和大数据；感谢父母，这么多年来，在学习、工作及生活上的帮助与支持；感谢妻子在生活中的照顾和谦让。

感谢杨福川和高婧雅给予本书出版的大力支持与帮助。

感谢冰夷老大和王贲老大让我有幸加入阿里，接触大数据应用；感谢和仲对 Galaxy 和 Garuda 耐心细致的讲解以及对 Spark 的推荐；感谢张中在百忙之中给本书写评语；感谢周亮、澄苍、民瞻、石申、清无、少侠、征宇、三步、谢衣、晓五、法星、曦轩、九翎、峰阅、丁卯、阿末、紫丞、海炎、涵康、云飏、孟天、零一、六仙、大知、井凡、隆君、太奇、晨炫、既望、宝升、都灵、鬼厉、归钟、梓撤、昊苍、水村、惜冰、惜陌、元乾等同仁在工作上的支持和帮助。

耿嘉安　于北京

HZ BOOKS
华章图书

Contents 目　录

前言

准　备　篇

第 1 章　环境准备 ····· 2

1.1　运行环境准备 ····· 2

1.1.1　安装 JDK ····· 3

1.1.2　安装 Scala ····· 3

1.1.3　安装 Spark ····· 4

1.2　Spark 初体验 ····· 4

1.2.1　运行 spark-shell ····· 4

1.2.2　执行 word count ····· 5

1.2.3　剖析 spark-shell ····· 7

1.3　阅读环境准备 ····· 11

1.4　Spark 源码编译与调试 ····· 13

1.5　小结 ····· 17

第 2 章　Spark 设计理念与基本架构 ····· 18

2.1　初识 Spark ····· 18

2.1.1　Hadoop MRv1 的局限 ····· 18

2.1.2　Spark 使用场景 ····· 20

2.1.3　Spark 的特点 ····· 20

2.2　Spark 基础知识 ····· 20

2.3　Spark 基本设计思想 ····· 22

2.3.1　Spark 模块设计 ····· 22

2.3.2　Spark 模型设计 ····· 24

2.4　Spark 基本架构 ····· 25

2.5　小结 ····· 26

核心设计篇

第 3 章　SparkContext 的初始化 ····· 28

3.1　SparkContext 概述 ····· 28

3.2　创建执行环境 SparkEnv ····· 30

3.2.1　安全管理器 SecurityManager ····· 31

3.2.2　基于 Akka 的分布式消息系统 ActorSystem ····· 31

3.2.3　map 任务输出跟踪器 mapOutputTracker ····· 32

3.2.4　实例化 ShuffleManager ····· 34

3.2.5　shuffle 线程内存管理器 ShuffleMemoryManager ····· 34

3.2.6　块传输服务 BlockTransferService ····· 35

3.2.7　BlockManagerMaster 介绍 ····· 35

HZ BOOKS 华章图书

3.2.8 创建块管理器 BlockManager……36
3.2.9 创建广播管理器 BroadcastManager……36
3.2.10 创建缓存管理器 CacheManager…37
3.2.11 HTTP 文件服务器 HttpFileServer……37
3.2.12 创建测量系统 MetricsSystem…39
3.2.13 创建 SparkEnv……40
3.3 创建 metadataCleaner……41
3.4 SparkUI 详解……42
3.4.1 listenerBus 详解……43
3.4.2 构造 JobProgressListener……46
3.4.3 SparkUI 的创建与初始化……47
3.4.4 Spark UI 的页面布局与展示……49
3.4.5 SparkUI 的启动……54
3.5 Hadoop 相关配置及 Executor 环境变量……54
3.5.1 Hadoop 相关配置信息……54
3.5.2 Executor 环境变量……54
3.6 创建任务调度器 TaskScheduler……55
3.6.1 创建 TaskSchedulerImpl……55
3.6.2 TaskSchedulerImpl 的初始化……57
3.7 创建和启动 DAGScheduler……57
3.8 TaskScheduler 的启动……60
3.8.1 创建 LocalActor……60
3.8.2 ExecutorSource 的创建与注册……62
3.8.3 ExecutorActor 的构建与注册……64
3.8.4 Spark 自身 ClassLoader 的创建…64
3.8.5 启动 Executor 的心跳线程……66
3.9 启动测量系统 MetricsSystem……69
3.9.1 注册 Sources……70
3.9.2 注册 Sinks……70
3.9.3 给 Sinks 增加 Jetty 的 ServletContextHandler……71
3.10 创建和启动 ExecutorAllocationManager……72
3.11 ContextCleaner 的创建与启动……73
3.12 Spark 环境更新……74
3.13 创建 DAGSchedulerSource 和 BlockManagerSource……76
3.14 将 SparkContext 标记为激活……77
3.15 小结……78

第 4 章 存储体系……79
4.1 存储体系概述……79
4.1.1 块管理器 BlockManager 的实现…79
4.1.2 Spark 存储体系架构……81
4.2 shuffle 服务与客户端……83
4.2.1 Block 的 RPC 服务……84
4.2.2 构造传输上下文 TransportContext……85
4.2.3 RPC 客户端工厂 TransportClientFactory……86
4.2.4 Netty 服务器 TransportServer……87
4.2.5 获取远程 shuffle 文件……88
4.2.6 上传 shuffle 文件……89
4.3 BlockManagerMaster 对 BlockManager 的管理……90
4.3.1 BlockManagerMasterActor……90
4.3.2 询问 Driver 并获取回复方法……92
4.3.3 向 BlockManagerMaster 注册 BlockManagerId……93
4.4 磁盘块管理器 DiskBlockManager……94
4.4.1 DiskBlockManager 的构造过程…94

HZ BOOKS 华章图书

4.4.2 获取磁盘文件方法 getFile ····· 96
4.4.3 创建临时 Block 方法 create-TempShuffleBlock ····· 96
4.5 磁盘存储 DiskStore ····· 97
4.5.1 NIO 读取方法 getBytes ····· 97
4.5.2 NIO 写入方法 putBytes ····· 98
4.5.3 数组写入方法 putArray ····· 98
4.5.4 Iterator 写入方法 putIterator ····· 98
4.6 内存存储 MemoryStore ····· 99
4.6.1 数据存储方法 putBytes ····· 101
4.6.2 Iterator 写入方法 putIterator 详解 ····· 101
4.6.3 安全展开方法 unrollSafely ····· 102
4.6.4 确认空闲内存方法 ensureFree-Space ····· 105
4.6.5 内存写入方法 putArray ····· 107
4.6.6 尝试写入内存方法 tryToPut ····· 108
4.6.7 获取内存数据方法 getBytes ····· 109
4.6.8 获取数据方法 getValues ····· 110
4.7 Tachyon 存储 TachyonStore ····· 110
4.7.1 Tachyon 简介 ····· 111
4.7.2 TachyonStore 的使用 ····· 112
4.7.3 写入 Tachyon 内存的方法 putIntoTachyonStore ····· 113
4.7.4 获取序列化数据方法 getBytes ····· 113
4.8 块管理器 BlockManager ····· 114
4.8.1 移出内存方法 dropFrom-Memory ····· 114
4.8.2 状态报告方法 reportBlockStatus ····· 116
4.8.3 单对象块写入方法 putSingle ····· 117
4.8.4 序列化字节块写入方法 putBytes ····· 118
4.8.5 数据写入方法 doPut ····· 118
4.8.6 数据块备份方法 replicate ····· 121
4.8.7 创建 DiskBlockObjectWriter 的方法 getDiskWriter ····· 125
4.8.8 获取本地 Block 数据方法 getBlockData ····· 125
4.8.9 获取本地 shuffle 数据方法 doGetLocal ····· 126
4.8.10 获取远程 Block 数据方法 doGetRemote ····· 127
4.8.11 获取 Block 数据方法 get ····· 128
4.8.12 数据流序列化方法 dataSerializeStream ····· 129
4.9 metadataCleaner 和 broadcast-Cleaner ····· 129
4.10 缓存管理器 CacheManager ····· 130
4.11 压缩算法 ····· 133
4.12 磁盘写入实现 DiskBlockObject-Writer ····· 133
4.13 块索引 shuffle 管理器 Index-ShuffleBlockManager ····· 135
4.14 shuffle 内存管理器 Shuffle-MemoryManager ····· 137
4.15 小结 ····· 138

第 5 章 任务提交与执行 ····· 139

5.1 任务概述 ····· 139
5.2 广播 Hadoop 的配置信息 ····· 142
5.3 RDD 转换及 DAG 构建 ····· 144
5.3.1 为什么需要 RDD ····· 144
5.3.2 RDD 实现分析 ····· 146
5.4 任务提交 ····· 152

5.4.1 任务提交的准备 …… 152
5.4.2 finalStage 的创建与 Stage 的划分 …… 157
5.4.3 创建 Job …… 163
5.4.4 提交 Stage …… 164
5.4.5 提交 Task …… 165
5.5 执行任务 …… 176
5.5.1 状态更新 …… 176
5.5.2 任务还原 …… 177
5.5.3 任务运行 …… 178
5.6 任务执行后续处理 …… 179
5.6.1 计量统计与执行结果序列化 …… 179
5.6.2 内存回收 …… 180
5.6.3 执行结果处理 …… 181
5.7 小结 …… 187

第 6 章 计算引擎 …… 188

6.1 迭代计算 …… 188
6.2 什么是 shuffle …… 192
6.3 map 端计算结果缓存处理 …… 194
6.3.1 map 端计算结果缓存聚合 …… 195
6.3.2 map 端计算结果简单缓存 …… 200
6.3.3 容量限制 …… 201
6.4 map 端计算结果持久化 …… 204
6.4.1 溢出分区文件 …… 205
6.4.2 排序与分区分组 …… 207
6.4.3 分区索引文件 …… 209
6.5 reduce 端读取中间计算结果 …… 210
6.5.1 获取 map 任务状态 …… 213
6.5.2 划分本地与远程 Block …… 215
6.5.3 获取远程 Block …… 217
6.5.4 获取本地 Block …… 218
6.6 reduce 端计算 …… 219
6.6.1 如何同时处理多个 map 任务的中间结果 …… 219
6.6.2 reduce 端在缓存中对中间计算结果执行聚合和排序 …… 220
6.7 map 端与 reduce 端组合分析 …… 221
6.7.1 在 map 端溢出分区文件，在 reduce 端合并组合 …… 221
6.7.2 在 map 端简单缓存、排序分组，在 reduce 端合并组合 …… 222
6.7.3 在 map 端缓存中聚合、排序分组，在 reduce 端组合 …… 222
6.8 小结 …… 223

第 7 章 部署模式 …… 224

7.1 local 部署模式 …… 225
7.2 local-cluster 部署模式 …… 225
7.2.1 LocalSparkCluster 的启动 …… 226
7.2.2 CoarseGrainedSchedulerBackend 的启动 …… 236
7.2.3 启动 AppClient …… 237
7.2.4 资源调度 …… 242
7.2.5 local-cluster 模式的任务执行 …… 253
7.3 Standalone 部署模式 …… 255
7.3.1 启动 Standalone 模式 …… 255
7.3.2 启动 Master 分析 …… 257
7.3.3 启动 Worker 分析 …… 259
7.3.4 启动 Driver Application 分析 …… 261
7.3.5 Standalone 模式的任务执行 …… 263
7.3.6 资源回收 …… 263
7.4 容错机制 …… 266
7.4.1 Executor 异常退出 …… 266

HZ BOOKS 华章图书

7.4.2 Worker 异常退出 …… 268
7.4.3 Master 异常退出 …… 269
7.5 其他部署方案 …… 276
7.5.1 YARN …… 277
7.5.2 Mesos …… 280
7.6 小结 …… 282

扩 展 篇

第 8 章 Spark SQL …… 284
8.1 Spark SQL 总体设计 …… 284
8.1.1 传统关系型数据库 SQL 运行原理 …… 285
8.1.2 Spark SQL 运行架构 …… 286
8.2 字典表 Catalog …… 288
8.3 Tree 和 TreeNode …… 289
8.4 词法解析器 Parser 的设计与实现 …… 293
8.4.1 SQL 语句解析的入口 …… 294
8.4.2 建表语句解析器 DDLParser …… 295
8.4.3 SQL 语句解析器 SqlParser …… 296
8.4.4 Spark 代理解析器 SparkSQL-Parser …… 299
8.5 Rule 和 RuleExecutor …… 300
8.6 Analyzer 与 Optimizer 的设计与实现 …… 302
8.6.1 语法分析器 Analyzer …… 304
8.6.2 优化器 Optimizer …… 305
8.7 生成物理执行计划 …… 306
8.8 执行物理执行计划 …… 308
8.9 Hive …… 311
8.9.1 Hive SQL 语法解析器 …… 311
8.9.2 Hive SQL 元数据分析 …… 313
8.9.3 Hive SQL 物理执行计划 …… 314
8.10 应用举例：JavaSparkSQL …… 314
8.11 小结 …… 320

第 9 章 流式计算 …… 321
9.1 Spark Streaming 总体设计 …… 321
9.2 StreamingContext 初始化 …… 323
9.3 输入流接收器规范 Receiver …… 324
9.4 数据流抽象 DStream …… 325
9.4.1 Dstream 的离散化 …… 326
9.4.2 数据源输入流 InputDStream …… 327
9.4.3 Dstream 转换及构建 DStream Graph …… 329
9.5 流式计算执行过程分析 …… 330
9.5.1 流式计算例子 CustomReceiver …… 331
9.5.2 Spark Streaming 执行环境构建 …… 335
9.5.3 任务生成过程 …… 347
9.6 窗口操作 …… 355
9.7 应用举例 …… 357
9.7.1 安装 mosquitto …… 358
9.7.2 启动 mosquitto …… 358
9.7.3 MQTTWordCount …… 359
9.8 小结 …… 361

第 10 章 图计算 …… 362
10.1 Spark GraphX 总体设计 …… 362
10.1.1 图计算模型 …… 363
10.1.2 属性图 …… 365
10.1.3 GraphX 的类继承体系 …… 367
10.2 图操作 …… 368
10.2.1 属性操作 …… 368

10.2.2 结构操作 368
10.2.3 连接操作 369
10.2.4 聚合操作 370
10.3 Pregel API 371
10.3.1 Dijkstra 算法 373
10.3.2 Dijkstra 的实现 376
10.4 Graph 的构建 377
10.4.1 从边的列表加载 Graph 377
10.4.2 在 Graph 中创建图的方法 377
10.5 顶点集合抽象 VertexRDD 378
10.6 边集合抽象 EdgeRDD 379
10.7 图分割 380
10.8 常用算法 382
10.8.1 网页排名 382
10.8.2 Connected Components 的应用 386
10.8.3 三角关系统计 388
10.9 应用举例 390
10.10 小结 391

第 11 章 机器学习 392

11.1 机器学习概论 392
11.2 Spark MLlib 总体设计 394
11.3 数据类型 394
11.3.1 局部向量 394
11.3.2 标记点 395
11.3.3 局部矩阵 396
11.3.4 分布式矩阵 396
11.4 基础统计 398
11.4.1 摘要统计 398
11.4.2 相关统计 399
11.4.3 分层抽样 401
11.4.4 假设检验 401
11.4.5 随机数生成 402
11.5 分类和回归 405
11.5.1 数学公式 405
11.5.2 线性回归 407
11.5.3 分类 407
11.5.4 回归 410
11.6 决策树 411
11.6.1 基本算法 411
11.6.2 使用例子 412
11.7 随机森林 413
11.7.1 基本算法 414
11.7.2 使用例子 414
11.8 梯度提升决策树 415
11.8.1 基本算法 415
11.8.2 使用例子 416
11.9 朴素贝叶斯 416
11.9.1 算法原理 416
11.9.2 使用例子 418
11.10 保序回归 418
11.10.1 算法原理 418
11.10.2 使用例子 419
11.11 协同过滤 419
11.12 聚类 420
11.12.1 K-means 420
11.12.2 高斯混合 422
11.12.3 快速迭代聚类 422
11.12.4 latent Dirichlet allocation 422
11.12.5 流式 K-means 423
11.13 维数减缩 424
11.13.1 奇异值分解 424
11.13.2 主成分分析 425

HZ BOOKS 华章图书

11.14 特征提取与转型 ····· 425
11.14.1 术语频率反转 ····· 425
11.14.2 单词向量转换 ····· 426
11.14.3 标准尺度 ····· 427
11.14.4 正规化尺度 ····· 428
11.14.5 卡方特征选择器 ····· 428
11.14.6 Hadamard 积 ····· 429
11.15 频繁模式挖掘 ····· 429
11.16 预言模型标记语言 ····· 430
11.17 管道 ····· 431
11.17.1 管道工作原理 ····· 432
11.17.2 管道 API 介绍 ····· 433
11.17.3 交叉验证 ····· 435
11.18 小结 ····· 436

附录 A Utils ····· 437
附录 B Akka ····· 446
附录 C Jetty ····· 450
附录 D Metrics ····· 453
附录 E Hadoop word count ····· 456
附录 F CommandUtils ····· 458
附录 G Netty ····· 461
附录 H 源码编译错误 ····· 465

HZ BOOKS
华章图书

准 备 篇

HZ BOOKS
华章图书

■ 第 1 章　环境准备
■ 第 2 章　Spark 设计理念与基本架构

Chapter 1

第 1 章

环境准备

凡事豫则立，不豫则废；言前定，则不跲；事前定，则不困。

——《礼记·中庸》

本章导读

在深入了解一个系统的原理、实现细节之前，应当先准备好它的源码编译环境、运行环境。如果能在实际环境安装和运行 Spark，显然能够提升读者对于 Spark 的一些感受，对系统能有个大体的印象，有经验的技术人员甚至能够猜出一些 Spark 采用的编程模型、部署模式等。当你通过一些途径知道了系统的原理之后，难道不会问问自己："这是怎么做到的？"如果只是游走于系统使用、原理了解的层面，是永远不可能真正理解整个系统的。很多 IDE 本身带有调试的功能，每当你阅读源码，陷入重围时，调试能让我们更加理解运行期的系统。如果没有调试功能，不敢想象阅读源码会怎样困难。

本章的主要目的是帮助读者构建源码学习环境，主要包括以下内容：

- 在 Windows 环境下搭建源码阅读环境；
- 在 Linux 环境下搭建基本的执行环境；
- Spark 的基本使用，如 spark-shell。

1.1 运行环境准备

考虑到大部分公司的开发和生成环境都采用 Linux 操作系统，所以笔者选用了 64 位的 Linux。在正式安装 Spark 之前，先要找台好机器。为什么？因为笔者在安装、编译、调试的过程中发现 Spark 非常耗费内存，如果机器配置太低，恐怕会跑不起来。Spark 的开发语言是

Scala，而 Scala 需要运行在 JVM 之上，因而搭建 Spark 的运行环境应该包括 JDK 和 Scala。

1.1.1　安装 JDK

使用命令 getconf LONG_BIT 查看 Linux 机器是 32 位还是 64 位，然后下载相应版本的 JDK 并安装。

下载地址：

http://www.oracle.com/technetwork/java/javase/downloads/index.html

配置环境：

```
cd ~
vim .bash_profile
```

添加如下配置：

```
export JAVA_HOME=/opt/java
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

由于笔者的机器上已经安装过 openjdk，所以未使用以上方式，openjdk 的安装命令如下：

```
$ su -c "yum install java-1.7.0-openjdk"
```

安装完毕后，使用 java –version 命令查看，确认安装正常，如图 1-1 所示。

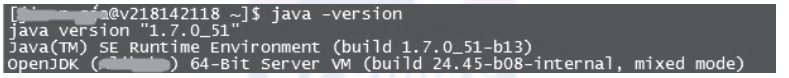

图 1-1　查看安装是否正常

1.1.2　安装 Scala

下载地址：http://www.scala-lang.org/download/

选择最新的 Scala 版本下载，下载方法如下：

```
wget http://downloads.typesafe.com/scala/2.11.5/scala-2.11.5.tgz
```

移动到选好的安装目录，例如：

```
mv scala-2.11.5.tgz ~/install/
```

进入安装目录，执行以下命令：

```
chmod 755 scala-2.11.5.tgz
tar -xzvf scala-2.11.5.tgz
```

配置环境：

```
cd ~
vim .bash_profile
```

添加如下配置：

```
export SCALA_HOME=$HOME/install/scala-2.11.5
export PATH=$PATH:$SCALA_HOME/bin:$HOME/bin
```

安装完毕后输入 scala，进入 scala 命令行说明 scala 安装正确，如图 1-2 所示。

图 1-2　进入 scala 命令行

1.1.3　安装 Spark

下载地址：http://spark.apache.org/downloads.html

选择最新的 Spark 版本下载，下载方法如下：

```
wget http://archive.apache.org/dist/spark/spark-1.2.0/spark-1.2.0-bin-hadoop1.tgz
```

移动到选好的安装目录，如：

```
mv spark-1.2.0-bin-hadoop1.tgz~/install/
```

进入安装目录，执行以下命令：

```
chmod 755 spark-1.2.0-bin-hadoop1.tgz
tar -xzvf spark-1.2.0-bin-hadoop1.tgz
```

配置环境：

```
cd ~
vim .bash_profile
```

添加如下配置：

```
export SPARK_HOME=$HOME/install/spark-1.2.0-bin-hadoop1
```

1.2　Spark 初体验

本节通过 Spark 的基本使用，让读者对 Spark 能有初步的认识，便于引导读者逐步深入学习。

1.2.1　运行 spark-shell

要运行 spark-shell，需要先对 Spark 进行配置。

1）进入 Spark 的 conf 文件夹：

```
cd ~/install/spark-1.2.0-bin-hadoop1/conf
```

2）复制一份 spark-env.sh.template，命名为 spark-env.sh，对它进行编辑，命令如下：

```
cp spark-env.sh.template spark-env.sh
vim spark-env.sh
```

3）添加如下配置：

```
export SPARK_MASTER_IP=127.0.0.1
export SPARK_LOCAL_IP=127.0.0.1
```

4）启动 spark-shell：

```
cd ~/install/spark-1.2.0-bin-hadoop1/bin
./spark-shell
```

最后我们会看到 spark 启动的过程，如图 1-3 所示。

```
15/07/09 11:24:50 INFO HttpServer: Starting HTTP Server
15/07/09 11:24:50 INFO Utils: Successfully started service 'HTTP class server' on port 41859.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.0
      /_/

Using Scala version 2.10.4 (OpenJDK (                ) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
15/07/09 11:24:55 INFO SecurityManager: Changing view acls to:
15/07/09 11:24:55 INFO SecurityManager: Changing modify acls to:
15/07/09 11:24:55 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users wit
: Set(                ); users with modify permissions: Set(                )
15/07/09 11:24:56 INFO Slf4jLogger: Slf4jLogger started
15/07/09 11:24:56 INFO Remoting: Starting remoting
15/07/09 11:24:56 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@localhost:
15/07/09 11:24:56 INFO Utils: Successfully started service 'sparkDriver' on port 32877.
15/07/09 11:24:56 INFO SparkEnv: Registering MapOutputTracker
15/07/09 11:24:56 INFO SparkEnv: Registering BlockManagerMaster
15/07/09 11:24:56 INFO DiskBlockManager: Created local directory at /tmp/spark-local-20150709112456-2a8c
15/07/09 11:24:56 INFO MemoryStore: MemoryStore started with capacity 265.4 MB
15/07/09 11:24:56 INFO HttpFileServer: HTTP File server directory is /tmp/spark-137e7243-75bf-42e5-99d3-e53a5
15/07/09 11:24:56 INFO HttpServer: Starting HTTP Server
15/07/09 11:24:56 INFO Utils: Successfully started service 'HTTP file server' on port 33509.
15/07/09 11:24:57 INFO Utils: Successfully started service 'SparkUI' on port 4040.
15/07/09 11:24:57 INFO SparkUI: Started SparkUI at http://localhost:4040
15/07/09 11:24:57 INFO Executor: Using REPL class URI: http://127.0.0.1:41859
15/07/09 11:24:57 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@localhost:32877/use
r
15/07/09 11:24:57 INFO NettyBlockTransferService: Server created on 50794
15/07/09 11:24:57 INFO BlockManagerMaster: Trying to register BlockManager
15/07/09 11:24:57 INFO BlockManagerMasterActor: Registering block manager localhost:50794 with 265.4 MB RAM,
iver>, localhost, 50794)
15/07/09 11:24:57 INFO BlockManagerMaster: Registered BlockManager
15/07/09 11:24:57 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala>
```

图 1-3　Spark 启动过程

从以上启动日志中我们可以看到 SparkEnv、MapOutputTracker、BlockManagerMaster、DiskBlockManager、MemoryStore、HttpFileServer、SparkUI 等信息。它们是做什么的？此处望文生义即可，具体内容将在后边的章节详细讲解。

1.2.2　执行 word count

这一节，我们通过 word count 这个耳熟能详的例子来感受下 Spark 任务的执行过程。启动 spark-shell 后，会打开 scala 命令行，然后按照以下步骤输入脚本。

1）输入 val lines = sc.textFile("../README.md", 2)，执行结果如图 1-4 所示。

```
scala> val lines = sc.textFile("../README.md", 2)
15/07/09 11:28:48 INFO MemoryStore: ensureFreeSpace(32768) called with curMem=0, maxMem=278302556
15/07/09 11:28:48 INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated size 32.
15/07/09 11:28:48 INFO MemoryStore: ensureFreeSpace(4959) called with curMem=32768, maxMem=278302556
15/07/09 11:28:48 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory (estimated si
15/07/09 11:28:48 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on localhost:42659 (size
15/07/09 11:28:48 INFO BlockManagerMaster: Updated info of block broadcast_0_piece0
15/07/09 11:28:48 INFO SparkContext: Created broadcast 0 from textFile at <console>:12
lines: org.apache.spark.rdd.RDD[String] = ../README.md MappedRDD[1] at textFile at <console>:12
```

图 1-4　步骤 1 执行结果

2）输入 val words = lines.flatMap(line => line.split(" ")），执行结果如图 1-5 所示。

```
scala> val words = lines.flatMap(line => line.split(" "))
words: org.apache.spark.rdd.RDD[String] = FlatMappedRDD[2] at flatMap at <console>:14
```

图 1-5　步骤 2 执行结果

3）输入 val ones = words.map(w => (w,1))，执行结果如图 1-6 所示。

```
scala> val ones = words.map(w => (w,1))
ones: org.apache.spark.rdd.RDD[(String, Int)] = MappedRDD[3] at map at <console>:16
```

图 1-6　步骤 3 执行结果

4）输入 val counts = ones.reduceByKey(_ + _)，执行结果如图 1-7 所示。

```
scala> val counts = ones.reduceByKey(_ + _)
15/07/09 11:29:10 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
here applicable
15/07/09 11:29:10 WARN LoadSnappy: Snappy native library not loaded
15/07/09 11:29:10 INFO FileInputFormat: Total input paths to process : 1
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:18
```

图 1-7　步骤 4 执行结果

5）输入 counts.foreach(println)，任务执行过程如图 1-8 和图 1-9[⊖]所示。输出结果如图 1-10 所示。

```
scala> counts.foreach(println)
15/07/09 11:29:31 INFO SparkContext: Starting job: foreach at <console>:21
15/07/09 11:29:31 INFO DAGScheduler: Registering RDD 3 (map at <console>:16)
15/07/09 11:29:31 INFO DAGScheduler: Got job 0 (foreach at <console>:21) with 2 output parti
15/07/09 11:29:31 INFO DAGScheduler: Final stage: Stage 1(foreach at <console>:21)
15/07/09 11:29:31 INFO DAGScheduler: Parents of final stage: List(Stage 0)
15/07/09 11:29:31 INFO DAGScheduler: Missing parents: List(Stage 0)
15/07/09 11:29:31 INFO DAGScheduler: Submitting Stage 0 (MappedRDD[3] at map at <console>:16
15/07/09 11:29:31 INFO MemoryStore: ensureFreeSpace(3544) called with curMem=37727, maxMem=2
15/07/09 11:29:31 INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated
15/07/09 11:29:31 INFO MemoryStore: ensureFreeSpace(2502) called with curMem=41271, maxMem=2
15/07/09 11:29:31 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (esti
15/07/09 11:29:31 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on localhost:426
15/07/09 11:29:31 INFO BlockManagerMaster: Updated info of block broadcast_1_piece0
15/07/09 11:29:31 INFO SparkContext: Created broadcast 1 from broadcast at DAGScheduler.scal
15/07/09 11:29:31 INFO DAGScheduler: Submitting 2 missing tasks from Stage 0 (MappedRDD[3] a
15/07/09 11:29:31 INFO TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
15/07/09 11:29:31 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost, PRO
15/07/09 11:29:31 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost, PRO
15/07/09 11:29:31 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
15/07/09 11:29:31 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
15/07/09 11:29:31 INFO HadoopRDD: Input split: file:/home/jiaan.gja/install/spark-1.2.0-bin-
15/07/09 11:29:31 INFO HadoopRDD: Input split: file:/home/jiaan.gja/install/spark-1.2.0-bin-
15/07/09 11:29:32 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1896 bytes result s
15/07/09 11:29:32 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1896 bytes result s
15/07/09 11:29:32 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 409 ms on l
15/07/09 11:29:32 INFO DAGScheduler: Stage 0 (map at <console>:16) finished in 0.438 s
15/07/09 11:29:32 INFO DAGScheduler: looking for newly runnable stages
15/07/09 11:29:32 INFO DAGScheduler: running: Set()
15/07/09 11:29:32 INFO DAGScheduler: waiting: Set(Stage 1)
15/07/09 11:29:32 INFO DAGScheduler: failed: Set()
15/07/09 11:29:32 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 426 ms on l
15/07/09 11:29:32 INFO DAGScheduler: Missing parents for Stage 1: List()
15/07/09 11:29:32 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all complete
15/07/09 11:29:32 INFO DAGScheduler: Submitting Stage 1 (ShuffledRDD[4] at reduceByKey at <c
15/07/09 11:29:32 INFO MemoryStore: ensureFreeSpace(2152) called with curMem=43773, maxMem=2
15/07/09 11:29:32 INFO MemoryStore: Block broadcast_2 stored as values in memory (estimated
15/07/09 11:29:32 INFO MemoryStore: ensureFreeSpace(1571) called with curMem=45925, maxMem=2
15/07/09 11:29:32 INFO MemoryStore: Block broadcast_2_piece0 stored as bytes in memory (esti
15/07/09 11:29:32 INFO BlockManagerInfo: Added broadcast_2_piece0 in memory on localhost:426
15/07/09 11:29:32 INFO BlockManagerMaster: Updated info of block broadcast_2_piece0
15/07/09 11:29:32 INFO SparkContext: Created broadcast 2 from broadcast at DAGScheduler.scal
15/07/09 11:29:32 INFO DAGScheduler: Submitting 2 missing tasks from Stage 1 (ShuffledRDD[4]
15/07/09 11:29:32 INFO TaskSchedulerImpl: Adding task set 1.0 with 2 tasks
```

图 1-8　步骤 5 执行过程部分（一）

```
15/07/09 11:29:32 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2, localhost, PROCESS
15/07/09 11:29:32 INFO TaskSetManager: Starting task 1.0 in stage 1.0 (TID 3, localhost, PROCESS
15/07/09 11:29:32 INFO Executor: Running task 0.0 in stage 1.0 (TID 2)
15/07/09 11:29:32 INFO Executor: Running task 1.0 in stage 1.0 (TID 3)
15/07/09 11:29:32 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks out of 2 blocks
15/07/09 11:29:32 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks out of 2 blocks
15/07/09 11:29:32 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 5 ms
15/07/09 11:29:32 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 5 ms
```

图 1-9　步骤 5 执行过程部分（二）

⊖　因截图时，一屏放不下，故分为两图。

```
(higher-level,1)
(need,1)
(guidance,3)
(Big,1)
(guide,,1)
(<class>,1)
(fast,1)
(uses,1)
(SQL,2)
(will,1)
(Java,,1)
(requires,1)
(,66)
(Documentation,1)
(web,1)
(cluster,2)
(using:,1)
(MLlib,1)
(shell:,2)
(Scala,,1)
(supports,2)
(built,,1)
(./dev/run-tests,1)
15/07/09 11:29:32 INFO Executor: Finished task 1.0 in stage 1.0 (TID 3). 824 bytes result sent
(sample,1)
15/07/09 11:29:32 INFO Executor: Finished task 0.0 in stage 1.0 (TID 2). 824 bytes result sent
15/07/09 11:29:32 INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 3) in 138 ms on loca
15/07/09 11:29:32 INFO DAGScheduler: Stage 1 (foreach at <console>:21) finished in 0.131 s
15/07/09 11:29:32 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 2) in 142 ms on loca
15/07/09 11:29:32 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed,
15/07/09 11:29:32 INFO DAGScheduler: Job 0 finished: foreach at <console>:21, took 0.733306 s
```

图 1-10　步骤 5 输出结果

在这些输出日志中，我们先是看到 Spark 中任务的提交与执行过程，然后看到单词计数的输出结果，最后打印一些任务结束的日志信息。有关任务的执行分析，笔者将在第 5 章中展开。

1.2.3　剖析 spark-shell

通过 word count 在 spark-shell 中执行的过程，我们想看看 spark-shell 做了什么。spark-shell 中有以下一段脚本，见代码清单 1-1。

代码清单1-1　spark-shell中的一段脚本

```
function main() {
    if $cygwin; then
stty -icanonmin 1 -echo > /dev/null 2>&1
        export SPARK_SUBMIT_OPTS="$SPARK_SUBMIT_OPTS -Djline.terminal=unix"
        "$FWDIR"/bin/spark-submit --class org.apache.spark.repl.Main "${SUBMISSION_
            OPTS[@]}" spark-shell "${APPLICATION_OPTS[@]}"
sttyicanon echo > /dev/null 2>&1
    else
        export SPARK_SUBMIT_OPTS
        "$FWDIR"/bin/spark-submit --class org.apache.spark.repl.Main "${SUBMISSION_
            OPTS[@]}" spark-shell "${APPLICATION_OPTS[@]}"
fi
}
```

我们看到脚本 spark-shell 里执行了 spark-submit 脚本，打开 spark-submit 脚本，发现其中包含以下脚本。

```
exec "$SPARK_HOME"/bin/spark-class org.apache.spark.deploy.SparkSubmit "${ORIG_
    ARGS[@]}"
```

脚本 spark-submit 在执行 spark-class 脚本时，给它增加了参数 SparkSubmit。打开 spark-class 脚本，其中包含以下脚本，见代码清单 1-2。

代码清单1-2 spark-class

```
if [ -n "${JAVA_HOME}" ]; then
    RUNNER="${JAVA_HOME}/bin/java"
else
    if [ `command -v java` ]; then
        RUNNER="java"
    else
       echo "JAVA_HOME is not set" >&2
       exit 1
    fi
fi

exec "$RUNNER" -cp "$CLASSPATH" $JAVA_OPTS "$@"
```

读到这里，应该知道 Spark 启动了以 SparkSubmit 为主类的 jvm 进程。

为便于在本地对 Spark 进程使用远程监控，给 spark-class 脚本追加以下 jmx 配置：

```
JAVA_OPTS="-XX:MaxPermSize=128m $OUR_JAVA_OPTS -Dcom.sun.management.jmxremote
    -Dcom.sun.management.jmxremote.port=10207 -Dcom.sun.management.jmxremote.
    authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
```

在本地打开 jvisualvm，添加远程主机，如图 1-11 所示。

右击已添加的远程主机，添加 JMX 连接，如图 1-12 所示。

图 1-11 添加远程主机

图 1-12 添加 JMX 连接

单击右侧的“线程”选项卡，选择 main 线程，然后单击“线程 Dump”按钮，如图 1-13 所示。

从 dump 的内容中找到线程 main 的信息，如代码清单 1-3 所示。

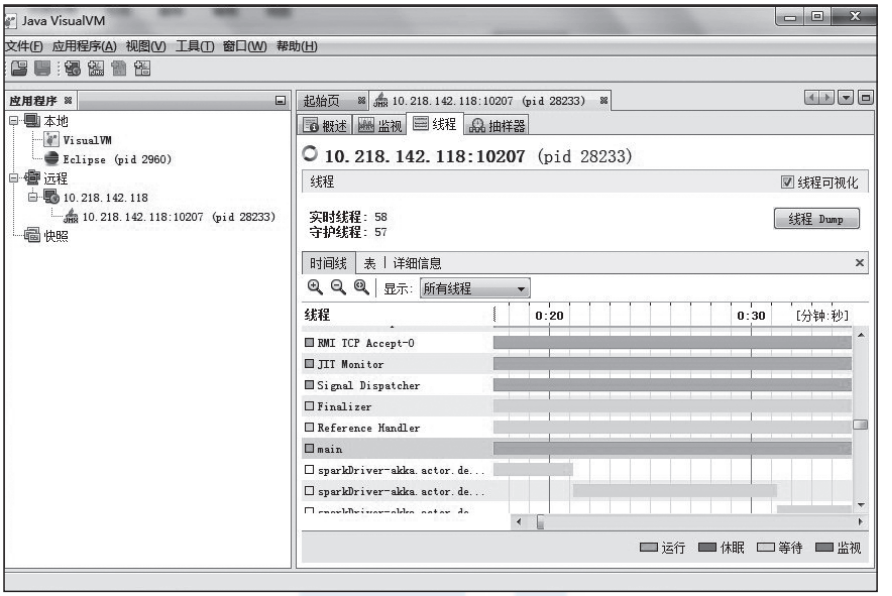

图 1-13 查看 Spark 线程

代码清单1-3 main线程dump信息

```
"main" - Thread t@1
    java.lang.Thread.State: RUNNABLE
        at java.io.FileInputStream.read0(Native Method)
        at java.io.FileInputStream.read(FileInputStream.java:210)
        at scala.tools.jline.TerminalSupport.readCharacter(TerminalSupport.java:152)
        at scala.tools.jline.UnixTerminal.readVirtualKey(UnixTerminal.java:125)
        at scala.tools.jline.console.ConsoleReader.readVirtualKey(ConsoleReader.
        java:933)
        at scala.tools.jline.console.ConsoleReader.readBinding(ConsoleReader.java:1136)
        at scala.tools.jline.console.ConsoleReader.readLine(ConsoleReader.java:1218)
        at scala.tools.jline.console.ConsoleReader.readLine(ConsoleReader.java:1170)
        at org.apache.spark.repl.SparkJLineReader.readOneLine(SparkJLineReader.
        scala:80)
        at scala.tools.nsc.interpreter.InteractiveReader$class.readLine(Interactive-
        Reader.scala:43)
        at org.apache.spark.repl.SparkJLineReader.readLine(SparkJLineReader.scala:25)
        at org.apache.spark.repl.SparkILoop.readOneLine$1(SparkILoop.scala:619)
        at org.apache.spark.repl.SparkILoop.innerLoop$1(SparkILoop.scala:636)
        at org.apache.spark.repl.SparkILoop.loop(SparkILoop.scala:641)
        at org.apache.spark.repl.SparkILoop$$anonfun$process$1.apply$mcZ$sp
        (SparkI-Loop.scala:968)
        at org.apache.spark.repl.SparkILoop$$anonfun$process$1.apply(SparkILoop.
        scala:916)
        at org.apache.spark.repl.SparkILoop$$anonfun$process$1.apply(SparkILoop.
        scala:916)
        at scala.tools.nsc.util.ScalaClassLoader$.savingContextLoader(ScalaClass
        Loader.scala:135)
        at org.apache.spark.repl.SparkILoop.process(SparkILoop.scala:916)
```

```
        at org.apache.spark.repl.SparkILoop.process(SparkILoop.scala:1011)
        at org.apache.spark.repl.Main$.main(Main.scala:31)
        at org.apache.spark.repl.Main.main(Main.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
        java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces-
        sorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.apache.spark.deploy.SparkSubmit$.launch(SparkSubmit.scala:358)
        at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:75)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
```

从 main 线程的栈信息中可看出程序的调用顺序：SparkSubmit.main → repl.Main → SparkILoop.process。SparkILoop.process 方法中会调用 initializeSpark 方法，initializeSpark 的实现见代码清单 1-4。

代码清单1-4　initializeSpark的实现

```
def initializeSpark() {
intp.beQuietDuring {
    command("""
        @transient val sc = {
            val _sc = org.apache.spark.repl.Main.interp.createSparkContext()
            println("Spark context available as sc.")
            _sc
        }
        """)
        command("import org.apache.spark.SparkContext._")
    }
}
```

我们看到 initializeSpark 调用了 createSparkContext 方法，createSparkContext 的实现见代码清单 1-5。

代码清单1-5　createSparkContext的实现

```
def createSparkContext(): SparkContext = {
valexecUri = System.getenv("SPARK_EXECUTOR_URI")
valjars = SparkILoop.getAddedJars
valconf = new SparkConf()
    .setMaster(getMaster())
    .setAppName("Spark shell")
    .setJars(jars)
    .set("spark.repl.class.uri", intp.classServer.uri)
if (execUri != null) {
                     conf.set("spark.executor.uri", execUri)
    }
sparkContext = new SparkContext(conf)
    logInfo("Created spark context..")
    sparkContext
}
```

这里最终使用 SparkConf 和 SparkContext 来完成初始化，具体内容将在第 3 章讲解。代码分析中涉及的 repl 主要用于与 Spark 实时交互。

1.3　阅读环境准备

准备 Spark 阅读环境，同样需要一台好机器。笔者调试源码的机器的内存是 8 GB。源码阅读的前提是在 IDE 环境中打包、编译通过。常用的 IDE 有 IntelliJ IDEA、Eclipse。笔者选择用 Eclipse 编译 Spark，原因有二：一是由于使用多年对它比较熟悉，二是社区中使用 Eclipse 编译 Spark 的资料太少，在这里可以做个补充。在 Windows 系统编译 Spark 源码，除了安装 JDK 外，还需要安装以下工具。

（1）安装 Scala

由于 Spark 1.20 版本的 sbt 里指定的 Scala 版本是 2.10.4，具体见 Spark 源码目录下的文件 \project\plugins.sbt，其中有一行：scalaVersion := "2.10.4"。所以选择下载 scala-2.10.4.msi，下载地址：http://www.scala-lang.org/download/。

下载完毕，安装 scala-2.10.4.msi。

（2）安装 SBT

由于 Scala 使用 SBT 作为构建工具，所以需要下载 SBT。下载地址：http://www.scala-sbt.org/，下载最新的安装包 sbt-0.13.8.msi 并安装。

（3）安装 Git Bash

由于 Spark 源码使用 Git 作为版本控制工具，所以需要下载 Git 的客户端工具，推荐使用 Git Bash，因为它更符合 Linux 下的操作习惯。下载地址：http://msysgit.github.io/，下载最新的版本并安装。

（4）安装 Eclipse Scala IDE 插件

Eclipse 通过强大的插件方式支持各种 IDE 工具的集成，要在 Eclipse 中编译、调试、运行 Scala 程序，就需要安装 Eclipse Scala IDE 插件。下载地址：http://scala-ide.org/download/current.html。

由于笔者本地的 Eclipse 版本是 Eclipse 4.4 (Luna)，所以选择安装插件 http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site，如图 1-14 所示。

图 1-14　Eclipse Scala IDE 插件安装地址

在 Eclipse 中选择 Help 菜单，然后选择 Install New Software…选项，打开 Install 对话框，如图 1-15 所示。

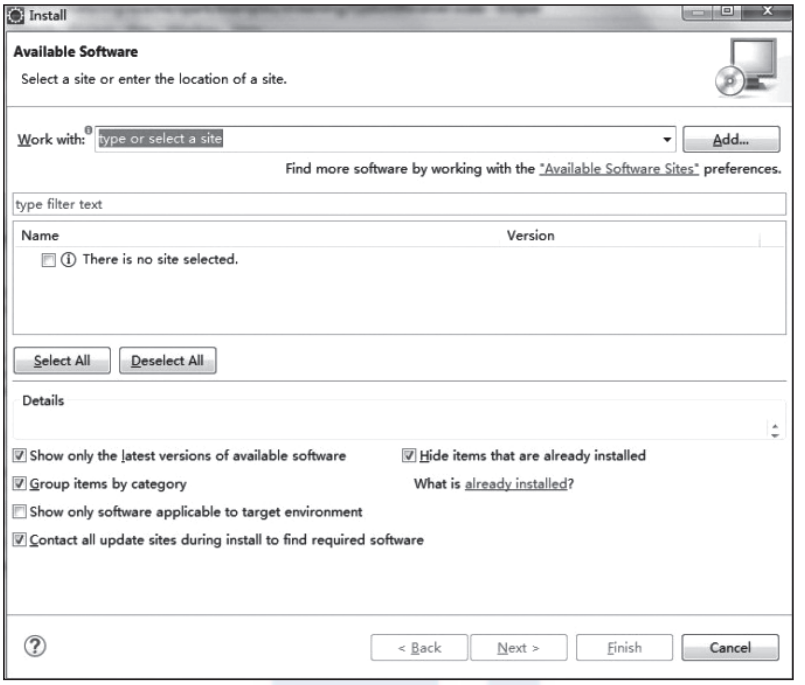

图 1-15　Install 对话框

单击 Add 按钮，打开 Add Repository 对话框，输入插件地址，如图 1-16 所示。

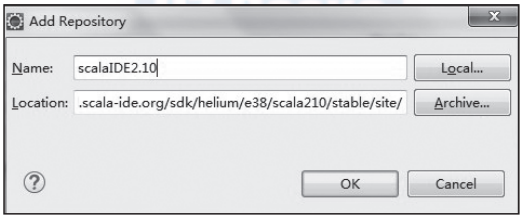

图 1-16　添加 Scala IDE 插件地址

全选插件的内容，完成安装，如图 1-17 所示。

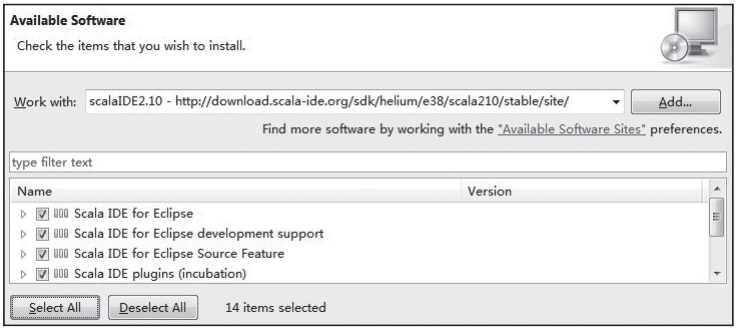

图 1-17　安装 Scala IDE 插件

1.4　Spark 源码编译与调试

1. 下载 Spark 源码

首先，访问 Spark 官网 http://spark.apache.org/，如图 1-18 所示。

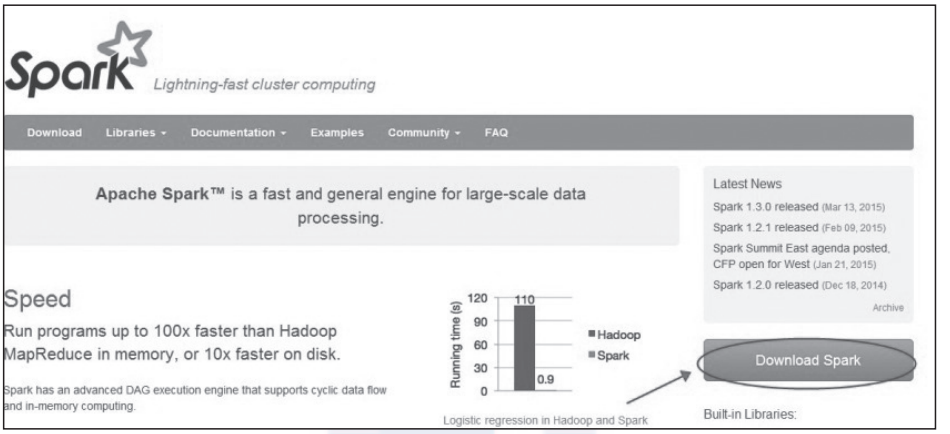

图 1-18　Spark 官网

单击 Download Spark 按钮，在下一个页面找到 git 地址，如图 1-19 所示。

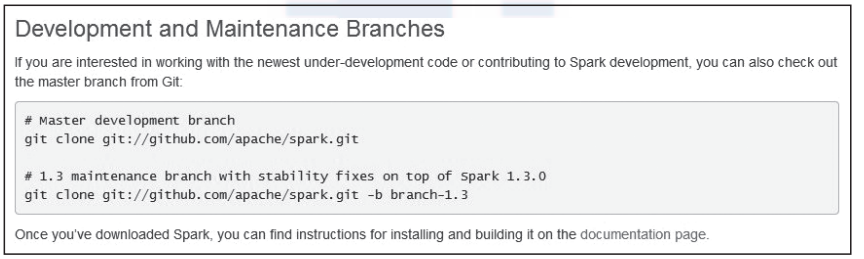

图 1-19　Spark 官方 git 地址

打开 Git Bash 工具，输入 git clone git://github.com/apache/spark.git 命令将源码下载到本地，如图 1-20 所示。

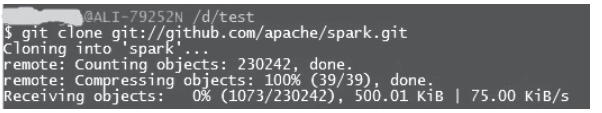

图 1-20　下载 Spark 源码

2. 构建 Scala 应用

使用 cmd 命令行进到 Spark 根目录，执行 sbt 命令。会下载和解析很多 jar 包，要等很长时间，笔者大概花了一个多小时才执行完。

3. 使用 sbt 生成 Eclipse 工程文件

等 sbt 提示符（>）出现后，输入 Eclipse 命令，开始生成 Eclipse 工程文件，也需要花费很长时间，笔者本地大致花了 40 分钟。完成时的状况如图 1-21 所示。

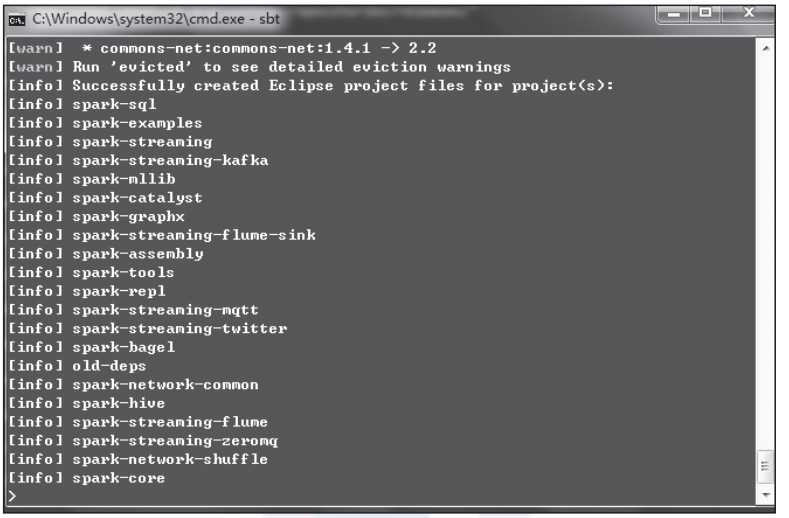

图 1-21　sbt 编译过程

现在我们查看 Spark 下的子文件夹，发现其中都生成了 .project 和 .classpath 文件。比如 mllib 项目下就生成了 .project 和 .classpath 文件，如图 1-22 所示。

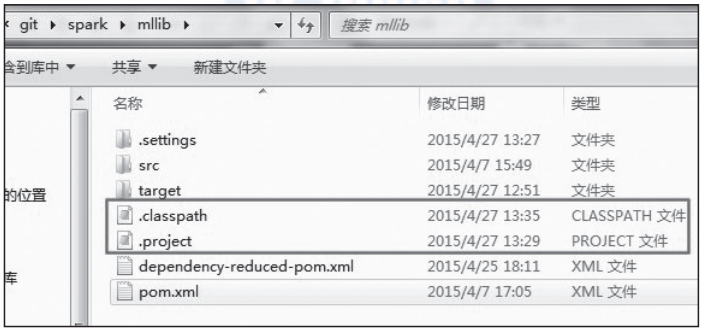

图 1-22　sbt 生成的项目文件

4. 编译 Spark 源码

由于 Spark 使用 Maven 作为项目管理工具，所以需要将 Spark 项目作为 Maven 项目导入 Eclipse 中，如图 1-23 所示。

单击 Next 按钮进入下一个对话框，如图 1-24 所示。

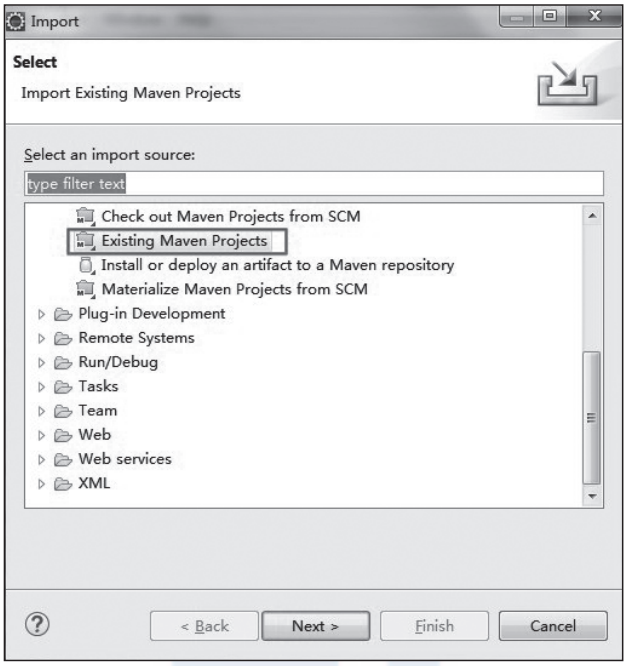

图 1-23　导入 Maven 项目

全选所有项目，单击 Finish 按钮，这样就完成了导入，如图 1-25 所示。

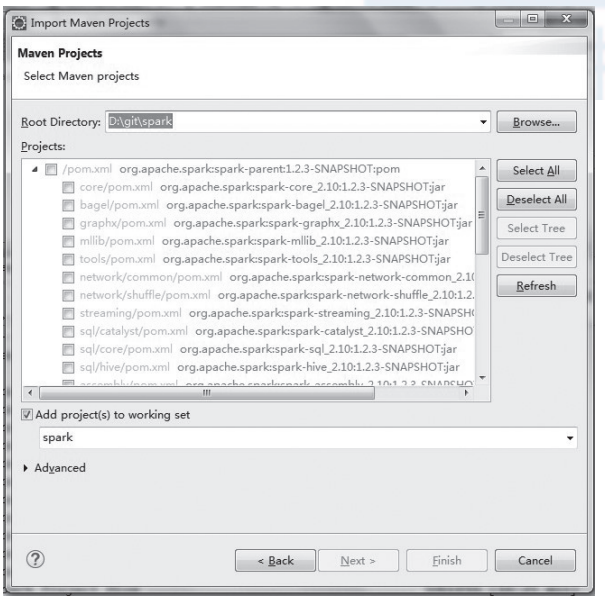

图 1-24　选择 Maven 项目

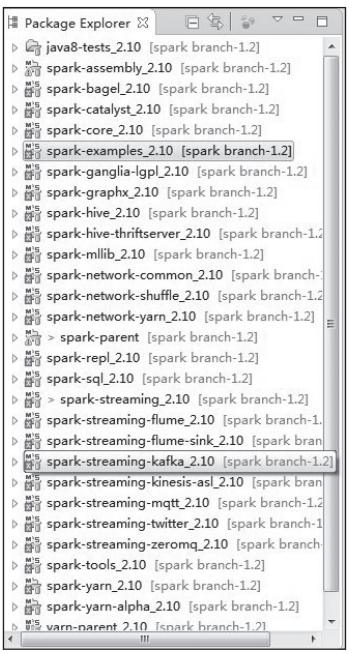

图 1-25　导入完成的项目

导入完成后，需要设置每个子项目的 build path。右击每个项目，选择“Build Path”→“Configure Build Path…”，打开 Java Build Path 界面，如图 1-26 所示。

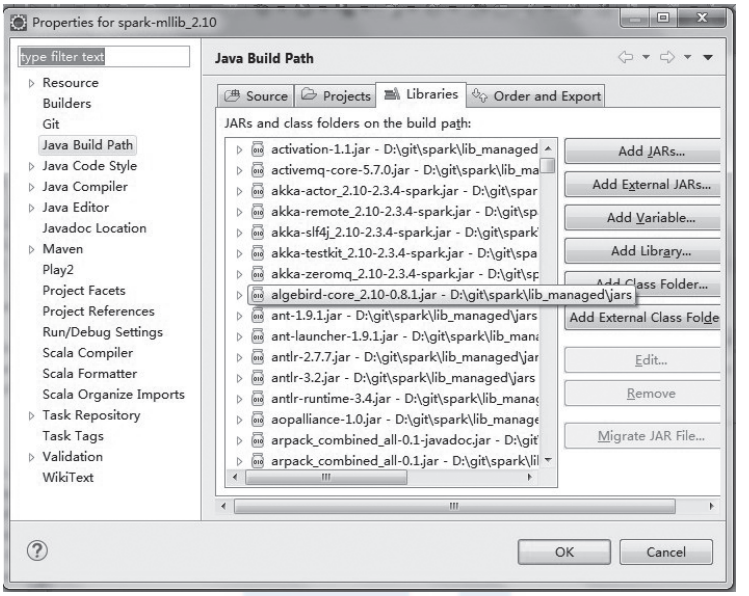

图 1-26　Java 编译目录

单击 Add External JARs 按钮，将 Spark 项目下的 lib_managed 文件夹的子文件夹 bundles 和 jars 内的 jar 包添加进来。

注意　lib_managed/jars 文件夹下有很多打好的 spark 的包，比如：spark-catalyst_2.10-1.3.2-SNAPSHOT.jar。这些 jar 包有可能与你下载的 Spark 源码的版本不一致，导致你在调试源码时，发生 jar 包冲突。所以请将它们排除出去。

Eclipse 在对项目编译时，笔者本地出现了很多错误，有关这些错误的解决建议参见附录 H。所有错误解决后运行 mvn clean install，如图 1-27 所示。

5. 调试 Spark 源码

以 Spark 源码自带的 JavaWordCount 为例，介绍如何调试 Spark 源码。右击 JavaWordCount.java，选择“Debug As”→“Java Application”即可。如果想修改配置参数，右击 JavaWordCount.java，选择“Debug As”→“Debug Configurations…”，从打开的对话框中选择 JavaWordCount，在右侧标签可以修改 Java 执行参数、JRE、classpath、环境变量等配置，如图 1-28 所示。

读者也可以在 Spark 源码中设置断点，进行跟踪调试。

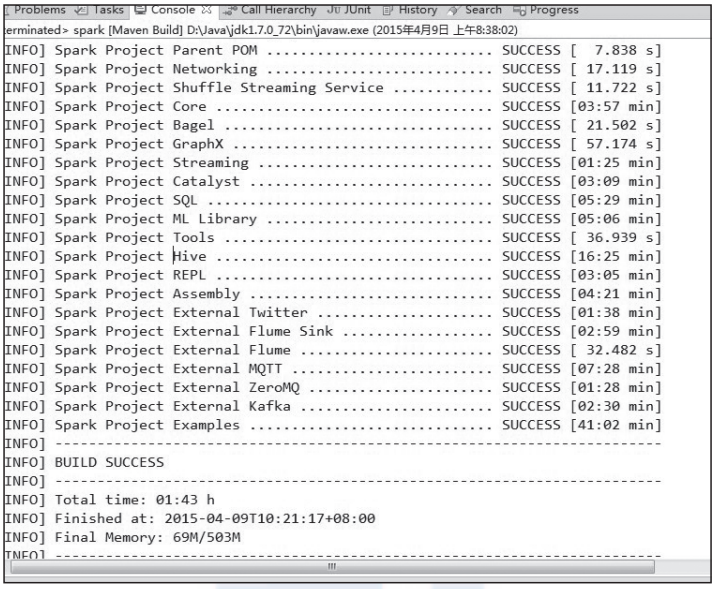

图 1-27　编译成功

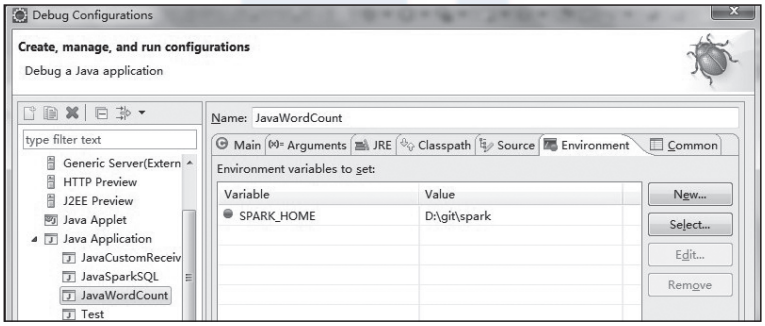

图 1-28　源码调试

1.5　小结

本章通过引导大家在 Linux 操作系统下搭建基本的执行环境，并且介绍 spark-shell 等脚本的执行，来帮助读者由浅入深地进行 Spark 源码的学习。由于目前多数开发工作都在 Windows 系统下进行，并且 Eclipse 有最广大的用户群，即便是一些开始使用 IntelliJ 的用户对 Eclipse 也不陌生，所以在 Windows 环境下搭建源码阅读环境时，选择这些最常用的工具，能降低读者的学习门槛，并且替大家节省时间。

Chapter 2

第 2 章

Spark 设计理念与基本架构

若夫乘天地之正，而御六气之辩，以游无穷者，彼且恶乎待哉？

——《庄子·逍遥游》

本章导读

上一章，介绍了 Spark 环境的搭建，为方便读者学习 Spark 做好准备。本章首先从 Spark 产生的背景开始，介绍 Spark 的主要特点、基本概念、版本变迁。然后简要说明 Spark 的主要模块和编程模型。最后从 Spark 的设计理念和基本架构入手，使读者能够对 Spark 有宏观的认识，为之后的内容做一些准备工作。

Spark 是一个通用的并行计算框架，由加州伯克利大学（UCBerkeley）的 AMP 实验室开发于 2009 年，并于 2010 年开源，2013 年成长为 Apache 旗下大数据领域最活跃的开源项目之一。Spark 也是基于 map reduce 算法模式实现的分布式计算框架，拥有 Hadoop MapReduce 所具有的优点，并且解决了 Hadoop MapReduce 中的诸多缺陷。

2.1 初识 Spark

2.1.1 Hadoop MRv1 的局限

Hadoop1.0 版本采用的是 MRv1 版本的 MapReduce 编程模型。MRv1 版本的实现都封装在 org.apache.hadoop.mapred 包中，MRv1 的 Map 和 Reduce 是通过接口实现的。MRv1 包括三个部分：

- 运行时环境（JobTracker 和 TaskTracker）；
- 编程模型（MapReduce）；

- 数据处理引擎（Map 任务和 Reduce 任务）。

MRv1 存在以下不足：

- **可扩展性差**：在运行时，JobTracker 既负责资源管理又负责任务调度，当集群繁忙时，JobTracker 很容易成为瓶颈，最终导致它的可扩展性问题。
- **可用性差**：采用了单节点的 Master，没有备用 Master 及选举操作，这导致一旦 Master 出现故障，整个集群将不可用。
- **资源利用率低**：TaskTracker 使用 slot 等量划分本节点上的资源量。slot 代表计算资源（CPU、内存等）。一个 Task 获取到一个 slot 后才有机会运行，Hadoop 调度器负责将各个 TaskTracker 上的空闲 slot 分配给 Task 使用。一些 Task 并不能充分利用 slot，而其他 Task 也无法使用这些空闲的资源。slot 分为 Map slot 和 Reduce slot 两种，分别供 MapTask 和 Reduce Task 使用。有时会因为作业刚刚启动等原因导致 MapTask 很多，而 Reduce Task 任务还没有调度的情况，这时 Reduce slot 也会被闲置。
- **不能支持多种 MapReduce 框架**：无法通过可插拔方式将自身的 MapReduce 框架替换为其他实现，如 Spark、Storm 等。

MRv1 的示意如图 2-1 所示。

Apache 为了解决以上问题，对 Hadoop 进行升级改造，MRv2 最终诞生了。MRv2 重用了 MRv1 中的编程模型和数据处理引擎，但是运行时环境被重构了。JobTracker 被拆分成了通用的资源调度平台（ResourceManager，RM）和负责各个计算框架的任务调度模型（ApplicationMaster，AM）。MRv2 中 MapReduce 的核心不再是 MapReduce 框架，而是 YARN。在以 YARN 为核心的 MRv2 中，MapReduce 框架是可插拔的，完全可以替换为其他 MapReduce 实现，比如 Spark、Storm 等。MRv2 的示意如图 2-2 所示。

图 2-1　MRv1 示意图[⊖]

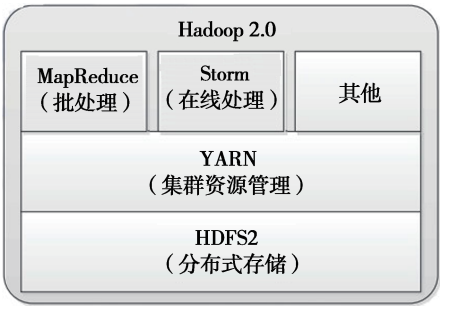

图 2-2　MRv2 示意图

Hadoop MRv2 虽然解决了 MRv1 中的一些问题，但是由于对 HDFS 的频繁操作（包括计算结果持久化、数据备份及 shuffle 等）导致磁盘 I/O 成为系统性能的瓶颈，因此只适用于离

⊖ 图 2-1 和图 2-2 都来源自 http://blog.chinaunix.net/uid-28311809-ud-4383551.html。

线数据处理，而不能提供实时数据处理能力。

2.1.2 Spark 使用场景

Hadoop 常用于解决高吞吐、批量处理的业务场景，例如离线计算结果用于浏览量统计。如果需要实时查看浏览量统计信息，Hadoop 显然不符合这样的要求。Spark 通过内存计算能力极大地提高了大数据处理速度，满足了以上场景的需要。此外，Spark 还支持 SQL 查询、流式计算、图计算、机器学习等。通过对 Java、Python、Scala、R 等语言的支持，极大地方便了用户的使用。

2.1.3 Spark 的特点

Spark 看到 MRv1 的问题，对 MapReduce 做了大量优化，总结如下：

- 快速处理能力。随着实时大数据应用越来越多，Hadoop 作为离线的高吞吐、低响应框架已不能满足这类需求。Hadoop MapReduce 的 Job 将中间输出和结果存储在 HDFS 中，读写 HDFS 造成磁盘 I/O 成为瓶颈。Spark 允许将中间输出和结果存储在内存中，避免了大量的磁盘 I/O。同时 Spark 自身的 DAG 执行引擎也支持数据在内存中的计算。Spark 官网声称性能比 Hadoop 快 100 倍，如图 2-3 所示。即便是内存不足，需要磁盘 I/O，其速度也是 Hadoop 的 10 倍以上。
- 易于使用。Spark 现在支持 Java、Scala、Python 和 R 等语言编写应用程序，大大降低了使用者的门槛。自带了 80 多个高等级操作符，允许在 Scala、Python、R 的 shell 中进行交互式查询。
- 支持查询。Spark 支持 SQL 及 Hive SQL 对数据查询。
- 支持流式计算。与 MapReduce 只能处理离线数据相比，Spark 还支持实时的流计算。Spark 依赖 Spark Streaming 对数据进行实时的处理，其流式处理能力还要强于 Storm。

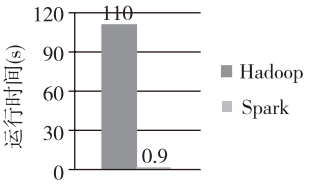

图 2-3 Hadoop 与 Spark 执行逻辑回归时间比较

- 可用性高。Spark 自身实现了 Standalone 部署模式，此模式下的 Master 可以有多个，解决了单点故障问题。此模式完全可以使用其他集群管理器替换，比如 YARN、Mesos、EC2 等。
- 丰富的数据源支持。Spark 除了可以访问操作系统自身的文件系统和 HDFS，还可以访问 Cassandra、HBase、Hive、Tachyon 以及任何 Hadoop 的数据源。这极大地方便了已经使用 HDFS、Hbase 的用户顺利迁移到 Spark。

2.2 Spark 基础知识

1. 版本变迁

经过 4 年多的发展，Spark 目前的版本是 1.4.1。我们简单看看它的版本发展过程。

1）Spark 诞生于 UCBerkeley 的 AMP 实验室（2009）。

2）Spark 正式对外开源（2010 年）。

3）Spark 0.6.0 版本发布（2012-10-15），进行了大范围的性能改进，增加了一些新特性，并对 Standalone 部署模式进行了简化。

4）Spark 0.6.2 版本发布（2013-02-07），解决了一些 bug，并增强了系统的可用性。

5）Spark 0.7.0 版本发布（2013-02-27），增加了更多关键特性，例如，Python API、Spark Streaming 的 alpha 版本等。

6）Spark 0.7.2 版本发布（2013-06-02），性能改进并解决了一些 bug，新增 API 使用的例子。

7）Spark 接受进入 Apache 孵化器（2013-06-21）。

8）Spark 0.7.3 版本发布（2013-07-16），解决一些 bug，更新 Spark Streaming API 等。

9）Spark 0.8.0 版本发布（2013-09-25），一些新功能及可用性改进。

10）Spark 0.8.1 版本发布（2013-12-19），支持 Scala 2.9、YARN 2.2、Standalone 部署模式下调度的高可用性、shuffle 的优化等。

11）Spark 0.9.0 版本发布（2014-02-02），增加了 GraphX，机器学习新特性，流式计算新特性，核心引擎优化（外部聚合、加强对 YARN 的支持）等。

12）Spark 0.9.1 版本发布（2014-04-09），增强使用 YARN 的稳定性，改进 Scala 和 Python API 的奇偶性。

13）Spark 1.0.0 版本发布（2014-05-30），Spark SQL、MLlib、GraphX 和 Spark Streaming 都增加了新特性并进行了优化。Spark 核心引擎还增加了对安全 YARN 集群的支持。

14）Spark 1.0.1 版本发布（2014-07-11），增加了 Spark SQL 的新特性和对 JSON 数据的支持等。

15）Spark 1.0.2 版本发布（2014-08-05），Spark 核心 API 及 Streaming、Python、MLlib 的 bug 修复。

16）Spark 1.1.0 版本发布（2014-09-11）。

17）Spark 1.1.1 版本发布（2014-11-26），Spark 核心 API 及 Streaming、Python、SQL、GraphX 和 MLlib 的 bug 修复。

18）Spark 1.2.0 版本发布（2014-12-18）。

19）Spark 1.2.1 版本发布（2015-02-09），Spark 核心 API 及 Streaming、Python、SQL、GraphX 和 MLlib 的 bug 修复。

20）Spark 1.3.0 版本发布（2015-03-13）。

21）Spark 1.4.0 版本发布（2015-06-11）。

22）Spark 1.4.1 版本发布（2015-07-15），DataFrame API 及 Streaming、Python、SQL 和 MLlib 的 bug 修复。

2. 基本概念

要想对 Spark 有整体性的了解，推荐读者阅读 Matei Zaharia 的 Spark 论文。此处笔者先介

绍 Spark 中的一些概念：

- RDD（resillient distributed dataset）：弹性分布式数据集。
- Task ：具体执行任务。Task 分为 ShuffleMapTask 和 ResultTask 两种。ShuffleMapTask 和 ResultTask 分别类似于 Hadoop 中的 Map 和 Reduce。
- Job：用户提交的作业。一个 Job 可能由一到多个 Task 组成。
- Stage：Job 分成的阶段。一个 Job 可能被划分为一到多个 Stage。
- Partition：数据分区。即一个 RDD 的数据可以划分为多少个分区。
- NarrowDependency ：窄依赖，即子 RDD 依赖于父 RDD 中固定的 Partition。NarrowDependency 分为 OneToOneDependency 和 RangeDependency 两种。
- ShuffleDependency ：shuffle 依赖，也称为宽依赖，即子 RDD 对父 RDD 中的所有 Partition 都有依赖。
- DAG（directed acycle graph）：有向无环图。用于反映各 RDD 之间的依赖关系。

3. Scala 与 Java 的比较

Spark 为什么要选择 Java 作为开发语言？笔者不得而知。如果能对二者进行比较，也许能看出一些端倪。表 2-1 列出了 Scala 与 Java 的比较。

表 2-1 Scala 与 Java 的比较

比项项	Scala	Java
语言类型	面向函数为主，兼有面向对象	面向对象（Java8 也增加了 lambda 函数编程）
简洁性	非常简洁	不简洁
类型推断	丰富的类型推断，例如深度和链式的类型推断、duck type、隐式类型转换等，但也因此增加了编译时长	少量的类型推断
可读性	一般，丰富的语法糖导致的各种奇幻用法，例如方法签名	好
学习成本	较高	一般
语言特性	非常丰富的语法糖和更现代的语言特性，例如 Option、模式匹配、使用空格的方法调用	丰富
并发编程	使用 Actor 的消息模型	使用阻塞、锁、阻塞队列等

通过以上比较似乎仍然无法判断 Spark 选择 Java 作为开发语言的原因。由于函数式编程更接近计算机思维，因此便于通过算法从大数据中建模，这应该更符合 Spark 作为大数据框架的理念吧！

2.3 Spark 基本设计思想

2.3.1 Spark 模块设计

整个 Spark 主要由以下模块组成：

- Spark Core：Spark 的核心功能实现，包括：SparkContext 的初始化（Driver Application 通过 SparkContext 提交）、部署模式、存储体系、任务提交与执行、计算引擎等。
- Spark SQL：提供 SQL 处理能力，便于熟悉关系型数据库操作的工程师进行交互查询。此外，还为熟悉 Hadoop 的用户提供 Hive SQL 处理能力。
- Spark Streaming：提供流式计算处理能力，目前支持 Kafka、Flume、Twitter、MQTT、ZeroMQ、Kinesis 和简单的 TCP 套接字等数据源。此外，还提供窗口操作。
- GraphX：提供图计算处理能力，支持分布式，Pregel 提供的 API 可以解决图计算中的常见问题。
- MLlib：提供机器学习相关的统计、分类、回归等领域的多种算法实现。其一致的 API 接口大大降低了用户的学习成本。

Spark SQL、Spark Streaming、GraphX、MLlib 的能力都是建立在核心引擎之上，如图 2-4 所示。

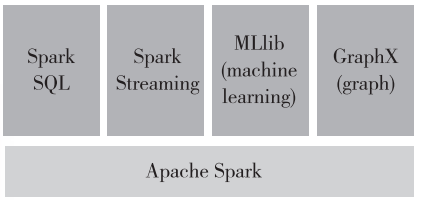

图 2-4　Spark 各模块依赖关系

1. Spark 核心功能

Spark Core 提供 Spark 最基础与最核心的功能，主要包括以下功能。

- SparkContext：通常而言，Driver Application 的执行与输出都是通过 SparkContext 来完成的，在正式提交 Application 之前，首先需要初始化 SparkContext。SparkContext 隐藏了网络通信、分布式部署、消息通信、存储能力、计算能力、缓存、测量系统、文件服务、Web 服务等内容，应用程序开发者只需要使用 SparkContext 提供的 API 完成功能开发。SparkContext 内置的 DAGScheduler 负责创建 Job，将 DAG 中的 RDD 划分到不同的 Stage，提交 Stage 等功能。内置的 TaskScheduler 负责资源的申请、任务的提交及请求集群对任务的调度等工作。
- 存储体系：Spark 优先考虑使用各节点的内存作为存储，当内存不足时才会考虑使用磁盘，这极大地减少了磁盘 I/O，提升了任务执行的效率，使得 Spark 适用于实时计算、流式计算等场景。此外，Spark 还提供了以内存为中心的高容错的分布式文件系统 Tachyon 供用户进行选择。Tachyon 能够为 Spark 提供可靠的内存级的文件共享服务。
- 计算引擎：计算引擎由 SparkContext 中的 DAGScheduler、RDD 以及具体节点上的 Executor 负责执行的 Map 和 Reduce 任务组成。DAGScheduler 和 RDD 虽然位于 SparkContext 内部，但是在任务正式提交与执行之前会将 Job 中的 RDD 组织成有向无关图（简称 DAG），并对 Stage 进行划分，决定了任务执行阶段任务的数量、迭代计算、shuffle 等过程。
- 部署模式：由于单节点不足以提供足够的存储及计算能力，所以作为大数据处理的 Spark 在 SparkContext 的 TaskScheduler 组件中提供了对 Standalone 部署模式的实现和 Yarn、Mesos 等分布式资源管理系统的支持。通过使用 Standalone、Yarn、Mesos 等部署模式为 Task 分配计算资源，提高任务的并发执行效率。除了可用于实际生产环境

的 Standalone、Yarn、Mesos 等部署模式外，Spark 还提供了 Local 模式和 local-cluster 模式便于开发和调试。

2. Spark 扩展功能

为了扩大应用范围，Spark 陆续增加了一些扩展功能，主要包括：

- Spark SQL：SQL 具有普及率高、学习成本低等特点，为了扩大 Spark 的应用面，增加了对 SQL 及 Hive 的支持。Spark SQL 的过程可以总结为：首先使用 SQL 语句解析器（SqlParser）将 SQL 转换为语法树（Tree），并且使用规则执行器（RuleExecutor）将一系列规则（Rule）应用到语法树，最终生成物理执行计划并执行。其中，规则执行器包括语法分析器（Analyzer）和优化器（Optimizer）。Hive 的执行过程与 SQL 类似。
- Spark Streaming：Spark Streaming 与 Apache Storm 类似，也用于流式计算。Spark Streaming 支持 Kafka、Flume、Twitter、MQTT、ZeroMQ、Kinesis 和简单的 TCP 套接字等多种数据输入源。输入流接收器（Receiver）负责接入数据，是接入数据流的接口规范。Dstream 是 Spark Streaming 中所有数据流的抽象，Dstream 可以被组织为 DStream Graph。Dstream 本质上由一系列连续的 RDD 组成。
- GraphX：Spark 提供的分布式图计算框架。GraphX 主要遵循整体同步并行（bulk synchronous parallell，BSP）计算模式下的 Pregel 模型实现。GraphX 提供了对图的抽象 Graph，Graph 由顶点（Vertex）、边（Edge）及继承了 Edge 的 EdgeTriplet（添加了 srcAttr 和 dstAttr 用来保存源顶点和目的顶点的属性）三种结构组成。GraphX 目前已经封装了最短路径、网页排名、连接组件、三角关系统计等算法的实现，用户可以选择使用。
- MLlib：Spark 提供的机器学习框架。机器学习是一门涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多领域的交叉学科。MLlib 目前已经提供了基础统计、分类、回归、决策树、随机森林、朴素贝叶斯、保序回归、协同过滤、聚类、维数缩减、特征提取与转型、频繁模式挖掘、预言模型标记语言、管道等多种数理统计、概率论、数据挖掘方面的数学算法。

2.3.2 Spark 模型设计

1. Spark 编程模型

Spark 应用程序从编写到提交、执行、输出的整个过程如图 2-5 所示，图中描述的步骤如下。

1）用户使用 SparkContext 提供的 API（常用的有 textFile、sequenceFile、runJob、stop 等）编写 Driver application 程序。此外 SQLContext、HiveContext 及 StreamingContext 对 SparkContext 进行封装，并提供了 SQL、Hive 及流式计算相关的 API。

2）使用 SparkContext 提交的用户应用程序，首先会使用 BlockManager 和 BroadcastManager 将任务的 Hadoop 配置进行广播。然后由 DAGScheduler 将任务转换为 RDD 并组织成 DAG，DAG 还将被划分为不同的 Stage。最后由 TaskScheduler 借助 ActorSystem 将任务提交给集群管理器（Cluster Manager）。

3）集群管理器（Cluster Manager）给任务分配资源，即将具体任务分配到 Worker 上，Worker 创建 Executor 来处理任务的运行。Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

2. RDD 计算模型

RDD 可以看做是对各种数据计算模型的统一抽象，Spark 的计算过程主要是 RDD 的迭代计算过程，如图 2-6 所示。RDD 的迭代计算过程非常类似于管道。分区数量取决于 partition 数量的设定，每个分区的数据只会在一个 Task 中计算。所有分区可以在多个机器节点的 Executor 上并行执行。

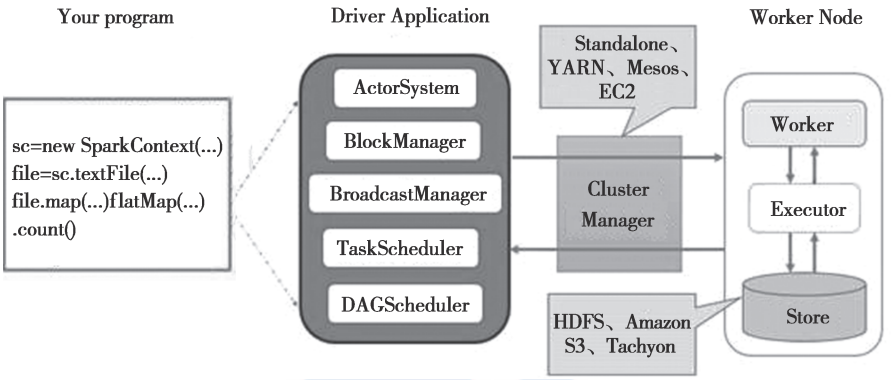

图 2-5　代码执行过程

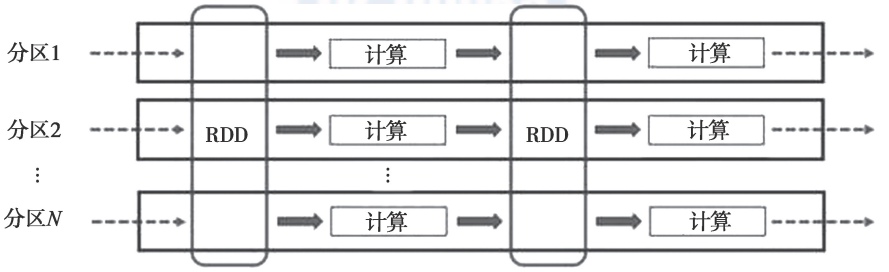

图 2-6　RDD 计算模型

2.4　Spark 基本架构

从集群部署的角度来看，Spark 集群由以下部分组成：

- Cluster Manager：Spark 的集群管理器，主要负责资源的分配与管理。集群管理器分配的资源属于一级分配，它将各个 Worker 上的内存、CPU 等资源分配给应用程序，但是并不负责对 Executor 的资源分配。目前，Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

- Worker：Spark 的工作节点。对 Spark 应用程序来说，由集群管理器分配得到资源的 Worker 节点主要负责以下工作：创建 Executor，将资源和任务进一步分配给 Executor，同步资源信息给 Cluster Manager。
- Executor：执行计算任务的一线进程。主要负责任务的执行以及与 Worker、Driver App 的信息同步。
- Driver App：客户端驱动程序，也可以理解为客户端应用程序，用于将任务程序转换为 RDD 和 DAG，并与 Cluster Manager 进行通信与调度。

这些组成部分之间的整体关系如图 2-7 所示。

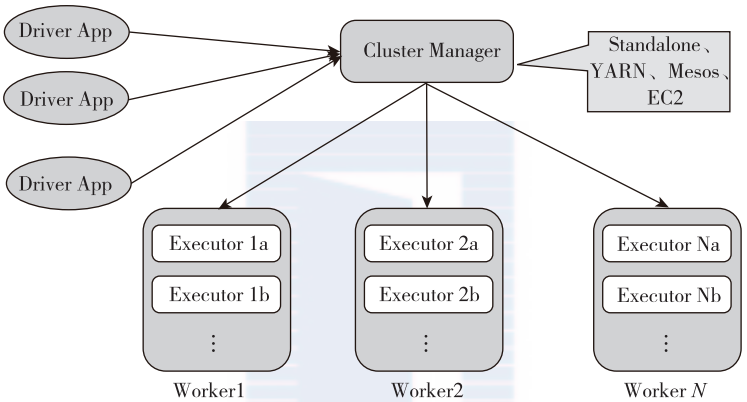

图 2-7 Spark 基本架构图

2.5 小结

每项技术的诞生都会由某种社会需求所驱动，Spark 正是在实时计算的大量需求下诞生的。Spark 借助其优秀的处理能力、可用性高、丰富的数据源支持等特点，在当前大数据领域变得火热，参与的开发者也越来越多。Spark 经过几年的迭代发展，如今已经提供了丰富的功能。笔者相信，Spark 在未来必将产生更耀眼的火花。

核心设计篇

HZ BOOKS
华章图书

- 第 3 章　SparkContext 的初始化
- 第 4 章　存储体系
- 第 5 章　任务提交与执行
- 第 6 章　计算引擎
- 第 7 章　部署模式

Chapter 3

第 3 章

SparkContext 的初始化

道生一，一生二，二生三，三生万物。

——《道德经》

本章导读

SparkContext 的初始化是 Driver 应用程序提交执行的前提，本章内容以 local 模式为主，并按照代码执行顺序讲解，这将有助于首次接触 Spark 的读者理解源码。读者朋友如果能边跟踪代码，边学习本章内容，也许是快速理解 SparkContext 初始化过程的便捷途径。已经熟练使用 Spark 的开发人员可以选择跳过本章内容。

本章将在介绍 SparkContext 初始化过程的同时，向读者介绍各个组件的作用，为阅读后面的章节打好基础。Spark 中的组件很多，就其功能而言涉及网络通信、分布式、消息、存储、计算、缓存、测量、清理、文件服务、Web UI 的方方面面。

3.1 SparkContext 概述

Spark Driver 用于提交用户应用程序，实际可以看作 Spark 的客户端。了解 Spark Driver 的初始化，有助于读者理解用户应用程序在客户端的处理过程。

Spark Driver 的初始化始终围绕着 SparkContext 的初始化。SparkContext 可以算得上是所有 Spark 应用程序的发动机引擎，轿车要想跑起来，发动机首先要启动。SparkContext 初始化完毕，才能向 Spark 集群提交任务。在平坦的公路上，发动机只需以较低的转速、较低的功率就可以游刃有余；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的需求。这些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。

SparkConf 的构造很简单，主要是通过 ConcurrentHashMap 来维护各种 Spark 的配置属性。SparkConf 代码结构见代码清单 3-1。Spark 的配置属性都是以“spark.”开头的字符串。

代码清单3-1　SparkConf代码结构

```
class SparkConf(loadDefaults: Boolean) extends Cloneable with Logging {
    import SparkConf._
    def this() = this(true)
    private val settings = new ConcurrentHashMap[String, String]()
    if (loadDefaults) {
        // 加载任何以spark.开头的系统属性
        for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark.")) {
            set(key, value)
        }
    }
//其余代码省略
```

现在开始介绍 SparkContext。SparkContext 的初始化步骤如下：

1）创建 Spark 执行环境 SparkEnv；

2）创建 RDD 清理器 metadataCleaner；

3）创建并初始化 Spark UI；

4）Hadoop 相关配置及 Executor 环境变量的设置；

5）创建任务调度 TaskScheduler；

6）创建和启动 DAGScheduler；

7）TaskScheduler 的启动；

8）初始化块管理器 BlockManager（BlockManager 是存储体系的主要组件之一，将在第 4 章介绍）；

9）启动测量系统 MetricsSystem；

10）创建和启动 Executor 分配管理器 ExecutorAllocationManager；

11）ContextCleaner 的创建与启动；

12）Spark 环境更新；

13）创建 DAGSchedulerSource 和 BlockManagerSource；

14）将 SparkContext 标记为激活。

SparkContext 的主构造器参数为 SparkConf，其实现如下。

```
class SparkContext(config: SparkConf) extends Logging with ExecutorAllocationClient {
private val creationSite: CallSite = Utils.getCallSite()
    private val allowMultipleContexts: Boolean =
        config.getBoolean("spark.driver.allowMultipleContexts", false)
    SparkContext.markPartiallyConstructed(this, allowMultipleContexts)
```

上面代码中的 CallSite 存储了线程栈中最靠近栈顶的用户类及最靠近栈底的 Scala 或者 Spark 核心类信息。Utils.getCallSite 的详细信息见附录 A。SparkContext 默认只有一个实例（由

属性 spark.driver.allowMultipleContexts 来控制，用户需要多个 SparkContext 实例时，可以将其设置为 true），方法 markPartiallyConstructed 用来确保实例的唯一性，并将当前 SparkContext 标记为正在构建中。

接下来会对 SparkConf 进行复制，然后对各种配置信息进行校验，代码如下。

```
private[spark] val conf = config.clone()
conf.validateSettings()

if (!conf.contains("spark.master")) {
    throw new SparkException("A master URL must be set in your configuration")
}
if (!conf.contains("spark.app.name")) {
    throw new SparkException("An application name must be set in your configuration")
}
```

从上面校验的代码看到必须指定属性 spark.master 和 spark.app.name，否则会抛出异常，结束初始化过程。spark.master 用于设置部署模式，spark.app.name 用于指定应用程序名称。

3.2 创建执行环境 SparkEnv

SparkEnv 是 Spark 的执行环境对象，其中包括众多与 Executor 执行相关的对象。由于在 local 模式下 Driver 会创建 Executor，local-cluster 部署模式或者 Standalone 部署模式下 Worker 另起的 CoarseGrainedExecutorBackend 进程中也会创建 Executor，所以 SparkEnv 存在于 Driver 或者 CoarseGrainedExecutorBackend 进程中。创建 SparkEnv 主要使用 SparkEnv 的 createDriverEnv，SparkEnv.createDriverEnv 方法有三个参数：conf、isLocal 和 listenerBus。

```
val isLocal = (master == "local" || master.startsWith("local["))
private[spark] val listenerBus = new LiveListenerBus
    conf.set("spark.executor.id", "driver")

    private[spark] val env = SparkEnv.createDriverEnv(conf, isLocal, listenerBus)
    SparkEnv.set(env)
```

上面代码中的 conf 是对 SparkConf 的复制，isLocal 标识是否是单机模式，listenerBus 采用监听器模式维护各类事件的处理，在 3.4.1 节会详细介绍。

SparkEnv 的方法 createDriverEnv 最终调用 create 创建 SparkEnv。SparkEnv 的构造步骤如下：

1）创建安全管理器 SecurityManager；

2）创建基于 Akka 的分布式消息系统 ActorSystem；

3）创建 Map 任务输出跟踪器 mapOutputTracker；

4）实例化 ShuffleManager；

5）创建 ShuffleMemoryManager；

6）创建块传输服务 BlockTransferService；

7）创建 BlockManagerMaster；

8）创建块管理器 BlockManager；
9）创建广播管理器 BroadcastManager；
10）创建缓存管理器 CacheManager；
11）创建 HTTP 文件服务器 HttpFileServer；
12）创建测量系统 MetricsSystem；
13）创建 SparkEnv。

3.2.1　安全管理器 SecurityManager

SecurityManager 主要对权限、账号进行设置，如果使用 Hadoop YARN 作为集群管理器，则需要使用证书生成 secret key 登录，最后给当前系统设置默认的口令认证实例，此实例采用匿名内部类实现，参见代码清单 3-2。

代码清单3-2　SecurityManager的实现

```
private val secretKey = generateSecretKey()

// 使用HTTP连接设置口令认证
if (authOn) {
    Authenticator.setDefault(
        new Authenticator() {
            override def getPasswordAuthentication(): PasswordAuthentication = {
                var passAuth: PasswordAuthentication = null
            val userInfo = getRequestingURL().getUserInfo()
            if (userInfo != null) {
                val  parts = userInfo.split(":", 2)
                passAuth  =  new  PasswordAuthentication(parts(0),  parts(1).
                    toCharArray())
                }
                return passAuth
            }
        }
    )
}
```

3.2.2　基于 Akka 的分布式消息系统 ActorSystem

ActorSystem 是 Spark 中最基础的设施，Spark 既使用它发送分布式消息，又用它实现并发编程。消息系统可以实现并发？要解释清楚这个问题，首先应该简单介绍下 Scala 语言的 Actor 并发编程模型：Scala 认为 Java 线程通过共享数据以及通过锁来维护共享数据的一致性是糟糕的做法，容易引起锁的争用，降低并发程序的性能，甚至会引入死锁的问题。在 Scala 中只需要自定义类型继承 Actor，并且提供 act 方法，就如同 Java 里实现 Runnable 接口，需要实现 run 方法一样。但是不能直接调用 act 方法，而是通过发送消息的方式 (Scala 发送消息是异步的) 传递数据。如：

```
Actor ! message
```

Akka 是 Actor 编程模型的高级类库，类似于 JDK 1.5 之后越来越丰富的并发工具包，简化了程序员并发编程的难度。ActorSystem 便是 Akka 提供的用于创建分布式消息通信系统的基础类。Akka 的具体信息见附录 B。

正是因为 Actor 轻量级的并发编程、消息发送以及 ActorSystem 支持分布式消息发送等特点，Spark 选择了 ActorSystem。

SparkEnv 中创建 ActorSystem 时用到了 AkkaUtils 工具类，见代码清单 3-3。AkkaUtils.createActorSystem 方法用于启动 ActorSystem，见代码清单 3-4。AkkaUtils 使用了 Utils 的静态方法 startServiceOnPort，startServiceOnPort 最终会回调方法 startService: Int => (T, Int)，此处的 startService 实际是方法 doCreateActorSystem。真正启动 ActorSystem 是由 doCreateActorSystem 方法完成的，doCreateActorSystem 的具体实现细节请见附录 B。Spark 的 Driver 中 Akka 的默认访问地址是 akka://sparkDriver，Spark 的 Executor 中 Akka 的默认访问地址是 akka:// sparkExecutor。如果不指定 ActorSystem 的端口，那么所有节点的 ActorSystem 端口在每次启动时随机产生。关于 startServiceOnPort 的实现，请见附录 A。

代码清单3-3　AkkaUtils工具类创建和启动ActorSystem

```
val (actorSystem, boundPort) =
    Option(defaultActorSystem) match {
        case Some(as) => (as, port)
        case None =>
            val actorSystemName = if (isDriver) driverActorSystemName else
                 executorActorSystemName
            AkkaUtils.createActorSystem(actorSystemName, hostname, port, conf,
                securityManager)
}
```

代码清单3-4　ActorSystem的创建和启动

```
def createActorSystem(
        name: String,
        host: String,
        port: Int,
        conf: SparkConf,
        securityManager: SecurityManager): (ActorSystem, Int) = {
    val startService: Int => (ActorSystem, Int) = { actualPort =>
        doCreateActorSystem(name, host, actualPort, conf, securityManager)
    }
    Utils.startServiceOnPort(port, startService, conf, name)
}
```

3.2.3　map 任务输出跟踪器 mapOutputTracker

mapOutputTracker 用于跟踪 map 阶段任务的输出状态，此状态便于 reduce 阶段任务获取

地址及中间输出结果。每个 map 任务或者 reduce 任务都会有其唯一标识，分别为 mapId 和 reduceId。每个 reduce 任务的输入可能是多个 map 任务的输出，reduce 会到各个 map 任务的所在节点上拉取 Block，这一过程叫做 shuffle。每批 shuffle 过程都有唯一的标识 shuffleId。

这里先介绍下 MapOutputTrackerMaster。MapOutputTrackerMaster 内部使用 mapStatuses：TimeStampedHashMap[Int, Array[MapStatus]] 来维护跟踪各个 map 任务的输出状态。其中 key 对应 shuffleId，Array 存储各个 map 任务对应的状态信息 MapStatus。由于 MapStatus 维护了 map 输出 Block 的地址 BlockManagerId，所以 reduce 任务知道从何处获取 map 任务的中间输出。MapOutputTrackerMaster 还使用 cachedSerializedStatuses：TimeStampedHashMap[Int, Array[Byte]] 维护序列化后的各个 map 任务的输出状态。其中 key 对应 shuffleId，Array 存储各个序列化 MapStatus 生成的字节数组。

Driver 和 Executor 处理 MapOutputTrackerMaster 的方式有所不同。

- 如果当前应用程序是 Driver，则创建 MapOutputTrackerMaster，然后创建 MapOutputTrackerMasterActor，并且注册到 ActorSystem 中。
- 如果当前应用程序是 Executor，则创建 MapOutputTrackerWorker，并从 ActorSystem 中找到 MapOutputTrackerMasterActor。

无论是 Driver 还是 Executor，最后都由 mapOutputTracker 的属性 trackerActor 持有 MapOutputTrackerMasterActor 的引用，参见代码清单 3-5。

代码清单3-5　registerOrLookup方法用于查找或者注册Actor的实现

```
def registerOrLookup(name: String, newActor: => Actor): ActorRef = {
    if (isDriver) {
        logInfo("Registering " + name)
        actorSystem.actorOf(Props(newActor), name = name)
    } else {
        AkkaUtils.makeDriverRef(name, conf, actorSystem)
    }
    }

    val mapOutputTracker =  if (isDriver) {
        new MapOutputTrackerMaster(conf)
    } else {
        new MapOutputTrackerWorker(conf)
}

    mapOutputTracker.trackerActor = registerOrLookup(
        "MapOutputTracker",
    new MapOutputTrackerMasterActor(mapOutputTracker.asInstanceOf[MapOutputTrack
         erMaster], conf))
```

在后面章节大家会知道 map 任务的状态正是由 Executor 向持有的 MapOutputTrackerMasterActor 发送消息，将 map 任务状态同步到 mapOutputTracker 的 mapStatuses 和 cachedSerializedStatuses 的。Executor 究竟是如何找到 MapOutputTrackerMasterActor 的？registerOrLookup

方法通过调用 AkkaUtils.makeDriverRef 找到 MapOutputTrackerMasterActor，实际正是利用 ActorSystem 提供的分布式消息机制实现的，具体细节参见附录 B。这里第一次使用到了 Akka 提供的功能，以后大家会渐渐感觉到使用 Akka 的便捷。

3.2.4 实例化 ShuffleManager

ShuffleManager 负责管理本地及远程的 block 数据的 shuffle 操作。ShuffleManager 默认为通过反射方式生成的 SortShuffleManager 的实例，可以修改属性 spark.shuffle.manager 为 hash 来显式控制使用 HashShuffleManager。SortShuffleManager 通过持有的 IndexShuffleBlockManager 间接操作 BlockManager 中的 DiskBlockManager 将 map 结果写入本地，并根据 shuffleId、mapId 写入索引文件，也能通过 MapOutputTrackerMaster 中维护的 mapStatuses 从本地或者其他远程节点读取文件。有读者可能会问，为什么需要 shuffle？Spark 作为并行计算框架，同一个作业会被划分为多个任务在多个节点上并行执行，reduce 的输入可能存在于多个节点上，因此需要通过“洗牌”将所有 reduce 的输入汇总起来，这个过程就是 shuffle。这个问题以及对 ShuffleManager 的具体使用会在第 5 章和第 6 章详述。ShuffleManager 的实例化见代码清单 3-6。代码清单 3-6 最后创建的 ShuffleMemoryManager 将在 3.2.5 节介绍。

代码清单3-6 ShuffleManager的实例化及ShuffleMemoryManager的创建

```
    val shortShuffleMgrNames = Map(
        "hash" -> "org.apache.spark.shuffle.hash.HashShuffleManager",
        "sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")
    val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
    val shuffleMgrClass = shortShuffleMgrNames.get
OrElse(shuffleMgrName.toLowerCase, shuffleMgrName)
    val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)

    val shuffleMemoryManager = new ShuffleMemoryManager(conf)
```

3.2.5 shuffle 线程内存管理器 ShuffleMemoryManager

ShuffleMemoryManager 负责管理 shuffle 线程占有内存的分配与释放，并通过 threadMemory：mutable.HashMap[Long, Long] 缓存每个线程的内存字节数，见代码清单 3-7。

代码清单3-7 ShuffleMemoryManager的数据结构

```
private[spark] class ShuffleMemoryManager(maxMemory: Long) extends Logging {
    private val threadMemory = new mutable.HashMap[Long, Long]()  // threadId ->
        memory bytes
    def this(conf: SparkConf) = this(ShuffleMemoryManager.getMaxMemory(conf))
```

getMaxMemory 方法用于获取 shuffle 所有线程占用的最大内存，实现如下。

```
def getMaxMemory(conf: SparkConf): Long = {
```

```
    val memoryFraction = conf.getDouble("spark.shuffle.memoryFraction", 0.2)
    val safetyFraction = conf.getDouble("spark.shuffle.safetyFraction", 0.8)
    (Runtime.getRuntime.maxMemory * memoryFraction * safetyFraction).toLong
}
```

从上面代码可以看出，shuffle 所有线程占用的最大内存的计算公式为：

Java 运行时最大内存 * Spark 的 shuffle 最大内存占比 * Spark 的安全内存占比

可以配置属性 spark.shuffle.memoryFraction 修改 Spark 的 shuffle 最大内存占比，配置属性 spark.shuffle.safetyFraction 修改 Spark 的安全内存占比。

注意　ShuffleMemoryManager 通常运行在 Executor 中，Driver 中的 ShuffleMemoryManager 只有在 local 模式下才起作用。

3.2.6　块传输服务 BlockTransferService

BlockTransferService 默认为 NettyBlockTransferService（可以配置属性 spark.shuffle.blockTransferService 使用 NioBlockTransferService），它使用 Netty 提供的异步事件驱动的网络应用框架，提供 web 服务及客户端，获取远程节点上 Block 的集合。

```
val blockTransferService =
    conf.get("spark.shuffle.blockTransferService", "netty").toLowerCase match {
        case "netty" =>
            new NettyBlockTransferService(conf, securityManager, numUsableCores)
        case "nio" =>
            new NioBlockTransferService(conf, securityManager)
    }
```

NettyBlockTransferService 的具体实现将在第 4 章详细介绍。这里大家可能觉得奇怪，这样的网络应用为何也要放在存储体系？大家不妨先带着疑问，直到你真正了解了存储体系。

3.2.7　BlockManagerMaster 介绍

BlockManagerMaster 负责对 Block 的管理和协调，具体操作依赖于 BlockManagerMasterActor。Driver 和 Executor 处理 BlockManagerMaster 的方式不同：

- 如果当前应用程序是 Driver，则创建 BlockManagerMasterActor，并且注册到 ActorSystem 中。
- 如果当前应用程序是 Executor，则从 ActorSystem 中找到 BlockManagerMasterActor。

无论是 Driver 还是 Executor，最后 BlockManagerMaster 的属性 driverActor 将持有对 BlockManagerMasterActor 的引用。BlockManagerMaster 的创建代码如下。

```
val blockManagerMaster = new BlockManagerMaster(registerOrLookup(
    "BlockManagerMaster",
    new BlockManagerMasterActor(isLocal, conf, listenerBus)), conf, isDriver)
```

registerOrLookup 已在 3.2.3 节介绍过了，不再赘述。BlockManagerMaster 及 BlockManager-MasterActor 的具体实现将在第 4 章详细介绍。

3.2.8 创建块管理器 BlockManager

BlockManager 负责对 Block 的管理，只有在 BlockManager 的初始化方法 initialize 被调用后，它才是有效的。BlockManager 作为存储系统的一部分，具体实现见第 4 章。BlockManager 的创建代码如下。

```
val blockManager = new BlockManager(executorId, actorSystem, blockManagerMaster,
    serializer, conf, mapOutputTracker, shuffleManager, blockTransferService,
        securityManager,numUsableCores)
```

3.2.9 创建广播管理器 BroadcastManager

BroadcastManager 用于将配置信息和序列化后的 RDD、Job 以及 ShuffleDependency 等信息在本地存储。如果为了容灾，也会复制到其他节点上。创建 BroadcastManager 的代码实现如下。

```
val broadcastManager = new BroadcastManager(isDriver, conf, securityManager)
```

BroadcastManager 必须在其初始化方法 initialize 被调用后，才能生效。initialize 方法实际利用反射生成广播工厂实例 broadcastFactory（可以配置属性 spark.broadcast.factory 指定，默认为 org.apache.spark.broadcast.TorrentBroadcastFactory）。BroadcastManager 的广播方法 newBroadcast 实际代理了工厂 broadcastFactory 的 newBroadcast 方法来生成广播对象。unbroadcast 方法实际代理了工厂 broadcastFactory 的 unbroadcast 方法生成非广播对象。BroadcastManager 的 initialize、unbroadcast 及 newBroadcast 方法见代码清单 3-8。

代码清单3-8 BroadcastManager的实现

```
private def initialize() {
    synchronized {
        if (!initialized) {
            val broadcastFactoryClass = conf.get("spark.broadcast.factory", "org.
                 apache.spark.broadcast.TorrentBroadcastFactory")
            broadcastFactory =
                Class.forName(broadcastFactoryClass).newInstance.asInstanceOf
                    [BroadcastFactory]
            broadcastFactory.initialize(isDriver, conf, securityManager)
            initialized = true
        }
    }
    }

    private val nextBroadcastId = new AtomicLong(0)

    def newBroadcast[T: ClassTag](value_ : T, isLocal: Boolean) = {
        broadcastFactory.newBroadcast[T](value_, isLocal, nextBroadcastId.
            getAndIncrement())
    }
```

```
    def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean) {
        broadcastFactory.unbroadcast(id, removeFromDriver, blocking)
    }
}
```

3.2.10　创建缓存管理器 CacheManager

CacheManager 用于缓存 RDD 某个分区计算后的中间结果，缓存计算结果发生在迭代计算的时候，将在 6.1 节讲到。而 CacheManager 将在 4.10 节详细描述。创建 CacheManager 的代码如下。

```
val cacheManager = new CacheManager(blockManager)
```

3.2.11　HTTP 文件服务器 HttpFileServer

HttpFileServer 的创建参见代码清单 3-9。HttpFileServer 主要提供对 jar 及其他文件的 http 访问，这些 jar 包包括用户上传的 jar 包。端口由属性 spark.fileserver.port 配置，默认为 0，表示随机生成端口号。

代码清单3-9　HttpFileServer的创建

```
val httpFileServer =
    if (isDriver) {
        val fileServerPort = conf.getInt("spark.fileserver.port", 0)
        val server = new HttpFileServer(conf, securityManager, fileServerPort)
        server.initialize()
        conf.set("spark.fileserver.uri",  server.serverUri)
        server
    } else {
        null
    }
```

HttpFileServer 的初始化过程见代码清单 3-10，主要包括以下步骤：

1）使用 Utils 工具类创建文件服务器的根目录及临时目录（临时目录在运行时环境关闭时会删除）。Utils 工具的详细介绍见附录 A。

2）创建存放 jar 包及其他文件的文件目录。

3）创建并启动 HTTP 服务。

代码清单3-10　HttpFileServer的初始化

```
def initialize() {
    baseDir = Utils.createTempDir(Utils.getLocalDir(conf), "httpd")
    fileDir = new File(baseDir, "files")
    jarDir = new File(baseDir, "jars")
    fileDir.mkdir()
    jarDir.mkdir()
```

```
    logInfo("HTTP File server directory is " + baseDir)
    httpServer = new HttpServer(conf, baseDir, securityManager, requestedPort,
        "HTTP file server")
    httpServer.start()
    serverUri = httpServer.uri
    logDebug("HTTP file server started at: " + serverUri)
}
```

HttpServer 的构造和 start 方法的实现中，再次使用了 Utils 的静态方法 startServiceOnPort，因此会回调 doStart 方法，见代码清单 3-11。有关 Jetty 的 API 使用参见附录 C。

代码清单3-11　HttpServer的启动

```
def start() {
    if (server != null) {
        throw new ServerStateException("Server is already started")
    } else {
        logInfo("Starting HTTP Server")
        val (actualServer, actualPort) =
            Utils.startServiceOnPort[Server](requestedPort, doStart, conf,
                serverName)
        server = actualServer
        port = actualPort
    }
}
```

doStart 方法中启动内嵌的 Jetty 所提供的 HTTP 服务，见代码清单 3-12。

代码清单3-12　HttpServer的启动功能实现

```
private def doStart(startPort: Int): (Server, Int) = {
    val server = new Server()
    val connector = new SocketConnector
    connector.setMaxIdleTime(60 * 1000)
    connector.setSoLingerTime(-1)
    connector.setPort(startPort)
    server.addConnector(connector)

    val threadPool = new QueuedThreadPool
    threadPool.setDaemon(true)
    server.setThreadPool(threadPool)
    val resHandler = new ResourceHandler
    resHandler.setResourceBase(resourceBase.getAbsolutePath)

    val handlerList = new HandlerList
    handlerList.setHandlers(Array(resHandler, new DefaultHandler))

    if (securityManager.isAuthenticationEnabled()) {
        logDebug("HttpServer is using security")
        val sh = setupSecurityHandler(securityManager)
        // make sure we go through security handler to get resources
        sh.setHandler(handlerList)
```

```
        server.setHandler(sh)
    } else {
        logDebug("HttpServer is not using security")
        server.setHandler(handlerList)
    }

    server.start()
    val actualPort = server.getConnectors()(0).getLocalPort

    (server, actualPort)
}
```

3.2.12　创建测量系统 MetricsSystem

MetricsSystem 是 Spark 的测量系统，创建 MetricsSystem 的代码如下。

```
val metricsSystem = if (isDriver) {
        MetricsSystem.createMetricsSystem("driver", conf, securityManager)
    } else {
        conf.set("spark.executor.id", executorId)
        val ms = MetricsSystem.createMetricsSystem("executor", conf,
             securityManager)
        ms.start()
        ms
    }
```

上面调用的 createMetricsSystem 方法实际创建了 MetricsSystem，代码如下。

```
def createMetricsSystem(
    instance: String, conf: SparkConf, securityMgr: SecurityManager):
        MetricsSystem = {
    new MetricsSystem(instance, conf, securityMgr)
}
```

构造 MetricsSystem 的过程最重要的是调用了 MetricsConfig 的 initialize 方法，见代码清单 3-13。

代码清单3-13　MetricsConfig的初始化

```
def initialize() {
    setDefaultProperties(properties)

    var is: InputStream = null
    try {
        is = configFile match {
            case Some(f) => new FileInputStream(f)
            case None => Utils.getSparkClassLoader.getResourceAsStream(METRICS_CONF)
        }

        if (is != null) {
            properties.load(is)
        }
```

```
    } catch {
        case e: Exception => logError("Error loading configure file", e)
    } finally {
        if (is != null) is.close()
    }

    propertyCategories = subProperties(properties, INSTANCE_REGEX)
    if (propertyCategories.contains(DEFAULT_PREFIX)) {
        import scala.collection.JavaConversions._

        val defaultProperty = propertyCategories(DEFAULT_PREFIX)
        for { (inst, prop) <- propertyCategories
            if (inst != DEFAULT_PREFIX)
            (k, v) <- defaultProperty
            if (prop.getProperty(k) == null) } {
        prop.setProperty(k, v)
        }
    }
}
```

从以上实现可以看出，MetricsConfig 的 initialize 方法主要负责加载 metrics.properties 文件中的属性配置，并对属性进行初始化转换。

例如，将属性

```
{*.sink.servlet.path=/metrics/json, applications.sink.servlet.path=/metrics/
    applications/json, *.sink.servlet.class=org.apache.spark.metrics.sink.
    MetricsServlet, master.sink.servlet.path=/metrics/master/json}
```

转换为

```
Map(applications -> {sink.servlet.class=org.apache.spark.metrics.sink.
    MetricsServlet, sink.servlet.path=/metrics/applications/json}, master ->
    {sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet, sink.
    servlet.path=/metrics/master/json}, * -> {sink.servlet.class=org.apache.
    spark.metrics.sink.MetricsServlet, sink.servlet.path=/metrics/json})
```

3.2.13 创建 SparkEnv

当所有的基础组件准备好后，最终使用下面的代码创建执行环境 SparkEnv。

```
new SparkEnv(executorId, actorSystem, serializer, closureSerializer, cacheManager,
        mapOutputTracker, shuffleManager, broadcastManager, blockTransferService,
    blockManager, securityManager, httpFileServer, sparkFilesDir,
metricsSystem, shuffleMemoryManager, conf)
```

注意 serializer 和 closureSerializer 都是使用 Class.forName 反射生成的 org.apache.spark.serializer.JavaSerializer 类的实例，其中 closureSerializer 实例特别用来对 Scala 中的闭包进行序列化。

3.3　创建 metadataCleaner

SparkContext 为了保持对所有持久化的 RDD 的跟踪，使用类型是 TimeStamped-WeakValueHashMap 的 persistentRdds 缓存。metadataCleaner 的功能是清除过期的持久化 RDD。创建 metadataCleaner 的代码如下。

```
private[spark] val persistentRdds = new TimeStampedWeakValueHashMap[Int, RDD[_]]
private[spark] val metadataCleaner =
    new MetadataCleaner(MetadataCleanerType.SPARK_CONTEXT, this.cleanup, conf)
```

我们仔细看看 MetadataCleaner 的实现，见代码清单 3-14。

代码清单3-14　MetadataCleaner的实现

```
private[spark] class MetadataCleaner(
        cleanerType: MetadataCleanerType.MetadataCleanerType,
        cleanupFunc: (Long) => Unit,
        conf: SparkConf)
    extends Logging
{
    val name = cleanerType.toString

    private val delaySeconds = MetadataCleaner.getDelaySeconds(conf, cleanerType)
    private val periodSeconds = math.max(10, delaySeconds / 10)
    private val timer = new Timer(name + " cleanup timer", true)

    private val task = new TimerTask {
        override def run() {
        try {
            cleanupFunc(System.currentTimeMillis() - (delaySeconds * 1000))
            logInfo("Ran metadata cleaner for " + name)
        } catch {
            case e: Exception => logError("Error running cleanup task for " + name, e)
        }
      }
    }

    if (delaySeconds > 0) {
        timer.schedule(task, delaySeconds * 1000, periodSeconds * 1000)
    }

    def cancel() {
        timer.cancel()
    }
}
```

从 MetadataCleaner 的实现可以看出其实质是一个用 TimerTask 实现的定时器，不断调用 cleanupFunc: (Long) => Unit 这样的函数参数。构造 metadataCleaner 时的函数参数是 cleanup，用于清理 persistentRdds 中的过期内容，代码如下。

```
private[spark] def cleanup(cleanupTime: Long) {
    persistentRdds.clearOldValues(cleanupTime)
}
```

3.4 SparkUI 详解

任何系统都需要提供监控功能，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。SparkUI 就是这样的服务，它的架构如图 3-1 所示。

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，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。

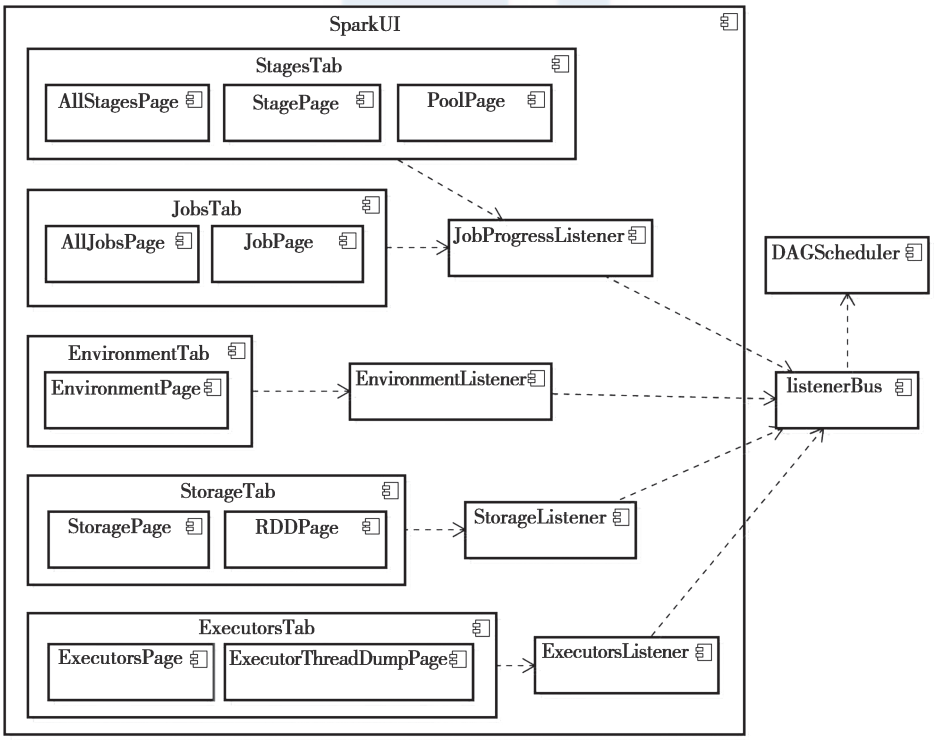

图 3-1 SparkUI 架构

我们先简单介绍图 3-1 中的各个组件：DAGScheduler 是主要的产生各类 SparkListenerEvent 的源头，它将各种 SparkListenerEvent 发送到 listenerBus 的事件队列中，listenerBus 通过定时器将 SparkListenerEvent 事件匹配到具体的 SparkListener，改变 SparkListener 中的统计监控数据，最终由 SparkUI 的界面展示。从图 3-1 中还可以看到 Spark 里定义了很多监听器 SparkListener 的实现，包括 JobProgressListener、EnvironmentListener、StorageListener、ExecutorsListener，它们的类继承体系如图 3-2 所示。

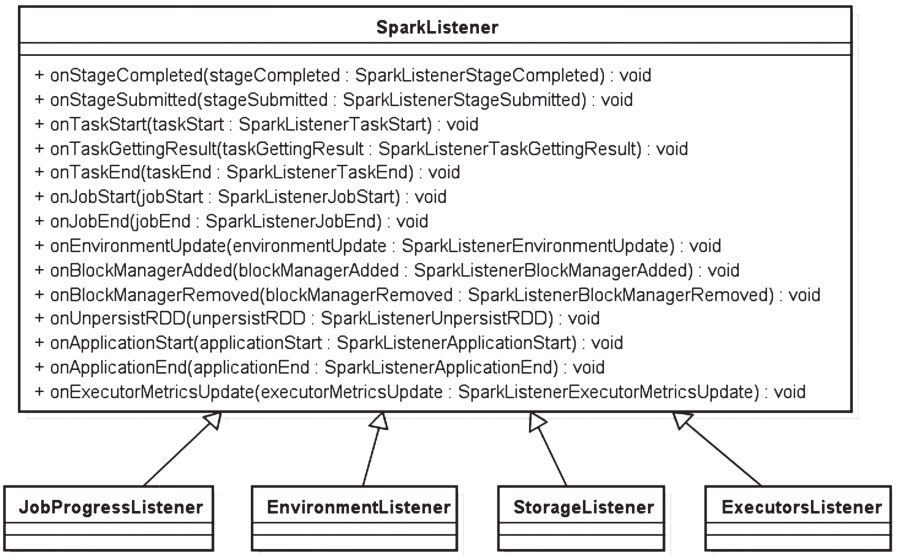

图 3-2　SparkListener 的类继承体系

华章图书

3.4.1　listenerBus 详解

listenerBus 的类型是 LiveListenerBus。LiveListenerBus 实现了监听器模型，通过监听事件触发对各种监听器监听状态信息的修改，达到 UI 界面的数据刷新效果。LiveListenerBus 由以下部分组成：

- 事件阻塞队列：类型为 LinkedBlockingQueue[SparkListenerEvent]，固定大小是 10 000；
- 监听器数组：类型为 ArrayBuffer[SparkListener]，存放各类监听器 SparkListener。
- 事件匹配监听器的线程：此 Thread 不断拉取 LinkedBlockingQueue 中的事件，遍历监听器，调用监听器的方法。任何事件都会在 LinkedBlockingQueue 中存在一段时间，然后 Thread 处理了此事件后，会将其清除。因此使用 listenerBus 这个名字再合适不过了，到站就下车。listenerBus 的实现见代码清单 3-15。

代码清单3-15　LiveListenerBus的事件处理实现

```
private val EVENT_QUEUE_CAPACITY = 10000
    private val eventQueue = new LinkedBlockingQueue[SparkListenerEvent](EVENT_
        QUEUE_CAPACITY)
```

```
    private var queueFullErrorMessageLogged = false
    private var started = false
    // A counter that represents the number of events produced and consumed in
          the queue
    private val eventLock = new Semaphore(0)

    private val listenerThread = new Thread("SparkListenerBus") {
      setDaemon(true)
      override def run(): Unit = Utils.logUncaughtExceptions {
        while (true) {
            eventLock.acquire()
            // Atomically remove and process this event
            LiveListenerBus.this.synchronized {
                val event = eventQueue.poll
                if (event == SparkListenerShutdown) {
                    // Get out of the while loop and shutdown the daemon thread
                    return
                }
                Option(event).foreach(postToAll)
            }
        }
    }
}

def start() {
    if (started) {
        throw new IllegalStateException("Listener bus already started!")
    }
    listenerThread.start()
    started = true
    }
def post(event: SparkListenerEvent) {
    val eventAdded = eventQueue.offer(event)
    if (eventAdded) {
        eventLock.release()
    } else {
        logQueueFullErrorMessage()
    }
}

def listenerThreadIsAlive: Boolean = synchronized { listenerThread.isAlive }

def queueIsEmpty: Boolean = synchronized { eventQueue.isEmpty }

def stop() {
   if (!started) {
        throw new IllegalStateException("Attempted to stop a listener bus that
            has not yet started!")
    }
    post(SparkListenerShutdown)
    listenerThread.join()
}
```

LiveListenerBus 中调用的 postToAll 方法实际定义在父类 SparkListenerBus 中，如代码清单 3-16 所示。

代码清单3-16　SparkListenerBus中的监听器调用

```
protected val sparkListeners = new ArrayBuffer[SparkListener]
    with mutable.SynchronizedBuffer[SparkListener]

def addListener(listener: SparkListener) {
    sparkListeners += listener
}

def postToAll(event: SparkListenerEvent) {
    event match {
        case stageSubmitted: SparkListenerStageSubmitted =>
            foreachListener(_.onStageSubmitted(stageSubmitted))
        case stageCompleted: SparkListenerStageCompleted =>
            foreachListener(_.onStageCompleted(stageCompleted))
        case jobStart: SparkListenerJobStart =>
            foreachListener(_.onJobStart(jobStart))
        case jobEnd: SparkListenerJobEnd =>
            foreachListener(_.onJobEnd(jobEnd))
        case taskStart: SparkListenerTaskStart =>
            foreachListener(_.onTaskStart(taskStart))
        case taskGettingResult: SparkListenerTaskGettingResult =>
            foreachListener(_.onTaskGettingResult(taskGettingResult))
        case taskEnd: SparkListenerTaskEnd =>
            foreachListener(_.onTaskEnd(taskEnd))
        case environmentUpdate: SparkListenerEnvironmentUpdate =>
            foreachListener(_.onEnvironmentUpdate(environmentUpdate))
        case blockManagerAdded: SparkListenerBlockManagerAdded =>
            foreachListener(_.onBlockManagerAdded(blockManagerAdded))
        case blockManagerRemoved: SparkListenerBlockManagerRemoved =>
            foreachListener(_.onBlockManagerRemoved(blockManagerRemoved))
        case unpersistRDD: SparkListenerUnpersistRDD =>
            foreachListener(_.onUnpersistRDD(unpersistRDD))
        case applicationStart: SparkListenerApplicationStart =>
            foreachListener(_.onApplicationStart(applicationStart))
        case applicationEnd: SparkListenerApplicationEnd =>
            foreachListener(_.onApplicationEnd(applicationEnd))
        case metricsUpdate: SparkListenerExecutorMetricsUpdate =>
            foreachListener(_.onExecutorMetricsUpdate(metricsUpdate))
        case SparkListenerShutdown =>
    }
}

private def foreachListener(f: SparkListener => Unit): Unit = {
    sparkListeners.foreach { listener =>
        try {
            f(listener)
        } catch {
            case e: Exception =>
```

```
                logError(s"Listener ${Utils.getFormattedClassName(listener)} threw an
                    exception", e)
            }
        }
    }
```

3.4.2 构造 JobProgressListener

我们以 JobProgressListener 为例来讲解 SparkListener。JobProgressListener 是 SparkContext 中一个重要的组成部分，通过监听 listenerBus 中的事件更新任务进度。SparkStatusTracker 和 SparkUI 实际上也是通过 JobProgressListener 来实现任务状态跟踪的。创建 JobProgressListener 的代码如下。

```
private[spark] val jobProgressListener = new JobProgressListener(conf)
listenerBus.addListener(jobProgressListener)

val statusTracker = new SparkStatusTracker(this)
```

JobProgressListener 的作用是通过 HashMap、ListBuffer 等数据结构存储 JobId 及对应的 JobUIData 信息，并按照激活、完成、失败等 job 状态统计。对于 StageId、StageInfo 等信息按照激活、完成、忽略、失败等 Stage 状态统计，并且存储 StageId 与 JobId 的一对多关系。这些统计信息最终会被 JobPage 和 StagePage 等页面访问和渲染。JobProgressListener 的数据结构见代码清单 3-17。

代码清单3-17 JobProgressListener维护的信息

```
class JobProgressListener(conf: SparkConf) extends SparkListener with Logging {

    import JobProgressListener._

    type JobId = Int
    type StageId = Int
    type StageAttemptId = Int
    type PoolName = String
    type ExecutorId = String

    // Jobs:
    val activeJobs = new HashMap[JobId, JobUIData]
    val completedJobs = ListBuffer[JobUIData]()
    val failedJobs = ListBuffer[JobUIData]()
    val jobIdToData = new HashMap[JobId, JobUIData]

    // Stages:
    val activeStages = new HashMap[StageId, StageInfo]
    val completedStages = ListBuffer[StageInfo]()
    val skippedStages = ListBuffer[StageInfo]()
    val failedStages = ListBuffer[StageInfo]()
    val stageIdToData = new HashMap[(StageId, StageAttemptId), StageUIData]
    val stageIdToInfo = new HashMap[StageId, StageInfo]
```

```
    val stageIdToActiveJobIds = new HashMap[StageId, HashSet[JobId]]
    val poolToActiveStages = HashMap[PoolName, HashMap[StageId, StageInfo]]()
    var numCompletedStages = 0    // 总共完成的Stage数量
    var numFailedStages = 0       // 总共失败的Stage数量

    // Misc:
    val executorIdToBlockManagerId = HashMap[ExecutorId, BlockManagerId]()
    def blockManagerIds = executorIdToBlockManagerId.values.toSeq

    var schedulingMode: Option[SchedulingMode] = None

    // number of non-active jobs and stages (there is no limit for active jobs
         and stages):
    val retainedStages = conf.getInt("spark.ui.retainedStages", DEFAULT_RETAINED_
         STAGES)
    val retainedJobs = conf.getInt("spark.ui.retainedJobs", DEFAULT_RETAINED_JOBS)
```

JobProgressListener 实现了 onJobStart、onJobEnd、onStageCompleted、onStageSubmitted、onTaskStart、onTaskEnd 等方法，这些方法正是在 listenerBus 的驱动下，改变 JobProgressListener 中的各种 Job、Stage 相关的数据。

3.4.3　SparkUI 的创建与初始化

SparkUI 的创建，见代码清单 3-18。

代码清单3-18　SparkUI的声明

```
private[spark] val ui: Option[SparkUI] =
    if (conf.getBoolean("spark.ui.enabled", true)) {
        Some(SparkUI.createLiveUI(this, conf, listenerBus, jobProgressListener,
            env.securityManager,appName))
    } else {
        None
    }

ui.foreach(_.bind())
```

可以看到如果不需要提供 SparkUI 服务，可以将属性 spark.ui.enabled 修改为 false。其中 createLiveUI 实际是调用了 create 方法，见代码清单 3-19。

代码清单3-19　SparkUI的创建

```
def createLiveUI(
        sc: SparkContext,
        conf: SparkConf,
        listenerBus: SparkListenerBus,
        jobProgressListener: JobProgressListener,
        securityManager: SecurityManager,
        appName: String): SparkUI =  {
    create(Some(sc), conf, listenerBus, securityManager, appName,
        jobProgressListener = Some(jobProgressListener))
  }
```

create 方法的实现参见代码清单 3-20。

代码清单3-20 creat方法的实现

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

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

    val environmentListener = new EnvironmentListener
    val storageStatusListener = new StorageStatusListener
    val executorsListener = new ExecutorsListener(storageStatusListener)
    val storageListener = new StorageListener(storageStatusListener)

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

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

根据代码清单 3-20，可以知道在 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 等。最后创建 SparkUI，Spark UI 服务默认是可以被杀掉的，通过修改属性 spark.ui.killEnabled 为 false 可以保证不被杀死。initialize 方法会组织前端页面各个 Tab 和 Page 的展示及布局，参见代码清单 3-21。

代码清单3-21 SparkUI的初始化

```
private[spark] class SparkUI private (
    val sc: Option[SparkContext],
    val conf: SparkConf,
    val securityManager: SecurityManager,
    val environmentListener: EnvironmentListener,
    val storageStatusListener: StorageStatusListener,
```

```
    val executorsListener: ExecutorsListener,
    val jobProgressListener: JobProgressListener,
    val storageListener: StorageListener,
    var appName: String,
    val basePath: String)
extends WebUI(securityManager, SparkUI.getUIPort(conf), conf, basePath,
    "SparkUI")
with Logging {

val killEnabled = sc.map(_.conf.getBoolean("spark.ui.killEnabled", true)).
     getOrElse(false)

/** Initialize all components of the server. */
def initialize() {
    attachTab(new JobsTab(this))
    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(
        createRedirectHandler("/stages/stage/kill", "/stages", stagesTab.
            handleKillRequest))
}
initialize()
```

3.4.4　Spark UI 的页面布局与展示

SparkUI 究竟是如何实现页面布局及展示的？JobsTab 展示所有 Job 的进度、状态信息，这里我们以它为例来说明。JobsTab 会复用 SparkUI 的 killEnabled、SparkContext、jobProgressListener，包括 AllJobsPage 和 JobPage 两个页面，见代码清单 3-22。

代码清单3-22　JobsTab的实现

```
private[ui] class JobsTab(parent: SparkUI) extends SparkUITab(parent, "jobs") {
    val sc = parent.sc
    val killEnabled = parent.killEnabled
    def isFairScheduler = listener.schedulingMode.exists(_ == SchedulingMode.FAIR)
    val listener = parent.jobProgressListener

    attachPage(new AllJobsPage(this))
    attachPage(new JobPage(this))
}
```

AllJobsPage 由 render 方法渲染，利用 jobProgressListener 中的统计监控数据生成激活、完成、失败等状态的 Job 摘要信息，并调用 jobsTable 方法生成表格等 html 元素，最终使用 UIUtils 的 headerSparkPage 封装好 css、js、header 及页面布局等，见代码清单 3-23。

代码清单3-23 AllJobsPage的实现

```
def render(request: HttpServletRequest): Seq[Node] = {
    listener.synchronized {
        val activeJobs = listener.activeJobs.values.toSeq
        val completedJobs = listener.completedJobs.reverse.toSeq
        val failedJobs = listener.failedJobs.reverse.toSeq
        val now = System.currentTimeMillis

        val activeJobsTable =
            jobsTable(activeJobs.sortBy(_.startTime.getOrElse(-1L)).reverse)
        val completedJobsTable =
            jobsTable(completedJobs.sortBy(_.endTime.getOrElse(-1L)).reverse)
        val failedJobsTable =
            jobsTable(failedJobs.sortBy(_.endTime.getOrElse(-1L)).reverse)

        val summary: NodeSeq =
            <div>
                <ul class="unstyled">
                    {if (startTime.isDefined) {
                        // Total duration is not meaningful unless the UI is live
                        <li>
                            <strong>Total Duration: </strong>
                            {UIUtils.formatDuration(now - startTime.get)}
                        </li>
                    }}
                    <li>
                        <strong>Scheduling Mode: </strong>
                        {listener.schedulingMode.map(_.toString).getOrElse("Unknown")}
                    </li>
                    <li>
                        <a href="#active"><strong>Active Jobs:</strong></a>
                        {activeJobs.size}
                    </li>
                    <li>
                        <a href="#completed"><strong>Completed Jobs:</strong></a>
                        {completedJobs.size}
                    </li>
                    <li>
                        <a href="#failed"><strong>Failed Jobs:</strong></a>
                        {failedJobs.size}
                    </li>
                </ul>
            </div>
```

jobsTable 用来生成表格数据，见代码清单 3-24。

代码清单3-24 jobsTable处理表格的实现

```
private def jobsTable(jobs: Seq[JobUIData]): Seq[Node] = {
    val someJobHasJobGroup = jobs.exists(_.jobGroup.isDefined)

    val columns: Seq[Node] = {
```

```
        <th>{if (someJobHasJobGroup) "Job Id (Job Group)" else "Job Id"}</th>
        <th>Description</th>
        <th>Submitted</th>
        <th>Duration</th>
        <th class="sorttable_nosort">Stages: Succeeded/Total</th>
        <th class="sorttable_nosort">Tasks (for all stages): Succeeded/Total</th>
    }

    <table class="table table-bordered table-striped table-condensed sortable">
        <thead>{columns}</thead>
        <tbody>
            {jobs.map(makeRow)}
        </tbody>
    </table>
}
```

表格中每行数据又是通过 makeRow 方法渲染的，参见代码清单 3-25。

代码清单3-25　生成表格中的行

```
def makeRow(job: JobUIData): Seq[Node] = {
    val lastStageInfo = Option(job.stageIds)
        .filter(_.nonEmpty)
        .flatMap { ids => listener.stageIdToInfo.get(ids.max) }
    val lastStageData = lastStageInfo.flatMap { s =>
        listener.stageIdToData.get((s.stageId, s.attemptId))
    }
    val isComplete = job.status == JobExecutionStatus.SUCCEEDED
    val lastStageName = lastStageInfo.map(_.name).getOrElse("(Unknown Stage Name)")
    val lastStageDescription = lastStageData.flatMap(_.description).getOrElse("")
    val duration: Option[Long] = {
        job.startTime.map { start =>
            val end = job.endTime.getOrElse(System.currentTimeMillis())
        end - start
        }
    }
    val formattedDuration = duration.map(d => UIUtils.formatDuration(d)).
         getOrElse("Unknown")
    val formattedSubmissionTime = job.startTime.map(UIUtils.formatDate).
         getOrElse("Unknown")
    val detailUrl =
        "%s/jobs/job?id=%s".format(UIUtils.prependBaseUri(parent.basePath), job.
            jobId)
    <tr>
        <td sorttable_customkey={job.jobId.toString}>
            {job.jobId} {job.jobGroup.map(id => s"($id)").getOrElse("")}
        </td>
        <td>
            <div><em>{lastStageDescription}</em></div>
            <a href={detailUrl}>{lastStageName}</a>
        </td>
            <td sorttable_customkey={job.startTime.getOrElse(-1).toString}>
            {formattedSubmissionTime}
```

```
            </td>
            <td sorttable_customkey={duration.getOrElse(-1).toString}>{formatted-
                 Duration}</td>
            <td class="stage-progress-cell">
                {job.completedStageIndices.size}/{job.stageIds.size - job.numSkipped-
                    Stages}
                {if (job.numFailedStages > 0) s"(${job.numFailedStages} failed)"}
                {if (job.numSkippedStages > 0) s"(${job.numSkippedStages} skipped)"}
            </td>
            <td class="progress-cell">
                {UIUtils.makeProgressBar(started = job.numActiveTasks, completed =
                    job.numCompletedTasks,
                failed = job.numFailedTasks, skipped = job.numSkippedTasks,
                total = job.numTasks - job.numSkippedTasks)}
            </td>
        </tr>
    }
```

代码清单 3-22 中的 attachPage 方法存在于 JobsTab 的父类 WebUITab 中，WebUITab 维护有 ArrayBuffer[WebUIPage] 的数据结构，AllJobsPage 和 JobPage 将被放入此 ArrayBuffer 中，参见代码清单 3-26。

代码清单3-26　WebUITab的实现

```
private[spark] abstract class WebUITab(parent: WebUI, val prefix: String) {
    val pages = ArrayBuffer[WebUIPage]()
    val name = prefix.capitalize

    /** Attach a page to this tab. This prepends the page's prefix with the tab's
         own prefix. */
    def attachPage(page: WebUIPage) {
        page.prefix = (prefix + "/" + page.prefix).stripSuffix("/")
        pages += page
    }

    /** Get a list of header tabs from the parent UI. */
    def headerTabs: Seq[WebUITab] = parent.getTabs

    def basePath: String = parent.getBasePath
}
```

JobsTab 创建之后，将被 attachTab 方法加入 SparkUI 的 ArrayBuffer[WebUITab] 中，并且通过 attachPage 方法，给每一个 page 生成 org.eclipse.jetty.servlet.ServletContextHandler，最后调用 attachHandler 方法将 ServletContextHandler 绑定到 SparkUI，即加入到 handlers ：ArrayBuffer[ServletContextHandler] 和样例类 ServerInfo 的 rootHandler（ContextHandlerCollection）中。SparkUI 继承自 WebUI，attachTab 方法在 WebUI 中实现，参见代码清单 3-27。

代码清单3-27　WebUI的实现

```
private[spark] abstract class WebUI( securityManager: SecurityManager, port: Int,
```

```
        conf: SparkConf, basePath: String = "", name: String = "") extends Logging {

    protected val tabs = ArrayBuffer[WebUITab]()
    protected val handlers = ArrayBuffer[ServletContextHandler]()
    protected var serverInfo: Option[ServerInfo] = None
    protected val localHostName = Utils.localHostName()
    protected val publicHostName = Option(System.getenv("SPARK_PUBLIC_DNS")).
        getOrElse(localHostName)
    private val className = Utils.getFormattedClassName(this)

    def getBasePath: String = basePath
    def getTabs: Seq[WebUITab] = tabs.toSeq
    def getHandlers: Seq[ServletContextHandler] = handlers.toSeq
    def getSecurityManager: SecurityManager = securityManager

    /** Attach a tab to this UI, along with all of its attached pages. */
    def attachTab(tab: WebUITab) {
        tab.pages.foreach(attachPage)
        tabs += tab
    }

    /** Attach a page to this UI. */
    def attachPage(page: WebUIPage) {
        val pagePath = "/" + page.prefix
        attachHandler(createServletHandler(pagePath,
        (request: HttpServletRequest) => page.render(request), securityManager,
            basePath))
    attachHandler(createServletHandler(pagePath.stripSuffix("/") + "/json",
        (request: HttpServletRequest) => page.renderJson(request), security-
            Manager, basePath))
}

    /** Attach a handler to this UI. */
    def attachHandler(handler: ServletContextHandler) {
        handlers += handler
        serverInfo.foreach { info =>
            info.rootHandler.addHandler(handler)
            if (!handler.isStarted) {
                handler.start()
        }
    }
}
```

由于代码清单 3-27 所在的类中使用 import org.apache.spark.ui.JettyUtils._ 导入了 JettyUtils 的静态方法，所以 createServletHandler 方法实际是 JettyUtils 的静态方法 createServletHandler。createServletHandler 实际创建了 javax.servlet.http.HttpServlet 的匿名内部类实例，此实例实际使用 (request: HttpServletRequest) => page.render(request) 函数参数来处理请求，进而渲染页面呈现给用户。有关 createServletHandler 的实现及 Jetty 的相关信息，请参阅附录 C。

3.4.5 SparkUI 的启动

SparkUI 创建好后，需要调用父类 WebUI 的 bind 方法，绑定服务和端口，bind 方法中主要的代码实现如下。

```
serverInfo = Some(startJettyServer("0.0.0.0", port, handlers, conf, name))
```

JettyUtils 的静态方法 startJettyServer 的实现请参阅附录 C。最终启动了 Jetty 提供的服务，默认端口是 4040。

3.5 Hadoop 相关配置及 Executor 环境变量

3.5.1 Hadoop 相关配置信息

默认情况下，Spark 使用 HDFS 作为分布式文件系统，所以需要获取 Hadoop 相关配置信息的代码如下。

```
val hadoopConfiguration = SparkHadoopUtil.get.newConfiguration(conf)
```

获取的配置信息包括：

- 将 Amazon S3 文件系统的 AccessKeyId 和 SecretAccessKey 加载到 Hadoop 的 Configuration；
- 将 SparkConf 中所有以 spark.hadoop. 开头的属性都复制到 Hadoop 的 Configuration；
- 将 SparkConf 的属性 spark.buffer.size 复制为 Hadoop 的 Configuration 的配置 io.file.buffer.size。

注意 如果指定了 SPARK_YARN_MODE 属性，则会使用 YarnSparkHadoopUtil，否则默认为 SparkHadoopUtil。

3.5.2 Executor 环境变量

对 Executor 的环境变量的处理，参见代码清单 3-28。executorEnvs 包含的环境变量将会在 7.2.2 节中介绍的注册应用的过程中发送给 Master，Master 给 Worker 发送调度后，Worker 最终使用 executorEnvs 提供的信息启动 Executor。可以通过配置 spark.executor.memory 指定 Executor 占用的内存大小，也可以配置系统变量 SPARK_EXECUTOR_MEMORY 或者 SPARK_MEM 对其大小进行设置。

代码清单3-28 Executor环境变量的处理

```
private[spark] val executorMemory = conf.getOption("spark.executor.memory")
        .orElse(Option(System.getenv("SPARK_EXECUTOR_MEMORY")))
        .orElse(Option(System.getenv("SPARK_MEM")).map(warnSparkMem))
        .map(Utils.memoryStringToMb)
        .getOrElse(512)
```

```
    // Environment variables to pass to our executors.
    private[spark] val executorEnvs = HashMap[String, String]()

    for { (envKey, propKey) <- Seq(("SPARK_TESTING", "spark.testing"))
        value <- Option(System.getenv(envKey)).orElse(Option(System.getProperty
            (propKey)))} {
        executorEnvs(envKey) = value
    }
    Option(System.getenv("SPARK_PREPEND_CLASSES")).foreach { v =>
        executorEnvs("SPARK_PREPEND_CLASSES") = v
    }
    // The Mesos scheduler backend relies on this environment variable to set
         executor memory.
  executorEnvs("SPARK_EXECUTOR_MEMORY") = executorMemory + "m"
    executorEnvs ++= conf.getExecutorEnv

    // Set SPARK_USER for user who is running SparkContext.
    val sparkUser = Option {
        Option(System.getenv("SPARK_USER")).getOrElse(System.getProperty("user.name"))
    }.getOrElse {
        SparkContext.SPARK_UNKNOWN_USER
    }
    executorEnvs("SPARK_USER") = sparkUser
```

3.6　创建任务调度器 TaskScheduler

TaskScheduler 也是 SparkContext 的重要组成部分，负责任务的提交，并且请求集群管理器对任务调度。TaskScheduler 也可以看做任务调度的客户端。创建 TaskScheduler 的代码如下。

```
private[spark] var (schedulerBackend, taskScheduler) =
    SparkContext.createTaskScheduler(this, master)
```

createTaskScheduler 方法会根据 master 的配置匹配部署模式，创建 TaskSchedulerImpl，并生成不同的 SchedulerBackend。本章为了使读者更容易理解 Spark 的初始化流程，故以 local 模式为例，其余模式将在第 7 章详解。master 匹配 local 模式的代码如下。

```
master match {
    case "local" =>
        val scheduler = new TaskSchedulerImpl(sc, MAX_LOCAL_TASK_FAILURES, isLocal = true)
        val backend = new LocalBackend(scheduler, 1)
        scheduler.initialize(backend)
        (backend, scheduler)
```

3.6.1　创建 TaskSchedulerImpl

TaskSchedulerImpl 的构造过程如下：

1）从 SparkConf 中读取配置信息，包括每个任务分配的 CPU 数、调度模式（调度模式有 FAIR 和 FIFO 两种，默认为 FIFO，可以修改属性 spark.scheduler.mode 来改变）等。

2）创建 TaskResultGetter，它的作用是通过线程池（Executors.newFixedThreadPool 创建的，默认 4 个线程，线程名字以 task-result-getter 开头，线程工厂默认是 Executors.defaultThreadFactory）对 Worker 上的 Executor 发送的 Task 的执行结果进行处理。

TaskSchedulerImpl 的实现见代码清单 3-29。

代码清单3-29 TaskSchedulerImpl的实现

```
var dagScheduler: DAGScheduler = null
var backend: SchedulerBackend = null
val mapOutputTracker = SparkEnv.get.mapOutputTracker
var schedulableBuilder: SchedulableBuilder = null
var rootPool: Pool = null
// default scheduler is FIFO
private val schedulingModeConf = conf.get("spark.scheduler.mode", "FIFO")
val schedulingMode: SchedulingMode = try {
    SchedulingMode.withName(schedulingModeConf.toUpperCase)
} catch {
    case e: java.util.NoSuchElementException =>
        throw new SparkException(s"Unrecognized spark.scheduler.mode: $scheduling-
            ModeConf")
}

// This is a var so that we can reset it for testing purposes.
private[spark] var taskResultGetter = new TaskResultGetter(sc.env, this)
```

TaskSchedulerImpl 的调度模式有 FAIR 和 FIFO 两种。任务的最终调度实际都是落实到接口 SchedulerBackend 的具体实现上的。为方便分析，我们先来看看 local 模式中 SchedulerBackend 的实现 LocalBackend。LocalBackend 依赖于 LocalActor 与 ActorSystem 进行消息通信。LocalBackend 的实现参见代码清单 3-30。

代码清单3-30 LocalBackend的实现

```
private[spark] class LocalBackend(scheduler: TaskSchedulerImpl, val totalCores: Int)
    extends SchedulerBackend with ExecutorBackend {

    private val appId = "local-" + System.currentTimeMillis
    var localActor: ActorRef = null

    override def start() {
        localActor = SparkEnv.get.actorSystem.actorOf(
            Props(new LocalActor(scheduler, this, totalCores)),
            "LocalBackendActor")
    }

    override def stop() {
        localActor ! StopExecutor
    }

    override def reviveOffers() {
        localActor ! ReviveOffers
```

```
    }

    override def defaultParallelism() =
        scheduler.conf.getInt("spark.default.parallelism", totalCores)

    override def killTask(taskId: Long, executorId: String, interruptThread:
        Boolean) {
        localActor ! KillTask(taskId, interruptThread)
    }

    override def statusUpdate(taskId: Long, state: TaskState, serializedData:
        ByteBuffer) {
        localActor ! StatusUpdate(taskId, state, serializedData)
    }

    override def applicationId(): String = appId
}
```

3.6.2　TaskSchedulerImpl 的初始化

创建完 TaskSchedulerImpl 和 LocalBackend 后，对 TaskSchedulerImpl 调用方法 initialize 进行初始化。以默认的 FIFO 调度为例，TaskSchedulerImpl 的初始化过程如下：

1）使 TaskSchedulerImpl 持有 LocalBackend 的引用。

2）创建 Pool，Pool 中缓存了调度队列、调度算法及 TaskSetManager 集合等信息。

3）创建 FIFOSchedulableBuilder，FIFOSchedulableBuilder 用来操作 Pool 中的调度队列。

initialize 方法的实现见代码清单 3-31。

代码清单3-31　TaskSchedulerImpl的初始化

```
def initialize(backend: SchedulerBackend) {
    this.backend = backend
    rootPool = new Pool("", schedulingMode, 0, 0)
    schedulableBuilder = {
        schedulingMode match {
            case SchedulingMode.FIFO =>
                new FIFOSchedulableBuilder(rootPool)
            case SchedulingMode.FAIR =>
                new FairSchedulableBuilder(rootPool, conf)
        }
    }
    schedulableBuilder.buildPools()
}
```

3.7　创建和启动 DAGScheduler

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，等等。创建 DAG-

Scheduler 的代码如下。

```
@volatile private[spark] var dagScheduler: DAGScheduler = _
    dagScheduler = new DAGScheduler(this)
```

DAGScheduler 的数据结构主要维护 jobId 和 stageId 的关系、Stage、ActiveJob，以及缓存的 RDD 的 partitions 的位置信息，见代码清单 3-32。

代码清单3-32　DAGScheduler维护的数据结构

```
private[scheduler] val nextJobId = new AtomicInteger(0)
private[scheduler] def numTotalJobs: Int = nextJobId.get()
private val nextStageId = new AtomicInteger(0)

private[scheduler] val jobIdToStageIds = new HashMap[Int, HashSet[Int]]
private[scheduler] val stageIdToStage = new HashMap[Int, Stage]
private[scheduler] val shuffleToMapStage = new HashMap[Int, Stage]
private[scheduler] val jobIdToActiveJob = new HashMap[Int, ActiveJob]

    // Stages we need to run whose parents aren't done
    private[scheduler] val waitingStages = new HashSet[Stage]
    // Stages we are running right now
    private[scheduler] val runningStages = new HashSet[Stage]
    // Stages that must be resubmitted due to fetch failures
    private[scheduler] val failedStages = new HashSet[Stage]

    private[scheduler] val activeJobs = new HashSet[ActiveJob]

    // Contains the locations that each RDD's partitions are cached on
    private val cacheLocs = new HashMap[Int, Array[Seq[TaskLocation]]]
    private val failedEpoch = new HashMap[String, Long]

    private val dagSchedulerActorSupervisor =
        env.actorSystem.actorOf(Props(new DAGSchedulerActorSupervisor(this)))

    private val closureSerializer = SparkEnv.get.closureSerializer.newInstance()
```

在构造 DAGScheduler 的时候会调用 initializeEventProcessActor 方法创建 DAGSchedulerEventProcessActor，见代码清单 3-33。

代码清单3-33　DAGSchedulerEventProcessActor的初始化

```
    private[scheduler] var eventProcessActor: ActorRef = _
private def initializeEventProcessActor() {
        // blocking the thread until supervisor is started, which ensures eventProcess-
             Actor is
        // not null before any job is submitted
        implicit val timeout = Timeout(30 seconds)
        val initEventActorReply =
            dagSchedulerActorSupervisor ? Props(new DAGSchedulerEventProcessActor(this))
        eventProcessActor = Await.result(initEventActorReply, timeout.duration).
            asInstanceOf[ActorRef]
}

initializeEventProcessActor()
```

这里的 DAGSchedulerActorSupervisor 主要作为 DAGSchedulerEventProcessActor 的监管者，负责生成 DAGSchedulerEventProcessActor。从代码清单 3-34 可以看出，DAGSchedulerActorSupervisor 对于 DAGSchedulerEventProcessActor 采用了 Akka 的一对一监管策略。DAGSchedulerActorSupervisor 一旦生成 DAGSchedulerEventProcessActor，并注册到 ActorSystem，ActorSystem 就会调用 DAGSchedulerEventProcessActor 的 preStart，taskScheduler 于是就持有了 dagScheduler，见代码清单 3-35。从代码清单 3-35 我们还看到 DAG-SchedulerEventProcessActor 所能处理的消息类型，比如 JobSubmitted、BeginEvent、CompletionEvent 等。DAGSchedulerEventProcessActor 接受这些消息后会有不同的处理动作。在本章，读者只需要理解到这里即可，后面章节用到时会详细分析。

代码清单3-34　DAGSchedulerActorSupervisor的监管策略

```
private[scheduler] class DAGSchedulerActorSupervisor(dagScheduler: DAGScheduler)
    extends Actor with Logging {

    override val supervisorStrategy =
        OneForOneStrategy() {
            case x: Exception =>
                logError("eventProcesserActor failed; shutting down SparkContext", x)
                try {
                    dagScheduler.doCancelAllJobs()
                } catch {
                    case t: Throwable => logError("DAGScheduler failed to cancel
                        all jobs.", t)
                }
                dagScheduler.sc.stop()
                Stop
    }

def receive = {
        case p: Props => sender ! context.actorOf(p)
        case _ => logWarning("received unknown message in DAGSchedulerActorSupervisor")
    }
}
```

代码清单3-35　DAGSchedulerEventProcessActor的实现

```
private[scheduler] class DAGSchedulerEventProcessActor(dagScheduler: DAGS-
    cheduler)
    extends Actor with Logging {
    override def preStart() {
        dagScheduler.taskScheduler.setDAGScheduler(dagScheduler)
    }
    /**
    * The main event loop of the DAG scheduler.
    */
    def receive = {
        case JobSubmitted(jobId, rdd, func, partitions, allowLocal, callSite,
            listener, properties) =>
```

```
            dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions,
                allowLocal, callSite,
                listener, properties)
        case StageCancelled(stageId) =>
            dagScheduler.handleStageCancellation(stageId)
        case JobCancelled(jobId) =>
            dagScheduler.handleJobCancellation(jobId)
        case JobGroupCancelled(groupId) =>
            dagScheduler.handleJobGroupCancelled(groupId)
        case AllJobsCancelled =>
            dagScheduler.doCancelAllJobs()
        case ExecutorAdded(execId, host) =>
            dagScheduler.handleExecutorAdded(execId, host)
        case ExecutorLost(execId) =>
            dagScheduler.handleExecutorLost(execId, fetchFailed = false)
        case BeginEvent(task, taskInfo) =>
            dagScheduler.handleBeginEvent(task, taskInfo)
        case GettingResultEvent(taskInfo) =>
            dagScheduler.handleGetTaskResult(taskInfo)
        case completion @ CompletionEvent(task, reason, _, _, taskInfo,
            taskMetrics) =>
            dagScheduler.handleTaskCompletion(completion)
        case TaskSetFailed(taskSet, reason) =>
            dagScheduler.handleTaskSetFailed(taskSet, reason)
        case ResubmitFailedStages =>
            dagScheduler.resubmitFailedStages()
    }
    override def postStop() {
        // Cancel any active jobs in postStop hook
        dagScheduler.cleanUpAfterSchedulerStop()
    }
```

3.8 TaskScheduler 的启动

3.6 节介绍了任务调度器 TaskScheduler 的创建，要想 TaskScheduler 发挥作用，必须要启动它，代码如下。

```
taskScheduler.start()
```

TaskScheduler 在启动的时候，实际调用了 backend 的 start 方法。

```
override def start() {
        backend.start()
    }
```

以 LocalBackend 为例，启动 LocalBackend 时向 actorSystem 注册了 LocalActor，见代码清单 3-30 所示。

3.8.1 创建 LocalActor

创建 LocalActor 的过程主要是构建本地的 Executor，见代码清单 3-36。

代码清单3-36　LocalActor的实现

```
private[spark] class LocalActor(scheduler: TaskSchedulerImpl, executorBackend:
    LocalBackend,
    private val totalCores: Int) extends Actor with ActorLogReceive with Logging {
    import context.dispatcher   // to use Akka's scheduler.scheduleOnce()
    private var freeCores = totalCores
    private val localExecutorId = SparkContext.DRIVER_IDENTIFIER
    private val localExecutorHostname = "localhost"

    val executor = new Executor(
        localExecutorId, localExecutorHostname, scheduler.conf.getAll,
            totalCores, isLocal = true)

    override def receiveWithLogging = {
        case ReviveOffers =>
            reviveOffers()

        case StatusUpdate(taskId, state, serializedData) =>
            scheduler.statusUpdate(taskId, state, serializedData)
            if (TaskState.isFinished(state)) {
                freeCores += scheduler.CPUS_PER_TASK
                reviveOffers()
            }

        case KillTask(taskId, interruptThread) =>
            executor.killTask(taskId, interruptThread)

        case StopExecutor =>
            executor.stop()
    }

}
```

Executor 的构建，见代码清单 3-37，主要包括以下步骤。

1）创建并注册 ExecutorSource。ExecutorSource 是做什么的呢？笔者将在 3.8.2 节详细介绍。

2）获取 SparkEnv。如果是非 local 模式，Worker 上的 CoarseGrainedExecutorBackend 向 Driver 上的 CoarseGrainedExecutorBackend 注册 Executor 时，则需要新建 SparkEnv。可以修改属性 spark.executor.port（默认为 0，表示随机生成）来配置 Executor 中的 ActorSystem 的端口号。

3）创建并注册 ExecutorActor。ExecutorActor 负责接受发送给 Executor 的消息。

4）urlClassLoader 的创建。为什么需要创建这个 ClassLoader？在非 local 模式中，Driver 或者 Worker 上都会有多个 Executor，每个 Executor 都设置自身的 urlClassLoader，用于加载任务上传的 jar 包中的类，有效对任务的类加载环境进行隔离。

5）创建 Executor 执行 Task 的线程池。此线程池[⊖]用于执行任务。

6）启动 Executor 的心跳线程。此线程用于向 Driver 发送心跳。

⊖ 是通过调用 Utils.newDaemonCachedThreadPool 创建的，具体实现请参阅附录 A。

此外，还包括 Akka 发送消息的帧大小（10 485 760 字节）、结果总大小的字节限制（1 073 741 824 字节）、正在运行的 task 的列表、设置 serializer 的默认 ClassLoader 为创建的 ClassLoader 等。

代码清单3-37　Executor的构建

```
    val executorSource = new ExecutorSource(this, executorId)
private val env = {
        if (!isLocal) {
            val port = conf.getInt("spark.executor.port", 0)
            val _env = SparkEnv.createExecutorEnv(
                conf, executorId, executorHostname, port, numCores, isLocal,
                    actorSystem)
            SparkEnv.set(_env)
            _env.metricsSystem.registerSource(executorSource)
            _env.blockManager.initialize(conf.getAppId)
            _env
        } else {
            SparkEnv.get
        }
    }

    private val executorActor = env.actorSystem.actorOf(
        Props(new ExecutorActor(executorId)), "ExecutorActor")

    private val urlClassLoader = createClassLoader()
    private val replClassLoader = addReplClassLoaderIfNeeded(urlClassLoader)
    env.serializer.setDefaultClassLoader(urlClassLoader)

    private val akkaFrameSize = AkkaUtils.maxFrameSizeBytes(conf)
    private val maxResultSize = Utils.getMaxResultSize(conf)

    val threadPool = Utils.newDaemonCachedThreadPool("Executor task launch worker")
    private val runningTasks = new ConcurrentHashMap[Long, TaskRunner]
    startDriverHeartbeater()
```

3.8.2　ExecutorSource 的创建与注册

ExecutorSource 用于测量系统。通过 metricRegistry 的 register 方法注册计量，这些计量信息包括 threadpool.activeTasks、threadpool.completeTasks、threadpool.currentPool_size、threadpool.maxPool_size、filesystem.hdfs.write_bytes、filesystem.hdfs.read_ops、filesystem.file.write_bytes、filesystem.hdfs.largeRead_ops、filesystem.hdfs.write_ops 等，ExecutorSource 的实现见代码清单 3-38。Metric 接口的具体实现，参考附录 D。

代码清单3-38　ExecutorSource的实现

```
private[spark] class ExecutorSource(val executor: Executor, executorId: String)
    extends Source {
    private def fileStats(scheme: String) : Option[FileSystem.Statistics] =
        FileSystem.getAllStatistics().filter(s => s.getScheme.equals(scheme)).
```

```
            headOption

    private def registerFileSystemStat[T](
            scheme: String, name: String, f: FileSystem.Statistics => T,
                defaultValue: T) = {
        metricRegistry.register(MetricRegistry.name("filesystem", scheme, name),
            new Gauge[T] {
            override def getValue: T = fileStats(scheme).map(f).getOrElse
                (defaultValue)
        })
    }
    override val metricRegistry = new MetricRegistry()
    override val sourceName = "executor"

metricRegistry.register(MetricRegistry.name("threadpool", "activeTasks"), new
    Gauge[Int] {
        override def getValue: Int = executor.threadPool.getActiveCount()
    })
    metricRegistry.register(MetricRegistry.name("threadpool", "completeTasks"),
        new Gauge[Long] {
        override def getValue: Long = executor.threadPool.getCompletedTaskCount()
    })
    metricRegistry.register(MetricRegistry.name("threadpool", "currentPool_
        size"), new Gauge[Int] {
        override def getValue: Int = executor.threadPool.getPoolSize()
    })
    metricRegistry.register(MetricRegistry.name("threadpool", "maxPool_size"),
        new Gauge[Int] {
        override def getValue: Int = executor.threadPool.getMaximumPoolSize()
    })

    // Gauge for file system stats of this executor
    for (scheme <- Array("hdfs", "file")) {
        registerFileSystemStat(scheme, "read_bytes", _.getBytesRead(), 0L)
        registerFileSystemStat(scheme, "write_bytes", _.getBytesWritten(), 0L)
        registerFileSystemStat(scheme, "read_ops", _.getReadOps(), 0)
        registerFileSystemStat(scheme, "largeRead_ops", _.getLargeReadOps(), 0)
        registerFileSystemStat(scheme, "write_ops", _.getWriteOps(), 0)
    }
}
```

创建完 ExecutorSource 后，调用 MetricsSystem 的 registerSource 方法将 ExecutorSource 注册到 MetricsSystem。registerSource 方法使用 MetricRegistry 的 register 方法，将 Source 注册到 MetricRegistry，见代码清单 3-39。关于 MetricRegistry，具体参阅附录 D。

代码清单3-39　MetricsSystem注册Source的实现

```
def registerSource(source: Source) {
    sources += source
    try {
        val regName = buildRegistryName(source)
        registry.register(regName, source.metricRegistry)
```

```
        } catch {
            case e: IllegalArgumentException => logInfo("Metrics already registered", e)
        }
    }
```

3.8.3 ExecutorActor 的构建与注册

ExecutorActor 很简单，当接收到 SparkUI 发来的消息时，将所有线程的栈信息发送回去，代码实现如下。

```
override def receiveWithLogging = {
    case TriggerThreadDump =>
        sender ! Utils.getThreadDump()
}
```

3.8.4 Spark 自身 ClassLoader 的创建

获取要创建的 ClassLoader 的父加载器 currentLoader，然后根据 currentJars 生成 URL 数组，spark.files.userClassPathFirst 属性指定加载类时是否先从用户的 classpath 下加载，最后创建 ExecutorURLClassLoader 或者 ChildExecutorURLClassLoader，见代码清单 3-40。

代码清单3-40 Spark自身ClassLoader的创建

```
private def createClassLoader(): MutableURLClassLoader = {
    val currentLoader = Utils.getContextOrSparkClassLoader

    val urls = currentJars.keySet.map { uri =>
        new File(uri.split("/").last).toURI.toURL
    }.toArray
    val userClassPathFirst = conf.getBoolean("spark.files.userClassPathFirst",
         false)
    userClassPathFirst match {
        case true => new ChildExecutorURLClassLoader(urls, currentLoader)
        case false => new ExecutorURLClassLoader(urls, currentLoader)
    }
}
```

Utils.getContextOrSparkClassLoader 的实现见附录 A。ExecutorURLClassLoader 或者 ChildExecutorURLClassLoader 实际上都继承了 URLClassLoader，见代码清单 3-41。

代码清单3-41 ChildExecutorURLClassLoader和ExecutorLIRLClassLoader的实现

```
private[spark] class ChildExecutorURLClassLoader(urls: Array[URL], parent: ClassLoader)
    extends MutableURLClassLoader {

    private object userClassLoader extends URLClassLoader(urls, null){
        override def addURL(url: URL) {
            super.addURL(url)
        }
```

```
    override def findClass(name: String): Class[_] = {
        super.findClass(name)
    }
}

private val parentClassLoader = new ParentClassLoader(parent)

override def findClass(name: String): Class[_] = {
    try {
        userClassLoader.findClass(name)
    } catch {
        case e: ClassNotFoundException => {
            parentClassLoader.loadClass(name)
        }
    }
}

    def addURL(url: URL) {
        userClassLoader.addURL(url)
    }

    def getURLs() = {
        userClassLoader.getURLs()
    }
}

private[spark] class ExecutorURLClassLoader(urls: Array[URL], parent: ClassLoader)
    extends URLClassLoader(urls, parent) with MutableURLClassLoader {

    override def addURL(url: URL) {
        super.addURL(url)
    }
}
```

如果需要 REPL 交互，还会调用 addReplClassLoaderIfNeeded 创建 replClassLoader，见代码清单 3-42。

代码清单3-42　addReplClassLoaderIfNeeded的实现

```
private def addReplClassLoaderIfNeeded(parent: ClassLoader): ClassLoader = {
    val classUri = conf.get("spark.repl.class.uri", null)
    if (classUri != null) {
        logInfo("Using REPL class URI: " + classUri)
        val userClassPathFirst: java.lang.Boolean =
        conf.getBoolean("spark.files.userClassPathFirst", false)
    try {
        val klass = Class.forName("org.apache.spark.repl.ExecutorClassLoader")
            .asInstanceOf[Class[_ <: ClassLoader]]
        val constructor = klass.getConstructor(classOf[SparkConf], classOf[String],
            classOf[ClassLoader], classOf[Boolean])
        constructor.newInstance(conf, classUri, parent, userClassPathFirst)
    } catch {
```

```
            case _: ClassNotFoundException =>
                logError("Could not find org.apache.spark.repl.ExecutorClassLoader on
                    classpath!")
                System.exit(1)
                null
            }
        } else {
            parent
        }
    }
```

3.8.5 启动 Executor 的心跳线程

Executor 的心跳由 startDriverHeartbeater 启动，见代码清单 3-43。Executor 心跳线程的间隔由属性 spark.executor.heartbeatInterval 配置，默认是 10 000 毫秒。此外，超时时间是 30 秒，超时重试次数是 3 次，重试间隔是 3000 毫秒，使用 actorSystem.actorSelection (url) 方法查找到匹配的 Actor 引用，url 是 akka.tcp://sparkDriver@ $driverHost:$driverPort/user/HeartbeatReceiver，最终创建一个运行过程中，每次会休眠 10 000 ～ 20 000 毫秒的线程。此线程从 runningTasks 获取最新的有关 Task 的测量信息，将其与 executorId、blockManagerId 封装为 Heartbeat 消息，向 HeartbeatReceiver 发送 Heartbeat 消息。

代码清单3-43　启动Executor的心跳线程

```
def startDriverHeartbeater() {
    val interval = conf.getInt("spark.executor.heartbeatInterval", 10000)
    val timeout = AkkaUtils.lookupTimeout(conf)
    val retryAttempts = AkkaUtils.numRetries(conf)
    val retryIntervalMs = AkkaUtils.retryWaitMs(conf)
    val heartbeatReceiverRef = AkkaUtils.makeDriverRef("HeartbeatReceiver",
         conf,env.actorSystem)
    val t = new Thread() {
        override def run() {
            // Sleep a random interval so the heartbeats don't end up in sync
            Thread.sleep(interval + (math.random * interval).asInstanceOf[Int])
            while (!isStopped) {
                val tasksMetrics = new ArrayBuffer[(Long, TaskMetrics)]()
                val curGCTime = gcTime
                for (taskRunner <- runningTasks.values()) {
                    if (!taskRunner.attemptedTask.isEmpty) {
                        Option(taskRunner.task).flatMap(_.metrics).foreach { metrics =>
                            metrics.updateShuffleReadMetrics
                            metrics.jvmGCTime = curGCTime - taskRunner.startGCTime
                            if (isLocal) {
                                val copiedMetrics = Utils.deserialize[TaskMetrics]
                                     (Utils.serialize(metrics))
                                tasksMetrics += ((taskRunner.taskId, copiedMetrics))
                        } else {
                            // It will be copied by serialization
```

```
                                tasksMetrics += ((taskRunner.taskId, metrics))
                            }
                        }
                    }
                }
                val message = Heartbeat(executorId, tasksMetrics.toArray, env.
                     blockManager.blockManagerId)
                try {
                    val response = AkkaUtils.askWithReply[HeartbeatResponse](message,
                         heartbeatReceiverRef,
                        retryAttempts, retryIntervalMs, timeout)
                    if (response.reregisterBlockManager) {
                        logWarning("Told to re-register on heartbeat")
                        env.blockManager.reregister()
                    }
                } catch {
                    case NonFatal(t) => logWarning("Issue communicating with driver
                        in heartbeater", t)
                }
    Thread.sleep(interval)
                }
            }
        }
        t.setDaemon(true)
        t.setName("Driver Heartbeater")
        t.start()
    }
```

这个心跳线程的作用是什么呢？其作用有两个：

- 更新正在处理的任务的测量信息；
- 通知 BlockManagerMaster，此 Executor 上的 BlockManager 依然活着。

下面对心跳线程的实现详细分析下，读者可以自行选择是否需要阅读。

初始化 TaskSchedulerImpl 后会创建心跳接收器 HeartbeatReceiver。HeartbeatReceiver 接收所有分配给当前 Driver Application 的 Executor 的心跳，并将 Task、Task 计量信息、心跳等交给 TaskSchedulerImpl 和 DAGScheduler 作进一步处理。创建心跳接收器的代码如下。

```
private val heartbeatReceiver = env.actorSystem.actorOf(
    Props(new HeartbeatReceiver(taskScheduler)), "HeartbeatReceiver")
```

HeartbeatReceiver 在收到心跳消息后，会调用 TaskScheduler 的 executorHeartbeatReceived 方法，代码如下。

```
override def receiveWithLogging = {
    case Heartbeat(executorId, taskMetrics, blockManagerId) =>
        val response = HeartbeatResponse(
            !scheduler.executorHeartbeatReceived(executorId, taskMetrics,
                blockManagerId))
        sender ! response
  }
```

executorHeartbeatReceived 的实现代码如下。

```
val metricsWithStageIds: Array[(Long, Int, Int, TaskMetrics)] = synchronized {
    taskMetrics.flatMap { case (id, metrics) =>
        taskIdToTaskSetId.get(id)
            .flatMap(activeTaskSets.get)
            .map(taskSetMgr => (id, taskSetMgr.stageId, taskSetMgr.taskSet.
                 attempt, metrics))
    }
}
dagScheduler.executorHeartbeatReceived(execId, metricsWithStageIds, blockManagerId)
```

这段程序通过遍历 taskMetrics，依据 taskIdToTaskSetId 和 activeTaskSets 找到 TaskSet-Manager。然后将 taskId、TaskSetManager.stageId、TaskSetManager .taskSet.attempt、TaskMetrics 封装到类型为 Array[(Long, Int, Int, TaskMetrics)] 的数组 metricsWithStageIds 中。最后调用了 dag-Scheduler 的 executorHeartbeatReceived 方法，其实现如下。

```
listenerBus.post(SparkListenerExecutorMetricsUpdate(execId, taskMetrics))
implicit val timeout = Timeout(600 seconds)

Await.result(
    blockManagerMaster.driverActor ? BlockManagerHeartbeat(blockManagerId),
    timeout.duration).asInstanceOf[Boolean]
```

dagScheduler 将 executorId、metricsWithStageIds 封装为 SparkListenerExecutorMetricsUpdate 事件，并 post 到 listenerBus 中，此事件用于更新 Stage 的各种测量数据。最后给 BlockManagerMaster 持有的 BlockManagerMasterActor 发送 BlockManagerHeartbeat 消息。BlockManagerMasterActor 在收到消息后会匹配执行 heartbeatReceived 方法（参见 4.3.1 节）。heartbeatReceived 最终更新 BlockManagerMaster 对 BlockManger 的最后可见时间（即更新 Block-ManagerId 对应的 BlockManagerInfo 的 _lastSeenMs，见代码清单 3-44）。

代码清单3-44　BlockManagerMasterActor的心跳处理

```
private def heartbeatReceived(blockManagerId: BlockManagerId): Boolean = {
    if (!blockManagerInfo.contains(blockManagerId)) {
        blockManagerId.isDriver && !isLocal
    } else {
        blockManagerInfo(blockManagerId).updateLastSeenMs()
        true
    }
}
```

local 模式下 Executor 的心跳通信过程，可以用图 3-3 来表示。

注意　在非 local 模式中，Executor 发送心跳的过程是一样的，主要的区别是 Executor 进程与 Driver 不在同一个进程，甚至不在同一个节点上。

接下来会初始化块管理器 BlockManager，代码如下。

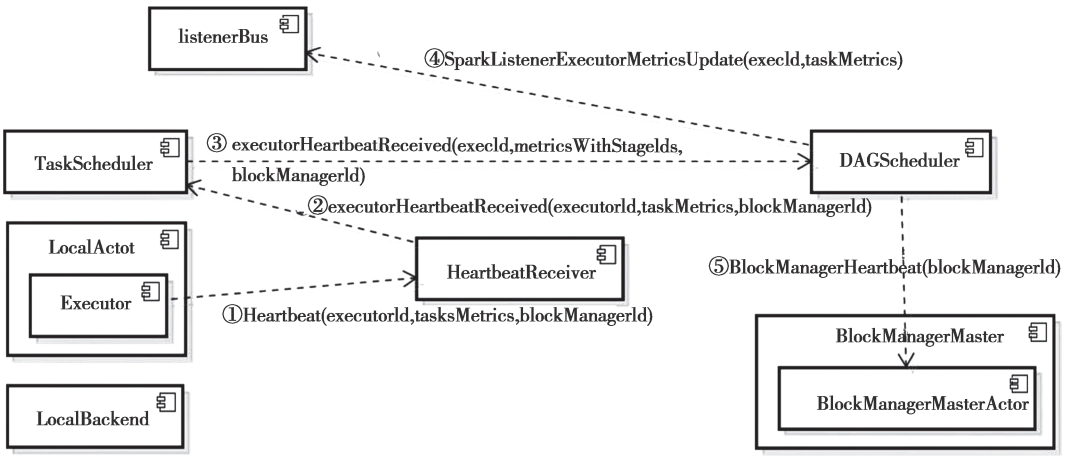

图 3-3　Executor 的心跳通信过程

```
env.blockManager.initialize(applicationId)
```

具体的初始化过程，请参阅第 4 章。

3.9　启动测量系统 MetricsSystem

MetricsSystem 使用 codahale 提供的第三方测量仓库 Metrics，有关 Metrics 的具体信息可以参考附录 D。MetricsSystem 中有三个概念：

- Instance：指定了谁在使用测量系统；
- Source：指定了从哪里收集测量数据；
- Sink：指定了往哪里输出测量数据。

Spark 按照 Instance 的不同，区分为 Master、Worker、Application、Driver 和 Executor。

Spark 目前提供的 Sink 有 ConsoleSink、CsvSink、JmxSink、MetricsServlet、GraphiteSink 等。

Spark 中使用 MetricsServlet 作为默认的 Sink。

MetricsSystem 的启动代码如下。

```
val metricsSystem = env.metricsSystem
    metricsSystem.start()
```

MetricsSystem 的启动过程包括以下步骤：

1）注册 Sources；

2）注册 Sinks；

3）给 Sinks 增加 Jetty 的 ServletContextHandler。

MetricsSystem 启动完毕后，会遍历与 Sinks 有关的 ServletContextHandler，并调用 attachHandler 将它们绑定到 Spark UI 上。

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

3.9.1 注册 Sources

registerSources 方法用于注册 Sources，告诉测量系统从哪里收集测量数据，它的实现见代码清单 3-45。注册 Sources 的过程分为以下步骤：

1）从 metricsConfig 获取 Driver 的 Properties，默认为创建 MetricsSystem 的过程中解析的 {sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet, sink.servlet.path=/metrics/json}。

2）用正则匹配 Driver 的 Properties 中以 source. 开头的属性。然后将属性中的 Source 反射得到的实例加入 ArrayBuffer[Source]。

3）将每个 source 的 metricRegistry（也是 MetricSet 的子类型）注册到 ConcurrentMap<String, Metric> metrics。这里的 registerSource 方法已在 3.8.2 节讲解过。

代码清单3-45 MetricsSystem注册Sources的实现

```
private def registerSources() {
    val instConfig = metricsConfig.getInstance(instance)
    val sourceConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.
         SOURCE_REGEX)

    // Register all the sources related to instance
    sourceConfigs.foreach { kv =>
        val classPath = kv._2.getProperty("class")
        try {
            val source = Class.forName(classPath).newInstance()
            registerSource(source.asInstanceOf[Source])
        } catch {
            case e: Exception => logError("Source class " + classPath + " cannot
                be instantiated", e)
        }
    }
}
```

3.9.2 注册 Sinks

registerSinks 方法用于注册 Sinks，即告诉测量系统 MetricsSystem 往哪里输出测量数据，它的实现见代码清单 3-46。注册 Sinks 的步骤如下：

1）从 Driver 的 Properties 中用正则匹配以 sink. 开头的属性，如 {sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet, sink.servlet.path=/metrics/json}，将其转换为 Map(servlet -> {class=org.apache.spark.metrics.sink.MetricsServlet, path=/metrics/json})。

2）将子属性 class 对应的类 metricsServlet 反射得到 MetricsServlet 实例。如果属性的 key 是 servlet，将其设置为 metricsServlet；如果是 Sink，则加入到 ArrayBuffer[Sink] 中。

代码清单3-46　MetricsSystem注册Sinks的实现

```
private def registerSinks() {
    val instConfig = metricsConfig.getInstance(instance)
    val sinkConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.SINK_
         REGEX)
    sinkConfigs.foreach { kv =>
        val classPath = kv._2.getProperty("class")
        if (null != classPath) {
            try {
                val sink = Class.forName(classPath)
                .getConstructor(classOf[Properties], classOf[MetricRegistry],
                     classOf[SecurityManager])
                .newInstance(kv._2, registry, securityMgr)
            if (kv._1 == "servlet") {
                metricsServlet = Some(sink.asInstanceOf[MetricsServlet])
            } else {
                sinks += sink.asInstanceOf[Sink]
            }
            } catch {
                case e: Exception => logError("Sink class "+ classPath + " cannot
                    be instantialized",e)
            }
        }
    }
}
```

3.9.3　给 Sinks 增加 Jetty 的 ServletContextHandler

为了能够在 SparkUI（网页）访问到测量数据，所以需要给 Sinks 增加 Jetty 的 ServletContextHandler，这里主要用到 MetricsSystem 的 getServletHandlers 方法实现如下。

```
def getServletHandlers = {
    require(running, "Can only call getServletHandlers on a running MetricsSystem")
    metricsServlet.map(_.getHandlers).getOrElse(Array())
}
```

可以看到调用了 metricsServlet 的 getHandlers，其实现如下。

```
def getHandlers = Array[ServletContextHandler](
    createServletHandler(servletPath,
        new ServletParams(request => getMetricsSnapshot(request), "text/json"),
             securityMgr)
)
```

最终生成处理 /metrics/json 请求的 ServletContextHandler，而请求的真正处理由 getMetricsSnapshot 方法，利用 fastjson 解析。生成的 ServletContextHandler 通过 SparkUI 的 attachHandler 方法，也被绑定到 SparkUI（creatServlethandler 与 attachHandler 方法在 3.4.4 节详细讲述过）。最终我们可以使用以下这些地址来访问测量数据。

- http://localhost:4040/metrics/applications/json。
- http://localhost:4040/metrics/json。
- http://localhost:4040/metrics/master/json。

3.10 创建和启动 ExecutorAllocationManager

ExecutorAllocationManager 用于对已分配的 Executor 进行管理，创建和启动 ExecutorAllocationManager 的代码如下。

```
private[spark] val executorAllocationManager: Option[ExecutorAllocationManager]
    =
    if (conf.getBoolean("spark.dynamicAllocation.enabled", false)) {
        Some(new ExecutorAllocationManager(this, listenerBus, conf))
    } else {
        None
    }
executorAllocationManager.foreach(_.start())
```

默认情况下不会创建 ExecutorAllocationManager，可以修改属性 spark.dynamicAllocation.enabled 为 true 来创建。ExecutorAllocationManager 可以设置动态分配最小 Executor 数量、动态分配最大 Executor 数量、每个 Executor 可以运行的 Task 数量等配置信息，并对配置信息进行校验。start 方法将 ExecutorAllocationListener 加入 listenerBus 中，ExecutorAllocationListener 通过监听 listenerBus 里的事件，动态添加、删除 Executor。并且通过 Thread 不断添加 Executor，遍历 Executor，将超时的 Executor 杀掉并移除。ExecutorAllocationListener 的实现与其他 SparkListener 类似，不再赘述。ExecutorAllocationManager 的关键代码见代码清单 3-47。

代码清单3-47 ExecutorAllocationManager的关键代码

```
private val intervalMillis: Long = 100
private var clock: Clock = new RealClock
private val listener = new ExecutorAllocationListener
def start(): Unit = {
    listenerBus.addListener(listener)
    startPolling()
}

private def startPolling(): Unit = {
    val t = new Thread {
        override def run(): Unit = {
            while (true) {
                try {
                    schedule()
                } catch {
                    case e: Exception => logError("Exception in dynamic executor
                        allocation thread!", e)
                }
                Thread.sleep(intervalMillis)
```

```
            }
        }
    }
    t.setName("spark-dynamic-executor-allocation")
    t.setDaemon(true)
    t.start()
}
```

注意　根据 3.4.1 节的内容，我们知道 listenerBus 内置了线程 listenerThread，此线程不断从 eventQueue 中拉出事件对象，调用监听器的监听方法。要启动此线程，需要调用 listenerBus 的 start 方法，代码如下。

```
listenerBus.start()
```

3.11　ContextCleaner 的创建与启动

ContextCleaner 用于清理那些超出应用范围的 RDD、ShuffleDependency 和 Broadcast 对象。由于配置属性 spark.cleaner.referenceTracking 默认是 true，所以会构造并启动 ContextCleaner，代码如下。

```
private[spark] val cleaner: Option[ContextCleaner] = {
    if (conf.getBoolean("spark.cleaner.referenceTracking", true)) {
        Some(new ContextCleaner(this))
    } else {
        None
    }
}
cleaner.foreach(_.start())
```

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。

ContextCleaner 的工作原理和 listenerBus 一样，也采用监听器模式，由线程来处理，此线程实际只是调用 keepCleaning 方法。keepCleaning 的实现见代码清单 3-48。

代码清单3-48　keep Cleaning的实现

```
private def keepCleaning(): Unit = Utils.logUncaughtExceptions {
    while (!stopped) {
        try {
            val reference = Option(referenceQueue.remove(ContextCleaner.REF_
                 QUEUE_POLL_TIMEOUT))
                .map(_.asInstanceOf[CleanupTaskWeakReference])
            // Synchronize here to avoid being interrupted on stop()
```

```
                synchronized {
                    reference.map(_.task).foreach { task =>
                    logDebug("Got cleaning task " + task)
                    referenceBuffer -= reference.get
                    task match {
                        case CleanRDD(rddId) =>
                            doCleanupRDD(rddId, blocking = blockOnCleanupTasks)
                        case CleanShuffle(shuffleId) =>
                            doCleanupShuffle(shuffleId, blocking = blockOnShuffleCleanupTasks)
                        case CleanBroadcast(broadcastId) =>
                            doCleanupBroadcast(broadcastId, blocking = blockOnCleanupTasks)
                        }
                    }
                }
            } catch {
                case ie: InterruptedException if stopped => // ignore
                case e: Exception => logError("Error in cleaning thread", e)
            }
        }
    }
```

3.12 Spark 环境更新

在 SparkContext 的初始化过程中，可能对其环境造成影响，所以需要更新环境，代码如下。

```
postEnvironmentUpdate()
postApplicationStart()
```

SparkContext 初始化过程中，如果设置了 spark.jars 属性，spark.jars 指定的 jar 包将由 addJar 方法加入 httpFileServer 的 jarDir 变量指定的路径下。spark.files 指定的文件将由 addFile 方法加入 httpFileServer 的 fileDir 变量指定的路径下。见代码清单 3-49。

代码清单3-49　依赖文件处理

```
val jars: Seq[String] =
    conf.getOption("spark.jars").map(_.split(",")).map(_.filter(_.size != 0)).
        toSeq.flatten

val files: Seq[String] =
    conf.getOption("spark.files").map(_.split(",")).map(_.filter(_.size != 0)).
        toSeq.flatten

// Add each JAR given through the constructor
    if (jars != null) {
        jars.foreach(addJar)
    }

    if (files != null) {
        files.foreach(addFile)
}
```

httpFileServer 的 addFile 和 addJar 方法，见代码清单 3-50。

代码清单3-50　HttpFileServer提供对依赖文件的访问

```
def addFile(file: File) : String = {
    addFileToDir(file, fileDir)
    serverUri + "/files/" + file.getName
}

def addJar(file: File) : String = {
    addFileToDir(file, jarDir)
    serverUri + "/jars/" + file.getName
}

def addFileToDir(file: File, dir: File) : String = {
    if (file.isDirectory) {
        throw new IllegalArgumentException(s"$file cannot be a directory.")
    }
    Files.copy(file, new File(dir, file.getName))
    dir + "/" + file.getName
}
```

postEnvironmentUpdate 的实现见代码清单 3-51，其处理步骤如下：

1）通过调用 SparkEnv 的方法 environmentDetails 最终影响环境的 JVM 参数、Spark 属性、系统属性、classPath 等，参见代码清单 3-52。

2）生成事件 SparkListenerEnvironmentUpdate，并 post 到 listenerBus，此事件被 EnvironmentListener 监听，最终影响 EnvironmentPage 页面中的输出内容。

代码清单3-51　postEnvironmentUpdate的实现

```
private def postEnvironmentUpdate() {
    if (taskScheduler != null) {
        val schedulingMode = getSchedulingMode.toString
        val addedJarPaths = addedJars.keys.toSeq
        val addedFilePaths = addedFiles.keys.toSeq
        val environmentDetails =
            SparkEnv.environmentDetails(conf, schedulingMode, addedJarPaths,
                addedFilePaths)
        val environmentUpdate = SparkListenerEnvironmentUpdate(environmentDetails)
        listenerBus.post(environmentUpdate)
    }
}
```

代码清单3-52　environmentDetails的实现

```
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

// System properties that are not java classpaths
val systemProperties = Utils.getSystemProperties.toSeq
val otherProperties = systemProperties.filter { case (k, _) =>
    k != "java.class.path" && !k.startsWith("spark.")
}.sorted

// Class paths including all added jars and files
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)
}
```

postApplicationStart 方法很简单，只是向 listenerBus 发送了 SparkListenerApplicationStart 事件，代码如下。

```
listenerBus.post(SparkListenerApplicationStart(appName, Some(applicationId),
    startTime, sparkUser))
```

3.13 创建 DAGSchedulerSource 和 BlockManagerSource

在创建 DAGSchedulerSource、BlockManagerSource 之前首先调用 taskScheduler 的 postStartHook 方法，其目的是为了等待 backend 就绪，见代码清单 3-53。postStartHook 的实现见代码清单 3-54。

创建 DAGSchedulerSource 和 BlockManagerSource 的过程类似于 ExecutorSource，只不过 DAGSchedulerSource 测量的信息是 stage. failedStages、stage. runningStages、stage. waiting-Stages、stage. allJobs、stage. activeJobs，BlockManagerSource 测量的信息是 memory. maxMem_MB、memory. remainingMem_MB、memory. memUsed_MB、memory. diskSpace-Used_MB。

代码清单3-53　创建DAGSchedulerSource和BlockManagerSource

```
    taskScheduler.postStartHook()

    private val dagSchedulerSource = new DAGSchedulerSource(this.dagScheduler)
    private val blockManagerSource = new BlockManagerSource(SparkEnv.get.
        blockManager)

private def initDriverMetrics() {
    SparkEnv.get.metricsSystem.registerSource(dagSchedulerSource)
    SparkEnv.get.metricsSystem.registerSource(blockManagerSource)
}

initDriverMetrics()
```

代码清单3-54　postStartHook的实现

```
override def postStartHook() {
        waitBackendReady()
    }

private def waitBackendReady(): Unit = {
    if (backend.isReady) {
        return
    }
    while (!backend.isReady) {
        synchronized {
            this.wait(100)
        }
    }
}
```

3.14　将 SparkContext 标记为激活

SparkContext 初始化的最后将当前 SparkContext 的状态从 contextBeingConstructed（正在构建中）改为 activeContext（已激活），代码如下。

```
SparkContext.setActiveContext(this, allowMultipleContexts)
```

setActiveContext 方法的实现如下。

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

3.15 小结

回顾本章，Scala 与 Akka 的基于 Actor 的并发编程模型给人的印象深刻。listenerBus 对于监听器模式的经典应用看来并不复杂，希望读者朋友能应用到自己的产品开发中去。此外，使用 Netty 所提供的异步网络框架构建的 Block 传输服务，基于 Jetty 构建的内嵌 web 服务（HTTP 文件服务器和 SparkUI），基于 codahale 提供的第三方测量仓库创建的测量系统，Executor 中的心跳实现等内容，都值得借鉴。

HZ BOOKS
华章图书